An Introduction to

VHDL

Steven J. Merrifield
Department of Electronic Engineering
La Trobe University

Contents

Chapter 1

Introduction

1.1 What is VHDL?

VHDL is an acronym which stands for VHSIC Hardware Description Language. VHSIC is yet another acronym standing for Very High Speed Integrated Circuit.

Thus, VHDL is a very high speed integrated circuit hardware description language. It is a programming language designed to describe the operation of digital hardware, and as such, combines the following features:

1.1.1 A Simulation Modeling Language

VHDL has many features appropriate for describing the behavior of electronic components ranging from simple logic gates to complete microprocessors and custom IC's. Features of VHDL allow electrical aspects of a circuit such as the rise and fall time of a signal and the propagation delay through a gate to be precisely described. The resulting VHDL model can then be used as a building block in a larger circuit.

1.1.2 A Design Entry Language

Much like Pascal or C, VHDL includes features useful for structured design techniques, and offers a rich set of control and data representation features. Unlike other programming languages, VHDL allows *concurrent* events to be described. This is important since the hardware described by VHDL is inherently concurrent in it's operation.

1.1.3 A Verification Language

One of the most important applications of VHDL is to capture the performance specification for a circuit in the form of a *test bench*. A test bench is a VHDL description of inputs to the system, and the expected outputs that verify the behavior of a circuit over time. Test benches should be an integral part of any VHDL project, and should be created in parallel with the hardware description source.

1.1.4 A Netlist Language

While VHDL is a powerful language with which to enter new designs at a high level, it is also useful as a low-level form of communication between different tools in a computer-based design environment. VHDL's structural language features allow it to be effectively used as a netlist language, replacing other netlist languages such as EDIF.

1.1.5 A Standard Language

One of the most compelling reasons to become familiar with VHDL is it's adoption as a standard in the electronic design community. Using a standard language such as VHDL virtually guarantees that you will not have to throw away and redesign your hardware when a new version of the software tools is released. Using VHDL also allows an exchange of ideas and concepts with other developers without disclosing the entire project.

1.2 VHDL as an IEEE Standard

VHDL was developed in the early 1980's as a spin-off from a high-speed integrated circuit research project funded by the United States Department of Defense. IBM, Texas Instruments and Intermetrics designed and implemented a new language-based design description method, which was released to the public for the first time in 1985. In 1987 the IEEE standardized the language, leading to the standard IEEE 1076-1987 which is the basis for virtually every simulation and synthesis product sold today. An enhanced and updated version was released five years later, known as IEEE 1076-1993.

1.2.1 Standard 1164

Although IEEE standard 1076 defines the complete VHDL language, there are aspects which make it difficult to write code which is completely portable

between different vendors. To improve this, the IEEE released another standard, numbered 1164, which allows commonly used declarations to be collected into an external library in much the same that C uses platform specific header files. Virtually all VHDL code you write will include the 1164 library as shown:

```
library ieee;
use ieee.std_logic_1164.all;
```

1.3 Simulation vs Synthesis

VHDL can essentially be thought of as two separate languages. One designed specifically for implementing hardware in an integrated circuit, and the other as a general purpose hardware language. VHDL code written with a hardware design in mind is known as *synthesizable* VHDL, since the code will be used to synthesize actual hardware. Synthesizable VHDL is a smaller subset of the complete VHDL specification, and uses a restricted set of commands. The entire VHDL command set can be used to *simulate* the hardware design in the form of test benches, since the code will be executed on a PC or workstation, and is not designed to be programmed into an IC. These lecture notes will concentrate on synthesizable VHDL since you will be required to design and implement a number of physical circuits during this course.

1.4 Design Structure

A VHDL design is usually broken into several small *design entities*, each having an entity and architecture block. The entity declaration describes the circuit as it appears from the top-most physical layer, detailing the circuit's input and output interfaces. An entity can be thought of as a block symbol on a schematic diagram. The architecture declaration describes the actual function, or contents, of the entity to which it is bound. Several entity/architecture pairs can be grouped together to form a *package*. The overall arrangement of these modules is shown in Figure 1.1.

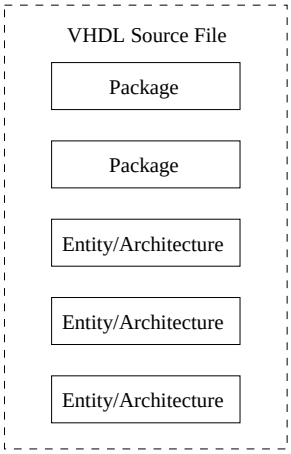

Figure 1.1: A modular VHDL design

Chapter 2

Objects and Data Types

There are three basic types of objects in VHDL - signals, variables and constants. Each object has a specific data type, and a unique set of possible values. The values that an object can take will depend on the definition of the type used for that object. For example, an object of type `bit` has only two possible values, `'0'` and `'1'`, while an object of type `real` has many possible values.

2.1 Signals, Variables and Constants

2.1.1 Signals

Signals are objects used to connect concurrent elements (such as components, processes, and concurrent assignments). Signals can be declared globally in an external package or locally within an architecture, block, or other declarative region.

As a minimum, a signal declaration must include the name of the signal and it's type. If more than one signal of the same type is required, multiple signals can be specified in a single declaration:

```
architecture example of my_circuit is
signal A   : std_logic;
signal X,Y : std_logic_vector(3 downto 0);
begin
  ...
  ...
end example;
```

In this case `A` is only visible within the example design unit, and will not

be visible from other design units. To make a signal accessible globally, the declaration needs to be made in an external package:

```
package my_package is
  signal A : std_logic;
end my_package;
...
use work.my_package.A;
...
```

The signal declaration can also be used to assign an initial value to a signal as shown:

```
signal Init : std_logic_vector(7 downto 0) := "01010011";
```

Signals are used in two primary ways. Firstly, signals are used to carry information between different functional parts of your design, such as between two components. And secondly, signals can be used within logic expressions and are assigned values directly.

2.1.2 Variables

Variables are objects used to store intermediate values between sequential VHDL statements. Variables are only allowed in processes, procedures and functions, and they are always local to those areas. Variables in VHDL are much like variables in a conventional software programming language. They immediately take on and store the values assigned to them (this is not true of signals), and they can be used to simplify a complex calculation or sequence of logical operations.

2.1.3 Constants

Constants are objects that are assigned a value once, when declared, and do not change their value during operation. Constants are useful for creating more readable design descriptions, and make it easier to change the design at a later date. Some examples of constant declaration follow:

```
architecture example of my_circuit is
  constant MaxCount : integer := 255;
  constant SRAM : std_logic_vector(7 downto 0) := "00001111";
  constant error : boolean := True;
begin
```

```
  ...
  process(...)
    constant StartCnt : integer := 57;
    ...
  end process;
  ...
end example;
```

Constant declarations can be located in any declaration area in your design description. They can also be placed in an external package in a similar way to signals.

2.2 Literals

Explicit data values that are assigned to objects or used within expressions are called *literals*. Literals represent specific values, but do not always have an explicit type. For example, the literal `'1'` could represent either a `std_logic` type or a `character`.

Character Literals

Character literals are single character ASCII values enclosed in single quotes, such as the values `'1', 'K'` and `'#'`.

String Literals

String literals are collections of one or more ASCII characters enclosed in double quotes, and they may be assigned to appropriately sized arrays of single-character data types such as `std_logic_vector` or to objects of the built-in type `string`.

Bit String Literals

Bit string literals are special forms of string literals that are used to represent binary, octal or hexadecimal numeric data values. Binary numbers must be preceded by the character 'B', and may only contain the characters '0' and '1'. An octal number is preceded by an 'O' and must only contain the characters '0' through '7'. Hexadecimal characters must be preceded by the character 'X' and may contain characters '0' through '9' and 'A' through 'F', with 'a' through 'f' also allowed. An underscore character may be included to improve readability. Some examples follow:

```
B"0111_1101"   -- decimal 253
O"654"         -- decimal 428
O"146_231"     -- decimal 52,377
X"C300"        -- decimal 49,920
```

Numeric Literals

There are two basic forms of numeric literals - integers and reals. These are used in exactly the same way as any other programming language, and may include an underscore to improve readability. For example, `1_276_801` is the integer value 1276801. Note that commas are not permitted in numeric literals. Also note, that strict type checking is used, so it is not possible to assign say `9` to an object of type `real`. You must use `9.0` instead.

Based Literals

Based literals are another form of integer or real values, but they are written in non-decimal form. To specify a based literal, you precede the literal with a base specification and enclose the non-decimal value inside a pair of #'s, as shown:

```
2#10010001#    -- integer value 145
16#FFCC#       -- integer value 65,484
```

Physical Literals

Physical literals are special types of literals used to represent physical quantities such as time, voltage, current, distance etc. Physical literals include both a numeric part (expressed as an integer) and a unit specification. Some examples are shown below:

```
30 ns
500 mA
850 kW
```

2.3 Types

There are four basic classes of data types -

- **Scalar types** represent a single numeric value, and may be either an integer, real, physical or enumerated.

- **Composite types** represent a collection of values. These may be either an array containing elements of the same type, or records containing elements of different types.
- **Access types** provide references to objects in a similar way that pointers function in C or Pascal, and,
- **File types** reference objects such as disk files that contain a sequence of values.

Each type has a defined set of values. For example, an integer has a defined range of at least -2147483647 to +2147483647. In most cases however, you will only be interested in a subset of values, and VHDL allows you to define your own type as shown:

```
subtype SHORT integer range 0 to 255;
```

You can also specify a constraint on an existing range when declaring an object of a given type:

```
signal ShortInt : integer range 0 to 255;
```

2.4 Operators

Tables 2.1 and 2.2 summarize the most commonly used operations in VHDL.

2.5 Attributes

Attributes are a feature of VHDL that allow you to extract additional information about an object that may not be directly related to the value that the object carries. There are five fundamental kinds of attributes, categorized by the results that are returned when they are used. The possible results returned from these attributes are: a value, a function, a signal, a type or a range. Predefined attributes are always applied to a prefix (such as a signal or variable name), as shown in the following example:

```
wait until clk = '1' and clk'event and clk'last_value = '0';
```

In this example, `clk` could be a signal, and the attributes `'event` and `'last_value` have been applied to it. Other examples of attributes include: `'left` which returns the left most element index of a given type or subtype. `'right` which returns the right most element. One of the most commonly used attributes is `'length` which returns the number of elements in an array:

and	And
or	Or
nand	Not And
nor	Not Or
xor	Exclusive Or
xnor	Exclusive Not Or
=	Equality
/ =	Inequality
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
+	Addition
−	Subtraction
&	Concatenation
∗	Multiplication
/	Division
mod	Modulus
rem	Remainder
∗∗	Exponentiation
abs	Absolute value
not	Logical negation

Table 2.1: Relational and Numeric Operators

```
type bit_array is array(0 to 31) of bit;
variable len : integer := bit_array'length;
```

In this case, `len` takes on the value 32.

A commonly used attribute which you will encounter during this course is `'event` which returns a true value if the signal had an event (changed its value) during the current cycle. Note that the example presented above which checks for a change in the `clk` signal can also be represented using `if rising_edge(clk)`, but such code is not synthesizable, although it is perfectly legitimate VHDL code.

2.6 Type Conversion

VHDL is a strongly typed language, meaning that you cannot simply assign a literal value or object of one type to an object of another type. To allow

sll	Shift left logical
srl	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic
rol	Rotate left logical
ror	Rotate right logical

Table 2.2: Shift Operators

the transfer of data between objects of different types, VHDL includes *type conversion* features for types that are closely related.

2.6.1 Explicit Type Conversions

The simplest type conversions are explicit type conversions, which are only allowed between closely related types. Two types are said to be closely related when they are either abstract numeric types, or if they are array types of the same dimensions and share the same types for all elements in the array.

2.6.2 Type Conversion Function

To convert data from one type to an unrelated type, you must make use of a type conversion function. A type conversion function is one that accepts one argument of a specified type, and returns the equivalent value in another type. This is often done to convert between an `integer` and `std_logic_vector` data types. Such conversion functions are generally provided in an external library, so it necessary to include that library in the source code header:

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

2.6.3 Ambiguous Literal Types

Functions and procedures are uniquely defined not only by their names, but also by the types of their arguments. If you were to write an assignment statement such as :

```
architecture ambig of my_circuit is
  signal int : integer;
begin
```

```
  int <= to_integer("00100011");
  ...
end ambig;
```

the compiler would produce an error message, because it would be unable to determine which of the two functions is appropriate - the literal "00100011" could be either a `string` or `std_logic_vector` data type. To remove this ambiguity, you can do one of two things. Either introduce an intermediate constant, signal, or variable like:

```
architecture unambig of my_circuit is
  constant const : bit_vector := "00100011";
  signal int : integer;
begin
  int <= to_integer(const);
  ...
end unambig;
```

or, the second alternative is to introduce a type mark to qualify the assignment as shown:

```
architecture unambig of my_circuit is
  signal int : integer;
begin
  int <= to_integer(std_logic_vector"00100011");
  ..
end unambig;
```

Chapter 3

Levels of Abstraction

3.1 Introduction

VHDL supports many possible styles of design description. These styles differ primarily in how closely they relate to the underlying hardware. Figure 3.1 maps the various points in a top-down design process to the three general levels of abstraction - structural, dataflow and behavioral. Each of these levels will be covered in more detail in the following chapters. As an example of these three levels of abstraction, it is possible to describe a complex controller in a number of ways. At the lowest level of abstraction (structural), we could use VHDL's hierarchy features to connect a sequence of predefined logic gates and flip-flops to form the complete circuit. To describe the same circuit at a dataflow level we might describe the combinational logic portion of the controller using higher-level Boolean logic functions and then feed the output of that logic into a set of registers that match the registers available in the target technology. At the behavioral level of abstraction, the target technology may be ignored completely, and the controller described in terms of it's response over time to various input stimuli.

3.2 Structural Description

The structural description method is used to describe a circuit in terms of its components. Structure can be used to create a very low-level description of a circuit, such as a transistor-level description, or a very high-level description like a block diagram.

In a gate-level description of a circuit, for example, components such as basic logic gates and flip-flops might be connected in some logical structure to create the circuit. This is often called a *netlist*. For a higher-level circuit,

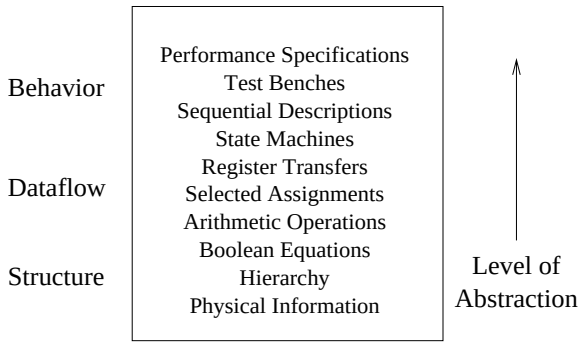

Figure 3.1: Levels of Abstraction

one in which the components being connected are larger functional blocks, a structural approach may be used to segment the design description into manageable parts.

Structural level VHDL features, such as components and configurations, are very useful for managing complexity. The use of components can dramatically improve your ability to re-use elements of your designs, and they make it possible to work using a top-down design approach.

Consider the RS flip-flop shown in Figure 3.2, and the corresponding entity declaration which describes the device. Note that this device requires active low inputs.

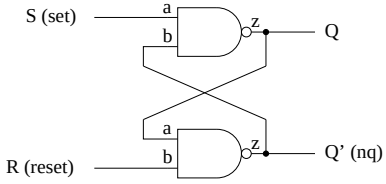

Figure 3.2: RS Flip-Flop with normally high (active low) inputs

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

The first line defines a new entity called `flipflop`. The last line concludes the definition, whilst the lines in between describe the interface, known as the *port clause*. The port clause contains a list of *interface declarations*, which define one or more signals as either inputs or outputs. Each interface declaration contains a list of names, a mode and a type. In this example,

s and r are assigned as inputs, and of type `std_logic`, which is normally a binary 0 or 1, but could also be a don't care or high impedance state. Note that a semi-colon is used as a statement de-limiter, except on the last line of a port clause.

Once an entity has been defined, an associated architecture must be described. Using a structural approach, the code to do this may look like the following:

```
architecture structure of flipflop is
  component nand_gate
    port(a, b : in  std_logic;
         z    : out std_logic);
  end component;
begin
  top: nand_gate
    port map (s, nq, q);
  bot: nand_gate
    port map (r, q, nq);
end structure;
```

The *component declaration* appears as the first part of the description and describes the interface of the `nand_gate` entity to be used as a component. It is assumed that the mathematical operation of this entity has already been defined. The second part of the architecture defines two *component instances*. The `port map` clause specifies which signals to connect to the interface of the component in the same order as they appear in the component declaration. The interface is specified as a, b, z and so when applied to the top gate, the instance connects to s, nq and q. A similar approach for the bottom gate maps a, b and z to r, q and nq.

3.3 Dataflow Description

In the dataflow level of abstraction, a circuit is described in terms of how data moves through the system. At the heart of most digital systems are registers, so in the dataflow level of abstraction, you describe how information is passed between registers in the circuit. The dataflow level is often called the register transfer level, or RTL. This level of abstraction is an intermediate level that allows the drudgery of combinational logic to be simplified, while the important parts of the circuit, the registers, are more completely specified.

There are some drawbacks to using a dataflow method of design in VHDL. First, there are no built-in registers in VHDL - the language was designed to be general-purpose, and the emphasis was placed on it's behavioral aspects. If you are going to write VHDL at the dataflow level of abstraction, you must first create behavioral descriptions of the register elements you will be using in your design. These elements must be provided in the form of components (using VHDL's hierarchy features) or in the form of subprograms (functions or procedures).

Using a data flow description, circuits are described by indicating how the inputs and outputs of in-built primitives are connected together. Several primitives are pre-defined in VHDL, and include entities such as AND, NAND, OR gates etc. Using the same example as before, shown in Figure 3.2, the data flow description may be coded as:

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

architecture dataflow of flipflop is
begin
  q  <= s nand nq;
  nq <= r nand q;
end dataflow;
```

Comparing the first two methods, it is clear that the data flow method of description is shorter, and more easily understood than the structural descriptive method. This is not always the case though. In this example, a pre-defined operation was used, thus simplifying the code, if another example was analyzed, a structural approach may be a better choice.

When a design has been described textually, it can then be simulated to verify it's operation. The scheme used to model a VHDL design is known as *discrete event time simulation*, which only updates signals when certain events occur, and these events occur only at discrete instances in time. An event on a particular signal is said to have occurred when the value of that signal changes. If data flows from signal A to signal B, and an event occurs on signal A, then it is necessary to determine the possibly new value of signal B. Since one event causes another, simulation proceeds in rounds. The simulator maintains a list of events that require processing, and during each round, all events in the list are processed, with any new events occurring being placed in a separate list for processing during a later round. This scheme is known as scheduling.

Taking the data flow architecture of the `flipflop` entity, the internal operation was essentially defined using the following code:

```
q  <= s nand nq;
nq <= r nand q;
```

It is seen that q depends on the values of s and nq, so if either of these values change, then a new value of q will be evaluated. If the result is different from the current value of the signal, an event will be scheduled.

Assume that at a particular moment during the simulation, the values of each signal are $s = 1, r = 1, q = 0$ and $nq = 1$. Now suppose the value of the signal s changes to 0. Since q depends on s, we must re-evaluate the expression `q <= s nand nq`, which now equals 1. Since the value of q must be changed to 1, a new event will be scheduled on the signal q. During the next round the event scheduled for q is processed and q's value is updated to be 1. Also, since nq depends on q, the expression `nq <= r nand q` must be re-evaluated. The result of this expression is 0, so an event is scheduled to update the value of nq. During the next round, when the event on nq is processed, the expression will be 1, and no new events will be scheduled because q is already 1.

Now suppose an external event causes s to return to 1. Since q depends on s, `q <= s nand nq` is evaluated again. The result of this is 1, and since q is already 1, no new events are scheduled. Thus it can be seen that this model correctly describes the operation of the flip-flop shown in Figure 3.2. When the signal s became 0, the output was set, and when s returned high, the output was unchanged, thus confirming the circuits operation as a latch. These discrete event, time simulation results are summarized in Table 3.1.

Table 3.1: Results of RS Flip-Flop Simulation

Round	s	r	q	nq	Comments
Start	1	1	0	1	
1	0	1	0	1	'1' is scheduled on q
2	0	1	1	1	'0' is scheduled on nq
3	0	1	1	0	No new events are scheduled
4	1	1	1	0	No new events are scheduled

The previous explanation of circuit simulation only dealt with the functional operation, and did not consider any timing information. All designs which are intended to be constructed in hardware will experience internal

timing delays, and this can be modeled using VHDL in two ways. The *inertial delay model* and the *transport delay model* will be discussed.

The inertial delay model is specified by the inclusion of an *after* clause in the signal assignment. This models the internal timing delays within the entity. For example, to include a delay in the previous flip-flop example, the code needs to be modified as shown:

```
q  <= s nand nq after 2ns;
nq <= r nand q  after 2ns;
```

The new operation of the circuit is shown in Figure 3.3, where it is clear that the q output does not change until 2ns after the s input changes state. Likewise, nq does not change until 2ns after q changes. In order to do this, the simulator must maintain track of the current time, and process events accordingly. When there are no events requiring processing at the current time, the time is updated to the time of the next earliest event and all events scheduled for that time will be processed.

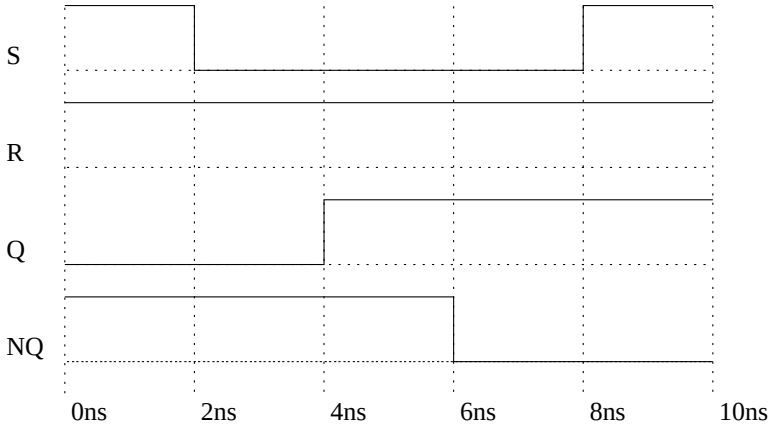

Figure 3.3: RS Flip-Flop timing diagram using the inertial delay model

Although useful for modeling inherent delays, the inertial model does have a rather obvious downfall. When faced with a input pulse shorter than the internal delay, the inertial model will not propagate that pulse through to the output. This is shown in Figure 3.4, where the value of q does not change because the pulse length of s was not long enough. It is said that the change in s did not gain enough inertia. This effect is quite useful for absorbing noise spikes or glitches which could otherwise cause havoc in the system.

When faced with the need to propagate short-width pulses, the transport delay model can be used. This is useful for bus systems where a time delay

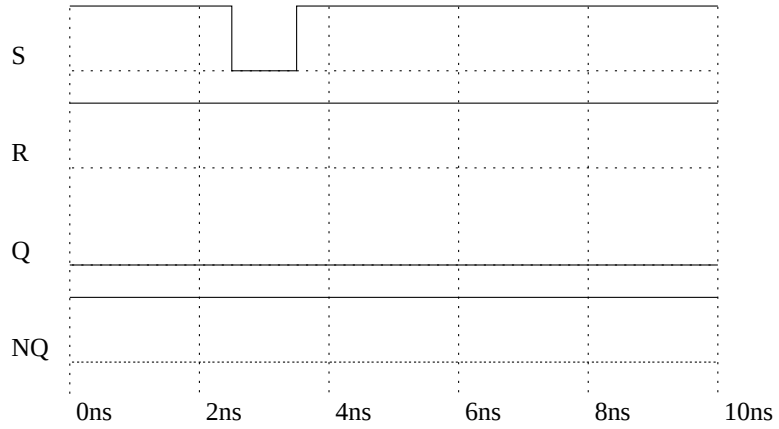

Figure 3.4: RS Flip-Flop timing diagram with a narrow input pulse

is required, and nothing traveling over that bus should be absorbed. This model simply delays the change in output by the time specified in the after clause. Typical application code is shown below, and a timing diagram is featured in Figure 3.5.

```
q  <= transport s nand nq after 2ns;
nq <= transport r nand q  after 2ns;
```

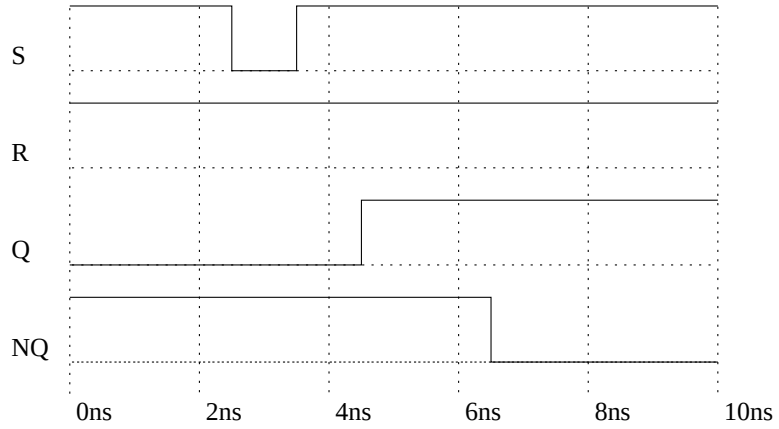

Figure 3.5: RS Flip-Flop timing diagram using the transport delay model

Up until now, all the signals have been of the type `std_logic`. VHDL provides the facility to use several other types, one of which is the type known as a `std_logic_vector`. This type is comparable to the array structure found in the Pascal or C programming languages, and is used to perform operations

on a collection of bits. A typical use may be to describe a bus, and as an example, an 8 line de-multiplexer is shown below.

```
entity decode is
port(s : in  std_logic_vector(2 downto 0);   -- 3 select inputs
     z : out std_logic_vector(7 downto 0)    -- 8 data outputs
);
end decode;

architecture rtl of decode is
  signal temp : std_logic_vector(7 downto 0);
begin
  temp(0) <= not s(2) and not s(1) and not s(0);
  temp(1) <= not s(2) and not s(1) and s(0);
  temp(2) <= not s(2) and s(1) and not s(0);
  temp(3) <= not s(2) and s(1) and s(0);
  temp(4) <= s(2) and not s(1) and not s(0);
  temp(5) <= s(2) and not s(1) and s(0);
  temp(6) <= s(2) and s(1) and not s(0);
  temp(7) <= s(2) and s(1) and s(0);
  z <= temp;
end rtl;
```

3.4 Behavioral Description

The highest level of abstraction supported in VHDL is called the behavioral level. When creating a behavioral description of a circuit, you will describe the circuit operation in terms of it's operation over time. The concept of time is the critical distinction between behavioral descriptions of circuits, and lower-level descriptions.

In a behavioral description, the concept of time may be expressed precisely, with actual delays between related events (such as the propagation delay within a gate), or it may simply be an ordering of operations that are expressed sequentially.

Both the structural and data flow methods of description deal with how the design is implemented - it is usual to start with a schematic diagram, and translate that into VHDL. The behavioral method though, differs in that a black-box approach to design is taken. It models what happens on the inputs and outputs of the design, but the process occurring inside the box is irrelevant. This is an advantage when dealing with complicated components

that would be tedious to model using either of the first two approaches. This might be the case for example, when interfacing a custom circuit to a microprocessor system. The internal operation of the microprocessor is not important, but it's external behavior can be easily modeled.

Behavioral descriptions are modeled using the *process* statement, following a similar line to the architectural definitions of the structural and data flow methods. In contrast though, a list of signals, called the *sensitivity list* is passed as a parameter to the process statement. The signal sensitivity list is used to specify which signals should cause the process to be re-evaluated. Whenever a change occurs to a signal in the sensitivity list, the process is executed. As an example, consider the following trivial segment of code:

```
compute_and: process(a,b)
begin
  z <= a and b;
end process;
```

Here, the sensitive signals are a and b, so when an event occurs on either of these signals, z will be re-evaluated. Unlike signal assignments which occur outside the process statement, the signal assignments within a process statement will only be updated when an event occurs on a signal in the sensitivity list. Therefore, it is essential that the appropriate signals are correctly specified. Following the execution of the last statement, the process is finished, and is said to be *suspended*. A process *resumes* when an event occurs on a signal in the sensitivity list and it commences execution. Note that the process name `compute_and` is optional, and is usually only included in large designs to make debugging and coding easier.

As is common in high level languages, a programmer has the ability to declare variables. The syntax and use of variables follows closely that of Pascal or C, and so will not be dwelt on here. To illustrate this point, consider the code shown below:

```
edges: process(in)
  variable count : integer := -1;
begin
  count := count + 1;
end process;
```

Once the simulation commences, the variable `count` will be incremented each time the signal `in` changes, since it has been declared in the sensitivity list. If `in` was defined to be a `std_logic` signal, then this process will count the number of rising and falling edges that occur on the signal `in`. This code can be modified to use a conditional `if` statement as shown below.

```
rising: process(in)
  variable count : integer := -1;
begin
  if in='1' and in'last_value='0' then
    count := count + 1;
  end if;
end process;
```

Note the use of the `last_value` *attribute* which is used to determine the last value a signal had. VHDL defines a number of attributes, which are specified using the name of the signal, followed by a ' (called a tick), and the attribute name. Hence, in this example, the conditional statement is only true if the current value of `in` is '1' and it's previous value was '0'. Now since this statement will only be executed when an event occurs on `in`, this condition only becomes true when a rising edge occurs on the signal `in`. Thus, we have modified our original double-edge counter to one which counts rising edges only.

A common use of conditional statements is in repetitive loops like that shown below. Here, a `for` loop is used to calculate the even parity of an 8-bit vector. Note that the temporary variable used as a loop counter does not need to be pre-defined. This is in contrast to many other high level languages which require all variables to be declared.

```
...
signal in : std_logic_vector(7 downto 0);
...
process(in)
  variable parity : std_logic;
begin
  parity := '0';
  for i in 7 downto 0 loop
    parity := parity xor in(i);
  end loop;
end process;
```

An important distinction should be made between the operation of a variable assignment and a signal assignment in the process statement. A signal assignment merely schedules an event to occur and does not have an immediate effect. When a process is resumed, it executes from top to bottom sequentially and any pending events are not processed until after the process is suspended. As an example, consider the following process where two events are scheduled on signals a and c.

```
...
signal a,b,c : std_logic;
...
process (b)
begin
  a <= b;
  c <= not a;
end process;
```

If the signal b changes, an event will be scheduled on a to make it's value the same as b. In addition, an event is also scheduled on c to make it's value the opposite of a. It is important to realize here, that the value of c will not be the opposite of b, because when the second statement is executed, the event on a has not been processed yet. This has been mentioned because it is not necessarily the intuitive behavior, and because the operation of variables differs. For example, in the following code segment, the value of the variable c would be the opposite of b, because the value of a is changed immediately.

```
...
process(b)
variable a,c : std_logic;
begin
  a := b;
  c := not a;
end process;
```

Chapter 4

Concurrent and Sequential Statements

4.1 Introduction

Understanding the fundamental difference between concurrent and sequential statements in VHDL is important to making effective use of the language. Figure 4.1 illustrates the basic difference between these two types of statements.

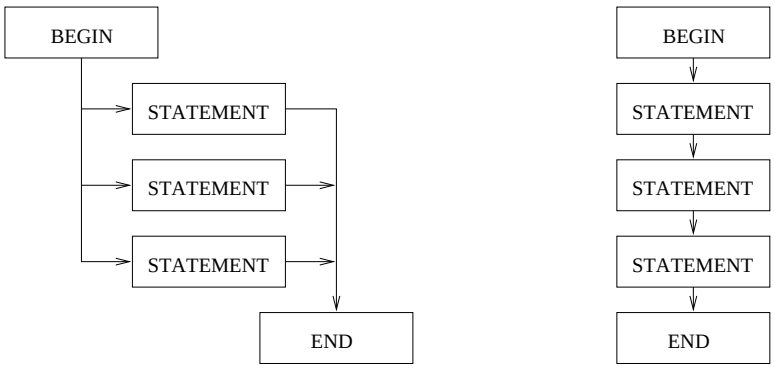

Figure 4.1: Concurrent and Sequential Statements

The left most diagram illustrates how concurrent statements are executed in VHDL. Concurrent statements are those which appear between the `begin` and `end` statements of a VHDL architecture clause. All statements in the concurrent area are executed at the same time, and there is no significance to the order in which the statements are entered. A *process* is considered a single concurrent statement, and each process in an architecture is executed at the same time.

The diagram on the right shows how sequential VHDL statements are executed. Sequential statements are executed one after the other in the order in which they appear between the `begin` and `end` statements of a VHDL process, procedure or function clause. Although a process is a concurrent statement, each statement within the process is executed sequentially.

4.2 Concurrent Statements

The most common concurrent statements are signal assignments such as those shown in the following example. In general, concurrent signal assignments are used to describe either combinational logic, or to describe the connections between lower level components. Since there is no order dependency to statements in the concurrent area, the following two architecture declarations are functionally equivalent to each other.

```
architecture arch1 of my_circuit is
signal A,B,C  : std_logic_vector(3 downto 0);
constant init : std_logic_vector(3 downto 0) := "0010";
begin
  A <= B and C;
  B <= init;
  C <= A and B;
end arch1;

architecture arch2 of my_circuit is
signal A,B,C  : std_logic_vector(3 downto 0);
constant init : std_logic_vector(3 downto 0) := "0010";
begin
  C <= A and B;
  A <= B and C;
  B <= init;
end arch2;
```

4.2.1 Conditional Signal Assignments

A conditional signal assignment is a special form of signal assignment, similar to the if-then-else statements found in other software programming languages, that allows you to describe a sequence of related conditions under which one or more signals are assigned values. The following example (a simple multiplexer) demonstrates the basic form of a conditional assignment:

```
entity my_mux is
port ( Sel      : in  std_logic_vector(1 downto 0);
       A,B,C,D  : in  std_logic_vector(0 to 3);
       Y        : out std_logic_vector(0 to 3)
     );
end my_mux;

architecture mux_arch of my_mux is
begin
  y <= A when Sel = "00" else
       B when Sel = "01" else
       C when Sel = "10" else
       D when others;
end mux_arch;
```

Note: *It is very important that all conditions in a conditional assignment are covered, as unwanted latches can easily be generated by synthesis tools for those conditions which are not covered. In the previous example it would be incorrect to replace* `D when others` *with* `D when Sel = "11";` *because the data type* `std_logic_vector` *had nine possible values for each bit. This means that there are actually 81 possible unique values that* `Sel` *could have at any given time.*

4.2.2 Selected Signal Assignment

A selected signal assignment is similar to a conditional signal assignment but differs in that the input conditions have no priority (in other words, there is no `else` condition). For example:

```
entity my_mux is
port ( Sel      : in  std_logic_vector(1 downto 0);
       A,B,C,D  : in  std_logic_vector(0 to 3);
       Y        : out std_logic_vector(0 to 3)
     );
end my_mux;

architecture mux_arch of my_mux is
begin
  with Sel select
  Y <= A when "00",
       B when "01",
```

```
       C when "10",
       D when others;
end mux_arch;
```

The selected expressions may include ranges and multiple values. For example, you could specify ranges for a `std_logic_vector` selection as follows:

```
with Address select
  CS <= SRAM when X"0000" to X"7FFF",
        PORT when X"8000" to X"81FF",
        UART when X"8200" to X"83FF",
        PROM when others;
```

You may also use the `unaffected` keyword as shown:

```
with Sel select
  Y <= A when "00",
       B when "01",
       C when "10",
       unaffected when others;
```

4.2.3 Procedure Calls

Procedures may be called concurrently within an architecture. When procedures are called concurrently, they must appear as independent statements within the concurrent area of the architecture. You can think if procedures in the same way you think of processes within an architecture - as independent sequential programs that execute whenever there is a change (an event) on any of their inputs. The advantage of a procedure over a process is that the body of the procedure (it's sequential statements) can be kept elsewhere (in a package for example), and called repeatedly throughout the design. In the following example, a procedure called `dff` is called from the concurrent area of the architecture:

```
architecture example of shift_reg is
...
begin
  ...
  dff(..);
  ...
end example;
```

4.2.4 Generate Statements

Generate statements are provided as a convenient way to create multiple instances of concurrent statements, most typically component instantiation statements. There are two basic varieties of generate statements. The following example shows how you might use a `for-generate` statement to create four instances of a lower-level component (in this case, a RAM block):

```
architecture example1 of my_entity is
  component RAM16x1
    port ( A0,A1,A2,A3,WE,Din : in std_logic;
           Dout : out std_logic
         );
  end component;
begin
  ...
  for i in 0 to 3 generate
    RAM : RAM16x1 port map(...);
  end generate;
  ...
end example1;
```

When this `generate` statement is evaluated, the compiler will generate four unique instances of component `RAM16x1`. Each instance will have a unique name that is based on the instance label, `RAM` and the index value. `For-generate` statements can be nested, so it is possible to create multi-dimensional arrays of component instances or other concurrent statements.

The `if-generate` statement is most useful when you need to conditionally generate a concurrent statement, and uses a similar syntax to that shown above.

4.2.5 Component Instantiations

Component instantiations are statements that reference lower-level components in your design, in essence creating unique copies (or instances) of those components. A component instantiation statement is a concurrent statement, so there is no significance to the order in which components are referenced. You must, however, declare any components that you reference in either the declarative area of the architecture (before the `begin` statement) or in an external package.

The example used in Section 3.2 is included again below to illustrate how component instantiations are written.

```
architecture structure of flipflop is
  component nand_gate
    port(a, b : in  std_logic;
         z    : out std_logic);
  end component;
begin
  top: nand_gate
    port map (s, nq, q);
  bot: nand_gate
    port map (r, q, nq);
end structure;
```

The mapping of ports in a component can be described in one of two ways. The simplest method (and the method used above) is called *positional association*. Positional association simply maps signals in the architecture (the actuals) to corresponding ports in the lower-level entity declaration (the formals) by their position in the port list. When using positional association, you must provide exactly the same number and types of ports as are declared for the lower-level entity.

Although it is quick and easy to use, positional association has some potential problems. The most troublesome is the lack of error checking. It is quite easy for example, to swap the order of two ports in the list, and yet the code will still compile without errors. A better way of mapping ports is to used *named association*. Named association is an alternative form of port mapping that includes both the actual and formal port names in the port map of a component instantiation.

When specifying port mappings using named association, lower level names are written on the left side of the `=>` operator while the top level names are written on the right. This is shown in the example below:

```
architecture structure of flipflop is
  component nand_gate
    port(a, b : in  std_logic;
         z    : out std_logic);
  end component;
begin
  top: nand_gate
    port map (a => s, b => nq, z => q);
  bot: nand_gate
    port map (a => r, b => q, z => nq);
end structure;
```

The benefits of named association go beyond simple error checking. Because named association removes the requirement for any particular order of the ports, you can enter them in whatever order you want. You can even leave one or more ports unconnected if you have provided default values in the lower-level component specification.

4.2.6 Generic Mapping

If the lower-level entity being referenced includes *generics* (described later), you can specify a generic map in addition to the port map to pass actual generic parameters to the lower level entity:

```
architecture structure of flipflop is
  component nand_gate
    port(a, b : in  std_logic;
         z    : out std_logic
  end component;
begin
  top: nand_gate
    generic map(tRise => 1ns, tFall => 1ns);
    port map (a => s, b => nq, z => q);
  bot: nand_gate
    generic map(tRise => 1ns, tFall => 1ns);
    port map (a => r, b => q, z => nq);
end structure;
```

4.3 Sequential Statements

Sequential VHDL statements allow you describe the operation of your circuit as a sequence of related events. Such descriptions are natural for order-dependent circuits such as state machines and for complex combinational logic that involves some priority of operations. Sequential statements are found within processes, functions and procedures, and differ from concurrent statements in that they have order dependency.

The *process* statement is the primary method used to enter sequential commands. Since a process is considered a concurrent statement, you can write as many processes and other concurrent statements as are necessary to describe your design, without worrying about the order in which the synthesizer will implement each concurrent statement.

The general form of a process statement is:

```
[process_name:] process[(sensitivity_list)]
[declarations]
  begin
    ...
    ...
  end process;
```

The process name appearing before the `process` keyword is optional and can be used to identify specific processes or to clearly identify elements such as local variables which may have common names in different processes. Immediately following the `process` statement is an optional list of signals enclosed in parentheses. This list of signals is called the sensitivity list, and it specifies the conditions under which the process is to begin execution. Any change in the value of a signal in the sensitivity list will result in immediate execution of the process. Although there is a definite order of operations within a process (from top to bottom), you can think of a process as executing in zero time. This means that a process can be used to describe circuits functionally, without regard to their actual timing, and multiple processes can be executed in parallel with little or no concern for which processes complete their operations first. In the absence of a sensitivity list, the process will execute continuously, but must be provided with at least one `wait` statement to cause the process to suspend periodically.

4.3.1 State Machines

State machines are a common form of sequential logic circuitry that are used for generating or detecting sequences of events. To describe a synthesizable state machine in VHDL, you should follow a well-established coding convention that makes uses of enumerated types and processes. Specifying state machine encodings can be a long and tedious process, the methods for which are outside the scope of this lecture series.

4.3.2 Signals and Variables

One important aspect of VHDL you should clearly understand is the relationship between sequential statements, and the scheduling of signal and variable assignments. Signals within processes have fundamentally different behavior from variables. Variables are assigned new values immediately, while signal assignments are scheduled and do not occur until the current process has been suspended. Deciding whether to use a signal or variable for a given object can require careful consideration of the requirements of your design.

Chapter 5

Modularization

Modular or structured programming is a technique that you can use to enhance your own design productivity, as well as that of your design team. A modular design approach allows commonly used segments of code to be re-used. It also enhances readability.

5.1 Subprograms

Functions and procedures are collectively known as subprograms, and are directly analogous to functions and procedures in languages such as C or Pascal. A procedure is a subprogram that has an argument list consisting of inputs and outputs, and no return value. A function is a subprogram that has only inputs in its argument list and has a return value.

Statements within a subprogram are sequential, regardless of where the subprogram is invoked. Subprograms can be called from with the concurrent area of an architecture or from within a sequential process or higher-level subprogram. They can also be invoked from within other subprograms.

It is useful to think of subprograms as a process that (a) is located outside the body of the architecture, and (b) operates only on its input (and in the case of procedures), and output parameters. Nesting of functions and procedures is allowed to any level of complexity, and recursion is also supported, although it is not synthesizable.

5.1.1 Global or Local?

Functions and procedures can be declared either globally, so they are usable throughout a design description, or they can be declared locally within the declarative region of an architecture, block, process or even within another

subprogram. An example of a global declaration is shown below:

```
package my_package is
  function my_function(...)
    return std_logic;
end my_package;

package body my_package is
  function my_function(...)
  return std_logic is
  begin
    ...
  end my_function;
end my_package;

use work.my_package.my_function;
entity my_design is
begin
  ...
end my_design;
```

Compare this to a locally declared function as shown:

```
architecture example of my_circuit is
begin
  process(...)
    function my_function(...)
    return std_logic is
    begin
      ...
    end my_function;
  begin
    ...
  end process;
end example;
```

5.1.2 Functions and Procedures

A function is a subprogram that accepts zero or more input arguments and returns a single output value. Because a function returns a value, it has a type associated with it. The following is an example of a function that accepts two integer arguments and returns the greater of the two as an integer value:

```
function maxval(arg1,arg2 : integer) return integer is
  variable result : integer;
begin
  if arg1 > arg2 then
    result := arg1;
  else
    result := arg2;
  end if;
  return result;
end maxval;
```

Functions are commonly used is situations where you require a calculation or conversion based on the subprogram inputs. Examples of this include arithmetic or logic functions, type conversion functions, and value checks such as you might use when writing a test bench.

Procedure differ from functions in that they do not have a return value, and their arguments may include both inputs and outputs to the subprogram. Because each argument to a procedure has a mode (in or out), they can be used very much like you would use an entity/architecture pair to help simplify and modularize a large and complex design description.

Procedures are used as independent statements, either within the concurrent area of an architecture or within the sequential statement area of a process or subprogram. The following example defines the behavior of a clocked JK flipflop with an asynchronous reset input:

```
procedure jkff(signal rst,clk,j,k : in std_logic;
               signal q,qbar : buffer std_logic) is
begin
  if rst = '1' then
    q <= '0';
  elsif clk = '1' and clk'event then
    if j = '1' and k = '1' then
      q <= qbar;
    elsif j = '1' and k = '0' then
      q <= '1';
    elsif j = '0' and k = '1' then
      q <= '0';
    end if;
  end if;
  qbar <= not q;
end jkff;
```

Note that variables declared and used within a procedure are not preserved between different executions of the procedure. This is unlike a process, in which variables maintain their values between executions.

5.1.3 Parameter Mapping

The examples presented above have used *positional association* to describe how the actual parameters are paired with the formal parameters of the subprogram. Just as with the mapping of ports in an entity, subprogram parameters can also use *named association*.

5.2 Partitioning Your Design

VHDL provides many high-level features to help you manage a complex design description. Design partitioning goes beyond simpler design modularity methods to provide comprehensive design management across multiple projects and allow alternative structural implementations to be tried out with minimal effort.

The design partitioning features of VHDL include:

- Blocks,
- Packages,
- Libraries,
- Components, and
- Configurations.

5.2.1 Blocks

Blocks in VHDL are analogous to sheets in a multi-sheet schematic. They do not represent re-usable components, but do enhance readability by allowing declarations of objects to be kept close to where those objects are actually used. Blocks are the simplest form of design partitioning. They provide an easy way to segment a large VHDL architecture into multiple self-contained parts. Blocks allow the logical grouping of statements within an architecture, and provide a place to declare locally used signals, constants and other objects as needed.

5.2.2 Packages

Packages are intended to hold commonly used declarations such as constants, type declarations and global subprograms. Packages can be included within the same source files as other design units, or may be placed in a separate source file and compiled into a named library. When items from a package are required in other design units, you must include a `use` statement to make the package and its contents visible.

The Package Body

Every package can have at most, one package body. Package bodies are optional, and are only required when a package includes subprograms or deferred constants.

5.2.3 Libraries

Design libraries are used to collect commonly used design units (typically packages and package bodies) into uniquely named areas that can be referenced from multiple source files in your design. If you do not specify a library, the design units are compiled into a default library named `work`. For simple designs, you will use the `work` library exclusively, and all you need to do is specify a `use` statement such as:

```
use work.my_package.all;
```

prior to each entity declaration in your design file for each package that you have declared in your source file. The `use` statement is quite flexible. You can specify exactly which items within a package are to be made visible, specify all items from that package, or all items from that library:

```
use my_lib.my_package.dff;  -- only the dff procedure
use my_lib.my_package.all;  -- all subprograms are visible
use my_lib.all;             -- everything in the library!
```

5.2.4 Components

Components are use to connect multiple VHDL design units (entity / architecture pairs) together to form a larger, hierarchical design. The following example describes the relationship between three such design units:

```
architecture structure of shiftcomp is
  component compare
```

```
    port(A,B in std_logic_vector(0 to 7);
         EQ : out std_logic
        );
  end component;
  component shift
    port(Clk,Rst,Load : in std_logic;
         Data : in std_logic_vector(0 to 7);
         Q : out std_logic_vector(0 to 7)
        );
  end component;
  signal Q : std_logic_vector(0 to 7);
  begin
    COMP1 : compare port map (Q,Test,Limit);
    SHIFT1 : shift port map (Clk,Rst,Load,Init,Q);
end structure;
```

In this example, the two lower level components, `shift` and `compare`, were instantiated in the higher level module, `shiftcomp`, to form a hierarchy of design units. Component instantiations are concurrent statements and therefore have no order dependency. Again, either named or positional association can be used to pass parameters into a component.

5.2.5 Generics

It is possible to pass instance-specific information other than actual port connections to an entity using a feature called generics. Generics are very useful for making design units more general-purpose, or for annotating information such as timing specifications to an entity at the time the design is analyzed. The following example shows how generics can be used to create a parameterizable model of a D flipflop:

```
library ieee;
use ieee.std_logic_1164.all;

entity dffr is
  generic(width : positive);
  port(Rst,Clk : in std_logic;
       signal D : in std_logic_vector(width-1 downto 0);
       signal Q : out std_logic_vector(width-1 downto 0)
      );
end dffr;
```

```
architecture behavior of dffr is
begin
  process(Rst,Clk)
    variable Qreg : std_logic_vector(width-1 downto 0);
  begin
    if Rst = '1' then
      Qreg := (others => '0');
    elsif Clk = '1' and Clk'event then
      for i in Qreg'range loop
        Qreg(i) := D(i);
      end loop;
    end if;
    Q <= Qreg;
  end process;
end behavior;
```

In this example, the `dffr` entity has a generic list in addition to a port list. This generic list contains one entry, a positive integer, that corresponds to the width of the `D` input and `Q` output. The architecture declaration uses a `for` loop in conjunction with the generic (`width`) to describe the operation of the D flipflop.

When instantiated in a higher level design, a generic map must be provided in addition to the port map, as shown in the example below, which illustrates how three instances of the `dffr` design unit can be created using different values for the generic:

```
architecture sample of reg is
  component dffr
    generic(width : positive);
    port(Rst,Clk : in std_logic;
         signal D : in std_logic_vector(width-1 downto 0);
         signal Q : out std_logic_vector(width-1 downto 0)
        );
  end component;
  constant width8 : positive := 8;
  constant width16 : positive := 16;
  constant width32 : positive := 32;
  signal D8,Q8 : std_logic_vector(7 downto 0);
  signal D16,Q16 : std_logic_vector(15 downto 0);
  signal D32,Q32 : std_logic_vector(31 downto 0);
```

```
  begin
    FF8: dffr generic map(width8) port map(Rst,Clk,D8,Q8);
    FF16: dffr generic map(width16) port map(Rst,Clk,D16,Q16);
    FF32: dffr generic map(width32) port map(Rst,Clk,D32,Q32);
end sample;
```

5.2.6 Configurations

Configurations are not generally supported by synthesis tools, but allow large, complex designs to be managed during simulation. One example of how you might use configurations is to construct two versions of a system-level design, one of which makes use of high-level behavioral descriptions of the system components, while a second version substitutes in a post synthesis timing model of one or more components. Since this course is more oriented towards synthesis, rather than simulation, an in-depth discussion on configurations will not be presented.

Chapter 6

Optimising for Synthesis

When using synthesis tools, the single most productive thing you can do is be aware of what kind of hardware you are describing. Writing a design description without considering the hardware, and just expecting the sythesis tool to do the design for you is a recipe for disaster. A common mistake is to create a design description, validate that description with a simulator, and assume that this *correct* specification must also be a *good* specification. Understanding the hardware that you are specifiying is the simplest rule for success. This is particularly important if you want to achieve critical timing goals.

To get the best results out of synthesis, and to achieve the highest possible portability between synthesis tools, it is important to describe your designs using well-established synthesis conventions, and to be aware of some common mistakes of first-time synthesis users. This chapter will look at some simple examples of typical circuit coding problems.

6.1 Simulation Optimized Code

In the following example, it is assumed that only one control input will be active at a time. This description is efficient for simulation, but is a poor logic description for synthesis because the independence of the control signals is not described within the VHDL code:

```
out1 <= '0';
out2 <= '0';
out3 <= '0';
if in1 = '1' then
  out1 <= '1';
  elsif in2 = '1' then
```

```
    out2 <= '1';
    elsif in3 = '1' then
      out3 <= '1';
end if;
```

The independence of the control signals needs to be contained within the design description, or inefficient synthesis will result. The modified design description may be slightly slower during simulation, but will result in a smaller logic implementation after synthesis:

```
out1 <= '0';
out2 <= '0';
out3 <= '0';
if in1 = '1' then
  out1 <= '1';
end if;
if in2 = '1' then
  out2 <= '1';
end if;
if in3 = '1' then
  out3 <= '1';
end if;
```

Note the issue here is not a long signal path, but an unclear specification of the design. The best optimizer in the world can't turn an inefficient algorithm into an efficient one. And an algorithm that is efficient from one viewpoint may not be efficient from another.

6.2 Testing For High Impedance

The following examples uses the IEEE 1164 standard logic data types and values in an attempt to describe the concept "if signal `sig` is floating". This is quite a reasonable test to perform in a simulation model. However, a synthesis tool has to transform this into a hardware element that matches this behavior.

```
if sig = 'Z' then
  -- do something
end if;
```

As written, if `if-then` test specifies a logic cell that looks at the drive of its fan-in, then outputs `true` if not driven, and `false` if driven `high` or

`low`. Such a cell does not exist in most programmable silicon. IEEE 1076.3 specifies that this comparison should always be false, so the statements inside the `if` are not executed, and no logic is generated. Tests for high impedance on inputs are therefore not meaningful in the context of synthesis.

6.3 Nested Ifs

Mutliple nested `if` or `elsif` clauses can specify long signal paths. The following example describes a chain of dependent `if-then` statements:

```
if sig = "000" then
  -- first branch
  elsif sig = "001" then
    -- second branch
    elsif sig = "010" then
      -- third branch
      elsif sig = "011" then
        -- fourth branch
        elsif sig = "100" then
          -- fifth branch
          else
            -- last branch
end if;
```

This code is an inefficient way to describe logic. A `case` statement would be much better.

```
case sig is
  when "000" =>
    -- first branch
  when "001" =>
    -- second branch
  when "010" =>
    -- third branch
  when "011" =>
    -- fourth branch
  when "100" =>
    -- fifth branch
  when others =>
    -- last branch
end case;
```

In practice, if the branches contain very little logic, or if there are few branches, then there may be little difference in the amount of logic generated. Howere the `case` statement generally results in a better implementation, so you should use `case` statements rather than `if-then-else` statements whenever possible.

6.4 Iterative Loops

Loops are very powerful, but each iteration of a loop replicates logic. A variable that is assigned in one iteration of a loop and used in the next iteration results in a long signal path. This signal path may not be obvious.

6.5 Using `buffer` or `inout`

Mode `inout` specifies a bi-directional dataflow. `Buffer` on the other hand, specifies unidirectional dataflow. There are very few occassions in hardware design when bi-directional is actually what you want, so you should use `buffer` for most cases in which you must (locally) read from an output port. Use `inout` only when you want to specify a signal path that is actually routed bidirectionally through a pin, such as when describing an I/O pad or a bus.

6.6 Initial Values

The initial value of an object is its value when created. Signals and variables declared in processes are created at *zero time*. Variables in subprograms are created when the subprogram is called.

The value at time zero has no meaning in the context of synthesis. Therefore, the initial value of singals and process variables must be used with care. This issue does not arise with the initial value of variables declared in subprograms.

A general rule is this: You should not depend on the initial value of signals or process variables if they are not completely specified in the process in which they are used. In this case, the compiler willl ignore the time zero condition and use the driven value. For example:

```
signal res : std_logic := '0';
begin
  process(tmp,init)
  begin
```

```
    if (tmp = 26) then
      res <= '1';
    elsif (init = '1') then
      res <= '1';
    end if;
  end process;
...
```

In this case, `res` is never assigned low - the code will be synthesized as a pull-up. However, during simulation at time zero, `res` starts at '0', and makes a transition to '1', then stays there. If this is really the intent, the proper solution is to use a flipflop.

6.7 Unintended Latches

Latches, whether intended or accidental, are inferred using an incomplete specification in an `if` statement. The following example specifies a latch gated by `address_strobe`, which may not have been the intent.

```
process(address, address_strobe)
begin
  if address_strobe = '1' then
    decode_signal <= address = "101010";
  end if;
end process;
```

This description specifies that when `address_strobe` is '0', then the signal `decode_signal` holds its previous value, resulting in a latch implementation. In this case the intent is probably to ignore `decode_signal` when `address_strobe` is '0'. The following, more explicit, code is correct:

```
if address_strobe = '1' then
  decode_signal <= address = "101010";
else
  decode_signal <= false;
end if;
```

Chapter 7

Test Benches

A test bench is a program written in VHDL, and is used to test your synthesizable VHDL program. When you first create a new circuit design, you should also create a test circuit in parallel with that design. Whereas synthesizable VHDL is restricted to a subset of the VHDL syntax, code intended to be used as a test bench can make full use of the language. Test benches can be quite simple, applying a sequence of inputs to the circuit over time. They can also be quite complex, perhaps even reading test data from a disk file and writing test results to the screen and to a report file. A comprehensive test bench can be more complex and lengthy (and can take longer to develop) than the synthesizable circuit being tested.

The simplest test benches are those that apply some sequence of inputs to the circuit being tested, so that it's operation can be observed in simulation. Waveforms are typically used to represent the values of signals in the design at various points in time. Such a test bench must consist of a component declaration corresponding to the unit under test, and a description of the input stimulus being applied.

The following example demonstrates the simplest form of a test bench, and tests the operation of a NAND gate:

```
library ieee;
use ieee.std_logic_1164.all;

use work.nandgate;

entity testnand is
end testnand;

architecture stimulus of testnand is
```

```
component nand
  port(a,b : in std_logic;
       z : out std_logic);
end component;

signal a,b,z : std_logic;

begin
  nand1 : nandgate port map(a=>a, b=>b, z=>z);
  process
  constant period : time := 50ns;
    begin
      a <= '1';
      b <= '1';
      wait for period;
      assert(z = '0')
        report "Test failed" severity ERROR;
      a <= '1';
      b <= '0';
      wait for period;
      assert(z = '1')
        report "Test failed" severity ERROR;
      a <= '0';
      b <= '1';
      wait for period;
      assert(z = '1')
        report "Test failed" severity ERROR;
      a <= '0';
      b <= '0';
      wait for period;
      assert(z = '1')
        report "Test failed" severity ERROR;
      wait;
  end process;
end stimulus;
```

Note that the entity clause does not include any port declarations, as it is the highest-level design unit when simulated. Also note that the process statement does not have a sensitivity list. It uses `wait` statements to provide a specific amount of delay (defined using `period`) between each new combination of inputs. `Assert` statements are used to verify that the circuit is

operating correctly for each combination of inputs. Finally, a `wait` statement without any condition expression is used to suspend simulation indefinitely after the desired inputs have been applied. In the absence of the final `wait` statement, the process would repeat forever, or as long as the simulator has been instructed to run.

7.1 Using Assert Statements

The `assert` statement provides a quick and easy way to check expected values and display messages from your test bench. When analyzed (either during execution as a sequential statement, or during simulator initialization in the case of a concurrent assert statement), the condition expression is evaluated. As in an `if` statement, the condition expression of an assert statement must evaluate to a boolean value. If the condition expression is `false` - indicating that the assertion failed - the text specified in the `report` statement clause is displayed. The `severity` statement clause then indicates to the simulator what action (if any) should be taken. The severity level can be specified using one of the following predefined severity levels: `NOTE, WARNING, ERROR` or `FAILURE`.

7.2 Loops and Multiple Processes

Test benches can be dramatically simplified through the use of loops, constants and other more advanced features of VHDL. Using multiple concurrent processes in combination with loops can result in very concise descriptions of complex input and expected output conditions.

The following example demonstrates how a loop (in this case a `while` loop might be used to create a background clock in one process, while other loops (in this case `for` loops are used to apply inputs and monitor outputs over potentially long periods of time:

```
clock1: process
  variable clktmp : std_logic := '1';
begin
  while done /= true loop
    wait for period/2;
    clktmp := not clktmp;
    Clk <= clktmp;
  end loop;
end process;
```

```
stimulus1: process
begin
  reset <= '1';
  wait for period;
  reset <= '0';
  mode <= '0';
  wait for period;
  mode <= '1';
  for i in 0 to 127 loop
    wait for period;
    assert(VS = '1')
      report"VS went high at the wrong place" severity ERROR;
  end loop;
  assert(VS = '1')
    report"VS was not detected" severity ERROR;
  wait for period;
  TestLoad <= '0';
  for i in 0 to 300 loop
    Data <= RandomData();
    wait for period;
  end loop;
  assert(EOF = '1')
    report"EOF was not detected" severity ERROR;
  done <= true;
  wait;
end process;
```

In this example, the process `clock1` uses a local variable, `clktmp` to describe a repeating clock with a period defined by the constant `period`. This clock is defined with a `while` loop, and it runs independently of all other processes in the test bench until the `done` signal is asserted `true`. The second process `stimulus1`, describes a sequence of inputs to be applied to the unit under test. It also makes use of loops - in this case `for` loops - to describe lengthy repeating stimuli and expected value checks.

Acknowledgments

Most of these notes are based on the book, "VHDL Made Easy" written by David Pellerin and Douglas Taylor, and published by Prentice Hall in 1996.

Steven J. Merrifield
Department of Electronic Engineering
La Trobe University
Bundoora
Victoria, 3083
AUSTRALIA
email: *sjm@ee.latrobe.edu.au*

March 1998