Notes for Digital Design using VHDL

Hardware Description Language: (HDLs):

A Hardware Description Language (HDL) is a software programming language used to model the intended operation of a piece of hardware.

VHDL

VHDL which stands for (Very high speed integrated circuit Hardware Description Language) is a programming language that has been designed and optimized for describing the behavior of digital systems.

Just as high-level programming languages allow complex design concepts to be expressed as computer programs, VHDL allows the behavior of complex electronic circuits to be captured into a design system for automatic circuit synthesis or for system simulation.

Like Pascal, C and C++, VHDL includes features useful for structured design techniques, and offers a rich set of control and data representation features. Unlike these other programming languages, VHDL provides features allowing concurrent events to be described. This is important because the hardware described using VHDL is inherently concurrent in its operation.

History of VHDL

1980:
The USA department of defense (DOD) wanted to make circuit design self documenting.

1983:
The development of VHDL began with a joint effort by IBM, Texas Instruments and Intermetrics.

1987:
The institute of Electrical and Electronics Engineers (IEEE) was presented with a proposal to standardize the language. The resulting standard, IEEE 1076-1987, is the basis for virtually every simulation and synthesis product sold today.

1993:
The VHDL language was revised to IEEE 1076-1993

1996:
A VHDL package for use with synthesis tools become part of the IEEE 1076 standard, specifically it is 1076.3. This greatly improved the portability of designs between different synthesis vendor tools. Another part of the standard, IEEE 1076.4 (VITAL), has been completed and sets a new standard for modeling ASIC and FPGA libraries in VHDL. This made life considerably easier for ASIC, FPGA and EDA tools vendors.

Verilog

Verilog was introduced first before VHDL, thus established itself as the de facto standard language for ASIC simulation libraries; Verilog has some advantage in availability of simulation models. Another important feature that is defined in Verilog is a programming language interface PLI. The PLI makes it possible for simluation model writers to go outside of Verilog when necessary to create faster simulation models, or to create functions (using the C language) that would be difficult or inefficient to implement directly in Verilog.

History of Verilog

1981:
A CAE (Computer Aided Engineering) software company called Gateway Design Automation was founded by Prabhu Goel.

1983:
Gateway released the Verilog hardware description language known as "Verilog HDL" together with a Verilog simulator.

1985:
The language and simulator has enhanced; the new version of the simulator was called "Verilog-XL".

1987:
Verilog-XL was becoming very popular and has been used by many ASIC vendors. Another start-up company, Synopsys, began to use the proprietary Verilog behavioral language as an input to their synthesis product.

1989:
Cadence bought Gateway.

1995:
The Verilog language was reviewed and adopted by IEEE as IEEE standard 1364.

First VHDL example:

As we previously declared that VHDL is a language that describes a hardware, so let us examine a very simple description of a schematic using VHDL. Our first example is a simple OR gate.

In VHDL we start our implementation of the hardware by describing its interface and this piece of information will be located under a section called the entity, we began by giving the entity a unique name and then continue to describe its inputs and outputs. A typical entity description of the OR gate will be like that:

```
entity or_gate is
  port (a, b: in bit;
        y: out bit);
end entity or_gate;
```

In this simple description we defined a block and then give it a name which is “or_gate” and then we described its interface, we specified “a, b, y” as the name of the ports of this block, and then classify these ports with “in” and “out” into inputs and outputs, we also declared the type pf these ports as “bit’ which specifies a type that can only handle two values “0” or “1”.

After that we need to write a description that specifies the function of this block or entity, that can be expressed as:

```
architecture behavior of or_gate
is begin
  y <= a or b;
end architecture behavior;
```

The functionality of the block was specified using the “architecture” structure which inside it we wrote the line “y <= a or b” to describe how the inputs and outputs are related.

The entity and architecture are called design units in VHDL. There are five kinds of design units in VHDL. These are:

entity;
architecture;
package;
package body; and
configuration.

The five kind of design unit are further classified as primary or secondary units. A primary design unit can exist on its own.

A secondary design unit cannot exist without its corresponding primary unit. In other words, it is not possible to analyze a secondary unit before its primary unit is analyzed.

The entity is a primary design unit that defines the interface to a circuit as we already declared. Its corresponding secondary unit is the architecture that defines the contents of the circuit. There can be much architecture associated with a particular entity, but this feature is rarely used

The package is also a primary design unit. A package declares, types, subprograms, operations, components and other objects which can then be used in the description of the circuit. The package body is the corresponding secondary design unit which contains the implementations of subprograms and operations declared in its package.

The configuration declaration is a primary design unit with no corresponding secondary. It is used to define the way in which a hierarchical design is to be built from a range of subcomponents. However, it is not used for logic synthesis and will not be covered in this course.

More on entities and architecture

An entity defines the interface to a circuit and the name of the circuit. An architecture defines the contents of the circuit itself. Entities and architecture therefore exist in pairs; a complete circuit will generally have both an entity and an architecture. It is possible to have an entity without an architecture, but such examples are generally trivial and of no real use. It is not possible to have an architecture without an entity.

An example of an entity is:

```
entity full_adder is
  port (a, b, c: in bit; sum, count: out
bit); end entity full_adder;
```

In this case the circuit full_adder has five ports: three input ports and two output ports. Note that the repeat of the circuit name full_adder after the end is optional.

The structure of architecture is given by the following example:

```
architecture behavior of full_adder
  is signal sum1, sum2, c1, c2: bit;
begin
  sum1 <= a xor b;
  c1 <= a and b;
  sum2 <= sum1 xor c;
  c2 <= sum1 and c;
  sum <= c1 or c2;
end architecture behavior;
```

The architecture has the name behavior and belongs to the entity full _adder. It is common practice to use the architecture name behavior for synthesizable architecture. As with the entity, the repeat of the architecture name after the end is optional. Common alternatives to architecture behavior are architecture RTL or architecture synthesis. Architecture names do not need to be unique; indeed, the use of the same architecture name throughout a VHDL design makes it easy to tell at a glance whether a VHDL description is system-level (architecture system), RTL (architecture behavior), or gate-level (architecture netlist). It does not matter what naming convention is used for architecture, but it is recommended that a consistent naming convention is adhered to.

The architecture has two parts: the declarative part and the statement part.

The *declarative part* is the part before the keyword begins. In this example, additional internal signals have been declared here signals are similar to ports but are internal to the circuit.

A signal declaration looks like:

```
signal sum1 : bit;
```

This declares a signal called *sum1*, which has a type, called *bit.* Types will be dealt with in later but for now it is sufficient to say that bit is a logical type which can be used for Boolean equation.

The *statement part* is the part after the begin. This is the description of the circuit itself, in this example the statement part only contains signal assignments describing the full adder as two half adders described by Boolean equations.

The simple signal assignment looks like:

```
sum1 <= a xor b;
```

The left-hand side of the assignment is known as the target of the assignment (in this case *sum1*). The assignment itself has the symbol "<=" which is usually read "gets", as in "signal sum1 gets a xor b".

The right-hand side of the assignment is known as the source of the assignment. The source expression can be as complex as you like. For example, the circuit of the full_ adder example could have been written using just two signals assignment:

```
sum   <= a xor b xor c;
count <= (a and b) or (a and c) or (b and c);
```

In this example the statements have been written in sequence so that the dataflow is from up to bottom. However, this is done readability only; the ordering of the statements is irrelevant. This is because each statement simply defines a relationship between its inputs (the source, on the right-hand side of the assignment) and its output (the target, on the left-hand side).

For example, the following architecture is functionally equivalent to the previous version:

```
architecture behavior of full_adder
  is signal sum1, sum2, c1, c2: bit;
begin
  count <= c1 or c2;
  sum <= sum2;
  c2 <= sum1 and c;
  c1 <= a andb;
  sum2 <= sum1 xor b;
  sum1 <= a xor b;
end architecture behavior;
```

Signals and ports

Signals are the carriers of data values around an architecture. Ports are the same as signals but also provide an interface through the entity so that the entity can be used as a subcircuit in a hierarchical design.

A signal is declared in the declarative part of an *architecture* (between the keywords is and begin) and the declaration has two parts:

```
architecture behavior of adder
  is signal a, b, c: bit;
begin
 …
```

The first part is a list of signal names; in this case there are three signals, a, b, and c. the second part, after the colon, is the type of the signals: in this case bit.

There can be many signal declaration in an architecture, each terminated by a semi-colon. The above declaration could be rewritten as three separate declarations:

```
architecture behavior of adder
  is signal a: bit;
  signal b: bit;
  signal c: bit;
begin
 …
```

Port declarations are enclosed by a *port specification* in the entity. The port specification has the following structure:

```
entity adder is
  port (port specification);
end entity adder;
```

Note that the port specification is always terminated by a semi-colon which is outside the parentheses.

A port is declared within the port specification. A port declaration has three parts:

```
entity adder is
  port (a, b, c : in bit);
end entity adder;
```

The first part is a list of port names: in this case a, b and c. the second part is the mode of the port; in this case the mode is in. the third part is the type as in the signal declaration; in this case the type is bit.

Each port declaration within the specification is separated by semi-colons from the others. Note that, unlike the signal declarations which are each terminated by semi-colon, port declarations are separated (not terminated) by semi-colons, so there is no semi-colon after the last declaration before the closing parenthesis. The full_adder entity example is reproduced here to show how multiple port declarations are listed:

```
entity full_adder is
```

```
    port (a, b, c: in bit; sum, count: out
  bit); end entity full_adder;
```

The mode of the port determines the direction of data-flow through the port. There are five port modes: in, out, inout, buffer, and linkage. If a mode is not given, then mode in will be assumed.

The meaning of the modes as they are used for logic synthesis are:

in input port – cannot be assigned to in the circuit, can be read;
out output port – must be assigned to in the circuit, cannot be read;
inout bidirectional port – can only be used for tristate buses; buffer output port – like mode out but can also be read; linkage not used by synthesis.

There is often confusion between mode *out* and mode *buffer*. Mode buffer is an anachronism and it is an understatement to say that the reason for its existence in the language is obscure. The full behavior of a buffer port is a restricted form of mode input. However, to make the mode usable for synthesis, the rules for buffer ports are constrained so that they act like mode out ports with the added convenience that it is possible to read from the port within the architecture.

The behavior of the modes is illustrated by the following example; inout mode is omitted because it is specific to tristates which are dealt with in chapter 12.

```
entity modes is
  port (input: in bit;
        out1: buffer bit;
        out2: out bit);
end entity modes;

architecture behavior of modes is
begin
  out1 <= input;
  out2 <= out1;
end architecture behavior;
```

Port inout, being mode in, cannot be assigned to and so can only appear on the right-hand side of a signal assignment. Port out2, being mode out, can only be assigned to and so can only appear on the left-hand side of a signal assignment. Port out1, however, being mode buffer, can be assigned to and read and so can appear on either side of an assignment. It has been named out1 in this example because it is acting as an output.

Unfortunately, the rules for mode buffer are more complex than this; this interpretation of mode buffer is a simplification to make the mode useful for synthesis and is common to most synthesizers. The full rules also prevent mode buffer and mode out from being mixed in a hierarchical design. That is to say, it is not possible to connect a buffer mode port of a subcomponent to an out mode port of a higher level component.

This restriction on the mixing of modes can cause inconvenience, at the very least. It is therefore recommended that only one of the output modes is ever used.

Furthermore, owing to the anachronistic nature of mode buffer, it is recommended that only out mode is used.

The problem of needing to read from an out mode port is illustrated by the following example of an and gate with true and inverted output. The first version shows an *illegal* description, because the out port z is read:

```
entity and_nand is
  port (a, b: in bit;
        z, zbar: out bit);
end entity and_nand;

architecture illegal of and_nand
is begin
  z <= a and b;
  zbar <= not z;
end architecture illegal;
```

The solution is to use an intermediate internal signal and read from that. The intermediate signal can then be assigned to the out mode ports.

The corrected example is:

```
entity and_nand is
  port (a, b: in bit;
        z, zbar: out bit);
end entity and_nand;

architecture behavior of and_nand
  is signal result: bit;
begin
  result <= a and b;
  z <= result;
  zbar <= not result;
end architecture behavior;
```

It is good practice always to use intermediate signals for outputs and to assign them to the out ports at the end of the architecture. By doing this consistently, the pitfall of trying to read an out port is always avoided.

Simple signal assignment

The simple signal assignment statement has already been used in the full_adder example. The section examines the signal assignment statement in more detail.

The simple signal assignment looks like this:

```
x <= a xor b;
```

The left -hand side of the assignment is known as the *target* of the assignment (in this case x). The right-hand side of the assignment is known as the *source* expression of the assignment (in this case a xor b).

The assignment itself is the symbol <=, which is formed by combining the less-than and the equals symbols to form an arrow. There must be no space between the two characters in the symbol. Beware of confusion with the less-than-or-equal operator <= which looks exactly the same. Fortunately, there is no real chance of confusion because there are no situations where both meanings would be allowed by the language. Nevertheless, it takes some getting used to.

The rules of VHDL insist that the source of an assignment is the same *type* as the target. This example uses type bit. The xor of two signals of type bit gives a result which is also a bit. Therefore, the source and target are of the same type.

The source expression can be as complex as desired. For example, the circuit of the full_adder example could have been written using just two signal assignments:

```
sum <= a xor b xor c;
count <= (a and b) or (a and c) or (b and c);
```

Combinational logic blocks

Multiplexers

A multiplexer selectively passes the value of one, of two or more iput signals, to the output. One or more control signals control which input signal's value is passed to the output, see figure 1.4. Each input signal, and the output signal, may represent single bit or multiple bits busses. The select inputs are normally binary encoded such that n select inputs can select from one of up to 2^n inputs.

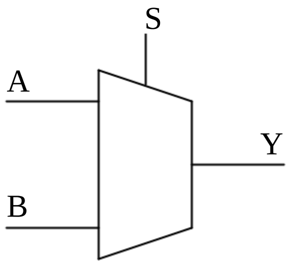

Figure 1.4 circuit symbol of 2-1 multiplexer

S	A	B	Y
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1

Table 1.1 Truth table of 2-1 multiplexer

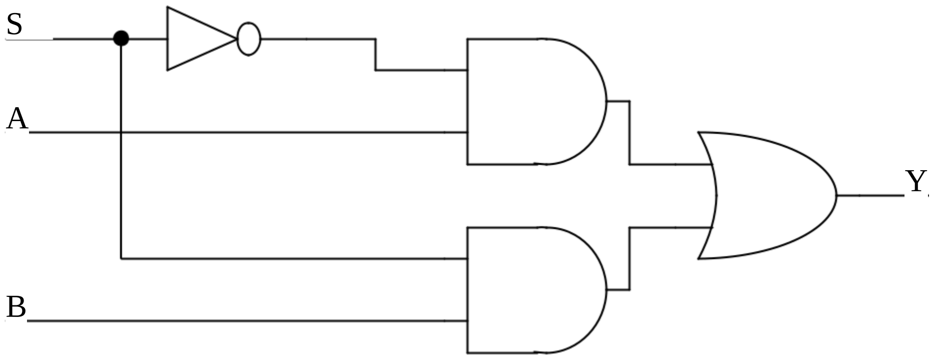

Figure 1.5 logic implementation of 2-1 multiplexer

Example 1.1 Modeling of a 2-1 multiplexer

The model of the 2-1 multiplexer described above is shown modeled by two different methods: first, using a concurrent selected signal assignment, second using the **if** statement in its simplest form.

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

```
entity mux_2_1 is
  port (s, a, b: in std_logic;
        y: out std_logic) ;
end entity mux_2_1;

architecture first of mux_2_1
is begin
  y <= a when s = ‘1’ else
       b;
end architecture first;

architecture second of mux_2_1
is begin
  process (s, a, b)
  begin
    if (s = ‘1’) then
      y <= a;
    else
      y <= b;
    end if;
  end process;
end architecture second;
```

Example 1.2 Modeling of a 4-1 multiplexer

Four ways of modeling a 4-1 mutliplexer are indicated. They are:

- Using a conditional signal assignment (concurrent)
- Using a selected signal assignment (concurrent)
- One **if** statement with multiple **elsif/ else** clauses
- Using a **case** statement

There is no incorrect modeling method, however using the case statement requires less code and is easier to read when compared with the if statement. This becomes more distinct with increasing inputs per output. The first and second models use concurrent signal assignments so reside outside a process. This means they are always active and so will usually take longer to simulate.

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

entity mux4_1 is
  port (sel: in std_logic_vector (1 downto 0);
        a, b, c, d: in std_logic;
        y: out std_logic);
end entity mux4_1;
architecture first of mux4_1
is begin
```

```
  y <= a when sel = “00” else
       b when sel = “01” else
       c when sel = “10” else
       d; --when sel = “11” end
architecture first;
architecture second of mux4_1
is begin
  with sel select
    y <= a when “00”,
         b when “01”,
         c when “10”,
         d when “11”,
         a when others;
end architecture second;

architecture third of mux4_1 is
begin
  process (sel, a, b, c, d)
  begin
    if (sel = “00”) then
      y <= a;
    elsif (sel = “01”) then
      y <= b;
    elsif (sel = “10”) then
      y <= c;
    else
      y <= d;
    end if;
  end process;
end architecture third;

architecture fourth of mux4_1
is begin
  process (sel, a, b, c, d)
  begin
    case sel is
      when “00”    => y <= a;
      when “01”    => y <= b;
      when “10”    => y <= c;
      when “11”    => y <= d;
      when others  => y <= a;
    end case;
  end process,
end architecture fourth;
```

Example 1.3 Modeling of a four-bit wide 8-1 multiplexer

A four-bit wide 8-1 multiplexer is modeled using selected signal assignment and **case** statement. The use of **case** statement instead of the **if** will make the codes easier to read. It is different from the previous example in that an integer data type is used for the select input sel which eliminates the need of the **others** clause as a default action.

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

entity mux8_1 is
  port (sel: in std_logic_vector(2 downto 0);
        a0,a1,a2,a3,a4,a5,a6,a7: in std_logic_vector(3
downto 0);
        y: out std_logic_vector(3 downto
0)); end entity mux8_1;

architecture first of mux8_1 is
begin
  with sel select
     y <= a0 when“000”,
         a1  when“001”,
         a2  when“010”,
         a3  when“011”,
         a4  when“100”,
         a5  when“101”,
         a6  when“110”,
         a7  when“111”,
         a0  whenothers;
 end architecture first;
 architecture second of mux8_1 is
 begin
   process (sel, a0, a1, a2, a3, a4, a5, a6, a7)
  begin
    case sel is
      when “000” => y <= a0;
      when “001” => y <= a1;
      when “010” => y <= a2;
      when “011” => y <= a3;
      when “100” => y <= a4;
      when “101” => y <= a5;
      when “110” => y <= a6;
      when “111” => y <= a7;
     when others => y <= a0;
    end case;
  end process;
 end architecture second;
```

Encoders

Discrete quantities of digital information, data are often represented in a coded form; binary being the most popular. Encoders are used to encode data into a coded form and decoders are used to convert it back into its original uncoded form. An encoder that has 2^n (or less) input lines encodes input data to provide n encoded output lines. The truth table for an 8-3 binary encoder (8 inputs and 3 outputs) is shown in Table 1.2. It is assumed that only one input has a value of 1 at any given time, otherwise the output has some undefined value and the circuit is meaningless.

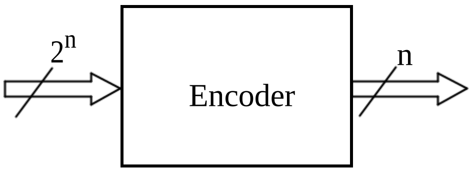

Figure 1.6 Encoder

inputs								outputs		
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Table 1.2 Truth table of a 8-3 binary encoder

All models of such a circuit must use a default “don’t care” value to minimize the synthesized circuit as only 8 of 256 (2^8) input conditions need to be specified. The synthesis tool, if capable, replaces “don’t care” values with logic 0 or logic 1 values as necessary in oreder to minimize the circuit’s logic.

Example 1.4 Modeling of a 8-3 binary encoder

An 8-3 encoder is modeled according to the truth table of Table 1.2 using conditional signal assignment, selected signal assignment, **if** and **case** statement

All models use a default assigned output value to avoid having to explicitly define all $2^8 - 8 = 248$ input conditions that should not occur under normal operating conditions. The default assignment is a “don’t care” value to minimize synthesized logic. If all 248 input conditions that are not explicitly defined are made like that they default to other value than “don’t care” (for example binary 000), more logic would be synthesized than is necessary.

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

entity encoder8_3 is
  port (a: in   std_logic_vector(7 downto 0);
        y: out std_logic_vector(2 downto
0)); end entity encoder8_3;
architecture first of encoder8_3
is begin
  y <=  “000” when a = “00000001” else
        “001” when a = “00000010” else
        “010” when a = “00000100” else
        “011” when a = “00001000” else
        “100” when a = “00010000” else
        “101” when a = “00100000” else
        “110” when a = “01000000” else
        “111” when a = “10000000” else
        “---”;
end architecture first;

 architecture second of encoder8_3 is
 begin
   with a select
     y <= “000” when “00000001”,
          “001” when “00000010”,
          “010” when “00000100”,
          “011” when “00001000”,
          “100” when “00010000”,
          “101” when “00100000”,
          “110” when “01000000”,
          “111” when “10000000”,
          “---” when others;
end architecture second;

architecture third of encoder8_3
is begin
  process (a)
  begin
    if (a = “00000001”) then y <= “000”;
    elsif (a = “00000010”) then y <= “001”;
    elsif (a = “00000100”) then y <= “010”;
    elsif (a = “00001000”) then y <= “011”;
    elsif (a = “00010000”) then y <= “100”;
    elsif (a = “00100000”) then y <= “101”;
    elsif (a = “01000000”) then y <= “110”;
    elsif (a = “10000000”) then y <= “111”;
    else y <= “---”;
    end if;
  end process;
end architecture third;
```

```
architecture fourth of encoder8_3
is begin
  process (a)
   begin
     case a is
       when “00000001” => y <= “000”;
       when “00000010” => y <= “001”;
       when “00000100” => y <= “010”;
       when “00001000” => y <= “011”;
       when “00010000” => y <= “100”;
       when “00100000” => y <= “101”;
       when “01000000” => y <= “110”;
       when “10000000” => y <= “111”;
       when others => y <= “---”;
    end case;
  end process;
end architecture fourth;
```

Decoders

Decoders are used to decode data that has been previously encoded using a binary, or possibly other, type of coded format. An n-bit code can represent up to 2^n distinct bits of coded information, so a decoder with n inputs can decode up to 2^n outputs. Various models of a 3-6 binary decoder are included in example 1.5.

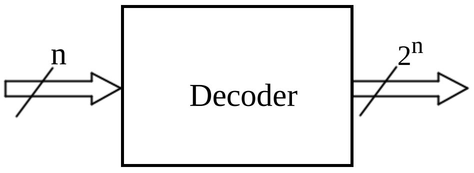

Figure 1.7 Decoder

inputs			outputs							
A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Table 1.3 Truth table of a 3-8 binary decoder

Example 1.5 Modeling of a 3-8 binary decoder

The models of the 3-8 binary decoders in this example conform to the truth table in Table 1.3. Different model versions using conditional, selected signal assignments along with typical **if**, and **case** statements.

Again, there is no write or wrong modeling technique. The **case** statement is commonly used because of its clarity, and the fact it is not a continuous assignment and so may simulate faster.

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

entity decoder3_8 is
  port (a: in   std_logic_vector(2 downto 0);
        y: out std_logic_vector(7 downto
0)); end entity decoder3_8;

architecture first of decoder3_8
is begin
       y <=  “00000001” when a = “000”
   else “00000010” when a = “001” else
```

```
        “00000100” when a = “010” else
        “00001000” when a = “011” else
        “00010000” when a = “100” else
        “00100000” when a = “101” else
        “01000000” when a = “110” else
        “10000000” when a = “111” else
        “00000001”;
end architecture first;

 architecture second of decoder3_8 is
 begin
   with a  select
     y <= “00000001” when “000”,
          “00000010” when “001”,
          “00000100” when “010”,
          “00001000” when “011”,
          “00010000” when “100”,
          “00100000” when “101”,
          “01000000” when “110”,
          “10000000” when “111”,
          “00000001” when others;
end architecture second;

architecture third of decoder3_8
is begin
  process (a)
  begin
    if (a = “000”) then y <= “00000001”;
    elsif (a = “001”) then y <=
    “00000010”; elsif (a = “010”) then y
    <= “00000100”; elsif (a = “011”) then
    y <= “00001000”; elsif (a = “100”)
    then y <= “00010000”; elsif (a =
    “101”) then y <= “00100000”; elsif (a
    = “110”) then y <= “01000000”; elsif
    (a = “111”) then y <= “10000000”; else
    y <= “00000001”; end if;
  end process;
end architecture third;

 architecture fourth of decoder3_8 is
 begin
   process (a)
   begin
     case a  is
       when “000” => y <= “00000001”;
       when “001” => y <= “00000010”;
       when “010” => y <= “00000100”;
       when “011” => y <= “00001000”;
       when “100” => y <= “00010000”;
       when “101” => y <= “00100000”;
```

```
      when “110” => y <= “01000000”;
      when “111” => y <= “10000000”;
      when others => y <= “00000001”;
    end case;
  end process;
end architecture fourth;
```

Example 1.6 Modeling of a 3-6 binary decoder with enable

Inputs				outputs					
En	A2	A1	A0	Y5	Y4	Y3	Y2	Y1	Y0
0	-	-	-	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	1	0
1	0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

- =don’t care

Table 1.4 Truth table of a 3-6 binary decoder with enable

The model version of the 3-6 binary decoder conforms to the truth table Table 1.4. This example is different from the previous one because it has a separate input enable signal and there are two unused binary values for the 3-bit input A. When the enable is inactive (En = 0), or A has an unused value, the 6-bit output must be at logic 0.

The model below uses an **if** statement to check the enable input En, separately from the enclosed **case** statement.

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

entity decoder3_6 is
  port (en: in std_logic;
        a: in std_logic_vector(2 downto 0);
        y: out std_logic_vector(5 downto 0));
end entity decoder3_6;

architecture first of decoder3_6
is begin
  process (a, en)
  begin
    if (en = ‘0’) then
      y <= “000000”;
    else
      case a is
        when “000” => y <= “000001”;
        when “001” => y <= “000010”;
        when “010” => y <= “000100”;
        when “011” => y <= “001000”;
        when “100” => y <= “010000”;
        when “101” => y <= “100000”;
        when others => y <= “000000”;
      end case;
    end if;
  end process;
end architecture first;
```

Comparators

A comparator compares two or more inputs using one, or a number of different comparisons. When the given relationship(s) is true, an output signal is given (logic 0 or logic1). Comparators are only modeled using the **if** statement with an **else** clause and no **else-if** clauses. Any two data objects are compared using equality and relational operators in the expression part of the if statement. Only two data objects can be compared at once, that is, statements like "**if** (a = b = c)" cannot be used. However, logical operators can be used to logically test the result of multiple comparisons, for example, "**if** ((a = b) and (a = c))". These equality, relational and logical operators are listed in Table 1.5.

Operators	VHDL
Equality & Relational	=
	/=
	<
	<=
	>
	>=
Logical	not
	and
	or

Table 1.5 Equality, relational and logical operators

Example 1.7 6-bit two input equality comparator:

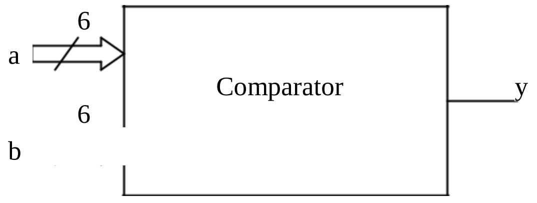

Figure 1.8 6-bit two input equality comparator

Typical comparator is shown in figure 1.8. The single bit output is at logic '1' when the two 6-bit input buses are the same, otherwise it is at logic '0'. Note that the inputs buses to be compared can't be of type std_logic_vector, they can be whether unsigned or signed types, as comparison must be preformed between signals or variables that have a defined values and since std_logic_vector does not indicate a value of the bus, so it should not be used in comparison.

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

entity simple_comparator is
  port (a, b: in unsigned(5 downto 0);
        y: out std_logic);
```

```
end entity simple_comparator;

architecture rtl of simple_comparator
is begin
  process (a, b)
  begin
    if (a = b) then
      y <= ‘1’;
    else
      y <= ‘0’;
    end if;
  end process;
end architecture rtl;
```

Example 1.8 Multiple Comparison Comparator:

Extra parentheses enclosing “c /= d **or** e >= f” means that either one of these conditions and “a = b” must be true for the output to be at logic 1.

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

entity multiple_comparator is
  port (a, b, c, d, e, f: in unsigned(2 downto 0);
        y: out std_logic);
end entity multiple_comparator;

architecture rtl of multiple_comparator
is begin
  process (a, b, c, d, e, f)
  begin
    if (a = b and (c /= d or e >=f))
      then y <= ‘1’;
    else
      y <= ‘0’;
    end if;
  end process;
end architecture rtl;
```

Latches

A latch is a level sensitive memory cell that is transparent to signals passing from the D input to Q output when enabled, and holds the value of D on Q at the time when it becomes disabled. The Q-bar output signal is always the inverse of the Q output signal, see Figure 1.9.

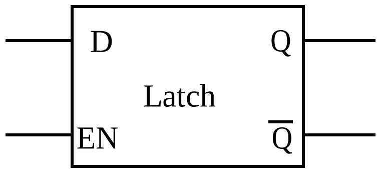

Figure 1.9 The level sensitive D-type transparent latch

inputs		outputs	
D	EN	Q+	$\overline{Q}$ +
–	0	Q-	$\overline{Q}$ -
0	1	0	1
1	1	1	0

– =don't care

Table 1.6 Characteristic table of a D-type transparent latch

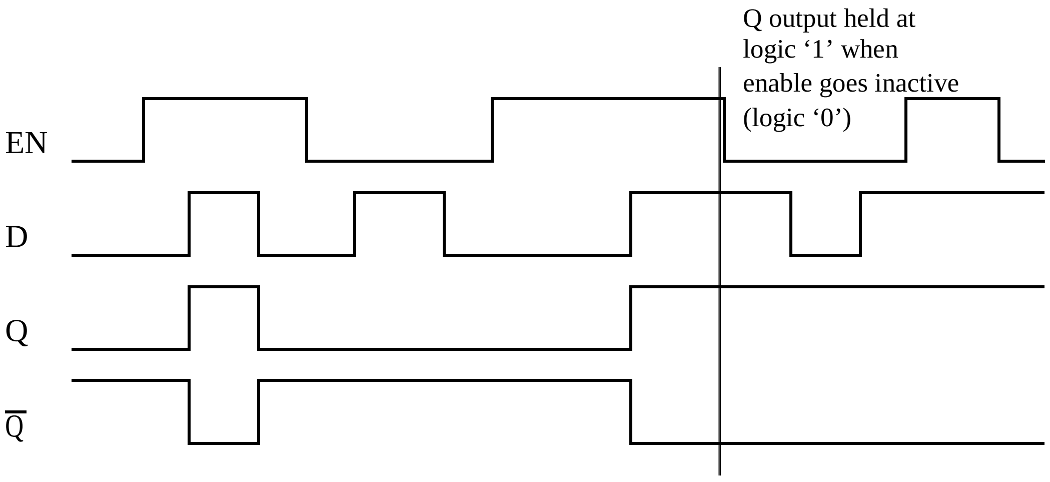
Q output held at logic '1' when enable goes inactive (logic '0')
EN
D
Q
$\overline{Q}$

Figure 1.10 Simulation of the level sensitive D-type transparent latch

Example 1.9 Modeling of a D-type transparent latch

The **if** statement without **else** clause is the simplest implementation of D-type transparent latch.

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

entity d_latch is
  port (d, en: in std_logic;
        q: out std_logic);
end entity d_latch;

architecture rtl of d_latch is
begin
  process (d, en)
  begin
    if (en = ‘1’) then
      q <= d;
    end if;
  end process;
end architecture rtl;
```

Example 1.10 Modeling latches with clear input

Latches with clear will set the output to ‘0’ whenever the clear signal goes to ‘1’. Latches with preset will preset the output to ‘1’ instead to ‘0’. Clear and preset inputs to a latch are always asynchronous with the enable. The example code above models a latch with a clear input

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

entity d_latch is
  port (d, en, clear: in std_logic;
        q: out std_logic);
end entity d_latch;

architecture rtl of d_latch is
begin
  process (d, en, clear)
  begin
    if (clear = ‘1’) then
      q <= ‘0’;
    elsif (en = ‘1’) then
      q <= d;
    end if;
  end process;
end architecture rtl;
```

D-Type Flip-Flop

The D-type flip-flop is an edge triggered memory device that transfers a signal's value on its D input, to its Q output, when an active edge transition occurs on its clock input. The output value is held until the next active clock edge. The Q-bar output signal is always the inverse of the Q output signal, see Figure 1.10.

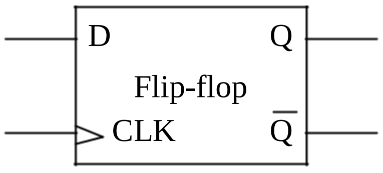

Figure 1.10 The level sensitive D-type flip-flop

inputs		outputs	
D	CLK	Q+	$\overline{Q}_+$
0	↑	Q-	$\overline{Q}_-$
1	↑		
-	0	Q-	$\overline{Q}_-$
-	1	Q-	$\overline{Q}_-$

Table 1.7 Characteristic table of a D-type flip-flop

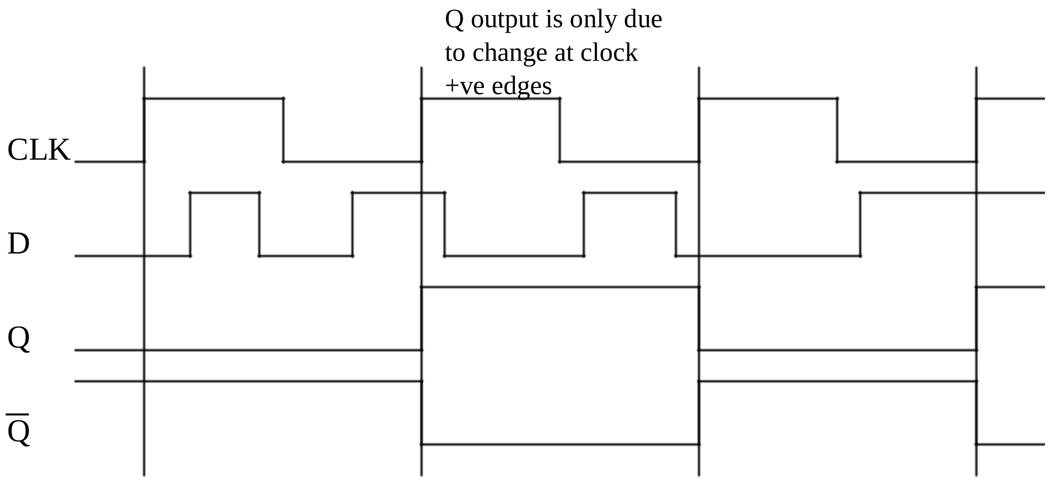

Figure 1.11 Simulation of a +ve edge sensitive D-type flip-flop

Example 1.11 Modeling of a D-type flip-flop

Different models of flip-flops with +ve and –ve edge triggered are shown in architectures below. The first and the second architectures use a VHDL attributes to detect the clk signal edge, while the third and fourth architectures use a function call for the same purpose.

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

```
entity d_flip_flop is
  port (d, clk: in std_logic;
        q: out std_logic);
end entity d_flip_flop;

architecture first of d_flip_flop
is begin
  process (clk)
  begin
    if (clk’event and clk = ‘1’)
      then q <= d;
    end if;
  end process;
end architecture first;

architecture second of d_flip_flop
is begin
  process (clk)
  begin
    if (clk’event and clk = ‘0’)
      then q <= d;
    end if;
  end process;
end architecture second;

architecture third of d_flip_flop
is begin
  process (clk)
  begin
    if (rising_edge(clk)) then
      q <= d;
    end if;
  end process;
end architecture third;

architecture fourth of d_flip_flop
is begin
  process (clk)
  begin
    if (falling_edge(clk)) then
      q <= d;
    end if;
  end process;
end architecture fourth;
```

Example 1.12 Modeling flip-flops with reset

Different flip-flops with asynchronous and synchronous resets are modeled. The first architecture models a D-type flip-flop with an asynchronous reset input, this is the commonly used D-type flip-flop, while the second architecture models D-type flip-flop with a synchronous reset as appears from the process sensitivity list and the **if** statements sequence.

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

entity d_flip_flop is
  port (d, clk, rst: in std_logic;
        q: out std_logic);
end entity d_flip_flop;

architecture first of d_flip_flop
is begin
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      q <= ‘0’;
    elsif (rising_edge(clk)) then
      q <= d;
    end if;
  end process;
end architecture first;

architecture second of d_flip_flop
is begin
  process (clk)
  begin
    if (rising_edge(clk)) then
      if (rst = ‘1’) then
        q <= ‘0’;
      else
        q <= d;
      end if;
    end if;
  end process;
end architecture second;
```

Example 1.13 Modeling flip-flop with enable and asynchronous reset

Flip-flop with enable input and asynchronous reset is modeled to illustrate the addition of the enable signal control, which is synchronous, accordingly with the asynchronous reset.

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

entity d_flip_flop is
  port (d, clk, rst, en: in std_logic;
        q: out std_logic);
end entity d_flip_flop;

architecture rtl of d_flip_flop is
begin
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      q <= ‘0’;
    elsif (rising_edge(clk)) then
      if (en = ‘1’) then
        q <= d;
      end if;
    end if;
  end process;
end architecture rtl;
```

Registers

A parallel register is simply a bank of D-type flip-flops, its implementation has the same structure as for one flip-flop but with extended inputs and outputs, see Figure 1.12.

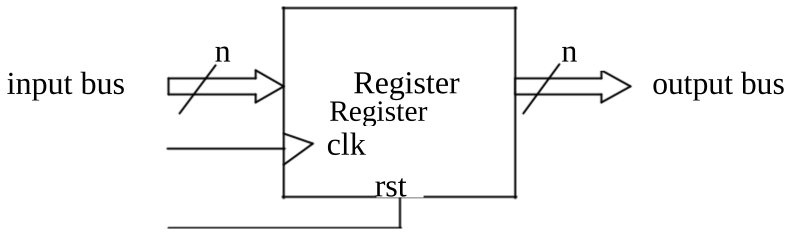

Figure 1.12 Register with n inputs and n outputs

Example 1.14 Modeling of a parallel 8 bits register

A typical example for a parallel 8 bits register is modeled, based on the flip-flop model shown previously but with inputs and outputs are extended as buses.

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

entity reg_par is
  port (clk, rst: in std_logic;
        reg_in: in std_logic_vector(7 downto 0);
        reg_out: out std_logic_vector(7 downto
0)); end entity reg_par;

architecture rtl of reg_par is
begin
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      reg_out <= “00000000”;
    elsif (rising_edge(clk)) then
      reg_out <= reg_in;
    end if;
  end process;
end architecture rtl;
```

Example 1.15 Modeling of a parallel 8 bits register with enable

The same as the previous example but with an added enable signal to provide control over the register, so that the register only register the value at its inputs when enable is active, otherwise remaining the contents unchanged. Also, an **others** statement in the reset action line is added to fill registers with any size with zeros. This model is also based on the flip-flop model shown in example 1.13.

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

entity reg_par is
  port (clk, rst, en: in std_logic;
        reg_in: in std_logic_vector(7 downto 0);
        reg_out: out std_logic_vector(7 downto
0)); end entity reg_par;

architecture rtl of reg_par is
begin
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      reg_out <= (others => ‘0’);
    elsif (rising_edge(clk)) then
      if (en = ‘1’) then
        reg_out <= reg_in;
      end if;
    end if;
  end process;
end architecture rtl;
```

Shift Registers

Shift registers are registers with added features that enable them to shift their contents in either directions. A typical parallel shift register that has added controls to achieve these features; signals shift_left and shift_ right see Figure 1.13. These signals control whether the register is to shift its contents to left or right. Depending on the usage of the register, the empty place appears after shifting is filled whether with ‘0’, or ‘1’, or the shifted bit from the other side of the register (barrel shifters), see Figure 1.14. Signals are simply a bank of D-type flip-flops, its implementation has the same structure as for one flip-flop but with extended inputs and outputs, see Example 1.16.

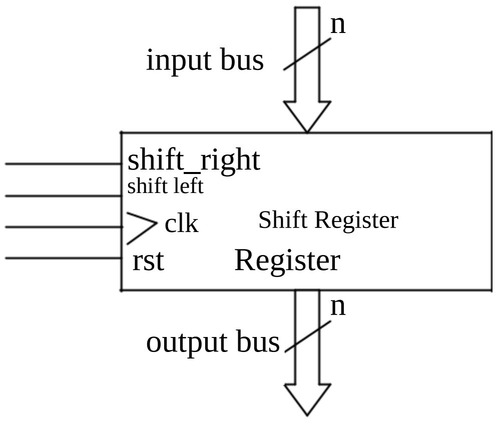

Figure 1.13 Parallel shift register with added control shift_left, and shift_right

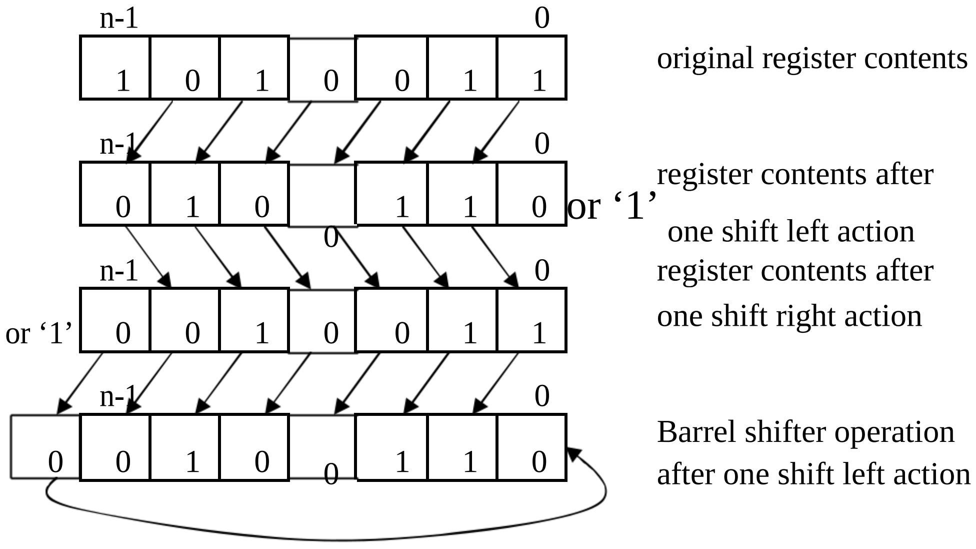

Figure 1.14 Parallel shift register operation due to shift_left, and shift_right actions

Example 1.16 Modeling of a parallel 8 bits shift register

A typical example for a parallel 8 bits shift register is modeled, added controls shift_left, and shift_right signals are combined together into one bus to facilitate the use of **case** statement afterwards. An intermediate signal definition is made to define a new bus (reg_ temp) to hold the register contents which is necessary because we cannot assign value to output and read it in the same time; (we cannot do something like that: reg_out <= reg_out because reg_out is defined as output), so we cannot read it inside (can not appear on the right hand side of any assignment). Before the end of the architecture the value of the intermediate bus is then passed to the output bus.

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

entity reg_par_shift is
  port (clk, rst: in std_logic;
        shift_right, shift_left: in std_logic;
        reg_in: in std_logic_vector(7 downto 0);
        reg_out: out std_logic_vector(7 downto
0)); end entity reg_par_shift;
architecture rtl of reg_par_shift is
  signal shift_control: std_logic_vector(1 downto 0);
  signal reg_temp: std_logic_vector(7 downto
0); begin
  shift_control <= shift_left & shift_right;
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      reg_temp <= (others => ‘0’);
    elsif (rising_edge(clk)) then
      case shift_control is
        when “00” => reg_temp <= reg_in;
        when “01” => reg_temp <= ‘0’ & reg_temp(7 downto
1);
        when “10” => reg_temp <= reg_temp(6 downto 0) &
‘0’;
        when others => reg_temp <= reg_temp;  --can be
omitted
      end case;
    end if;
  end process;
  reg_out <= reg_temp;
end architecture rtl;
```

Counters

A register that goes through a predetermined sequence of binary values (states), upon the application of input pulses on one or more inputs, is called a counter. Counters count the number of occurrences of an event, that is, input pulses that occur either randomly or at uniform intervals of time. Counters are used extensively in digital design for all kinds of applications. Apart from general purpose counting, counters can be used as clock dividers and for generating timing control signals.

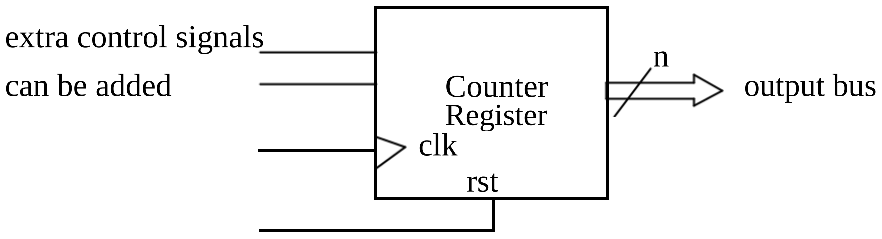

Figure 1.15 Counter with n outputs cycles over 2^n states

Example 1.17 Modeling of a 4 bits binary counter

The simplest and most common way of modeling a binary counter that can increment or decrement is by adding or subtracting a constant 1 using the "+" or "−" arithmetic operators in assignments residing in a section of code inferring synchronous logic.
An example for this implementation is shown for a 4 bits binary counter. An intermediate signal definition is made to define a new bus (count_temp) to hold the counter state which is necessary because we cannot assign value to output and read it in the same time; (we cannot do something like that: count_out <= count_out + 1 because count _out is defined as output), so we cannot read it inside (can not appear on the right hand side of any assignment), also the "+ 1" operation cannot by applied over std_logic_vector types as they do not implicitly have an equivalent value, instead an unsigned or signed types has to be used. Before the end of the architecture the value of the intermediate bus is then passed to the output bus after doing a type conversion to convert the type unsigned to std_logic_vector type used in the output bus definition in the entity declaration.

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

entity binary_count is
  port (clk, rst: in std_logic;
        count_out: out std_logic_vector(3 downto
0)); end entity binary_count;

architecture rtl of binary_count is
  signal count_temp: unsigned(3 downto 0);
begin
```

```
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      count_temp <= (others => ‘0’);
    elsif (rising_edge(clk)) then
      count_temp <= count_temp + 1;
    end if;
  end process;
  count_out <= std_logic_vector(count_temp);
end architecture rtl;
```

Example 1.18 Modeling of a 4 bits binary counter with enable

The same counter implemented before is used but with adding extra control over the counting event, so this counter only counts when the enable signal is active, otherwise the counter holds its state.

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

entity binary_count is
  port (clk, rst, enable: in std_logic;
        count_out: out std_logic_vector(3 downto
0)); end entity binary_count;
architecture rtl of binary_count is
  signal count_temp: unsigned(3 downto 0);
begin
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      count_temp <= (others => ‘0’);
    elsif (rising_edge(clk)) then
      if (enable = ‘1’) then
        count_temp <= count_temp + 1;
      else
        count_temp <= count_temp; --can be omitted
      end if;
    end if;
  end process;
  count_out <= std_logic_vector(count_temp);
end architecture rtl;
```

Example 1.19 Modeling of a counter that counts up by two and down by one

An example for a 4 bits binary counter with added control to make the counter counts up by two or down by one. The control signals count_up, and count_down are combined together into one bus to facilitate the use of **case** statement afterwards.

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

entity binary_count is
  port (clk, rst: in std_logic;
        count_up, count_down: in std_logic;
        count_out: out std_logic_vector(3 downto
0)); end entity binary_count;

architecture rtl of binary_count is
  signal count_temp: unsigned(3 downto 0);
  signal count_control: std_logic_vector(1 downto
0); begin
  count_control <= count_up & count_down;
  process (rst, clk)
  begin
    if (rst = ‘1’) then
      count_temp <= (others => ‘0’);
    elsif (rising_edge(clk)) then
      case count_control is
        when “00” => count_temp <= count_temp;
        when “01” => count_temp <= count_temp – 1;
        when “10” => count_temp <= count_temp + 2;
        when others => count_temp <= count_temp;
      end case;
    end if;
  end process;
  count_out <= std_logic_vector(count_temp);
end architecture rtl;
```