Introduction to VHDL

Solutions manual

R.D.M Hunter and T.T. Johnson

SPRINGER-SCIENCE+BUSINESS MEDIA, B.V.

Introduction to VHDL

Introduction to VHDL

Solutions manual

R.D.M. Hunter
University of Portsmouth, UK

and

T.T. Johnson
Summit Design Inc., USA

SPRINGER-SCIENCE+BUSINESS MEDIA, B.V.

ISBN 978-0-412-81340-5 ISBN 978-94-011-5838-1 (eBook)
DOI 10.1007/978-94-011-5838-1

First edition 1997

Originally published by Chapman & Hall in 1997

A catalogue record for this book is available from the British Library

Preface

This manual contains solutions to the end-of-chapter exercises for the textbook Introduction to VHDL. It is not claimed that the solutions presented here are, at all times, the best possible, or the only, solutions to the exercises; on occasion alternative solutions are offered. Where possible suitable designs have been synthesised. In a number of such cases the waveforms and schematics have been included, usually in appendices. While it is recognised that Test Benches are becoming an important issue in VHDL design these are considered beyond the scope of this volume.

N.B. The solutions to the exercises have been tested using certain proprietary tools. No guarantee is offered that the models contained herein will simulate, or synthesise, correctly on other vendors' tools.

Errata

P 76
It should not be inferred from the statement that 'IEEE.Std_logic_1164 recognises a signal type composed of 'X', '0', '1' and 'Z' ' that these are the *only* types recognised by this Standard. It should have been made clearer that the subtype X 0 1 Z would be used throughout, for simplicity, in this introductory text. However, the solutions in this manual do use the full nine-valued set where appropriate.

P 135
Line 11: App C *should read* App B

P 136
Line 9: inter_mediate : a **and** b ; *should read* inter_mediate := a **and** b ;

P 140
Wait statement not properly terminated, *should read*

```
wat on d, enable until enable = '1' ;                    -- semicolon missing
```

P 146
Architecture statement *should read*

```
architecture x of y is
     signal num, sum : integer := 0 ;  -- '=' sign was missing in signal initialisation
begin
sig_example : process ;
   begin
    wait for 10 ns ;
    num <= num + 1 ;
    sum <= sum + num ;                  -- Should be signal assignment, not variable
end process sig_example ;
end x ;
```

P 148
Process labels missing at end of processes viz.

```
                    •
begin
                    •
                    •
end process no_transport ;
                    •
begin
                    •
                    •
end process with_transport ;
```

P 152
Second line: **use** IEEE.Std_logic_1164.all ; -- Should use IEEE and not IEE

P 344
Line 12 Syntax error, *should read*

```
function "+" (in_1 :  new_state ; in_2 :  pe-op )        -- Semicolon after new_state
```

P 444
No terminating statement for component or1, *should read*

```
component or1
   port (x, y : in Std_ulogic ;
               z : out Std_ulogic) ;
end component ;
```

There are a number of other small typographical errors throughout the book which do not materially affect the reading of the text and so these have not been included among these errata.

Chapter 4

4.6 Exercises

4.6.1 Describe the purpose of
i) The Entity Declaration
ii) The Architecture Body

Solution:

i) Provides the entity with a name and defines the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus an entity declaration can, potentially, represent a class of design entities, each with the same interface. The formal representation is as follows:

```
entity_declaration ::=
   entity identifier is
         entity_header
         entity_declarative_part
  [ begin
         entity_statement_part ]
   end [ entity ] [ entity_simple_name ] ;
```

The entity simple name, if present, must repeat the identifier name. The entity statement part, if present, consists of concurrent statements that are present in each such design entity.

The entity header declares objects used for communication between a design entity and its environment.

```
entity_header ::=
   [ formal_generic_clause ] — see Chapter 16
   [ formal_port_clause ]
generic_clause ::=
   generic ( generic_list ) ;
port_clause ::=
   port ( port_list ) ;
```

In Section 4.5.1 the comment is made that an entity declaration without a **port** clause has no way of accepting data from outside itself and no way of transmitting data beyond itself. While this is true there is an exception with respect to the generation of test benches. In these cases there are no port

Solution:

i) nand gate is an illegal identifier, should be written as nand_gate
ii) Character ; after b in **port** statement should be a colon :
iii) Character missing after **in** BIT should be a semi-colon ;
iv) Character ; after c in **port** statement should be a colon :
v) entity not properly terminated should have **end** nand_gate ;
vi) illegal identifier nand gate repeated in architecture statement
vii) Operator (<=) illegal after first **else**.
viii) Must end signal assignment activity statement with an **else** i.e., in this case, **else** '1'
Note : Activity statement must end with ; (before the **end** statement).

4.6.3 Produce a design entity for a 3-input and gate and include commenting code where appropriate.

Solution:

```
entity and_gate is                                    -- identifies entity
   port (a, b, c : in BIT ;                           -- declares input ports
                 d : out BIT ) ;                      -- declares output port
end and_gate ;                                        -- ends entity declaration

architecture data_flow of and_gate is                 -- names architecture body
begin
   d <= '1' when a = '1' and b = '1' and c = '1' else '0' ;   -- dataflow description of
                                                              -- and_gate
end data_flow ;                                       -- ends architecture body statement
```

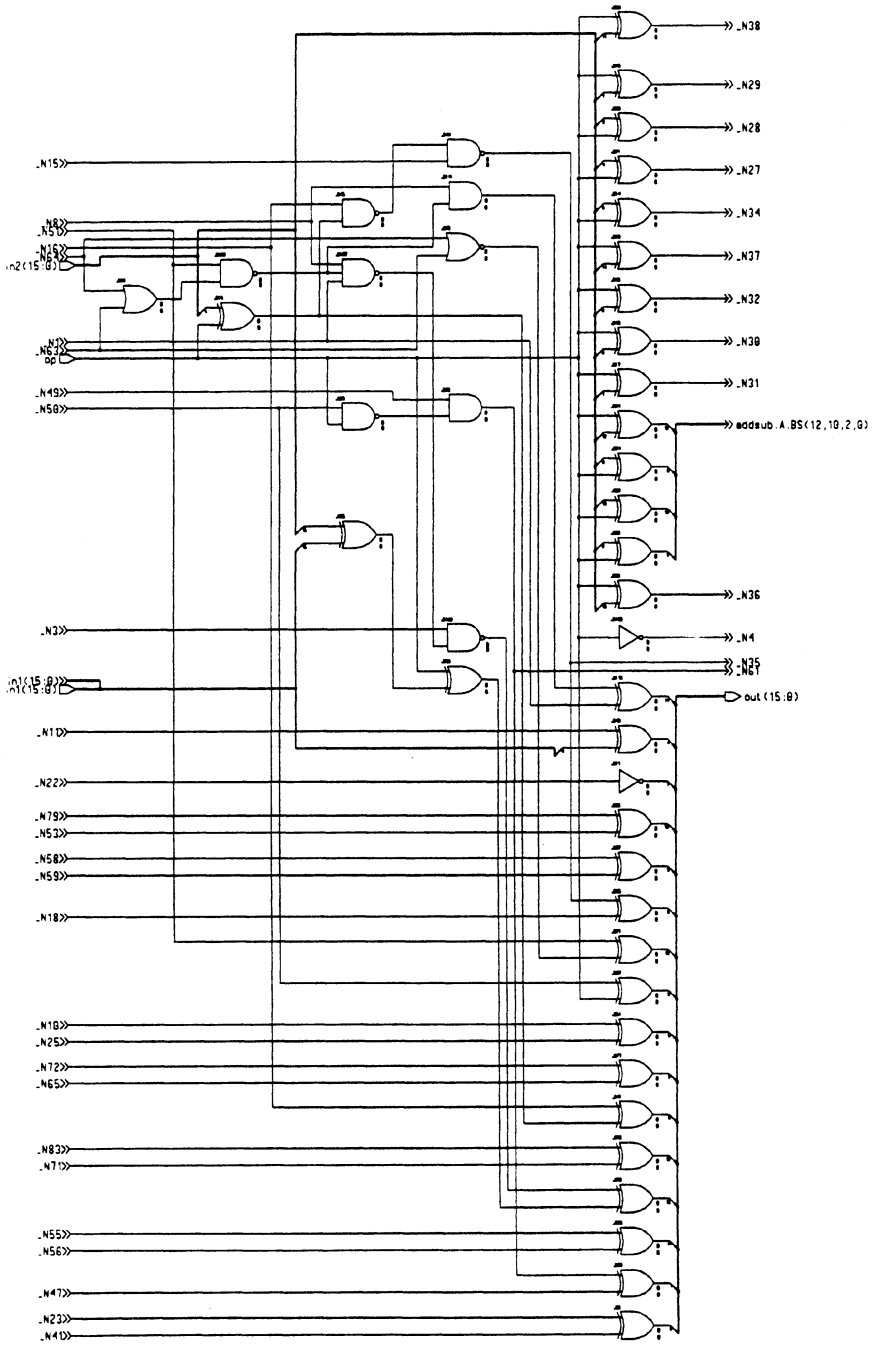
4.6.4 Produce a model for a '2-to-4' decoder.

Solution:

```
entity two_to_four_decoder is
   port ( a, b : in BIT ;
          y0, y1, y2, y3 : out BIT ) ;
end two_to_four_decoder ;

architecture data_flow of two_to_four_decoder is
begin
   y0 <= '1' when a = '0' and b = '0' else '0' ;
   y1 <= '1' when a = '0' and b = '1' else '0' ;
   y2 <= '1' when a = '1' and b = '0' else '0' ;
   y3 <= '1' when a = '1' and b = '1' else '0' ;
end data_flow ;
```

It may be noted at this point that the above implementation is not the only, or necessarily the best, way that the code for the decoder can be written. For example, use could have been made of bit vectors (see Chapter 5) or the **case** statement see Chapter 9).



```
entity sequence_detector is
   port (x, clk, reset : in BIT ;
                         y : out BIT ) ;
end sequence_detector ;

architecture synthesised of sequence_detector is
   type states is ( state0, state1, state10, state100, state 1001 ) ;
   signal current_state, next_state : states ;
begin
clock :process (clk, reset)
begin
   if reset = '0' then
         current_state <= state0 ;
   elsif clk'EVENT and clk'LAST_VALUE = '0' and clk = '1' then
         current_state <= next_state ;
   end if ;
end process clock ;

fsm : process (current_state, x)
begin
   y <= '0' ;
   case current_state is
   when state0 =>
         if x = '0' then
               next_state <= current_state ;
         else
               next_state <= state1 ;
         end if ;
   when state1 =>
         if x = '0 then
               next_state <= state10 ;
         else
               next_state <= current_state ;
         end if ;
   when state10 =>
         if x = '0' then
               next_state <= 100 ;
         else
               next_state <= state1 ;
         end if ;
   when state100 =>
         if x = '0' then
               next_state <= state0 ;
         else
               next_state <= state1001 ;
         end if ;
```

```
-- switch register without XOR--

ENTITY switch_register IS
        GENERIC                 (
                                reset_level         : Std_ulogic:='1'
                                );
        PORT            (
                                input_data,input_reset : IN Std_ulogic;
                                clock,reset         : IN Std_ulogic;
                                output    : OUT Std_ulogic
                                );
END switch_register;

ARCHITECTURE switch_register_arch OF switch_register IS
CONSTANT    clock_edge      : Std_ulogic:='1';
SIGNAL      memory : Std_ulogic:='0';
BEGIN
        PROCESS (clock,reset)
        BEGIN
                IF (clock=clock_edge and clock'EVENT) THEN
                        IF (reset=reset_level) THEN
                memory<=input_reset;
                        ELSE memory<=input_data;
                        END IF;
                END IF;
        END PROCESS;
        output<=memory;
END switch_register_arch;


--------------------
-- n bit register --
--------------------

ENTITY shift_register IS
        GENERIC                 (
                                reset_level                 :Std_ulogic:='1';
                                reg_stages                  :integer:=2#1011#
                                );
        PORT                    (
                                input,clock,reset  :IN Std_ulogic;
                                output             :OUT Std_ulogic
                                );
END shift_register;

ARCHITECTURE shift_register_arch OF shift_register IS
COMPONENT register_cell
        GENERIC                 (
                                reset_level         :Std_ulogic
                                );
        PORT            (
                                input,clock,reset  :IN Std_ulogic;
                                output             :OUT Std_ulogic
                                );
END COMPONENT;
```