**VHDL**

Table of Contents

[Input and Output Specifications 4](#_Toc64543200)

[Architecture 5](#_Toc64543201)

[Making Code Sequential 6](#_Toc64543202)

[Test Benches 7](#_Toc64543203)

[Typecasting 10](#_Toc64543204)

[Compositive Type Declaration 11](#_Toc64543205)

[Single Port RAM 12](#_Toc64543206)

[Seven Segment Display 14](#_Toc64543207)

VHDL stands for VHSIC Hardware Description Language, and VHSIC stands for Very High-Speed Integrated Circuit. It is a hardware description language. For example, say we have a field-programmable array (FPGA). We could upload some code to the FPGA chip. Unlike other ‘fixed’ hardware that only allows specific objectives that the manufacturer has designed into the chip, FPGA uses the code along with any input we give using switches to give us a customized output.

The following is a sample VHDL code:

library ieee;  
use ieee.std\_logic\_1164.all;  
use ieee.std\_logic\_signed.all;  
use ieee.numeric\_std.all;  
  
entity HALF\_ADDER is  
 port(  
 A : in std\_logic;  
 B : in std\_logic;  
 sum : out std\_logic;  
 carry: out std\_logic  
 );  
end HALF\_ADDER;  
  
architecture behavorial of HALF\_ADDER is  
begin  
 SUM <= A XOR B;  
 CARRY <= A AND B;  
end behavorial;

VHDL

This is quite relatable to the programming code we use normally. First, we have some libraries being imported. The first two libraries are always used with every VHDL code. Next, we have an entity being declared, a half-adder. In the entity section, we declare all the ports we will be using, which are the inputs and outputs. Finally, in the architecture section, we declare how the entity should be used. There could also be a configuration section at the end that is not shown here.

VHDL is used to code models for digital systems. Modelling is done for

* Documentation
* Testing via simulations
* Synthesis
* Class Assignments

The goal is to find the most reliable design process with the minimum cost and time consumption and also avoid design errors.

The difference between VHDL code and other software languages is that VHDL is not executed sequentially. Instead, every line in the code is executed at once, in parallel. Thus, it is called concurrent statements. This is similar to how the actual hardware would work, since every part of a circuit gets power at the same time, unless we specifically design it to not do that.

## Input and Output Specifications

We previously saw that when we declare an entity, we need to declare what our inputs and outputs will be. Revisiting the code, it looks a little like this:

entity HALF\_ADDER is  
 port(  
 A : in std\_logic;  
 B : in std\_logic;  
 sum : out std\_logic;  
 carry: out std\_logic  
 );  
end HALF\_ADDER;

VHDL

Notice that the inputs are given a type, std\_logic. This is part of the std\_logic\_1164 package and allows a total of different values. Alternatively, bit could have been used, which only allows a single value, or .

## Architecture

Just specifying the inputs and outputs is not enough. We also need to specify the actual circuit. This is done in the architecture section.

architecture Behavorial of HALF\_ADDER is  
begin  
 SUM <= A XOR B;  
 CARRY <= A AND B;  
end Behavorial;

VHDL

## Making Code Sequential

It is possible to force our VHDL code to behave in a sequential manner. This is done by using a ‘process’ in the architecture section.

architecture Behavorial of HALF\_ADDER is  
begin  
 process(A, B)  
 begin  
 SUM <= A XOR B;  
 CARRY <= A AND B;  
 end process;  
end Behavorial;

VHDL

The process section is given two parameters, A and B. Essentially, this means that this code will only run when both of these parameters are available.

## Test Benches

Test benches are programs that verify our VHDL code. It checks whether or not the hardware model we created does what it is supposed to do. These tests are run before uploading the code to an FPGA for example.

The main work of test benches is to generate some input and compare the outputs given by the code for that input against the expected output for the code.

library ieee;  
use ieee.std\_logic\_1164.all;  
use ieee.numeric\_std.all;  
  
entity tb\_HALF\_ADDER is  
end entity tb\_HALF\_ADDER;  
  
architecture behavorial of tb\_HALF\_ADDER is  
  
 -- Component Declaration for the Unit Under Test (UUT)  
 component HALF\_ADDER  
 port(  
 A : in std\_logic;  
 B : in std\_logic;  
 sum : out std\_logic;  
 carry: out std\_logic  
 );  
 end component;  
   
 signal A\_tb, B\_tb : std\_logic; -- inputs  
 signal sum\_tb, carry\_tb : std\_logic; -- outputs  
  
begin  
 uut: HALF\_ADDER port map(  
 A => A\_tb,  
 B => B\_tb,  
 sum => sum\_tb,  
 carry => carry\_tb;  
 );

-- stimulus process  
 stim\_process: process  
 begin  
 wait for 50 ns;  
 A\_tb <= '0'; B\_tb <= '0';  
 wait for 50 ns;  
   
 A\_tb <= '0'; B\_tb <= '1';  
 wait for 50 ns;  
   
 A\_tb <= '1'; B\_tb <= '0';  
 wait for 50 ns;  
   
 A\_tb <= '1'; B\_tb <= '1';  
 wait for 50 ns;  
   
 A\_tb <= '0'; B\_tb <= '1';  
 wait for 50 ns;  
   
 A\_tb <= '1'; B\_tb <= '1';  
 wait for 50 ns;

VHDL

The test bench code is somewhat similar to the VHDL code we write.

* The entity section is left empty this time, since we are not introducing anything new.
* In the architecture section, a component is added. This is the actual component that the entity we created will be tested against.
* Next, we have input and output signals that will be provided to the test bench. We need to map each of the inputs to the entity we created in our actual code to one of these signals.
* Finally, stimuli are provided and compared to the accurate results.

Oftentimes, we want to be able to specify a property separately for each instance of a component, such as the delay or bit width. VHDL allows us to create models with generic parameters. We can make a general model instead of specific ones with many different configurations of inputs, outputs and timing information. The required information is passed into the design description from its environment.

entity NAND\_GATE is

generic (

N: Natural := 2;  
 D: Time := 10 ns

);

port (

A: in Bit\_Vector(1 to N);  
 Z: out Bit

);  
end NAND\_GATE;

VHDL

## Typecasting

Say we are converting from std\_logic\_vector to integer. The first thing we need to think about is the data represented by std\_logic\_vector. Is it signed or unsigned? Signed data can be positive or negative, but unsigned can only be positive. Depending on this, we might need to use the unsigned() or the signed() typecast. Once this is done, it can be converted to integer.

signal input\_4 : std\_logic\_vector(3 downto 0);  
signal output\_4a : integer;  
signal output\_4b : integer;  
-- This line demonstrates the unsigned case  
output\_4a <= to\_integer(unsigned(input\_4));  
-- This line demonstrates the signed case  
output\_4b <= to\_integer(signed(input\_4));

VHDL

## Compositive Type Declaration

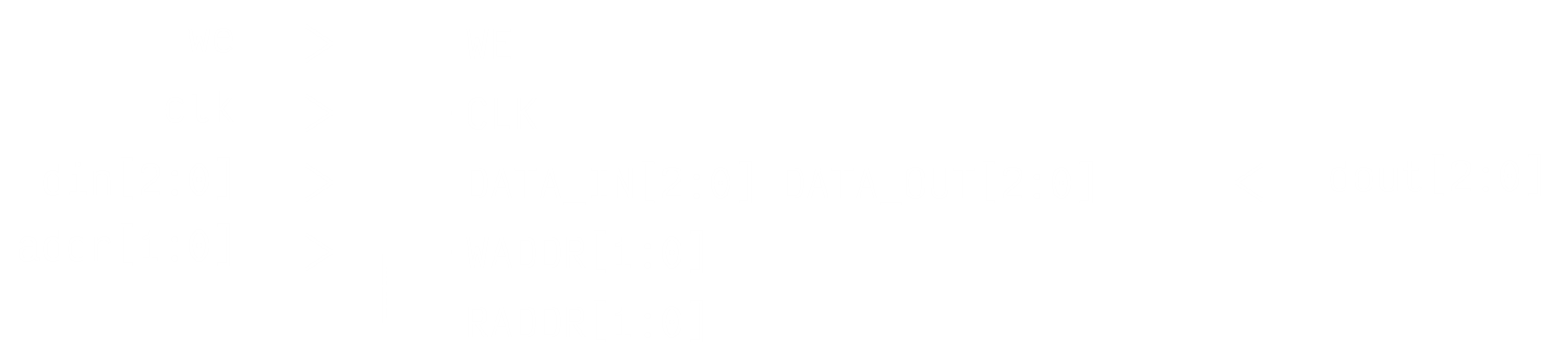
We can declare a type for creating arrays, records or unit objects.

type **word** is array(0 to 31) of bit;  
type **data** is array(7 downto 0) of word;  
type **mem** is array(natural range <>) of word;  
type **matrix** is array(integer range <>, integer range <>) of real;

VHDL

## Single Port RAM

Single port RAM has one input port (address line) which is used for both storing and retrieving data, as shown in the figure below. Here, addr[1:0] is used to both read and write.



We can create such a RAM using VHDL.

library ieee;  
use ieee.std\_logic\_1164.all;  
use ieee.numeric\_std.all;  
   
entity single\_port\_RAM is  
 generic (  
 addr\_width : integer := 2;  
 data\_width : integer := 3  
 );

port(  
 clk: in std\_logic;  
 we : in std\_logic;  
 addr : in std\_logic\_vector(addr\_width - 1 downto 0);  
 din : in std\_logic\_vector(data\_width - 1 downto 0);  
 dout : out std\_logic\_vector(data\_width - 1 downto 0)  
 );  
end single\_port\_RAM;  
  
architecture behavioral of single\_port\_RAM is  
 type **ram\_type** is array((2 \*\* addr\_width) - 1 downto 0)

of std\_logic\_vector(data\_width - 1 downto 0);  
 signal ram\_single\_port : **ram\_type**;

begin  
 process(clk)  
 begin  
 if (clk\_event and clk = '1') then  
 if (we = '1') then -- write data to address 'addr'  
 --convert 'addr' type to integer from std\_logic\_vector  
 ram\_single\_port(to\_integer(unsigned(addr))) <= din;  
 end if;  
 end if;  
 end process;  
  
 -- read data from address 'addr'  
 -- convert 'addr' type to integer from std\_logic\_vector  
 dout <= ram\_single\_port(to\_integer(unsigned(addr)));  
end architecture behavioral;

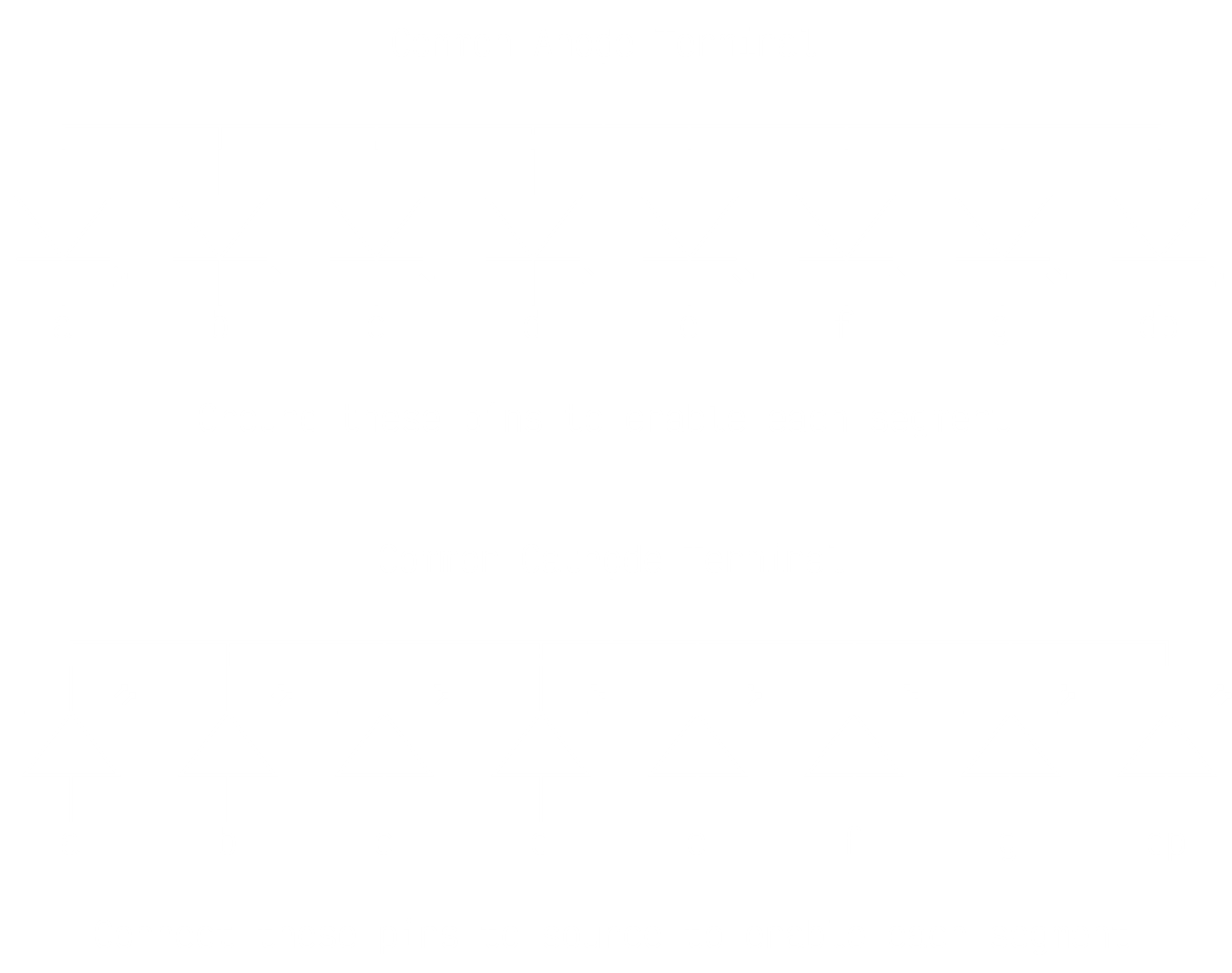
VHDL

## Seven Segment Display

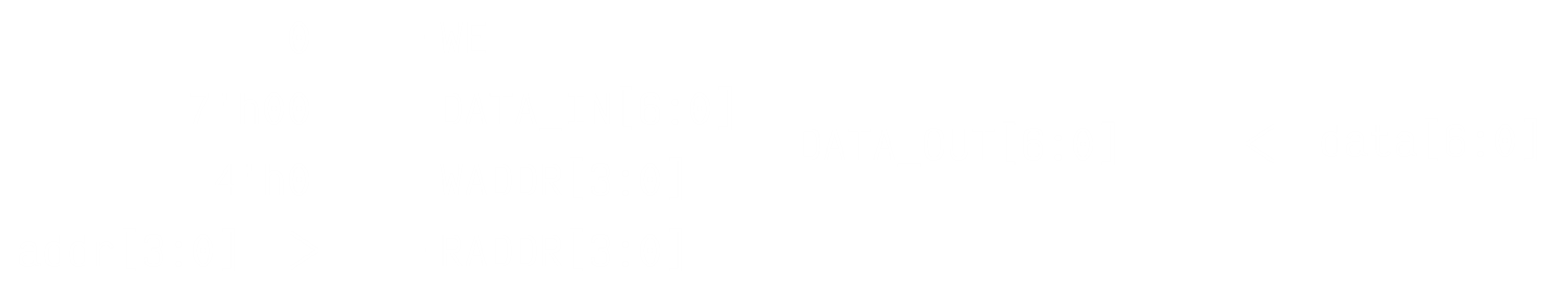
A seven-segment display is a set of seven bar-shaped LED or LCD elements arranged to form a squared-off figure 8. A few displayed use other devices like incandescent or neon lamps. A single digit number between 0 and 9 can be displayed by activating the elements in different combinations. Alphabets can also display, but that would require more elements in different arrangements.

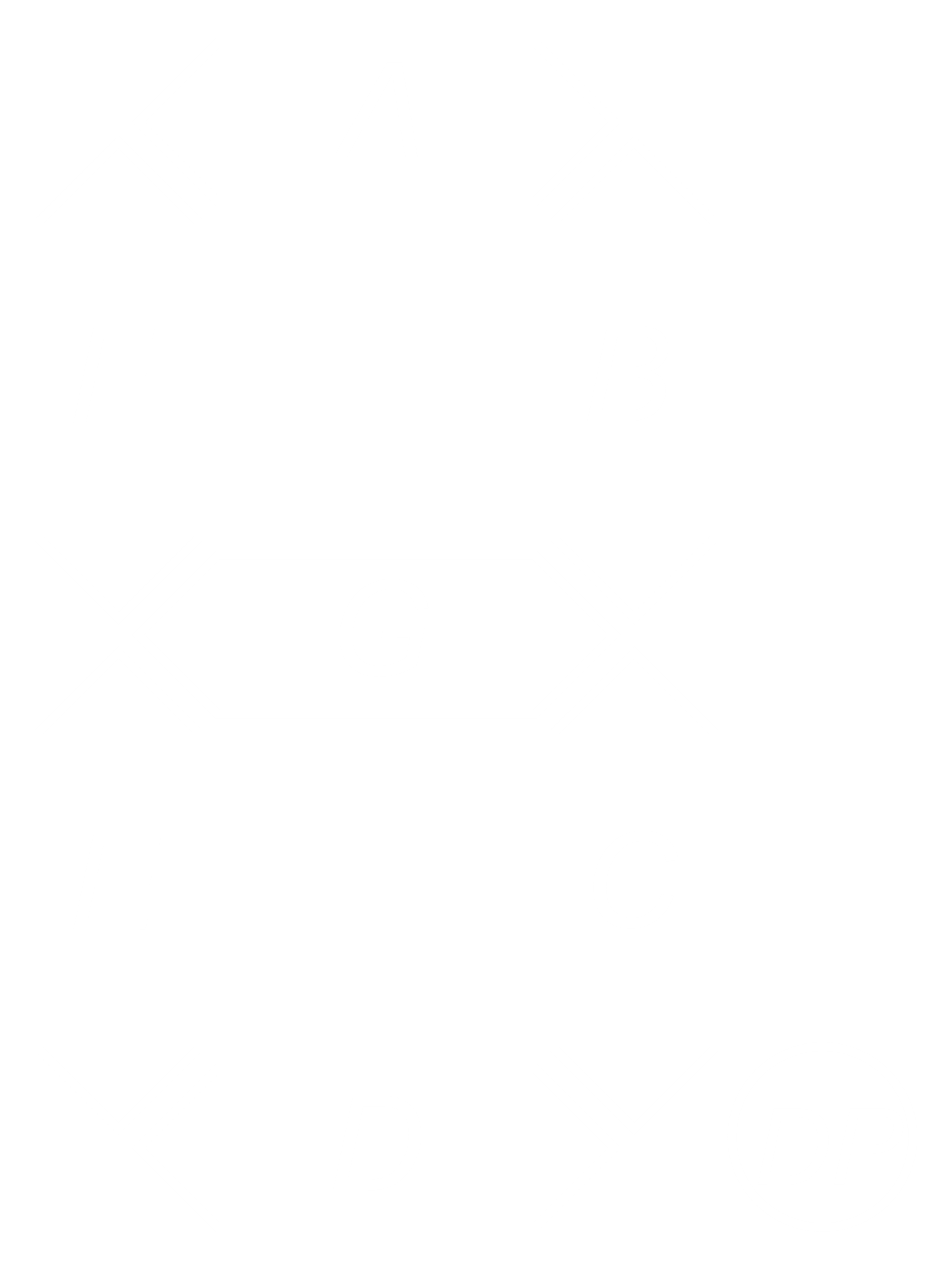
Seven segment displays are used in a wide range of devices like digital clocks, timers, wrist watches, calculators etc.

For each element, either the anode or the cathode is connected to a common terminal, while the other is left free. If the anode terminals are connected, it is called a common anode terminal and if the cathode terminals are connected, it is called a common cathode terminal. In order to make an element glow, the cathode has to be connected to the login and the anode has to be connected to the logic .



We can create a seven-segment display using VHDL.





|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Segment Inputs | | | | | | | 7 Segment  Display Output |
| a | b | c | d | e | f | g |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 3 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 5 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 6 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 9 |

library ieee;  
use ieee.std\_logic\_1164.all;  
use ieee.numeric\_std.all;  
  
entity ROM\_sevenSegment is  
 generic(  
 addr\_width : integer := 16; -- store 16 elements  
 addr\_bits : integer := 4; -- required bits to store 16 elements  
 data\_width : integer := 7 -- each element has 7-bits  
 );  
  
 port(  
 addr : in std\_logic\_vector(addr\_bits - 1 downto 0);  
 data : out std\_logic\_vector(data\_width - 1 downto 0)  
 );  
end ROM\_sevenSegment;

architecture behavioral of ROM\_sevenSegment is  
 type **rom\_type** is array(0 to addr\_width - 1)

of std\_logic\_vector(data\_width - 1 downto 0);

signal sevenSegment\_ROM : **rom\_type** := (  
 "1000000", -- 0, active low i.e. 0:display & 1:no display  
 "1111001", -- 1  
 "0100100", -- 2  
 "0110000", -- 3  
 "0011001", -- 4  
 "0010010", -- 5  
 "0000010", -- 6  
 "1111000", -- 7  
 "0000000", -- 8  
 "0010000", -- 9  
 "0001000", -- a  
 "0000011", -- b  
 "1000110", -- c  
 "0100001", -- d  
 "0000110", -- e  
 "0001110" -- f  
 );

begin  
 data <= sevenSegment\_ROM(to\_integer(unsigned(addr)));  
end architecture behavioral;

VHDL