Goals of this Tutorial:

① The main objectives of this tutorial is to introduce the students to

ⓐ HDL (Hardware Descriptive Languages)

ⓑ VHDL VHSIC (very high speed integrated circuit)
Hardware
Descriptive
Language

ⓒ How to describe your designs using VHDL.

ⓓ why use VHDL as an alternative to schematic capture

ⓔ syntax of VHDL

ⓕ History of VHDL

ⓖ Hierarchical Design

ⓗ synthesis

VHDL important modeling Techniques

1. Abstraction
2. Modularity
3. Hierarchy

- **Abstraction**: is basically hiding of details
  allows for the description of different parts of a system with different amount of detail
  "Modules which are needed only for simulation do not have to be described as detailed as modules that might be synthesized"

- **Modularity**: enables the designer to split big functional blocks and to write a model for each part

- **Hierarchy**: lets the designer build a design out of submodules which may consist of several submodules themselves

  Note: Each level of hierarchy may contain modules of different abstraction levels

* Hardware Descriptive Languages:

HDLs are used to describe hardware for the purpose of ① simulation ② modeling, ③ testing ④ Design and ⑤ Documentation.

"These languages provide a convenient and compact format for the hierarchical representation of functional and wiring details of digital systems"

Some HDLs consist of a simple set of symbols and notations that replace schematic diagrams of Digital circuits

Others are more formally defined and may present the hardware at one or more levels of abstraction.

Examples: Verilog - VHDL - UDL - M

- Verilog → Cadence Design Systems
- VHDL → IEEE Standard
- UDL → Japanese standard
- M → Mentor Graphic

What are available HDL's

1. ABEL (Advanced Boolean Equation Language)
   trademark of Data I/O corp

   was invented to allow designers to specify logic functions for realization in
   pLDs (programmable logic Devices)

2. Verilog (maybe an acronym to VERIfy LOGic)
   introduced by Gateway Design Automation in 1984
   Now owned by Cadence design system
   roots $\longrightarrow$ C

3. VHDL $\longrightarrow$ roots ADA

History:

1. VHDL was proposed as an HDL in 1981
2. in 1986 VHDL was proposed as an IEEE standard.
3. It went through a number of revisions and changes until it was adopted as the IEEE 1076 standard in Dec 1987.
4. A newer revision of the language referred to as VHDL'93 $\longrightarrow$ in the process of replacing VHDL'87

VHSIC program

① VHDL is an offshoot of the "very high speed IC" (VHSIC) program that was funded by DARPA in the late 1970s

* The goal was to produce the next generation of integrated circuits.

" Designers found out that the tools used to create large designs were inadequate for the task!! "

Initially, VHDL was designed to be
① Documentation Language
② simulation Language.

But later was embraced by synthesis tools to describe hardware design

Advantages

① could handle complex circuits
② standard, so all players in VHSIC program could distribute designs to other players in a standard form.

1. VHDL is a language, just as C and Java are languages.

   VHDL is used to describe, model, and synthesize a circuit

   just as C is used to describe, model and implement a solution to a problem.

2. Like C, VHDL supports Libraries

3. VHDL allows us to create modular designs so that we can take advantage of Hierarchical design

   (i.e building a big, complex circuit From a bunch of smaller simpler circuits.

4. Like Java, VHDL is Device Independent

   i.e we can design a circuit before we know which type of device it will be implemented on.

Concurrent vs sequential statements

* In a typical programming language such as C or pascal each assignment statement executed one after the other in a specified order.
* The order of execution is determined by the order of the statements in the source file.

VHDL

Concurrent (occur in parallel)

The order of execution is solely specified by events occurring on signals

```
q <= NOT (qb AND set) After 2ns
qb <= NOT (q AND reset) After 2ns
```

These statements are executed whenever

① qb and set have an event

② q and reset have an event

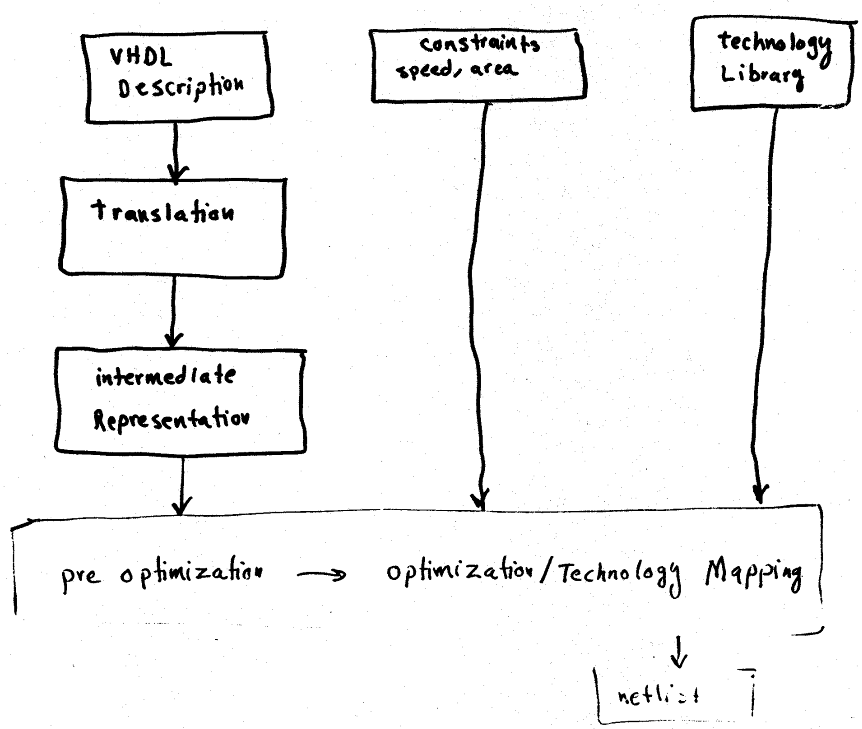
Sequential

The order of execution is sequential i.e execute one afte the other.

Once you have designed a circuit using VHDL, there are two main tasks that you can accomplish

- circuit simulation
- synthesis

High level Flow for logic synthesis



Using VHDL in the Design process

In general, there is a recipe of steps to follow when designing a circuit. These steps can be described as:

1. Define the design requirements.
2. Define (code) the design in VHDL; → structural* → Behavioral* → Data flow*
3. Simulate the VHDL "source" code;
4. Synthesize the design; (optimize)
5. Fit the design into a given device architecture (program the device for FPGA)

or

Implement the design on a chip "ASIC Design"
Application specific Integrated circuit

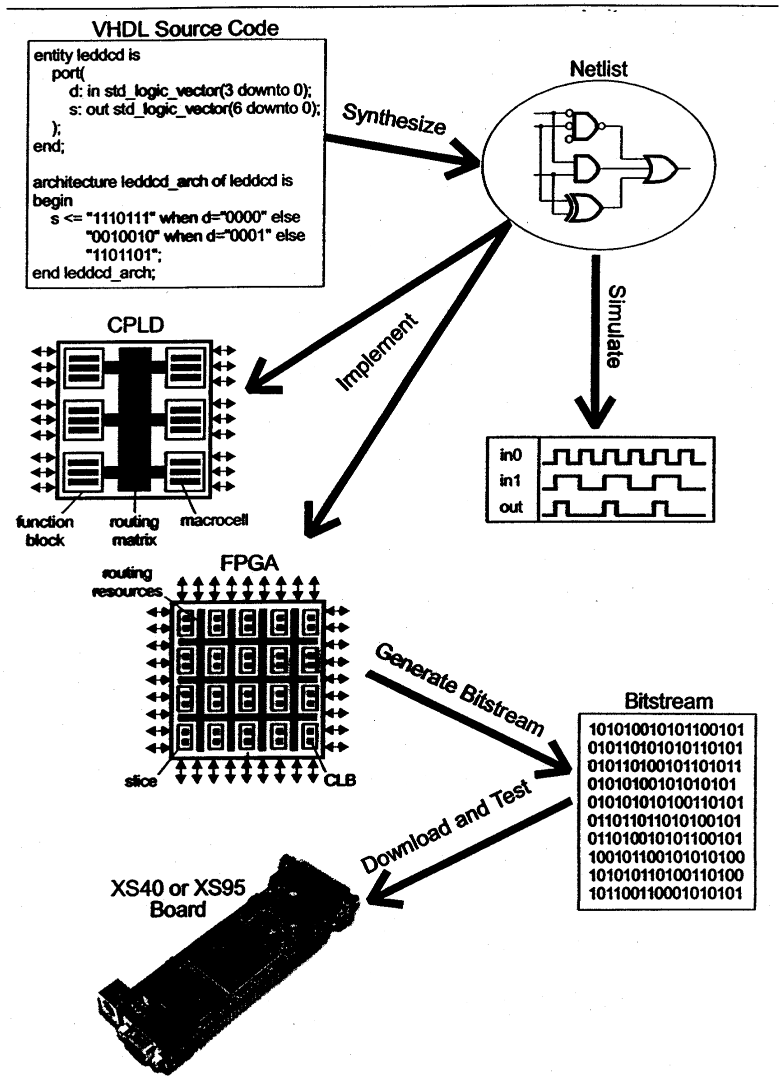


Figure 1: Steps in creating and testing an FPGA or CPLD-based design.

Using VHDL in the design process

In general there is a sequence of steps to follow when designing a circuit.

1. Define the design requirements
2. Define (code) the design in VHDL
3. simulate the VHDL "source" code
4. synthesize the design → place → route
5. Fit the design into a given device architecture
6. program the device.

Data types

- Every data object in VHDL can hold a value that belongs to a set of values
- This set of values is specified using a type declaration.

predefined types

facility to define new types

package standard

1. Boolean "false, true"
2. BIT (0, 1)
3. character " Ascii "
4. INTEGER $-(2^{31}-1)$ - $+(2^{31}-1)$
5. REAL
6. TIME hr = 60 min , min = 60 sec, ms, μs, ns, ps
7. STRING
8. BIT_vector → array of bit

Subtypes

- is a type with a constraint
- the constraint specifies the subset of values

Example

```
subtype MY_INTEGER is INTEGER range 48 to 156;

type DIGIT is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
subtype MIDDLE is DIGIT range '3' to '7';
```

* All the possible types that can exist in the language can be categorized into

① scalar → Enumeration, integer, physical, floating point

type MVL is ('U', '0', '1', 'Z');

f.p → type TTL-VoLtage is range -5.5 to -1.4;

② composite (i.e a collection of values)

- array types type ADDRESS_WORD is array (0 to 63) of Bi:

③ Access type → similar to pointers in C

④ File type

Data objects

- A data object holds a value of a specified type.
- It is created by means of an object declaration

Example   Variable COUNT : INTEGER

Every data object belongs to one of 4 classes

1. constant ⟶ Value cannot change
   Similar to C → Constant R_T : Time := 10ns;

2. Variable ⟶ different values can be assigned to the variable at different times
   Similar to C ⟶ Variable Done : Boolean

3. Signal ⟶ holds a list of values, which include the current value of the signal, and a set of possible future values
   Signals can be regarded as wires in a circuit
   Signal clock : Bit

4. File ⟶ Values can be read or written to the file using read procedures and write procedures.

Signals

* In digital circuits we concern ourselves with signals, and likewise in VHDL.

* So to accomodate all the possible signal values that could occur in a digital circuit, the IEEE approved a 9 value system.

* This value system is termed the IEEE 1164 standa

Value	Meaning
U	Uninitialized ✓
X	Forcing unknown ✓
0	Forcing 0 ✓
1	Forcing 1 ✓
Z ✓	High impedance ✓
W ✓	weak unknown
L ✓	weak 0
H ✓	weak 1
- ✓	Don't care

std_logic, std_logic_vector can take any of the values above

e.g trying to drive a signal to both '0', '1' → X

Types & constants

- All signals, variables, and constants in a VHDL program must have an associated "type"
- The type specifies the set or range of values that the object can take on

imp VHDL has just a few predefined types

bit	character	string
bit-vector	integer	time
boolean	real	

Bit, Bit-vector → '0, 1

Boolean, → true, false

character → ascii

integer → $-2^{31}+1$ through $+2^{31}-1$

Other ways to Declare objects

(imp) NOT all objects in a VHDL description are explicitly created using object declarations

1. ports of an entity. All ports are signal objects
2. Generics of an entity. These are constant objec
3. Formal parameters of functions and procedure

There are two other types of objects that are implicitly declared.

- indices of a $\boxed{\text{for .... loop}}$ statement
- indices of the $\boxed{\text{generate}}$ statement

e.g

```
for count in 1 to 100 loop
    sum := sum + count;
End loop;
```

here count is a constant that has implicit declaration of type integer (i.e cannot be explicitly declared)

Identifiers

two types

- Basic identifier
- Extended

Basic identifier

- The first character must be a letter — D345_high ✓
- last character may not be an underscore — RAM_address ✓
- case insensitive — count ≈ COUNT ✓
- Two underscore characters cannot appear consecutively.

Imp Comments in a VHDL line start with a hyphens

-- This is a line of comment

Extended identifier

is a sequence of characters written between two backslashes e.g \TEST\ \test\ \7last\

- \TEST\ is different than \test\
- \7400\ is legal

IMP The language defines a set of reserved words (also called keywords) that cannot be used as basic identifiers.!!

Operators

The predefined operators in the language are classified into the following six categories

1. logical ② Relational ③ shift ④ Adding
⑤ Multiplying ⑥ Miscellaneous

① logical → and, or, nand, nor, xor, xnor, not
↳ defined for BIT - BOOLEAN

② Relational → = /= < <= > >=
↳ the result type is always of type Boolean

③ shift → $\underbrace{\text{sll, srl}}_{\text{logical}}$, $\underbrace{\text{sla, sra}}_{\text{arithmetic}}$, $\underbrace{\text{rol, ror}}_{\text{rotate}}$
↳ takes an array of BIT or BOOLEAN

i.e "1001010" srl 3 is "0001001"

④ Adding → + - &
↳ concatenation

" & "
A := 5
B := 3
C := A+B

⑤ Multiplying → * / mod rem

⑥ Misc → abs **
↳ exponentiation

libraries & packages

- with any large body of software we need mechanisms for structuring programs, reusing software modules

- In conventional languages, these mechanisms have been available to us for some time.

- VHDL also provides support for such mechanisms through the concept of packages and libraries for sharing large bodies of code.

package { "Related groups of functions and procedures can be aggregated into a module that can be shared across many different VHDL models

can be placed in libraries (libraries are repositories for design units in general, and packages are one type of design unit.

In our case we will regurely declare

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

std_logic → represent standard logic

it specifies the values that may appear on inputs & outputs

→ among its nine values, includes

binary values 0, 1 X → unknown

U → undefined

to use std_logic, it is necessary to define values and operations

* For convenience, a <u>package</u> consisting of precompiled VHDL code is employed

* packages are usually stored in a directory referred to as <u>library</u>

So basic package is ieee.std_logic_1164.all

use i.eee.std_logic_1164.all

In order to use std_logic we include line

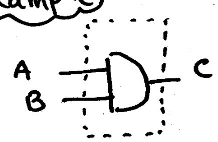
library i.eee

An additional library lcdf_vhdl contains a package called Func_prims mad. up of basic logi. gates, latches, flip flo

Entity

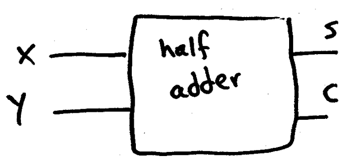
- The entity describes the interface.
- The interface description is like a pin description in a data book specifying the inputs and outputs to the block

Example



andgate

```
① ENTITY and2 IS
② PORT (a, b : IN BIT;
③ c : OUT BIT);
④ END and2;
```




```
① ENTITY half-adder IS
② port (x, y : IN BIT;
③ s, c : OUT BIT);
④ END half-adder;
```

in each case

line #1 indicates a definition of a new entity

line #4 marks the end of the definition

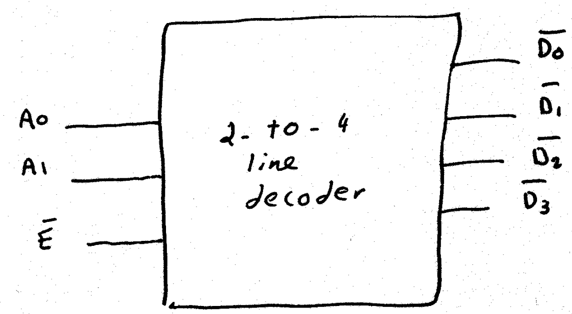
line #2 is the port clause ⟶ describes the interface

line #2 → list of names, a mode, type

- a mode → in, out, inout
- type → bit → '0', '1'; integer; Real

ENTity Declaration

Black Box representation of circuit



Basically we want to describe the Interface of the black box/entity using entity declaration

```
entity decoder_2-to-4 is port (
    En , A0, A1  : in  std-logic ;
    D0_n, D1-n , D2-n , D3-n : out  std-logic);
end  decoder-2-to-4 ;
```

- reserved words : entity, is, port, in, out, end

Another useful mode : inout mode

A Quick Reference

This section is not meant to cover *all* possible declarations, but hopefully the most relevant are present. Note that terms surrounded by square braces (`[...]`) in the code presented are optional, and therefore are not required (the square braces themselves are not to be included in the code).

A.1 Reserved Words

The following list of reserved words is by no means complete, but contains most (if not all) of the reserved words that are of interest to the beginner. The boolean functions **and**, **nand**, **nor**, **not**, **or**, **xnor**, **xor** are found in the `ieee` library, in the `std_logic_1164` package (as may be others).

Table 4: Reserved Words

abs	access	active	after	alias
all	and	architecture	array	ascending
assert	attribute	begin	bit	bit_vector
body	boolean	buffer	case	character
component	configuration	constant	downto	else
elsif	end	entity	error	event
exit	file	for	function	generic
high	if	in	inertial	inout
is	integer	last_active	last_event	last_value
left	length	library	line	loop
low	map	mod	nand	natural
nor	not	null	of	on
open	or	others	out	package
port	positive	procedure	process	range
read	real	reject	rem	report
return	right	rol	ror	severity
signal	sla	sll	sra	srl
string	subtype	text	transport	then
time	to	type	units	until
use	variable	wait	when	while
with	write	xnor	xor	-

How to represent circuits in VHDL

3-ways → the main difference between approaches is the level of detail required.

① Structural: ≈ schematic of a circuit

architecture body

Consists of VHDL netlists,
lists of signals
How they are joined by components

② Data flow: Describe a circuit in terms of function rather than structure and is made up of concurrent ~~st~~ assignment statements

internal working

→ executed concurrently (in parallel

③ Behavioral: Describes how the circuit behaves. (i.e internal behaviour) using process construct sec 3

Combinational Logic

can be written

sequential statement

* must be executed in a given sequential order

Used in Behavioral Descriptions

concurrent statements

* may be executed in parallel i.e concurrently

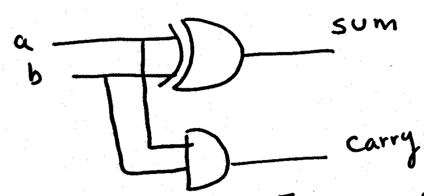
Found in

- Data Flow
- structural Descriptions

Entity & architecture

Entity describes the interface

Architecture describes the behaviour



a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

→ sum = $\bar{A}B + A\bar{B} = A \oplus B$

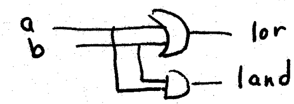
→ carry = AB

```
library ieee;
use ieee.std_logic_1164.all
entity half_adder is port (
        a, b      : in  std_logic;
    sum, carry    : out std_logic;
end half_adder;

architecture behaviour of half adder is
begin
    sum   <= (a xor b);
    carry <= (a and b);
end behaviour;
```

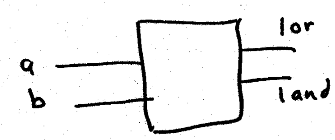
adhere to IEEE 1164 standard signal value.

Boolean statements

assume we want to build a circuit that will produce the logical-and and logical-or of two bits.




```
library ieee;
use ieee.std-logic-1164.all;
```




```
entity cct1 is port (
    a, b      : in  std.logic
    land, lor : out std.logic);

architecture arch-cct1 of cct1 is
begin
    land <= a and b;
    lor  <= a or b;
end arch-cct1;
```

defines the body → how it is connected or how it behaves or internal working

Structural style of Modeling:

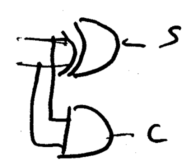
In the structural style of modeling, an entity is described as a set of interconnected components.

same as before

```
entity HALF_ADDER is
    port (A, B : in BIT ; SUM, Carry : out Bit);
end HALF_ADDER;
```

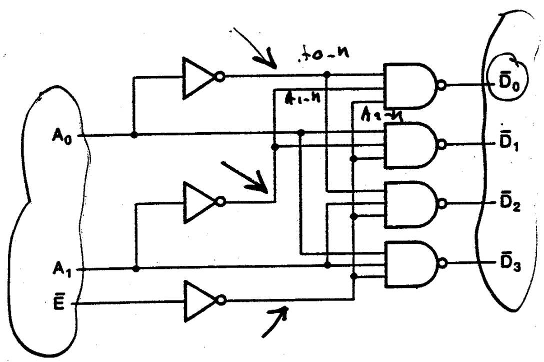
```
architecture HA_STRUCTURE of HALF_ADDER is

    Component XOR2
      port ( X, Y : in Bit ; Z : out Bit);
    end component ;

    Component AND2
      port ( L, M : in Bit ; N : out Bit);
    end component ;

  begin
    X1 : XOR2 port map ( A, B, SUM);
    A1 : AND2 port map ( A, B, CARRY);
  end HA_STRUCTURE
```

declarative part



statement part

* The structural description of a design is simply a textual description of a schematic!
  i.e we specify a netlist



(a) Logic diagram

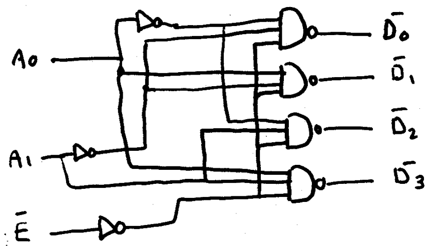
$\bar{E}$	A_1	A_0	$\bar{D}_0$	$\bar{D}_1$	$\bar{D}_2$	$\bar{D}_3$
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

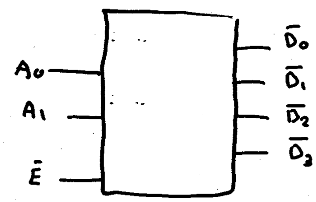
(b) Truth table

$$\bar{D}_0 = \overline{E\,\bar{A}_1\,\bar{A}_0}$$
$$\bar{D}_1 = \overline{E\,\bar{A}_1\,A_0}$$
$$\bar{D}_2 = \overline{E\,A_1\,\bar{A}_0}$$
$$\bar{D}_3 = \overline{E\,A_1\,A_0}$$

(c) Logic Equations

☐ **FIGURE 3-14**
A 2-to-4-Line Decoder






```
-- 2-to-4 Line Decoder: Structural VHDL Description
-- (See Figure 3-14 for logic diagram)
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity decoder_2_to_4 is
  port(E_n, A0, A1: in std_logic;
     D0_n, D1_n, D2_n, D3_n: out std_logic);
end decoder_2_to_4;

architecture structural_1 of decoder_2_to_4 is
  component NOT1
    port(in1: in std_logic;
        out1: out std_logic);
  end component;
  component NAND3
    port(in1, in2, in3: in std_logic;
        out1: out std_logic);
  end component;
  signal E, A0_n, A1_n: std_logic;
  begin
    g0: NOT1 port map (in1 => A0, out1 => A0_n);
    g1: NOT1 port map (in1 => A1, out1 => A1_n);
    g2: NOT1 port map (in1 => E_n, out1 => E);
    g3: NAND3 port map (in1 => A0_n, in2 => A1_n,
            in3 => E, out1 => D0_n);
    g4: NAND3 port map (in1 => A0, in2 => A1_n,
            in3 => E, out1 => D1_n);
    g5: NAND3 port map (in1 => A0_n, in2 => A1,
            in3 => E, out1 => D2_n);
    g6: NAND3 port map (in1 => A0, in2 => A1,
            in3 => E, out1 => D3_n);
end structural_1;
```

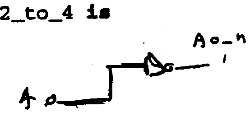


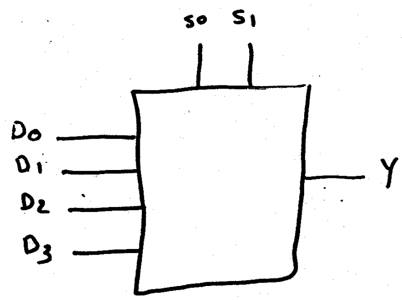
FIGURE 3-36
Structural VHDL Description of 2-to-4 Line Decoder

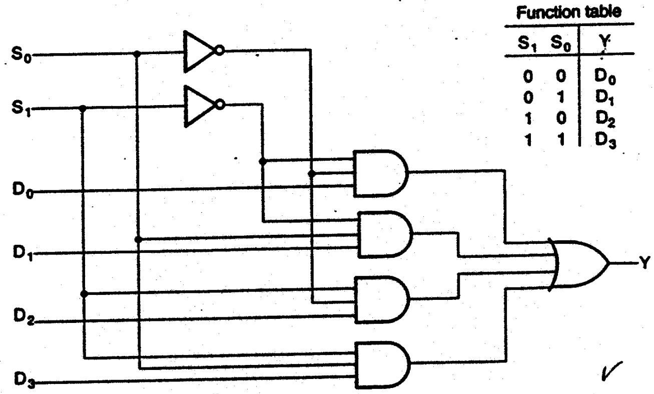
port map ( );

maps the input and output of the inverter to the signals to which they are connected.

```
-- 2-to-4 Line Decoder: Dataflow VHDL Description
-- (See Figure 3-14 for logic diagram)
Use library, use, and entity entries from 2_to_4_decoder_st;

architecture dataflow_1 of decoder_2_to_4 is

signal A0_n, A1_n: std_logic;
begin
  A0_n <= not A0;
  A1_n <= not A1;
  E <= not E_n;
  D0_n <= not ( A0_n and A1_n and E);
  D1_n <= not ( A0 and A1_n and E);
  D2_n <= not ( A0_n and A1 and E);
  D3_n <= not ( A0 and A1 and E);
end dataflow_1;
```

□ **FIGURE 3-38**
Dataflow VHDL Description of 2-to-4 Line Decoder





Function table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

☐ FIGURE 3-19
4-to-1-Line Multiplexer

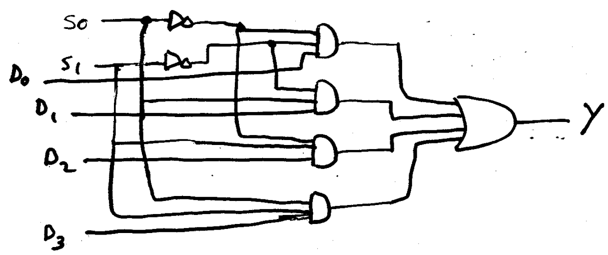
or std_logic

method #1

```
entity MUX_4-to-1_st is
  port ( D0 : in bit;
         D1 : in bit;
         D2 : in bit;
         D3 : in bit
         So : in bit
         S1 : in bit
         Y  : out bit);
end mux-4-to-1-st;
```

method #2

```
entity mux-4-to-1-st is
port ( D : in std_logic_vector (0 to 3)
       S : in std_logic-vector (0 to 1)
       Y : out std_logic);
end mux-4-to-1-st;
```




```
-- 4-to-1 Line Multiplexer: Structural VHDL Description
-- (See Figure 3-19 for logic diagram)
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity multiplexer_4_to_1_st is
  port(S: in std_logic_vector(0 to 1);
       D: in std_logic_vector(0 to 3);
       Y: out std_logic);
end multiplexer_4_to_1_st;

architecture structural_2 of multiplexer_4_to_1_st is
  component NOT1
    port(in1: in std_logic;
         out1: out std_logic);
  end component;
  component AND3
    port(in1, in2, in3: in std_logic;
         out1: out std_logic);
  end component;
  component OR4
    port(in1, in2, in3, in4: in std_logic;
         out1: out std_logic);
  end component;
  signal S_n: std_logic_vector(0 to 1);
  signal N: std_logic_vector(0 to 3);
  begin
    g0: NOT1 port map (S(0), S_n(0));
    g1: NOT1 port map (S(1), S_n(1));
    g2: AND3 port map (S_n(1), S_n(0), D0), N(0));
    g3: AND3 port map (S_n(1), S(0), D(1), N(1));
    g4: AND3 port map (S(1), S_n(0), D(2), N(2));
    g5: AND3 port map (S(1), S(0), D(3), N(3));
    g6: OR4 port map (N(0), N(1), N(2), N(3), Y);
end structural_2;
```

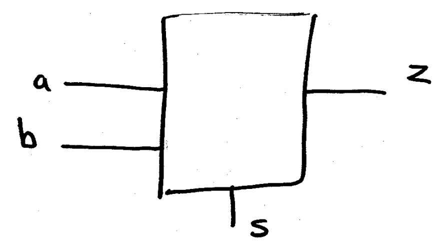
☐ **FIGURE 3-37**
Structural VHDL Description of 4-to-1 Line Multiplexer

Concurrent statements

3.1.2

① With - select - when statements

it is used in cases where a signal value is assigned based on the value of another signal (i.e selection signals of MUX)




```
with s select
    z <= a when '0' ;
         b when '1' ;
```

II with select when - when others

$\boxed{\text{Note}}$ A single bit does not necessarily have only two values (unless it is explicitly Boolean)

other possible value include

- high impedance
- unspecified

⋮

so in this situation we can use
when - others
as a catch-all for all other not specified
i.e like default in C, C++

Ex

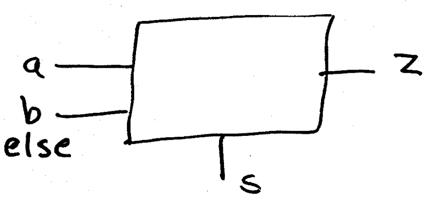
```
with s select
  z <= a when '0',
       b when others;
```

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

entity mux is port (
    In0, In1, In2, In3 : in  std_logic_vector (7 downto 0);
    sel                : in  std_logic_vector (1 downto 0);
    out                : out std_logic_vector (7 downto 0));
end mux;

architecture behavioral of mux is
begin
    with sel select
    out <= In0 when "00"        -- evaluated concurrently
           In1 when "01"
           In2 when "10",
           In3 when "11",
           "00000000" when others;   -- must cover full range
end behavioral;
```

III When - Else Statements

"when- else" statements are a version of then "with - select" statements where assignment is based on a condition

Ex




```
z <= a when (s = '0') else
     b;
```

Note The condition evaluated in this type of statement may be based on a single signal, or on multiple signals or multiple conditions

Ex

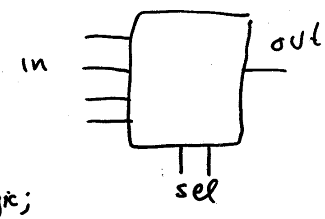
```
z <= a when ( s = '0' and cond1 = true) els
     b when ( s = '1' and cond2 = true) els
     z;
```

CS

when statement

conditional

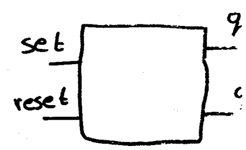
```
library ieee;
use ieee.std_logic

  entity mux is port(
     In0, In1, In2, In3   : in std-logic;
      sel                 : in std-logic;
      out                 : out std-logic;
  end mux;

 architecture behavioral of mux is
 begin
       out <= In0 when sel = "00" else
              In1 when sel = "01" else
              In2 when sel = "10" else
              In3 when sel = "11" else
              "000000";

end behavioral;
```



3 Behavioral Description (sequential)

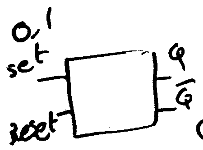
(mp) The fact that VHDL has so many possible representations for similar functionality is what makes learning the entire language a BIG TASK

```
Entity rsff is
  port ( set, reset : in Bit;
         q, qb : inout Bit);
END rsff;

Architecture sequential of rsff is
BEGIN
process (set, reset)
  IF set = '1' AND reset = '0' Then
     q <= '0' After 2 ns;
     qb <= '1' After 4 ns;
  ELSif set = '0' AND reset = '1' Then
     q <= '1' After 4 ns;
     qb <= '0' After 2 ns;
  ELSif set = '0' AND reset = '0' Then
     q <= '1' after 2 ns;
     qb <= '1' after 2ns;
  END IF;
END process;
End sequential;
```



process statements consists of :

① sensitivity list   process (set, reset)



This list enumerates exactly which signals causes the process statement to be executed

i.e only events on these signals cause the process statement to be executed

② Declarative Region   process (set, reset)

- - - - - - -

BEGIN

used to declare local variables or constants that can be used only inside the process

③ process statement part

starts at keyword BEGIN

ends at " END PROCESS

imp All of the statements enclosed by the process are sequential statements.

Sequential statements :

- IF
- case
- Loop
- Assert
- wait

process(set, reset)

* **Case** : This construct is well suited for situations in which several branches of execution need to be selected on the value of a single expression

imp ① case statement must cover all possible values of the expression upon which selection is being made.

② The others keyword may be used to catch all others not covered

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

entity half_adder is port(
   a,b       : in  std_logic;
   sum,carry: out std_logic);
end half_adder;

architecture behavioural of half_adder is
begin
   sum_proc: process(a,b)        -- using if statement
   begin
      if a = b then
         sum <= '0';
      else
         sum <= (a or b);
      end if;
   end process sum_proc;

   carry_proc: process(a,b)      -- using case statement
   begin
      case a is
      when '0' =>
         carry <= a;
      when '1' =>
         carry <= b;
      when others =>
         carry <= 'x';
      end case;
   end process carry_proc;
end behavioural;
```

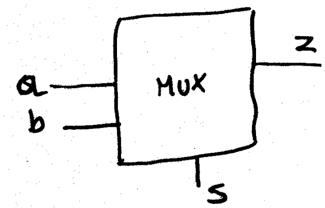
Figure 8: Half-Adder Code Using Process with If and Case Statements

Sequential statements

IF-then-Else

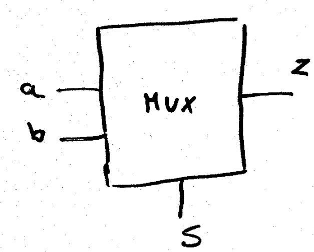
```
begin
process (s)
  begin
  if (s = '0') then
     z <= a;
  Elseif (s = '1') then
     z <= b;
  end if
end process;
```

Have same meaning in VHDL as they do in C-language



Case-when

```
begin
process (s)
 begin
   case s is
      when '0' => z <= a;
      when '1' => z <= b;
   end case;
 end process
```



③ loops

For

```
-- (standard) ieee library; entity declaration

architecture behavioural of the_entity is
   signal sum: std_logic_vector(7 downto 0);
begin
   init_proc: process(clk)
   begin
     for i in 7 downto 0 loop
        sum(i) <= '0';
     end loop;
       -- sequential statements
   end process init_proc;
end behavioural;
```

index is implied and thus only a name need be provided.

Figure 9: Simple For Loop, Initializing an 8-bit Vector

while

```
-- (standard) ieee library; entity declaration

architecture behavioural of the_entity is
   signal sum: std_logic_vector(7 downto 0);
begin
   init_proc: process(clk)
      variable i: integer := 0;
   begin
     while i < 8 loop
        sum(i) <= '0';
        i := i + 1;
     end loop
       -- sequential statements
   end process init_proc;
end behavioural;
```

Figure 10: Simple While Loop, Initializing an 8-bit Vector

Wait statement:

In all examples discussed so far, process execution has relied on a sensitivity list i.e process (s, r)

but there is an alternative : → wait statement

i.e wait for time expression;
wait on signal;
wait until condition;
wait;

wait for 10 ns
wait for A, B
wait until A = B

Once a wait statement is executed the process is suspended until either the time elapses or an event occurs on a signal or a condition is met and then the process resumes execution

Imp A process cannot have both sensitivity list & wait !!

```
process          -- No sensitivity list
   variable  TEMP1, TEMP2; Bit;
begin
   Temp1 := A and B
   Temp2 := C and D
   Temp1 := Temp1 or Temp2
   Z <= not Temp1;
   wait on  A, B, C, D;
```

* Null statement

null;

is also a sequential statement that does not cause any action to take place; execution continues with the next statement.

* Could be used in an if statement or in a case statement where for certain conditions it may be useful or necessary to explicitly specify that no actions needs to be performed!

* Exit statement

exit [loop-label] [when condition];

The exit statement is a sequential statement that can be used only inside a loop.

```
* loop
    wait on A, B
    exit when A = B
  end loop;
```

```
L3: loop
      :
    if sum > 100 then
      exit L3;
```

important   <=   vs   :=

1. sum <= (a xor b) ;

   signal assignment

Signal S1 : std_logic := '0';

   initialization

2. sum <= (a xor b) after 5 ns;

   more general form

3. Common syntax Errors

- Do not forget semicolon at end of statement.
- remember it is elsif and not else if
- it is an error to use endif → use end if
- Use underscore and not hyphens in label names.

  i.e half_adder ✓   half-adder ✗

- Do not forget to leave a space between the number and time base designations

  10 ns ✓   10ns ✗

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY switches IS
 PORT
 (
    dipsw: IN STD_LOGIC_VECTOR(8 DOWNTO 1); -- DIP SWITCHES
    spareb: IN STD_LOGIC;                   -- SPARE puchbutton
    resetb: IN STD_LOGIC;                   -- RESET pushbotton

    s: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);    -- XS Board LED Digit
    lsb: OUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- XStend Left LED Digit
    rsb: OUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- XStend Right LED Digit
    db: OUT STD_LOGIC_VECTOR(8 DOWNTO 1);  -- XStend Bargraph LED

    oeb: OUT STD_LOGIC; -- ouptput enable for all RAMS
    rst: OUT STD_LOGIC  -- microcontroller reset
 );
 END switches;

ARCHITECTURE switches_arch OF switches IS
BEGIN
 -- this prevents accidental activation of the RAMS or microcontroller
 oeb <= '1'; -- disable all the RAM output drivers
 rst <= '1'; -- disable the microcontroller

 -- light the XS Board LED digit with the pattern from the
 -- DIP switches if both pushbuttons are pressed
 -- these LED segments are active-high
 s <= dipsw(7 DOWNTO 1) WHEN (spareb = '0' AND resetb = '0') ELSE
       "0000000"; -- otherwise keep LED digit dark

 -- light the XStend Left LED digit with the pattern from the
 -- DIP switches if the RESET pushbutton is pressed
 -- these LED segments are active low.
 lsb <=NOT(dipsw) WHEN (spareb='1' AND resetb='0') ELSE
       "11111111"; -- otherwise keep the LED digit dark

 -- light the XStend right LED digit with the pattern from the
 -- DIP switches if the SPARE pushbutton is pressed
 -- these LED segments are active low.
 rsb <= NOT(dipsw) WHEN (spareb='0' AND resetb='1') ELSE
       "11111111"; -- otherwise keep the LED digit dark

 -- light the XStend bargraph LED with the pattern from the
 -- DIP switches if neither pushbutton is pressed
 -- these LED segments are active low.
 db <= NOT(dipsw) WHEN (spareb='1' AND resetb='1') ELSE
       "11111111"; -- otherwise keep the bargraph LED dark
END switches_arch;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity leddcd is
  port (
      d: in UNSIGNED (3 downto 0);
      s: out STD_LOGIC_VECTOR (6 downto 0)
  );
end leddcd;

architecture leddcd_arch of leddcd is
begin
  with d select
    s <= "1110111" when "0000",  -- 0
         "0010010" when "0001",  -- 1
         "1011101" when "0010",  -- 2
         "1011011" when "0011",  -- 3
         "0111010" when "0100",  -- 4
         "1101011" when "0101",  -- 5
         "1101111" when "0110",  -- 6
         "1010010" when "0111",  -- 7
         "1111111" when "1000",  -- 8
         "1111011" when "1001",  -- 9
         "1111110" when "1010",  -- A
         "0101111" when "1011",  -- B
         "1100101" when "1100",  -- C
         "0011111" when "1101",  -- D
         "1101101" when "1110",  -- E
         "1101100" when others;  -- F
end leddcd_arch;
```