Department of Electrical
and Computer Engineering

Hardware Description Languages (HDLs)

Introduction

- Two main hardware description languages will be treated in this course: Altera Hardware Description Language (AHDL), and Very High Speed Integrated Circuit Hardware Description Language (VHDL).
- VHDL is an IEEE Standard (IEEE Std 1076-1987 or 1076-1993).

Altera Hardware Description Language (AHDL)

Introduction

Limitations: These notes are not attempting to describe the full details of AHDL, but just to give the flavour of the language and point out some of its features. In most cases further detail can be obtained from the MAX+PLUS II on-line help system.

- Reference: “MAX+PLUS II Text Editor and AHDL Manual”, Altera.
- High level modular language that is completely integrated into the MAX+PLUS II development system.
- Main features of AHDL:

  (i) State machine, truth tables, boolean equations, and group operations are supported and implemented in a user friendly format.

  (ii) Text, graphic and waveform files can be intermixed in a design hierarchy.

  (iii) Frequently used constants and prototypes. including prototypes of standard TTL, bus, and EPLD optimized Macrofunctions can be stored in Include Files (**.inc**) and incorporated into any Text Design File (**.tdf**).

(iv) Device resources can be user specified or assigned automatically.

Text Design File Sections

(i) Title Statement (Optional) – provides comments for the Report Files (**.rpt**) generated by the system.

(ii) Constant Statement (Optional) – specifies a symbolic name that can be substituted for a constant.

(iii) Function Prototype Statement – declares the ports of a macrofunction or primitive and the order in which those ports must be declared in an in-line reference.

(iv) Include Statement (Optional) – specifies an Include File (**.inc**) that replaces the Include statement in the TDF.

(v) Options Statement (Optional) – sets the Turbo and Security Bits of Altera devices and determines how product terms are allocated.

(vi) Design Sections (Required/Optional) – specifies device, clique, chip, pin, and macrocell assignments, and logic options.

(vii) Subdesign Section (Required) – declares the input, output, and bidirectional ports of a design.

(viii) Variable Section (Optional) – declares variables that represent and hold internal information.

(ix) Logic Section (Required) – defines the logical operations of the design.

- AHDL is a concurrent language – i.e. all behaviour described in the logic section is evaluated at the same time and not sequentially as in a conventional programming language.
- To include lower level design files in a higher level TDF a function prototype statement must be included.

Department of Electrical
and Computer Engineering

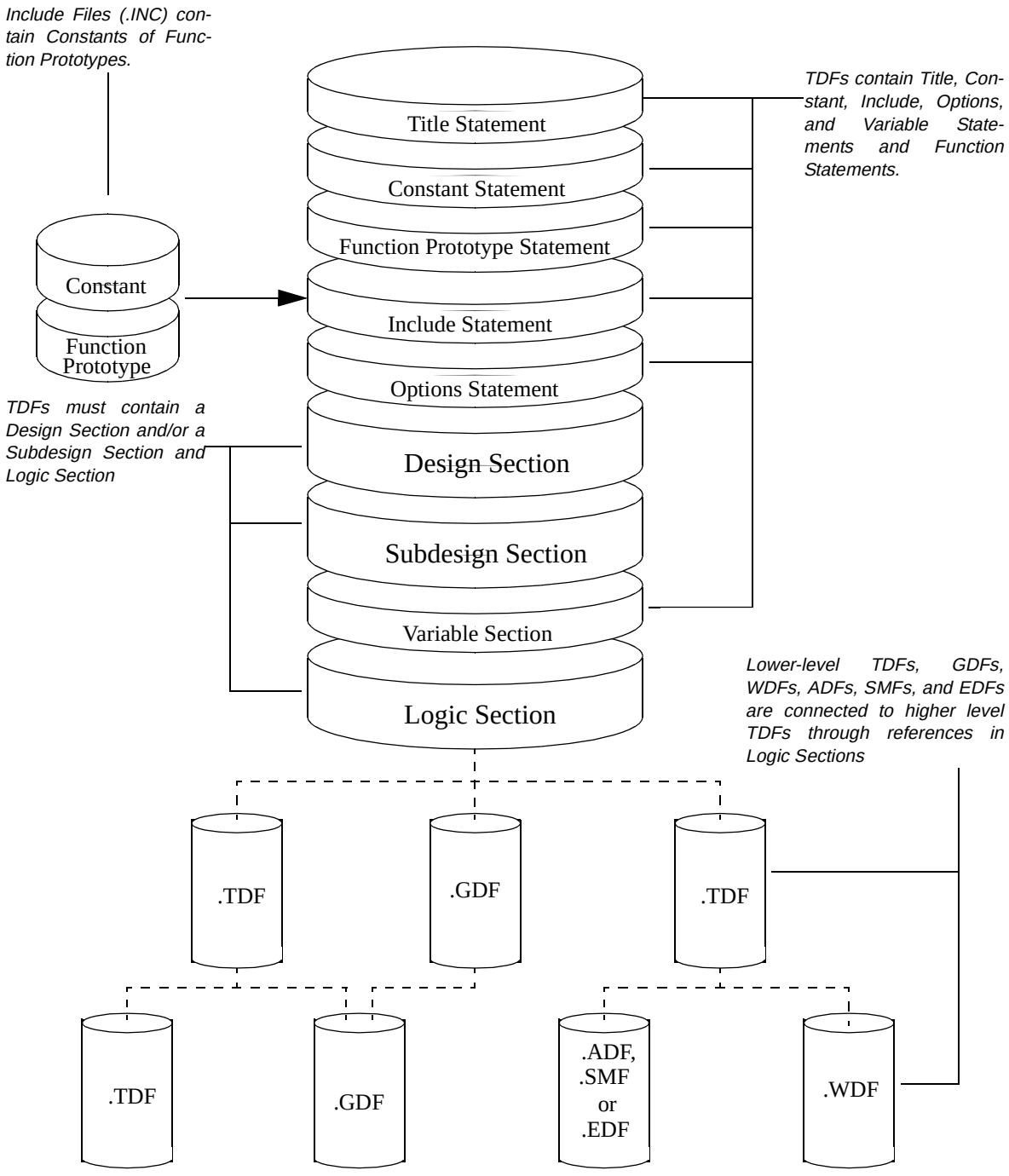


Figure 1 : Text Design File Structure

Department of Electrical
and Computer Engineering

Some tips

- TDF file is a standard ascii file.
- Lines in a TDF may be up to 255 characters long.
- AHDL is not case sensitive.
- Comments must be *enclosed* in percent symbols (%). Comments cannot be nested.
- See Table 1 for a list of reserves keywords. Keywords cannot be used in a design file, except if enclosed in single quotes – i.e. '*name*', where *name* is the keyword used in a name.
- The GND constant is defined as a low level input voltage and is equivalent to a low (0) or false. VCC constant is defined as a high level input voltage and is equivalent to high (1) or true.
- The rules for the symbols that can be used in AHDL variable names are much the same as standard programming languages.

Table 1: Reserved Keywords

Reserved Keywords		
AND	IF	OUTPUT
ANY	INCLUDE	PTERM_ALLOC
BEGIN	INPUT	RETURNS
BIDIR	IS	STATES
BIT0	LSB	SECURITY
BITS	MACHINE	SUBDESIGN
BURIES	MACRO	THEN
CASE	MSB	TITLE
CLIQUE	NAND	TABLE
CONSTANT	NODE	TURBO

Department of Electrical
and Computer Engineering

Table 1: Reserved Keywords

Reserved Keywords		
DEFAULTS	NOR	VARIABLE
DESIGN	NOT	VCC
DEVICE	OF	WHEN
ELSE	OFF	WITH
ELSIF	ON	X
END	OPTIONS	XNOR
FUNCTIONS	OR	XOR
GND	OTHERS	

- Unquoted symbolic names: can contain up to 32 characters consisting of A-Z, a-z, slash (/) and underscore (_). Cannot be keywords, and may not consist entirely of digits. e.g. legal names: a /a; illegal names: -foo node 55.
- Quoted symbolic names: enclosed in single quotes (‘). Allow the use of dashes (-), can use keywords, can consist entirely of digits.
- A list of symbols used in AHDL and their meanings appear in Table 2

Table 2: AHDL Symbols

Symbol	Function
_ (Underscore) / (Slash) - (dash)	Legal charters in symbolic names (i.e. user defined identifiers). See note about dash.
% (Percent)	Enclose comments

Table 2: AHDL Symbols

Symbol		Function
( )	(Left and right parentheses)	Enclose and define group names. Enclose pin names in Subdesign Sections and Function Prototypes. Enclose inputs and outputs of truth tables. Enclose states in State Machine Declarations. Enclose highest priority operations in boolean expressions. Enclose options in a Design Section (within a Resource Assignment Statement).
[ ]	(Left and Right Brackets)	Enclose the number range of a group.
'...'	(Single quotation marks)	Enclose quoted symbolic names.
"..."	(Double quotation marks)	Enclose text string in Title Statements. Enclose pathname in Include Statements. Enclose digits in non-decimal numbers. Enclose design name and device name in Design Sections (optional).
.	(Period)	Separates symbolic names of macrofunction or primitive variable from stubs. Separates extensions from filenames.
..	(Ellipsis)	Separates most significant bit (MSB) from least significant bit (LSB) in ranges.
;	(Semicolon)	Ends AHDL statements and sections.
,	(Comma)	Separates members of groups and lists.
:	(Colon)	Separates symbolic names from types in declarations and resource assignments.

Table 2: AHDL Symbols

Symbol	Function
@ (At)	Assigns symbolic nodes to device pins and macrocells in Resource Assignment Statements.
= (Equals)	Assigns defaults GND and VCC values to inputs in Subdesign Section. Assigns settings to options. Assigns values to state machine states. Assigns values in boolean equations.
=> (Arrow)	Separates inputs from outputs in truth table statements. Separates WHEN clauses from boolean expressions in Case statements.

Ports

- Ports are variables connected to inputs and outputs of primitives or macrofunctions

  e.g. if one declares a variable `cap` of type `LATCH` you may use the following ports in your design: `cap.d, cap.ena and cap.q`.

Table 3: Commonly used port definitions

Port	Definition
`.q`	Output of a flip-flop
`.d`	Data input of a D-type flip-flop or latch
`.t`	Toggle input of a T-type flip-flop
`.j`	J input of a JK flip-flop
`.k`	K input of a JK flip-flop

Table 3: Commonly used port definitions

Port	Definition
`.s`	Set input of an SR-type flip-flop
`.r`	Reset input of an SR-type flip-flop
`.clk`	Clock input of a flip-flop
`.ena`	Enable input of a flip-flop or latch
`.prn`	Active low preset input of a flip-flop
`.clrn`	Active low clear input of a flip-flop

Groups

- Symbolic names and ports of the same type may be declared and used as a group in boolean expressions and equations.
- Group may include up to 256 members and is treated as a collection of bits and acted upon as one unit.

Notation

Groups can be declared with the following two notations:

1. A symbolic name or port followed by a range of decimal numbers enclosed in brackets, e.g. `a[4..1]`. Only one range is allowed after a group identifier.

   ☞ Once the group has been defined, [ ] is a shorthand way of specifying the entire range.

2. A list of symbolic names, ports or numbers separated by commas and enclosed in parentheses, e.g. `(a, b, c)`. Groups with ranges can also be listed within the parentheses. For example, `(a, b, c[5..1])` is a legal group.

☞ This notation is useful for specifying ports. For example. the input ports of variable cap of type DFF can be written as `cap.(d, clk, clrn.prn)`.

e.g. Same group specified with different notations:

```
b[5..0]
(b5, b4, b3, b2, b1, b0)
b[ ]
```

Numbers

- May use numbers in decimal, binary, octal and hexadecimal in any combination in AHDL

  e.g. valid AHDL numbers
  `B"0110X1X10"`
  `Q"4671223"`
  `H"123AECF"`

Table 4: Radix Systems

Numbering System	Syntax
Decimal	<series of digits `0` to `9`>
Binary	`B"`<series of 0's, 1's, X's>`"` (where `X` = "don't care")
Octal	`O"`<series of digits `0` to `7`>`"` or `Q"`<series of digits `0` to `7`)"
Hexadecimal	`H"`<series from `0` to `9`, `A` to `F`)"

Following rules apply to AHDL numbers:

- The MAX+PLUS II compiler always interprets numbers as groups of binary digits.
- Numbers may not be assigned to single nodes in Boolean equa-

tions. Use `VCC` and `GND` instead.

Boolean Expressions

- Boolean expressions consist of operands (symbolic names, ports, groups or constants) separated by logical and arithmetic operators and comparators.
- Can also be used in Case and If statements.
- A boolean expression may be one of the following:
  - An operand (e.g., `a, b[5..1], 7, VCC`)
  - An in-line primitive or macrofunction reference.
  - A prefix operator (`!` or `-`) applied to a boolean expression (e.g. `!c`)
  - Two boolean expressions separated by a binary (non-prefix) operator (e.g. `d1 $ d3`).
  - A boolean expression enclosed in parentheses (e.g. `(!foo & bar)`).

☞ The result of every boolean expression must be the same width as the node or group (on the left side of an equation) to which it is eventually assigned.

Table 5: Logical Operators

Symbol	Example	Description
`!` `NOT`	`!tob` `NOT tob`	one's complement (prefix inverter)
`&` `AND`	`bread & butter` `bread AND butter`	AND

Table 5: Logical Operators

Symbol	Example	Description
`!&` `NAND`	`a[3..1] !& b[5..3]` `a[3..1] NAND b[5..3]`	AND inverter
`#` `OR`	`trick # treat` `trick OR treat`	OR
`!#` `NOR`	`c[8..5] !# d[7..4]` `c[8..5] NOR d[7..4]`	OR inverter
`$` `XOR`	`foo $ bar` `foo XOR bar`	exclusive OR
`!$` `XNOR`	`x2 !$ x4` `x2 XNOR x4`	exclusive NOR

Some Tips on Boolean Operations

- NOT of a group carries out a NOT on all members of the group, e.g. `!a[4..1]` is `(!a4, !a3, !a2, !a1)`.
- `(a, b, c) # (d, e, f)` is interpreted as `(a # d, b # e, c # f)`.
- Of one operand is a single node is it is being applied to a group then the single node is duplicated to make it the same size as the group and then applied, e.g. `a & b[4..1]` is interpreted as `(a & b4, a & b3, a & b2, a& b1)`.
- If both operands are numbers then the shorter number is sign extended to match the size of the larger number an the operation

is carried out.

- If one operand is a number and the other is a group, then the number is sign extended or truncated to match the size of the group. If significant bits are truncated an error message is generated.

Arithmetic Operators

Table 6: Arithmetic operators

Operator	Example	Description
+ (unary)	`+1`	positive (has not effect)
- (unary)	`-a[4..1]`	negative - carries out the two's complement.
+	`time[4..2] + day[3..1}`	binary addition
-	`(dig, wig, fig) - (cat, hat, bat)`	binary two's complement subtraction

Similar rules apply to those in the previous case.

Comparators

- Two types of comparisons – logical and arithmetic.

Table 7: Comparison operators

Comparator	Type	Example	Description
`==`	logical	`foo == bar`	equal
`!=`	logical	`b1 != b3`	not equal
`<`	arithmetic	`fame[] < power[]`	less than

Table 7: Comparison operators

Comparator	Type	Example	Description
<=	arithmetic	`money[] <= power[]`	less than or equal
>	arithmetic	`love[] > money[]`	greater than
>=	arithmetic	`y[5..0] >= z[5..0]`	greater than or equal

Priorities

- The operators have precedence rules, but I suggest that you use brackets to make the order of evaluation clear.
- Precedence order is essentially unary operators, arithmetic (+ and –), comparisons, AND and NAND, XOR and XNOR, and OR and NOR.

Department of Electrical
and Computer Engineering

Basic Logic Primitives

Table 8: MAX+PLUS II Flip-flops and Latches

Primitive	AHDL Function Prototype
LATCH	FUHCTION LATCH (d, ena) RETURNS (q);
DFF	FUNCTION DFF (d, clk, clrn, prn) RETURNS (q);
DFFE	FUNCTION dffe (d, clk, clrn, prn, ena) RETURNS (q);
JKFF	FUNCTION JKFF (j, k, clk, clrn, prn) RETURNS (q);
JKFFE	FUNCTION JKFFE (j,k,clk, clrn, prn, ena) RETURNS (q);
SRFF	FUNCTION SRFF (s, r, clk, clrn, prn) RETURNS (q);
SRFFE	FUNCTION SRFFE (s, r, clk, clrn, prn, ena) RETURNS (q);
TFF	FUNCTION TFF (t, clk, clrn, prn) RETURNS (q);
TFFE	FUNCTION TFFE (t, clk, clrn, prn, ena) RETURNS (q);

Notes:

`clk` = Register Clock Input

Department of Electrical
and Computer Engineering

`clrn`	= Clear Input (active low)
`d, j, k, r, s, t`	Input from the logic array
`ena`	Latch enable (latch active if ena high)
`prn`	Preset Input (active low)
`q`	Output

Table 9: MAX+PLUS II Logic Primitives

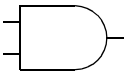
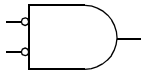
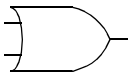
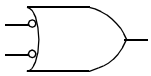
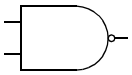
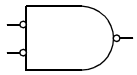
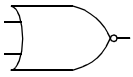
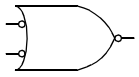
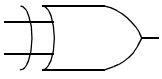
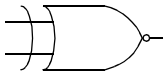
Sample Symbols	Description
 AND2  BAND2  OR2  BOR2  NAND2  BNAND2  NOR2  BNOR2  XOR  XNOR	Name: `AND2, AND3, AND4, AND6, AND8, AND12` Same for the other logic functions. Description: Output: `OUT` = logical <name> of inputs Input: `IN1, IN2,...IN12` = 2, 3, 4, 6, 8, 12 inputs

Table 9: MAX+PLUS II Logic Primitives

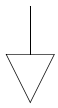
Sample Symbols	Description
 NOT	Name: `NOT` Description: Output: `OUT` = inverse of input Input: `IN1` = 1 input
 GND	Name: `GND` Description: Assigns a node to `GND`
VCC  VCC	Name: `VCC` Description: Assign a node to `VCC`

Table 10: MAX+PLUS II Input and Output Primitives

Pin Port	Description	Pin Port Declaration
`INPUT`	Input pin	`in:INPUT`
`OUTPUT`	Output pin	`out: OUTPUT`
`BIDIR`	Bidirectional pin	`input:BIDIR`

AHDL Design Structure

- An AHDL logic design must at a minimum contain a Subdesign Section and a Logic Section
- All other sections and statements are optional.

Subdesign Section

- Declares the input, output, and bidirectional ports of the design: e.g.

```
SUBDESIGN top
(
   foo, bar, clk1, clk2 : INPUT = VCC;
   a0, a1, a2, a3, a4 : OUTPUT;
   B[7..0] : BIDIR;
)
```

  - Subdesign name must conform to the file name.
  - List of ports and symbolic names are enclosed in parentheses.
  - If a Design Section exists some of the ports may be assigned to resources.
  - Input and bidirectional ports may be assigned default values.
  - Other port types: `MACHINE INPUT, MACHINE OUTPUT`.

Logic Section

- Specifies the logical operations of the design.
- Is the body of the Subdesign Section.
- Constructs that may be used:
  - Boolean equations
  - Truth Table Statement
  - Case Statement

- If Statement
- Defaults Statement
- In-line macrofunction or primitive reference.

- The BEGIN and END keywords enclose the Logic Section. A semicolon follows the END statement to end the Logic Section.

☞ AHDL is a ***concurrent language***. The compiler evaluates all the behaviour specified in the Logic Section of a TDF at the *same time* rather than sequentially. Equations that assign multiple values to the same AHDL node or variable are logically ORed.

Boolean Equations

- Represent the connection of wires, the flow of inputs into logical elements, and the flow of the outputs.

  e.g.

```
a[] = ((c[] & -B"001101") + e[6..1])
                              # (p,q,r,s,t,v)
```

Left side of equation may be symbolic name, port or group.

What happens in the above:

(i) Binary number `B"001101"` is two's complemented to become `B"110011"`.

(ii) `B"110011"` is ANDed with group `c[]`.

(iii) Result of (ii) is added to group `e[6..1]`.

(iv) Result of (iii) is ORed with the group `(p,q,r,s,t,v)`.

Final result is assigned to group `a[]`.

☞ For the above to be legal groups `a[]` and `c[]` must each have

size members.

Truth Table Statement

- Allows logical boolean statements to be specified using a truth table.

  e.g.

```
TABLE
  a0,  f[4..1].q   =>   f[4..1].d,   control;

  0,   H"0"        =>   H"1",        1;
  0,   H"4"        =>   H"2",        0;
  1,   B"0XXX"     =>   H"4",        0;
  X,   H"F"        =>   H"5",        1;

END TABLE;
```

- Input signals are `a0` and `f[4..1].q`.
- Output signals are `f[4..1].d` and `control`.
- Each signal has a one-to-one correspondence to the values in each entry.
- Nodes in heading can be either single nodes or groups.
- It is not necessary to list every possible combination of input values.
- The Defaults Statement assigns output values in cases when the actual inputs do not match the input values of the table.

  ☞ State names can be used as input and output values (see State Machines later).

Case Statement

- Similar to the CASE statement in a conventional programming language.

  e.g.

```
CASE f[].q IS
   WHEN H"00" =>
      addr[] = 0;
      s = a & b;
   WHEN H"01" =>
      count[].d = count[].q + 1;
   WHEN H"CF" =>
      f[3..0].d = addr[4..1];
   WHEN OTHERS =>
      f[].d = f[].q;
END CASE;
```

- Keywords `CASE` and `IS` enclose a Boolean expression.
- `CASE` Statement is terminated by the keywords `END CASE` and ‘`;`’.
- Keyword `WHEN` begins alternative. Comma separated list follows. If any expression following the `CASE` keyword evaluates to any member of the list the behavioural statements following the arrow are activated.
- If no alternative is true, then the keywords `WHEN OTHER` define the default alternative.

If Statement

- List a series of behavioural statements to be activated after the positive evaluation of one or more Boolean expressions.

  e.g.


```
IF (a[] == b[]) THEN
   c[8..1] = H"77";
   addr[3..1] = f[3..1].q;
   f[].d = addr[] + 1;
ELSIF (g3 $ g4) THEN
   f[].d = addr[];
ELSE
   d = VCC;
END IF;
```

- Expressions `(a[] == b[])` following `IF` and `(g3 $ g4)` following `ELSIF` are evaluated concurrently. If these statements evaluate to true then the expressions following the `THEN` statement are evaluated.

  ☞ Note that the second statement `(g3 $ g4)` effectively becomes: `(!(a[] == b[]) & (g3 $ g4))` to make the `ELSIF` control work correctly.

- `ELSIF` statements may be repeated for a large number of alternatives.
- `ELSE` is similar to the `WHEN OTHERS` statement in the `CASE` statement in that it provides a default alternative.

If Statement vs Case Statement

- Can often use either statement to achieve the same results.

  e.g. See Table 11

  Differences are:

  - Statements in If Statement may be any type of boolean expression.
  - Each expression following an IF or ELSIF statement may be unrelated to the others.

Table 11: Comparison of IF and CASE Statements

If Statement

```
IF (a[] == 0) THEN
   x = c & d;
ELSIF (a[] == 1) THEN
   x = foo & bar;
ELSIF (a[] == 2) THEN
   x = cats & dogs;
ELSIF (a[] == 3) THEN
   x = kumquat;
ELSE
   x = 0;
END IF;
```

Case Statement

```
CASE (a[1..3]) IS
   WHEN o =>
      x = c & d;
   WHEN 1 =>
      x = foo & bar;
   WHEN 2 =>
      x = cats & dogs;
   WHEN 3 =>
      x = kumquat;
   WHEN OTHERS =>
      x = 0;
END CASE;
```

- In a CASE statement one boolean expression is compared to a constant only.
- Interpretation of the If Statement can generate logic that is too complex for the compiler.

e.g.

Interpretation of an If Statement. Note that if `a` and `b` are complex expressions then the inversion is likely to be more complex.

If Statement

```
IF a THEN
     b = c;

ELSIF d THEN
     b = e;
```

Case Statement

```
IF a THEN
     b = c;
END IF;
IF (!a & d) THEN
b = e;
END IF;
```

If Statement

```
ELSE
     b = f;
END IF
```

Case Statement

```
IF (!a & !d) THEN
     b = f;
END IF;
```

Defaults Statement

- Allows specification of default constant values for variables used in Truth Table, If and Case Statements.
- Active-high signals automatically default to `GND`, Default Statements are only required for active-low signals.

  e.g.

```
BEGIN
   DEFAULTS
      a = VCC;
   END DEFAULTS

   IF x & y THEN
      a = GND;        % a is active low.
   END IF;
END;
```

  If the If Statement is undefined (i.e. `x & y` is not true) then the Defaults Statement is activated.

- Only one Defaults Statement is allowed in the Logic Section, and it must appear immediately before the `BEGIN` keyword.
- Multiple assignments to a variable are logically ORed, except when the default for the variable is VCC.

  e.g.

```
BEGIN
```


```
   DEFAULTS
      a = GND;
      bn = VCC;
   END DEFAULTS;

   IF c1 THEN
      a = a1;
      bn = b1n;
   END IF;

   IF c2 THEN
      a = a2;
      bn = b2n;
   END IF;

END;
```

This example is equivalent to:

```
a = (c1 & a1) # (c2 & a2);
bn = (!c # b1n) & (!c2 # b2n);
```

- Active-low variables that are assigned more than once should be given a default value of VCC.

Variable Section

- Used to declare any variables used in the Logic Section. Used for defining buried (internal) logic.

  e.g.

```
VARIABLE
   pen, pencil, eraser : NODE;
   temp : HALFADD;
```

The internal variables are `pen`, `pencil`, `eraser` of type `NODE`, and `temp`, an instance of the macrofunction `HALFADD`.

- `NODE` is an all purpose variable type used for holding input or output information. `NODE` can be used on either the left or the right side of an equation.
- Type `NODE` is similar to the `INPUT`, `OUTPUT`, and `BIDIR` resource and port types of the Design and Subdesign Sections and represents a single wire that propagates signals.

Register Definition

- Variable section is used to define names for registers, including D, T, JK, and SR flip-flops.

  e.g.

```
VARIABLE
  nod : TFF;
```

- After making the above declaration one may use the following input/output ports on the device:

```
nod.t
nod.clk
nod.clrn
nod.prn
nod.q
```

☞ One can use the name of a primitive without a stub (e.g. with .q), on the right side of an equation if one wishes to use the output. Similarly primitives that have a single primary input may use the name of a primitive without a stub on the left side of an equation.

e.g.

```
VARIABLE
   a, b : dff;

a = b;    % equivalent to a.d = a.q; %
```

Instances

- Instances of a particular primitive or macrofunction are declared in the Variables Section.

  e.g.

```
VARIABLE
   star : moonbeam;
```

  Variable `star` is an instance of he macrofunction `moonbeam`. which has the following ports:

```
green, yellow : INPUT;
blue, red : OUTPUT;
cycle : BIDIR;
```

  One may therefore use the following ports of `star`:

`start.green, start.yellow` etc, etc.

Design Section

- Provides more global specification of the design – for example one can specify the pin and buried macrocell assignments, as well as where the logic should be placed.
- Can also specify specific logic to specific EPLDs

Device Subsection

- Specify the EPLDs and the pins and macrocells on the EPLD to be used when the design is fitted.
  - EPLD Specification: can partition a project by specifying blocks of logic to be programmed into certain EPLDs.
  - Resource Assignment Statement: requests that nodes in the project be assigned to particular pins or macrocells.
  - Clique Assignment Statement: allows one to keep certain logic together in a single EPLD by making clique assignments.

☞ Refer to the on-line help for more details on this topic.

Function Prototype Statement

- Provide a shorthand description of a function, listing its name and its input, output, and bidirectional pins.

  e.g.

```
FUNCTION moonbeam (green, yellow)
   RETURNS (blue, red, cycle);
```

☞ A function prototype mist be placed outside of both the Design Section and the Subdesign Section and must be called before the macrofunction is called.

☞ State machines can be imported and exported through function prototypes. See later for details.

Title Statement

- Provided only for documentation purposes.

  e.g.

```
TITLE "Octopus Design";
```

Constant Statement

- Allows a meaningful symbolic reference to be made to a constant number.

  e.g.

```
CONSTANT UPPER_LIMIT = B"110";
```

Include Statement

- Allows one to import text from another file into the current file.

  e.g.

```
INCLUDE "const.inc"
```

- Searches in the following places: the project directory, any user libraries specified with the User Libraries command and listed in the `USER_LIB` variable of the MAXPLUS2.INI or <*project name*>.INI file, or the MAX2LIB or MAX2INC directories created during installation.

Options Statement

- Allows the setting of the Turbo and Security bits of the Device Subsection that follows them.
- Specify whether the lowest numbered bit of a group will be the

most significant bit or least significant bit.

- Specify product terms should be allocated.

☞ See the on-line help for more details on this.

Department of Electrical
and Computer Engineering

State Machines

- State machines are very elegantly and simply implemented with AHDL.
- Language is structured so that one can assign bits or states values oneself, or allow MAX+PLUS II do the work.
- Compiler uses proprietary advanced heuristic algorithms to make automatic state assignments.
- Formal state machine design normally requires the following steps:

  (i) Draw a state diagram and construct a next state table.

  (ii) Assign bits to the machine.

  (iii) Assign values to the states.

  (iv) Using manual logic minimization techniques derive the flip-flop excitation equations.
- With AHDL and MAX+PLUS II only the first step is above required. The compiler automatically does:
  - assigns bits, selecting either T or D type flip-flop for each bit;
  - assigns state values;
  - applies sophisticated logic synthesis techniques to derive the excitation equations.

State Machine Structure

- Can be imported or exported in AHDL using `MACHINE INPUT` or `MACHINE OUTPUT`
- The specify a state machine one must have three items in the design:
  - A State Machine Declaration in the Variable Section.
  - One or more Boolean control equations in the Logic Section.

- A single behavioural statement or construct in the Logic Section that specifies state transitions.

State Machine Declaration

e.g.

```
VARIABLE
   SS : MACHINE OF BITS (q1, q2, q3)
        WITH STATES  )  S1 = B"000",
                        S2 = B"010",
                        S3 = B"7"
                      );
```

- Name of the state machine (the machine variable) is SS.
- The bits `q1`, `q2` and `q3` are the *output of the registers* of this machine.
- The states of this machine are `S1`, `S2` and `S3`, each of which is assigned a numerical value for the state bits `q1`, `q2`, and `q3`. Note that only the list of states is required, the assignment of the bits to the states is optional. If unassigned then the compiler makes the assignment.

☞ Each state of a state machine is represented by a unique pattern of high and low signals inside a flip-flop. The state bits are the flip-flops required by the machine to store the states. The number of states has the following relationship to the number of bits in a state machine:

$$\#\ \text{states} \leq 2^{(\#\ \text{bits})}$$

Control Equations

- Boolean equations used in the Logic Section to set up the state machine clock and reset signals.

e.g

```
ss.clk = clk1;
ss.reset = a & b;
ss.ena = clk1ena;
```

- Port `ss.clk` must always be assigned. The `ss.reset` is optional unless the start state of the machine has been assigned to a value other than zero.

State Transitions

- To specify state transitions of a state machine, you must conditionally assign state variables within a single behavioural construct. Case or Truth Tables are recommended for this.
- State machine transition rules:
  - First state in the declaration is the power-up state. Normally assigned a numerical value of zero. If another value is assigned then one must also assign a signal to the `reset` port that will initialize the machine by taking on a value of `VCC` for a short time. The default power-up state is zero.

  ☞ The `reset` for a state machine is an active high, unlike the `DFF` for which it is an active low.

  - State transitions occur on the rising edge of the clock.
  - If no state transition is specified at a given clock edge, the machine will stay in the last state assigned.

```
e.g.
VARIABLE
   ss : MACHINE OF BITS (q2, q2, q3)
        WITH STATES (  S1 = B"000",
                       S2 = B"010",
                       S3 = H"7" );
```


```
BEGIN
   ss.clk = clk1;
   ss.reset = a & b;

   CASE ss IS
      WHEN s1 =>
         IF (addr[] > H"12") THEN
            ss = s2;
         ELSE
            ss = s3;
            control = VCC;
         END IF;
      WHEN s2 =>
         IF (addr[] > B"101") THEN
            ss = s1;
         ELSE
            control = VCC;
         END IF
      WHEN s3 =>
         ss = s1;
         control = VCC;
   END CASE;
END;
```

- If the above example `ss` starts out in state `S1`.
- If the group `addr[]` represents a number greater than `H"12"`, the variable control assumes the value `GND`, and the machine proceeds to S2.
- If not, control assumes the value `VCC` and control proceeds to state `S3`.
- If state `S2`, if the group `addr[]` represents a number greater than `B"101"` then control assumes the value `GND` and the machine pro-

ceeds to S1. Otherwise control assumes the value VCC and the machine stays in S2.

- If state S3, the machine automatically proceeds to state S1 and control assumes the value VCC.

Machine Input and Machine Output

- Allows the import and export of state machines between Text Design Files, Graphic Design Files and Waveform Design Files by specifying an input and output signal as MACHINE INPUT or MACHINE OUTPUT in the Subdesign Section.

☞ When you import or export a state machine, the Function Prototype representing the file must indicate which inputs and outputs are state machines.

e.g.

State Machine Export

```
SUBDESIGN ss_def (
   clock, reset, count : INPUT;
   ss_out : MACHINE OUTPUT;% export of the SM %
)

VARIABLE
   ss : MACHINE WITH STATES (S1, S2, S3, S4, S5);

BEGIN
   ss_out = ss;   % assign state machine to the
                  variable %
   CASE (ss) IS
      WHEN S1 =>
         IF (count) THEN
            ss = S2;
```


```
      ELSE
         ss = S1;
      END IF;

   WHEN S2 =>
      IF (count) THEN
         ss = S3;
      ELSE
         ss = S2;
      END IF;

   WHEN S3 =>
      IF (count) THEN
         ss = S4;
      ELSE
         ss = S3;
      END IF;

   WHEN S4 =>
      IF (count) THEN
         ss = S5;
      ELSE
         ss = S4;
      END IF;

   WHEN S5 =>
      IF (count) THEN
         ss = S1;
      ELSE
         ss = S5;
      END IF;
  END CASE;
  ss.(clk, reset) = (clock, reset);
END;
```

Department of Electrical
and Computer Engineering

State Machine Import

```
SUBDESIGN ss_use (
   ss_in : MACHINE INPUT;% import state machine %
   out : output;
)

BEGIN
   out = (ss_in == s2) OR (ss_in == s4);
END;
```

Department of Electrical
and Computer Engineering

Some Design Tips

Defining Clock, Reset, or Enable for an AHDL State Machine

- State machine resets and enables are defined in the following way:

```
<machine name>.clk = <signal name>;
<machine name>.reset = <signal name>;
<machine name>.ena = <signal name>;
```

- `reset` signal is active high.
- `enable` signal is a clock enable that is applied to all flip-flops of the state machine.

Handling Illegal States in an AHDL State Machine

- Logic in an AHDL file will never cause a state machine to enter an illegal state.
- However to avoid illegal state transitions that are caused by glitches, one can force an illegal state to a known legal state.
- One must name all the illegal states and use the WHEN OTHERS clause in the Case Statement to force the required transitions.

  ☞ The WHEN OTHERS clause only applies to states that have been declared but not mentioned in the WHEN clause.

  ☞ For an n-bit machine, 2^n possible states exist. One should add illegal state names until the number of possible states is a power of two.

  e.g.

Department of Electrical
and Computer Engineering

```
SUBDESIGN recover
(
   clk : INPUT;
   go  : INPUT;
   ok  : OUTPUT;
)
VARIABLE
   sequence : MACHINE
              OF BITS (q[2..0])
              WITH STATES (
                 idle,
                 one,
                 two,
                 three,
                 four,
                 illegal1,
                 illegal2,
                 illegal3);
BEGIN
   sequence.clk = clk;
   CASE sequence IS
      WHEN idle =>
         IF go THEN
            sequence = one;
         END IF;
      WHEN one =>
         sequence = two;
     WHEN two =>
         sequence = three;
      WHEN three =>
         sequence = four;
      WHEN OTHERS =>
         sequence = idle;
   END CASE;
```


```
   ok = (sequence == four);
END;
```

- In the above example there are 8 possible states (since there are three bits in the state machine) and consequently three extra states are added and named as illegal states.

☞ The above method only works if all illegal states are defined in the state machine.

State Machines with Synchronous Outputs

- Synchronous outputs occur from a state machine if the outputs only depend on the machine state.
- One can encode the outputs as state values in the WITH STATES clause of the State Machine Declaration.

  e.g.

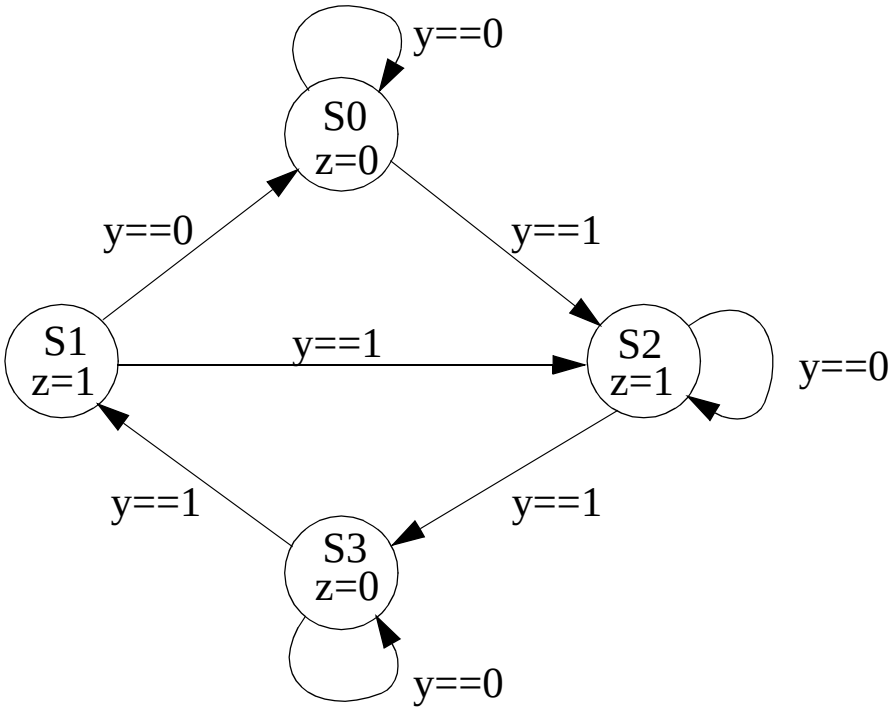


Figure 2 : State diagram of a machine with synchronous outputs.


```
SUBDESIGN moore1
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
VARIABLE
                %   current   current %
                %   state     output  %
   ss: MACHINE OF BITS (z)
       WITH STATES (s0    =   0,
                    s1    =   1,
                    s2    =   1,
                    s3    =   0);
BEGIN
   ss.clk   = clk;
   ss.reset = reset;
   TABLE
   %  current  current     next  %
   %  state    input       state %
      ss,      y      =>   ss;
      s0,      0      =>   s0;
      s0,      1      =>   s2;
      s1,      0      =>   s0;
      s1,      1      =>   s2;
      s2,      0      =>   s2;
      s2,      1      =>   s3;
      s3,      0      =>   s3;
      s3,      1      =>   s1;
   END TABLE;
END;
```

- When state values are used as outputs, the design may use few mac-

rocells, but the macrocells may require more logic to drive their flip-flop inputs.

- Logic synthesizer module may not be able to fully minimise the state machine in these cases.
- Alternate way to define synchronous state machines is to omit state assignments and explicitly declare output flip-flops.

  e.g.

```
SUBDESIGN moore2
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3);
   zd: NODE;
BEGIN
   ss.clk   = clk;
   ss.reset = reset;
   z = DFF(zd, clk, VCC, VCC);
   TABLE
   %  current  current     next     next   %
   %  state    input       state    output %
      ss,      y      =>   ss,      zd;
      s0,      0      =>   s0,      0;
      s0,      1      =>   s2,      1;
      s1,      0      =>   s0,      0;
      s1,      1      =>   s2,      1;
      s2,      0      =>   s2,      1;
      s2,      1      =>   s3,      0;
      s3,      0      =>   s3,      0;
```


```
      s3,      1      =>   s1,      1;
   END TABLE;
END;
```

☞ Don't need a function prototype in the above listing as the DFF is an inbuilt primitive.

- Instead of specifying the output with state value assignments in the State Machine Declaration, this example includes a "next output" column after the "next state" column in the Truth Table Statement. This method uses a D flip-flop (DFF)–called with an in-line reference–to synchronize the outputs with the Clock.

State Machines with Asynchronous Outputs

- Can also implement state machines with asynchronous outputs in AHDL
- The output of an asynchronous state machine can change at any time, regardless of the clock input, as the output is a function of the state flip-flops and the current input. Therefore if the current input changes then the output can change.

☞ Asynchronous state machines are not a good idea from a design point of view, and should generally be avoided,

e.g.

```
SUBDESIGN mealy
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
```

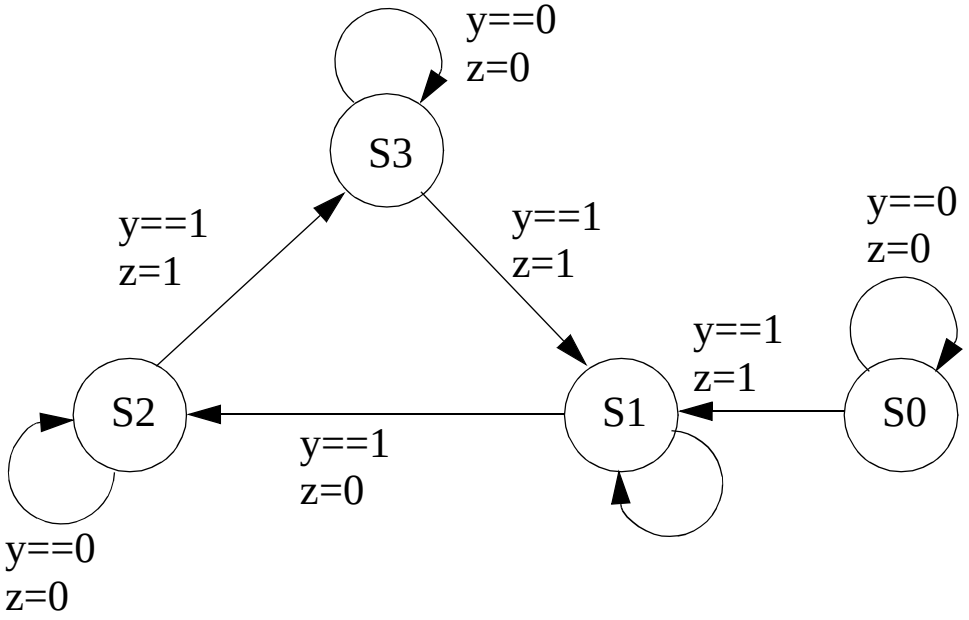


Figure 3 : State diagram of a machine with asynchronous outputs

```
VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3);
BEGIN
   ss.clk = clk;
   ss.reset = reset;
   TABLE
   %  current  current      current  next  %
   %  state    input        output   state %
      ss,      y       =>   z,       ss;
      s0,      0       =>   0,       s0;
      s0,      1       =>   1,       s1;
      s1,      0       =>   1,       s1;
      s1,      1       =>   0,       s2;
      s2,      0       =>   0,       s2;
      s2,      1       =>   1,       s3;
      s3,      0       =>   0,       s3;
      s3,      1       =>   1,       s0;
   END TABLE;
END;
```

Department of Electrical
and Computer Engineering

VHDL

(Very High Speed Integrated Circuit Hardware Description Language)

Notes based on: VHDL Primer (Revised Edition), J. Bhasker, Prentice Hall, 1995, ISBN 0-13-181447-8.

Introduction

- VHDL is a much more complex language than AHDL.
- Designed to be an amalgamation of the following language constructs:
  - sequential language
  - concurrent language
  - netlist language
  - timing specifications
  - waveform generation language

☞ language has constructs that enable one to express the concurrent or sequential behaviour of a digital system with or without timing.

☞ Language not only defines syntax but also defines very clear simulation semantics for each language construct. Therefore models written in this language can be verified using a VHDL simulator.

History

- Requirements first generated in 1981 under the VHSIC program (a program under the auspices of the US DoD).

- Born because of the requirement that several different companies be able to interchange chip designs.
- Version 7.2 of the language was developed by IBM, Texas Instruments, and Intermetrics in 1985.
- Standardized by the IEEE in 1987 (IEEE Std 1076-1987).
- New version of the language standardized in 1993 (IEEE std 1076-1993).
- In 1993 the logic values used were also standardized. The standard is a 9-logic value package called STD_LOGIC_1164, and the standard is IEEE Std 1164-1993.

Capabilities

- Can be used as an exchange medium between chip vendors and CAD tool users
- Can be used as a communication medium between different CAD and CAE tools – e.g. schematic capture may be used to create a design and a VHDL description may be generated. This can then be used as a input into a simulator.
- Language supports hierarchy – digital system can be modelled as a collection of interconnected components, and each component can be modelled as interconnected subcomponents.
- Supports top-down, bottom-up and mixed design philosophies.
- Language is not technology specific, but can support technology specific features.
- Supports synchronous and asynchronous timing models.
- Various digital modelling techniques, such as finite state machine descriptions, algorithmic descriptions, and boolean equations can be modelled using the language.
- Language is publicly available, and is human and machine read-

able.

- It is an IEEE and ANSI standard.
- Language supports three basic different description styles: structural, dataflow and behavioural. Any combination of these may be used in a single design.
- Supports a wide range of abstraction levels ranging from abstract behavioural description to very precise gate level descriptions. Does not support modelling at or below the transistor level.
- Arbitrarily large designs can be modelled.
- Language is structured so that handling large modelling jobs is easier.
- Nominal propagation delays, min-max delays, setup and hold times, timing constraints, and spike detection can all be described very naturally in the language.
- Use of generics and attributes in the models facilitate back-annotation of static information such as timing or placement information.
- Generics and attributes are also useful in describing parameterized designs.
- A model can contain information *about* the design, as well as the design itself. For example, a model can contain information on the total area and speed.
- Models written in this language can be verified by simulation since precise simulation semantics are defined for each language construct.
- Behavioural models that conform to a certain synthesis description style are capable of being synthesized to a gate level description.

VHDL View of a Device

- VHDL model of digital hardware specifies the external view and one or more internal views.
- The external specifies the interface to the device and the internal view specifies the functionality or structure.
- The device to device model mapping is one to many – there are many models for one particular hardware device; e.g. the data transfer into a device may be represented as integers instead of logic values in a high level model See Figure 4.
- Each device model is treated as a distinct representation of a unique device. These are called an *entity* (see Figure 5 below). This figure

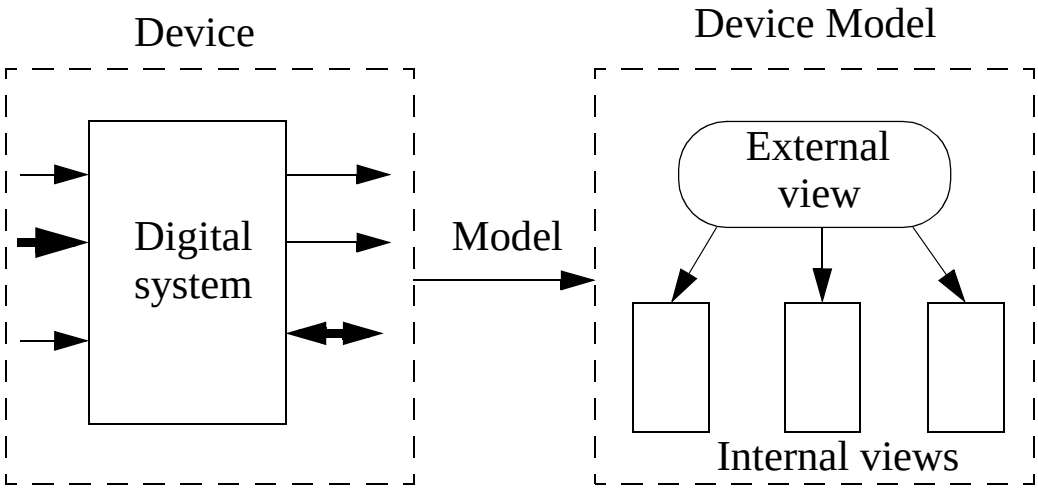
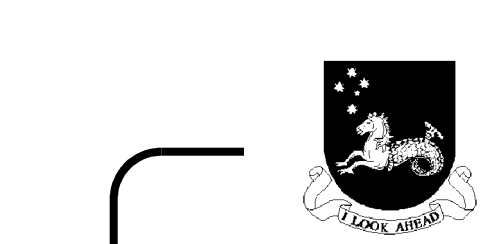


Figure 4 : Device and device model relationship

emphasizes that there is a one to one binding between the VHDL entity structure and a model representation. All the entities however represent the same physical device. Each entity is described using one model, which contains one external view and one or more internal views. The hardware device may be represented by one or more entities.



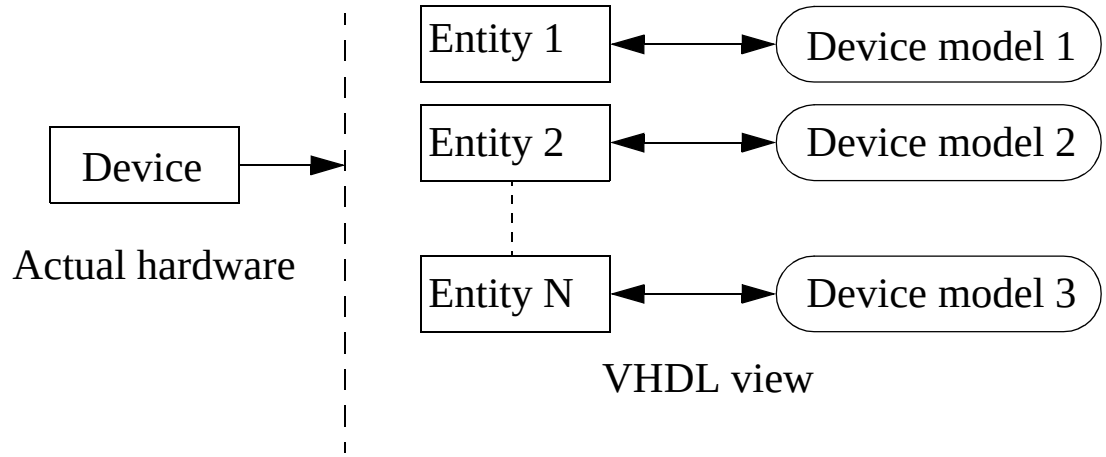


Figure 5 : The VHDL view of a device

A Tutorial

☞ This section introduces the basic features of VHDL. At the end of this section one should be able to write simple VHDL models.

Basic Terminology

- A VHDL abstraction of a digital system is called an *entity.*
- When entity X is used in entity Y, then entity X is said to be a *component* of entity Y.
- VHDL provides five different types of primary constructs called *design units*. They are:

(i) Entity declaration

(ii) Architecture body

(iii) Configuration declaration

(iv) Package declaration

(v) Package body

Entity

- Modelled using an entity declaration and at least one architecture body.
- Entity declaration describes the external view of the entity; e.g. the input and output names of the entity.
- Architecture body contains the internal description of the entity; e.g. the set of interconnected components that describe the structure of the entity, or a set of sequential statements that describe the behaviour of the entity. See Figure 6 shows the relationship between an entity and one possible model.

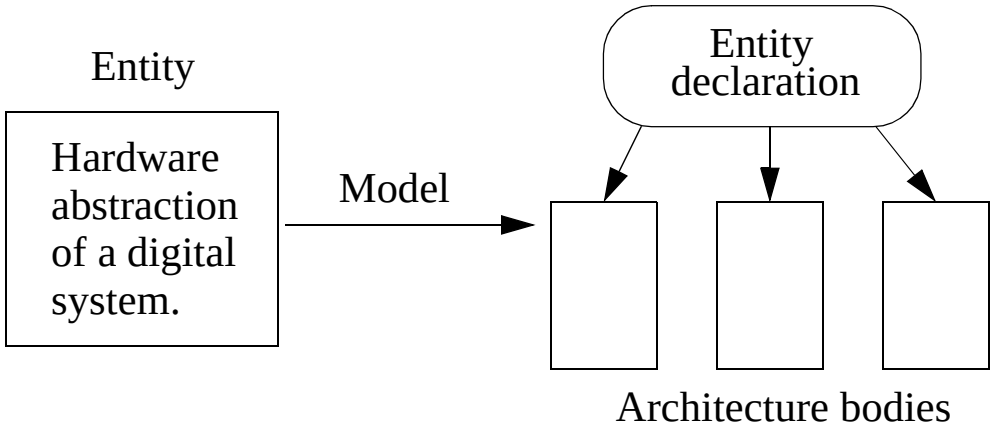


Figure 6 : An entity and its model

Configuration

- Configuration declaration is used to create a configuration for an entity.
- Specifies the binding of one architecture body from the many architecture bodies that may be associated with the entity.
- May also specify the bindings of components used in a selected architecture to other entities.
- An entity may have a number of different configurations.

Package

- Package declaration encapsulates a set of related declarations, such as type declarations, subtype declarations, and subprogram declarations, which can be shared across two or more design units.
- A package body contains the definitions of the subprograms declared in a package declaration.
- In programming terms a package is analogous to a module.

Example of these relationships

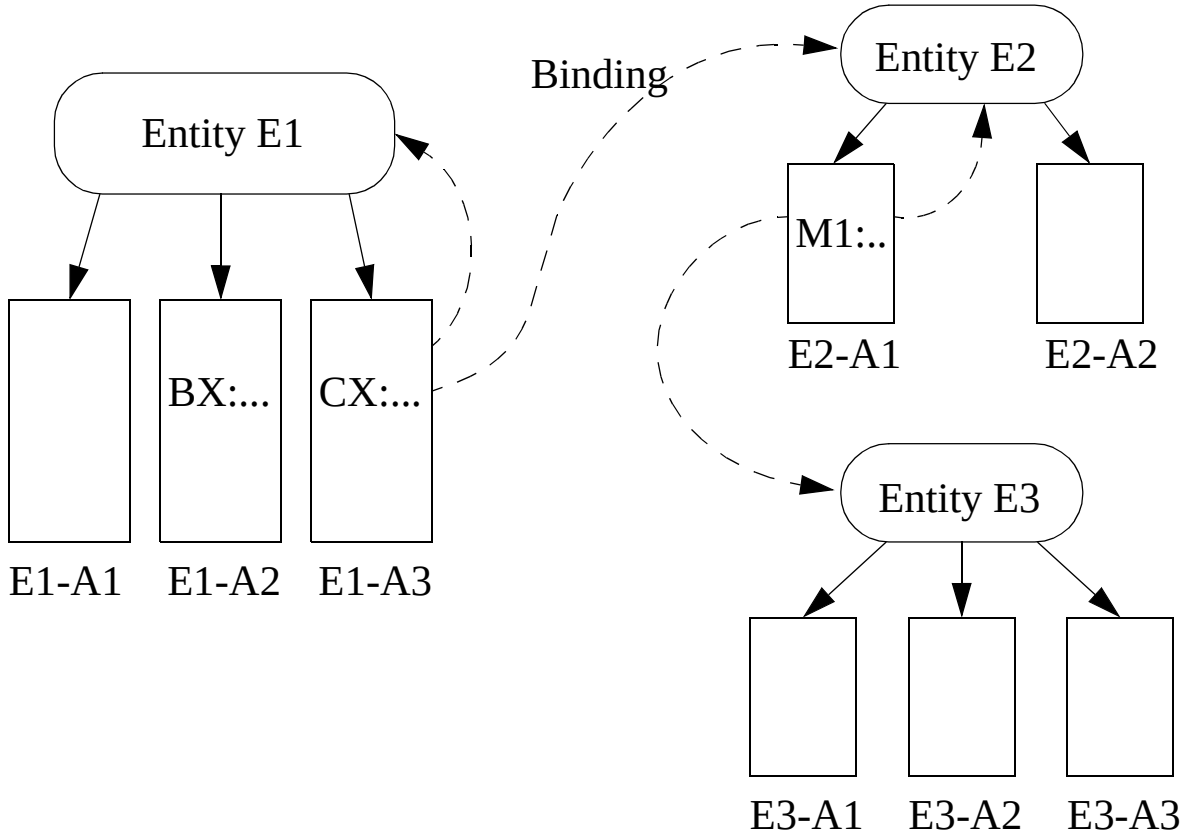


Figure 7 : A configuration for entity E1

- Figure 7 shows three entities called E1, E2 and E3.
- E1 has three architecture bodies, E1-A1, E1-A2, E1-A3.
- E1-A1 is a purely a behavioural model without any hierarchy.
- E1-A2 uses a component called BX, while E1-A3 uses a component called CX.
- Entity E2 has two architecture bodies, E2-A1, E2-A2, with E2-A1 using a component called M1.
- Entity E3 has three architecture bodies – E3-A1, E3-A2, E3-A3.
- Notice that each entity has a single entity declaration but multiple architecture bodies.

- The dashed lines represent bindings that may be specified in the configuration.
- Two types of bindings are shown: binding of an architecture body to its entity, and binding of a component used in an architecture body to another entity.

Design Validation

- Generate the VHDL model using an entities etc.
- The VHDL is then compiled into some intermediate format ready for simulation. This format is not specified by the VHDL standard. This step also validates the syntax of the language and performs some semantic checks.
- The compiled VHDL is then submitted to a simulator for functional testing.

A few syntax notes:

☞ VHDL is a case insensitive language.

☞ The language is free format much the same as Pascal and ‘C’.

☞ ‘--’ is the comment operator. It works the same as the ‘//’ comment operator in C++ in that it comments out the line that it starts on. The comment only goes for one line.

Department of Electrical
and Computer Engineering

Entity Declaration

- Specifies the name of the entity being modelled and lists the set of interface ports.
- Ports are signals through which the entity communicates with the other models in its external environment.

e.g.

```
entity half_adder is
   port(a, b: in bit; sum, carry: out bit);
end half_adder;
-- this is a comment line
```

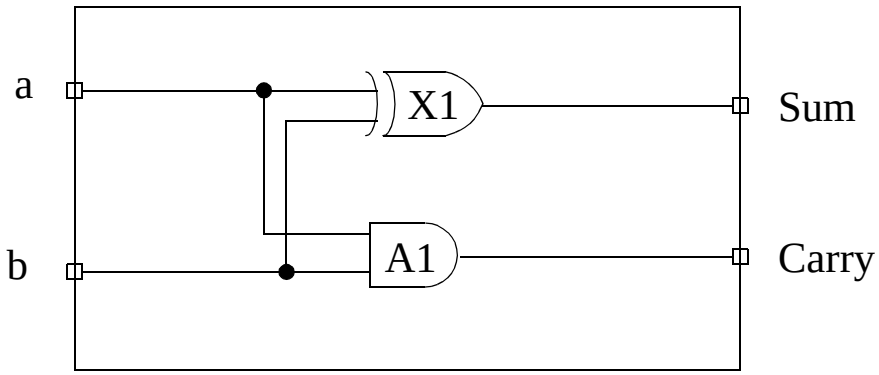


Figure 8 : A half adder

- Entity half adder has two input ports, `a` and `b` (specified by the keyword `in`), and two output ports, `sum` and `carry` (specified by the keyword `out`).
- `bit` is a predefined enumeration type containing the literals ‘0’ and ‘1’. In the context above it is used to indicate that the input and output ports can take the values of ‘0’ and ‘1’.

e.g.

```
entity decoder2x4 is
   port(a, b, enable: in bit;
                    z: out bit_vector(0 to 3));
end decoder2x4;
```

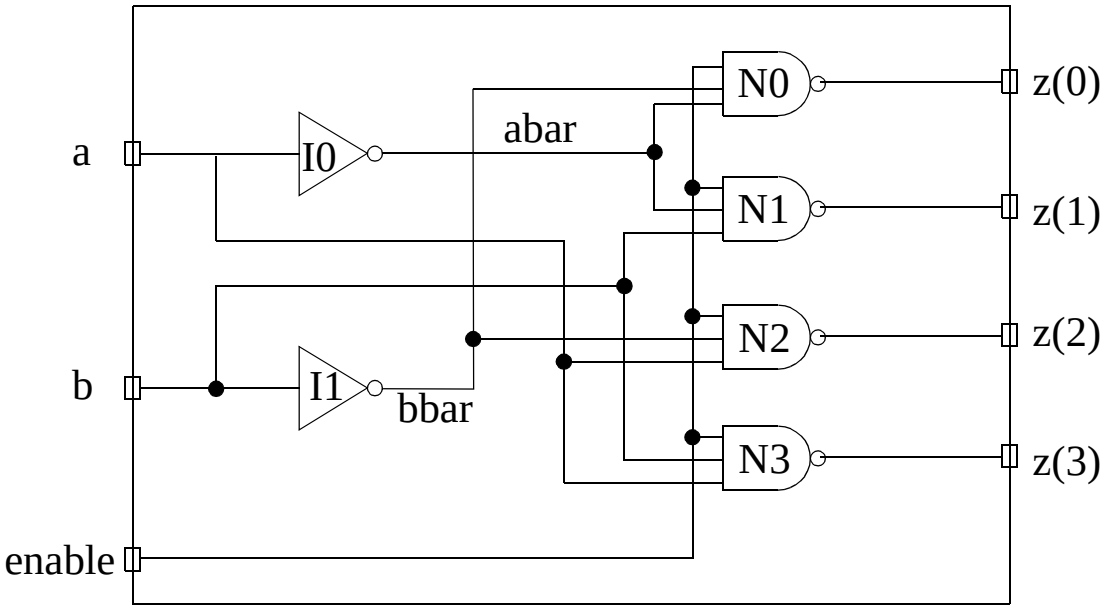


Figure 9 : A 2-to-4 decoder circuit

- Entity called `decoder2x4`: has three input ports and four output ports.
- `bit_vector` is a predefined unconstrained array of type `bit`. Unconstrained means that the size of the array is undefined. In this particular case the size of the array has been set to 4 bits, numbered from 0 to 3.
- In the code examples presented so far only the interface has been specified, and the internals of the entities have not been defined.

Department of Electrical
and Computer Engineering

Architecture Body

- Internal details of an entity are determined by the contents of an architecture body using the following modelling styles:

  (i) As a set of interconnected components (to represent structure).

  (ii) As a set of concurrent assignment statements (to represent data flow).

  (iii) As a set of sequential assignment statements (to represent behaviour).

  (iv) As any combination of the above.

Structural Modelling

- In this type of modelling an entity is modelled as a set of interconnected components.
- For example consider the half adder whose entity code was shown earlier:

```
architecture ha_structure of half_adder is
   component xor2
      port(x, y: 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;
```

- Name of the architecture body is `ha_structure`.

- The entity declaration for `half_adder` specifies the interface ports for this architecture body.
- Architecture body is composed of two parts:

(a) The declarative part (before the keyword `begin`).

(b) The statement part (after the `begin` keyword).

- The declarative part two components are declared: `xor2` and `and2`. These components could come from a library.
- These two components are instantiated in the statement part of the architecture body – `x1` and `a1` are the labels for these instantiations.
- `x1` shows that the signals `a` and `b` are connected to the `x` and `y` input ports of the `xor2` component, while the output is connected to the output port `sum` of the `half-adder` entity.
- Note that the structural representation of the `half_adder` does ***not say anything about its functionality***. Separate entity models would be needed for the components `xor2` and `and2`, each having its own entity declaration and architecture body.

e.g. decoder2x4:

```
architecture dec_str of decoder2x4 is
  component inv
    port(pin: in bit; pout: out bit);
  end component;

  component nand3
    port(d0, d1, d2: in bit; dz: out bit);
  end component;

  signal abar, bbar: bit;
```


```
begin
   v0: inv port map(a, abar);
   v1: inv port map(b, bbar);
   n0: nand3 port map(enable, abar, bbar, z(0));
   n1: nand3 port map(abar, b, enable, z(1));
   n2: nand3 port map(a, bbar, enable, z(2));
   n3: nand3 port map(a, b, enable, z(3));
end dec_str;
```

- In this case the architecture named `dec_str` is associated with entity `decoder2x4`. It therefore inherits the list of ports in the entity declaration.
- In addition to the two components `inv` and `nand3` the architecture body contains two signal declarations `abar` and `bbar` of type `bit`.
- The signal declarations represent wires that are used to connect various components together. Note that the signal declarations are local and cannot be seen outside the architecture body.
- The instantiation of the components is a concurrent statement, therefore the order of these statements is not important.
- The structural style of modelling describes only the interconnections of the components, without implying any functionality for the components. The components are treated as though they are black boxes.
- *The behaviour of the components is not apparent, nor is the functionality of the decoder as a whole*.

Dataflow Modelling

- The flow of data through the entity is expressed primarily using concurrent assignment statements.

- The structure of the entity is not explicitly specified in this modelling style, but it can be implicitly deduced.

e.g.

```
architecture ha_concurrent of half_adder is
begin
   sum <= a xor b after 8 ns;
   carry <= a and b after 4 ns;
end ha_concurrent;
```

- The listing above uses two *concurrent* signal assignment statements. Ordering of the statements is unimportant.
- The "<=" implies an assignment of the value computed on the right hand side to the target signal on the left hand side.
- The concurrent signal statements are *event driven* – the assignment only occurs if there is an event on one of the signals on the right hand side. Events are such things as a change in a signal logic level.
- Delay information is included in the signal assignment statements using `after` clauses. If in event occurs on the right hand side of the assignment at time T, then the right hand side is evaluated. The left hand side of the assignment gets the result of this evaluation after the delay period;

  e.g. in the above the `sum` signal gets the results of an event at time T on `a` or `b` at T+8 ns, and `carry` at T+4 ns.
- Note that the architecture body called `ha_concurrent` is associated with the entity called `half_adder`.

e.g.

```
architecture dec_dataflow of decoder2x4 is
   signal abar, bbar: bit;
begin
   z(3) <= not (a and b and enable);     -- #1
   z(0) <= not (abar and bbar and enable);-- #2
```


```
   bbar <= not b;                              -- #3
   z(2) <= not (a and bbar and enable);        -- #4
   abar <= not a;                              -- #5
   z(1) <= not (abar and b and enable);        -- #6
end dec_dataflow;
```

- Architectural body consists of one signal declaration and six concurrent signal assignment statements.
- Note that after the signal assignment statements no `after` clause has been specified, therefore no delay is explicitly specified. The default is 0 ns (known as the *delta delay*, representing an infinitesimally small delay).
- Let us consider the sequence of events that occur if there is an event on one of the input signals – say input port `b` at time T:

  (i) Concurrent signal assignment statements #1, #3 and #6 are triggered. The RHS of the expressions are evaluated, and the corresponding values would be scheduled to be assigned to the target signals at time ($T + \Delta$).

  (ii) At ($T + \Delta$) the new values are assigned to signals `z(3)`, `bbar`, and `z(1)`.

  (iii) Since bbar changes, this will in turn trigger signal assignment statements #2 and #4. Eventually at ($T + 2\Delta$) signals `z(0)` and `z(2)` will be assigned their new values.

e.g. Use of the after clause to generate a clock signal:

```
   clk <= not clk after 10 ns;
```

This generates a clock with a period of 20 ns.

Department of Electrical
and Computer Engineering

Behaviour Modelling

- Models the behaviour of an entity by executing a set of statements sequentially in the specified order.
- The statements do not specify the structure of the entity but merely its functionality.

e.g. the decoder2x4:

```
architecture dec_sequential of decoder2x4 is
begin
  process (a, b, enable)
    variable abar, bbar: bit;
  begin
    abar := not a;                  -- #1
    bbar := not b;                  -- #2

    if enable = '1' then            -- #3
      z(3) <= not (a and b);        -- #4
      z(0) <= not (abar and bbar);-- $5
      z(2) <= not (a and bbar);   -- #6
      z(1) <= not (abar and b);   -- #7
    else
      z <= "1111";                  -- #8
    end if;
  end process;
end dec_sequential;
```

- `process` statement is a *concurrent* statement.
- `process` statement has a declarative part (before the `begin` key-word) and a statement part (between `begin` and `end process`).
- A `process` never terminates, it only ever becomes suspended waiting on an event in the sensitivity list.
- Statements appearing within the statement part are *sequential* – i.e.

they are executed sequentially (and not concurrently as with the data flow statements).

- List of signals in parentheses after the keyword `process` constitute a sensitivity list – i.e. a list of signals which are monitored for an event. In the above example if an event occurs on `a`, `b`, or `enable` then the content of the process statement is executed sequentially.
- Variable declaration – starts with the keyword `variable`. In the above example two variables are declared: `abar` and `bbar`.
- Variables differ from signals in that the assignment operation always occurs instantaneously. Signals are always assigned their value after a certain delay (user assigned or the default delta delay). Uses the notation "`:=`" to differentiate the operation from signal assignment.
- Variables declared within a process have their scope limited to that process. Variables declared outside of a process or a subprogram are called *shared variables* (can be shared by a number of processes).
- The signal assignment statements in a process are called *sequential assignment statements*. They are executed sequentially independent of whether any signal changes on the right side of the statement.

  e.g.

  If there is an event on `a`, `b` or `enable` then statements #1 and #2 are executed. If `enable = 1` then the statements #4–#7 are executed regardless of whether there were changes on `a` or `b`. If `enable = 0` then statement #8 is executed. At the end of the process execution is suspended waiting for another event on the sensitivity list.
- Possible to use `case` or `loop` statements within a process. Semantics and structure are very similar to those in most high level languages.

- wait statement can also be used within a process. This causes a wait for a user specified time or until an event occurs.

  e.g.

  Clock generated using a process statement and a wait.

```
process
begin
   clk <= '0';
   wait for 20 ns;
   clk <= '1';
   wait for 12 ns;
end;
```

- Above process does not have a sensitivity list because there are explicit wait statements inside the process.
- Note that all processes are executed at least once during the initialisation phase, and will continue until they get suspended.

  e.g.

  Model of a level sensitive flip-flop.

```
entity ls_dff is
   port(q: out bit; clk: in bit);
end ls_dff;

architecture ls_dff_beh of ls_dff is
begin
   process(d, clk)
   begin
      if clk = '1' then
         q <= d;
      end if;
   end process;
end ls_dff_beh;
```

Edge sensitive dff

```
entity es_dff is
   port(q: bit; d, clk: bit);
end es_dff;

architecture es_dff_impl of es_dff is
begin
   process (clk)
      variable old_clk : bit = '0';
   begin
      if old_clk = '0' then
         q <= d;
         old_clk := clk;
      end if;
   end process;
end es_dff_impl;
```

☞ Note that at the start of a simulation the whole process is executed once. Therefore `old_clk` will be assigned the value of '0' initially, and then reassigned the value of `clk` (which would also normally be zero). Subsequent executions of the process (when there is an event on the `clk` signal) will only cause the statements between the `begin` and `end` of the process to be executed.

A note on the difference between dataflow and behavioural modelling.

- Concurrent assignment statements are executed whenever there is an event on a signal on the right hand side of an expression.
- Sequential statements are not event triggered, and are executed in

the sequence they are written in a process.

e.g. consider the following two architecture bodies:

```
architecture seq_sig_assign of fragment1 is
   -- a, b, and z are signals.
begin
   process (b)
   begin
      -- following are sequential assignment
      -- statements

      a <= b;
      z <= a;
   end process;
end;


architecture con_sig_assign of fragment2 is
begin
   -- following are concurrent assignment
   -- statements

   a <= b;
   z <= a;
end;
```

- In `seq_sig_assign` the signal assignments are sequential.
- Consider an event on `b` at time T – first assignment statement executed, and then the second in zero time.
- However, signal `a` is scheduled to get its new value at $T + \Delta$ and `z` is scheduled to be assigned the value of `a` (not `b`) as the value is stored at T and assigned at $T + \Delta$.

- In `con_sig_assign` the two statements are executed concurrently.
- When an event occurs on `b` (at time T), signal `a` gets the value of `b` after the delta delay (at time $T + \Delta$). An event then occurs on signal `a` which then causes the new value of `a` to be saved and scheduled to be assigned to `z` and time $T + 2\Delta$. Therefore the value of `b` is effectively being saved into the `z` signal in this case.

Delta delay revisited

- Delta time delay is the time assigned if no delay is specified.
- Delta time delay is infinitesimally small.
- It is not a real time delay but an artifice to allow the correct ordering of events in a simulation.

  e.g.

  Consider the following example of a chain of three inverters.

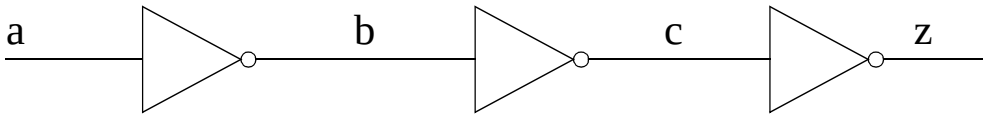


Figure 10 : Three inverting buffers in series

```
entity fast_inverter is
   port(aL in bit; z: out bit);
end;

architecture delta_delay of fast_inverter is
   signal b.c :bit;
begin
   -- the following statements are order
```


```
   -- independent.
   z <= not c;          -- #1
   c <= not b;          -- #2
   b <= not a;          -- #3
end;
```

- When an event occurs on signal a, say at 20ns, then this causes signal b to get the inverted value of a at 20ns + 1Δ. When time advances to 20ns + 1Δ, signal b changes. This in turn triggers a second signal assignment, which causes c to get the inverted value of b after another delta delay, i.e. at 20ns + 2Δ. This signal assignment actually occurs when the time actually advances to 20ns + 2Δ. A similar pattern occurs for the z signal.

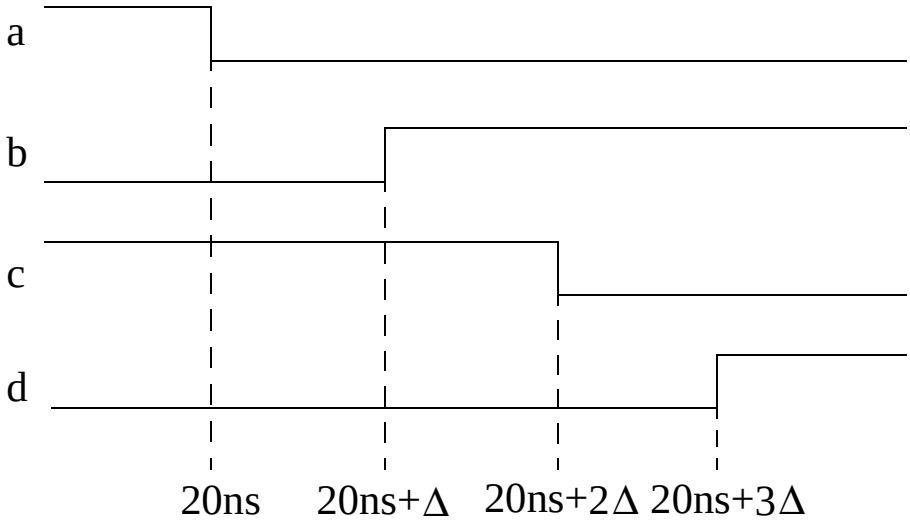


Figure 11 : Delta Delays in an inverter chain with concurrent signal assignment.

Mixed Mode Modelling

- Possible to mix the three modelling styles is single architecture body.

  e.g.

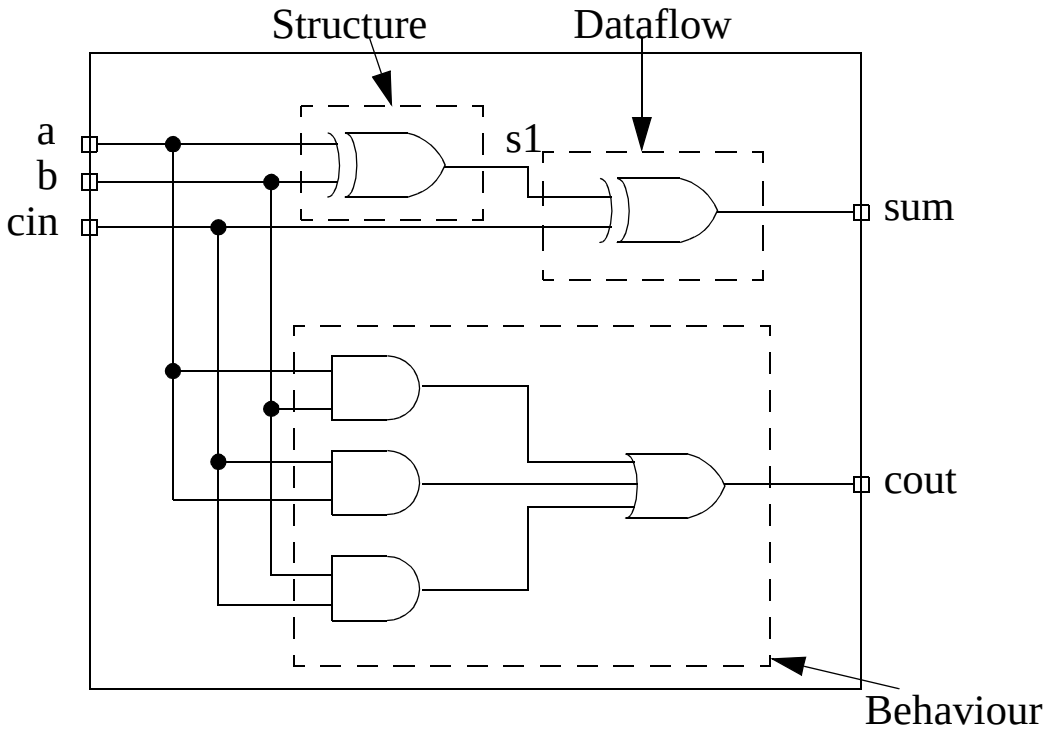


Figure 12 : A 1-bit full adder

The VHDL for Figure 12 is:

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

architecture fa_mixed of full_adder is
   component xor2
      port(p1, p2: in bit; pz: out bit);
   end component;
```


```
   signal s1: bit;

begin
   x1: xor2 port map(a, b, s1);     -- structure
   process(a, b, cin)               -- behaviour
      variable t1, t2, t3: bit;
   begin
      t1 := a and b;
      t2 := b and cin;
      t3 := a and cin;
      cout <= t1 or t2 or t3;
   end process;

   sum <= s1 xor cin;               -- dataflow
end fa_mixed;
```

Department of Electrical
and Computer Engineering

Configuration Declaration

- Used to select the many possible architecture bodies that an entity can have.
- Used to bind components, used to represent structure in an architecture body to entities represented by an entity architecture pair or by a configuration, which reside in a library.

  e.g.

```
library cmos_lib, my_lib;
configuration ha_binding of half_adder is

  for ha_structure

    for x1: xor2
      use entity cmos_lib:xor_gate(dataflow);
    end for;

    for a1: and2
      use configuration my_lib.and_config;
    end for;

  end for;

end ha_binding;
```

- First statement is a library clause that makes the library names `cmos_lib` and `my_lib` visible within the configuration declaration.
- Configuration is called `ha_binding`, and specifies a configuration for the `half_adder` entity.
- Next statement specifies that the architecture body `ha_structure` (described earlier in the structural modelling sec-

tion) is selected for this configuration.

- The `ha_structure` architecture contains two component instantiations, therefore it contains two component bindings:
  - the first for `x1:...end` for, binds the component instantiation with label `x1` to an entity represented by the entity-architecture pair – the `xor_gate` entity declaration, and the dataflow architecture body, which resides in the `cmos_lib` design library.
  - similarly `a1` is bound to a configuration of an entity defined by the configuration declaration, with name `and_config` residing in the `my_lib` design library.
- No behavioural or simulation semantics are associated with a configuration declaration – merely specifies a binding that is used to build a configuration for an entity.
- An architecture body that does not contain any component instantiation (e.g. when dataflow style is used) can also be selected to create a configuration.

  e.g.

  The `dec_dataflow` architecture body can be selected for the `decoder2x4` entity using the following configuration declaration:

```
configuration dec_config of decoder2x4 is
   for dec_dataflow
   end for;
end dec_config;
```

- `dec_config` defines a configuration that selects the `dec_dataflow` architecture body for the `decoder2x4` entity. This represents one possible configuration for the `decoder2x4` entity can now be simulated.

Department of Electrical
and Computer Engineering

Packages

Package Declaration

- Used to store a common set of declarations such as components, types, procedures and functions.
- Packages can be imported into other design units using a `use` clause.

  e.g.

```
package example_pack is
   type summer is (may, jun, jul, aug,sep);

   component d_flip_flop
      port (d, clk: bit; q, qbar: out bit);
   end component;

   constant pin2pin_delay: time := 125ns;

   function int2bit_vec (int_value: integer)
      return bit_vector;

end example_pack;
```

- Name of the package declared is `example_pack`.
- It contains type, component and function declarations.
- N.B. behaviour of `int2bit_vec` does not appear in the package declaration, only the function interface appears. Definition of the function is in the package body.

e.g.

Assume that the above package is compiled into a design library called design_lib:

```
library design_lib;
use design_lib.example_pack.all;
entity rx is .....etc, etc
```

- `library` clause makes the name of the design library `design_lib` visible within this description.
- `use` clause imports all declarations in the package `example_pack` into the entity declaration of `rx`.
- Possible to selectively import declarations from a package declaration into other design units:

  e.g.

```
library design_lib;
use design_lib.example_pack.d_flip_flop;
use design_lib.example_pack.pin2pin_delay;
architecture rx_structure of rx is .....etc, etc.
```

- Other techniques can also be employed to selectively chose declarations within a package.

Package Body

- Used to store the definitions of functions and procedures that were declared in the corresponding package declaration, and the complete constant declarations for any deferred constants that appear in the package declaration.
- Package body is always associated with ***one*** package declaration

  e.g.

  Package body for the package `example_pack`:


```
package body example_pack is
   function int2bit_vec (int_value: integer)
      return bit_vector is
   begin

      -- behaviour of function described here

   end int2bit_vec;
end example_pack;
```

- Name of the package body must be the same as that of the package declaration.

  ☞ A package body is not necessary if the corresponding package declaration has no function or procedure declarations and no deferred constant declarations.

  e.g.

  Another example of a package body:

```
package body another_package is

   -- a complete constant declaration
   constant total_alu: integer := 10;

   function pocket_money
      (month: design_lib.example_pack.summer)
                                 return integer is
   begin
      case month is
         when may => return 5;
         when jul|sep => return 6;
         when others => return 2;
      end case;
```


```
   end pocket_money;
end another_package;
```

Model Analysis using VHDL

- Once an entity is described in VHDL it can be validated using an analyser.
- The analyser takes a file that contains one or more design units (a design unit consisting of an entity declaration, an architecture body, a configuration declaration, a package declaration or a package body), and compiles them into an intermediate form (which is not defined in the standard). During this process the syntax and static semantics are checked. The generated file is stored in a specific design library that has been designated as the working library.
- Design libraries have logically names. the mapping of these names to the physical location where they are stored is carried out by the underlying host operating system.
- An arbitrary number of design libraries may exist simultaneously, one of which is designated as the working library and is given the logical name WORK. The analyser compiles the descriptions in this library, and this is the only library that is updated.
- Items compiled in a different design library can be imported into design units of the current design library by using `library` and `use` clauses, or by accessing them with a selected name.
- Design library with the logical name STD is predefined by the VHDL language environment. Contains two packages: STANDARD and TEXTIO.
- The STANDARD package contains declarations for standard predefined types such as `bit`, `time`, `integer`, etc.
- The TEXTIO package provides support for formatted text read and write operations.

- There is also an IEEE design library with a package in it called STD_LOGIC_1164. Defines a nine value logic type called STD_ULOGIC and associated subtypes, overloaded operator functions and other useful utilities.

Simulation

- After complication into one or more design libraries, next step is validation.
- For hierachical entities to be simulated, all of its lowest components must be described at the behavioural level.
- Simulation can be performed on the following:

  (a) An entity declaration and architecture body pair.

  (b) A configuration.
- There are two major steps before the actual simulation:

  (i) *Elaboration phase*: The hierachy of an entity is expanded and flattened, components are bound to entities in libraries, top level entity is built as a network of behavioural models ready to be simulated. Storage is allocated for signals, variables, and constants declared in design units.

  (ii) *Initialization phase*: Driving and effective values for all explicitly declared signals are computed, implicit signals (not discussed thus far) are assigned values, processes are executed once until they suspend, and simulation time is set to 0ns.
- Simulation commences by advancing time to that of the next event.
- Values that are scheduled to be assigned to signals at this time are assigned.
- If the value of a signal changes, and if that signal is present in the sensitivity list of a process, the process is executed until it suspends.
- Simulation stops when an assertion violation occurs or when the maximum time as defined by the language is reached.

Department of Electrical
and Computer Engineering

Assorted Aspects of VHDL

☞ This section presents a variety of aspects of VHDL not presented so far. The list is far from complete but is intended to give the reader a feel for the language beyond that attained in the initial tutorial.

Basic Language Elements

☞ As one can see from the previous tutorial many aspects of VHDL are similar to those available in languages such as ‘C’ or Pascal. Therefore, some features will only be mentioned assuming that the reader can make these connections. It will be assumed that the reader can guess that obvious features are in the language.

Port Pin Types

- `in`: The value of an input port can only be read with the entity model.
- `out`: The value of an output port can only be updated within the entity model; it cannot be read.
- `inout`: The value of a bidirectional port can be read and updated within the entity model.
- `buffer`: The value of the buffer port can be read and updated within the entity model. However, it differs from the inout mode in that it cannot have more than one source, and the only kind of signal that can be connected to it can be another buffer port or a signal with at most one source. i.e. a signal is driving another element is the design as well as an output port.

Data Objects

- holds a value of a specific type, e.g.

```
variable count: integer;
```

which results in a data object of type `integer` called `count`, which is an object of the `variable` class .

☞ There are **four classes** to which data objects can belong:

(i) *Constant*: data object of this class is assigned a single value at the start of a simulation and this value cannot be changed.

(ii) *Variable*: similar to the constant except that the value of the object can be changed during the course of the simulation using a variable assignment.

(iii) *Signal*: this data object holds a list of values, which includes the current value of the signal, and a set of possible future values that are to appear on the signal. Future values are assigned to the object using the signal assignment statement.

(iv) *File*: an object that contains a sequence of values that can be read and written.

☞ Signals can be regarded as wires in a circuit, whilst variables and constants are the same as their analogues in conventional high level programming languages. Signals typically used to model wires and flip-flops and variables and constants are typically used to model the behaviour of a circuit.

☞ The file object is used to model files in the host environment.

e.g.

Constant declarations


```
constant rise_time: time := 10ns;
constant bus_width: integer := 8;
```

Variable declarations

```
variable ctrl_status: bit_vector (10 downto 0);
variable sum: integer range 0 to 100 := 10;
variable found, done: boolean;
```

Signals declarations

```
signal clock: bit;
signal data_bus: bit_vector (0 to 7);
signal gate_delay: time := 10ns;
```

File declarations

```
-- first two lines declare a file type
type std_logic_file is file of std_logic_vector;
type bit_file is file of bit_vector;

-- file declarations are:
file stimulus: text open read_mode is
                  "/usr/home/reb/design.dat";
file vectors: bit_file is
                    "/usr/home/reb/vecdata.dat";
file pat1, pat2: std_logic_file;
```

Note:

- `text` is a predefined file type.
- the default mode for an open is `read_mode`.
- if no file is specified then the file is not opened during *elaboration*, but is opened during execution by an explicit open command.

☞ Not all objects are explicitly declared as shown above. Some are implicitly declared.

- ports of an entity – these are all signal objects.
- generics of entities (not discussed thus-far) – these are constants.
- formal parameters of functions and procedures – function parameters are constants or signals, procedure parameters can be any type.
- `for` loop increment variables are implicitly declared constants of type `integer` that only exist whilst the loop is being executed.

Data Types

- The set of values that a data object is allowed to have is specified by its *type declaration*.
- Similarly the operations allowed on a data object are defined by its type.

  e.g.

  `integer` is a predefined type with a minimum range defined by the VHDL standard of $-(2^{31}-1)$ to $(2^{31}-1)$, and allowed operations of +, -, / and *.
- Language provides the ability to define new user defined data types (similar to 'C' and Pascal).

Four major types exist in the language:

(i) *Scalar* types: Values belonging to these types appear in a sequential order – e.g. `integer`, `boolean`.

(ii) *Composite* types: These are composed of elements of a single type (i.e. an array type) or elements of different types (i.e. a record type).

(iii) *Access* types: These provide access to objects of a given type (via pointers).

(iv) *File* types: These provide access to objects that contain a sequence of values of a given type.

Subtypes

- A type with a range constraint.
- *Subtype* declarations are used to declare subtypes.

  e.g.

```
subtype my_integer is integer range 48 to 156;
-- digit is not a subtype - it is a user defined
-- enumeration type.
type digit is (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’,
                                ‘7’, ‘8’, ‘9’);
-- a subtype using the base type digit is shown
-- below.
subtype middle is digit range ‘3’ to ‘7’;
```

Scalar Types

- Four different scalar types:

  (i) Enumeration

(ii) Integer

(iii) Physical

(iv) Floating point

Enumeration Types

e.g.

```
type mvl is (‘U’, ‘0’, ‘1’, ‘Z’);
type micro_op is (Load, Store, Add, Sub, Mul,
                                    Div);

signal control_a: mvl;
-- implicit subtype declaration
signal clock: mvl range ‘0’ to ‘1’;
variable ic: micro_op := store;
variable alu: arith_op;
```

Integer Types

- Defines a type whose set of values fall within a specified integer range.

e.g.

```
type index is range 0 to 15;
type word_length is range 31 downto 0;
subtype data_word is word_length range 15
                                    downto 0;

-- declaration using these types:
constant mux_address: index := 5;
signal data_bus: data_word;
```

- `integer` is the only predefined integer type in the language.

Floating Point Types

- Has a set of values in a given range of real numbers.
- Only predefined floating point type is real – implementation dependent range, but at least must cover the range -1.0e38 to +1.0e38.

  e.g.

```
type ttl_voltage is range -5.5 to -1.4;
type real_data is range 0.0 to 31.9;

variable length: real_data range 0.0 to 15.9;
variable l1,l2,l3: real_data range 0.0 to 15.9;

-- alternative
subtype rd16 is real_data range 0.0 to 15.9;

variable length: rd16;
variable l1,l2,l3: rd16;
```

Physical Types

- Contains values that represent measurement of some physical quantity, like time, length, voltage, and current.
- Values of this type are integer multiples of the base unit.

  e.g.

```
type current is range 0 to 1e9
  units
    nA;             -- base unit is nano-ampere
    uA = 1000 nA;   -- micro-amp
    mA = 1000 mA;   -- milli-ampere
    Amp = 1000 mA; -- ampere
  end units;
```


```
subtype filter_current is current range 10 uA to
                                        5 mA;
```

- In the above example 2 uA occupies position 2000 while 100 nA occupies position 100.
- Physical values can be written as integer or floating point numbers.

  e.g.

  100 ns
  10 V
  50 sec
  Kohm -- implies 1 Kohm
  5.2 mA -- equivalent to 5200 uA
  5.6 nA -- is 5 nA. Fractional part is truncated since nA
  -- is the base unit for type current.
  5.2643 uA -- is 5264 nA.

- Only predefined physical type is time – range is $-(2^{31}-1)$ to $(2^{31}-1)$.

Composite Types

- Represents a collection of values.
- Two main types of composite types:

(a) Array types – a collection of values belonging to a single type.

(b) Record types – a collection of values that may belong to different types.

Array types

e.g.

```
type address_word is array (0 to 63) of bit;
```


```
type data_word is array (7 downto 0) of mvl;
type rom is array (0 to 125) of data_word;
type decode_matrix is array (positive range
     10 downto 1, natural range 3 downto 0)
                                 of mvl;
  subtype natural is integer range 0 to
                                   integer'high;
  subtype positive is integer range 1 to
                                   integer'high;
  -- t'high gives the highest value belonging
  -- to type t.

-- object declarations using the above types
variable rom_addr: rom;
variable address_bus: address_word;
variable decoder: decoder_matrix;  -- deferred
                                   -- constant.
variable decode_value: decode_matrix;
```

- Can have arbitrary number of dimensions for an array.
- Can assign an array of the same type to another using an assignment statement.

Record types

e.g.

```
type pin_type is range 0 to 10;
type module is
  record
    size: integer range 20 to 200;
    critical_dly: time;
    no_inputs: pin_type;
    no_outputs: pin_type;
  end record;
```


```
variable nand_comp: module;
   -- nand_comp is an object of record type module
nand_comp := (50, 20 ns, 3, 2);
   -- implies 50 is assigned to size, 20 ns is
   -- assigned to critical_dly, etc.
```

- Can assigned one record to another using a simple assignment statement.

Access Types

- These are pointers to dynamically allocated objects of some other type.
- Similar to pointers in the Pascal and ‘C’ languages.

  e.g.

```
-- module is a record type
type ptr is access module;
type fifo is array (0 to 63, 0 to 7) of bit;
type fifo_ptr is access fifo;
```

- `ptr` is an access type whose values are addresses that point to objects of type module.
- Similar to high level language here is a `null` pointer which does not point to any object.
- An allocator is used to generate a pointer to an object, and to generate the object it self.

  e.g.

```
mod1ptr := new module;
```

- References to access types:

(i) *obj-ptr*.all: Accesses the entire object pointed to by *obj-ptr*, where *obj-ptr* is a pointer to an object of any type.

(ii) *array-obj-ptr(element-index)*: Access the specified array element, where *array-obj-ptr* is a pointer to an array object.

(iii) *record-obj-ptr.element-name*: Accesses the specified record element, where *record-obj-ptr* is a pointer to a record object.

- For every access type there is a deallocate implicitly declared which returns the storage occupied by the object to the host environment.

  e.g.

```
procedure deallocate(p: inout ptr);
procedure deallocate(p: inout fifo_ptr);
```

Incomplete Types

- Possible to have an object that points to an object which has elements that are also access types.
- Allows recursive data types to be defined (similar to linked list data structures in other high level languages).

  e.g. An incomplete type declaration

```
type type-name;
```

- Once an incomplete type has been declared, the type-name can be used in any mutually dependent or recursive access type.
- A corresponding full type declaration must follow later.

  e.g.

```
-- Declare the names of the objects to be pointed
-- to before they are actually declared. This
-- introduces the names so that there will not be
-- compilation errors.
type comp;                  -- record contains name
```


```
                              -- and list of nets its
                              -- connected to.
type net;                     -- record contains net
                              -- name and list of
                              -- components its
                              -- connected to.

type comp_ptr is access comp;
type net_ptr is access net;
constant modmax: integer := 100;
constant netmax: integer := 2500;

type comp_list is array (1 to modmax) of
                                 comp_ptr;
type net_list is array (1 to netmax) of net_ptr;

type comp_list_ptr is access comp_list;
type netlist_ptr is access net_list;

-- Now the full declaration of comp and net
type comp is
   record
      comp_name: string(1 to 10);
      nets: netlist_ptr;
   end record;

type net is
   record
      net_name: string(1 to 10);
      components: complist_ptr;
   end record;
```

Example of self-referential access type:


```
type dfg;
type op_type is (add, sub, mul, div, shift,
                                        rotate);
type ptr is access dfg;
type dfg is
  record
    op_code: op_type;
    succ: ptr;   -- successor in linked list
    pred: ptr;   -- predecessor in linked list
  end record;
```

File Types

- Represent files in the host environment.
- Provide mechanism by which VHDL design communicates with the host environment.

  e.g.

```
type file-type-name is file of type-name;

-- type-name is the type of values contained in
-- the file.
```

  e.g.

```
type vectors is file of bit_vector;
type names is file of string;
```

- To access files one uses a set of access procedures:

  e.g.

```
procedure file_open (
                  status: out file_open_status;
                    file f: file-type-name;
```


```
                    external_name: in string;
                    open_kind: in file_open_kind
                    := read_mode
);

procedure file_close (file f: file-type-name);

procedure read (file f: file-type-name;
                value: out type-name);

procedure write (file f: file-type-name;
                value: in type-name
);
```

Department of Electrical
and Computer Engineering

Complete example using file types

```
entity fa_test is end;

architecture io_example of fa_test is
   component full_add
      port(cin, a, b: in bit; cout, sm: out bit);
   end component;

   subtype string3 is bit_vector(0 to 2);
   subtype string2 is bit_vector(0 to 1);
   type in_type is file of string3;
   type out_type is file of string2;

   file vec_file: in_type
      open read_mode is
                  "/usr/home/reb/vhdl/fadd.vec";

   file result_file: out_type
      open write_mode is
                  "/usr/home/reb/vhdl/fadd.out";

   signal s: string3;
   signal q: string2;

begin
   fa: full_add port map(s(0), s(1), s(2), q(0),
                                          q(1));

   process
      constant propagation_delay: time := 25ns;
      variable in_str: string3;
      variable out_str: string2;
   begin
```


```
    while not endfile(vec_file) loop
       read(vec_file, in_str);
       s <= in_str;
       wait for propagation_delay;
       out_str := q;
       write(result_file, out_str);
    end loop;
    report "Completed processing of all
                                        vectors";
    wait;           -- stop the simulation
  end process;
end io_example;
```

Department of Electrical
and Computer Engineering

Operators

- The operators in the language fall into the following categories:

(a) Logical operators

(b) Relational operators

(c) Shift operators

(d) Adding operators

(e) Multiplying operators

(f) Miscellaneous operators

Logical Operators

- Operate on bit and boolean types and one dimensional arrays of these.
- Operators: `and`, `or`, `nand`, `nor`, `xor`, `xnor`, `not`.

Relational Operators

- These are: =, /=, <, <=, >, >=
- Result of the application of all relational operators is a boolean.
- When applied to array types the comparison is carried out from the left to the right.

Shift Operators

- `sll`, `srl`, `sla`, `sra`, `rol`, `ror`
- Work much the same as shift operators in assembly language.
- Operator on arrays of `bit` or `boolean`.

  e.g.

```
-- Filled with bit'left, which is zero
"1001010" sll 2 is "0101000"
```

Adding Operators

- These are: `+`, `-`, & (the & is a concatenation operator).

Multiplying Operators

- These are: `*`, `/`, `mod`, `rem`

Miscellaneous Operators

- These are: abs, ** (exponentiation).

Department of Electrical
and Computer Engineering

Some Behavioural Modelling Constructs

Wait Statement

- Processes may be suspended by a sensitivity list. They may also be suspended by a `wait` statement.
- Three forms of the wait statement:

(a)wait on *sensitivity list*

(b)wait until *boolean-expression*

(c)wait until *time-expression*

- These three forms may also be combined:

```
wait on sensitivity_list until boolean-expression
                    for time-expression
```

e.g.

```
wait on a, b, c;                    -- statement 1
wait until a = b;                   -- statement 2
wait for 10 ns;                     -- statement 3
wait on clock for 20 ns             -- statement 4
wait until sum > 100 for 50 ms;     -- statement 5
```

- Statement 1 causes the process to suspend until an event occurs on `a` or `b` or `c`.
- Statement 2 causes the process to suspend until the condition `a = b` is true. When an event occurs on `a` or `b` then the condition is evaluated.
- Statement 3 causes the process to suspend for 10 ns when the `wait` statement is executed.
- Statement 4 causes the enclosing process to suspend and then wait for an event to occur. If no event occurs within 20 ns the

process resumes execution with the statement after the `wait`.

- Statement 5 is similar to statement 4 except that the logic condition is tested if there is an event on `sum`. If it does not become true within 50 ms of the `wait` being executed then the process is resumed, else it is resumed when the condition becomes true.

If Statement

- Selects a sequence of statements based on the value of a condition that evaluates to a boolean value.
- Works virtually identically to the `if` statements in common high level languages.

General structure:

```
if boolean-expression then
   sequential-statements

{elsif boolean-expression then   -- can have zero
   sequential-statements}        -- or more elsif
                                 -- clauses

[else                            -- optional else
   sequential-statements]        -- clause

end if;
```

- The conditions statements in the above `if` statement are executed sequentially.

Case Statement

- Much the same as the case or switch statements in Pascal or ‘C’.
- General form of the `case` statement:


```
case expression is
   when choices => sequential-statements
   when choices => sequential-statements
   -- can have any number of cases here
   --
   --
   -- optional default statement
   [when others => sequential-statements]
end case;
```

- The `expression` must be a discrete type or a one dimensional array.
- The choices may be expressed as single values or a range of values by using the `to` word (for a consecutive range) or the `|`(representing the 'or').

Loop Statement

- Used to iterate through a set of sequential statements.
- Three types of iteration schemes:

  (i) **`for identifier in range`**

  (ii)**`while boolean-expression`**

  (iii)**`label: loop`**

e.g.

```
factorial := 1;
for number in 2 to n loop
   factorial := factorial + number;
end loop;


j:= 0; sum := 10;
wh_loop: while j < 20 loop
```


```
   sum := sum * 2;
   j := j + 3;
end loop;


sum := 1; j := 0;
l2: loop
   j := j + 21;
   sum := sum * 10;
   exit when sum > 100;
end loop l2;
```

- Can also use the `next` statement in a loop. This statement results in the skipping of the remainder of the statements in the loop, and resuming with the next iteration at the top of the loop.

  Syntax:

  `next [loop-label] [when condition];`

More on Signal Assignments

Inertial Delay Model

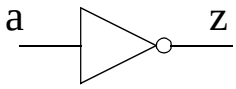
- Used for modelling delays in digital switching circuits.
- Means that an input value must be stable for a specified pulse rejection time before the value is allowed to propagate to the output. In addition the value appears at the output after the specified inertial delay.

  e.g.

```
signal-object <= [reject pulse-rejection-limit]
                 expression after
                              inertial-delay-time;
```

- If no pulse rejection limit is specified, the default rejection limit is the inertial delay itself.

e.g.



delay = 10 ns
reject lim = 4 ns

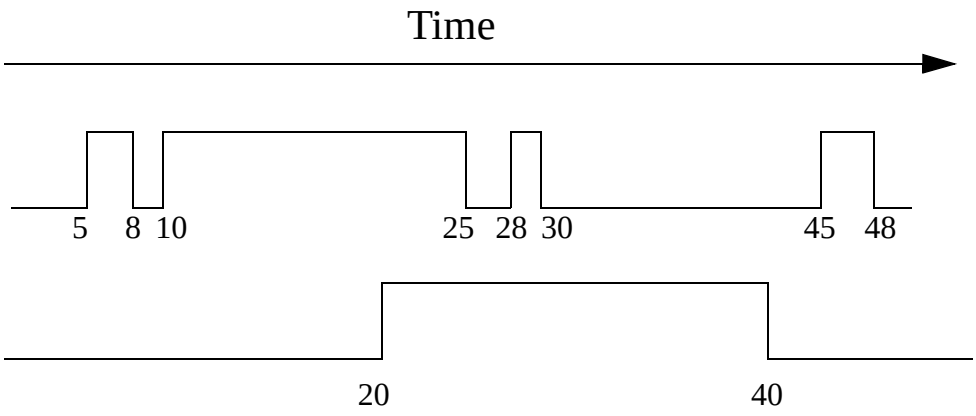


Figure 13 : Inertial/reject signal example

- In the example of Figure 13 the pulses at 5 and 8 ns are not stable long enough to exceed the pulse reject limit, and therefore do not appear at the output of the inverter.
- The edge at 10 ns stays stable for a time exceeding the pulse reject limit and therefore is propagated to the output after the inertial delay time.
- Inertial delay model is the default delay model (see next for another delay model), and therefore does not normally need to be specified with the keyword `inertial`.

Transport Delay Model

- Models pure propagation delay – that is any change on the input is propagated to the output after the delay.
- Usually used to model routing delays, as these don't have a pulse rejection concept as does logic.

e.g.

```
z <= transport a after 10 ns;
```

Creating Waveforms

- Possible to generate arbitrary waveforms with multiple assignment statements.

e.g.

```
phase1 <= '0', '1' after 8 ns, '0' after 13 ns,
          '1' after 50 ns;
```

General syntax:

```
signal-object <= [transport |
       [reject pulse-rejection-limit] inertial]
       expression [after time-expression],
       expression [after time-expression],
       .
       .
       expression [after time-expression];
```

Signal Drivers

- Every signal that is assigned a value in a process has associated with it a *driver.* In fact the *driver* gives a signal (as opposed to a variable) its properties. There is only one driver for a signal in a process.

- A driver holds the current value of a signal as well as all its future values as a sequence.

e.g.

```
process
begin
   .
   .
   reset <= 3 after 5 ns, 21 after 10 ns,
                          14 after 17ns;
end process;
```

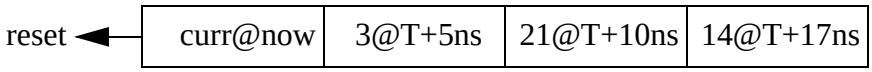


Figure 14 : Signal driver for the signal called reset

- All transactions are order in increasing order of time in the signal driver.
- When time advances to T+5ns then the first entry in the driver is deleted from the list and reset goes to the value of 3.
- At time 10ns then the entry 3@T+5ns is deleted from the list and reset goes to 21, etc.

☞ What happens if there are multiple assignments to a signal within a process? Depends on the delay model used.

Effect of transport delay on signal drivers

- Example of a process with three signal assignments to the same signal.

e.g.

```
signal rx_data: natural;
.
.
process
begin
   .
   .
   rx_data <= transport 11 after 10 ns;
   rx_data <= transport 20 after 22 ns;
   rx_data <= transport 35 after 18 ns;
end process;
```

- When first assignment is executed then 11@T+10ns is added to the driver.
- After second assignment then 20@T+22ns is appended to the driver.
- Third assignment causes 20@T+22ns to be deleted, and 35@T+10ns to be appended.

☞ With transport delay a new signal assignment causes all values in the driver whose delay is the same or longer than that being assigned to be deleted.

Effect of inertial delay on signal drivers

- Situation a little more complex with inertial delays – both the signal value being assigned and the delay value affect the deletion and addition of transactions to the driver.

e.g.

```
process
begin
   -- pulse rejection limit is 10 ns
```


```
   tx_dataj<= 11 after 10 ns;

   tx-data <= reject 15ns 22 after 20 ns;

   -- pulse reject limit is 15 ns
   tx_data <= 33 after 15 ns;

   wait;     -- wait indefinitely
end process;
```

- Transaction 11@10ns first gets added to the driver.
- Second transaction, 22@20ns causes 11@10ns transaction to be deleted. This is because the 11@10ns transaction falls in the pulse rejection period of 20ns back to 5ns (i.e. it is at 10ns), and it is a different value than that at 20ns. In other words the 10ns transaction is now regarded as a glitch and is rejected.
- The third statement causes the 22@20ns transaction to be deleted from the driver, since the delay of the new transaction (15ns) is less than the delay of the transaction of the 20ns transaction already in the driver.

Department of Electrical
and Computer Engineering

Other Aspects of Dataflow Modelling

Multiple Drivers

- Each *concurrent* signal assignment has its own driver.
- Question: What happens if there is more than one *concurrent* assignment to the same signal?

☞ This situation has to be resolved using a resolution function.

e.g.

```
architecture no_enity of dummy is
begin
   z <= '1' after 2 ns, '0' after 5 ns,
             '1' after 10 ns;
   z <= '0' after 4 ns, '1' after 5 ns,
             '0' after 20 ns;
   z <= '1' after 10 ns, '0' after 20 ns;
end no-entity;
```

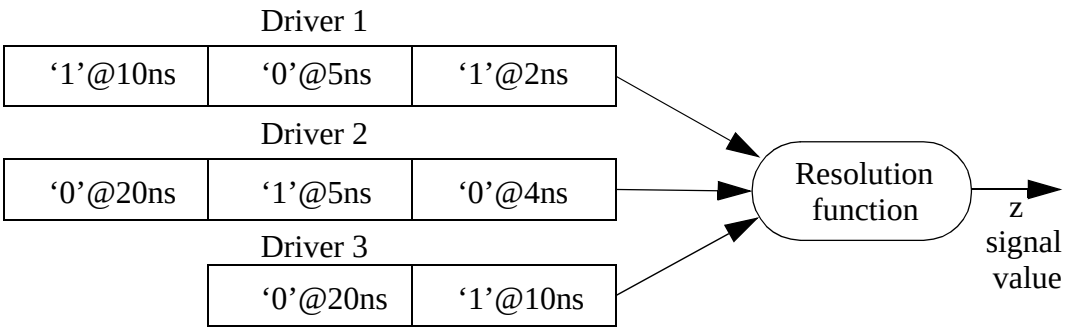


Figure 15 : Resolving signal drivers for dataflow descriptions

- The three drivers in the above example are put into a user written resolution function. The value returned by this function becomes

the resolved value of `z`.

- Have to associate a resolution function with a particular signal.

e.g.

```
signal z: wired_or bit;
```

☞ this associates the resolution function `wired_or` with the signal `z`. The inputs to the function are implicitly the current values of all the drivers for that signal.

e.g. A signal resolution function for the `wired_or`:

```
function wired_or (inputs: bit vector)
                                    return bit is
begin
   for j in inputs'range loop
      if inputs(j) = '1' then
         return '1';
      end if;
   end loop;
   return '0';
end wired_or;
```

Department of Electrical
and Computer Engineering

Aspects of Structural Programming

- As mentioned in the tutorial the entity is modelled as a set of components connected by signals.
- Behaviour of the entity is not explicitly apparent from its model.

e.g.

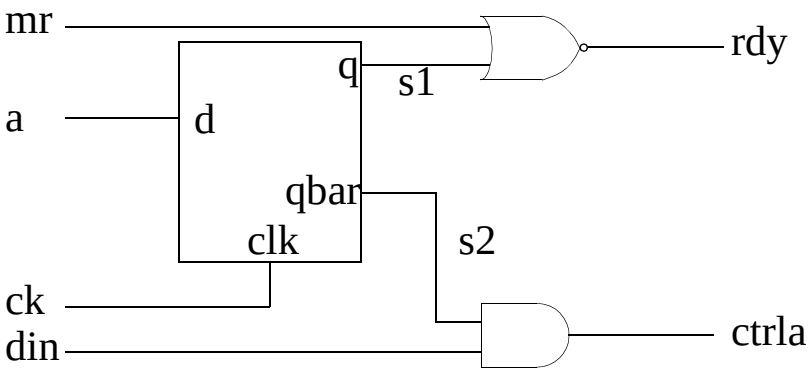


Figure 16 : A circuit generating control signals

VHDL model for the circuit of Figure 16 is:

```
entity gating is
   port (a, ck, mr, din: in bit; ctrla: out bit);
end gating;


architecture structure_view of gating is
   component and2
      port(x, y: in bit; z: out bit);
   end component;

   component dff
      port(d, clock: in bit; q, qbar: out bit);
   end component;
```


```
   component nor2
      port(da, db: in bit; dz: out bit);
   end component;

   signal s1, s2: bit;

begin
   d1: dff port map(a, ck, s1, s2);
   a1: and2 port map(s2, din, ctrla);
   n1: nor2 port map(s1, mr, rdy);
end structure_view;
```

- Instead of declaring the components in the architecture body, one can also use a package:

```
package comp_list is
   component and2
      port(x, y: in bit; z: out bit);
   end component;

   component dff
      port(d, clock: in bit; q, qbar: out bit);
   end component;

   component nor2
      port(da, db: in bit; dz: out bit);
   end component;
end comp_list;
```

- The previous structural description now becomes:

```
library des_lib;
use des_lib.comp_list.all;
architecture structure_view of gating is
   signal s1, s2: bit;
```


```
begin
   -- component instantiations here
end structure_view;
```

e.g. Another example

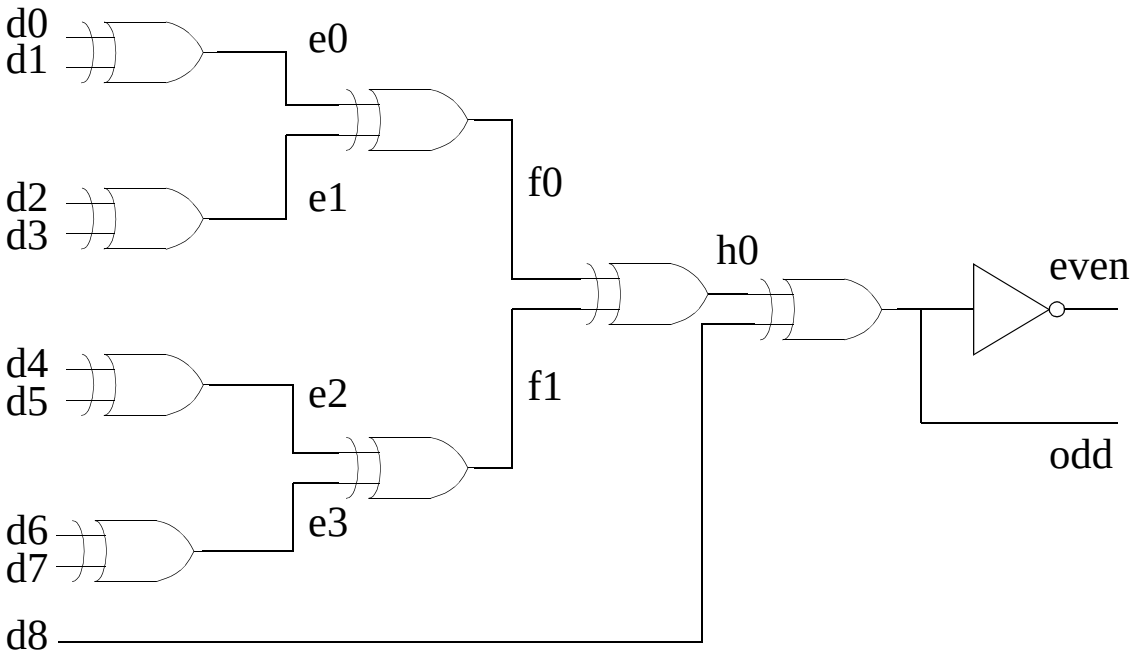


Figure 17 : A 9 bit parity generator circuit

```
entity parity_9_bit is
   port(d: in bit_vector(8 downto 0);
        even: out bit;
        odd: buffer bit);
end parity_9_bit;

architecture parity_str of parity_9_bit is
   component xor2
      port(a, b: in bit; z: out bit);
   end component;
```


```
   component inv2
      port(a: in bit; z: out bit);
   end component;

   signal e0, e1, e2, e3, f0, f1, h0: bit;

begin
   xe0: xor2 port map(d(0), d(1), e0);
   xe1: xor2 port map(d(2), d(3), e1);
   xe2: xor2 port map(d(4), d(5), e2);
   xe3: xor2 port map(d(6), d(7), e3);
   xf0: xor2 port map(e0, e1, f0);
   xf1: xor2 port map(e2, e3, f1);
   xh0: xor2 port map(f0, f1, h0);
   xodd: xor2 port map(h0, d(8), odd);
   xeven: inv2 port map(odd, even);
end parity_str;
```

☞ Port `odd` is of mode `buffer` since the value of this port is being read as well as written to inside the architecture.

Department of Electrical
and Computer Engineering

Generics and Configurations

Generics

- Often useful to pass information into an entity from its environment.

  e.g. rise and fall times, size of interface ports.

- The use of generics allows general purpose user configurable components to be easily constructed. These parts can then be put into a library.

e.g. and n input generic and gate.

```
entity and_gate is
   generic (n: natural);
   port(a: in bit_vector(1 to n); z: out bit);
end and_gate;

architecture generic_ex of and_gate is
begin
   process(a)
      variable and_out: bit;
   begin
      and_out := '1';
      for k in 1 to n loop
         and_out := and_out and a(k);
         exit when and_out = '0';
      end loop;

      z <= and_out;
   end process;
end generic_ex;
```

- Rules of generics:

- Declares a constant object of mode in (that is it can only be read) and can only be used in the entity declaration and the corresponding architecture bodies.
- The value of the constant can be specified as a globally static expression in one of the following:

  (a) Entity declaration

  (b) Component declaration

  (c) Component instantiation

  (d) Configuration specification

  (e) Configuration declaration
- Value of generic must be determinable at elaboration time.

• One can also specify a default value for a generic:

e.g.

```
entity nand_gate is
   generic(m: integer := 2);
   port(a: in bit_vector(m downto 1);
                                z: out bit);
end nand_gate;
```

☞ There are many other ways of constructing generics. For the sake of brevity these will not be presented here. However the material presented above gives one some idea of the flexibility that they give.

☞ One can surmise that VHDL generics can be used to build repetitive structures from some fundamental building blocks.

Department of Electrical
and Computer Engineering

Configurations

- Why have configurations? Two main reasons:

  (i) Sometimes convenient to specify multiple views for a single entity and use any one of these for simulation. Achieved by using one architecture body for each view and then using a configuration to bind the entity to the desired architecture body.

  (ii) Sometimes desirable to associate a component with any one of a set of entities. The component may have its name and the names, types and number of ports and generics different from those of its entities.

  e.g.

```
component or2
  port(a, b: in bit; z: out bit);
end component;
```

and the entities that the above component may possibly be bound to are:

```
entity or_generic is
  port(n: out bit; l, m: in bit);
end or_generic;

entity or_hs is
  port(x, y: in bit; z: out bit);
end or_hs;
```

The component names and the entity names, as well as the port names, and their are different. We may be interested in using the `or_hs` entity for the `or2` component, and in another case, the `or_generic` entity. This can be achieved by appropriately specifying a configuration for the component. The advantage is that when components are used in a design, arbitrary names for components and their interface ports can be used, and these can later be

bound to specific entities prior to simulation.

☞ We are not going to present any more on this issue. Clearly there is considerably more involved.

Wrap-up

- These notes have attempted to provide an introduction to both AHDL and VHDL.
- Because of the nature of the languages the introduction is incomplete – this is especially true for VHDL, which is a very large language.
- The introductory material gives the student some feel for the languages and allows simple programs to be written.