Skeleton of a Basic VHDL Program

This slide set covers the components to a basic VHDL program, including lexical elements, program format, data types and operators

A VHDL program consists of a collection of **design units**

Each program contains at least one **entity declaration** and one **architecture body**

Design units can NOT be split across different files

Entity Declaration

```
entity entity_name is
  port(
      port_names: mode data_type;
      port_names: mode data_type;
      ...
      port_names: mode data_type;
  );
end entity_name;
```

Skeleton of a Basic VHDL Program

The *mode* component can be **in**, **out** or **inout** (for bi-directional port)

```
entity even_detector is
  port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic);
end even_detector;
```

A common mistake with *mode* is to try to use a signal of mode **out** as an *input signal* within the architecture body

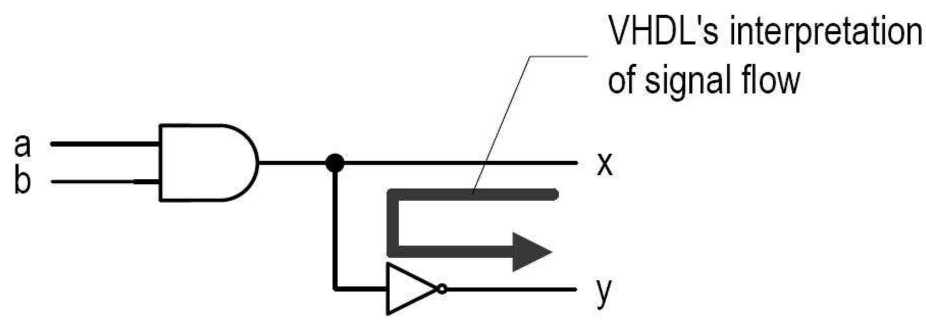
Consider:

```
library ieee;
use ieee.std_logic_1164.all;
entity mode_demo is
  port(
      a, b: in std_logic;
      x, y: out std_logic);
end mode_demo;
```

Skeleton of a Basic VHDL Program

```
architecture wrong_arch of mode_demo is
  begin
  x <= a not b;
  y <= not x;
end wrong_arch;
```

Since *x* is used to obtain *y*, VHDL considers *x* as an external signal that 'flows into' the circuit



Since *x* is declared as an **out** signal, this generates a syntax error

One solution is to change *x* to **inout**, but *x* is really not a bi-directional signal

This is bad practice

Skeleton of a Basic VHDL Program

The correct solution is to declare an *internal* signal as follows

```
architecture ok_arch of mode_demo is
  signal ab: std_logic;
  begin
  ab <= a and b;
  x <= ab;
  y <= not ab;
end ok_arch;
```

Architecture Body

The architecture body specifies the internal organization of a circuit

```
architecture arch_name of entity_name is
  declarations
  begin
  concurrent_stmt;
  concurrent_stmt;
end arch_name;
```

Skeleton of a Basic VHDL Program

The *declaration* part is optional and can include **internal signal declarations** or **constant declarations**

There are many possibilities for *concurrent_stmts*, which we will cover soon

Other **design units** (beyond **entity** and **architecture**) include

- Package declaration & body

  A *package* is a collection of commonly used items, such as data types, subprograms and components

- Configuration

  An entity declaration can be associated with **multiple** architecture bodies

  A *configuration* enables one of them to be instantiated during synthesis

A *VHDL library* is a place to store design units

The default library is 'work'

Skeleton of a Basic VHDL Program

IEEE has developed several VHDL packages, e.g., *std_logic_1164* and *numeric_std* packages

To use them, you must include the **library** and **use** statements

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

The first line invokes a library named *ieee*

The second line makes *std_logic_1164* package visible to the subsequent design units

This package is heavily used and is needed for the std_logic/std_logic_vector data type

Processing of VHDL code occurs in **three** stages

- Analysis: compiler checks each design unit for correct syntax and for some static semantic errors

  If no errors are found, the compiler translates the unit into an *intermediate form* and stores it in a designated library

Skeleton of a Basic VHDL Program

- Elaboration: binds architectures to entities using configuration data

  Many complex designs are coded in a hierarchical manner

  Compiler starts with designated top-level component and replaces all instantiates sub-components with their architecture bodies to create a single flattened description

- Execution

  The flattened design is used as input to a simulation or synthesis engine

Lexical Elements and Program Format

Lexical elements are basic syntactical units in a VHDL program and include

- Comments
- Identifiers
- Reserved words
- Numbers
- Characters
- Strings

Lexical Elements and Program Format

Comments start with two dashes, e.g.,

```
-- This is a comment in VHDL
```

An **identifier** can only contain alphabetic letters, decimal digits and underscore; the first character *must be a letter* and the last character **cannot** be an underscore

Also, two successive underscores are not allowed

Valid examples:

A10, next_state, NextState, mem_addr_enable

Invalid examples:

sig#3, _X10, 7segment, X10_, hi_ _there

VHDL is case **IN**sensitive, i.e., the following identifiers are the same

nextstate, NextState, NEXTSTATE, nEXTsTATE

Use CAPITAL_LETTERs for constant names and the suffix _n to indicate active-low signals

Lexical Elements and Program Format

VHDL Reserved Words

abs access after alias all and architecture array assert
attribute begin block body buffer bus case component
configuration constant disconnect downto else elsif end
entity exit file for function generate generic guarded
if impure in inertial inout is label library linkage
literal loop map mod nand new next nor not null of on
open or others out package port postponed procedure
process pure range record register reject rem report
return rol ror select severity signal shared sla sll
sra srl subtype then to transport type unaffected units
until use variable wait when while with xnor xor

Numbers can be written in several forms:

Integer: 0, 1234, 98E7

Real: 0.0, 1.23456 or 9.87E6

Base 2: 2#101101#

Lexical Elements and Program Format

Character:

'A', 'Z', '1'

Strings

"Hello", "101101"

Note, the following are different

0 and '0'

2#101101# and "101101"

VHDL is 'free-format': blank space, tab, new-line can be freely inserted

```
library ieee; use ieee.std_logic_1164.all;entity
even_detector is port(a: in std_logic_vector(2
downto 0);even: out std_logic);end even_detector;
architecture eg_arch of even_detector is signal p1,
p2, p3, p4: std_logic; begin even <= (p1 or p2) or
(p3 or p4); p1 <= (not a(0)) and (not a(1)) and
(not a(2)); p2 <= (not a(0)) and a(1) and a(2);
p3 <= a(0) and (not a(1)) and a(2); p4 <= a(0) and
a(1) and (not a(2)); end eg_arch;
```

BAD IDEA!

Lexical Elements and Program Format

Headers are a GOOD idea

```
--****************************************************
--
-- Author: p chu
--
-- File: even_det.vhd
--
-- Design units:
--    entity even_detector
--       function: check even # of 1s from input
--       input: a
--       output: even
--    architecture sop_arch:
--       truth-table based sum-of-products
--       implementation
--
-- Library/package:
--    ieee.std_logic_1164: to use std_logic
--
-- Synthesis and verification:
--    Synthesis software: . . .
--    Options/script: . . .
--    Target technology: . . .
--    Test bench: even_detector_tb
--
-- Revision history
--   Version 1.0:
--   Date: 9/2005
--   Comments: Original
--
--****************************************************
```

VHDL Objects

A *object* is a named element that holds a value of specific data type; there are four kinds of objects

- Signal
- Variable
- Constant
- File (cannot be synthesized)

And a related construct

- Alias

Signal: Declaration

```
signal signal_name, signal_name, ... : data_type
```

Signal assignment:

```
signal_name <= projected_waveform;
```

Are interpreted as *wires*

Ports in entity declaration are considered signals

VHDL Objects

Variable

Concept found in traditional programming languages, in which a name represents a symbolic memory location where a value can be stored and modified.

NO direct mapping between a variable and a hardware component

Declared and used only inside a process

Variable declaration:

```
variable variable_name, ... : data_type
```

Variable assignment:

```
variable_name := value_expression;
```

Contains **no timing information** (immediate assignment) -- no waveform is possible

Both signals and variables can be assigned initial values

Although useful in simulations, synthesis canNOT deal with them

VHDL Objects

Constant

Value cannot be changed, used to enhance readability

Constant declaration:

```
constant const_name, ... : data_type := value_expr;
```

E.g.,

```
constant BUS_WIDTH: integer := 32;
constant BUS_BYTES: integer := BUS_WIDTH/8;
```

It is a good idea to avoid "hard literals"

```
architecture beh1_arch of even_detector is
  signal odd: std_logic;
  begin ...
  tmp := '0';
  for i in 2 downto 0 loop
     tmp := tmp xor a(i);
  end loop;
```

VHDL Objects

Better way to do it

```
architecture beh1_arch of even_detector is
  signal odd: std_logic;
  constant BUS_WIDTH: integer := 3;
  begin
  ...
  tmp := '0';
  for i in (BUS_WIDTH - 1) downto 0 loop
     tmp := tmp xor a(i);
  end loop;
```

Alias

Not a object, but rather an alternative name for an object used to enhance readability

E.g.,

```
signal: word: std_logic_vector(15 downto 0);
```

VHDL Objects

```
alias op: std_logic_vector(6 downto 0) is
  word(15 downto 9);
alias reg1: std_logic_vector(2 downto 0) is
  word(8 downto 6);
alias reg2: std_logic_vector(2 downto 0) is
  word(5 downto 3);
alias reg3: std_logic_vector(2 downto 0) is
  word(2 downto 0);
```

Data type and operators

We'll consider data types and operators in each of

- Standard VHDL
- IEEE1164_std_logic package
- IEEE numeric_std package

Data Type: defined as

- A set of values that an object can assume
- A set of operations that can be performed on objects of this data type

Data Types and Operators

VHDL is a **strongly-typed** language

An object can only be assigned with a value of its type

Only the operations defined with the data type can be performed on the object

Rational for doing so is to catch errors early in design, i.e., the use of a character data type in an arithmetic operation

Data types in **standard VHDL**

There are about a *dozen* predefined data types in VHDL, but we will focus on only the following for synthesis

- integer:
  - Minimal range: -(2^31-1) to 2^31-1
  - Two **subtypes**: natural, positive
- boolean: (false, true)
- bit: ('0', '1')
- bit_vector: a one-dimensional array of *bit*

Data Types and Operators

The *bit* type is not versatile enough to handle other hardware values, high impedance (tri-state) and wired-or structures (shorting)

We'll see *std_logic* defined later to handle this problem

Data types such as *bit* and *bit_vector* are called **enumeration** data types since their values are enumerated in a list

Operators

There are about 30 operators in VHDL

Under the rules of a strongly-typed language, only certain data types can be used with a given operator

These are defined in the tables that follow, which are derived from VHDL-93

The *shift* and **xnor** operators are NOT defined in VHDL-87, or supported by the IEEE 1076.6 RTL synthesis standard

The tables list ONLY the synthesis-related operators

Data Types and Operators

operator	description	data type of operand a	data type of operand b	data type of result
a ** b	exponentiation	integer	integer	integer
abs a	absolute value	integer		integer
not a	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
a * b	multiplication	integer	integer	integer
a / b	division			
a **mod** b	modulo			
a **rem** b	remainder			
+ a	identity	integer		integer
- a	negation			
a + b	addition	integer	integer	integer
a - b	subtraction			
a & b	concatenation	1-D array, element	1-D array, element	1-D array
a **sll** b	shift left logical	bit_vector	integer	bit_vector
a **srl** b	shift right logical			
a **sla** b	shift left arithmetic			
a **srl** b	shift right arithmetic			
a **rol** b	rotate left			
a **ror** b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a **and** b	and	boolean, bit, bit_vector	same as a	boolean, bit, bit_vector
a **or** b	or			
a **xor** b	xor			
a **nand** b	nand			
a **nor** b	nor			
a **xnor** b	xnor			

Not automatically synthesizable

Precedence	Operator
Highest	** **abs** **not**
	* / **mod** **rem**
	+ - (ident/neg)
	& + - (add/sub)
	sll srl sla sra rol ror
Lowest	**and or nand nor xor xnor**

Note: **and** and **or** have SAME precedence -- use parenthesis!

Data Types and Operators

IEEE std_logic_1164 package: new data types

```
std_logic, std_logic_vector
```

To use:

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

The **std_logic** consists of 9 possible values

```
'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
```

- '0', '1': forcing logic 0 and forcing logic 1
- 'Z': high-impedance, as in a tri-state buffer
- 'L' , 'H': weak logic 0 and weak logic 1, as in wired-logic
- 'X', 'W': *unknown* and *weak unknown* (signal reaches an intermediate voltage value that can NOT be interpreted as either a logic *0* or logic *1*)
- 'U': for uninitialized (simulation only -- signal has not yet been assigned a value)
- '-': don't-care.

Only '0', '1' and 'Z' are used in synthesis ('L' and 'H' rarely used - wired logic rare)

Data Types and Operators

The **std_logic_vector** is an array of elements with *std_logic* data type

E.g.,

```
signal a: std_logic_vector(7 downto 0);
```

Most significant bit is 'labeled' 7 -- best representation for numbers

Another form can be used in cases where you are not representing numbers, but rather control signals

```
signal a: std_logic_vector(0 to 7);
```

Bits or a range of bits can be referenced as

```
a(1)
a(7 downto 3)
```

VHDL support **overloading** of operators, in which the same operator can be used with different data types

Which standard VHDL operators can be applied to *std_logic* and *std_logic_vector*?

Data Types and Operators

The logical operators are overloaded in the *std_logic_1164* package

overloaded operator	data type of operand a	data type of operand b	data type of result
not a	std_logic_vector std_logic		same as a
a **and** b a **or** b a **xor** b a **nand** b a **nor** b a **xnor** b	std_logic_vector std_logic	same as a	same as a

But the arithmetic operators are NOT!

We'll take a look at conversions between signed and unsigned, which do allow arithmetic operations, later in this slide set.

Data Types and Operators

Several operators are defined over the 1-D array data type, including *concatenation*, *relational* and *array aggregate*

Relational

Operands must have the same element type but their **lengths may differ**

Two arrays are compared element by element, from left to right, until a result is established

Shorter array is considered 'smaller' if end is reached before a decision is made

All of the following return true

```
"011"="011", "011">"010", "011">"00010", "0110">"011"
```

Be careful -- this always returns *false* is *sig1* is shorter than *sig2*

```
if (sig1 = sig2) then
```

Data Types and Operators

Concatenation operator (&)

Very useful operator -- can be used to *shift* elements

```
y <= "00" & a(7 downto 2);
y <= a(7) & a(7) & a(7 downto 2);
y <= a(1 downto 0) & a(7 downto 2);
```

Array aggregate (is not a VHDL operator)

It is a VHDL construct to assign a value to an array-typed object

```
a <= "10100000";
-- positional association
a <= (7=>'1', 6=>'0', 0=>'0', 1=>'0', 5=>'1',
      4=>'0', 3=>'0', 2=>'1');
-- named association
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
-- useful to cover remaining possibilities
a <= (7|5=>'1', others=>'0');
a <= (7 downto 3 => '0') & b(7 downto 5);
```

Data Types and Operators

So we can replace the first assignment with the second and not be concerned about changes in the length of *a*

```
a <= "00000000"
a <= (others=>'0');
```

IEEE numeric_std package

Standard VHDL and the *std_logic_1164* package support arithmetic ops only on *integer* data types:

```
signal a, b, sum: integer;
. . .
sum <= a + b;
```

But this is difficult to realize in hardware because integer does NOT allow the range (number of bits) to be specified

We certainly don't want a 32-bit adder when an 8-bit adder would do

The *numeric_std* package allows an array of 0's and 1's to be interpreted as an *unsigned* or *signed* number.

Data Types and Operators

Two new data types:

unsigned and **signed**

Both are defined as an array of elements with *std_logic* data type

For *signed*, the array is interpreted in **2's-compliment** format, with the MSB as the sign bit

Therefore, all of *std_logic_vector*, *signed* and *unsigned* are arrays of *std_logic* data type

But they are treated as independent data types in VHDL

This makes sense because they are interpreted differently, e.g., the bits "1100" represent 12 when interpreted as an unsigned number but -4 as a signed number

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
signal x, y: signed(15 downto 0);
```

Data Types and Operators

The goal of the *numeric_std* package is to support the arithmetic operations

The package *overloads* the operators **abs**, *, /, **mod**, **rem**, + and -

These operators can now take two operands with data types

- *unsigned* and *unsigned*
- *unsigned* and *natural*
- *signed* and *signed*
- *signed* and *integer*

The following are valid

```
signal a, b, c, d: unsigned(7 downto 0);
...
a <= b + c;
d <= b + 1;
e <= (5 + a + b) - c;
```

Note that the sum "wraps around" when overflow occurs, which models the physical adder

Data Types and Operators

The relational operators, =, /=, <, >, <=, >=, are also overloaded

The overloading **overrides** the left-to-right, element-by-element comparison procedure

Instead, the two operands are treated as binary numbers

For example:

```
-- return false if operands are either std_logic_vector
-- or unsigned
"011" > "1000"

-- but returns true if operands are signed because 3 is
greater than -8!
```

The *numeric_std* package also supports new functions, as in the following tables

Overload Operator and New Functions in IEEE *numeric_std* package

Overloaded Operators

overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a - a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

New Functions

function	description	data type of operand a	data type of operand b	data type of result
shift_left(a,b) shift_right(a,b) rotate_left(a,b) rotate_right(a,b)	shift left shift right rotate left rotate right	unsigned, signed	natural	same as a
resize(a,b) std_match(a,b)	resize array compare '-'	unsigned, signed unsigned, signed std_logic_vector, std_logic	natural same as a	same as a boolean
to_integer(a) to_unsigned(a,b) to_signed(a,b)	data type conversion	unsigned, signed natural integer	 natural natural	integer unsigned signed

Data Type Conversion

Conversion can be accomplished by a *type conversion function* or by *type casting*

There are three *type conversion functions* in *numeric_std* package

`to_unsigned, to_signed and to_integer`

The function *to_integer* converts from data types *unsigned* or *signed*

The functions *to_unsigned* and *to_signed* convert from an integer data type to a specific number of bits (second parameter)

Type casting is also possible between 'closely related' data types

data type of a	to data type	conversion function / type casting	
unsigned, signed	std_logic_vector	std_logic_vector(a)	Type casting
signed, std_logic_vector	unsigned	unsigned(a)	
unsigned, signed	integer	to_integer(a)	Type conversion
natural	unsigned	to_unsigned(a, size)	
integer	signed	to_signed(a, size)	

Data Type Conversion

Examples of type casting:

```
signal u1, u2: unsigned(7 downto 0);
signal v1, v2: std_logic_vector(7 downto 0);

u1 <= unsigned(v1);
v2 <= std_logic_vector(u2);
```

From the table, we note that the *std_logic_vector* data type is not interpreted as a number and therefore canNOT be directly converted to an integer and vice versa

Type conversion needs to be carefully studied in VHDL -- consider some examples:

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

Data Type Conversion

```
signal s1, s2, s3, s4, s5, s6:
  std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
```

Ok

```
u3 <= u2 + u1;  --- ok, both operands unsigned
u4 <= u2 + 1;   --- ok, operands unsigned and natural
```

Wrong

```
u5 <= sg;  -- type mismatch
u6 <= 5;   -- type mismatch
```

Fix

```
u5 <= unsigned(sg);     -- type casting
u6 <= to_unsigned(5,4); -- conversion function
```

Data Type Conversion

Wrong

```
u7 <= sg + u1;    -- + undefined over the types
```

Fix

```
u7 <= unsigned(sg) + u1; -- ok, but be careful
```

Wrong

```
s3 <= u3;  -- type mismatch
s4 <= 5;   -- type mismatch
```

Fix

```
s3 <= std_logic_vector(u3); -- type casting
s4 <= std_logic_vector(to_unsigned(5,4));
```

Wrong

```
s5 <= s2 + s1; -- '+' undefined over std_logic_vector
s6 <= s2 + 1; -- '+' undefined
```

Fix

```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1));
s6 <= std_logic_vector(unsigned(s2) + 1);
```

Data Type Conversion

Integer conversions (useful for BRAM address manipulation):

```
signal cur_sample: std_logic_vector(7 downto 0);
signal addra: std_logic_vector(7 downto 0);

subtype addra_type is integer range 0 to 2**8-1;
signal MED_RAM_addra: MED_addra_type;

cur_sample <= "00000011";
MED_RAM_addra <= to_integer(unsigned(cur_sample));
addra <= std_logic_vector(to_unsigned(MED_RAM_addra +
1,8))
```