Module: Electronics & Telecomunication, 5rd year

Programmable Logical Devices

Advanced data types

Agenda

- Predefined Types
- Extended Types
  - Enumerated Types
  - Subtypes
- Composite Types
  - Arrays
  - Records
- Impure functions
- Other types
  - Lines
  - Files
- Advanced ☺ code example

Predefined Types
„standard" package

```
package standard is
  type boolean is (false, true);
  type bit is ('0', '1');
  type character is (
       nul, soh, stx, etx, eot, enq, ack, bel,...
       ...'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',.... );
  type severity_level is (note, warning, error, failure);
  type integer is range -2147483647 to 2147483647;
  type real is range -1.0E308 to 1.0E308;
  type time is range -2147483647 to 2147483647
    units
       fs;
       ps = 1000 fs;
       ns = 1000 ps;
       us = 1000 ns;
       ms = 1000 us;
       sec = 1000 ms;
       min = 60 sec;
       hr = 60 min;
    end units;
```


Predefined Types
„standard" package

```
  subtype delay_length is time range 0 fs to time'high
  impure function now return delay_length;
  subtype natural is integer range 0 to integer'high;
  subtype positive is integer range 1 to integer'high;
  type string is array (positive range <>) of character;
  type bit_vector is array (natural range <>) of bit;
  type file_open_kind is (
    read_mode,
    write_mode,
    append_mode);
  type file_open_status is (
    open_ok,
    status_error,
    name_error,
    mode_error);
  attribute foreign : string;
end standard;
```

Extended Types

Enumerated Types

The enumeration type is a type with an ordered set of values, called enumeration literals, and consisting of identifiers and character literals. Each of enumeration literals must be unique within the given declaration type.

Syntax:

type *identifier* **is** **(***item* {, *item*}**) ;**

item: {*character_literal* | *identifier*}

Examples:

literals:
```
type fiveval is ('?', '0', '1', 'Z', 'X');
```
identifiers:
```
type light is (red, yellow, green);
type instr is (load, store, add, sub);
```

BTW. Many types defined within the standard package have enumerated type syntax

Extended Types

Enumerated Types

```
architecture behave of cpu is
type instr is (lda, sta, add);
begin process
variable a, b, data: integer;
variable opcode: instr;
begin
process (...)
..........
  case opcode is
    when lda => a := data;
    when sta => data := a;
    when add => a := a + data;
  end case;
  wait on data;
end process;
end behave;
```

Synthesis:

00 add

01 lda

10 sta

11 ---

```
type instr is
(add,lda,ldb,invalid);
```

All enumerated values are ordered and each of them has a numeric (integer) value assigned to it. The number indicates the position of the literal. The very first literal in the definition has position number zero and each subsequent has the number increased by one from its predecessor

Extended Types

Enumerated Types

Enumeration types troubles

Different enumeration types may use the same literals. In this case, it is said that such literals are overloaded. When such a literal is referenced in the source code, its is determined from the context, in which enumeration this literal has occurred.

```
type ENUM_1 is (AAA, BBB, 'A', 'B', ZZZ);

type ENUM_2 is (CCC, DDD, 'C', 'D', ZZZ);

AAA -- Enumeration identifier of type ENUM_1

'B' -- Character literal of type ENUM_1

CCC -- Enumeration identifier of type ENUM_2

'D' -- Character literal of type ENUM_2

ENUM_1'(ZZZ) -- Qualified because overloaded
```


VHDL data types

Type casting /function overloading

Example 1:

```
bit'('1') - '1' as a bit? or '1' as the std_logic ?
```

Example 1:

```
function ToInt (d: bit_vector) return (integer);
function ToInt (d: std_logic_vector) return (integer);

ToInt("1010");                  -- ???

ToInt(bit_vector'("1010"));     -- OK!
```

VHDL data types

Type convertion

VHDL is strongly typed language. Objects of user-defined types cannot directly be assigned to or from objects of even a closely related type. A type conversion allows the assignment to be made:

```
type BUS_VAL is range 0 to 255;
variable X_INT : integer := 22;
variable X_BUS : BUS_VAL;
...
X_BUS := X_INT; --illegal
X_BUS := CONVERT_BUS_VAL(X_INT); -- function required
```


VHDL data types

Type convertion

```
type fourval is ('X', 'L', 'H', 'Z');
type value4 is ('X', '0', '1', 'Z');
process ..........;
   variable abc: fourval;
   variable xyz: value4;
begin
   xyz := convert4val (abc); -- conversion function call
end process;
function convert4val (s: fourval) return value4 is
begin
   case s is
      when 'X' => return 'X';
      when 'L' => return '0';
      when 'H' => return '1';
      when 'Z' => return 'Z';
   end case; end convert4val;
```

VHDL data types

Type convertion

Very often we need to convert from/to `integer` from/to `std_logic_vector`.

Use packages `IEEE.std_logic_unsigned` and `IEEE.std_logic_arith` !

```
function conv_integer (arg: std_logic_vector)
  return integer;
function conv_std_logic_vector (arg: integer; size: integer)
  return std_logic_vector;
```

Example:

```
entity sel is
  port (a,b,s: in integer range 0 to 15;
             q: out std_logic_vector (3 downto 0));
end;

architecture good of sel is
begin
  q <= conv_std_logic_vector(a,4) when conv_integer(s) = 8 else
       conv_std_logic_vector(b,4);
end;
```


VHDL data types

Type convertion

If all signals are `std_logic_vector`, the code would be smarter:

```
architecture better of sel is
begin
  q <= a when conv_integer(s) = 8 else b;
end;
```

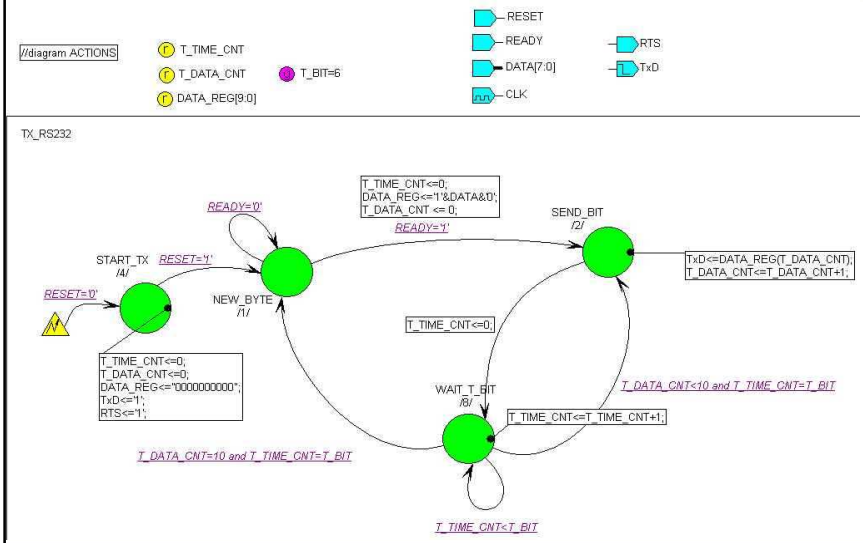
If synthesis supprts *operator overloading*, the code would be shorter :

```
architecture best of sel is
begin
  q <= a when s = 8 else b;
end;
```

From the synthesis point of view, the conversion function does not matter – there is no extra logic added (but it matter for simulation time!).

So, here comes a good advice:
Use std_logic_vector to get faster simulation
and predictable synthesis.

Extended Types
Enumerated Types – FSM example



Extende d Types
Enumerated Types – FSM example

```
architecture transmit of transmit is
-- diagram signals declarations
signal DATA_REG: STD_LOGIC_VECTOR (9 downto 0);
signal T_DATA_CNT: INTEGER range 0 to 11;
signal T_TIME_CNT: INTEGER range 0 to 200;
-- ONE HOT ENCODED state machine: TX_RS232
type TX_RS232_type is (NEW_BYTE, SEND_BIT, START_TX, WAIT_T_BIT);
attribute enum_encoding : string;
attribute enum_encoding of TX_RS232_type: type is
            "0001 " &               -- NEW_BYTE
            "0010 " &               -- SEND_BIT
            "0100 " &               -- START_TX
            "1000" ;                -- WAIT_T_BIT
signal TX_RS232: TX_RS232_type;
begin
-- concurrent signals assignments
-- diagram ACTIONS
-------------------------------------------------------------------
-- Machine: TX_RS232
-------------------------------------------------------------------
TX_RS232_machine: process (CLK, reset)
begin
if RESET='0' then
            TX_RS232 <= START_TX;
            -- Set default values for registered outputs/signals and for variables
            -- ...
            T_TIME_CNT <= 0;
            T_DATA_CNT <= 0;
            DATA_REG <= "0000000000";
            TxD <= '1';
elsif CLK'event and CLK = '1' then
            -- Set default values for registered outputs/signals and for variables
```

Extended Types

Enumerated Types – FSM example

```
TX_RS232_machine: process (CLK, reset)
begin
if RESET='0' then
            TX_RS232 <= START_TX;
            -- Set default values for registered outputs/signals and for variables
            -- ...
            T_TIME_CNT <= 0;
            T_DATA_CNT <= 0;
            DATA_REG <= "0000000000";
            TxD <= '1';
elsif CLK'event and CLK = '1' then
            -- Set default values for registered outputs/signals and for variables
            -- ...
            case TX_RS232 is
                        when NEW_BYTE =>
                                    if READY='1' then
                                                TX_RS232 <= SEND_BIT;
                                                T_TIME_CNT <= 0;
                                                DATA_REG <= '1'&DATA&'0';
                                                T_DATA_CNT <= 0;
                                    elsif READY='0' then
                                                TX_RS232 <= NEW_BYTE;
                                    end if;
                        when SEND_BIT =>
                                    TxD <= DATA_REG(T_DATA_CNT);
                                    T_DATA_CNT <= T_DATA_CNT+1;
                                    TX_RS232 <= WAIT_T_BIT;
                                    T_TIME_CNT <= 0;
                        when START_TX =>
                                    T_TIME_CNT <= 0;
                                    T_DATA_CNT <= 0;
                                    DATA_REG <= "0000000000";
...........
```


Extended Types

Subtypes

A subtype is a type with a constraint. It is used to limit the range on the original type.

Examples:

```
subtype digit is integer range 0 to 9;
variable msd, lsd: digit;
```

Same as:

```
variable msd, lsd: integer range 0 to 9;

type instr is (add, sub, mul, div, sta, stb, outa, xfr);
subtype arith is instr range add to div;
subtype pos is integer range 1 to 2147483647;
subtype nano is time range 0 ns to 1 us;
```

Extended Types

Subtypes

Important notes

- **A subtype declaration does not define a new type.**
- **A subtype is the same type as its base type; thus, no type conversion is needed when objects of a subtype and its base type are assigned (in either direction). Also, the set of operations allowed on operands of a subtype is the same as the set of operations on its base type.**
- **Using subtypes of enumerated and integer types for synthesis is strongly recommended as synthesis tools infer an appropriate number of bits in synthesized registers, depending on the range.**

Extended Types

Subtypes examples

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
```

NATURAL is a numeric subtype of the type INTEGER of the range from 0 to INTEGER'HIGH. The subtype values represent mathematical natural numbers. All of the basic mathematical functions such as addition, subtraction, multiplication, and division can be applied to operands of the NATURAL subtype.

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

POSITIVE is a numeric subtype of the type INTEGER of the range from 1 to INTEGER'HIGH. The subtype values represent mathematical natural numbers that are greater than 0. All of the basic mathematical functions such as addition, subtraction, multiplication, and division can be applied to operands of the POSITIVE subtype.

```
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
```

DELAY_LENGTH is a subtype of the TIME type of the range from 0 to TIME'HIGH. The DELAY_LENGTH subtype is used to represent the value of the simulation time.

```
subtype WIDTH is NATURAL;
```

WIDTH is a subtype representing NATURAL values used for specifying widths of output fields. The subtype is used in the FIELD parameter of the WRITE procedure that is provided in the TEXTIO package.

Extended Types

Subtypes examples

The std_logic_1164 package provides four additional subtypes of the STD_ULOGIC type. Like STD_LOGIC, all of them are resolved (using the same resolution functions RESOLVED) but their sets of values are narrowed.

Type name	Set of values
X01	'X', '0', '1'
X01Z	'X', '0', '1', 'Z'
UX01	'U', 'X', '0', '1'
UX01Z	'U', 'X', '0', '1', 'Z'

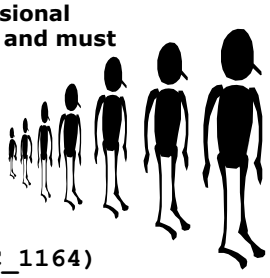
Syntax

```
subtype X01 is RESOLVED STD_ULOGIC range 'X' to '1';
subtype X01Z is RESOLVED STD_ULOGIC range 'X' to 'Z';
subtype UX01 is RESOLVED STD_ULOGIC range 'U' to '1';
subtype UX01Z is RESOLVED STD_ULOGIC range 'U' to 'Z';
```


Composite Types

Arrays

Array formal definition
A type, the value of which consists of elements that are all of the same subtype (and hence, of the same type). Each element is uniquely distinguished by an index (for a one-dimensional array) or by a sequence of indexes (for a multidimensional array). Each index must be a value of a discrete type and must lie in the correct index range.



- Usefull to buses & registers sets description

There are predefined arrays :

- `bit_vector (package STANDARD)`
- `string (package STANDARD)`
- `std_logic_vector (package STD_LOGIC_1164)`

To use the other arrays type, one must define them !
(like for `real` i `integer`)

Composite Types
Arrays

Syntax:

```
type name is array [index_constraint] of element_type;
index_constraint:  [range_spec]                    -- constrained
                   index_type range [range_spec]   -- constrained
                   index_type range <>             -- unconstrained
```

An array may be either constrained or unconstrained. The array is constrained if the size of the array is constrained. The size of the array can be constrained using a discrete type mark or a range. In both cases, the number of the elements in the array is known during the compilation.

Examples:

```
type word8 is array (1 to 8) of bit;
type word8 is array (integer range 1 to 8) of bit;
type word is array (integer range <>) of bit;
type ram is array (1 to 8, 1 to 10) of bit;
```


Composite Types
Arrays

When type is defined properly, it can be used to object (signal/variable) declaration.

Examples :

```
variable data_bus: word8;
variable register: word (1 to 10);
```

Enumerated type or subtype can be used to index an array.

Examples :

```
type instruction is (add, sub, mul, div, lda, sta, xfr);
subtype arithmetic is instruction range add to div;
subtype digit is integer range 1 to 9;

type Ten_bit is array (digit) of bit;
type Inst_flag is array (instruction) of digit;
```

Composite Types

Multidimensional Arrays

Usefull for memories (RAM or ROM)

Example:

```
type memory is array (0 to 7, 0 to 3) of bit;
constant rom: memory := (('0', '0', '0', '0'),
                         ('0', '0', '0', '1'),
                         ('0', '0', '1', '0'),
                         ('0', '0', '1', '1'),
                         ('0', '1', '0', '0'),
                         ('0', '1', '0', '0'),
                         ('0', '1', '1', '0'),
                         ('0', '1', '0', '1'));
```

```
data_bit := rom(5,3); -- word 5, bit 3
```


Composite Types

Arrays of the Arrays

Example :

```
type word is array (0 to 3) of bit;
type memory is array (0 to 4) of word;
variable addr, index: integer;
variable data: word;
constant rom_data: memory := (('0', '0', '0', '0'),
                              ('0', '0', '0', '1'),
                              ('0', '0', '1', '0'),
                              ('0', '1', '1', '1'),
                              ('0', '1', '1', '1'));

data := rom_data(addr);

rom_data(addr)(index) --access
```

Composite Types

Records

The *record type* allows declaring composite objects whose elements can be of different types. This is the main difference from *arrays*, which must have all elements of the same type

Example :

```
type two_digit is record
   sign: bit;
    msd: integer range 0 to 9;
    lsd: integer range 0 to 9;
end record;
process
variable acntr, bcntr: two_digit;
   begin
      acntr.sign := '1';
      acntr.msd := 1;
      acntr.lsd := acntr.msd;
      bcntr := two_digit'('0',3,6);
end process;
```


Composite Types

Instruction alias

The *alias* declares an alternative name for any existing object: signal, variable, constant or file. It can also be used for "non-objects": virtually everything, which was previously declared, except for labels, loop parameters, and generate parameters. *Alias* does not define a new object. It is just a specific name assigned to some existing object.

Example :

```
signal count: bit_vector (1 to 9);
   alias sign: bit is count (1);
   alias  msd: bit_vector (1 to 4) is count (2 to 5);
   alias  lsd: bit_vector (1 to 4) is count (6 to 9);

count := "1_1001_0000";
sign := '1';
msd := "1001";
lsd := msd;
```

Sequential statements

Impure function

Functions return a single value. When the function is called the formal parameters are given the values of the actual parameters.

Syntax:

[impure] function *name* [(*parameter*: *type*;...)] return *type* is
declarations
begin
    sequential statements;
end [*name*];

Functions can be either pure (which is default) or impure. Pure functions always return the same value for the same set of actual parameters. Impure functions may return different values for the same set of parameters. Additionally, an impure function may have "side effects", like updating objects outside of their scope, which is not allowed for pure functions.

```
impure function now return delay_length;
```


Sequential statements

Impure function

```
variable number: INTEGER := 0;
impure function strange_impure_function( A: INTEGER )
return INTEGER is
      variable counter: INTEGER;
begin
      counter := A + number;
      number := number + 10;
      return counter;
end;
```

Predefined text types

text & line types

TEXT is a file type representing files of variable-length text strings (ASCII records).
LINE is an access type designating a STRING value which is a line to write to a file or a line that has just been read from the file. The LINE type is the basic unit upon which all TextIO operations are performed.

STD library - TEXTIO Package:

```
readline, read, writeline, write
```

Example:

```
readline (F: in text; L: out line);
read (L: inout line; ITEM: integer);
```

IEEE library – STD_LOGIC_TEXTIO Package:

```
read, write
oread, owrite
hread, hwrite
```

Predefined:

```
endfile(filename), endline(linename)
```