VHDL Composite Data Types and Operations

VLSI CAD
Case Western Reserve University

Dan Saab

Composite Data Types

- Consist of related collections of data elements
  - Array
  - Record
- Treated as single object
- Can manipulate its elements individually.
- We will see how they are defined and manipulated.

Arrays

- Declaration
  - Package
  - Entity
  - Architecture
  - Process
  - Procedure
  - Function
- Syntax

  type type_name is array (range) of element_type;

Arrays

- An array contains multiple elements of the same type. When an array object is declared, an existing array type must be used.

- Examples:

  type NIBBLE is array (3 downto 0) of std_ulogic;
  type RAM is array (0 to 31) of integer range 0 to 255;
  signal A_BUS : NIBBLE;
  signal RAM_0 : RAM;
- Index does not have to be numeric

  type cpustates is (fetch, decode, execute, writeback);
  type state cpu is array (decode to writeback) of natural;

Unconstrained Arrays

- An array type definition can be unconstrained, i.e. of undefined length. String, bit_vector and std_logic_vector are defined in this way. An object (signal, variable or constant) of an unconstrained array type must have it's index type range defined when it is declared.

```
type INT_ARRAY is array (integer range <>) of integer;
variable INT_TABLE: INT_ARRAY(0 to 9);
variable LOC_BUS : std_ulogic_vector(7 downto 0);

type atype is array (natural range <>) of integer;
subtype abtype is atype(0 to 63);

variable a1: atype ( 0 to 100 );
variable a2: abtype;

constant a3:atype :=(1=>10,2=>20,3=>30,4=>40);-- range 1 to 4
constant a4:atype := (10, 20, 30, 40);-- range 0 to 3
```

Unconstrained Standard-Logic Arrays

- std_ulogic 1164 package provide an unconstrained standard-logic
- to use include at the beginning of your file

```
library ieee; use ieee.std_ulogic 1164.all;
type std ulogic is (
        'U', -- Uninitialized
        'X', -- Forcing unknown
        '0', -- Forcing zero
        '1', -- Forcing one
        'Z', -- High impedance
        'W', -- Weak unknown
        'L', -- Weak zero
        'H', -- Weak one
        '-'); -- Don't care

type std ulogic vector is array (natural range<>) of std ulogic;

subtype std_ulogic word is std ulogic vector(0 to 31);
signal a1: std_ulogic word
signal a2: std_ulogic vector(0 to 10) := "ZZZZZZ----";
signal a3: std_ulogic vector(0 to 7) := X"FF";
```

Unconstrained Arrays ports

- Allows a more general interface description
  - Connect signals of any size to that port
  - Definition of bound is computed on instantiation
- Example

```
entity and_bus is
        port ( i : in bit_vector; y:out  bit);
end entity and multiple;
architecture behavioral of and bus is
begin
        and_block: process (i) is
        variable result: bit;
        begin
                result:='1';
                for index in i'range loop
                        result := result and i(index);
                end loop;
                y <= result;
        end process and_block;
end architecture behavioral;
```

continue

```
entity testbench is end entity testbench;

architecture toplevel of testbench is
signal invec : bit_vector(7 downto 0);
signal zout : bit;
begin
    agate : entity work.and_bus(behavioral) port map ( i =>
    invec, y => zout);
    stinulus: process
    begin
        invec <= b"0000 0000"; wait for 10 ns;
        invec <= b"1010 1100"; wait for 10 ns;
        invec <= X"FF"; wait for 10 ns;
        ...
    end process stinulus;
end architecture toplevel;
```

Arrays

- Arrays with character elements such as string, bit_vector and std_logic_vector may be assigned a literal value using double quotes (see literals):
- Example

```
CONSTANT MSG_o: string := "Test 1 Completed";
...
A_BUS <= "0000";
LOC_BUS <= "10101010";
```

Arrays

- Arrays may also be assigned using concatenation (&), aggregates, slices, or a mixture. By default, assignment is made be position
- Example

```
A_BUS <= (A_BIT, B_BIT, C_BIT, D_BIT);
-- an equivalent assignment:
A_BUS <= (A_BIT & B_BIT & C_BIT & D_BIT);
-- rotate A_BUS to the left:
A_BUS <= A_BUS(2 downto 0) & A_BUS(3);;
```

Arrays

- Arrays of arrays may be declared. These are useful for memories, vector tables, etc.:
- Example

```
type NIBBLE is array (3 downto 0) of std_ulogic;
type MEM is array (0 to 7) of NIBBLE;
-- an array "array of array" type
variable MEM8X4 : MEM;
...
-- accessing the whole array:
MEM8X4 := ("0000", "0001", "0010", "0011",
           "0100", "0101", "0110", "0111");
-- accessing a "word"
MEM8X4(5) := "0110";
-- accessing a single bit
MEM8X4(6) (0) := '0';
```

```
subtype ramaddress is integer range 0 to 63;
entity memory1 is
        port ( rd, wr: in bit; addr: in ramaddress;
                din: in real; dout: out real);
end entity memory1;
architecture adisc of memory1 is
begin
        memory: process is
        type storedt is array (ramaddress) of real;
        variable ram storedt;
        begin
                for index in ramaddress loop
                        ram(index) := 0.0;
                end loop;
                loop
                        wait on rd, wr, addr, din;
                        if rd = '1' then
                                dout <= ram(addr);
                        end if;
                        if wr = '1' then
                                ram(addr) := din;
                        end if;
                end loop;
        end process memory;
end architecture adisc;
```

Multidimensional Arrays

- two (or more) dimensional arrays may also be declared:

Example:

```
type atype is (a, t, d, h, digit, cr, error);
type btype is range 0 to 6;
type twodim is array (atype, btype) of btype;
variable table1 : twodim;

table1(d, 5);

type T_2D is array (3 downto 0, 1 downto 0) of integer;
signal X_2D : T_2D;
...
X_2D <= ((0,0), (1,1), (2,2), (3,3));
X_2D(3,1) <= 4;
```

Multidimensional Arrays

- **Most logic synthesis tools accept one-dimensional arrays of other supported types. 1-D arrays of 1-D arrays are often supported. Some tols also allow true 2-D arrays, but not more dimensions**

```
type T_2D is array (3 downto 0, 1 downto 0) of integer;
signal X_2D : T_2D;
...
X_2D <= ((0,0), (1,1), (2,2), (3,3));
X_2D(3,1) <= 4;
```

Multidimensional Arrays

```
type atype is array (1 to 3) of real;
type btype is array (1 to 3, 1 to 3) of real;
...
variable a1, a2 : atype;
variable scale: btype;
...
for i in 1 to 3 loop
        a1(i) := 0.0;
        for j in 1 to 3 loop
                a1(i) := a1(i) + scale(i, j) * a2(j) ;
        end loop;
end loop;
```

Array Aggregates

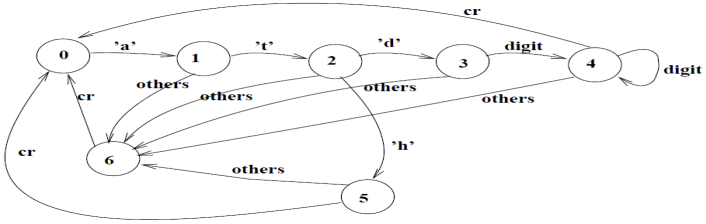
- Used to initialize a variable or constant of an array type.

Example:

```
type atype is array (1 to 3) of real;
subtype ramaddress is integer range 0 to 63;
...
constant a1 : atype := (0.0, 0.0, 0.0);
variable a2 : atype := (1.0, 2.0, 0.0);
variable a3 : atype := (1=>11.0, 2=>12.0, 3=>0.0);

type ram is array (ramaddress) of real;
variable memory1: ram := (0 => 1.6, 1 => 2.3, 2 => 1.6, 3 to 63 => 0.0);
-- others must be the last choice
variable memory2: ram := (0 => 1.6, 1 => 2.3, 2 => 1.6, others => 0.0);
-- 0 and 1 have same value 0 | 1 => 2.3
variable memory3: ram := (0 | 1 => 2.3, 2 => 1.6, others => 0.0);
```

Multidimensional Array Aggregates




```
modemcontroller: process is
        type symbols is ('a', 't', 'd', 'h', digit, cr, other);
        type sequence is array (1 to 20) of symbols;
        type state is range 0 to 6;
        type diagram is array (state, symbol) of state;
        constant nextstate : diagram
                (0 => ('a' => 1, others => 6),
                1 => ('t' => 2, others => 6),
                2 => ('d' => 3, 'h' => 5, others => 6),
                3 => (digit => 4, others => 6),
                4 => (digit => 4, cr => 0, others => 6),
                5 => (cr => 0, others => 6),
                6 => (cr => 0, others => 6));
```

FSM

```
aprocess: process is
        type symbols is ('a', 't', 'd', 'h', digit, or, other);
        type sequence is array (1 to 20) of symbols;
        type state is range 0 to 6;
        type diagram is array (state, symbol) of state;
        constant nextstate : diagram
                (0 => ('a' => 1, others => 6),
                1 => ('t' => 2, others => 6),
                2 => ('d' => 3, 'h' => 5, others => 6),
                3 => (digit => 4, others => 6),
                4 => (digit => 4, cr => 0, others => 6),
                5 => (cr => 0, others => 6),
                6 => (cr => 0, others => 6));
        variable command: sequence;
        variable PresentState : state := 0;
        begin
                ...
                for index in 1 to 20 loop
                        PresentState := nextstate( PresentState, command(index));
                        case PresentState is
                                when 0 =>
                                        ...
                                when 1 =>
                                ...
                                when 2 =>
                            end case;
                end loop;
        ...
end process aprocess;
```

Aggregate

- Each element position must contain a variable name
- Assignment must produce a value of the same composite type
- Each right hand side position is assigned to the corresponding left hand side position
- Example

```
variable avec: bit vector (0 to 3) := "0011" ;
...
signal af,bf,cf,df : bit;
...
(af,bf,cf,df) <= avec ;
        -- af <= avec(0)
        -- bf <= avec(1)
        -- cf <= avec(2)
        -- df <= avec(3)
```

Aggregate

- The second form is named association, where elements are explicitly referenced and order is not important:
- Example

```
signal Z_BUS : bit_vector (3 downto 0);
signal A_BIT, B_BIT, C_BIT, D_BIT : bit;
...
Z_BUS <= ( 2=> B_BIT, 1 => C_BIT, 0 => D_BIT, 3 => A_BIT);
```

Aggregate

- With positional association, elements may be grouped together using the | symbol or a range. The keyword others may be used to refer to all elements not already mentioned:
- Example

```
signal B_BIT : bit;
signal BYTE : bit_vector (7 downto 0);
...
BYTE<= (7 => '1', 5 downto 1 => '1', 6 => B_BIT, others => '0');
```

Array Attributes

- An array type of object A
- An integer N between 1 and the number of dimensions of A

A'left(N)	Left bound of index range of dimension N of A
A'right(N)	Right bound of index range of dimension N of A
A'low(N)	Lower bound of index range of dimension N of A
A'high(N)	Upper bound of index range of dimension N of A
A'range(N)	Index range of dimension N of A
A'reverse range(N)	Reverse of index range of dimension N of A
A'length(N)	Length of index range of dimension N of A
A'ascending(N)	true if index range of dimension N of A is an ascending range, false otherwise

Array Attributes

```
type A is array (1 to 4, 31 downto 0) of integer;

A'left(l) = 1
A'low(1) = 1
A'right(2) = 0
A'high(2) = 31
A'range(1) is 1 to 4
A'reverse range(2) is 0 to 31
A'length(1) = 4
A'length(2) 32
A'ascending(1) = true
A'ascending(2) = false

for i in A'range(1) loop
        for j in A'reverse range(2) loop
                count := count + 1;
        end loop;
end loop;
```

Array Operations

- Logical operators (and, or, nand, nor, xor, xnor) can be applied to
  - Two one dimentional arrays A and B of bits or boolean
  - A and B are of same length
  - A and B are of same type
- not can be applied to a single array of bits or boolean
- sll, sri, sla, sra, rol and ror can be applied to a single array of bits or boolean
- concatenation operator (&) can be applied to one dimensional array
  - (&) joins two arrays
- Results is an array of the same type and length

Examples

B"10001010" sll 3 = B"01010000"
B"10001010" sll -2 = B"00100010" -- negative, shift to the right
B"10010111" srl 2 = B"00100101"
B"10010111" srl-6 = B"11000000"
B"01001011" sra 3 = B"0000l00l"
B"10010111" sra 3 = B"llll00l0"
B"00001100" sla 2 = B"00110000"
B"00010001" sla 2 = B"01000111"
B"00010001" sra -2 = B"01000111"
B"00110000" sla -2 = B"00001100"
B"10010011" rol 1 = B"00100111"
B"10010011" ror 1 = B"11001001"
b“000" & b"1111," = b“000_1111"
"abc" & ’d’ = "abcd"
’w’ & "xyz" = "wxyz"
’a’ & ’b’ = "ab"

Relational operators

- Can be applied to one-dimensional arrays
  - Array can be any discrete type
  - Operand need not be of the same length
  - Operand need to be of the same type
- A and B are two arrays
  - A<B is false if A and B have length 0
  - A<B is false if B has length 0, and A has non-zero
  - A<B is true if A has length 0, and B has non-zero
  - A<B is true
    - if A(1) < B(1) or A(1)=B(1)
    - and the rest of A is < rest of B

Array Slices

- VHDL allows referencing a subset of elements of an array

```
type array1 is array (1 to 100) of integer;
type array2 is array (100 downto 1) of integer;
variable a1 array1;
variable a2 array2;

a1(11 to 20) -- range must have the same direction as the original array
a2(50 downto 41)

a1(11 to 20) := a2(50 downto 41) ;
-- a(11) := a(50)
-- a(12) := a(49)
-- a(13) := a(48)
...
-- a(20) := a(41)

a1(10 to 1) -- null slice
a2(1 downto 10) -- null slice
```

Records

- A record is a composite value comprising elements that may be of different types from one another. (struct in C)

```
type Arec is record
        a : integer range 0 to 59;
        b: integer range 0 to 59;
        c : integer range 0 to 23;
end record Arec;

variable rec1, rec2 : Arec;

-- read record elememt
avar := rec1.a;
bvar := rec2.b;

-- modify record element
rec1.a := count mod 60;
```

```
architecture system level of computer is
type opcodes is
        (add, sub, addu, subu, jmp, breq, brne, ld, st,...);

type reg number is range 0 to 31;
constant r0 : reg_number := 0; constant r1 : reg_number := 1;

type instruction is record
        opcode : opcodes;
        source reg1, source_reg2, des reg: reg_number;
        displacement: integer;
end record instruction;

type word is record
        instr: instruction;
        data: bit_vector(31 downto 0);
end record word;

signal address: natural; signal read_word, write_word : word;
signal mem_read, mem_write: bit := '0';
signal mem_ready: bit '0';
```

```
begin
cpu : process is
variable inst reg : instruction;
variable PC : natural;
... -- other declarations for register file, etc.
begin
        address < PC;
        mem_read <= '1';
        wait until mem_ready = '1';
        inst_reg := read_word.instr;
        mem_read <= '0';
        PC := PC +4;
        case inst_reg.opcode is -- execute the
instruction
                when add => ...
        end case;
end process cpu;

memory: process is
type memory array is array (0 to 2**14 - 1) of word;
variable store : memory..array
(       0 => ((ld, r0, r0, r2, 40 ), X"00000000"),
        1 => ((breq, r2, r0, r0, 5), X"00000000"),
        ....
        40 => ((flop, r0, r0, r0, 0 ), X"FFFFFFFE"),
        others => ((nop, r0, r0, r0, 0), X“00000000")
);
begin
```