Advanced Topics

Topics

- Physical Types
- Array Types
- Entity Attributes
- Access Types and Record Structures
- Shared Variables

Physical Types

Modeling Physical Quantities

- Physical quantities are represented by the type of their measured values
  - Integer, real, logical, etc.
- Precision, range, and type casting issues often require the programmer to manage quantization
- Hardware description languages expand the range of physical quantities to be represented and managed

Modeling Physical Quantities: Example

```
entity inv_rc is
generic (c_load: real:= 0.066E-12); -- farads
port (i1 : in std_logic;
      o1: out: std_logic);
constant rpu: real:= 25000.0; --ohms
constant rpd: real :=15000.0; -- ohms
end inv_rc;

architecture delay of inv_rc is
constant tplh: time := integer (rpu*c_load*1.0E15)*3 fs;
constant tpll: time := integer (rpu*c_load*1.0E15)*3 fs;
begin
o1 <= '1' after tplh when i1 = '0' else
    '0' after tpll when i1- = '1' or i1 = 'Z' else
    'X' after tplh;
end delay;
```

visible in all architectures

explicit type casting and range management

These are known/evaluated at compile time

Example adapted from "VHDL: Analysis and Modeling of Digital Systems," Z. Navabi, McGraw Hill, 1998.

Notion of Physical Types

- Purpose: to be able to create and manipulate objects that correspond to physical, measurable, quantities
  - Resistance, capacitance, time, inductance, etc.

- **time** is a pre-defined physical type in the language

```
type time is range <implementation dependent>
units
fs;
ps = 1000 fs;       -- femtoseconds
ns = 1000 ps;       -- picoseconds
us = 1000 ns;       -- microseconds
ms = 1000 us;       -- milliseconds
s = 1000 ms;        -- seconds
min = 60 s;         -- minutes
hour = 60 min;      -- hours
end units;
```

Other Examples

in terms of base units and only integer bounds

```
type power is range 1 to 1000000
units
uw;
mw = 1000 uw;
w = 1000 mw;
kw = 1000 w;
mgw = 1000 kw;
end units;
```

- Define a base unit and integer range that a variable or constant can take
  - Define aggregate units

Physical Types: Example (cont.)

```
type capacitance is range 0 to
     1E16
units
ffr;                  -- femtofarads
pfr = 1000 ffr;
nfr = 1000 pfr;
ufr = 1000 nfr
mfr = 1000 ufr
far = 1000 mfr;
kfr = 1000 far;
end units;
```

```
type resistance is range 0 to 1E16
units
l_o;                  -- milli-ohms
ohms = 1000 l_o;
k_o= 1000 ohms;
m_o = 1000 k_o;
g_o = 1000 m_o;
end units;
```

- Programmer must manage interpretations of the values
- Rather than mapping the values to the real numbers, create new physical types

Example adapted from "VHDL: Analysis and Modeling of Digital Systems," Z. Navabi, McGraw Hill, 1998.

Physical Types: Example (cont.)

```
entity inv_rc is
generic (c_load: capacitance := 66 ffr); -- farads
port (i1 : in std_logic;
        o1: out: std_logic);
constant rpu: resistance:= 25000 ohms;
constant rpd : resistance := 15000 ohms;
end inv_rc;

architecture delay of inv_rc is

constant tplh: time := (rpu/ 1 l_o)* (c_load/1 ffr) *3 fs/1000;
constant tpll: time := (rpu/ 1 l_o)* (c_load/1 ffr) *3 fs/1000;
begin
o1 <= ‘1’ after tplh when i1 = ‘0’ else
     ‘0’ after tpll when i1 = ‘1’ or i1 = ‘Z’ else
     ‘X’ after tplh;
end delay;
```

Define a new overloaded multiplication operator

This expression now becomes

rpu * c_load * 3

Example adapted from “VHDL: Analysis and Modeling of Digital Systems,” Z. Navabi, McGraw Hill, 1998.

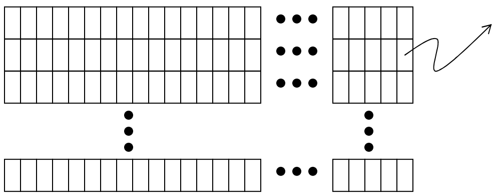
Basic Ideas

- Arithmetic operators are not defined for physical types
  - Convert the values to dimensionless quantities
  - Perform integer operations
  - Convert back to a physical type
    - One of the arithmetic operands is an integer and one is a physical type

- Many aspects of type management is moved from programmer to language

Array Types

Thinking About Arrays



All elements must be of the same type

- Types of multidimensional arrays
  - Multidimensional arrays
  - Arrays of arrays
- The type determines how elements in an array can be referenced
  - Indexing
  - Using range information → dependent on construction of the declaration

Referencing Array Data

type std_byte is array (7 downto 0) of std_logic;

type std_word is array (31 downto 0) of std_logic;

type 2Dmask is array (7 downto 0, 4 downto 0) of
std_logic;

Can mix ascending and descending ranges

type register_file is array (31 downto 0) of std_word;

Array Aggregates

std_word <= (3 => ‘1’, others => ‘Z’)
- Named associations

std_word <= (‘0’, 3 => ‘1’, others => ‘Z’);
- Positional association

std_word <= (4 downto 0 => ‘1’, others => ‘Z’);
- Specifying ranges
- Can mix descending and ascending ranges

- Aggregates apply to each dimension

Nesting Array Aggregates

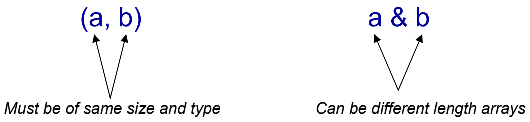
- Specification applies to each dimension of the array

2Dmask <= (others => (others => Z));

2Dmask <= (others => (‘1’, others => Z));

General Aggregate Operations

- This is the combination of one or more values into a more complex type



(a, b)

a & b

Must be of same size and type

Can be different length arrays

Generalizing Array Indexing

- Indices can be of types other than integers

- Array access follows the same principle → use the type value to define the corresponding array element

type std_byte is array (std_logic) of std_logic;

- Nine elements in this array type
- Indexed by the values of the std_logic type *in the order in which it is defined*

- Named associations, positional associations, and array aggregates can be mixed and matched

Unconstrained Arrays

- Useful for building generic, parametric models
- Type bit_vector is array (natural range<>) of bit

ascending or descending range

```
procedure write_v1d (
variable f: out text; v : in std_logic_vector) is
variable buf: line;
variable c : character;
begin
for i in v'range loop
case v(i) is
when 'X' => write(buf, 'X');
..
..
..
```

```
function wire_or (sbus :std_ulogic_vector)
return std_ulogic is
begin
for i in sbus'range loop
if sbus(i) = '1' then
return '1';
end if;
end loop;
return '0';
end wire_or;
```

For Hardware Generation

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

entity gregister is
port (din : in std_logic_vector;
     qout: out std_logic_vector;
     clk, we : in std_logic);
  end entity gregister;

  architecture behavioral of gregister is

component dff_en is
   Port ( d : in  STD_LOGIC;
         we : in  STD_LOGIC;
         clk : in  STD_LOGIC;
         q : out  STD_LOGIC);
end component dff_en;
begin
     dreg: for i in din'range generate
     reg: dff_en port map( d=>din(i), q=>qout(i), we=>we, clk=>clk);
     end generate;
end architecture behavioral
```

unconstrained arrays

Entity Attributes

Entity Attributes

- Enables identification of aspects of the specific entity such as
  - Name: entity_name**’simple_name**
  - Instance: entity_name**’path_name**
  - Path to this instance: entity_name’**instance_name**
- Useful in debugging programs
- Example

Example

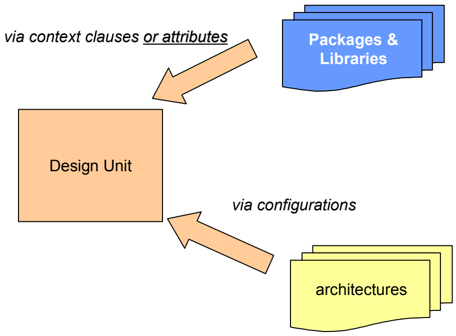
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use STD.textio.all;

entity nand2 is
    generic (gate_delay: time:= 2 ns);
  port ( a, b : in  STD_LOGIC;
        c : out  STD_LOGIC);

end entity nand2;

architecture behavioral of nand2 is
Begin

    c <= a nand b after gate_delay;

process
  variable buf: line;
  variable simple: string(1 to
    nand2'simple_name'length):= (others =>'.');
  variable path: string(1 to
    nand2'path_name'length):= (others =>'.');
variable instance: string(1 to
    nand2'instance_name'length):= (others =>'.');
begin
    simple := nand2'simple_name;
    path := nand2'path_name;
    instance := nand2'instance_name;

write (buf, simple);
writeline (output,buf);

write (buf, path);
writeline (output,buf);

write (buf,instance);
writeline (output,buf);
wait;
end process;
```

Adapted from: Z. Navabi, “VHDL: Analysis and Modeling of Digital Systems”, McGraw Hill 1998

User Defined Attributes

- These attributes do not have simulation or synthesis semantics. They are for the use by the designer

- This is another mechanism for communication information throughout a design



Record Types

- Records are a composite type where each element may be of a distinct type

```
type opcode is (add, sub, and, or, xor, sl, sr, ld, sw, rot, nop);
type reg_addr is integer range 0 to 31;
type addr is unsigned (17 downto 0);
type op_format is unsigned (12 downto 0);
```

```
type r_format is record
op : opcode;
dest: reg_addr;
source1 : reg_addr;
source2: reg_addr;
misc_op: op_format;
end record;
```

```
type i_format is record
op : opcode;
dest: reg_addr;
source1 : reg_addr;
mem_addr: addr;
end record;
```

op	dest	source1	source2	op_format

op	dest	source1	mem_addr

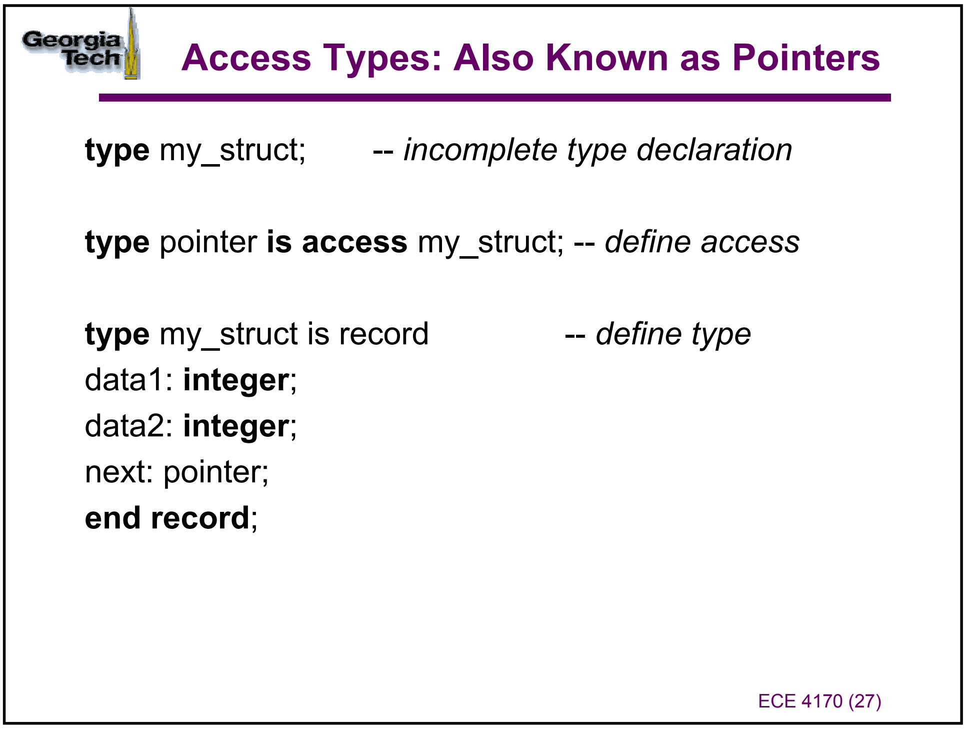
Alias

- Declare alternative labels for parts of a structure
  - For example, consider bits 4 through 8 of the memory address as a cache line address

```
signal current_instr : i_format:= (nop, 0,0,”000000000000000000”);
alias cache_line is  current_instr.mem_addr(7 downto 3);
..
..
index <= cache_line;
…
cache_line <= “1110”;
```

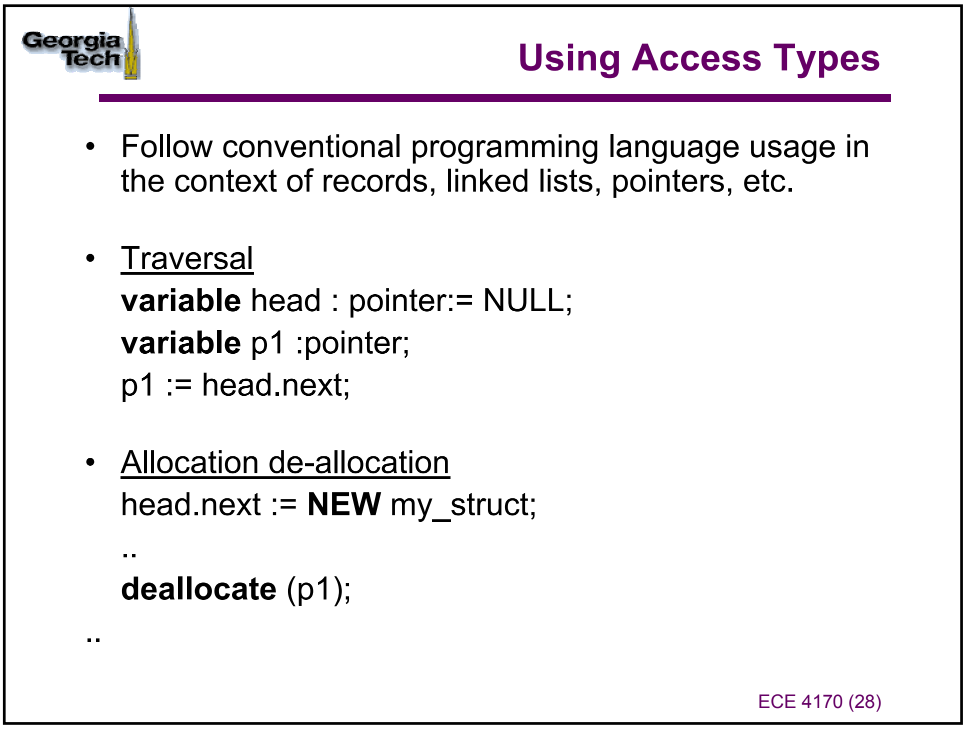
Comments

- VHDL is intended to model hardware structures at all levels of design
  - Device → timing, delay, physical attributes
  - Gate level → timing, delay, logic operations, physical attributes
  - Instruction set level → instruction formats, memory structures, operating system data, architecture state information
  - Block level: test bench, verification & validation
- Different aspects of the language are used at different levels of modeling
- This distinguishes VHDL from many domain-specific modeling languages



Access Types: Also Known as Pointers

```
type my_struct;        -- incomplete type declaration

type pointer is access my_struct; -- define access

type my_struct is record                -- define type
data1: integer;
data2: integer;
next: pointer;
end record;
```



Using Access Types

- Follow conventional programming language usage in the context of records, linked lists, pointers, etc.

- <u>Traversal</u>

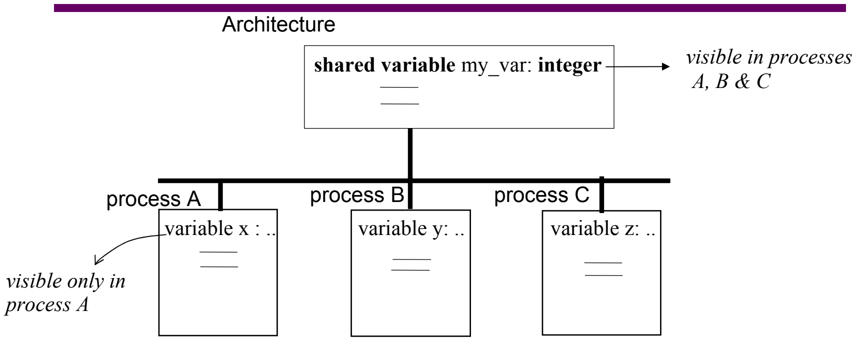
```
variable head : pointer:= NULL;
variable p1 :pointer;
p1 := head.next;
```

- <u>Allocation de-allocation</u>

```
head.next := NEW my_struct;
..
deallocate (p1);
```

..

Shared Variables


Architecture
shared variable my_var: integer
visible in processes A, B & C
process A
process B
process C
variable x : ..
variable y: ..
variable z: ..
visible only in process A

- Shared variables represent a way to change the visible scope of a variable
  - Now accessible to a range of procedures and processes
  - Effect is non-deterministic
- Examples

Blocks and Guarded Signal Assignments

- Blocks are a mechanism to identify a “part” of a design without treating it as a complete design unit
  - Entity/architecture pair need not be created
- Syntactically identify a part of the design
  - Treat it like a design entity in the sense that
    - It can have ports and generics
    - Has a declarative part
    - Has a concurrent statement part

Example: Blocks and Guards

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

entity my_dff is
   generic (gate_delay: time:= 5 ns);
   port (d, clk, we: in std_logic;
      q, not_q: out std_logic);
   end entity my_dff;

   architecture behavioral of my_dff is
      begin
        my_block: block (rising_edge(clk) and (we = '1')) is
                begin
                q<= guarded d after gate_delay;
                not_q<= guarded (not d) after gate_delay;
                end block my_block;
       end architecture behavioral;
```

Value of the implicit guard signal

More on Processes: Postponed and Passive

- Postponed processes
  – Execute the processes after all delta events on sensitive signals
  – Reduction in number of process invocations → reduce simulation time
  – Reduction in the number of events inserted/removed from signal drivers → reduce simulation time
- Passive processes
  – These are processes that do not alter the simulation state
  – They can be placed to perform checks

Example

```
entity dff is
generic (sq_delay,
    rq_delay,cq_delay: time:=6 ns)
port (d, set, rst, clk :in bit;
q, notq: out bit);
end entity dff;

architecture behavioral of dff is
begin
process (rst, clk, set)
type bit_time is record
state : bit;
sd_delay: time;
end record:
variable sd: bit_time:= ('0', 0 ns);
begin
if set ='1' the
sd := ('1', sq_delay);
elsif rst = '1' then
sd := ('0', rq_delay);
elsif (rising_edge(clk)) then
sd := cq_delay;
end if;

q <= sd.state after sd.delay;
notq <= not sd.state after
    sd.delay;
end process;
end architecture behavioral;
```

Adapted from: Z. Navabi, "VHDL: Analysis and Modeling of Digital Systems", McGraw Hill 1998

Example: Passive Processes

```
package body of my_package is
begin
type bit_time is record
state : bit;
sd_delay: time;
end record:
shared variable sd: bit_time:= ('0', 0 ns);
end package;
```

```
entity dff is
    generic (sq_delay, rq_delay,cq_delay: time:=6 ns)
    port (  d, set, rst, clk :in bit;
            q, notq: out bit);
process
    begin
        if set ='1' the
                sd := ('1', sq_delay);
        elsif rst = '1' then
                sd := ('0', rq_delay);
        elsif (rising_edge(clk)) then
                sd := cq_delay;
        end if;
    end process;
end entity dff;

architecture behavioral of dff is
begin
q <= sd.state after sd.delay;
notq <= not sd.state after sd.delay;
end process;
end architecture behavioral;
```

Adapted from: Z. Navabi, "VHDL: Analysis and Modeling of Digital Systems", McGraw Hill 1998

Disconnect Specification

- A signal can be disconnected from its driver by assigning the NULL transaction
  - The value then is determined by the signal *kind*
    - Register : use the last known value

      signal s1 : wired_or bus;
    - Bus: use a resolution function

      signal s2 : bit register:

- The availability of the disconnect specification