Type Definition Examples

utilities directory has a package called *memory* that defines types/functions/procedures that are useful for memory modeling

```
TYPE Bit_Memory      IS ARRAY (Natural RANGE <>) OF Std_Ulogic;
TYPE Nibble_Memory   IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(3 DOWNTO 0);
TYPE Byte_Memory     IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(7 DOWNTO 0);
TYPE Word_Memory     IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(15 DOWNTO 0);
TYPE LongWord_Memory IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(31 DOWNTO 0);
```


Multi-dimensional Array

This is an example of a multi-dimensional array type declaration.

```
TYPE Byte_Memory  IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(7 DOWNTO 0);
```

unconstrained — constrained

Only one array index range can be unconstrained :
i.e. "Natural Range <> "

The other ranges must be constrained. Would be illegal to define the type as:

```
TYPE A_Memory  IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(Natural RANGE <>);
```


Array Assignments

When assigning one array to another, the *slice* size must be the same as well as the data type.

```
TYPE Byte_Memory  IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(7
DOWNTO 0);
 TYPE Word_Memory  IS ARRAY (Natural RANGE <>) OF Std_Ulogic_Vector(15
DOWNTO 0);

variable   a_mem: Byte_Memory(0 to 1023);
variable   b_mem: Byte_Memory(0 to 2047);
variable   c_mem : Word_Memory (0 to 511);

a_mem (3 to 10) :=  b_mem (11 to 18);
a_mem (20 to 30) := b_mem (20 to 40);
c_mem(2)  := a_mem (2);
```

legal, slice size is the same.

illegal, slice size is different.

elements are different.

RECORD types

A *record* type is a composite object type whose elements are named:

```
type  myrec is record
   some_real:    real;
   some_int:     integer;
   a_string:     string (1 to 5);
   a_bool:       boolean;
end record;
```

Usage example:

```
variable   tmp;  myrec;
.....
tmp.some_real := -30.4
tmp. some_int  :=  10;
tmp.a_string    :=  "Hello";
tmp.a_bool      := TRUE;
```

Signals can be record types!!! Can be helpful for complex modeling.

Assigning Default Values to a Record Type

```
type  myrec is record
     some_real:    real;
     some_int:     integer;
     a_string:     string (1 to 5);
     a_bool:       boolean;
  end record;
```

Positional assignment (order is same as in record declaration)

```
variable   tmp: myrec := (1.0, -1, "hello", TRUE)
```

OR

```
variable   tmp:  myrec := (some_int => -1,
                           a_string => "hello",
                           some_real => 1.0,
                           a_bool => FALSE);
```

Using a name list, order is not important because record field names are used for assignment

Example: Tracking Transition Counts

Signal transition counts are useful in gate-level simulations for power estimation programs.

Each signal transition consumes power.

Transient computations (transient gate level switching) can be a large source of power consumption in some cases.

Problem: Have a netlist that has D-flip flops (DFFs) + LUT4 (4-input lookup tables).

1. Would like to track the total number of signal transitions over a period of time.
2. Would like to distinguish between DFF and LUT4 transitions.

power Package

```
package pwr is
--- power functions
 type power_model;
 type power_ptr is access power_model;

 type power_model is
  record
    iname:line;
    tcnt:integer;
    vcnt:integer;
   next_model: power_ptr;
  end record power_model;

 shared variable first_model: power_ptr := null;
 shared variable dopwr: boolean := FALSE;
 procedure report_power(fname:String;  samples: integer);
 procedure clear_stats;
end package pwr;
```

Pointer type to our record data type

Record type for tracking transitions.

'tcnt' incremented if DFF transition, 'vcnt' incremented if LUT4 transition.

'iname' is instance name.

'next_model' is pointer to next record.

Global variables

Head of linked list of power_model records.

Global Variables

- A variable can be declared outside of a procedure, function or process only if it is declared as *shared*
  - Will make this variable visible to all procedures, functions, processes
- Useful for keeping track of statistics, global data structures
- Must remember: cannot predict the order in which a global variable will be updated if multiple processes update it, and the processes are all triggered by the same event
  - Order of process execution for simultaneous event triggering is simulator dependent

Modeling Approach

- At startup, each instance in our netlist will create a record of type 'power_model'.
  - Insert this into a linked list of all such power_model records
  - A global shared variable will be used to point to the head of this linked list
- Each time a signal transition occurs on an input, increment a counter in the power_model
  - For DFFs, increment 'tcnt'
  - For LUT4s, increment 'vcnt'.
- Can enable/disable transition counter via a global variable called *dopwr*
  - Only increment transition counts if this variable is TRUE
- Print transition stats using 'report_power' procedure
- Clear stats using 'clear_stats' procedure

dfr model

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

architecture a of dfr is
begin
 process(clk2,rst3)
  variable model: power_ptr := null;
  begin
   if (model = null) then
    model := new power_model'( new string'(a'instance_name), 0, 0, first_model);
    first_model := model;
   end if;

   if (rst3 = '0') then
    q <= transport '0' after gdelay;
   elsif (clk2'event and clk2 = '1') then
    q <= transport data1 after gdelay;
    -- increment tcnt of DFFs for EVERY clock edge
    if (dopwr = TRUE) then model.tcnt := model.tcnt + 1;
    end if;
   end if;
  end process;
end a;
```

'new' allocates new record structure.

'*instance* name returns simulator dependent name

link into list of all such record structures

dopwr true if statistics keeping turned on, increment transition count

Other Comments

The code below allocates the new record structure and links into the global list.

```
process(clk2,rst3)
   variable model: power_ptr := null;
   begin
    if (model = null) then
      model := new power_model'( new string'(a'instance_name), 0, 0, first_model);
      first_model := model;
    end if;
```

Note that we have no guarantee of what order the processes corresponding to the DFRs/LUT4s are initially executed in so there is no particular order of the power records on the global linked list.

The code is executed only once since 'model' will be non-null afterwards. The initial value of 'first-model' global variable is null, so last record will have a 'next_model' value of NULL.

'*instance_name* Attribute

```
process(clk2,rst3)
   variable model: power_ptr := null;
   begin
    if (model = null) then
      model := new power_model'( new string'(a'instance_name), 0, 0, first_model);
      first_model := model;
    end if;
```

"a" is the architecture name of this entity.

a'*instance_name* returns a simulator dependent string that describes the hierarchical path from the root of the design heirarchy down to this component architecture.

'*instance_name* can be used with anything other than local ports or generics of a component declaration.

LUT4 Entity

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

entity lut4 is
    generic (
            gdelay: time := 5 ns;
            fmap: std_logic_vector(15 downto 0):= "XXXXXXXXXXXXXXXX");
    port( A, B, C, D : in std_logic;  O : out std_logic);
end lut4;
```

LUT4 is a 4-input LookUp table (such as used in Xilinx, Altera FPGAs).

Equivalent to a 16 x 1 SRAM (inputs A,B,C,D are address lines where A is MSB, D is LSB).

Generic 'fmap' used to specify contents of LUT4.


```
library IEEE; use IEEE.std_logic_1164.all;
library power;use power.pwr.all;
architecture a of lut4 is
 begin
  process (A,B,C,D)
   variable index, lastval:integer;
   variable lasttrig: time := 0 ns;
   variable model: power_ptr := null;
   begin
    if (model = null) then
      model := new power_model'( new string'(a'instance_name),0, 0, first_model);
      first_model := model;
    end if;
    index := 0;
    if (A = '1') then      index := index + 8;    end if;
    if (B = '1') then      index := index + 4;    end if;
    if (C = '1') then      index := index + 2;    end if;
    if (D = '1') then      index := index + 1;    end if;
   O <= transport fmap(index) after gdelay;
   if (lastval /= index and ((now - lasttrig) > 1 ns)) then
      if (dopwr = TRUE and (not nopower)) then       model.vcnt := model.vcnt + 1;
      end if;
      lastval := index;     lasttrig := NOW;
   end if;
  end process;
end a;
```

LUT4 Architecture

Compute LUT4 address

Only count transition if current address is different from last address. Filter spikes < 1 ns.

Traversing the Record List

clear_stats procedure is used to zero out statistics after recording some signal transitions.

```
procedure clear_stats   is
   variable head_ptr: power_ptr;
   begin
     head_ptr := first_model;
     while head_ptr /= null loop
           head_ptr.tcnt :=0;
           head_ptr.vcnt :=0;
           head_ptr := head_ptr.next_model;
     end loop;
  end;
```

report_power procedure traverses list in a similar fashion except it sums the transition counts and prints out values to screen

Comments on Packages

Packages consists of a *package declaration* and a *package body*.

Any subprogram (function, procedure) or variable that is to be public to users of this package must be in the declaration.

If an element is not in the declaration but is in the body, then the element can only be used by other elements in the body – it is not 'visible' externally of the body.

```
package test is
   function foo (a: integer) return integer;
   CONSTANT  aconst: real := 3.14;
end test;

package body test is
  CONSTANT PCONST: integer := -1;

  function foo (a: integer) is
   begin
     return ((a+1) * PCONST);
   end  foo;
end test;
```

Public

Private

foo implementation in package body

Package Dependencies

- If an entity/package/configuration uses a package, and that package declaration is changed, then must recompile the the entity/package/configuration that uses the package
  - If only package body changes, then don't have to recompile as long as package declaration and package body are in different files.
- The value of a CONSTANT does not have to be specified in the package declaration:

```
package test is
 CONSTANT  PI: real;
end test;

package body test is
  CONSTANT  PI: real := 3.14159;
end test;
```

Called a *deferred* constant. Can change this value without having to recompile dependent packages, entities, configurations.

An Example Procedure from *memory* package

```
PROCEDURE MemInit (MemoryName : INOUT Byte_Memory;
                   FillBit : Std_Ulogic) IS

 BEGIN
   MemoryName := (MemoryName'range =>
                  (MemoryName(MemoryName'left)'range => FillBit));
 END MemInit;
```

INOUT - needed if you read and write to parameter.

MemoryName'range – returns range (i.e. 0 to 127).
MemoryName(MemoryName'left) returns left most element.

generic way of writing:
MemoryName := (0 to 127 => (7 downto 0 => Fillbit))

An Example *function* from *memory* package

```
FUNCTION MemRead (MemoryName : Byte_Memory;
                  Address :   Std_Ulogic_Vector)
        RETURN Std_Ulogic_Vector IS
BEGIN
  IF (Is_X(Address)) THEN
    RETURN (MemoryName(MemoryName'left)'range => 'X');
  ELSE
    RETURN (MemoryName(To_Integer(Address)));
  END IF;
END MemRead;
```

What to do if address contains an 'X'?

Type conversion to integer type since index is of type NATURAL

An Example *procedure* from *memory* package

```
PROCEDURE MemWrite (MemoryName : INOUT Byte_Memory;
                    Address : Std_Ulogic_Vector;
                    Data : Std_Ulogic_Vector) IS
BEGIN
  IF (Is_X(Address)) THEN
    NULL;
  ELSE
    MemoryName(To_Integer(Address)) := Data;
  END IF;
END MemWrite;
```

Assign data to specified memory location.