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:

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

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

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

```vhdl
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));
```

Array Assignments

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

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

```vhdl
a_mem (3 to 10) := b_mem (11 to 18);
```

Illegal, slice size is different.

```vhdl
c_mem(2)  := a_mem (2);
```

Elements are different.

Array Assignments

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

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

```vhdl
a_mem (3 to 10) := b_mem (11 to 18);
```

Illegal, slice size is different.

```vhdl
c_mem(2)  := a_mem (2);
```

Elements are different.

Assigning Default Values to a Record Type

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

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

Usage example:

```vhdl
variable tmp: myrec;
```

```vhdl
tmp.some_real := -30.4
```

```vhdl
tmp.some_int := 10;
```

```vhdl
tmp.a_string := "Hello";
```

```vhdl
tmp.a_bool := TRUE;
```

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

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.
package pwr is
   -- power functions
   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;

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

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);
end if;
end process;

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);
end if;
end process;

'dopwr' true if statistics keeping turned on, increment transition count

'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);
end if;
end process;

'a' as 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 hierarchy 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.

LUT4 Architecture

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;

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.

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;

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

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 := 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) )
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;  

An Example function from memory package

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;