Subprograms

Prof. K. J. Hintz

Department of Electrical
and
Computer Engineering
George Mason University

Structural Descriptions

- Port Mappings Can Be Specified Two Ways
  - Positional association
    - Order is important
  - Named association
    - For connecting some signals
    - Easier to read which signals are connected to which internal signals
    - Order is not important
  - Can only use one type of association at a time

Structural Descriptions

- Entity Ports Declared in an Architecture Body Are Signals
  - These signals are not available outside the architecture unless connected to one of the architectures ports

Structural Descriptions

- Subelement Association
  - Separate, actual signals can be associated with individual elements of the architecture port
  - All elements of the composite port must be associated
  - All associations for a particular port must be grouped together with no intervening ports among them.
  - Ranges can be associated

Design Processing

- Simulation and Synthesis Require
  - Analysis of model
    - Performs syntax check
    - Places entities and architectures in design library
    - Dependency relations
      - primary units contain entity declarations
      - secondary units contain architecture bodies

Design Processing

- Elaboration
  - A collection of processes connected by nets
  - Entity declarations replaced by architecture bodies

Execution

- Simulation Time Set to Zero
- Signals Assigned Initial Values
- Every Process Executed at Least Once
- Advance Simulation Clock to Next Scheduled Timeout
- Perform Transactions and Update Signals
- Resume Sensitive Processes

Subprograms

- Define Algorithms for Computing Values or Exhibiting Behavior
  - Type conversion
  - Define resolution functions for signals
  - Define portions of a process

Subprograms

- Two types
  - Procedure
  - Function
- Return Statements Allow for Different Exit Points
  - Function must always have a return statement
  - Procedure does not return a value

Subprograms

- Declared Variables, Constants and Files Are Local and Instantiated When Called.
  - No signals allowed
- Procedure declarations can be nested
- Procedures can call procedures

Value of Subprograms

- Write Once, Use Often
- Can Be Called Recursively
- Can Be Called Repeatedly From Within Scope

Procedure Subprograms

Procedure

- Encapsulates a collection of sequential statements into a single statement
- Executed for their effect
- May return zero or more values
- May execute in zero or more simulation time
- Can modify a calling parameter because it is a statement and doesn’t return a value

Function Subprograms

- Functions
  - Algorithmically generates and returns only one value
    - May be on right hand side of expression
  - Must Alway returns a value
  - Executes in zero simulation time
    - *i.e.*, cannot contain a wait statement

Subprogram Declaration

- Names the Subprogram
- Specifies Required Parameters
- Contains the Sequential Statements Defining the Behavior of the Subprogram
- Defines the Return Type for Function Subprograms

Subprogram Declaration

- Local Declarations
  - Types
  - Subtypes
  - Constants and variables
  - Nested subprogram declarations

Procedure Syntax

```
procedure identifier
 [ parameter_interface _list ] is
 { subprogram_declarative_part }
 begin
  { sequential_statement }
 end [ procedure ] [ identifier ] ;
```

Procedure Syntax

```
parameter_interface _list <=

( [ constant | variable | signal ]  identifier
        { , . . . } :
         [ mode ] subtype_indication
         [ := static_expression ] )
        { ; . . . }
```

Procedure Parameter List

- Specifies Class of Object(s) Passed
  - **`Constant`** (assumed if mode is **`in`**)
  - **`Variable`** (assumed if mode is **`out`**)
  - **`Signal`** (passed by reference, not value)
    - » If **`wait`** statement is executed inside a procedure, the value of a signal may change before the rest of the procedure is calculated
    - » If mode **`inout`**, reference to both signal and driver passed

Procedure Parameter List

- Associates Identifier With Formal Parameter(s)
  - Allows reference to a parameter in procedure body
  - Formal parameters are replaced with actual values when called

Procedure Parameter List

- Specifies optional mode(s)
  - **`in`**
    - » assumed if not specified
  - **`out`**
    - » cannot use value for computations
  - **`inout`**
    - » both read & write

Procedure Parameter List

- Specifies Type(s)
  - Provides error checking for type mismatches
  - Unconstrained arrays are allowed (<>)
    - Attributes of unconstrained arrays can be used to set local constants or variables within procedures such as looping parameters

Procedure Parameter List

- Specifies Optional Default Value(s)
  - Values to be used if a parameter is not specified
  - If default value is desired, use keyword **open** for parameter
  - If default value is at end of list, can omit actual value or use **open**

Procedure Example*

```
procedure do_arith_op (
 op1, op2 : in integer ;
 op          : in func_code ) is
 variable result : integer ;
  begin
   case op is
    when add => result := op1 + op2 ;
    when subtract => result := op1-op2 ;
   end case ;
```

*Ashenden, p 197

Procedure Example*

```
  dest   <=  result         after Tpd ;
  Z_flag <= (result = 0) after Tpd;
end procedure do_arith_op ;
```

***Ashenden, p 197**

Procedure Calling

- Once a Procedure is Declared, It Can Be Called From Other Parts of Model
- A Procedure Is a Sequential Statement, So it Can Be Called From
  - Process
  - Other subprogram body

Procedure Calling Syntax

```
[  label : ]  procedure_name
   [  parameter_association_list ]   ;

parameter_association_list <=
  ( [ parameter_name => ] expression
    | signal_name
    | variable_name
    | open )
    { ,   . . . }
```

Procedure Calling

- Same Syntax As Ports
  - Positional association
  - Named association
  - Mix Positional and named
    - All Positional parameters must come first

Concurrent Procedure Calling

- A Shorthand for A Procedure Call Where Only a Concurrent Statement Would Be Allowed
- Identical Syntax to Procedure Call Statement
- Sensitive to Non-constant Parameters Passed to It

Concurrent Procedure Example*

```
procedure check_setup
  ( signal data, clock : in bit ;
    constant Tsu : in time ) is
begin
   if (clock'event and clock = '1') then
     assert data'last_event >= Tsu
    report "setup time violation" severity
  error ;
  end if ;
end procedure check_setup ;
```

***Ashenden, p 208**

Concurrent Procedure Example*

```
check_ready_setup : check_setup
            ( data => ready ,
              clock => phi2 ,
              Tsu => Tsu_rdy_clk ) ;
```

- Formal Parameters
  - data, clock, Tsu
- Actual Parameters
  - ready, phi2, Tsu_rdy_clk
  - Procedure is sensitive to signals ready and phi2

Concurrent Procedures

- Advantages
  - Easier to read programs
  - Write once, use often
  - Check timing constraints

Concurrent Procedures

- If No **`in`** or **`inout`** Signals in Parameter List
  - No sensitivity list, hence no equivalent wait statement
  - If procedure returns, it suspends indefinitely
  - Desirable if want to execute procedure once at startup
  - If wait statements are included in procedure, then behaves like process

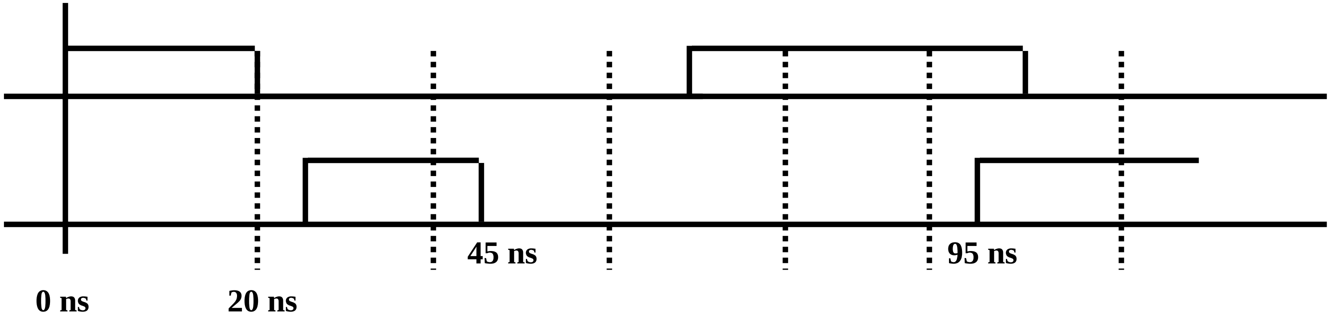
Concurrent Procedure Example*

```
procedure generate_clock
 ( signal clk : out bit ;
   constant Tperiod, Tpulse, Tphase : in time )
 is
begin
 wait for Tphase ;
 loop
  clk <= ‘1’, ‘0’ after Tpulse ;
  wait for Tperiod ;
 end loop ;
end procedure generate_clock ;
```

***Ashenden, p 208**

Concurrent Procedure Example*

```
signal phi1, phi2 : std_ulogic := ‘0’ ;
...
gen_phi1:generate_clock(phi1, Tperiod  => 50n ,
          Tpulse => 20 ns , Tphase => 0 ns ) ;
gen_phi2:generate_clock(phi2,Tperiod => 50 ns ,
          Tpulse => 20 ns , Tphase => 25 ns ) ;
```



Function Subprograms

- Generalized Expression
- Allows Definition of New Operators in Addition to Standard Ones (+, -, *, *etc.*)
- Allows Overloading of Standard Operators
- Must Contain at Least One **Return** Statement

```
[  label : ]  return expression ;
```

Function Syntax

```
[ pure | impure ]  function identifier
 [ parameter_interface_list ]
 return type_mark is
  { subprogram_declarative_part }
  begin
   { sequential_statement }
   [  label : ]  return expression ;
end [ function ] [ identifier ]  ;
```

Function Syntax

parameter_interface _list <=

([**constant** | **signal**] identifier{ , . . . }:

[**in**] *subtype_indication* [**:=** *static_expression*])

{ ; . . . }

Function Example*

```
function byte_to_int ( byte : word_8 )
 return integer is
  variable result : integer := 0 ;
begin
 for index in 0 to 7 loop
 result := result*2 + bit'pos(byte (index) ) ;
end loop ;
 return result ;
end function byte_to_int ;
```

***Ashenden, VHDL Cookbook**

Function Parameter List

- Specifies Class of Object(s) Passed
  - Constant (assumed)
  - Variable class is NOT allowed since the result of operations could be different when different instantiations are executed
  - Signal (passed by reference, not value)

Function Parameter List

- Associates Identifier With Formal Parameter(s)
  - Allows reference to a parameter in procedure body
  - Formal parameters are replaced with actual values when called
- Specifies Type of Return Value

Function Parameter List

- Optionally Specifies mode
  - `in` is the ONLY allowed mode
- Specifies Type(s)
  - Provides error checking for type mismatches
  - Unconstrained arrays are allowed (<>)
    - » Attributes of unconstrained arrays can be used to set local constants or variables within procedures such as looping parameters

Function Parameter List

- Specifies Optional Default Value(s)
  - Values to be used if a parameter is not specified
  - If default value is desired, use keyword **open** for parameter
  - If default value is at end of list, omit actual value or **open**

Function Calling

- Once Declared, Can Be Used in Any Expression
- A Function Is Not a Sequential Statement So It Is Called As Part of an Expression

```
[  label : ]   function_name
          [ parameter_association_list ] ;
```

Function Calling

- Same Syntax As Ports
  - Positional association
  - Named association
  - Mix positional and named
    - » All positional parameters must come first

```
(   [   parameter_name  =>  ]
        expression  |  signal_name
      | variable_name  |  open
  { ,  . . . }  )
```

Pure Functions

- Function Does Not Refer to Any Variables or Signals Declared by Parent
- Result of Function Only Depends on Parameters Passed to It
- Always Returns the Same Value for Same Passed Parameters
- If Not Stated Explicitly, a Function Is Assumed to Be Pure

Impure Functions

- Can State Explicitly and Hence Use Parents’ Variables and/or Signals for Function Computation
- May Not Always Return the Same Value
- *e.g.*,

```
Impure Function Now
        Return Delay_Length ;
```

Overloading

- Same Operation on Different Types
- More Than One Distinct Subprogram Can Be Defined With the Same Name Where Each Has
  - Different parameter types
  - Different number of parameters
- Context and Parameter List Determine Which Subprogram Is Executed

Overloading Example*

```
procedure increment ( a : inout integer ;
          n : in integer := 1  )
          is . . .
procedure increment (a : inout bit_vector ;
          n : in bit_vector := b”1” )
          is . . .
procedure increment (a : inout bit_vector ;
          n : in integer := 1 ) is . . .
```

***Ashenden, p 215**

Overloading Symbols

- Predefined Arithmetic Symbols Can Also Be Overloaded
- One Could Define Mixed Type Arithmetic and Write a Function
- Overloaded Boolean Operators Are Not “Short-Circuit” Evaluated

Overloading Symbols

- Predefined Arithmetic Symbol Is Quoted in Function Declaration, *e.g.*,

```
function “+” (left, right : in bit_vector )
return bit_vector ...
```

Subprogram Declaration Visibility

- Visibility Follows Normal Scoping Rules

- All Variables Are Local and "directly visible"
  - Allows one to use function without insuring variable name has not been used before

Visibility of Non-Local Variables

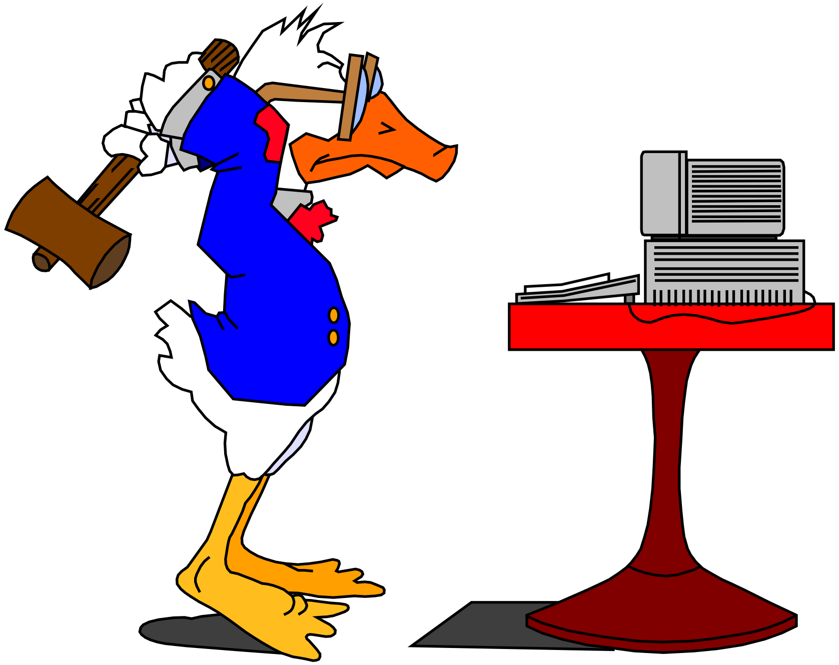
- Explicit reference can be made to non-local variables
  - "visible by selection"
  - prepend variable with name of procedure, *e.g.*,

```
p1.v
```

as opposed to directly visible local variable

```
v
```

End of Lecture



- Structural Model
- Procedures
- Functions
- Overloading