A bird’s eye view on VHDL

Politecnico di Milano
Conference Room, Bld 20
19 November, 2015

Antonio R. Miele
Marco D. Santambrogio
Politecnico di Milano
VHDL - Intro

- **VHSIC Hardware Description Language**
  - Very High Speed Integrated Circuit
  - Language for modeling ICs

- Three ways of describing ICs
  - Dataflow – Model the circuit using gate operators
  - Structural – Model the circuit using separate structural description
  - Behavioral – Using code (ifs, loops, etc.)

- Using simulation, logic designs can be verified
A first overview of a general description

use libreria

entity circuit is
  ...
end circuit;

architecture arch of circuit is
  -- istr 1
  -- iistr 2
begin
  pa: process
  begin
    -- istr pa_1
    -- iistr rpa_2
    -- ...
    end;
  pb: process
  begin
    -- istr pa_1
    -- ...
    end;
end;
A first example

A \leq B + C
entity COMPARE is
  port (a, b : in bit;
       c : out bit);
end COMPARE
Modeling Interfaces

- **Entity declaration**
  - describes the input/output *ports* of a module

```
entity reg4 is
  port ( d0, d1, d2, d3, en, clk : in bit;
        q0, q1, q2, q3 : out bit );
end entity;
```
VHDL Design Example

• Problem: Design a single bit half adder with carry and enable

• Specifications
  – Inputs and outputs are each one bit
  – When enable is high, result gets $x$ plus $y$
  – When enable is high, carry gets any carry of $x$ plus $y$
  – Outputs are zero when enable input is low
Entity Declarations

• The primary purpose of the entity is to declare the signals in the component’s interface
  – The interface signals are listed in the PORT clause
    • In this respect, the entity is akin to the schematic symbol for the component
  – Additional entity clauses and statements will be introduced later in this and subsequent modules

ENTITY half_adder IS
  GENERIC(prop_delay : TIME := 10 ns);
  PORT( x, y, enable : IN BIT;
        carry, result : OUT BIT);
END half_adder;
Entity Declarations

Port Clause

• PORT clause declares the interface signals of the
  object to the outside world

• Three parts of the PORT clause
  – Name
  – Mode
  – Data type

• Example PORT clause:

  PORT (signal_name : mode data_type);

  PORT ( input : IN BIT_VECTOR(3 DOWNTO 0);
        ready, output : OUT BIT );

  – Note port signals (i.e. ‘ports’) of the same mode and
    type or subtype may be declared on the same line
Entity Declarations
Port Clause (cont.)

• The port mode of the interface describes the direction in which data travels with respect to the component.

• The five available port modes are:
  – In - data comes in this port and can only be read
  – Out - data travels out this port
  – Buffer - data may travel in either direction, but only one signal driver may be on at any one time
  – Inout - data may travel in either direction with any number of active drivers allowed; requires a Bus Resolution Function
  – Linkage - direction of data flow is unknown
Entity Declarations

Generic Clause

• Generics may be used for readability, maintenance and configuration
• Generic clause syntax :

    GENERIC (generic_name : type [:= default_value]);

    – If optional default_value is missing in generic clause declaration, it must be present when component is to be used (i.e. instantiated)
• Generic clause example :

    GENERIC (My_ID : INTEGER := 37);

    – The generic My_ID, with a default value of 37, can be referenced by any architecture of the entity with this generic clause
    – The default can be overridden at component instantiation
Architecture Bodies

• Describe the operation of the component
• Consist of two parts:
  – Declarative part -- includes necessary declarations
    • e.g. type declarations, signal declarations, component declarations, subprogram declarations
  – Statement part -- includes statements that describe organization and/or functional operation of component
    • e.g. concurrent signal assignment statements, process statements, component instantiation statements

ARCHITECTURE half_adder_d OF half_adder IS
  SIGNAL xor_res : BIT;     -- architecture declarative part
  BEGIN                     -- begins architecture statement part
    carry <= enable AND (x AND y);
    result <= enable AND xor_res;
    xor_res <= x XOR y;
  END half_adder_d;
Modeling Behavior

• **Architecture body**
  – describes an implementation of an entity
  – may be several per entity

• **Behavioral architecture**
  – describes the algorithm performed by the module
  – contains
    • *process statements*, each containing
    • *sequential statements*, including
    • *signal assignment statements* and
    • *wait statements*
VHDL Design Example

Behavioral Specification

• A high level description can be used to describe the function of the adder

```vhdl
ARCHITECTURE half_adder_a OF half_adder IS
BEGIN
    PROCESS (x, y, enable)
    BEGIN
        IF enable = '1' THEN
            result <= x XOR y;
            carry <= x AND y;
        ELSE
            carry <= '0';
            result <= '0';
        END IF;
    END PROCESS;
END half_adder_a;
```

• The model can then be simulated to verify correct functionality of the component
VHDL Design Example
Data Flow Specification

• A second method is to use logic equations to develop a data flow description

```
ARCHITECTURE half_adder_b OF half_adder IS
BEGIN
    carry <= enable AND (x AND y);
    result <= enable AND (x XOR y);
END half_adder_b;
```

• Again, the model can be simulated at this level to confirm the logic equations
VHDL Design Example
Structural Specification

- As a third method, a structural description can be created from pre-described components.
- These gates can be pulled from a library of parts.

\[ x \\
\] \[ y \\
\] \[ \text{enable} \]

\[ \text{carry} \]
\[ \text{result} \]
VHDL Design Example
Structural Specification (Cont.)

ARCHITECTURE half_adder_c OF half_adder IS

COMPONENT and2
  PORT (in0, in1 : IN BIT;
  out0 : OUT BIT);
END COMPONENT;

COMPONENT and3
  PORT (in0, in1, in2 : IN BIT;
  out0 : OUT BIT);
END COMPONENT;

COMPONENT xor2
  PORT (in0, in1 : IN BIT;
  out0 : OUT BIT);
END COMPONENT;

FOR ALL : and2 USE ENTITY gate_lib.and2_Nty(and2_a);
FOR ALL : and3 USE ENTITY gate_lib.and3_Nty(and3_a);
FOR ALL : xor2 USE ENTITY gate_lib.xor2_Nty(xor2_a);

-- description is continued on next slide
-- continuing half_adder_c description

SIGNAL xor_res : BIT; -- internal signal
-- Note that other signals are already declared in entity

BEGIN

A0 : and2 PORT MAP (enable, xor_res, result);
A1 : and3 PORT MAP (x, y, enable, carry);
X0 : xor2 PORT MAP (x, y, xor_res);

END half_adder_c;
Simulation Cycle Revisited
Sequential vs Concurrent Statements

• VHDL is inherently a concurrent language
  – All VHDL processes execute concurrently
  – Concurrent signal assignment statements are actually one-line processes

• VHDL statements execute sequentially within a process

• Concurrent processes with sequential execution within a process offers maximum flexibility
  – Supports various levels of abstraction
  – Supports modeling of concurrent and sequential events as observed in real systems
Concurrent Statements

• Basic granularity of concurrency is the *process*
• Mechanism for achieving concurrency:
  – Processes communicate with each other via signals
  – Signal assignments require delay before new value is assumed
  – Simulation time advances when all active processes complete
  – Effect is concurrent processing
    • I.e. order in which processes are actually executed by simulator does not affect behavior

• Concurrent VHDL statements include:
  – Block, process, assert, signal assignment, procedure call, component instantiation
Sequential Statements

• Statements inside a *process* execute sequentially

```
ARCHITECTURE sequential OF test_mux IS
BEGIN
    select_proc : PROCESS (x,y)
    BEGIN
        IF (select_sig = '0') THEN
            z <= x;
        ELSIF (select_sig = '1') THEN
            z <= y;
        ELSE
            z <= "XXXX";
        END IF;
    END PROCESS select_proc;
END sequential;
```
Some examples...

```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```
Processes control: IF-THEN-ELSE

SIGNAL a : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL y : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL enable : STD_LOGIC;

PROCESS (enable)
BEGIN
  IF (enable = '1') THEN
    y <= a;
  ELSE
    y <= "000";
  END IF;
END PROCESS;
Processes control: CASE

SIGNAL a : STD_LOGIC;
SIGNAL y : STD_LOGIC;
SIGNAL enable : STD_LOGIC_VECTOR(1 downto 0);

PROCESS ( enable )
BEGIN
    CASE enable IS
        WHEN "00" =>
            y <= a;
        WHEN "01" =>
            y <= '0';
        WHEN "10" =>
            y <= '0';
        WHEN OTHERS =>
            y <= '1';
    END CASE;
END PROCESS;
SIGNAL y : STD_LOGIC_VECTOR(3 downto 0);
SIGNAL a : STD_LOGIC_VECTOR(1 downto 0);

PROCESS (a)
BEGIN
    CASE a IS
        WHEN "00" =>
            y <= "0001";
        WHEN "01" =>
            y <= "0010";
        WHEN "10" =>
            y <= "0100";
        WHEN "11" =>
            y <= "1000";
        WHEN OTHERS => ;
    END CASE;
END PROCESS;
entity ffd is
port(
    d: in std_logic;
    clk: in std_logic;
    q: out std_logic
);
end ffd;

ARCHITECTURE arch OF ffd IS
BEGIN
    PROCESS(clk)
    BEGIN
        IF (clk'EVENT AND clk = '1') THEN
            q <= d;
        END IF;
    END PROCESS;
END arch;
Sequential Circuit – FFD - Reset

entity ffd is
port(
d: in std_logic;
clk: in std_logic;
reset: in std_logic;
q: out std_logic
);
end ffd;

ARCHITECTURE arch OF ffd IS
BEGIN
PROCESS(clk, reset)
BEGIN
IF (reset = '1') THEN
q <= '0';
ELSIF (clk'EVENT AND clk = '1') THEN
q <= d;
END IF;
END PROCESS;
END arch;
Counter

ARCHITECTURE arch OF counter IS
BEGIN
  PROCESS(clk, reset)
  BEGIN
    IF (reset = '1') THEN
      q <= '000';
    ELSIF (clk'EVENT AND clk = '1') THEN
      IF q >= 9 THEN
        q <= '000';
      ELSE
        q <= q + 1;
      END IF;
    END IF;
  END PROCESS;
END arch;
ARCHITECTURE arch OF circ IS
SIGNAL s : BIT;
BEGIN
    PROCESS
    BEGIN
        s <= OP 1;
    END PROCESS;
    PROCESS
    BEGIN
        s <= OP 2;
    END PROCESS;
END arch;
FSM 1/3 - Entity

library ieee;
use ieee.std_logic_1164.all;

entity seq_design is
port(
a: in std_logic;
clock: in std_logic;
reset: in std_logic;
x: out std_logic
);
end seq_design;
FSM 2/3 – Architecture: process#1

architecture FSM of seq_design is

-- define the states of FSM model

type state_type is (S0, S1, S2, S3);
signal next_state, current_state: state_type;

begin

-- cocurrent process#1: state registers

state_reg: process(clock, reset)
begin

    if (reset='1') then
        current_state <= S0;
    elsif (clock'event and clock='1') then
        current_state <= next_state;
    end if;

end process;
-- concurrent process#2: combinational logic

comb_logic: process(current_state, a)
begin

-- use case statement to show the
-- state transition

    case current_state is

        when S0 =>
            x <= '0';
            if a='0' then
                next_state <= S0;
            elsif a='1' then
                next_state <= S1;
            end if;

        when S1 =>
            x <= '0';
            if a='0' then
                next_state <= S1;
            elsif a='1' then
                next_state <= S2;
            end if;

        when S2 =>
            x <= '0';
            if a='0' then
                next_state <= S2;
            elsif a='1' then
                next_state <= S3;
            end if;

        when S3 =>
            x <= '1';
            if a='0' then
                next_state <= S3;
            elsif a='1' then
                next_state <= S0;
            end if;

        when others =>
            x <= '0';
            next_state <= S0;

    end case;

end process;

end FSM;
Half Adder

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY HalvAdder IS
    PORT (
        a : IN  std_logic;
        b : IN  std_logic;
        s : OUT std_logic_vector(1 DOWNTO 0));
END HalvAdder;

ARCHITECTURE Bool OF HalvAdder IS
BEGIN  -- Bool
    s(1) <= a AND b;
    s(0) <= a XOR b;
END Bool;
VHDL: Data Types, Objects and Operator
Data Types

All declarations of VHDL ports, signals, and variables must specify their corresponding type or subtype.
VHDL Data Types

Scalar Types

- **Integer**
  - Minimum range for any implementation as defined by standard: -2,147,483,647 to 2,147,483,647
  - Example assignments to a variable of type integer:

```vhdl
ARCHITECTURE test_int OF test IS
BEGIN
  PROCESS (X)
  BEGIN
    VARIABLE a: INTEGER;

    a := 1;  -- OK
    a := -1;  -- OK
    a := 1.0;  -- illegal
  END PROCESS;
END test_int;
```
VHDL Data Types
Scalar Types (Cont.)

- Real
  - Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38
  - Example assignments to a variable of type real:

```vhdl
ARCHITECTURE test_real OF test IS
BEGIN
  PROCESS (X)
  VARIABLE a: REAL;
  BEGIN
    a := 1.3;  -- OK
    a := -7.5;  -- OK
    a := 1;  -- illegal
    a := 1.7E13;  -- OK
    a := 5.3 ns;  -- illegal
  END PROCESS;
END test_real;
```
VHDL Data Types
Scalar Types (Cont.)

• Enumerated
  – User specifies list of possible values
  – Example declaration and usage of enumerated data type:

```vhdl
TYPE binary IS ( ON, OFF );
... some statements ...
ARCHITECTURE test_enum OF test IS
BEGIN
  PROCESS (X)
    VARIABLE a: binary;
  BEGIN
    a := ON;  -- OK
    ... more statements ...
    a := OFF;  -- OK
    ... more statements ...
  END PROCESS;
END test_enum;
```
VHDL Data Types
Scalar Types (Cont.)

• Physical
  – Require associated units
  – Range must be specified
  – Example of physical type declaration:

```vhdl
TYPE resistance IS RANGE 0 TO 10000000
UNITS
  ohm;   -- ohm
  Kohm = 1000 ohm;  -- i.e. 1 KΩ
  Mohm = 1000 kohm;  -- i.e. 1 MΩ
END UNITS;
```

– Time is the only physical type predefined in VHDL standard
VHDL Data Types

Composite Types

• Array
  – Used to group elements of the same type into a single VHDL object
  – Range may be unconstrained in declaration
    • Range would then be constrained when array is used
  – Example declaration for one-dimensional array (vector)

```vhdl
TYPE data_bus IS ARRAY(0 TO 31) OF BIT;

VARIABLE X : data_bus;
VARIABLE Y : BIT;

Y := X(12); -- Y gets value of element at index 12
```
VHDL Data Types

Composite Types (Cont.)

• Records
  – Used to group elements of possibly different types into a single VHDL object
  – Elements are indexed via field names
  – Examples of record declaration and usage:

```vhdl
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
  RECORD
    status : BINARY;
    IDnumber : INTEGER;
  END RECORD;

VARIABLE switch : switch_info;
switch.status := ON;  -- status of the switch
switch.IDnumber := 30;  -- e.g. number of the switch
```
VHDL Objects

• There are four types of objects in VHDL
  – Constants
  – Variables
  – Signals
  – Files

• The scope of an object is as follows:
  – Objects declared in a package are available to all VHDL descriptions that use that package
  – Objects declared in an entity are available to all architectures associated with that entity
  – Objects declared in an architecture are available to all statements in that architecture
  – Objects declared in a process are available only within that process
VHDL Objects

Constants

• Name assigned to a specific value of a type
• Allow for easy update and readability
• Declaration of constant may omit value so that the value assignment may be deferred
  – Facilitates reconfiguration
• Declaration syntax:

  
  $\text{CONSTANT } \text{constant}_\text{name} : \text{type}_\text{name} [:= \text{value}]$;

• Declaration examples:

  
  $\text{CONSTANT PI : REAL := 3.14;}$
  $\text{CONSTANT SPEED : INTEGER;}$
VHDL Objects

Variables

• Provide convenient mechanism for local storage
  – E.g. loop counters, intermediate values
• Scope is process in which they are declared
  – VHDL ‘93 provides for global variables, to be discussed in the Advanced Concepts in VHDL module
• All variable assignments take place immediately
  – No delta or user specified delay is incurred
• Declaration syntax:
  ```vhdl
  VARIABLE variable_name : type_name [:= value];
  ```
• Declaration examples:
  ```vhdl
  VARIABLE opcode : BIT_VECTOR(3 DOWNTO 0) := "0000";
  VARIABLE freq : INTEGER;
  ```
VHDL Objects

Signals

• Used for communication between VHDL components
• Real, physical signals in system often mapped to VHDL signals
• ALL VHDL signal assignments require either delta cycle or user-specified delay before new value is assumed
• Declaration syntax :

  SIGNAL signal_name : type_name [:= value];

• Declaration and assignment examples :

  SIGNAL brdy : BIT;
  brdy <= '0' AFTER 5ns, '1' AFTER 10ns;
Signals and Variables

- This example highlights the difference between signals and variables

ARCHITECTURE test1 OF mux IS
  SIGNAL x : BIT := '1';
  SIGNAL y : BIT := '0';
BEGIN
  PROCESS (in_sig, x, y)
  BEGIN
    x <= in_sig XOR y;
    y <= in_sig XOR x;
  END PROCESS;
END test1;

ARCHITECTURE test2 OF mux IS
  SIGNAL y : BIT := '0';
BEGIN
  PROCESS (in_sig, y)
  VARIABLE x : BIT := '1';
  BEGIN
    x := in_sig XOR y;
    y <= in_sig XOR x;
  END PROCESS;
END test2;
VHDL Objects

Files

- Files provide a way for a VHDL design to communicate with the host environment
- File declarations make a file available for use to a design
- Files can be opened for reading and writing
  - In VHDL87, files are opened and closed when their associated objects come into and out of scope
  - In VHDL93 explicit `FILE_OPEN()` and `FILE_CLOSE()` procedures were added
- The package STANDARD defines basic file I/O routines for VHDL types
- The package TEXTIO defines more powerful routines handling I/O of text files
Operators

• Operators can be chained to form complex expressions, e.g.:

```
res <= a AND NOT(B) OR NOT(a) AND b;
```

  – Can use parentheses for readability and to control the association of operators and operands

• Defined precedence levels in decreasing order:
  – Miscellaneous operators -- **, abs, not
  – Multiplication operators -- *, /, mod, rem
  – Sign operator -- +, -
  – Addition operators -- +, -, &
  – Shift operators -- sll, srl, sla, sra, rol, ror
  – Relational operators -- =, /=, <, <=, >, >=
  – Logical operators -- AND, OR, NAND, NOR, XOR, XNOR
The concatenation operator &

VARIABLE shifted, shiftin : BIT_VECTOR(0 TO 3);
...
shifted := shiftin(1 TO 3) & '0';

<table>
<thead>
<tr>
<th>SHIFTIN</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Modeling Digital Systems
Modeling Digital Systems

• We seek to describe attributes of digital systems common to multiple levels of abstraction
  – events, propagation delays, concurrency
  – waveforms and timing
  – signal values
  – shared signals

• Hardware description languages must provide constructs for naturally describing these attributes of a specific design
  – simulators use such descriptions for “mimicing” the physical system
  – synthesis compilers use such descriptions for synthesizing manufacturable hardware specifications that conform to this description
Execution Models for VHDL Programs

• Two classes of execution models govern the application of VHDL programs

• For Simulation
  – Discrete event simulation
  – Understanding is invaluable in debugging programs

• For Synthesis
  – Hardware inference
  – The resulting circuit is a function of the building blocks used for implementation
    • Primitives: NAND vs. NOR
    • Cost/performance
Simulation Modeling

- VHDL programs describe the generation of events in digital systems
- Discrete event simulator manages event ordering and progression of time
- Now we can quantitatively understand accuracy vs. time trade-offs
  - Greater detail → more events → greater accuracy
  - Less detail → smaller number of events → faster simulation speed
Test Benches

• Testing a design by simulation
• Use a *test bench* model
  – an architecture body that includes an instance of the design under test
  – applies sequences of test values to inputs
  – monitors values on output signals
    • either using simulator
    • or with a process that verifies correct operation
Design Processing
Design Processing

- Analysis
- Elaboration
- Simulation
- Synthesis
Analysis

• Check for syntax and semantic errors
  – syntax: grammar of the language
  – semantics: the meaning of the model
• Analyze each *design unit* separately
  – entity declaration
  – architecture body
  – ...
  – best if each design unit is in a separate file
• Analyzed design units are placed in a *library*
  – in an implementation dependent internal form
  – current library is called *work*
Elaboration

• “Flattening” the design hierarchy
  – create ports
  – create signals and processes within architecture body
  – for each component instance, copy instantiated entity and architecture body
  – repeat recursively
    • bottom out at purely behavioral architecture bodies

• Final result of elaboration
  – flat collection of signal nets and processes
Elaboration Example

```
reg4(struct)

bit0
  d_latch
  d   q
  clk
  q0

bit1
  d_latch
  d   q
  clk
  q1

gate
  and2
  a  y
  b

bit2
  d_latch
  d   q
  clk
  q2

bit3
  d_latch
  d   q
  clk
  q3

d0

d1

d2

d3

en

clk

int_clk
```
Elaboration Example

process with variables and statements
Simulation

- Execution of the processes in the elaborated model
- Discrete event simulation
  - time advances in discrete steps
  - when signal values change—events
- A processes is sensitive to events on input signals
  - resumes and schedules new values on output signals
  - schedules transactions
  - event on a signal if new value different from old value
Simulation Algorithm

• Initialization phase
  – each signal is given its initial value
  – simulation time set to 0
  – for each process
    • activate
    • execute until a wait statement, then suspend
      – execution usually involves scheduling transactions on signals
        for later times
Simulation Algorithm

• Simulation cycle
  – advance simulation time to time of next transaction
  – for each transaction at this time
    • update signal value
      – event if new value is different from old value
    – for each process sensitive to any of these events, or whose “wait for …” time-out has expired
      • resume
      • execute until a wait statement, then suspend

• Simulation finishes when there are no further scheduled transactions
Synthesis

• Translates register-transfer-level (RTL) design into gate-level netlist

• Restrictions on coding style for RTL model

• Tool dependent
Basic Design Methodology

1. Requirements
   - RTL Model
     - Synthesize
     - Gate-level Model
       - Place & Route
         - Timing Model
           - Simulate
           - ASIC or FPGA
             - Test Bench
               - Simulate
Questions...

A bird’s eye view on VHDL

Antonio R. Miele
Marco D. Santambrogio
Politecnico di Milano

YOU MEAN TO TELL ME

SPOONS DON'T ACTUALLY SOUND LIKE AIRPLANES?