Introduction

VHDL

• What is VHDL?

\[ \text{VHILC} \rightarrow \text{Very High Speed Integrated Circuit} \]

\text{Hardware}

\text{Description}

\text{Language} \quad \text{IEEE Standard 1076-1993}
History of VHDL

• Designed by IBM, Texas Instruments, and Intermetrics as part of the DoD funded VHSIC program
• Standardized by the IEEE in 1987: IEEE 1076-1987
• Enhanced version of the language defined in 1993: IEEE 1076-1993
• Additional standardized packages provide definitions of data types and expressions of timing data
  – IEEE 1164 (data types)
  – IEEE 1076.3 (numeric)
  – IEEE 1076.4 (timing)

Traditional vs. Hardware Description Languages

• Procedural programming languages provide the how or recipes
  – for computation
  – for data manipulation
  – for execution on a specific hardware model
• Hardware description languages describe a system
  – Systems can be described from many different points of view
    • Behavior: what does it do?
    • Structure: what is it composed of?
    • Functional properties: how do I interface to it?
    • Physical properties: how fast is it?
Usage

• Descriptions can be at different levels of abstraction
  – Switch level: model switching behavior of transistors
  – Register transfer level: model combinational and sequential logic components
  – Instruction set architecture level: functional behavior of a microprocessor

• Descriptions can be used for
  – Simulation
    • Verification, performance evaluation
  – Synthesis
    • First step in hardware design

Why do we Describe Systems?

• Design Specification
  – unambiguous definition of components and interfaces in a large design

• Design Simulation
  – verify system/subsystem/chip performance prior to design implementation

• Design Synthesis
  – automated generation of a hardware design
Digital System Design Flow

- Design flows operate at multiple levels of abstraction
- Need a uniform description to translate between levels
- Increasing costs of design and fabrication necessitate greater reliance on automation via CAD tools
  - $5M - $100M to design new chips
  - Increasing time to market pressures

A Synthesis Design Flow

- Automation of design refinement steps
- Feedback for accurate simulation
- Example targets: ASICs, FPGAs
The Role of Hardware Description Languages

- Design is structured around a hierarchy of representations
- HDLs can describe distinct aspects of a design at multiple levels of abstraction

Interoperability: models at multiple levels of abstraction
Technology independence: portable model
Design re-use and rapid prototyping
The Marketplace

- Time to market delays have a substantial impact on product revenue
- First 10%-20% of design cycle can determine 70%-80% of the cost
- Costs are rising rapidly with each new generation of technology
- Need standards and re-use → automation centered around HDL based tools such as VHDL


Alternatives

- The Verilog hardware description language
  - Finding increasing use in the commercial world
    - SystemVerilog gaining prominence
  - VHDL dominates the aerospace and defense worlds

- Programming language based design flows
  - SystemC
    - C++ with additional hardware-based language elements
  - C-based design flows
    - C + extensions as well as ANSI C based
  - Other
    - Java, MATLAB, and specialized languages
Role of VHDL

VHDL

- System description and documentation
- System simulation
- System synthesis
Modeling Digital Systems

Systems Hierarchy

Level of Abstraction

- Functions
- CPUs, memories
- ALUs, registers
- Switches

Increasing
- Fidelity
- # of events
Describing Systems

• From Webster’s Dictionary:
  – **System**: “An assemblage of objects united by some form of regular interaction or dependence”

• What aspects of a digital system do we want to describe?
  – **Interface**
  – **Function**: behavioral and structural

What Elements Should be in a Description?

• Descriptions should be at multiple levels of abstraction
  – The descriptive elements must be common to multiple levels of hierarchy

• The elements should enable meaningful and accurate simulation of hardware described using the elements
  – Elements should have attributes of time as well as function

• The elements should enable the generation of hardware elements that realize a correct physical implementation
  – Existence of a mapping from elements to VLSI devices
What Elements Should be in a Description?

- VHDL was conceived for the description of digital systems
  - From switches to networked systems

- Keep in mind the pragmatic issues of design re-use and portability of descriptions
  - Portability across technology generations
  - Portability across a range of cost/performance points

- Attributes of digital systems serve as the starting point
  - Language features designed to capture the key attributes

Attributes of Digital Systems

- Digital systems are about signals and their values
- Events, propagation delays, concurrency
  - Signal value changes at specific points in time
- Time ordered sequence of events produces a waveform
Attributes of Digital Systems: Timing

- Timing: computation of events takes place at specific points in time
- Need to "wait for" an event: in this case the clock
- Timing is an attribute of both synchronous and asynchronous systems

Example: Asynchronous communication
- No global clock
- Still need to wait for events on specific signals
Attributes of Digital Systems: Signal Values

- We associate logical values with the state of a signal

- Signal Values: IEEE 1164 Value System

<table>
<thead>
<tr>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Uninitialized</td>
</tr>
<tr>
<td>X</td>
<td>Forcing Unknown</td>
</tr>
<tr>
<td>0</td>
<td>Forcing 0</td>
</tr>
<tr>
<td>1</td>
<td>Forcing 1</td>
</tr>
<tr>
<td>Z</td>
<td>High Impedance</td>
</tr>
<tr>
<td>W</td>
<td>Weak Unknown</td>
</tr>
<tr>
<td>L</td>
<td>Weak 0</td>
</tr>
<tr>
<td>H</td>
<td>Weak 1</td>
</tr>
<tr>
<td>-</td>
<td>Don't Care</td>
</tr>
</tbody>
</table>

Attributes of Digital Systems: Multiple Drivers

- Shared Signals
  - multiple drivers

- How is the value of the signal determined?
  - arbitration protocols
  - wired logic
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 "mimicking" 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 vs. Synthesis

- Simulation and synthesis are complementary processes

Simulation of Digital Systems

- Digital systems are modeled as the generation of events – value transitions – on signals
- Discrete event simulations manage the generation and ordering of events
  - Correct sequencing of event processing
  - Correct sequencing of computations caused by events
Discrete Event Simulation: Example

Initial state: \( a = b = 1, \) sum = carry = \( U \)

Simulation Time | Event List Head
--- | ---
0ns | \( U \rightarrow 1 \) carry@5ns, \( U \rightarrow 0 \) sum@5ns
5ns | \( U \rightarrow 1 \) carry@5ns, \( U \rightarrow 0 \) sum@5ns, \( 1 \rightarrow 0 \) a@5ns
10ns | \( 1 \rightarrow 0 \) carry@10ns, \( 0 \rightarrow 1 \) sum@10ns, \( 0 \rightarrow 1 \) a@10ns, \( 1 \rightarrow 0 \) b@10ns
15ns | \( 1 \rightarrow 0 \) a@15ns

Discrete Event Simulation

- Management of simulation time: ordering of events
- Two step model of the progression of time
  - Evaluate all affected components at the current time: events on input signals
  - Schedule future events and move to the next time step: the next time at which events take place
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

Synthesis and Hardware Inference

- Both processes can produce very different results!
Summary

• VHDL is used to describe digital systems and hence has language constructs for key attributes
  – Events, propagation delays, and concurrency
  – Timing, and waveforms
  – Signal values and use of multiple drivers for a signal

• VHDL has an underlying discrete event simulation model
  – Model the generation of events on signals
  – Built in mechanisms for managing events and the progression of time
  – Designer simply focuses on writing accurate descriptions
Basic Language Concepts

Describing Design Entities

- Primary programming abstraction is a design entity
  - Register, logic block, chip, board, or system
- What aspects of a digital system do we want to describe?
  - Interface: how do we connect to it
  - Function: what does it do?
- VHDL 1993 vs. VHDL 1987
Describing the Interface: The Entity Construct

- The interface is a collection of ports
  - Ports are a new programming object: signal
  - Ports have a type, e.g., bit
  - Ports have a mode: in, out, inout (bidirectional)

The Signal Object Type

- VHDL supports four basic objects: variables, constants, signals and file types (1993)
- Variable and constant types
  - Follow traditional concepts
- The signal object type is motivated by digital system modeling
  - Distinct from variable types in the association of time with values
  - Implementation of a signal is a sequence of time-value pairs!
    - Referred to as the driver for the signal
Example Entity Descriptions

entity ALU32 is
  port (A, B: in bit_vector (31 downto 0);
        C: out bit_vector (31 downto 0);
        Op: in bit_vector (5 downto 0);
        N, Z: out bit);
end entity ALU32;

entity D_ff is
  port (D, Q, Clk, R, S: in bit;
        Q, Qbar : out bit);
end entity D_ff;

Describing Behavior: The Architecture Construct

- Description of events on output signals in terms of events on input signals: the signal assignment statement
- Specification of propagation delays
- Type bit is not powerful enough for realistic simulation: use the IEEE 1164 value system
Example Entity Descriptions: IEEE 1164

entity ALU32 is
port(A, B: in std_ulogic_vector (31 downto 0);
    C: out std_ulogic_vector (31 downto 0);
    Op: in std_ulogic_vector (5 downto 0);
    N, Z: out std_logic);
end entity ALU32;

entity D_ff is
port(D, Q, Clk, R, S: in std_ulogic;
    Q, Qbar : out std_ulogic);
end entity D_ff;

Describing Behavior: The Architecture Construct

library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port(a, b : in std_logic;
    sum, carry : out std_logic);
end entity half_adder;

architecture behavioral of half_adder is
begin
    sum <= (a xor b) after 5 ns;
carry <= (a and b) after 5 ns;
end architecture behavioral;

• Use of the IEEE 1164 value system requires inclusion of the library and package declaration statements
Libraries and Packages

- Libraries are logical units that are mapped to physical directories
- Packages are repositories for type definitions, procedures, and functions
  - User defined vs. system packages

```
package
  package declaration
  package body
    specification of the package contents
    code blocks
```

Configurations

- Separate the specification of the interface from that of the implementation
  - An entity may have multiple architectures
- Configurations associate an entity with an architecture
  - Binding rules: default and explicit
- Use configurations (more later!)
Design Units

- Primary design units
  - Entity
  - Configuration
  - Package Declaration
  - These are not dependent on other design units

- Secondary design units
  - Package body
  - Architecture

- Design units are created in design files

- Now you know the layout of a VHDL program!

Simple Signal Assignment

library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
port (in1, in2, c_in: in std_ulogic;
  sum, c_out: out std_ulogic);
end entity full_adder;

architecture dataflow of full_adder is
  signal s1, s2, s3 : std_ulogic;
  constant gate_delay: Time := 5 ns;
begin
  L1: s1 <= (in1 xor in2) after gate_delay;
  L2: s2 <= (c_in and s1) after gate_delay;
  L3: s3 <= (in1 and in2) after gate_delay;
  L4: sum <= (s1 xor c_in) after gate_delay;
  L5: c_out <= (s2 or s3) after gate_delay;
end architecture dataflow;

Declarations
Simple Signal Assignment Statement

- The *constant* programming object
  - Values cannot be changed
- Use of *signals* in the architecture
  - Internal signals connect components
- A statement is executed when an event takes place on a signal in the RHS of an expression
  - 1-1 correspondence between signal assignment statements and signals in the circuit
  - Order of statement execution follows propagation of events in the circuit
  - Textual order **does not** imply execution order

Implementation of Signals

- Driver or projected waveform
- Transaction
- s <= (In1 nand In2) after gate_delay
- *value expression* *time expression*
- waveform element
Implementation of Signals (cont.)

• In the absence of initialization, default values are determined by signal type

• Waveform elements describe time-value pairs

• *Transactions* are internal representations of signal value assignments
  – Events correspond to new signal values
  – A transaction may lead to the same signal value

---

Implementation of Signals (cont.)

• *Driver* is set of future signal values: current signal value is provided by the transaction at the head of the list

• We can specify multiple waveform elements in a single assignment statement
  – Specifying multiple future values for a signal

• Rules for maintaining the driver
  – Conflicting transactions
Example: Waveform Generation

- Multiple waveform elements can be specified in a single signal assignment statement
- Describe the signal transitions at future point in time
  - Each transition is specified as a waveform element

```vhdl
signal <= '0', '1' after 10 ns, '0' after 20 ns, '1' after 40 ns;
```

Resolved Signal Types

- At any point in time what is the value of the bus signal?
- We need to “resolve” the value
  - Take the value at the head of all drivers
  - Select one of the values according to a resolution function
- Predefined IEEE 1164 resolved types are std_logic and std_logic_vector
Conditional Signal Assignment

library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
  port ( In0, In1, In2, In3 : in std_logic_vector (7 downto 0);
  Sel: in std_logic_vector(1 downto 0);
  Z : out std_logic_vector (7 downto 0));
end entity mux4;

architecture behavioral of mux4 is
begin
  Z <= In0 after 5 ns when Sel = "00" else
       In1 after 5 ns when Sel = "01" else
       In2 after 5 ns when Sel = "10" else
       In3 after 5 ns when Sel = "11" else
       "00000000" after 5 ns;
end architecture behavioral;

• First true conditional expression determines the output value

Unaffected Signals

library IEEE;
use IEEE.std_logic_1164.all;

entity pr_encoder is
  port (S0, S1, S2, S3: in std_logic;
  Z : out std_logic_vector (1 downto 0));
end entity pr_encoder;

architecture behavioral of pr_encoder is
begin
  Z <= "00" after 5 ns when S0 = '1' else
       "01" after 5 ns when S1 = '1' else
       unaffected when S2 = '1' else
       "11" after 5 ns when S3 = '1' else
       "00" after 5 ns;
end architecture behavioral;

• Value of the signal is not changed
• VHDL 1993 only!
Selected Signal Assignment Statement

library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
  port ( In0, In1, In2, In3 : in std_logic_vector (7 downto 0);
         Sel: in std_logic_vector(1 downto 0);
         Z : out std_logic_vector (7 downto 0));
end entity mux4;
architecture behavioral of mux4 is
begin
  with Sel select
  Z <= (In0 after 5 ns) when "00",
       (In1 after 5 ns) when "01",
       (In2 after 5 ns) when "10",
       (In3 after 5 ns) when "11"
         when others;
end architecture behavioral;

• The "when others" clause can be used to ensure that all options are covered
• The "unaffected" clause may also be used here

A VHDL Model Template

library library-name-1, library-name-2;
use library-name-1.package-name.all;
use library-name-2.package-name.all;

entity entity_name is
  port( input signals : in type;
        output signals : out type);
end entity entity_name;

architecture arch_name of entity_name is
begin
  -- specify value of each signal as a function of other signals
  output-signal-1 <= simple, conditional, or selected CSA;
  output-signal-2 <= simple, conditional, or selected CSA;
end architecture arch_name;

• Declare external libraries and visible components
• Define the interface
• Declare signals used to connect components
• Definition of how & when internal signal values are computed
• Definition of how & when external signal values are computed
Delay Models in VHDL

- Inertial delay
  - Default delay model
  - Suitable for modeling delays through devices such as gates
- Transport Delay
  - Model delays through devices with very small inertia, e.g., wires
  - All input events are propagated to output signals
- Delta delay
  - What about models where no propagation delays are specified?
  - Infinitesimally small delay is automatically inserted by the simulator to preserve correct ordering of events

Inertial Delays: Example

- \textit{signal} <= \textbf{reject} \textit{time-expression inertial value-expression after} \textit{time-expression};
- Most general form of a waveform element
- VHDL 1993 enables specification of pulse rejection width
Transport Delays: Example

architecture transport_delay of half_adder is
signal s1, s2: std_logic := '0';
begin
s1 <= (a xor b) after 2 ns;
s2 <= (a and b) after 2 ns;
sum <= transport s1 after 4 ns;
carry <= transport s2 after 4 ns;
end architecture transport_delay;

Delta Delays: Example

library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port (In1, In2: in std_logic;
z: out std_logic);
end entity combinational;
architecture behavior of combinational signal s1, s2, s3, s4: std_logic := '0';
begin
s1 <= not In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);
end architecture behavior;
**Delay Models: Summary**

- Delay models
  - Inertial
    - For devices with inertia such as gates
    - VHDL 1993 supports pulse rejection widths
  - Transport
    - Ensures propagation of all events
    - Typically used to model elements such as wires
  - Delta
    - Automatically inserted to ensure functional correctness of code blocks that do not specify timing
    - Enforces the data dependencies specified in the code
Summary

• Primary unit of abstraction is a design entity

• Design units include
  – Primary design units
    • entity, configuration, package declaration
  – Secondary design units
    • architecture, package body

• Concurrent signal assignment statements
  – Simple, selected, conditional
  – Can be coalesced to form models of combinational circuits
Modeling Complex Behavior

Outline

• Abstraction and the Process Statement
  – Concurrent processes and CSAs

• Process event behavior and signals vs. variables

• Timing behavior of processes

• Attributes

• Putting it together → modeling state machines
Raising the Level of Abstraction

- Concurrent signal assignment statements can easily capture the gate level behavior of digital systems
- Higher level digital components have more complex behaviors
  - Input/output behavior not easily captured by concurrent signal assignment statements
  - Models utilize state information
  - Incorporate data structures
- We need more powerful constructs

Extending the Event Model

- Combinational logic input/output semantics
  - Events on inputs causes re-computation
  - Re-computation may lead to events on outputs
- Computation of the value and time of output events can be a complex process
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
port (In0, In1, In2, In3: in std_logic_vector (7 downto 0);
      Sel: in std_logic_vector(1 downto 0);
      Z : out std_logic_vector (7 downto 0));
end entity mux4;

architecture behavioral-3 of mux4 is
begin
process (Sel, In0, In1, In2, In3) is
    variable Zout: std_logic;
begin
    if (Sel = "00") then
        Zout := In0;
    elsif (Sel = "01") then
        Zout := In1;
    elsif (Sel = "10") then
        Zout := In2;
    else
        Zout := In3;
    end if;
    Z <= Zout;
end process;

end architecture;

The Process Statement

- Statements in a process are executed sequentially
- A process body is structured much like conventional C function
  - Declaration and use of variables
  - if-then, if-then-else, case, for and while constructs
  - A process can contain signal assignment statements
- A process executes concurrently with other concurrent signal assignment statements
- A process takes 0 seconds of simulated time to execute and may schedule events in the future
- We can think of a process as a complex signal assignment statement!
Concurrent Processes: Full Adder

- Each of the components of the full adder can be modeled using a process
- Processes execute concurrently
  - In this sense they behave exactly like concurrent signal assignment statements
- Processes communicate via signals

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

entity full_adder is
  port (In1, c_in, In2: in std_logic;
        sum, c_out: out std_logic);
end entity full_adder;

architecture behavioral of full_adder is
  signal s1, s2, s3: std_logic;
  constant delay: Time:= 5 ns;
begin
  HA1: process (In1, In2) is
    begin
      s1 <= (In1 xor In2) after delay;
      s3 <= (In1 and In2) after delay;
    end process HA1;

  HA2: process(s1,c_in) is
    begin
      sum <= (s1 xor c_in) after delay;
      s2 <= (s1 and c_in) after delay;
    end process HA2;

  OR1: process (s2, s3) -- process describing the two-input OR gate
    begin
      c_out <= (s2 or s3) after delay;
    end process OR1;

end architecture behavioral;
```
Concurrent Processes: Half Adder

library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
  port (a, b : in std_logic;
        sum, carry : out std_logic);
end entity half_adder;

architecture behavior of half_adder is
begin
  sum_proc: process (a, b)
  is
  begin
    if (a = b) then
      sum <= '0' after 5 ns;
    else
      sum <= (a or b) after 5 ns;
    end if;
  end process;

  carry_proc: process (a, b)
  begin
    case a is
      when '0' =>
        carry <= a after 5 ns;
      when '1' =>
        carry <= b after 5 ns;
      when others =>
        carry <= 'X' after 5 ns;
    end case;
  end process;
end architecture behavior;

Processes + CSAs

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity memory is
  port (address, write_data: in std_logic_vector (7 downto 0);
        MemWrite, MemRead, clk, reset: in std_logic;
        read_data: out std_logic_vector (7 downto 0));
end entity memory;

architecture behavioral of memory is
  signal dmem0, dmem1, dmem2, dmem3: std_logic_vector (7 downto 0);
begin
  mem_proc: process (clk)
  begin
    -- process body
  end process mem_proc;
end architecture behavioral;
Process + CSAs: The Write Process

```vhdl
mem_proc: process (clk) is 
begin 
if (rising_edge(clk)) then -- wait until next clock edge 
if reset = '1' then -- initialize values on reset 
dmem0 <= x"00"; -- memory locations are initialized to 
dmem1 <= x"11";-- some random values 
dmem2 <= x"22";
dmem3 <= x"33";
elif MemWrite = '1' then -- if not reset then check for memory write 
case address (1 downto 0) is 
when "00" => dmem0 <= write_data; 
when "01" => dmem1 <= write_data; 
when "10" => dmem2 <= write_data; 
when "11" => dmem3 <= write_data; 
when others => dmem0 <= x"ff";
end case;
end if;
end if;
end process mem_proc;
```

Process + CSAs: The Read Statement

- Memory read is implemented with a conditional signal assignment

```vhdl
read_data <= dmem0 when address (1 downto 0) = "00" and MemRead = '1' else 
dmem1 when address (1 downto 0) = "01" and MemRead = '1' else 
dmem2 when address (1 downto 0) = "10" and MemRead = '1' else 
dmem3 when address (1 downto 0) = "11" and MemRead = '1' else 
x"00";
```

- A process can be viewed as single concurrent signal assignment statement
  - The external behavior is the same as a CSA
  - Processes describe more complex event generation behavior
- Processes execute concurrently in simulated time with other CSAs
Example: A Simple Multiplier

architecture behavioral of mult32 is
constant module_delay: Time := 10 ns;
beginn
mult_process: process(multiplicand,multiplier) is
  variable product_register : std_logic_vector (63 downto 0) := X"00000000000000000000000000000000";
  variable multiplicand_register : std_logic_vector (31 downto 0):= X"00000000";
begin
  multiplicand_register := multiplicand;
  product_register(63 downto 0) := X"00000000" & multiplier;
  for index in 1 to 32 loop
    if product_register(0) = '1' then
      product_register(63 downto 32) := product_register (63 downto 32) +
      multiplicand_register(31 downto 0);
    end if;
    -- perform a right shift with zero fill
    product_register (63 downto 0) := '0' & product_register (63 downto 1);
  end loop;
  -- write result to output port
  product <= product_register after module_delay;
end process mult_process;

• for loop index
  – Implicit declaration via “use”
    • Scope is local to the loop
  – Cannot be used elsewhere in model

• while loop
  – Boolean expression for termination

  while j < 32 loop
    ...
    ...
    j := j+1;
  end loop;
Outline

- Abstraction and the Process Statement
- Process event behavior and signals vs. variables
- Timing behavior of processes
- Attributes
- Putting it together \(\rightarrow\) modeling state machines

Process Behavior

- All processes are executed once at start-up
- Thereafter dependencies between signal values and events on these signals determine process initiation
- One can view processes as components with an interface/function
- Note that signals behave differently from variables!

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

entity sig_var is
  port (x, y, z: in std_logic;
        res1, res2: out std_logic);
end entity sig_var;

architecture behavior of sig_var is
  signal sig_s1, sig_s2: std_logic;
begin
  proc1: process (x, y, z) is -- Process 1
    variable var_s1, var_s2: std_logic;
  begin
    L1: var_s1 := x and y;
    L2: var_s2 := var_s1 xor z;
    L3: res1 <= var_s1 nand var_s2;
  end process;

  proc2: process (x, y, z) -- Process 2
    begin
    L1: sig_s1 <= x and y;
    L2: sig_s2 <= sig_s1 xor z;
    L3: res2 <= sig_s1 nand sig_s2;
  end process;

end architecture behavior;
```
Variables vs. Signals: Example

Variables vs. Signals: Example

proc1: process (x, y, z) is -- Process 1
variable var_s1, var_s2: std_logic;
begin
  L1: var_s1 := x and y;
  L2: var_s2 := var_s1 xor z;
  L3: res1 <= var_s1 nand var_s2;
end process;

proc2: process (x, y, z) -- Process 2
begin
  L1: sig_s1 <= x and y;
  L2: sig_s2 <= sig_s1 xor z;
  L3: res2 <= sig_s1 nand sig_s2;
end process;

• Distinction between the use of variables vs. signals
  – Computing values vs. computing time-value pairs
  – Remember event ordering and delta delays!

Variables vs. Signals: Example

• Writing processes
  – Use signals to represent corresponding hardware entities
  – Use variables when computing (future) values of signals

This transition is determined by process initiation
### Simulation and Signals

**Execute Process**
- Process is initiated
- Compute new signal values using current signal values
  - Use the value of the signal at process initiation
  - Ignore textual dependencies between signals

**Update Process State**
- Process state (signal values) are updated
- Process now waits for an event on its sensitivity list
- When initiated, process execution references these new signal values

**Using Signals in a Process**
- Entity signals are visible in a process
- Processes can encapsulate variable and signal assignment statements
- What is the effect on the model behavior between dataflow and process models?
- Actual waveforms will depend on how initialization is handled/performed
Using Signals in a Process

library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port (In1, In2: in std_logic;
   z : out std_logic);
end entity combinational;
signal s1, s2, s3, s4: std_logic:= '0';
begin
   s1 <= not In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);
end architecture behavior;

library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port (In1, In2: in std_logic;
   z : out std_logic);
end entity combinational;
signal s1, s2, s3, s4: std_logic:= '0';
begin
   sig_in_proc: process (In1, In2) is
   begin
      s1 <= not In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);
   end process sig_in_proc;
end architecture behavior;

Using Signals in a Process (cont.)

Using concurrent signal assignment statements

Using signal assignment statements within a process
Outline

• Abstraction and the Process Statement
  – Concurrent processes and CSAs

• Process event behavior and signals vs. variables

• Timing behavior of processes

• Attributes

• Putting it together → modeling state machines

The Wait Statement

library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
  port (D, Clk : in std_logic;
        Q, Qbar : out std_logic);
end entity dff;

architecture behavioral of dff is
begin
  output process is
    begin
      wait until (Clk'event and Clk = '1'); -- wait for rising edge
      Q <= D after 5 ns;
      Qbar <= not D after 5 ns;
    end process output;
  end architecture behavioral;

• The wait statements can describe synchronous or asynchronous timing operations
The Wait Statement: Waveform Generation

library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out std_logic);
end entity two_phase;
architecture behavioral of two_phase is
begin
  rproc: reset <= '1', '0' after 10 ns;
clock_process: process is
begin
  phi1 <= '1', '0' after 10 ns;
  phi2 <= '0', '1' after 12 ns, '0' after 18 ns;
  wait for 20 ns;
end process clock_process;
end architecture behavioral;

• Note the “perpetual” behavior of processes

Wait Statement: Asynchronous Inputs

library IEEE; use IEEE.std_logic_1164.all;
entity asynch_dff is
port (R, S, D, Clk: in std_logic;
  Q, Qbar: out std_logic);
end entity asynch_dff;
architecture behavioral of asynch_dff is
begin
  output: process (R, S, Clk) is
  begin
    if (R = '0') then
      Q <= '0' after 5 ns;
      Qbar <= '1' after 5 ns;
    elsif S = '0' then
      Q <= '1' after 5 ns;
      Qbar <= '0' after 5 ns;
    elsif (rising_edge(Clk)) then
      Q <= D after 5 ns;
      Qbar <= (not D) after 5 ns;
    end if;
  end process output;
end architecture behavioral;

execute on event on any signal
implied ordering provides asynchronous set reset

The Wait Statement

- A process can have multiple wait statements

- A process cannot have both a wait statement and a sensitivity list (it should have one or the other): why?

- wait statements provide explicit control over suspension and resumption of processes
  - Representation of both synchronous and asynchronous events in a digital systems

Outline

- Abstraction and the Process Statement
  - Concurrent processes and CSAs

- Process event behavior and signals vs. variables

- Timing behavior of processes

- Attributes

- Putting it together \(\rightarrow\) modeling state machines
Attributes

• Data can be obtained about VHDL objects such as types, arrays and signals.

  object' attribute

• Example: consider the implementation of a signal

  ![Diagram of a signal implementation]

• What types of information about this signal are useful?
  – Occurrence of an event
  – Elapsed time since last event
  – Previous value, i.e., prior to the last event

Classes of Attributes

• Value attributes
  – returns a constant value

• Function attributes
  – invokes a function that returns a value

• Signal attributes
  – creates a new signal

• Type Attributes
  – Supports queries about the type of VHDL objects

• Range attributes
  – returns a range
Value Attributes

• Return a constant value
  – type statetype is (state0, state1, state2, state3);
    • state_type’left = state0
    • state_type’right = state3

• Examples

<table>
<thead>
<tr>
<th>Value attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_name’left</td>
<td>returns the left most value of type_name in its defined range</td>
</tr>
<tr>
<td>type_name’right</td>
<td>returns the right most value of type_name in its defined range</td>
</tr>
<tr>
<td>type_name’high</td>
<td>returns the highest value of type_name in its range</td>
</tr>
<tr>
<td>type_name’low</td>
<td>returns the lowest value of type_name in its range</td>
</tr>
<tr>
<td>array_name’length</td>
<td>returns the number of elements in the array array_name</td>
</tr>
</tbody>
</table>

(31)

Example

```vhdl
clk_process: process
begin
  wait until (clk’event and clk = ‘1’);
  if reset = ‘1’ then
    state <= statetype’left;
  else
    state <= next_state;
  end if;
end process clk_process;
```

• The signal state is an enumerated type
  – type statetype is (state0, state1, state3, state4);

• signal state:statetype:= statetype’left;

(32)
Function Attributes

- Use of attributes invokes a function call which returns a value
  - if (Clk'event and Clk = '1')
- function call
- Examples: function signal attributes

<table>
<thead>
<tr>
<th>Function attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal_name'event</td>
<td>Return a Boolean value signifying a change in value on this signal</td>
</tr>
<tr>
<td>signal_name'active</td>
<td>Return a Boolean value signifying an assignment made to this signal. This assignment may not be a new value.</td>
</tr>
<tr>
<td>signal_name'last_event</td>
<td>Return the time since the last event on this signal</td>
</tr>
<tr>
<td>signal_name'last_active</td>
<td>Return the time since the signal was last active</td>
</tr>
<tr>
<td>signal_name'last_value</td>
<td>Return the previous value of this signal</td>
</tr>
</tbody>
</table>

Function Attributes (cont.)

- Function array attributes

<table>
<thead>
<tr>
<th>Function attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_name'left</td>
<td>returns the left bound of the index range</td>
</tr>
<tr>
<td>array_name'right</td>
<td>returns the right bound of the index range</td>
</tr>
<tr>
<td>array_name'high</td>
<td>returns the upper bound of the index range</td>
</tr>
<tr>
<td>array_name'low</td>
<td>returns the lower bound of the index range</td>
</tr>
</tbody>
</table>

- type mem_array is array(0 to 7) of bit_vector(31 downto 0)
  - mem_array'left = 0
  - mem_array'right = 7
  - mem_array'length = 8 (value kind attribute)
Range Attributes

• Returns the index range of a constrained array

    for i in value_array'range loop
    ...
    my_var := value_array(i);
    ...
    end loop;

• Makes it easy to write loops

Signal Attributes

• Creates a new “implicit” signal

<table>
<thead>
<tr>
<th>Signal attribute</th>
<th>Implicit Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal_name'delayed(T)</td>
<td>Signal delayed by T units of time</td>
</tr>
<tr>
<td>signal_name'transaction</td>
<td>Signal whose value toggles when signal_name is active</td>
</tr>
<tr>
<td>signal_name'quiet(T)</td>
<td>True when signal_name has been quiet for T units of time</td>
</tr>
<tr>
<td>signal_name'stable(T)</td>
<td>True when event has not occurred on signal_name for T units of time</td>
</tr>
</tbody>
</table>

• Internal signals are useful modeling tools
**Signal Attributes: Example**

```vhdl
architecture behavioral of attributes is
begin
    outdelayed <= data'delayed(5 ns);
    outtransaction <= data'transaction;
end attributes;
```

These are real (in simulation) signals and can be used elsewhere in the model.

---

**Outline**

- Abstraction and the Process Statement
  - Concurrent processes and CSAs

- Process event behavior and signals vs. variables

- Timing behavior of processes

- Attributes

- Putting it together → modeling state machines
State Machines

- Basic components
  - Combinational component: output function and next state function
  - Sequential component
- Natural process-based implementation

Example: State Machine

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity state_machine is
  port(reset, clk, x : in std_logic;
       z : out std_logic);
end entity state_machine;
architecture behavioral of state_machine is
  type statetype is (state0, state1);
  signal state, next_state : statetype := state0;
  begin
    comb_process: process (state, x) is
      begin
        -- process description here
    end process comb_process;
    clk_process: process is
      begin
        -- process description here
    end process clk_process;
  end architectural behavioral;
end behavioral;
```
Example: Output and Next State Functions

comb_process: process (state, x) is
begin
  case state is -- depending upon the current state
    when state0 => -- set output signals and next state
      if x = '0' then
        next_state <= state1;
        z <= '1';
      else
        next_state <= state0;
        z <= '0';
      end if;
    when state1 =>
      if x = '1' then
        next_state <= state0;
        z <= '0';
      else
        next_state <= state1;
        z <= '1';
      end if;
  end case;
end process comb_process;

• Combination of the next state and output functions

Example: Clock Process

clk_process: process is
begin
  wait until (clk'event and clk = '1'); -- wait until the rising edge
  if reset = '1' then -- check for reset and initialize state
    state <= statetype'left;
  else
    state <= next_state;
  end if;
end process clk_process;
end behavioral;

• Use of asynchronous reset to initialize into a known state
Summary

- Processes
  - variables and sequential statements
  - if-then, if-then-else, case, while, for
  - concurrent processes
  - sensitivity list

- The Wait statement
  - wait until, wait for, wait on

- Attributes
- Modeling State machines
  - wait on ReceiveData\^transaction
    - if ReceiveData\^delayed = ReceiveData then
      ..
Modeling Structure

Elements of Structural Models

- Structural models describe a digital system as an interconnection of components
- Descriptions of the behavior of the components must be independently available as structural or behavioral models
  - An entity/architecture for each component must be available
Modeling Structure

- Define the components used in the design
- Describe the interconnection of these components

architecture structural of full_adder is
component half_adder is -- the declaration
port (a, b : in std_logic; -- of components you will use
sum, carry : out std_logic);
end component half_adder;
component or_2 is
port(a, b : in std_logic;
c : out std_logic);
end component or_2;
signal s1, s2, s3 : std_logic;
begins
H1: half_adder port map (a => In1, b => In2, sum => s1, carry => s3);
H2:half_adder port map (a => s1, b => c_in, sum => sum, carry => s2);
O1: or_2 port map (a => s2, b => s3, c => c_out);
end architecture structural;

- Entity/architecture for half_adder and or_2 must exist
Example: State Machine

- Structural models can be easily generated from schematics
- Name conflicts in the association lists?
- The “open” attribute

Hierarchy and Abstraction

- Structural descriptions can be nested
- The half adder may itself be a structural model
Hierarchy and Abstraction

• Nested structural descriptions to produce hierarchical models
• The hierarchy is flattened prior to simulation
• Behavioral models of components at the bottom level must exist

Hierarchy and Abstraction

• Use of IP cores and vendor libraries
• Simulations can be at varying levels of abstraction for individual components
Generics

library IEEE;
use IEEE.std_logic_1164.all;

entity xor2 is
  generic (gate_delay : Time:= 2 ns);
  port(In1, In2 : in std_logic;
       z : out std_logic);
end entity xor2;

architecture behavioral of xor2 is
begin
  z <= (In1 xor In2) after gate_delay;
end architecture behavioral;

• Enables the construction of parameterized models

Generics in Hierarchical Models

architecture generic_delay of half_adder is
component xor2
  generic (gate_delay: Time);
  port (a, b : in std_logic;
        c : out std_logic);
end component;
component and2
  generic (gate_delay: Time);
  port (a, b : in std_logic;
        c : out std_logic);
end component;
begin
  EX1: xor2 generic map (gate_delay => 6 ns)
    port map(a => a, b => b, c => sum);
  A1: and2 generic map (gate_delay => 3 ns)
    port map(a=> a, b=> b, c=> carry);
end architecture generic_delay;

• Parameter values are passed through the hierarchy
Example: Full Adder

Example: Full Adder

H1: half_adder

--- top level

H2: half_adder

--- bottom level

full_adder.vhd

half_adder.vhd

or_2.vhd

and2.vhd

xor2.vhd

entity and2 is
generic (gate_delay : Time := 2 ns);

EX1: xor2 generic map (gate_delay => 6 ns)

port map(a => In1, b => In2, sum => s1, carry=>s2);

H2: half_adder ...

EX1: xor2 generic map (gate_delay => 6 ns)

A1: and2 generic map (gate_delay => gate_delay)

entity xor2 is
generic (gate_delay : Time := 2 ns);

A1: and2 generic map (gate_delay => gate_delay)

...
Precedence of Generic Declarations

architecture generic_delay2 of half_adder is
component xor2
generic (gate_delay: Time);
port(a,b : in std_logic;
c : out std_logic);
end component;

component and2
generic (gate_delay: Time:= 6 ns);
port (a, b : in std_logic;
c : out std_logic);
end component;

begin
EX1: xor2 generic map (gate_delay => gate_delay)
port map(a => a, b => b, c => sum);
A1: and2 generic map (gate_delay => 4 ns)
port map(a=> a, b=> b, c=> carry);
end generic_delay2;

• Generic map takes precedence over the component declaration

Generics: Properties

• Generics are constant objects and can only be read
• The values of generics must be known at compile time
• They are a part of the interface specification but do not have a physical interpretation
• Use of generics is not limited to “delay like” parameters and are in fact a very powerful structuring mechanism
Example: N-Input Gate

```vhdl
entity generic_or is
generic (n: positive := 2);
port (in1: in std_logic_vector ((n-1) downto 0);
z: out std_logic);
end entity generic_or;
architecture behavioral of generic_or is
begin
process (in1) is
variable sum: std_logic := '0';
begin
sum := '0'; -- on an input signal transition sum must be reset
for i in 0 to (n-1) loop
sum := sum or in1(i);
end loop;
z <= sum;
end process;
end architecture;
```

- Map the generics to create different size OR gates

Example: Using the Generic N-Input OR Gate

```vhdl
architecture structural of full_adder is
component generic_or
generic (n: positive);
port (in1: in std_logic_vector ((n-1) downto 0);
z: out std_logic);
end component;
...
begin
H1: half_adder port map (a => In1, b => In2, sum => s1, carry => s3);
H2: half_adder port map (a => s1, b => c_in, sum => sum, carry => s2);
O1: generic_or generic map (n => 2)
port map (a => s2, b => s3, c => e_out);
end structural;
```

- Full adder model can be modified to use the generic OR gate model via the `generic map (n)` construct
- Analogy with macros
Example: N-bit Register

entity generic_reg is
  generic (n: positive := 2);
  port (clk, reset, enable : in std_logic;
        d : in std_logic_vector (n-1 downto 0);
        q : out std_logic_vector (n-1 downto 0));
end entity generic_reg;
architecture behavioral of generic_reg is
begin
  reg_process: process (clk, reset)
  begin
    if reset = '1' then
      q <= (others => '0');
    elsif rising_edge(clk) then
      if enable = '1' then
        q <= d;
      end if;
    end if;
  end process reg_process;
end architecture behavioral;

• This model is used in the same manner as the generic OR gate

Component Instantiation and Synthesis

• Design methodology for inclusion of highly optimized components or “cores”
  – Optimized in the sense of placed and routed
  – Intellectual property cores for sale
  – Check out http://www.xilinx.com/ipcenter/index.htm

• Core generators for static generation of cores
  – Generation of VHDL/Verilog models of placed and routed designs
  – Component instantiation for using cores in a design

• Access to special components/circuitry within the target chip
**Example: Using Global Resources**

library IEEE;
use IEEE.std_logic_1164.all;

entity my_clocks is
  port (phi1, phi2: out std_logic);
end entity my_clocks;

architecture behavioral of my_clocks is
  component OSC4 is -- on chip oscillator
    port (F8M : out std_logic; -- 8 MHz clock
          F500k : out std_logic; -- 500Khz clock
          F15 : out std_logic; -- 15 Hz clock)
  end component OSC4;

  component BUFG is -- global buffer connection to low skew lines
    port (I : in std_logic;
          O : out std_logic)
  end component BUFG;

  signal local_F15, local_phi2 : std_logic; -- local signals
  begin
  O1: osc4 port map(F15 => local_F15); -- instantiate the oscillator
  B1: bufg port map(I => local_F15, O => phi1); -- instantiate the two global buffers
  B2: bufg port map(I => local_phi2, O => phi2);
  local_phi2 <= not local_F15; -- phi2 is the complement of phi1
end architecture behavioral;

- Component abstractions for special support within the chip
  - For global signals
  - For shared system level components

---

**Implementing Global Set/Reset in VHDL**

library IEEE;
use IEEE.std_logic_1164.all;

entity sig_var is
  port ( clk, reset, x, y, z: in std_logic;
        w : out std_logic);
end entity sig_var;

architecture behavior of sig_var is
  component STARTUP is
    port (GSR : in std_logic);
  end component STARTUP;

  signal s1, s2 : std_logic;
  begin
  U1: STARTUP port map(GSR => reset);
  process
  begin
  wait until (rising_edge(clk));
  if (reset = '1') then
    w <= '0';
    s1 <= '0';
    s2 <= '0';
  else
    L1: s1 <= x xor y;
    L2: s2 <= s1 or z;
    L3: w <= s1 nor s2;
  end if;
  end process;
end architecture behavior;

- GSR inferencing optimization
  - Connect your reset signal to this global net
- Note
  - improves “routability” of designs
Core Generators

- Xilinx Logic Core utility
- Parameterized modules
  - User controlled generation of VHDL modules
  - Instantiation within a design
  - Simulation and synthesis
- Third party view of the world of hardware design
  - Analogy with software and compilers
  - What is software vs. hardware anymore?

The Generate Statement

- What if we need to instantiate a large number of components in a regular pattern?
  - Need conciseness of description
  - Iteration construct for instantiating components!
- The generate statement
  - A parameterized approach to describing the regular interconnection of components

```vhdl
a: for i in 1 to 6 generate
  a1: one_bit generic map (gate_delay)
  port map(in1=>in1(i), in2=> in2(i), cin=>carry_vector(i-1),
  result=>result(i), cout=>carry_vector(i),opcode=>opcode);
end generate;
```
The Generate Statement: Example

• Instantiating an register

```vhdl
entity dregister is
  port ( d : in std_logic_vector(7 downto 0);
    q : out std_logic_vector(7 downto 0);
    clk : in std_logic);
end entity dregister;

architecture behavioral of dregister is
begin
  d: for i in dreg'range generate
    reg: dff port map ( (d=>d(i), q=>q(i), clk=>clk);
  end generate
end architecture dregister;
```

• Instantiating interconnected components
  – Declare local signals used for the interconnect

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

entity multi_bit_generate is
generic (gate_delay: time := 1 ns;
  width: natural :=8); -- the default is a 8-bit ALU
port ( in1 : in std_logic_vector(width-1 downto 0);
  in2 : in std_logic_vector(width-1 downto 0);
  result : out std_logic_vector(width-1 downto 0);
  opcode : in std_logic_vector(1 downto 0);
  cin : in std_logic;
  cout : out std_logic);
end entity multi_bit_generate;

architecture behavioral of multi_bit_generate is
begin
  signal carry_vector: std_logic_vector(width-2 downto 0);
  -- the set of signals for the ripple carry

  a0: one_bit generic map (gate_delay) -- instantiate ALU
    for bit position 0
    port map (in1=>in1(0), in2=>in2(0), result=>result(0),
      cin=>cin, opcode=>opcode, cout=>carry_vector(0));

  a2 to 6: for i in 1 to width-2 generate -- generate
    instantiations for bit positions 2-6
    a1: one_bit generic map (gate_delay)
      port map (in1=>in1(i), in2=>in2(i), cin=>carry_vector(i-1),
        result=>result(i), cout=>carry_vector(i), opcode=>opcode);
    end generate;

  a7: one_bit generic map (gate_delay) -- instantiate ALU
    for bit position 7
    port map (in1=>in1(width-1), in2=>in2(width-1), result=>result(width-1),
      cin=>carry_vector(width-2),
      opcode=>opcode, cout=>cout);
end architecture behavioral;
```

Using the Generate Statement

- Identify components with regular interconnect
- Declare local arrays of signals for the regular interconnections
- Write the generate statement
  - Analogy with loops and multidimensional arrays
  - Beware of unconnected signals!
- Instantiate remaining components of the design

Configurations

- A design entity can have multiple alternative architectures
- A configuration specifies the architecture that is to be used to implement a design entity
Component Binding

- We are concerned with configuring the architecture and not the entity
- Enhances sharing of designs: simply change the configuration

Default Binding Rules

- Search for entity with the same component name
- If multiple such entities exist, bind the last compiled architecture for that entity
- How do we get more control over binding?
• We can specify any binding where ports and arguments match

• Short form where applicable
  
  for all: half_adder use entity WORK.half_adder (behavioral);

• Not constrained by the name space

• Delayed binding when a specification is not present
  
  – Will be available at a later step
  
  – Analogous to unresolved symbol references during compilation of traditional programs
Configuration Declaration

```vhdl
configuration Config_A of full_adder is -- name the configuration
  -- for the entity
  for structural -- name of the architecture being configured
  for H1: half_adder use entity WORK.half_adder (behavioral);
  end for;
  --
  for H2: half_adder use entity WORK.half_adder (structural);
  end for;
  --
  for O1: or_2 use entity POWER.lpo2 (behavioral)
    generic map (gate_delay => gate_delay)
    port map (I1 => a, I2 => b, Z => c);
  end for;
  --
end Config_A;
```

• Written as a separate design unit
• Can be written to span a design hierarchy
• Use of the “for all” clause

Summary

• Structural models
  – Syntactic description of a schematic
• Hierarchy and abstraction
  – Use of IP cores
  – Mixing varying levels of detail across components
• Generics
  – Construct parameterized models
  – Use in configuring the hardware
• Configurations
  – Configuration specification
  – Configuration declaration
Subprograms, Packages, and Libraries

Essentials of Functions

function rising_edge (signal clock: std_logic) return boolean is
--
--declarative region: declare variables local to the function
--
begin
-- body
--
return (expression)
end rising_edge;

- Formal parameters and mode
  - Default mode is of type in
- Functions cannot modify parameters
  - Pure functions vs. impure functions
    • Latter occur because of visibility into signals that are not parameters
- Function variables initialized on each call
Essentials of Functions (cont.)

- Types of formals and actuals must match except for formals which are constants (default)
  - Formals which are constant match actuals which are variable, constant or signal
- Wait statements are not permitted in a function!
  - And therefore not in any procedure called by a functions

Placement of Functions

- Place function code in the declarative region of the architecture or process
Function: Example

architecture behavioral of dff is
function rising_edge (signal clock : std_logic) return boolean is
variable edge : boolean := FALSE;
begnin
dge := (clock = '1' and clock'event);
return (edge);
end rising_edge;

begin
output: process
begin
wait until (rising_edge(Clk));
Q <= D after 5 ns;
Qbar <= not D after 5 ns;
end process output;
end architecture behavioral;

Function: Example

function to_bitvector (svalue : std_logic_vector) return bit_vector is
variable outvalue : bit_vector (svalue'length-1 downto 0);
begnin
for i in svalue'range loop -- scan all elements of the array
case svalue (i) is
when '0' => outvalue (i) := '0';
when '1' => outvalue (i) := '1';
when others => outvalue (i) := '0';
end case;
end loop;
return outvalue;
end to_bitvector

• A common use of functions: type conversion
• Use of attributes for flexible function definitions
  – Data size is determined at the time of the call
• Browse the vendor supplied packages for many examples
Implementation of Signals

- The basic structure of a signal assignment statement
  - `signal <= (value expression after time expression)`
- RHS is referred to as a **waveform element**
- Every signal has associated with it a **driver**

- Holds the current and future values of the signal - a projected waveform
- Signal assignment statements modify the driver of a signal
- Value of a signal is the value at the head of the driver

Shared Signals

- How do we model the state of a wire?
- Rules for determining the signal value is captured in the **resolution function**
Resolved Signals

- Resolution function is invoked whenever an event occurs on this signal
- Resolution must be an associative operation

Resolution Function Behavior

- Physical operation
  - If any of the control signals activate the switch, the output signal is pulled low
- VHDL model
  - If any of the drivers attempt to drive the signal low (value at the head of the driver), the resolution function returns a value of 0
  - Resolution function is invoked when any driver attempts to drive the output signal
Resolved Types: std_logic

```vhdl
type std_ulogic is (
    'U', -- Uninitialized
    'X', -- Forcing Unknown
    '0', -- Forcing 0
    '1', -- Forcing 1
    'Z', -- High Impedance
    'W', -- Weak Unknown
    'L', -- Weak 0
    'H', -- Weak 1
    '-' -- Don't care
);

function resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;
```

Type only supports only single drivers

New subtype supports multiple drivers

Resolution Function: std_logic & resolved()

```
<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>X</th>
<th>0</th>
<th>1</th>
<th>Z</th>
<th>W</th>
<th>L</th>
<th>H</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>W</td>
<td>L</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>W</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>L</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>H</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>W</td>
<td>W</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>-</td>
<td>U</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

- Pair wise resolution of signal values from multiple drivers
- Resolution operation must be associative
Example

- Multiple components driving a shared error signal
- Signal value is the logical OR of the driver values

A Complete Example

- Use of unconstrained arrays
  - This is why the resolution function must be associative!
Summary: Essentials of Functions

- Placement of functions
  - Visibility

- Formal parameters
  - Actuals can have widths bound at the call time

- Check the source listings of packages for examples of many different functions

Essentials of Procedures

```vhdl
procedure read_v1d (variable f: in text; v :out std_logic_vector)
--declarative region: declare variables local to the procedure
--
begin
  -- body
  --
end read_v1d;
```

- Parameters may be of mode `in` (read only) and `out` (write only)
- Default class of input parameters is constant
- Default class of output parameters is variable
- Variables declared within procedure are initialized on each call
Procedures: Placement

architecture behavioral of cpu is
---
--- declarative region
--- procedures can be placed in their entirety here
---
begin
process_a: process
--- declarative region of a process
--- procedures can be placed here
begin
---
--- process body
---
end process_a;
process_b: process
--- declarative regions
begin
--- process body
end process_b;
end architecture behavioral;

Placement of Procedures

- Placement of procedures determines visibility in its usage
Procedures and Signals

procedure mread (address : in std_logic_vector (2 downto 0);
signal R : out std_logic;
signal S : in std_logic;
signal ADDR : out std_logic_vector (2 downto 0);
signal data : out std_logic_vector (31 downto 0)) is
begin
ADDR <= address;
R <= '1';
wait until S = '1';
data <= DO;
R <= '0';
end mread;

• Procedures can make assignments to signals passed as input parameters
• Procedures may not have a wait statement if the encompassing process has a sensitivity list

procedure mread (address : in std_logic_vector (2 downto 0);
signal R : out std_logic;
signal S : in std_logic;
signal ADDR : out std_logic_vector (2 downto 0);
signal data : out std_logic_vector (31 downto 0)) is
begin
ADDR <= address;
R <= '1';
wait until S = '1';
data <= DO;
R <= '0';
end mread;

• Procedures may modify signals not in the parameter list, e.g., ports
• Signals may not be declared in a procedure
• Procedures may make assignments to signals not declared in the parameter list
Concurrent vs. Sequential Procedure Calls

• Example: bit serial adder

Concurrent Procedure Calls

architecture structural of serial_adder is
component comb
port (a, b, c_in : in std_logic;
z, carry : out std_logic);
end component;
procedure dff(signal d, clk, reset : in std_logic;
signal q, qbar : out std_logic) is
begin
  if (reset = '0') then
    q <= '0' after 5 ns;
    qbar <= '1' after 5 ns;
  elsif (rising_edge(clk)) then
    q <= d after 5 ns;
    qbar <= (not D) after 5 ns;
  end if;
end dff;
signal s1, s2 : std_logic;
begin
  C1: comb port map (a => a, b => b,
c_in => s1, z => z, carry => s2);
  -- concurrent procedure call
  dff(clk => clk, reset => reset, d => s2,
  q => s1, qbar => open);
end architectural structural;

• Variables cannot be passed into a concurrent procedure call
• Explicit vs. positional association of formal and actual parameters
**Equivalent Sequential Procedure Call**

```vhdl
architecture structural of serial_adder is
component comb
port (a, b, c_in : in std_logic;
z, carry : out std_logic);
end component;
procedure dff(signal d, clk, reset : in std_logic;
signal q, qbar : out std_logic) is
begin
if (reset = '0') then
q <= '0' after 5 ns;
qbar <= '1' after 5 ns;
elsif (clk'event and clk = '1') then
q <= d after 5 ns;
qbar <= (not D) after 5 ns;
end if;
end dff;
signal s1, s2 : std_logic;
begin
C1: comb port map (a => a, b => b,
c_in => s1, z => z, carry => s2);
-- sequential procedure call
-- process
begin
dff(clk => clk, reset => reset, d => s2,
q => s1, qbar => open);
wait on clk, reset, s2;
end process;
end architecture structural;
```

**Subprogram Overloading**

- Hardware components differ in number of inputs and the type of input signals
- Model each component by a distinct procedure
- Procedure naming becomes tedious
Subprogram Overloading

- Consider the following procedures for the previous components:
  - `dff_bit (clk, d, q, qbar)`
  - `asynch_dff_bit (clk, d,qbar,reset,clear)`
  - `dff_std (clk,d,q,qbar)`
  - `asynch_dff_std (clk, d,q,qbar,reset,clear)`

- All of the previous components can use the same name → subprogram overloading

- The proper procedure can be determined based on the arguments of the call
  - Example
    ```
    function "**" (arg1, arg2: std_logic_vector) return std_logic_vector;
    function "++" (arg1, arg2 :signed) return signed;
    -- the following function is from std_logic_arith.vhd
    --
    ```

VHDL is a strongly typed language
Overloading is a convenient means for handling user defined types
We need a structuring mechanism to keep track of our overloaded implementations

Packages!
Essentials of Packages

- Package Declaration
  - Declaration of the functions, procedures, and types that are available in the package
  - Serves as a package interface
  - Only declared contents are visible for external use

- Note the behavior of the `use` clause

- Package body
  - Implementation of the functions and procedures declared in the package header
  - Instantiation of constants provided in the package header

Example: Package Header std_logic_1164

```vhdl
package std_logic_1164 is
type std_ulogic is ('U', -- Uninitialized
  'X', -- Forcing Unknown
  '0', -- Forcing 0
  '1', -- Forcing 1
  'Z', -- High Impedance
  'W', -- Weak Unknown
  'L', -- Weak 0
  'H', -- Weak 1
  '-' -- Don't care
);
type std_ulogic_vector is array (natural range <>) of std_ulogic;
ffunction resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_logic;
type std_logic_vector is array (natural range <>) of std_logic;
ffunction "and" (l, r : std_logic_vector) return std_logic_vector;
end package std_logic_1164;
```
Example: Package Body

```vhdl
package body my_package is
    --
    -- type definitions, functions, and procedures
    --
end my_package;
```

- Packages are typically compiled into libraries
- New types must have associated definitions for operations such as logical operations (e.g., and, or) and arithmetic operations (e.g., +, *)
- Examine the package std_logic_1164 stored in library IEEE

Essentials of Libraries

- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared
Design Units

Primary Design Units

- Distinguish the primary and secondary design units
- Compilation order

Visibility Rules

- When multiple design units are in the same file visibility of libraries and packages must be established for each primary design unit (entity, package header, configuration) separately!
  - Secondary design units derive library information from associated primary design unit
- The use clause may selectively establish visibility, e.g., only the function rising_edge() is visible within entity design-2
  - Secondary design inherit visibility
- Note design unit descriptions are decoupled from file unit boundaries
Summary

• Functions
  – Resolution functions

• Procedures
  – Concurrent and sequential procedure calls

• Subprogram overloading

• Packages
  – Package declaration - primary design unit
  – Package body

• Libraries
  – Relationships between design units and libraries
  – Visibility Rules
Basic Input and Output

File Objects

- VHDL objects
  - signals
  - variables
  - constants
  - Files
- The file type permits us to declare and use file objects

VHDL Program

```vhdl
file: type declaration operations
```

(1)

(2)
File Declarations

- Files can be distinguished by the type of information stored
  
  ```vhdl
  type text is file of string;
  type IntegerFileType is file of integer;
  ```

- File declarations VHDL 1987
  - `file infile: text is in "inputdata.txt";`
  - `file outfile: text is out "outputdata.txt";`

- File declarations VHDL 1993
  - `file infile: text open read_mode is "inputdata.txt";`
  - `file outfile: text open write_mode is "outputdata.txt";`

---

Binary File I/O (VHDL 1993)

```vhdl
entity io93 is -- this entity is empty
end entity io93;
architecture behavioral of io93 is
begin
  process is
    type IntegerFileType is file of integer; -- file declarations
    file dataout :IntegerFileType;
    variable count : integer:= 0;
    variable fstatus: FILE_OPEN_STATUS;
    ```
  ```vhdl
    file_open(fstatus, dataout,"myfile.txt",
      write_mode); -- open the file
    for j in 1 to 8 loop
      write(dataout,count); -- some random values to write to the file
      count := count+2;
    end loop;
    wait; -- an artificial way to stop the process
  end process;
end architecture behavioral;

- VHDL provides read(f, value), write(f, value) and endfile(f)
- VHDL 93 also provides File_Open() and File_Close()
- Explicit vs. implicit file open operations
Binary File I/O (VHDL 1987)

```
-- test of binary file I/O
--
entity io87_write_test is
end io87_write_test;
architecture behavioral of io87_write_test is
begin
process
variable check : integer := 0;
begin
for count in 1 to 10 loop
check := check + 1;
write(dataout, check);
end loop;
wait;
end process;
end behavioral;
```

- VHDL 1987 provides read(f, value), write(f, value) and endfile(f)
- Implicit file open operations via file declarations

The TEXTIO Package

```
line
read(buf,c)
write(buf,arg)
```

- A file is organized by lines
- read() and write() procedures operate on line data structures
- readline() and writeline() procedures transfer data from-to files
- Text based I/O
- All procedures encapsulated in the TEXTIO package in the library STD
  - Procedures for reading and writing the pre-defined types from lines
  - Pre-defined access to std_input and std_output
  - Overloaded procedure names
Example: Use of the TEXTIO Package

use STD.Textio.all;
entity formatted_io is -- this entity is empty
end formatted_io;
architecture behavioral of formatted_io is
begin
process is
file outfile :text; -- declare the file to be a text file
variable fstatus :File_open_status;
variable count: integer := 5;
variable value : bit_vector(3 downto 0):= X"6";
variable buf: line; -- buffer to file
begin
file_open(fstatus, outfile,"myfile.txt", write_mode); -- open the file for writing
write(buf, “This is an example of formatted I/O”);
write(outfile, buf); -- write buffer to file
write(buf, “The First Parameter is =”);
write(buf, count);
write(buf, ‘ ’);
write(buf, “The Second Parameter is = ”);
write(buf, value);
writeline(outfile, buf); -- flush the buffer to the file
write(buf, “…and so on”);
writeline(outfile, buf);
write(outfile, buf);
file_close(outfile);
end process
end architecture behavioral;

Result
This is an example of formatted IO
The First Parameter is = 5 The Second Parameter is = 0110
…and so on

Extending TEXTIO for Other Datatypes

• Hide the ASCII format of TEXTIO from the user

• Create type conversion procedures for reading and writing desired datatypes, e.g., std_logic_vector

• Encapsulate procedures in a package

• Install package in a library and make its contents visible via the use clause
Example: Type Conversion

- Text based type conversion for user defined types
- Note: writing values vs. ASCII codes

```vhdl
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');
  when 'U' => write(buf, 'U');
  when 'Z' => write(buf, 'Z');
  when '0' => write(buf, character('0'));
  when '1' => write(buf, character('1'));
  when '-' => write(buf, '-');
  when 'W' => write(buf, 'W');
  when 'L' => write(buf, 'L');
  when 'H' => write(buf, 'H');
  when others => write(buf, character('0'));
  end case;
end loop;
write(buf);
end procedure write_v1d;
```

```vhdl
procedure read_v1d (variable f: in text; v : out std_logic_vector) is
variable buf: line;
variable c : character;
begin
readline(f, buf);
for i in v'range loop
  read(buf, c);
  case c is
  when 'X' => v (i) := 'X';
  when 'U' => v (i) := 'U';
  when 'Z' => v (i) := 'Z';
  when '0' => v (i) := '0';
  when '1' => v (i) := '1';
  when '-' => v (i) := '-';
  when 'W' => v (i) := 'W';
  when 'L' => v (i) := 'L';
  when 'H' => v (i) := 'H';
  when others => v (i) := '0';
  end case;
end loop;
end procedure read_v1d;
```

- read() is a symmetric process
Useful Code Blocks (from Bhasker95)

• Formatting the output
  write (buf, "This is the header");
  writeline (outfile,buf);
  write (buf, "Clk =");
  write (buf, clk);
  write (buf, ", N1 =");
  write (buf, N1);

• Text output will appear as follows
  This is the header
  Clk = 0, N1 = 01001011

Useful Code Blocks (Bhaskar95)

• Reading formatted input lines
  # this file is parsed to separate comments
  0001 65 00Z111Z0
  0101 43 0110X001
  bit vector    integer    std_logic_vector

• The code block to read such files may be
  while not (endfile(vectors) loop
    readline(vectors, buf);
    if buf(1) = '#' then
      continue;
    end if;
    read(buf, N1);
    read (buf, N2);
    read (buf, std_str);
    convert to std_logic_vector

Useful Code Blocks: Filenames

```
process is
  variable buf : line;
  variable fname : string(1 to 10);
begin
  -- prompt and read filename from standard input
  write(output, "Enter Filename: ");
  readline(input,buf);
  read(buf, fname);
  -- process code
  --
end process;

• Assuming “input” is mapped to simulator console
  – Generally “input” and “output” are mapped to standard input
    and standard output respectively
```

Useful Code Blocks: Testing Models

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use Work.classio.all;
-- the package classio has been compiled into the working directory

entity checking is
end checking;

architecture behavioral of checking is
begin
  -- use file I/O to read test vectors and write test results
  Testing process
end architecture behavioral;
```
Useful Code Blocks: Testing Models (cont.)

```vhdl
process is
  -- use implicit file open
  --
  file infile : TEXT open read_mode is "infile.txt";
  file outfile : TEXT open write_mode is "outfile.txt";
  variable check : std_logic_vector (15 downto 0) := x"0008";
begin
  -- copy the input file contents to the output file
  while not (endfile (infile)) loop
    read_v1d (infile, check);
    write_v1d (outfile, check);
  end loop;
  file_close (outfile); -- flush buffers to output file
  wait; -- artificial wait for this example
end process;
end architecture behavioral;
```

Testbenches

- Testbenches are transportable
- General approach: apply stimulus vectors and measure and record response vectors
- Application of predicates establish correct operation of the model under test
Example

library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use WORK.classio.all;

entity srtester is -- this is the module generating the tests
port (R, S, D, Clk : out std_logic;
Q, Qbar : in std_logic);
end entity srtester;

architecture behavioral of srtester is
begin
 clk_process: process -- generates the clock waveform with
begin -- period of 20 ns
 Clk<= '1', '0' after 10 ns, '1' after 20 ns, '0' after 30 ns;
 wait for 40 ns;
 end process clk_process;

•Tester module to generate periodic signals and apply test vectors

Example (cont.)

io_process: process -- this process performs the test
file infile : TEXT is in "infile.txt"; -- functions
file outfile : TEXT is out "outfile.txt";
variable buf : line;
variable msg : string(1 to 19) := "This vector failed!";
variable check : std_logic_vector (4 downto 0);
begin
 while not (endfile (infile)) loop -- loop through all test vectors in
 read_vld (infile, check); -- the file
 -- make assignments here
 wait for 20 ns; -- wait for outputs to be available after applying
 if (Q /= check (1) or (Qbar /= check(0))) then -- error check
 write (buf, msg);
 writeln (outfile, buf);
 write_vld (outfile, check);
 end if;
 end loop;
 wait; -- this wait statement is important to allow the simulation to halt!
end process io_process;
end architectural;
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.classio.all; -- declare the I/O package
entity srbench is
end srbench;
architecture behavioral of srbench is
--
-- include component declarations here
--
-- configuration specification
--
for T1:srtester use entity WORK.srtester (behavioral);
for M1: asynch_dff use entity WORK.asynch_dff (behavioral);
signal s_r, s_s, s_d, s_q, s_qb, s_clk : std_logic;
begin
T1: srtester port map (R=>s_r, S=>s_s, D=>s_d, Q=>s_q, Qbar=>s_qb, Clk => s_clk);
M1: asynch_dff port map (R=>s_r, S=>s_s, D=>s_d, Q=>s_q, Qbar=>s_qb, Clk => s_clk);
end behavioral;

Stimulus Generation

- Stimulus vectors as well as reference vectors for checking
- Stimulus source
- "on the fly" generation
  - Local constant arrays
  - File I/O
- Clock and reset generation
  - Generally kept separate from stimulus vectors
  - Procedural stimulus
Stimulus Generation: Example (Smith96)

process
begin
    databus <= (others => '0');
    for N in 0 to 65536 loop
        databus <= to_unsigned(N,16) xor shift_right(to_unsigned(N,16),1);
        for M in 1 to 7 loop
            wait until rising_edge(clock);
        end loop;
        wait until falling_edge(Clock);
    end loop;
end process;

- Test generation vs. File I/O: how many vectors would be need?

Stimulus Generation: Example (Smith96)

while not endfile(vectors) loop
    readline(vectors, vectorline);
    if (vectorline(1) = '#') then
        next;
    end if;
    read(vectorline, datavar);
    read(vectorline, A); -- A, B, and C are two bit vectors
    read(vectorline, B); -- of type std_logic
    read(vectorline, C);
    --
    --signal assignments
    Indata <= to_stdlogic(datavar);
    A_in <= unsigned(to_stdlogicvector(A)); -- A_in, B_in and C_in are of
    B_in <= unsigned(to_stdlogicvector(B)); -- unsigned vectors
    C_in <= unsigned(to_stdlogicvector(C));
    wait for ClockPeriod;
end loop;
Validation

- Compare reference vectors with response vectors and record errors in external files
- In addition to failed tests record simulation time
- May record additional simulation state

The “ASSERT” Statement

```plaintext
assert Q = check(1) and Qbar = check(0)
report "Test Vector Failed"
severity error;
```

Example of Simulator Console Output

Selected Top-Level: sbench (behavioral)
: ERROR : Test Vector Failed
: ERROR : Test Vector Failed
: Time: 100 ns, Iteration: 0, Instance: /T1.

- Designer can report errors at predefined levels: NOTE, WARNING, ERROR and FAILURE (enumerated type)
- Report argument is a character string written to simulation output
- Actions are simulator specific
- Concurrent vs. sequential assertion statements
- TEXTIO may be faster than ASSERT if we are not stopping the simulation

(23)
Example: (Bhaskar 95)

```vhdl
architecture check_times of DFF is
constant hold_time : time := 5 ns;
constant setup_time : time := 2 ns;
begin
process
variable lastevent : time;
begin
if d'event then
assert NOW = 0 ns or (NOW - lastevent) >=
hold_time
report "Hold time too short"
severity FAILURE;
lastevent := NOW;
end if;
-- check setup time
-- D flip flop behavioral model
end process;
end architecture check_times
```

• Report statements may be used in isolation

---

Summary

• Basic input/output
  – ASCII I/O and the TEXTIO package
  – binary I/O
  – VHDL 87 vs. VHDL 93

• Testbenches

• The ASSERT statement
Programming Mechanics

Design Units

- Basic unit of VHDL programming is a design unit which is one of the following

- Primary Design Units
  - entity
  - configuration
  - package header

- Secondary Design Units
  - architecture-1
  - architecture-2
  - architecture-3
  - binding
  - package body
Name Spaces

- Multiple instances of a name can exist
  - Local names are the default
  - Provide the full path name for a name

- Visibility of names
  - Visibility of a name follows the hierarchy
    - Declared in a package → all design units that use the package
    - Entity → in architectures used for this entity
    - Architecture → processes used in the architecture
    - Process → in the process
    - Subprogram → subprogram

Compilation, Naming and Linking

- Design unit names are used to construct intermediate file names
- The libraries WORK and STD
Project Management

- Use packages to separate synthesizable from simulatable
- Change secondary design units without recompilation of the design hierarchy
- Use several configurations
  - Record/compare different architectures

Basic Steps: Simulation

- Analysis (Compilation) and Analysis Order
  - Primary vs. secondary design units
  - Organization of design units and files

```vhdl
architecture structural of full_adder is
  component half_adder is
    port (a, b : in std_logic;
        sum, carry : out std_logic);
  end component half_adder;

  component or_2 is
    port (a, b : in std_logic;
        c : out std_logic);
  end component or_2;

  signal s1, s2, s3 : std_logic;
begin
  H1: half_adder port map (a => In1, b => In2, sum => s1, carry=> s3);
  H2: half_adder port map (a => s1, b => c_in, sum => sum, carry => s2);
  O1: or_2 port map (a => s2, b => s3, c => c_out);
end architecture structural;
```
Compilation Dependencies

- Compilation dependencies follow hardware dependencies
  - Changes in the interface
  - Architecture changes can be insulated

- Note that recompilation is interpreted as a change
  - Locating architectures and entities in the same file
  - Creates dependencies that may not in fact exist

Basic Steps: Simulation

- Elaboration
  - Of the hierarchy produces a netlist of processes

- Of declarations
  - Generics and type checking
  - Storage allocation
  - Initialization
Basic Steps: Simulation

• Initialization
  – All processes are executed until suspended by wait statements or sensitivity lists
  – All nets initialized to default or user specified values
  – Initialize simulation time

• Simulation
  – Discrete event simulation
  – Two step model of time
    • Set net values
    • Execute all affected processes and schedule new values for nets
  – Simulator step time
Identifiers, Data Types, and Operators

• Identifiers
  – Basic identifiers: start with a letter, do not end with "_"
  – Case insensitive

• Data Objects
  – Signals
  – Constants
  – Variables
  – Files
VHDL Standard Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Range of values</th>
<th>Example declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>implementation defined</td>
<td>signal index: integer := 0;</td>
</tr>
<tr>
<td>real</td>
<td>implementation defined</td>
<td>variable val: real := 1.0;</td>
</tr>
<tr>
<td>boolean</td>
<td>(TRUE, FALSE)</td>
<td>variable test: boolean := TRUE;</td>
</tr>
<tr>
<td>character</td>
<td>defined in package STANDARD</td>
<td>variable term: character := '@';</td>
</tr>
<tr>
<td>bit</td>
<td>0, 1</td>
<td>signal In1: bit := '0';</td>
</tr>
<tr>
<td>bit_vector</td>
<td>array with each element of type bit</td>
<td>variable PC: bit_vector(31 downto 0)</td>
</tr>
<tr>
<td>time</td>
<td>implementation defined</td>
<td>variable delay: time := 25 ns;</td>
</tr>
<tr>
<td>string</td>
<td>array with each element of type character</td>
<td>variable name: string(1 to 10) :=</td>
</tr>
<tr>
<td>natural</td>
<td>0 to the maximum integer value in the implementation</td>
<td>variable index: natural := 0;</td>
</tr>
<tr>
<td>positive</td>
<td>1 to the maximum integer value in the implementation</td>
<td>variable index: positive := 1;</td>
</tr>
</tbody>
</table>

Data Types (cont.)

- Enumerated data types are particularly useful for constructing models of computing systems
  - Examples
    ```
    type instr_opcode is ('add', 'sub', 'xor', 'nor', 'beq', 'lw', 'sw');
    type state is ('empty', 'half_full', 'half_empty', 'empty');
    ```
- Array types
  ```
  type byte is array (7 downto 0) of std_logic;
  type word is array (31 downto 0) of std_logic;
  type memory is array (0 to 4095) of word;
  ```
Physical Types

**Physical Types**

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

type power is range 1 to 1000000
units
uw; -- base unit is microwatts
mw = 1000 uw; -- milliwatts
w = 1000 mw; -- watts
kw = 1000000 mw -- kilowatts
mw = 1000 kw; -- megawatts
end units;

In terms of base units

Physical Types: Example

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

Physical Types: Example (cont.)

```vhdl
-- 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;

-- Physical Types: Example (cont.)

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;

-- Physical Types: Example (cont.)

• Rather than mapping the values to the real numbers,
  create new physical types

```

Physical Types: Example (cont.)

```vhdl
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;
begine
  o1 <= '1' after tplh when i1 = '0' else
       '0' after tpll when i1 = '1' or i1 = 'Z' else
       'X' after tplh;
end delay;
```

Modeling with Physical Types

- Use packages to encapsulate type definitions, type conversions functions and arithmetic functions for new types
- Examples
  - Modeling power
  - Modeling silicon area
  - Modeling physical resources that are "cumulative"

Operators

- VHDL ‘93 vs. VHDL ‘87 operators

<table>
<thead>
<tr>
<th>logical operators</th>
<th>and</th>
<th>or</th>
<th>nan</th>
<th>d</th>
<th>nor</th>
<th>xor</th>
<th>xnor</th>
</tr>
</thead>
<tbody>
<tr>
<td>relational operators</td>
<td>=</td>
<td>/=</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
<td></td>
</tr>
<tr>
<td>shift operators</td>
<td>sll</td>
<td>srl</td>
<td>sla</td>
<td>sra</td>
<td>rol</td>
<td>ror</td>
<td></td>
</tr>
<tr>
<td>addition operators</td>
<td>+</td>
<td>–</td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unary operators</td>
<td>+</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiplying operators</td>
<td>*</td>
<td>/</td>
<td>mod</td>
<td>rem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miscellaneous operators</td>
<td>**</td>
<td>abs</td>
<td>not</td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- VHDL text or language reference manual for less commonly used operators and types