CMSC 611: Advanced Computer Architecture

Design & Simulation Languages

Practically everything adapted from slides by Peter J. Ashenden, VHDL Quick Start
Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Abstraction Hierarchy of Digital Design

- Digital designers often employ abstraction hierarchy, which can be expressed in two domains:
  - Structural domain: Components are described in terms of an interconnection of more primitive components
  - Behavior domain: Components are described by defining their input/output responses by means of a procedure
Design's Levels of Abstraction

<table>
<thead>
<tr>
<th>Level</th>
<th>Structural Primitive</th>
<th>Behavior Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS</td>
<td>CPU, memories, buses</td>
<td>Performance specifications</td>
</tr>
<tr>
<td>Chip</td>
<td>Microprocessor, RAM, UART</td>
<td>I/O response, algorithms</td>
</tr>
<tr>
<td>Register</td>
<td>ALU, counter, MUX</td>
<td>Truth table, state table</td>
</tr>
<tr>
<td>Gate</td>
<td>AND, OR, flip-flop</td>
<td>Boolean equations</td>
</tr>
<tr>
<td>Circuit</td>
<td>Transistor, R, L, and C</td>
<td>Differential equations</td>
</tr>
<tr>
<td>Silicon</td>
<td>Geometrical objects</td>
<td>Process specifications</td>
</tr>
</tbody>
</table>
Simulation continues until the event queue is empty or stopped externally by the designer.
Domains and Levels of Modeling

Structural

Functional

Geometric

high level of abstraction

low level of abstraction

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

Structural

Functional

Algorithm (behavioral)

Register-Transfer Language

Boolean Equation

Differential Equation

Geometric

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

Structural
- Processor-Memory Switch
- Register-Transfer
- Gate
- Transistor

Functional

Geometric

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

- Polygons
- Sticks
- Standard Cells
- Floor Plan

“Y-chart” due to Gajski & Kahn
Functional Simulator

- Program simulating behavior of design
  - Match interfaces
  - Use any language or algorithm
- Can use to develop external HW or SW
- Graphics examples
  - SGI modified software OpenGL
  - UNC process-per-board
Hardware Design Languages

• A hardware design language provides primitives for describing both structural and behavioral models of the design

• Hardware design languages are useful in
  – Documenting and modeling the design
  – Ensuring design portability

• Every hardware design language is supported by a simulator that helps in:
  – Validating the design
  – Mitigating the risk of design faults
  – Avoiding expensive prototyping for complicated hardware
• VHDL and Verilog are the most famous and widely used hardware design language

• Focus on VHDL:
  – Interfaces, Behavior, Structure, Test Benches
  – Analysis, Elaboration, Simulation, Synthesis
Modeling Digital Systems

• VHDL is for writing models of a system
• Reasons for modeling
  – requirements specification
  – documentation
  – testing using simulation
  – formal verification
  – synthesis
• Goal
  – most reliable design process, with minimum cost and time
  – avoid design errors!
Entity declaration
- describes the input/output ports of a module

```vhdl
entity reg4 is
  port ( d0, d1, d2, d3, en, clk : in bit;
        q0, q1, q2, q3 : out bit );
end entity reg4;
```
VHDL-87

- Omit entity at end of entity declaration

```vhdl
entity reg4 is
  port ( d0, d1, d2, d3, en, clk : in bit;
       q0, q1, q2, q3 : out bit );
end reg4;
```
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*
architecture behav of reg4 is
begin
  storage : process is
    variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    begin
      if en = '1' and clk = '1' then
        stored_d0 := d0;
        stored_d1 := d1;
        stored_d2 := d2;
        stored_d3 := d3;
      end if;
      q0 <= stored_d0 after 5 ns;
      q1 <= stored_d1 after 5 ns;
      q2 <= stored_d2 after 5 ns;
      q3 <= stored_d3 after 5 ns;
    wait on d0, d1, d2, d3, en, clk;
  end process storage;
end architecture behav;
Modeling Structure

• Structural architecture
  – implements the module as a composition of subsystems
  – contains
    • signal declarations, for internal interconnections
      – the entity ports are also treated as signals
    • component instances
      – instances of previously declared entity/architecture pairs
    • port maps in component instances
      – connect signals to component ports
    • wait statements
Structure Example

d0

bit0

<table>
<thead>
<tr>
<th>d_latch</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>

q0

d1

bit1

<table>
<thead>
<tr>
<th>d_latch</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>

q1

d2

bit2

<table>
<thead>
<tr>
<th>d_latch</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>

q2

d3

bit3

<table>
<thead>
<tr>
<th>d_latch</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>

q3

gate

<table>
<thead>
<tr>
<th>and2</th>
<th>a</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

en

clk

int_clk
First declare D-latch and and-gate entities and architectures

```vhdl
entity d_latch is
  port ( d, clk : in bit; q : out bit );
end entity d_latch;

architecture basic of d_latch is begin
  latch_behavior : process is begin
    if clk = ‘1’ then
      q <= d after 2 ns;
    end if;
    wait on clk, d;
  end process latch_behavior;
end architecture basic;

entity and2 is
  port ( a, b : in bit; y : out bit );
end entity and2;

architecture basic of and2 is begin
  and2_behavior : process is begin
    y <= a and b after 2 ns;
    wait on a, b;
  end process and2_behavior;
end architecture basic;
```
Structure Example

- Now use them to implement a register

```vhdl
architecture struct of reg4 is
  signal int_clk : bit;
begin
  bit0 : entity work.d_latch(basic)
    port map ( d0, int_clk, q0 );
  bit1 : entity work.d_latch(basic)
    port map ( d1, int_clk, q1 );
  bit2 : entity work.d_latch(basic)
    port map ( d2, int_clk, q2 );
  bit3 : entity work.d_latch(basic)
    port map ( d3, int_clk, q3 );
  gate : entity work.and2(basic)
    port map ( en, clk, int_clk );
end architecture struct;
```
Mixed Behavior and Structure

- An architecture can contain both behavioral and structural parts
  - process statements and component instances
  - collectively called *concurrent statements*
  - processes can read and assign to signals
- Example: register-transfer-level model
  - data path described structurally
  - control section described behaviorally
Mixed Example

- Multiplier
- Multiplicand
- Shift_reg
- Control_section
- Shift_adder
- Reg
- Product
entity multiplier is
  port ( clk, reset : in bit;
       multiplicand, multiplier : in integer;
       product : out integer );
end entity multiplier;

architecture mixed of multiplier is
  signal partial_product, full_product : integer;
  signal arith_control, result_en, mult_bit, mult_load : bit;
begin
  arith_unit : entity work.shift_adder(behavior)
    port map ( addend => multiplicand, augend => full_product,
              sum => partial_product,
              add_control => arith_control );
  result : entity work.reg(behavior)
    port map ( d => partial_product, q => full_product,
              en => result_en, reset => reset );
...

mixed_example

...  
multiplier_sr: entity work.shift_reg(behavior)  
  port map (d => multiplier, q => mult_bit,  
            load => mult_load, clk => clk);  
product <= full_product;  
control_section: process is  
  -- variable declarations for control_section  
  -- ...  
begin  
  -- sequential statements to assign values to control signals  
  -- ...  
    wait on clk, reset;  
  end process control_section;  
end architecture mixed;
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
entity test_bench is
end entity test_bench;

architecture test_reg4 of test_bench is
  signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;
begin
  dut : entity work.reg4(behav)
    port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
  stimulus : process is
    begin
      d0 <= '1';  d1 <= '1';  d2 <= '1';  d3 <= '1';  wait for 20 ns;
      en <= '0';  clk <= '0';  wait for 20 ns;
      en <= '1';  wait for 20 ns;
      clk <= '1';  wait for 20 ns;
      d0 <= '0';  d1 <= '0';  d2 <= '0';  d3 <= '0';  wait for 20 ns;
      en <= '0';  wait for 20 ns;
      ...
      wait;
    end process stimulus;
end architecture test_reg4;
Regression Testing

• Test that a refinement of a design is correct
  – that lower-level structural model does the same as a behavioral model

• Test bench includes two instances of design under test
  – behavioral and lower-level structural
  – stimulates both with same inputs
  – compares outputs for equality

• Need to take account of timing differences
architecture regression of test_bench is
  signal d0, d1, d2, d3, en, clk : bit;
  signal q0a, q1a, q2a, q3a, q0b, q1b, q2b, q3b : bit;
begin
  dut_a : entity work.reg4(struct)
    port map ( d0, d1, d2, d3, en, clk, q0a, q1a, q2a, q3a );
  dut_b : entity work.reg4(behav)
    port map ( d0, d1, d2, d3, en, clk, q0b, q1b, q2b, q3b );
stimulus : process is
begin
  d0 <= '1';  d1 <= '1';  d2 <= '1';  d3 <= '1';  wait for 20 ns;
  en <= '0';  clk <= '0';  wait for 20 ns;
  en <= '1';  wait for 20 ns;
  clk <= '1';  wait for 20 ns;
  ...
  wait;
end process stimulus;
...
... verify : process is begin
    wait for 10 ns;
    assert q0a = q0b and q1a = q1b and q2a = q2b and q3a = q3b
    report "implementations have different outputs"
    severity error;
    wait on d0, d1, d2, d3, en, clk;
end process verify;
end architecture regression;
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` input
  - `q` output
  - `clk` input

- **bit1**: `d_latch`
  - `d` input
  - `q` output
  - `clk` input

- **bit2**: `d_latch`
  - `d` input
  - `q` output
  - `clk` input

- **bit3**: `d_latch`
  - `d` input
  - `q` output
  - `clk` input

- **gate and2**
  - `a` input
  - `y` output
  - `b` input

- **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
  – specified in wait statements
  – 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

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