Verification

Vienna University of Technology
Department of Computer Engineering
ECS Group
Contents

- Levels of Abstraction
- Testbench
- Modelsim
Overview Hardware Modeling

- Hardware Specification
  - Functional Specification
  - High Level Requirements
  - Detailed Design Description

- Realisation
  - Hardware Description
  - Hardware Implementation

- Verification
- Review
- Formal Verification
- Simulation
Design Flow: Simulation

- VHDL
  - Behavioral Simulation
  - Functional Simulation
  - Prelayout Simulation
  - Postlayout Simulation

- "Code" → Circuit
- Circuit → Technology
- Place & Route
Simulations

- **Behavioral Simulation**
  - Entered VHDL code, no timing information
  - VHDL code not required to be synthesizable

- **Functional Simulation**
  - Simulation of circuit built from generic gates

- **Prelayout Simulation**
  - Technology specific gates
  - Optional Unit-Delays

- **Postlayout Simulation**
  - After Place&Route - real timing
Example: Counter

reg: process (clk, rst)
begin
  if rst = '0' then
    cnt_value_int <= 0;
  elsif rising_edge(clk) then
    cnt_value_int <= cnt_value_nxt;
  end if;
end process reg;

cnt_nxt: process (up, cnt_value_int)
begin
  cnt_value_nxt <= cnt_value_int;
  if up = '1' then
    cnt_value_nxt <= cnt_value_int + 1;
  end if;
end process cnt_nxt;
Example: Behavioral Simulation
Example: Behavioral Simulation

- Active clock edge
- Ideal, concurrent signal switching
Example: Prelayout Simulation
Example: Prelayout Simulation

Ideal, concurrent signal switching
Example: Prelayout Simulation

Ideal, concurrent signal switching

⇒ No standard delays used
Example: Postlayout Simulation
Example: Postlayout Simulation

Realistic signal timings
Example: Postlayout Simulation

No transition on MSB!?
Example: Postlayout Simulation

No transition on MSB!? 

$f_{\text{clk}} > f_{\text{max}}$
Postlayout Simulation

Active clock edge

Signal transitions

Glitch

Result is stored

Computation

\( f_{\text{clk}} < f_{\text{max}} \)
Design Verification

- Application of input stimuli
- Trace outputs
- Compare against expected outputs

Means for interacting with device under test (DUT):
- Using simulation framework for stimulus generation (e.g. Force-command in ModelSim)
- Better: Writing a testbench in VHDL
Testbench Structure

- Testbench consists of:
  - Entity
  - Architecture
  - Configuration

- Processes in architecture apply inputs
- Testbench does not need to be synthesizable
Testbench: Entity

entity xyz_tb is
end entity xyz_tb;

⇒ That's all!
Testbench: Architecture

- At least two processes
  - Clock generation process
  - Process for inputs stimuli

- Good practices
  - Use a constant for defining clock period
  - Write a function which waits a certain number of clock cycles: `icwait (n: integer)`
  - Use loops for repetitive input patterns
  - Use procedures for complex stimuli
Complex Testbenches

- Can contain multiple components
  - Test stubs
- Test automation
  - Comparison with reference data
  - „Golden Node“
  - Assertions for indicating failures
- File I/O
  - Input stimuli
  - Store outputs
Testbenches with Test Stubs

- Test Stub does not need to be synthesizable
- Implements some submodule of your design
- Reacts on output of the DUT
Automated Testcases

- Referenced node
  - VHDL design unit
  - File containing expected outputs
- Reduced efforts for output verification
- Reference node needs to be correct
IO-Files (1)

- Input-File: Provides stimuli
- Output-File: Stores output signal values
- Evaluation after a simulation run
IO-Files (2)

- Package for accessing text files
  - “use std.textio.all”
- Define file objects
  - file inputFile, outputFile: text;
- String variables
  - variable myString: string(MAX_LENGTH downto 1);
  - variable myLine: line;
Read Data

FILE_OPEN( inputFile, “input.txt”, READ_MODE);

while not endfile(inputFile) loop
    readline(inputFile, myLine);
    read(myLine, myString);
    -- Convert string into required data type
    -- Apply stimulus to DUT
end loop;

FILE_CLOSE (inputFile);
Write Data

FILE_OPEN(outputFile, "output.txt", WRITE_MODE);

while not test_finished loop
    -- Generate output sting
    myString := ...
    write(myLine, myString);
    writeline(outputFile, myLine);
end loop;

FILE_CLOSE (outputFile);
Coverage

- Metric for completeness of testcases
- Requirement Coverage
  - Is every requirement covered by a testcase?
  - This coverage needs to determined manually
- Code Coverage
  - Are all parts of the VHDL code covered with the executed testcases?
  - Tool support for coverage analysis
Code Coverage

- Typically used for behavioral simulation
- Code coverage: Analysis how each line of code is executed during a testcase
- Multiple testcases:
  - Coverage statistics need to be combined for overall numbers
Code Coverage

- A coverage below 100% indicates:
  - Incomplete set of test vectors
  - Unreachable code → dead code

- Code, which must NEVER be executed:
  - Insert assertions
Code Coverage - ModelSim

Coverage off / coverage on directives

```vhdl
...  
counter_nxt <= counter + 1;
assert counter < 12
   report "Counter Overflow" severity error
...

...  
counter_nxt <= counter + 1
-- coverage off
assert counter < 12
   report "Counter Overflow" severity error
-- coverage on
...
```

Not included for coverage stat.
architecture mux1 of hdl_bsp is

begin  -- mux1
  mux : process (A, B, C, D)
  begin  -- process mux1
    if C = '0' and D='1' then
      Y <= A;
    else
      Y <= B;
    end if;
  end process mux;
end mux1;
Testbench: Summary

- Use a testbench for simulation
- Testbench can be very simple or highly complex
- Use same testbench for behavioral and post-layout simulation
- Important: Assign valid values to all inputs
- Use code coverage for measuring the quality of your testcases
Modelsim

- Supports different HDLs
  - VHDL, VERILOG, SystemC, ...
- Co-Simulation possible
- User Interface
  - Command line
  - TCL scripts
  - GUI
Modelsim GUI

- **Workspace**
- **Objects**
- **Waveforms**
- **Console**

The image shows a screenshot of the Modelsim SE PLUS 6.1e GUI with various components highlighted:

- **Workspace** displays various folders and files.
- **Objects** show the properties of different objects.
- **Waveforms** show the simulation results.
- **Console** displays the output of the simulation.
**VCOM Command**

- **Compiling VHDL code**
  
  ```bash
  vcom -work work ../vhdl/counter.vhd
  ```

- **Code coverage option**
  
  ```bash
  -cover bcest
  ```

  - b .. branch, c .. condition, e expression, s .. statements, t .. toggle

- **Multiple source files → Pay attention to right order (feature: auto generate order)**
Example: Code Coverage Output
VLIB and VMAP Commands

- VLIB creates a "Design Library"
  vlib altera_mf

- Mapping a "Design Library" to a physical path
  vmap altera_mf /path/to/library
VSIM Command

- Starting the simulation
  
  vsim -t ps work.counter_tb

  time resolution  Testbench/Configuration

- Other arguments
  
  -coverage ... enables coverage evaluation
  -assertfile <filename> ... redirects assertions to file
  -sdfmin,-sdftyp or -sdfmax ... use timing informations
ADD WAVE Command

Add signals to waveform windows

```
add wave  -format logic /counter_tb/clk
```

- Display format
- Signal name

Other arguments

- `-label <name>` ... assign specific name to added signal
- `-divider <divider_name>` ... adds a separator to waveform window
- `-radix` ... specifies the radix
FORCE Command

- Drive signals without a testbench
  - Input Ports
  - Internal signals

```
force object value
```

- Signal name

- Other arguments
  - `-repeat <time>`  ... generate periodic signal transitions
  - `<time>`  ... assigns signal value at certain time
Do-Files

- Scripting mechanism for ModelSim
- Do-files contain
  - ModelSim commands
  - TCL scripts
- Execution of do-files:
  - ModelSim console: do sim.do
  - On startup of ModelSim:
    - vsim -do "sim.do"