Embedded Systems 1: Hardware Description Language (HDL) for Verification

Davide Zoni PhD
email: davide.zoni@polimi.it
webpage: home.deib.polimi.it/zoni
Outline

• How to test an RTL design
  ➔ Verification vs Validation
  ➔ Simulation-based Verification
  ➔ Formal Verification
  ➔ Assertion-based Verification

• Verilog for Simulation-based Verification
  ➔ System tasks: $display, $monitor, $write,$display, $finish
  ➔ Procedural Blocks: initial blocks, always blocks
  ➔ Simulation Delay Models

• Tools: Simulation, Synthesis, Implementation
  ➔ Xilinx ISE Overview

• Hands-on and Examples
  ➔ Combinatorial, Sequential, FSM
  ➔ Embedded Processor: An OpenRisc1000 Implementation

• “Hardware Verification with System Verilog”, Mike Mintz and Robert Ekendahl, Springer 2007
• **Verification**: set of techniques to check if the design satisfies the specification
  - Internal process: from the early design stages till post-implementation
  - It has to comply with a set of specifications that can eventually be ambiguous or incomplete, then leading to errors/bugs

• **Validation**: it answers the question: is the design fulfill with the customer’s expectations?
  - It is an external process
  - The fact that the design is doing something does not mean it is satisfactory
The goal of verification
- Ensure correct timing and functional behavior for the Design Under Test (DUT)
- Usually 50% / 70% of the whole design project
- Verification team is usually twice as big as the design team

Functional Verification
- Simulation
- Formal proof

Timing Verification (NOT COVERED IN THIS CLASS)
- Dynamic Timing Simulation (DTS)
- Static Timing Analysis (STA)
Functional Verification

- Simulation-Based Verification
  - Direct
  - Random
  - Constrained Random

- Formal Verification
  - Model Checking
  - Equivalence Checking
  - Theorem Proving

- Assertion-Based Verification
  - Dynamic Formal Verification
Traditionally, the quality of the verification was measured with the help of code coverage tools. The code coverage reflects how thorough the HDL code was exercised.

- Structural Coverage:
  - This will give information about how many lines are stressed, how many times expressions, branches executed.
  - This coverage is collected by the simulation tools
  - Users can use this coverage to reach those corner cases which are not hit by the random test-cases.

- Functional coverage:
  - This coverage will be defined by the user
  - User will define the coverage points for the functions to be covered in DUT
  - This is under user control
F1: sample the generated tests. The coverage stats are updated even if the test is not actually sent to the DUT

F2: sample the input vector at the DUT input. Collect DUT input coverage information

F3: sample intermediate values within the DUT. Traditionally assertion-based techniques are used to test this portion of the design space

F4: sample the output vector at the DUT output. Collect DUT output coverage information
The (Simplified) Hardware Design Flow

**Flow**

- **Verilog Code**
  - `input a,b;
  - output [1:0];
  - assign sum={1'b0,a}+{1b'0,b};`

- **Post Synthesis Code**
  - `G1 "and" n1 n2 n5
  - G2 "and" n3 n4 n6
  - G3 "or" n5 n6 n7`

- **Post Mapped Code**
  - `G1 "and_tlib" n1 n2 n5
  - G2 "and_tlib" n3 n4 n6
  - G3 "or_tlib" n5 n6 n7`

- **Post Place&Route Code**
  - Netlist + accurate propagation delays
  - Ready to go!
  - [ FPGA, ASIC ]

**Code Transformations**

**Possible Actions**

- **Behavioral simulation**
  - Semantic and syntax checks
  - No physical net/gate propagation delays

- **Post-synthesis simulation**
  - Semantic of the synthesized code
  - No physical net/gate propagation delays

- **Post-mapped simulation**
  - Semantic of the synthesized code
  - Physical gate propagation delays
  - No net propagation delays

- **Post-place and route simulation**
  - Semantic of the synthesized code
  - Physical gate propagation delays
  - No net propagation delays
The Simulation Flow (Verilog 2001)

From previous time slot

• **Blocking assignments**
  - Any order between different always blocks
  - Fixed order within begin/end in the same always
  - Evaluate Right Hand Side (RHS) of Non-blocking assignments (NBAs)

• **Continuous assignments**
• **$display**

Current time slot

• #0 blocking assignments (do not use #0 delay, it is useless)

• Update Left Hand Side of Non-blocking assignments (NBAs)

• $monitor, $strobe system tasks

To next time slot

Source: IEEE Std 1364-2001, i.e. Verilog-2001
The testbench parts:

- **design under test (dut)**: the RTL design that you want to test
- **tb_signal_generator**: it generates the stimuli for the dut
- **tb_monitor**: checks the output of the dut to get functional errors
The *testbench* module

```verilog
`timescale [timeunit]/[timeprecision]

module minimal_tb_template();

always #5 clk<=~clk; // clock generation

reg in1, in2; wire out1; // in/out to the dut

initial // initial block (even more than one)
begin
/* init var + procedures to generate input signals to dut */
end

// dut instance

dut #(parameter .P1(...), ...)
dut0(.in1(in1), .in2(in2), .out1(out1));

// tb_monitor to check the dut output
endmodule
```

- **timescale** directive specifies the unit of time and the precision to be used during the simulation
  - Example: `timescale 1ns/10ps`
  - Inherited from the last processed module if not specified in the .v file

- The *testbench* module has no primary inputs and outputs, but can have parameters

- Verilog comments have the same C/C++ syntax
  - // comment until the end of the line
  - /* multi-line comment */

- Input to the **dut** are **reg**
- Output from the **dut** are **wire**
- The **initial block** is a procedural block that is executed once at the simulation startup
  - Not synthesizable
Clock, Reset and Maintainability
Behavioral Verilog: Clock Signal

- The clock signal is made of non-synthesizable Verilog statements, i.e. delays

- **[RULE of THUMB]** Use blocking assignments to update the clock [in a synchronous design]:
  - Avoid races with other signals that are updated in the same time-step
  - Eventually use non-blocking assignments for all the other signals
  - The use of non-blocking clock update assignment is possible even if it is considered “improper”

```verilog
reg clk;
initial begin
  clk <= 1'b0;
  forever #10 clk = ~ clk;
end
```

```verilog
reg clk;
initial clk <= 1'b0;
always #10 clk = ~ clk;
```

```verilog
reg clk;
initial clk <= 1'b0;
always begin
  #5 clk = 1'b0;
  #5 clk = 1'b1;
end
```
**Behavioral Verilog: Reset Signal**

**IMPROPER DESIGN**
- The clock [clk] and the reset [rst] are updated during the same time-step
  - We can observe simulation misbehaviors

```verilog
always #5 clk=~clk

initial begin //can race!!!
  clk<=0; rst<=1;
  #5 rst=0;
end
```

**CORRECT DESIGN**
- The clock [clk] and the reset [rst] are updated during the same time-step
  - However, we assume clock has precedence to avoid races in the simulation

```verilog
always #5 clk=~clk

initial begin //no races!!!
  clk<=0; rst<=1;
  #5 rst<=0;
end
```
The `timescale directive: rounding and truncation errors

**Truncation**

```
`timescale 1ns/1ns
reg clk;
parameter clk_period=25;
always #(clk_period/2) clk=~clk;
```

**Rounding**

```
`timescale 1ns/1ns
reg clk;
parameter clk_period=25;
always #(clk_period/2.0) clk=~clk;
```

**Correct Implementation**

```
`timescale 1ns/100ps
reg clk;
parameter clk_period=25;
always #(clk_period/2.0) clk=~clk;
```

- The design is simulated using an event driven simulation approach
  - timeunit: the unit of time for any kind of delay
  - timeprecision: how many events the simulator can generate at most between two timeunit steps

- **timeunit** always bigger than **timeprecision**
  - They can be equal, while rounding and truncation errors arise

- Different simulators allow you to specify both **timeunit** and **timeprecision** directives without cluttering the hardware source files
  - Rounding errors: due to the precision of the used computing machine
  - Truncation errors: due to the used arithmetic
Behavioral Verilog: Maintainability 1

**CORRECT DESIGN**

- Parameters are better than defines:
  - parameters are attributes of the module
  - defines are just placeholders

**IMPROPER DESIGN**

- Use `parameter` and `define` to increase the reusability and maintainability of your testbenches
  - `parameter clk_period`
  - `define CLK_PERIOD`

```verilog
always #5 clk=~clk

initial begin //can race!!!
  clk<=0; rst<=1;
  #5 rst<=0;
end

parameter clk_period = 10;
parameter rst_delay=10;
...
always #(clk_period/2) clk=~clk

initial begin //no races!!!
  clk<=0; rst<=1;
  #(rst_delay) rst<=0;
end
```
Behavioral Verilog: Maintainability 2

**CORRECT DESIGN**

- The reset (rst) signal is clear after n_cycle of the clock (clk)
  - @(posedge clk) is a non-synthesizable delay construct
  - @(posedge clk) doesn’t mean anything from the physical viewpoint

**IMPROPER DESIGN**

- Use the clk to synchronize the update of your design synchronous inputs
  - NOTE: maybe not the best choice if the reset is asynchronous with respect to the clock!!!
Delay Models in Verilog
Verilog allows to model delays in the hardware description as well as in the testbenches for behavioral simulations

- **Inertial delay model**: the inertia of a circuit node to change value. It abstractly models the RC circuit seen by the node
  - **ALTERNATIVE DEFINITION**: measure of the input “hold time” to get a change in the output.
  - **NOTE**: A pulse of duration less than the inertial delay does not contain enough energy to cause the device to switch

- **Transport delay model**: the propagation time of signals from module inputs to its outputs
  - **NOTE**: Time taken by the signal to propagate through the net, also known as time of flight

Behavioral Verilog Delay Models: Blocking Assignments (BA\textsuperscript{21})

Adding RHS or LHS delays to model combinatorial logic is common while flawed:
- The behavioral delay is not synthesizable: it does not have a physical counterpart
- Depends on the timescale directive
- Functional Implications

\begin{verbatim}
always@(a) y=~a
\end{verbatim}

- No delay
- \textbullet{} the delay is imposed by the gate from the technology library used to map the NB statement (post synthesis)
- \textbullet{} Correct Implementation

\begin{verbatim}
always@(a) #5 y=~a
\end{verbatim}

- Left-hand-side delay
- \textbullet{} Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 timeunit
  - \textbullet{} What if an update happens at 2.5 timeunits?
  - \textbullet{} The first value is lost!!!

\begin{verbatim}
always@(a) y= #5 ~a
\end{verbatim}

- Right-hand-side delay
- \textbullet{} Immediately evaluate the RHS, then wait 5 timeunit before updating the LHS of the statement
  - \textbullet{} What if an update happens at 2.5 timeunits?
  - \textbullet{} The first value is kept!! (RHS eval immediately)

- Do not use LHS or RHS delays in synthesizable combinatorial logic
- Use LHS only for testbench statements to separate series of stimuli
Adding RHS or LHS delays is also possible for NBAs. While the same implications discussed for BAs are still valid, the RHS delay can be used to model the transport delay in combinatorial logic. Still not synthesizable!

### Always @ (a)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Delay Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always@ (a)</code></td>
<td>No delay</td>
<td>• the delay is imposed by the gate from the technology library used to map the NB statement (post synthesis)</td>
</tr>
<tr>
<td><code>y &lt;= ~a</code></td>
<td>(correct)</td>
<td></td>
</tr>
</tbody>
</table>

- Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 time unit
  - What if an update happens at 2.5 time units?
  - The first value is kept and the following are queued

### Always @ (a)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Delay Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always@ (a)</code></td>
<td>Left-hand-side delay</td>
<td>• Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 time unit</td>
</tr>
<tr>
<td></td>
<td>(HW nonsense)</td>
<td></td>
</tr>
<tr>
<td><code>#5 y &lt;= ~a</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What if an update happens at 2.5 time units?
- The first value is lost!!
- NBA are less efficient to simulate, thus LHS delay for NBA is also discouraged for testbenches.

### Always @ (a)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Delay Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always@ (a)</code></td>
<td>Right-hand-side delay</td>
<td>• Immediately evaluate the RHS, then wait 5 time unit before updating the LHS of the statement</td>
</tr>
<tr>
<td></td>
<td>(correct for transport delay modeling)</td>
<td></td>
</tr>
<tr>
<td><code>y &lt;= #5 ~a</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What if an update happens at 2.5 time units?
- The first value is kept and the following are queued

- Do not use LHS delays to model combinatorial logic using NBA (HW nonsense)
- Use RHS delays to model the transport delay in combinatorial logic using NBA (new output is queued)
Adding RHS or LHS delays to continuous assignments is also possible.

- It makes them not synthesizable
- Used to model the inertial delay in combinatorial logic to prevent undesired glitches

**assign y = ~a**

- No delay (correct)
- the delay is imposed by the gate from the technology library used to map the statement (post synthesis)

**assign #5 y = ~a**

- Left-hand-side delay (correct for inertial delay modeling)
- LHS delay in continuous assignment models the inertial delay of the logic
  - If multiple event updates a within the delay period, only the generated update event is kept
  - NOTE: continuous assignments do not queue new values, while keep the last within the #5 delay to model the inertial delay
  - **Question: why don’t BAs model the inertial delay?**
    - Hint: new event is rescheduled in the future it does not just update the value

**assign y = #5 ~a**

- Right-hand-side delay (illegal syntax)
Design Under Test (DUT)
Observability: System tasks
$monitor, $display, $write, $strobe

Unsynthesizable system tasks to print out useful information on the simulated design.

- **$strobe**: print the values at the end of the current timestep.
  - Executed in the postponed region
  - Example: $strobe("%t %b", time,signal);

- **$display**: print the immediate values
  - Executed in the active region
  - Example: $display("%t %b", time,signal);

- **$monitor**: print the values at the end of the current timestep if any values changed.
  - $monitor can only be called once; sequential calls will override the previous
  - Executed in the postponed region
  - Example: $monitor("%t %b", $time, signal);

- **$write**: same as $display but doesn't terminate with a newline (\n)
  - Executed in the active region
  - Example: $write("%t %b\n", time, signal);
The simulation time

Verilog allows to report the simulation time by accessing to specific system variables

- The time value is always scaled to the timeunit
- The time value can be stored in a variable of proper size if not directly printed

Examples

real time=$realtime;
integer=$stime;
integer=$time;

$display("%t",$time);

- $realtime returns a 64-bit floating point value scaled to the timeunit
- $stime return a 32-bit unsigned integer value
  - If current value is bigger than 32-bit, the lower 32-bit only are returned
- $time return 64-bit integer value scaled to the timeunit

- The $timeformat(<units>, <precision>, <suffix>,<minimum field width>) allows to specify the $time, $realtime, $stime return value
  - If not specified the default values:
    - <units>:
    - <precision>: 0
    - <suffix>: null string
    - <minimum field width>:
Design Under Test (DUT)
Observability: VCD
The **Value Change Dump** file can store all the transitions for all the signals in the design during the entire simulation.

- Verilog offers system tasks to instruct the simulator on the vcd dump procedure.
- Several simulators allow a specific, tool-dependant solution for the VCD file dump without affecting the .v files.
- The VCD can be read using a waveform visualizer:
  - Gtkwave [opensource]
  - Simvision [cadence]
  - Xsim waveform [xilinx, free*]

From within the testbench .v file:

```
$dumpfile("file.vcd");
$dumpvars(level,<module_instance/variable>);

- 0 dump all variable in the module and all of its submodules
- 1 dump all variables in the module
- <level> dump all variables in the module and all the variables in the children up to level = <level>-1
```