Logic Simulation with Xcelium

Prof. Adam Teman

Emerging Nanoscaled
Integrated Circuits and Systems Labs

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Introduction

- **In the lecture, we learned about writing RTL:**
  - Basic syntax
  - Constructs
  - Writing synthesizable RTL
  - Recommended coding style

- **We will now see how to run a logic simulation that performs a direct test**
  - We will write a Testbench that drives stimuli for our block
  - We will instantiate the DUT (RTL block) within the testbench
  - We will run the simulation and look at the signal waveforms
  - We will look at the Tcl commands that enable us to automate this process

Cadence Xcelium

- **Xcelium is Cadence's tool for logic simulation and verification.**
- **It encompasses several tools in one combined executable, called `xrun`.**
  The operations performed by `xrun` include (among others):
  - Compilation: Compile HDL code written in Verilog, System Verilog, VHDL, etc.
  - Elaboration: Elaborate the compiled design into an internal database.
  - Simulation: Simulate the behavior of the testbench and DUT
  - Debug: View signals on Waveforms and provide additional debugging tools
- **When running `xrun`, some important files are created, including:**
  - `xrun.log`: The logfile with all the errors and warnings.
  - `xrun.history`: The last command line used to invoke Xcelium
  - `worklib/`: scratch directory with compiled files

Running Xcelium

- **The `xrun` command does it all…**

```
xrun myfile.v
```

- **`xrun` has many control options to control it, such as:**
  - `<filename>.v`: Verilog files to compile.
  - `-debug` or `-access +rw`: provide access to simulation objects.
  - `-gui`: Opens the GUI of SimVision to see waveforms and debug.
  - `-y <directory>`: compiles RTL files in the directory you put here.
  - `-v <file.v>`: Specify the name of the library file to use.
  - `-clean`: Delete the `INCA_libs` directory before executing.
  - `-compile`: Parse/compile the source files, but do not elaborate.
  - `-elaborate`: Parse/compile the source files, elaborate the design, and generate a simulation snapshot, but do not simulate.
  - `-f <file.args>`: pass a file of arguments to `xrun` (e.g., a list of files to compile)
  - `-input <file.tcl>`: pass a TCL file to run after elaboration.

Commonly used flags

- **In addition to the previous options, several commonly used flags include:**
  - `-sv`: Accept System Verilog code (belongs to the xmvlog tool)
  - `-v93`: Accept VHDL 93 updates (belongs to the xmvhdl tool)
  - `-debug`: Saves data base info for displaying waveforms and debugging.
  - `-gui`: Opens up the Simvision gui.
  - `-timescale`: Adds a default `timescale directive for Verilog modules that are missing them.
  - `-define`: Define a value to send to the Verilog code
  - `-nospecify`: Disregards any annotated delays

- **These options are included in the xrun_options.rtl file, so just run:**

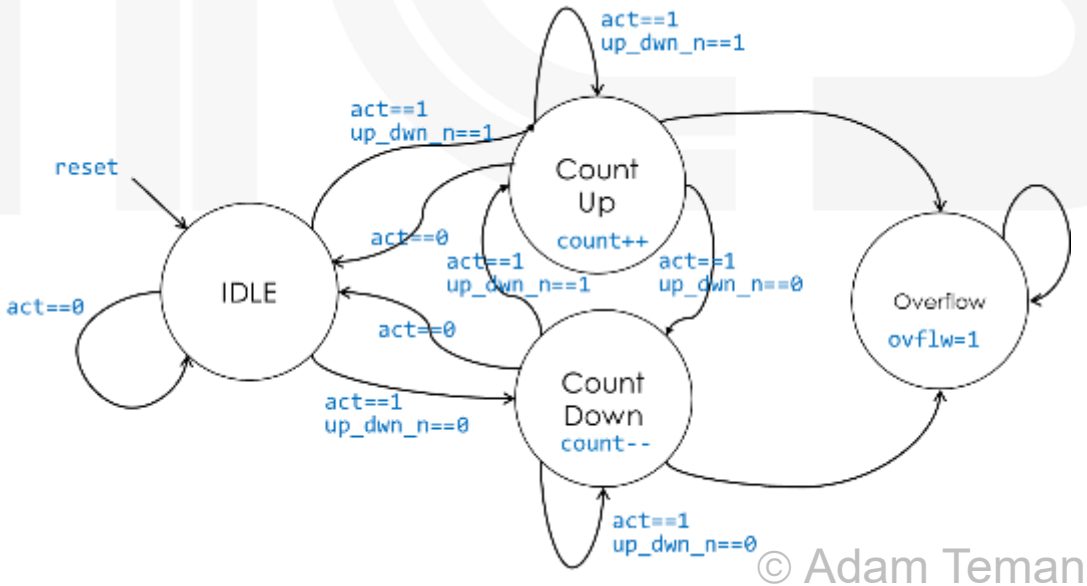
```
xrun -f ../scripts/xrun_options.rtl
```

4-bit Counter Example Reminder

- **A 4-bit counter**
  - Receives 4 inputs:
    - `clk` – the system clock
    - `rst_n` – an active low reset
    - `act` – the activate signal
    - `up_dwn_n` – count up (positive) or count down (negative)
  - Outputs 2 signals:
    - `count`: the current counted value
    - `ovflw`: an overflow signal

```
module sm #(parameter COUNTER_WIDTH = 4)
      (clk,rst_n,act,up_dwn_n,count,ovflw);
  input clk;
  input rst_n;
  input act;
  input up_dwn_n;
  output [COUNTER_WIDTH-1:0] count;
  reg [COUNTER_WIDTH-1:0] count;
  output ovflw
  reg [3:0] state, next_state;
```

```
localparam IDLE  = 4'b0001;
localparam CNTUP = 4'b0010;
localparam CNTDN = 4'b0100;
localparam OVFLW = 4'b1000;
```



Testbench Reminder

```
module sm_tb;
  parameter WIDTH = 5;
  reg clk, rst_n, act, up_dwn_n;
  wire [WIDTH-1:0] count;
  wire ovflw;
```

```
initial begin
  clk = 1'b1;
  rst_n = 1'b0; // Activate reset
  act = 1'b0;
  up_dwn_n = 1'b1;

  // Monitor changes
  $monitor("%t:
      rst_n=%b act=%b up_dwn_n=%b
      count=%d ovflw=%b\n",
      $time,rst_n,act,up_dwn_n,count,ovflw);

  // After 100 time steps, release reset
  #100 rst_n = 1'b1;
end
```

- **Define a clock:**

```
always
   #5 clk = ~clk;
```

- **Set stimuli:**

```
initial begin
  // @100, Start counting up
  //       until overflow
  #100  act = 1'b1;
        up_dwn_n = 1'b1;
  // Reset (100 cycles pulse)
  #1000 rst_n = 1'b0;
        act = 1'b0;
  #100  rst_n = 1'b1;
  // Do a count-up to 4 and
  //    then count-down to ovflw
  #100 act = 1'b1;
       up_dwn_n = 1'b1;
  #40  up_dwn_n = 1'b0;
 end
endmodule
```

Summary

- **In this tutorial, we saw how to use Xcelium to run a logic simulation.**
- **This type of simulation is better known as “direct test”, as the testbench was written specifically to test a specific scenario.**
- **This is by no means what we usually refer to as “verification”.**
  - For “functional verification”, we need to create stimuli to simulate as many possible scenarios as possible and assert all internal functionalities of our DUT.
  - For “formal verification”, we need to mathematically prove that our system is behaving correctly.
  - For “physical verification”, we need to ensure that the physical implementation of our system adheres to what we have designed and meets technology rules.
- **All of these are beyond scope of this series of demonstrations, but are critical components of the design process.**