Gate-level Simulation, Timing Backannotation, and Power Estimation

Prof. Adam Teman

Emerging Nanoscaled Integrated Circuits and Systems Labs

The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

Introduction

- **At this point we have:**
  - Written our RTL
  - Performed (at least very basic) functional verification
  - Synthesized our design

- **We now will perform three steps for further verification and analysis:**
  - Gate-Level Simulation (zero delay model)
  - Gate-Level Simulation with Timing Backannotation
  - Static* Vector-based Power Calculation

**I know it says "Static" but sounds like "Dynamic"… Bear with me…*

Gate-Level Simulation

- **Gate-level simulation (GLS) is a direct test that is run on a structural netlist (i.e., constructed exclusively of standard cells).**
  - GLS can be run at any stage from post-synthesis to post place and route.
- **Why run GLS?**
  - Ensure desired functionality is not lost in the translation from RTL to GTL.
  - Ensure timing delays do not break intended functionality.
- **Some specific motivations for GLS:**
  - Mistakes in SDC (e.g., wrong false paths or unconstrained paths)
  - Verifying system initialization and reset sequence, including power intent.
  - Simulations that become relevant after RTL, including DFT and CTS.
  - Verifying asynchronous interfaces and multi-cycle paths.
  - Capturing switching activity for power estimation.

Types of Gate-Level Simulation

- **Zero-Delay Simulation**

```
%> xrun -nospecify ...
```

  - Simulating the netlist without annotating any timing data.
  - `specify` block delays are ignored (`-nospecify` switch or `-delay_mode zero`)
  - Used for system initialization and reset sequence, DFT verification, verification of power up/reset operation of power domains.
- **Unit-Delay Simulation**

```
%> xrun -delay_mode unit ...
```

  - Every element has one unit delay.
  - More simple than SDF Backannotation and can reveal races and loops.
- **SDF Backannotation**
  - Annotates gates and nets with delays from STA/extraction.
  - Used to verify clock tree synthesis, desired frequency verification, STA tool limitations, revealing glitches and power estimation.

```
$sdf_annotate(<design_name>.sdf,<DUT>,,"sdf.log" ,"MAXIMUM");
```

SDF Backannotation

- **Timing backannotation is done with a "Standard Delay Format" (SDF) file**
  - In Genus/Innovus use the `write_sdf` command
- **The SDF file has delay information for every instance, wire and timing checks:**
  - Instance:

```
(CELL
   (CELLTYPE "DFF") (INSTANCE U1)
       (DELAY  (ABSOLUTE
          (IOPATH (posedge clk) (posedge q) (1.2:1.5:1.8))))
       (TIMINGCHECK
          (SETUP (posedge d) (posedge clk) (0.8:1.0:1.2))
          (HOLD  (posedge d) (posedge clk) (0.2:0.3:0.4))))
```

  - Wires:

```
(CELL
  (CELLTYPE "digital_top") (INSTANCE)
     (DELAY (ABSOLUTE
        (INTERCONNECT in1 top/my_reg/D (0.027::0.028) (0.029::0.030))
        (INTERCONNECT top/my_reg/Q my2_reg/D (0.0::0.002) (0.0::0.003))
```

SDF Backannotation

- In our Verilog testbench we add `$sdf_annotate` to read the SDF file:

```
initial
  begin
    $sdf_annotate(<design_name>.sdf,<DUT>,,"sdf.log" ,"MAXIMUM");
  end
```

- And we run our simulation with SDF flags:

```
%> xrun +sdf_verbose -sdfstats sdf_stats.txt ...
```

- And we can see the delays in the simulation waveforms:

Glitch Propagation (Delay Models)

- **Zero-delay simulation propagates signals through logic gates immediately.**
- **But once delays are added, varying arrival times create "glitches".**
  - Depending on the pulse width of the glitch, a digital gate may filter it out.
- **Therefore, simulators support two delay models for signal propagation:**
  - **Transport Delay**: Signals propagate through gates after a delay (no filtering).
  - **Inertial Delay**: The signal is filtered if the input signal had a short pulse.
- **The threshold for pulse propagation is relative to the delay and controlled by:**
  - `-pulse_r <arg>` / `-pulse_int_r <arg>`:
    A glitch (pulse) shorter than `<arg>`% of the gate/wire delay will be filtered.
  - `-pulse_e <arg>` / `-pulse_int_e <arg>`:
    A glitch (pulse) shorter than `<arg>`% of the gate/wire delay will flag an error.

Power Estimation

- **Power estimation is a critical part of the design and analysis process, and like just about everything else, opens up a whole new can of worms.**
- **Standard cells and other macros have power information, provided inside `.lib` files, which include:**
  - **Switching power**: the energy consumed by charging/discharging an output load. The switching power is different for each timing arc.
  - **Internal power**: the energy consumed due to switching of internal nodes or short circuit power, independent of the output load. This can depend on a pin toggling and/or a specific timing arc.
  - **Leakage power**: the power consumed by devices while they are not switching. This depends on the state of the inputs/internal nodes.
- **We will now use this information to calculate the system power consumption.**

Types of Power Calculation

- **Well, here's another somewhat confusing one…**
- **Cadence's power analysis tool, Voltus, differentiates between:**
  - **Static Power Analysis**: For calculating *average power/current* per cell.
  - **Dynamic Power Analysis**: For calculating *a power waveform* per cell.
- **While that may seem straightforward – it isn't, because both of these have:**
  - **Vectorless**: Based on average activity factors.
  - **Vector-based**: Based on actual simulation vectors.
- **But here we are only running "Static" analysis to calculate power.**

Vector-based Static Power Analysis

- **When we run `report_power` after synthesis:**
  - `create_clock` command is used to control clock toggle rate.
  - `set_switching_activity` command can be used to control activity on specific nets, such as primary inputs, clock gates and sequentials.
  - All other nets either get propagated activity or a globally defined factor.
- **But this is very inaccurate, especially if not carefully defined.**
- **Therefore, we should read in actual activity from a simulation:**
  - During simulation write an activity file (VCD, TCF, SAF, SAIF, FSDB)
  - Read in the activity file using the `read_activity_file` command
  - Calculate the actual energy consumed at every net with `report_power`

Vector-based Static Power Analysis

- **Dump the activity file (`VCD` or `TCF`) from your simulation**
  - `VCD`: Value Change Dump – all signal changes in the simulation.

```
database -open <design>.vcd -vcd
probe -create <DUT> -vcd -depth 9 -all -database <design>.vcd
```

  - `TCF`: Toggle Count Format – Statistics of toggling of each net.

```
dumptcf -output <design>.tcf -scope <TB>.<DUT>
run -time <simulation run time>
dumptcf -end
```

- **Then read the activity file in Voltus and report power:**

```
%> voltus -stylus
voltus> read_db <design>.db
voltus> read_activity_file -format VCD -scope <TB>.<DUT> <design>.vcd
voltus> report_power -method static -rail_analysis_format VS -out_file power.rpt
```

Summary

- **In this demonstration, we covered:**
  - The motivation for Gate-Level Simulation.
  - Running Zero-Delay Gate-Level Simulation
  - Back-annotating timing information from Synthesis to Simulation, and running a Gate-Level Simulation with applied delays.
  - Writing out activity information (VCD/TCF) and using Voltus for vector-based power estimation for a given (typical) workload.
  - This power estimation is much more accurate than vectorless analysis.

- **While this was demonstrated at the post-synthesis stage, everything that was shown here is applicable throughout place and route until signoff.**