18-360
Intro to Computer-Aided Digital Design
ECE Department

Carnegie Mellon

Out: 4/1/04   Project 2 Transition Counting   Due: 4/27/04

When building low power devices for handheld electronics, it is important to be able to model and measure the power dissipation of a circuit. That's the topic of this project. The goals of this project are to gain insight into the modeling and design of lower power circuits through a simulation model.

Work alone or in groups of two.

Power dissipation in CMOS circuits

The power dissipation in a CMOS circuit is the sum of a static and dynamic component. We will ignore the static component and estimate the dynamic power dissipation in a CMOS logic gate by counting the number of times the gate's output transits from zero-to-one. i.e., the gate only dissipates power changing logic value from zero-to-one. Although this isn't quite true, it is a reasonable approximation.

So, to estimate the power dissipation, all we need to do is to have gate models that count their output zero-to-one transitions, and drive the inputs with a bunch of test vectors. Not so fast! What if a gate's inputs change several times in zero time — a possibility when the various inputs may change or there are zero delay gates? In that situation, we might count several zero-to-one changes when there only should be one. Or we might count several when should be none! Clearly we need to be more clever.

Behavioral models for logic gates

Verilog gate primitives do not do transition counting. The approach we'll take is to write behavioral models of the gates we need. We'll use the statements in an always block to deal with the counting.

The model should also include parameters to specify different inertial and transport delays. That is, instead of assuming that they're equal, as Verilog's gate primitives do, we'll allow them to be different (of course, inertial must be less than or equal to transport). These will be determined by parameter specifications which can be overridden at instantiation time.

What function are we to implement?

You'll build two versions of a circuit to implement the following equation:

```
reg [15:0] a, b, c, d, e;
a = b + c + d + e;
```

Assume that each 16-bit add of two operands takes a combinational logic delay of time τ. We will load the results into behaviorally-modeled registers — there will be no transition counting in them. Here are two versions of the circuit to consider.

- The adds are organized like a balanced tree and the operations occur in 2τ time. That is, you will have a clock placed at 2τ (2τ is its period) that will load the results of the adds. The balanced tree will implement $a = (b + c) + (d + e)$.

- The adds are again organized like a balanced tree, but the operations are pipelined with a clock of period 1τ. Thus, $b + c$ and $d + e$ will be loaded into a register after 1τ. Then, the sum of these two registers will be loaded into register a after another 1τ. Given that the two stages of adds are occurring at the same time, test vectors will be applied every 1τ, the circuit will be pipelined, and a result will come out every 1τ, even though it still takes 2τ to produce the result.

Steps

Environment setup:

We are going to use Synopsys Design Compiler to generate gate-level circuits this time. So copy the Synopsys setup file from /afs/ece/class/ee360/bin/.synopsys_dc.setup into your home directory.

In your working directory, create a sub-directory called "work" which is the design library directory used by Design Compiler.

Copy the directory of /afs/ece/class/ee360/bin/cell_library/verilog/ into your working directory. This directory has verilog behavioral description of all the basic gates, buffers, flip-flops, etc. that the Design Compiler maps your design onto. Some of them might not be useful in this project.

Build the circuits and simulate:

Implement a 1-bit full adder with behavioral level description. Instantiate your 1-bit adder to make a 16-bit adder using the ripple-carry-adder technique. Build the two circuits above. and show that they correctly produce the same results. Set the clock rate the same for each circuit. The clock rate can be any value at this step.

Synthesize the circuits:

Synthesize the circuits using Synopsys Design Compiler using the default synthesis flags. Write the output into a gate-level Verilog description.

Add delay and transition counting code:

This is the really fun part. Remember in the "verilog" directory, we have behavioral description of all the gates. Now you can add delay and counting codes into those modules. You might not need to work on each of those. Look at your synthesized gate-level design, and figure out which gate modules you actually need to edit in order to simulate just your design.

For example, in your gate-level Verilog file, there is an AND gate:

and2_2x_2x z1(p1, p2, p3);

Accordingly you need to look for the file named "and2.v" in the "verilog" directory. However, the module name in that file is and2_1x_1x (here 1x means the driving capability, so and2_1x_1x has the same functionality as and2_2x_2x). You can just change the module name, or copy it to another new module named and2_2x_2x. So at the end of this step, you will have a gate-level design, and all the basic gate modules needed in your design with counting code added.

Specify various delays (inertial and transport delay) for the basic gates in your library. For example, for a third of gates have 3,5, (3 for inertial delay, and 5 for transport delay), a third have 2,3, and another third have 1,2. Maybe make a few of the gates have zero propagation delay. (We're doing this so that we get lots of ripples in the outputs of the gates — which is typical of real logic.)

Simulate and measure the power dissipation:

Set the clock period to be a reasonable value according to your gate delay — just enough time for all the logic changes to settle. Run a whole bunch of input vectors through the circuit and compare the power dissipation and energy dissipation of the two circuits. Explain the differences. Use the $random function to produce thousands of inputs.

Compare different synthesis flags:

In design compiler, set the compilation flag to be "minimize delay" or "minimize area". Compare these two designs with the one in Step 3 (area/delay/power/energy) and explain the differences.

Suggest a different structure for the adder.

Implement the two versions of the circuit as in Step 2. Synthesize them using the same synthesis flags as in Step 3. Compare the power/energy dissipation. Are the results the same as what you expected? Explain why.

Complications

You can count transitions in the whole circuit by having an integer in the top module that is initialized to zero at the start of the simulation. Through hierarchical naming conventions, each module in the design can reference that variable and increment it (or decrement it). Or you can call a function defined in the top module that updates it. Look up functions and hierarchical naming in the book.

The messy part here is making sure that we only count one zero-to-one transition per actual zero-to-one transition. Realize that a behavioral model may execute several times during zero time and its output may change each time. You don't know if the current execution of the model is the last. So, you may have to check the time to see if now is the first time you've run in a new time (meaning the last execution was the last time in previous time). Or, a zero-to-one transition may be cancelled because the inertial delay isn't met — don't want to count that transition!

Helpful hints:

- $time is a system function that returns the current version of time. It can be assigned to a variable of type "time." $stime is similar except that it returns an integer.
- There is an interesting feature of a non-blocking update that we didn't cover. Let's say that several non-blocking updates are made to the same register and are scheduled for the same time. in the simulation cycle where the simulator retrieves these updates from the event list, they are executed in the order in which they are originally entered in the list. i.e., the last one out was the last assignment made and is the final value of the register. (Play around with it some and check that out.)
- It's probably best to stick with one simulator.
- No "PLI" stuff.

Grading

Parts of your grade depend on:

- A demo — to see your design simulating and to ask you questions about your implementation.
- A write-up — describing and discussing the results of each of the "steps" above. Of specific interest is how you handled the "complications," and what you did to make your design more power/energy-efficient. i.e., explain your gate model.

The project is due in class on the due date. 10% per day late penalty.