UNIVERSITY OF CALIFORNIA AT BERKELEY
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Lab 4
Verilog Simulation Mapping

1 Motivation
In this lab you will learn how to use a hardware description language (verilog) to create a design. You will explore different levels of implementation, varying the level at which the definition of your design is crafted structurally versus behaviorally.

2 Introduction
Modern day circuit design is done mainly using hardware description languages (HDLs.) Two popular HDLs are verilog and VHDL. In this class we will use verilog. Verilog can be written using either structural or behavioral descriptions. The code can either use portable standard verilog or manufacturer specific components such as Xilinx primitives. For the purposes of this lab, we will attempt to restrict ourselves to the use of portable standard verilog.

Once written, verilog code can then be run through sophisticated CAD tools such as a logic synthesis tool that will generate the actual low-level gates. In the case of an FPGA the verilog will create a netlist of LUTS, FFs and CLBs. In the case of an ASIC the result may be a netlist of transistors. A simulation can be run on the functional verilog code or after synthesis on the generated netlist. The functional simulation verifies the functionality of the verilog code but does not include realistic timing information, such as gate delays. After the code is run through a logic synthesis tool the timing simulation run on the generated netlist includes timing statistics yielding more accurate results.

3 Prelab & Brief Overview of Structural vs. Behavioral
WARNING: ***You will not finish this lab during the lab section unless you do the Prelab beforehand!***

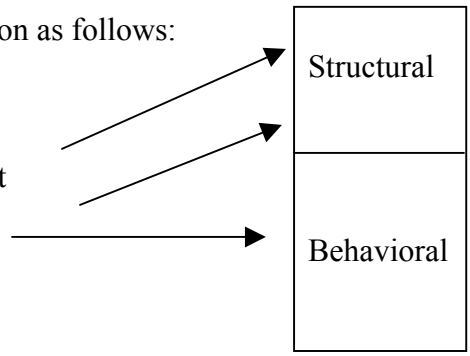
1. Read and understand the entire lab handout.
2. Create a verilog testbench to test your designs. A skeleton testbench is provided for you on the website (in the same place this write-up is located).
3. Write the verilog for all four parts of the lab. You will need as much time in lab as possible to debug your code.
4. This lab assumes some familiarity with the tool flow presented in the previous lab. Re-reading portions of the previous lab assignment to re-familiarize yourself with the tool flow may be beneficial.

Don’t expect to be able to write and debug your code during the lab session.

Structural vs. Behavioral.

One may consider the relation between the structural and behavioral nature of pieces of a design as being directly linked to the point at which subcomponents cease to have subcomponents.

One can visualize the situation as follows:



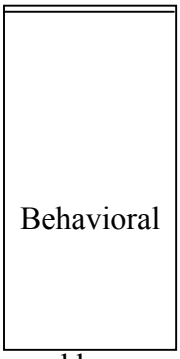
Each subcomponent exists at some particular level

If the level of design for a component is structural, then it will be composed of one or more sub modules, each of which exists closer to (or in) the behavioral portion of the design. The division of a design into structural and behavioral parts can be thought of as proportional to the degree of complexity of the behavioral components used in the design. Using this lab's accumulator as an example, the four parts of the lab break down as follows:

Part I.

(Hierarchy view)

Accumulator

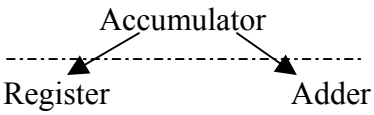


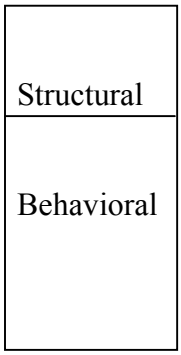
Modules:
Accumulator

Entirely behavioral, there are no structural modules, and hence no discernable division between the two portions of the design.

Part II.

(Hierarchy view)

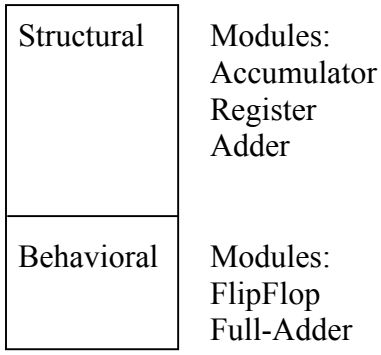


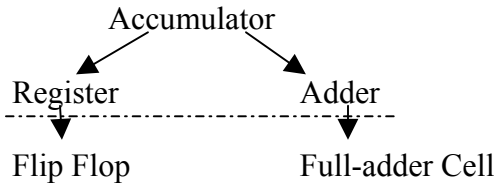


Modules:
Accumulator

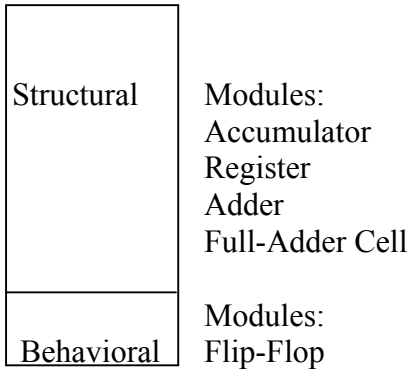
Modules:
Register
Adder

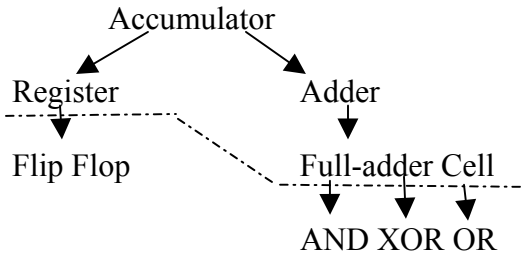
Part III.





Part IV.





4 Procedure

Part I – Behavioral Accumulator

1. In the text editor of your choice, create a new file and name it `top.v`. In this file create a module called top. Add four ports to your module:

Port Name	Width	Direction	Description
Clk	1	Input	Clock Signal
Rst	1	Input	Reset signal for any FFs
In	16	Input	The input to your design
Out	16	Output	The output of your design

2. Using a purely behavioral specification, create your entire accumulator in this module.
3. Using ModelSim, test the functionality of your design with your verilog test bench.
4. Once you are convinced your design is functional, use Synplify to synthesize your design. Open the Xilinx Design manager and proceed with the Placement and Routing stages.
5. Examine the reports generated, noting the timing constraints and number of LUTs used. Open up the Floorplanner and visually examine the layout.
6. Open ModelSim and use the timing constraints from the reports along with your testbench to verify the delay of the critical path.

Part II – Structural Accumulator from a Behavioral Register and a Behavioral Adder

1. Create a new verilog file called `top.v` with a module called `top` with the same port list as Part I (put it in another directory than the one in which the first top.v was defined).
2. Create your design using a behavioral specification for an adder and a behavioral specification for a register, each defined as its own module. Instantiate one of each of these, and connect them together (structurally) in your `top` module (in top.v).
3. Do steps 4 and 5 from Part I. If the mapping report shows a difference in the number and/or type of parts used, can these differences be noticed in your visual inspection via the floorplanner?
4. Do step 6 from from Part I

Part III – Structural Accumulator from Structural Register Composed of Behavioral FlipFlops and a Structural Adder Composed of Behavioral Full-Adder Cells

For this part follow the same steps as in Part II, except use verilog primitives and additional levels of hierarchy to build the adder and register. **Do not use the verilog +,-,/,* functions for the adder**. Create full-adder cells with boolean equations using `assign` statements with `&`, `|`, `^`, and `~` (Thus, this will be a `Dataflow` definition of the full-adder cells). Connect the full-adder cells together to form a ripple adder, and connect the FlipFlops together to form a register. The adder and register thus created should be compatible with your previous instantiations of a behavioral register and adder in your top module, so the top module should not require any code changes.

Part IV – Structural Accumulator from Structural Register Composed of Behavioral FlipFlops and a Structural Adder from Structural Full-Adder Cells Composed of Instances of Logic Gates.

Here one last level of hierarchy is introduced by changing each full adder cell into a collection of gates instead of a dataflow description. Only the module defining a full adder cell should require changes.

Repeat implementation, testing, and measurement procedures as per the previous sections.

6 Acknowledgments

Original lab by J. Wawryznek, N. Zhou, and Y. Patel.
UCB 3 2002
Additional changes by J. Sampson. 9 2002
EECS150 Fall 2002 Lab 4 *Verilog Simulation Mapping*

7. Checkoff

Name: _____ Name:_____

Section: _____

Part I

1. Analysis of mapping report TA:_____(5%)_____(2.5%)
2. Verification of critical path via testbench TA:_____(20%)_____(10%)

Part II

1. Analysis of mapping report TA:_____(5%)_____(2.5%)
2. Verification of critical path via testbench TA:_____(20%)_____(10%)

Part III

1. Analysis of mapping report TA:_____(5%)_____(2.5%)
2. Verification of critical path via testbench TA:_____(20%)_____(10%)

Part IV

1. Analysis of mapping report TA:_____(5%)_____(2.5%)
2. Verification of critical path via testbench TA:_____(20%)_____(10%)

Total

TA:_____/_____(100%)