Simulating Verilog Gates

Background

The Wikipedia article on Verilog starts off with "Verilog, … is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction." Most of you are taking or have taken a Digital Logic course, so should be very familiar with digital circuits by now. The first installment of the project dealt with the capability to simulate the actions of four very simple gate types, an INV gate, an AND gate, an OR gate and an XOR (exclusive or) gate. The second installment extends that capability by describing the simple logic gates using Verilog.

Verilog Infrastructure

We talked about the structure of a Verilog file, as well as the infrastructure I have created that reads a Verilog file and makes the information from that file available to you via a series of function calls to the Verilog infrastructure functions. Reread the notes for lecture P01 for a complete description of these functions.

I have also included a C program called "printCircuit.c" which is an example of how to use the Verilog infrastructure. Look at this code to get some idea of how to use the infrastructure.

You may also look through the code that implements the Verilog infrastructure, but there are some concepts used in that code that we haven't discussed yet. We will talk more about the code when we discuss these concepts.

Project Description

Create a C program called simGate.c. This program needs to:

1. Check to make sure there are at least 3 command line inputs specified. If not, print a message that indicates what the command line should be, and exit with a non-zero return code.
2. Invoke the Verilog openModule statement passing argv[1] in as the argument, and keeping track of the module number returned. (If there is a problem reading the Verilog file, openModule will exit the program for you, so there is no need to check to see if it worked.)
3. Loop through the remaining command line inputs. Invoke the "atoi()" library function to get the integer value represented by each command line argument, and use that value to set each module input pin in the Verilog circuit to that value, using the order the module input pins appear in the Verilog module header. If there are not enough command line values, print the names of the input module pins which were not set. If there were too many command line values specified, print the values not used in a message.
4. Propagate all input pin values to nets.
5. Propagate all net values to net sinks.
6. Evaluate all instances and set the instance output pin based on the input values.

7. Propagate all input pin values to nets.
8. Propagate all net values to net sinks.
9. Print the result to the user using the printModuleIO function.
10. Invoke the freeModule function for your module.
11. If your program worked correctly, return an exit code of zero.

The following is an example of a correctly coded program:

```
./simGate AND_2 1 0
AND_2 Inputs: A0=1, A1=0; Outputs: Z=0;
```

Hints and Suggestions

- Remember, argv[0] is the command itself. You might want to use this in your messages instead of hard coding “simGate”.
- In this installment of the project, you will use the entire command line argument string for each command line argument. Pass the first command line argument (other than the command itself) into the openModule() function. Pass the remaining command line arguments into the atoi() function.
- You can use the same logic you used to complete installment 1 to evaluate the gate instances. Feel free to cut and paste from your installment 1 submission (if you got it right.)
- Remember, you cannot compare two strings using the “==” or “!=” operators. You must include string.h, and use the library function strcmp to compare two strings. The strcmp function returns a zero if its two arguments are equal, or a non-zero if they are different. So for instance, to check to see what kind of gate is instanced, you will need to code: strcmp(instanceModuleName(in),”inv”), and check to see if the result is zero or no-zero.
- Don’t forget to include “verilog.h” in your program, and to compile using the command:

  ```
  gcc -g -Wall -o simGate simGate.c verilog.c
  ```

Project Submission

Upload your simGate.c file in the Project Installment 2 submission area on MyCourses in the Content area under “Project Submissions”.

Project Grading

After the due date, your submission will be graded as follows. The project is worth 100 points. Your code will be compared to all other student’s code using an automated code plagiarism checker that can detect copied code even if you try to “fix” it after copying. Then, your code will be compiled on a BU Linux machine. If there are compiler errors, the professor will attempt to fix your code. If your code can be fixed, the rest of the grading will be performed on that fixed code. Once compiled, your code will be tested with various simple Verilog circuits that contain a single gate with various inputs. The results will be compared to correct results, based on the requirements above. Points will be deducted for the following reasons:

Problem	Deduction
Bad submission (e.g. name not simGate.c)	-5 points
Compiler or logic error that can be fixed	-15 points per error
Unfixable error, or >4 errors	-60 points
Compiler warning messages	-5 points per type of warning message
Illegible or Poorly formatted code	Up to -10 points
Incorrect results	-10 points per invocation (up to -40 points)
Late submission	-10 points per every 24 hours late
Code matches another student’s code	-100 points