# Programming Assignment One: Reading Combinational Circuit and Evaluating Its Outputs

**Out: May 30, 2017; Due: June 14, 2017**

## I. Motivation

1. To give you experience in reading a circuit from a netlist file.
2. To give you experience in implementing a topological sorting algorithm to evaluate the output values of a circuit.

## II. Programming Assignment

You will read two files. The first file describes a combinational circuit. The second file describes the values for all the primary inputs of the circuit. Your task is to calculate the values for all the primary outputs of the circuit. For example, we could give you a netlist file describing the circuit shown in Figure 1 below and another file specifying the values of the inputs X1, X2, X3, and X4 as 1, 0, 0, and 1, respectively. We want you to calculate the value of the outputs X6 and X7 of the circuit. As we stated in lecture, you will represent a combinational circuit as a direct acyclic graph. To obtain the values for all the outputs, you can calculate the logical values for all the gates in a topological order.

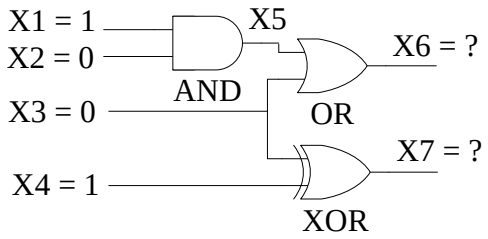


Figure 1. A combinational circuit.

## 1. Format of Netlist File

We use the "bench" format to describe the circuit netlist. For example, a description for the circuit in Figure 1 is shown below.

```
# A circuit with four inputs and two outputs
INPUT(X1)
INPUT(X2)
INPUT(X3)
INPUT(X4)
OUTPUT(X6)
OUTPUT(X7)

# Lines below describe the gates
X5 = AND(X1,X2)
X6 = OR(X3,X5)
X7 = XOR(X3,X4)
```

The file first specifies all the primary inputs of the circuits using the statement `INPUT(<SIG_NAME>)`, where `<SIG_NAME>` is the name of an input signal. Each line specifies one input. Then all the outputs are specified using the statement `OUTPUT(<SIG_NAME>)`, where `<SIG_NAME>` is the name of an output signal. Each line specifies one output.

Next, the file describes all the gates in the circuit, with each line depicting one gate. The general description of a gate uses the following format

<OUTPUT_SIG> = <GATE_TYPE>(<INPUT_SIG1>, <INPUT_SIG2>, ..., <INPUT_SIGN>)

<OUTPUT_SIG> is the name of the output signal of the gate in the netlist. <GATE_TYPE> specifies what kind of gate it is. In this project, we only consider the following gate types:

- AND gate: It is specified by the keyword `AND` or `and`. Its number of inputs is $\geq 2$.
- OR gate: It is specified by the keyword `OR` or `or`. Its number of inputs is $\geq 2$.
- NAND gate: It is specified by the keyword `NAND` or `nand`. Its number of inputs is $\geq 2$.
- NOR gate: It is specified by the keyword `NOR` or `nor`. Its number of inputs is $\geq 2$.
- XOR gate: It is specified by the keyword `XOR` or `xor`. Its number of inputs is $\geq 2$. The output value of an XOR gate with $N$ inputs is recursively defined as follows:
  $\text{XOR}(X_1, X_2, ..., X_N) = \text{XOR}(\text{XOR}(X_1, X_2, ..., X_{N-1}), X_N)$
- XNOR gate: It is specified by the keyword `XNOR` or `xnor`. Its number of inputs is $\geq 2$. The output value of an XNOR gate with $N$ inputs is recursively defined as follows:
  $\text{XNOR}(X_1, X_2, ..., X_N) = \text{XNOR}(\text{XNOR}(X_1, X_2, ..., X_{N-1}), X_N)$

- Inverter: It is specified by the keyword `NOT` or `not`. Its number of inputs is 1. Its output value is the complement of its input value.
- Buffer: It is specified by the keyword `BUF` or `buf`. Its number of inputs is 1. Its output value is the **same** as its input value.

Note: there are two choices of keywords for specifying each gate type: one with all letters capital and the other with all letters lowercase.

<INPUT_SIG1>, <INPUT_SIG2>, ..., <INPUT_SIGN> in a gate description specify all the input signals of the gate. The valid number of inputs for each gate type is mentioned above.

Examples: `X5 = AND(X1, X2, X3)` specifies a 3-input AND gate with output as X5 and 3 inputs as X1, X2, and X3.

As shown in the example at the beginning of this section, we also allow comments in the netlist file. They are lines with the first character as "#". You just ignore them when processing the file. Also, we could have empty lines as shown in the same example. Similarly, you just ignore them.

Finally, we allow a user to put spaces in describing the inputs, outputs, and gates. You should take care of these spaces. The rules are as follows:

- For each input or output specification, the user can put spaces after "(" or before ")". In other words, they can put spaces surrounding the signal name. For example, the following specifications are both valid:
  ```
  INPUT(abc)
  INPUT( abc )
  ```
- For each gate specification, the user can put spaces freely. This means that the spaces can be put around "=", "(", ",", and ")". In other words, the specification is valid as long as there is no space **within** the signal names or gate type keywords. For example, the following specifications are all valid:
  ```
  X5=AND(X1,X2,X3)
  X5 = AND(X1, X2, X3)
  X5=AND( X1,X2,X3 )
  ```

**Hint**: If you use C++, to realize the above required functionality, you may want to use the function `getline()`, the function `substr()` of the `string` class, and the function `find_first_of()` of the `string` class.

## 2. Format of Input Value File

The input value file specifies the value for each input. The number of lines in the file equals the number of inputs of the circuit. Each line has the following format:

```
<INPUT_SIG> <VALUE>
```

where <INPUT_SIG> is the input signal name and <VALUE> is the value for that signal. Below is an example for the circuit shown in Figure 1.

```
X1 1
X2 0
X3 0
X4 1
```

## 3. Output

Your program writes to the **standard output**. Each line shows the value of an output in the following format:

```
<OUTPUT_SIG> <VALUE>
```

where <OUTPUT_SIG> is the output signal name and <VALUE> is the value for that signal. The outputs are printed out in the same order as they are shown in the netlist file. For example, for the circuit shown in Figure 1, your program should print out

```
X6 0
X7 1
```

## III. Program Arguments and Error Checking

Your program takes two arguments. The first is the name of the netlist file and the second is the name of the input value file. Let your compiled program be `p1`. It should be invoked as

```
./p1 <netlist-file> <input-value-file>
```

You do not need to do any error checking. You can assume that all the inputs are syntactically correct.

## IV. Programming Language and Environment

We do not specify any programming language you use. However, we do recommend you to use C or C++ and develop your code in Linux environment (please also refer to Section VII Testing below). Ubuntu Linux operating system is recommended. You can download it from http://www.ubuntu.com/. You can install it directly on your physical machine or on a virtual machine that lives on your physical machine. For the latter choice, you need to install a virtual machine first. For example, you can use VMware Player, which can be downloaded from http://www.vmware.com/. The installation is pretty simple and won't take you too much time.

## V. Compiling

In order to let us test your code automatically, we ask you to provide us with a `Makefile`, if possible (For more information about the `Makefile`, you can read some online tutorials. For example, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ A demo of `Makefile` can be found in the Programming-Assignment-1-Related-Files.zip). Otherwise, please tell us how to compile you code.

## VI. Implementation Requirements, Restrictions, and Hints

- You must make sure that your code compiles successfully. **Your compiled program should be named as `p1` exactly.**
- Output should only be done where it is specified.
- **Hint**: to reduce the work load, you can use some available commonly-used standard libraries. For example, if you use C++, you can use the standard template library (STL).

## VII. Testing

To help you debug your code, we have given you one test case including the bench format netlist `test.bench`, the input value file `test.val`, and the correct output `test.out`. You can find these files in the Programming-Assignment-1-Related-Files.zip. This test case actually corresponds to the circuit shown in Figure 1.

To see if your program runs correctly on this test, copy these three files to your working directory and execute the following commands in Linux:

```
./p1 test.bench test.val > mytest.out
diff mytest.out test.out
```

This runs your program, taking inputs from the files `test.bench` and `test.val` and placing output into the file `mytest.out` instead of the screen. (Here, ">" is the Linux output redirection facility, which redirects the output from the screen to the file `mytest.out`.) Then, the `diff` program compares your test output `mytest.out` with the correct output `test.out`. If `diff` reports nothing, your program passes this test case. If `diff` reports any differences at all, you have a bug somewhere.

We will test your code using this test case, as well as a number of others. You should therefore definitely pass this test case. However, you should also create a number of test cases that exercises your program with different circuit netlists and input values, since the test case we have given you are not sufficient to catch all bugs.

## VIII. Submitting and Due Date

You should submit your source code files together with a `Makefile` or information on how to compile your code. The output program must be named `p1`. The submission deadline is 11:59 pm on June 14th, 2017. You should put all the files into a single `.tar/.zip/.rar` file and submit the `.tar/.zip/.rar` file through the assignment link on Canvas.

You should name the `.tar/.zip/.rar` file with your last name and first name, e.g., `QianWeikang.tar`.

## IX. Grading

We will grade your assignment by running a variety of test cases against your program, checking your solution using our automatic testing program. Your final grade is the percentage of the test cases for which your program produces the same answer as ours.