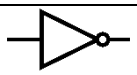
# Simulating Logic Gates

## Background

The Wikipedia article on logic gates starts off with "In electronics, a logic gate is an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs and produces a single binary output." Most of you are taking or have taken a Digital Logic course, so should be very familiar with logic gates by now. The first installment of the project deals 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.

## Logic Gate Truth Tables

The behavior of gates can be described using a truth table. The following truth tables describe the simplest case for the four gate types this project is dealing with. In these tables, the column "A" represents the truth value of the first input, "B" represents the truth value of the second input, and "Z" represents the resulting truth value at the output of the gate.

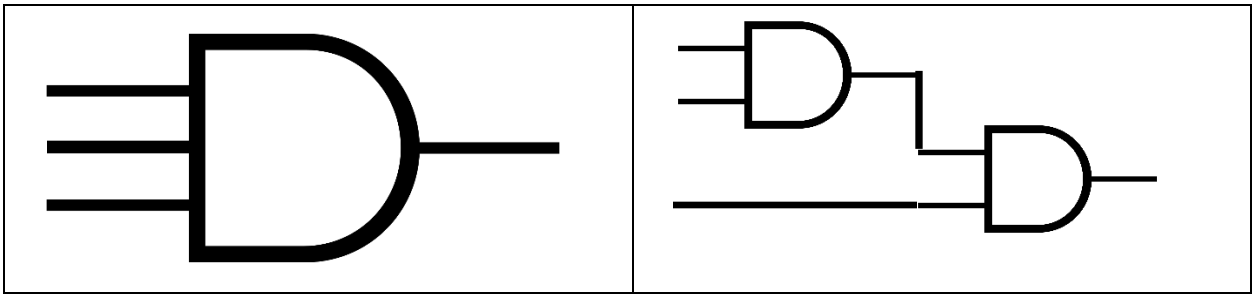
| Gate Type | A | Z |
|---|---|---|
|  INV (N) | 0 | 1 |
| | 1 | 0 |

| Gate Type | A | B | Z |
|---|---|---|---|
|  AND (A) | 0 | 0 | 0 |
| | 0 | 1 | 0 |
| | 1 | 0 | 0 |
| | 1 | 1 | 1 |

| Gate Type | A | B | Z |
|---|---|---|---|
|  OR (O) | 0 | 0 | 0 |
| | 0 | 1 | 1 |
| | 1 | 0 | 1 |
| | 1 | 1 | 1 |

| Gate Type | A | B | Z |
|---|---|---|---|
|  XOR (X) | 0 | 0 | 0 |
| | 0 | 1 | 1 |
| | 1 | 0 | 1 |
| | 1 | 1 | 0 |

An inverter may only have a single input, but the other three gates may have 1, 2, or more inputs. A non-inverter with 1 input acts like a buffer – the output value is the same as the input value. When there are more than two inputs, you can think of the multi-input gate as cascading two-input gates. For instance, a three-input AND gate can be thought of as two cascading two-input AND gates, as in the following picture:



## Project Description

Create a C program called qGate.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. Check the first command line input to make sure it is either 'N', 'A', 'O', or 'X'. If not, print a message and exit with a non-zero return code.
3. Loop through the remaining command line inputs. Check the input to make sure it is either a '0' or '1' to represent false and true respectively. If not, print an error message and exit with a non-zero return code. If there is more than one input for an inverter, issue an error message and exit with a non-zero return code.
4. Using all the input values, evaluate the resulting value.
5. Print the result to the user. Your result should be in the form "*T*(*A*,*B*,….) = *Z*", where *T* is the valid gate type supplied by the user, *A* is the first input value, *B* is the second input value, and so on, and *Z* is the result you have calculated. Your output should occur on its own line, and no other output should be produced by your program (unless the input values are incorrect.)
6. If your program worked correctly, return an exit code of zero.

The following is an example of a correctly coded program:

```
>./qGate A 1 0 1
A(1,0,1) = 0
>
```

## Hints and Suggestions

- Remember, argv[0] is the command itself. You might want to use this in your messages instead of hard coding "qGate".
- In this installment of the project, all you will need from the command line argument is the first character of each argument. To get the first letter of the $i^{th}$ command line argument, you can use the notation: argv[i][0].
- I found it easiest to predict the output of a gate, and then when I looped through the inputs, I used the next input value to modify that output value. When I finish looping through the inputs, I already have my answer calculated.
- One of the challenges is to make the invocation string (for instance, the "A(1,0,1)" string in the example above.) I started out by creating an area for this string as follows:

  char invoke[100]={0};

  This makes an "invoke" area that is 100 characters long (plenty big enough for this project), and initializes each character to a null terminator. Then, I can put the gate type in the first character, and the left parenthesis in the second by doing…

  invoke[0]=gateType; invoke[1]='(';

  Then, as I loop through the input characters, I can use the control variable for the loop

to figure out where the next input character should be in the invoke string. Note that each input will have two characters… the input value, and a comma When I'm done looping through the input, I just replace the last comma with a right parenthesis. Then, I can print the final result using:

```
printf("%s = %d\n",invoke,answer);
```

## Project Submission

Upload your qGate.c file in the Project Installment 1 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 values for gate type and input values (including invalid input), and the results will be compared to correct results, based on the requirements above. Points will be deducted for the following reasons:

| Problem | Deduction |
|---|---|
| Incorrect submission (e.g. name not qGate.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 |