ECE 321L - Introduction to Software Engineering
Lab assignment

1 Introduction

Logic gates are the heart of digital electronics. A gate is an electronic device which is used to compute a function on a two valued signal. Logic gates are the basic building block of digital circuits. We can connect any number of logic gates to design a required digital circuit. Practically, we implement the large number of logic gates in *Integrated Circuits (ICs)*, by which we perform complicated operations at high speeds. By combining logic gates, we can design many specific circuits like flip flops, latches, multiplexers, shift registers etc.

1.1 Boolean Expressions and Truth Tables

In basic algebra, every function has its own equation. Similarly in the field of digital electronics, every gate logic function has its own equation called a *Boolean expression*. The nineteenth century British mathematician, George Boole, invented a type of algebra that uses only two conditions or states. The two states are true and false. This type of algebra using only two states is called *Boolean algebra* in honor of Boole.

In Boolean algebra, the *true state* is represented by the number one, called *logic high* or *logic one*. The false state is represented by the number zero, called *logic low* or *logic zero*. In the field of digital electronics, logic high is represented by the presence of a voltage potential. Logic low is represented by the absence of a voltage potential. The logic high in a program mable logic controller is represented with five volts ($+5V$), and the logic low is represented with zero volts ($0V$).

By applying conventional algebra, you can plot a function's input and output points to create the characteristic of the function as a graph. This graph represents the function pictorially in an x-y coordinate system. The x-axis is for the input points and the y-axis is for the output points. In Boolean algebra, a table contains the digital input and output points. This table is called a *truth table*.

1.2 Combinational and Sequential Logic Gate Circuits

Combinational logic gates do not require clock pulses to operate. Their outputs depend only on their inputs. This means that the outputs of combinational logic gates are generated instantaneously. Generally, the combinational logic gates are simply called logic gates. Seven logic gates exist: NOT, AND, OR, NAND, NOR, XOR (exclusive OR), and XNOR (exclusive NOR). The gates in a circuit represent a simple Boolean expression. For example, two-input AND gates with inputs A and B and output X graphically represent the expression $X = A \cdot B$. Figure **??** presents the output (X) of all the seven gates for two inputs (A, B). By combining the aforementioned gates, more complicated circuits can be developed. Since the output depends only on the input, the final values depends on the combination of the logical gates. Figure **??** displays a logic gate circuit which shows the connection of logic gates for a Boolean expression.

Sequential logic devices have outputs that depend on their inputs as well as time. They require clock pulses. Therefore, an inherent delay time is always present for the sequential logic circuits. Flipflop devices such as reset-set (RS), JK, delay (D), and toggle (T) are sequential logic devices. Figure **??** displays a sequential logic circuit.

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
Symbol							
Truth Table	A X 0 1 1 0	B A X 0 0 0 0 1 0 1 0 0 1 1 1	B A X 0 0 1 0 1 1 1 0 1 1 1 0	B A X 0 0 0 0 1 1 1 0 1 1 1 1	B A X 0 0 1 0 1 0 1 0 0 1 1 0	B A X 0 0 0 0 1 1 1 0 1 1 1 0	B A X 0 0 1 0 1 0 1 0 0 1 1 1

Figure 1: Truth tables for seven gates.

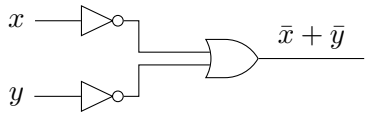


Figure 2: A two-input logic gate circuit

Figure 3: Sequential logic circuit. These circuits require clock pulses.

1.3 Logic Gate Circuits

By combining the aforementioned logic gates, we can design more complex circuits. Figure **??** displays a Boolean expression and its truth table. Note that the upper bar symbol indicates the inverse value of the input. Table **??** presents the truth table of the Boolean expression $Y = A \cdot B + \overline{A} \cdot C$.

Figure 4: Logic circuit for Boolean expression $Y = A \cdot B + \overline{A} \cdot C$

Table 1: Truth table of the Boolean expression $Y = A \cdot B + \overline{A} \cdot C$

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2 Lab assignment

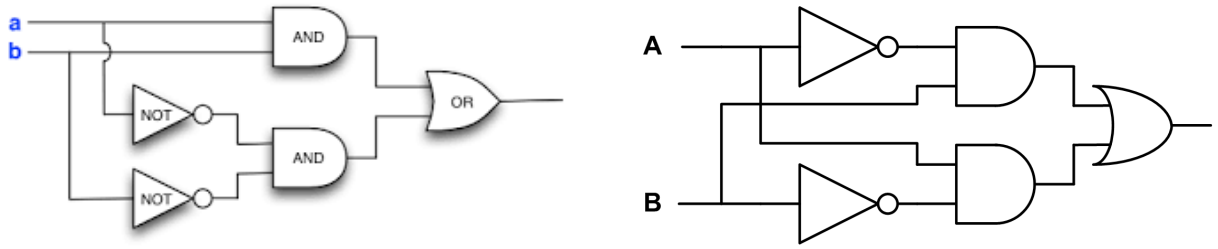
The focus of this lab series is to create a logic circuit simulator in `C` programming language. Specifically, there will be multiple milestones and students have to present the correct functionality of their implementation

2.1 Milestone 1: Logic gates library

The *first step* of this milestone is to create a file that contains the implementations in `C` programming language of the seven logical gates. Specifically, each gate, must be represented as a separate function with two integer arguments (except for the NOT gate that will have one argument). The return value should be the output of the gate. The body of the functions must be located in a separate file named `gates.c` and the corresponding prototypes in the file named `gates.h`.

The *second step* of this milestone is to write a program in a file named `logic_simulator.c` that utilizes the already developed logic gates. The program should ask from the user which gate to simulate and then ask for the appropriate input. After that, the program should print the output of the logical gate and verify it by printing the truth table as well. Each gate can be indexed according to Figure **??**. For example, when the program asks for the gate to simulate, the user can enter 1 for NOT, 2 for AND, 3 for NAND etc. Last, the program should be continuous until the user types 'Q'.

The *third step* of this milestone is to combine the developed logical gates in order to create more complicated circuits. Write a program in a file named `logic_simulator_circuit.c` which simulates the two circuits presented in Figure **??**. The program should ask form the user the values for the input for both circuits and then print the final output. Extra points if the calculation is performed without extra variables (i.e. only use variables for the input and the final output).



(a) Circuit 1 (b) Circuit 2

Figure 5: Custom circuits

2.2 Milestone 2: n-bits logic gates library

The goal of this milestone is to enhance the simulator and the library that you created in the previous deliverable in order to handle input with n-bits. Specifically, consider that your simulator takes two signals A and B and each signal has a size of n bits. Consequently, the logic gates must be modified in order to address cases with more than two bits.

(a) AND gate with 8 bits input (b) OR gate with 16 bits input

Figure 6: AND and OR gate with more than two bits as inputs. The AND gate has 8-bit input while the OR gate has 16-bit input

The *first step* is to enhance the files `gates.c` and `gates.h` by adding new functions `AND_N` and `OR_N` which will take as an argument an array with all the bits and perform the corresponding actions (logical and, logical or). We will consider that only these two logical operations work on more than two bits. For example, Figure **??** shows the AND gate with 8 bits. Specifically, from signal A we take the first 4 bits ($A[0, 3]$) and from the second signal we take the rest bits ($B[4, 7]$). Similarly, Figure **??** shows the OR gate with 16 bits input. The AND_N gate returns 1 if all values are 1 and the OR_N returns 1 if at least one input is 1.

(a) Circuit 1 with $n = 8$ (b) Circuit 2 with $n = 16$

Figure 7: Custom circuits with n-bits input. The brackets next to A and B represent which bits of the two signals are used as input to the gates

The *second step* of this milestone is to combine the developed logical gates in order to create more complicated circuits. Write a program in a file named `logic_simulator_circuit_n.c` which simulates the two circuits presented in Figure **??**. Initially, you should get the signals A and B. Each signal has a size n *which is the same for* A *and* B and it is mentioned in the caption of each circuit. This parameter n should be defined in your program as a directive and stay constant for a simulation (since the value of n is different for each circuit you need to change it and recompile your program every time you want to simulate a different circuit). Based on this value, you should define corresponding *arrays* to handle the bits of the input. Your declaration must be generic. The program should also ask if the user wants to give the values for each bit of the signals or the program initializes everything with random values. For this reason you can use the function `rand()` function defined in `stdlib.h`. Then, utilize your functions written in `gates.c` to calculate the output of the circuits. You may need to declare additional variables in order to separate the bits of the signals and calculate the output.

2.3 Milestone 3: Generic description of the circuit

The goal of this milestone is to enhance the simulator and the library that you created in Milestone 1 in order to handle complex circuits that are described in a separate file.

In this Milestone, we will consider only circuits with a single output composed of `AND`, `OR`, `XOR` and `NOT` gates. Each gate has at most two inputs, but it can have many outputs. Circuit inputs and outputs will be treated like gates. We will use a file to describe the topology of the circuit. The first line (line 0) contains the number of gates n in the circuit. This number includes the inputs and the single output of the circuit. Each next line is used to describe the properties of each gate. The remaining lines contain information for one circuit input, logic gate, or circuit output as described

below. All gates are identified by their line number in the file. Each line consists of the following fields:

- Field 0: the index i of the gate, followed by a colon.
- Field 1: the letter 'A', 'O', 'X', 'N', 'I', or 'Q'. This designates the gate as an `AND` gate, an `OR` gate, an `XOR` gate, a `NOT` gate, a circuit input, or a circuit output, respectively.
- Field 2: two gate indices. These represent the two gates that are directly wired as inputs to gate i. We use the value 0 to denote gate inputs that are not used. For example, `AND`, `OR` and `XOR` gates always use both gate inputs, `NOT` gates uses only one gate input; circuit inputs do not use any gate inputs; circuit output use only one gate input.

You should consider that the circuit is topologically ordered (a gate cannot use a signal that has not appeared in the file before it). Once the simulator has read the circuit description, it asks the user to enter a value (0 or 1) for each of the circuit inputs.

In order to calculate the output, you have to create a linked list (e.g. FIFO, stack, double linked list) that contains the gates. For each gate in the list you should calculate the output value based on the values of the previous nodes. An example of a file and the corresponding circuit is given below:

```
7
1:  I   0 0
2:  I   0 0
3:  N   2 0
4:  A   1 3
5:  O   1 4
6:  X   5 3
7:  Q   6 0
```

2.4 Milestone 4: Multiple-input generic description of a circuit

The goal of this milestone is to enhance the simulator that you created in Milestone 3 in order to handle complex circuits with multiple inputs.

In this Milestone, we will consider circuits with a single output composed of `AND`, `OR`, `XOR` and `NOT` gates. `AND` and `OR` gates can have multiple inputs while `XOR` and `NOT` gates have only one input. We will use a file to describe the topology of the circuit. The first line (line 0) contains the number of gates n in the circuit. Each line can have a different number of fields depending on the number of inputs. However, for all of them Field 0 represents the index i of the gate, followed by a colon and Field 1, which can be the letter 'A', 'O', 'X', 'N', 'I', or 'Q', designates the type of the gate. Once the simulator has read the circuit description, it asks the user to enter a value (0 or 1) for each of the circuit inputs. An example of a file and the corresponding circuit is given below:

```
11
1:   I
2:   I
3:   I
4:   I
5:   I
6:   I
7:   N    1
8:   A    7  2  3
9:   O    8  4  5
10:   X     9  6
11:   Q     10
```

For your implementation, you can use a variation of the input by omitting the first line and/or the : character.