CSCE3303 02 - Fundamental Microelectronics (Spring 2021)

Fundamental Microelectronics:

Final Project Report

CMOS Circuit SPICE Generator

Ahmed Essam Abdelaleem 900193476
Iman Ahmed Attia 900192510

Department of Computer Science and Engineering, AUC

TABLE OF CONTENTS	
1.	**SYSTEM OVERVIEW**
1.1	A Brief Description of the project
1.2	Algorithms Implementation & Data structures
1.3	Extra Features (Bonus)
1.4	User Guide for how to use our program
2.	**TESTING**
2.1	Test Cases
2.2	Schematics of the boolean expression
3.	**TEAM CONTRIBUTIONS**
3.1	Members Contribution
3.2	Reflection on the Experience

1. SYSTEM OVERVIEW

1.1 A Brief Description of the project

SPICE (Simulation Program with Integrated Circuit Emphasis) is a widely-used online tool in building electronic circuits. It is a great assist in describing the connections of the components of these circuits for the users by displaying the SPICE netlist or the SPICE deck.

In this project, we aim to implement a SPICE netlist generator using C++ programming language. First of all, we utilize the knowledge of how to implement logic circuits using CMOS transistors in generating the SPICE netlist of the implemented circuit. Thus, the program receives the input from the user as a boolean expression and implements the boolean expression using CMOS transistors to finally generate the SPICE netlist which describes the connections of the implemented circuit.

Keywords: SPICE netlist - CMOS - PUN - PDN - Logic Circuit - OR - AND - NOT

1.2 Algorithms Implementation & Data structures

I. Logic gates building functions

We start by building 3 functions to deal with any two variables which have AND or OR gate, besides dealing with one variable which needs to be converted. We rely on declaring a global vector called SPICE which stores any used transistor with its info. Hence, the 5 functions are similar in increasing the SPICE vector size and push back the needed transistors with distinctive information in the drain, gate, or the type. All the wires used are starting from 1 and incrementing while using more wires.

- **The NOT function** is declaring two transistors to make the final target which is converting the input. As we know, to make a logic AND gate, we need to implement two complementary transistors with different types to

take the input through the gate and pass the output through the drain.The output will be the upper case character if the input is lowercase, and vice versa.

- **The AND function** is responsible for taking two input variables and making their transistors in series with each other. There are two scenarios for using the AND function:
    - ANDing two variables: This case will result in two new transistors and connecting them in series with a new wire. Then, using another wire to be the drain of the second transistor and the output of the function.
    - ANDing one variable with a wire: This case happens when ANDing or ORing any two inputs, then ANDing another input with the resulting wire. In this case, one transistor is added to the netlist, since it will take the resulting wire as its source, and generate a new wire in the drain.

- **The OR function** is responsible for taking two input variables and making their transistors in parallel with each other. There are three scenarios for using the OR function (The first two are similar two AND function):
    - ORing two variables: This case will result in two new transistors and connecting them in parallel, so that the drain and source will be the same. Then, generating a new wire to be the drain of the second transistor and the output of the function.
    - ORing one variable with a wire: This case happens when ANDing or ORing any two inputs, then ORing another input with the resulting wire. In this case, one transistor is added to the netlist, since it will take the resulting wire as its drain, and VDD (in case of PMOS) or Ground (in case of NMOS) as its source.
    - ORing two wires with each other: In this case, it’s required to make the two wires connected so that it will be one wire with one name.

Thus, we make a for loop in the netlist two replace one of the two wires by the other one.

There are 3 more functions to handle different cases. The AND & OR functions are duplicated with few adjustments to be used in the pull-down network. In addition, there is a function called “only_not” to handle the case when the expression is just converting one input and making it the final output.

II. Converted-expression generator function

In the Pull-Down network, we have to convert the expression (applying De Morgen rule to the whole expression) to use the proper gates. For achieving this purpose, we will use **the converting function** which take the expression and do the following steps:

- Detecting the input characters and checking if it has a bar sign or not. If it has, then remove it. If it hasn’t, then calling the NOT function to convert it.
- Detecting the AND sign, then converting it to the OR sign.
- Detecting the OR sign, then converting it to the AND sign.

This function will result in a converted expression, but in a different order to execute the gates. For example, if the original expression is g = a & b’ | c
The converted expression will be: g’ = a’ | b & c’
However, The OR gate now should be executed first, then the AND gates. This change will be considered in the Pull-Down Network function.

III. Pull-Up & Pull-Down Network functions

The PUN & PDN functions are the two main functions to control all operations on the expression. They contain the same content except few adjustments:

- The PUN function takes the original input expression, while the PDN function takes the converted one.
- The AND has higher precedence than the OR in the PUN function, while the PDN has the OR as a higher precedence.

The two functions have the following steps:

- Detecting the equal sign and taking the output variable before it in a separated string variable to use it at the end of the function.
- Detecting the inputs, which has no NOT sign to be converted, since we need the converted inputs in the Pull-Up-Network. Calling the NOT function will generate the converted variable (Lower if upper & vice versa)
- Detecting the AND & OR signs and executing each of them separately by calling the corresponding logic gates functions. Then, replacing the part of the expression with the output wire to continue processing. For example, g = a' & b | c → g = a & B | C → g = 2 | C → g = 4
- Finally, using the output variable, which we stored in the first step, to make it appearing in the final netlist instead of the final wire. This functionality is accomplished by **the "connect_out" function.**

Since we have to detect the input characters before and after any detected logic-gate sign, in addition to erasing and inserting in the expression resulting in a change in the expression length, **the "update_operation" function** is used in this case to generate a boolean array with the size of the expression to detect if this character is one of the inputs or one of the operators.

IV. Detecting multiple expressions functions (semi - duplicates)

The "semi" function is used to detect any semi-colon in the input string from the user and use it to split the string to several expressions, storing them into a vector of strings for the expressions. Later on, we are looping on this vector to make the process on each expression separately.

Using multiple expressions leads to duplicate numbers of transistors with the same functionality; for example: g = a & b; h = g' | a

In this example, the system will deal with each expression so that it will have to convert the input "a" from the first expression by two transistors. Although, the system will do the same steps in the second expression which resulted in

repeated two transistors for converting the same input “a”. To solve this problem, we use **the “remove_duplicates” function** to loop on the SPICE netlist, detect the repeated transistors, and make such a flag to distinguish that this transistor is repeated. From here, I will be able to ignore it in **the print function,** which is looping on the SPICE netlist vector and printing its elements, which doesn’t have the deletion flag.

V. Validity & processing function

The validity function is responsible for handling the bugs, and outputting an error message for the invalid input cases. The types of these messages are explained, in detail, in the 1.4 section.

The processing function is responsible for calling the PUN function, converting the original expression, then calling the PDN function. Calling the processing function with giving it any input expression as a parameter is enough for filling the SPICE netlist with all needed transistors.

1.3 Extra Features (Bonus)

- **Allowing the parentheses in the input expression**

This feature will override the known precedence of the logic gates, and execute them according to the parentheses depth. To achieve this target, we take the input expression as a parameter for the parentheses function which do the following:

- Generating an int array with the same size of the expression length, this array will detect any right and left pair of parentheses, and give this pair the same and distinguished rank starting from 1. The deeper pair will have a bigger rank.
- Hence, we are looping on those parentheses and choosing each of them according to the higher rank.
- Then, calling the processing function with taking it the statement inside the parentheses, appended with an output character to be replaced in the updated expression. For example, if the input expression is **g = ((a | b) & (c | d) ’) ’**

→ Then, the (c | d) pair will have the highest rank, so it will be executed first by the processing function which take “Z = c | d” as its parameter. In this case, the variable “Z” will be the output wire from this pair. Therefore, the updated expression will be : **g = ((a | b) & Z ’) ’**

→ Now, moving to the less-rank pair, we will execute the pair (a | b) to give a “Y” variable as a resulting wire. Hence, the updated expression will be : **g = (Y & Z ’)’**

→ Then, we will take the last pair of parentheses to be executed. Therefore, the updated expression will be **g = w’**

→ Finally, we will pass this final expression to the processing function to convert the resulting wire and make it the output “g”.

1.4 User Guide for how to use our program

How to use the program?

- After you run the program, you will be asked to enter the boolean expression that you want to implement using CMOS technology.
- Write the expression while putting in mind the input specifications guidelines.
- Press Enter.
- The Output will be displayed in a form of SPICE netlist describing the connections of the circuit in the following format:

	Mname	drain	gate	source	body	type
Examples:	**M0**	**A**	**a**	**VDD**	**VDD**	**PMOS**
	M1	**A**	**a**	**0**	**0**	**NMOS**

Note: the ground is always referred to as 0 in the output.

Input Specifications:

An error message will appear if the user violated any of the following:

- It's not allowed to enter any spaces in the input characters.
- The symbol of the output shouldn't be used as one of the input symbols.
  Example: y=y&x|z is an invalid input.
- Consistency is required throughout the entered expression!
  Example: You cannot use 'A' and 'a' inside the same expression.
- The input must contain an '=' sign; and at most one sign for each entered expression.
- The input must contain at least one logical operator (&, |, ') in order to be considered as a valid expression.
- While using parentheses, make sure each left parenthesis _i.e. '(' _ has a right parenthesis _i.e. ')' _ to have a valid expression.
  Example: e=(a'&b|(c) is an invalid input.
- If you choose to use parentheses, avoid using the letters from u to z, either in capital or small case, in your expression.
- Variables should be made of single characters.
  Example: y=aa&bb is an invalid input.

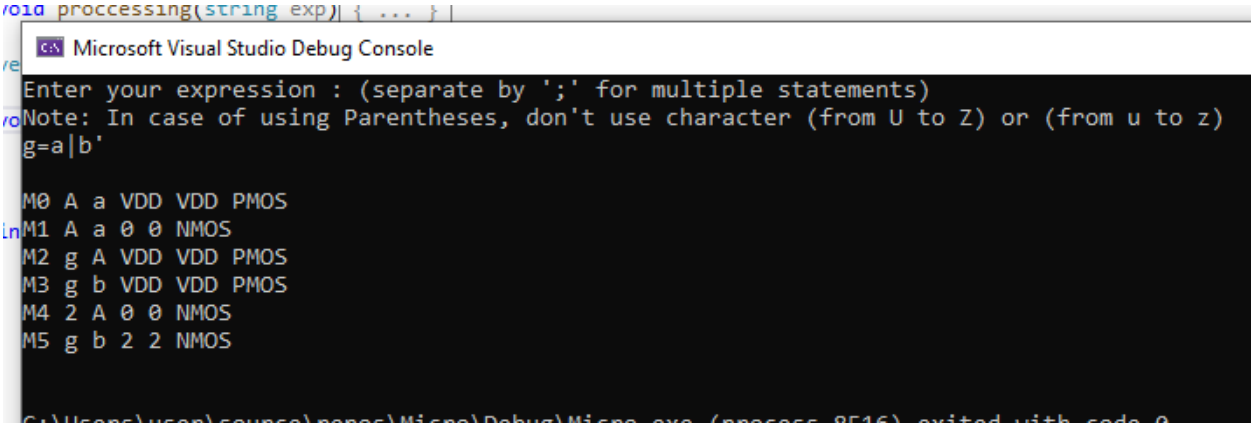
2. TESTING

2.1 Test Cases

We choose three different expressions with several cases, so that we can test all functionalities of the project.

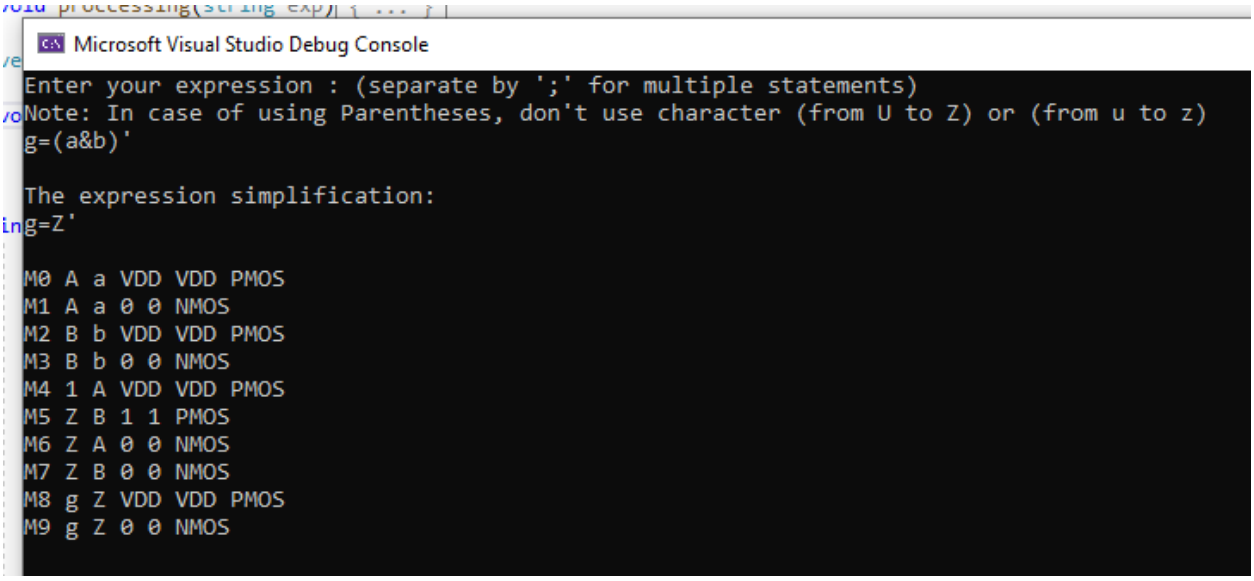
1. **g = a | b'**

   This expression is a simple one, executing the NOT function for the input "b", ORing the two inputs in the Pull-Up Network, then ANDing the two inputs in the converted expression in the Pull-Down Network. Finally, putting the output "g" instead of the resulting wire.



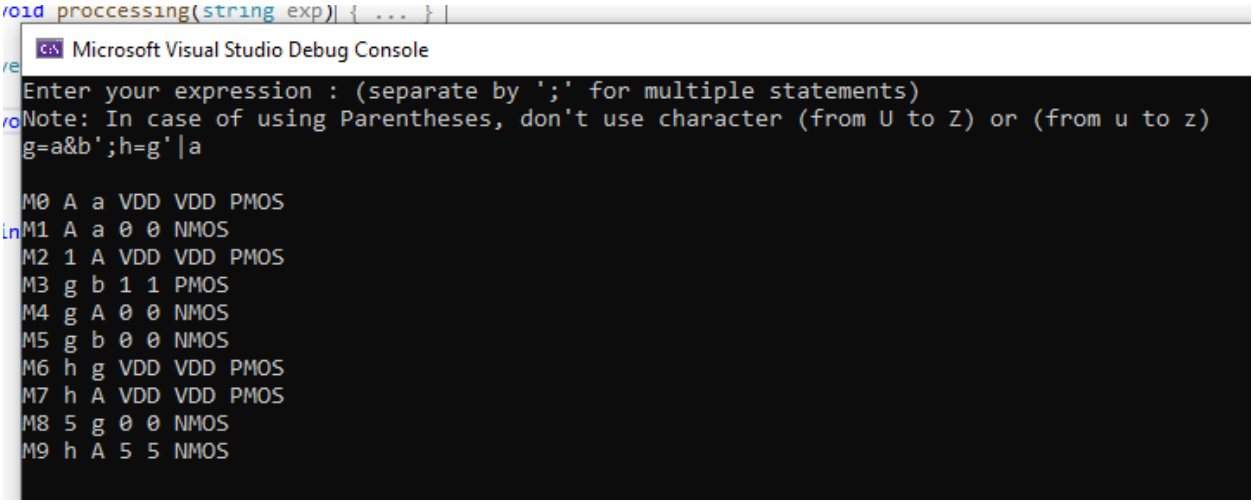
2. **g = (a & b) '**

   This expression will test the parenthesis feature. First, it detects the parentheses and executes the expression inside them, making the resulting wire the variable “Z”. Then, executing the updated expression and putting the output “g” in the final netlist.



3. **g = a & b' ; h = g' | a**

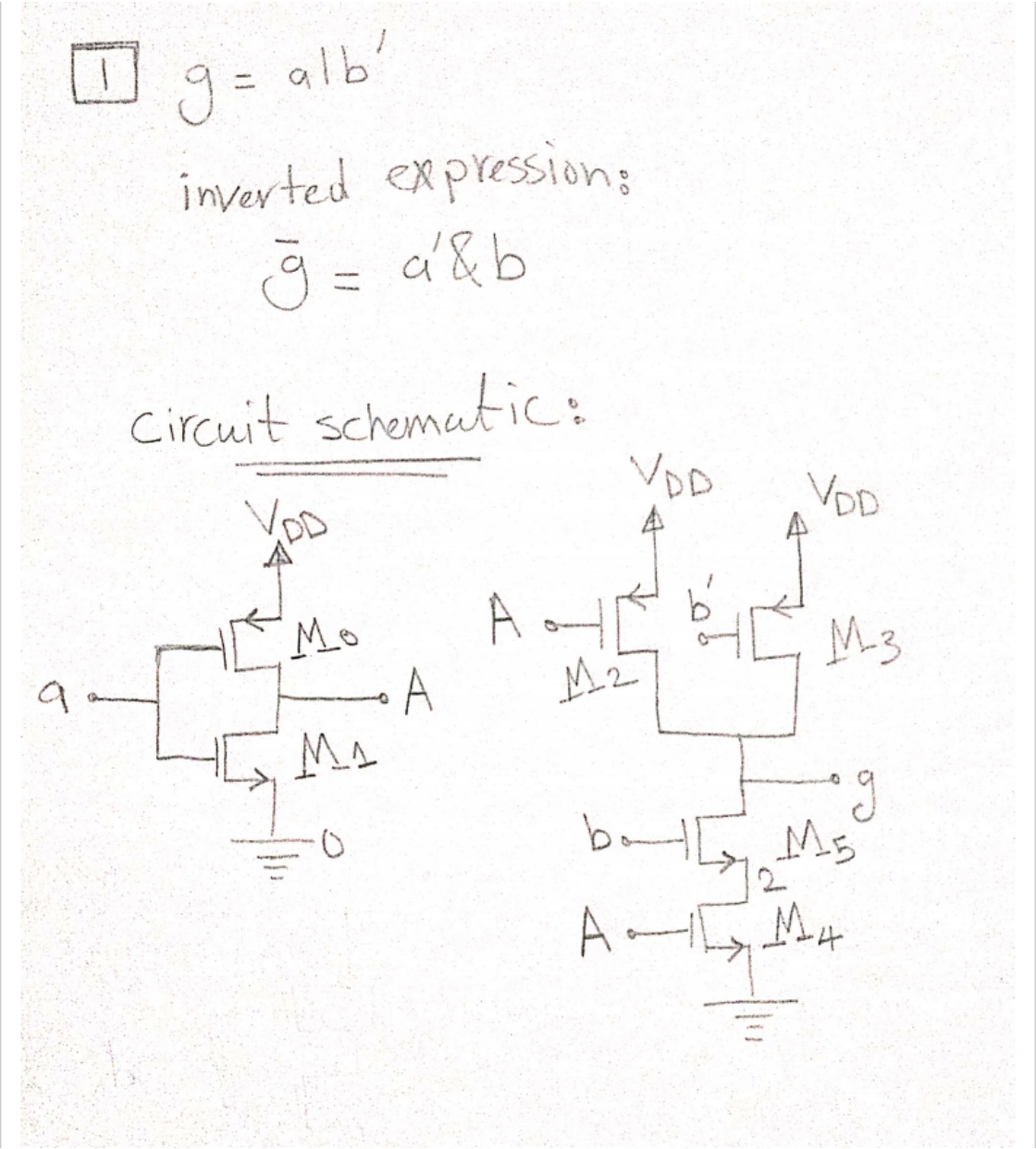
   This expression has multiple statements which test the ability of the user to deal with such a feature. First, the system will execute the first expression separately with the same technique in the first test case. Then, it will deal with the next expression by dealing the input “g” as a wire from the previous expression. Finally, it put the final output “h” in the netlist lines.



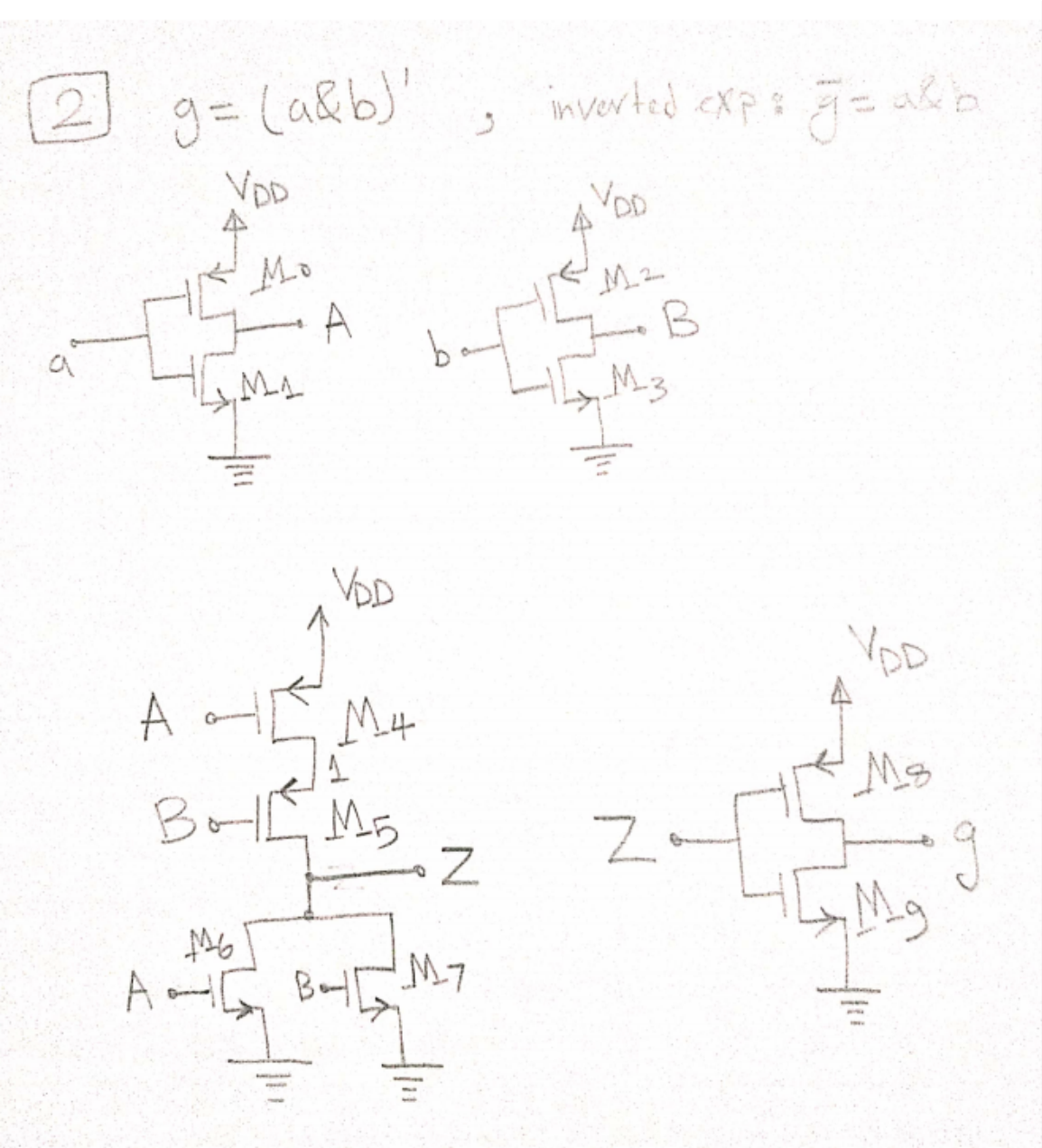
2.2 Schematics of the boolean expression

In this section, we will show the schematic circuit of the three test cases, mentioned above, to explain the netlist lines and wires, beside making sure that the system prints the output correctly.

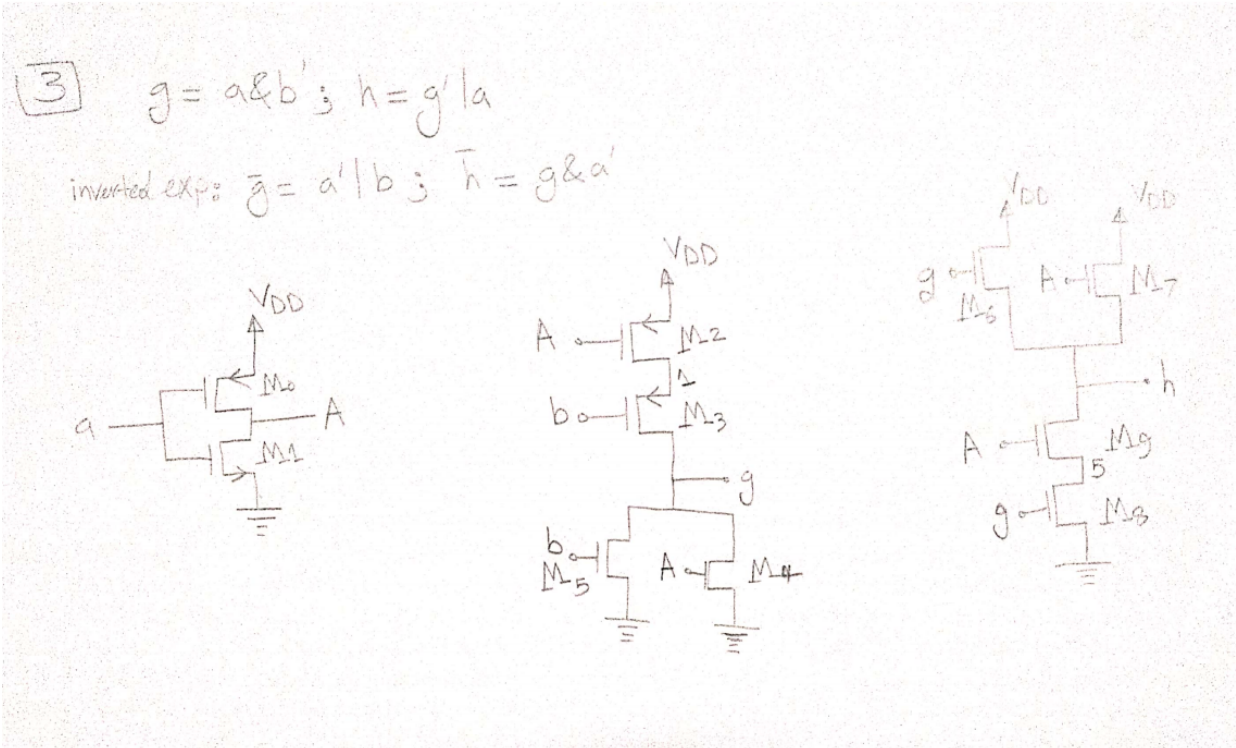
1. **Circuit schematic for the first test case:**



2. Circuit schematic for the second test case:



3. **Circuit schematic for the third test case:**



3. TEAM CONTRIBUTIONS

3.1 Members Contributions

The work we have produced in this project is a result of collaborative efforts exerted by our team; therefore, here we are going to specify which parts each one of us have worked on throughout the journey of this project.

Iman's work:

- Logic gates building functions (AND_PMOS, OR_PMOS, AND_NMOS, OR_NMOS, NOT functions)
- Converted-expression generator function

Ahmed's work:

- Validity & Processing function
- Pull-Up & Pull-Down Network functions
- Detecting multiple expressions functions (semi - duplicates)
- Bonus function

3.2 Reflection on the Experience

Our work on this project added up to our knowledge and experience in many aspects. For example, when we started working on the proposed problem, we found a much easier way to approach the problem, but it wasn't efficient at all. That approach took only a couple of hours to be implemented using C++. Back then, we could have just submitted that approach anyways; however, as computer engineering students, we give due care to efficiency and complexity, so we decided to find another approach which we have presented in this report. This taught us resilience and made us realize that it does not matter if you solve a problem, what matters is how you solve it.

Furthermore, in this project we were challenged but at the same time motivated by the bonus features requested in the project document. We already have started working on two of them: the parentheses feature, and the website. But due to the time limitation, we successfully managed to complete the first and managed to complete a great deal of the second one but not in a complete way as the website we made still lacks the final touches. Working on the website was a bit of a challenge and a bit of fun as it made us search into new techniques and read about HTML and several python libraries that was a great asist. Thus, we are planning to finalize the website and publish it in order to give the chance for people everywhere to benefit from our program.

P.S.The code of the first approach is attached in the appendix section if you are interested to take a look at it.

Appendix.

```
#include <iostream>

#include <string>

#include <vector>


using namespace std;


vector<vector<string>> SPICE(0);

int wires = 1;

int gates_out = 0;


void NOT(string in) {


	int idx = SPICE.size();

	SPICE.resize(SPICE.size() + 2);

	for (int i = idx; i < SPICE.size(); i++)

		SPICE[i].resize(6);


	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = in; SPICE[idx][3] = "VDD"; SPICE[idx][4] = "VDD"; SPICE[idx][5] = "PMOS";

	idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = in; SPICE[idx][3] = "0"; SPICE[idx][4] = "0"; SPICE[idx][5] = "NMOS";

	wires++;

	gates_out++;


}


void AND(string in1, string in2) {
```

```
	int idx = SPICE.size();

	SPICE.resize(SPICE.size() + 6);

	for (int i = idx; i < SPICE.size(); i++)

		SPICE[i].resize(6);


	// M - drain - gate - source - body - type


	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = in2; SPICE[idx][3] = "VDD"; SPICE[idx][4] = "VDD"; SPICE[idx][5] = "PMOS";

	idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = in1; SPICE[idx][3] = "VDD"; SPICE[idx][4] = "VDD"; SPICE[idx][5] = "PMOS";

	idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = in1; SPICE[idx][3] = to_string(wires + 1); SPICE[idx][4] = to_string(wires + 1); SPICE[idx][5] = "NMOS";

	idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires + 1); SPICE[idx][2] = in2; SPICE[idx][3] = "0"; SPICE[idx][4] = "0"; SPICE[idx][5] = "NMOS";

	idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires + 2); SPICE[idx][2] = to_string(wires); SPICE[idx][3] = "VDD"; SPICE[idx][4] = "VDD"; SPICE[idx][5] = "PMOS";

	idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires + 2); SPICE[idx][2] = to_string(wires); SPICE[idx][3] = "0"; SPICE[idx][4] = "0"; SPICE[idx][5] = "NMOS";


	wires = wires + 3;

	gates_out++;


}
```

```
void OR(string a, string b) {

	int idx = SPICE.size();

	SPICE.resize(SPICE.size() + 6);

	for (int i = idx; i < SPICE.size(); i++)

		SPICE[i].resize(6);


	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = a; SPICE[idx][3] = "0"; SPICE[idx][4] = "0"; SPICE[idx][5] = "NMOS"; idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = b; SPICE[idx][3] = to_string(++wires); SPICE[idx][4] = to_string(wires); SPICE[idx][5] = "PMOS"; idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires); SPICE[idx][2] = a; SPICE[idx][3] = "VDD"; SPICE[idx][4] = "VDD"; SPICE[idx][5] = "PMOS"; idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(--wires); SPICE[idx][2] = b; SPICE[idx][3] = "0"; SPICE[idx][4] = "0"; SPICE[idx][5] = "NMOS"; idx++;


	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires + 2); SPICE[idx][2] = to_string(wires); SPICE[idx][3] = "VDD"; SPICE[idx][4] = "VDD"; SPICE[idx][5] = "PMOS"; idx++;

	SPICE[idx][0] = "M" + to_string(idx); SPICE[idx][1] = to_string(wires + 2); SPICE[idx][2] = to_string(wires); SPICE[idx][3] = "0"; SPICE[idx][4] = "0"; SPICE[idx][5] = "NMOS"; idx++;

	wires = wires + 3;

	gates_out++;

}

void update_operation(vector <bool>& operation, string bool_exp) {

	operation.resize(bool_exp.length());

	for (int i = 0; i < bool_exp.length(); i++) {

		if (bool_exp[i] == '&' || bool_exp[i] == '|' || bool_exp[i] == '\'')

			operation[i] = true;

		else

			operation[i] = false;

	}
```

```
}
void taking_input() {
	string bool_exp;
	cout << "Enter the expression, please!\n";
	cin >> bool_exp;
	string out;
	int x = bool_exp.find('=');
	out.assign(bool_exp, 0, x);
	bool_exp.erase(0, x+1);
	vector <bool> operation(bool_exp.size(), false);
	while (bool_exp.find('&') <= bool_exp.length() || bool_exp.find('|') <= bool_exp.length() ||
bool_exp.find('\'') <= bool_exp.length()) {
		int idx;
		int start = INT_MAX;
		int end = INT_MAX;

		if (bool_exp.find('\'') <= bool_exp.length()) {
			update_operation(operation, bool_exp);
			idx = bool_exp.find('\'');
			for (int i = idx - 1; i >= 0; i--) {
				if (operation[i]) {
					start = i + 1;
					break;
				}
			}
			if (start == INT_MAX)
				start = 0;
			string str; str.assign(bool_exp, start, idx - start);
			NOT(str);
```

```
            bool_exp.replace(start, idx - start + 1, to_string(gates_out));
            start = INT_MAX;
            cout << "The statement now: " << bool_exp << endl;
    }

    if (bool_exp.find('&') <= bool_exp.length()) {
            update_operation(operation, bool_exp);
            idx = bool_exp.find('&');
            for (int i = idx - 1; i >= 0; i--) {
                    if (operation[i]) {
                            start = i + 1;
                            break;
                    }
            }
            if (start == INT_MAX)
                    start = 0;
            string str1; str1.assign(bool_exp, start, idx - start);


            for (int i = idx + 1; i < bool_exp.length(); i++) {
                    if (operation[i]) {
                            end = i;
                            break;
                    }
            }
            if (end == INT_MAX)
                    end = bool_exp.length();
            string str2; str2.assign(bool_exp, idx + 1, end - idx - 1);
```

```
            AND(str1, str2);
            bool_exp.replace(start, end - start, to_string(gates_out));
            start = INT_MAX;
            end = INT_MAX;
            cout << "The statement now: " << bool_exp << endl;
    }
    if (bool_exp.find('|') <= bool_exp.length()) {
            update_operation(operation, bool_exp);
            idx = bool_exp.find('|');

            for (int i = idx - 1; i >= 0; i--) {
                    if (operation[i]) {
                            start = i + 1;
                            break;
                    }
            }
            if (start == INT_MAX)
                    start = 0;
            string str1; str1.assign(bool_exp, start, idx - start);

            for (int i = idx + 1; i < bool_exp.length(); i++) {
                    if (operation[i]) {
                            end = i;
                            break;
                    }
            }
            if (end == INT_MAX)
                    end = bool_exp.length();
            string str2; str2.assign(bool_exp, idx + 1, end - idx - 1);
```

```
                    OR(str1, str2);
                    bool_exp.replace(start, end - start, to_string(gates_out));
                    start = INT_MAX;
                    end = INT_MAX;
                    cout << "The statement now: " << bool_exp << endl;
                }
        }

        for (int i = SPICE.size() - 2; i < SPICE.size(); i++)
                SPICE[i][1] = out;
}

int main() {
        taking_input();
        for (int i = 0; i < SPICE.size(); i++) {
                for (int j = 0; j < SPICE[i].size(); j++)
                        cout << SPICE[i][j] << " ";
                cout << endl;
        }
        return 0;
}
```