Logic Simulation

Ismaiel Sabet 900221277
Seif ElAnsari 900221511
Noor Emam 900222081

Table of Contents:

Abstract
As described in the project file, the goal of this project is to implement a simulator for the logic circuits. This principle is based upon the idea of events, and as such, our simulator is an event driven simulator. The definition of an event is a change of a particular input, and that change in of itself takes time to happen. What follows is a ripple effect, as the circuit is evaluated based on the change of this particular input. We also have that many inputs could be changed sequentially or in parallel.

For example, if we take our circuit to be A+B, then the event that changes A from 0 to 1 changes A+B from 0 to 1. The time it takes for that to quantify is the sum of the time it took to change the input and the propagation delay of the OR gate.

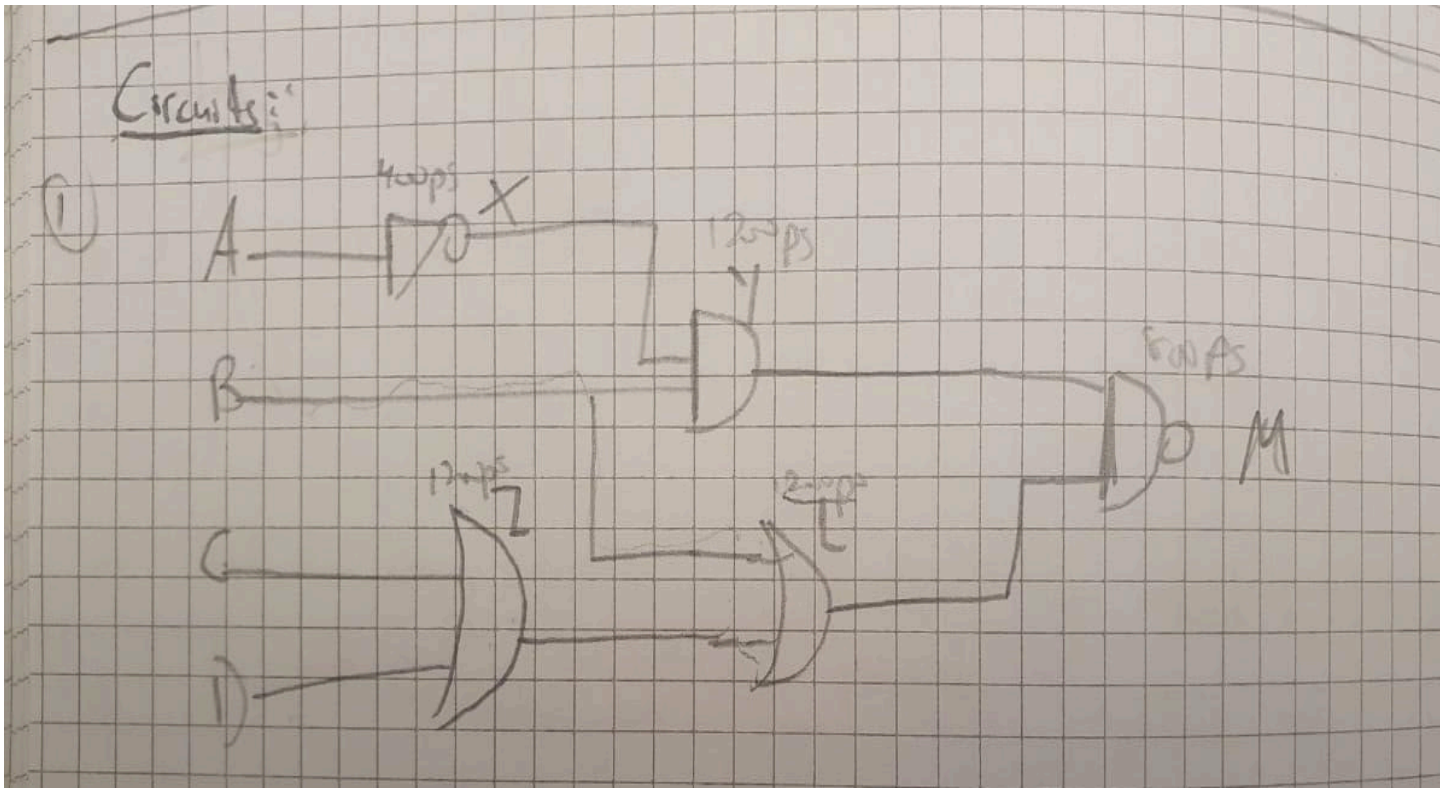
The goal of the project is to simulate the behaviors of logical circuits through events and to get the total propagation delay, making sure that each event takes time to take place and each gate has its own propagation delay.

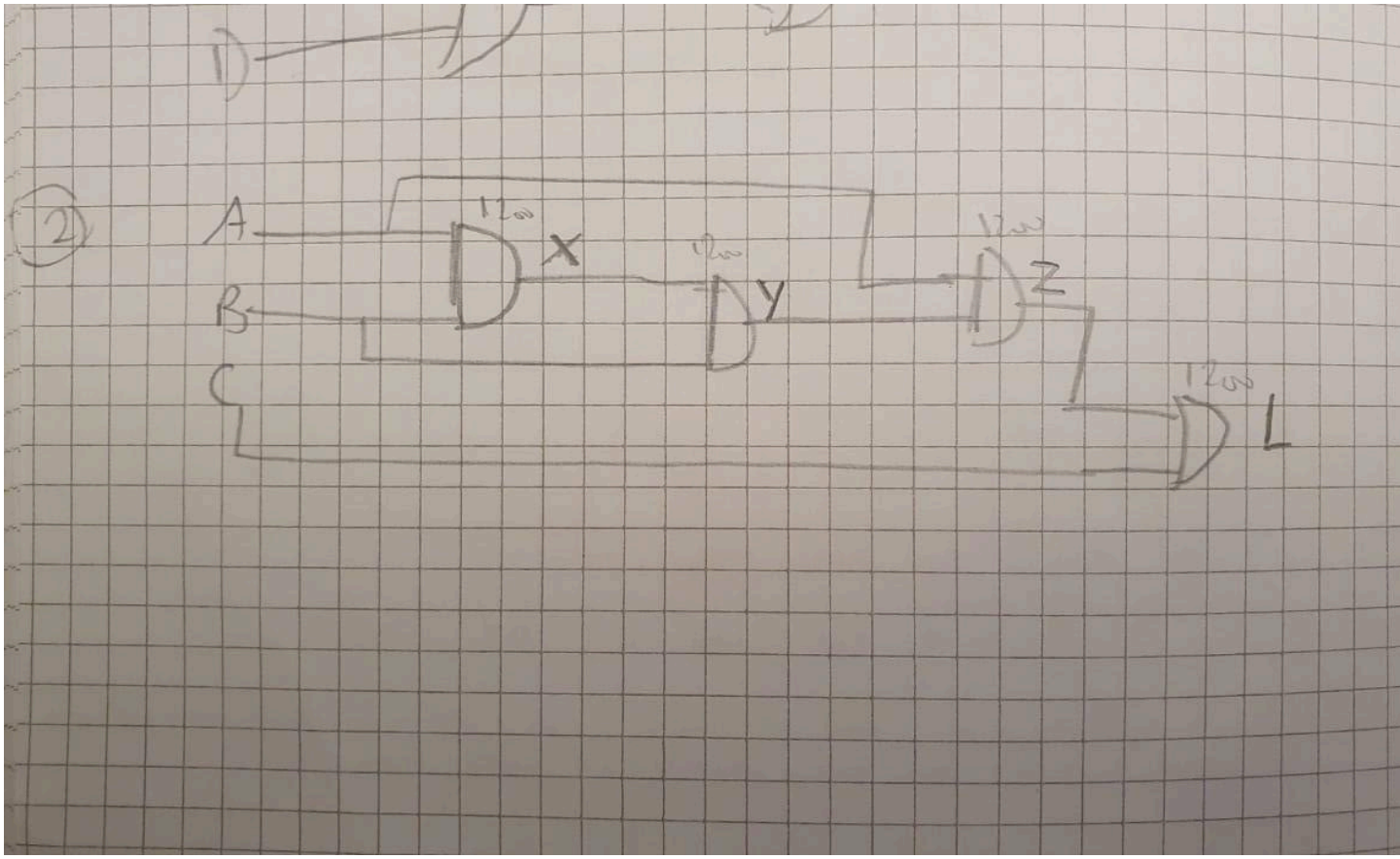
Design Process and Mindset

The design process of this project began with the research elements at the beginning, before a single line of code was written. This involved the following:

1) Realizing what was required of us
2) Understanding the 3 input files and their respective roles in the overall project.
3) Making a rough sketch of what to do

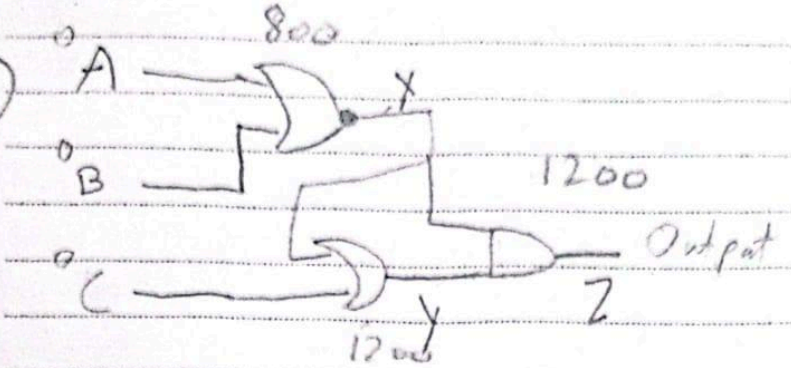
The first thing we did was develop 5 arbitrary test circuits. We tried to the best of our abilities to include the 6 major logic gates (AND, OR, NOT, NOR, NAND, XOR). We also defined propagation delays for each of the gates and used the same standard over the 5 different circuits. Below are the 5 circuits designed:



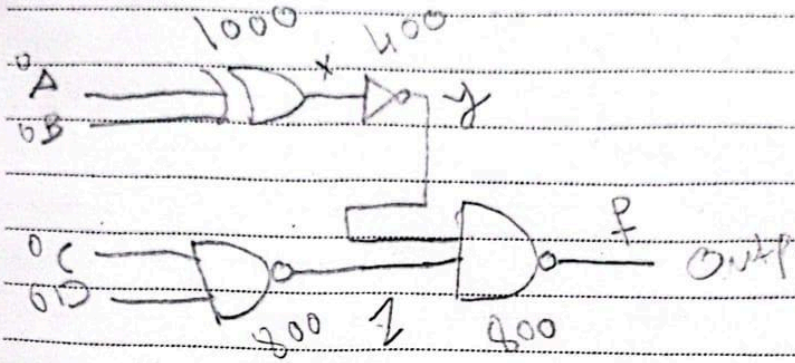

1)
2)
A
B
C
X
Y
Z
L

A

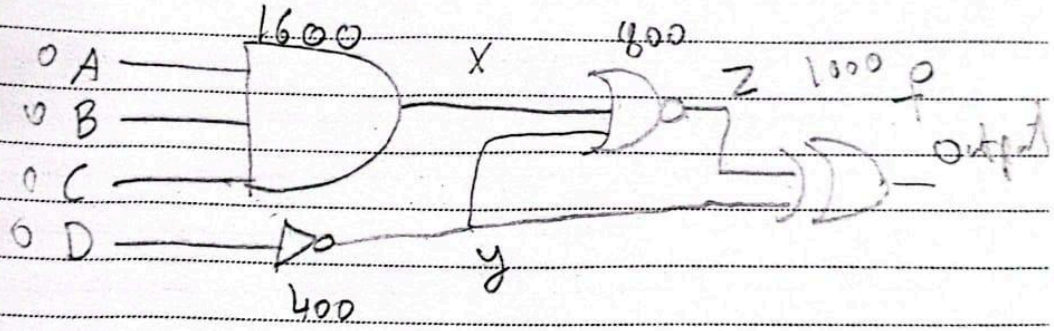
3)


800
A
X
B
1200
Output
C
Y
Z
1200

4)

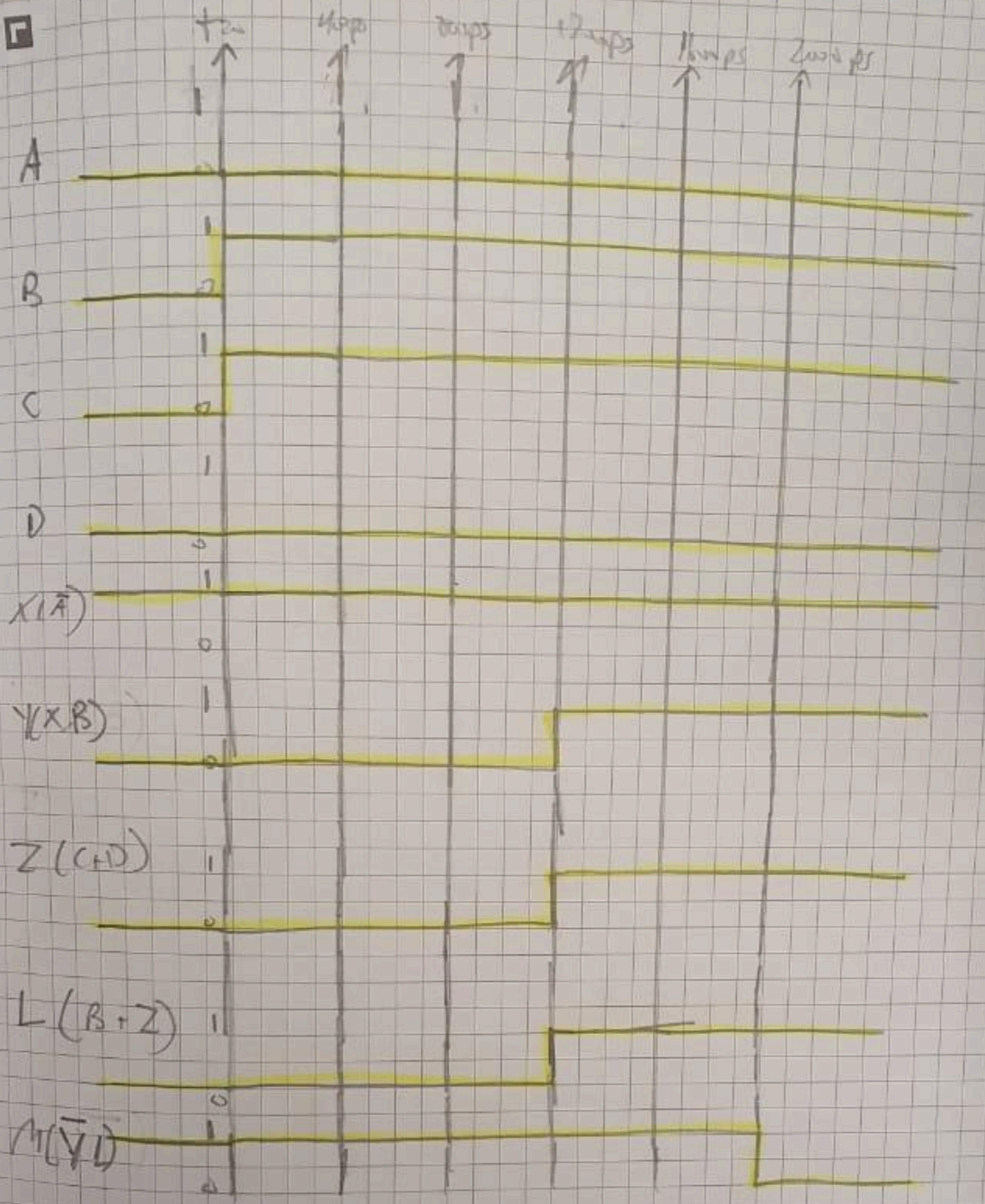

1000
400
A
X
Y
B
P
C
Output
D
800
2
800

5)


1600
800
A
X
Z
1000
P
B
Output
C
D
y
400

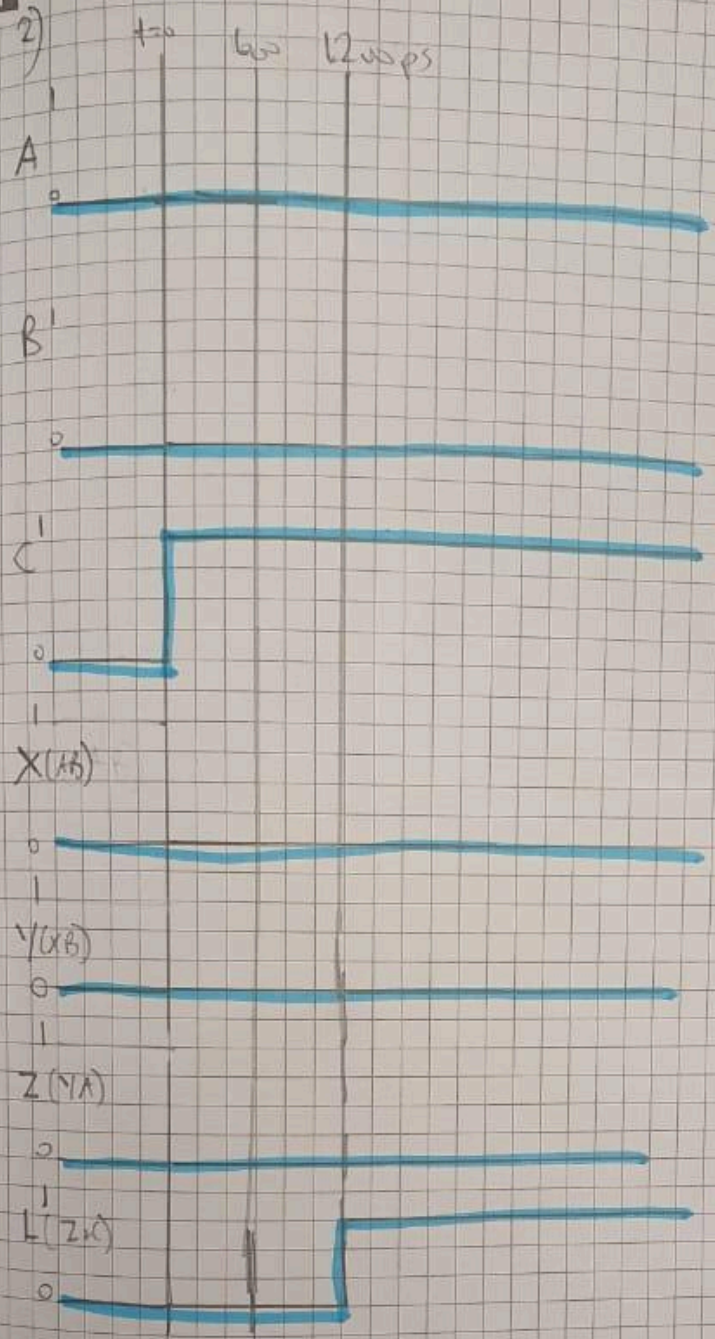
Next, we researched how to sketch the propagation delay. We opted not to use the actual representation of the delay which entailed curved lines and fluctuations (which is more accurate to an electronic circuit). Instead, we're using an ideal behavior, with sudden and sharp rises from 0 to 1 and vice versa. We also took our time axis to start from 0, and everything before we begin the simulation (before the 0) is initialized based on the input values which are all 0's. So for instance, if we have A and NOT A, then before we simulate A is 0 and NOT A is 1. We then move by a scale to keep things mathematically clear, and as such, the delay will be clearer than ever. Now in order to actually begin the simulation, we apply an event, which could change the value of 1 or multiple variables. Then, we begin to measure. It must also be mentioned that for these propagation delay diagrams, as it was relatively early in the design process, we did not anticipate that it would take time for the inputs to change. So, the change happens at 0ps, which is indeed a specific time but a unique one. But in general, we begin the simulation at time < 0. Nevertheless, below are the 5 propagation delays for the 5 circuits:

1)



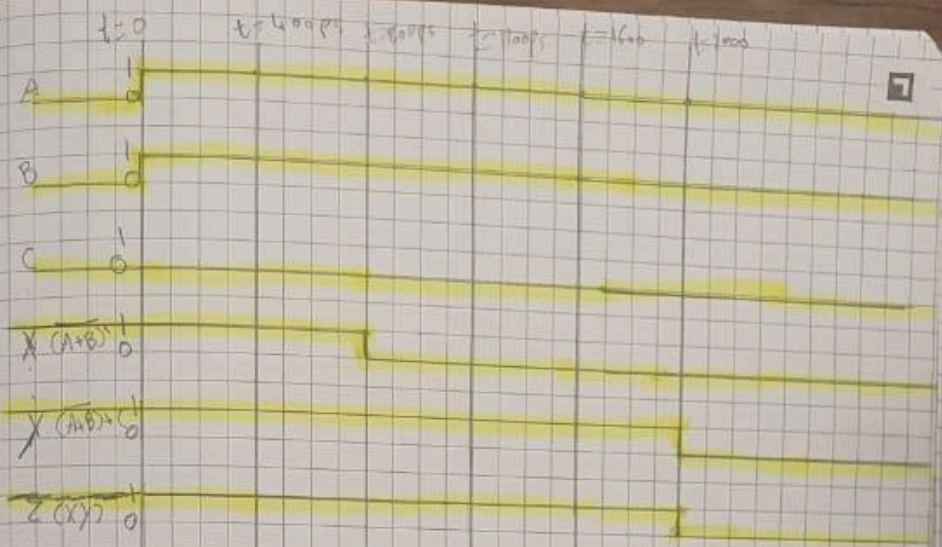
total propagation delay: 2000 ps

2)



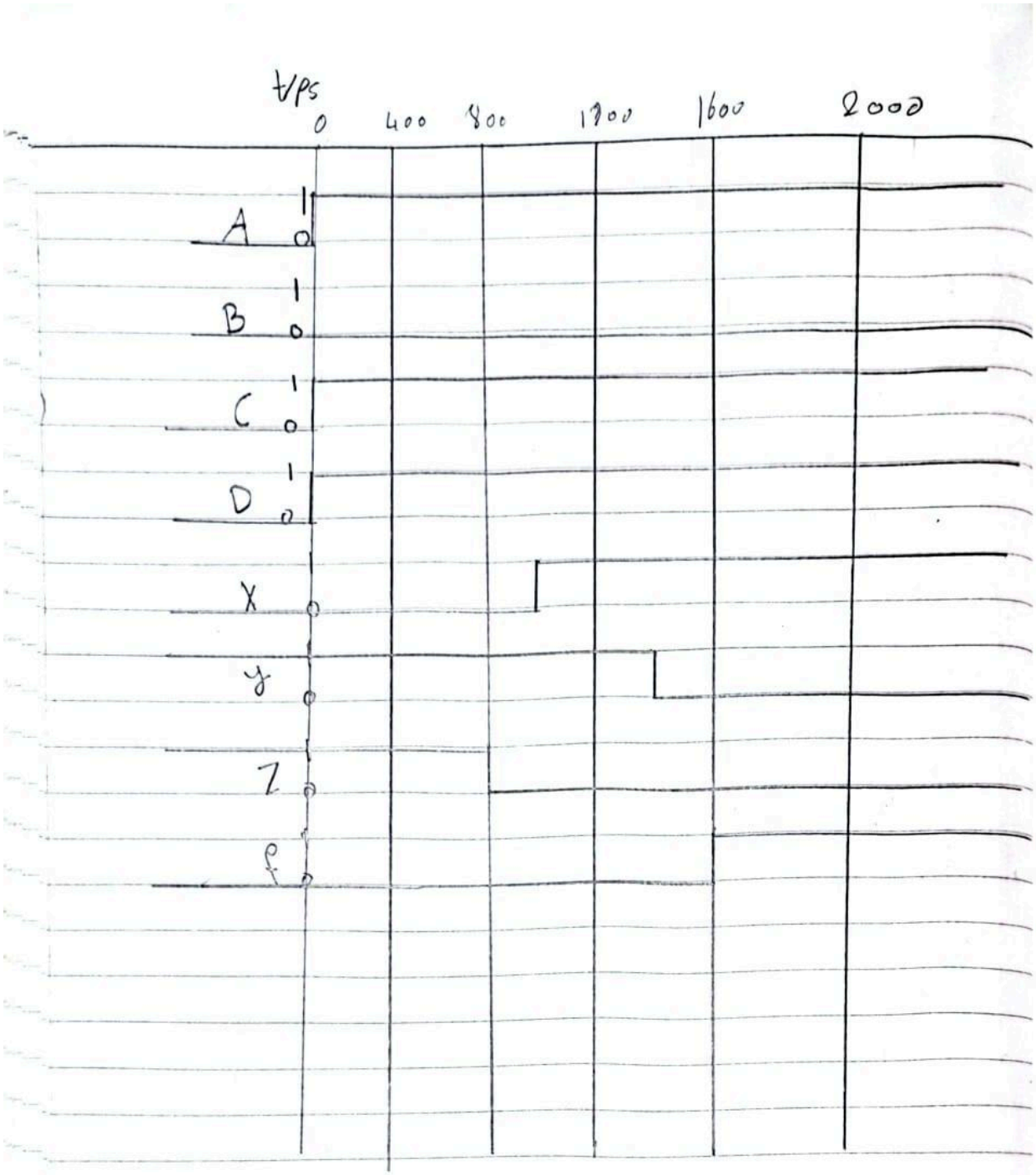
Total propagation delay = 1200 ps

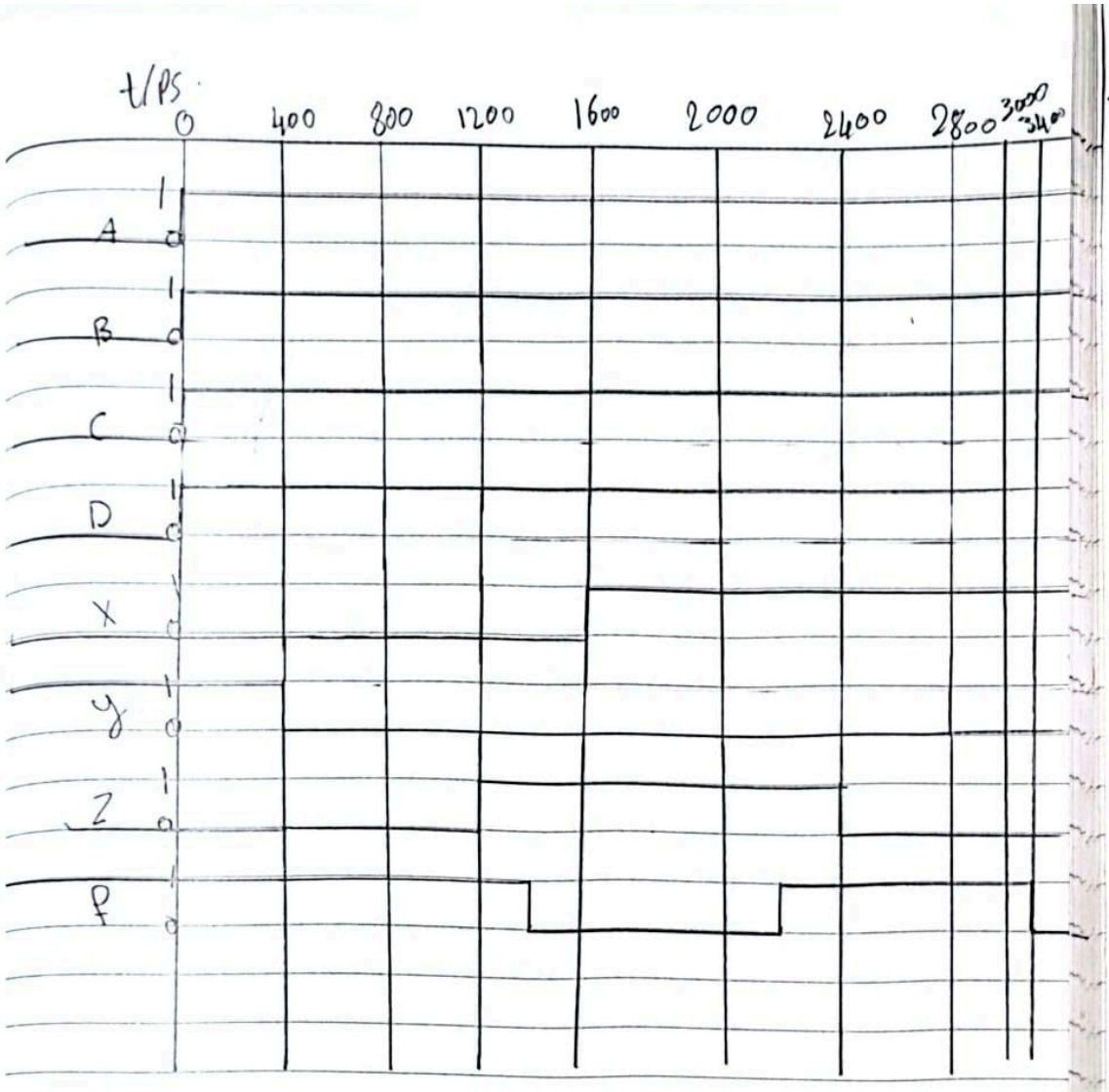
3)



Total propagation delay

2000ps


t/ps
0
400
800
1200
1600
2000
A
B
C
D
X
y
Z
f
1
0


t/ps
0
400
800
1200
1600
2000
2400
2800
3000
3400
A
B
C
D
X
y
Z
f
1
0

After that, we began to implement the project and we decided to use C++ as our language as it was the one we are all familiar with. We began by designing the 3 main classes - LogicGates, BoolVar, and Components - and implemented a function that optimizes all 3 input files - .circ , .lib, and .stim - by removing the spaces to unify all inputs.

With the classes complete, we researched how to make the user input any file. We began exploring an approach that however was not very consistent, meaning that it sometimes worked and sometimes not. We opted to then switch the coding philosophy, with our aim pointed at making something more consistent and also user-friendly. This is where we decided to implement our code in QT. This allowed for the very user-friendly user display that is shown in the output, and also unified everything for us, fixing 2 problems with 1 decision. Moving forward, the rest of the code was written as in normal C++, but making use of QT to input any file.

Following that, we implemented the stimuli file, which previously was not done in the beginning coding phase.

After that, we began considering the simulation and how it would work. To accomplish this, we carefully constructed an algorithm that worked towards that. The algorithm was not in C++, but it allowed us to map out the algorithm, and the mapped out algorithm was then translated to C++ in the following steps.

Then, we translated the algorithm to C++ code. To do this, we made use of the mathematical Post-fix principle to turn the functionality of the logic gate to an expression, and then, we converted that expression to boolean logic by making use of the precedence of operators and characters over others. We also effectively managed to get the proper time.

With the general plan revealed, let's now insert snippets from the code and explain how everything works.

CODE SNIPPETS:

```
class LogicGates //Logic Gates class based on the provided description in the project file
{
public:
    string component_name; //name of the component (AND, OR, NOT, NAND, NOR, OR XOR)
    int inputs; //how many inputs it takes
    string functionality; //what the gate does (the boolean expression)
    int delayps; //the delay of the gate expressed in ps
};

class BoolVar //class used to describe the BoolVar (used for both inputs and outputs)
{
public:
    string name; //name of variable
    bool value; //value of variable
    int currtime = 0; //the time of each variable (needed in the simulation)
};

class Components //class used to describe circuit components
{
public:
    string component_name; //the name of the component
    LogicGates gate; //the gate used to implement this particular operation
    vector<BoolVar*> inputs; //a vector of inputs (so we can change the size if another input is added/removed)
    BoolVar* output; //the output variable

};

class stimulus //class for the stimulus file
{
public:
    int time_stamp_ps; //time at which event begins
    BoolVar* input; //the variable that changes (instantiated using pointer of BoolVar)
    bool new_value; //the new value to be assigned to the variable

};
```

The above snippet contains the declarations of the 4 classes used in the project. The first is LogicGates. The LogicGates class has public variables component_name (the name of the gate being used), the number of inputs, the functionality of the gate, and the delay time of the gate. The number of inputs of the gate also makes an appearance in the name of the gate (for example AND2 or AND3). The second class is BoolVar. This class represents all the boolean variables in the code, whether input or output. Each instance of BoolVar has a name and a value, in addition to the time which will be used in the simulation. The Components class contains the name of the component, an instance of LogicGate, a vector of BoolVar pointer (as we can have many inputs and we want the changes to be reflected in all other instantiations), and similarly a BoolVar pointer output, to store and reflect the output across all instantiations. Finally, we have the stimulus file, which contains the timestamp at which the event takes place, a BoolVar pointer reflecting the variable whose value is to be changed, and the new value to be stored in that variable.

```
void ReadLibrary(vector<LogicGates*>& gates, QString path) //function that reads the Lib file
{
    ifstream inputFile(path.toStdString()); //reading the file

    if (inputFile.is_open()) //if successfully opened
    {
        string line;
        while (getline(inputFile, line, ',')) //while there are still lines
        {
            LogicGates* gate = new LogicGates(); //declare a gate
            gate->component_name = line; //gate component is inserted
            getline(inputFile, line, ','); //read the next part of the line
            gate->inputs = stoi(line); //number of inputs is inserted
            getline(inputFile, line, ','); //read the next part of the line
            line = fileOptimizer(line);
            gate->functionality = line; //the functionality is inserted
            getline(inputFile, line, '\n'); //read the next part
            gate->delayps = stoi(line); //add the delay component
            gates.push_back(gate); //push back in gates vectors
        }
        inputFile.close(); //close the file
    }
    else //if file wasn't opened
    {
        cout << "Unable to open file";
        exit(0);
    }
}
```

The above picture contains the function that reads the .lib file. The parameters of the function are vector of LogicGates pointer passed by reference from the main (this is the vector to which we will store the data) and a file path for reading the data from. We loop through the entire file through the getline function, meaning we continue to loop as long as we still have lines to explore. We then read the data and store it in its respective location, and then read the following piece of information. We make use of the stoi function that changes from string to integer when we see the number of inputs and the delay time. We also call a function called fileOptimizer, which will be explained later on. In general, this function takes in a .lib file, and stores its information in a vector.

```
void ReadCircuit(vector<LogicGates*>& gates ,vector<Components*>& components, vector<BoolVar*>& inputs, QString path)
{
    ifstream inputFile(path.toStdString()); //reading the circuit file
    string line;
    bool found = false;

    if (inputFile.is_open()) //if file opened
    {
        getline(inputFile, line); //read the line
        if (line == "INPUTS:") //if we're in the INPUTS section
        {
            while (getline(inputFile, line) && line != "COMPONENTS:") //while there are still inputs AND we haven't reached COMPONENTS
            {
                BoolVar* input = new BoolVar(); //create input instance
                input->name = line; //add the name
                input->value = false; //add the value
                inputs.push_back(input); //add to number of inputs vector
            }

        }


        if (line == "COMPONENTS:") //if we're in the components section
        {
            while (getline(inputFile, line, ','))
            {
                Components* component = new Components();
                component->component_name = line; //adding components name
                getline(inputFile, line, ',');
                line = fileOptimizer(line);
                for (int i = 0; i < gates.size(); i++) //opening the vector gates
                {
                    if (gates[i]->component_name == line) //if the component name matches what's in the file
                    {
                        component->gate = *gates[i]; //add it to the logic gate part of component
                    }
                }
                getline(inputFile, line, ','); //move to the next part
                line = fileOptimizer(line);
                component->output = new BoolVar();
                component->output->name = line; //add the output names to component
                component->output->value = false;
```

```
            inputs.push_back(component->output);
            for (int i = 0; i < component->gate.inputs; i++) //checking if inputs is repeated or no
            {
                if(i != component->gate.inputs - 1)
                {
                    getline(inputFile, line, ','); //move to the next part
                    line = fileOptimizer(line);
                }
                else
                {
                    getline(inputFile, line, '\n');
                    line = fileOptimizer(line);
                }

                found = false; //assuming no repetitions
                for (int j = 0; j < inputs.size(); j++) // looping over the number of inputs
                {
                    if (inputs[j]->name == line) //if inputs match
                    {
                        component->inputs.push_back(inputs[j]); //push the input
                        found = true; //it has been found!
                        break; //exit for loop
                    }

                }


                if (found == false)
                {
                    BoolVar* input = new BoolVar(); //create a new input
                    line = fileOptimizer(line) ;
                    input->name = line; //puts its name
                    input->value = false; //puts its value
                    component->inputs.push_back(input); //push back

                }


            }
            components.push_back(component); //push back the component

        }
    }
}
else
{
    cout << "Unable to open file";
    exit(0);
}

}
```

The above 2 pictures document the ReadCircuit function, which reads information from the .circ file. This information is read into 2 different arrays: one of Components pointers, and one of BoolVar pointers. The third array of type LogicGates is used in the insertion into Components. We begin by reading a line and checking its beginnings. If it starts with INPUTS, then we read the inputs. If it starts with COMPONENTS, then we read the components. As exemplified in the code, reading the inputs is much simpler than reading the components as the inputs are just made

up of BoolVars, while the components has the name of the component, the name of the gate, its outputs, and its inputs.

```
void ReadStimulus(vector <stimulus*> &stimuli, vector<BoolVar*>& Inputs, QString path)
{
    ifstream inputFile(path.toStdString()); //reading the file

    if (inputFile.is_open()) //if successfully opened
    {
        string line;
        while (getline(inputFile, line, ',')) //while there are still lines
        {
            stimulus* stimuluss = new stimulus(); //declare a gate
            stimuluss->time_stamp_ps=stoi(line); //gate component is inserted
            getline(inputFile, line, ','); //read the next part of the line
            line=fileOptimizer(line); //number of inputs is inserted
            for(int i = 0; i < Inputs.size(); i++)
            {
                if(line == Inputs[i]->name)
                {
                    stimuluss->input = Inputs[i];
                }
            }
            getline(inputFile,line,'\n');
            stimuluss->new_value=stoi(line); //the functionality is inserted
            SortedStimuli(stimuluss,stimuli);
            //stimuli.push_back(stimulus);
        }
        inputFile.close(); //close the file
    }
    else //if file wasn't opened
    {
        cout << "Unable to open file";
        exit(0);
    }
    for(int i = 0; i < stimuli.size();i++)
        cout << stimuli[i]->input << " " << stimuli[i]->time_stamp_ps << endl;
}
```

The image above captures ReadStimulus, which is a function that reads the .stim file. The information is added in the stimuli vector. Stimuluss is used for the storage of each individual event. The SortedStimuli function, explained later on, is a sorting function.

```
string fileOptimizer(string text){ //function that optimizes the file and generalizes it by removing spaces(takes in a string)

    string updated; //the string to be returned
    stack<char> s1; //first stack (push string inside it)
    stack <char> s2; //second stack (add every character that isn't a space)
    int n = 0; //varibale for length of text
    char c; //to push in stack
    while(n != text.length()){ //if the text is not done
        c = text[n]; //read the character
        s1.push(c); //push in the first stack
        n++; //go to next character
    }
    while(!s1.empty()){ //while there are still elements in the stack
        if(s1.top() != ' '){ //if the top character is not a space
            s2.push(s1.top()); //add it to second stack
        }
        s1.pop(); //go to next element in stack
    }
//text is now inverted in stack2. We need to bring it back
    while(!s2.empty()){ //while there are still elements
       updated += s2.top(); //write to string
        s2.pop(); //go to next element in stack
    }

    return updated; //now a string without spaces
}
```

The above function, fileOptimizer, takes in a text and optimizes it by removing any space. To accomplish this, we have 2 stacks. We push all characters into the first stack. Next, we push only the characters that are not space into the second stack (the characters are returned to their proper order once again). Finally, we pop from that second stack, and push into a string that now has no spaces.

```
void FileErrorHandling(QString path) //function that handles error
{
    if(path.isEmpty())
    {
        QMessageBox::critical(nullptr, "error", "Empty File"); //if user did not select a file at the beginning
        exit(0);
    }
}
```

This is a simple function that handles the case if the user did not select a file to be read from (whether .circ, .lib, .sim, or .stim) when the prompt is displayed.

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QMessageBox::information(nullptr, "Information", "Choose a .lib file");
    QString filePath = QFileDialog::getOpenFileName(nullptr, "Select a File", "", "Lib Files (*.lib);;All Files (*)");
    FileErrorHandling(filePath);
    QMessageBox::information(nullptr, "Information", "Choose a .cir file");
    QString filePath2 = QFileDialog::getOpenFileName(nullptr, "Select a File", "", "Circuit Files (*.cir);;All Files (*)");
    FileErrorHandling(filePath2);
    QMessageBox::information(nullptr, "Information", "Choose a .stim file");
    QString filePath3= QFileDialog::getOpenFileName(nullptr, "Select a File", "", "Stim Files (*.stim);;All Files (*)");
    FileErrorHandling(filePath3);
    QMessageBox::information(nullptr, "Information", "Choose a .sim file");
    QString filePath4= QFileDialog::getOpenFileName(nullptr, "Select a File", "", "Sim Files (*.sim);;All Files (*)");
    FileErrorHandling(filePath4);
    vector<LogicGates*> gates; //create instance of LogicGates
    vector<BoolVar*> inputs; //create instance of BoolVar
    vector<Components*> components; //create instance of Components
    vector<stimulus*> stimuli;
    vector<BoolVar> SortedOutput;
    ReadLibrary(gates, filePath); //read the library file and write into the gates vector
    ReadCircuit(gates, components, inputs, filePath2); //read the circuit file and write into components and inputs vectors
    ReadStimulus(stimuli,inputs,filePath3);
    Simulation(stimuli,components,inputs, SortedOutput);
    PrintInSim(filePath4, SortedOutput);
    DrawTimeGraphs(SortedOutput);
    for(int i= 0; i<SortedOutput.size(); i++)
    {
       cout << SortedOutput[i].currtime << ", " << SortedOutput[i].name << ", " << SortedOutput[i].value << endl;
    }
    return a.exec();
}
```

The above photo is just the main function of our project. Notice the 4 prompts asking the user to select 4 different files and how the extension of the files are incorporated into the prompts (meaning it's not possible to select a .circ file when the prompt is to pick a .lib file and so forth). Also, all of the parameters that were present in the previously discussed functions all find their declarations here, like for example gates and components. The logic of the remaining part of the main involves reading through the .lib, .circ, and .stim respectively, running the simulation, and finally outputting the results in a file. Beyond that, there's also the execution of DrawTimeGraphs, which displays the output graphically in terms of waveforms.

```
void Simulation(vector <stimulus*>& stimuli, vector <Components*>& Components, vector <BoolVar*>& Inputs, vector<BoolVar>& SortedOutput) //simulation function
{
    string postfix; //string to be used for postfix
    int time = 0; //time of simulation
    int c = 0;
    bool FirstSim = true;



    for (int j = 0; j < Components.size(); j++)
    {
        postfix_to_bool(Components[j], Postfix(Components[j]), time, FirstSim, SortedOutput); //will generate a boolean expression given components, a postfix expression,
    }

    for (int i = 0; i <Inputs.size(); i++)
    {
        if(Inputs[i]->value == 0)
        {
            SortedAddition(*Inputs[i], SortedOutput);
        }
    }


    FirstSim = false;

    for(int i = 0; i<stimuli.size(); i++)
    {
        c = i+1; //check the following input

        time = stimuli[i]->time_stamp_ps; //get the time of the input we are at

        InputChecker(stimuli,Inputs,i, time, SortedOutput); //apply the changes to the input through the event

        if(c <stimuli.size()) //if the next input is still within the stimuli
        {
            while(time == stimuli[c]->time_stamp_ps) //while the time of input we're at is the same as the one after
            {
                if(c !=stimuli.size())
                {
                    InputChecker(stimuli,Inputs,c,time, SortedOutput); //apply the changes to the next input through the event
                    c++; //go to the next character
                    i = c-1; //change the i accordingly (accelerating the for loop)

                }
            }
        }



        for (int j = 0; j < Components.size(); j++)
        {
            postfix_to_bool(Components[j], Postfix(Components[j]), time, FirstSim, SortedOutput); //get the boolean expression of the now changed inputs

        }


    }



    for(int i = 0; i<Components.size(); i++) //displaying all the components (output name and value)
    {
        cout << Components[i]->output->name << endl;
        cout << Components[i]->output->value << endl << endl;
    }
}
```

The above picture shows the Simulation function. We take in a stimuli vector, a components vector, an inputs vector, and an output vector. The output vector will later be written to a text file, namely the .sim file. Of course, we need the stimuli to run the simulation as this entire project is event driven, and the events are contained within the .stim file. We also need the components vector as components by definition have an instance of LogicGates, and the LogicGates by definition have a functionality. We need the functionality to be able to evaluate our boolean expressions (which as of now are not yet formed).

```
string Postfix(Components* component) //function that turns the components to an expression in Postfix
{
    string postfix; //the string that will be returned (in postfix)
    stack<char> holder; //stack used for the bitwise operators and the parentheses
    for(int i = 0; i<component->gate.functionality.size(); i++) //exploring the entirety of the functionality of component
    {
        if((component->gate.functionality[i] != '&') &&(component->gate.functionality[i] != '|') && (component->gate.functionality[i] != '~')&& (component->gate.functionality[i] != '^')&&
            (component->gate.functionality[i] != '(')&& (component->gate.functionality[i] != ')')) //if character is not a bitwise operator or a parenthesis
        {
            postfix.push_back(component->gate.functionality[i]); //add the character to the end of the postfix string
        }
        else if((component->gate.functionality[i] == '&')||(component->gate.functionality[i] == '|')||(component->gate.functionality[i] == '~')||(component->gate.functionality[i] == '^')) ,
        {
            while (!holder.empty() && ((Precedence(holder.top()) >= Precedence(component->gate.functionality[i])))) //while there are elements in holder and the precedence of the top eleme
            {
                postfix.push_back(holder.top()); //add the top of the holder to the postfix string
                holder.pop(); //explore next element in holder
            }
            holder.push(component->gate.functionality[i]); //push to holder stack
        }
        else if (component->gate.functionality[i] == '(') //if character is open parenthesis
        {
            holder.push(component->gate.functionality[i]); //push the character into the holder stack
        }
        else if (component->gate.functionality[i] == ')') //if character is closed parenthesis
        {
            while(holder.top() != '(' && !holder.empty()) //while the top element is not "(" and the holder isn't empty
            {
                postfix.push_back(holder.top()); //push the top to the end of postfix
                holder.pop(); //explore next top
            }

            holder.pop();
        }
    }

    while(!holder.empty()) //while there are still characters in holder
    {
        if(holder.top() == '(') //if the character at the top is "("
        {
            holder.pop(); //simply pop it and don't add it to the string
        }
        else
        {
            postfix.push_back(holder.top()); //else push the top of the holder to the end of postfix
            holder.pop(); //explore next element in holder
        }
    }

    return postfix; //return the components functionality now in postfix format as a string
}
```

The above function takes in a Components pointer and turns it into a postfix expression (a string). For example, if we have A&B, it should be AB& in postfix. To do this, we loop through the functionality part of the LogicGate, which belongs to Components, character by character. If the character is not a bitwise operator or parentheses, then simply push it at the end of our postfix string. If however the character is one of the 4 bitwise operators (&, |, ~, ^), then we make use of a stack called holder for the operators. Holder is also used when we deal with the parentheses, as shown in the code. So in regards to how we handle the bitwise operators, we also make use of another function called Precedence (explained ahead) that returns a ranked integer based on the operator (as the operators are not of equal priority). But in general, as long as there are elements in the holder stack (there are operators or parentheses), if the priority of the top of that stack is at least that of the character we're at, then we'll push the top to the back of the string. If our character is the open parenthesis, simply add it to the stacks. If the character is a closed parenthesis, we push to the end of the string the top of the stack and pop as long as the top is not an **open parenthesis*** (if we see an open parenthesis, this means that we "balanced" our expression out and we have that in a valid mathematical expression so boolean algebra included, the number of opening parenthesis should be equal to the closing parenthesis) and the stack is not empty. Note that the parentheses actually don't make an appearance in the postfix expression, but they are nonetheless still important in determining the order of execution of operations.

```
int Precedence(char A) //function that gets the precedence (step in the process of changing the functionality to a valid postfix expression)
{
    if(A == '~') //highest precedence is negation
    {
        return 5; //return the highest number
    }
    else if (A == '&') //& is second highest precedence
    {
        return 4; //return second highest number
    }
    else if (A == '^') //^ is 3rd highest precedence
    {
        return 3; //return 3rd highest number
    }
    else if (A == '|') //| is 4th highest precedence
    {
        return 2; //return 4th highest number
    }
    else if(A == '(') //( is 5th highest precedence
    {
        return 1; //return 5th highest number
    }

}
```

This is the Precedence function, as described previously. It returns an integer based on the priority of an operator or character (in this case parenthesis) in the priority hierarchy. The highest is negation (as such it takes the highest integer value), followed by conjunction, XOR, disjunction, and the open parenthesis symbol.

```
void postfix_to_bool(Components* component, string postfix, int& time, bool& firstsim, vector<BoolVar>& SortedOutput) //function that transforms a postfix expression to a boolean one
{//parameters are the components pointer, the postfix string (generated from the previous function), the time (for the simulation) and the output file
    BoolVar* holder1; //BoolVar pointer for input1
    BoolVar* holder2; //BoolVar pointer for input2
    stack<BoolVar*> holderstack; //stack of BoolVar pointer
    char NextChar;
    int index;
    int timecontroller;
    bool oldvalue;

    for(int i = 0; i<postfix.size(); i++) //while we are still in the postfix expression
    {
        if(postfix[i] != '&' && postfix[i] != '|' && postfix[i] != '^' && postfix[i] != '~') //if the character is "i", meaning it isn't bitwise or parenthesis
        {
            NextChar = postfix[++i]; //get the next character
            if(NextChar >= '0' && NextChar <= '9') //if next character is a number from 0 to 9
            {
                index = NextChar - '0'; //get its index
                holderstack.push(component->inputs[index-1]); //push it to stack
            }
            else if(NextChar >= 'a' && NextChar <= 'z'){
                index = NextChar - 'a';
                holderstack.push(component->inputs[index]); //push it to stack
            }
            else if(NextChar >= 'A' && NextChar <= 'Z'){
                index = NextChar - 'A';
                holderstack.push(component->inputs[index]); //push it to stack
            }
            else
            {
                throw QMessageBox::critical(nullptr, "error", "Invalid Naming scheme in the .Lib file");
                exit(0);
            }
        }
        else if(postfix[i] == '&' || postfix[i] == '|' || postfix[i] == '^') //if the character is a bitwise operator
        {
            holder2 = holderstack.top(); //get the top element and store it in holder2
            holderstack.pop(); //pop it
            holder1 = holderstack.top(); //get the new top and store it in holder1
            holderstack.pop(); //pop it
            holderstack.push(character_to_operator(holder2, holder1, postfix[i], component)); //push in the stack the result of the operation of postfix[i] on holder1 and 2 and store it in component
            time = Maximum(holder1->currtime, holder2->currtime); //the time it takes is the max of both holder1 and holder2

        }

        else if(postfix[i] == '~') //if the character is the negation operator
        {
            holder2 = holderstack.top(); //get the top in the stack
            time = holder2->currtime;
            holderstack.pop(); //pop it
            holderstack.push(character_to_operator(holder2,holder2,postfix[i], component)); //push in the stack the result of the operation of postfix[i] on holder2 in component



        }
    }
    oldvalue = component->output->value; //old value of component
    component->output ->value = holderstack.top()->value; //the new value of component (as reached above)
    if(firstsim)
    {
        component->output->currtime = 0;

    }
    else
    {
        component->output->currtime = time + component->gate.delayps;
    }
    //the time it took it reach such output (change of event in variable + gate delay time)
    if(component->output->value != oldvalue) //if the new value is not equal to the old one
    {
        SortedAddition(*component->output, SortedOutput);
    }
    holderstack.pop();
}
```

The above function is the opposite of the second to previous one. Previously, we explored how to generate a postfix expression from Component pointer. Now, we want to evaluate that expression, that is to give it some value either 0 or 1 depending on the functionality and the inputs. The only reason we even did postfix in the first place is to eventually convert it into a boolean expression. So, the function takes in a Component pointer, a string in postfix, a time variable, a boolean variable, and a BoolVar vector containing the output. Firstly, we loop over the entirety of the length of the postfix string character by character. We begin by checking that the character is not one of our connectives. The reason we do this check, instead of for example checking whether the character is ‘i’ is because what if in a certain functionality file the variable is Z, Q, or T. The possibilities of inputs are endless. This approach takes into account the infinite possibilities of input variables and their names. Next, we check the following characters. There are 3 possibilities, either it’s a number from 0 to 9, a small letter from a -> z, or a capital letter from A -> Z. Of course, there are infinitely many other choices that we couldn’t find a way to incorporate into the code.

```
BoolVar* character_to_operator(BoolVar* A, BoolVar* B, char C, Components* component) //function that handles the behaviors of the different boolean functions (using bitwise operators)
{
    BoolVar* Out = new BoolVar(); //the out to be returned
    Out->name = "PlaceHolder :D"; //just a name given to it (not that important but needed to be done to avoid errors in the instantiation)

    switch(C)
    {
    case '&': //if AND
        Out->value = A->value & B->value; //perform AND operation

        break;
    case '|': //if OR
        Out->value = A->value | B->value; //perform OR operation

        break;
    case '~': //if NOT
        Out->value = A->value ^ true; // perform NOT operation

        break;
    case '^': //if XOR
        Out->value = A->value ^ B->value; //perform XOR operation

        break;
    }

    return Out; //return the output
}
```

The above function is where the evaluation of the boolean expression takes place. 2 input variables are passed, in addition to a char representing the bitwise operator. The component parameter is actually not needed but we did not have time to remove it (as we'll need to trace the code and remove all instances of it in any calls). By utilizing the switch case, we decide what operation is performed based on the bitwise operator. Note that ~ is simply the value of the variable XOR with 1.

```
void InputChecker(vector <stimulus*>& stimuli, vector <BoolVar*>& Inputs, int i, int& time, vector<BoolVar>& SortedOutput)
{
    for(int j = 0; j<Inputs.size(); j++) //while there are inputs to explore
    {

        if(stimuli[i]->input->name == Inputs[j]->name) //if the inputs match
        {
            Inputs[j]->value = stimuli[i]->new_value; //change the value to that in the stimulus file
            Inputs[j]->currtime = stimuli[i]->time_stamp_ps; //update the time likewise to when the event takes place
            SortedAddition(*Inputs[j],SortedOutput);
        }

    }
}
```

The above function, InputChecker, assigns the new values to the desired input based on the event description in the stimulus. We firstly check whether the input mentioned in the stimuli is present in our BoolVar vector of inputs. If that is the case, we then do the assignments and call the function SortedAddition (explored later).

```
int Maximum(int A, int B)
{
    if (A > B)
    {
        return A;
    }
    else if (B > A)
    {
        return B;
    }
    else
    {
        return A;
    }
}
```

The above function Maximum is very trivial. It returns the maximum of 2 numbers. Otherwise, return the first.

```
void SortedAddition(BoolVar value, vector<BoolVar>& SortedOutput) {
    if (SortedOutput.empty()) {
        SortedOutput.push_back(value);
        return;
    }

    // Case: value's currtime is smaller than the first element in SortedOutput
    if (value.currtime <= SortedOutput.front().currtime) {
        SortedOutput.insert(SortedOutput.begin(), value);
        return;
    }

    // Case: value's currtime is larger than the last element in SortedOutput
    if (value.currtime >= SortedOutput.back().currtime) {
        SortedOutput.push_back(value);
        return;
    }

    // Find the appropriate position to insert value while maintaining sorted order
    for (int i = 0; i < SortedOutput.size(); ++i) {
        if (value.currtime <= SortedOutput[i].currtime) {
            SortedOutput.insert(SortedOutput.begin() + i, value);
            return;
        }
    }
}

void SortedStimuli(stimulus *value, vector<stimulus*>& SortedOutput) {
    if (SortedOutput.empty()) {
        SortedOutput.push_back(value);
        return;
    }

    // Case: value's time_stamp is smaller than the first element in SortedOutput
    if (value->time_stamp_ps <= SortedOutput.front()->time_stamp_ps) {
        SortedOutput.insert(SortedOutput.begin(), value);
        return;
    }

    // Case: value's time_stamp is larger than the last element in SortedOutput
    if (value->time_stamp_ps >= SortedOutput.back()->time_stamp_ps) {
        SortedOutput.push_back(value);
        return;
    }

    // Find the appropriate position to insert value while maintaining sorted order
    for (int i = 0; i < SortedOutput.size(); ++i) {
        if (value->time_stamp_ps <= SortedOutput[i]->time_stamp_ps) {
            SortedOutput.insert(SortedOutput.begin() + i, value);
            return;
        }
    }
}
```

The above two functions both properly inserted values into vectors. One inserts in a stimulus vector, and the other in a BoolVar vector. Both ensure sorted vectors are returned.

```
void PrintInSim(QString filePath4, vector<BoolVar>& SortedOutput)
{
    ofstream outputFile(filePath4.toStdString()); //the file that we will write to
    if(!outputFile.is_open()) //error handling if the file did not open
    {
        QMessageBox::critical(nullptr, "error", "Unable To Open The File");
        exit(0);
    }
    outputFile.clear();

    for(int i=0; i<SortedOutput.size(); i++)
    {
        if(SortedOutput[i].currtime != 0)
        {
            outputFile << SortedOutput[i].currtime << ", " << SortedOutput[i].name << ", " << SortedOutput[i].value << endl;
        }
    }

    outputFile.close();
}
```

The above function grabs the output of the simulation that is stored in the vector SortedOutput and writes to the output file (i.e. .sim file).

```
int MaxValueGrabber(vector<BoolVar>& SortedOutput)
{
    int max = SortedOutput[0].currtime;
    for(int i =1; i< SortedOutput.size(); i++)
    {
        if(SortedOutput[i].currtime > max)
        {
            max = SortedOutput[i].currtime;
        }
    }

    return max;
}
```

The above function grabs the max time value from the vector SortedOutput, which is where the output of the simulation is stored.

```
void DrawTimeGraphs(vector<BoolVar>& SortedOutput)
{
    QChartView* chartView = new QChartView();
    QChart*  chart = new QChart();
    vector<QLineSeries*> lines;
    QValueAxis* axisX = new QValueAxis;
    QValueAxis* axisY = new QValueAxis;
    vector<int> lastpoint;
    int max = MaxValueGrabber(SortedOutput);

    // Set range and labels for the X axis
    axisX->setRange(0, max);
    axisX->setLabelFormat("%.0f"); // Format for axis labels (optional)
    axisX->setTitleText("Time"); // Axis title

    // Set range and labels for the Y axis
    axisY->setRange(0, SortedOutput.size());
    axisY->setLabelFormat("%.0f"); // Format for axis labels (optional)
    axisY->setTitleText("Output"); // Axis title
    chart->setLocalizeNumbers(true);

    // Add axes to the chart
    chart->addAxis(axisX, Qt::AlignBottom);
    chart->addAxis(axisY, Qt::AlignLeft);

    for(int i = 0; i<SortedOutput.size(); i++)
    {
        if(SortedOutput[i].currtime==0)
        {
            lines.push_back(new QLineSeries());
            lines[i]->setName(QString::fromStdString(SortedOutput[i].name));
            lines[i]->append(2*SortedOutput.size(),static_cast<int>(SortedOutput[i].value)+i+1);
            lines[i]->append(SortedOutput[i].currtime,static_cast<int>(SortedOutput[i].value)+i+1);
        }
        else
        {
            for(int j =0; j<lines.size(); j++)
            {
                if(lines[j]->name().toStdString() == SortedOutput[i].name)
                {
                    lines[j]->append(SortedOutput[i].currtime,static_cast<int>(!SortedOutput[i].value)+j+1);
                    lines[j]->append(SortedOutput[i].currtime,static_cast<int>(SortedOutput[i].value)+j+1);
                    lines[j]->append(max,static_cast<int>(SortedOutput[i].value)+j+1);

                }

            }
        }

    }

    for(int i = 0; i<lines.size(); i++)
    {
        chart->addSeries(lines[i]);
        lines[i]->attachAxis(axisX);
        lines[i]->attachAxis(axisY);
    }

    chartView->setChart(chart);
    chartView->show();

}
```

The above function DrawTimeGraphs graphically represents the output in terms of waveforms.

Data Structures and Algorithms:

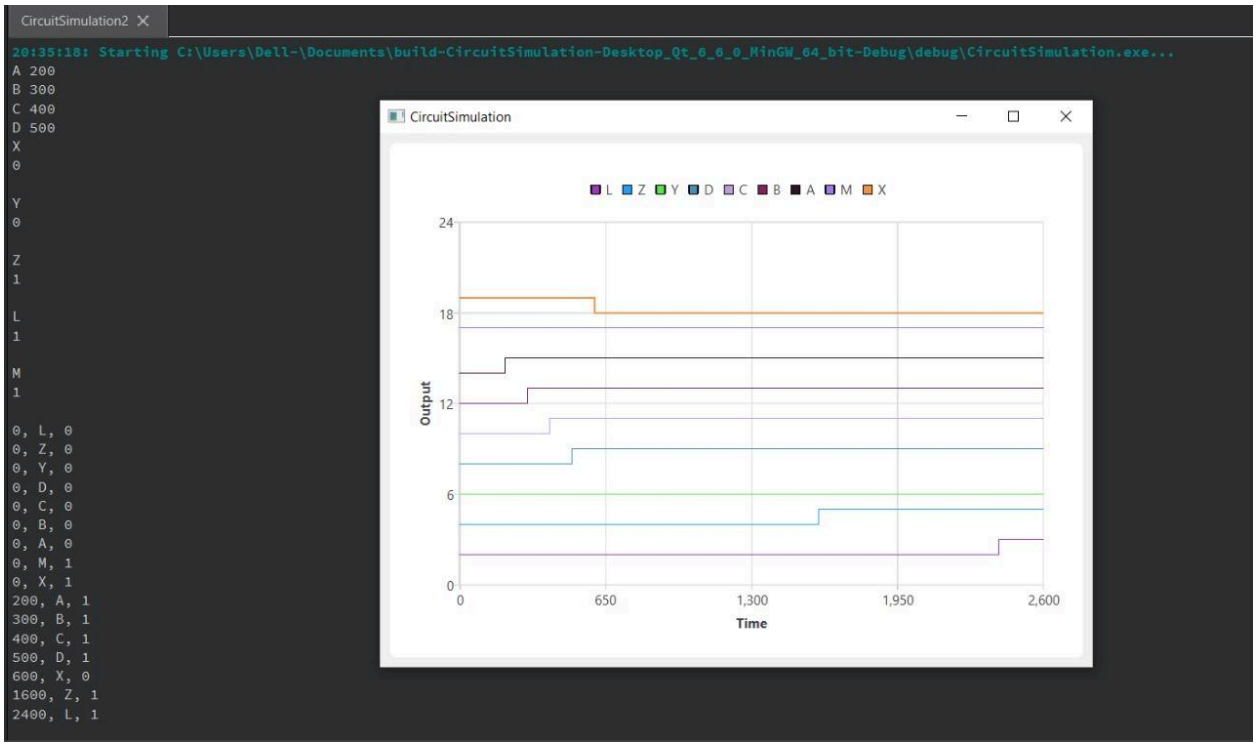
As shown in the above code snippets, we made use of lots of vectors and stacks. We also used sorting algorithms and get maximum algorithms.

Challenges:

Most of the challenges had to do with getting the correct output and time. The output took time to be fixed but was eventually completed; this is where we researched the idea of using postfix (and thus a stack) instead of a queue of pairs as the former was easier to implement. The time took more time but we also eventually figured it out.

Simulated Output:

Below is an image of the simulated output (and generally the output both graphically and on the CLI):



Contributions of Members:

1) Seif ElAnsari
    a) Drew 3 of the 5 test circuits (Circuit3, Circuit4, and Circuit5)
    b) Drew the Propagation delay of 3 of the 5 test circuits (Circuit3, Circuit4, and Circuit5)
    c) Designed the initial versions of LogicGates, BoolVar, and Components
    d) Wrote the functions ReadCircuit and ReadLibrary

e) Co-wrote the Simulation function
f) Wrote the PrintInSim function
g) Wrote the MaxValueGrabber function
h) Wrote the SortedAddition function
i) Wrote the Precedence function
j) Wrote the Postfix function
k) Co-wrote character_to_operator function
l) Implemented the graphical bonus

2) Ismaiel Sabet
   a) Drew 2 of the 5 test circuits (Circuit1 and Circuit2)
   b) Drew the Propagation delay of the 2 of the 5 test circuits (Circuit1 and Circuit2)
   c) Wrote the 5 .circ files and the lib file
   d) Wrote the fileOptimizer function
   e) Wrote the SortedStimuli function
   f) Designed the initial algorithm for the simulation in general pseudocode
   g) Improved the postfix_to_bool function
   h) Co-wrote character_to_operator function
   i) Wrote the comments for the code (apart from the ReadStimuli)
   j) Wrote the report (apart from the appendix)
3) Noor Emam
   a) Wrote the stimulus class
   b) Wrote the ReadStimulus function
   c) Wrote the comments for ReadStimulus
   d) Created the 5 .stim files
   e) Designed the postfix algorithm
   f) Co-wrote the Simulation function
   g) Wrote the maximum function
   h) Debugged syntax and run-time errors of the code, and checked for code correctness
   i) Wrote the appendix in the report

Conclusion:

So to wrap things up, in this project, we tried to implement an event driven simulator, and to that end, we succeeded. However, there is definitely room for improvement. Most notably, we do not believe that this code is efficient from a complexity analysis at all. This perspective was rooted in brute force. So the final reflecting thought: is it possible to improve the complexity of this code, and if so, how do we accomplish that?

Appendix:

Please note that the appendix serves as an additional material to prove that certain types of boolean algebra hold by using mathematical logic. The following material is based on Formal & Mathematical Logic. For instance, the notion of a formula and the validity of induction on formulas are both proved and explained in the course. Here is a citation of the lecture notes, which are available at the following link:
https://drive.google.com/file/d/1BQozLZPXhrnR_Y3gBLvDgvAan1jVJIzw/view

Reference: Daoud Siniora. Formal & Mathematical Logic Lecture Notes, 2024.

> * : We will prove that a formula has the property (o[w] = c[w]). The reason we are approaching this proof in the first place relates to the condition in a condition in the Postfix function related to the parentheses.

Case I) A propositional formula F is a propositional variable. However, we know that a propositional variable does not have opening or closing parentheses. Hence, o[w] = 0=c[w], as desired.

Case II) If there's a formula X that has the property, then G= ~X also has the property.

Assume that X has the property, then ~X has the property. However, we know that ~X does not have opening or closing parentheses, hence o[w]=0=c[w].

Case III) If there's a formula X and Y that have the property, then G= (X <> Y) has the property as well.

However, we know that O[((X <> Y))] = 1 + O[W]+ O[C] = 1 + C[W] + C[W] = C[(X<>Y)], hence proving that the property holds for G.

References:

- *Daoud Siniora. Formal & Mathematical Logic Lecture Notes, 2024.*
- Dr Shalan's slides
- GPT 3.5 (for recommendations & debugging):
    - Write a code in QT that allows us to open any file we want
    - Implement graphical representation using QChart