
Digital Circuit Simulation Project

PHYS30762, May 2022

Matylda Hoffman, 10453302

Abstract

The following project simulates a user-defined digital circuit through exploiting a range of features of object-oriented C++. Its main function consists of letting the user choose components from a provided range of derived class objects and arrange them into a logic circuit. Due to the time-dependent and sequential nature of this kind of circuit, the program makes the user compose it in order of how the logic would propagate (input to output) by storing parallel components of operations which can be performed simultaneously in the same 'timestep' object. Each step is displayed separately using a representative diagram, and, ultimately, all are joined together in series to create the final 'circuit' object. Finally, the program displays the truth table for the circuit, handling any number of circuit inputs and all their combinations

1. Introduction

Digital circuits are a fundamental concept in computer science. They use a selection of simple logic components, each performing a different operation, to construct complex logical functions. Such circuit can have an unlimited number of inputs, each taking either of the binary values 0 and 1, which are later propagated through a sequence of logical expressions until a final output is obtained. There are two kinds of logic gates: unary and binary, taking one and two inputs respectively in order to conclude an output. Unary gates include the NOT and buffer gates, meanwhile binary gates comprise of AND, OR, NAND, NOR, XOR, and XNOR. The consequent truth tables illustrating outcomes of each component depending on the input are shown in Tables 1 and 2 where A and B are inputs going into the gates.

Unary Gates		
A	NOT	BUFFER
0	1	0
1	0	1

Table 1 Unary logic gates truth tables.

Binary Gates							
A	B	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
1	0	0	1	1	0	1	0
0	1	0	1	1	0	1	0
1	1	1	1	0	0	0	1

Table 2 Binary logic gates truth table

Another functionality implemented in the program is the option to display those truth tables for each component, as well as one for the whole circuit after it has been constructed.

A digital circuit is an ordered sequence of events- consecutive outcomes depend on previous operations. The result of a logic expression depends on the order of gates that the circuit inputs are subjected to, therefore, when creating such circuit, it is essential to append operators in a desired order. In order to implement that in the program, the code consists of three main classes: 'circuit', 'timestep', and 'components'. The 'circuit' class stores the whole circuit build by a user, which consists of 'timestep' objects, which further consist of 'components'. Each timestep is a set of logic operations you can perform simultaneously corresponding to a set of gates connected in parallel. Figure 1 illustrates how this is done.

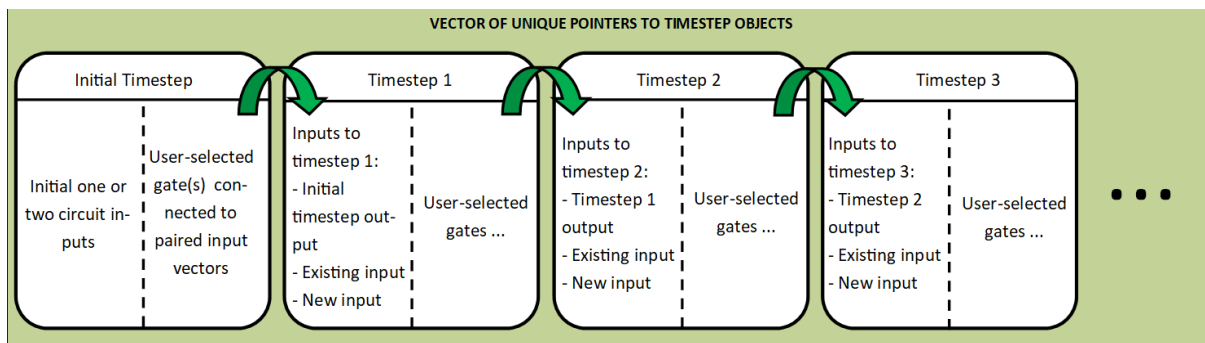


Figure 1 Standard containers used for generating the circuit and how they fit within each other. The left side of each timestep lists all possible types of pointers to objects it can store. The green block represents a vector of unique class pointers (a private variable of the circuits class), the white blocks are maps of vectors of shared pointers to component objects, to shared pointers to newly added component objects (private variable of timestep class).

2. Code Design and Implementation

The final program is constructed from nine separate files: five .cpp and four header extensions. The main function is contained in .cpp format, meanwhile the interface and each class's member functions are split between headers, containing just declarations, and .cpp's which enclose the function definitions. Each header file includes a 'header guard', which prevents a particular file from being declared more than once and causing a compiler error. Three of those contain a class declaration for the kinds of objects used in generating a circuit.

2.1. Classes

Firstly, the base class 'components' allows for the creation of individual gates used in a circuit. It is categorised further into two derived classes differentiating between unary and binary operators, which later differentiate further into the specific logic gates which can be appended to the circuit. An outstanding case is the IN derived class, which corresponds to an input to the circuit and can be initialised with a unique name by the user when building their circuit. These, when created, are additionally stored in a vector containing just circuit inputs through a shared pointer; and can be used later for initialising the custom logic sequence in a variety of combinations of binaries in order to generate a full truth table. 'Components' class uses the idea of polymorphism to specify the different properties of each gate. All items have the same set of possible characteristics (such as performing a logic operation or having a symbol which can be used to display it visually), however each characteristic is unique to the nature of the component, which was specified using pure virtual functions.

Secondly, the 'timestep' class created an object characterised by a map of base class pointers containing all operations occurring at that point in the circuit. This paired a vector of pointers to 'timestep' inputs to a pointer to a newly created component of user's choice. The timestep inputs

could either be one(unary) or two(binary) outputs of the previous timestep, one of the previously created circuit inputs, or a completely new input added by a user at that instance (Figure 1).

When creating each component, a shared smart pointer was appended to an appropriate container within a 'timestep' object. This allowed for creating copies of the pointer to that component and appending them to other containers as needed. Such instances being: storing an IN component in a separate vector of just inputs and storing each component in the timestep as *first* and *second* in a map needed for logic propagation. Furthermore, this pointer allows for a more reliable memory management as it deletes an object automatically once it goes out of scope. The program also uses a unique smart pointer when allocating timesteps to the vector in 'circuit' as those objects don't need to be accessed anywhere else.

2.2. Adding Parallel Components

New logic gates in each timestep were created using template functions. This made the code more concise as only one function was needed to create all class objects derived from the components class.

```
template<class T>
std::shared_ptr<T> add_unary_component(std::vector<std::shared_ptr<components>>& in)
{
    int out{in[0]->logic_operation()};
    std::shared_ptr<T> new_component{std::make_shared<T>(out)};
    return new_component;
}

template<class T>
std::shared_ptr<T> add_binary_component(std::vector<std::shared_ptr<components>>& in)
{
    int out{in[0]->logic_operation()};
    int out_2{in[1]->logic_operation()};
    std::shared_ptr<T> new_component{std::make_shared<T>(out,out_2)};
    return new_component;
}
```

Figure 2 Template functions for adding a new component. Separate functions for unary and binary components.

2.3. User Interface

The program serves the user and acts on what they want to do next. This decision process is navigated through the interface files. Each function is formatted to display a question about what the next action should be, together with possible options for the user to choose from by selecting a corresponding integer. Their input is validated using exception handling through try and except blocks. When checking for integers a type-error is thrown when an incorrect datatype is inputted, and a value-error when it is out of range. However, when handling input labelling, during which the user must choose a letter, an additional error is thrown when another input already has the same label. Later a lambda function is used to make sure the chosen letter fits the formatting of the circuit by capitalisation.

```

// Function to terminate timestep if all previous outputs covered
bool terminate_timestep()
{
    int user_choice;
    bool valid_data{false};
    std::cout << "\nYou have no more idle outputs left from the previous time step." << std::endl;
    << "\t1. Terminate this timestep" << std::endl;
    << "\t2. Connect another gate to input" << std::endl << std::endl;
    while (valid_data == false) {
        try {
            std::cout << "Enter option: ";
            std::cin >> user_choice;
            if (std::cin.fail()) {throw type_error;}
            else if (user_choice < 1 || user_choice > 2) {throw value_error;}
            else {valid_data = true;}
        }
        catch (int error) {
            if (error == type_error) {std::cerr << "You have not entered an integer. Please enter a number corresponding to one of the options." << std::endl;}
            else if (error == value_error) {std::cerr << "Your choice is out of the range. Please enter a number corresponding to one of the options." << std::endl;}
            std::cin.clear();
            std::cin.ignore();
            valid_data = false;
        }
    }
    if (user_choice == 1) {return false;}
    else {return true;}
}

```

Figure 3 Function validating user input for integer in desired range.

```

// Choose name for a new input
std::string choose_input_name(std::vector<std::shared_ptr<IN>>& ins_names)
{
    char label;
    std::string out;
    bool valid_data{false};
    while (valid_data == false) {
        std::cout << "\nLabel your new input with a letter (e.g. A, B, C ...): ";
        try {
            std::cout << "Enter option: ";
            std::cin >> label;
            if (std::cin.fail()) {throw type_error;}

            // lambda function to format the input letter to string and uppercase
            auto lambda = [&label] () {
                std::string tmp_string(1, std::toupper(label));
                return tmp_string;
            };
            for (std::shared_ptr<IN> n : ins_names) {
                if (n->get_gate_name() == lambda()) {throw repeat_error;}
            }
            out = lambda();
            valid_data = true;
        }
        catch (int error) {
            if (error == type_error) {std::cerr << "You have entered an invalid label(. Please enter a single LETTER." << std::endl;}
            else if (error == repeat_error) {std::cerr << "An input with that name already exists. Please choose again..." << std::endl;}
            std::cin.clear();
            std::cin.ignore();
            valid_data = false;
        }
    }
    return out;
}

```

Figure 4 Function illustrating exception handling when user input clashes with an already existing label. Moreover, a lambda function is used to format user input.

2.4. Circuit Display

After completing each timestep the user can see what it looks like visually. Input and output gates are printed in a suggestive manner so that their relationships are visible as shown below.

TIMESTEP (1)	TIMESTEP (2)
A----- --NOT---	--NOT---
A-----	+-XNOR--
+-AND---	+-AND---
B-----	

Figure 5 Example outputs of a two-timestep circuit with two inputs.

2.5. Logic Propagation

Each circuit input is binary, hence takes value of either zero or one. When creating a circuit with several inputs each permutation needs to be considered in order to generate a full set of possible results. This is achieved using a recursive function which can generate a set of all variations for a set of any length.

```
// Recursive function to generate all combinations of binary input for any number of inputs
void generate_combinations(int n, std::vector<int> combos, int i, std::vector<std::vector<int>>& all)
{
    if (i == n) {
        all.push_back(combos);
        return;
    }
    combos[i] = 0;
    generate_combinations(n, combos, i + 1, all);
    combos[i] = 1;
    generate_combinations(n, combos, i + 1, all);
}
```

Figure 6 Recursive function to generate all possible perturbations of n binary inputs.

All variations are then used to initiate circuit inputs, which is done using the convenient vector of shared pointers to IN objects, a private variable of the circuits class. In order to propagate these values throughout the remaining components of the circuit, a function is used which iterates through all timesteps, and components within those, to sequentially call their `logic_operation()` virtual functions. The output from that is then used to set input for a following gate etc. This occurs through set functions that all derived components have, which lets the program alter a private member data of an object despite encapsulation.

```
// Function to propagate logic through the circuit
int circuit::logic_output(std::vector<int>& input_set)
{
    // Set input values for all inputs
    for (size_t i{}; i < input_set.size(); i++) {
        circuit_inputs[i]->set_input(input_set[i]);
    }
    // iterate through the vector of timesteps - compiled_circuit
    for (auto& it : compiled_circuit) {
        std::map<std::vector<std::shared_ptr<components>>, std::shared_ptr<components>> t{it->get_step()};
        //From the timestep get each map and pass result of logical operation from first gates to second
        for (std::map<std::vector<std::shared_ptr<components>>, std::shared_ptr<components>>::const_iterator t_i = t.begin(); t_i != t.end(); ++t_i)
            if (t_i->second->get_gate_name() == "IN" || t_i->second->get_gate_name() == "NOT" || t_i->second->get_gate_name() == "BUFFER") {
                t_i->second->set_input(t_i->first[0]->logic_operation());
            }
            else {
                t_i->second->set_input_A(t_i->first[0]->logic_operation());
                t_i->second->set_input_B(t_i->first[1]->logic_operation());
            }
            if (t.size() == 1) {output = t_i->second->logic_operation();}
        }
    }
    return output;
}
```

Figure 7 Propagation of input variable through the circuit using virtual functions of different components. `logic_operation()` returns outcome of a gate which is late fed into a set function (`set_input()`) of a consecutive component in series.

Finally, the program displays a big truth table of outcomes for all possible input variations.

2.6. Displaying Individual Truth Tables

The program also allows for displaying a single truth table for a gate of choice. This is just an alternative option in the main menu and is simply implemented using cases which take in integer user input for efficiency. The last option is to print truth tables for all tables, and this case uses the iterative method (case 9).

```
void print_particular_table(int gate)
{
    std::cout << std::endl;
    switch (gate) {
        case 1:
            std::cout << " | A | B | AND | " << std::endl
                << " | 0 | 0 | 0 | " << std::endl
                << " | 1 | 0 | 0 | " << std::endl
                << " | 0 | 1 | 0 | " << std::endl
                << " | 1 | 1 | 1 | " << std::endl;
            break;
        case 9:
            for (int i{}; i <= 8; i++) {
                print_particular_table(i);
            }
    }
}
```

Figure 8 Iteration used to display all truth tables

Figure 9 Cases used to display particular truth table.

3. Results

As an example of this program's functionality lets build a circuit demonstrating the extensive features of the code. Circuit depicted in Figure 10 is made up of four timesteps, contains four inputs, and a wide variety of logic components, therefore demonstrates a wide range of functions.

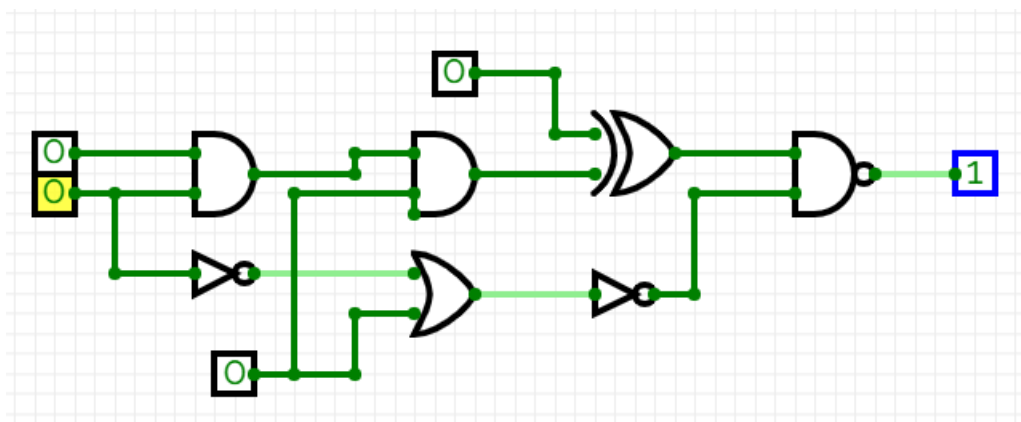


Figure 10 Example circuit. Built using <https://circuitverse.org/simulator>

Illustrative run

User choosing to build their own circuit. However, first inputting a number out of range (4), later a letter, and then finally a valid input.

```
*****
What would you like to do?
  1. Build a circuit
  2. Display the truth table for a particular logic gate
  3. Exit program

Enter option: 4
Your choice is out of the range. Please enter a number corresponding to one of the options.

Enter option: f
You have not entered an integer. Please enter a number corresponding to one of the options.

Enter option: 1
```

Instructions for the process of building a circuit are displayed:

```
*****
*                                     *
*               DIGITAL CIRCUIT SIMULATOR               *
*                                     *
* BUILDING YOUR OWN CIRCUIT - METHOD                        *
*                                     *
* Method uses timesteps which correspond to a set of simultaneous logic operations. They are created *
* one-by-one by the user. Within each timestep you can access previous circuit inputs, as well as add *
* completely new inputs to the circuit.                    *
*                                     *
* First Timestep                                           *
* 1. Create a circuit input                                *
* 2. Choose the type of gate to append                     *
*   If binary chosen then create another input to append to it *
* 3. Exit initial timestep or connect another logic gate to the inputs *
*                                     *
* Consecutive Timesteps                                    *
* 1. Choose an input to connect to out of:                 *
*   - Previous outputs                                     *
*   - An existing circuit input                             *
*   - Create a new circuit input                           *
* 2. Choose a gate to connect                              *
*   If binary chosen then create another input to append to it *
* 3. Repeat these until all previous outputs are covered and you decide to start adding *
*    components in series                                  *
*                                     *
* Circuit can only end when there is one conclusive output, hence the program will run until there *
* are no idle outputs left.                                *
*                                     *
*****
```

Timestep 1

Now building the initial timestep – here adding the AND gate.

Adding a starting input to the circuit...

Label your new input with a letter (e.g. A, B, C ...):
Enter option: a

Now choose a gate to connect to this input...

What kind of gate would you like to connect to this output?

1. Unary (1 input)
2. Binary (2 inputs)

Enter option: 2

There is only one input to this circuit so far. Adding another input to connect a binary gate...

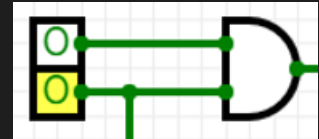
Label your new input with a letter (e.g. A, B, C ...):
Enter option: b

Which binary gate would you like to add?

1. AND
2. OR
3. NAND
4. XOR
5. XNOR
6. NOR

Enter option: 1

Added AND gate



Now adding the NOT gate to input B in parallel and terminating the timestep. The code prints off a visual representation of a circuit diagram for that timestep.

```

You have no more idle outputs left from the previous time step.
    1. Terminate this timestep
    2. Connect another gate to input

Enter option: 2

Which of these outputs would you like to connect?
    1. A
    2. B
Enter option: 2

What kind of gate would you like to connect to this output?
    1. Unary (1 input)
    2. Binary (2 inputs)
Enter option: 1

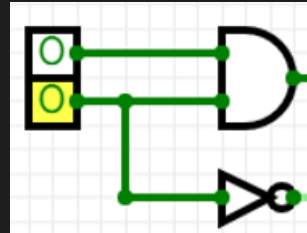
Which unary gate would you like to add?
    1. NOT
    2. Buffer
Enter option: 1
Added NOT gate

You have no more idle outputs left from the previous time step.
    1. Terminate this timestep
    2. Connect another gate to input

Enter option: 1

```

TIMESTEP	(1)
A-----	
	+-AND---
B-----	
B-----	--NOT---



Timestep 2

Now connect the output from the AND gate and a brand new input (C) to another gate (AND). The last line of the code below tells you how many idle inputs from the previous timesteps you still have to connect, and the code will not let you finish building until all outputs are covered ensuring logical propagation.

Which of these outputs would you like to connect?

1. AND
2. NOT

Enter option: 1

What kind of gate would you like to connect to this output?

1. Unary (1 input)
2. Binary (2 inputs)

Enter option: 2

Would you like to connect to:

1. Previous output
2. Existing circuit input
3. Create new circuit input

Enter option: 3

Label your new input with a letter (e.g. A, B, C ...):

Enter option: c

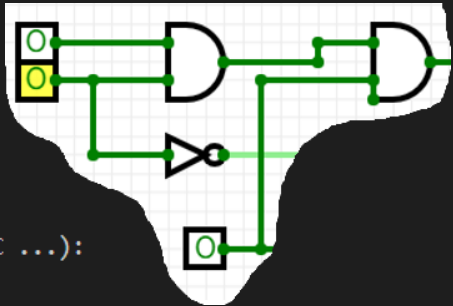
Which binary gate would you like to add?

1. AND
2. OR
3. NAND
4. XOR
5. XNOR
6. NOR

Enter option: 1

Added AND gate

You have 1 idle outputs left from the previous timestep. Decide what happens to them:



Next, connect the output from the NOT gate with the input you have just created (C) to an OR gate. The finish the timestep.

Which of these outputs would you like to connect?

1. NOT

Enter option: 1

What kind of gate would you like to connect to this output?

1. Unary (1 input)
2. Binary (2 inputs)

Enter option: 2

There is not enough outputs left from the previous timestep :(. Please choose a circuit input to connect to:

1. Existing circuit input
2. Create new circuit input

Enter option: 1

1. Input A
2. Input B
3. Input C

Enter option: 3

Which binary gate would you like to add?

1. AND
2. OR
3. NAND
4. XOR
5. XNOR
6. NOR

Enter option: 2

Added OR gate

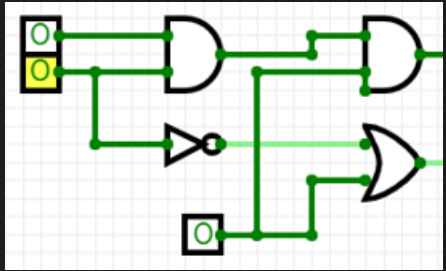
You have no more idle outputs left from the previous time step.

1. Terminate this timestep
2. Connect another gate to input

Enter option: 1

TIMESTEP	(2)
+AND---	+AND---
C-----	
--NOT---	
	+OR----
C-----	

You should connect to all outputs from the previous timestep



Timestep 3

Firstly, connect a unary NOT gate to the output coming from the previous OR gate.

Which of these outputs would you like to connect?

1. AND
2. OR

Enter option: 2

What kind of gate would you like to connect to this output?

1. Unary (1 input)
2. Binary (2 inputs)

Enter option: 1

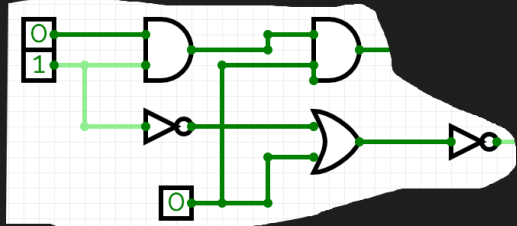
Which unary gate would you like to add?

1. NOT
2. Buffer

Enter option: 1

Added NOT gate

You have 1 idle outputs left from the previous timestep. Decide what happens to them:



You now need to connect the output from the AND gate to an XOR gate and create another input (e.g. D) to connect as the second input.

Which of these outputs would you like to connect?

1. AND

Enter option: 1

What kind of gate would you like to connect to this output?

1. Unary (1 input)
2. Binary (2 inputs)

Enter option: 2

There is not enough outputs left from the previous timestep :(. Please choose a circuit input to connect to:

1. Existing circuit input
2. Create new circuit input

Enter option: 2

Label your new input with a letter (e.g. A, B, C ...):

Enter option: d

Which binary gate would you like to add?

1. AND
2. OR
3. NAND
4. XOR
5. XNOR
6. NOR

Enter option: 4

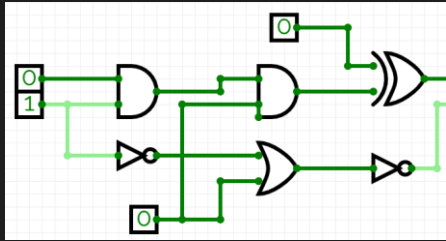
Added XOR gate

You have no more idle outputs left from the previous time step.

1. Terminate this timestep
2. Connect another gate to input

Enter option: 1

TIMESTEP (4)	
+AND---	
	+XOR---
D-----	
+OR----	--NOT---



Timestep 4

Finally, connect the two remaining outputs to a NAND gate and terminate circuit.

```
Which of these outputs would you like to connect?
  1. XOR
  2. NOT
Enter option: 1

What kind of gate would you like to connect to this output?
  1. Unary (1 input)
  2. Binary (2 inputs)
Enter option: 2

Would you like to connect to:
  1. Previous output
  2. Existing circuit input
  3. Create new circuit input
Enter option: 1
Please choose another input which you would like to connect this gate to.

Which of these outputs would you like to connect?
  1. NOT
Enter option: 1

Which binary gate would you like to add?
  1. AND
  2. OR
  3. NAND
  4. XOR
  5. XNOR
  6. NOR
Enter option: 3

Added NAND gate

You have no more idle outputs left from the previous time step.
  1. Terminate this timestep
  2. Connect another gate to input
Enter option: 1

|  Timestep  (6)  |
|-----|
| +-XOR---|
|           +-NAND--|
| --NOT---|
```

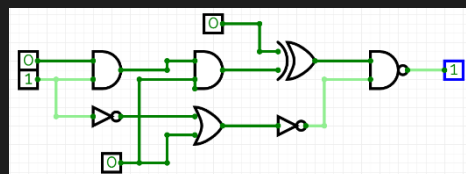
Now the circuit is finished, the program prints off a truth table for all perturbations of the four (in this case) inputs.

```
You have a single output coming out of the circuit, therefore you may compute a definite logic operation.
Would you like to terminate or add another component in series? (please type 1 or 2)
  1. Terminate circuit
  2. Add a step in series
Enter option: 1

* TRUTH TABLE *

| A | B | C | D | OUTPUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1       |
| 0 | 0 | 0 | 1 | 1       |
| 0 | 0 | 1 | 0 | 1       |
| 0 | 0 | 1 | 1 | 1       |
| 0 | 1 | 0 | 0 | 1       |
| 0 | 1 | 0 | 1 | 0       |
| 0 | 1 | 1 | 0 | 1       |
| 0 | 1 | 1 | 1 | 1       |
| 1 | 0 | 0 | 0 | 1       |
| 1 | 0 | 0 | 1 | 1       |
| 1 | 0 | 1 | 0 | 1       |
| 1 | 0 | 1 | 1 | 1       |
| 1 | 1 | 0 | 0 | 1       |
| 1 | 1 | 0 | 1 | 0       |
| 1 | 1 | 1 | 0 | 1       |
| 1 | 1 | 1 | 1 | 1       |

*****
```



The program later return to the main and you can choose to view a truth table for a single component, build another circuit, or exit. Here is what viewing the truth table for an XOR gate looks like:

```

What would you like to do?
  1. Build a circuit
  2. Display the truth table for a particular logic gate
  3. Exit program

Enter option: 2

*****

Which component's truth table would you like to look at?
  1. AND
  2. OR
  3. NAND
  4. XOR
  5. XNOR
  6. NOR
  7. NOT
  8. BUFFER
  9. All the above

Enter option: 5

| A | B | XOR |
| 0 | 0 | 0   |
| 1 | 0 | 1   |
| 0 | 1 | 1   |
| 1 | 1 | 0   |

*****

```

Conclusively, you can choose to exit the program.

```

What would you like to do?
  1. Build a circuit
  2. Display the truth table for a particular logic gate
  3. Exit program

Enter option: 3
Exiting program...

```

4. Discussion and Conclusion

Conclusively, the project successfully simulates the build of a digital circuit and allows the user to fully design their own setup. It utilises a wide variety of components and successfully propagates input values through to the final output. Also importantly, it produces a truth table for a custom-built circuit.

The program has a vast scope of potential additional functionalities as the subject matter has plenty of implementations and intricacies. It so far performs complex logic operations with simple logic gates; however, it could be taken further by adding more advanced logic features such as latches or flip-flops. These components would mean that a specialisation would have to be made into sequential and combinational digital circuits, as one (sequential) requires clock input in order to vary the latches' outputs. This would also require the implementation of feedback loops, which can all be integrated into the current design of the project. Due to the time orderings being elegantly considered in timesteps, they could be connected backwards to each other, and the logic propagation would occur in the same way. Another closely related concept to digital circuits is Boolean algebra. It is also a neat and clear way of describing logic expressions in a concise way. Therefore, this program would also benefit from being able to display a Boolean expression for a user-built circuit, as well as the ability to construct a circuit straight from a Boolean expression.

Overall, the straight-forward and unrestricted design of this program provides good basis for further extensions of the project.