GF2 Team 17

Ioannis Demetriades, Richard Monteiro, Yash Gaikwad

May 2022

1 Introduction

In this report we describe the plan and the fundamental ideas of our approach on how to develop the logic simulator.

For the development part, we focused on designing a simple to understand and easy to use language. Firstly, we included the required features and the we added some extra extension features which will make the language more efficient to use. We firmly believe in the object-oriented philosophy and have carefully crafted our language to allow the user to modularise and abstract different features of the language and build up the circuits in a simple and systematic manner. The following part explains the structure of our team and the tasks we will tackle.

1.1 Team Planning

In any software development project, it is important to modularise all tasks such that a group can work efficiently to the given time constraints. The team decision was to split up the development into two main sections: ”logic simulator” and ”GUI”. Logic simulator contains the coding of the Names, Scanner and Parser modules, while ”GUI” focuses on the development of a GUI interface. Both sections can be worked on independently for most of the project, until they are combined at the final stage.

Two team members (Ioannis and Yash) are chosen to work with the ”logic simulation” section, while another team member (Richard) will work on the GUI. However, to ensure every member participated in every aspect of code design, the members from one section will be chosen to write the testing for the other and vice-versa. This choice also has the benefits of being much more effective in writing good, complete tests for the system. Tests will generally be designed with the intent of breaking the code as much as possible, therefore also ensuring a robust design.

To plan and communicate the time structure of the project, a Gantt chart was built and shared amongst the team members. A section of it is displayed in appendix 1, but the full chart can be accessed through the link: `https://bit.ly/3LAuzap`.

2 EBNF syntax rules

For EBNF of circuit, please see the extension section. Comments start with `#` and end with a newline. Comments are removed by the scanner and hence are not included in the language specification.

```
(* Defining name and number *)

digit_excluding_zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
digit                = "0" | digit excluding zero ;
number = digit_excluding_zero, {digit,};

binary_digit  = ("0" | "1")
binary_number = binary_digit , {binary_digit};

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
```

```
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" ;

name = letter , { letter | digit | "_"  } ;

alpha = letter , { letter } ;
alphanum = ( letter | digit ) , { letter | digit } ;


default_sig = name , "." , default_port ;

default_port = ( default_input | default_output ) ;

default_input = "I" , number ;
default_output = "O" , number ;


signame = ( name , [ "." , alphanum ] ) ;
(* We are not being specific with the signal names *)
(* Like restricting inputs to I1, I2, ... and DATA, CLK, SET, CLEAR *)
(* This is because we want to give users the flexibility to rename inputs and outputs *)
(* This is particularly useful for circuits which are defined later *)
(* Most errors in signame will be semantic not syntactic *)


(* To allow user to create multiple devices at ones, we define loop_times *)
(* This is an optional parameter *)
(* which will allow writing statements like NAND nand[1 to 3](IN = 2); *)
(* This is equivalent to NAND nand1(IN = 2), nand2(IN = 2), nand3(IN = 2); *)

loop_times = "[" , number , "TO" , number , "]" ;


(* Defining a switch *)

switch = "SWITCH" , single_switch , { "," , single_switch } , ";"  ;
single_switch = name , [ loop_times ] , "=", binary_digit ;


(* Flipping a switch *)

switchset = "SETSWITCH" , name , "=" , binary_digit , ";" ;


(* Defining a clock *)

clock = "CLOCK" , name , [ loop_times ] , "(" , "PERIOD" , "=" , number ")" , ";"  ;
```

```
(* Defining AND, OR, NAND and NOR gate *)
(* We allow the user to create multiple gates in one go using loop_times *)

gate_list = gate_name , gate, { "," , gate } , ";" ;

gate = name , [ loop_times ] , in ;
in = "(" , "IN" , "=" , number , ")" ;

gate_name = ( "OR" | "AND" | "NAND" | "NOR" | "XORN" ) ;


(* Defining a XOR gate *)

xor_list = "XOR" , xor_gate , { "," , xor_gate } , ";" ;
xor_gate = name , [ loop_times ] ;


(* Defining a NOT gate *)

not_list = "NOT" , not_gate , { "," , not_gate } , ";" ;
not_gate = name , [ loop_times ] ;

(* Defining a DTYPE *)
(* For DTYPE the inputs by default are DATA, CLK, SET, CLEAR *)
(* and the outputs are by default Q and QBAR *)
(* They are accessed by dtype1.DATA, dtype1.Q etc *)

dtype = "DTYPE" , single_dtype , { "," , single_dtype } , ";" ;
single_dtype = name , [ loop_times ] ;


(* Making a connection *)

connectlist = "CONNECT" , connection , { "," , connection } , ";" ;
connection = signame , "->" , signame ;


(* Monitoring points *)

monitor_list = "MONITOR" , signame , { "," , signame } , ";" ;


(* Changing input/output label *)
(* Suppose you define NAND nand1(IN = 2); and you want the inputs to be *)
(* nand1.X and nand1.Y instead of nand1.I1 and nand1.I2 *)
(* Then use LABEL nand1(I1 = X, I2 = Y); *)
(* This is particularly useful for circuits, which are defined later *)

label = "LABEL" , single_label , { "," , single_label } ,  ";" ;

single_label = name , label_setting ;
label_setting = "(" , default_port , "=" ,  alpha , { "," , default_port , "=" ,  alpha  } , ")" ;
```

```
(* Circuits and devices can be defined in a different file *)
(* To import them *)

import_circuit_list = "IMPORT", name , ";";
```

3 Error Handling

Detecting and reporting errors requires using two of the main modules, 'Scanner' and 'Parser'. The process followed to detect and raise errors is shown below.

1. While the syntax is analysed, the 'Parser' module checks for syntactic and semantic errors which will report and 'Scanner' will record the exact position of the error.
2. In addition, the syntax errors will be detected using the `parse_network` method of the 'Parser' class/module and in combination with the 'devices', 'network' and other modules, the semantic errors will be detected.
3. Also, during this process when an error occurs the program will not be terminated, instead, once it detects a stopping symbol following the error detection, it will continue parsing.

A module will be created specifically for error handling, which will contain all possible errors codes mapped to their corresponding error messages. We will separate errors into syntax and semantic error, and will try to provide as much information to the user about the origin of the error. When an object is initiated, the error type, the line number and the position of the error will be passed as arguments in order to display a message as shown below.

```
error(type = syntax, num_line = 5, pos = 20)
Syntax error: Invalid qualifier
Line:5
    >>CONNECT AND and(IN=a);
                         ^

error(type = InputToInput, num_line = 6, pos = 14)
InputToInput: The input of a device is connected to another device input
Line:6
    >>CONNECT A.I1=B.I1;
                   ^
```

Once parsing is completed, it will return the number of errors, and display all the 'error' objects with the corresponding information.

3.1 Syntax errors

Syntax errors are the errors that do not adhere to the EBNF grammar rules. If an error is detected, it is necessary to skip the error and resume parsing rather than terminate the parser. We consider two cases which may arise due to syntax errors:

1. User has an error in a line, but the line is terminated using a semicolon. In this case, a semicolon can be used as a stopping symbol and the parsing can be resumed after it. However, since some of our command allow multiple definitions in a line, we will also be treating a comma as a stopping symbol. Consider the following example

   ```
   NAND nand1(IN = a), nand2(IN = 10), nand3(IN = 5);
   ```

Here, once the first erroneous definition `nand1(IN = a)` is skipped (IN requires a numerical argument), the parsing can resume after the comma. There are however certain subtleties to consider when doing this, for example, since we are in the same command, the parser needs to keep track of the context, ie we are defining `NAND` gates. Also, multiple errors in a single line may clutter the error reporting.

2. User forgets to terminate the current erroneous line using a semicolon. Consider the example below:

   ```
   NAND nand1(IN = 2), nand2(IN =
   OR or1(IN = 3);
   ```

   This error is rather difficult to detect since the specification requires the syntax to be free-format, so a linebreak cannot be used as a termination of a command. In this case, the first line is incomplete, so once an error is reported, the parser must resume operation at the next command. Since every command starts with a keyword, we have decided to treat keywords (in this case, it is `OR`) as stopping symbols.

Whenever, a syntactic name, number or non-terminal etc is missing or incorrect, we will raise an error to the user. Usually, a generic error is sufficient. For example in the case below,

```
NAND nand1(IN = a);
                ^
```

it is sufficient to say that that `number is expected`. However, our framework is easily extensible to include more specific errors as well, since which error code to pass to the error reporting function can be changed accordingly. For example, if a user types

```
XOR xor1(IN = 2);
```

it may be better to tell the user `XOR expects 2 inputs so define the gate as XOR gate_name;`, instead of `ERROR: expected a comma`.

3.2 Semantic errors

Semantic error is the category of errors which follow the syntax rules though, they do not obey the program logic and causes incorrect results. If the syntax is correct, the parser will call the functions in the respective modules, such as 'devices' and 'networks'. Semantic errors will be detected in the respective modules since it allows a good organisation of errors.

The following list shows a variety of possible semantic errors and how they will displayed. Some of the errors such as those related to the device qualifier, will display a general message followed by a device specific error. Also, we have made a rigorous checks in the syntax which will automatically prohibit many errors from occurring. For example, if user does not specify the number of inputs, then it will be syntax error rather than a `NoQualifier` semantic error. The message displayed are shown below:

- **InputToInput**: Trying to connect input of a device to another device input.
- **OutToOut**: Trying to output input of a device to another device output.
- **InputConnected**: Trying to connect a device to an input pin which is already connected to another device.
- **InvalidPort**: Trying to access an invalid input/output port.
- **InvalidDevice**: Trying to access or monitor a device which has not been created.
- **InvalidQualifier**: Invalid device qualifier value.

   1. GATES - Number of inputs gates must be positive (non-zero) and less that or equal to 16 (`nand1.I22` is invalid).
   2. SWITCH - Invalid initial state.
   3. CLOCK - Qualifier must be a non-zero.

- **QualifierPresent** - This device properties are already defined.
- **UndefinedDevice**: This device can not be created (it's not a predefined device like NAND, NOR etc).
- **DuplicateDevice**: This device name already exists.
- **InvalidName**: This device name is invalid because it is a keyword.
- **DuplicatedPort**: This port name is already in use by the device (used when relabelling the device ports).
- **InadequateInput**: More/Less number of input pins are connected to a device than specified.
- **UnMonitoredOuputID**: The chosen output ID does not belong to monitored device. Can arise when user tries to monitor an input instead of an output.
- **IterationError**: The first term (i) in a loop [i TO j] command need to be smaller than the second (j), arises when using NAND nand1[i TO j](IN = 2);.

4 Extensions

We will be trying to include these additional features if time permits.

4.1 Circuits

To allow users to encapsulate complicated logic circuits, we have decided to include circuits in our language. If for example the user frequently uses a JK flip flop, then user can just define a circuit to abstract the flip flop mechanism and then create a flip flop simply using `JK gate1(IN = 3, OUT = 2)`. Features of a circuit:

1. All gates defined within the circuit are encapsulated and cannot be accessed from outside the circuit. Once a circuit is defined, only its specified inputs and outputs can be accessed.
2. Defining circuits within circuits will be allowed since for the python handler, a circuit object is no different to a gate object.

EBNF for a circuit:

```
(* Defining a circuit *)

circuitdef = "CIRCUIT" , name , in_out_unordered , "{" , { command } , "}"

in_out_unordered = ( in_out_ordered_1 | in_out_ordered_2 )

in_out_ordered_1 = "(" , "IN" , "=" , number , "," , "OUT" , "=" , number , ")"
in_out_ordered_2 = "(" , "OUT" , "=" , number , "," , "IN" , "=" , number , ")"


(* Creating a circuit *)

circuit = name , [ loop_times ] , in_out_unordered ;
loop_times = "[" , number , "TO" , number , "]" ;

circuit_list = name , circuit, { "," , circuit } , ";" ;
```

One key issue for a circuit is that it is the only component in our language that requires input to input connections. Either the semantic conditions will need to be changed for the circuit, or a different non-terminal needs to be used for connections. Also, the way we have defined circuits essentially allow users to create their own keyword. This means we may need additional error check or else we may have to slightly change the way we create a circuit.

5 Examples

5.1 JK BISTABLE

The circuit diagram is shown in appendix Figure 3.

```
# Initialising devices
NAND nand[1 TO 2](IN = 3), nand[3 TO 4](IN = 2);
# Equivalent to saying
# NAND nand1(IN = 3), nand2(IN = 3), nand3(IN = 2), nand4(IN = 2);

SWITCH sw[1 TO 2] = 0;
# Equivalent to saying
# NAND sw1(IN = 3) = 0, nand2(IN = 3), nand3(IN = 2), nand4(IN = 2);

CLOCK clk(PERIOD = 50);

# Making connections
# Syntax: CONNECT output -> input, ...;
CONNECT sw1 -> nand1.I2, sw2 -> nand2.I2;
CONNECT clk -> nand1.I3, clk -> nand2.I1;
CONNECT nand3 -> nand1.I1, nand4 -> nand2.I3;
CONNECT nand1 -> nand3.I1, nand2 -> nand4.I2;
CONNECT nand4 -> nand3.I2, nand3 -> nand4.I1;

MONITOR nand3, nand4;
```

5.2 More Complicated logic gate

The circuit diagram is shown in appendix Figure 3.

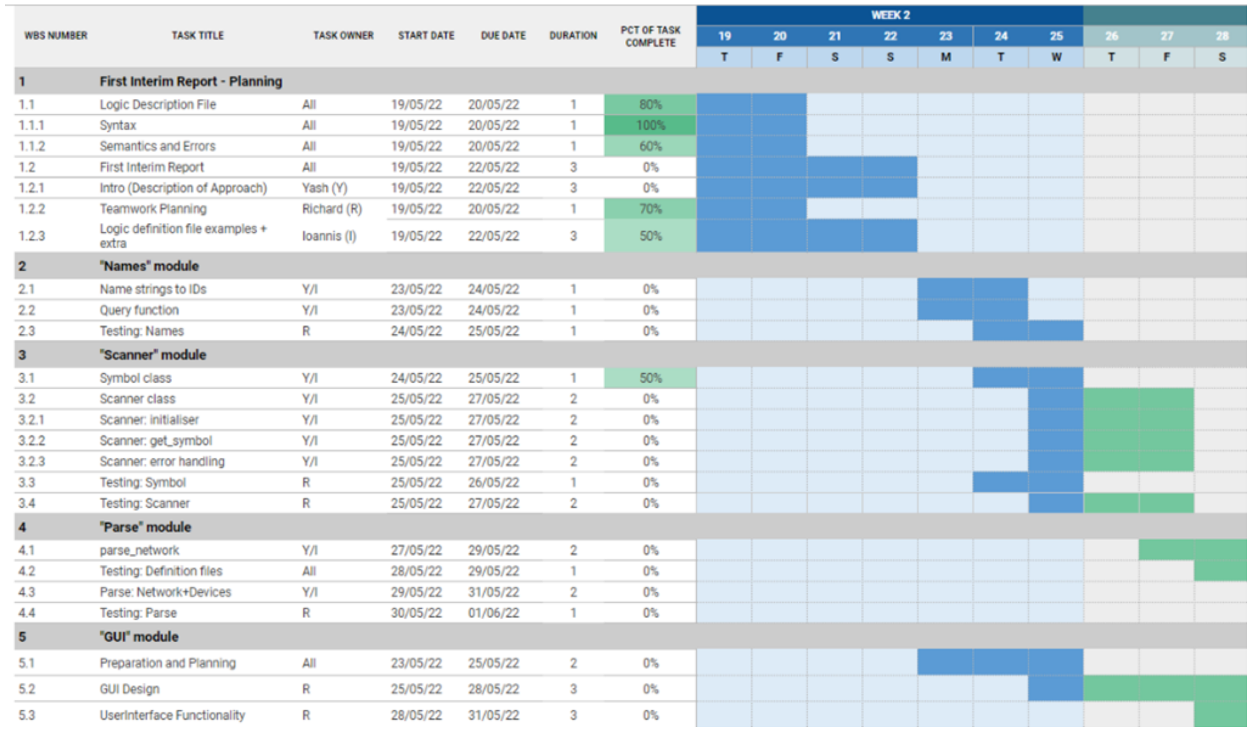
```
# Initialising devices
NOT not[1 TO 2];
AND and[1 TO 4](IN = 2);
OR or1(IN = 2);
SWITCH A = 0, B = 0, C = 0;

# Making connections
CONNECT A -> not1.I1, C -> not2.I1, C -> and4.I2;
CONNECT B -> and1.I2, B -> and4.I1;

CONNECT not1 -> and1.I1, not1 -> and2.I1
CONNECT not2 -> and2.I2;

CONNECT and1 -> and3.I1, and2 -> and3.I2;
CONNECT and3 -> or1.I1, and4 -> or1.I2;

MONITOR or1;
```

6 Examples for Circuit Extension

Using circuit (connecting complicated circuit output to JK input). The circuit diagram is shown in appendix Figure 4.

```
# Defining a circuit

CIRCUIT JK(IN=3, OUT=2)
{
    NAND nand[1 TO 2](IN = 3), nand[3 TO 4](IN = 2);

    CLOCK clk(PERIOD = 50);

    # Making connections
    CONNECT sw1 -> nand1.I2, sw2 -> nand2.I2;
    CONNECT clk -> nand1.I3, clk -> nand2.I1;
    CONNECT nand3 -> nand1.I1, nand4 -> nand2.I3;
    CONNECT nand1 -> nand3.I1, nand2 -> nand4.I2;
    CONNECT nand4 -> nand3.I2, nand3 -> nand4.I1;

    # Making input to input connections
    # Defining inputs for the circuit
    CONNECT JK.I1 -> nand1.I2, JK.I3 -> nand2.I2;
    CONNECT JK.I2 -> nand1.I3, JK.I2 -> nand2.I1;

    # Making output to output connections
    # Defining outputs for the circuit
    CONNECT nand3 -> JK.O1, nand4 -> JK.O2;

    # Re-labelling the inputs
    LABEL JK(I1 = J, I2 = CLK , I3 = K, O1 = Q, O2 = QBAR);
}


# Creating circuits
JK jk1;

# Import com2 from file stdcircuit
IMPORT stdcircuit;
com1 circ[1 TO 2];

# Defining devices
SWITCH A = 0, B = 0, C = 0;
NOT not[1 TO 3];
CLOCK clk1(PERIOD = 50);

# Making connections
CONNECT A -> circ1.I1, B -> circ1.I2, C -> circ1.I3;
CONNECT A -> not1.I1, B -> not2.I2, C -> not3.I3;

CONNECT not1 -> circ2.I1, not2 -> circ2.I2, not3 -> circ2.I3;

CONNECT circ1 -> jk1.J, circ2 -> jk1.K, clk1 -> jk1.CLK;

MONITOR jk1.Q, jk1.QBAR;
```

A Appendix



							WEEK 2									
WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE	19	20	21	22	23	24	25	26	27	28
							T	F	S	S	M	T	W	T	F	S
1	**First Interim Report - Planning**															
1.1	Logic Description File	All	19/05/22	20/05/22	1	80%	🟦	🟦								
1.1.1	Syntax	All	19/05/22	20/05/22	1	100%	🟦	🟦								
1.1.2	Semantics and Errors	All	19/05/22	20/05/22	1	60%	🟦	🟦								
1.2	First Interim Report	All	19/05/22	22/05/22	3	0%	🟦	🟦	🟦	🟦						
1.2.1	Intro (Description of Approach)	Yash (Y)	19/05/22	22/05/22	3	0%	🟦	🟦	🟦	🟦						
1.2.2	Teamwork Planning	Richard (R)	19/05/22	20/05/22	1	70%	🟦	🟦								
1.2.3	Logic definition file examples + extra	Ioannis (I)	19/05/22	22/05/22	3	50%	🟦	🟦	🟦	🟦						
2	**"Names" module**															
2.1	Name strings to IDs	Y/I	23/05/22	24/05/22	1	0%					🟦	🟦				
2.2	Query function	Y/I	23/05/22	24/05/22	1	0%					🟦	🟦				
2.3	Testing: Names	R	24/05/22	25/05/22	1	0%						🟦	🟦			
3	**"Scanner" module**															
3.1	Symbol class	Y/I	24/05/22	25/05/22	1	50%						🟦	🟦			
3.2	Scanner class	Y/I	25/05/22	27/05/22	2	0%							🟦	🟩	🟩	
3.2.1	Scanner: initialiser	Y/I	25/05/22	27/05/22	2	0%							🟦	🟩	🟩	
3.2.2	Scanner: get_symbol	Y/I	25/05/22	27/05/22	2	0%							🟦	🟩	🟩	
3.2.3	Scanner: error handling	Y/I	25/05/22	27/05/22	2	0%							🟦	🟩	🟩	
3.3	Testing: Symbol	R	25/05/22	26/05/22	1	0%						🟦	🟦			
3.4	Testing: Scanner	R	25/05/22	27/05/22	2	0%							🟦	🟩	🟩	
4	**"Parse" module**															
4.1	parse_network	Y/I	27/05/22	29/05/22	2	0%									🟩	🟩
4.2	Testing: Definition files	All	28/05/22	29/05/22	1	0%										🟩
4.3	Parse: Network+Devices	Y/I	29/05/22	31/05/22	2	0%										
4.4	Testing: Parse	R	30/05/22	01/06/22	1	0%										
5	**"GUI" module**															
5.1	Preparation and Planning	All	23/05/22	25/05/22	2	0%					🟦	🟦	🟦			
5.2	GUI Design	R	25/05/22	28/05/22	3	0%							🟦	🟩	🟩	🟩
5.3	UserInterface Functionality	R	28/05/22	31/05/22	3	0%										🟩

Figure 1: Gantt Chart for project

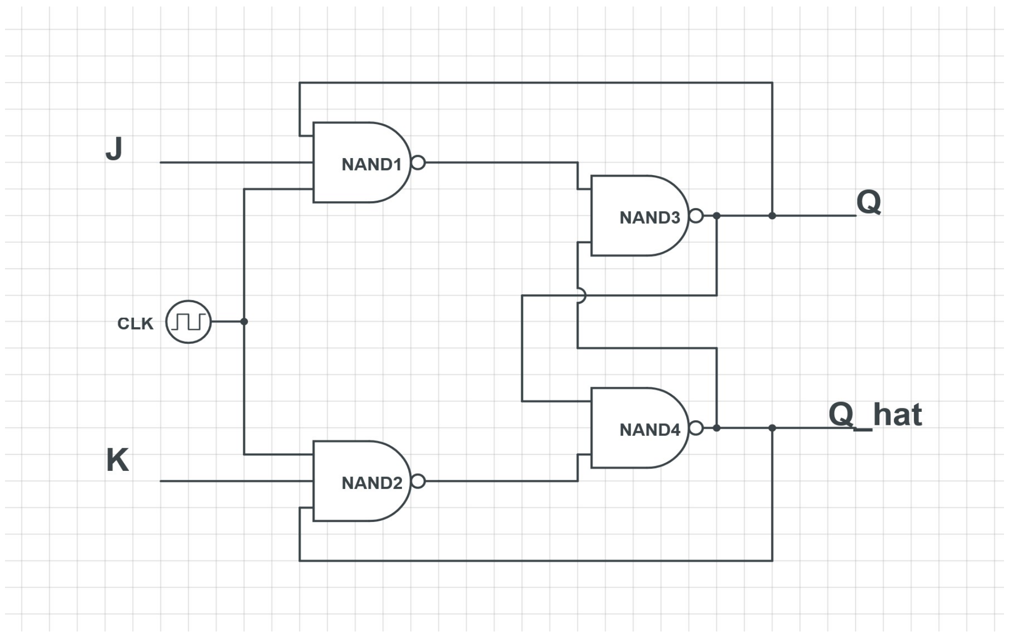


Figure 2: JK bistable

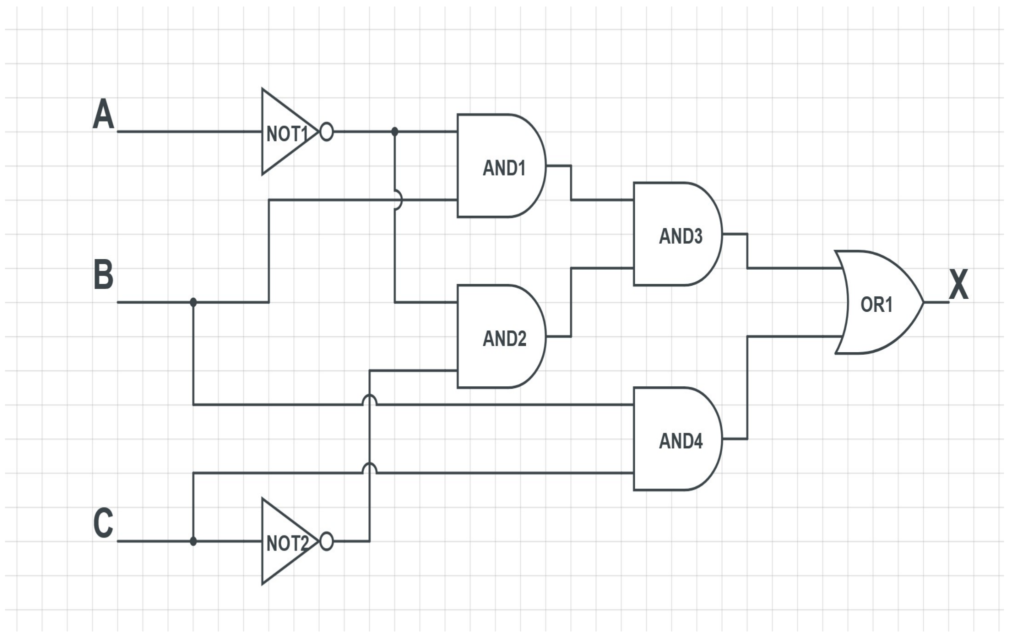


Figure 3: Complicated circuit

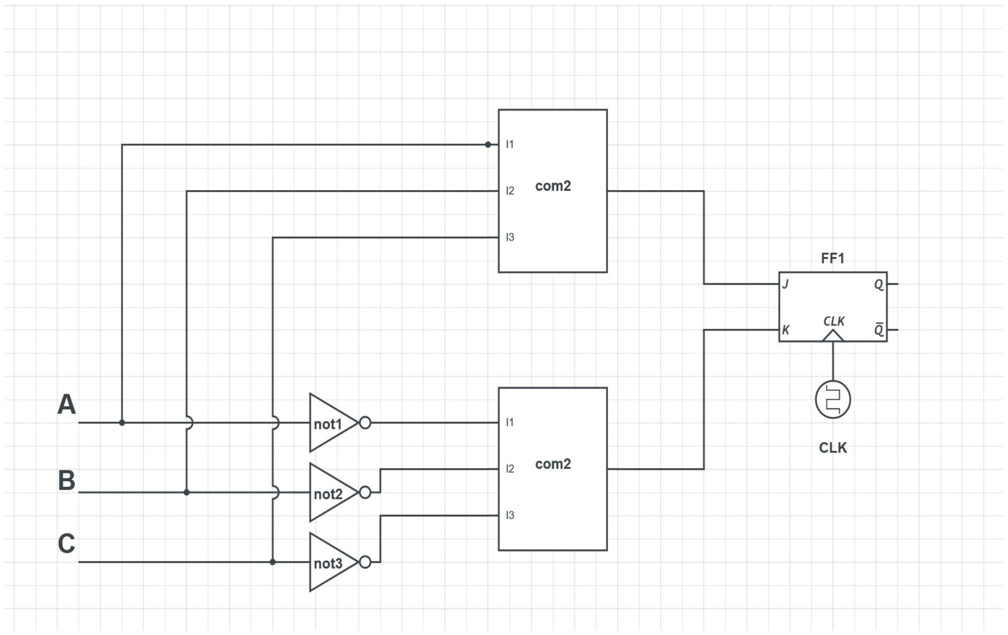


Figure 4: Extension circuit