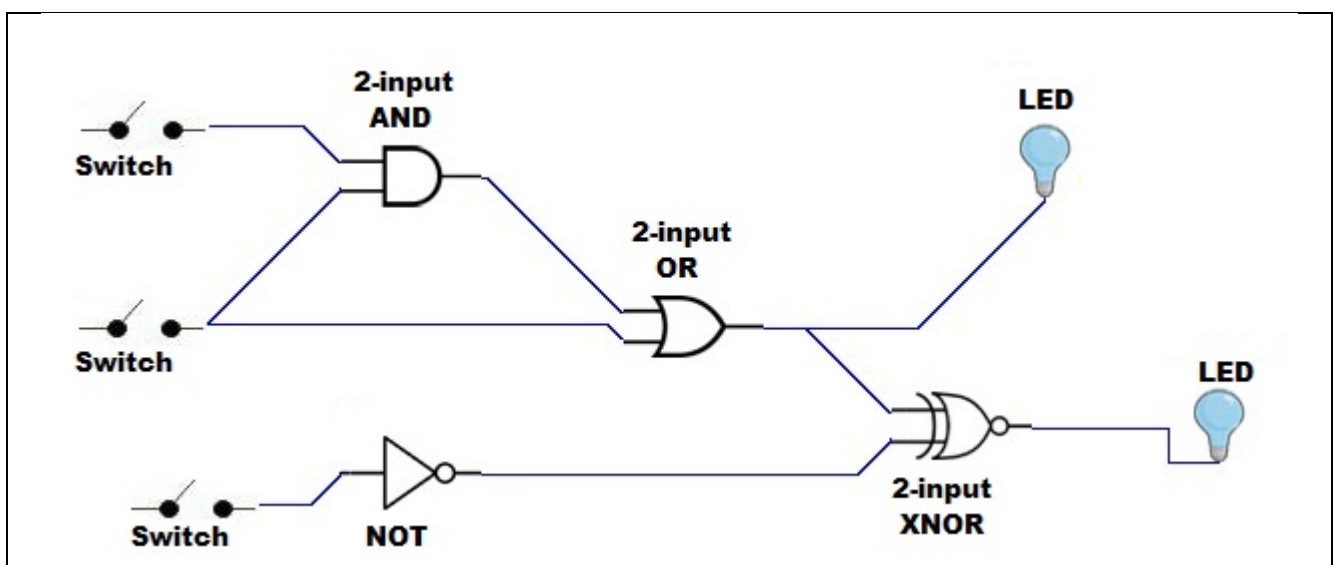# CIE202 Project

Logic Circuit Simulator

Introduction

A logic circuit is a collection of interconnected logic components, such as gates, switches, LEDs, etc. Each component has a set of input pins and a set of output pins. A connection can be created between any two components by connecting an output pin of a component to an input pin of another component. Gates, on the other hand, are components with one or more input pins and only one output pin. Each circuit component manipulates the values applied to its input pins to calculate corresponding output. Two special components are the switch and the LED. The switch is a component with no input pins but has one output pin. The LED on the other hand has one input pin and zero output pin.

Complicated logic circuits can be very hard to trace, that is why computer-aided design of logic circuits is necessary. One of the very powerful tools is a simulator, which simulates the operation of an entire logic circuit and predicts the outputs based on the inputs.

Your Task:
You are required to implement a simple logic circuit designer/simulator. Your implementation should be in C++ code. You must use ***object oriented programming*** to implement this application. Your application should help users draw a logic circuit using different components, connect them with connections, add switches to control inputs and LEDs to monitor outputs, and simulate the circuit operation. The application should accept input from the user as mouse clicks or key strokes and then take actions according to the input.

NOTE: The application should be designed so that the types of components and types of operations can be easily extended.

The rest of this document describes the details of the application you are required to build.

Project Schedule

Phase	Deliverables	Due Week	Weight
Phase 1 (Team grade)	UI Classes and basic gates and actions	Week 12	35%
Phase 2 (Team grade)	Final Project Delivery	Week 17	35%
Individual Participation	Individual Discussion	Week 17	25%

Note: At any delivery:
One day late makes you lose 1/4 of the grade.
Two days late makes you lose 1/2 of the grade.

Requirements, Grading, and Evaluation Criteria

The application should support two modes of operation; design mode and simulation mode (The default is the design mode)

[I] [43%] Design Mode Operations:

1- **[8%, phase1]** *Add* a new component to the circuit. This includes
   - ❑ Adding a new gate: The application should support the following gates
        i. Inverter gate
        ii. 2-input AND, OR, NAND, NOR, XOR, XNOR gates
   - ❑ Adding a new **switch** (a switch is a component with one output pin and no input pins)
   - ❑ Adding a new **LED** (a LED is a component with one input pin and no output pins)
   - ❑ **Each "Add" operation requires a derived <u>Action</u> class and a derived <u>Component</u> class. (See "Main Classes" section)**

2- **[3%, phase1]** *Connect* two components: this means to create a connection from an output pin of a component to an input pin of another component. To connect one output pin to many input pins, a separate connection should be created between the output pin and each of the input pins.

3- **[2%, phase1]** *Select*/*Unselect* **component/connection**. When the user clicks on one of the components, it should be highlighted.

4- **[2%, phase1]** *Label* **component/connection**.

5- **[2%, phase1]** *Edit* **component/connection**: To edit a component is to edit its label. To edit a connection is to edit its label or to change its source or destination pins

6- **[3%]** *Delete* **component/connection**: when deleting a component all its connections should be automatically deleted. Also, deleting a separate connection should be supported.

7- **[5%]** *Multiple-Delete***:** User can select multiple components (including connections) to delete them all

8- **[9% = 3x3]** *Copy-Cut-Paste* a component (not applicable for connections)
   - ❑ Copying a component copy only the component without its connections.
   - ❑ Cutting a component is to erase the original component with its connections then paste the component with no connections.

9- **[3%, phase1]** *Save* a circuit to a file (see file format section).

10- **[3%, phase1]** *Load* a circuit from a file (see file format section).

11- **[1%]** *Switch to simulation mode.* (only if the circuit is fully and correctly connected)

12- **[2%, phase1]** *Exit* the application: application should perform necessary cleanup before exiting.

Important:

   Application should limit user drawings to the design area only. Drawing on the toolbar or one the status bar is NOT allowed.

[II] [17%] <u>Simulation Mode Operations:</u>

Operations supported by this mode are:

1- **[1%]** *Circuit Validation***:** To simulate a circuit it must be fully connected and no pins are left floating. This operation runs automatically when the user tries to switch to simulation mode. User will be allowed to switch to simulation only if this operation returns valid.

2- **[10%]** *Simulate circuit***:** user can change the circuit inputs by changing the status of input switches. Any changes in the circuit inputs in this mode should be reflected at the output LEDs.

3- **[5%]** *Module Class***:** Each team should pick ONE module from appendix A below and **create a class** to implement it. The Module class should be built using ONLY classes mentioned above in "Design Mode Operations" points 1&2. Each team should pick a different module (this will be organized by instructors).
   Notes:
   ▪ Module class should be derived from Gate class
   ▪ The module should have an icon in the design menu bar to enable the user to add it as part of the circuit and to be part of the simulated circuit as well.

4- **[1%]** *Switch back to Design Mode*

Important Note:
   ❑ **Each operation should have a *corresponding action class*. Each "Add" operation should have a corresponding action class.   (see "Main Classes" section)**

[III] [10%, phase1] <u>User Interface Requirements:</u>

You are given a code that contains UI class partially implemented. You should complete it as follows:

1- **[3%]** Complete Menu drawing function:
   `UI::CreateDesignToolBar()` **and** `UI::CreateSimulationToolBar()`
   UI class should create two tool bars (menus); design tool bar and simulation tool bar with **full** icons in each of them.

2- **[2%]** Complete function `UI::GetUserAction()`
   UI class should detect all possible actions according to the coordinates clicked by the user.

3- **[5%]** Add a new drawing function for each component ()
   ▪ Similar to function `UI::DrawAND2( )`
   ▪ A function for each compoenent (DrawOR2( ), DrawSwitch( ), DrawLED( ),.. etc).
   ▪ Draw each component in all possible cases; normal, highlighted.
   ▪ DrawConnection function

[IV] [30%] <u>Individual Participation</u>**:**

Each member must be responsible for some part of the code and must answer some questions showing that he/she understands both the program logic and the implementation details.

Note: we will reduce the individual grade in the following cases:
   ❑ Not working enough
   ❑ Neglecting or preventing other team members from working enough

[10%] Bonus Operations:

1-   **[2%]** *Move* **a component**: user can drag a component to move it. All its connections should be redrawn during moving.

2-   **[3%] Complex Circuit Blocks**: User can enter "**Module Design Mode**" where he can create a valid circuit and with any number of input and output lines. Then he can convert it into one block and saves it so that he can use it later as a block as a part in a bigger circuit.

3-   **[3%] Undo/Redo** different actions

4-   **[2%]** *Create Truth Table*: The application must be able to create the truth table of the designed circuit. It should be able to display the truth table for up to 5 inputs. For circuits with inputs greater than 5, the truth table should be saved to a text **file**.

General Evaluation Criteria for any Operation:
1.   **Compilation Errors → MINUS 50%** of Operation Grade
   ❑   The remaining 50% will be on logic and object-oriented concepts (see **point no. 3**)

2.   **Not Running** (runtime error in its **basic functionality**) → **MINUS 40%** of Operation Grade
   ❑   The remaining 60% will be on logic and object-oriented concepts (see **point no. 3**)
   ❑   If we found runtime errors but in corner (not basic) cases, that's will be part of the grade but not the whole 40%.

3.   **Missing Object-Oriented Concepts → MINUS 30%** of Operation Grade
   ❑   **Separate class** for each item and action
   ❑   Each class does its **job**. No class is performing the job of another class.
   ❑   **Polymorphism:** use of pointers and virtual functions
   ❑   **See the "Implementation Guidelines" section which contains all the common mistakes that violates object-oriented concepts.**

4.   **For each corner case** that is not working → **MINUS 10% to 20%** of the Operation Grade according to instruction evaluation.

Notes:
The code of any operation does NOT compensate for the absence of any other operation; for example, if you make all the above operations except the delete operation, you will lose the grade of the delete operation no matter how good the other operations are.

Main Classes

Because this is your first object oriented application, you are given a **code framework** where we have **partially** written the code of some of the project classes. For the graphical user interface (GUI), we have integrated an open-source **graphics library (CMU graphics Library)** that you will use to easily handle GUI.

You should **stick to** the given design and complete the given framework by either: extending some classes or inheriting from some classes (or even creating new base classes but after instructor approval)

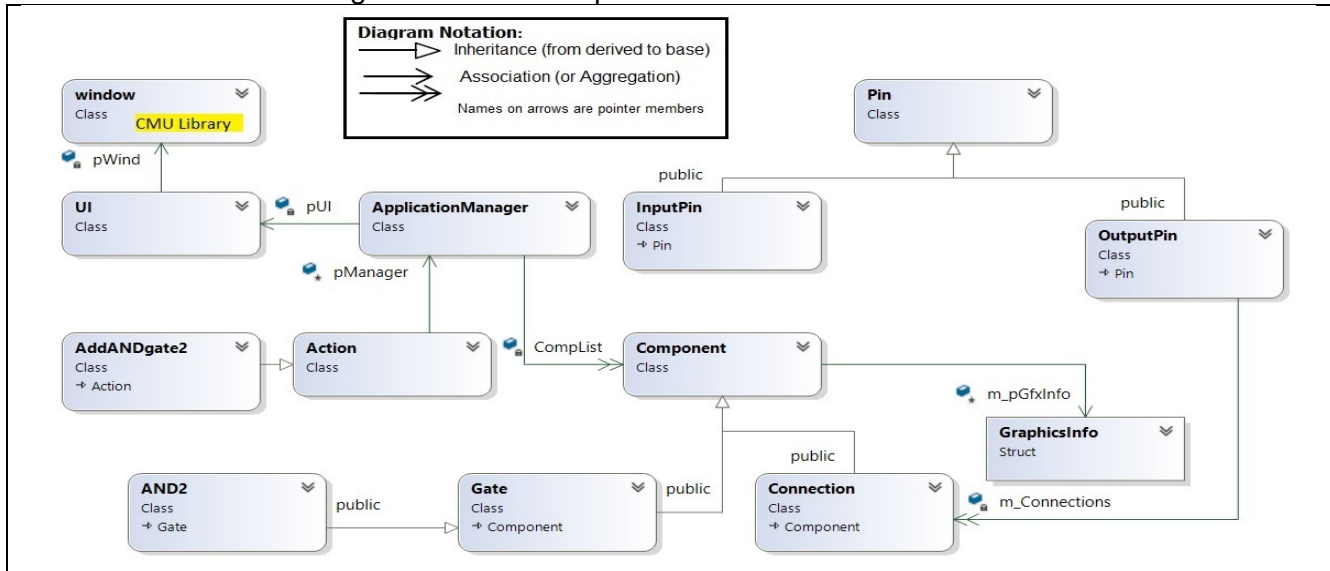Below is the class diagram then a description for the basic classes.

Figure 1 – Class Diagram of the Application

UI Class:

This is the only class that can interact with the user. All user inputs and program outputs and circuit drawing are done through this class. If any other class wants to read/display anything to the user, it must call one member function of this class.

You will extend this class and add member function to draw different components and show simulation results.

ApplicationManager Class:

This is the **maestro** class that controls everything in the application. It has a list of all components and also pointer to UI object. As its name shows, its job is to **manage** other classes **not** to do other classes' jobs. So it just instructs other classes to do their jobs.

Pin Class:

This is the base class for input and output pins (**InputPin** class and **OutputPin** class).

Component Class:

This is the base class for all types of circuit components (switches, gates, LEDs, and connections). To add a new component (a new gate for example), you must **inherit** it from this class. Then you should override its virtual functions. You can also add more details for the class Component itself if needed.

Action Class:

This is the base class for all types of actions (operations) to be supported by the application. To add a new action (Save action for example), you must **inherit** it from this class. Then you should override its virtual functions. You can also add more details for the class Action itself if needed.

Implementation Guidelines

❑ In general, each user operation is performed in 4 steps
- a. Get user action type.
- b. Create suitable action object for that input
- c. Execute action.
- d. Reflect action to the Interface. (update the interface)

❑ Polymorphism:
- a. Nearly all the parameters passed/returned to/from the functions should be pointers to be able to exploit polymorphism and virtual functions.
- b. Many class members should be pointers for the same reason in part (a).

❑ Incremental implementation:
- a. Compile each class separately first before compiling the entire project.
- b. If class B depends on class A, don't move to class B before making sure that all errors in class A code have been corrected.

❑ Classes responsibilities:
Each class should perform tasks that are related to its responsibilities only. No class performs tasks of another class.

❑ The only class that have a **direct** access to the graphics library is the **UI** class. Any class that needs to read input from the input window should do that by calling functions of the class Input. Similarly, any class that needs to draw or print on the output window should do that by calling functions of the class Output. In summary, **any interaction with the user interface should be done through the UI classes.**

❑ All gates classes **should** be implemented **similar** to the **AND2** class implementation. You **must inherit from Gate** class.

❑ Save/Load
- a. User can save/load incomplete circuit.
- b. Save/load **function** is a **virtual** function in the class Component. Each derived class should override this function to save/load itself.
- c. Save/Load **Action** just opens the file and then calls ApplicationManager::Save/Load function.
- d. ApplicationManager then saves/loads the list of statements/connectors by calling save/load function of each Component.

❑ Work load must be distributed among team members. A good way to divide work load is to assign some classes to each team member. A first question to the team at the project discussion and evaluation is "who is responsible for what?". An answer like" we all worked together" is a failure.

Example Scenario

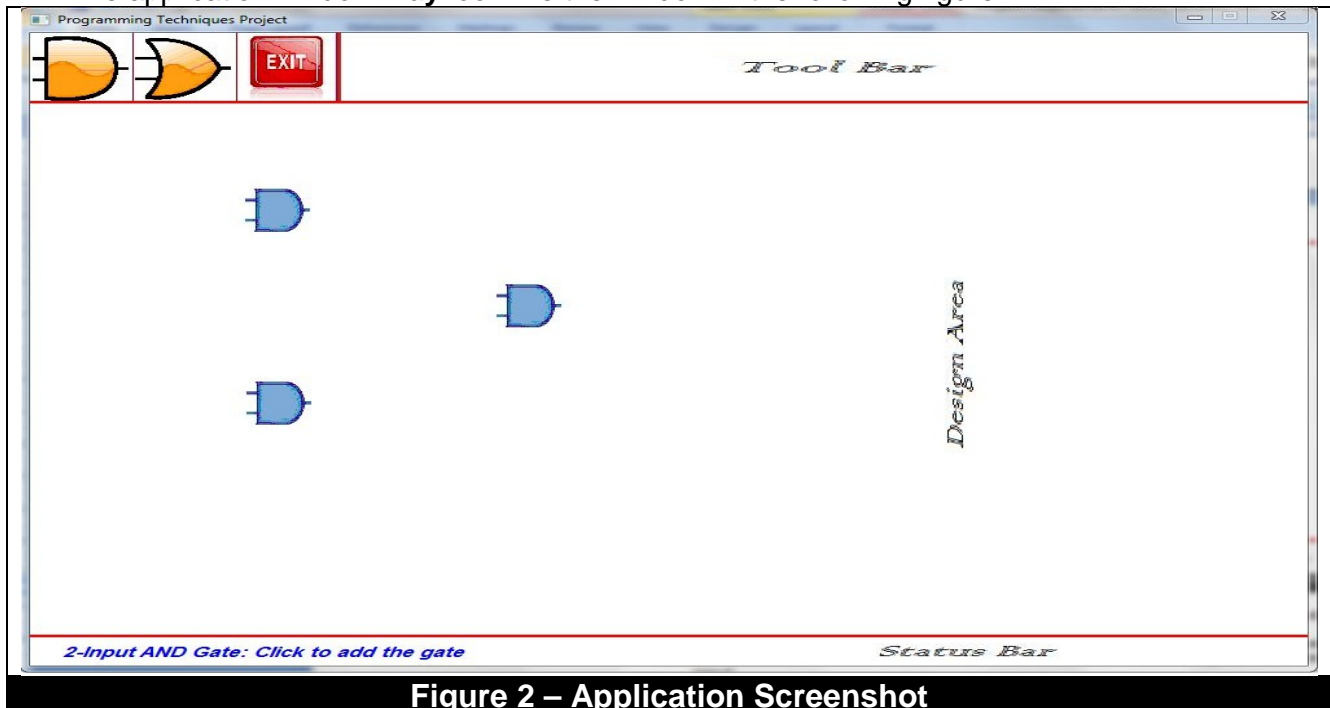The application window **may** look like the window in the following figure

Figure 2 – Application Screenshot

Here is an example scenario for adding an AND gate and drawing it on the output window. It is performed through the four main steps mentioned in the previous section (see main( ) function in the given framework code)

Step I: Get user action

1- The **ApplicationManager** calls the **UI** class and waits for user action.

2- The user clicks on the "AND gate" icon ⬡ in the toolbar

3- The **UI** class checks the area where the user clicked and recognizes he wants to add AND gate to the circuit. It returns **ADD_AND_GATE_2** (a constant indicating the required action) to the manager.

Step II: Create a suitable action object

4- **ApplicationManager::ExecuteAction** is called. It creates an object of type **AddANDgate2** action and calls **AddANDgate2::Execute** to execute the action.

Step III: Execute the action

5- **AddANDgate2::Execute**
   a. Calls the **UI** class to get the gate position from the user. Here, to print a message to the user, **Execute** calls the UI::PrintMsg function.
   b. Creates a Component of type **AND2** and asks the **ApplicationManager** to add it to the current list of Components. (by calling AddComponent)

Note: At this step, the action is complete but it is not yet reflected to the user interface.

Step IV: Reflect the action to the Interface (function **ApplicationManager::UpdateInterface)**

6- **ApplicationManager::UpdateInterface** calls the virtual function **Component::Draw** for each Component. (in this example, function AND2::Draw is called)

7- **AND2::Draw** calls **UI::DrawAND2** to draw AND gate.

File Format

Your application should be able to save and load a circuit from a simple text file. In this section, the file format is described together with an example and an explanation for that example. The application should enable the user to create a new circuit or to load an existing circuit. If the user wants to load an existing circuit, the application loads the circuit from the required file. Otherwise, a new empty window is created.

- **File Format**

Number_of_Components			
Comp_1_Type	Comp_ID	Label	Component_Graphics_info
Comp_2_Type	Comp_ID	Label	Component_Graphics_info
…………………………………………………..			
Comp_n_Type	Comp_ID	Label	Component_Graphics_info
Connections:			
Number_of_connections			
Source_Comp_ID	Target_Comp_ID	Pin_number	

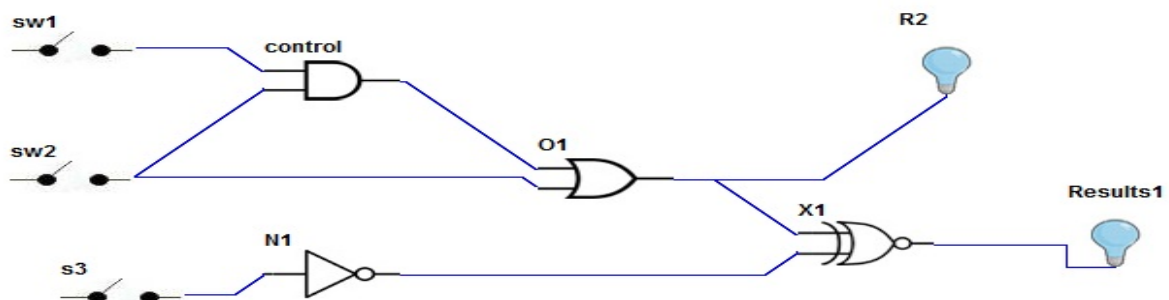- **Example:** The circuit shown in figure 3.a below is represented by the file in figure 3.b

Figure 3.a - Circuit Diagram

9				
SWTCH	1	sw1	90	154
SWTCH	**2**	**sw2**	**90**	**227**
AND2	4	control	170	169
OR 2	**9**	**O1**	**290**	**225**
SWTCH	13	s3	97	330
XNOR	**6**	**x1**	**352**	**270**
LED	8	R2	406	182
NOT	**5**	**N1**	**170**	**302**
LED	7	Result1	473	297
Connections				
9				
1	4	1		
2	**4**	**2**		
2	9	2		
13	**5**	**1**		
4	9	1		
5	**6**	**2**		
9	8	1		
9	**6**	**1**		
6	7	1		

Figure 3.b - File that represents the circuit

- **<u>Explanation of the above example</u>**
  Here is the example and the explanation of each line

```
9  //Circuit has 9 components


SWTCH   1     sw1    90     154
//Switch, ID=1, label="sw1", its rectangular area corner is (90,154)


SWTCH   2     sw2    90     227
//Switch, ID=2, label="sw2", its rectangular area corner is (90,227)


AND2    4     $      170    169
//2-input AND, ID=4, label="control", its rectangular area corner is (170,169)


……………..
…………….

LED     8     R2     406    182
//LED, ID=8, label="R2", its rectangular area corner is (406,182)


……………………
……………………

Connections   //Start of connections part
9        //no. of connections
1        4     1 //Component 1 (Switch) is connected to Comp 4 (AND2) at pin# 1
2        4     2 //Comp 2 (Switch) is connected to Comp 4 (AND2) at pin# 2
And so on….
```

<u>*Notes*</u>:

- ❑ You can select any IDs for the components. Just make sure ID is unique for each component.
- ❑ You are allowed to add some modifications to this file format if necessary. But before adding such modifications, get the approval from your instructor.
- ❑ To indicate every component type in the file, you can use numbers instead of text. You can give each component type a number (use **enum**).

Project Phases

1- Phase 1

The required parts of this phase are highlighted in the section labeled "Requirement, Grading, and Evaluation Criteria" above.

Deliverables*:*
Each team should submit a zipped file that contains
- Info.txt file: (Information about the team: name, IDs, team email)
- The code and resources files for required parts.
- Submitted file name must be: **CIE202_project_teamNumber**

2- Phase 2

In this phase, the remaining classes should be implemented. Start by implementing the base classes then move to derived classes. To save time, work should be divided between team members.

Deliverables*:*
(1) **Workload division**: a **printed page** containing team information and a table containing members' names and the classes each member has implemented.
(2) Submit one zipped file that contains the following
   a. ID.txt file. (Information about the team: name, IDs, team email)
   b. The workload division document.
   c. The project code and resources files.
   d. Sample circuit files: Three different circuits. For each circuit, provide:
      i. Circuit text file
      ii. Circuit screenshot for the circuit generated by your program
      iii. Screenshot of one case of circuit simulation

Submitted file name must be: **CIE202_project_teamNumber**

Appendix A – Modules

This appendix contains list of modules described by logical expressions.
Each module has 5 input lines (A, B, C, D, and E) and 2 output lines (F0 and F1).
We used our own notation to represent different gates:

Notation used:

Symbol	Gate
&	**2-input AND Gate**
!	**2-input NAND Gate**
+	**2-input OR Gate**
-	**2-input NOR Gate**
^	**2-input XOR Gate**
~	**2-input XNOR Gate**

List of Modules:

Module #1.	F0= ((C ! E) - (E ~ D)) + ((A & E) & (C + B))
	F1= ((A + D) - (E & D)) ! ((B & C) ~ (B + E))
Module #2.	F0= ((E - A) ^ (E + D)) ^ ((C & B) - (A ! D))
	F1= ((B ^ C) - (A ! C)) ! ((E ^ D) + (E ! C))
Module #3.	F0= ((E + B) & (C + B)) ~ ((A ~ C) ^ (A - D))
	F1= ((D - A) & (E ~ C)) - ((B & E) & (A ! C))
Module #4.	F0= ((B ^ E) + (D ^ C)) + ((B ! A) ^ (D - E))
	F1= ((C ~ B) & (E + A)) - ((D ^ E) & (A - B))
Module #5.	F0= ((A ^ B) - (D & A)) ~ ((E ! C) & (B ^ C))
	F1= ((B + E) & (C ! E)) ! ((D ! E) ^ (C ! A))
Module #6.	F0= ((A ~ B) ^ (C - E)) + ((E ^ A) ^ (D ^ C))
	F1= ((A & B) ~ (C ~ A)) ! ((D ~ E) + (C + B))
Module #7.	F0= ((E ^ B) & (D ! A)) - ((C & A) + (C + D))
	F1= ((A - C) ! (E - D)) - ((E ^ B) - (B + A))
Module #8.	F0= ((E + C) ! (B - D)) ^ ((C ^ A) ! (C & B))
	F1= ((C ! D) + (A & B)) + ((D ! E) - (C - E))
Module #9.	F0= ((E - C) ~ (B + D)) ~ ((B + A) + (B + D))
	F1= ((E - D) & (B ~ C)) & ((A & B) ~ (A ! E))

Module #10.	F0= ((C & E) & (E + D)) ! ((A & B) - (B & A))
	F1= ((E ^ D) ! (A - B)) ! ((C & B) - (A ~ B))
Module #11.	F0= ((E ! A) - (D ~ B)) ! ((C ~ A) ^ (B ~ D))
	F1= ((C - D) ~ (A ! E)) + ((B & E) ! (B + D))
Module #12.	F0= ((D ^ C) - (B - A)) + ((A & D) - (E ! A))
	F1= ((E + B) - (A ^ D)) & ((B ! A) + (C ! E))
Module #13.	F0= ((B + A) ^ (A & E)) ! ((C - E) ^ (A ~ D))
	F1= ((C + E) & (B - A)) ! ((E ! D) ~ (D + C))
Module #14.	F0= ((A ! E) & (B ~ D)) - ((B ~ A) - (C + D))
	F1= ((C - E) ! (A - E)) ! ((D - B) - (A ^ B))
Module #15.	F0= ((C + B) - (C ^ D)) & ((D & A) & (E ~ A))
	F1= ((A ~ E) + (C ^ D)) & ((E - B) ^ (C - B))
Module #16.	F0= ((B ^ A) + (D ^ C)) ! ((B + A) & (E & A))
	F1= ((C & A) ^ (C ! E)) & ((D & A) - (D & B))
Module #17.	F0= ((B ! D) - (C ! E)) & ((A ^ B) & (C + E))
	F1= ((C - B) - (E & C)) ~ ((A ^ C) ^ (E & D))
Module #18.	F0= ((A & C) + (D ! C)) ! ((B ~ E) + (C - E))
	F1= ((A ^ E) + (C ~ B)) & ((E ! D) & (C ! E))
Module #19.	F0= ((C & E) ^ (B ~ A)) ~ ((C + B) ! (A + D))
	F1= ((D ^ E) & (E ^ A)) ^ ((B ~ C) & (C ^ D))
Module #20.	F0= ((C ~ B) ~ (B - A)) - ((D & C) ! (E ~ A))
	F1= ((D & E) ~ (C ^ A)) & ((C ~ B) & (C + D))
Module #21.	F0= ((A - B) ^ (C ~ B)) ~ ((A & D) ^ (E & D))
	F1= ((D - A) ~ (D - A)) ! ((E ~ C) ~ (B ~ A))
Module #22.	F0= ((D ~ A) ! (B ~ E)) ~ ((E + C) ^ (A & C))
	F1= ((E + D) - (E - B)) + ((D ~ C) & (A - B))
Module #23.	F0= ((A + D) ! (E - B)) & ((C ^ A) ^ (A - C))
	F1= ((C + E) + (C ^ B)) & ((C ^ A) + (B - D))
Module #24.	F0= ((D ! A) ! (D ~ C)) + ((D & E) & (B - A))
	F1= ((E ^ A) ~ (D ! B)) & ((B ! C) ^ (A ^ D))
Module #25.	F0= ((A ~ D) & (B ^ D)) & ((E - C) - (D ! A))

	F1= ((E - A) - (A & C)) ~ ((C - D) ! (C & B))
Module #26.	F0= ((E & C) - (B - C)) & ((A ^ B) + (D ^ E))
	F1= ((A ~ D) ~ (B - D)) + ((A & C) ^ (B - E))
Module #27.	F0= ((B ! D) + (C + D)) ~ ((B ^ E) ! (A - E))
	F1= ((D - C) - (D + E)) - ((C - B) & (A ! D))
Module #28.	F0= ((E - B) + (B - A)) ^ ((A ! D) ! (C ! E))
	F1= ((A & C) & (D ~ A)) ~ ((E ! C) ^ (D ~ B))
Module #29.	F0= ((D ~ C) - (D ~ B)) ^ ((B + D) ! (A & E))
	F1= ((A ! C) ! (E + B)) ! ((C + E) - (A + D))
Module #30.	F0= ((D - B) + (D - E)) ! ((E & A) & (C - B))
	F1= ((E ~ D) ^ (A + B)) & ((C ^ A) & (A & C))
Module #31.	F0= ((D + E) ! (B - A)) ^ ((C - E) + (A - C))
	F1= ((E + D) ! (A & C)) ~ ((E ~ B) ! (E ! C))
Module #32.	F0= ((C - E) ! (D + A)) & ((D - C) ! (E & B))
	F1= ((A - C) & (C - A)) & ((C ~ B) + (D + E))
Module #33.	F0= ((D ! A) ! (C - A)) ^ ((B ^ E) & (E + D))
	F1= ((E ^ D) + (B ^ E)) ^ ((E & A) & (C + D))
Module #34.	F0= ((C ~ A) ^ (D & B)) ! ((D ! B) - (E ^ D))
	F1= ((E ~ A) - (B ~ C)) & ((A ^ E) + (A ^ D))
Module #35.	F0= ((A + D) ! (A + B)) & ((C ^ E) - (C ! E))
	F1= ((B ! D) ~ (E ! C)) - ((C ~ E) ~ (A ^ B))
Module #36.	F0= ((B ~ A) ! (B ^ E)) + ((E & A) ! (C & D))
	F1= ((D - E) - (C + A)) - ((C ~ E) + (B & C))
Module #37.	F0= ((C ^ E) ~ (D - A)) & ((A - C) & (D ^ B))
	F1= ((D - A) ^ (E & B)) - ((E ^ D) & (C - B))
Module #38.	F0= ((D & A) ~ (E ~ C)) ! ((E + B) ! (A ^ E))
	F1= ((B ! E) ^ (A & D)) & ((B + C) & (A & C))
Module #39.	F0= ((B + D) ! (A ^ D)) ! ((B ~ E) + (A ! C))
	F1= ((E - D) ~ (C ^ D)) - ((A ~ B) + (A & C))
Module #40.	F0= ((C - D) + (D & A)) ^ ((B ^ D) ! (C - E))
	F1= ((C & E) & (D - C)) - ((B ^ E) ! (E - A))