LOGIC GATE SIMULATOR

Highdown School and Sixth form center – 51319
Andrew Foot – 3675
AQA 7517 Computer Science Coursework 2020

Contents

Analysis

This is aimed at educational purposes and to teach younger years the basics and understand of how logic gates and logic circuits work. For the computer science department, they wanted a more sophisticated system that would allow more advance logic circuits like flip flop. They also required basic systems like save and open previous files and a help guide on how to use the program for the students.

This project required a good understanding of how to use WPF C# as this was my programming choice. I've had to come familiar with the XAML and how the classes in WPF worked. Because of the requirements I had to research into multithreading. This was convivence due to C# have a good support for it OOP and multithreading.

On research a product like the website logic.ly would provide a good learning platform for the school without the $59 price tag. It would also provide software that can be accessed without internet and requiring 30 students to search the web all to find the same website.

Interview

I've interviewed my computer science teacher (Mr Stephenson) on what functions the program should hold. I asked class mates on the design to the GUI and how it should interact. This allowed me to model and develop based directly on their requests and fulfil their needs.

Mr Stephenson wanted a clear and easy to use simulator with a file tools to plan lessons beforehand.

Features:

- Visible changes to let the user know where the actions are being taken
- Change speed of the simulator
- File save and open features
- Real time calculations
- Zoom in and out features
- Drag and Drop
- Interactive canvas

General Objectives

Achieve a fully working logic gate circuit simulator that will work flawlessly and fulfil the required needs of the departments. As a minimum it should incorporate all of the gates on the physics and computing specifications in addition to allowing multiple outputs from each gate.

It should be a responsive and lag free programme even on the worst of machines. The size of the screen shouldn't be an issue and resizes to whatever aspect ratio.

No known bugs be present in the end product.

Data sources & Destination

All data for the project that will be stored will be based on the class called Gate_class. The list of object will be saved so that the user can load it back up later if required. I believe this doesn't oppose a security threat as the data is not autonomous and only storing information on the formation and data values of the logic gates and how they're placed. Everything else will be reconstructed so that the file size is a small as possible. The file type will be JSON as they're perfect for storing class and are easy to read again.

Programmed in C#

C# is a powerful and useful language, when it comes to UI's WPF supports a wide range of projects that can be made. C# has a great support behind it and a lot of resources. This proved important as this was the first time using multithreading.

In WPF all UI elements are their own class, this allows you to inherit the class and create custom UI classes throughout your program. Because of this you can override existing methods that exist in the UI class. Giving you direct access to the event handlers that are fired by the UI.

C# for me is also a lot neater and better structured due to you have more control over other languages. This is a curse and a blessing as it means you know what's precisely happening at the moment but you're not helped through it and have to do it all yourself.

Justification of chosen solution

My software is bespoke as it was made esspically for Mr Stephenson. It's programmed using C# in WPF so that the UI is to a professional standard. It's a language I've learned and can achive a high standard in.

Pros cons of 2 other languages

It is a language I know well and suitable for the project due to it's OOP capabilities. It also gives a lot of control to the programmer which was required and will be talked about more. Mr Stephenson also wanted file saves which is more compatible when it's easy to convert classes to JSON. I programmed it inside of Visual Studio 2019 using windows presentation foundation.

Limitations

Due to how my UI is a canvas and not a box grid I could not implement a successful path finding algorithm for the wires that connects the gates. This was seemed reasonable by the computer science department as a task like that is implausible to add due to the infinite possibilities and hard to extrapolate data. I've worked around this by adding different colours to the wires so that it's easier for the eye to track.

Methods

I'm using methods that I've learnt in lesson and olline. These consist of searching algorithms like linear search and area checking.

I've needed to look into how methods work with multi-threading and the completely different rules that are brought in when using 2 cores. I have to deal with linking up varaibles and learning dispacters.

For the program it requires great knowlage of OOP and the C# Langauage to achive an efficient outcome.

Further research

Looked into the limitations of the C# and WPF language. The fact that C# doesn't support multiple inheritance means that I have to use interfaces. Upon research I found out that interfaces can be used to link 2 classes variables together.

Prototyping

I have 3 itterations of my project all with different level of coding. I started the first program just working out how things would layout and what would needed to be changed. This gave me insite into the problem at hand and was a good starting point that allowed me to improve on the second and third programs.

The first program gave me a complete solution. This allowed me to get feedback and reviews on what could be changed. This provede to be substanstial as the main HCI features are based on these feedback.

For the second program it was about improving the first project in the areas that lacked the quality that I expected. Most of the code created here was used in the finally design. The problem was that I hadn't researched full into the project and left the main structure of the code to a low standard.

This was fixed in the third iteration of the project. As I employed inheritance of WPF classes. This meant the structure of the code is more enclosed to the class. It also removed usless classes that before couldn't have been implemented into the main class.

Design

Input, output, process and storage

The user experience (UX) is an important aspect of modern design and I have considered from the start the interactions users will expect to find in the LGS interface, this includes common conventions and unique methods.

Input:

- Left click on UI objects, this is the main interaction of the program is will be controlled by mainly the mouse.
- scroll wheel, this is for zooming in and out for the program.
- mouse move, the program will use the mouse position to determen what the program should do.
- Button click, this is for each access of linking methods to an action by the user.
- Drop down menu, gives a compact and easy solution for different avinuos for the user to do something.
- textbox, the user can input a string and the program will then run a function when textbox contain changes so that the program is dynamic.
- Read Json file, allows for files and storage of setup of the program to be added to the program.

Process:

- Mouse move, the program is dynamic so it will react to each input you do, this requires mouse move to update the program each time.
- dynamic and trackable objects on a canvas, the data behind the program needs to match the information on the screen, this is why data binding is needed for the data to be in sync with each element of the program.
- dynamic and trackable UI Elements on a canvas, because the canvas doesn't use a grid-based system the program should work for an infinite value of inputs.
- Transfer object between canvases, the switch between different areas of the program needs to be seamless and memory efficient.
- Save and read files, loading and creating a file needs to be easy and secure, it needs to be repeatable and check for edits to the file.
- Scaling canvases, for the zooming in and out the rescaling should change everything in the canvas and for the program to adjust to the new size so that every method works to the new scale.
- Translating canvases, Visually the canvas needs to move but also each method should recognise the change of location.

Here I consider both the physical storage requirements of the project and the objects produced.

Storage:

- In a worst case scenario the LGS can manage 272 gates, with 1kb per gate, even with overhead the program will be require less than ____.
- The program will need to be stored on the device from which it is used, there is currently no online functionality planned. The program will be distributed as an executable, the source code will only be provided to the client.
- List of objects, all data is group together for easy access and readability.
- c# variables, require understandable names and declaring the variable should be in the lowest level that is possible.
- Json Files, this should match the format of the OOP class so that there is easy decode and creating, security shouldn't be a problem but should be added for difficulty to editing of the file.

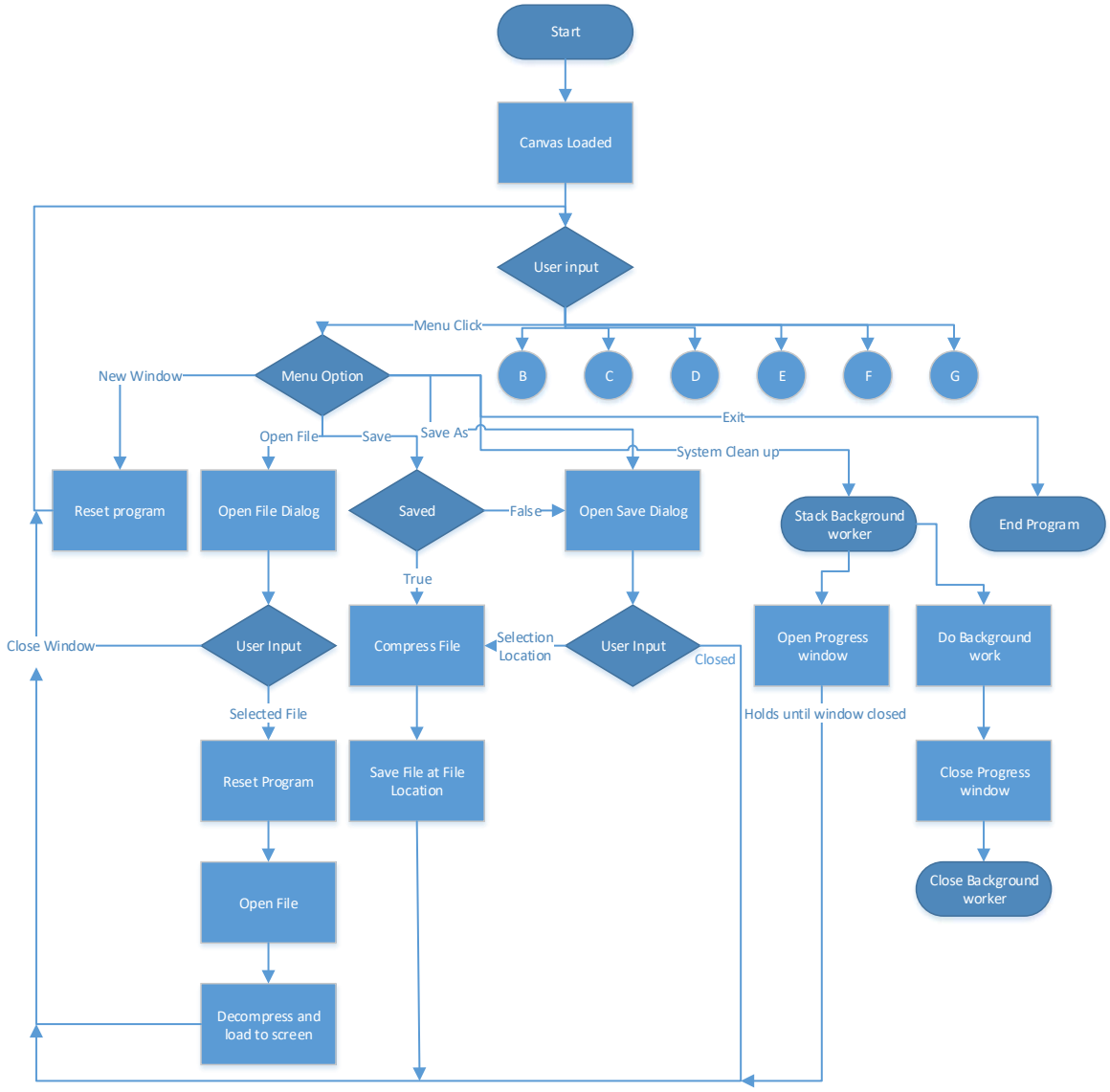
Output:

- Graphical UI, using the WPF format for the basic UI objects so that development is easy and readable due to large amount of resources and popularity.
- colour coded action, for easy understanding of the GUI for the user to follow the program.
- Create Json file, formatting should match the program data.

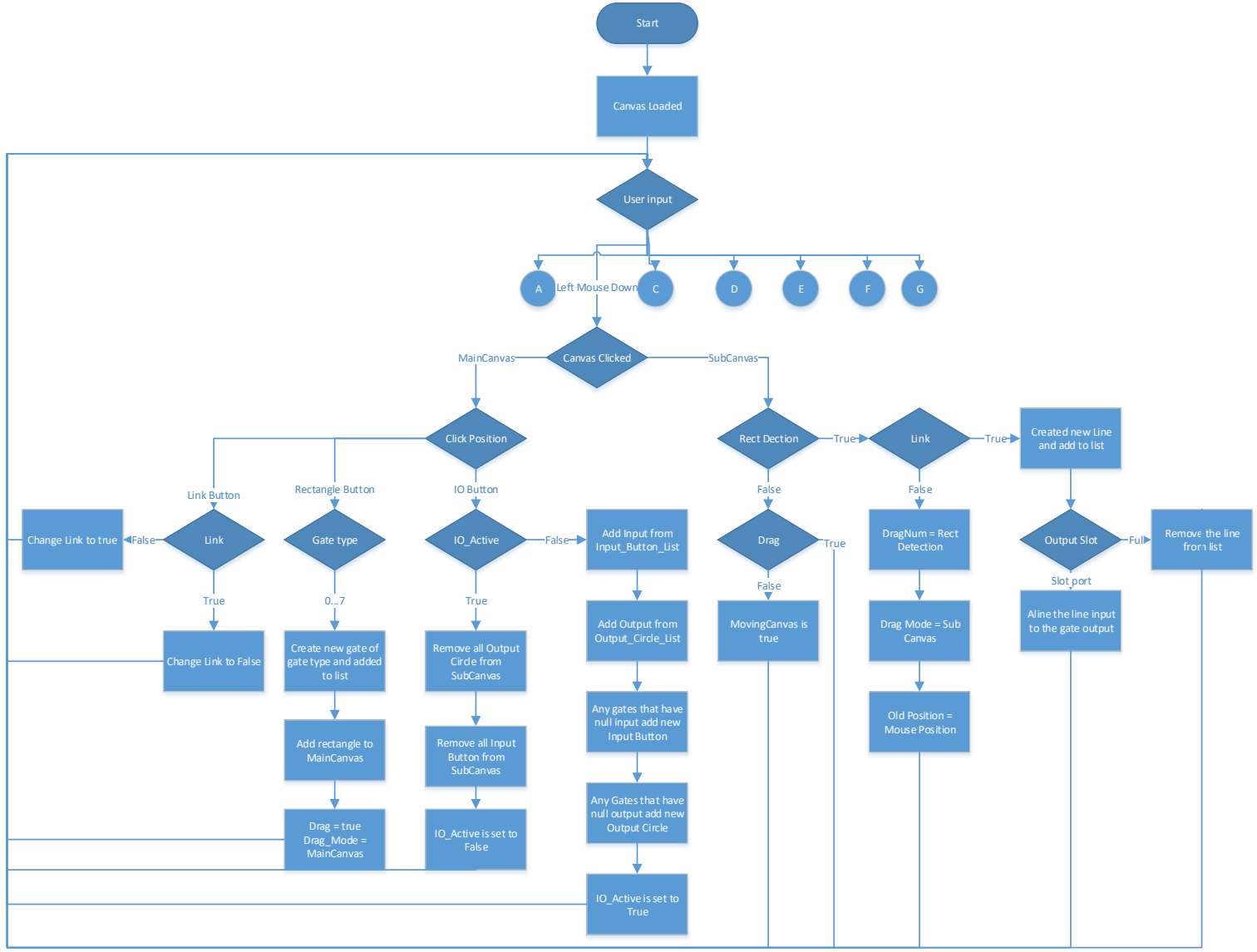
Diagrams

System Flow Chart

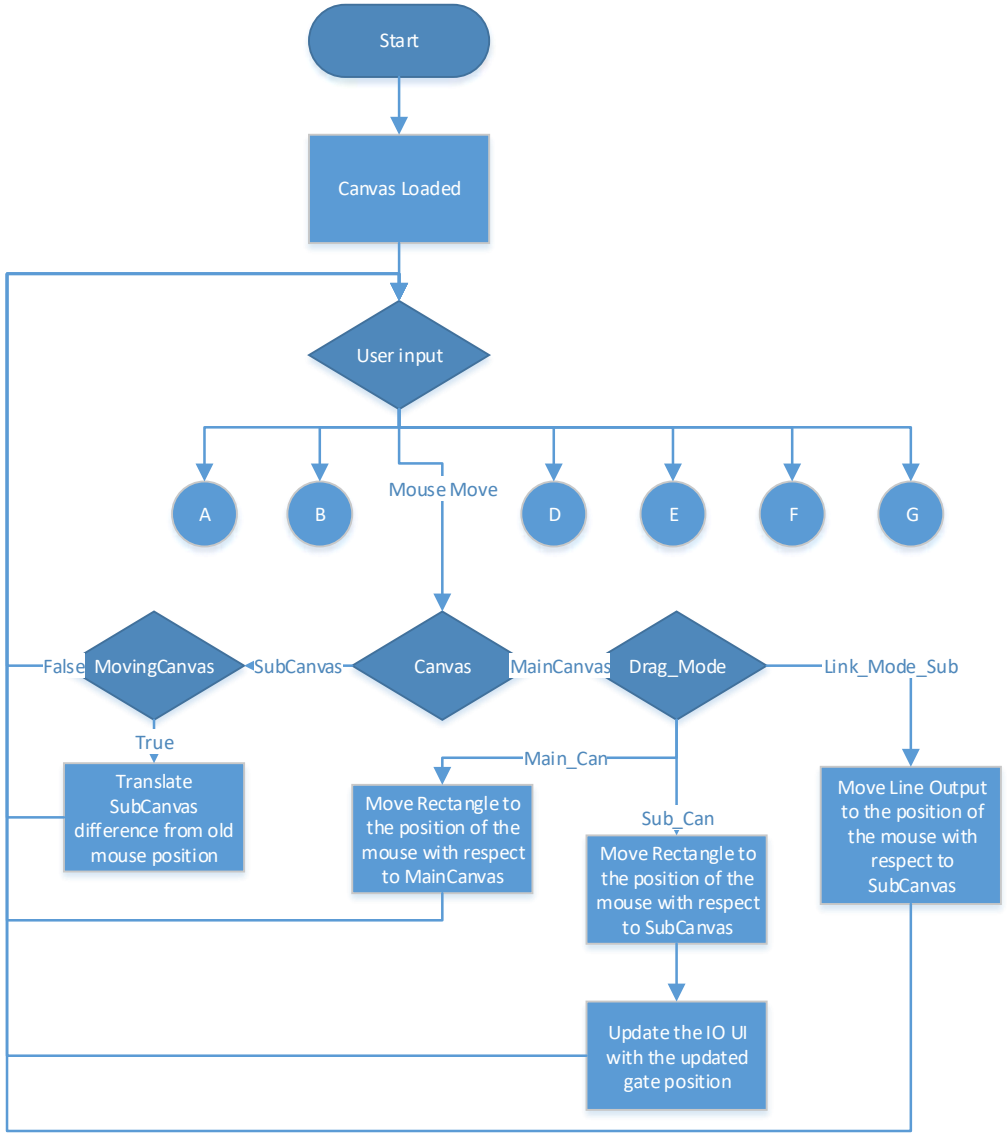
System Flow Chart A



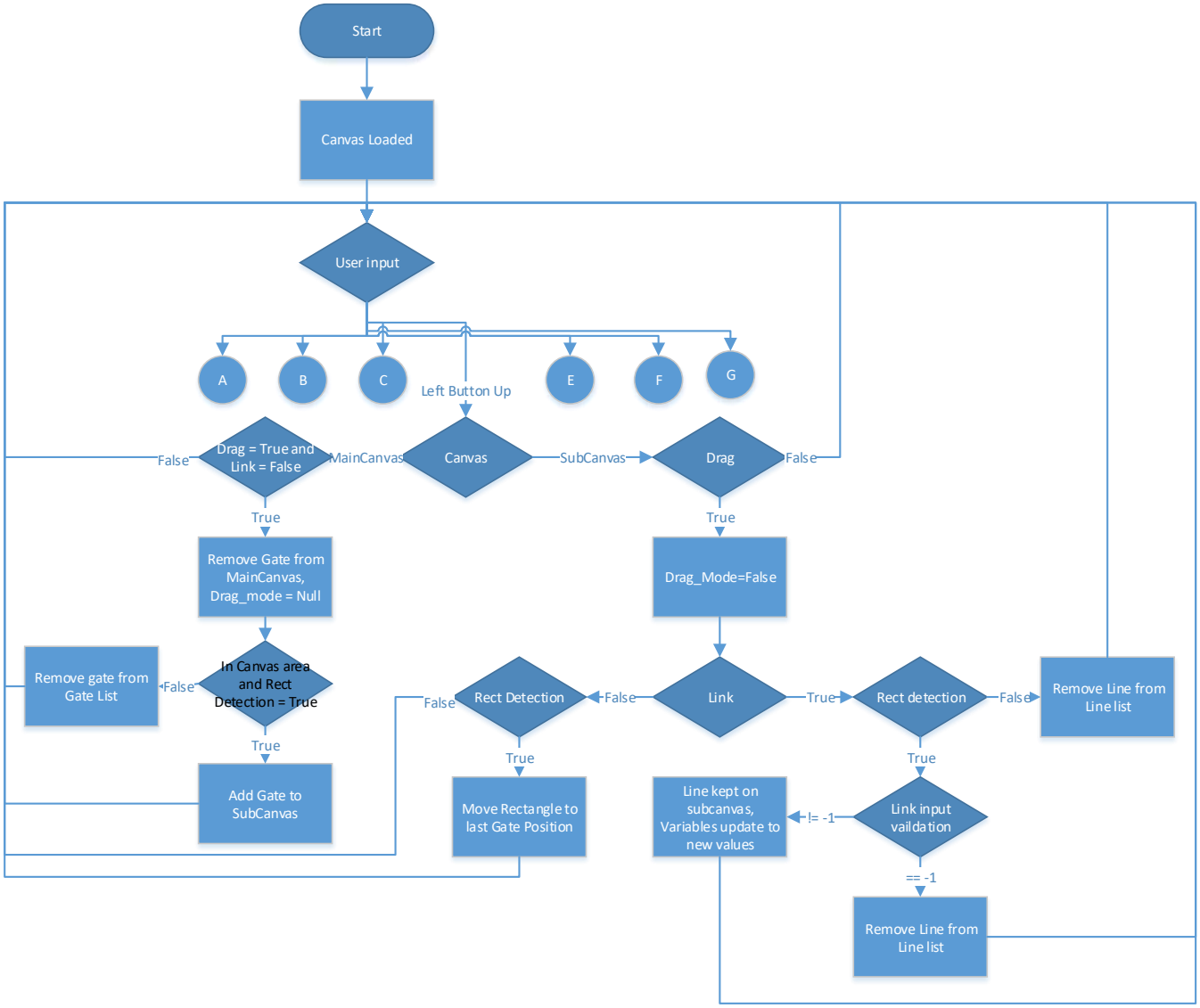
System Flow Chart B



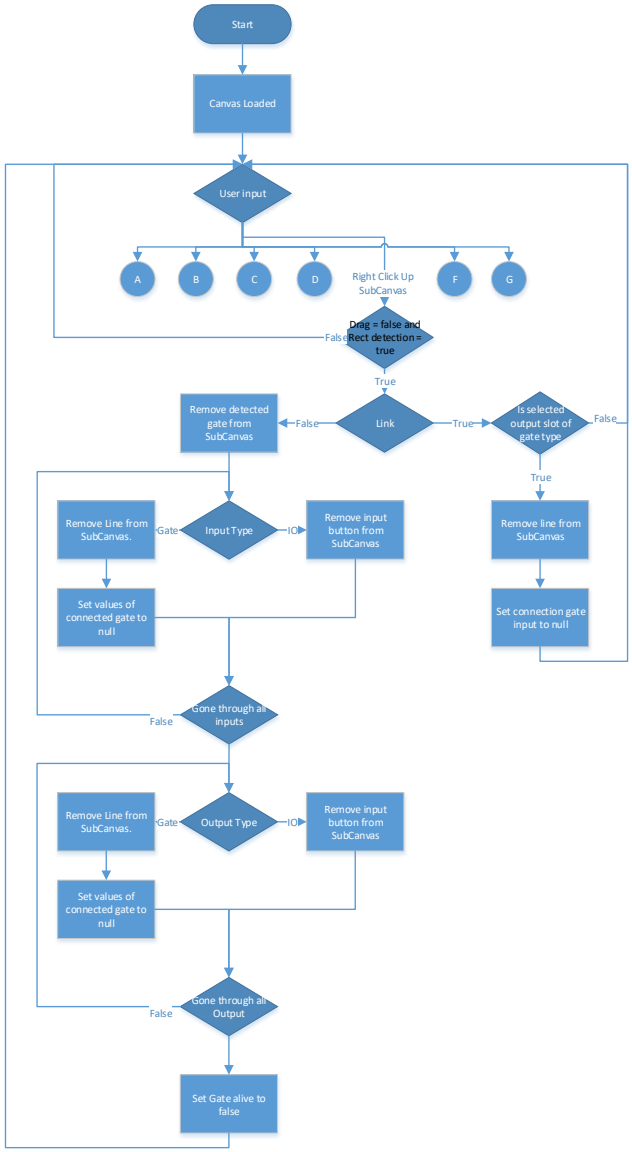
System Flow Chart C



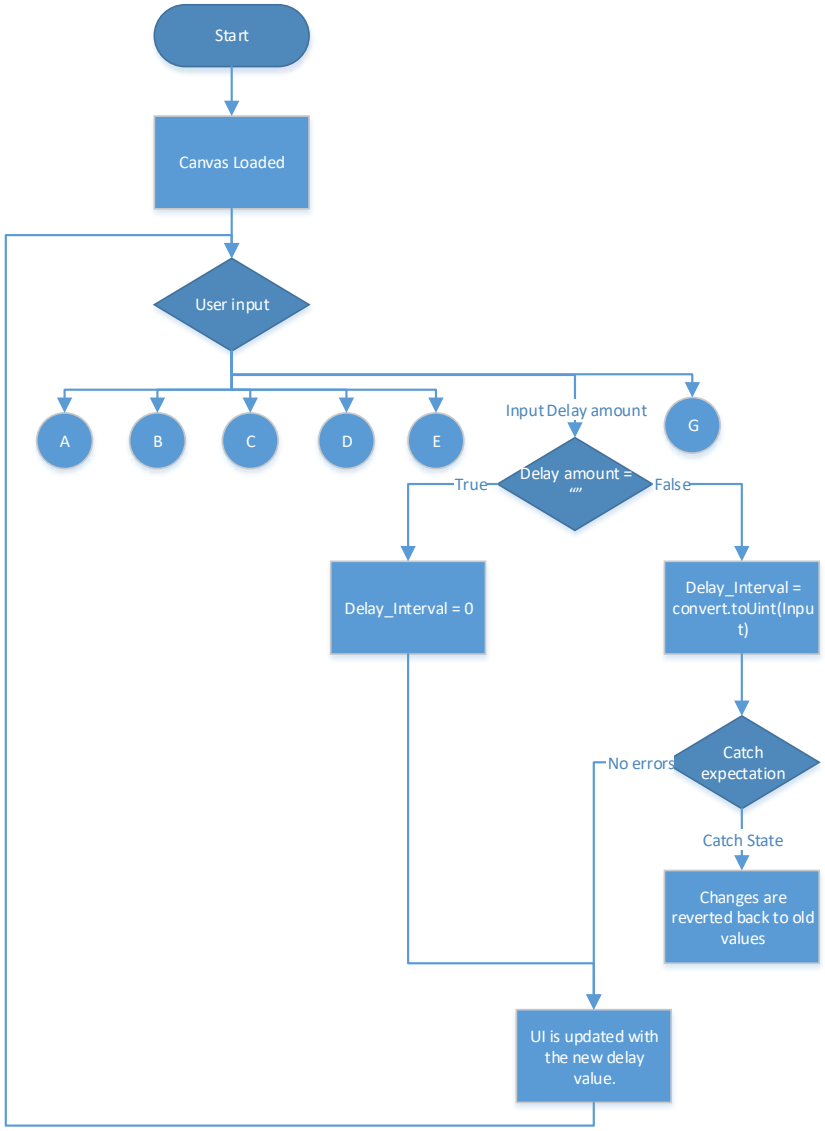
System Flow Chart D



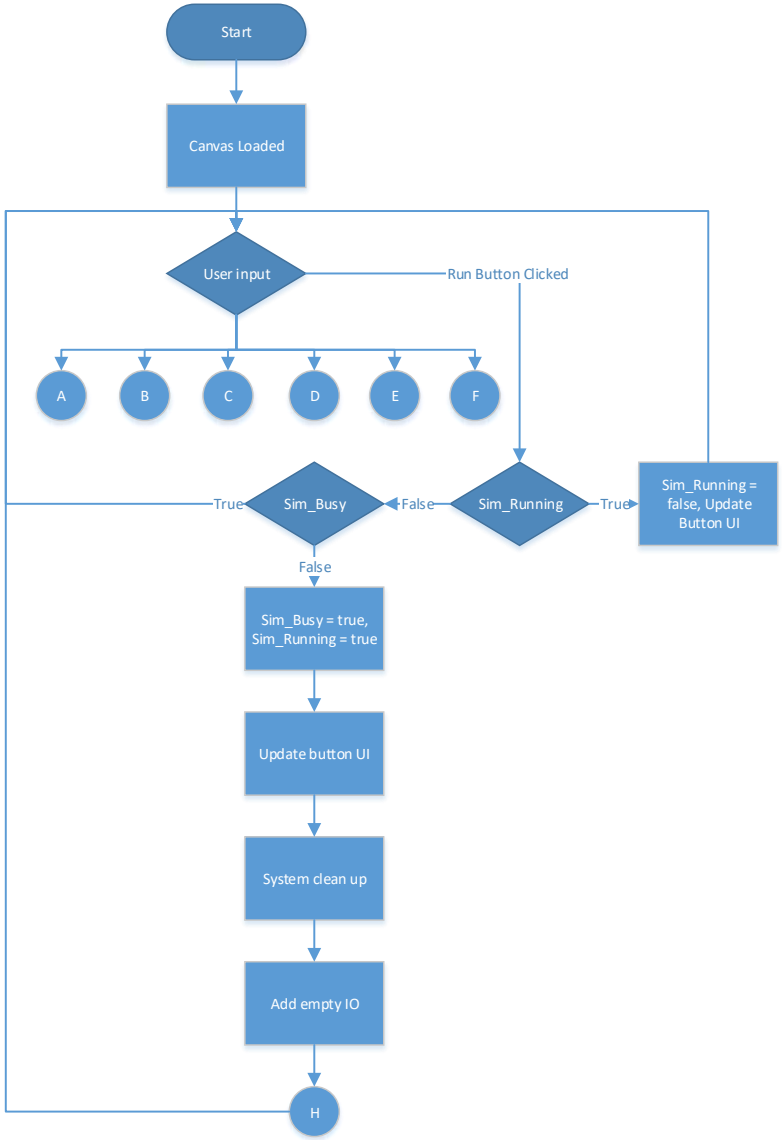
System Flow Chart E



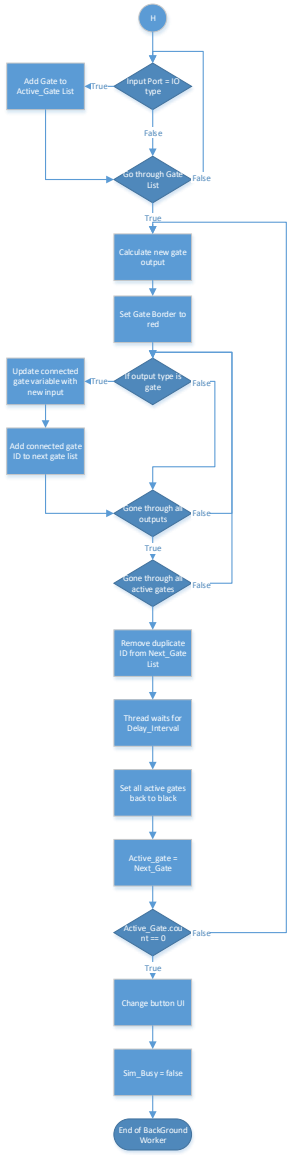
System Flow Chart F



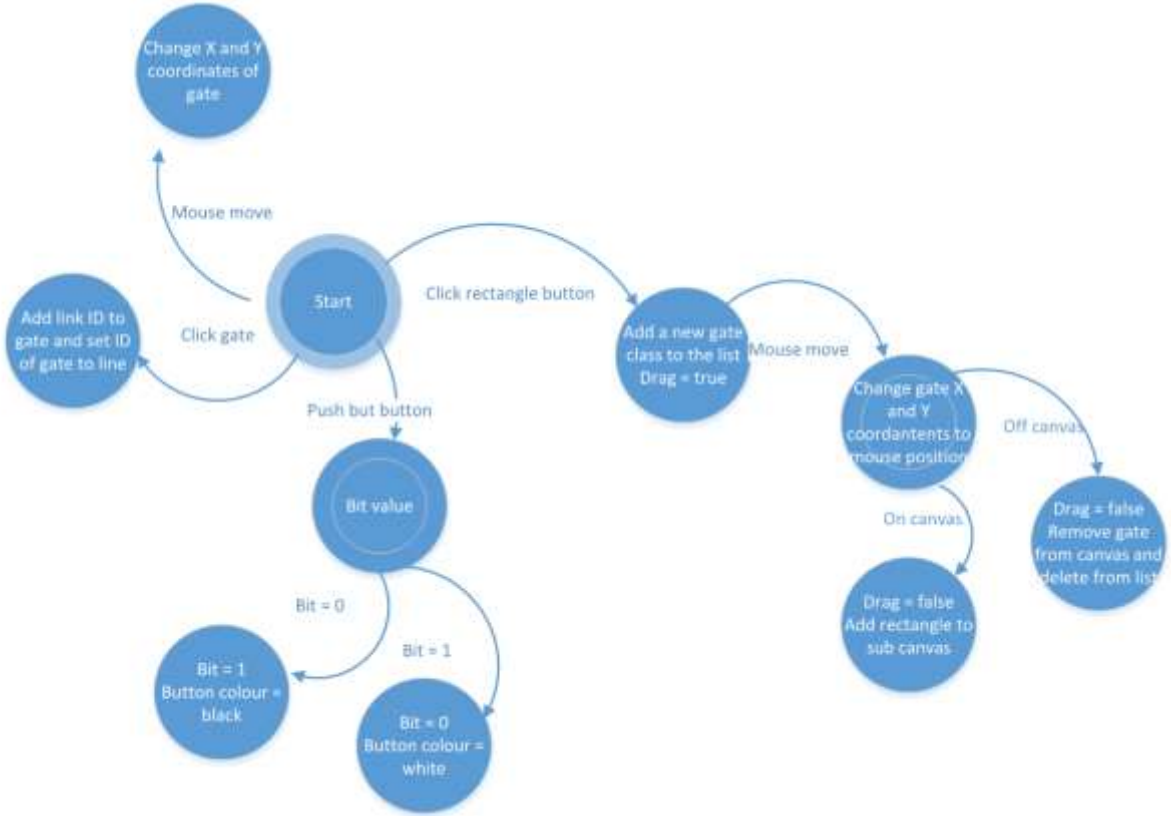
System Flow Chart G



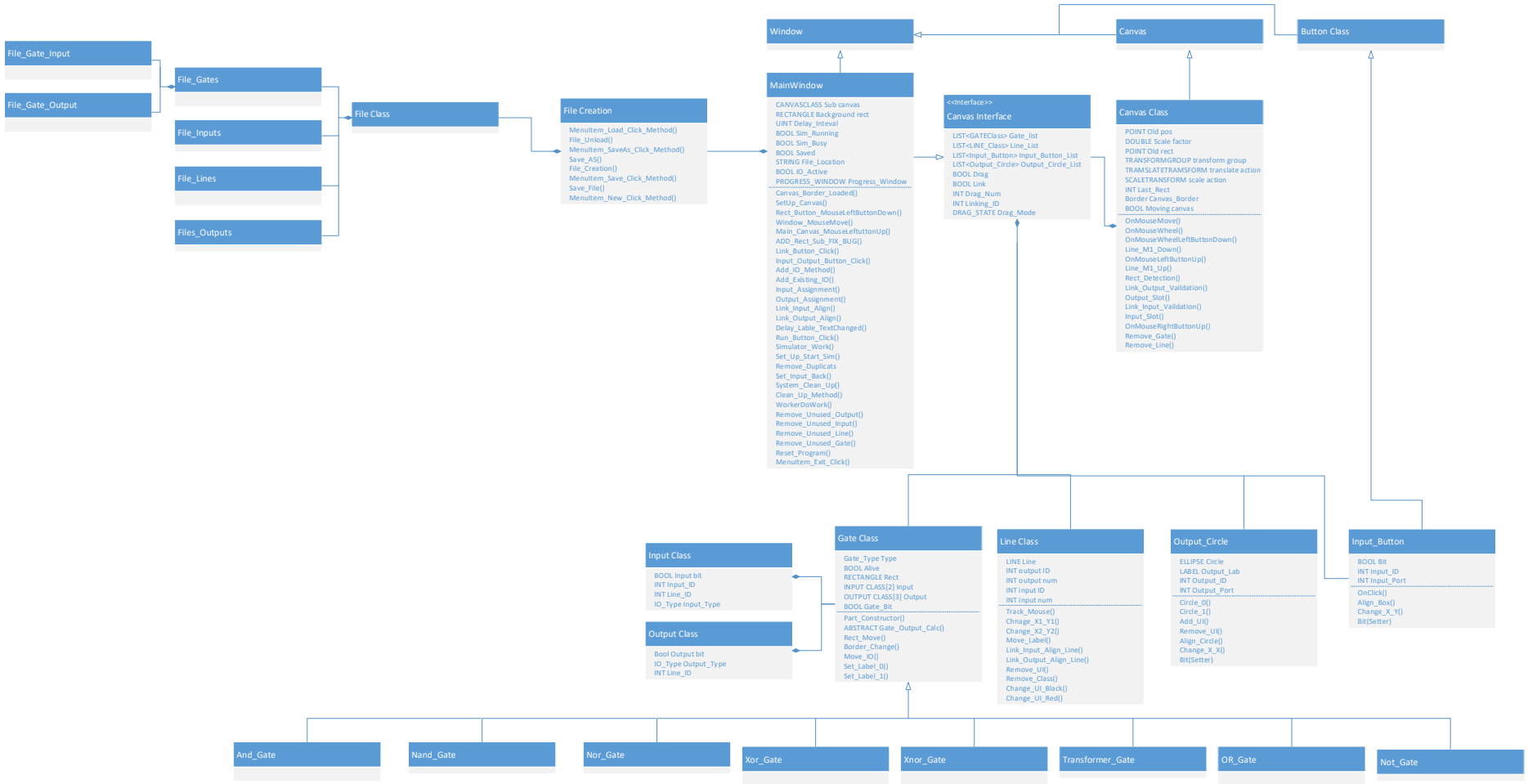
System Flow Chart H



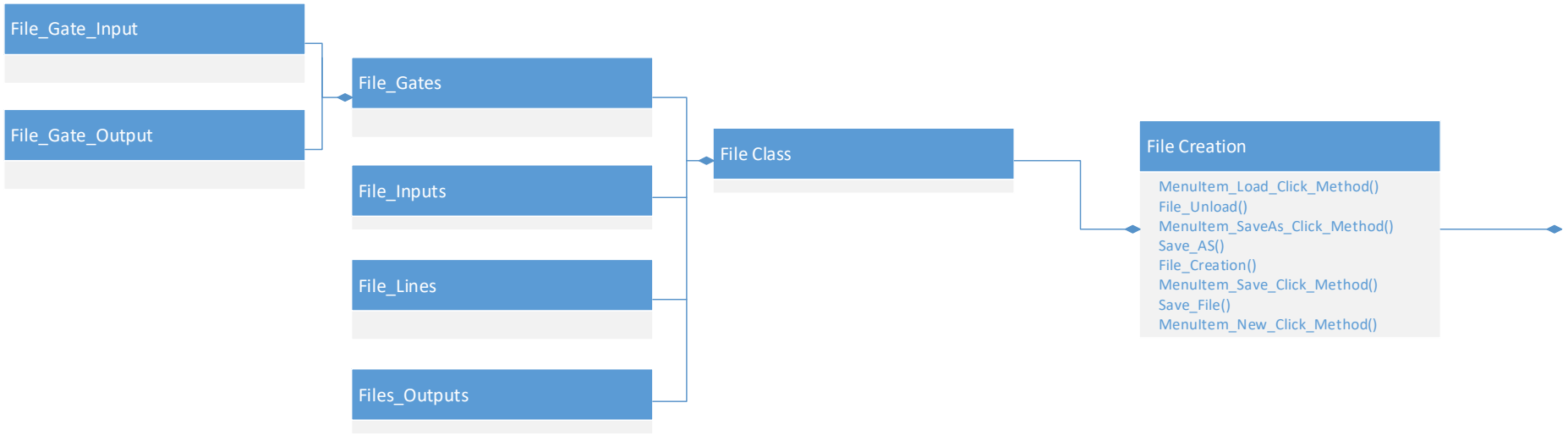
Data Flow Diagram



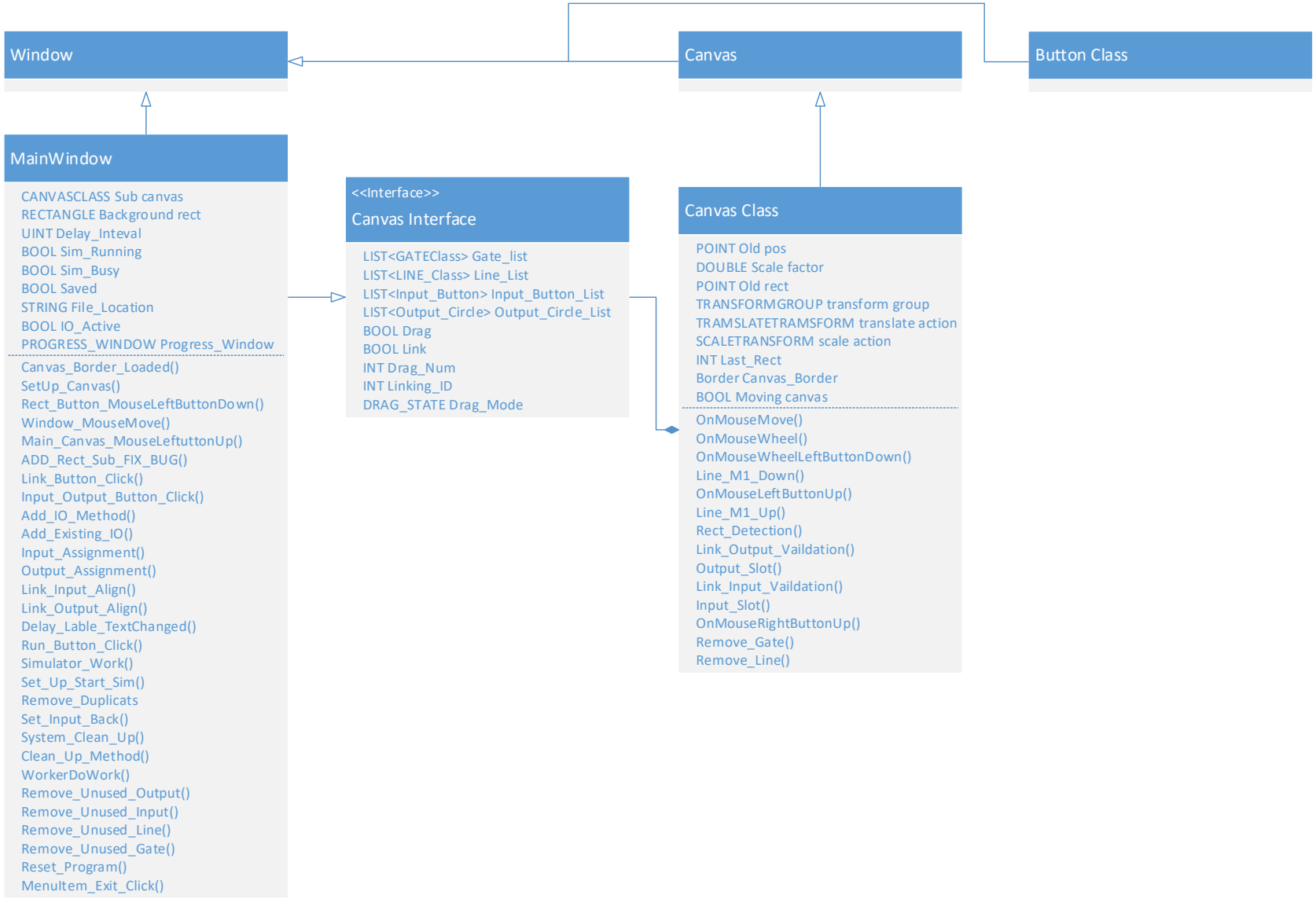
Class Diagram



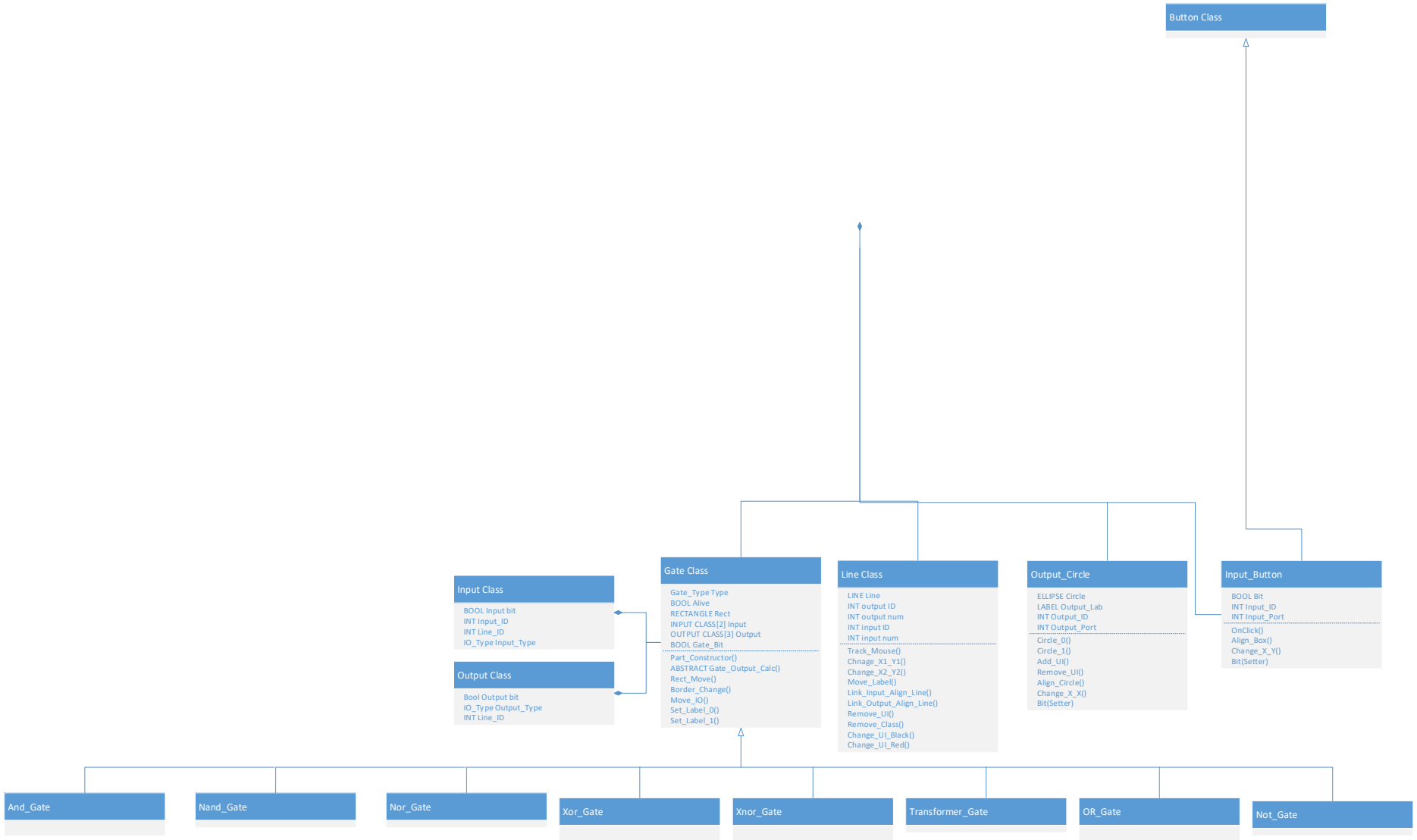
File Classes



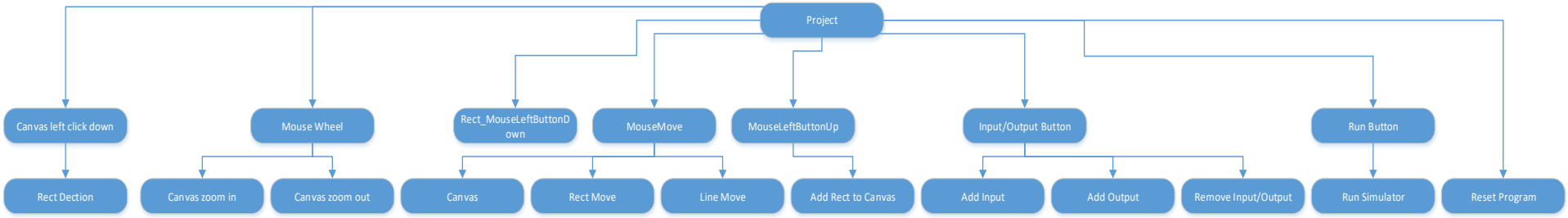
Main Class


Window
Canvas
Button Class
MainWindow
CANVASCLASS Sub canvas
RECTANGLE Background rect
UINT Delay_Inteval
BOOL Sim_Running
BOOL Sim_Busy
BOOL Saved
STRING File_Location
BOOL IO_Active
PROGRESS_WINDOW Progress_Window
Canvas_Border_Loaded()
SetUp_Canvas()
Rect_Button_MouseLeftButtonDown()
Window_MouseMove()
Main_Canvas_MouseLeftuttonUp()
ADD_Rect_Sub_FIX_BUG()
Link_Button_Click()
Input_Output_Button_Click()
Add_IO_Method()
Add_Existing_IO()
Input_Assignment()
Output_Assignment()
Link_Input_Align()
Link_Output_Align()
Delay_Lable_TextChanged()
Run_Button_Click()
Simulator_Work()
Set_Up_Start_Sim()
Remove_Duplicats
Set_Input_Back()
System_Clean_Up()
Clean_Up_Method()
WorkerDoWork()
Remove_Unused_Output()
Remove_Unused_Input()
Remove_Unused_Line()
Remove_Unused_Gate()
Reset_Program()
MenuItem_Exit_Click()
<<Interface>>
Canvas Interface
LIST<GATEClass> Gate_list
LIST<LINE_Class> Line_List
LIST<Input_Button> Input_Button_List
LIST<Output_Circle> Output_Circle_List
BOOL Drag
BOOL Link
INT Drag_Num
INT Linking_ID
DRAG_STATE Drag_Mode
Canvas Class
POINT Old pos
DOUBLE Scale factor
POINT Old rect
TRANSFORMGROUP transform group
TRAMSLATETRAMSFORM translate action
SCALETRANSFORM scale action
INT Last_Rect
Border Canvas_Border
BOOL Moving canvas
OnMouseMove()
OnMouseWheel()
OnMouseWheelLeftButtonDown()
Line_M1_Down()
OnMouseLeftButtonUp()
Line_M1_Up()
Rect_Detection()
Link_Output_Vaildation()
Output_Slot()
Link_Input_Vaildation()
Input_Slot()
OnMouseRightButtonUp()
Remove_Gate()
Remove_Line()

UI Classes


Button Class
Input Class
BOOL Input bit
INT Input_ID
INT Line_ID
IO_Type Input_Type
Output Class
Bool Output bit
IO_Type Output_Type
INT Line_ID
Gate Class
Gate_Type Type
BOOL Alive
RECTANGLE Rect
INPUT CLASS[2] Input
OUTPUT CLASS[3] Output
BOOL Gate_Bit
Part_Constructor()
ABSTRACT Gate_Output_Calc()
Rect_Move()
Border_Change()
Move_IO()
Set_Label_0()
Set_Label_1()
Line Class
LINE Line
INT output ID
INT output num
INT input ID
INT input num
Track_Mouse()
Chnage_X1_Y1()
Change_X2_Y2()
Move_Label()
Link_Input_Align_Line()
Link_Output_Align_Line()
Remove_UI()
Remove_Class()
Change_UI_Black()
Change_UI_Red()
Output_Circle
ELLIPSE Circle
LABEL Output_Lab
INT Output_ID
INT Output_Port
Circle_0()
Circle_1()
Add_UI()
Remove_UI()
Align_Circle()
Change_X_X()
Bit(Setter)
Input_Button
BOOL Bit
INT Input_ID
INT Input_Port
OnClick()
Align_Box()
Change_X_Y()
Bit(Setter)
And_Gate
Nand_Gate
Nor_Gate
Xor_Gate
Xnor_Gate
Transformer_Gate
OR_Gate
Not_Gate

Hierarchy diagram



Human Computer interaction

My project is heavily dependent on the GUI and how the user interacts with it. This is why I've done rigorous testing and take ideas from other programs. It's followed an iterative cycle of development that has allowed me to create an effective interface.

My research into HCI from the testing and examining other programs I came to the conclusion that a drag drop system works best and as little movement of the mouse for the user was desirable. In my research I looked into different types of output. I came to the conclusion that sound cues are overly accepted and more of an annoyance. Colour coded actions let the user know something has works or is working and therefore should be added and now incorporated into my design.

For my questioning I used my previous prototype version with different set up for how the GUI would works and then tweaked or changed it for the end product. The questions I asked are in the table below, while there is only 2 option per question, the question asked were up for interpretation.

Question	Option one	Vote for one	Option two	Vote for two
Should there be a drag drop or click-click system for UI object	When interaction with UI objects you just need to click and hold then release when done with it	4	When interaction with UI object you just click once and release then the UI object will follow you until you click again	2
Should there be a border around the gate	For a border	2	Against a border	2
Should a new rectangle spawn in the middle of the canvas or on your mouse	Spawn on your mouse courser	4	Spawn already on the canvas	0

As the program needed be as natural for the user to use, I needed to anticipate the user actions that they would take. As these actions would most likely be based around programs,

they use every day I examined the Microsoft office software for the actions that should take place.

It is important that the program is lag free and efficient as possible so that the user experience isn't hindered. I did research into wither it's faster to sort then binary search or just using hash tables and rehash. I came to the conclusion that hash table would be move adapted to the problem but not a vital part required as it's going to be around twenty objects being linear searched.

The intuitions interfaces by the Microsoft products is what's influenced me the most. They are widely liked and used around the word. This is why they made them a good choice to base mine off. The Colour difference between background and usable space and the border around everything when hovering over it is what really stand out for the HCI. Similarly, I looked at Paint and the tops up display with all the functions and settings and choose similarly. While these all have great features, I also changed them to be updated for modern technology. Because monitor sizes have become wider (16:9 instead of 3:4) having the setting on the side of the screen makes it more space efficient than on the top. And instead of greying out of adding a border to a highlighted object I went from black to red when highlighted. This made it stand out more and easily noticeable but had its draw backs for colour blind people.

Algorithms

Algorithms: UI_object search, Gate methods, gate order method, file creation and file load, validation of gates, pseudocode.

The backbone of the program is set around the GUI and how you interact with it. This is why many mechanics will be based around multiple event handlers. The main actions are called under their respective class and then sub methods will be done in the variable class. This allows the structure for program info to stay central and independent methods to be designed for the class to sort out.

Rectangle detection algorithm

There are many ways that can be done for detection algorithm. While some are better than others, they all depend on a case by case situation. The 3 ways are: sorting by 2D array and then binary search, Geometric hashing and linear search. 2D array and binary search would be the best if the list is extremely large, Hashing would be good as a middle ground and linear search is good for a small list. The problem with hashing and 2D array is due to the list always changing. The size is never fixed and this means the array will need to be resorted each time a new element is added. For the hash table it's not so bad you will only need to rehash it after a few rectangles have been added but if 1 is removed from the hash it will all need to be rehashed. This is due to if there is a collision and the object is moved up the list and then if an object between where it should be and where it is, is removed then the hash table will be corrupted. Mathematically the most gates that can fit on the canvas is 272. This means at maximum it will only need to do 272 checks to find the object it's looking for. This is why linear search is an acceptable solution. However, if development was for professional

use to make large logic gate sets for CPU design or chip design where they have billions of transistors then 1 of the other solutions will be required. If I had to guess their use case of the software then it would be 2D array due as they would make the design and execute it once everything is created.

Sorted with 2D array (bubble sort)

```
for i in range arr.count
     for x in range arr.count-i-1
          if arr[x].X > arr[x+1].X
               swap arr[x] and arr[x+1]
          else if arr[x].X == arr[x+1]
               if arr[x].Y > arr[x+1].Y
                    swap arr[x] and arr[x+1]
```

binary search 2D array

```
q=0
r=arr.length-1
mid = (r-q)/2 + q
while found == false or q-r>1
     if arr[mid] == find
          found = true
     else if arr[mid] > find
          r = mid
     else
          q = mid
     mid = (r-q)/2 + q
```

If sorting into hash table

```
hashtable[arr.length*1.2]  = [-1, … , -1
For i in range arr.length
```

```
    Num =     arr[i].X ^ arr[i].Y + arr[i].Y ^ arr[i].X
    Output = num mod arr.length*1.2
    While run == true
        If hashtable[output + z] == -1
            Hashtable[putput + z] = i
            Run = false
        Else
            Z = Z + 1
    Run = true
```

Searching the hash table

```
Num = pos.X ^ pos.Y + pos.Y ^ pos.X
Output = num mod arr.length*1.2
While run == true
    If hashtable[output+i] == Find
        Return output + i
    Else If hashtable[output+i] == -1
        Return -1
    Else If output + i == arr.count-1
        i = 0
    Else
        i = i + 1
```

Linear search

```
Run = true
Count = -1
Find = INPUT
```

```
WHILE Run == TRUE AND Count != RectList.count
     Count = Count + 1
     IF RectList[Count].ID == Find
          Run = FALSE
IF Run==TRUE
     Return -1
RETURN Count
```

Gate methods

The gates are the main functional part of the program that user cares about. As the gate had its own class and all the variables needed it were in it the method for the output would be in the class. I had 2 ways of doing it, just have 1 base class that stores all the methods for each gate and then when needed to work out the output would use a switch. Or make 8 children classes that inherit the base class and override the method and when need the output just call the method. Both have their pros and cons and even now I don't know which the best way is.

Order of gate method

When the program runs you want it to execute in the order from the start to the end. Due to being able to have infinite number of start positions and any number of end positions with gates in-between working out the sequence is important. This is due to the input for a gate may not be calculated yet and just be null. This will cause problems but also desired if you start at 2 different locations but merge to make 1 exit point.

In my prototype I calculated the order of execution by working out the maximum number of gates it would take to reach that gate. This works well for the prototype as it meant no gate is left without an invalid or not calculated input. It was also easy to code as it meant all you had to do was trace through each input until it found the exit and replace a variable if it was greater. There were a few problems with this method. 1 if there were no exit it would go in an endless loop and never be executable (or only be executed once if you add an error check). 2 if you wanted a circuit to loop it was impossible to have an infinitely long list with repeating gate order. While you could just say go to this point in the list again if you had 2 loops running at the same time but one was length 5 and the other was length 7 when the length 5 has ended then the length 7 would be reset. There was also a problem with if you wanted the starting gates to all start at once instead of the longest chain you couldn't do it.

This brings me onto my new method. Instead of calculating the order at the start the program will go through the start gates all at once and then send the output to every gate. It will then remove all the gates and add the gates input gates into the list. This method removes all the problems that the other algorithm had. This system also works better for

the program due to the easier coding of the simulation. It means real time I can make the changes to each gate and pause it when at the end of calculation. This is why making all the start gates start at once desirable due to being able to see the progress of the circuit as the message goes down the wire.

```
For x in range input_list.count
     If input_list[x].active = true
          Active_Gate.add(input_list[x].Input_ID)


For x in range Active_gate.count
     Bool Duplicate = false
     For (I=x+1) in range Active_Gate.count
          If Active_Gate[i] = Active_gate[x]
               Duplicate = true
     If duplicate = true
          Active_gate.removeat(x)
          X= x-1
```

File creation

There are 2 different file types that I could use, JSON and BinaryFormatter. JSON was a good choice because of its interlanguage capabilities. It's also excellent for readability but lacks in the security part. For BinaryFormatter it's secure and hard to edit but lacks the readability. It's smaller in file size and will be better for large sets of data. For testing JSON is the clear choice but as the user won't be messing with the file and doesn't need to the smaller file size of BinaryFormatter is the better choice.

Because this part is largely language dependent, I created each method in C#.

JSON File creation

```
List<data> _data = new List<data>();
_data.Add(new data()
{
     //class variables
});
```

//converts the array into a format that JSON can use

```
string json = JsonConvert.SerializeObject(_data.ToArray());
```

//uses the string file and write it into a document

```
System.IO.File.WriteAllText(@"D:\path.txt", json);
```

JSON File reading

//Read file and puts it in r

```
using (StreamReader r = new StreamReader("file.json"))
{
    //Puts everything into a string
    string json = r.ReadToEnd();
    //puts everything back into list of the class
    List<Item> items =
    JsonConvert.DeserializeObject<List<Item>>(json);
}
```

BinaryFormatter Creation

//FileStream reads the file

```
FileStream fs = new FileStream("File.dat", FileMode.Create);
BinaryFormatter bf = new BinaryFormatter();
```

// bf converts the file into binary format

```
bf.Serialize(fs, list);
```

//This ends the creation

```
fs.Close();
```

BinaryFormatter reader

```
List<Class_data> items = new List<Class_data>();
```

//opens the file and splits it into each class of the list

```
var bformatter = new BinaryFormatter();
using (Stream stream = File.Open("File.dat", FileMode.Open))
```

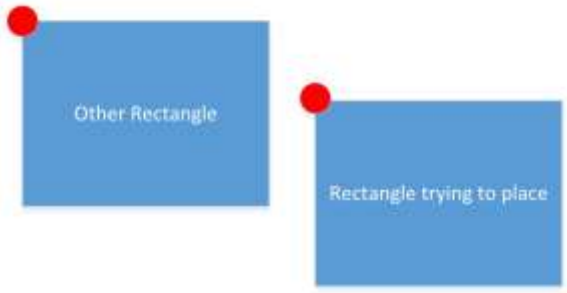
```
{
        //Reformat the file to fit the list of objects
        items = (List<Class_data>)bformatter.Deserialize(stream);
}
```

Validation of gates and lines

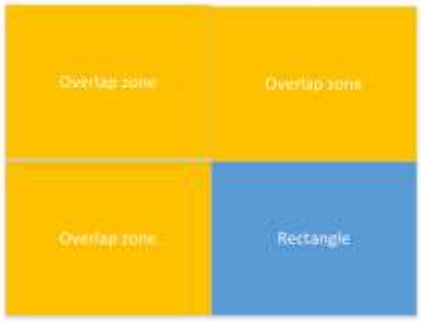
There are lots of checks that are required to make a valid circuit. Some are done while creation is taking place and others are when you try and run the simulation.

Rectangle overlapping

For making sure you don't have overlapping gates I used the top left position of each rectangle and then some maths to work it out.



Then you just create an area around the rectangle you're trying to place and see if there is another rectangle with their left point in that zone.



This way you can easily check the area by seeing if the coordinate lands in the rectangle. However, there are different size gates. So, the area that needs to be calculated all needs to take a variable of the size of the gate your comparing it to.

Line validation

The lines had a lot of flexibility in what they can do. I also wanted the user to feel like they can control them like they want. This meant they were required to go to the right input slot and output slot. This just meant that some checks on the mouse position based on the gate needed to take place. As the first click on a gate is always the output this was simple due to only the special gate having more than 1 output slot. For the input all but two had 2 input

slots. So, I just had to split the gate height in half and check if it's above or bellow to connect it. An additional check is required as I needed to see if the slot that was trying to be used was already taken. If they were then the program would just go through each slot starting from the top down and check for an available slot. If there wasn't then it would be removed.

There was check to make sure that there isn't a link between the same gate (output can't go into the input). This just meant I needed to check a new local variable to store the ID of the gate that is being linked and when the line is connected to an output gate that ID didn't equal the variable.

Data structure

Data structure: explain the list with all the classes in, Explain the variable in each class

This table stores the acceptable values and description of the variables in the program.

Field: Canvas

This field will store all the variable that class Canvas will use. The Canvas class inherits the library from window Canvas class (The base class in WPF) so a lot of variables are already created.

Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
Scale factor on a slider but represented as a double	Double	0 – 2 but increments by 0.0625 from 1	>0 <= 2	Because the double is 32 bits I used a slider of 0.0625 which is the 5th bit of a decimal. This means the number will never become impossible to represent in binary double.	0.9375	0.1 0 -1 2.0625
True or false for if a mode is	bool	True/False	None	It only has 2 states	True	Null

active or not						
Last position of the mouse when dragging the canvas	Point	Double[2]	None	It's stores 2 double	[0.1,152.1]	"Hello"
Last position of the gate before dragging	Point	Double[2]	None	It's stores 2 double	[4,21]	"Yes"
Is moving canvas active	Bool	True,False	None			Null
The ID of the last gate ID	Int	Greater than -1	-1<=x	Checks to make sure it's greater than or equal to -1	-1,0,404021	Overflow, -2

Field: Gate class

This is the variables for the simulation for each gate that will be stored in a list.

Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
Enum for the type of gate it is	Enum	From 0 to 7 and the string name for each type	Check if <0 or >7	There is only 8 gate in the program and by default it will be And Gate(0) so null isn't acceptable	0,7, And, Xor, "Transformer"	-1 8 "Hello"
Boolean for if it is a 1 or 0 for the gate output	Bool	True/false	none	Can't be null	false	Null
Boolean for if the gate is active on	Bool	True/false	None	Can't be null	True	Null

the canvas or not						

Field: Input Class

This class is to break up the gate class so that there can be an array of inputs to save having multiple groups arrays. It also makes the code easier to read as everything is under one name.

Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
The bit for if the input value is a 1 or zero	Boolean	True/False	none	Can't be null	true	NULL
The ID of the gate that is inputting	int	>-1	If >-1	Can be null	0	-2 2147483649
Stores the type of input it is	Enum	Enum are strictly typed and only certain input will be valid. This will be based on the Enum class but will contain null, gate, button	Must contain null, gate, button		gate	Banana
Int for the ID of the line that it is connected	Int	-1 to overflow	x>=-1		3231	-2

Field: Line Class

This field stores the variables for all the lines and mainly just a way of tracking the path and connections that each gate has.

Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
ID of the gate the line needs to connect to.	Int	-1>	If >-2	-1 equals null	100000	-100000 2147483649
This stores the output number for the line. The output number is the port from which the line comes out of.	int	0 or 1	>-1 <2	Can't be out of those ranges	0,1	-1 2
ID of the gate it is connected to	Int	-1>	If >-2	-1 equals null	100000	-100000 2147483649
The Port number that the line outputs of	int	0 or 1	>-1 <2	Can't be out of those ranges	0,1	-1 2
Double for the positions of the line ends in terms of X and Y	Double	4000	3000> x > -1000	This is the size of the canvas. The line can't be outside of this area.	421,-23	-23142, 432425

Field: Main

This field stores the variables for the state of the program and everything that is happening.

Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
State of if the program is being used right now	bool	True/false	none	Can't equal NULL	true	NULL

The integer value of the ID for the object that is being active	Int	>0	If-1>	Can't be a negative number	0	-1 2147483649
Boolean for flipping between the 2 states the program can have	Bool	True/False	None	Can't be null	True	Null
This stores what is active and where	Enum	Null, main, sub, linkSub	Strictly typed	Can't be anything else	linksub	Apple
The time delay of the simulator	Uint	Any positive number	x>=0	It needs to be Uint as time can't be negative	0,314314	-1
weather the sim should be running	Bool	True/False	None	Can't be Null	True	Null
Weather the sim is free	Bool	True/False	None	Can't be Null	True	Null
Has it been saved or not	Bool	True/False	None	Can't be Null	True	Null
The last know file location	String	Anything	None	No check is needed as the string is in a try catch statement	"", "document/…/File"	
The ID of the rectangle that is currently trying to be linked.	int		x>=-1	At a time no gates could be linking	-1, 422432	Wawa, overflow

Field: Canvas Interface

The canvas interface will store the variables that both MainWindow and Canvas

Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
The list of all the gates that the program will use.	List of class gate	No limit	none	none	Gate Class	Banana
List of line class	List of class Line	No Limit	None	None	Line class	Apple
List of input button	List of class input button	No Limit	None	None	Input Button Class	Pear
List of Output Circle	List of class output circle	No Limit	None	None	Output Circle Class	Orange
Boolean for if the program is dragging something	Bool	True/false	none	none	true	Null
Boolean for if the program is in the link mode	Bool	True/false	None	None	True	Null
Int the ID of the gate that is being dragged	Int	x>-1	x>-1	Greater than -1	-1,4134	Grape
The type of state that the drag is in.	Enum				Sub_Can, Main_Can	Berry
Int for the gate ID that is linking	Int	x>-1	x>-1	Greater than -1	0,24	Plum

Field: XAML

The XAML doesn't have variables but has attributes. There is only really one attribute that can carry over to the main program and that is tag.

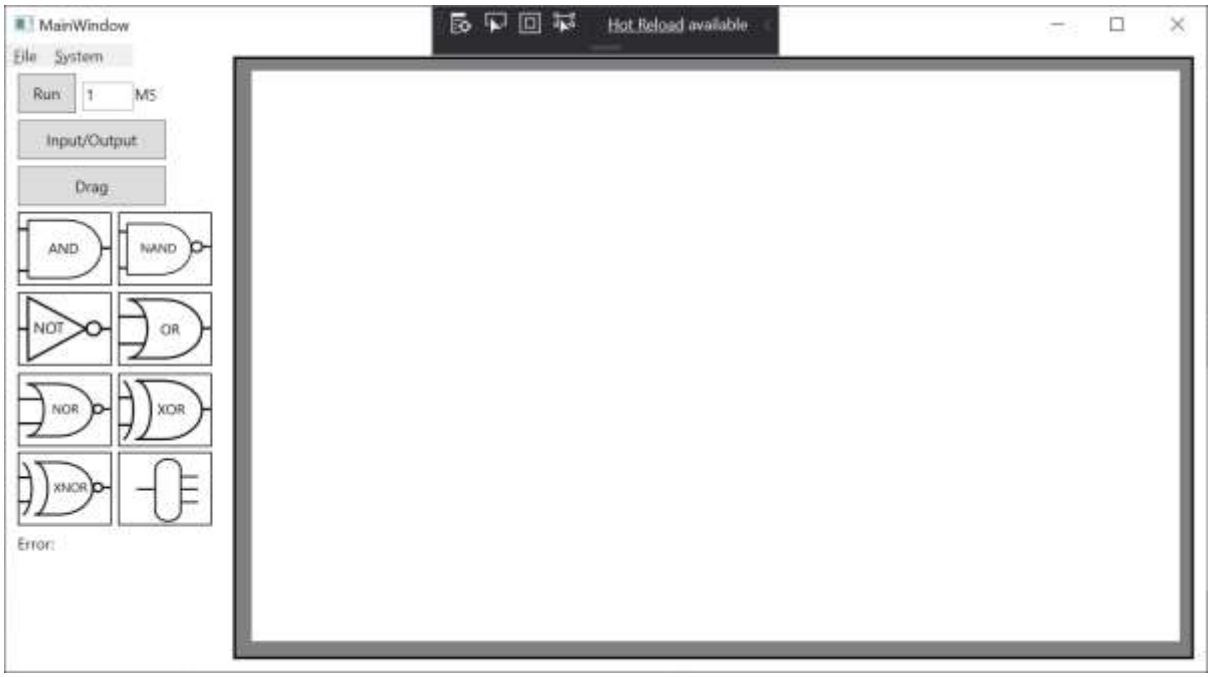
Variable description	Data type	length	Validation check	Validation description	Valid data	Erroneous data
The TAG can store the File location of the vector image that the rectangle that is being added to be stored	String	Strictly typed	Would be very hard to change the value	Hard coded	And_Gate_L	cherry

Graphical UI

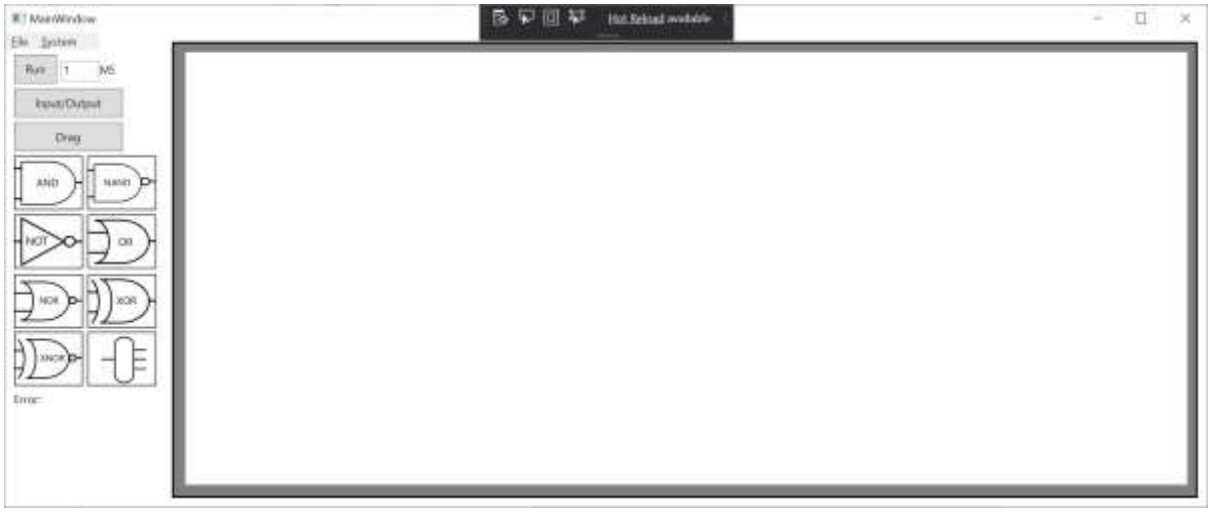
For the UI I had an idea of what it would look like and how it would interact. I made it a major goal to make the program to be easy to use and require the least amount of movement of the mouse. This is why the drag drop system was used. The buttons and objects should easily stick out so that the user knows what to do without much thought.

The UI needs to be dynamic so that it fits any monitor. This was a crucial goal as the school computers are 4:3 but all my monitors are 16:9. So during the development process I knew that it would also transfer over to the other aspect ratio.

UI when loaded:



UI stretched:



Everything that the user can interacte with is on the left hand side. This is the best place for it as people read from left to right. It's also in order of most important in the up to down.

Software development model

Before deciding on the LGS as my final project I made a prototype to investigate possible solutions to the problem, this code was quite crude but I learned how to develop many of the modules far further as a result. I restarted with the mind of using OOP as the main goal. While my style and structure would work with console applications it didn't work amazingly with WPF. This was due to lack of testing and understanding of the frame work of C#. So, I restarted again but knew I could reuse the methods and Xaml file in the old project. I used my research of custom Canvases and moved my methods into an override event handler, this made the code much better.

The overall development model of the project was the spiral SDLC model where you make prototype before you release the real thing. While this model has been inefficient, I don't think I would've got to where I ended up without restarting.

While the project was one model when coding I feel I followed 2 separate models for programming. The project can be broken down into 2 aspects the GUI and the functionality part. For the GUI each feature was almost independent of each other so I could follow a feature driven development approach. This worked really well as it made testing efficient and easy. It also meant code could easily be changed and maintained due to each function not affecting others.

For the functionality part of the project I will follow the agile model but keep it tight. What I mean by this is that I know the functions that need to happen and while I could just code them, I will keep in mind and plan out for how one function works with another. This will hopefully give me enough of a plan but also flexibility for me to follow feature driven development on the small parts.

File structure and organisation

File structure and organisation: Json for external storage, Window markup file for the gate in vector format.

I use external files two times in my project. One is to store the vector XAML file for the gate image in the rectangle UI object. The file type is Window Markup file which is basically just a text file but readable as XAML by programs. I used Microsoft expression design 4 to make the images with stock parts: line, ellipse and text. Then import it to the XAML vector format.

For storing the setup and loading of files for gate setup I've got the option of 2. I could use JSON or .dat/Serializable. While both are hard it depends on what I want it for. If I want security I will use .dat and serializable because it encrypts the data. If I want readability and easy changes then I would use JSON. While both aren't massively large file types .dat is smaller.

Security and integrity of data

Security and Integrity of Data: not really needed but talk about how everything is strictly typed

While my program doesn't hold personal data and if someone hacks it, it would only affect them, I want to keep the program code and how it works a secret.

Modular Structure

The code is heavily built in modules. I choose this way because most classes are independent of each other. The file classes are all in their own modul as it made the most sense and allowed for the methods to be independent from the bulk of the code.

Canvas class is a good example, As long as it has the canvas interface passing the program state values it can work independent and could be put into another program and work. This is great for testing as it's all inclosed it meant I could duplicate the canvas and allow the program to have multiple Canvas open at once. This feature only occurred to me once I was far into the project so the framework to make that work wasn't there but if I was to redo the project again that would be a priority to be made possible.

The Gates are nicely packaged with the input and output class being separate it made the code easier to read and much better funcitality when calling up a variable inside the input/output as each variable in the class are linked together.

For the MainWindow it's not desirable as you can only have one but there isn't much around it. This is why most methods would be pushed into another class. Only the event handlers and alignment methods are held in the class. This was the bear minimum that I could use.

Other

The multithreading is a big part of my program, I'm using the background workers in C#. This is an easy to use framework that does multithreading but also added event handlers which the user can use to easily start, run and stop the thread. It follows the same rules as generic multithreading but sending parameters to the work is easier.

I still had to learn about how to update and change varaibles from another thread using invoking dispatchers. This is was due to WPF objects being static and being apart of another thread.

It allows me to hold the program for the system clean up and allow the user to keep interacting with the UI while the simulator is working. These were both goals for me and the use of background workers allow it to happen.

Technical Solution

Things to Address

The code used in the implementation of this program is efficient and the result of significant research into user interfaces and Object Orientated Design. There are however some aspects of the code I feel it is important to address in MainWindow.Xaml.CS which implements the core module for this program. Specifically that at first glance it may appear that I have made use of "global variables" when in fact these are examples of public Enums in use as they are a declaration of a dictionary of named integer constants that are the blueprint for the Enum, as described in Microsofts documentation for the language C# (2019, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum).

```
namespace A_level_course_work_Logic_Gate
{
    public enum Drag_State  { Null,Main_Can,Sub_Can,Link_Mode_Sub}
    public enum IO_Type  { Null, Gate, IO }
    public enum Detection_State { Null, Detected }
    public enum Gate_Type { And, Nand, Not, Or, Nor, Xor, Xnor, Transformer }

    public partial class MainWindow : Window, Canvas_Variables
    {
```

As a result these are considered to be constant and do not hold a value as a result of them being Namespace variables and Enums are used in this manner across every class in the namespace. I have also considered the implementation of class structures to take into account the lack of multiple inheritance within the C# language, which results in more robust code and far more limited interplay/side-effects between functions.

When you create a method that just sets the variable to the value of the parameter that is just the same as `Variable {set;}` in C#. When you create a method that returns the variable that is just the same as `Variable {get;}` in C#. So you can either have 2 methods for every one of your variables taking up 7 lines of code which are generally far less efficient than the built in functions or you could just add `{ get; set;}` to each variables where they apply. This isn't about breaking the way getters and setters are used but taking advantage of the facilities available within the language.

My research into the use of Getters and setters included the following websites and texts:

- How to use getter and setter: https://javasolutionsguide.blogspot.com/2016/04/encapsulation.html
- C# Getter and setter: https://www.dotnetperls.com/property
- Programming C#: by O'Rielly (2018)

Instance/Class Variables

```
    public partial class MainWindow : Window, Canvas_Variables
    {
        //varaibles that need to be accessed all around the code
        public List<Gate_Class> Gate_List { get; set; } = new List<Gate_Class>();
        public List<Line_Class> Line_List { get; set; } = new List<Line_Class>();
        //make these custom classes
```

```
        public List<Input_Button> Input_Button_List { get; set; } = new
List<Input_Button>();
        public List<Output_Circle> Output_Circle_List { get; set; } = new
List<Output_Circle>();
        private File_Creation_Class File_Worker { get; set; }
```

Instance variables are variables that can be accessed in the object they are in, so the class. A good way of thinking about it is that global variables can be declared in one locations and accessed in another without any links, instance variables can't do this, and they required to be passed by attributes or interfaces.

Java but still covers the topic: https://www.quora.com/What-is-the-difference-between-global-variables-and-instance-variables-in-Java

Structure of project

The MainWindow is the main part of the project. It holds all the data and has most of the methods for the interactions. This is why a lot of the other class will have MainWindow as an attribute. This is bad practice if the mainwindow is bidirectional. So I made sure that the variable doesn't have a setter and it is only unidirectional.

The final setup for how each class was based off is due to how variables linked together and their composition with each other. This is because I could've put input_Button and Output_Circle inside gate class but that doesn't always make sense as a gate might have no input or output buttons but the gate will still waste the memory hold variables that aren't used. It's also the same case for the line class as they are always created on the output port.

I keep all my variables at the top of the class then the constructor. For the ordering of methods it's just based on when I created them.

I try and kept the amount of code in one method to a minimum, make functions for any overlapping methods and a method should only have one function. I think I achieved this requirements. Some methods might be deemed too long to by some but I personally don't like having methods that are 4-6 lines per each one.

The classes that the user interact with are all stored in list because new objects can be added without a limit and removed to never be returned.

The program is split into lots of documents. Each class gets their own document and if it's a group they get their own folder. This is for ease of access and readability of the class. There is also external documents that store the images for the gates. This is done with a xaml file structure and allows easy changes and third party software. It's also important for vector images and the rescaling factor that it provides.

MainWindow

Structure of class and variables:

The MainWindow is the default class that the code will start with. It inherits window but also an interface called canvas variables. This is because C# doesn't support multiple inheritance canvas class which should inherit MainWindow but can't due to it already

inheriting Canvas needs some variables. So I send them as an interface as those are the overlapping variables needed between those 2 classes.

The Variables are all laid out on the top with the constructor underneath. With the variables they should all be private as MainWindow never inherits (So really making them public, protect or private makes zero difference in this class.) however due to the state of Visual Studio 2019 a problem means I can't change them off public. So the problem is in an interface to change the declaration type to private instead of the default public you need C# 8.0 or greater. I'm using .net framework and the default C# is 7.4. You would think you could just change the C# version to 8.0 or greater as the framework does support it but VS19 doesn't. Microsoft says to change it manually in the program file but reports says it doesn't work and gets override (I didn't try as I don't even know where to start with messing that file type).

With the variables in this class they are the main data type and data storage for everything in the program. This is the backbone that every class works between and due to C# not support multiple inheritance I'm using composition classes (This causes a problem with the file system). I also have the custom getter and setter for Objects that need them.

```
public partial class MainWindow : Window, Canvas_Variables
    {
        //varaibles that need to be accessed all around the code
        public List<Gate_Class> Gate_List { get; set; } = new List<Gate_Class>();
        public List<Line_Class> Line_List { get; set; } = new List<Line_Class>();
        //make these custom classes
        public List<Input_Button> Input_Button_List { get; set; } = new
List<Input_Button>();
        public List<Output_Circle> Output_Circle_List { get; set; } = new
List<Output_Circle>();
        private File_Creation_Class File_Worker { get; set; }

        //program info
        public bool Drag { get; set; } = false;
        public uint Delay_Intervals { get; set; } = 1;
        public int Drag_Num { get; set; } = 0;
        private bool _link = false;
        public bool Sim_Running { get; set; } = false;
        public bool Sim_Busy { get; set; } = false;
        public bool Saved { get; set; } = false;
        public string File_Location { get; set; } = "";
        public bool Link
        {
            get { return _link; }
            set
            {
                _link = value;
                if (_link)
                {
                    Link_Button.Content = "Link";
                }
                else if (!_link)
                {
                    Link_Button.Content = "Drag";
                }
            }
        }
```

```
        private Drag_State drag_mode = Drag_State.Null;
        public Drag_State Drag_Mode
        {
            get { return drag_mode; }
            set
            {
                switch (value)
                {
                    case (Drag_State.Null):
                        Drag = false;
                        break;
                    default:
                        Drag = true;
                        break;
                }

                drag_mode = value;
            }
        }
        public int Linking_ID { get; set; } = 0;
        public bool IO_Active { get; set; } = false;
        //UI elements that can't be added in the XAML
        public Canvas_Class Sub_Canvas { get; set; }
        public Rectangle BackGround_Rect { get; set; }

        //Threads that do the work simulatnously with the UI element
        private BackgroundWorker _worker = new BackgroundWorker();
        private BackgroundWorker Simulator_Worker = new BackgroundWorker();

        Progress_Bar_Window Progress_Window = new Progress_Bar_Window(0);
```

Constructor:

There are 2 stages to the constructor. The first is the Mainwindow constructor which just allocates the methods for the background workers. It also sets up the file class.

The second constructor is for when the canvas has loaded in the WPF. This is because canvas class requires other variables from both XAML and Mainwindow needed to be loaded to be passed through to the canvas class. Canvas Class couldn't be added to the XAML directly as it's a custom override class of a UI object they needed to be added in the CS code that's the reasoning for adding it separate from the other UI objects. I also added the rectangle to the canvas just to make the UI look better and fix the area that the user has to use.

```
        public MainWindow()
        {
            InitializeComponent();
            _worker.DoWork += WorkerDoWork;
            _worker.RunWorkerCompleted += WorkerRunWorkerCompleted;

            Simulator_Worker.DoWork += Simulator_Work;
            File_Worker = new File_Creation_Class(this);
        }

        //Need the UI to load before adding the canvas as it's added in the CS instead
of the XAML
        private void Canvas_Border_Loaded(object sender, RoutedEventArgs e)
```

```
        {
            Sub_Canvas = new Canvas_Class(this, ref Canvas_Border, this);
            SetUp_Canvas();
        }

        public void SetUp_Canvas()
        {
            Canvas_Border.Child = Sub_Canvas;
            BackGround_Rect = new Rectangle { Height = 4000, Width = 4000, Fill =
Brushes.White };
            Sub_Canvas.Children.Add(BackGround_Rect);
            Canvas.SetLeft(BackGround_Rect, -1000);
            Canvas.SetTop(BackGround_Rect, -1000);
        }
```

Methods:

`Rect_Button_MouseLeftButtonDown:`

This is the method for which you click the rectangles and a new one spawns in. The first IF statement just checks to see that you aren't already dragging or doing something right now and that you are in the correct mode for adding a new rectangle.

It then switches the drag mode from Null to Main_Can this just tells the program that the object being dragged is on the Main Canvas which covers the whole window. For my system I had 2 options. The gates could spawn on the canvas when the button is pushed and then the user can move to where they wanted or the gate could spawn on their click and they drag it to where they wanted. I liked the second option more because it saves the user from dragging the mouse to the middle of the screen and then going to where they want to place it. With the second option they just need to drag it to where they want. This might save someone half a second but it will save them from getting frustrated if they have to place lots of gates at once.

It then enters a switch case with a string condition. The string is based on the sender tag, as all the rectangular buttons are connected to the same method the switch case decipherers which new gate type should be. As the tags for the rectangles are hard coded the string inputs are enclosed and there should be any worry for an undetected gate type.

At the end it just updates the drag number so that the new gate is known to the system to be moved when the mouse moves.

```
private void Rect_Button_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
        {
            if (!Drag && !Link)
            {
                Drag_Mode = Drag_State.Main_Can;

                switch ((sender as Rectangle).Tag)
                {
                    case "And":
                        Gate_List.Add(new And_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                    case "Nand":
                        Gate_List.Add(new Nand_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
```

```
                        break;
                    case "Not":
                        Gate_List.Add(new Not_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                    case "Or":
                        Gate_List.Add(new Or_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                    case "Nor":
                        Gate_List.Add(new Nor_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                    case "Xor":
                        Gate_List.Add(new Xor_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                    case "Xnor":
                        Gate_List.Add(new Xnor_Gate_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                    case "Transformer":
                        Gate_List.Add(new Transformer_Class(Main_Canvas,
Sub_Canvas.Scale_Factor, Output_Circle_List, Line_List, Input_Button_List));
                        break;
                }

                Drag_Num = Gate_List.Count() - 1;

            }
        }
```

`Main_Canvas_MouseLeftButtonUp:`

While all this method does is call method `Add_Rect_Sub_FIX_BUG` this is due to Canvas class also needs to access the same code. Because XAML eventhandlers are private and have 2 parameters sender and eventarg this meant the canvas class couldn't access the method. So to fix this I just moved the code to a new method which both classes could access and then called the method in the event arg. While this is a botched solution it's required as you either have to write out the same code twice or create a new method.

```
private void Main_Canvas_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
        {
            if (Drag && !Link)
            {
                Add_Rect_Sub_FIX_BUG();
            }
        }
        /// <rectangle added to subcanvas>
        /// first check is to make sure it's inside the subcanvas/background_rect
border.
        /// then added it to the subcanvas and set the variables.
        /// </summary>
        public void Add_Rect_Sub_FIX_BUG()
        {
            Point Pos_Rect = Mouse.GetPosition(BackGround_Rect);
            Point Pos_Border = Mouse.GetPosition(Canvas_Border);
            Point Pos_Sub = Mouse.GetPosition(Sub_Canvas);
```

```
            (Detection_State State, int ThrowAway) =
Sub_Canvas.Rect_detection(Gate_List[Drag_Num].Rect.Width,
Gate_List[Drag_Num].Rect.Height, Drag_Num);

            //checks if it's in side the border and if it's inside the area that is
allowed in the border.
            bool check = (Pos_Rect.X < 0 || Pos_Rect.X > 4000 || Pos_Rect.Y < 0 ||
Pos_Rect.Y > 4000) ||
               (Pos_Border.X < 0 || Pos_Border.X > Canvas_Border.ActualWidth ||
Pos_Border.Y < 0 || Pos_Border.Y > Canvas_Border.ActualHeight)
               || (State == Detection_State.Detected) ? false : true;

            Main_Canvas.Children.Remove(Gate_List[Drag_Num].Rect);
            Drag_Mode = Drag_State.Null;
            if (check)
            {
                Gate_List[Drag_Num].Rect.Width = Gate_List[Drag_Num].Rect.Width /
Sub_Canvas.Scale_Factor;
                Gate_List[Drag_Num].Rect.Height = Gate_List[Drag_Num].Rect.Height /
Sub_Canvas.Scale_Factor;
                Sub_Canvas.Children.Add(Gate_List[Drag_Num].Rect);
                Gate_List[Drag_Num].Rect_Move(Pos_Sub);
            }
            else
            {
                Gate_List.RemoveAt(Drag_Num);
            }

        }
```

Link_Button_Click:

Just checks to make sure the user isn't dragging anything or trying to break the system then reverses the mode type. This always reverses it just not's the current value.

```
        private void Link_Button_Click(object sender, RoutedEventArgs e)
        {
            if (!Drag)
            {
                Link = !Link;
            }
        }
```

Input_Output_Button_Click:

This method is split into to two parts. When the button is active it needs and then pressed it needs to remove the inputs, When the button is disabled it needs to add any existing inputs and fill the rest that are missing some. This is seen in the first if statement.

The first IF statement it goes through the input and output lists and checks the gates that they're connected to are using the inputs. This is because the input and output buttons can be added then removed then replaced with a line but when added again the line gets priority so they are never added but remain in the list. Why I choose this is because I want the code to run as seeming less as possible and not freeze while it updates the whole data structure of the program just to remove 1 output from the list. If the gate is taking inputs it change the variable in the gate and remove it from canvas. The same happens for the output list.

The second part of the IF statement is a bit more complicated but it's just to make sure that the existing input and output buttons are added before new ones are added. I wanted this because I think it will be better for the user to be able to keep a certain set up and without it being reset when they didn't ask it to. It was also needed for saving files. If a user saves a file with input and output it allows it to be opened exactly as they left it.

So a method called `Add_Existing_IO` just goes through the list of inputs and outputs and finds out if the port that they are allocated to is free and that the gate is still alive. If that is true then the existing input/output is added to the canvas and the variables are updated on the gate.

For the new input and outputs, it goes through the list of gates and will first check if it's on the canvas or not. After that it will do 2 checks for the inputs, these checks are to see if the input port is free and depending on what type of gate it they might only have 1 input port instead of the standard 2.

It's the same for the outputs but the normal is 1 port but if it's the transformer gate it will have 3 so there is a check for each of those.

For adding the new input and outputs the code could be slightly neater and robust if you sacrifice memory. Each gate class could hold the information for which gates ports they have. This would result in 1 for loop and 1 generic if statement. This would make it more robust system as if you wanted to add any new gate class it would work without changing this method. But I don't plan on adding any more gate types as I cover all the ones needed. Also the additional memory that would be needed for every single even though they would just be constants wouldn't be worth it.

After all the inputs and outputs are added the gate will just calculate their output just in case it's a Nand gate where 2 zeros will result in a 1.

```
private void Input_Output_Button_Click(object sender, RoutedEventArgs e)
        {
            if (IO_Active)
            {
                IO_Active = false;
                for (int i = 0; i < Input_Button_List.Count; i++)
                {
                    if
(Gate_List[Input_Button_List[i].Input_ID].Input[Input_Button_List[i].Input_Port].Input
_Type == IO_Type.IO)
                    {

Gate_List[Input_Button_List[i].Input_ID].Input[Input_Button_List[i].Input_Port].Input_
Type = IO_Type.Null;
                        Sub_Canvas.Children.Remove(Input_Button_List[i]);
                    }
                }
                for (int i = 0; i < Output_Circle_List.Count; i++)
                {
                    if
(Gate_List[Output_Circle_List[i].Output_ID].Output[Output_Circle_List[i].Output_Port].
Output_Type == IO_Type.IO)
```

```
                    {
Gate_List[Output_Circle_List[i].Output_ID].Output[Output_Circle_List[i].Output_Port].O
utput_Type = IO_Type.Null;
                        Output_Circle_List[i].Remove_UI();
                    }
                }
            }
            else
            {
                Add_IO_Method();
            }
        }


        /// <Adding IO buttons>
        /// This is it's own method as when you load a file this part of the method
needs to be called to add the gates.
        /// the first part is just checking already existing gates in the
input_button_List and output_Circle_List
        /// and adding them to the canvas.
        /// The second part is just find out which gate on the screen still doesn't
have any IO and adds it.
        /// </summary>
        private void Add_IO_Method()
        {
            IO_Active = true;
            Add_Existing_IO();

            for (int i = 0; i < Gate_List.Count; i++)
            {
                if (Gate_List[i].Alive)
                {
                    if (Gate_List[i].Input[0].Input_Type == IO_Type.Null)
                    {
                        Input_Assignment(i, 0);
                    }
                    if (Gate_List[i].Input[1].Input_Type == IO_Type.Null &&
Gate_List[i].Type != Gate_Type.Not && Gate_List[i].Type != Gate_Type.Transformer)
                    {
                        Input_Assignment(i, 1);
                    }


                    if (Gate_List[i].Output[0].Output_Type == IO_Type.Null)
                    {
                        Output_Assignment(i, 0);
                    }
                    if (Gate_List[i].Output[1].Output_Type == IO_Type.Null &&
Gate_List[i].Type == Gate_Type.Transformer)
                    {
                        Output_Assignment(i, 1);
                    }
                    if (Gate_List[i].Output[2].Output_Type == IO_Type.Null &&
Gate_List[i].Type == Gate_Type.Transformer)
                    {
                        Output_Assignment(i, 2);
                    }
                    Gate_List[i].Gate_Output_Calc();
                }
            }
        }
```

```
        private void Add_Existing_IO()
        {
            for (int i = 0; i < Input_Button_List.Count; i++)
            {
                int ID = Input_Button_List[i].Input_ID;
                if (Gate_List[ID].Input[Input_Button_List[i].Input_Port].Input_Type ==
IO_Type.Null && Gate_List[ID].Alive)
                {
                    Gate_List[ID].Input[Input_Button_List[i].Input_Port].Input_Type =
IO_Type.IO;
                    Gate_List[ID].Input[Input_Button_List[i].Input_Port].Input_ID = i;
                    Gate_List[ID].Input[Input_Button_List[i].Input_Port].Input_bit =
Input_Button_List[i].Bit;
                    Sub_Canvas.Children.Add(Input_Button_List[i]);

Input_Button_List[i].Align_Box(Gate_List[Input_Button_List[i].Input_ID]);
                }

            }
            for (int i = 0; i < Output_Circle_List.Count; i++)
            {
                int ID = Output_Circle_List[i].Output_ID;
                if
(Gate_List[ID].Output[Output_Circle_List[i].Output_Port].Output_Type == IO_Type.Null
&& Gate_List[ID].Alive)
                {

Gate_List[ID].Output[Output_Circle_List[i].Output_Port].Output_Type = IO_Type.IO;
                    Gate_List[ID].Output[Output_Circle_List[i].Output_Port].Output_ID
= i;
                    Output_Circle_List[i].Add_UI();

Output_Circle_List[i].Align_Circle(Gate_List[Output_Circle_List[i].Output_ID]);
                }
            }
        }
        /// <summary>
        /// Adds a new input_Button to input_Button_List and sets the value of the
gate to fit the new values it should have.
        /// </summary>
        /// <param name="i"></just the position in the list, Gate Num>
        /// <param name="Port"></which input slot it needs to be allocated it>
        public void Input_Assignment(int i, int Port)
        {
            Input_Button_List.Add(new Input_Button(i, Port, Gate_List, Sub_Canvas,
this));
            //can't be in constructor because UI_Elemtent isn't loaded until the
constructor finished.
            Input_Button_List.Last().Align_Box(Gate_List[i]);
            Gate_List[i].Input[Port].Input_Type = IO_Type.IO;
            Gate_List[i].Input[Port].Input_ID = Input_Button_List.Count - 1;
        }
        //Same as input(Above)
        public void Output_Assignment(int i, int Port)
        {
            Output_Circle_List.Add(new Output_Circle(i, Port, Sub_Canvas, this));
            Output_Circle_List.Last().Align_Circle(Gate_List[i]);
            Gate_List[i].Output[Port].Output_Type = IO_Type.IO;
            Gate_List[i].Output[Port].Output_ID = Output_Circle_List.Count - 1;
        }
```

Link_Input_Align:

This is used to tell the lines and input buttons the position they need to go to be in line with their port number.

It's the same for the output Method

```
public double[] Link_Input_Align(Gate_Class Gate, int Input_Num)
        {
            //not gate, input is in the center of the gate compare to the other gates
whic has 2 on the side
            if (Gate.Type == Gate_Type.Not)
            {
                return new double[] { Canvas.GetLeft(Gate.Rect) + 5,
Canvas.GetTop(Gate.Rect) + 38 };
            }
            //this is similar to the not gate but because it's a sqaure not a
rectangle it needed to be moved in the X axis more
            else if (Gate.Type == Gate_Type.Transformer)
            {
                return new double[] { Canvas.GetLeft(Gate.Rect) + 12.5,
Canvas.GetTop(Gate.Rect) + 38 };
            }
            //the rest all follow the same setup as the and gate
            else
            {
                if (Input_Num == 0)
                {
                    return new double[] { Canvas.GetLeft(Gate.Rect),
Canvas.GetTop(Gate.Rect) + 15 };
                }
                //else if not needed here but Input_ID isn't a secure variable type.
                else if (Input_Num == 1)
                {
                    return new double[] { Canvas.GetLeft(Gate.Rect),
Canvas.GetTop(Gate.Rect) + 62 };
                }
            }
            return new double[] { -1, -1 };

        }
        //Same as input
        public double[] Link_Output_Align(Gate_Class Gate, int Output_Num)
        {
            //special gate class with 3 exit
            if (Gate.Type == Gate_Type.Transformer)
            {
                if (Output_Num == 0)
                {
                    return new double[] { Canvas.GetLeft(Gate.Rect) + 75,
Canvas.GetTop(Gate.Rect) + 23.8 };
                }
                else if (Output_Num == 1)
                {
                    return new double[] { Canvas.GetLeft(Gate.Rect) + 75,
Canvas.GetTop(Gate.Rect) + 36 };
                }
                else if (Output_Num == 2)
                {
                    return new double[] { Canvas.GetLeft(Gate.Rect) + 75,
Canvas.GetTop(Gate.Rect) + 51 };
                }
```

```
            }
            //not gate
            else if (Gate.Type == Gate_Type.Not)
            {
                return new double[] { Canvas.GetLeft(Gate.Rect) + 109.5,
Canvas.GetTop(Gate.Rect) + 36 };
            }
            //every other gate
            else
            {
                return new double[] { Canvas.GetLeft(Gate.Rect) + 115,
Canvas.GetTop(Gate.Rect) + 35.7 };
            }
            return new double[] { -1, -1 };
        }
```

`Delay_Lable_TextChanged:`

This is the event handler for the amount of time that the simulator has to wait before it can carry on working. There were 2 ways of doing this code. I could use try and catch statements or use Uint.TryPhase. I didn't use TryPhase as even if you inputted a value that isn't accepted it wouldn't crash but it would reset the value to the previous accepted value. This is because with tryphase it takes to parameters. The second one is a new unsigned bit variable. If the string is rejected then the new value of the variable is null. This cause lots of problems and then you're left with another if statement to check that the variable isn't null. So I choose the try and catch statements as then it only required 1 if statement and no additional variable to store the value.

In the try statement if the value isn't accepted the value of Delay_Intervals doesn't change then then when you change the text of the label it will go back to the last accepted value. This is perfect for what I want as it tells the user straight away that the value isn't acceptable.

```
private void Delay_Lable_TextChanged(object sender, TextChangedEventArgs e)
        {
            try
            {
                if (Delay_Lable.Text == "")
                {
                    Delay_Intervals = 0;
                    Delay_Lable.Text = "0";
                }
                Delay_Intervals = Convert.ToUInt32(Delay_Lable.Text);
            }
            catch
            {
            }
            Delay_Lable.Text = Convert.ToString(Delay_Intervals);
            Delay_Lable.CaretIndex = Delay_Lable.Text.Length;
        }
```

`Run_Button_Click:`

This is where the calculations for what the simulator is doing. This is what the user cares about so it needs to not freeze the UI and I didn't want the simulator to just be just an animation. This button is to start and stop the simulator at any time. Just like every other button it's based on a Boolean variable that is either active or not. If it's active it will just

change the Simulator running variable to false. Because of how background workers work this also updates the variable in the thread. This will be clearer in the method for the simulator. The button content also changes to fit the new state.

If the simulator is not running and is not busy then the background worker can start up again. The reason behind needing 2 variables to be able to tell what is happening is due to the simulator wait time. Because there is a wait between each cycle of the simulator the user could press the button to stop it and changing the variable to false, then pressing it again to activate the simulator again while the other one is still active. Also due to the variable now being true again the original simulator wouldn't close down. While this wouldn't cause a bug or break anything this isn't an acquirable solution. So there is 2 variables, one to tell the simulator to end the loop and close down, and another to let the program know that the background worker has closed. This is important that both ends have their own Boolean variable so that each can confirm with each other on the state of the program.

I did experiment with using Backgroundworker.IsBusy variable which is available to the background worker class but there is an error which I will cover in simulator_Work.

```
private void Run_Button_Click(object sender, RoutedEventArgs e)
        {
            if (Sim_Running)
            {
                Sim_Running = false;
                Run_Button.Content = "Run";
            }
            else if(!Sim_Running && !Sim_Busy)
            {
                Sim_Busy = true;
                Sim_Running = true;
                Run_Button.Content = "Stop";
                Clean_Up_Method();
                Add_IO_Method();
                Simulator_Worker.RunWorkerAsync();
            }
        }
```

Simulator_Work:

This is the actually code that the background worker will do. It first finds all the active starting points for the code. This is important as you can have multiple branches of circuits that aren't connected but you want them all to run. It's also important to remove any duplicating Start points. This is because it will be possible to have 2 starting gates together and if the ordering was to be first, second and first again you could override one of the steps in the simulation. It's also better for efficiency.

It then enters a while loop with 2 conditions. It will either run out of nodes and then then it will exit it or the user can stop the simulator and it will exit.

It then cycles through the active gates by calculating the new output and changing the colour to red to help the user see what is happening. It will also go through all the outputs of the gate and see if it's linked to another gate. If it is then will add the gate ID to the next

gate list and update its input. The code also includes await dispatcher to change the window UI. This is because the worker thread is not the parent thread so an invoke has to happen.

When it's gone through all the gates it will remove all the duplicated next gate ID. Waits the designated time set by the user. If this is zero then the UI objects will still update even without a task delay and it will cause a horrible effect if the circuit is in a loop but otherwise it will give you an output instantly. After the delay the UI will all be set back to black. The active gate list will equal the next gate list and a check will be done to see if there is still gates left to be calculated.

When it exits the while loop the content of the button will be changed to Run and sim busy will be false as the worker will close when it reaches the end of the block of code. Sim Busy needs to be a variable even though I could do Simulator_worker.IsBusy for the program to see if it's free. IsBusy is true when it's working and false if it's not. This is the exact same as Sim_Busy but it's more hard coded into the class for better programming. For what ever reason though, when you use a await inside of a background worker it will permentally change it to false. This kinda makes sense as the thread will hold to send the await message but after the await message is done it won't change back. This is a problem that I couldn't resolve on the internet.

Through out the code I try and use as many background worker variables as possible. This is why I use .cancellationpending and CancelAsync. These are more hard coded into the class and a more robust system when working in threads.

```
        private async void Simulator_Work(object sender, DoWorkEventArgs e)
         {
             List<int> Active_Gates = Set_Up_Start_Sim();

             while (!Simulator_Worker.CancellationPending)
             {
                 List<int> Next_Gate = new List<int>();

                 for (int i = 0; i < Active_Gates.Count; i++)
                 {
                     Gate_List[Active_Gates[i]].Gate_Output_Calc();
                     await
Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() => Gate_List[Active_Gates[i]].Rect.Stroke = Brushes.Red));
                     for (int x = 0; x < 3; x++)
                     {
                         if (Gate_List[Active_Gates[i]].Output[x].Output_Type ==
IO_Type.Gate)
                         {
                             await
Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() =>
Line_List[Gate_List[Active_Gates[i]].Output[x].Line_ID].Change_UI_Red()));

Gate_List[Line_List[Gate_List[Active_Gates[i]].Output[x].Line_ID].Input_ID].Input[Line
_List[Gate_List[Active_Gates[i]].Output[x].Line_ID].Input_Num].Input_bit =
Gate_List[Active_Gates[i]].Gate_Bit;

Next_Gate.Add(Line_List[Gate_List[Active_Gates[i]].Output[x].Line_ID].Input_ID);
                         }
                     }
```

```
                }

                Next_Gate = Remove_Duplicats(Next_Gate);

                await Task.Delay(Convert.ToInt32(Delay_Intervals));

                Set_Input_Back(Active_Gates);

                Active_Gates = Next_Gate.ToList();
                Next_Gate.RemoveRange(0, Next_Gate.Count);
                if (Active_Gates.Count == 0)
                {
                    Simulator_Worker.CancelAsync();
                    Sim_Running = false;
                }
            }
            await
Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() => Run_Button.Content = "Run"));
            Sim_Busy = false;
        }
        private List<int> Set_Up_Start_Sim()
        {
            List<int> Active_Gates = new List<int>();
            for (int i = 0; i < Input_Button_List.Count; i++)
            {
                if (Gate_List[Input_Button_List[i].Input_ID].Alive)
                {
                    Active_Gates.Add(Input_Button_List[i].Input_ID);
                }
            }
            for (int i = 0; i < Active_Gates.Count - 1; i++)
            {
                bool OverLap = false;
                for (int x = i + 1; x < Active_Gates.Count; x++)
                {
                    if (Active_Gates[x] == Active_Gates[i])
                    {
                        OverLap = true;
                    }
                }
                if (OverLap)
                {
                    Active_Gates.RemoveAt(i);
                    i -= 1;
                }
            }
            return Active_Gates;
        }

        private List<int> Remove_Duplicats(List<int> Next_Gate)
        {
            for (int i = 0; i < Next_Gate.Count - 1; i++)
            {
                bool OverLap = false;
                for (int x = i + 1; x < Next_Gate.Count; x++)
                {
                    if (Next_Gate[x] == Next_Gate[i])
                    {
                        OverLap = true;
                    }
                }
```

```
                if (OverLap)
                {
                    Next_Gate.RemoveAt(i);
                    i -= 1;
                }
            }
            return Next_Gate;
        }
        private async void Set_Input_Back(List<int> Active_Gates)
        {
            for (int i = 0; i < Active_Gates.Count; i++)
            {
                await
Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() => Gate_List[Active_Gates[i]].Rect.Stroke = Brushes.Black));
                for (int x = 0; x < 3; x++)
                {
                    if (Gate_List[Active_Gates[i]].Output[x].Output_Type ==
IO_Type.Gate)
                    {
                        await
Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() =>
Line_List[Gate_List[Active_Gates[i]].Output[x].Line_ID].Change_UI_Black()));
                    }
                }
            }
        }
```

Clean_Up_Method:

This is the method that is used to start the clean-up of the unused data and variables in the program.

The method is in the IF statement because if it's running and you do a clean-up between the wait and start of a new loop then there is a potential for the gates to become out of line with the list of active nodes. So I thought it would just be easier to remove that option when it's running. When you are removing data from the list everything becomes out of order with the links If you were to add a new gate and a new link during this process of everything being reorganized you have a possibility to get an out of range error. This is why I have the progress window and it's running a showdialog. The ShowDialog command just means that the new window halts the mainwindow thread until it is closed. This is why the background worker is there. It creates a new thread and does the work of the reorganizing. Its thread doesn't get halted by the showdialog so when it's finished its work it will remove the progress window and then unfreezing the MainWindow thread.

```
private void System_Clean_Up(object sender, RoutedEventArgs e)
        {
            Clean_Up_Method();
        }
        /// <summary>
        /// Creates a new window UI that hold the progress bar.
        /// Starts the background worker to clean up the system.
        /// The MainWindow is holded untill the system clean up is complete.
        /// </summary>
        public void Clean_Up_Method()
        {
            if (!Sim_Busy)
```

```
            {
                Progress_Window = new Progress_Bar_Window(Gate_List.Count());
                _worker.RunWorkerAsync();
                Progress_Window.ShowDialog();
            }
        }
        /// <summary>
        /// goes through all the list in the program and removes unused objects and
resorts the lists to work with the change.
        /// </summary>
        private void WorkerDoWork(object sender, DoWorkEventArgs e)
        {
            //remove unused input and output
            Remove_Unsed_Output();
            Remove_Unsed_Input();
            Remove_Unsed_Line();
            Remove_Unsed_Gate();
        }
```

Remove_Unsed_X:

The X just means that it's for the different type of methods all with roughly the same name. Each of the methods goes through the list of their type and removes any unused variables. It will also readjust the variables that they linked to.

```
private void Remove_Unsed_Output()
        {
            for (int i = 0; i < Output_Circle_List.Count; i++)
            {
                if
(Gate_List[Output_Circle_List[i].Output_ID].Output[Output_Circle_List[i].Output_Port].
Output_Type == IO_Type.Gate || Gate_List[Output_Circle_List[i].Output_ID].Alive ==
false)
                {
                    for (int x = 0; x < Gate_List.Count; x++)
                    {
                        if
(Gate_List[x].Output[Output_Circle_List[i].Output_Port].Output_ID > i &&
Gate_List[x].Output[Output_Circle_List[i].Output_Port].Output_Type == IO_Type.IO)
                        {

Gate_List[x].Output[Output_Circle_List[i].Output_Port].Output_ID -= 1;
                        }
                    }
                    Output_Circle_List.RemoveAt(i);
                    i -= 1;
                }
            }
        }
        private void Remove_Unsed_Input()
        {
            for (int i = 0; i < Input_Button_List.Count; i++)
            {
                if (!Gate_List[Input_Button_List[i].Input_ID].Alive)
                {
                    for (int x = 0; x < Gate_List.Count; x++)
                    {
                        if
(Gate_List[x].Input[Input_Button_List[i].Input_Port].Input_ID > i &&
Gate_List[x].Input[Input_Button_List[i].Input_Port].Input_Type == IO_Type.IO)
                        {
```

```
Gate_List[x].Input[Input_Button_List[i].Input_Port].Input_ID -= 1;
                        }
                    }
                    Input_Button_List.RemoveAt(i);
                    i -= 1;
                }
            }
        }
        private void Remove_Unsed_Line()
        {
            for (int i = 0; i < Line_List.Count; i++)
            {
                if
(Gate_List[Line_List[i].Input_ID].Input[Line_List[i].Input_Num].Input_Type !=
IO_Type.Gate || Gate_List[Line_List[i].Input_ID].Alive == false)
                {
                    for (int x = 0; x < Gate_List.Count; x++)
                    {
                        for (int y = 0; y < 3; y++)
                        {
                            if (Gate_List[x].Output[y].Line_ID > i)
                            {
                                Gate_List[x].Output[y].Line_ID -= 1;
                            }
                        }
                        for (int z = 0; z < 2; z++)
                        {
                            if (Gate_List[x].Input[z].Line_ID > i)
                            {
                                Gate_List[x].Input[z].Line_ID -= 1;
                            }
                        }
                    }
                    Line_List.RemoveAt(i);
                    i -= 1;
                }
            }
        }

        private void Remove_Unsed_Gate()
        {
            for (int i = 0; i < Gate_List.Count; i++)
            {
                if(!Gate_List[i].Alive)
                {
                    for (int x = 0; x < Gate_List.Count; x++)
                    {
                        for (int z = 0; z < 3; z++)
                        {
                            if (Gate_List[x].Output[z].Output_Type==IO_Type.Gate &&
Gate_List[x].Output[z].Output_ID>i)
                            {
                                Gate_List[x].Output[z].Output_ID -= 1;
                            }
                        }
                        for (int z = 0; z < 2; z++)
                        {
                            if (Gate_List[x].Input[z].Input_Type == IO_Type.Gate &&
Gate_List[x].Input[z].Input_ID > i)
                            {
                                Gate_List[x].Input[z].Input_ID -= 1;
```

```
                        }
                    }
                }
                for (int z = 0; z < Input_Button_List.Count; z++)
                {
                    if(Input_Button_List[z].Input_ID>i)
                    {
                        Input_Button_List[z].Input_ID -= 1;
                    }
                }
                for (int z = 0; z < Output_Circle_List.Count; z++)
                {
                    if(Output_Circle_List[z].Output_ID >i)
                    {
                        Output_Circle_List[z].Output_ID -= 1;
                    }
                }
                for (int z = 0; z < Line_List.Count; z++)
                {
                    if(Line_List[z].Output_ID>i)
                    {
                        Line_List[z].Output_ID -= 1;
                    }
                    if(Line_List[z].Input_ID>i)
                    {
                        Line_List[z].Input_ID -= 1;
                    }
                }
                if(Sub_Canvas.Last_Rect>i)
                {
                    Sub_Canvas.Last_Rect -= 1;
                }
                Gate_List.RemoveAt(i);
                i -= 1;
            }
        }
    }
```

MenuItem_New_X:

For each of the menu methods they are linked to the methods that are found in the File_Creation class. This is because mainwindow class was doing too much and it didn't link directly to what MainWindow is trying to achieve.

```
private void WorkerRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
    {
        Progress_Window.Close();
    }
    private void MenuItem_Load_Click(object sender, RoutedEventArgs e)
    {
        File_Worker.MenuItem_Load_Click_Method();
    }
    private void MenuItem_SaveAs_Click(object sender, RoutedEventArgs e)
    {
        File_Worker.MenuItem_SaveAs_Click_Method();
    }

    private void MenuItem_Save_Click(object sender, RoutedEventArgs e)
    {
        File_Worker.MenuItem_Save_Click_Method();
    }
    private void MenuItem_New_Click(object sender, RoutedEventArgs e)
```

```
        {
            File_Worker.MenuItem_New_Click_Method();
        }

        private void MenuItem_Exit_Click(object sender, RoutedEventArgs e)
        {
            Environment.Exit(0);
        }
```

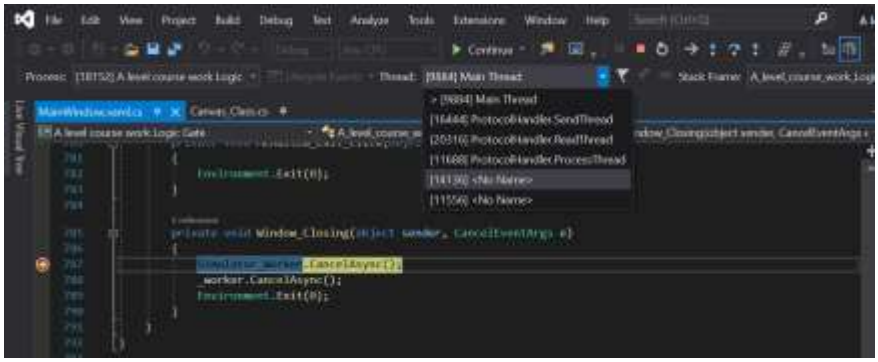
`Reset_Program:`

For resetting the program I had 3 options. I could reload the whole window and just remove all data, I could make MainWindow equal a new MainWindow or change the variables of MainWindow. I didn't want to reload the MainWindow as that would take a couple of seconds for the window to be rendered again. Just making the MainWindow equal a new mainwindow would reset everything without the lag but it would also reset variables like the time delay which the user would think wouldn't change. So I choose just to reset the variables that need to be reset in this function. This gives the quickest solution and also most desirable for the user.

```
public void Reset_Program()
        {
            Gate_List = new List<Gate_Class>();
            Line_List = new List<Line_Class>();
            Input_Button_List = new List<Input_Button>();
            Output_Circle_List = new List<Output_Circle>();
            Drag = false;
            Delay_Intervals = 1;
            Drag_Num = 0;
            _link = false;
            Sim_Running = false;
            Saved = false;
            File_Location = "";
            drag_mode = Drag_State.Null;
            Linking_ID = 0;
            IO_Active = false;
            Sub_Canvas = new Canvas_Class(this, ref Canvas_Border, this);
            SetUp_Canvas();
        }
```

`Window_Closing:`

This method is required as the program won't stop running if you just press close on the window. This is due to the threads (Don't know what the threads are but there there) running in the background while the mainthread has closed. So that's why the background workers are closed and then the program does `Environment.Exit(0);`. It's not good practice but I have no clue what these unnamed threads are.



Canvas Class

Structure of Class and variables:

Canvas Class is the custom UI class that the user places all their objects on. It can do this because it inherits the class from the WPF Canvas. This gives the class access to custom event handlers and features that can directly change the way it is presented. The variables that don't overlap with mainwindow are declared in the class and the ones that do are sent over as an attribute under the interface Canvas_Variables. I'm also sending the mainwindow over as canvas class needs to access methods held in there. This isn't a huge problem as it's only got a getter and no setter so there is no chance for accidental changes in the code.

In the constructor the attributes are linked to the variables of the class. Then the canvas is set up. The first part just makes the background grey. This servers 2 purposes, the first is to let the user know when they are out of bounds from the canvas, the second is due with how the detection works with WPF. When there is a colour change in the UI on the mouse cursor the UI will fire MouseMove events. It's really weird why this happens but a simple fix is just to change the colour of 2 objects by 1 bit colour or just do a completely different colour.

The second part is the rendering of the canvas. It has the scaler and translation. The scaler is used when the user zooms in and out and that works from a point. The translation is for when the user drags the canvas across the screen. These are both added to a transformation group so that they work together and don't over right each other.

```
public class Canvas_Class : Canvas
    {
        //variabls for the class
        public Point Old_Pos { get; set; } = new Point();
        public double Scale_Factor { get; set; } = 1;
        public Point Old_Rect { get; set; } = new Point();

        public TranslateTransform Translate_Action { get; set; }
        public ScaleTransform Scale_Action { get; set; }

        public TransformGroup Transforms_Group { get; set; }

        public bool MovingCanvas { get; set; } = false;
        public Border _Canvas_Border { get; set; }
        public int Last_Rect { get; set; } = -1;

        public Canvas_Variables variables;
        private MainWindow _MainWind { get; }
```

```
/// <summary>
///  The constructor just sets the value for the canvas and adds the transformation
set up.
/// </summary>
/// <param name="Variables"></This a more secure way of accessing and transfering
varaibles in a bidirectional way>
/// <param name="Canvas_Border"></Needed as a reference point for mouse postion>
/// <param name="MainWind"></Needed to call methods in the class>
        public Canvas_Class( Canvas_Variables Variables, ref Border Canvas_Border,
MainWindow MainWind)
        {
            variables = Variables;
            _Canvas_Border = Canvas_Border;
            _MainWind = MainWind;

            Background = Brushes.Gray;
            Translate_Action = new TranslateTransform(0, 0);
            Scale_Action = new ScaleTransform(1, 1, 0, 0);
            Transforms_Group = new TransformGroup();

            Transforms_Group.Children.Add(Translate_Action);
            Transforms_Group.Children.Add(Scale_Action);
            RenderTransform = Transforms_Group;
        }
```

OnMouseMove:

In mouse move event 2 things need to happen, if it's over a gate that gate border should highlight to let the user know that they can interact with it and if they are moving the canvas then the new position for the canvas should be used.

With the rectangle lighting up it need to change the old one to black this is why there is the IF statement.

For the moving of canvas it's based on the last known position of the mouse. It works out the difference and just translate it by that much.

```
protected override void OnMouseMove(MouseEventArgs e)
        {
            Detection_State State;
            int ID;
            (State, ID) = Rect_detection(0, 0, -1);
            if (State == Detection_State.Detected)
            {
                if (Last_Rect != -1)
                {
                    variables.Gate_List[Last_Rect].Rect.Stroke = Brushes.Black;
                }
                variables.Gate_List[ID].Rect.Stroke = Brushes.LightGreen;
                Last_Rect = ID;
            }

            if (MovingCanvas)
            {
                Translate_Action.X += (e.GetPosition(this).X - Old_Pos.X);
                Translate_Action.Y += (e.GetPosition(this).Y - Old_Pos.Y);
            }
            Old_Pos = e.GetPosition(this);
        }
```

OnMouseWheel:

For the zooming of the canvas I had to use a number which could be done fully with a floating point number. That's why I'm using 2^-4 as my scaling number. This is because any multiple of this number can be used as floating point and a rounding error won't occur.

There is a Boolean variable just to make sure there was a change, this is because although it won't change the scale of the canvas the middle with be repositioned as for the scale factor method I'm doing the where it's from the middle of the cursor.

```
protected override void OnMouseWheel(MouseWheelEventArgs e)
        {
            bool change = false;
            Point Pos = Mouse.GetPosition(_Canvas_Border);
            if (e.Delta > 0)
            {
                if (Scale_Factor != 0.0625)
                {
                    Scale_Factor -= 0.0625;
                    change = true;
                }
            }
            else if (e.Delta < 0)
            {
                if (Scale_Factor != 2)
                {
                    Scale_Factor += 0.0625;
                    change = true;
                }
            }

            if (change)
            {
                Scale_Action.ScaleX = Scale_Factor;
                Scale_Action.ScaleY = Scale_Factor;
                Scale_Action.CenterX = Pos.X;
                Scale_Action.CenterY = Pos.Y;
            }
        }
```

OnMouseLeftButtonDown:

3 things can happen when you click down, activate canvas moving, start to move a gate or a line will be added and start to be dragged.

For moving the canvas you just need to be clicking on nothing and can't be dragging anything. If this is true then it will change the variable for dragging canvas and the old position will be recorded so that it is the most up-to-date position of the canvas.

If it's moving a gate then you need to click a gate, not be in link mode and not currently dragging anything. This will update the drag number, change the drag mode and store the old position of the rectangle. This is because if you fail to move it to a new location then it will be sent back to the old position.

For adding a new line all the same conditions apply but it should be in link mode. It will then add a new line and check that the gate that is selected has a free output.

```
protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
        {
```

```
            (Detection_State State, int detected) = Rect_detection(0, 0, -1);

            if(!variables.Drag &&State == Detection_State.Null)
            {
                Old_Pos = e.GetPosition(this);
                MovingCanvas = true;
            }

            if (!variables.Drag && !variables.Link && State ==
Detection_State.Detected)
            {
                variables.Drag_Num = detected;
                variables.Drag_Mode = Drag_State.Sub_Can;
                Old_Rect = new
Point(Convert.ToDouble(Canvas.GetLeft(variables.Gate_List[variables.Drag_Num].Rect)),
Convert.ToDouble(Canvas.GetTop(variables.Gate_List[variables.Drag_Num].Rect)));
            }
            else if (!variables.Drag && variables.Link && State ==
Detection_State.Detected)
            {
                Line_M1_Down(detected);
            }
        }
        private void Line_M1_Down(int detected)
        {
            variables.Linking_ID = detected;
            variables.Drag_Num = variables.Line_List.Count();
            variables.Line_List.Add(new Line_Class(detected, this, _MainWind,
detected, true));
            //if X == -2 then there was no aviable output for the new link to be
made(already deleted the last line)
            if (variables.Line_List.Last().Output_Num != -2)
            {
                variables.Drag_Mode = Drag_State.Link_Mode_Sub;

variables.Gate_List[detected].Output[variables.Line_List.Last().Output_Num].Line_ID =
variables.Drag_Num;

variables.Line_List[variables.Drag_Num].Link_Output_Align_Line(variables.Gate_List[det
ected]);
            }
            else
            {
                variables.Line_List[variables.Drag_Num].Remove_Class();
            }
        }
```

OnMouseLeftButtonUp:

This does the opposite of mouse down. It will deactivate all current processes that are taken place.

The first IF statement is used to just fix a weird bug that would occur due to the ordering of the event handlers. Normally the event handler should be based on when they were added. As mainwindow left mouse click is added before the sub canvas left mouse click the mainwindow event handlers should get priority. But due to how mainwindow area overlaps with subcanvas this doesn't always happens. So I've just got an IF condition so that it will redirect it to the MainWindow function in this circumstance.

The second IF statement if you are dragging a rectangle. If the mouse is over a rectangle and there is overlap then the rectangle will move back to the old position otherwise it will be dropped where it is.

The third IF statement is for new lines. It will work out what input port it should go to base on the user. If it is full then it will try and find the next available port. If there is no available ports then the line will be removed. Otherwise it will change all the variables needed to know what the new links are between gates and lines.

```
protected override void OnMouseLeftButtonUp(MouseButtonEventArgs e)
        {
            MovingCanvas = false;
            if(variables.Drag && !variables.Link && variables.Drag_Mode ==
Drag_State.Main_Can)
            {
                _MainWind.Add_Rect_Sub_FIX_BUG();
            }

            if (variables.Drag && !variables.Link)
            {
                variables.Drag_Mode = Drag_State.Null;
                (Detection_State State, int Null) =
Rect_detection(variables.Gate_List[variables.Drag_Num].Rect.Width,
variables.Gate_List[variables.Drag_Num].Rect.Height, variables.Drag_Num);
                if (State == Detection_State.Detected)
                {
                    variables.Gate_List[variables.Drag_Num].Rect_Move(Old_Rect);
                    variables.Gate_List[variables.Drag_Num].Move_IO();
                }
            }
            else if(variables.Drag && variables.Link)
            {
                Line_M1_Up();
            }
        }
        private void Line_M1_Up()
        {
            int X = -1;
            (Detection_State State, int detection) = Rect_detection(0, 0, -1);
            variables.Drag_Mode = Drag_State.Null;
            if (State == Detection_State.Detected && detection !=
variables.Linking_ID)
            {
                X = Link_Input_Vaildation(detection);
            }

            if (X != -1)
            {
                //This a block of code that is only done one time and isn't assocaited
with each other so making a method wouldn't make a lot of sense but it's a lot of just
nothing.
                variables.Line_List[variables.Drag_Num].Input_ID = detection;
                variables.Line_List[variables.Drag_Num].Input_Num = X;

variables.Line_List[variables.Drag_Num].Link_Input_Align_Line(variables.Gate_List[dete
ction]);
                variables.Gate_List[detection].Input[X].Input_Type = IO_Type.Gate;
                variables.Gate_List[detection].Input[X].Input_ID =
variables.Linking_ID;
                variables.Gate_List[detection].Input[X].Line_ID = variables.Drag_Num;
```

```
variables.Gate_List[variables.Linking_ID].Output[variables.Line_List.Last().Output_Num
].Output_ID = detection;

variables.Gate_List[variables.Linking_ID].Output[variables.Line_List.Last().Output_Num
].Output_Type = IO_Type.Gate;
            }
            else if (X == -1)
            {
                variables.Line_List[variables.Drag_Num].Remove_Class();
            }
        }

Rect_detection:
```

This method just goes through the whole gate list in a linear search and tests the ranges of the boundaries and finds out if there is an overlap. It's adjustable for different types of gates and different areas that need to be scanned. This is desirable because it means the function can be used for if you want to find if the cursor is over a gate or if the gate that is being added to the canvas is overlapping with another gate.

The method returns a tuple of an Enum and int. The Enum could be a Boolean expression just saying if it detected an overlap or not but it was easier to understand if it was an Enum. The int is the ID of the rectangle that was detected. This is -1 by default.

The search to detect if there is an overlap is based on the 2 top left corners of the different points. If the cursor is in the orange/blue area then there is an overlap.

The orange zone will only be there if you are testing an object overlapping. There will be no orange for testing the cursor. It does this by testing the top left corner for both the object and gate.

```
public (Detection_State,  int) Rect_detection(double Width, double Height,int
Drag_Num)
        {
            Detection_State State = Detection_State.Null;
            int Detected = -1;
            Point Pos_Sub = Mouse.GetPosition(this);
            for (int i = 0; i < variables.Gate_List.Count(); i++)
            {
                Gate_Class Rect = variables.Gate_List[i];
                double Rect_X = GetLeft(variables.Gate_List[i].Rect);
                double Rect_Y = GetTop(variables.Gate_List[i].Rect);
                if (Pos_Sub.X + Width > Rect_X && Pos_Sub.X < Rect_X + Rect.Rect.Width
&& Pos_Sub.Y + Height > Rect_Y && Pos_Sub.Y < Rect_Y + Rect.Rect.Height && i !=
Drag_Num && Rect.Alive)
```

```
                {
                    Detected = i;
                    State = Detection_State.Detected;
                }
            }
            return (State, Detected);
        }
```

OnMouseRightButtonUp:

For the right button it only needs 1 activation and then the event happens. This is on the button being released as it means if you accidently push it down you can move it off the object that you don't want to remove.

It only has to 2 purposes, to remove a gate or to remove a line.

To remove an object the program shouldn't be dragging anything and it needs to be on a gate.

If it's removing a gate it will change the alive state of the gate to false and if it was the last rectangle that was touched by the cursor (Which it should be but not guaranteed) then the variable will change to -1. The rectangle will be removed from the canvas but not from the list, this is because I don't want the UI to freeze while it resorts all the data.

It will then enter 2 for loops, the first is for the outputs of the gate. This will check if the output type was an output or a connection. If it's an output then it will just remove the output circle from the canvas. If it's a connection then it will remove the line from the canvas and reset the values of the input that it was connected too to null. For the second for loop it will do the same but for the inputs.

If it's removing a line from the gate it will always be the output line. If it's the transformer gate type then it needs to work out which to remove based on the position of the cursor. Once that is determined then it will reset the output port and the corresponding input port to null.

```
protected override void OnMouseRightButtonUp(MouseButtonEventArgs e)
        {
            (Detection_State State, int detection) = Rect_detection(0, 0, -1);
            //checks you're not dragging and that you clicked on a gate.
            if (!variables.Drag && State == Detection_State.Detected)
            {
                //see which mode of the program you're in because each one will have a
different action
                if (!variables.Link)
                {    //completely romoves the gate.
                    Remove_Gate(detection);
                }
                else if (variables.Link) //if in linked mode then it should act as tho
you're doing the oppersite of adding a connection.
                {
                    Remove_Line(detection);
                }
            }
        }
        private void Remove_Gate(int detection)
        {
```

```
            variables.Gate_List[detection].Alive = false;

            if(detection == Last_Rect)
            {
                Last_Rect = -1;
            }

            Children.Remove(variables.Gate_List[detection].Rect);
            for (int i = 0; i < 3; i++)  //this is for each output of the gate after
it's been removed
            {
                if (variables.Gate_List[detection].Output[i].Output_Type ==
IO_Type.Gate)
                { //removes the line "connecting" the 2 gates.

variables.Line_List[variables.Gate_List[detection].Output[i].Line_ID].Remove_UI();
                    for (int x = 0; x < 2; x++) // this is to determin which input the
gate is connected to
                    {
                        if
(variables.Gate_List[variables.Gate_List[detection].Output[i].Output_ID].Input[x].Inpu
t_ID == detection &&
variables.Gate_List[variables.Gate_List[detection].Output[i].Output_ID].Input[x].Input
_Type == IO_Type.Gate)
                        { //changes the state of the gate it's connected to too null
so that it can accept another input.

variables.Gate_List[variables.Gate_List[detection].Output[i].Output_ID].Input[x].Input
_Type = IO_Type.Null;
                        }
                    }
                    variables.Gate_List[detection].Output[i].Output_Type =
IO_Type.Null;
                }
                else if (variables.Gate_List[detection].Output[i].Output_Type ==
IO_Type.IO)
                {

variables.Output_Circle_List[variables.Gate_List[detection].Output[i].Output_ID].Remov
e_UI();
                    //remove the ellipses from the canvas in the output_Circle list
with the ID in the
                }
            }
            for (int i = 0; i < 2; i++) //does the same but for input
            {
                if (variables.Gate_List[detection].Input[i].Input_Type ==
IO_Type.Gate)
                {

variables.Line_List[variables.Gate_List[detection].Input[i].Line_ID].Remove_UI();
                    for (int x = 0; x < 3; x++)
                    {
                        if
(variables.Gate_List[variables.Gate_List[detection].Input[i].Input_ID].Output[x].Outpu
t_ID == detection)
                        {

variables.Gate_List[variables.Gate_List[detection].Input[i].Input_ID].Output[x].Output
_Type = IO_Type.Null;
                        }
                    }
```

```
                }
                else if (variables.Gate_List[detection].Input[i].Input_Type ==
IO_Type.IO)
                {

Children.Remove(variables.Input_Button_List[variables.Gate_List[detection].Input[i].In
put_ID]);
                }
            }
        }

        private void Remove_Line(int detection)
        {
            int Output_Num = Output_Slot(detection);
            if (variables.Gate_List[detection].Output[Output_Num].Output_ID != -1 &&
variables.Gate_List[detection].Output[Output_Num].Output_Type == IO_Type.Gate)
            {
                for (int i = 0; i < 2; i++)
                {
                    if
(variables.Gate_List[variables.Gate_List[detection].Output[Output_Num].Output_ID].Inpu
t[i].Input_ID == detection)
                    {

variables.Gate_List[variables.Gate_List[detection].Output[Output_Num].Output_ID].Input
[i].Input_Type = IO_Type.Null;
                    }
                }

variables.Line_List[variables.Gate_List[detection].Output[Output_Num].Line_ID].Remove_
UI();
                variables.Gate_List[detection].Output[Output_Num].Output_Type =
IO_Type.Null;
            }
        }
```

Canvas Interface

While I cover above that you need C# 8.0 to get private variables in the interface you also need C#8.0 to store methods. This is the `Add_Rect_Sub_FIX_BUG` method due to it being required in canvas class. This is why canvas class requires both the interface and the whole MainWindow Class to be sent as attributes. To minimize the use of the MainWindow variable it only has a setter and is used just to call that method.

The main reason for using the interface to link between the 2 big classes is due to the overlapping variables and functions. Have a connection like this means that the canvas class can be more independent in the sense that if you took the class to another program you would only need to have the interface integrated in to have a working solution.

The variables that it holds is just program states and the list of the UI objects types.

```
public interface Canvas_Variables
    {
        List<Gate_Class> Gate_List { get; set; }
        List<Line_Class> Line_List { get; set; }
        List<Input_Button> Input_Button_List { get; set; }
        List<Output_Circle> Output_Circle_List { get; set; }
        bool Drag { get; set; }
        bool Link { get; set; }
```

```
        int Drag_Num { get; set; }
        Drag_State Drag_Mode { get; set; }
        int Linking_ID { get; set; }
    }
```

File Creation Class and File Classes

File Creation Class:

This class has all the methods for the menu item option. It has Save as, Save, Load and new file. The only variable it needs is the MainWindow. While the mainwindow and file creation class are linked closely together neither can inherit the other as they both depend on each other. This is why there is an association of MainWindow in the class. No additional variables are required for the class other than the parameters for each method.

```
        MainWindow MainWind { get; }
        public File_Creation_Class(MainWindow _MainWind)
        {
            MainWind = _MainWind;
        }
```

MenuItem_Load_Click_Method:

Depending on if the current work has been saved or not will determine what happens in the method. If the work has been saved then it will tell you that it's saved then resets the program and then open the load menu. The load menu is just the openfiledialog that windows supply.

If the work hasn't already been saved then it will tell you that your work will be deleted if you carry on. If they don't want it deleted then it will cancel the open file method by changing a Boolean variable. If they do want to delete their work then it will reset the program without saving it.

When that all happens then it will open the openfiledialog window where the user can pick the file they want to test. Once they have picked the file they want to open it will try and read it. The whole opening part of the program is in a try and catch state. This is so that if they open the wrong file they will get an error message.

If the file is accepted then it will be sent to another method that will unload the file. This is basically just decompiling the file class which can be serialized back into the class that the program uses. The order in which the decompile works is important. Because of classes depend on others only certain variables can be set at a time. It's really long lists of variables being set and isn't pleasant to look at but it's the only way. The compiler of the class can't be changed just to fit the unload needs and creating a new function doesn't server much purpose as this set of code is only used here (only reason would be to make it easier to look at and break down each part into smaller chunks but I don't mind myself because nothing of value is to be looked at here other than setting variables).

```
public void MenuItem_Load_Click_Method()
        {
            bool CarryOn = true;
            if (MainWind.File_Location != "")
            {
```

```
                Save_File();
                MainWind.Reset_Program();
                var result = MessageBox.Show("Your work has been saved", "New File",
MessageBoxButton.OK);
            }
            else
            {
                var result = MessageBox.Show("You haven't saved your work, Do you want
to delete it?", "New File", MessageBoxButton.OKCancel);
                if (result == MessageBoxResult.OK)
                {
                    MainWind.Reset_Program();
                }
                else
                {
                    CarryOn = false;
                }
            }
            if (CarryOn)
            {
                OpenFileDialog openFileDialog = new OpenFileDialog();
                if (openFileDialog.ShowDialog() == true)
                {
                    openFileDialog.InitialDirectory = @"C:\Documents";
                    MainWind.File_Location = openFileDialog.FileName;
                    Stream stream = new FileStream(MainWind.File_Location,
FileMode.Open, FileAccess.Read);
                    IFormatter formatter = new BinaryFormatter();
                    try
                    {
                        File_Class Loaded_File =
(File_Class)formatter.Deserialize(stream);
                        File_Unload(Loaded_File);
                    }
                    catch
                    {
                        MessageBox.Show("The file you tried to open doesn't work with
this program", "Failed", MessageBoxButton.OK);
                    }
                }

            }
        }

public void File_Unload(File_Class Loaded_File)
        {
            for (int x = 0; x < Loaded_File.Inputs.Count; x++)
            {
                MainWind.Input_Button_List.Add(new
Input_Button(Loaded_File.Inputs[x].Input_ID, Loaded_File.Inputs[x].Input_Port,
MainWind.Gate_List, MainWind.Sub_Canvas, MainWind));
                MainWind.Input_Button_List.Last().Change_X_Y(Loaded_File.Inputs[x].X,
Loaded_File.Inputs[x].Y);
            }
            for (int x = 0; x < Loaded_File.Output.Count; x++)
            {
                MainWind.Output_Circle_List.Add(new
Output_Circle(Loaded_File.Output[x].Output_ID, Loaded_File.Output[x].Output_Port,
MainWind.Sub_Canvas, MainWind));
                MainWind.Output_Circle_List.Last().Change_X_Y(Loaded_File.Output[x].X,
Loaded_File.Output[x].Y);
            }
```

```
            for (int x = 0; x < Loaded_File.Lines.Count; x++)
            {
                MainWind.Line_List.Add(new Line_Class(Loaded_File.Lines[x].Output_ID,
MainWind.Sub_Canvas, MainWind, Loaded_File.Lines[x].Input_ID, false));
                MainWind.Line_List.Last().Output_Num =
Loaded_File.Lines[x].Output_Num;
                MainWind.Line_List.Last().Input_Num = Loaded_File.Lines[x].Input_Num;
                MainWind.Line_List.Last().Line_Lable.Content =
Loaded_File.Lines[x].Content_Copy;
                MainWind.Line_List.Last().UI_Line.X1 = Loaded_File.Lines[x].X1;
                MainWind.Line_List.Last().UI_Line.X2 = Loaded_File.Lines[x].X2;
                MainWind.Line_List.Last().UI_Line.Y1 = Loaded_File.Lines[x].Y1;
                MainWind.Line_List.Last().UI_Line.Y2 = Loaded_File.Lines[x].Y2;
                MainWind.Line_List.Last().X1 = Loaded_File.Lines[x].X1;
                MainWind.Line_List.Last().X2 = Loaded_File.Lines[x].X2;
                MainWind.Line_List.Last().Y1 = Loaded_File.Lines[x].Y1;
                MainWind.Line_List.Last().Y2 = Loaded_File.Lines[x].Y2;
                MainWind.Line_List.Last().UI_Line.Stroke = Brushes.Black;
                MainWind.Line_List.Last().Move_Label();
            }
            for (int i = 0; i < Loaded_File.Gates.Count; i++)
            {
                switch (Loaded_File.Gates[i].Type)
                {
                    case (Gate_Type.And):
                        MainWind.Gate_List.Add(new
And_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Nand):
                        MainWind.Gate_List.Add(new
Nand_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Not):
                        MainWind.Gate_List.Add(new
Not_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Or):
                        MainWind.Gate_List.Add(new Or_Gate_Class(MainWind.Main_Canvas,
MainWind.Sub_Canvas.Scale_Factor, MainWind.Output_Circle_List, MainWind.Line_List,
MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Nor):
                        MainWind.Gate_List.Add(new
Nor_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Xor):
                        MainWind.Gate_List.Add(new
Xor_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Xnor):
                        MainWind.Gate_List.Add(new
Xnor_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    case (Gate_Type.Transformer):
```

```
                        MainWind.Gate_List.Add(new
Transformer_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                    default:
                        MainWind.Gate_List.Add(new
And_Gate_Class(MainWind.Main_Canvas, MainWind.Sub_Canvas.Scale_Factor,
MainWind.Output_Circle_List, MainWind.Line_List, MainWind.Input_Button_List));
                        break;
                }
                MainWind.Gate_List.Last().Alive = Loaded_File.Gates[i].Alive;
                MainWind.Gate_List.Last().Gate_Bit = Loaded_File.Gates[i]._Gate_Bit;
                for (int x = 0; x < 3; x++)
                {
                    MainWind.Gate_List.Last().Output[x].Output_ID =
Loaded_File.Gates[i].Output[x]._output_ID;
                    MainWind.Gate_List.Last().Output[x].Line_ID =
Loaded_File.Gates[i].Output[x]._line_ID;
                    MainWind.Gate_List.Last().Output[x].Output_Port =
Loaded_File.Gates[i].Output[x]._output_port;
                    MainWind.Gate_List.Last().Output[x].Output_Type =
Loaded_File.Gates[i].Output[x]._output_Type;
                }
                for (int x = 0; x < 2; x++)
                {
                    MainWind.Gate_List.Last().Input[x].Input_ID =
Loaded_File.Gates[i].Input[x]._Input_ID;
                    MainWind.Gate_List.Last().Input[x].Line_ID =
Loaded_File.Gates[i].Input[x]._line_ID;
                    MainWind.Gate_List.Last().Input[x].Input_Type =
Loaded_File.Gates[i].Input[x]._Input_Type;
                }
                MainWind.Main_Canvas.Children.Remove(MainWind.Gate_List.Last().Rect);
                MainWind.Sub_Canvas.Children.Add(MainWind.Gate_List.Last().Rect);
                Canvas.SetLeft(MainWind.Gate_List.Last().Rect,
Loaded_File.Gates[i].X);
                Canvas.SetTop(MainWind.Gate_List.Last().Rect, Loaded_File.Gates[i].Y);
            }
            for (int x = 0; x < Loaded_File.Inputs.Count; x++)
            {
                MainWind.Input_Button_List.Last().Bit = Loaded_File.Inputs[x]._Bit;
            }
        }
```

MenuItem_SaveAs_Click_Method:

SaveAs and Save methods are similar that the amount of overlapping code might as well be one method which they both call. There is only one difference and that is that save should check to see if the file has already been saved and just save it whereas SaveAs should no matter what open the file explorer and let you save it.

Save as method opens the SaveFileDialog and creates the file path and changes saved to true. It will then call save file method for which the overlap of SaveAs and Save code (Covered below).

```
public void MenuItem_SaveAs_Click_Method()
        {
            Save_AS();
```

```
        }
        private void Save_AS()
        {
            SaveFileDialog saveFileDialog = new SaveFileDialog();
            if (saveFileDialog.ShowDialog() == true)
            {
                MainWind.Saved = true;
                saveFileDialog.InitialDirectory = @"C:\Documents";
                MainWind.File_Location = saveFileDialog.FileName;
                Save_File();
            }
        }
```

Save_File:

In Save file this has the IF statement due to needing a file path to be able to save the file. If saved is true then the system will be cleaned up to reduce the file size as much as possible. The file will be decompressed into a file type that can be serialized. A new FileStream will be created with the designated file location. A formatter is also created with the binary file type used. It then saves the file as binary under the settings of the FileStream, after its finishes it will then close.

If the file hasn't been saved before then it will call the SaveAs method for which it will then be given a file path and then calls the method again and follows the process above.

```
public void MenuItem_Save_Click_Method()
        {
            Save_File();
        }
        public void Save_File()
        {
            if (MainWind.Saved)
            {
                MainWind.Clean_Up_Method();
                File_Class Save = File_Creation();
                Stream stream = new FileStream(MainWind.File_Location,
FileMode.Create, FileAccess.Write);
                IFormatter formatter = new BinaryFormatter();
                formatter.Serialize(stream, Save);
                stream.Close();
            }
            else
            {
                Save_AS();
            }
        }
```

File_Creation:

Because WPF objects can't be serialized custom classes that just store variables are created. These are used to reduce file space as well. They each have custom constructors so that the variables are assigned as soon at the class is created.

```
public File_Class File_Creation()
        {
            File_Class Save = new File_Class();
            for (int i = 0; i < MainWind.Gate_List.Count(); i++)
```

```
            {
                Save.Gates.Add(new File_Version_Gate(MainWind.Gate_List[i].Type,
MainWind.Gate_List[i].Alive, MainWind.Gate_List[i].Input,
MainWind.Gate_List[i].Gate_Bit, MainWind.Gate_List[i].Output,
Canvas.GetLeft(MainWind.Gate_List[i].Rect),
Canvas.GetTop(MainWind.Gate_List[i].Rect)));
            }
            for (int i = 0; i < MainWind.Line_List.Count; i++)
            {
                Save.Lines.Add(new
File_Version_Line(Convert.ToString(MainWind.Line_List[i].Line_Lable.Content),
MainWind.Line_List[i].Output_ID, MainWind.Line_List[i].Output_Num,
MainWind.Line_List[i].Input_ID, MainWind.Line_List[i].Input_Num,
MainWind.Line_List[i].X1, MainWind.Line_List[i].X2, MainWind.Line_List[i].Y1,
MainWind.Line_List[i].Y2));
            }
            for (int i = 0; i < MainWind.Input_Button_List.Count; i++)
            {
                Save.Inputs.Add(new
File_Version_Input(MainWind.Input_Button_List[i].Bit,
MainWind.Input_Button_List[i].Input_ID, MainWind.Input_Button_List[i].Input_Port,
Canvas.GetLeft(MainWind.Input_Button_List[i]),
Canvas.GetTop(MainWind.Input_Button_List[i])));
            }
            for (int i = 0; i < MainWind.Output_Circle_List.Count; i++)
            {
                Save.Output.Add(new
File_Version_Output(MainWind.Output_Circle_List[i].Output_ID,
MainWind.Output_Circle_List[i].Output_Port,
Canvas.GetLeft(MainWind.Output_Circle_List[i].Circle),
Canvas.GetTop(MainWind.Output_Circle_List[i].Circle)));
            }
            return Save;
        }
```

MenuItem_New_Click_Method:

Gives a yes no message box asking if you want to delete everything. If they click yes then it will reset the program without saving anything.

```
        public void MenuItem_New_Click_Method()
        {
            MessageBoxResult messageBoxResult = MessageBox.Show("Are you sure you want
to remove everything? Nothing will be kept!", "New Window", MessageBoxButton.YesNo);
            if (messageBoxResult == MessageBoxResult.Yes)
            {
                MainWind.Reset_Program();
            }
        }
```

Gate Classes

The Base class which each of the sub classes inherit off holds all methods and templates that the children classes should hold.

For the variables it holds the some characteristics about the properties of the gate state. It also hold the classes for inputs, output and output bit. The input and output classes are arrays because there is a fix amount of inputs and outputs that the gates can have.

For Gate_bit it has its own custom setter which means that as soon as the gate bit is set it will either change the output circle to the correct setting or update the variable of the gate it is linked to.

The base class has an abstract method called `Gate_Output_Calc` this is because each child class should override this method for each of their individual gate outputs.

```
public abstract class Gate_Class
    {
        public Gate_Type Type { get; set; }
        public Rectangle Rect { get; set; }
        public bool Alive { get; set; } = true;

        //data storages for the input and output

        public Input_Class[] Input = new Input_Class[] { new Input_Class(), new
Input_Class() };
        List<Output_Circle> _Output_Circle_List { get; set; }
        List<Line_Class> _Line_List { get; set; }
        List<Input_Button> _Input_Button_List { get; set; }


        /// <summary>
        /// depending on what type of output it is it will update and change the
corrosonding values.
        /// </summary>
        private bool _Gate_Bit;
        public  bool Gate_Bit {
            get
            { return _Gate_Bit; }
             set
            {
                _Gate_Bit = value;
                for (int i = 0; i < 3; i++)
                {
                    if(Output[i].Output_Type==IO_Type.IO)
                    {
                        if (_Gate_Bit == true)
                        {
                            _Output_Circle_List[Output[i].Output_ID].Bit = true;
                        }
                        else
                        {
                            _Output_Circle_List[Output[i].Output_ID].Bit = false;
                        }
                    }
                    else if(Output[i].Output_Type == IO_Type.Gate)
                    {
                        if(_Gate_Bit==true)
                        {
                            Set_Label_1(i);
                        }
                        else
                        {
                            Set_Label_0(i);
                        }
                    }
                }
            }
        }
```

```
        public Output_Class[] Output { get; set; } = new Output_Class[] { new
Output_Class(), new Output_Class(), new Output_Class() };



        public Gate_Class(List<Output_Circle> Output_Circle_List,List<Line_Class>
Line_List,List<Input_Button> Input_Button_List)
        {
            _Output_Circle_List = Output_Circle_List;
            _Line_List = Line_List;
            _Input_Button_List = Input_Button_List;
            Gate_Output_Calc();
        }

        protected void Part_Constructor(Canvas Main_Canvas)
        {
            Main_Canvas.Children.Add(Rect);
            Point Pos = Mouse.GetPosition(Main_Canvas);
            Rect_Move(Pos);
        }
```

Move_IO:

This is such a useful function as it means that the links that the gate has to each of its UI objects can be updated on a gate bases. All it does is go through each of the gate input and output and moves the UI object to be aligned with the new position of the gate.

```
public void Move_IO()
        {
            for (int i = 0; i < 3; i++)
            {
                if(Output[i].Output_Type == IO_Type.Gate)
                {
                    _Line_List[Output[i].Line_ID].Link_Output_Align_Line(this);
                }
                else if(Output[i].Output_Type == IO_Type.IO)
                {
                    _Output_Circle_List[Output[i].Output_ID].Align_Circle(this);
                }
            }
            for (int i = 0; i < 2; i++)
            {
                if (Input[i].Input_Type==IO_Type.Gate)
                {
                    _Line_List[Input[i].Line_ID].Link_Input_Align_Line(this);
                }
                else if(Input[i].Input_Type == IO_Type.IO)
                {
                    _Input_Button_List[Input[i].Input_ID].Align_Box(this);
                }
            }
        }
```

Children Classes

Each Children class has 3 things different about them. In their constructor they will have a unique setting for the rectangle UI. The height, width and fill will changed based on the type

of gate. The type will be different for each and the method that they need to override for the calculation of the output needs to be the truth table for their gate.

And:

```
        public And_Gate_Class(Canvas Main_Canvas, double _Scale_Factor,
List<Output_Circle> Output_Circle_List, List<Line_Class> Line_List, List<Input_Button>
Input_Button_List) : base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["And_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.And;
        }
        public override void Gate_Output_Calc()
        {
            if (Input[0].Input_bit && Input[1].Input_bit == true)
            {
                Gate_Bit = true;
            }
            else
            {
                Gate_Bit = false;
            }
        }
```

Nand:

```
        public Nand_Gate_Class(Canvas Main_Canvas, double _Scale_Factor,
List<Output_Circle> Output_Circle_List, List<Line_Class> Line_List, List<Input_Button>
Input_Button_List) : base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Nand_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Nand;
        }
        public override void Gate_Output_Calc()
        {
            if (Input[0].Input_bit && Input[1].Input_bit == true)
            {
                Gate_Bit = false;
            }
            else
            {
                Gate_Bit = true;
            }
        }
```

Not:

```
public Not_Gate_Class(Canvas Main_Canvas, double _Scale_Factor, List<Output_Circle>
Output_Circle_List, List<Line_Class> Line_List, List<Input_Button> Input_Button_List)
: base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Not_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Not;
```

```
        }
        public override void Gate_Output_Calc()
        {
            if (Input[0].Input_bit == true)
            {
                Gate_Bit = false;
            }
            else
            {
                Gate_Bit = true;
            }
        }
```

Or:

```
public Or_Gate_Class(Canvas Main_Canvas, double _Scale_Factor, List<Output_Circle>
Output_Circle_List, List<Line_Class> Line_List, List<Input_Button> Input_Button_List)
: base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Or_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Or;
        }
        public override void Gate_Output_Calc()
        {
            if (Input[0].Input_bit || Input[1].Input_bit == true)
            {
                Gate_Bit = true;
            }
            else
            {
                Gate_Bit = false;
            }
        }
```

Nor:

```
public Nor_Gate_Class(Canvas Main_Canvas, double _Scale_Factor, List<Output_Circle>
Output_Circle_List, List<Line_Class> Line_List, List<Input_Button> Input_Button_List)
: base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Nor_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Nor;
        }
        public override void Gate_Output_Calc()
        {
            if (Input[0].Input_bit || Input[1].Input_bit == true)
            {
                Gate_Bit = false;
            }
            else
            {
                Gate_Bit = true;
            }
        }
```

Xor:

```
public Xor_Gate_Class(Canvas Main_Canvas, double _Scale_Factor, List<Output_Circle>
Output_Circle_List, List<Line_Class> Line_List, List<Input_Button> Input_Button_List)
: base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Xor_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Xor;
        }
        public override void Gate_Output_Calc()
        {
            if ((Input[0].Input_bit == true && Input[1].Input_bit == true) ||
(Input[0].Input_bit == false && Input[1].Input_bit == false))
            {
                Gate_Bit = false;
            }
            else
            {
                Gate_Bit = true;
            }
        }
```

Xnor:

```
public Xnor_Gate_Class(Canvas Main_Canvas, double _Scale_Factor, List<Output_Circle>
Output_Circle_List, List<Line_Class> Line_List, List<Input_Button> Input_Button_List)
: base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 115 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Xnor_Gate_L"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Xnor;
        }
        public override void Gate_Output_Calc()
        {
            if ((Input[0].Input_bit == true && Input[1].Input_bit == true) ||
(Input[0].Input_bit == false && Input[1].Input_bit == false))
            {
                Gate_Bit = true;
            }
            else
            {
                Gate_Bit = false;
            }
        }
```

Transformer:

```
public Transformer_Class(Canvas Main_Canvas, double _Scale_Factor, List<Output_Circle>
Output_Circle_List, List<Line_Class> Line_List, List<Input_Button> Input_Button_List)
: base(Output_Circle_List, Line_List, Input_Button_List)
        {
            Rect = new Rectangle { Height = 75 * _Scale_Factor, Width = 85 *
_Scale_Factor, Stroke = Brushes.Black, Fill =
Application.Current.Resources["Transformer"] as Brush };
            Part_Constructor(Main_Canvas);
            Type = Gate_Type.Transformer;
        }
        public override void Gate_Output_Calc()
        {
            if (Input[0].Input_bit == true)
```

```
            {
                Gate_Bit = true;
            }
            else
            {
                Gate_Bit = false;
            }
        }
```

Input and Output Class

This is just an extension of gate class but it needed to be its own class because it's a group of data and repeating data due to it being in an array. The purpose is just to hold the value of each ports in the input and output of the gate.

For the Input/Output_Type variable it has a custom setter so that the other values changes with the appropriate state that it's in.

Output Class:

```
public class Output_Class
    {
        private int _output_ID = -1;
        private IO_Type _output_Type = IO_Type.Null;
        private int _line_ID = -1;
        public int Output_Port { get; set; }
        public int Output_ID {
            get {return _output_ID ; }
            set {_output_ID =value; } }

        public IO_Type Output_Type {
            get { return _output_Type; }
            set {
                switch(value)
                {
                    case (IO_Type.Null):
                        _output_ID = -1;
                        _line_ID = -1;
                        break;
                    case (IO_Type.IO):
                        _line_ID = -1;
                        break;
                }
               _output_Type = value;
            }
        }
        public int Line_ID {
            get {return _line_ID; }
            set { _line_ID = value; } }

        public Output_Class()
        {
            Output_Type = IO_Type.Null;
        }
    }
```

Input Class:

```
public class Input_Class
```

```
{
    private bool _input_bit = false;
    private int _input_ID = -1;
    private IO_Type _input_Type = IO_Type.Null;
    private int _line_ID = -1;
    public bool Input_bit {
        get { return _input_bit; }
        set { _input_bit = value; }
    }
    public int Input_ID {
        get { return _input_ID; }
        set { _input_ID = value; } }
    public IO_Type Input_Type
    {
        get { return _input_Type; }
        set {
            switch(value)
            {
                case (IO_Type.Null):
                    _input_ID = -1;
                    _line_ID = -1;
                    break;
                case (IO_Type.Gate):
                    _input_bit = false;
                    break;
                case (IO_Type.IO):
                    _line_ID = -1;
                    _input_bit = false;
                    break;
            }
            _input_Type = value;
        }
    }
    public int Line_ID {
        get { return _line_ID; }
        set { _line_ID = value; } }

    public Input_Class()
    {
        Input_Type = IO_Type.Null;
    }
}
```

Progress window

While there isn't much to say on a nearly empty window, this bit of the code is what the user will see when they aren't meant to interact with the MainWindow.

The only thing of use is to understand that the size of the work will change and so the progress bar is adjustable to however much is needed to be done.

```
public partial class Progress_Bar_Window : Window
{
    public Progress_Bar_Window(int max)
    {
        InitializeComponent();
        Bar.Minimum = 0;
        Bar.Value = 0;
        Bar.Maximum = max;
    }
}
```

Input Button Class

This is separate from input class as that is just a group of data that is linked to the input ports of the gate. This is about the UI objects that get added when you what to add your starting points. The Class inherits the button class so that it can have custom event handlers and all the variables needed for the input button class are in one place.

There is only 3 variables to this class because that's all it needs, the main one to look at is Bit. It is important because it has its own custom setter which will in real time update the gate output. This allows the user to test single gates truth tables without needing to run the simulator. However it won't then calculate the next gate after. This is the design I want because if it was to all change at once then the user wouldn't understand what happened and how they got that output.

So in the setter it will change the attributes of the button class that it inherits to fit the desired effect and change the gate class variable to fit the new change. It will also tell the gate to calculate the new input. This process doesn't repeat as when the gate sends the new output to another gate the class Input doesn't have the custom setter that will tell the gate to calculate a new value.

Input ID and Input Port is just the ID for the gate and the port slot that it is connected to.

Because the class inherits the Button class it allows for a custom click event for the input. It nots the current value, as this is Boolean it just turns into the other value. While all that changed is the variable it's due to the bit setter that also changes the visuals of the button and updates the gate.

```
public class Input_Button : Button
    {
        private bool _Bit = false;
        /// <summary>
        /// Changes the feature of the button for what input bit it is automatically.
        /// </summary>
        public bool Bit { get { return _Bit; }
            set
            {
                _Bit = value;
                if(!value)
                {
                    Content = 0;
                    Foreground = Brushes.Black;
                    Background = Brushes.White;
                    _Gate_List[Input_ID].Input[Input_Port].Input_bit = false;
                }
                else
                {
                    Background = Brushes.Black;
                    _Gate_List[Input_ID].Input[Input_Port].Input_bit = true;
                    Content = 1;
                    Foreground = Brushes.White;
                }
                _Gate_List[Input_ID].Gate_Output_Calc();
            }

        }
```

```
        public int Input_ID { get; set; }
        public int Input_Port;
        public List<Gate_Class> _Gate_List = new List<Gate_Class>();
        private MainWindow _MainWind { get; set; }
        public Input_Button(int ID, int Port_Num, List<Gate_Class> Gate_List,
Canvas_Class Sub_Canvas, MainWindow MainWind)
        {
            _MainWind = MainWind;
            _Gate_List = Gate_List;
            Input_ID = ID;
            Input_Port = Port_Num;
            Sub_Canvas.Children.Add(this);
            Background = Brushes.White;
            Content = 0;
            Foreground = Brushes.Black;
            Height = 20;
            Width = 20;
        }
        //make this bit depend. So when the bit variable changes so does everything
else.
        protected override void OnClick()
        {
            Bit = !Bit;
        }

        public void Align_Box(Gate_Class Gate)
        {
            double[] hold = _MainWind.Link_Input_Align(Gate, Input_Port);
            Change_X_Y(hold[0], hold[1]);
        }

        public void Change_X_Y(double x, double y)
        {
            Canvas.SetLeft(this, x-20);
            Canvas.SetTop(this, y-10);
        }
    }
```

Output Circle Class

Unlike the input class the circle class couldn't inherit the ellipsis class so it's just a variable of the class. The Class also contains a label for easy reading and understanding of what all the different colour system uses. As there is 2 UI Elements and they are both variables a few more methods were required. It also needs less variables as storing the bit of the value is pointless as it isn't separate from the gate bit it's connected to.

For the custom setter of Bit it updates the visual of the circle and label to relate to the new value. Because this will most like happen during the simulation part of the program which is working in a different thread from the main thread which has all the access to the WPF objects it needs to send a dispatch invoke. It calls a method as everything in that method just sets a different UI part. Before it was 3 lines of dispatching to change the values one at a time but now it does them all at ones.

Add and Remove UI are methods because I had lots of repeated code where they were being removed or added.

```
public class Output_Circle
    {
```

```
        public Ellipse Circle = new Ellipse { Height = 20, Width = 20, Fill =
Brushes.White, Stroke = Brushes.Black, StrokeThickness = 1 };
        public Label Output_Lab = new Label { Content = "0", Width = 16, Height = 29,
Foreground = Brushes.Black };

        public int Output_ID { get; set; }
        public int Output_Port;
        private MainWindow _MainWind { get; }
        public Output_Circle(int ID, int Port_Num, Canvas_Class Sub_Canvas,MainWindow
MainWind)
        {
            Output_ID = ID;
            Output_Port = Port_Num;
            Sub_Canvas.Children.Add(Circle);
            Sub_Canvas.Children.Add(Output_Lab);
            _MainWind = MainWind;
        }


        public bool Bit
        {
            set
            {
                if (value == false)
                {

Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() => Circle_0()));
                }
                else
                {

Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new
Action(() => Circle_1()));
                }
            }
        }
        private void Circle_0()
        {
            Circle.Fill = Brushes.White;
            Output_Lab.Foreground = Brushes.Black;
            Output_Lab.Content = "0";

        }

        private void Circle_1()
        {
            Circle.Fill = Brushes.Black;
            Output_Lab.Foreground = Brushes.White;
            Output_Lab.Content = "1";
        }

        public void Add_UI()
        {
            _MainWind.Sub_Canvas.Children.Add(Circle);
            _MainWind.Sub_Canvas.Children.Add(Output_Lab);
        }
        public void Remove_UI()
        {
            _MainWind.Sub_Canvas.Children.Remove(Circle);
            _MainWind.Sub_Canvas.Children.Remove(Output_Lab);
        }
```

```
        public void Align_Circle(Gate_Class Gate)
        {
            double[] hold = _MainWind.Link_Output_Align(Gate, Output_Port);
            Change_X_Y(hold[0], hold[1]);
        }

        public void Change_X_Y(double x, double y)
        {
            Canvas.SetLeft(Circle, x);
            Canvas.SetTop(Circle, y-10);
            Canvas.SetLeft(Output_Lab, x+2);
            Canvas.SetTop(Output_Lab, y-13);
        }

    }
```

Line Class

Line Class couldn't inherit the WPF Line Class so it's just a variable of the class. Like the same with Output Circle Class I have a label to let the user know what the output of the gate is.

All the rest of the variables are just to sort out the location and position of the line. The class doesn't need to store any values because when the gate class updates the input of another class it will just go straight to it.

In the constructor it has an IF statement, this is because when you load a file and it creates a new line class it will try and alien the output port with a gate. But as the gates have to be added last when the file is created the method can't find the gate that it's connected to. So the ordering is different if it's a new or loaded class.

```
public class Line_Class
    {
        public Line UI_Line { get; set; } = new Line { StrokeThickness = 4, Stroke =
Brushes.Red };
        public Label Line_Lable { get; set; } = new Label { Content = "0", Width = 16,
Height = 29, Foreground = Brushes.Black };
        public int Output_ID { get; set; } = -1;
        public int Output_Num { get; set; } = -1; //this should only change if the
gate type is the multiple output(type 7)
        public int Input_ID { get; set; } = -1;
        public int Input_Num { get; set; } = -1;
        public double X1,Y1,  X2,  Y2;
        public List<Line_Class> _Line_List { get; set; }

        public Canvas_Class _Sub_Canvas { get; set; }
        private MainWindow _MainWind { get; }
        public Line_Class(int _Output_ID, Canvas_Class Sub_Canvas, MainWindow
MainWind,int _Input_ID,bool New_Class)
        {
            _Sub_Canvas = Sub_Canvas;
            _MainWind = MainWind;
            _Sub_Canvas.Children.Add(UI_Line);
            _Sub_Canvas.Children.Add(Line_Lable);
            Output_ID = _Output_ID;
            _Line_List = MainWind.Line_List;
            Track_Mouse();
            Input_ID = _Input_ID;
            if(New_Class)
            {
                Output_Num = _Sub_Canvas.Link_Output_Vaildation(_Input_ID);
```

```
            }
        }

        public void Track_Mouse()
        {
            Point Pos = Mouse.GetPosition(_Sub_Canvas);
            UI_Line.X2 = Pos.X;
            UI_Line.Y2 = Pos.Y;
            X2 = Pos.X;
            Y2 = Pos.Y;
            Move_Label();
        }

        public void Change_X1_Y1(double X, double Y)
        {
            UI_Line.X1 = X;
            UI_Line.Y1 = Y;
        }
        public void Change_X2_Y2(double X, double Y)
        {
            UI_Line.X2 = X;
            UI_Line.Y2 = Y;
        }

        public void Move_Label()
        {
            double X = (X2 - X1)/2 - 5+X1;
            double Y = (Y2 - Y1)/2 - 23+Y1;
            Canvas.SetLeft(Line_Lable, X);
            Canvas.SetTop(Line_Lable, Y);
        }

        //change this so that the values are generic and then just have it so that the
X and Y coords are changed directly and don't need the method to do it.(A lot of work
:(
        public void Link_Input_Align_Line(Gate_Class Gate)
        {
            UI_Line.Stroke = Brushes.Black;
            double[] hold = _MainWind.Link_Input_Align(Gate, Input_Num);
            Change_X2_Y2(hold[0], hold[1]);
            X2 = hold[0];
            Y2 = hold[1];
            Move_Label();
        }

        public void Link_Output_Align_Line(Gate_Class Gate)
        {
            double[] hold = _MainWind.Link_Output_Align(Gate, Output_Num);
            Change_X1_Y1(hold[0], hold[1]);
            X1 = hold[0];
            Y1 = hold[1];
            Move_Label();
        }

        public void Remove_UI()
        {
            _Sub_Canvas.Children.Remove(UI_Line);
            _Sub_Canvas.Children.Remove(Line_Lable);
        }

        public void Remove_Class()
        {
```

```
            Remove_UI();
            _Line_List.Remove(this);
        }

        public void Change_UI_Black()
        {
            UI_Line.Stroke = Brushes.Black;
            Line_Lable.Foreground = Brushes.Black;
        }

        public void Change_UI_Red()
        {
            UI_Line.Stroke = Brushes.Red;
            Line_Lable.Foreground = Brushes.Red;
        }
    }
```

Testing

Test No	Type of data	Description	Expected	Material Reference	Passed/ Failed	Notes
1	Erroneous	Simulator speed input box. Test its string input with multiple char values and see if it returns an exception error. Due to how the boxed is coded it will change to accept an invalid char but will reject it and change back to the old input.	Only integer value in the box.	1A	Failed then passed	The boxed is programmed in a way so that if an invalid char is inputted it will go back to the previous valid input. Failed due to the value could be negative. Changed from int to Uint
2	Boundary	Test the Border for adding a new gate to the canvas. I'm going to drag and drop a rectangle in the middle of the canvas then drag and drop another off the canvas. Then going to move the canvas over to where the second rectangle would be added.	When the rectangle is added off screen it should added to the canvas.	2A	Passed	
3	Erroneous	Testing the border of the canvas to make sure it is aligned correctly.	Will remove if the cursor is not on the canvas.	Tested as close as I could to each corner going around to the 4	Passed	Each quadrant of the corner was top left, top right, bottom left, bottom right.

				quadrants and see if it will do the expected result. Passed the test.		
4	Boundary	Testing overlap of gates. Will add 1 gate to the canvas. Will create a new one and see if it gets deleted when there is an overlap.	If any corners overlap it will get deleted.	3A	Passed	Did it for each corner(bottom right with top left, top right with bottom left, top left with bottom right, bottom left with top right)
5	Black Box Testing	Gate Calculation	Each gate will have the correct output for their gate type	4A	Failed then passed	In the xor and Xnor gate I tried using a different type of syntax to make the code more efficient but it didn't work. Just syntax error.
6A	Error Checking	Lines with input and output. Testing to see if that they are correctly added and that line is added to port 2 instead of port 1.	The line output port should be 1 and input port is 1. When adding the input and output	5A	passed	For the transformer with 3 output they overlap but not to the level where it's restricted.

			buttons they should not overlap the line and for the different types of gate should change their set up.			
6B	Error Checking	The lines should be able to be removed without effecting the gates.	Just line is removed and the input buttons are in tack.	5B	Passed	
6C	Error Checking	When all inputs are full(lines or input buttons)	You shouldn't be able to add another line.	The line is removed.	Passed	
6D	Error Checking	When all outputs are full	The line shouldn't even be created in the first place	The line doesn't even get added to the canvas	Passed	
6E	Error Checking	When only parts of the inputs are full	If the port is full it should be	5C	passed	

			added to the next available port.			
6F	Error Checking	When the gates are moved	All Inputs outputs and lines should track the movement of the gate when in movement and when placed down.	5D Tracks gate when moving.	passed	The label for the line (which is 0) should also move directly in the middle of the line which it does.
7	Boundary	When a gate is already placed and is dragged over another rectangle.	It should return to the old position of the rectangle and so should it's input and output.	6A	Passed	
8	Error Checking	System clean up	Any Gates, Lines, input and output classes should be	7A	Failed then passed	Found a bug in the code when testing a partially hard system clean up where I was removing the object

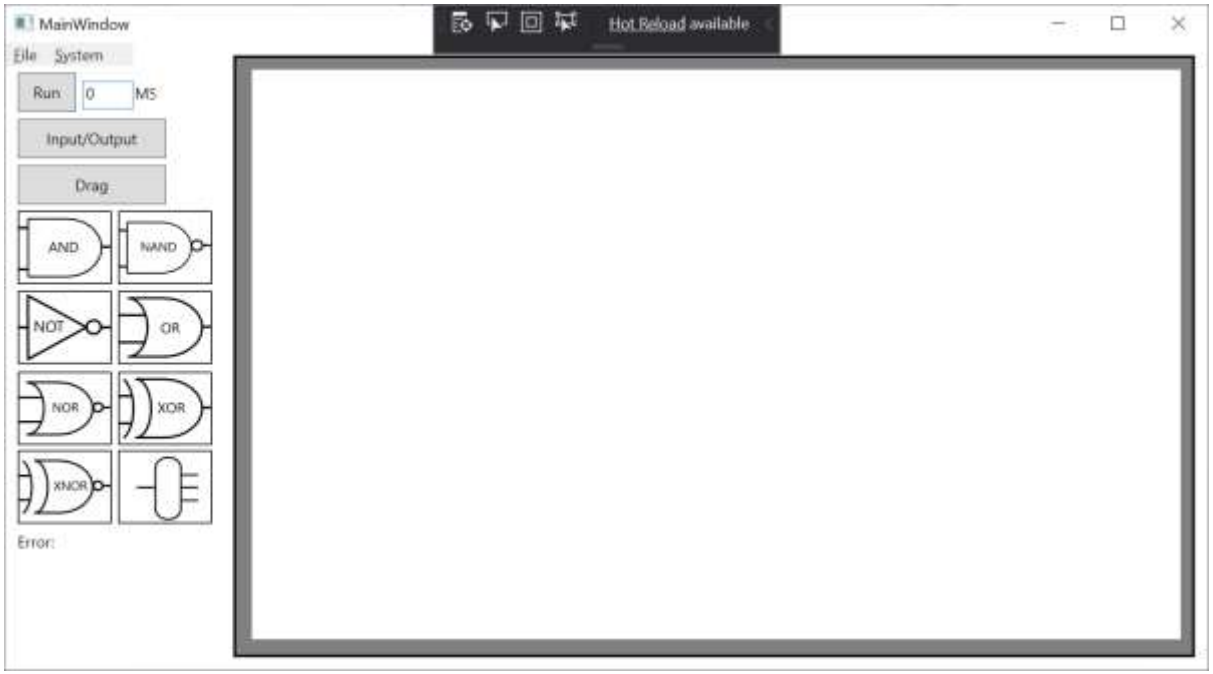
			removed if they're not being used anymore			before decreasing all the other variables by 1. This worked for all values in the list which weren't directly above it. So it's not immediately noticeable but if you keep making changes everything becomes out of line and will end in an out of range error.
9A	Black Box Test	Saving a file under save as.	File should be created	8A	Passed	
9B	Black Box Test	Load a Save file.	Layout of a flip flop circuit should be opened and loaded	8B	Passed	The input and output buttons do not alien correctly when added to the canvas. This can be fixed just by moving the gate slightly.
9C	Black Box Test	New file	Give you a message saying it's going to be deleted then remove all variables.	8C	Passed	

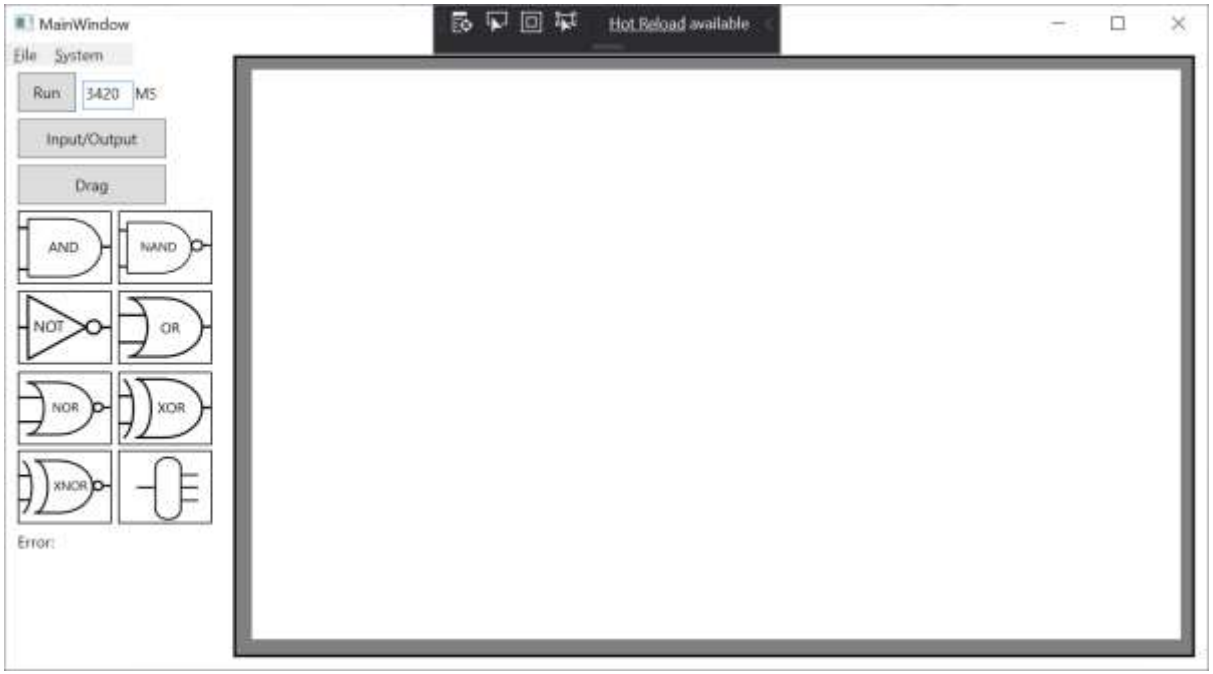
9D	Black Box Test	Save File that's already been saved	Updates the file	8D	Passed	Date modified updated and change in file size due to new objects.
9E	Black Box Test	Save File that hasn't been saved already	Should open Save AS method	8E	Passed	
9F	Black Box Test	Exit program button	The programme should completely end.	N/A	Passed	
10A	Black Box Test	Running the simulator when the time delay is greater than 0.	The First 2 gates should go red and then the And until it reaches the end which it then stops	9A	Passed	It has stopped at the end because the "Stop" Button has turned into a "Run" button again.
10B	Black Box Test	Running the simulator when the time delay is 0.	The rectangles should go red and all the gates should change at once.	9B	Passed	

10C	Black Box Test	Running the simulator when there is a loop.	The simulator should never reach an end.	9C	Passed	When it's 0 seconds the output will change as soon as the computer has worked it out so It will flash between 1 and 0 if it's an alternating circuit.
10D	Black Box Test	Multiple different Starting points for the simulator.	Each input button should be a starting point.	9D	Passed	
10E	Black Box Test	Remove Gate While simulator is running	The simulator should reach the remove gate and end then.	9E	Passed	
10F	Black Box Test	Remove Line while simulator is running	The Simulator should stop when it reaches the gate	9F	Passed	
10G	Black Box Test	Stop the simulator by pressing the button again	When it reaches the end of it's cycle the simulator	9G	Passed	The difference between screen shot 1 and 2 is that the run button has turned from stop to run. This is because it's

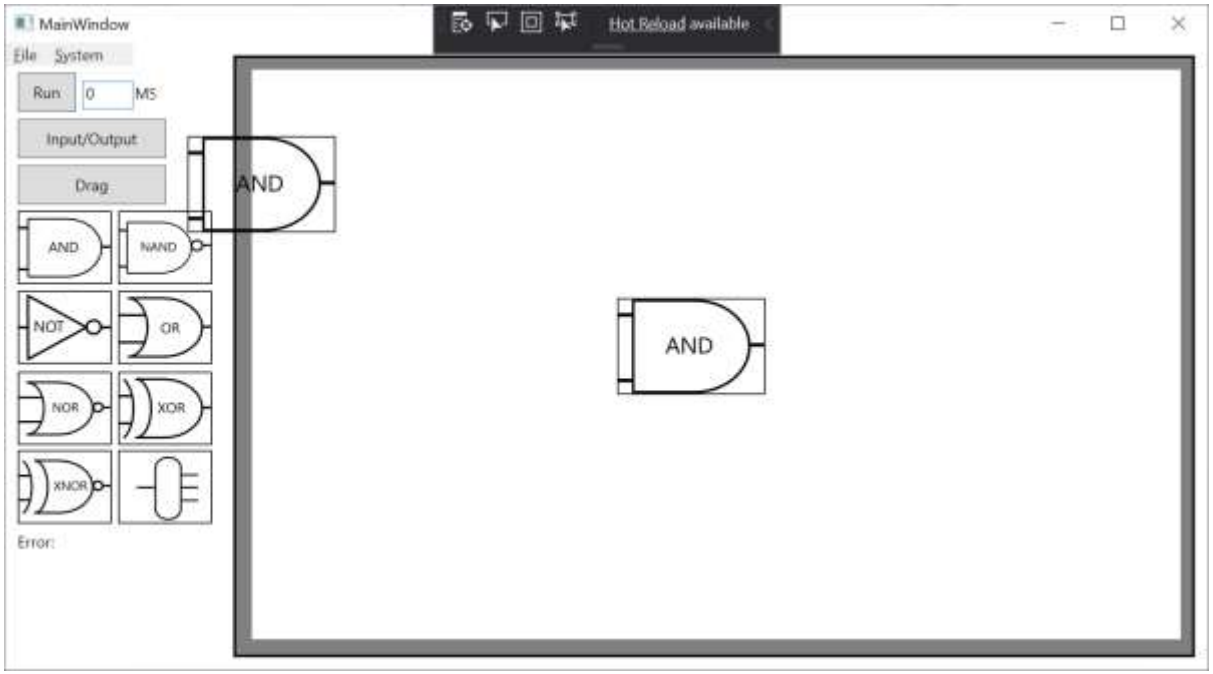
			should not move onto the next gate			what happens when you press it when it's running.
10H	Black Box Test	Try running the simulator again while the old one is still running.	The run button should not change to stop again while the simulator is still running	9H	Passed	Screen shot 2 and 3 are similar but they are 2 different screen shot. Nothing changes like expecting and therefore it works.
10I	Black Box Test	If the inputs aren't complete then the simulator should add the rest	The Simulator should add the missing inputs and output	9I	Passed	

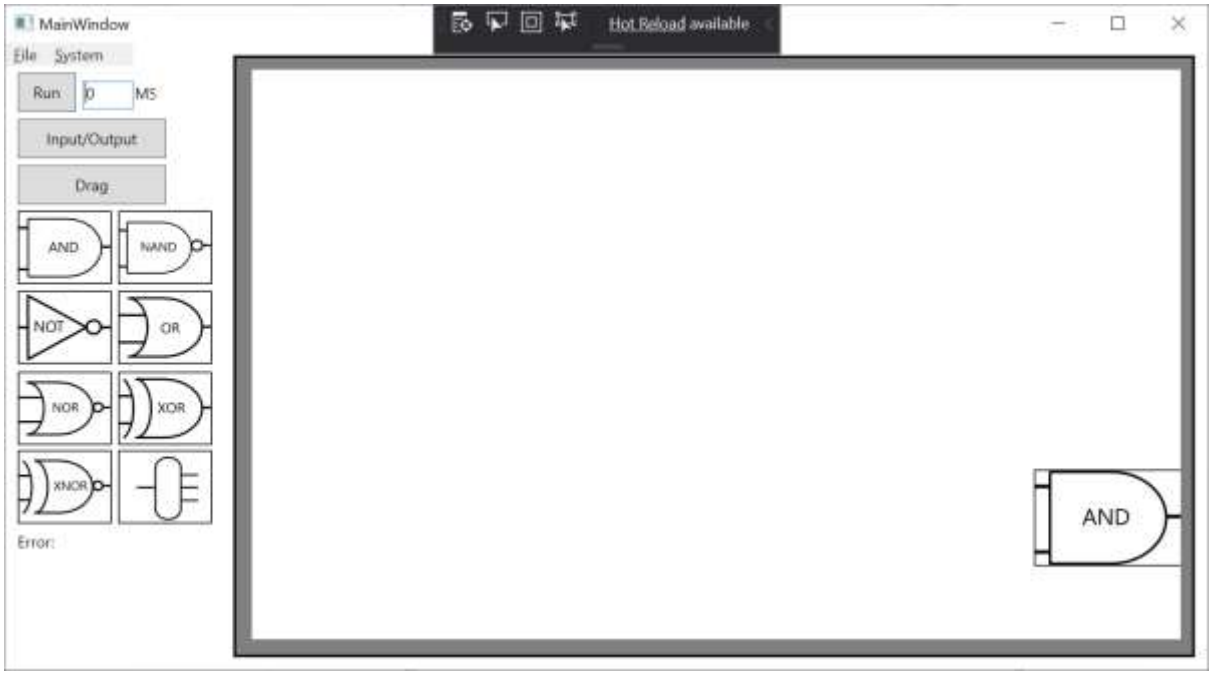
Reference

1A. 

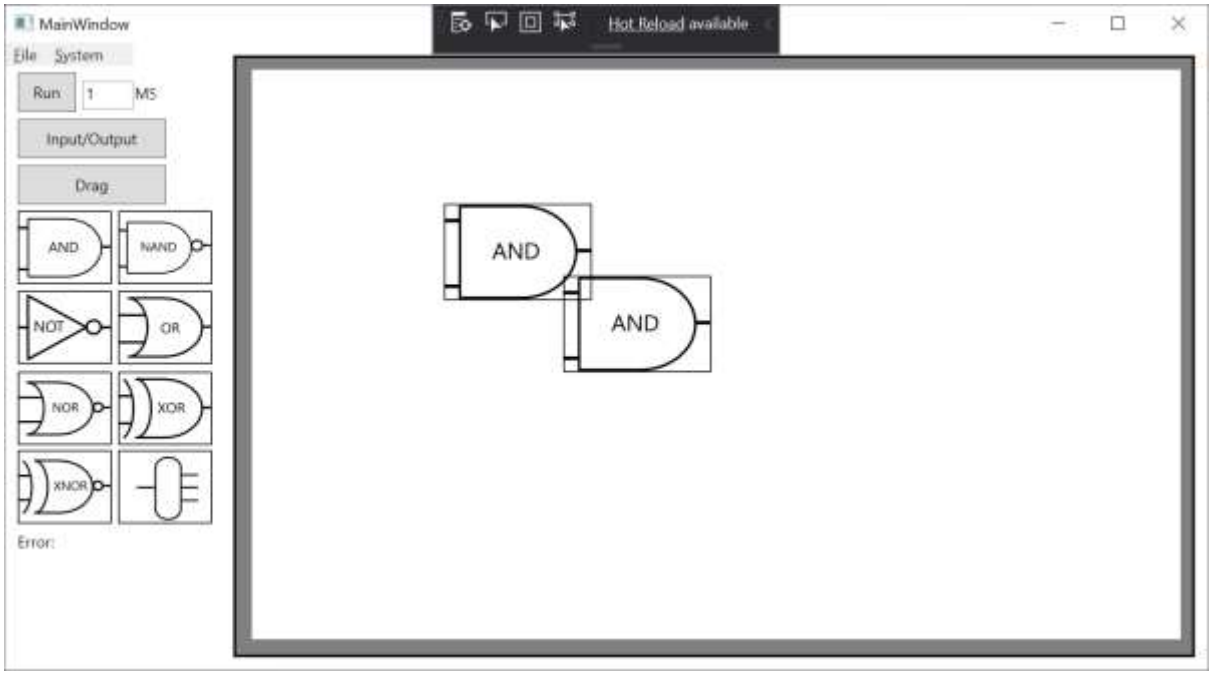

MainWindow
Hot Reload available
File System
Run
3420
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:

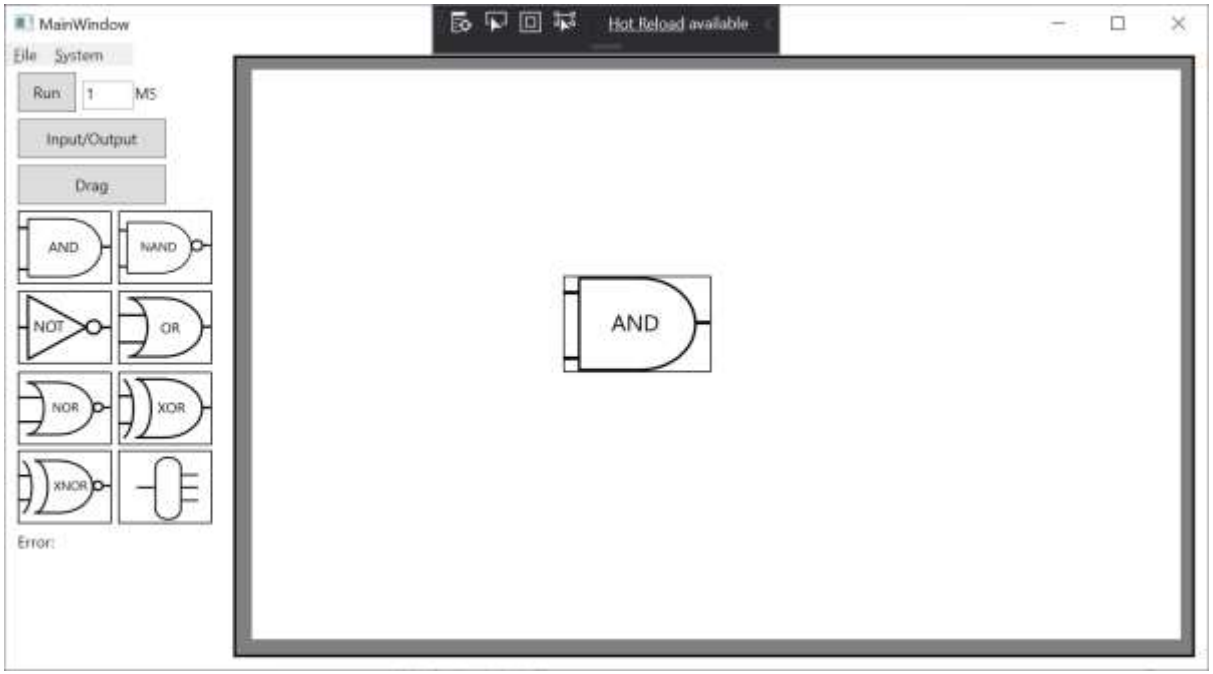
2A.


MainWindow
Hot Reload available
File System
Run
0
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND

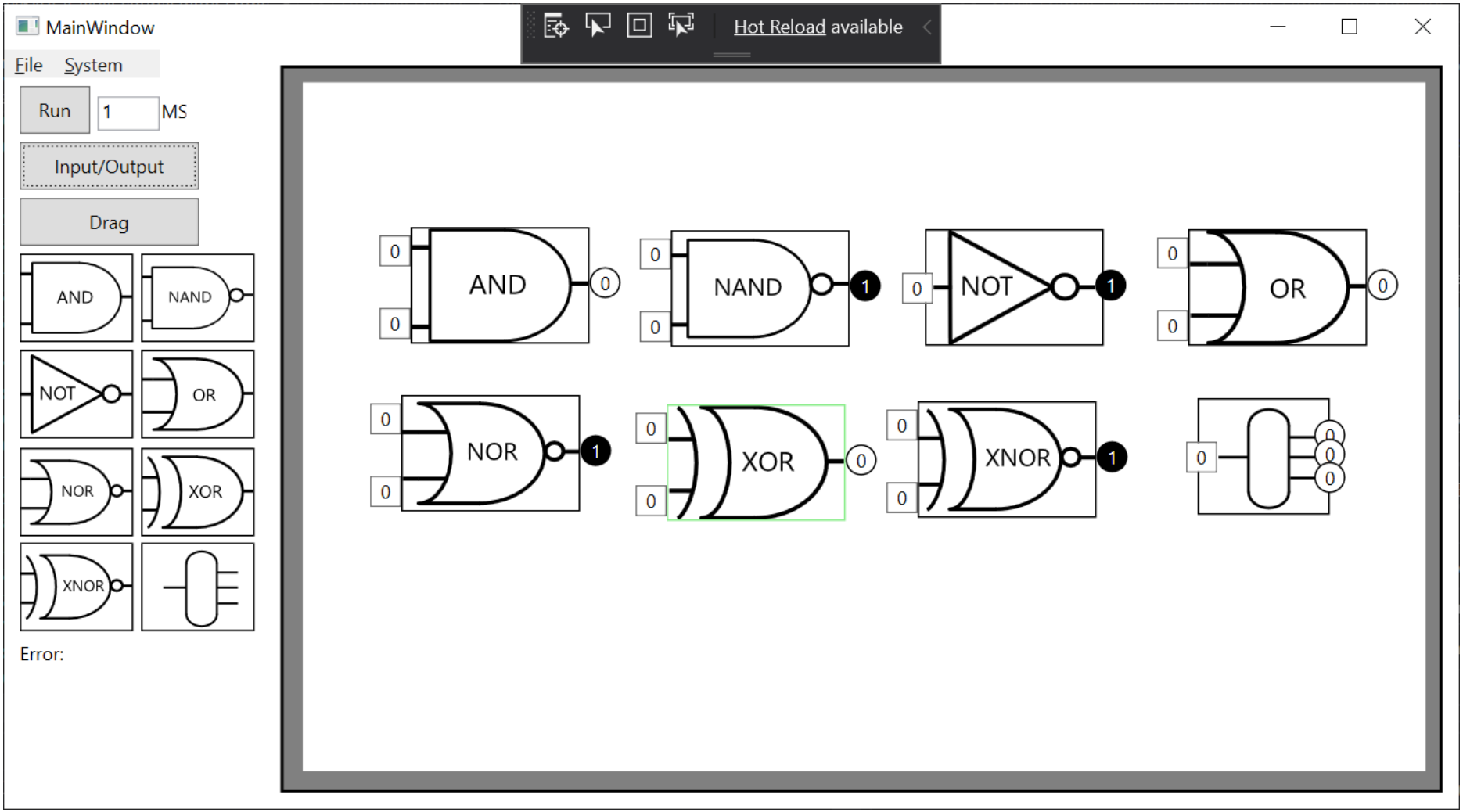

MainWindow
Hot Reload available
File System
Run
0
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND

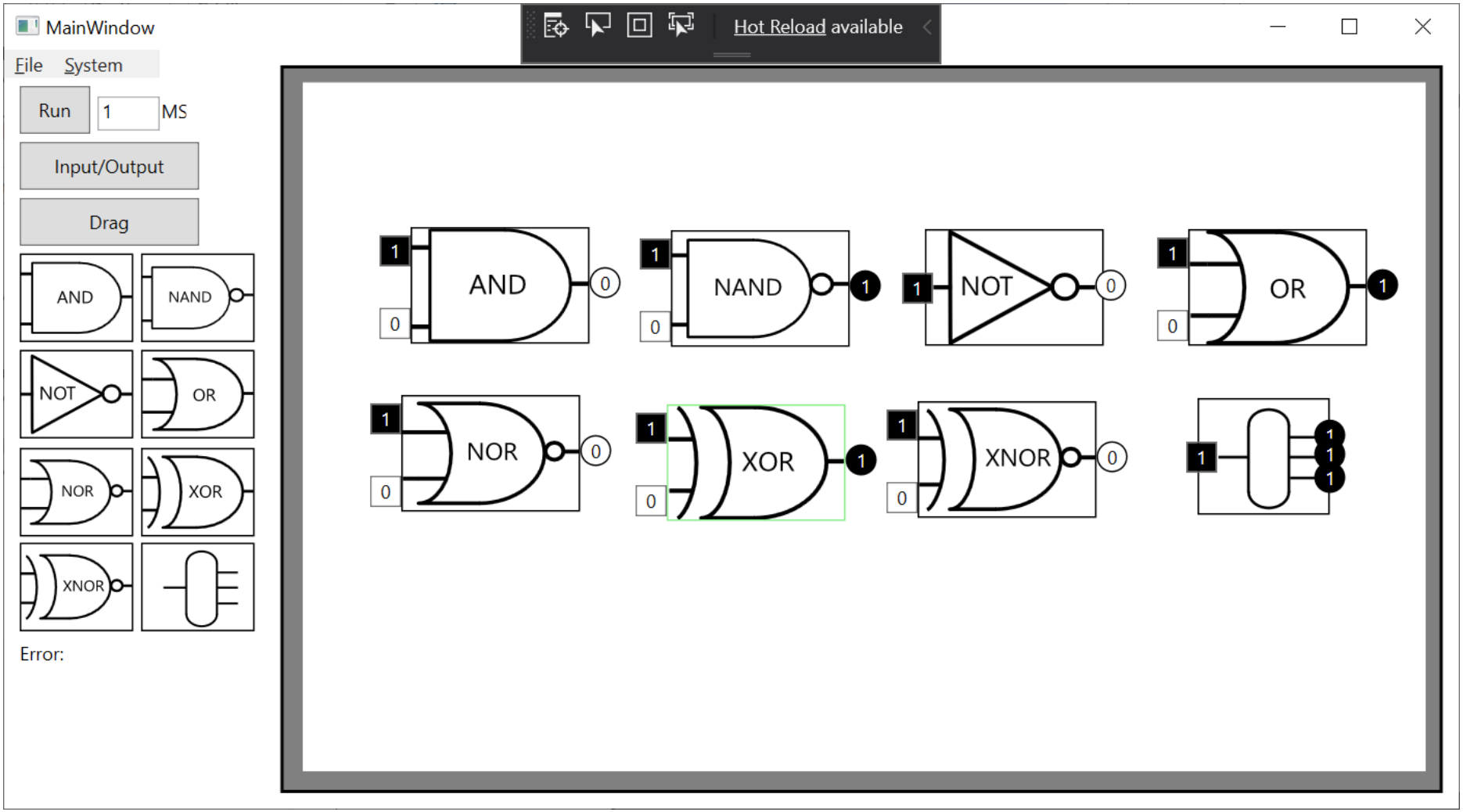
3A.

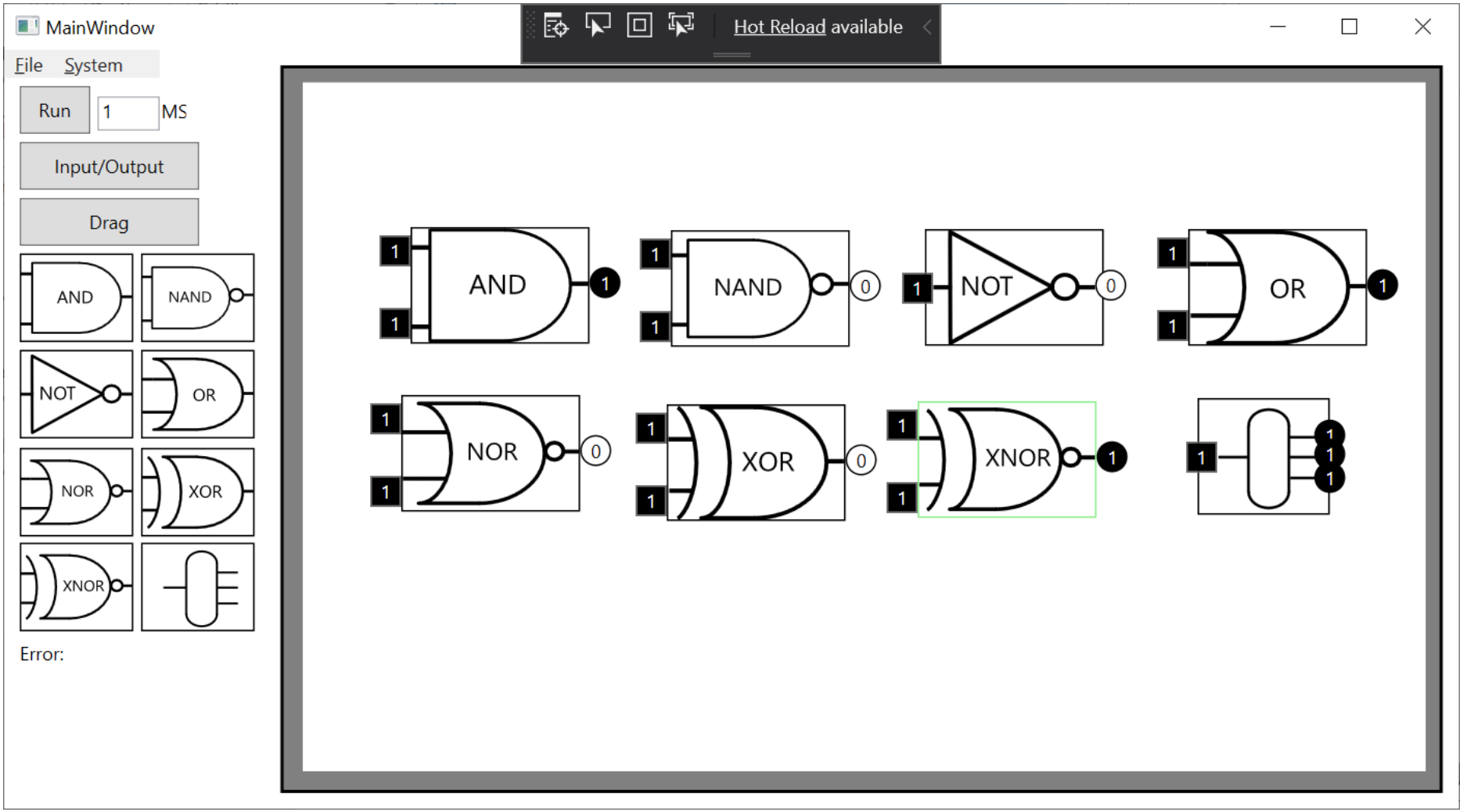

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
AND

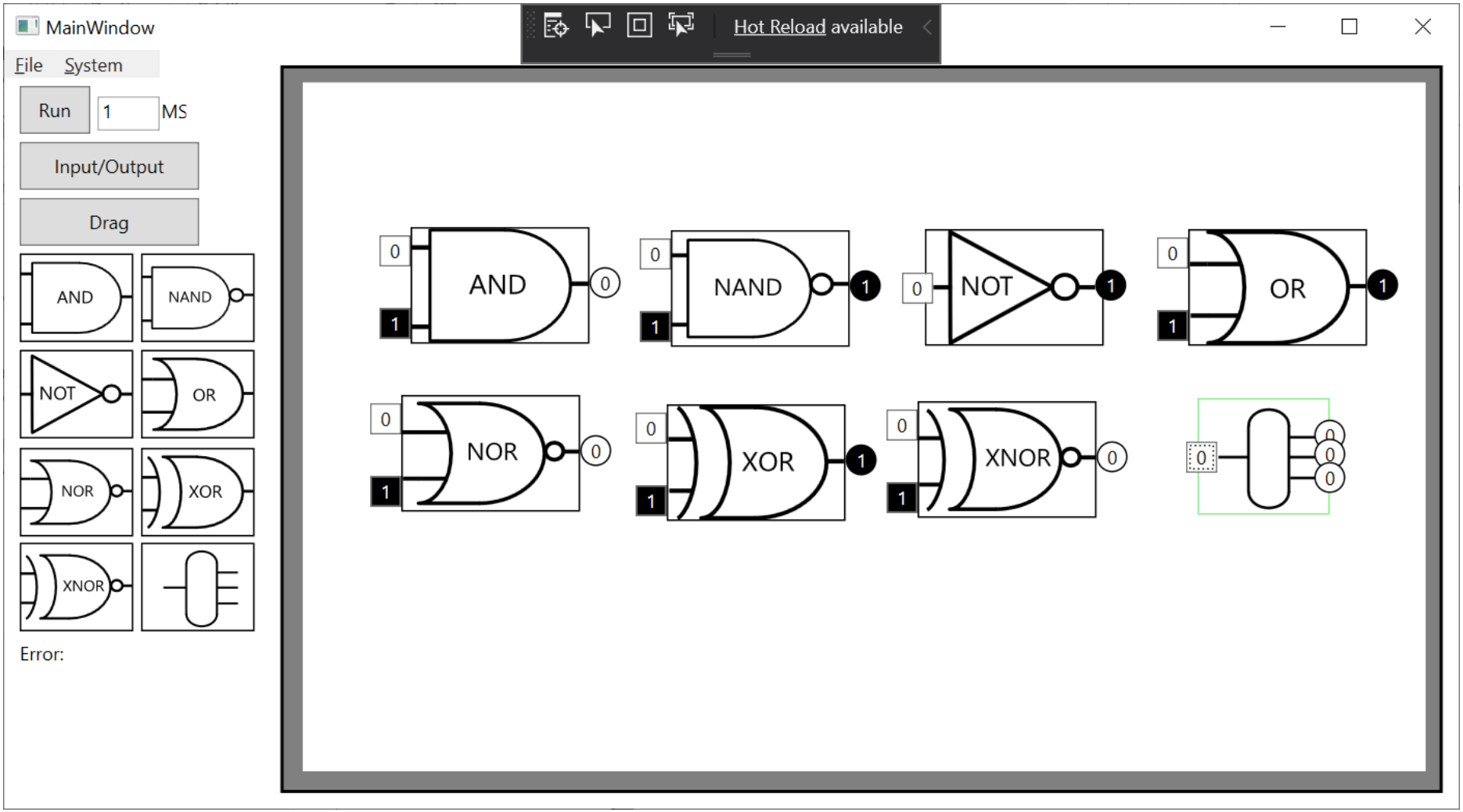

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND

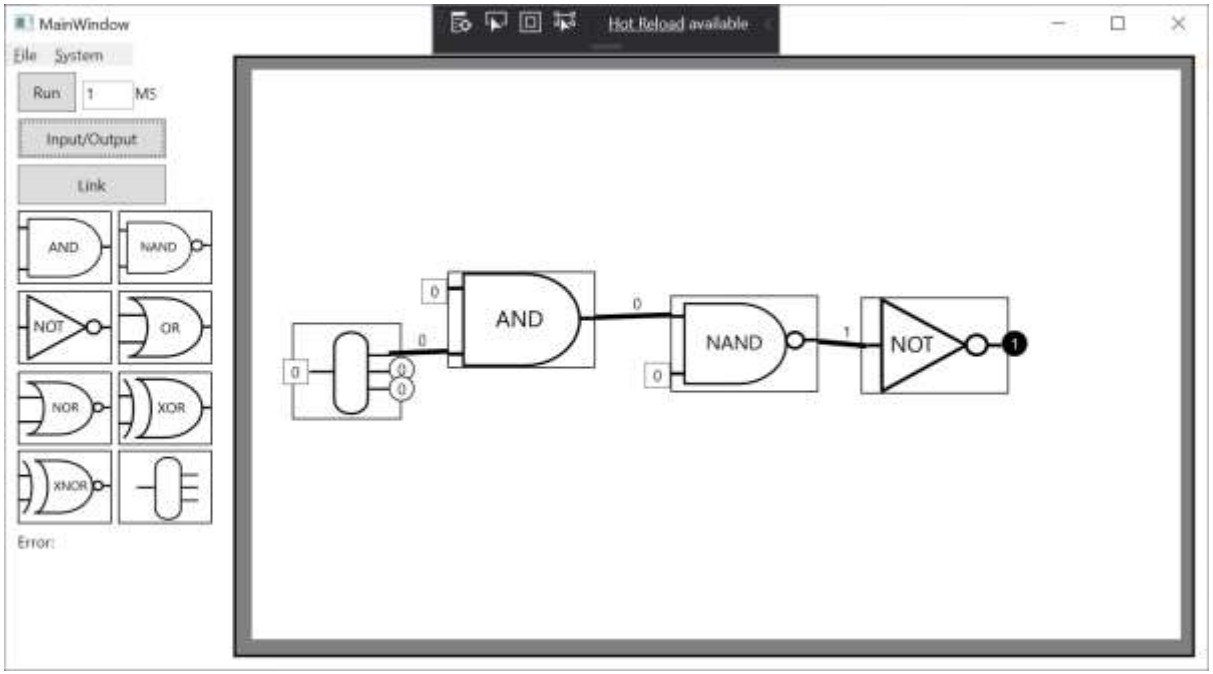
4A.


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
1

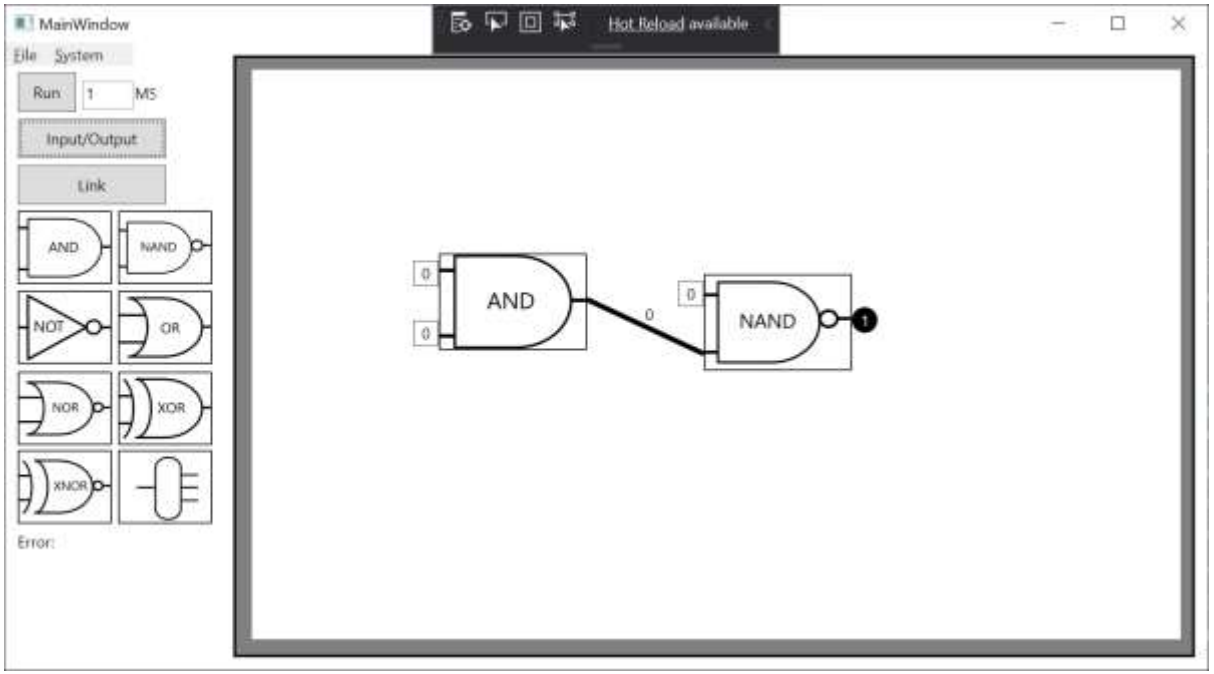

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
0
AND
NAND
NOT
OR
NOR
XOR
XNOR

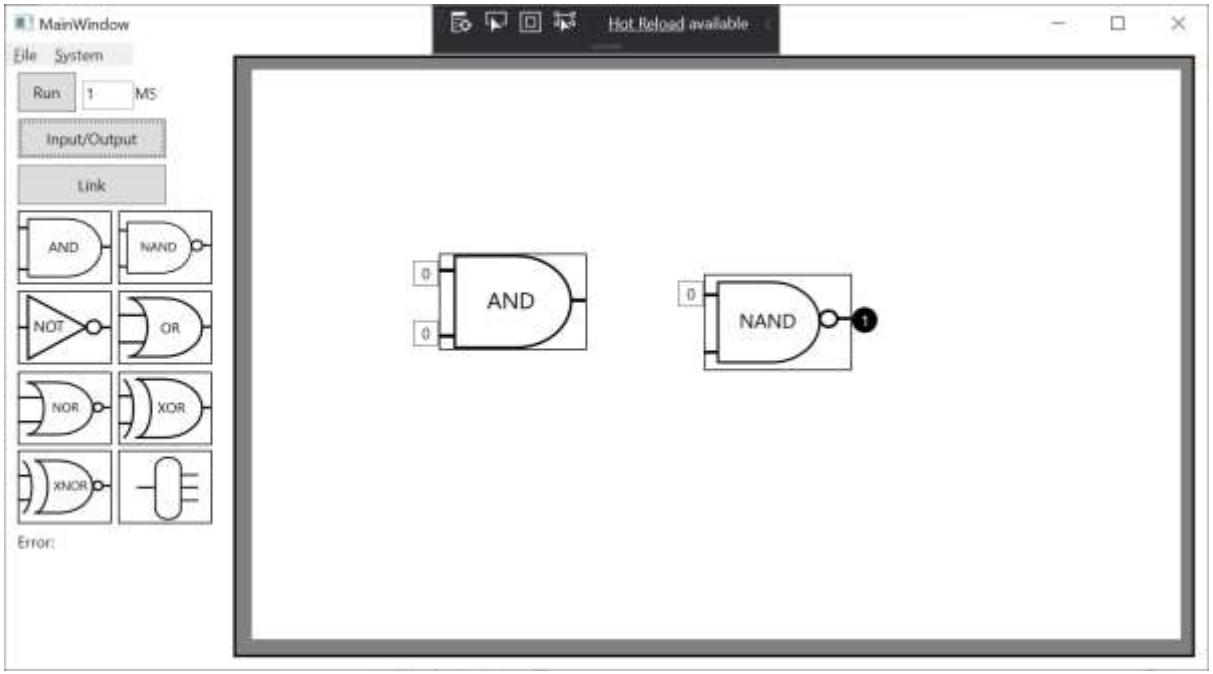

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
NAND
NOT
OR
NOR
XOR
XNOR


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
1

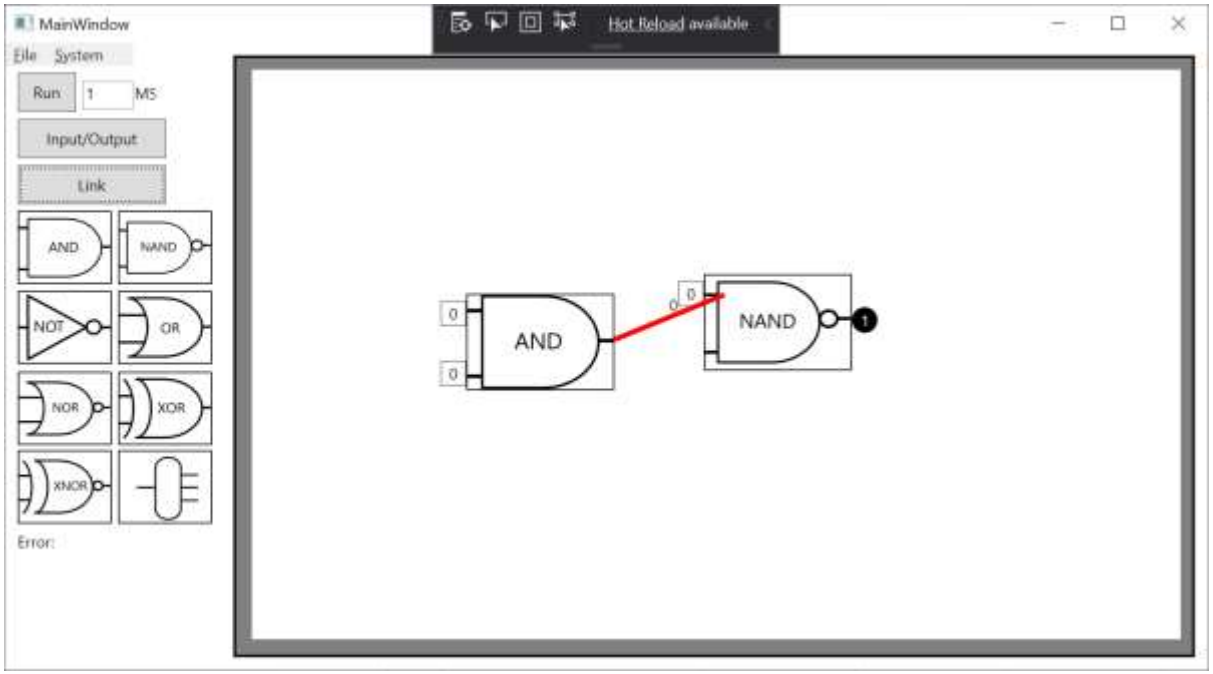
5A. 
MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
NAND
NOT

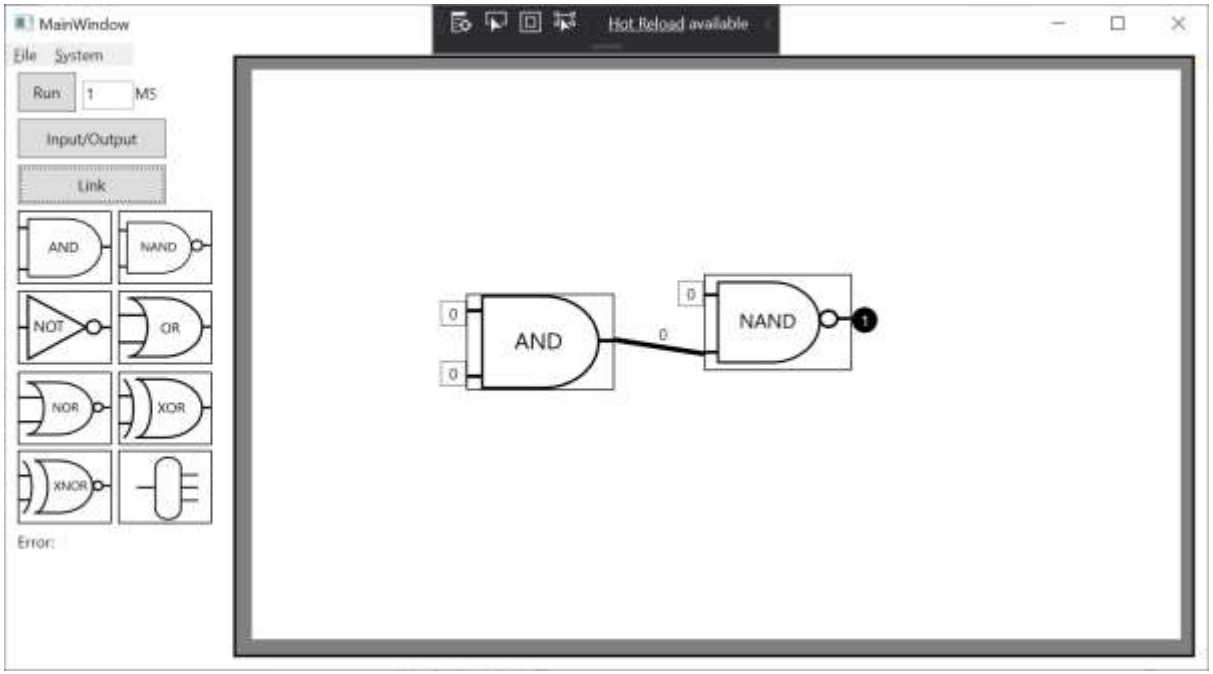
5B.


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
NAND
1

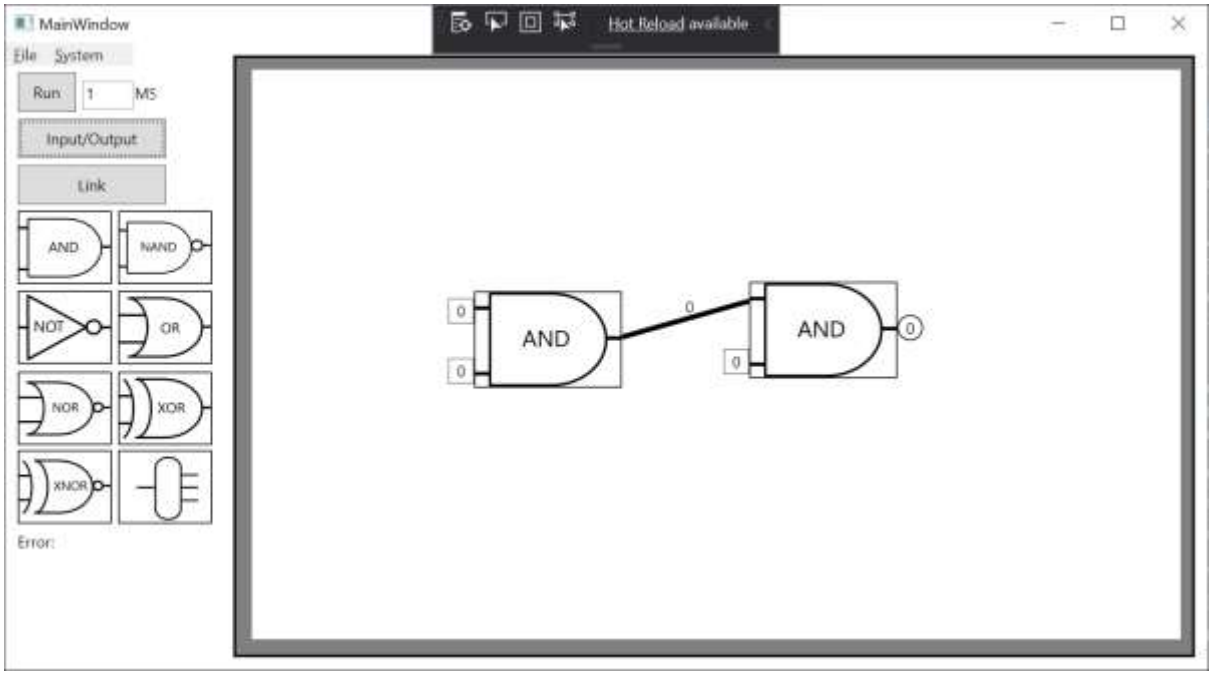

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
NAND
1

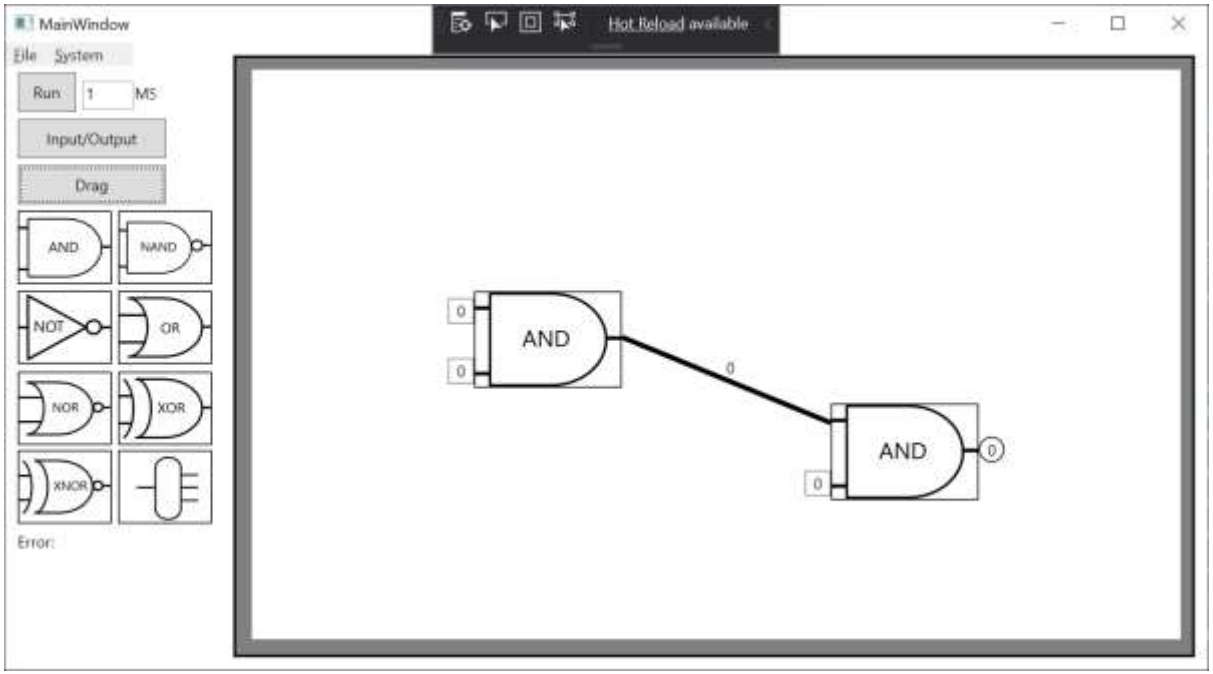
5C.


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
NAND
1

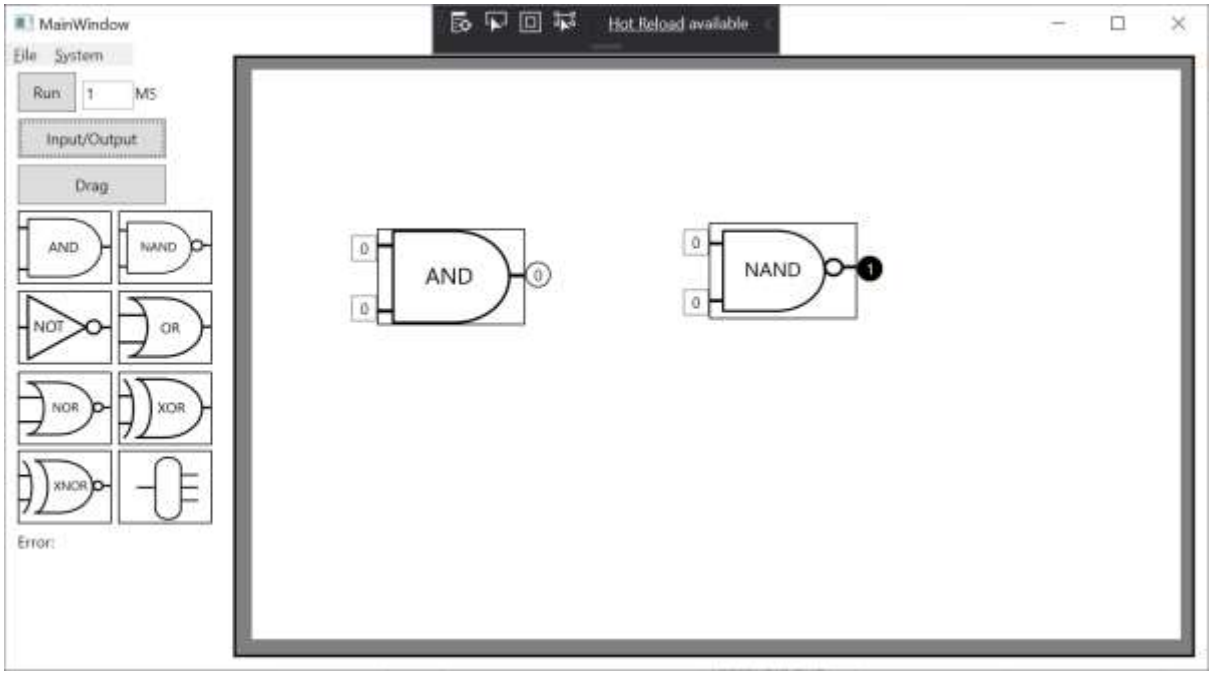

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
NAND
1

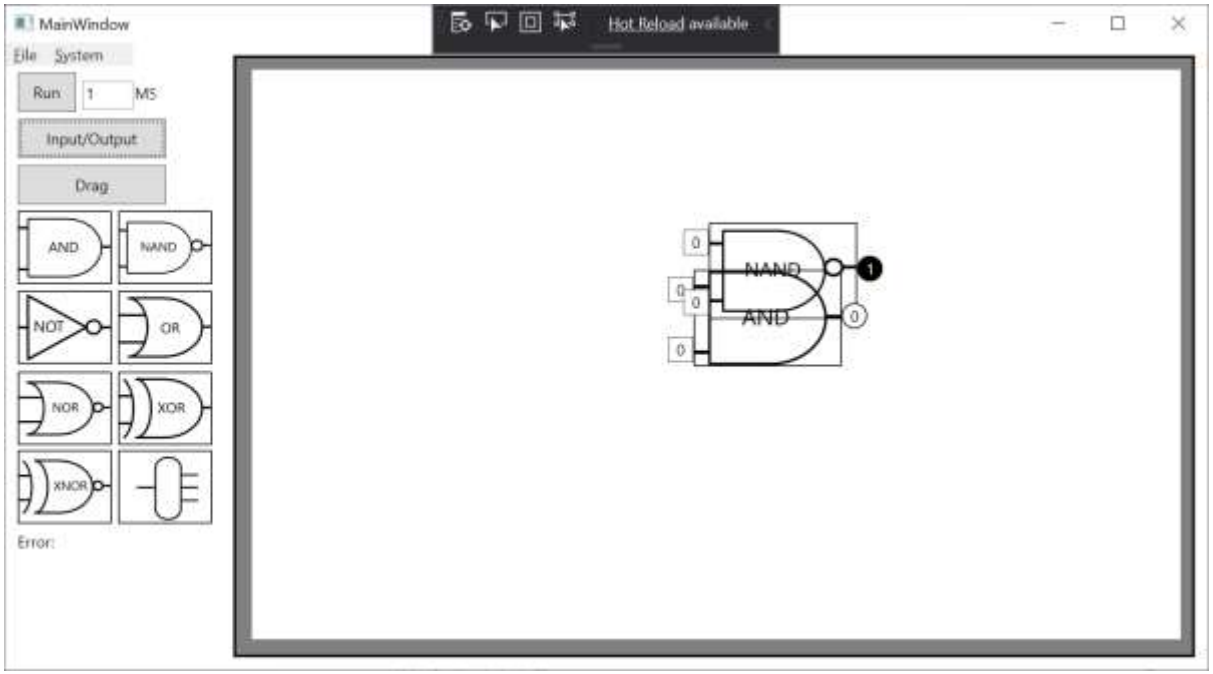
5D.

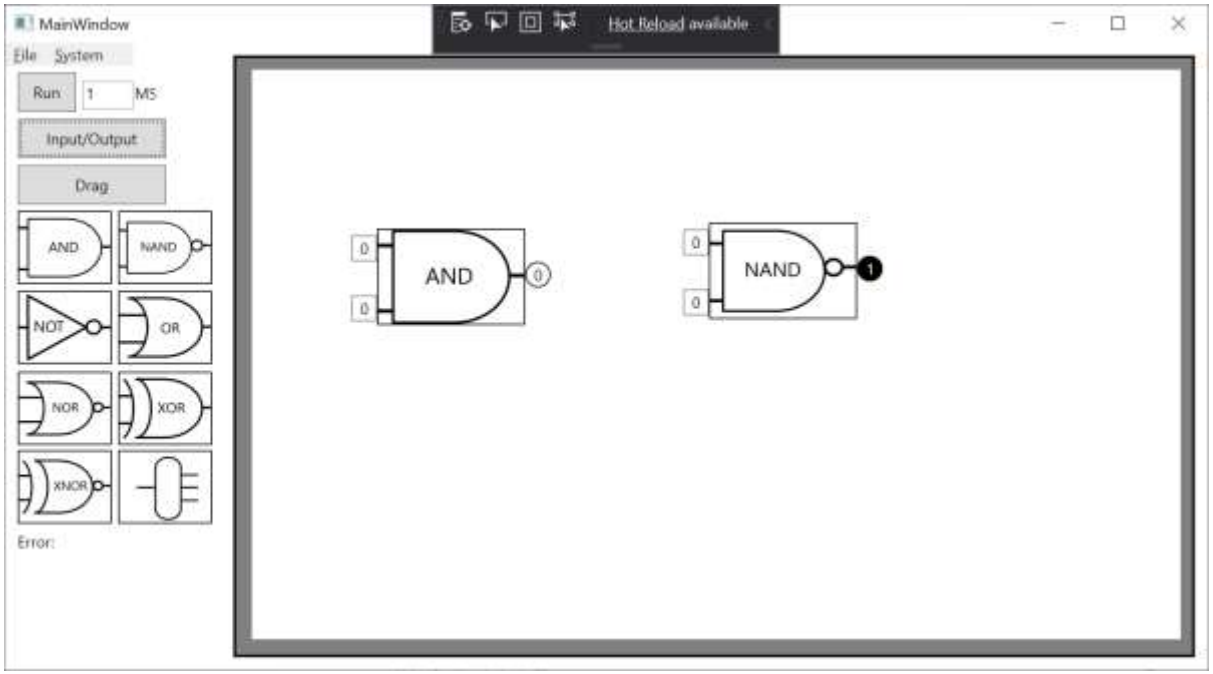

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
AND
0


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
AND
0
0

6A.

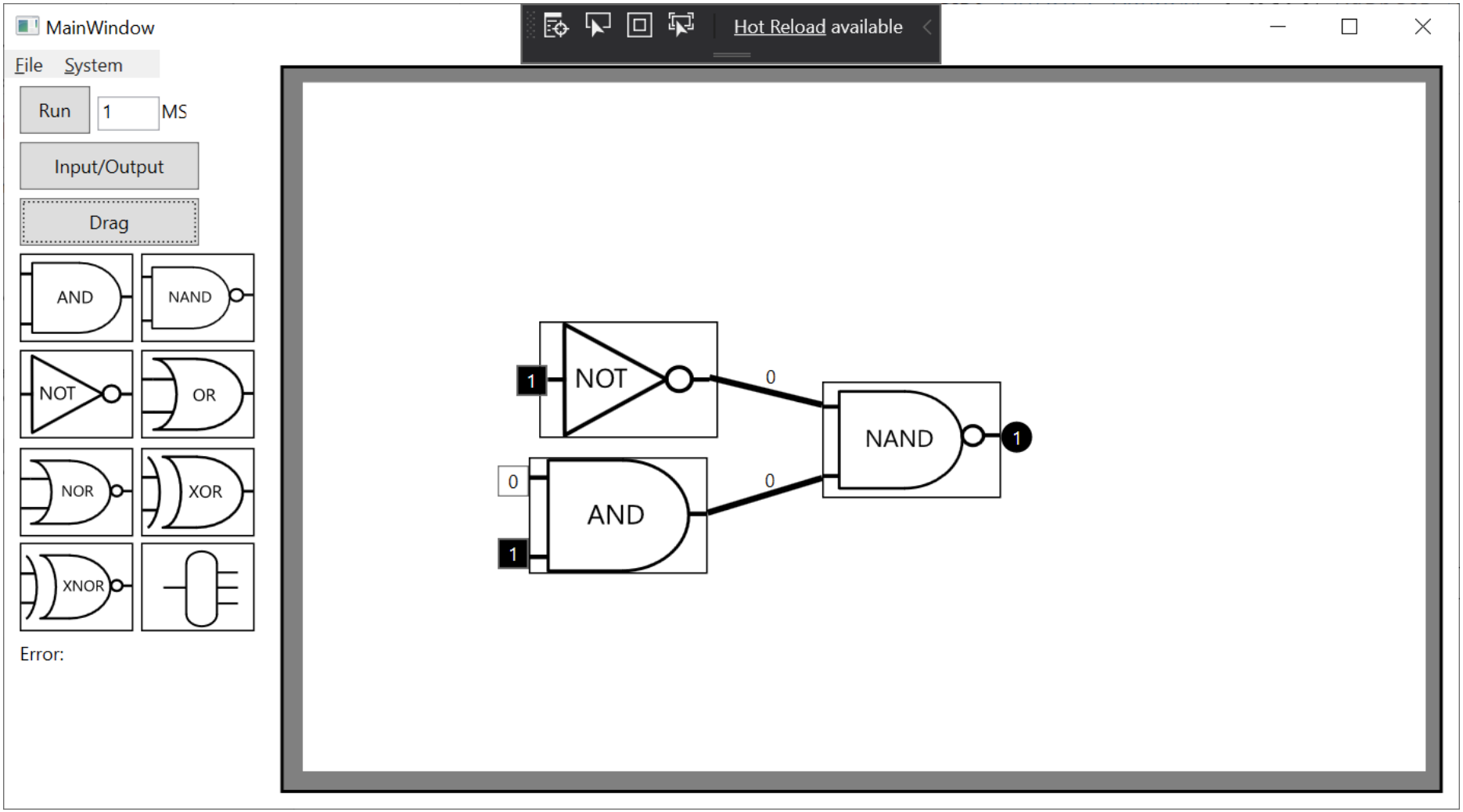

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
NAND
1

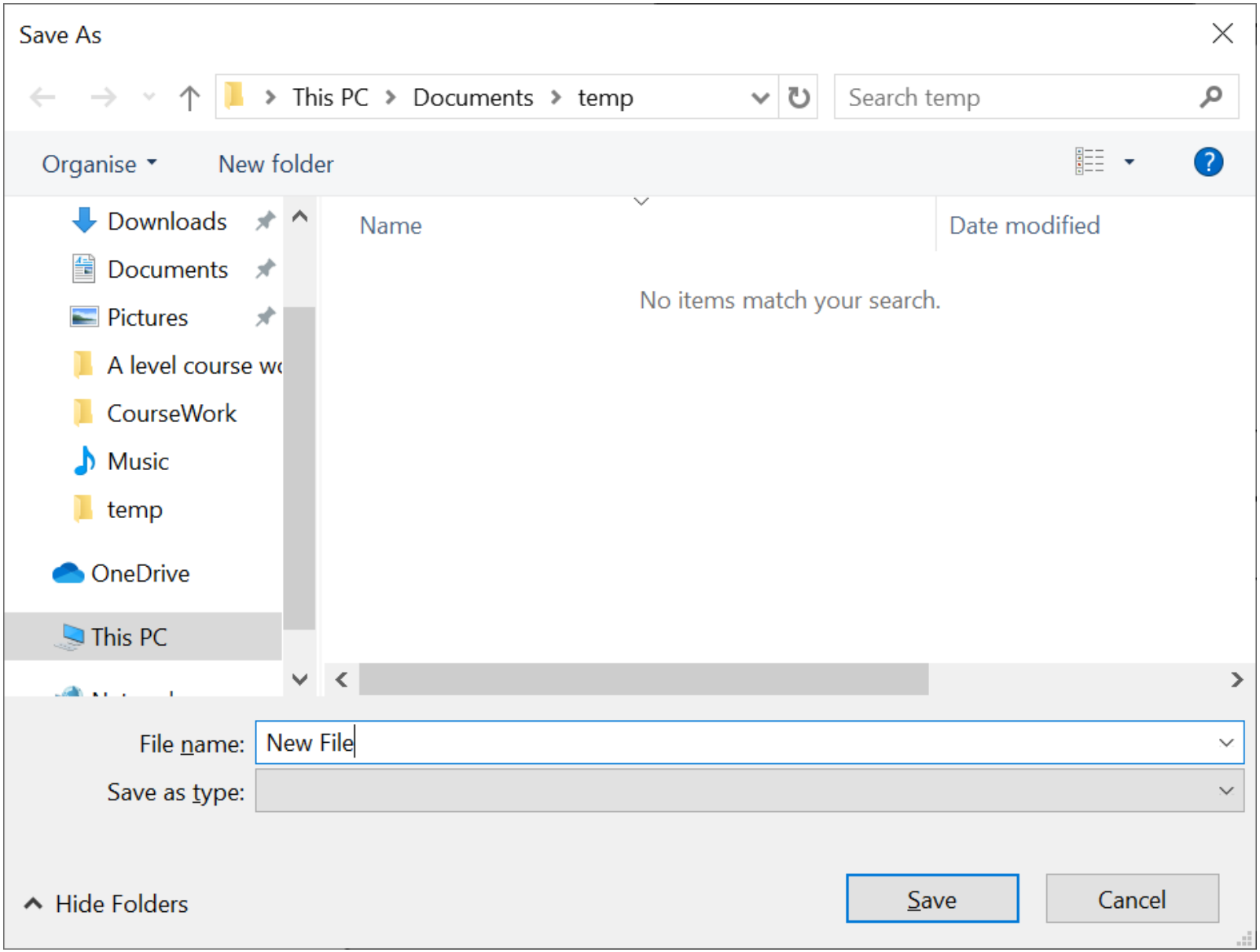

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
0
NAND
AND
1
0

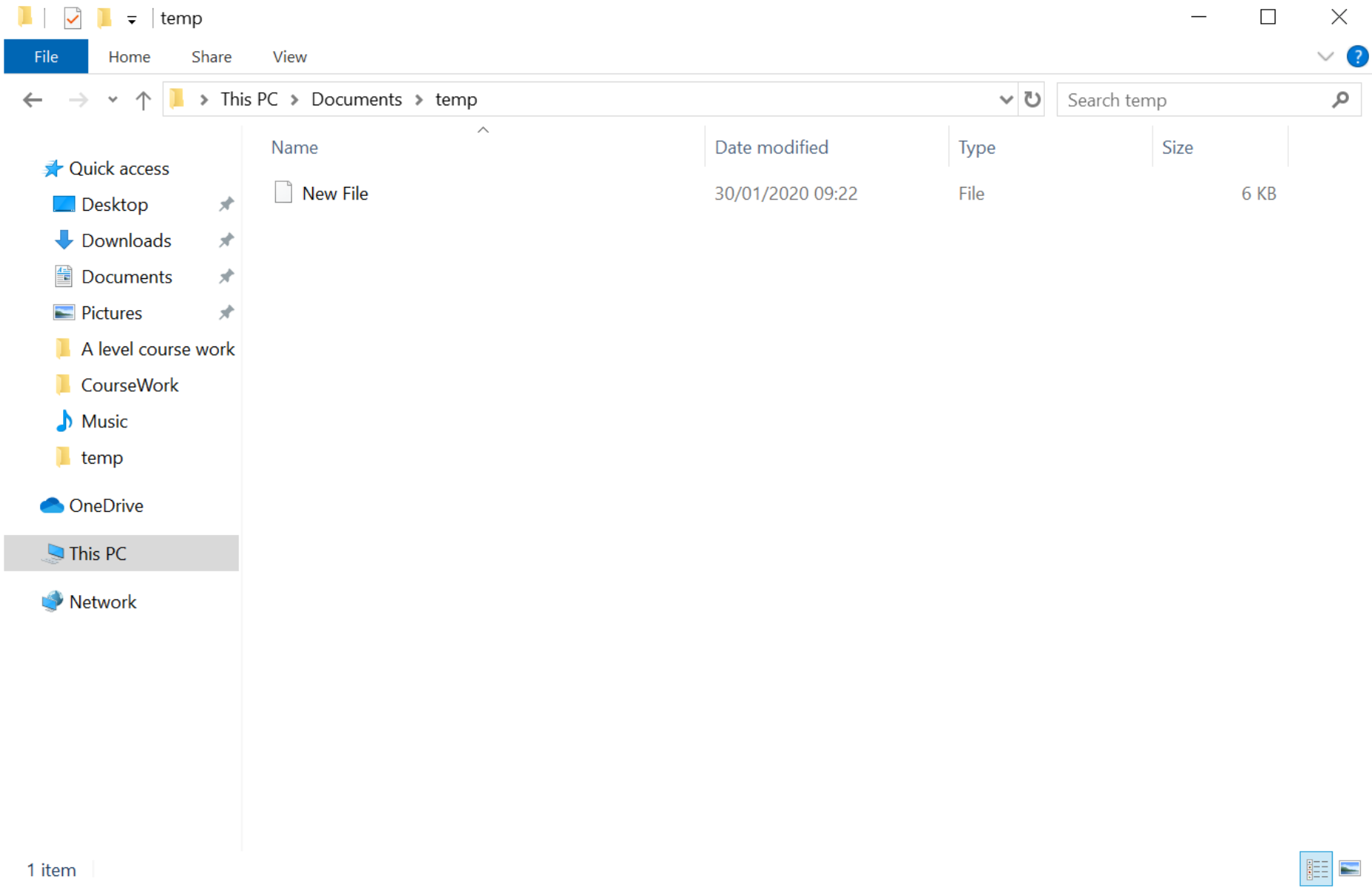

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
NAND
1

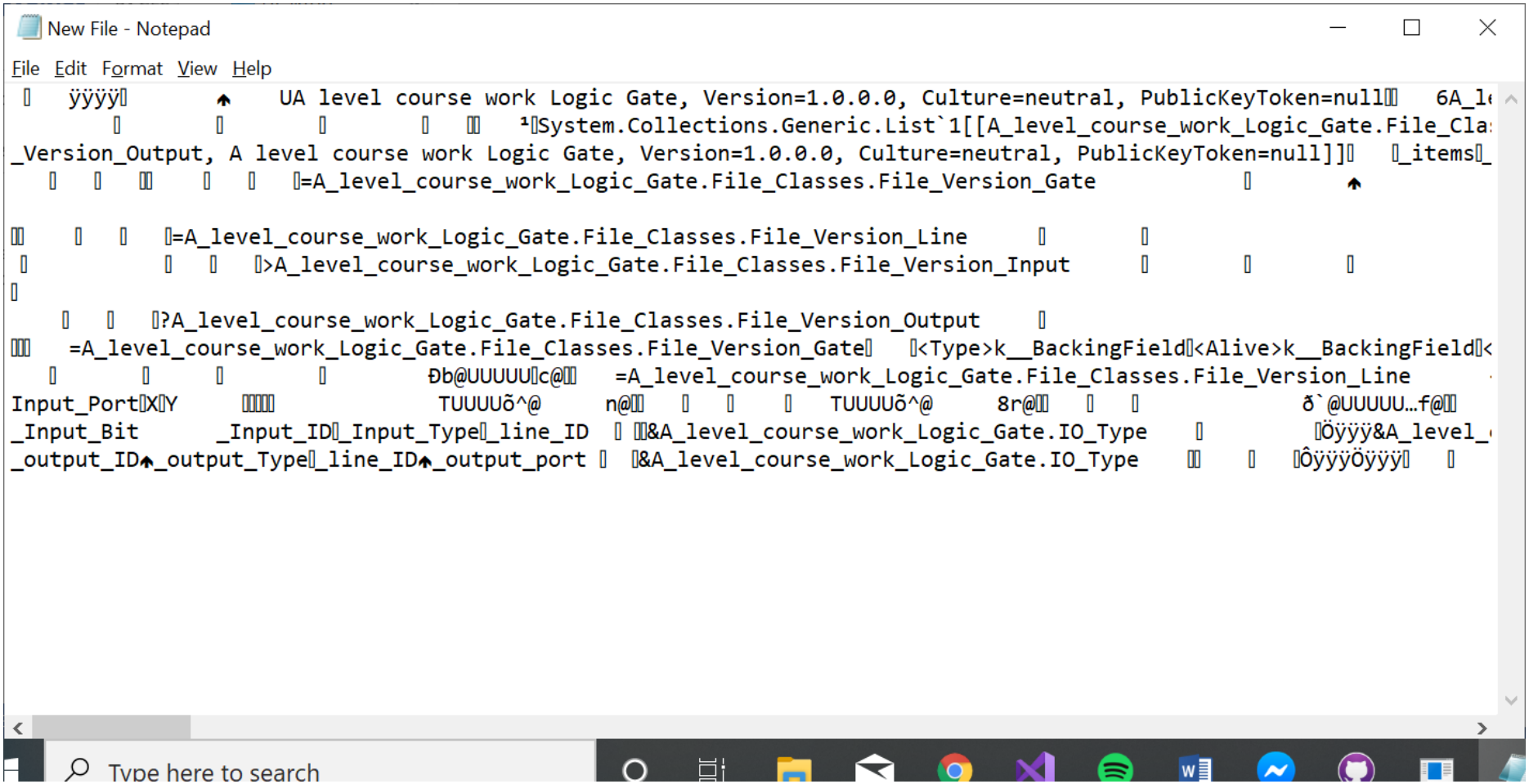
7A.

8A.

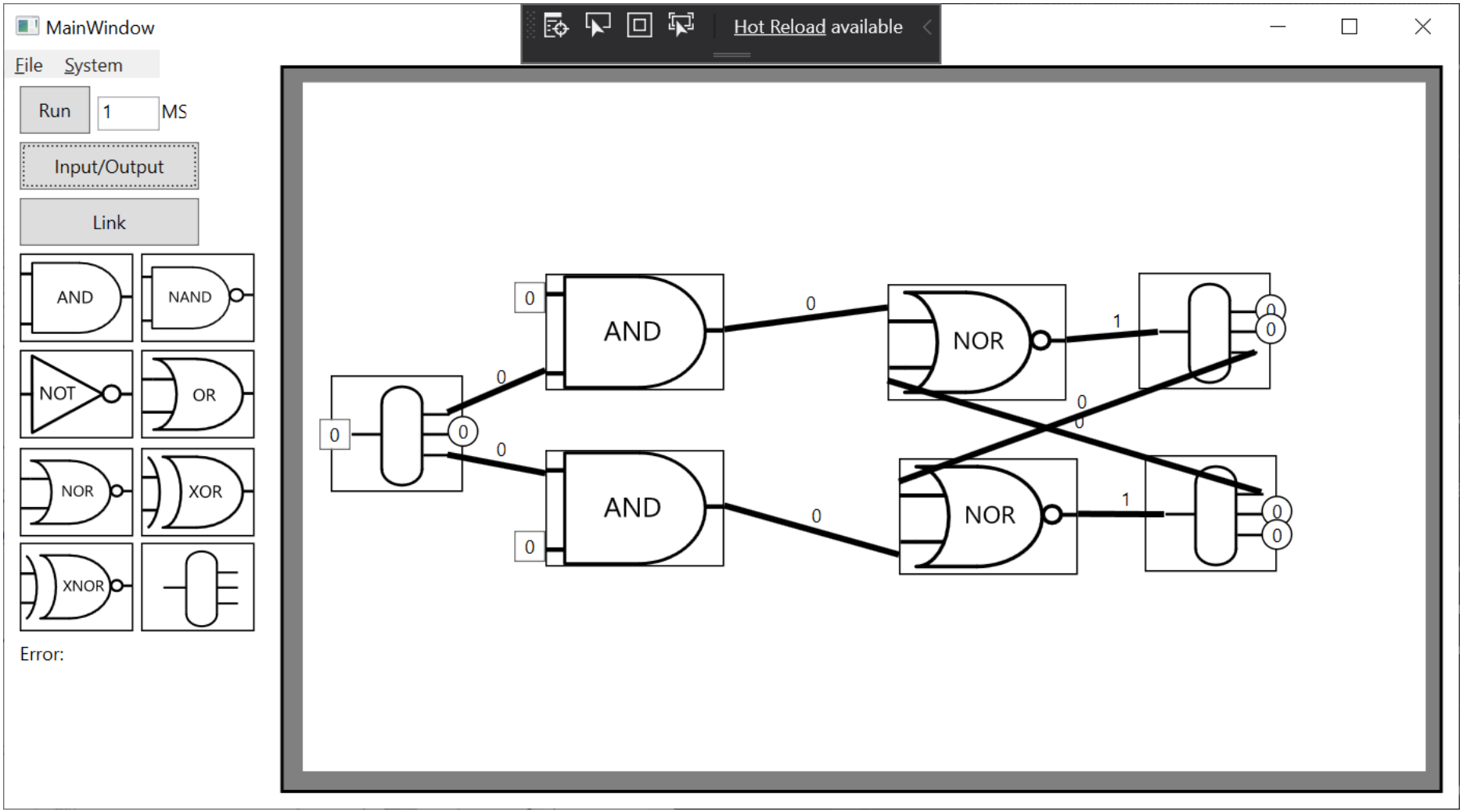

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
NOT
0
NAND
1
0
0
AND
1

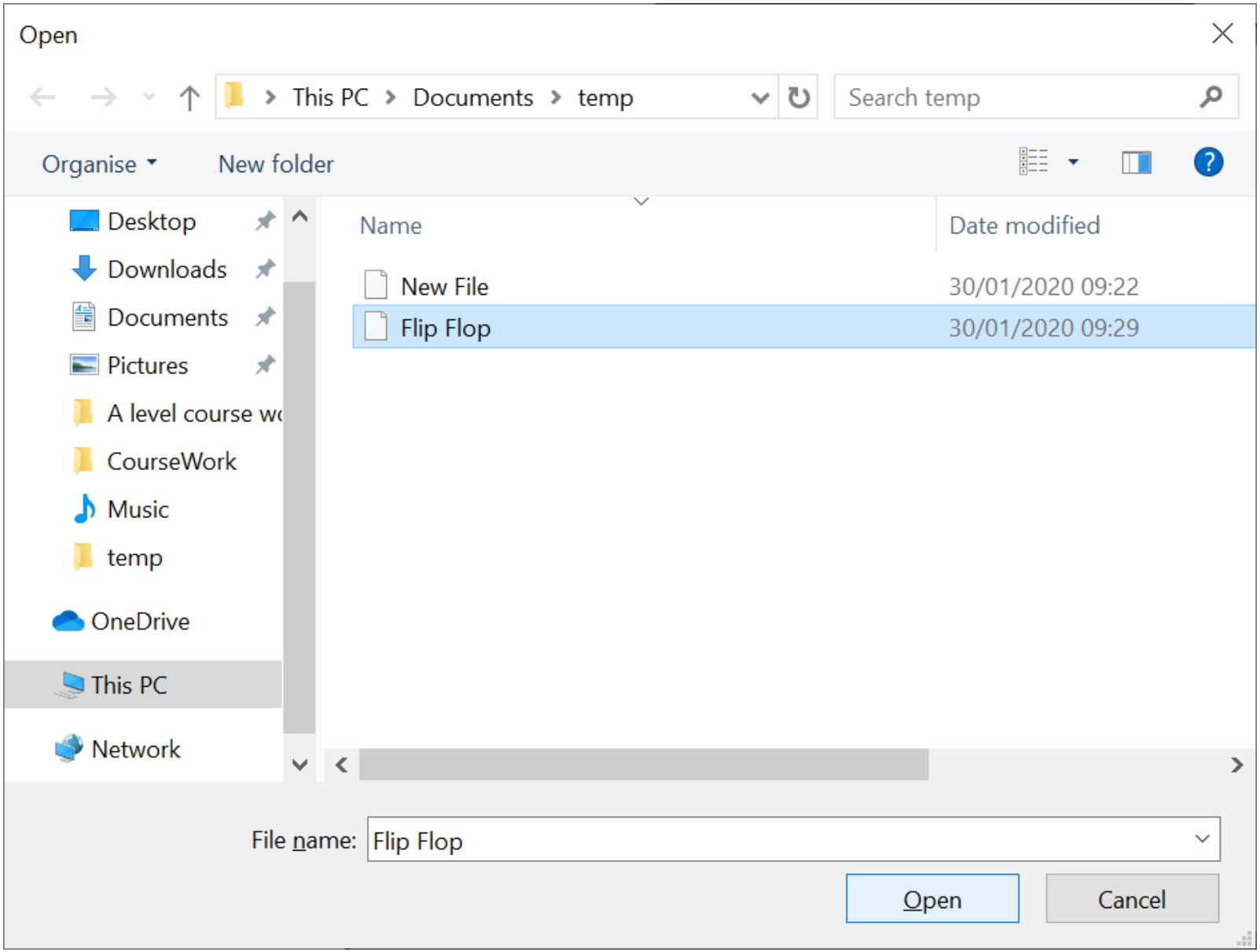

Save As
This PC
Documents
temp
Search temp
Organise
New folder
Downloads
Documents
Pictures
A level course wo
CourseWork
Music
temp
OneDrive
This PC
Name
Date modified
No items match your search.
File name:
New File
Save as type:
Hide Folders
Save
Cancel

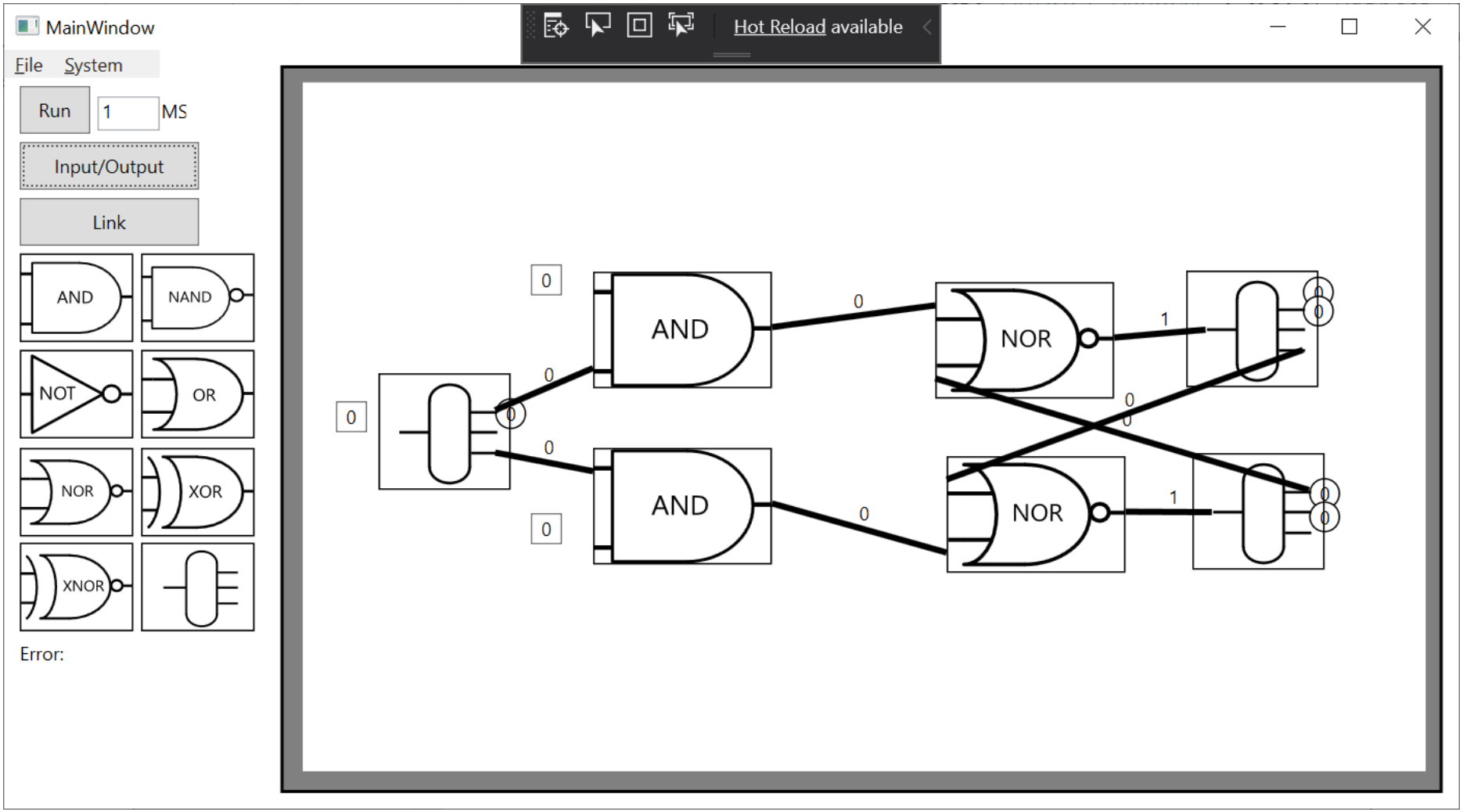

temp
File
Home
Share
View
This PC
Documents
temp
Search temp
Quick access
Desktop
Downloads
Documents
Pictures
A level course work
CourseWork
Music
temp
OneDrive
This PC
Network
Name
Date modified
Type
Size
New File
30/01/2020 09:22
File
6 KB
1 item


New File - Notepad
File Edit Format View Help
UA level course work Logic Gate, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
System.Collections.Generic.List`1[[A_level_course_work_Logic_Gate.File_Cla
_Version_Output, A level course work Logic Gate, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]]
_items
=A_level_course_work_Logic_Gate.File_Classes.File_Version_Gate
=A_level_course_work_Logic_Gate.File_Classes.File_Version_Line
>A_level_course_work_Logic_Gate.File_Classes.File_Version_Input
?A_level_course_work_Logic_Gate.File_Classes.File_Version_Output
=A_level_course_work_Logic_Gate.File_Classes.File_Version_Gate
<Type>k__BackingField
<Alive>k__BackingField
Db@UUUUUc@
=A_level_course_work_Logic_Gate.File_Classes.File_Version_Line
Input_Port
X
Y
TUUUUõ^@
n@
TUUUUõ^@
8r@
ð`@UUUUU…f@
_Input_Bit
_Input_ID
_Input_Type
_line_ID
&A_level_course_work_Logic_Gate.IO_Type
Öÿÿÿ&A_level_
_output_ID
_output_Type
_line_ID
_output_port
&A_level_course_work_Logic_Gate.IO_Type
ÔÿÿÿÖÿÿÿ
Type here to search

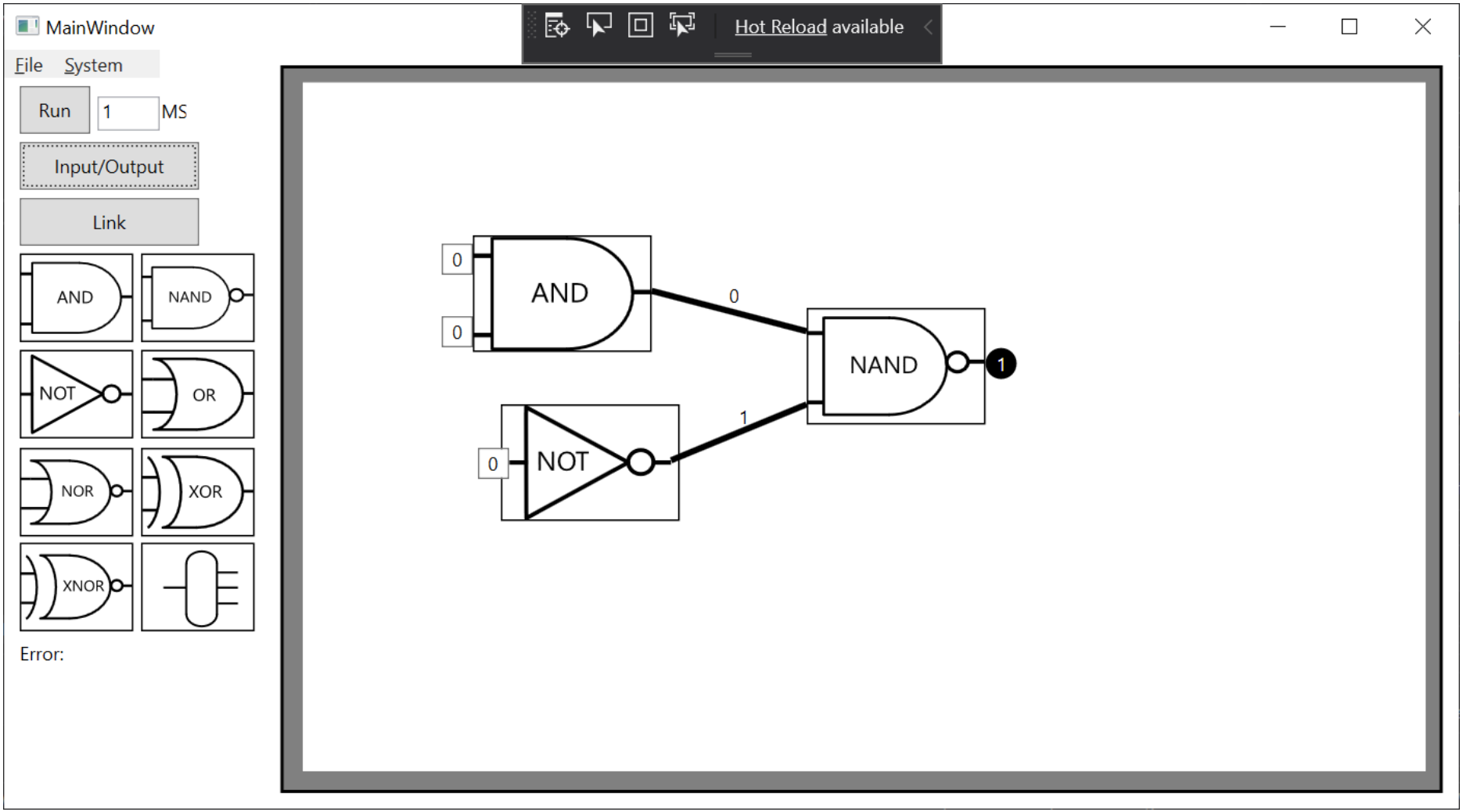
8B.


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
AND
NOR
1
AND
NOR
1


Open
This PC
Documents
temp
Search temp
Organise
New folder
Desktop
Downloads
Documents
Pictures
A level course wo
CourseWork
Music
temp
OneDrive
This PC
Network
Name
Date modified
New File
30/01/2020 09:22
Flip Flop
30/01/2020 09:29
File name:
Flip Flop
Open
Cancel


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
AND
AND
NOR
NOR
1
1

8C.

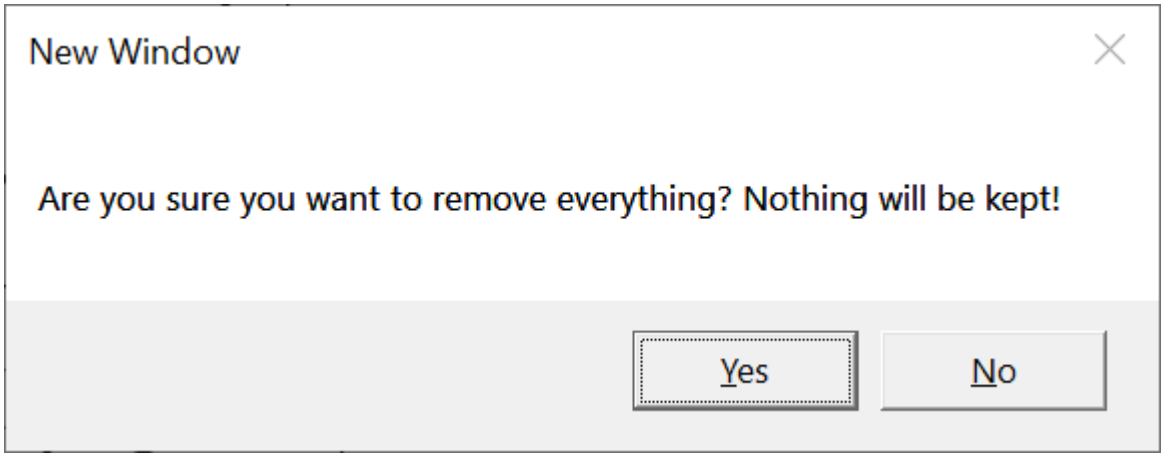

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
NAND
1
0
NOT
1

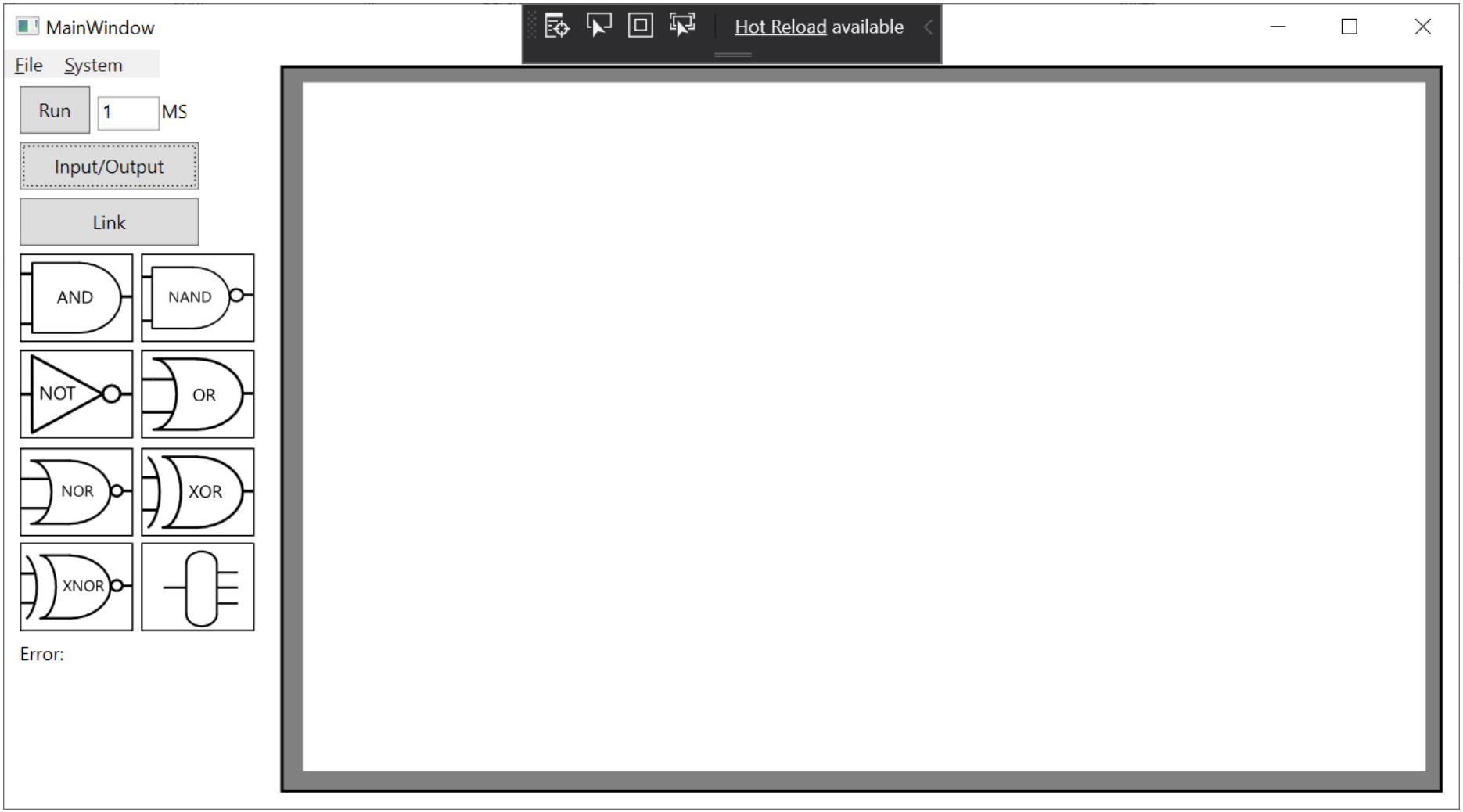

public List<Gate_Class> Gate_List { get; set; } = new List<Gate_Class>();
Gate_List
Count = 3


public List<Line_Class> Line_List { get; set; } = new List<Line_Class>();
//make these custom classes
Line_List Count = 2

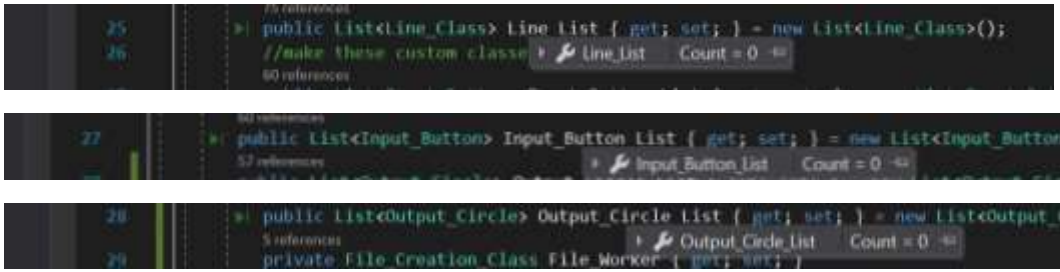

public List<Input_Button> Input_Button_List { get; set; } = new List<Input_Button>();
Input_Button_List Count = 3
public List<Output_Circle> Output_Circle_List { get; set; } = new List<Output_Circle>();


public List<Output_Circle> Output_Circle_List { get; set; } = new List<Output_Circle>();
Output_Circle_List Count = 1
private File_Creation_Class File_Worker { get; set; }

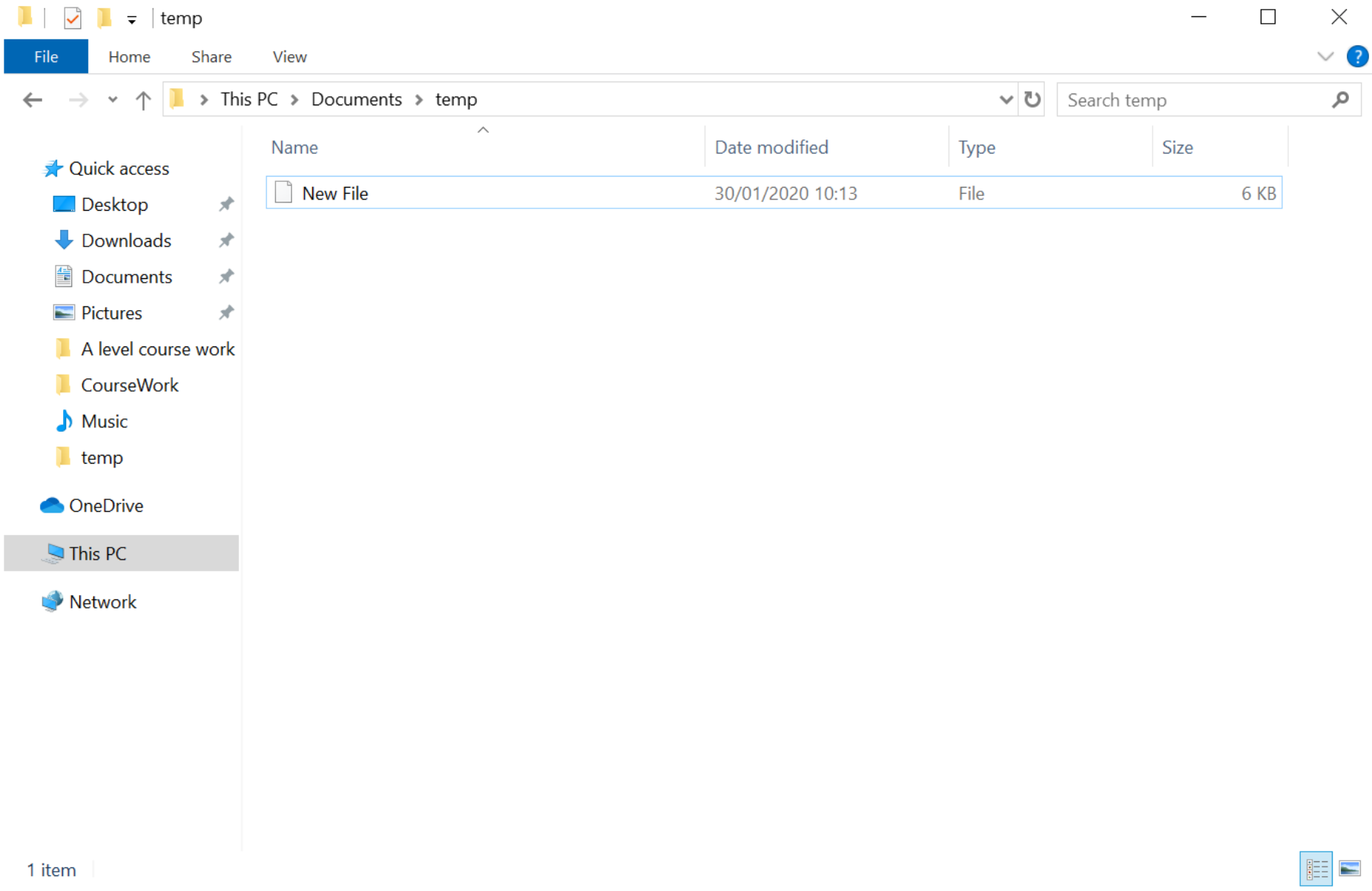

New Window
Are you sure you want to remove everything? Nothing will be kept!
Yes
No


MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:

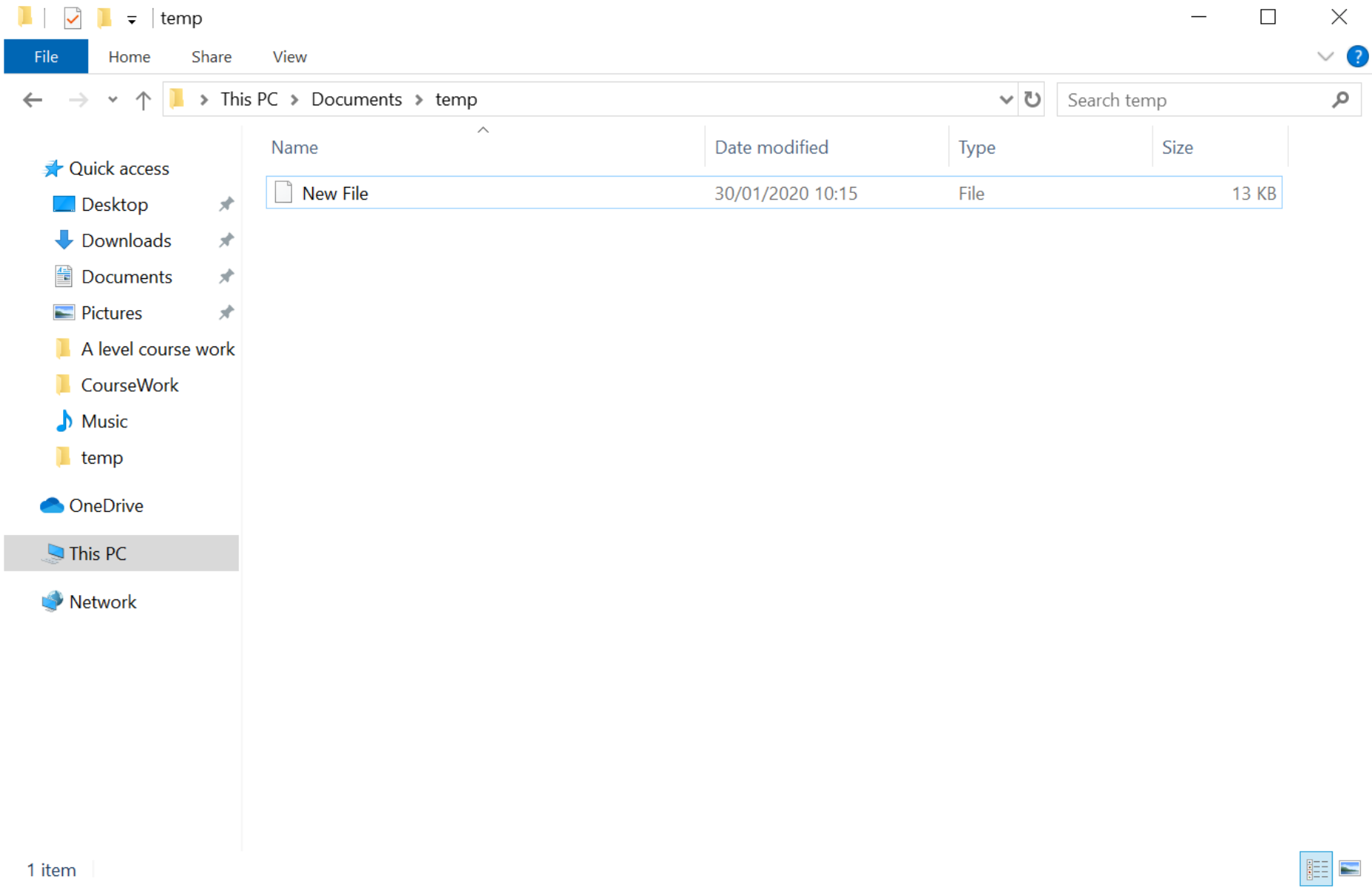

public List<Gate_Class> Gate_List { get; set; } = new List<Gate_Class>();
Gate_List Count = 0
public List<Line_Class> Line_List { get; set; } = new List<Line_Class>();


public List<Line_Class> Line_List { get; set; } = new List<Line_Class>();
//make these custom classe
Line_List Count = 0
public List<Input_Button> Input_Button_List { get; set; } = new List<Input_Button>();
Input_Button_List Count = 0
public List<Output_Circle> Output_Circle_List { get; set; } = new List<Output_Circle>();
Output_Circle_List Count = 0
private File_Creation_Class File_Worker

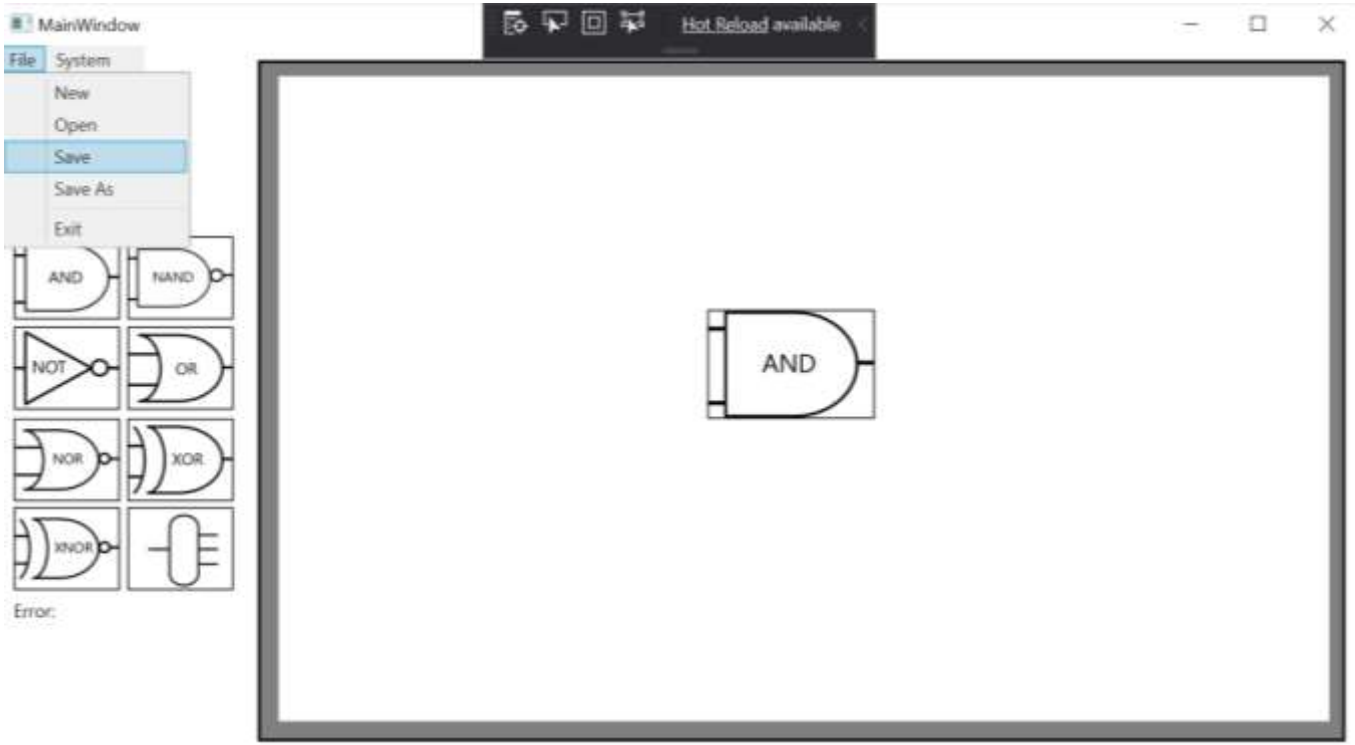
8D.

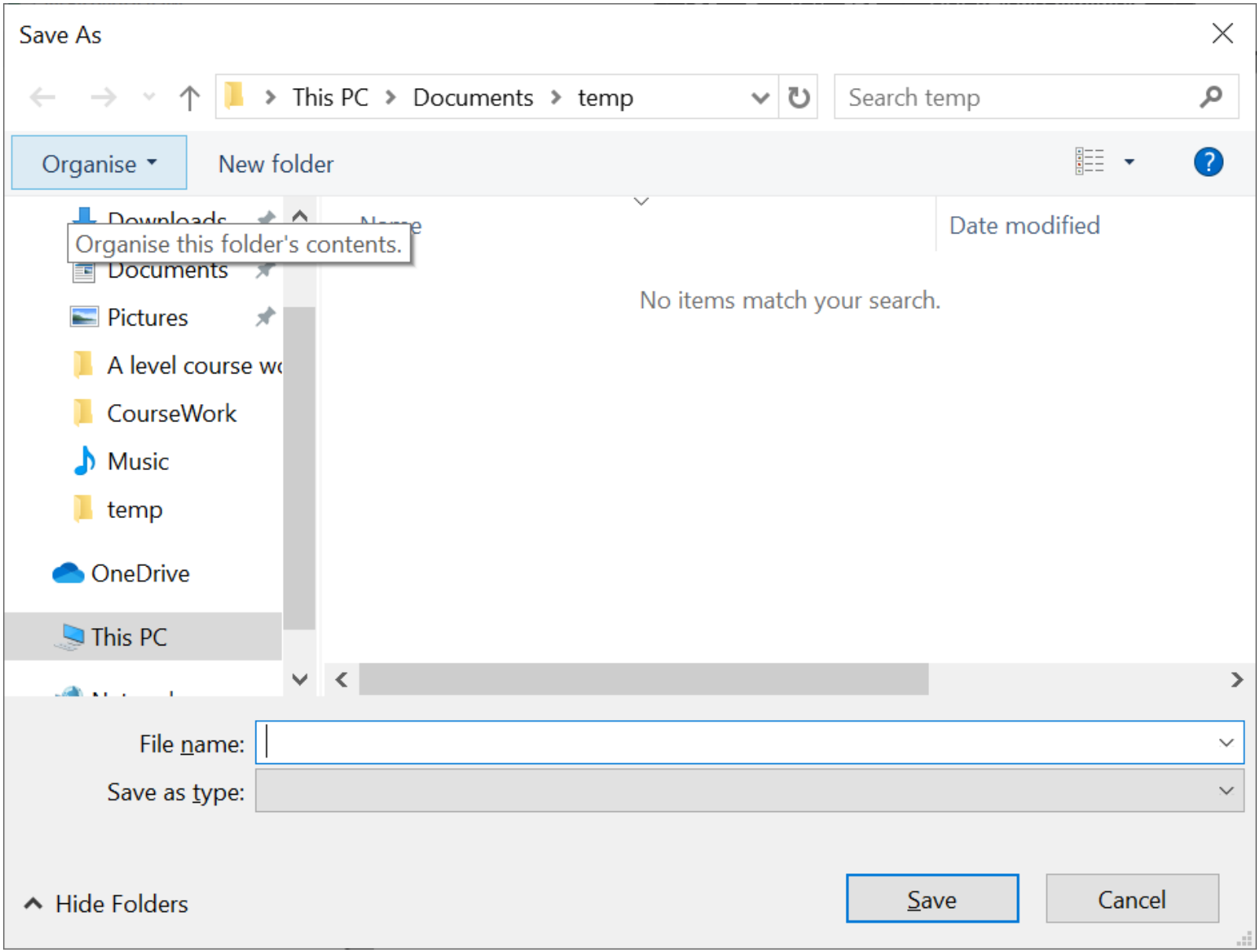

temp
File
Home
Share
View
This PC
Documents
temp
Search temp
Quick access
Desktop
Downloads
Documents
Pictures
A level course work
CourseWork
Music
temp
OneDrive
This PC
Network
Name
Date modified
Type
Size
New File
30/01/2020 10:13
File
6 KB
1 item


MainWindow
File
System
New
Open
Save
Save As
Exit
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
Hot Reload available

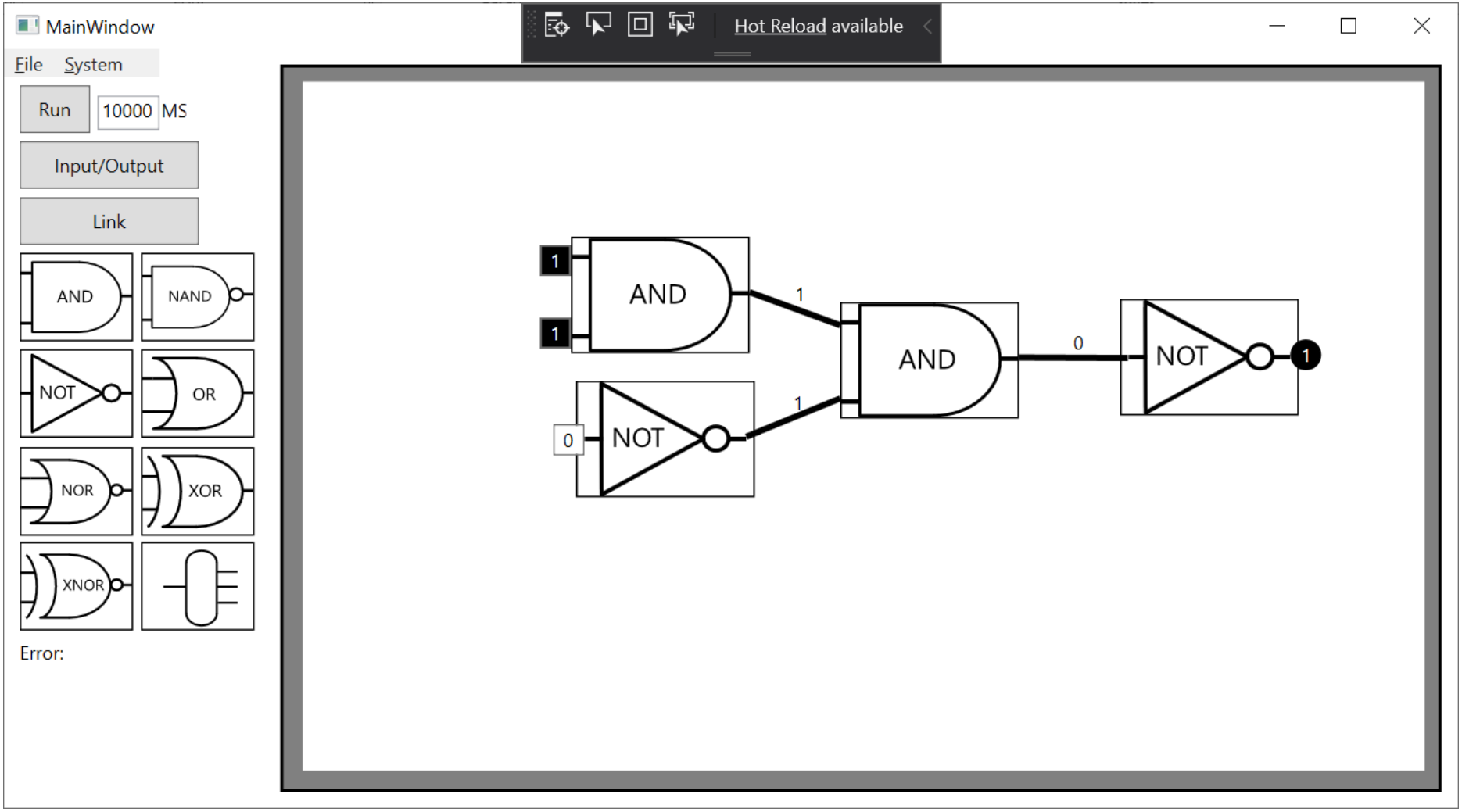

temp
File
Home
Share
View
This PC
Documents
temp
Search temp
Quick access
Desktop
Downloads
Documents
Pictures
A level course work
CourseWork
Music
temp
OneDrive
This PC
Network
Name
Date modified
Type
Size
New File
30/01/2020 10:15
File
13 KB
1 item

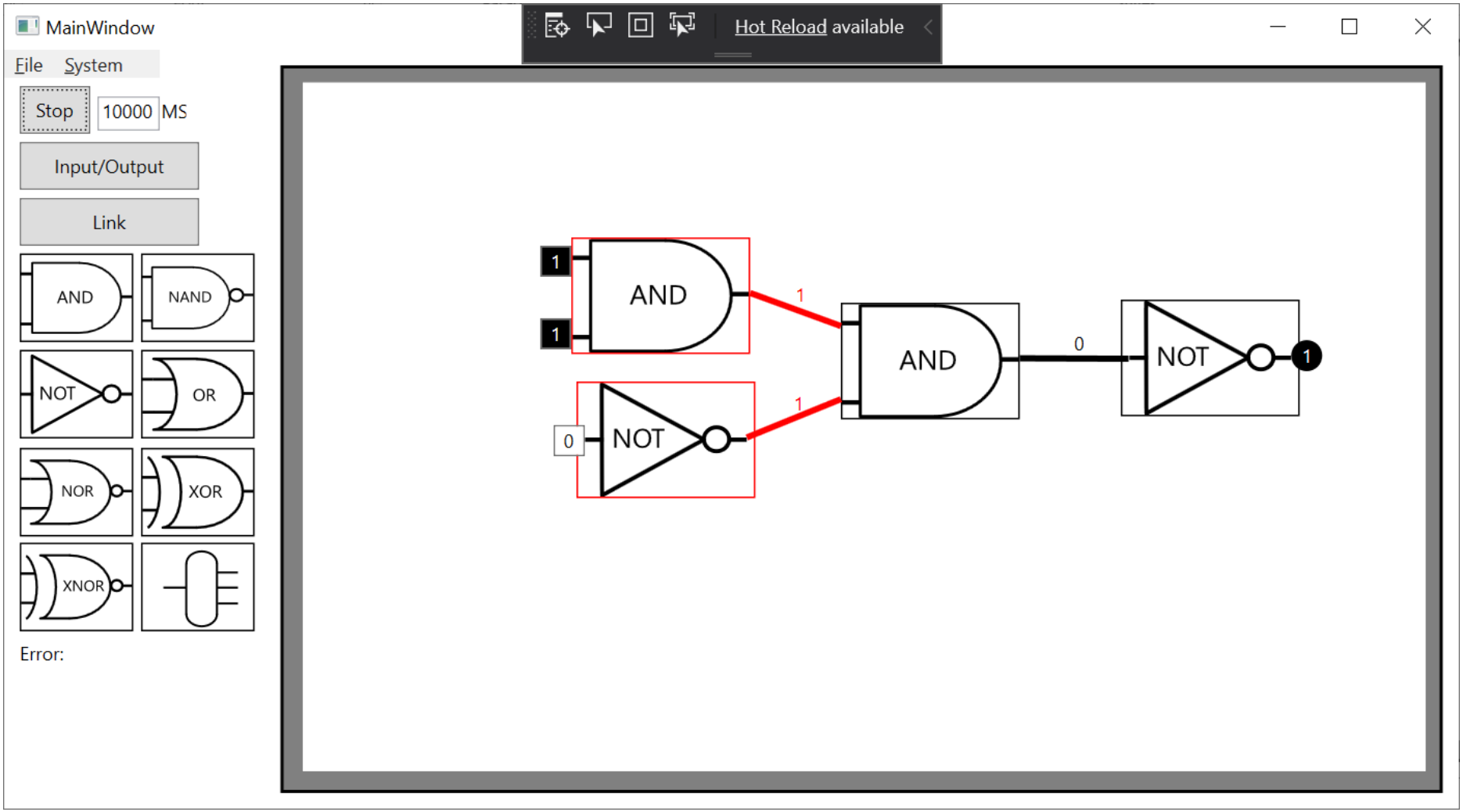
8E.

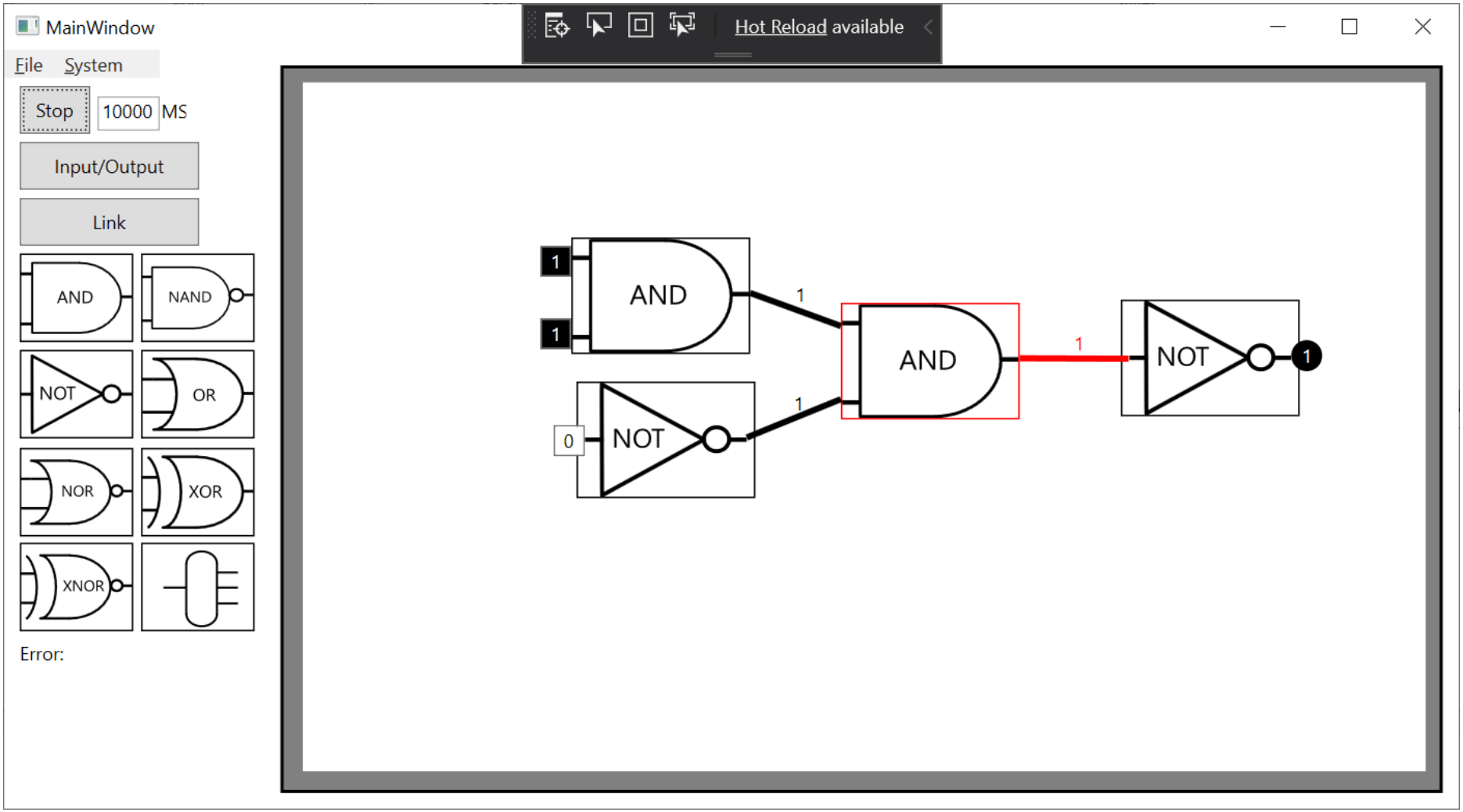

MainWindow
Hot Reload available
File
System
New
Open
Save
Save As
Exit
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND

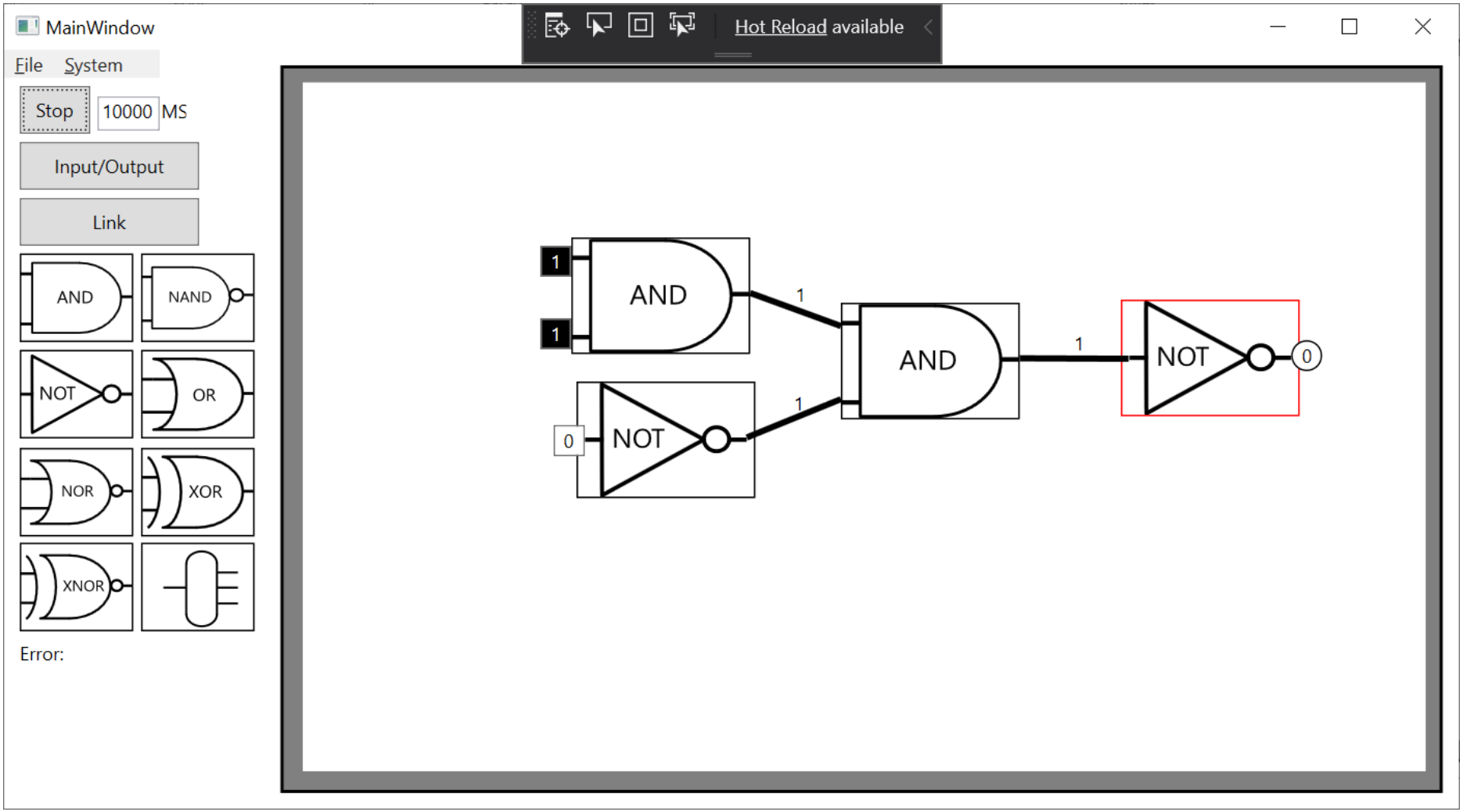

Save As
This PC
Documents
temp
Search temp
Organise
New folder
Organise this folder's contents.
Documents
Pictures
A level course wo
CourseWork
Music
temp
OneDrive
This PC
Date modified
No items match your search.
File name:
Save as type:
Hide Folders
Save
Cancel

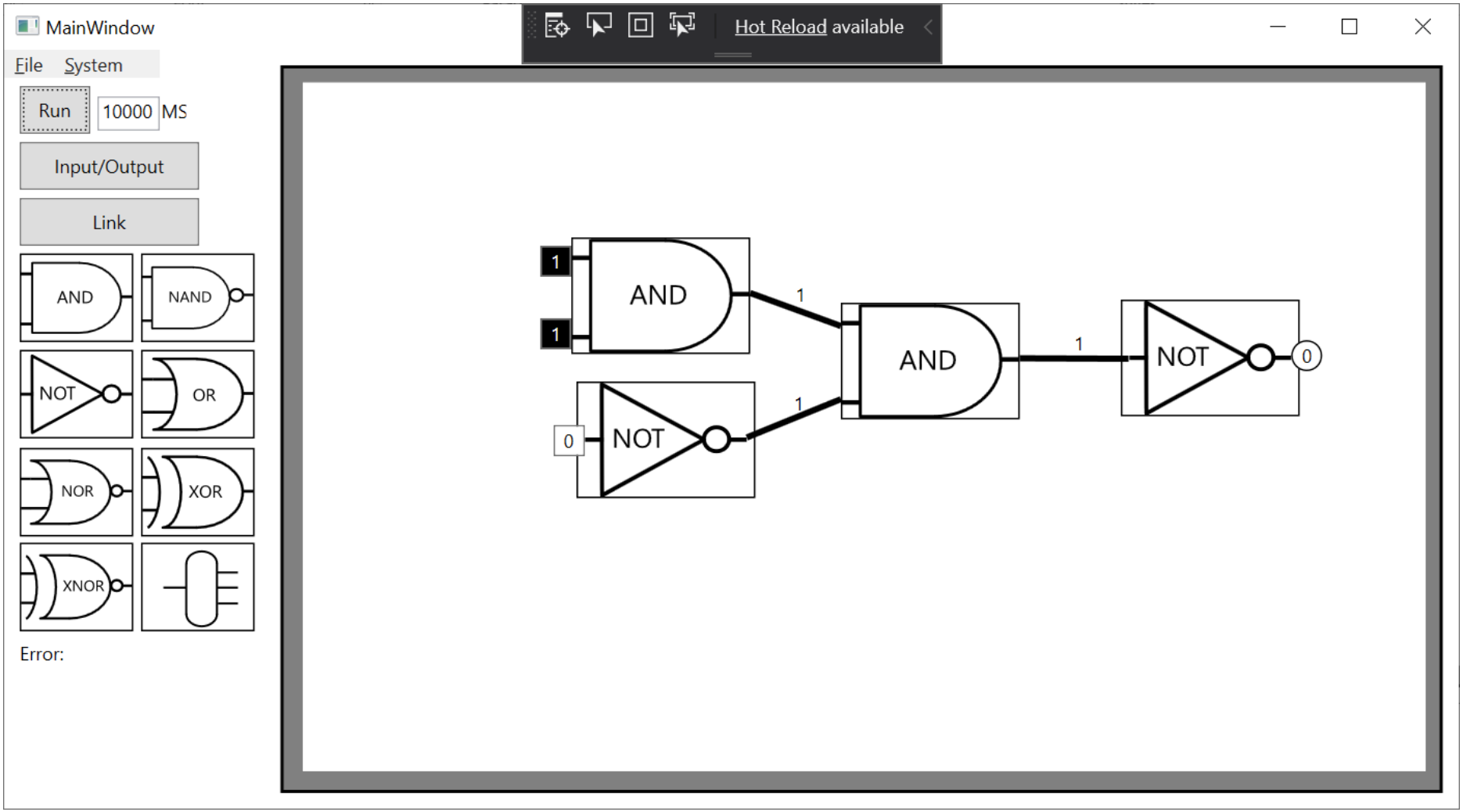
9A.


MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
0
NOT
1
AND
0
NOT
1

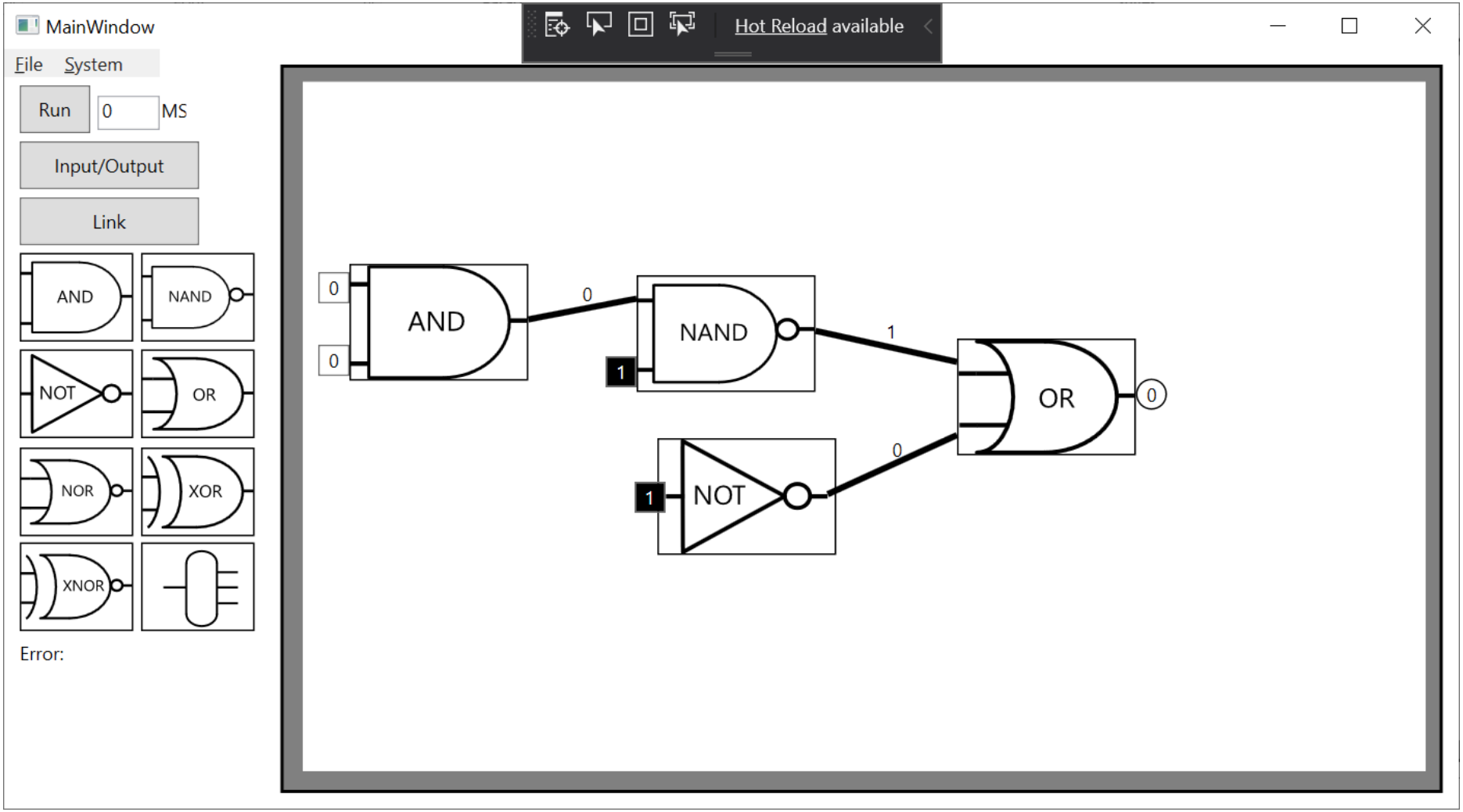

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
0
NOT
1
AND
0
NOT
1

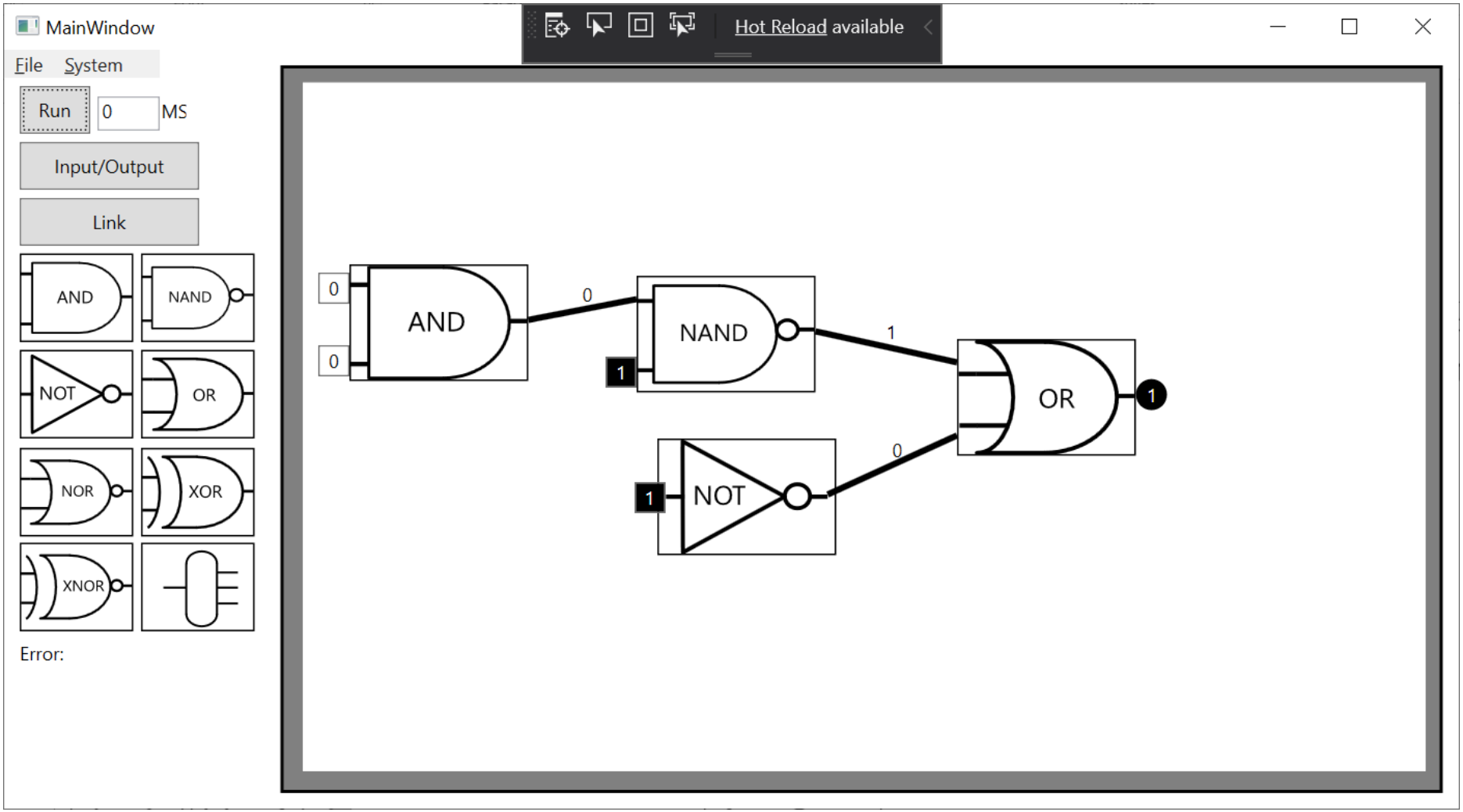

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
0
NOT
1
AND
1
NOT
1


MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
0
NOT
1
AND
1
NOT
0

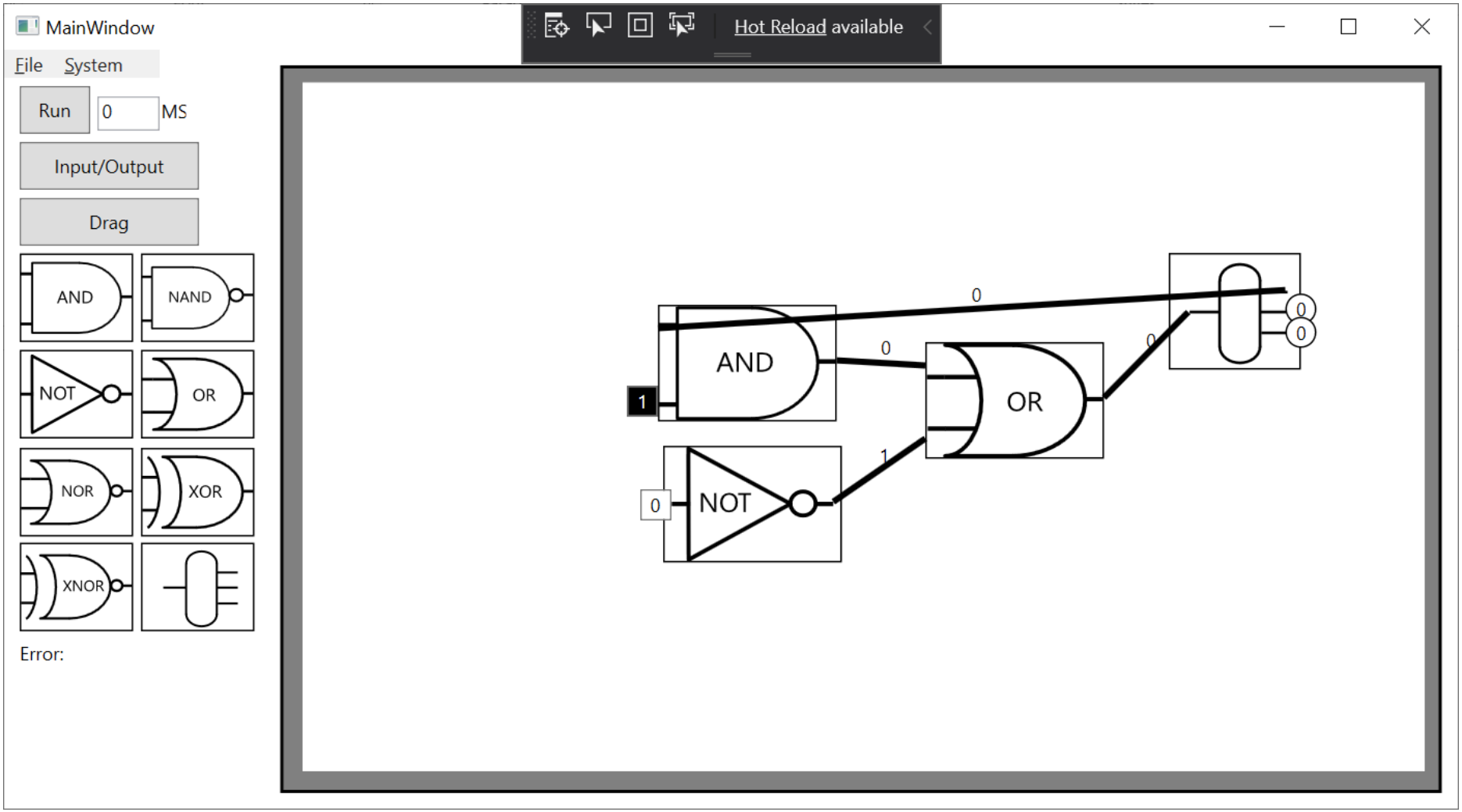

MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
AND
1
NOT
0
0
NOT

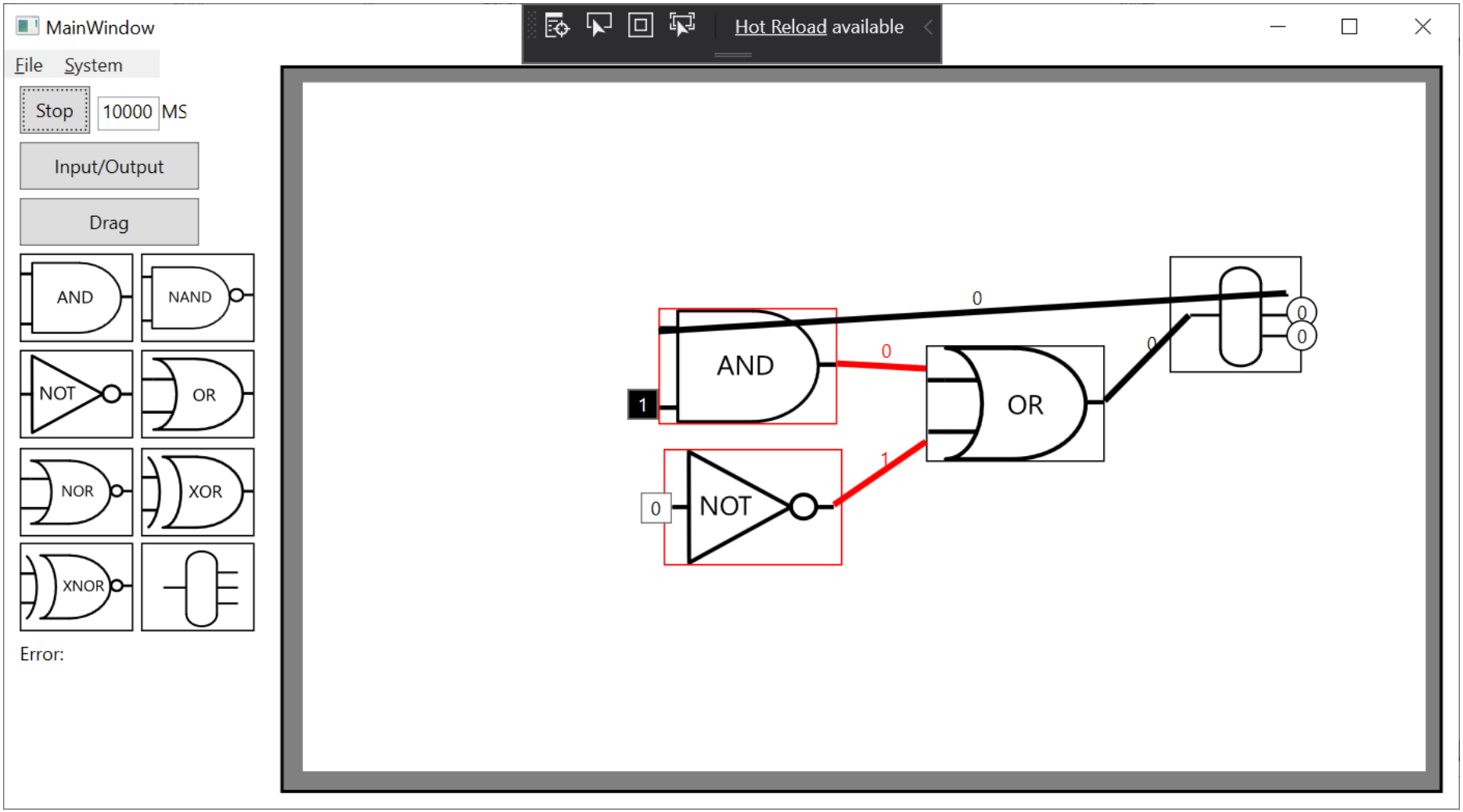
9B.

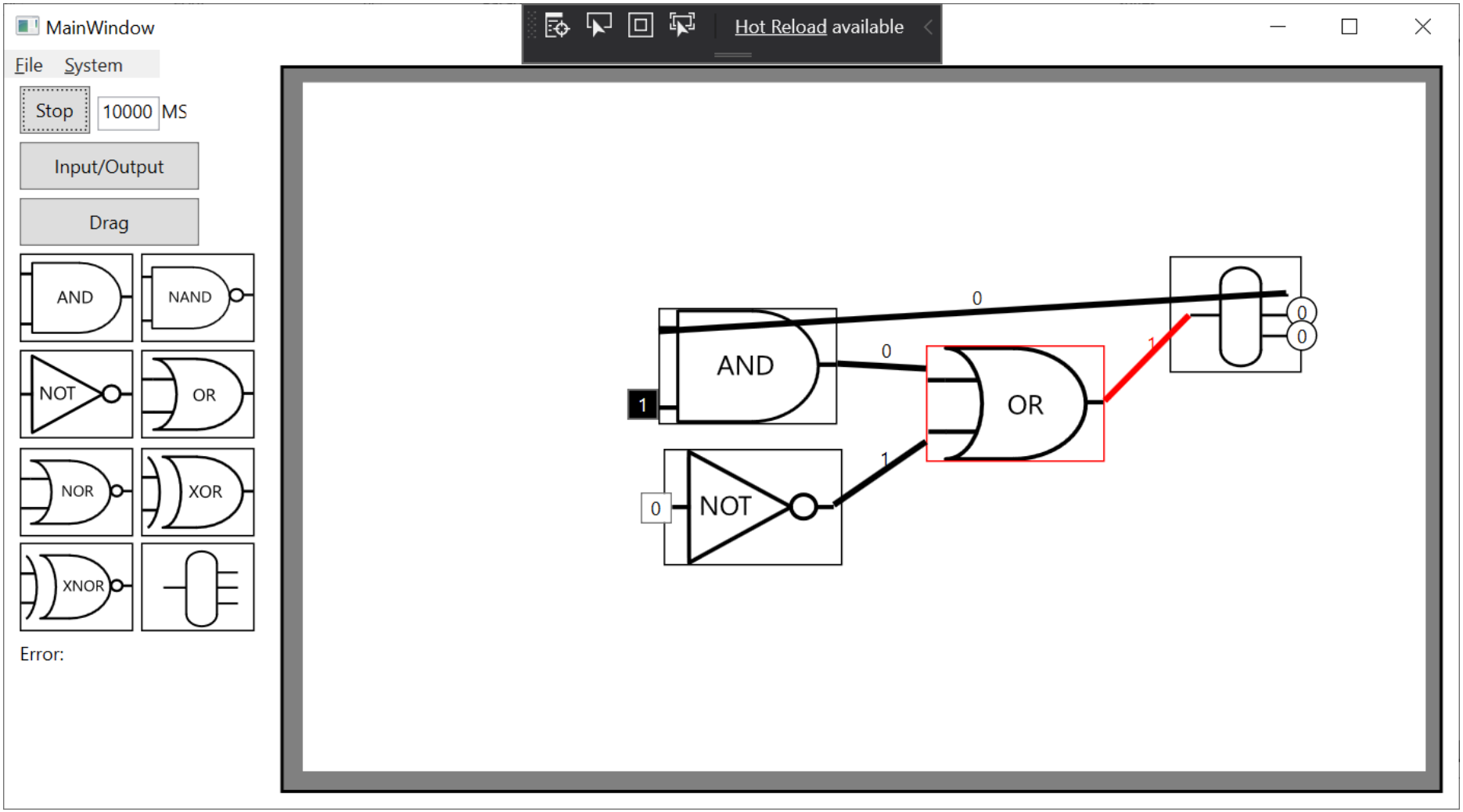

MainWindow
Hot Reload available
File System
Run
0
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
NAND
1
1
OR
0
0
1
NOT

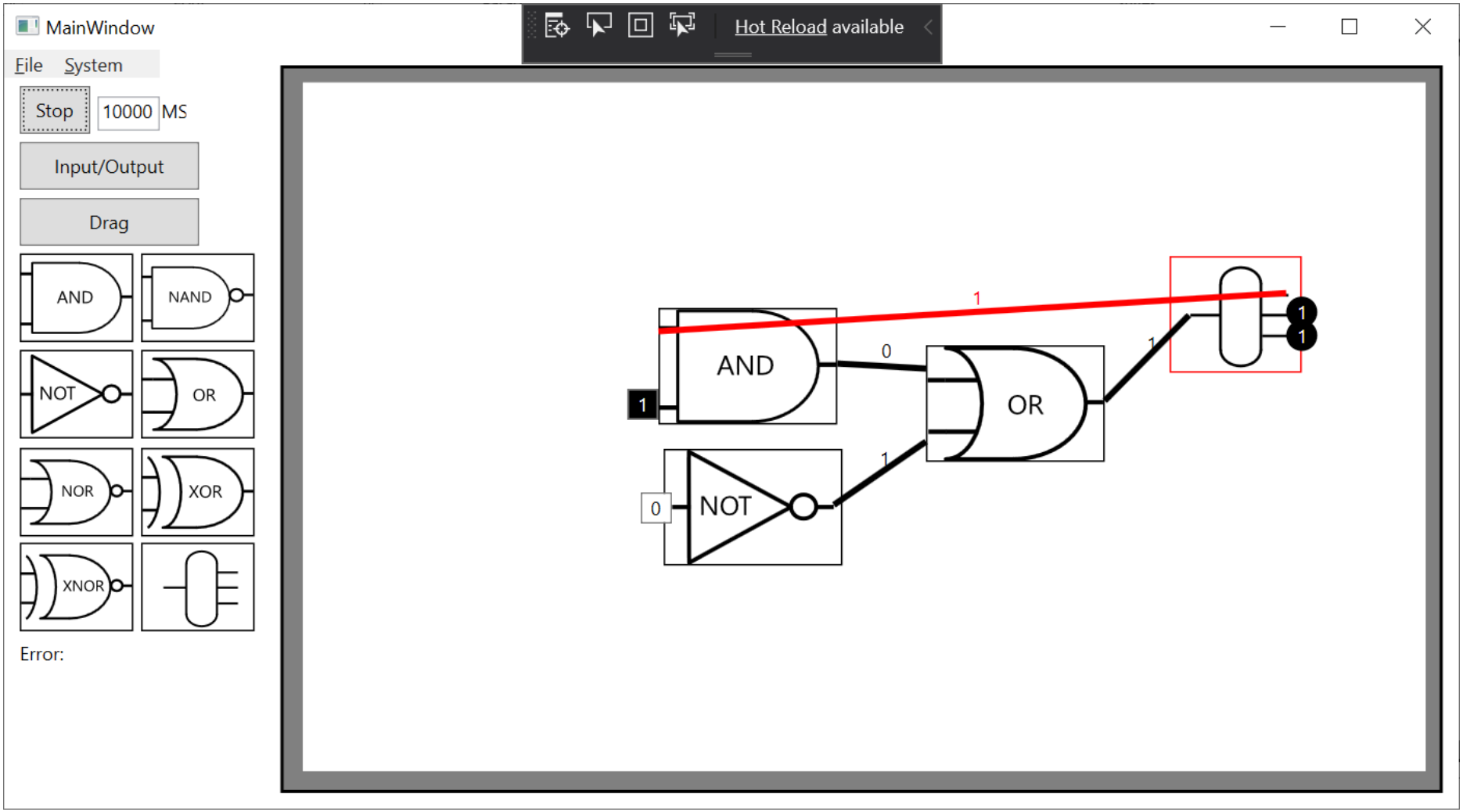

MainWindow
Hot Reload available
File System
Run
0
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
NAND
1
1
OR
1
1
NOT
0

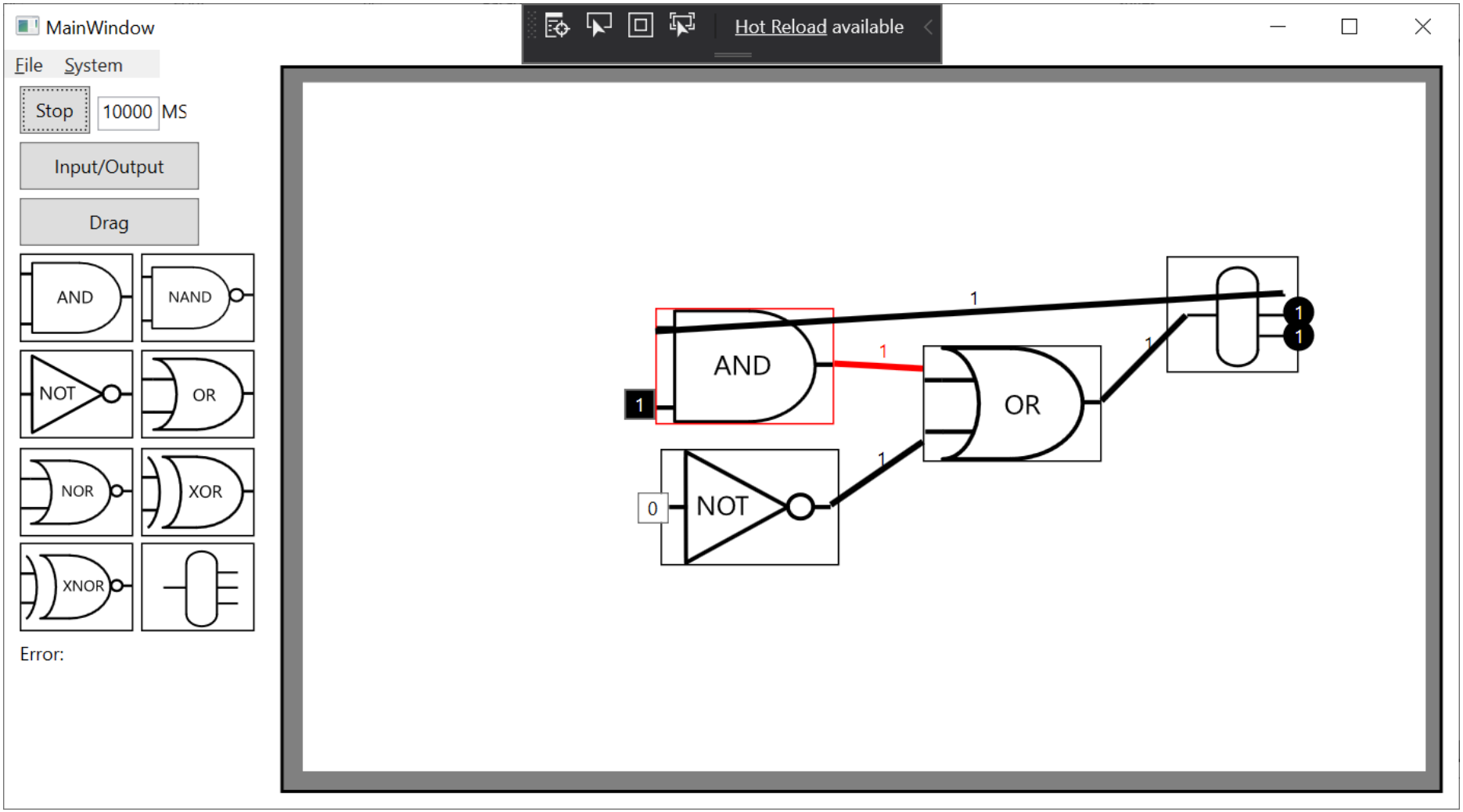
9C.

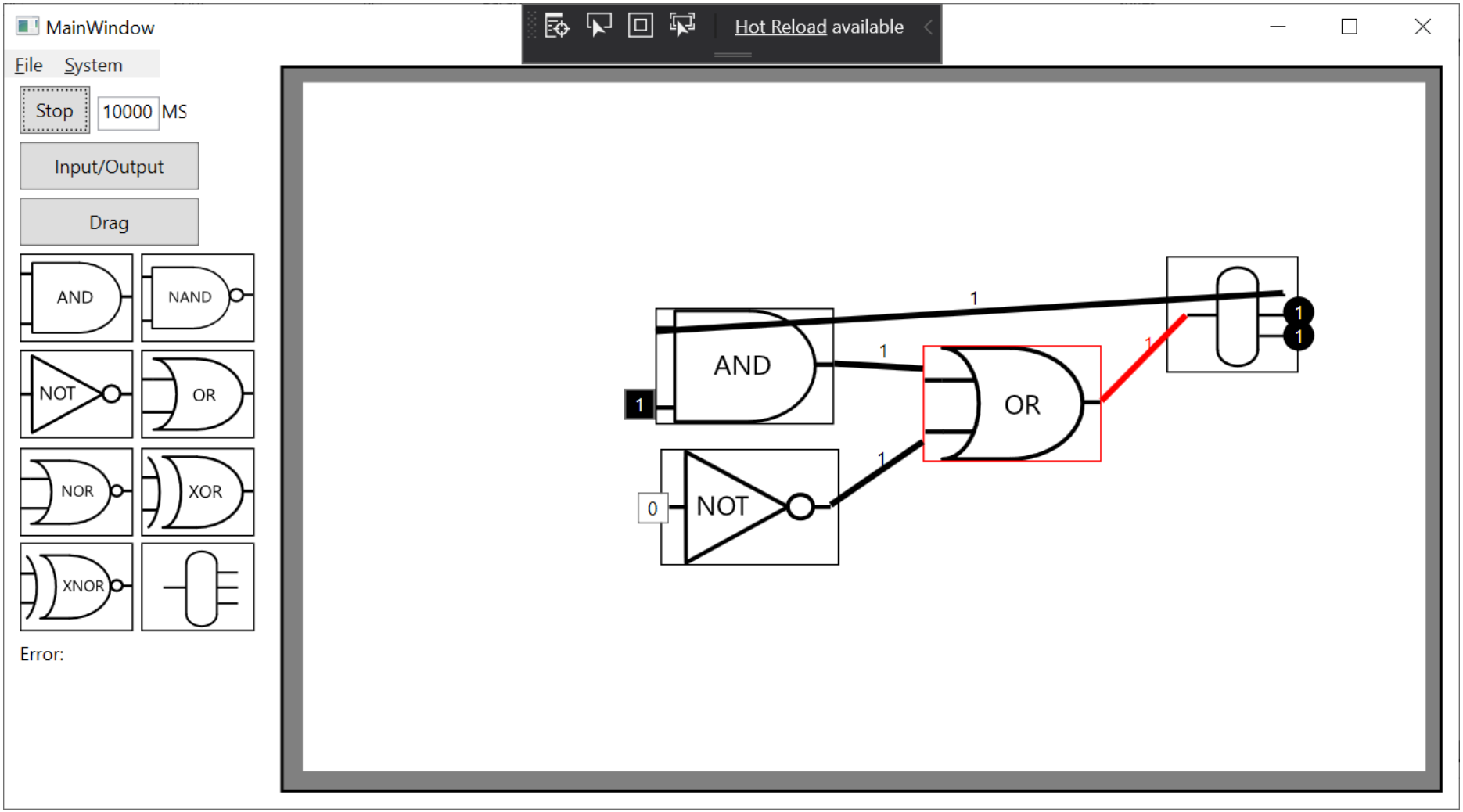

MainWindow
Hot Reload available
File System
Run
0
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
OR
NOT
1
0
0
0
0
1
0
0
0


MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
OR
NOT
1
0
0
0
0
1
0
0
0

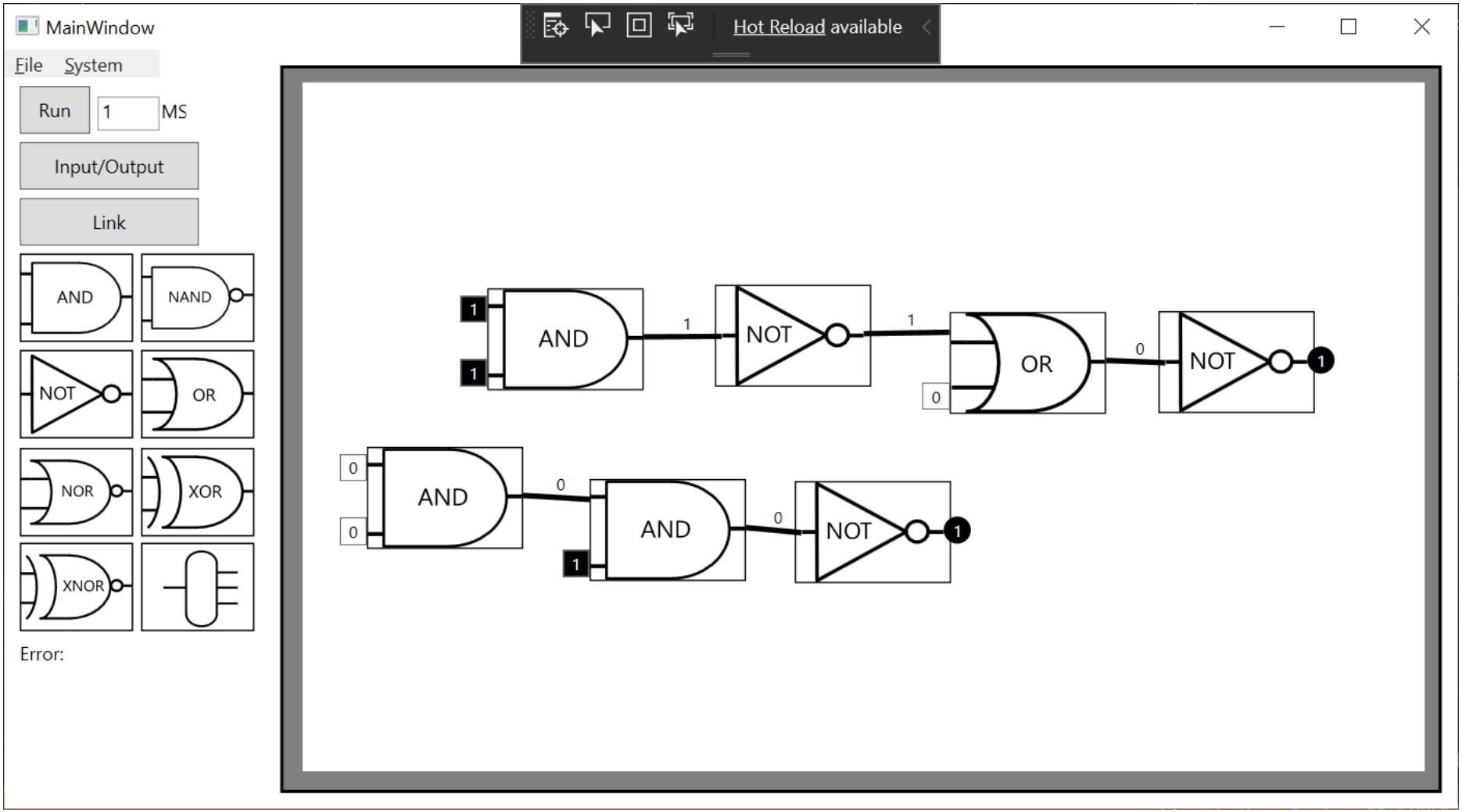

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
OR
NOT
0
1
0
0
1
1
0
0

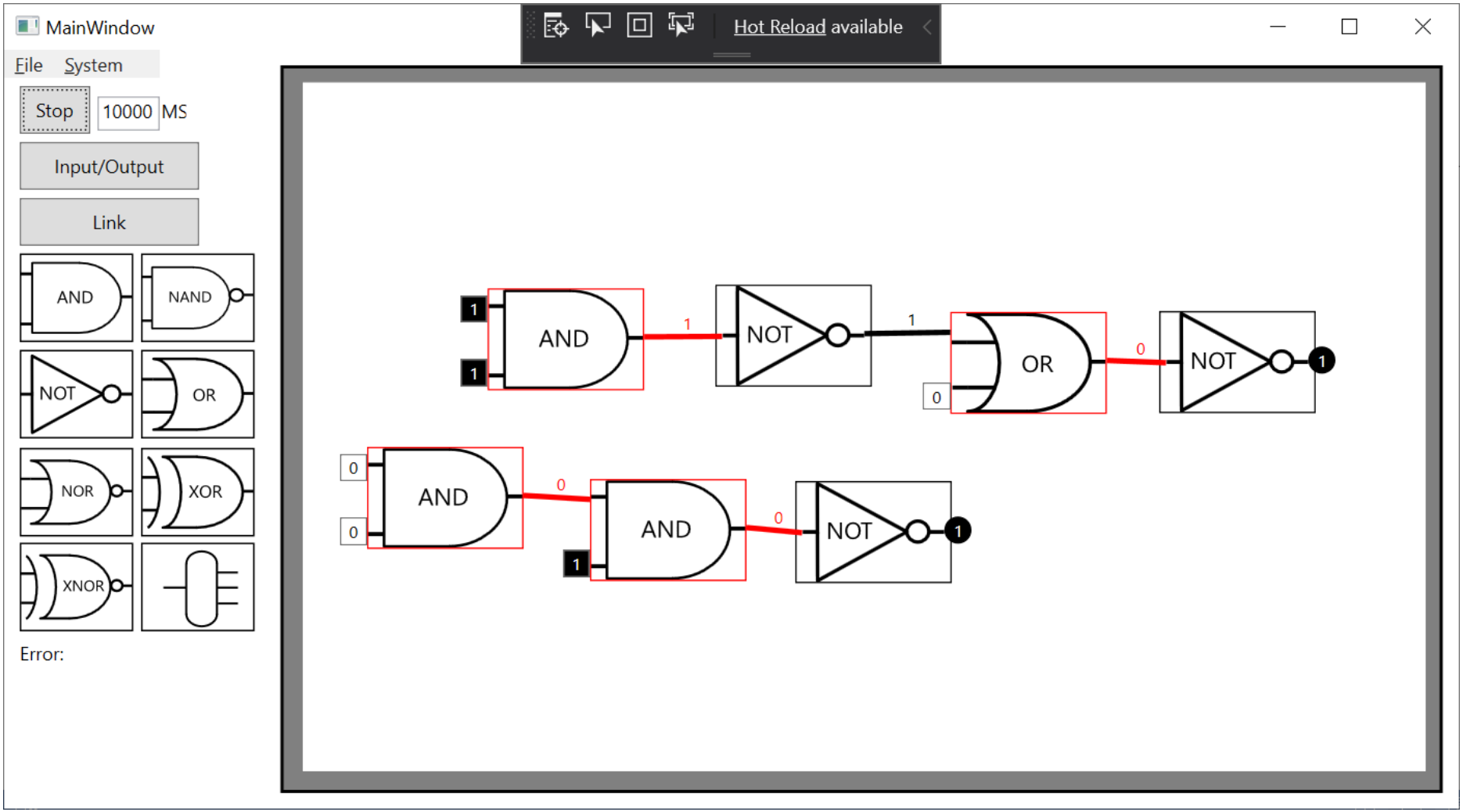

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
OR
NOT
1
0
1
0
1
1
1
1

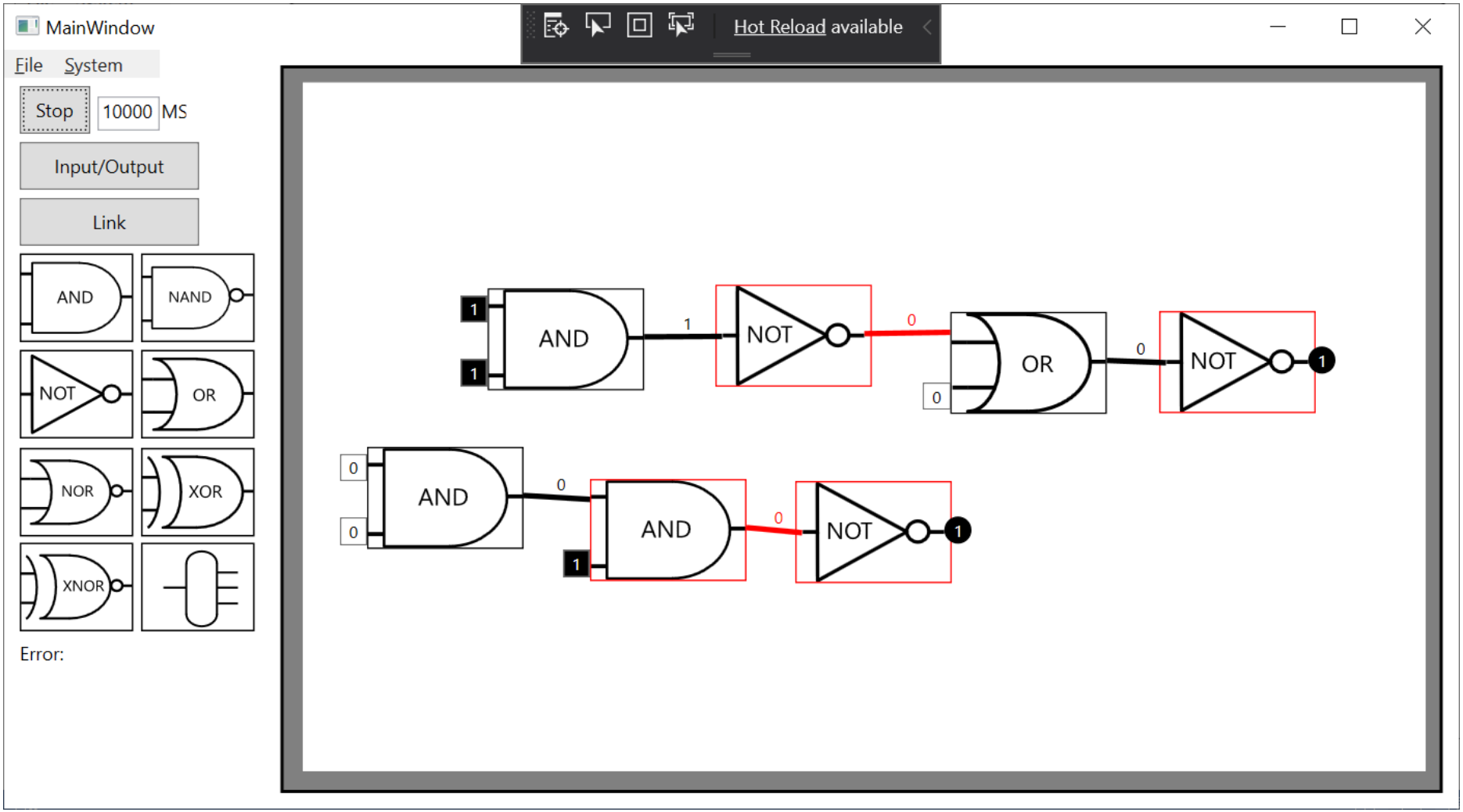

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
OR
NOT
1
0
1
1
1
1
1
1

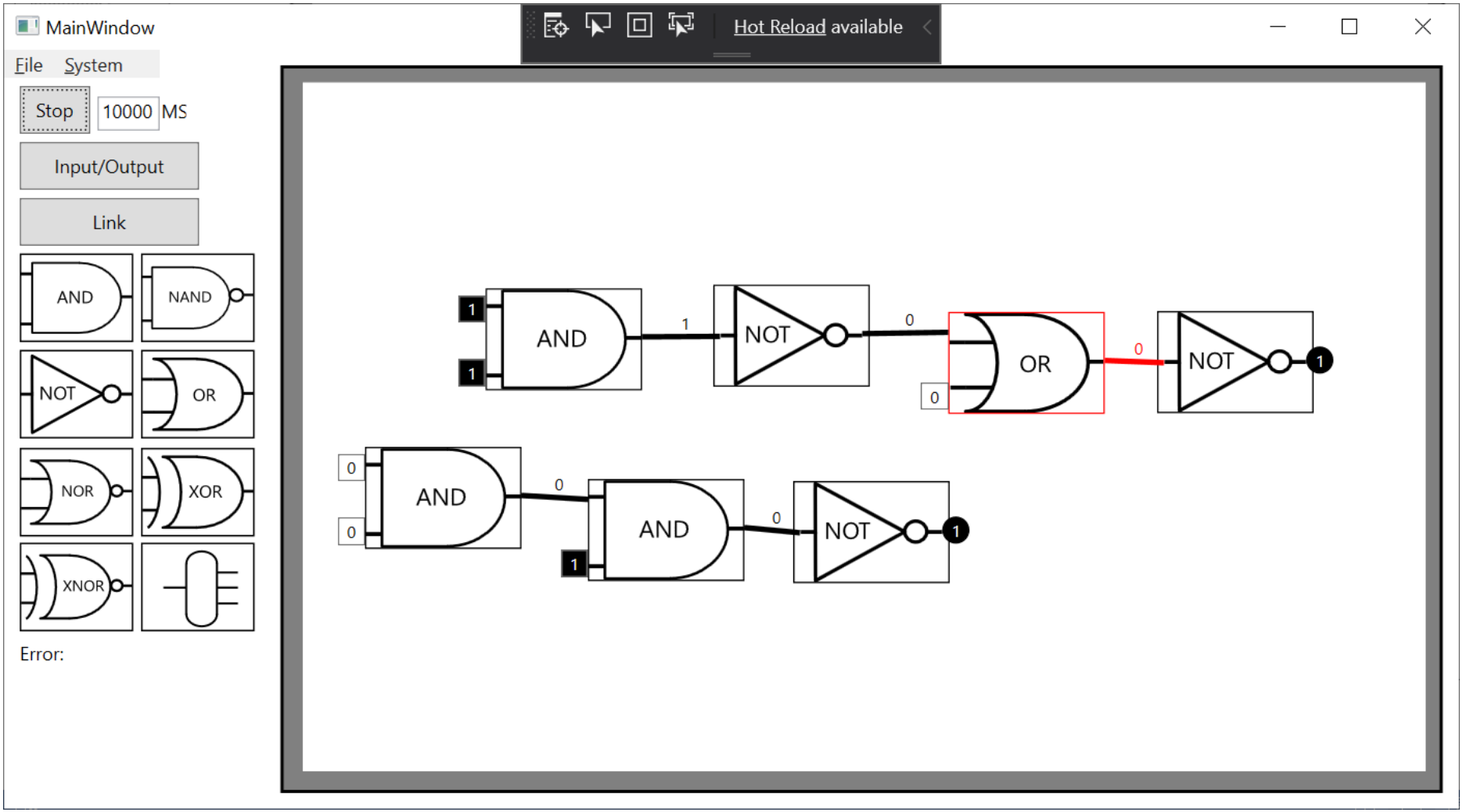

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
AND
OR
NOT
1
0
1
1
1
1
1
1
1

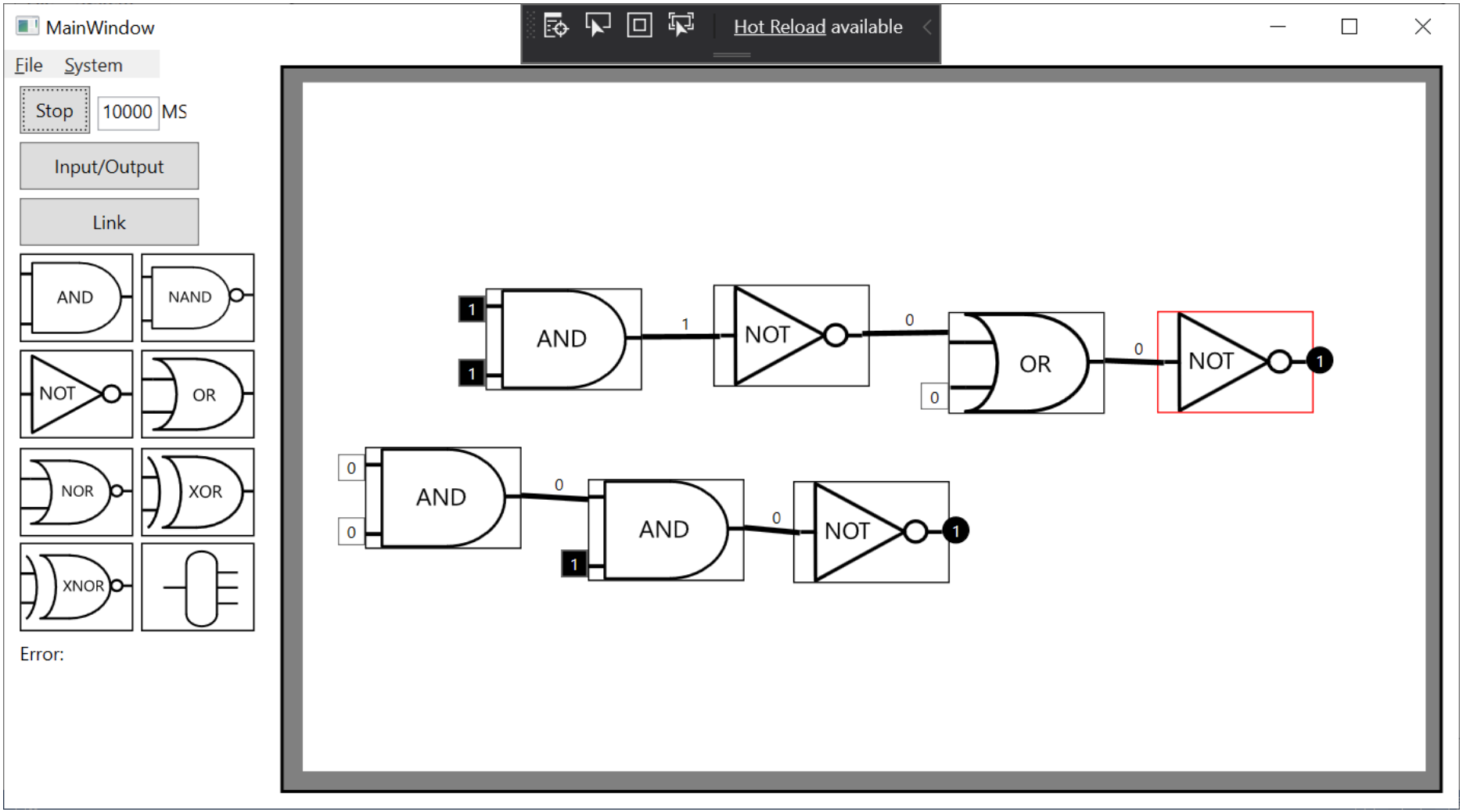
9D.

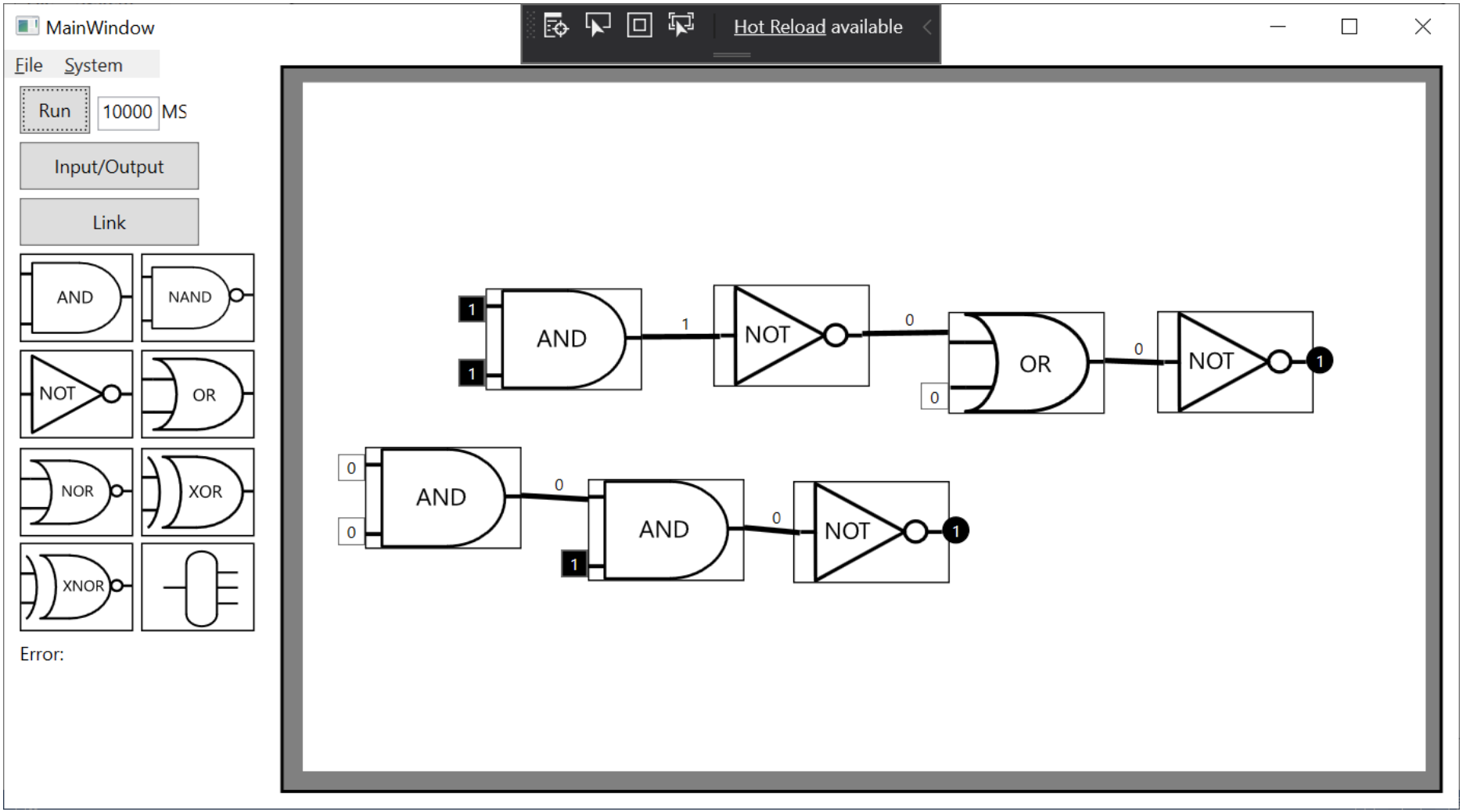

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
NOT
1
0
OR
0
NOT
1
0
0
AND
0
AND
1
0
NOT
1


MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
NOT
1
0
OR
0
NOT
1
0
0
AND
0
AND
1
0
NOT
1

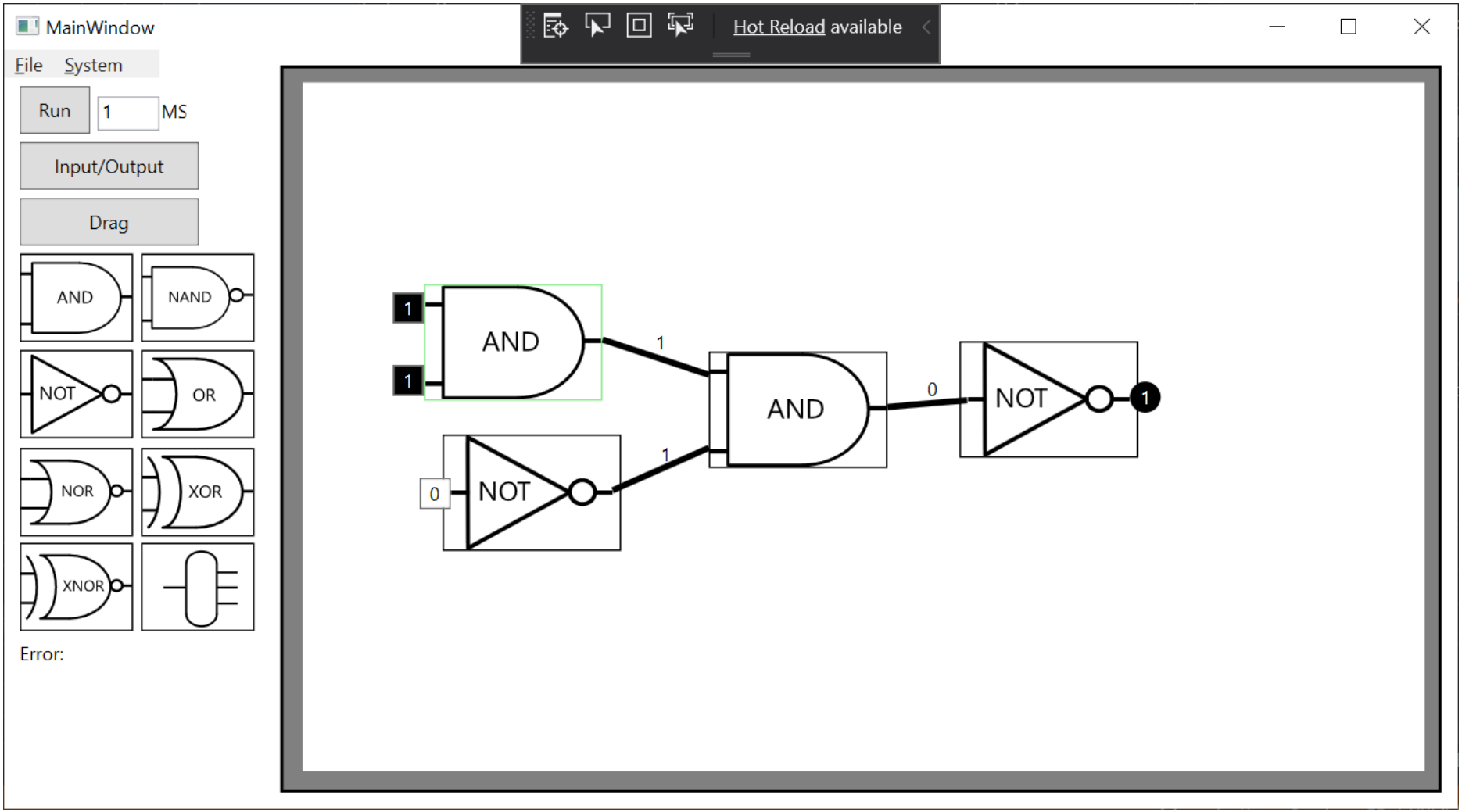

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
NOT
0
OR
0
0
NOT
1
0
0
AND
0
AND
1
0
NOT
1

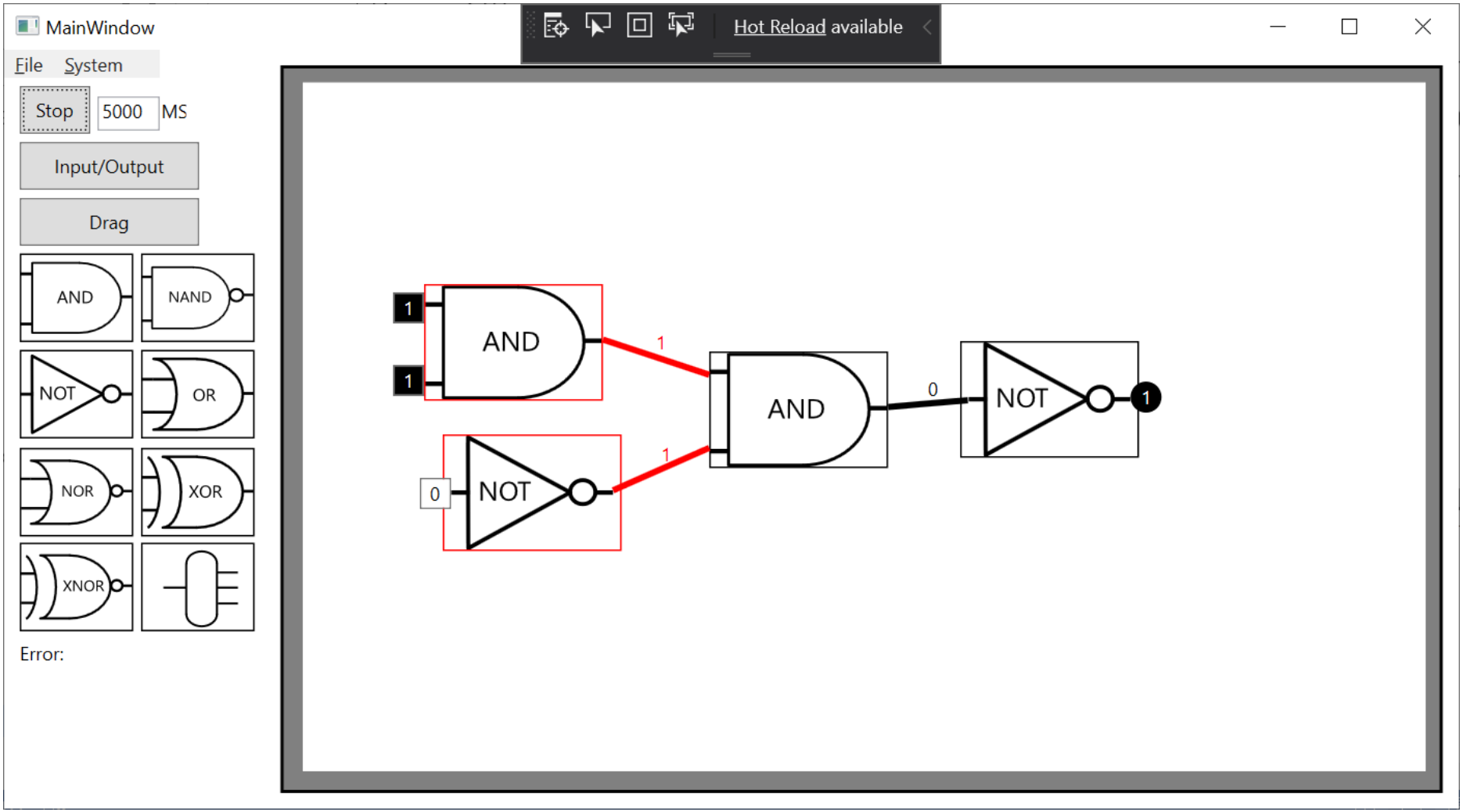

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
NOT
0
0
OR
0
NOT
1
0
0
AND
0
AND
1
0
NOT
1

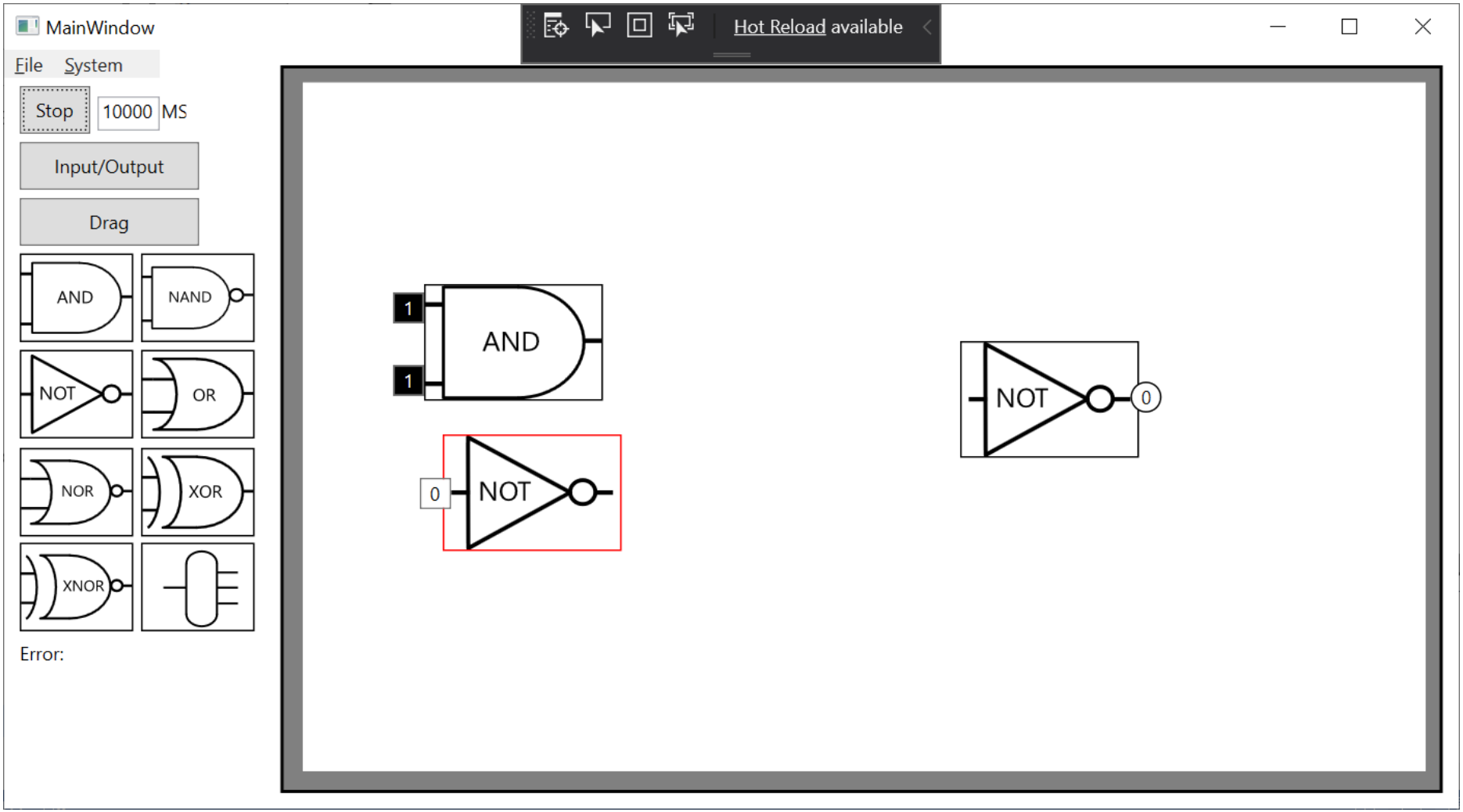

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
NOT
0
0
OR
0
NOT
1
0
0
AND
0
AND
1
0
NOT
1

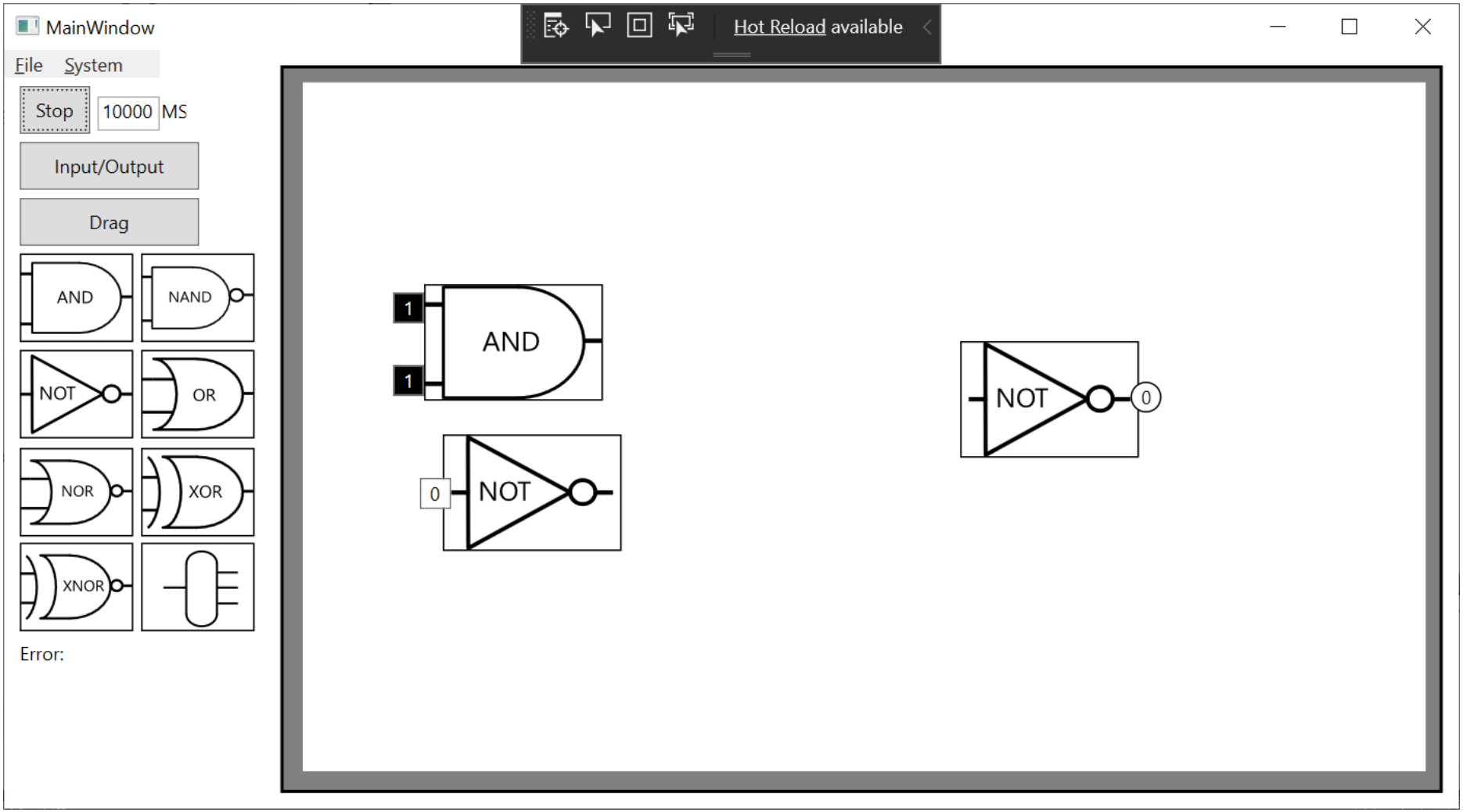

MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
NOT
0
0
OR
0
NOT
1
0
0
AND
0
AND
1
0
NOT
1

9E

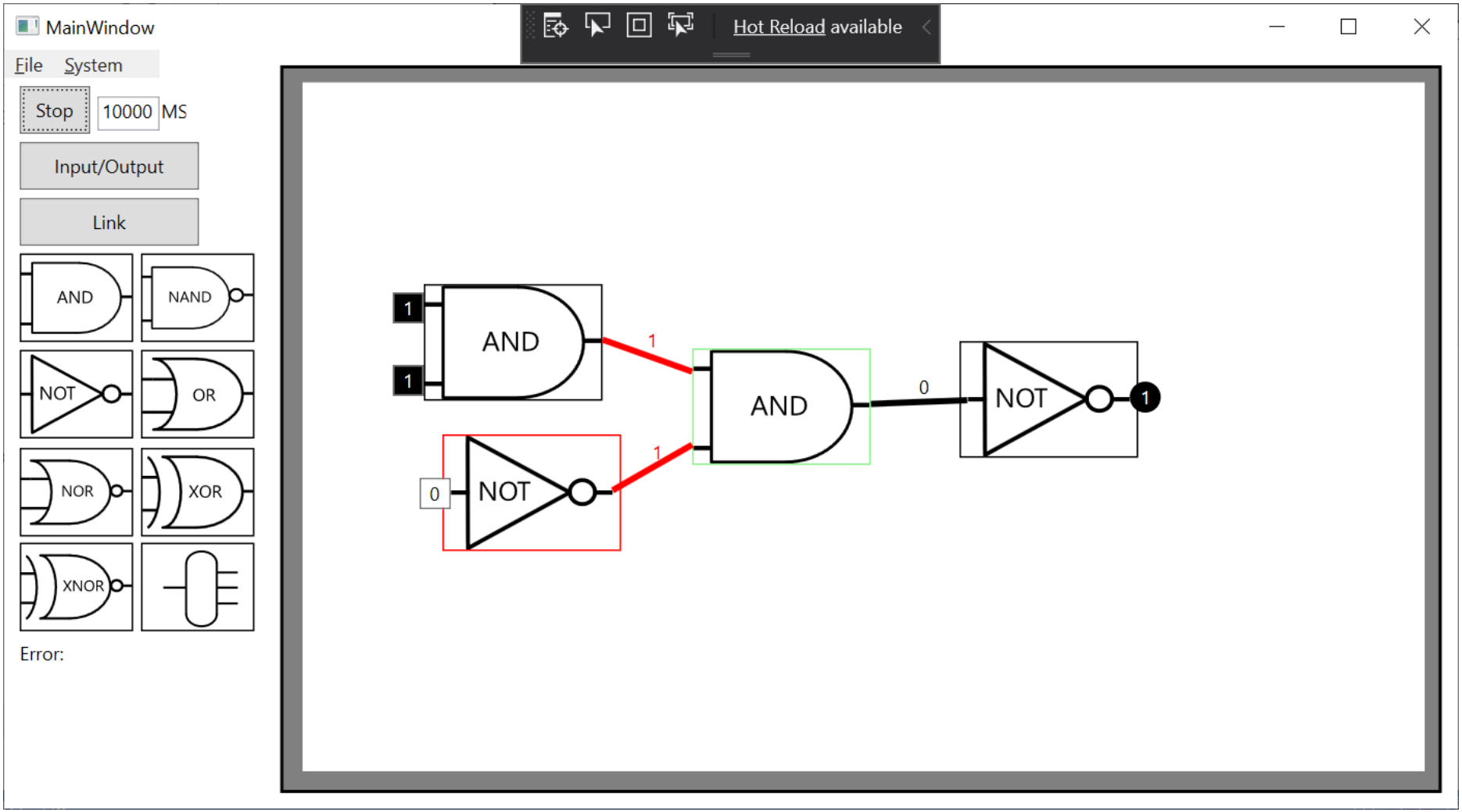

MainWindow
Hot Reload available
File System
Run
1
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
AND
0
NOT
1
0
NOT
1

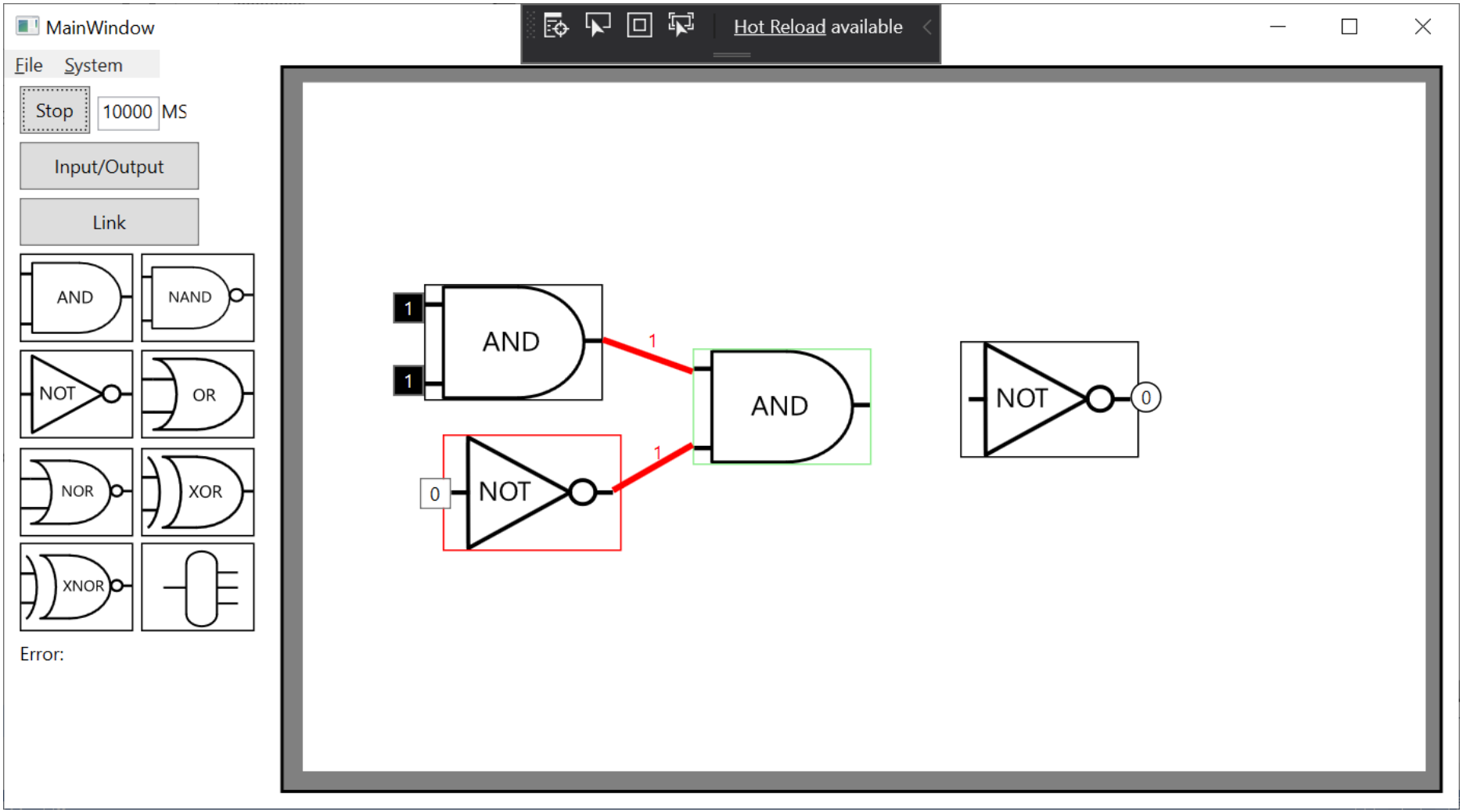

MainWindow
Hot Reload available
File System
Stop
5000
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
AND
0
NOT
1
0
NOT
1

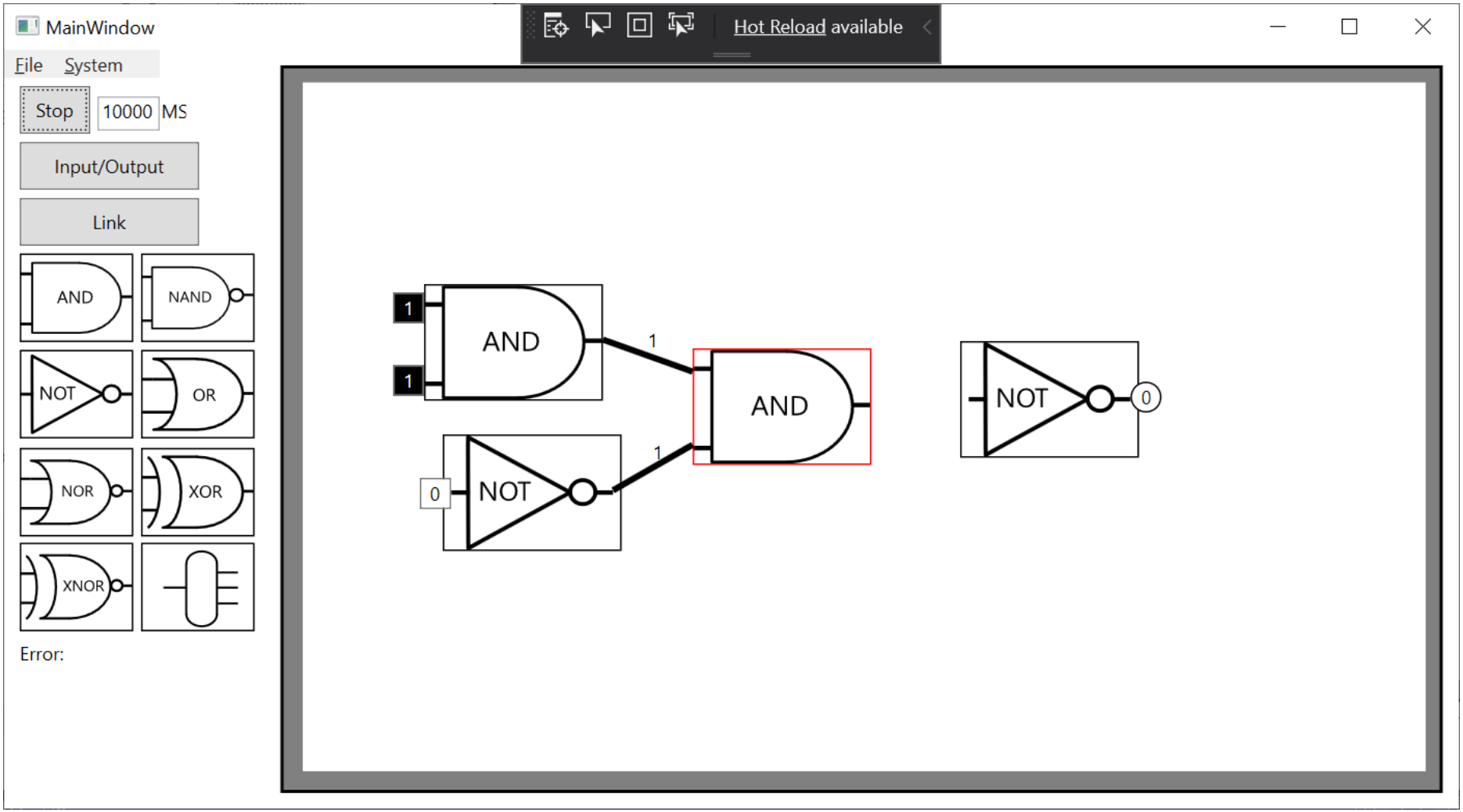

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
0
NOT
NOT
0

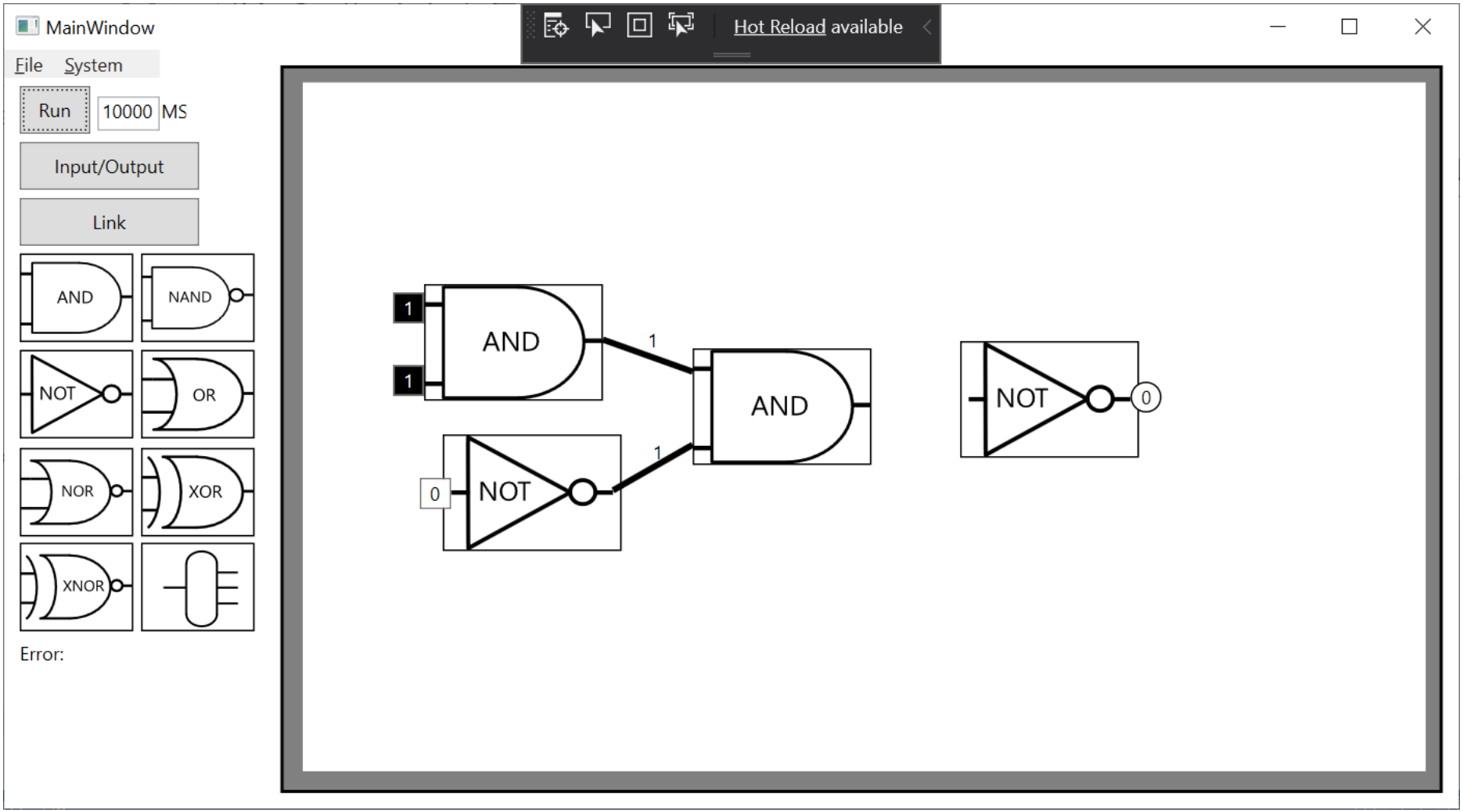

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
AND
1
0
NOT
NOT
0

9F

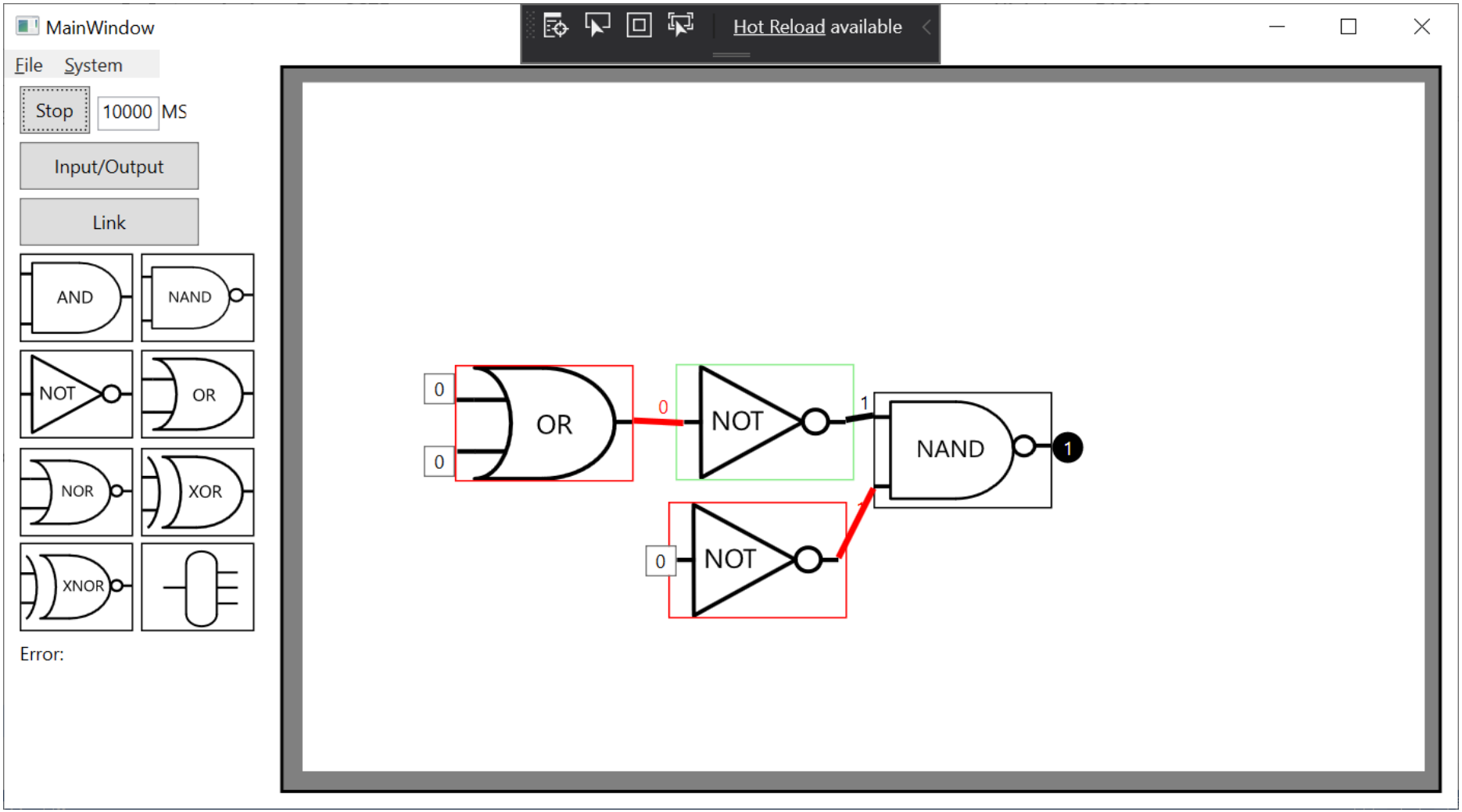

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
0
NOT
1
AND
0
NOT
1

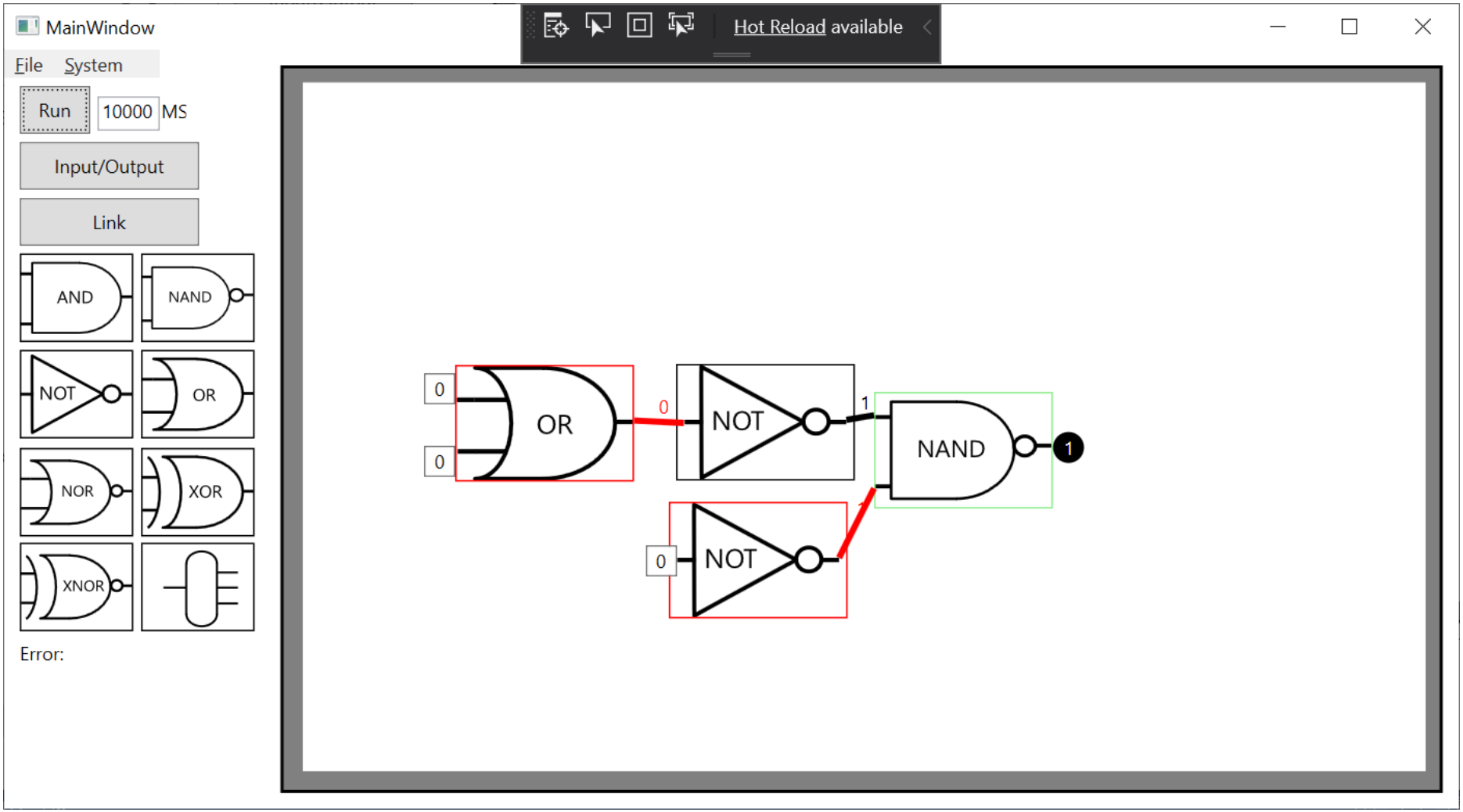

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
AND
0
NOT
1
NOT
0

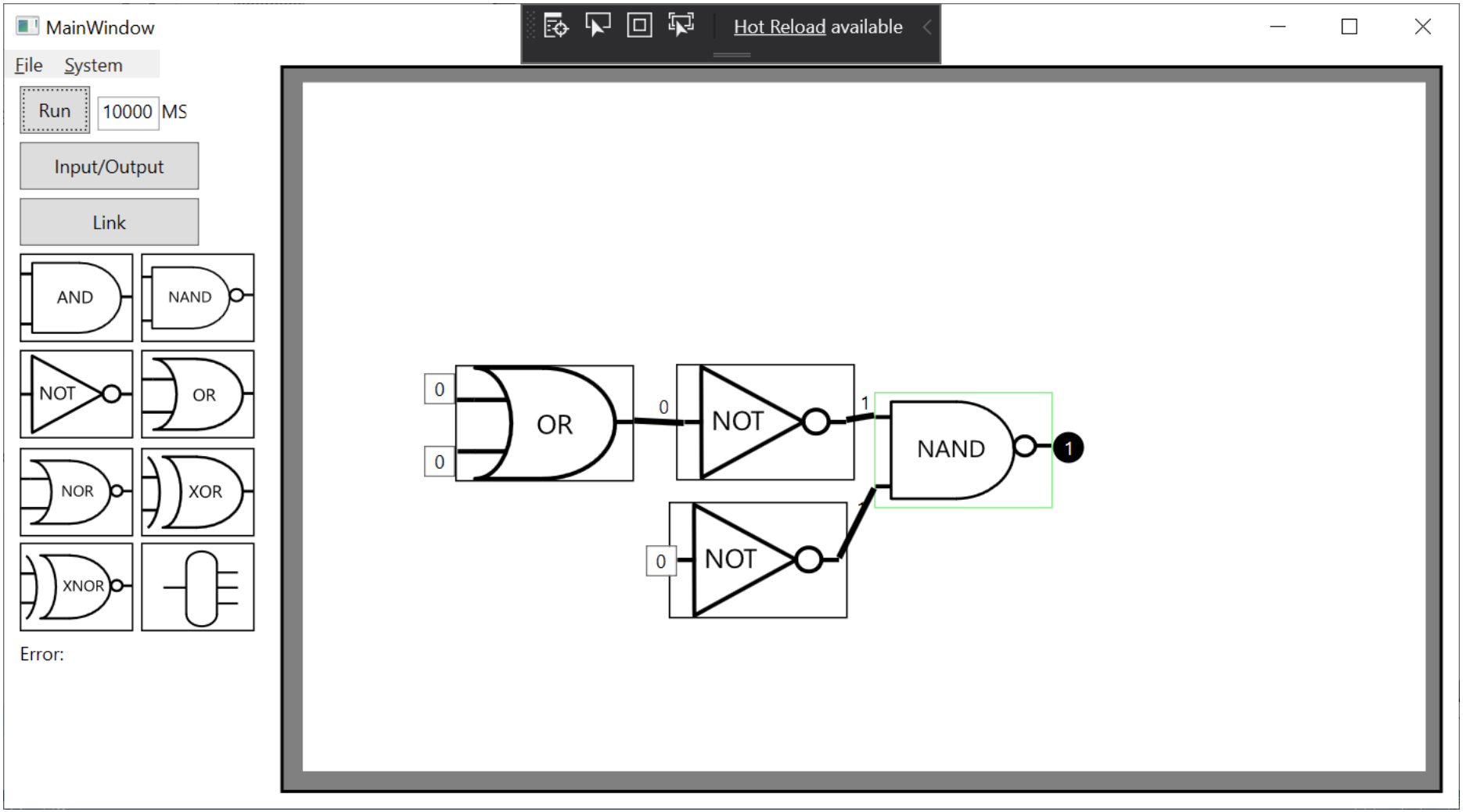

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
AND
0
NOT
1
NOT
0


MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
1
1
AND
1
AND
0
NOT
1
NOT
0

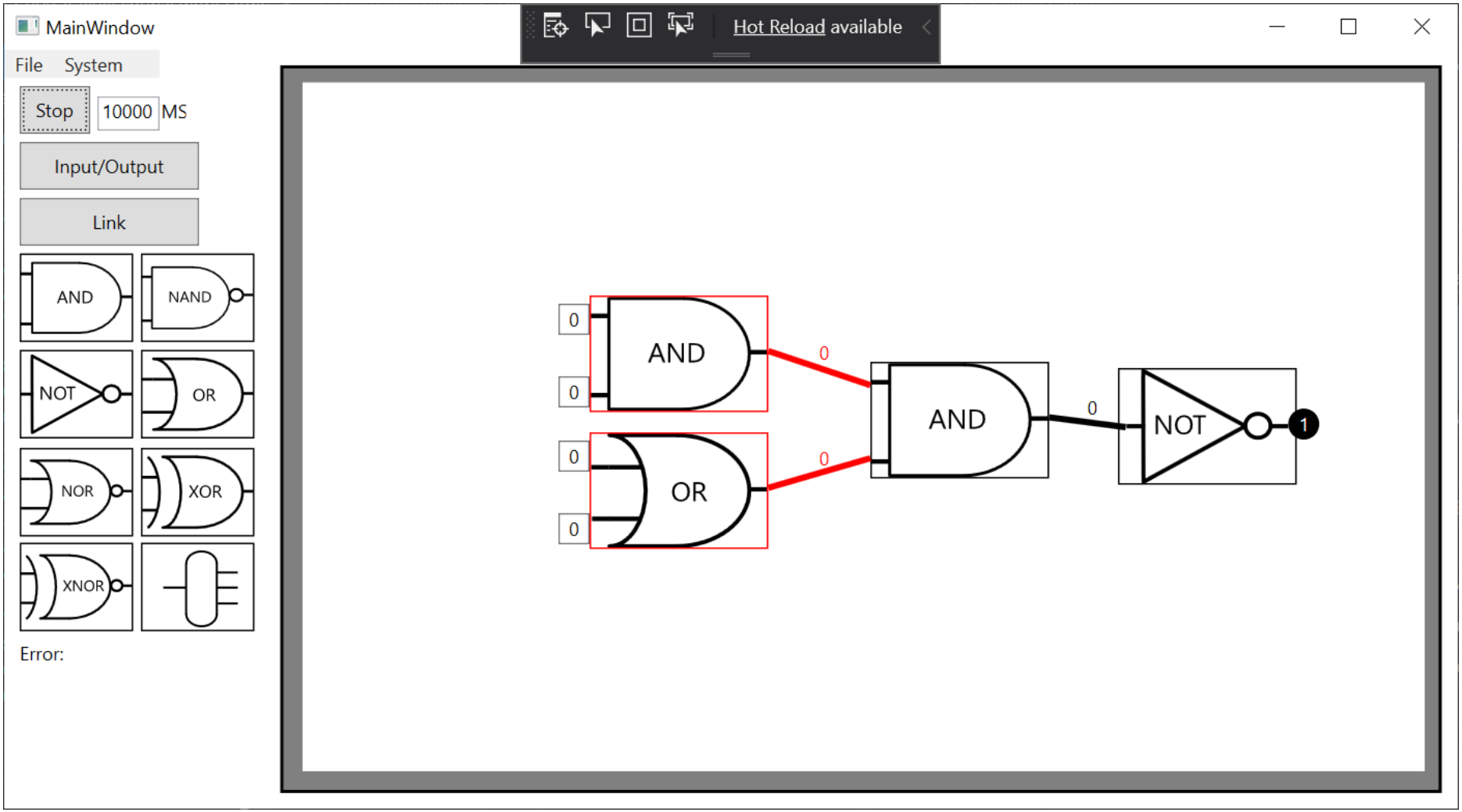
9G

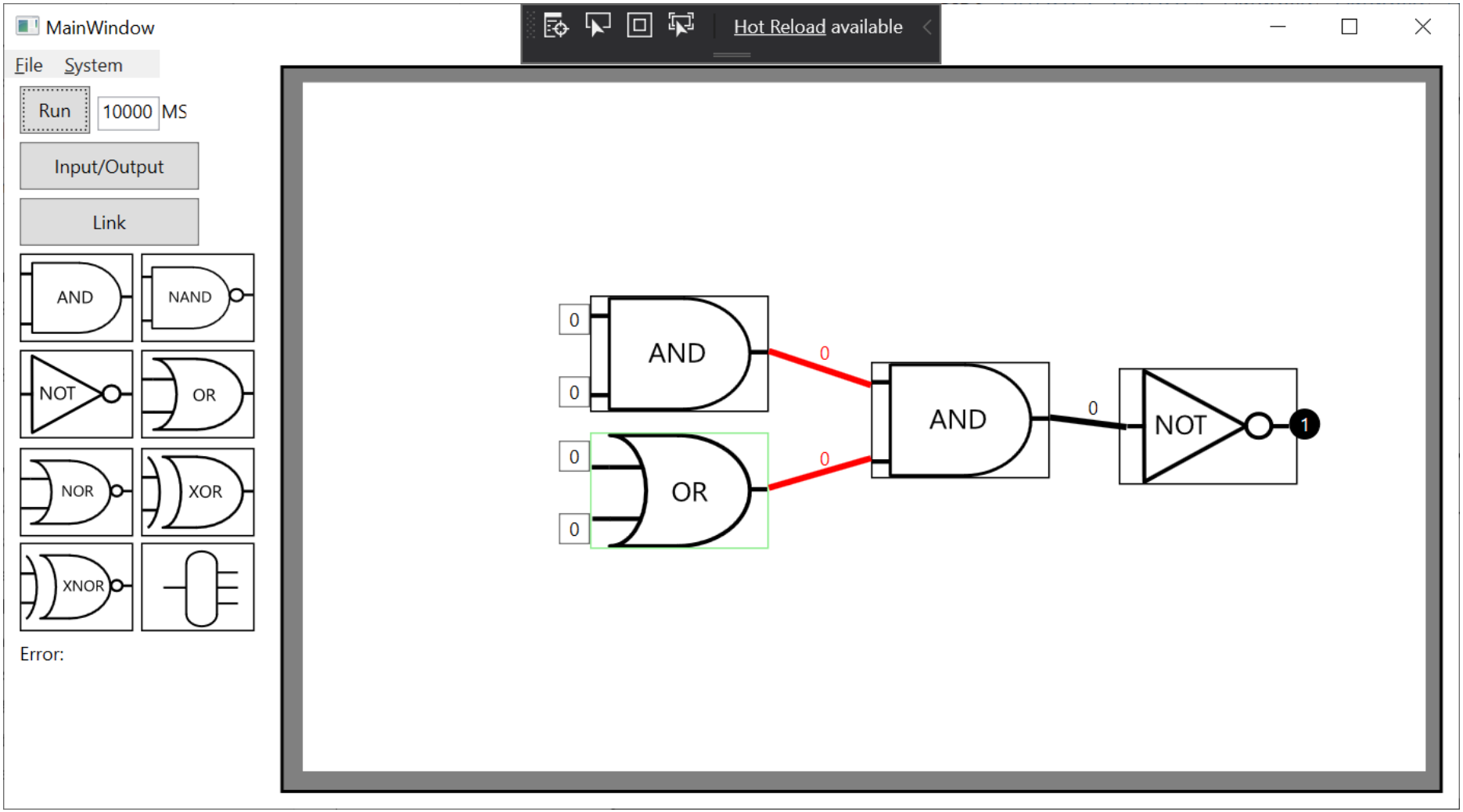

MainWindow
Hot Reload available
File System
Stop
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
OR
0
NOT
1
NAND
1
0
NOT

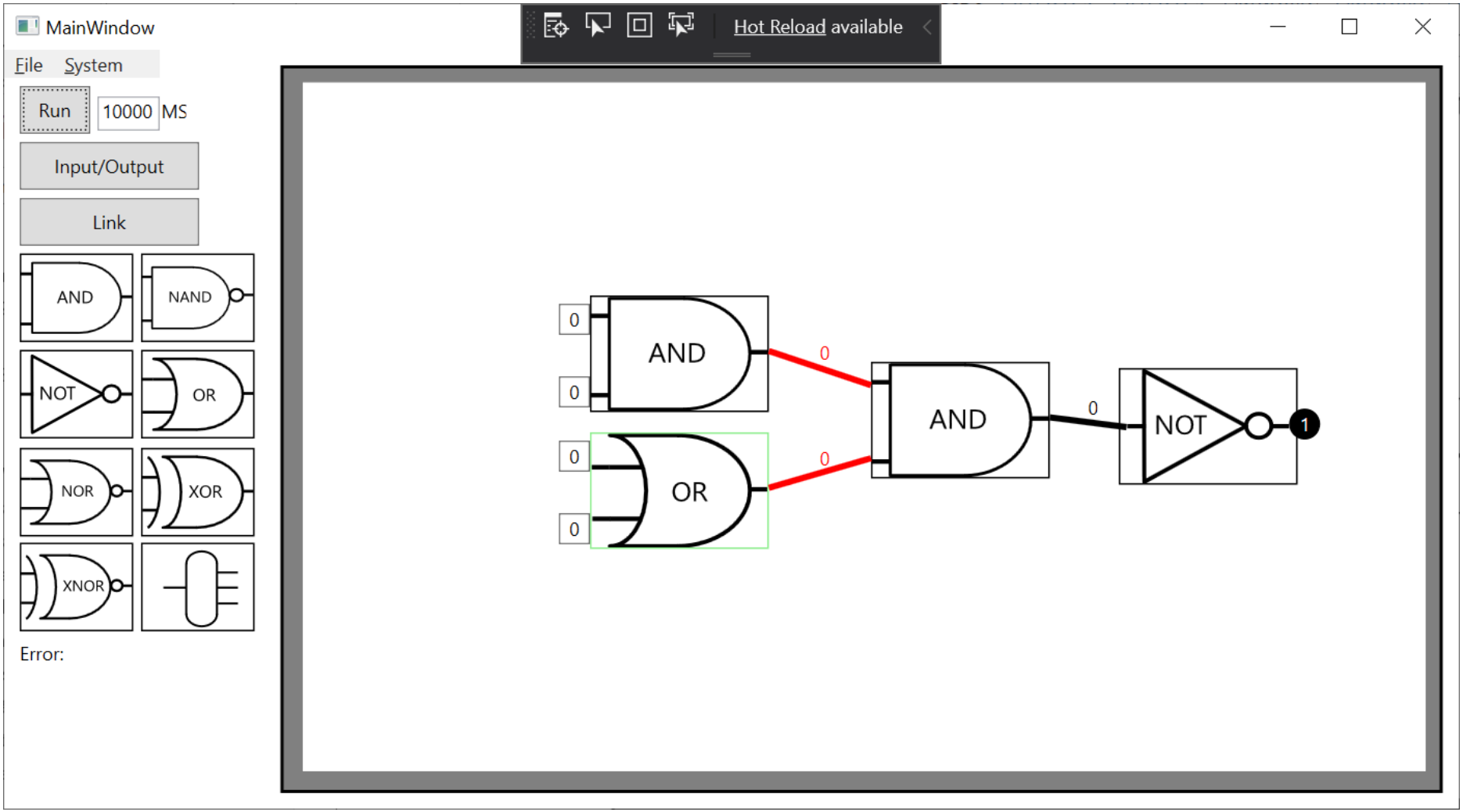

MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
OR
0
NOT
1
NAND
1
0
NOT


MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
OR
0
NOT
1
NAND
1
0
NOT

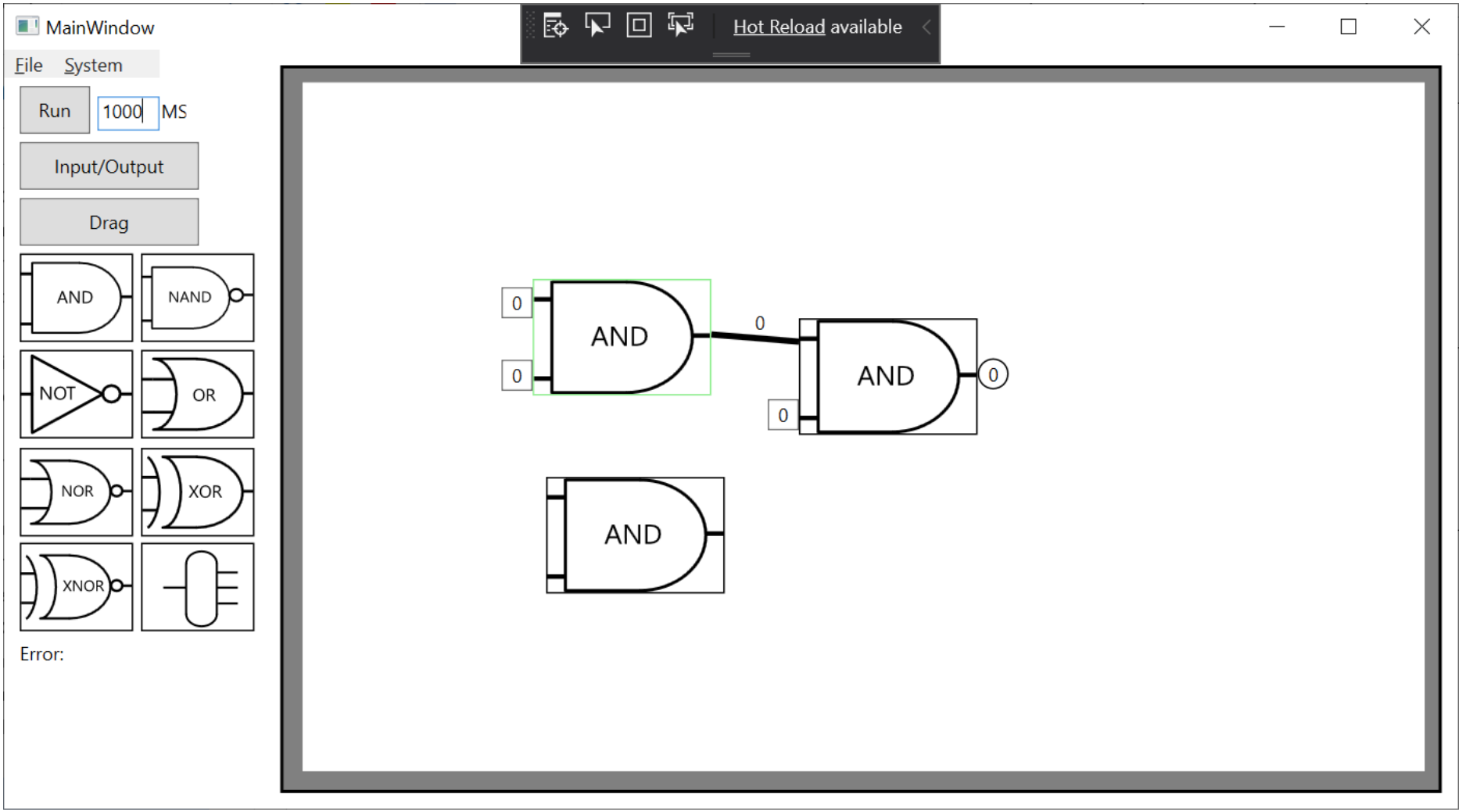
9G

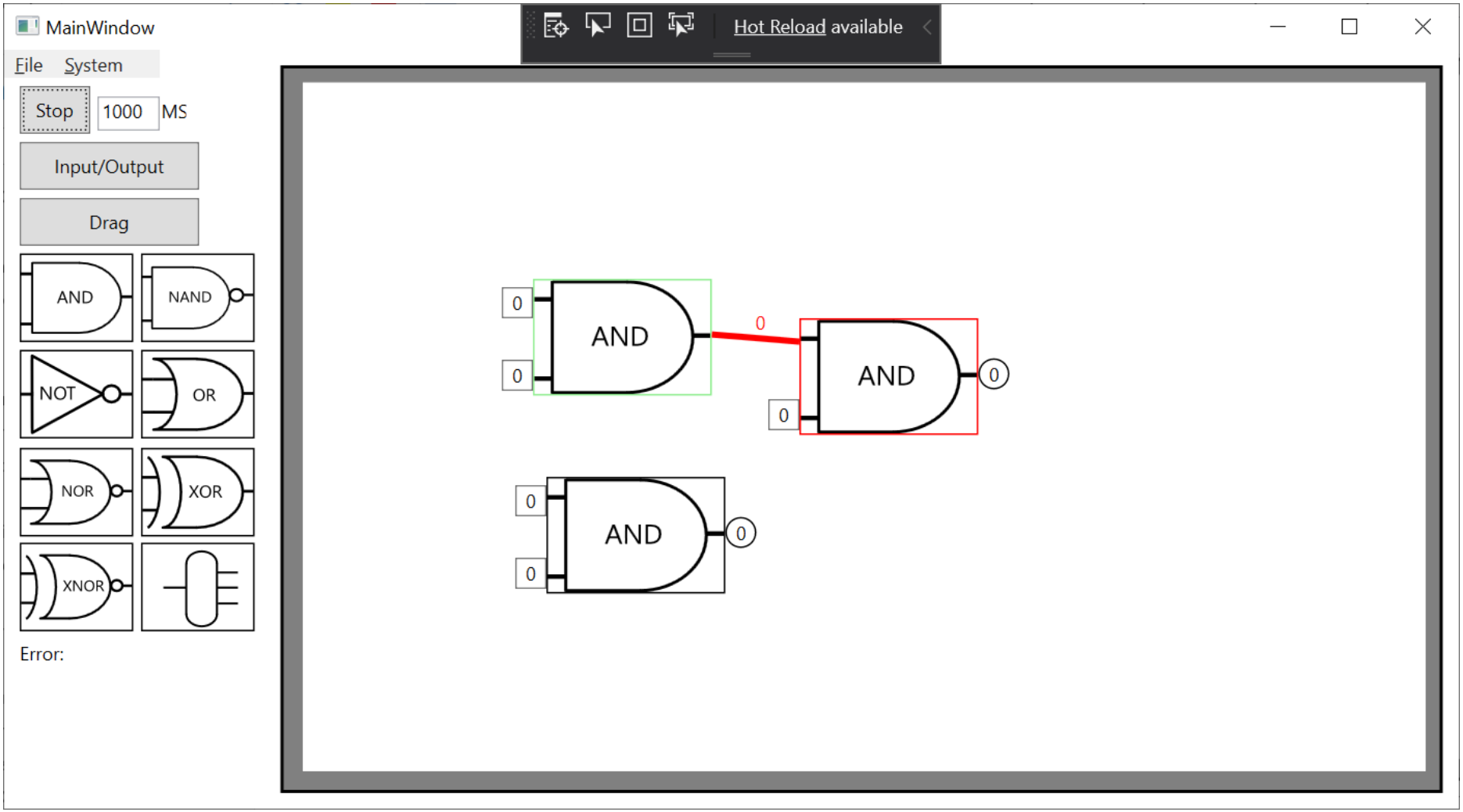

MainWindow
Hot Reload available
File
System
Stop
10000
MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
0
OR
0
AND
0
NOT
1

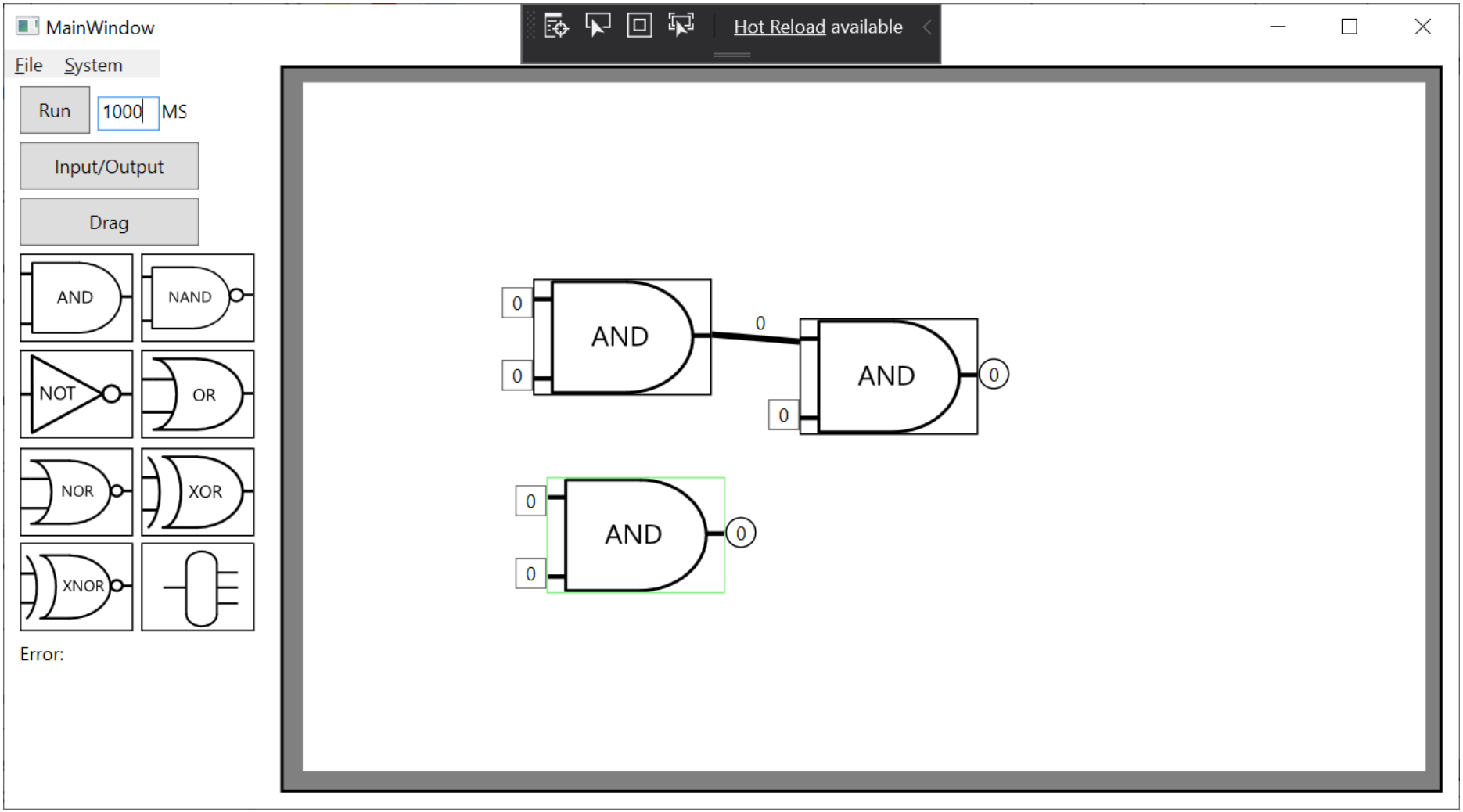

MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
0
OR
0
AND
0
NOT
1


MainWindow
Hot Reload available
File System
Run
10000 MS
Input/Output
Link
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
0
OR
0
AND
0
NOT
1

9I


MainWindow
Hot Reload available
File System
Run
1000
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
AND
0
0
AND


MainWindow
Hot Reload available
File System
Stop
1000
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
0
AND
0
0
AND
0
0
0
AND
0


MainWindow
Hot Reload available
File System
Run
1000
MS
Input/Output
Drag
AND
NAND
NOT
OR
NOR
XOR
XNOR
Error:
0
AND
0
0
AND
0
0
0
AND
0

Conclusion

I've completed every task that was set:

- Visible changes have been applied when the simulator is running and when the user puts their cursor over a gate.
- The speed of the simulator can be changed and remove the delay if they want.
- Files tools are fully available to the user and provide everything they will need.
- When the user changes an input in a gate it will update the system to accomidate for the new change.
- The subcanvas can be zoomed in and out to allow for custimizable UI.
- Objects are integrated well with the mouse events and movements.
- A fully working flip-flop cuircit can be created and simulated.

Things I could improve:

- Make the line align for each individual classes instead of a rough guide for their port setup (2 input & 1 output or 1 input & 1 output).
- When zooming in and out on the canvas due to how the canvas zooms on the mouse it will create a werid effect. This is because when the canvas zooms in on the mouse location the center of the canvas will now be there but the mouse doesn't move to the center. So each time you zoom in your also translating the canvas a ratio of mouse location minus distance from center of canvas.
- Work out why there are 2 threads with no name running in the background that is keeping the simulator open.
- Work out why the IsBusy is permently set to false when await is called.
- Make the input Button Class and output Circle Class inherit from the base Input Class and Output Class

I believe the finished outcome gives a professional look and functions fully without any bugs. It's clear when the simulator is running what is happening and where. This program will fulfil its goal of teaching kids logic cuircits

<u>Feedback</u>

Fellow students:

Sam Coulson:

"The program could be clearer if a help guide was added in. One output port should have multiple connections available."

Mo M:

"The lines should align up with the UI of the gate."