LOGIC CIRCUIT EMULATION ON PIC16F1939

Jan Ahmetspahić

Contents

1. Abstract

In the contents of this project, the task was to implement an emulator of behavior of basic logic circuits, for the microcontroller PIC16F1939. The logic circuits are inputed through the software Logisim, by drawing a logic diagram. Afterwards, based on an XML export from the aforementioned software and based on the code implemented as part of this project, a code is generated with the purpose of uploading to the microcontroller, which emulates the behavior of the logic circuit. For the input pins, port D is allocated, whereas for the output pins port B is allocated, which already has soldered LEDs on the development board. Therefore, at most 8 inputs and 8 outputs are supported. The number of components themselves is limited only by the program memory. The supported circuits are a voltage follower, NOT, AND, OR, NAND, NOR, XOR and XNOR, of which all except for the follower and NOT circuits (which have one input port) support up to 5 input ports.

All of the code included in this project, in Python, can be found on this github repository, as well as a guide for utilizing the project.

2. Introduction

Logic functions are functions which take only 2 possible values, with variables which also take only 2 possible values, and are defined by:

$y = y(x_1, x_2, \dots, x_n)$

In the expression, the values x_i represent logic variables, and y represents the value of the logic function. All of the aforementioned variables can only take values 0 and 1 (false or true). We most often define logic functions by a corresponding truth table, which for every possible combination of input variables gives the output value. On table 2.1. we can see common basic logic functions of 2 variables, given in table form.

		INPUTS		OUTPUTS					
		A	B	AND	NAND	OR	NOR	EXOR	EXNOR
NOT gate		0	0	0	1	0	1	0	1
A	$\overline{A}$	0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	0	1	0
1	0	1	1	1	0	1	0	0	1

table 2.1.

Many more complex logic expressions are obtained by combining basic logic functions. So, logic functions are also defined in the form of logic expressions. The common labels for the set of all logic functions of 2 variables are given in table 2.2..

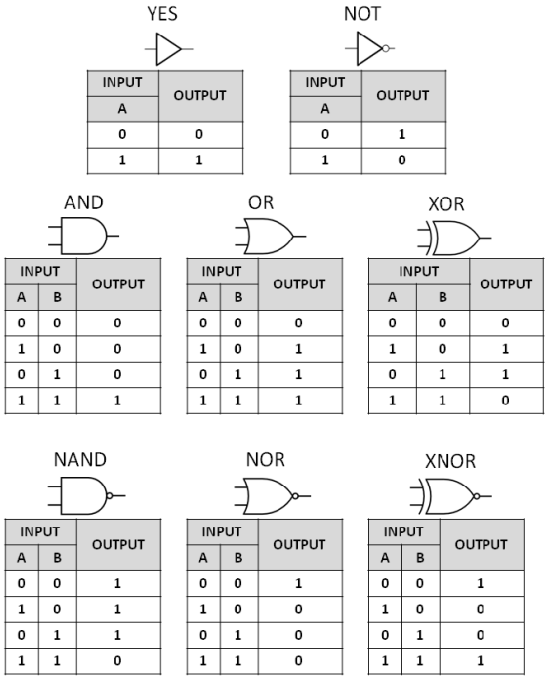
k	3	2	1	0	Naziv funkcije	Oznaka funkcije
x_1	1	1	0	0		
x_2	1	0	1	0		
y_0	0	0	0	0	Konstanta nula	$y_0 = 0$
y_1	0	0	0	1	Pierceova funkcija	$y_1 = x_1 \downarrow x_2$
y_2	0	0	1	0	Zabrana po x_1	$y_2 = x_2 \Delta x_1$
y_3	0	0	1	1	Inverzija x_1	$y_3 = \overline{x_1}$
y_4	0	1	0	0	Zabrana po x_2	$y_4 = x_1 \Delta x_2$
y_5	0	1	0	1	Inverzija x_2	$y_5 = \overline{x_2}$
y_6	0	1	1	0	Suma po modulu 2	$y_6 = x_1 \oplus x_2$
y_7	0	1	1	1	Shefferova funkcija	$y_7 = x_1 \mid x_2$
y_8	1	0	0	0	Konjunkcija	$y_8 = x_1 x_2$
y_9	1	0	0	1	Ekvivalencija	$y_9 = x_1 \sim x_2$
y_{10}	1	0	1	0	Promjenljiva x_2	$y_{10} = x_2$
y_{11}	1	0	1	1	Implikacija x_1 ka x_2	$y_{11} = x_1 \rightarrow x_2$
y_{12}	1	1	0	0	Promjenljiva x_1	$y_{12} = x_1$
y_{13}	1	1	0	1	Implikacija x_2 ka x_1	$y_{13} = x_2 \rightarrow x_1$
y_{14}	1	1	1	0	Disjunkcija	$y_{14} = x_1 \vee x_2$
y_{15}	1	1	1	1	Konstanta 1	$y_{15} = 1$

table 2.2.

An example of a logic expression, derived from the combination of basic logic circuits, is expression 2.1, which defines logic equivalency of 2 variables.

$$y = A \leftrightarrow B = (A \wedge B) \vee (\neg A \wedge \neg B) \quad (2.1.)$$

Additional simplification can be introduced (e.g. logic AND can be represented by a multiplication symbol, and can therefore also be omitted). But still, the truth table and logic expressions can be appropriate representations in certain use cases, for others they can be highly tedious. In those cases, there is a tendency to represent logic functions in the form of logic diagrams. A logic diagram is a representation of a logic function or a set of logic functions, shown graphically. Symbols used, for the aforementioned basic logic functions (with 2 inputs), are displayed on image 2.1.



YES	
INPUT A	OUTPUT
0	0
1	1

NOT	
INPUT A	OUTPUT
0	1
1	0

AND		
INPUT A	INPUT B	OUTPUT
0	0	0
1	0	0
0	1	0
1	1	1

OR		
INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	1

XOR		
INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	0

NAND		
INPUT A	INPUT B	OUTPUT
0	0	1
1	0	1
0	1	1
1	1	0

NOR		
INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	0

XNOR		
INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	1

image 2.1.

Anything connected to the input pins represents an input variable in the corresponding logic function, and the value of the logic function goes out from the output port of the logic gate (which is then further connected to other logic gates). This type of representation can be more readable and effective for engineering applications, so it's often used in those cases. For example, an equivalent logic diagram for XNOR (logic equivalency), using negation, AND and OR circuits, is shown on image 2.2.

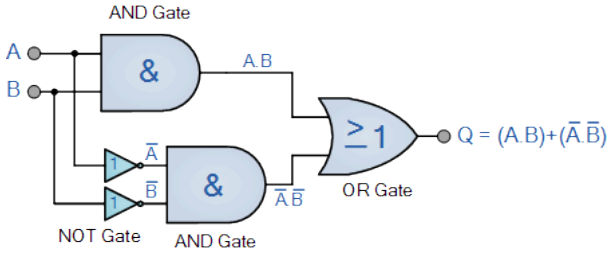


image 2.2.

Our task is to implement an appropriate electronic implementation of the logic functionality, using the microcontroller PIC16F1939. Based on the given logic circuits we have to program the microcontroller to exhibit the same behavior.

In the electronic solution, which is most relevant for practical problems, the logic inputs and outputs are implemented using electric units (mostly voltage). Logic circuits are still mostly shown in the form of a logic diagram, as it's easy to go from a logic diagram to a logic circuit and backwards.

In logic circuits, logic 1 is defined by a certain voltage level, and logic 0 is defined by a different voltage level (the voltage level for 1 is generally higher). In the ideal case, logic signals should only take these 2 values, but in reality this is not the case. Therefore, usually a cutoff voltage is defined or imposed by a specific technology, over which the voltage level is treated as being logic one, and another cutoff voltage is defined under which the signal is treated as being logic zero.

There are many different technologies in practice which implement logic circuits. We are only interested in the fact that the same principles of voltage level and cutoffs, also hold for the PIC16F1939 microcontroller. In fact, it's digital inputs and outputs are basically logic inputs and outputs, which represent logic one by 5 V, and logic zero by 0 V, whereby there is a certain rounding process for all values between the 2 extremes (for all real values). So, by using these inputs and outputs we can emulate, by using internal programming logic, the behavior of logic circuits.

Motivation

The behavior of many practical systems can be described by relatively simple logic circuits. Of course, in practice, it is often necessary to do specific adjustments which depend on the electronics in use, sensors, and so on. Be that as it may, for basic functionality of systems it's sufficient to use circuits synthetized from a group of simple logic circuits. For example, a system which opens and closes moving doors, a system which activates and deactivates a pump, any kind of switching regulation, etc.

Instead of programming the specific logic, every time, there is a motivation here to automatize the process. Many engineers would rather design a logic diagram to generate a certain logic functionality, rather than implement a code, as the process of drawing a diagram can be faster and more readable. That's why there are existing boards and corresponding software, which offer this capability. However, it's possible that some of those boards aren't available or that the system has additional functionalities which need to be implemented, aside from the logic. In cases like these, software written for a specific microcontroller can be useful. Of course, the implementation of such software is also interesting as a project.

Even more so, the implementation of logic diagrams on the given development system, gives a very robust and flexible implementation. Of course, we are using a much more powerful tool than is strictly necessary, for the given applications, but it could happen in practice that we need to put all the logic onto one controller. In those cases, this implementation could be appropriate.

Project description

In this project, logic diagrams are only made up of basic logic gates (follower, NOT, AND, OR, NAND, NOR, XOR, XNOR). Also, feedback loops are not allowed in the realization of the diagram, which would present additional difficulties, bringing in the need for synchronization. All logic circuits are defined by the user in the Logisim development environment. Logisim is a simple and light software for drawing logic diagrams, based on Java (which means it works on basically everything). Logisim saves the logic diagrams in ".circ" files, which are none other than a modified version of an XML file.

Not long ago, we listed all the advantages of a schematic approach of displaying logic functions or systems of logic functions. However, from a coding point of view, the most appropriate representation of logic functions are logic expressions. Therefore, our main task is to take a logic diagram and transform it back to a form of a logic expression or set of logic expressions.

The project is implemented in Python. The user needs to take the logic diagram and save it in the same folder as the project (main.py) under the name "logic_circuit.circ". It's necessary for all the inputs and outputs to be labeled appropriately, such that RD0 to RD7 are used for inputs, and RB0 to RB7 for outputs (only ports B and D are in use). After running main.py, a C file will automatically be generated (main.c), which needs to be compiled and uploaded to the microcontroller, using appropriate software (MPLAB, PICPGM).

In essence, most of the C code is already written (in the file default_code.c). Only a part of the code is generated when the project is run, and that's the part which implements the functionality of the logic functions (a file main.c is generated).

3. Algorithm

First, we'll describe the generating algorithm in a few short points, from a high level. The assumptions for running the algorithm are an appropriate logic circuit, defined in a ".circ" file and a "skeleton" code which contains in it basic declarations and basic functionality: the implementation of basic logic functions, the declarations of input and output ports, configuration parameters for the microcontroller.

The skeleton code is used in every instance which is sent to the microcontroller, whereby only the implementation of a single function in the code actually changes. The output from the algorithm is exactly that modified code. The flow chart on a high level is shown on image 3.1.

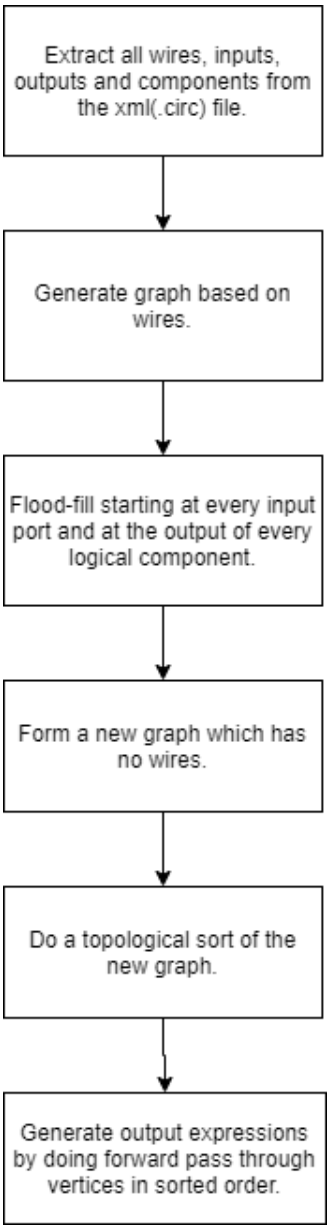


image 3.1.

4. Introduction to graphs

Some of the steps of the algorithm can be reduced to graph processing, so we first have to make an introduction to graphs and the basic concepts surrounding graphs. First, we will describe the intuitive meaning of a graph. A graph is a structure which is made up of a set of objects, which we call nodes or vertices, and fixed connections between those objects, which we call edges or arcs. Both nodes and edges can be basically anything. A typical example would be crossroads and roads, where the crossroads are represented by the nodes, and the roads are the edges which connect the crossroads (nodes). Another typical example would be a computer network, in which the routers are the nodes, and the cables which connect them are the edges. An example of a graph, displayed in a visual format, is shown on image 4.1.

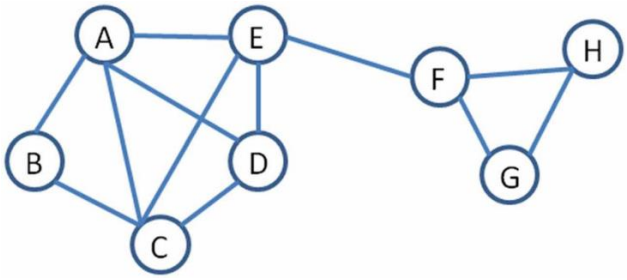


image 4.1.

The nodes are labeled with circles, the letters being their labels, whereas the edges are represented by lines which connect them. We can immediately list some plausible properties and/or limitations associated with edges. For example, edges could be directional (e.g. a one-directional road), they can have a corresponding length (length of a road, flow capacity), weight (density of traffic), etc.

Formally, there are many definitions of a graph, of which the following is one of the most common and simple – a graph is an ordered pair (V, E), where V is the set of nodes, and E is a binary relation defined over the set of nodes V. A directed graph is a graph in which the given relation E is antisymmetric. Ergo, if a pair of nodes are connected by an edge (u, v), then they musn't be connected by an edge (v, u). This coincides with the intuition of directed edges as one-directional roads – they are passable only in one direction, where as in the other they may as well not exist. The visual representation of a directed graph can be seen on image 4.2.

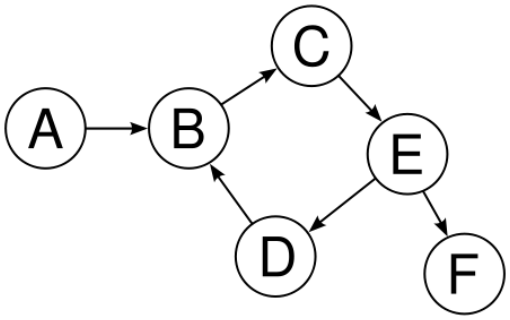


image 4.2.

An undirected (bidirectional) graph is a graph in which the binary relation E is symmetric, or in other words, intuitively, a graph where all the roads are bidirectional. A visual representation of a graph of this type was already shown on image 4.1. If the graph contains combinations of bidirectional and one-directional edges, we are describing a generalization of the notion of a graph, which is called a digraph.

Vertices u, v in an undirected graph, which are connected by edge $e = (u, v)$, are called neighboring nodes or just neighbors, and the edge e, by which they are connected, is is said to be incident to the vertices. For example, on image 4.1. nodes A and B are neighbors, as are nodes F and H. In a directed graph, the notion of neighbors are defined a bit differently. A node v is said to be a neighbor of node u, if there exists an edge $e = (u, v) \in E$. As the binary relation E is antisymmetric for directed graphs, the reverse doesn't hold, or in other words node u is not a neighbor of node v. For example, on image 4.2. B is a neighbor of A, but A is not a neighbor of B.

A path in the graph (V, E) starting at node u and ending at node v, is defined as the alternating set of nodes and edges $(v_1, e_1, v_2, e_2, \dots, v_{k-1}, e_{k-1}, v_k)$, whereby for every k, it holds that $e_k = (v_k, v_{k+1}) \in E$. This is a very intuitive definition, and it's equivalent to the concept of an alternating sequence of crossroads and roads, which lead from one crossroad to another. On image 4.3. a single path is labeled through an undirected graph, from node a to node h.

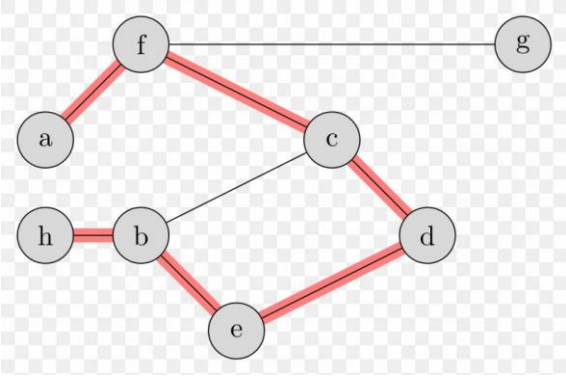


image 4.3.

A cycle in a graph is defined as a path which has the same starting and ending nodes. On image 4.4. we can see an example of a cycle in the same graph as the previous image.

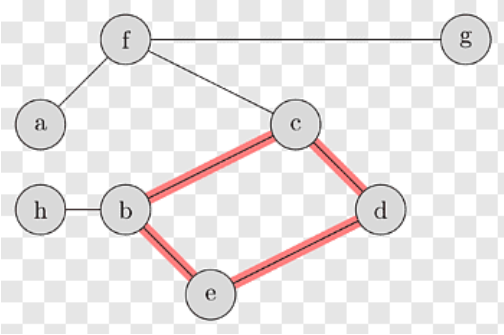
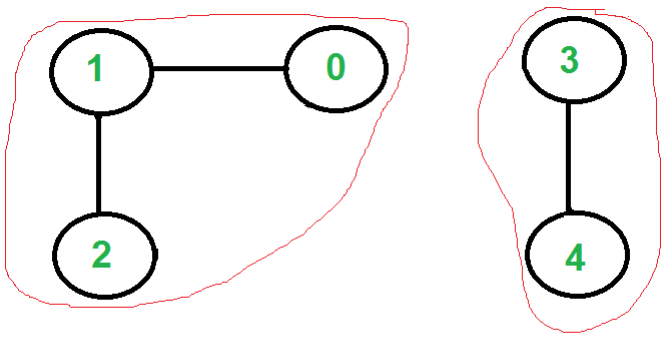


image 4.4.

Now that we have defined the notion of a path, we will define the notion of a connected component. Let's say we have an undirected graph (V, E). We can define a connected component in the graph as an equivalency relation C_i, for which it holds that for all pairs of nodes $(u, v) \in C_i \times C_i$, there exists a path which leads from u to v. It clearly follows that if there exists a path from u to v, then there also exists a path from v to u, being that the graph is undirected. Because of this fact, we can partition the graph into a set of, pairwise disconnected, connected components. Again, intuitively, in the component in which some node u is located, all the other nodes are reachable from u by moving through the graph, starting at u. A partitioning of an undirected graph into connected components is shown on image 4.5.



There are two connected components in above undirected graph
0 1 2
3 4

image 4.5.

Now that we have all the necessary definitions for the following part, we will describe an algorithm which we can utilize to find all connected components of an undirected graph. First, we need to introduce a graph search algorithm, which will allow us to find all nodes connected to a certain starting node. The algorithm we will describe for this purpose is called depth first search (DFS). The steps this algorithm takes are the following:

1. In the beginning, all the nodes in the graph are unmarked, except the starting node.
2. Call the procedure visit, applied to the starting node.

Within the procedure visit:

1. Let's say the procedure is called with node u as it's argument.
2. Mark node u as visited.
3. If u has an unmarked neighbor v, recursively call the procedure visit, passing node v as the argument.

This algorithm has an extremely simple implementation and the pseudocode is shown on 4.1.

```
function dfs(u)
      visited(u) := true
      while node u has unmarked neighbor v:
            dfs(v)

for all nodes u:
      visited(u) := false
dfs(start_node)
```

pseudocode 4.1.

The algorithm is very intuitive. For a node u, which has an unmarked neighbor v, we visit everything we can from the neighbor v, because we can then visit all that from node u, as well (because v is a neighbor, all we have to do is go from node u to v, and visit everything possible from v). Of course, within the algorithm, we never return to a previously marked node, because that makes no sense (we won't be able to explore anything that we haven't already). This, in turn, also makes sure that we don't end up in a cycle during the traversal. If we recursively apply the described procedure, after the algorithm terminates, all the nodes in the same component as the start node will now be marked.

The behavior of the DFS algorithm is best demonstrated on video or as part of a presentation, because that's a good way to demonstrate the order in which the nodes are visited. As this is a written document, we will have to show that behavior on an image (4.6.).

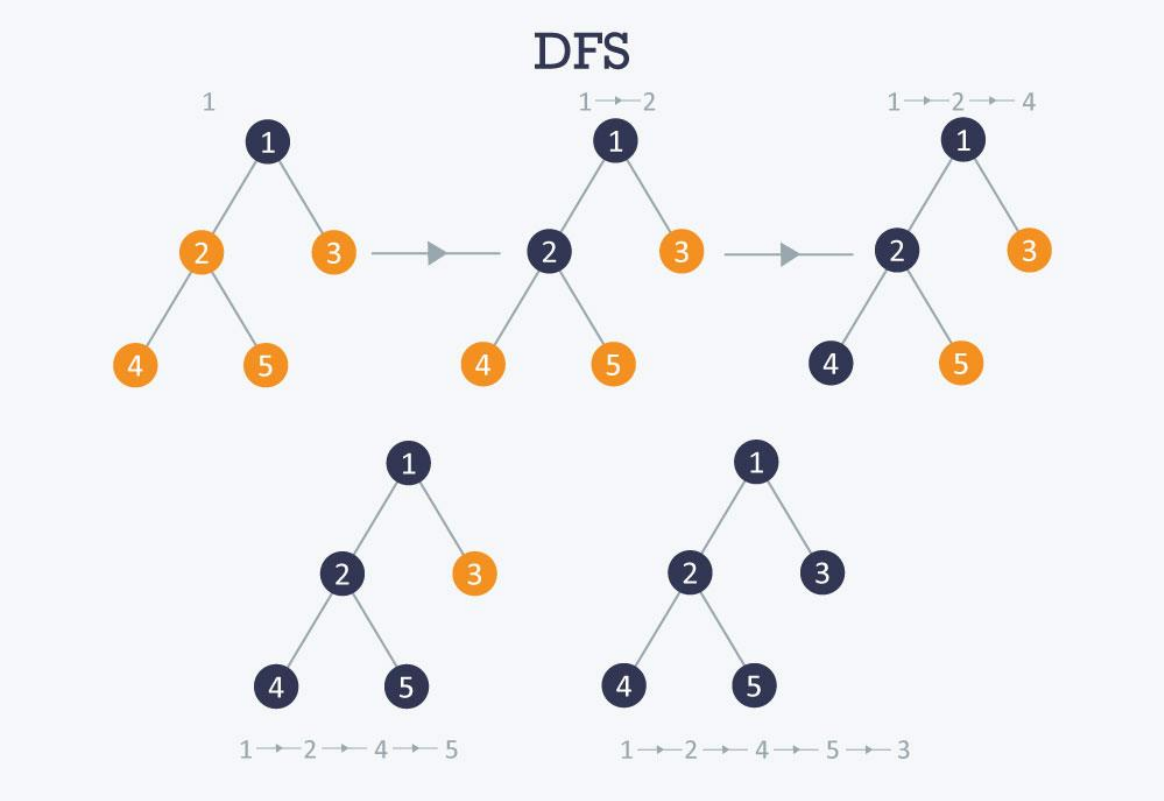


image 4.6.

The unmarked nodes are colored orange, and the marked nodes are colored black. Notice that, even if there had been (for example) an edge between nodes 1 and 5, node 5 wouldn't have been visited for a second time, because DFS never returns to nodes which have already been marked.

Now that we can, for each node, find all other nodes which lie in the same connected component, it's easy to find and label all the connected components. The following algorithm is usually referred to as Flood-fill, and the pseudocode is shown on 4.2.

```
function dfs(u,L)
      label(u) := L
      while there is an unmarked neighbor v:
            dfs(v,L)

for all nodes u:
      label(u) = null
while there is an unmarked node u:
      dfs(u,u)
```

pseudocode 4.2.

The algorithm isn't much different compared to DFS. Now, instead of labeling the nodes using true and false, we mark them using labels. At the start, all nodes are unmarked. Then, when we find an unmarked node, we will apply DFS with that node as the starting node, setting the labels of all nodes in the corresponding connected component to the index of that starting node. Considering that all the connected components are disconnected amongst one another, at the end it's guaranteed that every component has a unique index which is the index of one of the nodes inside the component.

We can go back to example from image 4.5. At the start, the nodes are unmarked. The unmarked node with the smallest index is node 0, so let's start the search from it, even though it pretty much doesn't matter which of the unmarked nodes we choose. After we make the call dfs(0), we will visit all the nodes which are in the same component as 0, and we're going to apply the label 0 to all of them. Therefore, now the nodes 0, 1 and 2 will all be labeled 0, and nodes 3 and 4 are still going to be unmarked. The next unmarked node with minimal index is 3, so by calling dfs(3) we will visit 3 and 4, which both get the label 3. Now that we have visited all the vertices of the graph, the algorithm terminates.

As can be seen, we have 2 components, one of which has the label 0, while the other has the label 3. It's common to use the ordinal number of the component as the label, instead of the index of the node from which the search started. In other words, when we first call the function dfs, we pass 1 as the label, the second time we pass 2, and so on. It doesn't really matter how we label the components if it holds that each component has a unique label.

5. Forming the graph

After performing a rather straightforward but tedious processing of the xml file, we will extract the following information:

- Wires – the start and end positions.
- Pins – if it's input or output and the position.
- Gates – type of logic gate and position.

First, it's important to describe how the xml file is generated, based on the logic diagram. Logisim treats it's surface as a grid with coordinates, so all components have a corresponding position in the grid. Wires have defined start and end coordinates. Pins are simple, because their contact is positioned in exactly the same place as they are.

Regarding the logic gates, their locations are actually the coordinates of their outputs, while their inputs are a fixed distance away west of the output. This fixed distance depends on the gate. For one-input gates (NOT and follower), their input is on the same height level as the output. For the multiple-input gates (all of which are up to 5 inputs), their input ports are, in comparison to the height level of the output port: 20 units above, 10 units above, leveled, 10 units bellow and 20 units bellow, respectively.

This all looks pretty tedious and complicated, but this actually **helps** with generating the graph. If we make a graph over the positions as nodes, and we treat wires as edges between the nodes, then the only wire which is connected with the location of any component is the output wire. All the wires connected to the input pins are a fixed distance west of the component location. We also know that there are no feedback loops, and that the input pins can't cross each other, or in other words that we can't connect wires exiting from 2 different input pins to the same point.

This means that we can partition the graph into components, where there is exactly one input pin or logic gate output corresponding to each component. Based on this information, we can label all the connected components, applying Flood-fill on the graph, starting in all the input pins and logic gate outputs. This is guaranteed to cover the whole graph.

For every relevant coordinate, where a coordinate is relevant if it's one of the endpoints of a wire or if it's a location of some gate or pin, we now have the information about which connected component it belongs to.

Wireless graph

Based on this, we can construct a graph with no wires. For every logic gate and output pin, we generate all the coordinates which correspond to their inputs – this depends on the component, and it's necessary to "hard-code" all the cases. If any of these coordinates happens to be irrelevant, we can conclude that they aren't connected to anything in the graph, so therefore there is no input signal on them. In this case, we treat that input pin as nonexistent, and view the logic gate as having lesser input dimensionality.

For all relevant coordinates, we see to which input pin or logic gate output they are connected, and we know that exactly which signal made it to that input port. Based on this information, we can form a directed graph, in which the nodes are logic gates and pins. The graph is certainly directed and acyclic, as there are no feedback loops and as there is a clear direction imposed. The direction is defined by propagating logic variables through the graph, starting from the inputs and towards the outputs.

From this point on, we will apply operations to this graph, because it gives us the ability to directly observe the behavior of the logic function. Furthermore, if we wanted, we could directly obtain the value of the logic function from the graph – by fixing the input values, and then propagating the values through the graph, from input to output. This would require a recursive implementation, which isn't very suited for use on the microcontroller (which has a call stack of size 15), so we have to look for an alternative (iterative) approach.

6. Topological sort

In order to press forward with the explanation of the steps of the algorithm, we first need to introduce the notion of a topological sort of a graph. Topological sorting only has meaning in the case where the graph is directional. The intuitive sense behind it is easy to present with the following setup:

Let's say a factory is producing some kind of product, which is made out of many parts. Some of these parts are independent, while others are composed out of previous parts or require that some of the previous parts be completed for the process of their production to begin. In the example of car production, cutting sheet metal might be independent in relation to other processes, but the installation of doors or tires requires for certain parts to already be completed (e.g. the skeleton of the car has been constructed).

For every process, we can establish relations of the kind: "this process can only begin after that process ends". The task now becomes to order the processes, so that a process never begins whose preconditions are not met. In all real situations, this ordering is going to exist, because otherwise it wouldn't be possible to produce the product. Let's now express this problem in the language of graph theory.

We're going to represent processes by nodes, and relations of the type "v can only begin after u ends", we're going to represent by directed edges (u, v). Now, the problem reduces to the following: Let's say we have a directed graph (V, E). We need to find an ordering of the nodes $(v_i) = (v_1, v_2, \dots, v_n)$, such that for every $(u, v) \in E$, it holds that v is a predecessor of u in the ordering (v_i).

It's not hard to prove that this ordering exists, if and only if, there are no cycles in the directed graph. If a cycle exists, every node in the cycle, imply a condition to one another in a circle. This is a problem, as none of the nodes can be put in front of any others. If the graph is acyclic and directed, then the topological sort certainly exists.

An example of a topologically sorted directed graph is given on image 6.1.

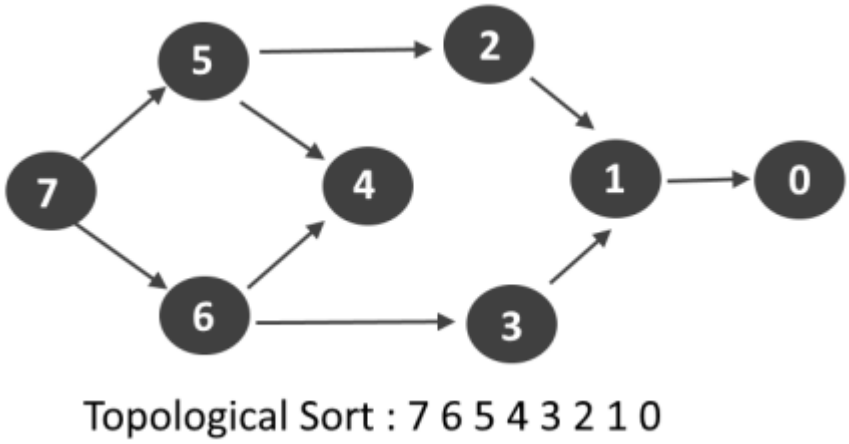


image 6.1.

We can see, that for every edge in the graph, the second node (the sink) is always positioned after the first node (the source), in the topological ordering. Therefore, this topological ordering is valid.

There are several algorithms for generating a topological ordering, but here we will describe the one which is probably conceptually easiest, and that's Kahn's algorithm. This method is made up of the following steps:

1. At the beginning, the list (v_i) of topologically sorted nodes is empty, and all nodes are unmarked.
2. We choose any unmarked node, which has no ingoing edges. We put that node at the end of the list (v_i), and afterwards we remove all edges which exit it, and finally we mark it. Afterwards, we return to step 2. If there is no node of this kind, we go to step 3.
3. If all the nodes are marked, we terminate the algorithm, and get the solution from the list (v_i).
4. If some of the nodes are unmarked, then all unmarked nodes have ingoing edges. We conclude that the graph has a cycle and that a topological ordering of the graph doesn't exist.

This algorithm is very intuitive. When it occurs that all ingoing edges of some node are removed – all nodes v, which imply a condition on node u (edge (v, u) exists in the starting graph), are already in the list of topologically sorted nodes. It follows that we can freely place u as the next node of the list, which can only help the following nodes (by removing the outgoing edges of node u).

We can observe, that this algorithm can give different solutions for the same problem. This can occur, as it is possible that there exist several valid topological orderings of the nodes in the graph. We don't care which ordering is obtained, out of all the possible topological orderings. We only care that it satisfies the conditions that every topological ordering satisfies.

7. Forming expressions

When we've formed the graph over the logic gates and pins, it's not hard to verify that it'll be directed and acyclic. It is directed because the output of every logic gate is defined by the values on it's input, which means that the input values have to be defined before the logic gate value is calculated. Graph is acyclic because there are no feedback loops in the diagram.

Under these conditions, it's clear that we can topologically sort the graph, so we will do so. After we obtain the topological ordering, we can be sure that the inputs of the logic gate or output pin, depend only on the output values of logic gates or input pins, which are preceding in the topological ordering of the graph. This means that we can do a forward pass through the topological ordering of the graph, and for every logic gate or pin u, we can compute the logic expression, based on the **previously computed** logic expressions, for logic gates or pins v.

This makes it possible to obtain a non-recursive approach, which in the end generates logic expressions for the output signal for every logic gate and output pin. The values at the output pins are exactly the ones we care about, so it's sufficient to place them into the function which implements the logic (with some additional processing to obtain a valid C code), and we are done.

The method of computing these expressions, can be represented by the following steps:

1. We do a forward pass through the topological ordering of the graph nodes. Let's say the currently observed node is u.
2. For all nodes v_i whose edges enter into u, we know the value $expression(v_i)$.
3. If node u is an output pin, then it has to have only one ingoing edge, as connecting multiple signals to the same point makes no sense. In this case $expression(u) = expression(v_1)$.
4. If node u is a logic gate, it could have one more inputs. Let l_u be the label of logic gate u (NOT, AND, OR, ...). Let also $(v_1, v_2, \dots, v_k)$ $(k \leq 5)$ be the active inputs of node u. Then, the following relation holds: $expression(u) = l_u(l_u(l_u(\dots), expression(v_{k-1})), expression(v_k))$. As we have already obtained all of the values $expression(v_i)$, we can now compute the value $expression(u)$.

The labels l_u are defined in the code, and they represent the names of the functions, which take 2 arguments, and mirror the behavior of one of the basic logic functions, which in this case are Buffer (follower), NOT, AND, OR, NAND, NOR, XOR and XNOR. Clearly, we have implicitly assumed that all the elementary 2-port operators are left associative, and that a multiple-port operator is just the 2-port operator applied under the rules of left associativity. This assumption is common for most operators, but should be kept in mind.

8. Simulation and testing

Testing

Let's now show what happens, when we apply the code to several different logic diagrams, in order to determine if the results really coincide with the expectations. We will list five examples, which test different possibilities. We will show only the part of the code which is actually generated, or in other words the function which emulates the behavior of the logic diagram. The first diagram is shown on image 8.1.

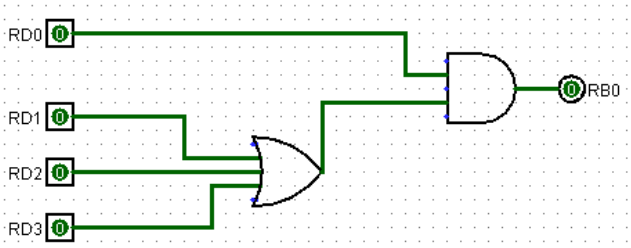


image 8.1.

After running the code for this diagram, we obtain the result shown in code snippet 8.1.

```
void logic_function(void) {
   PORTBbits.RB0 =
AND(PORTDbits.RD0,OR(OR(PORTDbits.RD1,PORTDbits.RD2),PORTDbits.RD3));
}
```

code snippet 8.1.

This is equivalent to logic expression 8.1.

$$y = RD0 \wedge \big((RD1 \vee RD2) \vee RD3\big) = RD0(RD1 \vee RD2 \vee RD3) \quad (8.1.)$$

It's not difficult to see, that the logic expression is equivalent to the given diagram. With this example, we have seen that the multi-input single-output behavior checks out. Also, the behavior of multi-input operators was also correctly emulated, under the assumption of left associativity of the operators.

So let's move on to the next example, shown on image 8.2.

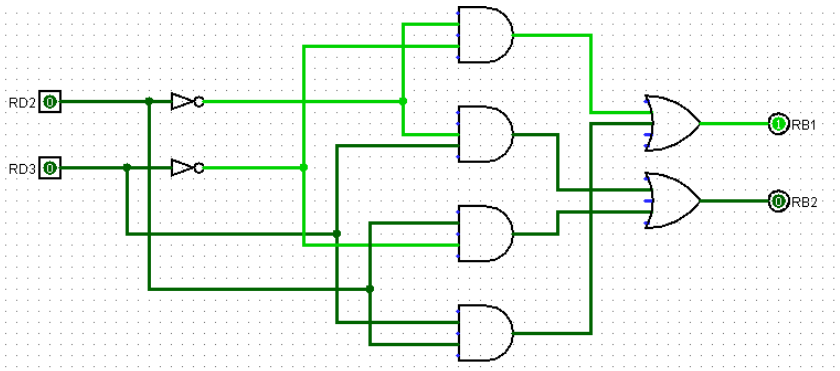


image 8.2.

Here it's somewhat harder to see which functionality is exhibited. This is actually a realization of XNOR and XOR using AND, OR and NOT. At pin RB1 XNOR is generated, while on pin RB2 XOR is generated. The equivalent system of logic expressions is given by expressions 8.2. and 8.3.

$$y(RB1) = y_1 = (\neg RD2 \wedge \neg RD3) \vee (RD2 \vee RD3) \quad (8.2.)$$

$$y(RB2) = y_2 = (\neg RD2 \wedge RD3) \vee (RD2 \vee \neg RD3) \quad (8.3.)$$

The code, generated for this example, is given in code snippet 8.2.

```
void logic_function(void) {
   PORTBbits.RB2 =
OR(AND(NOT(PORTDbits.RD2),PORTDbits.RD3),AND(PORTDbits.RD2,NOT(PORTDbits.RD3)));
   PORTBbits.RB1 =
OR(AND(NOT(PORTDbits.RD2),NOT(PORTDbits.RD3)),AND(PORTDbits.RD3,PORTDbits.RD2));
}
```

code snippet 8.2.

We can observe that the code is equivalent to the logic expressions. With this example, we have tested the functionality of multi-output diagrams. The diagram for example 3 is shown on image 8.3.

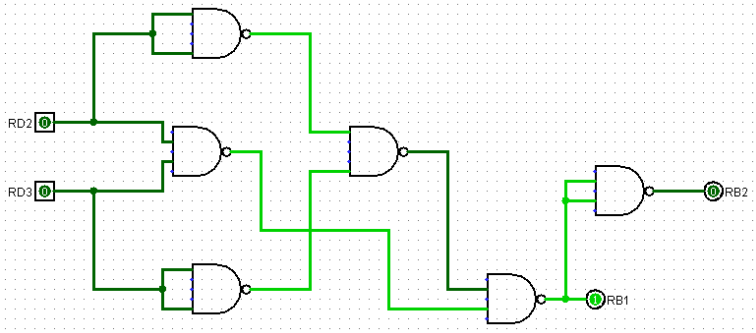


image 8.3.

This is, in essence, the realization of the same functionality as the previous diagram, only instead implemented using just NAND blocks. The equivalent system of expressions for this case is given by 8.4. and 8.5.

$$y_1 = (RD2 \uparrow RD3) \uparrow \big((RD2 \uparrow RD2) \uparrow (RD3 \uparrow RD3)\big) \quad (8.4.)$$

$$y_2 = y_1 \uparrow y_1 \quad (8.5.)$$

The code generated for this example, is given in snippet 8.3.

```
void logic_function(void) {
   PORTBbits.RB2 =
NAND(NAND(NAND(NAND(PORTDbits.RD2,PORTDbits.RD2),NAND(PORTDbits.RD3,PORTDbits.RD3)),N
AND(PORTDbits.RD2,PORTDbits.RD3)),NAND(NAND(NAND(PORTDbits.RD2,PORTDbits.RD2),NAND(PO
RTDbits.RD3,PORTDbits.RD3)),NAND(PORTDbits.RD2,PORTDbits.RD3)));
   PORTBbits.RB1 =
NAND(NAND(NAND(PORTDbits.RD2,PORTDbits.RD2),NAND(PORTDbits.RD3,PORTDbits.RD3)),NAND(P
ORTDbits.RD2,PORTDbits.RD3));
}
```

code snippet 8.3.

It's somewhat hard to verify because of the length, but we can see that the first part is actually NAND with both operands being the second part. We can also see that the second part implements the correct behavior. Example 4 is given on image 8.4.

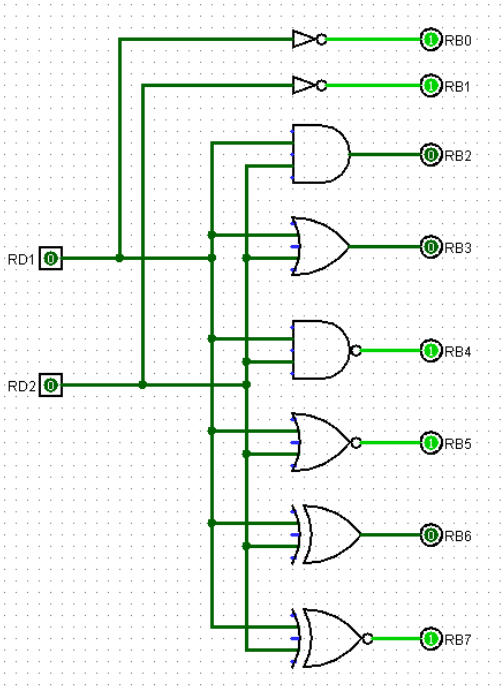


image 8.4.

We can see that this circuit in fact tests all relevant two-input operators and NOT. If this test passes, we have good reason to believe that the logic of connecting logic gates with their inputs is correct, and that all logic gates are correctly implemented in the code. There is no need to write the system of expressions for this example. The code for this example is given in snippet 8.4.

```
void logic_function(void) {
   PORTBbits.RB4 = NAND(PORTDbits.RD1,PORTDbits.RD2);
   PORTBbits.RB0 = NOT(PORTDbits.RD1);
   PORTBbits.RB6 = XOR(PORTDbits.RD1,PORTDbits.RD2);
   PORTBbits.RB7 = XNOR(PORTDbits.RD1,PORTDbits.RD2);
   PORTBbits.RB1 = NOT(PORTDbits.RD2);
   PORTBbits.RB5 = NOR(PORTDbits.RD1,PORTDbits.RD2);
   PORTBbits.RB2 = AND(PORTDbits.RD1,PORTDbits.RD2);
   PORTBbits.RB3 = OR(PORTDbits.RD1,PORTDbits.RD2);
}
```

code snippet 8.4.

We can see that all of the outputs are correctly realized, in the correct order. This means that the basic functionalities are satisfied for every possible logic gate. Let's test one more final example, which is a somewhat more complex diagram – so that we can observe the behavior in those kinds of cases. The diagram is given on image 8.5.

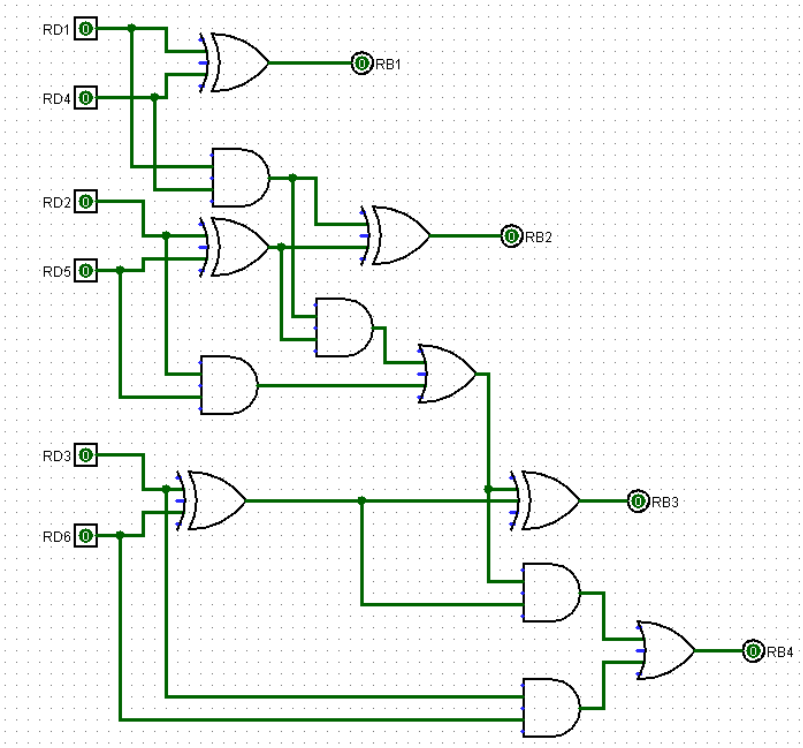


image 8.5.

The diagram on this picture, is actually a 3-bit adder with a carry bit. The code generated for this example is given in snippet 8.5.

```
void logic_function(void) {
   PORTBbits.RB4 =
OR(AND(OR(AND(AND(PORTDbits.RD1,PORTDbits.RD4),XOR(PORTDbits.RD2,PORTDbits.RD5)),AND(
PORTDbits.RD2,PORTDbits.RD5)),XOR(PORTDbits.RD3,PORTDbits.RD6)),AND(PORTDbits.RD3,POR
TDbits.RD6));
   PORTBbits.RB2 =
XOR(AND(PORTDbits.RD1,PORTDbits.RD4),XOR(PORTDbits.RD2,PORTDbits.RD5));
   PORTBbits.RB3 =
XOR(OR(AND(AND(PORTDbits.RD1,PORTDbits.RD4),XOR(PORTDbits.RD2,PORTDbits.RD5)),AND(POR
TDbits.RD2,PORTDbits.RD5)),XOR(PORTDbits.RD3,PORTDbits.RD6));
   PORTBbits.RB1 = XOR(PORTDbits.RD1,PORTDbits.RD4);
}
```

code snippet 8.5.

It's somewhat harder to observe that the functionality of this snippet is correct, because the expressions are pretty long and tedious. However, it's easy to see that for RB1 and RB2, the functionality is correct. Even for RB3 we can observe, with some effort, that the logic is correct, which pretty much implies that it's almost certainly true for RB4 as well, as the diagram is repeating. In this example we can see that the functionality of the code holds even for more complex diagrams.

Simulation

As part of the project, a video has been submitted, in which the code generated for example 4, is being executed, which is a good way to check the functionality of the code. Several images are given as part of that testing. On image 8.6. and image 8.7. we see the case where both input pins, RD1 and RD2 are logic one.

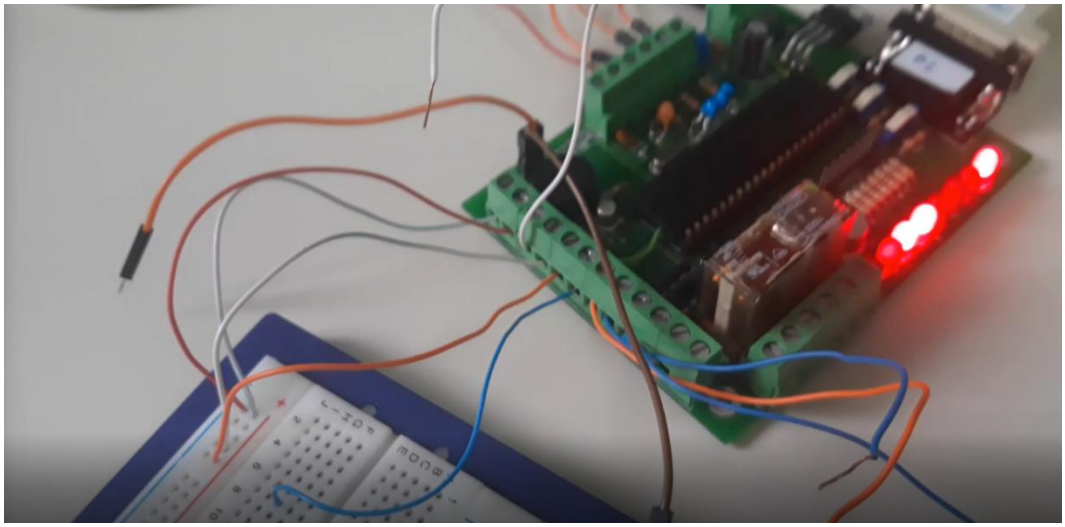


image 8.6.

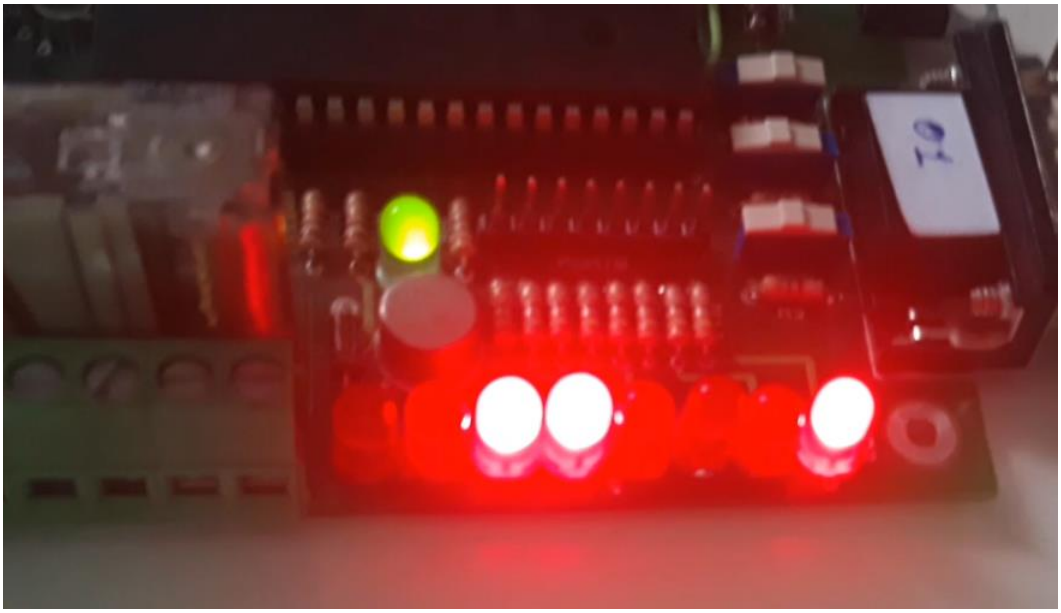


image 8.7.

The pins from port B are read from the bottom up, so we can see that the values are exactly those which we were expecting – both the negations are zero, whereas for the 2-input gates, the corresponding outputs are AND = 1, OR = 1, NAND = 0, NOR = 0, XOR = 0, XNOR = 1. It's easy to see that these are the correct values.

On image 8.8. we have shown the testing for the case where one of the input pins is logic zero, and the other is logic one.

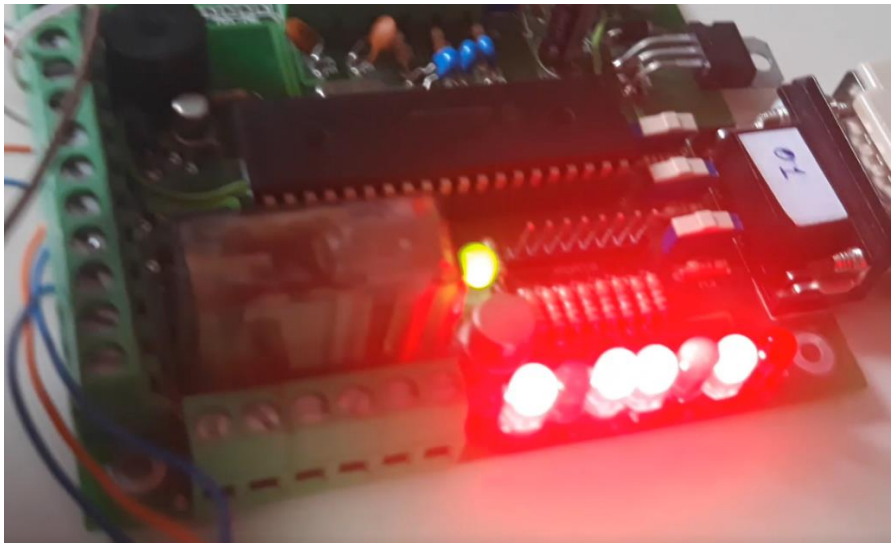


image 8.8.

We can se that the value of one negation is 0, and the other one is 1, which should happen. Now the value of AND has become 0, so the value of NAND has become 1. OR and NOR remain the same as the last case. XOR has now become 1, so XNOR has become zero. Therefore, all is in accordance to what we were

expecting. It’s sufficient to test only one case where the inputs are different, because the basic logic gates are all commutative.

On image 8.9. we have shown the testing for the case where both inputs are logic zero.

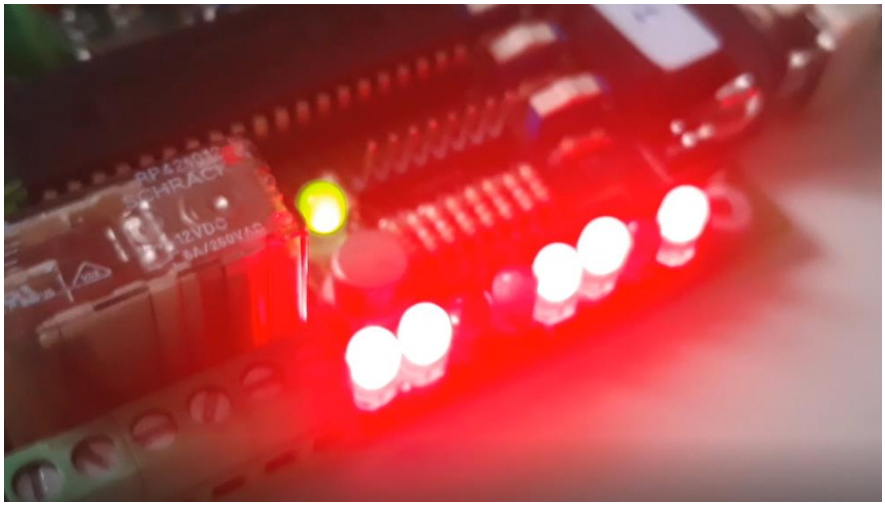


image 8.9.

Both negations are logic one. AND and OR are now zero, so NAND and NOR are both one. XOR is logic zero, so XNOR is logic one. It’s clear that this is consistent with what we’ve been expecting. Therefore, the testing has demonstrated. that the logic for example 4 is correctly implemented.

9. Uploading the code to the microcontroller

After the algorithm has generated the appropriate code main.c, we need to upload the code to the microcontroller. The steps for uploading the code to the microcontroller are the following:

1. In the MPLABX programming environment, select File->New project and in the category Microchip Embedded select Standalone Project. Click next.
2. In the drop-down menu Device select PIC16F1939 and click next.
3. For Hardware Tools, select Simulator and click next.
4. In the dialog Select Compiler select XC8 compiler and click next.
5. Choose a name for the project and select a location to which you wish to save the project. Afterwards, click Finish.
6. Right click on Source Files, and then choose New->C Main file.
7. Name the file main(.c) and click Finish.
8. Navigate to the location where the folder has been saved, and replace the file main.c in the folder with main.c generated by the algorithm.
9. Build the project with the new main.c. This will generate a .hex file in the subfolder dist.
10. Using the PICPGM software, choose the .hex file from the project folder. Detect the controller connected to the PC, and finish writing the .hex file to the controller. With the conclusion of this step, we have finished the process.

10. Conclusion

We have seen, that even for a very concrete implementation problem, sometimes it's necessary to utilize quite abstract concepts from computer science, or in this case graph theory. Of course, in this case, the concepts used were only the basic ones. It should be said though, that the abstractions given by the theory and all the guides that come with those abstractions, are definitely useful for solving some concrete problems.

We have determined that the expected functionality has been implemented. The code generates the correct C code for the microcontroller, for an arbitrary logic diagram. However, several points should be made about things which were ignored in the project implementation, and which are either easy to add or useful, or both, for expanded versions.

Firstly, the current version of the code doesn't support rotations of the components in 4 directions, which is one option covered by the Logisim software. In the project it was assumed that all gates are oriented from west to east, as are all input pins, while all output pins are oriented from east to west. It's relatively easy to include the feature of component rotation. The only part of the code which would be changed, would be the one which concerns itself with connecting the logic gates with their inputs. Of course, there would need to be adjustments to the processing of the ".circ" file, to obtain the new information.

The second feature which is not included, is that arbitrary pins could be inputs our outputs. This is a simplification, but as in the previous case, it's easy to modify the code so that it supports even the more general functionality. It would be necessary to read all the pins from the ".circ" file, at the beginning, from which we could then extract the information about which pins are input and which output. Then we would have to modify an additional part of the C code, which in the project version is hardcoded, and which concerns itself with the declaration of ports as inputs or outputs.

The third thing which would be neat to add to the project, is a makefile, which would run the Python project, then run the MPLAB C compiler, and finally program the microcontroller using the software PICPGM, all in one command.

All 3 of these functionalities are quite simple to add, and each one of them could be added in the span of a few hours. However, these functionalities aren't key for the functioning of the software, so for simplicity sake they aren't included in the current version.

The final addition, which is in turn the most interesting one, would be adding multiplexers and memory cells (flip-flops). Considering that we would have multiplexers on our disposal, the flexibility of creating logic diagrams would be considerably improved, given that we would be able to emulate an arbitrary logic function and not only use the basic ones. Because of the presence of flip-flops, we would be able to design automata, which would drastically increase the number of practical systems to which the software could be applied.

Implementing the aforementioned components would make the project considerably more difficult, so considering the scope of this project, they are not implemented.

11. References

- “Logički dizajn”, Melita Ahić-Đokić, Elektrotehnički fakultet Sarajevo, 2006
- “Digitalni integrirani krugovi”, Mustafa Musić, Abdulah Akšamović, Sarajevo 2018.
- “Diskretna matematika za studente tehničkih nauka”, Ž. Jurić, ETF Sarajevo, 2011.
- “Osnove operacionih istraživanja”, T. Mateljan, Ž. Jurić, R. Turčinhodžić, ETF Sarajevo, 2018.
- Lecture notes for “Praktikum automatike”, Doc.dr Samim Konjicija
- Kahn, Arthur B. (1962), "Topological sorting of large networks", *Communications of the ACM*, **5** (11): 558–562, doi:10.1145/368996.369025
- Logisim documentation
- Python documentation
- MPLAB documentation
- PICPGM documentation
- PIC16F1939 documentation
- http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/index.html
- https://www.researchgate.net/figure/Summary-of-the-common-Boolean-logic-gates-with-symbols-and-truth-tables_fig3_291418819
- https://www.electronics-tutorials.ws/logic/logic_8.html
- https://www.researchgate.net/figure/An-example-of-a-network-to-illustrate-terminology-used-in-graph-theory-Stam-2004_fig2_304990430
- https://favpng.com/png_view/graph-theory-path-graph-graph-theory-png/YKcnbMPY
- https://www.pngfuel.com/free-png/uytye
- https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/
- https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/
- https://algorithms.tutorialhorizon.com/topological-sort/