

Logic Gate Simulation

Jan Johannsen

December 2023

Contents

1	Introduction	1
1.1	What are Logic Gates?	1
1.2	An Example Logic Gate Combination	2
1.3	Problem	2
2	Event Based Simulation	3
2.1	Problems with the time step approach	3
2.2	Event based Simulation	3
3	Software Structure	4
4	Experiment Description	5
5	Experiment Results	5

List of Figures

1	Model of function: $((A \vee B) \wedge \neg C) \vee (B \vee \neg D)$	2
2	Logic Gate Simulation Program Structure UML class diagram . .	4
3	Outputs for GateOut	6
4	Outputs for InputA	6
5	Outputs for InputB	7
6	Outputs for InputC	7
7	Outputs for InputD	8
8	Truth Table for experiment function	9

1 Introduction

1.1 What are Logic Gates?

Logic gates are devices that perform a Boolean function based on given binary inputs. They are essential parts of most modern electronic devices. One of

the main use cases for logic gates are processors. Logic gates have a base set of Boolean operations. These can be used in isolation or combined into more complex Boolean functions. These operations are AND, OR, XOR and NOT. With AND, OR and XOR each receiving a set number of two inputs and NOT only allowing a single input. The AND operation returns true only if both of the given inputs are true. The OR operation returns true if either of the given inputs is true. The XOR operation returns true if exactly one of the inputs is true. Finally the NOT operation flips the given inputs resulting in true if the input was false and false if the input was true.

1.2 An Example Logic Gate Combination

One possible combination of these base operations is the following Boolean function:

$$((A \vee B) \wedge \neg C) \vee (B \vee \neg D) \quad (1)$$

This function is visualized in figure 1. With the Boxes A through D representing the binary inputs. Each of these inputs is connected with the input of a following base operation. Depending on the type each operation can receive either one or two incoming connections, while allowing for an arbitrary number of outgoing connections. Finally the output has exactly one input, which is the result of the function.

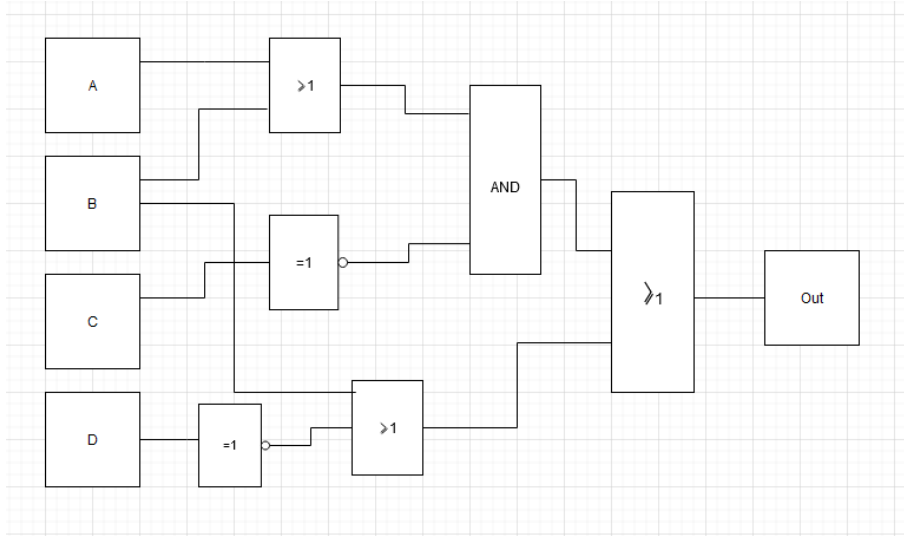


Figure 1: Model of function: $((A \vee B) \wedge \neg C) \vee (B \vee \neg D)$

1.3 Problem

Since Logic gates are in reality not able to calculate the entire function they are based on instantaneously and usually run for as long as they continue getting

inputs, time becomes the main problem when trying to simulate logic gates. If for example the calculation delay for an OR Gate would take $10\text{ }\mu\text{s}$ then the result of the calculation would reach the following gates or the output only after that delay. This results in the same state persisting in the following gates until the delay is over even if the inputs were changed. To demonstrate the problem and simultaneously keep the complexity of the simulation manageable this paper will only consider delays for the calculation of the base functions. This means transportation, input and output delays are assumed to be negligible. Given this the goal of the simulation will be to calculate the current output value for each gate for a given time during the simulation time frame.

2 Event Based Simulation

The following will introduce event based simulation. First the problems with the time step based approach regarding this problem will be addressed. Following this event based simulation will be introduced as an approach to avoid the mentioned problems.

2.1 Problems with the time step approach

While a possible way to simulate a function through logic gates, is to iterate over every input and gate of the function for each time step Δt it is far from optimal for this type of problem. Take the following example. The example function from figure 1 is simulated for a total duration of $100\text{ }\mu\text{s}$. During this simulation only the value of input D changes every $10\text{ }\mu\text{s}$. Taking the approach described above one would need to iterate over every input and gate with a chosen step distance. This would result in the calculations D has no effect on being redundant. Furthermore the problem of picking the right time steps remains. While both of these problems could be optimized to an extend, this would raise the complexity of this simulation drastically.

2.2 Event based Simulation

A better approach for the given concept is event based simulation. Instead of simulating the whole function at given intervals, changes in inputs are regarded as events. These events have to be managed by a central instance regarded as Eventmanager in the following. This means that given the changes in the inputs in form of events the Eventmanager has to alter the gates following the inputs. Furthermore this includes the creation of new events resulting from following gates. For each event to be executed at the proper time, each event needs to include a timestamp. This enables the Eventmanager to sort the list of upcoming events in regards to this timestamp. This also means that Events resulting from gates with delays will be added to the list of events with a timestamp in the future based on the delay of the specific gate.

3 Software Structure

The following will discuss a possible program structure to simulate the given problem through event based simulation. A basic visualization through a UML class diagram can be found in figure 2.

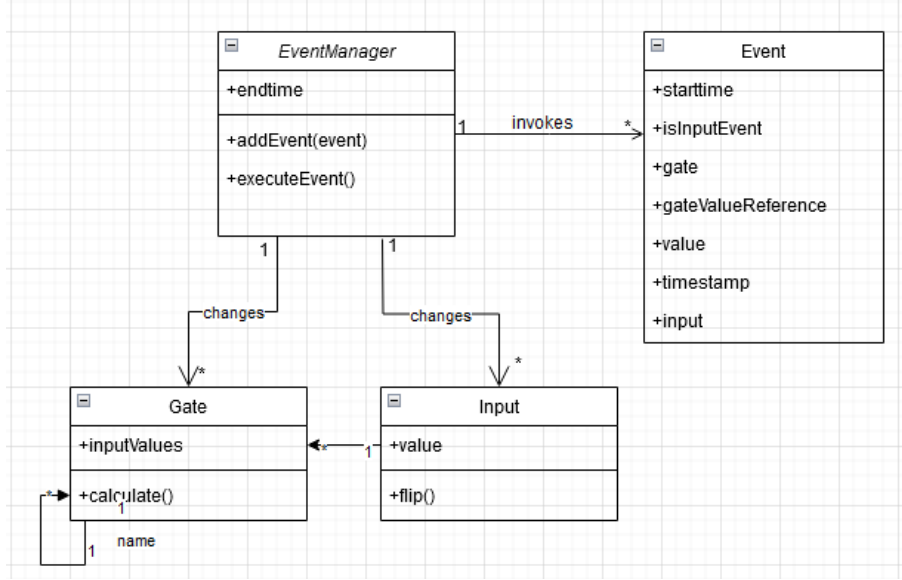


Figure 2: Logic Gate Simulation Program Structure UML class diagram

As described above the core of the program will be the EventManager. This Object will manage the simulation. It contains the end time of the simulation, a list of events and the list of inputs and gates contained in the Boolean function. The list of events should be sorted by timestamp with new events entering at the correct position based on their timestamp. The EventManager has the methods `addEvent(event)` and `execute()`. The first one taking an event and adding it at the correct position inside the event list. The latter executing the event which is up next in the sorted event list. This results in the event being removed from the list, the referenced object getting changed and new Events getting created for each following gate referenced in the changed gate or input. To simulate the behaviour of the gates one can now initialize the event list with events containing the changes for the inputs over time and call the execute method in a loop. Through the execute method the event up next will be worked through with the `addEvent` method being used to add events resulting from the changes. This simulation runs until either the events run out or the timestamp of the current up next event is later than the specified end time.

4 Experiment Description

Taking the structure mentioned above as a basis for an implementation, the effectiveness should be tested afterwards. For this paper this is done through an Experiment, while logging the changes throughout the gates. The experiment is set up as follows. Taking the gate layout from 1 the inputs A through D are interpreted as one of four bit. With A being the least significant and D being the most significant bit. During the initialization of the events, the necessary events for each number going from 0 to 15 and from 15 to 0 represented in binary will be added to the event queue. Taking this base configuration the program is run with each gate logging changes in their outputs. A change in output for a gate is logged as the time step of the change occurring and the subsequent value. To better compare the results of each binary pattern to the truth table for the example function, the events for each binary pattern are delayed 30 ns. This value is specific to the given gate layout and set up to be slightly higher than the duration for an input to travel from the start to the output gate following the longest path.

5 Experiment Results

To visualize the results of the described experiment the figures 3, 4, 5, 6, 7 were plotted using matplotlib. It is important to note, that the changes in output weren't plotted with their exact time stamp but rather the current value of the output for each gate with 30 ns intervals was used. Furthermore while matplotlib draws straight lines between each point, the values for each output don't gradually change and instead instantly go from zero to one and the other way around. That being said, taking a look at the outputs for the inputs A through D from figures 4, 5, 6 and 7 one can see the expected behaviour of the frequency in changes decreasing when going from LSB to MSB (Least/Most significant Bit). With the exception of the inputs representing 15 being repeated once the input for the LSB A change every time step of 30 ns. In contrast to this the input for the MSB D only changes to true and back once during the experiment. When comparing the outputs for the output gate in 3 with the functions truth table in 8 one can see that a first change to True happens at around 270 ns. This is according to the truth table having a first change in output only after the input D changes for the first time. Due to the simulation starting at time 0 ns the ninth input happens at 240 ns with the the delay of the gates resulting in a change around 30 ns later. This is also true for the patterns for 12 and 13, where the changes are matching the truth table again. While this stays for the first half of the experiment slight inconsistencies are noticeable in the second half. Namely the results are inverted until the pattern for value 8, but afterwards remain true. This is likely due to an error with the implementation and won't be further investigated in the following due to time constraints.

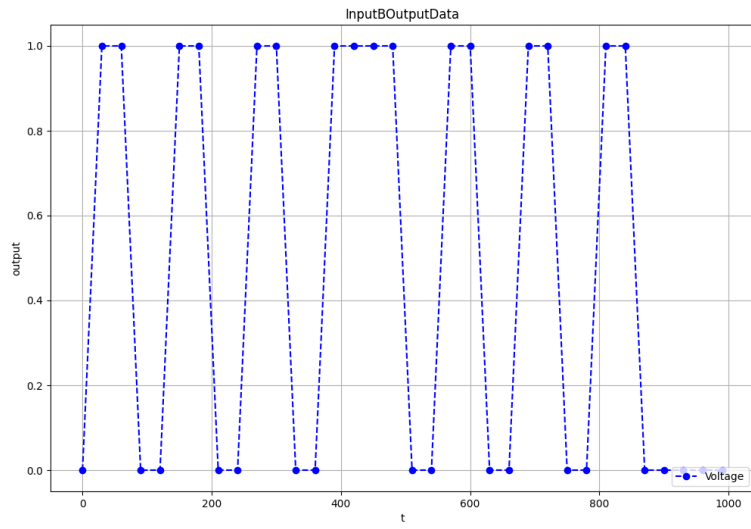


Figure 5: Outputs for InputB

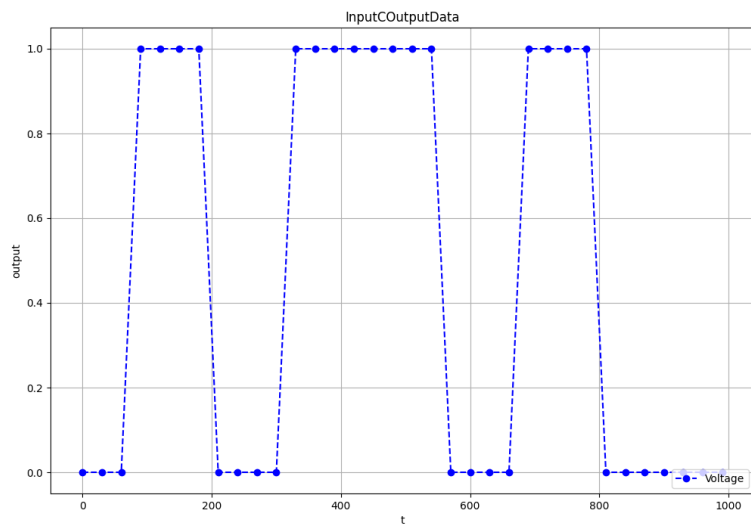


Figure 6: Outputs for InputC

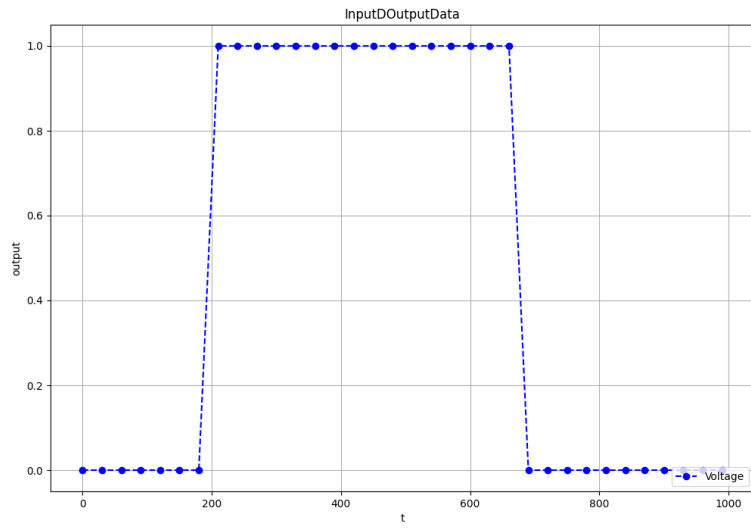


Figure 7: Outputs for InputD

YOUR INPUT

Find the truth table for $((a \vee b) \wedge \neg c) \vee (b \vee \neg d)$.

ANSWER

d	c	b	a	$((a \vee b) \wedge \neg c) \vee (b \vee \neg d)$
True	True	True	True	True
True	True	True	False	True
True	True	False	True	False
True	True	False	False	False
True	False	True	True	True
True	False	True	False	True
True	False	False	True	True
True	False	False	False	False
False	True	True	True	True
False	True	True	False	True
False	True	False	True	True
False	True	False	False	True
False	False	True	True	True
False	False	True	False	True
False	False	False	True	True
False	False	False	False	True

Figure 8: Truth Table for experiment function