Boolean Logic II [1][2]

IT Systems, Summer Term 2026

Dr. Matthes Elstermann

We start our journey towards a computer with Boolean logic. This topic is covered in two presentations, of which this is the second one.

1 Introduction

- Part 1
  - Introduction
  - Boolean Logic
- Break for self-study
- Part 2
  - Boolean Circuits
  - Sample Transformations
  - Multiplexors and De-Multiplexors, Project 1
  - Conclusions

In part 1, basics of Boolean logic were introduced.

In part 2, we examine the construction of Boolean Circuits. Towards this goal, we explore Sample Transformations to simplify Boolean expressions, before their implementation.

Then we look at the functionality of Multiplexors and De-Multiplexors, with a specific focus on the implementation of these and other chips in Project 1. Finally, we draw Conclusions.

2 Boolean Circuits

We now turn to the implementation of Boolean functions in terms of circuits.

A circuit is a device that consists of multiple gates or chips, which are wired together to form more complex functionality.

2.1 Symbols for Logical Gates

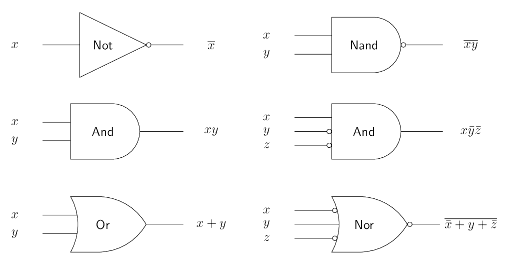


Figure 1: "Symbols for logical gates" under CC0 1.0; from GitLab

As building blocks for circuits we use logical gates, for which you see symbolic representations here.

On the left, we see symbols for the operations Not, And, Or. On the right, small circles denote negation, which may appear for inputs or the output. Nand is the name for a negated And. Nor is the name for a negated Or.

Also, gates may have different numbers of inputs, which is introduced as fan-in on the next slide.

[1]This PDF document is an inferior version of an OER in HTML format; free/libre Org mode source repository.
[2]Material created by Jens Lechtenbörger; see end of document for license information.

2.2 Fan-In and Circuits

- Gate's interface defines number of inputs
  - Gates and chips have **pins** for inputs and outputs
  - Number of inputs called **fan-in**

    The interface of a logical gate defines how many inputs it accepts and how many outputs it produces. In fact, in hardware, the inputs and outputs may be visible as pins, which can be wired to other gates or chips to form larger circuits.

    The number of inputs for a gate is called its **fan-in**.

  - Fan-in is 2 for `Nand`, `And`, `Or` in case of Nand2Tetris
    - With associative and commutative operations, gates of any fan-in work in any order of computation

      For typical operations such as Nand and Or, the fan-in is 2 in Nand2Tetris.

      If you think about it, we might also build gates with a larger fan-in.

    - Consider

$$f_2(x_1, x_2, x_3) =$$
$$\bar{x_1}\bar{x_2}x_3 + x_1\bar{x_2}\bar{x_3} + x_1\bar{x_2}x_3 + x_1x_2\bar{x_3}$$

x_1	x_2	x_3	$f_2(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Consider the function f_2 shown here. If we want to implement f_2 as circuit starting from the DNF, the fan-in defines how many gates we need.

- Alternatives
  - Fan-in of 2

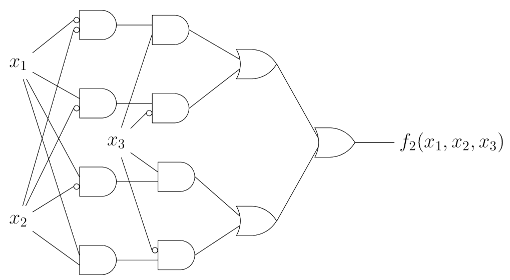


Figure 2: Figure under CC0 1.0

  - Fan-in of 4

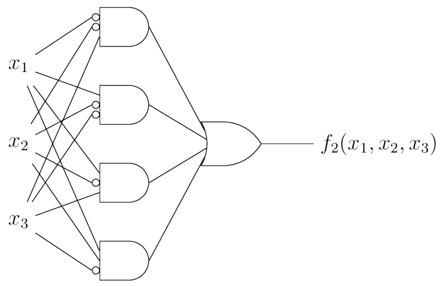


Figure 3: Figure under CC0 1.0

Clearly, with a fan-in of 2 we need more gates than with a fan-in of 4. Please draw such circuits on your own.

2.3 Multi-Bit and Multi-Way Gates

- **Multi-bit** gates/chips: Inputs are n-bit operands each
  - E.g., `And16`: And for 16-bit numbers is applied bit-by-bit
    - `And16(1100110011001100, 00001111000011110000) = 0000110000001100`
    - Use 16 `And` gates in implementation

So far, we manipulated individual bits. Of course, when building a computer, we also want to be able to operate on larger quantities, e.g., on 16-bit numbers. For this purpose, we use **multi-bit** versions of gates and chips. E.g., if we want to apply some logical operation to 16-bit numbers in a bit by bit fashion, we can build a chip which embeds 16 gates that operate on individual bits each.

- **Multi-way** gates/chips: More than 2 inputs, i.e., fan-in larger than 2
  - E.g., `Or8Way`: 8 inputs; `out = 1` if at least one of them is 1
    - Use suitable number of `Or` gates in implementation

In addition, we can build so-called **multi-way** chips, which have a fan-in larger than 2. E.g., in Nand2Tetris, you are supposed to build an Or chip with 8 inputs. It is easy to see that you can do so by using a suitable number of Or gates that operate on 2 inputs each.

3 Sample Transformations

We now apply some laws to simplify Boolean expressions.

3.1 Sample Algebraic Simplifications

- Simplify $f_2(x_1, x_2, x_3)$

$$
\begin{aligned}
&= \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3 \\
&= (\bar{x}_1 + x_1)\bar{x}_2x_3 + (\bar{x}_2 + x_2)x_1\bar{x}_3 \\
&= \bar{x}_2x_3 + x_1\bar{x}_3
\end{aligned}
$$

Let us simplify f_2, which is in disjunctive normal form.

In the first step, we look for pairs of minterms that differ only in the negation of one variable. On such pairs, which are shown in the same color, we use distributivity to factor out common terms. (Actually, associativity and commutativity are also used to bring the minterms and their variables into forms and positions where distributivity is applicable.)

In the second step, we eliminate the complementing variables.

If this step is not obvious to you, please revisit the laws shown earlier: A sum of a variable with its negation results in 1, and a multiplication with 1 does not change the result, which is why it can be omitted.

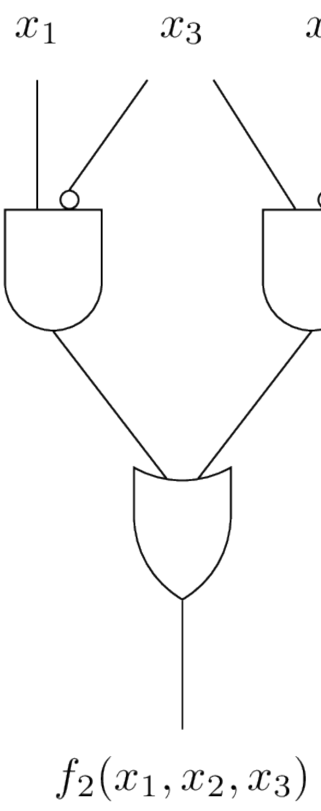
- Simplify $\bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3 + x_1x_2x_3$

  Please try to proceed similarly here.

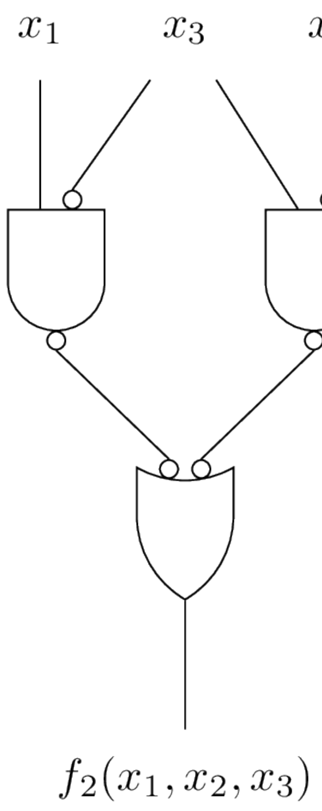
  - $\bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2x_3 + x_1x_2x_3$
  - $(\bar{x}_1 + x_1)\bar{x}_2x_3 + (\bar{x}_2 + x_2)x_1x_3$

    Here, we first duplicate the second minterm, which is allowed as x equals x + x. Afterwards, we can use each of the duplicates for a separate application of distributivity.

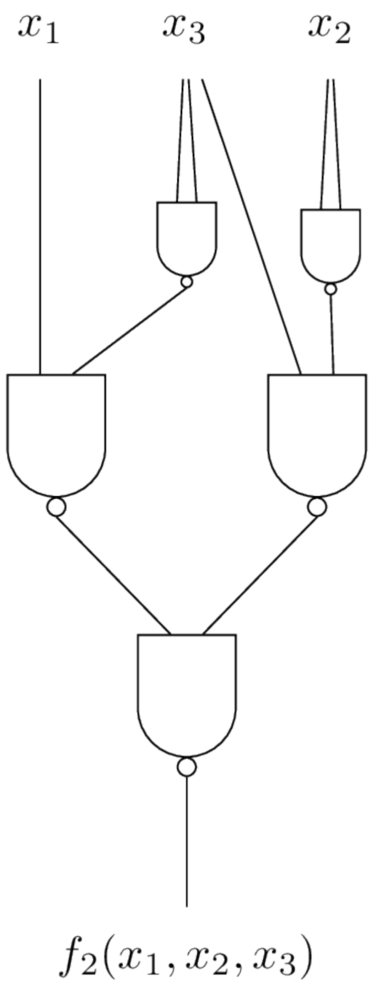
3.2 DNF to All-Nand, Graphically



(a) Figure under CC0 1.0



(b) Figure under CC0 1.0



(c) Figure under CC0 1.0

As an exercise in Boolean logic, let us see an easy way to transform any sum of products into a circuit containing only Nand gates.

We start from a circuit for the simplified expression on the previous slide.

In the first step we introduce double negations on the paths from And gates to Or gates. As an easily verified law of Boolean logic, two negations cancel each other out. Thus, this step does not change the implemented function.

Next, we use a De Morgan rule to replace the Or gate and its negated inputs with a Nand gate. Besides, we replace the remaining Not gates with Nand gates. Again, this step does not change the implemented function.

We are now left with a circuit that contains only Nand gates.

Note that the transformations shown here are meant as an exercise in Boolean logic. In general, we do **not** aim for implementations of circuits that are restricted to Nand gates. Instead, we will build and use more and more complex chips subsequently.

4 Multiplexors and De-Multiplexors, Project 1

We now introduce multiplexors and de-multiplexors, which you build as part of Project 1. Later on, those chips turn out to be important to select among multiple choices and to perform routing in more complex circuits.

4.1 Mux

- Truth table

a	b	sel	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- Specification (from `Mux.hdl`)
  - `If sel==1 then out=b else out=a.`

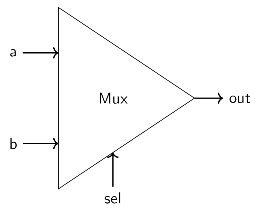


Figure 5: Figure under CC0 1.0

A multiplexor, or mux for short, implements the if statement shown here. Based on a selector bit, it forwards one of two inputs to its single output.

  - Alternative view on truth table

sel	out
0	a
1	b

The truth table corresponds exactly to this specification.

- Self-study
  - Implement `Mux`
    - Start from truth table, simplify (task in Learnweb)
    - Use Not, And, Or gates
- Maybe in class
  - Implement `Mux8Way16`

Implement the Mux yourself.
In class, we may take a look at a more complex multi-way, multi-bit variant, if you want to.

4.2 DMux

- Truth table

in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

- Specification (from `DMux.hdl`)

```
{a,b} = {in,0} if sel==0
        {0,in} if sel==1
```

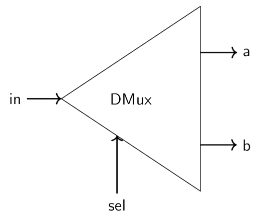


Figure 6: Figure under CC0 1.0

A DMux is a routing device, which forwards a single input based on a selector bit to one of two outputs.
The specification says that in case of the selector bit being 0, the input is forwarded as output `a`, while output `b` is 0.
If the selector bit is 1, the input is forwarded as output `b`, while output `a` is 0.
Again, the truth table on the left specifies exactly the same behavior as the if statement on the right.

- Self-study
  - Implement `DMux`
    - Read from truth table
    - Use one Not, two And gates
- Maybe in class
  - Implement `DMux8Way`

Implement the DMux yourself.
In class, we may take a look at a more complex multi-way, multi-bit variant.

4.3 Project 1

- Given: `Nand(a,b)`, `false`

a	b	Nand(a, b)
0	0	1
0	1	1
1	0	1
1	1	0

- Build:
  - Not(a) = . . .
  - true = . . .
  - And(a,b) = . . .
  - Or(a,b) = . . .
  - Mux(a,b,sel) = . . .
  - Etc. - 12 gates altogether
- See Chapter 1 of (Nisan and Schocken 2005)

As part of project 1 of Nand2Tetris, your task is to implement several chips. For some chips, slides contain hints. For others, please try yourself, also in class. Please do not hesitate to ask.

Importantly, recall that Chapter 1 of our book ends with tips.

5 Conclusions

Let us conclude.

5.1 Summary

- Boolean logic is formal foundation for functionalities of gates and chips
- DNF provides canonical representation for Boolean functions
  - Can be read off truth table
  - Simplification with laws
- Project 1 builds on Boolean logic to construct gates and circuits
  - Starting from {Nand}, which is functionally complete

Boolean logic serves as the formal foundation for the operations of gates and chips, providing a rigorous framework for digital functionality. Within this framework, Disjunctive Normal Form provides a canonical representation for Boolean functions.

DNF can be directly derived from the truth table and further simplified using established laws. Project 1 leverages Boolean logic as its foundation to construct gates and circuits. It starts from the functionally complete Nand gate, laying the groundwork for subsequent circuit design and implementation.

5.2 Q&A

- Please ask questions and provide feedback on a regular basis
  - Something confusing?
    - What did you understand? Where did you get lost?
  - Maybe suggest improvements on GitLab
    - Did you create exercises, experiments, explanations?
  - Use Learnweb: Shared, anonymous pad and MoodleOverflow

This slide serves as reminder that I am happy to obtain and provide feedback for course topics and organization. If course material is confusing, please let us know. This will be most effective if you describe your current understanding, which might allow us to identify root causes of misunderstandings. Please ask questions that allow others to help you, either in a shared pad or in our forum. Most questions turn out to be of general interest; please do not hesitate to ask and answer where others can benefit.

In addition, if you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, …), please share. Maybe suggest improvements to course material on GitLab.

Bibliography

Nisan, Noam, and Shimon Schocken. 2005. *The Elements of Computing Systems: Building a Modern Computer from First Principles.* The MIT Press. `https://www.nand2tetris.org/`.

The bibliography contains references used in this presentation.

License Information