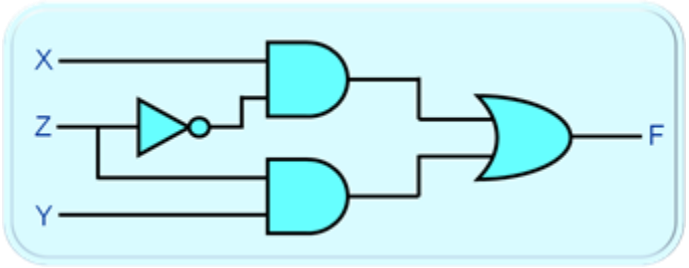


Logic Circuits on FPGAs

study text of the course

Logic Systems and Processors

Richard Šusta

DCE	**Department of Control Engineering** **CTU-FEL in Prague**	

Version 2.0 of 5. listopadu 2023

Copyright (c) 2023, Richard Susta.

Permission is granted to copy and distribute this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled:
GNU Free Documentation License.

Author: Richard Susta, richard@susta.cz, https://susta.cz/

Figures: * Richard Susta, except for three introductory illustrations:
Figure 1 from Xilinx Inc., and Figures 2 and 3 from Terasic Inc.

Publisher: Department of Control Engineering CTU-FEE in Prague,
Technicka 2, 166 00 Prague 6
http://dce.fel.cvut.cz/

Datum of issue: November 2023

Length: 35 pages

Homepage of the document:

https://dcenet.fel.cvut.cz/edu/fpga/guides.aspx

Table of Content

1 About the textbook Designing Logic Circuits on FPGAs

Today, circuits are designed mainly by textual statements of some HDL (Hardware Description Language). For example, we can utilize VHDL, Verilog, or System Verilog. But first, we should understand the properties of actual logic circuits.

The minimum helpful knowledge was summarized in two textbooks:

The binary pre-requisite explains the encoding of signed and unsigned integers, their hexadecimal and BCD notation and conversions, and the coding of characters. These are the basics; other textbooks and lectures assume their reliable knowledge. We believe readers are familiar with binary codes, so we separated this part for cases of knowledge refresh.

Logic Circuits on FPGAs, which you are reading now, covers the main logic designs and principles, without which it isn't easy to design anything. You will find here general knowledge about logic circuits without descriptions of them in HDL languages. Everything is explained with schematics, and for some of them, the circuit versions are given when implemented on FPGAs, more about which will be right on p. 7.

- First, direct applications of Boolean logic theorems to circuitry are reviewed.
- The next chapter on logic functions begins with their specification and continues to minimization using Karnaugh maps.
- Then, the internal structure of CMOS gates and their basic properties will be presented. They are essential for the construction of circuits and understanding their behavior.
- The next chapter deals with basic combinational circuits. It starts with decoders and multiplexers. It also looks inside the FPGA circuit, but only through the eyes of the user.
- The following part concerns the properties of general arithmetic, such as adders, comparators, and multipliers. Appropriate conversions of slow division to multiplication are also given.
- The textbook concludes with a chapter on synchronous circuits. They are not complicated in substance, but many students encounter them for the first time. Their correct use is sometimes more challenging, at least in my teaching experience.

The textbook "Logic Circuits on FPGAs" serves as a springboard for design that can solved with any HDL language. In our Control Engineering Department, we have chosen VHDL, which we consider more convenient for beginners. We describe it in our other textbook, not finished yet, "Circuit Design in VHDL 2008 for C Programmers".

If somebody chooses to continue with Verilog or SystemVerilog, there are many books from other authors.

Why learn logic circuits?

The importance of logic circuits depends on our future professional direction. In the personal computer environment, we will need minimal knowledge of the logic since we will use it only in conditions, such as decision statements like if-then-else or while.

The situation changes when we use non-standard computing systems, for example, in developing drones or other experimental devices. If we use a processor designed for applications in practical devices, then we need to connect peripherals to it, and not all of them have mass-produced service modules.

The data flow can look like this:

1) technical equipment of inputs or outputs;
2) **logic circuits connected to it that read/write/process data;**
3) **processor bus interface;**
4) operating system driver;
5) user program.

We have to solve the peripheral connection ourselves or have someone design it, but knowing what can be implemented in the circuit here is helpful. Interface handling is often associated with **data preprocessing**.

Some algorithms can be converted to circuitry that works faster than their computation in the processor. Here, we can mention, for example, image signal filtering or FFT, the fast Fourier transform. Similar circuit solutions are called **hardware accelerators**. They save computational time and free up space for other tasks.

Why learn the structure of circuits when they are described in HDL anyway?

I have been teaching logic for decades. I saw that most of my students do not design circuits. They program them. They mimic the constructs they learned in higher programming languages like C or Java. You can use some, but certainly not all, because there are different resources inside circuits than in assembly language instructions.

The designer should always see the created circuit behind the HDL description, not a program. Only isolated parts for simulation are converted to it. Everything else is plugged in, so it pays to know how our code is implemented first and then start with HDL descriptions.

1.1 Linguistic notes on text and pictures

The textbook was translated from the original Czech language by a DeepL and then manually corrected with the aid of the Grammerly checker. Readers may encounter some sentence that escapes proofing. They will be purified in the following versions.

The textbook contains over 200 original drawings, but only 170 are numbered, as many figures were inserted without titles, if only to expand on the earlier illustration.

We do not use LaTeX. DeepL does not support its documents. We selected the MS Word editor, which does not allow inserting some cases of cross-references by a simple number. Some of them can also look clumsy.

1.2 How is the logic implemented?

In the past, logic circuits were built by wiring small components such as gates, counters, registers, etc. This way is too slow and remains probably only in fun building sets. And the complete design of a monolithic integrated circuit is costly.

Universal chips offer a cheaper way for small series. Their widespread representative is the FPGA (Field Programmable Gate Array), which also provides a good solution for debugging monolithic integrated circuits' prototypes.

Here, we must emphasize that the term "**program**" was understood as "**configure**" at the time when the first FPGA predecessors appeared (ca. 1983). Later, it became associated with pro-

gramming languages[1] . And the word "program" certainly does not fit with circuit designs, which are fundamentally not programmed in today's meaning but designed!

Today, the abbreviation FPGA should be more accurate as FCGA (Field Configurable Gate Arrays), but it will probably never change. When we mention the established term "FPGA programming," we will always mean that the FPGA configuration was loaded to modify its behavior to a new circuit.

Some families of FPGAs contain entire processors, such as the Zynq-7000 from AMD Xilinx, which includes two ARM Cortex-A9 processor cores. The MZAPO tutorial board, the left part of the picture below, is an example of its usage. It was developed by Pavel Píša and Petr Porazil from the Department of Control Engineering at the Faculty of Electrical Engineering in Prague. The yellow part of the Zynq-7000 internal structure contains freely programmable logic, which they used to create the MZAPO peripheral operator.

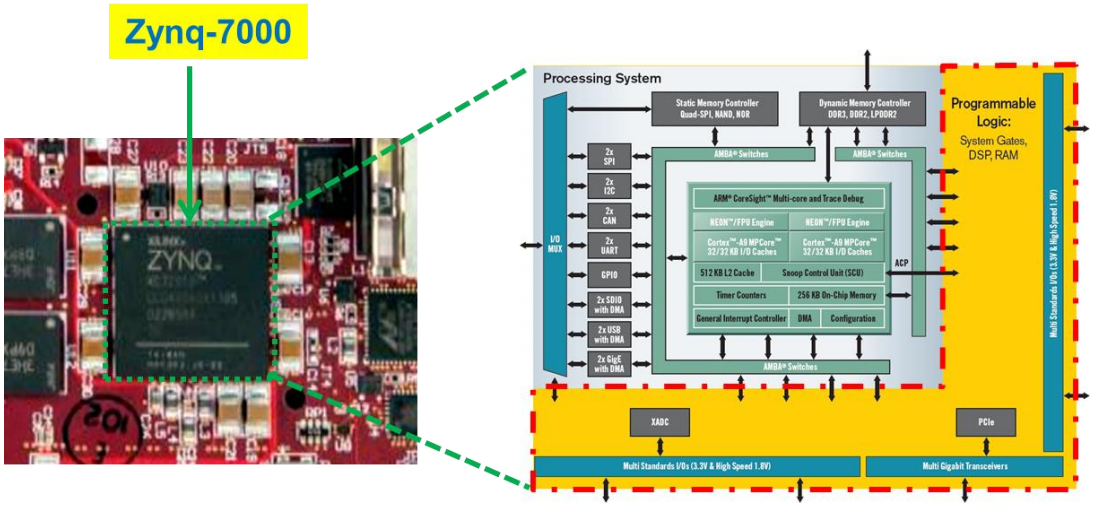


Figure 1 - Zynq™-7000 (source of right image Xilinx)

Tens of thousands of logic functions can be configured in the yellow section, depending on the type of Zynq-7000 family circuit. For example, the XC7z010 FPGA used in MZAPO allowed peripherals to be built using up to 17,600 logic functions, each capable of six inputs. There are also 35200 flip-flops, over two megabits of fast SRAM memory, and other support units such as hardware multipliers and bus service modules.

Some types of FPGAs are sold without processors, just with programmable logic. It can also be used to create a processor, called a soft-core processor, which also offers high variability. And there can be several of them, like whole networks of them.

1.3 What do we get by using FPGAs?

- FPGA components have been developed primarily for individual or small series of chips, where they overtop too costly monolithic integrated circuit designs. Wiring using FPGAs is often cheaper and faster than soldering a printed circuit board with individual components.
- A monolithic integrated circuit cannot be repaired if a fault is found. We can correct a printed board, but not in all cases, and usually laboriously. But we will quickly modify FPGA. We only load the new configuration into it. Space applications often use their ability

[1] The meaning of more computer terms has shifted. After all, even the term "hacker" was around 1960 used to describe a glorified computer expert. Later it did take on a more pejorative connotation when some experts began to abuse their knowledge.

to repair or improve a function remotely. For these, FPGA types with enhanced radiation resistance are produced.

- Processor emulators are also typical FPGA applications. They can substitute old processors that are no longer available or types under development. The emulators also allow software development before starting a chip's production.
- No FPGA overruns monolithic circuits of the same level of integration in performance and in the density of elements used since it composes circuitry only at the level of logic functions. It cannot dip below that. Monolithic circuits decompose operations to the level of transistors, allowing them constructions outside of FPGA possibilities.
- There are tasks in which FPGA implementation outperforms even multi-core processors or graphics cards, but it is not equal to them in many jobs. So, we need to distinguish what is worth solving in FPGAs and what is not. It is also a topic of the textbook you are reading.
- FPGAs have the disadvantage of a more demanding design than classical programs. The debugging of program source code is faster. However, the increased laboriousness of the circuit solution of a given problem, or a sub-part of it, will bring a distinct advantage. It will save CPU time and power consumption, which will be especially beneficial for battery-powered electronics.

FPGAs are sold stand-alone or on development boards that can be used immediately and directly embedded into end-user or user devices.

For example, we present the DE2-115 part of the Terasic VEEK-MT2 development board, developed for teaching. It is used not only in our faculty but also in hundreds of leading universities worldwide. It contains many additional elements suitable for solving student problems. At its core is an Intel EP4CE115F29 FPGA, about which more will be discussed on p. 88.

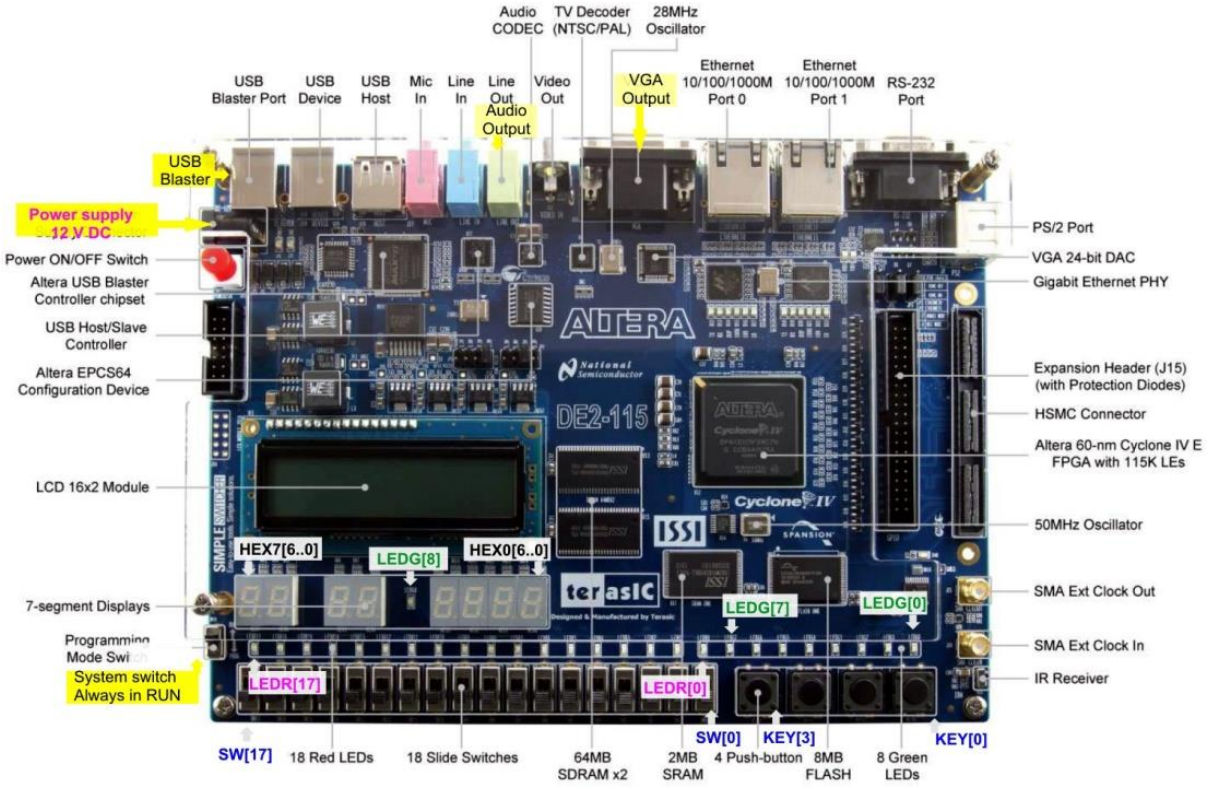


Figure 2 - DE2-115 part of the VEEK-MT2 development board (taken from Terasic)

Experimental devices can prefer boards without additional teaching elements, such as the de-

velopmental DE0 nano, which starts at $87 (in 2023). It is suitable for direct embedding into drones as well.

It contains an FPGA type EP4CE22F17, which offers the same configurable elements as the Cyclone IV from the previous DE2-115 board, only fewer of them (from 1/8 to 1/4). Still, FPGA itself costs ~$4, a fifteenth of the price of its advanced FPGA counterpart, and can also use a free version of the Quartus development environment.

Note: In the picture below, the suffix "C6N" after the FPGA part specifies an internal manufacturer code specifying the speed *grade of the circuit. Here, C6N means fastest.*

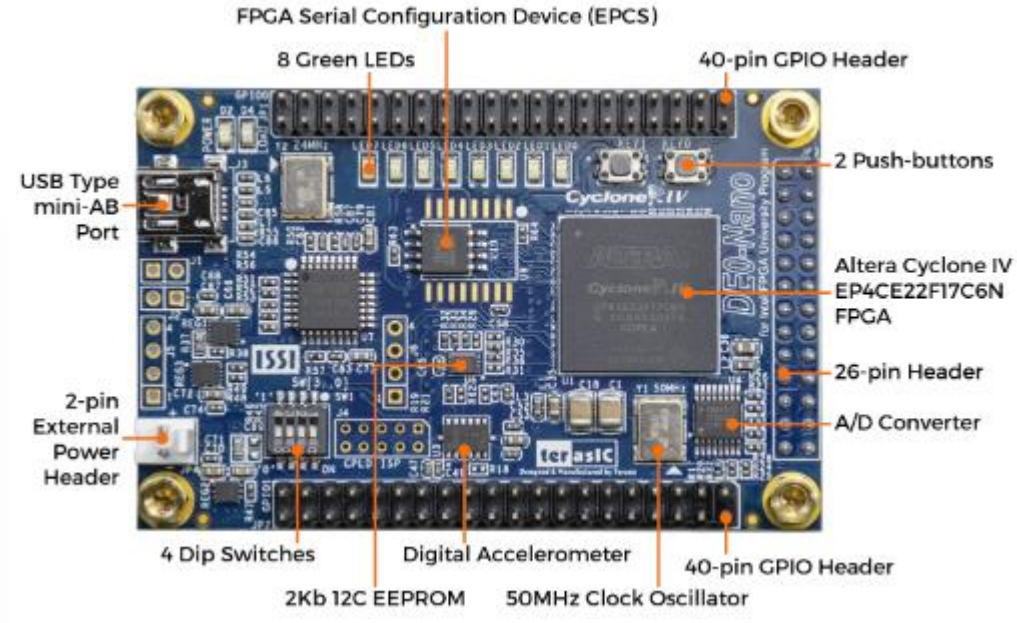


Figure 3 - Front side of the DE0 nano development board (Adapted from Terasic)

1.4 History of the text

The Faculty of Electrical Engineering (FEL), part of the CTU in Prague, teaches the courses "Architecture of Computers" (APO) and "Logical Systems and Processors" (LSP), in which my home Department of Control Engineering participates.

During their many years of teaching, I have created many tutorials. I gradually selected the most essential topics from these, which I expanded. The resulting textbooks allow me to keep the teaching tolerable and yet broader in scope but still understandable to all. It is impossible to include passages intended for complete beginners in college lectures. Exposing trivialities aimed only at them would take time away from the more exciting parts, and more experienced students would get bored.

Lectures will be more attractive if they assume knowledge of the more manageable parts, which are not demanding to understand, for which only reading the text is sufficient. In the interpretation, the main themes will be approached first and foremost. More curious listeners will find their extension from our textbooks. They were primarily designed for reading in sequence, but they can be used as handbooks.

I first created the **APOLOS** prerequisite in 2012 with the summarization of the minimum entry knowledge required to pass the LSP and APO. Its first half discussed simple logic circuit concepts, and the second half discussed number coding.

In 2019, I finished another textbook introducing concurrent style VHDL, which I followed up

in 2021 with another volume dedicated to behavioral style VHDL, explaining both VHDL and circuit elements. Its pages grew and were approaching a hundred, and it still didn't clarify the necessary things. And its clarity was diminished by mixing circuit engineering with VHDL explanation.

I decided to rearrange my study materials. I moved the first half of APOLOS to the beginning of the new textbook, **Logic Circuits on FPGAs**, and expanded with application sections. After these, I inserted circuit techniques from the unfinished VHDL behavioral style textbook. I have added passages from basic logic to make a comprehensive textbook for the LSP subject. It deals only with circuit structure, so it can be used elsewhere, such as in courses at our college that use Verilog instead of VHDL.

The sections on number coding in APOLOS have not changed. They are now stand-alone, only containing material common to LSP and APO. The change has also clarified the subsequent textbooks, which now discuss VHDL code styles without lengthy inserts on circuitry.

The VHDL textbook is currently being updated. Until now, we have been forced to use the 1993 version of VHDL because newer compilers did not support our older tutorial boards. We now have more contemporary development boards and can switch to the more convenient VHDL 2008 version.

The textbook "**Circuit design in VHDL 2008 for C programmers**" is currently under development. As its title suggests, it will attempt to explain the differences in circuit design to anyone who knows the C programming language, which is all of our students.

1.5 Acknowledgments

I want to thank everyone who contributed to the improvement of the textbook with their comments and advice.

My gratitude goes to Ing. Jaroslav Houdek and Ing. Jan Kelbich, the developers in practice, who willingly provided me with expert proofreading, during which they found several egregious errors.

I would also like to acknowledge my students who used the unfinished version of the textbook and sent me typos and errors to correct.

I am sure there are still other flaws in the textbook, and I would appreciate it if you could bring these to my attention.

Richard Šusta

2 Logical functions

Everybody is probably familiar with logical '1' (TRUE) and logical '0' (FALSE), at least from programming languages where there are types called Boolean or bool, and with logical operations as unary NOT (negation), AND (logical product) and OR (logical sum).

Logic diagrams represent them by graphical elements and describe the evaluation of expressions by tree. Its nodes, called gates, implement a given operation. They send to their output a value determined by their immediate inputs. The diagram thus describes the flow of data in the hardware.

T**he NOT operation** is often abbreviated to a bubble at the output of a gate.

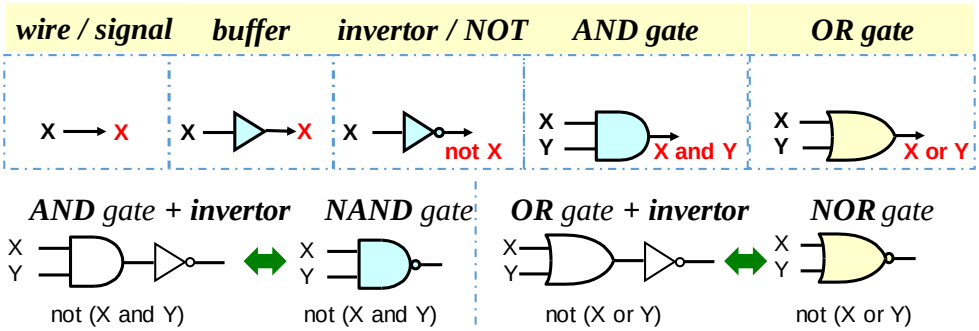
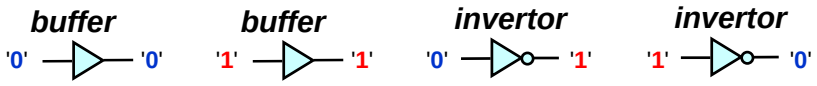


Figure 4 - Basic logical operations and their symbols

We introduce the ordering of logical values by the rule **'0' < '1'**; in other words, logical '0' is less than logical '1'. With this, we can easily remember the basic operations of logic:

- **The buffer** or **wire** copies its input value to its output. If it is just a connection, it is physically realized by a wire. Sometimes, an electronic element is used for decoupling, obtaining a higher output current, or changing the voltage level. In such cases, the fact is indicated by the term *BUFFER*.
- ***The NOT* logic function**, implemented by an inverter, aka negation or complement gate, changes the minimum '**0**' to maximum '**1**' and maximum '**1**' to minimum '**0**'.



- **The logical AND function** sends a logical '**1**' to its output only when both its inputs are in logical '**1**'. Thus, it performs a **minimum value selection of its inputs**, i.e., if any of its inputs are in logic '**0**', then AND sends a minimum value of '**0**' to its output.
- **The logical OR function** behaves as an inverse to the AND function. Its output will only be a logical '0' if all its inputs are in logical '**0**'. It thus implements the **selection of the maximum value of its inputs**, i.e., if any of its inputs are in logic '**1**', then the maximum value will be '**1**'.

Remember:

- AND (the choice of minimum) has '**1**' output only for one combination of its inputs, when all are at logic '**1**', i.e., all inputs have the maximum value.
- OR (the choice of maximum) has '**0**' output only for one combination of its inputs, when all are at logic '**0**', i.e., all inputs have the minimum value.

Understanding the logical functions AND and OR as selections of minima and maxima allows their straightforward extension to any number of inputs.

- **A logical** AND function **with n inputs** F=and(x_{n-1} ,...,x_1 , x_0), selects the minimum values of all its n inputs. F will be at logical '**1**' if and only if it has all its inputs x_i = '**1**'. If it will be a logical '**0**' on one or more inputs, then the minimum is '**0**'.
- **A logical** OR **function with n inputs**, G=or(x_{n-1} ,...,x_1 , x_0), selects the maximum of the values of all its n inputs. G will be at logical '**0**' if and only if all inputs x_i = '**0**'. If there is a logical '**1**' on one or more inputs, then the maximum will be '**1**'.

If our item contains all the variables of some specified set, it is called **minterm** when they are concatenated by AND operators and **maxterm** when using OR. *Note: For the general concept, see p. 37, where we extend the terms to the more general implicants.*

Examples:

(X and Y and Z) - is a **minterm** that outputs '**1**' only for all inputs in '**1**', otherwise '**0**'.

(X and not Y) - is a **minterm** giving '**1**' when X='**1**' and Y='**0**' (not Y='**1**').

(not X or not Y) - is a **maxterm** that outputs '**0**' only for X='**1**' and Y='**1**', otherwise '**1**'.

(not X or Y or not Z) - is a **maxterm** giving '**0**' only for X='**1**', Y='**0**' and Z='**1**', otherwise '**1**'.

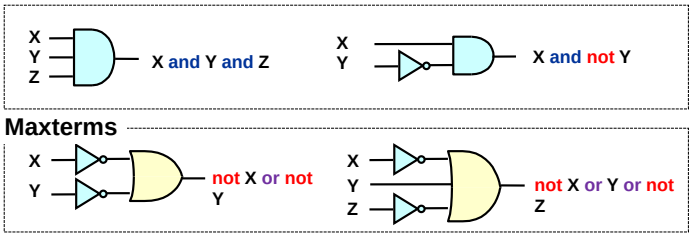
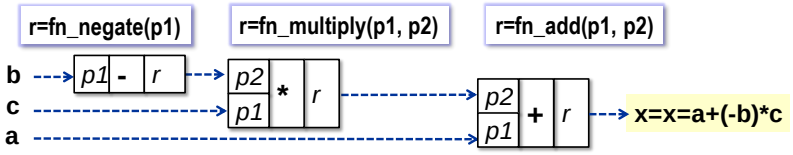


Figure 5 - Implementation of minterms and maxterms s

The minterm can be connected with one AND gate, while the maxterm with an OR gate.

2.1 Operators and logical functions

Let's take a common mathematical expression, for example, x=a+(-b)*c. The syntactic parser must convert the shorthand operators and organize their evaluation into chained function calls according to their priority. An expression tree can express its result:



Mathematically, the tree is written: x = fn_add(**a**, fn_multiply(**c**, fn_negate(**b**))). The unary function fon_negate has one input parameter p1, and returns r (result), while the remaining functions are binary, i.e., they have two input parameters p1 and p2.

2.1.1 Logic diagrams

A logic function can be written as an expression, or we can express it graphically by a logic diagram or a flowchart that indicates their evaluation procedure. The operations here are characterized by graphical symbols of the elements used, which are interconnected.

For technical reasons (characters on the computer keyboard), the AND operation is often writ-

ten in the logical expression with the symbol **+**. The unary NOT is characterized by a postfix apostrophe. A summary of the various NOT, AND, and OR notations, i.e., established formalities, can be given for comparison.

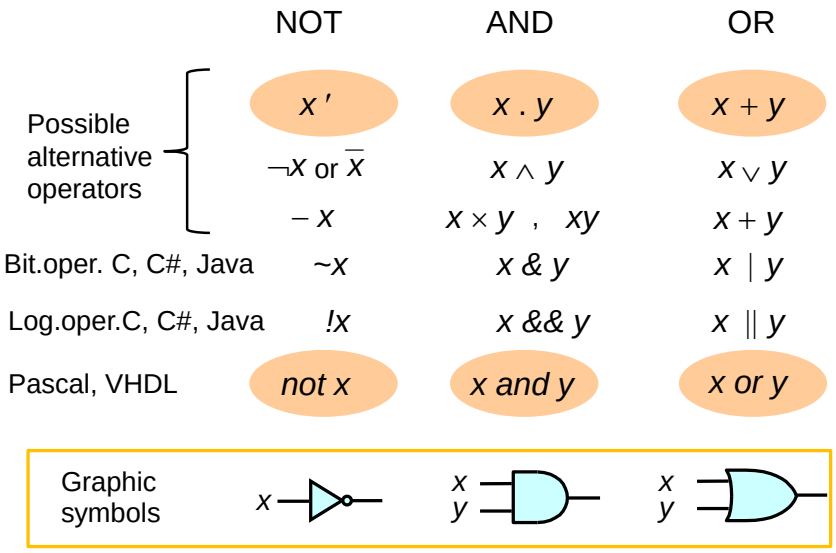


Figure 6 - Logical Operation Operators

In the programming languages C, C#, and Java, the logical operators !, &&, and || are evaluated until the result is explicit. Bitwise operators are evaluated entirely, making them more like logic in which each element operates concurrently.

For example, let the logical function be Y= (not (A and B)) or (C and D). Since unary operations generally have higher precedence than binary operations, we omit the bold red brackets and write the function as Y= not (A and B) or (C and D), respectively, also using the operator abbreviations "+" , "." and "´" (the apostrophe denotes negation), as Y= (A . B)´ + (C . D).

Its evaluation can start, for example, with the left AND operation: λ_0 =A.B [λ_0 =A and B], where λ_0 denotes its intermediate result. Then its negation λ_1 =(A.B)´ [λ_1 =not (A and B)] is performed. Next, another AND operation is computed: λ_2 =C.D [λ_2 =C and D]. Finally, the two intermediate results λ_1 and λ_2 are combined by OR operation to Y =λ_1 +λ_2 = (A.B)´ + (C.D) [Y = not (A and B) or (C and D)].

The evaluation procedure is shown in Figure 7. At the top, the individual operations are written with the names of logic functions. At the bottom, the same scheme is drawn more commonly using schematic markers for logical operators, i.e., logic gates.

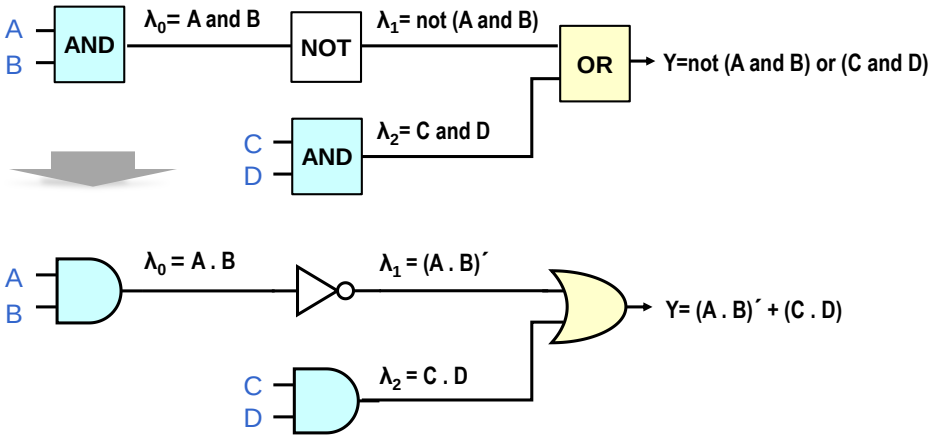


Figure 7 - Logic diagram and its logical expression

2.2 The laws of Boolean logic

Boolean logic contains two values, the familiar logical '**0**' and '**1**', two binary operations AND and OR, and one unary NOT.

Here, we must point out:

1. In Boolean logic, both AND and OR operations have the **same precedence!** In many programming languages, the AND operation has priority over the OR operation to simplify writing expressions. However, we must not assume AND precedence in logical manipulations; otherwise, we will get incorrect results. It is advisable to add parentheses.

   Example: previous function Y (Figure 7 on p. 14) would be written in C using the statement: Y=!(A && B) || C&&D. In Boolean logic, however, the precedence must be indicated by parentheses Y=(A.B)' + (C.D). We will prefer in this textbook the usage of unambiguous verbal operators: Y = not (A and B) or (C and D).

2. Boolean logic knows only '**0**' and '**1**'.
3. Boolean logic is the reduction of more general Boolean algebra that utilizes more values than '**0**' and '**1**'. Tables define the NOT, AND, and OR operations in it. Up to nine values are commonly used in designs. When explaining the three-state gate, we will discuss one additional value, 'Z', on page 65.

We will use Boolean logic in the text, i.e., just two values, '**0**' and '**1**'.

Boolean logic satisfies Huntington's postulates. They are accepted without proofs as a theoretical basis and specify the minimum requirements to be Boolean logic at all. Other theorems can then be derived from them.

In practice, clumsy algebraic modifications of logic functions are hardly used because of their complexity, which increases the risk of error. Safer methods exist. However, the basic rules of Boolean logic are of indispensable importance in the design of logic circuits. We will present mainly their applications, i.e., what a given theorem or postulate allows us to do in circuits.

The first postulate is **closure**, i.e., the result of any operations with logical '0' and '1' will be again '0' or '1' in Boolean logic. Nothing else will appear in it.

The next postulate is **commutative law**.

Postulate	*OR version*	*And the versions*
Commutative Law	$x + y = y + x$	$x \bullet y = y \bullet x$

Using multi-input gates, we can connect the signals to gate inputs in any order. The result of the operation will be the same in every case.

In programming languages, the result may sometimes depend on the order in which the members of a commutative expression are listed due to side effects because they are evaluated sequentially. In logic circuits, however, everything runs **in parallel**. It is a fundamental property of logic circuits. All components run concurrently, as each element was evaluated on a separate processor core. Applications that simulate circuit behaviors emulate this in many ways, for example, queuing events, sometimes processed in random order, such as ModelSim.

We get the associativity theorem if we substitute expressions in place of the variables x and y into the commutativity theorem>

Theorem	*OR version*	*And the versions*
Associativity Associative Law	$a + (b + c) = (a + b) + c$	$a \bullet (b \bullet c) = (a \bullet b) \bullet c$

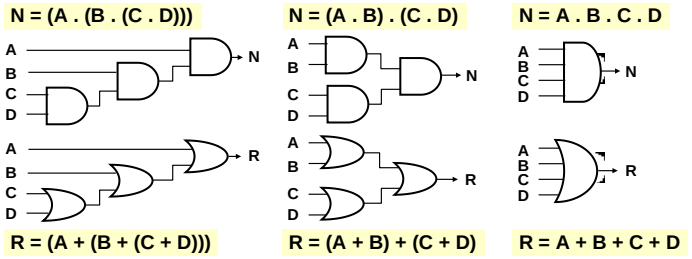


Figure 8 - Associativity

If gates have fewer inputs than we need, we can create multi-input OR or AND gates by connecting them. From the point of view of logical operation, it won't matter how we layer them. On the other hand, associative decomposition can also speed up the operations, as we will show later with the addition and subtraction of constant 1 in Chapter 6.1.2 on p. 103.

But the cascade interconnection, shown on the left, may not even evaluate slower because of the longer path from input D to output N or R. The design environment that will stand between our circuit description and its implementation inside the FPGA minimizes our design. It can implement the final circuit completely differently but with the same functionality. Thus, all ways of obtaining R and N drawn above give our desired result, which is the most important.

Postulate	*OR version*	*And the versions*
Distributive Law	$x + (y \bullet z) = (x + y) \bullet (x + z)$	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$

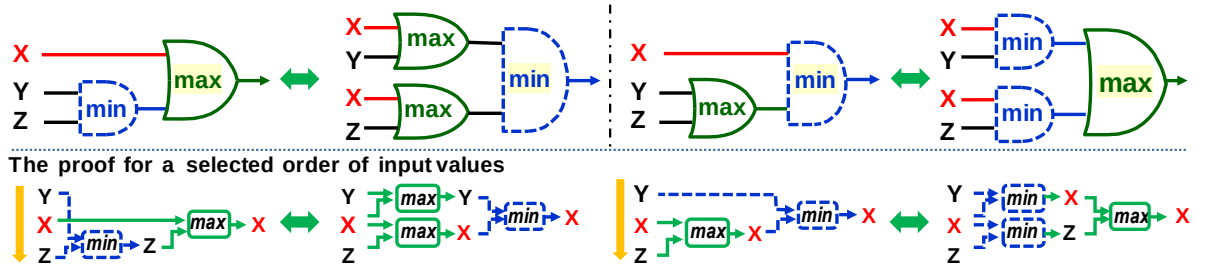


Figure 9 - Distributivity

The distributivity theorem shows that the AND and OR operations have equal status. For example, its proof can be done by checking all six possible size cases[2] of the three input values X, Y, and Z. The calculation at the bottom of the figure above demonstrates one chosen randomly, where Y has the maximum value and Z is the smallest. Here, we have considered a Boolean algebra with multiple values to emphasize that the distributivity of the minimum and maximum operations holds not only in two-valued Boolean logic but in any domain in which the ordering is defined by introducing a ≤ relation, perhaps even for real numbers.

[2] For proof, see for example:
https://proofwiki.org/wiki/Max_and_Min_Operations_are_Distributive_over_Each_Other

The symbols ● and + elsewhere denote arithmetic multiplication and addition, which are not mutually distributive, so the theorem looks unnatural from their point of view. In logic, however, they represent the minimum (AND) and maximum (OR) selection operators with the same precedence.

Postulate	*OR version*	*And the versions*
Complementation	$a + a' = '1'$	$a \bullet a' = '0'$

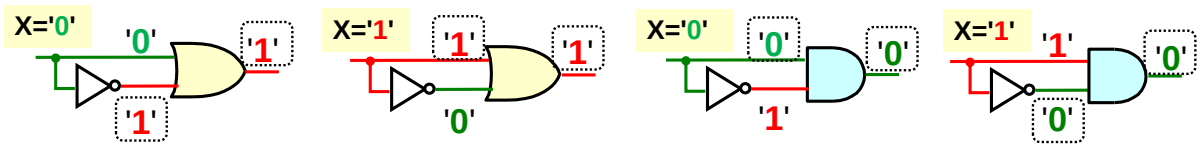


Figure 10 - Complementarity

When an input and its negation are simultaneously connected to a gate, i.e., a value and its complement, one will surely be logic '0' and the other '1'. The OR operation (selecting the maximum) chose '1', which will be its constant output, while the AND operation, as the selection of minimum, will always result in '0'.

We will use complementarity later to minimize logic functions.

Postulate	*OR version*	*And the versions*
Identity Law	$x + '0' = x$	$x \bullet '1' = x$

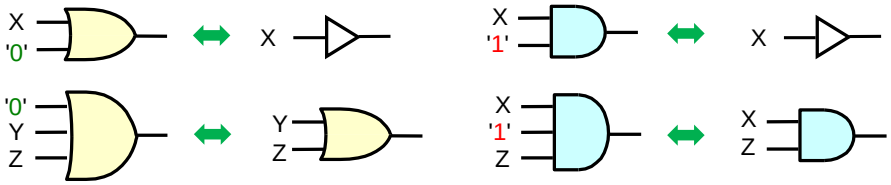


Figure 11 - Identity

The identity law presents the property of selecting the **maximum** (OR). If any input is at the minimum, i.e., at '0', it cannot change the results given by the other input values. In the case of **minimum** selection (AND), any input at the maximum, i.e., at '1', cannot affect the result.

Theorem	*OR version*	*And the versions*
Annulment Law	X + '1' = '1'	X ● '0' = '0'

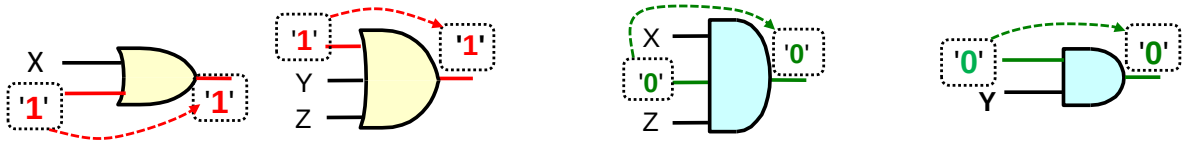


Figure 12 - Aggression

Annulment law also follows directly from the AND and OR functions. If one OR input has a maximum value of '1', the output will always be '1' regardless of the other inputs. The possible maximum has already been reached. Analogously, when any input is minimum, i.e., at '**0**', the AND chooses its value, i.e., '0', and the other inputs do not influence the result.

Remember:

- **AND** (the **minimum** selection) has identity value '**1**' and '**0**' as aggressive annulment.
- **OR** (the **maximum** selection) has identity value '**0**' and '**1**' as aggressive annulment.

Theorem	*OR version*	*And the versions*
Idempotence Idempotent Law	x + x = x	x ● x = x

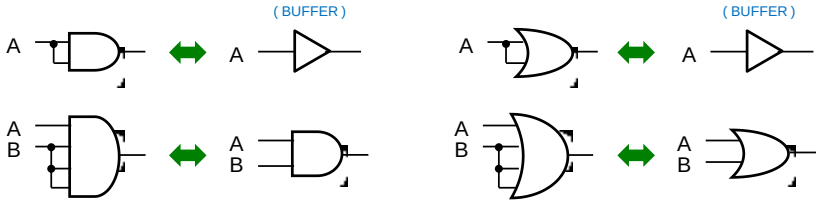


Figure 13 - Idempotence

The AND and OR operations are idempotent, i.e., repeated usage of the same input produces the same output, similar to its single use.

In practice, we can also use any multi-input gate with fewer inputs. We connect the wire of more inputs. Here, we can also connect the redundant inputs to the neutral element of the operation, to '1' for AND and to '0' for OR. The result will be identical. It is up to us what we like.

Theorem	
Double negation Double negation	not (not x) = x

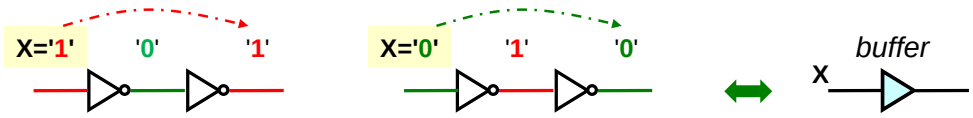


Figure 14 - Double negation

Double negations are nullified, so they behave like a buffer in terms of pure logic. A bubble at the output often truncates the inverter.

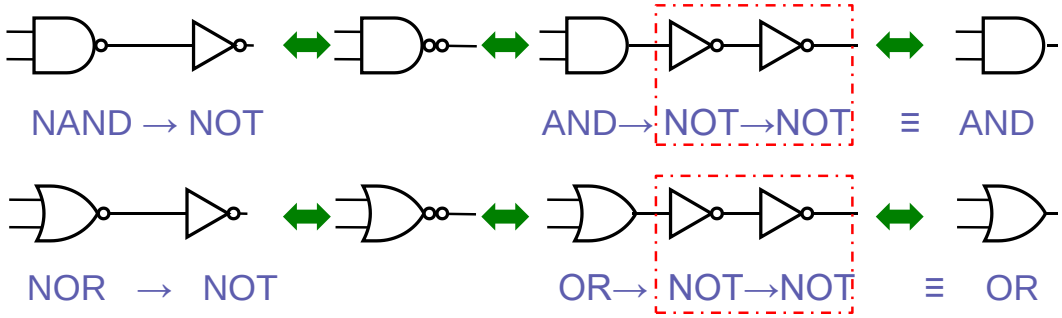


Figure 15 - Double negation at gates

Double negation is used to manipulate gates, especially in conjunction with De Morgan's theorem, which we will present as the following theorem.

The bubble sometimes replaces inverters located before the input or after the output of a func-

tion, especially in space-saving schemes.

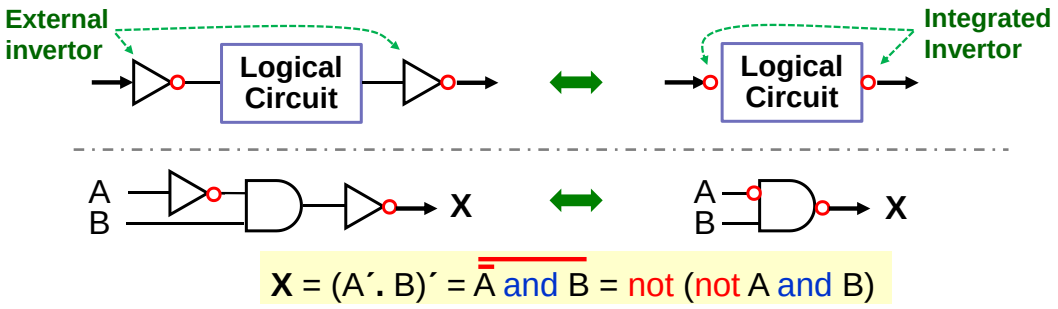


Figure 16 - Bubble as inverter shortcuts

Theorem	*OR version*	*And the versions*
De Morgan	not (x + y) = not x ● not y	not (x ● y) = not x + not y

DeMorgan's theorem is frequently applied, allowing breaking down the NOT before the parenthesis. Its validity can be proved in several different ways. The easiest one is to create a truth table of the logical function on its left and right sides.

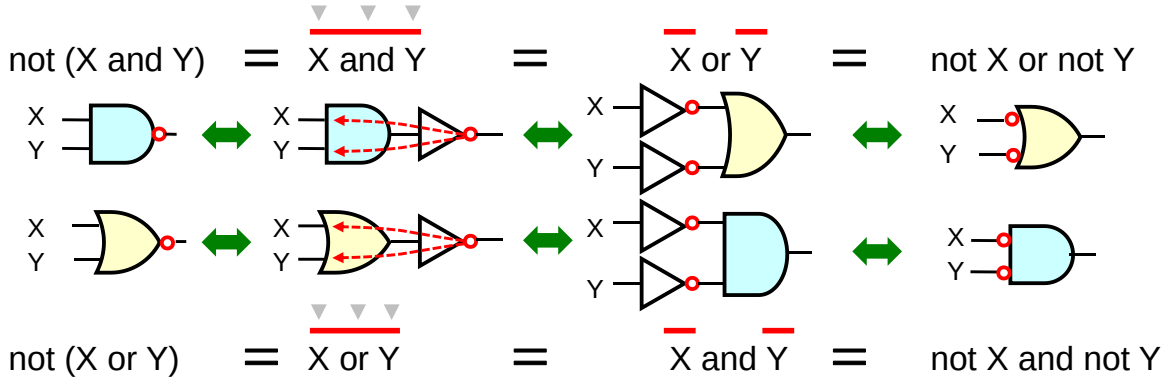


Figure 17 - DeMorgan's theorem

De Morgan's theorem can also change the type of a logical operation. On bipolar transistors used in TTL logic, the NAND gate can be more easily implemented by a multiple-emitter layout. The first ones were produced in 1963. Because of this, all logic functions were converted by inserting double negations and breaking them down to compose logical functions only from inverters and NAND gates. An analogous procedure can convert them to exclusive NOR and inverters.

In modern designs, there is no need for gate conversions. The development environments reorganize them anyway according to the available finite elements.

However, changing the gate type is utilized at the internal integrated circuits level because gates with output negation flip faster, as we will discuss later in Chapter 4, which deals with internal CMOS structures.

The modification principle involves moving the negation bubble through an AND/OR gate. Its type is changed to the opposite through it, and the inputs/outputs are inverted. However, the negation bubble must always originate from or terminate at all its inputs.

We can also insert two bubbles in a series if it suits us. According to the double negation theorem, see p. 18, they cancel each other. Then, we move one bubble through the gate.

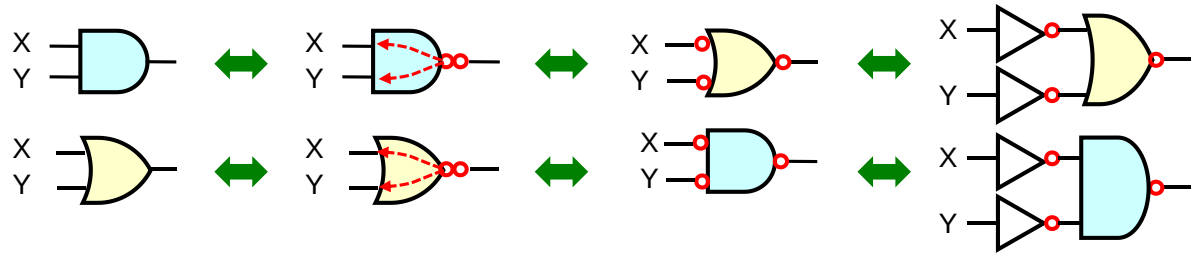


Figure 18 - Conversion between AND and OR gates

By this way, we perform a graphical application of De Morgan's theorem. When applying it to an expression, we must keep the order of execution of the logical operations. Let's have a three-bit equality test. They must all be in logic '1' or '0'. The function is easy to write:

EQ3(X,Y, Z) = X . Y . Z + X' . Y' . Z'

But we have here a **misleading entry**! In Boolean logic or algebra, functions AND and OR have the same precedence. The preceding expression mimics programming languages that artificially introduce a preference for AND over OR to simplify the writing. In logic, we should specify the order of operations with parentheses and change the post-fix negation to the unary not operator.

For further interpretation, we write down the expression using unambiguous word names for the operators:

EQ3(X, Y, Z) = (X and Y and Z) or (not X and not Y and not Z) (1)

Now, we can see the possibility of pointing out not before the highlighted bracket. We insert two not operators in front of the bracket and break out the second one by De Morgan's theorem. Inside the parenthesis, we change and to or and remove the negation of the terms (by the double negation theorem):

EQ3(X, Y, Z) = (X and Y and Z) or not (not X and not Y and not Z)

= (X and Y and Z) or not (X or Y or Z) (2)

We also create a negated function NEQ3 by adding a negation to (2). Note that we apply De Morgan's theorem to the terms of the expression, which are the logical functions here! Two not operators are canceled, again according to the double negation theorem (p. 18):

NEQ3(X,Y) = not ((X and Y and Z) or not (X or Y or Z)) (3)

= not (X and Y and Z) and not (X or Y or Z)

= not (X and Y and Z) and (X or Y or Z) (4)

We could also decompose the left term of relation (4) by De Morgan's theorem into:

NEQ3(X,Y) = (not X or not Y or not Z) and (X or Y or Z) (5)

But the function (5) is more complicated, so we prefer to leave it in the form (4). We can easily verify the validity of the expression (4) by considering:

- Maxterm (X and Y and Z) be in '1' only when X='1', Y='1', and Z='1', hence its negation is '0'. The whole NEQ3 is '0' due to the annulment theorem for AND operation.
- Maxterm (X or Y or Z) is '0' only when X='0' and Y='0' and Z='0', hence also NEQ3.
- NEQ3 will be at '1' in all other cases, reporting that the three bits are not identical.

The application of the theorem to EQ3 can also be done graphically. The numbers in the fig-

ure below correspond to the previous equations.

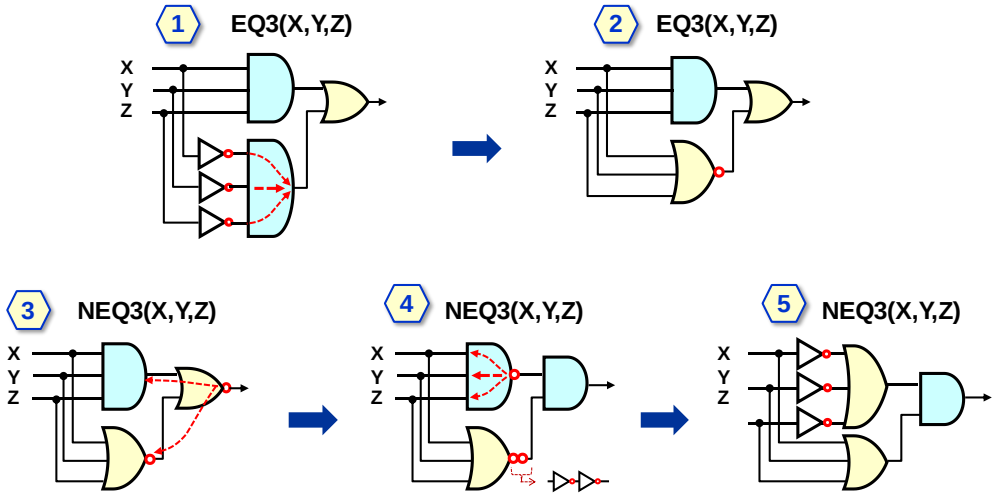


Figure 19 - Graphical application of De Morgan's theorem to EQ3

2.3 Logic functions of one and two input variables

Figure 20 shows all the logical functions of **a single input variable**, including their schematics symbols. We already know two of them: Wire/Buffer and NOT/Inverter. The two remaining represent constants '0' and '1', namely F0 always having a logical '0' at the output and F3 having a permanent logical '1'.

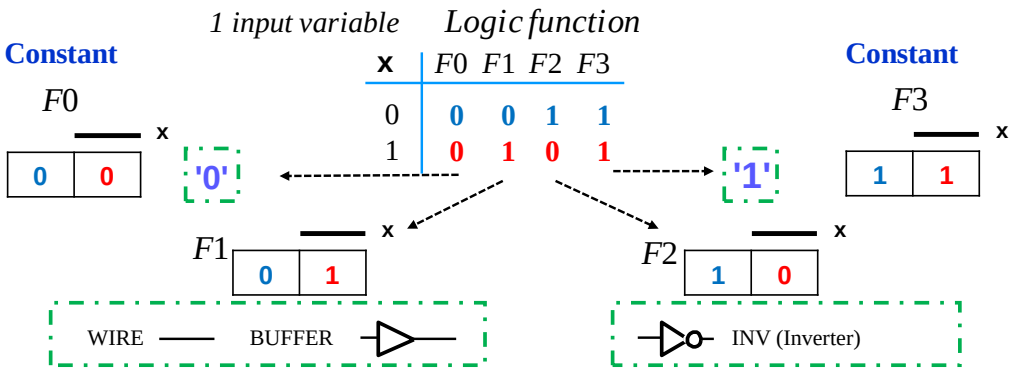


Figure 20 - Logic functions of one input variable

The constant function of logic '0' is commonly called Gnd (Ground) in schematics, while logic '1' is specified by Vcc. These are traditional markings dating back to the days of transistors wired with a common collector. Development environments have retained them.

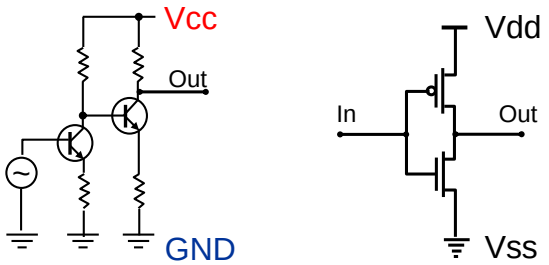


Figure 21 - Voltage designations in circuits

The following voltage symbols are utilized (in the most common positive voltage logic in computers today, where logic '1' is the higher voltage and logic '0' is the lower voltage):

GND ground - the symbol of common zero, which is usually 0 V in circuits. Inputs that are supposed to be permanently at logic '0' are connected to GND in schematics.

V_{CC} (aka U_{CC} in some literature) comes from Common Collector Voltage. It is the generally accepted abbreviation for the supply voltage used. Inputs permanently at logic level '1' are connected to V_{CC} .

Note: The size of the V_{CC} depends on the used component. It can be 24 V (industrial logic), 5 V (TTL), 3.3 V (LVTTL), or 1.2 V (in some CMOS), but even smaller, e.g. 0.6 V on 7 nm CMOS technology.

V_{DD} comes from Voltage-Drain Voltage CMOS circuits. In some publications, it is used instead of V_{CC} , as it is more accurate today since logic circuits are mainly CMOS-based. However, many programming tools use the traditional V_{CC} for supply voltage, so we also prefer it.

V_{SS} Voltage for Substrate & Sources represents the lowest voltage of CMOS circuits, which could also be minus for some types.

Figure 22 below shows all the **logical functions of the two input variables**, again including their schematic symbols. If you look at it, you will see 16 of them, but 6 with italic blue font can be replaced by logical functions of one variable.

The F0 and F15 functions do not depend on inputs. These are the GND and Vcc constants known from Figure 20, only in the two-input version.

Other functions, BUFX, BUFY, INVX, and INVY, depend on only one input so that they can be replaced by BUFFER or inverter, INV* elements, for the input affecting the output.

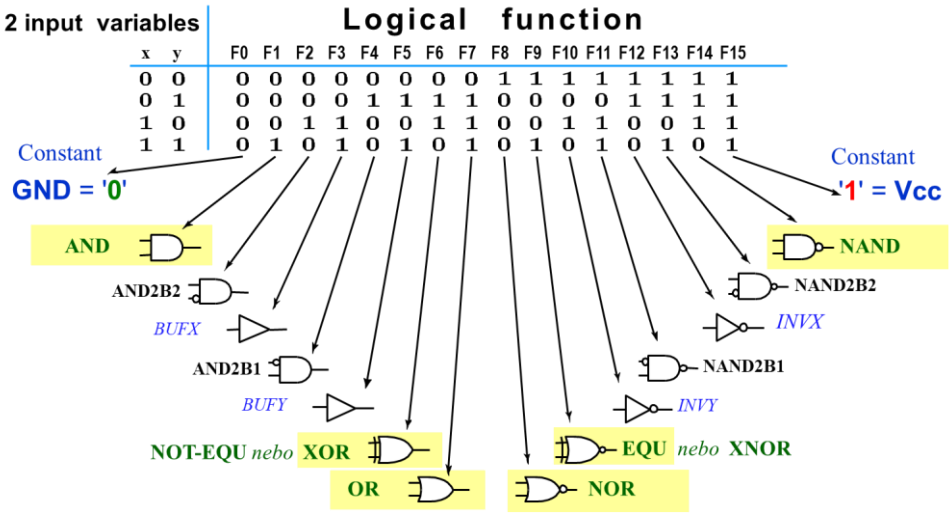


Figure 22 - Logic functions of two input variables

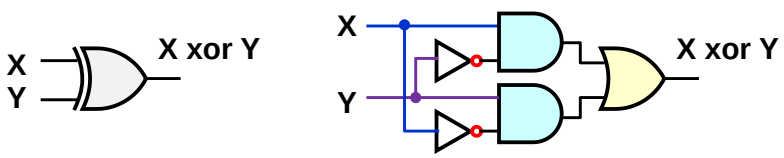
Of the remaining 10 logic functions, only 6 are used in practice, as they are easy to remember – they are highlighted in yellow in the figure, namely AND, XOR, OR, NOR, XNOR, and NAND. The other unmentioned functions do exist, but that is usually where their contribution ends. If they are needed, we write them by logical expressions.

2.3.1 XOR function

The XOR logic function is used so often that we describe it in a separate chapter.

- **The logical XOR function**, eXclusive OR (exclusive OR), gives '1' on its output if it has an odd number of its inputs in logical '1'.
- The two-input operator **xor** exists in almost all design environments. The programming languages C, C#, and Java use binary bitwise operation ^. Mathematics prefers more precise notation ⊕. However, the last symbol is not on computer keyboards, so the operator is often written as xor.
- We can create the logical XOR by NOT, AND, and OR operations. An odd number of entries in '1' occurs only in X='1' and Y='0', or in X='0' and Y='1'. We will describe both of these as minterms.

$$X \text{ xor } Y = (X \text{ and not } Y) \text{ or } (Y \text{ and not } X) \qquad (6)$$



We obtain an exciting property of XOR by setting one of its inputs to '0', alternatively to '1'.

No matter which input we use, XOR is a commutative operation.

X xor '0' = (X and not '0') or ('0' and not X)

= (X) or ('0') = X (7)

X xor '1' = (X and not '1') or ('1' and not X)

= ('0') or (not X) = not X (8)

We can see that the XOR can serve as a controlled element that is either a buffer when its second input is in '0', or an inverter by setting it to '1'.

Figure 23 - XOR as a controlled inverter

Using two-input XOR in circuits:

- Frequent application of XOR gate is switching between buffer or inverter behavior for the second input. In Chapter 6.1.1., we create modification addition to subtraction by XORs.
- Furthermore, XOR is the primary member of binary adders. If we do not consider transfers to higher orders, then (in this paragraph, + will be binary addition) '0'+'0'='0' and '1'+'1'='0', while '1'+'0'='1', '0'+'1'='1'. In other words, a binary sum is a logical '1' only when an odd number of inputs in '1' is just a property of XORr.
- XOR allows the detection of inequality. Its output is '1' on the odd number of inputs in '1'. It happens only for X='1' and Y='0' or for X='0' and Y='1'; i.e., X and Y are different.

Negated XOR, i.e., XNOR, eXclusive NOT OR, gives a logical '1' on even inputs at '0', i.e., its both inputs have the same value. Thus, notation EQU, EQUivalency, is sometimes used.

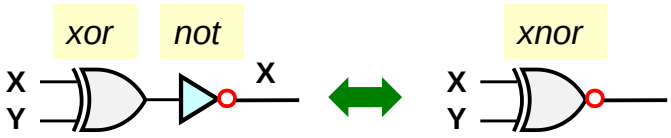


Figure 24 - Functions xnor from xnor

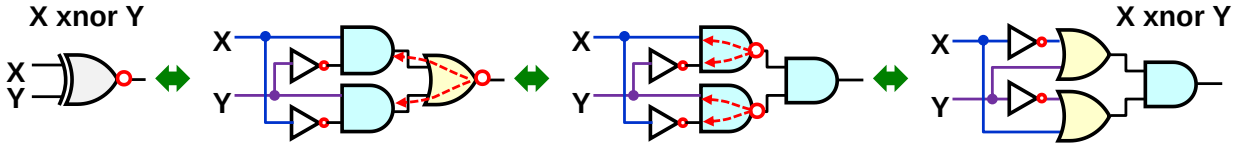
Repeated usage of equ (xnor), **does not give equality of input**. The expression X equ Y equ Z will be equal to '1' only if the even number inputs are in **'0'**, i.e., two or none. Because of this, we consider XNOR as the more accurate notation.

De Morgan's theorem relates to NOT, AND, OR operations. We cannot apply it to the function XOR as a whole. XOR is a composite operation given by the logical equation. We must de-compose XOR and apply the theorem to its terms. We start from expression (6)

X xnor Y = not (X xor Y) = not ((X and not Y) or (Y and not X)) (9)

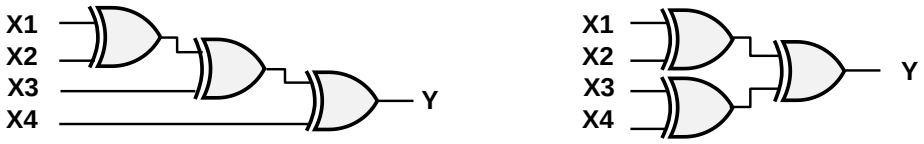
= not (X and not Y) and not (Y and not X) (10)

= (not X or Y) and (not Y or X) (11)



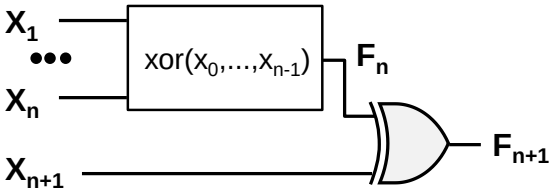
The XOR logic function can also extend to more inputs, returning the '1' on an odd number of its input values in '1'. We can connect its two-input XOR in any way we want to obtain a multi-input version. The result will always be the same.

Y= ((X1 xor X2) xor X3) xor X4 = (X1 xor X2) xor (X3 xor X4)



Multi-input XOR is more commonly called **parity**, i.e., by its most frequent application. The parity means appending an extra bit to the transmitted or stored data word. By this way, we always obtain the total number of bits in '1' even or odd, including the parity bit. XOR calculates even parity; it adds '1' when the number of data bits in '1' is odd.

For example, we can prove it by mathematical induction for the arrangement on the left in the figure above.



- Suppose we have a logic function Fn for n-inputs, n>=2, which returns '1' on an odd number of its inputs in '1'. We know such a function as a two-input XOR.
- If F_N outputs '1', then an odd number of X_1 to X_N inputs will be in '1'. F_{n+1} gives '1' if and only if X_{n+1} is in '0'. Thus, the odd number of inputs in '1' is preserved after the extension of X_{N+1} .
- If F_N has an output of '0', then the even number of X_1 to X_N inputs will be in '1'. F_{N+1} will output '1' only if X_{N+1} is in '1', i.e. if there are an odd number of inputs in bits X_1 to X_{N+1}; otherwise, it is '0'.
- The previous considerations lead to the conclusion that even F_{N+1} will be in '1' if and only if there are an odd number of entries in '1', which we wanted to prove.

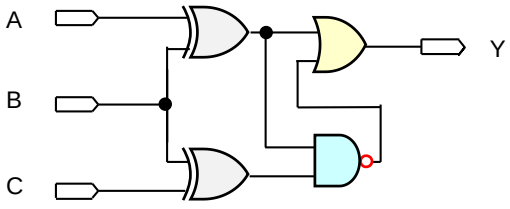
The proof can be done analogously for the tree structure. For it, we arrive at the same result.

Similarly, we can also show that if we use XNOR in the figures above instead of XORr, then the output will be in '1' for an even number of inputs in logical **'0'**.

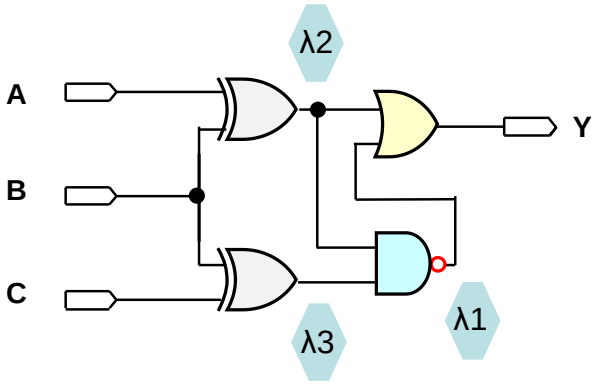
2.4 Converting a logical schema to its expression

We will conclude this section by showing how to convert a logical schema into its logical expression. Since a schema represents a procedure for evaluating a logical function, we only need to step through it and write the operations, as demonstrated in the following example.

Example: Write a Boolean expression corresponding to the Boolean function in the figure:



Solution: We can construct the logical expression from the left, i.e., in the direction of its evaluation, or from the end. We will show the second way, which is more versatile. It can also convert complex circuits with inner loops. First, we mark the outputs of blocks.



The output of Y is an OR function:

$$Y = \lambda 2 \text{ or } \lambda 1 \qquad \text{(eq1)}$$

λ1 is given by an AND function with a negation bubble, so λ1 = not (λ2 and λ3). We substitute the relation for λ1 into (eq1) to get:

$$Y = \lambda 2 \text{ or not } (\lambda 2 \text{ and } \lambda 3) \qquad \text{(eq2)}$$

Next, we calculate λ2 = A xor B and add it after its two occurrences in (eq2)

$$Y=(A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } \lambda 3) \qquad \text{(eq3)}$$

All that remains is to determine λ3 = B xor C and insert this into equation (eq3) to obtain the result:

$$Y=(A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } (B \text{ xor } C)) \qquad \text{(eq4)}$$

The equation (eq4) can also be written using symbols for operators. We preserve only xor.

$$Y=(A \text{ xor } B)+((A \text{ xor } B).(B \text{ xor } C))´$$

~o~

We know how to create a graphical diagram from a logical function and vice versa, but we still construct expressions more by intuition than by a method.

Even if we use design environments, we still need to specify the behavior of the logic function. We can certainly describe it by a list of '0's and '1's, but in many cases, the expression is faster. So, we will look at the methodology of how it can be created before we start explaining the circuit internals.

In the next section, we look at possible specifications of the values of logic functions, which we use to optimize their expressions using Karnaugh maps. With these, we will have everything we need to interpret the basic circuits later.

3 Logic Function Description

Let us have logical variables that take values only from some finite set B.

A Completely Specified Logic Function (CSLF) of **n input variables** $y = f(x_1, x_2, x_3, ...x_n)$ is called a view:

$$B^n \rightarrow B, \text{ where } (x_1, x_2, x_3, ..x_n) \in B^n, x_i \in B, y \in B.$$

If B contains only a logical zero and a one, B={'0','1'}, then it also has cardinality |B|=2 and defines a **two-valued logic**[3] (two-valued logic).

Cartesian product B^N creates all possible tuples from B; for |B|=2 is $|B^N|=2^N$. We assign them output values by mapping $B^n \rightarrow B$. We obtain 2^{2^n} different assignments for N>=0 logical variables. Each of them describes one logical function.

For N=0, there are only 2 constants GND='0' and Vcc='1'. For N=1, we have $2^{2^1} = 2^2 = 4$, for N=2 we have $2^{2^2} = 2^4 = 16$, for N=3, we get $2^{2^3} = 2^8 = 256$ logic functions.

Example: let's have B={'0','1'}. We write the logic function of the two inputs as y=f(x1, x2). The Cartesian product B^2 generates four pairs, i.e. B^2 = { ('0', '0'), ('0', '1'), ('1', '0'), ('1', '1') }. Let's choose one of them. We only assign a logical '1' to an output if there is an odd number of inputs in '1', which we know as the xor function. We define y=xor(x1, x2) by displaying:

xor: B2→B =		simplified notation	
	('0', '0') → '0'		0 0 → 0
	('0', '1') → '1'		0 1 → 1
	('1', '0') → '1'		1 0 → 1
	('1', '1') → '0'		1 1 → 0

The truth table is just another way of writing it: we write the view in it

such as

x1	x2	xor
0	0	0
0	1	1
1	0	1
1	1	0

or even like this:

x1	x2	xor
1	1	0
1	0	1
0	1	1
0	0	0

or else:

x1	x2	xor
0	0	0
1	1	0
1	0	1
0	1	1

All tables above specify the same logic function. The order of their rows doesn't matter. We can list them in any one, but all outputs must be defined. The physical implementation of a logic function requires knowing the output value for every possible combination of input values. *Note: HDL languages allow assigning all not yet specified by one statement.*

We define a **combinational logic circuit** as a list of m logic functions of the form:

$$y_k = f(x_1, x_2, x_3, ..., x_n); \text{ where } k=1 \text{ to } m$$

When some inputs change, temporary transients occur in the combinational circuit, but after they settle down, the outputs y_k appear. Its value depends only on the current inputs; in other words, the same values of inputs x_1 to x_n lead to the same values of outputs y_1 to y_m .

[3] When designing logic circuits we can assign more values than ´0´ and logic ´1´ to output In this text, we will soon introduce 3-valued logic by adding the value X (don´t care). In professional work, the 9-value logic MVL-9 is used a lot, and we will talk more about it in the textbook on VHDL.

Note: Sequential logic circuits are different; we present them in the final Chapter 7. They contain memory members, so their outputs depend on the sequence of previous input and data values in the memories. The same immediate inputs may give different outputs each time.

Listing all possible inputs is tedious, so several functions are often combined into one table. For example, we can write other common logic functions along with xor:

x1	x2	xor	xnor	and	nand	or	nor
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	0	1	1	0	1	0

Sometimes, we can reduce the number of rows. For example, interrupt processing must know the highest input xi in logical '1' to serve the most priority request.

For example, for 3 inputs, the function can have a table on the right.

- Output p3 is 00 if no input is in '1'.
- Output p3 goes to 01 if only input x1 is '1'.
- If x3 is at '0' and x2 is at '1', then the output of p3 is 10, regardless of the input of x1, because we want x2 to have a higher priority than x1.
- Output p3 will be 11 with the highest priority input x3 at '1' regardless of the state of the other inputs.

x3	x2	x1	p3		Index
0	0	0	**0**	**0**	**0**
0	0	1	**0**	**1**	**1**
0	1	0	**1**	**0**	**2**
0	1	1	**1**	**0**	**2**
1	0	0	**1**	**1**	**3**
1	0	1	**1**	**1**	**3**
1	1	0	**1**	**1**	**3**
1	1	1	**1**	**1**	**3**

We shorten the previous table by using the flag that the same output value is repeated for the same input bits in '1' and '0'. We replaced these input values with a - (dash) character representing a wildcard. For example:

x3	x2	x1	p3			x3	x2	x1	p3	
0	0	0	0	0	→	0	0	0	0	0
0	0	1	0	1	→	0	0	1	0	1
0	1	0	1	0	merge 2 rows →	0	1	-	1	0
0	1	1	1	0						
1	0	0	1	1						
1	0	1	1	1	merge 4 rows →	1	-	-	1	1
1	1	0	1	1						
1	1	1	1	1						

Table 1 - Input merging by wildcards

The new table (top right) has only 4 rows. The wildcards application shortens the entry by merging inputs, so it is a way to generate table rows.

We can now quickly write a more extensive function for 10 interrupt inputs that returns the number of the highest request. Instead of the 2^{10} =1024 rows we would have had to include when listing the entire table, we only needed 11:

x10	x9	x8	x7	x6	x5	x4	x3	x2	x1	p10				Index
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	1	1
0	0	0	0	0	0	0	0	1	-	0	0	1	0	2
0	0	0	0	0	0	0	1	-	-	0	0	1	1	3
0	0	0	0	0	0	1	-	-	-	0	1	0	0	4
0	0	0	0	0	1	-	-	-	-	0	1	0	1	5
0	0	0	0	1	-	-	-	-	-	0	1	1	0	6
0	0	0	1	-	-	-	-	-	-	0	1	1	1	7
0	0	1	-	-	-	-	-	-	-	1	0	0	0	8
0	1	-	-	-	-	-	-	-	-	1	0	0	1	9
1	-	-	-	-	-	-	-	-	-	1	0	1	0	10

The last row of the table with 9 wildcards, `1---------`, actually represents a recipe that generates 2^9 =512 rows since each wildcard used takes on 2 values, both '0' and '1'. All rows generated have the same output p10=1010 (=index 10).

Another example: the table on the left is a shortened entry of the table on the right:

c	b	a	y
-	0	-	1
-	1	0	0
0	1	1	1
1	1	1	0

→

c	b	a	y
0	0	0	1
0	0	1	1
1	0	0	1
1	0	1	1
0	1	0	0
1	1	0	0
0	1	1	1
1	1	1	0

For certain functions, we prefer wildcards, such as the previous priority function `p10` for ten inputs. However, when writing them manually, their excessive use reduces the clarity; see the left table above, from which we cannot tell at first glance whether we have listed all possible combinations of inputs. The use of wildcards is not a necessity, except as an aid to make writing easier for ourselves. In particular, they are widely used to specify truth tables during computer minimization of logic functions.

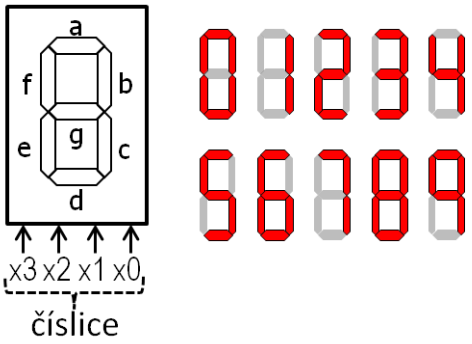
3.1 Value X - don´t care

For example, we write a truth table for a decoder converting decade digits to a 7-segment display but only for input values 0 to 9 (binary encoded as an unsigned integer, i.e., "0000" to "1001").

But what output values should we assign to inputs 10 to 15 (unsigned 1010 to 1111), which are not specified in the specification? We can fill something in them at design time, but we don't know if our randomly chosen values do not complicate later operations, such as minimizing logic functions. It is wiser to postpone the decision about their values for now.

As a sign of deferred decision, we use a flag called **don't care,** often written as `X`, which specifies that we do not care about the output value.

Using `X` and the wildcard '-', we fill the 7-segment decoder table. We suppose that the individual LEDs light up at logic '1'.



Digits	bits numbers				LED						
	x3	x2	x1	x0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X

Figure 25 - 7-segment display

7segment display table is almost professional, except for separating inputs and outputs into individual columns. In more concise notation, logical values are often combined into sequences or vectors, which makes the table much smaller.

For example, instead of:

x3	x2	x1	x0
0	0	0	0

we write:

x3 x2 x1 x0
0000

We truncated the table from Figure 25 to the more concise entry on the right:

Digits	bits numbers				LED						
	x3	x2	x1	x0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X

Digits	Binary x:3210	LED abcdefg
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1110011
10-11	101-	XXXXXXX
12-15	11--	XXXXXXX

The logical '0' and '1' sequences also have practical applications for shortening logical function notations in professional development tools, where logical values are often treated as vectors to shorten code. On the other hand, the designers hardly utilize the tedious process of defining a logical function by filling in tables divided into individual columns.

More about "don't-care"

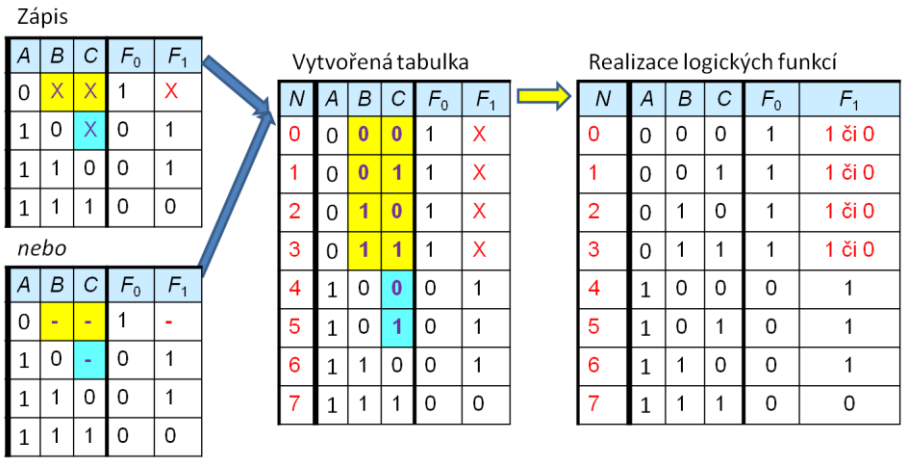
- "don't-care" refers to the designer's comment that the value of the output will be decided during the following steps, where it obtains a more advantageous value. It is thus a sign of a deferred decision (i.e., something like a to-do sign).

- "don't care" does not mean an unknown output, although it is sometimes misinterpreted as such.
- The "don't-care" cannot be physically implemented in circuits, so all "don't-care" symbols are eventually replaced during design by some specific implementable logical values, e.g., logical '0' or logical '1'.[4]
- "don't-care" is not used with inputs. The physical implementation of logic functions always requires their knowledge. In inputs, we can write wildcards, see p. 28, to merge definitions. Their values are given. The value of "don't-care" is chosen later.

Publications have not established a **uniform syntax** for wildcards merging inputs and "don't care" outputs. Sometimes, they are marked by the same symbols, usually X's, which are much more distinctive than a simple dash. Regardless of the characters used, however, they are easily recognized by their location in the truth table.

The meaning depends on whether the symbol is in the inputs or outputs section:

- The input of the logical function: for example, the code "0 - -" or "0 X X" (according to the notation used by the author) generates 4 lines of inputs 000, 001, 010, and 011 with the same output value, because the character, whether 'X' or '-', has here the status of a wildcard, i.e., a rule for generating input values.
- Output of a logic function: for example, the code "1X" or "1-" in an output will mean a deferred decision on its value. Here, the symbol always means "don't care". We can't use any wildcard generation for outputs ─ each must always have only one fixed value in the final table used to implement the logic function, and we can only temporarily defer the decision on what it will be.



[4] In an attempt to achieve maximum accurency, we avoid the claim that X (don´t care) must always and everywhere be defined either to logical ´0´ or to ´1´. This is usually the case, but there are other possibilities such as the aforementioned high impedance 'Z' state, more on p. 49, which is needed on bidirectional parallel computer buses. Furthermore, the output can also be realized by an open collector, for example on an I2C serial bus deployed in some audio-technology, which already belongs to other technical subjects.

3.2 Writing a truth table using an enumeration of values

Table 2 describes 4 logic functions whose outputs F0 to F3 only take the value '1' for one logic combination of inputs, thus reporting its presence. They are known as **one-hot decoders,** in our case 1 of 4. This essential logic design element is a cornerstone of many other functions. There will be more in the chapter 5.1 on p. 79.

N	B	A	F0	F1	F2	F3
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1

Table 2 - "One-hot" decoder - 1 of 4

More elegantly, we specify its functions by the set of input values in which they take the logical '1', which we call **on-set**. We encode the combinations of input bits as an unsigned binary number. Table 2 is then reduced to a single row, the list of onsets.

$$F0^{on} = \{ 0 \}, F1^{on} = \{ 1 \}, F2^{on} = \{ 2 \}, F3^{on} = \{ 3 \},$$

In other states, outputs F0 to F3 take '0'. Writing indices is not very convenient. The concept of minterm as an AND member was introduced in Figure 5 on p. 13. What we know about it is that it outputs a '1' for only one input combination. So we list used minterms. We denote them by lowercase m and put the binary values of the inputs in parentheses as an unsigned number.

$$F0 = m(0), F1 = m(1), F2 = m(2), F3 = m(3)$$

Table 3 on the next page describes a cognitive decoder called **one-cold** because outputs F0 to F4 will be just '0' for one input combination.

N	B	A	F0	F1	F2	F3
0	0	0	**0**	1	1	1
1	0	1	1	**0**	1	1
2	1	0	1	1	**0**	1
3	1	1	1	1	1	**0**

Table 3 - "One-cold" decoder - 1 of 4

Here, the description using on-sets is not convenient, so we use off-sets[5] , which means the set of inputs for which the output is in '0'. The one-cold decoder functions are again easy to write:

$$F0^{off} = \{ 0 \}, F1^{off} = \{ 1 \}, F2^{off} = \{ 2 \}, F3^{off} = \{ 3 \}$$

Again, the unlisted values are in logical '1'. An easier notation writes the list of all maxterms with capital M:

$$F0 = M(0), F1 = M(1), F2 = M(2), F3 = M(3)$$

---

[5] The name off-set is quite misleading, because it is usually used in engineering and mathematics to refer to a deviation or offset, but it is indeed used in the literature on logic circuits. The mathematical notation for the *on-set*, *off-set* and *don't care set* notations also varies by author. The descriptions Fon , Foff and Fdc given here are not stable.

On the other hand, lowercase **m** (from minterm) and offset as uppercase **M** (from Maxterm), and dc (don't care) are commonly used. After all, these are also much shorter notations☺

We utilize the same principle also for logical functions with output values don't care. For them, we add a don't care set.

N	C	B	A	X	Y
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	X	0
3	0	1	1	X	0
4	1	0	0	1	0
5	1	0	1	1	X
6	1	1	0	1	1
7	1	1	1	1	1

The logic functions X(C,B,A) and Y(C,B,A) defined by the table above can be written as follows:

$$X: X^{off} = \{ 0,1 \}, X^{dc} = \{ 2, 3 \} ; Y: Y^{on} = \{ 6,7 \}, Y^{dc} = \{ 5 \} ,$$

respectively X: M(0,1) dc(2,3); Y: m(6,7), dc(5)

We chose the method that gave us the most minor work in a given function. Here, we described X output with the aid of the off-set and don't care set; the '0' outputs were less frequent than '1'. We created output Y using the on-set and don't care set for a similar reason.

Example: Let's write basic logic functions using minterms and maxterms:

Weight of input	+2	+1						
Unsigned index	x	y	xor	xnor	and	nand	or	nor
0	0	0	0	1	0	1	0	1
1	0	1	1	0	0	1	1	0
2	1	0	1	0	0	1	1	0
3	1	1	0	1	1	0	1	0

XOR: m(1,2); XNOR: m(0,3) but also XOR: M(0,3), XNOR:M(1,2)

AND: m(3); OR:M(0); NAND: M(3), NOR:m(0)

3.3 Karnaugh maps

In engineering practice, logic functions with fewer inputs are frequently drawn by the Karnaugh maps, abbreviation KM, as faster and more straightforward notation. We derive 4-input KM from the truth table of a logic function Y=f(D,C,B,A) with 4 inputs D,C,B and A, where D has the highest weight. Its output Y has 16 values, of logical '0', '1', or X (don't care), which we will denote only by the logical constants y00 to y15, whose indices indicate the output order.

D	C	B	A	Y
0	0	0	0	y00
0	0	0	1	y01
0	0	1	0	y02
0	0	1	1	y03
0	1	0	0	y04
0	1	0	1	y05
0	1	1	0	y06
0	1	1	1	y07
1	0	0	0	y08
1	0	0	1	y09
1	0	1	0	y10
1	0	1	1	y11
1	1	0	0	y12
1	1	0	1	y13
1	1	1	0	y14
1	1	1	1	y15

		0	0	1	1	B
D	C	0	1	0	1	A
0	0	y00	y01	y02	y03	
0	1	y04	y05	y06	y07	
1	0	y08	y09	y10	y11	
1	1	y12	y13	y14	y15	

Figure 26 - Truth table drawn in matrix form

The truth table on the left has 16 rows, and their 4 consecutive rows have always identical inputs D and C. We rewrite the table in shorter 4x4 matrix notation, see right, where the input values of row elements are determined by their position in the column and row.

We adjust the matrix in Figure 26 by swapping its last two columns and last two rows to obtain the middle table in Figure 27, our Karnaugh map of a logic function. The logical '1's of the input values lie next to each other, so instead of writing 0 and 1, we just draw a line to symbolize where the input has the value '1'; see the table on the right.

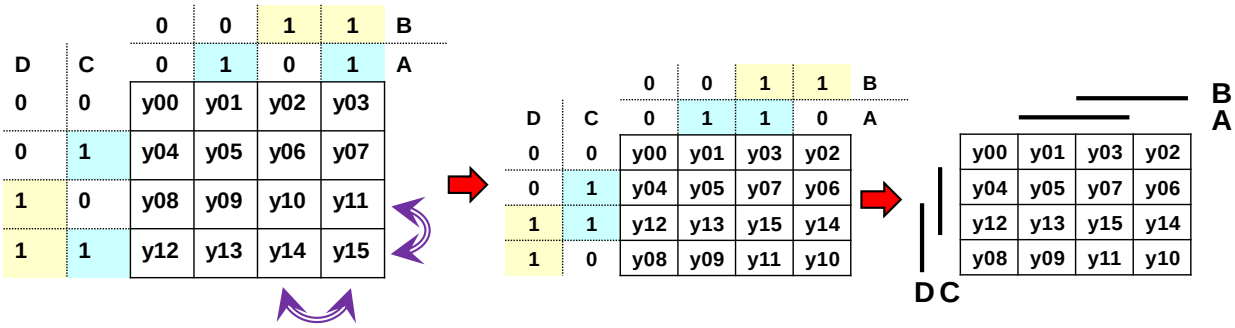


Figure 27 - The genesis of the Karnaugh 4x4 map

The most important feature of a Karnaugh map, and its necessary condition, is that **only one input variable changes** for any vertical or horizontal move.

For example, output y00 has inputs DCBA=0000 and output y04 one line below 0100. So, when going from y00 to y04, only input C changed from '0' to '1'. The property is also valid over the map's borders, for example, when moving from the first to the fourth row in the same column.

We take the last column as an example. Output y02 has inputs DCBA = 0010, and output y10 has inputs DCBA = 1010. Only D has changed from '0' to '1'. Output y14 at the end of the third row has inputs DCBA=1110, and output y12 at the beginning of the same row gives DCBA=1100. Only B has changed from '1' to '0'.

A Gray code is the ordering of the output values of a logic function so that only one of its

input variables changes as it moves horizontally in a row or vertically in a column. It is used not only in Karnough maps but also in position sensors and information transmission, for example, to correct errors in digital television.

The indices of yi outputs in the Karnaugh map are not in sequential order. If we compare their numbers in the same row, we see that the index in the second column on the same row is always greater by +1 than in the first column, in the third column by +3 , and in the fourth column by +2 .

The property follows from the input variables. Each bit has a weight given by a power of 2^n . If we arrange the inputs from the most significant D to the right, i.e., DCBA, then input A has weight $1=2^0$, input B has weight $2=2^1$, input C has weight $4=2^2$, and input D has weight $8=2^3$. The sum of the variable weights (row+column) determines the value of the index in the corresponding field of the Karnaugh map.

The second column has indexes +1 higher than the first column because it applies the variable A with weight +1. Compared to the first, the third column will have two variables A+B, so it will have indexes +3 higher than the first. Similarly, the last column has only variable B in addition to the first column, and it weights +2.

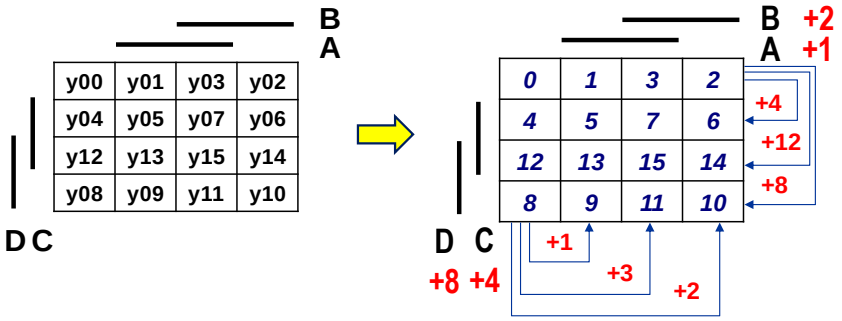


Figure 28 - Dependencies in the Karnaugh 4x4 map

We derive the differences between the indices in the columns from this property. The second row always has index values +4 greater than the corresponding element of the same column of the first row (+C=4). The third row has indexes greater by +12 (+D+C) than the first and fourth rows by +8 (+D). So, we need to correctly assign the indexes to the first row, and we can mechanically derive the rest.

Karnaugh's map, abbreviated KM, shown in Figure 27, is not the only way to draw it or to arrange the input variables. Authors often use different styles according to their habits. Some of the many possible options are given in Figure 29.

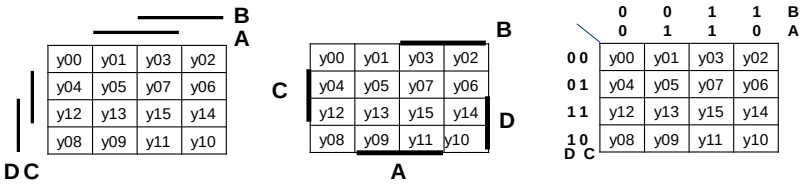


Figure 29 - Some possible variable labels for the Karnaugh 4x4 map

Also, we can rearrange the variables so that they will obtain different weights, which gives us a different ordering of the indices. There are many possible Gray codes[6] ; for example, 5712 for 4 bits.

[6] For a list of other Gray codes, see Wikipedia: https://en.wikipedia.org/wiki/Gray_code

The "binary-reflected Gray code" used in Figure 29 is the most commonly used in logic and programs due to its simple conversion algorithm explained in the Binary prerequisite.

We can draw a Karnaugh map in any Gray code, i.e., the variable ordering that satisfies the condition that only **one input variable is changed when we perform any horizontal or vertical move inside one column or row, including transitions across map borders**.

Example: draw a Karnaugh map (KM) for the e-LED of a 7-segment display.

Solution: Figure 25 on page 30 describes the truth table of a 7-segment display. We select the logic function of e-LED.

We don't have much experience drawing KM yet ☺, so we prefer to proceed through an intermediate step to avoid making an unnecessary error.

First, we draw an auxiliary KM, filling in the index numbers of the individual fields according to **our chosen ordering** of the input variables. According to it, we then write the values into the final truth table of e-LED.

N	x:3210	e
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10-11	101-	X
12-15	11--	X

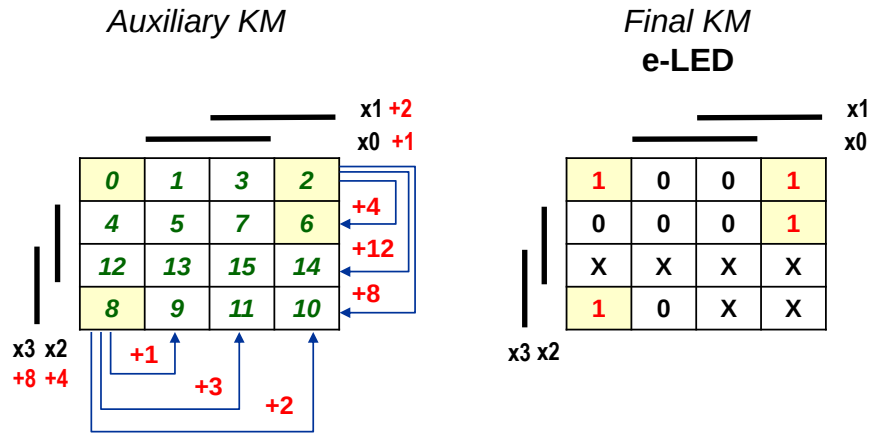


Figure 30 - Karnaugh map of the e-LED 7segment display

3.3.1 Karnaugh maps of various sizes

Karnaugh maps allow fast manual minimizing or writing logic functions, but only for a few inputs. Their complexity increases exponentially with the number of variables. Figure 31 presents some variations (selected from many possible ones) of constructing a Karnaugh map for numbers of inputs other than 4x4, including the resulting field numbering. In all of them, we used the "binary-reflected " Gray code.

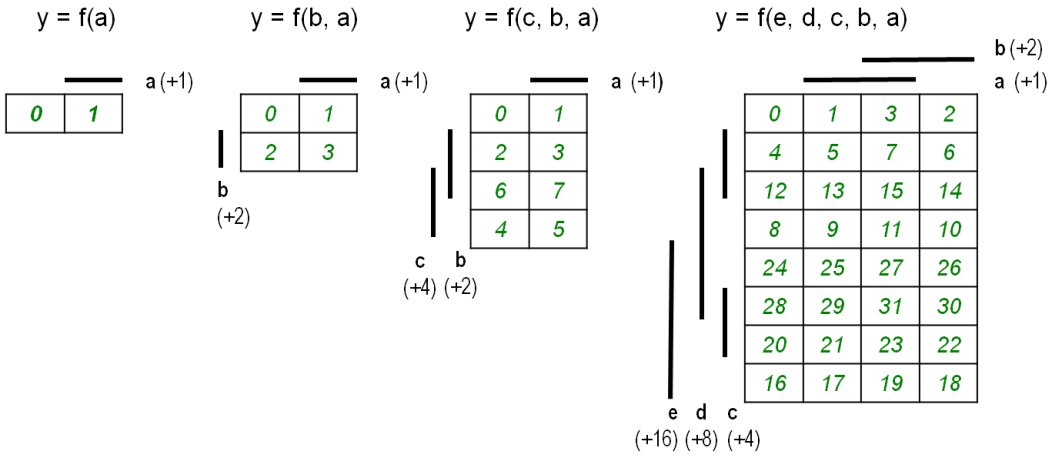


Figure 31 - Karnaugh charts for sizes other than 4x4

Fortunately, it is possible to reduce the number of variables in a logic function by various decompositions; for example, we will soon see the Shannon expansion, see p. 49 and 52. Thus, in manual designs, we can always make do with maps up to 4x4☺.

3.3.2 The principle of minimizing Karnaugh maps by the SoP method

So far, we have used the terms minterm and maxterm, which concatenate all variables from some set by AND (minterm) or OR (maxterm) operators. So, their result is '1' (minterm) or '0' (maxterm) for only one combination of input values.

Consider arbitrary logic function f() with N different input variables. From them, we can create expressions with Q distinct variables, Q=1 to N, chained by AND operators, for example, the expression x1 and not x2 and x4. If it does not contain all the inputs, only some of them, it defines multiple outputs of a logic function.

We introduce the more general notion of an AND implicant. By analogy, we define an OR implicant for concatenating with only OR operators, e.g., x1 or not x3.

- The AND implicant with Q members will determine 2^{N-Q} logical outputs of f() in '1'; the others will be in '0'.
- An OR implicant with Q members defines 2^{N-Q} logical outputs of f() in '0'; the others will be in '1.

One AND or OR implicant specifies the output values of f() in the count 1, 2, 4, 8, 16, ...

If **Q=N** AND implicant specifies a **single** '1' in f(), it will also be a **minterm.**
The OR implicant, on the other hand, specifies a **single** '0', it will also be a **maxterm**.

Note: The term implicant is bound to a given set of N inputs. It is a precise term, but in some publications, AND and OR implicants are always called minterms and maxterms to simplify explanations. Groups or other notations are also introduced for implicants.

We take, for example, AND implicants for N=4. They specify 1, 2, and 4 logical outputs.

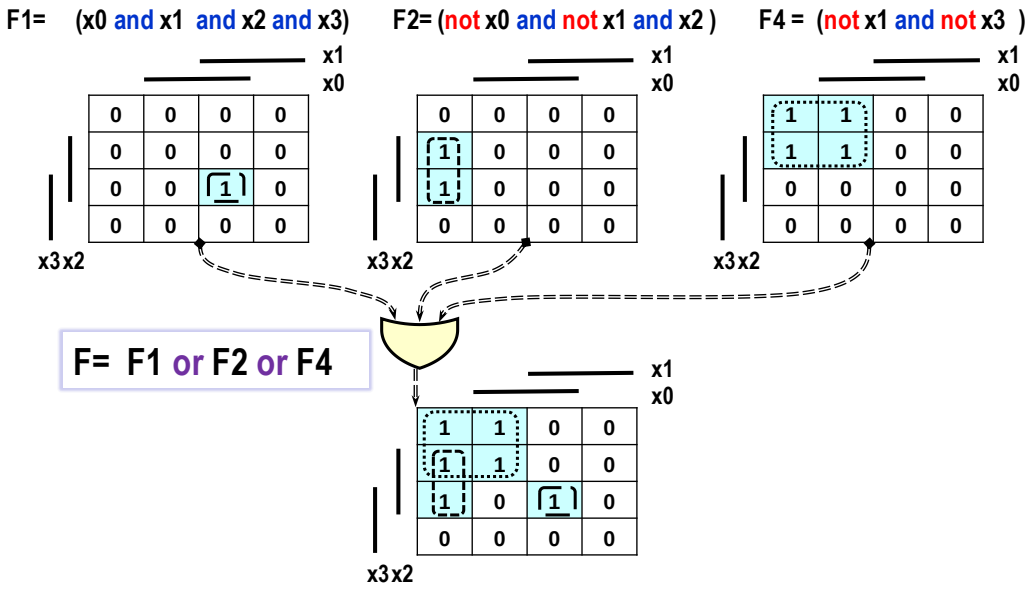


Figure 32 - Principle of the PoS method

If we create a logical function F by concatenating the AND implicants of F1, F2, and F4 with OR operators, then its output will be given by the logical sum.

In minimization, we reconstruct the original terms of such expressions. In the Karnaugh map,

we look for AND implicants, and as the largest as possible, that cannot be extended further. They are then called **primary implicants** and determine the number of logical outputs given by a power of 2. We say that the implicant covers these outputs.

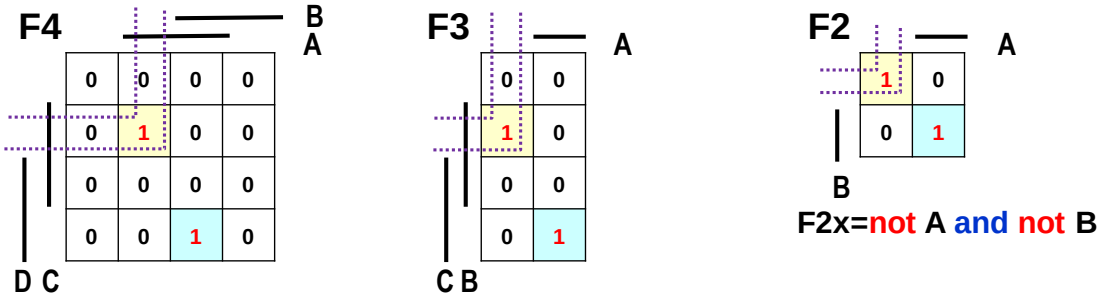
When we find AND implicants that cover the whole Karnaugh map together, we concatenate them by OR operator; therefore, the method was named **SoP - Sum of Products.**

We will illustrate the procedure by individual cases of coverages of 1, 2, 4, and 8 elements.

3.3.3 Demonstration of SoP situations

Coverage of 1 element

Suppose that he has three functions F4(A,B,C,D), F3(A,B,C) and F2(A,B) given as Karnaugh maps. We start building the expression from the upper ones highlighted in yellow.



The implicants, also called minterms here, are written directly from KM's '1' positions. Each covering just one '1' will contain all input variables.

We can see that the yellow '1' **is under** A, **not under** B, **next to** C, and **not next to** D. The word "not" is implemented here by adding NOT operator before the variable:

F4x = A and not B and C and not D

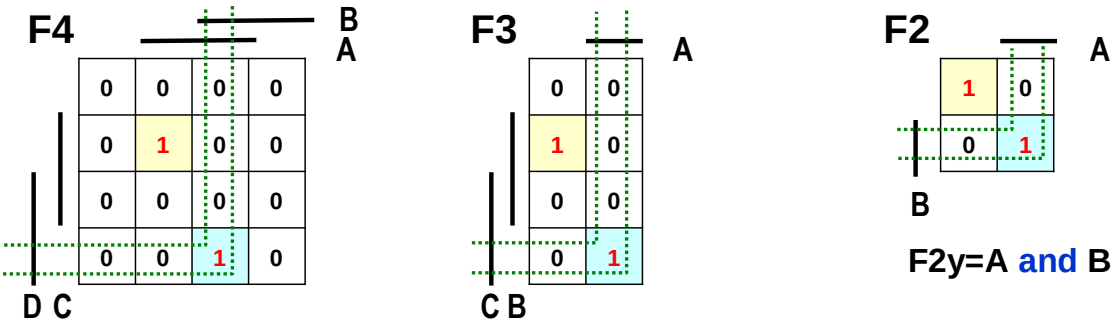
The F3x is described analogously: **it is not under** A, it **is next to** B, and it **is not next to** C.

F3x = not A and B and not C

We express F2x similarly: **it is not below** A and **not next to** B.

F2x = not A and not B

We apply a similar procedure to the next green highlighted 1 and get F4y, F3y and F2y



The resulting functions are obtained by concatenating their members by OR operations. As soon as any minterm is in '1', the OR (selection of the maximum) will also be in logical '1':

F4=F4x or F4y=(A and not B and C and not D) or (A and B and not C and D)

F3=F3x or F3y=(not A and B and not C) or (A and not B and C)

F2=F2x or F2y=(not A and not B) or (A and B)

Coverage of 2 elements

If we have two '1's next to each other, then we cover them with an AND implicant that will have one less variable than the number of inputs of the function.

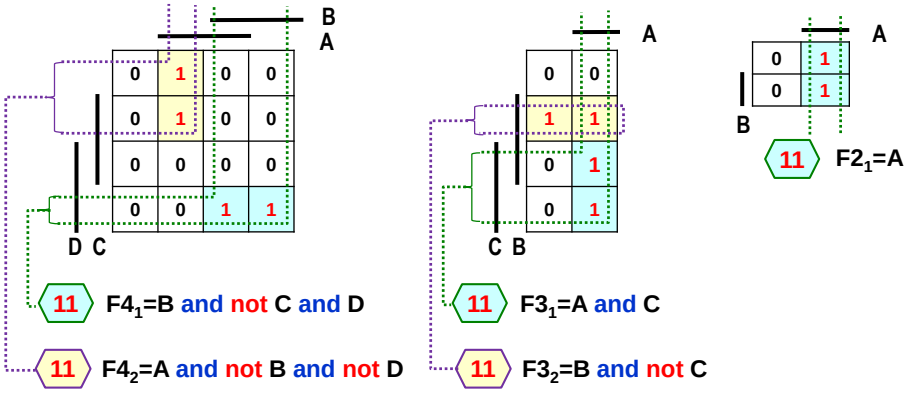


Figure 33 - Implicants of two '1's

If we were to write each green highlighted lower '1' using two minterms (AND implicants covering a single element), then we would get:

(lower left '1') $F4_{1L}$ =A and B and not C and D

(bottom right '1') $F4_{1R}$ =not A and B and not C and D

After combining the two minterms of the OR operation, we get

$F4_1$ = $F4_{1L}$ or $F4_{1R}$ = (A and B and not C and D) or (not A and B and not C and D)

We extract (B and not C and D) from the expression above

$F4_1$ = (A or not A) and (B and not C and D)

and apply the complementarity rule, see Figure 10 on p. 17.

$F4_1$ = **(B and not C and D)**

However, the above AND implicant with three terms can be constructed directly from KM by the position of the two green highlighted ones. In a Karnaugh map, any movement perpendicularly or horizontally will change the value of only one input variable. So, it is possible to apply the complementarity rule graphically.

We write that the two green highlighted '1's **are under** B, **not next to** C and **next to** D, as

$F4_1$ = B and not C and D

By analogy, we construct a ternary AND implicant for the top '1' highlighted in yellow, both of which are **under** A but **not under** B and **not next to** D:

$F4_2$ = A and not B and not D

We combine them with the OR operator to get the resulting logic function described by KM:

F4= $F4_1$ or $F4_2$ = (B and not C and D) or (A and not B and not D)

Middle KM, Figure 33 top, describes a logic function with three variables that have 4 logical '1's, but no implicant cannot cover them. We will show the arrangement of the 4 logical '1's for which this is possible below. Here, we must use a pair of implicants with two terms, one

less than the number of variables of the function.

The two green highlighted lower '1's lie **below A** and **are next to C,** hence **$F3_1$ =A and C**

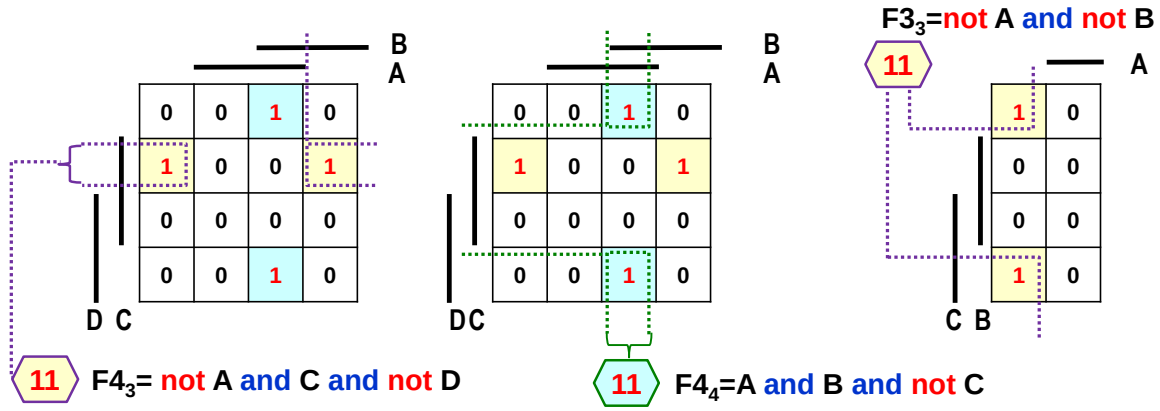
The top two '1's highlighted in yellow are **next to B** and **not next to C, $F3_2$ =B and not C**

The resulting logic function will be

$$F3 = F3_1 \text{ or } F3_2 = (A \text{ and } C) \text{ or } (B \text{ and not } C)$$

The two-input logic function forms a trivial case. Both ones are under A, so $F2_1$ =A.

If '1's lay at the beginning and end of the map, we can cover them with the same implicant since the Karnaugh map preserves the rule of changing the value of only one input variable even when moving horizontally or vertically across its border. In the same column, the top box is adjacent to the bottom box, which is analogous to the row.



The yellow highlighted ones $F4_3$ **are not under** A, they **are next to** C and **not next to** D.

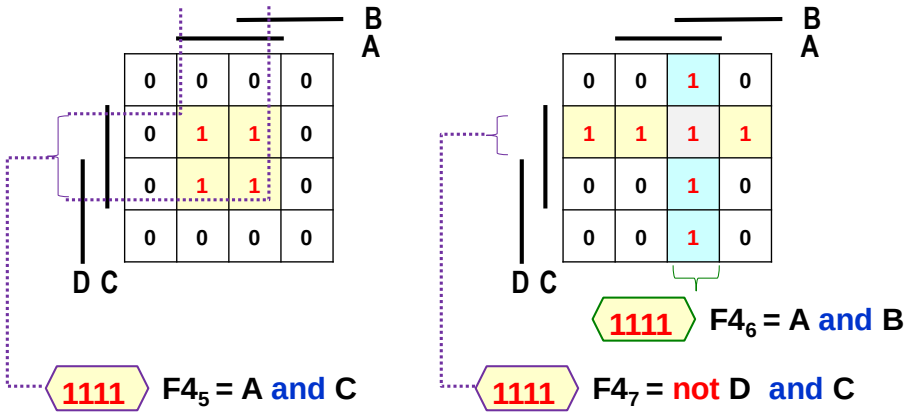
$$F4_3 = \text{not A and C and not D}$$

The green highlighted ones $F4_4$ are **under** A and B, but **not next to** C.

$$F4_4 = A \text{ and } B \text{ and not } C$$

Coverage of 4 elements

If implicants are shorter by 2 variables than the number of input variables, they will specify 4 1's in KM.



Again, we build the corresponding implicants from positions: $F4_5$ is **below** A and **next to** C, so we write $F5$ =A and C. A pair of implicants can cover the KM on the right. We construct them in the same way. Its resulting function is:

$$F4_{67} = F4_6 \text{ or } F4_7 = (A \text{ and } B) \text{ or } (\text{not } D \text{ and } C)$$

The greyed-highlighted box of inputs A='1', B='1', C='1', and D='0' lie in both AND implicants $F4_6$ and $F4_7$. Both cover it. The OR function, selecting the maximum, will be in '1' for one or more of its inputs in '1'. So, each '1' can be included in multiple AND implicants as we see fit.

In the coverage of 4 elements, there can be over-edge coverage, as shown in the figure above. We build again from the positions.

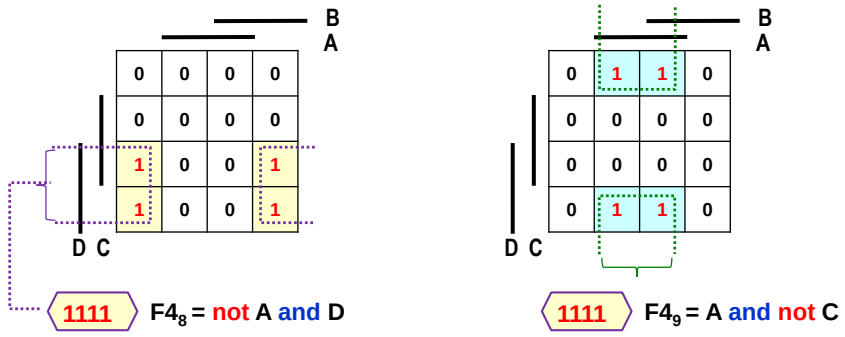


Figure 34 - Coverage of 4 elements over an edge

The most exciting case occurs with corner coverage, where the F_{10} function (shown in the figure below left) applies edge contiguity.

The four 1s in the corners **are not under** A and **not next to** C, i.e.

$$F_{10} = \text{not A and not C}$$

To clarify the rule, we scroll KM circularly by 1 row and 1 column.

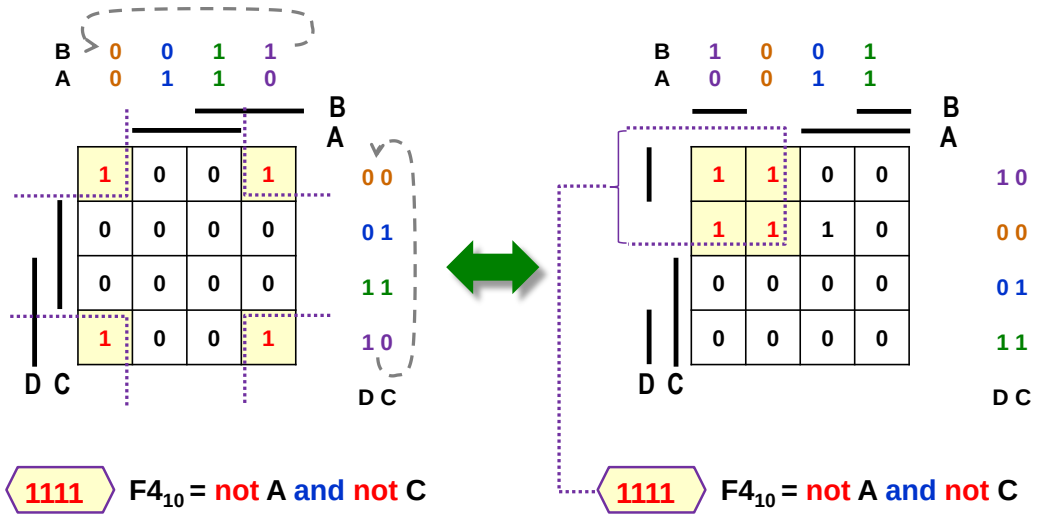


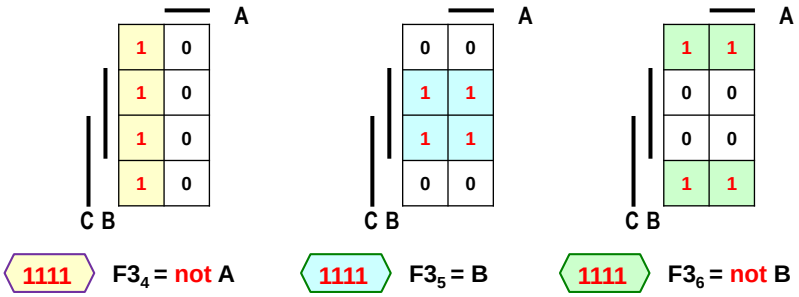
Figure 35 - Coverage of the corners of the Karnaugh map

Figure 35 shows the same logic function in the Karnaugh maps on the left and the right.

Only the right KM does not use the "reflected binary" Gray's code, but another one. It also satisfies the requirement of one variable change when moving in a row or column, including transitions across table borders. And the quadruple '1' here lies with each other also graphically [7] .

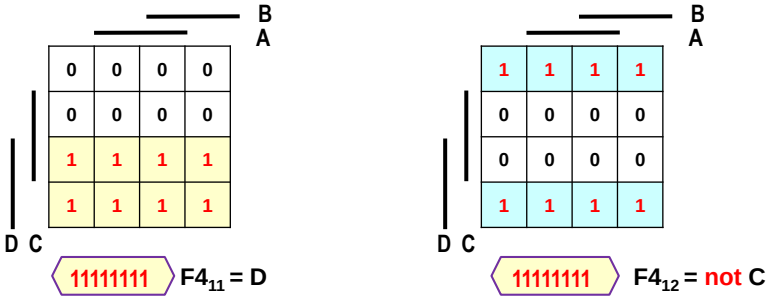
[7] In the lectures you will learn that the Karnaugh map of four variables represents the graph of a logic function on a four-dimensional cube. Each vertex has four neighbors. For the KM of three variables, it is then a three-dimensional cube with three adjacent vertices. This implies a connection across the right end of a row to its left beginning, or in a column from top to bottom. The shell of the cube has no ends and no beginnings, these are only created by unfolding it into a plane. The position of '0' and '1' in the KM then depends on which vertex we start expanding the cube from.

An AND implicant smaller by two terms for KM dependent on three inputs reduces to just one variable and primitive coverage cases.



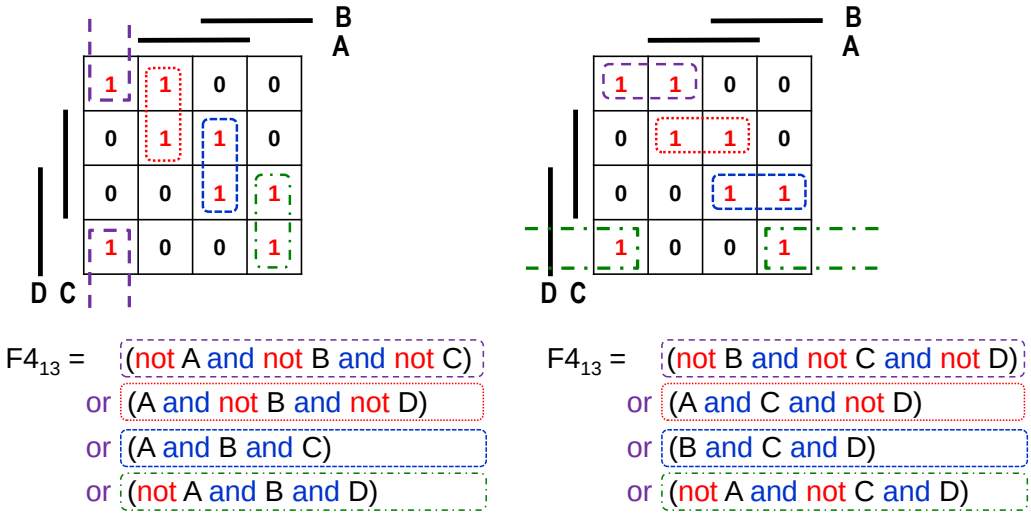
Coverage of 8 elements

For a logic function with four inputs, an AND implicant shorter by three terms is used, i.e. only one variable, as we derived in chapter 3.3.2 on p. 37 (here we have $8=2^{4-1}$).



Example: coverage of KM by multiple implicants:

More complicated Karnaugh maps are covered by more implicants, from which we choose as the largest as possible, i.e., **primary implicants**. There may be multiple possible coverages.



We can cover the KM in the figure above differently with the same complexity.

F4$_{13}$ =(not A and not B and not C) or (A and not B and not D) or (A and B and C) or (not A and B and D)

=(not B and not C and not D) or (A and C and not D) or (B and C and D) or (not A and not C and D)

Here, we see the disadvantage of logical expressions. Even if they have different forms, they describe the same logical function.

If we construct their KMs with keeping the same variable order, we obtain the same results,

because KMs are only the style of drawing truth tables. And logical functions with the equal truth tables are identical[8].

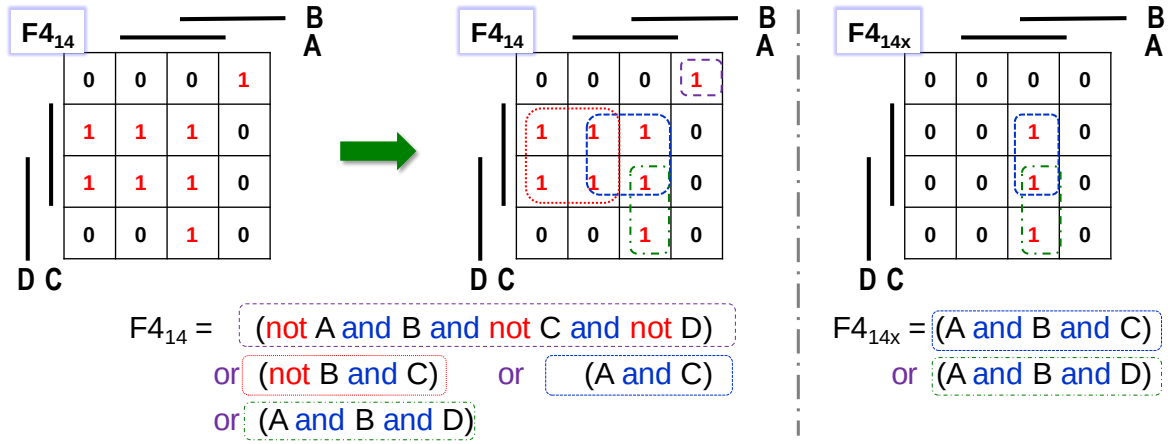


Figure 36 - Multiple Implicant Coverage

The logical '1' in the upper left corner in $F4_{14}$ is isolated and will be used as an AND implicant containing all input variables, which makes it a minterm. We will cover the middle block with six '1's as two AND implicants, each including four '1's. We will write the bottom pair as an AND implicant.

The right $F4_{14x}$ demonstrates a situation where three '1's lie side by side. Since the implicants exclusively cover the number of '1's given by the power of 2, we must use two overlapping AND implicants, each including two '1's.

[8] So we could say that for logical functions it is enough to construct their KM as a proof of their consistency. Unfortunately, the complexity of KM increases with 2^N , where N is the number of input variables. They can easily be applied to functions of up to 4 or 5 variables, where minimizing the KM is faster than inserting the data into a program. More complex functions are often handled with the aid of decompositions, which will be discussed later in the chapter on combinational circuits.

Example: Using don't care

If a Boolean function contains don't care characters, we can choose which ones to include in coverage and which not to. All of them we cover with the SoP method will be set to logical '1'; the others will be '0'. At this moment, we decide the value of don't care.

Figure 30 on p. 36 contains a Karnaugh map of the e-LED 7-segment display. We write it as a logic function using two implicants, each involving four logical '1's. We first use the corner coverage shown earlier; see Figure 35 on p. 41. In the second, we take up the entire right column.

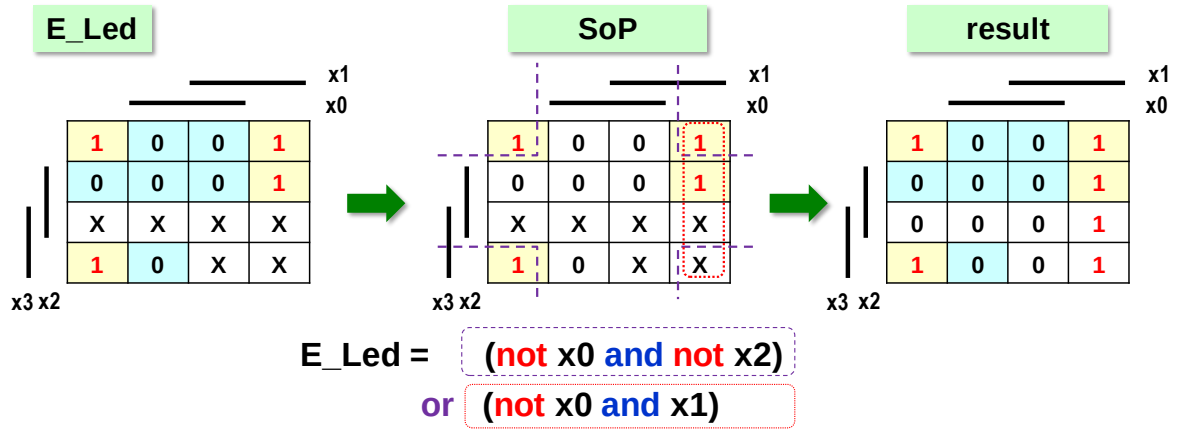
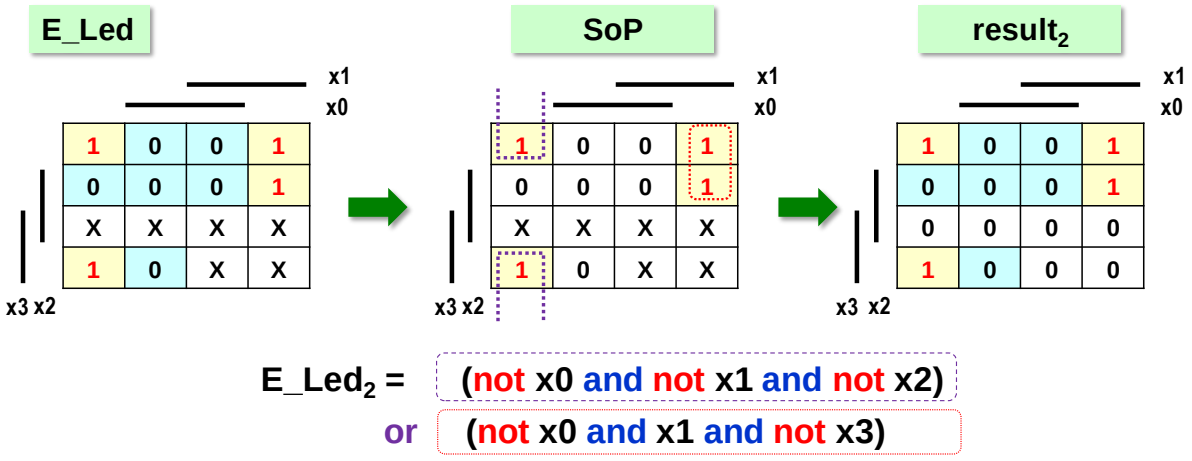


Figure 37 - Example of using don't care

Our coverage also **defined** all don't care (deferred value decisions). It has assigned them '0' and '1'; see the table on the right. Covered '1', uncovered '0'.

What happens if we omit don't care in our implicants? All we get are slightly more complex expressions with more members.



Note here that the E_Led_2 logic function gives a different output than the previous E_Led, but both match in all required values, i.e., where KM prescribes '0' and '1' .

One may ask whether the **E_Led_2 proposal will be a bug.** After all, we didn't use possible corner coverage in it and don't care!

The answer depends on how the logical function is physically implemented. In the early days of logic, when everything was wired with solder and wires, the E_Led_2 design would be called a gross error because it requires multiple gates.

Today, the equations of logical expressions are inserted into the design environment, which

performs their optimization. If the target physical implementation is an FPGA, then it will use configurable logic elements, LEs, which use a LUT, Lookup Table. We will discuss these in the FPGA internals section of Chapter 5.4, starting on p. 86. Each LE can usually perform a logic function with four or more inputs. Thus, just one LE is consumed, both on E_Led and E_Led_2 .

The E_Led_2 expression is not a mistake today. After all, it has provided the desired output, our primary requirement. We can only name it a suboptimal design, as it unnecessarily writes more extended expressions.

Question: And can the circuit design environment be given an E_Led logic function without covering the Karnaugh map?

Answer: Yes. In HDL circuit design languages, a logic function can be defined by a logical expression, an enumeration of output values, and other means. However, the logical expression is often shorter and more straightforward. Because of this, we are explaining minimization.

3.3.4 Minimization of Karnaugh maps by PoS method

Covering the Karnaugh map by the SoP method may not yield optimal results if it contains many ones. If it includes far fewer logical '0's, we cover if by OR implicants more quickly. We combine these with an AND operation (selecting the minimum), against which '0' (the minimum) is the aggressive element. If any OR implicant is in '0', the output will also be '0'.

The method is called **PoS - Product-of-Sum** and is demonstrated in the picture below. The procedure is the same. We look for the primary OR implicants, i.e., the largest possible ones. We apply everything we know.

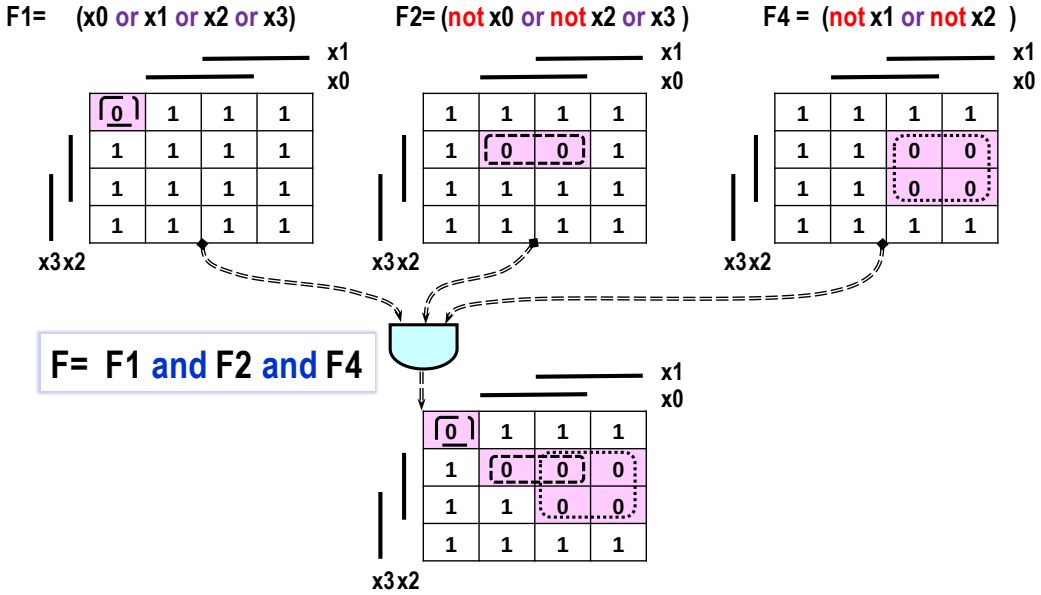


Figure 38 - Principle of the PoS method

OR implicants, however, derive NOT operators in the opposite way to AND implicants.

For example, the pair '0' covered by the OR implicant F2= (not x0 or not x2 or x3) is below x0 and next to x2, which is written with NOT operators in the OR implicant. In contrast, next to x3 has no NOT before x3.

The difference follows from De Morgan's theorem (p. 19). Suppose we cover the negated F2 by the SoP method. In that case, the resulting expression can be converted by negation to the original F2, yielding an expression identical to the coverage of OR implicants.

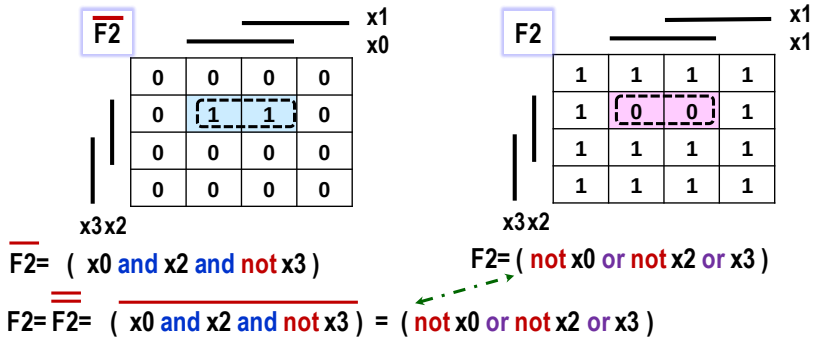


Figure 39 - Comparison of AND and OR implicant coverage

We will show an analogy to the coverage that was demonstrated in Figure 33 on p. 39.

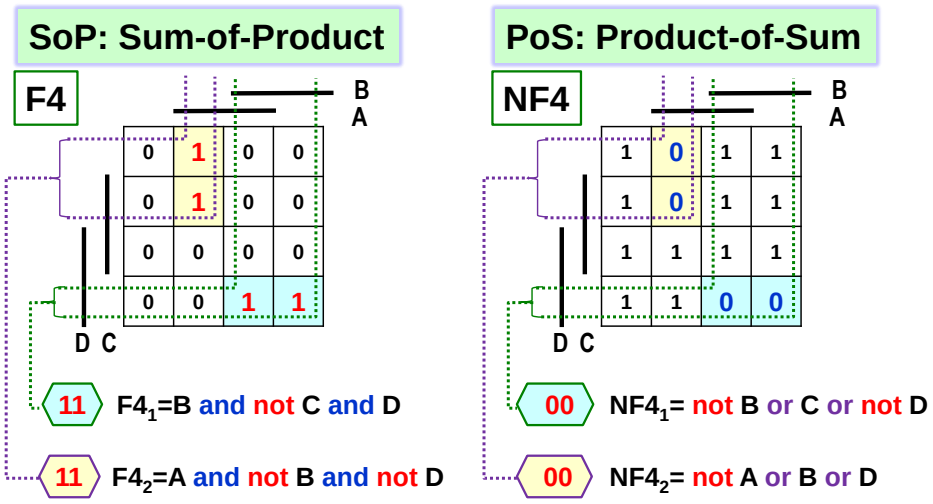


Figure 40 - SoP, coverage '1', versus PoS, coverage '0'

The OR implicant of the NF_1 function is formed according to its position **is under** B, **is not next to** C and **is next to** D. The unary NOT is written before variables in opposite situations to AND implicants:

$$NF4_1 = \text{not } B \text{ or } C \text{ or not } D$$

Similarly, we express the second OR implicant as **being under** A, **not under** B, and **not next to** D

$$NF4_2 = \text{not } A \text{ or } B \text{ or } D$$

The resulting function is then constructed with both implicants by concatenating them AND:

$$NF4 = NF4_1 \text{ and } NF4_2 = (\text{not } B \text{ or } C \text{ or not } D) \text{ and } (\text{not } A \text{ or } B \text{ or } D) \quad (n1)$$

We will again show the connection with De Morgan's theorem (p. 19). We write down NF4 as the negation of F4 covered by implicants.

$$NF4 = \text{not } F4 = \text{not } (F4_1 \text{ or } F4_2) \quad (n2)$$

$$= \text{not } (\text{ (B and not C and D) or (A and not B and not D) }) \quad (n3)$$

Now, we will expand the not operators before the parenthesis according to De Morgan's theorem, changing the or operator to and. We also move the not operators before both members of the expression. In the next step, we repeat for them as well.

$$NF4 = \text{not } (B \text{ and not } C \text{ and } D) \text{ and not } (A \text{ and not } B \text{ and not } D) \quad (n4)$$

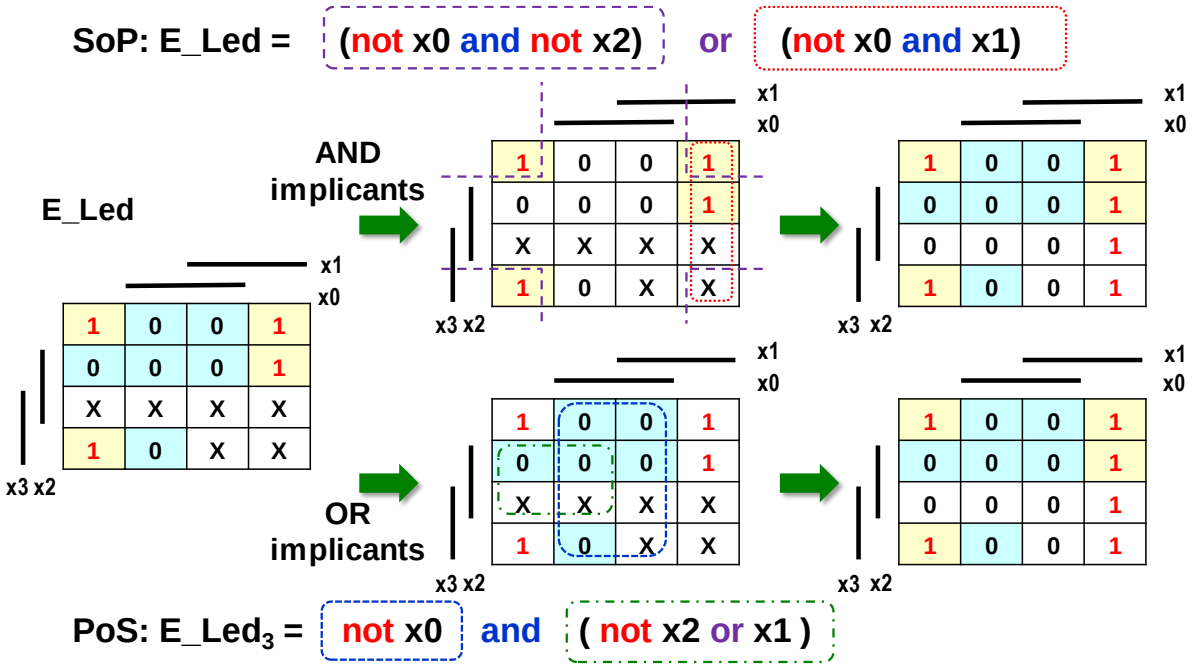
$$= (\text{not } B \text{ or } C \text{ or not } D) \text{ and } (\text{not } A \text{ or } B \text{ or } D) \quad (n5)$$

The expression (n5) is identical to (n1). Thus, DeMorgan's theorem has its analogue in the coverage of the negated KM by the PoS method and the subsequent negation of the result.

3.3.5 Comparing coverage with the use of don't care

Coverage of AND implicants, i.e., using '1', must include all '1's and may include some don't care; these will then be redefined to logical '1'. The rest will be in '0'. When working with OR implicants, we must cover all '0' and can add suitable don't care ones. These will thus be redefined to '0' while the others will be '1'.

Let us compare these methods of E_Led 7segment display coverage:



The Boolean function E_Led$_3$ needs two OR implicants. One lies below x0, so will be not x0, while the other is next to x2 and not below x1, giving not x2 or x1. The resulting function:

E_Led$_3$ =not x0 and (not x2 or x1)

The uncovered don't care is redefined to '1', giving us the same resulting function as coverage '1', which we demonstrate by multiplying the not x0 term by the parenthesis.

The following example shows different coverages of '0', i.e. OR implicants. Both representations are correct, but by differently defining don't care, they lead to other logical functions, but identical in the prescribed '0' and '1':

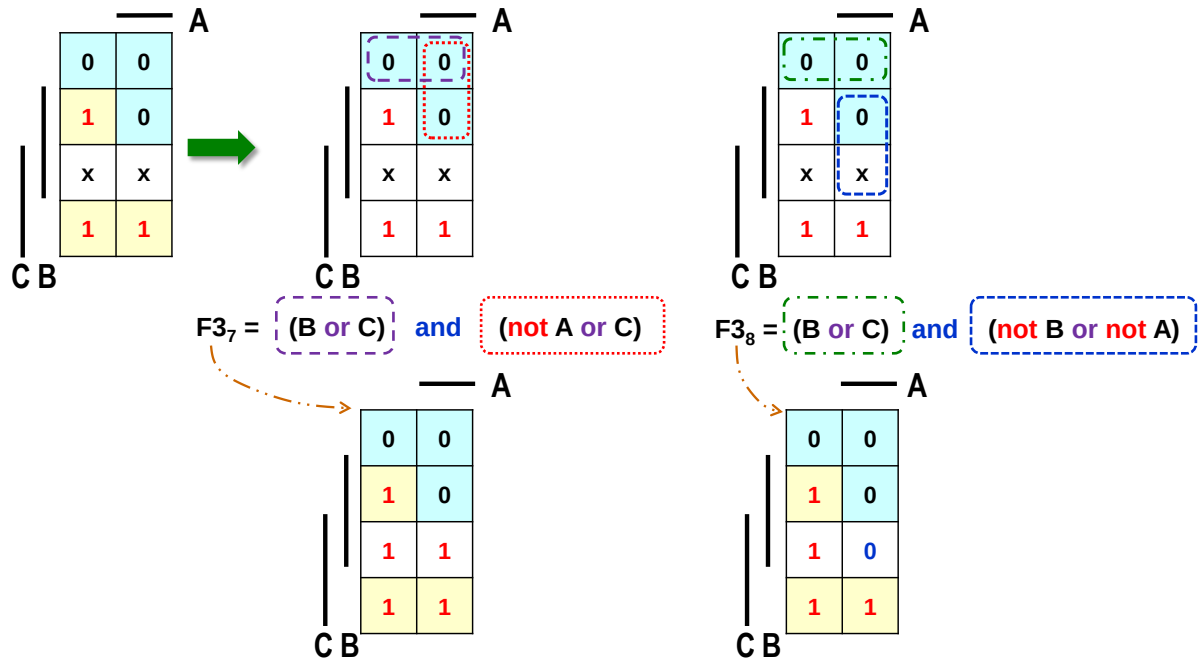


Figure 41 - Definition of don't care at coverage '0' (PoS)

3.3.6 Shannon's expansion of Karnaugh's map

Karnaugh's map for 4 variables is the largest for which the logical '1' of all AND implicants (or '0' of OR implicants) appear together considering also transitions across the ends of the table.

For larger KM, they will be scattered. The figure below shows a map of 8 variables; the map of 4 variables is its slice.

An implicant with Q=4 terms determines N=8 variables in a logic function

$$2 = 2 = 2 = 16^{N-Q\ 8-4\ 4}$$

logical '1'. The map on the right shows the logical '1' from the AND implicant of A and B and C and D. These no longer lie side by side, complicating coverage.

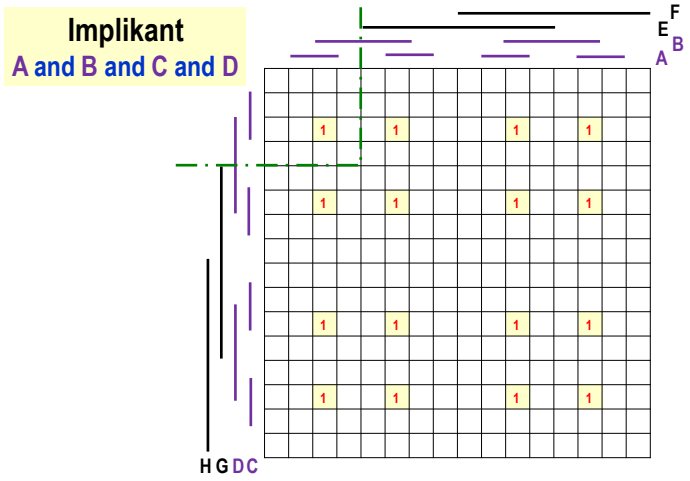
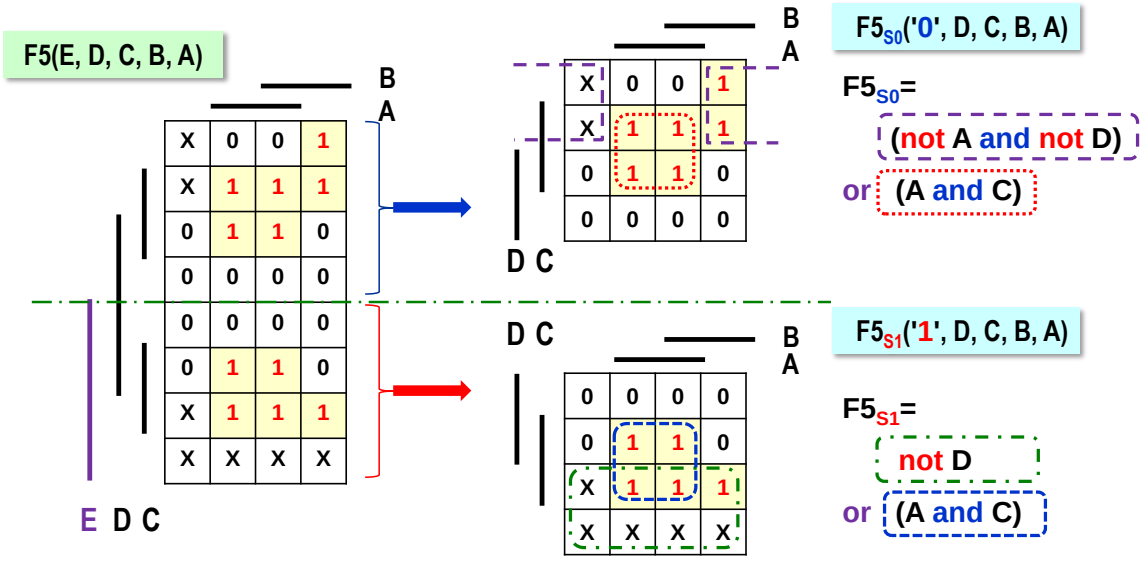


Figure 42 - Karnaugh map of 8 input variables

But what if we have a larger KM and we lack a suitable program? Then, we can break it into half KMs based on one variable. In the figure below, we have chosen E. The top KM will have E='0', while the bottom KM will have E='1'. We can quickly minimize the two sub KMs of dimension 4x4.



We concatenate the result so that the output values of $F5_{S0}$ only apply when E='0', while $F5_{S1}$ only applies when E='1', which is achieved by the or operation and by adding the notes E and E before the subfunctions.

$$F5_S = (\text{not } E \text{ and } F5_{S0}) \text{ or } (E \text{ and } F5)_{S1} \qquad \text{(F5-1)}$$

$$F5_S = (\text{not } E \text{ and } ((\text{not } A \text{ and not } D) \text{ or } (A \text{ and } C)) \text{ or } (E \text{ and } (\text{not } D \text{ or } (A \text{ and } C)) \qquad \text{(F5-2)}$$

Is the result optimal? It is not. A direct minimization of the whole KM shows that the $F5_{S0}$ alone would cover all the '1's.

$$F5_m = (\text{not } A \text{ and not } D) \text{ or } (A \text{ and } C) = F5_{S0} \qquad \text{(F5-3)}$$

We can also try to apply OR implicants:

$F5_M$ = (not A or C) and (A or not D) (F5-4)

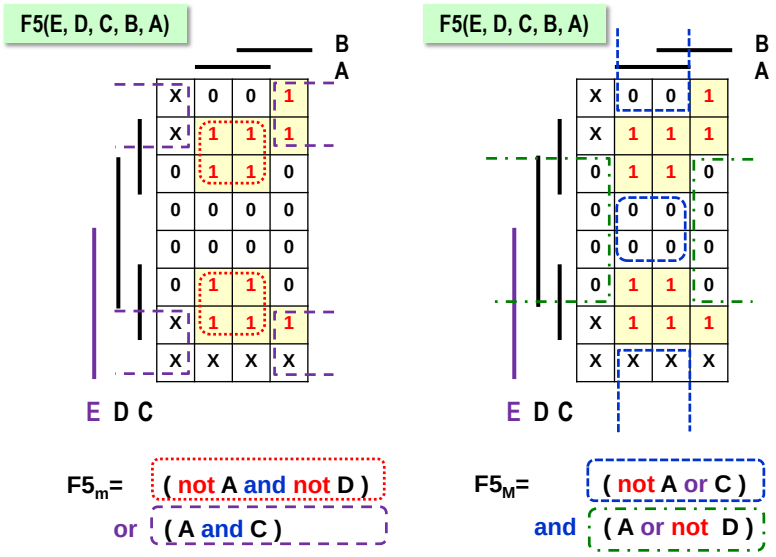


Figure 43 - Direct minimization of F5

The logical $F5_m$ obtained from (F5-3) and the $F5_M$ given by (F5-4) are logically equal. The SoP method $F5_m$ (coverage '1') redefined the included *don't care* to '1' the others to '0'. In contrast, the PoS method $F5_M$ (coverage '0') specified covered *don't care* to '0', non-covered to '1', but with identical results.

The SoP in $F5_S$, decomposing the table by the Shannon expansion, gives a different result because it covers other *don't cares*. However, all three logic functions agree in the prescribed '0' and '1', which is the most important.

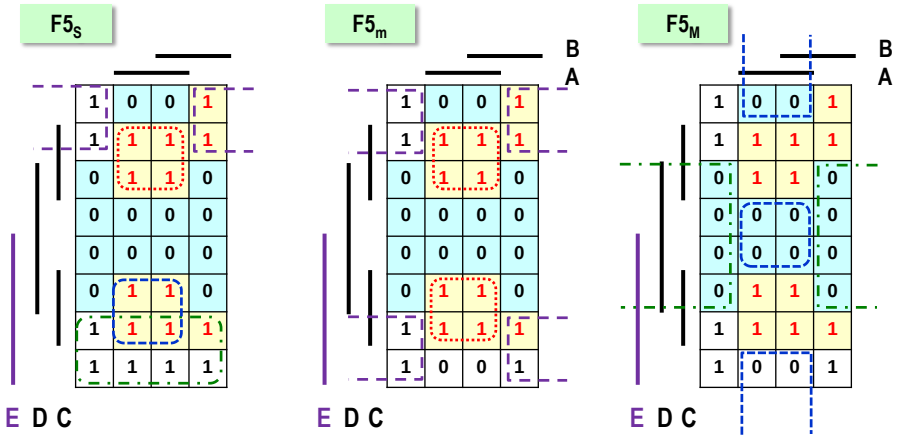


Figure 44 - Comparison of F5 results

For larger KMs, the division by the chosen variable can be repeated until we reach the 4x4 dimension, i.e., the KM of a logical function of 4 variables.

However, if we want perfect optimization, we entrust logical functions with 5 or more variables to a minimization program. Even if we lose time entering the values '0', '1' and don't care, the result will be free of the errors that we already risk with larger KMs due to the discontinuous placement of their implicants.

We will show further applications of the significant **Shannon expansion** on p. 52, in Problem 3.

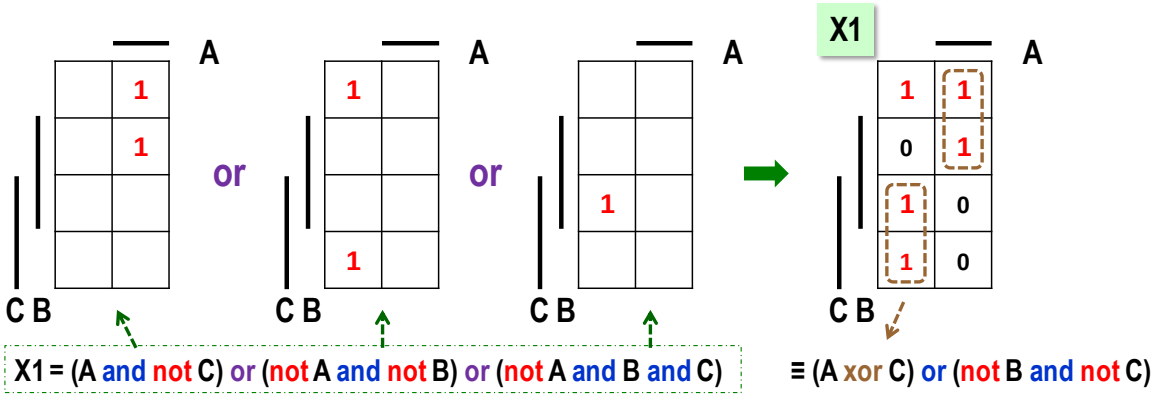
3.4 Using Karnaugh maps to evaluate a logic function

3.4.1 Task 1: Use SoP to determine the KM of a logic function

X1 = (A and not C) or (not A and not B) or (not A and B and C)

We can insert values after A, B, C and evaluate the function, but this will be tedious and with the risk of unwanted errors. However, if we notice that the expression X1 is of the form SoP (coverage '1'), we solve it fast. We draw an empty Karnaugh function map of the 3 input variables and enter the '1' generated by each implicant. The other fields will be '0'.[9]

The logical '1' **(A and not C) of** the initial implicant **lie under** A and are **not next to** C. The other expressions are constructed analogously. If we still remember the XOR function, we truncate the expression in the final map on the right.

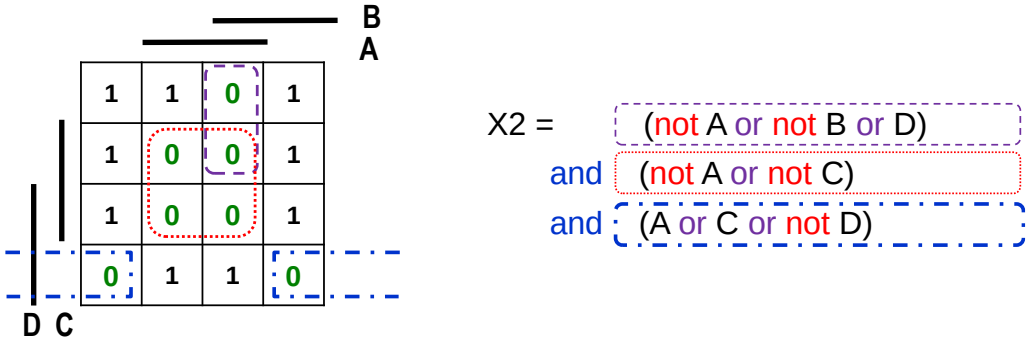


3.4.2 Task 2: Use PoS to create a KM logic function:

X2 = (not A or not B or D) and (not A or not C) and (A or C or not D)

Now, the expression has PoS form (coverage '0'). So, we draw an empty KM of the four variables' logical functions and enter the OR implicants' outputs. Here, we are careful to note that the not operator stands in front of the variables in the opposite situation to that of the AND implicants.

Thus, the first implicant (not A or not B or D) is **under** A, **under** B and not next to D, emphasized in the figure below by a violet dashed border. The next one is again constructed analogously. The unfilled fields will be logical '1' since the PoS method covers '0'.



[9] Recall that don't care symbols cannot appear in the output of fully designed logic function, as they denote a deferred value decision. However, their selection has already occurred during the construction of the logical expression. In it, they are already decided long ago.

3.4.3 Task 3: Use Shannon expansion to calculate the logic function

Y5(A,B,C,D,E) = (A and D) or (A and not B and E) or (not A and E) or (not C and not E)

The function has 5 input variables and calls for a computer solution. If it is processed manually, it can be reduced **by Shannon expansion**.

We create two functions, the first by substituting '0' after E and the second by substituting '1' after E, thus splitting the truth table in half.

$Y5_0$ =Y5(A,B,C,D,'0') = (A and D) or (A and not B and '0') or (not A and '0') or (not C and not '0')

= (A and D) or (not C and '1')

= (A and D) or not C

$Y5_1$ =Y5(A,B,C,D,'1') = (A and D) or (A and not B and '1') or (not A and '1') or (not C and not '1')

= (A and D) or (A and not B) or not A or (not C and '0')

= (A and D) or (A and not B) or not A

Both functions have SoP shapes. The implicants of the logic function found $Y5_0$ fill the upper part of the Karnaugh map of 5 variables, in which E='0'. Using the implicants of $Y5_1$ we create the lower part where E='1'. They can be written directly into the resulting KM.

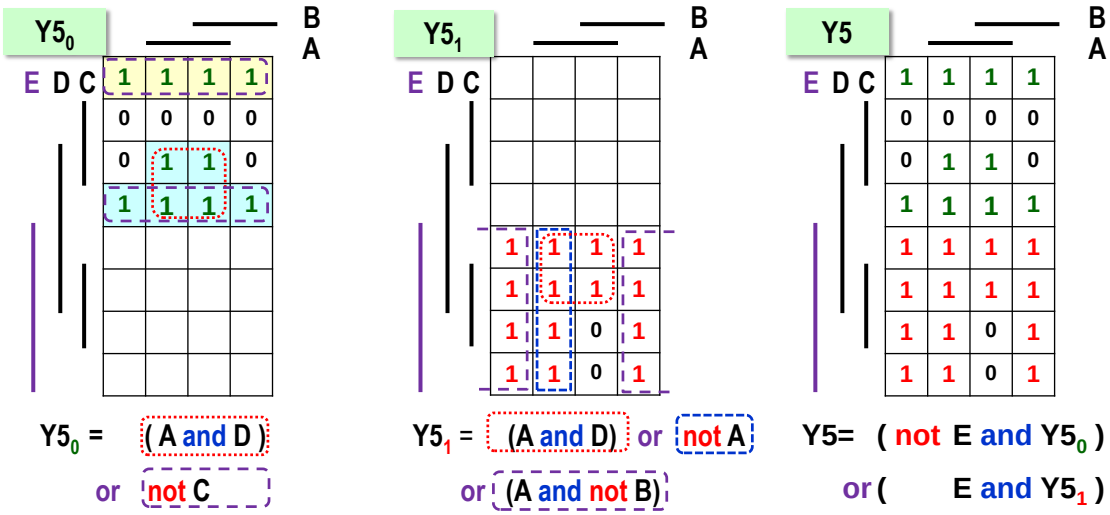


Figure 45 - Using the Shannon expansion

The function Y5 was composed as **Y5 = (not E and $Y5_0$) or (E and $Y5_1$)**, i.e., SoP coverage style. It represents the notation in the form of a Shannon identity.

We proceed analogously to the example above if we have $f(x_n , x_{n-1} ,..., x_1 , x_0)$ Boolean function. We choose a suitable variable, for example x_n , and decompose the function f() by Shannon expansion, i.e., simply substituting '0' and '1' after x_n . We get f_{0xn} ('0',x_{n-1} ,..., x_1 , x_0) and f_{1xn} ('1',x_{n-1} ,..., x_1 , x_0), which are called the Shannon cofactors of f() with respect to x_n , or Shannon cofactors of f() with respect to x_n .

The original function can be decomposed using the Shannon identity as:

$$f(x_n , x_{n-1} ,..., x_1 , x_0) = (\text{not } x_n \text{ and } f_{0n} ('0',x_{n-1} ,..., x_1 , x_0)) \text{ or } (x_n \text{ and } f_{1n} ('1',x_{n-1} ,..., x_{10}))$$

If we draw the identity as a diagram, we see that it describes a 2:1 multiplexor. Multiplexers will be the subject of a later chapter 5.3 on p. 82.

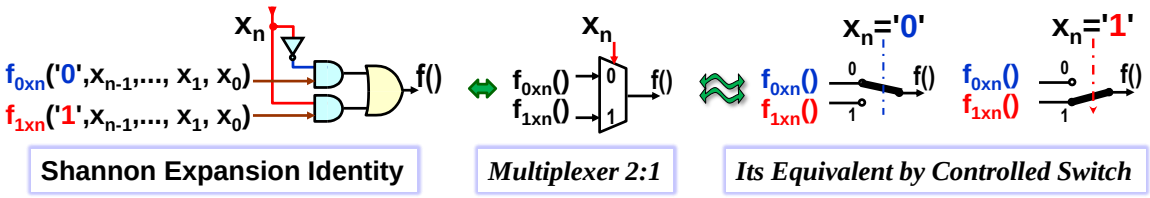


Figure 46 - Shannon expansion

The cofactors can be decomposed by subsequent expansions into even simpler ones with less extensive KMs, i.e., smaller truth tables, perhaps of up to one variable, thus performing the enumeration of the logic function.

In computing, Shannon expansions produce **BDDs**, Binary Decision Diagrams[10] , a powerful tool in, for example, program verification and elsewhere where many logical expressions are repeatedly enumerated. There is also a selection of freeware BDD libraries for common programming languages.

Note: The Shannon expansion gives results that depend on the choice of the variable under which we decompose. Because of this, the heuristic is to start from a variable such that the maximum of the members of the logistic function is eliminated. Even so, the complexity of the cofactors may not always decrease. Some combinatorial logic functions don't even have a suitable decomposition, such as adders or multipliers, although these are the ones for which we could use one:-) So the Shannon expansion can only simplify a ***subset of*** *logic functions.*

[10] See for example: https://en.wikipedia.org/wiki/Binary_decision_diagram

3.4.4 Task 4: Simplify the expression

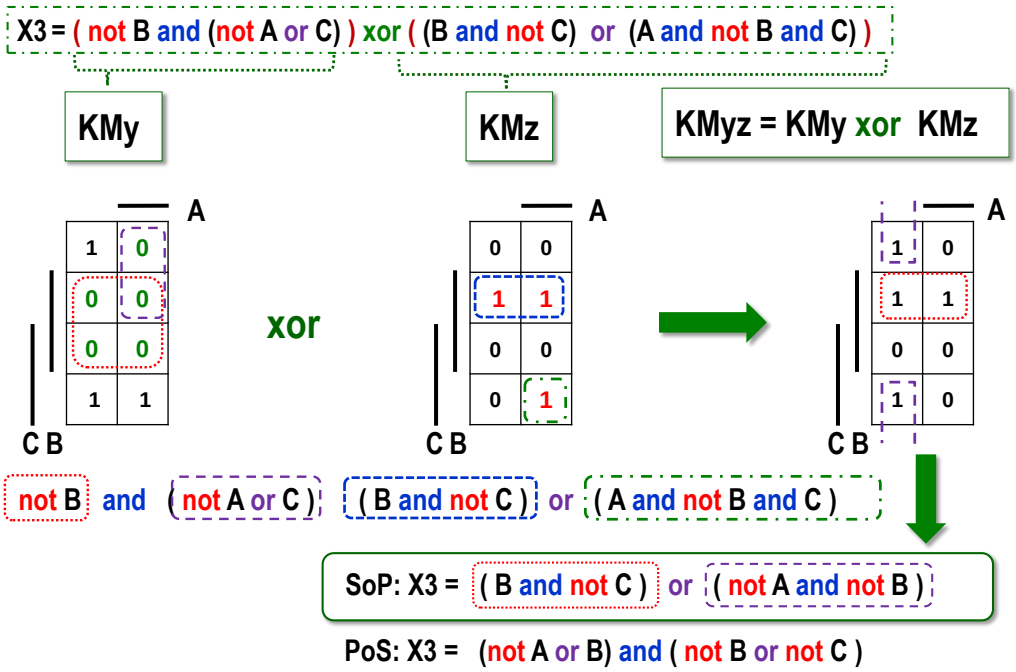
X3 = (not B and (not A or C)) xor ((B and not C) or (A and not B and C)) (xr1)

The xor function complicates the operation. It can be broken down according to the relation from the chapter 2.3.1 on p. 23:

Y xor Z = (Y and not Z) or (not Y and Z) (xr2)

but we would be substituting expressions for the Y and Z terms, which would only complicate the equation.

We'd better try the KM-trick. Let's create Karnaugh maps KMy and KMz of the Y and Z terms of the function xor. From these we then calculate the resulting KMyz. The expression KMy (the left one in the function xor), has the form PoS (coverage '0'). So we write '0' according to the OR implicants and fill in the remaining fields to '1'. The right-hand expression of xor corresponds to SoP (coverage '1'), so we fill in '1' according to the AND implicants and fill in the rest to '0'.



The resulting KMyz is assembled in a purely mechanical way, patch by patch. After all, we know that xor outputs '1' for an odd number of '1's on its inputs, so we write '1' in those boxes of KMyz where KMy will be different from KMz, inserting '0' when their values match.

The final KMyz can be converted to an expression using the SoP method (coverage '1'), but PoS can also be used.

3.5 Computer minimization algorithms

In the sections on minimization, we mentioned computer algorithms several times. We will not discuss in detail their exact algorithmization, which is covered more thoroughly in other publications, but we will only list the properties of the most well-known tools.

- **The Quine-McCluskey method** works analogously to covering logical '1's with the SoP method but in numerical form. It starts from a list of '1' and don't care rows of a given truth table, i.e., from an on-set style description, see p. 32, which specifies minterms covering a single KM element.

Among them, it searches for members that differ only in negating a single Boolean variable, i.e., following the procedure of p. 39. These are used to construct all possible coverages of two elements. The result is used to search for all coverages of four members, and so on until any possible merge can be found.

The result is a list of **primary implicants**, i.e., the maximum possible ones. Finally, a suitable coverage of all the specified '1's is selected from these. The running time of the method depends on the number of terms in the simplified expressions. The maximum complexity can be up to $O(3^N / \sqrt{N})$, where N is the number of input variables[11] .

- Professional tools use, for example, the **Espresso** algorithm[12] , which manipulates logical cubes using heuristics. It will usually terminate in a fraction of the time compared to running the Quine-McCluskey method. It will find an optimal form for small Boolean functions, but for large ones with hundreds of variables, it will only ever present a solution. It will not finish its run in a reasonable time for complicated assignments.
- Another algorithm is, for example, **Boom**[13] , which works with tree structures and can often find solutions even faster than Espresso, but also not always.

Why was the minimization of logic functions discussed? The computer can do it all!

There are several reasons:

- Of course, we can enter our idea of a logic function into the design environment using the output list of its truth table. It is sometimes done. For example, we describe by such way decoders for a 7-segment display. Simple functions are more easily expressed in logical expressions, which often provide a better understanding of their behavior. The sequence of '0' and '1' outputs in their truth tables rarely indicate anything.
- The truth tables of logic functions grow exponentially with the number of variables. The best algorithm does not change the essence of minimization, whose complexity is among the NP-complete problems[14] . Even the added heuristics cannot solve extremely complicated functions in an acceptable time in all cases. Some of them necessarily be decomposed into smaller subparts, which requires understanding the available building blocks, the topic of Chapter 5.
- Here, we must also mention that some combinatorial logic functions have too isolated implicants that cannot be combined with others, so we obtain a massive count of implicants. Their minimization does not make sense. The truth tables of adders are the most common examples. We present them on page 97. Such logical functions must be decomposed into smaller blocks to obtain useable results. Thus, we need to know the logical functions as suitable building blocks.

---

[11] *Wikipedia* gives a brief description of the method, and the Banerji article discusses it in detail with C code. S.: Computer Simulation Codes for the Quine-McCluskey Method of Logic Minimization, 2014, available at https://arxiv.org/pdf/1404.3349. Open source codes can also be found, for example *Github*.

[12] See https://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer Its source code is also listed on Github.

[13] J. Hlavicka and P. Fiser, "BOOM-a heuristic Boolean minimizer," *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, 2001, pp. 439-442, doi: 10.1109/ICCAD.2001.968667

[14] For example, the specification of the NP-complete problem is mentioned on the wiki: https://en.wikipedia.org/wiki/NP-completeness

4 Implementation of logic gates

So far, we have assumed that we have ideal logic gates. The real ones are built with various technologies, for example, pneumatic systems or relays. Still, they are exceptions designed for harsh environments where strong electromagnetic interference cannot be excluded, e.g., industrial production. Transistors otherwise form logic gates.

Occasionally, bipolar transistors are still used in some parts of circuits, and then we talk about TTL (Transistor-Transistor-Logic) or LVTTL (Low voltage TTL). For example, the Cyclone II and IV families of FPGAs form external inputs and outputs with LVTTL because of its higher electrostatic breakdown resistance and power level. However, this is a less common solution.

Usually, the logic is implemented by unipolar CMOS, Complementary Metal-Oxide-Semiconductor transistors, pronounced "sea-moss." If we will be their users, then we only need to know some of their properties. First, there is the time delay, a critical parameter in most designs. And the complexity of the circuitry is also worth considering.

So, let's take a quick look inside CMOS gates, which are built based on semiconductors. Readers are probably already familiar with these, but perhaps recalling some of their properties relevant to explaining CMOS transistors doesn't hurt.

4.1 A reminder of the properties of semiconductors

The basis of semiconductors are usually elements that have four valence electrons, today often silicon (Silicon Si), but also GaAs (Gallium Arsenide). Their normal non-conductivity is changed by doping, i.e., by adding small amounts of another substance.

If we want to create an N-type semiconductor, we add an element with five valence electrons. Four are bound to silicon, but the fifth remains a free electron, which carries a negative charge and can participate in the conduction of electric current. A P-type semiconductor is formed by adding an element with three valence electrons, but only three will bind. At the fourth position, one electron will be missing, creating a hole carrying a positive charge, which may be involved in conducting an electric current.

The difference in doping strength, i.e., the proportion of impurity, is highlighted with a + sign if it is heavy and a minus sign for a weaker. For example, n+ indicates a semiconductor with more dopant atoms than n, while a p- semiconductor has less dopant atoms than p.

It is significant for CMOS that free electrons in N and holes in P semiconductors are their majority carriers. Still, due to impurities, there are also **minority opposite carriers** in them.

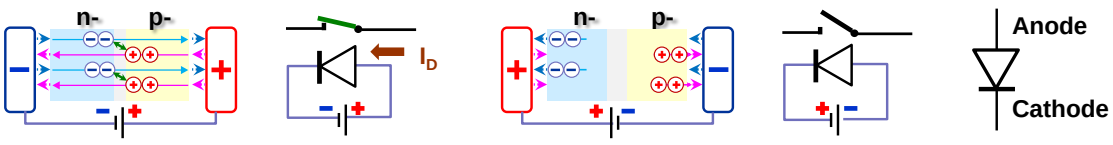


Figure 47 - Diode principle

PN diode is formed by connecting P and N semiconductors with weak doping. Between them there is a thin transition, the depletion zone, in which neither free electrons nor holes exist. Roughly speaking, they cancel each other, so the area isolates the P and N semiconductors from each other.

If the positive pole is connected to the P semiconductor, the anode of the diode, then it repels

holes and attracts free electrons to itself. A negative voltage applied to the N semiconductor, the cathode of the diode, in turn, repels free electrons and attracts holes. The concentration of carriers near the transition region between the semiconductors increases, the depletion zone becomes thinner, and holes and free electrons exchange and move. The diode leads. When the voltage is applied to it in reverse, the holes are attracted to the negative pole and the electrons to the positive. The non-conductive depletion zone expands, and the diode does not conduct.

A suitable combination of NPN or PNP semiconductors forms a bipolar transistor. Think of it as 2 diodes joined anode to anode at the base lead for an NPN transistor and cathode to cathode for a PNP. When the base is disconnected, they do not conduct.

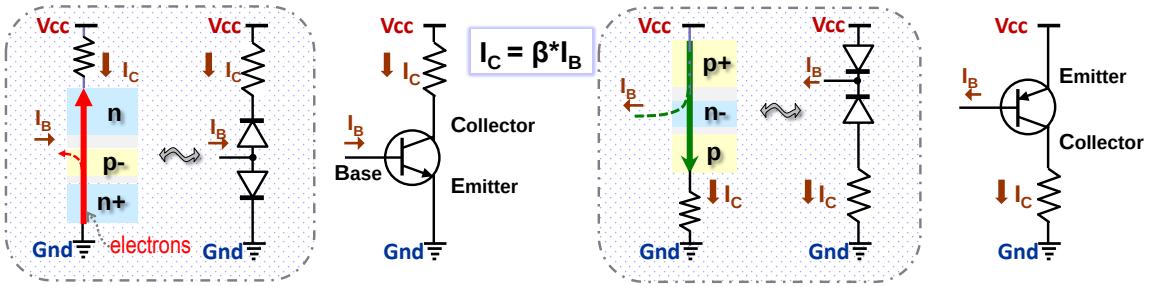


Figure 48 - Bipolar transistor principle

However, if an electric current flows between its base (B) and emitter (E), then in an NPN transistor, the depleted base becomes crowded with free electrons flowing from the heavily doped emitter [15]. Once the depletion barriers between the semiconductors are weakened, electrons flow from the emitter directly to the collector. In a PNP transistor, on the other hand, these are holes, and they move in the direction of the current.

The name of the transistor pins comes from the movement of the majority carriers. The emitter emits electrons, which are collected by its collector. Note that an NPN transistor has its collector in its typical circuit, facing Vcc, while a PNP transistor has its emitter at Vcc.

4.2 CMOS principle

There are two basic types, NMOS (N-channel MOSFET) and PMOS (P-channel MOSFET), which **use a minority carrier-based conduction channel**. The P-type semiconductor, the substrate of NMOS, has holes as its majority carriers. Still, the conduction channel is formed under the electrode G, to which the minority carriers are electrons that are attracted by voltage. PMOS has an N-type substrate; in it, the electrons are the majority carriers and the holes are the minority carriers.

Several CMOS technologies exist, distinguished by transistor geometry and applied dopants. Their description and analysis are beyond the scope of our textbook, so we will only mention the basic facts. With heavier doping levels, we decrease the semiconductor's resistance and the carriers' mobility, so CMOS uses base substrates with very weak doping, i.e., high resistance and mobility of their majority and minority carriers.

All technological modes can be divided into enhancement and depletion, which differ only by fabricating a partially conductive channel between electrodes S and D created by weak N or P doping. Thus, the transistor is slightly conductive in the initial state, not entirely, but only

[15] Recall that electrons move against the direction of the imaginary electric current. In 1752, long before their discovery, Ben Franklin chose the opposite direction. He left because of a number of pre-existing lessons.

half. The enhancement technology does not create a conductive channel, so CMOS is non-conductive at rest. We will explain **the origin of the names** in the following paragraphs.

The electrode names of CMOS transistors are based on the carrier motions. The **Source electrode, S**, is their source, similar to the emitter of bipolar transistors, while the **Drain, D**, will be their receiver, thus analogous to the collector. Since in NMOS, the carriers in the conduction channel are negative free electrons, its S electrode needs a lower voltage than D. PMOS wants a higher voltage on S than on D since the carriers in its channel are positive holes.

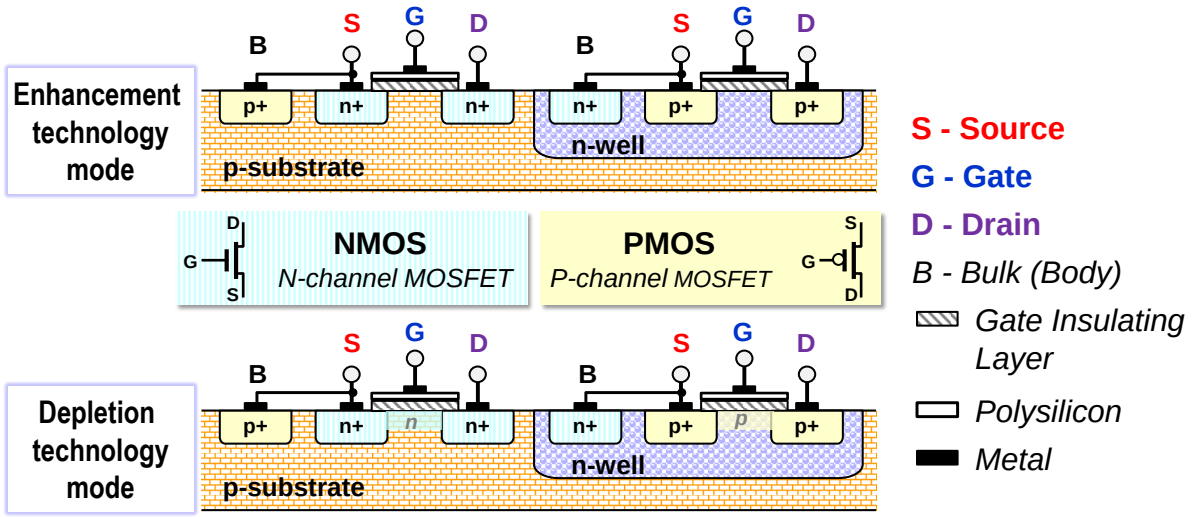


Figure 49 - Basic CMOS Technology

CMOS electrode **G, Gate**, affects the conductivity between S and D electric fields. The electrostatic force under G attracts minority carriers. In enhancement technology, a conductive channel is formed under G when the voltage across it is higher than the threshold voltage. Its conductivity then depends nonlinearly on the voltage. It is enhanced by it, hence the **technology name.**

Auxiliary electrode **B (Body)** is an external pin only in CMOS transistors manufactured as separate discrete components. Inside integrated circuits, it is internally connected to electrode S (Source). Its existence creates the voltage conditions in the substrate to turn on only after the threshold voltage at G (Gate) is exceeded. The first CMOS, discovered in 1959, had no B and behaved like a voltage-controlled resistor throughout its range. The addition of B improved its characteristics as a switch.

Depletion technology was introduced in the 1970s as an improvement. Its transistors partially conduct at 0 V to G. By applying a voltage to it, the conductance of their channel is either boosted, as in enhancement CMOS. However, a **depletion** mode was added, where the opposite voltage on the G electrode weakens the initial channel conductance and expands the non-conducting depletion region, hence the **technology name**. However, it needs a dual power supply, positive and negative. Its CMOS exhibits zero residual current between S and D due to perfect closure and higher resistance to electrostatic breakdown.

In today's ICs, however, CMOS enhancement is preferred because it switches faster and only needs a positive power supply. Depletion CMOS is still manufactured, but mainly to replace resistors due to their partial conductivity at 0 V. They are also used to form analog-oriented parts of circuits, such as voltage-controlled resistors. They are also suitable for current sources.

There are several reasons for preferring CMOS enhancement in gates, not just their faster switching. In technologies below 180 nm, the residual current between S and D has dropped to a negligible value compared to other parasitic CMOS phenomena, such as quantum tunneling effects in semiconductors.

Depletion CMOS has thus lost its main advantage. In addition, they have conductivity at 0 V to G, both in NMOS and PMOS, which can occur unintentionally and for a long time during accidental power failure or signal attenuation. Then, a circuit overheats and is destroyed. Enhancement CMOS has no such drawbacks.

4.2.1 CMOS Brands

The following figure shows the general schematic CMOS symbols and others with the specifications of the technology mode behind them. The triangles in the middle symbols do not stand for arrows but **diodes**. Depending on their position to G, they specify either an NP transition, i.e., NMOS, or a PN transition for PMOS.

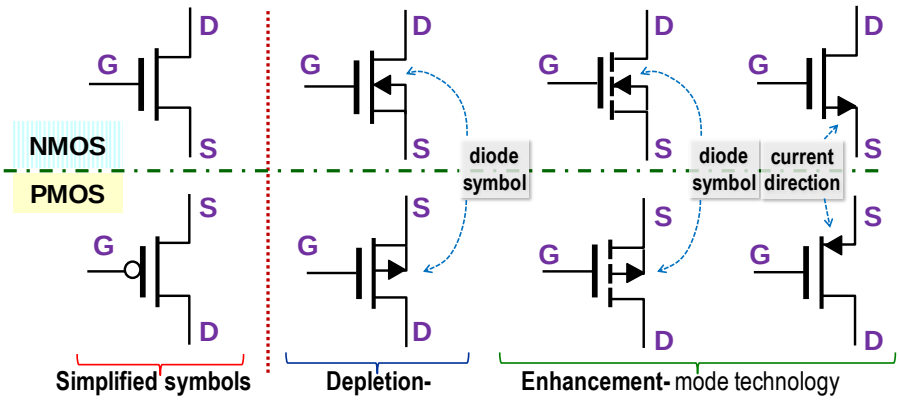


Figure 50 - Overview of CMOS transistor brands

In contrast, the symbol on the right with three pins, recommended by the IEEE standard, assumes an internal connection between the S and B electrodes. Here, arrows specify the direction of movement of the electric current, i.e., analogous to the markings of NPN and PNP bipolar transistors.

In the following, we will only use simplified symbols in which the bubble at the input of PMOS indicates its opposite behavior to NMOS.

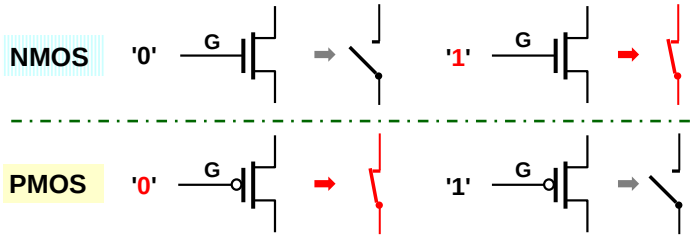


Figure 51 - CMOS transistors as switches

- **NMOS is closed in logic '1'.** Its prefix N means that a free electron (negative) based conductive channel is formed by G voltage. Roughly speaking, a positive voltage at its control electrode (gate) attracts electrons, and the channel becomes conductive.
- **PMOS, on the other hand, opens in logic '1'**, which is also indicated by the inverter bubble in front of its control input G. The prefix P suggests that a semiconductor channel of positive holes is formed by electrode G voltage. A positive '1 voltage at G repels holes, and the channel closes.

- **The voltage controls the conductivity of the CMOS transistors**. An electric field controls their switching on and off. In an ideal CMOS, current does not flow into its control electrode G (gate), but in a real CMOS, it increases unpleasantly as the nanometer technology decreases.
- **The control electrode G must always be connected!** If G is not connected anywhere, it is a floating input and the risk of damaging the circuit increases.

In general, we cannot consider "nothing" as equivalent to a voltage of 0 V, or a logic '0', and of course not even a '1'. Both logic '0' and '1' are stiff voltage sources. They have low internal resistance and change their voltage only slightly with load. Any unconnected input, on the other hand, has a high input resistance. Disturbing peaks are easily induced and cause the gate to knock randomly, which dangerously increases both its draw from the source and problematic voltage peaks. The reason for this will be discussed on p. 66.

Note: Some publications state that an unconnected logic gate input behaves as '1'. It could be only for bipolar TTL technology, but even here, it was always advised to connect all inputs.

Switches allow a subset of logic functions to be implemented, certainly not all of them, but they can express implicants. The serial connection of switches describes an AND operation and a parallel OR operation, as shown in the figure below. It is based on the idea that an input variable with a logical value of '0' does not press the switch, whereas at '1' it does[16] . Thus, negated variables are represented by the using on or off buttons:

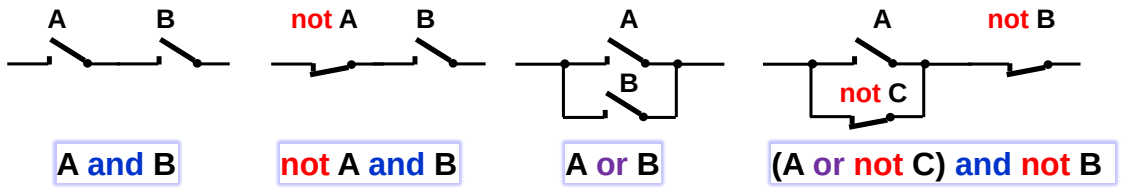


Figure 52 - Logic using switches

However, we must ensure that the gate output remains connected to the voltage each time, either '0' or '1', as it will lead to the downstream CMOS inputs, and these must be kept at defined levels. So, we will use two groups of switches, thus accelerating the switching process as well. When its condition is met, the upper group connects the output to Vcc, i.e., to '1', according to the required logic function. The lower one is then its negation.

4.3 Inverter and buffer

The inverter has an not X, i.e., PMOS, in its upper group, and X, i.e. ,NMOS, in its lower group, by which the output connects to '0' on Gnd when the upper condition is not met. The figure below shows the CMOS implementation of the inverter and its switching analogy when its X is input at '0' and '1'.

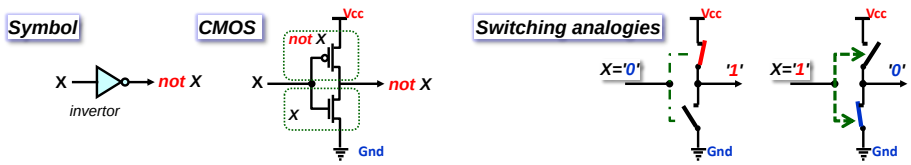


Figure 53 - CMOS inverter

[16] The switch analogy is taken from ladder logic, i.e., the graphical language of today's industrial PLCs, Programmable logic controllers.

The buffer is opposite to the inverter. It copies the input to the output, perhaps for decoupling and current boost or to increase the time delay of a logic path. Its direct wiring does not work well. We create its faster version with two inverters in a series, placed close together, i.e., connected by a wire of negligible length. Paradoxically, the signal passes through them sooner.

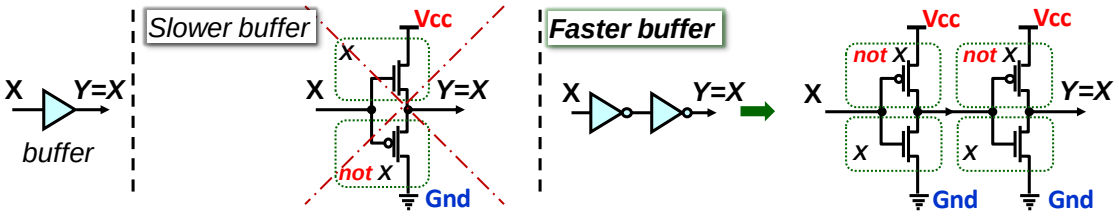


Figure 54 - CMOS buffer

But why? The reasons require a deeper understanding of the physical characteristics of CMOS transistors, which we will leave to other publications. We will only lightly outline the main reasons.

- While holes in semiconductors maintain the direction of the imaginary electric current, electrons move against it, from the negative pole to the positive. Thus, NMOS types with an electron-based conduction channel work better in the lower group, and PMOS, where the conduction channel is made of holes in the upper group. **NMOS should only be used in the lower group and PMOS in the upper group.**
- Analog technology knows circuits similar to the buffer gate shown on the left in the Figure. Their output rises/falls with the input voltage, and their gain (gain) stays below 1. However, analog signal levels hover around the middle of the supply voltage, but the logic needs the fastest possible transitions to their extreme states of Vcc and GND.
- The inverted versions of the gates change the voltage of their outputs in the opposite direction to the inputs. The upper or lower CMOS groups are favorably affected in the inverted version. A voltage change, decrease or increase, on one group will affect the opposite towards accelerating its action, i.e., speed up flipping by positive feedback.
- Each inverter has a gain of roughly ten or more during its run, measured with analog eyes. Thus, two inverters in a row separate the input from the loads behind the output and improve the signal edges' steepness.

4.4 Logic gates AND, NAND, OR, and NOR

A NAND gate has the logic function not (X and Y) = not X or not Y after decomposition by De Morgan's theorem. This will be in the upper group, and we plug its negation X and Y into the lower group.

The AND and OR gates are more often formed at the CMOS level from NAND and NOR, after which inverters are added for the reason discussed in the previous section. The result will be faster than directly forming AND and OR by unrecommended swapping the upper and lower groups of CMOS transistors.

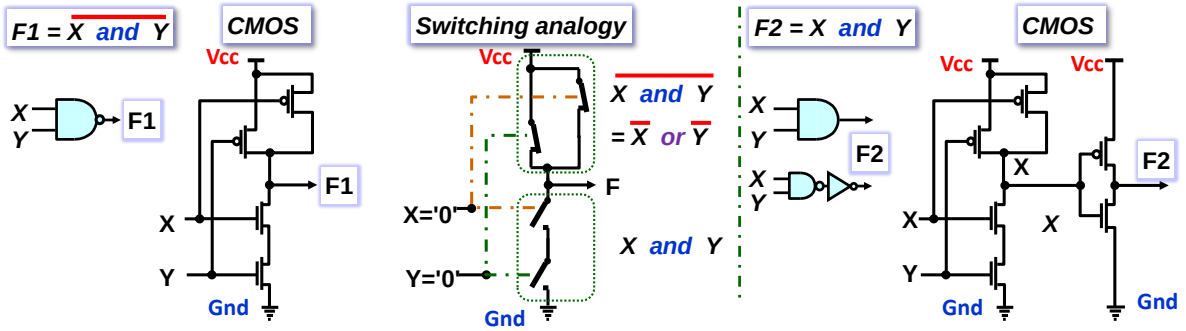


Figure 55 - NAND gate wiring and AND

The function of the NAND gate is shown in the figure below:

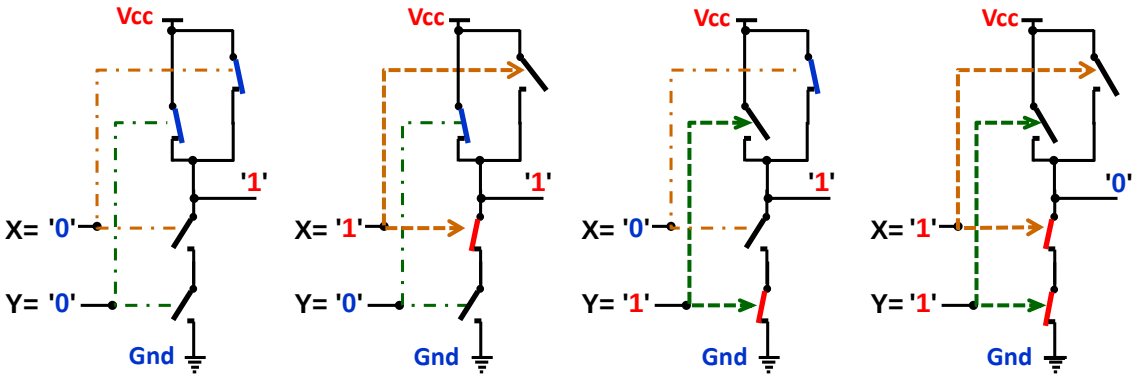


Figure 56 - NAND gate switch analogies

The multi-stage gates are formed by additional pairs of CMOS transistors in the upper and lower groups.

Figure 57 - NAND and OR multi-input gates

- The NOT, NAND, and NOR gates need one pair of CMOS transistors for each input.
- The AND, OR, and BUFFER gates also add an output inverter with a CMOS pair.
- **Multi-input gates are slower**. They have more transistors in series in their upper or lower group, which degrades the gate's ability to flip the output between '1' or '0'. We will show the reason for this in the chapter 4.8.3 on p. 69.

4.4.1 AND-OR scaffold

The gate design need not be limited to essential logical functions but can also evaluate more complex expressions. For example, let's build an AND-OR gate, which will be helpful for adders that need to propagate their carries to a higher order.

Using De Morgan's theorem, we convert its equation into a negated function, which has only PMOS in the upper CMOS group and only NMOS in the lower one.

C = (X and Y) or G = not not ((X and Y) or G) = not (not (X and Y) and not G)

= not((not X or not Y) and not G)

The upper group will be (not X or not Y) and not G; the lower by its negation: (X and Y) or G.

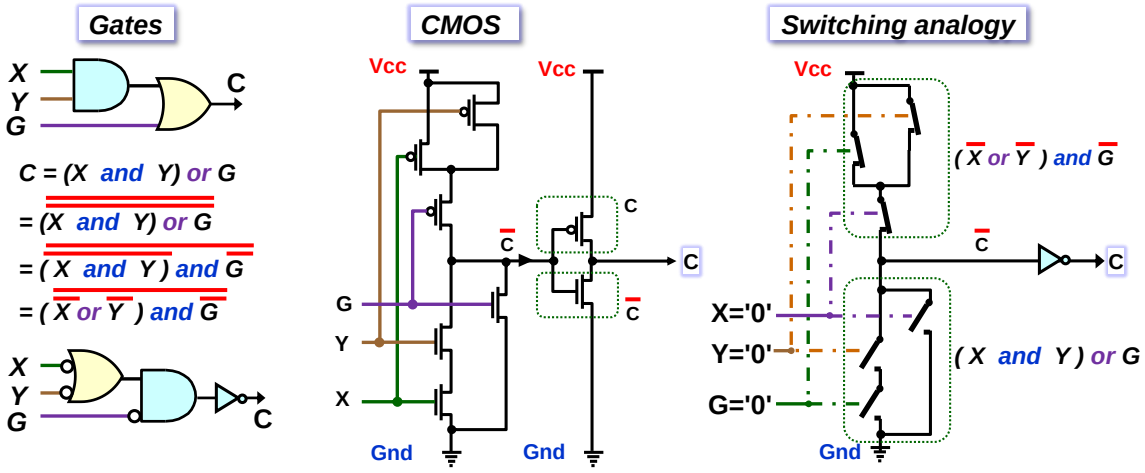


Figure 58 - AND-OR railing

4.5 Transmission gate

The term transmission gate is sometimes referred to as a pass-through gate, which has a structure similar to pass transistor logic, PTL, and is sometimes considered its synonym. However, in many publications, PTL is more closely associated with analog signal switching. The transmission gate suggests implementation optimized to transmit logic '0' and '1' levels.

It is an essential building block of integrated circuits as it acts as a bidirectional switch.

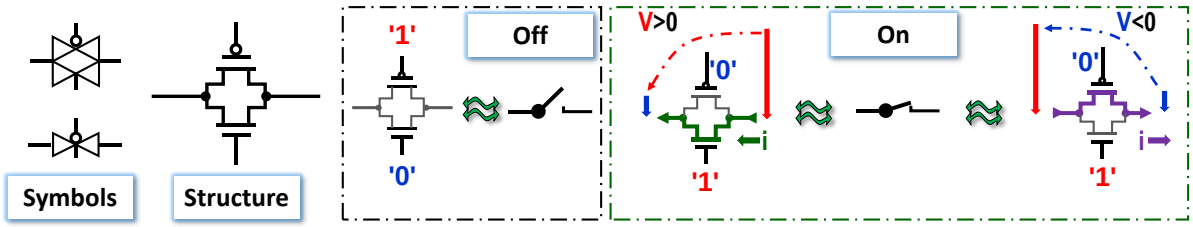


Figure 59 - Transmission gate or PTL

CMOS transistors conduct in both directions, but only better in one direction. In contrast, in the opposite direction, they are closing by dropping their threshold voltage at electrode G. So, we connect opposite types in parallel. They are both closed and mimic an open switch in the deactivated state. When activated, they behave according to the voltage between electrode G and their conducting channel.

- At a positive voltage, current flows through the NMOS, which is carried by negative electrons that move against the direction of our imaginary electric current. The positive right end attracts them from the negative left pole. The PMOS closes with increasing voltage.
- At negative voltage between the right and left ends, the PMOS, which carries the positive holes, takes over the current conduction. They flow in the direction of the imaginary current to the negative pole. NMOS, on the other hand, closes.

In the switched state, the transmission gate element has a resistance dependent on the nanometers of its technology, even in the hundreds of ohms for small ones. A little voltage is then lost on each one. You can't line up many of them in a row. They are mainly used to build the internal structure of integrated circuits, as this is where their exact loading is known. They are used to form internal multiplexers, synchronous circuits, and configurable jumpers in FPGAs.

4.6 XOR Gate

An XOR gate is a composite function XOR(X,Y) = (X and not Y) or (not X and Y) by SoP covering its KM. It is not suitable for direct wiring in CMOS. The positive and negative terms in its expression could only be realized by series coupling of NMOS and PMOS, which is not allowed.

So one must either add input inverters or cover the XOR function with logical '0's and apply De Morgan's rule to make the AND, OR and NAND gates work out better technologically.

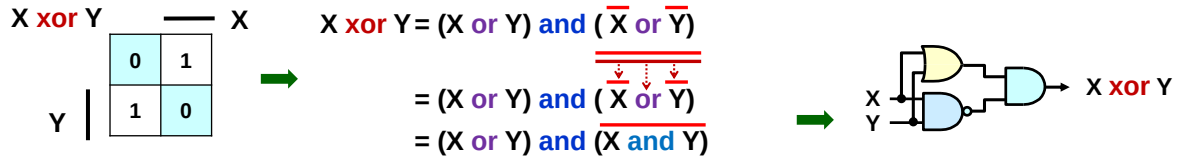


Figure 60 - XOR railing by PoS method

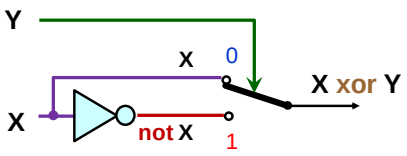
More convenient implementation uses XOR property as a controlled negation; see Chapter 2.3.1 on p. 23. We later generalized its decomposition to the Shannon expansion; see Chapter 3.4.3 on p. 49. For example, we choose the cofactors according to Y :

XOR(X,'0') = (X and not '0') or (not X and '0') = X
XOR(X,'1') = (X and not '1') or (not X and '1') = not X

So the XOR itself can be written:

XOR(X,Y) = (not Y and XOR(X,'0')) or (Y and XOR(X,'1'))

The Y variable thus controls a two-pole switch that sends either X to the output when Y='0' or not X when Y='1'. It can be implemented using transmission gates.



The output switch needs an inverter Y to drive it, but also two transmission gates. If the CMOS voltage characteristics are used, one can be omitted and block inverter X with the Y signal. The resulting circuit is implemented by what we will already call circuit magic. Its CMOS incantations☺ are explained in a technical article by the authors of its implementation[17] .

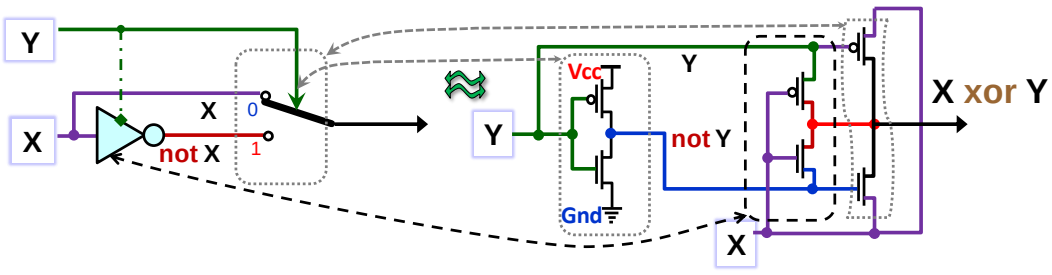


Figure 61 - XOR with 6 CMOS transistors

There are other tricks to make an XOR with only 4 CMOS transistors, the same complexity as an AND gate, and even versions with only 3 CMOS have been invented. [18]

The example was mainly used to demonstrate the wide possibilities of CMOS technology, in which many things can be integrated more conveniently than we can see in the diagrams.

---

[17] N. Ahmad and R. Hasan, "A new design of XOR-XNOR gates for low power application," *2011 International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, 2011, pp. 45-49, doi: 10.1109/ICEDSA.2011.5959039.

[18] Different ways of implementing the XOR gate are discussed in the article by Yann Guidon, Paris, France: https://hackaday.io/project/8449-hackaday-ttlers/log/150147-bipolar-xor-gate-with-only-2-transistors/

4.7 Three-state gate

The three-state gate, aka tri-state gate, is a building component of many circuits[19] . Its output can be brought into a high impedance state, for which the term 'Z' or hi-Z has been introduced. Its logic value can appear at outputs in addition to logic '0' and '1'.

Additional input, often called OE (Output Enable), disconnects the gate output, allowing other devices to drive the wire without interference from the tri-state buffer. If OE is inactive, the gate behaves like a regular buffer.

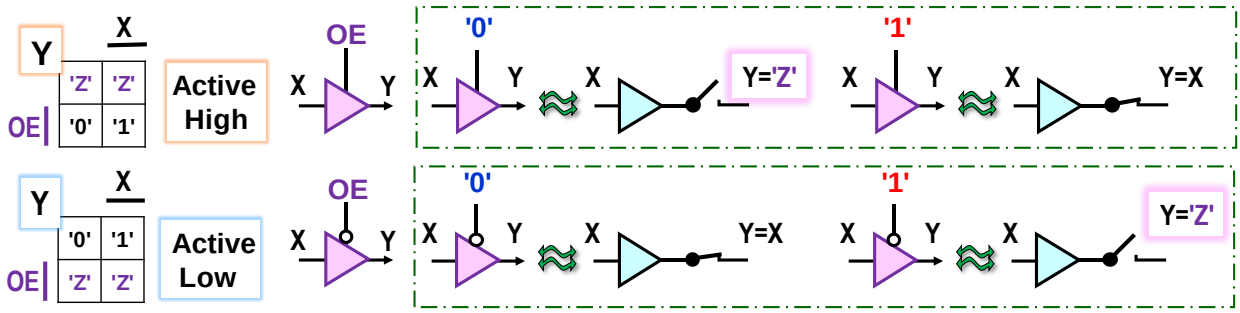


Figure 62 - Three-state buffer

Any gate can, of course, also be extended with the possibility of three-state. For example, Logic Element, chapter 5.5.6, on page 90, utilizes three-state gates. A three-state inverter is often formed by the style on the left, adding decoupling to the upper and lower inverter-controlled groups.

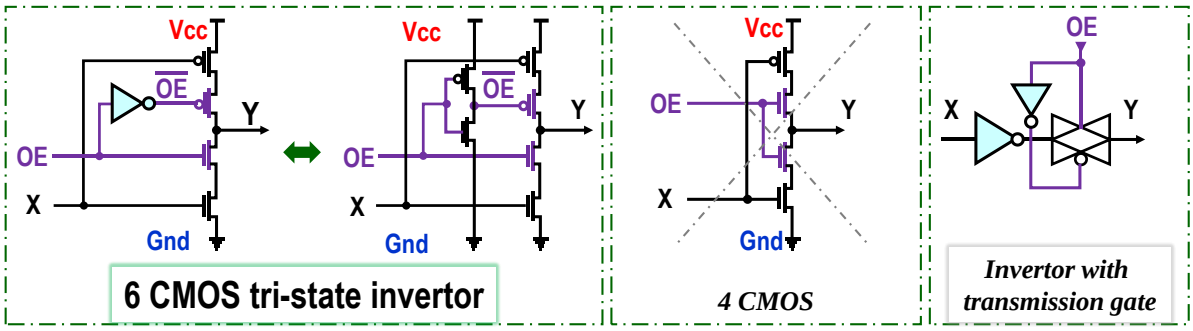


Figure 63 - Examples of some internal structures of a three-state inverter

The picture in the middle shows the illicit solution. If the internal inverter of the OE signal were replaced by using NMOS in the upper group, two transistors would be saved, but NMOS would be in the upper group, where it does not have the right conditions to operate, and in series with PMOS! The two types of CMOS may have close parameters, but they cannot be made to be identical just because they use different carriers. Holes in semiconductors have roughly one-third the mobility of electrons.

The option on the right is permissible, but only as an addition to a more complex logic function, not for a simple inverter. The transmission gate degrades the quality of the output.

[19] The three-state gate is used on bidirectional parallel computer buses, but these are now being phased out. Today's unidirectional serial lines do transfer data bit by bit, but paradoxically much faster as a result, for a number of reasons. For example, with a serial line we are not delayed by the time synchronization of several signals transmitted in parallel, and mutual interference, so-called crosstalk, where the electromagnetic field generated by the signal is induced into adjacent parallel wires, is also better suppressed. Transmission over a serial link is then commonly accelerated by using several of them, each carrying a portion of the data independently of the others. For example, the video output of the Display Port carries video over four serial lines, with another for audio and auxiliary information.

4.8 Dynamic model of two inverters

We first model the cascade of two CMOS inverters by their physical analogy, using water channels, since the propagation of an electrical signal along a transmission line shows phenomena close to water, such as a gradual increase in voltage (level) and the appearance of waves due to reflections.

A great animation of the events on the conductor was at the time of writing this publication at the end of the article: https://practicalee.com/transmission-lines/, and it makes clear that water can mimic a specific subset of electrical phenomena.

4.8.1 Water model of two inverters

Mere switches will approximate both CMOS transistors, i.e., their most common use in logic. We emulate them with sliding gates; see Figure below. An open switch corresponds to a retracted gate; the water stops and does not flow, while a closed switch is an open gate, i.e., full flow. The cross-section of the channel emulates the resistance of the closed state. The wire capacitance represents the channel volume to be filled by water.

If we have two inverters in series, their initial state is shown in the figure below.

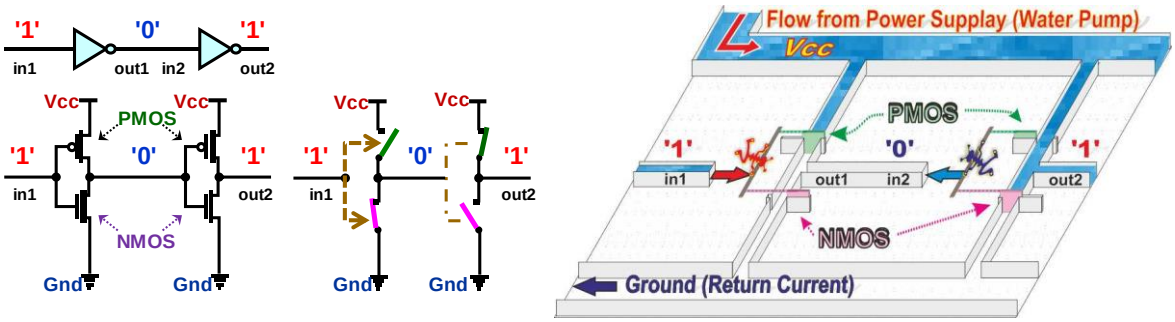


Figure 64 - Water model - initial state

The electric field pushes on the door at logic '1' while it pulls it out at '0'. The pushed PMOS gate blocks (disconnects) the channel. The pulled PMOS opens it. NMOS gates are retracted on the other side and work the opposite way.

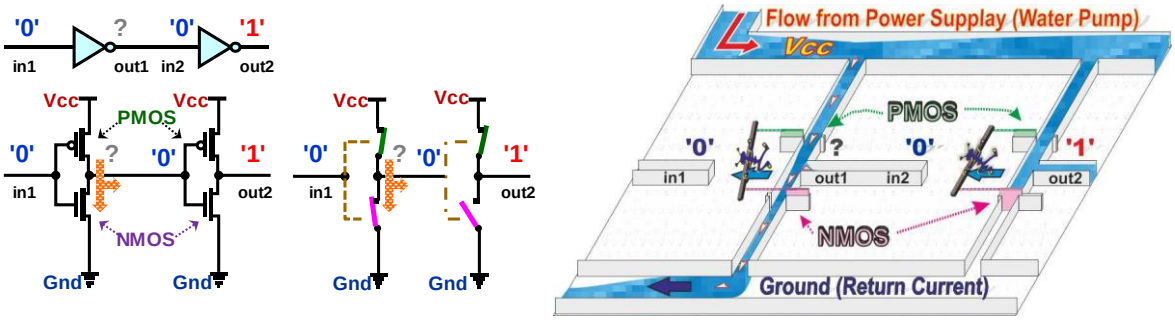


Figure 65 - Water model - temporary short circuit condition

A level decreased in channel in1 flips left inverter, but not immediately. The water flow speeds up the pulling of the upper PMOS gate and slows down the lower NMOS gate. The water fills the out1 channel, but most of it escapes through the short circuit current. Both CMOS gates are now fully open (in their saturation state). In the Vcc supply, the power level is temporarily reduced. The output voltage of out1 will now be '?', so somewhere between '1' and '0'.

A short circuit current appears every time the gate is switched, but it lasts only a few picoseconds in newer circuits. It contributes to the total power consumption in units of percent.

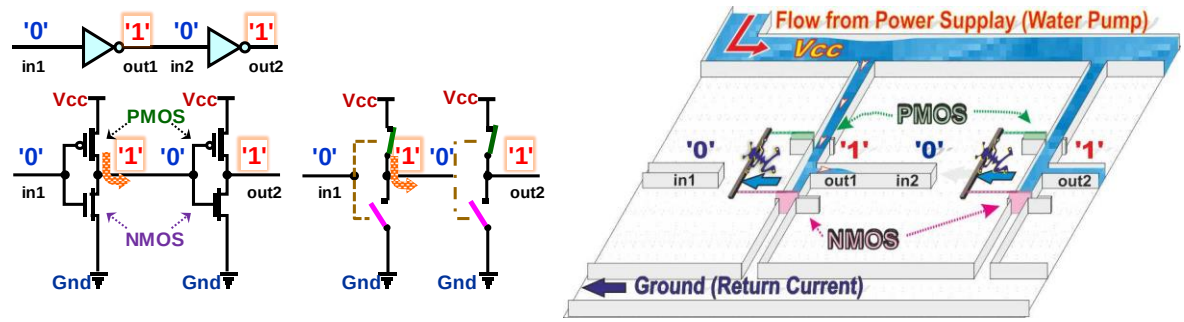


Figure 66 - Water model of two inverters - both gates in logic '1'

The left NMOS gate is completely closed, and water fills the channel from out1 to in2. Note that in **logic '1', the current flows outwards from the output of the driving gate.**

The right gate still has '0' at the input; it does not know about the change because the flood wave is still propagating through the channel from out1 to in2. At its end, the level has not yet risen above the 50% decision level, so the right inverter has not yet switched.

We have another intermediate state where **both inverters have identical outputs**. We will come back to this in a later interpretation of the metastability of flip-flop circuits.

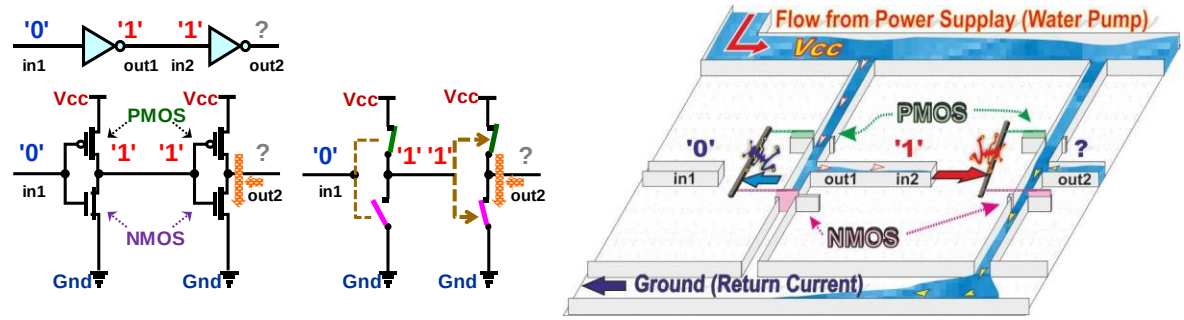


Figure 67 - Water model of two inverters - right gate in short circuit

The channel has already filled up, and the potential at its in2 end has increased to the level of logic '1'. The lower gate of the right inverter has been pulled faster by water pressure, but the upper gate has not yet closed. The right inverter has both transistors in a saturation state, with a **short circuit current** flowing through them for a few picoseconds.

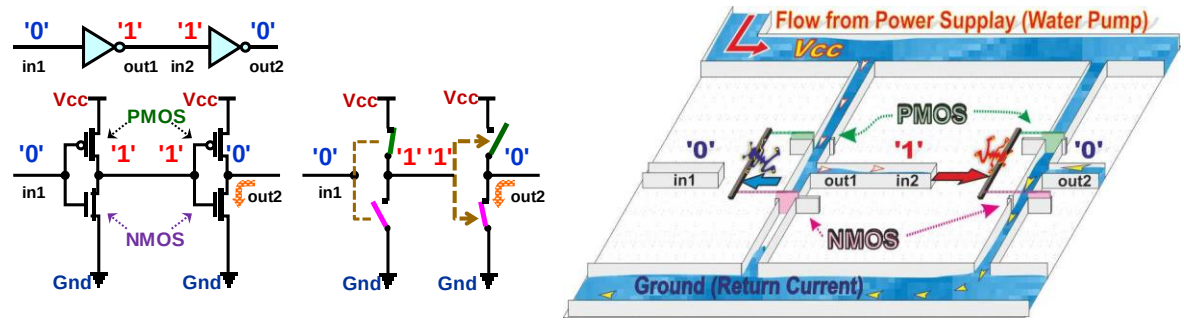


Figure 68 - Water model of two inverters - right gate switched

The upper PMOS gate has already closed in the right inverter, and the lower NMOS gate is leaking water. We already have a logic '0' at the out2 output of the right gate, which discharges its capacitance, so the change propagates downstream. The out2 channel is emptied into the Ground drain, where the level rises briefly before the surge is diverted.

Note that the **current in logic '0' flows into the output of the driving gate.**

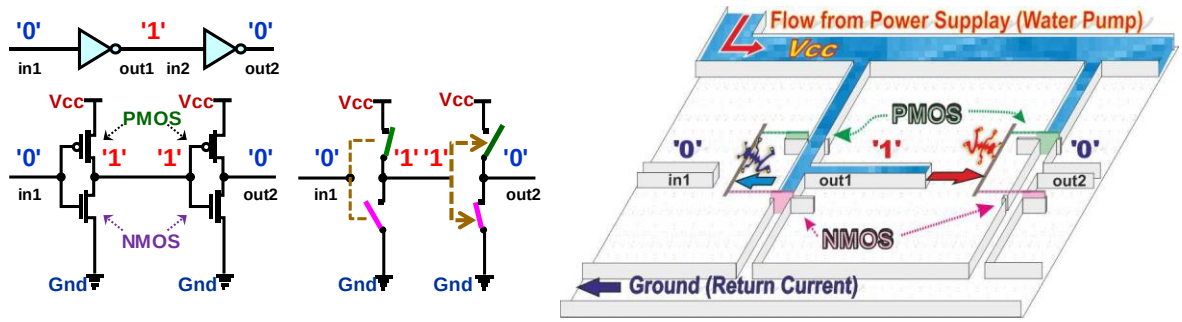


Figure 69 - Water model of two inverters - steady state

The output channel has already emptied and a steady state has occurred in which the inverters will remain until the next change in input in1.

The inaccuracy of our water model lies mainly in the influence of the switching by the mere height of the water level. The electrical voltage is the potential difference at two points. So, the pulling or pushing of the gates should correctly depend on the difference in level in the input channel relative to the state in the GND drain. A similar model is possible based on pressure differences but would lose illustrative power, so we have reduced it.

For the same reason, we also did not simulate the coupling between the upper gate and the lower gate, which occurs in logic gates between the PMOS transistors of the upper and NMOS of the lower group and accelerates the flip-flopping.

4.8.2 Gate static pickup

The water model demonstrated that the gates impact the power source as they flip. At rest, they consume only parasitic leakage currents due to tunneling effects in the semiconductors, called quantum tunneling sustained regardless of the state of the gate outputs.

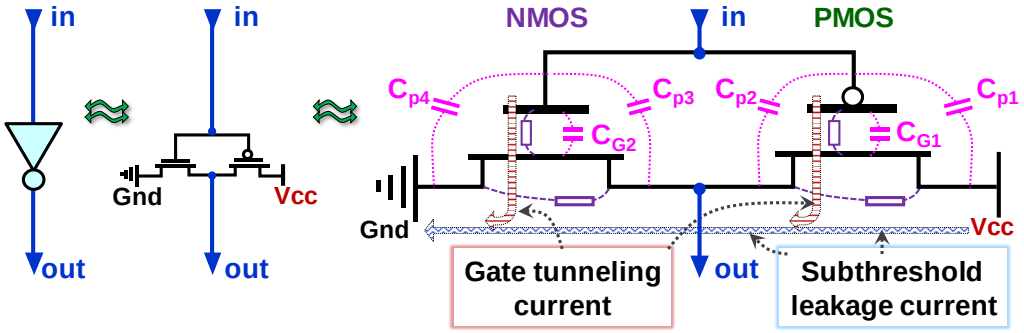


Figure 70 - Parasitic capacitances and currents in CMOS

The largest contributor for enhancement technologies above 180 nm is the leakage current between the S and D electrodes in CMOS closed state, aka the subthreshold leakage current. Below 180 nm, it isn't essential with the comparison to other phenomena. In our water model, it can be considered a gate leakage.

As nm decreases, the insulating layer under the G electrode thins to a few atoms thick. Its quantum tunneling, gate tunneling current, increases. We can imagine it roughly as water soaking from the inlet to the outlet through cracks in the walls. The current into electrode G increases to the dominant current draw as the gate nm decreases. In 7 nm technology, its contribution is reported to be as high as 80% of the total circuit power consumption, and microelectronics engineers are intensively looking for ways to reduce it.

CMOS transistors also have parasitic capacitances between their parts, as only thin layers sepa-

rate these. In our water model, the space between the gates and the next channel was filled during switching, which in CMOS corresponds to charging the parasitic capacitors. Its filling delays the passage of the signal. We discuss the phenomenon in more detail on p. 74.

4.8.3 Resistance model of two CMOS inverters

A more precise analysis of the processes during inverter switching would require a deeper delving into the structure of CMOS transistors and would exceed the scope of our textbook. Thus, we roughly approximate the switching of CMOS transistors only by capacitors that are charged through resistors, i.e., by analogues of RC cells, also known as integration cells. Their behavior can be described by the differential equation[20] .

If we are interested in the time it takes for the voltage of the capacitor V_C to rise to 50% of the voltage Vin, then solving it gives the time constant tp = ln(2) RC [s], which is commonly approximated by tp = 0.7 RC, since we rarely know the resistance and capacitance to better than 10% to 20% accuracy.

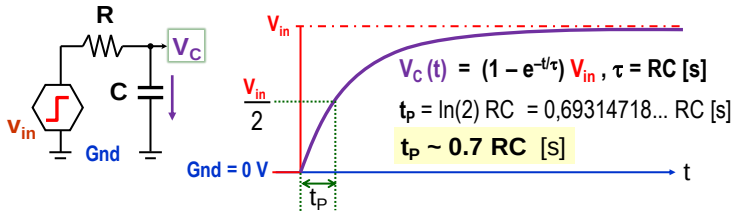


Figure 71 - RC article

We model the signal delay using the RC circuit through the left inverter. Capacitor C includes the sum of the parasitic capacitances at the left inverter's output, the connected wire, and the right inverter's input. Its value can be considered constant in the model.

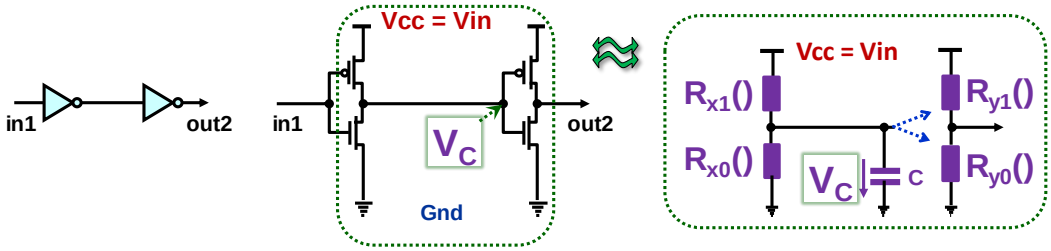


Figure 72 - Resistance model of two inverters

However, the resistor sizes vary significantly according to the applied voltages between the triple electrodes of CMOS transistors. They can be approximated at small values by voltage-controlled resistors, both linearly and nonlinearly dependent on the voltage. Saturation also occurs in which CMOS transistors appear more like current sources. We have indicated the variability of the resistors R_{x0} (), R_{y0} (), R_{x1} (), and R_{y1} () by brackets.

What are the values of the RC time constants?

NMOS and PMOS transistors manufactured as discrete components can have a resistance in the closed state of even less than 1 ohm. However, gates need larger values because of short circuit effects during their switching; see the water model Figure 65 and Figure 67.

The conductivity of CMOS depends on many parameters. One of them is the ratio of the width and length of the conduction channel under the electrode G. It is chosen so that a maximum current corresponding to a resistance of hundreds of ohms, even kiloohms, flows through the

[20] You can find the derivation of the equation here: https://www.electronics-tutorials.ws/rc/time-constant.html

transistors in the closed state. Fast circuit variants are designed with lower resistances, i.e., higher peak currents, to charge capacitances faster. Components designed for applications with low power consumption will favor higher spare resistances at CMOS switching, hence their lower short-circuit currents and surges.

As the size of CMOS technologies decreases, we have smaller areas of parasitic capacitors mentioned in Figure 70 on p. 68. Their capacitance decreases, reducing the time to charge or discharge them and power consumption.

The picture below shows the output of the left inverter; the right one will be similar. We omit the picosecond moments of short circuit current. They have a negligible effect on the delay. We choose to flip the right inverter at 50% Vcc, which occurs in 0.7 RC [s].

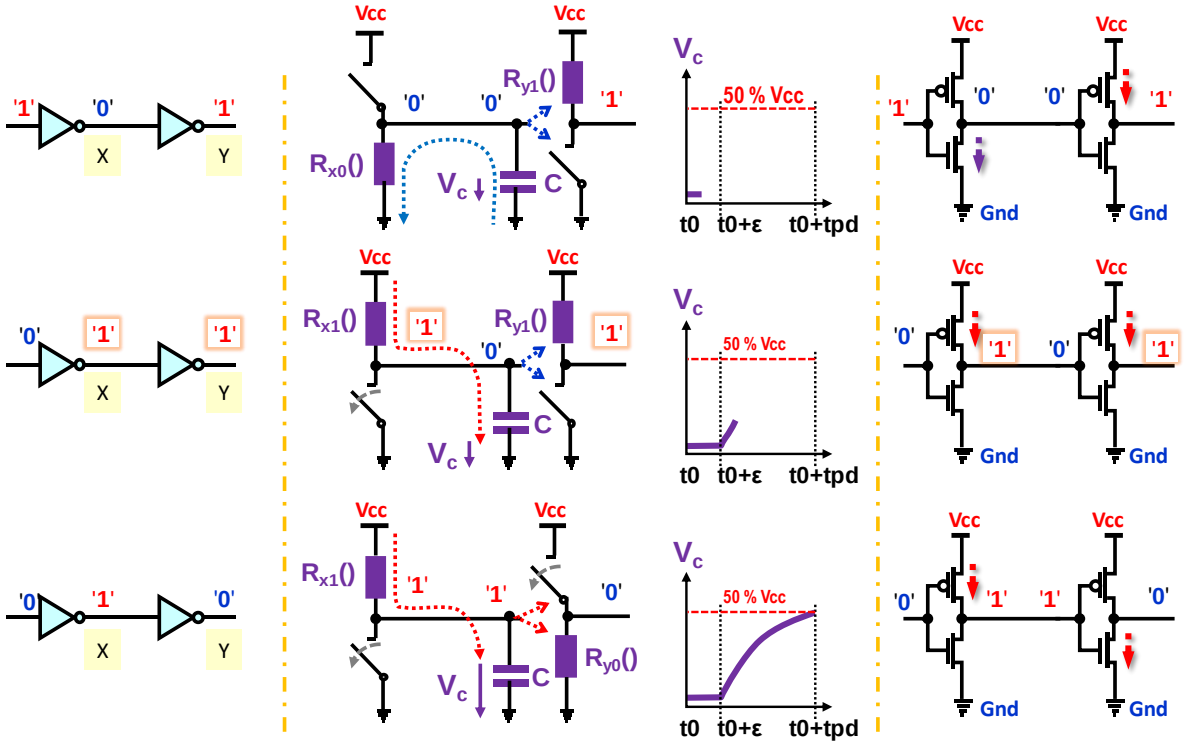


Figure 73 - Delay on inverter pair

- **time t0** - Let both inverters be steady in the initial state. The left one has '0' at its output X. Its upper PMOS is closed, and we will roughly consider it as a disconnected ideal switch. We replace the lower NMOS with a resistor R_{X0} (). Capacitor C is discharged, and its voltage V_C asymptotically approaches some lower value. **Current now flows towards the X output of the left inverter**. Output Y of the right inverter will be at '1'. We approximate its closed upper PMOS by a resistor R_{Y1} () and lower NMOS by an open switch.
- **time t0+ε** - The left gate has gone from '0' to '1', and the resistance R_{X1} () of the closed upper PMOS charges the capacitor C to a voltage V_{CC}, which is still below the decision level of 50% Vcc. It hasn't switched the right inverter yet, so **both have '1' values** at their outputs. The **current now flows in the direction out of its X output**.
- **time greater than or equal to t0+tpd**, where tpd denotes the propagation delay. The voltage V_{CC} has already exceeded the decision level of 50% Vcc. The right inverter has flipped. Its upper PMOS has opened, and its lower NMOS, which we model with the resistor R_{y0} (), has closed.

The tpd delay has properties in all types of gates:

- tpd depends linearly on the time constant of the RC circuit;
- The tpd decreases as the supply voltage increases since the currents flowing through CMOS increase with the voltages between their electrodes. Thus, the resistance of the closed CMOS decreases;
  - tpd varies with temperature, where several different factors work against each other. For small nm technologies, it may even decrease with increasing temperature, and for larger nm technologies, it is often longer;
- tpd is different when switching to '1' or '0'. The two CMOS groups are not entirely symmetrical because NMOS transistors are placed in the lower group, forming a conduction channel based on free electrons, which have three times higher mobility than the holes in PMOS. *Note: The mobility depends on the carrier velocity in the semiconductor. If it is higher, the current and frequency characteristics are improved.*

In practice, only the average delay is considered. The figure below denoted it as tpd, propagation delay time. The catalogs give it for different temperatures inside the circuit, selected values from 0 °C to 125 °C, and the various allowable power supply Vcc.

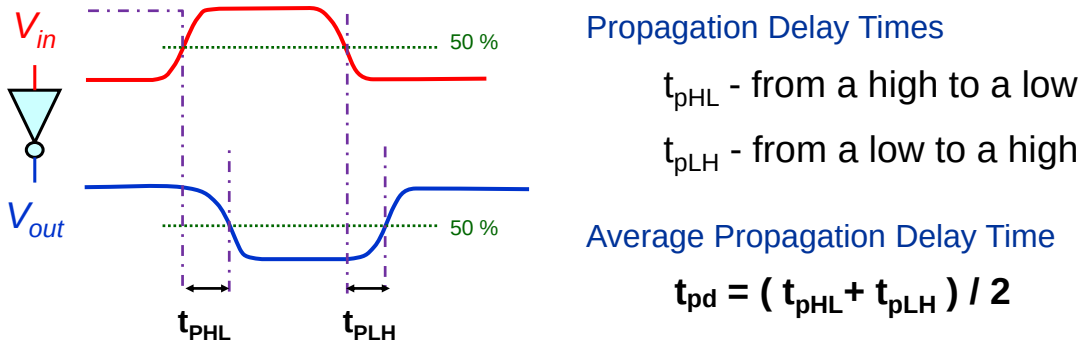


Figure 74 - Delay on inverter

We are still left with the question, what will be the inverter delay? We can only give rough indications, as the delay depends a lot on both the technology used and the geometry of the CMOS transistors, and is not constant! It is affected by temperature and power supply fluctuations. Multi-input gates are generally slower than an inverter because they tend to have multiple CMOS in series, where the voltage on each is spread out and drops, reducing the current through them. Capacitances are then slower to charge/discharge. The publications[21] mention inverter delays in the tens of picoseconds for 45 nm and above technologies. The smaller technologies can have it in units of picoseconds, even 2.5 ps for 7 nm CMOS.

Although the integration density increases with decreasing nm, the gate speeds become less improved due to other unfavorable phenomena. The 3 nm technology is expected to be slightly better[22] , about 2 ps for the simpler gate with no driving load.

What limits the working frequency? The available publications differ in their theoretical estimates of the maximum frequency at which semiconductor gates can operate in the appropriate choice of materials. The most common values are somewhere above 100 GHz, but iso-

[21] For example, Aaron Stillmaker, Bevan Baas, Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm, Integration, Volume 58, 2017, Pages 74-81, link.

[22] Etienne Sicard, Lionel Trojman: Introducing 3-nm Nano-Sheet FET technology in Microwind.2021. hal-03377556

lated studies claim that logic could operate at frequencies as high as over 1 THz.

The Core i9-13900K processor runs at up to 5.8 GHz and was the fastest commercially available type at the time of writing (2023). Most processors then remained at clock speeds up to 4 GHz.

Although some parts of the circuit can run at higher frequencies, such as serial bus drivers, any circuit is limited by overheating problems and power supply consumption. Already on the water model, we saw that the gate, when flipping, either requested a surge from the power supply or released a voltage wave at the ground connection. Contemporary technologies cannot yet deliver enough power to all gates at higher frequencies and dissipate the peeks from the ground connections.

Parallelization of operations offers a much more affordable solution to accelerate computing power today. Processor cores are being added, and accelerators created by logic circuits are also being used.

4.9 Introduction of logical '0' and '1'

In the previous text, for the sake of simplicity, we have assumed the most common situation, namely positive voltage logic (realization of logic '1' by higher voltage and '0' by lower voltage). However, we have seen in the timing characteristics of the CMOS inverter that the transition from logic '1' to '0', and back again is not instantaneous, but the voltage changes gradually as the capacitances are charged. The figure on the next page shows its output waveform. Its output Vout never has a full voltage of Vcc or 0 V.

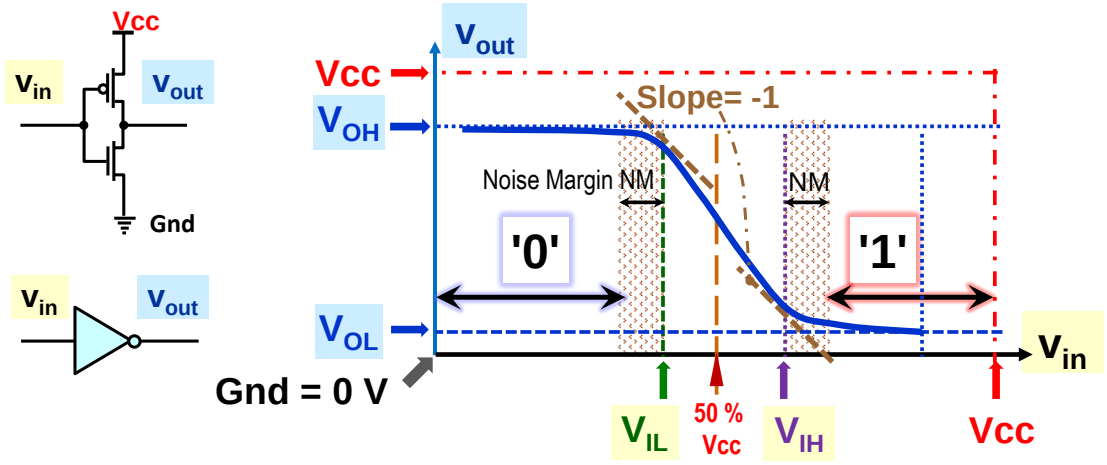


Figure 75 - Introduction of logical '0' and '1'

During V_{out} there are four essential values that manufacturers present in their catalogs.

- V_{OL} (output low) indicates the output voltage of the gate when it is at logic '0'.
- V_{IL} (input low) specifies the input voltage at which the output begins to change. In an inverter, the tangent of the output voltage waveform has a directive of -1.
- V_{IH} (input high) is a point similar to V_{IL} , but at the upper end of the V_{out} waveform.
- V_{OH} (output high) indicates the measured output voltage in logic '1'.

We insert our desired noise immunity NM, Noise Margin, into the waveform. In positive voltage logic, we declare as logical '1' any voltage values higher than V_{IH} +NM, the upper decision level, and we take as logical '0' anything lower than V_{IL} -NM, the lower decision level. We have been given the voltage ranges of logic '0' and '1'. Anything outside of these we will call an illegal or unwanted level. It will be there every time the gate is knocked, but only briefly.

How extensive are the ranges '0' and '1'? It depends first on the manufacturers' data concerning the supply voltage and then on our chosen NM noise immunity.

We give an example of data from a Cyclone IV E[23] family FPGA having two different power supplies. We selected two cases from all the allowed values. The most significant part of the circuit where the logic is formed, labeled FPGA core, has a voltage of 1.2 V. The external pins of the circuit case are routed through the gates of the LVTTL bipolar logic powered by a higher voltage of 3.3 V to facilitate the connection of downstream circuits.

Vcc	VOL [V]	VIL [V]	VIH [V]	VOH [V]
1.2 V (FPGA core)	0.3	0.42	0.78	0.9
3.3 V (In/Out Pins)	0.33	1.0	1.65	3.0

[23] Catalogue data: Altera: Cyclone IV Device handbook, page 1-12, I/O Standard Specifications, 2016.

Notice that the logic '1' range is greater than logic '0' for inputs and outputs. If the output is at '1' (i.e., at V_{OH}), it will have a higher noise immunity. For that reason, so-called **negative logic is used for** some mostly active signals only for brief moments, such as a circuit zeroing after power is turned on, which is no longer active.

In positive voltage logic, a higher voltage is represented by a logic '1' and a lower voltage by a logic '0'. The '1' and '0' ranges are swapped in negative logic.

In practice, we can choose various physical values for representing logical '0' and '1'. On serial buses, they are preferably created by a current, e.g., 20 mA, where the logic '0' will be -20 mA, i.e., current flowing in the opposite direction. Logic '0' and '1' can also be signal phase changes, for example, in Manchester coding or pulses in fiber optic cables.

The actual signal waveforms can be even more complex due to reflections on the lines and may go negative both above Vcc and below Gnd. Further fogging of the output will add the ubiquitous noise caused by crosstalk and peaks in from the power source. No zeros and ones are running in logic circuits, but complicated signal waveforms.

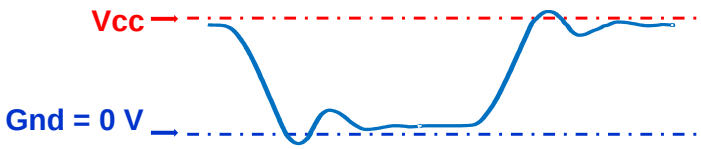


Figure 76 - Example of the actual output of a logic gate

How do we work with signals in logic designs? Simple. We take them as logical '0' and '1' and don't care about their exact physical realization or voltage. We consider actual values only in exceptional situations, e.g., when adapting the inputs and outputs of a circuit to its environment.

Logical '1' and '0' simplify the design — they reduce complex transients to abstract levels.

4.10 Effect of delays on signals

We will not now consider the actual voltage waveforms. We simplify our view to '0' and '1' states, i.e., situations where the V_{in} and V_{out} voltages are below or above the decision level, and draw a more straightforward graph of the gate delay. It belongs to an inertial delay category that adds a time delay of the output compared to the input and changes its waveform.

Shorter pulses are rejected because they do not manage to charge or discharge the parasitic capacitance. Thus, they do not pass through the gate, so the output is unchanged.

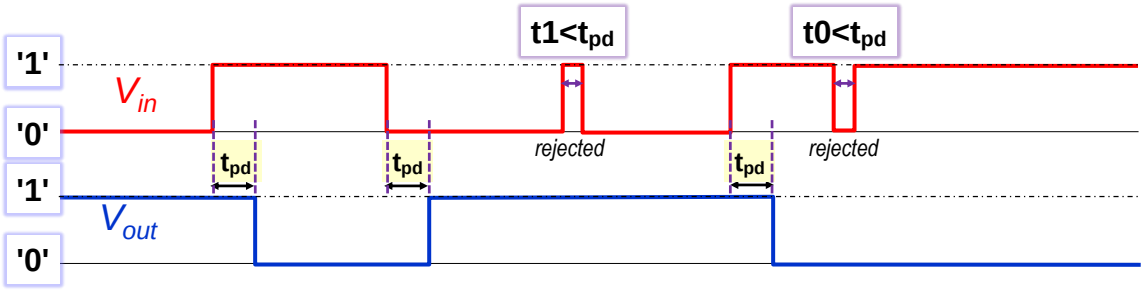


Figure 77 - Inertial delay at the gate

Another time delay category, which only shifts a signal in time without changing its waveform, is called transport delay (or wire delay). For example, ideal wires have it.

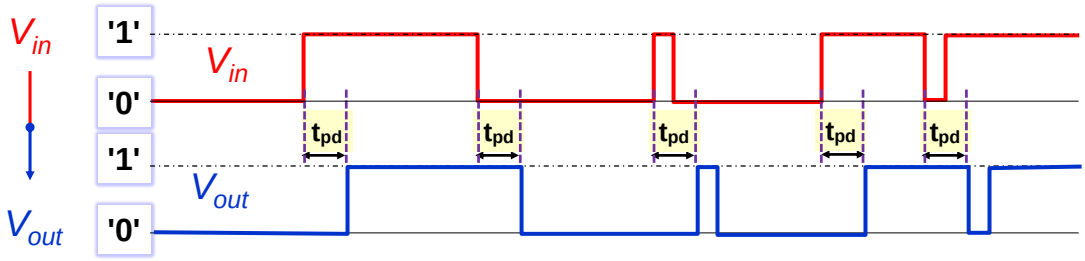


Figure 78 - Transport delay on ideal wire

When multiple inputs are connected to a gate output, their number is called **fan-out.** All parasitic capacities, presented in Figure 70 on p. 68, are added together.

If we have measured gate delay for a single input, it can be modeled by a relation in which the constant k_{inv} represents its total resistance:

$$t_{pd1} = k_{inv} (C + C)_{DG}$$

When we connect the inverter output into five inputs, the delay increases under the $C_D \approx C_G$ assumption to:

$$t_{pd5} = k_{inv} (C_D + 5*C_G) = 3 * t_{pd}$$

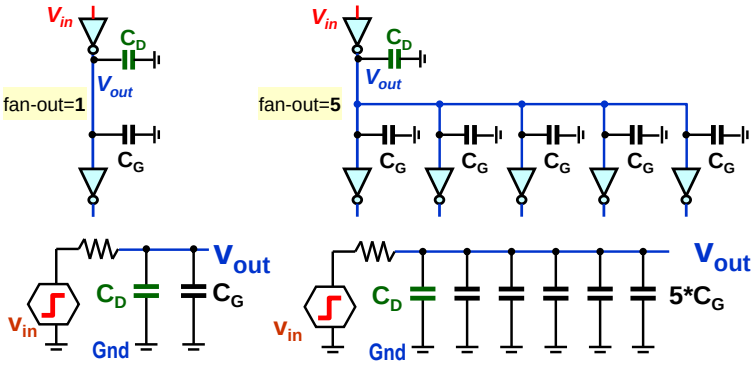


Figure 79 - Effect of input load on delay

Design environments will carefully monitor the fan-out count and insert separating elements, known to us as buffer elements, to reduce it if necessary. Sometimes, it is helpful to optimize the design so that the signal is not distributed over too many inputs. Each additional one increases the delay.

Notes:

- An interconnection of more outputs, i.e., the analogy of a short circuit, is also reported as exceeding the **fan-out**.
- The term **fan-in** indicates the number of inputs of the element. So an inverter has fan-in=1, while a four-input AND gate has fan-in=4. As mentioned before, gates with larger fan-in tend to be slower because their upper/lower CMOS group contains more transistors connected in series, which will reduce the output current and slow down the capacitance charging.

4.10.1 Hazards – transients in logic circuits

The propagation time delay of logic gates causes transient effects in circuits. Their output signals can have different time shifts in the circuitry, which sometimes cause temporary unwanted pulse glitches. If a logic function generates them, we say it has a hazard.

The term "hazard" comes etymologically from the Arabic word "*az-zahr*", a game of dice in which many had lost their fortunes. In logical circuits, we must take the existence of hazards into account; otherwise, our proposal may also be entirely wasted.

If hazards occur in a circuit, they do not always happen, but only at specific transitions ac-

cording to the internal structure of circuits.

A distinction is made between hazards:

- **static-0 —** in steady state '0' an unwanted pulse to '1' appears.
- **static-1 —** in steady state '1' an unwanted pulse to '0' occurs.
- **dynamic —** the transition from '0' to '1', or vice versa from '1' to '0', is not a smooth edge but a series of pulses.

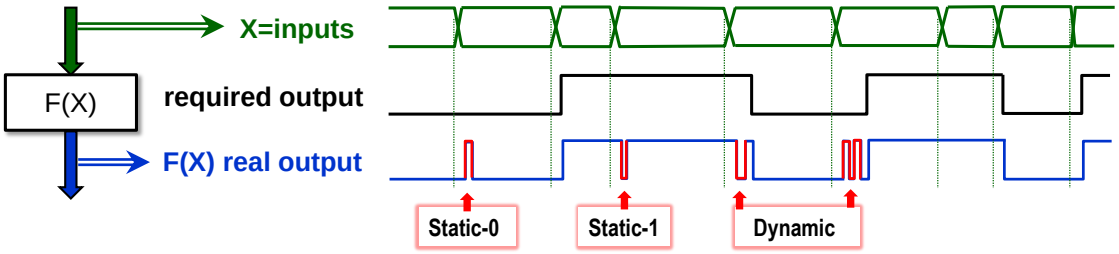


Figure 80 - Gambling

Let's consider hazards in the $F3_8$ function, which we created on p. 48 (Figure 41).

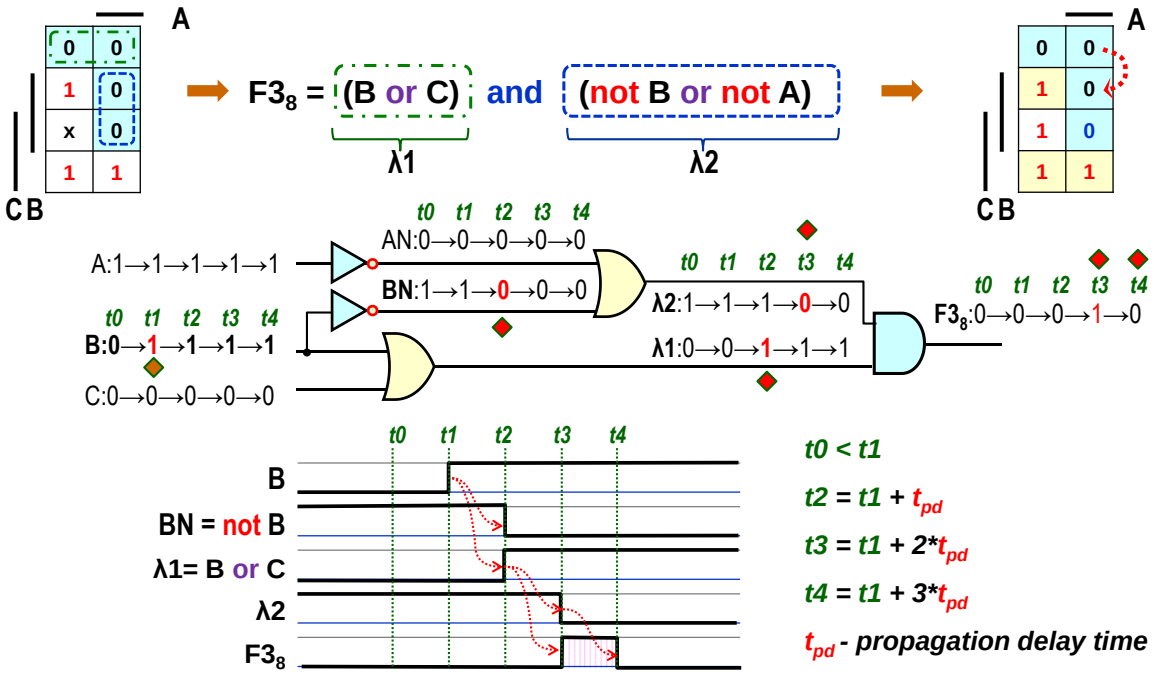


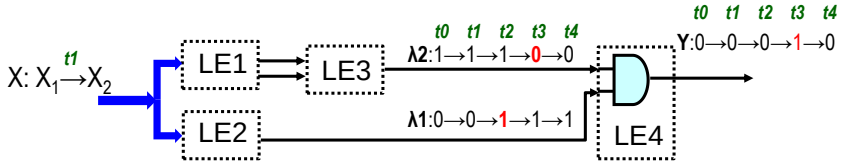
Figure 81 - Gambles in logic functions

At time t0, let the inputs $F3_8$ (A,B,C) have values A='1', B='0', and C=0. Its output is steady.

1) at time t1>t0, input B goes from logic '0' to '1'. Its change will propagate avalanche-like through the circuit due to t_{pd} gate delay;
2) only at time $t2=t1+t_{pd}$ affects the output of the BN inverter and λ1 of the lower OR-implicant so that the output AND-implicant $F3_8$ now has a logic '1' at both its inputs;
3) Thus, at time $t3=t1+2*t_{pd}$, not only the upper OR-bar λ2 is flipped, but also the AND-bar, changing the output of $F3_8$ to a logical '1', even though it should remain at '0' when A='1', B='1' and C=0.
4) at time $t4=t1+3*t_{pd}$, the output AND-implicant finally settles in the correct state '0'.

The glitch appears because the AND-lattice was at '0' at inputs A='1', B='0' and C=0 due to the implicant λ1, whereas it was held there by the implicant λ2 at A='1', B='1' and C=0, which changes later.

In FPGAs, functions are formed by logic elements, but they also generate different paths. If we draw the analogy of the figure above, in which we replace gates with LEx blocks that implement some general logic functions. They can even be more complex, e.g., with multiplications and additions. An analogous situation can arise when we change the input X from X1 to X2 in case of a suitable internal structure.



Can hazards be removed by changing the wiring in the logical combination functions?

It isn't easy on conventional FPGAs. Their design environments can only try to reduce them by balancing the delay times of partial paths in the logic circuit, but they cannot eliminate them. Delays are not constant but vary with temperature, which may differ inside chip areas.

Hazards are also the main reason to avoid using latches in FPGAs, which we will discuss in Chapter 7.2 on page 7. 122.

Hazards can be eliminated in combinational circuits only if we build directly from gates. In addition, it is necessary to satisfy strict Fundamental-mode operation conditions, such as changing only one variable at a time, which can only be fulfilled in exceptional cases.

If values of several inputs change at close time instants, transients may occur before an output is stabilized, resulting in a glitch. These are so-called functional glitches, i.e., they are based on the mode in which we use the circuit, and they cannot be eliminated by simply changing the circuitry.

Hazards can be suppressed everywhere by using synchronous circuits to clock the circuit. We always consider that the output of any logic circuit will settle down after a specific time, the so-called Worst-case propagation delay. Design environments calculate it and determine the slowest propagation paths from the inputs to the output.

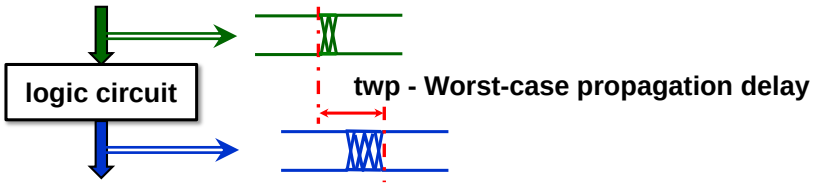


Figure 82 - Worst-case Propagation Delay

If a change is made to the inputs, we wait for them to subside, and after some time, slightly greater than twp, the outputs are sampled, and their values are stored. We obtain clean signals without hazards. Synchronous circuits will be discussed in a separate later chapter.

When do we care about hazards?

- We can completely ignore hazards if the result of the logic function is fed to much slower elements, such as the input of a 7-segment display. Their LEDs have a million times slower responses.
- We must worry about hazards mainly when we use synchronous circuits, the last part of the textbook. Their clock and asynchronous clear inputs respond to even short pulses, so they must not be connected to the output of a logic function that can generate glitches.

- Design environments sometimes add buffer gates to balance data paths, especially in synchronous circuits.
- **Inverter and buffer do not generate hazards**, as these are generated exclusively when multiple paths exist within a logic circuit. Thus, we can insert an inverter and buffer even in critical signals.

However, beware of the clock signal distribution! If we insert an inverter or buffer into their paths, the signal will not be distorted by hazards, but we create another unwelcome effect. Both gates will time-delay the clock signal, so some synchronous circuits flip on a little later than others, which is undesirable.

Because of this, we recommend keeping clock paths without inserting gates if they can be avoided. Of course, sometimes we have to add an inverter, for example, when changing the rising edge to a falling edge, or a buffer for decoupling, but with care.

In some systems, it is common to slow or shut clocks down for part of the circuit to save power, e.g., in processors, so-called clock gating. In this regular way, we reduce power consumption. If we think of the clock path as a tree that grows from the oscillator as a source, then significant savings can be achieved by making currently unused branches unavailable.

However, this is already a challenging solution that requires more complex synchronous logic for controlling the blocking or releasing clocks or slowing down their frequency. We must guarantee a suitable time of their change in which no disturbing pulses are generated.

5 Basic combination circuits

Combinational circuits can rarely be designed as solo. They are more frequently utilized as partial building blocks, so let's look at used elements.

5.1 Decoder 1 of N

We have already mentioned decoders 1 of N in the chapter 3.2 on p. 32. They have M address inputs and up to N outputs, where $N=2^M$, and they exist in two versions:

- **One Hot** - each of its N outputs takes '1' for only one input value.
- **One Cold** - each of its N outputs will be at '0' for only one input value.

	Inputs		*One Hot*				*One Cold*			
N	**x1**	**x0**	**F0**	**F1**	**F2**	**F3**	**G0**	**G1**	**G2**	**G3**
0	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'1'	'1'	'1'
1	'0'	'1'	'0'	'1'	'0'	'0'	'1'	'0'	'1'	'1'
2	'1'	'0'	'0'	'0'	'1'	'0'	'1'	'1'	'0'	'1'
3	'1'	'1'	'0'	'0'	'0'	'1'	'1'	'1'	'1'	'0'

Table 4 - Decoders 1 of 4

From the table above, we can see that the output functions of the decoders are minterms for One Hot and maxterms for One Cold. We can draw their Karnaugh maps, or even without them, we can write logic equations directly, from which we can then draw a schematic.

The equations of the Gx functions of the One Cold decoder can also be derived from the One Hot relations using De Morgan's rule since they are negations of Fx, e.g.,

$$G0 = \text{not } F0 = \text{not } (\text{not } x0 \text{ and not } x1) = x0 \text{ or } x1.$$

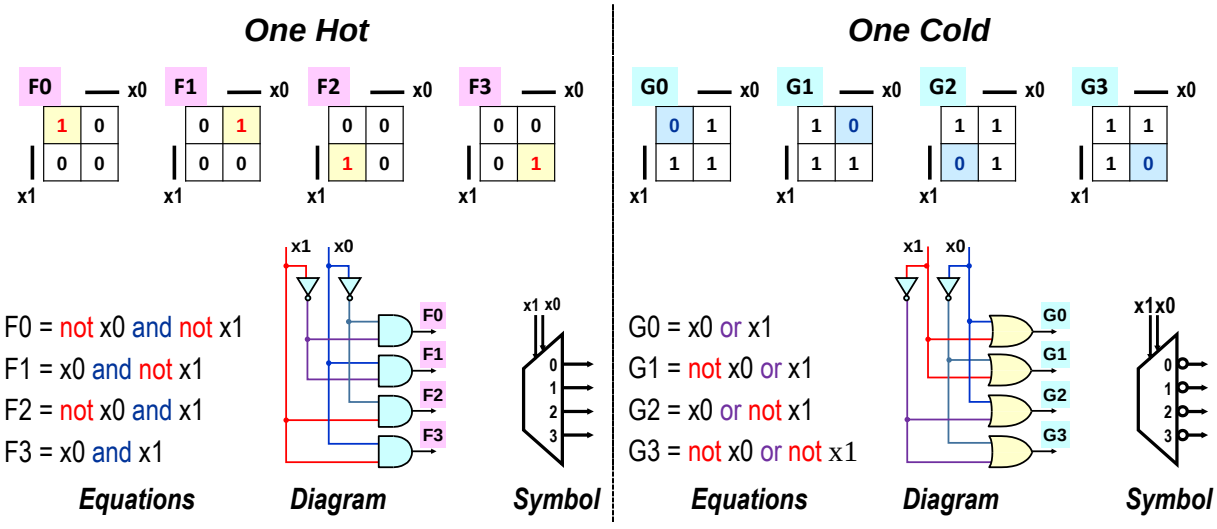


Figure 83 - Decoders 1 of 4

The figure above also shows the symbols used to identify the decoders in the schematics. The One Cold decoder branding differs only by the addition of inverter bubbles. The outputs do not need to be always connected, so a 1 of N decoder can have a shorter length than $N=2^M$, thus fewer output functions.

The decoder's output functions convert the input into a binary unsigned number to an encoding called 1 of N. For example, if we have values in the range 0 to 9, we store them as 10-bit

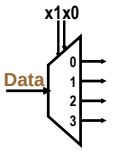
vectors with only one bit in the '1'. Its index indicates the value.

The 1 of N encoding is preferably used, for example, in finite state machines to encode their states. Although this representation has a longer bit length, it leads to simpler functions that detect whether the desired state has been reached. We only need to test one Fx or Gx.

5.2 Demultiplexer

We use decoder 1 from N to switch the data input to the output according to the chosen address. In that case, we get a demultiplexer, commonly abbreviated to Demux. It has little use as a standalone element but more as a building block for other circuits. We create it if we use the logic equations, see Figure 83, and add one term to each function Fx or Gx. For One hot, we add "and Data", while for One cold, we add "or not Data", since its outputs are negated.

D0 = not x0 and not x1 and Data
D1 = x0 and not x1 and Data
D2 = not x0 and x1 and Data
D3 = x0 and x1 and Data



DN0 = x0 or x1 or not Data
DN1 = not x0 or x1 or not Data
DN2 = x0 or not x1 or not Data
DN3 = not x0 or not x1 or not Data

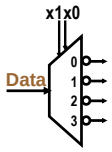


Figure 84 - Demultiplexor or Demux 1:4

Demux 1:4 distributes its single data input to four different outputs. It can be assembled directly from the 1 of N decoder or the gates. Let's demonstrate the procedure with the example of One hot 1 of 4. In the figure below, all four schemes implement an equivalent function. The conversion from the second scheme from the left to the third was done based on the associative law, see page 16.

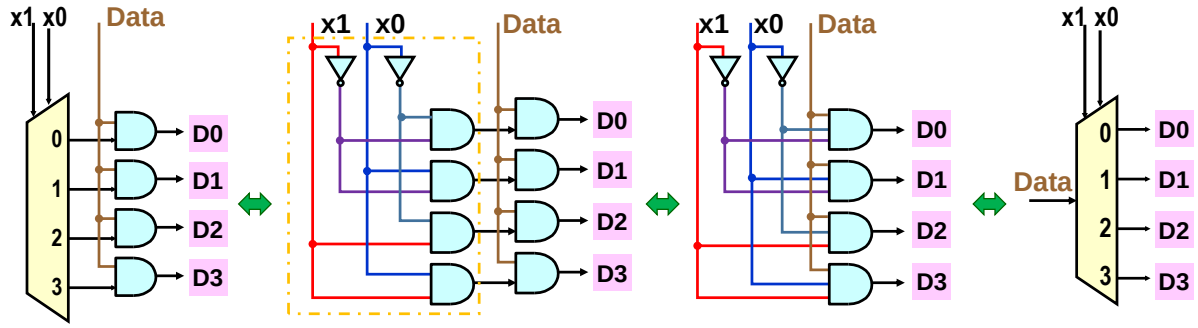


Figure 85 - Composition of a 1:4 demultiplexer from a 1 in 4 decoder

For example, Demux can create a flashing snake out of LEDs. Suppose we have a three-bit binary counter. We'll show how to build it in 7.5 on page 7. 138. We will connect its outputs to our Demux 1:4. We'll drive the output of counter Q0, with the lowest weight, to the Data input, connect Q1 to address x0, and the highest bit of Q2 to x1.

Small-technology CMOS circuits have low operating voltages and currents and might not fully light LEDs, which typically require around 20 mA and 1 V to 4 V for maximum brightness, depending on their size and emitted color. They need higher voltages, and we'll label this V_{LED} , which will be 9 V in our example.

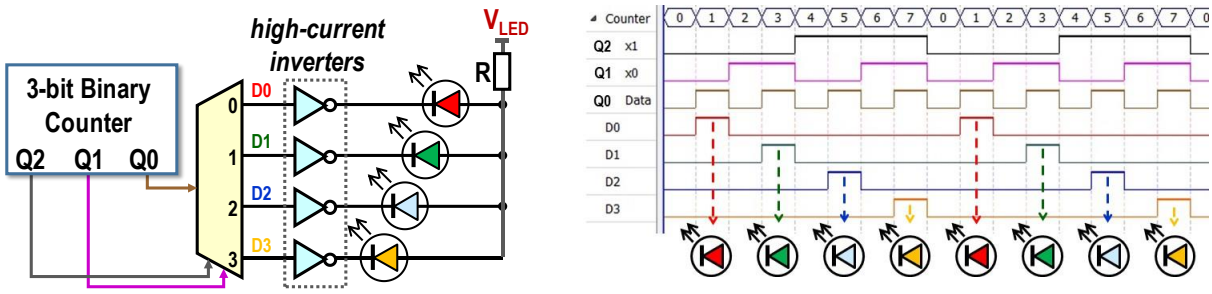


Figure 86 - Using Demux 1:4 on a flashing light snake

After the Demux outputs, we connect decoupling inverters for level conversion. These are manufactured as separate components. They will switch our LEDs powered by V_{LED} = 9 V. The resulting circuit produces the effect of a flashing light that cycles.

We can also use a Demux with multiple outputs and more bit counter, which would give us a longer flashing snake.

5.2.1 Group minimization and Demux 1:16

It is not handy to plug Demux 1:16 directly from its equations, as they would lead to 16 minterms like D0 = not x0 and not x1 and not x2 and not x3 and Data. To these must be added four address inverters and a buffer on the Data input distributed over 16 gates to reduce the *fan-in* of the circuit. Five-input AND gates will also be slower.

Let's try another solution. Let's build a 1:16 Demux from 5 1:4 Demux circuits.

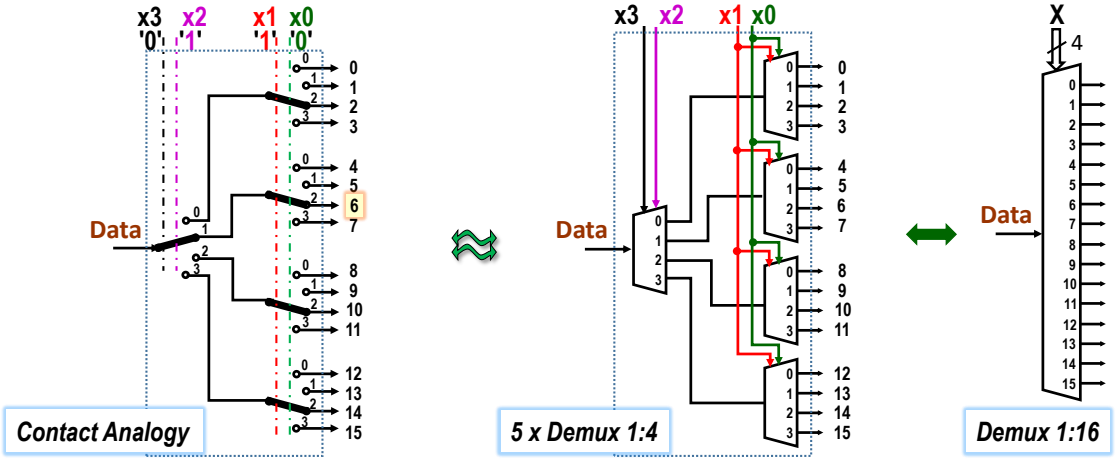


Figure 87 - Demux 1:16 of 5 Demux 1:4

Each Demux 1:4 contains 4 ANDs with 3 inputs and 2 inverters (1 input). We can reduce the number of gates even further if we share decoded addresses, see the figure below.

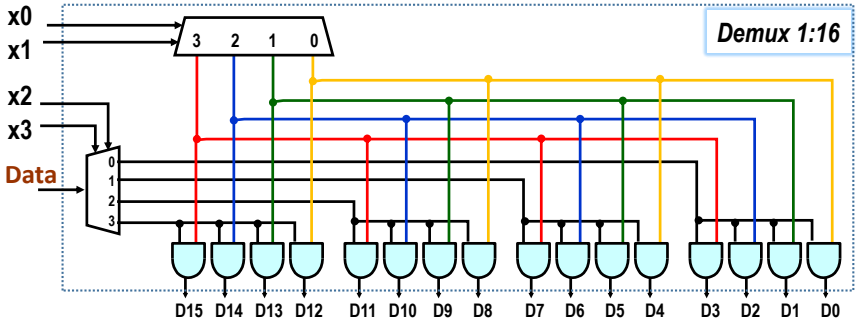


Figure 88 - Optimised Demux 1:16

The Data input of the 1:4 demultiplexer is sent to a single quadruple of two-input AND gates, which select the top two bits of the X address. The other three quad gates are latched with logic '0'. Decoder 1 of 4 simultaneously releases, according to the two lower bits of the address, i.e., only one AND gate of the activated quad, and deactivates the others with logic '0'. The output delay for our Demux 1:16 has increased by one AND gate compared to the Demux 1:4.

This procedure is called **group minimization**. In it, we do not optimize over a single function, but consider the minimum solution of the whole.

How is group minimization handled in circuit design? If we are working in a design environment, we can consider it, but usually, we don't care too much about it at first and leave it entirely to computer algorithms. We only specify what we want from our circuit description, like the Demux 1:16 mentioned earlier.

When everything is working, we improve in the second design phase. We can try a different description if we don't like some part of the solution, for example, because of its response or element consumption. Thus, we force the design environment to change implementation, for instance, in the style shown in Figure 88.

However, we should not be surprised when we view the resulting wiring automatically implemented by the design environment. We can find a differently decomposed 1:16 Demux, which may not look like the previous one, optimized for the CMOS used.

For example, if the Demux is implemented from other components, such as FPGA logic elements, it may use two 1:8 Demuxes and one 1:2 Demux.

Alternatively, it can be created by a 1 in 16 decoder whose outputs activate one of the sixteen AND gates, which primarily minimizes the delay from Data to Dx outputs.

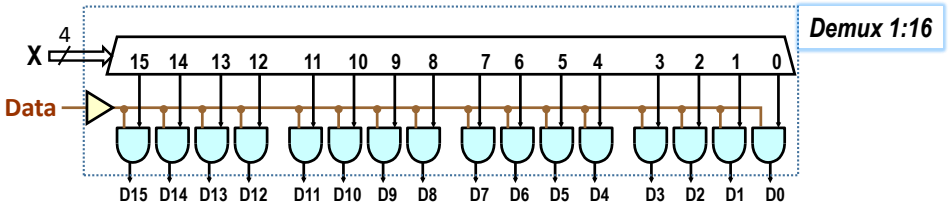


Figure 89 - Another Demux 1:16 solution using the 1 in 16 decoder

The second input of the AND gate has the Data diverged from the output of a powerful buffer capable of handling a fan-out of 16. Alternatively, you can use the Data diverging from several parallel buffer elements, for example, to drive only the inputs of two quads of gates.

The example of group minimization demonstrated that there is not necessarily a single optimal solution in circuits. Even the simple 1:16 Demux could be implemented in several ways. Each of them brought different advantages.

5.3 Multiplexor

The multiplexor works inversely to the demultiplexer. It has 2^M data inputs, where M is the number of bits of the input address. According to its value, the input sends Di to the output. Its name is commonly abbreviated to Mux, alternatively, the term "data selector" is also used, as it works analogously to a rotary multi-position switch.

A 4:1 multiplexer can be built from a 1 in 4 decoder and gates or directly from its equations:

```
Y =     ( not x0 and not x1 and D0 ) or ( x0 and not x1 and D1 )
     or  ( not x0 and x1 and D2 ) or ( x0 and x1 and D3 )
```

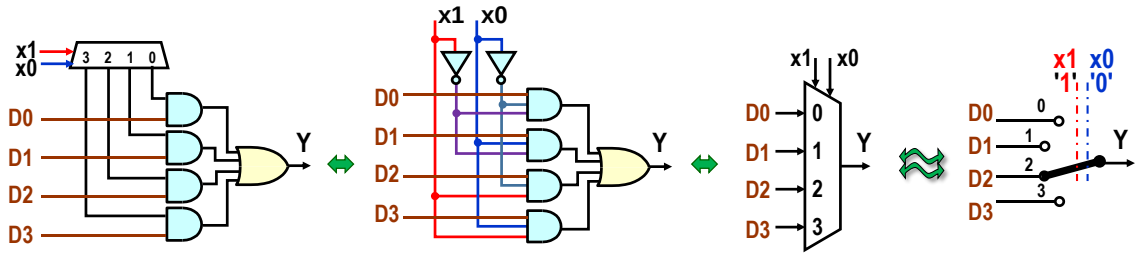


Figure 90 - Multiplexor 4:1

For the Demux element, or decoder 1 of N, we could have up to 2^M outputs. We need not connect each output; however, the inputs must always be defined.

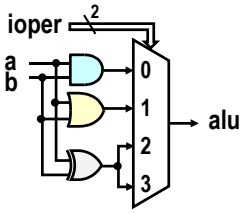
The N:1 multiplexer must know the values of all its N=2^M inputs. If an application doesn't need that many, it must still specify all the remaining ones, just wired to logical '0' or '1' so that each address from **0** to **2^M** -1 assigns an output value.

What is interpreted by the N:1 multiplexer in hardware?

For example, switch operations similar to the C **switch**() statement are converted to it, but the circuit implementation requires that there is always a default value.

C++ language	Hardware

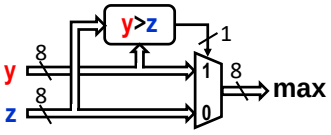
```
bool alu(int ioper, bool a, bool b)
{    switch(ioper)
     { case 0: return a && b;
  case 1: return a || b;
  default: return a ^ b;
     }
}
```



Switching on multiplexors also implements conditional **if-then-else** statements or conditional assignments, leading to 2:1 multiplexors.

C++ language	Hardware

```
byte max(byte y, byte z)
{    return y>z ? y : z;
 // if(z>y) return z; else return y;
}
```



The previous circuit uses a comparator; we will discuss it in Chapter 6.1 on p. 97. The result of its comparison drives eight simple 2:1 multiplexers, which can be considered analogous to the switch, as discussed on page 5. 53.

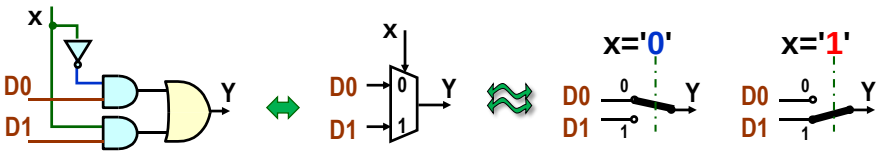


Figure 91 - Multiplexor 2:1 as a switch

When switching buses, one bit is used for each bit, i.e., as many wires as we need. The connection of the input x of the address to each of the eight multiplexers is more economically expressed with the signal x routed through them, the style most often used to draw networks

of multiplexers or other elements.

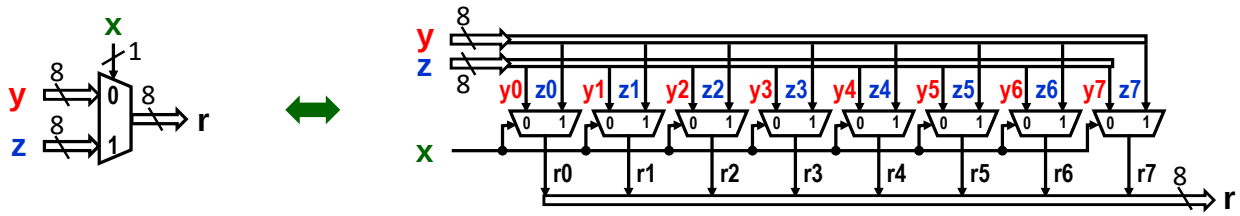


Figure 92 - 2:1 8-bit bus multiplexer

Mux 2:1 connects one of its two inputs to output r. It performs the analogy of selecting an element from an array, here with two elements, according to the index given by x input.

Example: The expression with multiple xor was discussed in chapter 2.3.1 on p. 23 , in which we proved by mathematical induction that it returns '1' for an odd number of input bits. For three inputs, it has the equation:

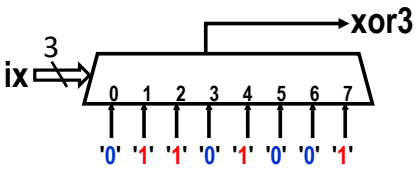
xor3(ix2, ix1, ix0)=ix2 xor ix1 xor ix

Alternatively, it can be described by a list of minterms, see chapter 3.2 on p. 32, as shortcut of its truth table. We then describe xor3 by just the list of indices where there are an odd number of input bits: xor(ix2, ix1, ix0): m(1,2,4,7)

We can also implement the function by a multiplexer with an address X of length 3 bits, which selects an element from an array of size $2^3 = 8$ elements, in which we store '1' at the given indices, and elsewhere they will be '0'. Multiplexers allow any combinational logic function to be expressed in this way[24] .

C++ language	Hardware

```
bool xor3(int ix)
{ // xor(ix2, ix1, ix0): m(1,2,4,7)
    bool array[8] = { 0, 1, 1, 0, 1, 0, 0, 1 };
    return array[ix & 0x7];
}
```



Multi-input multiplexers can preferably be assembled from smaller ones. For example, if we need a 16:1 multiplexer, then it can be made from, for example, fifteen Mux 2:1, or two Mux 8:1 and one Mux 2:1, or five Mux 4:1, as shown in the figure below, with a switch analogy indicating the state of the selectors at address x="x3 x2 x1 x0"="1001 converted as unsigned to 9.

[24] Recall that the Shannon expansion performs a decomposition of the logic function into two cofactors, i.e., functions on the inputs of a 2:1 multiplexer, see Figure 46 - Shannon expansion on p. 42. And cofactors can be further decomposed into smaller ones.

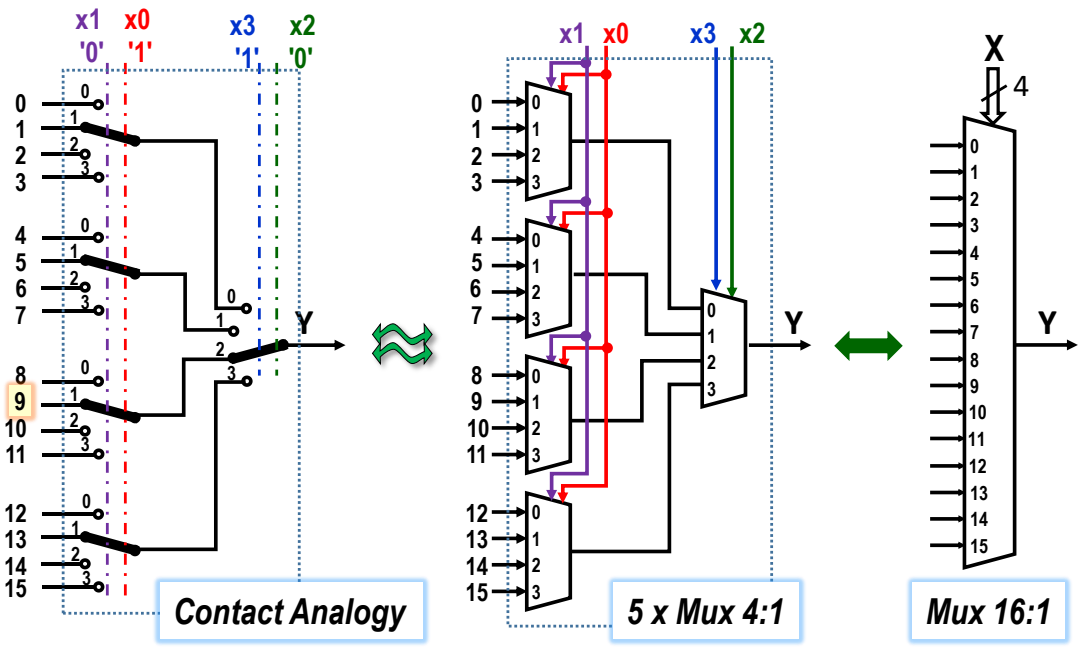


Figure 93 - Multiplexor 16:1

Note the **order of the address bits**. The left multiplexer row uses the lower X address bits because it has less priority in value selection than the 4:1 output Mux. The second row lies after it and so must receive the upper bits of the address. This way leads to numbering the inputs in a continuous series.

If the left row of Mux 4:1 has addresses x3 and x2 bit groups and the second x1 and x0, we create an erroneous conversion of $x3*2^1+x2*2^0+x1*2^3+x0*2^2$ binary number interpreted as unsigned. Changing X from 0 to 15 would send the inputs to the Y output in order:

0, 4, 8, 12, 1, 5, 9, 13, 2, 6,1 0, 14, 3, 7, 11,15.

5.4 FPGA LUT tables

In FPGAs, the combinational logic is implemented using LUTs. Their input X selects the value of the logic function from its truth table. During the FPGA configuration, the logic function's values are stored in memory cells. Then, they behave as constants and hold their values until the entire FPGA is changed to a new circuit.

This selection logic is called **LUT, Lookup** Table[25] . In circuits, it can be elegantly solved by a multiplexer. The figure below shows a type of LUT with four inputs, for which the name 4-LUT is often introduced:

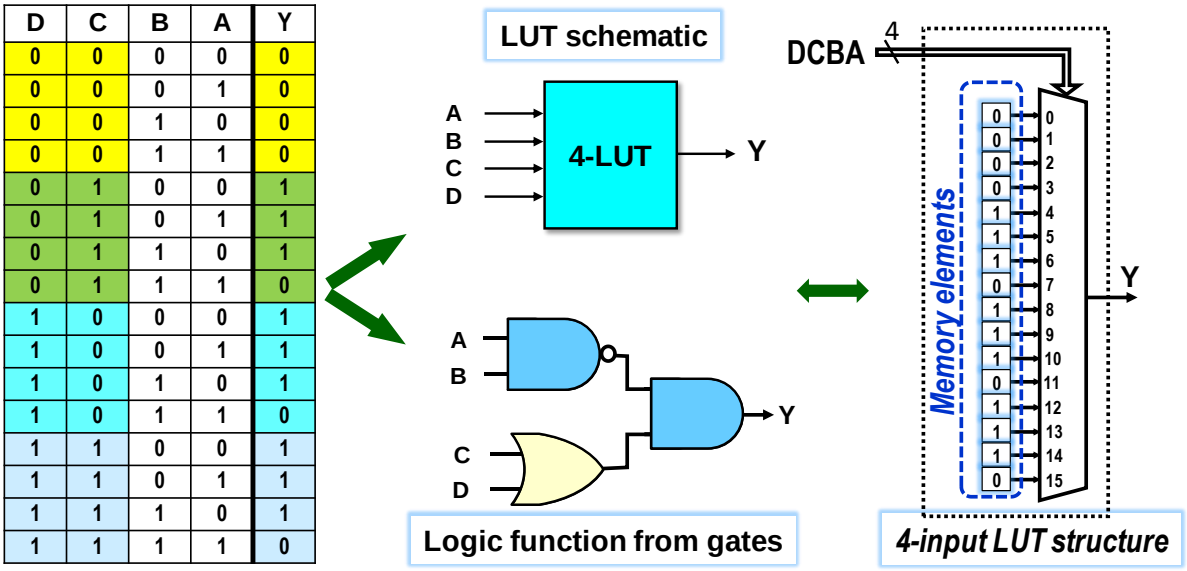


Figure 94 - 4-LUT - 4-input LUT

LUTs can have different internal structures. They can also be built from Mux 2:1. The figure shows its switch analogy with the path highlighted in the case of DCBA="1001" inputs.

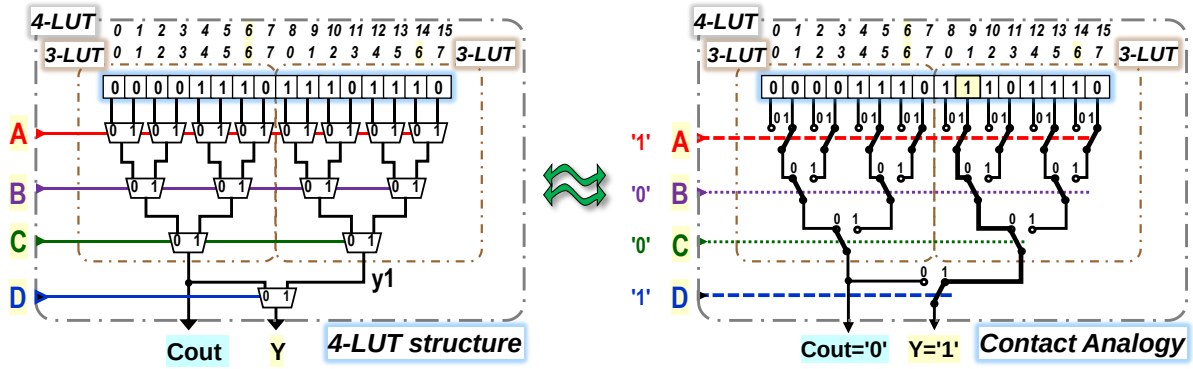


Figure 95 - Possible 4-LUT solution

Each 4-LUT input has a different propagation delay. The fastest change in the value of input D is seen at output Y, when only one of the two higher-order outputs controlled by input C is selected from the four B-series outputs. The signal path from input C will lead to output Y through two rows of multiplexers. From input B it will extend to three.

[25] From a mathematical point of view, the LUT performs the restriction of the representation to a limited definitional domain. It is implemented by a data structure that replaces the calculation by finding a value. LUTs are used not only in FPGAs, but also in classical programming to quantify complex functions where possible intermediate values can be determined by interpolation.

Input A has the longest delay, with four multiplexers. All eight upper Mux 2:1 are switched at once when it is changed. The eight values selected by them from the 16 memory members are then propagated through the lower rows, whose momentary states determine which will make it through to output Y.

Faster LUT inputs are preferred if the implemented logic function has fewer inputs. For example, a two-input XOR is described by the truth table of the logic function Y = C xor D. The unused slower LUT inputs, here A and B, are connected, for example, to '1'.

However, the XOR delay will not be half that of the four-input logic function, only shorter by two multiplexers. The connections to the LUT go through additional members each time, whose delay is also added. If we have a circuit composed of multiple two-input functions, then even a tiny speedup of each will positively affect the overall time delay.

In the internal structure of the FPGA, the Mux 2:1 is connected from the transmission gates discussed in Chapter 4.5 on p. 63. The time delay between the inputs D0 and D1 of the multiplexer and its output depends only on the charging capacitances of connected wires, and will be negligible for short ones.

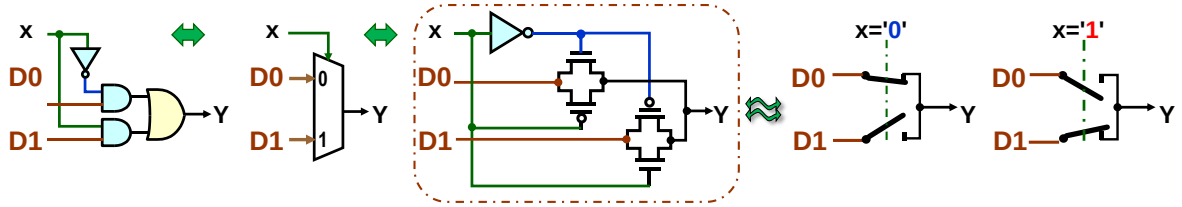


Figure 96 - MUX 2:1 from transmission gates

Mux 2:1 consists of two transmission gates and an inverter and is shared by other members with the same address in multiplexer networks, e.g., the whole eight controlled by input A.

There are also more economical implementations of LUT selection logic, for example, based on current sources[26] , in which simple NMOS transistors replace transmission gates.

However, the total consumption of a CMOS transistor per LUT is mainly determined by its memory cells. Each of them also needs logic to adjust itself during the programming of the FPGA circuit to the new circuit. Depending on its specific design, if memory cells are based on CMOS RAMs, each cell needs 8 to 12 transistors.

In many circuits as adders or comparators, fast carry propagation is needed. Each sub-stage handles two input bits plus and low-order carry and generates both the result bit and its carry destined for the higher order. The results of the whole chain will only be definite after all the transfers have settled, which determines the speed of the entire component.

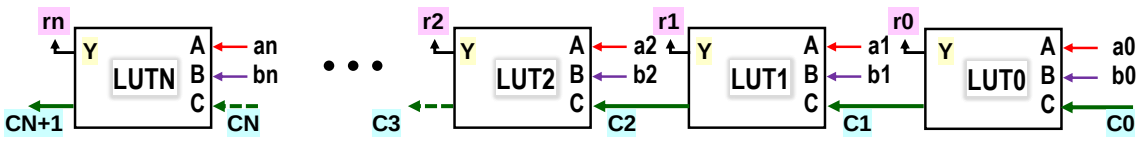


Figure 97 - Transmission propagation

The LUT can be configured to speed up transfers. In the following figure, a 4-LUT is decomposed into two 3-LUTs. In a primary 4-LUT configuration, the D input switches between them. If D is internally connected to '1', output Cout, Carry Out, is run out from the lower 3-LUT. Between the Cin input and the Cout output now lies the delay of a single 2-input multiplexer, a

[26] Suzuki, Daisuke et al. "Area-efficient LUT circuit design based on asymmetry of MTJ's current switching for a nonvolatile FPGA." *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)* (2012): 334-337.

relatively fast element. The two 3-LUTs share the A and B inputs as their working and C as their carry.

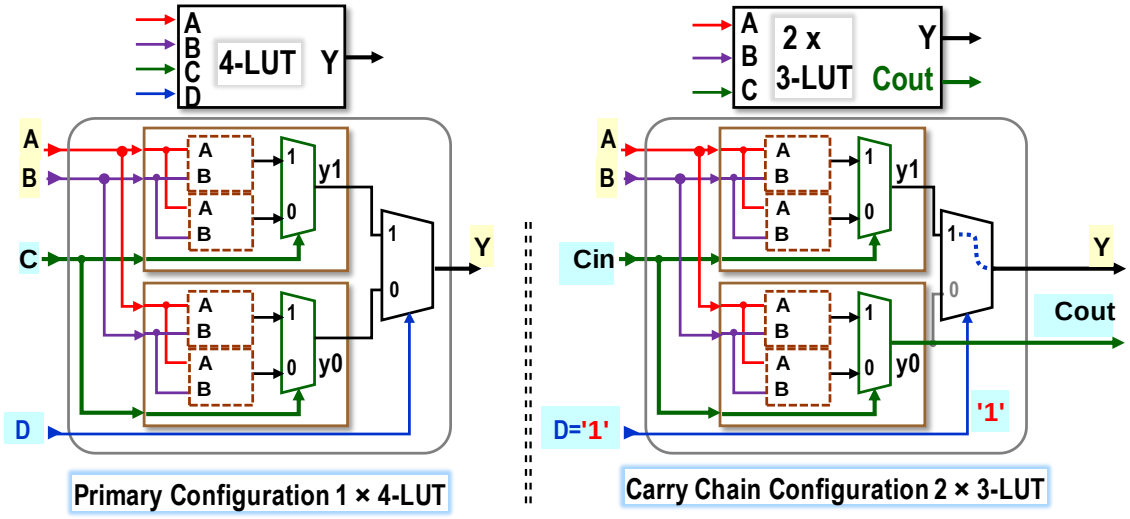


Figure 98 - 4-LUT configuration for accelerated transmission propagation

5.5 The internal structure of an FPGA circuit

What circuit did we connect? The question arises after we describe some wiring in the chosen design environment that optimizes our task and loads the result into our FPGA circuit. Then, we should always look at how everything was implemented, whether our description led to an acceptable internal wiring structure, or whether choosing a different technique for decomposing the problem into subparts will be necessary. Because of this, we also need to know a little about the internal structure of FPGA circuits.

We can find many internal FPGA layouts in manufacturers' catalogs according to their specific family and type.

For the demonstration, we chose an older 90 nm technology circuit FPGA Cyclone II type EP2C35F672, which has fewer elements than the FPGA Cyclone IV E used in our newer development boards (Figure 2 on p. 9), but contains all of its building blocks.

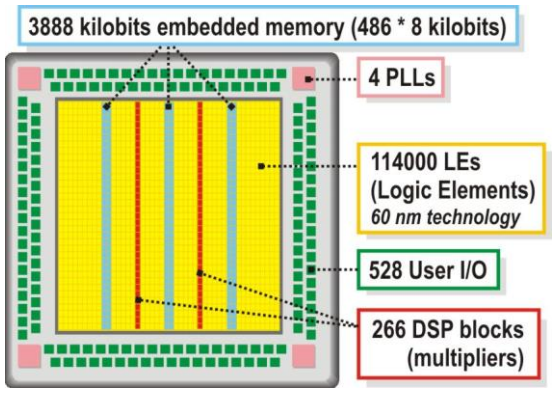


Figure 99 - Intel Cyclone II FPGA

Other FPGAs use similar elements, so the description applies to them as well.

5.5.1 User I/O Pins

FPGA housings have hundreds of pins, most of which are user's pins, to which we can connect our external physical inputs or outputs, making PCB, Printed Circuit Board, layout easier. The FPGA in the picture above has 672 pins, 475 of which we can use according to our needs and circuit structure.

If we buy a development board, the FPGA is soldered to its PCB, and the positions of inputs and outputs are fixed. We just read their layout from a Pin Assignments list containing optional symbolic names of inputs and outputs. Thus, we write, for example, only CLOCK_50 instead of the exact index of the physical pin.

5.5.2 DSP blocks

The DSP, Digital Signal Processing, blocks embedded in FPGA circuits are manufactured to be versatile. Most commonly, they are designed to perform various multiplications, including floating-point operations. Other DSPs implement, for example, digital filters, or FFT, fast Fourier transform.

The Cyclone II FPGA includes 35 DSP blocks. All contain 18x18 bits of integer hardware multipliers. However, each can be configured to two independent 9x9-bit integer multipliers. We will look at the principle of hardware multipliers in chapter 6.3.6 on p. 112.

5.5.3 PLL - Phase Lock

The abbreviation PLL comes from a Phase-locked Loop and refers to a circuit that belongs to the standard equipment of more advanced chips, including FPGAs and other techniques. It has numerous applications, to pick just a few. For example, they reconstruct clocks from data transmitted over serial lines, called clock recovery, and there are also analog PLLs for frequency demodulation.

Our selected FPGA contains four digital PLLs that can multiply the input clock frequency by a fraction whose numerical value can be greater than 1. They can be independently configured to produce our desired frequencies derived from the base clock.

The piezoelectric crystals used in oscillators can only be made up to frequencies of tens of MHz. PLLs create higher frequencies. For example, during the overclocking of CPUs or graphics cards, we only change the fraction's value by which the crystal oscillator's frequency is multiplied. The principle of operation will be approached in the lectures of our LSP course.

5.5.4 Firmware

Most FPGA circuits include all the equipment needed to configure them[27] . Their firmware mostly consists of the JTAG serial bus interface[28] , an established industry standard for configuring circuits and monitoring their internal values. Development boards are commonly sold with converters between USB and JTAG. We only install the appropriate driver.

5.5.5 On-chip Memory

Memories placed directly in the FPGA are also called **embedded memory**. We utilize them, for example, for tabling goniometric functions or for buffers of received data such as FIFO, First In, and First Out. In FPGAs, memories are assembled from memory blocks of fixed kilobit size. The entire memory block is always used, even if we store a single bit into it.

Our selected FPGA contains 472 kilobits divided into 105 memory blocks called M4K. Each block stores 4 kilobits plus an additional 512 bits for internal parity. We can configure it into either a 4kb×1 memory, i.e., 4096 bits, or a 2kb×2 if we need a 2 bit output, or in other variants, such as a 512 byte memory with parity. Memories with larger sizes are constructed from multiple blocks. Their required configuration is specified through the manufacturer's development environment, so creating and using the memory is relatively easy. We will demonstrate it in the exercises of our LSP course.

---

[27] Only FPGA circuits based on antifuse memory elements require an external programming device due to the need for voltage pulses. Their configuration is already permanent, it cannot be changed.

[28] You can read about JTAG on Wikipedia or https://www.xjtag.com/about-jtag/what-is-jtag/

Internally, the memory blocks are based on SRAM memory types, i.e., Static RAM, Random Access Memory, in most other FPGAs. RAM contents can also be initialized during FPGA configuration and used as ROM, Read-Only Memory.

However, the SRAM principle requires that the input address be stored in a register, as it must remain constant for the time necessary to equip the data. However, they are fetched with variable delays, so it is recommended to add their output register as well. In the figure below, after changing the address to the value marked A3, the data appears at the output with a delay of almost two periods of T clocks.

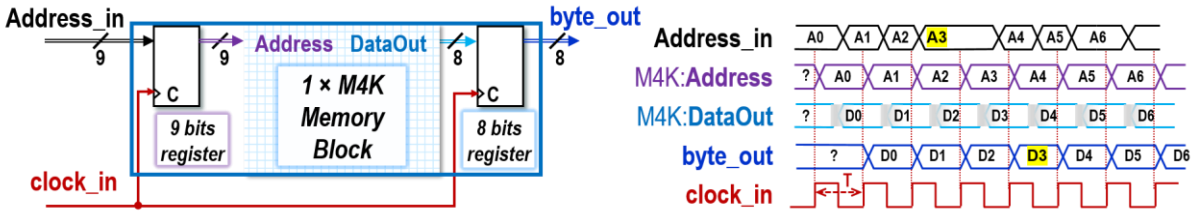


Figure 100 - M4K in 512 byte ROM configuration

SRAMs in FPGAs commonly have dual selection logic, *2-port SRAMs* that allow reading or writing data from/to two distinct addresses simultaneously with access controlled by different clocks. Dual, triple, or more SRAM select logic is used internally in many advanced chips.

5.5.6 Logical elements and jumpers

The logic elements, LEs, are the basic building blocks of all FPGA. They consist of at least one LUT with a synchronous flip-flop circuit and configuration logic. To zoom in on their structure, we use a magnifying glass to zoom in on the logic elements.

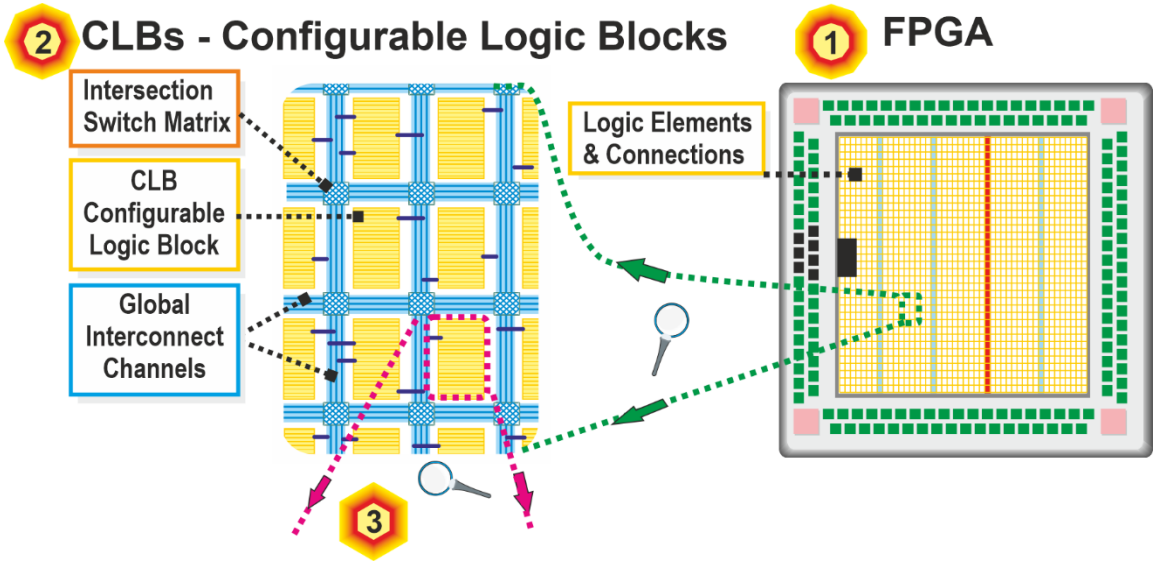


Figure 101 - FPGA structure: configurable logic blocks CLBs

Inside the FPGA, see [1], the logic elements are grouped into configurable logic blocks **CLBs**, cutout [2]. We will look at the structure of CLBs in the following cutout [3].

Configurable interconnections around the CLBs are the most critical components of all existing FPGA circuits. They are grouped in Interconnect Channels and have varied lengths, from short to long, sometimes with repeater signals. The synthesis tool of the development environment selects connections from these according to their need and availability. It determines their interconnection by Intersection Switch Matrixes, configurable matrixes of switches.

Cutout [3] shows the inner part of a single configurable logic block CBL. It contains 16 logic elements, LEs, which we will look at in the following cutout [4].

In one CLB, its logic blocks, LEs, can be connected using wires in the local connection channel. The inputs or outputs of an LE can also be fed to global wires if we need to fed a signal from/to another CLB.

There are also direct connections between LEs, but only to the physically following LE. These are used to implement fast transfers, Carry, or join LEs into multi-bit logic, e.g, data register.

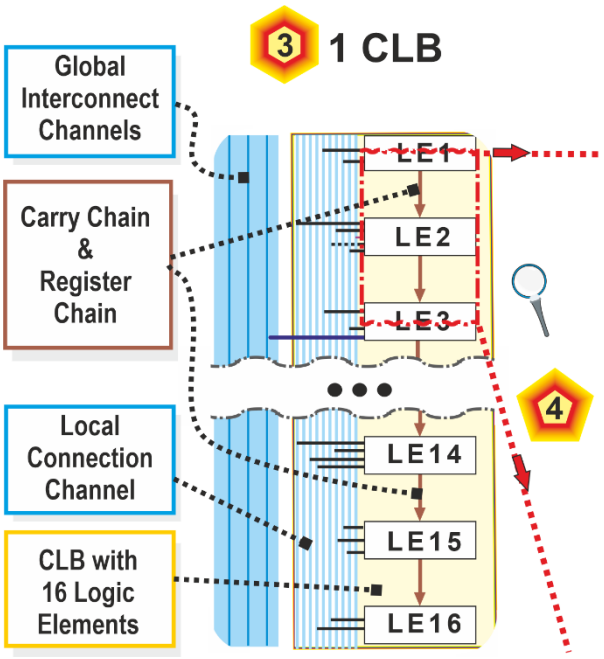


Figure 102 - FPGA structure: Configurable Logic Block CLB

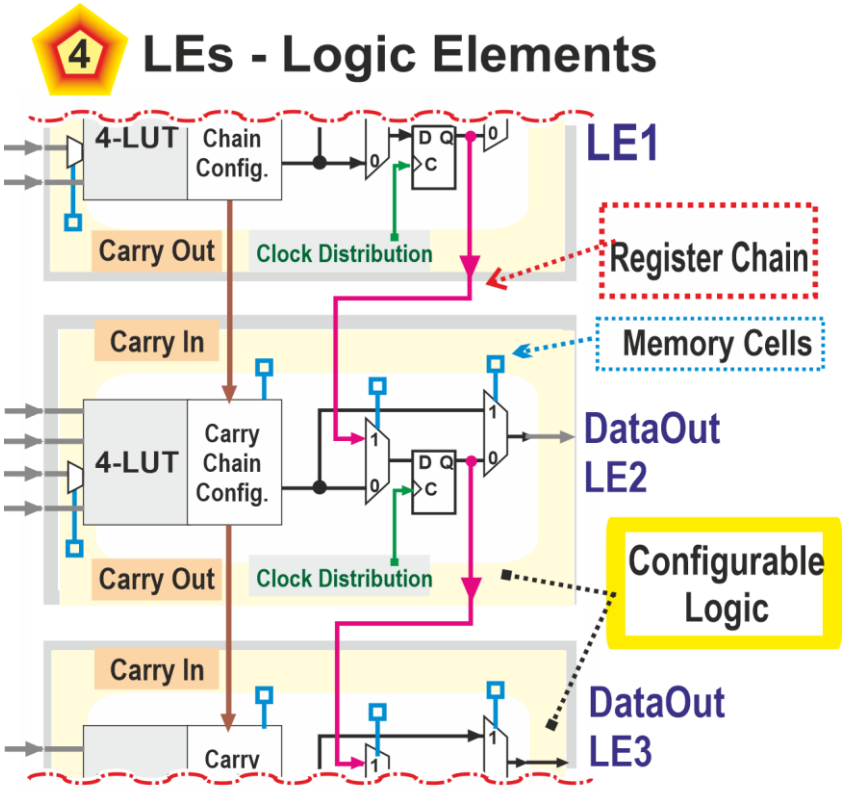


Figure 103 - Figure 100 - FPGA structure: LEs-logic elements

The last cutout [4] shows the direct connection of LE2 with the previous LE1 and the subsequent LE3. One of the direct links is the Carry Chain mentioned earlier; see also the following section 6.1. The other type, Register Chain, concatenates the flip-flop circuits within LEs. So, the output of one register leads to the input of the next, which is helpful, for instance, in shift registers. The interconnect is controlled by 2:1 multiplexers whose address input is connected to the Memory Cell with content set during FPGA is configured. Then, the memory cell holds its value until a new connection is loaded.

We devote an entire chapter to configuration memory cells 0 starting on p. 95.

The switch matrix, Intersection Switch Matrix, mentioned in the cutout [2], Figure 101, contains adjustable intersections. The figure below shows a simple disjoint matrix, which connects the signal by six transmission gates only in the diagonal of their crossing[29] .

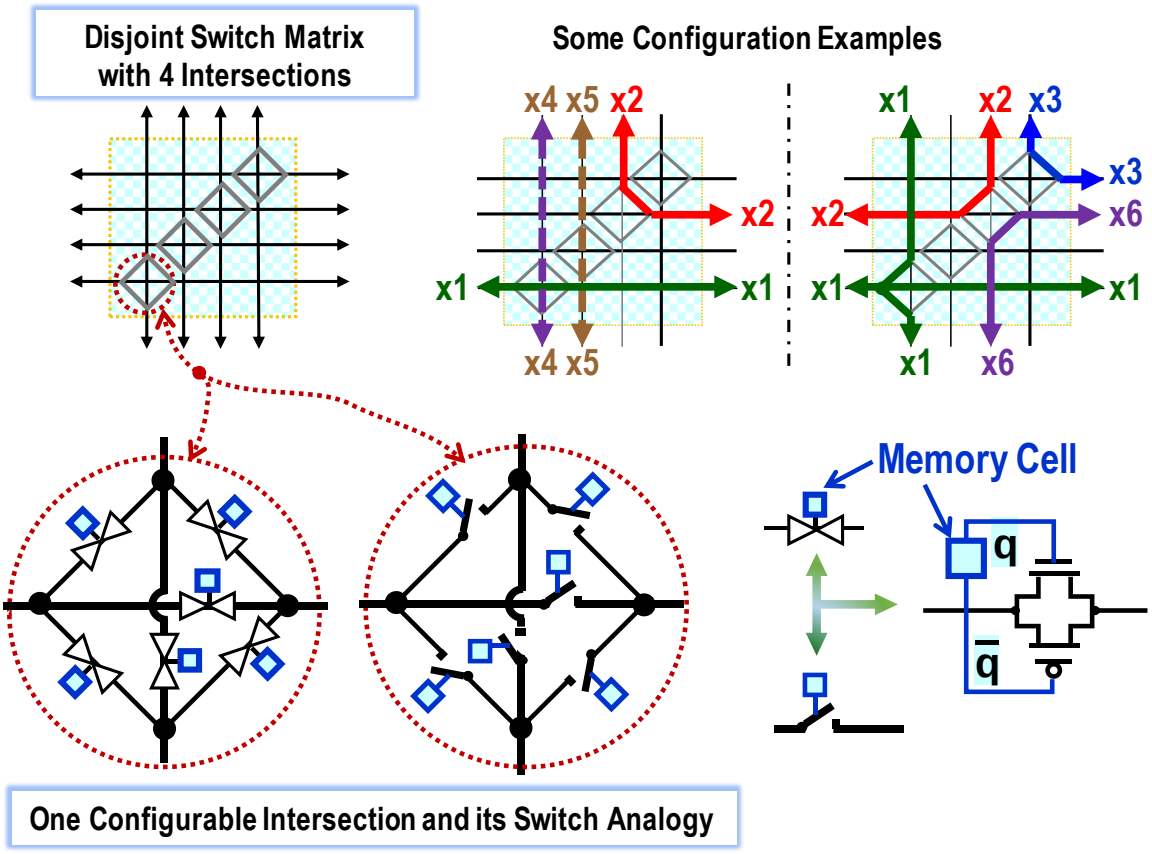


Figure 104 - *Disjoint* type connection nut

Cutout 3 on the previous page, Figure 102, showed only the resulting logic element interconnections for simplicity. They are implemented in a Connection Box, located at each logic element, which has a different structure depending on the manufacturer. Its components are again controlled by the memory cells set when programming the FPGA to our circuitry.

The outputs can, for example, be implemented by three-state inverters, which are fast (p. 60 and 64). In turn, the inputs are connected through transmission gates, which have high impedances in the disconnected state, by which they do not load the wire. As they have resistances of kiloohms in the switched state, voltage drops occur. When receiving a signal, its level should be restored to the full '0' and '1'. Different variations are used.

The following figure shows one possible solution with inverters and input pull-up resistors connected to the supply voltage distribution Vcc. They also guarantee logic '1' levels on the inputs even if all jumpers remain disconnected.

The size of the pull-up resistor can be selected in the order of megaohms[30] . So we sent the output to the wire through the inverter, and now it is received through another inverter, so the original signal state is restored.

[29] Other types of connection nut can be found in https://www.researchgate.net/publication/221224917.

[30] The specific value of the pull-up resistance depends on the parameters of the CMOS technology. It is often implemented by NMOS transistor depletion technology, see chapter 4.2 on p. 54.

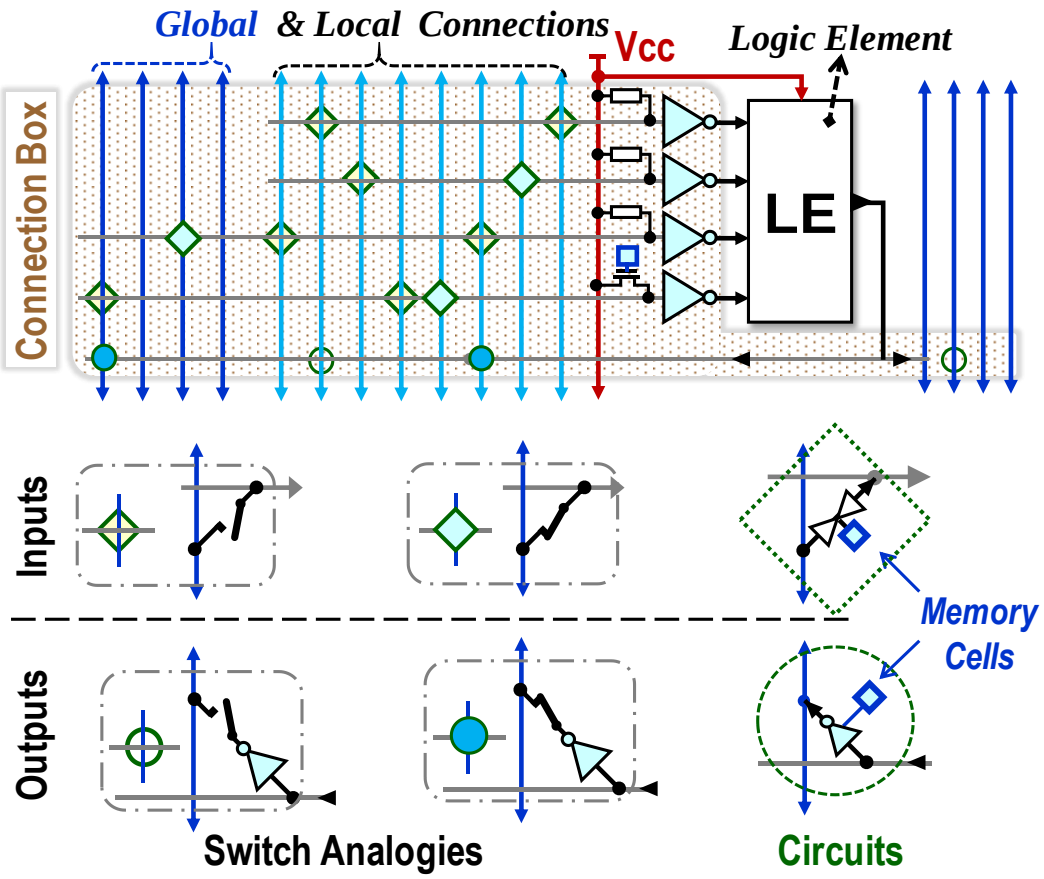


Figure 105 - Example of one possible interconnect array solution

The lower inverter of the LE also suggests an alternative solution, namely a configurable connection of the gate input to the Vcc. Unlike bidirectional transmission gates, the memory cell controls a single NMOS transistor, which is sufficient because the current only passes through it in one direction.

The positions of the connection points do not usually follow a regular pattern. For example, in a local channel, each of the sixteen logical elements of a single configurable logical block, CLB, may have them swapped differently. The manufacturers choose their placement according to their analysis of the interconnect algorithms results to produce a combination leading to the statistically best results.

Even the wires inside the CLB's local connection channel are divided into different segments so that one output does not exhaust the connection to all logic elements. In some FPGAs, there are also interconnection matrices between them, increasing the variability of their usage.

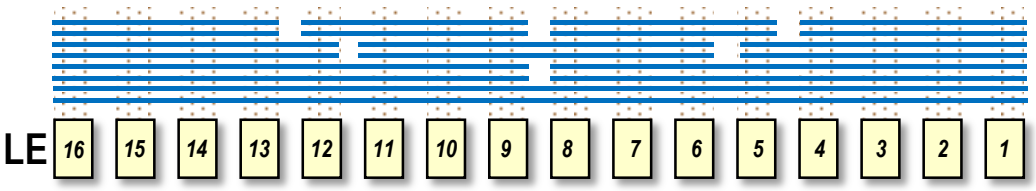


Figure 106 - Example of a possible local wire segmentation

Once the design environment has found the optimal layout of our circuit according to the known structure of our particular type of FPGA, it starts to solve the placement of the result into logic elements and the interconnections between them, i.e., placement and routing.

The task takes the longest time, increasing with the complexity of the circuit. It considers the

appropriate placements of the logic elements, their connections and loads, i.e., fan-out.

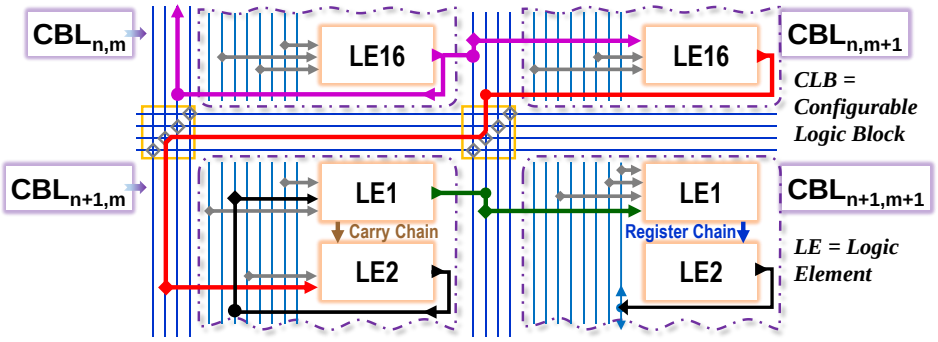


Figure 107 - Example of interconnecting logic elements in an FPGA

Heuristic compiler algorithms will give an acceptable result in polynomial time, but not always. Our experience shows that, on average, 70 to 90% of the available logic elements in the FPGA are exhausted. The deployment fails, and the most common reason is the missing connection. The specific percentage depends not only on the complexity of the circuit but also on its appropriate description. Non-optimal designs reduce the FPGA usability even below 60%.

5.5.7 Comparison of Cyclone II and Cyclone IV

The previous description focused on the smaller Cyclone II in our older DE2 development boards. Newly, LSP course uses more advanced VEEK-MT2 boards with Cyclone IV, which contains similar elements, just in more significant numbers. We provide a comparison of the two FPGAs.

Circuit class	**Cyclone II**	**Cyclone IV**
Type	**EP2C35F672**	**EP4CE115F29**
Technology	90 nm	60 nm
Logical elements	**33216** spread over 2076 CLBs	**114480** distributed in 7155 CLBs
Memory blocks	483840 bits (**105** M4K blocks)	3981312 bits (**432** M9K blocks)
DSP multipliers	**35** (18x18), or 70 (9x9)	**266** (18x18), or 532 (9x9)
User I/O	**475**	**528**
Price (year 2022)[31]	~ $ 20	~ $ 65

Table 5 - Comparison of FPGA Cyclone II with Cyclone IV

Both types contain 4 digital PLLs (Phase-locked Loops) for frequency multiplication and DSP hardware multipliers 18x18 bits, which can be individually configured to two independent 9x9 bit multipliers.

Memory bits are allocated in M4K blocks in Cyclone II. Each M4K contains 4096 bits + 512 parity bits usable only in a byte-width output configuration. In Cyclone IV, they are allocated in M9K blocks, each M9K having 8192 bits + 1024 parity bits for the byte configuration.

[31] When buying larger quantities, a discount is usually given, for example up to 20% for a purchase of ten thousand pieces.

5.6 Configuration memory elements in FPGAs

The flip-flop circuits in logic elements are built from CMOS transistor-based components and memory blocks implemented in the same way.

This section will discuss the memory cells suitable to configure the FPGA when programmed into a new circuit. We will briefly outline the characteristics of their three most common types. A more detailed discussion would require more space and will be left to specialized publications.

FPGAs, in which the configurable memory cells consist of CMOS transistors and thus resemble SRAM, are the most sold (according to US data for 2022). They are synchronously loaded during FPGA programming and have their value permanently available. Configuration of the FPGA is fast and without limitations on the number of repetitions.

Even our Cyclone II and Cyclone IV series FPGAs are SRAM-based.

Although the cell contents are lost when the power is turned off, adding an external permanent memory circuit to the FPGA is common. It stores its initial configuration, which is automatically loaded into the FPGA when the power is turned on, thus restoring the circuitry in the FPGA.

In the frequency of usage, the second place is occupied by FPGA types with antifuse memory elements named after their opposite behavior towards fuses. In the default state, they do not conduct; a voltage pulse irreversibly punctures them. They need external programming devices, cannot be configured soldered on the PCB, and are slow. Tens of minutes per circuit are quoted in publications.

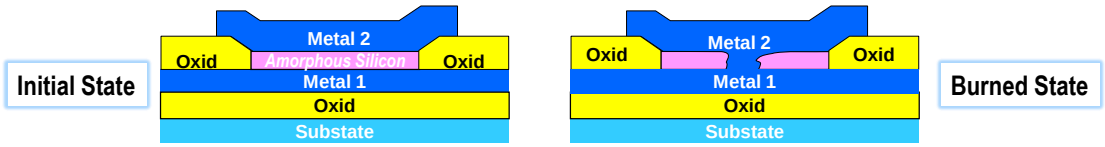


Figure 108 - Antifuse

Antifuses have inherently high radiation resistance and long-term stability. However, it is not possible to test their functionality in production. We will find out during configuration. The catalogs show statistics that the success rate is better than 95%, the so-called programming yield. In other words, if 100 units are configured, less than five will be faulty. And the manufacturers make it a condition that no complaints are accepted. Customers must take this into account and buy more chips.

Another popular type is reconfigurable FPGAs with **flash** memory elements, the same as in SSDs. They offer the features of retaining their contents after power is turned off and low idle power consumption when powered on. They are slower to program than SRAM FPGAs but much faster than antifuse.

FPGA circuits using SRAM or Flash for their configuration are more radiation-sensitive but are still deployed in space applications. For them, radiation-hardened or radiation-tolerant types are manufactured and equipped with shielding. If necessary, a new circuit configuration can be remotely loaded into them.

Are there other cheap options?

Some manufacturers offer universal semi-finished products, such as gate arrays, sometimes called ULAs, Uncommitted Logic Arrays. Another type of these is standard cells, which contain a collection of prefabricated small circuits, such as counters, shift registers, etc. They can then be further interconnected into our complete circuit on the same or later metal layers. It means we pay only for a few final integrated circuit technology steps.

At first, we buy several wafers with pre-made dies from a manufacturer. Then, we will place an order to a proper company for their adaptation to our circuit design debugged on an FPGA. It connects pre-made components by metal interconnecting wires isolated by dielectric layers to obtain our desired electrical circuit. Then, the wafer is cut into dies, which are tested and packaged into chips.

The total cost will be lower than in the case of all the integrated circuit technology steps but not negligible. It will pay off approximately only for a series starting around two thousand pieces. Small productions are nowadays created cheaper and faster on FPGAs.

6 Arithmetic combinational circuits

We will not plug in the comparators, adders, and multipliers ourselves; we specify the desired arithmetic operation in the design environment, which takes care of the implementation. It is helpful to know at least a little bit about their wiring properties. A combinational arithmetic circuit cannot share something, such as invoking a function or repeating a loop body.

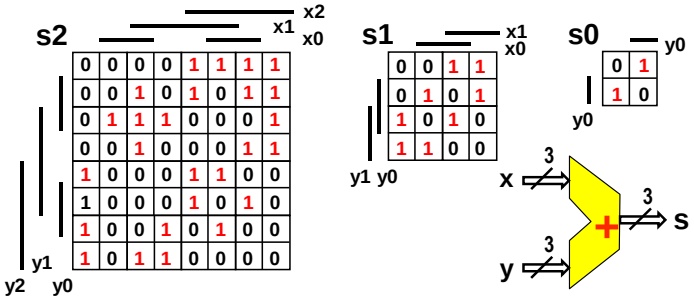
In a circuit, each operation is converted by inserting another computational block, and so the calculations are performed using the inline expansion technique. For example, a for loop is replaced by repeated insertion of its body.

We have to keep each operation optimal because they will all be present in the final circuit. And even seemingly small things will help to improve the result. For example, arithmetic operations with constants dividable by 2^N will execute faster. They have zeros in the least significant bits, so the higher bits are processed only. In contrast, the processor ALU works with whole numbers, so it adds the constant 79 as fast as 80.

6.1 Addition and subtraction

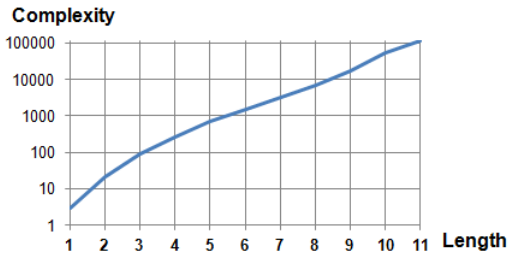
While we can describe the entire multi-bit adder by logical functions, we cannot minimize them efficiently. The reason is suggested by Karnaugh's adder maps, which contain small groups of both logical '0' and logical '1'. We would need a lot of implicants.

The fact is demonstrated in the figure on the left by the example of adding two three-bit binary numbers x = |x2|x1|x0| and y = |y2|y1|y0|, where x0 and y0 denote their lowest bits. The Karnaugh maps indicate the bits of their resulting sum s=|s2|s1|s0|.



The complexity of the direct adder implementation grows exponentially, as shown in the graph below, found by the Boom minimization algorithm mentioned in Chapter 3.5, p. 54.

The horizontal axis indicates the bit length of the single adder, and the vertical axis shows its complexity in implicant counts in all logic functions. The experiment stopped at the 11-bit adder because the time to wait for the result grew exponentially as its complexity.



The adder must necessarily be decomposed into smaller sub-elements, for which a simple KM of the lowest bit s0 is offered, which minimizes to s0 = x0 xor y0.

The half adder is a one-bit adder that does not consider a carry from the lower order. It has two outputs. The sum is S=x xor y. The carry to higher order bit is only generated when x='1' and y='1' and x+y is decadically equal to 2, binary "10" = | G | S |, i.e., G=x AND y.

$(x+y)_{dec}$	x	y	$(x+y)_{bin}$ G	S
0	0	0	0	0
1	0	1	0	1
1	1	0	0	1
2	1	1	1	0

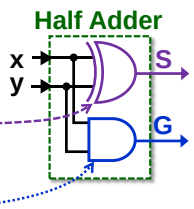


Figure 109 - Half-counting machine

The half adder is a building block of other adders, most notably the full one-bit adder, the Full Adder. It is created by combining two half-adders. The first one adds the inputs x and y. The second one adds the lower-order transfer Cin, Carry In, to its result.

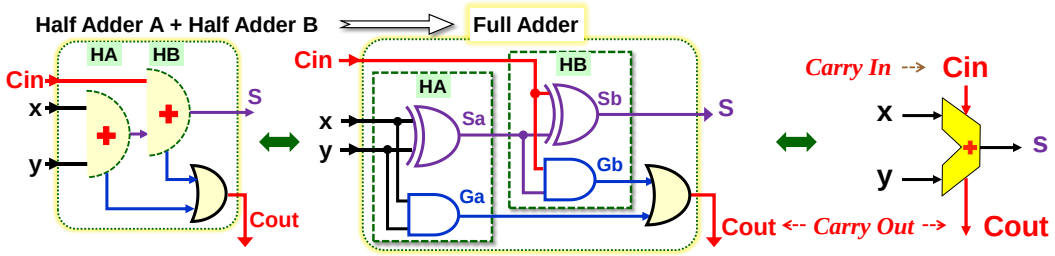


Figure 110 - Full adder, Full Adder

We concatenate the Ga and Gb outputs, generating the transfers of the two half adders by OR, since they are never both in logic '1'. Gb can only appear when Sa='1'. In that case, Ga='0'.

We've created a full adder. We can connect full adders in a series, so Cout of each one leads to Cin of the higher bit. The carriers are indexed according to the level to which they are sent.

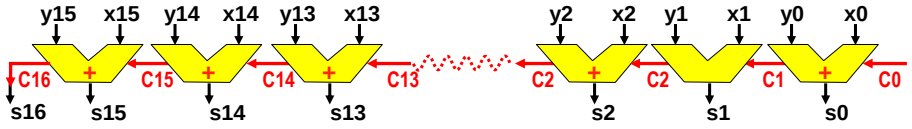


Figure 111 - 16 bit RCA type Ripple Carry Adder

The result is named the Ripple Carry Adder, RCA. It's the least carry input C0 is equal to '0' if it works independently. Each change of some Ci carry is propagated in series to higher bits in the style of waves[32] . The result will be valid only after all carries have settled. For example, the sum of $x=2^{16}-1$ and y=1 takes the longest time because the carry wave will run from s0 to s15, and the result will be 0 and s16=C16='1'.

The last C16 will also be the highest bit s16 of the sum since the sum of two 16-bit numbers gives up to a 17-bit result. If we add two unsigned numbers and take into account only the 16-bit result, then C16='1' is the overflow flag. Our sum no longer fits within the 16-bit limit.

How fast will our transmission count be? Schematics today present a human-readable description of the desired function of a circuit, not the exact internal structure of its circuitry. The physical implementation of the circuit uses shorthand constructs; see, for example, the XOR gate structure outlined in Chapter 4.6 on p. 64, which also did not build precisely according to its logic equation.

[32] The analogous ripple spread is called a domino or avalanche effect in economics and a chain reaction in physics.

We draw details of the RCA from the picture above a mark the critical path assuming that all its inputs x, y, and C0 have changed at once.

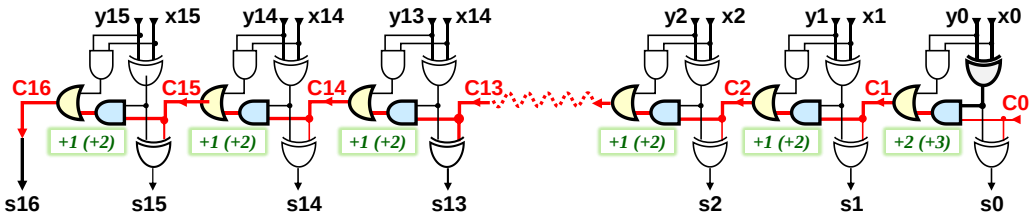


Figure 112 - Critical path in RCA

The summing time depends on the circuit implementation. In Chapter 4.4.1 on p. 62, we have created a unique AND-OR gate through which the transmission will propagate with a delay of one logic term. When we use it, C1 is delayed by two logical terms. In the meantime, however, the other half adders precalculate outputs. Carries only a pass through the AND-OR gates so that 16-bit RCA will settle for at most 2+15*1=17 elements of delay.

Let's see how the full adder is implemented in the LUT configuration of the FPGA for the carry propagation, in which we have two 3-LUTs, see Figure 98 on p. 88. We decompose the full adder by Shannon expansion cofactors with respect to Cin input. We obtained the pair of 2-input multiplexers with Cin as their shared address.

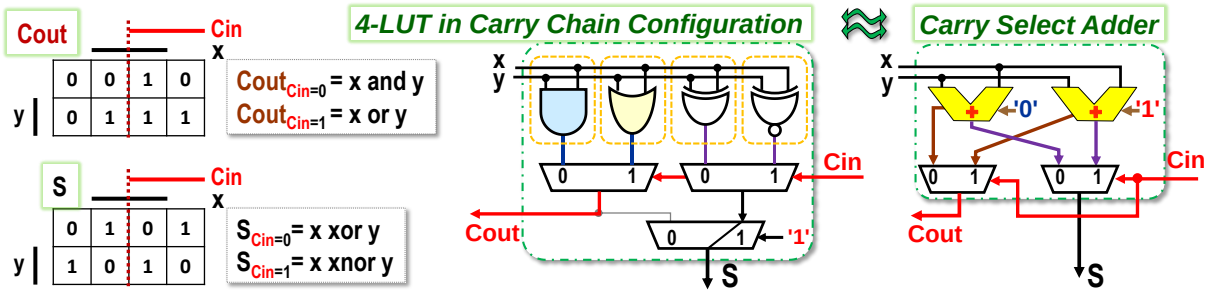


Figure 113 - Full adder in 4-LUT as Carry Select Adder

The full adder is duplicated into two with the same inputs, but one of them has its Cin permanently at '0' while the other is at '1'. Cin input toggles which one is selected. Internal FPGA implementation creates a full adder by a circuit called CSelA, Carry Select Adder. The 4-LUT is configured to two 3-LUTs; one calculates cofactors of Sum and the second of Carry.

The full adders are connected RCA style, but their transmissions will propagate through the multiplexers expressly, as shown in the figure below. Each input of a Ci carry only switches multiplexers of both 3-LUTs at once.

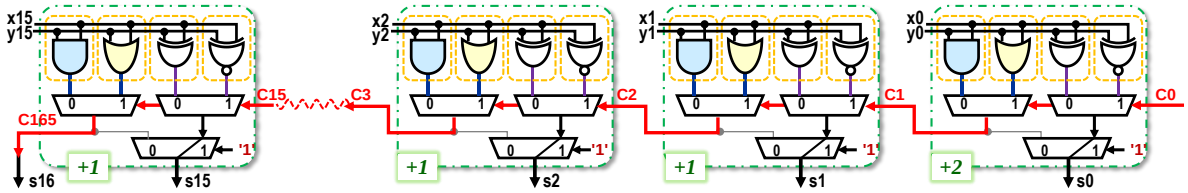


Figure 114 - FPGA implementation of 16-bit Ripple Carry Adder

If x, y, and C0 are changed at the same time, the carry to the output C1 is in 3-LUT generated by three rows of multiplexers. However, multiplexers are switched at once and are internally implemented by the transmission gates. Thus, the selected values of the higher multiplexer LUT row pass only through the resistors. The total delay of the least significant bit can also be considered as 2 gates. For the other bits, it is determined by switching multiplexers, which

means adding only one additional member in each step. Therefore, a 16-bit RCA adder has a delay of 17 elements in the FPGA, so it is as optimal as one implemented directly in CMOS.

The RCA adder won't be very fast. Another variant of the adder is based on Carry Select Adder, though, if we use multiplexers to switch RCA adders of progressively increasing bit lengths since the subsequent ones have had a longer time to settle their partial outputs.

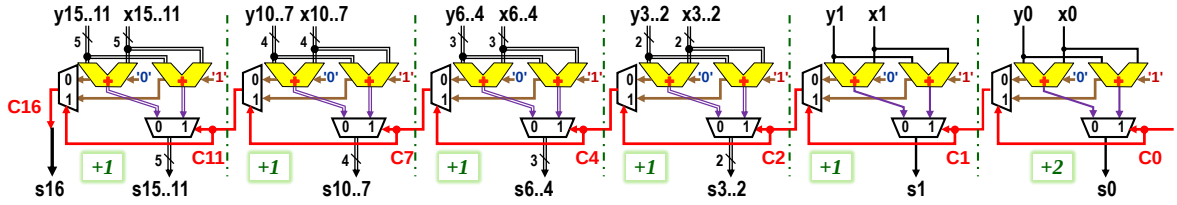


Figure 115 - 16-bit CSelA - Carry Select Adder

The delay for a 16-bit CSelA adder is only 7 elements, 2 for the first stage and 1 each for subsequent blocks. A 32-bit CSelA would only require the addition of three more blocks, and so its delay would be only 11 terms, roughly one-third that of the FPGA implementation of a 32-bit RCA. However, CSelA is not efficient either in the number of CMOS transistors used or in power consumption. It contains two RCA adders plus many bus multiplexers.

A carry prediction adder belongs among frequently used solutions known as CLA, Carry Lookahead Adder. It also uses half adders. Their sum outputs are here called Propage. In the figure below, these are P0 to P3. Their carry outputs are named Generate and used to predict the C1 to C4 transfers of the final sum gates of the XOR.

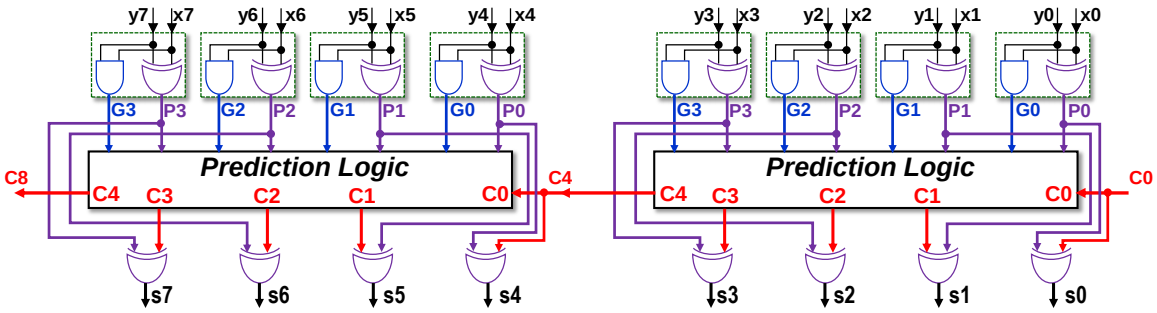
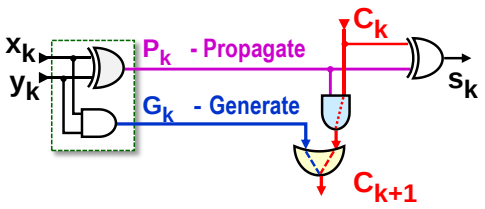


Figure 116 - First eight bits of the CLA counter with 4-bit prediction

The naming of outputs follows from the behavior of bits with index k in the RCA adder.

- A lower-order transfer of C_k passes or propagates to the output of C_{k+1} if and only if P_k ='1'.
- And G_k ='1' always generates the output C_{i+1} = '1'.



Their properties lead to prediction equations, in which we use the notation **.** and **+** for AND and OR to shorten them.

$$C_{k+1} = G_k + P_k . C_k; \qquad \text{(Ca1)}$$

In other words, we can express (Ca1) that a carry is sent to a higher order only if the full adder either generates it or allows a carry from a lower order to pass through. Let us break down the logic functions for carries C1 to C4 (For longer CLAs, Cj is written analogously for j>4):

$$C1 = G0+P0.C0;\ C2=G1+P1.C1;\ C3=G2+P2.C2;\ C4=G3+P3.C3;\ \ldots \qquad \text{(Ca2)}$$

RCA computes logic functions iteratively using the results of the lower bits, but these are the ones we need to put into relations if we want to speed it up. We expand the logic functions to

implicants:

C1 = G0+P0.C0;

C2 = G1+P1.(G0+P0.C0) =G1+P1. G0+P1.P0.C0;

C3 = G2+P2.(G1+P1.(G0+P0.C0))=G2+P2.G1+P2.P1. G0+P2.P1.P0.C0;

C4 = G3+P3.(G2+P2.(G1+P1.(G0+P0.C0)))

= G3+P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0; (Ca3)

In other words, we describe the relations (Ca3) such that a carry generated in some lower order propagates through the higher order units until all have their Propage outputs in '1'. A logical OR of all influences then gives the resulting Ck.

The number of terms in each equation C_k , where k>0 is the predicted bit, increases with (k+2)(k+1)/2, the sum of the arithmetic series. CLAs with up to 8 bits of prediction are also used, but most often only predict over 4 bits. A CLA adder with 4-bit prediction can, at the monolithic integrated circuit level, be implemented with a CMOS transistor count that is units of percent less than an RCA adder and with a power consumption that is only fifty percent higher than the RCA adder[33] .

The speed of the CLA counter again depends on its actual wiring. Implementing the prediction exactly according to relation (Ca3) is not convenient because the C4 expression has an AND implicant leading to a 5-input AND gate. We know from the CMOS chapter that it will be slower. In any CLA block, a change in its input C0 would propagate to C4 with a delay of up to 3 logic terms, not just two. Compared to RCA, CLA would only be a quarter faster, as the work mentioned in the note on the previous page demonstrates.

There are several tricks to engage the CLA better. For example, we can use our AND-OR gate as with RCA. The first four-bit block of the CLA will again operate with a delay of about 3 terms, since it predicts only after the results supplied by the input half-adders.

The following 4-bit CLAs utilize the prediction decomposition. While waiting for the transmission wave, they will precalculate intermediate results, e.g., for critical C4 they will be terms independent of their C0 input:

C4g = G3+P3.G2 + P3.P2.G1 + P3.P2.P1.G0; **C4p** = P3.P2.P1.P0**;** (Ca4)

When the transmission wave arrives at their C0, they quickly send the C4 output through the AND-OR gate:

C4= **C4g** + **C4p.**C0; (Ca5)

The higher CLA quads then add one delay to each of their AND-OR gates, except for the last one, including its output XOR. The improved 16-bit CLA can thus stabilize the result with a delay of 3+1+1+2=7 terms, comparable to CSelA and 2.4 times faster than RCA.

Are there any faster adders? Yes, they are called prefix adders, but they are more commonly referred to as **Prefix** Parallel **Adders**, PPAs, which already specify their function more precisely. The word "prefix" refers only to the mathematical notation used by the authors.

PPA uses relations (Ca3) but computes them by merging pairs of Generate and Propagate on a parallel binary tree structure whose nodes contain pairs of simple logic functions, namely AND and AND-OR called an operator. If the length of the PPA is doubled, only one additional layer

[33] R. Uma, Vidya Vijayan, M. Mohanapriya, & Sharon Paul. (2018). Area, Delay and Power Comparison of Adder Topologies. https://doi.org/10.5281/zenodo.1410195

is added to the prediction tree. Thus, the delay is increased by only one element.

The PPA predicts all carries of the adder, perhaps even over 128 bits, counting transfers from C1 to C128 at once. However, they have significant drains from the power source and complex interconnections. They are mainly used in large processors for long adders. For shorter ones, it does not excel as much.

The fastest known PPA adder is KSA, the Kogge-Stone Adder, which has a complete prediction tree but also the most power consumption and chip size. There are quite a few other PPA implementations that try to remedy this and reduce the tree without dropping the enumeration speed too much.

While the PPA principle is an example of a perfect implementation of the algorithm at the CMOS level, we will leave it to specialized publications. Its structure will be only slightly approximated on p. 104.

Why do FPGA logic elements use slow RCA adders? There are several reasons for this:

- According to data in various literature, RCAs have the lowest power consumption of all possible adders.
- The linear RCA arrangement is straightforward to connect.
- The carry propagation, Carry Chain, to which the LUTs in the FPGA logic elements are configured, will be advantageously used elsewhere, for example, in comparators. Prediction logic would only serve CLA.
- In circuits, we often work mainly with shorter numbers, which RCA can handle in a reasonable time.
- **The circuit is mainly accelerated** by suitably decomposing its function into parallel structures. PPA adders do not use better gates, just more convenient interconnections.
- More powerful FPGA types apply acceleration more naturally to them. They have variable LUTs with multiple inputs, up to eight, for example, in the Intel Stratix IV FPGA. They can also be configured for two outputs so that a single logic element can implement a two-bit adder. Transmissions are then propagated in two-bit jumps, i.e., twice as fast. Additional logic elements can implement higher-bit prediction logic. Their adders then perform comparably to CLA.

6.1.1 Subtraction

A full subtractor, a full subtractor, can be connected from two half subtractors.

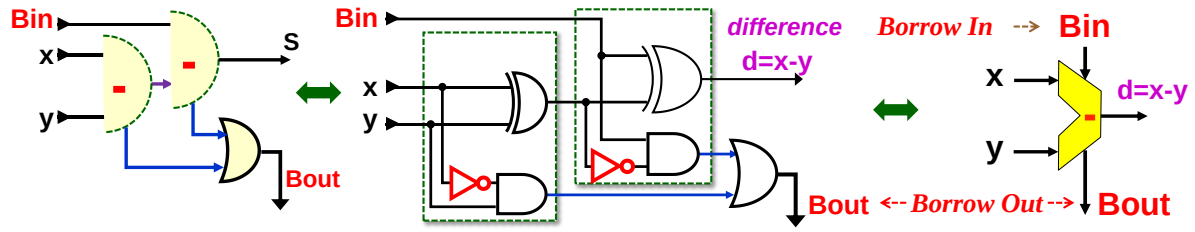


Figure 117 - Full subtractor folded in two halves

Compared to the full adder discussed earlier, Figure 110 on p. 98, we see only minor changes:

- The result is now called the difference, and so has the symbol d.
- The transfer of the subtractor is called Borrow, because in the operation '0'-'1' must borrow bit from the higher order. While this is a precise circuit term, the literature often

does not distinguish between Borrow and Carry. In particular, arithmetic processor units use the carry designation in both cases.

- In the carry path in the half subtractors, only inverters were added in the minors, i.e., the input from which we subtract. If this is equal to '0', then subtracting '1' will set the Borrow output.

The subtractor is also easy to decompose into LUTs in its Carry Chain configuration, so it works as fast as the adder.

In processors, subtraction is more often replaced by adding a negative number. The subtracted input y is converted to it by negating all its bits (the first complement) and setting C0 to '1', i.e., adding +1, to obtain -y in two's complement, the most widely used signed format in today's computers. We explained in our publication Binary Prerequisite.

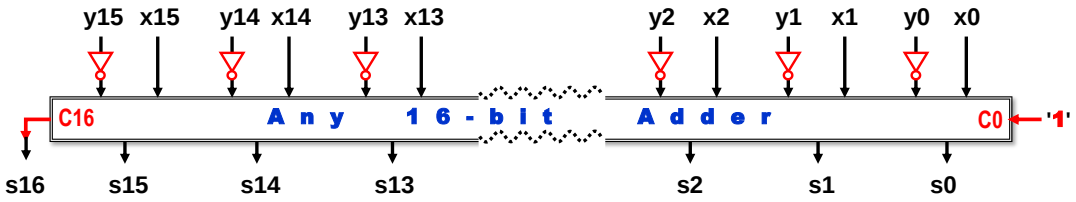


Figure 118 - Implementation of x-y in 16-bit arithmetic

Let's recall part of the prerequisite. The addition of signed and unsigned numbers is, in terms of physical implementation, done precisely the same way, and only the overflow of the result is evaluated differently. In the case of unsigned, it is indicated by the carry of the high bit. For signed, we are talking about overflow, which tests the validity of the adders' highest bits (signs) and the sum. It is set when the result is nonsensical, such as a negative sum of two positive numbers, etc.

Switching between addition and subtraction is often helpful in processors. Inverters are then replaced by XOR gates, which we know about in Chapter 2.3.1 on p. 23, that behave like a gate switchable between buffer and inverter behavior for one of their inputs.

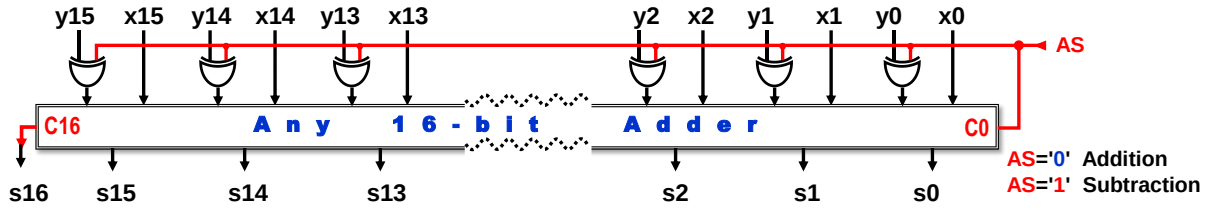


Figure 119 - Universal adding and subtracting machine

6.1.2 Addition and subtraction of constants

Here, let us first recall a few obvious facts:

- The constant C, which is divisible by some power of 2^M , has its M lower bits set to zero, and so in the operations x+C and x-C, respectively, bits 0 to M-1 of input x lead directly to the output. Only the upper part of x is added or subtracted to speed up the operation.
- Design environments also minimize adders or subtractors according to the bits of the attached constant.

When adding or subtracting powers of 2, the circuit is significantly reduced. We will show its construction for adding 1, i.e., 2^0.

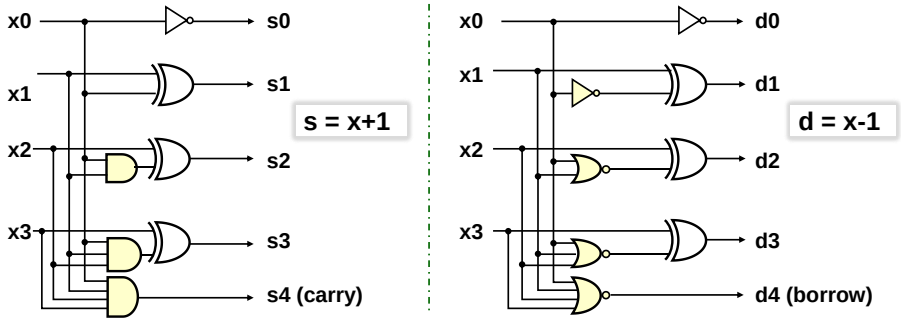


Figure 120 - Adding and subtracting 1 for a 4-bit number

In other words, the adder can be expressed by a pair of rules:

- The lowest bit is always inverted, both when adding 1 or subtracting 1;
- When adding 1, the bit is inverted, using XOR as a buffer/NOT controlled element, if all lower order bits are equal to '1', it is the property of a series of binary numbers:

  0000 00**01** 0010 0**011** 0100 01**01** 0110 **0111** 1000 etc.

- When 1 is subtracted, on the other hand, it is tested whether the lower bits are all in '0'.

  0000 1111 11**10** 1101 1**100** 1011 10**10** 1001 **1000** 0111... etc.

If the number x has more bits, then the length of the AND/NOR gates starts to increase. We limit their growth by calculating in parallel relations of the type

$$AND\_m\_0 = x_m \text{ and } x_{m-1} \text{ and } \ldots \text{ and } x_1 \text{ and } x_0$$

We decompose them using the associativity theorem (p. 16) into a tree structure in which we impose a limit on the use of AND gates with at most 4 inputs. To increase clarity, we do not draw the whole tree but only a quarter of it. The other AND_m_0 would be determined analogously. For the subtractor tree, only OR gates would be used instead of AND gates, and the inverter would be put after their result, as will be hinted on the next page.

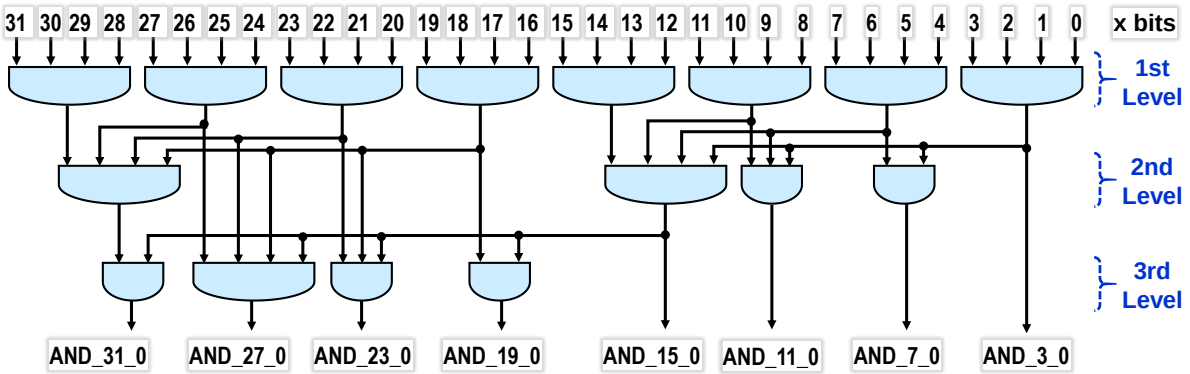


Figure 121 - Calculating AND_i_0 on a parallel tree structure

We can see that 3 levels of gates in parallel evaluate all the AND_m_0 terms needed for a 32-bit adder +1. Three levels would be also sufficient for a 64-bit length, only 128-bit numbers would need to add another layer of AND gates.

Our adder adds +1 in half the time of the fastest known adder KSA, whose parallel prefix basis also uses a merge tree. In it, however, it combines predictions with more complex expressions that can only be concatenated in pairs. Thus, 32-bit KSA needs 5 layers of the merge tree, plus 1 for prep and 2 for termination. Our adder or subtractor 1 needs only 3 layers of gates and one terminator with XOR gates.

The C programming language got its famous ++ and -- operations precisely because of the express operations of adder and subtractor 1, which brought significant speedups, especially in the days when processors ran at megahertz frequencies. Special circuits implemented the addition and subtraction of 1. Machine codes included byte-lenght instructions for increment and decrement. On Intel processors, the assembly codes were INC and DEC, to which the ++ and -- operations were translated.

With the development of pipeline instruction processing, the importance of both instructions declined. Compilers of programming language generally replace them with the addition of +1 or -1, which newer powerful processor adders, while taking slightly longer to execute, also set additional status bits in the result. When running in 64-bit mode, Intel processors no longer have INC and DEC ─ their short codes have been allocated to other instructions, which are now more critical for program processing[34] .

At the FPGA level, using a constant for addition and subtraction saves mainly logical elements. The carry will again be propagated through the Carry Chain. For +1/-1 operations, a single member drops out of the adder/subtractor at bit x0, see figure below, which means only a slight speedup. The OR gates are chained, the invertors is after them. Then again, more powerful FPGAs with variable LUTs that allow two-bit outputs will also speed up running in pairs of bits.

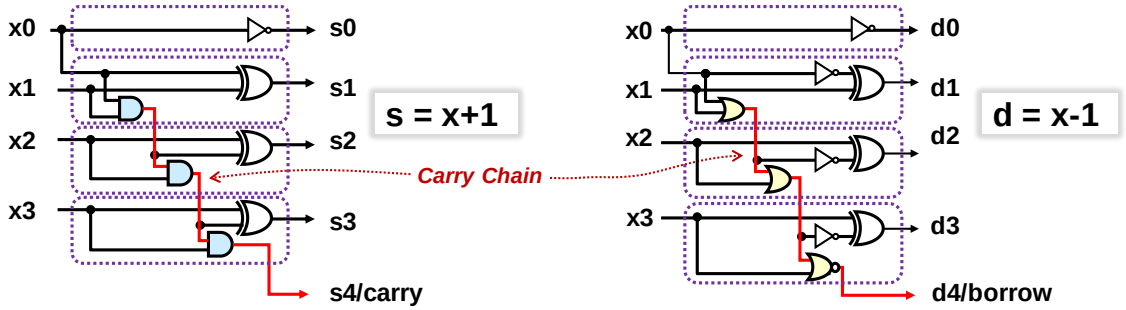


Figure 122 - Implementation of 4-bit adder and subtractor 1 in FPGA logic elements

6.2 Comparators

The equality test of two numbers is effectively involved by parallel bit comparisons, e.g. XOR gates that return '1' when the bits are different. If the AND gate concatenates the results, we get an inequality comparator, and when by the NOR gate, we get an equality comparator. The figure below indicates the operation layout on FPGA logic elements containing 4-LUTs.

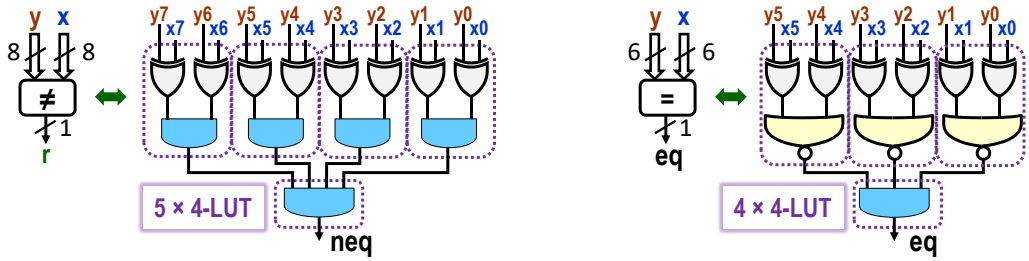


Figure 123 - Inequality and equality comparator on 4-LUT

A general comparison can be implemented by a Mux 2:1 cascade, in which each stage receives

[34] INC and DEC codes are assigned REX.R prefixes in x64 assembler to specify that the following instruction will use either access to the extended register set or a 64-bit modification of an older instruction from the i386 subset.

information on whether all lower bits have satisfied the condition and sends its result upwards.

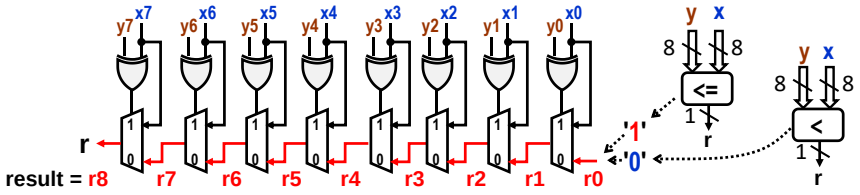


Figure 124 - Principle of the comparator y<=x and y<x

Comparisons x<=y and x<y work the same but differ in the last bit. If in the m-indexed bit, xm does not equal ym, i.e., ym xor xm='1', the lower bits are meaningless. Thus, only the result of a comparison to a higher order as a partial result r_{m+1} . For her, the comparison is valid under xm='1' and ym='0', and false under xm='0' and ym='1'.

If the bits xm and ym are equal, the lower order condition r_m is passed to the output r_{m+1} .

Only when the lowest bit is matched, the type x<=y or x<y is decided by the result sent by the least bits comparison, for x<=y it is '1' because the condition is met, for x<y it is '0'.

FPGA logic elements take advantage of their Carry Chain configuration. Translating the above conditions into logic functions, we obtain a decomposition into logic elements LE0 to LE7.

The lowest LE0 has a standard configuration but sends its output to a direct connection to LE1. Higher logic elements operate in a Carry Chain configuration. Unlike the adders, the sum bit is not sent out when comparing, so only one 3-LUT is used. Between them, the partial ri results propagate as fast as in the adder. The last r8 is the result of the comparison.

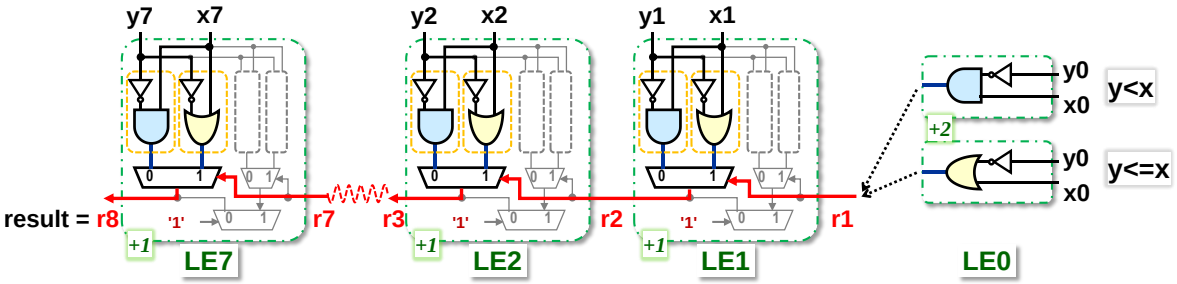


Figure 125 - Comparator decomposed into logic elements with 4-LUT

6.2.1 Comparison with constant

We can easily derive two basic rules:

- A comparison of the type x equals K, where K is an integer constant, or x does not equal K, requires the logical minterm since equality occurs for a single value of x.
- Other comparisons with K needs less LEs if both of the following conditions are met:
  a) If K is divisible by 2^M ; M>0 without remainder, then K has M zero lower bits from 0 to M-1 positions.
  b) The comparison condition can be written either in the form x<K or K<=x.

  In that case, the lower M bits of the number x then do not affect the result since they have weights 2^p ; p=0..M-1, less than 2^M of the first non-zero bit of the constant K. The compiler of our design only inserts a circuit comparing the upper bits of x. The comparator is smaller and faster

It can be beneficial for counters that are often tested to achieve a desired value. If it is not

strictly necessary to compare for equality for some reason, then the **inequality leads to a simpler** circuit. Although the FPGA has a lot of logic elements, it is still helpful to form more favorable conditions.

6.3 Constants used for multiplication, division, and modulo

Let K>0 be an integer constant and x a signed or unsigned binary integer. We will study the circuit implementation of operations that have notations in the C language:

x * K; x / K;

x % K; // *the remainder after division, written in VHDL as* x mod K (A1)

6.3.1 The power of two: $K=2^M$; M>0

Powers of two are popular values in computer science. In circuits, they are the fastest operations to execute (A1) expressions because they are implemented by simply connecting wires, as shown in the figure below.

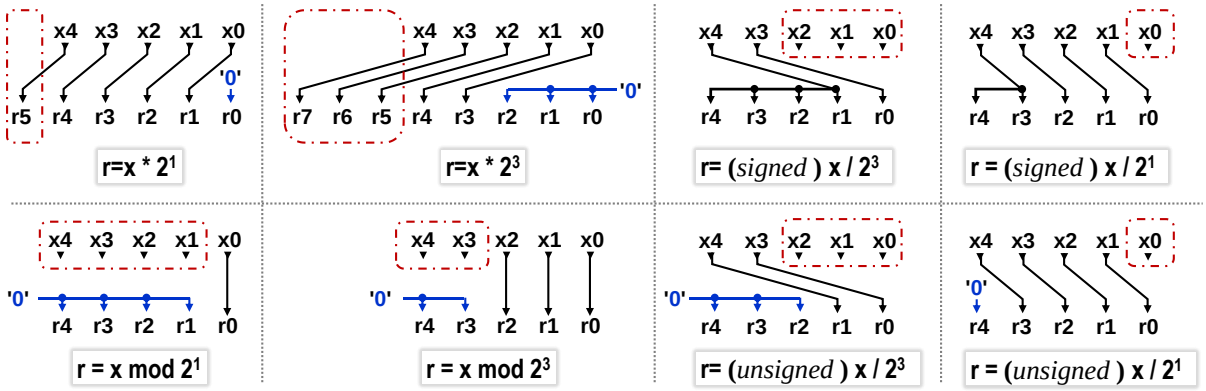


Figure 126 - Power of two constants and 5-bit number x

- Multiplying $K=2^M$ is shifting left[35] by M bits. If a result of the same bit length as the input x is desired, the upper bits disappear. They are marked with a frame.
- The division $K=2^M$ corresponds to a rightward shift that drops the lower bits 0 to M-1, so **integer arithmetic truncates everything below the binary ordinate**, which we will consider in later chapters.
- The difference is in the division of unsigned numbers, the unsigned and signed types. In the case of unsigned x, the upper bits of r are filled with logical '0's, while in signed, the highest bit of x, x4 in the figure, is copied in because it specifies the sign that must be preserved.
- The modulo operation, the remainder after division, is implemented by simply selecting bits with indices 0 to M-1. The remaining bits of the result r are filled with '0'.

Note: Even in programming languages, the above operations with 2^M are often compiled to shifts. The remainder of the division is implemented by a mask that selects the M lower bits by C-language &operator, bitwise logical AND.

[35] Shift directions in circuits are always given by the weights assigned to the bits, not by their distribution on the schematic. A shift to the left corresponds to moving a bit to a position where it will have a higher weight, the opposite is true to the right. In the language, a left shift is << and a right shift is >>. Processors implement both quickly on multiplexers.

6.3.2 Multiplication by the sum of powers of two

If an integer constant can be expressed as $K=2^{M1}+2^{M2}$ where integers $M1>=0$ and $M2>=0$, then all multiplications $x*K$ are realized as sums of two values in the circuit, namely the input x shifted to the left by $M1$ bits and by $M2$ bits. Although hardware multipliers could evaluate the result just as quickly, see Chapter 6.3.6 pg. 112, they are all located at fixed positions inside the FPGA chip. The adder can be built from logic elements anywhere, giving the compiler more freedom in placing the circuit elements.

Try the usage of constants in this form if it is possible. The rule is helpful, for example, for selecting the dimensions of matrices stored in SRAM.

Example: In a circuit, we need values from a 22x30 matrix. If we store it in row-major order, elements in the same row are stored consecutively. We calculate their addresses from indices by multiplying 30, the row length; see figure below. Rearranging in column-major order doesn't help; we would multiply by 22.

If we need to calculate the memory address of a matrix element at more locations, each evaluation will require a hardware multiplier in the circuit.

If we have enough free memory, we prefer redundancy. We add columns, perhaps filled with 0, to appropriately align their number to a more convenient constant, preferably a power of 2. We expand our matrix by two zero columns to 22x32. We then multiply by 32, which is involved simply by reconnecting the wires.

	0	1		29
0	x0,0	x0,1	…	x0,29
1	x1,0	x1,1	…	x1,29
	…	…	…	…
21	x21,0	x21,1	…	x21,29

memory address = 30 * row_index + column_index

address	0	1		29	30	31		30*21-1
element	x0,0	x0,1	…	x0,29	x1,0	x1,1	…	x21,29

	0	1		29	30	31
0	x0,0	x0,1	…	x0,29	0	0
1	x1,0	x1,1	…	x1,29	0	0
	…	…	…	…	0	0
21	x21,0	x21,1	…	x21,29	0	0

The simpler evaluation of memory addresses
address = 32 * row_index + column_index

address	0	1		29	30	31	32	33		61	62	63	..	32*21-3	32*21-2	32*21-1
element	x0,0	x0,1	…	x0,29	0	0	x1,0	x1,1	…	x1,29	0	0	..	x21,29	0	0

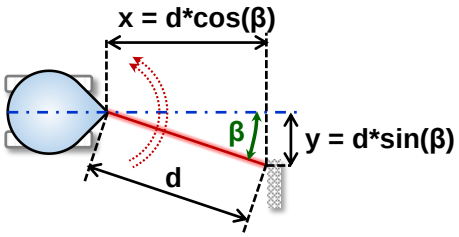
Figure 127 - Perimeter-adapted matrix

It is also possible to adjust the number of columns to the sum of powers of two. The calculation is done by adding the two shifted numbers without the hardware multiplier.

6.3.3 Multiplying by real numbers

In FPGAs, sometimes real numbers are unavoidable. We can give an example of recalculating information from a laser rangefinder whose beam is swept by a rotating prism, which creates its oscillation from the right side to the left side. The data flow is extreme, so the measured data must be preprocessed by a hardware accelerator so as not to overwhelm the processor.

In the first stage of the accelerator, the measured distances d are converted to positions x and y to the drone axis in a preprocessing step containing filtering. Still, we limit the example to its initial combinational block of conversion to x and y coordinates.



In the recalculation, we address two issues:

1. The incremental sensor sends the instantaneous rotation of the rotating prism, which we convert by integer arithmetic operations to the angle β from the drone axis, for which we calculate the **sine and cosine**. Goniometric functions, or other complex ones, are not computed in the circuits but tabulated in ROM. FPGAs commonly contain 2-port memories that allow reading from two different addresses simultaneously; we will try these in our LSP course assignment. We convert the angle value to ROM addresses, where we read sin(β) and cos(β).
2. Goniometric functions give real numbers in the interval <-1;1>. How do we store them? The answer from her is that we use fixed-point, fix-point arithmetic.

Fixed-point arithmetic expresses all numbers as fractions with the same denominator 2^N, where integer N>0. The choice of N depends on our desired precision. We consider that the angle β and d distance are measured with errors that can be expected to be around 1%.

In fixed-point, we have to check the range to avoid underflow of the result; otherwise, we manipulate them in the same way as with integer numbers, according to the rules for fractions with the same denominator. Multiplying an integer by a constant is easy, but the product of two numbers in a fix-point is then corrected by shifting to the right by N bits to keep the denominator the same. However, this cuts off the lower bits, so we add rounding by adding half of 2^{N-1}, i.e., half of 2^N.

$$\frac{Pa}{2^N} \pm \frac{Pb}{2^N} = \frac{\boldsymbol{Pa \pm Pb}}{2^N};\ K * \frac{Pa}{2^N} = \frac{\boldsymbol{K * Pa}}{2^N};\ \frac{Pa}{2^N} * \frac{Pb}{2^N} = \frac{Pa * Pb}{2^{2N}} = \frac{(\boldsymbol{Pa * Pb + 2^{N-1}})/\boldsymbol{2^N}}{2^N}$$

If we choose N=10, then 2^N = 1024. We table the values of the sine function in the appropriate step. We only need the range 0 to 90 degrees from which to derive the other values. However, each value is multiplied by 1024 and stored as an 11-bit integer.

The 1.0 number is converted to 1024. If the angle β has a value, for example, 10 degrees, then its sine will be 0.173648..., but in the ROM, it will be multiplied by 1024 and rounded to **178**. The cosine value can be found in the same table at an angle of 80 degrees, 0.984807 is multiplied by 1024 and rounded to **1008**.

If the rangefinder reports 900 mm, our accelerator calculates xfix=900*1008=907200 and converts the result back to an integer by shifting it to the right by N bits, preceded by 2^9 for rounding. Calculate x=(**907200+2^9**)/2^{10} = 886. Exact x=886.327. Our x has an error of only 0.03%.

By analogy, we get yfix=900*178=160200, of which y=(160200+2^9)/2^{10} = 156 after rounding. Its value has an error of 0.2%. (Exact y=156.283). Thus, we are sending coordinates to the next accelerator stages with smaller errors than the initial data.

6.3.4 Division by a small constant

Even the number 1/K can be expressed as a fix-point, i.e., a fraction of $P/2^Q$. We will show one possible way to find more accurate integer constants P and Q by hand.

We will use the basis rule[36] , that every integer K>0 can be decomposed into $K=2^Q * D$; where Q>=0 and D is an odd number. It also holds that integers P>0 and N>0 exist for every odd integer D, such that $D*P=(2^N -1)$. However, sometimes we can quickly find the decomposition $D*P=(2^N +1)$ that exists only for some D, for example, $17 = 2^4 +1$.

When we find the decomposition, we can approximate the division by two possibilities:

$$\frac{x}{K} \approx E_{LT} = x * \frac{P}{2^N} \quad \text{expression } E_{LT} < x/K; \text{ It has } \textbf{-error} = -\frac{1}{K * 2^N} \tag{A2}$$

$$\frac{x}{K} \approx E_{GT} = x * \frac{P+1}{2^N} \quad \text{expression } E_{GT} > x/K; \text{ It has } \textbf{+error} \approx \frac{K-1}{K} * \frac{1}{2^N} \tag{A3}$$

With a suitable N, however, the errors are beyond the resolution of our representation of the numbers.

Example 1: Convert division by x/10 to multiplication.

Number $10 = 2*5$. We express 5 as $5=(2^4 -1)/3$ and modify it by the identity $(x^m +1)*(x^m -1)=x^{2m} -1$.

$$\frac{1}{5} = \frac{3}{2^4-1} * \frac{2^4+1}{2^4+1} = \frac{3*(2^4+1)}{2^8-1} = \frac{51}{2^8-1} \approx \frac{51}{2^8}$$

We can further refine, according to our current needs, by adding more steps:

$$\frac{1}{5} = \frac{3*(2^4+1)}{2^8-1} * \frac{2^8+1}{2^8+1} = \frac{3*(2^4+1)*(2^8+1)}{2^{16}-1} = \frac{13107}{2^{16}-1} \approx \frac{13107}{2^{16}}$$

But we want 1/10, so we will divide by 2 more, so 2^{17}. We can't do more than that because the bit length of the product results would grow too much.

The approximation (A2) has a small error, but negative, integer operations cut off the bits below the binary dot, mainly affecting divisible K values without reminder.

The more convenient approximation **(A3)** gives better results than rounding:

$$x/10=(\mathbf{x*13108+2^{16}})/2^{17} \tag{A5}$$

so for integer division, where its positive error correctly sets the part above the binary dot:

$$x/10 \approx (\mathbf{x*13108}) / 2^{17} \tag{A6}$$

Both approximations (A5) and (A6) divide by ten exactly the numbers x, which also have 14-bit lengths. The first time it errs is at 15 bits, which is already close to the bit length of the chosen base.

Example 2: Convert division by x/11 to multiplication.

The closest form is found in $11*3=33=2^5 +1$.

$$\frac{1}{11} = \frac{3}{2^5+1} * \frac{2^5-1}{2^5-1} = \frac{3*(2^5-1)}{2^{10}-1} = \frac{93}{2^{10}-1} \approx \frac{93}{2^{10}}$$

$$\frac{1}{11} = \frac{3*(2^5-1)}{2^{10}-1} * \frac{2^{10}+1}{2^{10}+1} = \frac{95325}{2^{20}-1} \approx \frac{95325}{2^{20}}$$

It approximates by using (A3) as either $x/11 \approx (\mathbf{95326+2^{19}})/2^{20}$, i.e., with or without rounding of the fraction $x/11 \approx 95326/2^{20}$. However, the constant is 17 bits long, and in 32-bit arithmetic, it can be applied to numbers of length 14 bits, both with and without rounding.

---

[36] The relationship can be proved through congruence theory and simple reasoning. The binary representations of the number $2^N -1$ are all 1. The odd number D in turn has its lowest bit d0 = 1. We will therefore add its left shifted values to D so that the bit d0 is successively filled with all 0's, both the original and the 0's generated. The sum of the bit weights of the shifts gives the number P.

6.3.5 More accurate integer multiplication and division by a real number

Both operations are equivalent. We convert division to inverse multiplication, but as the fixed-point fraction's denominator increases, the intermediate results' bit length increases.

In classical programming, we can choose, for example, the double format, but the design environment usually can only automatically perform integer operations up to 32 bits long. Longer than that, we have to decompose. There are more options. Horner's scheme for computing polynomials best approximates integer division, slower but more accurate. Integer division will cut off the lower bits one at a time in smaller chunks.

Example of division by 10: We convert the operation to multiplication by the real number 0.1, whose conversion to binary gives an inexact number in which groups of bits "1100" are repeated endlessly. If we want precision to 6-decade digits, we try several powers of two around 2^{24} , until we find a nice hexadecimal form. $0.1 \approx 838861*2^{-23} = 0xCCCCD*2^{-23}$. It rounds up even at the cost of the last digit of D. We need the positive error of the result to compensate for the truncation of the lower bits. Let's decompose the number in any radix of power 2, say $16=2^4$, i.e., by hex digits.

$$x * 0.1 \approx \frac{x * 12 * 2^{20} + x * 12 * 2^{16} + x * 12 * 2^{12} + x * 12 * 2^{8} + x * 12 * 2^{4} + x * 13}{2^{23}}$$

We truncate the fraction by dividing by 2^{20}. To share the x*12 value, we adjust the last term to 12*x+x.

$$= \frac{x * 12 + x * 12 * 2^{-4} + x * 12 * 2^{-8} + x * 12 * 2^{-12} + x * 12 * 2^{-16} + (\boldsymbol{x * 12} + \boldsymbol{x}) * 2^{-20}}{2^{3}}$$

We write Horner's scheme for calculating polynomials backward in order to arrange the terms in the direction of their evaluation since the addition must start from the smallest term.

$$x*0.1 \approx (((((\ \mathbf{x*12 + x}\ )/2^4 + x*12\ )/2^4 + x*12\ )/2^4 + x*12\ )/2^4 + x*12\ )/2^7 \qquad (F1)$$

We test the proposed approximation in the program but do not write it in one expression, which the compiler could decompose differently or even multiply. We want a precise order of operations. We'll use the iteration x*12, which will be performed in the circuit as the sum of two shifted x's to x*8+x*4, so our division by ten will use only adders in a circuit.

```
int div10(int x) { int xk=12*x ;   int r = (xk + x) >> 4 ; r = (xk + r) >> 4;
                   r = (xk + r) >> 4; return (xk + r) >> 7; }
```

The algorithm also divides 22-bit x. We create a version with rounding by adding half the number it divides by before each rightward shift, which will be 8 and 64 at the end.

```
int div10(int x) { int xk=12*x ;   int r = (xk+x+8)>> 4; r = (xk+r+8) >> 4; r = (xk+r+8) >> 4;
                   r = (xk+r+8) >> 4; return (xk + r+64) >> 7; }
```

We have accurate results up to 21-bit input x. Let's try a more prolonged decomposition x*0.1 in a higher radix $256=x^8$. Its 8-bit constants work on 32-bit arithmetic exactly up to 23-bit x.

```
int div10ex(int x) { int xk=x*204; int r=(xk+x) >> 8; r=(xk+r) >> 8; return (xk+r)>>11; }.
```

By the same way, we can approximate multiplication by any real number. For example, the sine of 10 degrees, from Chapter 6.3.3, is approximated in radix 2^{10} by the expression: $((x*252+512)/2^{10} + x*269+512)/2^{10} + 711*x+2048)/2^{12}$ with rounding of intermediate results. Its 10-bit constants will compute sums even with 21-bit numbers x in 32-bit integer arithmetic, with a guaranteed error < 0.1%, but with an average of only $2.6*10^{-7}$ % compared to the value obtained by round(x*sin(M_PI*10/180)) computing in double precision.

The division with rounding can be computed more quickly by decomposing into parallel calculations[37] but with shifts to the right up to several orders of magnitude of bit lengths of radixes. These are poorly compensated for if we want to approximate integer division, which we will need right away in Chapter 6.4.1 on p. 114.

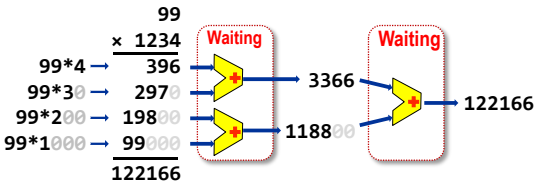
The approximations given here assume **integer arithmetic** with truncation of the lower digits. More significant errors appear if they were computed in float-point without truncations.

6.3.6 Hardware multipliers

FPGAs usually contain hardware multipliers, which we will only use, but we will still look at their principle. First, we already know all their components; we'll connect them. Secondly, this is again an excellent demonstration of the technique of decomposing an algorithm into parallel runs in circuits.

For simplicity, we will explain the algorithm of the hardware multiplier, which applies a principle similar to hand calculations, which we will describe in a more decadic system.

If we multiply, for example, the integers 99 and 1234, we get the results of four partial multiplications, always shifted by an order of magnitude. We can add them in parallel in pairs and then add the intermediate results, which will be the final product.



However, such an arrangement is not advantageous. We are not interested in the intermediate results of the left counting series. We only want to know the final sum of our product. Moreover, we are waiting for the two rows of adders to settle down, in which the transmissions are propagated as the lower-order carries are added. And that's holding us up.

What if we don't count them immediately but only at the end? We can bring them out in intermediate steps as another output. Let's illustrate the principle first on the top three results of the partial multiplications, namely 396+2970+19800. We decompose each number into its orders: sums of tens of thousands, thousands, hundreds, tens, and ones, which we process independently. The grey zeros indicate the order of the digit, we will copy those.

For example, add hundreds as three digits 3+9+8=20 and two zeros to the result. We don't expect any carries here. We add all orders in parallel and independently. We then construct two addends from their sums, writing the digit of the order in the first and the one that has flowed over it in the second. In the figure below, the underline highlights them.

```
  396 =                 300 + 90 +6
 2970 =          2000 + 900 + 70          11066
+19800 = 10000 + 9000 + 800              +12100
-------  -----  -----  ----  --  --      -------
 23166   10000 +11000 +2000 +160 +6  ➡   23166
```

Figure 128 - The principle of the CSA adder

[37] Other parallel methods suitable in FPGAs are discussed in Ugurdag F.,Dinechin F., Gener Y., Gören S., Didier L.: Hardware division by small integer constants. IEEE Transactions on Computers, 2017,

We reduced the original three addends to two by parallel computation, thus reducing the number of addends whose sum still gives the correct result.

An adder that can reduce is called a CSA, Carry-skip Adder. It is more commonly referred to as Carry-bypass Adder in some publications. Unfortunately, we could not find any consistent Czech terms. By successive addition, we can add the fourth number 99000.

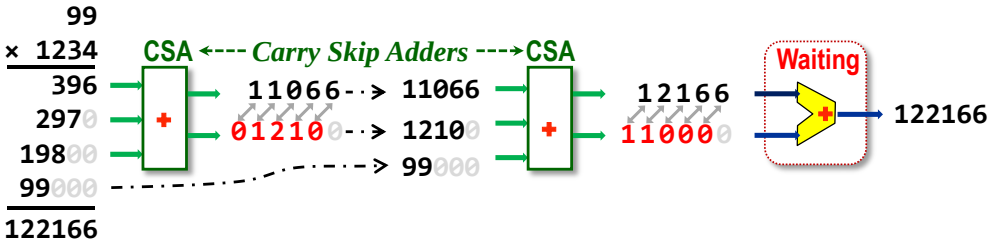


Figure 129 - Hardware multiplier principle with Wallace tree

Using CSA, we created a hardware multiplier called a Wallace tree after its author. Its structure explans why FPGA hardware multipliers have bit-length divisible by three, such as 18×18 or 9×9, to merge the partial multiplication results effectively. Wallace's addition tree reduces the multiplication time. It does not wait for carries in intermediate steps and creates the product with a slight delay compared to the adder. But it tends to be bulky for larger bitlengths. Other types of multiplier derived from it mainly try to reduce it without much slowdown.

However, the Wallace tree structure from the previous picture does not work for signed numbers. Negative results of partial multiplications in the binary system would need signed extensions to the entire length of the final result to remain negative. However, minimal intervention can modify the Wallace tree to the Baugh-Wooley algorithm. In it, one need only negate selected bits in the results of partial multiplications. Errors in adding negative numbers then cancel each other out. The mathematical explanation is beyond the scope of our textbook[38] .

But how do we plug in the CSA three-number adder? We don't have to; we already have it. After all, the full adder discussed earlier added three bits, the x and y inputs, and the lower-order C transfer. We output its transmission sent to the higher order as its following output.

The figure shows a comparison of RCA and CSA 3-bit counters. For CSA, they just introduced a different designation for the transfer input; now, it will be another number and removed the interconnection via transfers.

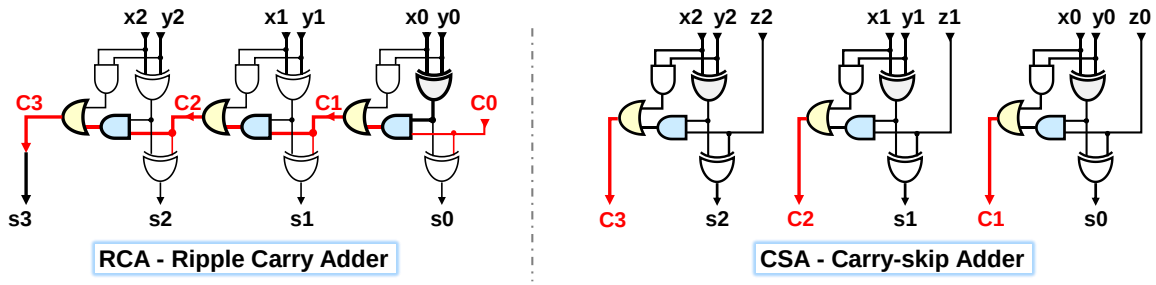


Figure 130 Comparison of RCA and CSA counters

The CSA adder also uses our AND-OR gate at the CMOS level, so it operates with a delay of only two terms. Thus, the intermediate steps of the Wallace tree do not delay much, and the hardware multiplication is only slightly slower than the addition.

[38] An outline of the Baugh-Wooley algorithm can be found, for example, at: https://www.dsprelated.com/showarticle/555.php.

6.3.7 Problematic general division of two numbers

General division cannot yet be parallelized efficiently. We can connect it with a sequential subtraction algorithm like a manual calculation, but we get a slow circuit with many logic elements. It can be used in a pinch if we can't find another way, but it's better to avoid it.

There are also fast divider circuits, but all known ones have complex circuitry that must be implemented at the CMOS level to remain efficient. High-performance processors often apply some variant of the High-Radix Division algorithm, which forms the result in groups of bits, hence the name, thus reducing the number of steps[39] . They select the groups of the highest bits of the immediate division remainder, the divisor, and the dividend. They concatenate them into a ROM address[40] , in which they load the probable partial quotient and the new remainder. They refine the estimate by iterations during the addition of partial multiplication results in the parallel multiplication tree.

A method well applicable in FPGAs is also to gradually expand the fraction from the divisor and divisor by $1+2^M$, until the error is reduced below the arithmetic resolution[41]. We convert division to multiplication but at the cost of a large consumption of FPGA elements.

The general division is not very suitable for implementation in FPGA logic elements. It is helpful to choose FPGA types with its hardware support if necessary.

6.4 Example: converting the algorithm to a circuit

6.4.1 Example 1: Converting a binary number to BCD

The number conversion is done in C by the function prinf(). Even in circuits, it is possible to use conversion, for example, to display a number on a 7-segment display. Then, we need to have each decade digit of the result 0 to 9 separately in four bits. A similar method is called the BCD format, Binary Coded Decimals, see Binary Prefix. Its form is identical to hexadecimal number notation, with the difference being that only the binary codes from 0 to 9 are in the quad bits. They do not contain 10 to 15, hexadecimally written as A to F.

We try first with a program, perhaps in C. We know algorithms are implemented in a combinational circuit using the inline expansion technique. Thus, cycles that depend on an input value, here input x, and for which the number of repetitions is not fixed, are unsuitable.

Our initial experiment may be based on a conventional algorithm with successive division. In it, we have the division and the remainder of the division close to each other. The C language compiler will then more easily detect that we need both the results of the DIV machine instruction, the quotient, and the remainder.

```
int byte2BCD_v1( byte x )
{   int bcd = 0, d, m;
    for (int ix = 0; ix <= 1; ix++)    { d = x / 10; m = x % 10; // our hint to C compiler
                                         bcd |= (m << (4 * ix)); x /= d; }
    return bcd |= (d << 8); // max. upper digit can be 2
}
```

[39] For example, the tutorial https://www.utdallas.edu/~ivor/ce6305/m13.pdf gives a clear description.

[40] It was 5 bad values in ROM memory that caused the famous division error in the Pentium processor (1994).

[41] The description states, for example. Paim, P. Marques, E. Costa, S. Almeida and S. Bampi, "Improved goldschmidt algorithm for fast and energy-efficient fixed-point divider," 2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2017, pp. 482-485, doi: 10.1109/ICECS.2017.8292070.

If we are going to wire the circuit according to the C code, then we have not designed it but programmed it. :-(We will convert division by 10 to multiplication by a fraction instead. The input is a byte, so we use the procedure from Chapter 6.3.4 on p. 109, which approximates integer division by 10 without rounding. The remainder is obtained by subtraction, thus: d = (13108 * ix) >> 17; m = x - d * 10;

We'll try a modified version that the compiler probably converts to inline expansion to make the for-cycle disappear, which is short and with few loop repeats. It would just stall.

Corrected code

```
int byte2BCD_v2(byte x)
{   int bcd = 0, d, m;
    for (int ix = 0; ix <= 1; ix++)
    {   d = (13108 * x) >> 17; m = x - d*10;
     bcd |= (m << (4 * ix)); x = d;
    }
    return bcd |= (d << 8);
}
```

Inline expansion

```
int byte2BCD_v2(byte x)
{  int bcd = 0, d, m;
   d = (13108 * x) >> 17; m = x - d * 10; // ix=0
   bcd |= m; x = d;
   d = (13108 * x) >> 17; m = x - d * 10; // ix=1
   bcd |= m<<<4; x = d;
   return bcd |= (d << 8);
}
```

Multiplication by ten is replaced in the FPGA by adding the two shifted values, $x*2^3 + x*2$, and the hardware multipliers quickly perform the d*13108 operation.

It is a better solution, but it is still close to programming. Each conversion to BCD is inserted as a separate circuit in the circuit, thus consuming two hardware multipliers. Even there, though, we can mimic function calls and share their blocks by using synchronous circuits that drive a finite automaton, the Finite State Machine, FSM. However, if we only need a few BCD conversions, we would add unnecessary complexity to our design.

How about doing without multiples? That's also possible. We know from Binary Prerequisite that an integer can be converted to binary by repeatedly dividing it by 2, which emulates right shifts. The vanishing lowest bits, i.e., remainders after division, are binary digits, only they go in order from bit 0 to the higher bits.

Example of conversion by integer division by 2 in decadic notation

13	13÷2=6; mod **1**	6÷2=3; mod **0**	3÷2=1; mod **1**	1÷2=0; mod **1**
1101	1101→0110 | **1**	0110→0011 | 0	0011→0001 | 1	0001→0000 | 1

Conversion using unsigned *shifts to the right*

The conversion can be reversed. We can also shift the bits of the binary number being converted from the highest to the lower ones. However, we must otherwise multiply by two.

Consider two BCD digits stored in an 8-bit number x, whose upper (bits x7 to x4) are 0000. We shift the BCD digits to the left along with the binary number x being converted, thus inserting its next upper bit, marked φ; φ=0 or 1. For BCD codes <= 4, the shift works correctly:

BCD	**0**	**0**	**x**	**0**	**1**	**x**	**0**	**2**	**x**	**0**	**3**	**x**	**0**	**4**	**x**
	0000	0000	φ--	0000	0001	φ--	0000	0010	φ--	0000	0011	φ--	0000	0100	φ--
←	0000	000φ	--	0000	001φ	--	0000	010φ	--	0000	011φ	--	0000	100φ	--
BCD	**0**	**0+φ**		**0**	**2+φ**		**0**	**4+φ**		**0**	**6+φ**		**0**	**8+φ**	

BCD digits >= 5 are multiplied by 2 by the offset but will increase to values >= 10, thus becoming illegal in its encoding; see the following table:

BCD	0	5	x	0	6	x	0	7	x	0	8	x	0	9	x
	0000	0101	φ--	0000	0110	φ--	0000	0111	φ--	0000	1000	φ--	0000	1001	φ--
←	0000	101φ	--	0000	110φ	--	0000	111φ	--	0001	000φ	--	0001	001φ	--
BCD	0	10+φ		0	12+φ		0	14+φ		1	0+φ		1	2+φ	

To get the correct result, we need to skip the six values 10 to 15 missing in the BCD, which we do by correcting each digit, i.e., four bits, separately before shifting. We adjust by adding +3 to each value greater than 4. After multiplying by the left shift, the change will be +6, the required skipping of values outside the BCD format. The algorithm is called Double Dabble.

	0	5	0	6	0	7	0	8	0	9
BCD before correction	0000 0101		0000 0110		0000 0111		0000 1000		0000 1001	
Temporary values after +3	0000 1000		0000 1001		0000 1010		0000 1011		0000 1100	
BCD after shift left	**0001** 000φ		**0001** 001φ		**0001** 010φ		**0001** 011φ		**0001** 100φ	
	1	0+φ	1	2+φ	1	4+φ	1	6+φ	1	8+φ

Let's try the algorithm in C. We will use the top three bits of input x, for which no correction will be performed yet, to initialize the variable bcd. However, we only correct the lower two BCD digits, because converting a byte value to BCD has a third digit of at most "0010"=2.

```
int byte2BCD(byte x)
{       int bcd = (x & 0xE0)>>5; // Variable bcd is initialized by upper 3 bits
        for (int ix = 4; ix >= 0; ix--)
        {       if ((bcd & 0xF) >= 5) bcd += 3; // We correct the least significant BCD digit
                if ((bcd & 0xF0) >= 0x50) bcd += 0x30; // Correcting the second BCD digit
                bcd = (bcd << 1) | ((x >> ix) & 1);
                //the complex statement above leads to simple connections in a circuit.
        }
        return bcd;
}
```

Our final version of the C program verifies the circuit algorithm. It is suitable as a program only for computational elements without hardware float point units. It executes slowly on pipeline processors. It's not for them. It simulates circuitry, not some "C" code! It contains numerous branching if statements whose conditions depend on the input data.

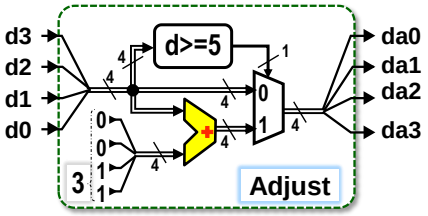
The CPU unit that controls the pipeline run loads the machine instructions ahead. And here, it does not guess what will follow the execution of the if statement, which will be executed until sometime in the distant nanosecond future. It would randomly choose a single branching path. After a wrong prediction, 20 or more already loaded and preprocessed machine instructions must be flushed. The control unit starts loading new from the actually executed branching. Our last code would slow down the execution considerably.

But the circuits don't mind the branching. All its cases are processed in parallel, and the condition selects their final result.

In the example, we demonstrated the difference between circuit design and programming; each implementation tool wants its natural procedures.

Using the circuit procedures, we will connect the converter from the byte type input to three digits of BCD using the already-known elements.

To correct for +3, we build an Adjust circuit. Insert the adder of the constant 3 ("0011") to the input d, which is a 4-bit BCD digit code. We implement the if condition with a 2:1 MUX multiplexer whose address input is controlled by the d>=5 comparator. When satisfied, d+3 is sent to the output of the da circuit; otherwise, d.[42]



Using Adjust, we plug in the byte2BCD() conversion according to the last C code. We insert the body of its for-loop repeatedly using the inline expansion technique, inserting values 4 to 0 after the index ix. We convert the shifts to the left[43] to mere connections to the next series of Adjust circuits.

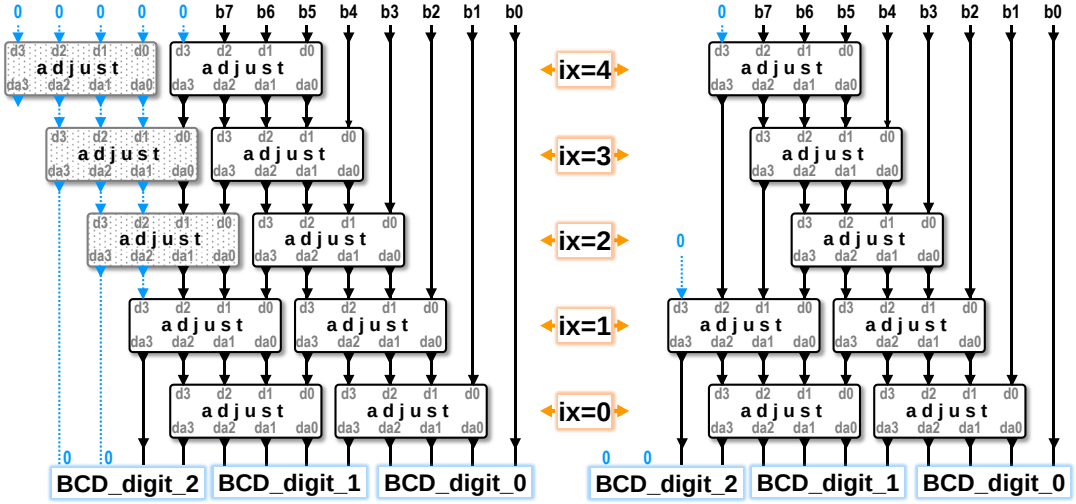


Figure 131 - Byte to BCD conversion

Looking at the computational scheme on the left, we can see that the three Adjust never add 3. Either they have all their inputs zero, or they are receiving at most 2 bits of BCD digits that could differ from 0, so they are only passing their inputs to the outputs.

The design environment has the structure on the left as input. It skips the redundant Adjust and plugs in the final schema shown on the right.

We'll write up the conversion to BCD in our next tutorial, which explains the VHDL style. There, we implement the conversion to BCD of numbers of arbitrary length by both a series of Adjust blocks and a finite state machine (FSM) that mimics the reuse of the for-cycle body by working in a clock. It takes longer to convert, which is fine if we send the output to a display segment. The human eye won't notice that the value appeared a few microseconds later:-)

Let us mention the complexity of the circuits in FPGAs according to the shown algorithms, which extended to the conversion of even longer input numbers than bytes and more BCD digits.

---

[42] We're getting ahead of ourselves, but for the sake of argument, in HDL languages, an Adjust containing a simple 2:1 MUX is described with a single command. In Verilog: assign y = x>=5 ? x+3 : x; In VHDL: y<=x+3 when x>=5 else x;

But we need to understand what we are really creating. After all, HDL, Hardware Description Language, means circuit description. We need to know the circuit first, then specify it with commands:-)

[43] Again, recall that the directions of the shifts are always determined by the weights the bits get after them, not from their orientation on the drawn diagram. The output bits of Adjust, after the shifts, always reach the inputs of the next line on which they have higher positions (weights). Because of this, we speak of shifts to the left.

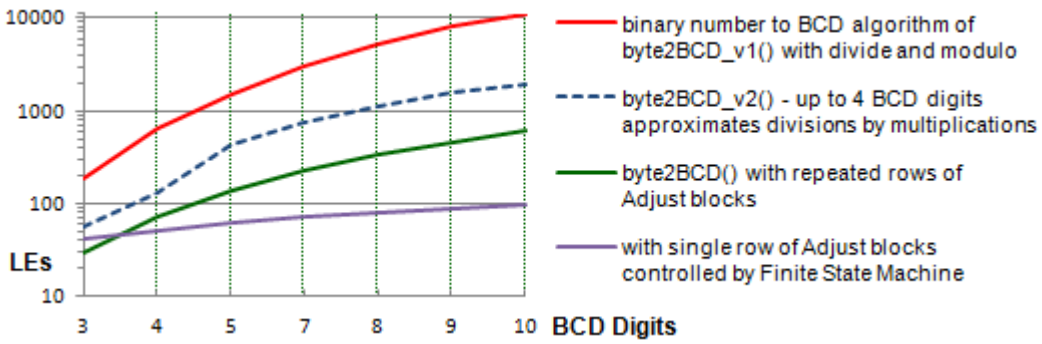


Figure 132 - Complexity of FPGA circuits created by the algorithms shown

The graph shows that the initial programming solution with division and remainder operation is entirely unusable. It is only suitable for large processors, which work better than our other codes.

Approximation of division by one multiplication can only be used in case of inputs up to 14 bits; then we have to choose a more complicated method according to chapter 6.3.5, which causes a jump in complexity. It is not large since multiplication and subtraction still calculate the remainder after division. However, even so, the graph suggests that the algorithm is unsuitable for converting to a circuit.

Our version with the Adjust series has excellent parameters for shorter lengths, but its complexity increases as the input number increases. The finite state machine version more economically converts more extended numbers.

However, we must also include **our time** spent on designing the design to evaluate the quality of the design, as it is also an optimized variable. The most common length of five to seven displayed digits is most easily converted by many series of Adjust. After all, they are directly connected, so the FPGA can simply implement them. The circuit's complexity will be slightly higher than the FSM version.

Unless the saving of consumed elements is necessary for other reasons, then there is no point in creating a finite automaton. An FPGA contains tens of thousands of logic elements.

6.4.2 Task 2: Connect the fast adder to the FPGA

Here, we shrug our shoulders in embarrassment. We can plug in anything to test the function. FPGA circuits can handle real-time numerical solutions of differential equations and other tricks. Still, adders belong to the circuit category that require optimization at the CMOS transistor level to get faster. FPGAs break it down to mere logic functions.

We might as well test the CLA, Carry Lookahead Adder. We know its prediction equations from Chapter 6.1, but the circuit won't be quick. On the Cyclone IV used in our LSP course, the CLA will consume twice as many logic elements and run three times slower than the RCA automatically produced by the design environment. Tested:-)

Although we skillfully apply various tricks of layout prediction in CLA to take advantage of the Carry Chain configuration of logic elements, we outperform FPGA RCA adders to extreme lengths, such as processing 200-bit and more extended numbers[44] .

[44] Hui Li, Zhidong Liang, Hanwen Li, and Yazhou Ye. 2021. A High-Performance Wide FPGA Adder Based on Carry Chains. In Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering (EITCE 2020). https://dl.acm.org/doi/10.1145/3443467.3443868.

Engaging the fastest PPA adder, which is KSA, Kogge-Stone Adder, would provide better results[45] . Its 16-bit version would be only 20% slower than the default RCA but consume four times as many logic elements. Indeed, even with it, the FPGA cannot deploy CMOS tricks, such as our AND-OR gate style, which also dramatically accelerates the KSA tree structure.

However, if the length of the adders increases, our KSA already starts to equal the default RCA. Its 128-bit FPGA variant is even five percent faster but consumes ten times as many logic elements as RCA.

If we need to implement high-speed arithmetic in the circuit, then we choose an FPGA type that supports arithmetic operations in its additional hardware blocks. They are created at the CMOS transistor level using all their capabilities. Many FPGA circuits also include entire processors; for example the introductory Figure 1 on p. 8. Complex arithmetic calculations are then done on them.

Logic elements excel most in implementing parallel cooperating logic operations, not in the acceleration of one of them, which also requires CMOS-level optimization. At the same time, FPGAs can handle only the logic function level.

[45] VHDL description of KSA can be found for example at https://github.com/sehraf/genericKSA.
For a version in Verilog, see: https://github.com/jeremytregunna/ksa

7 Sequential circuits

We introduce sequential circuits by calculating the sum of two six-sided dice.

- If we want to know the sum of two dice, we utilize an adder that is a combinational circuit according to the definition; see Chapter 3 on p. 27. Its output depends solely on its inputs.
- We will need a sequential circuit if we ask for a moving average, for example, of the current and previous dice roll, which we must keep in memory. The result now depends on the sequence's history of two dice rolls.
- We can also extend the moving average to 16 dice throws, which we iteratively solve by a queue (FIFO memory) and a sum register. We subtract the queue head from the sum and add the current dice throw, also appending at the tail of FIFO. The output now depends on a sequence of 16 inputs.
- If we are only interested in the sum of all rolls, then we need much less internal memory, but the sequence will be extended to the beginning of dice experiments.
- The throws' values can also be generated by an autonomous sequential circuit, for example, by a pseudo-random generator based on shift registers with linear feedback, LFSR, which will be discussed in the lectures of our course. The circuit has no X input. It behaves as a mere generator.

We can illustrate the example with the circuit below. The input X will be a composite of the values of two dice rolls with a length of n=6 bits, i.e., two 3-bit numbers. Compared to the combinational variant, which only adds, the sequential circuit needs extra memory elements.

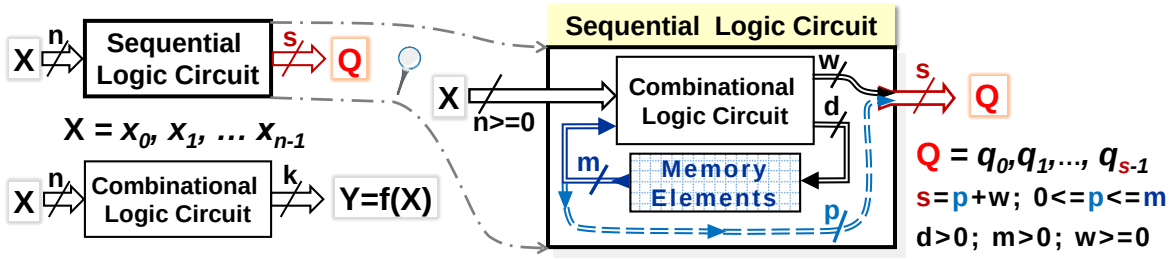


Figure 133 - Example of a sequential circuit

It also contains a combinational logic, but its inputs are a composition of two components, only one of which we can see from the outside; namely our input X. The other is retrieved from internal memory elements.

The combinational part inside our sequential circuit produces internal outputs, some of which are used to update the memory elements, which include the sum register. Others may be sent to the Q output, such as a moving average or possibly information about memory status, e.g., FIFO head has a non-zero value; hence, the queue is all full, and the result is relevant.

We can also output some values of memory elements to the external output Q if we are interested in them. The schematic shown in the figure above is nearly universal; it will suit the most sequential circuits.

7.1 Terminology of sequential circuits

We now summarize the terms used in sequential circuits. Some of them are very established, but others are not.

Clock signal

Any signal can be selected as a clock. Usually, we use some periodic signal, but generally, we can choose any. The choice is ours. Periodic signal has the following parameters:

- **Period** is the time between repetitions.
- **Duty-cycle** is the percentage of the period for which the signal remains in the '1' state.
  It is also rarely specified as a time ratio of higher and lower levels, where 1:1 corresponds to 50% of the crumb or filling factor.

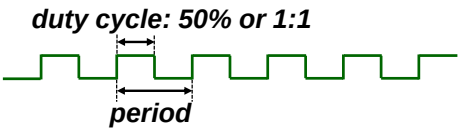


Figure 134 - Clock sequence, duty cycle

Synchronous and asynchronous to the clock

The other signal can be either synchronous or asynchronous in relation to the clock.

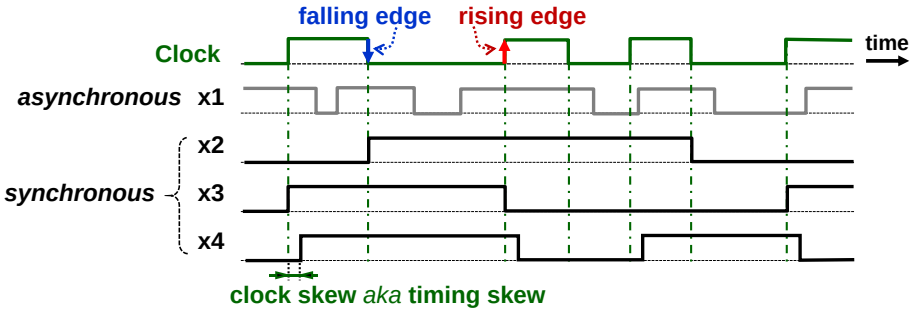
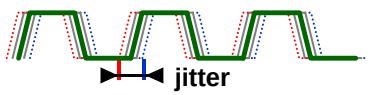


Figure 135 - Synchronous and asynchronous

In the picture above, we deliberately used a non-periodic clock, a less common but permissible choice. Its source can be the output of another synchronous circuit.

- The x1 signal is asynchronous to the Clock as it changes independently.
- The x2 signal is synchronous with the falling edge of the Clock. If x2 changes, it will happen at moments near the transition of Clock from '1' to '0'.
- The x3 signal is synchronous with the rising edge of the Clock signal when Clock '0'→'1'.
- The x4 signal is also synchronous with the rising edge of the Clock but with a deviation called **clock skew**, alternatively timing **skew**. If x4 is delayed, then the clock skew has a positive value. When x4 precedes the clock due to various delays in its distribution paths, then its clock skew is negative.

A periodic signal can also have reduced quality due to a jitter effect. It shows a random phase shift around the exact period.



The jitter frequently occurs when communicating with an external device. Its random nature distinguishes it from clock skew, which, on the contrary, has a relatively constant value.

Naming sequential circuits

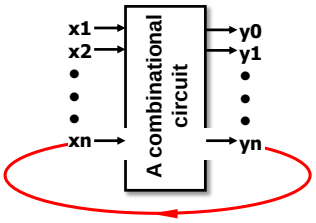
English has two terms, namely latch and flip-flop. The former refers to a latch on a door and dates back to the days of relay technology when a type of memory relay was also made on a similar mechanical principle, known in this country as a "latch relay". The second term flip-flop means a backward flip, a sharp 180-degree turn. The same name is given to flip-flop shoes from the sound they make when walking.

We list the English terminology according to its use in circuit development tools:

- Latch defines only asynchronous latch circuits, such as RS latch and D-latch.
- Transparent latch is a precise technical term, it is more often written as D-latch instead.
- Edge-triggered latch includes a whole category of latch circuits that change their output only when an edge of a clock signal arrives.
- Flip-flop - defines the most common type of edge-triggered latch implementation; see below. Its established abbreviation is DFF, Data Flip-Flop.

7.2 RS Latch circuit

If we create a latch from FPGA logic elements, we always make a **severe error in** our design. The compiler will report it with a warning, either "combinational loops" or "inferring latch(es)..".



RS latch is easy to create if we inadvertently connect some output of the combinational circuit to its input. A loop, called feedback in other scientific fields, exhibits a memory character.

Note: Connecting outputs to inputs is an error only in purely combinational parts. It is commonly used in synchronous circuits, but they behave as memory elements themselves, so we cannot force them to store the output value by closing a loop.

If we write by concurrent commands, they better guard the proper structure, and we create a loop only by a gross error. In behavioral style, loops are a common mistake of beginners. So we need to know what we've done to avoid it next time.

If the loop closes over an odd number of inverters, it wildly oscillates because it does not have a stable state.

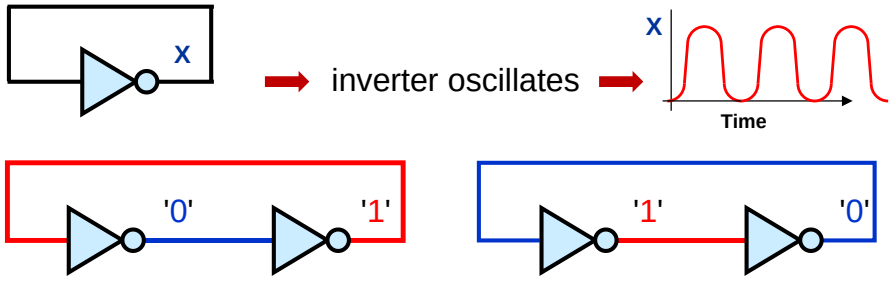
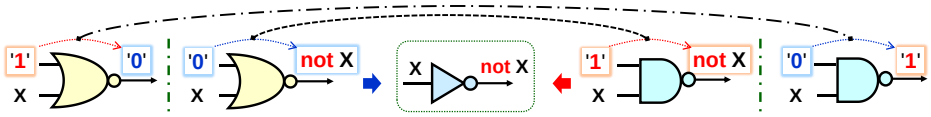


Figure 136 - Inverter loop

However, if a loop runs through an even number of inversions, it is suitable for memorizing information since it has two stable states. We have drawn it as two inverters connected in series since the buffer gate is assembled from them; see Chapter 4.3 on p. 60.

We can change loop values by replacing the inverters with NOR or NAND gates, using annulment and identity laws[46] neutral logic input values (p. 17).



First, we show the RS latch variant, in which NOR gates replace the inverters:

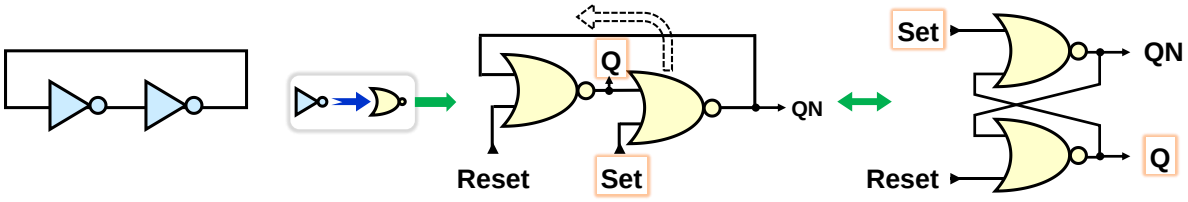


Figure 137 - RS latch of NOR gates

The diagram is often drawn in the arrangement on the right, in which the second gate is graphically moved above the first. The main output of the circuit is called Q, and the one opposite it is called QN, whose suffix does suggest negation of Q. We show later that it can be Q negation, but not always. We named the accessible inputs Set and Reset[47] :

- **Input Set** - If equal to an aggressive '1' for the NOR gate, then its output QN is set to '0'. The upper gate will have '1' on its inputs, and its Q output will go to '0'.
  Note: The naming Set the input of NOR gate with Q output is frequent mistake in written exams.
- **The Reset input** will analogously set QN to '1', changing Q to '0'.

The more common RS latches use NAND gates. However, these have logic '0' as their annulment input values. Thier inputs are often referred to as negated, or inverters are added directly in front of them which is suitable for the usage in other derived circuits.

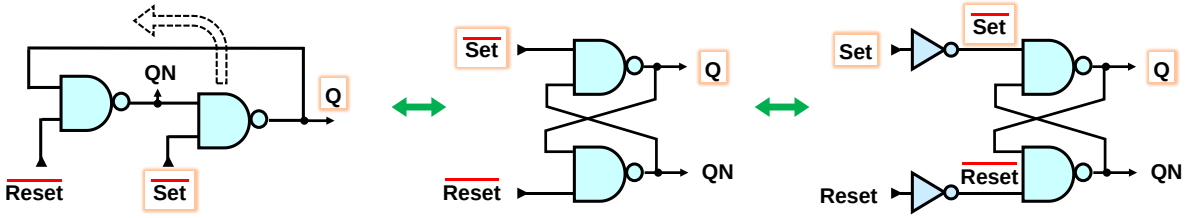


Figure 138 - RS latch from NAND gate

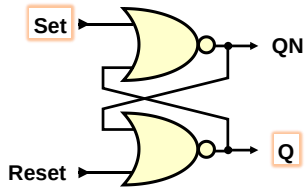
Both versions show similar behavior, but not in all cases, as their logical equations will immediately show. We will construct them using the procedure explained in detail in Chapter 2.4 on p. 25. We express the output Q as a logic function:

$$Q=fn(Set, Reset, Q),$$

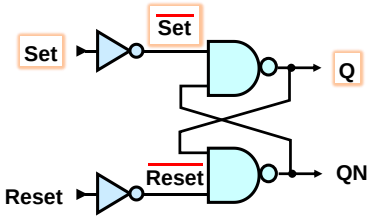
[46] *Local language note*: For unknown reasons, Czech terminology activists renamed annulment and identity laws to aggressive and neutral laws.

[47] *General language note:* RS latch inputs are commonly referred to as Set and Reset in English literature. These are traditional names. However, the word "set" is a broad polysemantic term in English. The Merrian-Webster dictionary lists 16 different meanings for it as a verb, 11 as an adjective, and 47 as a noun. For setting to '1', other circuits (for a function quite identical to Set) prefer the name **Preset**, and the Reset operation, i.e. clearing to '0', is referred to as **Clear**, as these are more unambiguous terms. We can also see in them circuit synonyms to the terms Set and Reset.

The Q output is located on its left side as the output of the function and on the right side as its input, which means the loop mentioned earlier. In the circuit, the value reading is always realized by wire connection from the output to the input because there is no other way.



Q =not (Reset or QN)
=not (Reset or not (Set or Q))¨
=not Reset and (Set or Q)



Q =not (not Set and QN)
=not (not Set and not (not Reset and Q))
=Set or (not Reset and Q)

Figure 139 - RS-latch logic equations

We will also construct their truth tables for both their outputs Q and QN by adding values after the inputs to better see the differences.

NOR RS latch

inputs		*outputs*	
Set	**Reset**	**Q**	**QN**
0	**0**	***memory***	
0	**1**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**0**

NAND RS latch

inputs		*outputs*	
Set	**Reset**	**Q**	**QN**
0	**0**	***memory***	
0	**1**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**1**	**1**

Figure 140 - RS-latch truth tables

The two variants agree on three essential lines:

- In the memory state marked as memory, the Q and QN outputs hold their last values, either Q='1' and QN='0', or Q='0' and QN='1'.
- In the reset state, the output Q is set to '0' and the opposite QN to '1'.
- When setting Q='1', while QN='0'.

If the Set and Reset inputs are both in logic '1', then we see different behavior:

- NOR version of RS latch has both Q and QN in logical '0'. We can, therefore, say that its Reset input has a higher priority than Set, as shown by the logic equations.
- In contrast, the NAND variant of the RS latch favors Set and sets both Q and QN to logic '1' in the same situation.

But the Set='1' and Reset='1' state is forbidden!

- No, the gates have no prohibited inputs!
  The output of a NOR gate goes to logic '0' if any of its inputs are '1', and a NAND gate goes to '1' if any of its inputs are '0'. We indeed don't forbid them to do that! After all, this is a primary requirement for their operation.
- **The forbidden state is created only by adding the constraint QN = not Q.** We obtained the prohibited state only in the RS steady state concerning this assertion.
  The so-called "forbidden state" is mainly the **working state of** the RS latch, which would not flip without it, which we demonstrate by analyzing its dynamic behavior, see the following figure.

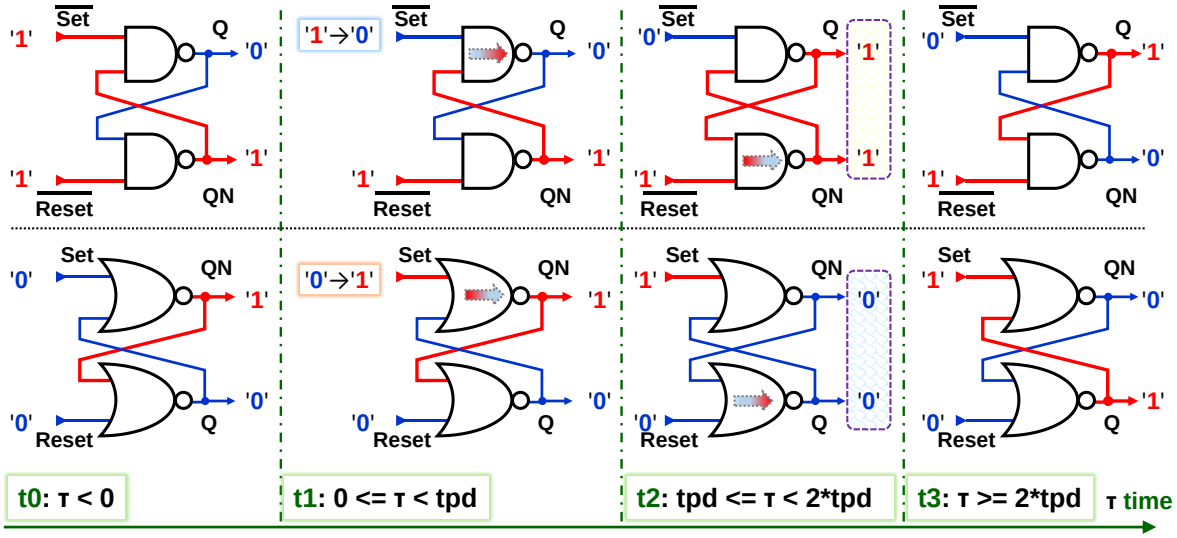


Figure 141 - RS latch dynamic behavior

Let τ denote the relative time when the Set input was changed. Then, in times:

t0: τ<0, we assume that both RS latch circuits now have their inputs such that they remember. Their outputs retain their previous values, Q='0' and QN='1' in the figure.

t1: At time τ=0, the Set inputs of both RS latch circuits have changed to values that cause their output Q to be '1'. However, the time elapsed is still shorter than the propagation delay of the gate, **tpd**, i.e., τ<tpd. The output of the upper gate has not yet changed.

t2: Time τ>=tpd elapsed, so the upper gate changes its output. Now, the lower gate is waiting. **Both RS now temporarily have their outputs Q = QN**.

t3: Only after time τ>=2*tpd, the second gate output is also affected by the change, and both RS latches reach their new steady states.

7.2.1 Metastability

In the same initial situation as above, let the Set inputs receive a pulse with a polarity suitable for setting Q='1', but with a length longer than tpd but shorter than 2*tpd, so only their upper gates manage to flip. The circuit has reached its normal operating state, where Q=QN for a temporary instance. The situation is shown at the beginning of the part marked m∞

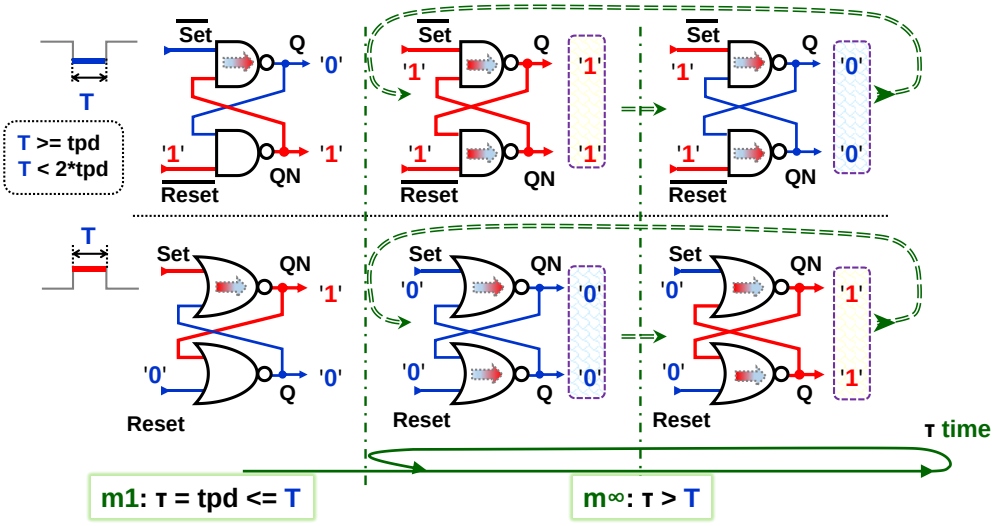


Figure 142 - Metastability of RS latch

However, there is no indication to the latch where to proceed next, whether to the outputs

Q='0' and QN='1', or vice versa to Q='1' and QN='0', as the pulse has already ended and the Set and Reset inputs have returned to the values with which the RS latch remembers its last state. Thus, all gates have their input values flipped to their opposite values, but in turn, those will have one of their inputs at an annulment value, forcing them to return to their previous states. And from them back again. The outputs are racing.

Theoretically, they will flip forever, but practically not. After some time, they will settle down to either '0' and '1' or '1' and '0' due to circumferential minor asymmetries. However, we don't know its final state. And their oscillations load the power supply and generate disturbances at the earth connections; see the water model on page 66.

However, the gates do not oscillate up to '0' and '1', but near the metastable equilibrium state. If we draw the voltage waveforms of the inverters in a loop, see 4.9 pg. 73, we can combine them into a single graph (left side of the figure). The waveforms intersect at the point where their input and output voltages balance. The loop is in equilibrium.

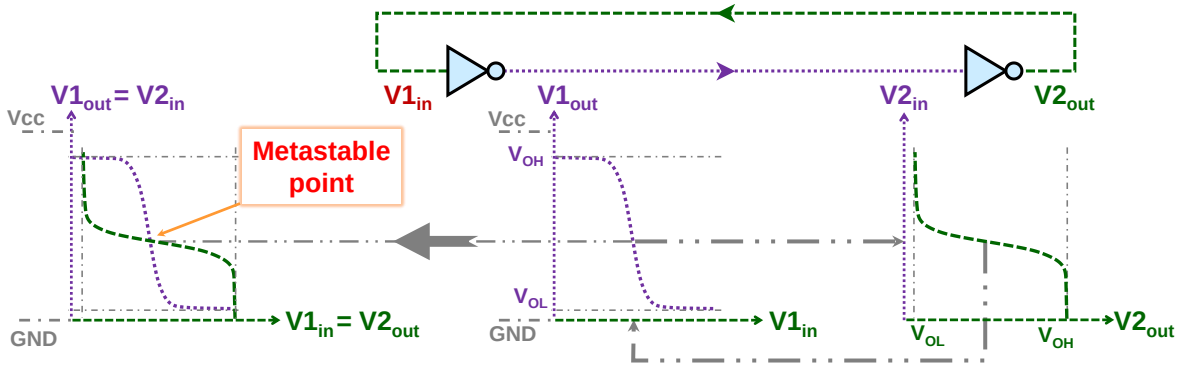
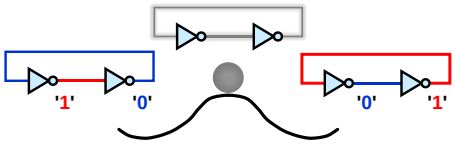
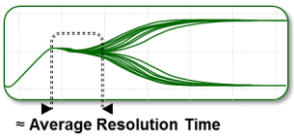


Figure 143 - Metastable point

However, it is stuck in a metastable point, often depicted as a ball on top of a round hill. If we balance it well there, it won't fall off the top, at least for a while. A slight impulse, however, and it will roll left or right. We don't know in advance where or when.





The time remaining in a metastable state is referred to as resolution time. It depends on the CMOS technology. For those used today, its average value can be expected to be in the order of picoseconds. Sometimes, it can be measured with an oscilloscope if we trigger repeated occurrences. The waveforms will show that the output oscillates around the equilibrium point for a while. It can also be estimated from the increase in delay time[48] , which is just the decision time.

In catalogs, manufacturers usually only list MTBF, the Mean Time Between Failures, a statistical parameter calculated from it. Its explanation is beyond the scope of our textbook. It is described at ASIC publication[49] .

In the FPGA, our RS latch will not consist of gates but of logic elements. They also create all the combinational logic controlling the Set and Reset inputs. Various delays on the internal

[48] B. Medved Rogina, P. Skoda, K. Skala, I. Michieli, M. Vlah and S. Marijan, "Metastability testing at FPGA circuit design using propagation time characterization," 2010 East-West Design & Test Symposium (EWDTS), St. Petersburg, Russia, 2010, pp. 80-85, doi: 10.1109/EWDTS.2010.5742050..

[49] Jakub Šťastný: Techniques of synchronization of asynchronous signals, ASIC Centrum.cz, 2023.

paths can generate glitches, i.e., very short interfering pulses; see Chapter 4.10.1 on p. 75. The RS latch will respond to these by randomly changing both its outputs and its delay at metastability in which its output voltage is held near the center of the supply, a level easily affected by the induction of noise generated by other sources on both the '0' and '1' level sequences.

If we want a reliable circuit, we need to reduce the risk of metastability in it, which leads to a principle emphasized in almost every technical publication devoted to FPGA design.

We must not use logical elements to create circuits of type LATCH!

7.2.2 D-latch from gates

The D-latch circuit eliminates some of the problems of the RS latch. Let's look at its structure since it will be created from logic elements more often than RS latch in our wrong designs. Knowing what we've made and need to change our code immediately helps us remove this problematic element from the circuit.

We're coming out of the RS latch. The Reset input is still derived by the inverter from Set, which we rename to D (Data). We'll replace both input inverters with NAND gates that will share another input, ENA (Enable), allowing latching.

Note: ENA is also abbreviated to En or E; respectively, T is used. But it is not appropriate to mark it as C or CLK, established clock inputs of synchronous circuits, which D latch is not yet.

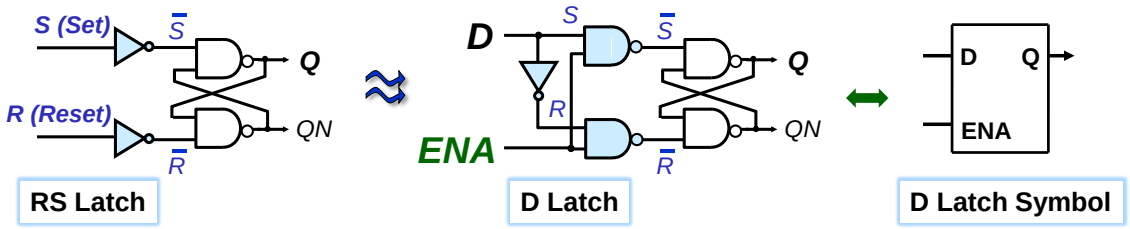


Figure 144 - D latch

Let's draw its functional analogies, which we derive by connecting logical '0' and '1' to the ENA input.

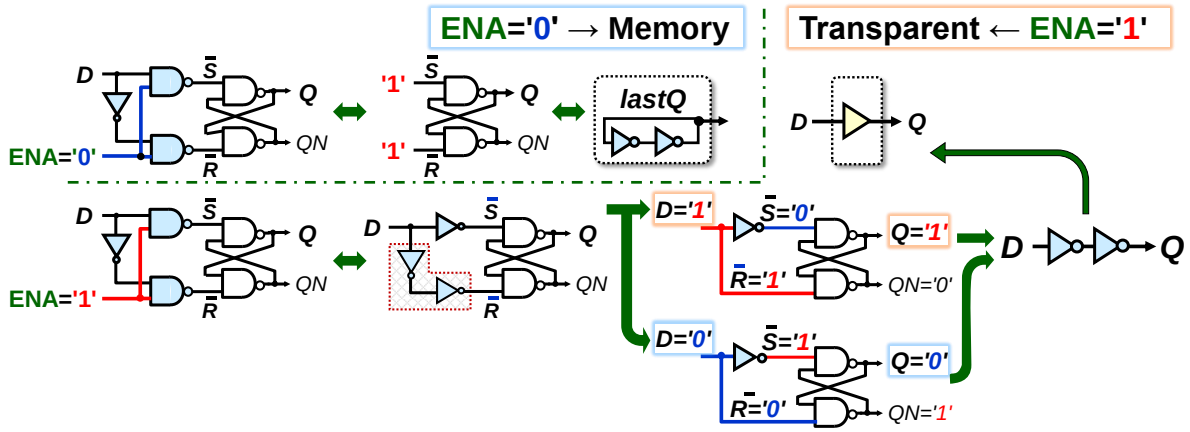


Figure 145 - Behaviour of D-latch in envy on ENA

The D-latch circuit switches between a pair of its functions with the ENA input value:

- When ENA='1', the input NAND gates behave as inverters of D. Two in a series cancel each other according to the double negation theorem (see p. 18). After substituting '0' and '1' to D, we see that the value of D is copied to the output of Q with a delay of roughly two in-

verters. We replace them with a buffer gate. The D-latch now behaves like a **transparent** circuit[50] , and the name of the group of transparent latches to it D-latch belongs.

- When ENA='0', the internal loop of the inverters is disconnected from input D, which loses its influence on the output. The RS latch **remembers** its last value, passing it to Q output.

We use the switching between the two modes to construct the waveforms. For clarity, we will ignore the propagation delay between input D and output Q.

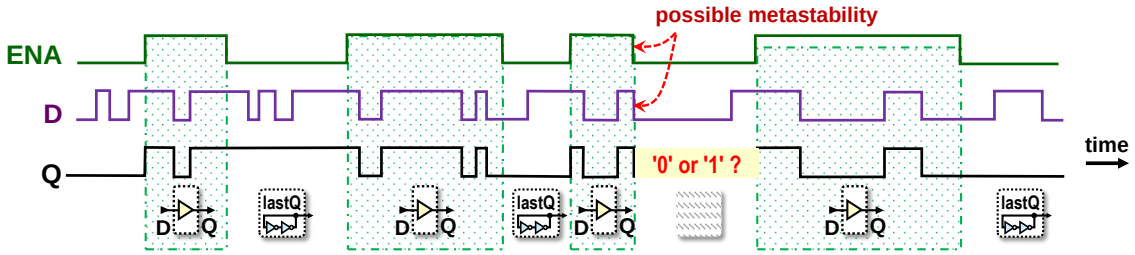


Figure 146 - D latch behavior

We can save the inverter in the D-latch front part by group minimization; see Chapter 5.2.1 on p. 81. We leave to a reader the proof of the functional equivalency of both variants.

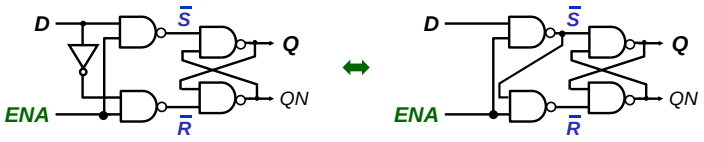
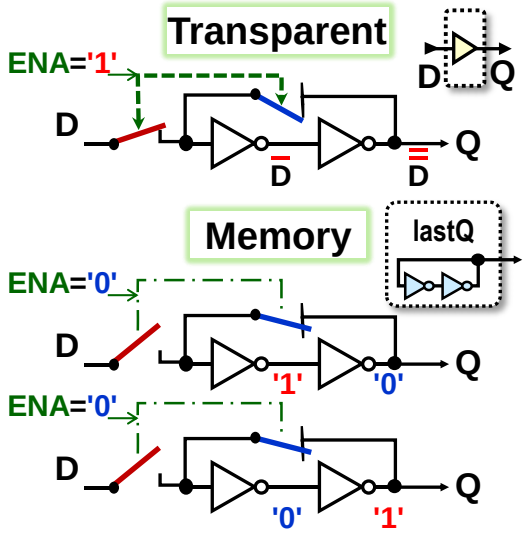


Figure 147 - Two functionally identical versions of the D latch

7.3 D latch at CMOS level

The D-latch version implemented at the CMOS level is the primary building block of the most common flip-flop synchronous circuits, and its characteristics are reflected in them. We have to consider them in our designs. The CMOS versions switch between closed and open loops, i.e., between memory and transparency. They use transmission gates (see p. 63), which work in counter-phase.



When ENA='1', the loop is disconnected, and the value from D input passes transparently to output Q through two inverters connected in series, i.e., through the equivalent of a buffer gate.

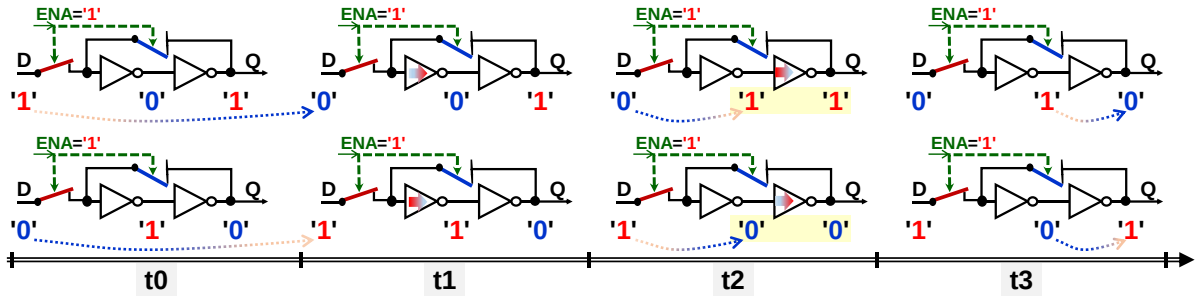
When the falling edge of ENA arrives, i.e., its change from '1'→'0', the loop disconnects from input D and closes. It now remains in its last state.

Output Q will be permanently either '0' or '1' for as long as ENA='0'. Once ENA goes to '1' again, the process repeats.

Figure 148 - D-latch - Transparent and Memory Mode

[50] Somewhere the D-latch is alternatively called a gated latch, because its flipping is blocked by NAND gates.

We will study the open loop behaviour, i.e., when ENA='1'. Let tpd be the delay of one inverter, then the change of input D at time t0 will propagate gradually. At times t0 to t3, the following events will occur:



- **t0** - just before D change, the open loop has a steady state;
- 0<t1<tpd - although the input D has changed its value to the opposite one, the new value is still propagating through the left inverter. It will arrive at its output after tpd.
- tpd<=t2<=2*tpd - the left inverter has already flipped. The loop is in a temporary working intermediate state where both its inverters have identical output values, they are either at '0' or '1'. Now, the loop must definitely not close by changing ENA to '0' to avoid metastability.
- t3>=2*tpd - the loop has already reached a steady state, so the ENA closing edge, '1'→'0', is allowed, since ENA='0' already correctly switches it to the memory configuration.

The CMOS D-latch again has a risk area around the closing edge of the ENA, where working interstices threatened metastability. We establish two necessary conditions with the specific times depending on the technology. The manufacturers present them in the documentation.

1. The ENA pulse must last for at least t_min to give the loop time to settle.
2. In the vicinity of the ENA falling edge, input D must not change within the relative interval defined by the **setup** and **hold** times. They are not equal. Setup time can sometimes have a negative value if it takes a while to activate loop closure, then input D is allowed to change even after ENA goes from '1' to '0' for some picosecond period.

The following figure shows the conditions.

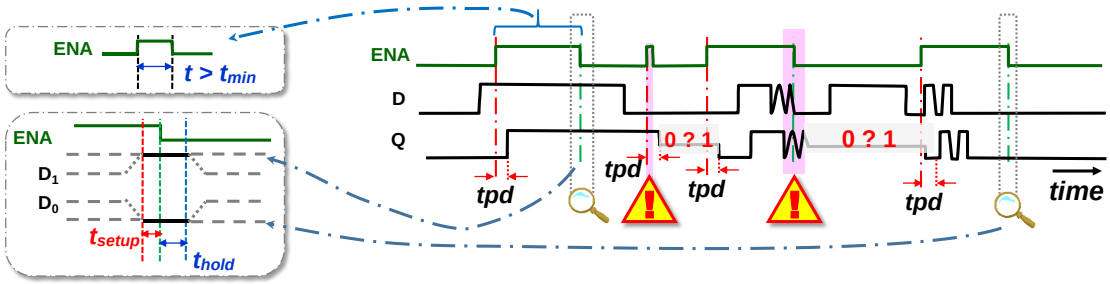


Figure 149 Timing conditions and the consequence of violating them

The risk of metastability arises from the loop principle itself and cannot be eliminated, only prevented by observing the t_setup, t_hold and t_min times according to the manufacturers' catalogs.

The switches are implemented by the transmission gates at circuit level, leading to the following overview of the CMOS D-latch circuitry:

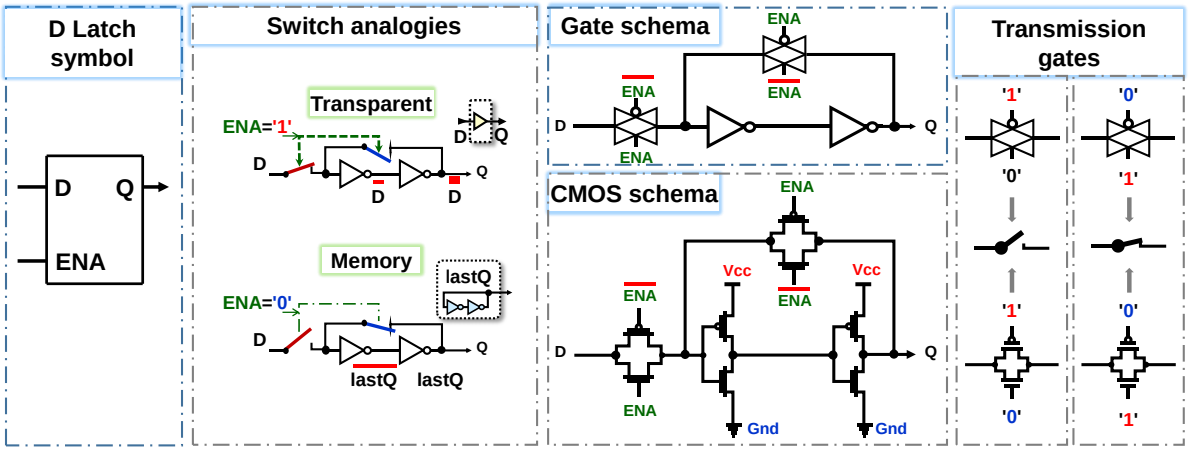


Figure 150 - D-Latch Wiring Diagram

7.4 Flip-Flop Circuit DFF - Data Flip-Flop

DFF, Data Flip-Flop, samples its D input with the edge of the clock signal. It no longer has a transparent D-latch mode, which at ENA='1' transferred everything from its input to its output.

There are several DFF structures. For interest, we present one in the figure on the right, taken from Wikimedia. It is called the Earle latch after its author.

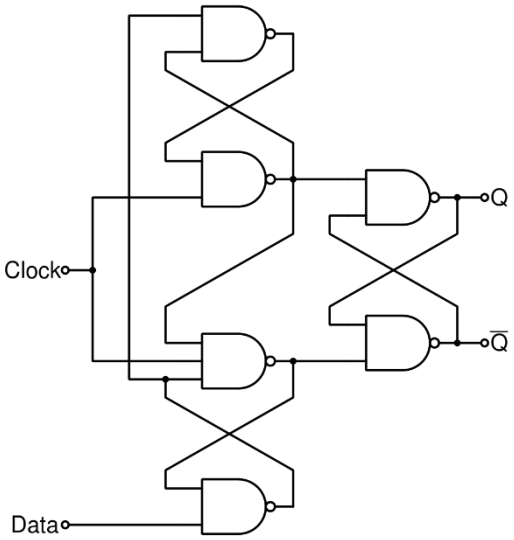


Figure 151 - DFF structures Earle Latch

Internally, it consists of three RS latches, with one of the two input RSs always held in the logic '1' state of both its outputs, which is inaccurately called forbidden or prohibited.

The circuit was used for a long time in TTL logic with the code name integrated circuit 74 by many manufacturers, including the Czech company Tesla as type MH7474.

It offers numerous advantages but needs 6 NAND gates. Because of this, more economical implementation is preferred today.

The vast majority of DFFs in use sample the input either always only on the rising edge of the clock or always only on the falling edge.

The most widely used DFF, created by cascading two D-latches working in counter-phase, also flips only on the rising edge of the clock. For half a century, its structure was called by the established term Master-Slave, which is now considered socially unacceptable. The designers have thus lost the unambiguous natation. When writing the textbook, the replacement name has not yet been settled[51] . In the text, we label D-latch with **Primary** and **Replica**.

A pair of CLK clock inverters controls the ENA inputs of both D-latches. The first one isolates the circuit and reduces the load of the clock signal distribution, its fan-out.

[51] An overview of the proposed replacement names can be found at https://en.wikipedia.org/wiki/Master/slave_(technology)

Suppose the CLK clock starts at '1'. The Primary D-latch has its ENA='0', so its loop remains in memory mode, holding its last value. The ENA input in polarity identical to CLK drives the Replica D-latch that transparently copies the Primary D-latch output to the Q output of the DFF circuit because its internal loop is opened; see the following figure.

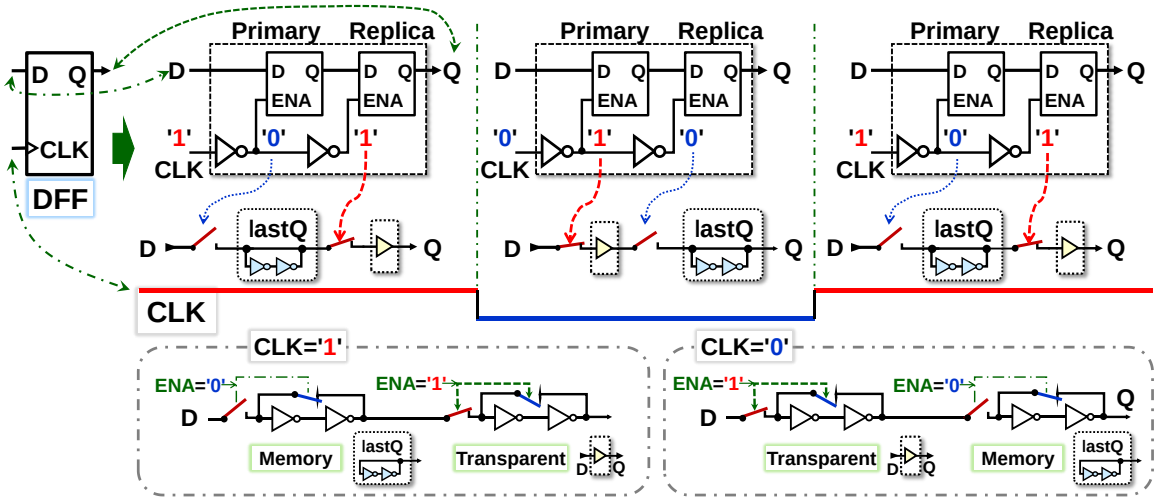


Figure 152 - DFF principle

If CLK goes to '0', the Primary D-latch connects to the D input of the DFF circuit, but at the same time, the Replica D-latch detaches from it and goes into memory mode. It still sends its last state to the Q output of the DFF circuit, the value last copied from the Primary D-latch.

At the rising edge, i.e., at the transition CLK from '0'→'1', the Primary D-latch disconnects from the D input of the DFF circuit and holds its last value in its loop. However, this is now copied to the output by Replica D-Latch with an opened loop, so it is in transparent mode.

Its output Q manifests as a sampling of the D input value with the rising edge of CLK.

Beware, metastability can occur at the rising edge of CLK if the Primary D-latch loop has failed to stabilize because the setup and hold times of the D input have not been met. They varied around the rising edge of CLK when the Primary D-latch has reached the falling edge of its ENA.

Note: The falling edges of CLK do not need timing constraints, although Replica D-Latch closes its loop during it. However, it is steady; before closing, it only copied the stable output of the Primary D-latch, which was in memory mode at the time.

The coupling between the DFF stages used by the principle scheme above would delay the output of Q by four inverters. Because of this, the two D-latches connect through their centers. The Primary latch sends the negated output, but the Replica latch inverts it again, so the result gets the correct polarity.

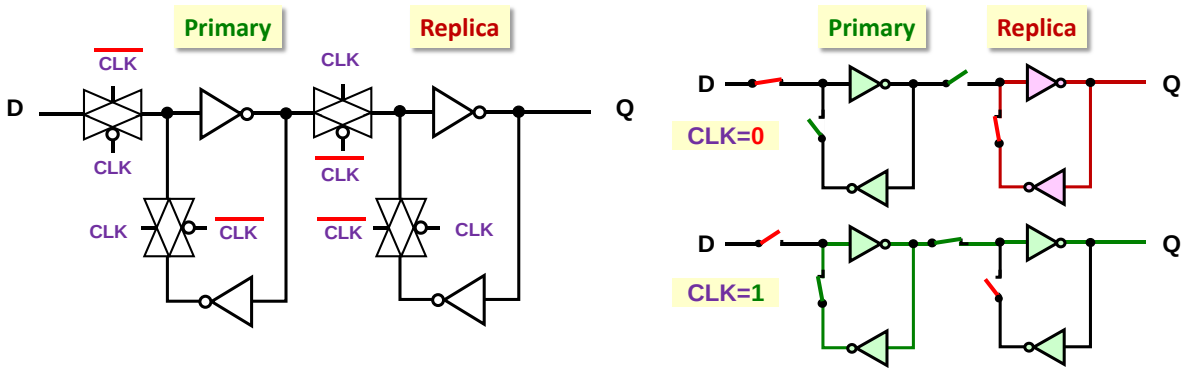


Figure 153 - Actual linking of Primary and Replica

The operation of the DFF is best outlined in the following figure. Together with it, we will indicate how the same inputs would act on the D latch. Its different outputs are highlighted. Moreover, D latch is slower. In DFF, at the rising edge of the clock, the value stored in the Primary D latch is propagated through a single inverter in the transparent Replica D latch to the output Q. In contrast, the standalone D latch switches its ENA to transparent mode in the same situation, and the new value propagates from its D input through two inverters.

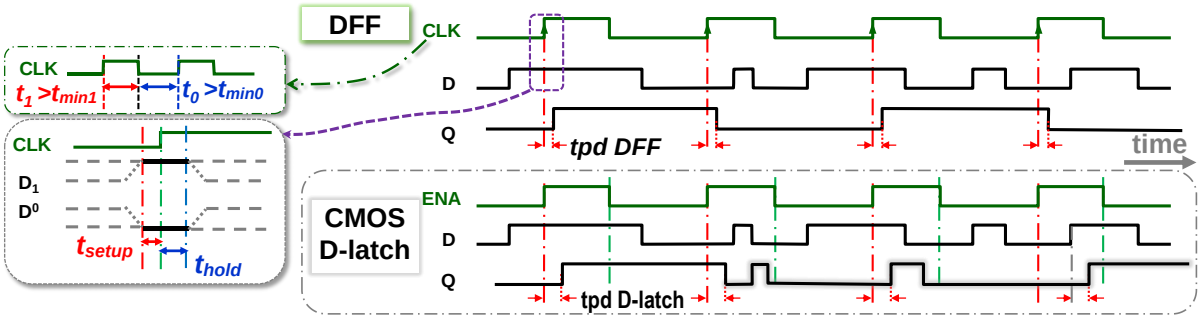


Figure 154 - Comparison of D-Latch and DFF behaviour

In diagrams, the input of the DFF clock is often indicated by a triangle pointing to a mark at the rising edge. Conversely, it heads outward when the DFF responds to the falling edge. Standardized markings are not often followed, but it is a well-established practice to assign the abbreviation CLK or CLOCK, or at least C, to the clock input. The name of its input is sometimes omitted. The triangle mark specifies it.

By inverting the polarity of the clock, a circuit sensitive to the rising edge can be made sensitive to the falling edge, or vice versa. Just add an inverter before the clock input as shown in the picture below.

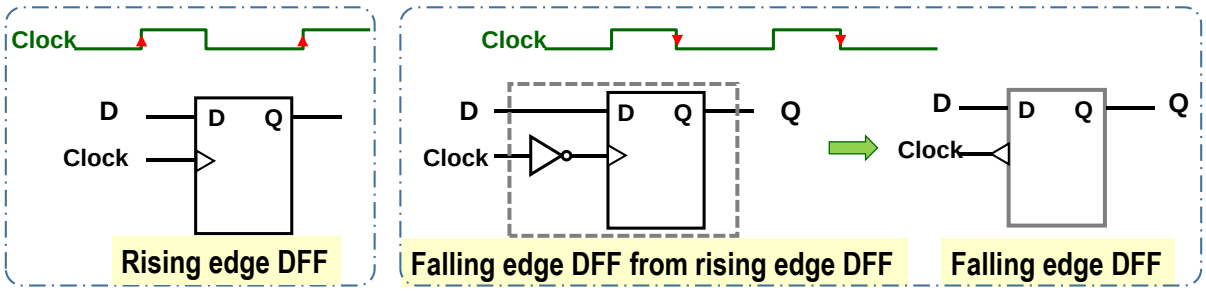


Figure 155 - DFF marker for leading/trailing edge sensitive DFF

There are also DET DFF (Dual Edge Triggered Data Flip-Flop) flip-flop circuits that are sensitive to both rising and falling edges of the clock but have higher complexity. Most FPGAs offer the user only a single-edge-sensitive DFF with a Primary-Replica internal layout, which comes out the simplest at the CMOS level.

> **The DFF output has no hazards.** After all, it is formed by inverter loops in which they do not arise. We can use it as a **clock of other circuits** if we keep the timing conditions of its input signals, thus eliminating the metastability of its Primary loop.

But we have to be very careful about crossing **clock domains**. These refer to a group of circuits that are synchronized by one clock. Signal transmission from one clock domain to another is at risk of metastability and requires careful design. We present an example of a possible solution in Section 7.4.2 on p. 136.

7.4.1 Addition of DFF with Enable and asynchronous zeroing

Combinational logic must not be inserted into the clock distribution, as we mentioned in Chapter 4.10.1 on p. 75. The inverter and buffer gates are the only exceptions. A DFF changes its state only on the rising edge of the clock. How do we ensure it doesn't flip when we don't want it to? It acts like a memory and needs initialization when power is applied, i.e., in the power-up phase. These are two necessary improvements:

1. We need to suppress DFF's flipping without blocking his clock.
2. An asynchronous input is helpful to hold the DFF loops in a known state during the entire time when the crystal oscillators are just coming up after power-up and so are not yet generating clock signals, which can sometimes take tens of milliseconds.

The first improvements are easy to accomplish. If we want the DFF to keep its output Q unchanged, even though it constantly receives clocks, we connect its output Q to input D. The DFF will record its value, which it already has. The loop inverters keep their original states, so they do not claim dynamic energy consumption; see Chapter 4.8 starting on p. 66.

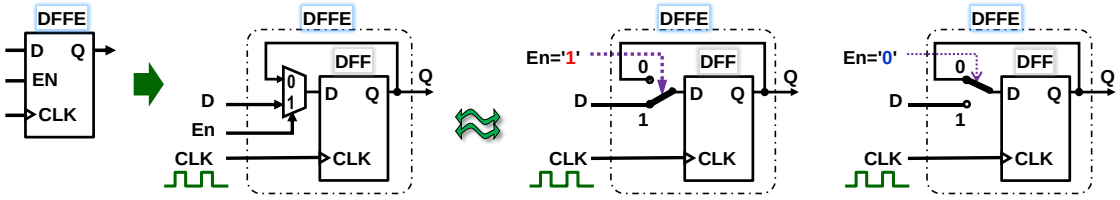


Figure 156 - DFFE flap circuit

The new circuit is called DFFE, DFF with Enable, and works identically to DFF when EN='1'. At EN='0', it records its output Q value. The versatility has been increased at the cost of adding a multiplexer, so DFFEs tend to be a standard part of all manufacturers' FPGAs.

> The EN multiplexer signal selects which value to connect to the D input of the internal DFF, so it must also not change around the active edge of the clock signal within the interval defined by the setup and hold times from the manufacturer's catalog.

An input with asynchronous behavior is added to initialize after power-up. It affects the Q output immediately and independently of the CLK clock. It is referred to as CLR, Clear, and sometimes has a suffix N, i.e. **CLRN**, to emphasize that it is active at '0'. The most accurate abbreviation would be ACLRN, Asynchronous Clear Negative, which we will prefer. However, the shorter **CLR** is more commonly used in schematics and is often just indicated obly by the position at the bottom of the DFFE symbol, or even at the DFF.

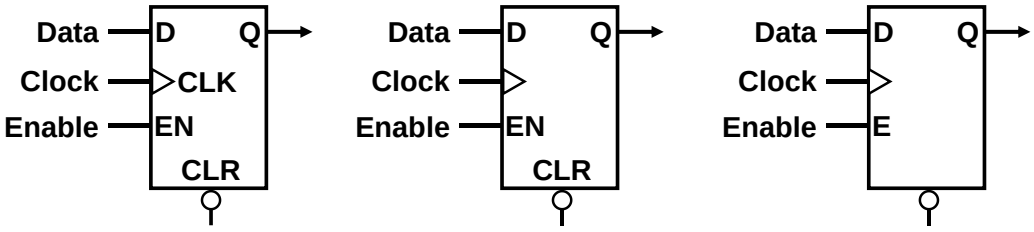


Figure 157 - Some variations of the schematic mark DFFE

Unused ACLRN and EN inputs are connected to '1', which is usually done automatically by the development environment. FPGAs contains built in logic elements to disable them.

Asynchronous clearing is added by replacing one suitable inverter in each of its loops with a NAND gate. These behave as inverters of the second input when ACLRN='1' and do not change the function of the DFF. Under ACLRN='0', the outputs will be logic '1', which sets the internal values in the circuit such that Q='0' regardless of the states of the CLK and EN inputs.

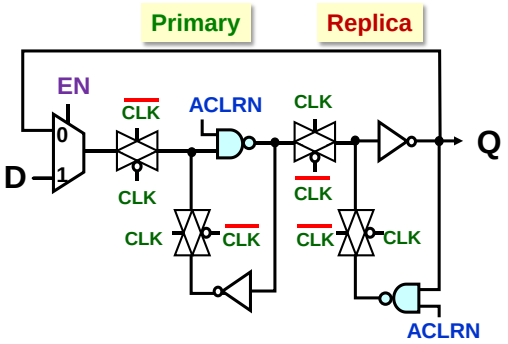


Figure 158 - Asynchronous Zeroing

Most FPGAs have only flip-flop circuits in their logic elements that include asynchronous zeroing. They are set to '1' by negating the output and input. The resulting circuit will then behave as if initialized to '1', but will be slightly more complex.

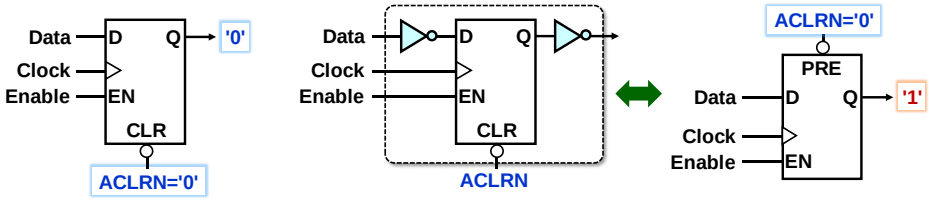


Figure 159 - Changing the initialization of the DFFE circuit

If we look in the manufacturer's catalog and see what all the DFF/E of the logic element offers, then we can prefer to set it to a more natural asynchronous value.

In addition, the FPGA with SRAM usually loads the configuration from the additional Flash memory when power is turned on, and many types state that all flip-flop circuits will be in the default zero state. Modern FPGAs often allow some to specify a default setting of '1' as well.

If we also ask for an emergency reset, all DFF/Es do not need a reset in this case anyway, for example frequency dividers. After some time, they will get themselves to the correct state from anywhere. Replacing asynchronous initializations with synchronous ones is recommended, which have wider possibilities.

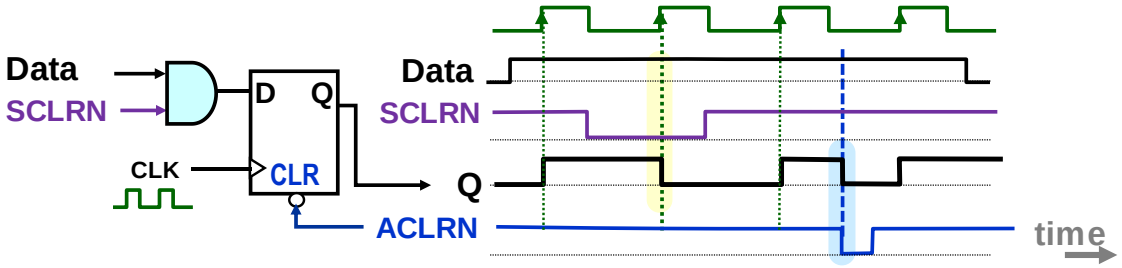


Figure 160 - Synchronous and asynchronous initialization

Synchronous initialization is done by uploading a new value, while asynchronous initialization is done immediately, independent of the clock. We derive the timing conditions on the ACLRN waveform ourselves from the loop behavior of the inverters:

- An asynchronous ACLRN must remain in logic '0' longer than the delay of the two inverters; otherwise, there is a risk that a loop that is currently in memory mode will not settle and remain in a working intermediate state in which its members have identical outputs. It may then become metastable. **The condition precludes the generation of ACLRNs by logic functions** as they may have hazards. In an FPGA, all circuits can gen-

erate them except for invertors and buffers.

- The ACLRN shall change from '0' to '1' around the CLK clock's rising edge if a non-zero input D value is sent to the Primary D-latch. If its inverters failed to stabilize, metastability would again occur.

An example of the function of a DFFE circuit that has an ACLRN is demonstrated in the figure below. The first state of ACLRN='0 held the circuit in its default state during the power-up period and is the correct usage.

The following pulse ACLRN='0 was identified as a time bomb, which would be if a logic function from other internal signals generated it!

Fast loops will also be a problem here. They will arise when the logic function generating ACLRN='0' also depends on the values of the outputs of the circuits that zero it. Once ACLRN affects some outputs and they change values, then it will cancel itself, i.e., ACLRN='1', which will terminate the clearing. And it may not be necessary to catch all the elements to which the ACLRN is distributed:-)

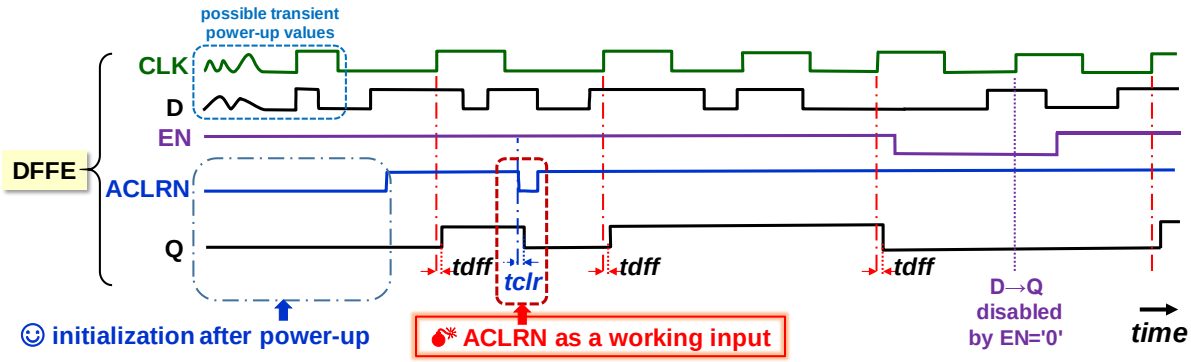


Figure 161 - DFFE flap circuit with asynchronous zeroing

On the web, you can find numerous examples that use synchronous inputs as working and generate clear signals logic functions. Similar schematics refer to slower bipolar TTL logic, which had delays from about 7 ns upwards, and thus lower sensitivity to noise pulses, and could use asynchronous inputs as normal working ones.

The delay of today's gates is measured in tens of picoseconds. They allow designs with asynchronous working inputs but done in extra carefully tuned techniques implemented directly at the CMOS level, in which we eliminate hazards, the sources of short illegal pulses. However, this is generally not recommended.

Asynchronous clearing or setting
is used in FPGAs exclusively to initialize the entire circuit,
which we need, for example, at power-up or for an emergency restart.

We'll add a real story. At the beginning of this millennium, our stubborn student insisted on using asynchronous clears as working inputs in FPGA to solve his credit assignment. He said they always worked reliably. We let him. There is nothing like personal experience.

He carried a circuit board with a functional circuit implemented with slow TTL logic. He experimented for two months with creating its analogy in a fast FPGA before he convinced himself that this was not the way to go. And because of that, he missed the last deadline for his credit thesis. However, he was rewarded for creating an enlightening story with an individual extension without penalty:-)

7.4.2 Synchronizers and ACLRN creation

The DFF needs unchanged D and EN inputs around the rising edge of clocks, which we can ensure for signals we generate ourselves in the circuit but do not influence externally. They are not synchronized with our clock domain, i.e., the circuit group that depends on a single clock. Crossing a domain boundary requires a synchronizer, also called a resynchronization circuit.

The asynchronous ACLRN will always be an external input, either from a pushbutton or RC cell, whose transition to '0' and '1' can occur at any time. If we want to ensure it does not induce metastability by terminating at an inopportune moment, we modify it with a synchronizer. We create it from at least two DFFs in the cascade.

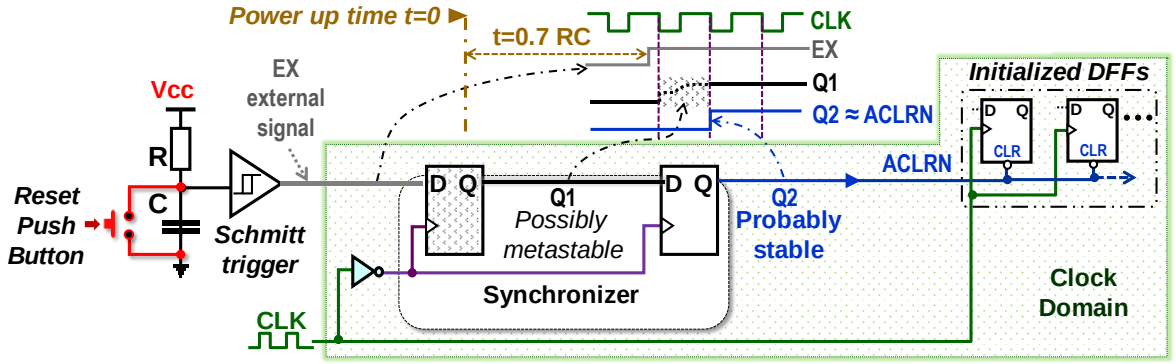


Figure 162 - Synchronizer on clock domain input

To generate the ACLRN, we use a well-known RC cell. Its capacitor C is charged to half of the supply voltage in the time given by the solution of the differential equation, which results in t=0.7 RC. We know the formula from Chapter 4.8.3 on p. 69. It can also serve emergency initialization as a reset button, discharging the capacitor that recharges again.

The Schmitt trigger shapes the slow rise in voltage across the capacitor, see below, which guarantees a clean output '1'.

The EX output exceeds the clock domain boundary, so we route it through the DFF cascade. The first DFF can still sometimes have metastability if EX terminates at an inopportune time. Assume that it recovers to state '1'. It will load a '1' in the next clock stroke if it fails.

The second DFF will most likely already have a clean output. We use it as an ACLRN signal.

Both DFF circuits flips on the clock's falling edge, so ACLRN goes to '1' off the rising edge. We distribute it to all DFFs that respond to it and need initialization.

In the explanation, we also mentioned the Schmitt trigger circuit, which belongs to the commonly produced parts. It acts like a voltage comparator with hysteresis, as it needs a higher input voltage to go to logic '1' than to return from it to logic '0'.

It engages in many different ways, including operational amplifiers[52] . We can show one of its CMOS designs that uses the familiar memory loop inverters.

[52] See for example https://en.wikipedia.org/wiki/Schmitt_trigger

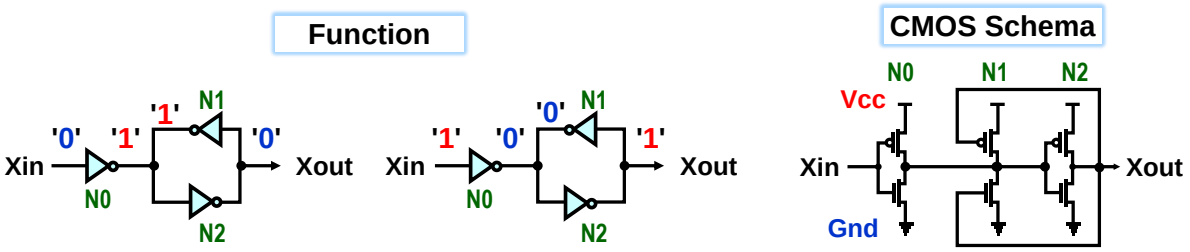


Figure 163 - Example CMOS Schmitt flip-flop circuit

The input inverter N0 switches the loop by short-cutting it, which looks strange, but only at first glance[53] . The invertor loop instantly changes to a new state in which the output of its inverter N1 is identical to the N0 gate output.

Thus, the output of N0 drives the loop together with the N1 gate, causing a shift in the threshold voltages of the CMOS input in N0. These are then increased/decreased by a bit. Now, we need a higher force at the N0 input to balance the massive logic load of its output. It is similar to a "two-arm swing" with a heavier load that we want to rebalance to its opposite state, which N1 and N2 loop will follow.

We obtained a circuit with hysteresis that converts the rippled input signals to clean waveforms and accelerates the Xout output flipping.

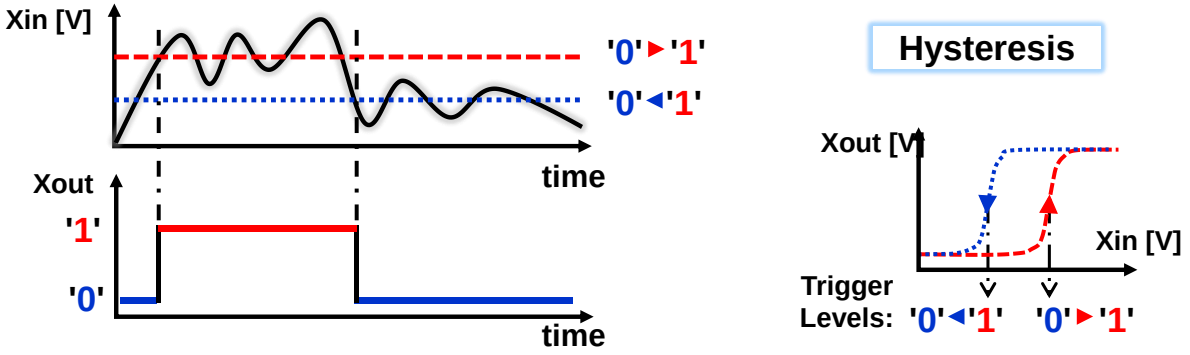


Figure 164 - Example of Schmitt trigger circuit operation

[53] Cells in SRAM memories are also set by a similar loop short circuit.

7.5 Registries and counter

Different circuits can be built from DFF or DFFE circuits. The simplest of these is a register that stores just as many bits in parallel as we design it.

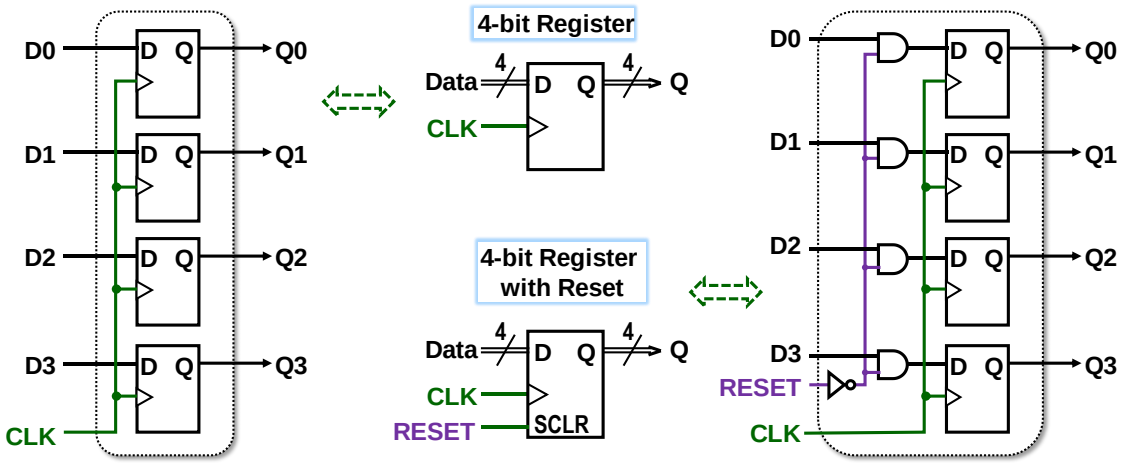


Figure 165 - 4-bit register

We can also add synchronous zeroing, for which the name RESET or SCLR has been established to distinguish it from asynchronous zeroing CLEAR, which could also be added if the asynchronous DFF inputs (omitted in the figure) were connected.

However, as mentioned earlier, asynchronous initialization should only be used in critical parts. *Note: Design environments occasionally convert our asynchronous initializations to synchronous when they find them unnecessary.*

We can create a counter if we connect +1 adder in front of a register. For example, we can build a three-bit counter, which we used in the flashing snake, Figure 86 on page 81.

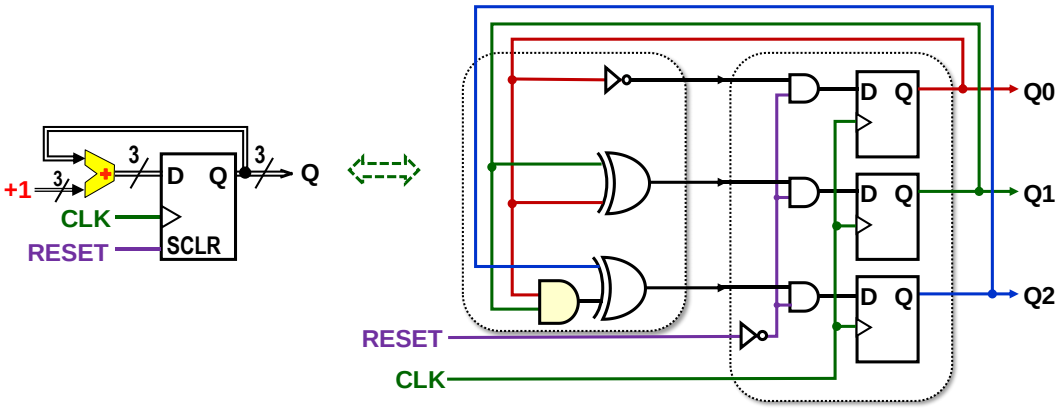


Figure 166 - Three-bit counter for flashing snake

The counter runs in an endless cycle, flipping on the rising edge. For example, it might have the following output if we initialize it at the beginning and then sometime in the middle:

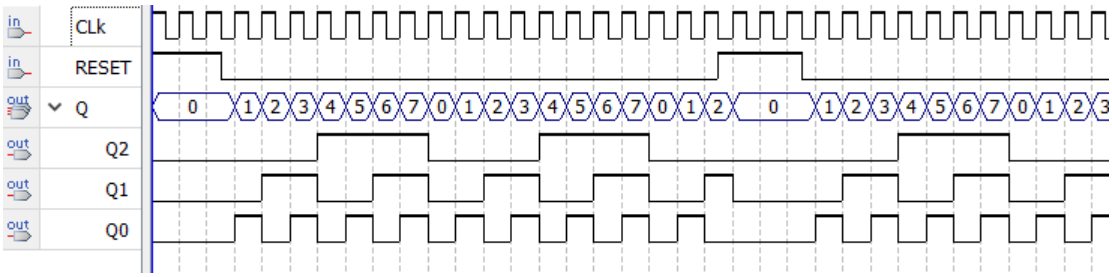


Figure 167 - Example of 3-bit counter output

If we want a counter with a shorter read cycle, we add an adder and comparator to the 4-bit register. We will solve the clearing with a multiplexer because we need it in two cases.

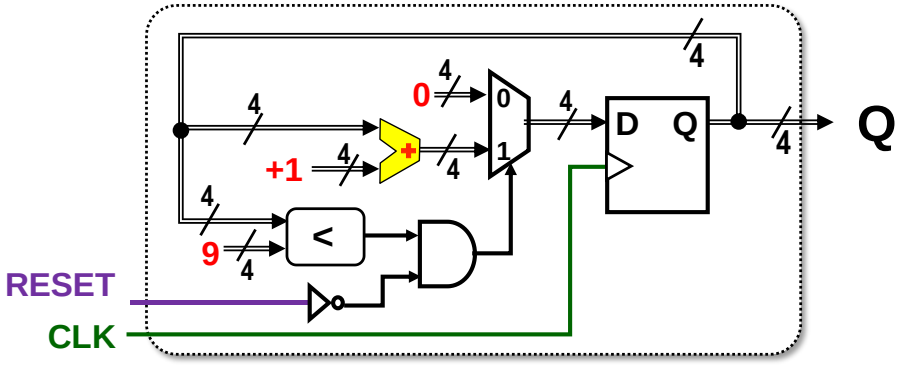


Figure 168 - Decadic counter

This time, we did not draw the circuit diagram with gates; it would be too complicated. If we were to extend the Q output to more decades and deal with the transfers between them, the block diagram would lose its clarity. We can poorly compare the versions of figures. Even if they depict the same circuit, they may have different graphical layouts of its elements.

As the complexity of circuits grew, sometime before 1980, there was more and more talk of a design crisis that initiated the development of HDLs for textual descriptions of wirings.

In HDL Verilog, the decade counter is created much faster than the schema:

```
module DecadeCounter(input CLK, output reg [3:0] Q);
always @( posedge CLK )
begin
   if (Q<9 && !RESET) Q<=Q+1; else Q<=0; end
endmodule
```

If we want to see what has been built, the development environment draws a diagram for us, but uses its symbols. The figure below is an economical notation of the internal AHDL language[54] . The figure below expresses the same circuit as the schema above, with a different graphical layout.

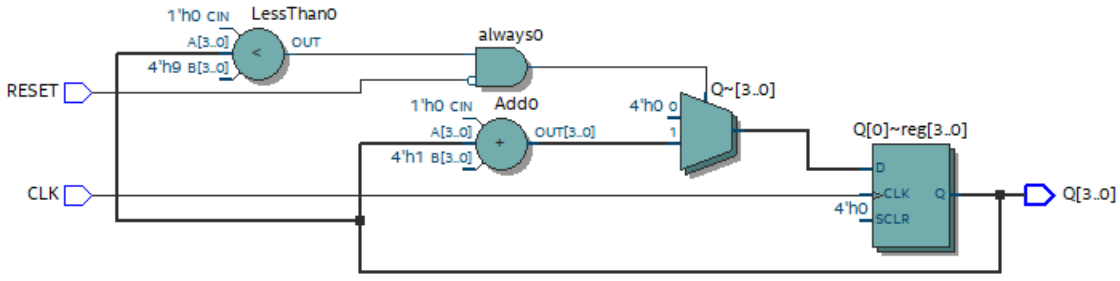


Figure 169 - Schematic of the circuit created from the code

The displayed result corresponds to the internal meta-schema built from our code. In the next steps, it will be optimized and distributed to the specific structure of the FPGA we are using.

[54] Altera Hardware Description Language

The text version of the circuit can also be simulated to verify that we have done the design correctly.

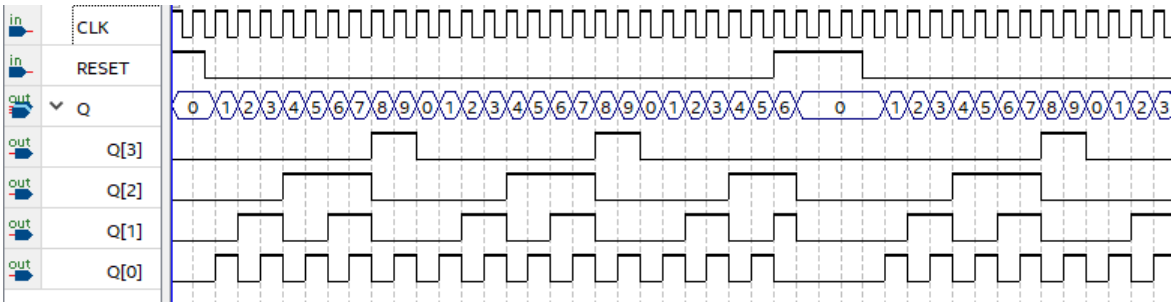


Figure 170 - Simulation of a decadic counter

Simulation is always practical. Verilog does not guard many circuit transgressions; it assumes its user knows what he is doing. Many professional designers prefer it for shorter specifications that allow writing faster, while others choose the more rigorous VHDL.

In VHDL 2008, the decade counter code will be slightly longer. It not only includes libraries that can be used to increase variability, but it also uses data types and offers the possibility of multiple architectures (internal circuitry), which is useful for debugging:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity DecadeCounter is
port (CLK, RESET : in std_logic; Q : out unsigned(3 downto 0));
end entity;

architecture RTL of DecadeCounter is
begin
  process(CLK)
  begin
     if rising_edge(CLK) then
          if Q<9 and RESET='0' then Q<=Q+1; else Q<=X "0"; end if;
     end if;
  end process;
end architecture;
```

However, we don't write the more extended VHDL code all by ourselves☺ Library headers exist in every VHDL file, these are copied from pre-made templates that include prototypes of entity and architecture blocks. Longer keywords, such as std_logic, are again inserted for us by autocompletion, now a common feature of code editors, so circuit description takes only slightly more time than in Verilog.

The time to write the code is not the most critical parameter, as HDL codes tend to be much shorter than C source files. Circuit development is most accelerated when we make minimal errors in the circuit description. These take longer to find than in a traditional program. And VHDL can reduce errors and shorten the debugging time by its strict checks.

Here, we end our explanation. Why? More complex synchronous circuits are not worth studying without the possibility of trying their functionality. We want to combine their explanation with HDL language so that readers can test them and become more familiar with their function.

7.6 What to do next?

We can recommend that readers choose some HDL language for themselves.

From a tutor's point of view, we can advise the more rigorous VHDL, which remains popular in Europe, and we can better learn the correct style. It gives us more hope that our design will work after passing through all its strict type checks, sometimes seemingly *tedious*. In Verilog, beginners quickly make mistakes that take a long time to fix.

Once we've honed our style on the numerous examples we've solved, we decide whether to continue with VHDL. Design experts say the eventual switch from VHDL to Verilog is usually straightforward, but the reverse movement is generally more complex. Anyone who wants to do more circuitry needs to know both languages anyway.

The newer SystemVerilog is also an up-and-coming alternative. It was inspired not only by Verilog, but also by VHDL and C++. The IEEE standardized it in 2005 and again in 2009 when it merged with Verilog.

If the reader is not a professional whom the company pays for development tools, then he can proceed as follows:

- First, he looks at what free development tools and circuit simulators are available. We chose Intel® Quartus® Prime Lite Edition Design Software in our LSP course because it includes the ability to draw circuits in symbolic diagrams, which is handy for the initial steps. There is also a freely available simulator for it, Altera-ModelSim, but only up to version 20.1.1 (as of November 2020).
- The user then finds out what HDL languages are supported by his free environment and chooses one that his simulator can allow since most of them do not allow the use of all HDL constructs but only a subset that is easier to process.

The free versions limit the range of FPGA types that can be used. Accordingly, we then choose a development board that we like and contain supported FPGAs.

For VHDL, users can also use the GHDL open-source simulator that offers only command-line tools but allows fast simulations. A large community uses and still improves it! In Windows, GHDL is installed with the aid of MSYS2 Software Distribution and Building Platform for Windows and takes only 2.2 GB, half of ModelSim.

8 Conclusion

The textbook covers only the circumferential basics suitable for learning HDL languages. It may eventually be expanded to include an explanation of memories, shift registers, and various counters. It would also benefit from a description of the finite automaton, the FSM - Finite State Machine, which tends to be a common feature of circuits.

We are leaving further improvements to future versions of it, and for now, we will let these passages into the HDL textbooks. You can't just read about circuits. The same is true for them as for programming languages, where we don't advance beyond a specific limit just by studying their syntax but have to create and test programs. Of course, we can also use other people's code as our initial inspiration and start by slightly editing it so that we know how to write our own.

However, the situation with HDL languages is more complex. The development community is smaller and more closed. Readers can be advised here to copy very carefully from the web. Professional companies rarely publish their designs. Most of the code you will find freely has been created by novice designers and sometimes contains convoluted constructs that could have been developed more profitably. How do we know the less good ones from the better ones? When we make more circuit designs ourselves, it is very easy. Until then, we prefer to choose carefully the materials we are inspired by.

If someone decides to experiment in VHDL 2008, they can use our tutorial in time: Circuit Design in VHDL 2008 for C Programmers. It is created by updating older texts from VHDL 1993 to VHDL 2008. So we can expect it to be completed quickly. And it will contain dozens of tested examples.

9 Appendix

9.1 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the

document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You

may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections

Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does

not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

9.2 List of numbered figures and tables

Note: Many other pictures were inserted without naming, if they only expanded thematically other illustrations.