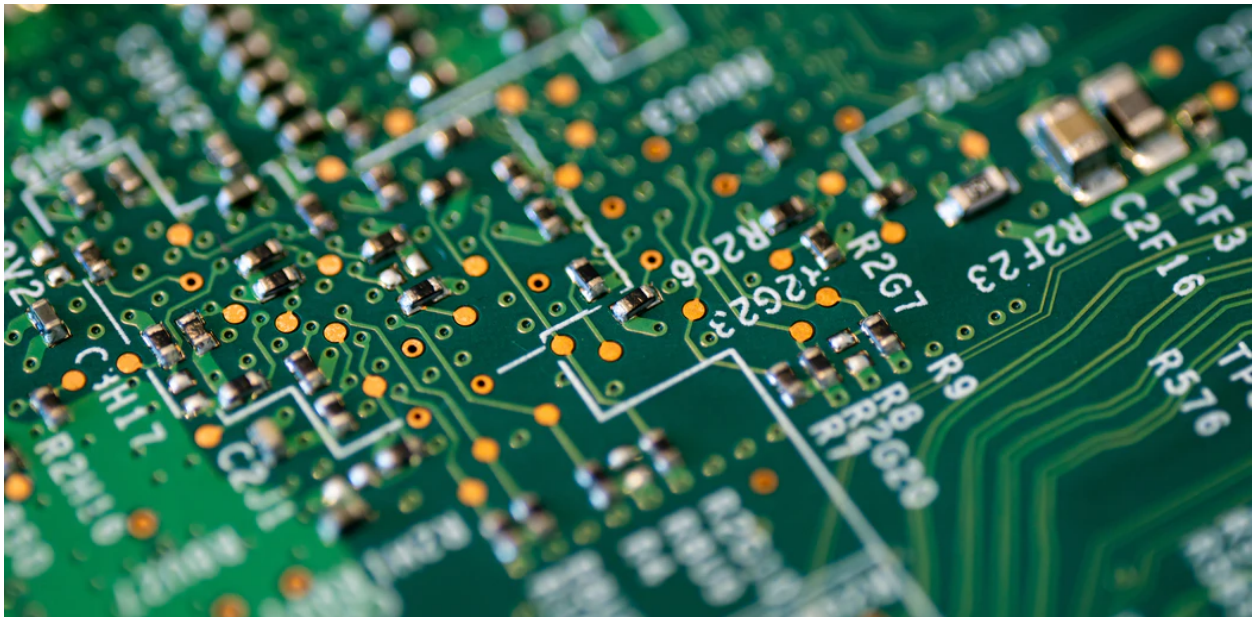
COMPUTER ARCHITECTURE AND ALU



Today computers have come a long way from their initial form, where addition was the most complex operation you could perform. Addition is still important today, because it is the most basic operation a computer can do, and in this chapter we are going to explain how a computer actually knows how to add two numbers together.

LMC (LITTLE MAN'S COMPUTER)

Before we describe LMC, we should remind ourselves of abstraction. Earlier we said that abstraction is used to hide complexity. In LMC the complexity is everything that is happening after we give the input. The whole idea of LMC is that there is a man in a box that does all the work.

In real life, a similar situation is happening almost every day. You go to the store and buy a box-cake. You add eggs and oil, mix it with the batch, pour it in a bowl and bake. You know what you need to add (input), you know in what order to do things (decode instructions), you work (execute work) on your kitchen counter (memory), and in the end you get a delicious cake (output).

LMC works in the same way, it reads the instructions in a form in which it understands what it needs to do. Check the previous lectures for more details about LMC, its instructions and how it works.

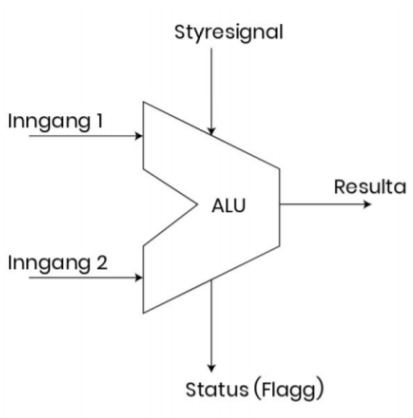
Kode	Navn	Beskrivelse
0xx	HLT	Stopper eksekveringen
1xx	ADD	Adderer verdien i angitt minnelokasjon med akkumulatoren
2xx	SUB	Subtraherer verdien i angitt minnelokasjon med akkumulatoren
3xx	STO	Lagrer akkumulatoren i angitt minnelokasjon
4xx	-	Ikke i bruk
5xx	LDA	Henter verdi fra minnet til akkumulatoren
6xx	BRA	Hopper til angitt adresse
7xx	BRZ	Hopper til angitt adresse hvis akkumulatoren er 0
8xx	BRP	Hopper til angitt adresse hvis akkumulatoren er ≥ 0
901	INP	Leser verdi fra input, og legger svaret i akkumulatoren
902	OUT	Skriver ut verdien i akkumulatoren
922	OTC	Skriver ut ASCII-tegn (ikke i boka)

Arithmetic and Logic Unit

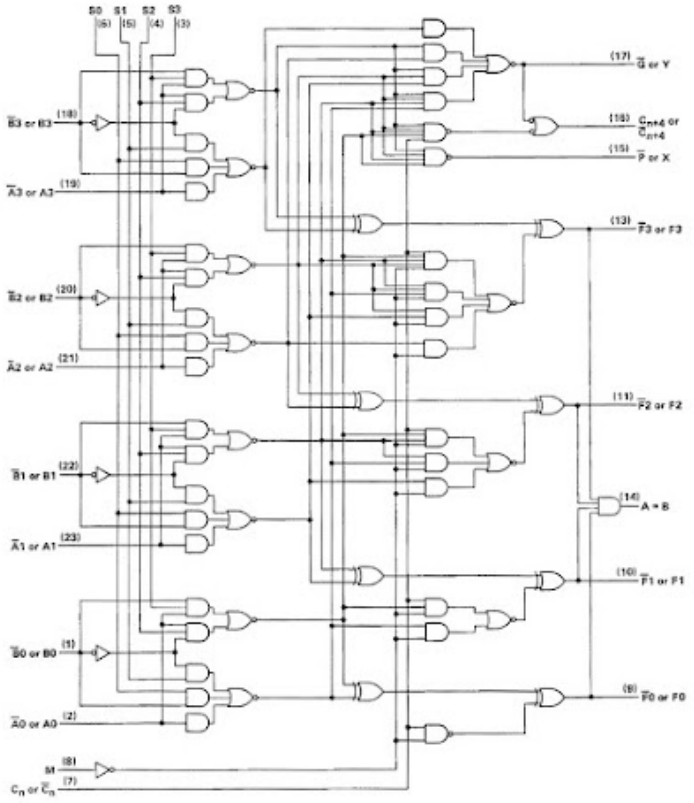
Most important part of any computer is the CPU. And the most important part of the CPU is the Arithmetic and Logic Unit (ALU). We could say that CPU is the CEO and the ALU is the CTO in our computer. It performs basic arithmetic and logic operations for the CPU. Examples of arithmetic operations are addition, subtraction, multiplication, and division, and logic operations are AND, NOT, OR etc. Later we will explain how to perform binary addition, and how we use logic gates to represent addition.

Even though it is smart, it doesn't understand decimal numbers, so the first thing we have to do is “translate” our number to binary. Once we’ve done that we can remove one layer of abstraction and take a look at what is really going on behind that oddly shaped symbol.

Looking at the symbol, we can see that it has three inputs and two outputs. Inputs are two data sets that we need to perform some operation on, and one input is for instruction (code) on what operation is to be performed. Output is result and status flag. Status flag indicates if there was an overflow, carry bit, or if the result was zero or negative number.



Below is the picture of an INTEL 74LS181 ALU chip, the first complete ALU that fits entirely inside a single chip. It provides thirty two 4-bit arithmetic and logic functions. Right now it might seem complicated, so let's get back to the basics - how to add two 4-bit binary numbers?



ADDERS

BINARY ADDING

To add two 4-bit binary numbers, we use the same technique as if we were to add 2 decimal numbers. We write one below the other, we sum them up from right to left, and if there is a carry, we add it to the next two numbers on the left. Example on how to add 9 and 5 in decimal and binary is explained below.

1		
	9	
+	5	
1	**4**	

			1	
	1	0	0	1
+	0	1	0	1
	1	**1**	**1**	**0**

LOGIC GATES

Now that we know how to add two binary numbers, how do we explain it to the computer how it's done? Computer doesn't speak Norwegian or English. Computers only understand 0 and 1, on and off, true and false. Luckily we have developed a way to speak “computer” language - boolean logic and logic gates!

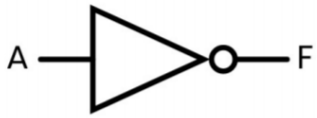
Most important logic gates are NOT, AND and OR. Let's say: I am hungry and I want to eat pizza, and I am not too lazy to walk, or I have a delivery app. The output of that statement would be: I am eating pizza. Logical.

Each gate has its own set of rules on what will be the output for a given input, and we call that a logic table. Three mentioned gates are shown below, firstly it says how we write it in boolean algebra, then the logic table, and finally the symbol.

INV - port

$F = A'$

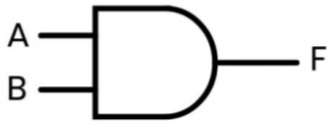
A	F
0	1
1	0



AND - port

$F = AB$

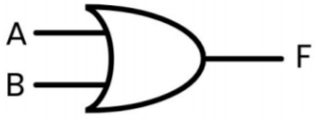
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



OR - port

$F = A + B$

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

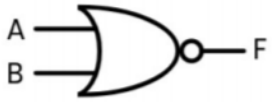


The circle after the triangle, and not the triangle itself, is the one that defines the negation of the statement. That same circle is used to negate two mentioned gates, we call them NAND and NOR (short for NotAND and NotOR).

NOR - port

$F = (A + B)'$

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

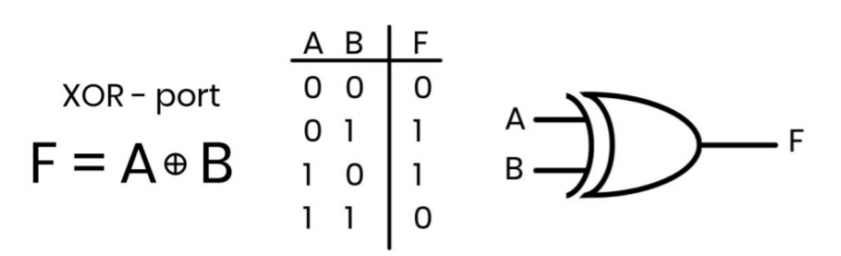


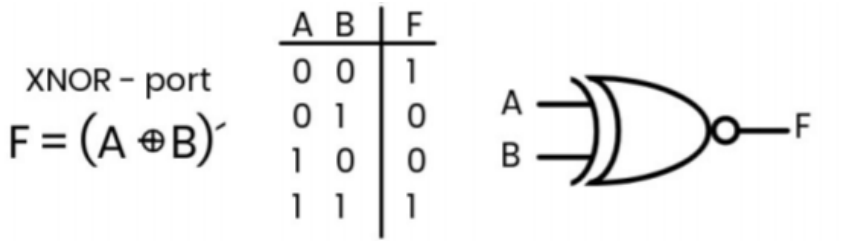
NAND - port

$F = (AB)'$

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

So, how does this help us with binary adding? For that we need to introduce one more gate - XOR. XOR gate is most important in binary adding because of its logic table. It is the same as an OR table, except when we have 1 and 1 as an input, 0 is the output. Why is that important? Well, when we add 1 and 1, we get a 0 and 1 carry bit. Finally some logic in this logic. Symbol is shown below, and it also has a negation option that we call XNOR.

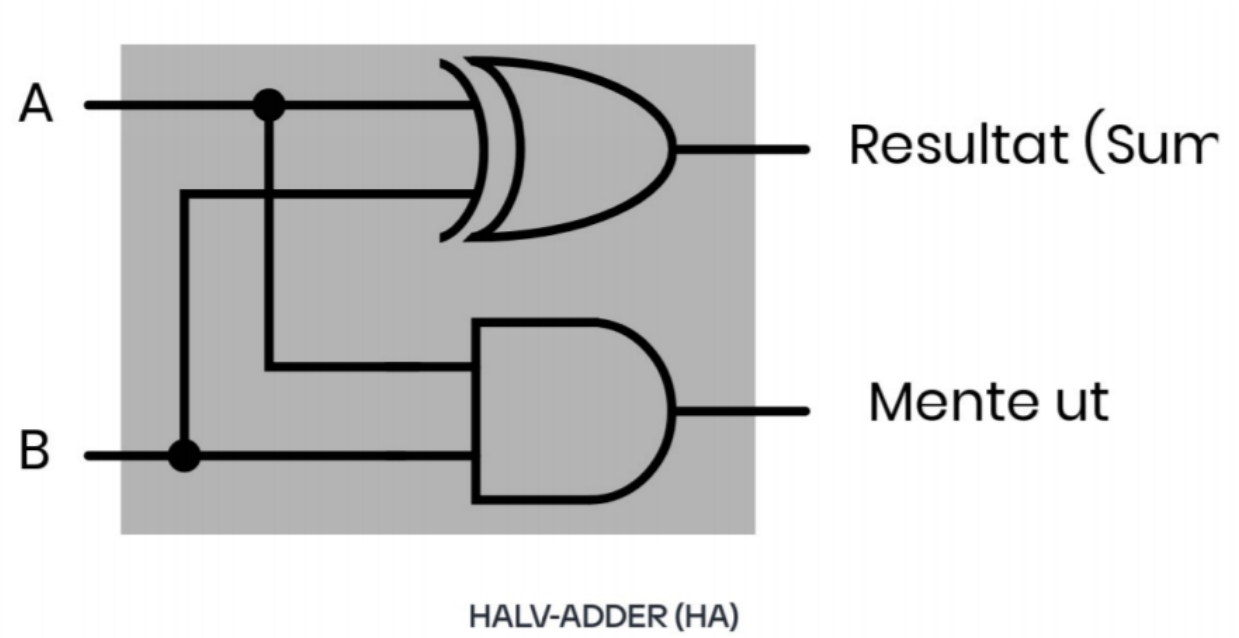




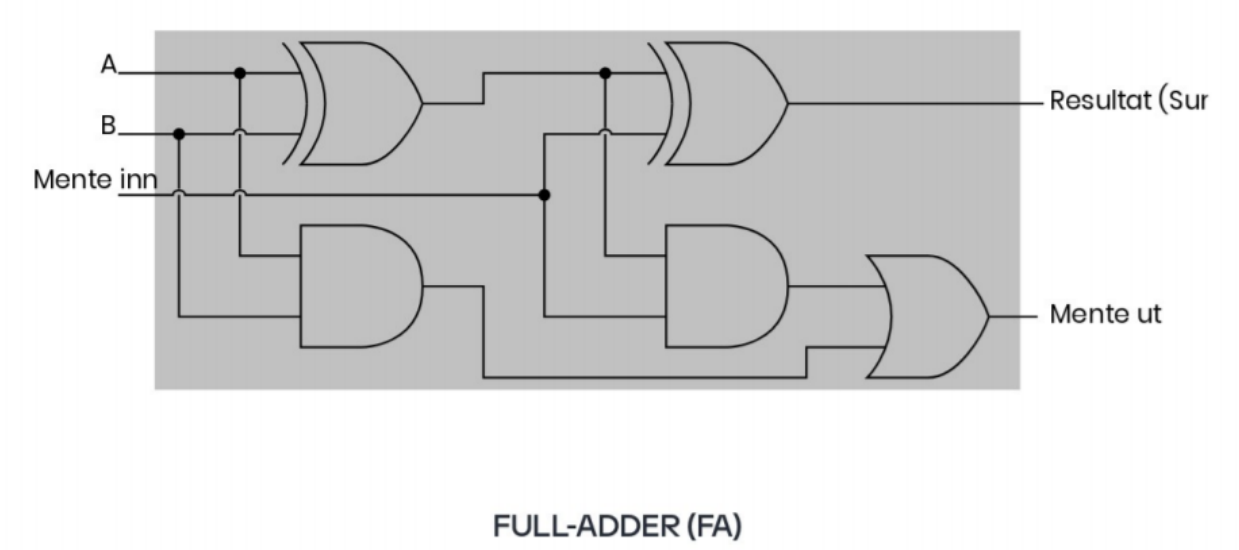
This is just an introduction to logic gates. Next chapter will discuss it in more depth, but for now this is enough for understanding adders that are the main ALU component.

ADDER

Now that we have all the information we need on how to speak “computer” language to add two numbers, let's arrange some logic gates to do the work for us. Let's start by adding two bits that are represented with A and B, we need an output (notice how we used XOR gate for this), and we need a carry bit, if there is any (now we use AND, because it only gives 1 if both inputs are 1).



Now you think, why is this called half adder if it does all the work we need to add two binary numbers. Good question. Look at the input. What if we need to add the carry bit as well? We have a solution to that problem as well - Full Adder!



FULL-ADDER (FA)

Things got complicated really fast here, 3 more additional gates for such a simple operation. We don't have to know all of this to sum two 4-bit numbers, so we will use abstraction and hide all of these logic gates into a box called FA (short for Full Adder). If we combine four full adders, we create one serial adder and we finally know how the 4-bit ALU performs its magic.