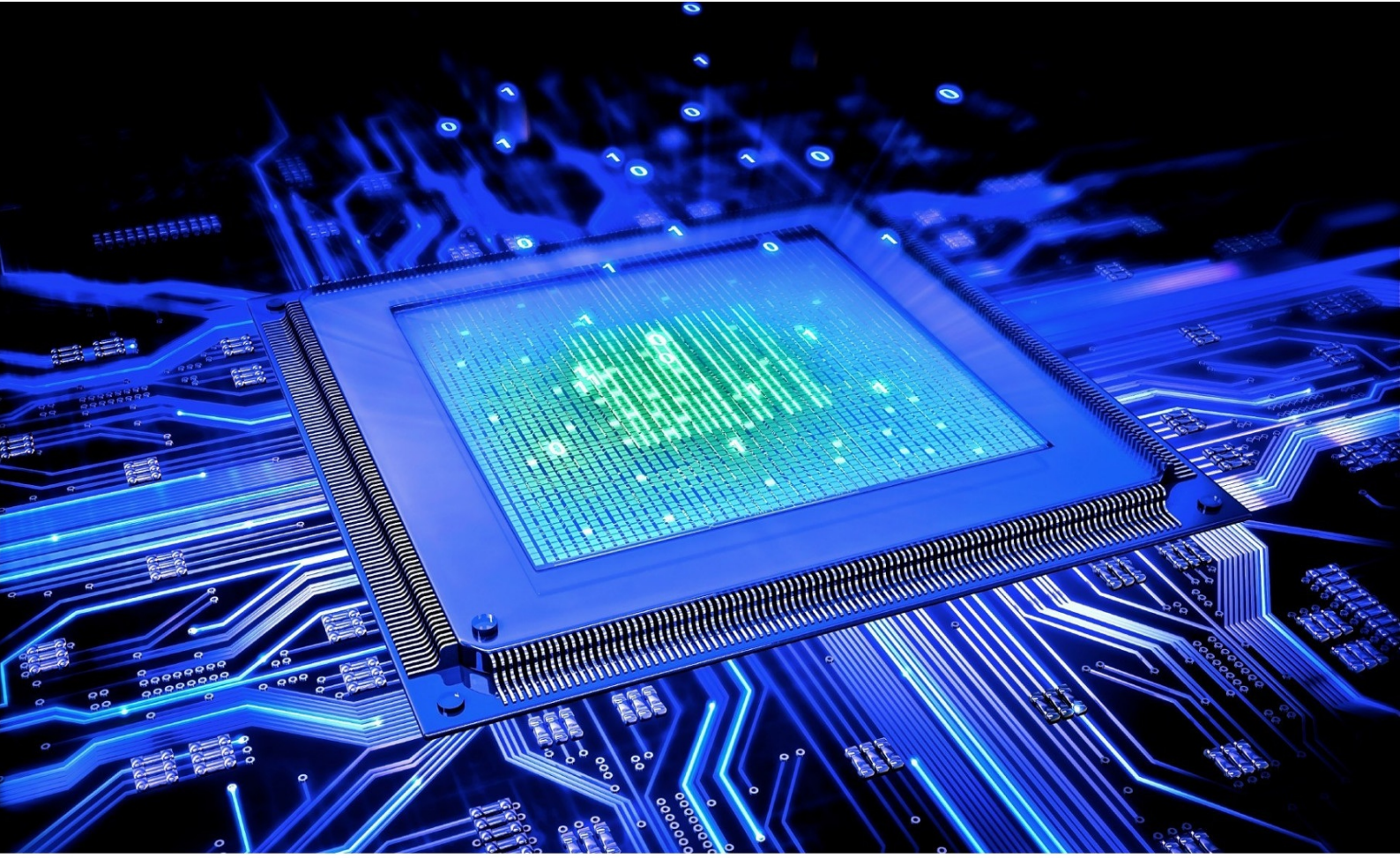


Microcontrollers, Design and Implementation

Author:
Muhammadreza Haghiri

Table of Contents

License

Introduction

I always wanted to design a micro controller or micro processor, and I managed to model them. Now, I've decided to show people how a micro controller is designed. And, I'm inspired by But How Do It Know? - The Basic Principles of Computer for Everyone by **J.Clark Scott**. I've read this book, and also studied MIPS, ARM and PowerPC architectures, so, in this book, I try to simplify a micro processor and turn it to a micro controller.

Audience

Of course, you need to be geek enough to read this book. I try to simplify topics, but if you have knowledge of logical circuits or micro processors, you'll understand this book easier.

What we learn?

First, you will understand how mankind communicated with machines for years, and the way people communicated with computers is unchanged. Then, you start learning *logical circuits* which is base of computer architecture and organization. And importance of knowledge of logical circuits is clear, in university (at least all Iranian universities), you should first pass the *Logical Circuits* course, and then you can take *Computer Architecture* course. After learning logical circuits, we start designing simple circuits and even computers, memory blocks, etc. At the end, we combine everything we designed, and it will become our computer. Finally, we can program our computer, and we actually understand how a computer can understand human's language.

Chapter 1 : What's a micro controller?

A microcontroller is a small computer, and usually *System on the Chip* (SoC), which include a processor core, memory unit and programmable input/output unit. The most famous family of microcontrollers is **AVR** or **Advanced Virtual RISC** family. This family of microcontrollers are used in boards such as Arduino, and they're used for industrial and educational purposes. A microcontroller, can control a toy car, and it also can control a US army drone. And this depends on microcontroller type, and the program we write for it.

What does a micro controller consist of?

In this section, I only explain process core, and in other chapters, we'll study other parts such as memory unit and I/O system. The process core is usually made up of :

1. **Arithmetic and Logical Unit**, also known as ALU. This unit, does every logical and Arithmetic operation, and it's the most important part of a microcontroller.
2. **Data Bus**, this part is simply a bunch of wires which transfers data between to parts of process core.
3. **Registers** are small memory blocks, and they can store data temporarily and transfer them. We always need them, because they're our temporary memory blocks and while we run a program, we need to store inputs and outputs.

But, How do it know?! ...

OK, now let's talk about **How does even a computer know what we want?** . The answer is simple, for example, when we write this code for x86 family :

```
MOV DX, OFFSET MSG
MOV AH, 09
INT 21H
```

and we assemble and run the code, it shows us a message (for example `Hello`). And computers can understand their owners by programs, let's go preciser. Imagine you travel Iran, and of course you can understand Iranians by speaking Persian. And Iranians can communicate with you in Persian. So, if we consider computer a foreigner, the assembly language (and in better word, machine code) is its language and we need

to communicate with it in its own language. In this book, you'll find how can we generate a simple machine language, and we can do simple projects with our small computer, and enjoy!

Chapter 2 : How to talk to computer?

In previous chapter, we talked about how a computer can understand us. In this chapter, we learn how to communicate with our computer, and how to command it. Let's take a look at a common *programming language*, for example, C++. When we want to print `Hello, World!` on C++, we write some code like this:

```
#include <iostream.h>
using namespace std;

int main(){
 cout<<"Hello, World!\n";
 return 0;
}
```

This is called a program, and when we *compile* it, it turns to a file, usually named `a.out`. And let's run our program:

```
~:$ ./a.out
Hello, World!
~:$
```

And now, we've communicated with our computer, but in language which is close to ours. This kind of programming languages are called **High Level** languages. They're close to human language, and they're easy to learn. There are a lot of high level languages such as `C++`, `Python`, `Ruby`, `Go`, etc. We use them when we need. For example, for coding a school project you may use C, or Go. For coding a commercial project, you may use Python, as it's easy to learn and also have a lot of libraries. But, what if you want to code directly to your hardware?! Now, we need a language which is close to machine's language.

The machine code

Machines are awesome, because they only use 0 and 1 to communicate with each other. Humans are not as smart as computers in this case, because we use lots of letters in daily communications. English includes 26 letters, Persian includes 32 letters, and there are languages with more than 100 letters. Computers have only two letters, `0` and `1`. Now, let's take a look at these letters!

Everything Binary...

OK, we told you we have two letters in machine language. So, we need to convert usual letters and numbers to words that computer can understand. We use **base 2** as a key to machine language. In base 10, the regular format of daily mathematics, we have digits 0 to 9, but in base 2, we have only 0 and 1. Every digit here is called a *binary digit* , or in short, a **bit**. But a single bit alone is not enough for us.

Word size

Imagine the word `hello` in English language. It has 4 characters. Also, imagine the number `42` , it has 2 digits. In computers, we need a fixed size, and our words can't be bigger or smaller than that. We call this fixed space **word size**. We need word size to compute carry digit, overflow and underflow, etc. One of the most popular word sizes, is a **byte**, which is made up of 8 bits. A simple byte is mapped like this :

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

Now, we initialized one byte using only zeros. But, we have two forms for a bit, zero or one. So, how can we convert a binary number to a decimal one?!

Convertion, it's easy!

Now, let's take a look at a simple binary number for example : `1001` . This number is a 4 bit one, but we defined our word size 8 bits. We convert it to an 8 bit number, we just need to replace missing bits with zeros :

7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1

And now, we know column 3 and 0 have value one, it's so easy to calculate:

```
2^3 + 2^0
```

now, we know `2^3` is equal to 8, and `2^0` is equal to one. So, the result is:

```
8 + 1 = 9
```

It's fine to convert binary numbers to decimal, but how can we convert a decimal number to binary? Now, we try to convert 135 to binary. We all know 135 is equal to `128 + 7` . Now, we can write this number like this :

```
2^7 + 2^2 + 2^1 + 2^0
```

So, we shall enter 1 in columns 7, 2, 1 and 0. Our result is just like this :

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	1

And yes! 10000111 is our result. And there's a qestion, is there any other formats for communicating with computers? Yes! We can use base 16, too. base 16 is known as hexadecimal, and we have digits 0 to F in that format. It's wierd, isn't it?

Hexadecimal world welcomes you

Now, let's talk about digits. In base 10, we have 10 digits. In base 3, we have 3 digits, but what about 16? In base 16 we have sixteen digits. But how can we code our digits? It's so easy, we use Latin letters instead of numbers. This table, shows you a simple convertion among binary, decimal and hexadecimal :

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

These are our digits, and we learned from this table that every 4 bit is equat to one hex digit. So, we can convert a decimal number to hexadecimal one using this table, and our knowledge of binary calculations. Let's think about 158. 158 is `128 + 30`. So, We can write this number like this :

```
2^7 + 2^4 + 2^3 + 2^2 + 2^1
```

And now, we can draw binary conversion table for this number like this :

7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	0

Oh, 10011110, is our result. let's convert it hexadecimal! It has 8 bits, it means it has 2 digits in hexadecimal format. So, we draw a hexadecimal conversion table for this number :

1	0
1001	1110

According to table, `100` is equal to 9, and `1110` is equal to E. So, our result is `9E`. In this sections, we learned about how computer sees data. A computer reads data in binary format, but we simplify it by showing binary numbers in hexadecimal, and of course we translate hexadecimal instructions for computer later.

Mathematics?

We learned how to communicate with a computer, and now, we need to know how to do arithmetic operations using a computer. In next chapter, we just take a look at how a computer can do arithmetic operations.

Chapter 3 : Arithmetic Operations

We do simple arithmetic operations in our daily life. For example, when you buy a candy for $2 , and you give the seller $5 banknote, he will give you $3. This is a simple subtraction we use in daily life. Or, if you want to buy a toy car for $100, and you have $80 in your pocket, you may go to bank and you will take $20 from your account. This is a simple addition and subtraction in daily life. But how a computer do that?!

In previous chapter, we talked about how computer reads data. We talked about binary digits. In this chapter, we take a look that addition and subtraction of them.

Addition

How do you add two decimal numbers? for example 105 and 55. This is how we add these to numbers :

	1	
1	0	5
	5	5
1	6	0

We start from ones, then we add **carry** of ones to decimal. Now, we can do the same for 1100 and 0100. I know we have decided to use 8 bit word size, and so I draw 8 bits table now :

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
			1	1			
0	0	0	0	1	1	0	0
0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0

Did you see, we used to empty bits for our carry. If we use a smaller word size, this carry can't be displayed in output, and that will result an error. OK, we just did a simple addition. Let's talk about a bigger one, 11000000 and 01000000. I'll draw a table for this addition :

C	7	6	5	4	3	2	1	0
1	1							
	1	1	0	0	0	0	0	0
	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

In column **C**, we've stored our carry. This means, we need a word size bigger than 8 bits, if we care about the final result. But, currently we don't care about it, and we just store it to a memory block called **Carry Flag** , and we'll talk about it later. Addition in binary is very, very easy. But, what about subtraction? we need a subtraction for our microcontroller, too. But how can we implement our subtraction?

Subtraction

It's a bit difficult to do a subtraction in computers, because we have no subtraction circuit. We have *Negator* and *Adder* instead of subtractor. So, we have different ways to make a number negative. I show you three popular way, and at the end, we choose on of them as our standard.

Sign/Modulus System

In this system, we have a sign bit, which is equal to the most valuable bit. If it's 1, number is negative, and if it's 0, number is positive. Let's see two examples , for example we convert 36 and -66 to a S/M number :

Decimal	S	6	5	4	3	2	1	0
36	0	0	1	0	0	1	0	0
-66	1	1	0	0	0	0	1	0

Oh, let's think about this system. This system is very easy-to-use, but have two great problems :

1. It makes our word size smaller, as one of our bits wasted.
2. It makes negative zero condition, and we need to use algorithms designed for this system. These algorithms are not popular, so we don't use S/M.

There's another easy way, and it's called **One's complement**.

One's Complement System

In this system, we just invert bits. Every 0 becomes 1, and every 1 becomes 0. Let's calculate -36 in this system.

Decimal	7	6	5	4	3	2	1	0
36	0	0	1	0	0	1	0	0
-36	1	1	0	1	1	0	1	1

This system is much better than S/M. but still have a great problem. It still makes negative zero, and it's not good for us. So, we have a final solution! that's using two's complement system!

Two's Complement System

This system is very similar to one's complement, but we just add 1 to one's complement, and it makes two's complement for us. Let's calculate -67 in this system :

Decimal	7	6	5	4	3	2	1	0
67	0	1	0	0	0	0	1	1
	1	0	1	1	1	1	0	0
								1
-67	1	0	1	1	1	1	0	1

In this system, we have no negative zero conditions, and all bits used efficiently. Now, we choose this system as our standard, and we'll design our subtraction parts using this system.

How to subtract?

OK, now we are able to make negative numbers. So, a subtraction operation is simply *addition of a number, to a negative number* . So, for example :

```
125 - 36
```

is actually :

```
125 + (-36)
```

So, first, we calculate -36 :

Decimal	7	6	5	4	3	2	1	0
36	0	0	1	0	0	1	0	0
	1	1	0	1	1	0	1	1
								1
-36	1	1	0	1	1	1	0	0

And now, we do our addition :

C	7	6	5	4	3	2	1	0
	1	1	1	1	1			
	0	1	1	1	1	1	0	1
	1	1	0	1	1	1	0	0
1	0	1	0	1	1	0	0	1

The carry is here normal, and every subtraction shall have one, now, we can do every subtraction with this method. The carry has a role now, we call it **Sign Flag** and when we want to design our final process core, we will consider a bit for this bit, too. In next chapter, we'll learn logical operations, which are very important in every computer.

Now, what should we do?

In our *proccess core* , we will use two's complement system, as the main system. Now, we need another type of operations, called *Logical Operations*. In next chapter, we learn logical operations.

Chapter 4 : Logical Operations

Let's talk about logic! Do you know what logic is? Logic is somekind of mathematics, mixed with philosophy in simple word. It's started by Greek philosopher, *Aristotle*. Simply, logic says : **John is taller than Ali, Ali is taller than Ahmed, so, John is taller than Ahmed**. This was a very, very simple example of a logical problem in our daily life. We can do this example for everything measurable in our daily life, souch as area, height, weight, etc.

How many logical operations we have?

We have some simple logics, and we review all of them here, and we solve some simple problems, and create new logics.

NOT

As we have *binary* system, we use 0 for everything **off** and 1 for everything **on**. 0 for everything **true** and 1 for everything **false**. So, here we have just one variable called **A**. Let's do NOT operation on A!

NOT	A	~A
	0	1
	1	0

We show `NOT A` in this form :

```
~A
```

This notation, helps us write *logical functions*. Functions are operations we do on one or more variables, and it has unique answer per inputs from a bunch of variables.

AND

This operation, is very simple, but has two logic inputs. The table of AND is like this :

AND	A	B	Out
	0	0	0
	0	1	0
	1	0	0
	1	1	1

We show `A AND B` in our logical notation like this :

```
AB
```

OR

Of course, you remember famous question from Shakespear's novel Hamlet, **To be, or not to be; this is the question!**. Now, we're going to explain what OR is. This table can explain this operation :

OR	A	B	Out
	0	0	0
	0	1	1
	1	0	1
	1	1	1

One of operations needs to be **True**, and it makes whole answer true. And, we show `A OR B` like this :

```
A + B
```

Truth Table

You saw, we used a table, which includes inputs and outputs, and also operation. This table is called **Truth Table**.

Let's play a game!

So, let's combine some logics and make new ones! The simplest ones are :

NAND

It means :

```
~(AB)
```

It's `NOT(AND(A, B))` in a simple word. And we draw truth table like this :

NAND	A	B	Out
	0	0	1
	0	1	1
	1	0	1
	1	1	0

NOR

It means :

```
~(A+B)
```

It's `NOT(OR(A, B))` in a simple word. The truth table is like this :

NOR	A	B	Out
	0	0	1
	0	1	0
	1	0	0
	1	1	0

Is there any more logics? of course yes! But, we will have a view on two others, in this chapter.

Complex Logics

There are to other logics, which are made from other logics, I would like to call them "Complex". Because They're not as simple as NAND or NOR. Also, we can call them "Exclusive Logics". This is what other engineers called them.

Exclusive OR

This logic, is implemented like this :

```
~AB + A~B
```

So, we will have a truth table like this :

XOR	A	B	Out
	0	0	0
	0	1	1
	1	0	1
	1	1	0

Exclusive NOR

This logic is the same as XOR, but with a little difference! If you apply a NOT function to XOR, you'll get XNOR. But, the best implementation of XNOR is this function :

```
~A~B + AB
```

And we'll get this truth table :

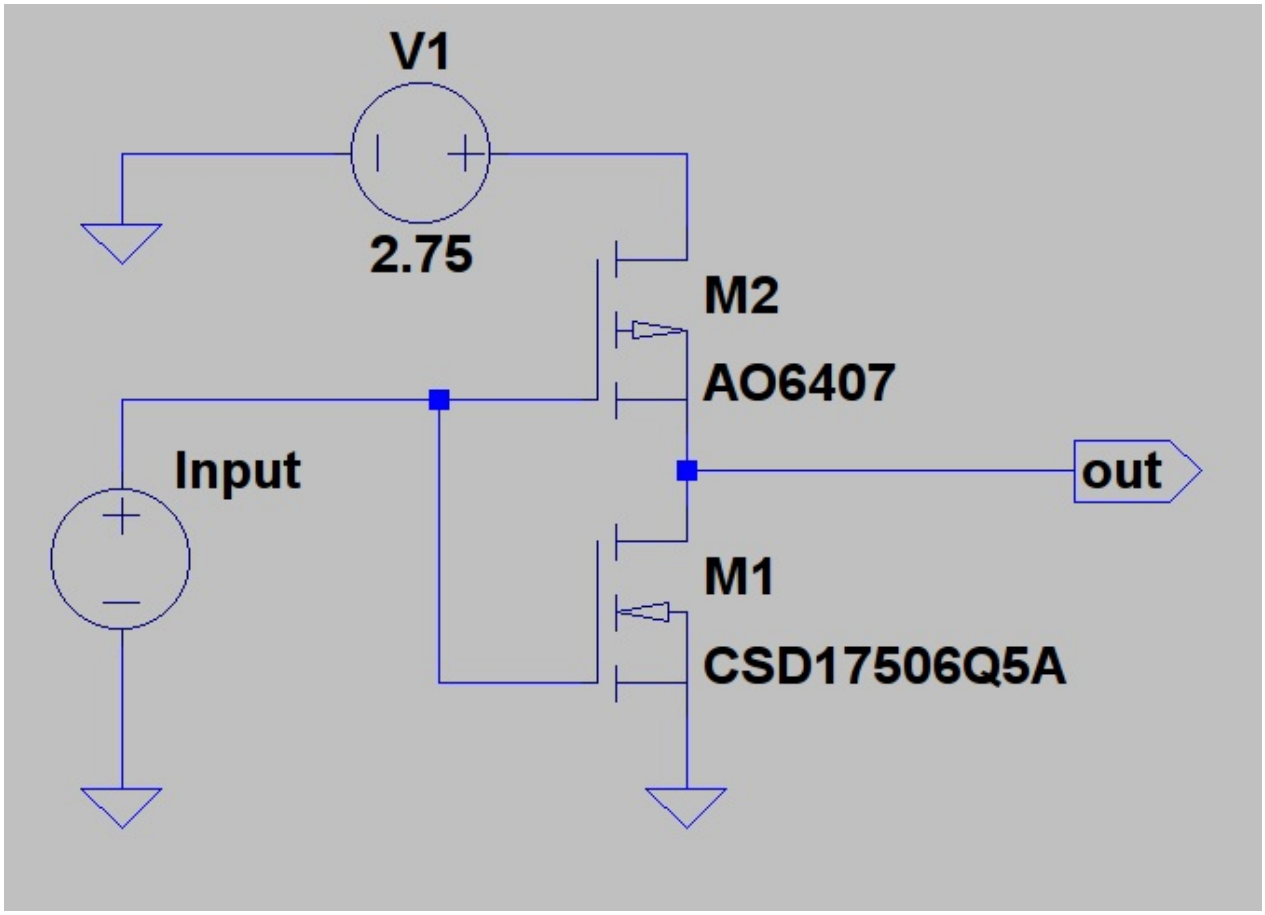
XNOR	A	B	Out
	0	0	1
	0	1	0
	1	0	0
	1	1	1

The journey to computer architecture!

Now, we know logics, and we need to learn about **Logical Circuits**, which are representation of these logics in computer science and electronics. In next chapter, we will learn how to use and design simple logical circuits, and then, we start designing and implementing our dear micro-controller. Of course you needed these chapters to learn the computer language, but after learning the language, you need to know how a computer is built!

Chapter 5 : Logical Circuits

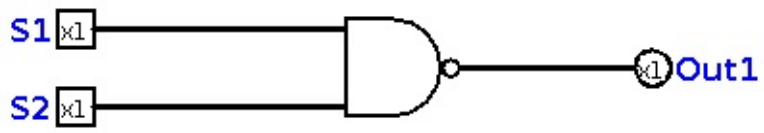
Do you remember two logics **NAND** and **NOR** ? These logics are called *Universal Logic*, because in digital electronics, we make all other logics using these two logics! The smallest part of a logical circuit, is called a **gate**. each gate is a specific arrangement of transistors. For example, this is a *NOT gate* using MOSFET transistors :



I picked this picture from one of my old projects, and in this book, we won't use any transistor, we just use symbolic schematics of logic gates to design logical circuits!

The NAND

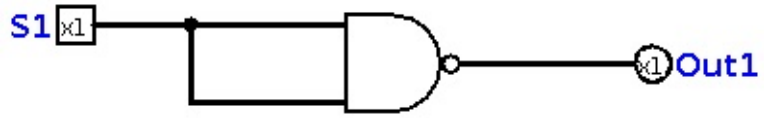
Although we can make all logics using NOR, I prefer using NAND. This is just my personal opinion, and after reading and understanding the previous chapter and this chapter, you can make all of these logics using NOR. This is what I call "The magic of boolean algebra". First, we need to know how NAND works! It works like a key with two switches, and when to switches are off, the output will be on! This is the simplest definition of NAND. When we want to show it on circuit, we use this shape :



A D-shaped thing with a bubble at the end. This is NAND! You know how it works, because you read the chapter four and you learned what are these functions!

NOT Gate

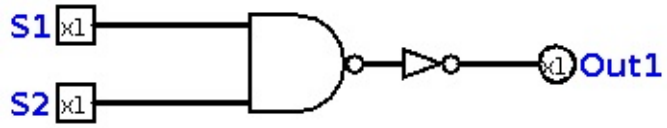
The NOT gate is another simple and basic gate, you need to know. It's built using a NAND Gate like this:



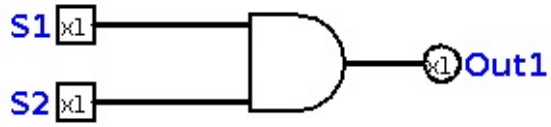
Did you see how it works? Yes! We simply connect two inputs of a NAND gate to a switch. The NOT gate in general, is represented like this :

AND Gate

As you remember from the previous chapter, we made NAND function by adding a NOT to AND. So, we know `~(~A) = A`. This means we can add a NOT gate in output of NAND, and get AND function. Just like this :

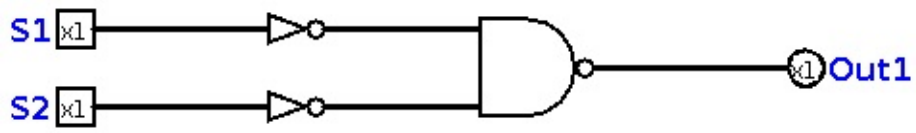


But in reallity, if you remove the bubble from NAND gate, you will have AND :

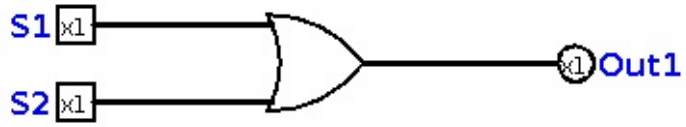


OR Gate

This is the last of these gates! Yes, this is the last, because we will design exclusive logics in next chapters, so the OR gate is the last gate we will know here! If we apply two NOT gates in the inputs of a NAND gate, it'll become an OR gate. like this :



But, this is the actuall OR :



Now, we're ready!

Actually, when you know how to represent logic gates, and you know how their functions work, you can design and implement logical circuits. A computer is much simpler than you think, and the hard part in design and implementation of a computer, is the correct usage and combination of logics. In next chapter, we design the simplest combinational circuits, and then we start desgining bigger ones.

Chapter 6 : Combinational Circuits

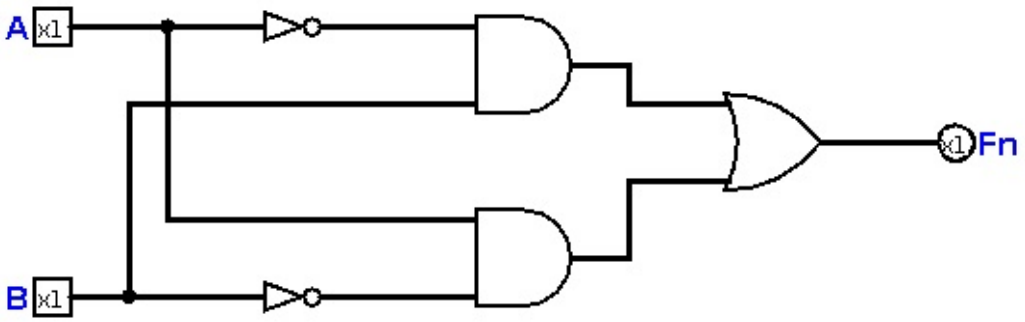
In the previous chapter, we saw how we can make logic gates. Of course, the easy way! In this chapter, we're going to make circuits which are famous as *Combinational*. Because we pick some well-known gates, and make a circuit out of them.

The Exclusive OR

Do you remember XOR from chapter four? The logical function of XOR was like this:

```
~AB + A~B
```

As you see, we need two NOT gates, two AND gates and one OR gate! Ok, let's make the circuit :



We also have a gate for XOR, because it's very common in circuits, and in transistor level, it's implemented much simpler. This is the XOR gate :

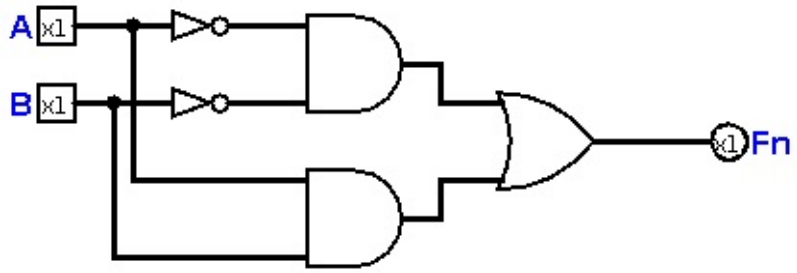
This was the simplest combinational circuit we could ever make. Let's make another one!

The Exclusive NOR

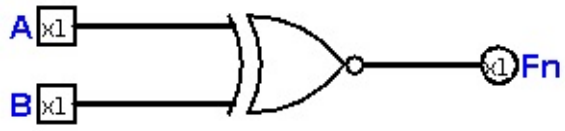
This is another combinational circuit we make in this chapter, the function is almost the same as XOR, but it has a little difference.

```
~A~B + AB
```

And now, we can make the circuit :



We also have a gate for this function. The gate looks like this :



More Logics?

Of course yes! We will make another useful logics in the next chapters. We all studied these six chapters for design and implementation of more logics.

Chapter 7 : The First Computer

Now, you know logic and arithmetic, two basics of computers. Of course, every device with the ability of solving logical and arithmetic problems, is a computer! Humans are computers, calculators are computers, and that expensive (and almost useless) iPhones are computers, too. In this chapter, we just make the simplest computer. We need this computer in near future, as a part of other computer. That's interesting, isn't that?

The Function

When we decide to design a device, we need to define its function. For example, imagine a mechanical engineer who wants to design an engine, but he doesn't describe its function! So, no one will buy that engine, because no one knows how it works or how it's made, or what's its function! As a good and practical design, I considered **Addition Machine**. It's a simple calculator (or computer) we can design using a few gates.

Boolean Algebra!

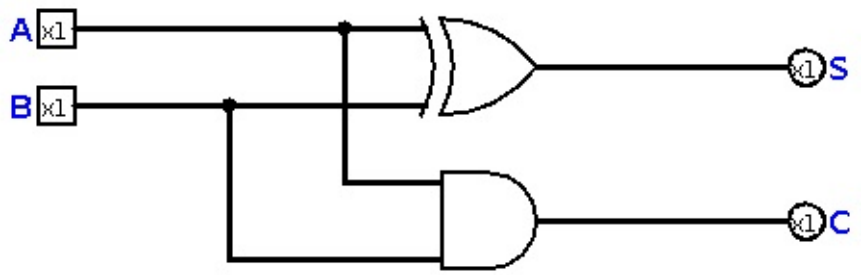
We learned boolean algebra in chapter four and for now, we just see it in action! For designing an *Addition Machine* , we need this simple function :

```
Sum = ~AB + ~BA
Carry = AB
```

Did you find the point? we just need a XOR and an AND gate!

The Half Adder

Imagine this circuit, which is the implementation of the function in previous part :



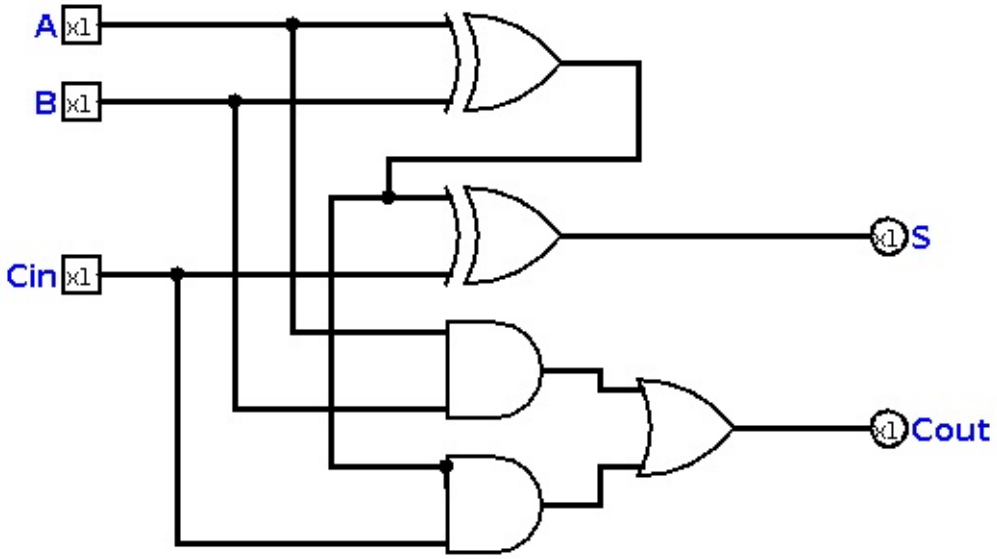
The truth table for this circuit will be :

H.A	A	B	Carry (C)	Sum (S)
	0	0	0	0
	0	1	0	1
	1	0	0	1
	1	1	1	0

It makes 0, 1 and 2 for us! It's another interesting thing we can design and implement! but, it's not complete yet! Why? It does not make 3 for us, it has 2 inputs but not 4 outputs! So, we still need some improvements on the circuit!

The Full Adder

Half adder is good, but it's not everything. Of course, it can't help us make bigger adders, so we need to connect two half adders, and make a new adder wich is called a **full adder**. A full adder, can actually make all expected outputs for us. This is the full adder :



Truth table of a full adder is like this :

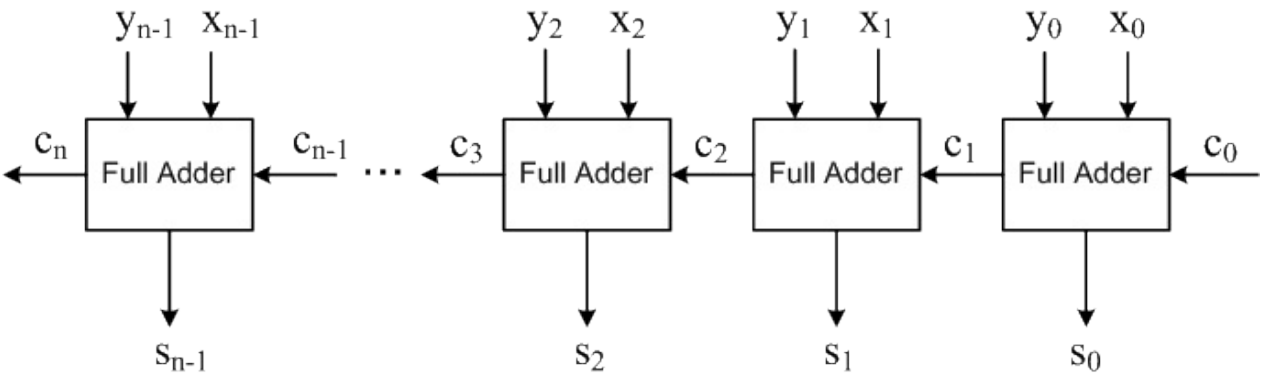
F. A	Carry-in	A	B	Carry(Cout)	Sum(S)
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
	0	1	1	1	0
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
	1	1	1	1	1

Now, we have ability of making a big adder! And that big adder will be our dear *Addition Machine*.

Ripple-Carry adder

The full adder we've designed is actually one bit. If we want to design a computer with the word size of one bit, we can consider that full adder as a simple computer. Now, as you remember from chapter two, we decided to make a microcontroller with word size of eight bits! So, I want to make a 8-bit adder. But How?!

As you can see, each full adder has a carry-in and a carry-out pin, what we need is connecting 8 full adders together, and we need these pins. If we put one adder , and connect its carry-out to next one's carry-in, then I'll get a *Ripple-Carry adder*. A ripple-carry adder looks like this :



Not only 8 bits, we can make the adder with a custom word-size, according to the design of ripple-carry adder. Congratulations! You made your first computer!

Let's talk about computers!

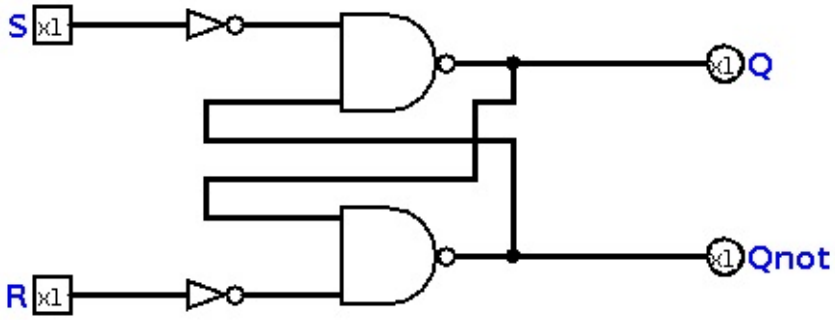
In this chapter, we just designed and implemented a simple computer known as a *Ripple Carry Adder* or *Addition Machine*. It's a complete computer with a single instruction, but of course it's not functional yet. Because we didn't design a memory unit, a simple control unit, etc. In next chapter, we will add some memory blocks to this computer and then, we start designing a complete and **functional** computer.

Chapter 8 : Memory

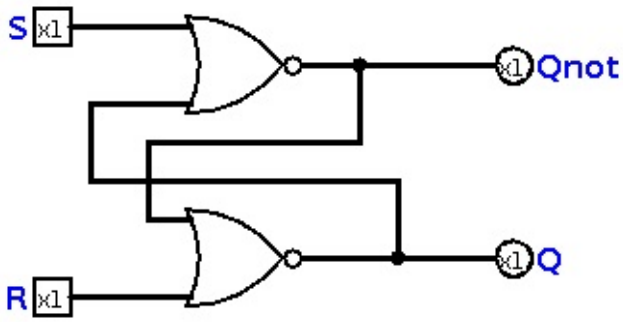
Although our *Addition Machine* was a complete computer, but actually, a computer without memory is like a car without seats. You know a car can be driven without seats, but we add seats, because seats can keep passengers! And this is the point! We want a certain space for keeping data. According to the chapter six, we know that a *combinational circuit* is a circuit which can solve a logic/arithmetic problem without storing the results. In this chapter, we are going to study a new family of circuits, which are called **Sequential Circuits**.

Everything is NAND!

You need NAND, even here. Of course, we need to know how NAND works and how can make memory blocks only using NAND. The simplest memory block is this :



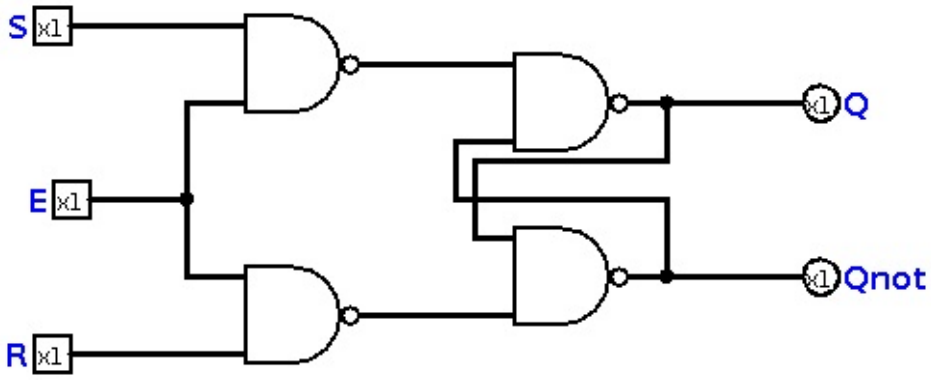
This is called an **active-low S-R flip flop**. If we replace NAND gates with NOR, we will have an **active-high S-R flip flop**. So, as all of the circuits of this book are active-high, let's see active-high version of our flip flop :



As you can see, `Q` and `~Q` are replaced in the new circuit. It doesn't matter what kind of flip flop you use, but I actually prefer the active-high one.

Basic Improvement

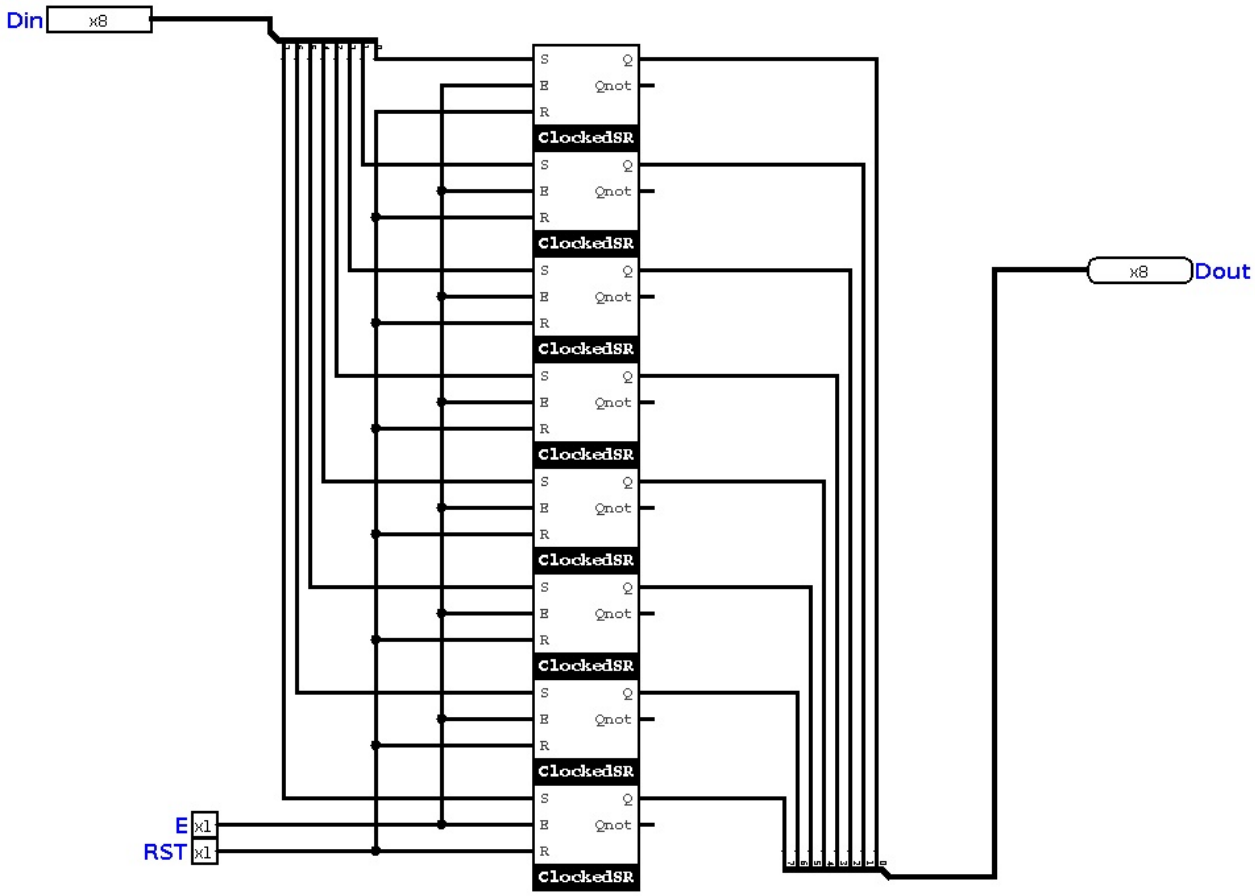
Consider that lovely NAND based implementation of SR, and if we modify that like this :



I added two gates, and one pin. The new pin is called **enable**. It enables the circuit, that means I actually need to turn it on for applying all changes in the input! The new circuit is called *SR flip-flop with enable*. Later, we will call that **Clocked SR**.

Register

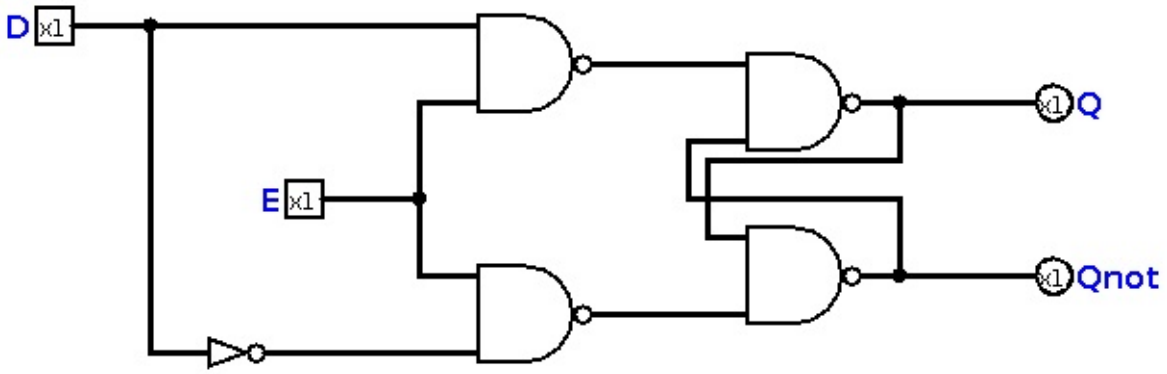
We can consider this flip flop as *one bit memory block* or *one bit register*. But, as we decided in chapter two, we want 8 bits registers. So, we need to connect eight memory blocks together! How is it possible? We just need to pick eight of them, then connect a common enable and reset button to them. And the input (S pins) will be parallel. The schematics of a register, is like this :



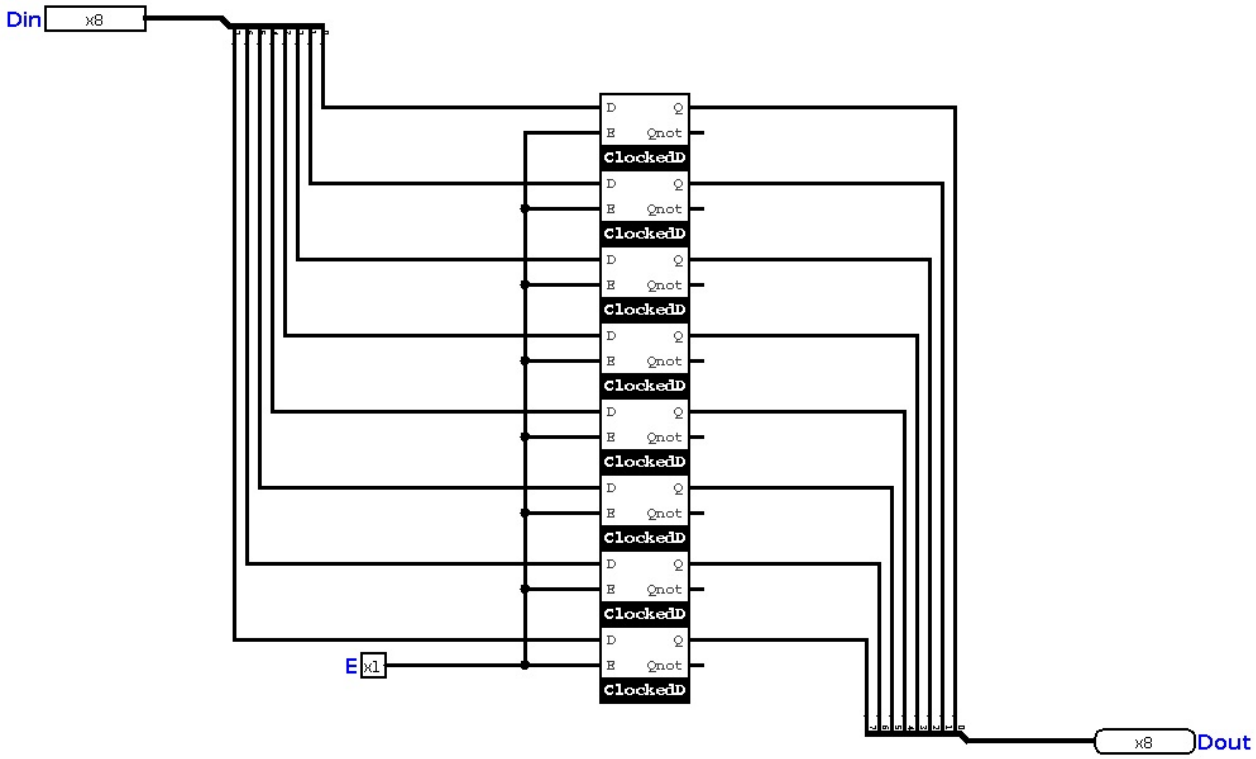
This is the simplest register we can make, but we need a better design for our flip flops to prevent noises and oscillations! So, we don't use this kind of register in our computer!

The new flip-flop

As we tested the S-R one, we found that S-R can't handle noises, and oscillations. This means we will have a lot of ***meta-stable*** conditions. As a good solution, we can use this kind of flip-flop :



Now, we connect 8 bits to D inputs, and one Enabler to all E's we have. finally, we'll get this :



The final register!

The schematics of a register, is just like this :

It has 8-bit *data-in* or ***Din*** input and also *data-out* or ***Dout*** output. Also, it has an enabler, which is shown by *E*.

What we need now?

Now, we need some organization for *memory management*. Also, we need to see how memory blocks work in action. In next chapters, we will add some memory blocks to our *Addition Machine* and see how it works, then we design more memory devices for our micro-controller.

Chapter 9 : Register File

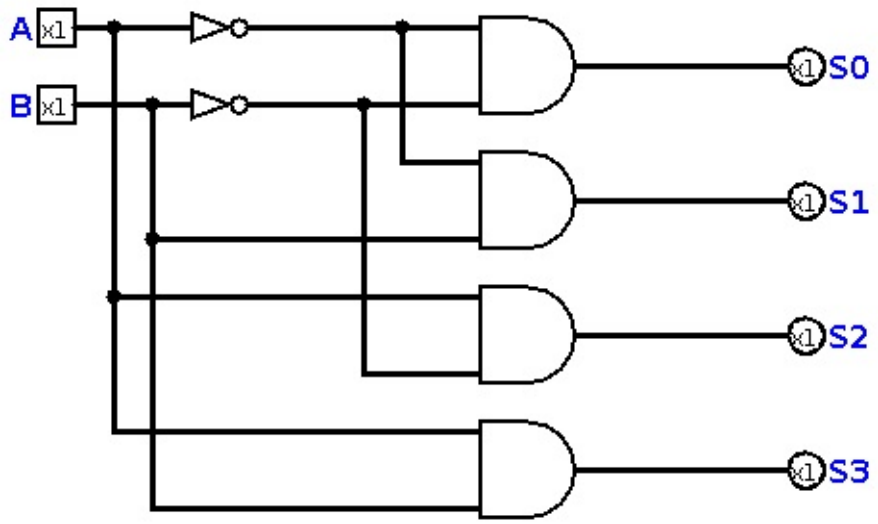
In previous chapter, we made 1 bit flip flops, and connected them together, then we had registers. Registers are needed, but we can't say "We are great computer engineers because we have registers!". Registers have special organization in a computer, and that's called ***Register File***. In this chapter, we will design a simple one!

The Decoder

Let's make another combinational circuit here. The circuit we are making here, is called ***Decoder*** . Decoders have *n* selectors, and `2^n` outputs. This is why they're decoders! You give them a binary number as input and you take a hexadecimal number in the output. Of course, you never get the *exact* hex number at the output, but you can find hexadecimal notation of input by using a decoder. Let's make one! The simplest decoder we make is a 2x4 decoder. According to `2^n` , we can also make 1x2 decoder, but it's not actually a real decored. It's an AND gate! The logical function of a 2x4 decoder is like this :

```
S0 = ~A~B
S1 = ~AB
S2 = A~B
S3 = AB
```

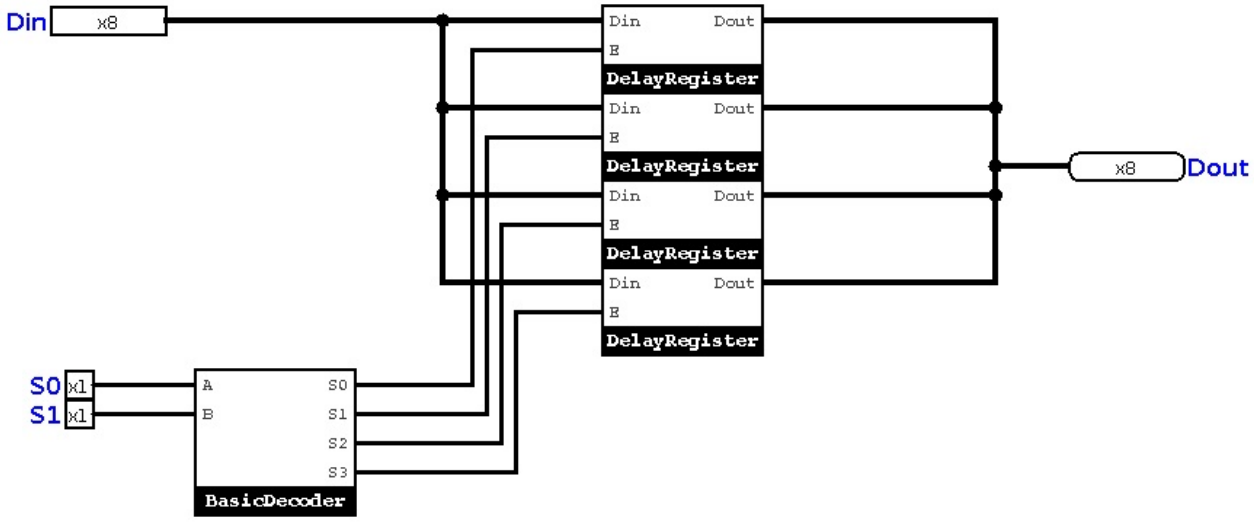
So, we can implement our decoder like this :



But, how we use this in a *Register File* ?

Simple Register File

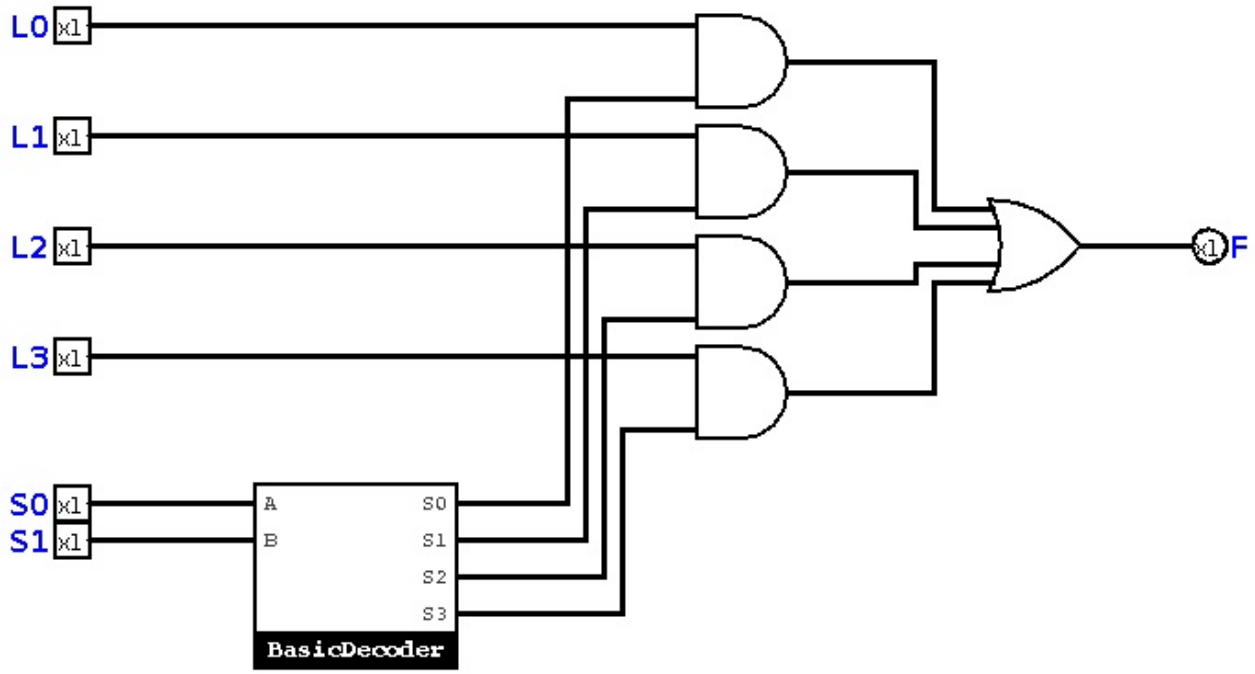
We have a 2x4 decoder. So, for now we can make a simple register file with four registers. The outputs of decoder, will be connected to *Enabler* pin of registers. Just like this :



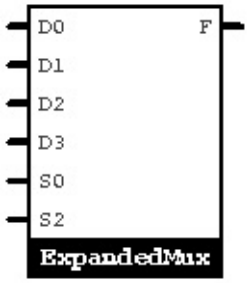
This is actually not a good design, it can generate a lot of noises, so we need another device, which allows us to select one of outputs!

The Multiplexer

You know, we need a device which acts like a decoder, but it does a selection among input data lines. This device is called a *Multiplexer*. In this book, We call it **Mux**. A mux can be implemented using a decoder, and a bunch of AND/OR gates. Like this : This is a simple mux :



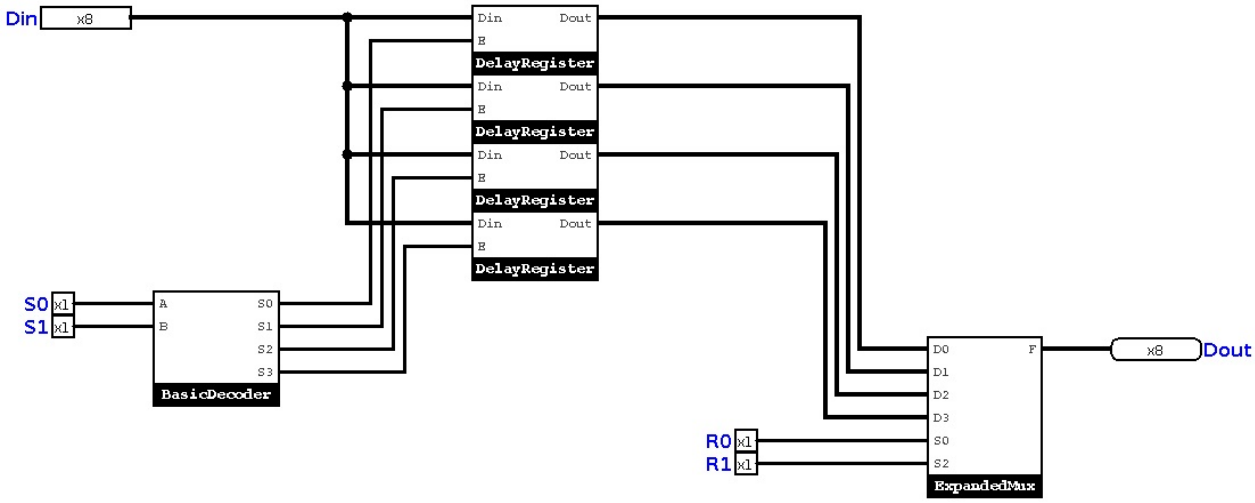
For our register file, we need a mux which can handle 8 bits input and output. So, I connect 8 muxes together, and I'll have a big mux like this :



Now, we can go back and complete our register file.

Advanced Register File

Now, we add a mux to the register file we designed, so we get this :



S0 and S1 are selectors for *moving* data to registers. we call that situation **Store**. And R0 & R1 help us *read* data from registers. This is what we call **Load**. We can claim that we have a ***Load/Store Architecture***.

Ready for Architecture!

Computer architecture is not only engineering. It includes mathematics, philosophy, analysis, etc. We need to combine all of them, to design an architecture. In next chapter, we will take a look on the philosophy behind computer architecture. Then, we will start design and implementation of our microprocessor.

Chapter 10 : Computer Architecture

This chapter is only theory of computer architecture. Actually, we need to know this part before we can start design and implementation. There are a lot of concepts we should know. In this chapter, I tried to cover the most important concepts.

Computer Architecture

The term *architecture* usually used for everything engineers build, and it means all engineers need to know the architecture of what they made. *Computer Architecture* is simply operations that a computer can do. Do you remember our ***Addition Machine***? That computer had a simple architecture, it could add two numbers, and if one of inputs was negative (according to the operator of computer, not computer itself!) it could help us subtract. That's nice, isn't that? So ***All instructions and operations of a computer is its architecture***. This is the only concept you actually feel when you are a *user*. When you are *engineer* you need to know more concepts.

Backward Compatibility

Backward compatibility is the most important theory in computer architecture and organization. In 70's, *Intel* made 8085 processor. Later, they made 8086. Let's see how they implemented Backward compatibility! Imagine a program written for 8085, 8086 must support and execute that. Every newer designs, should be compatible with older ones. For now, I have a laptop with *Intel Core i5* processor, and I can execute Windows XP on it. This is Backward compatibility.

Computer Organization

You know what *computer architecture* means, but what about organization? computer organization, is a level before architecture. In architecture, we just analyze computers instructions, and after analyzing it, we can tell ***what instructions this computer can do.***. But, the **structure of instructions* is not revealed yet! When we start studying structure of instructions, we actually study computer organization.

So, when we say *our computer can multiply*, we just talked about its architecture. But, when we say *our computer includes a multiplier, which is made up of adders* , we spoke about its organization. So, when we are talking about *what a computer can do* we are speaking about architecture. But, when we start talking about *how instructions are implemented* , we are talking about organization.

Complex or Reduced? This is the question

A computer is measured by number of its instructions, so, when this number is less than 100, we call the design ***Reduced Instruction Set Computer*** or in short, **RISC**. When we have more than 100 instructions in a computer, we call that ***Complex Instruction Set Computer*** or **CISC**. The computer we design in this book, will be a RISC, because RISCs are easy to study, learn, implement and understand. Also, RISCs are faster than CISCs.

Decisions!

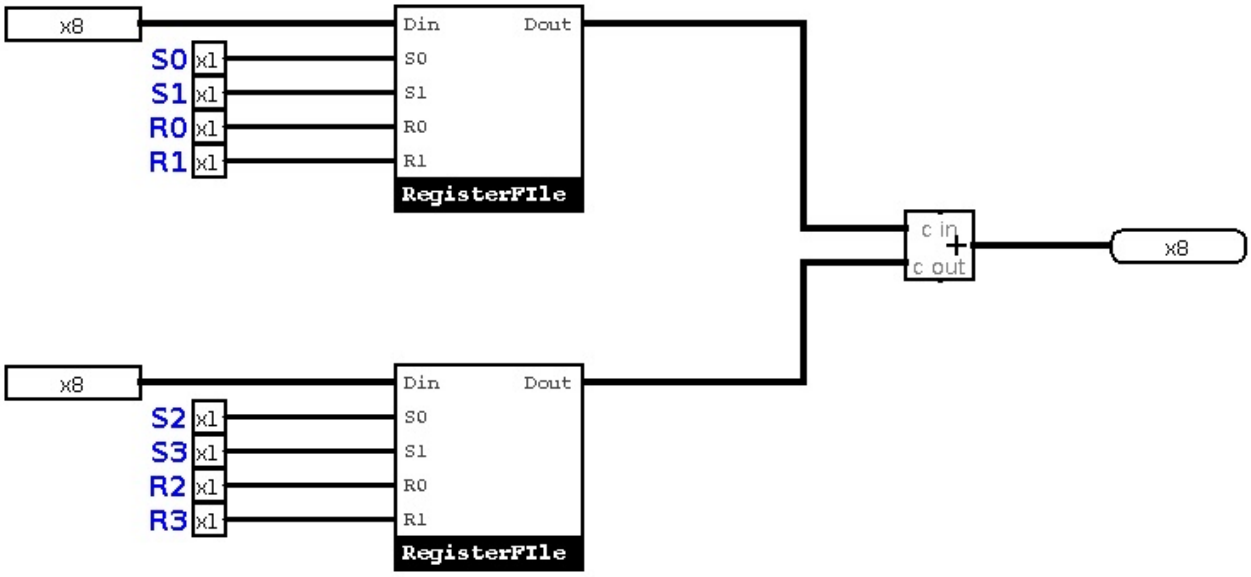
In chapter two, we decided to design a computer with the word size of 8 bits. So, we need to make decisions about its architecture and organization. Next chapter, will be about our design, and then, we start making our computer!

Chapter 11 : Design, Advanced Addition Machine!

In chapter seven we designed the must simple (and almost useless) computer, *The Addition Machine*. Now, we need to have a background of design and implementation of complex machines, so we add some useful features to our addition machine. Our machine had no memory blocks and we couldn't save our inputs and outputs. In this chapter, we will learn how to add memory blocks to our addition machine.

Managing inputs

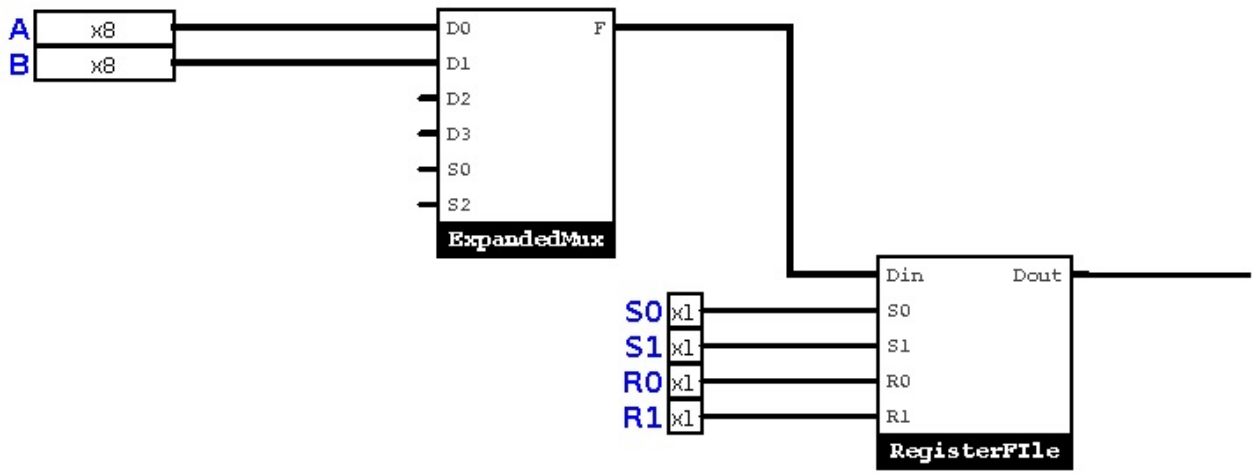
Remember our Register File? For now, I want to store my inputs in a register, and then, read from those registers. If you look at your simple (or scientific) calculator, you will see a button labled *M* or *M+*. That button is used for inserting the inputs or results in calculator's memory. So, I want to make a memory button for our ***Addition Machine***. Our circuit will be like this :



This is a good design now, but no! We can make it better, but that's enough for an ***Advanced Addition Machine***.

A new device?

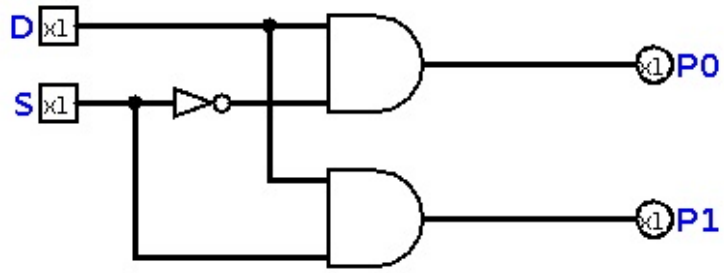
As we want only *one* register file, we need a multiplexer in the input. It looks like this :



So, for new system, we only waste four bits for control. But, we need a device which can help us define the path of our data! A device which can be used for unicast (only one direction), multi-cast (more than one direction) and broadcast (sending data to all directions)!.

Demultiplexer

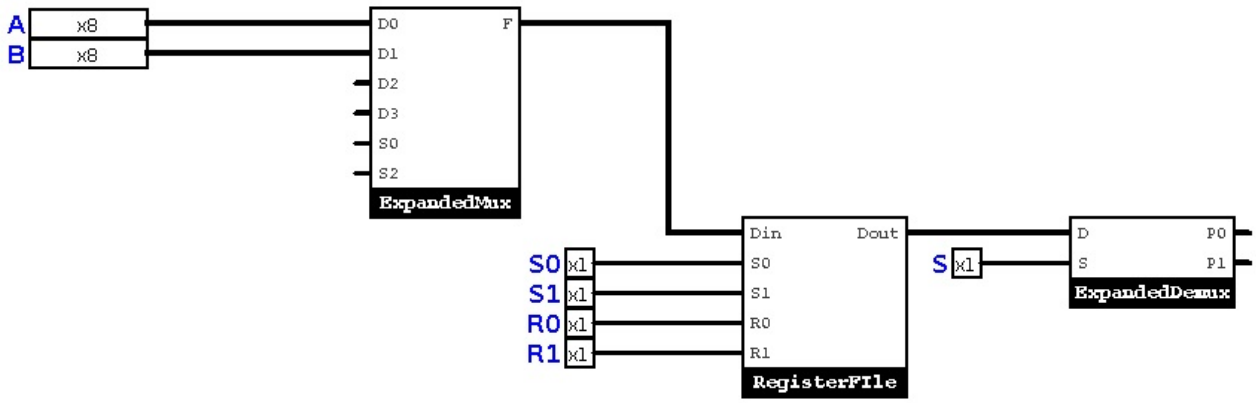
A Demultiplexer acts like a backward multiplexer. With a multiplexer, you choose one of data lines, but with a demultiplexer, you can send a single data line to a certain destination. The simplest one you can make, is a combination of NOTs and ANDs, like this :



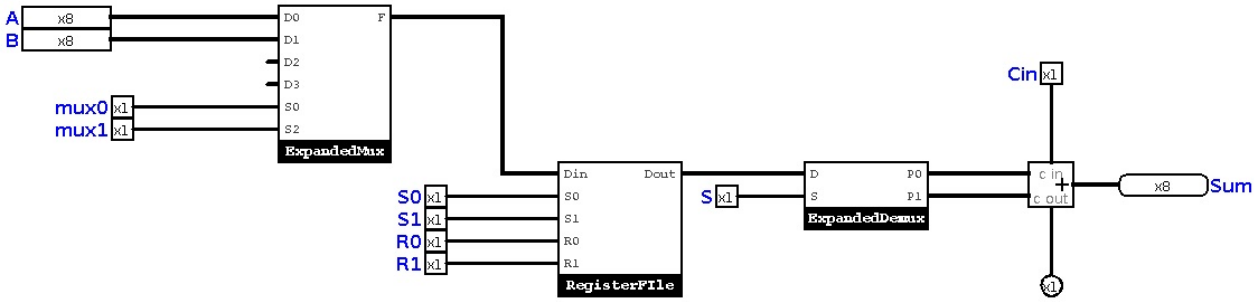
The input line, is connected to input of all ANDs, and *S*, which is our selector, connected with a NOT to first AND, and without NOT to second one. This means, with applying changes in selector(s) we can send our data to different lines! Let's add some deumx's to our ***Advanced Addition Machine***.

Selection!

Now, we need to add an 8 bit demux to our Addition Machine. The input lines will look like this :



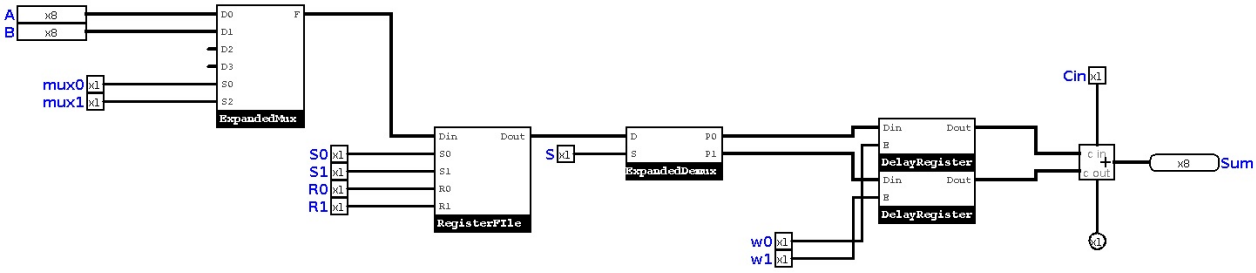
Now, we can select A or B inputs of the Adder! Let's add our adder :



This doesn't *add* anything, because we only switch our inputs? what's our final solution?

Temporary Registers

As you may find, there's no addition in our ***Advanced Addition Machine***. So, we can add two registers to our addition machine, and those are called ***Temprorary Registers***. Our machine, with temprorary registers will look like this :



Let's go!

Now, we designed a simple computer which can work, and it can help us make bigger machines or devices. But, in next chapter, we start our design of a real micro-processor. The future chapters, help you understand architecture and organization of the computers which we use everyday.

Chapter 12 : The Computer (Theory)

As we learned some theory and definitions in chapter ten, in this chapter we're going to continue our ***Theory of Design***. A computer without a strong theory behind it, is like a car without engine! This is the saddest truth about computer engineering! If you can't present a good documentation of you design or build, you'll fail! To avoid fails, we learn how to design a computer (theory phase) in this chapter!

The Instruction Set

Every computer, has an ***Instruction Set***. The way instrucions are implemented, is called ***Instruction Set Architecture*** or in short, ***ISA***. We need to design one for our dear computer, of course! A computer without ISA, is really impossible! There are a few tips we should follow in our design of ISA :

- As we want to design a RISC computer, we need to simplify every instrucion we need. For example, a NOR gate can be designed with an AND gate with inverted inputs, So we don't need to use both NOR and AND gates!
- We should document every step of our design, because it's our ***Computer Organization***. And in final product, that's important to be mentioned!
- The last tip is that we need to model everything we designed, later, we learn more about modeling.

What Instructions we need?

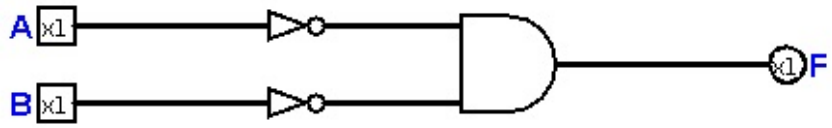
As a *real* computer, we need a computer which can handle at least one **logical** and one **arithmetic** Instruction. So, We can make a computer with these instructions :

- AND
- OR
- NAND
- NOR
- ADD (Addition)
- SUB (Substract)

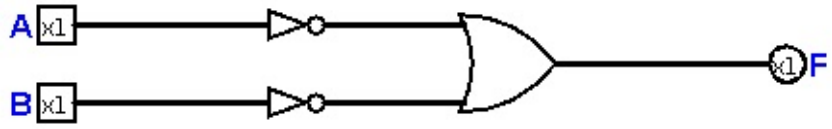
So, we need to design a unit which can handle *Arithmetic and Logical* instructions. we call this unit ***Arithmetic and Logical Unit*** or ***ALU***.

Computer Organization

This is the most difficult part, In this part you will learn how to think like an engineer! We know, Addition and Subtraction commands are implemented by XOR and AND and NOT gates. But, what about NAND and NOR? We can implement NOR using an AND gate like this :



Also, we can implement NAND, using an OR gate like this :



Memory Unit

I think, using a ***Read Only Memory*** or ***ROM*** is a good idea. But it's actually not! Because it can be programmed easily. So, we need another memory block, which is known as ***Random Access Memory*** or ***RAM***. In this book, I never detail how to make a RAM or ROM, it makes this book too hard to understand for people who have no idea about RAM or ROM. So, we know what kind of memory we have.

Starting Implementation

In this chapter, we took a look on the theory side of designing a computer. But, as engineers, we need to join the darker side, and start implement what we need. In the next chapter, We'll start design of our ALU, then start to connect other things we need to it, it'll be our awesome computer!

Chapter 13 : Arithmetic and Logical Unit

This is the start point! In this chapter, we build the main part of our computer, and we will be able to use it in a logical simulator. In old days, when computers were weak and expensive, ***ALU*** was an independent part, and it was not integrated with CPU. For example, a lot of computers used 74181 IC as the ALU (e.g VAX). Today, ALU, Register file, RAM, etc. Are all integrated in one chip and it's called a ***Microcontroller***. But, to understand computer better, we start a modular design, then we put all together, and we'll have a complete microcontroller.

Tools we need

The main tool you need is a computer, and this is the funny part that *we use a computer to design another computer*. But, you need one, because we don't want to make a computer in real life, we only want to simulate it. It doesn't matter which operating system your computer runs, Windows, OS X, Linux, etc. You need to install Java on your computer, and you know Java is freely available. Then, you need the most useful tool, Logisim Evolution, a good logical simulation software, which is free to use, distribute and advertise. All the schematics we designed in this book, are designed by this software.

A note on schematics

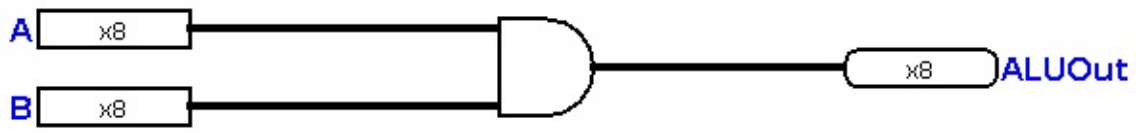
In the book, we built a lot of devices ourselves. But, ***Logisim*** is pre-packaged with good and useful devices such as Registers, Flip-Flops, Decoders, Multiplexers, etc. So, In this chapter and next chapters, we are going to use pre-made devices.

Start Point

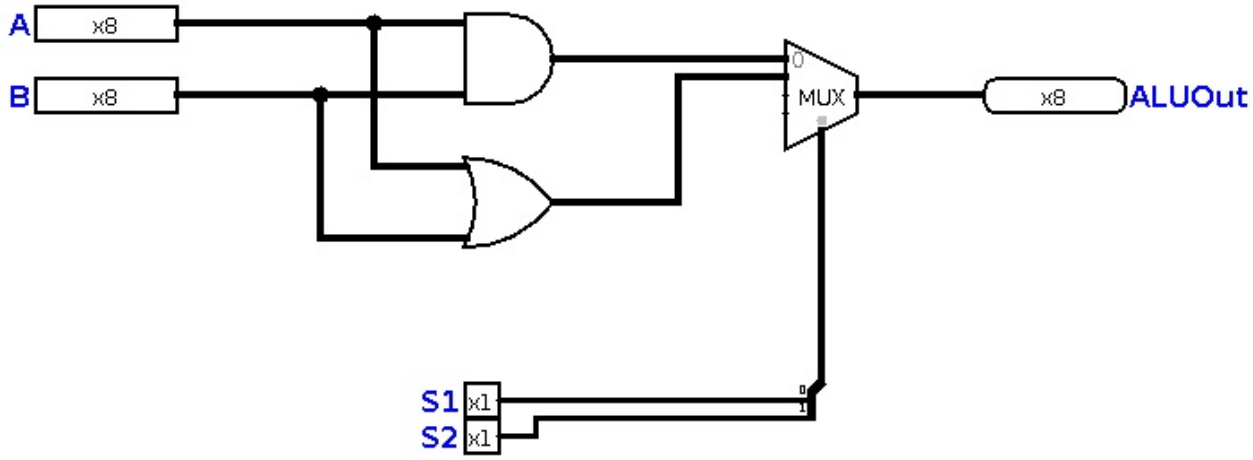
In chapter twelve, we decided about our instructions. I want to assign a code to each instruction, and that's called ***Instruction Code*** or ***Operation Code***. We will have table like this at the end:

Code	Instruction
We put a hexadecimal/binary code here	We put the instruction here

But, we don't have any instructions right now, let's implement AND, our very first instruction :



Then, we need to add OR, but wait! Two outputs for one ALU? Is it possible? of course not! So, we use something called a **Multiplexer**. So, I add a 4:1 mux, which is large enough to cover all we need! Then, I add the OR instruction, and we will get something like this :



Now, our table will be like this :

S2	S1	Instruction
0	0	AND
0	1	OR

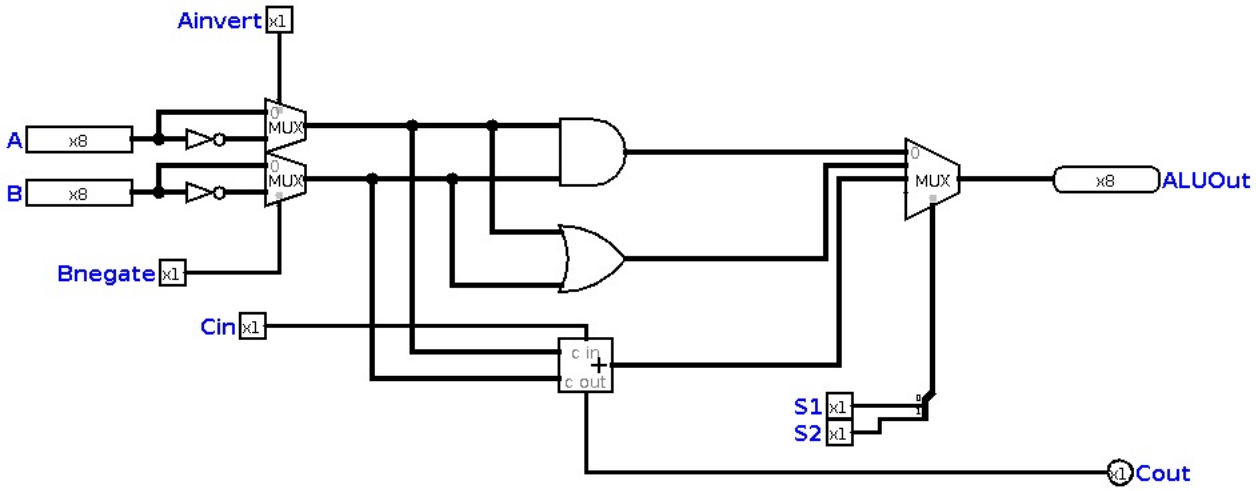
And our microcontroller's ALU, can run two simple programs! But this is not enough. You know, even the simplest processors classified as RISC, such as MIPS, can do more than these two instructions. These are logical instructions, but we need at least one or two arithmetic instructions. Lets add the instruction *ADD*. After adding that instruction, we will have this :

Cin	S2	S1	Instruction
0	0	0	AND
0	0	1	OR
0	1	0	ADD

We actually don't need *Cin* in our instruction codes, because we won't use this ALU to do ***signed addition***. But, we will need that for subtraction.

More instructions?

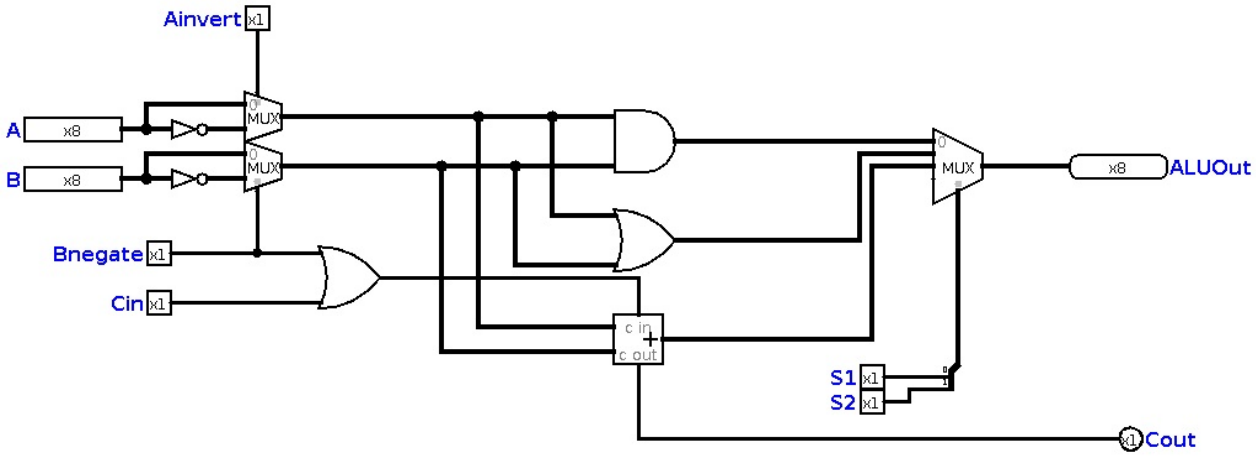
As you know, we are going to make a computer which can do AND, OR, NAND, NOR, ADD and SUB. We implemented AND, OR and ADD. But, how can we add NAND or NOR? We will a NOT gate, and a 2:1 multiplexer. This is how we can implement these two instructions :



Now we have a table like this :

Ainvert	Bnegate	S2	S1	Instruction
0	0	0	0	AND
0	0	0	1	OR
1	1	0	0	NOR
1	1	0	1	NAND
0	0	1	0	ADD

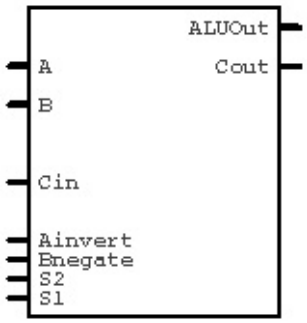
Now, our ALU is almost complete, but we still don't have subtraction. Subtraction is a bit complex, there are a lot of different ways to do subtraction, but we want a perfect way here. As you may remember from chapter three, we just suggested ***Two's Complement*** system for our dear computer. And you remember how that works! First, we need to invert all of the bits (1's become 0, 0's become 1), then we add 1 unit to the inverted number. *Cin* works like a *plus one* button for us, and *Bnegate* inverts *B* for us. If we add an OR gate, then we will have subtraction. Like this :



When *Bnegate* is high, automatically *Cin* becomes high on the adder. Now, we can have our ***Instruction Table*** like this :

Ainvert	Bnegate	S2	S1	Instruction
0	0	0	0	AND
0	0	0	1	OR
1	1	0	0	NOR
1	1	0	1	NAND
0	0	1	0	ADD
0	1	1	0	SUB

We implemented the most simple ALU we could, and after some modifications, ALU module will look like this :



I didn't remove *Cin* , because it will be needed when we want to expand our ALU (in following chapters, we will talk about expanding ALU)

Next Step

In these 13 chapters, we learned logic and design, and we've designed a simple ALU. But, ALU is not everything a computer needs. We need a simple memory block, which can control programs, etc. So, In next chapter, we will learn how to define a program structure, and then we can add memory block we need. At the end, we can really make our very own programmable microcontroller.

Chapter 14 : Program Structure

In previous chapter, we built the most important part of our microcontroller. In this chapter, we will take a look on the theory side of ***programs*** and we will define how our computer will understand programs. This is important indeed, because we need to write a simple *assembler* for our computer which can generate *machine code* for us.

Programming for a typical computer

You have a typical computer, which is compatible with `Intel x86 Family`. At least more than %90 of computers around me are like that. So, I can write this piece of art and run it on my own computer, yours, my friend's, etc.

```
#include <iostream>

using namespace std;

int main(){
    cout << "Piece of Art!\n";
    return 0;
}
```

But wait! If I install a C/C++ compiler on my mobile phone, tablet or gaming console, it will work! Why? The answer is easy. We write *programs* for all devices, everyone can use our program. In case of lower level languages like C or C++, we need to install the compiler on the target device, but actually we can use compiler's options to compile our code for different machines. But, in case of higher level languages, like Python, we only need the interpreter on the target device, and it will work!

But let's go deeper, deep inside the Intel x86 instruction set! Let's print the expression ***Piece of Art*** on console using assembly language :

```
STK SEGMENT
    DW 100 DUP(?)
STK ENDS

DTS SEGMENT
    TXT DB 'Piece of Art!', 10, 13, '$'
DTS ENDS

CDS SEGMENT
    ASSUME CS:CDS, SS:STK, DS:DTS
    MAIN PROC FAR
        MOV AX, SEG DTS
        MOV DS, AX
        MOV DX, OFFSET TXT
        MOV AH, 09H
        INT 21H
        MOV AH, 4CH
        INT 21H
    MAIN ENDP
CDS ENDS
END MAIN
```

This is not that hard to write and understand. Anyway, we are not going to talk about assembly programming here. So, have you seen the code? But It's not actually what computer sees! The computer only sees & understands a bunch of 0 and 1's. So, imagine one the above code lines. For example this :

```
MOV DS, AX
```

When you *assemble* the code using assembler, it will turn this line to something like this :

```
B8 0000
```

NOTE : THIS IS NOT THE EXACT MACHINE CODE OF THAT INSTRUCTION

Letter `B` stands for 4 bits, also 8 stands for 4 bits. We have 8 bits *Instruction Code* and 16 bits *hidden data* in x86 family ***Object Code***. In this chapter, we actually decide for a simple object code for our microprocessor.

Object Code

Now, we need to decide about a simple *Object Code* structure for our microcontroller. As I mentioned before, we won't have a register file in our microcontroller and we directly read data from RAM. You may ask why, because we can keep it simple for future developments and studies. Let's take a look at our *Instruction Table*, but this time, Hexadecimal :

Instruction	Code
AND	0x0
OR	0x1
NOR	0xC
NAND	0xD
ADD	0x2
SUB	0x6

Now, we need to upload our *Programs* to the microcontroller. But how? How it can detect the operands? This is a simple structure I suggest for that :

Instruction Code	Input A	Input B
4 bits	8 bits	8 bits

We have a simple 20-bit object code for our microcontroller. Let's see if we want to add two numbers, for example 15 and 8, how does it look like? It will be like this :

Instruction Code	Input A	Input B
0x2	0x0f	0x08

Now, we need to verify which bits are for which part. The most valuable bits can be for our instruction code, and others can be for inputs. This is what I suggest :

Instruction Code	Input A	Input B
Bits 16 - 19	Bits 8 - 15	Bits 7 - 0

So, the code `0x20f08` is the correct object code for `ADD 15, 8`.

The Final Step?

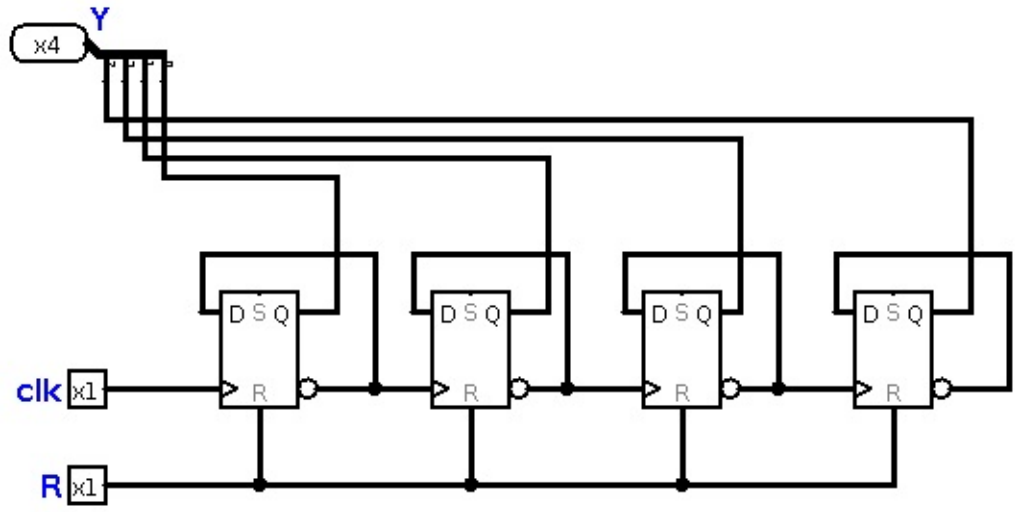
Now we have ALU, and we will add RAM to our ALU. Then, we program it and enjoy! The book is almost finished, and there's nothing more to say about the hardware side of a computer. But what about software and operating system? Be patient, we will take a look at software and operating systems in following chapters!

Chapter 15 : Microcontroller

In this chapter, we make our dear computer! You've studied previous chapters to learn how to make your very own computer! Actually, the most important part was *ALU*. In this chapter, we will design a simple and maybe stupid *Control Unit* and we add a *RAM* from logisim standard library. Then, we can program it! Let's know how we can do this!

Control Unit

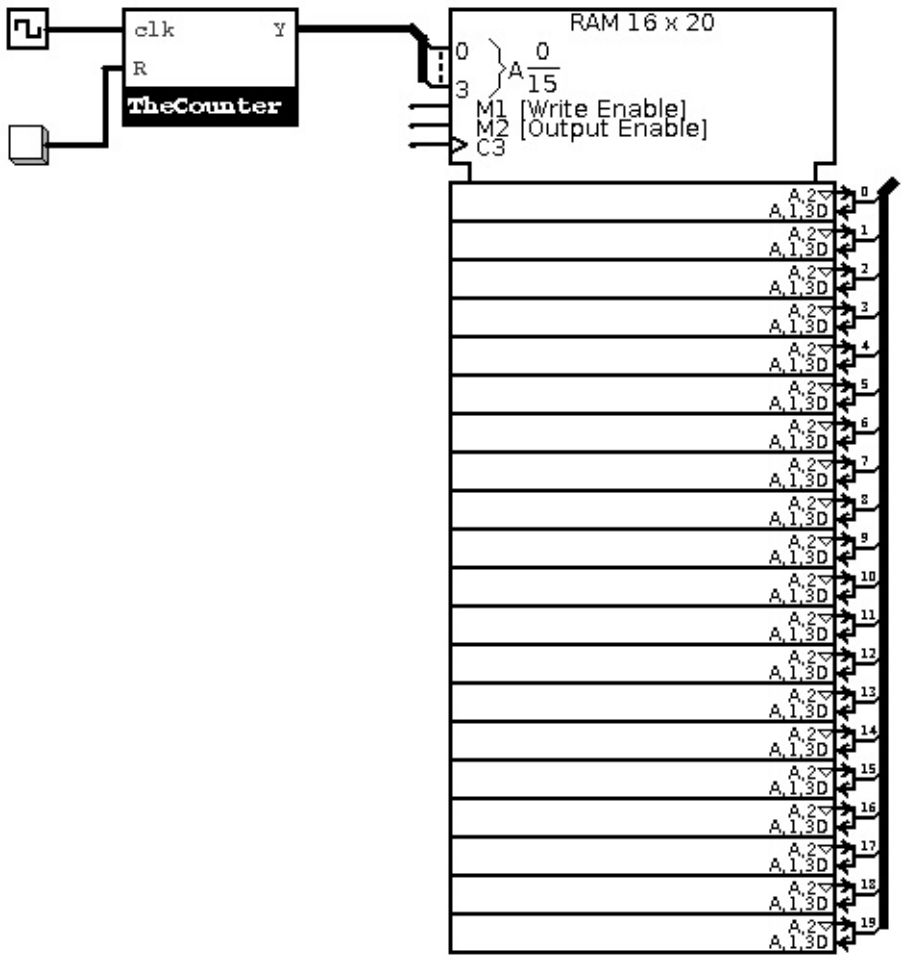
This one, is a bit harder to understand. But don't worry. We're just going to make the simplest ones. We will take a deeper look on this unit in following chapters (and maybe books!) but for this book, we make a simple one. Let's first define this unit. Control unit, controls programs loaded in our main memory, and controls what happens in ALU. Actually, control unit is made up of a device which is called *counter*. A counter is a register , or a bunch of registers which can count! How? the counting process is controlled by *clock pulse*. But why is it important? A counter can store address of a program, then its output can be input of the ***Address Bus*** of the RAM. This is the simplest use of a counter in a computer. But how can we make a counter? we can connect four D flip flops in this arrengement and make our counter :



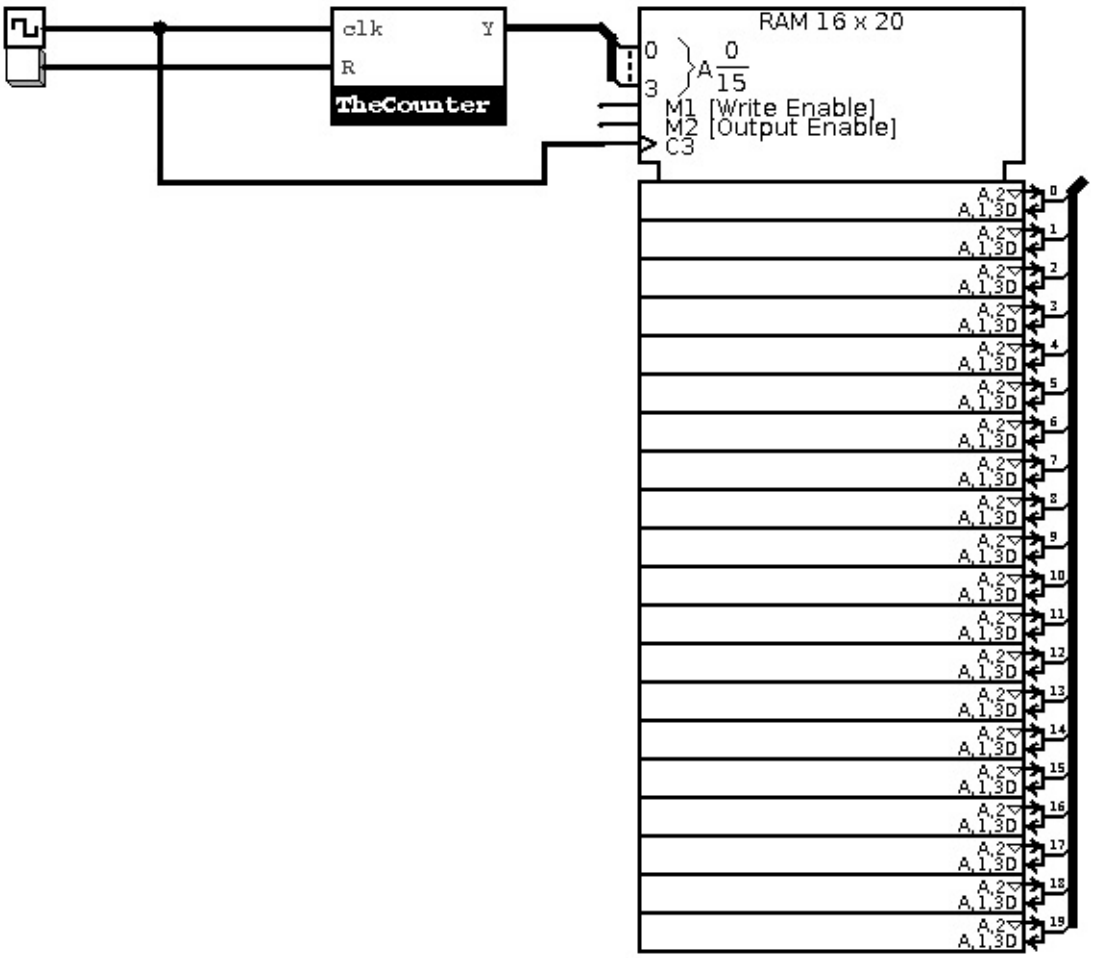
It has a common *reset* pin, and a clock input. Also, it has a 4-bit output labled *Y*. This is the simplest binary counter we can make. As it has four bits, it will count from 0 to 15 for us. This means we need a RAM block with 4-bit address bus. An actual control unit is not that simple, usually it includes some decoders, multiplexers and AND gates to control and activate/deactivate computer parts. But for our simple programmable computer, this one is enough!.

Memory Unit

We made a counter, which counts address for us. But, which address? Actually, we need a memory block like a RAM. If we look at RAM simply, it's very similar to register file. But, an actual and functional RAM is much more different. Let's add the RAM :



Now, we can store our programs in RAM! But wait, if you look closer at the picture above, you'll find that RAM is not connected to the clock. So it won't work. This is what you actually need :



Now, let's make the microcontroller!

Combination of Things!

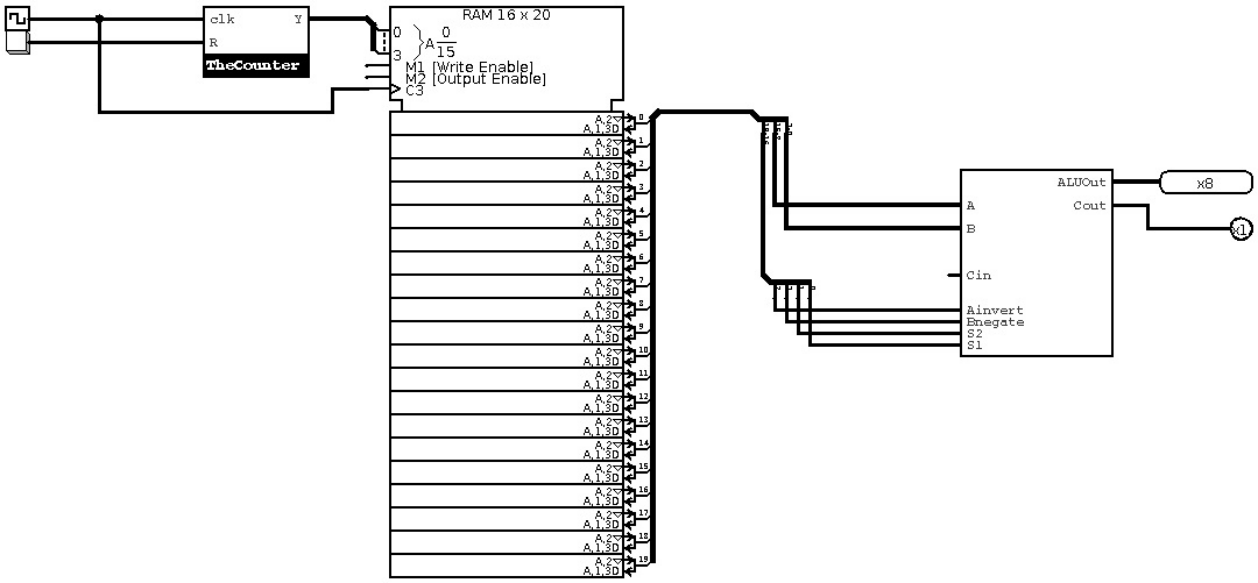
We made a bunch of devices, connected them together and made new devices, and finally, we made ALU. Let's connect the ALU to RAM. Remember this?

Instruction Code	Operand A	Operand B
4 bits	8 bits	8 bits

And now, we need a special device which is called *Splitter* to split parts we need. Then, we will need smaller splitters to split instruction code to bits. If you remember, we had this table for instruction codes :

Ainvert	Bnegate	S2	S1	Instruction
0	0	0	0	AND
0	0	0	1	OR
1	1	0	0	NOR
1	1	0	1	NAND
0	0	1	0	ADD
0	1	1	0	SUB

let's connect them together! And now, we have this :



What should we learn now?

After reading these fifteen chapters, you learned how to make your very own computer in gate-level. Now, you can learn programming and program your computer and test it, or you can learn digital electronics and make it in transistor-level. In following chapter, we will take a quick look at *assembly* language of our microcontroller, and we try to make some programs we need. It will help you understand how your real computer works!

Chapter 16 : Programming and Operating System

The most interesting part of computer engineering is the programming! Specially when you know how it works, you can make better programs. Now, we designed one, and we want to program it. But wait, let's take a look at computers made before.

Intel computers!

As you may remember from previous chapters, we had this for showing a simple character on console:

```
STK SEGMENT
 DW 32 DUP(?)
STK ENDS

CDS SEGMENT
 ASSUME CS:CDS, SS:STK
 MAIN PROC FAR
  MOV DL, 'A'
  MOV AH, 02
  INT 21h
  MOV AH, 4Ch
  INT 21h
 MAIN ENDP
CDS ENDS

END MAIN
```

And in chapter fourteen, we learned that this code will become ***Machine Code***. And now, we can assemble our own code, and write programs for that.

The assembler

Assembler is actually a software, used to generate machine code. Days we had no assembler, we had to generate machine code ourselves. This is hard, isn't it? But that's the only way when you have no operating system or other computer to assemble your code. Actually, a computer uses *Assembler* , then *Linker* to understand the programs. But in case of simple computers like ours, a simple assembler is enough. We just want to fill RAM blocks. The assembler sees our code. For example :

```
AND 10, 12
```

and then, as we have two 8 bit inputs, it converts the 10 to `000010101` and 12 to `00001100`. And it will look at this table :

Ainvert	Bnegate	S2	S1	Instruction
0	0	0	0	AND
0	0	0	1	OR
1	1	0	0	NOR
1	1	0	1	NAND
0	0	1	0	ADD
0	1	1	0	SUB

and finds that code for AND is `0000`. Then generates this `0x00A0C` as the final code! Let's write long numbers in hexadecimal. When we use hex, a 20-bit number will be a 5 digits number. So, we are the assembler in this case. Let's write our codes :

```
AND 10, 12
# Hex : 0x00A0C

ADD 3, 8
# Hex : 0x20308

SUB 10, 8
# Hex : 0x60A08
```

And this is a simple program in the machine language :

```
0000 00001010 00001100
0010 00000011 00001000
0110 00001010 00001000
```

Yes, computer only understands one and zero (please re-study chapter two, if you forgot this.) and we need to generate a bunch of ones and zeros to make a computer work! But, computers actually need a good interface between the system and the user. What's that?

The operating system

Operating System, is the interface! It connects the system to the user. Operating System is a software, which lets other software be installed and ran on your computer. If you have a PC in home, it may run Windows or Linux as the operating system. If you have a Mac, it may run macOS. And if you have an iPhone, it runs iOS. There are a lot of operating systems in the world, but a few operating systems are usable. Why? because other ones are only made for a special purpose. For example, MINIX is made for educational purposes. If you want to learn how UNIX works, you can study MINIX. But operating systems such as Linux, Windows or macOS are general-purpose. They're made to be the interface.And now you may ask , will we write an operating system for our computer? Sorry, No! If you pay attention to our assembly code, you will find this line :

```
MOV AH, 4Ch
INT 21h
```

The part including `INT` is actually an interrupt. We have no interrupts in our computer. This is the first problem. We also have no loops in our computer, in instruction set, we didn't define anything to make a loop and any recursive structure. 8086 and many other *real* processors, usually have instructions like `branch` or `jump`, and those instructions help us make real software, including operating systems! So, we never write operating system for our little computer.

Making a real computer?

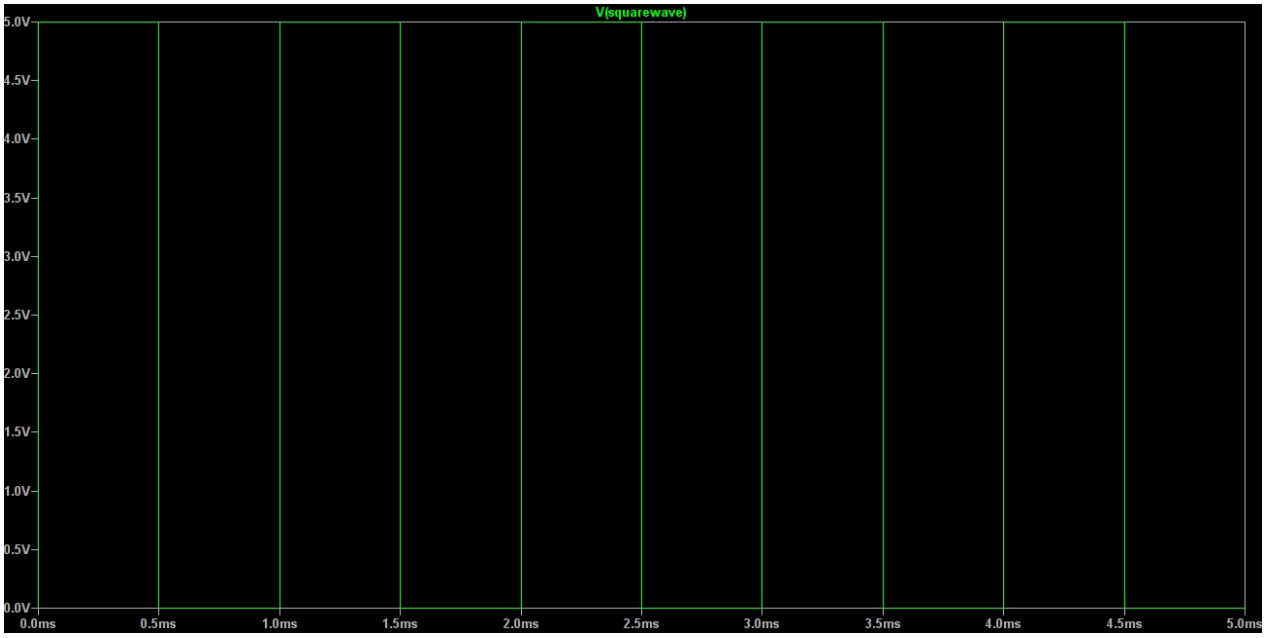
Now, you know a lot about computer architecture and organization. Also, you can make and program a computer in gate-level. This is necessary, trust me. You always need to know how computers work, otherwise you won't be able to make software. But I'm sure you're curious about real hardware design. The next chapter (and the last one!) is about that. You will learn how companies design and produce their own hardware.

Chapter 17 : The Dark Side of The Moon

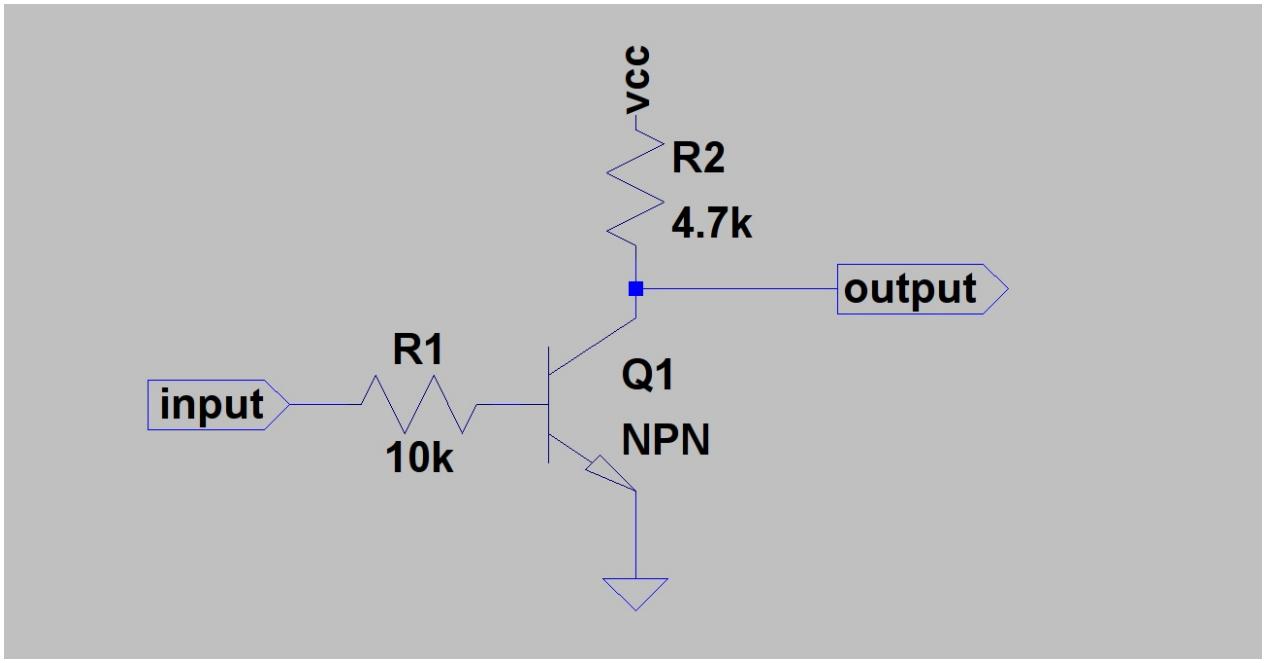
This is not about Pink Floyd's album, this book has nothing to do with music of course. But, I chose this name because a lot of *computer engineers* have no idea about design and implementation of the hardware. So, it can be the dark side! Some people, including some computer engineers and electrical engineers, joined the dark side and made computers. But this is not all! We had computers before transistors. They were usually electro-mechanical or mechanical. But the theory of computation is much older than what you think! ***Khawrazmi***, a Persian scientist, is one of the most known people in history of computer science. Even the word ***Algorithm*** is taken from his name. In 1947, *Transistor* discovered, and that was the starting point of ***Electronic Computers***.

Digital Electronics

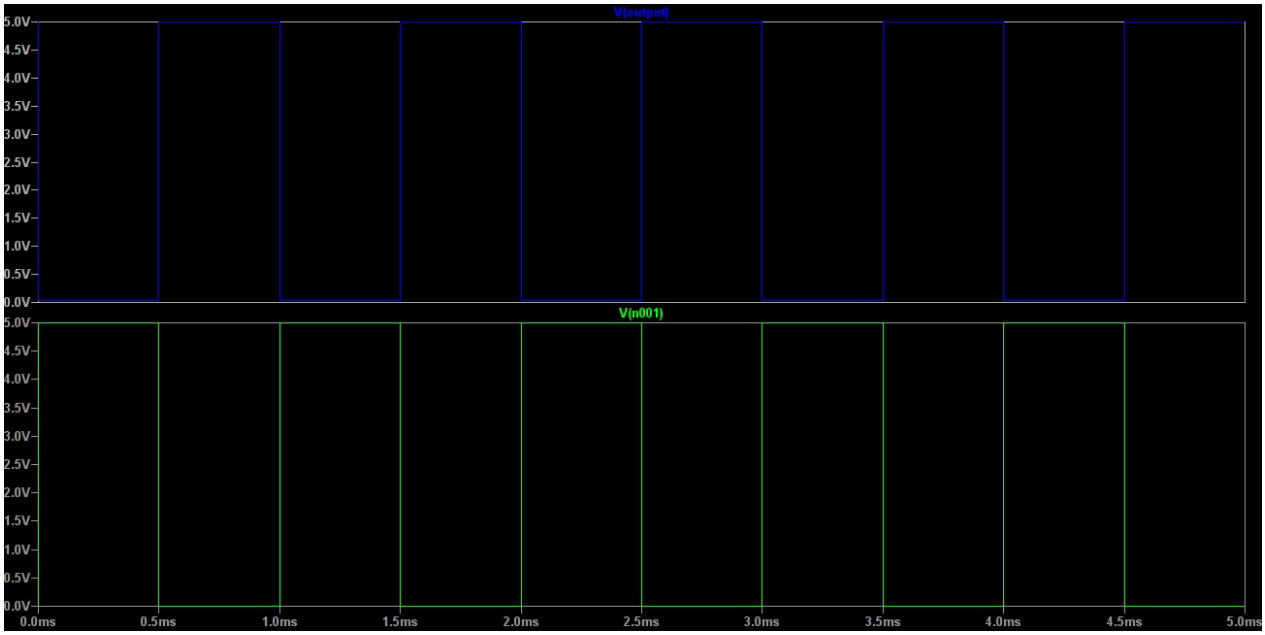
When transistors discovered and manufactured, scientists and engineers found that they can acts like switches. Then, they decided to use them as switches. A transistor can act like a switch, but first, look at this square wave:



It starts at zero volts and ends at five volts. It's like a switch which is connected to a five volts battery, and someone pressed button in defined times. This pulse, can be input or output of a transistor. First, let me show you a NOT gate which is made by a ***Bipolar Junction Transistor*** or in short, **BJT** transistor. BJT transistors are like two diodes connected in a specific arrengement. Usually, we use *NPN* transitors to make logical circuits:



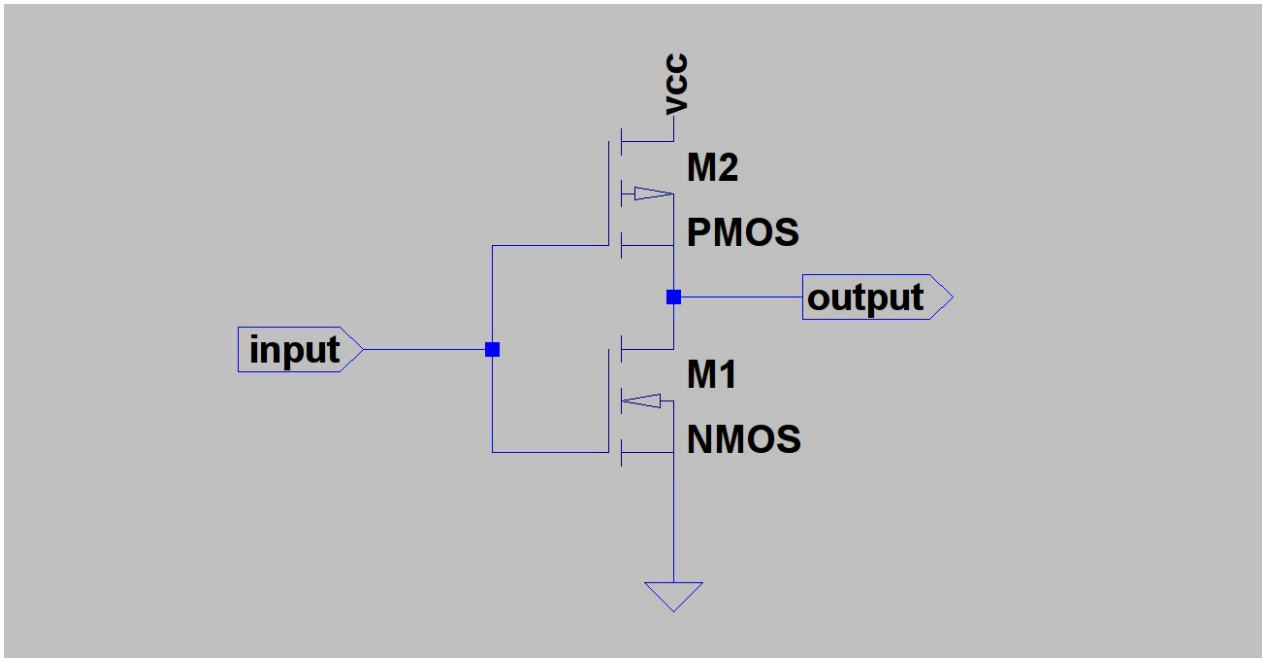
If we apply a voltage on *input* we will have a ***wave form analysis*** like this :



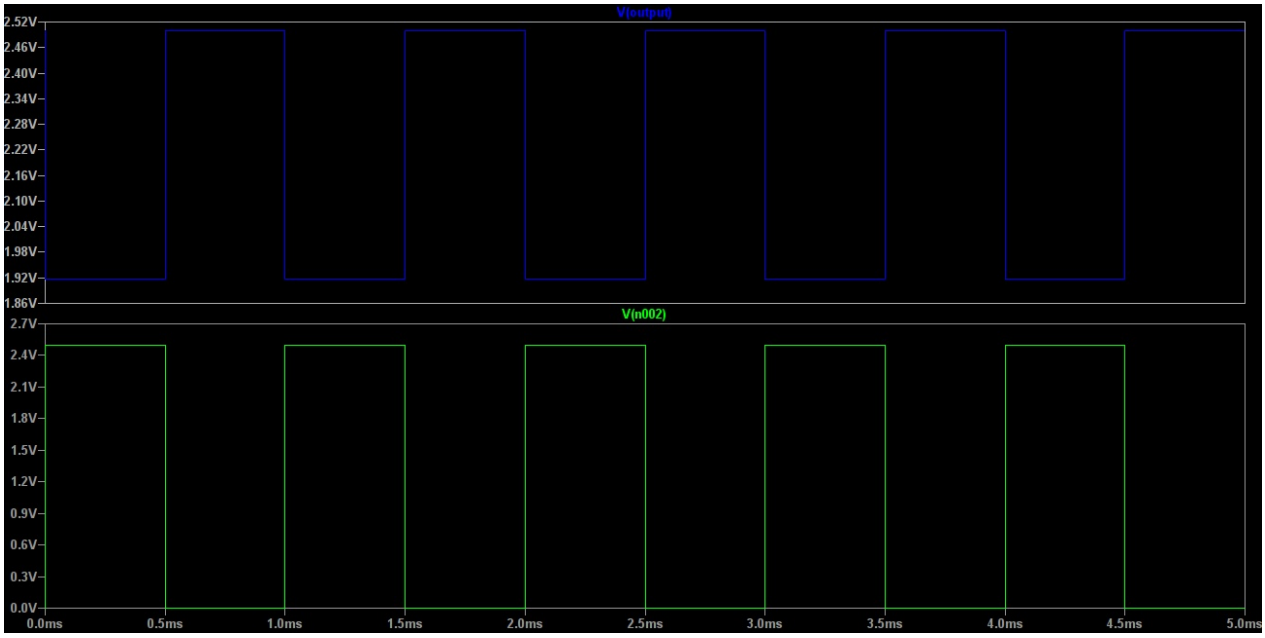
As you can see, everytime input is high, output is low, and everytime input is low, output is high. Why? when *input* is high, current can move to the ground, but when it's low, current find another way to the ground, which is our output.

RTL/TTL technologies are good, but later engineers found that ***Field Effect Transistors*** are better. Then, they decide to use ***Metal Oxyde Semiconductor Field Effect Transistor***s or in short **MOSFET** transistors for their artworks! FETs have different structure, but still can act as switch, and even better than BJTs. Also, they can be minimalized better. This is why a lot of digital chips around us is made up of MOSFETs. MOSFETS are usually two types, nMOS, or ***Negative Channel*** and pMOS or ***Poisitive Channel***. As engineers wanted the best

performance, they just used both of them. When you use both nMOS and pMOS transistors, you actually used ***Complementary Metal Oxyde Semiconductor*** or *CMOS* technology. This is a CMOS inverter (or the same NOT gat):



Let's apply voltage on input, and see what happens!



As you can see, the result is very similar to TTL/RTL design. When you design a digital circuit, you should consider everything! Waste of power, price, noise-resistance and size. For example, TTL is a good design as it's cheap and has good resistance against noise, but it's not as small as we want. But CMOS is small enough to be on a chip! Also, there's a method for minimalizing CMOS designs, and implementation of more logics by less transistors, which is called ***Very Large Scale Integration*** or in short, **VLSI**. I think this is enough, let's take a look on other ways of implementing a computer in real life.

Integrated Circuits

Imagine a rectangular black thing, which includes twenty transistors. This is actually an integrated circuit. Integrated circuits are made up of tiny transistors, and they easily can be find in electronics stores. They integrated a lot of circuits on a piece of semiconductor, connected metal legs to that, and covered with plastic, then you can buy them and use them in your projects. Also, IC's classified by their technology. For example, you know 7400 series are TTL IC's, or 28 Series are EEPROM's. Anyway, there is one other way, and you may find that the easiest way to join the dark side!

Program The Hardware

There are some programming languages, which are called ***Hardware Description Language*** or in short, **HDL**. They're easy to use, and they're actually similar to *C* or *Assembly* programming languages. One of my favorite HDL's is *Verilog*. I program an AND gate in HDL like this :

```
module AND(A, B, F);
 input wire A;
 input wire B;
 output wire F;

 assign F = A & B;

endmodule
```

And for that, we need a ***Complex Programmable Logical Device***, CPLD or a ***Field-Programmable Gate Array***, FPGA. Those devices have logical applications, but they have no logics, and we have to program them to make them functional. A lot of ligical simulator programs, like *logisim* can generate Verilog or VHDL code of the designed logical circuit. And then, you can upload the code on your FPGA and have your very own logical device. And any other way to design and implement a real hardware? Of course there is a lot of other ways. But these ways are the easiest ways to do that with a low-budget (not sure of this!) and basic knowledge of computers and electronics.

The last part!

This is the last part of the book! I tried to do my best for this one. Actually, this is my first English book, but I'm glad, because I could share my knowledge and experiences. This book is written to make understanding a computer easier. First, I started it in Summer (2016) and

it finished in Spring (2017). About 8 or 9 months spent on this book, and I hope you enjoy!

Bibliography

1. But How Do It Know? - The Basic Principles of Computer for Everyone by **J.Clark Scott**
2. Computer Organization and Design, The Hardware/Software Interface, Revised Fourth Edition by **Davide Patterson**, **John Henessy**, **Morgan Kaufman**
3. Logic and Computer Design Fundamentals, Fifth Edition by **Morris Mano**
4. Digital Design: With an Introduction to the Verilog HDL by **Morris Mano**