

CONTENTS

CHAPTER 0 ► ABOUT THESE NOTES **PAGE 1**

0.1	License	1
0.2	Bibliography and references	2

CHAPTER 1 ► NUMERIC SYSTEMS **PAGE 3**

1.0.1	Conversion from Decimal to Binary	3
1.0.2	Conversion from Binary to Decimal	4
1.1	Hexadecimal system	4
1.1.1	Conversion from Binary/Decimal to Hexadecimal	5
1.1.2	Conversion from Decimal to Hexadecimal	5
1.1.3	Conversion from Hexadecimal to Binary/Decimal	5
1.2	Bits, Bytes and Nibbles	6
1.3	Addition between binary numbers	6
1.4	The sign of the binary numbers	6
1.4.1	“Taking the two’s complements” method	7
1.5	Increasing Bit Width	8
1.6	Binary Multiplication	8
1.7	Logical and Arithmetic Shift	8
1.7.1	Logical Left Shift ($A \ll N$)	8
1.7.2	Logical Right Shift ($A \gg N$)	9
1.7.3	Arithmetic Shifts ($A \ggg N$)	9
1.8	Fractional Numbers	10
1.8.1	Fixed-point notation	10
1.8.2	Floating point notation	11
1.8.3	Addition with floating point numbers	13
1.8.4	Multiplication with floating point numbers	14
1.8.5	Subtraction with floating point numbers	14
1.8.6	Division with floating point numbers	14
1.8.7	Approximation of binary numbers	14

CHAPTER 2 ► LOGICS **PAGE 15**

2.1	Logic gates	15
2.2	Combinational and Sequential logic	16

2.3	Boolean algebra	17
2.3.1	Canonical forms (SOP and POS)	17
2.3.2	Axioms and theorems	18
2.3.3	Proof of the theorems	19
2.3.4	Simplifying an equation	19
2.3.5	Theorem 11 - Consensus	20
2.3.6	Converting from POS to SOP	20
2.3.7	Converting from SOP to POS	21
2.3.8	Theorem 12 - DeMorgan's Theorem	21
2.4	Bubble pushing	22
2.5	Completeness	22
2.5.1	From SOP/POS forms to NAND-NAND or NOR-NOR	22
2.6	From Logic to Gates	23
2.6.1	Don't Cares	23
2.7	Circuit schematics rules	24
2.8	X and Z	24
2.9	Karnaugh Maps (K-Maps)	25
2.10	Quine-McCluskey algorithm	26
2.11	Multilevel Combinational Logic	26
2.12	Combinational building blocks (MUX, DEMUX, Decoders)	27
2.13	Timing of combinational circuits	28
2.14	Glitches	29

CHAPTER 3 ► **SEQUENTIAL CIRCUITS** **PAGE 30**

3.1	Latches and Flip-Flops	30
3.1.1	SR Latches	31
3.1.2	D Latches	32
3.1.3	D Flip-Flop	32
3.1.4	Registers	33
3.1.5	Resettable, Settable and Enabled flip-flops	34
3.2	Synchronous Logic Design	34
3.3	Finite State Machines (FSM)	35
3.3.1	FSMs Initialization and Unreachable States	36
3.3.2	FSM Encoding	37
3.3.3	Moore vs Mealy FSMs	37
3.3.4	Glitches	38
3.3.5	From Moore to Mealy	39
3.3.6	From Mealy to Moore	39
3.4	Deriving a FSM from a schematic	39
3.5	Timing	40
3.6	Clock Skew	42
3.7	Parallelism	42

CHAPTER 4 ► DIGITAL BUILDING BLOCKS **PAGE 44**

4.1	Arithmetic Combinational circuits	44
4.1.1	1-Bit and <i>N</i> -Bit Adders	44
4.1.2	Subtractors and Comparators	47
4.1.3	Arithmetic Logic Unit (ALU)	48
4.1.4	Shifters and Rotators	50
4.1.5	Multipliers	51
4.2	Sequential circuits	51
4.2.1	Counters	51
4.2.2	Shift registers	52
4.3	Memory	53
4.3.1	RAM	54
4.3.2	Memory trade-offs and ports	54
4.3.3	ROM	54
4.3.4	PLAs and FPGAs	55

CHAPTER 5 ► HARDWARE DESCRIPTION LANGUAGES **PAGE 57**

CHAPTER 6 ► COMPUTER STRUCTURE **PAGE 58**

6.1	Memory	58
6.2	CPU	59
6.3	Programs and instructions	60
6.4	Architecture of the CPU	63
6.4.1	CPU's main components	64
6.4.2	Further developing the architecture	65
6.4.3	ALU control and Main Control Unit	68
6.5	Pipeline	69
6.6	Caches	72
6.7	Virtual Memory	75
6.8	Parallelism	76

Chapter 0

About these notes

0.1 License

This work is licensed under a [Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International”](#) license.



The decision of copyrighting this work was taken since these notes come from **university classes**, which are protected, in turn, by the **Italian Copyright Law** and the **University's Policy** (thus Sapienza Policy). By copyrighting these works I'm **not claiming as mine** the materials that are used, but rather the creative input and the work of assembling everything into one file.

All the materials used will be listed here below, as well as the names of the professors (and their contact emails) that held the courses.

The notes are freely readable and can be shared, but **can't be modified**. If you find an error, then feel free to contact me via the socials listed in my [website](#). If you want to share them, remember to **credit me** and remember to **not** obscure the **footer** of these notes.

Those notes were made during my first year of university at Sapienza. These notes **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my [website](#). Good luck!

Leonardo Biason

0.2 Bibliography and references

- [1] Sarah Harris, David Harris. (2015). *Digital Design and Computer Architecture, ARM Edition*. Morgan Kaufmann, Elsevier
- [2] David Patterson and John Hennessy. (2020). *Computer Organization and Design RISC-V Edition (Second Edition)*. Morgan Kaufmann, Elsevier

Professor's name	Contact	Teaching Course
Daniele De Sensi	desensi@di.uniroma1.it	Computer Architecture Unit 1 (2022-2023) Chapters 1 - 5
Alessandro Mei	alessandro.mei@uniroma1.it	Computer Architecture Unit 2 (2022-2023) Chapter 6

Inside those notes, there are multiple boxes, and each of them has a different meaning. Based on the color of their stripe on the left, boxes can be separated into 4 macro categories

Theory boxes	Practice boxes
Remarks boxes	Curiosities boxes

Chapter 1

Numeric systems

In computers, everything works with 0 and 1, mainly because other number systems would **need more voltage** (while computers work with 2 types of voltage). But what is a number system? Let's take for example the following number:

$$5374_{10}$$

This means that the number is expressed in the decimal system (because of the base). There are many number systems outside the decimal system, for example the binary or the hexadecimal system. An example of binary number could be:

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

With an n -digit decimal number, **how many** numbers can we write? We can write 10^n numbers, and the **range** goes from 0 to $10^n - 1$. For the N -bit binary numbers however, we can represent 2^n numbers, and the **range** goes from 0 to $2^n - 1$.

1.0.1 Conversion from Decimal to Binary

To convert a decimal number to a binary number:

1. We take the highest power of 2 that is smaller than the number that we want to convert;
2. We then subtract the number that we found to the first number. We then repeat the same thing over and over until we get to 0;
3. If some bases are jumped (so are not taken into account) then we write 0 in place of those numbers. If we have a base then we write 1. The number is written in the opposite way of the base obtaining (first the last digits)

Example 1.0.1

$$\begin{aligned} 53_{10} &\Rightarrow 53 - (32 \times 1) \rightarrow 2^5 \times 1 \\ 53 - 32 &= 21 \rightarrow 21 - (16 \times 1) \rightarrow 2^4 \times 1 \\ 21 - 16 &= 5 \rightarrow 5 - (4 \times 1) \rightarrow 2^2 \times 1 \\ 5 - 4 &= 1 \rightarrow 1 - (1 \times 1) \rightarrow 2^0 \times 1 \end{aligned}$$

We get 110101 because:

- the powers are 25, 24, 22, 20, so we place a 1 in the 1st, 2nd, 4th and 6th place;

- the gaps are at 23 and 21, so we place a 0 in the 3rd and 5th place

1.0.2 Conversion from Binary to Decimal

In order to convert a binary number into a decimal number:

- We multiply, starting from the right bit, every bit for a power of 2 (starting from 20), that increases by 1 for every bit;
- We then make an addition between all the numbers that we obtained.

Example 1.0.2

$$11011_2 \Rightarrow 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 27_{10}$$

1.1 Hexadecimal system

Since writing in binary could be tedious and lead to various errors, there is also the **hexadecimal** system, that has ciphers that go from 0 to 15 (for a total of 16 ciphers). However, since there aren't enough arabic ciphers, the **first 6 letters** of the alphabet are used instead. Moreover, the hexadecimal system has more possibilities, since it works with the **powers of 16** instead of the powers of 2.

0 1 2 3 4 5 6 7 8 9 A B C D E F

Each cipher in the hexadecimal corresponds to 4 bits, so for example the number 0110 1011 in the hexadecimal system is 6B₁₆ or 0x6B. An hexadecimal number can be shown in 2 ways:

- Either with the base XX₁₆ at the end (for example 5DB913A8₁₆)
- Or with the prefix 0x (for example 0x6FC380DA or 0x4AF)

How do we convert an hexadecimal number into both binary numbers and decimal numbers? Let's take for example 0x000004AF: we would have to convert each letter into a number and then, according to its position, **multiply** the cipher for a power of 16:

$$4 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 1199_{10}$$

The conversion of the ciphers follows the following table (Decimal₁₀ / Hexadecimal₁₆):

0 ₁₀ / 0 ₁₆	1 ₁₀ / 1 ₁₆	2 ₁₀ / 2 ₁₆	3 ₁₀ / 3 ₁₆	4 ₁₀ / 4 ₁₆	5 ₁₀ / 5 ₁₆	6 ₁₀ / 6 ₁₆	7 ₁₀ / 7 ₁₆
8 ₁₀ / 8 ₁₆	9 ₁₀ / 9 ₁₆	10 ₁₀ / A ₁₆	11 ₁₀ / B ₁₆	12 ₁₀ / C ₁₆	13 ₁₀ / D ₁₆	14 ₁₀ / E ₁₆	15 ₁₀ / F ₁₆

1.1.1 Conversion from Binary/Decimal to Hexadecimal

We know that for each 4 bits in the binary system there is an equivalent hexadecimal cipher, so in order to convert a number from a binary system to the hexadecimal system:

- 1) We divide the number into groups of 4 bits starting from the right (the **Least Significant Bit** (LSB));
- 2) We convert each group of 4 bits into an hexadecimal number;
- 3) We put all the hexadecimal ciphers together.

Example 1.1.1

$$\begin{array}{ccc}
 \alpha & \beta & \gamma \\
 11010110101_2 \Rightarrow 110_2 \ 1011_2 \ 0101_2 = \\
 \alpha) 110_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6_{16} \\
 \beta) 1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = B_{16} \\
 \gamma) 0101_2 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{16}
 \end{array}$$

We obtained 6_{16} , B_{16} and 5_{16} . The result is $0x6B5$ or $6B5_{16}$ (depending on the notation that is taken in account)

1.1.2 Conversion from Decimal to Hexadecimal

tbd

1.1.3 Conversion from Hexadecimal to Binary/Decimal

To convert a hexadecimal number into a binary number we must first convert each hexadecimal cipher into the corresponding binary number, then place each 4-bit group in order

Example 1.1.2

$$\begin{array}{ccc}
 \alpha & \beta & \gamma \\
 3A8_{16} \Rightarrow 3_{16} \ A_{16} \ 8_{16} = \\
 \alpha) 3_{16} = 0011_2; & \beta) A_{16} = 1010_2; & \gamma) 8_{16} = 1000_2 \Rightarrow 001110101000_2
 \end{array}$$

Although, if we aim to convert a hexadecimal number into a decimal number the steps are different:

- 1) Starting from the cipher at the right, we convert every hexadecimal cipher into a decimal cipher;
- 2) For each cipher, starting from the right, we multiply the cipher for a power of 16 with an increasing exponent for each cipher (starting from 16⁰ and then following with 16¹, then 16², etc...);
- 3) We then make the sum of all the products that we obtained from Step 2.

Example 1.1.3

$$3_{16} \Rightarrow 3_{10} \quad A_{16} \Rightarrow 10_{10} \quad 8_{16} \Rightarrow 8_{10}$$

$$3A8_{16} \Rightarrow (3_{10} \times 16^2) + (10_{10} \times 16^1) + (8_{10} \times 16^0) = 768_{10} + 160_{10} + 8_{10} = 936$$

1.2 Bits, Bytes and Nibbles

A **bit** in the binary system is a cipher. In a number like 10010110_2 :

- the 1 at the left is the **Most Significant Bit** (MSB);
- the 0 at the right is the **Least Significant Bit** (LSB).

A number composed of **8 bits** it's called a **byte** (so 10010110_2 it's a byte), and a byte can be split into two subgroups of **4 digits**: the **nibbles** (for example 1001 1011)

1.3 Addition between binary numbers

The operations with binary digits work a bit differently than with decimal numbers. Let's take for example $1011+0011$. The result will be 1110 because:

$$\begin{array}{rcccc} 1 & 0 & 1 & 1 & + \\ 0 & 0 & 1 & 1 & = \\ \hline 1 & 1 & 1 & 0 & \end{array}$$

All the possible results between binary numbers are the following:

$0 + 0 = 0$	$0 + 1 = 1$	$1 + 0 = 1$	$1 + 1 = 0$ with an additional 1 brought as a carry
-------------	-------------	-------------	--

When doing operations with binary numbers, it is possible to get too large numbers, that sometimes computers may not be able to handle. In such cases we talk about **overflow**.

Definition: Overflow

Digital systems operate on a fixed number of digits, so when the result is too big to fit in the available number of bits there is an overflow. Normally an overflow occurs when there is a carry out of the most significant bit's column

1.4 The sign of the binary numbers

Decimal numbers can be both negative and positive, but what about binary numbers? There are two ways of writing signed numbers:

- 1) **The "Sign/Magnitude" notation:** there can be a 1-sign bit (the most significant one) that represents the sign. If:

- the digit is 0 then the number is **positive**;
- the digit is 1 then the number is **negative**.

There are although two issues with Sign/Magnitude numbers, which are the following:

- ordinary addition won't work, since the MSB would be considered as an effective bit and not as a sign bit;
- that +0 and -0 are written differently: +0 is 0, while -0 is 1, which is nonsense;

2) **The “Two’s Complement” notation:** the MSB has a value of -2^{n-1} , such that 0 has only one representation (0000_2) and ordinary additions work.

With this notation the most positive number has a 0 in the MSB (since the MSB still represents the sign, it still keeps a specific value; in the Sign/Magnitude notation it had no value) and all 1's elsewhere, while the most negative number has 1 in the MSB and all 0's elsewhere. It's obvious that the same number it's different for each notation:

$$\begin{array}{ccc}
 \text{S/M} & & \text{TsC} \\
 100101_2 & \neq & 100101_2 \\
 \Downarrow & & \Downarrow \\
 -5_{10} & \neq & -27_{10}
 \end{array}$$

1.4.1 “Taking the two’s complements” method

When there is the need of making operations with negative binary numbers we have to make every number positive; the process to invert a number's sign is called “Taking the two's complements”. Following this method, in order to invert a number's sign:

- 1) First invert all the bits in the number, so that each 0 becomes a 1 and vice versa;
- 2) After inverting the numbers you must make the sum between the inverted number and 1.

It's important to remember that this is useful to find the magnitude of a negative number, but it's not equal to the same negative number. Every time that a number is inverted in order to do an operation then it must be inverted again so that the real sign will be shown. A trick to check the result of the inversion of a number consists on looking at the MSB: the result **must** start with the MSB that is the opposite of the starting number (if it starts with 0 the flipped number must start with 1 and vice versa)

Example 1.4.1

$$\begin{array}{ccc}
 \text{Normal number} & & \text{Inverted number} \\
 1101010_2 & \Rightarrow & 0010101_2 + 1_2 = 0010110_2
 \end{array}$$

When summing 2 equal numbers of opposite signs the result will always be 0; if there is an overflow it doesn't matter, since overflows that happen because of carried extra bits gets automatically discarded. There could be an overflow though if you sum two

numbers of the same sign and you get a number of opposite sign.

There is a number for which there doesn't exist a positive representation, and this number is 1000. It's a weird number because if we flip it, we get the following:

$$1000_2 \implies 0111_2 + 1_2 = 1000_2$$

If we had more bits to express a binary number, then it would be possible to represent -8 (so 1000_2), but then the same problem would be presented if, given a bit width of n (with $n > 1$), we would've had a number with the MSB equal to 1 and $n - 1$ zeroes after it.

1.5 Increasing Bit Width

If we want to extend a signed number from N bits to M bits (with $M > N$) we have to:

- 1) Copy the sign bit to the MSBs
- 2) Be sure that the number remains the same.

Example 1.5.1

4-Bit number \implies 8-Bit number

$$0011 \implies 0000\ 0011$$

$$1100 \implies 1111\ 1100$$

1.6 Binary Multiplication

Binary multiplication works pretty similar to decimal multiplication, since you have to multiply each bit as in the decimal multiplication:

$$\begin{array}{r} 1100 \times \\ 011 = \\ \hline 1100 + \\ 1100/ + \\ 0000// = \\ \hline 100100 \end{array}$$

1.7 Logical and Arithmetic Shift

They are useful to multiply numbers or to divide binary numbers. It does work with unsigned and "two's complements" numbers.

1.7.1 Logical Left Shift ($A \ll N$)

You have to move A to the left by N positions. It is equal to $A \ll N = A \times 2^N$, and by doing so the value of the number **increases**, since the ciphers are moving towards the MSB. Be careful that the number can be shifted without having to discard important ciphers. **Overflow** may occur with **2's Complements numbers**, specifically if the number goes

beyond the range of allowed numbers (for example, a 4-bit number has a range of possible numbers going from -8 to 7, so if we shift a number like -5, we'll end up with 10. This is an example of arithmetic overflow).

Example 1.7.1

01010 \ll 3

1st shift) 10100

2nd shift) 101000

3rd shift) 1010000

Note: the **red numbers** are the one shifted that get discarded because of **overflow**

$(10 \times 2^3 = 80)$

1.7.2 Logical Right Shift ($A \gg N$)

You have to move A to the right by N positions.

Example 1.7.2

10110 \gg 3

1st shift) 01011

2nd shift) 00101

3rd shift) 00010

1.7.3 Arithmetic Shifts ($A \ggg N$)

You have to move A to the **right** (excluding the sign bit) by N positions, **extending** with the **sign bit**. **Arithmetic Left Shift** has **no sense** since it will **change the sign**, so it's not typically used. Doing an Arithmetic Right Shift is equal in doing $A \ggg N = A/2^N$, since moving numbers away from the **MSB** makes them smaller.

Example 1.7.3

01000 \ggg 2

1st shift) 00100

2nd shift) 00010

$(8/2^2 = 2)$

10000 \ggg 2 \Rightarrow 11100

$(-16/2^2 = -4)$

1.8 Fractional Numbers

There are two common notations for fractional numbers:

- 1) **Fixed-point notation**: it has a **restricted** range of values, although it's **faster to compute**, it's more **power-efficient** and it's **easier to implement** in hardware;
- 2) **Floating point**: the binary **point floats** to the **right** of the **most significant 1**; it allows a **wider range of values**, but it's **slower to compute** and it asks for **more resources**.

Unsigned and **Two's Complements** numbers approach this in two different ways:

- **Unsigned numbers**: With N bits we can represent 2^n values, with the range being $[0, 2^n - 1]$. To represent a value A , we need $\lfloor \log_2(A) \rfloor + 1$ bits;
- **Two's Complements numbers**: with N bits we can represent $2^n - 1$ values, with the range being $[-2^{n-1}, 2^{n-1} - 1]$. To represent a value A , we need $\lfloor \log_2(|A|) \rfloor + 2$ bits (only 1 bit more if $A = -2^{n-1}$).

1.8.1 Fixed-point notation

A) From binary to decimal:

- Let's say we want to use 4 integer bits and 4 fraction bits;
- we first divide the binary number into 2 groups of 4 bits;
- we then compute first the integer bits, then the fraction bits. For the fraction bits we start from 2^{-1} and we add -1 at the exponent for each fraction bit

Example 1.8.1

$$\begin{array}{ccc}
 \alpha & & \beta \\
 0110 \ 1100_2 & \Rightarrow & 0110.1100_2 \\
 & & \alpha \quad \beta \\
 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6 + 0,75 = 6,75 \\
 \text{With } \alpha \text{ as the integer bits and } \beta \text{ as the fraction bit}
 \end{array}$$

B) From decimal to binary (**Method 1**):

Example 1.8.2

$$\begin{array}{l}
 7,5_{10} \Rightarrow 7_{10} + 0,5_{10} \\
 \text{Integer: } 7_{10} = 0111_2 \\
 \text{Fraction: } 0,5_{10} = 1000_2 \\
 0111.1000_2
 \end{array}$$

C) From decimal to binary (**Method 2**):

- 1) Separate the integer bits from the fraction bits;
- 2) For the integer part the computation follows the standard computation algorithm;

- 3) consider the fractional bits as a variable $p = 2p$;
- 4) put $p > 1$:
 - * if true then write 1 as one of the ciphers of the fractional part and then put $p = p - 1$ and repeat steps 3 and 4 until $p = 0$;
 - * if false ($p < 1$) then write 0 and repeat the steps 3 and 4 until $p = 0$;
- 5) put all the ciphers together.

Example 1.8.3

$$\begin{aligned}
 &7,5625_{10} \Rightarrow p = 0,5625_{10} \\
 p = 2p &\Rightarrow 0,5625_{10} \times 2 = 1,125_{10} \geq 1 \Rightarrow \text{true, then we save 1} \\
 &\quad p = 1,125_{10} - 1_{10} = 0,125_{10} \\
 p = 2p &\Rightarrow 0,125_{10} \times 2 = 0,25_{10} \geq 1 \Rightarrow \text{false, then we save 0} \\
 p = 2p &\Rightarrow 0,25_{10} \times 2 = 0,50_{10} \geq 1 \Rightarrow \text{false, then we save 0} \\
 p = 2p &\Rightarrow 0,50_{10} \times 2 = 1_{10} \geq 1 \Rightarrow \text{true, then we save 1} \\
 &\quad p = 1_{10} - 1_{10} = 0, \text{ thus we finished. The result is } 0111.1001_2
 \end{aligned}$$

1.8.2 Floating point notation

Similarly to the scientific notation, the binary point floats to the right of the most significant 1. Every number in scientific notation is composed by a mantissa, a base and an exponent.

$$273_{10} = 2,73_{10} \cdot 10^2$$

In order to write floating points numbers, we use the **IEEE 754 floating-point standard**, described by the following table:

Sign	Exponent	Mantissa
1 bit	8 bits	23 bits

How do we write numbers in such format?

- 1) Convert decimal to binary;

$$228_{10} \Rightarrow 1110 \ 0100_2$$

- 2) Write the number in binary scientific notation (with the point at the right of the most significant 1);

$$1110 \ 0100_2 = 1.11001_2 \times 2^7$$

- 3) Fill in each field of the 32-bit floating point number. The first number of the mantissa is **always 1**, although **we don't write it** in the **mantissa**, since we know that every mantissa will start with 1. When converting back the number from the IEEE 754 Standard to another form we must remember to add again the "1.".

1	100 0010 0	110 1001 0000 0000 0000 0000
---	------------	------------------------------

- 4) How do we represent negative exponents? Via a biased exponent (we sum the bias to the exponent):

- a) The bias is equal to **127** ($0111\ 1111_2$);
 b) The exponent will be then (from the previous example):

$$127 + 7 = 134 = 1000\ 0110_2$$

- **Theory:**

- c) Biased exponents in the range $[0, 126]$ are smaller than 0;
 d) Biased exponents in the range $[127, 255]$ are instead greater or equal to 0;
 e) The range of possible exponents goes from -126 to 127 (then $[-126, 127]$);
 f) The formula to convert back from the IEEE 754 Standard is:

$$(-1)^S \times 2^{(132-127)} \times 1.F$$

Example 1.8.4

1	100 0010 0	110 1001 0000 0000 0000 0000
---	------------	------------------------------

$$\begin{aligned}
 &(-1)^1 \cdot 2^{(132-127)} \cdot 1.1101001 \Rightarrow \\
 &\Rightarrow (-1) \cdot 2^5 \cdot 1.1101001 \Rightarrow \\
 &\Rightarrow -1.11101001 \cdot 2^{-2} \Rightarrow \\
 &\Rightarrow -0,25 \cdot 233 = -58,25_{10}
 \end{aligned}$$

There are some exceptions though, mainly because of the removal of the first 1 of the mantissa. The exceptions are listed here:

Number	Sign	Exponent	Fraction
0	Any	00000000	000000000000000000000000
∞	0	11111111	000000000000000000000000
$-\infty$	1	11111111	000000000000000000000000
NaN	Any	11111111	non-zero
Denormals	Any	00000000	$F \neq 0 \Rightarrow (-1)^S \times 2^{-126} \times 0.F$

For NaN (Not a Number) we intend numbers such as $\sqrt{-1}$. Since it doesn't exist, the computer evaluates it as NaN.

When a number is so small that requires higher computational power than the ones available from the computer, an **underflow** occurs. To express such numbers, the IEEE 754 standard gives the possibility to represent such numbers as **denormal numbers**. Those numbers' **mantissa isn't** 1 as in all the other numbers, but **is instead** 0. This is

made in order to **avoid underflow** and to reduce the loss of precision.

We saw that the IEEE 754 standard uses **32 bits-wide numbers**; those numbers are called **single-precision numbers**. The standard also defines **double-precision numbers** (with **64 bits**) and **half-precision numbers** (**16 bits**).

Name	Sign Bits	Exponent	Mantissa	Bias
Single-precision (32 bits)	1	8	23	127
Half-precision (16 bits)	1	5	10	15
Double-precision (64 bits)	1	11	52	1023

There are 4 methods to round the numbers:

- 1) Down (with the floor function);
- 2) Up (with the ceiling fraction);
- 3) Towards zero (truncation);
- 4) To the nearest (the default method)

If a number is too big to be represented **overflow** occurs: the number gets rounded to the **nearest**, which is ∞ . It can also occur that a number can't be represented because it's too small: we'll have an **underflow**, and the number gets rounded to the **nearest**, which is 0.

Example 1.8.5

Round 1.100101_2 ($1,578125_{10}$) to **3 fraction bits**

Method 1) 1.100

Method 2) 1.101

Method 3) 1.100

Method 4) 1.101 (because $1,578125$ it's closer to $1,625$ than $1,5$ is)

1.8.3 Addition with floating point numbers

Let's take two numbers such as

$$2^{(E_1-127)} \times (-1)^{S_1} \times 1.F_1 + 2^{(E_2-127)} \times (-1)^{S_2} \times 1.F_2$$

Let's assume that both the numbers are positive

$$2^{(E_1-127)} \times 1.F_1 + 2^{(E_2-127)} \times 1.F_2$$

- If E_1 is equal to E_2

$$2^{(E_1-127)} \times (1.F_1 + 1.F_2)$$

- If E_1 is not equal to E_2 (for instance if $E_1 = E_2 + c$, with $c \in \mathbb{N}$)

$$2^{(E_1-127)} \times 1.F_1 + 2^{(E_2-127+c)} \times 1.F_2$$

1.8.4 Multiplication with floating point numbers

- 1) First step is to sum the two exponents and subtract the bias;
- 2) Then we multiply the mantissas;
- 3) If needed, we normalise the number;
- 4) We then adjust the sign and at the end we assemble the result.

1.8.5 Subtraction with floating point numbers

There are two ways of doing subtraction with floating point numbers:

- 1) A more standard way, similar to the addition (for unsigned numbers);
- 2) Using the two complement's numbers.

Method 1:

$0 - 0 = 0$	$1 - 1 = 0$	$1 - 0 = 1$	$0 - 1 = 1$ if there is a digit at the right of the number and if it's equal to 1. For the subtraction we borrow the 1 and the starting 0 becomes 10_2 (which is different from 10_{10} , since 10_2 it's 2_{10})
-------------	-------------	-------------	---

Method 2:

For this method the subtraction becomes a sum between two numbers written with the two complement's number notation

1.8.6 Division with floating point numbers

placeholder

1.8.7 Approximation of binary numbers

Say that we have the number 0.11001_2 and we want to approximate it to the first 2 MS Fraction Bits: is it closer to 0.11_2 or 1.00_2 ? There is a method to round it without converting it to base 10:

- if there is a 0 at the most significant fractional bit then we round down;
- if there is a 1:
 - followed by another 1, then we round up;
 - followed by a 0, then we round down;

Following the given example, 0.11001_2 is rounded up to 1.00_2

Chapter 2

Logics

In digital electronics, a **circuit** is a **network** that processes **discrete-valued variables**, and is composed of:

- **Inputs**;
- **Outputs**;
- **Functional specification** (it's the way we express what a circuit does);
- **Timing specifications** (the delay between the inputs changing and the outputs responding).

Circuits are composed of **elements** and **nodes**. An **element** (called with EN) is a circuit with **inputs** (such as A, B, C, \dots), **outputs** (Y and Z), **internals** (connections between the elements in the circuit) and a **specification**. Nodes (called with nN) are **wires** that convey discrete-valued variables and are classified as inputs, outputs or internals.

Name	Equation	Truth Table		
		A	B	Y
AND	$Y = A \times B$	0	0	0
		0	1	0
		1	0	0
		1	1	1
OR	$Y = A + B$	0	0	0
		0	1	1
		1	0	1
		1	1	1
XOR	$Y = A \oplus B$	0	0	0
		0	1	1
		1	0	1
		1	1	0
NAND	$Y = \overline{A \times B}$	0	0	1
		0	1	1
		1	0	1
		1	1	0
NOR	$Y = \overline{A + B}$	0	0	1
		0	1	0
		1	0	0
		1	1	0
XNOR	$Y = \overline{A \oplus B}$	0	0	1
		0	1	0
		1	0	0
		1	1	1

2.1 Logic gates

Logic gates are simple circuits working on binary variables, and they perform logic functions:

- **The AND gate** ($Y = A \times B$): if both the two inputs are equal then it outputs 1, otherwise it outputs 0;
- **The OR gate** ($Y = A + B$): if at least one of the two bits is 1 then it outputs 1, otherwise it outputs 0;
- **The XOR gate** ($Y = A \oplus B$): the X stands for “exclusive”; if only one of the two bits is 1 then it outputs 1, otherwise it outputs 0;
- **The NAND gate** ($Y = \overline{AB}$): it's an AND gate but inverted; it outputs 0 whenever both the inputs are 1;

- **The NOR gate** ($Y = \overline{A + B}$): it's an OR gate that outputs 1 if all the inputs are 0;
- **The XNOR gate** ($Y = \overline{A \oplus B}$): it's an exclusive and negative OR gate, that outputs 1 only if both the inputs are either 0 or 1;

Name	Equation	Truth Table	
		A	Y
NOT	$Y = \overline{A}$	0	1
		1	0
BUF	$Y = A$	0	0
		1	1

There are also other 2 types of logic gates, the **BUF** and the **NOT** gates:

- **The NOT gate** ($Y = \overline{A}$): the $\overline{}$ on the top of a letter means the opposite (so for example \overline{A} is equal to the opposite of A); it inverts the input ($0 \Rightarrow 1$ and $1 \Rightarrow 0$);
- **The BUF gate** ($Y = A$): logically it has no importance (it's equal to a wire), but physically it's used to amplify the signal;

2.2 Combinational and Sequential logic

Circuits are classified in 2 groups:

1) Combinational circuits:

- Their outputs are determined only by the current values of the inputs;
- The previous property makes them "memory-less";
- They are noted with the following character: \mathbb{C} ;

2) Sequential circuits:

- Their outputs are determined not only from their current inputs, but from previous outputs too;
- Because of the previous property we say that they have a memory.

Regarding **combinational logic** there are some rules that allow to transform a smaller circuit into multiple bigger circuits. Those rules are **sufficient** but **not necessary**; we, however, prefer to stick to it. The rules are the following:

- Every element is combinational;
- Every node is either an input or connects to exactly one output of an element;
- The circuit contains no cyclic paths.

2.3 Boolean algebra

Boolean equations are functional specifications that can be expressed via **truth tables** and vice versa (since they have only **2 values**, because 0 equals to False and 1 equals to True). The Boolean equations are a branch of **Boolean algebra**.

There are some **common definitions** for the **Boolean equations**, or in general for Boolean algebra:

- **OR**: is known as **Sum** ($Y = A + B \Rightarrow A \text{ OR } B$);
- **AND**: is known as **Product** ($Y = AB \Rightarrow A \text{ AND } B$);
- **Complement**: is the inverse of a variable ($\bar{A} \Rightarrow \text{NOT } A \Rightarrow \text{Complement of } A$);
- **Literal**: is the variable or its complement (A, \bar{A}, \dots);
- **Implicant**: is the product of one or more literals ($AB\bar{C}D \Rightarrow A \text{ AND } B \text{ AND } \bar{C} \text{ AND } D$);
- **Minterm**: it's a product that includes all input variables ($AB\bar{C}D$);
- **Maxterm**: it's a sum that includes all input variables ($A + \bar{B} + C + \bar{D}$).

In Boolean algebra the precedence for the operations is the following:

- 1) NOT ;
- 2) AND ;
- 3) OR .

2.3.1 Canonical forms (SOP and POS)

Any truth table can be expressed via 2 canonical forms: **Sum Of Products** (SOP) and **Product Of Sums** (POS).

Sum Of Products (SOP):

- 1) Each row has a **minterm** (a product of literals);
- 2) Each **minterm** is True for that and only that row;
- 3) The minterms get named from m_0 to m_n . We'll have a table with p inputs $\Rightarrow n = 2^p$;
- 4) All equations can be written in SOP form;
- 5) We can then form a function by ORing the minterms where the output is 1; thus, a **sum** (OR) of **products** (AND)

A	B	Y	Minterm	Minterm name
0	0	0	$\overline{A}\overline{B}$	m_0
0	1	1	$\overline{A}B$	m_1
1	0	0	$A\overline{B}$	m_2
1	1	1	AB	m_3

$$Y = F(A,B) = \overline{A}B + AB = \sum(1,3)$$

Product Of Sums (POS):

- 1) Each row has a **maxterm** (a sum of literals);
- 2) Each **maxterm** is False for that and only that row;
- 3) The maxterms get named from M_0 to M_n . We'll have a table with p inputs $\Rightarrow n = 2^p$;
- 4) All equations can be written in POS form;
- 5) The function is given by ANDing the maxterms where the output is 0; it's a product (AND) of sums (OR)

A	B	Y	Maxterm	Maxterm name
0	0	0	AB	M_0
0	1	1	$A\overline{B}$	M_1
1	0	0	$\overline{A}B$	M_2
1	1	1	$\overline{A}\overline{B}$	M_3

$$Y = F(A,B) = \overline{A}B + AB = \prod(0,2)$$

2.3.2 Axioms and theorems

SOP and POS can also sometimes lead to larger sets of logic gates, and **the more** are **the expression terms** for the gates **the more expensive and power consuming** will be the circuit. There are though some **axioms** and **theorems** that **simplify the Boolean equations**. This leads to Boolean algebra: it's like regular algebra, but it's simpler, since the variables have only 2 possible values (1 and 0). There are also the duals of each axiom and theorem; in order to do the duals:

- Each 0 must be converted into 1 and vice versa;
- Each + must be converted into \times and vice versa.

All the duals are here listed together with the original axiom / theorem. The **most known** Boolean **axioms** are:

Number	Axiom	Dual	Name
A1	$B = 0 \text{ if } B \neq 1$	$B = 1 \text{ if } B \neq 0$	Binary Field
A2	$\overline{0} = 1$	$\overline{1} = 0$	NOT
A3	$0 \times 0 = 0$	$1 + 1 = 1$	AND / OR
A4	$1 \times 1 = 1$	$0 + 0 = 0$	AND / OR
A5	$0 \times 1 = 1 \times 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND / OR

Here instead we list some **Boolean theorems**, both simple theorems (from T1 to T5) and theorems of several variables (from T6 to T10):

Number	Theorem	Dual	Name
T1	$B \times 1 = B$	$B + 0 = B$	Identity
T2	$B \times 0 = 0$	$B + 1 = 1$	Null Element
T3	$B \times B = B$	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$		Involution
T5	$B \times \overline{B} = 0$	$B + \overline{B} = 1$	Complements
T6	$B \times C = C \times B$	$B + C = C + B$	Commutativity
T7	$B \times C \times D = B \times (C \times D)$	$B + C + D = B + (C + D)$	Associativity
T8	$B \times (C + D) = B \times C + B \times D$	$B + (C \times D) = (B + C) \times (B + D)$	Distributivity
T9	$B \times (B + C) = B$	$B + (B \times C) = B$	Covering
T10	$B \times C + B \times \overline{C} = B$	$(B + C) \times (B + \overline{C}) = B$	Combining

2.3.3 Proof of the theorems

There are two methods for proving the theorems:

- Proof by **perfect induction**: also called “*proof by exhaustion*”, you check every possible input value, drawing a truth table for each equation; if the two truth tables are equal then the equations are equal;
- Proof by using other theorems and axioms;

For instance, let us consider the following example:

Example 2.3.1

With $B = B \times (B + C)$

Truth table

B	C	B + C	B × (B + C)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Theorem

Number	Theorem	Name
T9	$B \times (B + C) = B$	Covering

We see that the B column is actually equal to the $B \times (B + C)$ one, thus we can safely assume that the statement holds. Moreover, the statement is equal to T9. Therefore, via proving that the statement is true we also proved the validity of the theorem T9, making it True

2.3.4 Simplifying an equation

Axioms and theorems are used for simplifying other equations (since the B and the C used in the theorems before can be other equations in turn). The goal is to reduce an equation to the fewest number of implicants, where each implicant has the fewest literals. Sometimes is also possible to expand the equations, always using the theorems or the

axioms; this can be useful when we are in need of a certain term that would eventually simplify the equation

Example 2.3.2

Let the following Boolean equation:

$$Y = A \times (AB + ABC)$$

$$Y = A \times (AB \times (C + 1))$$

$$Y = A \times AB$$

$$Y = AAB$$

$$Y = AB$$

2.3.5 Theorem 11 - Consensus

Another important theorem is the theorem of consensus, also known as T11:

Number	Theorem	Dual	Name
T11	$(B \times C) + (\bar{B} \times D) + (C \times D) = (B \times C) + (\bar{B} \times D)$	$(B + C) \times (\bar{B} + D) \times (C + D) = (B + C) \times (\bar{B} + D)$	Consensus

We can prove the theorem by using other axioms and theorems:

$$\begin{aligned}
 & (B \times C) + (\bar{B} \times D) + (C \times D) = \\
 & = BC \times (B + \bar{B}) + \bar{B}D \times (B + \bar{B}) + CD \times (B + \bar{B}) = \\
 & = BBC + B\bar{B}C + B\bar{B}D + \bar{B}BD + BCD + \bar{B}CD = \\
 & = BC + \bar{B}D + BCD + \bar{B}CD = \\
 & = BC \times (1 + D) + \bar{B}D \times (1 + C) = \\
 & = BC + \bar{B}D
 \end{aligned}$$

2.3.6 Converting from POS to SOP

We say that an equation is written in SOP form when all products are made of only literals. For instance, $Y = ABC + DE$ is an equation in SOP form, but $Y = A\bar{B}(C + DE)$ is not. When converting from SOP to POS, we should apply T8' (which is $W + XZ = (W + X) \times (W + Z)$) as soon as possible. When a SOP form equation contains all the present literals in the equation, then we say that such equation is in **canonical SOP form** (for instance, $Y = ABC + ABC$ is in canonical POS form, while $Y = AB + ABC$ is not).

Example 2.3.3

Let the following Boolean equation:

$$Y = (A + C + D + E) \times (A + B)$$

$$A + ((C + D + E) \times B) =$$

$$\begin{aligned}
 &= A + (BC + BD + BE) = \\
 &= A + BC + BD + BE
 \end{aligned}$$

2.3.7 Converting from SOP to POS

Similarly to the SOP form equations, we say that an equation is in POS form when all sums contain only literals. For example, we say that $Y = (A + B) \times (C + \overline{D}) + E$ is an equation in POS form, while $Y = (AB + C) \times (D + E)$ is not in POS form. There exists also the **canonical POS form**, which conceptually is identical to the canonical SOP form (so for example $Y = (A + B + C) \times (A + B + \overline{C})$ is an equation in canonical POS form). For converting an equation from SOP to POS form, it's important to apply T8' as well as soon as possible.

Example 2.3.4

Let the following Boolean equation:

$$\begin{aligned}
 Y &= A + \overline{B}CDE \\
 &= (A + \overline{B}C) \times ADE = \\
 &= (A + \overline{B}) \times (A + C) \times (A + D) \times (A + E)
 \end{aligned}$$

Example 2.3.5

Let the following Boolean equation:

$$\begin{aligned}
 Y &= AB + \overline{C}DE + F \\
 &= (AB + \overline{C}) \times (AB + DE) + F = \\
 &= (A + \overline{C}) \times (B + \overline{C}) \times (AB + D) \times (AB + E) + F = \\
 &= (A + \overline{C}) \times (B + \overline{C}) \times (A + D) \times (B + D) \times (A + E) \times (B + E) + F = \\
 &= (A + \overline{C} + F) \times (B + \overline{C} + F) \times (A + D + F) \times (B + D + F) \times (A + E + F) \times (B + E + F)
 \end{aligned}$$

2.3.8 Theorem 12 - DeMorgan's Theorem

One of the last theorems is the **DeMorgan's theorem**. This theorem is particularly helpful when dealing with the bubbles inside the circuits.

Number	Theorem	Dual	Name
T12	$\overline{B_0 \times B_1 \times B_2 \times \dots} = \overline{B_0} + \overline{B_1} + \overline{B_2} + \dots$	$\overline{B_0 + B_1 + B_2 + \dots} = \overline{B_0} \times \overline{B_1} \times \overline{B_2} \times \dots$	DeMorgan

It's important to mention that $\overline{A \times B} \neq \overline{A} \times \overline{B}$. Following the **DeMorgan's theorem** we can extrapolate that a NAND gate is equal to an OR gate with the inputs' complements, while a NOR gate is equal to an AND gate with the inputs' complements.

Example 2.3.6

Let the following Boolean equation:

$$\begin{aligned} Y &= \overline{A + \overline{BD} \times \overline{C}} \\ &= \overline{(A + \overline{BD}) + \overline{C}} = \\ &= (\overline{A} \times \overline{\overline{BD}}) + C = \\ &= \overline{A}BD + C \end{aligned}$$

2.4 Bubble pushing

We can push the “**bubbles**” (circuit that inverts the input/output) both backwards and forwards:

- **Backwards:** the bubble is transferred **from** the **output** to the **inputs**, changing the gate from a NAND to an OR;
- **Forwards:** the bubbles are transferred **from** the **inputs** to the **output**, changing the gate from an AND to a NOR.

Bubble pushing works following certain rules:

- They begin at the output, then they work to move to the inputs;
- We have to push bubbles on final output back;
- We then have to draw gates in a form such that bubbles get cancelled.

2.5 Completeness

Theoretically (and even practically), any circuit could be done with only NAND and/or NOR gates and it would be recognised universally, since that two gates are called **functionally complete gates**. The **CMOS** fabrication process (**C**omplementary **M**etal-**O**xide **S**emiconductor) prefers the use of NAND and NOR gates.

2.5.1 From SOP/POS forms to NAND-NAND or NOR-NOR

As we previously said, NAND and NOR gates are considered functionally complete gates, meaning that we can express any circuit using just NANDs or NORs. But how can an expression be transformed into an equation of only NANDs or NORs?

If we want to transform an equation into a NAND-only equation we have first to bring the equation to SOP form, then double negate it. If we use the DeMorgan's theorem once, we'll obtain a negated multiplication of all the terms negated. The NAND-NAND form is a NAND gate where all its inputs are other NAND gates.

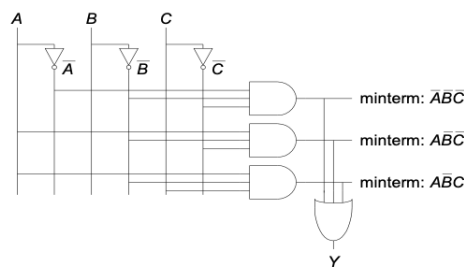
$$A\overline{B}\overline{C} + A\overline{B}D + ABCD = \overline{\overline{A\overline{B}\overline{C}} + \overline{A\overline{B}D} + \overline{ABCD}}$$

Similarly, we can deduce a way to turn any equation into a NOR-NOR: given an equation, turn it into POS form, then double negate it and solve with the DeMorgan's theorem only one negation, until you obtain the form of a NOR of multiple NORs

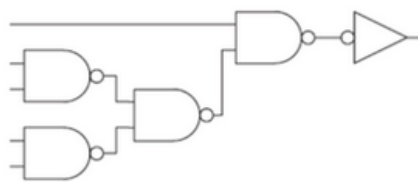
$$\overline{A + (B + \overline{C})} = \overline{A + (\overline{B}C)} = (\overline{A + \overline{B}}) \times (\overline{A + C}) = \overline{(\overline{A + \overline{B}}) \times (\overline{A + C})} = \overline{(\overline{A + \overline{B}}) + (\overline{A + C})}$$

2.6 From Logic to Gates

When we talk about logic, we mainly refer to two types of logic: **multilevel logic** and **two-level logic**. Those are simply two different ways to build up a **schematic**.



A two-level logic is a logic made of ANDs followed by ORs. Two-level logic is always in a SOP form, so we first have a level of AND gates, which is then connected to a level of OR gates. This is also known as a **Programmable Logic Array (PLA)**. A PLA can be transformed to a NAND-NAND network. It's important to mark that NAND or NOT ports are not associative ($\overline{\overline{ABC}} \neq \overline{ABC}$).

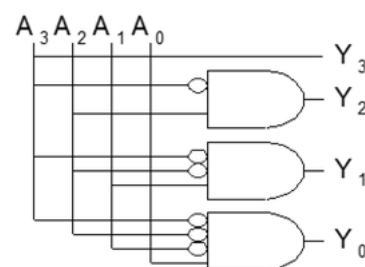


We also have **multi-output logic**. An example of multiple-output logic is the **priority circuit**, where the outputs asserted corresponds to the most significant True input. The equation of a priority circuit is the following:

$$(Y_i = 1) \longleftrightarrow (X_j = 0; \forall j > i)$$

Figure 2.1: On top, an example of **two-level logic**. Below, an example of any **multilevel logic**

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0



2.6.1 Don't Cares

Let us examine the priority circuit: it outputs 1 for the most significant True input. As we can see, where A_3 is 1, it doesn't matter what A_2 , A_1 or A_0 are. When we have these situations, we say that we **don't care** about the input. Don't cares are marked as an X in truth tables, and can be interpreted both as 0 and as 1. For instance, the table of the priority circuit becomes the following:

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

It's important to note that Don't cares can also appear in the output, generically this happens either when we don't really care about the value of the output or when a combination of inputs is actually never possible.

2.7 Circuit schematics rules

There are 4 important rules to follow when drawing a schematic:

- The inputs go to the left;
- The outputs go on the right;
- The gates flow from left to right;
- It's better to use straight wires.

Wires can connect in 2 specific ways:

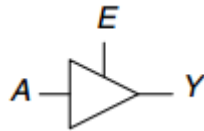
As a T (Connected)	With a dot (Connected)	Without dots (Not connected)

2.8 X and Z

0 and 1 are discrete values, but in reality, they have continuous physical quantities. A binary variable A is always represented with a specific voltage on the wire. For example, 0 volts means 0, while (for example) 1,2V means 1.

- **Contention: X** \Rightarrow when two gates with contrasting outputs drive the signal to the same Y node, an error happens, since the value that comes from the union of a 1 (HIGH) and a 0 (LOW) is unknown (also said **illegal**). This error, or situation, is called **contention**, and is an error that should be **avoided** when possible. The **actual voltage** of an unknown value is **between 0 and V_{DD}** . Physically, contention draws into the circuit a flow of power, causing the circuit to heat and possibly damage. On a more practical point of view, the X value is also called "Don't Care", since it's a value that can be both 1 and 0. This only happens on a truth table though, since in a circuit it has an unknown value;
- **Floating: Z** \Rightarrow when the circuit doesn't drive a node to either 0 or 1, but in something in between the two values, then we say that floating occurred. It's not a problem per se, as long as other elements from the circuit are able to change the Z into either a 0 or a 1. How can we reproduce a floating? A particular circuit, the **tristate buffer**, has 3 possible outputs: HIGH, LOW and floating. Such circuit has an input A , an output Y and an enable signal E . When enable is True, then the

tristate buffer works as a simple buffer; when enable is False though, the tristate buffer outputs (among the possible outputs) a floating. Tristate buffers are used on busses that connect multiple chips and components in a computer that need to connect to the memory. This happens because the requests to the memory can't run all simultaneously, otherwise the data would get compromised.



E	\bar{E}	A	Y_0 (with E)	Y_1 (with \bar{E})
0	0	0	Z	0
0	0	1	Z	1
1	1	0	0	Z
1	1	1	1	Z

2.9 Karnaugh Maps (K-Maps)

It's a way to simplify circuits graphically: this method aims to take the least Boolean equations to draw a circuit, since terms can be combined in order to minimize the equations. For example, with K-Maps we can get the minimized version of an equation like $PA + P\bar{A}$.

In order to use the K-Maps we need a truth table, such that after we can place the combinations of variables on the top of both rows and columns (from column to column / from row to row: the number can only **graphically** change of 1 bit per column/row, so for example $000 \Rightarrow 001 \Rightarrow 011 \Rightarrow 010$ is correct, but $000 \Rightarrow 001 \Rightarrow 010 \Rightarrow 011$ is wrong because from 001 to 010 2 bits have changed. This order of disposing is called **Gray Code**) Each square of the table corresponds to a minterm. For instance, let us consider this example:

$$Y = \bar{A}\bar{B}$$

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

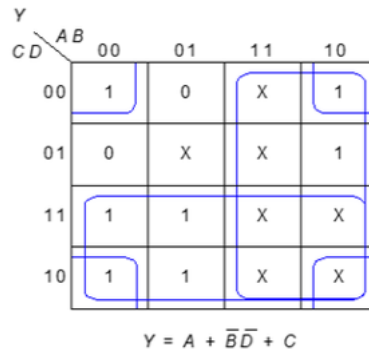
$\bar{A}\bar{B}$	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	$A\bar{B}C$	ABC

$\bar{A}\bar{B}$	00	01	11	10
0	1	0	0	0
1	1	0	0	0

When we draw down the K-map, we proceed to circle all the adjacent ones (you can also go from one side to the other) in groups of n elements, where n is a power of 2. The final formula is written as a SOP of all the groups. When taking a group (or **minterm**), such minterm is made of the variables that are equal for each element of the minterm.

When don't cares are present, they can be either taken or not. It's a good practice to take as many of them as possible, if they do help in forming bigger minterms. For instance, let us consider this example:

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



We can also use K-Maps to create a minimal POS form: we just have to collect the 0 instead of the 1 and remember that each cell represents a **maxterm**, while each group represents a sum. We can include don't cares (if needed) and we consider them as 0.

2.10 Quine-McCluskey algorithm

The **Quine-McCluskey** algorithm (shorted to QMC), also called “*method of prime implicants*”, is an algorithm that shrinks the K-Maps into a one-dimension map. Is functionally the same as doing a K-Map, although it's more efficient because of the form of the tabular.

2.11 Multilevel Combinational Logic

We said earlier that logic in SOP/POS form is called **two-level logic**, although that's not the only form of logic. Often, designers make circuits in more than two levels. The trade-off is that the bigger the circuit, the more power it needs to be powered on and to use it. On the other side, less hardware may be used to do the same circuit on multiple levels. Let's consider for example a three-input XOR gate (image at the right); now, using the previous studied techniques, we can try to answer to the following questions:

- 1) *How many AND gates do you need to build an 8-inputs XOR, using an AND / OR 2-levels network?*

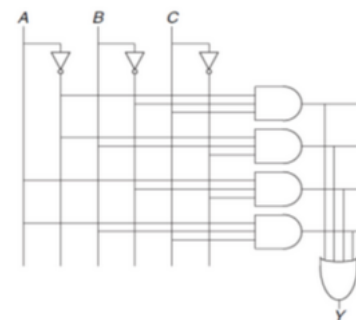
You'd need 128 AND gates, because there are 128 1 in the output column, corresponding to 128 minterms;

- 2) *How many inputs does the AND gate have?*

It has 88 inputs, one for each input of the XOR;

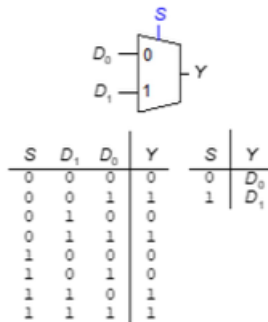
- 3) *How many inputs does the OR gate have?*

You would need 128, one for each AND .



2.12 Combinational building blocks (MUX, DEMUX, Decoders)

When building computer logic, we can use normal combinational logic to build more complex blocks. A couple examples of complex logic blocks are **Multiplexers**, **Demultiplexers** and **Decoders**.



A **Multiplexer** (MUX) selects between one of N inputs to connect to an output. The output is decided thanks to a special bit, the **select input**, also known as the **control signal**.

The equation of a MUX like the one in the picture would be:

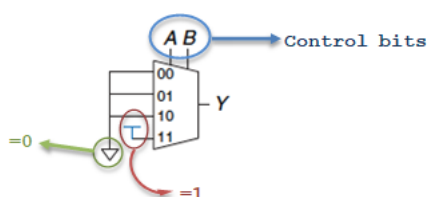
$$Y = \bar{S}D_0 + SD_1$$

Where S is the control signal and D_n are the inputs

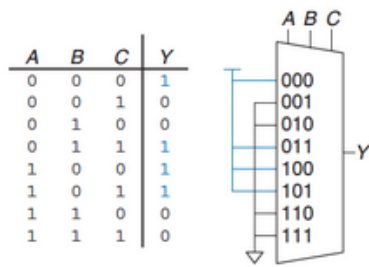
Multiplexers are implemented in 2 ways:

Logic gates (with SOP)	Tristates buffers
<p>Y = $D_0\bar{S} + D_1S$</p>	<p>For an N-input MUX, we use N tristates. We turn on exactly one to select the appropriate input</p>

We can have also for example 4-input multiplexers, but it's important to remark that they will need more than one control bit. The number of control bits must be equal to $\log_2(N)$, where N is the number of inputs.



A MUX can be also shown as in this figure on the left (it serves as an AND gate).



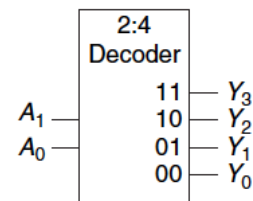
Multiplexers with 4 and more inputs can even be reduced, depending on the function that we want them to have. They can also replace some logic gates and serve as lookup tables (you can have as control bits the inputs, while the multiplexer's inputs would be:

- 1 where our function would output 1;
- 0 where it would output 0.

Decoders are a particular logic that have N inputs (for example A and B) and $2N$ outputs (Y_0, Y_1, Y_2, Y_3). Based

on the combination of both A and B the output may vary.

Decoders' output is **one-hot**, which means that **only one bit at a time** is HIGH (1), while all the others are LOW (0). The opposite of a **Decoder** is the **Encoder**: this circuit takes N inputs and returns $2N$ outputs. Contrary to the Decoders, the Encoders' **output** is **not** a **one-hot** output, but **the input** is.



2.13 Timing of combinational circuits

One of the most difficult challenges while designing a circuit is to make it run fast.

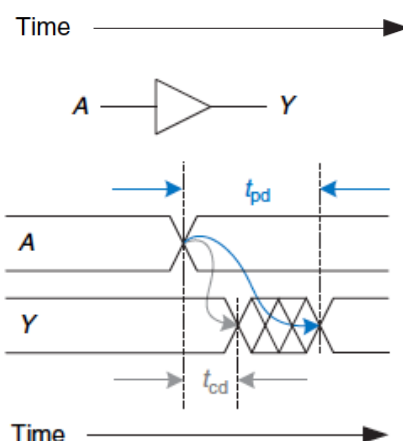
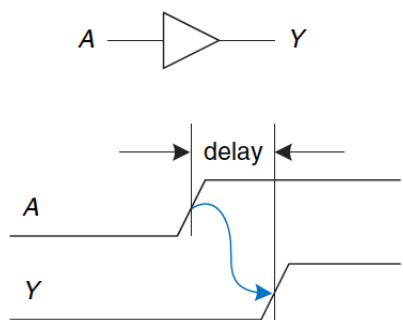


Figure 2.2: Above, a timing diagram of a BUF gate. Below, the same diagram is shown with the variation of the outputs, as well with t_{pd} and t_{cd}

Each circuit has a diagram called **timing diagram**, where the **transient response** of a circuit is portrayed. Each transition from LOW (0) to HIGH (1) is called **rising edge**, while from HIGH to LOW is called **falling edge**. In these examples, blue arrows are used to indicate from what's caused a rising / falling edge.

Delays though can happen in a circuit (mainly because of a circuit's resistances and capacitors, but also because of the limitation of the speed of light). We distinguish 2 types of delays:

- **Propagation delay (t_{pd}):** it's the time that passes from when an input changes until the output(s) reaches its final state;
- **Contamination delay (t_{cd}):** it's the minimum time that passes from when an input changes to the first moment that an output starts to change.

The propagation delay and the contamination delay can be different because of various reasons such as:

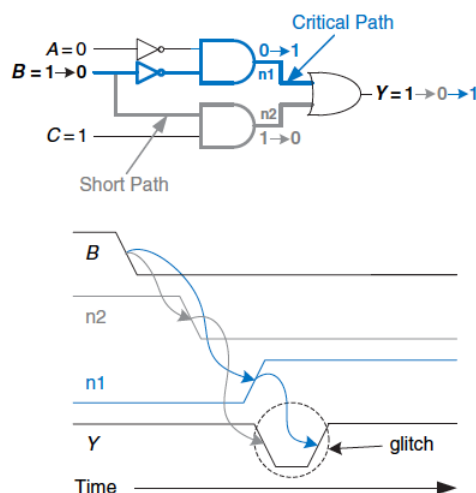
- Different rising and falling delays;
- Abundance of different inputs and outputs, which may require more time to be processed;
- Circuits slow down when the temperature becomes high and speed up when the temperature is cold.

Normally all the delays of all the gates are listed by the producers. Another reason that can significantly affect the propagation and contamination delays is the **path** that a signal has to traverse. We call **critical path** the longest and, by consequence, this is **slowest path** in the circuit. That said, we can then come up with the definition of **Propagation Delay** and **Contamination Delay** of a circuit (differently from the ones explained above, these ones can help in finding a formula to get the actual timing of a circuit):

- the **total propagation delay** of a combinational circuit is the **sum** of the **propagation delays** through each element on the critical path;
- the **total contamination delay** is the **sum** of the **contamination delays** through each element on the short path.

2.14 Glitches

When designing a circuit, it may occur that while transitioning one input (so for example with the input $B = 1 \Rightarrow B = 0$) there may be multiple occurrences of a single output. This is called a **glitch** or **hazard**, and while it usually doesn't cause any problem, it's a good thing to be aware of them.



In this example, the input Y changes briefly to 0 before becoming again 1, and that happens because B has to pass both through the **Critical Path** and the **Short Path**, causing the circuit to return an incorrect output for a short time. In order to **avoid glitches** we can **add more gates**, in order to **add some delay** and solve the issue.

Chapter 3

Sequential circuits

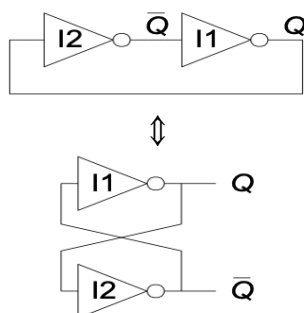
We saw with combinational logic that a circuit provides certain inputs and the outputs depend only on the given inputs. But what if a circuit could remind a previous output and use it as an input? In that case, we would be talking about **sequential circuits**.

Sequential circuits are different from combinational circuits because they have a “*memory*”, and their **outputs** are used as **inputs** to **influence** the **newer outputs**. Sequential logic may **remember** explicitly a **previous output** or may remember **some information** of it: these information are called **states of the system**.

But how does a circuit remember something? Via some **data memory banks** (on **short term**) that can store some bits of the states: those memory banks are either **latches** and/or **flip-flops**, and they are both called **bi-stable elements**, because of their double stable state.

3.1 Latches and Flip-Flops

As we mentioned previously, latches and flip-flops are bi-stable elements, because they have **two states**, which are both **stable**. Those states will be called Q and \bar{Q} .



For example, a simple bi-stable circuit might be a circuit with two NOT gates one connected to each other, where there are no inputs but there are two outputs. The gates are connected cyclically.

Let us consider the case where $Q = 0$: if Q is 0 then \bar{Q} is 1, and the system will be consistent. If instead $Q = 1$, then $\bar{Q} = 0$, and the system will be consistent again. There might be a third invisible state, where both Q and \bar{Q} are neither 1 nor 0, and this state is called **meta-stable state**.

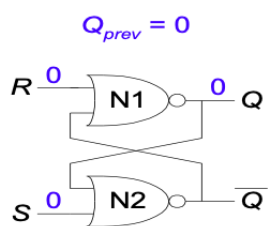
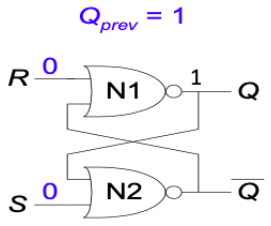
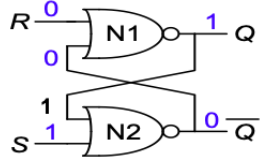
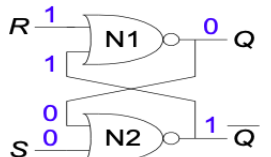
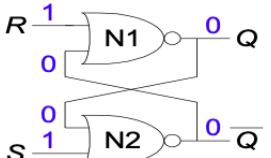
This way we can store 1 bit of state in the variable Q , even though there is no way to control it, since there are no outputs. However, following this simple logic, we can assume that, with N stable states, we can store $\log_2(N)$ bits of information. It's important to mention that, in this previous circuit, when power will be firstly applied to it, we won't know with precision the beginning state, so it may change between each time that it gets powered on.

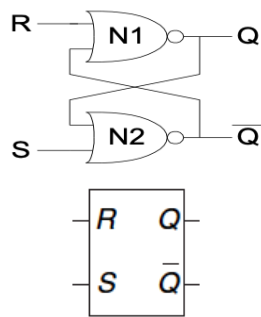
3.1.1 SR Latches

SR Latches are simple bi-stable circuits composed by two NOR gates. In both gates the inputs are:

- an external input (either R or S);
- the output of the previous inputs.

S and R stand for Set and Reset and are used to control the circuit. Let us now analyse the truth table of this circuit:

R	S	Q	\bar{Q}	Details
0	0	Q_{prev}	\bar{Q}_{prev}	<p>N1 has as inputs both R and \bar{Q}, but we can't determine it. The same happens with N2, that has as inputs R and Q. We're stuck in a condition where we are missing a variable. Since Q is either 1 or 0 though, we can study both cases and get a finite result</p> <p>$Q_{prev} = 0$</p>  <p>$Q_{prev} = 1$</p> 
0	1	1	0	<p>N1 has no TRUEs, so it will output a TRUE. N2 instead has 2 TRUEs, so it will output a FALSE. The output is set to 1</p> 
1	0	0	1	<p>N1 has at least one TRUE (from R), thus the output will be FALSE. Since N2 has two FALSEs, then it will produce a TRUE. This way, the output will be reset to 0</p> 
1	1	0	0	<p>If both R and S are 1, then both outputs will be 0. Having $Q = \bar{Q} = 0$ is nonsense though</p> 



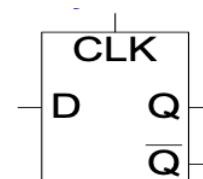
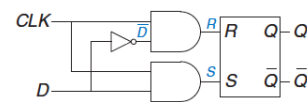
The first case (where $S = R = 0$) gave us a state where the previous output is “remembered”, like if it was saved in memory. This is the leading logic behind the bi-state logic. We have although a problem: the state where $R = S = 1$. That state is considered **invalid** and must be **avoided**. In order to avoid such problem, there is another type of latch that can help, and such latch is called **D Latch**.

3.1.2 D Latches

The strangeness of the SR Latch is that not only problems arise when both S and R are simultaneously 1, but that the output depends on what the input is and when the input changes. The D Latch avoids these problems, by using 2 inputs, CLK and D:

- **CLK** (or clock): determines when the input should change;
- **D**: is the **input**.

The D Latch is, in other words, an SR Latch with different inputs, but the principle behind it remains the same. R gets extended, in order to become an AND gate between CLK and \overline{D} , while S becomes an AND gate between D and CLK. When CLK is 1 the D Latch is said to be **transparent**, while when CLK is 0 the D Latch is called **opaque**.



What does this change mean? First, the data changes only when CLK is 1 (so when CLK is 1 and D is 1, S is 1 and R is 0, and when D is 0 then S is 0 and R is 1), but second, and more importantly, it's **impossible** that R and S are both simultaneously 1.

Although the D Latch fixes the main problem of the SR Latch, there is a slight problem with it: in every moment that the CLK signal is 1 the data can change, so it can change multiple times during the time window given by $CLK = 1$. We'll see that it may be a problem and cause some issues, since it's not always wanted as a thing.

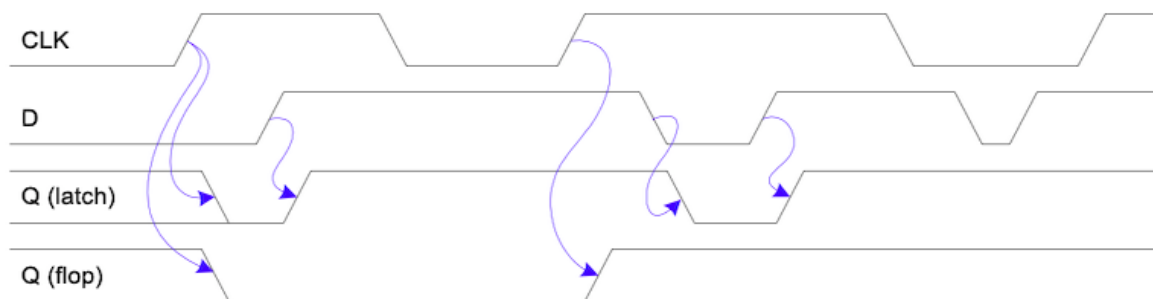
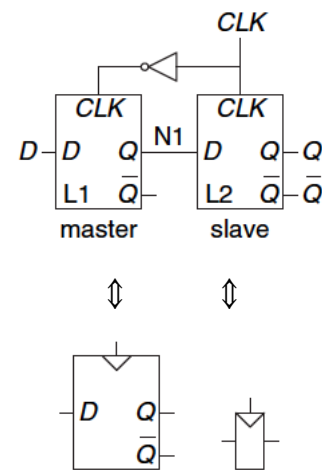
As in the bi-state circuit described in the beginning and the SR Latch, when the circuit gets powered on, we don't know the state of Q. This is why, in a **waveform**, it's **signed as X** (so as a Don't Care).

3.1.3 D Flip-Flop

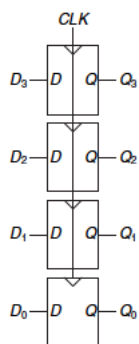
The D flip-flop solves the problem of the D Latch, where the output may change for an indefinite number of times while CLK is 1. When CLK is 1, D passes through Q, and when CLK is 0 the circuit holds the previous value of Q. Plus, the output changes only when CLK is 1; this is the reason why the D flip-flop is called **edge-triggered**.

The D flip-flop works in this way: when CLK is 0, the master D Latch ($L1$) is active, and saves the input D into the output Q ; via the $N1$ wire, the output of the first D Latch goes into the slave D Latch ($L2$). When CLK will become 1 the output Q of $L1$ will go inside the second D Latch ($L2$). We can thus conclude that a D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times.

Here below is shown the waveform of both a D Latch and a D flip-flop:



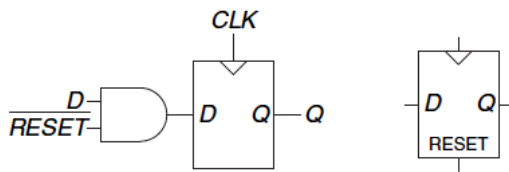
3.1.4 Registers



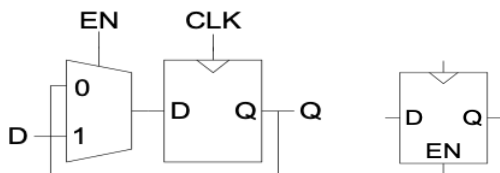
Simply put, registers are a bank of D flip-flops with a shared CLK signal. Registers may save multiple bits (so we may have a N-bits register), and based on the number of bits corresponds the number of necessary D flip-flops.

Registers are important in sequential circuits, since they have the possibility of storing bits for the circuit's states. Nevertheless, their price in the market is particularly high: this is the reason why our computer's memory isn't made with registers but with other techniques that will be explained further in those notes.

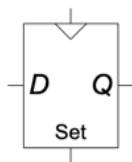
3.1.5 Resettable, Settable and Enabled flip-flops



Resettable flip-flops: they have a second input called RESET. When RESET is 0, the resettable flip-flop behaves like a normal D flip-flop, while when RESET is 1 the flip-flop forces its state to be 0. This kind of flip-flops may be either asynchronous or synchronous: asynchronous means that the flip-flop resets his state whenever RESET is TRUE, while synchronous means that the flip-flop resets itself on the positive edge of the CLK. This type of flip-flop comes in handy when we have to force a given state;



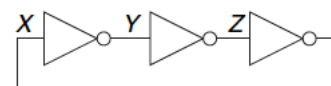
Enabled flip-flops: they have a multiplexer that controls whether the signal has to pass or not. If EN is TRUE, then the flip-flop will store inside the value of D, while if EN is FALSE then the value Q stored inside the flip-flop will be reinserted. An alternative implementation has in place of the CLK signal an AND gate where the two inputs are the EN and CLK signals;



Settable flip-flops: The logic behind it is similar to the one of the Resettable flip-flops: there is another input called SET, and whenever SET is TRUE the flip-flop's value Q will be equal to 1. If SET is FALSE, then the flip-flop behaves like a normal flip-flop.

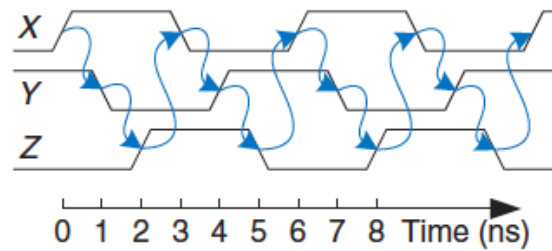
3.2 Synchronous Logic Design

Generally all the non-combinational circuits are sequential circuits, so, in other terms, circuits whose output is not dictated only by the current inputs. There are two examples of sequential circuits though that are particular for the way they work:



- **Astable circuits:** an example would be a circuit with 3 NOT gates one by the other, and in this circuit there would be no inputs since the output of the last NOT gate is the input of the circuit itself (so the output of Z is the input of X). The waveform

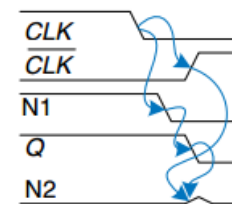
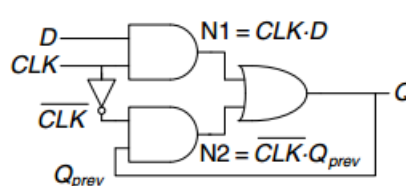
shows the problem with this circuit: suppose that X is 0, then Y would be 1, Z would be 0 and the output of Z (which is X) would become 1, having a situation where the signal changes constantly. This is possible, but the circuit doesn't have a defined state, and therefore is called **astable circuit**. From the waveform we can see that each $6ns$ the situation is back as in the starting moment, it's a periodic circuit. We name this kind of circuits (so the circuits that have zero inputs and 1 output constantly oscillating) **ring oscillators**. The period of the ring oscillator depends from multiple factors, such as manufacturing delays, voltage, temperature, etc etc. . .



- **Race conditions:** suppose that we have a circuit as in the image below: we want to prove if such circuit is an enhanced version of the standard D Latch. Normally, even by looking at the truth table, it would seem like it's actually a better version. But let's examine the following scenario: let D be 1; when CLK becomes 1, then the latch becomes transparent and registers the new bit. When CLK becomes 0 though, there could be a moment where Q becomes \overline{Q} . This could happen if the inversion of the bit takes some more time, compared to the time needed for the new latch to output Q and to throw it back into the circuit. This causes what we call a **race condition**, because certain gates are slower than others, making the circuit output some unwanted values.

$$Q = CLK \times D + \overline{CLK} \times Q_{prev}$$

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



3.3 Finite State Machines (FSM)

As we saw in the previous section, when designing a sequential circuit a lot of things can happen: there can be **race conditions**, some **cyclic paths** can lead to **astable circuits**, and so on... How can we make a proper **sequential logic**?

In order to answer to that question, let us review quickly what a sequential circuit is: it's a circuit whose **outputs influence** the **new outputs**, since they are used as inputs. Can we say that a sequential circuit is just a combinational logic where we save the state of the output and then we return it back? Yes, we can. In fact, most of the sequential circuits are made that way: there is one (or more) combinational part, and after such part the output(s) gets saved and used back.

There are two types of sequential circuits: **synchronous** and **asynchronous**. Synchronous sequential circuits have a **clock input** (a CLK signal), whose **rising edge** indicates the moment when the circuit **transitions** from one state to the other. There are **4 rules** that a sequential circuit has to respect in order to be said *synchronous*:

- Every element of the circuit is **either a combinational logic** or a **register**;
- There is **at least one register**;
- All registers act on the same CLK signal;
- **Every cyclic path** contains **at least one register**.

The most simple type of synchronous sequential circuit is the flip-flop. Other two types of synchronous sequential circuits that occur multiple times in computer science are the **Finite State Machines** (FSMs) and the **pipelines**.

Finite State Machines are basically sequential circuits that have a defined number of states. FSMs have M inputs, N outputs and k bits of state (thus a maximum of 2^k states). On every rising edge of the CLK, the circuit advances to its next state, which is given by the combination of the output of the previous state with the next inputs. Two types of FSMs exist:

- **Moore FSMs**: their outputs depend uniquely from the current state of the machine;
- **Mealy FSMs**: their outputs depend on both the current state and the current inputs.

An example of Moore FSM is a traffic light: You don't need any special input between one state and the other, you only need to know the previous state. A Mealy FSM could be the previous example but with some buttons that let the traffic light jump to another state

3.3.1 FSMs Initialization and Unreachable States

When a Flip-Flop is turned on, its state is indeterminate, we don't know if it's 0 or 1. The initial state is then not defined, since it could be any state. In order to avoid such problems, FSMs always have a reset signal, which forces them into a specific state.

If, for example, our circuit had 3 states encoded as 00, 01 and 10, what about state 11? The state that holds the state 11 is called **unreachable**. Unreachable states cannot be reached with any sequence of inputs. Depending on the initial state of the FSM, the "*unattainability*" changes. This happens if the number of states is not a power of 2. In order to avoid unattainability, FSMs use the same reset signal as mentioned before, which forces the machine in (usually) a 00...0 state (so a default state); from this state the machine can start working.

3.3.2 FSM Encoding

FSM's states can be encoded in 2 ways:

- **Binary encoding:** for example, to represent 4 states we have 00, 01, 10 and 11;
- **One-hot encoding:** following this way, there can be only one state bit per state (we'll have then 0001, 0010, 0100 and 1000). It has pros and cons:
 - **Cons:** you need more state bits, so more flip-flops;
 - **Pros:** sometimes the next state and the output logic can be simpler.

3.3.3 Moore vs Mealy FSMs

In order to explain the pros and cons of both **Moore** and **Mealy** FSMs, let us analyse the following example:

Example 3.3.1

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. On each clock cycle, the snail crawls to the next bit. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.

Solution to Exercise 3.3.1

We encode our outputs in 2 ways:

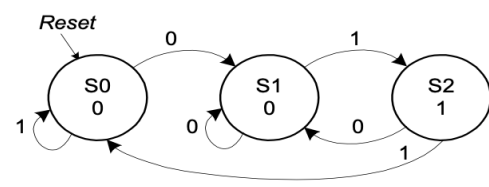
- 0: don't smile;
- 1: smile;

We have then not 4, but 3 states:

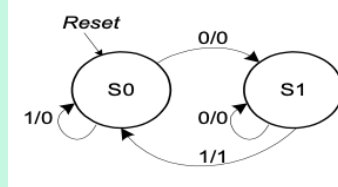
- I saw a 0 (the output will be 0) \Rightarrow **State 1 (S1)**;
- I saw a 1 but the next bit is 1 (the output will be 0) \Rightarrow **State 0 (S0)**;
- I saw a 1 but the next bit is 0 (the output will be 1) \Rightarrow **State 2 (S2)**.

The circuit will have the reset point in S0.

Moore FSM



Mealy FSM



On **Mealy** FSMs, each arc indicates both inputs and outputs, following the scheme input/output. **Moore** FSMs instead have the output written inside the state. How do we draw a **Moore** FSM table?

For the **Moore** FSM

Current State		Inputs	Next State	
S_1	S_0	A	S'_1	S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	0	0

State encoding	
S_0	00
S_1	01
S_2	10

For the **Mealy** FSM

Current State		Inputs	Next State	Output
S_0		A	S'_0	Y
0		0	1	0
0		1	0	0
1		0	1	0
1		1	0	1

State encoding	
S_0	00
S_1	01
S_2	10

For **Moore** both input, output and status change at each clock, but for **Mealy** the output changes as soon as the input changes, even if the clock hasn't ended yet.

Moore FSMs:

- Typically has more states than a Mealy machine;
- It's safer than the Mealy machine, since the output changes only on a clock edge.

Mealy FSMs:

- Typically has fewer states;
- It reacts faster to inputs since it doesn't wait for the clock;
- Its asynchronous outputs could be dangerous and, if the input glitches, then this glitch is propagated to the output.

Registered Mealy FSMs:

- A **registered Mealy** FSM is a machine that has register on the outputs. The temporal behaviour is similar to the one of a Moore machine. This FSM though can avoid glitches.

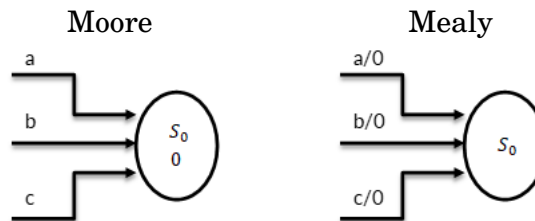
3.3.4 Glitches

For glitches we mean those short periods during the execution of a Mealy FSM where the output, if it had to be for example 1, will briefly turn to 0 and then return 1 (it can occur also with the opposite, so from 0 it turns to 1 and then back to 0). Normally glitches occur when in a FSM some circuits use less time than others to compute the output. They are

common and can occur, unlike bugs they are expected to happen. As in combinational circuits, they usually don't have a big impact, but it's important to be aware of them and plan ahead.

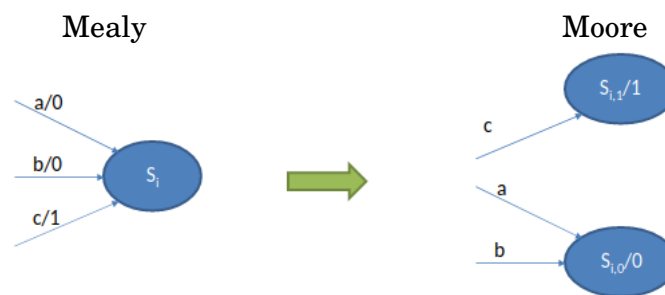
3.3.5 From Moore to Mealy

Each Moore FSM can be transformed into a Mealy FSM, by associating the output from the states to the arcs.



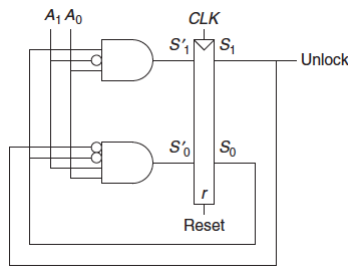
3.3.6 From Mealy to Moore

Each Mealy FSM can become a Moore FSM, just by creating different statuses depending on the outputs. When the output differs, we create a new state with the same specification, but that only differs for the output



3.4 Deriving a FSM from a schematic

- 1) We examine the circuit, stating inputs, outputs, and state bits;
- 2) Write the next state and output equations;
- 3) Create the next state and output tables;
- 4) Reduce the next state table to eliminate unreachable states;
- 5) Assign each valid state bit combination a name;
- 6) Rewrite next state and output tables with state names;
- 7) Draw state transition diagram;
- 8) State in words what the FSM does.



Current State S_1 S_0	Input A_1 A_0	Next State S_1' S_0'
0 0	0 0	0 0
0 0	0 1	0 0
0 0	1 0	0 0
0 0	1 1	0 1
0 1	0 0	0 0
0 1	0 1	1 0
0 1	1 0	0 0
0 1	1 1	0 0
1 0	0 0	0 0
1 0	0 1	0 0
1 0	1 0	0 0
1 0	1 1	0 0
1 1	0 0	0 0
1 1	0 1	1 0
1 1	1 0	0 0
1 1	1 1	0 0

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Let us examine the FSM on the left. First of all, this FSM is a **Moore** FSM, since the output depends only on the clock's output. The first AND gate has as inputs $S_0 \times \overline{A_1} \times A_0$, while the second AND's formula is $\overline{S_1} \times \overline{S_0} \times A_1 \times A_0$.

We then write the table with all the possible cases (the second image on the left). We see that 11 is never reached from the circuit (as next state), so we define it as our unreachable state. We get from the table that we pass to S_1 only in 2 cases.

We can then write a more compact table (the third image on the left) that describes when the output will be reached. We can thus derive (both from the table and the circuit) that we don't care if S_0 is either 0 or 1, we just need that S_1 is 1. For this reason, we name 3 states:

- S_0 : it's the initial state, and it changes to S_1 only in one case: when A is 3;
- S_1 : we reach it only when S_0 is 3 and leads to the output if it's 1.

3.5 Timing

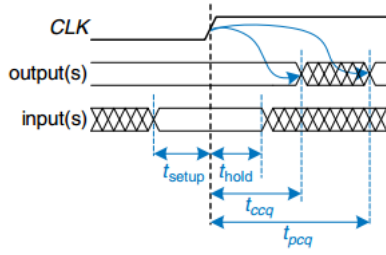
The time between two subsequent rising edges of the CLK is called clock period, and it's written as T_c . We call $f_c = 1/T_c$ the clock **frequency**. Such frequency is measured in Hertz (Hz).

We define 2 constraints for the input:

- Setup time (t_{setup}): it's the time before the clock edge, where the input must be stable. It depends from the maximum delay through the combinational logic from a register R_1 to a register R_2 ;
- Hold time (t_{hold}): it's like t_{setup} , but after the clock edge; in this period, the input must be stable.

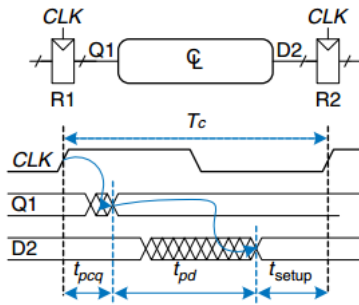
The input D must be stable when sampled (so stored into a register). If that doesn't happen, then **meta-stability** can occur (so a value in between 0 and 1 will be registered into the register).

Smaller \Rightarrow Bigger	1 GHz (1 GHz = 1000 MHz)
	1 MHz (1 MHz = 1000 Hz)
	1 Hz



The sum between t_{setup} and t_{hold} is called **Aperture Time** of the circuit ($t_a = t_{\text{setup}} + t_{\text{hold}}$). There are also constraints for the output:

- Clock Propagation delay (t_{pcq}): it's the time after the clock edge where the output Q is guaranteed to be stable;
- Clock Contamination delay (t_{ccq}): it's the time after the clock edge where it's possible for contaminations to occur.



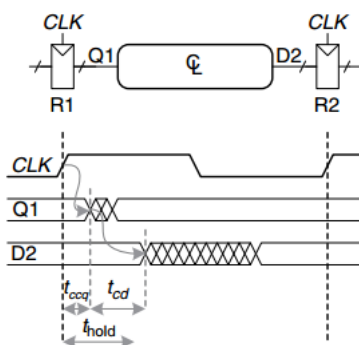
Following the **dynamic discipline** (the branch of rules regarding timing on sequential circuits), synchronous sequential circuit inputs must be stable during both t_{setup} and t_{hold} . The timing constraint for the **setup time** depends on the maximum delay of the combinational part, from when a signal departs from a register R_1 up when it arrives to a register R_2 . It has to be bigger than the sum of t_{pcq} , t_{pd} and t_{setup} :

$$T_c \geq t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup}}$$

Since t_{pcq} and t_{setup} are given by the manufacturer of the circuit, we can rearrange the equation to obtain the propagation delay:

$$t_{\text{pd}} \leq T_c - (t_{\text{pcq}} + t_{\text{setup}})$$

The terms in the brackets are called **sequencing overhead**. It's important that the timing of the circuit respects the propagation delay through the combinational logic, since if t_{pd} was too long, then the output of the combinational logic wouldn't be able to reach R_2 on time, causing a malfunctioning of the circuit. This is why the previous equation, called **setup time constraint**, or **max-delay constraint**, is important.



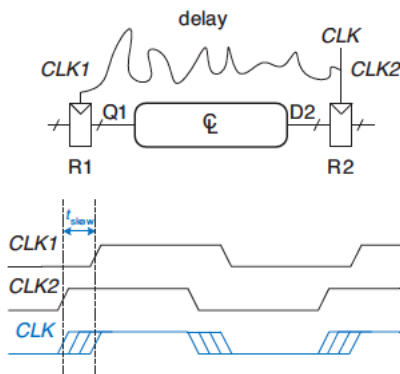
We also need another time constraint, to ensure that the data that come from the register R_1 won't be contaminated before entering the register. We call this constraint the **hold time constraint**. The data (in our case D_2) should be stable because registers, even if they become transparent on the rising clock edge, they need some time to effectively become transparent, and the input of the register must be stable in that period. The hold time has to be before t_{cd} (so before the output of the combinational part changes, which is the input for R_2) plus t_{ccq} . This gives rise to the following formula:

$$t_{\text{hold}} < t_{\text{ccq}} + t_{\text{cd}}$$

As a consequence, we get that

$$t_{\text{cd}} > t_{\text{hold}} - t_{\text{ccq}}$$

3.6 Clock Skew



We assumed previously that the clock reaches all the registers at the same time, but that could not be true every time, since there could be some variations. We call these variations in clock edges **clock skew** (t_{skew}). This is a common problem that must be acknowledged since it's impossible that all wires have the same length. Clock gating moreover delays the clock. How should we deal with that? We must consider the worst-case scenario, and make it standard for all the registers, such that it will work evenly for all the registers.

For example, the circuit in the image above has 2 clocks: CLK_2 is earlier than CLK_1 , so our worst-case scenario is CLK_1 ; we'll then proceed to make CLK_1 our worst case scenario.

3.7 Parallelism

The speed of a system is characterized by both latency and throughput of the information's moving through it. We define as:

- **Token:** a group of inputs that get processed to produce a group of outputs;
- **Latency:** it's the time that a token needs to pass from the start to the end (the lower the better);
- **Throughput:** is the number of tokens produced per unit time (the higher the better).

Throughput can be improved by processing more trays at the same time. We call this process **parallelism**. There are two types of parallelism:

- **Spatial parallelism:**
 - In this type of parallelism, multiple tasks are accomplished by multiple copies of the hardware at the same time.
- **Temporal parallelism:**
 - The tasks are broken into multiple stages (as an assembly line);
 - Each task must cross all the stages, but different stages work at the same time on different tasks;
 - This sort of parallelism is also called **pipelining**.

Temporal and spatial parallelism can be combined to increase the proficiency of the hardware. We call “filling of the pipeline” the first instructions that, in a temporal parallelism, don't affect neither the throughput nor the latency.

Generally, in a sequential circuit we denote the latency with L , while the throughput is denoted with $1/L$.

- With spatial parallelism with N hardware copies, the throughput is N/L .
- With temporal parallelism, things become a bit trickier:
 - Ideally, we would be able to split the tasks in N stages of equal length, so we would have a throughput of N/L ;
 - In reality, that's nearly impossible: the throughput will depend on the longest latency L_1 , and it will be equal to $1/L_1$.

Chapter 4

Digital Building Blocks

We already saw some digital building blocks, such as gates, multiplexers, decoders and registers, but there are much more; some examples are:

- arithmetic circuits;
- counters;
- memory arrays;
- logic arrays.

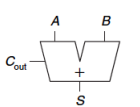
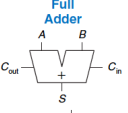
Building blocks are used to build microprocessors, and they have 3 important properties:

- **hierarchy** of simpler components;
- **modularity** because they have well-defined interfaces and functions;
- **regularity** that allows to easily extend their size.

4.1 Arithmetic Combinational circuits

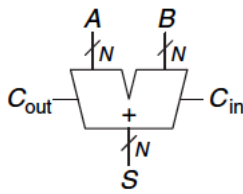
Arithmetic is essential in computers since every function is based on binary arithmetic. Some important building blocks are related to arithmetic's.

4.1.1 1-Bit and N-Bit Adders

1-Bit Adders																																																																		
Half Adder	Full Adder																																																																	
 <table><thead><tr><th>A</th><th>B</th><th>C_{out}</th><th>S</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></tbody></table> <p>$S = A \oplus B$ $C_{out} = AB$</p>	A	B	C _{out}	S	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	0	<p>Full Adder</p>  <table><thead><tr><th>C_{in}</th><th>A</th><th>B</th><th>C_{out}</th><th>S</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></tbody></table> <p>$S = A \oplus B \oplus C_{in}$ $C_{out} = AB + AC_{in} + BC_{in}$</p>	C _{in}	A	B	C _{out}	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
A	B	C _{out}	S																																																															
0	0	0	0																																																															
0	1	0	1																																																															
1	0	0	1																																																															
1	1	1	0																																																															
C _{in}	A	B	C _{out}	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																																																														

Since addition is an important operation for computers, there are building blocks that do exactly that. Although, in order to do additions with N -Bits numbers, we must start with a **1-Bit adder**.

1-Bit adders are divided in **Half Adders** and **Full Adders**. A Half Adder (such as the one on the left image), takes two bits A and B as input and performs the addition between them. The result will be the output S , and in case of a Carry, it will be outputted as C_{out} . The difference between the Half Adder and the Full Adder is that the full adder accepts a Carry as an input (C_{in}).



When multiple adders are used, they usually propagate the Carries from one adder to the other: we call the adders on this chain **Carry Propagate Adders** (CPAs), and we have three ways of doing this “chain” of CPAs:

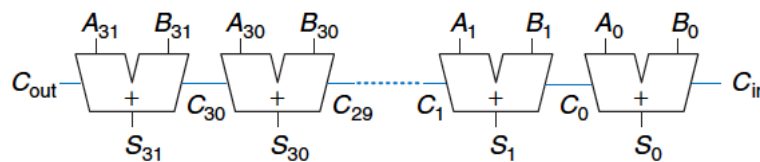
- **Ripple-carry** (slow);
- **Carry-lookahead** (fast);
- **Prefix** (fastest).

Carry-lookahead and **Prefix** CPAs are faster but require more hardware than the **Ripple-Carry**.

A Ripple-Carry works as the following: there is an adder for every bit of the number, and we place a **Full Adder** for each bit. This is extremely slow.

$$t_{\text{ripple}} = N \times t_{FA}$$

where t_{FA} is the delay of the Full Adder and N the number of bits



Ripple-Carry is slow because the carry signal has to propagate through all the adders. **Carry Look-Ahead Adders** (CLAs) represent an improvement over the Ripple-Carry: in order to achieve a higher speed, the adder gets split into multiple blocks (so for instance a 32-bit adder gets split into 8 blocks of 4 adders each), and then the blocks get added via a Ripple-Carry adder. The adders can also be called **columns** when talking about CLAs. Via some circuitry, as soon as a carry out from a block is known, it gets propagated. This way, each block looks ahead for the presence of carries in all the other blocks.

In order to implement the *look-ahead* functionality, we use two signals: **generate** (G) and **propagate** (P):

- the **generate** signal is equal to 1 if the block (or column) n produces a carry out independently from the presence of a carry in;

$$G_n = A_n \times B_n$$

- the **propagate** signal is 1 if the block (or column) n produces a carry out because of a carry in from the previous block (or column) (so $C_n(\text{block}_{(-1)})$)

$$P_n = A_n + B_n$$

We can say that a block n generates a carry out C_o if it either generates a carry (G_n) or if it propagates one ($P_n C_{n-1}$).

$$C_n = G_n + P_n C_{n-1} = (A_n \times B_n) + (A_n + B_n) \times C_{n-1}$$

We define $G_{i:j}$ and $P_{i:j}$ as the generate and propagate signal from column i to column j .

For instance, let us assume that we have a CLA adder, where we have 4 blocks and where the block size is 4 bits: how would the carry go through the CLA? We know that the Carry Out for a block is given by

$$C_{n:0} = G_{n:0} + P_{n:0}C_{n-1}$$

We have thus to inspect both $G_{3:0}$ and $P_{3:0}$, since the carry out is basically the OR between them. First, let's examine the behaviour of $G_{3:0}$ for a single block: the generate signal is either:

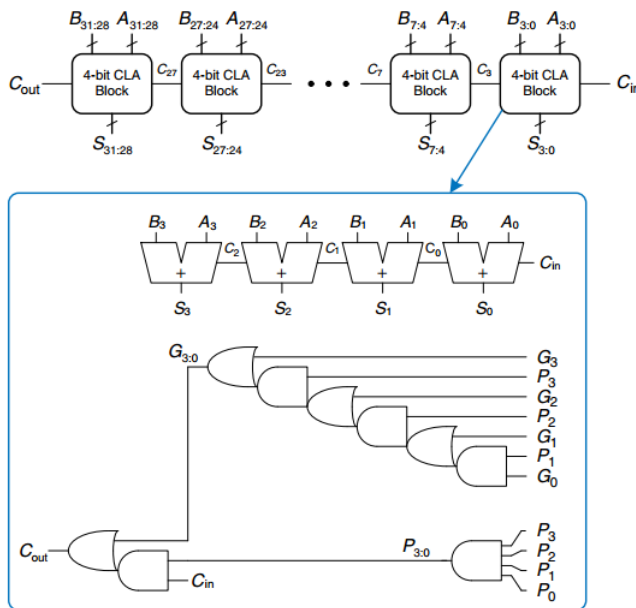
- generated by column 3, or
- generated by column 2 and propagated through column 3, or
- generated by column 1 and propagated through columns 2 and 3, or
- generated by column 0 and propagated through columns 1, 2 and 3.

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0)))$$

The propagate signal instead is 1 if it's propagated by all the columns:

$$P_{3:0} = P_3P_2P_1P_0$$

We can visually represent what we have done so far with the formulas:



We can see how a block is made of both adders and G and P signals. Except for the first block, where the Carry has to be computed, after C_{in} is set, then we can see the value of C_{out} just after the time of both the AND and OR gates (t_{AND_OR}).

A CLA works, more practically, in this way:

- **Step 1:** Compute G_n and P_n for each column of the block (so for each bit), and it takes the delay of a single AND or OR gate (t_{pd});
- **Step 2:** Compute then G and P for all the blocks in the adder, taking t_{pg_block} ;
- **Step 3:** Propagate C_{in} through each propagate/generate logic, meanwhile compute the sum of the numbers $((N/k - 1) \times t_{AND_OR})$;
- **Step 4:** Compute the sum for the most significant block ($k \times t_{FA}$).

In total, a CLA takes the following amount of time to compare a sum of two N -bit wide numbers with blocks of k bits:

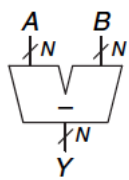
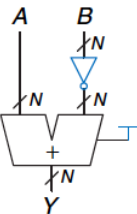
$$t_{\text{CLA}} = t_{pg} + t_{pg_block} + (N/k - 1) \times t_{\text{AND_OR}} + k \times t_{FA}$$

where:

- t_{pg} is the delay to generate all P_n and G_n (a single AND or OR gate);
- t_{pg_block} is the delay to generate all $P_{i:j}$ and $G_{i:j}$;
- $t_{\text{AND_OR}}$ is the delay from C_{in} to C_{out} of the final AND/OR gate. It's repeated for the number of blocks minus the most significant one, because for that block we account $k \times t_{FA}$, since it's larger than $t_{\text{AND_OR}}$;
- t_{FA} is the delay of a full adder multiplied by the number of bits.

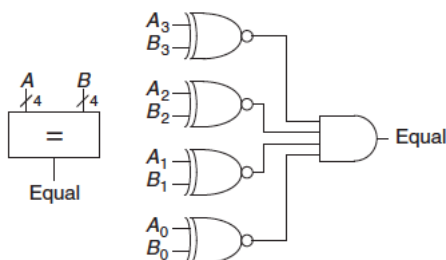
Even if it represents an improvement compared to the ripple-carry adder, the delay still grows linearly with the size of the numbers. There is a third type of adders, which is the **Prefix Adders**. This kind of adders extends the CLA logic of generate and propagate by first computing G and P for a pair of columns, then with blocks of 4, then with blocks of 8, and so on and so forth until it computes the carries for all the columns.

4.1.2 Subtractors and Comparators

Symbol	Implementation
	

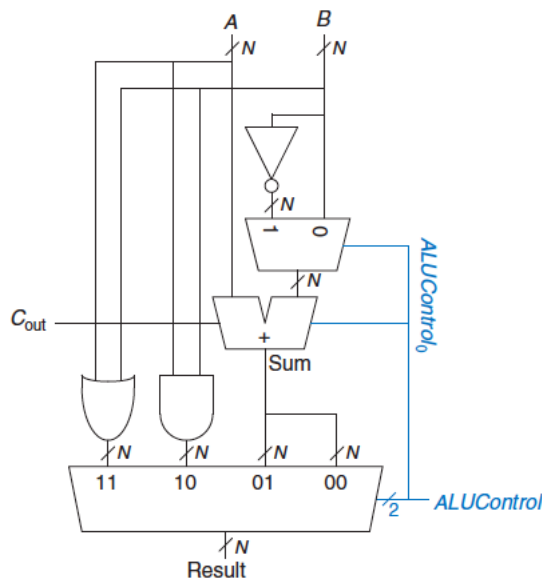
Subtraction between binary numbers can be done via the Two's Complement numbers: if we want to do $A - B$, we'll do an addition between A and the Two's Complement of B . That's the same logic of an adder, only with modified data. It doesn't work with overflows though.

Comparators are building blocks that, as the name says, compare two numbers. There are two types of comparators:



- **equality comparators:** those comparators compare if two numbers A and B are equal, via using a XNOR gate for each column of the numbers;
- **magnitude comparators:** those kind of comparators compute the subtraction $A - B$ and, based on the sign of the result, they output if a number A is greater, smaller, or equal to a number B . They don't work with overflow.

4.1.3 Arithmetic Logic Unit (ALU)



ALUControl [1:0]	
Operation	Signal
ADD	00
SUB	01
AND	10
OR	11

Arithmetic Logic Units (ALUs) are one of the most important building blocks in a processor. They are circuit that are able to perform multiple types of operations on two numbers A and B , such as AND, OR, **addition** (ADD) and **subtraction** (SUB).

But how does an ALU work?

- 1) First of all, we need a signal to let the ALU know what we want it to compute. It is going to compute everything on the same time, although we are going to select only one output via a 4:1 multiplexer;
- 2) We then proceed to consider the final 4:1 multiplexer: we know that with ALUControl equal to 00 we do the sum, so 00 will make the multiplexer return the output of an adder; with 01 we have to return the subtraction, so it has to return the output of an adder with \bar{B} and with a C_{in} equal to 1; with 10 we want to do the AND, while with 11 we want to do the OR;
- 3) We can unify some elements: for instance, we can unify the adder of both the sum and the subtraction. Since they share the same common MSB of the ALUControl, we can use the bit in position 0 to decide whether to make the sum or the subtraction: more in practice, 0 means sum and 1 means subtraction;
- 4) ALUControl₀ becomes both the C_{in} and the select bit for the multiplexer that selects between B and \bar{B} . The C_{out} of the adder is drawn out of the circuit;
- 5) Since the adder is only one, then we propagate the output of the adder to both the input ports 10 and 11 of the final multiplexer.

The ALU as of now it's not complete though: how do we handle overflows for instance? We can output some other data via other flags that allow the CPU to have more information regarding the operations made by the ALU.

Let's find a way to output the presence of a **Carry**: we can draw from the ALU the C_{out} signal, but that's not enough: in case of an AND or an OR we'll probably have a C_{out} signal, but we don't want it to be out put, because that data is not relative to the operation that we are performing. We thus use a control bit to make the ALU output the carry only if we are actually performing an arithmetic operation. This bit is $\overline{\text{ALUControl}}_1$: whenever

$ALUControl_1$ is 0 it means that we are doing an arithmetic operation, so we can use it and rest assured that the carry flag will be valid only for when an arithmetic operation gets performed.

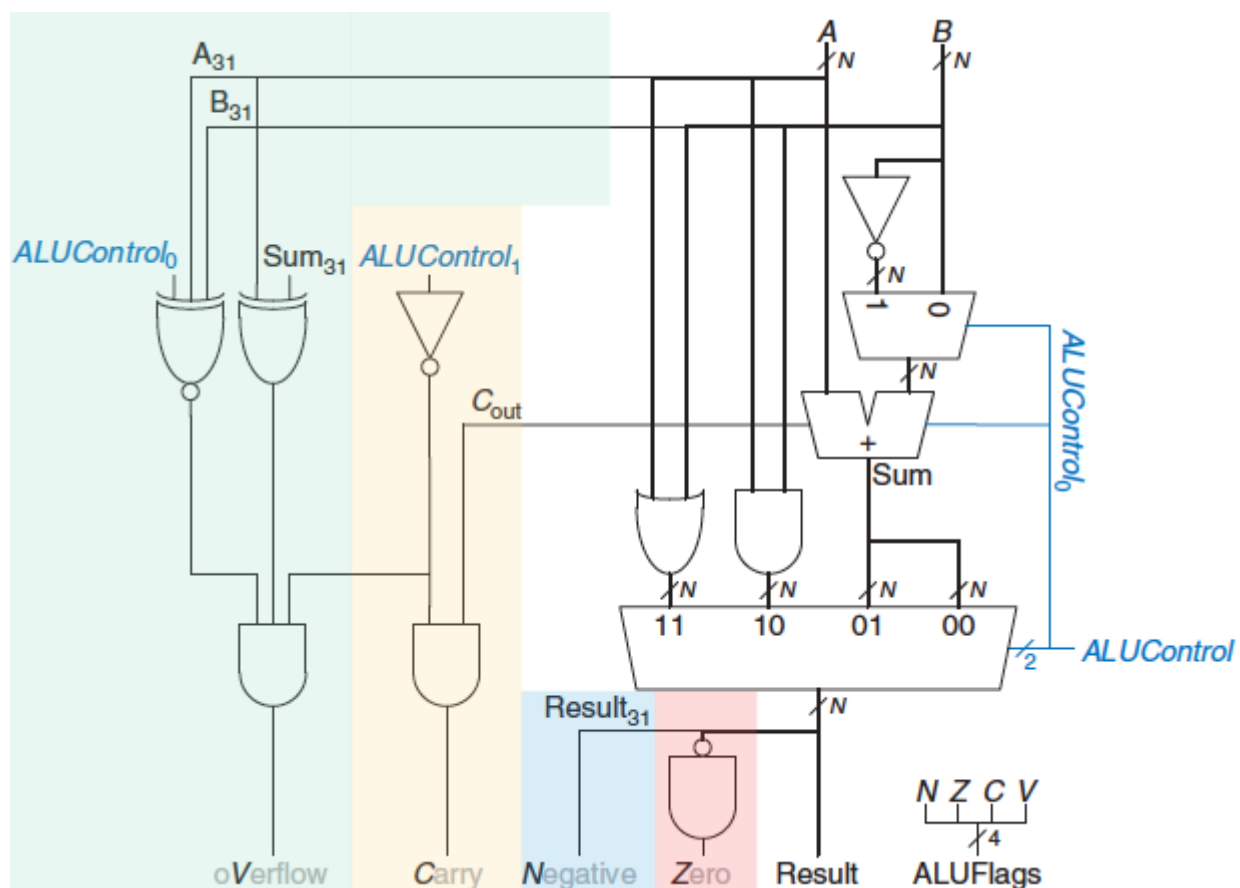
Now, after the result gets through the 4:1 multiplexer, we can make other two flags: **Negative** is detected through $Result_{31}$, since the MSB is usually the sign bit in signed numbers; **Zero** is 1 when all numbers are 0: in order to detect it, we make the NOR between all the bits (or, thanks to DeMorgan, the AND of all the negated bits).

How do we detect **Overflows**? We have an overflow if the following 3 conditions are met:

- 1) We are performing an addition or a subtraction (thus if $ALUControl_1$ is 0);
- 2) If the sign of a sum and the one of A are the opposite (so $A_{31} \oplus Sum_{31}$);
- 3) If A and B have the same sign and we are performing a sum or if A and B have different sign and we are performing a subtraction (so $\overline{ALUControl_0} \oplus A \oplus B$)

In general, we have the following situation:

$$\text{Overflow} = \overline{ALUControl_1} \times A_{31} \oplus Sum_{31} \times \overline{ALUControl_0} \oplus A \oplus B$$



4.1.4 Shifters and Rotators

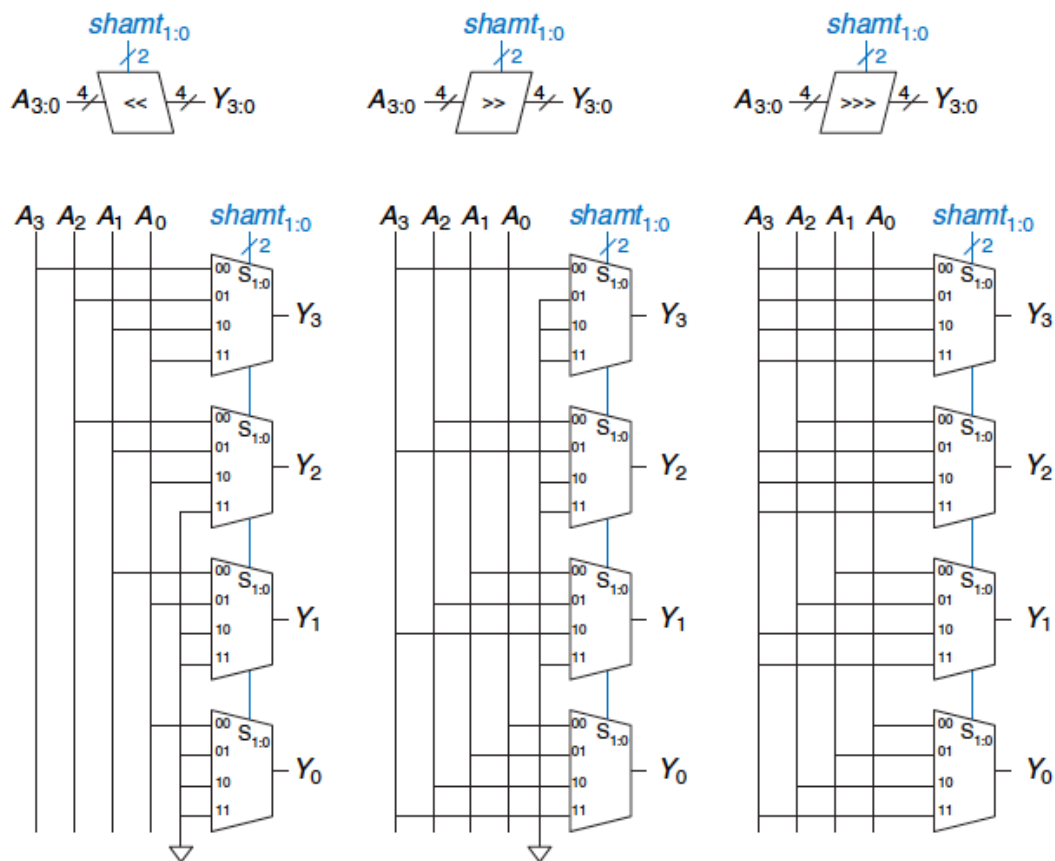
There are 2 types of shifters:

- **Logical shifter:** shifts a value to the left or right and fills empty spaces with 0's;
- **Arithmetical shifter:** when using logical shifters, problems with Two-Complement numbers would arise; in order to avoid these problems, Arithmetical Shifts were taken into account when designing circuits. Arithmetical shifters are essentially equal to logical shifters, but when it has to shift to the right, it fills the blank spaces with the MSBs;

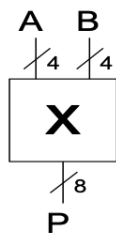
We then introduce **rotators**, which rotate bits in a circle, such that bits shifted off one end are shifted into the other end. For example:

	ROR 2	
00110	00011	01001
Initial state	1 st rotation	2 nd rotation

Rotators can rotate an X number either to the right (X ROR N) or to the left (X ROL N) by N times.

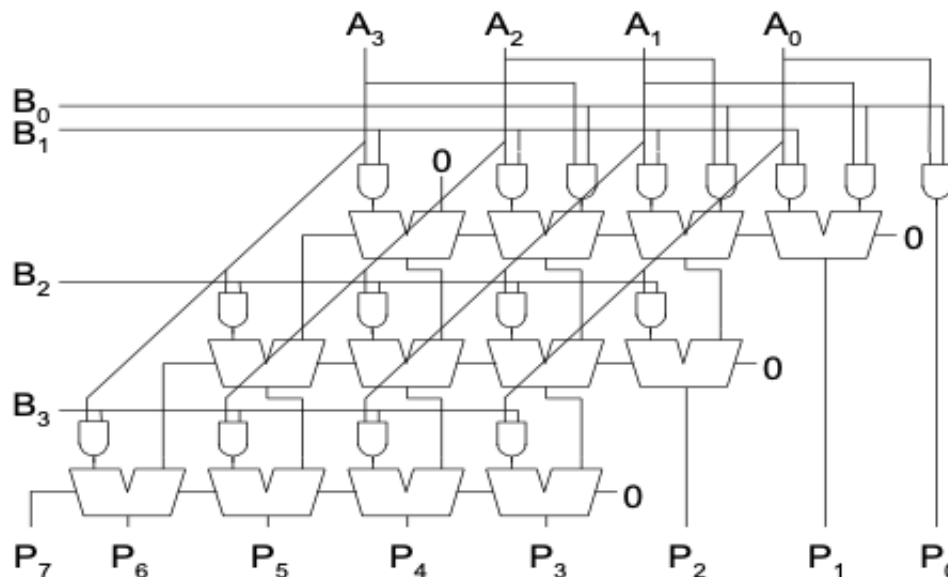


4.1.5 Multipliers



Multipliers do multiplications between two N -bit numbers, returning a $2N$ -bit result. When doing a multiplication, the partial product is formed by multiplying the same digit of the multiplier with the multiplicand. After having multiplied all the numbers, the shifted partials products get summed in order to obtain the final result.

Multiplicands (so the lines that are obtained by multiplying the number A for each digit of B) are either $A_{N:0}$ or 0. We may notice here how a multiplication is nothing more than an AND gate between a cipher A_i and a cipher B_j . There is a difference from signed and unsigned multiplication though: if we consider for instance the two numbers 0xFE and 0xFD we get two different results depending if we consider such numbers as signed or unsigned. If we consider them unsigned (so), we get 0xFB06, while if we consider them as signed we get 0x0006. We call the process of multiplying two numbers and then adding such result to a third number **Multiply ACcumulate** operations (or MACs)

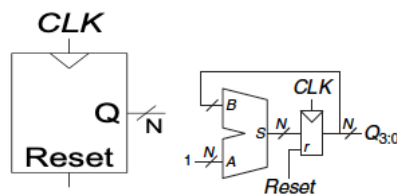


What about divisions though? If the division is between a number and a power of two, we can use shifters; if the division is with a number that is not a power of two, then we use dividers. Dividers are expensive and have high latency (dividing a N -bits number has a latency of N^2), so they aren't used that much. In many programming languages (like C) developers prefer to stick to data structures that have sizes based on the powers of two.

4.2 Sequential circuits

So far we saw only combinational circuits. These following circuits are called sequential because they have a memory.

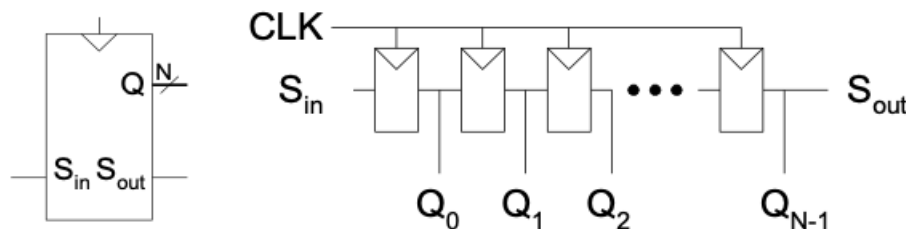
4.2.1 Counters



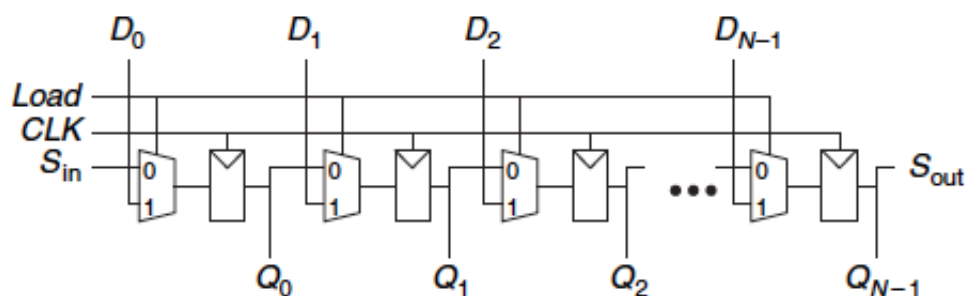
As the name says, they count numbers. On each clock edge, they produce a number, up to 2^N (where N is the size of the output). They are used to cycle through numbers. For example, for $N = 3$, we'll have 000, 001, 010, \dots , 110, 111, 000, 001, \dots . They are different from registers despite the form, since they don't have any input, and they have only one output and a reset signal.

4.2.2 Shift registers

On each clock edge, the shift registers shift in a new bit and shift out another bit. It can be seen as a **serial-to-parallel** converter, since it shifts bits serially (one by one, through S_{in}) to a parallel output ($Q_{0:N-1}$). After N cycles, the past N inputs are again available in parallel in Q .



There are also shift registers with *parallel load*, which are basically shift registers but with two more inputs: one is the load signal, and the other is D (so a piece of data whose bit width is $0 : N - 1$). When load is 0, then the whole circuit acts as a normal shift register (so it takes one bit per clock cycle and stores it into the registers); when load is 1 instead, the flip-flops load the bits from D . This circuit acts as a **serial-to-parallel converter** (S_{in} to $Q_{0:N-1}$) when load = 0, but when load = 1 it becomes a **parallel-to-serial converter** ($D_{0:N-1}$ to S_{out}).



Shift registers can be useful when making a serial transmitter: we could have a signal D in parallel and, for instance, we could need to send it to another part of the circuit. One way to do so could be to use a shift register with load = 1 to transform the parallel signal into a serial signal, send the serial signal somewhere else in the circuit, and then bring it again in parallel through another shift register with load = 0.

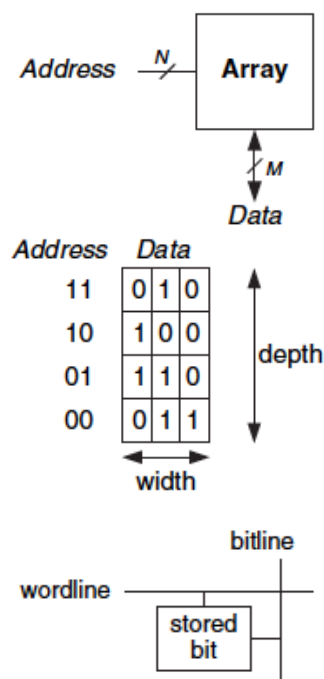
Why should a computer use serial instead of parallel communication in some cases? Firstly, it's because of **physical reasons**, since serial avoids the **crosstalk** effect (so the possible damaging or loss of data on long distances), secondly, serial communication

avoids the problem of having all the bits synchronized (in parallel, all the bits must arrive at the destination on the same time, which could be an issue when dealing with sensitive data that must travel on long distance. Serial instead doesn't need the data to arrive all at the same moment), and last, even if the data sent per clock cycle is less, the cycles are much smaller than the ones needed for parallel communication

4.3 Memory

Registers and flip-flops are memories made for small amount of data, so to store much larger data we use memory arrays (since they are even more efficient). There are 3 types of memory arrays, which are differentiated in 2 big classes:

- **RAM: Random Access Memory** (which is considered **volatile**):
 - **DRAM: Dynamic Random Access Memory** (uses capacitors);
 - **SRAM: Static Random Access Memory** (uses cross-coupled inverters);
- **ROM: Read Only Memory** (considered **non-volatile**, ROM is not anymore a Read-Only memory: this name was used in the '60, and eventually the name remained the same.), it's quick to read, but it's slow to write (if not impossible in some cases, such as in the past).



M -bit data are stored in particular N -bit addresses (the number of available spaces are equal to 2^N). Memory arrays can be used even to implement logic functions.

We define:

- **Depth:** number of rows, equal to 2^N ;
- **Width:** number of columns, equal to the size of the data we are storing inside the array (M);
- **Array size:** depth \times width = $2^N \times M$

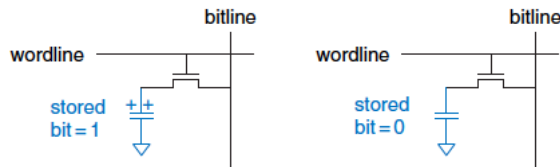
Memory arrays are called like that because they are arrays of memory cells. Generally, we have a **wordline** (the wordline is the line that transmits the address bits) and a **bitline** (which contains the bits relative to the data).

The wordline is considered as an enable signal. All the wordlines derive from a decoder. In a memory array, a single row is read / written at the same time, which corresponds to a unique address. Only one wordline is set to HIGH (1) at once.

When the wordline is equal to 1, the stored bit is transferred **to** or **from** the bitline; if the wordline is equal to 0, then the bitline is disconnected from the cell. The circuits for storing data change from manufacturer to manufacturer, so there is no standard way.

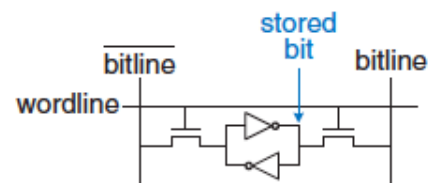
4.3.1 RAM

RAM is **volatile** (it loses data when the computer is turned off), but it's read and written quickly. The main memory in computers is RAM (DRAM). It was historically called random because any data word is accessed as easily as the others. Previously, we would have to scan the whole memory (in OS' like MsDOS).



In the DRAM, bits are stored on capacitors, and the connection between the wordline and the bitline is granted through an nMOS transistor. It's called **dynamic** because the value needs to be rewritten both periodically and both after it has been read (charge leakage from the capacitor degrades the value, and reading destroys the stored value).

SRAM doesn't need to be refreshed periodically (that's why it's called static), which is why sometimes is preferred over DRAM. Data bits are stored using cross-coupled inverters. In some cases also a bitline signal can be present in the cell. The CMOS fabrication standard prefers to use two NOT gates instead of two BUF gates because in order to make a BUF gate you would need two NOT gates.

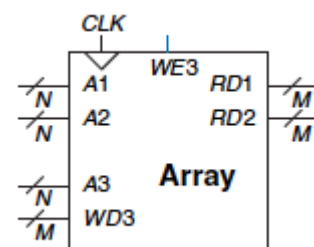


4.3.2 Memory trade-offs and ports

Memory type	Transistors per cell	Latency
flip-flops	~ 20	fast
SRAM	6	medium
DRAM	1	slow

The more transistor a device has, the less the latency there will be, but not as the costs. DRAM has a higher latency because the charge must move from the capacitor to the bitline. The size of the memory impacts too, since a larger memory has a higher latency.

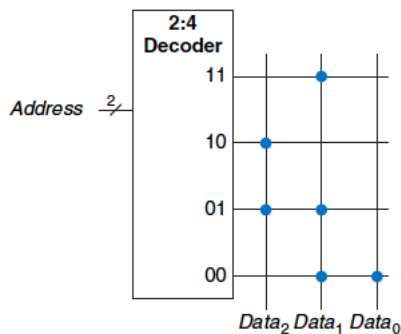
With multiple registers we can create a **register file**, which is a group of registers that is usually implemented in some circuits to keep track of some piece of data. Usually register files allow to read from two registers at the same time and to write on one register at a time. Similarly, memory can be built as an array in the figure: given two addresses, we can access to two pieces of data into the memory on the rising edge of the clock, and given an address A_3 , we can write some content on that address (the content is passed through WD_3).



4.3.3 ROM

ROM is non-volatile (it retains data even when there is no power), it reads quickly but the writing is, depending on its technology, either slow or impossible. Flash memory

in cameras or thumb drives are examples of ROM. Previously ROM couldn't be edited, but today it can be reprogrammed.

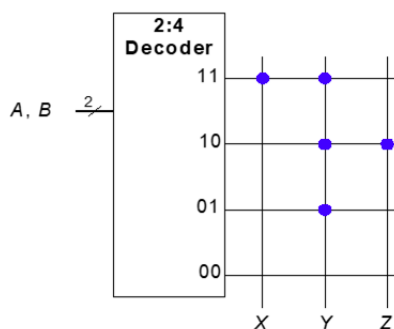


ROM can also be used in combinational logic, either to perform operations or to store some data. It can also be used in sequential circuits (for instance in the FSMs as the combinational parts). The notation for ROM is either the dotted one (as in the image on the left) or the one with gates (normally used when performing combinational logic with ROM).

A way to use ROM to implement logic is the following: suppose that we have the following Boolean equations:

$$X = AB \quad \text{and} \quad Y = A + B \quad \text{and} \quad Z = A\bar{B}$$

A	B	X	Y	Z
0	0	0	0	0
0	1	0	1	0
1	0	0	1	1
1	1	1	1	0

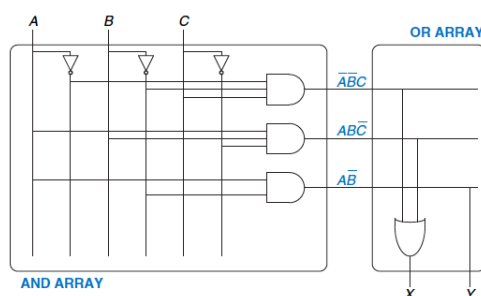


First we have to produce the truth table of the three equations, then we place a dot on each intersection between the bitlines and the wordlines for each 1 in the output. Each bitline represents one output, and each wordline represents a different combination of inputs. The input then becomes the address of the ROM, which will then output for each bitline the corresponding output for all the equations.

Regarding sequential circuits, ROM can be used for example to store the lookup table of the combinational part which determines the next state of the circuit and the outputs. The inputs could be the current state and the input, and the outputs could be the next states and the circuit outputs. After being output from the ROM, the bits

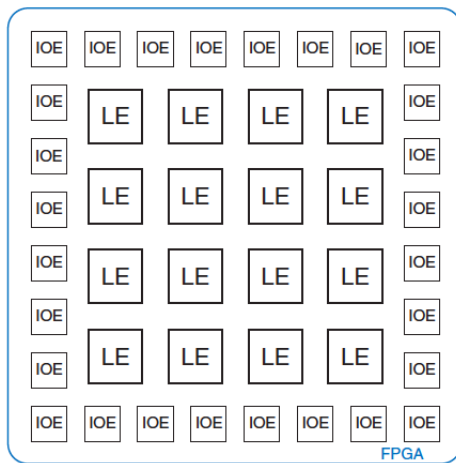
that encode the next state have to be saved first in a register, then after being registered they can be brought back as inputs of the ROM.

4.3.4 PLAs and FPGAs



A while ago, we talked about two-level logic and multilevel logic. Now we'll further expand such topics. An example of two-level circuit is the **Programmable Logic Array** (PLA): such circuit consists of an array of AND gates followed by an array of OR gates. PLAs are used only for combinational logic, and have fixed internal connections.

We said that PLAs only work with combinational logic. A particular type of re-programmable circuit that can work also as sequential logic is the **Field Programmable Gate Array** (FPGA).



FPGAs are arrays of re-configurable gates, and can be re-programmed via some software that allows to simulate an actual circuit. They are made mainly of 3 components:

- **Logic Elements (LEs):** they are elements able to perform logic operations (also called CLBs). They are made out of multiple elements:
 - **Look-Up Tables (LUTs):** they perform combinational logic;
 - **flip-flops:** they perform sequential logic;
 - **multiplexers:** they connect the LUTs and the flip-flops;
- **Input/Output (I/O) Blocks (or Input/Output Elements, IOEs):** they serve as an interface with the outer world;
- **Programmable Interconnections:** they connect LEs and IOEs.

Some FPGAs have also some other building blocks such as RAM and multiplexers. FPGAs are more versatile of PLAs, because they allow for more complex circuits, and they also give the possibility to reproduce sequential circuits. In general, FPGAs are also more powerful.

LEs are set via a CAD tool (**Computer-Aided Design**): we first enter the design of the circuit that we want using either a schematic or an hardware description language (HDL), then we simulate the design to see if the outputs of the circuit match with the desired outputs, then the circuit gets synthesized to compute which is the most optimal way to make it (so defining the configuration bitstream) and finally it gets downloaded onto the FPGA.

Chapter 5

Hardware Description Languages

placeholder

Chapter 6

Computer Structure

A computer is made out of various components: CPU, memory, IO (Input/Output).

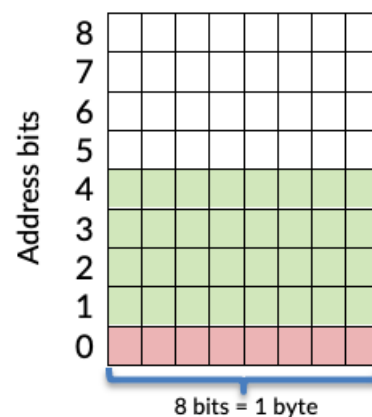
6.1 Memory

Inside a computer, there are various type of memory. An example is the **RAM**, which is used to store data while the computer is running. RAM is volatile, so if power gets cut off then all the data stored will be deleted. Historically, RAM were made out of groups of 8 flip-flops. We have two types of RAM: **Static** and **Dynamic**.

- **Static RAM (SRAM)**: is faster but more expensive than DRAM, and uses a different technology rather than flip-flops;
- **Dynamic RAM (DRAM)**: is slower but cheaper, and is made out of capacitors; capacitors can “remember” a bit for a few milliseconds, so they need to be recharged every then and while; recharge though take time, so that is why DRAM is smaller;
- **Flash Memory**: is slower than DRAM, but is cheaper, plus it’s non-volatile.

Generally in the RAM, every level of flip-flops (so a byte) is marked by an address bit. We can use 1 byte to store integers, generally in 2’s Complement notation (so a number x is stored following the equation $2^8 - x$). But numbers in 2’s Complement can represent numbers in the range of $[-128; 127]$, so what if we need to store bigger values?

We call a **word** a group of **32 bits**. In a RAM we would need **4 bytes** to store a word. A string has a different way of being encoded, and it’s for this reason that in the 60’s the **ASCII** encoding was created. ASCII encoding encodes every letter, **every character** into a **number**, so for example the letter A is 65, B is 66, C is 67, etc... Even the new line has an encoding.



Curiosity: Encoding of the newline (`\n`)

Back in the days, Unix systems would encode the new line as 10, macOS (back then it was called macOSX) as 13 and DOS based systems with both 10 and 13.

This happened because, within the printers' drivers, 10 would be used to roll down paper, while 13 was used to bring the head of the printer to the beginning of the line.

So how do computers **encode** strings? We need various bits, such as for example a bit that expresses the **length** of a string (typically in the 70's the length was saved in the first byte, so that we would know in advance how much memory would be needed in order to store the word. Now this method isn't used anymore) or a bit that expresses when the **word starts** and one that expresses the **end of the word**. It all depends on the encoding that we choose. For instance, **ASCII** and **Unicode** don't use any byte to encode the string's length or its beginning or end.

6.2 CPU

As we previously mentioned, a computer is made of multiple parts, and the main core of it is called CPU

Definition: Central Processing Unit (CPU)

A **CPU** (also called processor) is the active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

Instruction 1
Instruction 2
...
Byte 1
Byte 2
Byte 3

Imagine if in a CPU we have 3 bytes of memory, and we want to store into the 3rd byte the sum between the 1st and the 2nd byte. What would we need? We would first need an ALU, capable of summing the data, and then a memory. In **RISC-V** processors we have a series of **32 registers**, which are called from **x0** to **x31**. An important thing about registers is that in RISC-V processors the register **x0** will always be full of zeroes, and if we try to store something in it, it won't work.

In machine code, we would make the processor do a given operation between the data contained into some registers. Let us consider the following example:

```
add x3, x1, x2
```

we would do the sum between register 1 and 2 and store the result into register 3. But how does a CPU remember what to do? Previously **there was a separate memory** built into the CPU dedicated only to store the instructions, but **now** in more modern CPUs it's **all built in into the**

main registers set.

Each instruction needs **4 bytes** to be saved, so each 4 bytes we can find an instruction, starting from the memory address **0x00400000** (so 0x00400000, 0x00400004, 0x00400008, 0x00400012, ...), while words start from address **0x10100000**. Each RISC-V CPU needs to remember the operation that he is doing, and that's where the **PC (Program Counter)** comes in handy: the PC is a temporary memory that gets edited each time that an operation is being done.

6.3 Programs and instructions

We consider a **program** as a **set of instructions**, which is also called **text**. But how do we encode the instructions in the registers? It's important to say that there are different formats of instructions, because of different needs.

Let us suppose that we have a simple instruction like `add x3, x4, x5`: we want to sum `x4` and `x5` and store their sum into `x3`. The encoding for `add` is the **R-Type**, and it works in this way:

F7	2 nd register	1 st register	F3	Destination Register	Opcode
From 31 to 25	From 24 to 20	From 19 to 15	From 14 to 12	From 11 to 7	From 6 to 0
0 0 0 0 0 0 0	0 0 1 0 1	0 0 1 0 0	0 0 0	0 0 0 0 1	0 1 1 0 0 1 1

The encoding includes 6 parts:

- 0-6 **Opcode**: stands for **Operation Code**, it specifies the operation that must be done;
- 7-11 **Destination Register**: it's the address of the register where we want to store the result of our operation;
- 12-14 **F3**: it stands for **function 3**, and it's a further specification of the opcode. That happens because internally, in the CPU, much instructions are similar, so in order to differentiate them, we need a further encoding layer;
- 15-19 **1st register**: it's the address of the **first register**;
- 20-24 **2nd register**: it's the address of the **second register**;
- 25-31 **F7**: like F3, it stands for **function 7**, and it's a further specification of the operation that must be done.

Let's try with the subtraction instead:

`sub x7, x5, x7`

Here is shown the full code of the instruction:

F7	2 nd register	1 st register	F3	Destination Register	Opcode
From 31 to 25	From 24 to 20	From 19 to 15	From 14 to 12	From 11 to 7	From 6 to 0
0 1 0 0 0 0 0	0 0 1 1 1	0 0 1 0 1	0 0 0	0 0 1 1 1	0 1 1 0 0 1 1

We note that the opcode is the same for both addition and subtraction, but **F7** differs. What about if we want to move data from a register to another? There is no instruction for moving data between registers, so how do we do it? We can use the fact that the register **x0** is **always full of zeroes**. We can then for example do `add x8, x10, x0`, or `sub x8, x10, x0`.

Below here is shown a table with all the instructions available for RISC-V processors:

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	$rd = rs1 + rs2$	
sub	SUB	R	0110011	0x0	0x20	$rd = rs1 - rs2$	
xor	XOR	R	0110011	0x4	0x00	$rd = rs1 \oplus rs2$	
or	OR	R	0110011	0x6	0x00	$rd = rs1 \mid rs2$	
and	AND	R	0110011	0x7	0x00	$rd = rs1 \& rs2$	
sll	Shift Left Logical	R	0110011	0x1	0x00	$rd = rs1 \ll rs2$	
srl	Shift Right Logical	R	0110011	0x5	0x00	$rd = rs1 \gg rs2$	
sra	Shift Right Arith*	R	0110011	0x5	0x20	$rd = rs1 \gg rs2$	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	$rd = (rs1 < rs2)?1:0$	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	$rd = (rs1 < rs2)?1:0$	zero-extends
addi	ADD Immediate	I	0010011	0x0		$rd = rs1 + imm$	
xori	XOR Immediate	I	0010011	0x4		$rd = rs1 \oplus imm$	
ori	OR Immediate	I	0010011	0x6		$rd = rs1 \mid imm$	
andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$	
slli	Shift Left Logical Imm	I	0010011	0x1	$imm[5:11]=0x00$	$rd = rs1 \ll imm[0:4]$	
srlr	Shift Right Logical Imm	I	0010011	0x5	$imm[5:11]=0x00$	$rd = rs1 \gg imm[0:4]$	
srai	Shift Right Arith Imm	I	0010011	0x5	$imm[5:11]=0x20$	$rd = rs1 \gg imm[0:4]$	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 < imm)?1:0$	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 < imm)?1:0$	zero-extends
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$	
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$	
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$	
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$	zero-extends
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$	zero-extends
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$	
sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$	
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$	
beq	Branch ==	B	1100011	0x0		$\text{if}(rs1 == rs2) PC += imm$	
bne	Branch !=	B	1100011	0x1		$\text{if}(rs1 != rs2) PC += imm$	
blt	Branch <	B	1100011	0x4		$\text{if}(rs1 < rs2) PC += imm$	
bge	Branch ≥	B	1100011	0x5		$\text{if}(rs1 \geq rs2) PC += imm$	
bltu	Branch < (U)	B	1100011	0x6		$\text{if}(rs1 < rs2) PC += imm$	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		$\text{if}(rs1 \geq rs2) PC += imm$	zero-extends
jal	Jump And Link	J	1101111			$rd = PC+4; PC += imm$	
jalr	Jump And Link Reg	I	1100111	0x0		$rd = PC+4; PC = rs1 + imm$	
lui	Load Upper Imm	U	0110111			$rd = imm \ll 12$	
auipc	Add Upper Imm to PC	U	0010111			$rd = PC + (imm \ll 12)$	
ecall	Environment Call	I	1110011	0x0	$imm=0x0$	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	$imm=0x1$	Transfer control to debugger	

What if for example we want to add a **fixed real number** to a register? We can use the **addi** instruction: this instruction will sum a real number to a register, but we would have to **change encoding** type. Why do we have to? Because we don't need a second register to be added, so we can just reserve the **12 MSBs** to the number (this way we can sum a number in the range $[0, 2047]$). This is called **I-Type encoding** (because of the presence of the **immediate** field):

Immediate	1 st register	F3	Destination Register	Opcode
From 31 to 20	From 19 to 15	From 14 to 12	From 11 to 7	From 6 to 0
0 0 0 0 0 0 0 1 0 1 0 1	0 0 1 1 1	0 0 0	0 0 1 0 0 0	0 0 0 1 0 0 1 1

An example of the addi instruction could be

```
addi x8, x7, 21
```

Another important instruction is the **Load Upper Immediate** (**lui** *xDest*, *imm*). This instruction is used to load a **20-bits wide immediate** into a register, but it will need another encoding format. We need the **U-Type**:

Immediate	Destination Register	Opcode
From 31 to 12	From 11 to 7	From 6 to 0
0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 0 0	0 0 1 0 0 0	0 0 0 1 0 0 1 1

If we need to load a word from the memory into a register we can use the **Load Word** instruction (`lw xDest, address`). The `lw` operation has **20 bits** for the memory address, giving the possibility of having **at least 220 addresses** (around **1MB** of memory). We can also use a **custom bias** to reach further memory addresses up to **212** (for example if the maximum that our register can save is 2048 but we need to access to the address 2052, we can do `2048+4`; the operation will be `lw x16 4(x18)`). The address **has to contain a register**, but the **bias isn't necessary**. The **bias** can be a **hexadecimal** number too, but it must be **lower than 12 bits**.

For traditional reasons, we start to store data in memory after the address **0x10010000**.

Exercise 6.3.1

Load the words from addresses **0x10010000** into register **x3** and **0x10010004** into register **x4**.

```
lui x5, 0x10010
lw x3, 0(x5)
lw x4, 4(x5)
```

First we store part of the address into a register (so the first 5 hexadecimal digits, since they represent the first 20 MSBs in binary) with Load Upper Immediate (`lui`) and then we can use a bias up to 212 to fix the address.

What about a larger address? Let's try with `0x67F59C1D`; we can do:

```
lui x5, 0x67F59
ori x5, x0, 0xC1D
lw x6, 0(x5)
```

This way with the `ori` instruction we can compute the whole number and store it into a register.

But what if we want to store a word into a memory address? We use the **Save/Store Word** instruction (`sw reg2, bias(reg1)`), which uses the **S-Type** encoding format:

Immediate [11:5]	2 nd register	1 st register	F3	Immediate [4:0]	Opcode
From 31 to 25	From 24 to 20	From 19 to 15	From 14 to 12	From 11 to 7	From 6 to 0
0 0 0 0 0 0 0	0 0 1 0 1	0 0 1 0 0	0 0 0	0 0 0 0 1	0 1 1 0 0 1 1

The immediate is used to store the **bias / offset**, since the **address** is **stored** into the **1st register**. This instruction stores the word of the **2nd register** into the **memory address** given by the sum of the content of the first register and the immediate.

Exercise 6.3.2

Save the sum of the previous two words into the memory address `0x10010008`.

```
add x6, x3, x4
sw x6, 8(x5)
```

When we don't have to use anymore the CPU (or the computer in general) it's a good habit to terminate the process. In order to terminate the process, we have to make a call to the **Operating System** with the instruction **ecall**. Depending on the content of the register **x17**, the **ecall** will have different results. In order to terminate the process we have to store into the register **x17** the number **10**. So the instruction would be:

1	Prints an integer
4	Prints a string
10	Exits

```
ori x17, x0, 10
ecall
```

ecall doesn't need any parameter, it's all within the instruction. As we previously said, **ecall** can do more rather than just terminating the process, and depending on the content of **x17** it can do multiple things: it can for example print in console some numbers or a string.

The instruction **beq** (**B**ranch if **E**qual) takes three inputs **reg1**, **reg2** and an **offset**: if the content of **reg1** is equal to the content of **reg2**, then this operation will store into the **PC** (Program Counter) the **value** of the **PC** itself **plus an offset**. To **encode** the instruction we need the **B-Type** encoding (note the difference with the S-Type: the immediate is encoded with the bits 12:1, plus the bit in position 11 is stored into another part of the instruction):

Immediate [12 10:5]	2 nd register	1 st register	F3	Immediate [4:1 11]	Opcode
From 31 to 25	From 24 to 20	From 19 to 15	From 14 to 12	From 11 to 7	From 6 to 0
0 0 0 0 0 0 0	0 0 1 0 1	0 0 1 0 0	0 0 0	0 0 0 0 1	0 1 1 0 0 1 1

Similar instructions are the **bne** instruction (**B**ranch if **N**ot **E**qual), **blt** (**B**ranch if **L**ower **T**han), **bge** (**B**ranch if **G**reater or **E**qual). Why don't we have a Branch if Lower or Equal Than instruction? Because for that we can simply use the **bge** instruction with the registers inverted. These instructions may exist in some compilers, but then they would be "*translated*" in other instructions. Such instructions are called **pseudo-instructions**.

For example, if we have **beq x5, x6, 0x0c** and if the content of **x5** is equal to the content of **x6**, then the CPU will load into the **PC** the operation at **0x00400000 + 0x0C**.

6.4 Architecture of the CPU

When executing an instruction, the steps are mostly the same: first, the program counter updates the memory address of the instruction, and second, the instruction gets decoded and executed. In order to explain the architecture of a RISC-V CPU, we'll make a distinction of 3 types of instructions:

- **memory-reference**: it's the set of all instructions that require the CPU to **access** the **memory**, either for **storing** or **retrieving** data;
- **arithmetic-logical**: it's the set of all the instructions that involve the **manipulation** of the registers' data, so the sum, the subtraction, the AND between two values, the OR, and so on...;

- **branches:** it's the set of instructions that involve the **change** of the **PC's address**, so the instructions like `beq`, `blt`, `bgt`, etc...

Normally, all the instructions require the use of the ALUs, and then their path differs depending on the aforementioned classes.

We previously saw that a CPU is built of various components, such as for example the **set of registers**, the **program counter** and the **ALUs** (we can see it more clearly in the [Figure 6.1](#)) In the following pages we'll proceed to analyse the three main components of a CPU.

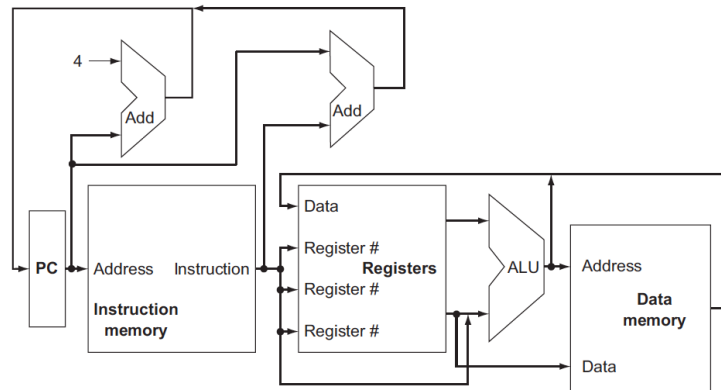


Figure 6.1: High-level representation of a multi-purpose CPU

6.4.1 CPU's main components

We start by analysing more in the detail what a **PC** is. Essentially, since it's a **register** itself, the program counter has 3 inputs (a **CLK** signal, a **RESET** signal and an input for the **current instruction address**) plus **one output** (which is equal to the input. We recall that the next instruction address is computed elsewhere).

The memory isn't built inside the CPU itself, but there is a little circuitry that lets the CPU interface with the memory: such circuitry is represented by a bus, which connects the CPU to the RAM. These two components are also connected to an important circuitry: the **I/O Bridge**. Such bridge is the only way for the user to communicate with the computer. From the I/O Bridge there are two important busses:

- the **System bus**: it connects the I/O Bridge with the CPU;
- the **Memory bus**: it connects the I/O Bridge with the RAM.

All the bridges transfer signals in 32 bits, since operations and data are also stored in groups of 32 bits.

How is data stored and read into the registers though? There are two types of storing/reading options:

- **Big endianness**: the data are stored/read from the **MSB** to the **LSB**;
- **Little endianness**: the data are stored/read from the **LSB** to the **MSB**;

Nowadays most of the CPUs use little endianness, but back in the previous century most computers used big endianness.

We know that the memory is made of a **decoder** that, given an address, it outputs the needed data in either 32 or 64 bits. The memory is accessed in two different moments:

first, the CPU accesses to the **instruction memory**, in other words is where all the instructions of the program are stored; then, after all the data pass through the ALUs, the CPU can access to the **data memory**, where we can either store or retrieve the needed data.

The CPU contains a **register file** (a group of **32 registers**) that has **four inputs**:

- two inputs are for the **addresses** of the registers the computer wants to access, so reg[1] and reg[2] (we recall that the addresses are in 5 bits);
- the other two are used when we have to store into the registers the result of an operation. These two inputs are called reg[3] and data.

The register file has also **two outputs**, since in the same clock cycle we can **access up to 2 registers** simultaneously.

6.4.2 Further developing the architecture

In the [Figure 6.1](#) we saw a high-level implementation of a CPU, but obviously, because it's an approximate image, there are some circuits that are missing:

- first of all, in the image all the circuits are wired to each other, but that would mean that some data, once processed, would just mix with other data. This can't happen, we have to select from where data should come from, and which data has to be processed. We need then a circuit that is capable of selecting from where the data should come. This circuit can be just a **multiplexer**, or **data selector**, but it's important in order to compute the right results;
- the second one is a circuit that, once it recognises an instruction, should tell the various components how to behave. This is necessary, since we saw that there are three main types of instructions, and each instruction (also inside the macro-categories that we described earlier) has different scopes and different ways to compute the results that we need. This circuitry is called **control unit**.

Given these previous considerations, we may sketch a new circuit, that we can see in [Figure 6.2](#). We can see that the instruction not only selects the **registers**, but now it also goes to the **control unit**. Furthermore, we notice how the zero signal of the ALU is connected, via an AND gate, to the Branch signal of the control unit. In addition, there are now 3 multiplexers: one is in between the output of the register file and the ALU; the second selects the data that goes into the register

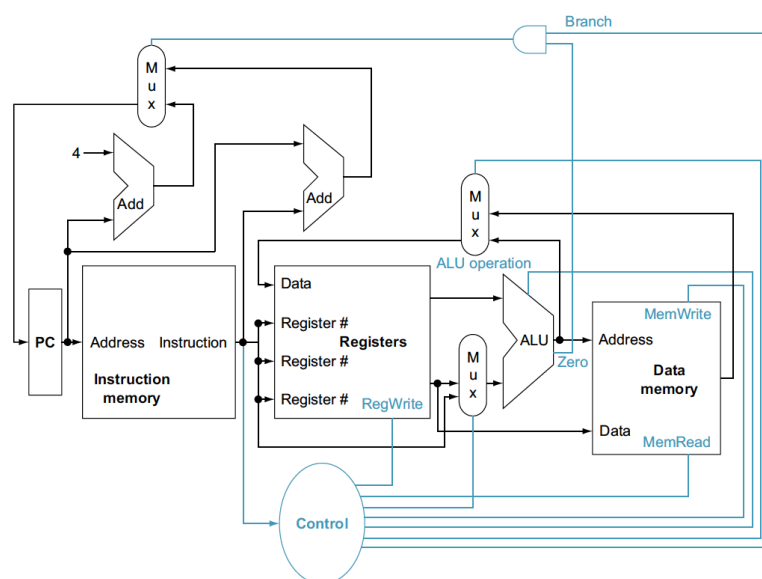


Figure 6.2: Improved circuit in respect to [Figure 6.1](#)

file (so if it has to be the result of the ALU computations or the data of the memory) and the third that selects, depending if we have to branch or not, the value that gets added to the PC.

An important detail regarding the CPU is that all the circuits that handle the data values are **combinational**, while the others are mostly **sequential** circuits: for instance, the ALU is a combinational circuit, while for example the memory is sequential. We know that sequential circuits act only on the positive edge of the CLK signal (the posedge in SystemVerilog): the PC for instance is one of these sequential elements, as well as the memory and the register file.

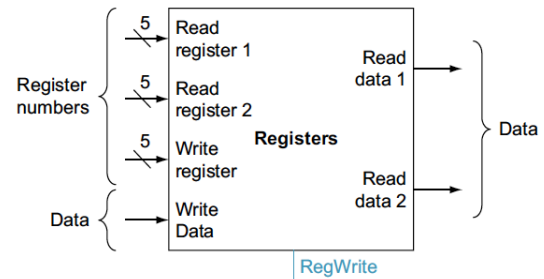


Figure 6.3: A closer inspection to the register file

Now that we have a better idea of how our architecture works, we could start by looking if some instructions could work on it. Thus, let us assume that we are asking the computer to run this instruction:

```
add x5, x6, x7
```

These are (roughly) the operations that will be done in order to accomplish the task:

1. the **PC** passes to the **instruction memory** the address of the instruction, and passes the current address to the adders on top in order to get the next instruction;
2. the address is passed to the **instruction memory**, which retrieves the instruction and outputs it;
3. the instruction is distributed:
 - to the adder in case of a branch (even if it's not our case);
 - to the register file, where we select reg[1] as x6, reg[2] as x7, and finally reg[3] as x5;
 - to the **control unit**, where the multiple signals are controlled (we'll further explain all the control signals);
4. the **register file** outputs to the **ALU** the required data;
5. the ALU add the two values;
6. the values, since they don't have to be stored into the **data memory**, are instead sent back to the **register file**;
7. the **CPU** passes to the next instruction.

But what if we want to use an instruction such as `lw x5, 4(x6)`? First of all, these instructions need to either read from the registers and store into the memory (in the case of the save/store instructions) or read from the memory and store into the registers (in the case of the load instructions). Plus, we are also using an offset to reach certain addresses

(in our case, it's the 4), how do we handle it? In order to answer to the last question, we need a circuit called **immediate generator**: such circuit, given an instruction, computes an immediate that is then used inside the main ALU. We'll then need to control, via various signals, which paths the data should take:

- I. RegWrite: directly wired to the register file, if 1 then the register file accepts new data, to overwrite the previous registers;
- II. ALUSrc: controls a multiplexer that decides if the second input of the main ALU has to be an immediate or the content of a register. If 0, then it selects the content of the register, else the immediate;
- III. MemWrite: directly wired to the data memory, if 1 then it allows the CPU to write into the data memory;
- IV. MemRead: directly wired too to the data memory, if 1 then it allows the CPU to read some data from the data memory;
- V. MemtoReg: used to control a multiplexer, which controls whether the data that has to be stored inside the register must be the one from the memory or the one from the ALU. If it is 0 then it will select the result of the ALU, otherwise the one from the data memory.

What about branches then (so instructions like `beq x6, x7, offset`)? We have 3 operands: the two registers and the offset. The branch-like instructions operate directly on the PC, but how do they work? The CPU decides whether to **take a branch** or not based on the result of the subtraction of the two registers. If the result is for example 0, then the ALU's Zero signal will be 1 (all ALUs have such exit signal). Once the branch is taken, then via a secondary ALU, dedicated only to summing values for the PC, we'll compute the sum between the offset and the actual address stored into the PC. We also need one more signal: PCSrc. Such signal controls a multiplexer that decides what number we have to sum to the PC's value. [Here below](#) is shown a picture of the whole CPU with all the elements that we added so far.

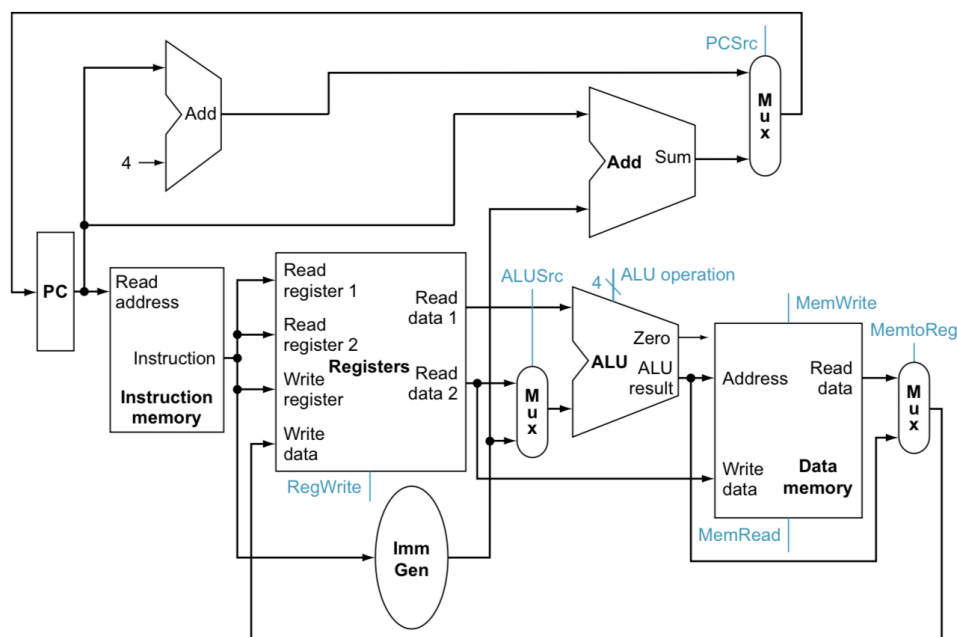


Figure 6.4: A general overview of the CPU, valid for arithmetic-logical, memory reference and branch-like instructions

6.4.3 ALU control and Main Control Unit

We previously talked about both the control unit and the various signals that we have in a CPU, but we didn't talk about one specific signal: the ALU operation (or ALUOp) signal, which is passed to a circuit designed to control the ALUs: the **ALU control**.

Depending on the type of instruction, we need to instruct the ALU on how to behave, and to determine which operation must be done, we generate a 2 bit wide flag, which is the aforementioned ALUOp. This flag can have 3 possible values: 00, 01 and 1X. Each flag determines the kind of operation that must be done, accordingly to the type of instruction that we are executing. For example, in the case of **arithmetic-logical** instructions, the ALU can do 4 operations: AND, OR, add and subtract. These operations are defined by the bits of both func3 and func7, mainly if the ALUOp is 1X (for branches, the func3 signal is used too, but not the func7, since the **T-Type** doesn't have such signal).

For load/store-like instructions, we just need to add the offset to the address. We need to set a different signal for this type of instructions, but why? Simply because the **I-Type** and **S-Type** encoding don't have a func7 slot, so we need to tell to the ALU that it has to sum the two values in another way.

If we want to use a branch-like instruction, we need to make a subtraction between the content of reg[1] and reg[2]. Depending on the result of the subtraction of the two we can decide whether we have to branch or not. For example, suppose that we have the following situation:

```
reg[1] = 0x42; reg[1] = 0x42  $\Rightarrow$  beq reg[1], reg[2], myLabel
```

The instruction says that we want to branch if the content of both reg[1] and reg[2] are equal, but if that's not the case, then we don't branch. How can we check if the two contents are equal? We can make the subtraction between the two, and if the result is zero then we know that the two results are equal, letting us branch. In the case that the instruction was

```
blt reg[1], reg[2], myLabel
```

then if the result of reg[1] - reg[2] is negative, it means that reg[1] < reg[2], and thus we can branch.

Now that we described the ALU control, we proceed on describing the most important control unit of the CPU: the **Main Control Unit** (MCU). Such unit has the task to distinguish the various type of instructions, and to send the right signals to the various parts

Instruction opcode	ALUOp	Operation	Func7 field	Func3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXX	XXX	add	0010
sw	00	store word	XXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALUOp		Func7 field							Func3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

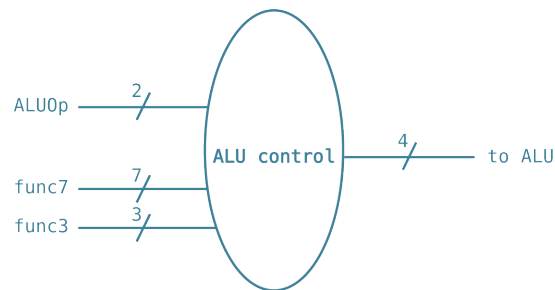


Figure 6.5: The two tables show the construction of the ALU control signals, while the circuit above shows the signals in input and output of the ALU control

the CPU do only one operation at a time, so how can we increment the productivity? This is where we can introduce the concept of **pipelining**.

In pipelining, we call each step in which the data passes through a **stage**, and all stages operate concurrently. The key point of pipelining is the efficiency: we're not reducing the time needed to perform a given task, but we are improving the **throughput** of the CPU, by making more operations in a more concentrated time window.

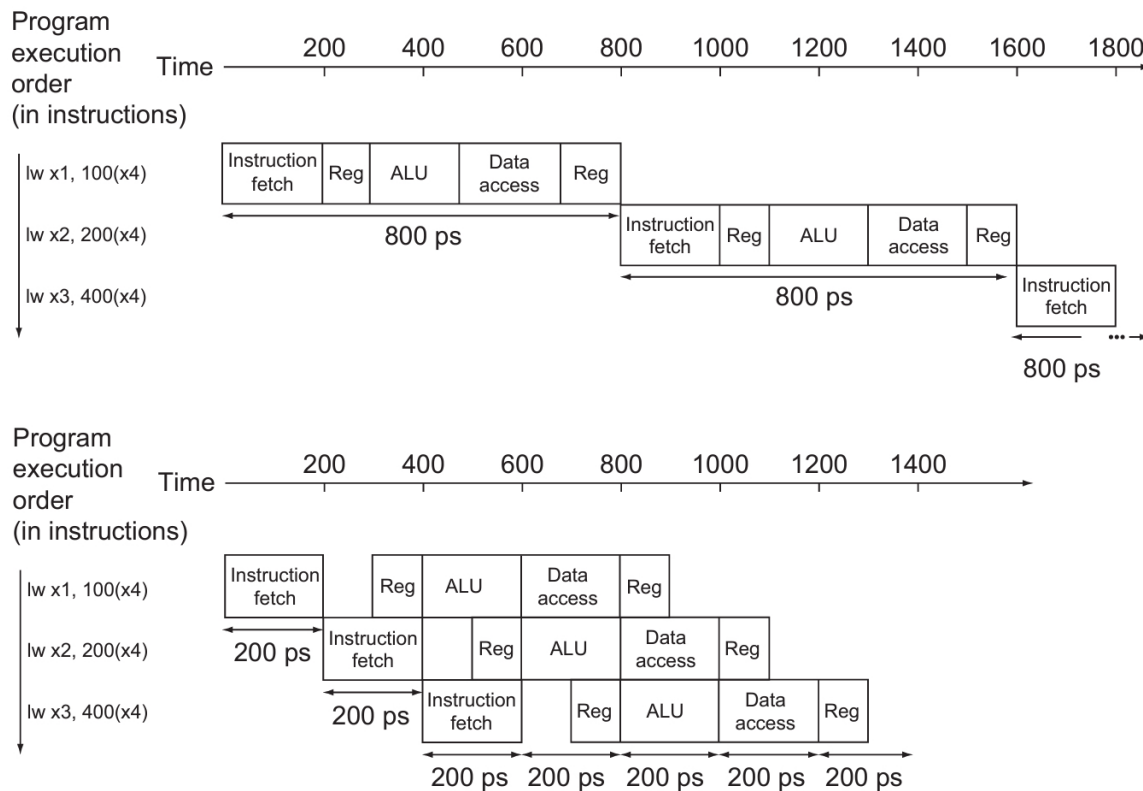


Figure 6.7: Two graphs showing the execution of 3 `lw` instructions: in the graph above, the instructions are executed without the use of pipelining; in the graph below, throughput is optimized via the use of pipelining

In RISC-V processors, there are 5 main stages:

- I. **IF (Instruction Fetch)**: the CPU fetches the next instruction from the instruction memory;
- II. **ID (Instruction Decode)**: the instruction gets decoded into its many parts;
- III. **EX (Execution)**: the operations that must be executed via the ALU get executed, such as the arithmetic operations or the calculation of a memory address;
- IV. **MEM (Data Memory Access)**: if necessary, memory is accessed, either in read or in store mode;
- V. **WB (Write Back)**: if necessary, data gets written inside the register file.

We call the time between each positive edge t , and we indicate the frequency with $\frac{1}{t}$. Now, if we have for example to do more processes, we would have to wait for each instruction to finish, and then a new instruction would begin to be executed.

6.6 Caches

In order to be performant, a computer needs a fast memory. **SRAM** (Static RAM) is the fastest type of memory (outside of flip-flops) that we can have, while **DRAM** (Dynamic RAM) is way slower than **SRAM**.

In order to use at the best of its capability the memory, we don't want to access it randomly. Let us take the following example:

```
1  function somma (x: array of integers, n: integer) {  
2      int s = 0  
3      for (i = 0, to n - 1) {  
4          s = s + x[i]  
5      }  
6      return s  
7  }
```

Let us examine where all these variables are stored:

- `x` is stored in the memory, we are passing to the function just the address to the array. Say that we store the address in `x10`;
- `n` is stored in a register, that could be for instance `x11`;
- `s` could be stored into the stack, and then retrieved at the end of the function;
- `i`, similarly to `n`, is stored into a register.

We can talk about **locality**: following this principle, we state that programs access a relatively small portion of their address space at any instant of time. There are two types of locality:

- **temporal locality** is that type of locality that states that, if an item gets referenced at any time, then there will be a tendency to be referenced again in the near future;
- **spatial locality** is the type of locality for which if a data location is referenced, then data locations with nearby addresses will tend to be referenced soon.

What do we mean with **cache** then? We refer to a small and very fast memory, placed in between the CPU and the memory. So for instance, when a value is used from the CPU, then it gets stored first in the cache, and then it gets stored into the cache. When the values are not used anymore, they get deleted from the cache, but they don't get lost, since they get always saved into the main memory. This aforementioned implementation allows us to use the concept of temporal locality. Actually, in the cache we don't store only the values that the CPU frequently used, but also some of the nearby values in the memory (for example, if we have a word in the address `0x1001020`, then the cache loads the values from `0x1001000` to `0x1001040`).

Suppose that we have the two following pieces of code:

```

for (i = 0 to n-1) {
    for (j = 0 to m-1) {
        s = s + a[i, j]
    }
}

for (j = 0 to m-1) {
    for (i = 0 to n-1) {
        s = s + a[i, j]
    }
}

```

In the first for loop, we have a more efficient use of the cache, since a matrix is stored in the memory row by row; the second for loop doesn't implement efficiently the cache, since it accesses the matrix column by column. The result is the same, but the runtime will be slightly different.

We previously mentioned that the cache stores within itself a block of addresses, but how many addresses? Suppose that each 64 bytes we have a block: we would have the blocks go like this: block 0 includes the addresses [0:60], block 1 includes the addresses [64:124], block 2 includes the addresses [128:188], and so on and so forth...

In order to determine the block where a certain value is stored, we compute the following:

$$\left\lfloor \frac{x}{64} \right\rfloor \quad \text{where } x \text{ is the address where a value is stored}$$

A more general formula could be the following:

$$\text{block} = \left\lfloor \frac{\text{address in the memory}}{\text{size of a block}} \right\rfloor$$

In general, when we access memory, we have the following data structure that defines the address:

0:5 an offset to add to the actual block;

6:31 the block's address.

Normally, programs write in the cache for the 13% of their execution, while for the remaining 87% they read from the memory. When you have to replace a block, you actuate a process called **write-back**: during this process, the block that you want to replace has to be saved in the memory; although the read speed is fast, the write speed is usually slower.

Since writes are less frequent than the reads, when we store something into the cache, we store it directly into the main memory. This process is called **write-through**.

Cache also works with blocks, but unlike memory's blocks, they are called **sets**. There are n sets in the cache, where n is a power of 2. What happens when we have to write a block into a set? We can store a block one by the other: if a cache is made of 16 sets, then we'll have set 0, set 1, set 2, etc... up until set 15. When we have to store a block into a set, we begin with set 0, then we go to the next one, and so on. When we reach set 15 we start over with set 0. The block where we have to store is determined by the remainder of the following operation (where n is the total number of sets into the cache):

$$\text{set where we store} = \text{block number} \% n$$

Because of this introduction, we need to change the format of the encoding of the data in the cache: we would need

- some bits for the block's address;
- a tag for defining the sets' number. When we want to read/write the cache, if we find in the same set a block with an equal tag, then we say that we have an hit, otherwise we have a miss;
- a flag bit to define if the block is valid or not. We use this block because this way we can detect if some garbage fell into the cache;
- another flag bit to determine whether the block is *dirty* or not. This bit is useful because if we edit the memory, then the data in the cache are not valid anymore, and we would need to copy back the data from the memory. Because of this, we flag this behaviour with a control bit.

Let us examine the following example in RISC-V:

```

1  .data
2  x:  .word 3, 56, ...
3  n:  .word 1024
4
5  .text
6      la x8, 0x10010
7      lw x9, n
8      li x10, 0
9
10     ciclo: lw x5, 0(x8)
11         add x10, x10, x5
12         addi x5, x5, 4
13         addi x9, x9, -1
14         bne x9, x0, ciclo

```

Block's number	Tag	Set number
1	000	0001
17	001	0001
33	010	0001
49	011	0001
65	100	0001

Figure 6.8: Example of addresses formatting, where the addresses are all going into set 1. This shows an alternative way to find tag and set number: by writing the block number in binary, the 4 LSB are the set, while the others are the tag

In this program, we are computing the sum of all the 1024 items in the array.

CONTINUE

In computers, we don't have just one single cache: the quantity of data that the computer handles for each complex operation is much larger than the size of a single cache. In computers, we can normally see multiple caches, passing from what we call **one-way associative cache** to the **two-ways associative cache** (and also in much more larger sizes, but for the moment we'll stick to the two-ways associative cache). In such caches, we have two one-way caches and a signal bit that changes depending on the last used slot. For instance, suppose that we have a cache of 4 blocks, and in a given set n we store a given value. The first value will be stored in one of the two caches. If we want to store another value in the same position, we'll check the signal bit: if 1, then the CPU will proceed to save the value in the other cache, on the same address, while if the signal bit is 0, then it means that the last used slot is the other one, meaning that we can write on the first cache. This way of storing inside the cache is called **Least Recently Used** policy. The trade-off for multiple ways associative caches is that the size of the cache remains the same: for instance, a single-way associative cache with the block size of 32, if transformed to two-ways, it becomes a group of two single-way associative caches with 16 sets. The same would happen if we did a four-way associative cache, having blocks of 8 sets.

6.7 Virtual Memory

When we run multiple processes in a computer, we may come to a point where we have to access simultaneously to the cache. This obviously is a problem, since data could be wrongly exchanged, or maybe even corrupted. This is where **virtual memory** comes in help.

This technology was invented in the '70s, when memory weren't as large as in today's computers (now it's normal to have 8GiB or 16GiB of memory, but at the time computers had around 1KiB of memory). The core concept of virtual memory is to have a **non-physical memory**, with addresses that are not the ones of the physical memory. In the virtual memory, we have multiple groups of bytes, of the size of 4KiB, that are called **pages**. The virtual pages can be either stored into the physical memory or into the hardisk.

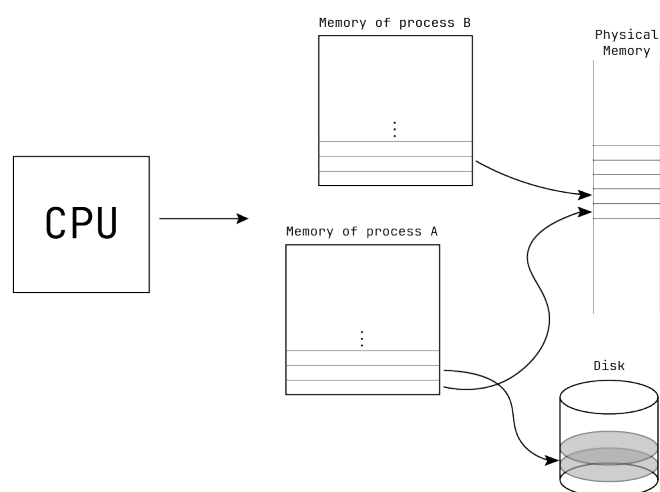
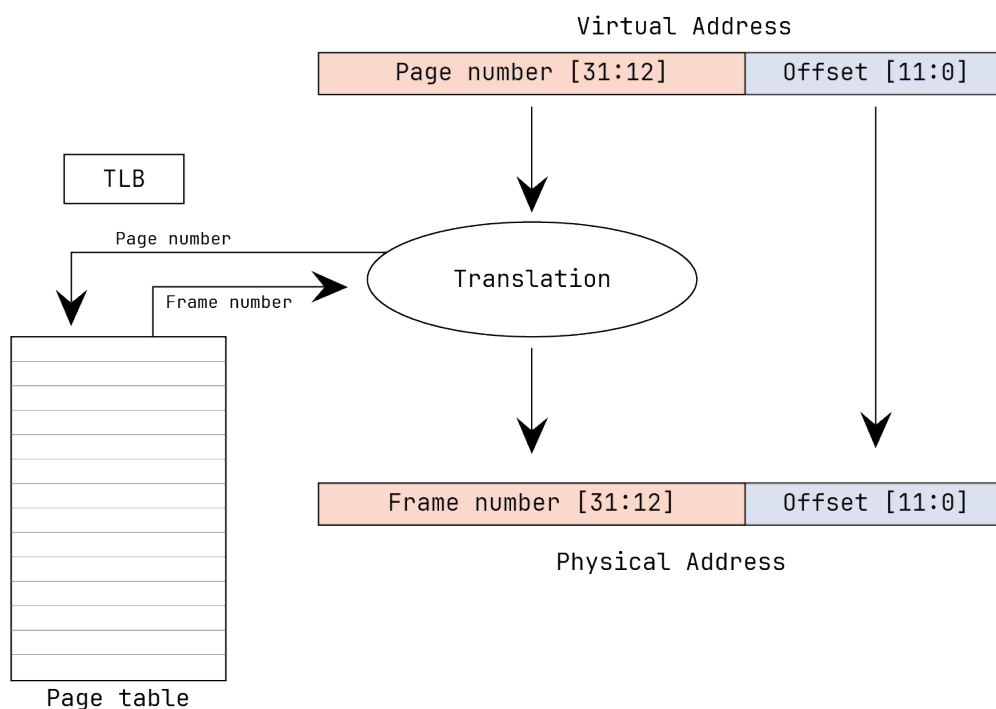


Figure 6.9: Visual representation of two processes A and B using Virtual Memory

Back in the days, virtual memory was used to put larger programs in a smaller memory, now it's used for running multiple programs at once. A property of Virtual Memory is that the memory of process A doesn't have any access to the memory of process B.

This way, if a program crashes, then its crash doesn't affect the other programs.

By itself, the CPU doesn't know the addresses in the memory of the virtual memories. The address itself is given by the following notation: we need 12 bits in the form of an offset, but also 20 bits for the page number. How does the CPU access to the right page in the virtual memory? We have the number of the page, but we don't know where it is. There is a little circuit that translates the address with the page number to the address in the physical memory (which could be either in the RAM or in the hardisk. Such circuit uses a part of the memory called **page table** where all the pages are stored. But this way we would take too much time, since in order to access a specific address we need to go through the memory two times. How can we fix it? There is a small cache, called **Translation Look-aside Buffer (TLB)** that remembers the last used pages, such that if the CPU has to access a page that was recently accessed, then it can retrieve it directly from the cache.



6.8 Parallelism

Essentially, when we talk about **parallelism**, we mean executing multiple processes at the same time. An example in software could be multi-threading, while on the hardware side we could talk about CPU cores. We can also talk about **instruction-level parallelism**, so the execution of multiple instructions all at once (also referred to as **Static Multiple Issue**). There are multiple examples of it, but we'll see a specific example with the RISC-V processor, where we try to execute 2 instructions together.

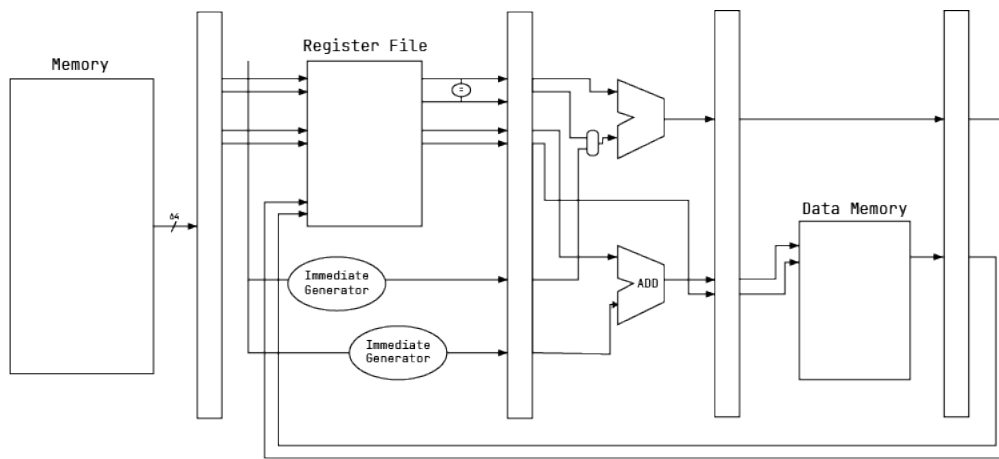


Figure 6.10: Example of a RISC-V pipeline executing two instructions at once

The image on the top ([Figure 6.10](#)) shows an example of a RISC-V pipeline with 2 instructions executed at the same time.

[...]

We can improve our loop in this way: normally, our code would be something like the following

```
for i in range(n-1):
```

We can, though, try to do a **Loop Unroll** and separate the for loop