POLITECNICO DI TORINO

COMPUTER ARCHITECTURE – Prof. Montuschi Paolo

Lecture notes *by Filippo Bracco*

Academic Year 2019/2020

0.1 – COURSE PRACTICAL INFORMATION

Prof. Montuschi Paolo
DAUIN -Dipartimento di Automatica e Informatica
Tel: 011090 7014 / 011090 6629
E-mail: paolo.montuschi@polito.it
Site: staff.polito.it/paolo.montuschi/

Course topics

- Basics, examples and exercises of simple combinational and sequential circuit design and related issues, such as testing of the correct behaviour, memory/area/speed trade-off, energy consumption, delay and critical path;
- The components of a microprocessor-based system and their interactions: CPU, cache memory, main memory, secondary memory, peripherals, Input / Output devices and related addressing and communication issues, buses and addressing modes;
- Some "milestones" of Computer Engineering: virtualization, parallelization of operations, operating systems, reduced instruction set computers, configurable devices.
- An introduction to the Assembly language.

Course structure

- Class lectures: about 62% of the course duration;
- Class exercise time: about 20% of the course duration;
- Assisted laboratories: about 18% of the course duration.

Assessment and grading criteria

Exam: written test; compulsory oral exam;

The written exam lasts about 45 minutes. The written exam (accounting up to 24 points) is composed of:

- 4 closed-ended questions (correct +2, wrong −1, missing −0.5)
- 4 open-ended questions (up to 4 points each)

on the whole program of the course. An oral exam (accounting up to 9 more points) follows the written exam. The oral and the written parts cover the whole program of the course. In order to have access to the oral exam, it is necessary a minimum score of 12 points for the written part. Otherwise a rejection will be recorded.

If the result of the written part is larger than or equal to 18, then the student can request the registration of the final grade of the exam. In all other cases, the student will (can) have an oral exam, consisting of at most three additional questions adding up to 9 more points.
The three questions will span the full program and can also involve the discussion of the laboratory exercises. Failure to satisfactorily responding a question, will imply a negative score for that question and the immediate termination of the oral exam.
If less than 18 points are obtained, a rejection will be registered.

Professor has the right to ask at any time oral questions to get a better and more complete picture of the student's preparation.

The final grade will be determined by adding up all the points collected by the student and rounding the numerical result. Laude will be granted to all students whose number of points exceeds or is equal to 31.5.

Overall, the exam is targeted at evaluating the students both about their knowledge of basic computing systems architectures and their design.

Several problems proposed as previous written parts will be made available to the students through the web page of the course.

Material + Links

- V.C. Hamacher et al., Computer Organization, McGraw-Hill, 2005
- M. Morris Mano, C. R. Kime, Logic and Computer Design Fundamentals, 4th edition, Pearson Prentice-Hall, 2008
- Stallings W., Computer Organization and Architecture, 2016

Course Material collected by Student Luigi Zevola in 2013 - sites.google.com/site/fulladder2013/

Xilinx ISE Simulation 14.7:
www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html
xilinx.com→download area→Xilinx ISE 14.7, no vivado - 7GB . Note for Windows 10: supports only Pro 64-bit version

Drawing logic circuits: logic.ly

Author's note:

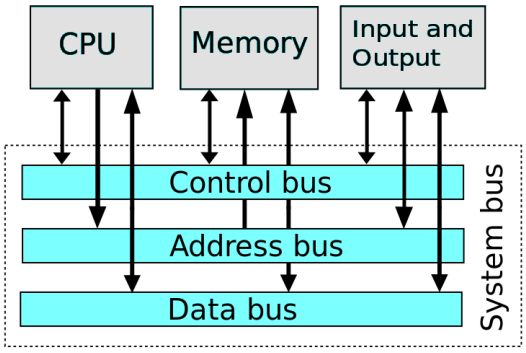
I wrote these notes during the 2020 pandemic: the objective of this work was to help everyone who needed, *distant but together*.

I hope you find here what you are looking for, but keep in mind that the lectures are fundamental for understanding this course.

Trust the notes, but at your own risk: as Professor Montuschi says, *it depends*!

Filippo Bracco
Site: firustuff.blogspot.com/p/polito-notes.html

1 – VON NEUMANN ARCHITECTURE



This idea, known as the ***stored-program concept***, is usually attributed to the ENIAC designers, most notably the mathematician John ***von Neumann***, who was a consultant on the ENIAC project. Alan Turing developed the idea at about the same time. The first publication of the idea was in a 1945 proposal by Von Neumann for a new computer, the EDVAC (Electronic Discrete Variable Computer).
In 1946, Von Neumann and his colleagues began the design of a new stored-program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers.

The Von Neumann architecture scheme is based on four fundamental components:

1. ***CPU*** (Central Processing Unit) which contains
    1. *ALU* (Arithmetic and Logic Unit), which performs logic and arithmetic operations
    2. *Program Counter* (PC), which holds the next instruction to be fetched and executed
    3. *Instruction Register* (IR), which holds the current instruction to be fetched and executed, taken from PC before it is updated in order to point out to next instruction
    4. *Control Decoding Unit* (CDU), which decodes instructions from Instruction Register and generates proper control signals
    5. *General Purpose Registers*, used to temporary store any kind of information
2. ***Memory Unit***, intended as main memory (usually RAM, Random Access Memory)
3. ***Input / Output Unit***, which enables the user to enter any kind of digital information and provides results/information to the user
4. ***Bus***, a channel so that any device can communicate with each other

Assembly language is a low-level programming language which uses symbols, lack variables, functions and which work directly with CPU; it is coded differently for every type of processor.

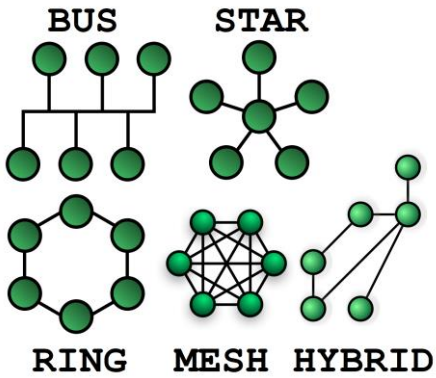
Managing the hardware can make things faster and even more secure, but the hardware cannot be updated.

An ***embedded system*** is a controller with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts.
A combination of computer hardware and software either fixed in capability or programmable, that is specifically designed for a particular function.
Embedded systems that are programmable are provided with programming interfaces, and embedded systems programming is a specialized occupation.

Computers collaborate to achieve tasks. The selection of a computer architecture depends on money, time, constraints, purpose etc.

Architecture topologies describe the arrangement of the elements composing the computer itself:

- ***Bus***, all the nodes are connected in series by a single central bus;
- ***Star***, all the elements refer to a single one which is independent from the rest;
- ***Ring or circular***, everything is connected directly and sequentially in a closed loop, with data travelling around the ring in one direction only;
- ***Mesh***, any element is connected to all of the others;
- ***Partially mesh*** or ***hybrid***, combines two or more topologies in such a way that the resulting network does not exhibit one of the standard topologies; it is used to optimise the results and reducing the disadvantages of the others structures.



Disadvantages for each topology:

- ***Bus***: there needs to be a bus nearby; if the bus is down all the components are down, furthermore if there is just one bus connecting everything it would cause priority issues (can be overloaded with data);
- ***Star***: any failure of the main component affects all the others;
- ***Ring***: it is cheap and limited in length, but if one component is broken there may be no way to communicate with the others;
- ***Mesh***: it is the fastest but more expensive topology (n elements would require n^2 connections).

Architecture topology examples:

- Peer-to-peer connection has a star structure;
- Skype has a hybrid structure where bandwidth and network are shared.

2 – LOGIC GATES

The basic operations of Boolean algebra are as follows:

Operator	*Operation name*	*Notation*			
AND	Conjunction	$x \wedge y$	$x \times y$	$x \cdot y$	xy
OR	Disjunction	$x \vee y$		$x+y$	
NOT	Negation	$\neg x$	$\bar{x}$		x'

Combining these operations, we can obtain others:

Operator	*Operation name*	*Notation*
XOR	Exclusive OR	$x \oplus y$
NAND	Not AND	$\overline{AB}$
NOR	Not OR	$\overline{A+B}$
XNOR / EQ	Material equivalence	$x \odot y$
IMPLY	Material implication	$x \rightarrow y$

Logic operators and their rules:

AND: $xy \rightarrow$ output equal to 1 only if both x and y are equal to 1; equal to 0 otherwise.

OR: $x+y \rightarrow$ output equal to 1 whenever at least one operand is equal to 1; equal to 0 when $x = y = 0$.

NOT: $x' \rightarrow$ output equal to 1 if $x = 0$, and vice versa.

XOR: $x \oplus y \rightarrow$ output equal to 1 whenever x and y have different values; equal to 0 otherwise.

XNOR: $(x \oplus y)' \rightarrow$ output equal to 1 whenever x and y have equal values; equal to 0 otherwise. (it's XOR complement function)

NOR: $(x+y)' \rightarrow$ output equal to 1 whenever x and y are both equal to 0; equal to 0 otherwise. (it's OR complement function)

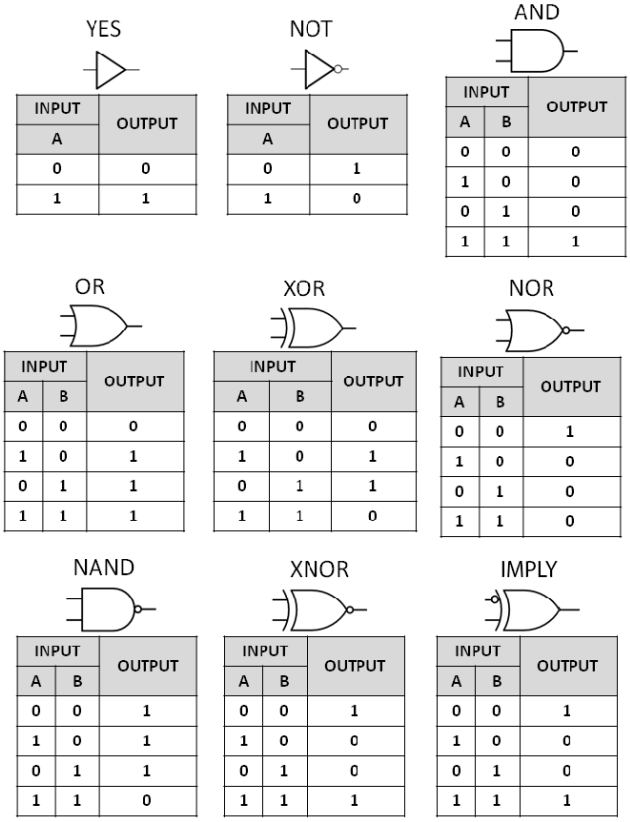
NAND: $(xy)' \rightarrow$ output equal to 0 whenever x and y are both equal to 1; equal to 1 otherwise. (it's AND complement function)

Logic gate	*Operation syntax*
NOT / INV	`Y = !A`
AND	`Y = A × B = A·B = AB`
OR	`Y = A + B`
NAND	`Y = (A × B)'`
NOR	`Y = (A + B)'`
XOR	`Y = A·B' + A'·B`
XNOR	`Y = A·B + A'·B'`
IMPLY	`Y = (A·B')' = A' + B`

A ***logic gate*** is a device implementing a Boolean function: it performs a logical operation on one or more binary inputs and produces a single binary output. By convention, FALSE = 0 and TRUE = 1.

Logic gates ***cannot be reversed***: firstly, because they are electronic devices and as such they cannot be reversed. Supposing to have a technology capable of reversing the signal from output to input, what would be the new output values? We cannot know logically, and therefore gates cannot be reversed.

Truth tables of logic gates:



YES

INPUT A	OUTPUT
0	0
1	1

NOT

INPUT A	OUTPUT
0	1
1	0

AND

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	0
0	1	0
1	1	1

OR

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	1

XOR

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	0

NOR

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	0

NAND

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	1
0	1	1
1	1	0

XNOR

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	1

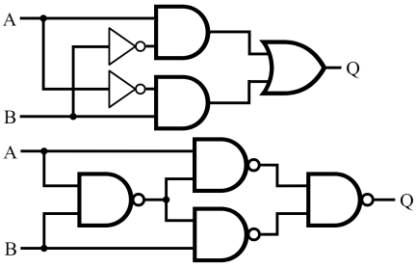
IMPLY

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	1

Complete Sets are sets of logic gates that enable the user to implement any other Boolean function:

- AND, OR, NOT
- AND, NOT
- AND, OR → not complete! it's not possible to realize the NOT function
- NAND
- NOR
- XOR
- XNOR
- OR, NOT

There are transformation rules so that a circuit can be made all of the same gates: for example, a 2-input XOR gate can be made using 4 NAND gates.



What is the difference between these two circuits? There is no difference, but it is much more preferred to implement a circuit whose gates are all of the same types, so the second one: all gates have the same function and characteristics.

3 – BOOLEAN FUNCTIONS

They are based on 2-values Boolean algebra (0,1 values only) and they let us design a variety of digital circuits. A ***combinational circuit*** is an interconnected set of gates whose output at any time is a function only of the input at that time.
A combinational circuit consists of n binary inputs and m binary outputs. As with a gate, a combinational circuit can be defined in three ways:

- ***Truth table***: For each of the 2^n possible combinations of input signals, the binary value of each of the m output signals is listed.
- ***Graphical symbols***: The interconnected layout of gates is depicted.
- ***Boolean equations***: Each output signal is expressed as a Boolean function of its input signals.

Any Boolean function can be expressed in two ***canonical*** or ***standard*** forms:
- ***SOP***, ***Sum of Products***, where each product contains n *literals* (input letters) and is called *minterm*, for a total of 2^n different minterms, each composed of n literals.
- ***POS***, ***Product of Sums***, where each sum contains n literals (literal = input letter) and is called a *maxterm*, for a total of 2^n different minterms, each composed of n literals.

Minterm means the term that is true for a minimum number of combination of inputs. That is true for only one combination of inputs.
Maxterm means the term or expression that is true for a maximum number of input combinations or that is false for only one combination of inputs.

Minterm example: `101` → $x\bar{y}z$
Maxterm example: `101` → $\bar{x} + y + \bar{z}$

<u>*General conversion procedure*</u>
To convert from one canonical form to another, interchange the symbols ***summation*** $\sum$ and ***multiplication*** $\mathbf{\Pi}$ and list those numbers missing from the original form. In order to find the missing terms, one must realize that the total number of minterms or maxterms is 2^n, where n is the number of binary variables in the function.
We use SOP form to describe input combinations whose associated output is equal to 1, and POS for those whose output is equal to 0.

What to do if we want to express a Boolean function in XXX form and it is currently expressed in YYY form?
- Write numbers NOT appearing in POS (maxterms) [or SOP (minterms)] form in SOP [or POS] form.

● If we want to obtain the complement of the actual function currently expressed in POS form, we have to:
- Use the numbers appearing in POS form and write them in SOP form.
- Use the numbers NOT appearing in POS form and write them in POS form.

● If we want to obtain the complement of the actual function currently expressed in SOP form we have to:
- Use the numbers appearing in SOP form and write them in POS form.
- Use the numbers NOT appearing in SOP form and write them in SOP form.

The POS-SOP conversion is also called ***principle of duality***: given any logic expression, its dual is formed by replacing all + with ·, and vice versa, replacing all 0s with 1s and vice versa.

Boolean Algebra Axioms:

1a	`0·0=0`	1b	`1+1=1`
2a	`1·1=1`	2b	`0+0=0`
3a	`0·1=1·0=0`	3b	`1+0=0+1=1`
4a	If `x=0` then `x'=1`		
4b	If `x=1` then `x'=0`		

Boolean Algebra Single Variable Theorems:

5a	`x·0=0`	5b	`x+1=1`
6a	`x·1=x`	6b	`x+0=x`
7a	`x·x=x`	7b	`x+x=x`
8a	`x·x'=0`	8b	`x+x'=1`
9	`(x')'=x`		

Boolean Algebra Two or Three Variables Properties:

10a	`x·y=y·x`	*Commutative*
10b	`x+y=y+x`	
11a	`x·(y·z)=(x·y)·z`	*Associative*
11b	`x+(y+z)=(x+y)+z`	
12a	`x·(y+z)=x·y+x·z`	*Distributive*
12b	`x+y·z=(x+y)·(x+z)`	
13a	`x+x·y=x`	*Absorption*
13b	`x·(x+y)=x`	
14a	`x·y+x·y'=x`	*Combining*
14b	`(x+y)·(x+y')=x`	
15a	`(x·y)'=x'+y'`	*DeMorgan's Theorem*
15b	`(x+y)'=x'·y'`	
16a	`x+x'·y=x+y`	
16b	`x·(x'+y)=x·y`	

With a simpler Boolean expression, fewer gates will be needed to implement the function. Three methods that can be used to achieve simplification are:

- Algebraic simplification
- Karnaugh maps
- Quine–McCluskey tables

Algebraic simplification involves the application of the identities above mentioned to reduce the Boolean expression to one with fewer elements.

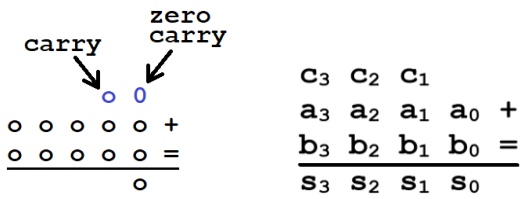
These rules can be used in order to simplify Boolean functions expressions, thus reducing the overall cost of the circuit, at least in terms of gates used.

Another method of function minimization, valid for a number of inputs between 2 and 4, relies on the use of *Karnaugh Maps* (which will be seen after the adders).

4.0 – ARITHMETIC AND ADDERS

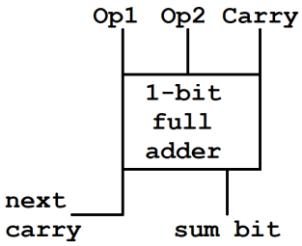
• *Adder*

Binary addition always generates a carry, that can be 0 or 1; the generated carry is placed on the left column. The first column on the right always has a zero carry; all the others depend on the operation done.



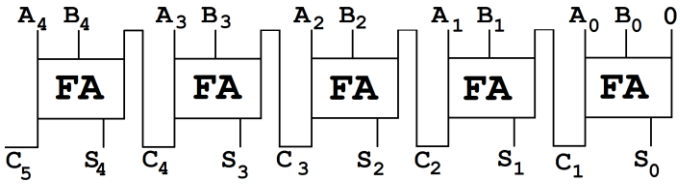
An ***adder*** is a digital circuit that performs addition of numbers.
The ***half adder*** adds two single binary digits A and B. It has two outputs, ***sum*** (S) and ***carry*** (C).

The ***full adder*** adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers (the operands A and B, the previous carry C_{in}) and produces a two-bit output (output carry C_{out} and sum S).



A ***ripple-carry adder*** (***RCA***) is a logical circuit using multiple full adders to add N-bit numbers: each full adder inputs a C_{in}, which is the C_{out} of the previous adder. Each carry bit "ripples" to the next full adder. The first (and only the first) full adder may be replaced by a half adder (under the assumption that its $C_{in} = 0$).

How to build a general 5-bit adder? I need a RCA made of five full adders.

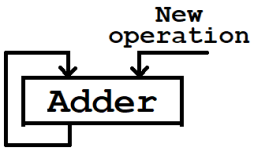


Linear property: the more the bits, the more full adders and more carry.

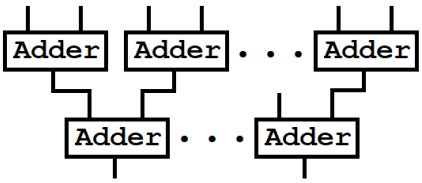
What is the delay?
Carry propagation delay of a full adder is the time taken by it to produce the output carry bit.
Sum propagation delay of a full adder is the time taken by it to produce the output sum bit.
Worst-case delay of a ripple carry adder is the time after which the output sum bit and carry bit becomes available from the last full adder.

The time of each full adder is 1. How long does it take for 5-bits and 2 numbers? $5t_{FA} = 5$.

► *Exercise – Adding 1000 100-bit numbers*
Add a thousand of numbers each one of 100 digits.
Let's start with a linear approach.
For two 100-bit operands it takes a delay of $t = 100$ per sum.
The number of sums is $N-1=999$, so the overall delay is $T \simeq 1000 \cdot t_{ADDER}$
In case of an RCA: $T \simeq 1000 \cdot 100 \cdot t_{FA} \simeq 10^5\ t_{FA}$



We can change the architecture and get a better delay by doing the ***additions in parallel***.



I get the first 2 operands and calculate the result. Instead of recycling, I take the third and the fourth operands and add them up using another RCA. Adding up using pairs, this is repeated 500 times (1000 numbers grouped 2 by 2). Then I add the RCA of 1st and 2nd operands and the RCA of 3rd and 4th operands and use the outputs as inputs on another RCA, repeating this with all the other RCA by pairs: each cycle will halve the number of operations.
With N operands, the *number of levels* corresponds to $\lceil log_2(N) \rceil = 10$ (ceiling of the logarithm base 2 of N): the delay is reduced from linear to logarithmic.
The area can be calculated roughly considering that the number of gates is decreasing as a geometric series, so the sum of all gates is equal to the double of the highest number: $A \simeq 2 \cdot 500\ A_{ADDER} = 1000\ A_{ADDER}$

Comparing the delay and the area:
- Linear → $T=N \cdot t_{ADDER}$, needing 1 adder
- Parallel → $T=\log_2(N) \cdot t_{ADDER}$, needing N adders

The parallel solution is 100 times faster, but 1000 times larger and more expensive.

A *full adder* is a multi-function circuit, having three inputs and two outputs. This was the original design of computer circuits, but it is not used anymore unless it is the only possibility to solve a problem.
The full adder receives:
- $c_{i+1} = f(a_i, b_i, c_i)$
- $s_i = f(a_i, b_i, c_i)$

Therefore, the truth table is:

	a_i	b_i	c_i	c_{i+1}	s_i
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
α	**0**	**1**	**1**	**1**	0
	1	0	0	0	1
β	**1**	**0**	**1**	**1**	0
γ	**1**	**1**	**0**	**1**	0
δ	**1**	**1**	**1**	**1**	1

Looking to c_{i+1}, we focus on the lines where output is true. Overall function f can be alternatively:

$$f = \alpha + \beta + \gamma + \delta$$

In line `1 0 0 | 0` : α, β, γ and δ are not true so the output is false.
The overall function is the addition of all the cases.

α itself is a function: $\alpha = \overline{a_i} b_i c_i$

A function is true when each single term is true.

$$\left.\begin{array}{l} a_i = 0 \rightarrow \overline{a_i} = 1 \\ b_i = 1 \\ c_i = 1 \end{array}\right\} \rightarrow \alpha = 1$$

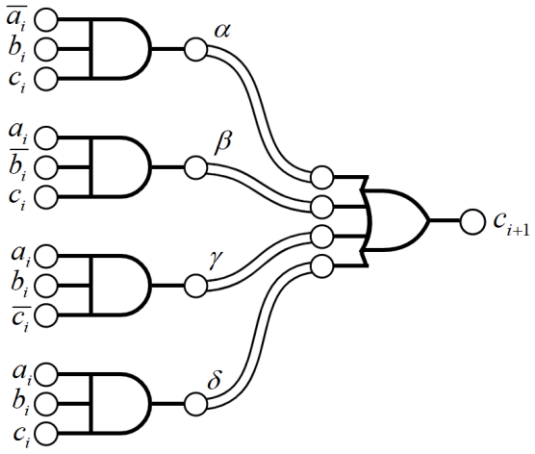
$$\beta = a_i \overline{b_i} c_i = 1$$

$$\left.\begin{array}{l} a_i = 1 \\ b_i = 0 \rightarrow \overline{b_i} = 1 \\ c_i = 1 \end{array}\right\} \rightarrow \beta = 1$$

$$\gamma = a_i b_i \overline{c_i} = 1$$

$$\left.\begin{array}{l} a_i = 1 \\ b_i = 1 \\ c_i = 0 \rightarrow \overline{c_i} = 1 \end{array}\right\} \rightarrow \gamma = 1$$

$$c_{i+1} = \underset{\alpha}{\overline{a_i} b_i c_i} + \underset{\beta}{a_i \overline{b_i} c_i} + \underset{\gamma}{a_i b_i \overline{c_i}} + \underset{\delta}{a_i b_i c_i}$$



We have the same three inputs that give an input to the four elements. We need to be careful since if the lines cross then there is a shortcut.

What is the overall cost of the ***carry circuit*** (time, power consumption, area)?
We have to consider:

- 4 AND gates (3 inputs)
- 1 OR gate (4 inputs)
- Wires

Assumptions to approximate:

- OR, AND have the same delay
- NOT gate is neglected because it's really cheap
- Wires do not count for the same reason as NOT
- Not considering the inputs, AND, OR have the same delayed area. (***this is really false!***)

We can say that the cost is of 5 gates (4 AND + 1 OR).

• ***XOR***

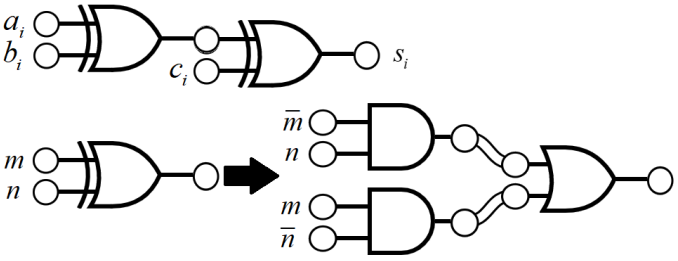
XOR is not a basic operator, but a combination of an OR with two AND, in which the first and second operator are alternatively negated: $m \oplus n = \overline{m} n + m \overline{n}$

Truth table:

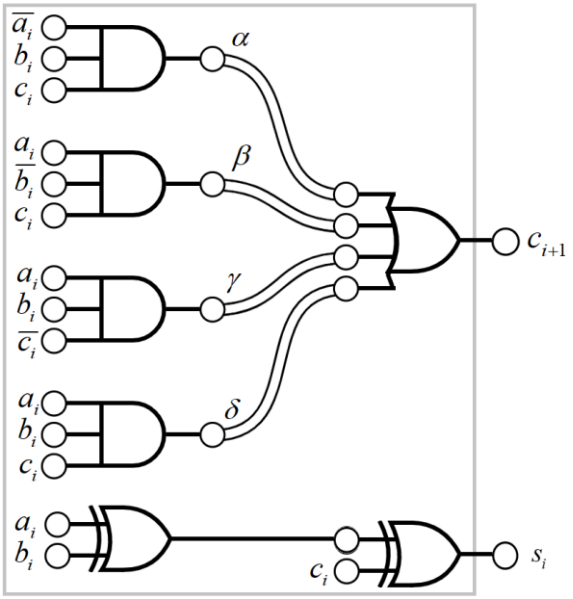
m	n	$m \oplus n$
0	0	0
0	1	1
1	0	1
1	1	0

XOR is also called *difference comparator* because it is true only when the two operands are different.
The Boolean XOR is equivalent to the ***arithmetic addition*** (for just 1 bit).

$$s_i = (a \oplus b) \oplus c = \left(\bar{a} b + a \bar{b}\right) \oplus c = a \bar{b} c + \bar{a} b \bar{c} + \bar{a} b c + a \overline{bc}$$
$$(\bar{a} b) \oplus c + (a \bar{b}) \oplus c$$



Therefore, this is the ***structure of a full adder***:



The overall time delay is $t_{FA} = max(2,4) = 4$
The area would be

$$A_{FA} = A(c_{i+1}) + A(s_i) = (4 \times 1 + 1) + (3 \times 2) =$$
$$= 5 + 6 = 11 \text{ gates}$$

Can we do better to minimise the function?

$$f = \bar{a}_i b_i c_i + a_i \bar{b}_i c_i + a_i b_i \bar{c}_i + a_i b_i c_i + a_i b_i c_i + a_i b_i c_i + a_i b_i c_i$$

I can add more $a_i b_i c_i$ because in Boolean algebra if $h=m$ and $h'=m+m+m$ then $h=h'$.

I group the terms:

$$f = \boxed{\bar{a}_i b_i c_i} + \boxed{a_i \bar{b}_i c_i} + \boxed{a_i b_i \bar{c}_i} + a_i b_i c_i + a_i b_i c_i + a_i b_i c_i$$

and obtain:

$$f = b_i c_i (\bar{a}_i + a_i) + a_i c_i (\bar{b}_i + b_i) + a_i b_i (\bar{c}_i + c_i)$$

Using the following properties:

$$k = n + \bar{n} = 1$$

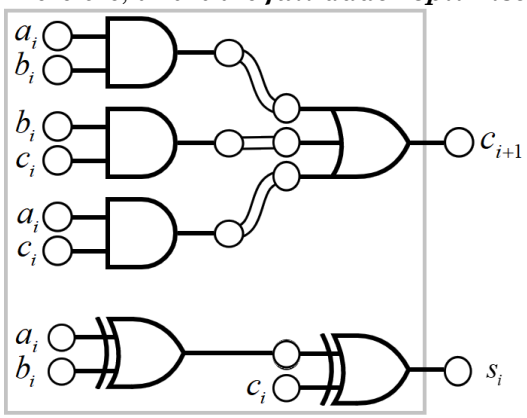
$$j = l \cdot 1 = l$$

We obtain that:

$$f = \underset{\alpha'}{b_i c_i} + \underset{\beta'}{a_i c_i} + \underset{\gamma'}{a_i b_i}$$

The function has been simplified and we can see that we do not need any NOT.

Therefore, this is the ***full adder optimised circuit***.



The delayed area of a full adder is $\mathbf{A_{FA} = 10\ gates}$.
The time delay is $t_{FA} = max(2,4) = 4$ time units.

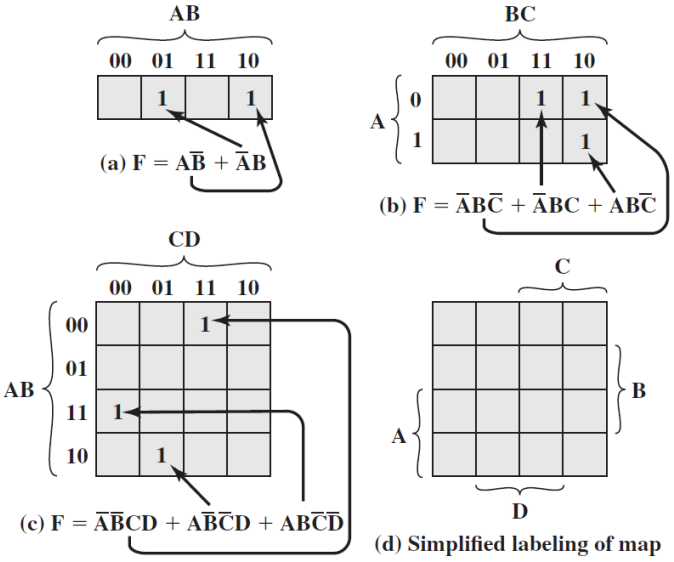
Building a circuit, if the needed blocks are more than the physic space, we need to learn to recombine blocks in order to stay in the physic conditions.

• ***Karnaugh map (map structures)***

The Karnaugh map (KM or K-map) is a method of simplifying Boolean algebra expressions of a small number (up to four) of variables, compacting their representation and therefore optimizing the circuit. The map is an array of 2^n squares, representing all possible combinations of values of n binary variables. It is essential for later purposes to list the combinations in the order 00, 01, 11, 10 (*can only change 1 bit each time*).

To convert a Boolean expression to a map, it is necessary to put the expression into its canonical form: each term in the expression must contain each variable.

Example: two, three and four variables arrangements



For a 4 input boolean function, the associated Kmap is

AB \ CD	00	01	11	10
00	0 1	1 x	2 0	3 1
01	4 x	5 1	6 0	7 1
11	8 0	9 0	10 x	11 1
10	12 1	13 0	14 0	15 X

f (a,b,c,d)
m0 = a'b'c'd' (minterm)
M0 = a+b+c+d (maxterm)

Each minterm/maxterm is equal to 1 or 0, and we substitute the proper value by inspecting the truth table or the boolean function expressed in SOP/POS form. For each group of 1,2,4,8,16 (***only powers of 2!***) of adjacent terms equal to 1, we can circle them and express the resulting minterm, eliminating each literal that changes its value (1 to 0 or 0 to 1) when passing from a square to an adjacent one inside the circle. Alternatively, we can proceed in the same way by circling adjacent terms equal to 0, and using the corresponding maxterms to write the function in POS form.

Note that we can consider '*adjacent*' any couple of squares whose inputs are equal, except for one of them; so we can consider adjacent, for example, m4 and m6 in both 3 and 4 variables kmap, and m8 and m0 in both 3 and 4 variables kmap.

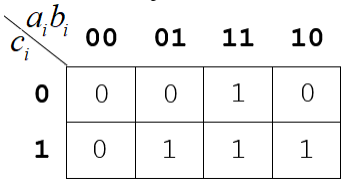
Summary:
- If the function is expressed in POS form, it is preferable to express the resulting minimized function expression through a combination of maxterms, circling terms equal to 0.
- Otherwise, when expressed in SOP, it is preferable to look for terms equal to 1, circling them, then express the resulting minimized function through a combination of minterms.

A product term is an *implicant* of a function if the function has the value 1 for all minterms of the product term. Any '1' or group of '1's that can be combined on a Kmap represents an ***implicant*** of a function.
An implicant is a ***prime implicant*** if it cannot be combined with another implicant to remove a variable.
Essential *Prime Implicant*: needed to form a minimum solution (if a minterm of a function is included in only one prime implicant, that prime implicant is said to be essential.)
Nonessential *Prime Implicant*: not necessarily needed to form a minimum solution, since it covers '1's already covered by other Essential Prime Implicants

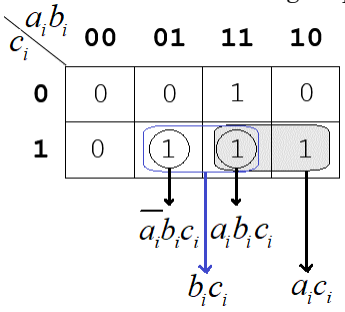
A collection of implicants that account of all valuations for which a given function is '1' is called a ***cover*** of that function. The ***cost*** is the number of gates plus the total number of inputs to all gates in the circuit.

To *minimise the costs,* we must seek for the ***largest groups possible***, *without complete overlapping*: this will give us less gates and less inputs.

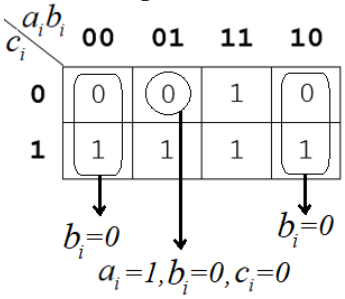
►*Example* - Considering the full adder's truth table for the ***carry*** c_{i+1} with a_i , b_i , c_i:



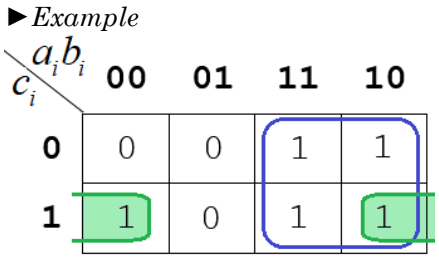
We look at the "1" and group them to get single terms:



Considering α' = b_ic_i

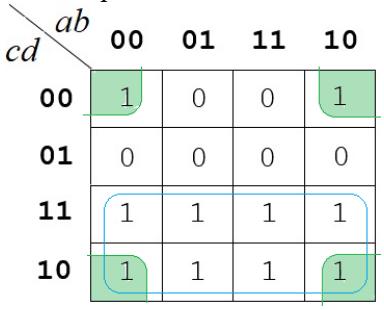


►*Example*



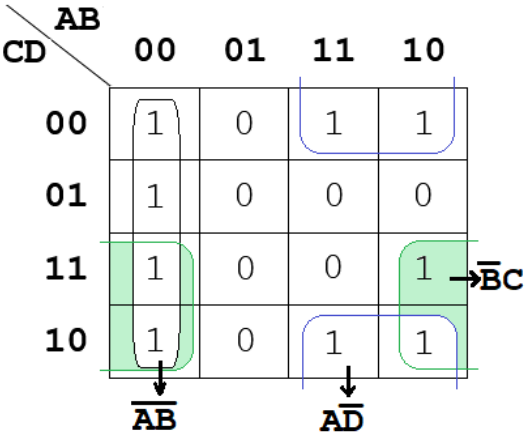
The optimised function is $f = a_i + \overline{b}_i c_i$
The following instead is not optimised, because obtained by choosing the smaller group as only one member $f = a_i + \overline{a_i b_i} c_i$

►*Example*



Because we can consider a ***kmap as developed on a sphere***, we can select the two groups in this way. The larger is equal to ***c***; the other one is the combination of the ***four corners*** so **not(*bd*)**. $f = c + \overline{bd}$

►*Exercise 1*



Considering the SOP in the kmap, the function would be $f = \overline{AB} + \overline{AD} + \overline{BC}$

► *Exercise 2*

C \ AB	00	01	11	10
0	1	1	1	1
1	1	0	1	1

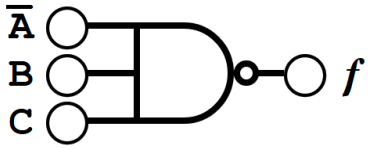
To simplify the kmap, I inverse all the values:

C \ AB	00	01	11	10
0	0	0	0	0
1	0	1	0	0

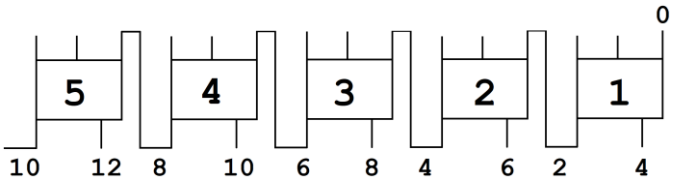
And from this I can say that $h = \overline{ABC}$, therefore (recalling that I am using negative values)

$f = h' = \overline{\overline{ABC}}$

Its logic gate representation is



Returning on the first problem, regarding the calculus of ***what is the delay*** of a 5-bit addition in an RCA.
The delay of a full adder is $t_{FA} = max(2,4)$
All the inputs have a variable at time zero.



As soon as the carry comes, the inputs get in.
The numbers written at the bottom are the times at which every term is ready to be read.
We were expecting an overall delay of $4 \times 5 = 20$, instead it is equal to 12.

The rippling of the carry is the ***critical path***, which causes the most delay to the circuit. So, *the* ***delay of the circuit*** *is the maximum delay present in such circuit*, associated to its critical path.

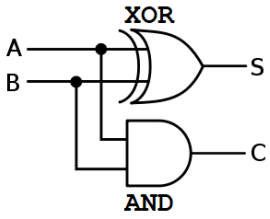
Can we do better? Yes, but if and only if we have a detailed knowledge of the internal layout of the full adder.

4.1 – COMBINATIONAL CIRCUITS

Combinational Circuits are circuits composed of logic gates with *no feedback paths*. Any output of any gate should *not feed a previous gate* along the same path, and ***the final outputs depend only on the current input values***.
Combinational circuits components are:
- *combinational logic gates* (AND, OR, NOT, XOR, …)
- the gates are connected together by *interconnections*, *wires* (direct connection between two points), *links*
- *pin in/out* (permits the connection of devices with the external world)

Most Common Combinational Circuits

• *Half Adder*



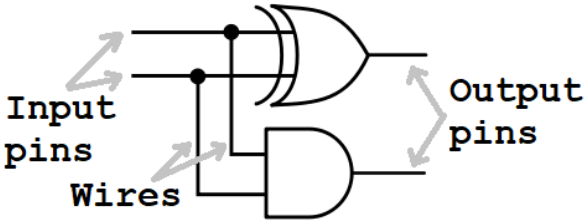
The half adder (or *1-bit adder*) is a circuit that can sum up two bits (A, B), considering NO input carry, and generates both carry (C) on next couple of bits and the correct sum (S).
The carry signal represents an overflow into the next digit of a multi-digit addition.

```
S = A'B + AB'
C = AB
```

Truth table for the half adder. →

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Here are the components of such circuit:



• *Full Adder*

The full adder is a circuit that can sum up two bits (A, B), considering an input carry (C_i), and generates both carry (C_{i+1}) on next couple of bits and the correct sum (S).

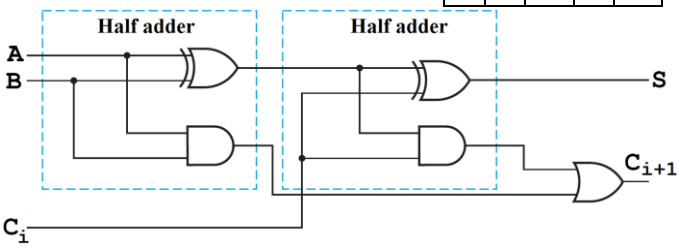
$S = A \oplus B \oplus C_i$

$C_{i+1} = AB + (A \oplus B) C_i$

Truth table for the full adder. →

A	B	C_i	S	C_{i+1}
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

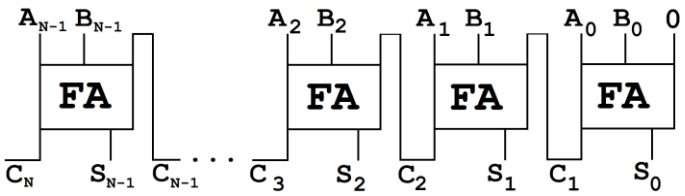
The full adder is built up through the usage of two half adders and an OR gate.



• *Ripple Carry Adder*

The ripple carry adder is a logical circuit using multiple full adders to add N-bit numbers; it is the result of a connection in cascade (in series) of N full adders, so that the final carry "ripples" through them all. A ripple carry adder can be seen as an N-bit full adder.

► *Example: an N-bit RCA*



The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals.

The total propagation time is equal to the ***propagation delay*** of a typical gate, times the number of gate levels in the circuit. The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. In this regard, consider output S_{n-1} … Y_{n-1} and X_{n-1} are available as soon as input signals are applied to the adder.
However, input carry C_{n-1} does not settle to its final value until C_3 is available from the previous stage. Similarly, C_3 has to wait for C_2 and so on down to C_0.

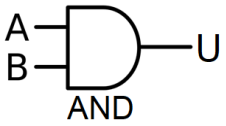
Thus, only after the carry propagates and ripples through all stages will the last output S_{n-1} and carry C_n settle to their final correct value.

The number of gate levels for the carry propagation can be found from the circuit of the full adder. The signal from the input carry C_i to the output carry C_{i+1} propagates through an *AND* gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry C_4 would have *2 * 4 = 8* gate levels from C_0 to C_4. For an *n*-bit adder, there are 2^n gate levels for the carry to propagate from input to output.
This performance drop can be partially solved using what is called a *Carry-Lookahead Adder*.

• *Design methods*

(Combinational Logic Design)
We start from the definition of the truth table, which is already a solution of our circuit. It identifies the combination of inputs and outputs.

► *Example:* AND



In the *truth table*, we list on the left all the possible combinations, on the right all the outputs of any possible combination.

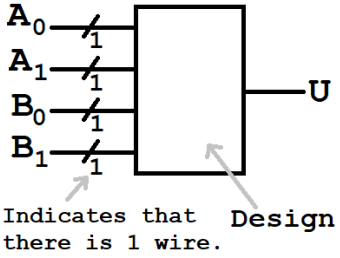
A	B	U
0	0	0
1	0	0
0	1	0
1	1	1

Another way is to use the *Boolean function* (which is the mathematical expression that represents the gate behaviour).

From this we obtain a ***schematic*** (the *circuit diagram*) which is a representation of the circuit.
To design a circuit, we start from the truth table or the Boolean functions, which are mechanical processes.

► *Example: 4 different signals, each of 1 bit*

$U=1$ only if $A_0=B_0$, $A_1=B_1$



Its truth table will be →

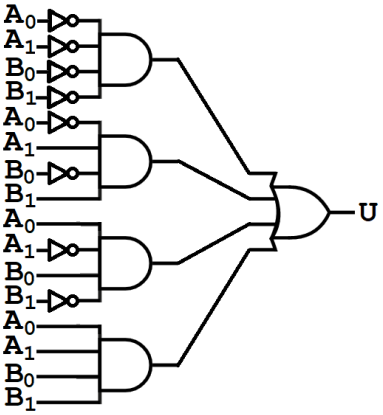
A_0	A_1	B_0	B_1	U
0	**0**	**0**	**0**	**1**
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	**1**	**0**	**1**	**1**
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	**0**	**1**	**0**	**1**
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	**1**	**1**	**1**	**1**

We need to know the *specification of our circuit*, or else we will not be able to set the truth table.

From the truth table, by selecting the rows with output equal to 1, we obtain the Boolean expression:

$$U = \overline{A_0}\overline{A_1}\overline{B_0}\overline{B_1} + \overline{A_0}A_1\overline{B_0}B_1 + A_0\overline{A_1}\overline{B_0}\overline{B_1} + A_0A_1B_0B_1$$

The result of the Boolean expression will be the *schematic*. We can draw the schematics starting from the output; consider that such circuit ***is not simplified***.



• *Optimisation*

The optimisation starts considering the circuit from the truth table or the Boolean function.

Considering XOR and XNOR truth tables, we can apply XNOR to the condition of the circuit "$U=1$ only if $A_0=B_0$, $A_1=B_1$".

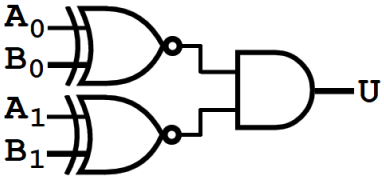
XOR		
A	**B**	**U**
0	0	0
0	1	1
1	0	1
1	1	0

XNOR		
A	**B**	**U**
0	0	1
0	1	0
1	0	0
1	1	1

Simplified expression:

$$U = (A_0 \odot B_0)(A_1 \odot B_1) \Rightarrow U = \overline{(A_0 \oplus B_0)}\,\overline{(A_1 \oplus B_1)}$$

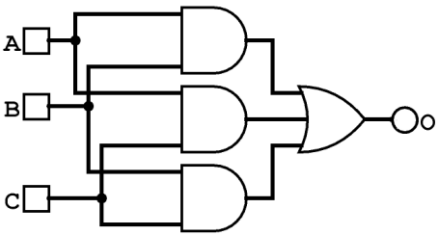
Therefore, the optimised circuit would be the following. →



► *Exercise*

Design a circuit with three input signals A, B and C whose output is at the logic value '1' only when the majority of the inputs are equal to 1.

$$U = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$
$$= AB + AC + BC$$



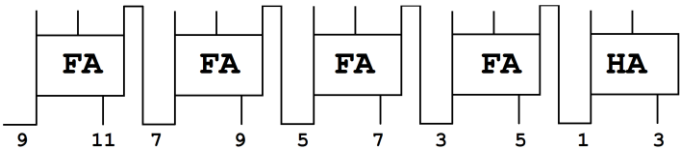
A	**B**	**C**	**O**
0	0	0	0
0	0	1	0
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
1	**0**	**1**	**1**
1	**1**	**0**	**1**
1	**1**	**1**	**1**

► *Exercise*
Going back to a previous example. There are a thousand of numbers each one of 3000 digits. What circuit to use for the addition and what would be the delay?

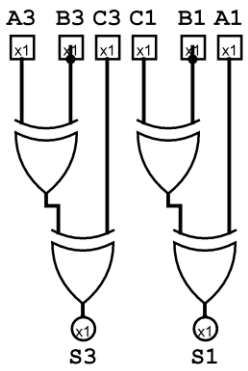
Adding up 5-bits numbers would require a 5-bit RCA, but from the second cycle it would require to be able to compute 6 bits, otherwise it would cause an overflow. We must ask ourselves: what would be the maximum number that could emerge as result of the addition? (Therefore, we do a study of the worst case)

With 5 bits, the *maximum representable number* is 11111, and the *number of possible values* is 2^5-1.
For 1000 numbers of 3000 bits, the maximum value is $2^{3000}-1$ and the number of values is
$\sim 1000\times2^{3000} = 2^{10}\times2^{3000} = 2^{3010}$

By adding bits, we can find the first object to the right being a half adder, taking into account that the first input carry would be 0. Doing so, the 5-bit RCA circuit is optimised and the delay is reduced to 11.

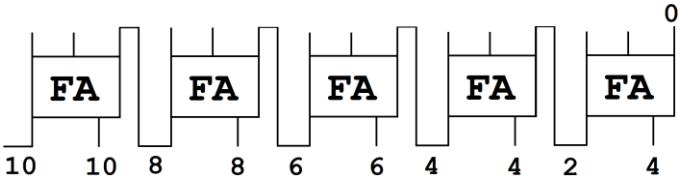


A RCA with a half adder as first adder is a possibility of optimisation but not a solution, because it will not be usable in other calculations that require a FA.



How does a full adder structure work?
Let us consider the sum part of two full adders. A and B are always immediately available, while regarding the carry C only C1 is available at the beginning. For the following Cs, the FA must wait for them to arrive from the previous FA.

The two structures would give the same results as operations, but:
- if we swap A1 with C1, the delay does not change
- ***if we swap A3 with C3***, the FA3 should wait for C3 to arrive before computing the first XOR, therefore causing more delay. Otherwise, if the first XOR is computed between A3 and B3 that are immediately available, it will have to wait for computing only the second XOR.



Now the circuit delay of the 5-bit RCA is 10.

This teaches us that if we have a previous knowledge of the problem and structures, it will help with ***optimisation***!
In this case, we have an improvement of 2 time units, which is good on a small RCA. But on a bigger circuit, let's say a 64-bit RCA, such optimisation would be minimal and negligible.

*The **critical path** is the rippling of the carry*. The correct result from the RCA will be provided only after 10 time units.

What does it happen if we take a picture at time 7? In the previous circuit only the first three FA are done, but there are cases where the result would be already available. This is the reason we always have to consider the worst delay.

● ***Boolean Circuits Characteristics***
A logic circuit is composed of gates and wires that provide inputs and outputs to and from a gate.
It is characterized by:
• *Hardware complexity / Area occupancy*: number of gates (plus number of inputs, but we ignore it).
• *Propagation Delay / Computation time*: for each gate there is a minimum time delay needed for the signal to propagate fully. This delay is the major obstacle in the development of high-speed computers and is called the interconnect ***bottleneck*** in IC systems; there are possible improvements to reduce such delay and improve the critical path.
• *Critical Path*: the longest path through which an input arrives at the end of the circuit.
• *Fan-In*: maximum number of inputs applicable to a gate.
• *Fan-Out*: maximum number of outputs that can be provided from a gate without using buffers.
• *Levels*: each gate of a digital circuit stays on a "x" level; starting from inputs, gates whose inputs are the circuit initial inputs are said to be on first level. Following gates level depends upon how many gates are present along the same path (NOT gates do not count as levels).

How to optimise further the addition circuit?
Consider the carries in a RCA:

$C_0 = 0$

$C_1 = A_0B_0 + A_0C_0 + B_0C_0$

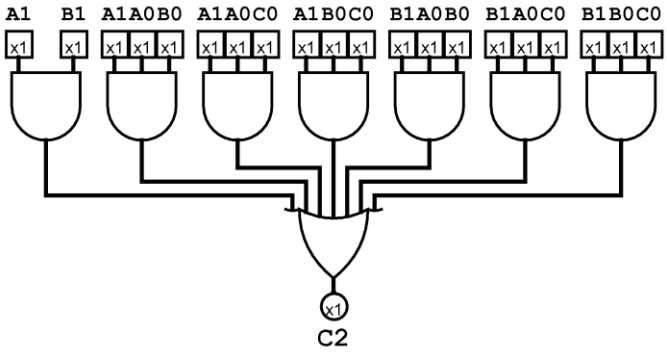
$C_2 = A_1B_1 + A_1C_1 + B_1C_1 =$

$= A_1B_1 + A_1(A_0B_0 + A_0C_0 + B_0C_0) +$

$+ B_1(A_0B_0 + A_0C_0 + B_0C_0)$

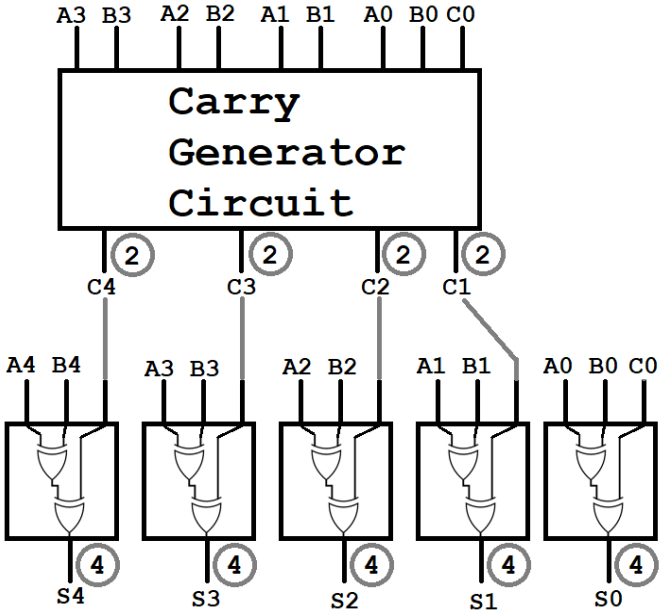
$C_3 = A_2B_2 + A_2C_2 + B_2C_2 = \ldots$

Computing C_2 requires an OR gate with 7 inputs.



All these values therefore depend only on C_0.

We create a calculator for the carries and then use them to complete the final result in a circuit that calculates only the sums. *Data received from a third party* can be used ***in parallel***.

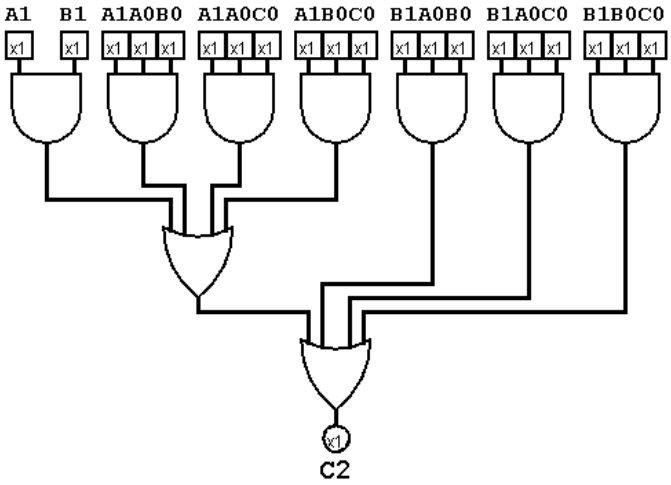


The delay is written inside the circles.

How to optimise further the circuit?
- By rearranging the circuit
- By changing the algorithm

It seems that adding will take always 4 time units. *What is wrong*? We must take into account the physical implementation: the larger number of gates, the higher the number of levels will be because of the impact of physical requirements.
It is more common and cheaper to implement a gate with fewer inputs; therefore, we will need more levels to process all the values.

For example, a closer real physical implementation for computing C_2 is the following.



As we can see, from C_2 *there are 3 levels instead of 2.*

Levels required for the calculations and gates composing them:
- $C_1 \rightarrow$ 2 levels – 2-input AND (x3), 3-input OR (x1)
- $C_2 \rightarrow$ 3 levels – 3-input AND (x7), 7-input OR (x1)
- $C_3 \rightarrow$ 3 levels – 4-input AND (x15), 15-input OR (x1)
- $C_4 \rightarrow$ 5 levels at least because there will be 5-input AND ($\rightarrow$ 2 lev.) and more than 16-input OR ($\rightarrow$ 3 lev.)

• *Fan-out, fan-in and buffer*

Fan-out is the *maximum number of digital inputs that the output of a single logic gate can* ***feed***. Usually the fan out of logic gates corresponds to 4. This is a limitation due to the passage from logical *to physical implementation*: exceeding such threshold likely makes the current and corresponding voltage not capable to drive the operation mode to the next logic gate (the signal less clear and less understandable).
A device called ***buffer*** is used ***to split a signal*** to more than 4 gates and overcome this limit.

A ***logical inverter*** (also called a NOT gate) can serve the function of a ***buffer*** / ***amplifier*** (the symbol on the left) to restore the signal: this element is implemented by **2 NOT**s (one is enough to buffer the signal, the second one restores the signal to the original value).

Fan-in is the *maximum number of digital inputs that a single logic gate can accept/receive* (number of inputs that can be *directly implemented*). A typical logic gate has a fan-in of 1 to 4. To implement more than 4 inputs, a cascade of the same gate can be used to amplify the signal without modifying its value (for example: above an OR with fan-in of 7 has been split in 2 OR with fan-in of 4 each, increasing the levels of gates).

The number of inputs (F_{IN}) has a logarithmic correspondence with the number of levels (L) needed:

$$L = \lceil \log_4(F_{IN}) \rceil$$

Because of the fan-in / fan-out, the delay of production of S_i bits is not constant, as it is reasonable to expect (and as it is in the reality).
How good is the approach outlined before? How much is the area? How big are the time advantages? When can we use the resulting values? We need to compute.

• *Carry-Lookahead Adder*

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. Although *the adder* –as any combinational circuit– *will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs* (***validity of values in memory***!).
Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical.

Employing faster gates with reduced delays obviously reduces the carry propagation delay: however, physical circuits have a limit to their capability. Another solution is to increase the ***complexity*** of the equipment in such a way that the carry delay time is reduced. There are several ways for reducing the carry propagation time in a *parallel adder*. The most widely used employs the principle of *carry-lookahead logic*.

A ***carry-lookahead adder*** (***CLA***) or ***fast adder*** or ***prediction adder*** is a type of adder used in digital logic. A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits: it calculates the carry bits before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder.

If we define two new binary variables

$P_i = A_i \text{ XOR } B_i$ $\qquad$ $G_i = A_i B_i$

the output sum and carry can respectively be expressed as

$S_i = P_i \text{ XOR } C_i$ $\qquad$ $C_{i+1} = G_i + P_i C_i$

G_i is called a ***carry generate***, and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i. P_i is called a ***carry propagate***, because it determines whether a carry into stage `i` will propagate into stage `i+1` (i.e., whether an assertion of C_i will propagate to an assertion of C_{i+1}).

We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

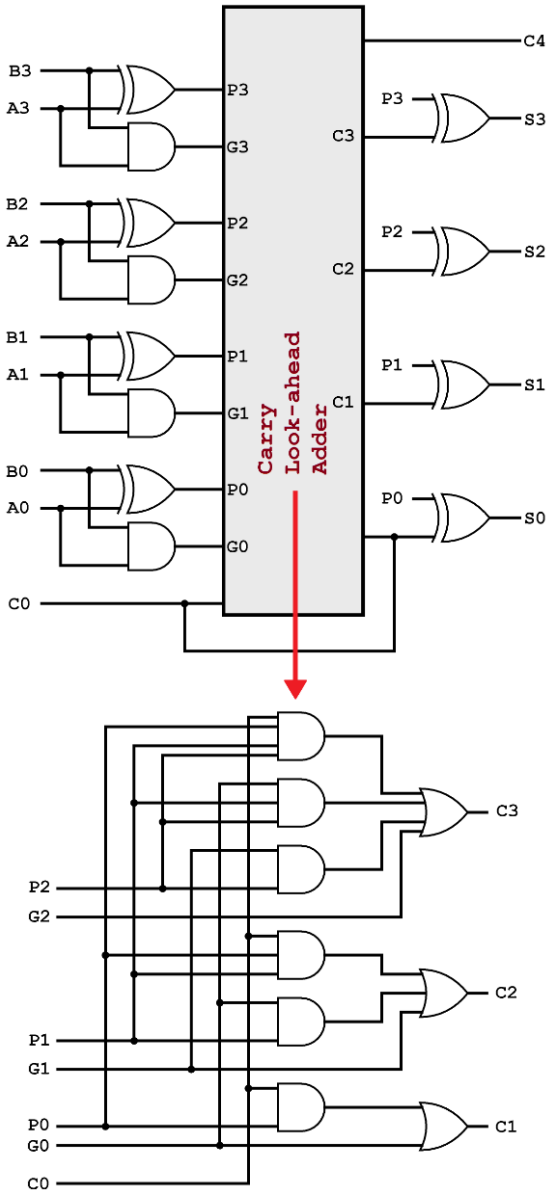
C_0 = input carry

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) =$
$= G_1 + P_1 G_0 + P_1 P_0 C_0$

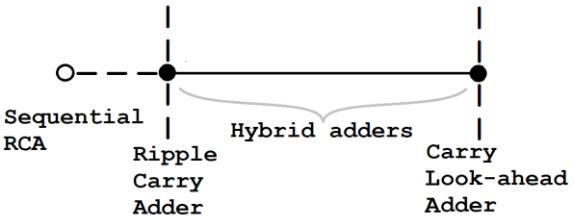
$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0)$

Note that this circuit can add in less time because C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 ***is propagated at the same time*** as C_1 and C_2. This gain in speed of operation is achieved at the expense of additional complexity (hardware).



Most problems are complex; therefore, looking for the average solution does not allow solving all the cases because of all the issues and parameters that we cannot think of. An approach to this is *studying the boundary conditions*, so the best and worst cases.

Let us study the ranges of adders we have considered:



We have to compromise between 3 characteristics: ***cost***, ***speed*** and ***complexity***. The RCA requires a *small area* (so it is cheap) but is *slow*; its design is *flexible* (which means it is not complex, build with basic blocks). The CLA requires a *huge area* and is very *fast*, but it is not *flexible* for more than 8 bits (the design for a 9-bit CLA is very different from an 8-bit). Hybrid adders combine RCA and CLA characteristics.

The *delay of a n-bit CLA* has more or less the following order of magnitude O: $delayCLA_n \simeq O(log_2 n)$

Time advantages
The delay of an **8-bit CLA** is equal to $\mathbf{3T_{FA}}$ (T_{FA} is the time delay of a full adder).
The delay of an **8-bit RCA** is equal to $\mathbf{8T_{FA}}$.
The *carry-lookahead adder* recomputes C_1, C_3, C_3… only when necessary, optimising time.

Area
$\boldsymbol{A_C}$ → the area needed for generating the basic carry, which corresponds to ***4 gates***.

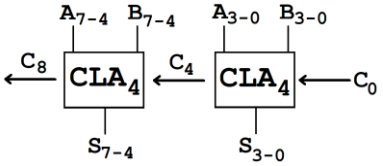
$C_1 = G_0+P_0C_0 = A_0B_0 + (A_0 \text{ XOR } B_0)C_0$
AND OR XOR AND = 4 gates

- C_1 → requires A_C area
- C_2 → requires $3A_C$ area ~ $4A_C$
- C_3 → requires $7A_C$ area ~ $8A_C$
- C_4 →requires $15A_C$ area ~ $16A_C$

The rough formula of the *area* of each new carry appears to be an exponential growth: $\mathbf{A_i = 2A_{i-1}+1}$

The ***threshold*** or ***cut-off point*** happens when the advantages in delay for computing a new C_i are not worth the huge amount of area increase.
The threshold for a CLA is 8 bits, therefore it is calculated C_8 at most.

► *Exercise 1 – design an 8-bit hybrid adder (2 CLA_4)*



The 2 blocks of CLA_4 have:
Overall Area = 2 × Area of CLA_4
Overall delay = 2 × Delay of CLA_4

Area of CLA_4? We must compute:

- C_1 → requires	A_C	1 A_C
- C_2 → requires	3 A_C	~ 4 A_C
- C_3 → requires	7 A_C	~ 8 A_C
- C_4 →requires	15 A_C	~ 16 A_C
	26 A_C	29 A_C

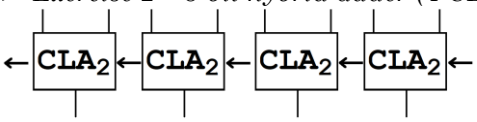
Delay $CLA_4 = \log_2(4)\times t_{FA} = 2\ t_{FA}$
We have 2 CLA_4, so the overall delay is 4 t_{FA}

How many gates are there? Every XOR is composed by 3 gates, so for 8 XORs: 8 × 3 = 24 gates

26 A_C + *sum part* =
= 26 × 4 gates + 8 XORs × 3 =
= 104 + 24 = 128 gates

So, Area of 2 CLA_4 = 2 × 128 = 256 gates

► *Exercise 2 – 8-bit hybrid adder (4 CLA_2)*



Overall Area = 4 × Area of CLA_2
Area of CLA_2?
- C_1 → requires A_C
- C_2 → requires 3 A_C
4 A_C

4 A_C + *sum part* =
= 4 × 4 gates + 4 XORs × 3 =
= 16 + 12 = 28 gates
So, Area of 4 CLA_2 = 4 × 28 =112 gates

Delay $CLA_2 = \log_2(2)\times t_{FA} = 1\ t_{FA}$
We have 4 CLA_2, so the overall delay is 4 t_{FA}

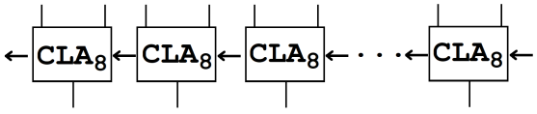
► *Comparison:*
Delay of an 8-bit RCA? 8 t_{FA}
Area of RCA? 8 × A_{FA} = 8 × (4 + 2 × (3)) = 80 gates
(A_{FA} = 4 gates for the carry + 2 XOR = 10 gates.)

Therefore, comparing these three 8-bits circuits:
- Simple RCA Area = 80 gates Delay = 8 t_{FA}
- Hybrid 1 Area = 256 gates Delay = 4 t_{FA}
- Hybrid 2 Area = 112 gates Delay = 4 t_{FA}

We see that the hybrid adders are twice as fast as a full RCA, but their area is larger.

The delay of an 8-bit CLA is 3 t_{FA} (3 = $\log_2(8)$).
Its area would be tremendously huge, so it is not convenient to use it. A hybrid solution is preferable.

► *Exercise – 64-bit hybrid adder*
The best way to implement a 64-bit adder is a hybrid solution: we connect in ripple 8 blocks of CLA_8.



Because of flexibility, to obtain a CLA that can add more than 8 bits ***we ripple them in series***.

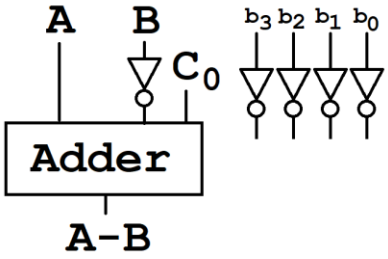
● *Binary Adder-Subtractor*

What to do if we want to build a circuit capable of performing both addition and subtraction between two numbers of N bits?

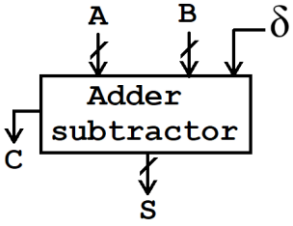
Two's complement is the most efficient system to represent negative numbers, and it is easy to be implemented in circuits.

In order to compute the two's complement of `B`:
- use inverters (so that every bit is complemented and we obtain the 1's complement)
- set the input carry C_0 to 1 (so that we add 1 to the least significant bit).

The following is the design of a simple subtractor. The `NOT` has to be applied for each bit (b_0, b_1, b_2, b_3 …).



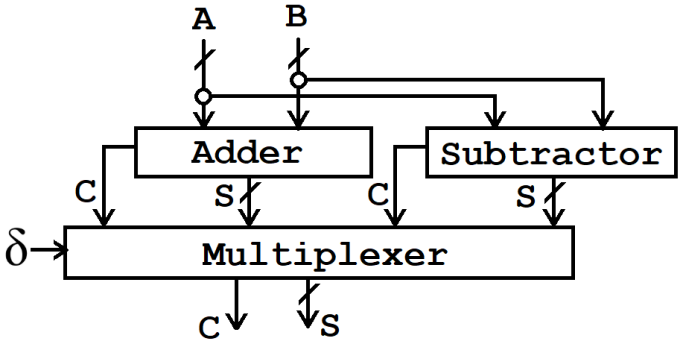
Adder / Subtractor - Approach 1



I set delta δ as a control signal:
- δ= 0 → `S=A+B`
- δ= 1 → `S=A−B`

This is an expensive approach; therefore, it is not commonly used.

Adder / Subtractor - Approach 2



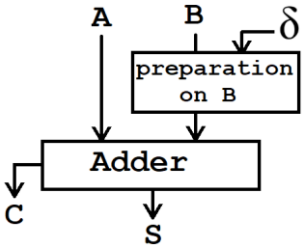
The multiplexer acts as a traffic light, letting pass the values from the adder or the subtractor according to the signal from δ.
The delay of this circuit is the sum of the delays of Adder/Subtractor with the delay of the multiplexer; the area of the circuit is about 2 times the area of an adder plus the area of the multiplexer. Therefore, it is a huge circuit with a high delay, not very convenient.

Adder / Subtractor - Approach 3

The steps to design a circuit using two's complement properties are the following:

$S \leftarrow A - B = A + (-B) \quad , \quad B \rightarrow \bar{B} + 1$

- δ= 0 → leave B unchanged
- δ= 1 → compute $\bar{B} + 1$

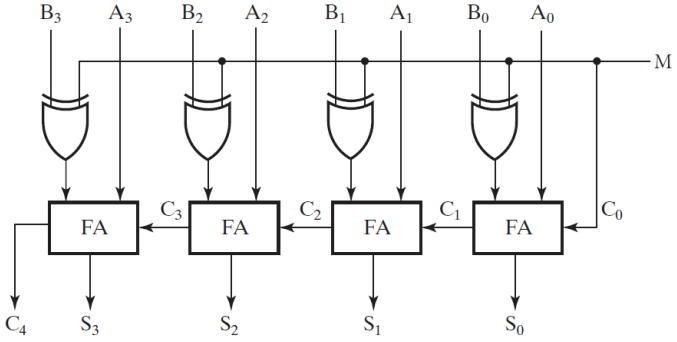


To transform `B` into `B'` we use the function

$B' = f(B, \delta) = B \oplus \delta = \bar{B}\delta + B\bar{\delta}$

Using a `XOR` gate for each b_x bit and a `M` / δ (enable) signal, we can build up a circuit that is capable of performing both addition and subtraction:
- ***Subtractor***, set `M` / δ to 1 (B_x `XOR 1 =` B_x`'`)
- ***Adder***, set `M` / δ to 0 (B_x `XOR 0 =` B_x).

► *Example of a 4-bits adder-subtractor*



Area = area of a RCA_4 + area of 4 `XOR`
Delay = delay of a RCA_4 + delay of a `XOR`

● *Addition, subtraction and overflow*

When two numbers with `n` digits each are added and the sum is a number occupying `n+1` digits, we say that an overflow occurred. This is true for binary or decimal numbers, signed or unsigned.
When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum.
Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains `n+1` bits cannot be accommodated by an `n`–bit word. For this reason, many computers detect the occurrence of an overflow and, when it occurs, a corresponding *flip-flop* is set that can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.
When *two unsigned numbers* are added, an overflow is detected from the end carry out of the most significant position.
In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2's-complement form.
When *two signed numbers* are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow. An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both either positive or negative.

To see how this can happen, consider the following example: two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary −128 (adopting 2's complement as binary number system). Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for −70 and −80.

The two additions in binary are shown next, together with the last two carries:

```
carries: 1              carries: 1   11
+70    0 1000110        −70      1 0111010
+80    0 1010000        −80      1 0110000
+150   1 0010110        −150     0 1101010
```

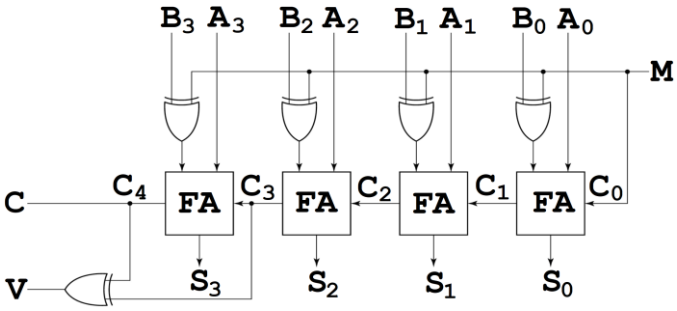
Note that the 8-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the 8-bit result that should have been negative has a positive sign bit.
If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within 8 bits, we say that an overflow has occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the examples in which the two carries are explicitly shown.

If the two carries are applied to a XOR gate, an overflow is detected when the output of the gate is equal to 1. For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

If the two binary numbers are unsigned, then the `C` bit detects a carry after addition or a borrow after subtraction.
If the numbers are signed, then the `V` bit detects an overflow. If `V = 0` after an addition or subtraction, then no overflow occurred and the n-bit result is correct. If `V = 1`, then the result of the operation contains `n+1` bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred.
The $(n+1)^{th}$ bit is the actual sign and has been shifted out of position.

► *4-bits adder-subtractor (with overflow detection)*



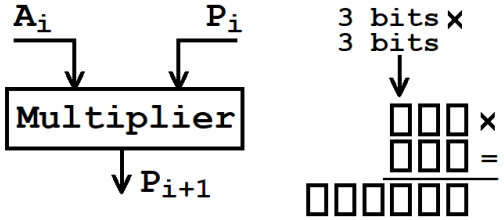
● Imprecise arithmetic / Approximate computing

Approximate computing is a computation technique which returns a possibly inaccurate result rather than a guaranteed accurate result, and can be used for applications where an approximate result is sufficient for its purpose. One example of such situation is for a search engine where no exact answer may exist for a certain search query and hence, many answers may be acceptable. Similarly, occasional dropping of some frames in a video application can go undetected due to perceptual limitations of humans. Approximate computing is based on the observation that in many scenarios, although performing exact computation requires large amount of resources, allowing bounded approximation can provide disproportionate gains in performance and energy, while still achieving acceptable result accuracy.

Consider the following problem: what is the complexity of sorting n numbers? $n \cdot log(n)$
An algorithm is a solution to a problem: in this case, sorting types of algorithms are *bubble sort* ($n^2/2$) and *merge sort* ($n \cdot log(n)$).
Counting sort is another algorithm, with complexity $O(n) < O(n \cdot log(n))$. The solution is faster than the problem itself: counting sort does not apply to the general sorting, but it applies only to a (well known) set of values within a given and known range.

► *Example*
Compute the product P of 100 terms.

$$P = \prod_{i=1}^{100} A_i \quad P_{i+1} = P_i \cdot A_i \quad P = \left((A_1 \cdot A_2) \cdot A_3\right) \cdot A_4 \ldots$$

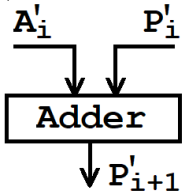


Considering the worst case: a 3 per 3 bits multiplication may have a result up to 6 digits in the case of 999×999.
Therefore, for 100 bits the result may have from 100 up to 100+100=200 bits as an output, which is not affordable and very slow.

To solve the problem, we don't need a multiplier but an adder.

$$P = \prod_{i=1}^{100} A_i \rightarrow P' = \sum_{i=1}^{100} A'_i$$

To pass from a product to an addition we can use the logarithm: $A'_i = \log_{10}(A_i)$, $P' = \log_{10}(P) \rightarrow P = 10^{P'}$
(in this case we take base 10 as an example)

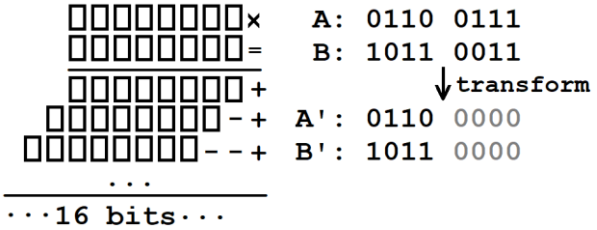


The problem cannot be solved precisely because it would need huge resources: switching to the logarithm will add some error and it will not be possible to undo it and recover the previous precision.

The approach to solve multiplication problems:
- Transform to logarithmic form
- Do the addition
- Go back to the anti-logarithms

In programming, a loop is used to initialise a huge array: to do it, some preparations are required (index, counter, zeroing, pointer, loop control…).
In case of a small array (ex. 3 elements) a loop is useless, because only 3 instructions are needed instead of at least 5 (in the loop case). We must always look for the convenient path.

► *Example: multiply two 8-bit numbers A, B*
The approach is exactly the same as in decimal.
In case of 1 → copy the line; in case of 0 → line of 0s.
The time-consuming task is adding them all.
The idea to go faster consists in using less precision, for example considering only the MSB and discarding the LSB.



$$A \cdot B \simeq A' \cdot B' = (0110\ 0000) \cdot (1011\ 0000)$$

It becomes a 4×4 bit multiplication, with the corresponding cost in time and area. The output will have an 8-bit MSB computed part and an 8-bit LSB lost part.

Other options could be neglecting the carry for the LSB, or start multiplicating from the MSB.

With limited resources, a threshold has to be respected: we need to find the compromise between hardware (resources, memory…) and the algorithm (which needs to fit such resources).

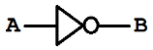
Extra note: Iteration and recursion are two opposed approach to solving problems in computer science.

Iteration is the technique marking out of a block of statements within a computer program for a defined number of repetitions.
Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem (the program computes a result and such result is used again in the same function). To do so, we must keep separated the two domains of *input* and *output*.

5 – SEQUENTIAL CIRCUITS

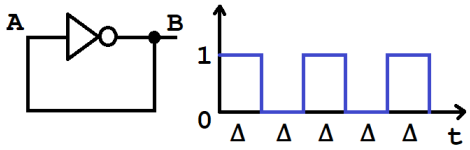
• *Sequential Circuits*

So far, we have only seen *combinational circuits* (the values of t*he output depend only on the values of the input*), like the ripple carry adder or the carry look-ahead adder.

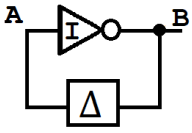


Let's consider a NOT gate.

$B = \overline{A}$



Assume we are looking to the value of B: every certain time it changes. In the ideal case, the propagation is instantaneous, so the value 0 and 1 coexists; in reality, there is a delay, as shown in the graph above.



The I inside the `NOT` gate indicates that it is an ideal gate, with no delay. At the bottom, there is a delay block Δ: this causes the value to *depend on the* ***previous history*** *of the circuit*.
This is a *sequential circuit*.

A ***sequential circuit*** is a circuit characterized by at least one *feedback path*, so that ***at least one output of a gate is also an input to another gate on the same or on a previous level of the circuit***.
Sequential circuits output depends not only on the current inputs but also on the previous state of the circuit, on *its previous history in terms of input and outputs, and they have "memory", so are suitable to build memory storage elements*.

There are two kinds of sequential circuits:
• ***Asynchronous***: the behaviour of an asynchronous sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change.
• ***Synchronous***: a synchronous sequential circuit is a system whose behaviour can be defined from the knowledge of its signals at discrete instants of time.

Asynchronous sequential circuits may lead to some instability, due to the feedback paths present, that have to be added to the always-present propagation delay times of each gate.
It is preferable to use synchronous ones, where a ***clock generator*** provides a clock signal having the form of a periodic train of clock pulses: the clock signal manages to control the delay and make it uniform.

There are two types of synchronous / asynchronous sequential circuits:
• ***Latches*** are *level-triggered*, the change of their state (in terms of outputs) happens during a precise clock signal level (high level, logic 1 / low level, logic 0).
• ***Flip-flops*** are *edge-triggered*, the change of their state (in terms of outputs) happens during a precise clock signal *transition* (from 0 to 1 or from 1 to 0).

Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits. However, flip-flops can be built starting from latches.

Types of latches:
- SR(Set-Reset): normal or with enable signal, Synchronous or Asynchronous

Types of flip-flops:
- D(Transparent) Flip-flop
- JK Flip-flop
- T flip-flop

• *SR Latch (set-reset)*

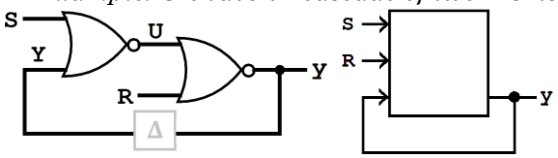
An SR latch is a sequential circuit capable of providing two stable states:
- set `S=1, R=0` → `y=1, y'=0`
- reset `S=0, R=1` → `y=0, y'=1`

It operates with signal levels, whenever they are, usually, at logic 1 value.
`S` and `R` inputs should *never* be both equal to 1, as this combination would put the circuit in an unstable and unpredictable state (`y` would result equal to `y'`, thus breaking the relationship between `y` and `y'`). That is why `S = R = 1` combination is ***forbidden***.

In order to eliminate the problem, we should insert an enable signal (`C`), which disables the circuit whenever the signal is equal to 0, or enables it whenever the signal is equal to 1.
This enhancement makes us able to have two states during which outputs `y` and `y'` do not change their values.

A *synchronous SR latch* has a clock generator, which acts as an enable signal as well.
An *asynchronous SR latch* has just a feedback path between output gate and the first gate.

► *Example:* ***SR latch*** *- cascade of two NORs*

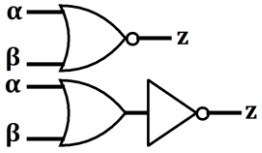


Since it has a loop back, it is a sequential circuit.
- `S` and `R` are inputs
- `y` is an output
- `Y` is an input

Assume having a looped delay from output to input: `y` will become `Y`, it is just a matter of time.

S	R	Y	y
1	0	–	1
–	1	–	0
0	0	Y	Y

What is the behaviour of the circuit?



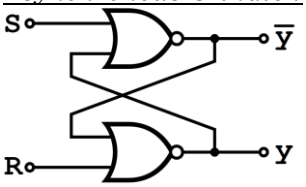
α	β	z
T	T	F
F	F	T

Independently of all the rest of the values, when:
- S=1, R=0 → y = 1
- R=1 → y = 0
- S=R=0 → y = Y (***memory configuration***)
- S=R=1 → *forbidden configuration*

Note: when S=0, the circuit becomes equivalent to two consecutives NOT, so the output value y is equal to its input Y.
The value indicated as "–" are independent from the rest of all the other values.
S = R = 1 combination is *forbidden*, because we don't know what would be the output value depending on the relative values of the inputs.

The circuit acts as a ***static memory cell***: it keeps the result previously computed and reads it in the next cycle. Combinational circuits alone cannot act as memory: none of the outputs is looped back to the inputs. The looped back variables (Y) are called ***state variables,*** because the next result depends on them.

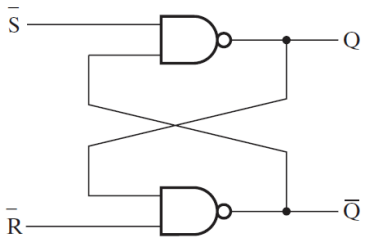
Asynchronous SR latch



S	R	y	$\bar{y}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

This circuit does the same thing of the previous one (*cascade of two NORs*), so they share the same truth table. In this case we have two loop backs, that permit us to have 2 outputs without adding any hardware.

$\overline{S\,R}$ Latch with NAND Gates



(a) Logic diagram

$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

(b) Function table

Comparing the NAND latch with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR. Because the NAND latch requires a 0 signal to change its state, it is referred to as an $\overline{SR}$ latch.

► *Why S = R = 1 combination is forbidden?*
Consider the cascade of NORs circuit.
Let us assume that the first NOR has delay Δ_1 and the second NOR has delay Δ_1, with $\Delta_1 \neq \Delta_2$.
The values of S and R are switching from 1 to 0.
Because of the different NOR delay, a conversion occurs before the other.

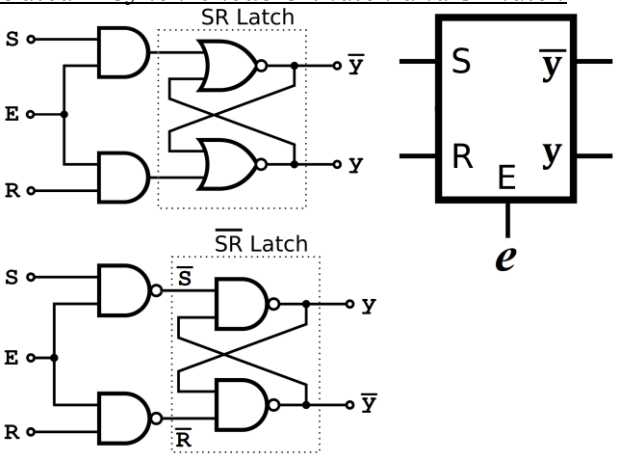
S=1, then U=0. R=1, then (R+U)'=y_1=0.
Now S=0 and y=0, the first output U=1.

- $\Delta_1 < \Delta_1$: the first NOR is faster and produces U=1 as result. By the time R is making to have 0, U=1 is passed. At time Δ_2, the second NOR computes (0,1), so y=0.
- $\Delta_1 > \Delta_2$: the second NOR is faster, its input is (0,0), so y=1. At time Δ_1, the first NOR computes (0,1), so U=0.

It is unlikely that S and R switch at the same time
In case S=R=0 (called *memory state*), the configuration has no impact.

We cannot manage to know which element changes first, ***the behaviour of the circuit is unpredictable***, so the combination is forbidden. To avoid such problem, we need an ***enable signal*** (or *clock signal*).

Gated / Synchronous SR latch and $\overline{SR}$ latch



With E=1 (enable true), the signals can pass through the input gates to the encapsulated latch; all signal combinations except for (0,0) = hold then immediately reproduce on the (y, $\bar{y}$) output.
With E=0 (enable false) the latch is closed and there are no new input from S and R. The latch remains in the state it was left the last time E = 1, acting as a memory.

With the enable signal, ***S and R are not bouncing anymore from 0 to 1: the enable signal will go to 1 only when both S and R signal are stable***; this lasts until both signals are propagated, then E=0.

Function table of a gated $\overline{SR}$ latch →

C	S	R	**Next state of y**
0	X	X	No change
1	0	0	
1	0	1	y = 0 reset state
1	1	0	y = 1 set state
1	1	1	Undefined

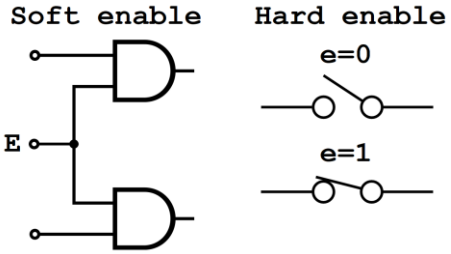
● *Enable signal*

The enable signal can have two values:

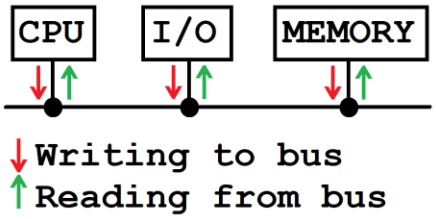
- false (or *disable*, `e=0`)
- true (`e=1`) forces the circuit to work properly.

It can also be of different types.

- the ***soft enable*** is implemented with gates. When false (`e=0`), it forces all its *outputs to be 0*; the circuit would be still connected electrically because it is obtained using gates.
- the ***hard enable*** is implemented physically. When false, it cuts the wires and forces an electrical disconnection.



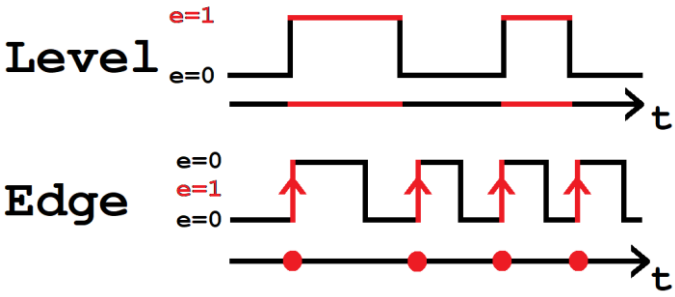
An important use for the ***hard enable*** signal is in the Von Neumann architecture.



If the connection dots were simple connections, in case of more than one device writing the information delivered over the buss would be a mess: it would become the combination of the data written by the two devices to the bus and therefore corrupted. Therefore, *only one **writing** can occur at any time, but more than one **reading** can take place*: this is permitted by using a hard disable to prevent writing (a soft disable keeps producing 0 as an output, so it is not the correct choice for this case). The connection between the bus and each computer elements is physically implemented by ***two monodirectional lines*** (instead of a single bidirectional line, which is a simplification in representing it).

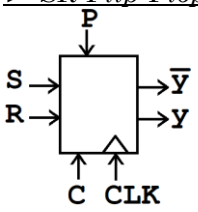
Another distinction of enable signals is the following.

- ***Level enable*** (or *pulse triggered*): it depends on the level of the signal, so it may last both a short or a long time; it is implemented using an `AND` gate. When the clock signal is true (=1) the circuit works, when false (=0) the circuit keeps its state (no action). The consequence is that, while the enable is 1, any transition which occurs is transmitted to the output.
- ***Edge enable***: the enable occurs when the signal goes *from low to high level*; it requires complex circuits so it is more expensive than the level type. This type is very precise: it is like a level enable with a very narrow window of time.
  A *rising edge* (or positive edge) is the low-to-high transition; a *falling edge* (or negative edge) is the high-to-low transition.
  Edge-triggered flip-flop trigger only during a transition of the clock signal (from 0 to 1 or from 1 to 0) on the clock and it is disabled at all other times, including for the duration of the clock pulse.



As said before, the difference between a latch and a flip-flop is that a latch is ***level-triggered*** (outputs can change as soon as the inputs changes), a flip-flop is ***edge triggered*** (only changes state when a control signal goes from high to low or low to high).

► *SR Flip-Flop with preset and clear*



S → set
R → reset
P → preset
C → clear
CLK → clock
y , $\overline{y}$ → output

Clock signal input is indicated as a small triangle inside the schematic.

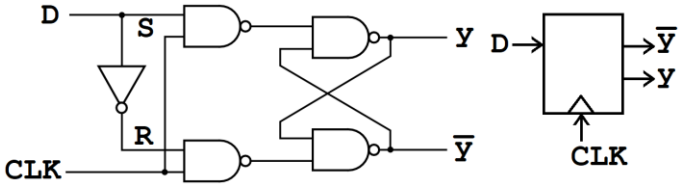
Preset and clear are two asynchronous (not related to the clock) input lines:

- `P` stores a 1 in the flip-flop
- `C` stores a 0 in the flip-flop

They are used in case of recovery and emergency, therefore not very frequently.

● *D Flip-Flop (delay)*

D flip-flops can be built by using SR latches; they are commonly used for counters, shift-registers and input synchronisation.

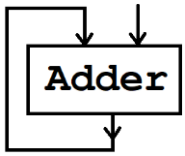


Because D arrives as S input before R, a problem is the possibility of multiple state changes as long as the enable (clock) signal is 1. To avoid such problem, the clock is fundamental: in a D flip-flop, the output can be only changed at the *clock edge*, and if the input changes when the clock is disabled, the output will be unaffected and the flip-flop will ***act as a 1-bit of static memory cell***.

D	CLK	y	$\overline{y}$
0	↑	0	1
1	↑	1	0
X	0	y_0	$\overline{y_0}$
X	1	y_0	$\overline{y_0}$

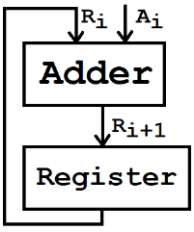
● *Registers*

Let us consider the following adder. Would it be working properly? No, because the result is not stored: it is looped back too fast. While the adder is still computing, some bit value would arrive at the looped back input.



The consequence is that it modifies the input, and the whole circuit is compromised. Sometimes it may work, sometimes not: we cannot rely on dependences cases of a circuit.

To solve this problem, we need a register: each value sent from the adder is stored in the register.



$$R_{i+1} = R_i + A_i \qquad R = \sum_{i=0}^{n} A_i$$

$$R_0 = 0$$

$$R_1 = R_0 + A_0 = A_1$$

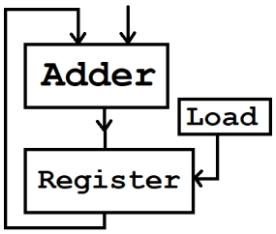
$$R_2 = R_1 + A_1 = A_0 + A_1$$

A register *is a group of* ***D flip-flops****: each of them share a common* ***clock*** *and is capable of storing one bit of information*.
An *n*-bit register consists of a group of *n* flip-flops and it is capable of storing *n* bits of binary information.

The register has a delay because it needs some time to store the data inside itself.
Overall delay = adder delay + register delay
The designer has to tune the clock of the register according to the delay of the circuit, taking in account the delay of the adder, so that the signals R_i and A_i are synchronised together:
Improving the critical path of the adder in this case would mean speeding up the clock, having a faster circuit.

In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation. The flip- flops hold the binary information, and the gates determine how the information is transferred into the register.

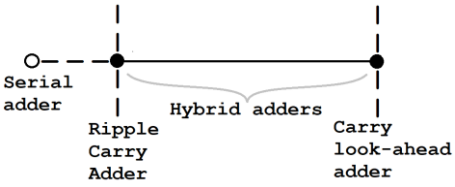
The ***load*** element is a control system equivalent to the enable signal, which tells the register when to store data and when to flush it. Otherwise, the register would be flipping signals.



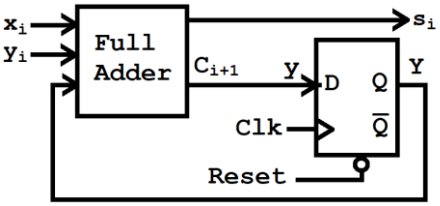
A change in value on the control input allows the state of a latch in a flip-flop to switch. This change is called a ***trigger***, and it enables, or triggers, the flip-flop.

● *Serial adder*

Here are, from the left to the right, all types of adders ordered by the cheaper to the more expensive.



The *serial binary adder* (or bit-serial adder or *sequential adder*) is a digital circuit that performs binary addition of two *m*-bit numbers (X, Y) bit by bit, lowest (LSB) to highest (MSB), one per clock cycle.
It is composed by a *D flip-flop* and a *full adder*.



The FA receives the first couple of bits x_0 and y_0, saves the carry C_1 to the D flip-flop (computed at time 2) and sends the sum s_0 to the output (computed at time 4). Then the cycle restarts, receiving as inputs x_i, y_i and the eventual c_{i+1}.
The D flip-flop avoids the new carry to arrive at the adder before the sum is calculated, which would cause an error because it would replace the old carry.

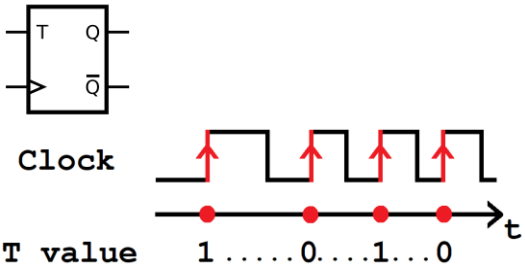
Delay = (*time of a FA + time to store D-FF*) × *m*
Area = ($Area_{FA} + Area_{FF}$)

The serial adder is the cheapest and slowest adder.
In the RCA and CLA all the inputs arrive in parallel at the same time; in the serial adder inputs arrive sequentially from LSB to MSB.

● *T Flip-Flop (toggle)*

A T flip-flop has just two inputs (*clock* and *T*) and one output. Depending on the input T and the clock, it can assume two states:

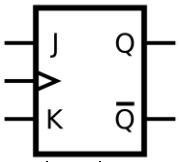
- Memory configuration, previous state is kept (T = 0)
- Output is the complement of the stored value (T=1); the T flip-flop is *edge-triggered*, so the complement of the previous value is computed only at the rising/decreasing edge of the clock.



● *JK Flip-Flop*

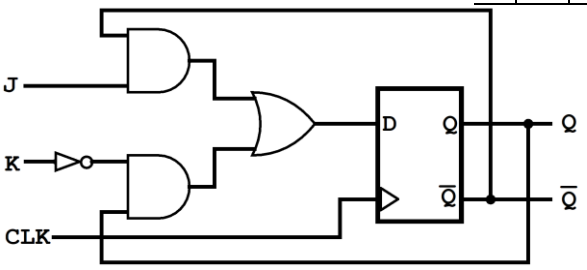
A JK flip-flop has the following states:

- No change
- Complement Output
- Reset
- Set



J	K	$Q_{(t+1)}$
0	0	$Q_{(t)}$
0	1	0
1	0	1
1	1	$\overline{Q_{(t)}}$

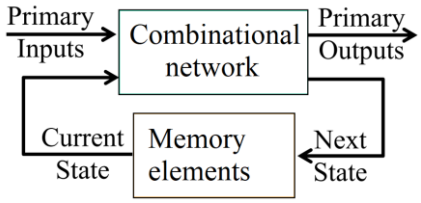
`D = JQ'+K'Q`

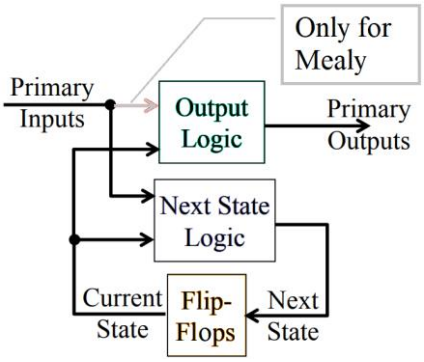


Setting `J = K = 0` maintains the current state.
To *synthesize a D flip-flop*, simply set `K` equal to the complement of `J`.
To *synthesize a T flip-flop*, set `K` equal to `J`.
The JK flip-flop is therefore a *universal flip-flop*, because it can be configured to work as an SR flip-flop, a D flip-flop, or a T flip-flop.

● *Huffman Model*

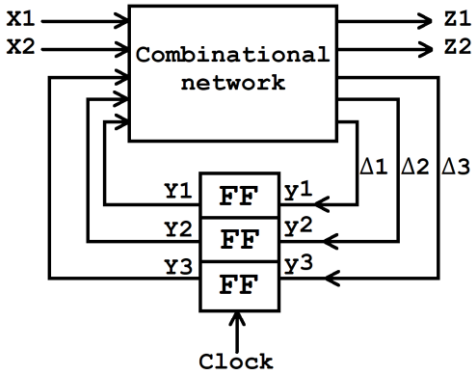
Huffman model is an architecture model used to describe sequential synchronous circuits and their difference from combinational circuits.





Mealy model circuits: the outputs depend on the inputs, as well as on the states.
Moore model circuits: the outputs depend only on the states (they are function of current state only, so they can change only when the current state changes).

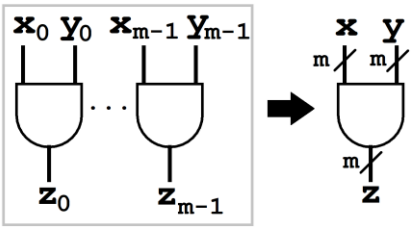
In a Mealy model, some of the outputs are looped back as inputs. The flip-flop stores the N outputs from the combinational network that are looped back (the number of flip-flops needed is $\log_2 N$).
The ***looped back outputs may have different delays*** ($\Delta 1$, $\Delta 2$, $\Delta 3$), so I set a clock that at some time flushes the flip-flops stored values back to the combinational network.
The memory elements are synchronous ***D flip-flops***.



The part of the combinational circuit that generates the signals for the inputs of flip-flops can be described by a set of Boolean functions called *flip-flop input equations*.

● Register-transfer level

Some sequential circuits with complex specifications require a long time to be implemented correctly: because of this, working at logical gates level is inconvenient.
It exists a higher level, called register-transfer level (RTL): it is a design abstraction, which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals.

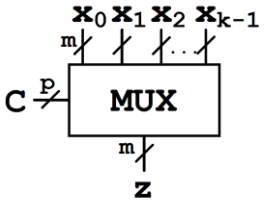


This model uses the *word* (sequence of N bits) instead of the bit as elementary unit.

Given two words of m bits each, an AND logic gate will have a word z as output, always m bits long.
In z each bit is obtained by the AND implementation between each i^{th} bit respectively of x and y.

● Multiplexer

A multiplexer (MUX) is a combinational circuit that selects one of the inputs based on ***control signals*** applied to the circuit.
A multiplexer (represented in the 1st figure) has:
- an input of k words, each of m bits;
- a p-bit control signal C ($p=log_2 k$, so for a ***2^p input*** multiplexer, there are ***p selection signals***);
- an output z of m bits.

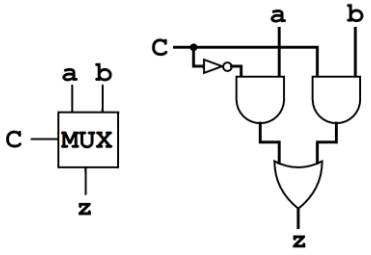


The simplest multiplexer has k=2, m=1 (2-1 MUX). It is also referred as ***fundamental multiplexer***. Its truth table is the following.

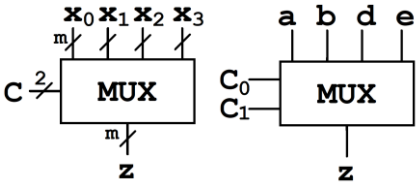
C	z
0	a
1	b

From the truth table of the 2-1 MUX, I obtain its function and therefore its diagram.

$$z = f(c,a,b) = \bar{c}a + cb$$



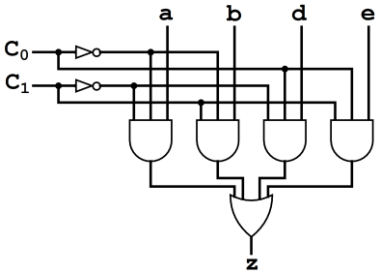
In a 4-to-1 line multiplexer (4-1 MUX) there are:
- four inputs,
- two selection lines,
- an output line.



In particular (2nd figure), when C_0=C_1=0, **a** is selected and it's provided at the output; when C_0=0, C_1=1, **b** is selected and it's provided at the output line, and so on, according to the following truth table.

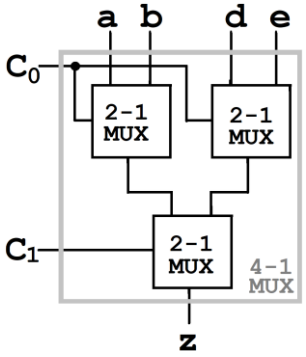
From the truth table of the 4-1 MUX, I obtain its function and therefore its diagram.

$$z = \overline{c_0}\,\overline{c_1}a + \overline{c_0}c_1 b + c_0\overline{c_1}d + c_0 c_1 e$$



C0	C1	z
0	0	a
0	1	b
1	0	d
1	1	e

A 4-1 MUX can be designed using three 2–1 MUXes:
- the first level can share the same control signal;
- the second has its own.



Comparing the two designs of the 4-to-1 MUX:
- *Boolean* is faster, cheaper (area = 5)
- *Hierarchical* is slower and more expensive (area = 9) but also scalable, easier to design and more practical to test.

Scalability is the property of a system to handle a growing amount of work by adding resources to the system (therefore extending the circuit).

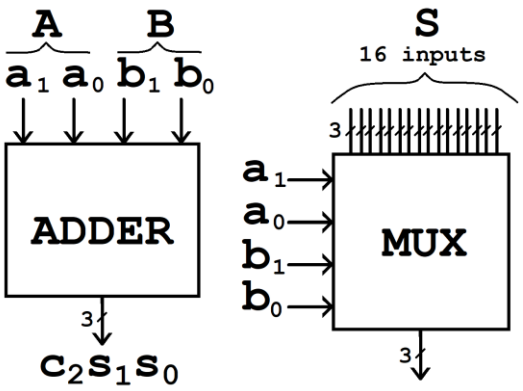
Some circuits can be used in a non-conventional way.

►*Using the MUX as an adder*
Add up two numbers A, B each one of 2 bits.
The truth table of the corresponding adder is the following.

A		B		S		
a_1	a_0	b_1	b_0	C_2	s_1	s_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	1

Obviously, ***some combinations are repeated*** because the addition is a symmetrical operation.

Looking at the truth table, there are:
- 16 possible values of S, so 16 inputs to the MUX
- 4 control signals ($p=log_2 k \rightarrow 4=log_2 16$) corresponding to the values $a_1\ a_0\ b_1\ b_0$ from the truth table.

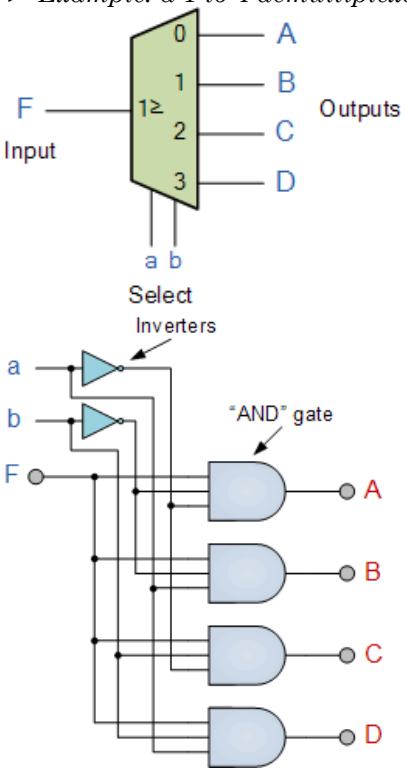


The MUX we obtained is equivalent to the adder.

A multiplexer approach can be used to implement any truth table, but too many inputs would require a huge area and that would be unnecessary (ex. the addition of A, B with 64-bit each would have $p=128$ control signals, $k=2^{128}$ inputs).
MUX is convenient only when p is small.

• *Demultiplexer*
A demultiplexer (or ***demux***) is used when a circuit wishes to send a signal to one of many devices: it takes one single input data line and then switches it to any number of individual output lines one at a time.
The demultiplexer converts a serial data signal at the input to a parallel data at its output lines (for 2^N outputs, there should be at least N select signals).

► *Example: a 1 to 4 demultiplexer*



► *Note: Demultiplexer is not requested for the exam* ◄

● *Finite state machine (Sequence Recognizer)*

The functional relationships among the inputs, outputs, and flip-flop states of a sequential circuit can be enumerated in a ***finite state machine table***.
A ***state table*** consists of four sections, labelled *present state*, *input*, *next state* and *output*:
- the present state designates the state of flip-flops before the occurrence of a clock pulse;
- the next state shows the states of flip-flops after the clock pulse;
- the output section lists the value of the output variables during the present state.

The information available in a state table may be represented graphically in the form of a state diagram. Each state is represented by a circle, and the transition between states is indicated by directed lines (or arcs) connecting the circles.

►*Example of search algorithm (sequence recognizer)*
Recognise the sequence 00 .
1. This is the text to search through.
2. The first value does not correspond to the search.
3. The second value corresponds to the first element.
4. Second and third values correspond to the search: sequence found!
5. If asked to look for all the sequences ***without overlap***, we skip to the fourth value and go on.
6. If asked to look for all the sequences ***with overlap***, the third and fourth values correspond as well.

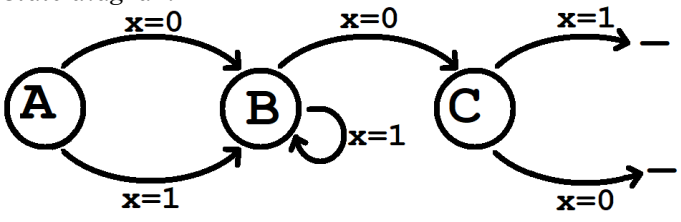
```
1.  1000100
2.  1000100
3.  1000100
4.  1000100
5.  1000100
6.  1000100
```

►*Example of state diagram and table*
Draw the state diagram to find the sequence –0:
- without overlap and stop the search immediately after a recognition;
- with overlap and keep searching.

"–" or "*don't care*" symbol means any digit, both 0 or 1.

- ***Without overlap and stop the search***
"*stop the search*" can be also found in the form "*the first occurrence*" or "*execute as soon as / immediately*".

State diagram



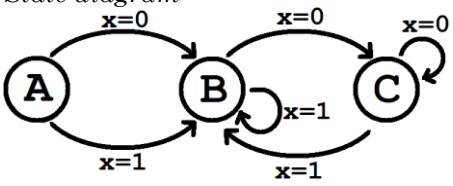
The *state table* is the following.
x is the input bit
z is the output (1 / on = seq. found , 0 / off = not found)

	I am in	*Input value*	Current state	Go to	Future state	*Output value*
		x	$Y_1\ Y_0$		$y_1\ y_0$	z
Nothing found	A	0	0 0	B	0 1	0
	A	1	0 0	B	0 1	0
First found	B	0	0 1	C	1 0	**1**
	B	1	0 1	B	0 1	0
String found	C	–	1 0	–	– –	–

State D doesn't exist: it would correspond to 11, but we don't care about it, as we are asked only to arrive at C and stop the recognition. We still have to represent it on the eventual K-map, because any map has 2^n columns: there cannot exist a 3-columns map.

- ***With overlap and keep searching***
"*keep searching*" can be also found in the form "*find all the occurrences*", " *immediately starts a new one (in correspondence of the next input)*" or " *restart the search as soon as the next symbol is received*"

State diagram



State table

	I am in	*Input value*	Current state	Go to	Future state	*Output value*
		x	$Y_1\ Y_0$		$y_1\ y_0$	z
Nothing found	A	0	0 0	B	0 1	0
	A	1	0 0	B	0 1	0
First found	B	0	0 1	C	1 0	**1**
	B	1	0 1	B	0 1	0
String found	C	0	1 0	C	1 0	**1**
	C	1	1 0	B	0 1	0

Let's obtain a *variable equation* from the state table.
K-map of y_1 (obtained from the state table)

x \ Y_1Y_0	00	01	11	10
0	0	1	–	1
1	0	0	–	0

Therefore, the corresponding equation:
$y_1 = \bar{x}Y_1 + \bar{x}Y_0$

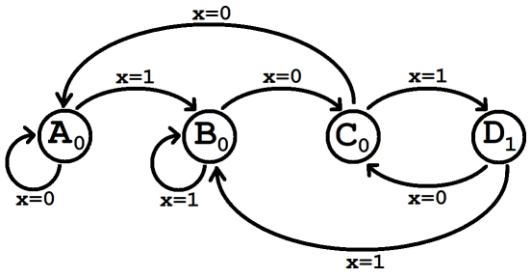
"***restart recognition after a reset symbol***" or "***returning to the initial searching state after one input***" means that, after finding the sequence, we have to go back to the starting point A (with a "don't care") and restart.

There are cases in which with and without overlap state machines and tables may correspond.
Note: for the computation of all equations please look at the exams in the next pages.

► *Test 2013-06-02 – Exercise 5*

Please provide the picture of the state machine of a sequencer recognizing the serial input 101 and immediately starting a new recognition even in partial overlap with the current just ended. Please also provide the maps and equations the final circuit implementing the sequencer.

The *state diagram* is the following.



We need 2 flip-flops to code the 4 states:
A_0 → nothing found
B_0 → first found 1
C_0 → second found 10
D_1 → third found 101

When the state A_0 receives:
- 0 → return to A_0
- 1 → change state

We are recycling, but this will imply an overlap so we jump to the previous step (D_1 → C_0).
This model is called "cellular automata".

To give a name to each state we need two bits; a possible choice can be:

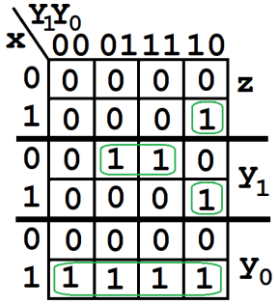
		Y_1	Y_0
Nothing found	A_0	0	0
First found	B_0	0	1
Second found	C_0	1	0
Third found	D_1	1	1

We have:
- 3 inputs (name of the state we are in and the bit that has just arrived)
- 3 outputs (destination state and light/output)

The *state table* is the following.
x is the input bit
z is the light (1 / on = seq. found , 0 / off = not found)

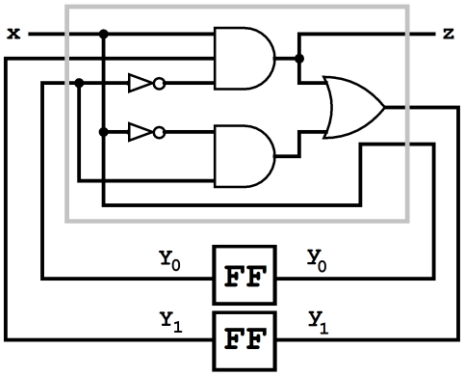
	I am in	*Input value*	Current state	Go to	Future state	*Output value*
		x	Y_1 Y_0		y_1 y_0	*z*
Nothing found	A	0	0 0	A_0	0 0	0 / OFF
	A	1	0 0	B_0	0 1	0 / OFF
First found	B	0	0 1	C_0	1 0	0 / OFF
	B	1	0 1	B_0	0 1	0 / OFF
Second found	C	0	1 0	A_0	0 0	0 / OFF
	C	1	1 0	D_1	1 1	**1 / ON**
Third found	D	0	1 1	C_0	1 0	0 / OFF
	D	1	1 1	B_0	0 1	0 / OFF

The exam asks to study the outputs of the four states. Let us look to the four states' truth tables (Karnaugh maps) for each flip-flop plus one for the output.



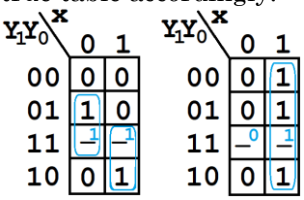
From the table of *z*: $z = x Y_1 \overline{Y_0}$
From the table of y_1: $Y_1 = \overline{x} Y_0 + x Y_1 \overline{Y_0}$
From the table of y_0: $y_0 = x$

The corresponding circuit is the following.



Here is a variation of the problem:
1. Go back to the second if 1, go back to nothing if 0.
2. If I am asked to keep light on, then if 0 or 1 arrive I go to the third state.
3. No obligation to keep on or switch off, it does not matter!

Let us consider the last two rows again and draw their true table accordingly.



The – indicates that it could be either a 0 or a 1, it depends on us (= ***don't care***).

This means that all remains the same as before, except for the column 11: from such column, we can jump to any state.
Therefore, it is better to put 11 in the first table and 01 in the second, because we avoid having additional terms: $Y_1 = \overline{x} Y_0 + x Y_1$ $y_0 = x$

The suggestion in case we have a do not care, it is better to put 0 unless I need a 1 for collecting.

► *Test 2020-02-10 – Exercises 1, 3, 5*

Exercise 1

A synchronous counter:
A) Is a sequential circuit
B) Normally is based on F-Flip-Flops
C) Normally is based on multiplexers
D) None of previous answers
Solution: A

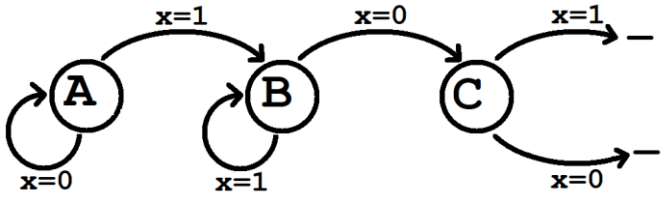
Exercise 3

A 4-bit ripple carry adder (at least one is correct):
A) Is a combinational circuit
B) Requires only AND and OR gates to be implemented
C) Requires only 8 multiplexers, without additional logic
D) Can be implemented by using D flip-flops
E) Can store up to 16 values
F) None of the above
Solution: A

Exercise 5

Please provide the picture of the state machine of a sequencer recognizing the serial input 10 and ending the search *as soon* the first sequence has been recognized. Please also provide the maps and equations of the final circuit implementing the sequencer.

The *state diagram* (or *state machine*) is the following.



When a state receives:
- A → 0, keep looking for 1
  → 1, found first symbol, go to B
- B → 1, keep looking for 0
  → 0, string found, go to C
- C → because we found what we looked for, either there is 1 or 0 we don't care, the designer is free to do whatever he wants.

We need 2 flip-flops to codify the 4 states:
A → nothing found
B → found first symbol 1
C → found the string 10
To give a name to each state we use two bits (state variables); a possible choice can be:

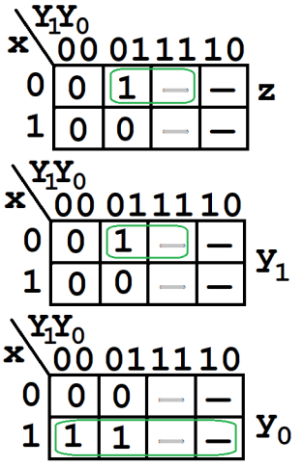
		Y_1	Y_0
Nothing found	A	0	0
First found	B	0	1
String found	C	1	0

The *state table* is the following.
z is the light (1 / on = seq. found , 0 / off = not found)

	I am in	*Input value*	Current state	Go to	Future state	*Output value*
		x	Y_1 Y_0		y_1 y_0	*z*
Nothing found	A	0	0 0	A	0 0	0
	A	1	0 0	B	0 1	0
First found	B	0	0 1	C	1 0	1
	B	1	0 1	B	0 1	0
String found	C	–	1 0	–	– –	–

State D doesn't exist: it would correspond to 11, but we don't care about it, as the exam asks only to arrive at C. But we still have to represent it on the K-map, because any map has 2^n columns: there cannot exist a 3-columns map.

We have:
- 3 inputs (1 input *x* , 2 current state Y_1 , Y_0)
- 3 outputs (1 output *z* , 2 future state y_1 , y_0)



The exam asks to study the outputs of the 4th state.

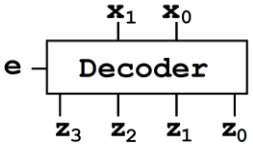
Let us look to the Karnaugh maps:
- from the table of *z*: $z = \bar{x}Y_0$
- from the second table: $Y_1 = \bar{x}Y_0$
- from the third table: $Y_0 = x$

• *Decoder*

A decoder is a combinational circuit that converts binary information from *n encoded inputs* and provides a maximum of $k=2^n$ *decoded outputs*.
A decoder may have one or more enable signal:
- when true, the decoder works according to the truth table;
- when false / disabled, all decoder outputs are forced to their inactive states.

For each input combination there can be a single output as true, all the others are false.

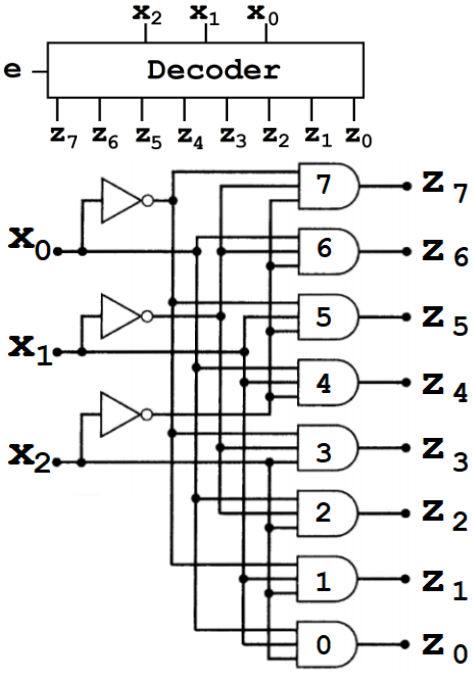
► *Here is a 2–4 decoder and its truth table.*



x_1	x_0	e	z_3	z_1	z_2	z_0
0	0	1	0	0	0	1
0	1	1	0	0	1	0
1	0	1	0	1	0	0
1	1	1	1	0	0	0
–	–	0	–	–	–	–

The decoder works as a sort of selector: the input selects an output line to set to 1, all the others are 0.

► *Here is a 3–8 decoder and its truth table.*

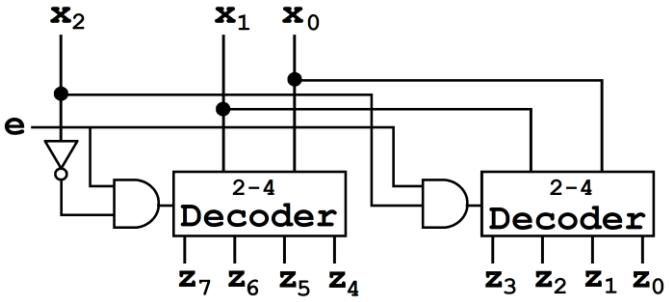


x_2	x_1	x_0	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**

► *Implement a 3–8 decoder with two 2–4 decoders.*

The problem asks to scale up the circuit from 2-4 to 3-8 decoder.
From the previous truth table, we have that when:
- $x_2=0 \rightarrow z_7, z_6, z_5, z_4$
- $x_2=1 \rightarrow z_3, z_2, z_1, z_0$

How to connect x_2 to the two decoders? Firstly, we thought to use it directly as the enable signal of the decoders, but to make also this circuit ***scalable*** we combine it with the actual enable:
- $e=0$ → everything is switched off
- $e=1$, $x_2=0$ → the decoder to the left works
- $e=1$, $x_2=1$ → the decoder to the right works



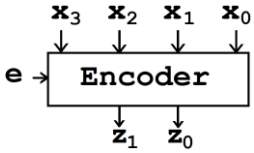
This structure is equivalent to the previous 3–8 decoder.

• *Encoder*

An encoder is a combinational circuit that is capable of encoding 2^n inputs and providing n outputs lines, which should represent the encoded inputs. It does the reverse function of a decoder.

It is ***not*** possible to reverse an encoder to obtain a decoder and vice-versa, as well as any other logic component: transistors and ***gates cannot be reversed***, it is both electrically and logically impossible (only the function can be reversed).

► *Here is a 4–2 encoder truth table.*



x_3	x_2	x_1	x_0	z_1	z_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	1	0	1	Forbidden	

In a ***pure encoder*** it is ***guaranteed*** that there is one and only one input set to true: only the configurations having a single 1 are considered by classical encoders, because they require one and only one input true (and all the others false).
Considering this case, the 12 configurations with more than one input as 1 are all *forbidden*.

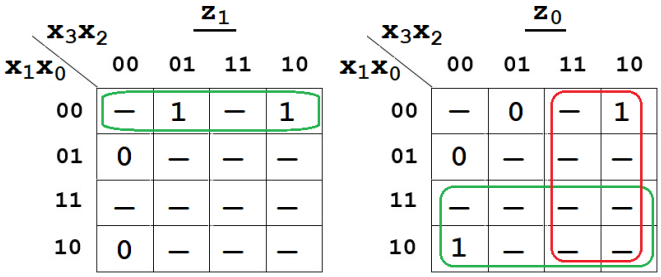
• *Priority Encoder*

When two or more inputs are equal to 1 at the same time, the output is obviously not correct. The problem can be solved by using the *priority encoder*, which inspects most significant bits (leftmost positions) and, if one of them is equal to 1, the output can be delivered without having to worry about least significant bits, whose values are expressed as *don't cares*. In a priority encoder there is at least a "true" in the group of inputs: the all-zero configuration is forbidden.

Note that the truth table has fewer inputs because *don't cares* can assume both 1 and 0 values, thus representing two different numbers on the same line.

x_3	x_2	x_1	x_0	z_1	z_0
0	0	0	1	0	0
0	0	1	–	0	1
0	1	–	–	1	0
1	–	–	–	1	1

From the pure encoder truth table, the Karnaugh maps for z_1 and z_0 are the following:



We are interested only in the valid values of z_1 and z_0: all the forbidden configurations are represented as don't cares.

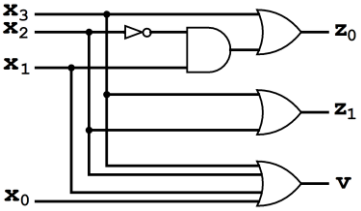
The corresponding functions are:

- $z_1 = \overline{x_1}\,\overline{x_0}$ or $z_1 = x_2 + x_3$
- $z_0 = x_3 + x_1$

Encoders have small issues with a soft enable:
- if it is true (e=1), the circuit works properly;
- if it is false (e=0), the circuit emits 0 as output signal, so the circuit cannot know if this means that the soft enable is disabled or encoding the 0001 configuration.

► *4-2 priority encoder with validity check*

v informs the user of inputs validity, as at least one input should be equal to 1.

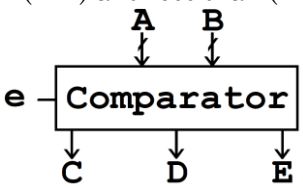


x_3	x_2	x_1	x_0	z_1	z_0	v
0	0	0	0	–	–	0
0	0	0	1	0	0	1
0	0	1	–	0	1	1
0	1	–	–	1	0	1
1	–	–	–	1	1	1

• *Comparator*

Along with being able to add and subtract binary numbers, we need to be able to compare them and determine whether the value of input A is greater than, smaller than or equal to the value at input B.

There are two main types of digital comparator.
- ***Identity Comparator***: it has only one output terminal for when A=B, either A=B=1 (HIGH) or A=B=0 (LOW).
-
- ***Magnitude Comparator***: it has three output terminals, one each for equality (A=B), greater than (A>B) and less than (A<B).



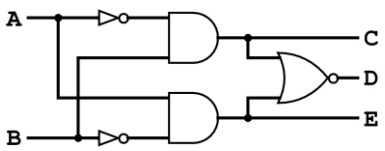
$C = \overline{A}B \rightarrow A < B$

$D = \overline{\overline{A}B + A\overline{B}} = AB + \overline{A}\,\overline{B} \rightarrow A = B$

$E = A\overline{B} \rightarrow A > B$

► *1-bit Digital Comparator*

Here are a 1-bit comparator diagram and truth table.



A	B	C	D	E
0	0	0	1	0
1	0	1	0	0
0	1	0	0	1
1	1	0	1	0

The boolean functions for computing the equality between two bits a and b are:

1. $a \oplus b \rightarrow \begin{matrix} \mathrm{T},\ a \neq b \\ \mathrm{F},\ a = b \end{matrix}$

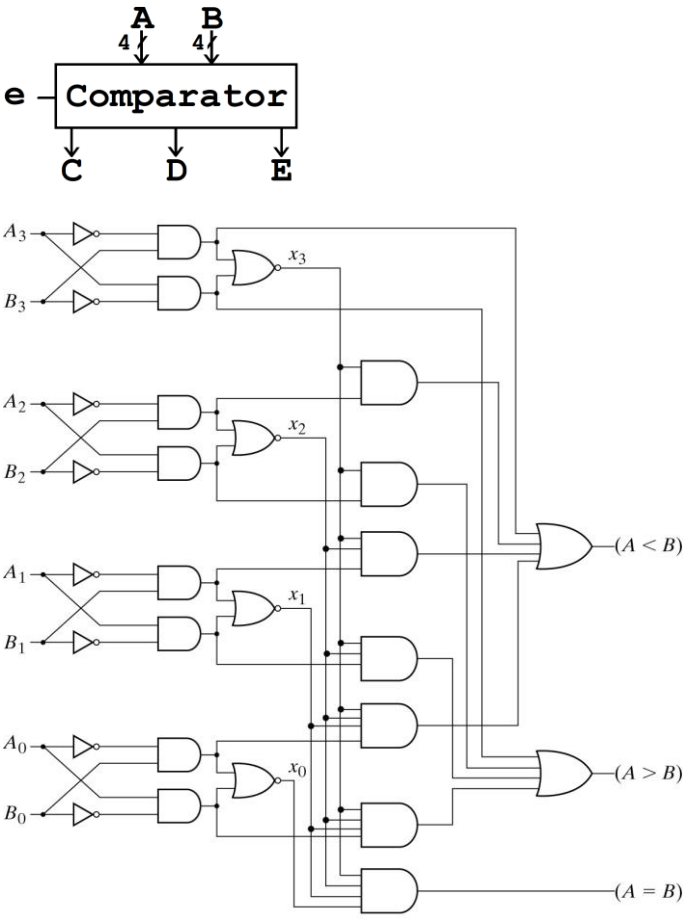
$f : \overline{a \oplus b} = \overline{\overline{a}b + a\overline{b}} = (a + \overline{b}) \bullet (\overline{a} + b) \Rightarrow a = b$

2. $\alpha = \overline{a},\ \alpha \oplus b \rightarrow \begin{matrix} \mathrm{T},\ a = b \\ \mathrm{F},\ a \neq b \end{matrix}$

$f : \overline{a}b + a\overline{b} = ab + \overline{a}\,\overline{b}$

► *4-bit and 8-bit Digital Comparator*

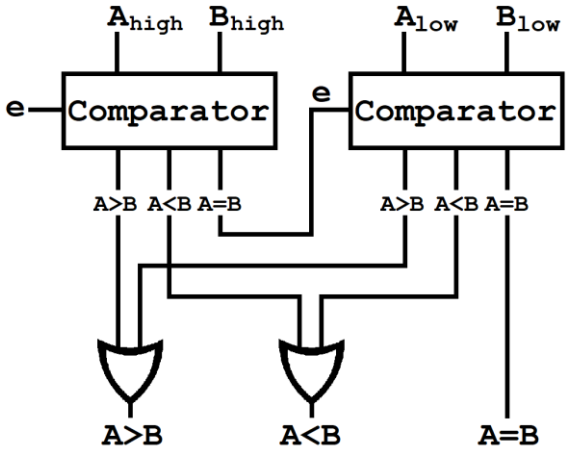
Here is the circuit diagram of a 4-bit comparator.



How to design a 8-bit binary comparator?

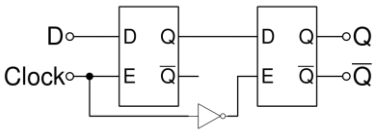
We can use two 4-bit comparators, checking bit by bit, starting from the MSB (highest position) and proceeding to the LSB (lowest), so that as soon as we find that A≠B we can end the comparison and obtain the result.

The A=B result is used as enable signal for the next comparator: when A≠B the other comparator will be disabled, when A=B the other comparator will run.



● *Master–slave flip-flop*

A master–slave D flip-flop is created by connecting two gated D latches in series, and inverting the enable input to one of them: the second latch in the series changes only in response to a change in the first (master) latch.

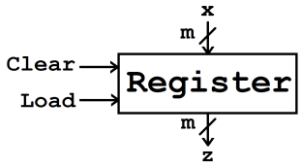


For a *positive-edge* triggered master–slave D flip-flop, when the clock signal is low (= 0) the "enable" seen by the first or "master" D latch (the inverted clock signal) is high (= 1). This allows the "master" latch to store the input value *when the clock signal transitions from low to high*. As the clock signal goes high (0 to 1) the inverted "enable" of the first latch goes low (1 to 0) and the value seen at the input to the master latch is "locked". Nearly simultaneously, the twice inverted "enable" of the second or "slave" D latch transitions from low to high (0 to 1) with the clock signal. This allows the signal captured at the rising edge of the clock by the now "locked" master latch to pass through the "slave" latch. When the clock signal returns to low (1 to 0), the output of the "slave" latch is "locked", and the value seen at the last rising edge of the clock is held while the "master" latch begins to accept new values in preparation for the next rising clock edge.

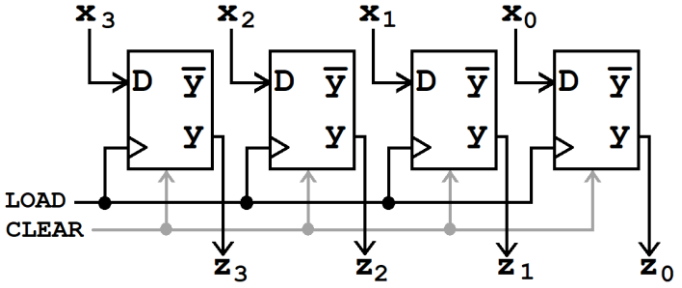
● *Registers (2)*

A register is a sequential circuit that can store *m* bit of data: each bit is stored in a *D flip-flop*. In addition to the input, there are two other signals:

- *Clear*, to set all the bits stored as 0;
- *Load*, that makes the register flush the data to the output.



► *4-bit register*



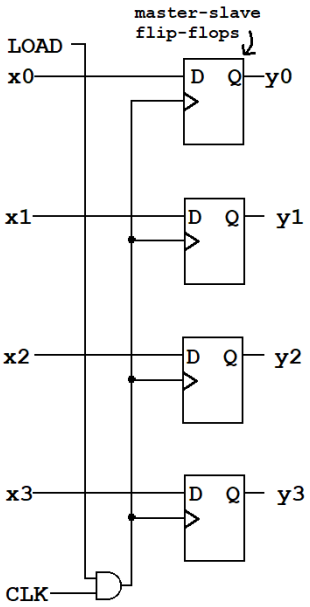
● *Parallel Load Registers*

Registers with parallel load are a fundamental building block in digital systems. It is important that you have a thorough understanding of their behaviour.

Synchronous digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied to all flip-flops and registers in the system. The master clock acts like a drum that supplies a constant beat to all parts of the system.
A separate control signal must be used to decide which register operation will execute at each clock pulse. The transfer of new information into a register is referred to as ***loading*** or ***updating*** the register.

A ***parallel load*** happens if all the bits of the register are loaded simultaneously with a common clock pulse (so the loading is done in parallel).
A clock edge applied to the `Clock` inputs of the register will load all four inputs in parallel, if `Load` is set to `1`. If the contents of the register must be left unchanged, there are two options:
- the inputs must be held constant;
- the clock must be inhibited from the circuit.

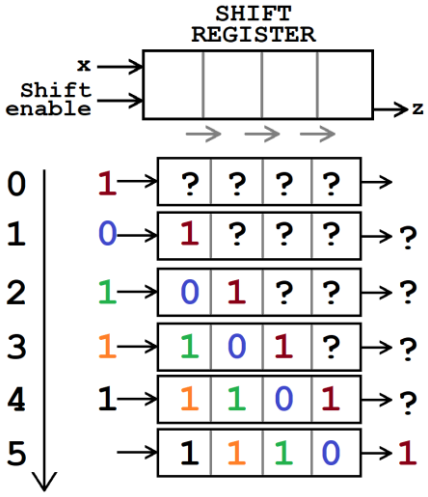
In the first case, the data bus driving the register would be unavailable for other traffic.
In the second case, the clock can be inhibited from reaching the register by controlling the clock input signal with an enabling gate. However, *inserting gates into the clock path is ill advised* because it means that logic is performed with clock pulses. The insertion of logic gates produces *uneven propagation delays* between the master clock and the inputs of flip-flops.
To fully synchronize the system, we must ensure that all clock pulses arrive at the same time anywhere in the system, so that all flip-flops trigger simultaneously.
Performing logic with clock pulses inserts variable delays and may cause the system to go out of synchronism. For this reason, it is advisable to *control the operation of the register with the* `D` *inputs*. This creates the effect of a gated clock, but without affecting the clock path of the circuit.



● *Shift registers*

A *shift register* is capable of shifting the binary information held in each cell to its neighbouring cell, in a selected direction. It can be built to *shift left*, *shift right*, or in order to be able to do both, eventually both *serially* or in *parallel* (*universal shift register*).

Here is a 4-bit right shift register.

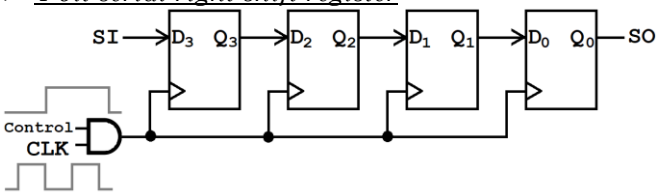


The logical configuration of a shift register consists of a chain of ***D flip-flops*** in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next. Furthermore, the data path of a digital system is said to operate in ***serial mode*** when information is transferred and manipulated one bit at a time. Information is transferred one bit at a time by shifting the bits out of the source register and into the destination register. This type of transfer is in contrast to ***parallel transfer***, whereby all the bits of the register are transferred at the same time.

In the *parallel mode*, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse.

In the *serial mode*, the registers have a single serial input and a single serial output. The information is transferred one bit at a time while the registers are shifted in the same direction.

► *4-bit serial right shift register*



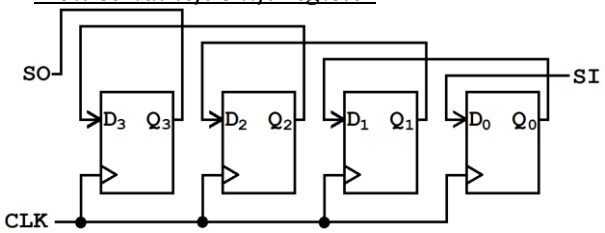
This block diagram consists of three synchronous D flip-flops, which are cascaded left to right.
The *serial output* (SO) of one D flip-flop is connected as the *serial input* (SI) of next D flip-flop: these shift registers are doing a serial transfer of information.

For every positive edge triggering of clock signal, the data shifts from one stage to the next.
Therefore, we can receive the bits serially from the output (SO) of right most D flip-flop.

SO of register 3 is connected to SI of register 2: the initial content of register 2 is shifted out through its serial output to register 1, then to register 0.
From register 0 such data is lost unless it is transferred to a third shift register.

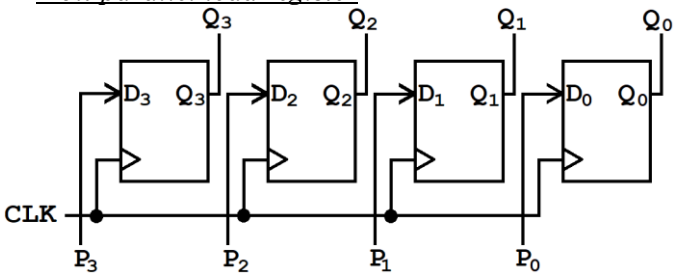
The *shift control input* (or *shift enable*) determines when and how many times the registers are shifted.
For illustration here, this is done with an AND gate that allows clock pulses to pass into the CLK terminals only when the shift control is active.

► *4-bit serial left shift register*



What are the uses of shift registers? They can implement addition, multiplication, division serialisation and parallelisation…

► *4-bit parallel load register*

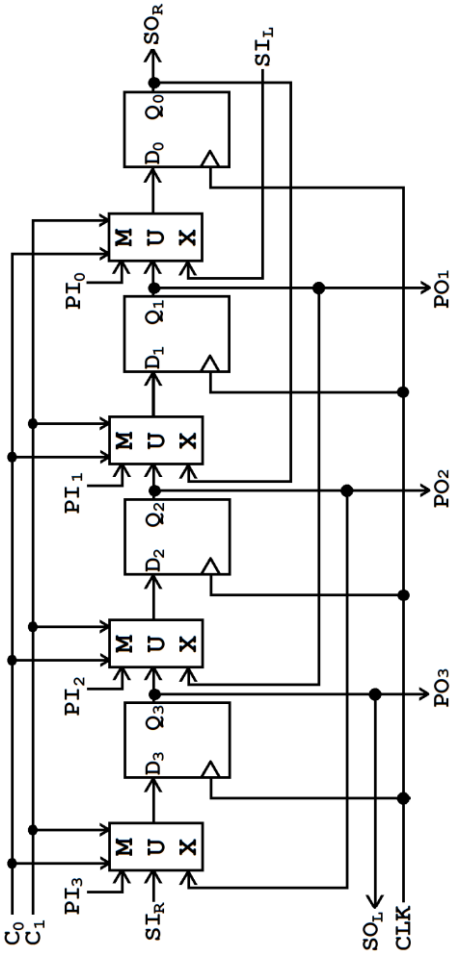


This type of register also acts as a temporary storage device or as a ***time delay device***, with the amount of time delay being varied by the frequency of the clock pulses. In addition, in this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required.

More the features of a register, more expensive it will be.

► *4-bit register with shift and parallel load*

We need two control signals (C_0, C_1) or a 2-bit control signal for the MUX to choose which input needs to proceed.
Using C_0, C_1 we combine the following features:
- 00 *the circuit is left open (disable)*
- 01 *shift right to left* (SI_L , SO_L)
- 10 *shift left to right* (SI_R , SO_R)
- 11 *parallel load* (get inputs PI + load registers PO)



For *multiplication*, *left shift registers* are used.
In multiplication by 10, values are shifted from right to left by one position. The result is correct only when the *register drops a 0 from the left*; when it drops a 1 there is an overflow and the result is incorrect.

```
              Drop  Result
00123 × 10 →   0   01230
12300 × 10 =   1   23000 ERROR!
```

For *division*, *right shift registers* are used.
In division by 10, values are shifted from left to right by one position. *Whatever the register drops* from the right, the division is correct: such division is between two integers, so there cannot be any value after the unit. The value dropped from the register is the eventual remainder (or residual) of the division.

```
              Result  Drop
12345 / 10 = 01234     5
```

How works a multiplication by integers?

```
00123 ×
   12 =
00246 +  ← 00123 × 2
01230    ← 00123 × 10
```

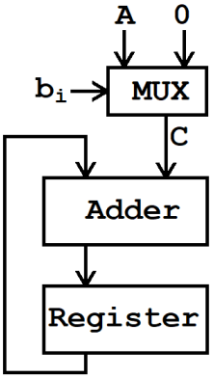
How works a binary multiplication?

```
      001011 ×
      011011 =
     0001011 +  ← 001011 × 1
    00010110 +  ← 001011 × 10
   00000000 +  ← 001011 × 000
  0001011000 +  ← 001011 × 1000
 0001011 0000 +  ← 001011 × 10000
000000000000 =  ← 001011 × 000000
000100101001
```

Each digit in the multiplier generates a *partial product*. All these partial products are added to produce the final product value.
When the multiplier bit zero, the partial product is zero, and when the multiplier bit is 1, the resulted partial product is the multiplicand.
Each successive partial product is shifted one position left relative to the preceding partial product before summing all partial products.
Therefore, this multiplication uses n-shifts and adds to multiply n-bit binary number. The combinational circuit implemented to perform such multiplication is an *array multiplier* or *combinational multiplier*.

<u>*How to implement a multiplier*</u>?
We use an adder and a register.

```
A = 001011
B = 001011
A × B
```



The MUX works according to b_i:
- b_i=0 the output (C)is 0
- b_i=1 the output (C) is A, followed by a proper number of 0 according to the position of i.

Therefore:
- A moves in a left shift register
- B moves in a right shift register

Each time a row is produced, the MUX decides what to pick and flush to the adder, so that it can be summed with the previous result.

How to simplify the MUX?
The A left shift register each time drops its value and then is filled with a 0.
The B right shift register each time drops a b_i.
A and b_i are connected to an AND gate:
- b_i=0 the output (C) is 0
- b_i=1 the output (C) is A

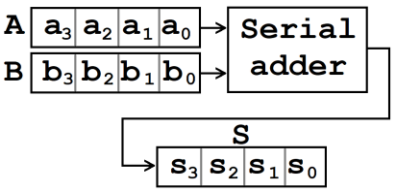
```
                              A3   A2   A1   A0 ×
                              B3   B2   B1   B0 =
                            B0A3 B0A2 B0A1 B0A0 +
                       B1A3 B1A2 B1A1 B1A0      +
                  B2A3 B2A2 B2A1 B2A0           +
             B3A3 B3A2 B3A1 B3A0                +
P7   P6   P5   P4   P3   P2   P1   P0
```

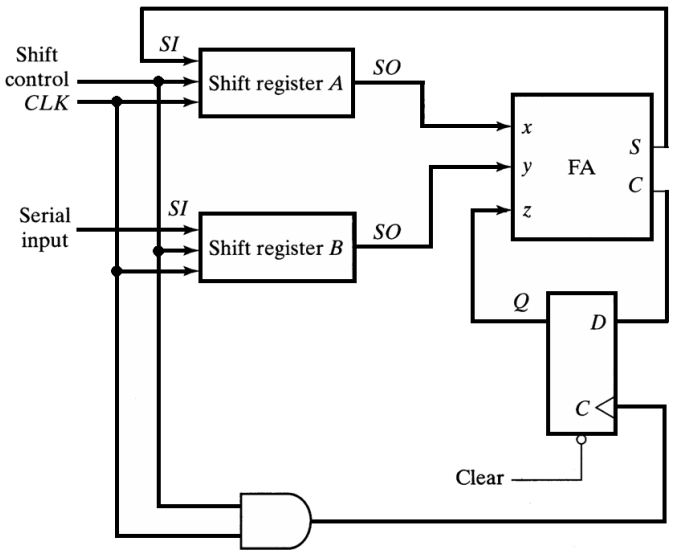
```
P0 = B0A0
P1 = B0A1 + B1A0
P2 = B0A2 + B1A1 + B2A0 + (Carry of P1)
...
P7 = Carry of P6
```

● Serial Adder Implementation

Operations in digital computers are usually done in parallel because that is a faster mode of operation. Serial operations are slower because a data path operation takes several clock cycles, but serial operations have the advantage of requiring fewer hardware components.

To implement a serial adder, we can use shift registers.





The two binary numbers to be added serially are stored in two ***shift right registers***. Beginning with the least significant pair of bits, the circuit adds one pair at a time through a single full-adder (FA) circuit, as shown in figure.
The carry out of the full adder is transferred to a D flip-flop, the output of which is then used as the carry input for the next pair of significant bits.
The sum bit from the S output of the full adder could be transferred into a third shift right register. By shifting the sum into A while the bits of A are shifted out, it is possible to *use one register for storing both the augend and the sum bits*.
The serial input of register B can be used to transfer a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is as follows:
- Register A holds the augend, register B holds the addend, and the carry flip-flop is cleared to 0.
- The outputs (SO) of A and B provide a pair of significant bits for the full adder at x and y. Output Q of the flip-flop provides the input carry at z.
- The shift control enables both registers and the carry flip-flop, so at the next clock pulse, both registers are shifted once to the right, the sum bit from S enters the leftmost flip-flop of A, and the output carry is transferred into flip-flop Q.
- The shift control enables the registers for a number of clock pulses equal to the number of bits in the registers. For each succeeding clock pulse, a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right.

This process continues until the shift control is disabled. Thus, the addition is accomplished by passing each pair of bits together with the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, into register A.

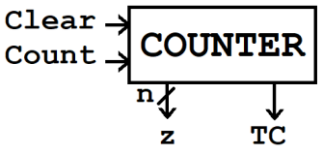
Initially, register A and the carry flip-flop are cleared to 0, and then the first number is added from B. While B is shifted through the full adder, a second number is transferred to it through its serial input. The second number is then added to the contents of register A, while a third number is transferred serially into register B. This can be repeated to perform the addition of two, three, or more four-bit numbers and accumulate their sum in register A.

Comparing the serial adder with the parallel adder (binary adder-subtractor), we note several differences. The parallel adder uses registers with a parallel load, whereas the serial adder uses shift registers. The number of full-adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full-adder circuit and a carry flip-flop.
Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit that consists of a full adder and a flip-flop that stores the output carry.

This design is typical in serial operations because the result of a bit-time operation may depend not only on the present inputs, but also on previous inputs that must be stored in flip-flops.

● *Counter (increment register)*

A counter is sequential circuit that counts events and stores its value in a register, going through a prescribed sequence of states upon the application of input pulses.



`Clear` sets to `0` the value of `z` (`Preset` sets it to all `1`). `Step` (or `count` or `increment`) is the value of which the counter (as `z`) in incremented / decremented each time. When `z` reaches its maximal / minimal value, at the next step it will reset to `0/MAX`. The terminal count (`TC`) warns the user when a reset of `z` happens.

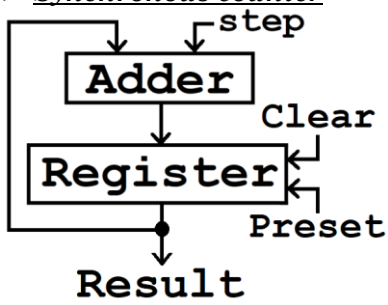
In a *programmable counter* the step value can be modified.

When step is equal to:
- +1, the counter is called *up counter*
- –1, the counter is called *down counter*

A counter with n-bit digits can count up to 2^n values: it is referred as ***counter on n bits*** or ***counter modulo*** $\boldsymbol{2^n}$: for example, a counter on 4 bits it is also referred as counter modulo 16.

The *program counter* (PC), commonly called the *instruction pointer* (IP), in microprocessors is a processor register that indicates where a computer is in its program sequence. The PC is incremented after fetching an instruction, and holds the memory address of ("points to") the next instruction to execute.

► ***Synchronous counter***



This synchronous counter is made combining an ***adder*** with ***D flip-flops***.
All the components are already explained above.

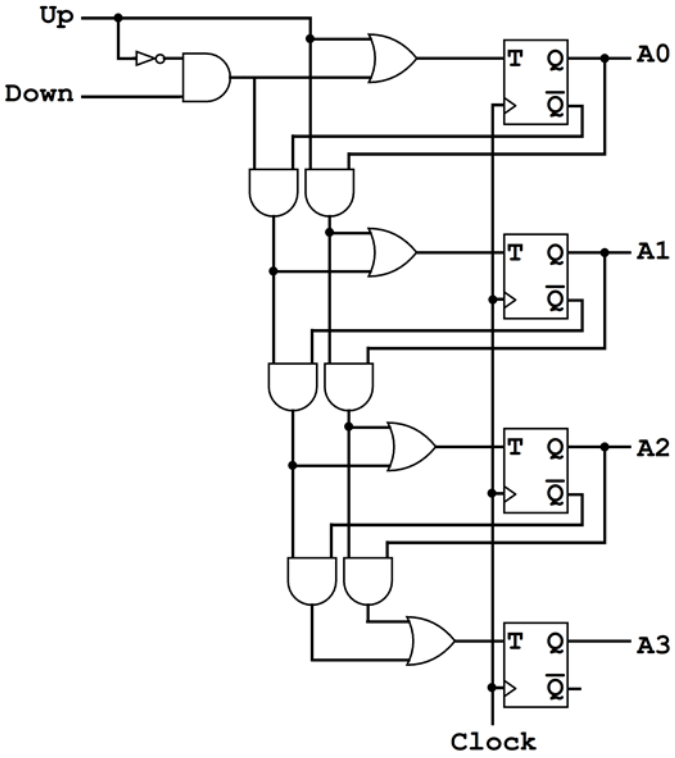
This counter is synchronous because the register is loading the output coming from the adder each time according to the clock. The output from the register is both sent as result (`z` above) and looped back to the adder for the next operation.

Its basic element can also be the ***T flip-flop****, as showed in the next example.*

► *Up-down counter on 4 bits*

A synchronous countdown binary counter goes through the binary states in reverse order, from 1111 down to 0000 and back to 1111 to repeat the count. It is possible to design a countdown counter in the usual manner, but the result is predictable by inspection of the downward binary count. The bit in the least significant position is complemented with each pulse. A bit in any other position is complemented if all lower significant bits are equal to 0.

For example, the next state after the present state of 0100 is 0011. The least significant bit is always complemented. The second significant bit is complemented because the first bit is 0. The third significant bit is complemented because the first two bits are equal to 0. The fourth bit does not change, because not all lower significant bits are equal to 0.

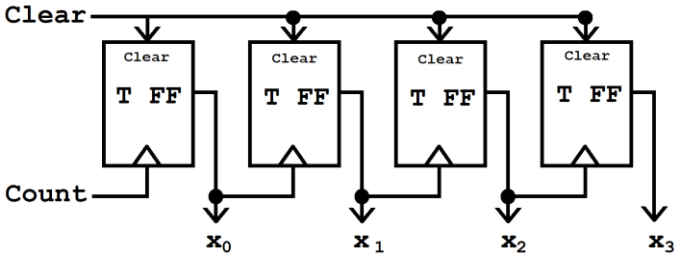


The two operations can be combined in one circuit to form a counter capable of counting either up or down. It has an up control input and a down control input. When the up input is 1, the circuit counts up, since the T inputs receive their signals from the values of the previous normal outputs of the flip-flops. When the down input is 1 and the up input is 0, the circuit counts down, since the complemented outputs of the previous flip-flops are applied to the T inputs.

When the up and down inputs are both 0, the circuit does not change state and remains in the same count. When the up and down inputs are both 1, the circuit counts up. This set of conditions ensures that only one operation is performed at any given time.

Note: *the up input has priority over the down input.*

● *Asynchronous counter*



*Its basic element is the **T flip-flop**.*
The `count` signal represents the clock input of the first TFF; it goes out from the output and proceeds to the clock input of the next TFF.

What is the circuit's behaviour?
- 1st TFF commutes at each tick
- 2nd TFF commutes each 2 ticks
- 3rd TFF commutes each 4 ticks…

State table

Time	LSB			MSB
(edge)	x_0	x_1	x_2	x_3
0	0	0	0	0
1	1	1	1	1
2	0	1	1	1
3	1	0	1	1
4	0	0	1	1
5	1	1	0	1
6	0	1	0	1
7	1	0	0	1
8	0	0	0	1
9	1	1	1	0

This is a ***down counter***; it is cheap and easy to make. The circuit is triggered on the rising edge of the clock. The delay grows proportionally to the number of TFF, because each of them has a frequency dependent to the previous TFF: not all the counting steps have the same duration, therefore such circuit is *asynchronous*.

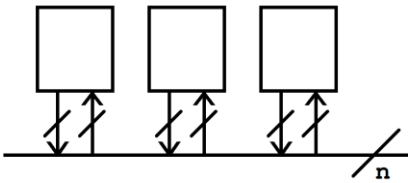
From when the clock signal arrives, each transition requires about the delay of a TFF to happen.

How to change this circuit to an up-counter?
The trigger has to be on the falling edge. There would still be the problem of the position of LSB and MSB, a solution could be negating the output.

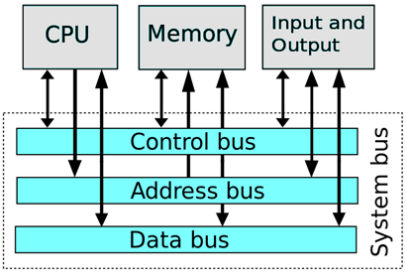
What is the difference in implementation between a ***synchronous counter*** and an ***asynchronous counter*** both using ***T flip-flops*** as basic elements?
It depends entirely on the *clock propagation*:
- in synchronous counters, the clock arrives *in parallel* (at the same time) to all the T flip-flops;
- in asynchronous counters, the clock arrives only at the first T flip-flop, then it *is rippled* to all the others as the TFF output (so it arrives with a delay depending on the times it has been rippled).

6 – BUS

As already said in the introduction, the bus belongs to the fourth class of elements we can find in a computer: it is a communication system that transfers data between components inside a computer (CPU, memory, I/O). This expression covers all related hardware components (wire, optical fiber, etc.) and software, including communication protocols. Since many devices have the possibility to access simultaneously the bus, we should find a way to provide bus access to one device at a time.



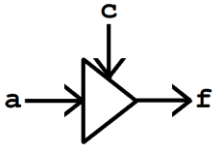
A bus has **n** lines connected to modules able to read and/or write on it. Only one module can write on the bus at each time; meanwhile, all the other modules can read the written information (so it can be used by more components at the same time).



• *Tri-state buffer*

The three-state (or tri-state) buffer is a device that allows the correct behaviour of a bus, by controlling when an output signal makes it to the bus.

When the tri-state buffer's control bit is active, the input of the device makes it to the output: this happens when the "valve" is open.
When it is not active, the output of the device is Z, which is high-impedance or, equivalently, nothing: this happens when the "valve" is closed, and no electrical signal is allowed to pass to the output.



Elements of this tri-state buffer:
- **c** control signal
- **a** input
- **f** output

c	a	f
0	0	Z
0	1	Z
1	0	0
1	1	1

The Z indicates high-impedance state.

Tri-state buffers are located between each module and the bus.
The CPU controls the state of the buffer.

To be included in any architecture, each device needs to be able to be hard enabled.

7 – MEMORY ELEMENTS

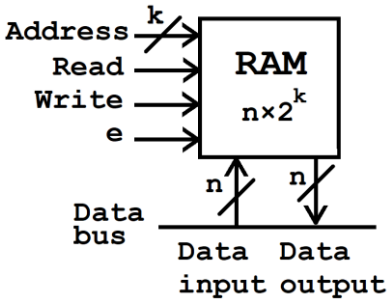
The memory is a collection of cells, each one having capability of storing information for later retrievement.
The maximum size of the memory that can be used in any computer is determined by the addressing scheme. An ***address bus*** made of ***n bits*** can identify up to 2^n ***different addresses***.

Useful measure of the speed of memory units:
- ***memory access time*** is the time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation.
- ***memory cycle time*** is the minimum time delay required between the initiation of two successive memory operations, for example, the time between two successive read operations.

The cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit.

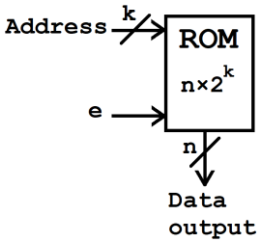
• *RAM*

Random Access Memory (RAM) is a particular implementation of memory characterized by an *equal access time for any cell inside the RAM* (the delay is independent of the position).
RAM memory can be both read and written.



The `data-in` and `data-out` here are represented as two physical opposite connections, but they can also be indicated as a bidirectional logical connection.

● *ROM*



Read-only memory (ROM) contains a permanent pattern of *data that cannot be changed*. ROM is non-volatile: no power source is required to maintain the bit values in memory. Therefore, ROM is cheaper, faster and uses less energy than RAM.

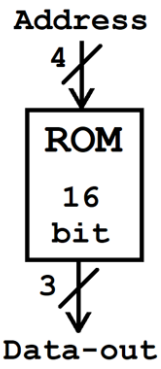
While it is possible to read a ROM, it is not possible to write new data into it, apart from when the manufacturer programs it definitely during the manufacturing process.

► *Use ROM as an adder*

By using the ROM as we did with the MUX, it can be implemented as an adder.
The unique "input" to the ROM is the address, which choses the memory cell from which to read the data.
By using a 4-bit address bus, we can point up to $2^4=16$ memory cells. If we use set the addresses as A and B, and the previously stored data in the corresponding cell as the sum S, the ROM acts as an adder and it outputs the 3-bit S.
For example:
- `A=00 , B=00` → address `0000` → data-out `000`
- `A=00 , B=01` → address `0001` → data-out `001`
- `A=10 , B=11` → address `1011` → data-out `101`

It is easier to design and use than the MUX; it is convenient for little data obviously.

A	B	S
00	00	000
00	01	001
00	10	010
00	11	011
01	00	100
01	01	101
01	10	110
01	11	111
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	111

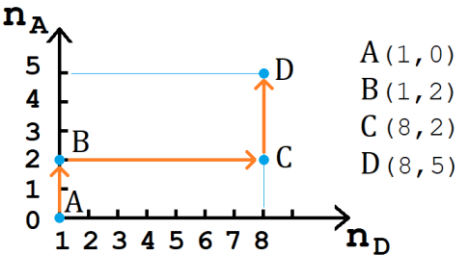


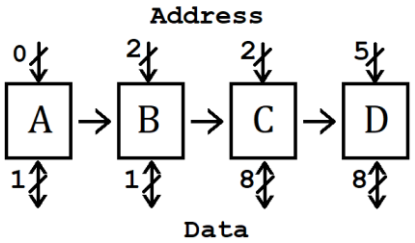
● *Memory organisation*

Single-bit memory is implemented using D flip-flop: it is accessed through 1-bit data bus and does ***not*** need an address bus (there is only 1 bit available, so only that cell is read or written).

How to switch from single-bit to multiple-bit memory? We use a *hierarchical approach* (= memory designed before). There are two ways:
- increase the *number of memory cells*
- increase the *parallelism*

The *address* is directly proportional to the number of memory cells; *data* is related to parallelism capability of each cell to store one or more data bits.





In the graph:
- n_A indicates the number of bits of the address bus
- n_D indicates the number of bits of the data bus

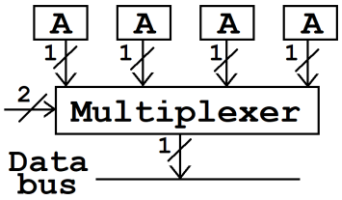
n_A starts from 0 because $n_D = 2^{n_A} = 2^0 = 1$
We proceed as following:
- start from the basic cell (origin) **A**`(1,0)`: n_D=1, n_A=0
- we move to **B**`(1,2)`: n_D=1, n_A=2
- we move to **C**`(8,2)`: n_D=8, n_A=2
- we move to **D**`(8,5)`: n_D=8, n_A=5

By moving, we increase the bits of the buses and therefore the capability of the cell to store data: we first increased the number of addressable cells and then parallelised them.
The solution to switch from single to multiple-bit memory is to combine both ways (increase the number of cells and the parallelism), in the preferred order.

Note: during 2020 lectures this notation was used instead. The approach is the same, only numbers change.
Block A → Block α Block B → Block β
Block C → Block δ Block D → Block η
Address case α → case A Address case β → case B

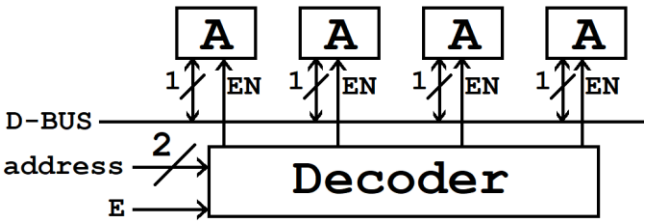
We proceed as following:
Block α`(1,0)`: n_D=1, n_A=0
Block β`(1,3)`: n_D=1, n_A=3 (made of 8 blocks α + decoder)
Block δ`(8,3)`: n_D=8, n_A=3 (made of 8 blocks β, in parallel)
Block β`(8,5)`: n_D=8, n_A=5 (made of 4 blocks δ + decoder)

► *Starting from 1 cell of 1 bit (block* A*), build a memory of 4 cells of 1 bit each (block* B*)*
block A: $n_D=1, n_A=0$ → block B: $n_D=1, n_A=2$



Let us consider a 1-bit memory cell, block A, and replicate it to have four cells.
By linking the 2-bit address line with a multiplexer we can indicate 4 cells and select from which to read.
This architecture an only be suitable to a ROM:
- between MUX and data bus only 1 bit can be exchanged per time
- ***we cannot write*** to the MUX (it cannot be reversed) and therefore to the cells!

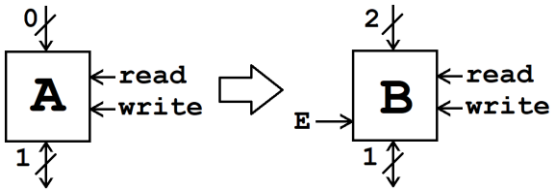
The only solution is to have an internal 1-bit bus.
Problems:
- we need a control bus to manage the communication
- a selection mechanism is needed to choose from which cell to read or write; if missing, the resulting signal would be the sum of all memory cells signals.



A *hard enable* (EN) signal allows the block to work or to be electrically disconnected and it is managed by a decoder with 2-bit address bus (A-BUS). Let's assign the hard enable as a property of each A block: the decoder sends a true or false and each A block acts accordingly.

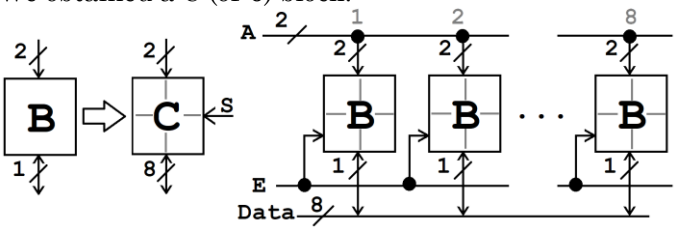
The decoder has a *soft enable* (E): its disable sets all outputs to 0, so no output is sent.
write and read control signals are delivered in parallel to all cells; information is sent through the data bus (D-BUS).

We obtained a B block.



► *Using eight 4-bit blocks B, build a memory (block* C*) that can write and read from all in parallel*
Each block B has its separated address line, so we connect all of them in parallel to have a common address (there is not a selection / *competition*, all the blocks ***collaborate*** together because in parallel).
At a time each block stores 1 bit, from MSB (in the first block) to LSB (in the last block).
The *internal position* where the value is stored/read is the same for all blocks (i.e. the higher left cell).

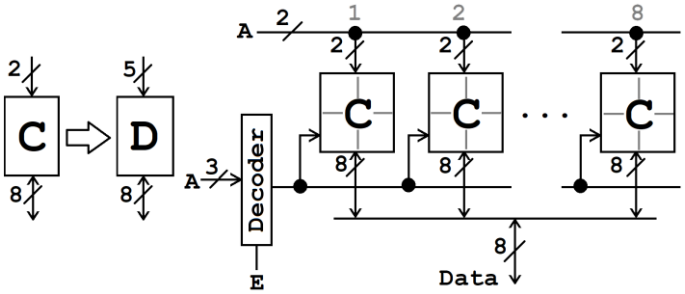
We obtained a C (or δ) block.



write and read control signals, and from now E as well, are not present in the schematic because already included in the previous blocks.
All blocks are *enabled together* without any problem in respect to the D-BUS because they collaborate (and not in competition).

► *Using eight C blocks, build a memory that can access each of them one at a time.*
To access each of the 8 C blocks at a time we need a $\log_2 8$= 3-bit A-BUS. We need to increase the size of the A-BUS (from 2 to 5 bits) while keeping the same data parallelism (8 bits).
Each C block receives 2 bits of address and provides 8 bits of data. We aim to have a final data signal of 8 bit, so we need to be in *competition* mode: a decoder allows selecting a block from which to retrieve the data.

We obtained a D (or η) block.



Because of the *hard-enable* feature for the competition mode, we collect the data input/output of all C blocks with an *"internal" short bus* and then transmit it to the actual D-BUS. (C blocks have a soft enable, the hard enable is in the A blocks)
The A-BUS for connecting the D block has 5 bits:
- 2 bits link in parallel all the C blocks (and therefore the B blocks inside them), as seen previously;
- 3 bits go to the decoder to choose a C block to enable.

The digits of the A-BUS have two possible organisations (based on which order they are connected, from MSB to LSB):
- *Case α* 2 (C blocks) + 3 (decoder)
- *Case β* 3 (decoder) + 2 (C blocks)

Memory cells addresses are processed through one or more decoders, according to two organizations:

- ***Alpha***, which establish that the address to be decoded is located on the lower part of the string of bits on the address bus, so that, for different addresses, all the *values on the higher part are put on different memory blocks*, which may contain multiple cells.
- ***Beta***, which establish that the address to be decoded is located on the higher part of the string of bits on the address bus, so that, for the same address `000`, all the *values on the lower part are put on the same memory block*, which may contain multiple cells.

Case β – 3 higher bits + 2 lower bits
The 1-bit cells of the array have these addresses:

0 → **000**00 1st instruction
1 → **000**01 2nd instruction
2 → **000**10 3rd instruction
3 → **000**11 …
00100 → 2nd block is selected

0	1
2	3

4	5
6	7

8	9
10	11

. . .

Case α – 2 higher bits + 3 lower bits
The 1 bit cells of the array have these addresses:

0 → **00**000 1st instruction
1 → **00**001 2nd instruction
2 → **00**010 3rd instruction
3 → **00**011 …

The first two address bits point to the position, the last three to the block.

0	8
16	24

1	9
17	25

2	10
18	26

. . .

7	16
23	31

Which one is better? It depends.
We must compare advantages and drawbacks for each solution (they are complementary of each other).

Reliability can be defined as the probability that a system will produce correct outputs up to some given time. Reliability is enhanced by features that help to avoid, detect and repair hardware faults. A reliable system does not silently continue and deliver results that include uncorrected corrupted data. Instead, it detects and, if possible, corrects the corruption.
Resilience indicates the capability of fast recovery from a degraded system state.

Because of the disposition of addresses:

- *α* is ***faster*** but *not reliable* (if a cell has a failure and any information is lost, everything is likely to be compromised because every data is distributed between all blocks)
- *β* is much more ***reliable*** and ***resilient*** but slower (if some block is broken at the extremities, the information may not be compromised because information is distributed block by block)

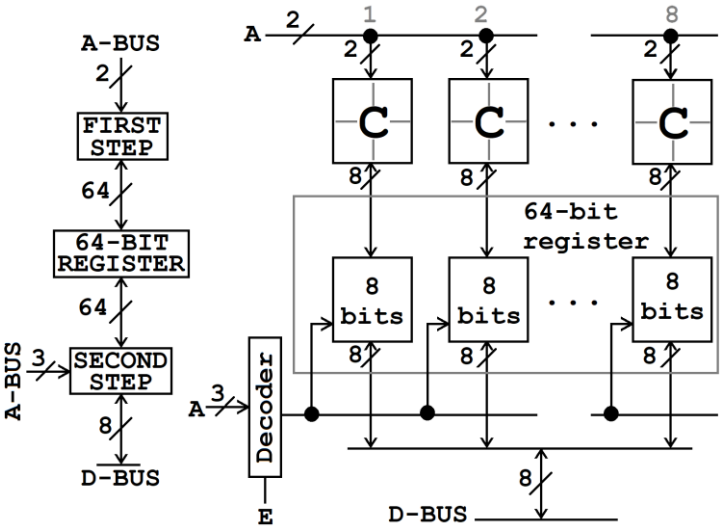
Why α **could be faster**?
Consider the *theoretical parallelism* by comparing the time to transfer a program of 8 instructions:
- *α* : needs 1 cycle to get all the 8 (parallel access)
- *β* : needs 8 cycles, one for each block (serial access)

A computer is organised in several boards and separate chips, to enhance at most the parallelism advantages. It will be still working with some empty memory slots.

How should be α organised to have the most advantages from parallelism?
By increasing the data bus there is collaboration, by increasing the address bus there is competition.

To read/write in parallel 64 bits (therefore from/to all the blocks together) we upgrade the data bus from 8 to 64 bits: the action is done in parallel now and a decoder is no more needed.
There is no more competition but now the selection is only partial, not full as before. The operation of selecting the information is split in two cascaded steps:

1. we temporarily don't care of the low address bits (previously sent to the decoder); the 2 higher bits are used to send in parallel the address to all the blocks. The data retrieved from all the blocks together is sent in parallel to the register.
2. Data is loaded to a 64-bit register.
3. The selection of the data from one out of the 8 blocks (so 8 bits on 64) is driven by the 3 lower bits of the A-BUS through a decoder.



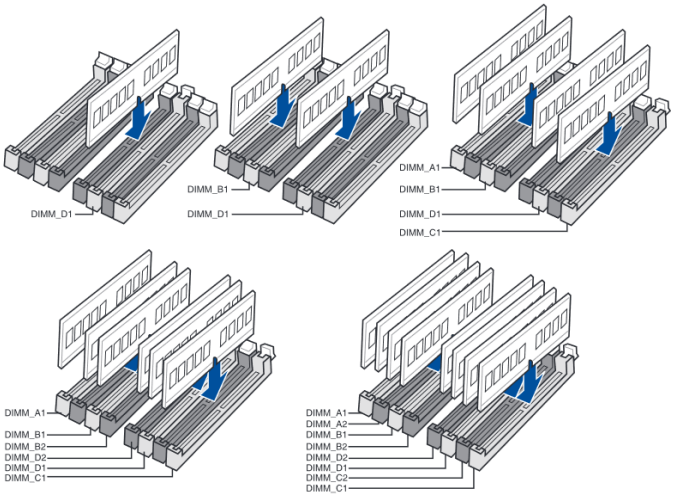
This architecture loads or unloads in parallel all the block of such design: this is useful to understand how ***cache memory*** (the register in our schematic) collaborates and interacts with RAM.
Therefore, cache memory has a case *α* organisation.

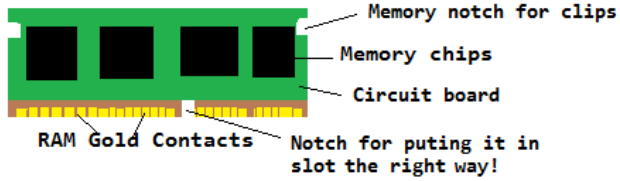
► *Example*
A 32-bit architecture is able to address 2^{32} cells, which correspond to $4 \times 10^9 = 4$ G cells.
Let us consider that each cell corresponds to 1 byte, therefore the maximum memory is 4 GB.

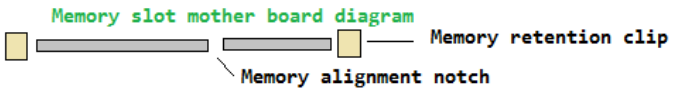
The computer has four memory slots.
Do we need 1GB of memory installed per slot? No, we can have several combinations.

4GB	0	0	0
2GB	0	0	0
1GB	1GB	0	0

What are the allowed combinations? They are reported in the datasheet of each board. Some examples:
- the bigger memory must be in the first slot
- they have to be put in pairs







Some slots are empty: for this to be allowed, the organisation of slots is of β type. To take advantage of the parallelism, we would need to have α addressing: the solution is to use hybrid type.

The 32-bit ***full address*** has the decoder first and the internal address after, according to β.

Decoder	*Internal address to card memory*

The internal address is organised according to α, so that there is the parallelism enhancement.

Internal chip addressing	*Card decoder*

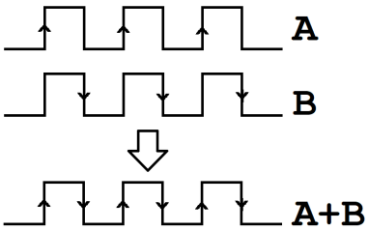
All the memory chips are equal: they are soldered to the RAM board, therefore the RAM layout is fixed. The internal card decoder selects which chip to use.

The card decoder is removed because chips are addressed with α mode: we differ the selection to a later stage, so that the memory is faster.

Each memory card has several chips and the entire memory is distributed along all of them.

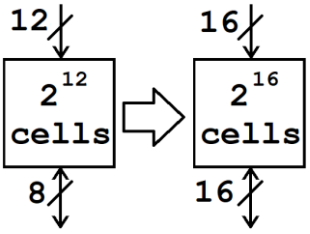
Because the chips are organised in α mode, if a chip fails the memory stick becomes unusable.

Some motherboards required memory installed in pairs, to have parallel access and higher speed.
Explanation:
each memory responds to edge level signal.
Memory on side A responds when low to high edge.
Memory on side B responds when high to low edge.
Therefore, the delay is halved by putting them together: the request is distributed between both.



► *Example – What is the **size of the decoder** used to design a memory of 2^{16} cells (output 16 bits) using blocks of 2^{12} cells (output 8 bits)?*
The A-BUS size to address 2^{12} cells is 12 bits, therefore we must increase it of 4 to be able to address 2^{16} cells. The 4 more bits of the A-BUS go through the decoder and it will manage the $2^{16} / 2^{12} = 2^4 = 16$ blocks of 2^{12} cells: so the size of the decoder is 4-16.

The decoder does not impact the data output size.

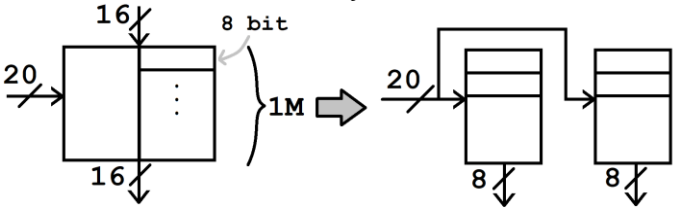


What is the size of the decoder used to design a memory of 2^{10} cells (output 8 bits) using blocks of 2^{14} cells (output 8 bits)?
A decoder would be useless! We would use a big block with limited resources.

● *Memory bank*

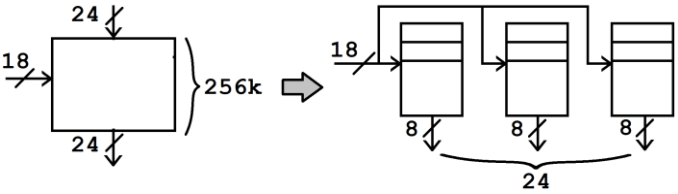
A memory bank is a logical unit of storage that consists of multiple rows and columns of storage units, across several chips. A single read or write operation accesses only one bank, therefore the number of bits in a column or a row, per bank and per chip, equals the memory bus width in bits (single channel). The number of bits in a column and a row, per chip, multiplied by the number of chips in a bank determines the size of a bank.

► *Make a 1M×16-bit memory with 1M×8-bit memories*



► *Make a 256k×24-bit memory with 256k×8-bit memories*

$1k = 2^{10} cells \Rightarrow \log_2 256k = 18$ bit address bus



► *Make a 1M×8-bit memory with 256k×8-bit memories*

1M×8-bit mem address bus: $\log_2 1M = \log_2 2^{20} = 20$ bit
256k×8-bit mem address bus: $\log_2 256k = 18$ bit

1M / 256k = 4 → four 256k memories are needed

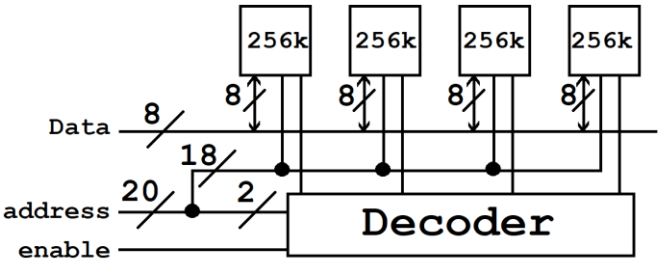
How to connect them together?
1M memory has a 20-bit address; 256k memory has 18-bit address: the first two bits of any address in 1M memory identifies to which block of 256k memory data belongs, the remaining 18 bits identify the internal address of such blocks.

1st memory	*2nd memory*
0000000000000000000	**01**00000000000000000
0000000000000000001	**01**00000000000000001
0000000000000000010	**01**00000000000000010
0000000000000000011	**01**00000000000000011
00...	**01**...
0011111111111111111	**01**11111111111111111

3rd memory	*4th memory*
1000000000000000000	**11**00000000000000000
1000000000000000001	**11**00000000000000001
1000000000000000010	**11**00000000000000010
1000000000000000011	**11**00000000000000011
10...	**11**...
1011111111111111111	**11**11111111111111111

Therefore, a 2-bit decoder can link the four 256k blocks and choose to which sub-memory send the data.

The decoder manages the enable signal, so according to the input it will send a bit 1 to the chosen memory and a bit 0 to all the others (and disable them).
Data input has size of 8 bit and it is linked to each block in parallel.



► *Make a 1K×8-bit memory with 512×8-bit memories*

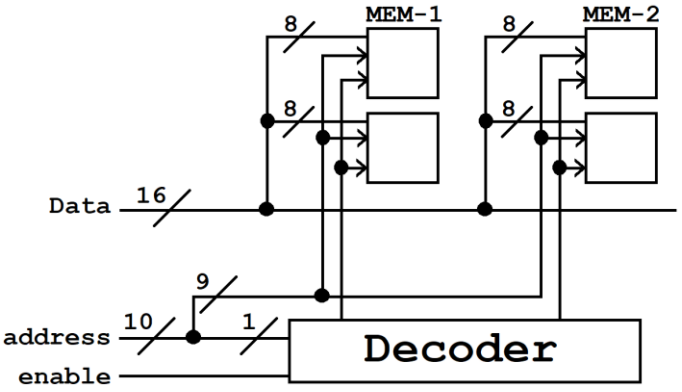
1K×8-bit mem address bus: $\log_2 1K = \log_2 2^{10} = 10$ bit
512×8-bit mem address bus: $\log_2 512 = \log_2 2^9 = 9$ bit

1K / 512 = 2 → two 512 memories are needed

How to connect them together?
1K memory has a 10-bit address; 512 memory has 9-bit address: the first bit of any address in 1K memory identifies to which block of 512 memory data belongs, the remaining 9 bits identify the internal address of such blocks.

1st memory	*2nd memory*
0000000000	**1**000000000
0000000001	**1**000000001
0000000010	**1**000000010
0000000011	**1**000000011
0...	**1**...
0111111111	**1**111111111

Therefore, a 1-bit decoder can link the two 512 blocks and choose to which sub-memory send the data.



● *Static and Dynamic Memories*

We can distinguish between:

- ***Static Memories***
  Advantages: very fast
  Drawbacks: high cost (area, power consumption), lower capacity with respect to dynamic memories.
  The basic elements are transistors (*D flip-flops*); static memories are able to maintain information as long as power is applied (they are volatile).
  They are often used as cache memory.
- ***Dynamic Memories***
  Advantages: cheaper than static memories
  Drawbacks: slower than static memories
  The basic elements are capacitors; dynamic memories are able to maintain information as long as capacitors' charge is restored frequently (they are volatile).

How is memory read?
In DRAM, a voltage higher than a given threshold corresponds to the logical 1 (there is charge "stored" into the capacitor), otherwise to the logical 0 (no charge).
To check if the capacitor is charged or discharged, we measure the voltage at its ends: by doing so we use the charge and electrons which were previously stored in the capacitor itself.
Therefore, reading is a destructive operation:
- if there was a charge (=1), we restore it;
- if there was no charge (=0), we leave it empty.

This behaviour explains why such memory is slow: a logical reading is composed by a real reading followed by a restoring step, which means a physical writing.

Because capacitors are real and not ideal, electrons leak out of them. In a matter of a few milliseconds a full bucket becomes empty. Therefore, for dynamic memory to work, either the CPU or the memory controller has to come along and recharge all of the capacitors holding a 1 before they discharge. This ***refresh*** operation (the periodic restoration of information because of leakages of the capacitor) happens automatically thousands of times per second. To do this, the memory controller reads the memory and then writes it right back: while this *periodic maintenance* is taken the dynamic memory cannot be used.

In static memory, a form of flip-flop holds each bit of memory. A flip-flop for a memory cell takes 4 or 6 transistors along with some wiring, but never has to be refreshed. This makes static memory significantly faster than dynamic memory. However, because it has more parts, a static memory cell takes a lot more space on a chip than a dynamic memory cell. Therefore, you get less memory per chip, and that makes static memory a lot more expensive.

● *Value Certification and Validity*

How can one be sure of the validity of bits contained in a memory cell? Is it valid because a programmer wrote into that cell? In computer engineering each data, circuit etc. needs to be certified.
Self-certification is rarely accepted, so other certification methods (third-part certification) have to be developed or requested.

We can understand if a value is valid by analysing:

- its range (carry flag)
- its final result

● *Error codes*

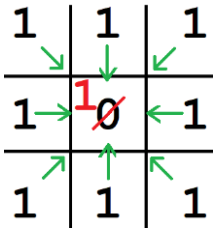
Semiconductor memory systems are subject to errors, which can be categorized as hard failures and soft errors.

A ***hard failure*** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear.

A ***soft error*** is a random, non-destructive event that alters the contents of one or more memory cells without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials.

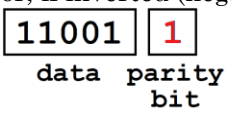
Both hard and soft errors are clearly undesirable and most modern main memory systems include logic for both detecting and correcting errors.
The dynamic physical interaction of the electrical signals affecting the data path of a memory unit may cause occasional errors in storing and retrieving the binary information.

For example: if a cell has value 0 and all the cells around it have value 1, it is very likely that the proximity and reciprocal influence will make the internal cell result with 1 as a value. This is not a permanent error, but we cannot prevent it from happening.

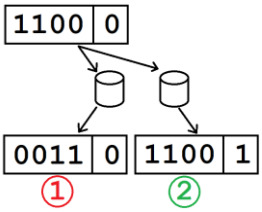


The reliability of a memory unit may be improved by employing ***error-detecting*** and ***error-correcting codes***.

The most common error detection scheme is the parity bit. A ***parity bit*** is generated and stored along with the data word in memory; the parity bit can be ***even*** or, if inverted (negated) ***odd***, and vice-versa.



In case of even parity bit, the number of ones in the data and in the parity bit must be even: in the example there are 3+1=4 ones, which is even.
The parity bit is assigned to each cell according to the values in it: if there were 4 ones in data, the parity bit would have been 0.
Such data is now stored in the memory. The parity of the word is checked after reading it from memory:
- the data word is accepted if the parity of the bits read out is correct (if the parity mode was set as even, the system checks that the number of ones is even).
- if the parity checked results in an inversion (it is found that the number of ones does not respect the mode set previously), *an error is detected, but it cannot be corrected*.



The data 1100 0 is written to two memories, and when read it gives two different values:
1) 0011 0 – the parity bit was even, therefore there is no error! Knowing the original value, we know there is an error anyway: the parity bit can spot only 1, 3, 5, 7… errors (so an odd number of errors), here there were 4 errors and therefore it cannot know they happened.
2) 1100 1 – the parity bit was even, therefore there is an error! The parity bit as soon as it is computed and stored becomes an "information" exactly as data: the parity bit can only spot the existence of an error but not its position, therefore *it does not provide any support to correct the error*.

We can say that the parity bit has:
- Advantages: very *small cost* (*), easy to compute (therefore *fast*);
- Drawbacks: is a *simple* and *weak* error-detecting system (it does not correct neither help to correct the errors; it does not spot even number of errors)

An error-correcting code generates multiple parity check bits that are stored with the data word in memory. Each check bit is a parity over a group of bits in the data word. When the word is read back from memory, the associated parity bits are also read from memory and compared with a new set of check bits generated from the data that have been read.
- If the check bits are correct, no error has occurred.
- If the check bits do not match the stored parity, they generate a unique pattern, called a ***syndrome***, that can be used to identify the bit that is in error. A single error occurs when a bit changes in value from 1 to 0 or from 0 to 1 during the write or read operation. If the specific bit in error is identified, then the error can be corrected by complementing the erroneous bit.

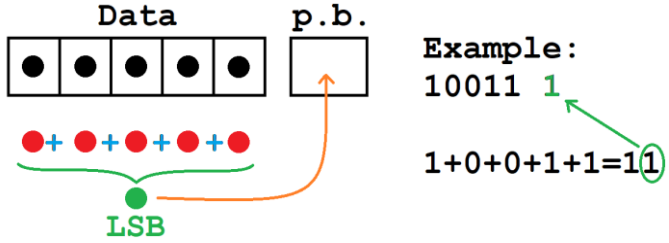
The *probability of errors* happening is:
- p for single error
- $\sim p^2$ for double error

If p is very small, it is very unlikely to get a double error and even more for further errors.
When the parity bit says that the data and p.b. are:
- wrong, we are absolutely sure of it;
- correct, we cannot be 100% sure of it (the trustworthy cases are $1–p^2$, the uncertainty starts from the double error case which cannot be spotted)

The cost of using the p.b. codes correspond to the cost of the additional memory needed to store the p.b. itself, which means 1 bit more every n bits of data.

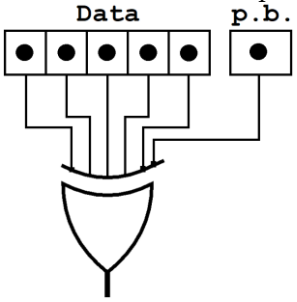
$n \rightarrow n+1$, there is an increase of $1/n$ for each n bits
To minimise the cost, n should be as large as possible
How to maximise the "power" of the p.b. to help spotting errors (depending on n)? n should be small.
Therefore, it is always a trade-off (hybrid) between cost and power; it depends on the probability of the error, which type of data is protected and many more issues.

►*Circuit to compute the parity bit (even) and check*
Instead of counting the number of ones, we can add all the bits together and compute the result on a single bit, which corresponds to their sum's LSB and it will be set as parity bit.

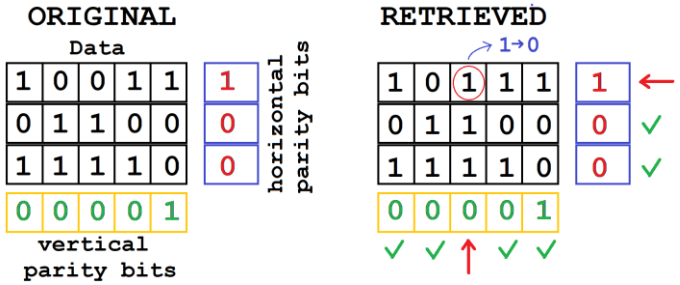


The p.b. is computed using a XOR with all of the data bits as single inputs and the LSB / p.b. as output.

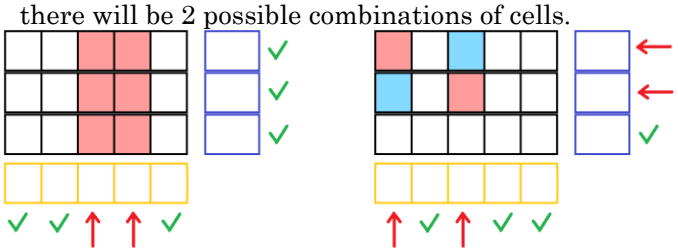
The circuit for the detection of a single error is implemented again with a XOR:
- output is 0, there has been (likely) no error
- output is 1, there has been an error



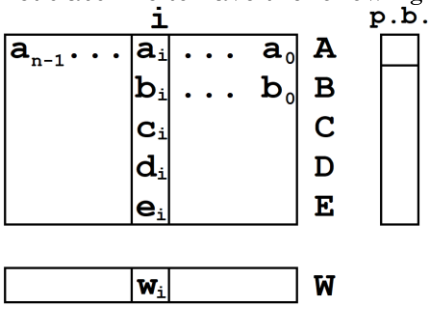
To find the position of the error, we can create a grid to store data and compute both ***horizontal and vertical parity bits***; this will cost more as space and time.



In the case of ***1 error***, it is correctly detected, found and ***corrected***; if there are ***2 or more errors*** they can be ***detected***, but this approach cannot find their positions.
For example:
- in case 2 errors are on the same row, we can identify the columns but cannot know which row they reside, because all horizontal parity bits will have correct values;
- in case 2 errors are on different rows and columns, there will be 2 possible combinations of cells.



Let's assume to have the following data block:



$$w_i = a_i \oplus b_i \oplus c_i \oplus d_i \oplus e_i$$

Let us assume that the third row C is replaced by F: (C) → (F)

$$w_i' = a_i \oplus b_i \oplus f_i \oplus d_i \oplus e_i$$

The writing of a word in memory (from C to F) requires updating W to W', which requires 5 readings + 1 writing.
How to reduce the number of readings? By removing the contribution of C from W, then adding F. By computing $w_i \oplus c_i$, we remove c_i from w_i.

$$w_i \oplus c_i = a_i \oplus b_i \oplus \not{c}_i \oplus d_i \oplus e_i \oplus \not{c}_i$$

$$\Rightarrow w_i' = w_i \oplus c_i \oplus f_i$$

Writing of a word requires reading of the old value previously stored in vertical parity (W) and in the row (C) and the writing of the new row (F) and the new w_i' computed as $w_i' = w_i \oplus c_i \oplus f_i$.

One of the most common error-correcting codes used in RAMs was devised by R. W. Hamming. In the ***Hamming code***, k parity bits are added to an n-bit data word, forming a new word of $n + k$ bits.

► *Note: the following part is not requested for the exam* ◄
The bit positions are numbered in sequence from 1 to $n + k$. Those positions numbered as a power of 2 are reserved for the parity bits. The remaining bits are the data bits. The Hamming code can be used for data words of any length.

In general, the Hamming code consists of k *check bits* and n *data bits*, for a total of $n + k$ bits. The syndrome value C consists of k bits and has a range of 2^k values between 0 and $2^k - 1$. One of these values, usually zero, is used to indicate that no error was detected, leaving $2^k - 1$ values to indicate which of the $n + k$ bits was in error. Each of these $2^k - 1$ values can be used to uniquely describe a bit in error. Therefore, the range of k must be equal to or greater than $n + k$, giving the relationship $2^k - 1 >= n + k$.
Solving for n in terms of k, we obtain $2^k - 1 - k >= n$.
This relationship gives a formula for establishing the number of data bits that can be used in conjunction with k check bits.

For example, when $k = 3$, the number of data bits that can be used is n … $(2^3 - 1 - 3) = 4$. For $k = 4$, we have $2^4 - 1 - 4 = 11$, giving n … 11. The data word may be less than 11 bits, but must have at least 5 bits; otherwise, only 3 check bits will be needed.

Data on CDs is still readable when there are radial scratches (that go from the centre to the borders). The error codes of a CD are different from the ones of hard disks (HD): CDs are physically exposed to destructive actions from the real world; HDs are packed and the surface of the plates is less exposed. When errors start happening on an HD, the medium is likely to be soon unusable: HD plates are sealed inside a box, so if scratches happen, they will produce dust, which will circulate in that closed environment and keep doing damage to other plates. Circular scratches on CDs or HD plates cause important errors, therefore to "clean" the CDs a radial motion is used.

CD data is written from the centre to the borders.
CD data is associated to ***strong*** *error correcting codes* because of the high risk of damaging its surface. In the HD this is not done because of the sealed space. When the scratches are too much or too much wide, data cannot be recovered.

A magnetic head reads and writes on the plates of the HD: because this is done with a rotation motion, bumps that make the head touch the plates will cause circular scratches; no code is strong enough to correct this type of errors, so only error detection codes are needed in this case.

CRC (***cyclic redundancy check***) is an error-detecting code known as *majority voting*: it corrects any bit according to a comparison between the original string of bits and two replicas, that were stored together with the original (several copies of the same information are stored and they are kept aligned in time). A *bad CRC alert* means that the HD is close to die.

There are 3 copies of the same data: the size occupied by the actual data is just 1/3 of the total, and the remaining 2/3 are the copies.

Data	Data Copy 1	Data Copy 2

Example: if the bit to be checked is equal to 1 both in original and in replica #2, then it's assumed that the correct bit value was 1.

101	001	011

101 data
001 1st replica
011 2nd replica
001 final corrected value

A continuous backup done with mirroring will increase the reliability of the system.
If the error is present in more copies, it may be considered as "correct data" and the correct data as "error": only by increasing backup copies we can insure the correctness of data and reliability.

● Associative memory

In *conventional memories* (ex. RAM) the user supplies a memory address and the RAM returns the data word stored at that address (*search by address*).
Instead, associative memory (also called ***content-addressable memory***) compares input search data (***tag***) against a table of stored data and returns the address of matching data; this is called *search by tag*.
This is equivalent to a loop, because we use some contents to search for some other partial contents.

Each cell should store two contents: *properties of contents* and *"real" contents*.

Properties of contents	"Real" contents

The first are used to search and find the wanted elements; the latter are the results of such search. The search query is *compared* to each properties of content, scanning through all the memory: the feedback is *not found* or *found + content position*.

Searching can be *extremely slow* if extra hardware is not provided to help the parsing: for each key corresponds a *comparator*, so that all of them can work in parallel. If the found results are more than one, a *priority encoder* is needed to return the *address of the cell* to be taken ordered from most to least significant.

Associative memory is *expensive*: the "search hardware" made of priority encoders and comparators rise its price but they are needed to have a system efficient in time.

Associative memory

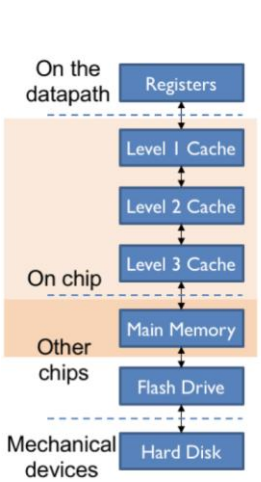
Conventional memory		Search hardware
	↔	
	↔	
	↔	

● *Memory types and memory hierarchy*

The performance of most modern computers is limited by the bandwidth of the connection between the CPU and main memory, the so-called memory bottleneck. The fundamental trade-off in current memory technologies is the following: *as the memory's capacity increases, so does it access time*.

It takes some architectural cleverness to build a memory system that has a large capacity and a small average access time. The cleverness is embodied in the cache, a hardware subsystem that lives between the CPU and main memory. Modern CPUs have several levels of cache, where the modest-capacity first level has an access time close to that of the CPU, and higher levels of cache have slower access times but larger capacities.

CPU's registers have the *highest performance and endurance*, but these advantages correspond to a *higher cost per bit*; moving down to slower memories the *cost* and *frequency of access are reduced*, while *capacity*, *latency* and *persistence get higher*.



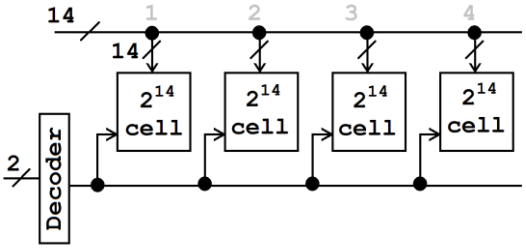
Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

Registers, cache and main memory are *inboard memory elements*; magnetic disks, CDs, DVDs and blu-rays are *outboard storage*.

► *Test 2020-02-10 – Exercise 4*

We have a memory of $\mathbf{2^{16}}$ cells organized in banks of $\mathbf{2^{14}}$ cells. The decoder used to select the proper bank is:
A) 2-to-1
B) No decoder is necessary, but we need an encoder
C) 1-to-2
D) None of the previous answers because…
Solution: D

To obtain $\mathbf{2^{16}}$ memory we use $2^{16}/2^{14} = \mathbf{4}$ blocks of $\mathbf{2^{14}}$ cells. We need a 2-to-4 decoder, with the 2 missing bits of the address bus entering to the decoder.



► *Test 2019-09-19 – Exercises 1, 3, 4*

Exercise 1

A carry lookahead adder on 8 bits
A) Is, very likely, not worth to be implemented
B) Requires 8 cascaded full adders in order to work
C) Is a sequential circuit
D) None of the previous answers
Solution: D

A is wrong because 8-bit CLA is the biggest CLA worth to be implemented.
B is wrong because 8 cascaded FA compose an 8-bit RCA.
C is wrong because previous history is not considered, it is a combinational circuit.

Exercise 3

Sequential circuits (mark all the correct answers)
A) Include registers and memories
B) Can be modelled by the Huffman model
C) Present some outputs which are looped back to the input
D) Require combinational gates to be implemented
E) None of the previous answers
Solution: A, B, C, D

Exercise 4

The boolean function for computing the equality between two bits a and b is:
A) (a AND (NOT b)) OR (b AND (NOT a))
B) a AND b
C) Does not exist
D) None of the previous answers because…
Solution: D

1. $a \oplus b \rightarrow \begin{matrix} \mathrm{T},\ a \neq b \\ \mathrm{F},\ a = b \end{matrix}$

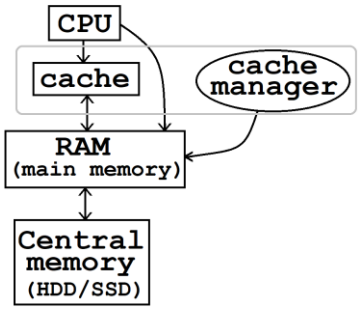
$$f : \overline{a \oplus b} = \bar{a}b + a\bar{b} = (a + \bar{b}) \cdot (\bar{a} + b) \Rightarrow a = b$$

2. $\alpha = \bar{a},\ \alpha \oplus b \rightarrow \begin{matrix} \mathrm{T},\ a = b \\ \mathrm{F},\ a \neq b \end{matrix}$

$$f : \bar{\alpha}b + \alpha\bar{b} = ab + \bar{a}\bar{b}$$

● *Cache Memory*

A cache is a relatively small fast memory interposed between a larger, slower memory (RAM) and the logic circuitry (CPU) that accesses the larger memory. The cache holds recently accessed data, and is designed to *speed up subsequent access to the same data*.



The CPU ***fetches*** (reads from memory) an instruction, ***decodes*** it (understands what are the instruction) and then, depending on the instruction, ***executes*** it.

Data from the central memory (HD) is stored in the RAM to be read faster; data from RAM is stored into cache to further improve the access time from CPU. Therefore, CPU "believes" that only cache memory exists and that it is as large as RAM memory.
This method, a compromise between costs and speed, allows *CPU-data access time to be the lowest possible*.

All the interactions between the CPU and the memory should be ***virtualised*** by the cache memory: each time the CPU needs some data, it asks the *cache manager* if such data is already available; if not, the request is forwarded to the main memory.
Cache content changes dynamically, according to the ***cache manager*** (or *cache controller*) commands.

A ***cache hit*** occurs whenever the CPU requested block has been found in cache, otherwise a ***cache miss*** occurs and the block of information is to be searched into main memory, read and sent to the CPU.
The average access time to the cache is t_A: with hit result is t_c; instead a miss operation and get data from the RAM is t_R, $t_R > t_c$. The management time t_G is the time needed to unload data once it has been retrieved.
The ***hit ratio*** value h indicates the proportion between hit and miss of a certain number of operations. The more it tends to 1, more is reliable the cache controller.
Average access time: $t_A = h \times t_c + (1-h) \times t_R + t_G$

► *Example*
$t_G = 0.1$ $\quad h = 1 - 10^{-6}$
$t_c = 1$ time unit $\quad t_R = 1000$ time units $= 10^3$

$t_A = (1 - 10^{-6}) \times 1 + (1 - 1 - 10^{-6}) \times 10^3 + 0.1 =$
$= 1 + 10^{-3} - 10^{-6} + 0.1 \approx 1$ time unit

Therefore, this memory is very slow.

Let us repeat the calculus with $h = 0.9$.
$t_A = 0.9 \times 1 + (1 - 0.9) \times 10^3 + 0.1 =$
$= 0.9 + 10^2 + 0.1 = 101$ time units

We have to consider three elements:
- h: the higher the better; try to maximise it.
- t_c/t_R : the lower the better ($t_c << t_R$); otherwise, there would be no advantages to implement the cache.
- t_G : is very influent; the lower the better.

t_R and t_c are defined by choosing the technology. t_G and h are driven by the architectural choices.

When ***cache is full***, the cache manager frees it:
A) *Least Recently Used*: using a counter per slot; the manager writes the time stamp of each element; when cleaning is necessary it needs to identify the minimum age. For 1000 elements there are 999 comparisons, therefore this approach is too slow.
B) *Least Frequently Used*: each element has a counter, which is updated each time it is used; the cache manager frees the slots with smallest numbers in the counter. → same delay as A
C) *Oldest in usage*: there is a time stamp for each element; no update is needed, so it is lighter than A and B, but the comparison is still a bottleneck.
D) *Randomly*: some elements are deleted to make space (there is not a full wipe of the cache); there is no need of comparison. This is a good approach in case of many slots available.

The search mechanism in cache memory needs to be fast: if not, the performance will be negatively affected.

Cache organisation

Cache is made of *m* blocks, called ***lines***, that correspond to the smallest unit of transfer between the cache and main memory.
Each line consists in a combination of:
- a *data block*, where data from the RAM is copied;
- the address **`TAG`** identifies which particular block is currently being stored; it is a portion of the main memory address.

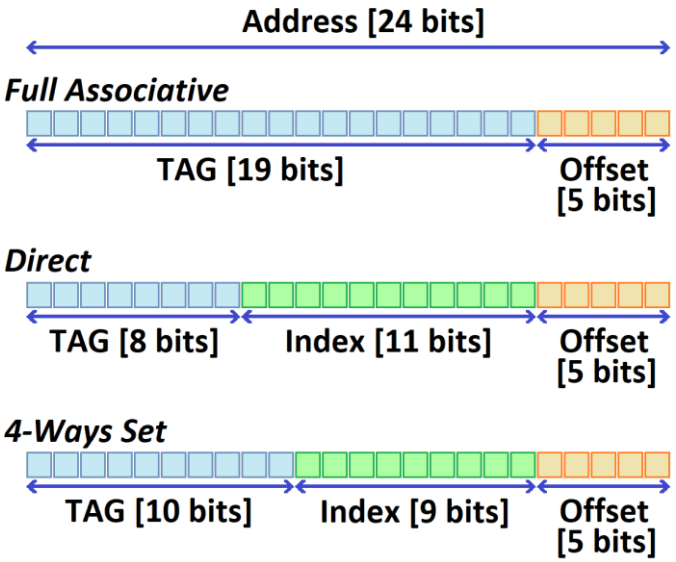
The length of a line, not including `TAG` and control bits, is the ***line size***. The line size may be as small as 32 bits, with each "word" being a single byte; in this case, the line size is 4 bytes.

When a word in a block of memory is read, it is transferred to one of the lines of the cache. There are many more memory locations than cache lines, so many addresses of the main memory are mapped to the same cache line and the cache will only be able to hold the data for one of those addresses at a time: the choice of the ***mapping function*** dictates how the cache is organized and therefore which main memory block currently occupies a cache line.

Each ***cache address*** may store different parts of the corresponding main memory location address.
When a program performs a lookup, it checks for:
- **`TAG`** identifies which *memory block* mapped into this cache position (or ***set***) is currently resident in the cache, so it is used for a comparison check;
- ***index*** (or *word*) locates the corresponding *cache line* (or ***entry***)
- ***offset*** determines which ***data*** (or ***word***) is retrieved from the cache line's data block.

Cache

SET 0	Entry 00	Word 0 Word 1
	Entry 01	Word 0 Word 1
SET 1	Entry 10	Word 0 Word 1
	Entry 11	Word 0 Word 1



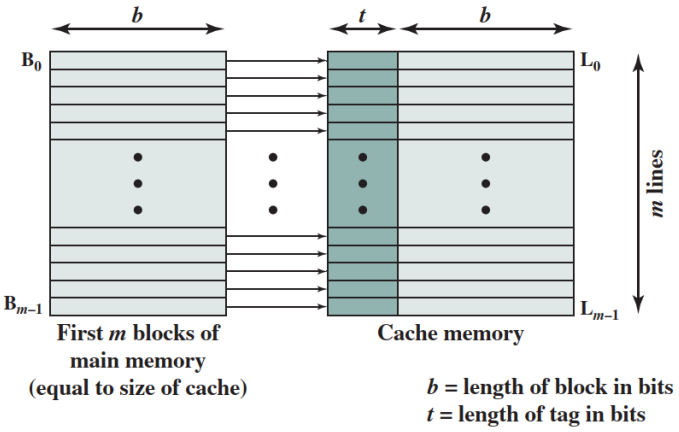
• *Cache Address Mapping*

A ***memory map*** is a structure of data that indicates how main memory is laid out into cache: the memory management unit maps the addresses from logical to physical space.

Three mapping techniques can be used:
- *direct mapping*
- *full associative mapping*
- *k-ways set associative mapping*

• *Direct Mapping*

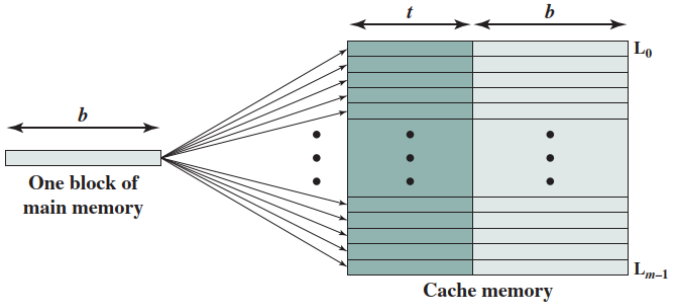
Each main memory block is associated with only one possible cache line by the equation $i = j \ modulo \ m$
i = cache line to be selected,
j = main memory block number
m = total number of lines of the cache.



DM is simple and inexpensive to implement.
Drawback: there is a fixed cache location for any given block: if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio *h* will be low (rewriting many times the same line is known as ***thrashing***).

• *Full Associative Mapping*

It overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache. The main memory block address organization is the same as the one used for DM, except for *block* (or *index*) bits, which are added to `TAG` ones, thus increasing bits assigned to `TAG`.



FAM gives flexibility as to which block to replace when a new block is read into the cache.
Disadvantage: a complex circuitry is required to examine the `TAG`s of all cache lines in parallel.

• ***Set Associative Mapping***
Cache lines are grouped into ***sets***, thus providing what is called a ***k-way set associative*** cache.

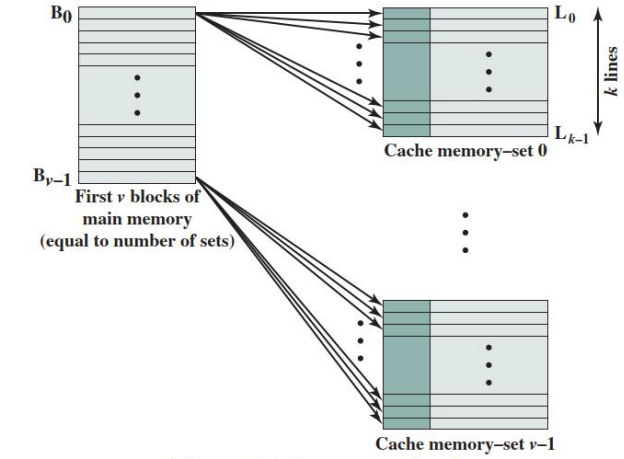
m = number of lines in the cache v = number of sets
k ($=2^x$) is the number of lines/entries in each set

The relationships are: $m = v \times k$ $i = j$ *modulo* v

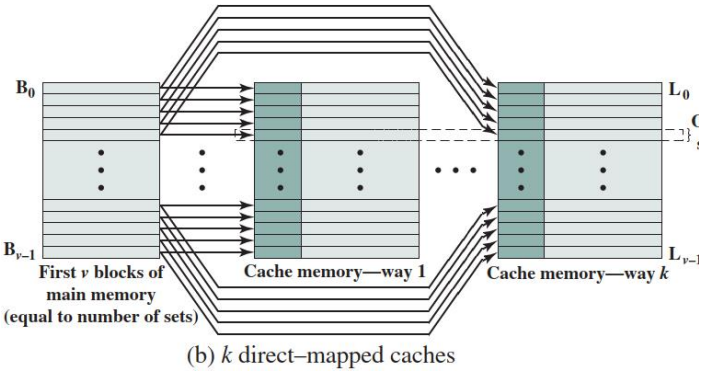
This approach gets the best from DM and FAM:
- using DM equation to decide in which set to store that main memory block;
- using k lines per set in order to partially solve the memory contention problem.

Set-associative cache can be physically implemented as v associative caches or as k direct mapping caches.

The `TAG` in a memory address in SAM is much smaller than in FAM and is only compared to the k `TAG`s within a single set, consequently requiring a cheaper comparator.



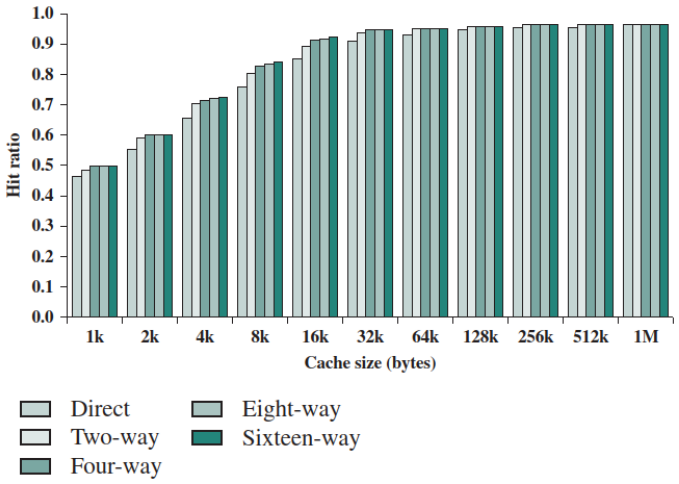
(a) v associative–mapped caches

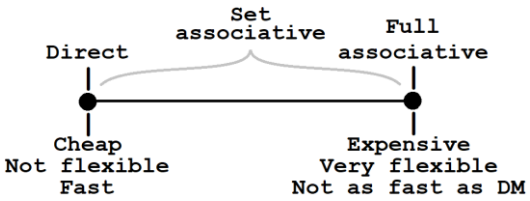


(b) k direct–mapped caches

Extreme cases of set-associative technique:
- $v = m$, $k = 1$, it reduces to ***direct mapping***
- $v = 1$, $k = m$, it reduces to ***full associative mapping***.

The use of two lines per set ($v = m/2$, $k = 2$) is the most common set-associative organization and it significantly improves the hit ratio over direct mapping.
Four-way set associative ($v = m/4$, $k = 4$) makes a modest additional improvement for a relatively small additional cost. Further increases in the number of lines per set have little effect.





<u>Summary of address mapping techniques</u>

- *Direct mapping*
  # `Entries` = Total memory / Entry size
  Offset = $\log_2$(`Entries` size)
  Index = $\log_2$(# `Entries`)
  `TAG` = Address – Index – Offset

- *Full associative mapping*
  Offset = $\log_2$(`Entries` size)
  `TAG` = Address – Offset

- *k-ways set associative mapping*
  # `SETs` = (# `Entries`) / k
  Offset = $\log_2$(`Entries` size)
  Index = $\log_2$(# `SETs`)
  `TAG` = Address – Index – Offset

► *Example*
We have an 8-bit address bus for a cache with 4 entries and 2 words per entry.

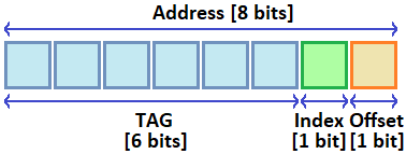
Cache

SET 0	Entry 00	Word 0 Word 1
	Entry 01	Word 0 Word 1
SET 1	Entry 10	Word 0 Word 1
	Entry 11	Word 0 Word 1

- *Full associative mapping*
  Offset = $\log_2$(`Entries` size) = $\log_2 2$ = 1 bit
  `TAG` = 8 – 1 = 7 bits
- *Direct mapping*
  Index = $\log_2$(# `Entries`) = 2 bit
  `TAG` = Address – Index – Offset = 8 – 2 – 1 = 5 bits

- *2-ways set associative mapping*
  # SETs = 2
  Index = $\log_2$(# SETs) = $\log_2 2$ = 1 bit
  TAG = Address – Index – Offset = 8 – 1 – 1 = 6 bits

2-wat set associative mapping address representation:



► *Example*
Consider a byte-addressable computer with 24-bit addresses, a cache capable of storing a total of 64Kbytes of data and blocks of 32 bytes.
Show the format of a 24-bit address for:

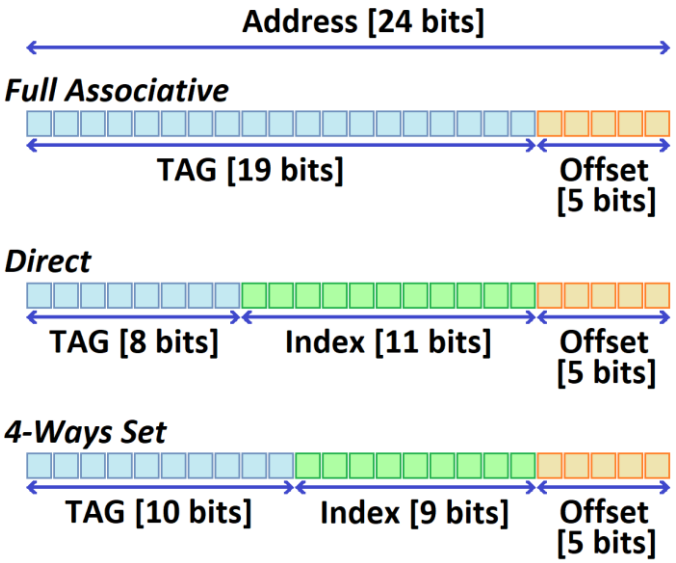
- Full associative mapping
- Direct mapping
- 4-way set associative mapping

Data / ***Word*** = 1 byte ("byte-addressable")
Address size = 24 bits
Total memory = 64 Kbytes = $2^6 \times 2^{10}$ bytes
Block / ***Entry*** / ***Cache line*** = 32 bytes = 2^5 bytes

- *Full associative mapping*
Offset = $\log_2$(Entries size) = $\log_2 2^5$ = 5 bits
TAG = Address – Offset = 24 – 5 = 19 bits

- *Direct mapping*
Entries = Total memory / Entry size =
= ($2^6 \times 2^{10}$) / 2^5 = 2^{11} entries
Offset = $\log_2$(Entries size) = $\log_2 2^5$ = 5 bits
Index = $\log_2$(# Entries) = $\log_2 2^{11}$ = 11 bits
TAG = Address – Index – Offset = 24 – 11 – 5 = 8 bits

- *4-way set associative mapping*
SETs = (# Entries) / k = $2^{11}/4 = 2^{11}/2^2 = 2^9$ sets
Offset = $\log_2$(Entries size) = $\log_2 2^5$ = 5 bits
Index = $\log_2$(# SETs) = $\log_2 2^9$ = 9 bits
TAG = Address – Index – Offset = 24 – 9 – 5 = 10 bits



● ***Prediction and principle of locality***

How to ***maximise the hit ratio h***?

1. Place the data which is likely to be used to the cache.
2. Store to the cache the most frequent addressed entries.
3. Have the cache size *large enough* to store the quantity of data that is usually elaborated at the same time, reducing as much as possible the *trashing* action.

The first two objectives are to keep blocks in the cache that are likely to be referenced in the near future. However, it is not easy to determine which blocks are about to be referenced.

The effectiveness of the cache is based on the property called ***locality of reference*** of programs and data. Most of programs' execution time is spent in routines in which many instructions are executed repeatedly: these instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. The actual detailed pattern of instruction sequencing is not important—the point is that *many instructions in localized areas of the program are executed repeatedly during some time period*.
The principle of locality manifests itself in two ways:

- ***Temporal***, a recently executed instruction is likely to be executed again very soon (*next in time*).
- ***Spatial***, where instructions close to a recently executed instruction (*following in space*) are also likely to be executed soon.

The ability to choose which data the CPU will probably need in the next few minutes (*speculate the future*) is called ***prediction***. The work is done before it is known whether it is actually needed, to prevent a delay that would have to be incurred by doing the work after it is known that it is needed. If it turns out the work was not needed after all, most changes made by the work are reverted and the results are ignored. Prediction is strictly correlated to the locality of reference principles: therefore, it is not possible to use these properties in case of *jump instructions*.

The third objective is related to the capacity of storing as much data as may be needed: *nearby locations need to be held in the cache at the same time*, so they will have to be mapped to different cache lines.

Having a "large enough cache" is a trade-off between the *size*, the *cost* and the target *h*: ***benchmark*** programs are used to assess a system efficiency and capabilities by doing simulations and putting it under stress.

● ***Replacement Algorithms***

In a *direct-mapped* cache, the position of each block is predetermined by its address: hence, there is no need of a replacement strategy neither there is a choice to make.

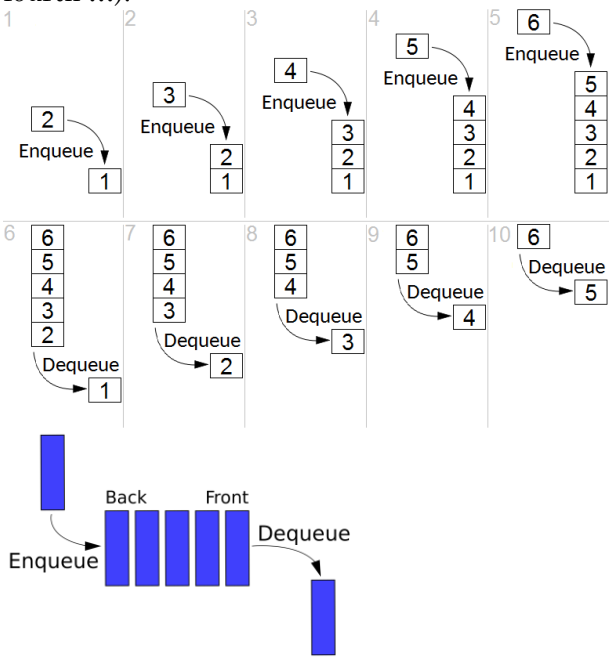
In *full associative* and *set-associative* caches, there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide *which of the old blocks to overwrite*. This is an important issue, because the decision can be a strong determining factor in system performance (keeping an high h and small t_G).

When a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the ***least recently used*** (LRU) block, and the technique is called the *LRU replacement algorithm*.
To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds, therefore a counter per entry and one or more comparators are required.
Suppose it is required to track the LRU block of a four-block set in a set-associative cache. A 2-bit counter can be used for each block.

- When a *hit* occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged.
- When a *miss* occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one.
- When a *miss* occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

The LRU algorithm has been used extensively: although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache.
Performance of the LRU algorithm can be improved by *introducing a small amount of randomness* in deciding which block to replace.

Another algorithm is to remove the "oldest" block from a full set when a new block must be brought in (***FIFO***, ***first in first out***): however, this does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove.
FIFO would require a time stamp to compute blocks' age and one or more comparators; it is simple and more easily implemented with the *round-robin* or *circular buffer* technique, requiring no more a comparator but only a pointer for each set (data is placed in order of arrival, a pointer follows the oldest element and, when the block is full and new data needs to be written, the earliest element is deleted and the pointer passes to the second one, then to the third, fourth …).



Another possibility is ***least frequently used*** (***LFU***): replace that block in the set that has experienced the fewest references.
LFU requires a counter and one or more comparators.

The simplest algorithm is ***random replacement*** (***RR***): it ***randomly*** chooses the block to be overwritten, not requiring keeping any information about the access history. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

● Write Policy

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one *write operation* has been performed on a word in that line of the cache, then ***main memory must be updated*** by writing the line of cache out to the block of memory before bringing in the new block.

These two problems can be defined as ***mis-alignment between cache and memory*** (or ***consistency***). First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

The simplest technique is called ***write through***: all write operations are made to main memory as well as to the cache, ensuring that main memory is *always* valid (aligned). Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache (in the case of the same RAM shared between different processors). The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck and huge delay ($t_{writing} = t_{RAM}$).

An alternative technique, known as ***write back***, minimizes memory writes: updates are made only in the cache. When an update occurs, a ***dirty bit*** (or ***use bit*** or ***modified bit*** or ***write flag***) ***M*** associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck.

When power is first turned on, the cache contains no valid data. A control bit, usually called the ***valid bit***, must be provided for each cache block to indicate whether the data in that block are valid. This bit should not be confused with the *modified bit* mentioned earlier. The valid bits of all cache blocks are set to 0 when power is initially applied to the system. Some valid bits may also be set to 0 when new programs or data are loaded from the disk into the main memory. If the memory blocks being updated are currently in the cache, the valid bits of the corresponding cache blocks are set to 0. As program execution proceeds, the valid bit of a given cache block is set to 1 when a memory block is loaded into that location. The processor fetches data from a cache block only if its valid bit is equal to 1.

The use of the valid bit in this manner ensures that the processor will not fetch ***stale data*** (invalid data) from the cache.

A similar precaution is needed in a system that uses the write-back protocol. Under this protocol, new data written into the cache are not written to the memory at the same time. Hence, data in the memory do not always reflect the changes that may have been made in the cached copy. It is important to ensure that such stale data in the memory are not transferred to the disk. One solution is to ***flush*** the cache, by forcing all dirty blocks to be written back to the memory before performing the transfer. Flushing the cache does not affect performance greatly, because such disk transfers do not occur often. The need to ensure that two different entities use identical copies of the data is referred to as a ***cache-coherence*** problem (the most widely used approach is the *MESI* (modified / exclusive / shared / invalid) protocol.

The ***write policies*** on write hit often distinguish cache designs:

Write through - the information is written to both the block in the cache and to the block in the lower-level memory.
Advantage:
- read miss never results in writes to main memory
- easy to implement
- main memory always has the most current copy of the data (consistent)

Disadvantage:
- write is slower
- every write needs a main memory access
- as a result, uses more memory bandwidth

Write back - the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a dirty bit is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean, the block is not written on a miss.
Advantage:
- writes occur at the speed of the cache memory
- multiple writes within a block require only one write to main memory
- as a result, uses less memory bandwidth

Disadvantage:
- harder to implement
- main memory is not always consistent with cache
- reads that result in replacement may cause writes of dirty blocks to main memory

● *Unified versus split caches*

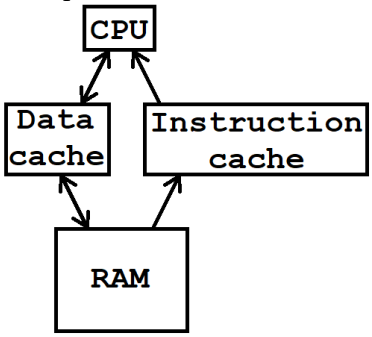
When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to ***split the cache into two***: one dedicated to ***instructions*** and one dedicated to ***data***.
These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

Unified cache has two potential advantages:
- For a given cache size, a unified cache has a higher hit rate than split caches because it ***balances the load between instruction and data fetches automatically***. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
- Only one cache needs to be designed and implemented.

However, ***split cache*** design eliminates contention for the cache between the instruction *fetch/decode unit* and the *execution unit*. This is important in any design that relies on the ***pipelining*** of instructions.
Typically, the processor will fetch instructions ahead of time and fill a buffer, or pipeline, with instructions to be executed.

In the case of a unified instruction/data cache: when the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the *instruction prefetcher* issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction.

This cache contention can *degrade performance* by interfering with efficient use of the instruction pipeline.
The split cache structure overcomes this difficulty.



Data cache will need a write policy that ensures memory consistency; instruction cache works only one-way from the RAM to the CPU so we do not need to update the original blocks in the main memory.

● *Cache and instructions workflow*

t_A average access time to the cache
t_c access time with hit result
t_R access time with miss result + get data from RAM
t_G management time, to unload data once it has been retrieved
h hit ratio value

Executions phases of an instruction are:
- *Fetch*, which consists in reading an instruction from the RAM through the instruction cache and storing it there;
- *Decode*, when the instruction is interpreted inside the CPU.
- *Execute*, retrieve the needed data (if not already available from the CPU registers, from data cache or RAM) and proceed to do what the instruction asks.

0) At the beginning, the cache is empty.
   The program is copied from the HD to the RAM.
1) The CPU asks the cache, a miss is returned. Data is *fetched* from RAM and copied to cache. → t_R
2) Then, n instructions are *decoded* and *executed*: all of them are found in the cache. → $n \times t_c$
3) At a certain point, the next instruction is not found in the cache: it is fetched again from the RAM. → t_R

In case of jump instructions, they may be found or not in the cache: in the second case, the loop is broken and the delay will be t_R.

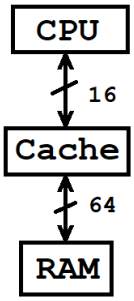
▶ *Example*
Instruction: `ADD CX, AX, VARIABLE`
Which corresponds to `CX ← AX+VARIABLE`

`CX, AX` are registers, already inside the CPU.
The only operand to retrieve outside the CPU is `VARIABLE`, which is stored in data cache or RAM.

● *Hardware requirements*

For the locality principle, it is better, while reading a RAM cell, to bring to the cache also the adjacent cells. By doing so, each time the loop is repeated and gets a hit result, h increments.

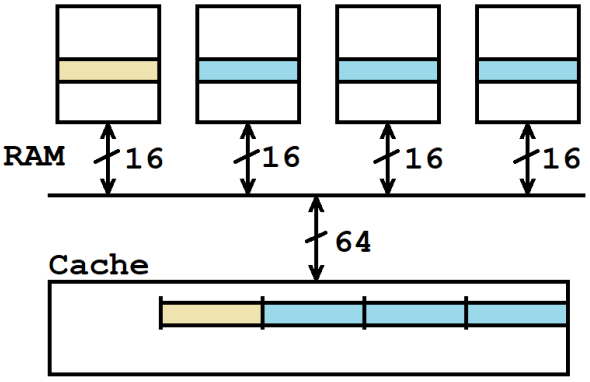
Assume to have:
- RAM cell size (data size) = CPU-cache bus size = 16 bits
- Cache-RAM bus size = 64 bits



For the first execution, when the CPU requests and accesses the RAM, the bus connecting RAM and cache should be enough large to transfer n elements in parallel (in this case, 64/16= 4 elements at a time). This allows to read the interested cell and its adjacent ones in only 1 clock cycle.
If the neighbour cells are needed in the following time, the CPU will be able to access them directly from the cache, avoiding making more delay for RAM reading.

As said, in this case only 4 elements can be moved from RAM to cache in parallel: no more are moved because it would cost another clock cycle and an eventual jump would mean wasting such delay.

What are the RAM and cache architectures that permit reading in parallel the 4 cells?
The RAM is organised in a mode (without a decoder, from ⇨*Memory organisation* lecture), so this process is as fast as possible regarding consequential data. The selection phase of which element has to be used is left to a second stage.



Each cache entry contains the data from all the RAM cells together (in this case all the 64 bits).

Therefore, splitting the RAM in more chips and permitting a parallel work (removing the decoder) gives a better interface between RAM and cache.

● *Virtual Memory*

In most modern computer systems, the physical main memory is not as large as the address space of the processor.
For example, a processor that issues 32-bit addresses have an addressable space of 4G bytes. The size of the main memory in a typical computer with a 32-bit processor may range from 1GB to 4GB.

If a program does not completely fit into the main memory (RAM), the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk (HD). As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory: this process is similar to the caching done by the RAM.

A common HD has a speed of 7200rpm (revolutions per minute), which correspond to 120Hz.
DDR4 SDRAM has a speed of 1600-3200MHz, which correspond to the order of magnitude of 10^9Hz. (see ⇨*Memory types* lecture)
By comparing the 10^9Hz of RAM with 10^2Hz of HD, the latter is absolutely slow because of the different technologies they use. The gap between RAM and cache is about 10, while between RAM and HD is 10^7.

In the case of RAM and cache, the management time t_G needs to be much less than the RAM access time t_R otherwise it would impact negatively the delay and correspond to no benefit from the architecture; to accomplish this the cache manager is hardware.
In the case of HD and RAM, the management time t_G can be less than the HD access time also if it is implemented in software: the ***operating system*** includes and acts as the ***virtual memory manager***.

Users and application programmers do not need to be aware of the limitations imposed by the available main memory (or, in fact, ***real memory***): they prepare programs using the entire address space of the processor (allocated on the disk and corresponding to the ***virtual memory***) and the operating system, as an *interface*, will interact with the hardware and take care of the rest ("hiding" the hardware from the user).

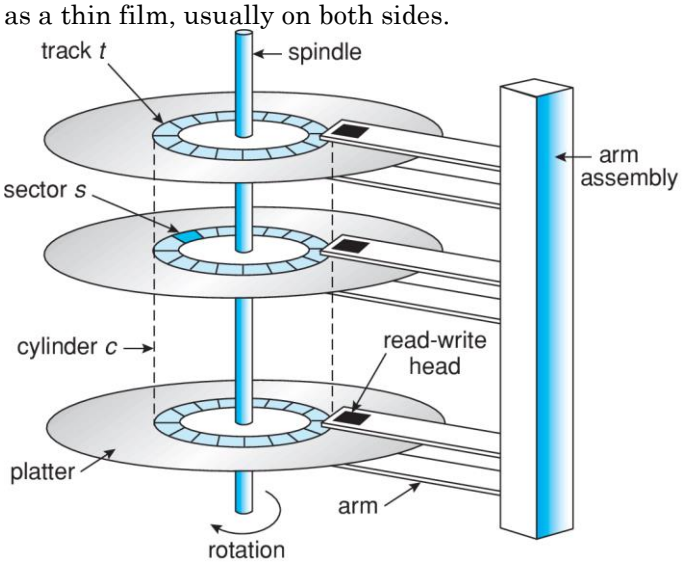
Under a virtual memory system, programs, and hence the processor, reference instructions and data in an address space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called *virtual* or *logical addresses*. These addresses are translated into *physical addresses* by a combination of hardware and software actions.

If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.

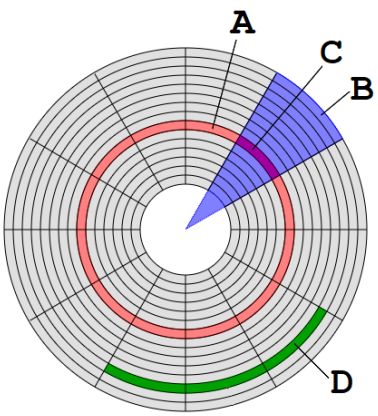
A special hardware unit, called the *Memory Management Unit* (MMU), keeps track of which parts of the virtual address space are in the physical memory. When the desired data or instructions are in the main memory, the MMU translates the virtual address into the corresponding physical address. Then, the requested memory access proceeds in the usual manner. If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory.

• ***Serial Memories: Magnetic Hard Disks***

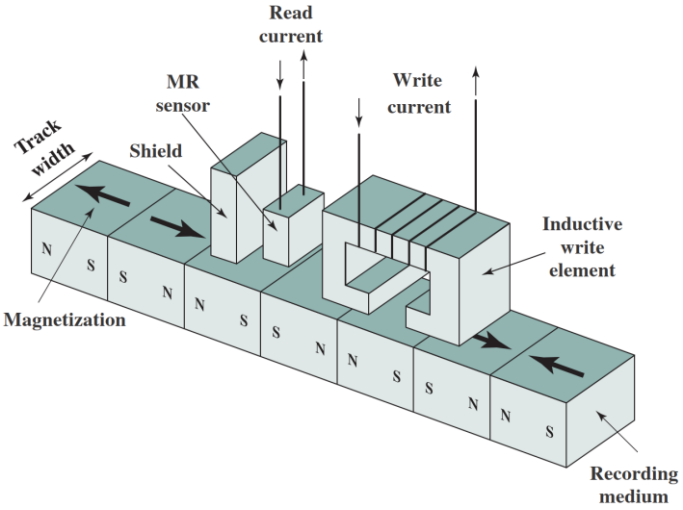
A hard-disk drive (HDD) contains one or more rotating platters mounted on a common spindle; the platters rotate at a constant speed ranging from 5400 to 15000 RPM. Each platter is coated with a magnetic material as a thin film, usually on both sides.



A read/write head (consisting of a magnetic yoke and a magnetizing coil) is positioned in close proximity above the surface of a platter and it can detect or change the orientation of the magnetization of the magnetic material below. The read/write head is mounted on an actuator that allows it to be positioned over and access different circular tracks by moving radially.



Data are stored on concentric ***tracks*** (A); each track has the same width as the head. Adjacent tracks are separated by *intertrack gaps* to prevent, or at least minimize, errors due to misalignment of the head or simply interference of magnetic fields.
A ***geometrical sector*** (B) is a portion of a disk between a centre, two radii and a corresponding arc, shaped like a slice of a pie.
A *disk sector* or ***block*** (C) refers to the intersection of a track and geometrical sector: it is identified with track number + geometrical sector number.
To reduce the overhead of managing on-disk data structures, the filesystem does not allocate individual disk sectors by default, but contiguous groups of sectors, called ***clusters*** (D): a cluster is the smallest logical amount of disk space that can be allocated to hold a file.



Three signals are used in HD: ***data***, ***clock*** and ***voltage*** (circuit attached to the ***ground***).

Digital information can be *stored* on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately underneath the head to switch to a direction parallel to the applied field. The same head can be used for *reading* the stored information. In this case, changes in the magnetic field in the vicinity of the head caused by the movement of the film relative to the yoke induce a ***voltage*** in the coil, which now serves as a sense coil. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film.

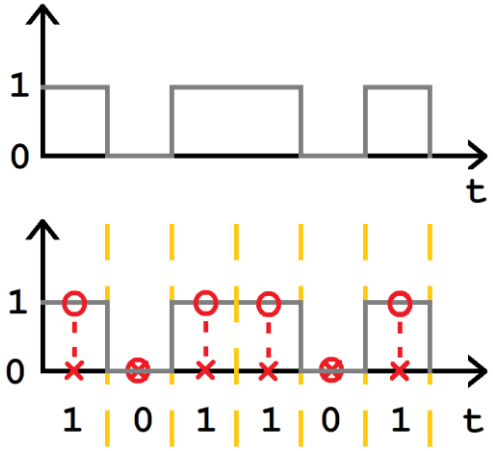
Only changes in the magnetic field under the head can be sensed during the *read* operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization (and therefore ***data***), a voltage is induced in the head only at 0-to-1 and at 1-to-0 transitions in the bit stream. A long string of 0s or 1s causes an induced voltage only at the beginning and end of the string. Therefore, to determine the number of consecutive 0s or 1s stored, a ***clock*** must provide information for synchronization: *using the clock signal as a reference, the data stored on other tracks can be read/wrote correctly*.

► *Manchester encoding*

In early designs, the clock was stored on a separate track, on which a change in magnetization is forced for each bit period. The modern approach is to ***combine the clocking information with the data***: only two wires are therefore needed, *clock+data* and ***ground*** (or ***voltage***).

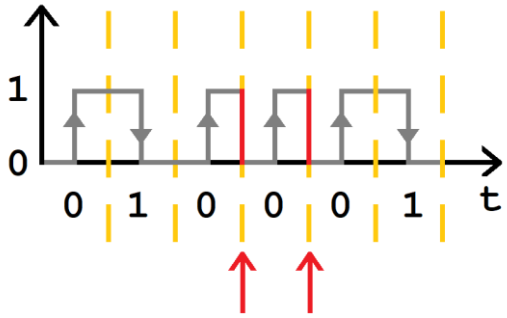
Several different techniques have been developed for such encoding. One simple scheme is known as *phase encoding* or ***Manchester encoding***, which is an edge type of encoding.

*Example – Sampling window in **level** encoding*



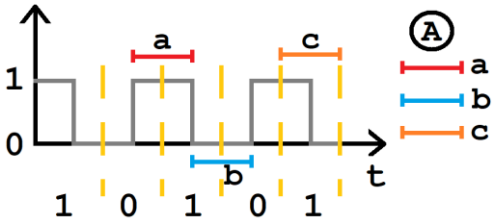
The transitions of signal value occur at the ***boundaries*** (beginning and end) of the ***sampling window***, which width depends on the clock frequency; sampling is done in the ***middle*** of the sampling windows, which is the most far point from the transitions and permits to be quite sure of the read value.

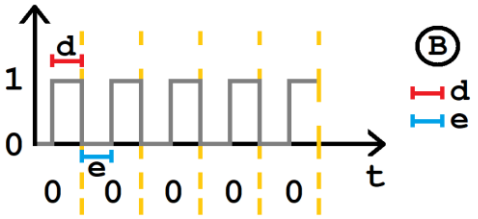
*Example – Sampling window in **edge** encoding*



In this an ***edge*** type of encoding, rising edge corresponds to 1 and falling edge corresponds to 0. Edge encoding always has a transition at the ***middle*** of each bit period and may (depending on the information to be transmitted) also have a transition at the ***start/end*** of the period. *The direction of the mid-bit transition indicates the data*. Transitions at the period boundaries do *not* carry information.

How to obtain three consecutive zeros? We use two up-to-low transitions (pointed out with the arrows) on one of the boundaries, which do not communicate any information because they do not occur in the middle of the sampling window.





From the two graphs we can see that:

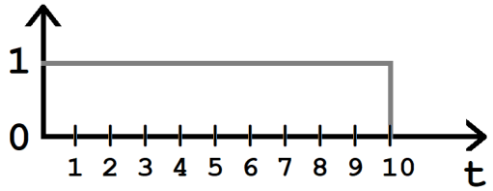
A) `a, b, c` have the same length, which corresponds to the duration of the clock. Therefore, the signal waveform and the clock have the same frequency.

B) `d, e` have half the length of the clock, therefore the double of its frequency.

Fourier analysis is the study of the way general functions may be represented or *approximated* by sums of simpler trigonometric functions.
By applying Fourier's rules to the signal, we process it through an electric ***band filter***: the filter identifies and isolates the frequency of the clock (f_C , f_{2C}).

Example – Transmit and receive the signal

Let's suppose to have a level encoding with the following signal.



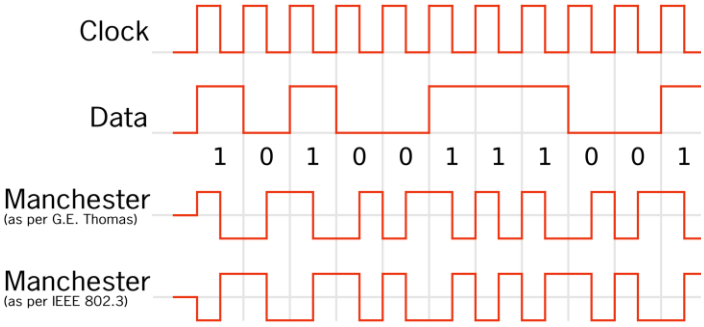
Suppose the sampling window (clock frequency) is $\Delta=1$: time is equal to 10 time units, the transmitted signal would be ten "1".

If the sampling window is $\Delta=1\pm20\%$
- the writer (transmitter) transmits with $\Delta=1$
- the reader (receiver #1) receives with $\Delta'=1-20\%=0.8$
- the reader (receiver #2) receives with $\Delta''=1+20\%=1.2$

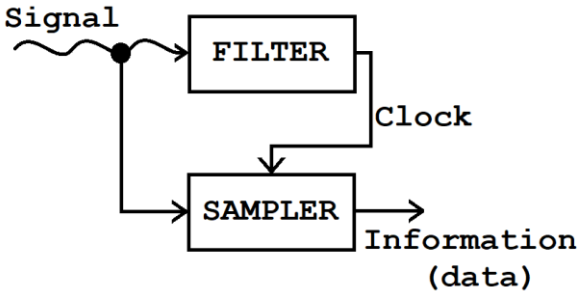
How many "1" are sampled by each receiver?
- #1 $\Delta'=0.8 \rightarrow 10/\Delta'=12.5=12$ "1" are received
- #2 $\Delta''=1.2 \rightarrow 10/\Delta''=8$ "1" are received

Therefore, a measure to guarantee that what was transmitted is received also when the clock is misaligned. Solutions:
- do not rely on the receiver's clock, but send the transmitter's clock with the data (*Manchester encoding*)
- have short transmissions to contain the error and do not accumulate more/less values than the ones transmitted, then resynchronising the transmission.

Manchester encoding is used to ***solve the synchronization problem*** for reading from and writing to a hard drive (avoids that transmitted and received data are misaligned): by combining clock with data, it is not necessary to reserve a track on the disk only for the clock.



In the Manchester encoding:
- the transmitter sends the signal
- the receiver has a clock which approximatively corresponds to the one on the transmitter
- the signal is doubled; one is passed through a filter which extracts the clock
- the other copy of the signal is processed through a sampler circuit with the help of the extracted clock
- the output of the sampler circuit is the relevant information (data)



What is the critical part? We need to add a delay block to allow the clock to be extracted and properly used since the first part of the signal. This approach arises another problem: the delay block must be tuned with the delay of the filter, which is very difficult and therefore this idea is discarded.

The solution is to add a ***preamble*** (or ***jamming bits***) and a ***queue*** to the relevant information:
- the preamble corresponds to a series of 1010, which corresponds to the values that assumes the clock and helps the filter to extract it. If the preamble arrives at the sampler circuit before the clock there are no consequences, because it does not carry the real information.
- the queue can carry the parity bit, to confirm that the information is actually correct.

The length of preamble and queue depend on the standards of the system; they are set by the manufacturer.

Therefore, the signal corresponds to the following

Preamble	**Relevant information**	Queue (parity bit)

Obviously, this system is inefficient because to transmit a certain amount of data we need to send more (preamble and queue) to be sure it will be received correctly. This is the cost to use two signals (clock+data and ground) instead of three.

Changes in magnetization occur for each data bit, as shown in the figure. Clocking information is provided by the change in magnetization at the midpoint of each bit period.

The drawback of Manchester encoding is its poor bit-storage density. The space required to represent each bit must be large enough to accommodate two changes in magnetization.
We use the Manchester encoding example to illustrate how a *self-clocking* scheme may be implemented, because it is easy to understand. Other, more compact codes have been developed. They are much more efficient and provide better storage density. They also require more complex control circuitry. The discussion of such codes is beyond the scope of these notes.

► *Winchester technology*
Read/write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities and reliable Read and Write operations. When the disks are moving at their steady rate, air pressure develops between the disk surface and the head and forces the head away from the surface. This force is counterbalanced by a spring-loaded mounting arrangement that presses the head toward the surface. The flexible spring connection between the head and its arm mounting permits the head to fly at the desired distance away from the surface in spite of any small variations in the flatness of the surface.

In most modern disk units, the disks and the read/write heads are placed in a sealed, air-filtered enclosure. This approach is known as *Winchester technology*.
In such units, the read/write heads can operate closer to the magnetized track surfaces, because dust particles, which are a problem in unsealed assemblies, are absent. The closer the heads are to a track surface, the more densely the data can be packed along the track, and the closer the tracks can be to each other. Thus, Winchester disks have a larger capacity for a given physical size compared to unsealed units. Another advantage of Winchester technology is that data integrity tends to be greater in sealed units, where the storage medium is not exposed to contaminating elements.

► *Disk layout*

The read/write heads of a disk system are movable. There is one head per surface. All heads are mounted on a comb-like arm that can move radially across the stack of disks to provide access to individual tracks. To read or write data on a given track, the read/write heads must first be positioned over that track.

The disk system consists of three key parts.
- One part is the assembly of disk platters, which is usually referred to as the *disk*.
- The second part comprises the electromechanical mechanism that spins the disk and moves the read/write heads; it is called the *disk drive*.
- The third part is the *disk controller*, which is the electronic circuitry that controls the operation of the system. The disk controller may be implemented as a separate module, or it may be incorporated into the enclosure that contains the entire disk system.

The term disk is often used to refer both the combined package of the disk drive and the disks (*platters*) it contains. Each surface is divided into concentric *tracks*, and each track is divided into *sectors*. The set of corresponding tracks on all surfaces of a stack of disks forms a logical *cylinder*. All tracks of a cylinder can be accessed without moving the read/write heads. Data are accessed by specifying the surface number, the track number, and the sector number. Read and Write operations always start at sector boundaries.

► *Disk formatting*

Data bits are stored on each *track* in many serial blocks. Each *block* (or *physical sector*) may contain 512 or more bytes.
There are small *inter-sector gaps* that enable the disk control circuitry to distinguish easily between two consecutive sectors.
The *sector header* (or *ID field*) contains:
- *Synch byte* (or *synch info*), which is the jamming code
- *Address block*, where the identification (addressing) information of the block is stored (*track #* , *head #*, *sector #*)
- *CRC* (*cyclic redundancy check*), an error detecting code (see ⇨*Error codes* lecture)

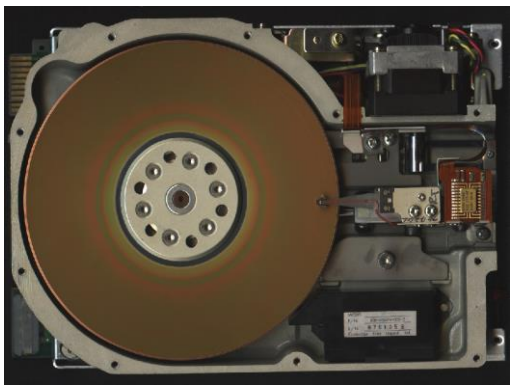
The *data field* contains:
- *Synch byte* (or *synch info*)
- *Data*,
- *CRC* (*cyclic redundancy check*)

The *Gap 2* between ID field and Data field is needed for checking that the ID field is actually corresponding to the one requested.

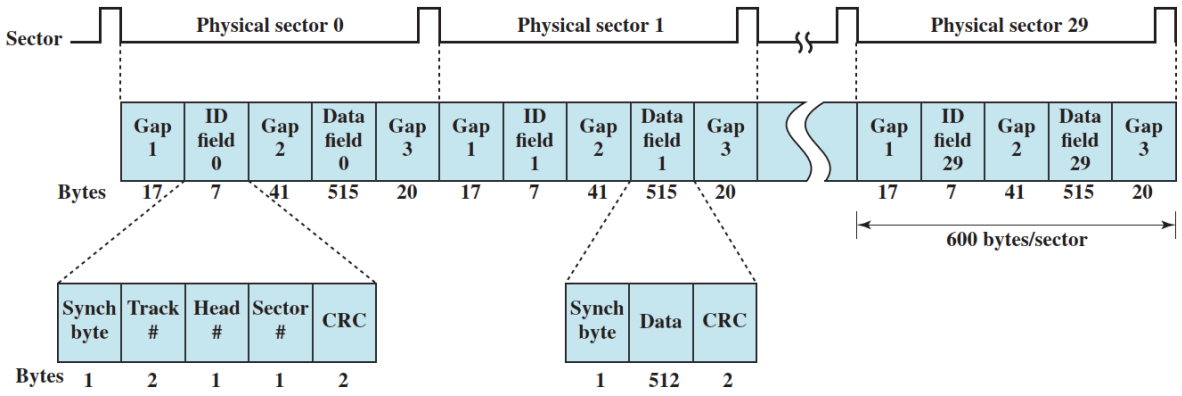
An unformatted disk has no information on its tracks. The ***formatting process*** writes markers that divide the disk into tracks and sectors. During this process, the disk controller may discover some sectors or even whole tracks that are defective. The disk controller keeps a record of such defects and excludes them from use. The ***formatting information*** comprises sector headers, ECC bits, and inter-sector gaps.
The capacity of a formatted disk, after accounting for the formatting information overhead, is the proper indicator of the disk's storage capability. After formatting, the disk is divided into logical partitions.

Example: 2MB floppy drives had only 1.44MB of usable space after formatting; the missing 0.56MB was taken by the formatting information.

Each track has the same number of sectors, which means that *all tracks have the same storage capacity*. In this case, the stored information is packed more ***densely*** on inner tracks than on outer tracks. It is also possible to increase the storage density by placing more sectors on the outer tracks, which have longer circumference. This would be at the expense of more complicated access circuitry.



Seagate ST506 (↑) and its Winchester Disk Format (↓)
First 5.25-inch hard disk drive, introduced in 1980. It stored up to 5 megabytes after formatting and costed US$1,500.



► *Note: THIS PAGE is not requested for the exam* ◄

► *Access Time*

There are two components involved in the time delay between the disk receiving an address and the beginning of the actual data transfer. The first, called the *seek time*, is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. Average values are in the 5- to 8-ms range. The second component is the *rotational delay*, also called *latency time*, which is the time taken to reach the addressed sector after the read/write head is positioned over the correct track. On average, this is the time for half a rotation of the disk. The sum of these two delays is called the disk *access time*. If only a few sectors of data are accessed in a single operation, the access time is at least an order of magnitude longer than the time it takes to transfer the data.

► *Data Buffer/Cache*

A disk drive is connected to the rest of a computer system using some standard interconnection scheme, such as SCSI or SATA. The interconnection hardware is usually capable of transferring data at much higher rates than the rate at which data can be read from disk tracks. An efficient way to deal with the possible differences in transfer rates is to include a *data buffer* in the disk unit. The buffer is a semiconductor memory, capable of storing a few megabytes of data. The requested data are transferred between the disk tracks and the buffer at a rate dependent on the rotational speed of the disk. Transfers between the data buffer and the main memory can then take place at the maximum rate allowed by the interconnect between them.

The data buffer in the disk controller can also be used to provide a caching mechanism for the disk. When a Read request arrives at the disk, the controller can first check to see if the desired data are already available in the buffer. If so, the data are transferred to the memory in microseconds instead of milliseconds. Otherwise, the data are read from a disk track in the usual way, stored in the buffer, then transferred to the memory. Because of locality of reference, a subsequent request is likely to refer to data that sequentially follow the data specified in the current request. In anticipation of future requests, the disk controller may read more data than needed and place them into the buffer. When used as a cache, the buffer is typically large enough to store entire tracks of data. So, a possible strategy is to begin transferring the contents of the track into the data buffer as soon as the read/write head is positioned over the desired track.

► *Disk Controller*

Operation of a disk drive is controlled by a *disk controller* circuit, which also provides an interface between the disk drive and the rest of the computer system. One disk controller may be used to control *more than one drive*.

A disk controller that communicates directly with the processor contains a number of registers that can be read and written by the operating system. Thus, communication between the OS and the disk controller is achieved in the same manner as with any I/O interface, as discussed before. The disk controller uses the DMA scheme to transfer data between the disk and the main memory. Actually, these transfers are from/to the data buffer, which is implemented as a part of the disk controller module. The OS initiates the transfers by issuing Read and Write requests, which entails loading the controller's registers with the necessary addressing and control information.

Typically, this information includes:

Main memory address—The address of the first main memory location of the block of words involved in the transfer.

Disk address—The location of the sector containing the beginning of the desired block of words.

Word count—The number of words in the block to be transferred. The disk address issued by the OS is a logical address. The corresponding physical address on the disk may be different. For example, bad sectors may be detected when the disk is formatted. The disk controller keeps track of such sectors and maintains the mapping between logical and physical addresses. Normally, a few spare sectors are kept on each track, or on another track in the same cylinder, to be used as substitutes for the bad sectors. On the disk drive side, the controller's major functions are:

Seek—Causes the disk drive to move the read/write head from its current position to the desired track.

Read—Initiates a Read operation, starting at the address specified in the disk address register. Data read serially from the disk are assembled into words and placed into the data buffer for transfer to the main memory. The number of words is determined by the word count register.

Write—Transfers data to the disk, using a control method similar to that for Read operations.

Error checking—Computes the error correcting code (ECC) value for the data read from a given sector and compares it with the corresponding ECC value read from the disk. In the case of a mismatch, it corrects the error if possible; otherwise, it raises an interrupt to inform the OS that an error has occurred. During a Write operation, the controller computes the ECC value for the data to be written and stores this value on the disk.

● *RAID Techniques*

RAID is an acronym that stands for *Redundant Array of Independent Disks* (previously Redundant Array of Inexpensive Disks): it is a technique that improves reliability, performance (speed), or both by using multiple hard disks combined in arrays of disks that operate ***independently*** and ***in parallel***.
RAID is managed by the ***disk controller***.

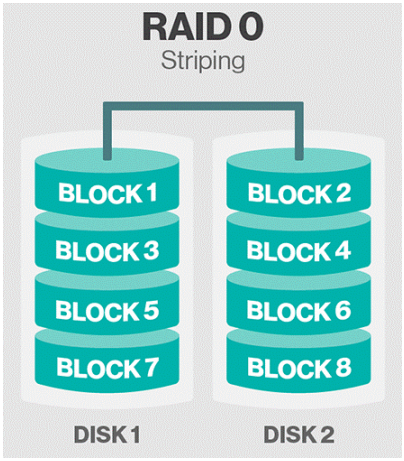
The RAID scheme consists of seven levels, zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:
1. RAID is a set of physical disk drives viewed by the operating system as a *single logical drive*.
2. Data are distributed across the physical drives of an array in a scheme known as ***striping***.
3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure. (not supported by RAID 0 and RAID 1)

Allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, but using multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

► ***RAID 0*** – Also known as ***striping***: data are divided in stripes and equally distributed over N(=2^n) disks, so that every access can be performed in parallel.

The advantages of this level are:
- *increased performance* since more writers and readers can access bits of data simultaneously. For locality principle, because near blocks are stored on different disks, it likely that operations are mostly done in parallel.
- a *larger logical disk* is obtained from multiple small physical disks. RAID 0 can use disks of differing sizes, but the storage space added to the array by each disk is limited to the size of the smallest disk.

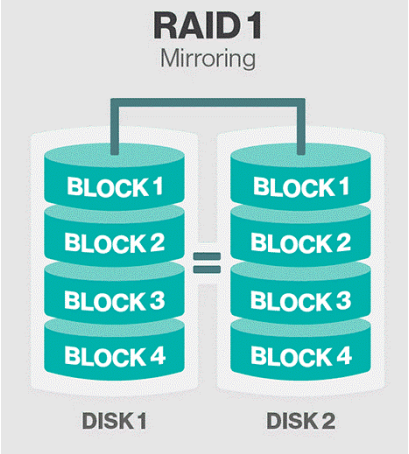
The disadvantage is the *lack of data redundancy*: the failure of a single disk (stopping working or working incorrectly) results in data loss (which cannot be recovered).



► ***RAID 1*** – Also known as ***disk mirroring***, this configuration consists of at least two drives that *duplicate* the storage of data. There is no striping neither parity. Read performance is improved since either disk can be read at the same time. Write performance is the same as for single disk storage.

The advantages to this configuration are that if one disk fails, the second disk keeps running with no interruption of data availability and there is no loss of data capacity.

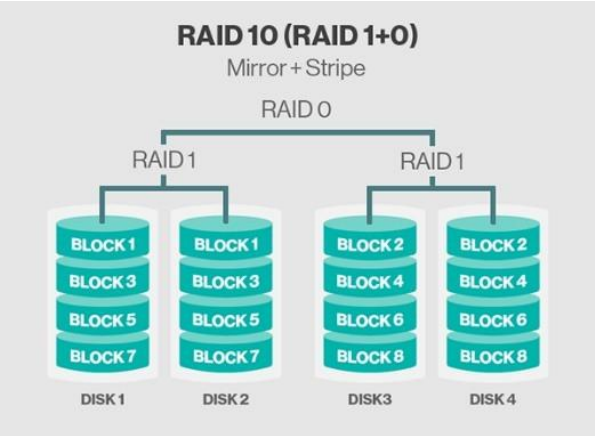
The disadvantages to this level are:
- the *cost*, since it requires twice as much disk space for the mirroring.
- it *does not provide protection* against data corruption, neither accidental file changes, deletions nor any other data specific changes since these changes are instantly mirrored to every drive in the array segment.
- the *storage capacity* is only as large as the smallest drive.



***Note**: in Computer Architecture course and exams RAID 1 will be called **disk mirroring**; **RAID 0+1** will be considered as actual RAID 1.*

► ***RAID 10*** (or ***RAID 1+0***) – the data is mirrored and the mirrors are striped. It is an improvement of RAID 0 (higher reliability with disk mirroring) and RAID 1 (higher performance through striping) but at a much higher cost: it requires a minimum of four disks, and stripes data across mirrored pairs. As long as one disk in each mirrored pair is functional, data can be retrieved. If two disks in the same mirrored pair fail, all data will be lost because there is no parity in the striped sets.

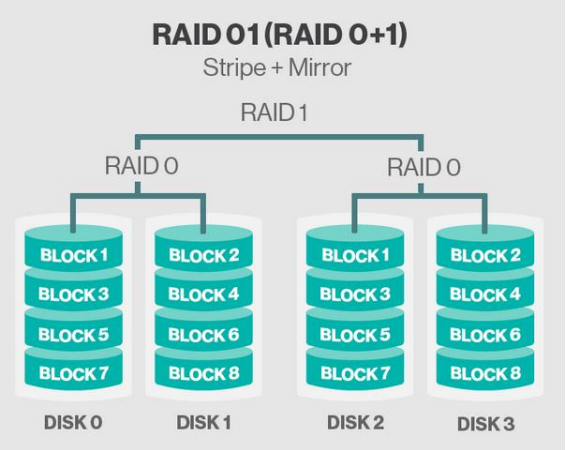
RAID 10 provides redundancy and performance, and is the best option for I/O- intensive applications.
One disadvantage is that at most only 50% of the total raw capacity of the drives is usable due to mirroring.



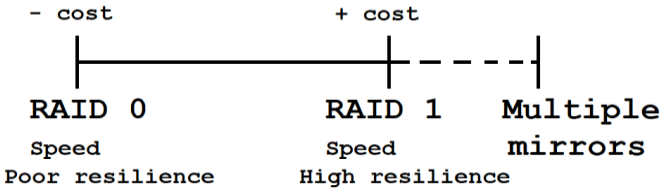
► ***RAID 01*** (or ***RAID 0+1***) – very similar to RAID 1+0: rather than creating a mirror and then stripping the mirror, RAID 0+1 creates a stripe set and then mirrors the stripe set.

The advantages of this configuration are:
- higher speed, larger logical space (better than RAID 1)
- backup disks for better redundancy in case of a hardware failure of a disk (better than RAID 0).

The disadvantages are:
- high cost (requires at least 4 disks)
- no protection against data corruption or deletion (each data change is instantly mirrored)
- capacity is only n times the size of the smallest drive (n = number of disks / 2 , because of mirroring)



What are the solutions to the trade-offs between speed, resilience and cost?



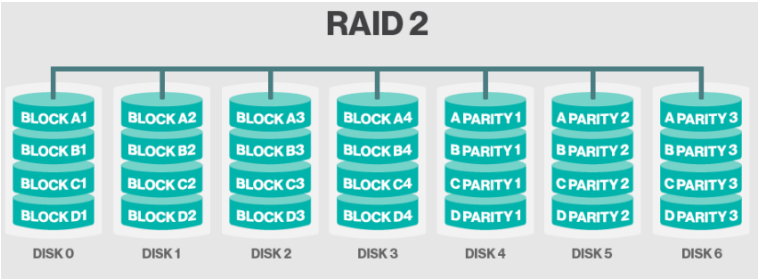
Considering the speed of an HD being 7200rpm (120Hz), using the RAID 0 or RAID 01 we obtain a theorical combined speed of 15000rpm (250Hz). Obviously, it is better than the original speed, but it will still be absolutely inferior to the RAM speed (GHz = 10^9Hz).

If there are *too many disks*, the locality principle is not valid anymore: only while it holds the virtual combined speed is usable. It is very unlikely that a program needs more than a certain amount of blocks in parallel. Furthermore, the wires needed to connect all of them would be too many (over a certain cut-off point), corresponding in a decrease of the speed, which is not reasonable. The controller would be very complex, and therefore expensive.

Can the cost (number of mirror disks) be traded off with a decrease of resilience? Yes.
Disk mirroring can be considered as an "expensive error correcting code", which can be used to reconstruct damaged data by overwriting original data with the mirror copy. It is robust (can correct multiple errors) but expensive (requires the double of the storage).
Using *parity code* instead of disk mirroring corresponds to a lower cost and a lower resilience.

► ***RAID 2*** – This configuration uses striping across disks with some disks storing error checking and correcting (***ECC***) information (specifically, *Hamming codes*).

It has no advantage over RAID 3 and is no longer used.



► *Note: RAID 2 is not requested for the exam* ◄

► ***RAID 3*** – This technique uses striping and dedicates ***one drive*** to storing ***parity*** information; is rarely used in practice.

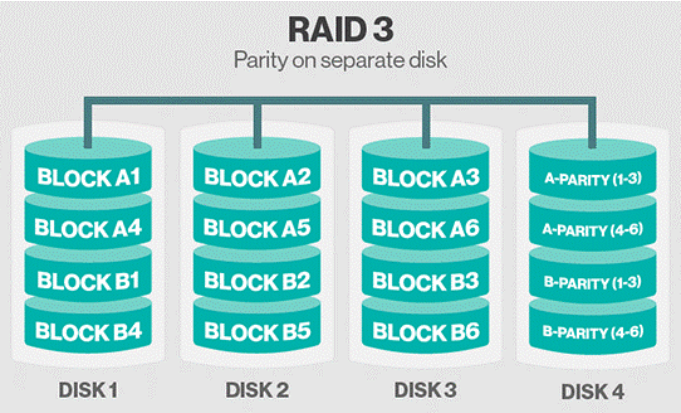
The embedded ECC information is used to detect errors. Data recovery is accomplished by calculating the exclusive OR (XOR) of the information recorded on the other drives: $D4 = D1 \oplus D2 \oplus D3$.

Example:
Disk 1 (D1) has corrupt / lost data, how to recover it?
By applying the XOR property from parity disk D4.
$D4 = D1 \oplus D2 \oplus D3$
$\rightarrow \quad D1 = D4 \oplus D2 \oplus D3 = (D1 \oplus D2 \oplus D3) \oplus D2 \oplus D3$

Since an I/O operation addresses all drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.

The advantages are the *high read speed* and the high *resilience*.

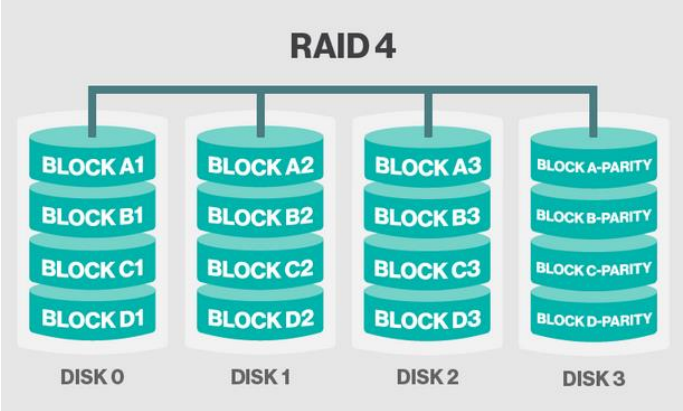
The disadvantages of this level are:
- the *write speed* is inadequate – the reason is the necessity of checksums calculating (even RAID hardware controllers cannot solve this problem).
- when a *disk failure* happens, the whole system will work much slower (*reduced mode*). Although RAID 3 is resistant to breakdown (in case of failure of one disk in the array), replacing a damaged disk is very costly.
- the *parity* (or *calibration* or *checksum*) *disk* is the bottleneck in the performance of the entire array. No matter which disk are the data written in, related information in parity disk must be re-written. However, because of the applications needing more writing operations, the load of calibration disk is very big. Therefore, it cannot meet the demand for program speed and the whole RAID system performance descends.



► ***RAID 4*** – This level uses large stripes, which means you can read records from any single drive. This allows to use overlapped I/O for read operations. Since all write operations have to update the parity drive, no I/O overlapping is possible.
RAID 4 offers no advantage over RAID 5.

The advantages are:
- *higher performance* than RAID 3, since RAID 4 allows each member of the set to act independently when only a single block is requested. If the controller permits it, the set can service multiple read requests simultaneously.
- it also provides *good fault tolerance* (both for error detecting and error correcting thanks to the CRC and parity bit) since data remains fully available in the event a disk fails. The parity bit helps reconstructing all the data lost from one of the other disks.

The disadvantage is that every *write* operation must involve the *parity disk*, which therefore can become a bottleneck.



In case a ***double error*** happens on data disks, they can be detected from CRC and not from parity (parity bit detects only odd number of errors): the consequence is that they ***cannot be corrected***.
The CRC of each block notifies the presence of an error (and therefore the disk where happened) but does not give any information regarding its position.

To *increase the protection*, we have to increase the number of backup disks (for example using more than one parity bit and setting a code, as in RAID 6).

► ***RAID 5*** – This level is based on ***block-level*** striping with parity. The parity information is striped across each drive (i.e. in a round-robin allocation scheme), allowing the array to function even if one drive were to fail.

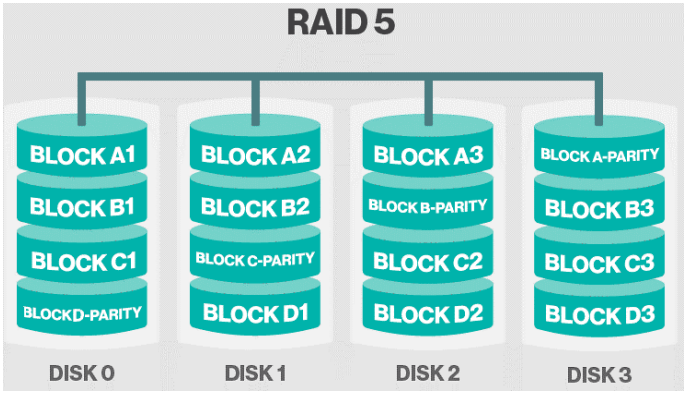
The array's architecture allows read and write operations to span multiple drives. This results in performance that is usually better than that of a single drive, but not as high as that of a RAID 0 array. RAID 5 requires at least three disks, but it is often recommended to use at least five disks for performance reasons.

RAID 5 arrays are generally considered to be a poor choice for use on write-intensive systems because of the performance impact associated with writing parity information.

When a disk does fail, it can take a long time to rebuild a RAID 5 array. Performance is usually degraded during the rebuild time and the array is vulnerable to an additional disk failure until the rebuild is complete.

The advantages to this level are that it balances data availability and read/write performance while providing *higher performance* than RAID 4 by eliminating the single parity disk bottleneck.

The disadvantage is possible performance degradation during drive rebuilds and it is expensive (requires between three and five drives per RAID group).

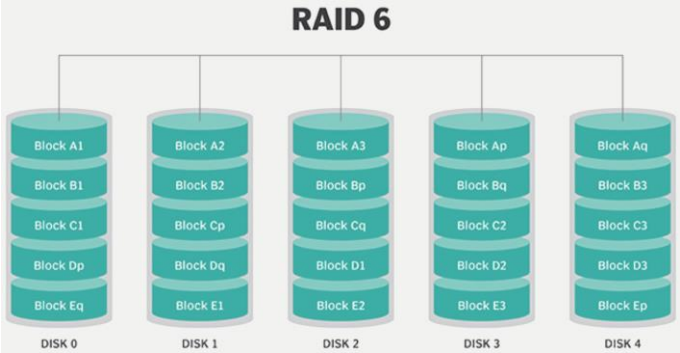


► *Note: RAID 5 is not requested for the exam* ◄

► ***RAID 6*** – two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require N disks consists of N + 2 disks.

P and Q are two different data check algorithms: this makes it possible to regenerate data even if two disks containing user data fail. Three disks would have to fail within the repairing time interval to cause data to be lost.

Compared to RAID 5, the reading performance is similar, while the writing performance is worse.



► *Note: RAID 6 is not requested for the exam* ◄

► *Example of RAID exercise*

Combine in RAID 0 mode the following disks:

- 0.5 TB 5400rpm
- 0.75 TB 3600rpm

What is the outcome?
The logical storage is the double of the smaller one; the remaining space of the bigger disk cannot be used in RAID 0 mode.
The virtual speed in case of locality principle validity is the double of the slower.

The result is a disk with:

- Logical storage $0.5 \times 2 = 1$ TB
- Virtual speed $3600 \times 2 = 7200$rpm

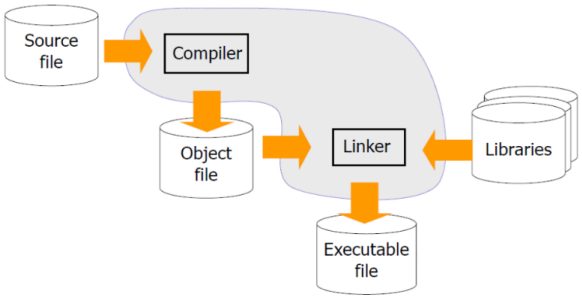
Summary of RAID characteristics

RAID	*Advantages*	*Drawbacks*
0	Speed + logical space	Poor reliability
1	Good reliability (for hardware failures)	High cost No error protection Storage = smallest disk
1+0 0+1	Performances = RAID 0 Logical space = RAID 0 Reliability = RAID 1	High cost > RAID 1 No error protection Capacity issues = RAID 1
3	High read speed Good reliability	Poor write speed Parity is bottleneck
4	Performances > RAID 3 Cheaper than RAID 1 Good reliability	Slower than RAID 1 because of parity disk
5	Speed > RAID 4 Cheaper than RAID 1	Slow in writing

8 – 8086 ASSEMBLY LANGUAGE

• *Summary of programming workflow*

A high-level language is a human-friendly language that uses variables and functions and it is independent of computer architecture. The programmer writes code with general purpose without worrying about the hardware integration part. A program written in high-level language needs to be first interpreted into machine code and then processed by a computer.



Workflow of programming in high-level language:
- The *program source* is written in high-level language
- The source is processed through a *compiler*, which
  - Translates the source in *object code*, making *object files*
  - Checks the absence of *syntactic errors* of the language
- The *linker* links objects files with the *libraries*
- The output of the linker is the actual *executable file*

Compiler and linker are themselves programs.

Programs are:
1) stored in the HD
2) when run, operating system launches the *loader*, which task is to copy the program executable from HD to RAM.
3) OS launches the executable from the "RAM copy"
4) and forwarded to the CPU.

When the size of a program is larger than the RAM, if the OS supports it, it is virtualized and only the portion used in that moment is forwarded to the RAM. When the execution of the first block of instruction is completed, the OS gets from the HD the second block and proceeds to execute them.

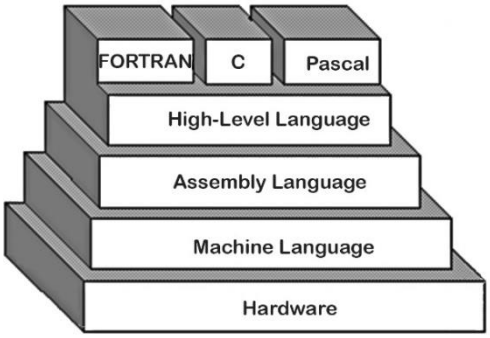
Following the instruction workflow (*fetch*, *decode*, *execute*): the executable's instructions are fetched from the RAM and sent to the CPU; they are processed through the *Control Decoding Unit* and translated into machine language, then executed.

► *Example*

High-level instruction	a ← b + c + d + e
Low-level translation	a ← b + c a ← a + d a ← a + e
Coding (machine language)	001101100…
Corresponding action (a ← a + d)	Add the contents of the fourth register to the first register and store the result in the first register. …
Assembly language (pseudo code)	ADD AX, DX

• *Introduction to 8086 Assembly Language*

Assembly language is a low-level programming language, used to execute precise machine level instructions.



Assembly language is coded differently for every type of processor because it depends completely on the architecture used: i.e., x86 and x64 processors have a different code of assembly language for performing the same tasks.
Assembly language has the same commands as machine language, but instead of 0 and 1 it uses names.

Performance and accuracy of assembly language code are better than a high-level: this permits to control and optimize programs and data with fine tuning even at the bit-level; however complex instructions at high level are much more manageable, while at low level they may not be performed or take a lot of time (see the example above for adding 4 variables).

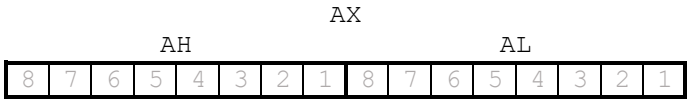
Assembly code is processed through an ***assembler*** and translated to machine language (similarly as the compiler processes high-level language).

● *8086 Microprocessor Architecture*

8086 is a 16-bit machine, which means the *data bus* size is 16 bits; *registers* of the CPU have a size of 16 bits as well. The *address bus* size is 20 bits.

Introduced (year)	1978
Clock speeds	5 MHz, 8 MHz, 10 MHz
Bus width	16 bits
Number of transistors	29,000
Feature size (mm)	3
Addressable memory	1 MB

A user-available register is called `AX`: it is divided into two parts, each one made of 8 bits, respectively called `AH` and `AL` and indicating the higher (most significant bits) and the lower (least significant bits) parts of `AX`.



► *Example*
`ADD AH, BL` which means `AH ← AH+BL`
This instruction adds the contents of `BL` (`BX low`) and `AH` (`AX high`) and stores the result into `AH`.

Other 16-bit registers equivalent to `AX` (with high and low parts) are `BX`, `CX`, `DX`.
`SP`, `BP`, `SI`, `DI` are 16-bits only registers.

Registers of 8086 machine:

General registers	
AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointers and index	
SP	Stack pointer
BP	Base pointer
SI	Source index
DI	Destination index

Segment	
CS	Code
DS	Data
SS	Stack
ES	Extract

Program status	
IP	Instruction pointer
	Flags

Instruction pointer `IP` (or *program cou*nter `PC`) points to the next instruction to be executed.

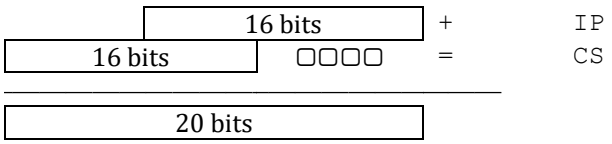
Stack pointer `SP` points at the *top of the stack* (`TOS`).
Stack segment `SS` points at the *upper bound* (which corresponds to the *minimum address of memory cell*) of the stack.

Stack pointer provides an address to the RAM where the code of next instruction is to be stored. During the fetch operation (reading the code of the instruction from the RAM) the CPU asks to go to the cell "pointed" by `IP` and retrieves its contents.

What is the gap?
- IP register is 16 bits
- A-BUS requires 20 bits to access the memory

What are used for the missing 4 bits?
A 16-bit A-BUS can address 64Kb of memory.
A 20-bit A-BUS can address 1Mb of memory.
Designers of 8086 wanted to make a processor that could be extended without changing all the architecture.

How are cells accessed?
For example, combining `IP` and `CS`: to make them fit, the `CS` part is shifted by 4 bits.



This technique uses segment registers `CS`, `DS`, `SS`, `ES`.

Anyway, to simplify our approach, let's suppose memory can be addressed with 16 bits only, ignoring the 20-bit A-BUS.

● *8086 Memory Byte Ordering*

Endianness refers to the order of bytes (or sometimes bits) within a binary representation of a number.
Considering a multi-byte number:
- *big-endian* ordering places the most significant bit first (lower address cell) and the least significant bit last (higher address cell);
- *little-endian* ordering places LSB first and MSB last.

For a given multibyte scalar value, big endian and little endian are byte-reversed mappings of each other.

Suppose we have the 32-bit hexadecimal value `12345678` and that it is stored in a 32-bit word in byte-addressable memory at byte location 184. The value consists of 4 bytes, with the least significant byte containing the value `78` and the most significant byte containing the value `12`.

Big endian	
Address	*Value*
184	12
185	34
186	56
187	78

Little endian	
Address	*Value*
184	78
185	56
186	34
187	12

The concept of endianness arises when it is necessary to treat a multiple-byte entity as a single data item with a single address, even though it is composed of smaller addressable units.

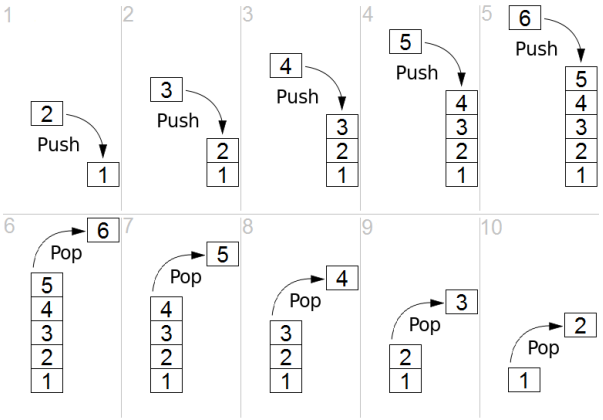
The byte ordering depends on the architecture, and therefore on the decisions of the designer: in the case of ***8086***, it is operating with ***little-endian ordering***.

• *8086 Stack implementation*

The ***stack*** is an *abstract data type* that serves as a collection of elements "stacked" on top of each other: this structure makes it easy to take an item off the ***top of the stack*** (TOS), while getting to an item deeper in the stack may require taking off multiple other items first.

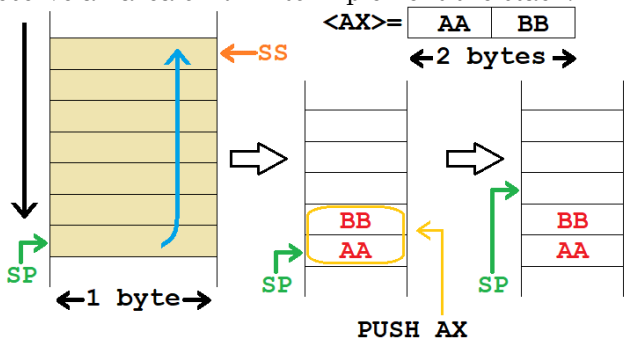
Stack has two principal operations:
- ***push***, which adds an element to the collection;
- ***pop***, which removes the most recently added element that was not yet removed.

The order in which elements come off a stack gives rise to its alternative name, ***LIFO*** (*last in, first out*).
Simple representation of a stack runtime with push and pop operations:



In 8086 one 16-bit element can be moved at a time, using push and pop instructions. Depending on the designer's implementation of stack in 8086, more features "can" be added.

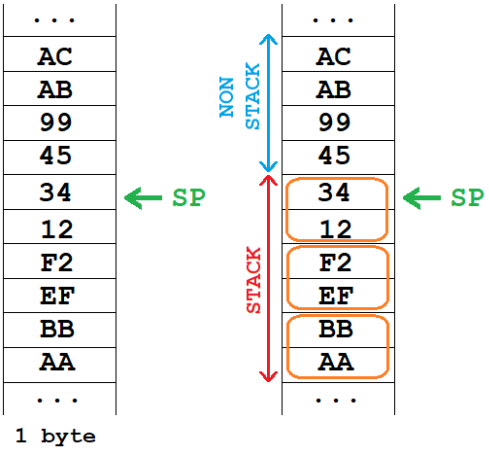
In 8086, stack is implemented in the RAM. An assembly program contains a definition of a variable (AX); this translates in a command to the compiler to reserve an area of RAM to implement the stack.



- RAM cell size is 1 byte (= 8 bits)
- 16-bit register AX contains 8-bit higher (AA) and 8-bit lower (BB) part
- direction of addresses (address value) increases from up to down (left arrow);
- growth of stack has a direction from higher to lower address cell (arrow on the cells); the empty stack has SP on the beginning cell and SS on the TOS;
- operation PUSH AX copies the values of the register AX to the stack, 8 bits per cell, beginning from SP and moving up; AA (MSB) is copied first and BB (LSB) is copied last because of little-endian ordering;
- the new SP position is the first free cell.

► *Example*

Suppose the following is the dump of the memory; the position of SP is known. Cell size is 1 byte; push/pop operate on 16-bit only. How many elements are in the stack and what are their values?

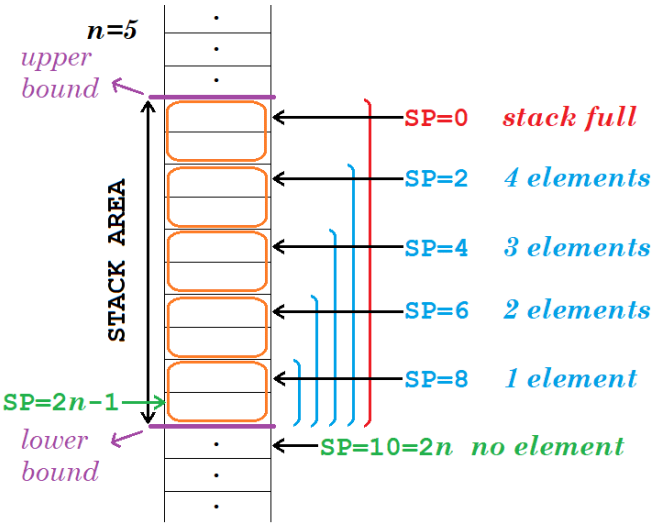


Stack is identified as the 6 cells at the bottom because:
- each stack elements occupies 2 cells (16 bits);
- SP must always point the TOS;
- 3 elements of the stack are found;
- the non-stack part is still free to host new elements of the stack, up to where SS is found.

We do not know the size of the stack!

► *Example*

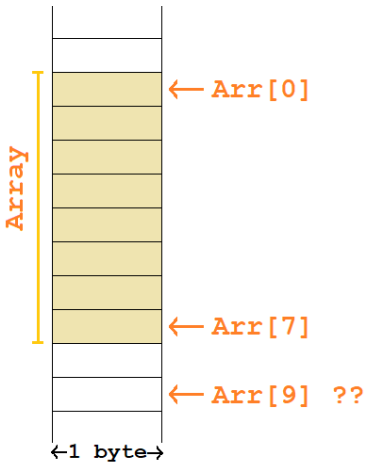
The size of the stack (or the stack area) corresponds all the cells between:
- *stack segment* SS, which points the maximum position in the stack (*upper bound*), its address is the minimum address of memory inside the stack, its value is set by the compiler and cannot change while the program is running;
- *stack pointer* SP, which at the beginning is pointing an empty cell at the bottom of the stack (*lower bound*), its value is also set by the compiler at the beginning such that if the stack can hold n entries, SP ← 2×n.



The user asks the compiler for enough space to store n elements, the compiler reserves the required cells for the stack in the memory.

SP *decreases* when an item is pushed to the stack.
SP *increases* when an item is popped from the stack.

► *Example*



Suppose having an array `Arr` of 8 cells: its boundaries are `Arr[0]` and `Arr[7]`.

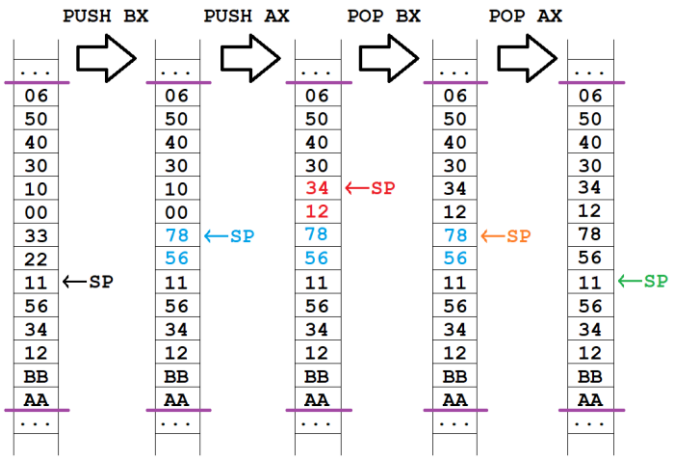
What would be the position of `Arr[9]`? It is a cell outside the domain of the array.
Is there a support to help understanding that we are out of the array? It depends.
- If a support is present (usually the OS), a segmentation error occurs!
- If there is not, a cell is invaded:
  > in case of reading, that value will be trash;
  > in case of writing, over-write happens.

8086 does not provide such support.

► *Example*

Initial values:
`<AX> = 1234  <BX> = 5678  <CX> = ABCD`

Instruction	*Corresponding action*	*New SP value*
`PUSH BX`	`[SP] ← BX`	`SP ← SP-2`
`PUSH AX`	`[SP] ← AX`	`SP ← SP-2`
`POP BX`	`BX ← [SP]=1234`	`SP ← SP+2`
`POP AX`	`AX ← [SP]=5678`	`SP ← SP+2`



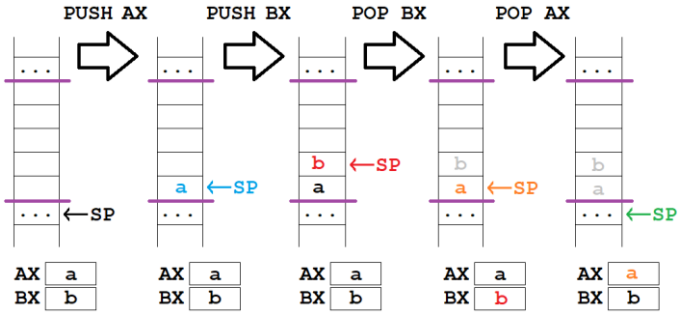
`[SP] ← BX` means "store to the cell pointed by `SP` the value stored in `BX`".

What did this group of instructions do? Comparing the final and the initial data, the contents of the registers `AX` and `BX` have been ***swapped***.

► *Example*

Initial values:
`<AX> = a  <BX> = b`

Instruction	*Corresponding action*
`PUSH AX`	`[SP] ← BX`
`PUSH BX`	`[SP] ← AX`
`POP BX`	`BX ← [SP]=b`
`POP AX`	`AX ← [SP]=a`



After the last instruction the stack is empty (it contains only trash values).
The values of the registers `AX` and `BX` are the same, but some time has passed during the execution.

The following operations are ***forbidden***:

Instruction	*Corresponding action*
`SUB SP, 2`	`SP ← SP-2`
`ADD SP, 2`	`SP ← SP+2`

Both the operations move the pointer to a new `TOS`; they respectively correspond to:
- half of "push", it is missing the part where some source is copied to the stack. This could be used to restore some previously popped values, but it is not recommended for basic users.
- half of "pop", it is missing the part where the contents of `SP` are copied to some destination. Advanced users do this to empty the stack of an element; it is not recommended for basic users.

Any stack operation requires accessing the RAM, updating the `SP` value, updating the memory content. They are quite expensive operations.

● ***Program Status Word registers***

`PSW` is a set of 16-bit registers that contain *status flags* and *control flags*. A *set* flag has value 1, *reset* is 0.

Status flags:
- ***Sign*** (`SF`): Contains the sign bit of the result of the last arithmetic operation.
- ***Zero*** (`ZF`): Set when the result of the operation is 0.
- ***Carry*** (`CF`): Set if an operation resulted in a *carry* (addition) into or *borrow* (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- ***Auxiliary carry*** (`AF`): set when some arithmetic operations generate carry after the lower half and sends it to upper half.
- ***Parity*** (`PF`): set when result has even number of 1; otherwise 0 for odd number of 1s.
- ***Equal***: Set if a logical compare result is equality.
- ***Overflow*** (`OF`): Used to indicate arithmetic overflow.

Control flags:
- ***Interrupt Enable/Disable*** (`IF`): Used to enable or disable interrupts.
- ***Supervisor***: Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions or areas of memory can be executed/accessed only in supervisor mode.

● ***Format of assembly instructions***

A statement in assembly language has this form:

Label	**Mnemonic**	**Operand(s)**	**;comment**
Optional	Opcode name or directive name or macro name	Zero or more	Optional

Label - If present, the assembler defines the label as equivalent to the address into which the first byte of the object code generated for that instruction will be loaded. The assembler replaces the label with the assigned value when creating an object program.

Mnemonic - The mnemonic is the name of the ***operation*** or function of the assembly language statement. A statement can correspond to a machine instruction, an assembler directive, or a macro.

Operand(s) - An assembly language statement includes zero or more operands. Each operand identifies an immediate value, a register value, or a memory location.
In the x86 instruction set:
- the first operand listed is the ***destination*** *operand*;
- the second operand listed is the ***source*** *operand*.

comment - A comment can either occur at the right-hand end of an assembly statement or can occupy an entire text line. The comment always begins with a *special character* (**;**)that signals to the assembler that the rest of the line is a comment and has to be ignored. A comment must be meaningful, to understand the role of the instruction in the program.

► *Example*

```
CYCLE : MOV AX, 0  ;initialize the accumulator
```

This instruction sets `AX` value as 0, initializing the accumulator later used for a sum of elements of an array.

The label can be any name; it is used because:
- it makes a program location easier to find and remember.
- it can easily be moved to correct a program. The assembler will automatically change the address in all instructions that use the label when the program is reassembled.
- The programmer does not have to calculate relative or absolute memory addresses, but just uses labels as needed.

Studying the syntax of a CPU means studying all the possible operations (studying the *instruction set*) and operands (studying the *addressing modes*); operations and operands are likely to be orthogonal.
Orthogonality is a principle by which two variables are independent of each other. In the context of an instruction set, the term indicates that other elements of an instruction are independent of (not determined by) the opcode.

● ***Type of Assembly Language Statements***

Instructions – They are symbolic representations of machine language instructions (*operation codes* or ***opcodes***). There is a one-to-one relationship between an assembly language instruction and a machine instruction. The assembler resolves any symbolic references and translates the assembly language instruction into the binary string that comprises the machine instruction.
*The **microprocessor** executes instructions at **runtime**.*

Directives (or ***commands*** or ***pseudo-instructions***) - They are statements not directly translated into machine language instructions: they are *specified actions that the **assembler** must perform doing the **compilation time*** (also called assembly process).
Examples are:
- Define constants
- Designate areas of memory for data storage
- Initialize areas of memory
- Place tables or other fixed data in memory
- Allow references to other programs

Instructions can incorporate “partial” commands.

Macro definitions - They are a section of code that the programmer writes once, and then can use many times by calling them from any point in the program. The main difference with subroutines is that, when the assembler encounters a macro call, it does a ***macro expansion*** (replaces the macro call with the macro itself).

Comments – Which were seen previously.

● *8086 Addressing Modes*

The instruction set provides various methods to ***address operands***. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other.

Addressing modes are divided in:
- inside the CPU
  - register
  - constant
- outside the CPU (RAM)
  - direct
  - indirect

There are two forms to define the constraints of addressing. If there are 2 operands, one of the two:
- should be a register (*easy form*)
- should be inside the CPU (*complete form*)

Therefore, 8086 cannot manage more than one operand from RAM!

► ***Register Addressing*** → Using register addressing, the instruction selects one or more registers which represent the operand or operands. This is the most compact addressing method, because the register addresses are encoded in the instruction.

Example

`MOV AX, BX` The content of `BX` is copied to `AX`.

```
MOV AL, CH      MOV CX, BP      MOV SI, DI
```

Forbidden: `MOV AL, BX` `MOV CX, BH`

Both operands must be of the *same data type* casting (ex. same length / size / type in bits).

The advantages of register addressing are:
- only a small address field is needed in the instruction
- no time- consuming memory references are required.

The disadvantage is the very limited address space.

► ***Immediate*** (or ***Constant***) ***Addressing*** → Some instructions have a data value encoded in the instruction so that it is available immediately as the source operand.
The immediate data can be 8 bits, 16 bits, or 32 bits, which are sign-extended to fit the size of the destination operand (two's complement form).

Example

```
MOV AL, 23
MOV BX, 11100B (binary)
```

```
ADD AX, 2
MOV AH, 99
MOV AX, 1A23H (hex)
```

Forbidden:

`MOV 23, AL` value of a constant cannot change

`MOV AH, 3000` source size is larger than destination

The advantage is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle delay.
The disadvantage is that the size of the number is restricted to the size of the address field, which is small compared with the word length.

When a value is larger than the representation in the destination, an error occurs (i.e. 3000 to 8-bit).

A command is something that the compiler is expected to take care of: for example, imposing to consider some value as *binary* (`11100`**`B`**) or *hexadecimal* (`1A23`**`H`**).

► ***What is the difference between command and instruction?***

`VAR DW 93` Declaration of a variable `VAR` (define name, type, content) that is processed by the compiler (*this is a command*).

`MOV AX, VAR` Compiler translates `VAR` in the corresponding address, defined previously (*this is an instruction*).

The initialization of `VAR` at the beginning of the program consists in:
- reserving 2 bytes in the memory
- setting up the equivalence between `VAR` and the cell address (i.e. 1234_H)
- storing the value (`93`) to the cell

Do not initialize a variable to some value, because that value is true only at the beginning of the program: one cannot know if later, during the execution of the program, such cell still contains the value.

It is better to initialize ***variables*** using this kind of definition (with a ***random initialization***): **`VAR DW ?`**

Instruction	***Command***
Executed by *microprocessor*	Processed by *compiler*
At *runtime*	At *compile time*
Considering a variable `VAR`, what is available at respective time?	
Can access *content*	Cannot access *content*
Cannot access *address*	Can access *address*

Which is doing what? At what time?

Variable	Address	Content
`VAR`	1234_H	`93`

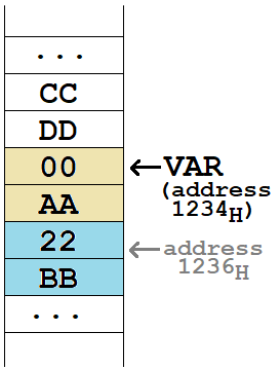
At *compile time*, the *compiler* has only access to the *address* of the variable and not the content (i.e. in case of loop instructions, the content changes at runtime).
At *runtime*, the *microprocessor* is given the address of the variable (already translated by the compiler) and can access and modify the *content*.

► ***Direct Addressing*** → One of the operands is in the RAM and the address field contains the effective address of the operand.
The memory operand can be either the source or the destination operand and is addressed by a segment selector and a 32-bit or 64-bit offset that is part of the instruction.
The combination of the selector and the offset generates an effective address that points *directly* to the memory operand. The segment selector can be any of the segment registers (`CS`, `DS`, `SS`, `ES`).

Example `MOV AX, [1A33D4H]`

Example
```
VAR DW ?
MOV AX, VAR+2
```

In the second instruction:
1. `VAR` is handled by the compiler and replaced with the corresponding address (`1234H`)
2. `VAR+2` is in the operand field: + is not an instruction but a way to identify the operand; + is managed by the compiler
3. `MOV` is the operation, asking to move `AX` with the content of the variable at address $1234_H+2=1236_H$
4. `<AX>=BB22` , the content of element with the address 1236_H is copied to `AX`.



• *8086 Array Declaration*
To declare an array, we specify:
- the name of the array
- the dimension of the array
- the size of every element
- the special system words **`DUP`** (*duplicated*).

Size of element can be **`DB`** (for *data byte*) or **`DW`** (for *data word*, which means 2 bytes).

Example
```
ARR_A   DB  1, 2, 3, 4, 5, 6, 7, 8, 10
ARR_B   DB  DIM  DUP(?)
ARR_C   DB  10 DUP(?)
```

`ARR_A` is an array with initial values.
`ARR_B` is an array of dimension `DIM` (a constant) without initial values.
`ARR_C` is an array of 10 bytes without initial values.

To initialize an array to all 0 we can:
- set each element of the array to 0, one by one;
- create a loop that sets each element to 0.

Suppose `ARR_C` first element has address `1001H`:
```
MOV ARR_C, 0        set first element to 0
MOV ARR_C+1, 0      set second element to 0
MOV ARR_C+2, 0      set third element to 0
…
MOV ARR_C+9, 0      set tenth (last) element to 0
```

► ***Indirect Addressing*** → It is used when a register contains the address of the data, rather than the actual data to be accessed.
With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range: one solution is to make the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand.

This method of addressing uses general-purpose registers (i.e. `[BX]`, `[EAX]`, `[EBX]`…) or pointer and index registers (i.e. `[BP]`, `[SI]`, `[DI]`) within brackets: the brackets are to be interpreted as meaning *contents of*; inside the brackets there is a *pointer* to the data within a specific segment.

Example
```
MOV AX, [BX]
MOV ARR_C[BX], 0
```

The advantage is that for a word length of N, an address space of 2^N is now available.
The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

► *Test 2020-02-10 – Exercises 2, 7*

Exercise 2

We have a ***full associative mapping*** cache memory with globally 65536=2^{16} lines. Each line is hosting 32 data. How many bits are necessary for the `TAG` for a 24 bits address bus?

A) 3
B) 19
C) 21
D) None of the previous answers because…

Solution: B

The number of lines is irrelevant because of the FAM.
`Entries/Data` size = 32 = 2^5
The 24 bits A-BUS supports a block divided in two fields:
- Offset = $\log_2$(`Entries` size) = $\log_2 32 = \log_2 2^5 = 5$ bits
- `TAG` = Address – Offset = 24–5=19 bits

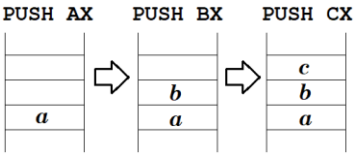
Exercise 7

Please provide detailed explanations about what is the final value stored in `AX`, `BX`, `CX` after the execution of the following assembly code. Assume that at the beginning `AX` stores *a*, `BX` *b* and `CX` *c*, with 0<*a*<*b*<*c*<1000.

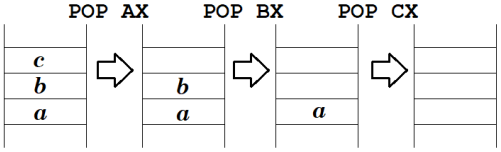
```
PUSH AX
PUSH BX
PUSH CX
ADD AX, BX
ADD BX, CX
ADD CX, AX
POP AX
POP BX
POP CX
```

Solution:

<AX>=*a* <BX>=*b* <CX>=*c*



Instruction	*Corresponding action*
`ADD AX, BX`	`AX` ← *a*+*b*
`ADD BX, CX`	`BX` ← *b*+*c*
`ADD CX, AX`	`CX` ← *c*+*a*+*b*



Instruction	*Corresponding action*
`POP AX`	`AX` ← `c`
`POP BX`	`BX` ← `b`
`POP CX`	`CX` ← `a`

Final values:
<AX>=*c* <BX>=*b* <CX>=*a*

► *Examples of addressing modes*

Register addressing `MOV AX, BX`
`AX` is register #1, `BX` is register #2

Constant addressing `MOV AX, 2`
`AX` is a register, `2` is the value of a constant

Indirect addressing `MOV AX, [BX]`
`AX` is a register, `[BX]` is an address referring to the register #2 `BX` (the content of `BX` is a pointer to the first element of an array).

The microprocessor at runtime receives *the coded instructions* (all the parts of the program are translated to code names and addresses). It does not know anything of the original code.
The three `MOV` may seem all equal, but they are not: *they are differently encoded by the compiler* according to the interpretation of the operands.

Instruction	MOV	AX,	BX
Code name	**$3F_H$**	01_H	02_H
Instruction	MOV	AX,	2
Code name	**$4F_H$**	01_H	02_H
Instruction	MOV	AX,	[BX]
Code name	**$5F_H$**	01_H	02_H

Each `MOV` is coded *according to the addressing mode* of the respective instruction and operands.
Operands does not carry any information of how to be interpreted: the microprocessor is told what to do by the *different code of the operation*. To each operation is associated a certain number of different codes corresponding to how many different addressing modes it supports.

The compiler associates the right operation code to the instruction according to the operands' definitions.
*Data has to be **declared** and **defined*** to be later interpreted correctly by the compiler.

Declaration contains: name of the variable, type of value it holds and the initial value if any it takes.
Definition says where the variable gets stored.

In the case of `MOV AX, 2`, the value of the constant `2` becomes integral part of the code of the instruction. It is not a data variable: it does not need to be stored in the memory and cannot change because it is embedded into the *instruction code*, which *by definition cannot change*. Constant are stored directly into the code.

As seen in ⇨*Unified versus split caches* lecture, at runtime instruction are only read from CPU, while data can be also written.

Where are variables stored? In the *RAM*.
How does the microprocessor access a variable?
Through its *address*: the address of a variable cannot change during runtime, while its content can.

► *Examples*

```
1.      MOV AX, VAR
2.      MOV AX, AX
3.      MOV BH, AX
4.      MOV [BX], 3
5.      MOV CX, 1F00H
6.      MOV BL, 10     MOV BL, AH
```

1. What is `VAR` in the first instruction? It can be anything (variable, string, array…), it depends on its *declaration*: the statement therefore can be either valid or invalid.
In case the declaration was `VAR DB 9`, we need to consider that:

- `DB` size is 8 bits
- `AX` size is 16 bits

Therefore, an error occurs. We must look not only to the instruction, but also to the operands and their *definitions*.

2. What is happening? The content of `AX` is copied to `AX`, while some time passes. Initial and final values are the same.

3. The size of the destination `BH` (8 bits) is smaller than the source `AX` (16 bits): an error occurs.

4. The compiler would have problems to understand and translate, though it may be an acceptable instruction.

Brackets usage:

- `[BX]` is used to address a cell into memory
- <BX> means "*the content of*" `BX`

Which kind of `MOV` operation is this one?

- It is an *indirect addressing* (*first operand*) `MOV`
- Is it a MOV acting on *byte* (1B) or *word* (2B) size?

In the *byte* case, the content of the cell is set to 3.
In the *word* case, the content of the higher part is set to 3 and the lower part to 0.

The compiler does not know if it is a byte or word type!
The programmer has to provide this information.

Therefore, the correct statement would be the following:

1. `MOV BX, 3`
The compiler already knows `BX` is a word, nothing else is needed.

2. `MOV` **`BYTE PTR`**`[BX], 3`
3. `MOV` **`WORD PTR`**`[BX], 3`

`BYTE PTR` and **`WORD PTR`** are ***pointer directives***, which force the compiler to *treat the operand following them as a pointer of specified type* (respectively byte and word size).

5. `1F00H` is a constant in hexadecimal format occupying 16 bits. The destination size is 16 bits. Everything is correct.

6. In the first instruction, `10` is considered a 10 in base 10, its size is 8 bits as the destination operand, so everything is correct.

Translating `10` to hexadecimal we obtain `A`, but we need to add `H` at its end, therefore it becomes `AH`.

Consider `MOV BL, EFFEH`
We do not know `EFFEH` data type, therefore we cannot understand say if

- it is a hexadecimal number
- this is a valid operation or not.

Going back to `MOV BL, AH`
Is `AH` the higher part of register `AX`?
Is `AH` the hexadecimal representation of 10_{10}?

The compiler will read `AH` as a register and `EFFEH` as a name of a variable (string of characters).

How to make the compiler not misunderstand the instruction?
To pass a hexadecimal number correctly to the compiler:

- set the final `H`;
- if it starts with a digit (0 to 9 value), it will be interpreted correctly;
- if it starts with a letter, we need to add a digit (i.e. 0) to let the compiler know it is a number.

`123EH` is already ok
`EFFEH` becomes `0EFFEH` → `MOV BL, 0EFFEH`
`AH` becomes `0AH` → `MOV BL, 0AH`

● *Main Instructions*

We will list only a few instructions, just to know what happens inside a CPU when they are executed.
Opcodes depend on the machine.
We can categorize instruction types as follows.

Data transfer: Movement of data into or out of register and or memory locations.
- `MOV AX, BX` → moves the contents of `BX` into `AX`.
- `PUSH AX` → pushes the `AX` register contents at the top of memory stack.
- `POP AX` → retrieves the contents of the top of the memory stack and stores them inside `AX`.
- `XCHG AX, BX` → exchanges the contents of two registers (*swap*)
- `PUSHF "PSW" / POPF "PSW"` → Pushes or pops a `PSW` flag register on / off the stack.
- `PUSHA / POPA` → Pushes or pops all general-purpose registers on / off the stack.

Data movement: *I/O* instructions.
- `IN AX, DX` → Data read from port is stored in `AX`.
- `OUT AX, DX` → Data read from `AX` is sent to port.

Flow control: *Loop* and *branch* instructions.
- `CMP AX, BX` → *Compare*: subtracts `BX` from `AX`; *does not save result* but updates the *flags* accordingly.
- `Jxxx "label"` → Conditional jump: if the condition is true, then go to the label; else execute the following instruction (next in memory).
- `JMP "label"` → Unconditional jump: go to the destination

Data processing: *Arithmetic* and *logic* instructions.
- `ADD AX, BX` → adds content of `BX` and `AX`, stores the result into AX.
- `SUB AX, BX` → subtracts the content of `BX` from `AX`, stores the result into AX.
- `ADC AX, BX` → same as `ADD`, if `CF`=1adds an extra 1.
- `ADC AX, BX` → same as `SUB`, if the `CF`=1 subtracts an extra 1.
- `INC AX` → adds 1 to `AX`.
- `DEC AX` → subtracts 1 from `AX`.
- `NEG AX` → Negate: inverts the sign of an integer `AX` (two's complement).
- `MUL BX` → Unsigned Multiply (pure binary) of `AX` by `BX`, stores the result into AX.
- `IMUL BX` → Signed Multiply (two's complement) of `AX` by `BX`, stores the result into AX.
- `DIV BX` → Unsigned Division (pure binary) of `AX` by `BX`.
- `IDIV BX` → Signed Division (two's complement) of `AX` by `BX`.

- `NOT AX` → complements every bit of `AX` (1's complement).
- `AND AX, BX` → logical `AND` between `AX` and `BX`. The result is stored into `AX`.
- `OR AX, BX` → logical `OR` between `AX` and `BX`. The result is stored into `AX`.
- `XOR AX, BX` → exclusive `OR` between `AX` and `BX`. The result is stored into AX.
- `TEST AX, BX` → logical `AND` between `AX` and `BX`; does not save result but updates the flags accordingly.

Bit manipulation
- `SHL / SAL AX, N` → shift `N` bits of `AX` to the left. (`AX*2`^{N}).
- `SHR AX, N` → shift `N` bits of `AX` to the right. (`AX/2`^{N}).
- `SAR AX, N` → shift `N` bits of `AX` to the right, sign bit is replicated. (`AX*2`^{N}).
- `ROL AX, N` → rotates `AX` bits to the left by `N` times; all data pushed out the left side re-entering on the right.
- `ROR AX, N` → rotates `AX` bits to the right by `N` times; all data pushed out the right side re-entering on the left.

`CF` contains the value of the last bit shifted or rotated out.

- `RCL AX, N` → rotates `AX` bits to the left by `N` times; all data pushed out the left side re-entering on the right.
- `RCR AX, N` → rotates `AX` bits to the right by `N` times; all data pushed out the right side re-entering on the left.

`CF` holds the last bit rotated out.

● *Data transfer opcodes*

The most fundamental type of machine instruction is the data transfer instruction.
The data transfer instruction must specify:
- the *location of the source and destination operands*. Each location could be memory, a register, or the top of the stack.
- the *length of data* to be transferred.
- the *mode of addressing* for each operand, as with all instructions with operands.

In terms of processor action, data transfer operations are perhaps the simplest type.
If *both source and destination are registers*, then the processor simply causes data to be transferred from one register to another; this is an operation internal to the processor.
If *one or both operands are in memory*, then the processor must perform some or all of the following actions:
1. Calculate the memory address, based on the addressing mode.
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

`MOV AX, BX` → *moves* the contents of `BX` into `AX`.
Corresponding action: `AX ← BX`
Constraints in addressing, values, data size have already been seen previously.

`POP AX` → *retrieves* the contents of the top of the memory stack and stores them inside `AX` (destination).
Corresponding action: `AX ← TOS`

`PUSH AX` → *pushes* the `AX` register contents (source) at the top of memory stack.
Corresponding action: `TOS ← AX`

Constraints of `PUSH` and `POP` are:
- the *elementary element* is two cells of 8 bits each, therefore they can only manage 16 bits elements at a time (no 8 bits)
- they cannot manage *constants*

`XCHG AX, BX` → exchanges the contents of two registers (*swap*)
Corresponding action: `<AX> ⇄ <BX>`
`XCHG` can be replaced with some instructions:
- advantage: its opcode is not always found in assembly language
- disadvantage: loss in readability

Suppose `<AX>=a , <BX>=b.`
Equivalent algorithms to `<AX> ⇄ <BX>` would be:

	Solution 1	*Solution 2*	*Solution 3a*
1	`PUSH BX`	`TMP DW ?`	`MOV CX, AX`
2	`PUSH AX`	`MOV TMP, AX`	`MOV AX, BX`
3	`POP BX`	`MOV AX, BX`	`MOV BX, CX`
4	`POP AX`	`MOV BX, TMP`	

	Solution 3b	*Solution 4*	*Solution 5*
1	`PUSH CX`	`PUSH AX`	`XOR AX, BX`
2	`MOV CX, AX`	`MOV AX, BX`	`XOR BX, AX`
3	`MOV AX,BX`	`POP BX`	`XOR AX, BX`
4	`MOV BX, CX`		
5	`POP CX`		

Solution #	*Memory Accesses (speed)*		*Extra memory usage*
1	`4 RAM`		
2	`2 RAM, 1 CPU`	*<1*	*>1*
3a	`3 CPU`	*<2*	*< 2*
3b	`2 RAM, 5 CPU`	*>3a*	
4	`2 RAM, 3 CPU`	*>2*	
5	`0 RAM, 3 CPU`	*<2*	–

Solution 1 is slow.

Solution 2 requires to define an additional variable.

Solution 3a is wrong! Even if it is the most straight forward, `CX` is not always available, so we have to assume everything is busy and cannot overwrite (if we do it, the previous content of `CX` is lost).

Solution 3b solves the problem of *3a* by ***saving the original content*** of `CX` ***to the stack and then restoring it later***.

Solution 4 is a mix between *1* and *3a*; its speed is approximately the same as *2* but it does not require the declaration of a variable. The disadvantage is that *4* is slightly slower than *2*: implementation of the stack requires the usage of `SS` and `SP` registers + memory access + update of `SP` value.

Solution 5 is correct; its calculus is the following:

Instruction	*Corresponding action*
`XOR AX, BX`	`AX ← a⊕b`
`XOR BX, AX`	`BX ← b⊕AX = b⊕(a⊕b) = a`
`XOR AX, BX`	`AX ← (a⊕b)⊕a = b`

Because `b⊕b = a⊕a = 0`.
Advantages: *5* is faster and requires less space than *2*.
Disadvantages: the readability as a group is lower.

`PUSHF "PSW" / POPF "PSW"` → Pushes or pops a `PSW` flag register on / off the stack.
Corresponding action: `TOS ← "PSW"` / `"PSW" ← TOS`
`"PSW"` has to be replaced with the wanted 16-bit flag register (`SF`, `CF`, `OF` … see ⇨*Program Status Word registers* lecture).

`PUSHA / POPA` → Pushes or pops all general-purpose registers on / off the stack.
Registers are processed in the following order:
- `PUSHA` → `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI`, `DI`. The value of `SP` is the value before the actual push of `SP`.
- `POPA` → `DI`, `SI`, `BP`, `SP`, `DX`, `CX`, `AX`. The `SP` value popped from the stack is actually discarded.

`IN AX, DX` → Data read from the port is stored in `AX`.
`OUT AX, DX` → Data read from `AX` is sent to the port.
The port number is specified in `DX`; if in the range of `0-255`, it can be specified as an immediate.
These opcodes are used for accessing peripheral devices when configured in isolated I/O mode.

► *Test 2019-09-19 – Exercises 2, 6, 7*

Exercise 2

We have a ***full associative mapping*** cache memory with 2^{10} lines. Each line is hosting 32 data. How many bits are necessary for the `TAG` for a 32 bits address bus?
A) 24
B) 18
C) 6
D) None of the previous answers because…
Solution: D

The number of lines is irrelevant because of the FAM.
`Entries/Data` size = $32 = 2^5$
The 32 bits A-BUS supports a block divided in two fields:
- Offset = $\log_2$(`Entries` size) = $\log_2 32 = \log_2 2^5 = 5$ bits
- `TAG` = Address – Offset = 32–5=27 bits

Exercise 6

Please shortly explain what is the Manchester encoding, how it works and why it is useful. Please provide significant and clear picture(s).

Solution:
Manchester encoding is used to solve the synchronization problem of reading data from Hard Drives: clock and data are encoded together as a single signal, saving one line (the other line is voltage/ground). This also avoids reserving a track on the disk for the clock only.
It considers transitions low-to-high and high-to-low as values 0 and 1. Decoding takes the signal, extracts the clock through a filter and uses it to sample and return information unpacked. To synch signal and clock for the sampling, either there is a delay between signal and sampler or jamming bits are added to the signal.
See ⇨*Manchester encoding* lecture.

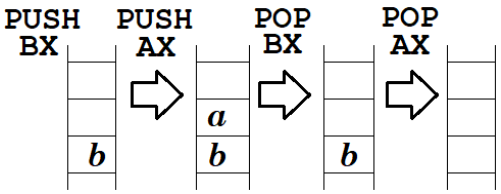
Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of `AX` and `BX`. Assume `AX` and `BX` storing values represented in two's complement.

```
PUSH BX
PUSH AX
POP BX
POP AX
SUB AX, BX
```

Solution:

`<AX>=a` `<BX>=b`



`AX←b` `BX←a`

`AX←b-a`

Final values: `<AX>=b-a` `<BX>=a`

► Test 2019-07-08 – Exercises 2, 3, 4, 6, 7

Exercise 2

We have a ***full associative mapping*** cache memory with 2^{16} lines. Each entry is hosting 32 data. How many bits are necessary for the `TAG` for a 28 bits address bus?

A) 5
B) 12
C) 23
D) None of the previous answers because…

Solution: C

The number of lines is irrelevant because of the FAM.

`Entries/Data` size = $32 = 2^5$

The 28 bits A-BUS supports a block divided in two fields:

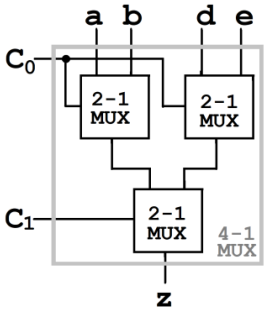
- Offset = $\log_2$(`Entries` size) = $\log_2 32 = \log_2 2^5 = 5$ bits
- `TAG` = Address – Offset = 28–5=23 bits

Exercise 3

The implementation of a 4-1 multiplexer following the hierarchical design technique…

A) Requires four 2-1 multiplexers on one level
B) Requires three 2-1 multiplexers on one level
C) Requires four 2-1 multiplexers on two levels
D) Requires three 2-1 multiplexers on two levels
E) None of the previous answers because…

Solution: D



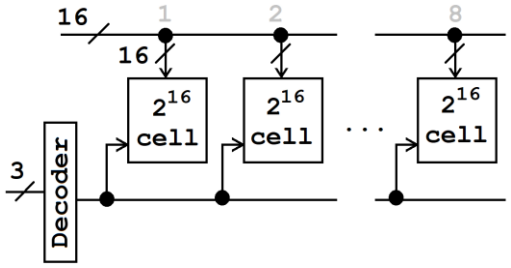
Exercise 4

We have 8 banks of memory of $\mathbf{2^{16}}$ cells. What is the size of the memory and the type of decoder used?

A) 2^{24} and 8-to-256
B) 2^{13} and 3-to-8
C) 2^{13} and 8-to-3
D) None of the previous answers because…

Solution: D

The total memory of $8=2^3$ banks having 2^{16} cells each is a $2^{16}\times 2^3 = \mathbf{2^{19}}$ memory. We need a **3-to-8** decoder, with the 3 missing bits of the 19-bit address bus entering to the decoder.

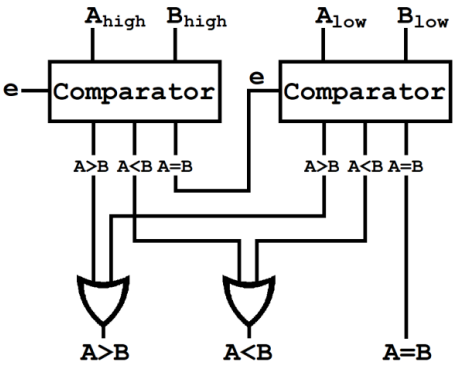


Exercise 6

Please shortly explain and provide a significant picture about how to design an architecture for comparing two 16-bits binary numbers A and B using 4-bits binary number comparators plus some other logical gates.

Solution:

Divide 16 bits of A and B into two groups of 8 bits each and apply this circuit. Then, as 8-bits comparators are not available here, each 8-bits comparator has to be replaced by the same architecture on the left.



Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of `AX`. Solving (and demonstrating) the problem for the general case of any value of `AX` and `BX` will bring 5 points; solving the problem for the specific numerical case `AX=128` and `BX=255` (or any other value) will bring 4 points.

```
OR AX, BX
NOT AX
AND AX, BX
```

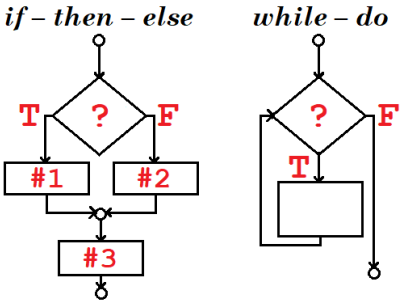
Solution: *(general)*
`<AX>=a` `<BX>=b`

Instruction	*Corresponding action*
`OR AX, BX`	`AX` ← $a+b$
`NOT AX`	`AX` ← $\overline{a+b}=\overline{a}\cdot\overline{b}$ (for De Morgan)
`AND AX, BX`	`AX` ← $\overline{a}\cdot\overline{b}\cdot b = 0$

Final values: `<AX>=0` `<BX>=b`

• Flow control opcodes

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to *update the program counter* to contain the address of some instruction in memory.



► *Branch (or jump) instructions*

They have as one of their operands the address of the next instruction to be executed.
Most often, the instruction is a ***conditional branch*** *instruction*. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual).
A branch instruction in which the branch is always taken is an ***unconditional branch***.

To implement the *check block* we need two operations.
1) Compare two operands.
2) Operate a jump based on the value of "flags".

Suppose the condition is `AX=BX`. The code would be:

```
CMP AX, BX
JE AX EQ BX     ; jump to AX EQ BX if AX=BX
#2              ; if we are here, AX≠BX
...
JMP CONT
AX EQ BX: #1    ; if we are here, AX=BX
...
CONT: #3
```

In the comment it is explained where the jump `JE` should go. **`AX EQ BX`** is the label name of the first instruction of block **`#1`**. If the condition of the *conditional jump* `JE` is not true, its *following* instruction (next in memory, so block **`#2`**) is executed. At the end of block **`#2`** I need to place an *unconditional jump* `JMP` instruction to skip block **`#1`** and go to block **`#3`**.
Then I place the label **`AX EQ BX`** and corresponding block **`#1`**.
At the end of block **`#1`** I can place the label **`CONT`** and corresponding block **`#3`**.

`CMP AX, BX` → ***Compare***: subtracts source from destination; it *does not save result* but updates the *flags* accordingly.
Flags can subsequently be checked for conditions.
Modifies Flags: `AF`, `CF`, `OF`, `PF`, `SF`, `ZF`
Corresponding action: `"flag"` ← `AX-BX`
Constraint: the two operands must have the same length; cannot compare two memory allocations.

For flags, see ⇨*Program Status Word registers* lecture.

`Jxxx "label"` → ***Conditional jump***: if the condition is true, then go to the label; else execute the following instruction (next in memory).
`xxx` is a suffix that specifies the flag condition.

There are three groups of conditional jumps; their condition is related to:
- a single flag;
- the result of a comparison;
- the content of `CX` register.

The condition is related to a single flag
Formatting: **`Jxxx "label"`**
There are two kinds of instructions:
- jump is the flag is 1;
- jump is the flag is 0.

Opcode	*Meaning*	*Flag*
JZ	Jump if Zero	ZF=1
JNZ	Jump if Not Zero	ZF=0
JS	Jump if Signed (S)	SF=1
JNS	Jump if Not Signed (S)	SF=0
JO	Jump if Overflow (S)	OF=1
JNO	Jump if Not Overflow (S)	OF=0
JC	Jump if Carry	CF=1
JNC	Jump if Not Carry	CF=0
JP or **JPE**	Jump if Parity Jump if Parity Even	PF=1
JNP or **JPO**	Jump if No Parity Jump if Parity Odd	PF=0

(S) = signed

The condition is related to the result of a comparison
These conditional jump instructions also verify a condition on the status flags; in general, this condition involves more than one flag.
There are two kinds of instructions depending on the comparison between:
- signed numbers (*two's complement*);
- unsigned numbers (*pure binary*).

The comparison between characters is associated to the one between unsigned numbers.

Formatting: **`CMP "dest", "src"`**
`Jxxx "label"`

In the case of *comparison between signed numbers*:

Opcode	*Corresponding action*	*Flag*
JL or **JNGE**	Jump if *destination* < *source*	SF != OF
JG or **JNLE**	Jump if *destination* > *source*	ZF=0 and SF=OF
JLE or **JNG**	Jump if *destination* ≤ *source*	ZF=1 or SF!=OF
JGE or **JNL**	Jump if *destination* ≥ *source*	SF=OF
JE	Jump if *destination* = *source*	ZF=1
JNE	Jump if *destination* ≠ *source*	ZF=0

In the case of *comparison between unsigned numbers*:

Opcode	*Corresponding action*	*Flag*
JB or **JNAE**	Jump if *destination* < *source*	CF=1
JA or **JNBE**	Jump if *destination* > *source*	CF=0 and ZF=0
JBE or **JNA**	Jump if *destination* ≤ *source*	CF=1 or ZF=1
JAE or **JNB**	Jump if *destination* ≥ *source*	CF=0
JE	Jump if *destination* = *source*	ZF=1
JNE	Jump if *destination* ≠ *source*	ZF=0

`L` → Less **`G`** → Greater **`E`** → Equal
`B` → Below **`A`** → Above

The condition is related to the content of `CX` register
`JCXZ "label"` → if the content of `CX` is `0`, then go to the label; else execute the following instruction (next in memory).

Summary of all conditional jump instructions

Opcode	*Jump condition*
JB or **JNAE**	CF=0 and ZF=0
JA or **JNBE**	CF=0
JAE or **JNB**	CF=1
JBE or **JNA**	CF=1 or ZF=1
JC	CF=1
JCXZ	CX=0
JE	ZF=1
JG or **JNLE**	ZF=0 and SF=OF
JGE or **JNL**	SF=OF
JL or **JNGE**	(SF=1 and OF=0) or (SF=0 and OF=1)
JLE or **JNG**	ZF=1 or (SF=1 and OF=0) or (SF=0 and OF=1)
JNC	CF=0
JNE	ZF=0
JNO	OF=0
JNP or **JPO**	PF=0
JNS	SF=0
JNZ	ZF=1
JO	OF=1
JP or **JPE**	PF=1
JS	SF=1
JZ	ZF=1

`JMP "label"` → ***Unconditional jump***: go to the destination (expressed as a *label* or *indirect address*); this jump is always executed, without any condition-check.

Unconditional jump `JMP` can do:
- *direct jumps*
- *indirect jumps*

In ***direct jumps*** it is specified the address where to jump; there are three types of them:
- ***short*** – the *difference* (gap) between the *actual value* of `IP` register and the *offset of the destination instruction* can be contained in a *byte*;
- ***near*** – the difference between the actual value of `IP` register and the offset of the destination instruction can be contained in a *word*;
- ***far*** – the address of the destination instruction (offset and segment register `CS`) can be contained in a *doubleword* (32 bits = 4 bytes)

In ***indirect jumps*** it is not specified the address where to jump, but a cell which content is the address where to jump. Such address can be stored in a register, a variable, a label or a pointer to a cell.

► *Examples of unconditional jumps*
1. `JMP PIPPO`
2. `JMP VAR`
3. `JMP BX`
4. `JMP [BX]`

1. `PIPPO` is a label; this operation jumps to the instruction that has been labeled "`PIPPO`" and continues from there the execution.

2. `VAR` is a variable identifier, previously declared as following: `VAR DW (1234H)`. The content of `VAR` is an address: this operation jumps to the instruction which address corresponds to `1234H`.

3. `BX` is a register. The content of `BX` is an address: this operation jumps to the instruction which address is stored in `BX`.

4. `[BX]` is a ***pointer*** to a certain cell. The content of the cell pointed by `[BX]` is an address: this operation jumps to the instruction which address is stored in such cell. This is the most common jump instruction.

Therefore, all these operands are *considered as addresses to jump to* by the compiler when it translates the code.

► *Example*
Suppose the condition is `AX≥BX`, both in pure binary representation. The code would be:

```
CMP AX, BX
JAE AX GE BX ; if AX≥BX
#2 ; if we are here, AX<BX
...
JMP CONT
AX GE BX: #1 ; if AX≥BX
...
CONT: #3
```

If other instructions are executed between `CMP` and `Jxxx`, the jump will consider the flags at the moment of its execution (and not at the time just after `CMP`). This produces ***errors***, therefore `CMP` and `Jxxx` must follow each other and not be split.

What are the differences between `MOV` and `JMP`/`Jxxx`?

	MOV	**JMP**	**Jxxx**
Formatting	`MOV AX, BX`	`JMP target`	`CMP AX, BX` `Jxxx label`
Operation	*Copies* the contents of the source to the destination.	*Transfers* the controller to the destination, changing the execution flow of the program.	

What are the differences between `SUB` and `CMP`?

	SUB	**CMP**
Formatting	`SUB AX, BX`	`CMP AX, BX`
Operation	Subtracts source from destination; stores the result to the destination.	Subtracts source from destination; *does not save result* but updates the *flags* accordingly.

● *Arithmetic opcodes*

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers. Often, they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions:
- Absolute: Take the absolute value of the operand.
- Negate: Negate the operand.
- Increment: Add 1 to the operand.
- Decrement: Subtract 1 from the operand.

`ADD "dest", "scr"` → ***Addition*** of the contents of source and destination, store into destination.
Corresponding action: `AX ← AX+BX`

`SUB "dest", "scr"` → ***Subtraction*** of the contents of source and destination, store into destination.
Corresponding action: `AX ← AX-BX`

Modifies Flags: `AF`, `CF`, `OF`, `PF`, `SF`, `ZF`

Constraints:
- same data type;
- destination must be a register or memory location;
- source must be a register, memory location or immediate value;
- cannot be both memory locations.

The addition algorithm does not look at the representation of the operands (numbers can be both pure binary and two's complement).

Carry and ***overflow detection*** systems are present: each time any sum is done, **`CF`** and **`OF`** flags are updated.
Checks must be done just after the `ADD` trough these conditional jumps:

- Pure binary	carry	`JC`	`CF=1`
- Two's complement	overflow	`JO`	`OF=1`

`ADC "dest", "scr"` → *ADd with* ***Carry***: adds the contents of source, destination and value of `CF`; the result is stored into destination.
If `CF` is set (there is a carry), a 1 is added to the destination; else it is the same as `ADD`.
Corresponding action: `AX ← AX+BX+CF`

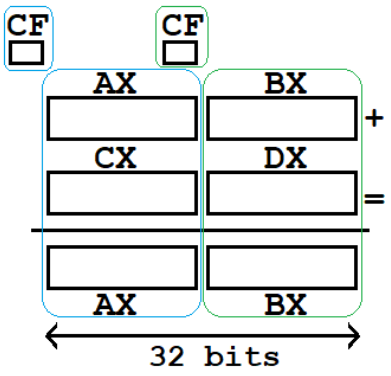
`SBB "dest", "scr"` → *SuBtract with* ***Borrow***: subtracts the contents of source from destination; the result is stored into destination.
If `CF` is set (there is a borrow), a 1 is subtracted from the destination; else it is the same as `SUB`.
Corresponding action: `AX ← AX-BX-CF`

Modifies Flags: `AF`, `CF`, `OF`, `PF`, `SF`, `ZF`

`ADC` and `SBB` are helpful with doubleword (32 bits) sums: 8086 supports at maximum 16-bit values, therefore we have to split larger operations in blocks.

►*Example – addition of two 32-bits pure binary nums*

```
ADD BX, DX
ADC AX, CX
JC
```



Lower bits (`BX`, `DX`) and higher bits (`AX`, `CX`) are added; using `ADC` the carry of the lower part is added correctly to the higher part's LSB; the results are stored in the destinations `AX` (higher bits) and `BX` (lower bits).
After the two additions, we use conditional jump `JC` to check the overflow, in case the higher bits sum has a carry (`CF=1`).

`INC "dest"` → ***Increment destination by one***
Corresponding action: `AX ← AX+1`

`DEC "dest"` → ***Decrement destination by one***
Corresponding action: `AX ← AX-1`

Destination can be a register or memory location.
Modifies Flags: `AF`, `OF`, `PF`, `SF`, `ZF` (not `CF`)

`NEG "dest"` → ***Negate***: inverts the sign of an integer, assuming that the destination is a number with two's complement represented numbers.
Corresponding action: `AX ← -<AX>`

Destination can be a register or memory location.
Modifies Flags: `AF`, `CF`, `OF`, `PF`, `SF`, `ZF`

How to replace `NEG`? `DX ← -<DX>`

1.
```
PUSH AX              ; save AX content
MOV AX, 0            ; set AX to 0
SUB AX, DX           ; AX=0-DX
MOV DX, AX           ; DX=AX=-DX
POP AX               ; restore AX
```
2. Complement `DX` (switch all 1 and 0); add 1 to LSB:
   a.
   ```
   NOT DX
   ADD DX, 1
   ```
   b.
   ```
   XOR DX, 0FFFFH
   ADD DX, 1
   ```

`ADD DX, 1` can be replaced by `INC DX`.

CBW → *Convert Byte to Word*: Converts byte in AL to word value in AX by extending sign of AL throughout register AH. It is helpful in case of sum between a byte number and a word number.

Example:
```
MOV AL, VAL ; copy VAL to AL
CBW         ; convert
ADD SI, AX  ; SI = SI+AX
```

MUL "src" → ***Unsigned Multiply (pure binary)*** of the accumulator by the source.
Corresponding action: AX ← AX*"SRC"

IMUL "src" → ***Signed Multiply (2's complement)*** of accumulator by "src" with result placed in the accumulator.
Corresponding action: AX ← AX*"SRC"

Modifies Flags: CF, OF (AF, PF, SF, ZF undefined)
If "src" is a byte value, it is multiplied by AL and the result stored in AX.
If "src" is a word value, it is multiplied by AX and the result is stored in DX:AX (16 bits of the higher part in DX, 16 bits of the lower part in AX).

DIV "src" → ***Unsigned Integer Division (pure binary)*** of accumulator by source.
Corresponding action: AX ← AX/"SRC"

IDIV "src" → ***Signed Integer Division (2's complement)*** of accumulator by source.
Corresponding action: AX ← AX/"SRC"

Modifies Flags: (AF, CF, OF, PF, SF, ZF undefined)
If "src" is a byte value, AX is divided by "src": the quotient is placed in AL and the remainder in AH.
If "src" is a word value, DX:AX is divided by "src"; the quotient is stored in AX and the remainder in DX.

► *Note: MUL, IMUL, DIV, IDIV are not requested for the exam* ◄

Multiplication and division by a constant
With pure binary numbers, it is asked not to use MUL and DIV but to write an ad hoc code to do the multiplication or division.

• Bit manipulation opcodes
These opcodes can be classified in two groups:
- Boolean logic operations
- Shift and rotate operations

► *Boolean logic instructions*
They implement basic logical operations: AND, OR, XOR, NOT, TEST (logical compare).

NOT "dest" → ***One's Complement Negation (Logical NOT)*** inverts the bits of the "dest" operand forming the 1st complement.
Corresponding action:
```
AL ← !AL
AL:  01110001
!AL: 10001110
```

AND "dest", "src" → ***Logical AND*** of the two operands, replacing the destination with the result.
Modifies Flags: CF, OF, PF, SF, ZF (AF undefined)

OR "dest", "src" → ***Inclusive Logical OR*** of the two operands returning the result in the destination. Any bit set in either operand will be set in the destination.
Modifies Flags: CF, OF, PF, SF, ZF (AF undefined)

XOR "dest", "src" → ***Exclusive OR*** of the two operands, replacing the destination with the result.
Modifies Flags: CF, OF, PF, SF, ZF (AF undefined)

► *Example* XOR
```
<R1> = 10100101
<R2> = 11111111
<R1> XOR <R2> = 01011010
```

► *Examples*
1. AND BX, 0
2. OR CX, 0FFFFH
3. XOR DX, DX

1. Equivalent to MOV BX, 0, setting all BX bits to 0.

2. Equivalent to MOV CX, 0FFFFH, setting all CX bits to 1.

3. Equivalent to MOV DX, 0, setting all DX bits to 0 (it requires less time than the MOV operation).

`TEST "dest", "src"` → ***Test For Bit Pattern***: it performs a logical `AND` of the two operands, updating the flags register without saving the result (similar to the compare operation `CMP`).
Modifies Flags: `CF`, `OF`, `PF`, `SF`, `ZF` (`AF` undefined)

► *Note: `TEST` is not requested for the exam* ◄

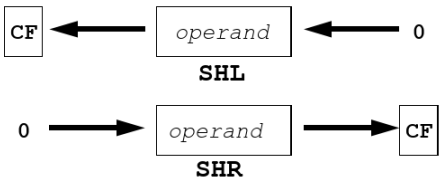
► *Shift and rotate instructions*
They allow to modify bits positions, moving them left or right for any value.
- ***Shift*** operations have the last bit in the shift direction copied to the flag `CF`, while the first bit is set to `0` or a value corresponding to the sign one.
- ***Rotate*** operations have the last bit in the rotation direction copied at the place of the first bit.

Shift:
Formatting: **`OPCODE "dest", "count"`**
Modifies Flags: `CF`, `OF`, `PF`, `SF`, `ZF` (`AF` undefined)

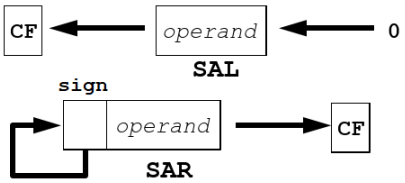
Shift right of n position is equivalent to ***integer division*** by 2^n; *shift left* of n positions is equivalent to ***multiplicating*** by 2^n.

`SHL "dest", "count"` → ***Shift Logical Left***: shifts `"dest"` left by `"count"` bits with zeroes shifted in on right. `CF` contains the last bit shifted out.
`SHR "dest", "count"` → ***Shift Logical Right***: shifts `"dest"` right by `"count"` bits with zeroes shifted in on left. `CF` contains the last bit shifted out (equivalent to the *residual of the division*).

The *logical shift* operation treats the data as a ***string*** or a ***pure binary*** value representation.



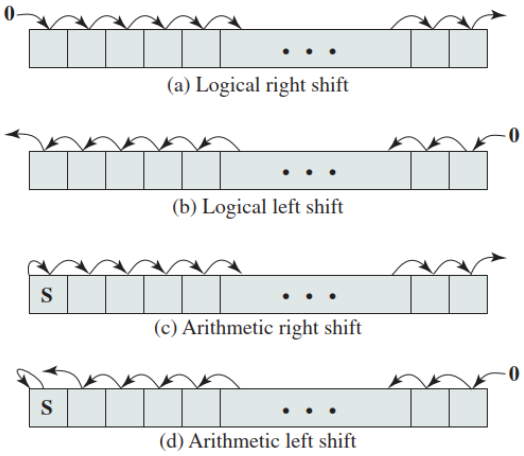
`SAL "dest", "count"` → ***Shift Arithmetic Left***: shifts `"dest"` left by `"count"` bits with zeroes shifted in on right. `CF` contains the last bit shifted out.
`SAR "dest", "count"` → ***Shift Arithmetic Right***: shifts `"dest"` right by `"count"` bits with the current sign bit (MSB) replicated in the leftmost bit. `CF` contains the last bit shifted out.



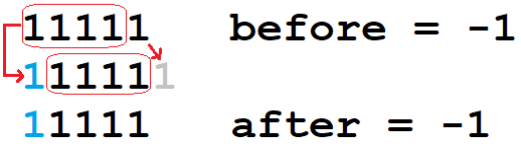
The *arithmetic shift* operation treats the data as a signed integer (***two's complement*** representation) and *does not shift the sign bit*.
On a *right* arithmetic shift, the sign bit is replicated into the bit position to its right.
On a *left* arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained.
These operations can speed up certain arithmetic operations. With numbers in two's complement notation, a right arithmetic shift corresponds to a division by 2, with *truncation for odd numbers*.
Both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow.

If ***overflow*** occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number.
In case of:
- `SHL` with pure binary, overflow happens when `CF=1`;
- `SAL` with two's complement, overflow is detected by comparing the sign bit before and after the shift (it happens when signs are different, `CF≠MSB`);

`SHR` and `SAR` cannot generate an overflow.



► *Example of particular cases*
Apply `SAR` to `11111` in two's complement.



Apply `SAR` to `01101` in two's complement (positive number): after 4 right shift it becomes `00000`.

Apply `SAR` to `10001` in two's complement (negative number): after 4 right shift it becomes `11111`.

Rotate:

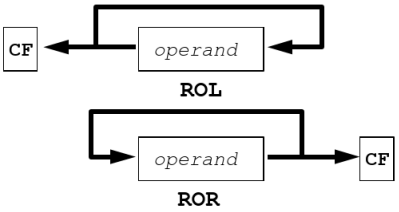
Rotate, or cyclic shift, operations preserve all of the bits being operated on. One use of a rotate is to bring each bit successively into the leftmost bit, where it can be identified by testing the sign of the data (treated as a number).

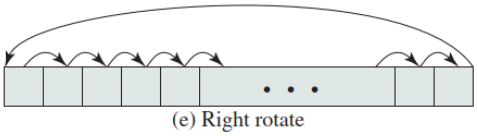
Formatting: **`OPCODE "dest", "count"`**
Modifies Flags: `CF`, `OF`

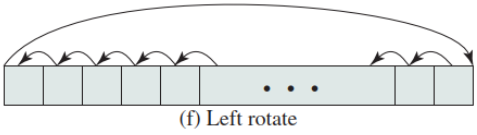
`ROL "dest", "count"` → ***Rotate Left***: rotates the bits in `"dest"` to the left by `"count"` times with all data pushed out the left side re-entering on the right.

`ROR "dest", "count"` → ***Rotate Right***: rotates the bits in `"dest"` to the right by `"count"` times with all data pushed out the right side re-entering on the left.

`CF` contains the value of the last bit rotated out.



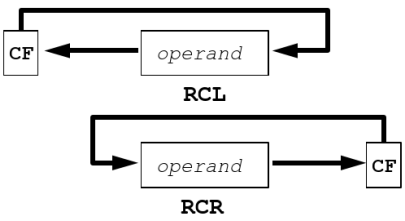

(e) Right rotate


(f) Left rotate

`RCL "dest", "count"` → ***Rotate Through Carry Left***: rotates the bits in `"dest"` to the left by `"count"` times with all data pushed out the left side re-entering on the right.

`RCR "dest", "count"` → ***Rotate Through Carry Right***: rotates the bits in `"dest"` to the right by `"count"` times with all data pushed out the right side re-entering on the left.

`CF` holds the last bit rotated out, as it was an additional bit at the left or right of the operand's bits!

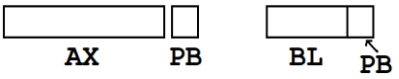


► *Examples of Shift and Rotate Operations*

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

► *Example*

Compute the parity bit even of `AX` and store to `BL`.



Possible solution: shift left 16 times and copy `CF` (this is a loop).
Let's shift left and focus on the internal part of the loop: `SHL AX, 1 ; a bit is in CF`

- if `CF=1` then increment a counter → `JC`
- if `CF=0` then move to next bit → `JNC`

Counting is not the only alternative.

The circuit we studied to implement the parity bit was a big `XOR` doing the sum of all bits.
How to sum `CF` to `BL`? `BL ← <BL> + CF`
`ADC BL, "src"` `; BL ← <BL> + CF + "src"`
`"src"` should be 0.

The program would be:
- Initialization of the loop

- Loop environment (repeated 16 times):

```
SHL AX, 1  ; bit in CF
ADC BL, 0  ; this bit is added to BL
```

```
ABD BL, 1 ; BL has the number of 1 in AX
```

The program would work but eventually `AX` is filled with `0`.
To avoid losing content of `AX` there are two solutions:
- use `PUSH / POP`

```
PUSH AX
 SHL AX, 1
 ADC BL, 0   16 times
AND BL, 1
POP AX
```

- use rotate left

```
ROL AX, 1
ADC BL, 0   16 times
AND BL, 1
```

► *Example*
What does the program do?
Pure binary `<AX> = a ≤ 1000`

```
MOV BX, AX        ; BX ← a
SHL AX, 1         ; AX ← 2a
SHL AX, 1         ; AX ← 2*2a = 4a
ADD AX, BX        ; AX ← 4a+a = 5a
SHL AX, 1         ; AX ← 2*5a = 10a
ADD AX, BX        ; AX ← 10a+a = 11a
```

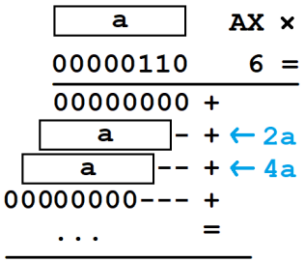
Final values are: `<AX> = 11a  <BX> = a`
The program multiplies `AX` by 11_{10}.

► ***Multiply by a generic constant***

Find a program that multiplies a register by a generic constant.

Pure binary `<AX> = a ≤ 65`

Multiply by 6
$6_{10} = 110_2$



Approach 1

```
MOV BX, AX        ; BX ← a
SHL AX, 1         ; AX ← 2a
MOV BX, AX        ; BX ← 2a
SHL AX, 1         ; AX ← 2*2a = 4a
ADD AX, BX        ; AX ← 2a+4a = 6a
```

Multiply as much as possible (shift left), then add when necessary (`AX+BX`).

Approach 2 !!

```
MOV BX, AX        ; BX ← a
SHL AX, 1         ; AX ← 2a
ADD BX, AX        ; AX ← 2a+a = 3a
SHL AX, 1         ; AX ← 2*3a = 6a
ADD AX, 0         ; AX ← 6a+0 = 6a (useless)
```

Converting 6 to binary:

```
6/2=3 , 0 remainder
3/2=1 , 1 remainder
1/2=0 , 1 remainder
```

The binary value is the sequence of remainders from last to first: `110`.

We reconstruct the number from the end:

```
6=(1*2+1)*2+0
```

Multiply by 9
$9_{10} = 1001_2$

Converting 9 to binary:

```
9/2=4 , 1 remainder
4/2=2 , 0 remainder
2/2=1 , 0 remainder
1/2=0 , 1 remainder
```

We reconstruct the number from the end:

```
9=((1*2+0)*2+0)*2+1
```

```
MOV BX, AX        ; BX ← a
SHL AX, 1         ; AX ← 2a
SHL AX, 1         ; AX ← 2*2a = 4a
SHL AX, 1         ; AX ← 2*4a = 8a
ADD AX, BX        ; AX ← 8a+a = 9a
```

► ***Exercise***

Please provide convincing explanations on what is the final value stored in `AX` after having completed the execution of the following assembly code. Assume that at the beginning `AX` stores a and `BX` b.
Find the maximum positive value of a to prevent ***overflow*** in case:
- a is expressed in *pure binary* notation
- a is expressed in *two's complement* notation

```
MOV BX, AX
ADD AX, AX
ADD AX, BX
SHL AX, 2
ADD AX, BX
```

Solution:
`<AX>=`a `<BX>=`b

Instruction	*Corresponding action*
`MOV BX, AX`	`BX` ← a
`ADD AX, AX`	`AX` ← $a+a = 2a$
`ADD AX, BX`	`AX` ← $2a+a = 3a$
`SHL AX, 2`	`AX` ← $(3a)*2^2 = 12a$
`ADD AX, BX`	`AX` ← $12a+a = 13a$

Final values:
`<AX>` = $13a$

To prevent overflow
`AX` is a 16-bit register.
- *Pure binary*:
  it can contain up to the value of 2^{16}=64K=65.536.
  Maximum a to prevent overflow: $a \leq {}^{2^{16}}/_{13}$
  The value of 13 needs 4 bits to be stored.
  The order of magnitude of a must be ${}^{2^{16}}/_{13} \cong {}^{2^{16}}/_{2^4} = \mathbf{2^{12}}$
- *Two's complement*:
  the MSB is reserved for the sign, so it can contain up to the value of 2^{15}=32K=32.768.
  Order of magnitude of a: $a \leq {}^{2^{15}}/_{13} \cong {}^{2^{15}}/_{2^4} = \mathbf{2^{11}}$

► *Test 2019-06-25 – Exercises 1, 2, 4, 7*

Exercise 1

A D-flip-flop…
A) is the basic block for designing an asynchronous counter
B) is a combinational block
C) is (basically) a dynamic memory cell
D) None of the previous answers because…

Solution: D
T flip-flop is the basic block for designing an asynchronous counter. D-flip-flop is a sequential block. D-flip-flop is basically a static memory cell; dynamic requires refreshing.

Exercise 2

We have a ***4-way-set associative mapping*** cache memory with ***globally*** 65536=2^{16} lines. Each entry is hosting 128 data. How many bits are necessary for the `TAG` for a 28 bits address bus?
A) 5
B) 6
C) 7
D) None of the previous answers because…

Solution: C
`Entries / Lines` = 65536=2^{16}
4-way-set: there are 4 memory blocks → k = 4
`SETs` = (# `Entries`) / k = $2^{16}/4 = 2^{16}/2^2 = 2^{14}$ sets
`Entries / Data` size = 128=2^7
The 28 bits A-BUS is a block divided in 3 fields:
- `Offset` = $\log_2$(`Entries` size) = $\log_2 128 = \log_2 2^7$ = 7 bits
- `Index` = $\log_2$(# `SETs`) = $\log_2 2^{14}$ = 14 bits
- `TAG` = Address – Index – Offset = 28 – 14 – 7 = 7 bits

Exercise 4

We have a memory of $\mathbf{2^{21}}$ cells organised in banks of $\mathbf{2^{13}}$ cells. The decoder used to select the proper bank is…
A) 8-to-3
B) 3-to-8
C) Such a memory cannot be implemented
D) None of the previous answers because…

Solution: C - D
The number of banks is $2^{21}/2^{13}=2^8=256$.
We would need a **8-to-256** decoder, with may be possible but really expensive in area, power and economic costs.

Exercise 7

Please provide convincing explanations on what is the final value stored in `AX`, `BX`, `CX` after having completed the execution of the following assembly code. Assume that at the beginning `AX` stores *a*, `BX` *b* and `CX` *c*; *a, b, c* are all in pure binary and not larger than ten thousand (in base 10).

```
XOR BX, CX
XOR BX, CX
XOR AX, BX
XOR AX, BX
XOR CX, AX
XOR CX, AX
```

Solution:
<AX>=*a* <BX>=*b* <CX>=*c*

Instruction	*Corresponding action*
`XOR BX, CX`	`BX` $\leftarrow b \oplus c$
`XOR BX, CX`	`BX` $\leftarrow (b \oplus c) \oplus c = b$
`XOR AX, BX`	`AX` $\leftarrow a \oplus b$
`XOR AX, BX`	`AX` $\leftarrow (a \oplus b) \oplus b = a$
`XOR CX, AX`	`CX` $\leftarrow c \oplus a$
`XOR CX, AX`	`CX` $\leftarrow (c \oplus a) \oplus a = c$

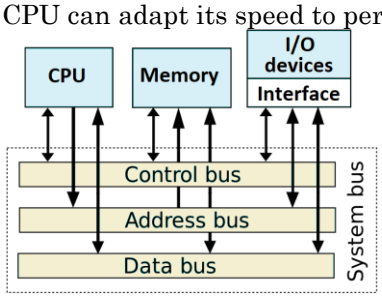
Final values:
<AX>=*a* <BX>=*b* <CX>=*c*
Contents are left as they were.

9 – PERIPHERAL DEVICES

• *I/O Organization*

Every computer needs devices capable of communicating with the external world, by sending and receiving information. This is achieved through the use of ***Input / Output devices***, often called ***peripherals***.

How CPU can interact with peripheral devices? How CPU can adapt its speed to peripherals?



The data transfer rate of peripherals is often much slower than the one of memory or processor. Thus, it is impractical to use high-speed system bus to communicate directly with a peripheral.
The ***device interface*** is a subsystem which manages the peripherals, including their communications with the CPU: this is what fills the gap of speed between CPU and peripherals, by doing a sort of "***virtualisation***" of the devices for the CPU.
Interface role can be divided in *managing the I/O hardware* and *communication with the CPU*.

The interface is a module connecting I/O devices to the interconnection network (system bus): it provides the means for *data exchange*, *status notification* and *command execution* needed to facilitate the data transfers and govern the operation of the device.

The interface includes registers that can be accessed by the processor:
- a register may serve as a buffer for data transfers,
- another may hold information about the current status of the device,
- another may store the information that controls the operational behaviour of the device.

These ***data***, ***status***, and ***control registers*** are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor.

Control signals determine the function that the device will perform, such as send data to the I/O module (`INPUT` or `READ`), accept data from the I/O module (`OUTPUT` or `WRITE`), report status, or perform some control function particular to the device (i.e., position a disk head).
Data signals are in the form of a set of bits to be sent to or received from the I/O module.
Status signals indicate the state of the device: i.e. `READY`/ `NOT-READY` to show whether the device is ready for data transfer.

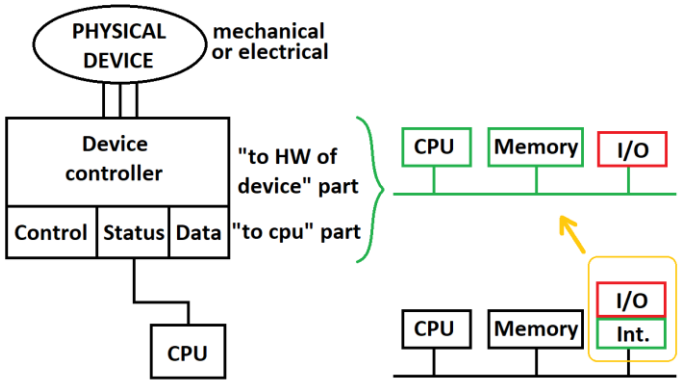
Register	***I/O interface can***	***CPU can***
Status	`Write`	`Read`
Command	`Read`	`Write`
Data input device	`Write`	`Read`
Data output device	`Read`	`Write`

Therefore, *generally* CPU considers I/O as "three registers" independently of the type and speed of the device.

Looking at the interface+I/O part of the Von Neumann architecture of the previous image, we can see there is an identical sub-architecture:
- the interface registers correspond to the memory;
- the "sub-CPU" is slower and much cheaper than the principal CPU, so the sub-CPU manages only the devices and the main CPU is free from this task;
- the I/O part is the *device controller*, connecting with the physical device.

This way the gap between main CPU and I/O devices is filled in the sense of order of magnitude. The main CPU can access the interface registers when needed, fetching the data.

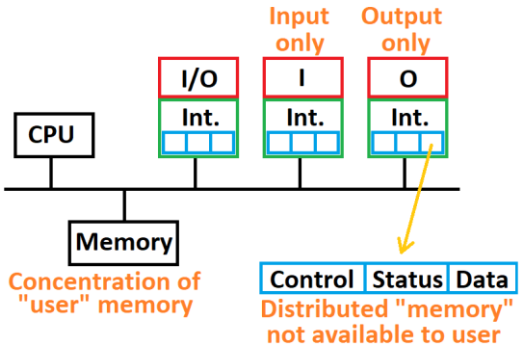
So, the sub-CPU virtualizes devices data in the registers for the main CPU (as the cache controller does for the cache and RAM).



An exception is graphic boards (GPU): they connect the computer to the screen; they cost and have a processing power comparable to the main CPU, because their role is reserved to operations regarding graphical calculus (games, animation, photography, video-editing…).

So, an I/O interface:
- Provides a register for temporary storage of data
- Includes a status register containing status information that can be accessed by the processor
- Includes a control register that holds the information governing the behaviour of the interface
- Contains address-decoding circuitry to determine when it is being addressed by the processor
- Generates the required timing signals
- Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port.

• *Peripherals addressing modes*

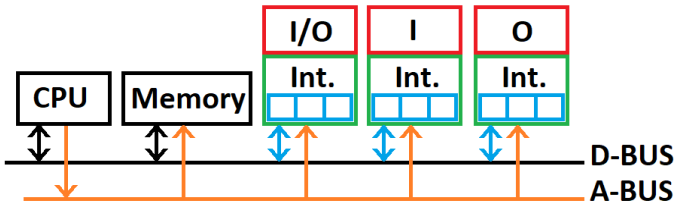


"User" memory (RAM) is available for programs to use.
Interface registers are only usable by the peripherals and they form a kind of distributed memory.
Therefore, we need an addressing mode to access "user" memory and peripherals registers separately.

How can the CPU interact with a peripheral device?
How can the CPU univocally identify each I/O device, as if an address could be assigned to each device?
There are two solutions:
- Memory mapped I/O
- Isolated I/O

► *Memory mapped I/O*

There is a ***single address space*** for both memory locations and I/O devices: peripherals are identified in the same way as RAM. The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both main memory and I/O devices.



`D-BUS` for I/O is coloured differently because it isn't always bidirectional, as in CPU and memory.

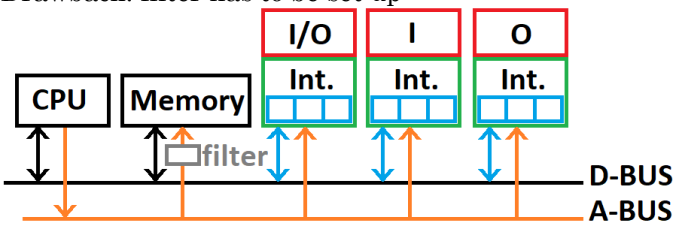
Bus	*CPU*	*Memory*	*I/O*	*I*	*O*
`A-BUS`		`Read`	`Read`	`Read`	`Read`
	`Write`				
`D-BUS`	`Read`	`Read`	`Read`	`Read`	`Read`
	`Write`	`Write`	`Write`	`Write`	`Write`

Example: with 10 address lines, a combined total of 2^{10}= 1024 memory locations and I/O addresses can be supported, in any combination.

With memory-mapped I/O, a single read line and a single write line are needed on the bus.

Advantage: no additional hardware is necessary.
Drawbacks:
- Sharing of the BUS may slow the performance.
- Less address available for the memory.
- There is a maximum number of I/O devices.
- If there are more than one device sharing the same address in memory and peripherals, we need to get the response from only one of them to avoid collisions happening. Therefore, ***addresses of memory and I/O should be separated***.

Solutions to the drawbacks
○ We need a "***filter***" to the `A-BUS` (may be hardware or software) that, in case of multiple corresponding addresses, lets propagate the signal only to the device that we want (and not to the memory). To do so, some addresses are reserved to I/O only: the consequence is that some memory cells will not be accessible because of this filter.
Advantage: there is more flexibility
Drawback: filter has to be set up



○ An alternative is to use ***two different addressing spaces*** (i.e., all addresses starting with `0` are assigned to memory, if starting with `1` are assigned to I/O).
Another example of rule:
- Addresses `000xxxx...x` are assigned to I/O devices
- All other addresses to memory

For example, the architecture in our previous picture may be configured in this way:

Peripheral	*Addresses*

`I/O`	`000 000 00`
	`000 000 01`
	`000 000 10`
	`000 000 11`

`I`	`000 001 00`
	`000 001 01`
	`000 001 10`
	`000 001 11`

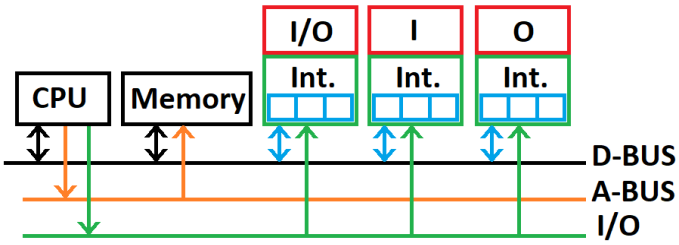
`O`	`000 010 00`
	`000 010 01`
	`000 010 10`
	`000 010 11`

`Memory`	`001 .....` `...`

To solve the drawbacks of memory mapped I/O, another solution is found: isolated I/O.

► *Isolated I/O*

The bus is equipped with memory read/write plus input/output command lines. The command line specifies whether the address refers to a memory location or an I/O device: the *full range of addresses may be available for both*.
Because the address space for I/O is isolated from that for memory (as if there was a dedicated bus), this is referred to as isolated I/ O.



Example: with 10 address lines, the system may now support both 1024 memory locations and 1024 I/O addresses.

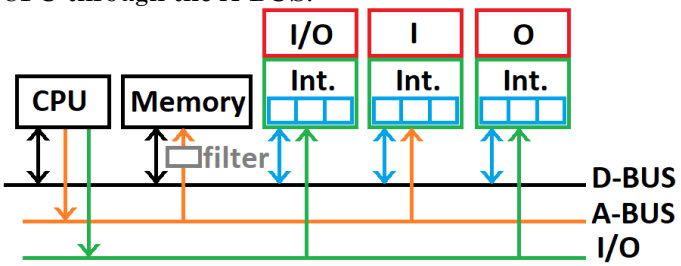
In isolated I/O, a separate address space of the CPU is reserved for I/O operations, totally different from the address space used for memory devices. In other words, a CPU has two distinct address spaces, one for memory and one for input/output.

Unique CPU instructions are associated with the I/O space, which means that if those instructions are executing on the CPU, then the accessed address space will be the I/O space and hence the devices mapped on the I/O space.
The ***8086*** is an example of isolated I/O:

- with **`MOV`** instruction, CPU can receive / send data to ***memory***
- with **`IN/OUT`** instructions, CPU can receive / send data to a ***peripheral device***

Disadvantage: the I/O interface will become complex and expensive.
If partial decoding is used to reduce the complexity of the I/O interface, then a lot of memory addresses will be consumed.

There can also be ***hybrid*** solutions, where only some peripherals are connected to the I/O BUS: the remaining are memory mapped and connected to the CPU through the A-BUS.



Why is it convenient to use a hybrid solution?
The I/O BUS can be implemented with really few bits.

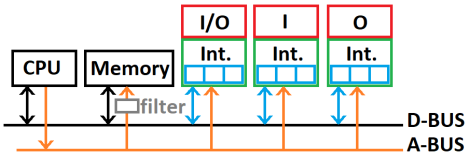
Example: an I/O BUS with 4 bits could manage up to 16 addresses and therefore 16/3=5 peripheral devices. It is needed to connect a 6th peripheral: it cannot be supported by the I/O bus, therefore the A-BUS can be used.

How will the CPU differentiate between these two address spaces? How will the system components know whether a particular transfer is meant for memory or an I/O device?
The answer is simple*: by using signals from the control bus*, the CPU will indicate which address space is meant during a particular transfer.

► ***Test 2020-02-10 – Exercise 6***

Please explain shortly and clearly, by using significant pictures, the ***memory mapped I/O framework***.
Please highlight advantages and drawbacks.

Solution:



There is a ***single address bus*** for both memory locations and I/O devices: peripherals are identified in the same way as RAM.
Advantage: no additional hardware is necessary.
Drawbacks: more than one device share the same address in memory and peripherals; to avoid collisions we reserve some addresses to I/O only: the consequence is that some memory cells will not be accessible because of this.
This is implemented by:

- using a "***filter***" to the `A-BUS` that lets propagate the signal only to the device that we want.
- using two different addressing spaces (i.e., all addresses starting with 0 are assigned to memory, if starting with 1 are assigned to I/O).

See ⇨ *Memory mapped I/O* lecture.

► *Test 2019-06-10 – Exercise 1, 2, 4, 6, 7*

Exercise 1

A T-flip-flop…
A) is the basic block for designing an asynchronous counter
B) is a combinational block
C) is ***not*** a sequential block
D) None of the previous answers

Solution: A

Exercise 2

We have a ***full associative mapping*** cache memory with ***globally*** 65536=2^{16} lines. Each entry is hosting 128 data. How many bits are necessary for the `TAG` for a 28 bits address bus?
A) 20
B) 21
C) 22
D) None of the previous answers because…

Solution: B

The number of lines is irrelevant because of the FAM.
`Entries/Data` size = 128 = 2^7
The 28 bits A-BUS supports a block divided in two fields:
- Offset = $\log_2$(`Entries` size) = $\log_2 128 = \log_2 2^7 = 7$ bits
- `TAG` = Address – Offset = 28–7= 21 bits

Exercise 4

We have a memory of $\mathbf{2^{21}}$ cells organised in banks of $\mathbf{2^{13}}$ cells. The decoder used to select the proper bank is…
A) 8-to-3
B) 3-to-8
C) Such a memory cannot be implemented
D) None of the previous answers because…

Solution: C - D

The number of banks is $2^{21}/2^{13}=2^8=256$.
We would need a **8-to-256** decoder, with may be possible but really expensive in area, power and economic costs.

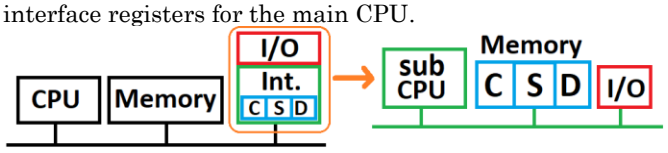
Exercise 6

Please explain shortly and clearly ***how an interface virtualizes a peripheral device to the CPU***. It is mandatory to use significant pictures.

Solution:

The ***device interface*** is a subsystem which manages the peripherals, including their communications with the CPU. The interface consists in two parts:
- a slow and cheap "sub-CPU", managing the device
- 3 registers (Status register, Data register, Control register)

Because peripherals are very slow in respect to the main CPU, the sub-CPU to waits for and manages them, while the main one is freed from this task and avoids wasting its resources. Only when necessary, the main CPU interacts with the devices through read/write operations on the 3 register: this way, the peripherals are virtualized as the interface registers for the main CPU.



See ⇨ *I/O Organization* lecture.

Exercise 7

Please provide convincing explanations on what is the final value stored in `AX` after having completed the execution of the following assembly code. Assume that at the beginning `AX` stores *a*, `BX` *b* and `CX` *c*; *a, b, c* are all in pure binary and not larger than 10.000_{10}.

```
MOV CX, AX
ADD AX, BX
SHL AX, 1
SUB AX, BX
SUB AX, CX
SHR AX, 1
```

Solution:

<AX>=*a* <BX>=*b* <CX>=*c*

Instruction	*Corresponding action*	*Overflow check (max value)*
`MOV CX, AX`	`CX` ← *a*	10.000
`ADD AX, BX`	`AX` ← *a+b*	20.000
`SHL AX, 1`	`AX` ← *2*(a+b)*	40.000
`SUB AX, BX`	`AX` ← *2*(a+b)-b = 2a+b*	30.000
`SUB AX, CX`	`AX` ← *2a+b-a = a+b*	20.000
`SHR AX, 1`	`AX` ← *(a+b)/2*	10.000

Final values:
<AX>=*(a+b)/2*
Average of `AX`, `BX`.
`AX` is a 16-bit register, so it can contain up to the pure binary value of 2^{16}=64K=65.536: there is no overflow.

► *Test 2019-06-25 – Exercise 3*

Memory mapped…
A) is a virtual device
B) is a specifically targeted to the addressing of peripheral devices
C) is generally more hardware-expensive than isolated I/O
D) cannot be used for hard disks
E) cannot efficiently handle very slow interfaces
F) does not require an additional bus, to data, address and control

Solutions (true): B, E, F

► *Test 2019-07-08 – Exercise 1*

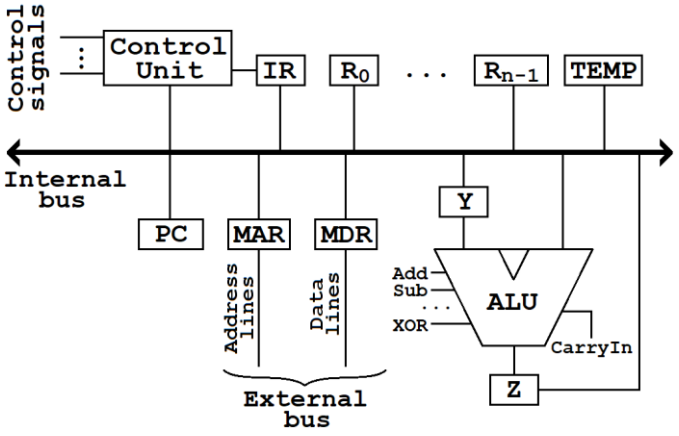
Memory mapped…
A) is most of times faster than an interrupt controller
B) is a combinational element
C) requires one additional bus
D) None of the previous answers

Solution: D

A: it is not a combinational element
B: isolated I/O requires the bus, memory mapped does not
C: Interrupt controller is a method to manage requests from peripherals, so it does not depend on the addressing method of peripherals

10 – CPU ARCHITECTURE

A picture of a general CPU internal organization follows:



A ***Central Processing Unit (CPU)*** consists of:

- ***Control and Decoding Unit (CDU)***: unit responsible for the decoding of instructions contained in IR and generating proper control signals.
- ***Instruction Register (IR)***: stores the code of the *current* instruction, which has to be executed (taken from the PC before it is updated in order to point out to next instruction); it divides automatically the received data between *opcode* and *operands* according to the CPU architecture specifications.
- ***Program Counter (PC)***: points to the address in memory of the next instruction to be fetched and executed. Once the current instruction has been fetched and is transferred to the IR, the program counter address is incremented to point to the next instruction in memory.
- ***Registers***: general purpose registers (R0, R1, T0, T1… in 8086 their names are AX, BX…) used to temporary store any kind of information.
  TEMP are non-user internal registers, used by CDU and CPU for storing as intermediate transfer of data.
  Y and Z are particular registers used as *buffers*:
  - **Y** serves to hold first input data for ALU operations
  - **Z** holds any ALU operation final result.
- ***Internal and External Buses***: used by units and registers to communicate between each other.
  The internal bus is a *data bus*.
- ***Arithmetic and Logic Unit (ALU)***: unit capable of performing arithmetic and logic operations.
- ***Memory Address Register (MAR)***: holds a main memory block address from which information has to be retrieved or into which it has to be written.
  It connects the CPU to the *address bus* (A-BUS).
- ***Memory Data Register (MDR)***: acts as a *buffer*, holding data that has been read from or has to be written to main memory.
  It connects the CPU to the *data bus* (D-BUS).

All components are registers, used to store certain data, except for ALU and CDU.

● *Microinstructions*

Example of execution of the instruction ADD R0, R1

In order to do it, we need to:

- move content of R0 to the ALU
- move content of R1 to the ALU
- do the addition into the ALU

The ALU has two bus inputs, but it is impossible to read at the same time both the contents of R0 and R1:

- Content of R0 is sent first and stored into Y
- Then, content of R1 is sent and directly delivered to the ALU.

The role of Y is to store the content of one operand in case of two operands operations, because of the impossibility of the bus to transmit both of them at the same time; this operation requires one more cycle.
ALU performs the addition and sends as output its result, but we need to store it temporarily to Z: the propagation of the result requires some time, if it was done directly to the internal bus it would merge with the R1 signal read at the same time, causing an error.
Then the result of the addition is delivered from Z back to R0.

ADD R0, R1	
Workflow	*Corresponding microinstruction*
Copy R0 to Y	$R0_{OUT}$, Y_{IN}
Send R1 to ALU Command ALU to take one operand from Y and the other from the internal bus, add them together and store the result to Z	$R1_{OUT}$, ALU_{ADD}
Copy Z to R0	Z_{OUT} , $R0_{IN}$

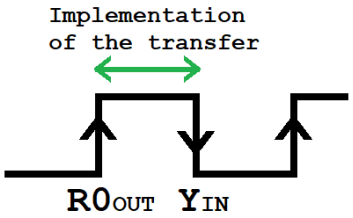
This is a sequence of ***micro-instructions*** or ***sub-instructions***: they are the "language" of the control unit.
IR stores the code of the instruction under execution (ADD R0, R1).
The CDU understands the instruction and translates it into the four steps of microinstructions, then it commands all the required elements to execute all the steps one by one until it completes the sequence (the result of the addition is found into R0 as requested).

Inside the CDU there is a memory containing the ***firmware*** of the architecture: the firmware is the *collection of all the microinstructions necessary to implement the execution of the Assembly instruction set of the CPU.*
The firmware may be updated: this is a dangerous procedure, but it permits to correct errors or add new features and more efficient ways to implement execution of instructions.

As said previously, MAR and MDR are respectively the gateways of the CPU to the address and data bus. Why are they so important?

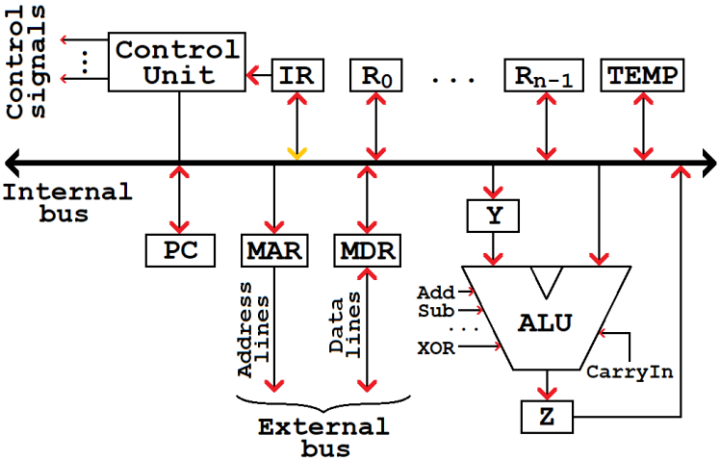
If MAR and MDR were missing, an instruction involving an operation external from the CPU (i.e. with memory or peripherals) would keep the internal bus of the CPU occupied feeding the external buses until completion of the operation.
By separating external and internal buses with MAR and MDR, it is their duty to feed the A-BUS and D-BUS: the internal bus is free and the CPU can keep performing other operations.

Even when microinstructions are listed in the same line (i.e. $R0_{OUT}$, Y_{IN}), *they are not performed simultaneously (at the same time)*, but consequently one of the other: during a clock cycle, to perform them as they were "together" the first is done during the *rising edge* while the second during the *falling edge*. The CDU decides how many clock cycles are required to perform any instruction and implements them accordingly.



Microinstruction	*Corresponding Action*
X_{IN}	Block X is reading from internal bus (write to X)
X_{OUT}	Block X is writing to internal bus (read from X)

Which CPU elements can read or write?



Element	*In respect to the internal bus, it can*
IR	WRITE: to feed the CDU with the code of the current instruction to be executed
	READ: the instruction from MDR, through the internal bus during the fetching operation
	WRITE*: feeding the internal bus happens only in *particular cases*.
R_0...R_{n-1} TEMP	READ: to store data temporary
	WRITE: to deliver the data previously stored
Y	READ: to store temporary an operand
	WRITE: when the other operand is available, the operand in Y is sent to the ALU
Z	READ: to store temporary the result from ALU
	WRITE: when the operation is complete and the result is fully available, it is delivered through the internal bus
PC	READ: to implement a JUMP operation
	WRITE: to communicate the address of the next instruction to be executed
MAR	READ: to address a cell of the memory, the address is written to MAR by the CPU and then sent out through the A-BUS
MDR	READ: to write data from the CPU to the memory (RAM), it is stored into the MDR and then sent out through the D-BUS
	WRITE: after reading data from the memory (RAM) through the D-BUS, it is stored into the MDR and then delivered from there through the internal bus

Computer Instruction Cycle

As seen previously, an instruction cycle is divided in:
- *Fetch*, when the code of the instruction to be executed is read from the RAM and stored into IR. The PC is updated accordingly, to point the next instruction.
- *Decode*, when the instruction is read by the CDU and translated into machine language
- *Execute*, when it is retrieved the needed data and done what the instruction code was asking.

Example: implement the "fetch"

FETCH	
Workflow	*Corresponding microinstruction*
Copy the address of the cell in memory containing the instruction	PC_{OUT} , MAR_{IN}
Read from memory	`Read from memory`
Copy the code read from memory to `IR`	MDR_{OUT} , IR_{IN}
Update the `PC` to point to the next instruction: `INC PC`	PC_{OUT} , ALU_{INC}
Copy `Z` to `PC`	Z_{OUT} , PC_{IN}

The *update of the `PC` value* and the operation of *reading from the memory* can theoretically be done ***in parallel***, because one operation is internal and the other external from the CPU (so only if the internal bus at that moment is free).

The operation "`Read from memory`" consists in:

MAR_{OUT},ext
Read from Memory (includes MDR_{IN},ext)

The operation "`Write to memory`" consists in:

MAR_{OUT},ext
MDR_{OUT},ext

`int` and `ext` indicate if the bus involved in the operation is internal or external to the CPU: they may be specified only for elements that communicate with both internal and external buses (`MAR`, `MDR`).

Cache ***prefetching*** is a technique used to boost execution performance by *fetching instructions or data* from their original storage in main memory to a faster local memory *before it is actually needed*. Modern processors have fast and local cache memory in which prefetched data is held until it is required. Accessing cache memories is much faster than accessing main memory, so prefetching data and then accessing it from caches is usually many orders of magnitude faster than accessing it directly from main memory.

Prefetching is optimised with ***pipelining*** of instructions: prefetched instructions are stored in a buffer, or *pipeline*, that delivers them one after the other. This allows overlapping execution of multiple instructions with the same circuitry, divided up into stages and each stage processes a specific part of one instruction at a time, passing the partial results to the next stage.

► *Example – microinstructions for* `DEC [R0]`

Content of the cell pointed by `R0` is decremented of 1.

$R0_{OUT}$, MAR_{IN},int
Read from memory (wait time)
MDR_{OUT},int , ALU_{DEC}
Z_{OUT} , MDR_{IN},int
Write to memory (wait time)

When can ***`IR` write to the internal bus***?
- Operations with constants, which are stored into the code of the instruction
- Indirect addressing pointed cells

► *Example – microinstructions for* `ADD R0, 3`

Add the constant `3` to the value stored in `R0`.

$R0_{OUT}$, Y_{IN}
IR_{OUT} , ALU_{ADD}
Z_{OUT} , $R0_{IN}$

► *Example – microinstructions for* `MOV R0, VAR+1`

Assembly code:
```
VAR DW (?)
MOV R0 , VAR+1
```

The compiler already knows:
- +1
- the address of `VAR`

`VAR+1` means the next cell "after" `VAR`

Therefore, the address is computed at compile time (by adding `+1` to the address of `VAR`) and stored to `IR`.

All the operations translate to:

IR_{OUT} , MAR_{IN}
Read from memory
MDR_{OUT} , $R0_{IN}$

► *Example – microinstructions for* `MOV R0, [R1]+3`

`[R1]+3` means the cell "`3` bytes after the one pointed by `R1`" (the content of `R1` is an address): this cannot be known at compile time (it is indirect addressing mode), so it has to be computed at runtime.

$R1_{OUT}$, Y_{IN}
IR_{OUT} , ALU_{ADD} ; <R1>+3 ("content of R1"+3)
Z_{OUT} , MAR_{IN}
Read from memory
MDR_{OUT} , $R0_{IN}$

► *Example – microinstructions for* `SUB [R0], 5`

We subtract 5 to the content of the RAM cell pointed by the address stored in `R0`.

$R0_{OUT}$, MAR_{IN},int
Read from memory , **IR_{OUT}** , Y_{IN} ; store 5 to
Y_{IN} while reading from memory
MDR_{OUT},int , ALU_{SUB} ; content of cell -5
Z_{OUT} , MDR_{IN},int
MDR_{OUT},ext ; write to memory

► *Example – microinstructions for* `MOV [R1], R4`

The content of `R4` is copied to the memory (RAM) cell pointed by the address stored in `R1`.

$R4_{OUT}$, MDR_{IN},int
$R1_{OUT}$, MAR_{IN},int
MAR_{OUT},ext
MDR_{OUT},ext

► *Test 2020-02-10 – Exercise 8*

Please write down the microinstructions to implement the instruction `MOV R0 , [R1]`, moving the content of the memory cell pointed by `R1` to the register `R0`. Please add a short but significant explanation to each single microinstruction.

Solution:

```
R1OUT , MARIN,int   ; copy address in R1 to MAR
                    prepare to read from RAM

MAROUT,ext          ; point the cell in RAM
Read from memory (includes MDRIN,ext)

MDROUT,int , R0IN   ; copy retrieved data to R0
```

► *Test 2019-09-19 – Exercise 8*

Please write down the microinstructions to implement the instruction `SUB [R1], R1`. Please add a short but significant explanation to each single microinstruction.

Solution:

Retrieve the data of the cell pointed by `R1`

```
R1OUT , MARIN,int   ; copy address in R1 to MAR
MAROUT,ext          ; point the cell in RAM
Read from memory (includes MDRIN,ext)
MDROUT,int , YIN    ; copy retrieved data to Y
```

Perform the subtraction

```
R1OUT , ALUSUB      ; do <cell>-<R1>,store in Z
ZOUT , MDRIN,int    ; prepare to send result
```

Write the result to the RAM cell

```
R1OUT , MARIN,int   ; copy address in R1 to MAR
MAROUT,ext          ; point the cell in RAM
MDROUT,ext          ; write to the cell in RAM
```

The workflow can be ***optimised*** as following (because only 1 element can write at a time, but all the others can read in parallel):

```
R1OUT , MARIN,int , YIN     ;so <R1> is already
                            ready for the ALU
```

The code for the subtraction would change as:

```
MDROUT,int , ALUSUB ; do <cell>-<R1>,store in Z
```

This permits to save at least one cycle.

Comparing the codes:

First	Optimised
R1OUT , MARIN,int	R1OUT , MARIN,int , YIN
Read from memory	Read from memory
MDROUT,int , YIN	MDROUT,int , ALUSUB
R1OUT , ALUSUB	ZOUT , MDRIN,int
ZOUT , MDRIN,int	Write to memory
Write to memory	

We do not know how the subtraction is performed (what is the order of operands), so the explanation of the microinstruction is fundamental to tell the reader what is our opinion:

- first case `Y - internal bus`
- optimised case `internal bus - Y`

• *Control and Decoding Unit*

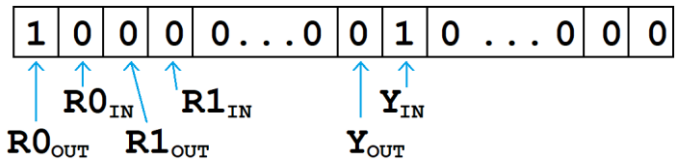
Let us consider microinstruction `R0OUT , YIN`
The register `Y` is a collection of D flip-flops that store the data arriving from `R0`. The enable signal that commands this action can be considered Y_{IN}.

We can say that *microinstructions are equivalent to a collection of (control) enable signals*.

As previously said, the internal bus architecture permits more devices to read but only one to write: $R0_{OUT}$ is a write → *hard-enable signal* (when active, all other elements are disconnected).

The *control unit* sends out control signals to any other element, according to the instruction code received from `IR`.
Every element needs at least two control signals (`IN/OUT`, i.e. for register `R0` there are $R0_{IN}$ / $R0_{OUT}$): to implement all these signals, ***control signal words*** are used.
A control signal word has a ***one to one correspondence*** to the *coding of a microinstruction*.
The consequence is that there needs to be a microinstruction for any combination of operands of each operation, so that the correct control signal can be delivered.

Suppose having 10 registers and an ALU requiring 40 control signals for all its operations: the total number of bits required for the coding would be 60 bits.
In such word, $R0_{OUT}$, Y_{IN} may be coded like this:

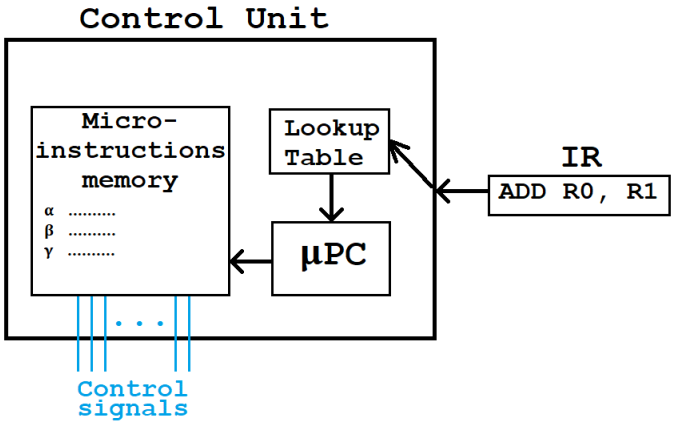


The enabled signals ($R0_{OUT}$, Y_{IN}) are set as 1, while all the others as 0 (remaining 58 bits): `R0` is activated to write to the bus and `Y` to read.
Writing control bits come before reading ones: the control unit takes care of the coding and priority.

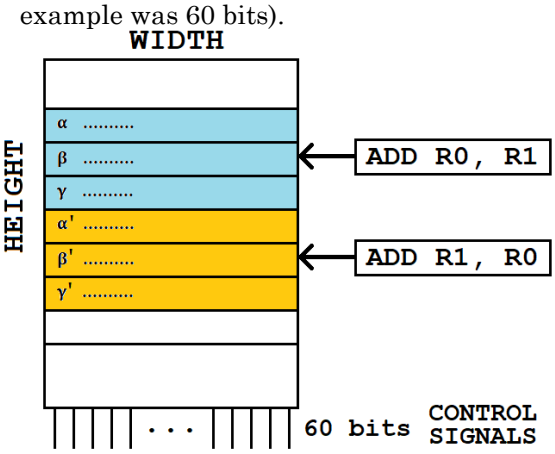
Control signal words are stored into the control unit: the `CDU` has a *memory block reserved for the coding of microinstructions and related signals*.

For example, the instruction `ADD R0, R1` is translated to the microinstructions:
$R0_{OUT}$, Y_{IN} → α
$R1_{OUT}$, ALU_{ADD} → β
Z_{OUT} , $R0_{IN}$ → γ
These three microinstructions are translated and stored to the ***microinstruction memory*** inside the **CDU**: to execute them one after the other, the **CDU** needs inside itself a ***micro-program counter*** (**MPC** or **µPC**) pointing to the next microinstruction (*addressing the microinstruction memory*), which will auto-increment during the execution.
The *microinstruction memory* and *micro-program counter* roles inside the CDU are equivalent to the *instruction register* and *program counter* of the CPU.

The translation from instruction to microinstruction is done through a ***lookup table*** inside the CDU.



The *microinstruction memory* has two dimensions:
- *height*, the number of µinstr it can contain;
- *width*, which depends on the length of µinstr (in our example was 60 bits).



Consider `ADD R0, R1`: it would require three lines of memory. If we do `ADD R1, R0`, with swapped operands, would it require other three lines?
By using this sort of notation, only the combinations of possible operands (i.e. 200) for the `ADD` operation would occupy 3×200=600 entries of µinstr memory.

This *fat microinstruction memory* is an example of ***horizontal microprogramming***.
Beyond a certain threshold, such memory would be too huge and expensive to be implemented on a CPU. How can we reduce its size without losing information?
By using an algorithm of ***compression without loss***.

• *Compression Algorithms*

Data compression is a technique in which the ***size of data is reduced***.
There are two categories of compression methods, depending on the form of data after *decompression*:
- ***lossless compression*** (or *compression without loss*), which restores and rebuilds the data in its original form;
- ***lossy compression*** (or *compression with loss*), does not restore the data in its original form

Characteristics of *lossy compression*:
- eliminates the data which is not noticeable.
- a file cannot be restored or rebuilt in its original form.
- data's quality is compromised.
- reduces the size of data (comparing original and decompressed).
- has more data-holding capacity.

Characteristics of *lossless compression*:
- does not eliminate the data which is not noticeable.
- a file can be restored in its original form.
- does not compromise the data's quality.
- does not reduce the size of data (comparing original and decompressed).
- has less data-holding capacity than lossy compression technique.

An example of lossless method is when a document file (.doc) is compressed in an archive (.zip) occupying less space; then it can be decompressed and restored as it was originally. Using the reduced-size file, it can be transmitted more easily.

Run-length encoding (RLE) is a form of lossless data compression in which *runs of data* (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run.

► *Example of run-length encoding algorithm*
The original sequence of data is:
`11111111000003355555555`

Count the occurrences:
`eight 1, five 0, two 3, eight 5`
This is already a compressed version of the original data, which can be restored (decompressed) without loss.

Another form of coding may be: `81502385`

Let's count the number of symbols per each representation:
- original data: 23 symbols
- first compression: 31 symbols (spaces included)
- second compression: 8 symbols

The second compression is obviously the most efficient.

► *Examples of **run-length encoding** algorithm*
Original data: `123423179`
Compressed: `111213141213111719`
Compression result: 9 to 18 symbols
This compression is not efficient at all.

Original data: `123123123123123`
Compressed: `5x123`
Compression result: 15 to 5 symbols
This compression is efficient because we adapted the algorithm to the form of the data.

Therefore, the "characteristics", "qualities", "properties" of the data must be known to perform an efficient compression.

Curiosity: RLE schemes were employed in the transmission of analog television signals as far back as 1967. In 1983, run-length encoding was patented by Hitachi. RLE is particularly well suited to palette-based bitmap images such as computer icons, and was a popular image compression method on early online services such as CompuServe before the advent of more sophisticated formats such as GIF.

Another example of lossless data compression is the ***Huffman Coding***. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Prefix codes remain in wide use because of their simplicity, high speed, and lack of patent coverage.

► *Example of lossy compression*
A camera takes a picture with very high PPI (pixel per inch, which is the resolution density unit for digital images; DPI, or dots per inch, is the printer's density of dot unit): for example, a DSLR medium-end camera shots high quality images at 24MP (6000x4000 pixels), 300 PPI.
A pixel (or picture element) is the smallest single component of a digital image: for convenience, in a digital image, pixels are normally arranged in a regular two-dimensional grid.

There is always a minor gradient variation between each of them. What happens when two near pixels have more or less the same value?
RLE alone does not work well on continuous-tone images such as photographs, although JPEG uses it on the coefficients that remain after transforming and quantizing image blocks: with the quantization are obtained many blocks of pixels with ***approximately*** the same value (color), then RLE can be applied for grouping them and doing another compression.

► *Examples of RLE with **lossy compression***
Original data: `112112221122111234443434`

By applying an approximation of the value, depending on the near values most similar and with most occurrences: `11`**`1`**`11`**`111`**`11`**`11`**`1113`**`333`**`33`**`3`

Now we can compress with RLE: `15x1,9x3`
Compression result: 24 to 8 symbols

Now we can decompress the data and compare it to the original one: `111111111111111133333333`

The decompressed data is different from the original uncompressed: the original sequence cannot be restored because there is no way to know where has been applied the approximation (that corresponds to ***adding errors*** *to the data so that the compression can be maximized*). The price for obtaining a reduced size is the loss of information.
Compressing data requires its *analyzation to identify the best compression algorithm possible*: this may require a lot of time and resources. Decompression instead is usually fast and does not require lot of computation power: there is an asymmetric correlation between compression and decompression.

Using lossy or lossless compression depends on the kind of data is processed:
- medical images need to be identical to the original ones when studied, so lossless compression is used;
- social media posts instead use lossy compression to post and send data to optimize the impact on the network and the accessibility by all users.

Compression of PDF files is often be useless because the obtained file may be larger than the original: this happens because PDF is already a type of optimized / compressed file.

Another example of compressions is for video streaming, broadcasting and transfer.
Advanced Video Coding (***AVC***), also referred to as ***H.264*** and MPEG-4 AVC, is a video compression standard introduced in 2003 and used by 91% of video industry developers as of September 2019. The standard describes the format of the encoded data and how the data is decoded; it is typically used for lossy compression, although it is also possible to create truly lossless-coded regions within lossy-coded pictures or to support rare use cases for which the entire encoding is lossless.
High Efficiency Video Coding (***HEVC***), also known as ***H.265*** and MPEG-H Part 2, was introduced in 2013 and designed as a successor to the H.264; used by 43% of video industry developers as of September 2019.
In comparison to AVC, HEVC offers from 25% to 50% better data compression at the same level of video quality, or substantially improved video quality at the same bit rate; obviously HEVC compression requires more time and resources than AVC. It supports resolutions up to 8192×4320, including 8K UHD.

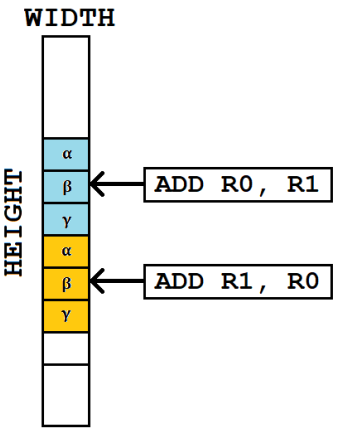
Online streaming services analyze the connection speed of each user and automatically choose the optimal video quality: fast connections will get a video source with higher quality (4K, Full HD), slow connections will get a lower quality (HD or less).

Vertical microprogramming

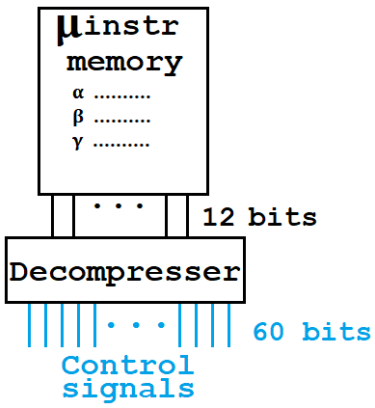
Going back to the microinstructions' memory: for each microinstruction we had a line of 60 bits with only two "1" bits and all the other 58 bits as "0". Can we compress it? Obviously yes.

How to compress? By storing the positions of the "1" only.
To identify a number from 0 to 60 we need 6 bits (2^6=64): in the case of two "1" only, we store their positions in two 6-bit blocks.
We compressed a 60-bit line to two 6-bit block, occupying in total only 12 bits.

Therefore, going back to the examples of `ADD R0, R1` and `ADD R1, R0`: we occupied for each instruction 6 slots of 6 bits each. In total they are 72 bits instead of 360 bits (6 lines).



Advantage: the compression permits us to have a "slimmer" memory.
Drawback: we need the decompression (using a decompressor, a sort of decoder) before "using" the µinstr control signal, spending more time and hardware. The delay will not be the memory only, but the sum of memory delay and decompressor delay. Consider also the reduced flexibility of the architecture: if our memory lines can store only two positions at a time, we will need to think how to implement more than two "1" µinstr.



Elements using the approach with compression are called "***vertical microprogrammed control units***".

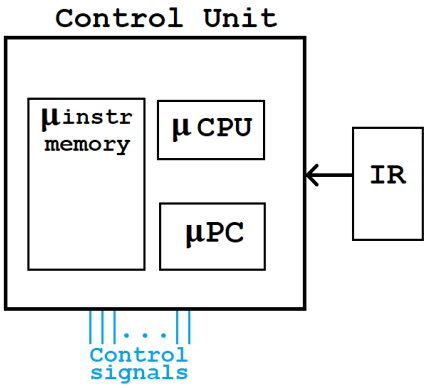
To summarise, the `CDU` can be characterized by control signals generated according to:

- ***Horizontal Microprogramming***: the `CDU` has multiple control signals as outputs, that do not need to be decoded in order to be correctly processed. Although they are immediately available, this microprogramming approach needs a large memory, does not use it efficiently, and it's costly.

- ***Vertical Microprogramming***: as soon as control signals are generated, they are expressed in an encoded and compressed form, so there is the need for decoding circuitry in order to correctly process these control signals. That's why it needs a smaller memory and it's a cheaper approach: it is slower with respect to the horizontal microprogramming approach because of the decoding phase.

Another approach may be ***recycling*** *the µinstr operations*: for example, the code of an addition with two operands may be generic and then have an element that selects the operands. *The entries of the µinstr memory would no longer be control signals*: control signals are statics, while these lines of data need to be "*adapted*" at runtime depending on operands. To do so, an intelligent system is needed, like ***another CPU inside the CDU***.

The actual architecture of the Control and Decoding Unit is therefore the following.



The interconnection inside is too complex to draw; we only list the elements composing it:

- the microinstructions memory
- the microprogram counter (µPC)
- the ***micro CPU*** (µCPU)

In the µCPU we will find another control unit, called nano-control unit, doing the equivalent work the control unit does for the CPU.

All the computer architecture is based on this ***hierarchical approach***.

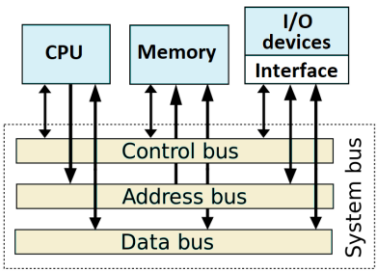
At the beginning of the Microinstructions lecture we talked about the ***firmware*** of the architecture, defined as the *collection of all the microinstructions necessary to implement the execution of the Assembly instruction set of the CPU*.

What is a ***firmware upgrade***?

Most of the times, it *changes the operating system*, which is executed in a loop by the device as interface for the user to access the hardware.
When the firmware upgrade *fails*, it causes a brick of the device and it becomes impossible to reach the hardware. There exist some recovery procedures to recover the connection and reload the operating system.

Few times instead, with firmware upgrade we mean *changing the content of the control unit's microprogrammed memory* (in the case of motherboard BIOS, routers…) which is very close to the basic of the operating system itself.
In the case of a *failed upgrade*, a hard brick happens and there may be no solution to restore a previous configuration.

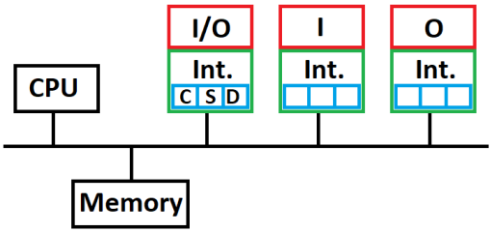
• CPU and peripheral devices interaction



The CPU is the owner of the bus: when a peripheral wants to use the bus, it needs to ask the CPU the permission. How to do it? By asking through the bus the permission to use the bus: this does not make any sense, because it is an infinite loop.
While memory is usually a passive element only, I/O devices instead are active elements of the architecture, both receiving and sending data ***in an asynchronous way*** in respect to the CPU.

For example: typing on a keyboard can be done at any time, instead of a predefined time in respect to the CPU. When a key is pressed, the device tries to deliver as fast as possible the value to the CPU. This is obviously an asynchronous action in respect to the clock of the processor.

The problems between CPU and peripherals can be summed up as:
- peripheral act *asynchronously*, while the CPU refers to a clock;
- elements *outside* the CPU need to be managed differently in respect to the elements inside the CPU;
- need for a *communication* with the CPU started by the I/O device (which needs the permission to use the bus).



What can be the approaches to solve these architecture limits?

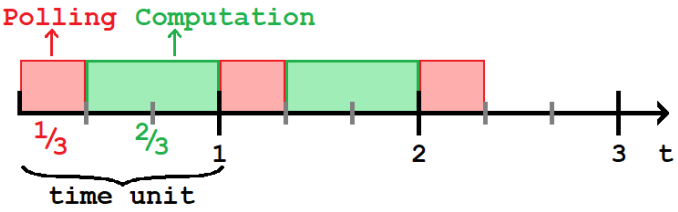
Polling is the process where the processor actively checks the status register of each peripheral device regularly (as a synchronous activity): it may be implemented as a round-robin algorithm, checking first to last device and then restarting the loop.
The device that wants to communicate updates, through its interface, its status register and provides the data into the data register.
When it is found that the device needs to communicate some information, the cycle stops and the bus is reserved for the device to use, according to the peripheral's needs.

Let's analyse characteristics of ***software polling***.
Advantages:
- polling is cheap and easy to implement
- easy to customize the process
- easy to assign priorities to devices

Drawbacks: ***waste*** of CPU cycles, because the devices are constantly checked also when they do not need attention, therefore constantly spending time for a useless task.

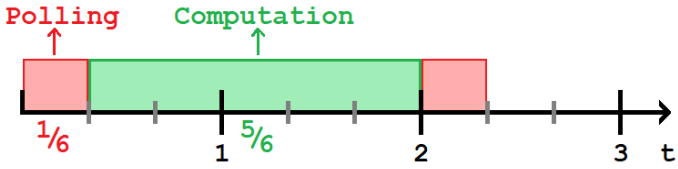
This waste of time is negatively impacting the performance of the system because:
- the CPU, which is the *most expensive* and fast *element of the system*, is asked to keep doing a useless task instead of computing something else;
- there is a *gap in speed* between the CPU and the peripherals (requests of attention are *not frequent* with respect to the CPU timing; i.e. typing on a keyboard may have a frequency of 3-4 keys/s, so 3-4Hz; the CPU has a frequency of 3GHz, with a gap of 10^9Hz in frequency)

When to use polling? When the CPU is cheap.
Let's consider a CPU that uses:
- $^1/_3$ of its time for polling (I/O management)
- $^2/_3$ of its time for net computation (or efficiency)



Most of time the polling phase returns that no device requires "attention", because of the gap previously mentioned!

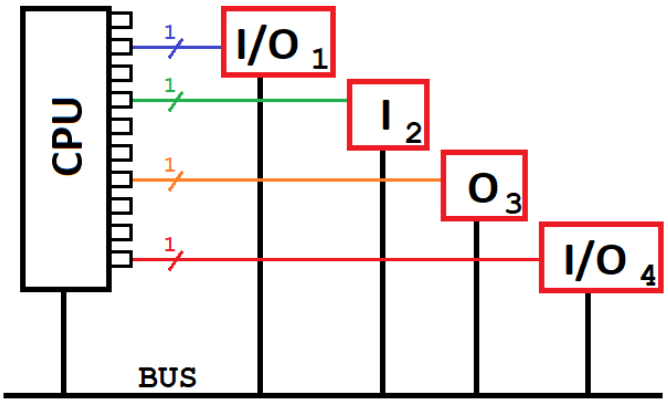
How can this approach be improved? For example, by *reducing the frequency of polling* to every 2 time-units, obtaining the following ratio:
- $^1/_6$ of time for polling
- $^5/_6$ of time for efficiency



By decreasing the polling frequency, the efficiency increases.
What is the drawback? The CPU becomes ***less responsive to "urgent" requests by devices***.

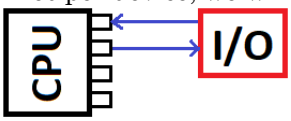
To overcome these limits, we need to switch to another solution.

The complete opposite approach to polling is a ***full hardware solution*** (or p2p, peer-to-peer).



This is implemented by connecting with a single and unique 1-bit line each device directly to a pin of the CPU. This line is only used to signal when the device needs to interact with the CPU: when the processor receives such signal, it knows what particular device requires attention and gives it permission to use the bus.
Polling is not needed anymore because the request is directly received, so the peripheral is immediately identified.

Note: in the picture it is drawn only one line per device, but this is not correct: there need to be two lines per device, we will understand why later.

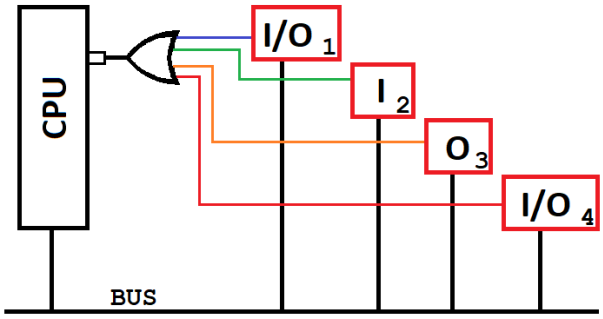


Advantages:
- no waste of time, the CPU manages a device only when necessary

Drawbacks:
- cost of wires
- poor flexibility
- the number of pins is limited, which corresponds to a limited number of I/O devices

On demand polling / Triggered polling

How to solve the problem of limited number of pins?
By using an OR gate, or in case of many devices a cascade of OR gates.
This is a scalable architecture, not bounded by the number of pins.



Advantages:
- device request detection is done in hardware
- flexibility
- scalability
- no waste of time
- not limited anymore by the number of pins

Drawbacks:
- cheaper than full hardware approach, but still expensive
- the OR gate cannot be reversed, it is not bidirectional, so the device which generated the signal is unknown
- the consequence is that an *identification mechanism* is needed (i.e. software polling)

Starting from the OR gate solution above, let's analyse two common *identification mechanisms*:
- *on demand polling* (identification via software)
- *daisy chaining*
- *interrupt controller*

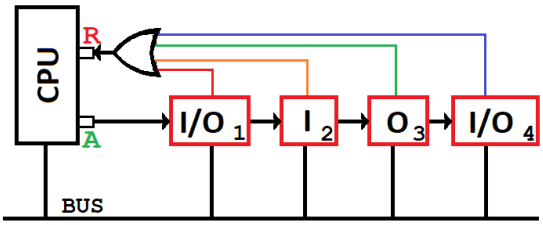
When the request of attention to the CPU becomes true, a ***on demand software polling cycle*** is started and proceeds to identify the active device through the bus.

Advantages:
- cheaper than hardware solutions (but more expensive than full software)
- flexible
- no waste of time because when polling is started there is at least one device requesting attention

Drawbacks:
- it is implemented in software, so it may be slow

Daisy chaining

The CPU has two separate pins (and corresponding lines): ***request*** and ***acknowledgment***. When the request of attention to the CPU becomes true, a signal is sent in series through all devices, so that the active device knows the request has been received and the bus is free.
When the acknowledge signal is received by the active peripheral, it stops there and *devices after that one will not receive any feedback signal*. If there are more than one active device at the same time, the signal on the OR gate remains high and another acknowledgment signal is sent.



Advantages:
- fast because implemented in hardware
- simple

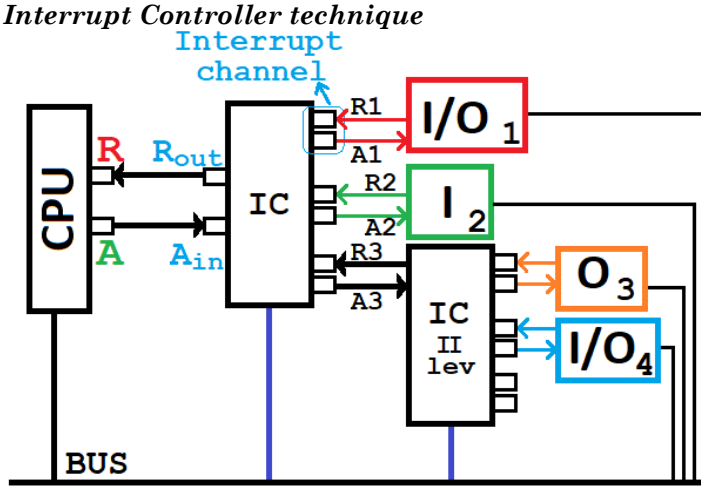
Drawbacks:
- fixed priority
- not resilient to failures (if a device has a problem / is not working / is sleeping, the signal cannot propagate to the peripherals after it)
- if a device has several requests one after the other (therefore there is a small gap between them), the attention is constantly given to it; when high in priority, the signal would never propagate to lower priority devices, excluding them to the system.

Setting up a double acknowledge signal may contain partially the lack of resilience: a signal goes in 1-2-3-4 direction, the other one the opposite 4-3-2-1. If two devices are not working at the same time, the transmission chains are blocked: this approach fails as the previous one.

In the case a peripheral has constantly several requests, it is more convenient to use a sub-processor in charge only of such peripheral, freeing the main expensive processor for other operations.

Different solutions can be found for each problem, each one with corresponding advantages and limits.

Interrupt Controller technique



The system is implemented using an ***interrupt controller*** (`IC`): it is an intelligent device having a dedicated CPU with limited computational power, enough to take care of few tasks.
In this case it has three ***interrupt channels*** (`R1/A1`, `R2/A2`, `R3/A3`) each channel consists of a couple of pins, one for request and the other for acknowledge, connected to a peripheral device. In respect to the CPU, there are two more pins: one for *request output* (R_{out}), the other for *acknowledge input* (A_{in}), both connected to the CPU's pins.
We need to connect four peripherals: two are directly attached to the interrupt controller, how to implement the others? Again, here is a problem of ***limited pins***, which can be solved in an equivalent way as before: by *cascading another interrupt controller* (`IC II level`) and connecting the remaining two devices to it.

Suppose device 3 sends a request: it arrives to second level `IC`, which forwards it to its superior `IC` (first level) and then it is sent to the CPU. The CPU responds with an acknowledge signal, which the `IC` assigns and redirects to the proper channel, according to some priority: now the priority is managed by the `IC`. Keep in mind that the acknowledge signal works in the same way as in daisy chain.

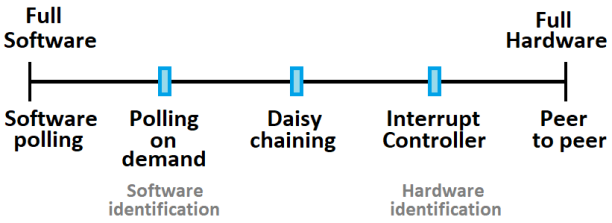
Advantages:
- scalable and flexible
- resilience regarding peripherals is improved in respect to daisy chaining

Drawbacks:
- the bottleneck now are the interrupt controllers, both for resilience and speed. Anyway, this is better than in the case of daisy chaining: `IC`s are internal elements of the architecture, if one of them stops working it is likely that the whole architecture has problems and must be repaired (while if an external element like one key of the keyboard stops working, the whole system must keep working independently of that).

CPU usually has two pairs of pins to connect `IC`: one is for low priority devices and the other for high priority devices; it is up to the designer to connect the peripheral ports to the appropriate set of pins.

How can the CPU know ***which device needs attention*** and why?
■ First approach: ***the device identifies itself***.
As seen previously, the peripheral stores the *details of its request into its status register* and the data into its data register.
Once received the acknowledge signal, the device sends its *ID* through the `D-BUS` to the CPU and the operating system immediately knows which is the device by comparing it to a table with all device IDs and *status register addresses*. The CPU proceeds to access and read the status register of the device (according to the addressing mode of the peripherals), then it does the corresponding operations.
■ Second approach: ***through interrupt controller***.
Each `IC` has alco a line connecting it to the `D-BUS`.
The last level `IC`, when delivering the acknowledge signal to the peripheral, sends through the `D-BUS` the ID of the device. Then everything proceeds as above.

CPU - peripheral devices interaction techniques



What means for the CPU to ***"handle" an I/O service request***?
- the CPU is doing some computations or instructions.
- it receives a ***interrupt request***: it responds with an acknowledge signal and ***stops doing the previous computations***.
- it *runs a "service"* ***routine*** *for that peripheral device*: this corresponds to install or run the corresponding device ***drivers***, which is a software operation loaded into the operating system.
- once the requests of the device are completed, the CPU returns to the computations paused before.

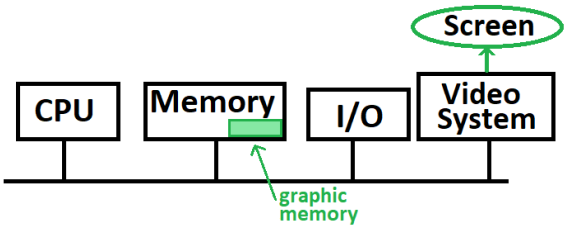
If a device is really common, it is likely to be compatible to *generic or universal drivers*, embedded in the system. If not, specific drivers are installed and made part of the permanent memory of the OS so that they can be retrieved and run when needed.

The CPU cannot execute requests while processing other instructions, because a request of attention corresponds to executing other instructions belonging to the service routine.
Executing service routine is equivalent to a ***call to a procedure*** in a program: it is a routine executed by the CPU on need (on demand) by the peripheral device. Even if interrupt requests arrive *asynchronously*, the CPU must finish executing the current instruction and only then it can go to *other instructions*: the consequence is that the service routine *will be synchronized* by the CPU at the time it is received.

Role of the GPU in a computer

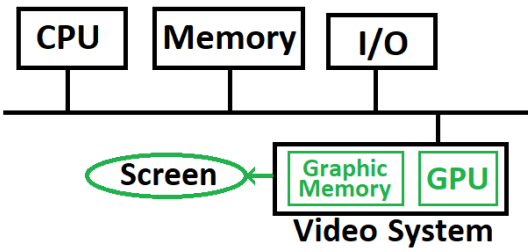
In a computer, the image displayed on the ***screen*** is elaborated by a processor and *stored in some memory*, then delivered with a one-to-one correspondence through the ***video system***.
In notebooks the image is stored in a shared area of the RAM. If the video system is not equipped with a dedicated CPU:

- the signal begins from the main CPU and goes to the graphic memory to store and retrieve data
- goes back to the main CPU to process the data
- then it is delivered to the screen through the bus.



In case of animations, graphic elaborations, streaming and other processes that need quick response require an ***intensive use of the communication memory-screen***, they need a lot of resources, which in this case are computed by the main CPU: keeping the main CPU busy for graphic requests is not a reasonable architecture choice.

How to improve the system? By implementing the video system with a ***dedicated graphic board*** (or ***graphic card***), which has an integrated ***GPU*** (*Graphics Processing Unit*) and a ***graphic memory***. All requests are now managed internally of the video system, not in conflict with the resources of the main system. If the requested data is stored in the main memory, it is copied to the graphic memory at the beginning and then processed there. If graphic memory is too small, the GPU will use a part of RAM.

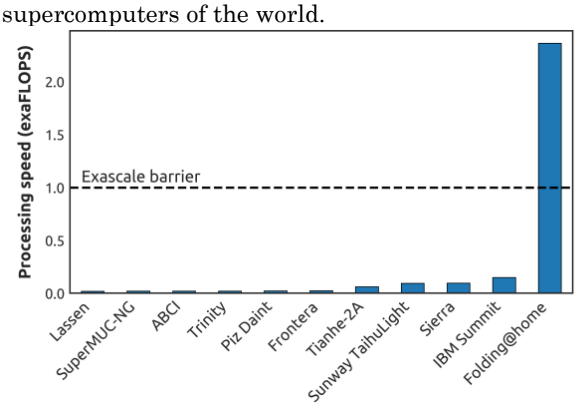


Cheap notebooks have GPU embedded inside the CPU.

There are also hybrid solutions, adopted in example by middle-end notebooks to keep an accessible price: the system has a GPU, but the graphic memory is shared inside the main memory.

The GPU functionality is an example of *direct memory access* (*DMA*), a technique used when large volumes of data are to be moved or processed.
Anyway, DMA is not part of the course.

Curiosity: during the search for a cure of COVID-19, the distributed computing project ***Folding@Home*** (foldingathome.org , for simulating protein dynamics, including the process of protein folding and the movements of proteins implicated in a variety of diseases), using GPU and CPU of home computers and data centres, achieved the computing power of ***2.4 exaFLOPs*** as of April 2020. Such result is larger than the sum of performance of the top 500 supercomputers of the world.



► *Note: The following part is an explanation from previous years' notes , simplified and using a different approach* ◄

Three main techniques are possible for I/O operations.

With ***programmed I/O***, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete.
The CPU continuously "polls" (polling phase) every I/O device to check if any of them needs attention. If the processor is faster than the I/O module, this is wasteful of processor time.

With ***interrupt-driven I/O***, the processor issues an ***I/O command***, called interrupt service routine, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work.

With both ***programmed and interrupt I/O***, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

There is somewhat of a trade-off between these two drawbacks. Consider the transfer of a block of data. Using simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a rather high rate, at the cost of doing nothing else. Interrupt I/O frees up the processor to some extent at the expense of the I/O transfer rate. Nevertheless, both methods have an adverse impact on both processor activity and I/O transfer rate.

– – – – – – – –

● *RISC, CISC*

There are fundamentally two types of processor instruction sets:

- ***Reduced Instruction Set Computers*** (RISC)
- ***Complex Instruction Set computers*** (CISC)

RISC style is characterized by:

- Simple addressing modes
- All instructions fitting in a single word
- Fewer instructions in the instruction set, as a consequence of simple addressing modes
- Arithmetic and logic operations can be performed only on operands in processor registers
- Load/store architecture does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
- Simple instructions are conducive to fast execution by the processing unit using techniques such as pipelining.
- Programs tend to be larger in size, because more but simpler instructions are needed to perform complex tasks

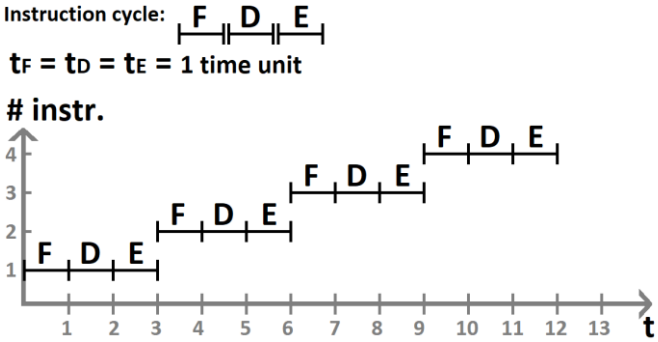
CISC style is characterized by:

- More complex addressing modes
- More complex instructions, where an instruction may span multiple words
- Many instructions implement complex tasks
- Arithmetic and logic operations can be performed on memory operands as well as operands in processor registers
- Transfers from one memory location to another by using a single Move instruction
- Programs tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks

• *Pipelining*

A pipeline organization is an architectural choice to improve "performances" of computation of a processor.

Consider the instruction cycle steps (or ***subsets*** or ***phases***): *fetch* (F), *decode* (D) and *execute* (E).



System without pipeline – instruction cycle graph

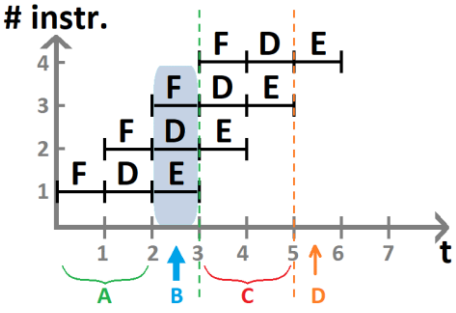
Overall, it takes *12 time-units* to execute this program.

The difference between "next" and "following" instruction is:
- ***next*** refers to time
- ***following*** refers to space

There is a correspondence between ***next and following*** when there is ***no jump***!

Can we do better? Yes, by parallelizing: it can be done applying the characteristics of fetch, decode and execute and reinventing the pipeline.

If each part of the cycle is carried over by a separate "unit", we can take advantage of such organization:
- do the fetch of the 1st instruction
- when finished, the task is passed to the decode "unit" and the fetch "unit" is free again, so it can take care of the fetch phase of the 2nd instruction
- when finished, the task is passed to the next "unit", so we have 1st instr. → execute; 2nd instr. → decode, 3rd instr. → fetch
- and so on, until the program completes the cycles.



Ideal pipeline *system – instruction cycle graph*

It takes only *6 time-units* to execute the program!

We can divide the whole program in three:
A. when the pipe is being filled / fed with instructions (before green vertical line)
B. the pipe is full
C. while the pipe is full (after green vertical line), at each passing time unit an execute phase ends
D. last "filled pipe" instruction ends (after orange vertical line)

The pipe ***starts producing results*** only after the first end of execute: all three units F, D, E are working in parallel, the pipe is full and keeps being full until last instruction.

Principles of pipelining

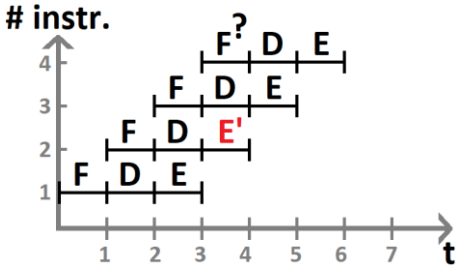
1. The basic requirement for any ideal pipeline is that fetch, decode and execute should ***use different resources***.

1b. How many "***phases***" for our pipeline?

2. Another issue is that the pipeline is effective only when ***next and following instructions correspond***: this happens only when it is possible to predict what operation will be the next!
3. The system is defined as ***ideal*** because it supposes that fetch, decode and execute phases have ***all the same duration***: this is not always the case.

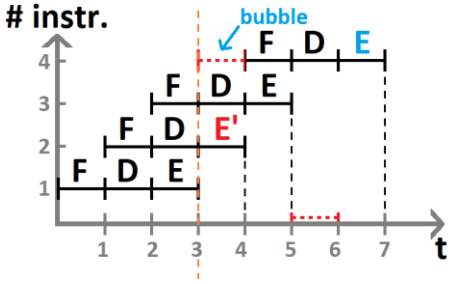
1 – F, D, E using different resources

Suppose F and E sometimes use the same resources.
We adopt this notation:
- **E'** , E is using a resource used also by **F**
- **E** , E is not using resources used also by **F**



What happens to **F** when requesting the same resources of **E'** (or vice versa)?
Can both work in parallel? Clearly not.
Therefore, the pipeline manager will need to give priority to one of the two: *execute* **E'** *will have the priority*.
Why **E'**? It is a matter of finishing the first task: **E'** is real, we are executing it now, ***it is the current instruction***.
The system cannot know if **F** of the 4th instruction will actually be carried out because at this moment it is just a ***prediction***, while it is certain of **E'**.

Therefore, the actual graph of the program is the following.



The idle time while **F** is waiting for the completion of **E'** is called *bubble*: this collision causes a delay that affects all the program; it is applied to the time unit between 5 and 6, during which there will not be a completion of an E phase.

What does correspond to each phase in our case?
- F is fetch, reading an instruction *from memory*
- E is execute, does not access memory
- **E'** is execute, writing or reading an instruction *to/from memory*

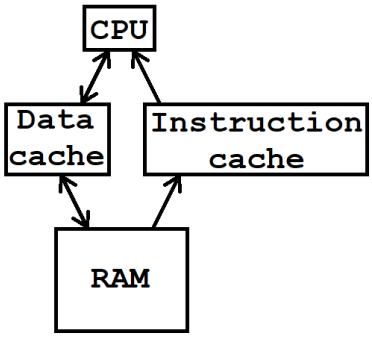
Examples of **E'** are:

```
MOV VAR , BX
VAR DW 12
```

To avoid this collision of **F** and **E'** we need to:
- Prohibit E on the memory (RISC does it partially)
- Separate the memory by ***using a split cache*** (data cache and instruction cache are separated)

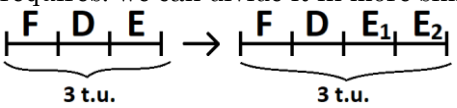
See ⇨ *Unified versus split caches* lecture.

With a sort of virtualisation, the CPU believes there are two separates memory to read from and so there is no more conflict between **F** and **E'**.



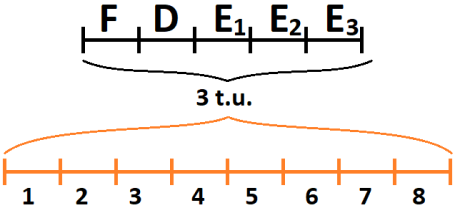
1b – How many phases?

Some instructions may have an execution phase E short or long, according to the number of operations it requires: we can divide it in more simple phases E_1, E_2.



For example: `ADD VAR , BX`
This requires two memory accesses (1 read and 1 write).

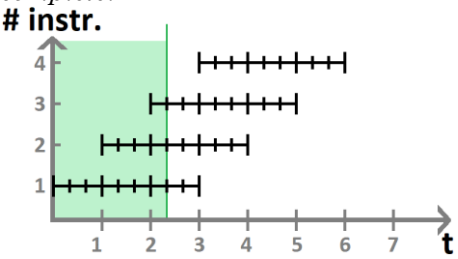
Why not splitting E in 3 phases then?
Or the whole instruction into 8?



Or even 10? 100? One million?
However, each splitting does not change the total time of the instruction, which is always *3 time-units*.

The *apparent* advantage of splitting in more phases after the pipe has been filled in, we will have that after every ***phase duration*** one execution will be completed.

Example
Suppose having instructions made of 9 phases.
Total time per instruction is still 3 time-units.
Therefore, *each phase requires 1/3 of a time-unit to complete.*



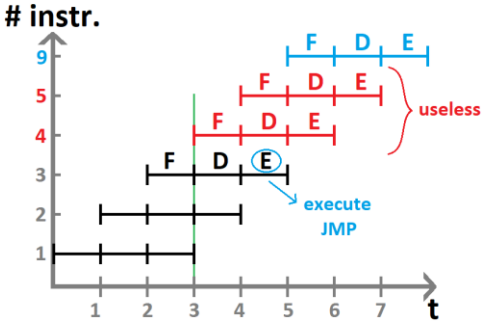
As we were saying, from the moment when the pipe has been filled (so instructions 1, 2 and 3 start working simultaneously, after the green part), per definition of ideal pipeline an execution gets completed after each phase, so ***each 1/3 of a time-unit***.
It seems like going faster! By splitting it more (100, 1 million) would it go even faster?
But *can we actually split an instruction into n phases with n large such that we get a boost in performance?*
No, that is impossible!

Why so?
- A certain task cannot go faster than the clock speed
- Microinstructions inside the CPU are the smallest possible split
- Having too much phases will never fill the pipe completely and therefore prevent the speedup
- *Conflict in the **use of resources** has to be avoided*!

The last reason is the most important.

Calculating floating points or graphics operations may use 7, 8, 9 phases on some CPU, but never more.

2 – Do next and following instructions correspond?
When they do, the pipeline works fine.
When they do not, problems arise.



Suppose instruction **3** is a jump to instruction **9**:
- during **D** of **3**, starts **F** of **4**
- during **E** of **3**, starts **D** of **4** and **F** of **5**

The CPU sees there will be a jump at the end of **D** of **3**, but it will actually acknowledge it only when it happens, so at the end of **E** of **3**.
Therefore, ***the work done in parallel*** for pipelining **4** and **5** ***becomes useless*** because of the jump to **9**!

The ***pipe*** will restart with the **F** of **9**: the previous content is discarded and it will be ***empty***, corresponding in a ***loss of performance***.

How to address the jump problem?
- In the case of an unconditional jump `JMP`, we can identify the destination instruction during **D** of **3** and start the phase **F** of **9** while doing the **E** of **3**.
- In a general approach instead, ***experience*** makes the difference: if that jump instruction has already been ***executed in the past*** and the CPU can have a memory of it, the organisation can be optimised accordingly.

By adding a ***branch prediction unit*** (**BPU**) collaborating to the pipeline mechanism, it will help to identify which is likely to be the next instruction.

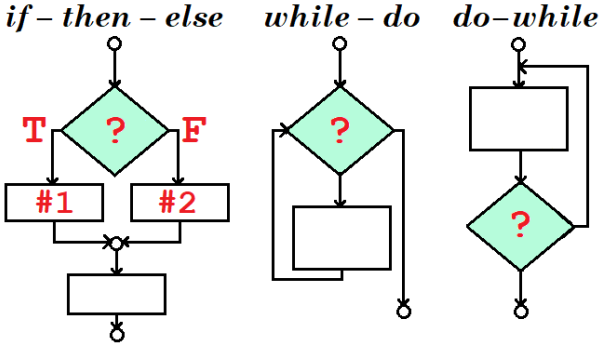
The `BPU` is like a blacklist containing:
- the address of *instructions with jumps*
- the address of their corresponding *next instructions*

In all the program next and following instructions correspond, except for the ones in the `BPU` list, for which are ***applied special rules***.

Like in a cache, at the beginning obviously everything is empty: the program starts and when a jump instruction is met there will be a loss of performance. Such instruction will be noted into the `BPU` as an exception, as well for any other jump: this experience list is written while the program is running and *applied each time the jump instructions are met again*.

Note that, at the end of the execution of the program, the `BPU` list is usually cleared.

Types of branch instructions



We are considering here conditional jumps cases (marked with "**?**").

Let's consider first the "*while-do*" case, excluding "*do-while*" because they are both ***loops***, one the variation of the other.
Entering the loop corresponds to an *unconditional jump*. Suppose the condition is checked 100 times (loop is repeated 99 times):
- if the CPU supposes avoiding the loop, an error in the fetch is made at the first cycle;
- if the CPU supposes entering the loop, no error is made;
- then the remaining 98 times it will run without errors ***thanks to the*** **BPU**;
- surely an error is made when exiting the loop, because the CPU assumes repeating it again.

The prediction was right at least 98% of cases!

Obviously, loops and consequently `BPU` are useful only when the number of repetitions n is large enough.

The worst case for the `BPU` is "*if-then-else*" case, but probably also the less impacting: there is a 50% of probability for each choice, so the prediction is randomly successful and the `BPU` becomes almost useless.

How to manage the problem of jumps in a loop without the `BPU` or any additional hardware?
The ***compiler*** can help sometimes, by ***unfolding the loop*** (also said ***unrolling*** or ***unwinding***).
It is a loop transformation technique that attempts to optimize a program's execution speed at the expense of having a longer code, undertaken by an optimizing compiler. The goal is to increase a program's speed by reducing or eliminating instructions that control the loop, reducing branch penalties.

To do the unfolding, loops are ***re-written as a repeated sequence of similar independent statements***.

For example, initialising an array without a loop means setting each element of the array to 0.

Another advantage of loop unfolding is the higher parallelization of instruction in general in the system, not only regarding the pipeline.

Loop unfolding advantages:
- no pipeline "problems"
- no branch penalties

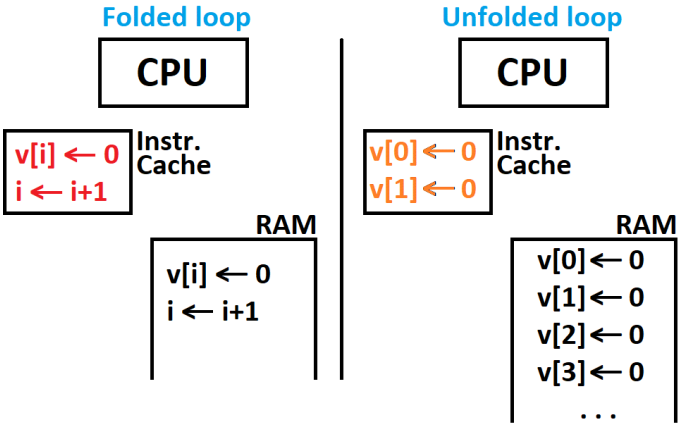
Drawbacks:
- longer code
- cache becoming less impacting

All the *case 2* reconnects to the *case 1b*, analysing the consequences of dividing in many phases each instruction.

When there are *3* phases, 3 instructions are involved into filling up the pipe before having it working full.

What if there are *n* phases? *n* instructions need to be involved to fill the pipeline, so ***the pipe would be full only after much more time***.
This is why *n* cannot be too much large.
Furthermore, if *n* =100, we would need 100 phases not sharing the same resources.
It is also ***likely that a jump will happen*** (the next is not the following) and that ***spoils the pipeline***.

Why is the ***cache less impacting***?

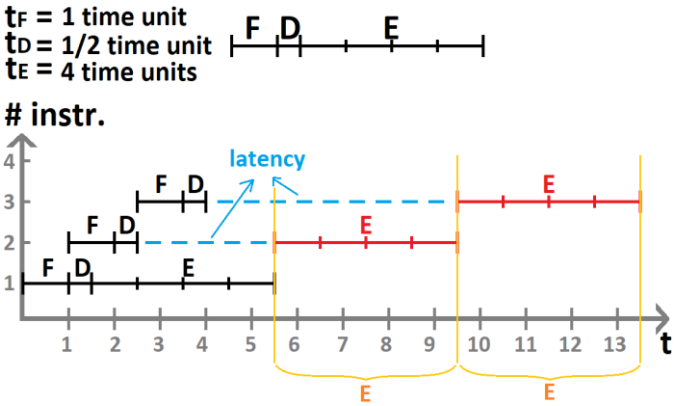


Let's compare two versions of the same program, one as it is and the other with loops unfolded.
Suppose the cycle has *n*=100 and the instruction cache has only 2 slots available.

From the CPU viewpoint during the program runtime:

	Folded loop (normal)	*Unfolded loop (no loop)*
t.u.	*Actions*	
1	- instructions are in RAM - both are read and copied to cache v[i] ← 0 i ← i+1	- instructions are in RAM - first two are read and copied to cache v[0] ← 0 v[1] ← 0
2	99 next times instructions are in cache	2nd instruction is in cache
3		- instructions are in RAM - cache is deleted - 3rd and 4th are read and copied to cache v[2] ← 0 v[3] ← 0
4		4th instruction is in cache
…		- instructions are in RAM - cache is deleted …
100		100th instruction is in cache
	How much is efficient cache?	
	99% of times	50% of times

3 – Duration of the phases
What if all the phases do not have the same duration?



Each phase must finish to let an equivalent phase start (because of resource competition), we have that the phase **E** of **2** cannot start until **E** of **1** is completed.
The consequence is that there is a latency equivalent to the remaining 3 time-units of **E** of **1**.
Considering then **E** of **3**, it cannot start until **E** of **2** is finished, which is equal to a 4 time units ***delay*** and this will hold ***for any next instruction***.

The "production" of executed instructions will last the duration of the slowest phase!

As a summary from all the cases:
1 – F, D, E using different resources
1b – How many phases?
2 – Do next and following instructions correspond?
3 – Duration of the phases

We found that
1 – The number of phases ***n*** should be ***small*** (or not too large)
2 – Having a branch prediction unit ***BPU*** is highly appreciated
3 – Phases should have ***comparable duration***
4 – Instructions with same (similar) duration would be ideal for pipelining

PRACTICE

► Test 2017-01-24 – Exercise 1, 2, 3, 4, 6, 7, 8
► Test 2017-06-27 – Exercise 1, 2, 3, 4, 7, 8
► Test 2017-07-10 – Exercise 1, 2, 3, 7, 8
► Test 2017-09-04 – Exercise 1, 2, 3, 7, 8
► Test 2018-01-22 – Exercise 1, 2, 3, 6, 7, 8
► Test 2019-06-10 – Exercise 3, 5, 8
► Test 2019-06-25 – Exercise 5, 8
► Test 2019-07-08 – Exercise 5, 8
► Test 2019-09-19 – Exercise 5
► Test 2020-05-14 – Exercise 1, 2, 3, 4, 5, 6, 7, 8
► Test 2020-05-21 – Exercise 1, 2, 3, 4, 5, 6, 7, 8
► Test 2020-06-26 – Exercise 1, 2, 3, 4, 5, 6, 7, 8
Multiple choice questions database
Theory questions database

► *Test 2017-01-24 – Exercise 1, 2, 3, 4, 6, 7, 8*

Exercise 1

A priority encoder …
A) Requires only OR and AND gates
B) Has the number of outputs which is always larger than the number of inputs
C) Has the same truth table of a non-priority encoder
D) None of previous answers

Solution: D

An encoder is a combinational circuit that is capable of encoding 2^n inputs and providing n outputs lines, which should represent the encoded inputs. A *priority encoder*, inspects most significant bits (leftmost positions) and, if one of them is equal to 1, the output can be delivered without having to worry about least significant bits.

Encoder

x_3	x_2	x_1	x_0	z_1	z_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	1	0	1	Forbidden	

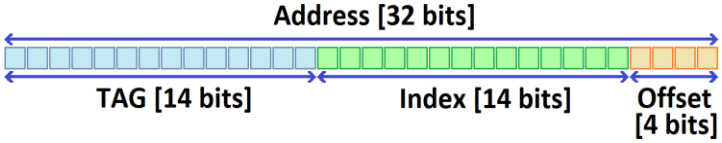
Priority Encoder

x_3	x_2	x_1	x_0	z_1	z_0
0	0	0	1	0	0
0	0	1	–	0	1
0	1	–	–	1	0
1	–	–	–	1	1

Exercise 2

We have a ***2 way set-associative mapping*** cache memory with 2^{15} entries ~~per set~~. Each entry is hosting 16 data. How many bits are necessary for the TAG for a 32 bits address bus?
A) 19
B) 17
C) 15
D) None of previous answers because…

Solution: D



2-way-set means that there are 2 memory blocks:
- Globally there are 2^{15} lines (# Entries)
- Each block contains $2^{15}/2=2^{14}$ lines = # Sets
- Each line hosts $16=2^4$ cells = Entries size

The cache address has:
- `Offset` = $\log_2$(Entries size) = $\log_2 16$ = 4 bits
- `Index` = $\log_2$(# Sets) = $\log_2 2^{14}$ = 14 bits
- `TAG` = Address – Offset = 32–14–4 = 14 bits

Exercise 3

Vertical microprogramming…
A) Is a high level language programming technique
B) Does NOT require any output decoding hardware
C) Is NOT a way to design and implement control units
D) Is a synonym for microprogram counter
E) None of previous answers because…

Solution: E

Vertical microprogramming is a low-level approach or architecture for the control unit inside the CPU, in particular regarding the microinstructions' memory: all signals are stored in a compressed/encoded way, to have a slimmer memory (instead of a fat one, in which the coded microinstructions are ready for sending control signal). Because microinstructions are compressed, there needs to be a decompressor after the microinstructions that restores them as the corresponding control signal.

Exercise 4

We have a memory of 2^{13} cells organized in banks of 2^9 cells. The decoder used to select the proper bank is
A) It is necessary an encoder and not a decoder
B) 16-to-4
C) 4-to-16
D) None of previous answers because …

Solution: C

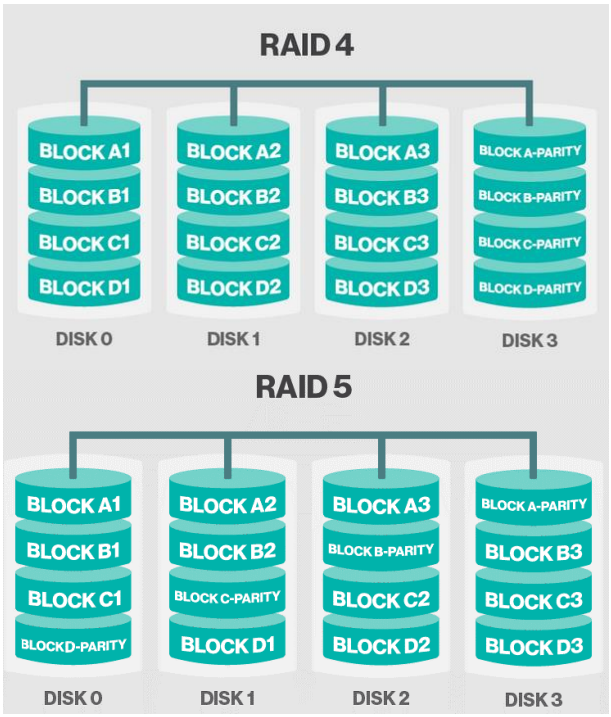
The number of banks is $2^{13}/2^9=2^4=16$.
The address bus needs 13 bits to address all of the cells.
We would need a **4-to-16** decoder, that takes 4 bits to select the correct block and uses the remaining 9 bits to address the cells inside them.

Exercise 6

Please shortly explain how works either RAID 4 or RAID 5, including advantages and drawbacks.

Solution:

See ⇨ *RAID* lecture.



Exercise 7

Please provide convincing explanations on what is the final value stored in AX (the contents of the other registers is of no interest) after having completed the execution of the following portion of assembly code. Assume that at the beginning AX stores a, BX b and CX c.

```
XOR AX, BX
XOR BX, CX
XOR CX, AX
XOR AX, BX
XOR AX, CX
```

Solution:

<AX>=a <BX>=b <CX>=c

Instruction	*Corresponding action*
XOR AX, BX	AX ← $a \oplus b$
XOR BX, CX	BX ← $b \oplus c$
XOR CX, AX	CX ← $c \oplus (a \oplus b)$
XOR AX, BX	AX ← $(a \oplus b) \oplus (b \oplus c) = a \oplus c$
XOR AX, CX	AX ← $(a \oplus c) \oplus (c \oplus a \oplus b) = b$

Final values:
<AX> = b
The content of AX becomes b.

Exercise 8

Please write down the microinstructions to implement the instruction XOR R0, R0 xoring the contents of R0 to the contents of R0 itself and writing the result to R0. Please add a short but significant explanation to each single microinstruction.

Solution:

$R0_{OUT}$, Y_{IN} , ALU_{XOR} ; Send R0 both to Y and ALU, do the XOR operation, store the result to Z
Z_{OUT} , $R0_{IN}$; Copy the content of Z back to R0

XOR R0, R0	
Workflow	*Microinstruction*
Send the value R0 to the internal bus, so it is: 1) stored to the temporary register Y 2) then it is read and processed by ALU Command ALU to take one operand from Y and the other from the internal bus, XOR them and store the result to Z	$R0_{OUT}$, Y_{IN} , ALU_{XOR}
Copy the content of Z back to R0	Z_{OUT} , $R0_{IN}$

Xoring a value to itself corresponds zeroing it:
<R0> XOR <R0> = 0

[$R0_{OUT}$, Y_{IN} , ALU_{XOR}]
is the optimised form of
[$R0_{OUT}$, Y_{IN}][$R0_{OUT}$, ALU_{XOR}]

Note: operations on the same line are not done in parallel, but consequently one after the other according to the right order.
When we use more than two operations per line, we take advantage of the already open communication output of the first one, so we don't need to close and reopen it (*optimisation*!).

► Test 2017-06-27 – Exercise 1, 2, 3, 4, 7, 8

Exercise 1

A ***priority decoder*** …
A) Implements the reverse function of a priority encoder
B) Is a necessary element to design a microcontrolled control unit
C) Has n inputs and n outputs
D) None of previous answers

Solution: D

Exercise 2

We have a ***full associative mapping*** cache memory with 2^{17} entries. Each entry is hosting 128 data. How many bits are necessary for the TAG for a 32 bits address bus?
A) 15
B) 8
C) 25
D) None of the previous answers because…

Solution: C
The number of lines is irrelevant because of the FAM.
The 32 bits A-BUS supports 2^{32} bits of total data.
The cache address has:
- Offset = $\log_2$(Entries size) = $\log_2 128 = 7$
- TAG = Address – Offset = 32–7 = 25 bits

Exercise 3

Manchester encoding ....
A) Is technique to design and implement software interfaces
B) Is a technique to design cache memories
C) Is a way to design and implement operating systems in software
D) Does not exist
E) None of the previous answers because…

Solution: E
Manchester encoding is used to solve the synchronization problem of reading data from Hard Drives: by encoding together clock and data as a single signal, saving one line (the other line is voltage/ground). This also avoids reserving a track on the disk for the clock only.
It considers transitions low-to-high and high-to-low as values 0 and 1. Decoding takes the signal, extracts the clock through a filter and uses it to sample and return information unpacked. To prevent misalignment, there is a delay between signal and sampler to make signal and clock synch.
See ⇨*Manchester encoding* lecture.

Exercise 4

We have a memory of 2^{16} cells organized in banks of 2^{12} cells. The decoder used to select the proper bank is
A) 16-to-16
B) 16-to-4
C) 4-to-16
D) None of previous answers because …

Solution: C
The number of banks is $2^{16}/2^{12}=2^4=16$.
The address bus needs 16 bits to address all of the cells.

We would need a **4-to-16** decoder, that takes 4 bits to select the correct block and uses the remaining 12 bits to address the cells inside them.

Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of CX.

```
OR AX, AX
AND BX, BX
XOR CX, CX
ADD CX, BX
SUB CX, AX
```

Solution:

<AX>=a <BX>=b <CX>=c

Instruction	*Corresponding action*
OR AX, AX	AX $\leftarrow a+a=a$
AND BX, BX	BX $\leftarrow bb=b$
XOR CX, CX	CX $\leftarrow c \oplus c = 0$
ADD CX, BX	CX $\leftarrow 0+b=b$
SUB CX, AX	CX $\leftarrow b-a$

Final values:
<CX> = $b-a$
The content of CX becomes the result of the subtraction between BX and AX.

Exercise 8

Please write down the microinstructions to implement the instruction SUB R0, R1 subtracting the contents of R1 from R0 and storing the final result back to R0. Please add a short but significant explanation to each single microinstruction.

Solution:

$R0_{OUT}$, Y_{IN} ; Copy R0 to the temporary register Y
$R1_{OUT}$, ALU_{SUB} ; Send R0 to ALU, do the SUB operation (the second operand from the bus is subtracted from the first operand stored into Y), store the result to Z
Z_{OUT} , $R0_{IN}$; Copy the content of Z back to R0

SUB R0, R1	
Workflow	*Microinstruction*
Copy R0 to the temporary register Y	$R0_{OUT}$, Y_{IN}
Send R1 to ALU Command ALU to take one operand (from the internal bus) and subtract it from the other (stored in Y), storing the result to Z	$R1_{OUT}$, ALU_{SUB}
Copy the content of Z back to R0	Z_{OUT} , $R0_{IN}$

► Test 2017-07-10 – Exercise 1, 2, 3, 7, 8

Exercise 1

A T flip-flop…
A) Implements the reverse function of a priority encoder
B) Complements its state in correspondence of the raising (or decreasing) edge of the clock input
C) Has a state value depending on the value of the level of the clock
D) None of previous answers

Solution: B

Exercise 2

We have a ***8-way set-associative mapping*** cache memory with 2^{17} sets (where each set has 8 entries). Each entry is hosting 16 data. How many bits are necessary for the TAG for a 28 bits address?
A) 15
B) 8
C) 11
D) None of previous answers because…

Solution: D

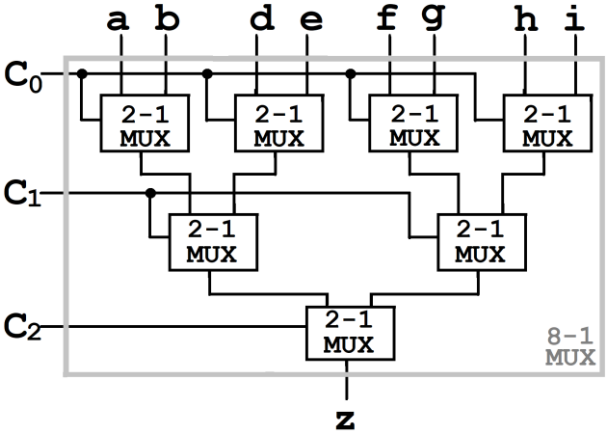
8-way-set means that there are 8 memory blocks:
- Globally there are $8 \times 2^{17} = 2^{20}$ lines (# Entries)
- # Sets = 2^{17}
- Each entry hosts $16 = 2^4$ cells = Entries size

The cache address has:
- Offset = $\log_2$(Entries size) = $\log_2 16$ = 4 bits
- Index = $\log_2$(# Sets) = $\log_2 2^{17}$ = 17 bits
- TAG = Address – Index – Offset = 28–17–4 = 7 bits

Exercise 3

The implementation of a 8-1 multiplexer following the hierarchical design technique....
A) Requires eight 2-1 multiplexers
B) Requires seven 2-1 multiplexers
C) Requires seven 2-1 multiplexers and one decoder
D) Requires four 4-1 multiplexers
E) None of previous answers because …

Solution: B



Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of AX and BX.

```
MOV BX, AX
SHL AX,8
SHR BX,8
ADD AX, BX
SHR BX,8
```

Solution: AX=[ah][al] BX=[bh][bl]

Instruction	*Corresponding action*
MOV BX, AX	BX ← AX=[ah][al] Content of AX is copied to BX
SHL AX,8	AX ← [al][0] Higher part of AX (AH) is replaced with the lower part (AL), by filling with 8 "0"s from right. Now AH=al, AL=0
SHR BX,8	BX ← [0][ah] Lower part of BX (BL=al) is replaced with the higher part (BH=ah), by filling with 8 "0"s from left. Now BH=0, BL=ah
ADD AX, BX	AX ← <AX> + <BX> = [al][ah] Higher part of AX (AH) remains the same, while the lower part (=0) gets the value from BX (BL=ah). Now AH=al,AL=ah
SHR BX,8	BX ← [0][0] BX is filled with other 8 "0"s from left, so it is completely zeroed. Now <BX>=0

Final values:
AX has its higher and lower parts swapped, BX is zeroed.

Exercise 8

Please write down the microinstructions to implement the instruction SUB R0, [R1] subtracting the contents of the cell pointed by R1 from R0 and storing the final result back to R0. Please add a short but significant explanation to each single microinstruction.

Solution:

$R1_{OUT}$, MAR_{IN},int ; Copy the address in R1 to MAR
Read from memory , $R0_{OUT}$, Y_{IN} ; copy R0 to Y_{IN} while reading from memory
MDR_{OUT},int , ALU_{SUB} ; content of R0 – content of cell pointed by R1
Z_{OUT} , $R0_{IN}$; Copy the content of Z back to R0

SUB R0, [R1]	
Workflow	*Microinstruction*
Copy the address in R1 to the memory address register MAR	$R1_{OUT}$, MAR_{IN},int
Do in parallel: - Reading from the memory the content of the cell pointed by R1 - copy R0 to Y_{IN}	Read from memory , $R0_{OUT}$, Y_{IN}
The data retrieved from the memory has been copied to MDR; send it to the internal bus. Command ALU to take one operand (from the internal bus) and subtract it from the other (stored in Y), storing the result to Z	MDR_{OUT},int , ALU_{SUB}
Copy the content of Z back to R0	Z_{OUT} , $R0_{IN}$

► Test 2017-09-04 – Exercise 1, 2, 3, 7, 8

Exercise 1

A T flip-flop…
A) Is a combinational element
B) Is usually used when designing ***synchronous*** counters
C) Changes its stored contents in correspondence of the raising (or decreasing) edge of the clock input
D) None of previous answers

Solution: B

Answer C could have been correct but a T flip-flop:
- "complements" (different from "change") its stored values
- only when both these conditions are met: input T=1 + clock edge.

Exercise 2

We have a ***2-way set-associative mapping*** cache memory with 2^{17} entries distributed (in pairs) across the sets (i.e., each set has 2 entries). Each entry is hosting 16 data. How many bits are necessary for the TAG for a 28 bits address?
(HINT: compute first the number of sets!)
A) 15
B) 8
C) 11
D) None of previous answers because…

Solution: B

2-way-set means that there are 2 memory blocks:
- # Entries = 2^{17}
- # Sets = (# Entries) / k = 2^{17} / 2 = 2^{16}
- Each entry hosts 16=2^4 cells = Entries size

The cache address has:
- Offset = $\log_2$(Entries size) = $\log_2 16$ = 4 bits
- Index = $\log_2$(# Sets) = $\log_2 2^{16}$ = 16
- TAG = Address – Offset = 28–16–4 = 8 bits

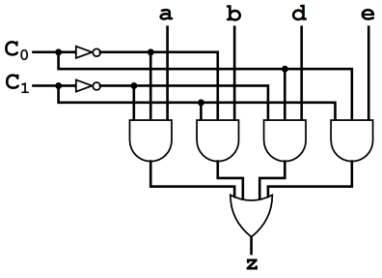
Exercise 3

The implementation of a 4-1 multiplexer NOT following the hierarchical design technique and using gates with fan in and fan out = 4, …
A) Requires 4 AND gates & 1 OR gate, on 2 levels
B) Requires 8 AND gates & 2 OR gates, on 3 levels
C) Requires 2 AND gates & 1 OR gate, on 2 levels
D) Requires 4 AND gates & 1 OR gate, on 3 levels
E) None of previous answers because …

Solution: A

From the truth table of the 4-1 MUX, I obtain its function and therefore its diagram.

$$z = \overline{c_0}\,\overline{c_1}a + \overline{c_0}c_1b + c_0\overline{c_1}d + c_0c_1e$$



C_0	C_1	z
0	0	a
0	1	b
1	0	d
1	1	e

Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of AX and BX.

```
XOR BX, BX
OR AX, BX
AND BX, AX
ADD BX, AX
```

Solution:

<AX>=a <BX>=b

Instruction	*Corresponding action*
XOR BX, BX	BX $\leftarrow b \oplus b = 0$
OR AX, BX	AX $\leftarrow a+0 = a$
AND BX, AX	BX $\leftarrow 0 \cdot a = 0$
ADD BX, AX	BX $\leftarrow 0+a = a$

Final values:

<AX>=a <BX>=a

AX does not change, BX takes the value of AX.

Exercise 8

Please write down the microinstructions to implement the instruction NOT [R1] complementing (to one) the contents of the cell pointed by R1 storing the final result back to the same cell pointed by R1. Please add a short but significant explanation to each single microinstruction.

Solution:

$R1_{OUT}$, MAR_{IN},int ; Copy the address in R1 to MAR
Read from memory ; here is a delay of time to retrieve the data from the RAM
MDR_{OUT},int , ALU_{NOT} ; content of cell pointed by R1 is sent to ALU and negated
Z_{OUT} , MDR_{IN},int ; Copy the content of Z to MDR
Write to memory ; here is a delay of time to write the data to the RAM

NOT R0, [R1]	
Workflow	*Microinstruction*
Copy the address in R1 to the memory address register MAR	$R1_{OUT}$, MAR_{IN},int
There is a delay of time to retrieve the data from the RAM	Read from memory
The data retrieved from the memory has been copied to MDR; send it to the internal bus. Command ALU to read the internal bus and negate such value, storing the result to Z	MDR_{OUT},int , ALU_{SUB}
Copy the content of Z to MDR	Z_{OUT} , MDR_{IN},int
There is a delay of time to write the data back to the RAM	Write to memory

► Test 2018-01-22 – Exercise 1, 2, 3, 6, 7, 8

Exercise 1

A flip-flop…
A) Is a combinational element
B) Is a sequential element
C) Is the basic block of a CLA
D) None of previous answers

Solution: B

Exercise 2

We have a ***full associative*** cache memory with 2^{17} entries. Each entry is hosting 32 data. How many bits are necessary for the TAG for a 28 bits address?
A) 6
B) 12
C) 23
D) None of the previous answers because…

Solution: C

The cache address has:
- Offset = $\log_2$(Entries size) = $\log_2 32 = 5$
- TAG = Address – Offset = 28–5 = 23 bits

Exercise 3

The implementation of a 4-1 multiplexer following the hierarchical design technique…
A) Requires four 2-1 multiplexers on one level
B) Requires three 2-1 multiplexers on one level
C) Requires four 2-1 multiplexers on two levels
D) Requires three 2-1 multiplexers on two levels
E) None of previous answers because …

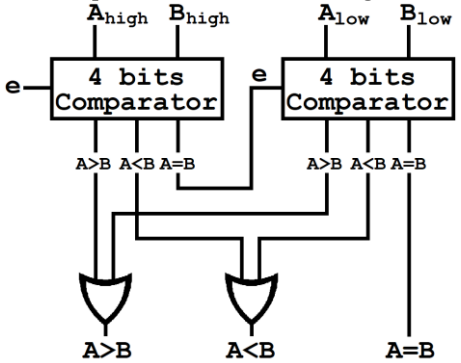
Solution: D

Exercise 6

Please shortly explain and provide a significant picture about how to design an architecture for comparing two 8-bits binary numbers A and B, using comparators for two 4-bits binary numbers, plus some other logical gates.

Solution:

The left comparator will process the 4 MSB, the right comparator will process the remaining 4 LSB. A=B from the left comparator acts as enable signal of the right one.



Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of AX and BX.

```
XOR BX, BX
XOR BX, AX
AND BX, AX
XOR BX, AX
```

Solution:

<AX>=a <BX>=b

Instruction	*Corresponding action*
XOR BX, BX	BX $\leftarrow b \oplus b = 0$
XOR BX, AX	BX $\leftarrow 0 \oplus a = a$
AND BX, AX	BX $\leftarrow a \cdot a = a$
XOR BX, AX	BX $\leftarrow a \oplus a = 0$

Final values:

<AX>=a <BX>=0

BX is cleared, AX remains the same.

Exercise 8

Please write down the microinstructions to implement the instruction INC [R1] incrementing (by one) the contents of the cell pointed by R1, storing the final result back to the same cell pointed by R1. Please add a short but significant explanation to each single microinstruction.

Solution:

$R1_{OUT}$, MAR_{IN},int ; Copy the address in R1 to MAR
Read from memory ; here is a delay of time to retrieve the data from the RAM
MDR_{OUT},int , ALU_{INC} ; content of cell pointed by R1 is sent to ALU and incremented
Z_{OUT} , MDR_{IN},int ; Copy the content of Z to MDR
Write to memory ; here is a delay of time to write the data to the RAM

INC [R1]	
Workflow	*Microinstruction*
Copy the address in R1 to the memory address register MAR	$R1_{OUT}$, MAR_{IN},int
There is a delay of time to retrieve the data from the RAM	Read from memory
The data retrieved from the memory has been copied to MDR; send it to the internal bus. Command ALU to read the internal bus and increment such value, storing the result to Z	MDR_{OUT},int , ALU_{INC}
Copy the content of Z to MDR	Z_{OUT} , MDR_{IN},int
There is a delay of time to write the data back to the RAM	Write to memory

► Test 2019-06-10 – Exercise 3, 5, 8

Exercise 3

A vertical microprogrammed memory…

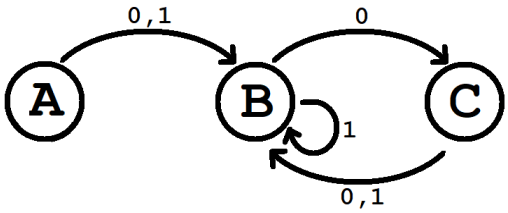
A) Is a virtual device
B) Is used as main memory in a computer
C) Is normally found in a control unit
D) Is always larger than a horizontally microprogrammed memory
E) Is normally used to store user's data and instructions
F) Is commonly used in the interface of the main memory

Solution: C

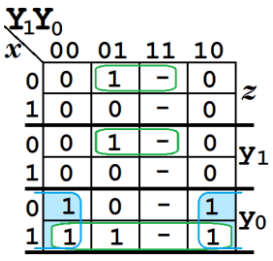
Exercise 5

Please provide the picture of the state machine of a sequencer recognizing the serial input –0 and restarting the search, **without overlap**, in correspondence of the next symbol received after one recognition is completed. Please also provide the maps and equations of the final circuit implementing the sequencer.

Solution:



	I am …	*Input value* x	Current state $Y_1\ Y_0$	Go to	Future state $y_1\ y_0$	*Output value* z
Nothing found	A	0	0 0	B	0 1	0
	A	1	0 0	B	0 1	0
First found	B	0	0 1	C	1 0	**1**
	B	1	0 1	B	0 1	0
String found	C	0	1 0	B	0 1	0
	C	1	1 0	B	0 1	0
–	D	–	1 1	–	– –	–



Functions: $z=\bar{x}Y_0$ $y_1=\bar{x}Y_0$ $y_0=x+\overline{Y_0}$

Exercise 8

Please write down the microinstructions to implement the instruction `XOR R1,R1` s xoring, bit by bit, the contents of `R1` with the contents of `R1`, and writing the result back into `R1`. Please add a short but significant explanation to each single microinstruction.

Please DO NOT replace this instruction with `MOV R1, 0`. If your solution is one of the possible optimized, 0.5 additional points will be awarded.

Solution:

$R1_{OUT}$, Y_{IN} , ALU_{XOR} ; Send R1 both to Y and ALU, do the XOR operation, store the result to Z

Z_{OUT} , $R1_{IN}$; Copy the content of Z back to R1

XOR R1, R1	
Workflow	*Microinstruction*
Send the value R0 to the internal bus, so it is: 1) stored to the temporary register Y 2) then it is read and processed by ALU Command ALU to take one operand from Y and the other from the internal bus, XOR them and store the result to Z	$R1_{OUT}$, Y_{IN} , ALU_{XOR}
Copy the content of Z back to R1	Z_{OUT} , $R1_{IN}$

Xoring a value to itself corresponds zeroing it:
`<R1> XOR <R1> = 0`

[$R1_{OUT}$, Y_{IN} , ALU_{XOR}]
is the optimised form of
[$R1_{OUT}$, Y_{IN}][$R1_{OUT}$, ALU_{XOR}]

Important note!

Let's compare `XOR R1, R1` with `MOV R1, 0`.
The microinstruction to implement `MOV R1, 0` would be:

IR_{OUT} , $R1_{IN}$; Copy the constant "0" from IR to R1

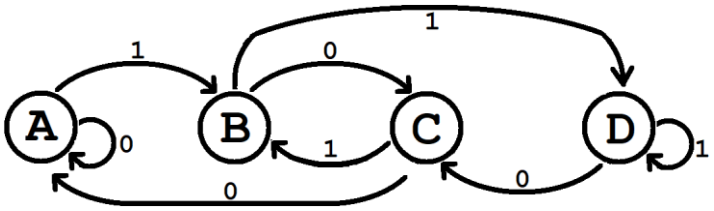
In both cases R1 is zeroed: *they are equivalent instructions*!
What tells us that we the XOR operation is present inside the ALU?
The fact that it is present in the assembly code which we are given means it is also present in the ***CPU's instructions set***: using the `MOV` operation would give us the same result, but we can do that only in the case the `XOR` was not present, otherwise ***it would be an error*** saying that `XOR R1,R1` is implemented with the same microinstructions as `MOV R1, 0`.

► Test 2019-06-25 – Exercise 5, 8

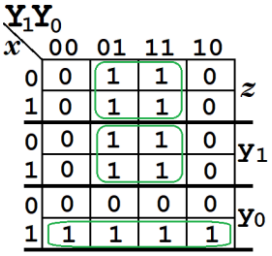
Exercise 5

Please provide the picture of the state machine of a sequencer recognizing the serial input 1– and restarting the search, with overlap, in correspondence of the next symbol received after one recognition is completed. Please also provide the maps and equations of the final circuit implementing the sequencer.

Solution:



	I am in	*Input value* x	Current state Y_1 Y_0	Go to	Future state y_1 y_0	*Output value* z
Nothing found	A	0	0 0	A	0 0	0
	A	1	0 0	B	0 1	0
First found	B	0	0 1	C	1 0	1
	B	1	0 1	D	1 1	1
Second found (0)	C	0	1 0	A	0 0	0
	C	1	1 0	B	0 1	0
Second found (1)	D	0	1 1	C	1 0	1
	D	1	1 1	D	1 1	1



Functions: $z=Y_0$ $y_1=Y_0$ $y_0=x$

Exercise 8

Please write down the microinstructions to implement the instruction `SUB [R1],5` subtracting 5 the memory cell pointed by R1, storing the final result back to the same cell pointed by R1. Please add a short but significant explanation to each single microinstruction.

Solution:

$R1_{OUT}$, MAR_{IN},int ; copy address in R1 to MAR

Read from memory , IR_{OUT} , Y_{IN} ; retrieve data from memory, copy the constant from IR to temporary register Y

MDR_{OUT},int , ALU_{SUB} ; send the retrieved data to the ALU and do <cell>-5, store in Z

Z_{OUT} , MDR_{IN},int ; copy the result from Z to MDR

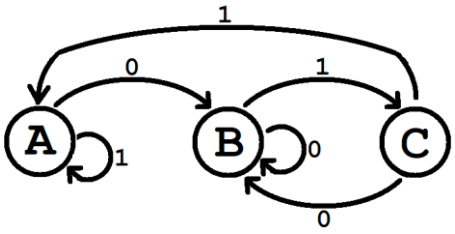
Write to memory ; address the cell pointed by R1 and write there the obtained value

► *Test 2019-07-08 – Exercise 5, 8*

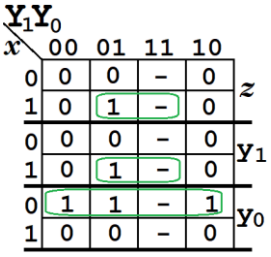
Exercise 5

Please provide the maps and equations of the final circuit implementing the sequencer recognizing the serial input 01 which, after a sequence recognition has been completed, immediately starts a new one (in correspondence of the next input), ***with overlap*** with the previous.

Solution:



	I am in	*Input value* x	Current state $Y_1\ Y_0$	Go to	Future state $y_1\ y_0$	*Output value* z
Nothing found	A	0	0 0	B	0 1	0
	A	1	0 0	A	0 0	0
First found	B	0	0 1	B	0 1	0
	B	1	0 1	C	1 0	1
Second found	C	0	1 0	B	0 1	0
	C	1	1 0	A	0 0	0
_	D	–	1 1	–	– –	_



Functions: $z=xY_0$ $y_1=xY_0$ $y_0=\bar{x}$

Exercise 8

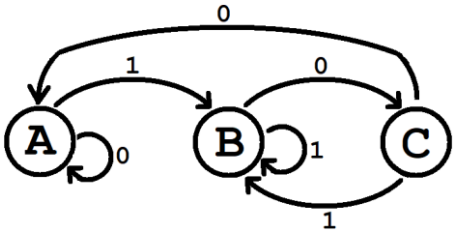
Please write down the microinstructions to implement the instruction `MOV [R1],0` storing 0 to the memory cell pointed by `R1`. Please add a short but significant explanation to each single microinstruction.

```
R1OUT , MARIN,int      ; Copy the address in R1 to MAR
IROUT , MDRIN,int      ; Copy the constant 0 from IR to
                         the memory data register MDR
Write to memory ; address the cell pointed by R1 and
                  write there the constant 0
```

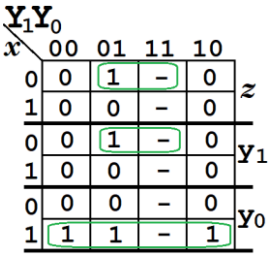
► *Test 2019-09-19 – Exercise 5*

Please provide the maps and equations of the final circuit (state variables and output) implementing the sequencer recognizing the serial input 10 which, after a sequence recognition has been completed, immediately starts a new one (in correspondence of the next input), with NO overlap with the previous.

Solution:



	I am in	*Input value* x	Current state $Y_1\ Y_0$	Go to	Future state $y_1\ y_0$	*Output value* z
Nothing found	A	0	0 0	A	0 0	0
	A	1	0 0	B	0 1	0
First found	B	0	0 1	C	1 0	1
	B	1	0 1	B	0 1	0
Second found	C	0	1 0	A	0 0	0
	C	1	1 0	B	0 1	0
_	D	–	1 1	–	– –	_



Functions: $z=\bar{x}Y_0$ $y_1=\bar{x}Y_0$ $y_0=x$

► *Test 2020-05-14 – Exercise 1, 2, 3, 4, 5, 6, 7, 8*

Exercise 1

A ripple carry adder on 6 bits …

A) Can be implemented by using AND and OR gates only
B) Requires a priority encoder in order to work
C) Is based on Full Adders where the carry flows from right to left
D) None of previous answers

Solution: C

Exercise 2

We have a ***direct mapping*** cache memory with 2^{16} entries. Each entry is hosting 64 data. How many bits are necessary for the `TAG` for a 32 bits address?

A) 6
B) 10
C) 16
D) None of the previous answers

Solution: B

The cache address has:

- `Offset` = $\log_2$(Entries size) = $\log_2 64 = 6$
- `Index` = log2(# Entries) = $\log_2 2^{16} = 16$
- `TAG` = Address – Index – Offset = 32–16–6= 10 bits

Exercise 3

In order to get a benefit from a cache, it is necessary that (PLEASE MARK ALL ANSWERS THAT ARE CERTAINLY CORRECT)

A) The hit ratio is reasonably small
B) The cache is reasonably slower than the RAM
C) The control unit is horizontally microprogrammed
D) The control unit is vertically microprogrammed
E) None of previous answers

Solution: E

Exercise 4

The boolean function for computing the sum bit in a Full Adder (with input a, b, c) is

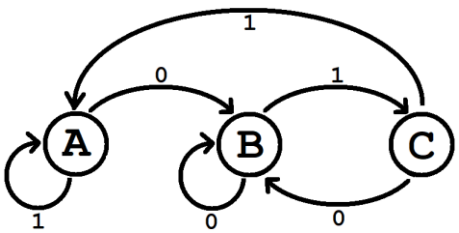
A) a XOR b XOR c
B) ab+ac+bc
C) Does not exist
D) None of previous answers

Solution: A

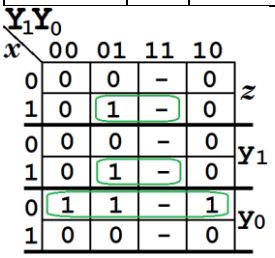
Exercise 5

Please provide the maps and equations of the final circuit (state variables and output) implementing the sequencer recognizing the serial input 01 which, after a sequence recognition has been completed, ***immediately starts*** a new one (in correspondence of the next input), with ***NO overlap*** with the previous.

Solution:



	I am in	*Input value*	Current state	Go to	Future state	*Output value*
		x	$Y_1\ Y_0$		$y_1\ y_0$	z
Nothing found	A	0	0 0	B	0 1	0
	A	1	0 0	A	0 0	0
First found	B	0	0 1	B	0 1	0
	B	1	0 1	C	1 0	1
Second found	C	0	1 0	B	0 1	0
	C	1	1 0	A	0 0	0
_	D	–	1 1	–	– –	_



Functions: $z = xY_0$ $y_1 = xY_0$ $y_0 = \bar{x}$

Exercise 6

Please shortly explain what is the Manchester encoding, how it works and why it is useful. Please provide significant and clear picture(s).

Exercise 7

Please provide convincing explanations on what the following portion of assembly code is doing (as a whole and not just as single instructions), in terms of final contents of `AX`; please consider only the case when `AX<128`.

```
MOV CX,AX
ADD AX,CX
SHL AX,1   ; shift left ax by one position
```

Solution:

<AX>=a <CX>=c

Instruction	*Corresponding action*
`MOV CX,AX`	`CX` ← a
`ADD AX,CX`	`AX` ← $a+a = 2a$
`SHL AX,1`	`AX` ← $2a \times 2 = 4a$

Final values:

<AX>=$4a$ <CX>=a

Content of AX is copied into CX, then content of AX is multiplied by 4.

Exercise 8

Please write down the microinstructions to implement the instruction `ADD R0, [R1]`. For each microinstruction, please provide a significant comment on the operation(s) performed.

Solution:

`R1`$_{OUT}$, `MAR`$_{IN}$`,int` ; Copy the address in `R1` to `MAR`

`Read from memory` , `R0`$_{OUT}$, `Y`$_{IN}$; while retrieving the data from memory, copy content of `R0` to the register `Y`

`MDR`$_{OUT}$`,int` , `ALU`$_{ADD}$; The `ALU` does the addition of the content of `Y` with the content of `MDR` (where was stored the data retrieved from RAM)

`Z`$_{OUT}$, `R0`$_{IN}$; Copy the result of the operation from `Z` to `R0`

► *Test 2020-05-21 – Exercise 1, 2, 3, 4, 5, 6, 7, 8*

Exercise 1

A D-flip flop…

A) Consumes more energy than a dynamic memory bit cell
B) Can be obtained by a T flip-flop
C) Can be obtained from a multiplexer
D) None of previous answers

Solution: A

Exercise 2

We have a ***direct mapping*** cache memory with globally 2^{16} lines. Each entry is hosting 128 data. How many bits are necessary for the `TAG` for a 32 bits address bus?

A) 9
B) 12
C) 26
D) None of the previous answers

Solution: A

The cache address has:

- `Offset` = $\log_2$(Entries size) = $\log_2 128 = 7$
- `Index` = log2(# Entries) = $\log_2 2^{16} = 16$
- `TAG` = Address – Index – Offset = 32–16–7= 9 bits

Exercise 3

We have a memory of 2^{16} cells organized in banks of 2^{13} cells. The decoder used to select the proper bank is

A) 8-to-3
B) Priority, 8-to-3
C) 16-to-4
D) None of previous answers

Solution: D

# banks: $2^{16}/2^{13}=2^3=8$
Address = $\log_2 2^{16}$ = 16 bits = 13 to cells + 3 to decoder.
We need a **3-to-8** decoder.

Exercise 4

A microprogrammed memory .... Please mark all answers that are correct (at least one is correct).

A) is a virtual device
B) is strongly related to the virtual memory
C) contains the microinstructions
D) is normally found in a control unit
E) is normally used as a second level cache memory
F) is a component of a peripheral's interface

Solution: C, D

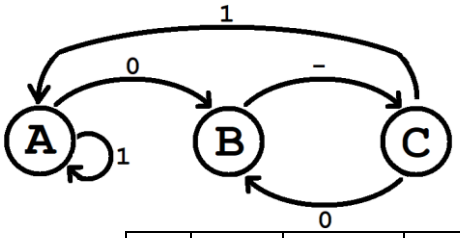
Exercise 6

Please explain how an interface virtualizes a peripheral device to the CPU. Please also use pictures.

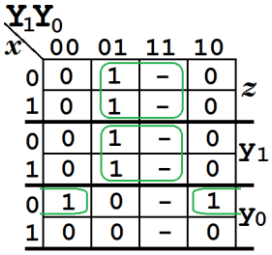
Exercise 5

Please provide the picture of the state machine of a sequencer recognizing the serial input 0– and restarting the search, ***without overlap***, in correspondence of the next symbol received after one recognition is completed. Please also provide the maps and equations of the final circuit implementing the sequencer.

Solution:



	I am in	*Input value* x	Current state $Y_1\ Y_0$	Go to	Future state $y_1\ y_0$	*Output value* z
Nothing found	A	0	0 0	B	0 1	0
	A	1	0 0	A	0 0	0
First found	B	0	0 1	C	1 0	1
	B	1	0 1	C	1 0	1
Second found	C	0	1 0	B	0 1	0
	C	1	1 0	A	0 0	0
–	D	–	1 1	–	– –	–



Functions: $z=Y_0$ $y_1=Y_0$ $y_0=\bar{x}\,\overline{Y_0}$

Exercise 7

Please provide convincing explanations on what is the final value stored in `AX` (the contents of the other registers is of no interest) after having completed the execution of the following portion of assembly code. Assume that at the beginning `AX` stores a, `BX` b and `CX` c.

```
MOV CX, BX
XOR AX, BX
XOR AX, CX
SUB CX, BX
XOR AX, CX
```

Solution:

<AX>=a <BX>=b <CX>=c

Instruction	*Corresponding action*
`MOV CX, BX`	`CX` $\leftarrow b$
`XOR AX, BX`	`AX` $\leftarrow a \oplus b$
`XOR AX, CX`	`AX` $\leftarrow (a \oplus b) \oplus b = a$
`SUB CX, BX`	`CX` $\leftarrow b - b = 0$
`XOR AX, CX`	`AX` $\leftarrow a \oplus 0 = a$

Final values: <AX>=a <BX>=b <CX>=0
`AX` has the same value as at the beginning.

Exercise 8

Please write down the microinstructions to implement the instruction `DEC R1`, decrementing the contents of `R1` and rewriting the result in the same `R1`. Please add a short but significant explanation to each single microinstruction.

Solution:

```
R1OUT , ALUDEC   ; R1 content is sent to the bus, the ALU
                   reads it and does the decrement by 1, the
                   result is stored to Z
ZOUT , R1IN      ; Copy the result from Z back to R1
```

► *Test 2020-06-26 – Exercise 1, 2, 3, 4, 5, 6, 7, 8*

Exercise 1

A T-flip flop…
A) Is a combinational device
B) Can be used to replace in full a D flip-flop (i.e. it has exactly the same functions and design as a D flip-flop, but just a different name for the same device)
C) Can be used in the design of asynchronous counters
D) None of other answers
Solution: C

Exercise 2

We have a ***direct mapping*** cache memory with globally 2^{16} lines. Each entry is hosting 32 data. How many bits are necessary for the TAG for a 28 bits address bus?
A) 7
B) 12
C) 23
D) None of the previous answers
Solution: A
The cache address has:
- Offset = $\log_2$(Entries size) = $\log_2 32 = 5$
- Index = log2(# Entries) = $\log_2 2^{16} = 16$
- TAG = Address – Index – Offset = 28–16–5= 7 bits

Exercise 3

We have a memory of 2^{18} cells organized in banks of 2^{14} cells. The decoder used to select the proper bank is
A) 4-to-2
B) 4-to-16
C) We do not need a decoder, but an encoder
D) None of previous answers
Solution: B
# banks: $2^{18}/2^{14}=2^4=16$
Address = $\log_2 2^{18}$ = 18 bits = 14 to cells + 4 to decoder.
We need a **4-to-16** decoder.

Exercise 4

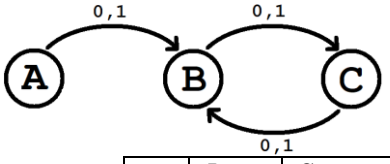
The instruction register... Please mark all answers that are correct (at least one is correct).
A) Stores a microinstruction
B) Stores the code of the instruction currently being executed
C) Is a register which is found inside the CPU (= it is one of the blocks found in a CPU)
D) Is a component of the cache manager
E) Stores a copy of the program counter
F) None of the other answers
Solution: B, C

Exercise 6

Please explain clearly and schematically (i.e. not too much verbosely, but by points), the architecture of RAID 1, highlighting strengths and weaknesses with respect to a "normal" non-RAID disk architecture. Please use pictures to depict the architecture inclusive of the organization of data.
Solution: ⇨ RAID 0+1 technique (p. 62-63)

Exercise 5

Please provide the picture of the state machine of a sequencer recognizing the serial input – – (don't care, don't care) and restarting the search, ***without overlap***, as soon as the next symbol received after one recognition is completed. Please also provide the maps and equations of the final circuit implementing the sequencer.
Solution:



Found	I am in	*Input value* x	Current state $Y_1\ Y_0$	Go to	Future state $y_1\ y_0$	*Output value* z
Nothing	A	0	0 0	B	0 1	0
	A	1	0 0	B	0 1	0
First	B	0	0 1	C	1 0	1
	B	1	0 1	C	1 0	1
Second	C	0	1 0	B	0 1	0
	C	1	1 0	B	0 1	0
–	D	–	1 1	–	– –	–

x \ Y_1Y_0	00	01	11	10	
0	0	1	–	0	z
1	0	1	–	0	
0	0	1	–	0	y_1
1	0	1	–	0	
0	1	0	–	1	y_0
1	1	0	–	1	

Functions: $z=Y_0$ $y_1=Y_0$ $y_0=\overline{Y_0}$

Exercise 7

Please provide convincing explanations on what is the final value stored in AX (the contents of the other registers is of no interest) after having completed the execution of the following portion of assembly code. Assume that at the beginning AX stores a and BX b.
+ 1 extra point: what if the instruction XOR AX, BX is added to this program? How will the content of AX change?

```
MOV AX, BX
XOR BX, AX
XOR AX, BX
SUB AX, BX
```

Solution: <AX>=a <BX>=b

Instruction	*Corresponding action*
MOV AX, BX	AX ← b
XOR BX, AX	BX ← $b \oplus b = 0$
XOR AX, BX	AX ← $b \oplus 0 = b$
SUB AX, BX	CX ← $b - 0 = b$
XOR AX, BX	AX ← $b \oplus 0 = b$ *(extra)*

The final value of AX is the original content of BX: <AX>=b
Extra instruction: the content of AX does not change

Exercise 8

Please write down the microinstructions to implement the instruction INC R1, incrementing the contents of R1 and rewriting the result in the same R1. Please add a short but significant explanation to each single microinstruction.

Solution:

```
R1OUT , ALUINC   ; R1 content is sent to the bus, the ALU
                   reads it and does the increment operation
                   (add 1 when INC has only one operand),
                   the result is stored to Z
ZOUT , R1IN      ; Copy the result from Z back to R1
```

Multiple choice questions database

The boolean function for computing the equality between two bits a and b is:
True
$ab + \overline{ab} = (a\ AND\ b)\ OR\ ((NOT\ a)\ AND\ (NOT\ b))$ $\overline{a \oplus b} = NOT(a\ XOR\ b) = \big(a\ OR\ (NOT\ b)\big) AND ((NOT\ a) or\ b)$
False
(a AND (NOT b)) OR (b AND (NOT a)) a AND b

The boolean function for computing the sum bit in a Full Adder (with input a, b, c) is
True
a XOR b XOR c
False
ab+ac+bc

The boolean function for computing the carry bit in a Full Adder (with input a, b, c) is:
True
ab+ac+bc
False
a XOR b XOR c abc+ac+NOT(a)b

A 4-bit ripple carry adder
True
Is a combinational circuit Is based on Full Adders where the carry flows from right to left
False
Requires only AND and OR gates to be implemented Requires only 8 multiplexers, without additional logic Can be implemented by using D flip-flops Can store up to 16 values Requires a priority encoder in order to work

A carry lookahead adder on 8 bits
True
False
Is, very likely, not worth to be implemented Requires 8 cascaded full adders in order to work Is a sequential circuit

Sequential circuits
True
Can be modelled by the Huffman model Include registers and memories Present some outputs which are looped back to the input Require combinational gates to be implemented
False

A flip-flop…
True
Is a sequential element
False
Is a combinational element Is the basic block of a CLA

A SR flip-flop…
True
Is a sequential element Always requires a reset in order to work properly
False
Can be obtained from a D flip-flop Consumes more energy than a static memory bit cell

A D-flip-flop…
True
Can be obtained from a SR flip-flop Implements a static memory cell Consumes more energy than a dynamic memory bit cell Is an element of a serial adder
False
is the basic block for designing an asynchronous counter is a combinational block / circuit is (basically) a dynamic memory cell Can be obtained by a T flip-flop Can be obtained from a multiplexer Requires a capacitor to be correctly implemented Is the basic element of a bus Implements a dynamic memory cell Consumes less energy than a dynamic memory bit cell Is the basic element of a parallel adder Sometimes requires an initialization in order to work properly

A T flip-flop…
True
Complements its state in correspondence of the raising (or decreasing) edge of the clock input Is usually used when designing synchronous counters
False
Is a combinational element Changes its stored contents in correspondence of the raising (or decreasing) edge of the clock input Has a state value depending on the value of the level of the clock Implements the reverse function of a priority encoder

A synchronous counter:
True
Is a sequential circuit Can be based on T-Flip-Flops
False
Normally is based on F-Flip-Flops Normally is based on multiplexers

A 4-1 multiplexer…
True
Is a combinational circuit
False
Can be obtained from a SR flip-flop Can be implemented using AND gates only Is a sequential circuit Is a necessary element to design a full adder

A priority encoder …
True
Has 2^n inputs and n outputs
False
Requires only OR and AND gates Is based on OR ports only Has the number of outputs which is always larger than the number of inputs Has the same truth table of a non-priority encoder Is a sequential circuit Is the basic element of a parallel adder Is a necessary element to design a full adder Can be implemented by using a D flip flop

A priority decoder …
True
Does not exist
False
Implements the reverse function of a priority encoder Is a necessary element to design a microcontrolled control unit Has n inputs and n outputs Is a ROM-based memory circuit Is a necessary element to design a full adder

Manchester encoding ...
True
False
Is technique to design and implement software interfaces Is a technique to design cache memories Is a way to design and implement operating systems in software Does not exist

In order to get a benefit from a cache, it is necessary that
True
False
The hit ratio is reasonably small The cache is reasonably slower than the RAM The cache is not too much slower than the RAM The control unit is horizontally microprogrammed The control unit is vertically microprogrammed The size of a block is at most two bytes The cache is less expensive than the RAM

Memory mapped…
True
is a specifically targeted to the addressing of peripheral devices cannot efficiently handle very slow interfaces does not require an additional bus, to data, address and control
False
is most of times faster than an interrupt controller is a combinational element requires one additional bus is generally more hardware-expensive than isolated I/O cannot be used for hard disks is a virtual device

Isolated I/O…
True
Is a technique (architecture) to address peripheral devices
False
Is an approach to implement registers Is a technique to design ALUs Is a way to design and implement the control units Is a synonym for microprogram counter Is a software emulation of peripheral devices Is a way to design and implement peripheral devices Is a type of peripheral interface Is a hardware emulation of peripheral devices

Microprogramming...
True
Is a way to design and implement the control units Is a mandatory component for parallel machines
False
Is an assembly language Is a way to design and implement memories Is a name of a particular flip flop Is commonly used in the interface of the main memory

Horizontal microprogramming...
True
Is a way to design and implement the control units
False
Is technique to design and implement software interfaces Is a technique to design cache memories Is a way to design and implement operating systems in software

Vertical microprogramming...
True
For the same instruction set and architecture, requires a smaller microcode memory than the corresponding horizontal microprogramming Is a way to design and implement the control units
False
Is a high level language programming technique Does NOT require any output decoding hardware Is NOT a way to design and implement control units Is a synonym for microprogram counter Is faster than horizontal microprogramming Is a synonym for microprogram counter Is commonly used in the interface of the main memory Is an approach to implement registers Is a technique to design ALUs Is a technique to design RAM memories Is a synonym for writing short and portable programs Is a way to design and implement software operating systems

A microprogrammed control unit has the role...
True
To decode the instructions received from the IR and generate the corresponding control signals
False
To decode and control the elements of a computer architecture, in order to have a correct execution Of offering the computational support to the execution of the instructions To control the operation modes of the caches To provide a correct management of the RAM

A vertically microprogrammed control unit...
True
Is usually slower than the corresponding horizontally microprogrammed unit
False
Is not an internal part of the CPU Is not a hardware physical component Is, in general not too much reliable

A vertically microprogrammed memory...
True
Is normally found in a control unit
False
Is a virtual device Is used as main memory in a computer Is always larger than a horizontally microprogrammed memory Is normally used to store user's data and instructions Is commonly used in the interface of the main memory

The instruction register...
True
Stores the code of the instruction currently being executed Is a register which is found inside the CPU (= it is one of the blocks found in a CPU)
False
Stores a microinstruction Is a component of the cache manager Stores a copy of the program counter

*When a microprocessor receives an interrupt request **through** an interrupt controller…*
True
one or more peripherals are requesting the "attention" of the CPU (i.e. the execution of their corresponding service routine)
False
the interrupt controller requires a reprogramming by the CPU the CPU immediately stops executing the current instruction and starts executing an interrupt management routine

Theory questions database

⇨ *Comparator*
Please shortly explain and provide a significant picture about how to design an architecture for comparing two 8-bits binary numbers A and B, using comparators for two 4-bits binary numbers, plus some other logical gates.
► Test 2018-01-22 – Exercise 6 (p. 112)
► Test 2019-07-08 – Exercise 6 (p. 78)

⇨ *Multiplexer*
Please depict the hierarchical design of a 8-1 multiplexer using 2-1 multiplexers. Please provide brief comments on how this system is working.
► Test 2017-07-10 – Exercise 3 (p. 110)

⇨ *Cache Address Mapping*
Please list and explain the advantages and drawbacks of direct mapping vs. full associative mapping.

Direct mapping: each main memory block is associated with only one possible cache line.
- Advantages: simple and inexpensive to implement.
- Drawbacks: there is a fixed cache location for any given block, so if a program happens to reference words repeatedly from two different blocks thrashing happens.

Full Associative Mapping: each main memory block can be loaded into any line of the cache.
- Advantages: gives flexibility as to which block to replace when a new block is read into the cache.
- Drawbacks: a complex circuitry is required to examine the TAGs of all cache lines in parallel, so it is expensive.

⇨ *Replacement Algorithms*
Please list what could be possible replacement policies of a cache line, by also highlighting the corresponding characteristics, advantages and drawbacks.

- Least recently used (LRU): overwrite the block not referenced for the longest time. It performs well for many access patterns, but can lead to poor performance in some cases (i.e. when accesses are made to sequential elements of an array that is slightly too large to fit into the cache).
- First in first out (FIFO): remove the "oldest" block from a full set when a new block must be brought in. It does not consider the recent pattern of access to blocks in the cache.
- Least frequently used (LFU): replace that block in the set that has experienced the fewest references.
- Random replacement (RR): it randomly chooses the block to be overwritten.

⇨ *Write policy*
Please list what are the write-to-ram cache policies, by highlighting advantages and drawbacks.

They correct mis-alignment between cache and memory.
- write through: all write operations are made to main memory as well as to the cache, so the main memory is always aligned. It generates a huge delay ($t_{writing} = t_{RAM}$).
- write back: updates are made only in the cache, minimizing memory writes. A dirty bit associated with the line is set in case of an update: when a block is replaced, it is written back to main memory only if the dirty bit is set. Portions of main memory will be invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck.

⇨ *Manchester encoding*
Please shortly explain what is the Manchester encoding, how it works and why it is useful (or "what are the problems solved by this type of encoding"). Please provide significant and clear picture(s).
► Test 2019-09-19 – Exercises 6 (p. 76)

⇨ *RAID*
Please shortly explain how works RAID 1, including advantages and drawbacks. Please also explain the main differences with RAID 5 (or 4, your choice)

Please shortly explain how works either:
- RAID 0
- RAID 0+1
- RAID 4
- RAID 5

including advantages and drawbacks. Please provide significant picture.

⇨ *I/O Organization*
Please explain, shortly and clearly, how an interface virtualizes a peripheral device to the CPU.
Please also use pictures.
► Test 2019-06-10 – Exercises 6 (p. 90)

⇨ *Peripherals addressing modes*
Please explain the memory mapped I/O framework, including advantages and drawbacks. Please also use pictures.
► Test 2020-02-10 – Exercise 6 (p89)

Please explain the Isolated I/O technique, including goals, advantages and drawbacks. Please also use pictures.

Please shortly list side-by-side the characteristics, advantages and drawbacks of the isolated I/O and memory mapped peripheral addressing modes.

⇨ *Control and Decoding Unit*
Please provide an explanation of what is the role of the control unit and how it works.

⇨ *CPU and peripheral devices interaction*
What are the main advantages and drawbacks of interrupt versus CPU polling? Please provide convincing explanations.

⇨ *Prediction and principle of locality + Pipelining*
Please discuss about the locality principles and what is their impact on the pipelining techniques.

Appendix – Bibliography

The notes are mostly an elaboration of Professor Montuschi lectures of Computer Architecture course at Politecnico di Torino.

Some other sources have been used, here are only the most helpful.

Andrea Spitale – Lecture Notes of Computer Architecture course (2014-2015) https://web.archive.org/web/20180505100117/https://staff.polito.it/paolo.montuschi/notes-from-my-courses.html

Books:

Stallings W. – Computer Organization and Architecture, 2016

Grosso, Prinetto, Rebaudengo, Sonza Reorda – La programmazione in Assembler x86

Sites:

Wikipedia - wikipedia.org

M.I.T. Computation Structures (lecture notes and videos) - computationstructures.org

Leoš Literak www.penguin.cz/~literakl/intel/intel.html

Microprocessor Design - en.wikibooks.org/wiki/Microprocessor_Design