ARC: Computer Architecture

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du April 29, 2024

Tanguy Risset

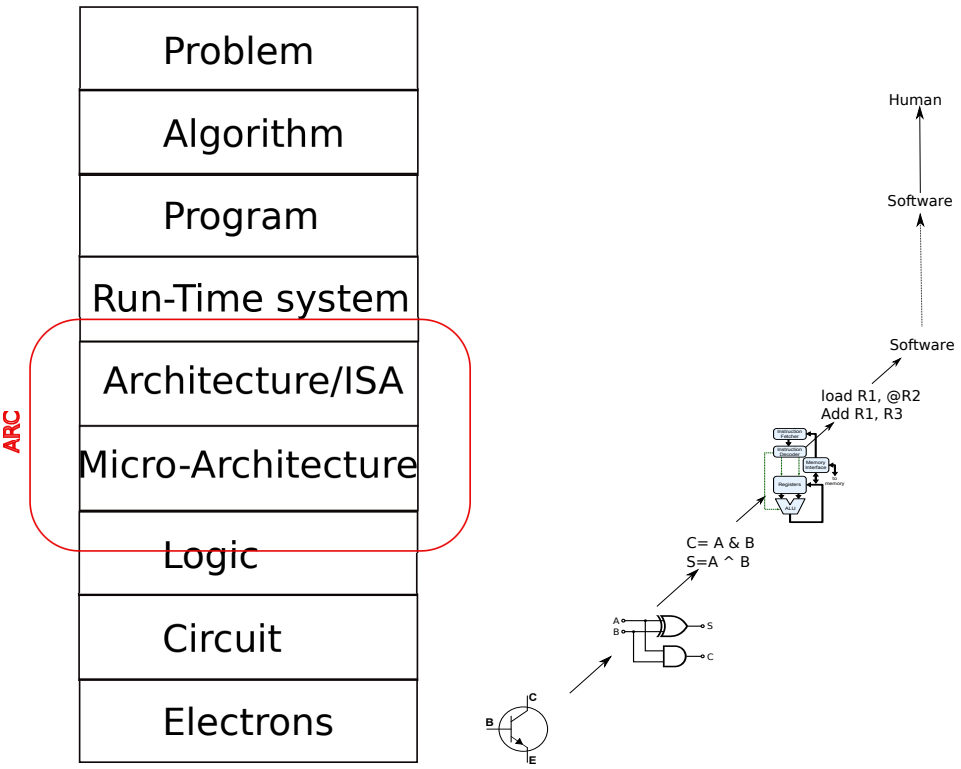
April 29, 2024

Table of Contents

ARC course presentation

- Schedule:
  - Course 6h
  - labs (TP) 20h
  - Evaluation (In french): un QCM et un devoir papier en fin de cours
- skills and knowledge learned in ARC cours:
  - Bolean logic, arithmetics
  - combinatorial and sequential logic circuits, automata.
  - Processor architecture, datapath, compilation process, RISC architecture
  - Assembly code, link with high level programming languages
  - Simple processor design, simple assembly program analysis.
  - Link with compilation, operating systems and programming
- Moddle (open): frames, labs, various document
- Course based on the two IF architecture course: AC and AO (open courses on Moodle).

From electron to Von-Newman CPU



Computer architecture usefulness

- How to solve a problem with electrons:
- ARC is useful
  - For general knowledge of a computer scientist
  - To understand pro/cons of modern complex architectures
  - For embedded system programming

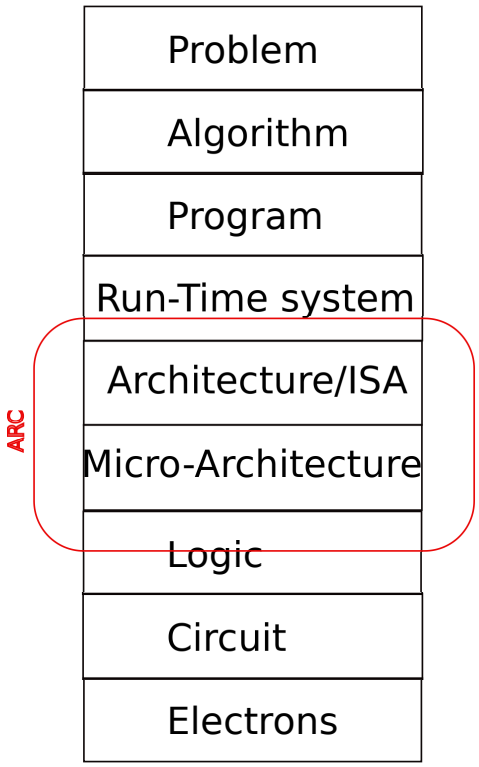
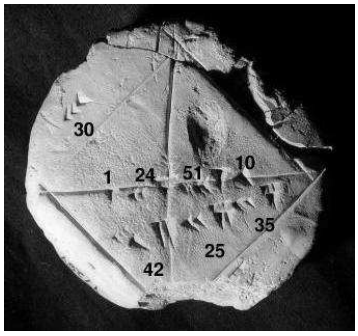


Table of Contents

History of computing

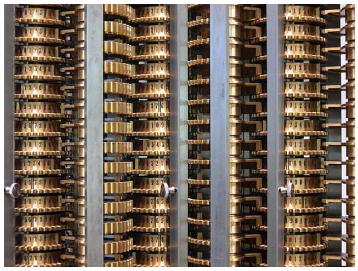
- Ancient time: various arithmetics systems
- 17th century (Pascal and Leibniz): notion of mechanical calculator
- 1822 Charles Babbage Difference engine (tabulate polynomial functions)
- 1854 Georges Boole proposes the so-called Boolean logic.
- (More details on the poly or on Internet)

from Yale Babylonian Collection, $\simeq$ 1600 BC



http://www.math.ubc.ca/~cass/Euclid/ybc/ybc.html

Difference Machine close-up

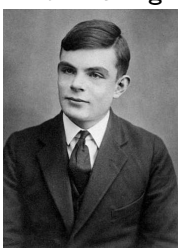


By By Carsten Ullrich - Own work, CC BY-SA 2.5

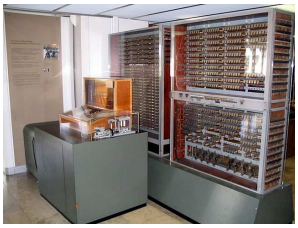
History of computers

- 1936: Alan Turing's PhD on a universal abstract machine
- 1941: Konrad Suze builds the Z3 first programmable computer (electro-mechanic)
- 1946: ENIAC is the first electronic calculator
- 1949: Turing and Von Neumann build the first universal electronic computer: the Manchester Mark 1
- (More details on the poly or on Internet)

Alan Turing



Z3 computer at Deutches Museum, Munich



By Venusianer, CC BY-SA 3.0

Manchester Mark 1 1948

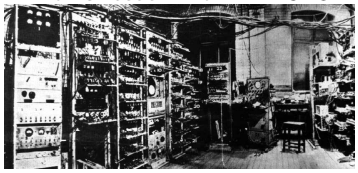
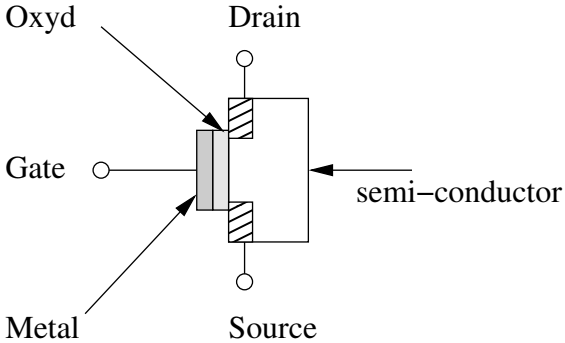


Table of Contents

Transistor

- Discovered in 1947 at Bell Labs: (**trans**fer res**istor**)
- Could replace the thermionic triode (vacuum tube) that allow radio and telephone technologies.
- Principle: flow or Interrupt current between Source and Drain, depending on Gate status

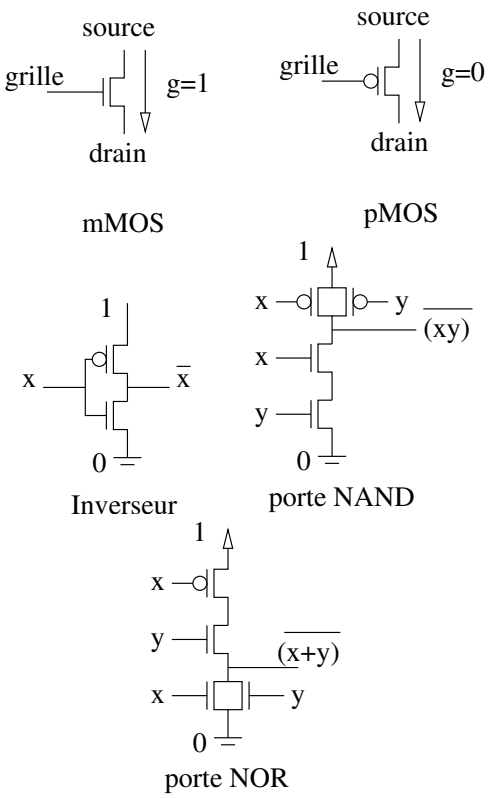
- Can be seen as a switch
- Wildly used after Integrated Circuit invention (1958)



Mosfet technology

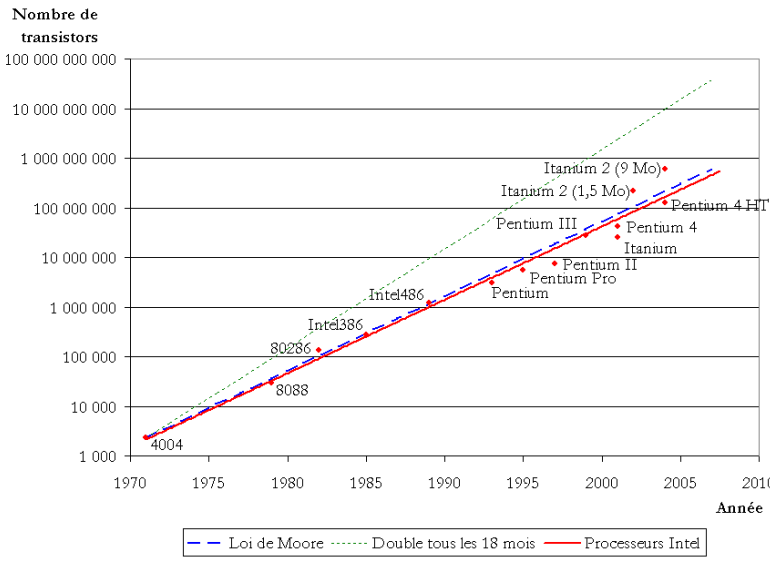
Popular Transistor technology: CMOS

- CMOS: Complementary Metal Oxide Semiconductor
- Two logical levels : 0 = 0V and 1 = 3V
- Two types of transistors
  - nMOS : current flows if gate is 1
  - pMOS : current flows if gate is 0
- Mainly used to realize basic logical gates (NOT, NAND, NOR, etc.)



Moore's low

- Gordon Moore, co-founder of Fairchild Semiconductor and Intel, predicted in "a doubling every two year in the number of components per integrated circuit"
- Contributed to world economic growth
- Slow down in 2015 and is ended now.



Boolean functions

Boole Algebra is equipped with three operations

- a unary operation, **negation**, noted `NOT`;
- two binary commutative, associative operations:
  - **conjunction** — `AND`, with 1 as neutral element;
  - **disjunction** — `OR`, with 0 as neutral element;
- `AND` is distributive over `OR`

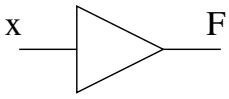
If a and b are 2 boolean variables, we write:

$$\text{NOT}(a) = \overline{a}, \quad \text{AND}(a, b) = ab = a.b, \quad \text{OR}(a, b) = a + b$$

Boolean Cheat Sheet

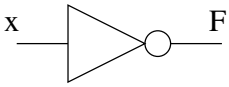
- neutral elements: $a + 0 = a, \quad a \cdot 1 = a$
- absorbing elements: $a + 1 = 1, \quad a \cdot 0 = 0$
- idempotence: $a + a = a, \quad a \cdot a = a$
- tautology/antilogy: $a + \overline{a} = 1, \quad a \cdot \overline{a} = 0$
- commutativity: $a + b = b + a, \quad ab = ba$
- distributivity: $a + (bc) = (a + b)(a + c), \quad a(b + c) = ab + ac$
- associativity: $a + (b + c) = (a + b) + c = a + b + c,$
  $a(bc) = (ab)c = abc$
- De Morgan's law: $\overline{ab} = \overline{a} + \overline{b},$
  $\overline{a + b} = \overline{a} \cdot \overline{b}$
- others: $a + (ab) = a, \quad a + (\overline{a}b) = a + b,$
  $a(a + b) = a, \quad (a + b)(a + \overline{b}) = a$

Elementary logical gates



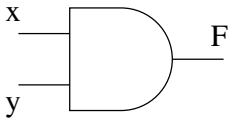
Amplifier:
$F = x$

x	F
0	0
1	1



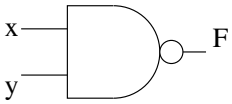
NOT: $F = \overline{x}$

x	F
0	1
1	0



AND: $F = x\ y$

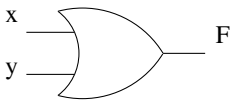
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1



NAND:
$F = \overline{(x\ y)}$

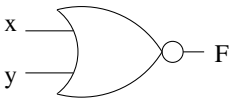
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

Elementary logical gates



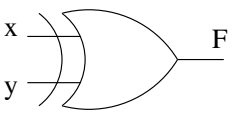
OR:
$F = x + y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1



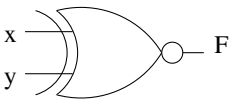
NOR:
$F = \overline{(x + y)}$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0



XOR:
$F = x \oplus y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0



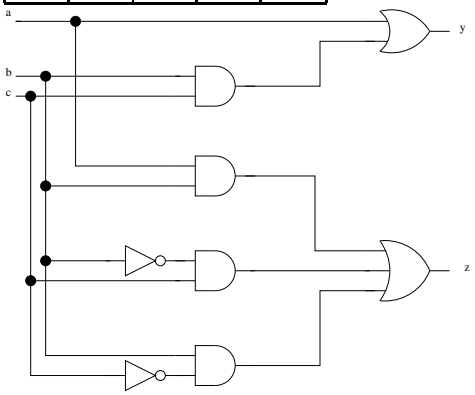
XNOR:
$F = x \odot y$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

Combinatorical circuit Design

1. Boolean description of the problem:
   - Compute y and z from a, b and c
   - y is 1 if a is 1 or b and c are 1.
   - z is 1 if b or c is 1 (but not both) or if a, b et c are 1.
2. Truth table
3. Logic equation
   - $y = \overline{a}bc + a\overline{b}\overline{c} + a\overline{b}c + ab\overline{c} + abc$
   - $z = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}c + ab\overline{c} + abc$
4. Optimized logic equations
   - $y = a + bc$
   - $z = ab + \overline{b}c + b\overline{c}$
5. logic gates

input			output	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1



Disjunctive Normal Form (DNF)

- In Boolean logic, a logical formula in Disjunctive Normal Form (*Forme normale disjonctive* in French) if:
  - It is a disjunction of one or more clauses
  - where the clauses are conjunction of literals
  - a literal is a variable, a constant or 'not' a variable
- Otherwise put, it is an OR of ANDs.
- Example of DNF:
  - $x.\bar{y}.\bar{z} + \bar{t}.u.v$
  - $(a \wedge b) \vee \neg c$
- Example not in DNF:
  - $\overline{(x + y)}$
  - $a \vee (b \wedge (c \vee d))$

Conjunctive Normal Form (CNF)

- In Boolean logic, a formula is in conjunctive normal form (*forme normale conjonctive* in French) if:
  - it is a conjunction of one or more clauses,
  - where a clause is a disjunction of literals;
  - a literal is a variable, a constant or 'not' a variable
- Otherwise put, it is an AND of ORs.
- Example of CNF:
  - $(x + y + \bar{z})(\bar{x} + z)$
  - $(a + \bar{b} + \bar{c})(\bar{d} + \bar{a})$
  - $x + y$
- Example not in CNF
  - $\overline{(x + y)}$
  - $x(y + (z.t))$

From Truth table to DNF

- Back to previous example (z is 1 if b or c is 1 (but not both) or if a, b et c are 1.)
- Truth table on the right, z is 1 if and only if one of the five condition identified occurs.
- It is easy to find a conjunction that is valid in a unique case: example: $\bar{a}.\bar{b}.c$ is 1 if and only if: $a = 0$, $b = 0$ and $c = 1$ (double arrow on the right)
- by adding all the conjunction valid only on each of the five cases identified on the right, we get a DNF formulae that has exactly that truth table.

input				
a	b	c	z	
0	0	0	0	
0	0	1	1	⇐
0	1	0	1	←
0	1	1	0	
1	0	0	0	
1	0	1	1	←
1	1	0	1	←
1	1	1	1	←

Hence the possible formulae for z: $z = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}c + ab\overline{c} + abc$

How can it be simplified?

Simple Boolean optimization: Karnaugh Table (1)

- Karnaugh map (*tables de Karnaugh*) use a "visual" reprentation of a simple property:
  $(a.\bar{b}) + (a.b) = a.(\bar{b} + b) = a$
- The first step in the method is to transform the truth table (3 or 4 input variables) of the function in a two-dimensional array (split into two parts of the set of variables)
- Rows and columns are indexed by the valuations of the corresponding variables in such a way that between two rows (or columns) only one boolean value changes.
- In our example (3 variables):

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

Simple Boolean optimization: Karnaugh Table (2)

- Then, we try to cover all '1' of the table by forming groups.
  - each group contains only adjacent '1'
  - must form a rectangle
  - the number of elements of a group must be a power of two.
- each group correspond to a possible optimization of the DNF
- In our example:

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

- example : Three groups:
  - $\bar{a}.b.\bar{c} + a.b.\bar{c}$ simplifies to $b.\bar{c}$
  - $a.b.\bar{c} + a.b.c$ simplifies to $a.b$
  - $a.\bar{b}.c + \bar{a}.\bar{b}.c$ simplifies to $\bar{b}.c$
- hence $z = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}c + ab\overline{c} + abc$ simplifies to $z = a.b + \bar{b}.c + b.\bar{c}$

Well formed cicruits

As far as combinatorial circuits are concerned, a "Well formed" circuit is:

- A logic gate
- A wire
- Two well formed circuits next to each other
- Two well formed circuits, the outputs of one being the inputs of the other
- Two well formed circuits sharing a common input

It can be shown that it correspond to an acyclic graph of logic gates.

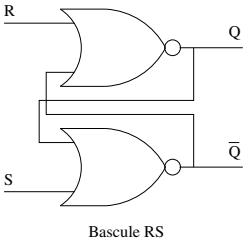
- No cycles, no ouptuts conected together

Usefull combinatorics logic components

- n input multiplexer
- decoder $log(n) \rightarrow n$
- n bits adder
- n bits comparator
- n bits ALU
- etc.

Memorizing: latches and Flip-Flops

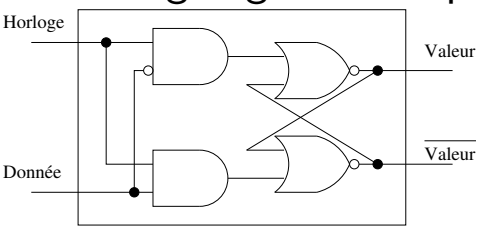
- Set-Reset Latch (SR latch, *Bascule RS*): When R and S are reset, Q and $\overline{Q}$ keep their previous value.



Bascule RS

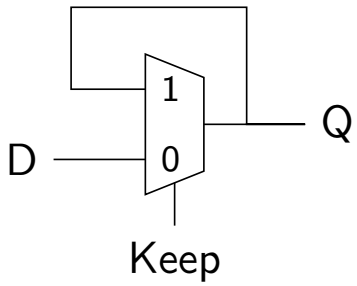
S	R	Q	$\overline{Q}$
0	1	0	1
1	1	forbidden	forbidden
1	0	1	0
0	0	Q_{n-1}	$\overline{Q_{n-1}}$

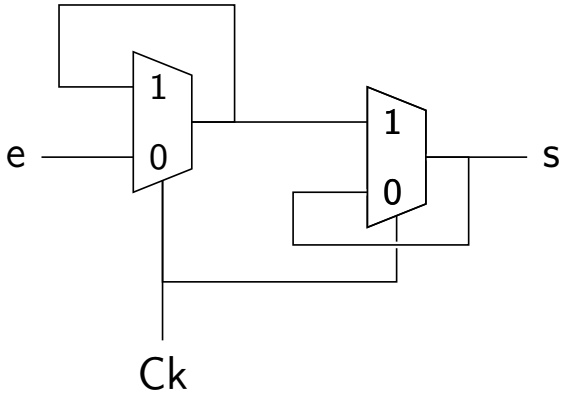
- Gated D latch (Flip-flop, register, *Bascule D*): sample input data on clock rising edge and keeps the value when clock is 0.



latches and Flip-Flops: other common representation

- Latch (*verrou*)





- Flip-Flop (register)

Sequential logic

Sequential logic combines logic function and memorizing, it opens the way to synchronous circuits, automata, programs, algorithms....

- n bits register
- n bits counter
- state machine
- CPU
- Computer

Sequential circuit design

- Extremely complex in general.
- Many computation models:
  - Sequential
    - State machine
    - control + data-path
  - task parallelism (communicating tasks)
  - Data parallelism (data-flow)
  - Asynchronous circuits
- Important notion use every where: finite state machine (*automate*)

Logic in ARC: Digital software

In ARC: use of Digital software (`https://github.com/hneemann/Digital`)

- Design basic logic components (TD1)
- Design of a memory (sequential component, TD2)
- Design of dedicated circuit: integer division (TD3).

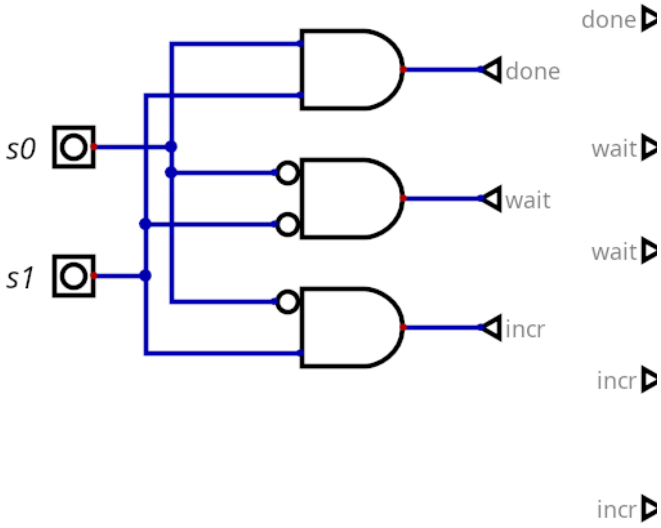
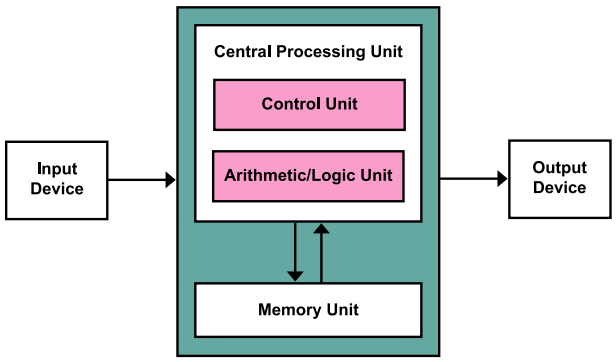


Table of Contents

What is a Von Neumann machine?



- Computer architecture Model (also called *Princeton* architecture) proposed after J. Von Neumann report: "First Draft of a Report on the EDVAC".
- Usually abstracted as a processor connected to a memory
- The memory is accessed (*randomly*) with an address (i.e. unlike a Turing machine)
- The memory contains both data and program (unlike a Harvard machine).

How does it work?

Compilation, Assembly code and binary code

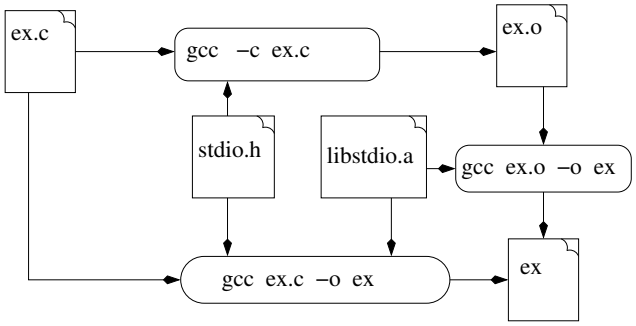
High Level Language ⇒	Assembly code ⇒	Binary code ⇒
`int a,b,c;`	`load R0, @b`	`01001011...10101`
`a = b + c;`	`load R1, @c`	`01001010...10001`
	`add R3,R0,R1`	`...`
	`store R3, @a`	`10010011...00011`

Fast compilation thanks to Donald Knuth (and others..)

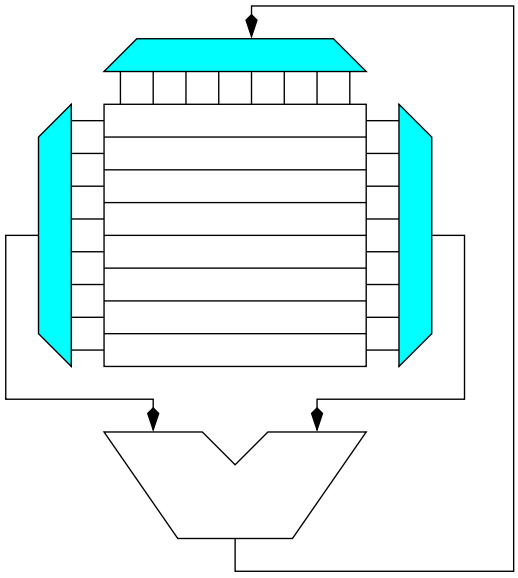
- The programmer:
  - Write a program (say a C program: `ex.c`)
  - Compiles it to an object program `ex.o`
  - links it to obtain an executable `ex`

content of `ex.c`

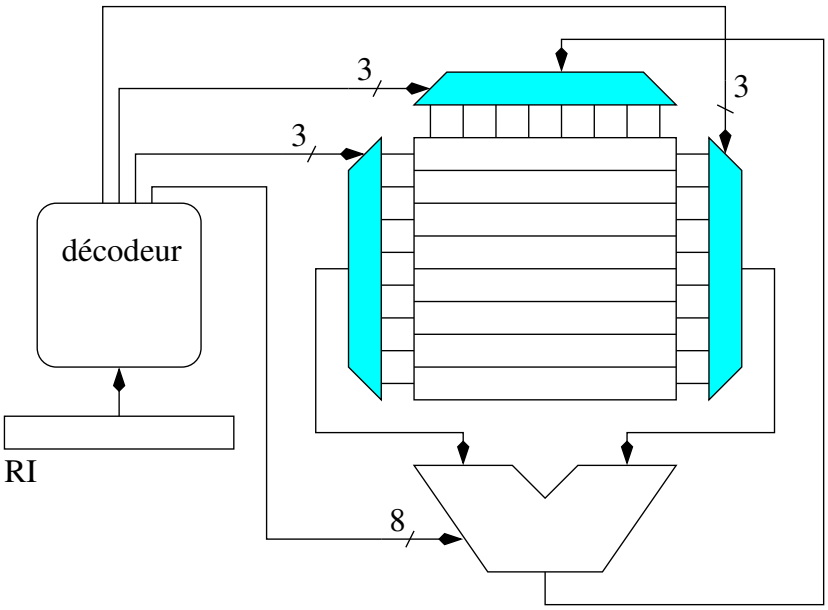
```
#include <stdio.h>

int main()
{
  printf("hello World\n");

  return(0);
}
```



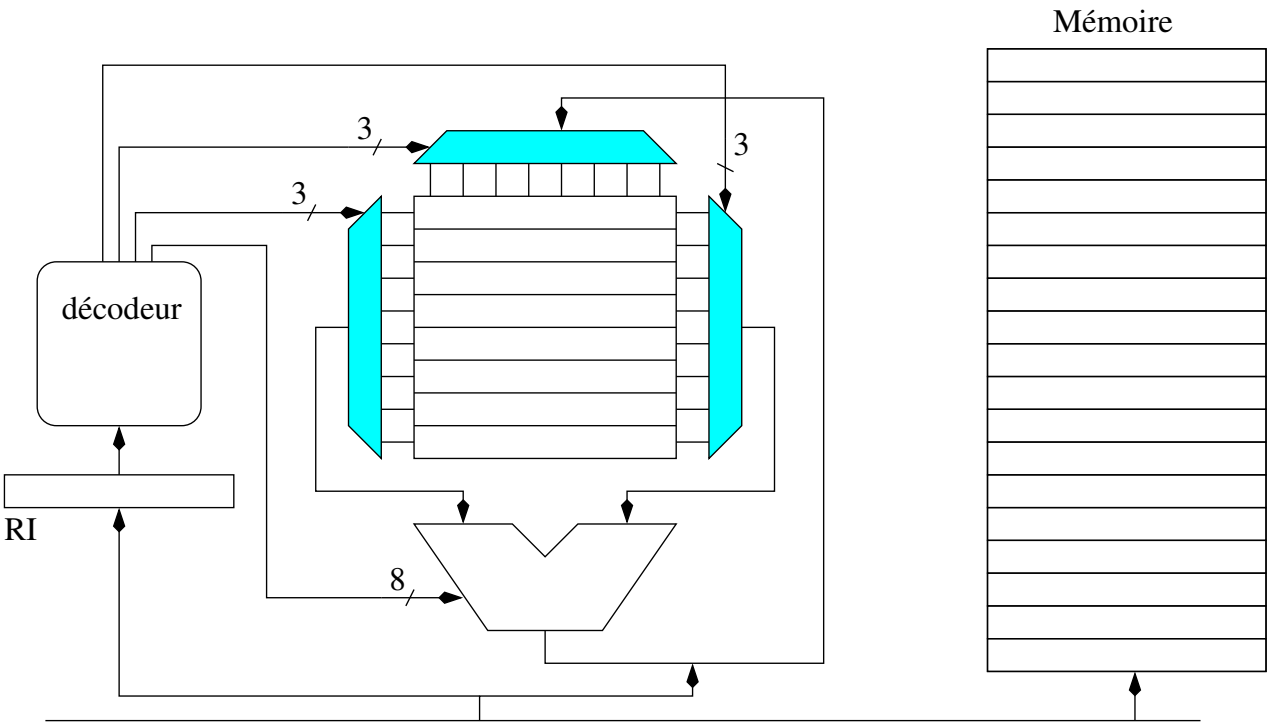
Program execution on a Processor (8 general purpose registers)



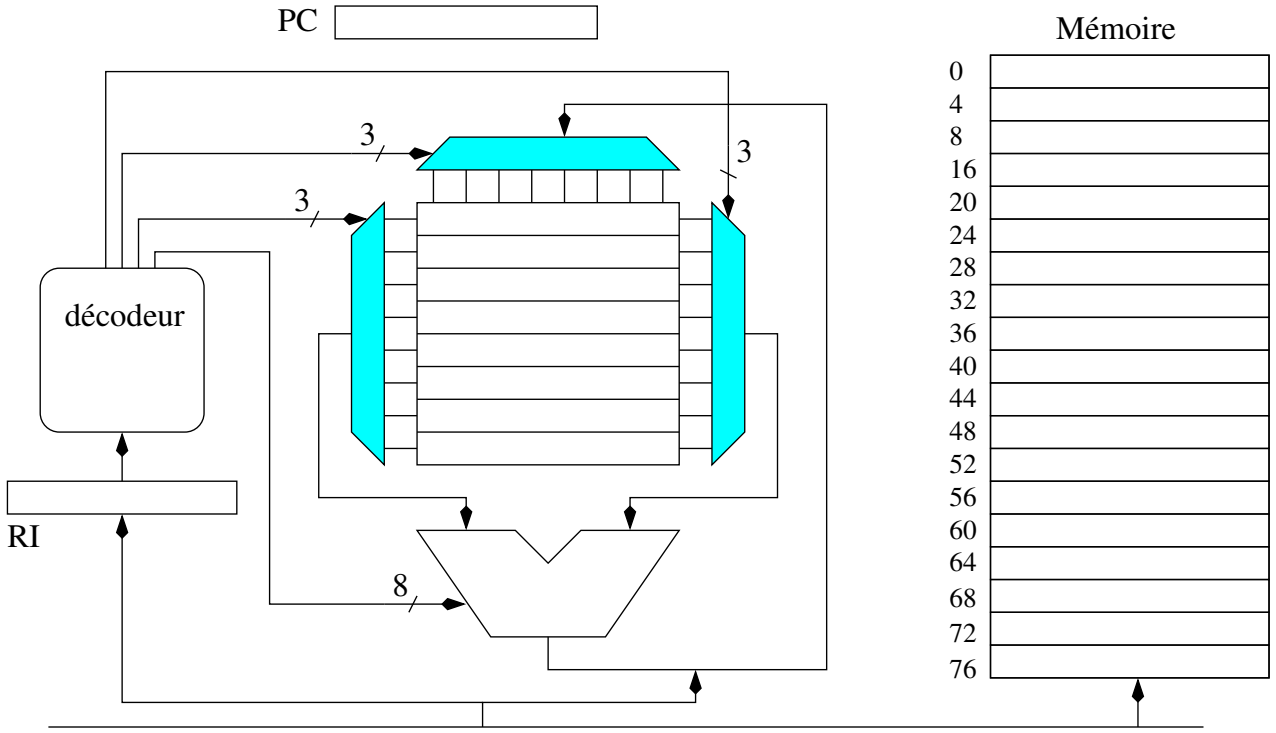
Program execution on a Processor (8 general purpose registers)



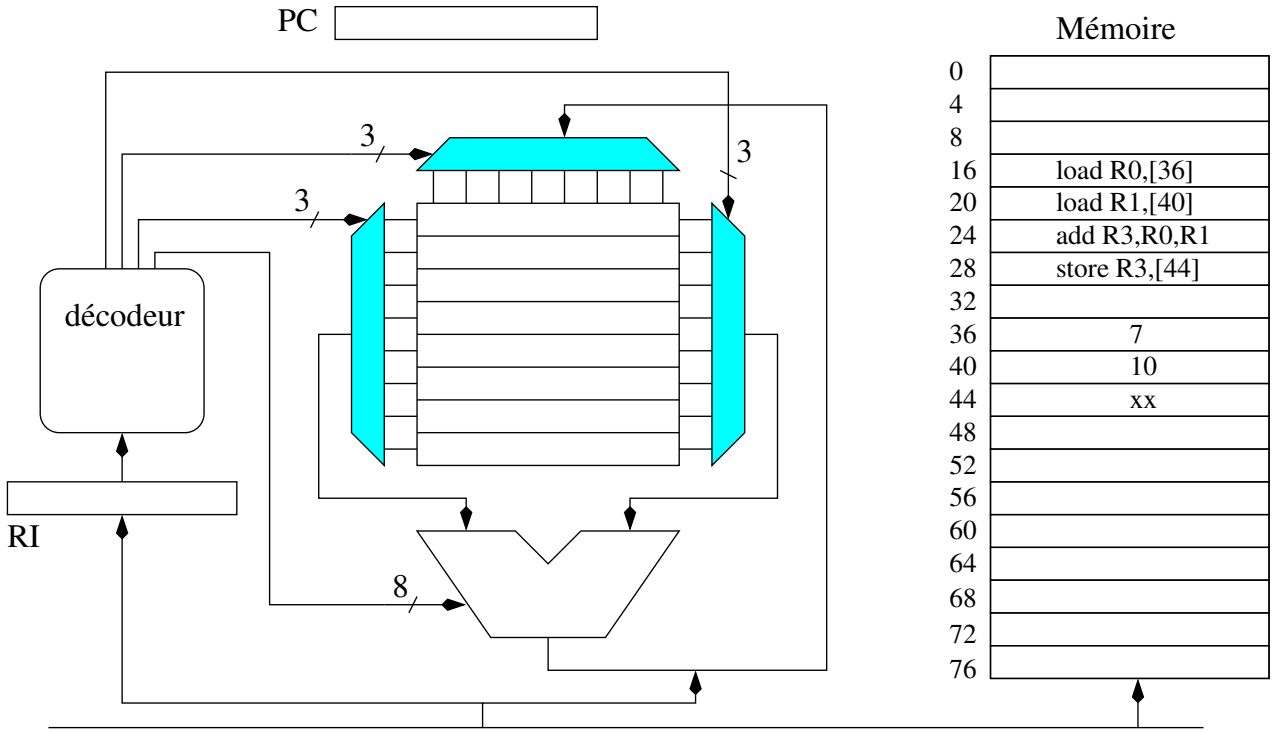
Program execution on a Processor (8 general purpose registers)



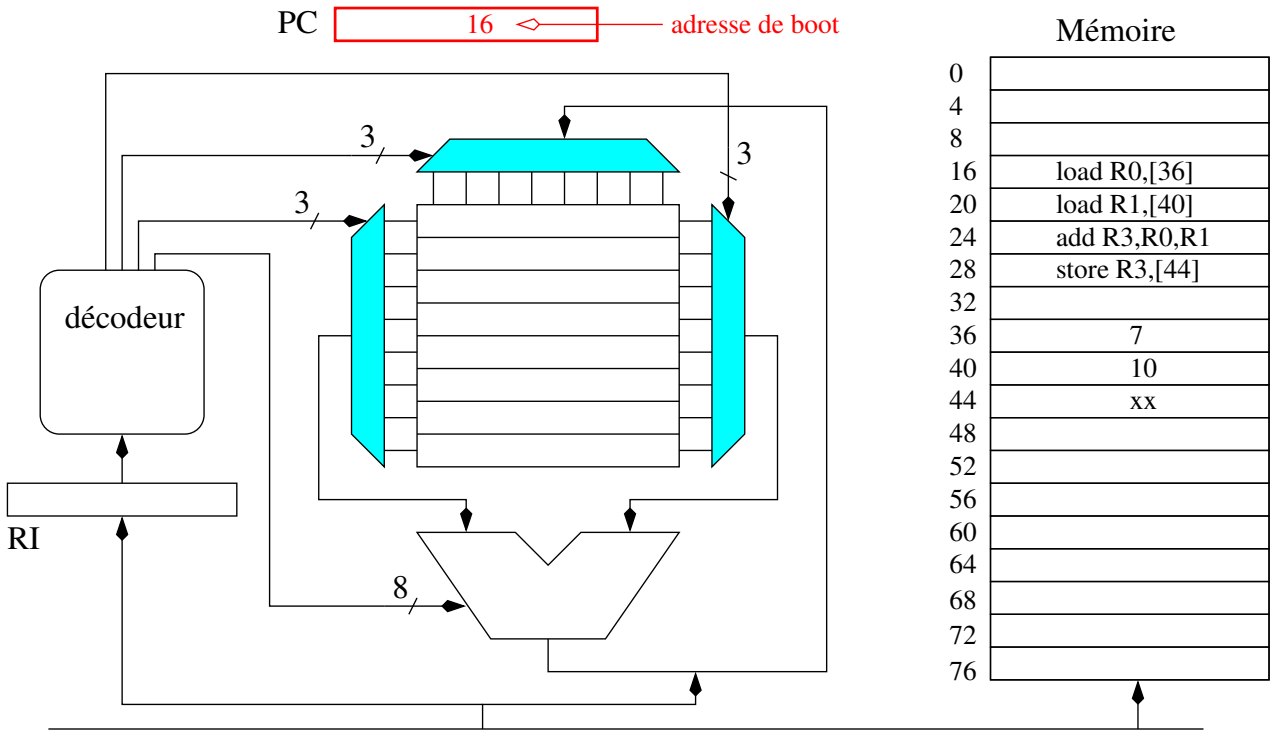
Program execution on a Processor (8 general purpose registers)



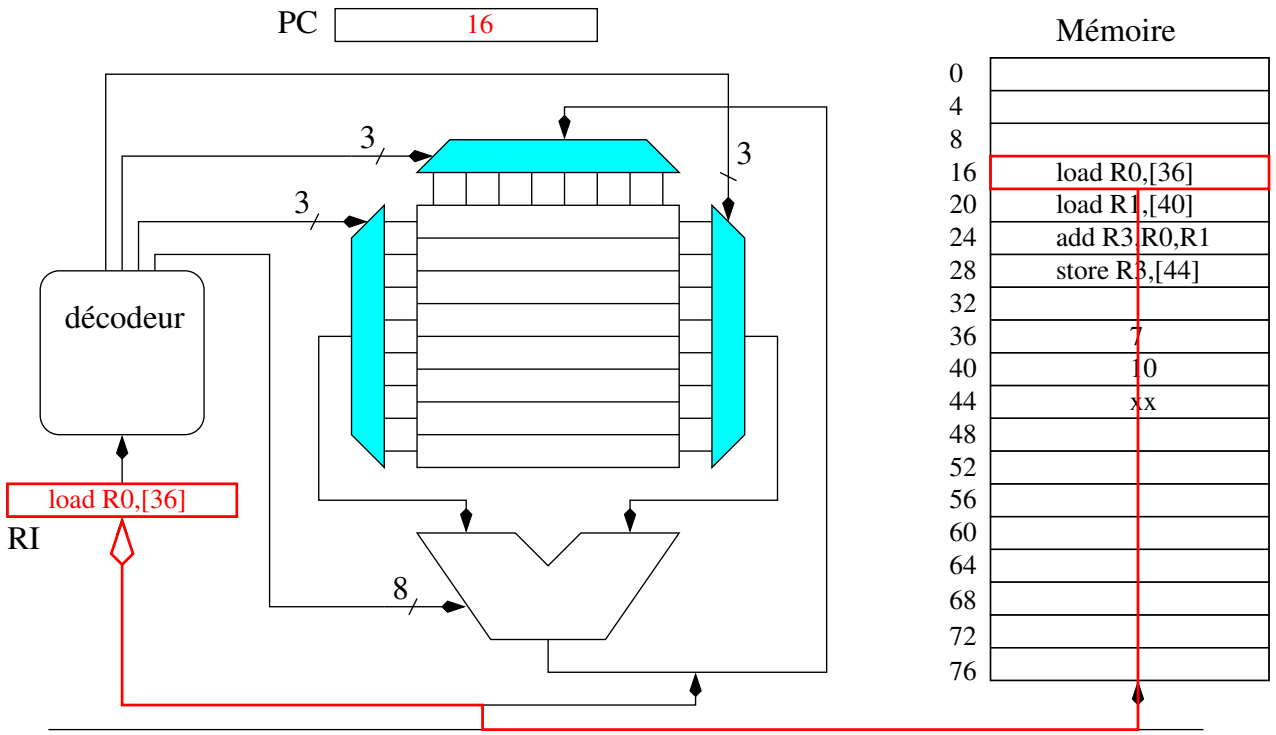
Program execution on a Processor (8 general purpose registers)



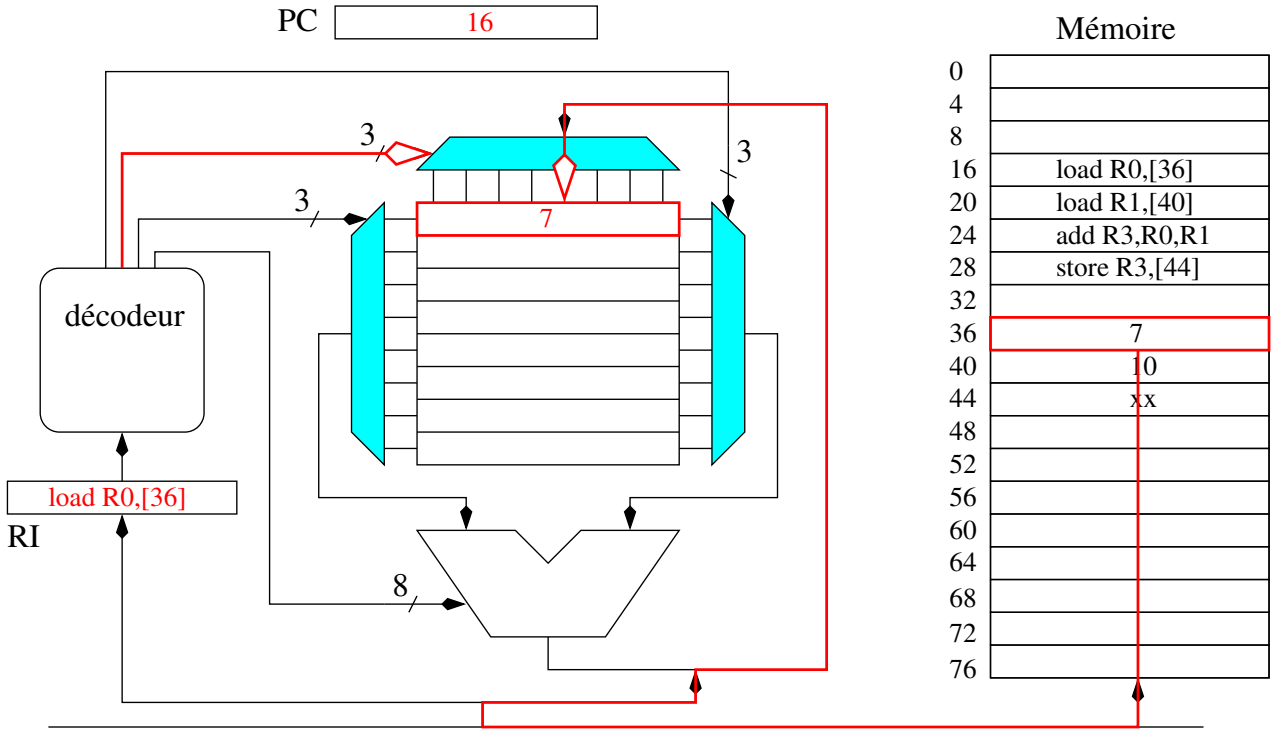
Program execution on a Processor (8 general purpose registers)



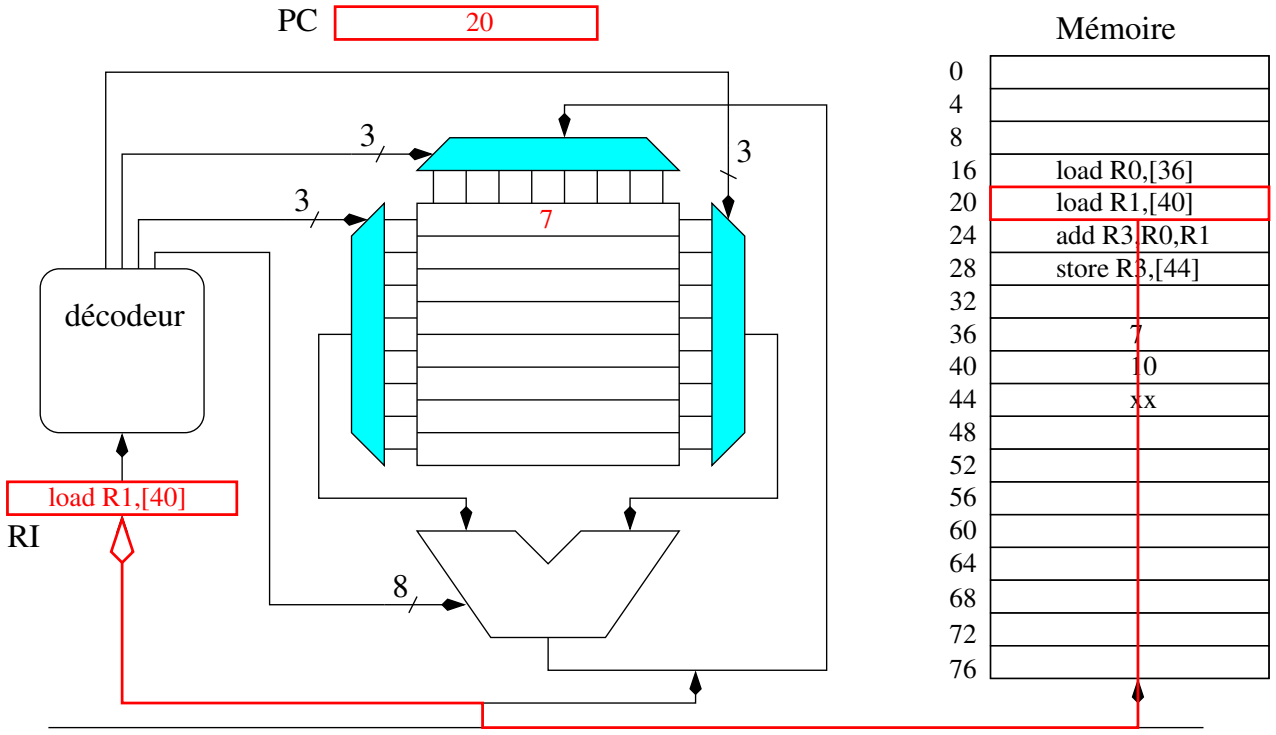
Program execution on a Processor (8 general purpose registers)



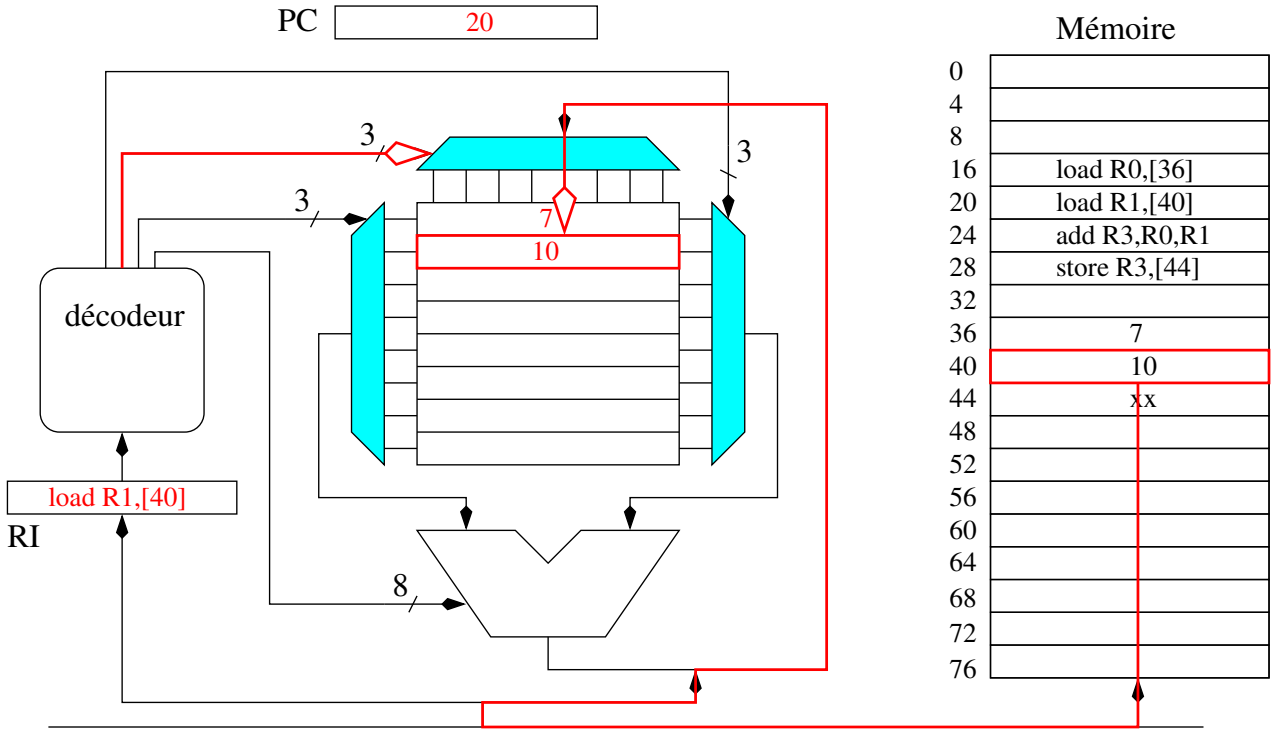
Program execution on a Processor (8 general purpose registers)



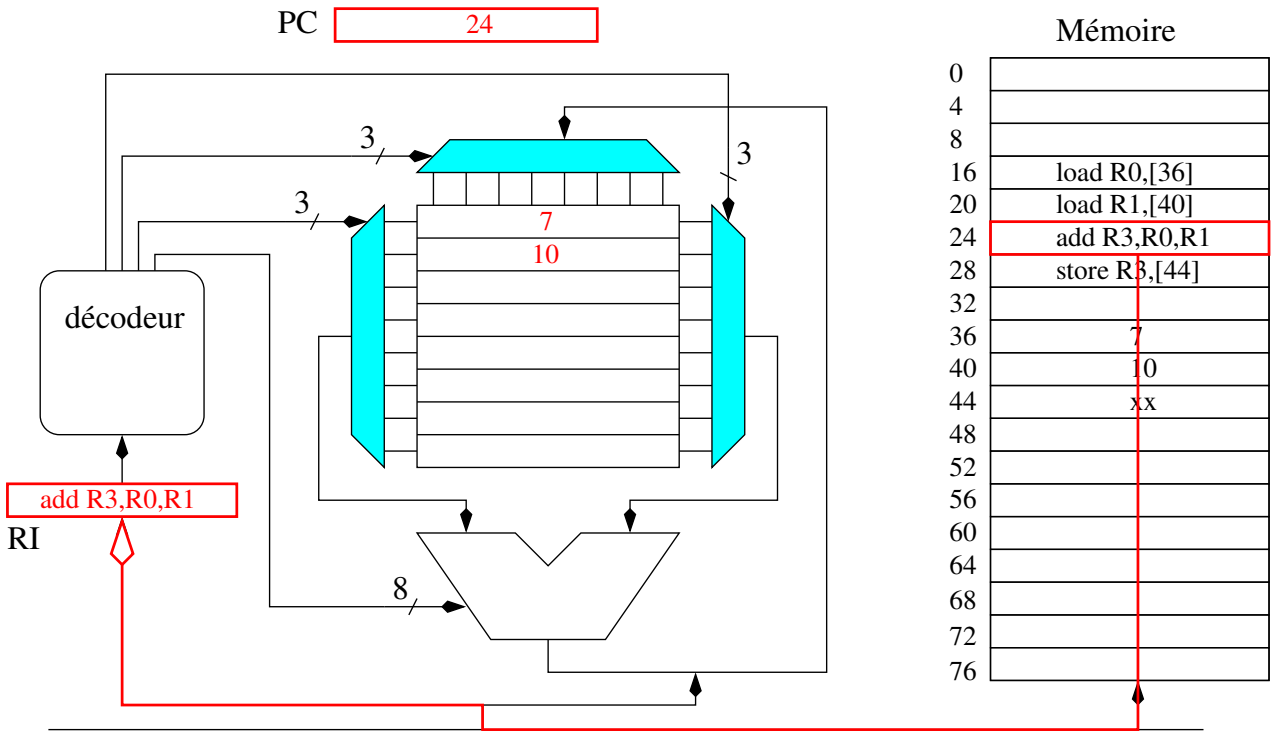
Program execution on a Processor (8 general purpose registers)



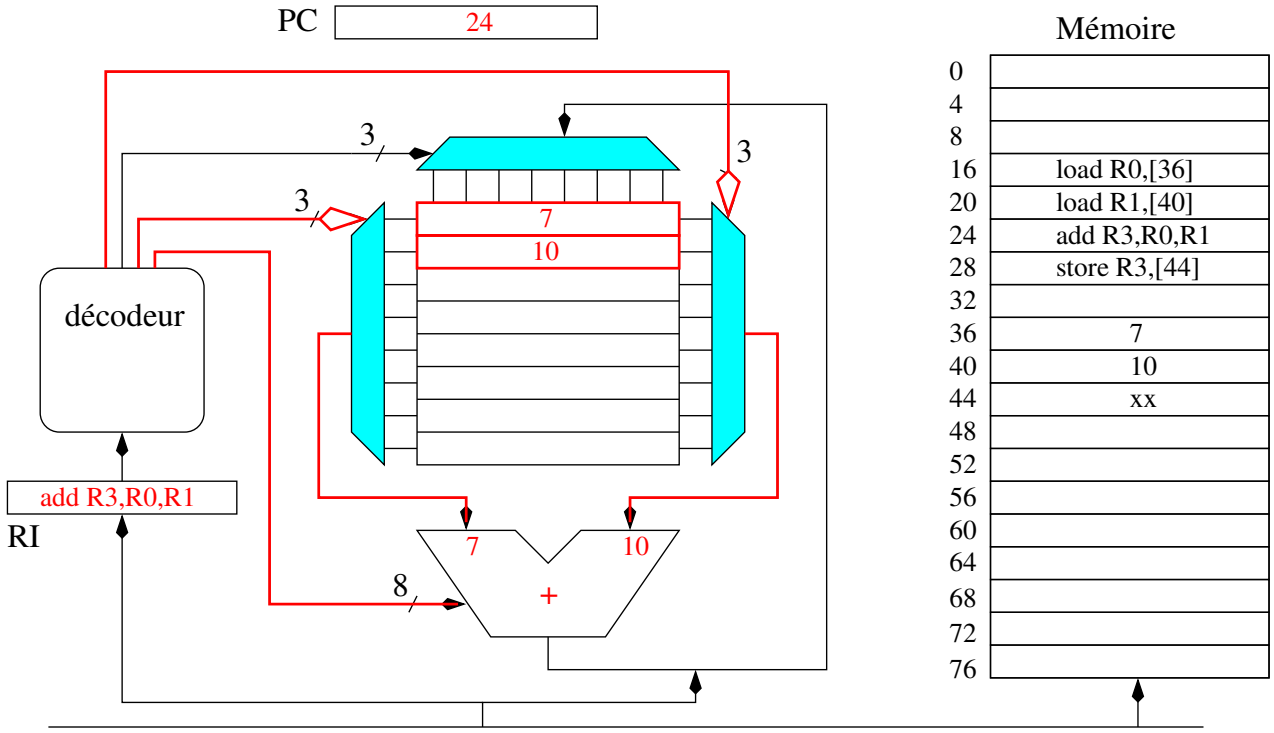
Program execution on a Processor (8 general purpose registers)



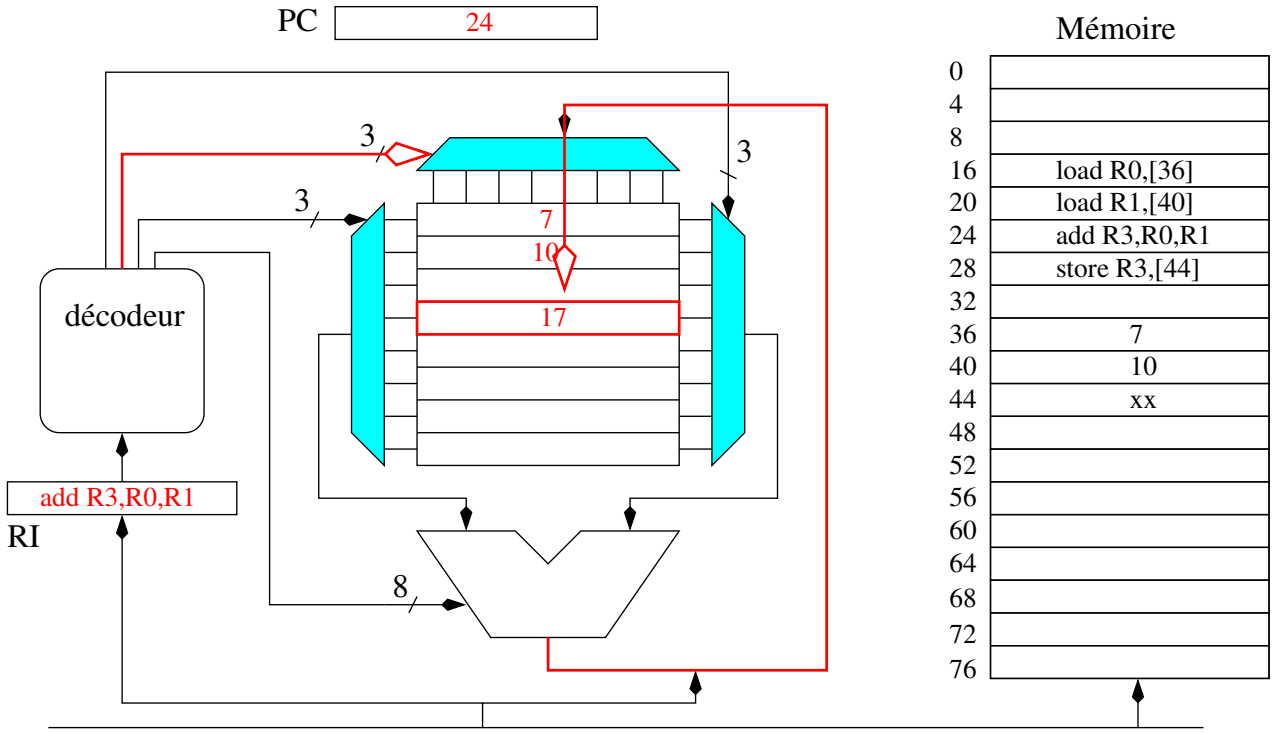
Program execution on a Processor (8 general purpose registers)



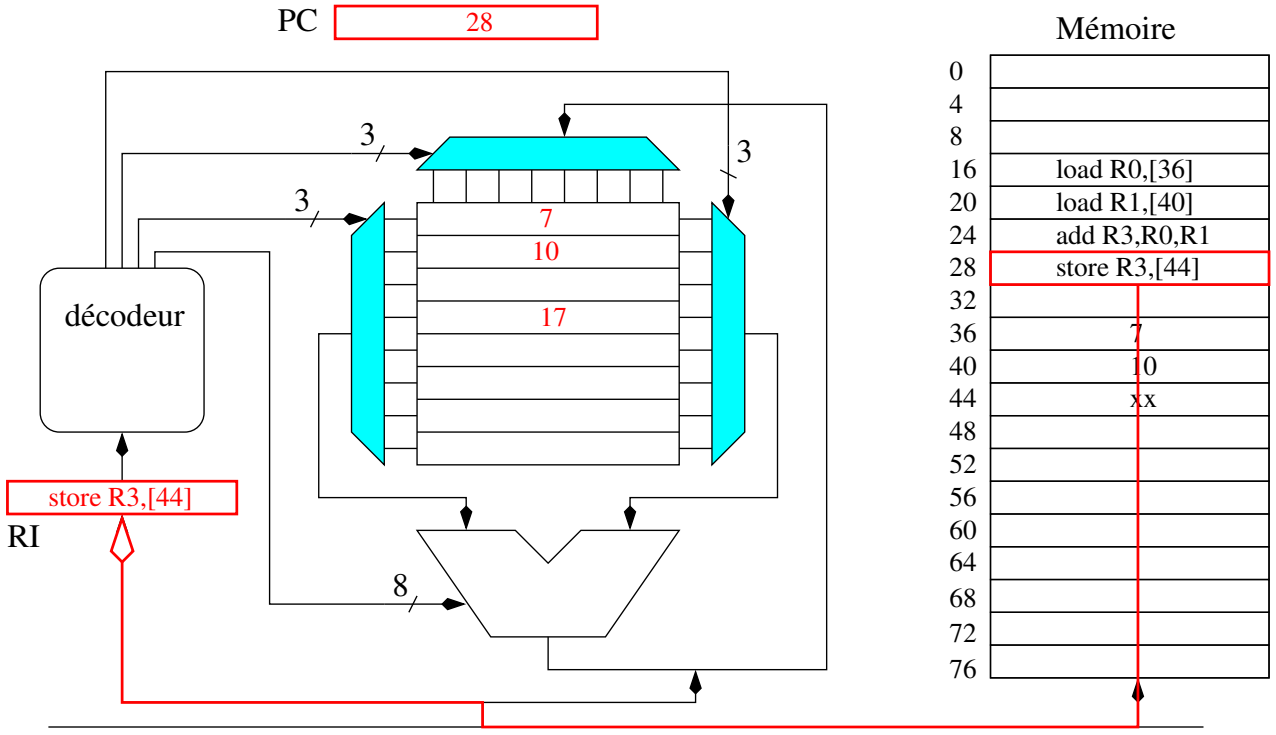
Program execution on a Processor (8 general purpose registers)



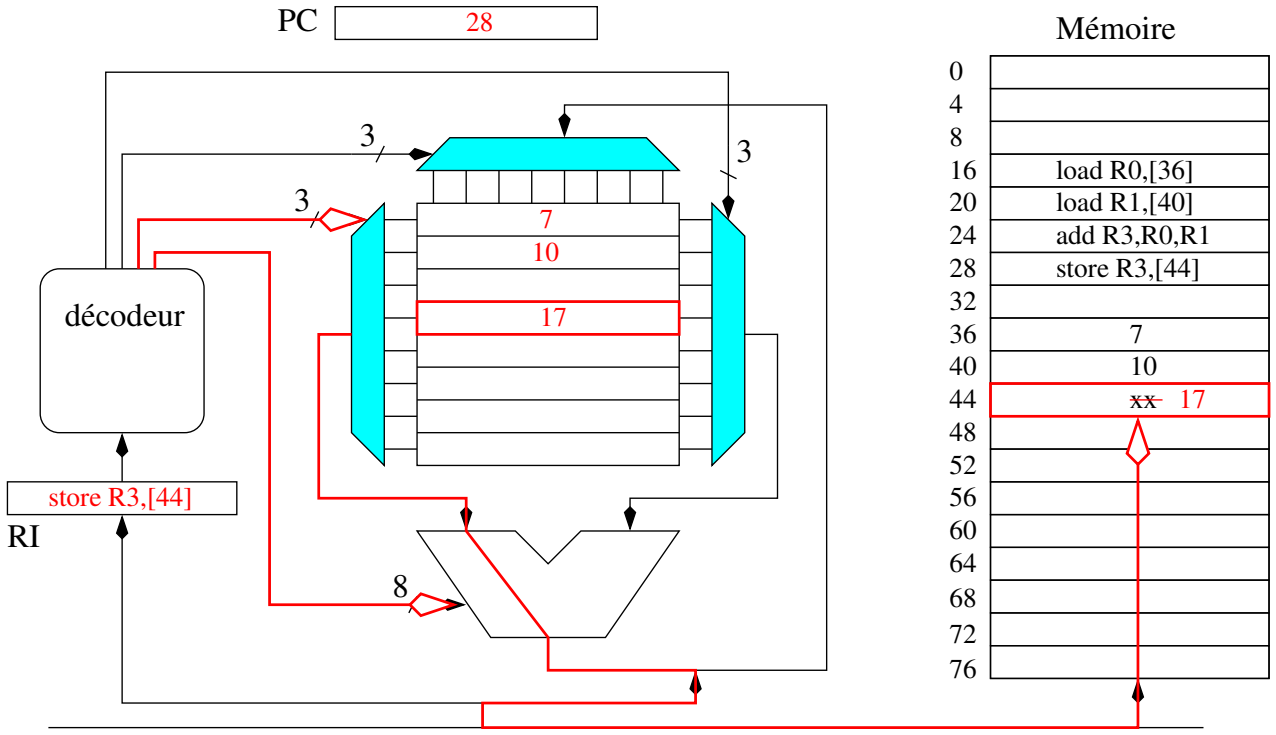
Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



Computer Architecture in ARC

- Design of a simple dedicated circuit in logisim
- Study of a simple processor in logisim
- Overview of assembly code principles
- Compilation basics
- embedded system case study

Add on: two's complement representation

- Two's complement (*complément à deux*) is the most common representation for negative integers
- For a number on N bits:
  - Positive integers from 0 to $2^{N-1}-1$ are represented with usual binary encoding
  - Negative integer x from -2^{N-1} to -1 are represented by coding in binary the positive number $2^N - |x|$
  - Hence Negative integers always have the last (i.e. most significant) bit at 1, and positive always have the last bit at 0
- Example with $N = 3$
  - Integers between -4_{10} and 3_{10} can be represented
  - -1_{10} is represented as 111_2 $(2^3 - 1 = 7)$
  - -2_{10} is represented as 110_2 $(2^3 - 2 = 6)$
  - -4_{10} is represented as 100_2 $(2^3 - 4 = 4)$

Add on: two's complement representation (2)

- Two's complement have an important property: Addition "classical" algorithm works (except that the overflow should be ignored).
- Example:
  - $-1_{10} + (-2_{10}) = 111_2 + 110_2 = 1101_2 =$(ignoring the carry/overflow)$101_2 = -3$
  - $-1_{10} + 2_{10} = 111_2 + 010_2 = 1001_2 =$(ignoring the carry/overflow)$001_2 = 1$
- For $x > 0$, $x \leq 2^{N-1}$, The representation of $-x$ on N bit two's complement can be obtained by:
  - Complementing each bits of x
  - adding 1 to the resulting integer
- Example:
  - with $N = 3$ and $x = 3_{10} = 011_2$, complement of x is 100_2 adding 1 gives $101_2 = -3_{10}$
  - With N=8 and $x = 96_{10} = 01100000_2$ complement of x is 10011111, adding one is $-96_{10} = 10100000_2$, indeed $256 - 96 = 160 = 10100000_2$

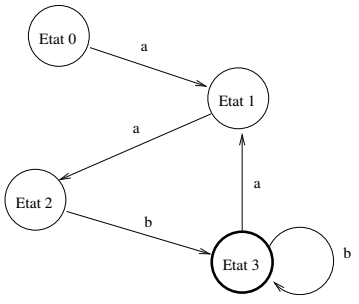
Table of Contents

Automata

- Definition (Wikipedia): An automaton is a self-operating machine, or a machine or control mechanism designed to automatically follow a predetermined sequence of operations, or respond to predetermined instructions.
- In computer science:
  - Used in language theory to build compilers
  - Used in any technical domain: to describe predetermined behaviour.
  - Used in computer architecture: to design dedicated circuit.
  - A computer is a specific automaton.

Notion d'automate

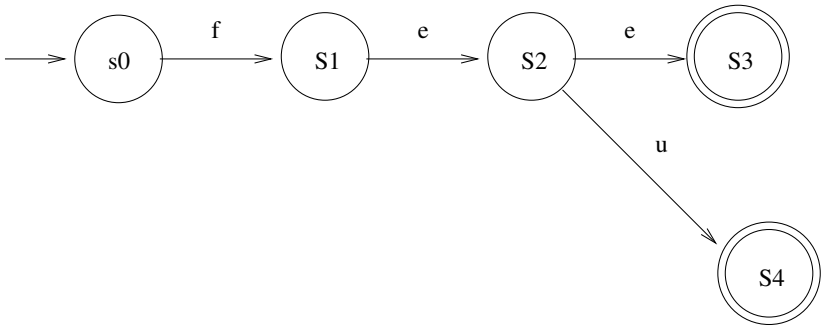
- Un automate est une collection de K états numérotés de 0 à K-1, ainsi qu'une collection de transitions
- Un état particulier est l'état initial.
- Tous les états sont soit des états d'acceptation et soit des états de refus
- Les transitions, sont étiquetées
  1. soit par des actions (par exemple, je lis la lettre `x`)
  2. soit par des condition (par exemple, la lettre `x` est présente)
- le triplets (état 1, lettre `x`, état 2) signifie: si je suis dans l'état 1 et que je lis la lettre `x`, alors je vais dans l'état 2.



Notion d'automate

- Fonctionnement d'un automate
  - Initialisation de l'automate dans l'état
  - il lit les lettres du mot une par une
    - s'il trouve une transition possible, il l'exécute,
    - sinon il répond «le mot n'appartient pas au langage»;
  - si l'automate arrive à effectuer des transitions jusqu'à la dernière lettre du mot, il regarde alors dans quel état il termine:
    - si l'état appartient à la classe d'acceptation, l'automate répond «le mot appartient au » (on dit que le mot est reconnu),
    - sinon, il répond «le mot n'appartiennent pas au langage».

Notion de mot reconnu



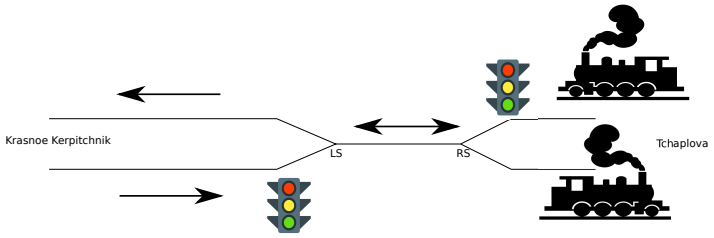
- `fee` → reconnu
- `feu` → reconnu
- `fei` → non reconnu (impossible de lire `'i'`)
- `fe` → non reconnu (arrêt dans un état non final)

Link with architecture: Computers are automata

- Every computing machine is an automata
- Computer are *universal* in the sense that the program gives much flexibility in the action performed.
- In fact the basic action of a computer is very repetitive:
  - Read the instruction at $PC in memory
  - decode the instruction
  - send the decoding to the ALU (or to memory if it is a load)
  - increment $PC
- Dedicated circuits (ASICs) are automata designed for specific tasks.

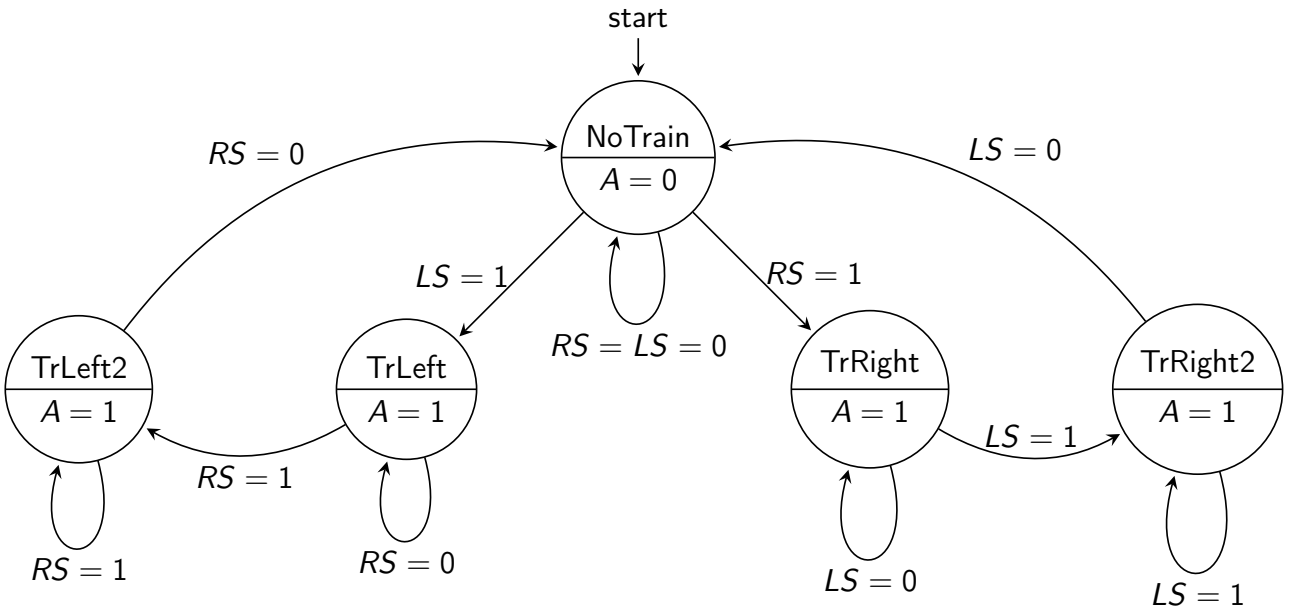
Table of Contents

Example from the poly

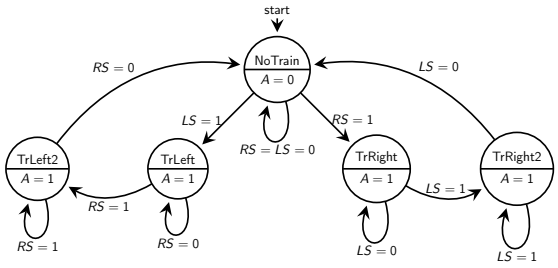


- A piece of unique train track for both train directions between the cities `T.` et `K.`
- Sensors triggered by train weight on rallways will command red lights when the track is used by a train.
- Modeling:
  - A booleen `A` (for 'Ampoule') indicating the state of the red light
  - Two booleans (LS for Left Sensor and RS for Rigth sensor) indicating the states of the sensors
  - An automaton to command the red lights

The Russian train automaton

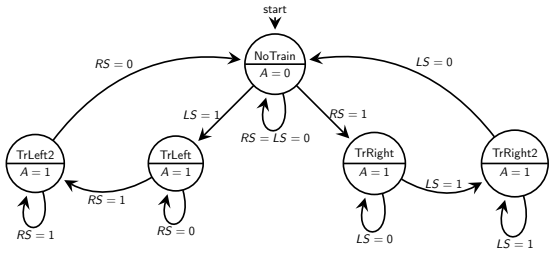


The Russian train automaton



- Circles are *states* of the automaton (e.g. `NoTrain` state models the cases where no train stand on the track).
- States specifies output Values (here only one: `A`)
- Arrows are *transitions*, labeled by event that triggered them.

Back to the Russian train example

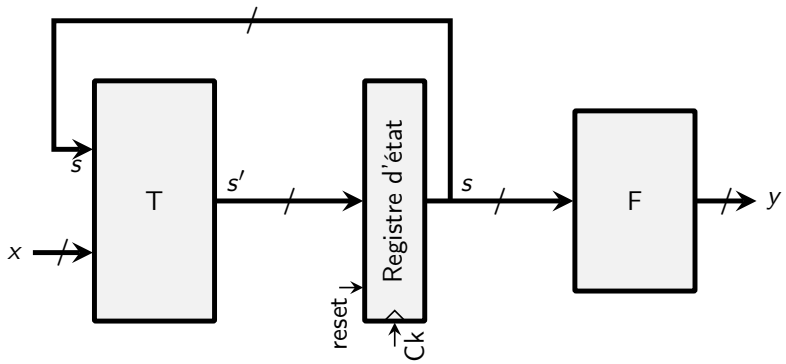


- The Inputs are `RS` and `LS` sensors Boolean values
- The Output is the value of Boolean `A`
- The functions (Transition and Output) can be defined by tables ⇒
- `X` means 'don't care'

s	x=(LS, RS)	s'=T(s,x)
NoTrain	00	NoTrain
NoTrain	01	TrRight
NoTrain	10	TrLeft
NoTrain	11	XXX
TrRight	0X	TrRight
TrRight	1X	TrRight2
TrRight2	1X	TrRight2
TrRight2	0X	NoTrain

s	y=F(s)
NoTrain	0
TrRight	1
TrRight2	1

Implementation of a synchronous automaton as a circuit

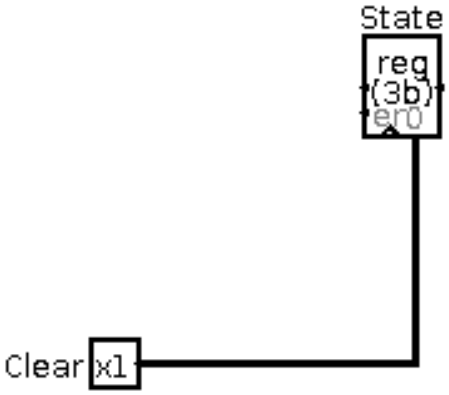


- `s` is current state, `s'` is next state, `x` are input bits, `y` are output bits.
- `Ck` and `reset` are not considered as inputs
- State change will occur on each rising edge of the Clock.

Implementation in Logisim

- We need to store 5 States, hence we need at least 3 bits:

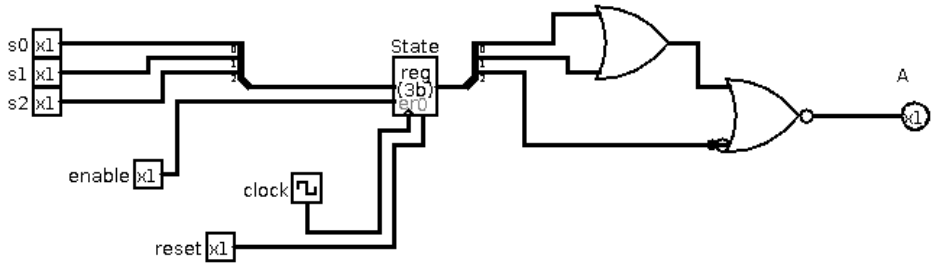
value (binary)	state
100	NoTrain
000	TrRight1
001	TrRight2
010	TrLeft
011	TrLeft2



Russian train output function

- The output function is easy: A is on iff state is ''NoTrain'

s	y=F(s)
NoTrain	0
TrRight	1
TrRight2	1



Russian train Transition function: more complicater

s	x=(LS, RS)	s'=T(s,x)
100 (NoTrain)	00	NoTrain
100 (NoTrain)	01	TrRight
100 (NoTrain)	10	TrLeft
100 (NoTrain)	11	XXX
000 (TrRight)	0X	TrRight
000 (TrRight)	1X	TrRight2
001 (TrRight2)	1X	TrRight2
001 (TrRight2)	0X	NoTrain
010 (TrLeft)	X0	TrLeft
010 (TrLeft)	X1	TrLeft2
011 (TrLeft2)	X1	TrLeft2
011 (TrLeft2)	X0	NoTrain

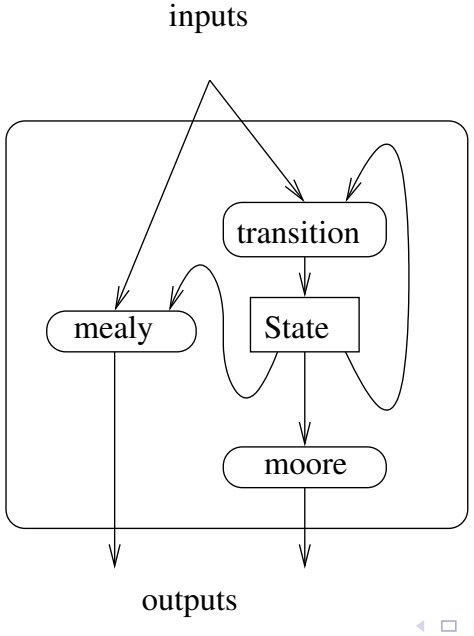
Table of Contents

Comming back to automata

- Automata are very widely used in computer science in different domains.
- In ARC we use them to *control the execution of dedicated synchronous circuits*
- As soon as a dedicated circuit is designed, there is an automaton designed.

Mealy and Moore automata

- We have seen a *Moore automaton*: output only depend on the state (not on the input), usually simpler to handle.
- The most general form of an automaton has a moore output and a mealy output



Summery: from Algorithm to Circuit

- From algorithm to automata (states and input/output)
- From automata to synchronous automata
- From synchronous automata to digital design

Lab topic: circuit for integer division

```
n := entrée N
p := entrée P
x := 0
q := 0
tant que x+p ⩽ n
    x := x+p
    q := q+1
fin tant que
sortie Q := q
```


Lab topic: proposed circuit to realize it