ARC: Computer Architecture

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

Tanguy Risset

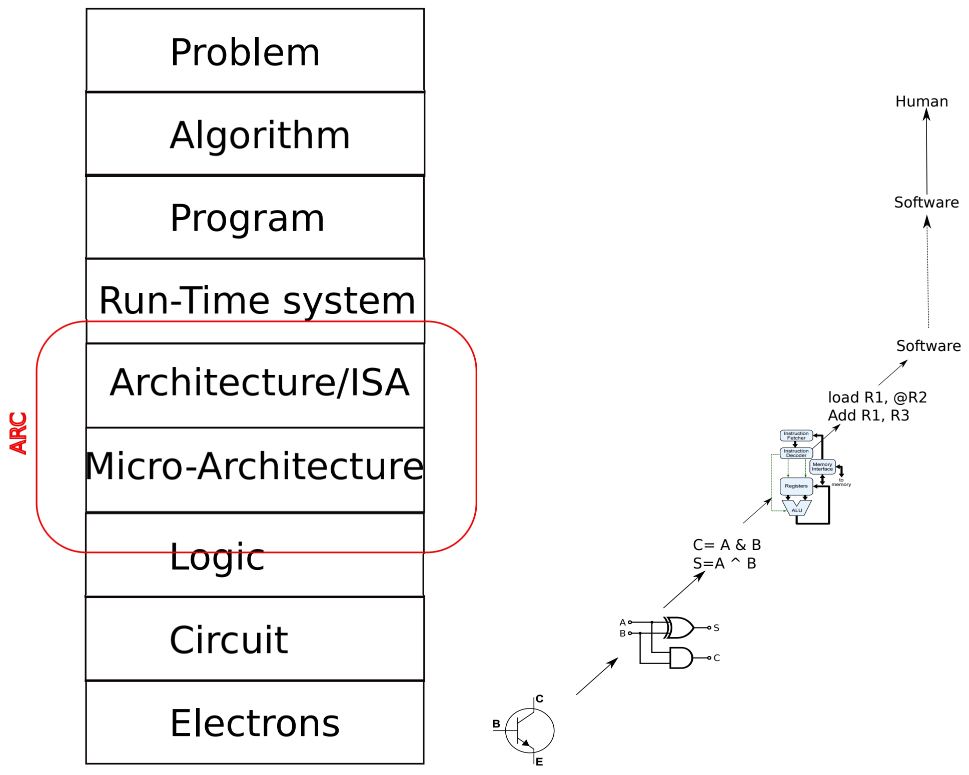
April 29, 2024

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

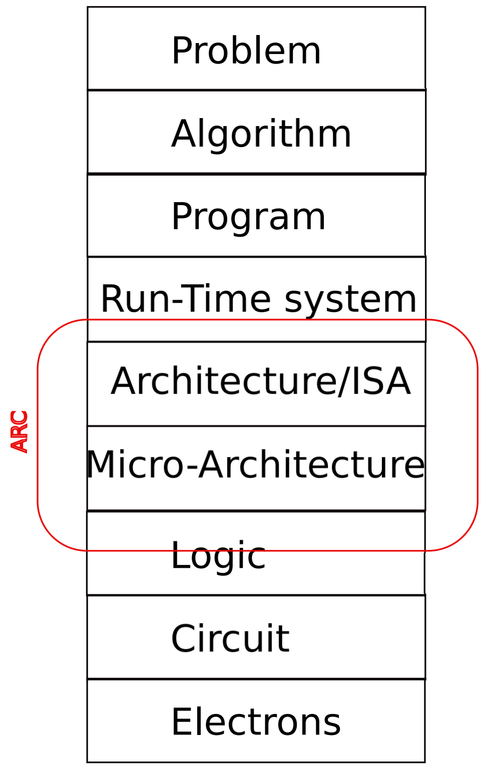
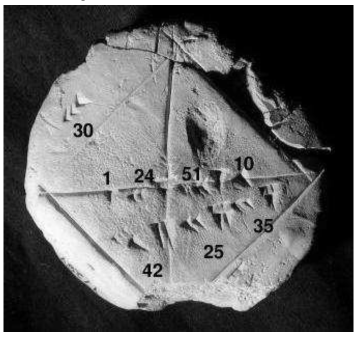


Table of Contents

History of computing

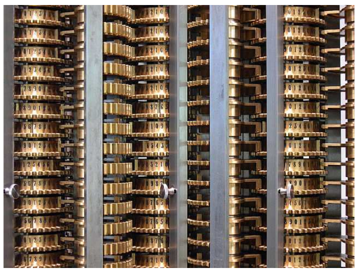
- Ancient time: various arithmetics systems
- 17th century (Pascal and Leibniz): notion of mechanical calculator
- 1822 Charles Babbage Difference engine (tabulate polynomial functions)
- 1854 Georges Boole proposes the so-called Boolean logic.
- (More details on the poly or on Internet)

from Yale Babylonian Collection, ≃ 1600 BC



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

Difference Machine close-up

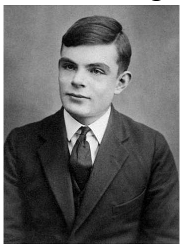


By By Carsten Ullrich - Own work, CC BY-SA 2.5

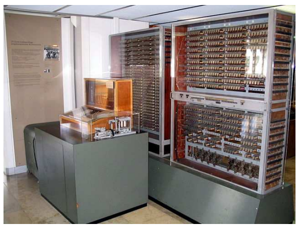
History of computers

- 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

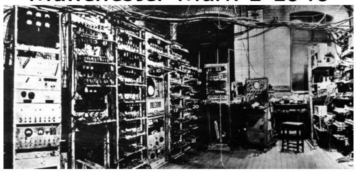
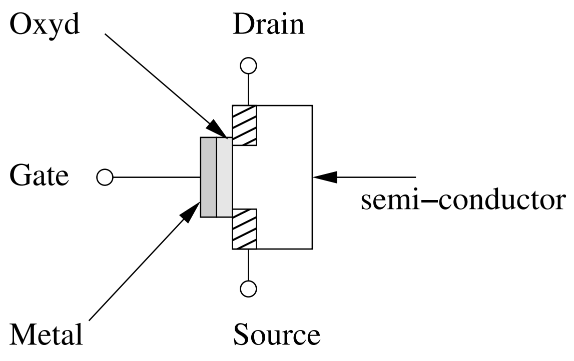


Table of Contents

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

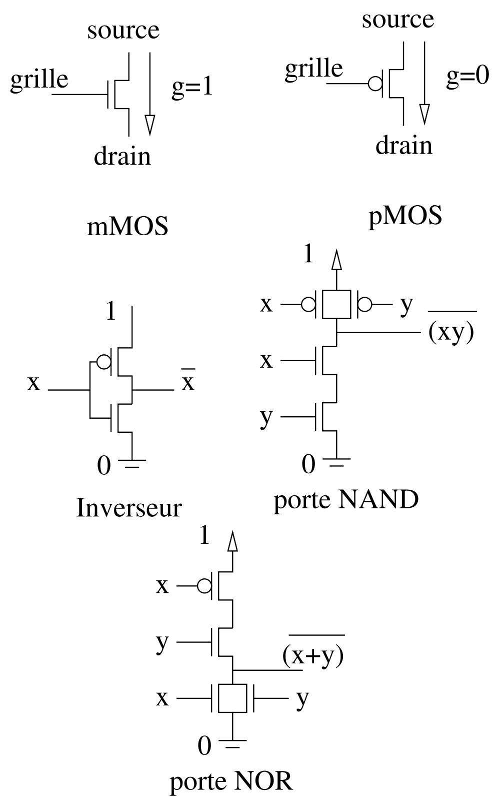
- Can be seen as a switch
- Wildly used after Integrated Circuit invention (1958)



Mosfet technology

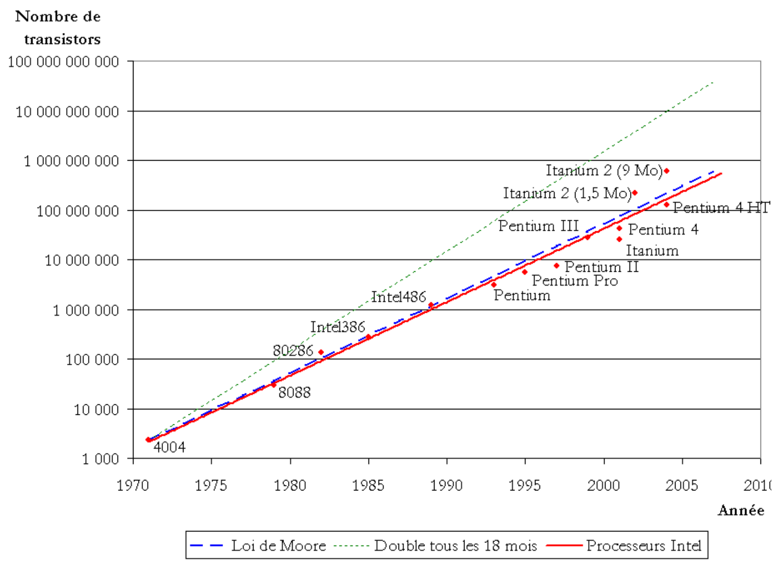
Popular Transistor technology: CMOS

- 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`

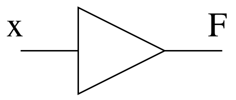
If a and b are 2 boolean variables, we write:

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

Boolean Cheat Sheet

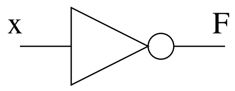
- neutral elements: $a + 0 = a, \quad a{\cdot}1 = a$
- absorbing elements: $a + 1 = 1, \quad a{\cdot}0 = 0$
- idempotence: $a + a = a, \quad a{\cdot}a = a$
- tautology/antilogy: $a + \bar{a} = 1, \quad a{\cdot}\bar{a} = 0$
- commutativity: $a + b = b + a, \quad ab = ba$
- distributivity: $a + (bc) =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

Elementary logical gates



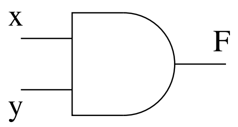
Amplifier:
$F = x$

x	F
0	0
1	1



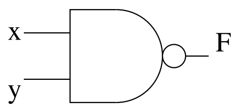
NOT: $F = \overline{x}$

x	F
0	1
1	0



AND: $F = x\ y$

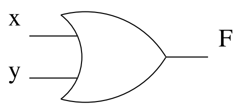
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1



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

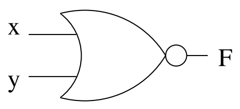
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

Elementary logical gates



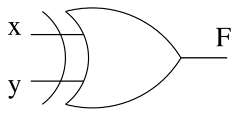
OR:
$F = x + y$

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



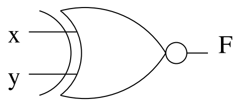
NOR:
$F = \overline{(x + y)}$

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



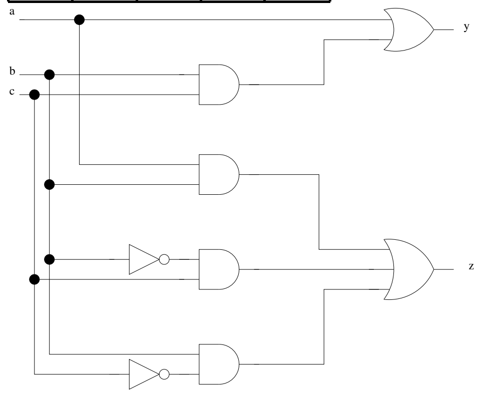
XNOR:
$F = x \odot y$

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:
  - $x.\overline{y}.\overline{z} + \overline{t}.u.v$
  - $(a \wedge b) \vee \neg c$
- 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: $z = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}c + ab\bar{c} + abc$
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 $z = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}c + ab\bar{c} + abc$ 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.

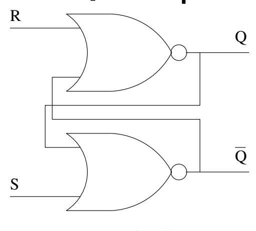
- No cycles, no ouptuts conected together

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

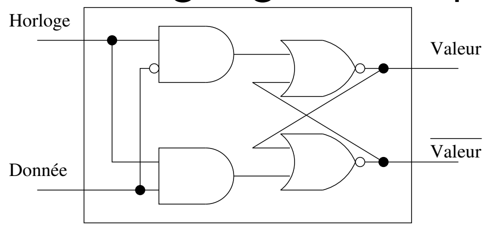
- Set-Reset Latch (SR latch, *Bascule RS*): When R and S are reset, Q and $\overline{Q}$ keep their previous value.



Bascule RS

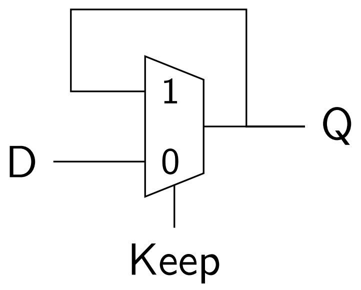
S	R	Q	$\overline{Q}$
0	1	0	1
1	1	forbidden	forbidden
1	0	1	0
0	0	Q_{n-1}	$\overline{Q_{n-1}}$

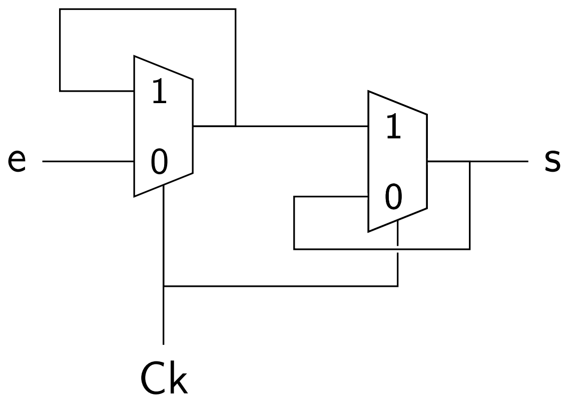
- Gated D latch (Flip-flop, register, *Bascule D*): sample input data on clock rising edge and keeps the value when clock is 0.



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).

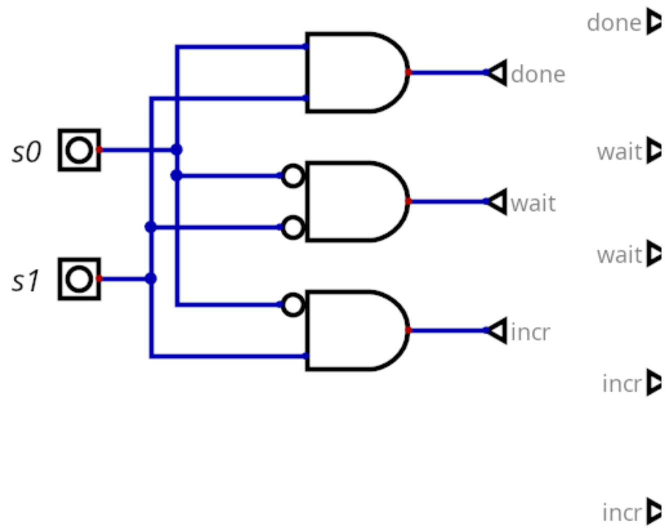
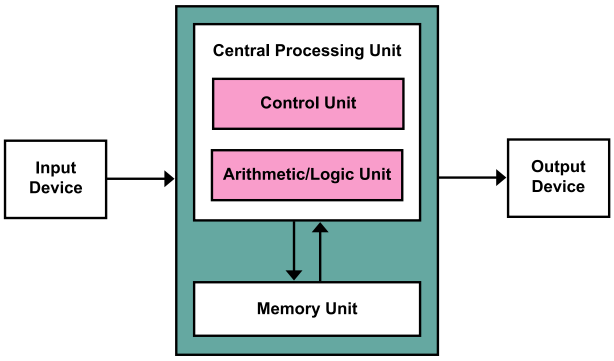


Table of Contents

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

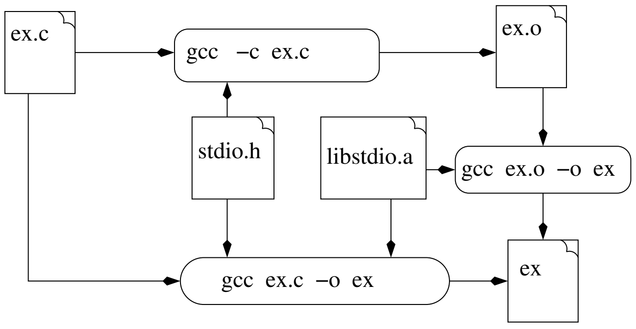
High Level Language ⇒	Assembly code ⇒	Binary code ⇒
`int a,b,c;`	`load R0, @b`	`01001011...10101`
`a = b + c;`	`load R1, @c`	`01001010...10001`
	`add R3,R0,R1`	`...`
	`store R3, @a`	`10010011...00011`

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

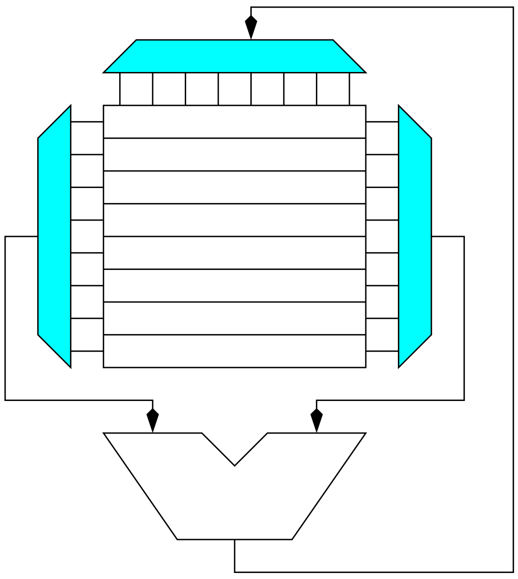
- The programmer:
  - Write a program (say a C program: `ex.c`)
  - Compiles it to an object program `ex.o`
  - links it to obtain an executable `ex`

content of `ex.c`

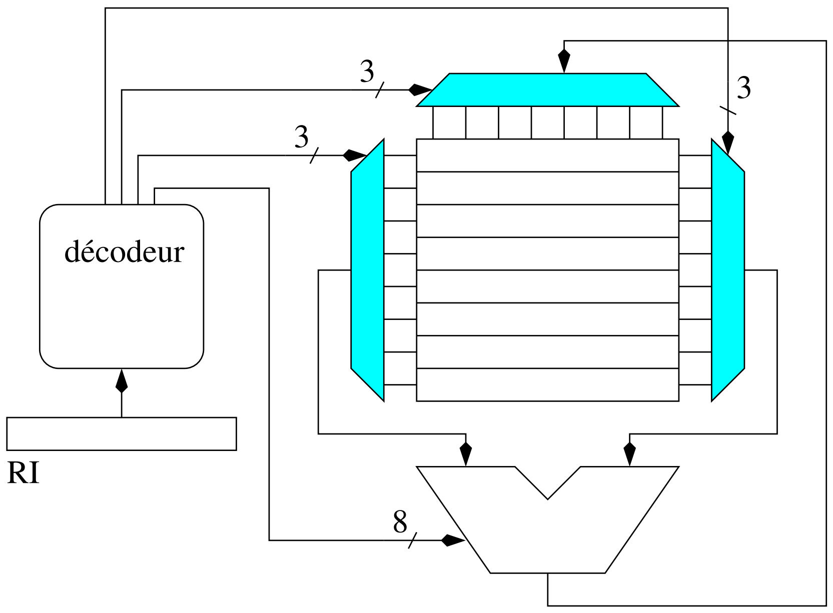
```
#include <stdio.h>

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

  return(0);
}
```



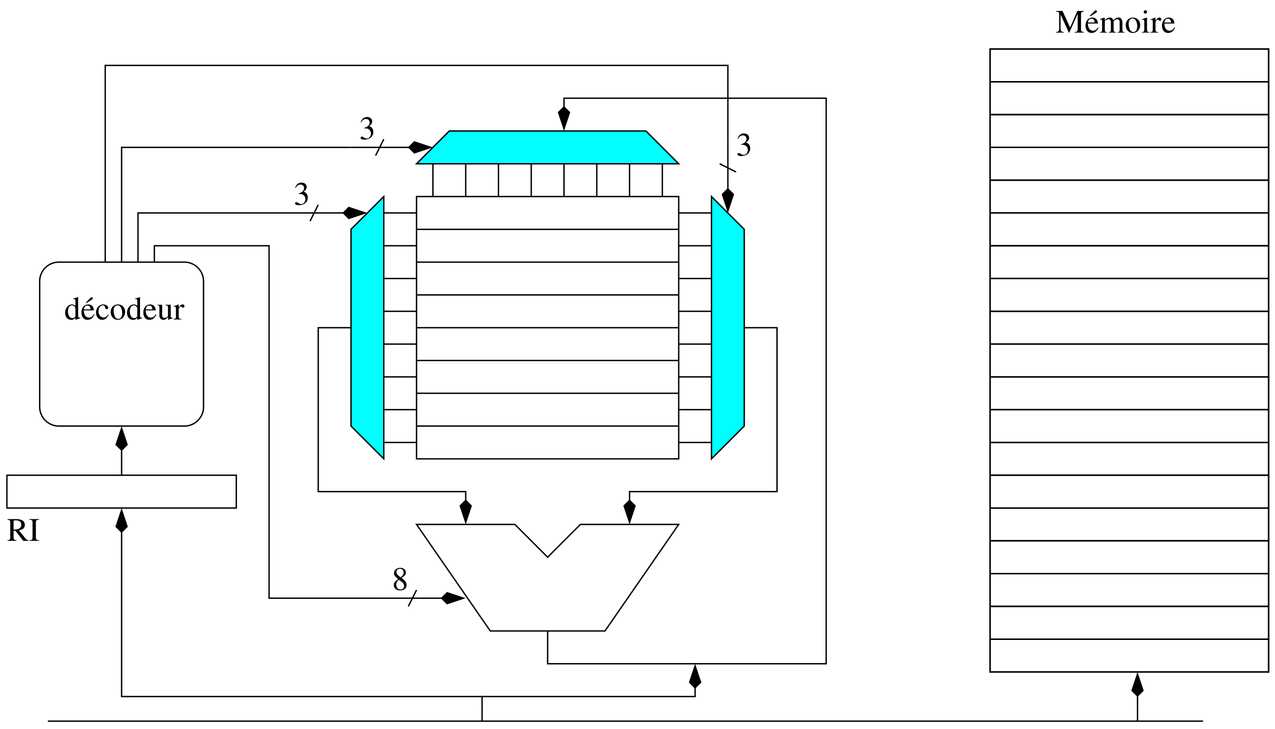
Program execution on a Processor (8 general purpose registers)



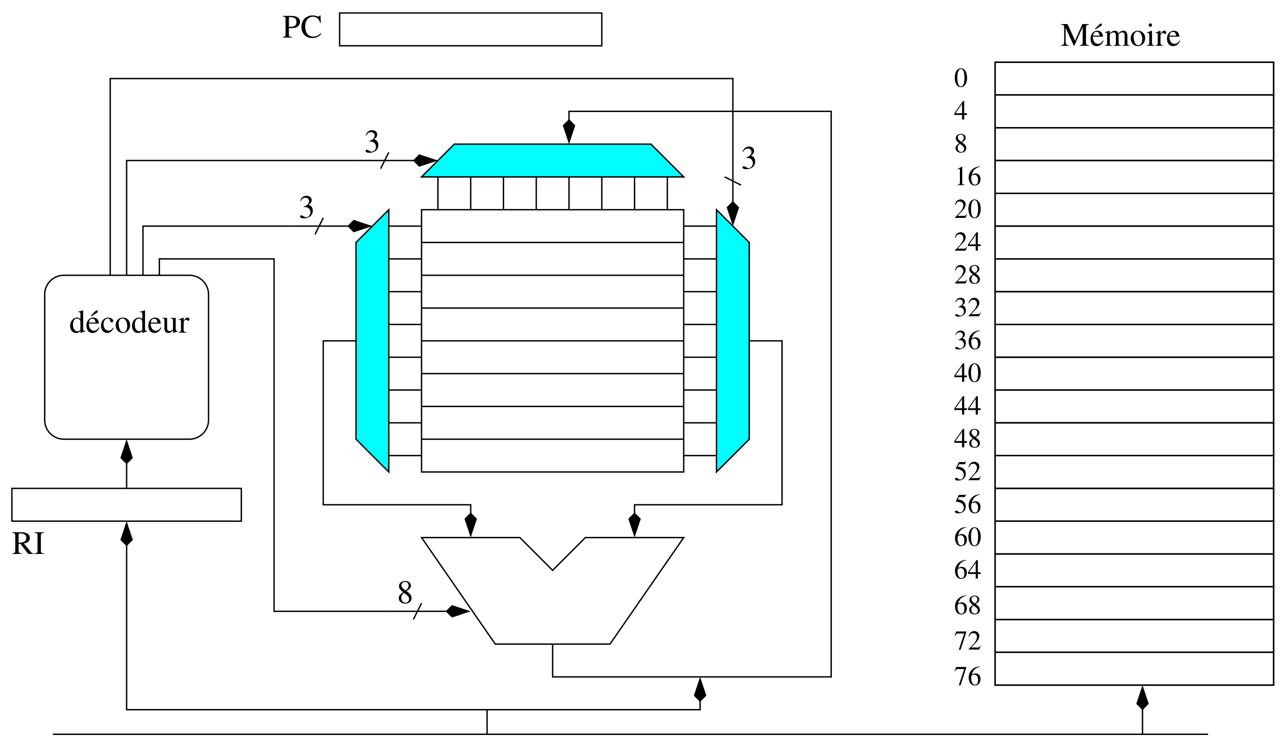
Program execution on a Processor (8 general purpose registers)



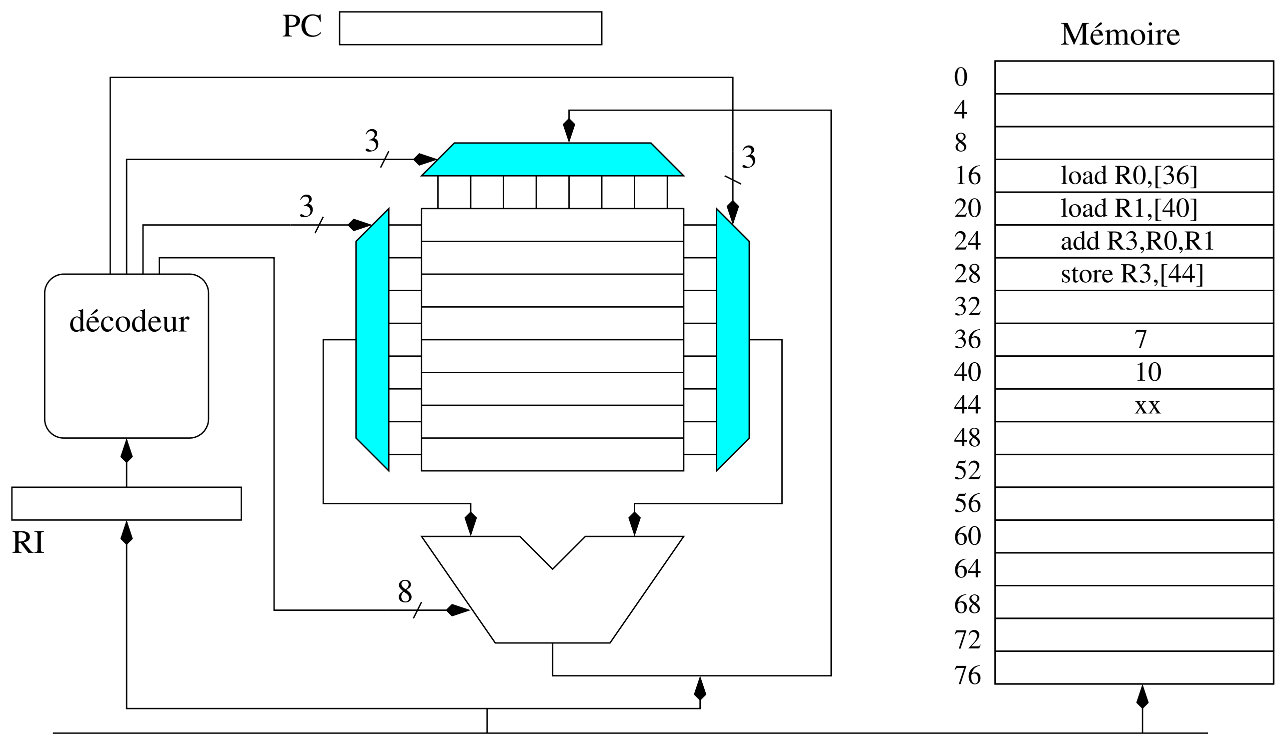
Program execution on a Processor (8 general purpose registers)



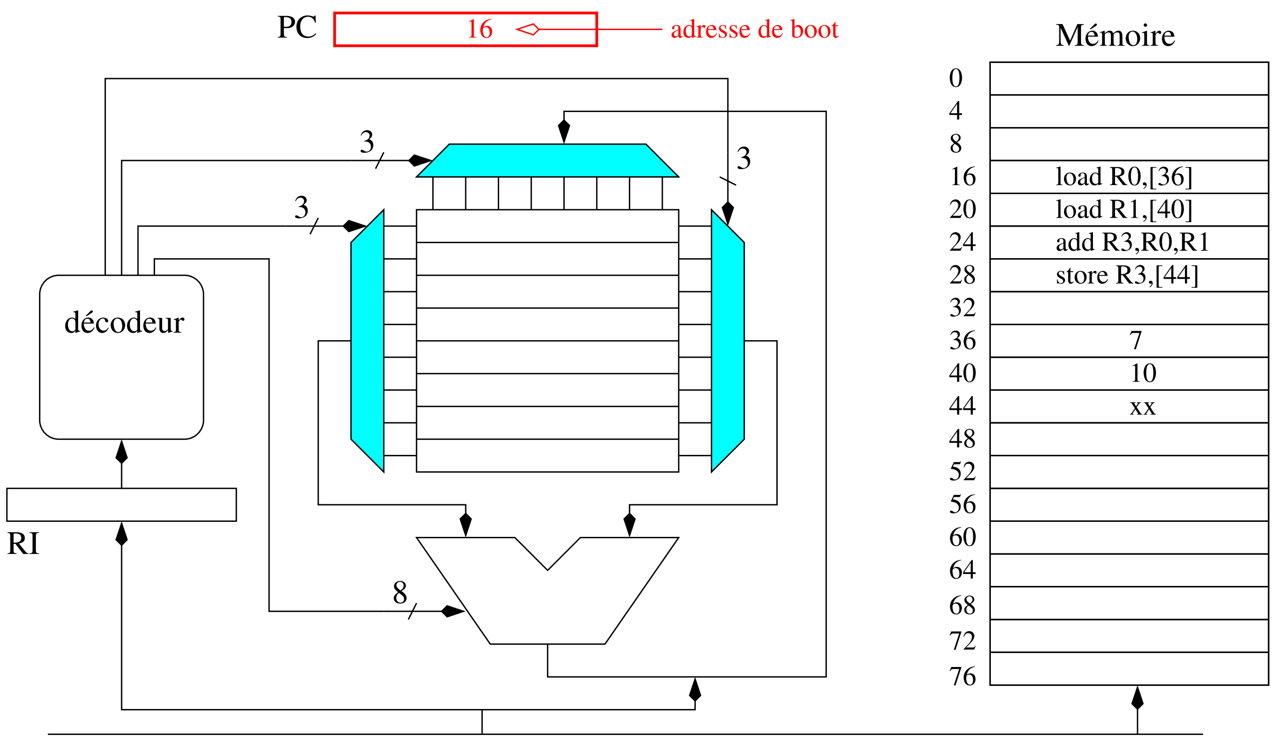
Program execution on a Processor (8 general purpose registers)



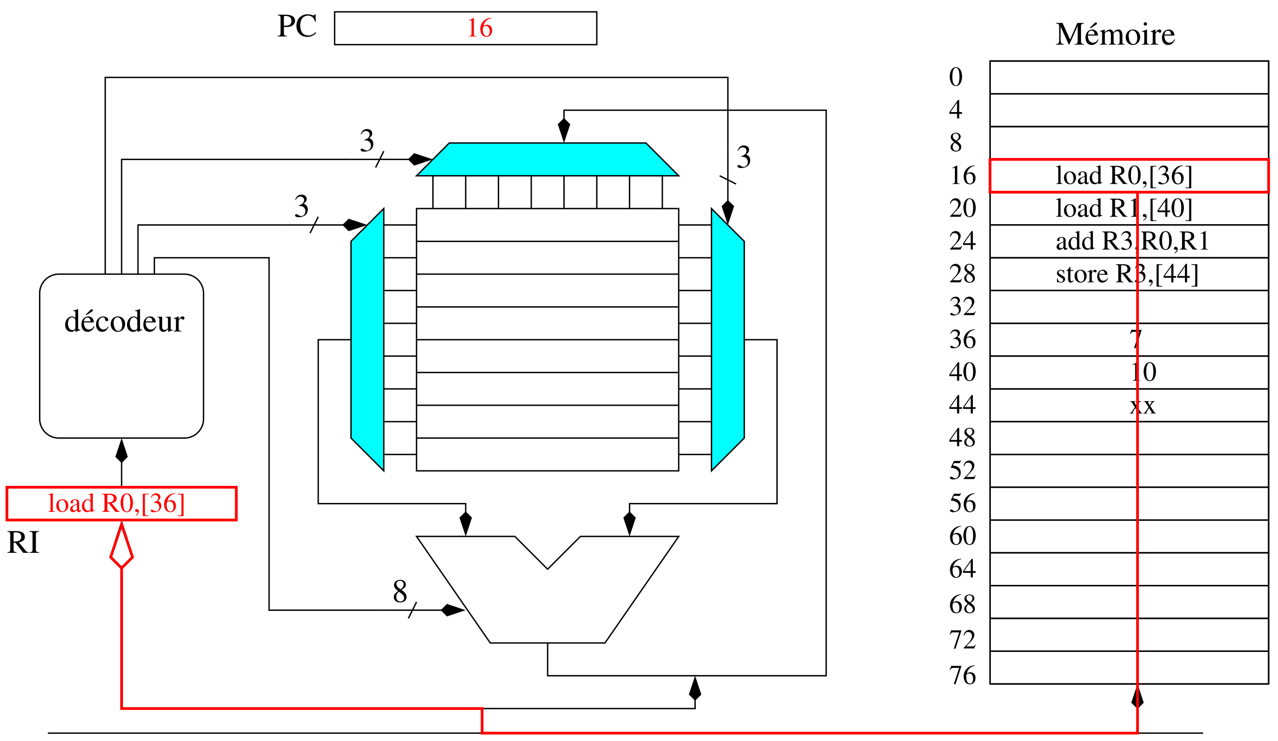
Program execution on a Processor (8 general purpose registers)



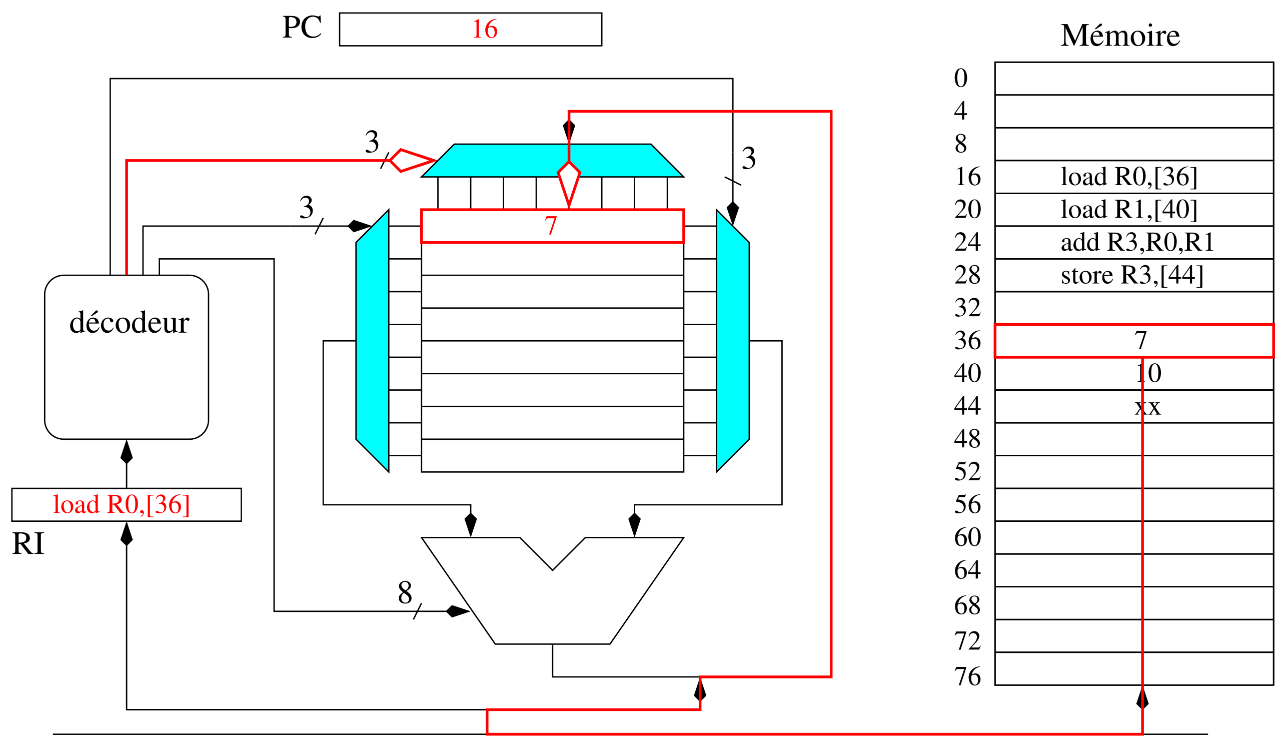
Program execution on a Processor (8 general purpose registers)



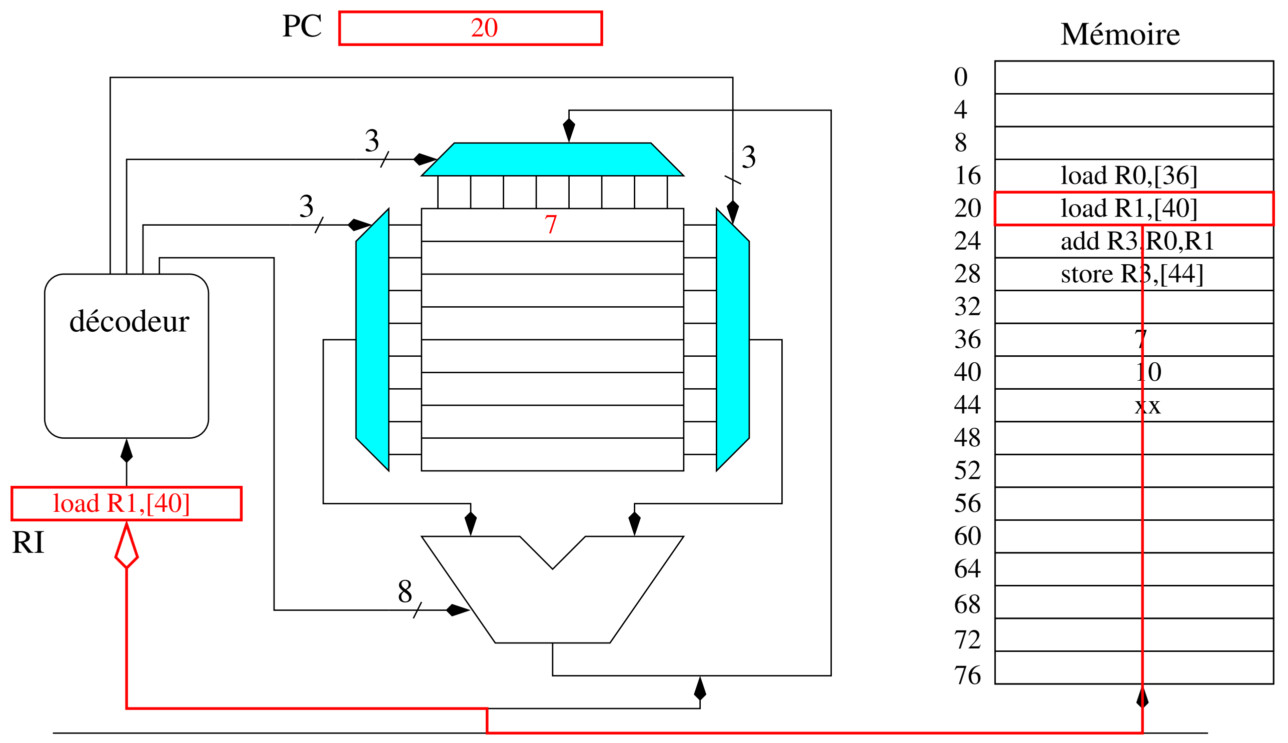
Program execution on a Processor (8 general purpose registers)



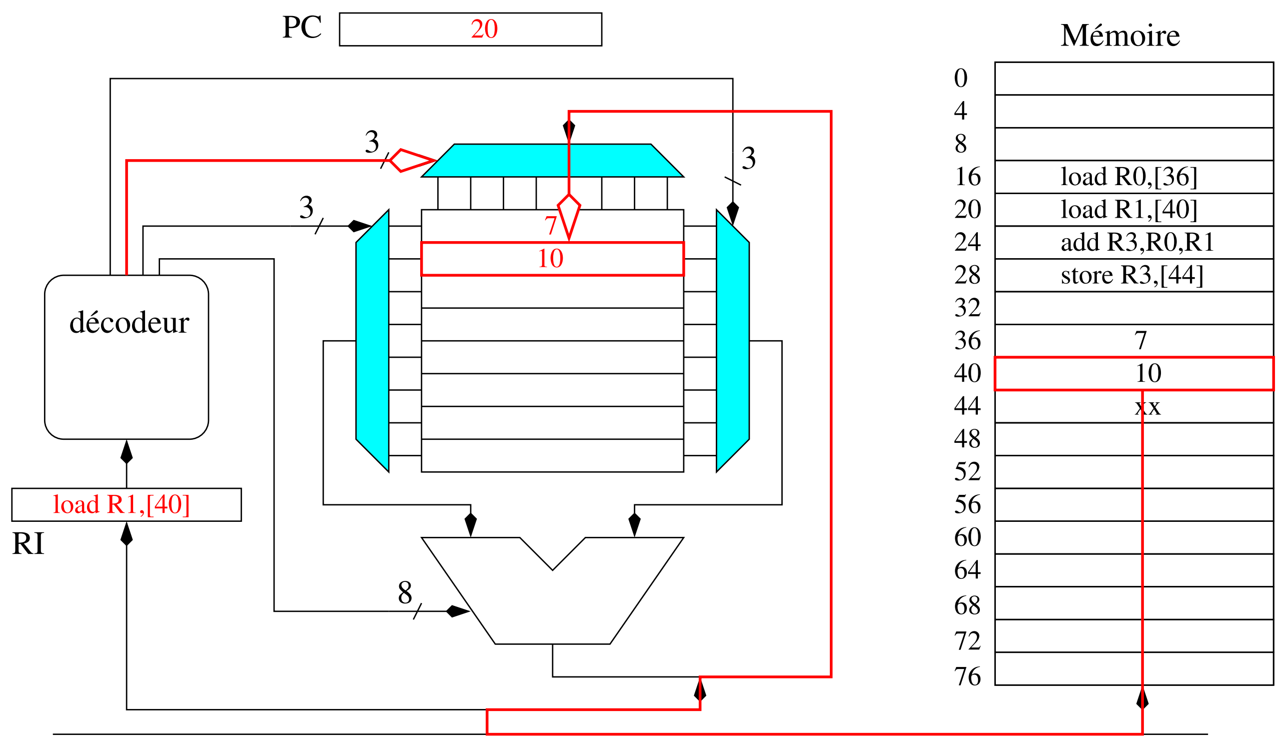
Program execution on a Processor (8 general purpose registers)



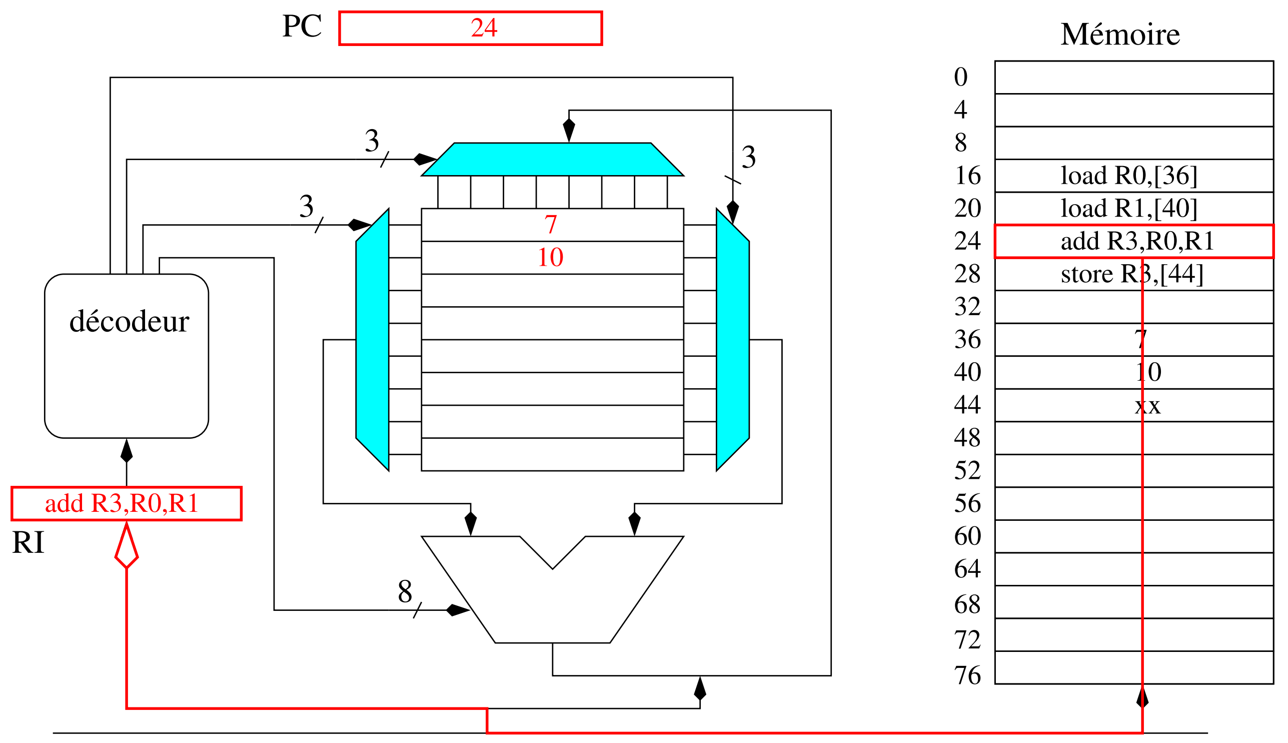
Program execution on a Processor (8 general purpose registers)



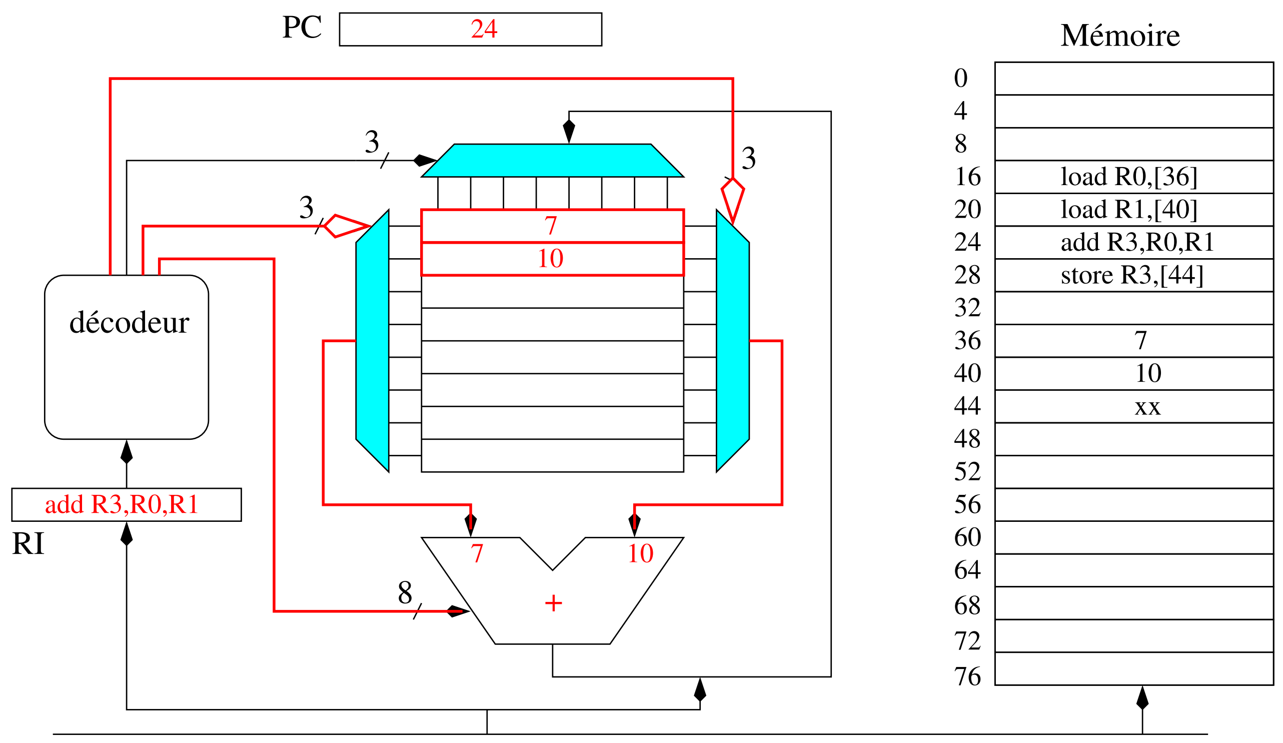
Program execution on a Processor (8 general purpose registers)



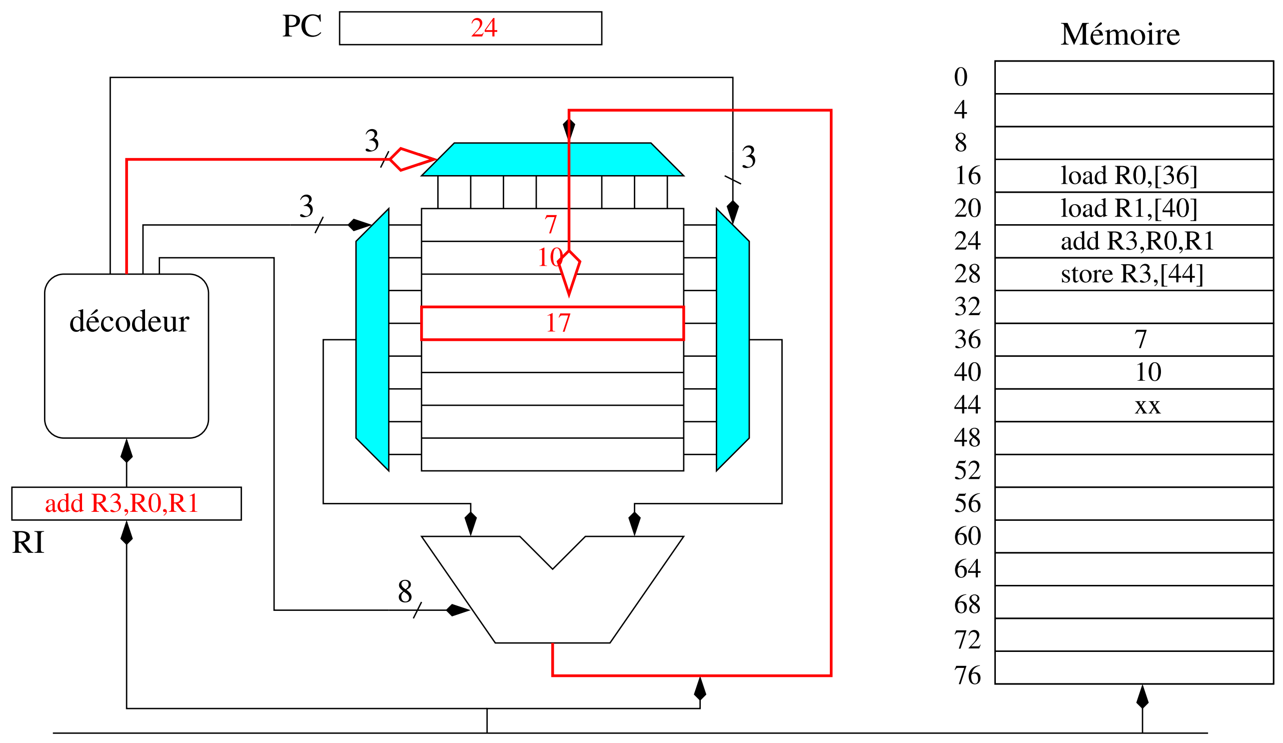
Program execution on a Processor (8 general purpose registers)



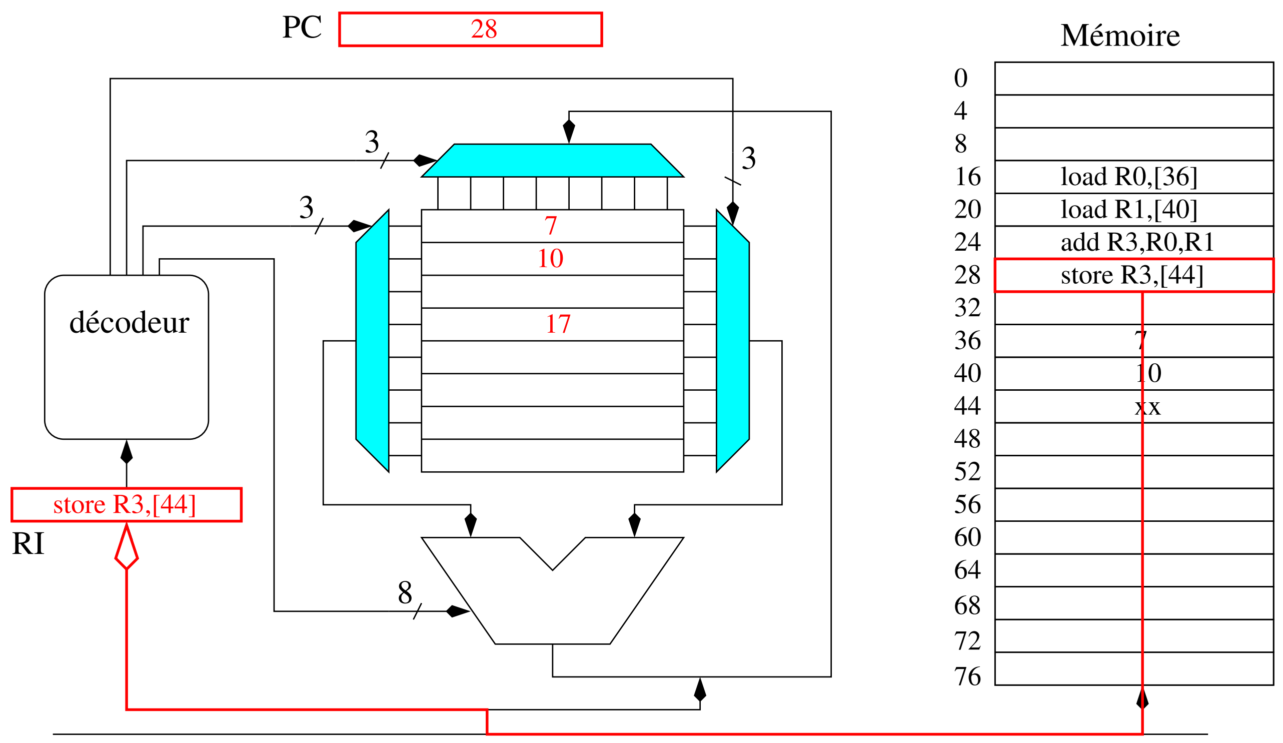
Program execution on a Processor (8 general purpose registers)



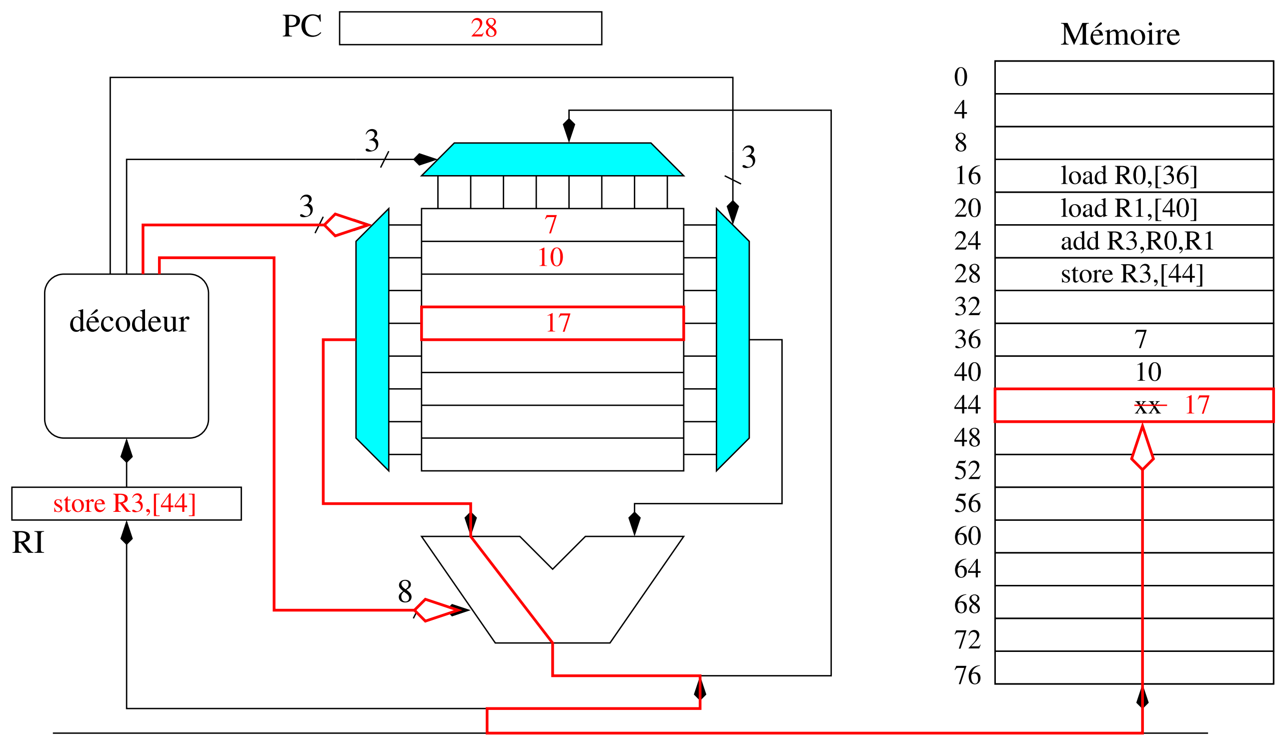
Program execution on a Processor (8 general purpose registers)



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 $256 - 96 = 160 = 10100000_2$

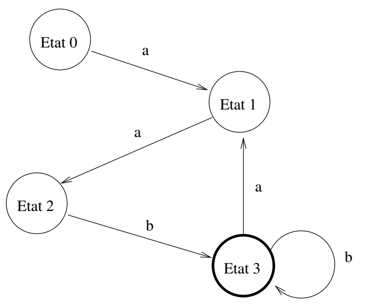
Table of Contents

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

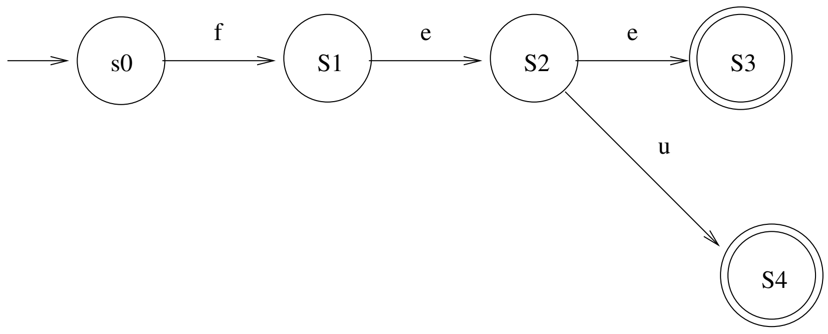
- Un automate est une collection de K états numérotés de 0 à K-1, ainsi qu'une collection de transitions
- Un état particulier est l'état initial.
- Tous les états sont soit des états d'acceptation et soit des états de refus
- Les transitions, sont étiquetées
  1. soit par des actions (par exemple, je 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



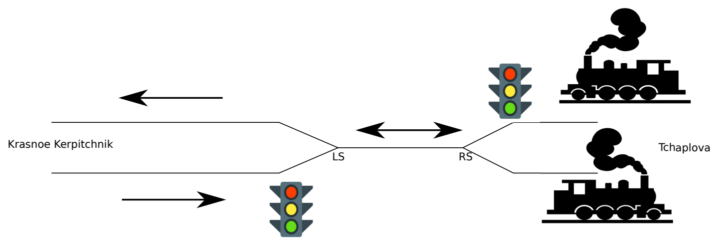
- `fee` → reconnu
- `feu` → reconnu
- `fei` → non reconnu (impossible de lire `'i'`)
- `fe` → non reconnu (arrêt dans un état non final)

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.

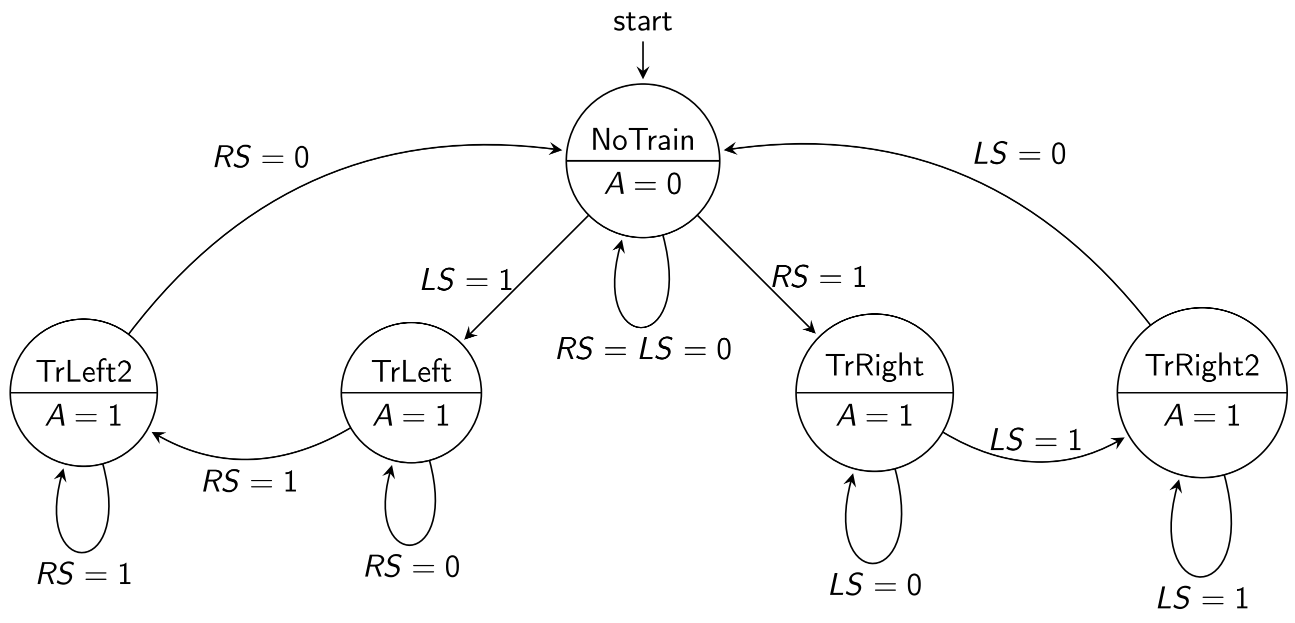
Table of Contents

Example from the poly

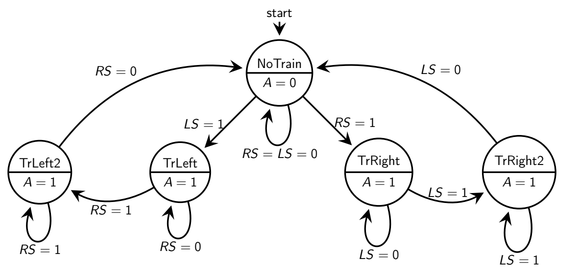


- A piece of unique train track for both train directions between the cities `T.` et `K.`
- Sensors triggered by train weight on rallways will command red lights when the track is used by a train.
- Modeling:
  - A booleen `A` (for 'Ampoule') indicating the state of the red light
  - Two booleans (LS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

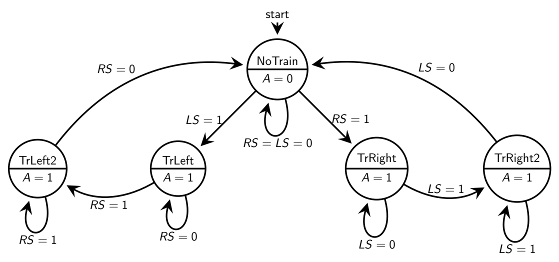


The Russian train automaton



- 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

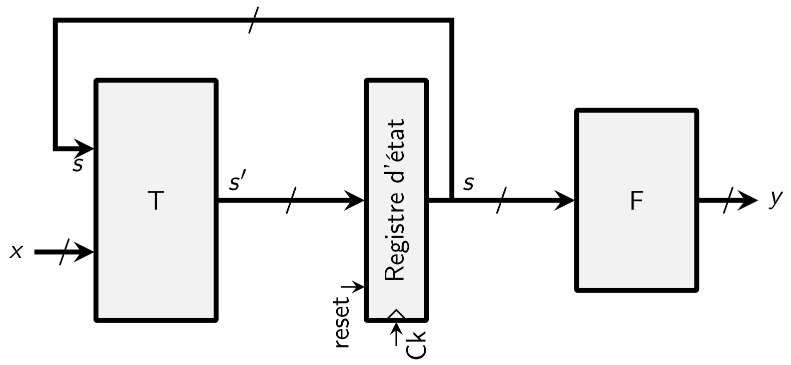


- The Inputs are `RS` and `LS` sensors Boolean values
- The Output is the value of Boolean `A`
- The functions (Transition and Output) can be defined by tables ⇒
- `X` means 'don't care'

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

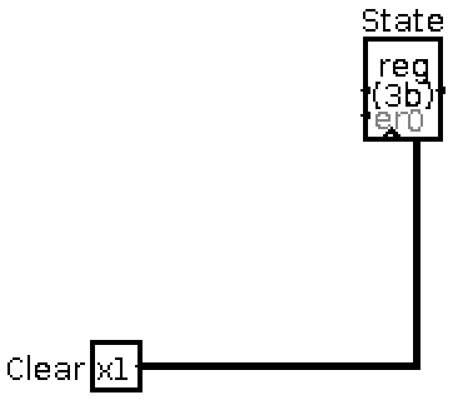


- `s` is current state, `s'` is next state, `x` are input bits, `y` are output bits.
- `Ck` and `reset` are not considered as inputs
- State change will occur on each rising edge of the Clock.

Implementation in Logisim

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

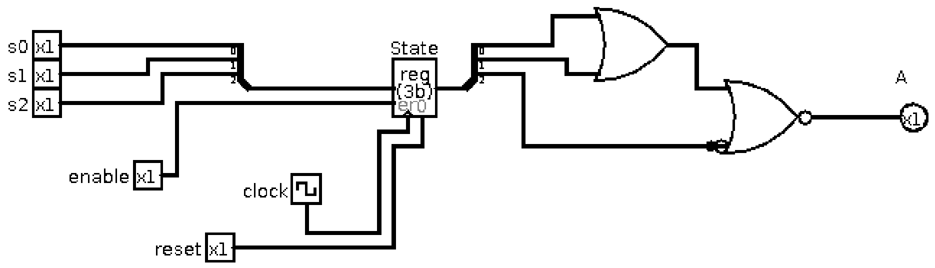
value (binary)	state
100	NoTrain
000	TrRight1
001	TrRight2
010	TrLeft
011	TrLeft2



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

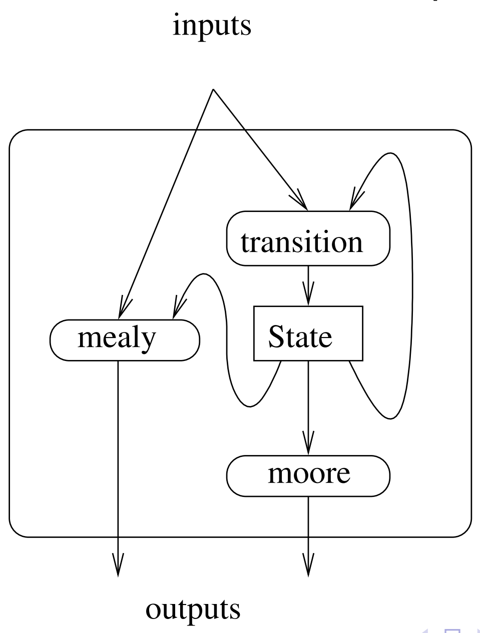
Table of Contents

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 $\leqslant$ n
    x := x+p
    q := q+1
fin tant que
sortie $Q$ := q
```

Lab topic: proposed circuit to realize it

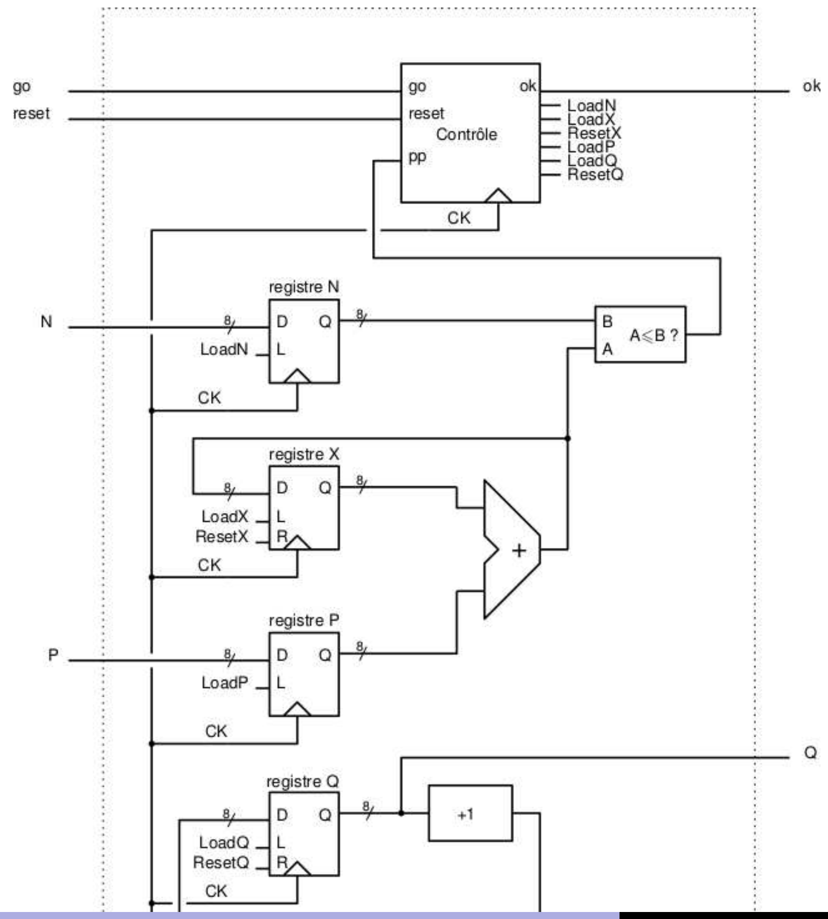


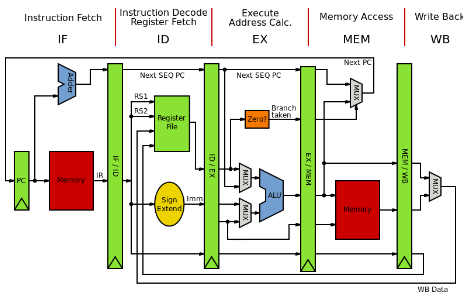
Table of Contents

Study a real ISA: MIPS

- We study in more detail a particular assembly code
- Course inspired from
  - Architecture course of Peter Niebert and Séverine Fratani (U. Marseille) `http://pageperso.lif.univ-mrs.fr/~peter.niebert/archi2014.php`
  - MIPS web site `https://www.mips.com/`
  - And of Course F. de Dinechin IF Architecture course (with bits of Christian Wolf)

MIPS Processor

- MIPS stands for *Microprocessor without Interlocked Pipeline Stages*
- MIPS designed by MIPS Computer Systems in 1985.
- Many version up to today (MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V and MIPS32, MIPS64 as well)
- Used in PC, and servers (DEC, NEC, Silicon Graphics) and for video games (Nintendo 64, Sony PlayStation, PlayStation 2)
- Gave birth to RISC-V, an open-source architecture.



MIPS Processor organisation

- a *register-to-register* (or *load/store*) architecture
- MIPS use 3-adress instructions (destination is the first operand)
- 32 registers
- A program counter ($PC) of 32 bits
- an Instruction register ($IR) of 32 bits
- Addressable memory of 2^{32} bytes
  - $\Leftrightarrow$ 2^{30} words of 4 bytes

understanding MIPS assembly

- From C to assembly:

```
mipsel-linux-gcc prog.c -S -o prog.s
```

prog.c

```
  ...
  i = N*N + 3*N;
  ...
```

prog.s

```
...
  lw      $t0, 4($gp)         # fetch N
  mult    $t0, $t0, $t0       # N*N
  lw      $t1, 4($gp)         # fetch N
  ori     $t2, $zero, 3       # 3
  mult    $t1, $t1, $t2       # 3*N
  add     $t2, $t0, $t1       # N*N + 3*N
  sw      $t2, 0($gp)         # i = ...
...
```

MIPS assembly: compiler optimization (academic)

- From C to optimized assembly:

```
mipsel-linux-gcc prog.c -S -O3 -o prog.s
```

prog.c

```
  ...
  i = N*N + 3*N;
  ...
```

prog.s

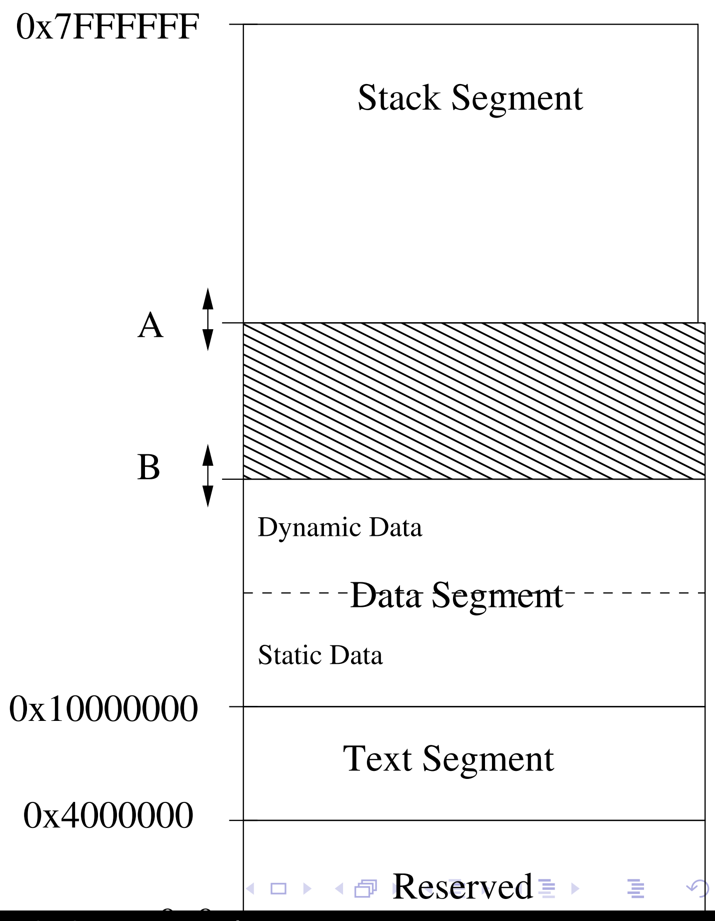
```
...
lw      $t0, 4($gp)         # fetch N
add     $t1, $t0, $zero     # cp N to $t1
addi    $t1, $t1, 3         # N+3
mult    $t1, $t1, $t0       # N*(N+3)
sw      $t1, 0($gp)         # i = ...
...
```

MIPS register

- 32 registers in the *register file*
- Named
  - by their number: `$0 $1 ...$31`
  - or by their name `$zero $at $v0 $v1 $a0 ...$a3 ...`
- `$0` (`$zero`) contains value 0
- `$a0 ...$a3` are used to pass (first four) arguments of a function call
- `$v0 $v1` are used to transmit functions result
- `$s0 ...$s7` and `$t0 ...$t9` are working registers, used for CPU computations
- `$sp` is the stack pointer
- `$fp` is the frame pointer (explained later)
- `$ra` contains the return address (after the end of current function)
- `$gp` is a pointer to global area
- `$k0, $k1` and `$at` are reserved register (for kernel and assembler)

MIPS Memory map

- The Memory Map is a convention to organize memory that must respect each code to be compatible with others.
- The MIPS memory map (very similar to all memory map) is simple
- Here we have only one physical memory chip: the RAM.



MIPS assembly addressing mode

- Addressing mode means: how the address is computed in an assembly instruction

format	address computation
$register	content of register
imm	immediate value
imm ($register)	immediate + content of register
label	addresse of label
label ± imm	addresse of label ± immediate value
label ± imm (register)	addresse of label ± (immediate value + content of register)

Example of MIPS adressing mode

- `add $s0, $s2, $s1`
  - puts in $s0 the value of $s1 plus the value of $s2.
  - `$s0=$s1+$s2`
- `addi $s0, $s1, 1`
  - puts in $s0 the value of $s1 plus 1.
  - `$s0=$s1+1`
- `lw $s0, 10($s3)`
  - puts in $s0 the value situated in memory at the address obtained by adding 10 to the content of $s3.
  - `$s0=Memory[$s3+10]`
- `bne $s0, $s3, label`
  - branch to address of `label` if values in $s0 and $S3 are different.
  - `if ($s0 != $s3) then $PC=label`

Format of MIPS instructions

- 3 types of format: R-Type, I-Types and J-Types
- R-types:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	func

  - Used for 3-register instructions
  - op is the operation code or *opcode* that specifies the operation
  - `rs` and `rt` are the first and second source register
  - `rd` is the destination register
  - `shamt` is used for shift instruction
  - `func` is used with op to select arithmetic operation

I-Types instruction

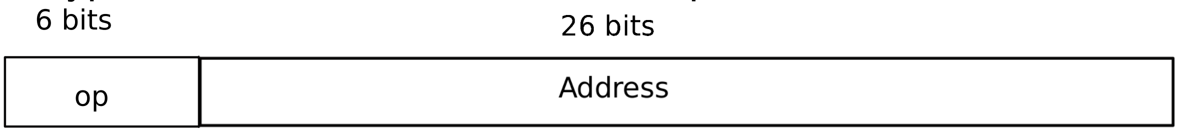
- I-Types instruction are used for load, store, branch and immediate instruction.

6 bits	5 bits	5 bits	16 bits
op	rs	rt	Address

  - `rs` is a source register (an address) for loads, store
  - `rs` is an operand for conditionnal branch
  - `rt` is a source register for branch
  - `rt` is a destination register for other I-Types instruction
  - The `address` field is a 16 bit's integer in two's-complement code , ranging from -32 768 to 32 767 (remind that this is a problem in many cases)

J-Types instruction

- J-Types instruction are used for Jump to absolute address



  - The `address` field is a 26 bit's integer containing the address of the *word*, hence the real address is obtain by multiplying by four (shifting two bits).
  - can jump from address 0 to 2^{28}=256MB from $PC.
  - For longer jump, on can use the instruction `jr`:
    `jr $ra`
    jump to 32 bit address contained in register $ra

Basic arithmetic and logic instruction

- R-Types instructions: add, sub, mul, div, and, or, xor
  - `add $t0, $t1, $t2      // $t0 = $t1 + $t2`
  - `mul $s0, $s1, $a0      // $s0 = $s1 * $a0`, pseudo
- I-types for immediate operand operation:
  - `addi $t0, $t1, 4      // $t0 = $t1 + 4`
  - `addi $t0, $0, 4      // $t0 = 4`
  - `li $t0, 4      // $t0 = 4, pseudo`

Load and store

- MIPS load and store operation use *indexed addressing*
  - the address operand specifies a signed constant and a register
  - These values are added to generate effective address
- byte instruction: `lb` and `sb` transfer one byte
  - `lb $t0, 20($a0)      // $t0=Memory[$a0+20]`
  - `sb $t0, 20($a0)      // Memory[$a0+20]=$t0`
  - `sb` stores only the lowest byte of operand register
- Word instruction: `lw` and `sw` operates on word (i.e. 32 bits)
- Remind that address have to be aligned to 32 bit world, hence must be multiple of 4.

Branches

- Conditional branch
  - `bne $t0, $t1, Label`
  - if `$t0` and `$t1` have different values, the next instruction to execute is at address Label
  - `beq $t0, $t1, Label      // same thing if $t0=$t1`
- Unconditionnal branch
  - `j toto // next instruction executed is at address toto`
  - `jr $s2 // next instruction executed is at address contained in $s2`
- These are the only way of implementing loops in assembly:

```
li $t2, 0
li $t3, 1
while: beq $t1, $0, done
       add $t2, $t1, $t2
       sub $t1, $t1, $t3
       j while
done:
```

```
t2=0
while (t1 != 0) {
   t2 = t2 + t1
   t1=t1-1
}
```

Function control flow in MIPS

- MIPS uses the *jump-and-link* (`jal`) instruction to call functions
  - Example:

    ```
    jal Fact
    ```
  - saves the return address (i.e. the address of the following instruction) in the `$ra` register and jumpt to the address of `Fact`
- At the end of the execution of `Fact`, the instruction `jr $ra` jumps back to the address stored in `$ra`
- Arguments transmited to `Fact` are stored in registers `$a0` ...`$a3`
- Return values of `Fact` are stored in registers `$v0` `$v3`

Who save the register during Function call?

- When a function call occurs: `jal Fact`, who save the register?
  - The Caller (who knows which register he will use after the call)?
  - Or the callee (who knows which register he will use during its execution)?
- This convention is part of the *calling convetion* or ABI *application binary interface*.
- For MIPS:
  - `$t0 - $t9 $a0 - $a3 $v0 $v1` are caller saved (if needed)
  - `$s0 - $s7 $ra` are callee saved (if needed)

Function call example with MIPS

- Let says: function B calls function C
- Function B wants to save `$t0`, `$t1` and `$a0` because it will need them after the return of C.
- this is done using the stack via the stack pointer `$sp`

The Stack

- The stack is use to store all *local* information (in the sense local to the current function)
- That includes (say for function C):
  - local variable
  - Callee saved register if needed
  - Return address (i.e. the instruction following the `jal C` instruction).
  - (sometimes) the parameters passed to C
  - (sometimes) the result of C
  - In many ISA, the parameters and the results are passed through dedicated registers
- All these data constitute the frame of the fonction instance.
- the frame pointeur points to the frame of the current function
- For MIPS, the frame pointer is `$fp`

Function B calls C

```
B    ...                 beguinning of  B
     ...
     sw $t0,0($sp)       saving  $t0 in stack
     sw $t1,-4($sp)      saving  $t1 in stack
     sw $a0,-8($sp)      saving  $a0 in stack
     sub $sp,$sp,12      correct stack pointer
     jal C               call to C function
     lw $a0,4($sp)       restoring return addresse of B from stac
     lw $t1,8($sp)       restoring $s1 from stack
     sw $t0,12($sp)      restoring  $s0
     add $sp,$sp,12      adjusst  stack pointeur value
     ...
     jr $ra              end of B
     ...
```

Sketching code of C function

```
C:
    subu    $sp,$sp,40     # C need 40 Bytes for its frame
    sw      $ra,32($sp)    # store return address (inst. in B)
    sw      $fp,28($sp)    # store frame pointer
    sw      $s0,24($sp)    # store $s0 (because C uses it)
    move    $fp,$sp        # $fp <- $sp: frame pointer of C se
       ....
      ....
    lw      $ra,32($sp)    # $ra <- return address (in B)
    lw      $fp,28($sp)    # $fp  <- frame pointeur of B
    lw      $s0,24($sp)    # restore $s0
    addu    $sp,$sp,40     # $sp <- $sp+40, restore B stack po
    j       $ra            # return to $ra (B function)
```

Table of Contents

Procedure abstraction

- Let's pause a while to come back to high level langage
- What is a function (or a procedure)?
- How its isolation mecanisme (local variable) is implemented?
- This is implemented with a very fundamental mecanism: the Stack and the Activation Record (or Frame) of each procedure.

Notion of procedure

- Procedures (or functions) are the basic units for compilers
- Three important abstraction:
  - Control abstraction: parameter passing and result transmission
  - Memory abstraction: variable lifetime (local variables)
  - Interface: procedure's signature

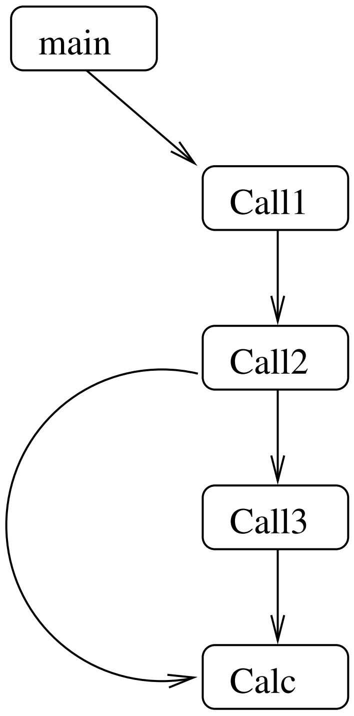
Procedure Control Transfer

- Transfer mechanism of control between procedures:
  - when calling a procedure, the control is given to the procedure called;
  - when this called procedure ends, the control is returned to the calling procedure.
  - Two calls to the same procedure create two em independent instances (or invocations).
- two useful graphic representations:
  - The call graph: represents the information written in the program.
  - The call tree: represents a particular execution.

Call Graph

```
procedure calc;
begin { calc}
    ...
end;
procedure call1;
    var y...
    procedure call2
        var z: ...
        procedure call3;
            var y....
            begin { call3}
                x:=...
                calc;
            end;
        begin { call2}
            z:=1;
            calc;
            call3;
        end;
    begin { call1}
        call2;
        ...
    end;
```

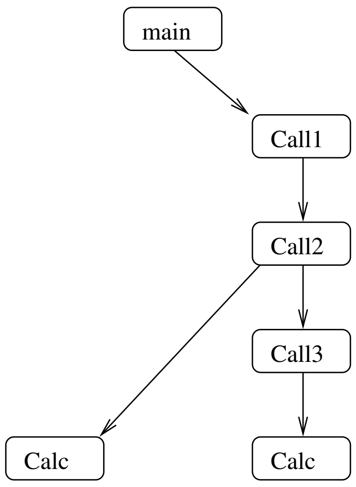
Call Graph:



Call Tree

```
procedure calc;
begin { calc}
    ...
end;
procedure call1;
    var y...
    procedure call2
        var z: ...
        procedure call3;
            var y....
            begin { call3}
                x:=...
                calc;
            end;
        begin { call2}
            z:=1;
            calc;
            call3;
        end;
    begin { call1}
        call2;
    end;
```

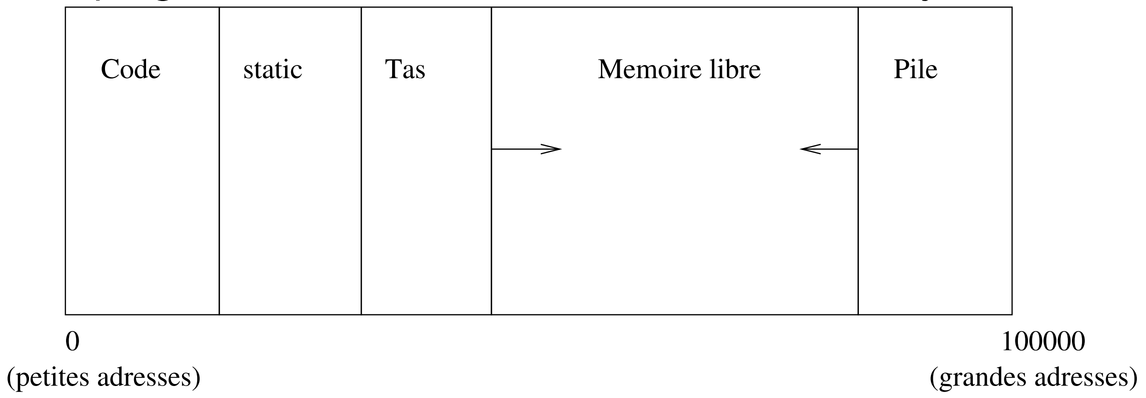
Call tree for one particular execution:



main calls *call*$_1$
call$_1$ calls *call*$_2$
call$_2$ calls *calc*
calc returns to *call*$_2$
call$_2$ calls *call*$_3$
call$_3$ calls*calc*
calc returns to *call*$_3$
call$_3$ returns to *call*$_2$
call$_2$ returns to *call*$_1$
call$_1$ returns to *main*

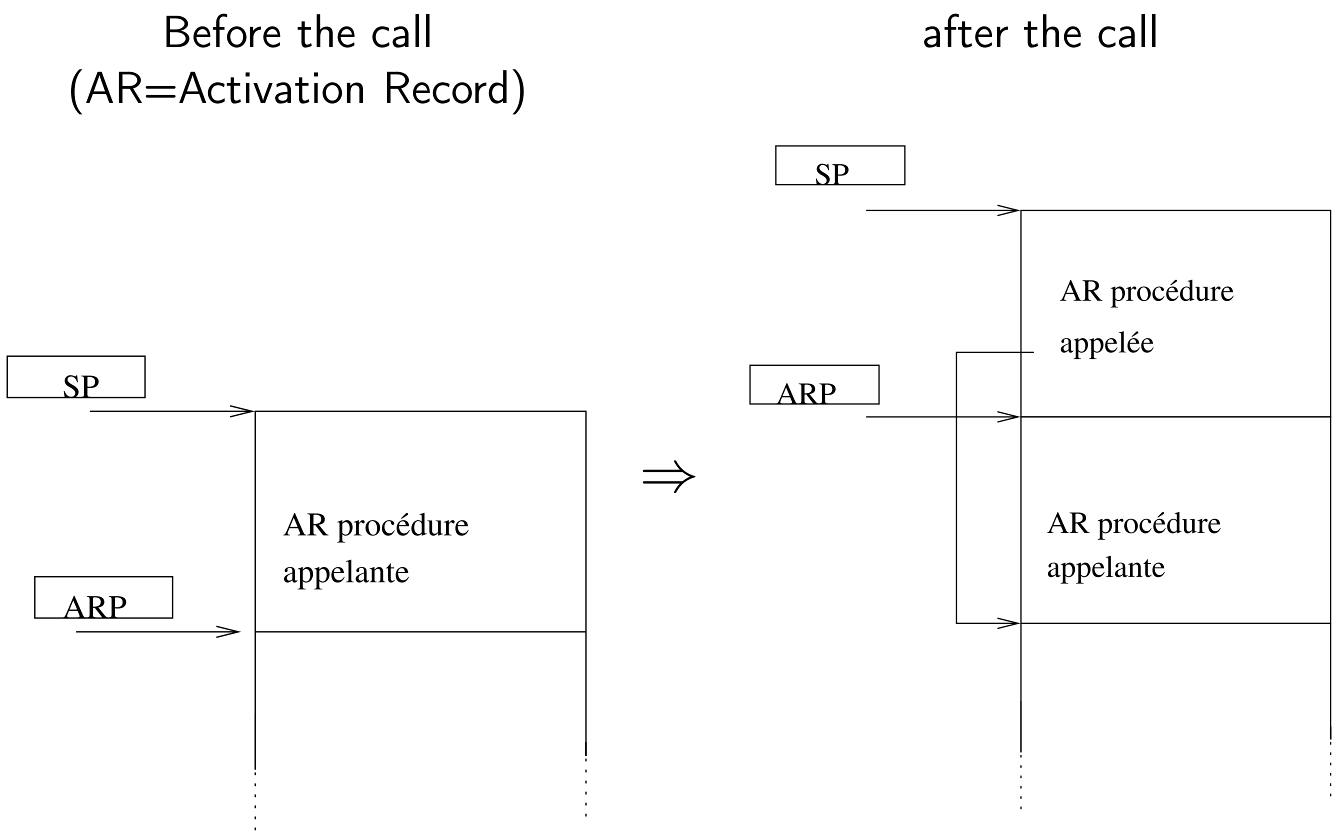
Execution Stack

- The transfer of control mechanism between procedures is implemented thanks to the *execution stack*.
- The programmer has this vision of virtual memory:



- The *heap* is used for dynamic allocation.
- The *stack* is used for the management of contexts of procedures (local variable, etc.)

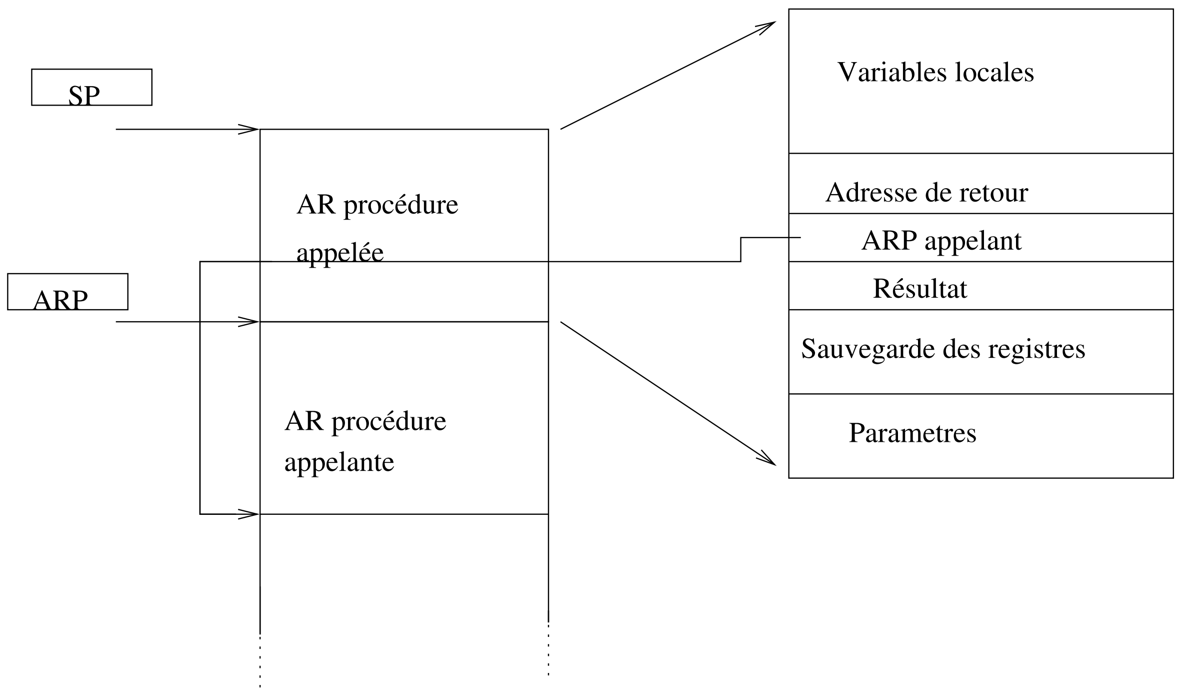
Function call: status of the stack



Activation record

- Calling a procedure: Stacking the *activation record* (or *frame*).
- Need of a dedicated pointer for that: the *activation record pointer* (ARP) or *frame pointeur* (`$fp`))
- The frame allows to set up the *context* of the procedure.
- This frame contains
  - The space for local variables declared in the procedure
  - Information for restoring the context of the calling procedure:
    - Pointer to the frame of the calling procedure (ARP or FP for em frame pointer).
    - Address of the return instruction (statement following the call of the appellant proceedings).
    - Eventually save the state of the registers at the time of the call.

Content of the Frame



Return to calling function

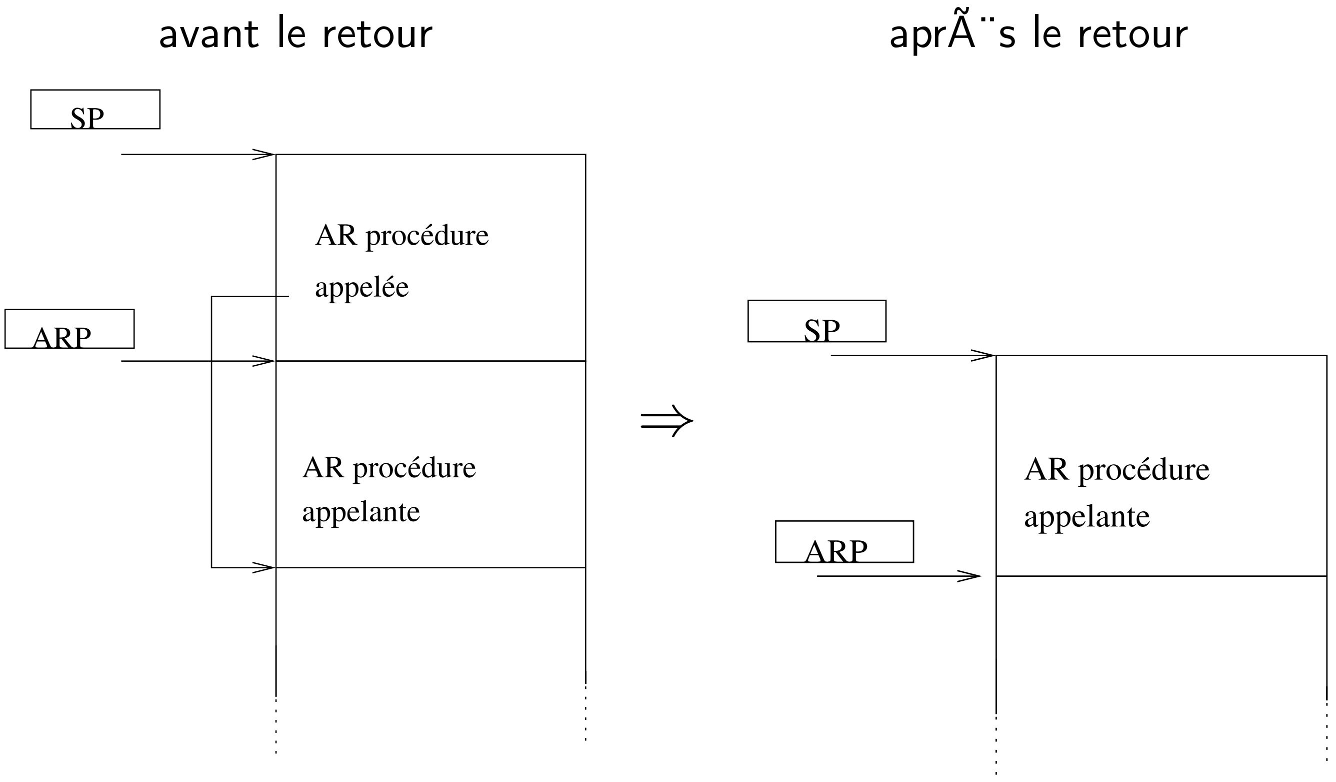


Table of Contents

Coming back to previous call example with B and C

- Let says: function B calls function C
- Function B wants to save `$t0`, `$t1` and `$a0` because it will need them after the return of C.
- this is done using the stack via the stack pointer `$sp`

The Stack

- The stack is use to store all *local* information (in the sense local to the current function)
- That includes (say for function C):
  - local variable
  - Callee saved register if needed
  - Return address (i.e. the instruction following the `jal C` instruction).
  - (sometimes) the parameters passed to C
  - (sometimes) the result of C
  - In many ISA, the parameters and the results are passed through dedicated registers
- All these data constitute the frame of the fonction instance.
- the frame pointeur points to the frame of the current function
- For MIPS, the frame pointer is `$fp`

Function B calls C

```
B    ...                  beguinning of  B
     ...
     sw $t0,0($sp)        saving  $t0 in stack
     sw $t1,-4($sp)       saving  $t1 in stack
     sw $a0,-8($sp)       saving  $a0 in stack
     sub $sp,$sp,12       correct stack pointer
     jal C                call to C function
     lw $a0,4($sp)        restoring return addresse of B from stac
     lw $t1,8($sp)        restoring $s1 from stack
     sw $t0,12($sp)       restoring  $s0
     add $sp,$sp,12       adjusst  stack pointeur value
     ...
     jr $ra               end of B
     ...
```

Sketching code of C function

```
C:
    subu    $sp,$sp,40      # C need 40 Bytes for its frame
    sw      $ra,32($sp)     # store return address (inst. in B)
    sw      $fp,28($sp)     # store frame pointer
    sw      $s0,24($sp)     # store $s0 (because C uses it)
    move    $fp,$sp         # $fp <- $sp: frame pointer of C se
       ....
      ....
    lw      $ra,32($sp)     # $ra <- return address (in B)
    lw      $fp,28($sp)     # $fp  <- frame pointeur of B
    lw      $s0,24($sp)     # restore $s0
    addu    $sp,$sp,40      # $sp <- $sp+40, restore B stack po
    j       $ra             # return to $ra (B function)
```

MIPS Assembly for programme fib

Fibbonacci suite program:

```
int fib (int i)
{
  if (i<=1) return(1);
  else return(fib(i-1)+fib(i-2));
}

int main (int argc, char *argv[])
{
  fib(2);
}
```

Assembleur MIPS pour programme fib

```
fib:
        .frame       $fp,40,$ra       # vars= 8, regs= 3/0, args= 16, extra= 0
        .mask       0xc0010000,-8
        .fmask       0x00000000,0
        subu       $sp,$sp,40        # SP <- SP-40 :AR de 40 octet (10 mots)
        sw        $ra,32($sp)        # stocke adresse retour SP+32
        sw        $fp,28($sp)        # stocke ARP appelant SP+28
        sw        $s0,24($sp)        # sauvegarde registre $s0
        move       $fp,$sp           # ARP <- SP
        sw        $a0,40($fp)        # stocke Arg1 dans la pile (ARP+40)
        lw        $v0,40($fp)        # charge Arg1 dans $v0
        slt        $v0,$v0,2         # $v0 <- 1 si $v0<2 0 sinon
        beq        $v0,$0,$L2        # branch L2 si $v0==0
        li        $v0,1              # $v0 <- 0x1 ($v0 sera le registre contenant le  res)
        sw        $v0,16($fp)        # stocke le resultat dans la pile
        j        $L1                 # saute Ã  L1
$L2:
        lw        $v0,40($fp)        # charge Arg1 dans $v0
        addu       $v0,$v0,-1        # retranche 1
        move       $a0,$v0           # $a0 <- $v0 ($a0 contient Arg1 pour l'appel recursif)
        jal        fib                # jump and link fib ($ra<-next instr)
        move       $s0,$v0           # $s0 <- $v0 ($v0: res appel fib)
        lw        $v0,40($fp)        # charge Arg1 dans $v0
        addu       $v0,$v0,-2        # retranche 2
        move       $a0,$v0           # $a0 <- $v0 ($a0: contient Arg1 pour l'appel recursif)
        jal        fib                # jump and link fib ($ra<-next instr)
        addu       $s0,$s0,$v0       # $s0 <- $s0+$v0 ($v0: res appel fib)
        sw        $s0,16($fp)        # stocke le resultat dans la pile
```

Assembleur MIPS pour programme fib

```
$L1:
       lw       $v0,16($fp)         # $v0 <- resultat
       move       $sp,$fp           # SP <- ARP
       lw       $ra,32($sp)         # $ra <- adresse retour
       lw       $fp,28($sp)         # ARP <- ARP appelant
       lw       $s0,24($sp)         # restaure $s0
       addu       $sp,$sp,40        # SP->SP+40
       j       $ra                  # jump adresse retour
       .end       fib
       .align       2
       .globl       main
       .ent       main
main:
       .frame       $fp,24,$ra      # vars= 0, regs= 2/0, args= 16, extra= 0
       .mask       0xc0000000,-4
       .fmask       0x00000000,0
# partie ajoutÃ©e pour afficher le resultat
.data
str: .asciiz  "Le resultat est "
.text
       subu       $sp,$sp,24     # SP <- SP-24 :AR de 24 octet (6 mots)
       sw       $ra,20($sp)      # stocke adresse retour SP+20
       sw       $fp,16($sp)      # stocke ARP appelant SP+16
       move       $fp,$sp        # ARP <- SP
       sw       $a0,24($fp)      # stocke Arg1 dans la pile (ARP+24)
       sw       $5,28($fp)       # stocke Arg2 dans la pile (ARP+48)
       li       $a0,2            # $a0 <- 2 ($a0: Arg1)
       jal       fib             # jump and link fib ($ra<-next instr)
# partie ajoutÃ©e pour afficher le resultat
       move $16,$2          # $16 <- resultat de l'appel a fib
       li $v0, 4          # $v0 <- code pour afficher une chaine (4)
```

Table of Contents

Assember directives

.align n	Align the next datum on specified byte boundary (0=byte, 2=word, etc.).
.ascii str	store the string in memory, but do not null-terminate it.
.asciiz str	Store the string in memory and null-terminate it.
.byte b1,..., bn	Store the n values in successive bytes of memory.
.data <addr>	The following data items should be stored in the data segment
.double d1,..., dn	Store the n floating point double precision numbers in successive memory locations.
.extern sym size	Declare that the datum stored at sym is size bytes large and is a global symbol.
.globl sym	Declare that symbol sym is global and can be referenced from other files.
.space n	Allocate n bytes of space in the current segment.
.text <addr>	The next items are put in the user text segment.
.word w1,..., wn	Store the n 32-bit quantities in successive memory words.

example 1 (Fratini/Niebert)

```
    bne $s0, $s1, Test
    add $s2, $s0, $s1
Test:
```

example 2 (Fratini/Niebert)

```
    beq $s4, $s5, Lab1
    add $s6, $s4, $s5
    j Lab2
Lab1:sub $s6, $s4, $s5
Lab2:
```

example 3 (Fratini/Niebert)

```
    li $t2, 0
    li $t3, 1
while:beq $t1, $0, done
    add $t2, $t1, $t2
    sub $t1, $t1, $t3
    j while
done:
```

example 4 (U. Illinois)

```
        .data
var1:   .word  23          # declare storage for var1; initial
                           # value is 23

        .text
__start:
        lw $t0, var1          # load contents of RAM location in
                              # register $t0:  $t0 = var1
        li $t1, 5             #  $t1 = 5   ("load immediate")
        sw $t1, var1          # store contents of register $t1
                              #into RAM:  var1 = $t1
done
```

example 5 (U. Illinois)

```
        .data
array1: .space 12          #  declare 12 bytes of storage to
                           # hold array of 3 integers
        .text
__start: la $t0, array1          #  load base address of array
                                 #register $t0
        li $t1, 5                #$t1 = 5   ("load immediate")
        sw $t1, ($t0)            #first array element set to 5;
                                 #indirect addressing
        li $t1, 13               #$t1 = 13
        sw $t1, 4($t0)           #second array element set to 1
        li $t1, -7               #$t1 = -7
        sw $t1, 8($t0)           #third array element set to -7
done
```

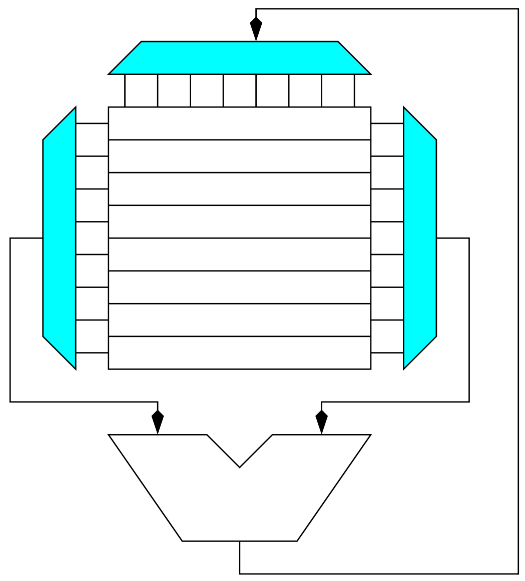
Documentation on MIPS assembly

More precise documentation on MIPS assembly code can be obtained at:

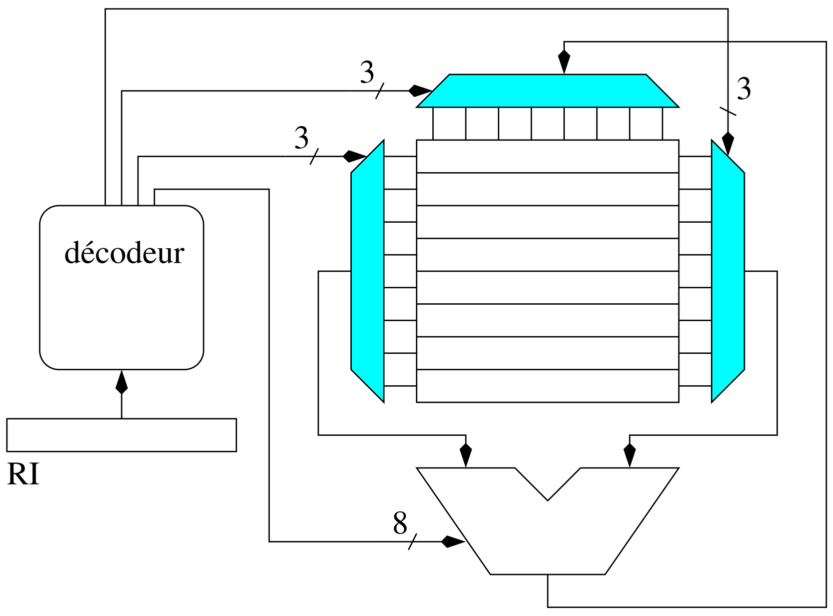
- `http://igm.univ-mlv.fr/ens/IR/IR1/2007-2008/Archi/ManuelSPIM.php` (brief documentation from U. Marne la vallÃ©e)
- `http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm` (brief documentation from U. of illinois at Chicago).
- `https://en.wikibooks.org/wiki/MIPS_Assembly`, wikibook
- `https://www.cs.unibo.it/~solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf`, MIPS Assembly langage programmer's Guide.

Table of Contents

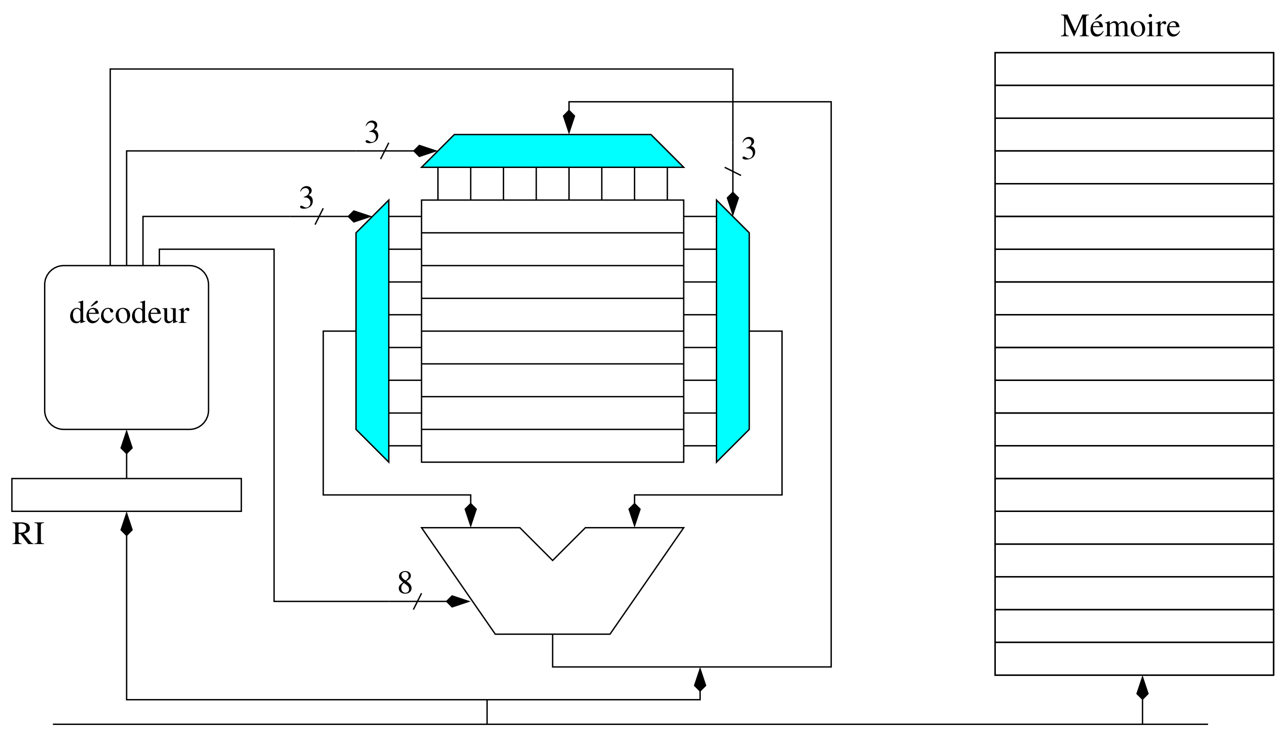
Program execution on a Processor (8 general purpose registers)



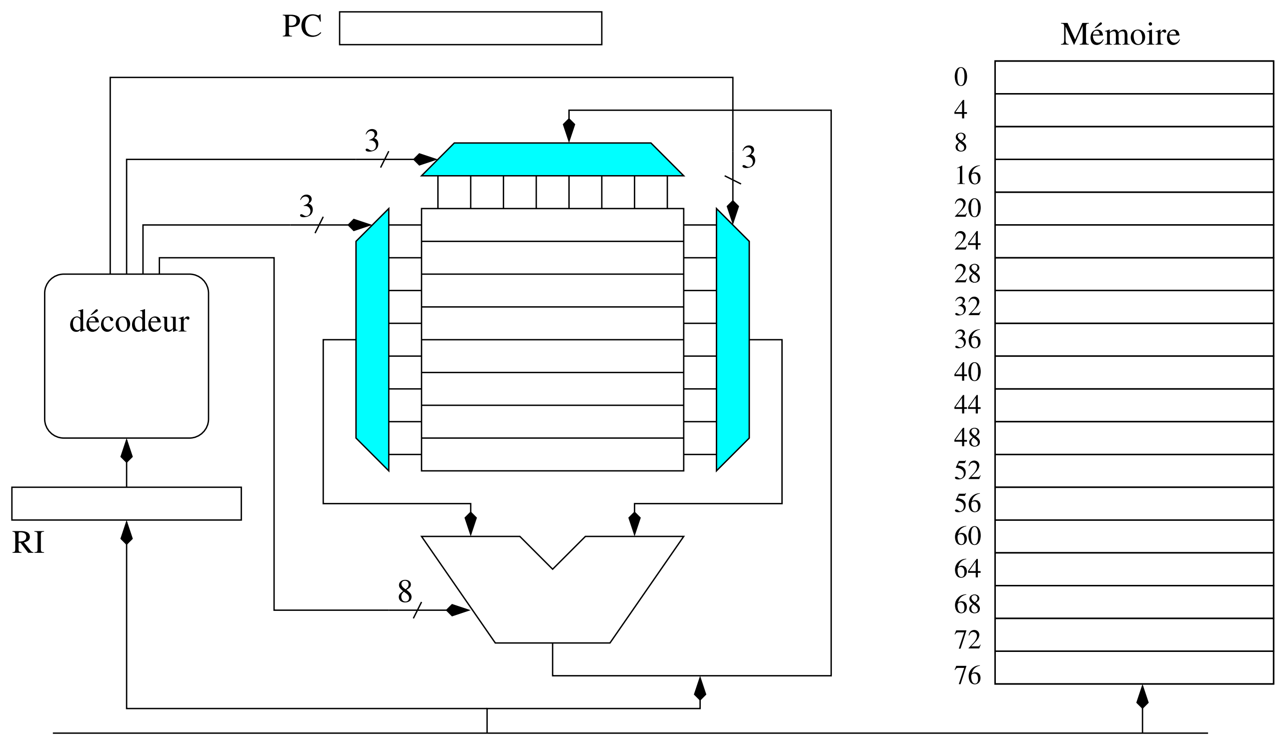
Program execution on a Processor (8 general purpose registers)



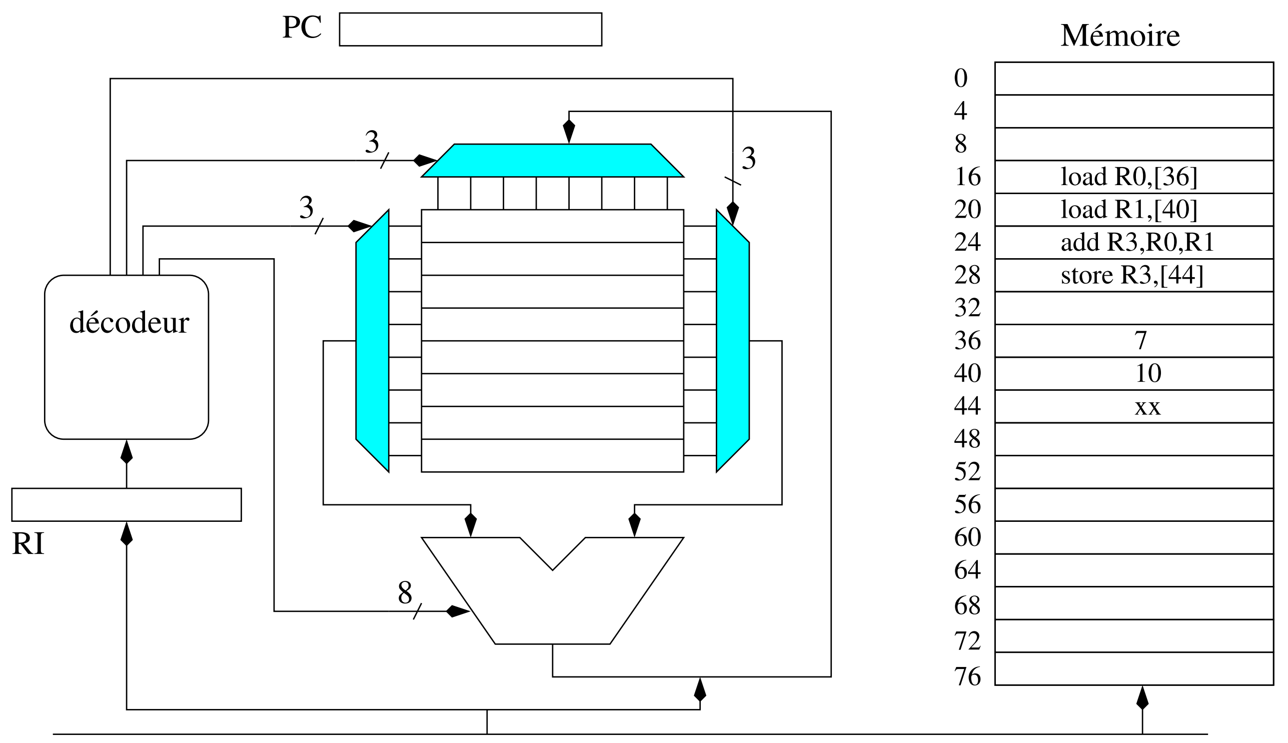
Program execution on a Processor (8 general purpose registers)



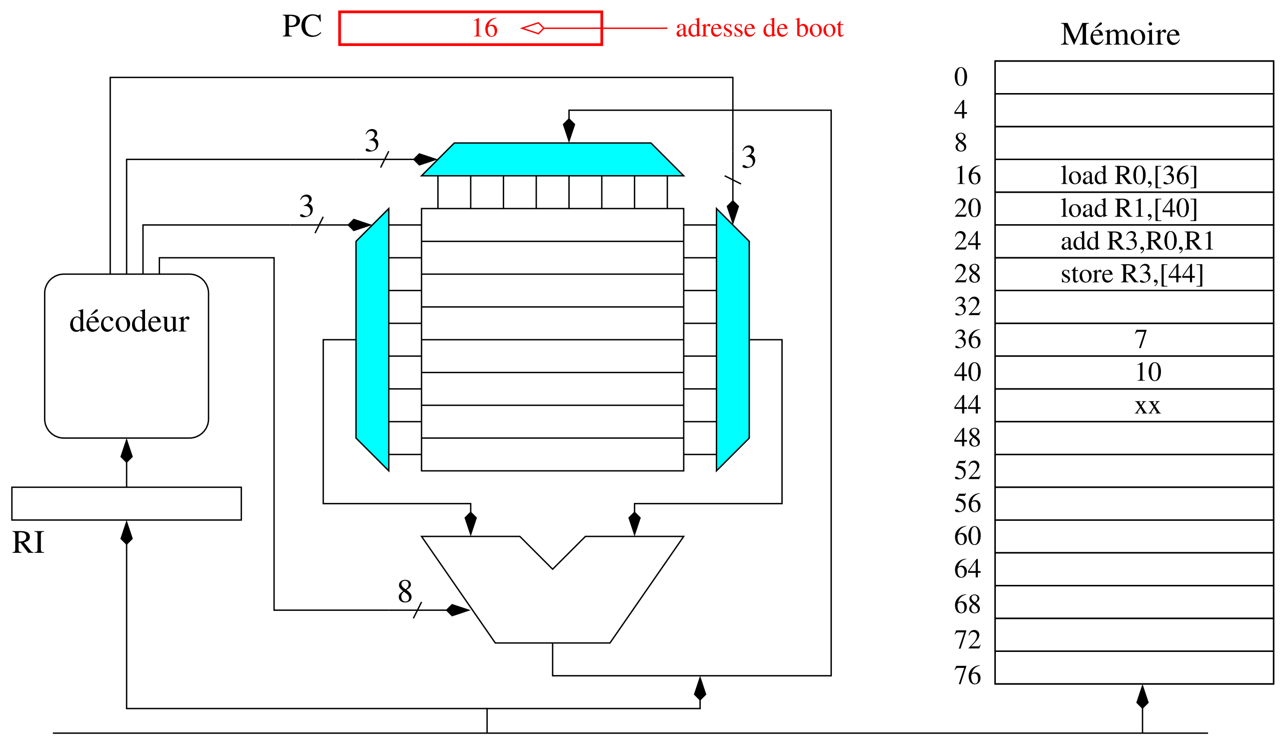
Program execution on a Processor (8 general purpose registers)



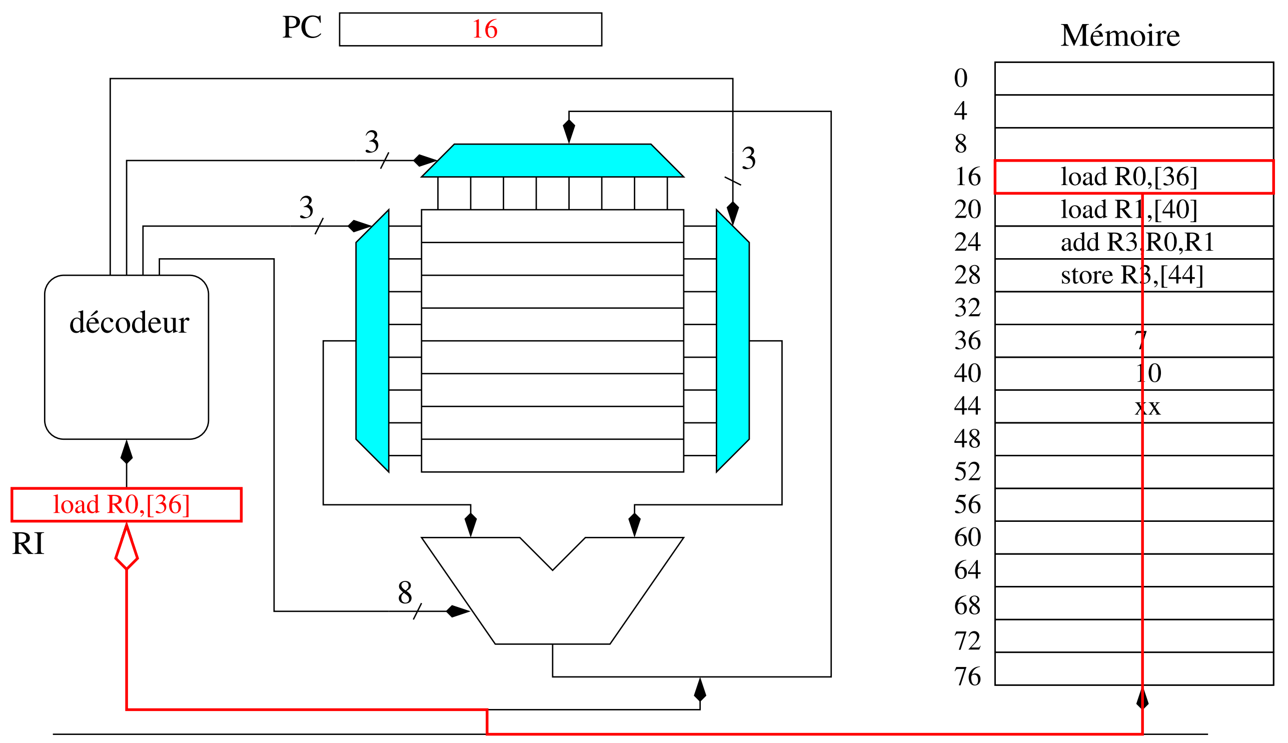
Program execution on a Processor (8 general purpose registers)



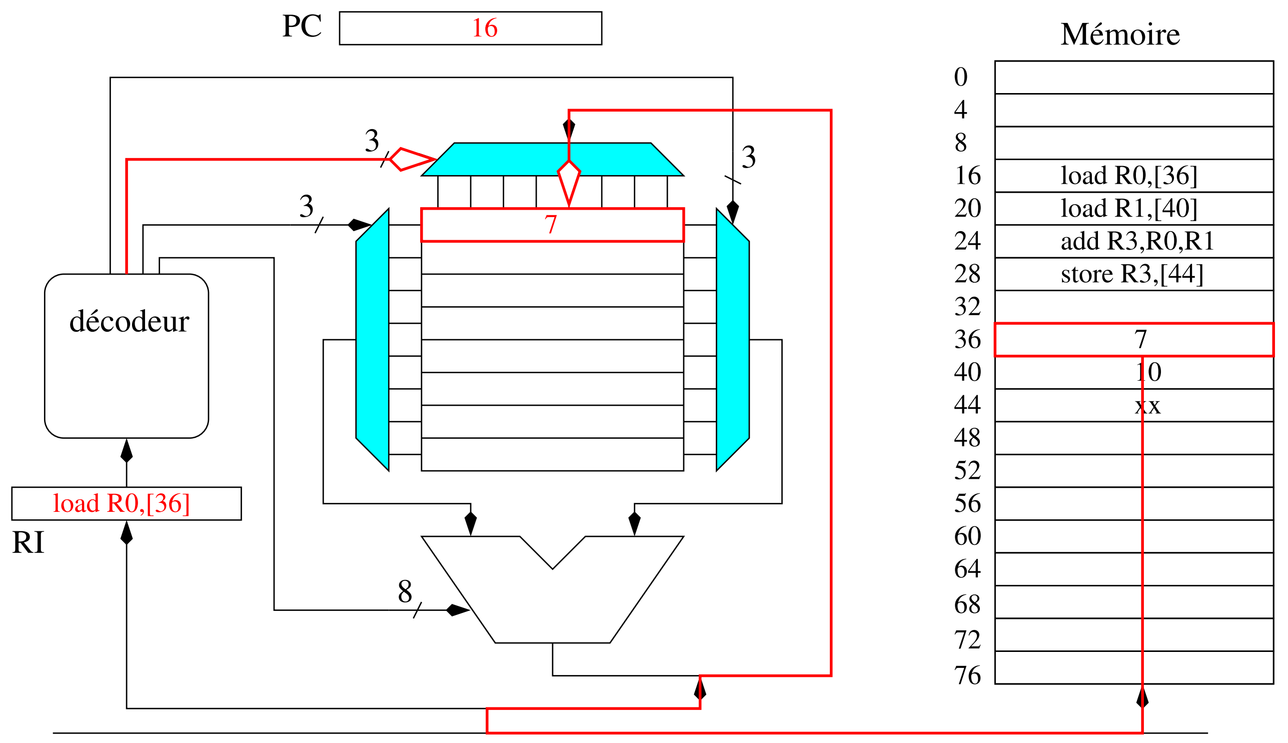
Program execution on a Processor (8 general purpose registers)



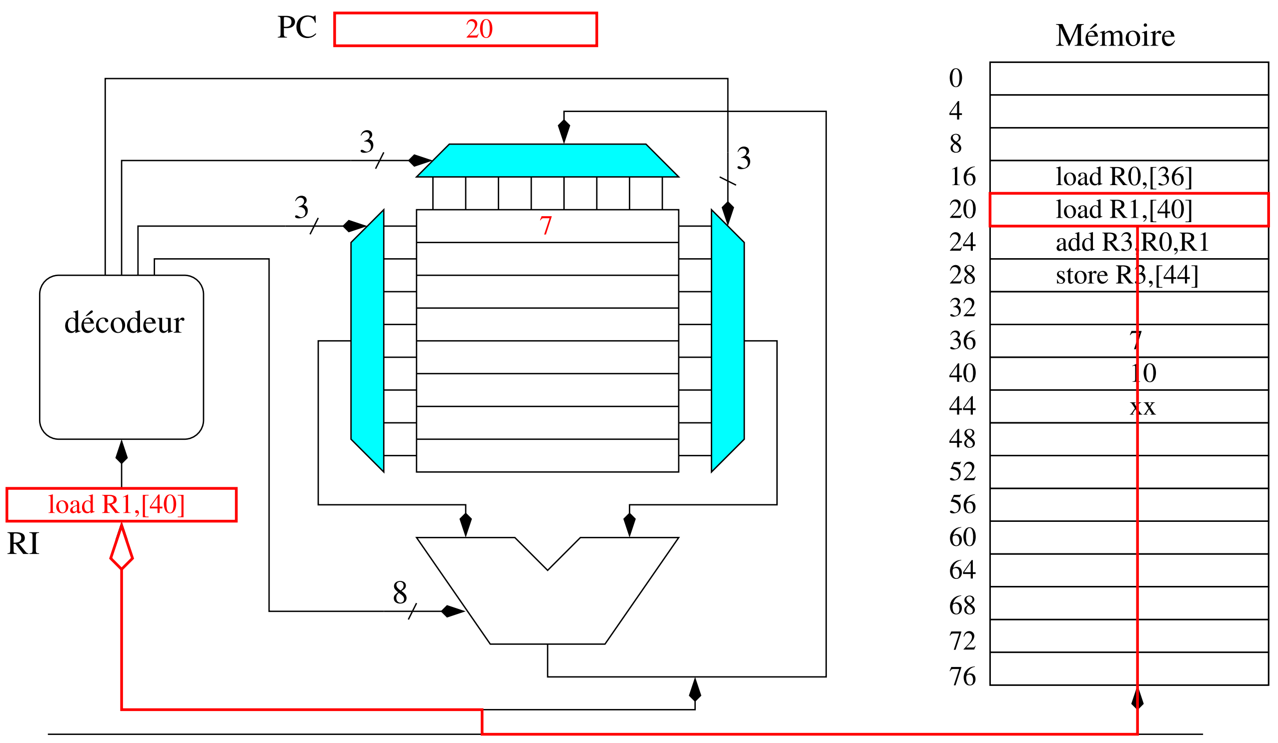
Program execution on a Processor (8 general purpose registers)



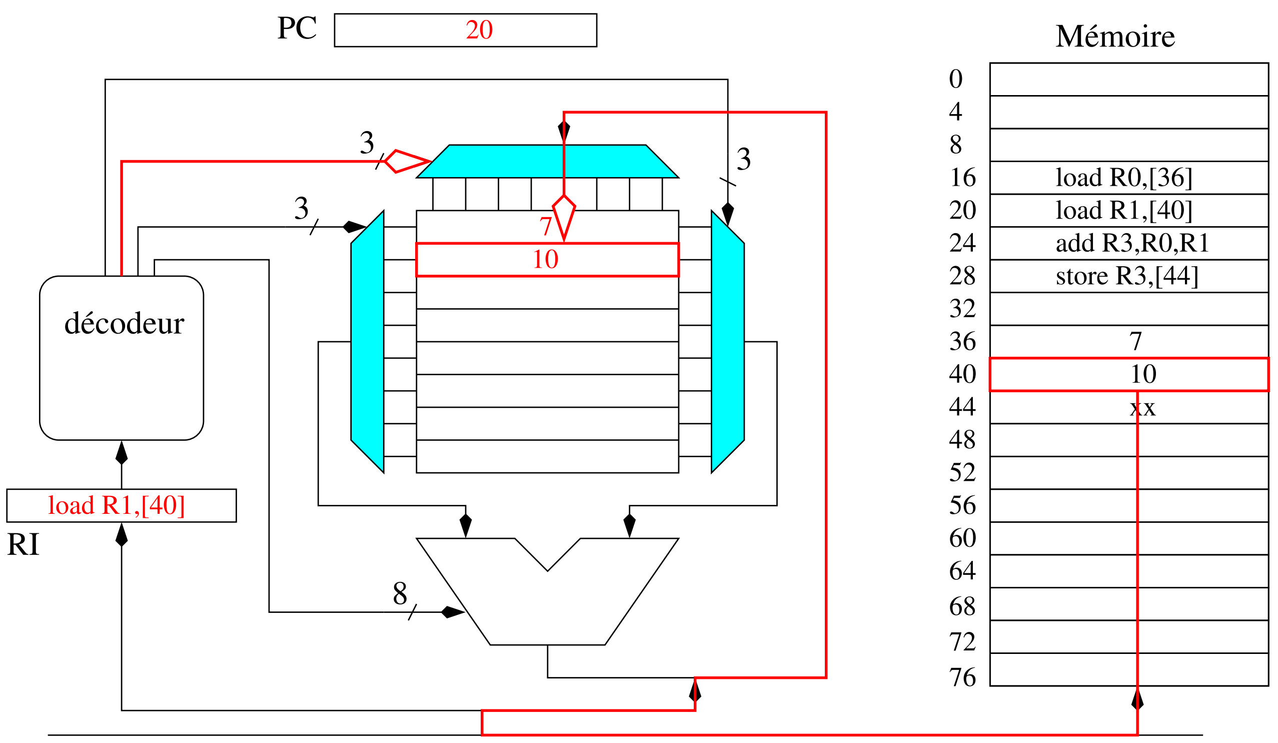
Program execution on a Processor (8 general purpose registers)



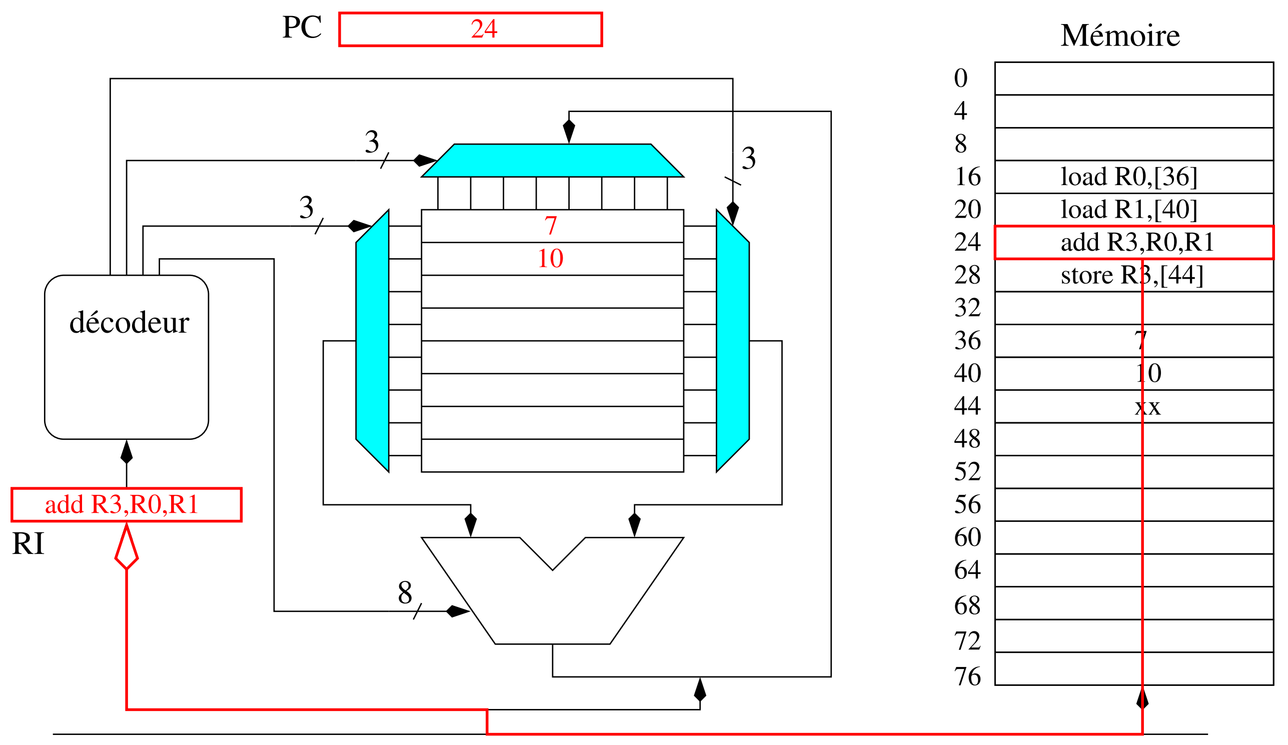
Program execution on a Processor (8 general purpose registers)



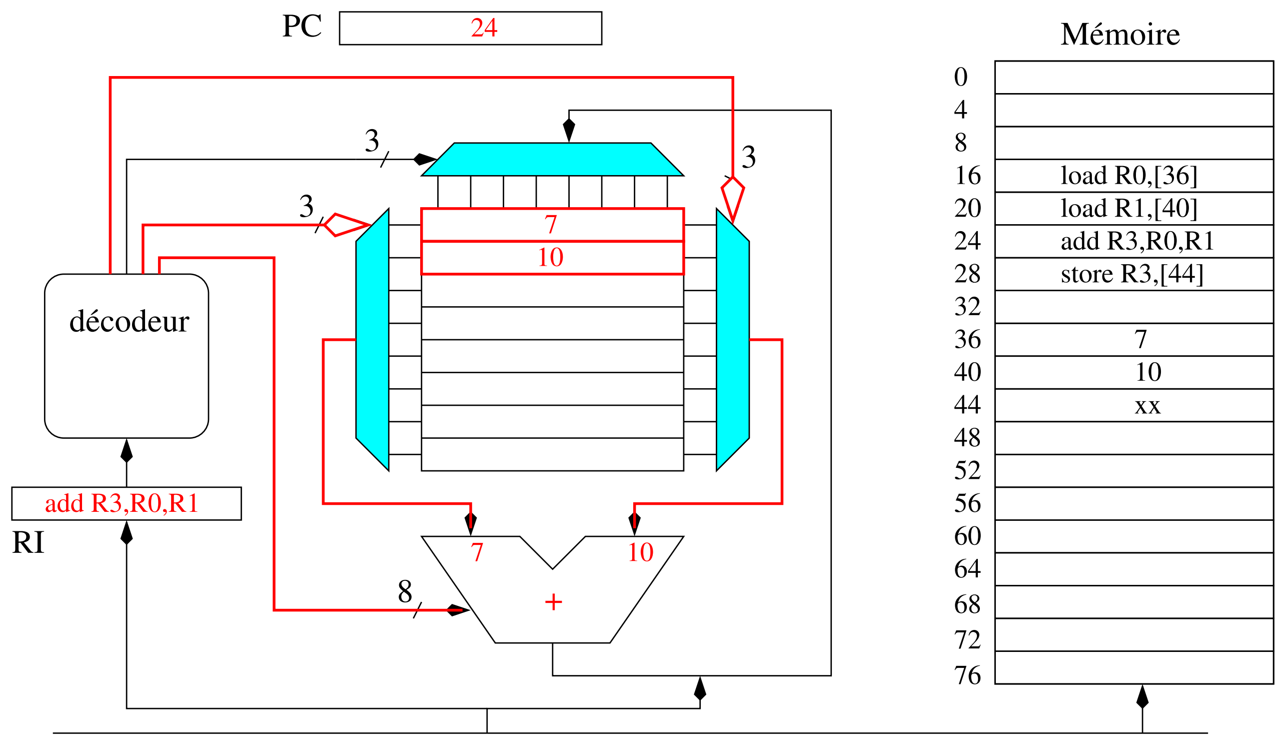
Program execution on a Processor (8 general purpose registers)



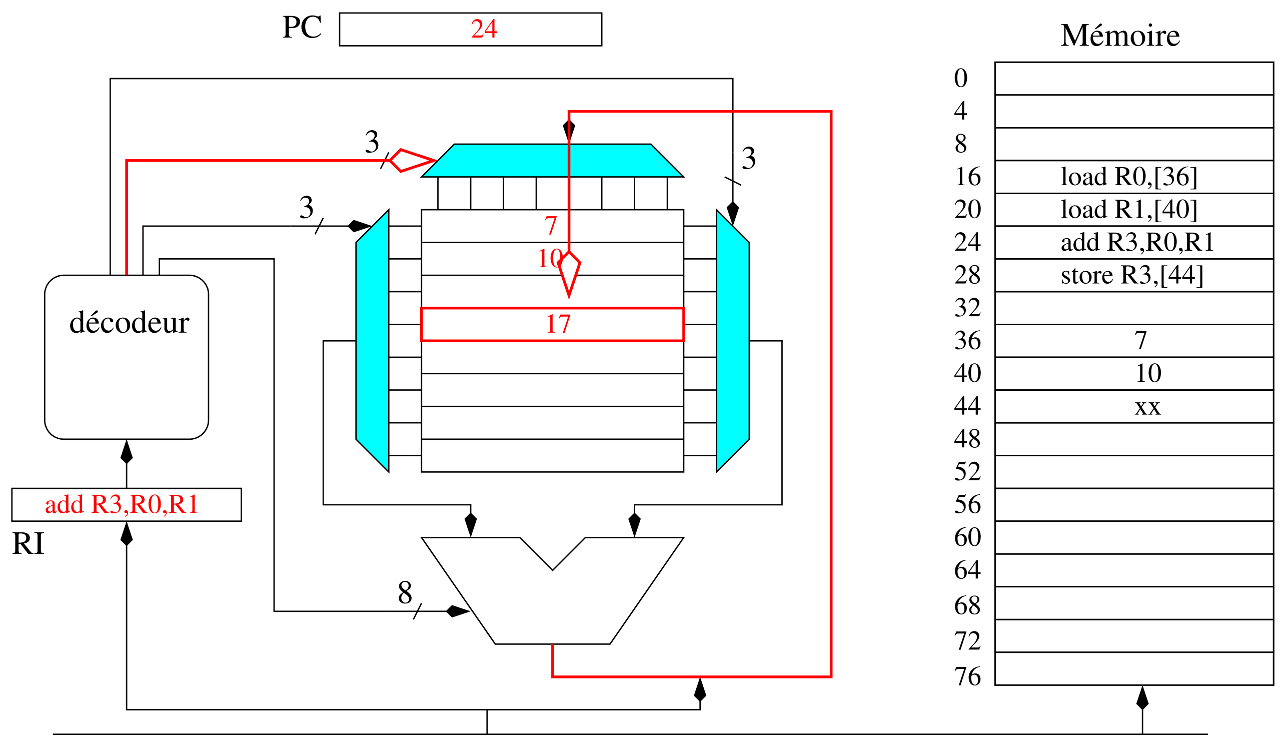
Program execution on a Processor (8 general purpose registers)



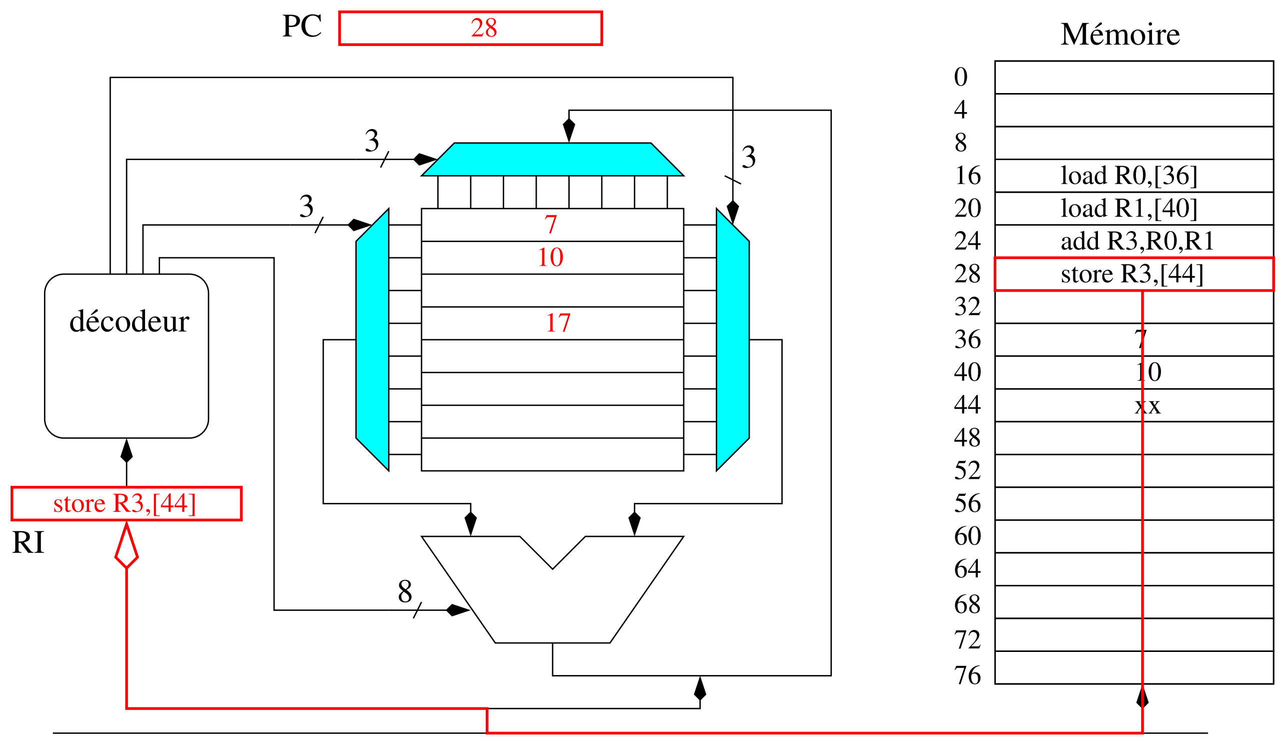
Program execution on a Processor (8 general purpose registers)



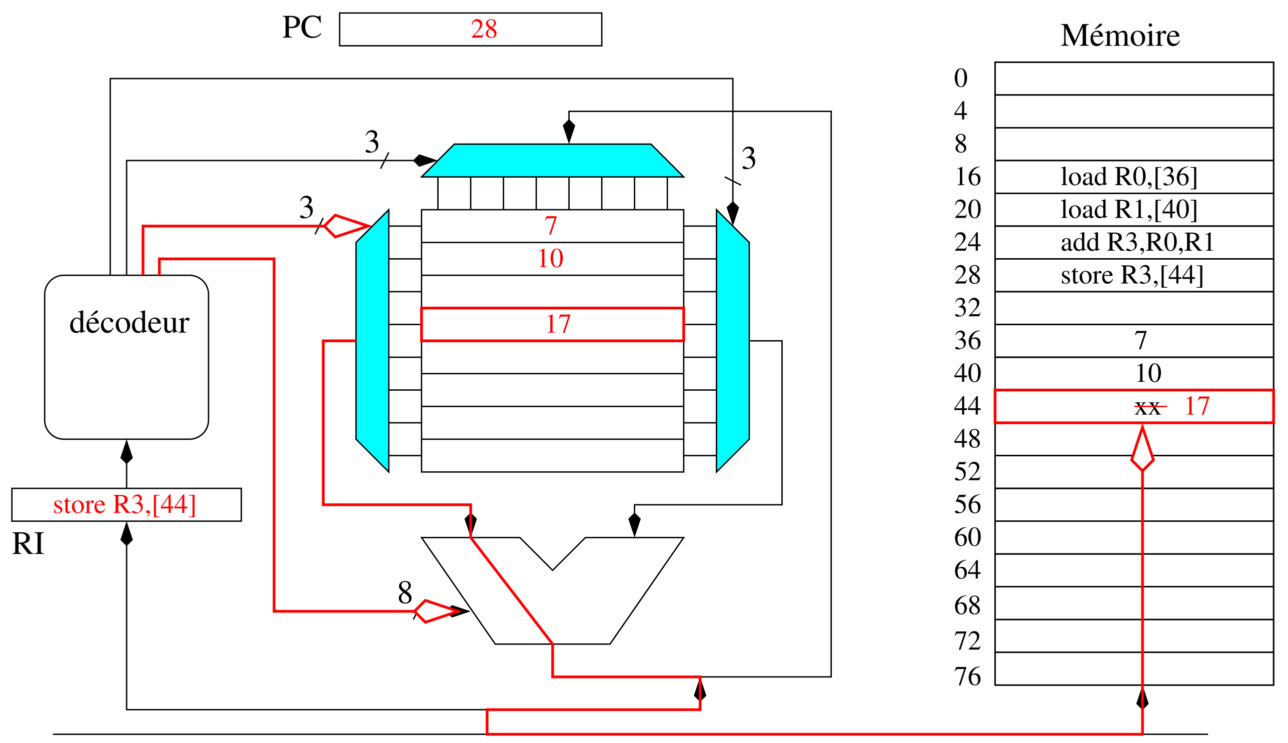
Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)

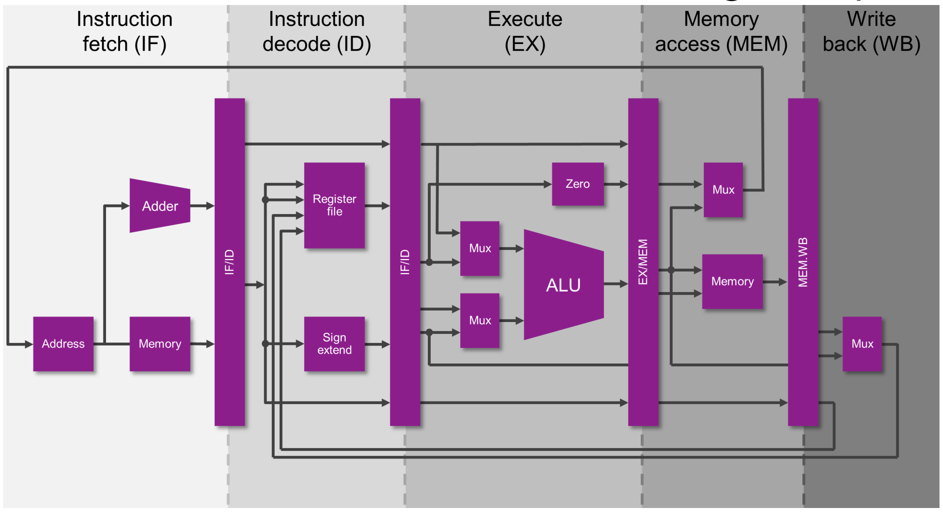


The “Von Neumann cycle”

- The so-called Von Neumann cycle is simply the decomposition of the execution of an instruction in several independent stages.
- The number of stages depend on the processor, usually 5 stages are commonly used as example:
  - **Instruction Fetch** (IF)
    - Reads the instruction from memory (at address $PC) and write it in $IR.
  - **Instruction Decode** (ID)
    - computes what needs to be computed before execution: jump address destination, access to register, etc.
  - **Execute** (EX)
    - executes the instruction: ALU computation if needed
  - **Memory Access** (MEM)
    - Loads (or stores) data from memory if needed
  - **Write Back** (WB)
    - Writes the result into the register file if needed

The MIPS example

- The RISC paradigm was invented by Berkeley and popularized by Henessy and Patterson in the book on MIPS
- MIPS stands for *Microprocessor without Interlocked Pipeline Stages* and propose and architecture to execute each stage independently



from MIPS website `https://www.mips.com/`

Christian Wolf's slides

- Use Christian Wolf slides for explaining MIPS instruction pipeline
- Here

example of MIPS pipeline CPU architecture

- Taken from Henessy/patterson book

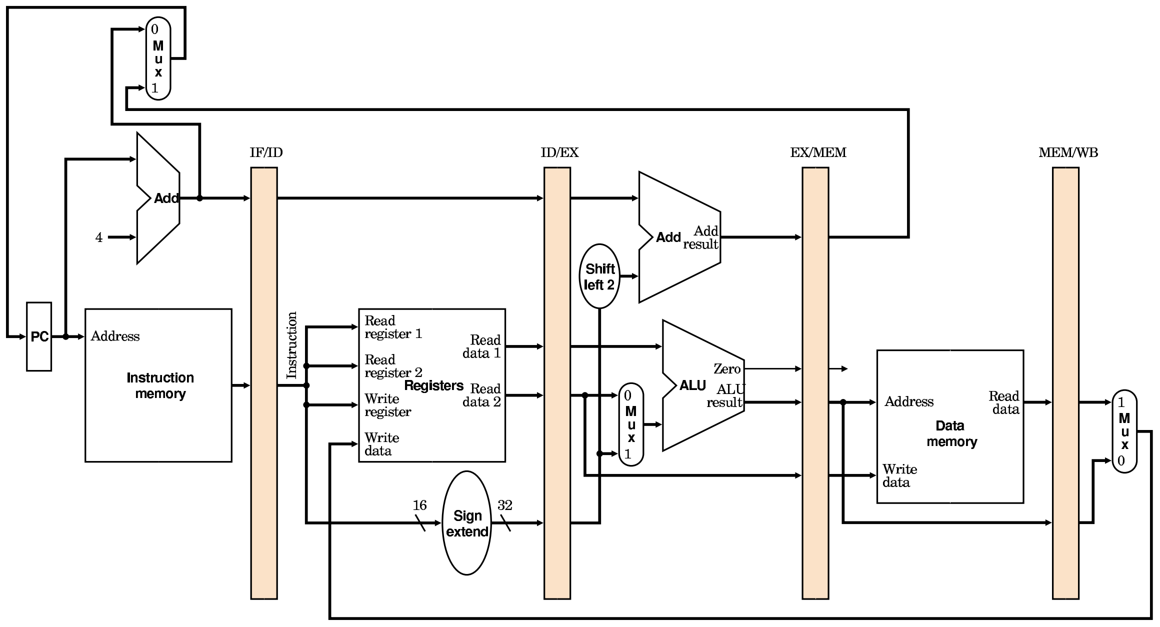
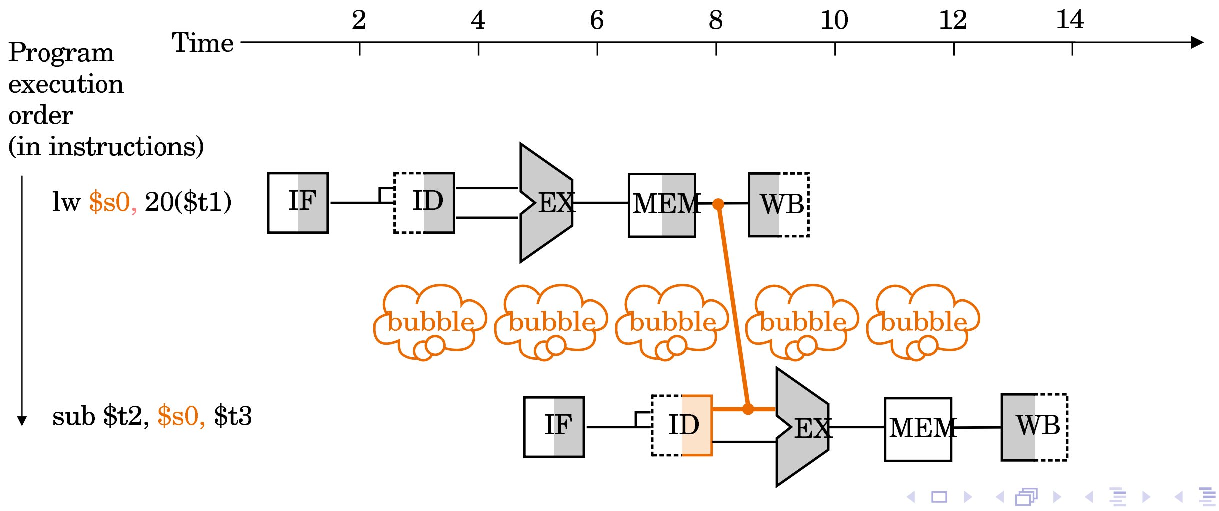


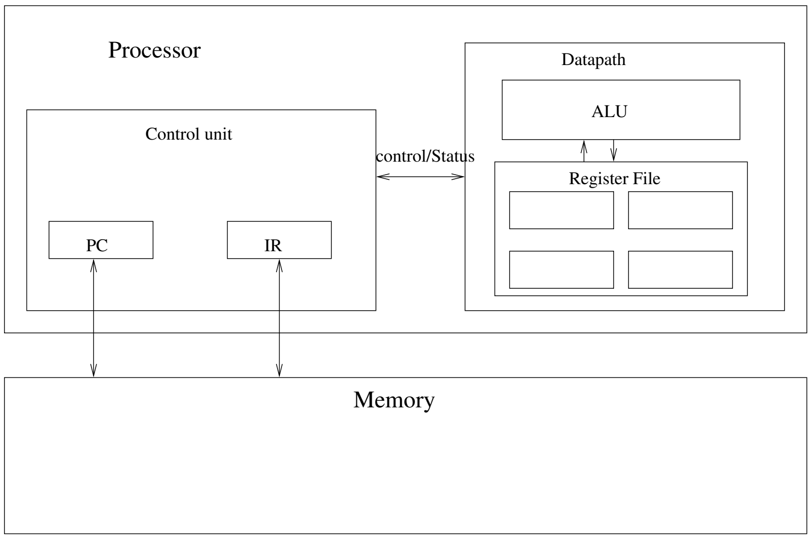
Illustration of bubble on MIPS

- When next instruction cannot be fetched directly (because it need the result of previous instruction for instance) it creates a “bubble”
- For instance: an addition using a register that was just loaded
- The value of the register will be available after the MEM stage of first instruction, hence we can delay on only on cycle, provided there is a *shortcut*.



Another illustration of instruction pipeline

- Go back to our previous representation of the processor and memory:
  - Von Neumann computer= Memory + CPU
  - CPU= = control Unit + Datapath
  - Datapath= ALU + Register file



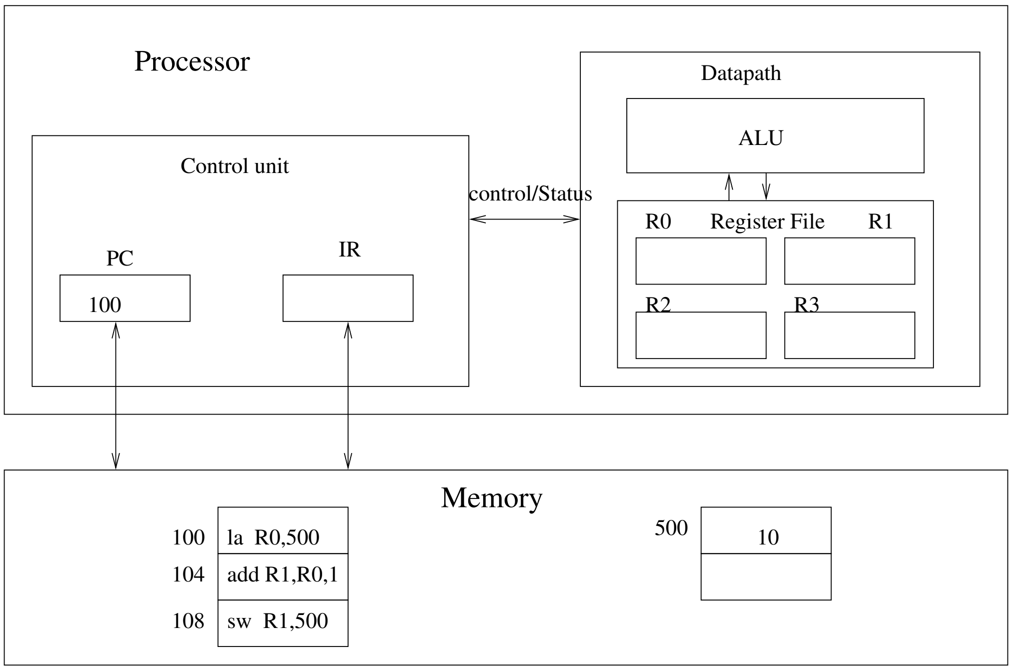
A pipeline example from MIPS

- Execute the sequence of assemby instruction:
  - load value at address 500 in register R0
  - Add 1 to R0 and put result in R1
  - store value of Register R1 at address 500
- (Think of `i=i+1`)
- Code:

```
la R0,500
add R1, R0, 1
sw  R1,500
```

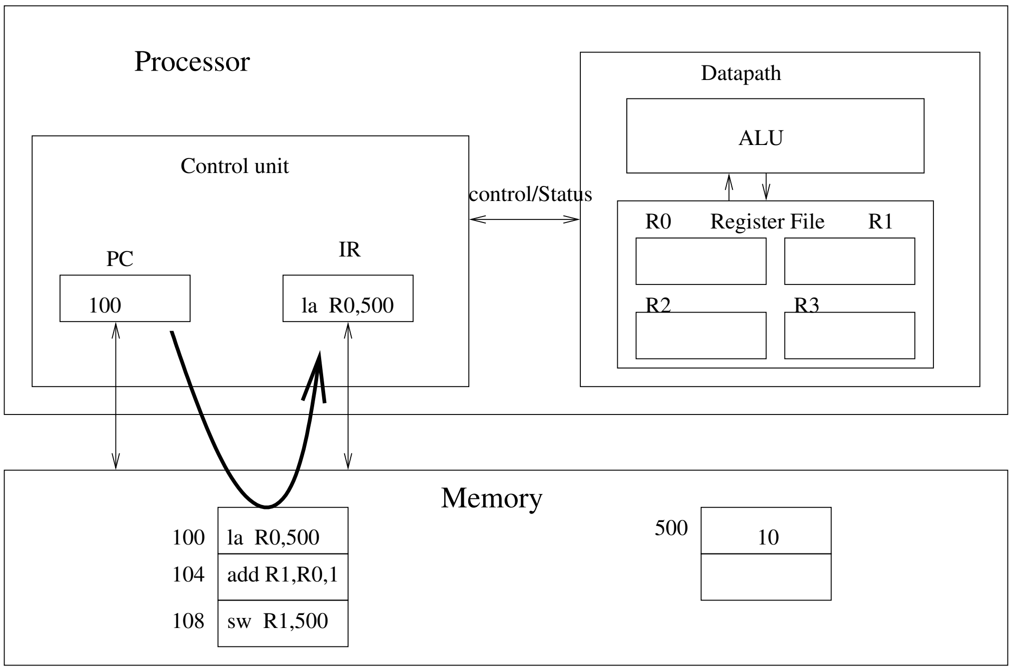
First possible execution: without pipeline

- Before execution starts, $PC contains the address of the first instruction: 100



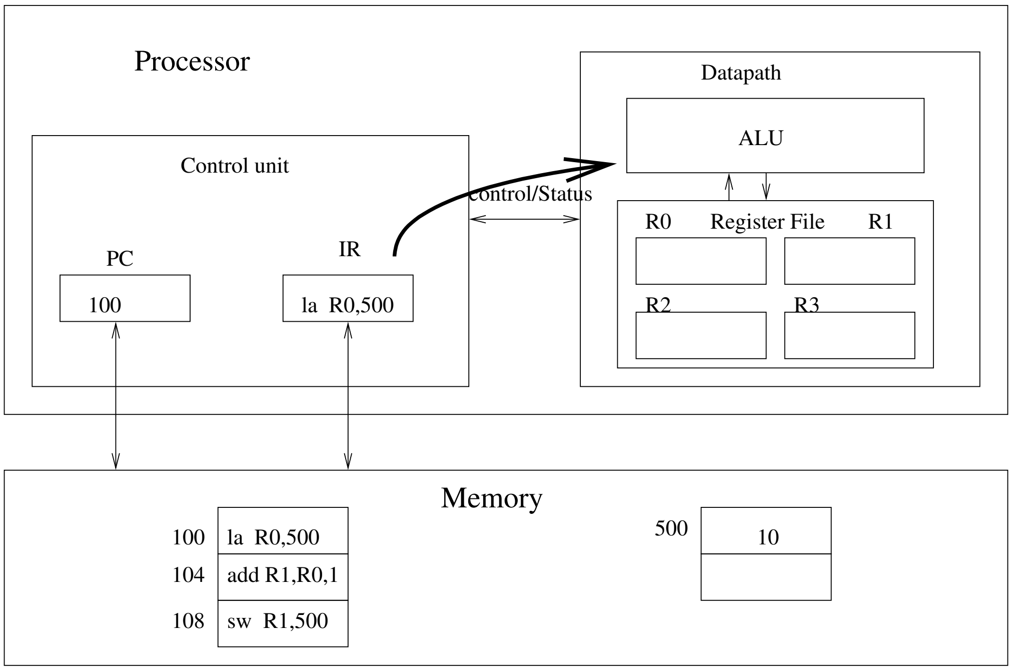
cycle 1

- Instruction Fetch



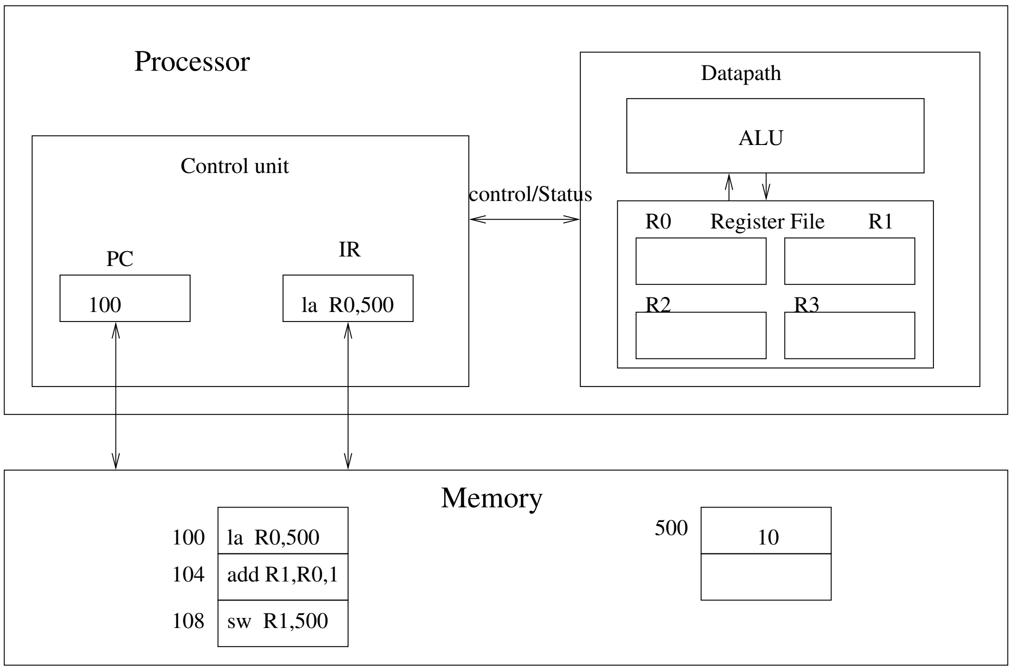
cycle 2

- Instruction Decode



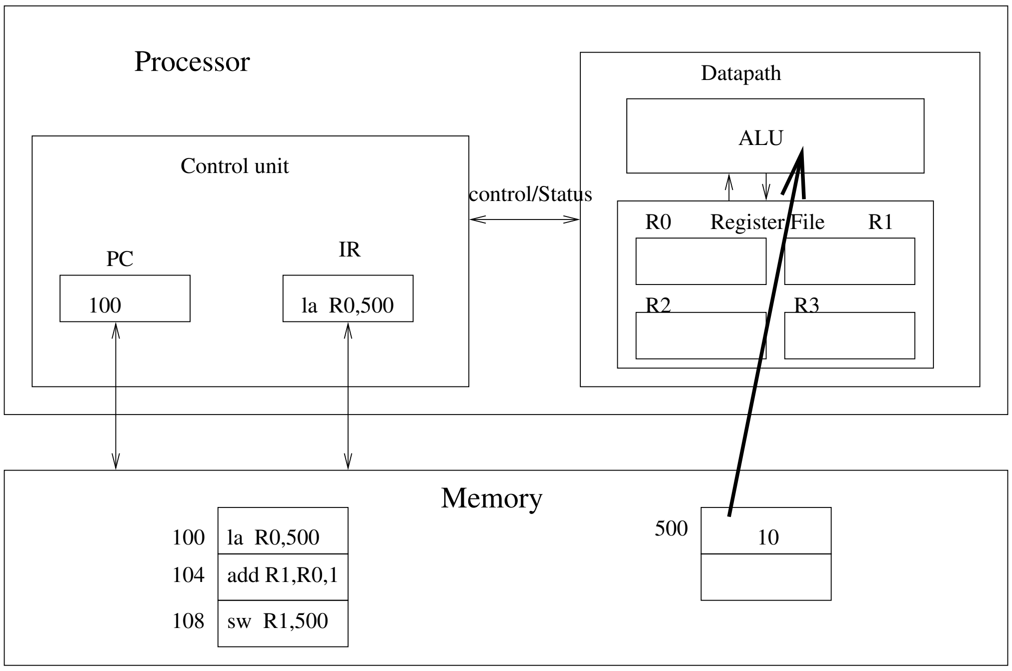
cycle 3

- Execute (nothing for load)



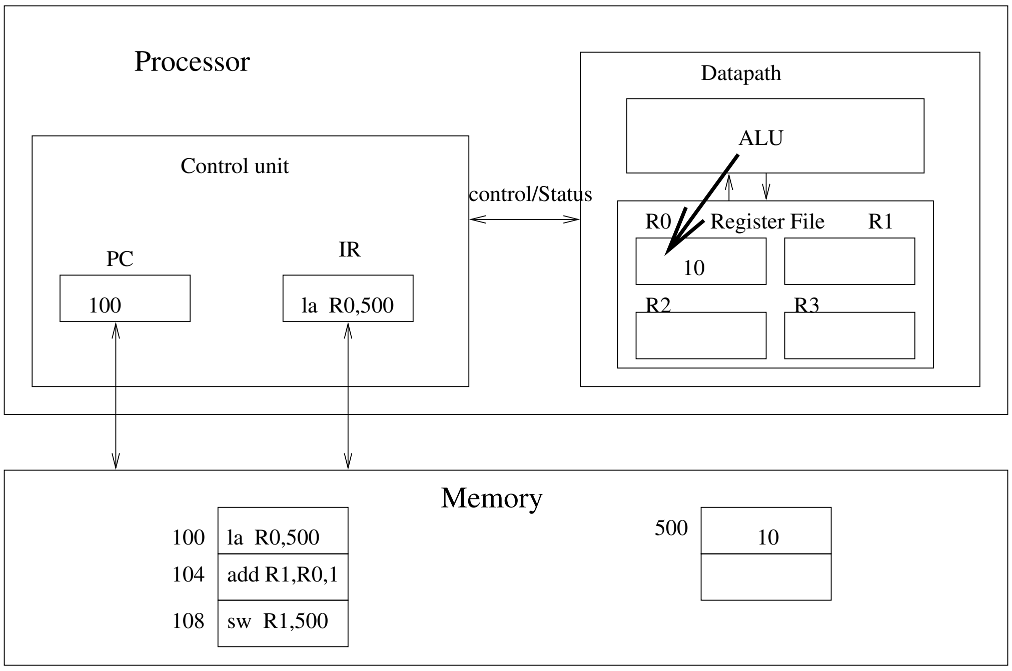
cycle 4

- Memory access



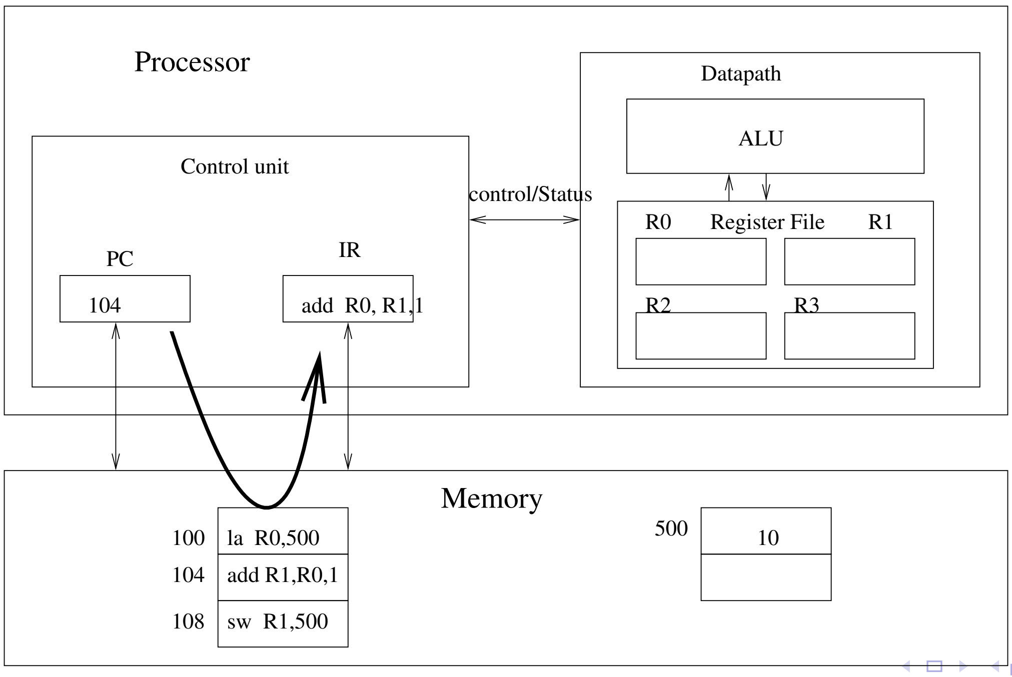
cycle 5

- Write Back



cycle 6

- increment $PC
- Fetch next instruction
- etc. etc.

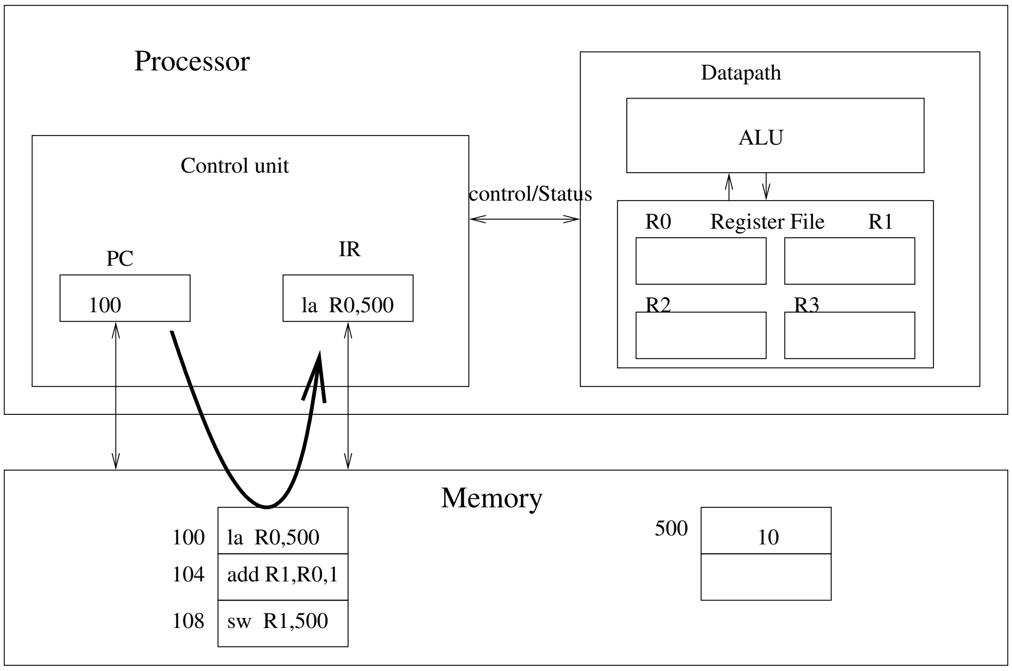


Counting CPI for non-pipelined architecture

- CPI= Cycle per instruction
- 5 cycles for executing on instruction
- ⇒ 15 cycles for 3 instructions.

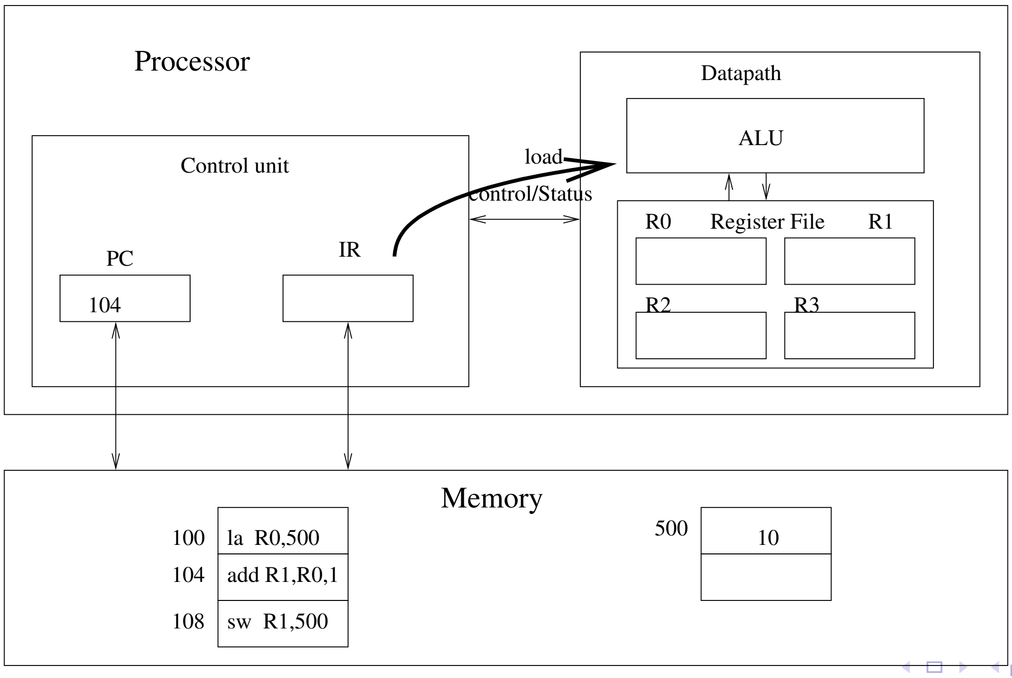
Example of pipelined execution

- Instruction Fetch (for 'load' instruction)



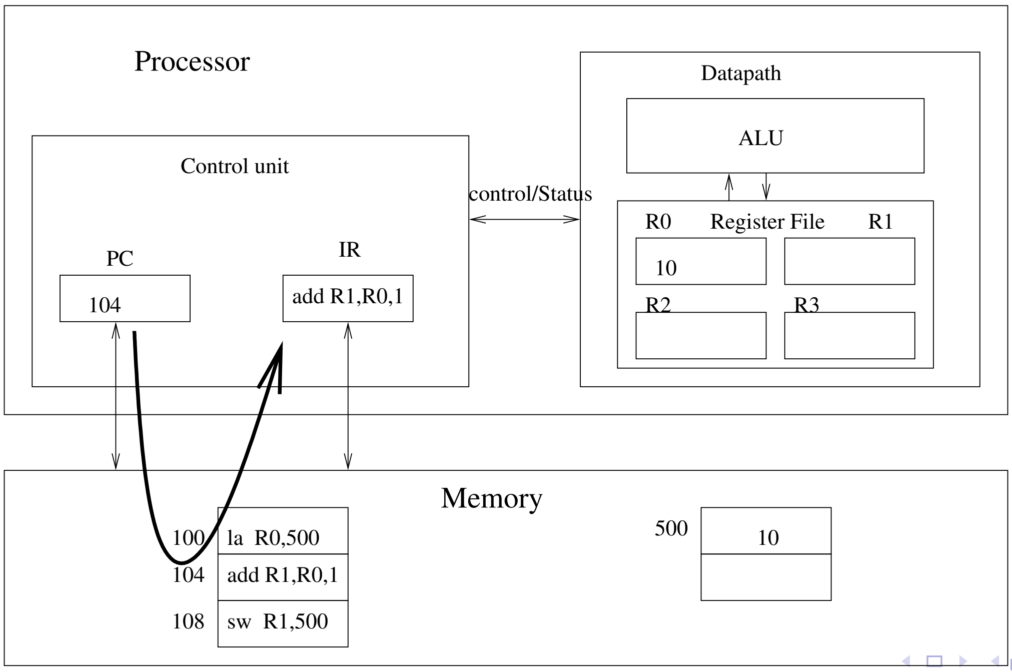
cycle 2

- Instruction Decode (for load)
- Instruction Fetch (for 'nothing' because of a bubble: instruction 'add' delayed)



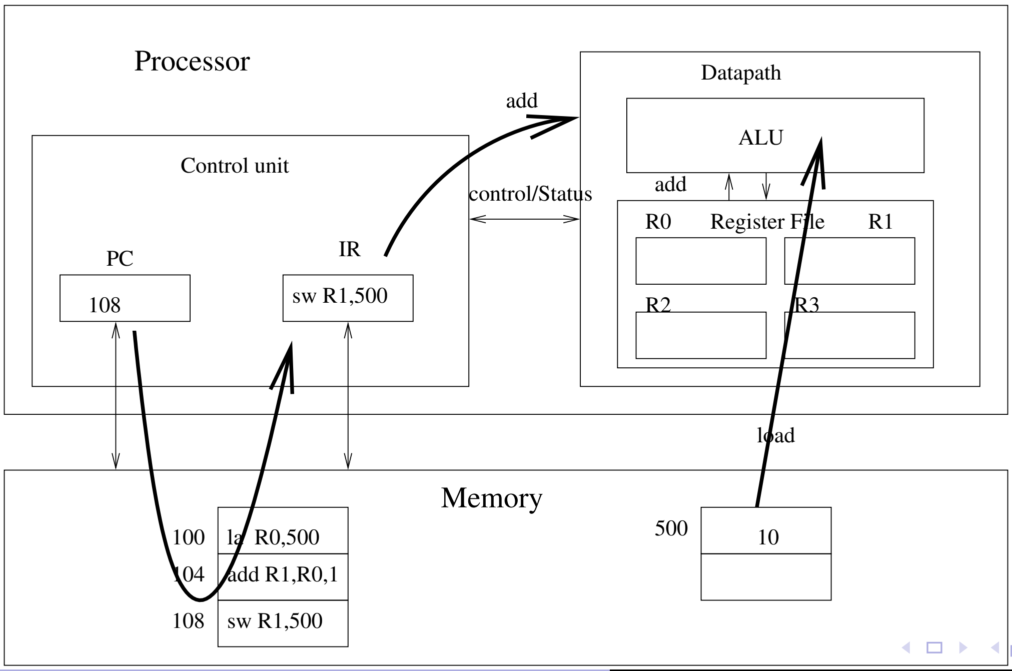
cycle 3

- Execute (for load: nothing to do)
- Instruction Decode (for 'nothing')
- Instruction fetch (for 'add')



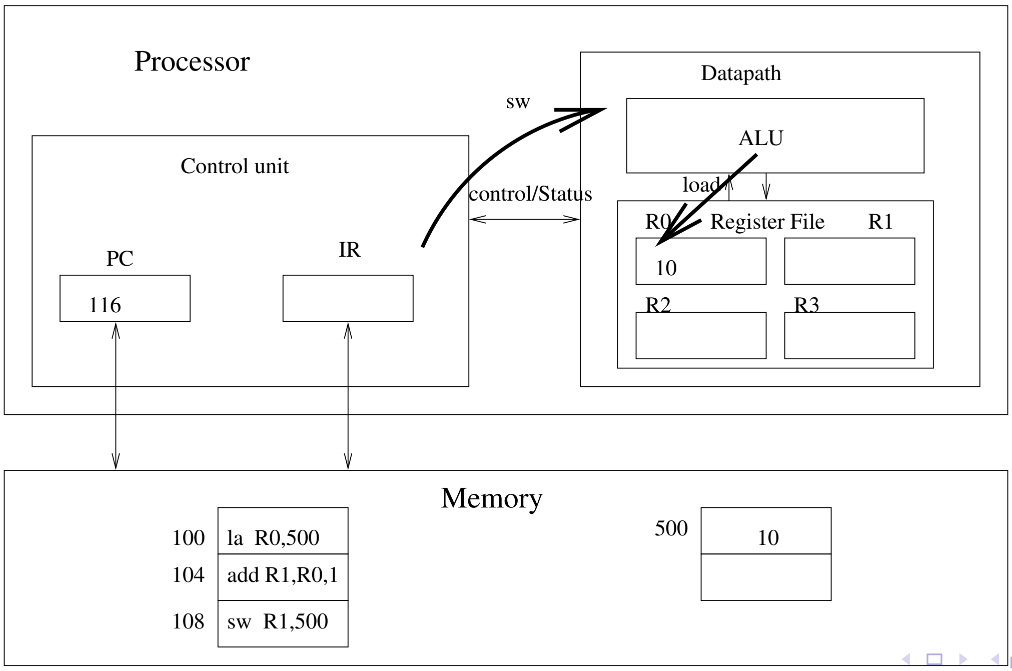
cycle 4

- Memory access (for load)
- Execute (for 'nothing')
- Instruction Decode (for add)
- Instruction fetch (for store)



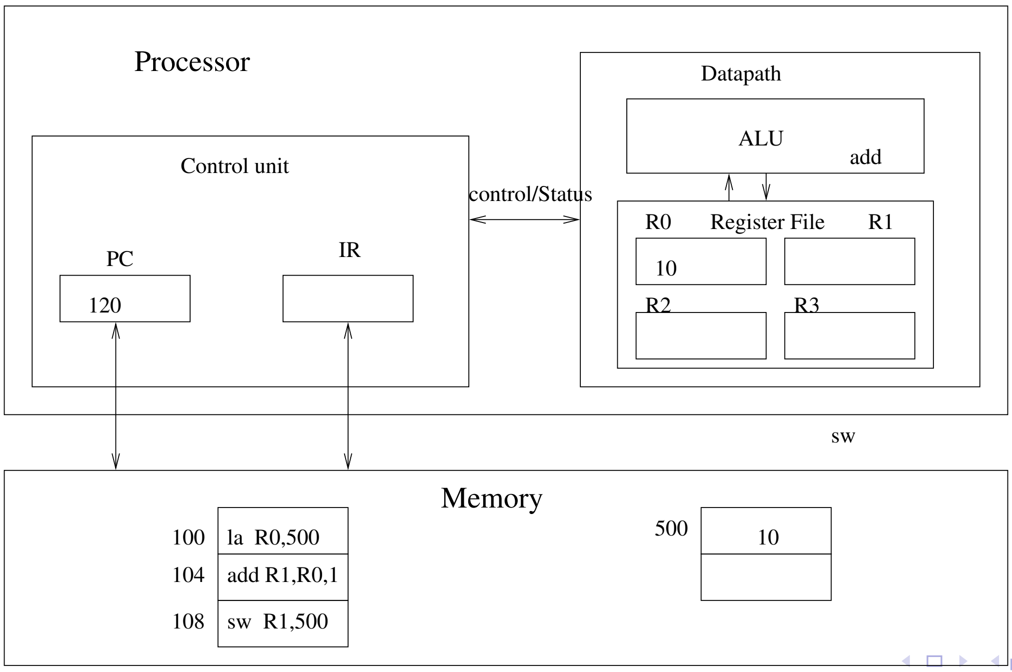
cycle 5

- Write Back (instruction load)
- Memory access (for 'nothing')
- Execute (instruction add: bypass)
- Instruction Decode store



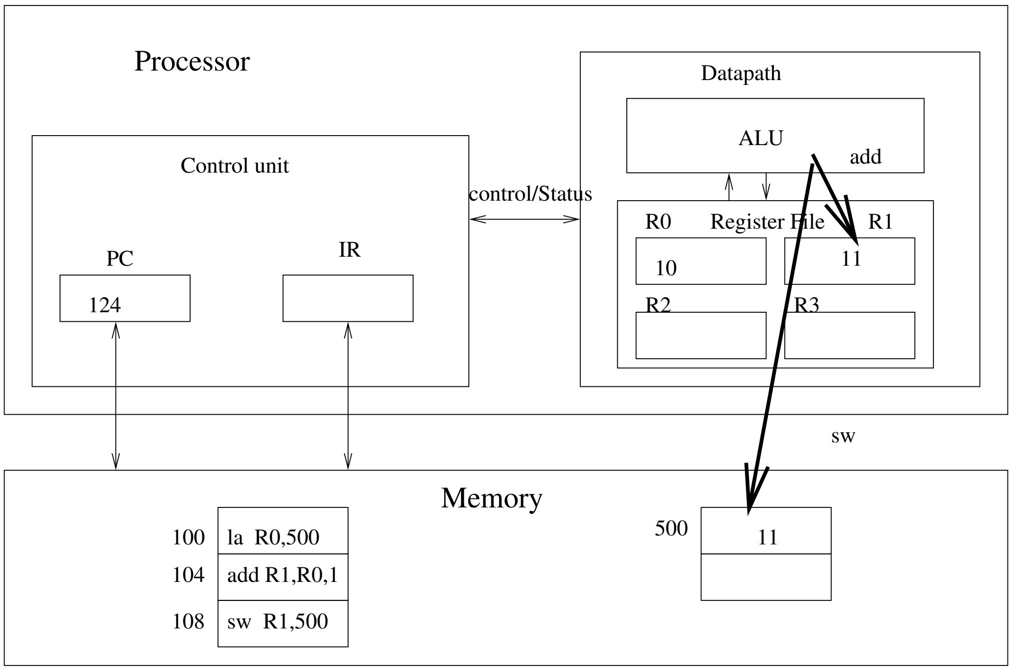
cycle 6

- Write Back (for 'nothing')
- Memory access (instruction add, nothing to do)
- Execute (instruction store: nothing to do)



cycle 7

- Write Back (instruction add)
- Memory access (instruction store: bypass)



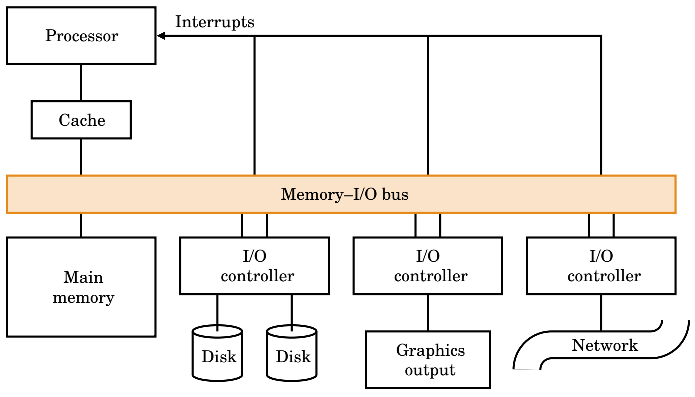
Counting CPI for both architectures

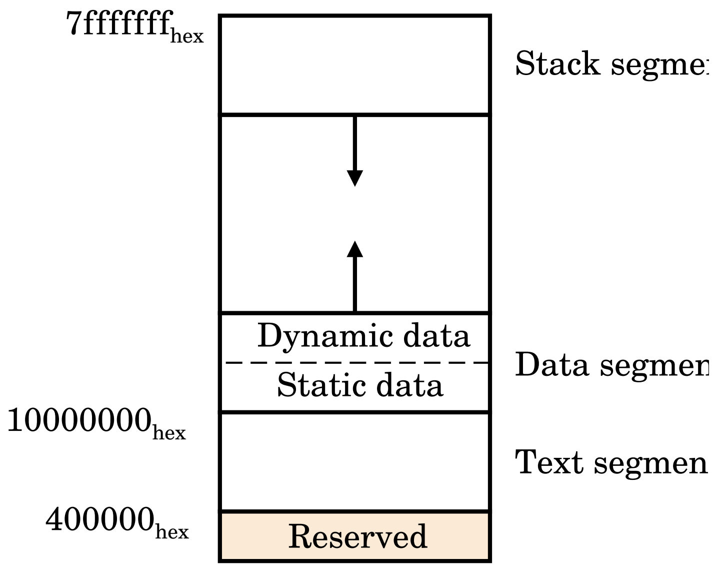
- Non-pipelined architecture:
  - 5 cycles for one instruction
  - ⇒ 15 cycles for 3 instructions.
- Pipelined architecture:
  - 5 cycles for one instruction
  - 8 cycles for 3 instructions.
  - ⇒ without bubbles, one instruction per cycle
  - A 'jump' instruction interrupt the pipeline (need to wait for the address decoding to fetch next instruction) ⇒ *pipeline stall*
  - Some ISA allow to use these *delay slots*: one or two instruction *after* the jump are executed before the jump occurs.

Du langage à l'exécution

-

Rappels d'architecture



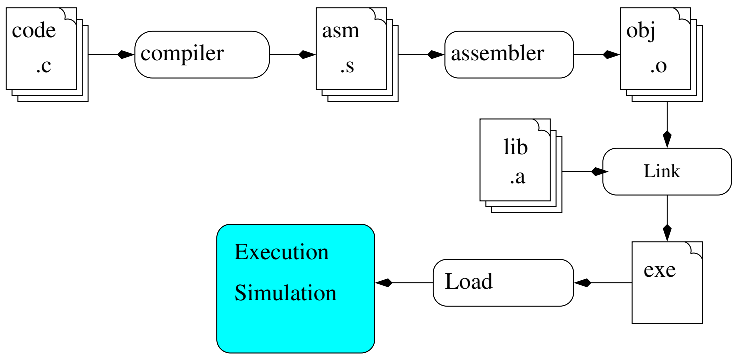


Architecture view from the programmer

- Modern systems allow
  - To run multiple independent programs in parallel (process)
  - To access memory space larger than physical memory available (virtual memory)
- For the programmer: all this is transparent
  - Only one program runs with very large memory available
- The processor view memory contains:
  - The code to execute
  - Static data (size known at compile time)
  - Dynamic data (size known at runtime: the heap, and the space needed for the execution itself: the battery)
- The programmer sees only the data (static and dynamic)

compilation process

- the complete process will translate a C program into code executable (loading and execution will take place later).



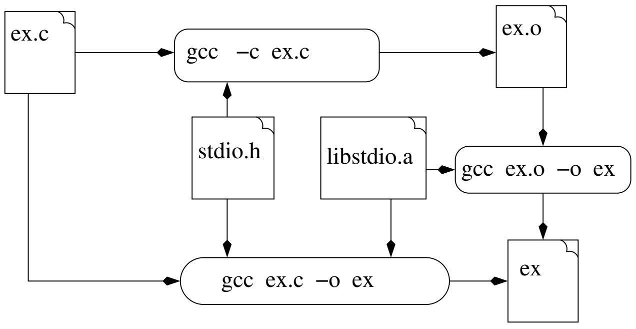
- We often call *compilation* the set compiler + assembler
- The gcc compiler also includes an assembler and linking process (accessible by options)

Your compilation process

- 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`

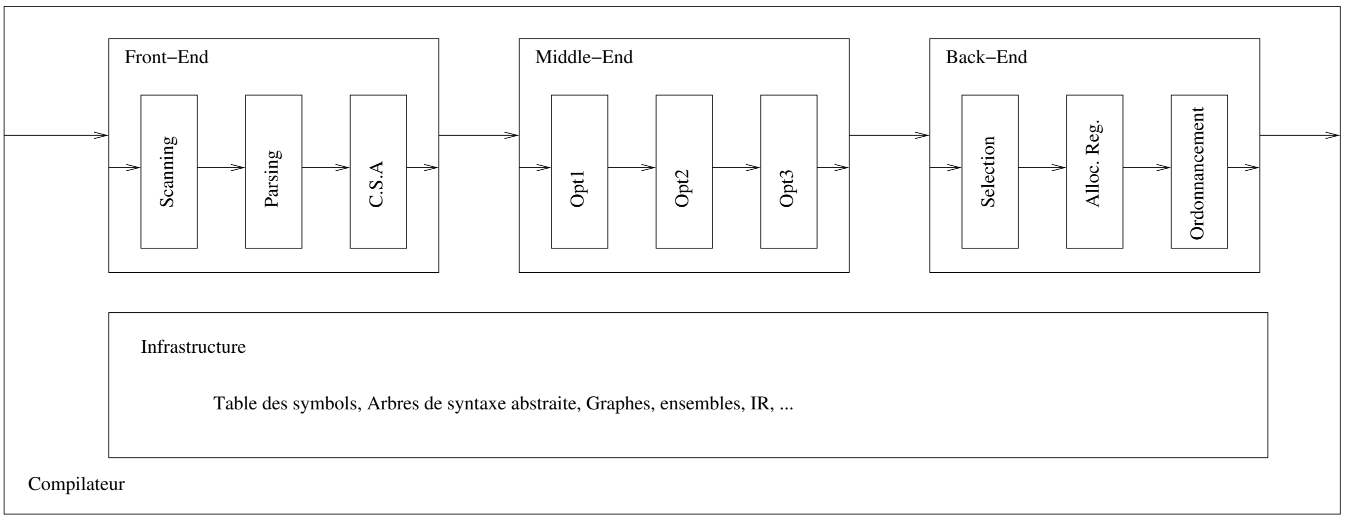
content of `ex.c`

```
#include <stdio.h>

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

  return(0);
}
```



Zooming on "compilation"

- The compilation process is divided in 3 phases:

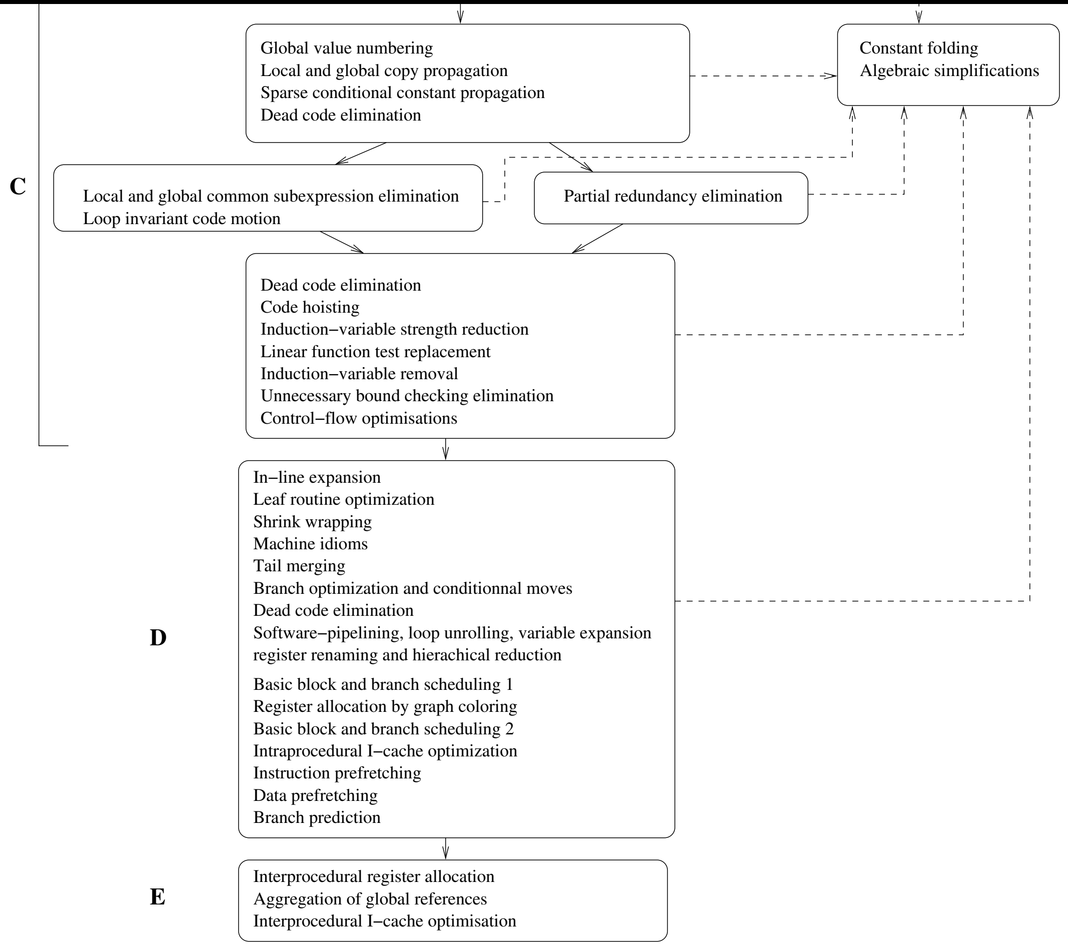


Compilation: the front-end

- The front end of an embedded code compiler uses the same techniques as traditional compilers (we can want to include assembler parts directly)
- Parsing LR(1): the parser is usually generated with dedicated *metacompilation* tools such as `Flex` et `bison` for GNU

Compilation: The middle-end

- Some phases of optimizations are added, they can be very calculative
- Some example of optimisation independent of the target machine architecturre
  - Elimination of redundant expressions
  - dead code elimination
  - constant propagation
- Warning: optimization can hinder the understanding of the assembler (use the -O0 options with tt gcc)


Global value numbering
Local and global copy propagation
Sparse conditional constant propagation
Dead code elimination
Constant folding
Algebraic simplifications
C
Local and global common subexpression elimination
Loop invariant code motion
Partial redundancy elimination
Dead code elimination
Code hoisting
Induction−variable strength reduction
Linear function test replacement
Induction−variable removal
Unnecessary bound checking elimination
Control−flow optimisations
In−line expansion
Leaf routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimization and conditionnal moves
Dead code elimination
D
Software−pipelining, loop unrolling, variable expansion
register renaming and hierachical reduction
Basic block and branch scheduling 1
Register allocation by graph coloring
Basic block and branch scheduling 2
Intraprocedural I−cache optimization
Instruction prefretching
Data prefretching
Branch prediction
E
Interprocedural register allocation
Aggregation of global references
Interprocedural I−cache optimisation

Compilation: The back-end

- The code generation phase is dedicated to architecture target. Retargetable compilation techniques are used for architectural families.
- The most important steps important are:
  - Code selection
  - Register allocation
  - instruction scheduling

GCC

- The gcc command runs several program depending on the options
  - The pre-processer cpp
  - The compiler `cc1`
  - The assembleur `gas`
  - The Linker `ld`
- `gcc -v` allow to visualize the different programs called by gcc

The pre-processer cpp or gcc -E

- the task of the pre-processor are :
  - elimination of comments,
  - inclusion of source files
  - macro substitution (#define)
  - conditionnal compilation.
- Example:

ex1.c

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
...
f=MAX(3,b);
```

ex1.i

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
...
f=((3) > (b) ? (3) : (b));
```

The compiler cc1 or gcc -S

- generate assembly code
- `gcc -S main.c -o main.S`
- Exemple :

```
void main()
{ int i;
  i=0;

  while (1)
   {
     i++;
     nop();
   }
}
```

Assembly code generated (for MSP430)

```
mov    #2558,  SP          ; stack initialization de la pile
mov    r1,  r4             ; r4 <- SP
mov    #0,  0(r4)          ; i initialization
inc    0(r4)               ; i++
nop                        ; nop();
jmp    $-6                 ; unnconditionnal jump (PC-6):
incd    SP                 ;
br    #0x1158               ;
```

Assembly code produce by mspgcc -S

```
        .text
        .p2align 1,0
.global         main
        .type         main,@function
main:
/* prologue: frame size = 2 */
.L__FrameSize_main=0x2
.L__FrameOffset_main=0x6
        mov      #(__stack-2), r1
        mov      r1,r4
/* prologue end (size=3) */
        mov      #llo(0), @r4
.L2:
        add      #llo(1), @r4
        nop
        jmp      .L2
/* epilogue: frame size=2 */
        add      #2, r1
        br       #__stop_progExec__
/* epilogue end (size=3) */
/* function main size 14 (8) */
```

Assembler as ou gas

- transform an assembly code into object code (binaire representation of symbolic assembly code)
- Option -c of gcc allow to conbine compilation et assembly
  `gcc -c main.c -o main.o`

Linking: `ld`

- Produce the executable (`a.out` by default) from object code of programs and library used
- There are two ways to use libraries in a program
  - Dynamic or shared libraries (default option): the code of the library is not included in the executable, the system dynamically loads the code of the library in memory when calling the program. We need than only *one* version of the library in memory even if several programs use the same library. The library must be em installed on the machine, before running the code.
  - Static libraries: the code of the library is included in the executable. The executable file is bigger but you can run it on a machine on which the library is not installed.

Binary file manipulation

Some usefull command:

- `nm`
  Allow to know symboles (i.e. label: function names) used in an object file or executable
  ```
  trisset@hom\$ nm fib.elf | grep main
  000040c8 T main
  ```
- `objdump` allow to anlayze a binary file. For instance it can get correspondance between binary representation and assembly code
  ```
  trisset@hom$ objdump -f fib
  fib:     file format elf32-msp430
  architecture: msp:43, flags 0x00000112:
  EXEC_P, HAS_SYMS, D_PAGED
  start address 0x00001100
  ```