

Architecture des ordinateurs

Frédéric Goualard

Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241
Bureau 208



- ▶ Ressources sur *madoc*
- ▶ 9 cours magistraux d'1h20
- ▶ 14 séances de travaux dirigés d'1h20
(+ contrôle continu)
- ▶ 16 heures de travaux pratiques (créneaux de 2h/1h20)
(dont 6h de projet)
 - ▶ C/C++
 - ▶ Circuits logiques ([logisim](#))
 - ▶ Assembleur MIPS ([MARS](#))
- ▶ Travail personnel attendu (CMs, TDs, TPs)



► Objectifs du cours

- ▶ Connaître la représentation interne des données/instructions en binaire
- ▶ Comprendre les mécanismes de fonctionnement d'un processeur
- ▶ Comprendre les interactions entre processeur et périphériques (mémoire, unités de stockage de masse, E/S,...)
- ▶ Appréhender l'impact de l'architecture d'un ordinateur sur les performances de programmes écrits dans des langages de haut niveau

► Notation

Contrôle continu	contrôle « sur table »	30%
	Exercice de TP	5% 50%
	Projet de TP	15%
Examen		50%



Pourquoi un module sur l'architecture des ordinateurs ?

Exemple de la boîte de vitesses d'une voiture : si on connaît son principe de fonctionnement, on sait mieux utiliser le véhicule :

- ▶ Prise directe
- ▶ Démarrage en seconde ou marche arrière sur glace



Motivations (2)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
    int *mat, val;
    unsigned int matSZ, ligne, colonne;

    printf("Taille de la matrice ? ");
    scanf("%d", &matSZ);
    mat = (int*)malloc((matSZ*matSZ)*
                        sizeof(int));
    printf("Position de l'élément ? ");
    scanf("%d %d",&ligne, &colonne);
    printf("Valeur de l'élément ? ");
    scanf("%d",&val);
    if (ligne >= matSZ || colonne >= matSZ) {
        printf("Position invalide\n");
    } else {
        mat[ligne*matSZ+colonne] = val;
    }
}

import java.util.Scanner;

public class scan {
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.printf("Enter i: ");
        int i = in.nextInt();

        int[] M = new int[i*i*4];
        M[5] = 3;
    }
}
```



Motivations (2)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
    int *mat, val;
    unsigned int matSZ, ligne, colonne;

    printf("Taille de la matrice ? ");
    scanf("%d", &matSZ);
    mat = (int*)malloc((matSZ*matSZ)*
                        sizeof(int));
    printf("Position de l'élément ? ");
    scanf("%d %d", &ligne, &colonne);
    printf("Valeur de l'élément ? ");
    scanf("%d", &val);
    if (ligne >= matSZ || colonne >= matSZ) {
        printf("Position invalide\n");
    } else {
        mat[ligne*matSZ+colonne] = val;
    }
}
```

```
import java.util.Scanner;

public class scan {
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.printf("Enter i: ");
        int i = in.nextInt();

        int[] M = new int[i*i*4];
        M[5] = 3;
    }
}
```

- ▶ Avec `sizeof(int)==4` : si `matSZ==32768`, allocation de 0 octet dans le tas
- ▶ Écrasement possible d'autres variables dans le tas, même avec une saisie contrôlée

Autre motivation : débogage d'un problème de pile.



- ▶ John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, September 2006
- ▶ Noam Nisan and Shimon Schocken. *The Elements of Computing Systems : Building a Modern Computer from First Principles*. MIT Press, illustrated edition, 2008
- ▶ Daniel Page. *Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer London, 2009
- ▶ Charles Petzold. *CODE : The Hidden Language of Computer Hardware and Software*. Microsoft Press, 2000



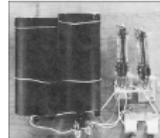
1. Représentation de l'information
2. Performances
3. Logique, circuits combinatoires et circuits séquentiels
4. Architecture du processeur MIPS
(gestion des instructions, chemins de données)
5. Assembleur MIPS
6. Pipelines et caches

Représentation de l'information



Représentation des informations en mémoire

George Stibitz, model K 1937 (« kitchen ») :



Claude Shannon, 1937

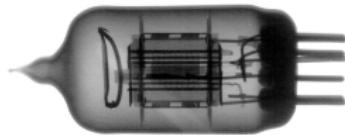
John V. Atanasoff, 1937–1939 (ABC)



Relais électromécanique (Henry, 1835)



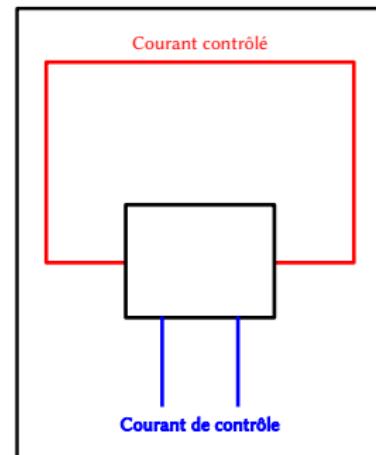
Transistor
(Shockley, Bardeen & Brattain, 1947)



Tube thermionique
"lampe à vide" (De Forest, 1907)



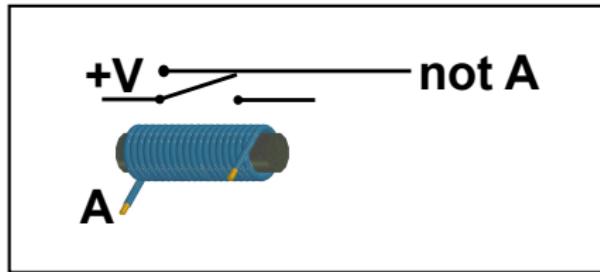
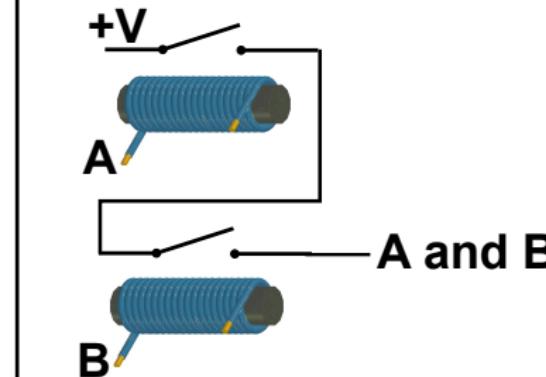
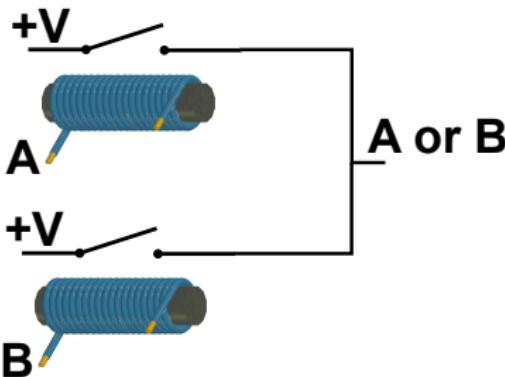
Circuit intégré (Kilby, 1958)



Ordinateur numérique (\neq analogique) : architecture utilisant l'absence ou la présence de courant (relais, tubes ou transistors)



Interrupteurs et logique binaire





Les nombres binaires

- ▶ Deux états internes symbolisés par 0 et 1
- ▶ Informations :

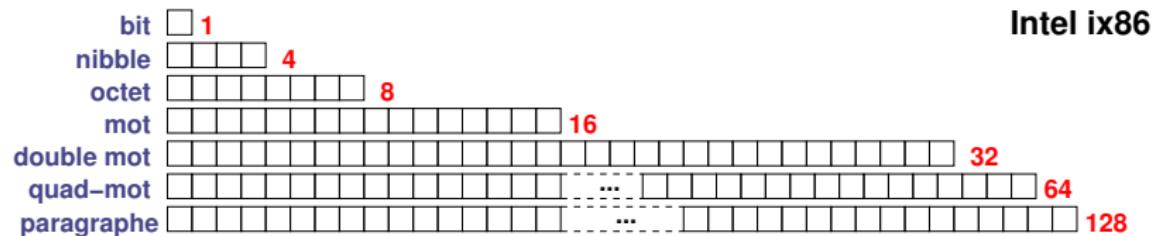
Nom	Valeur
<i>Bit</i>	0 ou 1
<i>Octet</i>	00000000 à 11111111

Multiples définis depuis 1998 :

Nom	Notation	Valeur
1 kibibit	1 Kibit	$2^{10} = 1\,024$ bits
1 kilobit	1 kbit	$10^3 = 1\,000$ bits
1 mebioctet	1 MiB/1 Mio	$2^{20} = 1\,048\,576$ octets
1 megaoctet	1 MB/1 Mo	$10^6 = 1\,000\,000$ octets
1 gibioctet	1 GiB/1 Gio	$2^{30} = 1\,073\,741\,824$ octets
1 gigaoctet	1 GB/1 Go	$10^9 = 1\,000\,000\,000$ octets



Types de données fondamentaux



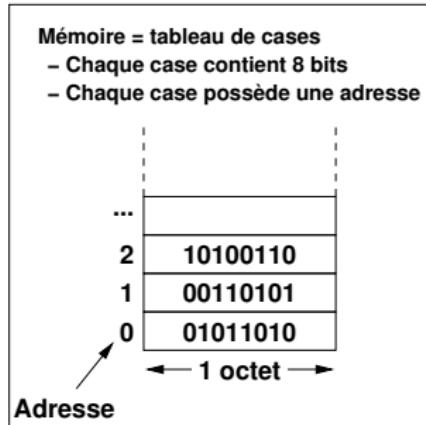


Types de données fondamentaux

		Intel ix86	
bit	<table border="1"><tr><td>1</td></tr></table>	1	
1			
nibble	<table border="1"><tr><td>4</td></tr></table>	4	
4			
octet	<table border="1"><tr><td>8</td></tr></table>	8	
8			
mot	<table border="1"><tr><td>16</td></tr></table>	16	
16			
double mot	<table border="1"><tr><td>32</td></tr></table>	32	
32			
quad-mot	<table border="1"><tr><td>64</td></tr></table>	64	
64			
paragraph	<table border="1"><tr><td>128</td></tr></table>	128	
128			

OU

		MIPS	
bit	<table border="1"><tr><td>1</td></tr></table>	1	
1			
nibble	<table border="1"><tr><td>4</td></tr></table>	4	
4			
octet	<table border="1"><tr><td>8</td></tr></table>	8	
8			
demi-mot	<table border="1"><tr><td>16</td></tr></table>	16	
16			
mot	<table border="1"><tr><td>32</td></tr></table>	32	
32			
double-mot	<table border="1"><tr><td>64</td></tr></table>	64	
64			
paragraph	<table border="1"><tr><td>128</td></tr></table>	128	
128			

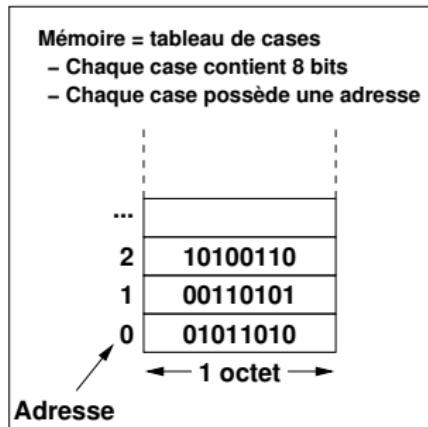


On veut stocker :

- ▶ des entiers positifs (12, 534256, ...)
- ▶ des entiers négatifs (-56, -435345, ...)
- ▶ des caractères ('a', 'Z', '5', '+', ...)
- ▶ des chaînes de caractères ("bonjour", ...)
- ▶ des réels (12.34, -670.5552, ...)
- ▶ des instructions

Mais :

Une case contient uniquement des bits



On veut stocker :

- ▶ des entiers positifs (12, 534256, ...)
- ▶ des entiers négatifs (-56, -435345, ...)
- ▶ des caractères ('a', 'Z', '5', '+', ...)
- ▶ des chaînes de caractères ("bonjour", ...)
- ▶ des réels (12.34, -670.5552, ...)
- ▶ des instructions

Mais :

Une case contient uniquement des bits

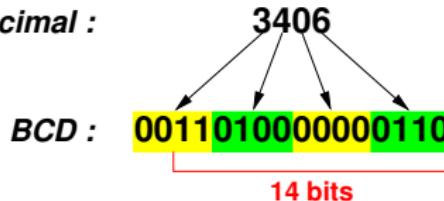
⇒ tout coder sous forme d'*entiers positifs en binaire*



Représentation « décimale codée binaire »

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
xxxx	illégal

Décimal :

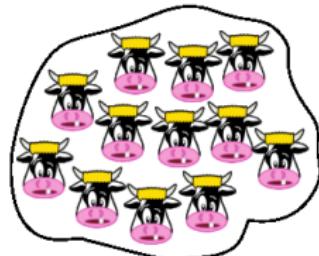


- ▶ Codage BCD gourmand
(12 bits seulement en binaire pour coder 3406)
- ▶ Opérations arithmétiques compliquées et pas efficaces
- ▶ Entrées/sorties facilitées



Représentation usuelle : 12

$$= 1 \times 10^1 + 2 \times 10^0$$



→ Représentation *positionnelle* :

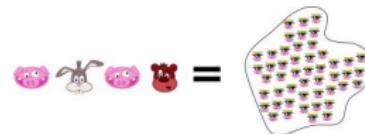
- ▶ Choix d'une base b : (ex. : 10, 2, ...)
- ▶ Choix de b symboles

Exemples :

Base 2 (0, 1) : 1100_2 ($= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12_{10}$)

Base 3 « *Toons* » :

$$\begin{array}{c} \text{Dog} \\ \equiv \\ \text{Cow} \end{array} \quad \begin{array}{c} \text{Pig} \\ \equiv \\ \text{Dog} \end{array} \quad \begin{array}{c} \text{Bunny} \\ \equiv \\ \text{Cow} \end{array}$$



$$1 \times 3^3 + 2 \times 3^2 + 1 \times 3^1 + 0 \times 3^0 = 48$$



Expression d'un nombre a en base b :

$$\begin{aligned} a_b &= (a_n a_{n-1} \dots a_2 a_1 a_0. a_{-1} a_{-2} \dots a_{-m})_b \\ &= a_n b^n + \dots + a_2 b^2 + a_1 b + a_0 + a_{-1} b^{-1} + \dots + a_{-m} b^{-m} \end{aligned}$$

a_n : chiffre *le plus significatif*

a_{-m} : chiffre *le moins significatif*

Exemples :

$$98_{10} = 9 * 10^1 + 8 * 10^0$$

$$101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5_{10}$$

$$136_8 = 1 * 8^2 + 3 * 8^1 + 6 * 8^0 = 94_{10}$$

$$3A_{16} = 72_8 = 58_{10}$$

$$110.01_2 = 1 * 2^2 + 1 * 2^1 + 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} = 6.25_{10}$$

Avec $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, ...



Changement de base : exemple

Passage de 23_{10} en base 10, 2, 3 ?

Division par 10, 2, 3 itérées.

$$\begin{array}{r} 23 \longdiv{10} \\ 3 \quad 2 \longdiv{10} \\ \hline 2 \quad 0 \end{array}$$

$$\begin{array}{r} 23 \longdiv{2} \\ 1 \quad 11 \longdiv{2} \\ \hline 1 \quad 5 \longdiv{2} \\ 1 \quad 2 \longdiv{2} \\ \hline 0 \quad 1 \longdiv{2} \\ \hline 1 \quad 0 \end{array}$$

$$\begin{array}{r} 23 \longdiv{3} \\ 2 \quad 7 \longdiv{3} \\ \hline 1 \quad 2 \longdiv{3} \\ \hline 2 \quad 0 \end{array}$$



Changement de base : exemple

Passage de 23_{10} en base 10, 2, 3 ?

Division par 10, 2, 3 itérées.

$$\begin{array}{r} 23 \mid 10 \\ 3 \quad 2 \mid 10 \\ \hline 2 \quad 0 \end{array}$$

$$\begin{array}{r} 23 \mid 2 \\ 1 \quad 11 \mid 2 \\ 1 \quad 5 \mid 2 \\ 1 \quad 2 \mid 2 \\ \hline 0 \quad 1 \quad 2 \\ \hline 1 \quad 0 \end{array}$$

$$\begin{array}{r} 23 \mid 3 \\ 2 \quad 7 \mid 3 \\ 1 \quad 2 \mid 3 \\ 2 \quad 0 \end{array}$$

Résultat : $(23) = 23_{10} = 10111_2 = 212_3$



Passage de la représentation de u en base r à sa représentation en base R ?

$$\begin{aligned} u &= (x_{k-1}x_{k-2}\dots x_0)_r \\ &= (X_{K-1}X_{K-2}\dots X_0)_R \end{aligned}$$

Représentation de Horner :

$$\begin{aligned} u &= (X_{K-1}R^{K-1} + X_{K-2}R^{K-2} + \dots + X_0)_r \\ &= (R(\dots R(R \times 0 + X_{K-1}) + X_{K-2}) + X_{K-3})\dots + X_0)_r \end{aligned}$$

en exprimant toutes les constantes et en faisant toutes les opérations dans la base r .

⇒ Algorithme : divisions itérées par R en base r



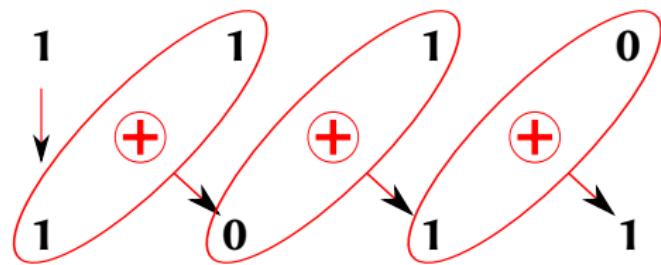
- ▶ Choix d'un ordre pour entiers binaires tel que chaque élément diffère par un seul bit du suivant et du précédent :

Code de Gray	BRGC	Ordre naturel	Nombre
	000	000	0
	001	001	1
	011	010	2
	010	011	3
	110	100	4
	111	101	5
	101	110	6
	100	111	7

- ▶ Ordre non unique (*Binary Reflected Gray Code*)
- ▶ Utilisation : roue codeuse, tableau de Karnaugh, code de correction d'erreur, ...

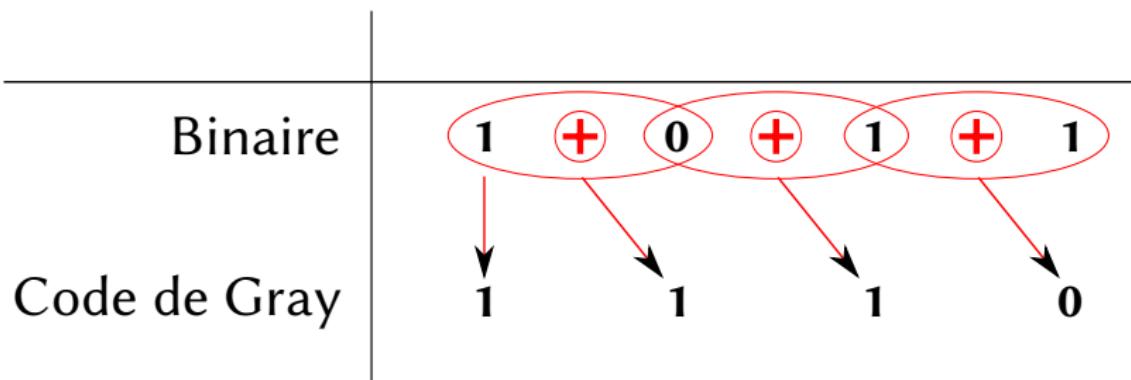
Code de Gray

Binaire


$$\begin{array}{c|cc} \oplus & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$$



Conversion binaire/Gray



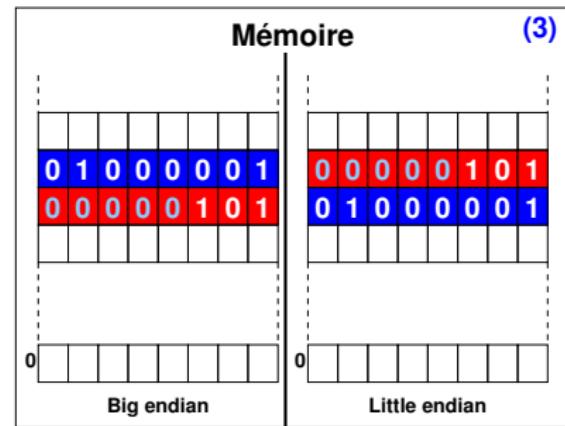
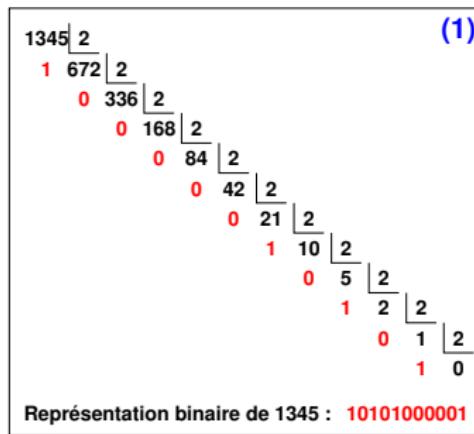
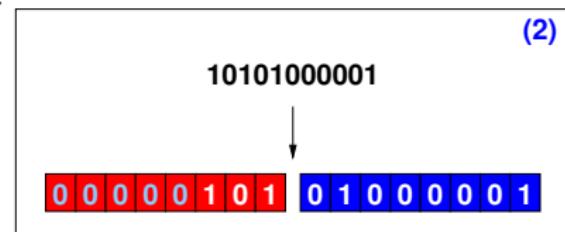
\oplus	0	1
0	0	1
1	1	0



Codage des entiers positifs (non-signés)

Stockage d'un entier positif en mémoire :

1. Représentation du nombre en binaire
2. Découpage de la représentation binaire en octets
3. Stockage de chaque octet consécutivement





Big endian vs. little endian

- ▶ Architecture ix86 : adressage par octet *little-endian*
- ▶ Stockage d'infos sur plus d'un octet :
 - ▶ msb (*Most Significant Byte*) à l'adresse la plus petite
→ **big-endian**
 - ▶ lsb (*Least Significant Byte*) à l'adresse la plus petite
→ **little-endian**

```
unsigned long int x = 3345467;  
{ => x = 0x00330c3b; }
```

	7	0
10000005	...	
10000004	3b	x
10000003	0c	
10000002	33	
10000001	00	
10000000	...	

big-endian

	7	0
10000005	...	
10000004	00	
10000003	33	
10000002	0c	
10000001	3b	x
10000000	...	

little-endian



Les entiers positifs (non-signés)

Entiers représentables sur 1 octet :

Base 2 Base 10 Base 16

11111111	255	FF
11111110	254	FE
...	...	
00010001	17	11
00010000	16	10
00001111	15	0F
00001110	14	0E
00001101	13	0D
00001100	12	0C
00001011	11	0B
00001010	10	0A
00001001	9	09
00001000	8	08
00000111	7	07
00000110	6	06
00000101	5	05
00000100	4	04
00000011	3	03
00000010	2	02
00000001	1	01
00000000	0	00



Les entiers positifs (non-signés)

Entiers représentables sur 1 octet :

Base 2

Base 10 Base 16

11111111	255	FF
11111110	254	FE
...	...	
00010001	17	11
00010000	16	10
00001111	15	0F
00001110	14	0E
00001101	13	0D
00001100	12	0C
00001011	11	0B
00001010	10	0A
00001001	9	09
00001000	8	08
00000111	7	07
00000110	6	06
00000101	5	05
00000100	4	04
00000011	3	03
00000010	2	02
00000001	1	01
00000000	0	00



Passage de base 2 à base 16 et inversement :

Base 2 :

1000101001

Base 16 :

2 2 9

B2	B16
1111	F
1110	E
1101	D
1100	C
1011	B
1010	A
1001	9
1000	8
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0



Entiers non-signés : ensemble d'entiers positifs

Entiers signés : ensemble d'entiers positifs et négatifs

Comment représenter des entiers négatifs ?



Entiers non-signés : ensemble d'entiers positifs

Entiers signés : ensemble d'entiers positifs et négatifs

Comment représenter des entiers négatifs ?

Convention de recodage des chaînes de bits

- ▶ Magnitude signée
- ▶ Complément à 1
- ▶ Complément à 2
- ▶ Biaisée



Dans un entier de k bits, le bit de poids fort code le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Exemples (sur 8 bits) :

- ▶ $+25_{10} = 00011001_2$
- ▶ $-25_{10} = 10011001_2$

Inconvénient = deux représentations pour 0 :

- ▶ $+0_{10} = 00000000_2$
- ▶ $-0_{10} = 10000000_2$

Sur 8 bits : -127..+127



Représentation par magnitude signée

Dans un entier de k bits, le bit de poids fort code le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Exemples (sur 8 bits) :

- ▶ $+25_{10} = 00011001_2$
- ▶ $-25_{10} = 10011001_2$

Inconvénient = deux représentations

- ▶ $+0_{10} = 00000000_2$
- ▶ $-0_{10} = 10000000_2$

Sur 8 bits : -127..+127

chaîne de bits	non signé	magnitude signée
1111	15	-7
1110	14	-6
1101	13	-5
1100	12	-4
1011	11	-3
1010	10	-2
1001	9	-1
1000	8	-0
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0



Le bit de poids fort correspond au signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en complémentant bit à bit sa valeur absolue avec 1 (cf. complément à 9 de la pascaline)

Exemple : représentation de -25_{10} sur 8 bits :

- ▶ $25_{10} = 00011001_2$
$$\begin{array}{r} 11111111 \\ - 00011001 \\ \hline = 11100110 \end{array}$$
- ▶ D'où :

Deux représentations pour 0 : 00000000_2 et 11111111_2
Nombres représentés sur 8 bits : -127..+127



Complément à 1

Le bit de poids fort correspond au signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en complément à 1 de son opposé absolue avec 1 (cf. complément à 9 dans les nombres décimaux)

Exemple : représentation de -25_{10} sur 8 bits

- ▶ $25_{10} = 00011001_2$
$$\begin{array}{r} 11111111 \\ - 00011001 \\ \hline = 11100110 \end{array}$$
- ▶ D'où : $\begin{array}{r} - 00011001 \\ \hline = 11100110 \end{array}$

Deux représentations pour 0 : 00000000 et 11111111

Nombres représentés sur 8 bits : -127 à 127

chaîne de bits	non signé	complément à 1
1111	15	-0
1110	14	-1
1101	13	-2
1100	12	-3
1011	11	-4
1010	10	-5
1001	9	-6
1000	8	-7
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0



Le bit de poids fort indique le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement).

Exemple : représentation de -25_{10} ?

- ▶ $+25_{10} = 00011001_2$
- ▶ Complément à 1 de $+25_{10} = 11100110_2$
- ▶ Ajout de 1 : $11100110_2 + 1 = 11100111_2$

Une seule représentation pour 0 : 00000000_2

Nombres représentés sur 8 bits : -128..+127

Le bit de poids fort indique le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en prenant la valeur absolue (et inversement)

Exemple : représentation de -2

- ▶ $+25_{10} = 00011001_2$
- ▶ Complément à 1 de $+25_{10}$
- ▶ Ajout de 1 : $11100110_2 + 1$

Une seule représentation pour 0 et -0

Nombres représentés sur 8 bits

chaîne de bits	non signé	complément à 2
1111	15	-1
1110	14	-2
1101	13	-3
1100	12	-4
1011	11	-5
1010	10	-6
1001	9	-7
1000	8	-8
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0



Représentation des nombres négatifs par ajout d'un biais les rendant positifs.

Le biais est ajouté aussi aux nombres positifs

Exemple : codage sur 8 bits avec un biais de 127

- ▶ -12_{10} est codé par $-12 + 127 = 115$ i.e. 01110011_2
- ▶ 30_{10} est codé par $30 + 127 = 157$ i.e. 10011101_2

Nombres représentés sur 8 bits avec biais de 127 : -127..128



Représentation des nombres négatifs en utilisant les représentations rendant positifs.

Le biais est ajouté aussi aux nombres négatifs.

Exemple : codage sur 8 bits avec biais de 7

- ▶ -12_{10} est codé par $-12 + 127 = 115$
- ▶ 30_{10} est codé par $30 + 127 = 157$

Nombres représentés sur 8 bits

chaîne de bits	non signé	codage avec biais de 7
1111	15	8
1110	14	7
1101	13	6
1100	12	5
1011	11	4
1010	10	3
1001	9	2
1000	8	1
0111	7	0
0110	6	-1
0101	5	-2
0100	4	-3
0011	3	-4
0010	2	-5
0001	1	-6
0000	0	-7



Les nombres négatifs : résumé

binaire	décimal	signe + magnitude	complément à 1	complément à 2	représentation biaisée
0000	0	0	0	0	-7
0001	1	1	1	1	-6
0010	2	2	2	2	-5
0011	3	3	3	3	-4
0100	4	4	4	4	-3
0101	5	5	5	5	-2
0110	6	6	6	6	-1
0111	7	7	7	7	0
1000	8	-0	-7	-8	1
1001	9	-1	-6	-7	2
1010	10	-2	-5	-6	3
1011	11	-3	-4	-5	4
1100	12	-4	-3	-4	5
1101	13	-5	-2	-3	6
1110	14	-6	-1	-2	7
1111	15	-7	-0	-1	8

(biais = 7)



Les nombres négatifs : résumé

binaire	décimal	signe + magnitude	complément à 1	complément à 2	représentation biaisée
0000	0	0	0	0	-7
0001	1	1	1	1	-6
0010	2	2	2	2	-5
0011	3	3	3	3	-4
0100	4	4	4	4	-3
0101	5	5	5	5	-2
0110	6	6	6	6	-1
0111	7	7	7	7	0
1000	8	-0	-7	-8	1
1001	9	-1	-6	-7	2
1010	10	-2	-5	-6	3
1011	11	-3	-4	-5	4
1100	12	-4	-3	-4	5
1101	13	-5	-2	-3	6
1110	14	-6	-1	-2	7
1111	15	-7	-0	-1	8

(biais = 7)

Quelle est la valeur de la chaîne de bits : 1010 ?



Plusieurs formats pour représenter des caractères (imprimables et de contrôle) sous forme binaire :

- ▶ EBCDIC (*Extended Binary-Coded Decimal Interchange Code*)
 - ▶ Représentation sur 8 bits (256 caractères possibles)
 - ▶ Utilisé autrefois sur les mainframes IBM
- ▶ ASCII (*American Standard Code for Information Interchange*)
 - ▶ Représentation sur 7 bits (pas d'accents)
 - ▶ ASCII étendu : sur 8 bits mais pas de normalisation
- ▶ Unicode : encodage sur 16 bits (65536 possibilités) pour représenter tous les caractères de toutes les langues

Chaîne de caractères : suite de caractères stockés consécutivement en mémoire (terminateurs ?)



Exemple : table ASCII restreinte

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Source : <http://upload.wikimedia.org/wikipedia/commons/1/b/ASCII-Table-wide.svg>



Opérations $+$, $-$, \times , \div sur :

- ▶ Nombres non-signés
- ▶ Nombres signés *en complément à 2*

Le calcul se fait indépendamment de l'interprétation des chaînes de bits



L'addition se fait classiquement avec les règles :

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0 \quad \text{avec retenue de 1}$$

Exemples :

11		1 1 1 1 1	
0 0 0 1 1 0 1 0	26	0 0 0 1 0 0 1 1	19
+ 0 0 0 0 1 1 0 0	12	+ 0 0 1 1 1 1 1 0	62
<hr/>	38	<hr/>	81
		0 1 0 1 0 0 0 1	

Résultat sur 9 bits :

- ▶ Non signé : dépassement de capacité
- ▶ Signé : pas de signification



Soustraction binaire entière

La soustraction suit les règles suivantes :

$$0 - 0 = 0$$

$0 - 1 = 1$ et on prend 1 à gauche

$$1 - 0 = 1$$

$$1 - 1 = 0$$

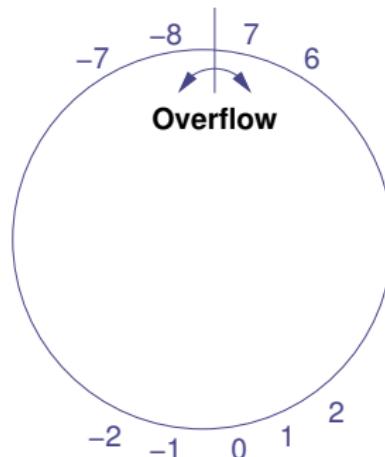
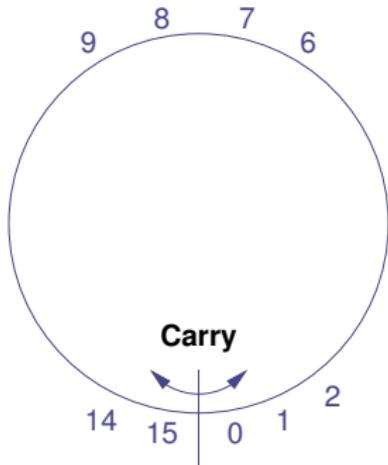
Exemples :

$001\color{red}{1}00101$	37	$001\color{red}{1}10111$	51
$- 00010001$	17	$- 00010110$	22
$\underline{- \quad 1}$	20	$\underline{- \quad 111}$	29

On peut aussi faire une addition avec le complément à 2 du deuxième opérande.



Carry vs. Overflow



Non signé

$$\begin{array}{r} 1110 \\ + 1011 \\ \hline 11001 \end{array}$$

$$\begin{array}{r} 14 \\ 11 \\ \hline 25 > 15 \end{array}$$

Signé

$$\begin{array}{r} 1110 \\ + 1011 \\ \hline 11001 \end{array}$$

$$\begin{array}{r} -2 \\ -5 \\ \hline -7 \end{array}$$

Non signé

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

$$\begin{array}{r} 7 \\ 1 \\ \hline 8 \end{array}$$

Signé

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

$$\begin{array}{r} 7 \\ 1 \\ \hline -8 \end{array}$$

Détection :
1 bit de plus en sortie

Détection :
Opérandes de même signe
Résultat de signe différent



Multiplication binaire entière

La multiplication suit les règles suivantes :

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Exemple :

$$\begin{array}{r} 00101001 & 41 \\ \times 00000110 & 6 \\ \hline 00000000 \\ 00101001 \\ 00101001 \\ \hline 001110110 & 246 \end{array}$$

On peut aussi faire des additions itérées



Division binaire entière

Division obtenue par itération de soustractions jusqu'à ce que le résultat de la soustraction soit inférieur au diviseur :

- ▶ Quotient = nombre de soustractions
- ▶ Reste = résultat de la dernière soustraction

Exemple : division de 7 par 3

$$\begin{array}{r} 00000111 & 7 & 000001\overset{1}{0}\overset{1}{0} & 4 \\ - 00000011 & 3 & - 00000011 & 3 \\ \hline 00000100 & 4 & \hline 00000001 & 1 \end{array}$$

1 2

Résultat : quotient = 2 et reste = 1

On peut aussi faire comme une division classique en décimal



- ▶ Infinité de nombres entiers
 - ➡ Mais représentation correcte dans un intervalle
- ▶ Infinité de nombres réels
 - ▶ Impossibilité de représentation correcte même d'un petit intervalle :

$$\forall \mathbf{a}, \mathbf{b} \in \mathbb{R} \ \exists \mathbf{c} \in \mathbb{R} \quad \text{t.q.} \quad \mathbf{a} \leq \mathbf{c} \leq \mathbf{b}$$

⇒ Représentation d'un sous-ensemble de \mathbb{Q}

- ▶ Nombres en virgule fixe :

$$110.011 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 6.375$$

- ▶ Nombres en virgule flottante (notation scientifique) :

$$101010.10 = 1.0101010 \times 2^5$$

$$0.0010001 = 1.00011 \times 2^{-3}$$

- ⇒ Usage de l'arithmétique en virgule flottante majoritaire



Passage d'un nombre réel de base 10 vers base 2 en virgule fixe :

- ▶ Partie entière : comme pour les entiers
- ▶ Partie décimale : multiplications itérées par 2

Exemple : conversion de 14.375_{10} en base 2 ?

- ▶ $14_{10} = 1110_2$ (divisions itérées par 2)
- ▶ $0.375_{10} = ???_2$

$$\begin{array}{rcl} 0.375 & \times 2 & = 0.75 \\ 0.75 & \times 2 & = 1.5 \\ 0.5 & \times 2 & = 1.0 \end{array}$$

Résultat : $14.375_{10} = 1110.\textcolor{red}{011}_2$



1936 : machines Z de K. Zuse

60's–70's : Design des FPUs = anarchie totale

- ➡ Portabilité nulle
- ➡ Peu de propriétés ($a = b \Leftrightarrow a - b = 0$)

1976 : création du i8087 par Intel (*best arithmetic*)

- ➡ Rapport Kahan–Coonen–Stone

1985 : Rapport K-C-S devient la norme *IEEE 754*

Aujourd'hui : norme implantée de fait sur tous les ordinateurs (sauf certains Cray).



Nombre flottant x en binaire :

- ▶ un bit de signe s
- ▶ un exposant E
- ▶ un *signifiant* m

$$x = (-1)^s \times m \cdot 2^E$$

Représentations équivalentes :

- (a) 0.0000111010 · 2⁰
- (b) 0.000000111010 · 2²
- (c) 1.111010 · 2⁻⁵

Taille de significant fixée \Rightarrow forme c plus précise



Représentation IEEE 754 (1)

- ▶ Représentation *normalisée* (forme c) ;
- ▶ Toujours un 1 avant la virgule \Rightarrow pas codé (*hidden bit*)

$$m = 1.00101 \longrightarrow f = 00101$$

- ▶ exposants négatifs et positifs : codage par biais :

$$E \longrightarrow e = E + \text{biais}$$

Intérêt : comparaison lexicographique

single (1,8,23)
double (1,11,52)
ix87 reg. (1,15,64)

s	e	f
1	10101010101	101010101010...10100101

Signe Exposant biaisé

Partie fractionnaire

Exemple : format tiny sur 5 bits (1, 2, 2) de biais 1 :

Nombres positifs représentables :

$$0\ 00\ 00 \rightsquigarrow 1.00 \times 2^{-1} = 0.5$$

$$0\ 00\ 01 \rightsquigarrow 1.01 \times 2^{-1} = 0.625$$

$$0\ 00\ 10 \rightsquigarrow 1.10 \times 2^{-1} = 0.75$$

$$0\ 00\ 11 \rightsquigarrow 1.11 \times 2^{-1} = 0.875$$

$$0\ 01\ 00 \rightsquigarrow 1.00 \times 2^0 = 1$$

$$0\ 01\ 01 \rightsquigarrow 1.01 \times 2^0 = 1.25$$

$$0\ 01\ 10 \rightsquigarrow 1.10 \times 2^0 = 1.5$$

$$0\ 01\ 11 \rightsquigarrow 1.11 \times 2^0 = 1.75$$

$$0\ 10\ 00 \rightsquigarrow 1.00 \times 2^1 = 2$$

$$0\ 10\ 01 \rightsquigarrow 1.01 \times 2^1 = 2.5$$

$$0\ 10\ 10 \rightsquigarrow 1.10 \times 2^1 = 3$$

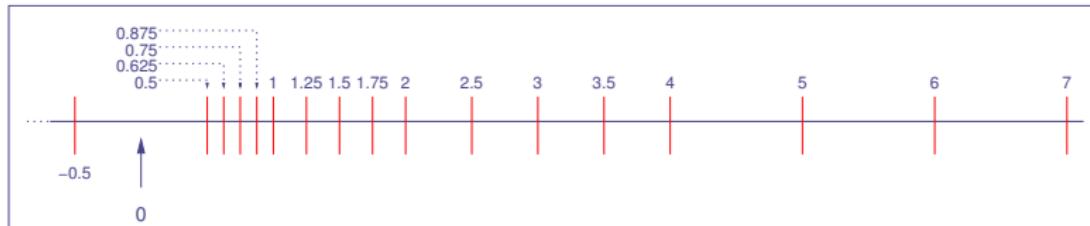
$$0\ 10\ 11 \rightsquigarrow 1.11 \times 2^1 = 3.5$$

$$0\ 11\ 00 \rightsquigarrow 1.00 \times 2^2 = 4$$

$$0\ 11\ 01 \rightsquigarrow 1.01 \times 2^2 = 5$$

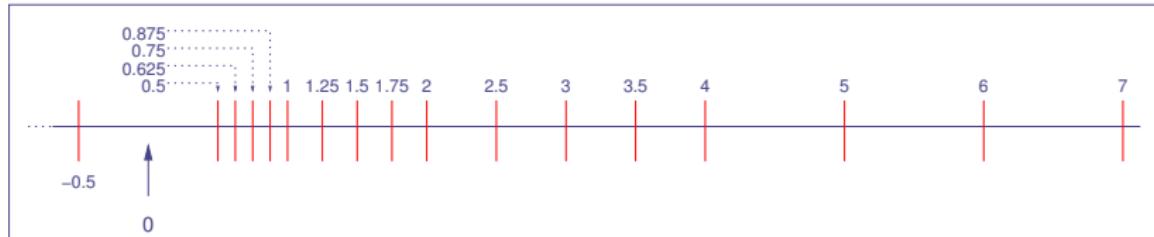
$$0\ 11\ 10 \rightsquigarrow 1.10 \times 2^2 = 6$$

$$0\ 11\ 11 \rightsquigarrow 1.11 \times 2^2 = 7$$





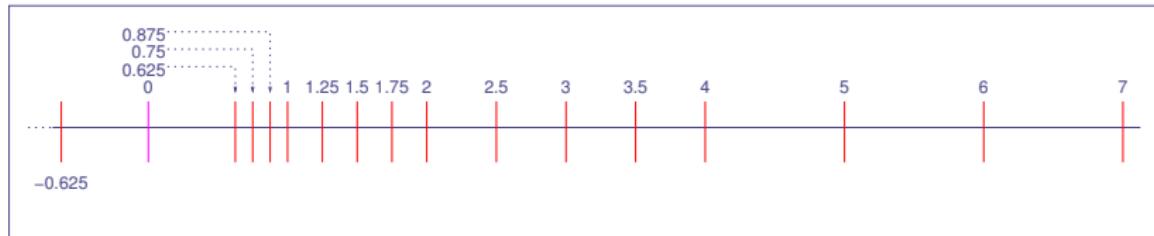
Représentation IEEE 754 (3)



- ▶ Pas de codage pour 0



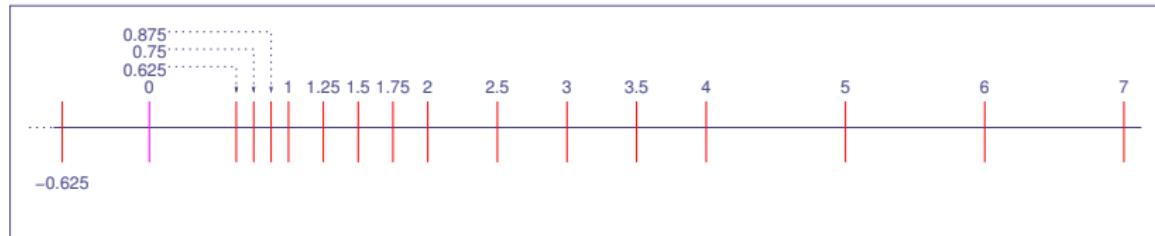
Représentation IEEE 754 (3)



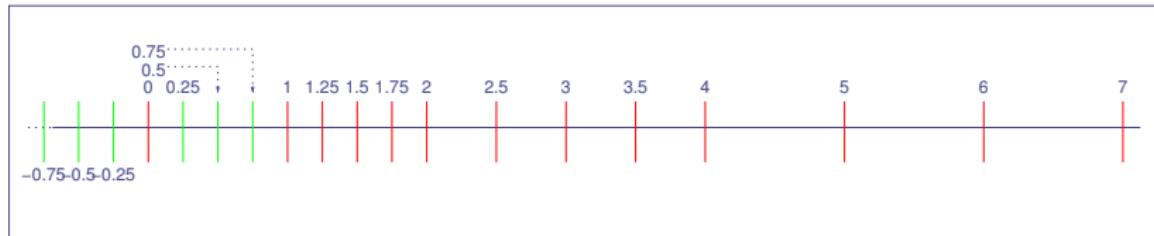
- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)



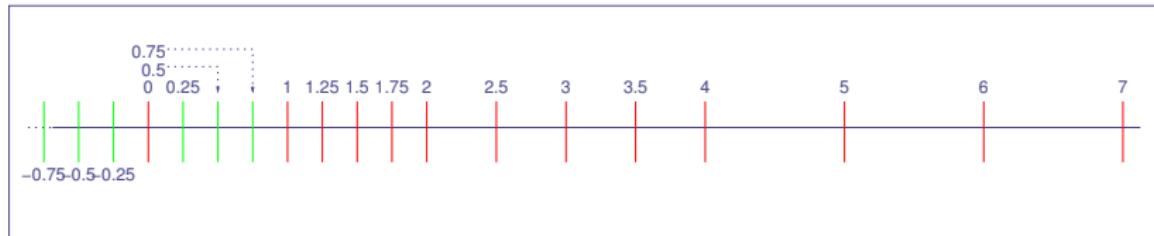
Représentation IEEE 754 (3)



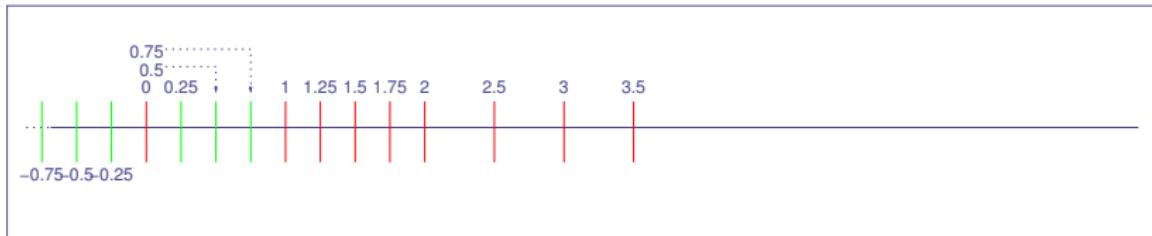
- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0



- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0
 - ▶ Réserver $e = 0$ pour les nombres *dénormalisés*
 - ➡ Plus de *hidden bit* à 1



- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0
 - ▶ Réserver $e = 0$ pour les nombres *dénormalisés*
 - ➡ Plus de *hidden bit* à 1
- ▶ Notions d'infinis mathématiques et de résultat indéfini :



- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0
 - ▶ Réserver $e = 0$ pour les nombres *dénormalisés*
 - ➡ Plus de *hidden bit* à 1
- ▶ Notions d'infinis mathématiques et de résultat indéfini :
 - ▶ Réserver $e = 3$

Interprétation des bits :

$$\left\{ \begin{array}{lll} e = 3, & f \neq 0 & : v = \text{NaN} \\ e = 3, & f = 0 & : v = (-1)^s \times \infty \\ 0 < e < 3 & & : v = (-1)^s \times (1.f) \cdot 2^{e-1} \\ e = 0, & f \neq 0 & : v = (-1)^s \times (0.f) \cdot 2^0 \\ e = 0, & f = 0 & : v = (-1)^s \times 0 \end{array} \right.$$

00000	0.5	00000	0
00001	0.625	00001	0.25
00010	0.75	00010	0.5
00011	0.875	00011	0.75
00100	1	00100	1
00101	1.25	00101	1.25
00110	1.5	00110	1.5
00111	1.75	00111	1.75
01000	2	01000	2
01001	2.5	01001	2.5
01010	3	01010	3
01011	3.5	01011	3.5
01100	4	01100	$+\infty$
01101	5	01101	NaN
01110	6	01110	NaN
01111	7	01111	NaN



- ▶ Not a Number : cas indéfinis (*non ordonnés*)

- ▶ Infinis :
$$\begin{cases} 1/+\infty = +0 \\ 1/-\infty = -0 \\ 1/-0 = -\infty \\ 1/0 = +\infty \end{cases}$$

$x + y$	<i>NaN</i>	$-\infty$	0	$+\infty$
<i>NaN</i>	✓	✓	✓	✓
$-\infty$	✓			✓
0	✓			
$+\infty$	✓	✓		

\sqrt{x}	
<i>NaN</i>	✓
$-\infty$	
$x < 0$	✓
0	
$+\infty$	

$x \times y$	<i>NaN</i>	$-\infty$	0	$+\infty$
<i>NaN</i>	✓	✓	✓	✓
$-\infty$	✓		✓	
0	✓	✓		✓
$+\infty$	✓		✓	

$x - y$	<i>NaN</i>	$-\infty$	0	$+\infty$
<i>NaN</i>	✓	✓	✓	✓
$-\infty$	✓		✓	
0	✓			
$+\infty$	✓			✓

x/y	<i>NaN</i>	$-\infty$	0	$+\infty$
<i>NaN</i>	✓	✓	✓	✓
$-\infty$	✓		✓	
0	✓			✓
$+\infty$	✓		✓	✓



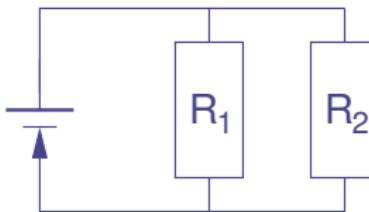
Cas spéciaux pour les opérateurs non définis par la norme IEEE
754 :

x	$\exp(x)$	$\log(x)$	$\sin(x)$	$\cos(x)$	$\tan(x)$	$\text{acos}(x)$	$\text{asin}(x)$	$\text{atan}(x)$
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$-\infty$	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$x < -745.13$	0.0							
$x < -1$								
$x < 0$		NaN						
0		$-\infty$						
$x > 1$								
$x > 709.78$	$+\infty$	$+\infty$	NaN	NaN	NaN	NaN	NaN	NaN
$+\infty$	$+\infty$	$+\infty$						

Source : librairie *fclibm* sur les doubles



Calculer la résistance totale du circuit :



Formule :

$$T = \frac{1}{1/R_1 + 1/R_2}$$

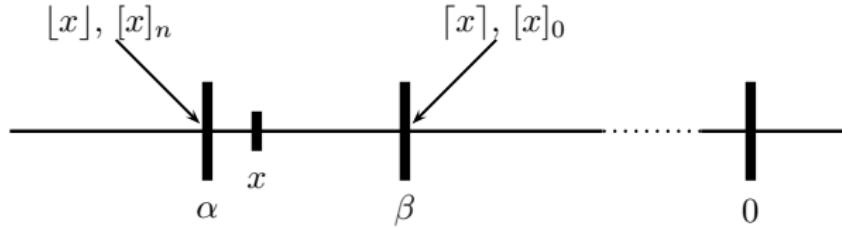
Cas où l'une des résistances est nulle ?

→ Résistance totale nulle

- ▶ Ensemble des flottants \mathbb{F} non clos pour les opérations de base
- ▶ conversion exacte décimal/binaire pas toujours possible

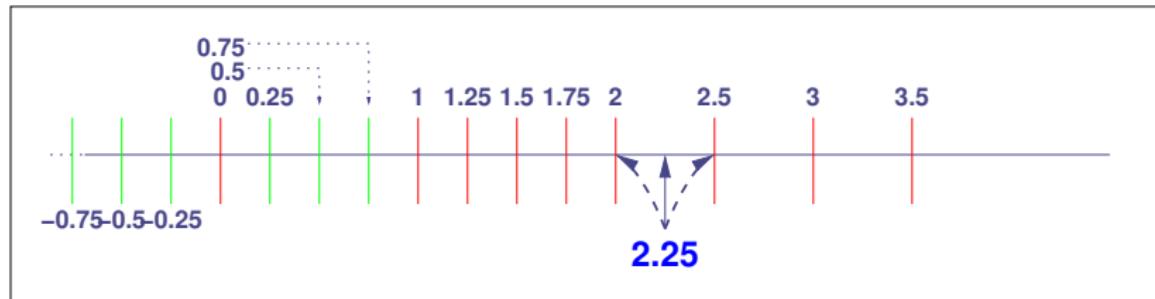
⇒ Fonctions d'arrondi :

$$\left\{ \begin{array}{ll} \lfloor x \rfloor & \text{arrondi vers } -\infty \\ \lceil x \rceil & \text{arrondi vers } +\infty \\ [x]_0 & \text{arrondi vers 0} \\ [x]_n & \text{arrondi au plus proche pair} \end{array} \right.$$

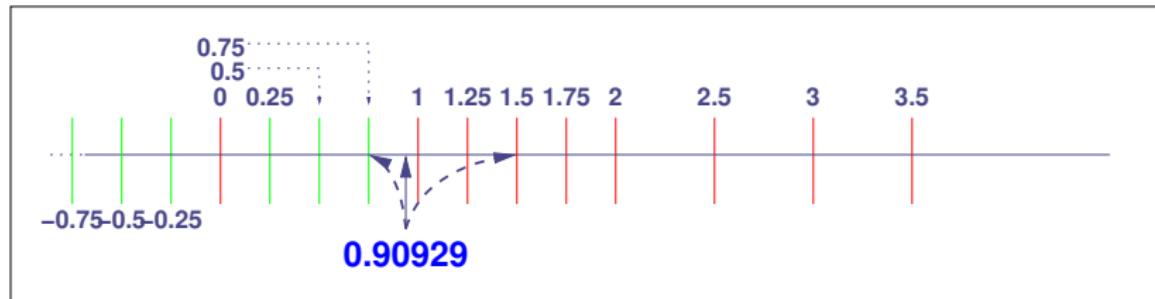




Opérations de base



- ▶ Opérations $\{+, -, \times, /, \sqrt{\}\}$:
arrondi correct suivant le mode courant
(erreur $< 1 \text{ ulp.}$) ;
 - ▶ Exemple : $2.0 + 0.25 = 2.25$ arrondi à 2 ou 2.5



- ▶ Opérations $\{+, -, \times, /, \sqrt{\}\}$:
arrondi correct suivant le mode courant
(erreur $< 1 \text{ ulp.}$) ;
 - ▶ Exemple : $2.0 + 0.25 = 2.25$ arrondi à 2 ou 2.5
- ▶ fonctions transcendantales : ***aucune garantie***
 - ▶ Exemple : $\sin(2.0) = 0.90929$ arrondi vers 0.75 ou 1.5

- ▶ Addition possible si et seulement si les opérandes ont même exposant
- ▶ Exposants différents \Rightarrow décalage du nombre de *plus petit* exposant

Exemple :

$$\begin{array}{r} 10.375 \\ + 6.34375 \\ \hline 16.71875 \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 1.0100110 \times 2^3 \\ + 1.1001011 \times 2^2 \\ \hline \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 1.0100110 \times 2^3 \\ + 0.1100101 \times 2^3 \\ \hline 10.0001011 \times 2^3 \end{array}$$

~~10.375~~ ~~1.1001011~~ ~~10.0001011~~

↓

$16.6875 = 1.0000101 \times 2^4$

Attention : bit de garde



Soustraction en nombres flottants

- ▶ Soustraction possible si et seulement si les opérandes ont même exposant
- ▶ Exposants différents \Rightarrow décalage du nombre de *plus petit* exposant

Exemple :

$$\begin{array}{r} 16.75 \\ - 15.9375 \\ \hline 0.8125 \end{array} \quad \begin{array}{r} 1.0000110 \times 2^4 \\ - 1.111111\textcircled{1} \times 2^3 \\ \hline \end{array} \quad \begin{array}{r} 1.0000110 \times 2^4 \\ - 0.1111111 \times 2^4 \\ \hline 0.0000111 \times 2^4 \\ \downarrow \\ 0.875 \end{array}$$

The diagram illustrates floating-point subtraction. It shows three columns: the first column shows the decimal result 0.8125; the second column shows the binary representation of 16.75 minus 15.9375, which is 0.8125 in binary; the third column shows the step-by-step subtraction of 1.1111111 from 1.0000110. A red circle highlights the least significant bit of the subtrahend (1.1111111) that is being subtracted. Red arrows point from the first column to the second, and from the second to the third. A large red 'X' is drawn over the second column's result, indicating it is incorrect. A red dotted arrow points from the first column to the third column's result, 0.875.

Attention : bit de garde



- ▶ Multiplication des significants et ajout des exposants

Exemple :

$$\begin{array}{r} 10.375 \\ \times \quad 2.5 \\ \hline 25.9375 \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 1.0100110 \times 2^3 \\ \times 1.0100000 \times 2^1 \\ \hline 10100110 \\ 10100110 \\ \hline 1.10011111000000 \times 2^4 \end{array}$$

Diagram illustrating the floating-point multiplication process:

- Left:** A decimal multiplication problem 10.375×2.5 is shown.
- Right:** The significands are converted to binary: $10.375 = 1.0100110 \times 2^3$ and $2.5 = 1.0100000 \times 2^1$.
- Calculation:** The significands are multiplied: $1.0100110 \times 1.0100000 = 1.10011111000000$. The exponent is added: $3 + 1 = 4$.
- Final Result:** The result is $1.10011111000000 \times 2^4$, which is equivalent to 25.875 .

Annotations:

- A red dashed arrow points from the decimal multiplication to the binary multiplication.
- A red dotted arrow points from the decimal result 25.9375 to the binary result 1.10011111000000 .
- A red 'X' is drawn over the decimal multiplication problem.
- A red arrow points from the binary result 1.10011111000000 down to the final result 25.875 .

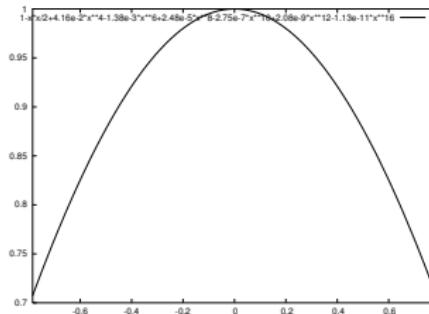
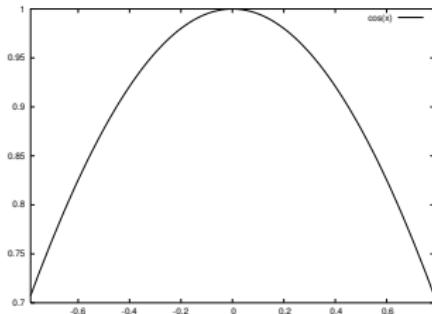


- ▶ Procédure plus compliquée
- ▶ Parfois implémentée par utilisation de la méthode de Newton

- ▶ Fonctions cos, sin, exp, ... :
 - ▶ Utilisation d'approximations polynomiales
 - ▶ Implémentation dans une librairie (e.g. en C)
 - ▶ Implémentation codée dans l'*unité arithmétique flottante*

Exemple : Pour $x \in [-\frac{\pi}{4}, \frac{\pi}{4}]$:

$$\cos x \approx 1 - \frac{x^2}{2} + 4.16 \cdot 10^{-2}x^4 - 1.38 \cdot 10^{-3}x^6 + 2.48 \cdot 10^{-5}x^8 - 2.75 \cdot 10^{-7}x^{10} + 2.08 \cdot 10^{-9}x^{12} - 1.13 \cdot 10^{-11}x^{16}$$





- ▶ *Arrondi.*
$$\begin{cases} x \rightarrow \tilde{x} = x(1 + \delta), & \delta \leq \varepsilon/2 \\ x + y \rightarrow x \oplus y = (x + y)(1 + \delta) \end{cases}$$
- ▶ *Absorption.*

Addition opérandes de magnitudes différentes :

$$1.345 \cdot 10^5 + 1.45 \cdot 10^1 = 1.345 \cdot 10^5$$

Alignment \Rightarrow chiffres significatifs de $1.45 \cdot 10^1$ éliminés.

- ▶ *Cancellation bénigne.* Voir arrondis.
- ▶ *Cancellation catastrophique.*

Soustraction opérandes entachés d'erreurs :

$$a = 1.22, b = 3.34, c = 2.28, b^2 - 4ac = 11.2 - 11.1 = .1 \neq .0292$$



$$x \oplus y = y \oplus x \quad (\text{commutativité de l'addition})$$

$$x \otimes y = y \otimes x \quad (\text{commutativité de la multiplication})$$

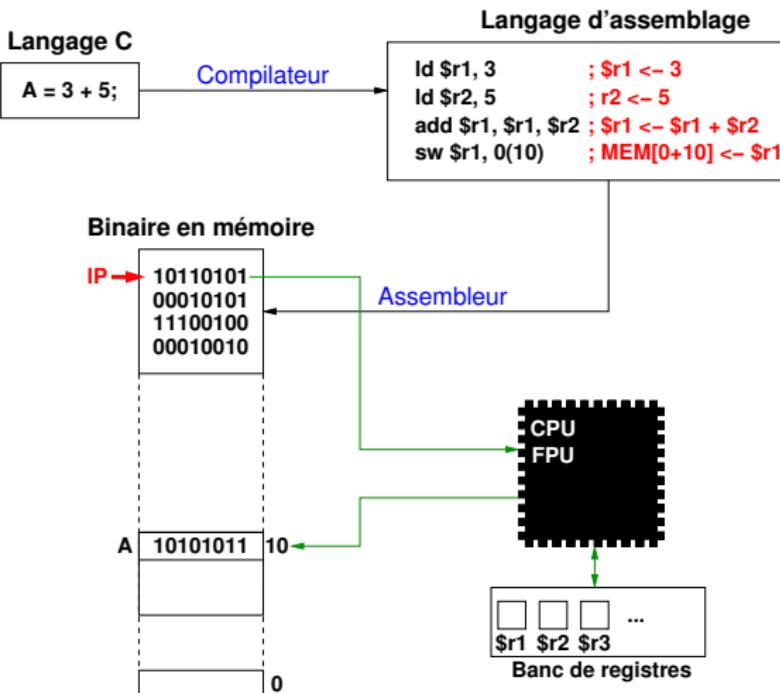
$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z \quad (\text{non associativité})$$

$$x \otimes (y \oplus z) \neq x \otimes y \oplus x \otimes z \quad (\text{non distributivité})$$

\Rightarrow ordre des calculs pas indifférent. (e.g. $\sum_{i=1}^n x_i$)



Codage des instructions machine (1)





Codage des instructions machine (2)

Deux types d'architecture :

	RISC <i>Reduced Instruction Set Computer</i>	CISC <i>Complex Instruction Set Computer</i>
Nb. d'instr.	Peu	Beaucoup
Taille d'instr.	Fixe	Variable
Arité des instr.	Fixe	Variable
Adressage	Peu	Beaucoup

- ▶ RISC (Sparc, MIPS) : instructions simples et rapides
 - ▶ Optimisations faciles
 - ▶ Programmes complexes longs
- ▶ CISC (ix86) : instructions plus ou moins complexes
 - ▶ Programmes plus petits
 - ▶ Optimisation plus difficiles



Types de jeux d'instructions

$A \leftarrow 3 + 5 ?$

Machine à pile	Machine à accumulateur	Machine à registres
PUSH X $top(pile) \leftarrow X$	LOAD X $acc \leftarrow X$	LOAD R_i, X $R_i \leftarrow X$
POP X $X \leftarrow top(pile)$	STORE X $X \leftarrow acc$	STORE R_i, X $X \leftarrow R_i$
ADD $POP t_2 ; POP t_1$ $PUSH t_1 + t_2$	ADD X $acc \leftarrow acc + X$	ADD R_i, R_j, R_k $R_i \leftarrow R_j + R_k$
push 3 push 5 add pop A	load 3 add 5 store A	load R1, 3 load R2, 5 add R3, R1, R2 store R3, A



Codage des instructions

Zepto 0 :

- 4 registres de 8 bits
- Mémoire de 16 octets
- 4 instructions : add, sub, mul, div

Format d'instructions

INSTR	OP	OP	Rd	Rs1	Rs2
add	00				
sub	01				
mul	10				
div	11				

Exemple de programme

Calcul de $(3 \cdot 5 + 12) / 7$

Mettre 3 dans R0
Mettre 5 dans R1
Mettre 12 dans R2
Mettre 7 dans R3

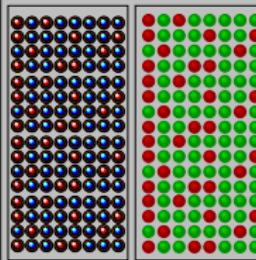
```
mul R0, R0, R1      # R0 <- R0 * R1
add R0, R0, R2      # R0 <- R0 + R2
div R0, R0, R3      # R0 <- R0 / R3
```

Résultat dans R0

Registres



Mémoire code



Zepto 0

Codage du programme

1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
1	1	0	0	0	0	1	1

Performances



- ▶ Comparaison des performances de deux systèmes ?
Intel I7 @ 2.66 GHz vs. AMD phenom xII 965 @ 3.4 GHz ?
- ▶ Meilleur rapport qualité/prix ?
- ▶ Performances dépendant des applications (calcul scientifique, traitement de texte, ...)
- ▶ Impact des choix d'architecture sur les performances ?
 - Achat d'un nouveau processeur vs. achat d'un meilleur compilateur
 - Impact d'une nouvelle fonctionnalité sur les performances



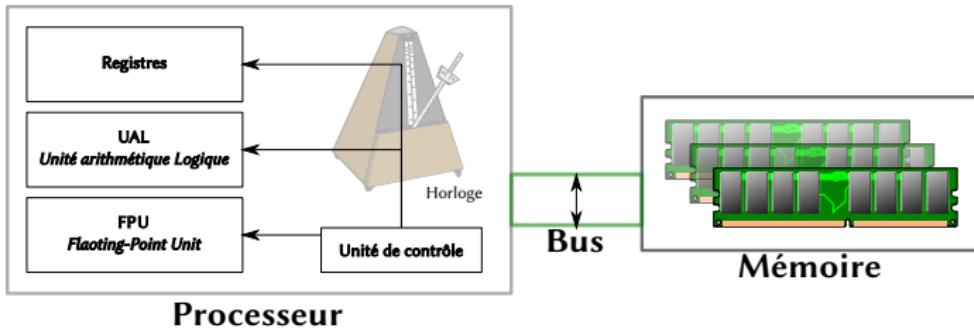
Évaluation du temps d'exécution

$$\text{Performance} = \frac{1}{\text{Temps d'exécution}}$$

Définition du *temps d'exécution* :

- ▶ *Wall time* : temps total à attendre la fin de l'exécution
 - ▶ *Process time* : temps de travail du CPU sur le problème
 - ▶ *User time* : temps passé dans le code de l'utilisateur
 - ▶ *System time* : temps passé en appels système
 - ▶ *Idle time* : temps passé en attente

```
% time pwgen
real    0m0.034s  # <- Wall time
user    0m0.000s  # <- user time
sys     0m0.030s  # <- system time
```

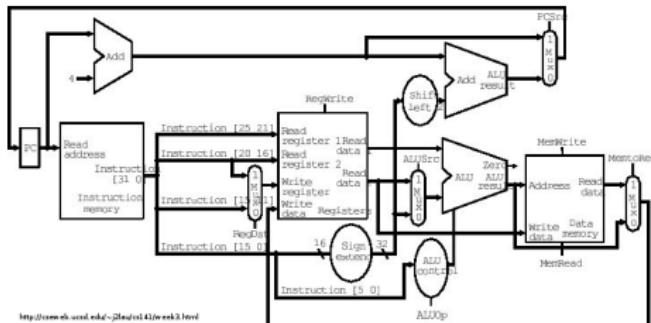


- ▶ Temps de cycle
 - ▶ Période d'horloge P (secondes)
 - ▶ Fréquence d'horloge F (Hz)

$$P = \frac{1}{F}$$



Caractéristiques d'un CPU



- Temps de cycle

- Période d'horloge P (secondes)
- Fréquence d'horloge F (Hz)

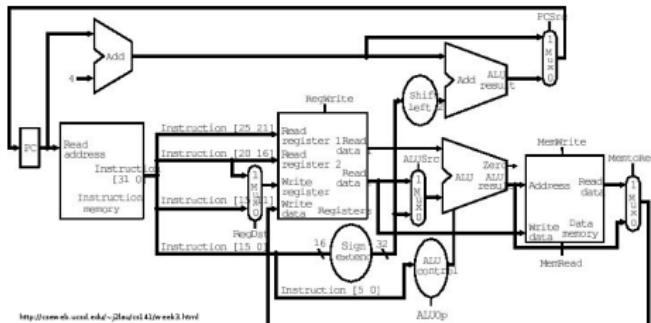
$$P = \frac{1}{F}$$

- Largeur du *chemin de données*

Taille de l'information traitée (32, 64, 128 bits)



Caractéristiques d'un CPU



- Temps de cycle

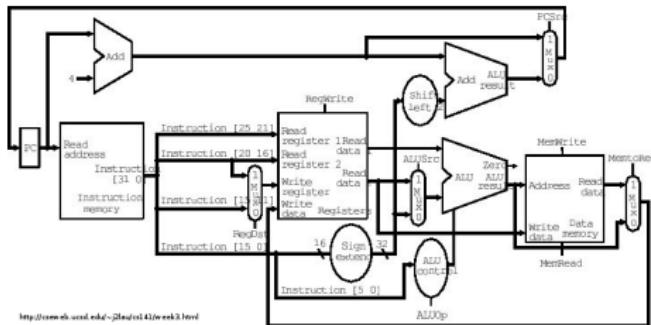
- Période d'horloge P (secondes)
- Fréquence d'horloge F (Hz)

$$P = \frac{1}{F}$$

- Largeur du *chemin de données*

Taille de l'information traitée (32, 64, 128 bits)

- Nombre de cycles par instruction (CPI)



- ## ► Temps de cycle

- ▶ Période d'horloge P (secondes)
 - ▶ Fréquence d'horloge F (Hz)

$$P = \frac{1}{F}$$

- #### ► Largeur du *chemin de données*

Taille de l'information traitée (32, 64, 128 bits)

- ▶ Nombre de cycles par instruction (CPI)
 - ▶ Quantité de travail par instruction



Relations entre métriques

$$\text{Temps CPU} = \#\text{cycles} \times \text{période d'horloge}$$

$$= \frac{\#\text{cycles}}{\text{fréquence d'horloge}}$$

$$\#\text{cycle} = \#\text{instructions} \times \text{CPI}$$

$$\text{Temps CPU} = \frac{\#\text{instructions} \times \text{CPI}}{\text{fréquence d'horloge}}$$



- ▶ **MIPS (*Million of Instructions Per Second*)**

$$\text{MIPS} = \frac{\#\text{instructions}}{\text{temps d'exécution} \times 10^6} = \frac{\text{fréquence d'horloge}}{\text{CPI} \times 10^6}$$

- ▶ Indépendant du jeu d'instructions (quantité de travail effectué)
- ▶ Dépendant du programme considéré

- ▶ **MFLOPS (*Million of FLOating instructions Per Second*)**

$$\text{MFLOPS} = \frac{\#\text{instructions flottantes}}{\text{temps d'exécution} \times 10^6}$$

- ▶ Indépendant du jeu d'instructions flottantes (quantité de travail effectué)
- ▶ Répartition instructions flottantes lentes/rapides ?

Circuits logiques

Circuits combinatoires



Ordinateur *numerique* binaire \rightarrow exprimer toutes les opérations sous forme de **fonctions logiques** :

Fonction n -aire f :

$$f: \{0, 1\}^n \rightarrow \{0, 1\}$$
$$(a_1, \dots, a_n) \mapsto f(a_1, \dots, a_n)$$

complétement ou incomplètement définie



Exemples de fonctions logiques

$f(a_1, a_2, a_3)$			
a_1	a_2	a_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Définition complète

$g(a_1, a_2, a_3)$			
a_1	a_2	a_3	g
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	x
1	0	0	x
1	0	1	1
1	1	0	0
1	1	1	x

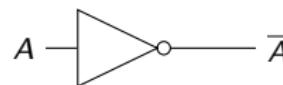
Définition incomplète

Tables de vérité



Fonctions logiques (1/3)

Négation $\neg A$



A	\bar{A}
0	1
1	0

ET logique $A \wedge B$



A	B	$A \times B$
0	0	0
0	1	0
1	0	0
1	1	1

OU logique $A \vee B$



A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1



Associativité de \vee	$A + (B + C) = (A + B) + C$
Associativité de \wedge	$A \times (B \times C) = (A \times B) \times C$
Commutativité de \vee	$A + B = B + A$
Commutativité de \wedge	$A \times B = B \times A$
Distributivité de \wedge p.r. \vee	$A \times (B + C) = (A \times B) + (A \times C)$
Distributivité de \vee p.r. \wedge	$A + (B \times C) = (A + B) \times (A + C)$
Identité pour \vee	$A + 0 = A$
Identité pour \wedge	$A \times 1 = A$
Élément absorbant pour \wedge	$A \times 0 = 0$
Élément absorbant pour \vee	$A + 1 = 1$
Idempotence pour \vee	$A + A = A$
Idempotence pour \wedge	$A \times A = A$
Absorption	$A \times (A + B) = A$
Absorption	$A + (A \times B) = A$
Complémentarité	$A \times \bar{A} = 0$
Complémentarité	$A + \bar{A} = 1$
Complémentarité	$\bar{\bar{A}} = A$
Simplification	$A + (\bar{A} \times B) = A + B$
Simplification	$A \times (\bar{A} + B) = A \times B$
Simplification	$(A \times B) + (A \times \bar{B}) = A$
Simplification	$(A + B) \times (A + \bar{B}) = A$



Dualité : lois de De Morgan

Dualité \vee / \wedge :

$$\overline{A + B} = \overline{A} \times \overline{B}$$

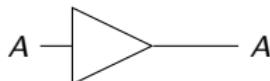
$$\overline{A \times B} = \overline{A} + \overline{B}$$

Passage d'un connecteur à l'autre



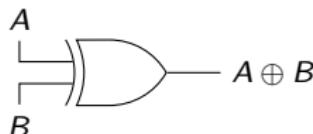
Fonctions logiques (2/3)

Identité A



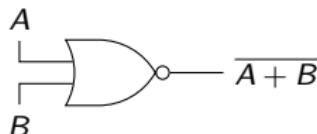
A	A
0	0
1	1

OU exclusif $A \oplus B$



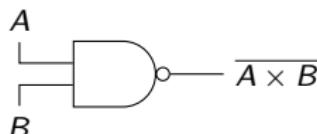
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

« NON OU » (NOR) $\overline{A + B}$



A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

« NON ET » (NAND) $\overline{A \times B}$



A	B	$\overline{A \times B}$
0	0	1
0	1	1
1	0	1
1	1	0



Fonctions logiques (3/3)

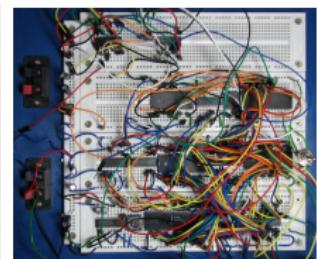
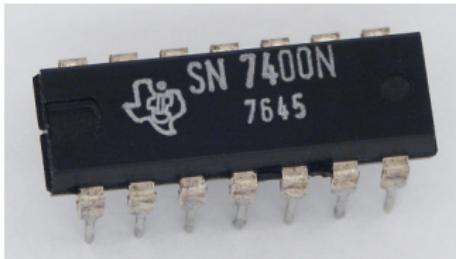
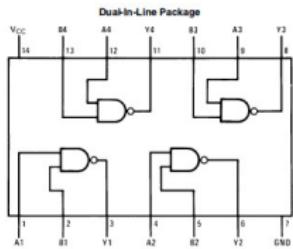
- ▶ Nombreuses autres fonctions logiques :
 - ▶ « NON OU »exclusif
 - ▶ Fonctions ternaires
 - ▶ « NOR » à trois entrées (*Apollo Guidance Computer*)
 - ▶ Fonction de Toffoli
 - ▶ ...
 - ▶ ...

Définition

Un ensemble de connecteurs logiques S est fonctionnellement complet si toute fonction logique peut s'écrire en utilisant uniquement les connecteurs présents dans S .

Ensemble fonctionnellement complet minimal : on ne peut retirer de connecteur de S sans perdre la complétude fonctionnelle

- ▶ L'ensemble (ET, OU, NON) est fonctionnellement complet (non minimal)
- ▶ (OU, NON) : fonctionnellement complet et minimal
- ▶ (NAND) et (NOR) : fonctionnellement complets et minimaux





Produits de sommes / sommes de produits

- ▶ En général, pas de représentation unique d'une fonction logique :

$$AB\bar{C} + CD + C\bar{D} \equiv AB + C$$

$$A \oplus B \equiv A\bar{B} + \bar{A}B$$

- ▶ Nombreuses formulations équivalentes utilisant les différents connecteurs logiques

Forme somme. Somme (OU) de termes

$$A\bar{B}C + A\bar{B}C + A\bar{C}$$

Forme produit. Produit (ET) de termes

$$(A + B + C) \times (A + \bar{B} + C) \times (\bar{A} + \bar{B} + C)$$



Définition (Forme canonique produit de sommes)

Une expression logique est sous forme canonique « produit de sommes » si :

- ▶ Toutes les variables apparaissent dans chaque facteur (A ou \bar{A}) ;
- ▶ Chaque facteur est une disjonction (OU) de variables ;
- ▶ Tous les facteurs sont différents ;
- ▶ Les facteurs sont connectés par des conjonctions (ET).

Exemple :

$$f(A, B, C) = (\bar{A} + \bar{B} + C) \times (\bar{A} + \bar{B} + C)$$

Définition (Forme canonique somme de produits)

Une expression logique est sous forme canonique « somme de produits » si :

- ▶ Toutes les variables apparaissent dans chaque facteur (A ou \bar{A}) ;
- ▶ Chaque facteur est une conjonction (ET) de variables ;
- ▶ Tous les facteurs sont différents ;
- ▶ Les facteurs sont connectés par des disjonctions (OU).

Exemple :

$$f(A, B, C) = \overline{ABC} + \overline{ABC}$$



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$f(A, B, C) =$$



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$f(A, B, C) = \overline{ABC}$$



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$f(A, B, C) = \overline{ABC} + AB\overline{C}$$



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$f(A, B, C) = \overline{ABC} + AB\overline{C} + ABC$$



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$	$\overline{A} \overline{B} \overline{C}$
0	0	0	0	1
0	0	1	0	0
0	1	0	1	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	1	0
1	1	1	1	0

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



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$\overline{\overline{A} \overline{B} \overline{C}} = A + B + C \text{ (De Morgan)}$$

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



Formes canoniques et table de vérité

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$\begin{aligned}f(A, B, C) = & (A + B + C) \times \\& (A + B + \overline{C}) \times \\& (A + \overline{B} + \overline{C}) \times \\& (\overline{A} + B + C) \times \\& (\overline{A} + B + \overline{C})\end{aligned}$$

$$f(A, B, C) = \overline{ABC} + AB\overline{C} + ABC$$



Boîte noire dont les sorties S_i ($i \in \{1, \dots, m\}$) ne dépendent que des entrées E_j ($j \in \{1, \dots, n\}$)



- ▶ Objectif : réaliser un circuit à partir d'une fonction booléenne
- ▶ Méthode :
 1. Calculer une forme canonique à partir de la table de vérité
 2. Simplifier l'expression
 - ▶ Manipulations algébriques
 - ▶ Tableau de Karnaugh
 - ▶ ...
 3. Associer une porte logique à chaque opérateur

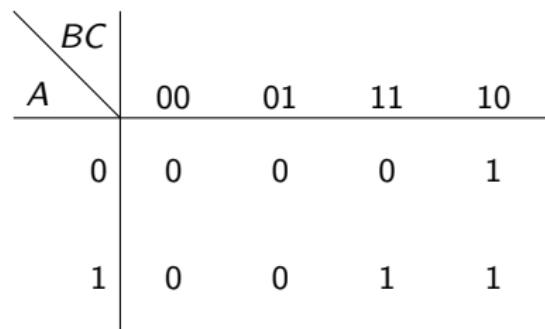
Synthèses particulières :

- ▶ Circuit avec uniquement des NOR
- ▶ Circuit avec uniquement des NAND



Tableau de Karnaugh (1/2)

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

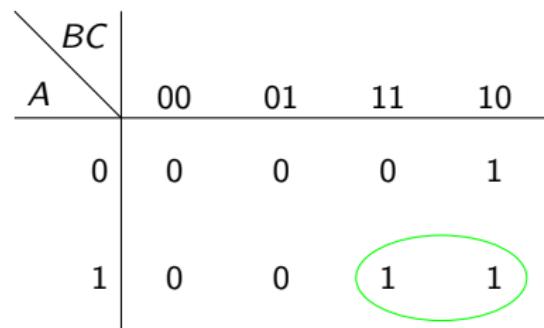


$$f(A, B, C) = \overline{AB}\overline{C} + A\overline{B}\overline{C} + ABC$$



Tableau de Karnaugh (1/2)

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Simplifications :

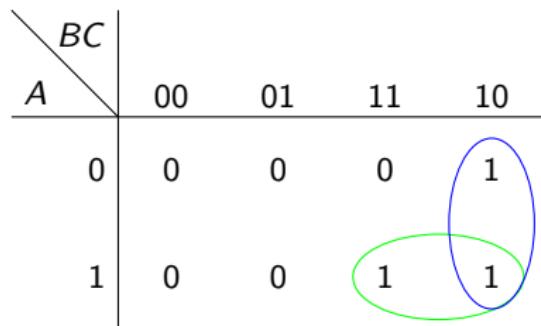
$$\blacktriangleright ABC + AB\bar{C} \equiv AB$$

$$f(A, B, C) = \overline{AB}\overline{C} + AB\overline{C} + ABC$$



Tableau de Karnaugh (1/2)

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Simplifications :

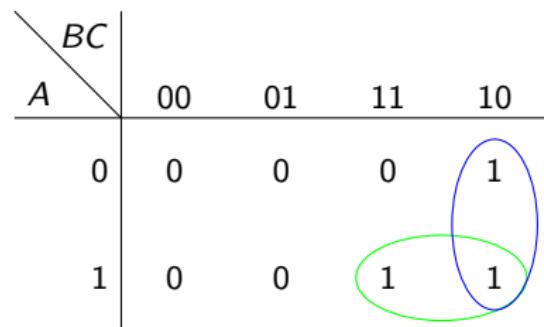
- ▶ $ABC + AB\bar{C} \equiv AB$
- ▶ $\bar{A}B\bar{C} + AB\bar{C} = B\bar{C}$

$$f(A, B, C) = \bar{A}B\bar{C} + AB\bar{C} + ABC$$



Tableau de Karnaugh (1/2)

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



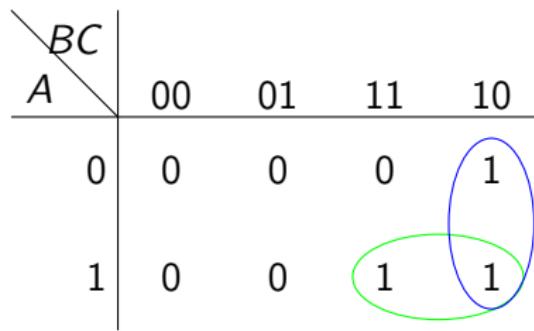
Simplifications :

- ▶ $ABC + AB\bar{C} \equiv AB$
- ▶ $\bar{A}B\bar{C} + AB\bar{C} = B\bar{C}$

$$f(A, B, C) = \bar{A}B\bar{C} + AB\bar{C} + ABC$$

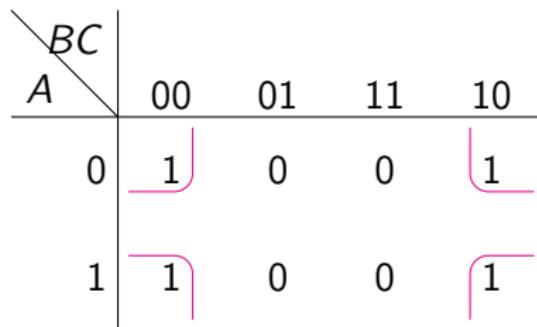
$$\text{D'où : } f(A, B, C) = AB + B\bar{C}$$

Réduction de fonction booléenne $\Sigma(\Pi)$ (minimisation du nombre de termes)



- ▶ Couvrir tous les « 1 »
- ▶ Faire le minimum de groupes
- ▶ Faire des groupes les plus larges possibles (côtés « en 2^x »)

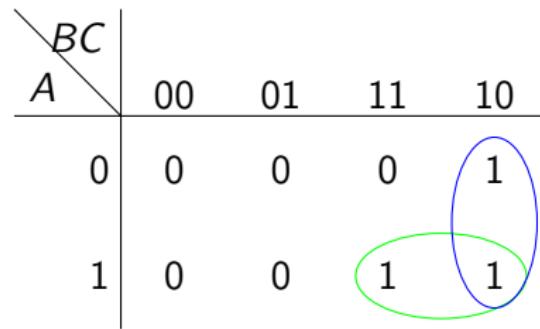
Réduction de fonction booléenne $\Sigma(\Pi)$ (minimisation du nombre de termes)



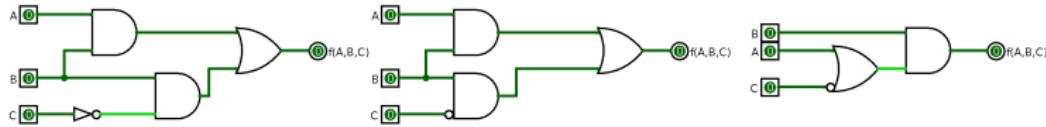
- ▶ Couvrir tous les « 1 »
- ▶ Faire le minimum de groupes
- ▶ Faire des groupes les plus larges possibles (côtés « en 2^x »)



Exemple de synthèse



$$f(A, B, C) = AB + B\bar{C} = B(A + \bar{C})$$





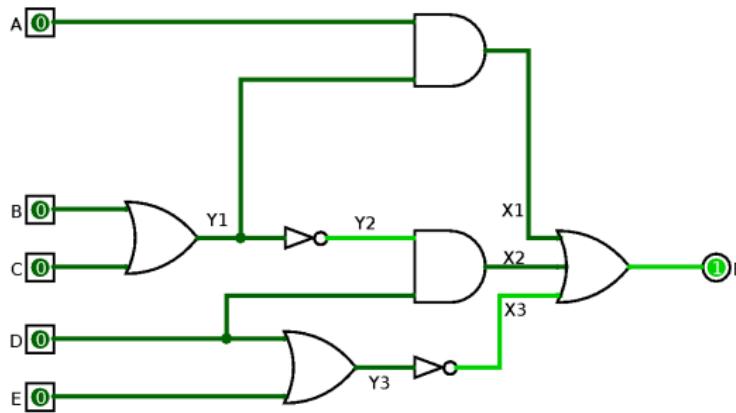
Objectif : obtenir une fonction booléenne à partir d'un schéma logique

Méthode :

1. Partir de la sortie du circuit ;
2. Remplacer la porte par l'opérateur booléen correspondant ;
3. Simplifier ;
4. Remonter le circuit récursivement jusqu'aux entrées.



Exemple d'analyse

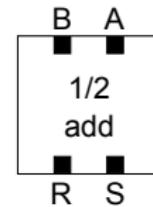


$$\begin{aligned}F &= X_1 + X_2 + X_3 \\&= A \times Y_1 + Y_2 \times D + \overline{Y_3} \\&= A \times (B + C) + \overline{Y_1} \times D + \overline{D + E} \\&= A \times (B + C) + \overline{B + C} \times D + \overline{D + E}\end{aligned}$$



Demi-additionneur 1 bit

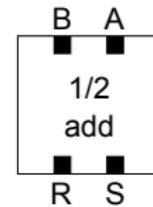
$$\begin{array}{r} & 1 & 0 & 1 & 1 & 0 & \textcolor{red}{1} & \longleftarrow A \\ + & 0 & 1 & 1 & 0 & 1 & \textcolor{red}{1} & \longleftarrow B \\ \hline & 1 & 1 & 1 & 1 & \textcolor{red}{1} & \longleftarrow R \\ & 1 & 0 & 0 & 1 & 0 & 0 & \textcolor{red}{0} & \longleftarrow S \end{array}$$





Demi-additionneur 1 bit

$$\begin{array}{r} & 1 & 0 & 1 & 1 & 0 & \textcolor{red}{1} & \longleftarrow A \\ + & 0 & 1 & 1 & 0 & 1 & \textcolor{red}{1} & \longleftarrow B \\ \hline & 1 & 1 & 1 & 1 & \textcolor{red}{1} & \longleftarrow R \\ & 1 & 0 & 0 & 1 & 0 & 0 & \textcolor{red}{0} & \longleftarrow S \end{array}$$

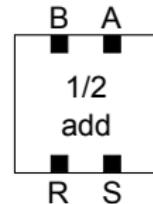


A	B	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

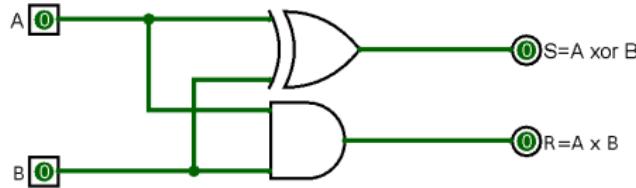


Demi-additionneur 1 bit

$$\begin{array}{r} & 1 & 0 & 1 & 1 & 0 & \textcolor{red}{1} \\ + & 0 & 1 & 1 & 0 & 1 & \textcolor{red}{1} \\ \hline & 1 & 1 & 1 & 1 & \textcolor{red}{1} & \\ & 1 & 0 & 0 & 1 & 0 & 0 & \textcolor{red}{0} \\ & & & & & & & S \end{array}$$



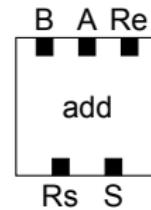
A	B	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1





Additionneur 1 bit

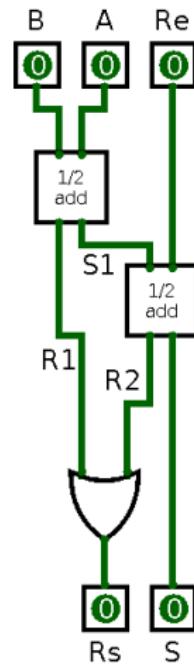
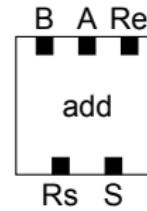
$$\begin{array}{r} & & & & & \textcolor{red}{0} & \longleftarrow R_e \\ & & & & & \textcolor{red}{1} & \longleftarrow A \\ + & 1 & 0 & 1 & 1 & 0 & \textcolor{red}{1} & \longleftarrow B \\ & & 0 & 1 & 1 & 0 & 1 & \textcolor{red}{1} & \longleftarrow R_s \\ \hline & 1 & 1 & 1 & 1 & \textcolor{red}{1} & 0 & \longleftarrow S \end{array}$$



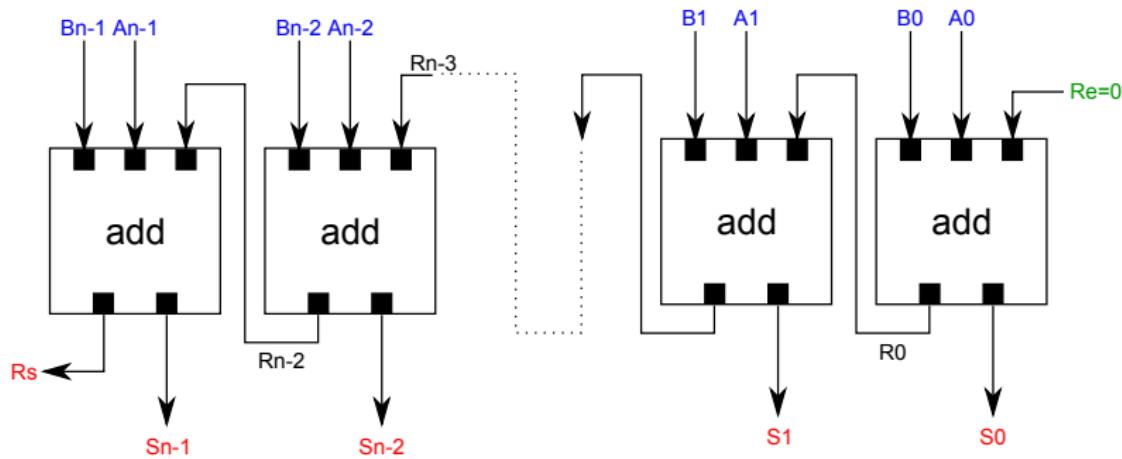
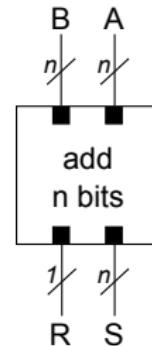


Additionneur 1 bit

$$\begin{array}{r} & & & & & & 0 & \leftarrow R_e \\ & & & & & & 1 & \leftarrow A \\ & & & & & & 1 & \leftarrow B \\ + & 1 & 0 & 1 & 1 & 0 & 1 & \leftarrow R_s \\ & 0 & 1 & 1 & 0 & 1 & 1 & \leftarrow S \\ \hline & 1 & 1 & 1 & 1 & 1 & 0 & \leftarrow R_s \\ & 1 & 0 & 0 & 1 & 0 & 0 & \leftarrow S \end{array}$$



Chaînage des additionneurs 1 bit :

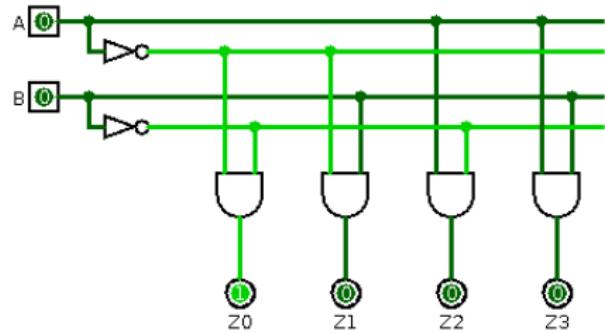




- Une seule sortie à 1 parmi 2^k — la i -ème — avec i sur k bits

Cas $i = 2$:

A	B	Z ₀	Z ₁	Z ₂	Z ₃
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

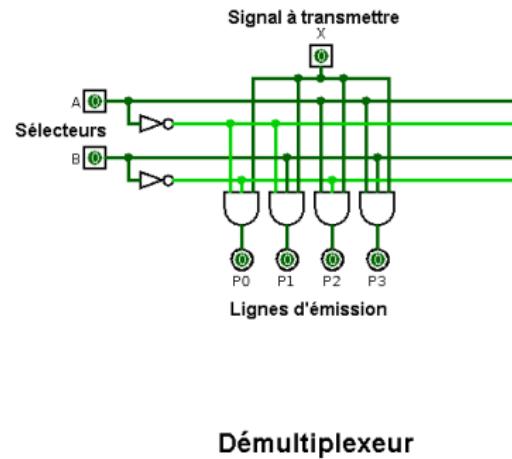
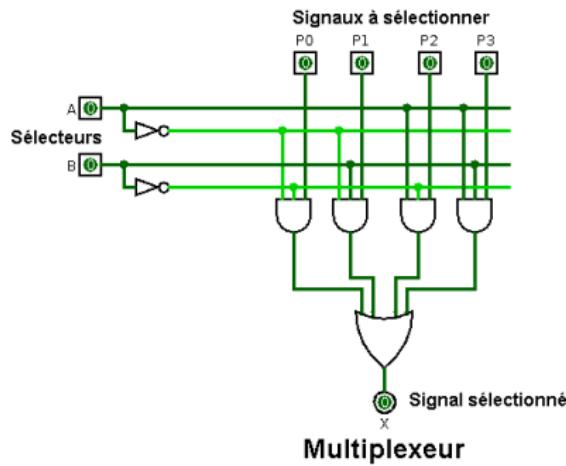




Multiplexeur/démultiplexeur

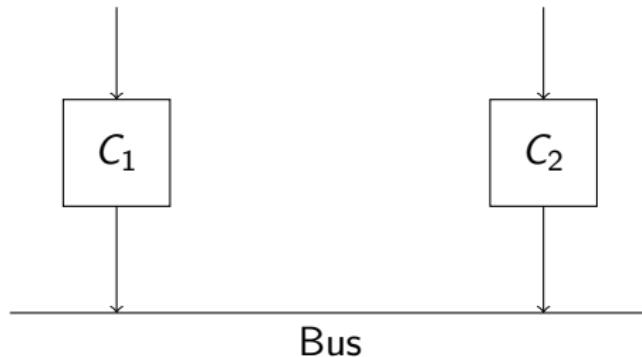
Multiplexage. Choix du signal à envoyer sur une ligne parmi n

Démultiplexage. Envoi d'un signal sur une ligne choisie parmi n



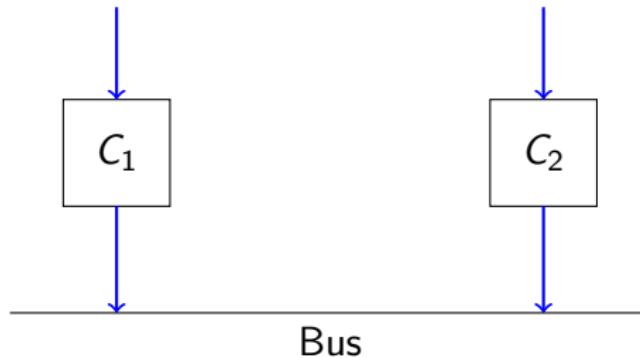


Logique à trois états



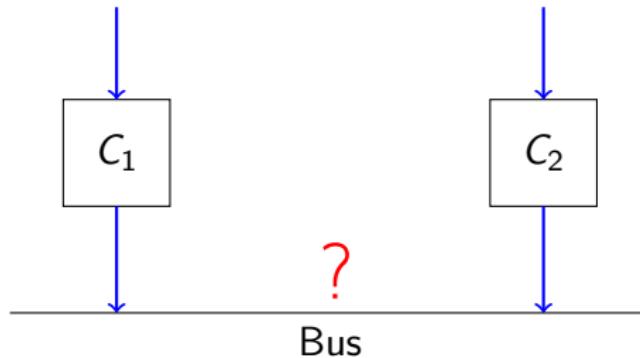


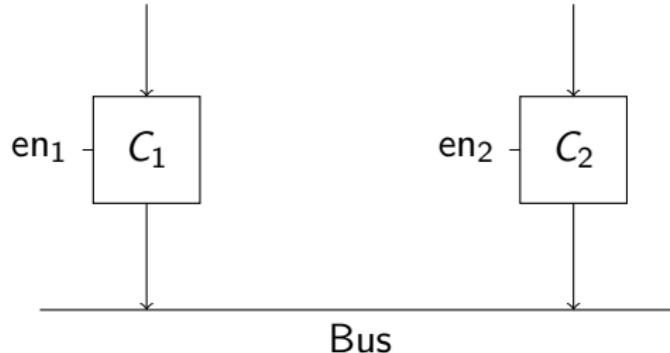
Logique à trois états





Logique à trois états



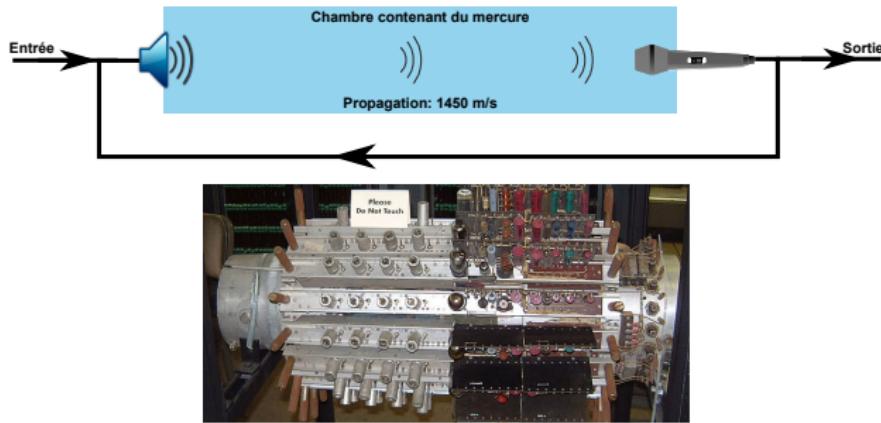


In	en	Out
0	0	Z
1	0	Z
Z	0	Z
0	1	0
1	1	1
Z	1	Z

- ▶ Introduction d'un état à haute impédance Z
- ▶ Dans l'état Z , aucun courant possible en entrée/sortie
- ▶ Possibilité de connecter plusieurs circuits à un bus en s'assurant qu'un seul bit *enable* est à 1

Circuits séquentiels

Mémoire à mercure :



- ▶ Boucle sortie/entrée (*feedback*) pour conserver l'information
- ▶ Utilisation de la vitesse de propagation finie dans le mercure

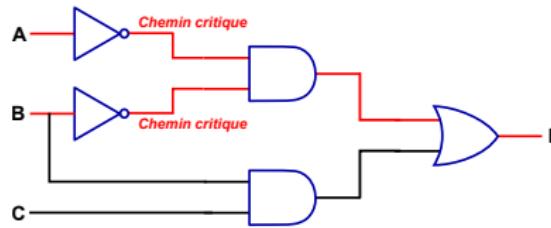
Utilisation de la vitesse de propagation finie dans un circuit électronique

Temps de propagation dans une porte. Temps nécessaire pour que la sortie change lorsque l'entrée d'une porte logique est modifiée (vitesse du courant électrique, capacitance, résistance) ;

Temps de propagation. Temps *maximum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Temps de contamination. Temps *minimum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Chemin critique. Chemin dans le circuit correspondant au temps de propagation.



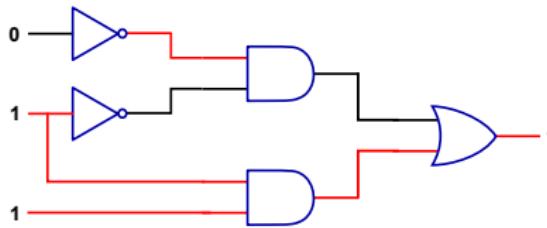
Utilisation de la vitesse de propagation finie dans un circuit électronique

Temps de propagation dans une porte. Temps nécessaire pour que la sortie change lorsque l'entrée d'une porte logique est modifiée (vitesse du courant électrique, capacitance, résistance) ;

Temps de propagation. Temps *maximum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Temps de contamination. Temps *minimum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Chemin critique. Chemin dans le circuit correspondant au temps de propagation.



Glitch. Passage ($A = 0, B = 1, C = 1$) à ($A = 0, B = 0, C = 1$)

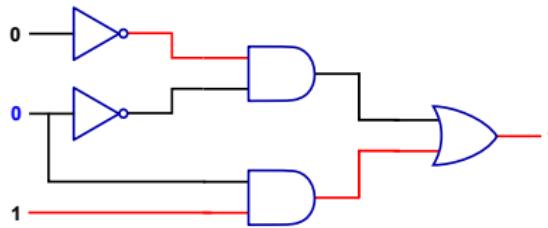
Utilisation de la vitesse de propagation finie dans un circuit électronique

Temps de propagation dans une porte. Temps nécessaire pour que la sortie change lorsque l'entrée d'une porte logique est modifiée (vitesse du courant électrique, capacitance, résistance) ;

Temps de propagation. Temps *maximum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Temps de contamination. Temps *minimum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Chemin critique. Chemin dans le circuit correspondant au temps de propagation.



Glitch. Passage ($A = 0, B = 1, C = 1$) à ($A = 0, B = 0, C = 1$)

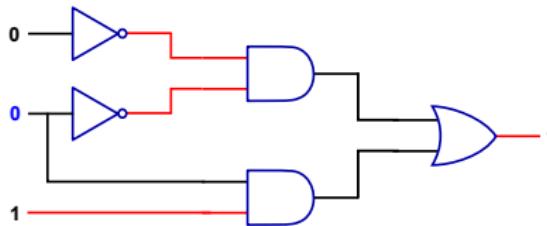
Utilisation de la vitesse de propagation finie dans un circuit électronique

Temps de propagation dans une porte. Temps nécessaire pour que la sortie change lorsque l'entrée d'une porte logique est modifiée (vitesse du courant électrique, capacitance, résistance) ;

Temps de propagation. Temps *maximum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Temps de contamination. Temps *minimum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Chemin critique. Chemin dans le circuit correspondant au temps de propagation.



Glitch. Passage ($A = 0, B = 1, C = 1$) à ($A = 0, B = 0, C = 1$)

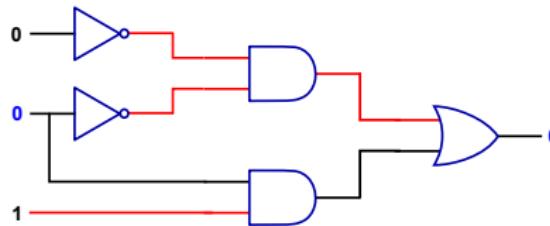
Utilisation de la vitesse de propagation finie dans un circuit électronique

Temps de propagation dans une porte. Temps nécessaire pour que la sortie change lorsque l'entrée d'une porte logique est modifiée (vitesse du courant électrique, capacitance, résistance) ;

Temps de propagation. Temps *maximum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Temps de contamination. Temps *minimum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Chemin critique. Chemin dans le circuit correspondant au temps de propagation.



Glitch. Passage ($A = 0, B = 1, C = 1$) à ($A = 0, B = 0, C = 1$)

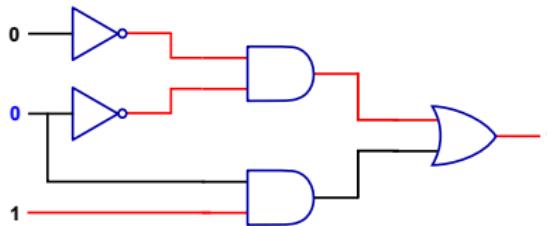
Utilisation de la vitesse de propagation finie dans un circuit électronique

Temps de propagation dans une porte. Temps nécessaire pour que la sortie change lorsque l'entrée d'une porte logique est modifiée (vitesse du courant électrique, capacitance, résistance) ;

Temps de propagation. Temps *maximum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Temps de contamination. Temps *minimum* pour que la sortie d'un circuit reflète les modifications appliquées à ses entrées ;

Chemin critique. Chemin dans le circuit correspondant au temps de propagation.

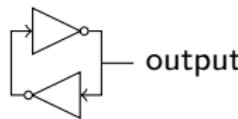


Glitch. Passage ($A = 0, B = 1, C = 1$) à ($A = 0, B = 0, C = 1$)



Mémorisation dans un circuit logique

Conservation de l'information dans un circuit logique ?



- ▶ Initialisation ?
- ▶ Modification de la valeur stockée ?

Verrou avec deux NORs

A	B	$A + B$
0	0	1
0	1	0
1	0	0
1	1	0



Mémorisation dans un circuit logique

Conservation de l'information dans un circuit logique ?

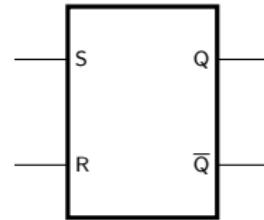
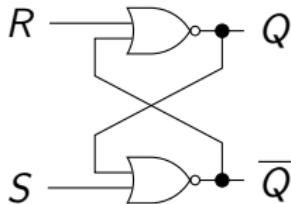


- ▶ Initialisation ?
- ▶ Modification de la valeur stockée ?

Verrou avec deux NORs

A	B	$A + B$
0	0	1
0	1	0
1	0	0
1	1	0

- ▶ Sortie du circuit
- ▶ Set : force la sortie à 1
- ▶ Reset : force la sortie à 0
- ▶ Deux états stables pour $A = 0 \wedge B = 0$
- ▶ Deux états instables lors du passage simultané $A: 1 \rightarrow 0 || B: 1 \rightarrow 0$



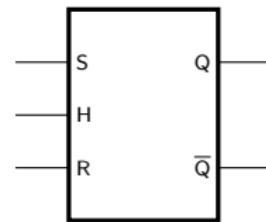
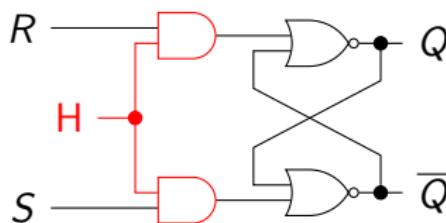
- ▶ Asynchrone : modifications des entrées immédiatement répercutées
- ▶ Sauvegarde d'un bit (mise à 1 par *Set* et à 0 par *Reset*)

Table caractéristique

R	S	Q'	Commentaire
0	0	Q	Sauvegarde état
0	1	1	<i>Set</i>
1	0	0	<i>Reset</i>
1	1	X	Non autorisé

Table d'excitation

$Q \rightarrow Q'$	R	S
$0 \rightarrow 0$	X	0
$0 \rightarrow 1$	0	1
$1 \rightarrow 0$	1	0
$1 \rightarrow 1$	0	X



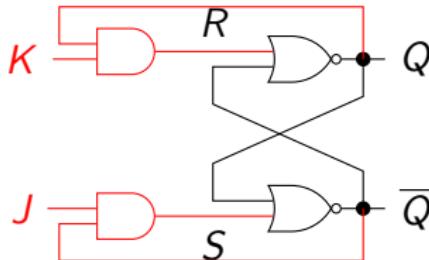
- ▶ Bit de contrôle H
- ▶ Modifications des entrées répercutées pour $H = 1$

Table caractéristique

H	R	S	Q'	Commentaire
0	X	X	Q	Sauvegarde état
1	0	0	Q	Sauvegarde état
1	0	1	1	<i>Set</i>
1	1	0	0	<i>Reset</i>
1	1	1	X	Non autorisé

Table d'excitation

$Q \rightarrow Q'$	R	S
0 → 0	X	0
0 → 1	0	1
1 → 0	1	0
1 → 1	0	X



- ▶ But : supprimer le cas indéterminé de la bascule RS
- ▶ R et S jamais simultanément à 1

Table caractéristique

J	K	Q'	Commentaire
0	0	Q	Sauvegarde état
0	1	0	<i>Reset</i>
1	0	1	<i>Set</i>
1	1	\bar{Q}	Inversion

Table d'excitation

$Q \rightarrow Q'$	J	K
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 0$	X	1
$1 \rightarrow 1$	X	0

$$\text{Équation caractéristique : } Q' = J\bar{Q} + \bar{K}Q$$

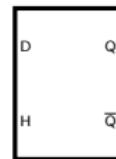
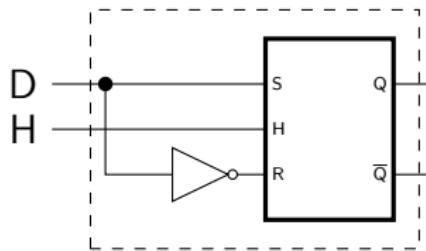


Table caractéristique

H	D	Q'
0	X	Q
1	0	0
1	1	1

Table d'excitation

$Q \rightarrow Q'$	D
0 \rightarrow 0	0
0 \rightarrow 1	1
1 \rightarrow 0	0
1 \rightarrow 1	1

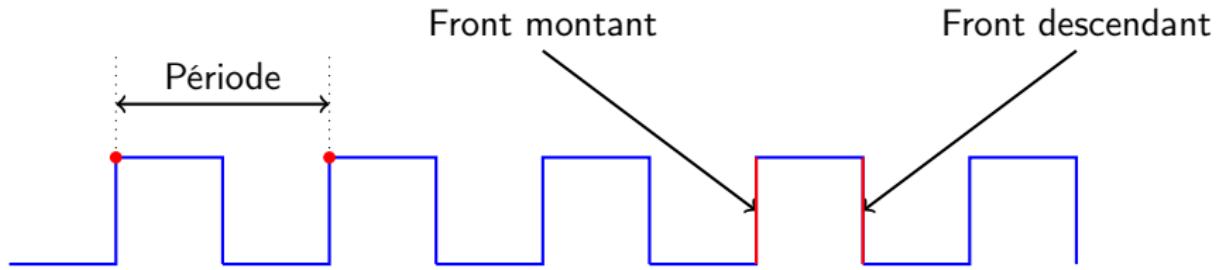
- ▶ Mémoire de 1 bit
- ▶ Plus de risque d'état interdit



- ▶ Verrou (*latch*)
 - ▶ Sorties modifiées dès que les entrées changent
 - ▶ Problème si valeurs des entrées non disponibles toutes en même temps (*Glitch*)
- ▶ Verrou protégé (*gated latch*)
 - ▶ Ajout d'un signal H pour restreindre le moment de prise en compte des modifications
 - ▶ *level triggered latch* (sur 0 ou 1)
- ▶ Bascule (*flip-flop*)
 - ▶ Utilisation d'un signal périodique carré (*horloge*)
 - ▶ Les sorties évoluent en fonction des entrées seulement « au signal d'horloge »
 - ▶ *edge triggered flip-flop* (sur front montant ou descendant)

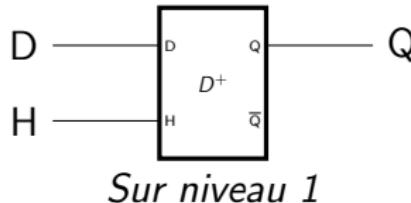


Signal périodique carré oscillant entre 0 et 1

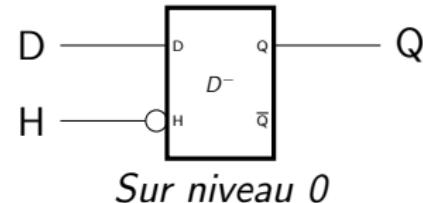




Exemple : le verrou protégé D

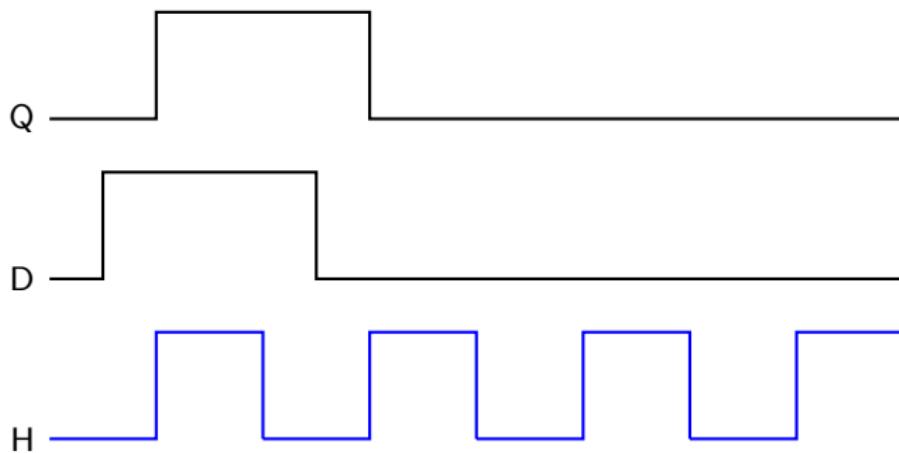


Sur niveau 1



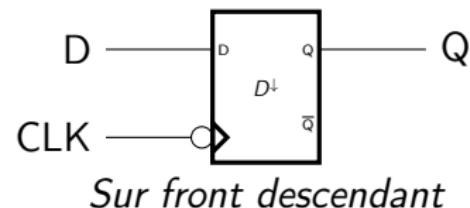
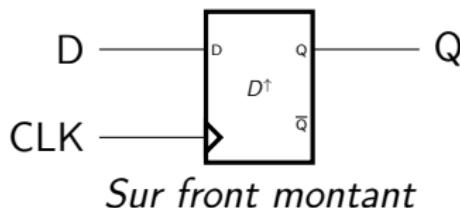
Sur niveau 0

Chronogramme pour D^+ :

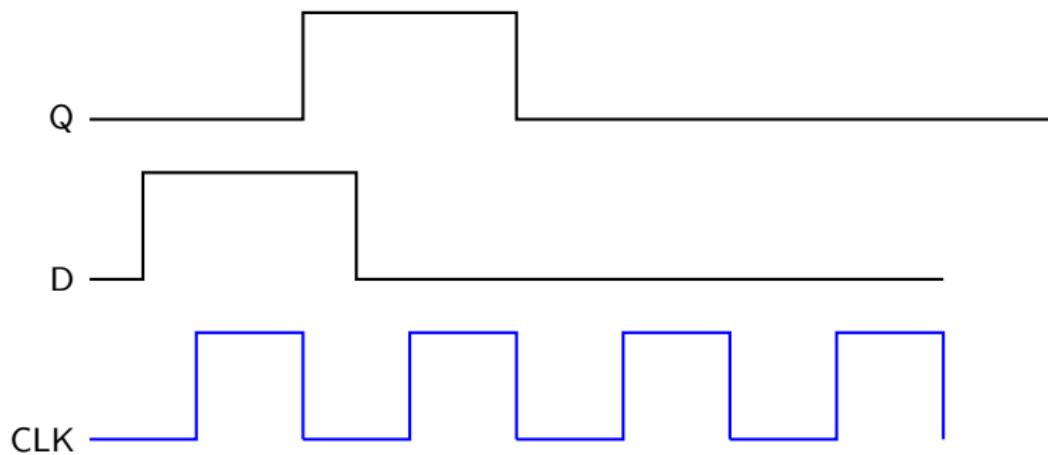




Exemple : la bascule D



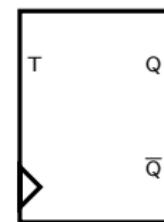
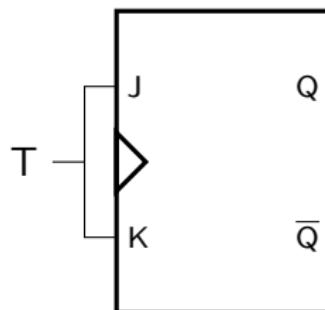
Chronogramme pour D^\downarrow :





Bascule D sur front descendant

Implémentation

*Table caractéristique*

T	Q'
0	Q
1	\bar{Q}

Table d'excitation

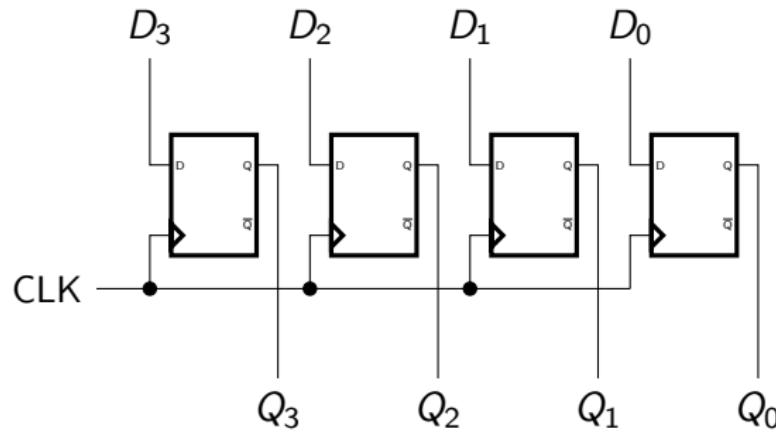
$Q \rightarrow Q'$	T
0 → 0	0
0 → 1	1
1 → 0	1
1 → 1	0

$$\text{Équation caractéristique : } Q' = T \oplus Q$$



Registre à lecture/écriture parallèle

Exemple de registre sur 4 bits :

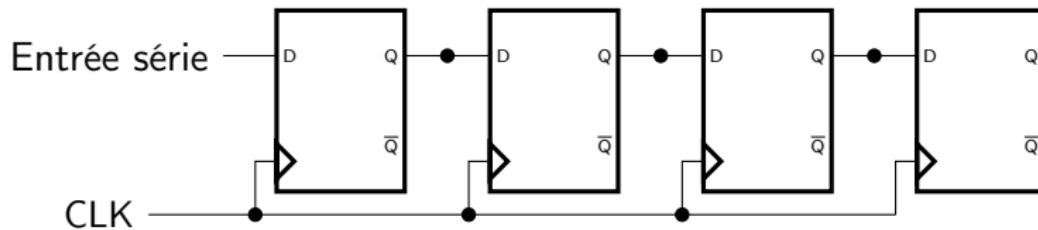




Registre à décalage

Lecture/écriture en série.

Exemple de registre sur 4 bits :



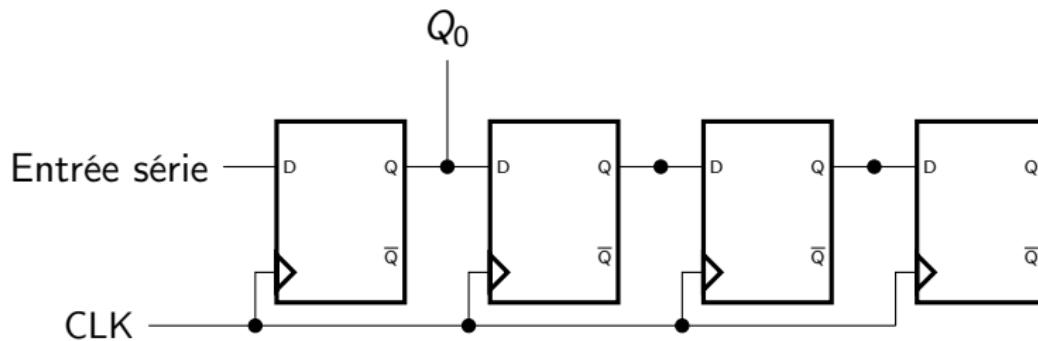
Implémentation



Registre à décalage

Lecture/écriture en série.

Exemple de registre sur 4 bits :



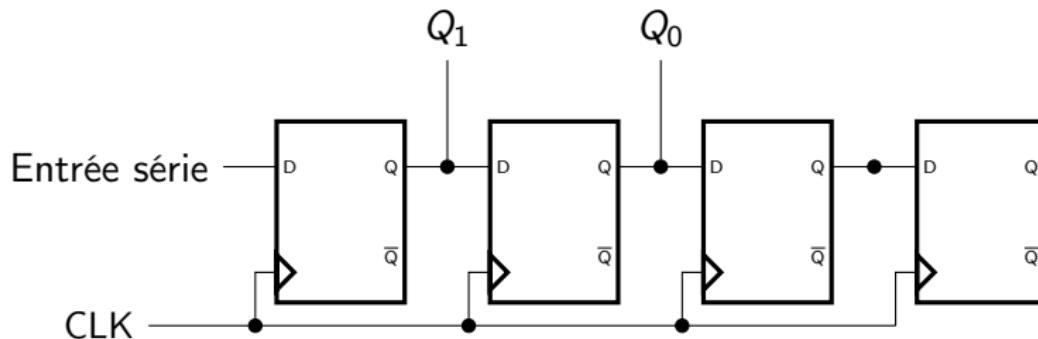
Implémentation



Registre à décalage

Lecture/écriture en série.

Exemple de registre sur 4 bits :



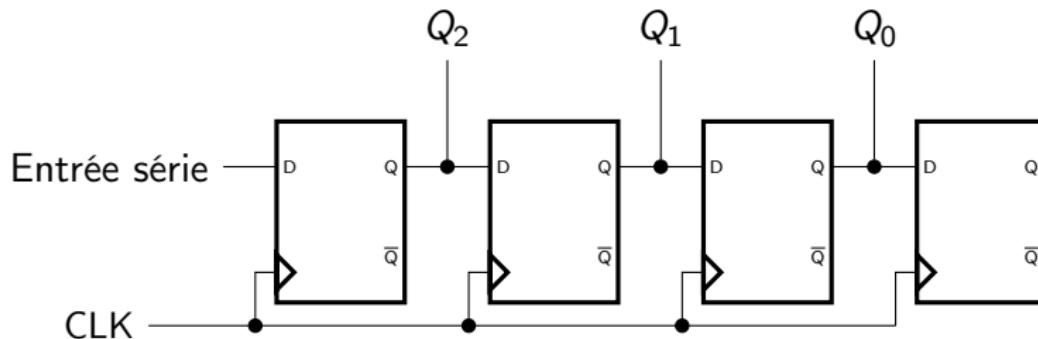
Implémentation



Registre à décalage

Lecture/écriture en série.

Exemple de registre sur 4 bits :



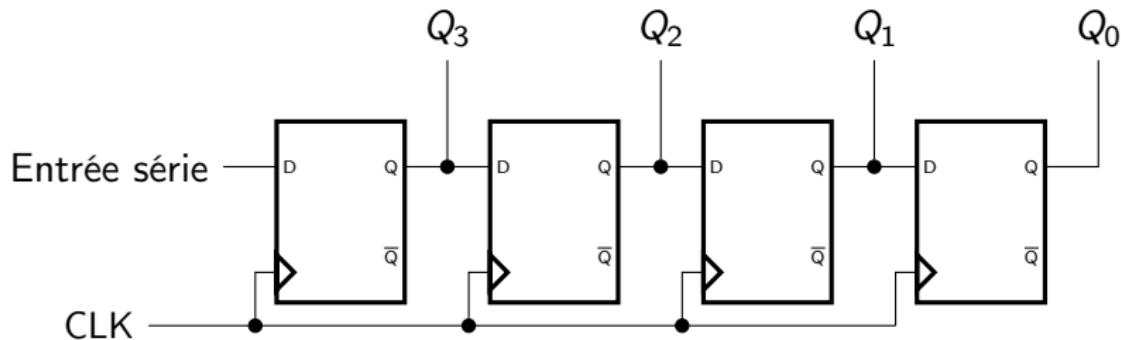
Implémentation



Registre à décalage

Lecture/écriture en série.

Exemple de registre sur 4 bits :



Implémentation

Comptage en binaire des occurrences d'un phénomène (e.g. tic d'horloge)

Compteur asynchrone. Impulsions appliquées sur la première bascule (propagation d'une bascule à l'autre)

Compteur synchrone. Impulsions appliquées sur toutes les bascules en parallèle



Compteur asynchrone

Exemple sur 3 bits



Compteur synchrone

Exemple sur 4 bits



Distributeur de boissons à 3 € :

- Un senseur de pièces sur 2 bits :

I	J	Événement
0	x	Pas de pièce
1	0	1 €
1	1	2 €

- Deux sorties :

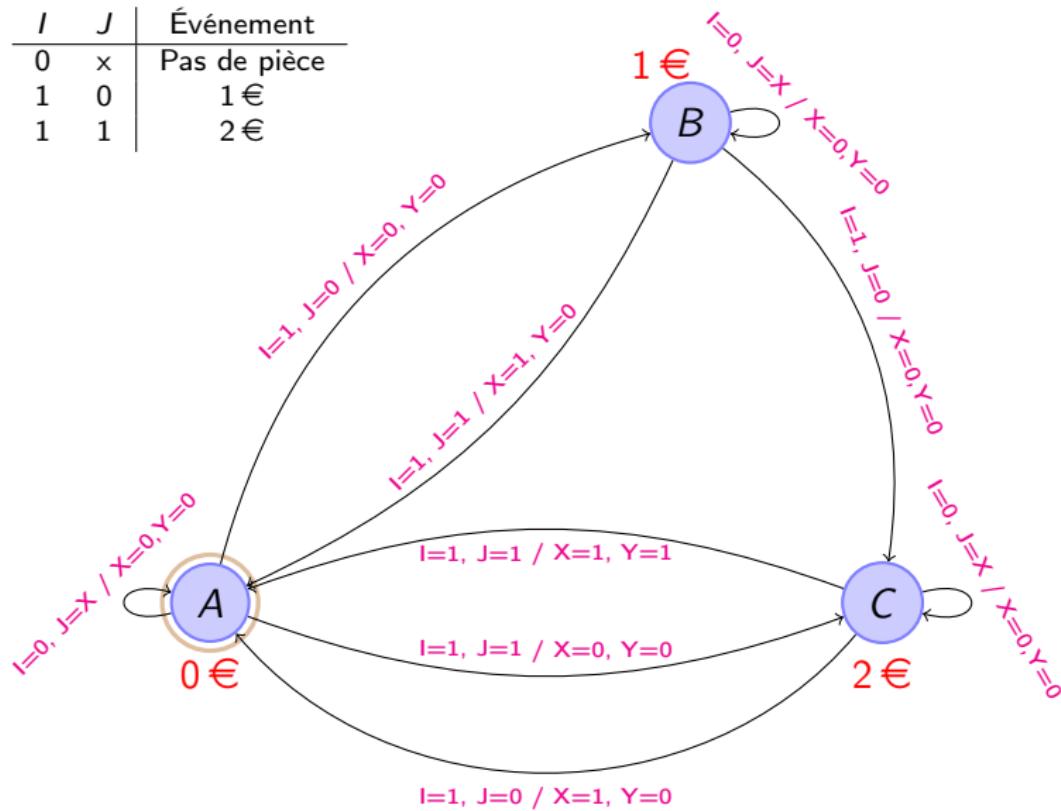
- X à 1 pour délivrer une boisson si la somme entrée est d'au moins 3 €
- Y à 1 pour rendre 1 € si l'utilisateur a inséré 4 €

Réalisation du circuit logique de contrôle ?



Modélisation par automate

I	J	Événement
0	x	Pas de pièce
1	0	1€
1	1	2€





Automate fini déterministe :

- ▶ Entrées E
- ▶ Sorties S
- ▶ Fonction du temps $Q(t)$ (états de l'automate)
- ▶ Fonctions de sorties F et de transitions G

avec :

- ▶ $Q(t)$ déterminé par l'histoire de l'automate avant t
- ▶ $S(t+1) = F(Q(t), E(t))$ (**automate de Mealy**)
- ▶ $S(t+1) = F(Q(t))$ (**automate de Moore**)
- ▶ $Q(t+1) = G(Q(t), E(t))$



1. Description du système sous forme d'automate
2. Choix d'un codage pour les états [et les entrées/sorties]
3. Détermination des tables pour les fonctions de sortie F et de transition G
4. Traduction de la fonction de transition en fonction des bascules utilisées (table d'excitation)
5. Réalisation du circuit de contrôle implémentant l'automate à l'aide de portes logiques et de bascules



Problème du distributeur

Codage des états :	codage dense		codage à jetons
	A	B	C
	00		001
	01		010
	10		100

Fonction de sortie
 $(X, Y) = F(I, J, q_0, q_1)$

$$\begin{aligned} X &= q_1 IJ + q_0 I \\ Y &= q_0 IJ \end{aligned}$$

Fonction de transition
 $q'_0 q'_1 = G(I, J, q_0, q_1) :$

$$\begin{aligned} q'_0 &= \overline{q_0} \overline{q_1} IJ + q_1 I\bar{J} + q_0 \bar{I} \\ q'_1 &= \overline{q_0} \overline{q_1} I\bar{J} + q_0 \bar{I} \end{aligned}$$



Interprétation en termes de bascules D

Fonction de transition
 $q'_0 q'_1 = G(I, J, q_0, q_1)$:

$$q'_0 = \overline{q_0} \overline{q_1} IJ + q_1 I\bar{J} + q_0 \bar{I}$$

$$q'_1 = \overline{q_0} \overline{q_1} I\bar{J} + q_0 \bar{I}$$

Table d'excitation

$Q \rightarrow Q'$	D
$0 \rightarrow 0$	0
$0 \rightarrow 1$	1
$1 \rightarrow 0$	0
$1 \rightarrow 1$	1

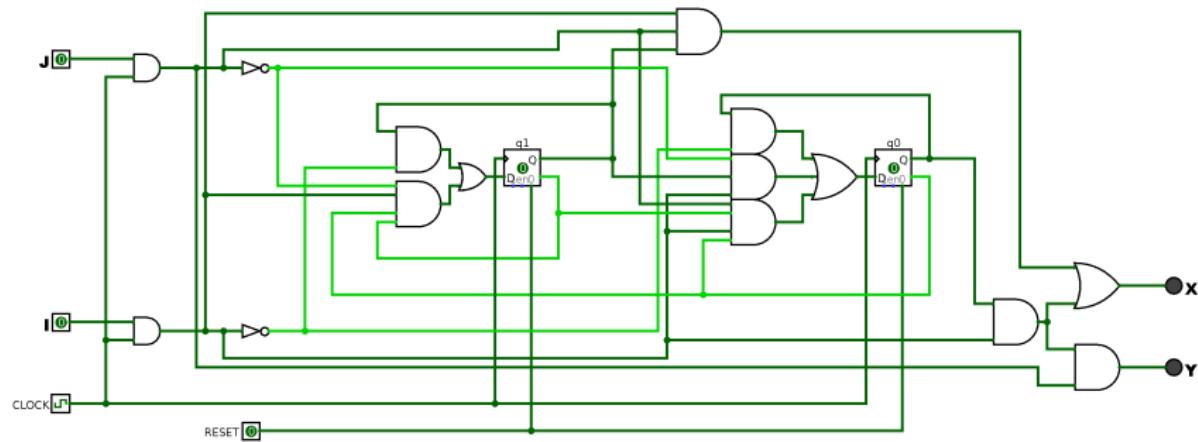
Changements d'états avec des bascules D :

$$D_0 = \overline{q_0} \overline{q_1} IJ + q_1 I\bar{J} + q_0 \bar{I}$$

$$D_1 = \overline{q_0} \overline{q_1} I\bar{J} + q_0 \bar{I}$$



Implémentation avec des bascules D



Simulation



Interprétation en termes de bascules JK

Fonction de transition
 $q'_0 q'_1 = G(I, J, q_0, q_1)$:

$$q'_0 = \overline{q_0} \overline{q_1} IJ + q_1 I\bar{J} + q_0 \bar{I}$$

$$q'_1 = \overline{q_0} \overline{q_1} I\bar{J} + q_0 \bar{I}$$

		Table d'excitation	
		<i>Q</i> → <i>Q'</i>	<i>J</i>
		0 → 0	0
		0 → 1	1
		1 → 0	X
		1 → 1	X
			0

Changements d'états avec des bascules JK :

$$J_0 = IJ\overline{q_1} + I\bar{J}q_1$$

$$K_0 = I$$

$$J_1 = I\bar{J}\overline{q_0} \overline{q_1}$$

$$K_1 = I$$

Langage d'assemblage MIPS



De traduction en traduction

Langage C

```
int fmax(int (*f)(int, int),  
         int a, int b)  
{  
    if (a >= b) {  
        return f(a,b);  
    } else {  
        return f(b,a);  
    }  
}
```

Compilation

Assembleur MIPS

```
fmax:  
    sub $sp, $sp, 8  
    sw $ra, 0($sp)  
  
    move $t0, $a0  
if:   blt $a1, $a2, else  
then:  
    move $a0, $a1  
    move $a1, $a2  
    jalr $t0  
    b endif  
else:  
    move $a0, $a2  
    jalr $t0  
endif:  
    lw $ra, 0($sp)  
    add $sp, $sp, 8  
    jr $ra
```

Code machine

```
20 00 01 20  
22 e8 a1 03  
14 00 bf af  
10 00 be af  
...
```

Assemblage



De traduction en traduction

Langage C

```
int fmax(int (*f)(int, int),  
         int a, int b)  
{  
    if (a >= b) {  
        return f(a,b);  
    } else {  
        return f(b,a);  
    }  
}
```

Compilation

Assembleur MIPS

```
fmax:  
    sub $sp, $sp, 8  
    sw $ra, 0($sp)  
  
    move $t0, $a0  
    if: blt $a1, $a2, else  
    then:  
        move $a0, $a1  
        move $a1, $a2  
        jalr $t0  
        b endif  
    else:  
        move $a0, $a2  
        jalr $t0  
    endif:  
    lw $ra, 0($sp)  
    add $sp, $sp, 8  
    jr $ra
```

Code machine

```
20 00 01 20  
22 e8 a1 03  
14 00 bf af  
10 00 be af  
...
```

Assemblage



Intel x86

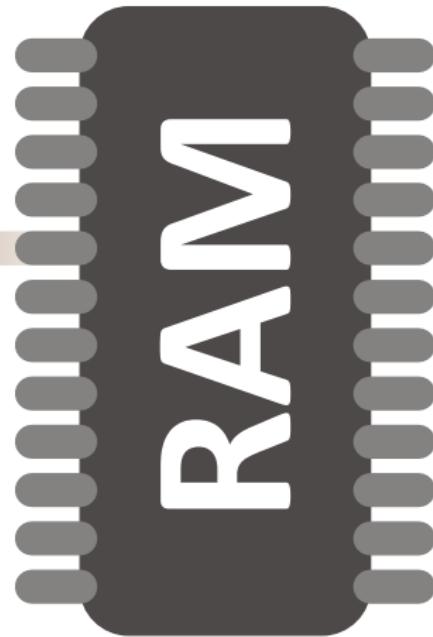
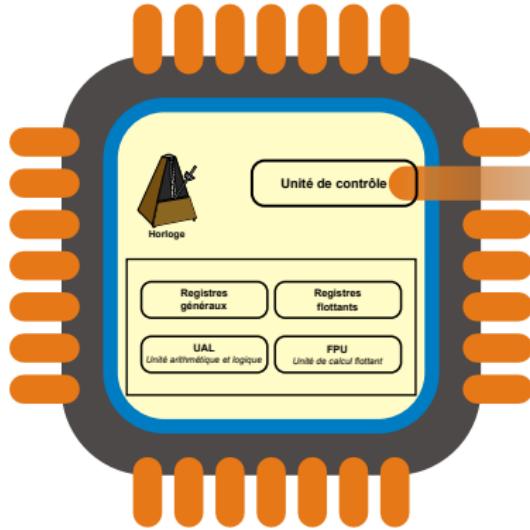
- Architecture CISC
(peu de registres, beaucoup d'instructions)
- **Langage d'assemblage complexe**
- Usage : ordinateur de bureau, serveur

MIPS

- Architecture RISC **pipelinée**
(beaucoup de registres, peu d'instructions)
- **Langage d'assemblage simple**
- Usage : ordinateurs de bureaux (SGI),
lecteurs DVD, appareils photos, imprimantes,
consoles de jeux (Sony PlayStation PSX), ...



Processeur / mémoire





- ▶ Chaque processeur possède son propre jeu d'instructions machine directement compréhensibles par lui
- ▶ Chaque instruction machine possède un identifiant numérique : l'*opcode*
- ▶ Programmation directe du processeur avec les instructions machines :
 - ▶ Difficile et long
 - ▶ Compréhension quasi-impossible
- ➡ Utilisation d'un langage de plus haut niveau associant un mnémonique à chaque instruction machine : le *langage d'assemblage* (aka **assembleur**)



Avantages :

- ▶ Accès à *toutes* les possibilités de la machine
- ▶ Vitesse d'exécution du code
(modulo une parfaite connaissance de la machine)
- ▶ Petite taille du code généré
- ▶ Permet une meilleure connaissance du fonctionnement de la machine

Inconvénients :

- ▶ Temps de codage plus long
- ▶ Fastidieux
- ▶ Pas de structures évoluées
- ▶ Gardes-fous minimaux
- ▶ Absence de portabilité
(y compris sur même machine avec système d'exploitation différent)



Exemple complet

hello.asm

```
.data  
msg: .asciiz 'Hello, world!\n'  
  
.text  
.globl __start  
  
__start:  
    li $v0, 4  
    la $a0, msg  
    syscall  
  
    li $v0, 10  
    syscall
```

Notion de segment

Déclaration de variable + initialisation

Nom du prog. principal

Début de programme

\$v0 = 4

\$a0 = @msg

Appel du service print_string

\$v0 = 10

Appel du service exit



Organisation du fichier source :

- ▶ Directives d'assemblage
 - ▶ Découpage en 2 segments symboliques :
 - ▶ `.text` : segment contenant le code
 - ▶ `.data` : segment contenant les données initialisées
 - ▶ Définition de « variables »
- ▶ Code des procédures utilisées
- ▶ Programme principal `.globl` nommé communément `__start`
- ▶ Commentaires (commencés par '#')



32 registres de 32 bits (+ registres flottants)

Numéro	Nom	Usage
\$0	\$zero	Constante zéro
\$1	\$at	Expansion de pseudo-ops
\$2–\$3	\$v0–\$v1	Résultat de fonction
\$4–\$7	\$a0–\$a3	Paramètres de fonction
\$8–\$15, \$24, \$25	\$t0–\$t9	Temporaire
\$16–\$23	\$s0–\$s7	Sauvegardé
\$26–\$27	\$k0–\$k1	Réserve
\$28	\$gp	Pointeur global
\$29	\$sp	Pointeur de pile
\$30	\$fp/\$s8	Pointeur de cadre de pile
\$31	\$ra	Adresse de retour

La plupart des usages correspondent à des *conventions*



- ▶ Stockage :
 - ▶ des opérandes lors d'opérations logiques ou arithmétiques (entières)
 - ▶ des opérandes pour des calculs d'adresses
 - ▶ des pointeurs vers la mémoire

Pas de registre de statut sur le MIPS (overflow, carry, ...)



label: mnémonique arg₁, arg₂, arg₃

label. Identificateur suivi d'un ':' identifiant une position en mémoire dans le code ou les données

mnémonique. Nom réservé pour une *classe* d'opcodes ayant la même fonction

arg_i. Entre 0 et 3 opérandes

- ▶ Notation Intel : destination = opérande de gauche



```
.data  
val: .word 14  
  
.text  
debut: lw $t0, val  
li $t1, 7  
bgt $t0, $t1, debut # saut si $t0 > $t1
```

54	...
154	xx
153	xx
152	xx
151	0E
150	...
32	
28	...
24	bgt \$t0, \$t1, -12
20	li \$t1, 7
16	lw \$t0, @151
12	...

MEMOIRE



Types d'opérandes d'une instruction

- ▶ *Registres.* Correspond à un registre du processeur
- ▶ *Mémoire.* Correspond à une zone de la mémoire identifiée par son adresse logique
- ▶ *Immédiat.* Constante numérique (stockée dans l'instruction et non dans les données)



- ▶ Syntaxe d'un label :
 - ▶ Mot composé de lettres, chiffres, '_', ':'
 - ▶ Le premier caractère ne doit pas être un chiffre
- ▶ Constantes numériques :
 - ▶ Entiers :

```
100 # décimale  
0xa2 # hexadécimal
```
- ▶ Caractères et chaînes :

```
'h'      # caractère 'h'  
"hello" # chaîne 'hello'
```



Déclarations de « cases initialisées » (dans .data) :

```
.data
# Déclaration et initialisation à 40 de deux octets
L1:    .half  40
# Déclaration du tableau d'octets (85, 86, 87)
T:     .byte 0x55, 0x56, 0x57
# Déclaration d'un double
Pi:    .double 3.14159
```

La directive donne la taille à réserver *par entrée* :

Type	directive	taille (bits)
octet	.byte	8
demi-mot	.half	16
mot	.word	32
simple	.float	32
double	.double	64
espace	.space <i>n</i>	$8n$

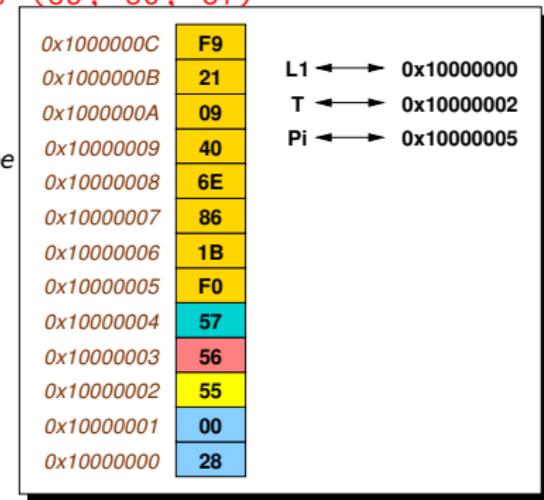


Déclarations de « cases initialisées » (dans .data) :

```
.data
# Déclaration et initialisation à 40 de deux octets
L1: .half 40
# Déclaration du tableau d'octets (85, 86, 87)
T: .byte 0x55, 0x56, 0x57
# Déclaration d'un double
Pi: .double 3.14159
```

La directive donne la taille à réserver par entrée

Type	directive	taille (bits)
octet	.byte	8
demi-mot	.half	16
mot	.word	32
simple	.float	32
double	.double	64
espace	.space n	8n





- ▶ Chargements (de/vers un registre)

```
li $t0, 4  
la $t1, tab  
sb $t0, var  
move $t1, $t2  
sw $a0, T
```

- ▶ Branchements / Sauts

```
b debut  
beq $t0, $v0, finsi  
jal proc
```

- ▶ Instructions logiques & arithmétiques

```
and $t0, $t1, $t4  
add $t2, $t0, $t3
```



Instruction move : move dest, source

- ▶ source : immédiat, registre général, registre de segment, adresse mémoire
- ▶ destination : registre général, registre de segment, adresse mémoire
- ▶ Deux opérandes de même taille

Exemples :

```
move $t3, $t0 # $t3 = $t0
lw $t1, ($t2) # $t1 = 32 bits à l'adresse donnée par $t2
lb $t1, 4($t3) # $t1 = 8 bits à l'adresse donnée par $t3+4
sh $t2, ($t0) # Stocke 16 bits de $t2 à l'adresse donnée par $t0
```

Instruction move : move dest, source

- source : immédiat, registre général, registre de segment, adresse mémoire
- destination : registre général, registre de segment, adresse mémoire
- Deux opérandes de mêm

Exemples :

```
move $t3, $t0 # $t3 = $t0  
lw $t1, ($t2) # $t1 = 32 bits  
lb $t1, 4($t3) # $t1 = 8 bits à  
sh $t2, ($t0) # Stocke 16 bits
```

0x10000009	40	\$t0	\$t1
0x10000008	C1	0x10000000	0x3699A21C
0x10000007	A2	\$t2	\$t3
0x10000006	89	0x10000005	0xF46A2010
0x10000005	F2		
0x10000004	AA		
0x10000003	5E		
0x10000002	4F		
0x10000001	12		
0x10000000	30		

Instruction move : move dest, source

- source : immédiat, registre général, registre de segment, adresse mémoire
- destination : registre général, registre de segment, adresse mémoire
- Deux opérandes de mêm

Exemples :

```
move $t3, $t0 # $t3 = $t0
lw $t1, ($t2) # $t1 = 32 bits
lb $t1, 4($t3) # $t1 = 8 bits à
sh $t2, ($t0) # Stocke 16 bits
```

0x10000009	40	\$t0	\$t1
0x10000008	C1		0x3699A21C
0x10000007	A2	\$t2	\$t3
0x10000006	89		0x10000005
0x10000005	F2		0x10000000
0x10000004	AA		
0x10000003	5E		
0x10000002	4F		
0x10000001	12		
0x10000000	30		

Instruction move : move dest, source

- source : immédiat, registre général, registre de segment, adresse mémoire
- destination : registre général, registre de segment, adresse mémoire
- Deux opérandes de mêm

Exemples :

```
move $t3, $t0 # $t3 = $t0
lw $t1, ($t2) # $t1 = 32 bits
lb $t1, 4($t3) # $t1 = 8 bits à
sh $t2, ($t0) # Stocke 16 bits
```

0x10000009	40	\$t0	\$t1
0x10000008	C1	0x10000000	0xC1A289F2
0x10000007	A2	\$t2	\$t3
0x10000006	89	0x10000005	0x10000000
0x10000005	F2	AA	
0x10000004		5E	
0x10000003		4F	
0x10000002		12	
0x10000001		30	
0x10000000			

Instruction move : move dest, source

- source : immédiat, registre général, registre de segment, adresse mémoire
- destination : registre général, registre de segment, adresse mémoire
- Deux opérandes de même taille

Exemples :

```
move $t3, $t0 # $t3 = $t0
lw $t1, ($t2) # $t1 = 32 bits
lb $t1, 4($t3) # $t1 = 8 bits à l'adresse de $t3
sh $t2, ($t0) # Stocke 16 bits
```

0x10000009	40	\$t0	\$t1
0x10000008	C1	0x10000000	0x000000AA
0x10000007	A2	\$t2	\$t3
0x10000006	89	0x10000005	0x10000000
0x10000005	F2		
0x10000004	AA		
0x10000003	5E		
0x10000002	4F		
0x10000001	12		
0x10000000	30		

Instruction move : move dest, source

- source : immédiat, registre général, registre de segment, adresse mémoire
- destination : registre général, registre de segment, adresse mémoire
- Deux opérandes de mêm

Exemples :

```
move $t3, $t0 # $t3 = $t0  
lw $t1, ($t2) # $t1 = 32 bits  
lb $t1, 4($t3) # $t1 = 8 bits à  
sh $t2, ($t0) # Stocke 16 bits
```

0x10000009	40	\$t0	\$t1
0x10000008	C1	0x10000000	0xC1A2AAF2
0x10000007	A2	\$t2	\$t3
0x10000006	89	0x10000005	0x10000000
0x10000005	F2		
0x10000004	AA		
0x10000003	5E		
0x10000002	4F		
0x10000001	00		
0x10000000	05		



Transfert de ou vers une adresse mémoire :

Format	Mémoire adressée	Exemple
(reg)	MEM[reg]	lw \$t0, (\$t1)
imm	MEM[imm]	lb \$t0, 0x10000000
imm(reg)	MEM[imm+reg]	lh \$t0, 4(\$t1)
étiq	MEM[étiq]	lw \$t0, tab
étiq±imm	MEM[étiq±imm]	lw \$t0, tab+8

Formats d'adressage étendu (format original : imm(reg) seul supporté)



Étiquettes ou labels :

- ▶ Correspondent à une *position* en mémoire
 - ➡ « pointeur »
- ▶ Adresse ou contenu en fonction de l'instruction utilisée

Exemple :

```
.data
age: .word 45
.text
# $t0 = adresse de la case contenant 45
la $t0, age # adresse toujours sur 32 bits
# $t0 = 45
lw $t0, age
```



Opérations sur les entiers :

- ▶ add, sub, mul, div, ...

Exemple

```
add $t0, $t1, $t2 # $t0 = $t1 + $t2
```

- ▶ Pas d'indicateur pour indiquer un *overflow*.
- ▶ En cas d'overflow : exception levée
- ▶ Instructions spéciales pour éviter l'exception
 - ▶ Exemple : addu au lieu de add



Sauts relatifs (courts)

- ▶ Saut inconditionnel : b finsi
- ▶ Sauts conditionnels :

Instruction	Interprétation
beq \$t0,\$t1, étiq	saut si $\$t0 == \$t1$
bne \$t0,\$t1, étiq	saut si $\$t0 != \$t1$
blt \$t0,\$t1, étiq	saut si $\$t0 < \$t1$
ble \$t0,\$t1, étiq	saut si $\$t0 \leq \$t1$
bgt \$t0,\$t1, étiq	saut si $\$t0 > \$t1$
bge \$t0,\$t1, étiq	saut si $\$t0 \geq \$t1$
...	



Contrôle de flot (2/3)

```
.text
li $t0, 3
li $t1, -2 # ou 4294967294
if: ble $t0, $t1, endif
then:
    add $t2, $t2, 3
    b endif
endif:
```

La valeur dans \$t0 est-elle plus grande que celle de \$t1 ?



Contrôle de flot (2/3)

```
.text
li $t0, 3
li $t1, -2 # ou 4294967294
if: ble $t0, $t1, endif
then:
    add $t2, $t2, 3
    b endif
endif:
```

La valeur dans \$t0 est-elle plus grande que celle de \$t1 ?

Tests conditionnels différents suivant le type des opérandes :

Test	Signé	Non signé
<	blt	bltu
>	bgt	bgtu
\leq	ble	bleu
\geq	bge	bgeu



Sauts absolus (longs) :

- ▶ Saut inconditionnel : `j finprog`
- ▶ Saut sur registre : `jr $ra`
- ▶ Saut avec lien : `jal prog, jalr prog`
 - ▶ Saut à l'étiquette `prog` après avoir sauvé dans `$ra` l'adresse de retour (adresse de l'instruction suivant le `jal/jalr`)



Instructions logiques

- ▶ Instructions logiques *bit à bit* (and, or, not, xor, ...)

AND	0	1	OR	0	1	XOR	0	1	NOT	0	1
0	0	0	0	0	1	0	0	1	1	0	0
1	0	1	1	1	1	1	1	0	0	1	1

- ▶ Applications :

- ▶ Mise à 0 d'un registre :

```
xor $t0, $t0, $t0 # $t0 = 0
```

- ▶ Masquage, mise à 0 ou 1 (and ou or) ou flip-flap (xor) d'un bit particulier, complément à 1 :

$$\begin{array}{r} 10100101 \\ \text{and } 00001000 \\ \hline 00000000 \end{array}$$

$$\begin{array}{r} 10100101 \\ \text{or } 00001000 \\ \hline 10101101 \end{array}$$

$$\begin{array}{r} 10101101 \\ \text{xor } 00001000 \\ \hline 10100101 \end{array}$$

$$\begin{array}{r} \text{not } 10100101 \\ \hline 01011010 \end{array}$$



Traduction du if then else

```
if ($t0 == 5) {  
    $t1 = 6;  
    $t2 = $t2 + 1;  
} else {  
    $t1 = 9;  
}
```

```
# test inversé  
si:    bne $t0, 5, sinon  
alors:  
        li $t1, 6  
        add $t2, $t2, 1  
        b finsi  
sinon:  
        li $t1, 9  
finsi:
```



Traduction du while do

```
while ($t0 > 2) {  
    $t1 = $t1*2;  
    $t0 = $t0 - 1;  
}
```

```
# test inversé  
tantque: ble $t0, 2, ftq  
          sll $t1, $t1, 1  
          sub $t0, $t0, 1  
          b tantque  
ftq:
```

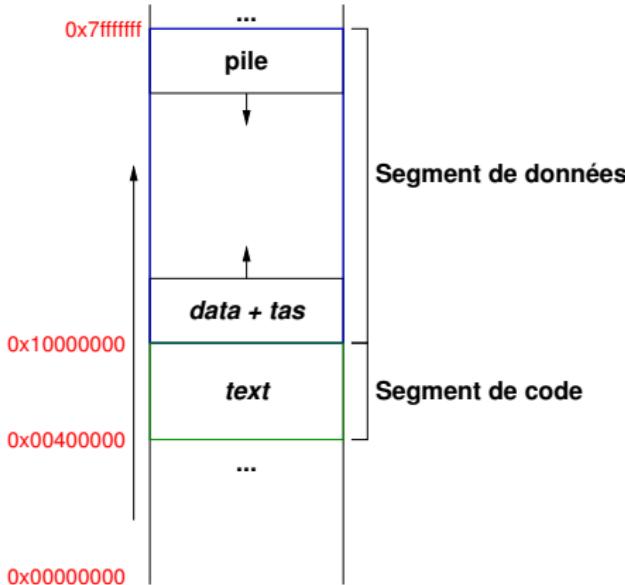


Utilisation de la mémoire par un programme s'exécutant :

- ▶ **.text** : portion de la mémoire occupée par le code du programme
- ▶ **.data** : variables statiques + mémoire allouée dynamiquement (*tas*)
- ▶ **pile (stack)** : variables locales aux procédures/fonctions, paramètres



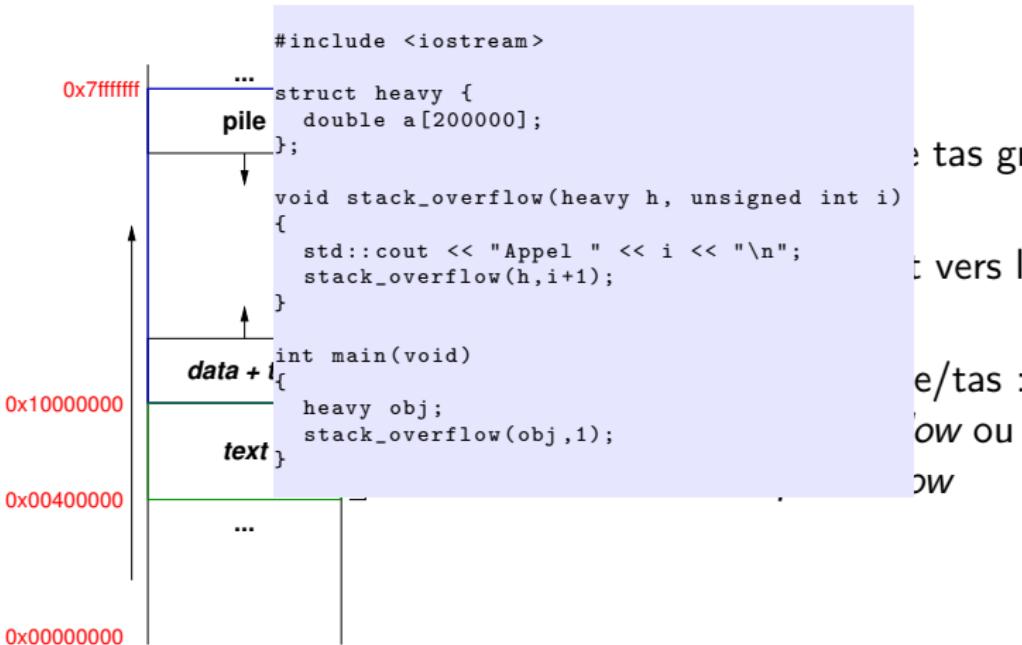
Positionnement en mémoire



- ▶ La pile et le tas grandissent l'un vers l'autre
- ▶ La pile croît vers les adresses plus petites
- ▶ Collision pile/tas : *stack overflow* ou *heap overflow*



Positionnement en mémoire



Les tas grandissent l'un

vers les adresses

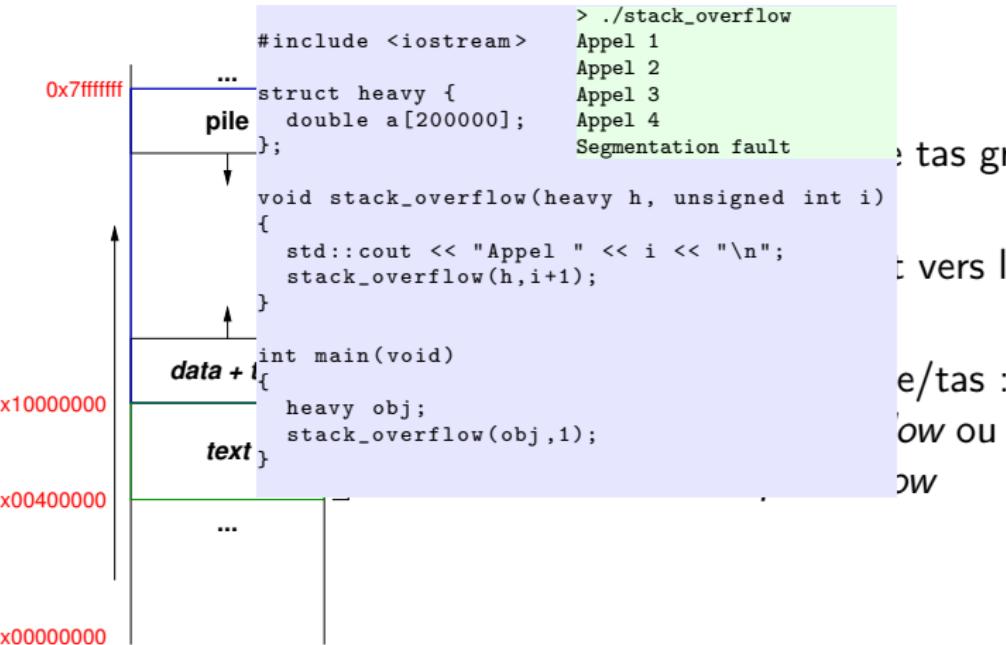
de/tas :

OW OU

DW



Positionnement en mémoire



Les tas grandissent l'un

vers les adresses

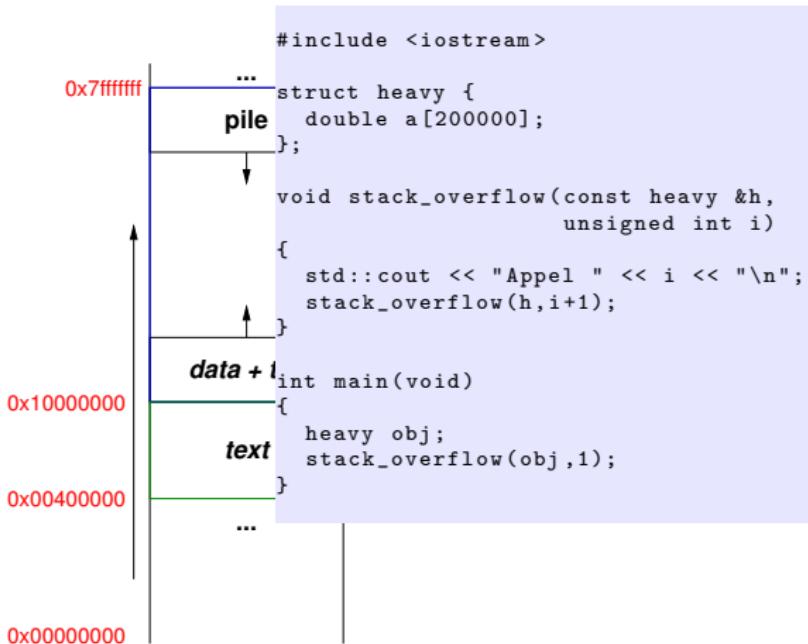
de tas :

OW OU

OW



Positionnement en mémoire



• tas grandissent l'un

• vers les adresses

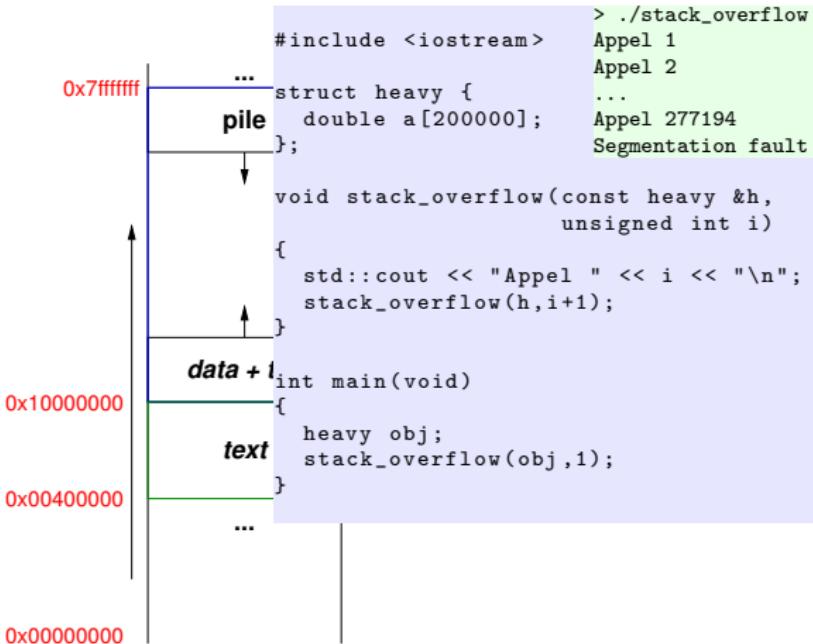
e/tas :

OW OU

DW



Positionnement en mémoire



• tas grandissent l'un

• vers les adresses

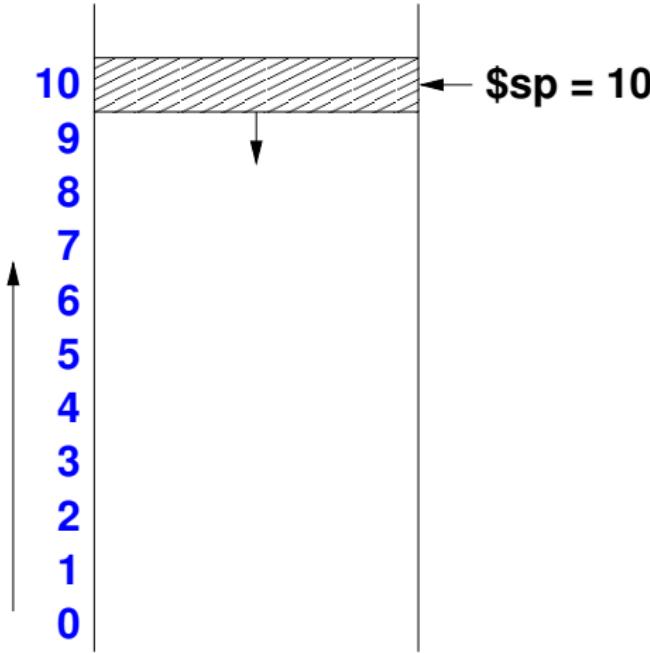
e/tas :

OW OU

OW



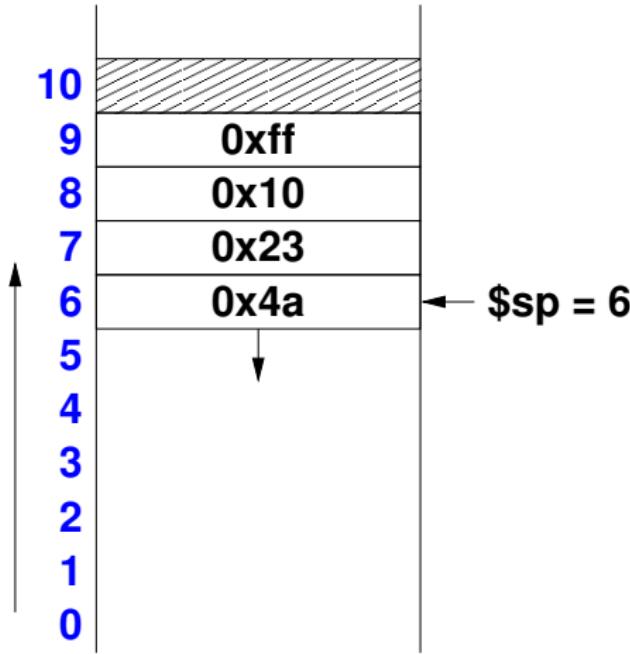
La pile (cas général)



```
li $t0, 0xff10234a  
sub $sp, $sp, 4  
sw $t0, ($sp)  
sub $sp, $sp, 2  
sh $t0, ($sp)  
add $sp, $sp, 2  
lw $t1, ($sp)  
add $sp, $sp, 4
```



La pile (cas général)

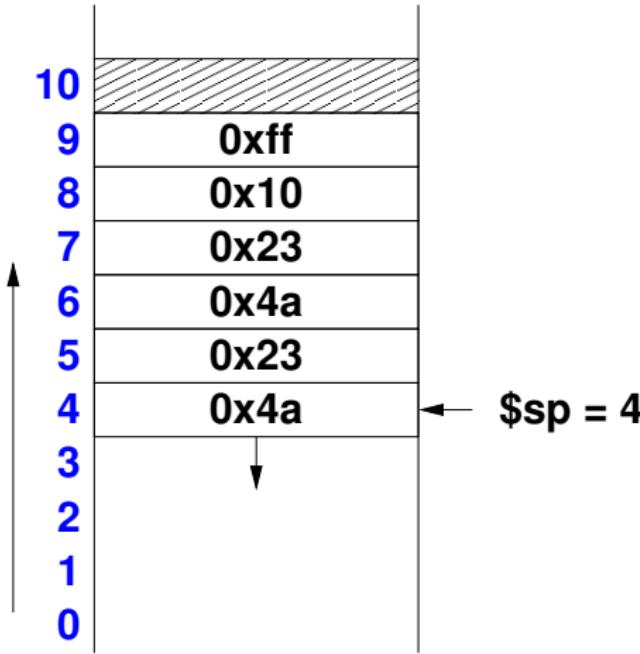


PUSH \$t0

```
li $t0, 0xff10234a  
sub $sp, $sp, 4  
sw $t0, ($sp)  
sub $sp, $sp, 2  
sh $t0, ($sp)  
add $sp, $sp, 2  
lw $t1, ($sp)  
add $sp, $sp, 4
```



La pile (cas général)

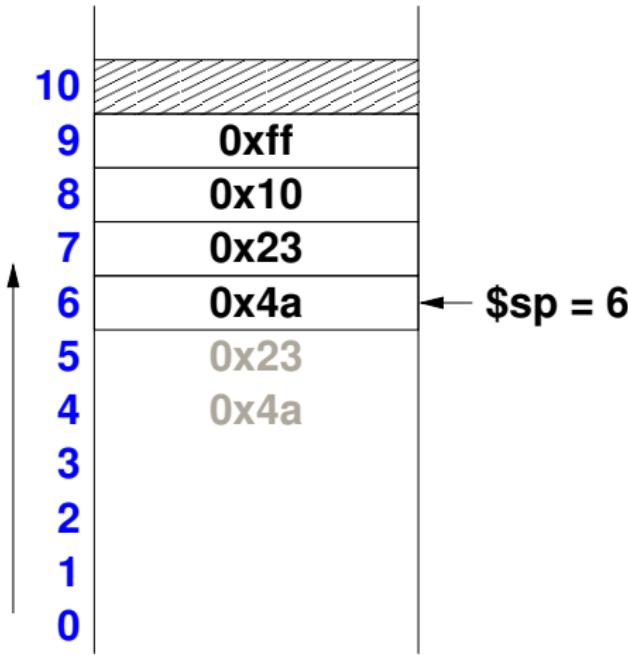


PUSH low(\$t0)

```
li $t0, 0xff10234a  
sub $sp, $sp, 4  
sw $t0, ($sp)  
sub $sp, $sp, 2  
sh $t0, ($sp)  
add $sp, $sp, 2  
lw $t1, ($sp)  
add $sp, $sp, 4
```



La pile (cas général)

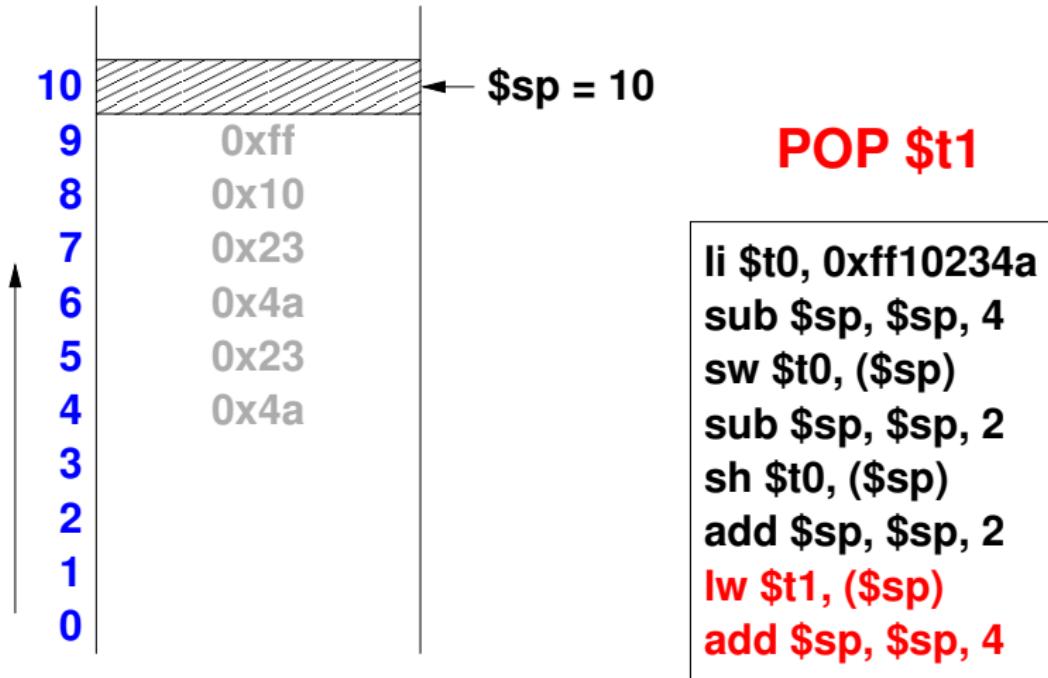


POP

```
li $t0, 0xff10234a
sub $sp, $sp, 4
sw $t0, ($sp)
sub $sp, $sp, 2
sh $t0, ($sp)
add $sp, $sp, 2
lw $t1, ($sp)
add $sp, $sp, 4
```

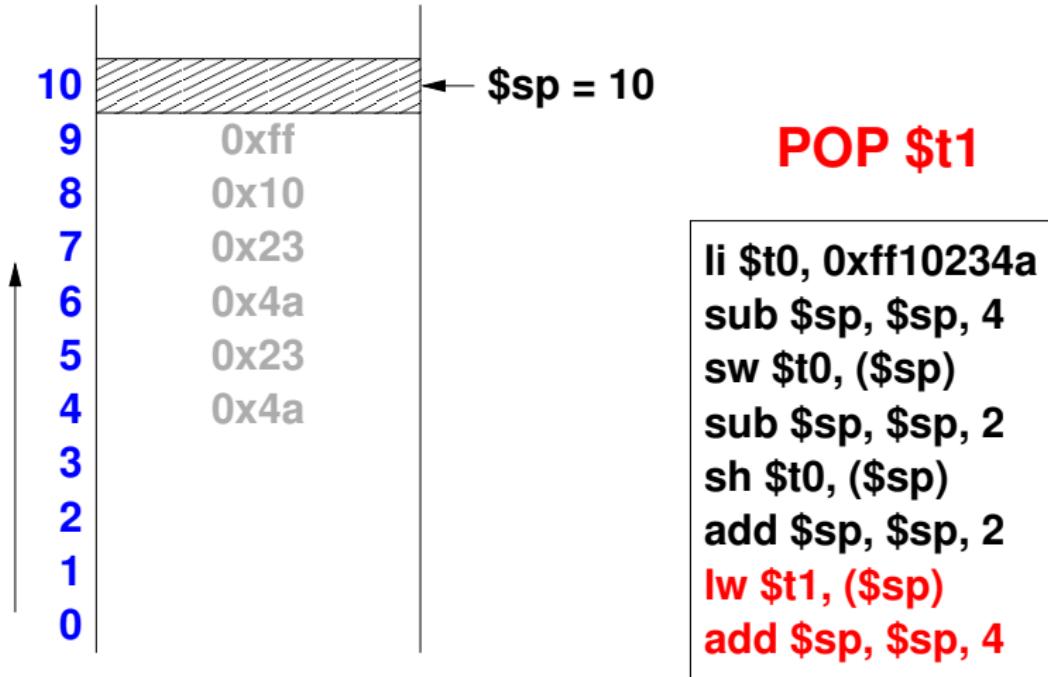


La pile (cas général)





La pile (cas général)



Attention : convention MIPS

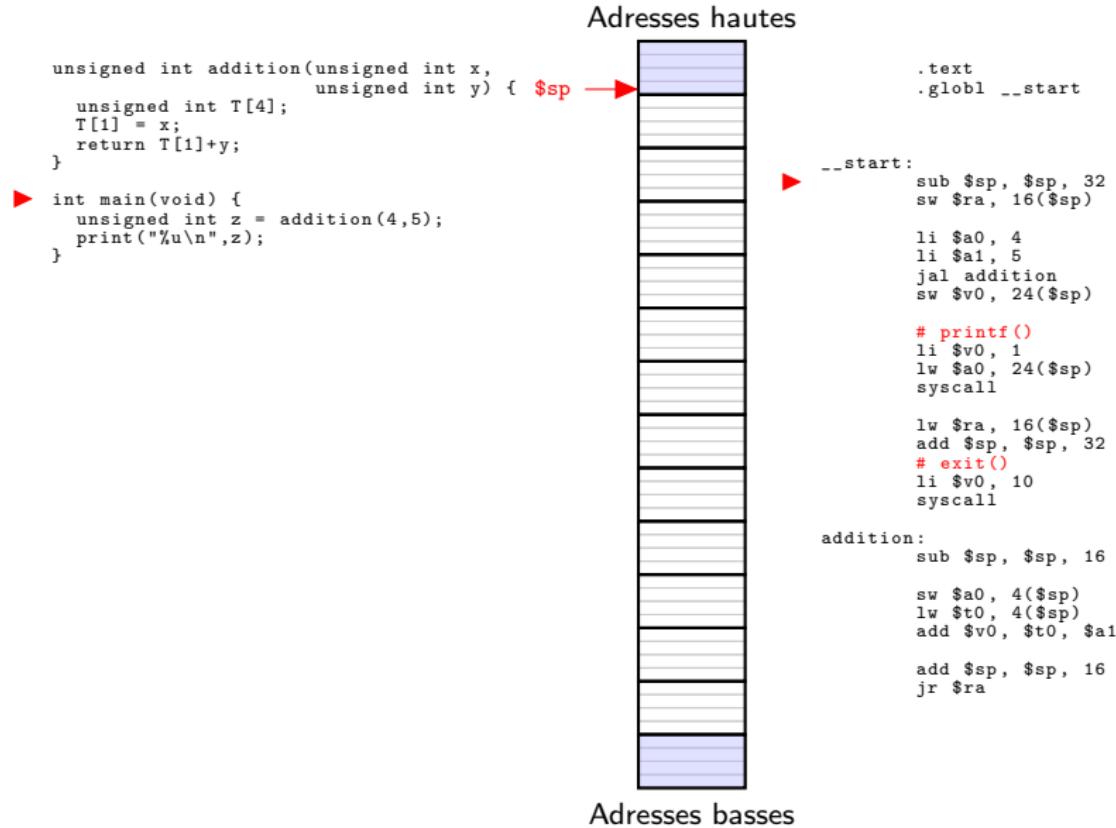
\$sp pointe toujours sur une adresse multiple de 8



- ▶ Stockage de résultats temporaires si manque de registres
- ▶ Stockage des variables locales aux procédures/fonctions
- ▶ Passage de paramètres entre procédures/fonctions
- ▶ Attention à la taille des données :
 - ▶ Empilement d'un mot puis d'un double-mot
 - ▶ **dépilement d'un mot puis d'un double-mot**
 - ▶ **dépilement d'un double-mot puis d'un mot**



Passage de paramètres : exemple

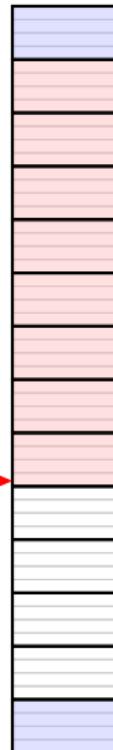




Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
}  
  
► int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

Adresses hautes



Adresses basses

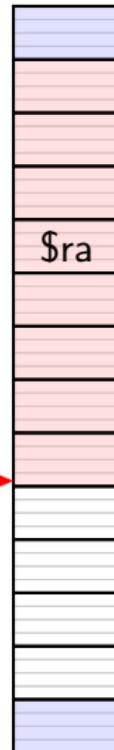
```
.text  
.globl __start  
  
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)  
  
    # printf()  
    li $v0, 1  
    lw $a0, 24($sp)  
    syscall  
  
    lw $ra, 16($sp)  
    add $sp, $sp, 32  
    # exit()  
    li $v0, 10  
    syscall  
  
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```



Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
}  
  
► int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

Adresses hautes



Adresses basses

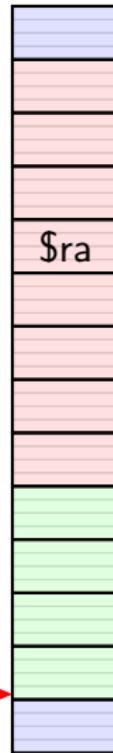
```
.text  
.globl __start  
  
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)  
  
    # printf()  
    li $v0, 1  
    lw $a0, 24($sp)  
    syscall  
  
    lw $ra, 16($sp)  
    add $sp, $sp, 32  
    # exit()  
    li $v0, 10  
    syscall  
  
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```



Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
}  
  
int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

Adresses hautes



Adresses basses

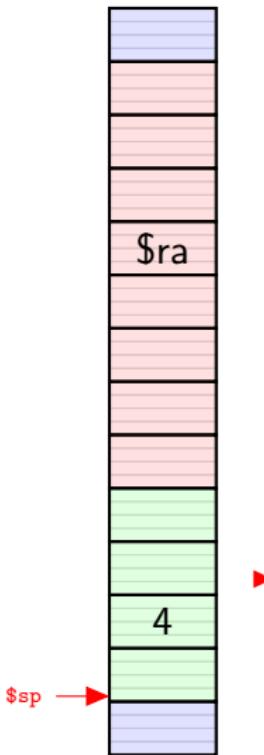
```
.text  
.globl __start  
  
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)  
  
    # printf()  
    li $v0, 1  
    lw $a0, 24($sp)  
    syscall  
  
    lw $ra, 16($sp)  
    add $sp, $sp, 32  
    # exit()  
    li $v0, 10  
    syscall  
  
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```



Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
  
► }  
  
int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

Adresses hautes



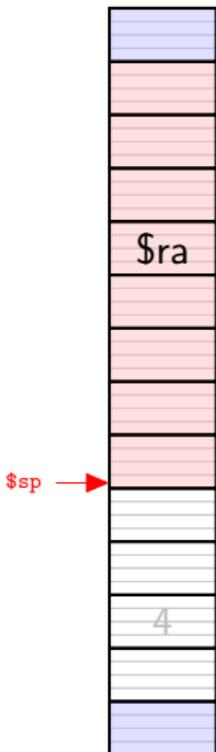
```
.text  
.globl __start  
  
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)  
  
    # printf()  
    li $v0, 1  
    lw $a0, 24($sp)  
    syscall  
  
    lw $ra, 16($sp)  
    add $sp, $sp, 32  
    # exit()  
    li $v0, 10  
    syscall  
  
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```



Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
}  
  
► int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

Adresses hautes



Adresses basses

```
.text  
.globl __start  
  
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)  
  
    # printf()  
    li $v0, 1  
    lw $a0, 24($sp)  
    syscall  
  
    lw $ra, 16($sp)  
    add $sp, $sp, 32  
    # exit()  
    li $v0, 10  
    syscall  
  
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```



Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
}  
  
► int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

Adresses hautes



```
.text  
.globl __start
```

```
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)
```

```
# printf()  
li $v0, 1  
lw $a0, 24($sp)  
syscall
```

```
lw $ra, 16($sp)  
add $sp, $sp, 32  
# exit()  
li $v0, 10  
syscall
```

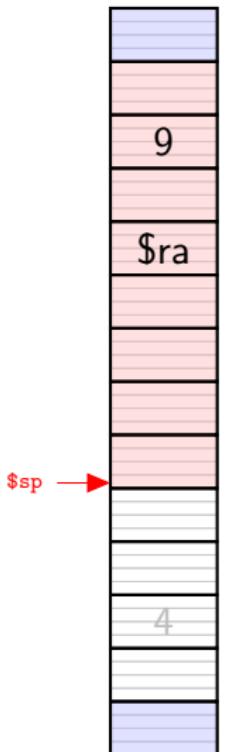
```
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```



Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,  
                     unsigned int y) {  
    unsigned int T[4];  
    T[1] = x;  
    return T[1]+y;  
}  
  
int main(void) {  
    unsigned int z = addition(4,5);  
    print("%u\n",z);  
}
```

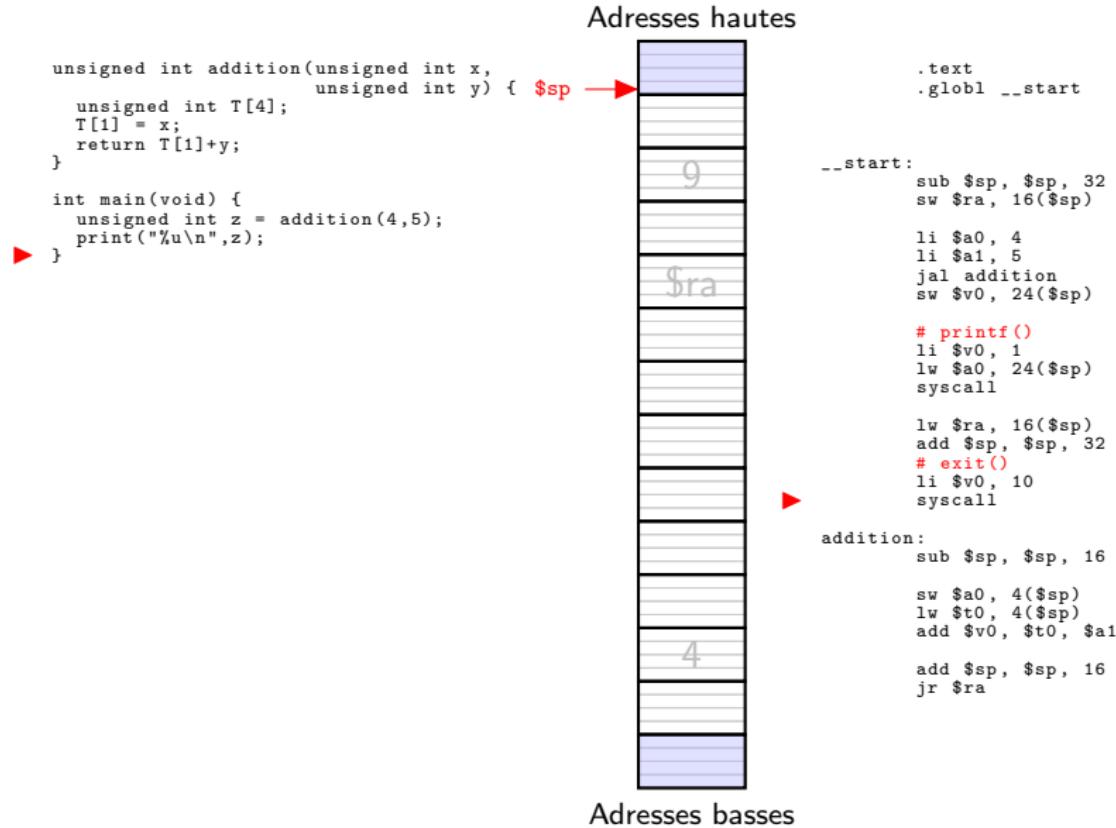
Adresses hautes



```
.text  
.globl __start  
  
__start:  
    sub $sp, $sp, 32  
    sw $ra, 16($sp)  
  
    li $a0, 4  
    li $a1, 5  
    jal addition  
    sw $v0, 24($sp)  
  
    # printf()  
    li $v0, 1  
    lw $a0, 24($sp)  
    syscall  
  
    lw $ra, 16($sp)  
    add $sp, $sp, 32  
    # exit()  
    li $v0, 10  
    syscall  
  
addition:  
    sub $sp, $sp, 16  
  
    sw $a0, 4($sp)  
    lw $t0, 4($sp)  
    add $v0, $t0, $a1  
  
    add $sp, $sp, 16  
    jr $ra
```

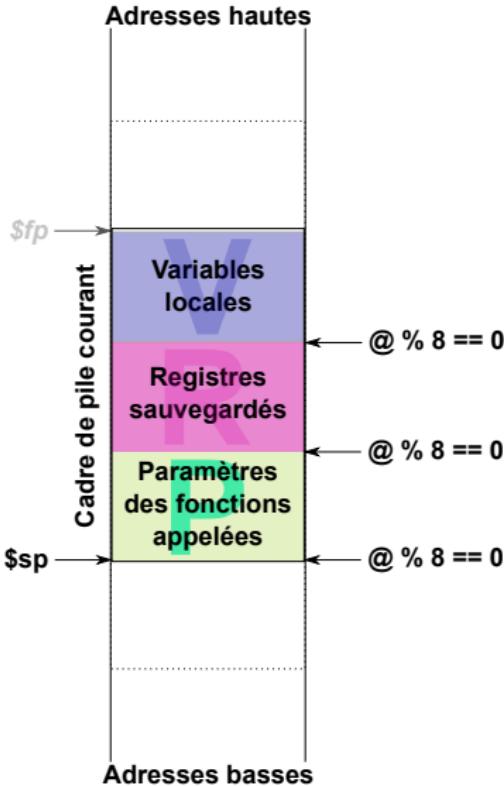


Passage de paramètres : exemple





Cadre de pile général



- ▶ Chaque bloc $\{P, R, V\}$ est optionnel
- ▶ Taille(P) ≥ 16 octets si P est présent (place pour \$a0, \$a1, \$a2, \$a3 de l'*appelé*)
- ▶ Taille(P) doit être au moins égale au plus grand nombre d'octets requis pour appeler une fonction dans la fonction courante
- ▶ Utilisation de \$fp pas toujours nécessaire
- ▶ Taille du cadre en octets multiple de 8
- ▶ Chaque bloc $\{P, R, V\}$ à une adresse divisible par 8 (*padding* si nécessaire)



Fonction terminale simple

```
int f(int a, int b)
{
    return a+b;
}
```

- ▶ Inutile de sauver \$ra
- ▶ Pas de variable locale
- ▶ Pas d'utilisation de registre à sauver

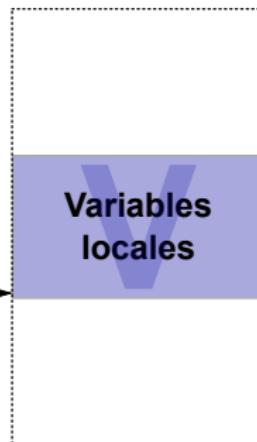
⇒ pas de cadre de pile à créer



Fonction terminale avec variables locales

```
int f(int a, int b)
{
    int T[20];
    ...
    return T[0];
}
```

\$sp



- ▶ Pas besoin de sauver \$ra
- ▶ Présence de variables locales
- ▶ Pas de modification de registre à sauvegarder

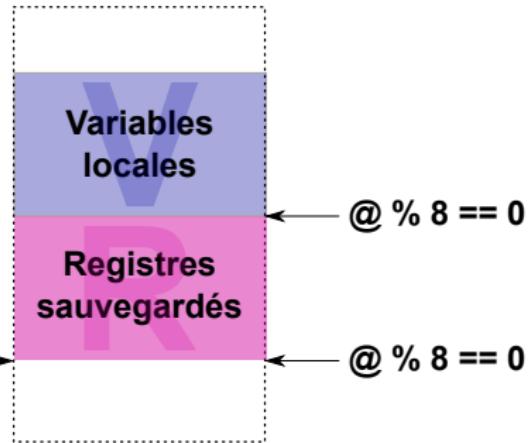
⇒ Création d'un cadre de pile {V}



Fonction terminal évoluée

```
int f(int a, int b)
{
    int T[20];
    // Calcul élaboré utilisant,
    // e.g., $s0
    return T[0];
}
```

\$sp →

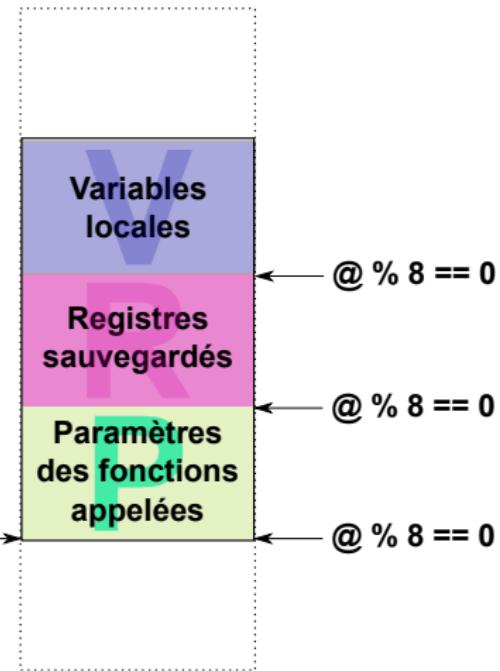


- ▶ Pas besoin de sauver \$ra
- ▶ Présence de variables locales
- ▶ Registre(s) à sauvegarder

⇒ Création d'un cadre de pile {R, V}

```
int f(int a, int b)
{
    int T[20];
    // Calcul élaboré utilisant,
    // e.g., $s0
    T[0] = f(T[1], b);
    return T[0]+T[1];
```

- ▶ Appel de f : sauver \$ra
- ▶ Appel de f : créer un bloc P \$sp
- ▶ Présence de variables locales
- ▶ Registre(s) à sauvegarder



⇒ Crédit d'un cadre de pile {P, R, V}



- ▶ Passage des 4 premiers paramètres dans \$a0, \$a1, \$a2, \$a3
- ▶ Passage des paramètres suivant dans le cadre de pile de l'*appelant* (bloc P)
- ▶ Chaque paramètre est sauvé dans la pile sur au moins 32 bits
- ▶ Retour des résultats dans \$v0 et \$v1
- ▶ Registre \$ra contient l'adresse de retour
- ▶ Registre \$fp contient l'adresse de la première case au-dessus du cadre courant (rarement utilisé)
- ▶ Registre \$sp contient l'adresse de la case la plus basse dans le cadre courant

Instructions & chemins de données



Définition d'un jeu d'instructions :

- ▶ Instructions disponibles ?
 - ▶ Jeu complet
 - ▶ Implémentation efficace
 - ▶ (Facile d'utilisation)
- ▶ Type de jeu d'instructions ?
 - ▶ Registres, mémoire, pile
- ▶ Représentation en mémoire ?



Types de jeux d'instructions

$A \leftarrow 3 + 5 ?$

Machine à pile	Machine à accumulateur	Machine à registres
PUSH X $top(pile) \leftarrow X$	LOAD X $acc \leftarrow X$	LOAD R_i, X $R_i \leftarrow X$
POP X $X \leftarrow top(pile)$	STORE X $X \leftarrow acc$	STORE R_i, X $X \leftarrow R_i$
ADD $POP t_2 ; POP t_1$ $PUSH t_1 + t_2$	ADD X $acc \leftarrow acc + X$	ADD R_i, R_j, R_k $R_i \leftarrow R_j + R_k$
push 3 push 5 add pop A	load 3 add 5 store A	load R1, 3 load R2, 5 add R3, R1, R2 store R3, A



Choix du type de jeu d'instructions

Registres (add r0, r1, r2).

- Simplicité, nombre de cycles
- Augmente le nombre d'instructions d'un programme

Registre/mémoire (add r0, MEM).

- Bonne densité du code, pas besoin de chargement des données
- Destruction d'un opérande, nombre de cycles variable

Mémoire/mémoire (add MEM0, MEM1, MEM2).

- Code très compact (pas de load/store)
- Problème d'accès mémoire



- ▶ *Endianness*
- ▶ Contraintes d'alignement
- ▶ Modes d'adressage :

Modes	Exemple	Sens
Registre	add r0, r1, r2	$r_0 \leftarrow r_1 + r_2$
Immédiat	add r0, 4	$r_0 \leftarrow r_0 + 4$
Direct	add r0, (0x1001)	$r_0 \leftarrow r_0 + \text{MEM}[0x1001]$
Indirect	add r0, (r1)	$r_0 \leftarrow r_0 + \text{MEM}[r_1]$
Basé	add r0, 10(r1)	$r_0 \leftarrow r_0 + \text{MEM}[r_1 + 10]$
Indexé	add r0, (r1+r2)	$r_0 \leftarrow r_0 + \text{MEM}[r_1 + r_2]$



Formats d'instructions MIPS

31	26	25	21	20	16	15	11	10	6	5	0
Opcode	rs		rt		rd		decval		fonct		Format R

add, sub, ...

31	26	25	21	20	16	15	0
Opcode	rs		rt			Imm16	Format I

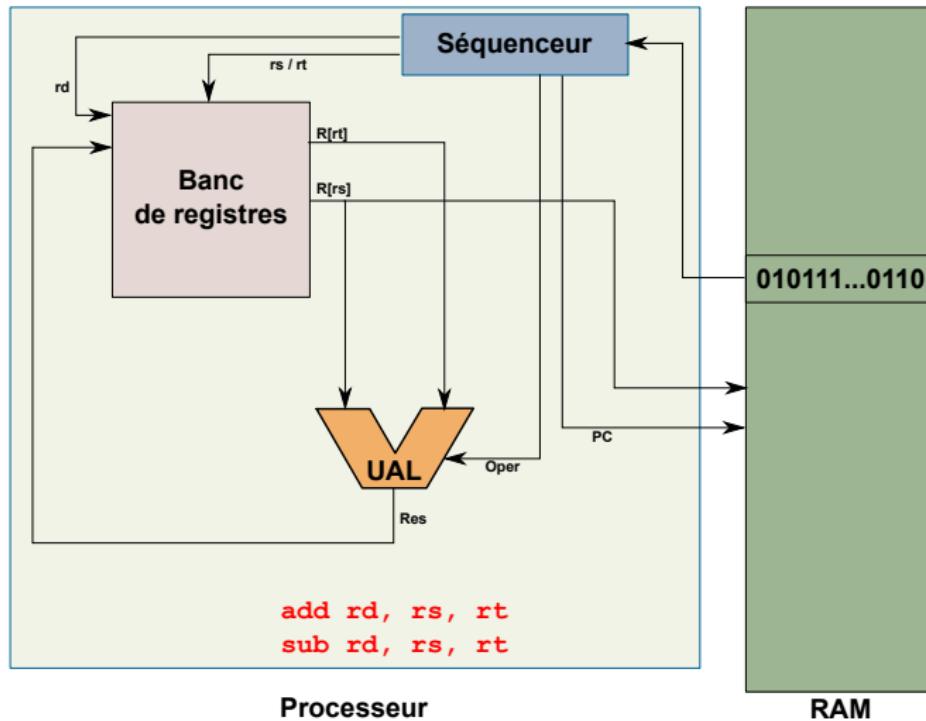
lw, sw, addi, beq, ...

31	26	25	0	
Opcode	Adresse			Format J

jal, j, ...



Vision simplifiée du chemin de données





- ▶ Architecture du jeu d'instruction
 - ➡ Langage Transfert de Registres (*RTL*)
- ▶ RTL
 - ➡ Composants du chemin de données
 - ➡ Interconnection des composants
- ▶ Composants du chemin de données
 - ➡ Signaux de contrôles
- ▶ Signaux de contrôles
 - ➡ Logique de contrôle

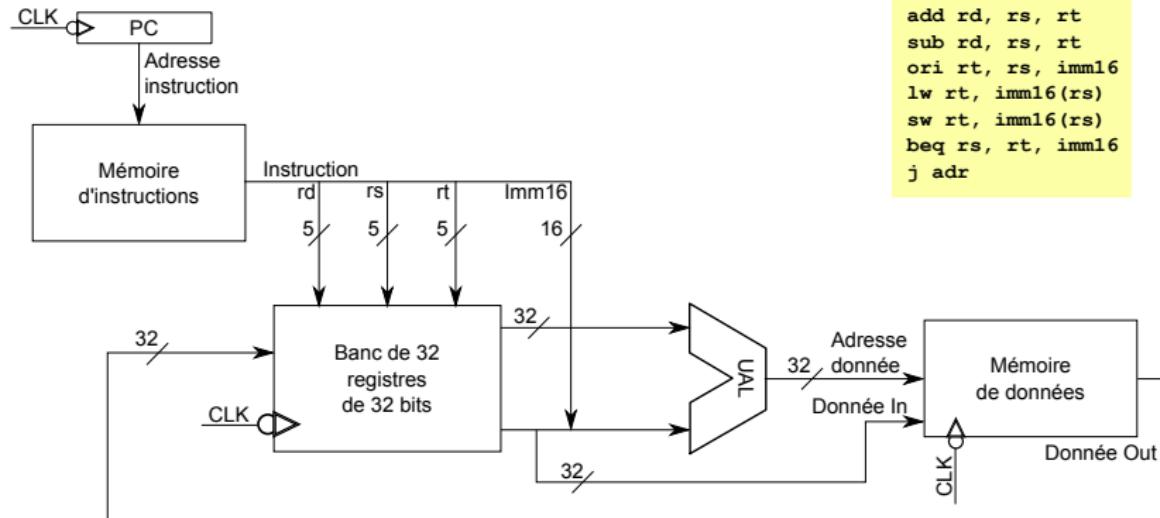


Exemple : un sous-ensemble de MIPS

- ▶ Addition et soustraction
 - ▶ add rd, rs, rt
 - ▶ sub rd, rs, rt
- ▶ « Ou » immédiat
 - ▶ ori rt, rs, imm16
- ▶ Chargement et rangement
 - ▶ lw rt, imm16(rs)
 - ▶ sw rt, imm16(rs)
- ▶ Branchement conditionnel
 - ▶ beq rs, rt, imm16
- ▶ Saut inconditionnel
 - ▶ j adr

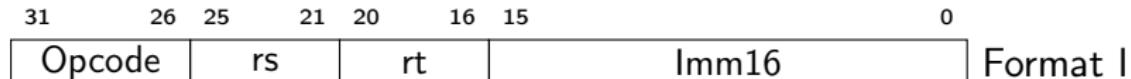


Processeur simplifié





RTL : l'instruction lw

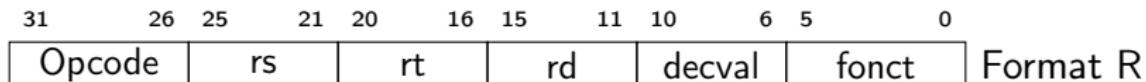


lw rt, imm16(rs)

1. $\text{MEM}[\text{PC}]$ *Extraire l'instruction de la mémoire*
2. $\text{Adr} \leftarrow \text{R}[\text{rs}] + \text{SignExt}(\text{Imm16})$ *Calcul de l'adresse mémoire*
3. $\text{R}[\text{rt}] \leftarrow \text{MEM}[\text{Adr}]$ *Chargement de la donnée*
4. $\text{PC} \leftarrow \text{PC} + 4$ *Calcul adresse prochaine instruction*



RTL : l'instruction add

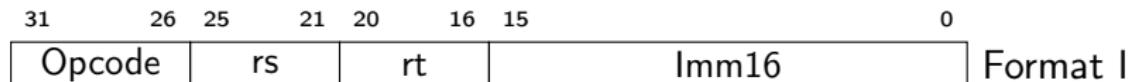


add rd, rs, rt

1. $\text{MEM}[\text{PC}]$ *Extraire l'instruction de la mémoire*
2. $\text{R}[\text{rd}] \leftarrow \text{R}[\text{rs}] + \text{R}[\text{rt}]$ *Addition*
3. $\text{PC} \leftarrow \text{PC} + 4$ *Calcul adresse prochaine instruction*



RTL : l'instruction ori



ori, rt, rs, imm16

1. $\text{MEM}[\text{PC}]$ *Extraire l'instruction de la mémoire*
2. $\text{R}[\text{rt}] \leftarrow \text{R}[\text{rs}] \vee \text{ZeroExt}(\text{Imm16})$ *Ou bit-à-bit*
3. $\text{PC} \leftarrow \text{PC} + 4$ *Calcul adresse prochaine instruction*



RTL : l'instruction j

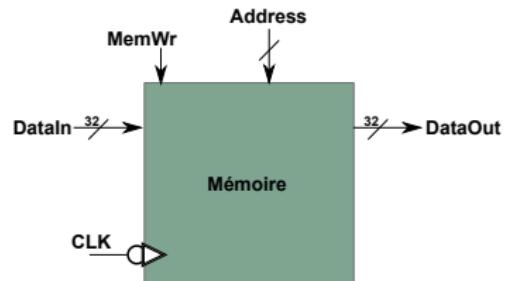
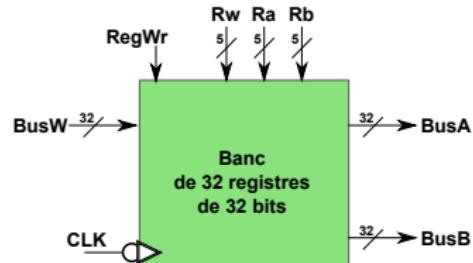
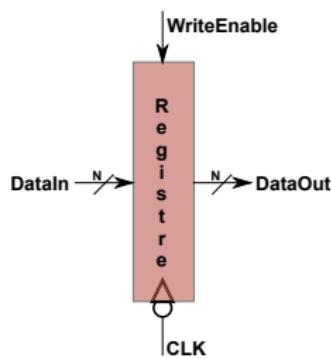
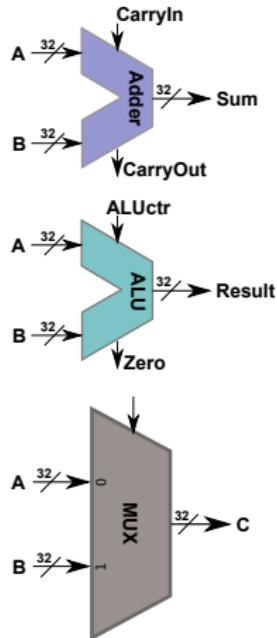


j adr

1. MEM[PC] *Extraire l'instruction de la mémoire*
2. $\text{PC}_{31:2} \leftarrow (\text{PC}_{31:28} \ll 26) | \text{Adresse}_{25:0}$ *Calcul adresse saut*



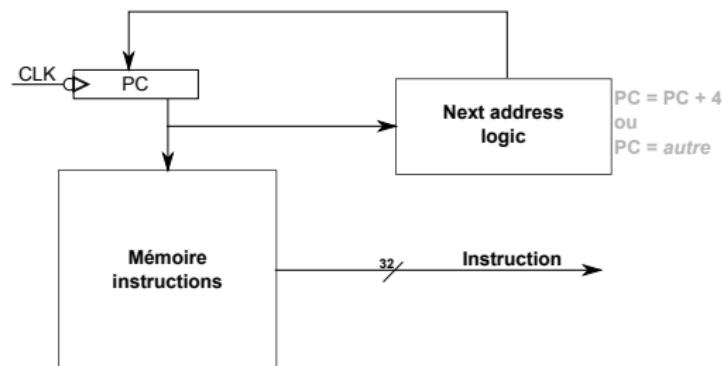
Briques de base du processeur simplifié





L'unité de *Fetch*

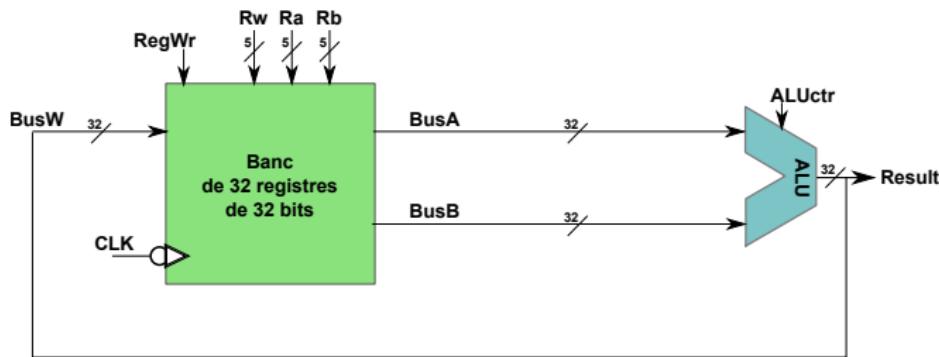
Extraction de l'instruction courante et mise à jour de PC





Chemin de données pour add/sub

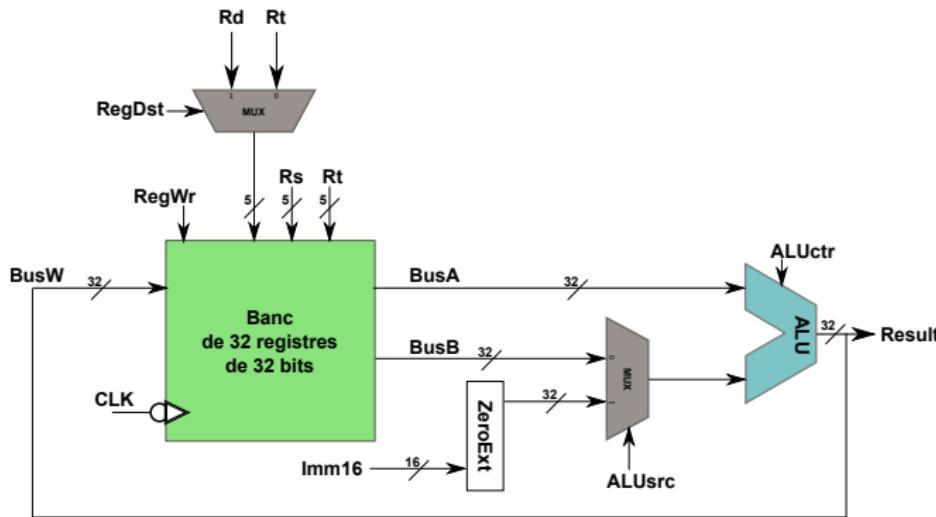
31	26	25	21	20	16	15	11	10	6	5	0
Opcode		rs		rt		rd		decval		fonct	





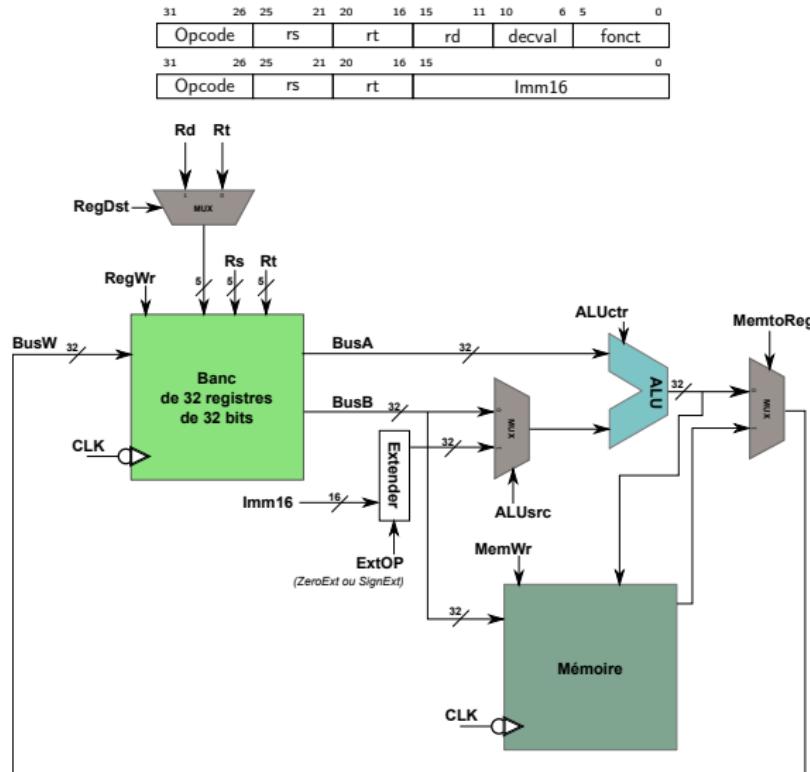
Chemin de données pour ori

31	26	25	21	20	16	15	11	10	6	5	0
Opcode	rs		rt		rd		decval		fonct		
31	26	25	21	20	16	15					0
Opcode	rs		rt				Imm16				





Chemin de données pour lw/sw





- ▶ PC : adresse sur 32 bits de l'instruction suivante
- ▶ Chaque instruction est sur 4 octets
 - ➡ 2 bits de poids faible toujours nuls
 - ➡ PC stocké sur 30 bits

Modification de PC :

Séquentiel. $\text{PC}_{31:2} \leftarrow \text{PC}_{31:2} + 1$

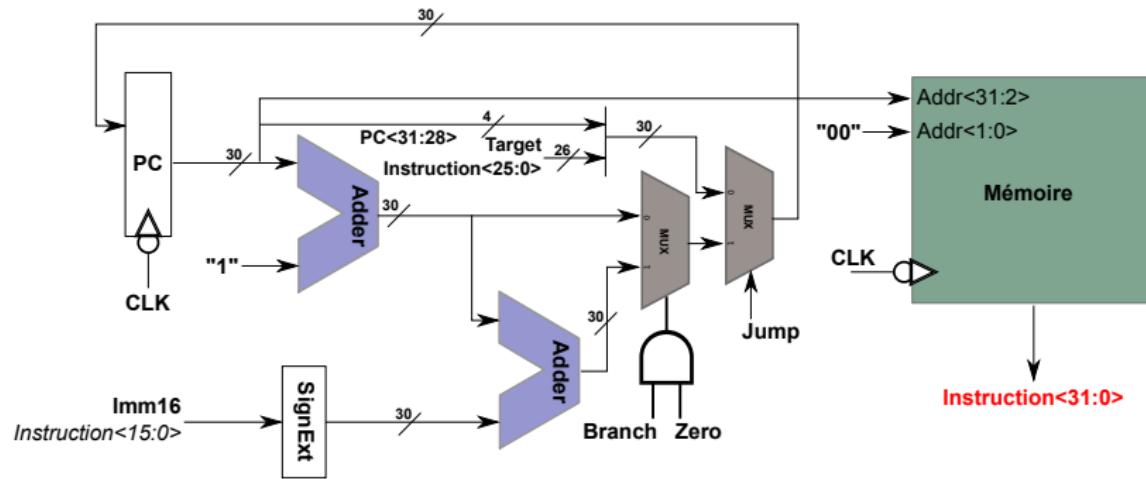
Saut relatif. $\text{PC}_{31:2} \leftarrow \text{PC}_{31:2} + 1 + \text{SignExt}(\text{Imm16})$

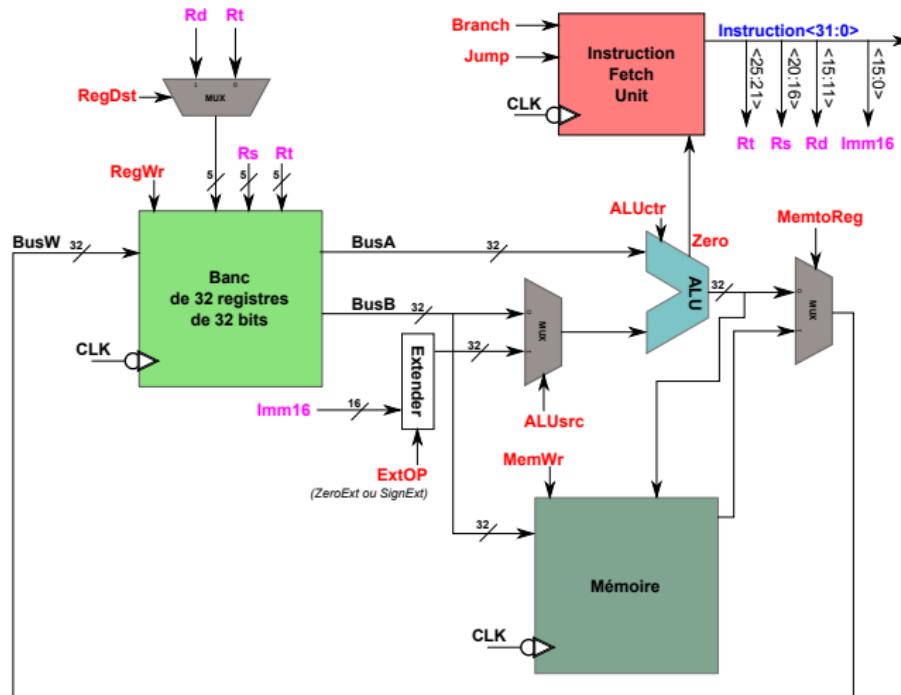
Saut absolu. $\text{PC}_{31:2} \leftarrow \text{PC}_{31:28} \mid \text{Adresse}_{25:0}$

Mémoire adressée : $\text{PC}_{31:2} \ll 2$

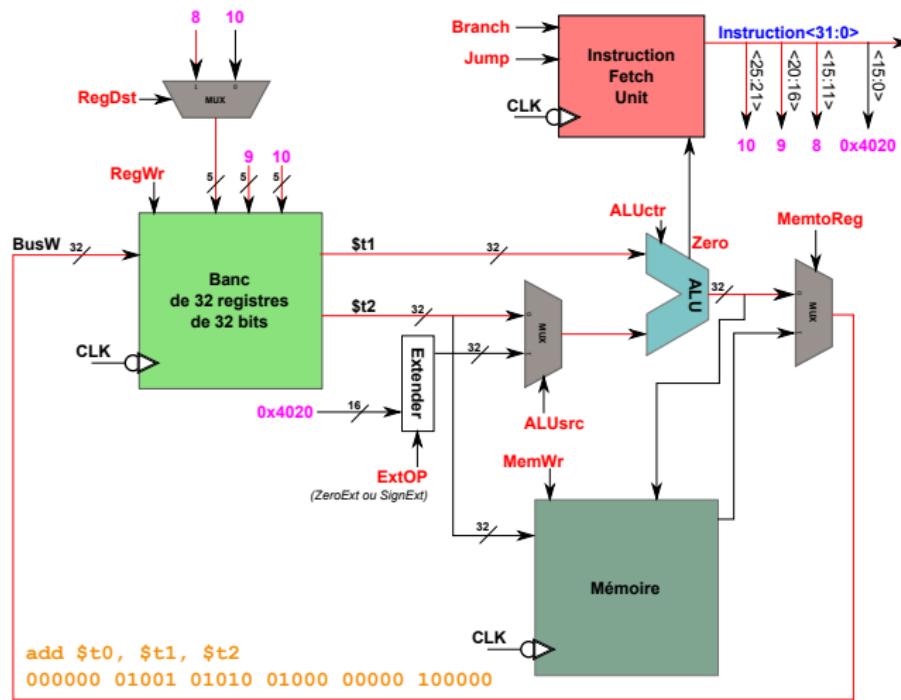


Instruction Fetch Unit





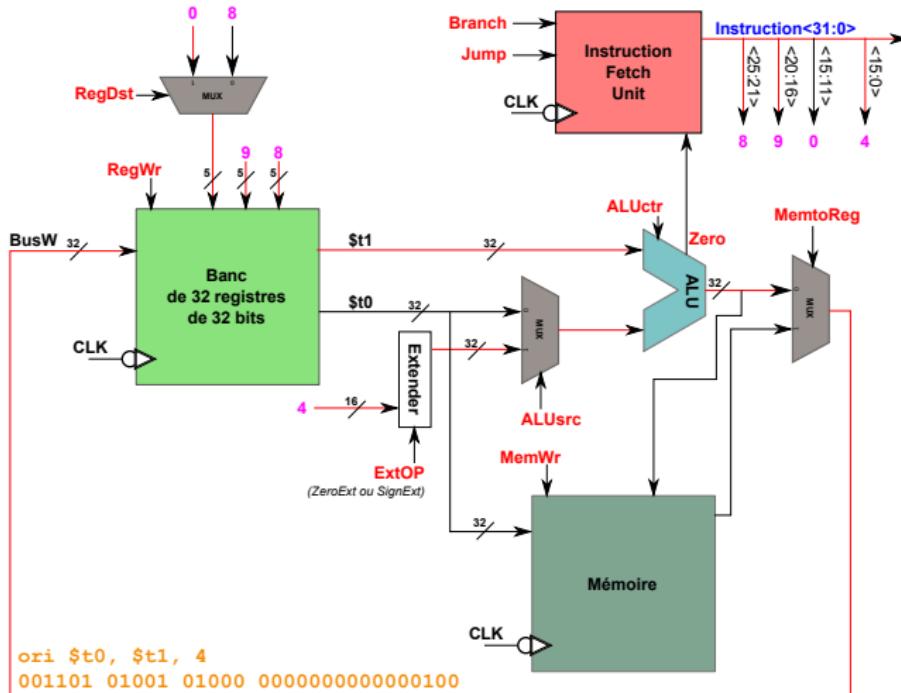
Détermination des signaux en fonction des instructions



Détermination des signaux en fonction des instructions



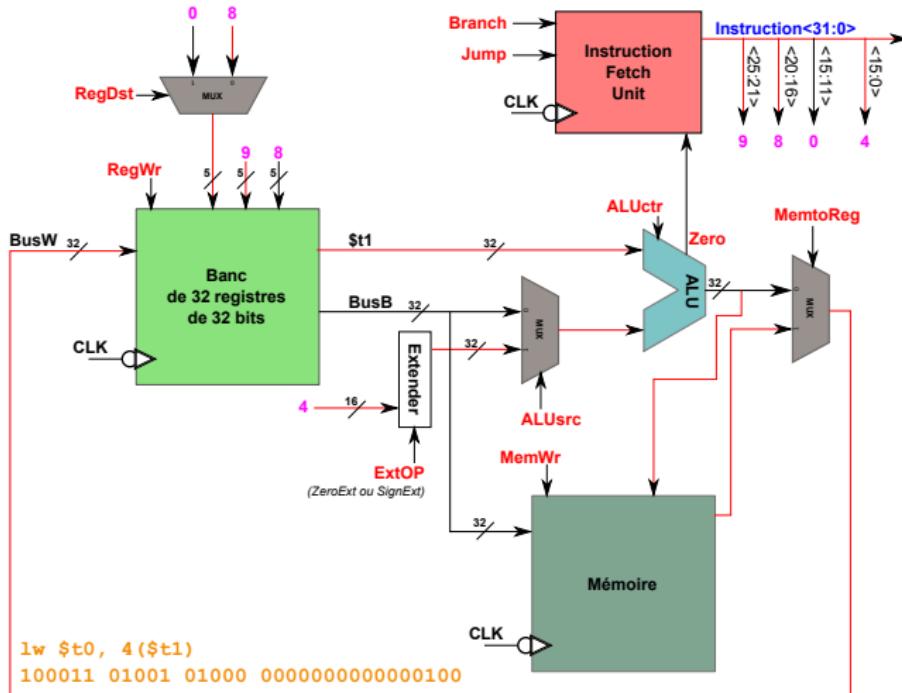
Chemin de données à cycle unique



Détermination des signaux en fonction des instructions



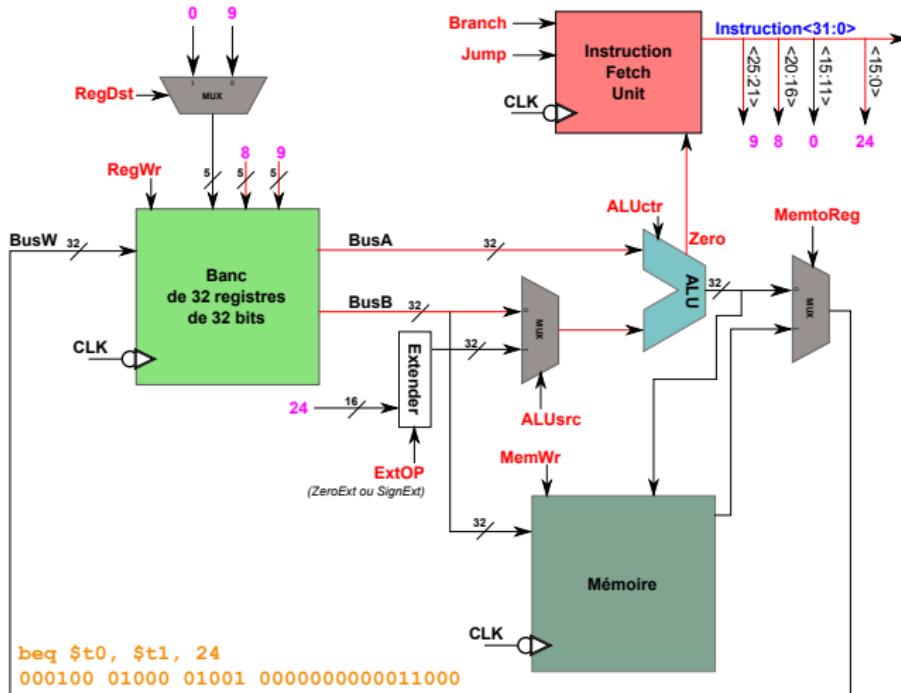
Chemin de données à cycle unique



Détermination des signaux en fonction des instructions



Chemin de données à cycle unique



Détermination des signaux en fonction des instructions



Processeur mono-cycle

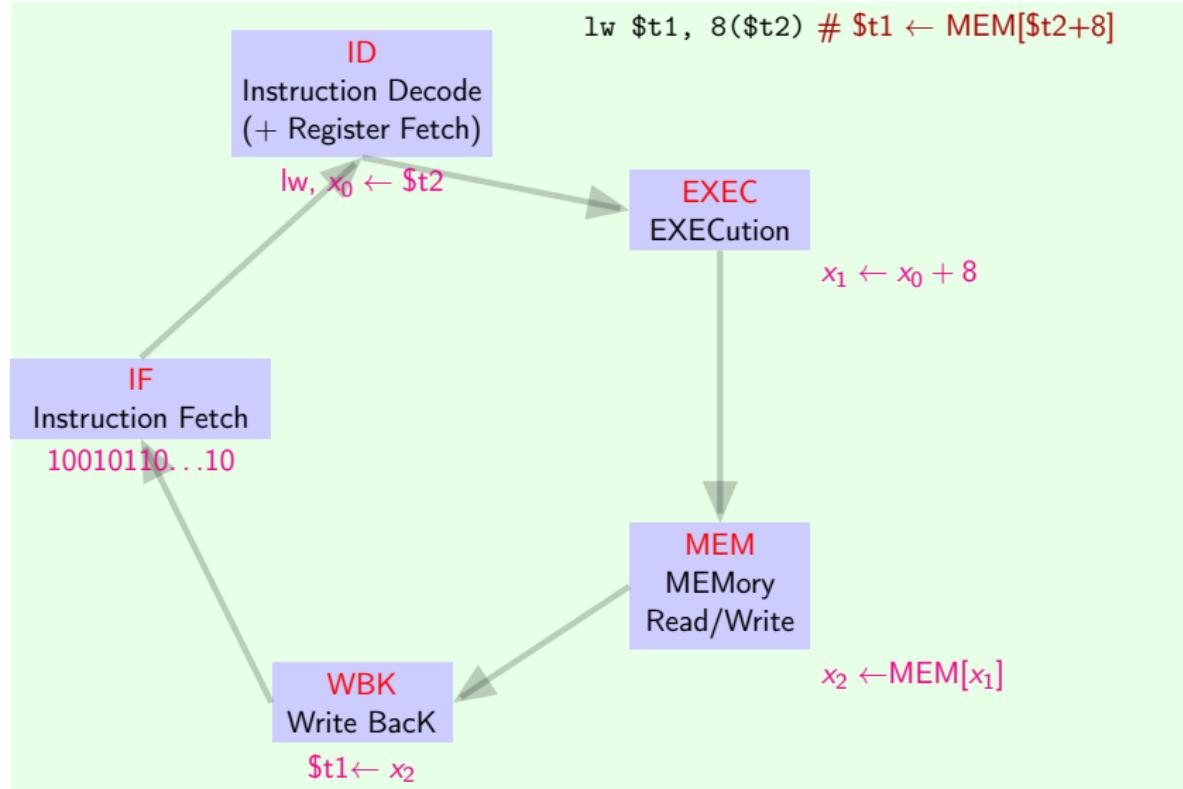
- 😊 Simple à implémenter
- 😢 Temps de cycle dépend de l'instruction la plus longue

Solution :

- ▶ Décomposer chaque instruction en étapes exécutées en un cycle : *processeur multi-cycle*

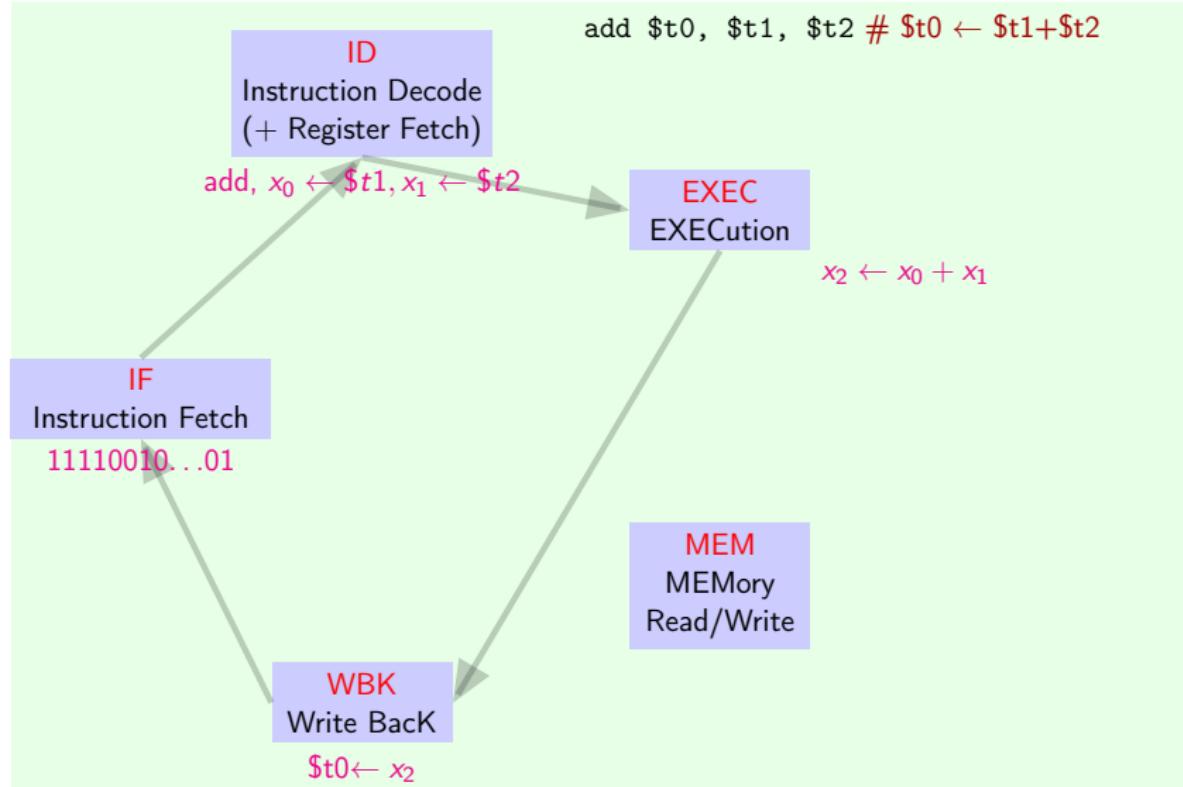


Processeur mono-cycle et multi-cycle





Processeur mono-cycle et multi-cycle





Processeur mono-cycle

- 😊 Simple à implémenter
- 😢 Temps de cycle dépend de l'instruction la plus longue

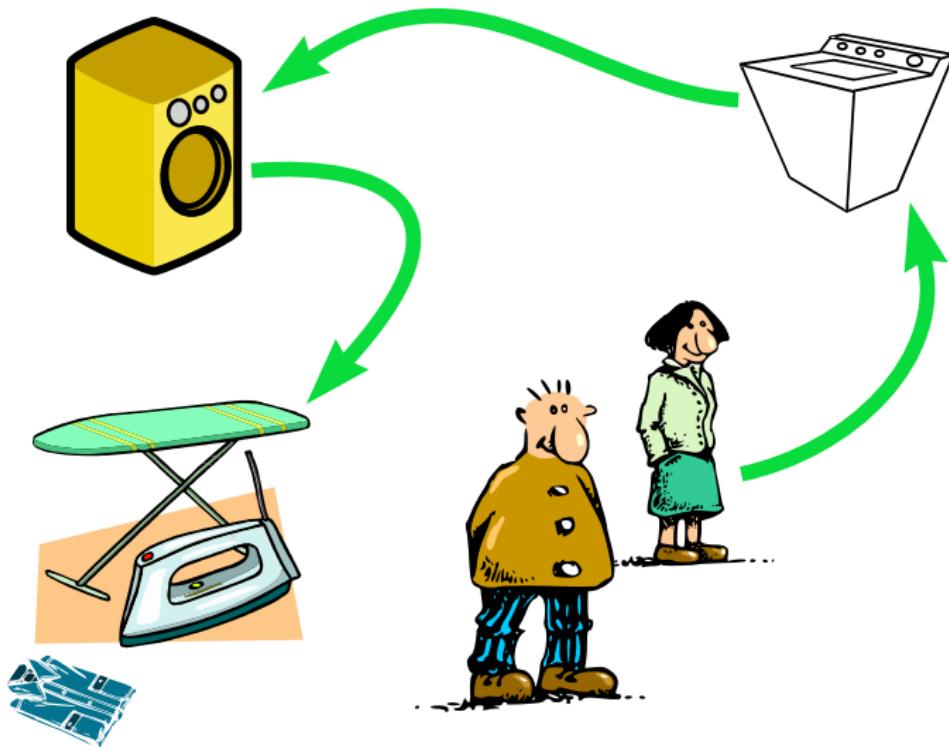
Solution :

- ▶ Décomposer chaque instruction en étapes exécutées en un cycle : *processeur multi-cycle*
- ▶ Optimisation : optimiser l'utilisation des unités fonctionnelles : *pipeline*

Pipelines



Pipeline (1)





Pipeline (2)

```
add $r0, $r1, $r2  
add $r3, $r4, $r5
```

IF : Instruction Fetch

ID : Instruction Decode (+Registers Fetch)

EXEC : EXECution

MEM : MEMory Read/Write

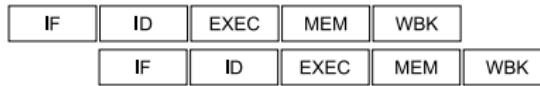
WBK : Write Back



SANS PIPELINE



AVEC PIPELINE

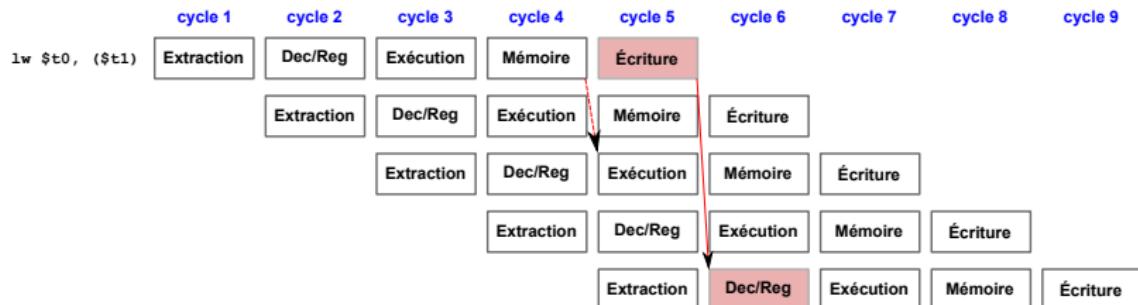




- ▶ Raccourcissement du temps de cycle
- ▶ Économe : unités fonctionnelles réutilisées
- ▶ Cinq unités indépendantes :
 - ▶ Extraction de l'instruction en mémoire (*Instruction Fetch*)
 - ▶ Décodage de l'instruction et lecture des registres (*Instruction Decode*)
 - ▶ Exécution de l'instruction par accès à l'UAL (*EXECution*)
 - ▶ Accès en lecture/écriture à la mémoire de données (*MEM*)
 - ▶ Accès aux registres en écriture (*Write Back*)



Problème du chargement différé



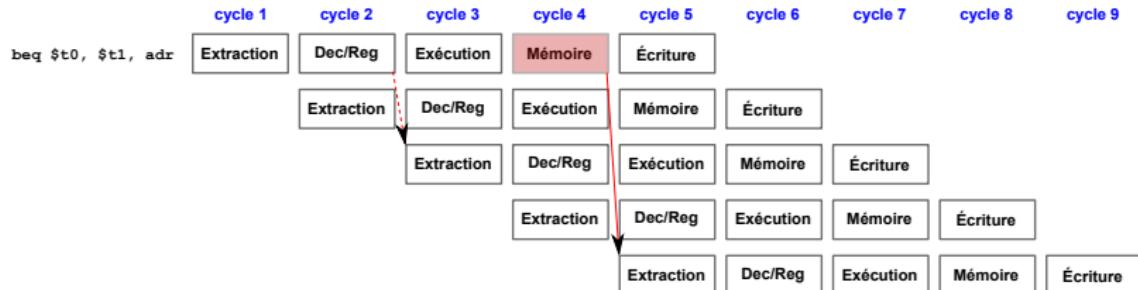
- ▶ Instruction de chargement extraite au cycle 1
- ▶ Valeur de `$t0` disponible seulement au cycle 6
 - ➡ Délai de trois instructions

Solutions possibles :

- ▶ Envoi de la donnée dès le cycle 4
- ▶ *Read after Write*



Problème du branchement différé



- ▶ Instruction de branchement extraite au cycle 1
- ▶ Adresse de destination écrite dans PC en fin de cycle 4
- ▶ Instruction de destination chargée au cycle 5
 - ➡ Délai de trois instructions

Solutions possibles :

- ▶ Envoi de la donnée dès le cycle 2 (ajout de matériel)
- ▶ Prédiction



Aléas empêchent une instruction de s'exécuter dans son cycle

Aléas structurels. Le matériel ne supporte pas la combinaison d'accès (e.g., lecture et écriture en mémoire dans le même cycle)

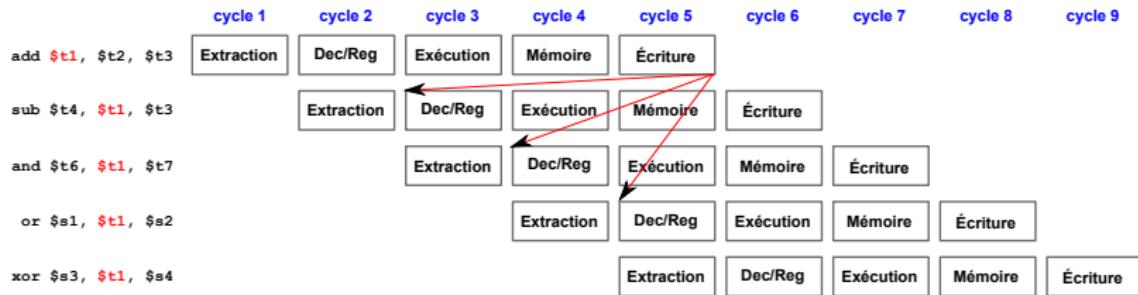
Aléas de données. Une instruction dépend du résultat d'une instruction dans le pipeline (e.g., cas du chargement)

Aléas de contrôle. Des instructions suivant un branchement sont chargées sans que l'on en sache encore si elles doivent être exécutées

- ▶ Solution matérielle : bulles de suspensions
- ▶ Solution logicielle : ajout de nops
- ▶ Solution interne : envoi de données

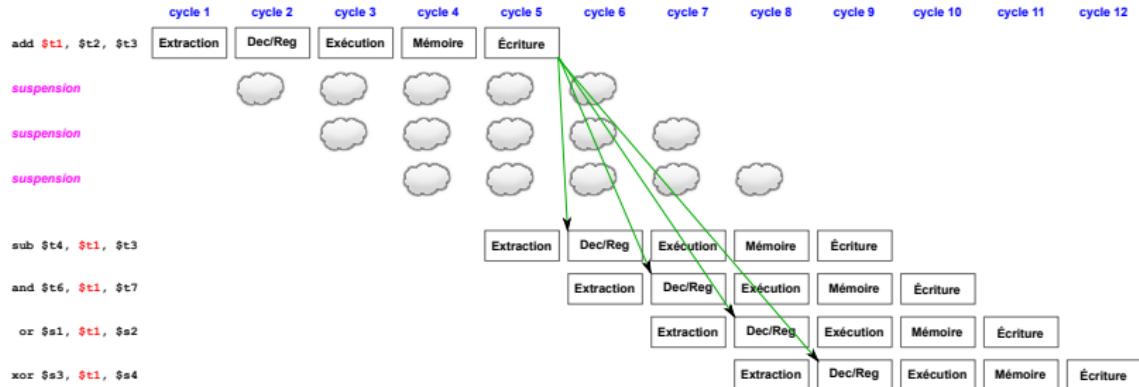


Gestion des aléas



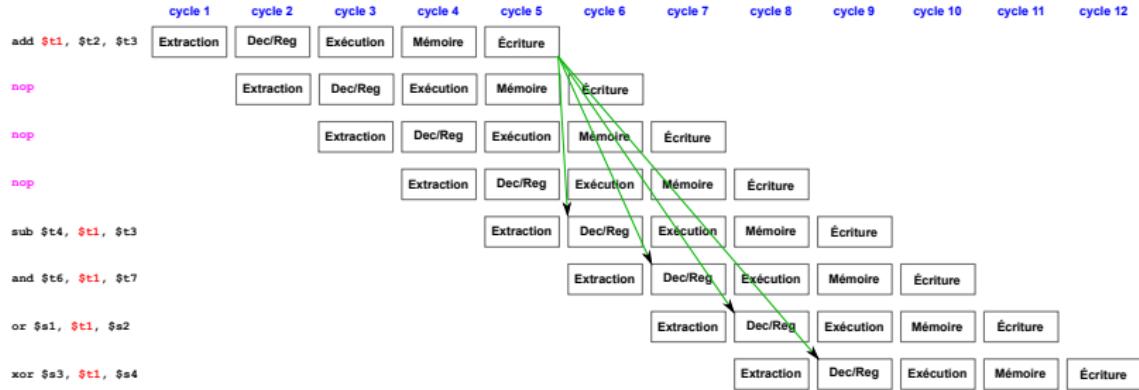


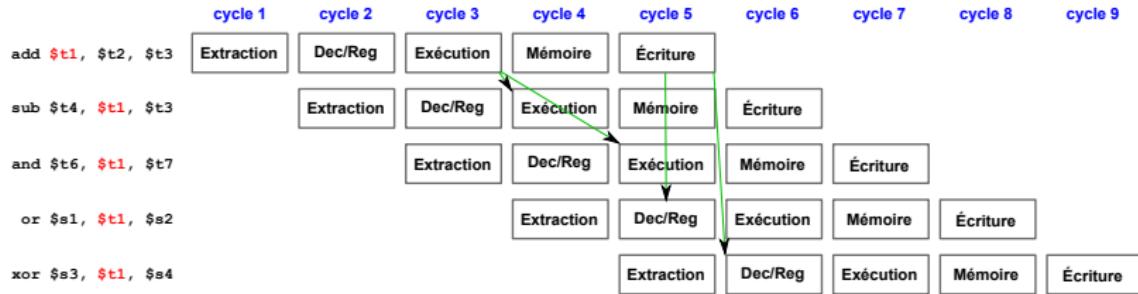
Gestion des aléas





Gestion des aléas

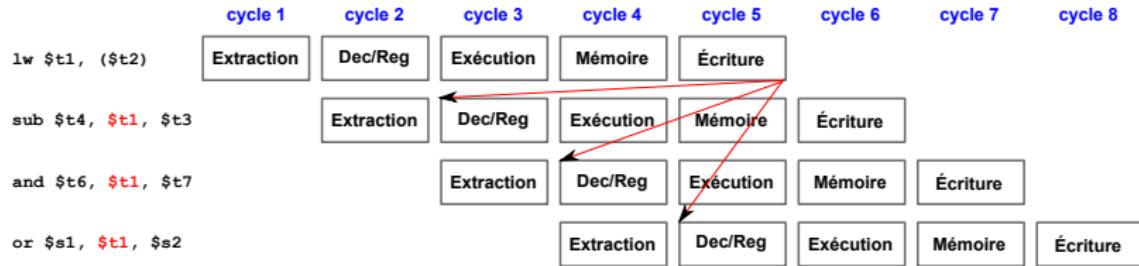




► *Read after Write*

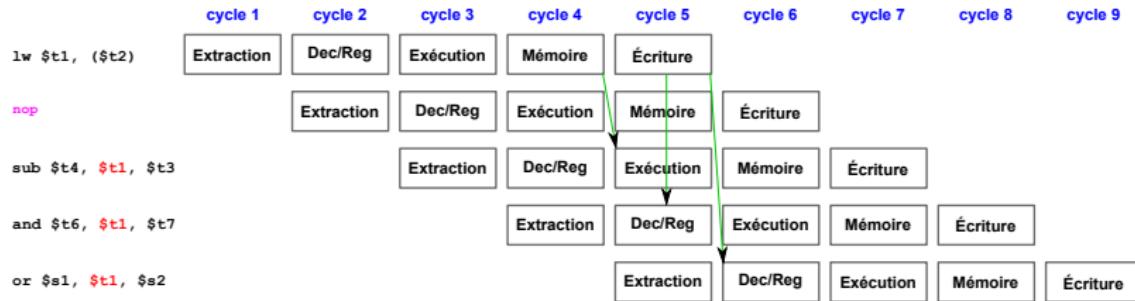


Aléas de données pour le chargement





Aléas de données pour le chargement



- ▶ Envoi de donnée dès qu'elle est disponible dans le pipeline
- ▶ *Read-after-write*

Caches



RAM (*Random Access Memory*) :

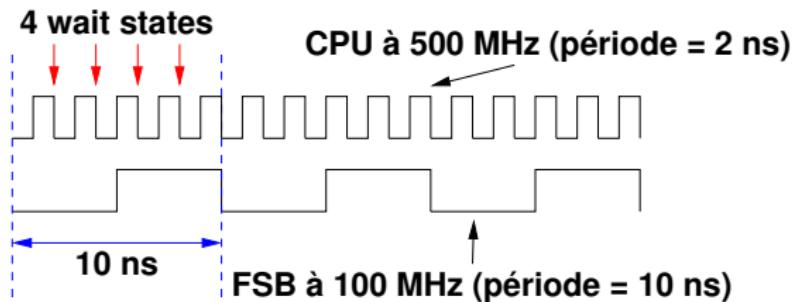
- ▶ Adressage direct et non séquentiel
- ▶ Mémoire volatile : contenu disparaît à l'extinction du PC
- ▶ **Vision conceptuelle** : tableau d'octets

Barette SIMM (*Single In-line Memory Module*) :



Temps d'accès mémoire : temps mis entre une demande d'opération (lecture/écriture) par le CPU et la fin de son exécution.

- ▶ CPU et RAM travaillent à des vitesses très différentes
- ▶ Échange de données à une cadence différente de celle de l'horloge du CPU
 - le FSB (*Front-Side Bus*) a sa propre horloge (66–133 MHz)
 - Protocole d'échange semi-synchrone
 - Insertion d'états d'attente du CPU (*wait states*)





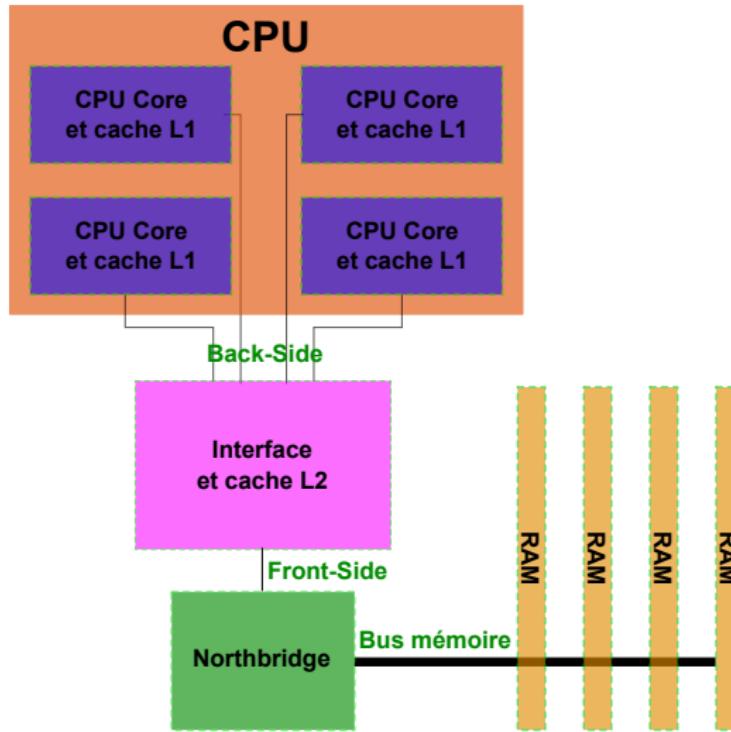
- ▶ Introduction des *wait states* : coûteux en temps
- ▶ Élimination des *wait states*?
 - ▶ Utilisation de mémoire plus rapide (cher)
 - ▶ Utilisation des propriétés des programmes

```
for (int i=0; i <10; ++i) {  
    T[i] = 0;  
}
```

- ▶ Localité temporelle des références :
 - ▶ i est accédé répétitivement
- ▶ Localité spatiale des références :
 - ▶ Toutes les cases consécutives en mémoire représentant T sont accédées



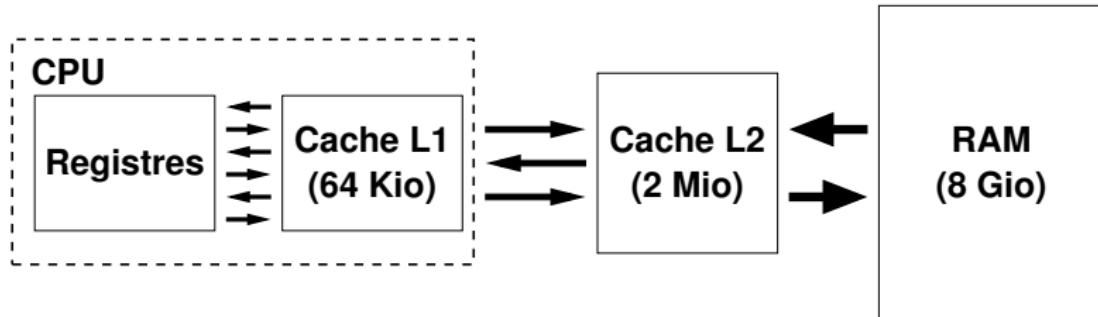
Connection CPU/RAM





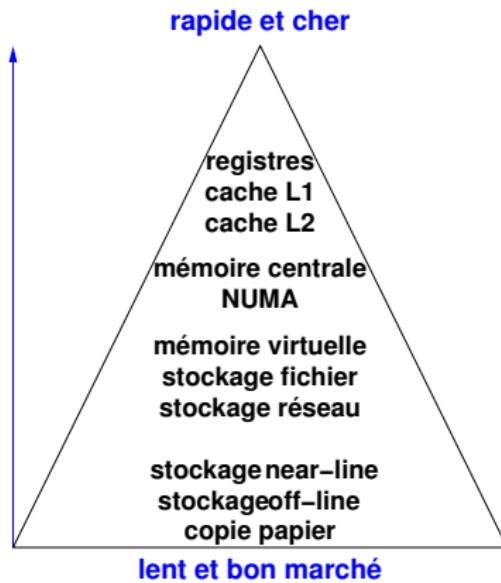
Idée : utiliser de la mémoire très rapide en petite quantité pour stocker les cases de la RAM récemment accédées ou susceptibles de l'être bientôt

- ▶ Mémoire cache de niveau 1 (cache L1) sur le CPU
- ▶ Mémoire cache de niveau 2 (cache L2) à côté de la RAM





Hiérarchie des mémoires



NUMA : *Non-Uniform Memory Access* (e.g., mémoire sur un autre processeur)

near-line : jukebox automatisé

off-line : montage manuel



Succès. La donnée recherchée est trouvée au niveau attendu

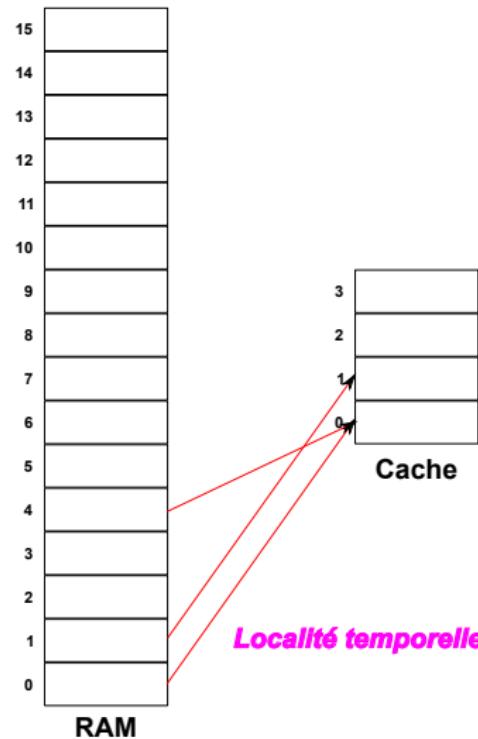
Défaut. La donnée recherchée n'est pas trouvée au niveau attendu

Taux de succès. Part des accès donnant lieu à un succès

Taux de défaut. Part des accès donnant lieu à un défaut

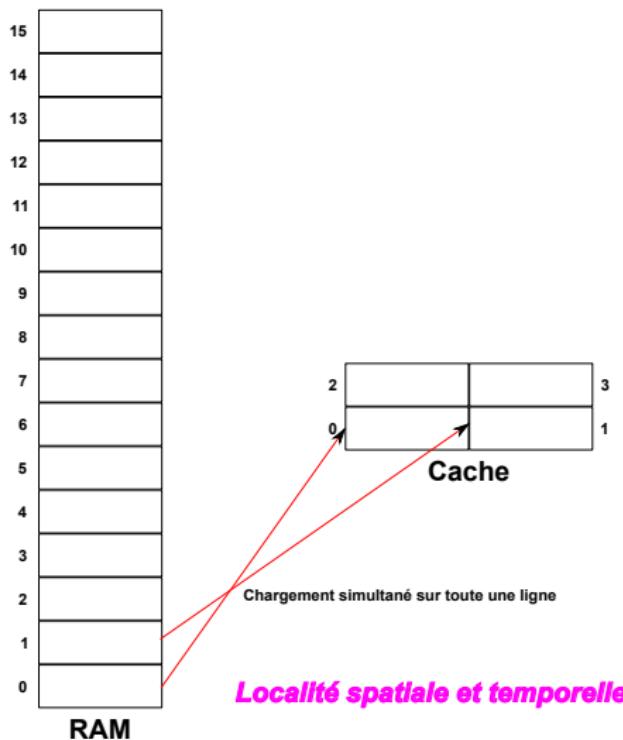


Cache à correspondance directe :





Cache à correspondance directe :



Fin du cours