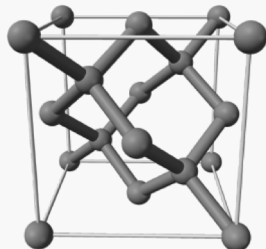
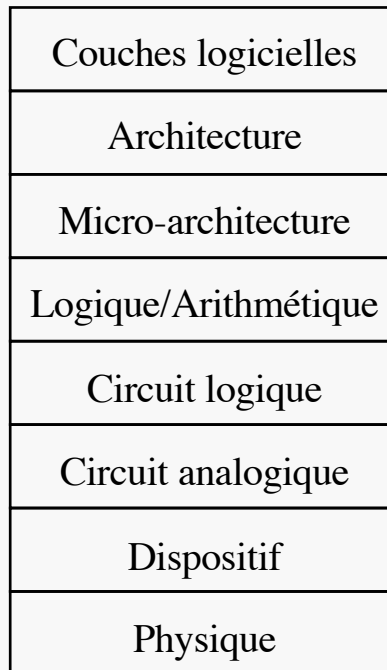


```

public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        TcpClient server;
        try
        {
            server = new TcpClient("127.0.0.1", 8080);
        }
        catch (IOException ex)
        {
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int recv = ns.Read(data, 0, data.Length);
        string data = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(string.Format("Received: {0}", data));
        while (true)
        {
            input = Console.ReadLine();
            if (input == "exit") break;
            newchill.Properties["url"] = "http://www.google.com/search?q=" + input + "&btnG=Search";
        }
    }
}

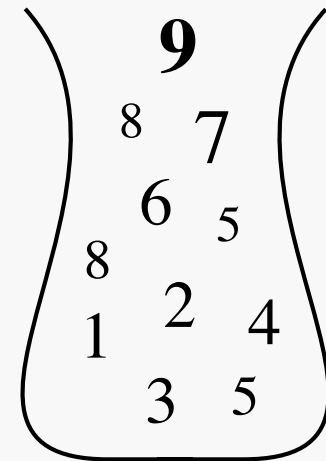
```



MIPS

version 32 bits

Reduced
Instruction
Set
Computer

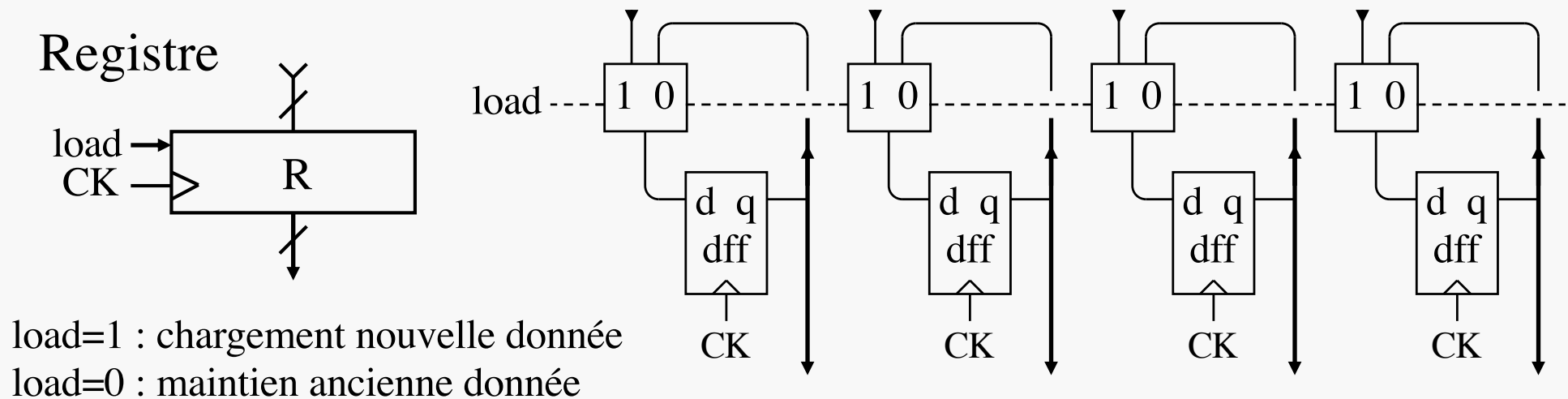


ARCHITECTURE ET COMMANDE D'UN (CŒUR ÉLÉMENTAIRE DE) MICROPROCESSEUR

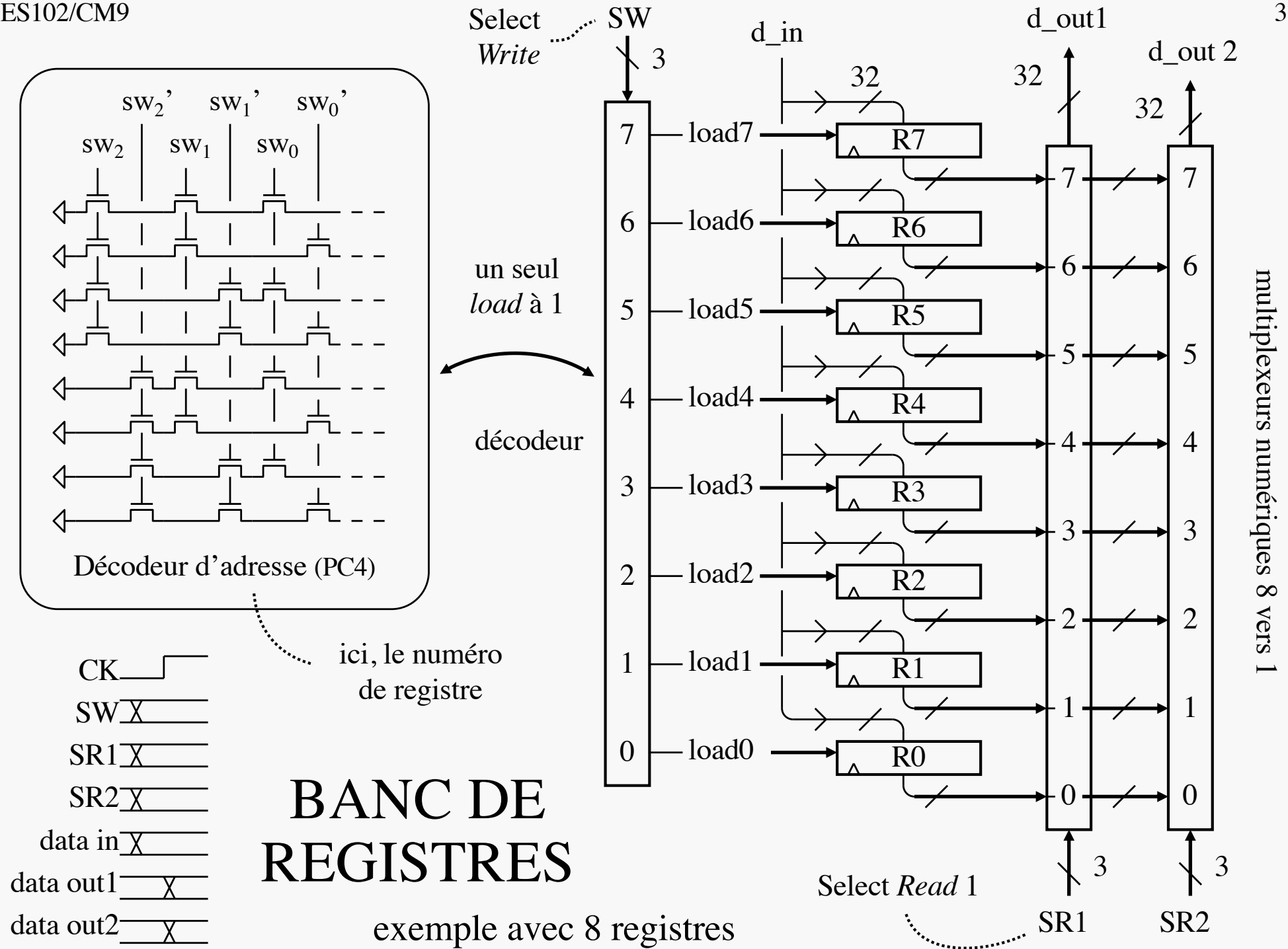
ES102 / CM9

BESOIN DE PLUS DE REGISTRES

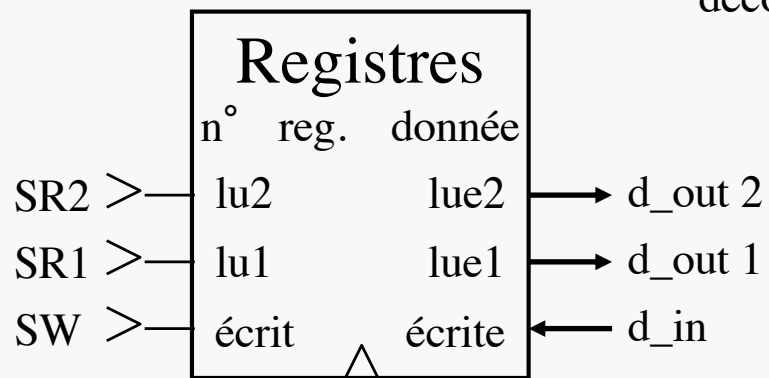
(que dans la calculette primitive) pour supporter des algorithmes courants



- chaque registre portera typiquement une variable de calcul
- on veut pouvoir en combiner 2 au choix à chaque période d'horloge
pour cela, il faut pouvoir récupérer le contenu de n'importe quel couple de registres parmi de multiples accessibles, faire une opération sur les deux opérandes ainsi récupérés, et ranger le résultat dans n'importe quel registre
→ registres regroupés en *banc* pour répondre à ce besoin



BANC DE REGISTRES (2)

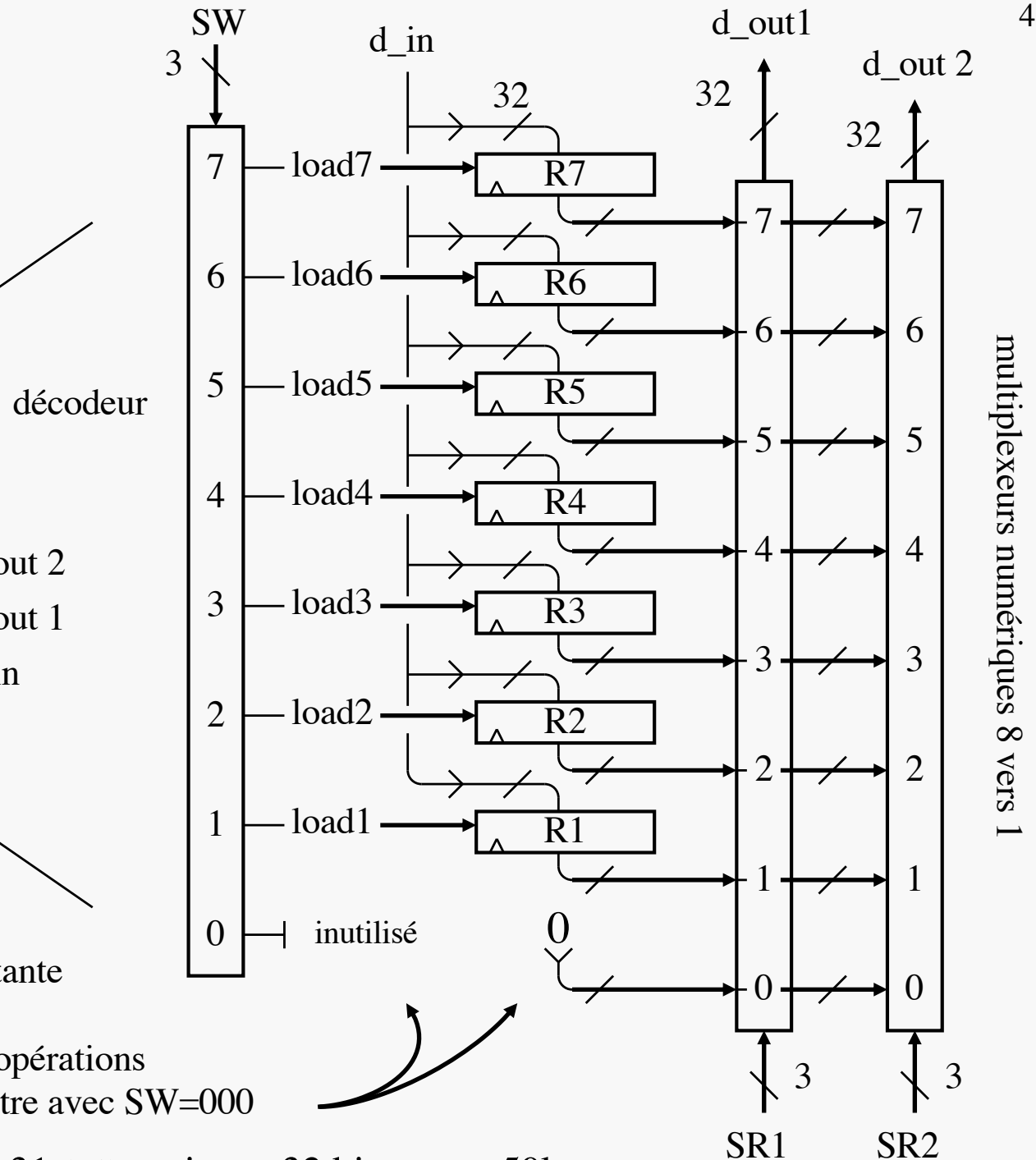


Les registres ont aussi des « petits noms », selon certaines spécificités : t0, s1, ra, a2, v0, sp...

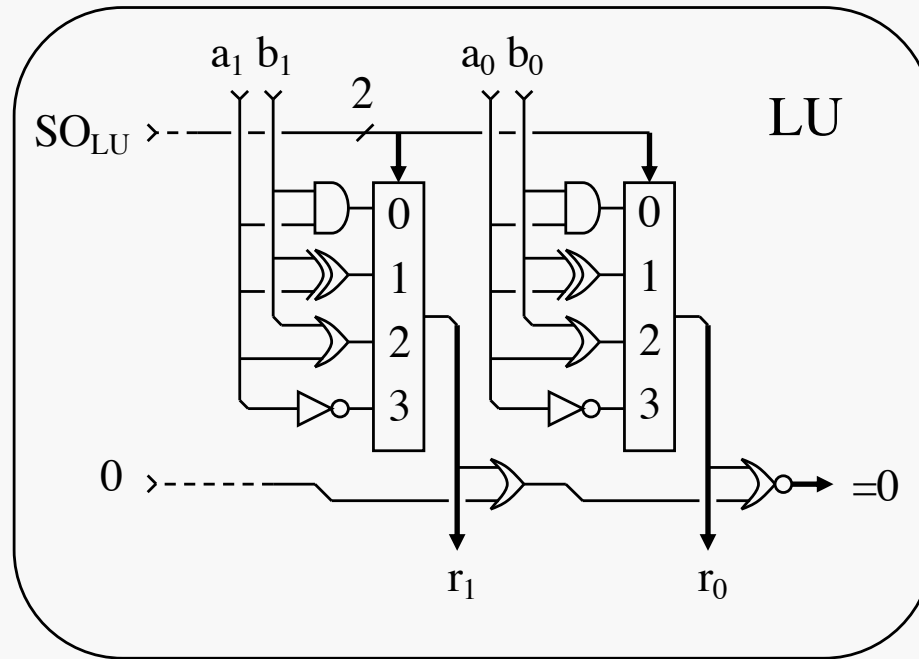
R0 usuellement remplacé par la constante numérique 0 et appelé *registre zéro* :

- utile pour particulariser certaines opérations
- permet de n'écrire sur aucun registre avec SW=000

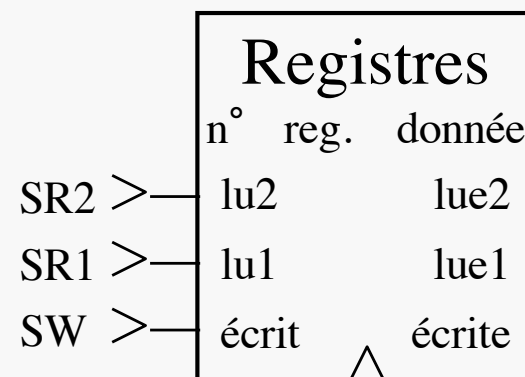
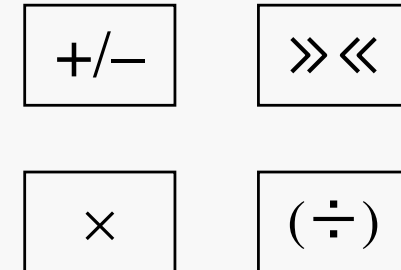
Désormais, architecture MIPS I : 31 (-1) registres 32 bits $\rightarrow \sim 50\text{kt}$



CHEMIN DE DONNÉES POUR PROCESSEUR



SO :
Select
Operation



SO
 ≥ 0
 $= 0$

ALU = Unité
Logique &
Arithmétique

ALU

{ $SR1$, $SR2$, SO , SW } : commande appliquée au CD

DONNÉES EN MÉMOIRE

Quelques données seulement dans les registres, toutes les autres en mémoire

Adresse	Exemple de contenu→ par ligne/mot de 32 bits, soit 4 octets
0 0-000000	00000000000000000000000000000001	? entier 1
4 0-000100	11111111111111111111111111111111	? entier -1
8 0-001000	01111111110101010101010101010101) réel 1/6 en format flottant 64 bits (cf. PC2/Exo2)
12 0-001100	01010101010101010101010101010101	
16 0-010000	01000101010100110011000100110000) chaîne « ES102 » en code ASCII / UTF-8
20 0-010100	00110010000000000000000000000000	
24 0-011000	
..	

Chaque octet
a une adresse

‘E’ ‘S’ NUL ‘0’

En C, les chaînes
de caractères sont
stockées octet par
octet et « terminées »
par un NUL

Et même plusieurs
si nécessaire, pour
finir la ligne
(alignement)

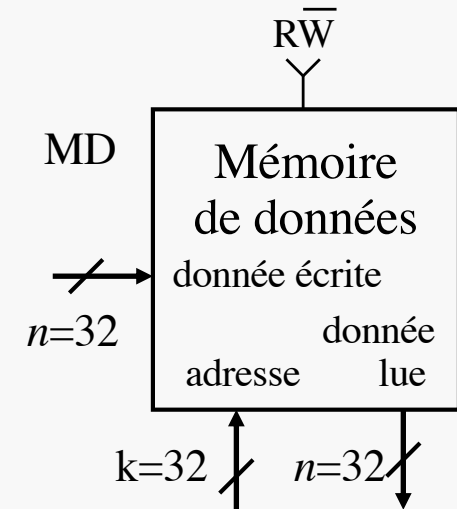
.....→ Chaque ligne aussi :
celle de son 1er octet
⇒ cette adresse s’incrmente
de 4 à chaque ligne

fin de
« ES102 »
à l’adresse
21 ici

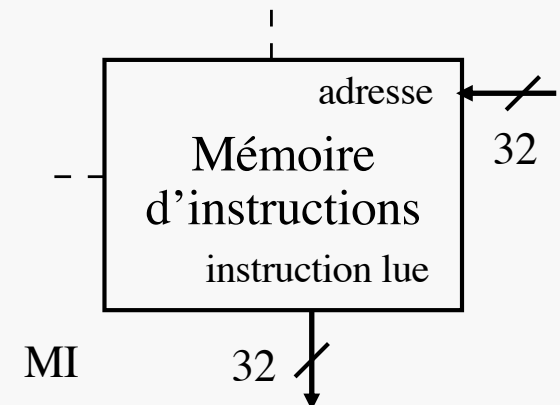
MÉMOIRE

- tableau de bits
 - de capacité 2^k octets, chacun avec une adresse propre : son numéro, sur k bits
 - $k=32 \rightarrow 4$ gigaoctets (désormais ridicule)
 - $k=64 \rightarrow 16$ exaoctets (gigantesque)
 - de largeur n bits donc comportant $8 \cdot 2^k / n$ lignes chacune accessible en lecture ($R\bar{W} = 1$) ou en écriture ($R\bar{W} = 0$)
 - sur présentation de son adresse
 - comparé aux registres, stockage plus dense mais accès plus lent, car séquençement plus complexe
 - *black-out* : mémoires considérées combinatoires ci-après
 - accueillant des données de différentes tailles, selon leur type, multiple de n sauf exception
 - il faut sauter du bon nombre d'octets pour passer à la donnée suivante d'un tableau...
 - accueillant aussi des adresses de données
 - ou encore des instructions (mémoire dédiée ici)

(R : read
W : write



MIPS I :
 $k = n = 32$



*Planche retirée de l'édition 2020, mais
fantôme maintenu par souci de cohérence
de numérotation avec la vidéo.*

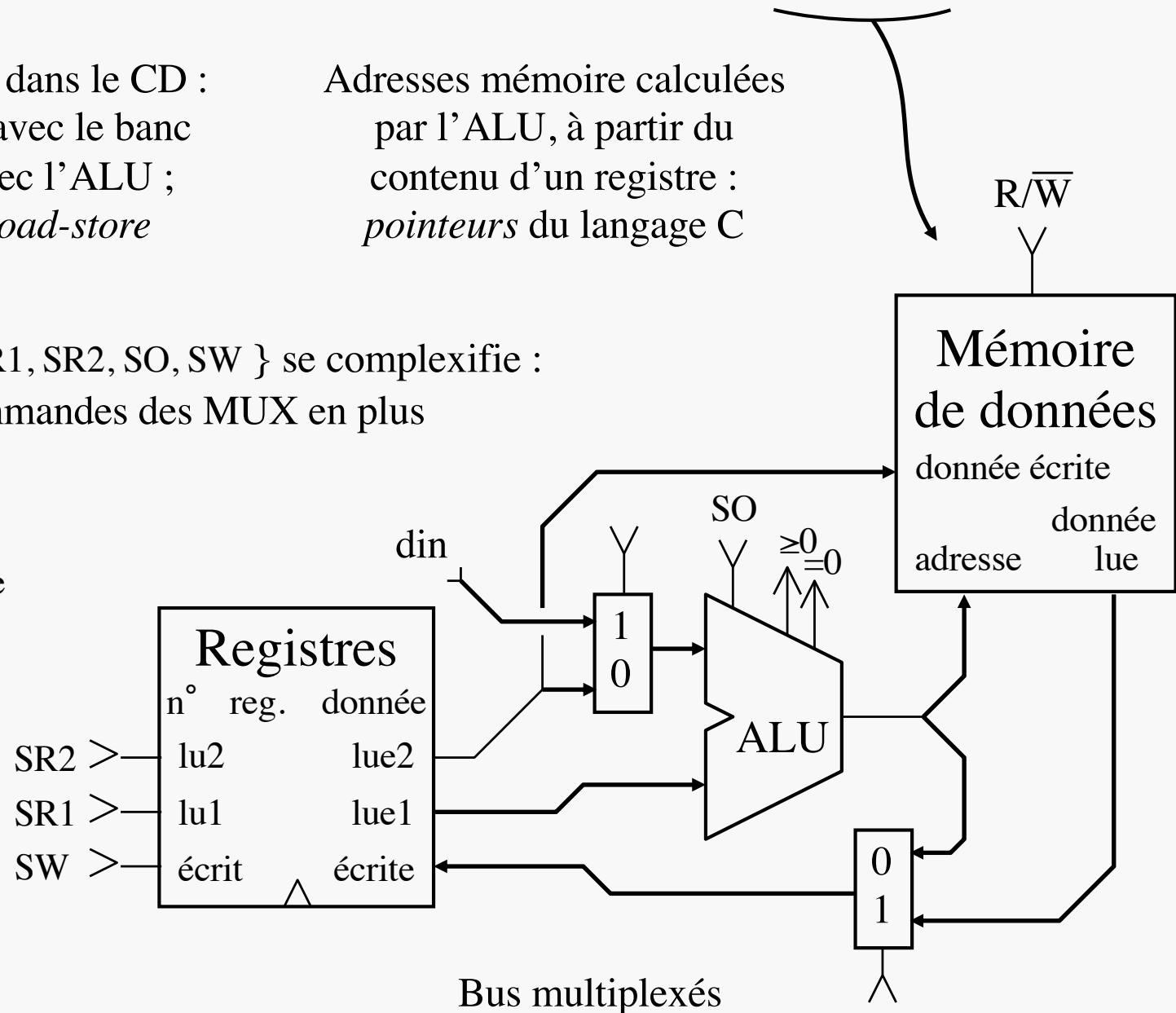
CHEMIN DE DONNÉES ÉTENDU

Mémoire de données dans le CD :
échange seulement avec le banc
de registres, pas avec l'ALU ;
architecture dite *load-store*

Adresses mémoire calculées
par l'ALU, à partir du
contenu d'un registre :
pointeurs du langage C

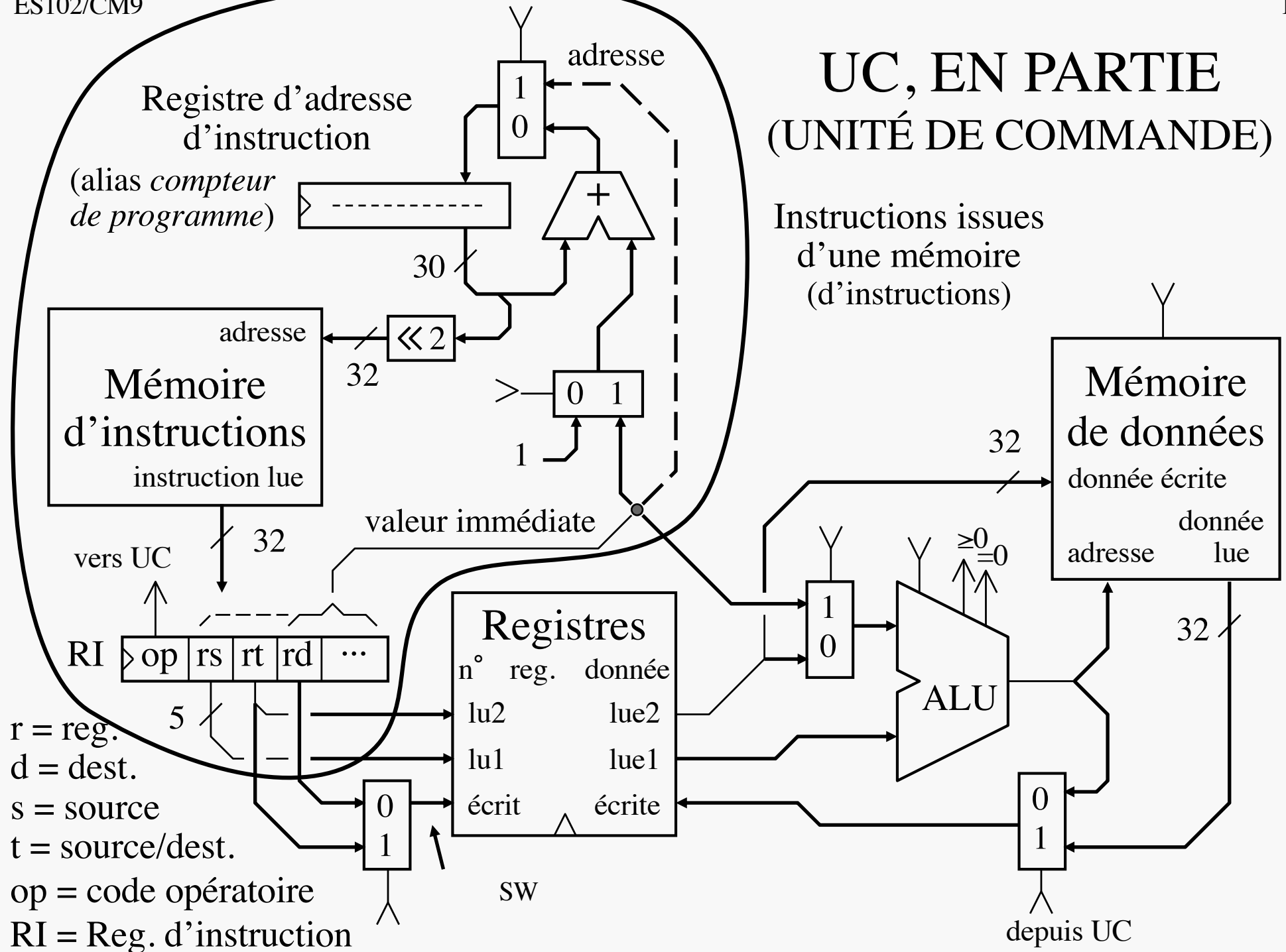
La commande { SR1, SR2, SO, SW } se complexifie :
R/ \overline{W} et commandes des MUX en plus

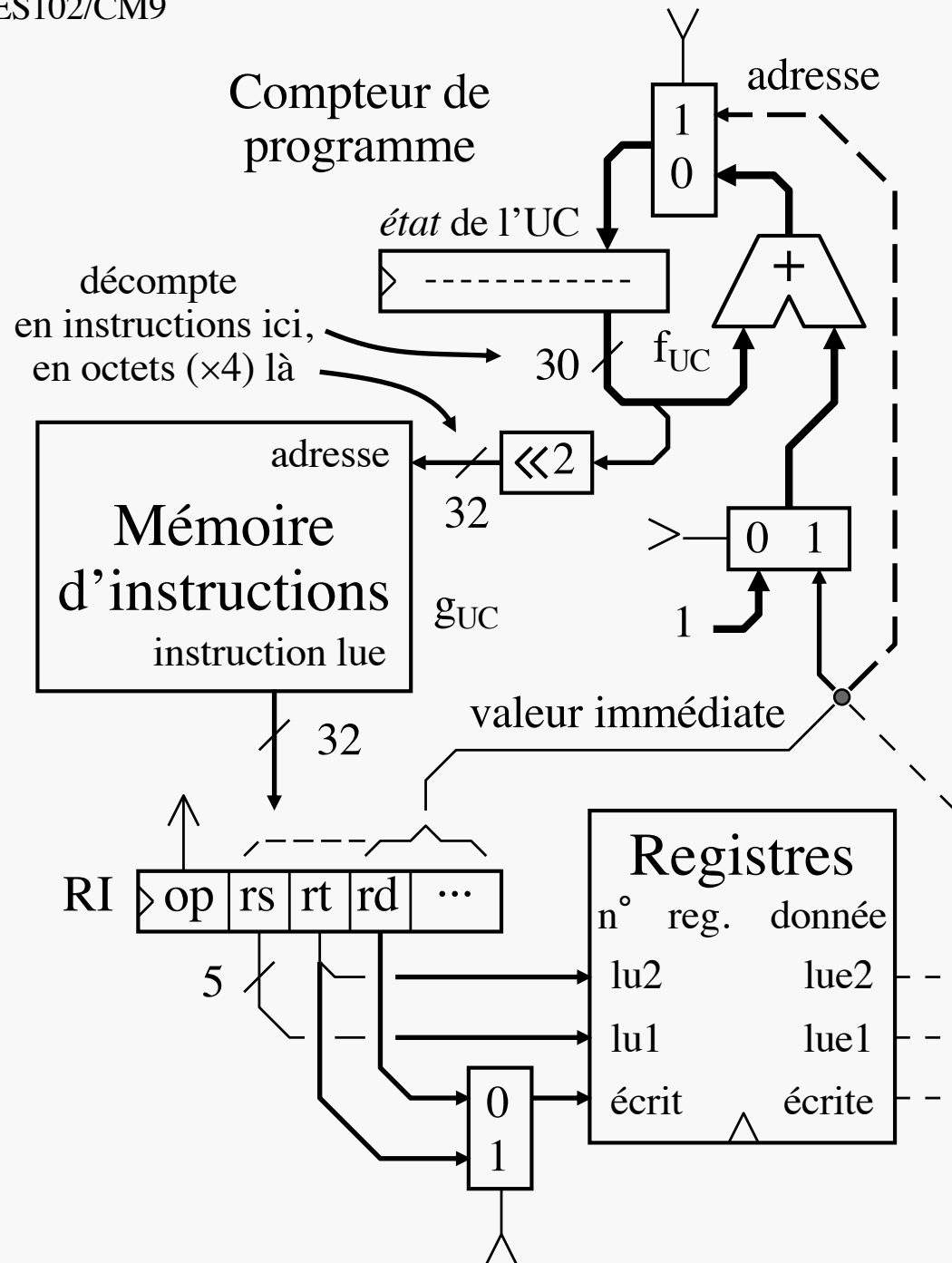
Entrée *din* pour usage
ultérieur spécifique



UC, EN PARTIE (UNITÉ DE COMMANDE)

Instructions issues
d'une mémoire
(d'instructions)





CHARGEMENT INSTRUCTION SUIVANTE

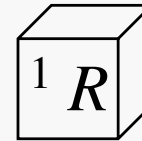
- Mécanisme atomique pour exécuter une suite prédéterminée d'opérations
 - indépendante des données
 - DE = chaîne d'états liés par des transitions inconditionnelles
- Comportement implanté avec une mémoire d'instructions (g_{UC}) adressée par un compteur (f_{UC})
- L'adresse présentée à la mémoire d'instructions est incrémentée de 4 à chaque période d'horloge pour désigner l'instruction suivante, située 4 octets plus loin (car largeur d'instruction = 32 bits)

INSTRUCTIONS

- 3 types :

- Opérations sur les données :

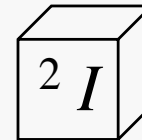
- addition, soustraction, multiplication
 - opérations logiques
 - décalages (tournant ou pas), copie



fait intervenir les
registres seulement

- Transferts de données :

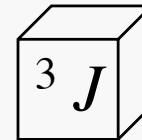
- de registre vers mémoire (écriture, *store*)
 - de mémoire vers registre (lecture, *load*)



accès *indirect*
aux données
via leur adresse
en mémoire → *pointeur*

- Gestion de séquençement :

- sauts (*jump*) et branchements (*branch*)
 - sous-programme : appel & retour
(*call & return*)



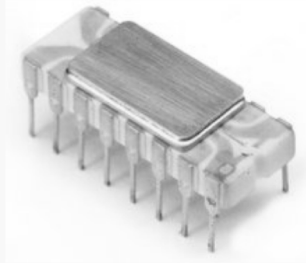
procède par saut
d'instructions (*jump*)

- environ 50 instructions au total, d'où les 6 bits alloués au code opératoire

→ examen de 2-3 exemples caractéristiques pour chaque type

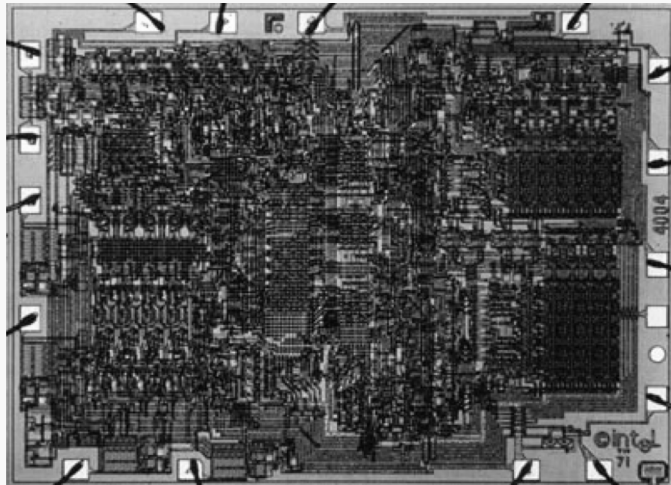
- leur raison d'être (logicielle) et leur réalisation (économe) sur CD + UC,
 - leur représentation symbolique → *langage assembleur MIPS*

1) *add, addi* 2) *lw, sw* 3) *beq, j, jal, jr*



Intel 4004

LE PREMIER MICROPROCESSEUR



novembre 1971



Federico Faggin

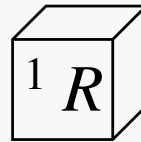


Ted Hoff

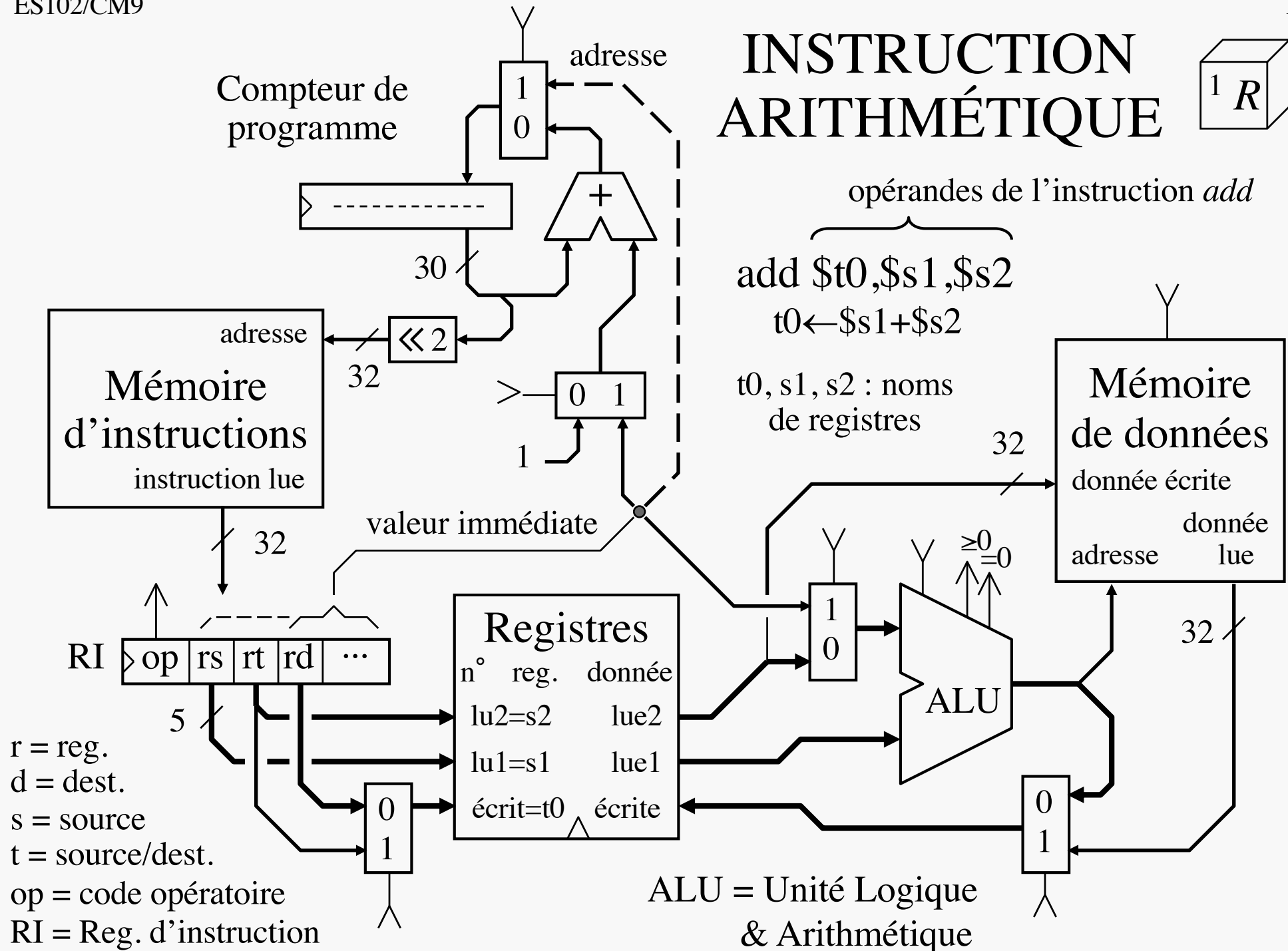
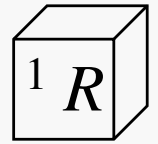


Stan Mazor

3kt, pMOS $10\mu\text{m}$, 740kHz, 16 registres entiers 4 bits, adresses 12 bits, 46 instructions 8 bits
(\neq MIPS I, où tout est sur 32 bits)



INSTRUCTION ARITHMÉTIQUE



ZOOM SUR ADDI

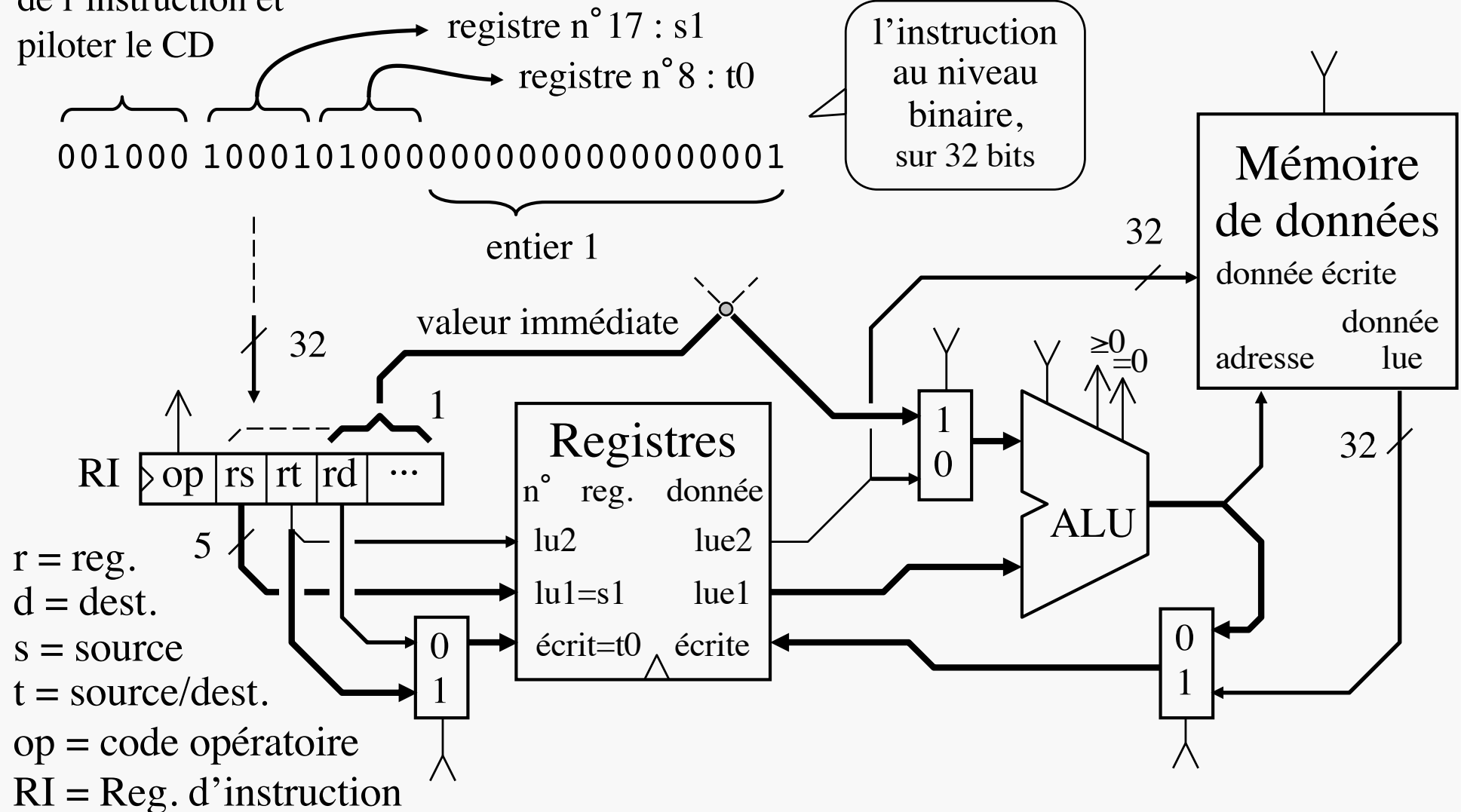
niveau symbolique

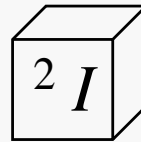
code opératoire de *addi*, à
décoder par l'UC, pour savoir
comment exploiter le reste
de l'instruction et
piloter le CD

`addi $t0,$s1, 1`

représentation en langage
assembleur de l'instruction
d'addition immédiate

l'instruction
au niveau
binaire,
sur 32 bits





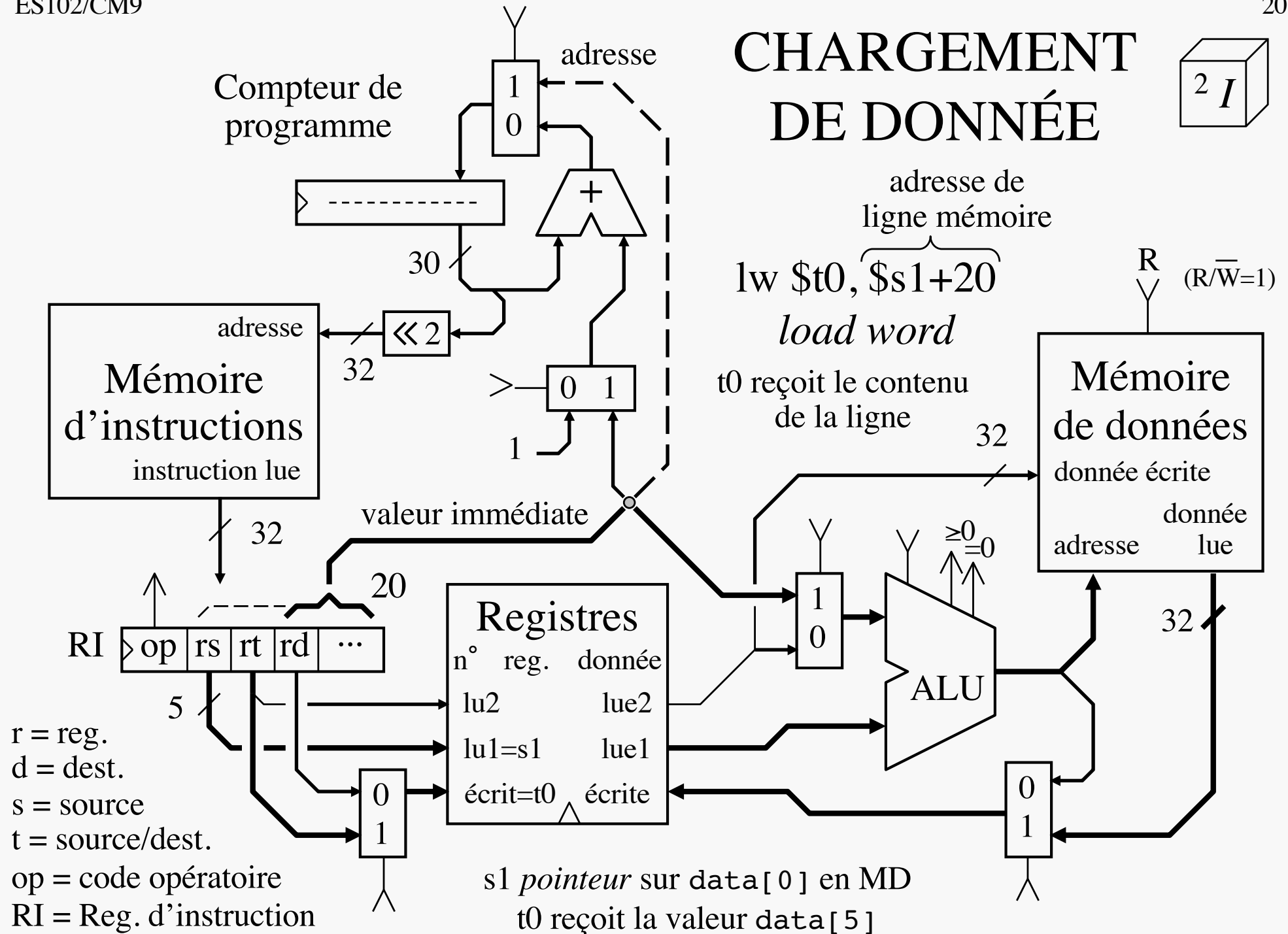
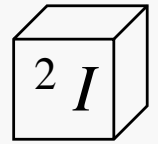
La mémoire de données (MD) entre en jeu.

Ci-après, la MD héberge un tableau d'entiers `data`, à partir d'une certaine adresse, adresse chargée dans le registre `s1`. L'élément `data[5]` se trouve donc à l'adresse `$s1+20`

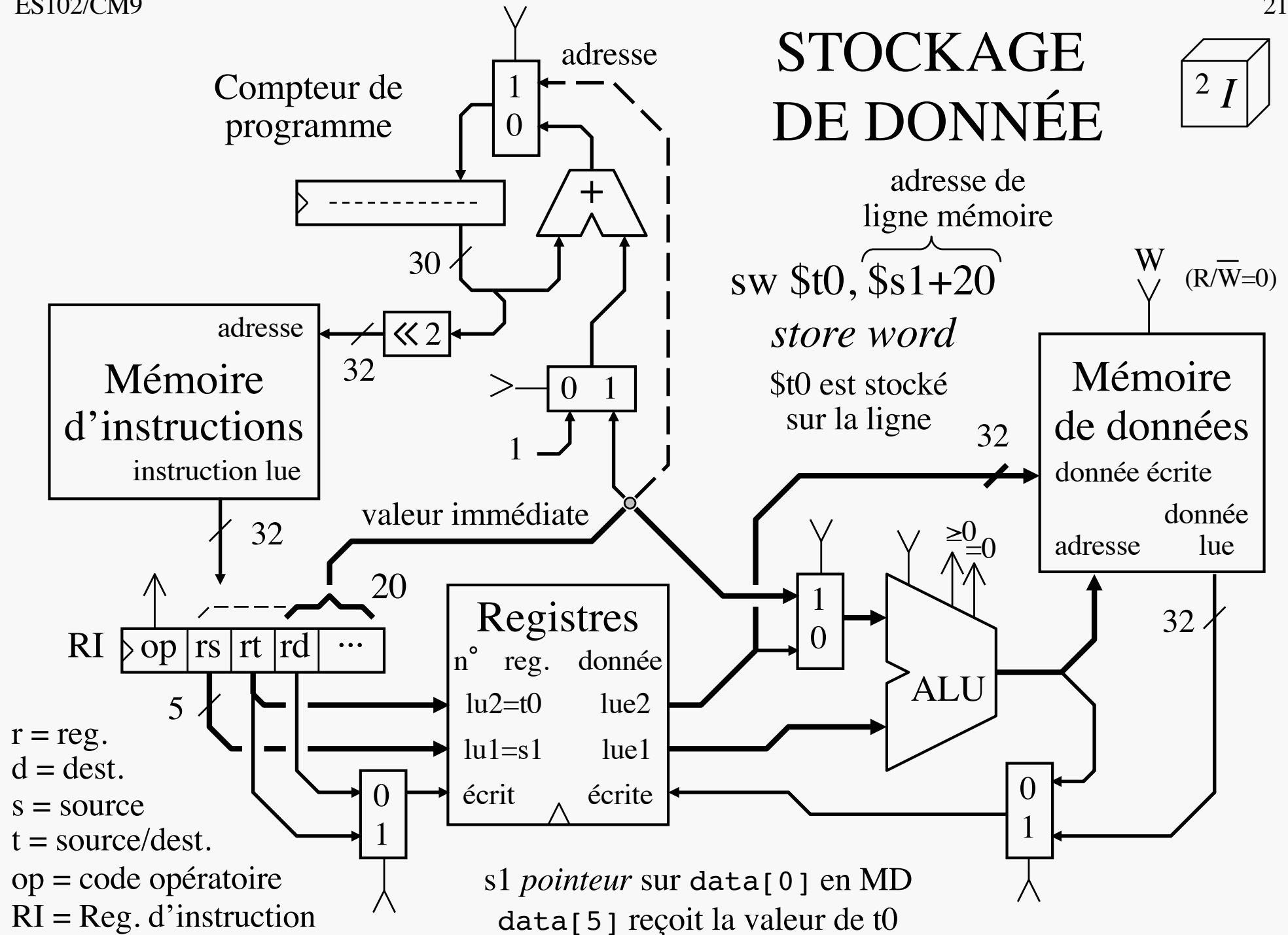
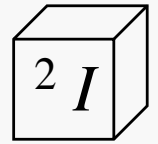
pointeur du langage C

$$20 = 5 \times (4 \text{ octets})$$

CHARGEMENT DE DONNÉE



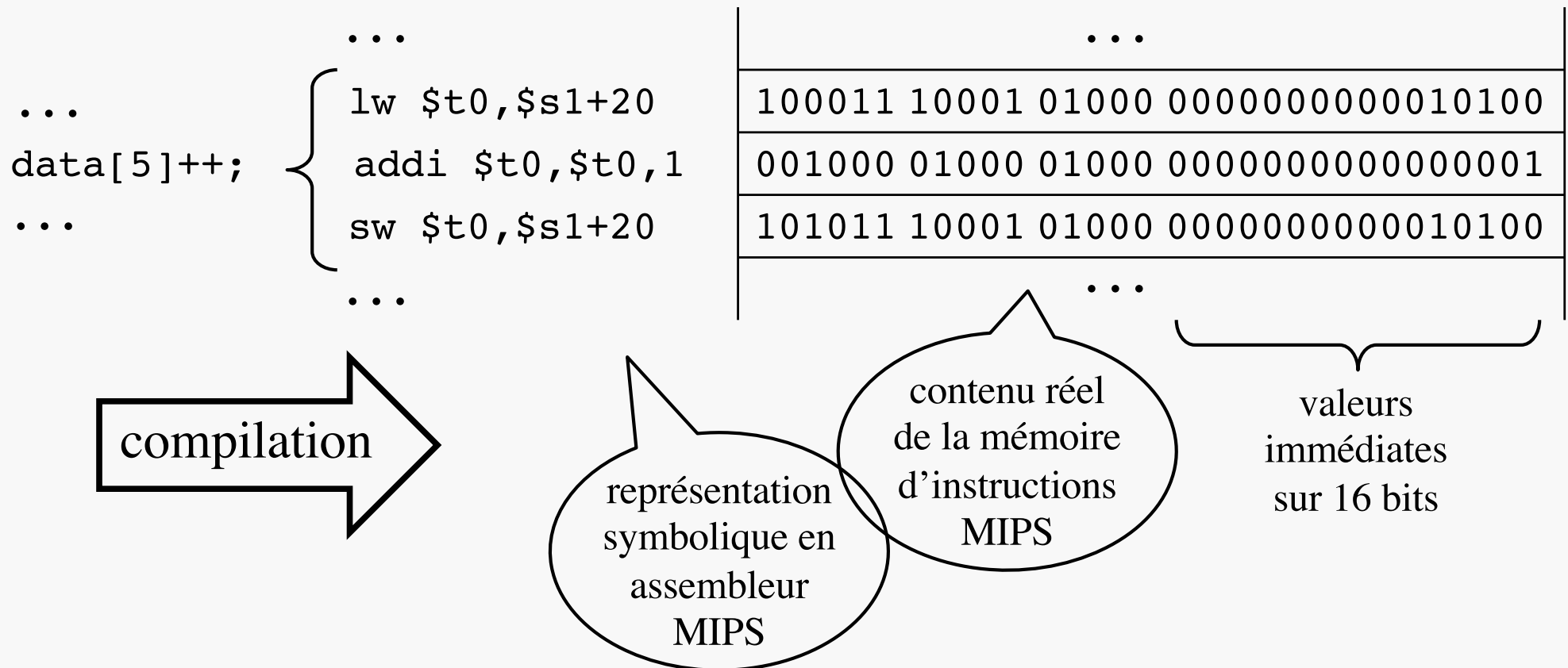
STOCKAGE DE DONNÉE

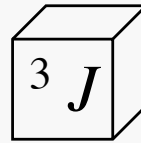


C COMPILÉ VERS MIPS

`data` : tableau d'entiers dont l'adresse de base a déjà été chargée dans `s1`

code C \longrightarrow code assembleur \equiv code machine



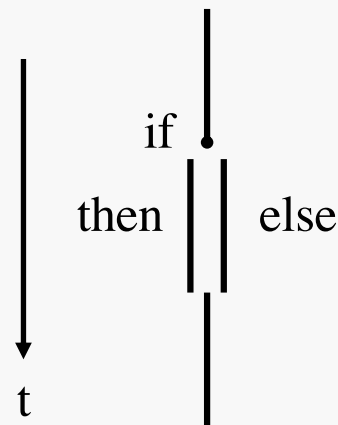
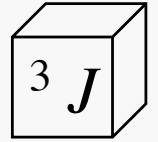


Jusqu'ici, la prochaine instruction exécutée était
la suivante en mémoire d'instructions (f_{UC} : compteur).

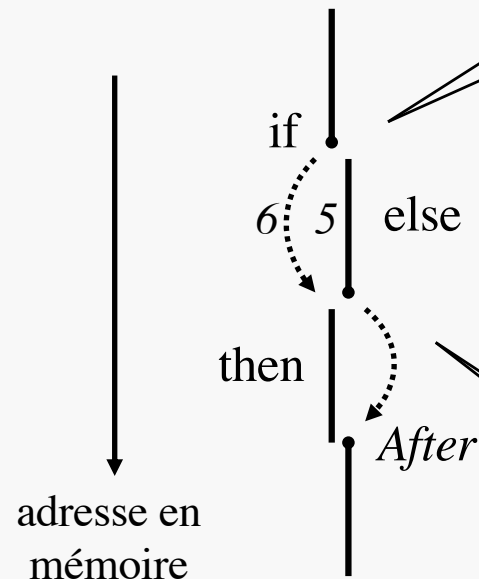
Mais certains besoins logiciels essentiels exigent de casser cette routine.

Idée : utiliser des instructions dédiées pour altérer le fonctionnement du compteur.

IF THEN ELSE



A l'exécution, selon le résultat du test *if*, on exécute soit *then* soit *else*



En mémoire d'instructions, ces segments de code sont installés les uns **derrière** les autres : les segments *else* et *then* doivent donc pouvoir être sautés.

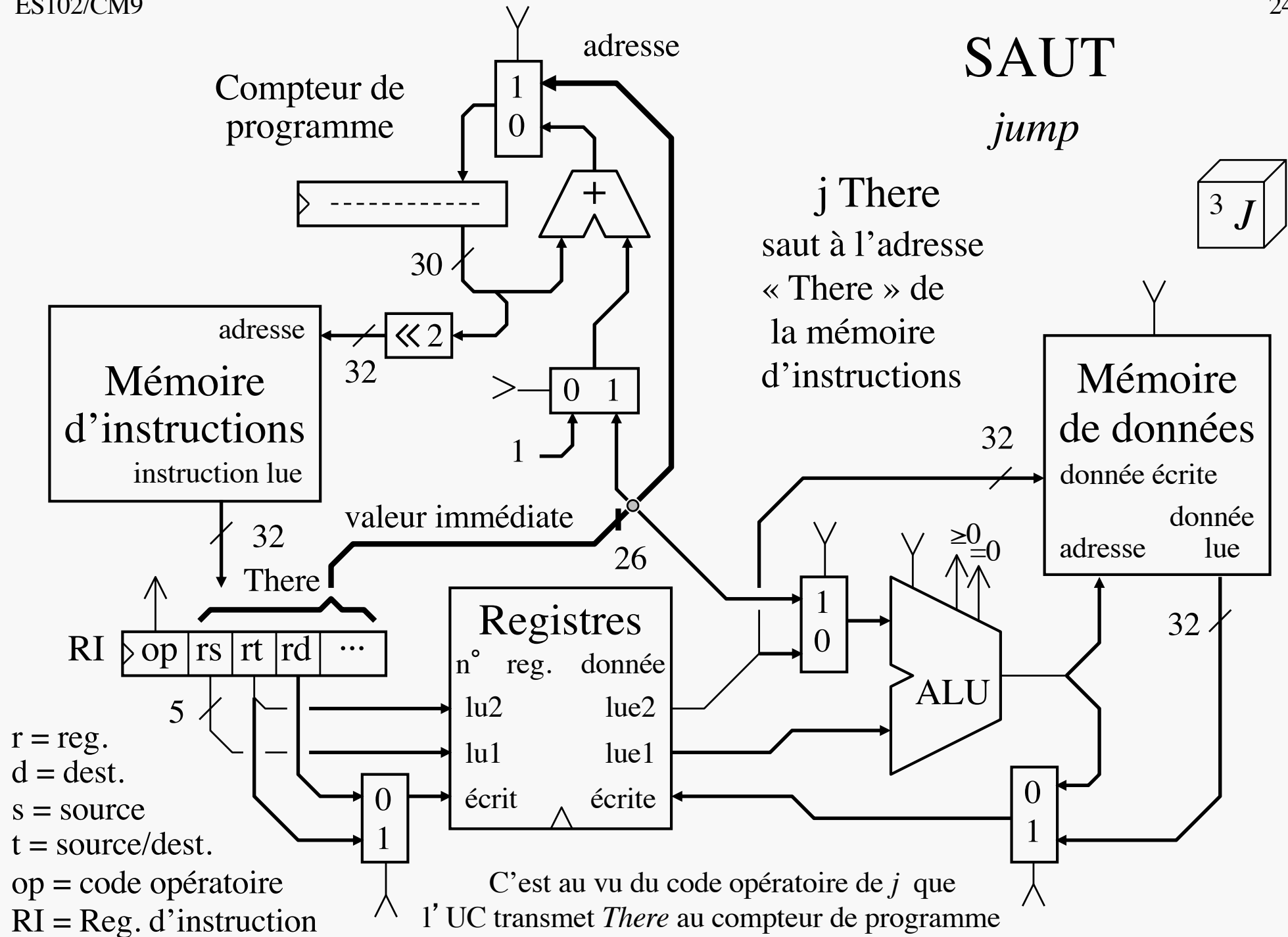
Besoin d'un *branchement* (*branch*) qui, selon le résultat du test *if*, saute vers *then* ou poursuit avec l'instruction suivante, débutant ainsi *else*

ex. avec *else* supposée de longueur 5 :
`beq $s1, $s2, +6` (*branch if equal*)
 Si $\$s1 = \$s2$, sauter 6 instructions plus loin, sinon avancer à la suivante (comme d'habitude).

Une fois *else* exécuté, besoin d'un *saut* (*jump*) systématique par dessus *then* pour atteindre *After*

`j After` (*jump*)
After = adresse symbolique, aussi appelée *étiquette*, de la première instruction après la structure if-then-else

SAUT
jump

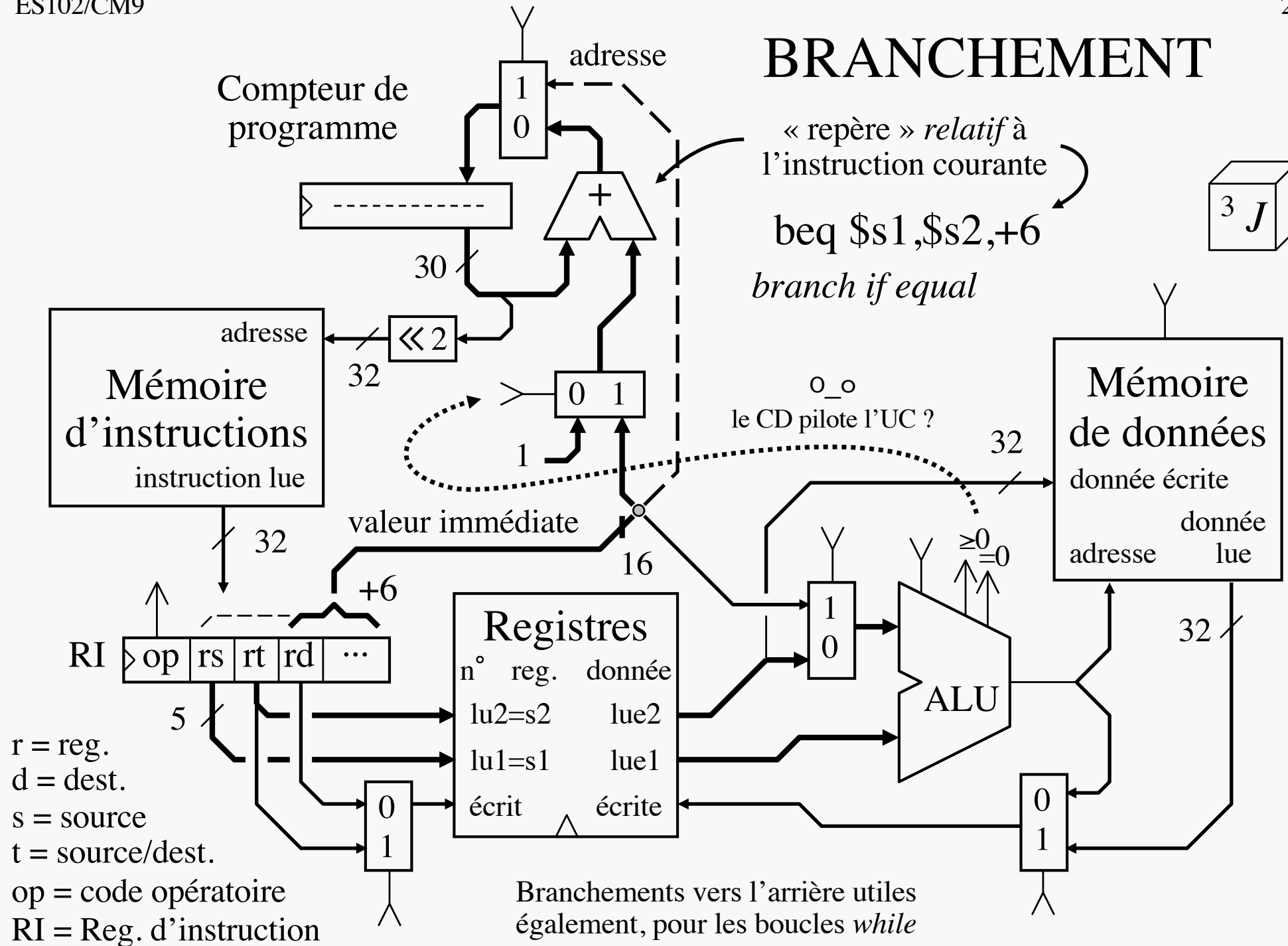
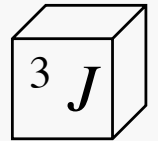


BRANCHEMENT

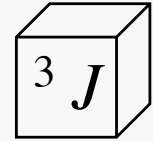
« repère » *relatif* à l'instruction courante

beq \$s1,\$s2,+6

branch if equal



SOUS-PROGRAMMES



...

n=fact(k);

...

...

int fact(int i) {

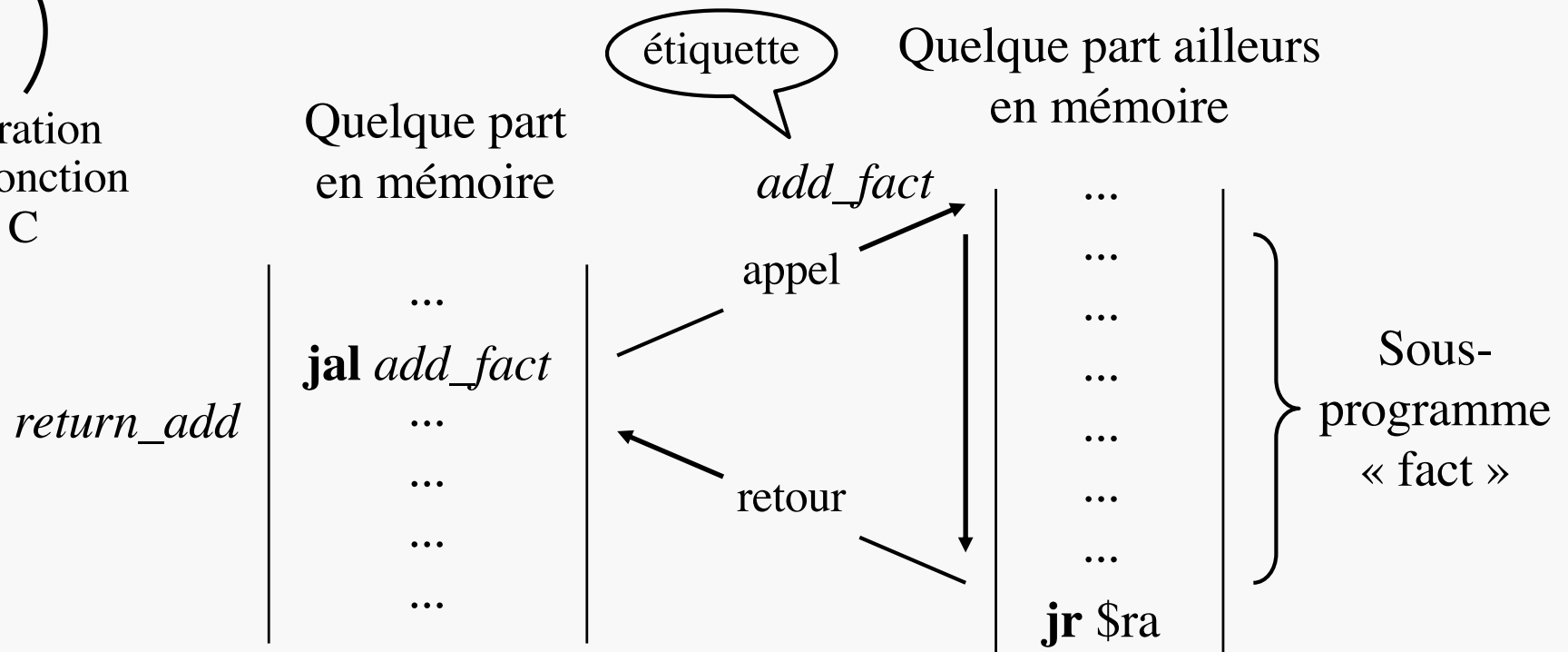
...

}

Déclaration
d'une fonction
en C

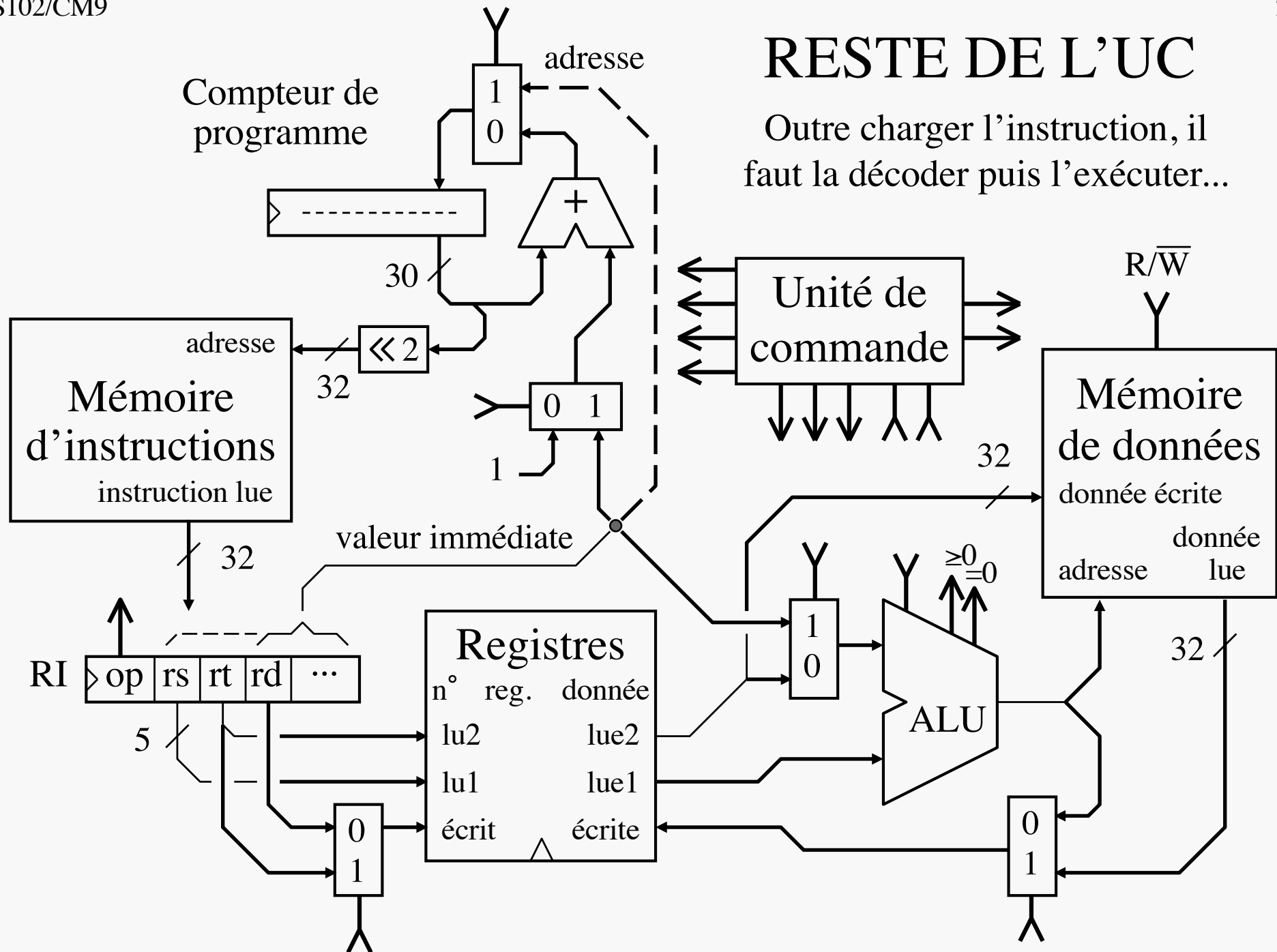
Appel { **jal** : jump and link - saute vers *add_fact* et
mémorise *return_add* dans le registre *ra*

Retour { **jr \$ra** : jump register - saute vers
l'adresse contenue dans le registre *ra*



RESTE DE L'UC

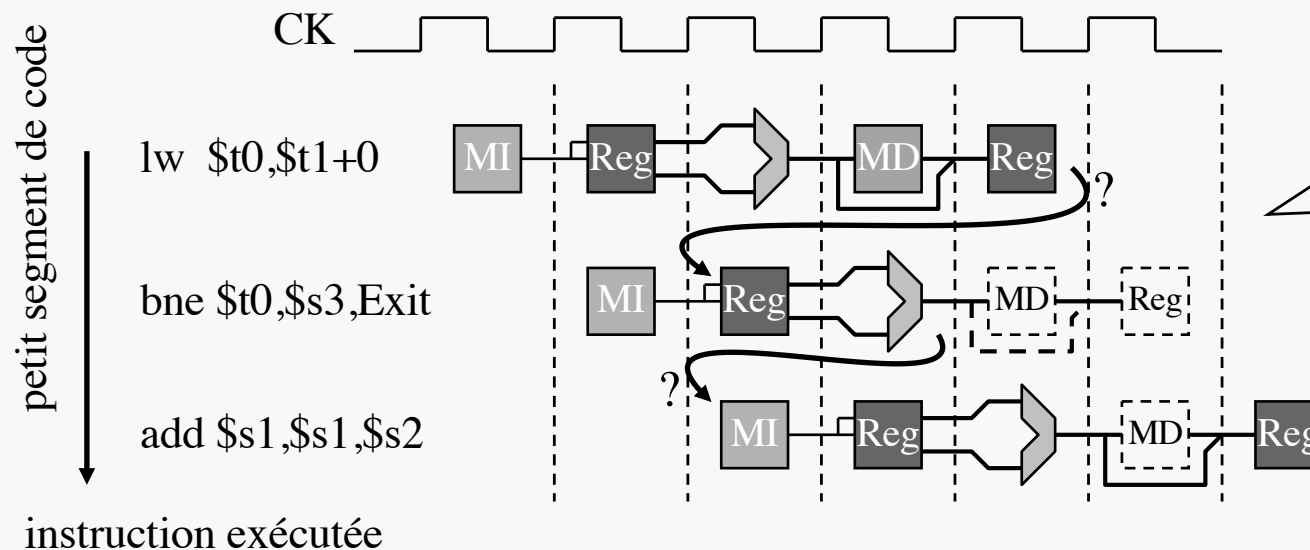
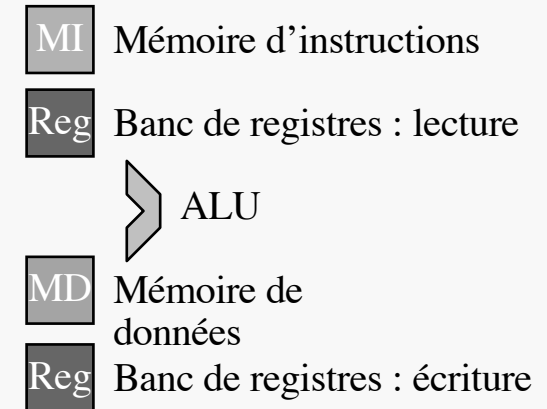
Outre charger l'instruction, il faut la décoder puis l'exécuter...



L'instruction *lw* (*load word*) de chargement de données apparaît la plus longue :

- 5 étapes successives irréductibles :
 - lecture/décodage de l'instruction
 - lecture du registre portant l'adresse de base du tableau où se trouve la donnée en mémoire
 - calcul de l'adresse de la donnée, à partir de son indice, passé en valeur immédiate, et de la base
 - récupération de la donnée en provenance de la mémoire
 - écriture dans un registre
- tentant de pipeliner ces 5 étapes en 5 étages :

PIPE-LINE



Mais :

comment utiliser
\$t0 alors qu'il n'est pas
encore calculé ?

quid de charger
une instruction sans savoir
si c'est la bonne ?

→ à l'UC de gérer...



JAGUAR CORE

Branch predict

 μ code ROM

Integer

Data tag / TLB

Bus unit

Data cache

Test

Instr. tag / TLB

Instruction cache

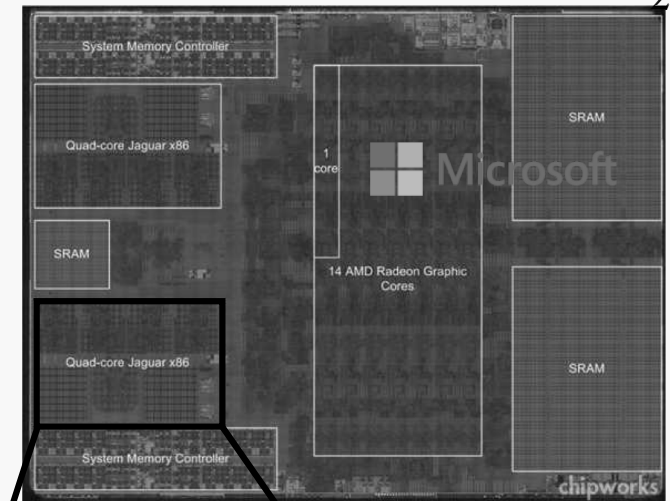
x86 Decode

Debug

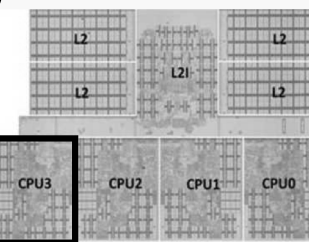
ReOrder Buffer

Load / Store

Floating point



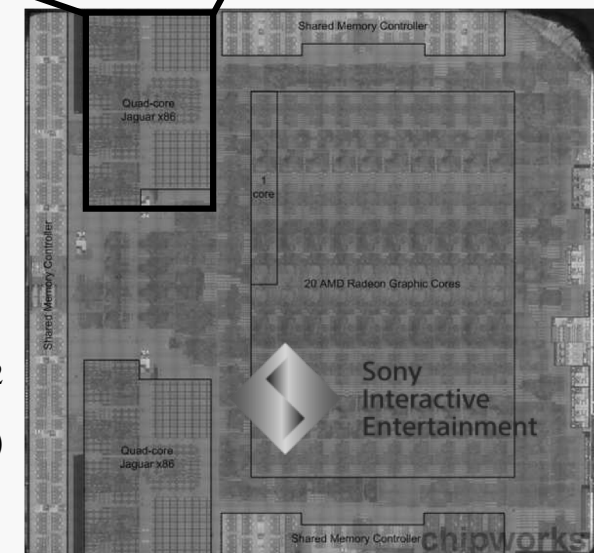
Xbox One



PS4



328 mm²
(4Gt ?)



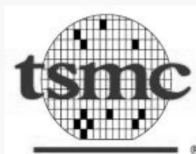
Cœur « léger » 64 bits entier / 128 bits flottant

2 voies - out of order - cache L1 64ko

3,1mm² en 28nm (40Mt ??)

basse consommation

prod. 2S13



$A_{UC} \approx A_{CD}$ ici, $A_{UC} > A_{CD}$ sur processeur + complexe