Ch. 5
Processeur

B. Quoitin
(bruno.quoitin@umons.ac.be)

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Objectifs

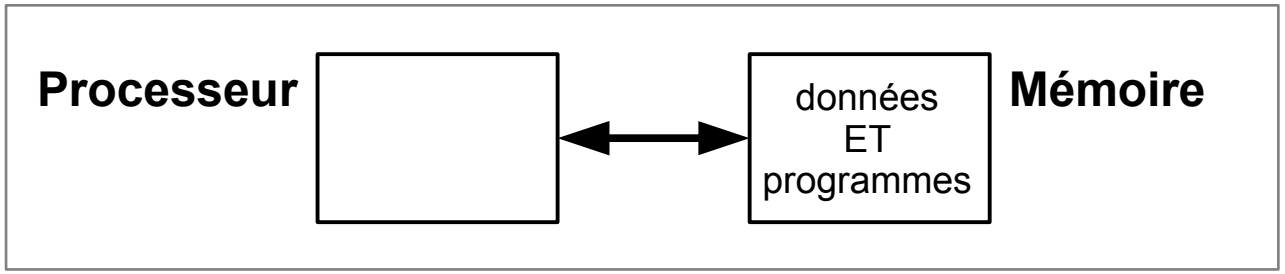
Du jeu d'instructions à la micro-architecture

- Découvrir et utiliser une architecture de processeur inspirée de l'architecture réelle **MIPS**
- Comprendre l'encodage et le rôle des différentes instructions ainsi que le fonctionnement d'un processeur.
- Acquérir des notions de langage d'assemblage. Utiliser un assembleur et un simulateur MIPS (SPIM).
- Comprendre comment des programmes complexes (conditions, boucles, fonctions/procédures) sont exprimés dans le langage du processeur.

Introduction

Programme stocké en mémoire

- A la différence d'une simple calculatrice, un ordinateur peut être adapté à un grand nombre de tâches : *du traitement de texte au jeu en réseau, en passant par la compilation et les applications scientifiques*.
- Derrière cette adaptabilité, se trouve un concept important de l'informatique: le fait qu'un ordinateur exécute un **programme stocké** en mémoire (*stored program*).
  - Il est plus facile de modifier un programme (*software*) que « re-câbler » le matériel (*hardware*).
  - En mémoire, un programme peut être manipulé comme de simples données.

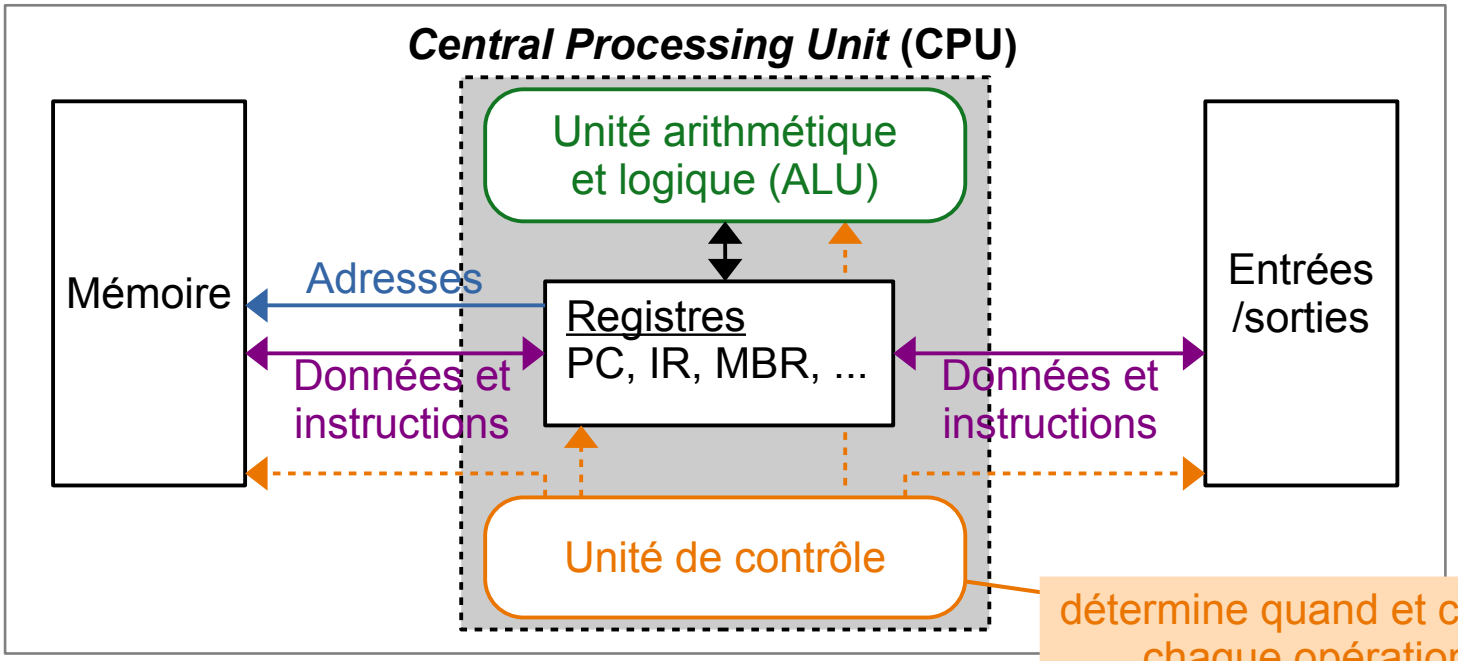


- Cette idée a priori simple et déjà formulée dans les années 1930 est encore au cœur de l'informatique d'aujourd'hui.

Introduction

Architecture de Von Neumann

- L'idée du *stored-program* est attribuée à John von Neumann, un mathématicien américain (d'origine hongroise), bien que d'autres pionniers de l'informatique comme Konrad Zuse et Alan Turing l'aient également proposée à la même époque.
- Cette idée a été mise en pratique dans l'ordinateur de l'*Institute for Advanced Study* (IAS) à Princeton dont l'architecture est souvent appelée **Architecture de Von Neumann**.



Introduction

Architecture (ISA)

- L'**architecture** (*instruction set architecture*) d'un ordinateur définit ce qu'il faut en savoir pour pouvoir le programmer.
- Elle comprend notamment
  - le **jeu d'instructions** (*instruction set*) : la définition de l'ensemble des instructions supportées.
  - sur quoi ces instructions portent : p.ex. **registres** et mémoire.
- Le **langage machine** est un encodage des instructions sous forme de mots binaires, compréhensibles par le processeur. Un programme en langage machine est une séquence de mots binaires.
- Il existe un grand nombre d'architectures différentes : x86 (IA-32 et x86_64), ARM, AVR, MIPS, RISC-V, PowerPC, … Chacune a un langage machine qui lui est propre.
- Il est impossible de couvrir toutes les architectures. Cependant, un grand nombre de concepts sont communs à celles-ci.

Introduction

Architecture MIPS

- Ce cours se base sur l'architecture **MIPS** - ***M**icroprocessor without **I**nterlocked **P**ipeline **S**tages*, mise au point dans les années 1980 à l'Université de Stanford, sous la direction de John Hennessy.
- Le **R2000** est le premier processeur MIPS commercialisé (1985). Entretemps, la société MIPS a commercialisé l'architecture MIPS à des fabricants de processeurs tels que p.ex. Microchip ou Broadcom.
- Raisons du choix de MIPS pour l'enseignement
  - Architecture réelle (p.ex. PS2 de Sony, routeurs Cisco, …)
  - Architecture propre et relativement simple
  - Architecture typique des processeurs modernes
  - Largement répandue dans l'enseignement
  - Simulateurs, assembleurs, compilateurs disponibles

- L'architecture MIPS a fortement inspiré l'architecture open-source et sans licence **RISC-V** récemment proposée par l'université de Berkeley. La société MIPS propose depuis 2023 son premier processeur RISC-V.

Architecture MIPS en bref

- Chemin de données
  - Largeur 32 bits
- Instructions
  - Taille fixe de 32 bits
  - Petit nombre d'instructions (architecture RISC)
  - Trois formats d'instructions (R, I et J)
- Registres internes
  - Chaque registre a une taille de 32 bits
  - Un compteur de programme `PC`
  - Une banque de 32 registres temporaires (`GPR`)
- Fonctionnement de type "*load and store*"
- Exécution de plusieurs cycles d'instruction en parallèle (*pipelining*)
- Co-processeurs pour des extensions (p.ex. gestion des exceptions)

Make the common case fast

Simplicity favors regularity

Smaller is faster

Documentation MIPS

- De nombreuses sources de documentation sont disponibles à propos de l'architecture MIPS et de son jeu d'instructions.
- Documentation **MIPS32® Architecture for Programmers** disponible sur le site web de la société MIPS, Incorporated (http://www.mips.com)
  - Volume I: Introduction to the MIPS32® Architecture
  - Volume II: The MIPS32® Instruction Set
  - Volume III: The MIPS32® Privileged Resource Architecture

- L'**annexe B** de l'ouvrage de référence (Patterson & Hennessy) donne une liste d'instructions MIPS et une courte spécification.

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

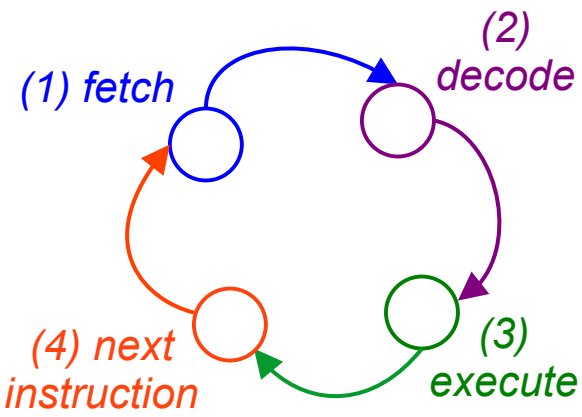
Registres PC et IR

Cycle d'Instruction

- Un programme en langage machine est une séquence d'**instructions** élémentaires, chacune encodée sous forme d'un mot binaire.

- L'exécution d'un programme par le processeur consiste en un **cycle d'instruction** sans fin dans lequel le processeur effectue les opérations suivantes :

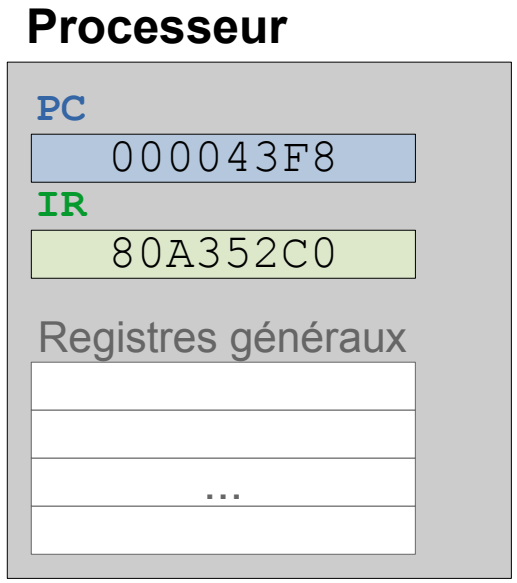
  1) récupère une instruction en mémoire (*fetch*)
  2) décode l'instruction
  3) exécute l'instruction
  4) passe à l'instruction suivante



Registres PC et IR

Instruction Fetch

- Deux registres spéciaux de 32-bits sont utilisés lors de l'I*nstruction Fetch* :
  - **Compteur de Programme** (*Program Counter,* `PC`) contient l'adresse du 1er octet de la prochaine instruction.
  - **Registre d'Instruction** (*Instruction Register,* `IR`) contient la dernière instruction chargée, en vue d'être décodée et exécutée.



- Passer à l'instruction suivante consiste à incrémenter le registre `PC` de la taille d'une instruction, soit 4 octets dans le cas de MIPS, ce que nous noterons

  `PC` ← `PC` + 4

- Le nombre max. d'instructions adressables est contraint par la taille du registre `PC`. Dans le cas de MIPS, il est possible d'adresser 2^{30} instructions différentes.

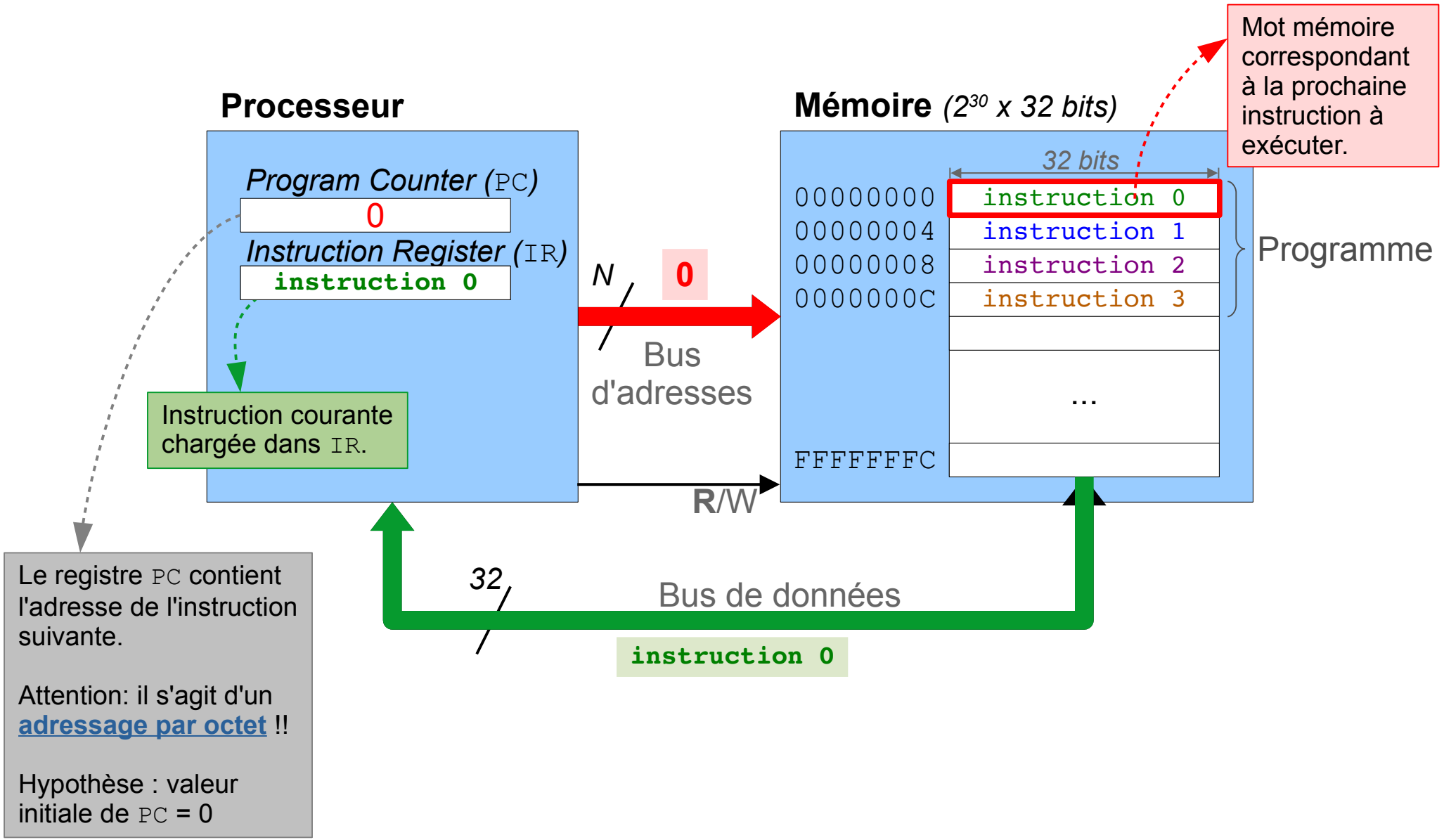
(1) Parfois aussi appelé Pointeur d'Instruction (*Instruction Pointer* – `IP`), p.ex. dans les architectures Intel (IA-32 et Intel 64).

Registres PC et IR

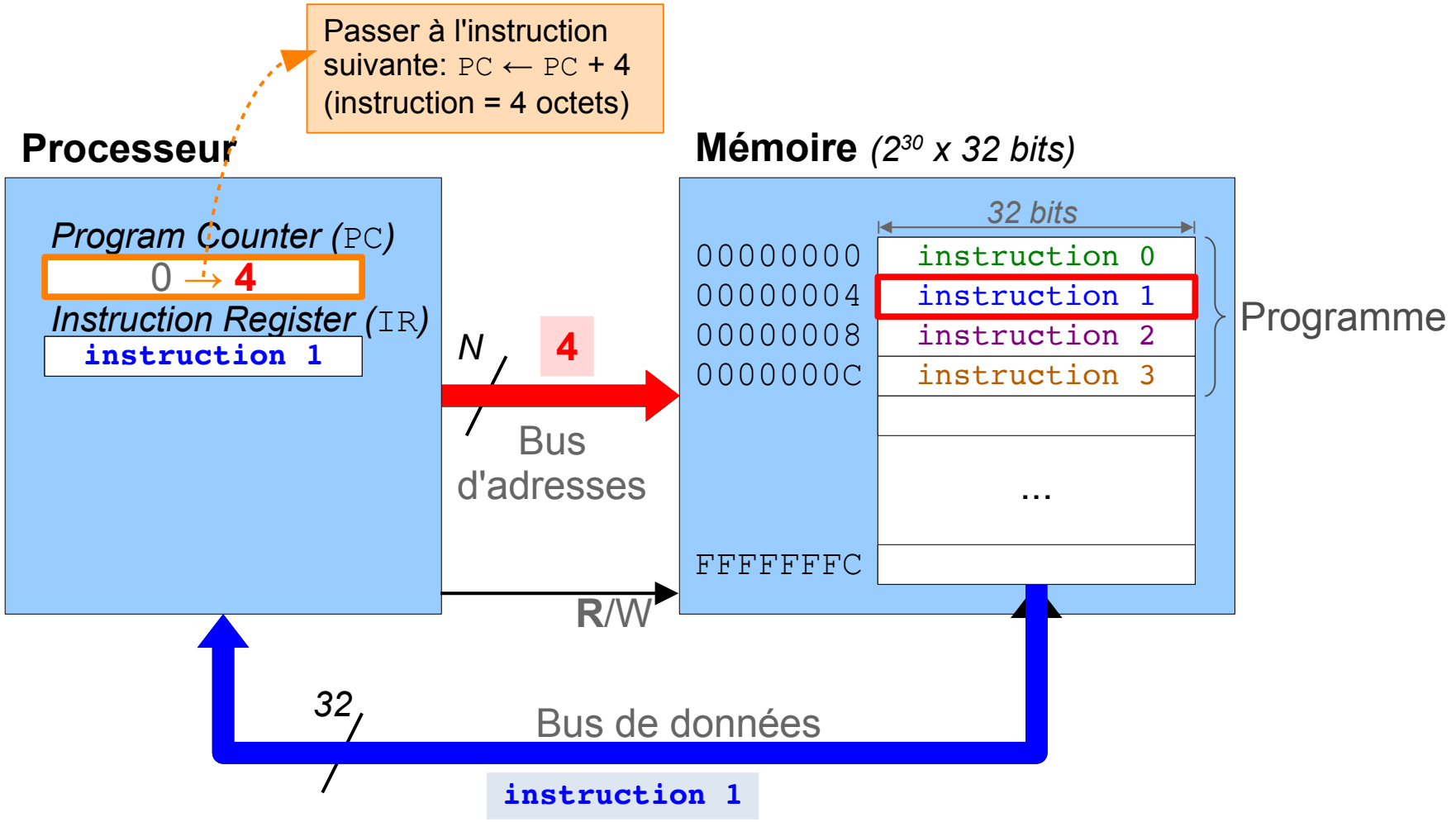
Convention

- Les conventions suivantes sont utilisées pour le registre `PC`
  - Le processeur utilise un **adressage par octet**. Le registre `PC`. contient l'adresse du premier octet (sur 4) d'une instruction.
  - Le registre `PC` contient un multiple de 4, i.e. une **adresse alignée** sur la taille d'une instruction
  - Le contenu du registre `PC` est toujours celui qui **suit** l'*instruction fetch*, i.e. l'adresse de l'instruction suivante
  - Les adresses sont notées en **hexadécimal**.

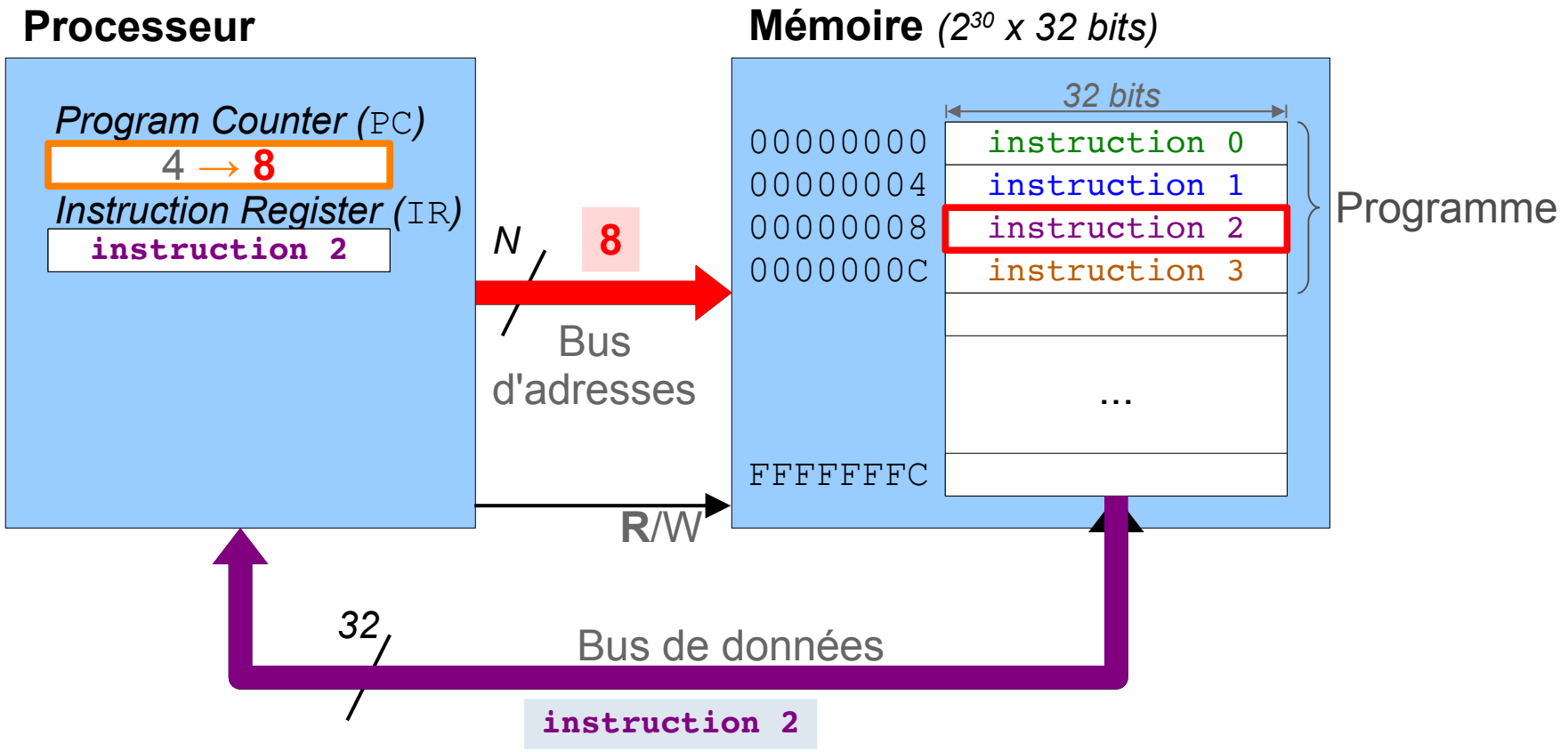
Registres PC et IR



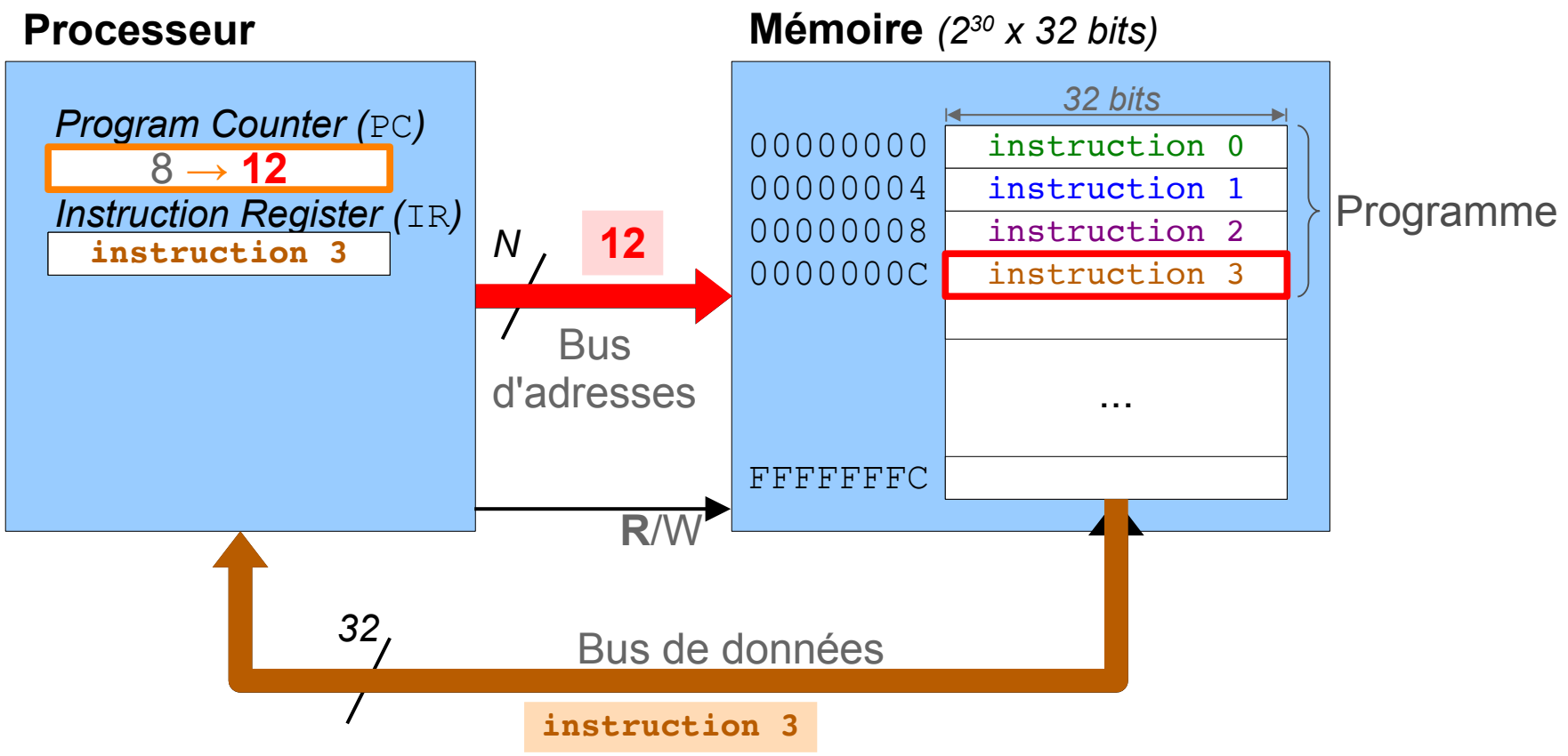
Registres PC et IR



Registres PC et IR



Registres PC et IR



Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

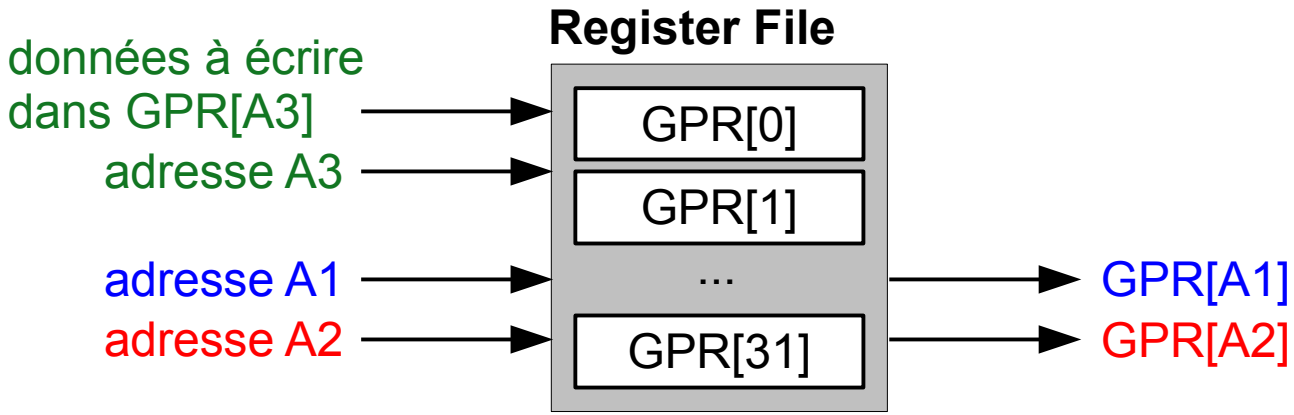
- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Registres GPR

Registres généraux

- Dans un processeur MIPS, toutes les opérations arithmétiques et logiques sont réalisées sur des **registres temporaires** aussi appelés registres d'usage général (*general-purpose registers* – GPR)[(1)]. Les registres spéciaux tels que `PC` ou `IR` ne peuvent être manipulés par ces instructions.
- Les registres GPR sont regroupés dans une **banque de registre** (register file)[(2)]. Les registres lus et/ou écrits sont spécifiés avec leurs **adresses** (ou index).



(1) ou « *scratchpad registers* »
(2) parfois appelée « *scratchpad memory* »

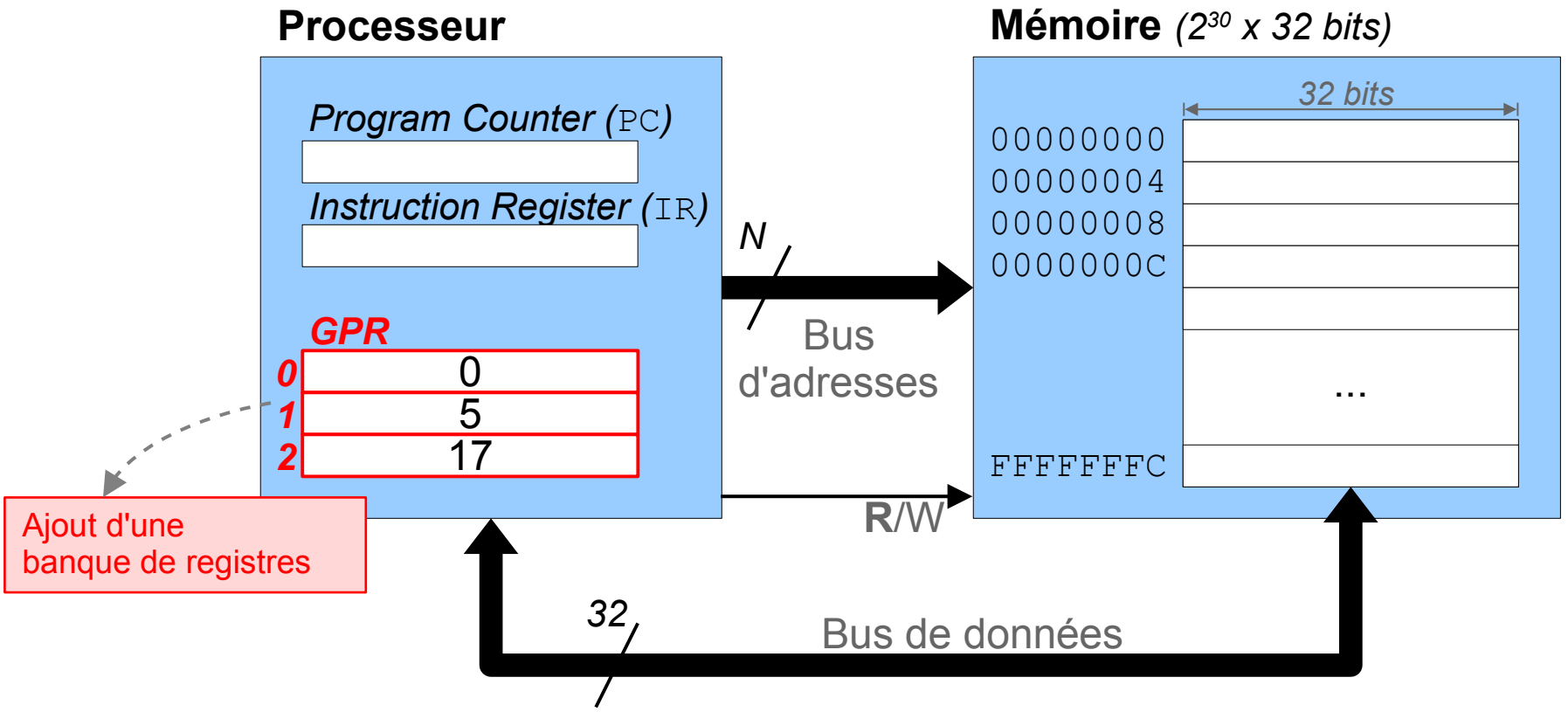
Registres GPR

Registres généraux

- Parmi les 32 registres GPR, certains registres sont réservés pour des usages spécifiques tels que le pointeur de pile ou l'adresse de retour lors d'un appel de fonction.
- L'usage des registres peut aussi être soumis à des conventions, p.ex. registres contenant les arguments de fonctions.

Nom	Index	Description
zero	0	constante 0
at	1	reservé pour l'assembleur
v0,v1	2,3	évaluation d'expression / retour de fonction
a0-a3	4-7	arguments de fonction
t0-t7	8-15	temporaire (non préservé à l'appel)
s0-s7	16-23	temporaire (préservé à l'appel)
t8,t9	24,25	temporaire (non préservé à l'appel)
k0,k1	26,27	réservé pour le noyau de l'O.S.
gp	28	pointeur vers espace global
sp	29	pointeur de pile (*stack pointer*)
fp	30	pointeur de cadre de pile (*frame pointer*)
ra	31	adresse de retour de fonction

Registres GPR



Note : par convention, GPR**[0]** (nommé `zero`) contient toujours la valeur 0.

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Langage d'assemblage MIPS

- Le langage machine utilise des mots binaires pour encoder les instructions. C'est peu pratique pour un humain :
  - il est nécessaire de connaître l'encodage de chaque instruction
  - la programmation est fastidieuse, lente, sujette aux erreurs

- Le **langage d'assemblage** (*assembly langage*) utilise une représentation lisible par les humains, sous forme de texte.
  - Chaque instruction est désignée par un **mnémonique** (*mnemonic*), un symbole facile à mémoriser (souvent une suite de caractères).
  - Les opérandes d'une l'instruction sont spécifiées à la suite du mnémonique, dans un ordre et selon une syntaxe propres au langage d'assemblage.

- Un programme spécial, l'**assembleur** effectue la conversion de programmes exprimés en langage d'assemblage vers le langage machine.

Exemple

- Supposons que l'on souhaite effectuer l'addition des registres `a0` et `a1` et placer le résultat dans le registre `v0`. En langage d'assemblage MIPS, il est possible d'utiliser l'instruction ADD.

```
add   $v0, $a0, $a1
      destination  sources
```

  - « **add** » est le mnémnonique de l'instruction
  - `a0`, `a1` et `v0` sont les noms des registres, précédés du caractère `$`
  - après le mnémonique vient la destination, puis les sources (syntaxe « Intel »)

- De la même manière, le mnémonique « **sub** » désigne la soustraction.

```
sub   $v0, $v0, $a2
```

Langage d'assemblage

Assembleur

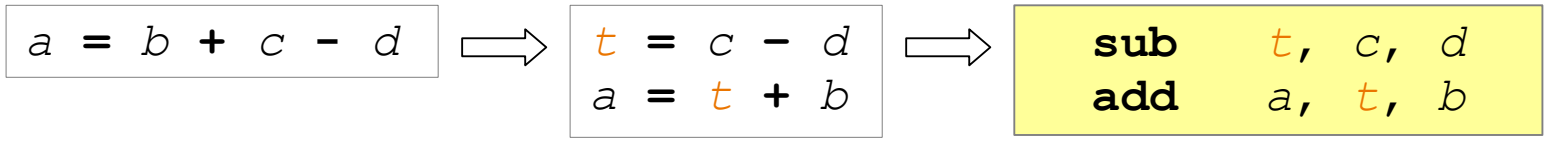
- L'assembleur est un programme qui traduit les instructions exprimées en langage d'assemblage vers des mots binaires du langage machine.

Langage d'assemblage	**Assembleur**	*Langage machine*
`add $v0, $a0, $a1`	⇒	**0x00851020** 00000000100001010001000000100000
`sub $v0, $v0, $a2`	⇒	**0x00461022** 00000000010001100001000000100010

Opérations plus complexes

- Afin de maintenir le jeu d'instructions court, seules les opérations les plus courantes sont fournies sous forme d'instructions. C'est un choix de conception caractéristique des architectures RISC (« *Make the common case fast* »).

- Des opérations plus complexes telles qu'on peut les exprimer dans un langage de haut-niveau tel que C, Java ou Python doivent être découpées en opérations élémentaires.



- Il est possible, par exemple
  - réaliser la soustraction de *c* et *d*; le résultat étant stocké temporairement dans un registre (*t*)
  - effectuer l'addition entre *t* et *b* et placer le résultat dans *a*.
- En faisant l'hypothèse que *a*, *b*, *c*, *d* et *t* sont des registres MIPS, il alors possible de traduire le tout en instructions.

Exercice

- Comment traduire la suite d'expressions suivante en langage d'assemblage MIPS ?

```
a = b - c
d = (a + e) - (f + g)
```

  - Hypothèses : les variables b et c sont contenues dans les registres `a0` et `a1`, la variable d dans le registre `v0` et les variables e à g dans les registres `s1` à `s3`. Les variables temporaires peuvent être stockées dans les registres `t0` à `t7`.

Directives d'assemblage

- En plus des instructions, le langage d'assemblage supporte plusieurs directives.

Directive	Description
`.data` [*addr*]	place les données/instructions suivantes dans le segment de données.
`.text` [*addr*]	place les données/instructions suivantes dans le segment de code
`.byte` $b_1, \ldots, b_n$	définit des octets
`.half` $h_1, \ldots, h_n$	définit des mots de 16 bits
`.word` $w_1, \ldots, w_n$	définit des mots de 32 bits
`.ascii` *str*	définit une chaîne de caractères ASCII
`.asciiz` *str*	définit une chaîne de caractères ASCII à zéro terminal
`.align` *n*	aligne la donnée suivante sur 2^n

Etiquettes (*labels*)

- Les étiquettes sont des moyens d'identifier la position d'une instruction ou d'une donnée en mémoire, sans devoir en connaître l'adresse exacte.
- Une étiquette est un identifiant suivi du symbole « : », généralement placé en début de ligne.

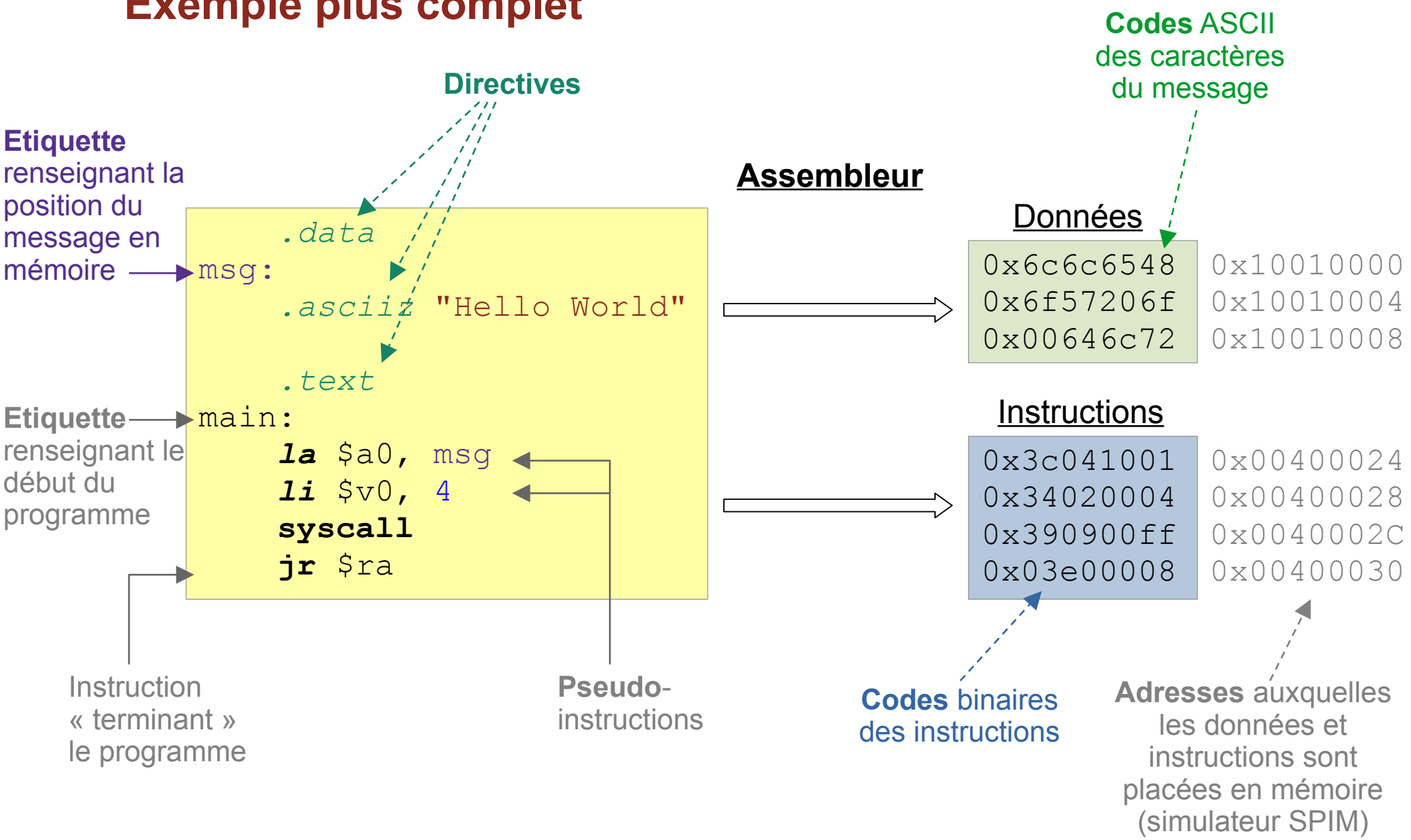
Pseudo-instructions

- En plus des instructions MIPS, le langage d'assemblage permet l'utilisation de pseudo-instructions afin de rendre certaines opérations plus faciles à exprimer. Ces pseudo-instructions sont traduites par l'assembleur en instructions réelles.

Pseudo-instruction	Description
`li` *reg*, *imm*	Charge la constante *imm* dans le registre *reg*
`la` *reg*, *addr*	Charge l'adresse *addr* dans le registre *reg*

Langage d'assemblage

Exemple plus complet



Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

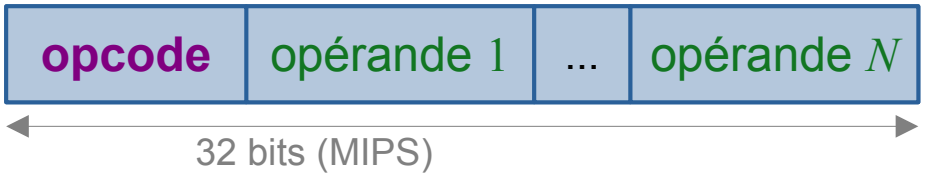
- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Jeu d'instructions

Instruction

- Une **instruction** désigne de façon unique un ensemble d'opérations à réaliser par le processeur. L'ensemble des instructions d'un processeur est appelé le **jeu d'instructions** (*instruction set*).
- En **langage machine**, chaque instruction est représentée par un mot binaire, compréhensible par le processeur.

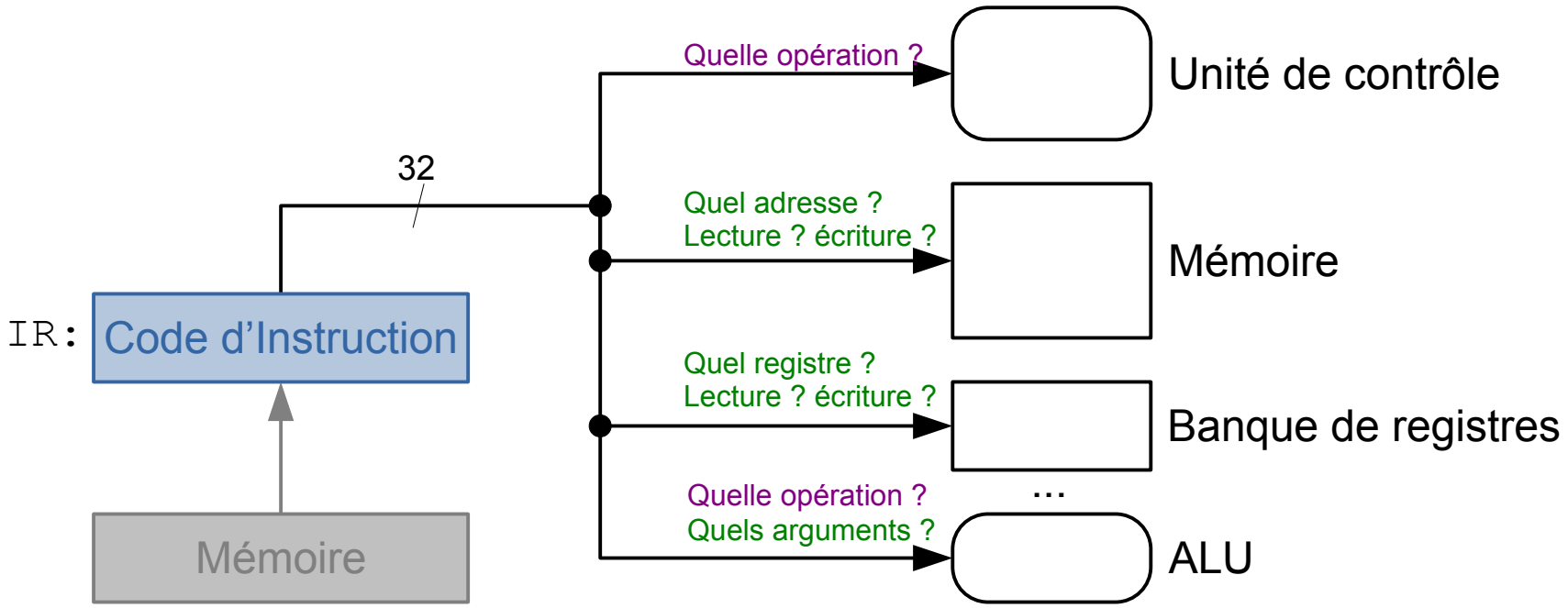


- L'encodage d'une instruction comprend deux parties distinctes :
  - **code d'opération** (*opcode*) : indique quelle opération cette instruction désigne. L'opcode est unique pour chaque instruction.
  - **opérandes** (*operands*) : fournissent des paramètres tels que les registres sources et destination d'une opération. Une instruction peut avoir de 0 à plusieurs opérandes.

Jeu d'instructions

Décodage / exécution d'instruction

- Le code d'instruction est utilisé à l'intérieur du processeur
  - par l'unité de contrôle du processeur et éventuellement par l'ALU pour indiquer **quelle opération doit être effectuée**.
  - par différents éléments du processeur (ALU, banque de registres) et par la mémoire pour indiquer **sur quoi l'opération doit être effectuée**.



Jeu d'instructions

Catégories d'instructions

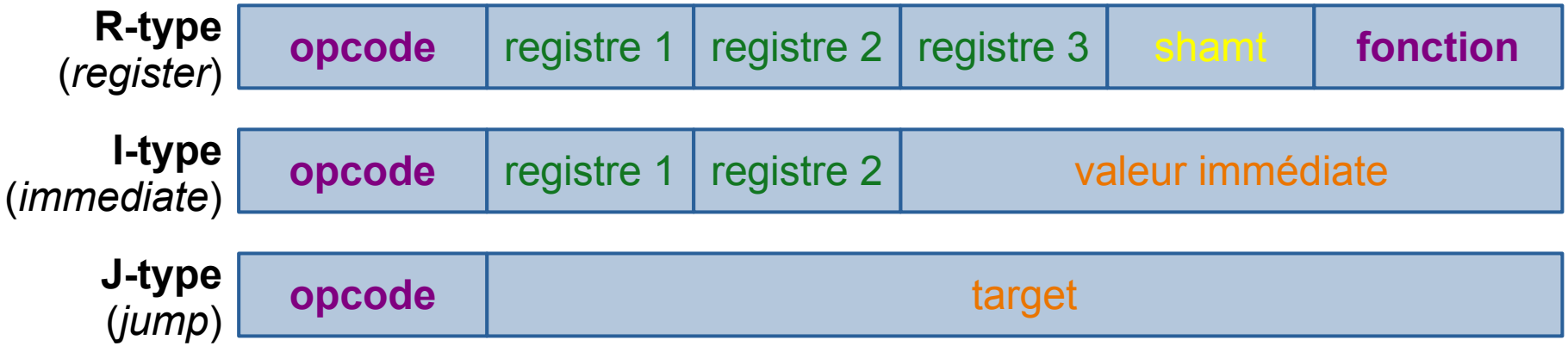
- Le jeu d'instructions MIPS est structuré en plusieurs catégories d'instructions
  - les instructions logiques et arithmétiques
  - les instructions d'accès à la mémoire
  - les instructions de contrôle de flux
  - les instructions d'appel de procédures/fonctions
  - les instructions spéciales

Jeu d'instructions

Formats d'instructions

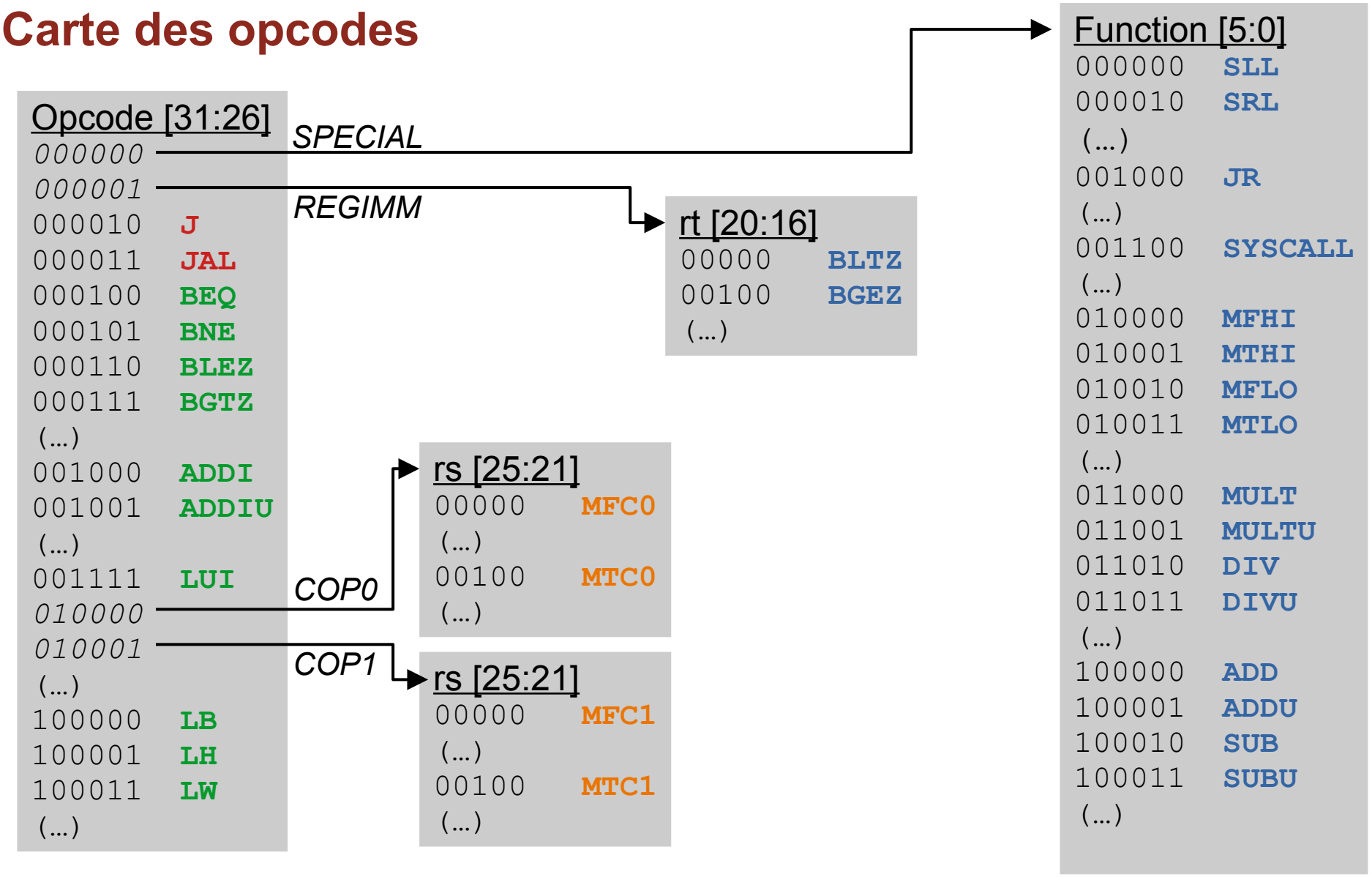
- Toutes les instructions MIPS ont une **taille fixe de 32 bits** qui correspond à un mot mémoire.
  - Avantage : simplicité de l'architecture.
  - Certaines instructions n'ont peut-être pas besoin des 32 bits pour encoder leurs opérandes, dans ce cas une partie de l'instruction n'est pas utilisée.

- Trois formats différents sont utilisés pour la plupart des instructions :



Jeu d'instructions

Carte des opcodes



R-type I-type J-type autres

Basé sur : ***MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set***, Revision 3.02

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Opérations arithmétiques et logiques

Types d'opérations

- Dans cette catégorie d'instructions, on retrouve
  - addition, soustraction, multiplication et division de nombres entiers signés ou non-signés
  - décalages à gauche et à droite (arithmétique et logique), rotations
  - opérations logiques « bit-à-bit » : et, ou, ou-exclusif
  - des tests de condition : comparaison de nombres entiers

- Toutes ces opérations sont réalisées sur des registres.
- Deux formats d'instructions sont utilisés
  - **Type R** (*register*) pour les opérations de la forme *r3* ← *r1* OP *r2*

  - **Type I** (*immediate*) pour les opérations de la forme *r2* ← *r1* OP *const*

Opérations arithmétiques et logiques

Instruction *Add Word* (`ADD`)

- L'instruction d'addition (`ADD`) effectue l'addition de deux registres source (`rs` et `rt`) et stocke son résultat dans un troisième registre destination (`rd`).
- Elle est spécifiée comme suit

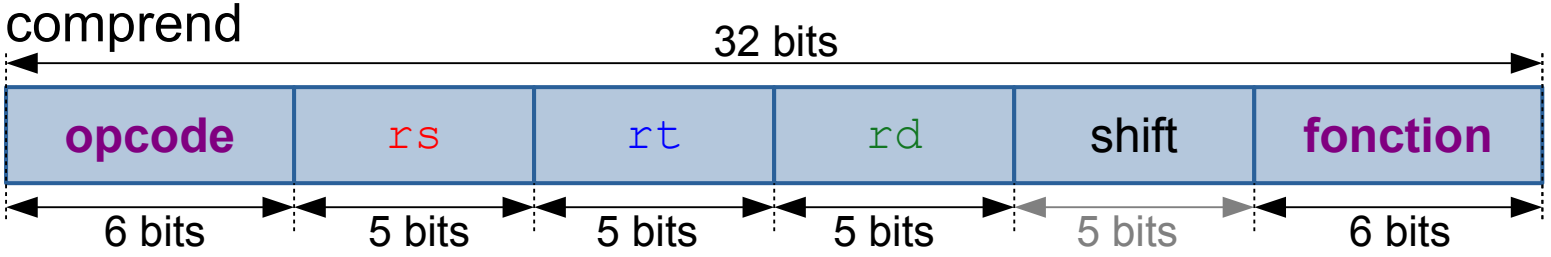
```
GPR[rd] ← GPR[rs] + GPR[rt]
```

  - placer dans le registre GPR d'adresse `rd` le résultat de l'addition des registres GPR d'adresses `rs` et `rt`.

Opérations arithmétiques et logiques

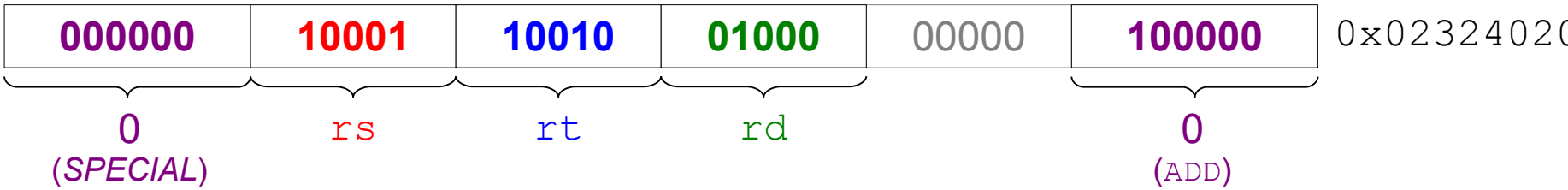
Instruction *Add Word* (ADD)

- L'instruction `ADD` est encodée au format de **type R**. Le mot binaire comprend

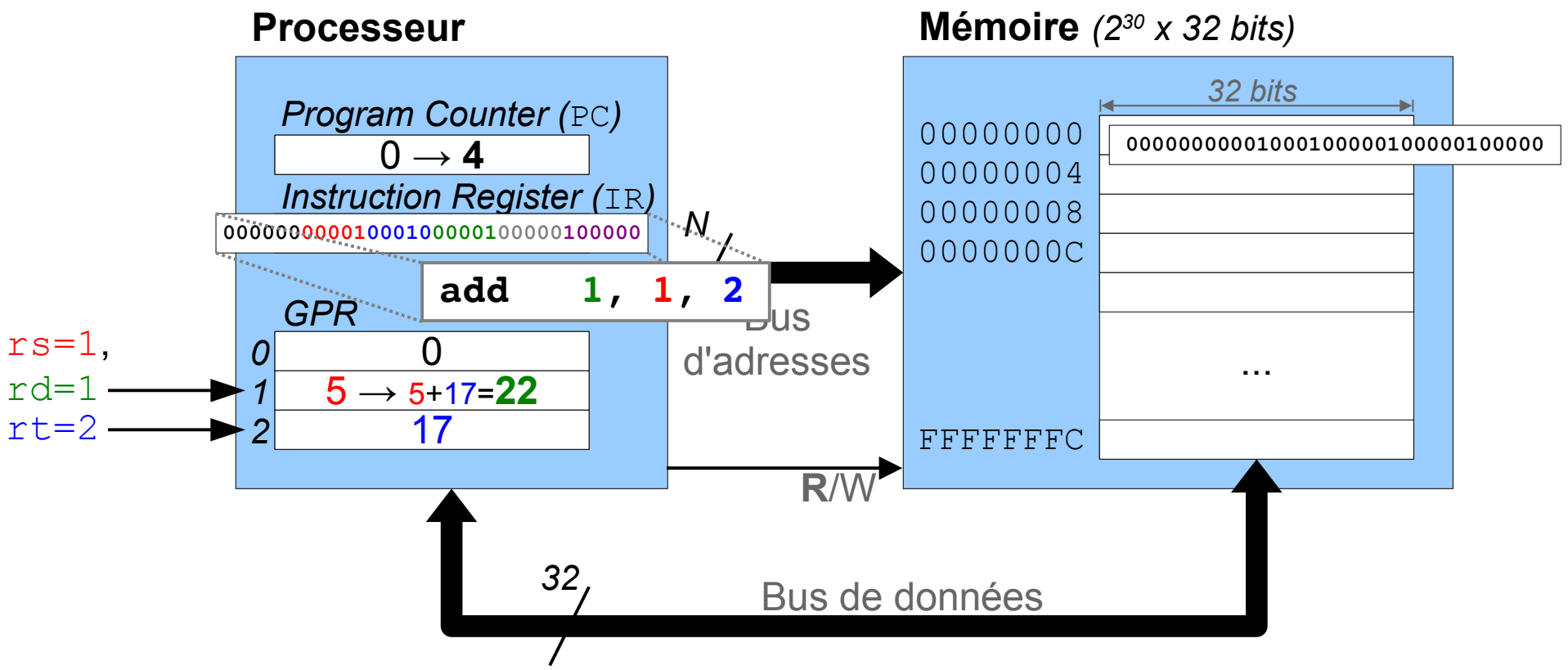


  - l'opcode, encodé sur les 6 bits de poids fort, vaut 0 (*SPECIAL*) pour une opération arithmétique
  - les index des registres `rs`, `rt` et `rd`, encodés chacun sur 3 groupes de 5 bits suivants (il y a $32=2^5$ registres différents).
  - un code de fonction arithmétique, encodé sur les 6 bits de poids faible, vaut 32 pour `ADD`

- Exemple : **`add`** `$t0, $s1,`



Opérations arithmétiques et logiques



Add

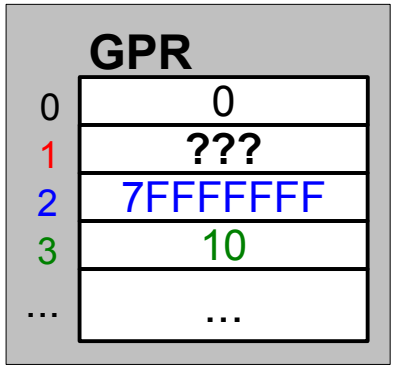
syntaxe: **add** rd, rs, rt

description: GPR[rd] ← GPR[rs] + GPR[rt]

Instruction *Add Unsigned Word* (`ADDU`)

- L'instruction `ADD` effectue une addition signée. En cas de dépassement (*underflow* or *overflow*), une situation d'erreur (exception) est générée.
- Il existe une variante de `ADD`, appelée `ADDU` qui effectue une addition non-signée modulo 2^{32}. `ADDU` ne génère pas d'exception en cas de dépassement.

- Exemple :
  - supposons le contenu suivant des registres GPR




```
add   1, 2, 3
```

⇨ génère une exception
ne modifie pas GPR[1]

```
addu   1, 2, 3
```

⇨ ne génère pas d'exception
GPR[1] ← 0x7FFFFFFF + 10 = 0x80000009

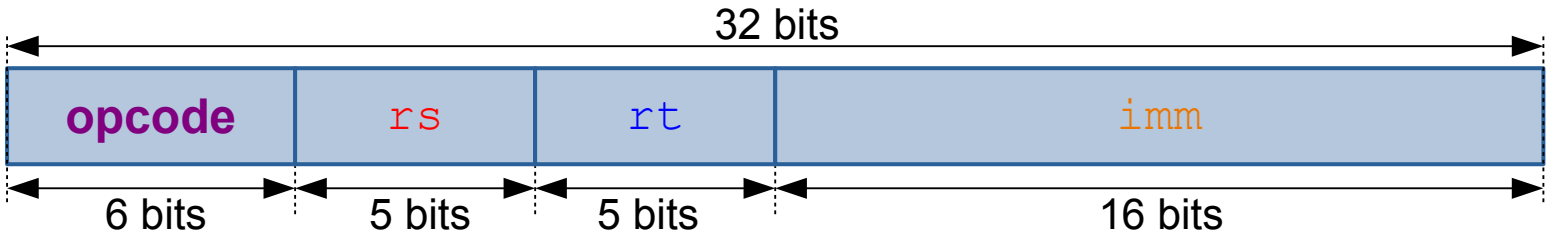
Opérations arithmétiques et logiques

Instruction *Add Word Immediate* (`ADDI`)

- L'instruction `ADDI` effectue l'addition d'un registre et d'une valeur constante (valeur immédiate).

```
GPR[rt] ← GPR[rs] + s-ext(imm)
```

- `ADDI` est encodée au format de **type I**. Le mot binaire comprend



  - l'opcode, encodé sur les 6 bits de poids fort, vaut 8 (`ADDI`).
  - les index des registres `rs` et `rt`.
  - une valeur immédiate `imm` (constante) encodée sur 16 bits.

- Exemple : **`addi`** `$t0, $s1, -9`

Opérations arithmétiques et logiques

Instructions de type R

- La liste ci-dessous reprend quelques instructions de type R.

Le processeur détermine l'instruction sur base du couple (*opcode*, *fonction*). Celui-ci est différent pour chaque instruction.

sa = *shift amount* (quantité de décalage)

Instruction	Description	OpCode	Fonction
No Operation			
NOP		000000	000000
Shift Left Logical			
SLL rd, rt, sa	GPR[rd] ← GPR[rt] << sa (logique)	000000	000000
Shift Right Logical			
SRL rd, rt, sa	GPR[rd] ← GPR[rt] >> sa (logique)	000000	000010
Shift Right Arithmetic			
SRA rd, rt, sa	GPR[rd] ← GPR[rt] >> sa (arithmétique)	000000	000011
Shift Left Logical Variable			
SLLV rd, rt, rs	GPR[rd] ← GPR[rt] << GPR[rs] (log.)	000000	000100
Shift Right Logical Variable			
SRLV rd, rt, rs	GPR[rd] ← GPR[rt] >> GPR[rs] (log.)	000000	000110
Shift Right Arithmetic Variable			
SRAV rd, rt, rs	GPR[rd] ← GPR[rt] >> GPR[rs] (arithm.)	000000	000111

Note : **NOP** est en réalité **SLL** avec un décalage (*shift amount*) de 0

Opérations arithmétiques et logiques

Instructions de type R (suite)

- La liste ci-dessous reprend quelques instructions de type R.

Instruction	Description	OpCode	Fonction
Multiply			
MULT rs, rt	(HI, LO) ← GPR[rs] * GPR[rt]	000000	011000
Multiply Unsigned			
MULTU rs, rt	(HI, LO) ← GPR[rs] * GPR[rt] (non signé)	000000	011001
Divide			
DIV rs, rt	(HI, LO) ← GPR[rs] / GPR[rt]	000000	011010
Divide Unsigned			
DIVU rs, rt	(HI, LO) ← GPR[rs] / GPR[rt] (non signé)	000000	011011
Add			
ADD rd, rs, rt	GPR[rd] ← GPR[rs] + GPR[rt] Exc	000000	100000
Add Unsigned			
ADDU rd, rs, rt	GPR[rd] ← GPR[rs] + GPR[rt] (non signé)	000000	100001
Subtract			
SUB rd, rs, rt	GPR[rd] ← GPR[rs] - GPR[rt] Exc	000000	100010
Subtract Unsigned			
SUBU rd, rs, rt	GPR[rd] ← GPR[rs] - GPR[rt] (non signé)	000000	100011

Opérations arithmétiques et logiques

Instructions de type R (suite)

- La liste ci-dessous reprend quelques instructions de type R.

Instruction	Description	OpCode	Fonction
And			
AND rd, rs, rt	GPR[rd] ← GPR[rs] AND GPR[rt]	000000	100100
Or			
OR rd, rs, rt	GPR[rd] ← GPR[rs] OR GPR[rt]	000000	100101
Exclusive Or			
XOR rd, rs, rt	GPR[rd] ← GPR[rs] XOR GPR[rt]	000000	100110
Not Or			
NOR rd, rs, rt	GPR[rd] ← GPR[rs] NOR GPR[rt]	000000	100111
Set on Less Than			
SLT rd, rs, rt	GPR[rd] ← GPR[rs] < GPR[rt]	000000	101010
Set on Less Than Unsigned			
SLTU rd, rs, rt	GPR[rd] ← GPR[rs] < GPR[rt] (non signé)	000000	101011

Opérations arithmétiques et logiques

Instructions de type R (suite)

- La liste ci-dessous reprend quelques instructions de type R.

Instruction	Description	OpCode	Fonction
Move From HI Register			
MFHI rd	GPR[rd] ← HI	000000	010000
Move To Hi Register			
MTHI rs	HI ← GPR[rs]	000000	010001
Move From LO Register			
MFLO rd	GPR[rd] ← LO	000000	010010
Move To LO Register			
MTLO rs	LO ← GPR[rs]	000000	010011

Opérations arithmétiques et logiques

Instructions type I

- La liste ci-dessous reprend quelques instructions de type I.

Instruction	Description	OpCode
And Immediate		
ANDI rt, rs, imm	GPR[rt] ← GPR[rs] AND *0-ext*(imm)	001100
Or Immediate		
ORI rt, rs, imm	GPR[rt] ← GPR[rs] OR *0-ext*(imm)	001101
Exclusive Or Immediate		
XORI rt, rs, imm	GPR[rt] ← GPR[rs] XOR *0-ext*(imm)	001110
Set on Less Than Immediate		
SLTI rt, rs, imm	GPR[rt] ← GPR[rs] < *s-ext*(imm)	001010
Set on Less Than Immediate Unsigned		
SLTIU rt, rs, imm	GPR[rt] ← GPR[rs] < *s-ext*(imm) (non signé)	001011

Note :
0-ext(imm) signifie que la valeur immédiate sur 16 bits est étendue à 32 bits en ajoutant 16 bits de poids fort à 0.
s-ext(imm) signifie qu'une extension de signe est effectuée (cf. chapitre 2).

Opérations arithmétiques et logiques

Instructions type I (suite)

- La liste ci-dessous reprend quelques instructions de type I.

Instruction	Description	OpCode
Add Immediate		
ADDI rt, rs, imm	GPR[rt] ← GPR[rs] + *s-ext*(imm)	001000
Add Immediate Unsigned		
ADDIU rt, rs, imm	GPR[rt] ← GPR[rs] + *s-ext*(imm)	001101
Load Upper Immediate		
LUI rt, imm	GPR[rt] ← imm || $0_{15..0}$	001111

Note :
`ADDIU` effectue une addition modulo 2^{32}. Par conséquent, si le résultat de l'addition provoque un dépassement (*overflow*), aucune erreur n'est générée. Il s'agit cependant bien d'une addition “signée” contrairement à ce qu'indique le nom de l'instruction.

Opérations arithmétiques et logiques

Opérations logiques « bit à bit » (*bitwise logic op*)

- Exemple à construire

```
     0100 0100 1110 1010 0101 0010 0101 0111
     0101 0010 0101 0010 0100 1010 0100 0010
AND  0100 0000 0100 0010 0100 0010 0100 0010

     0100 0100 1110 1010 0101 0010 0101 0111
     0101 0010 0101 0010 0100 1010 0100 0010
OR   0101 0110 1111 1010 0101 1010 0101 0111

     0100 0100 1110 1010 0101 0010 0101 0111
     0101 0010 0101 0010 0100 1010 0100 0010
XOR  0001 0110 1011 1000 0001 1000 0001 0101
```

Opérations arithmétiques et logiques

Instruction *Load Uppper Immediate* (`LUI`)

- Les pseudo-instructions `LI` et `LA` qui permettent respectivement de charger une constante ou une adresse dans un registre sont traduites en instructions réelles par l'assembleur.
- Avec MIPS, il est impossible de spécifier une valeur immédiate de 32 bits dans une instruction (*pourquoi ?*). Il faut donc parfois effectuer le chargement d'une constante en 2 instructions.
- L'instruction `LUI` permet notamment de charger un mot de 16 bits dans la partie haute d'un registre. Elle est spécifiée comme suit :

```
GPR[rt] ← imm || 0^16
```

- Exemple :

Assembleur

const. < 2^{16}

```
li $t0, 0x3A54
```

⇒

```
ori $t0, $zero, 0x3A54
```

$t0:	?
	0x00003A54

const. ≥ 2^{16}

```
li $s3, 0xFEDCBA98
```

⇒

```
lui $s3, 0xFEDC
ori $s3, $s3, 0xBA98
```

$s3:	?
	0xFEDC0000
	0xFEDCBA98

Opérations arithmétiques et logiques

Exemple

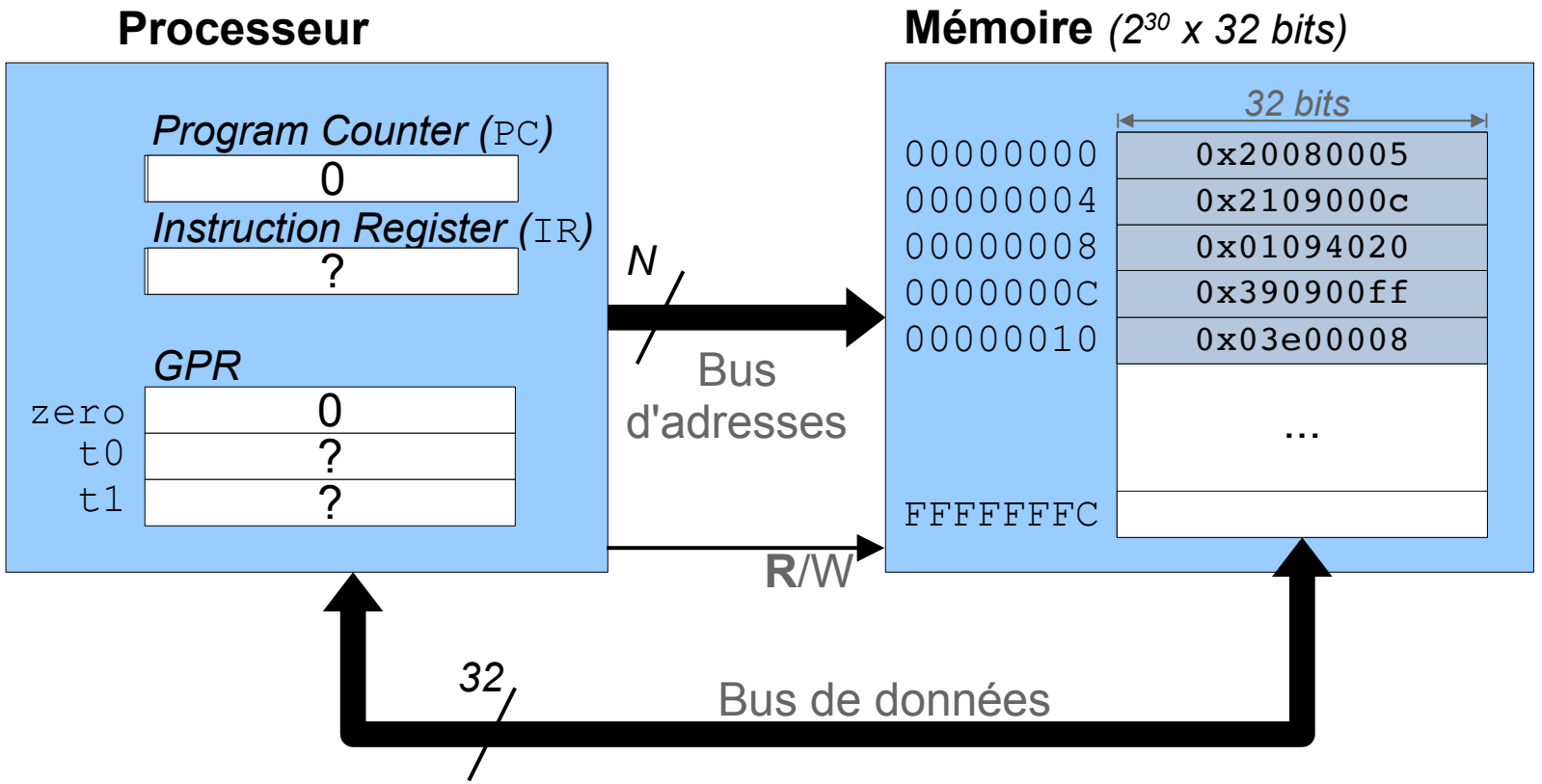
- Exécution pas-à-pas du programme suivant

```
main:
   addi   $t0, $zero, 5
   addi   $t1, $t0, 12
   add    $t0, $t0, $t1
   xori   $t1, $t0, 255
   jr     $ra
```

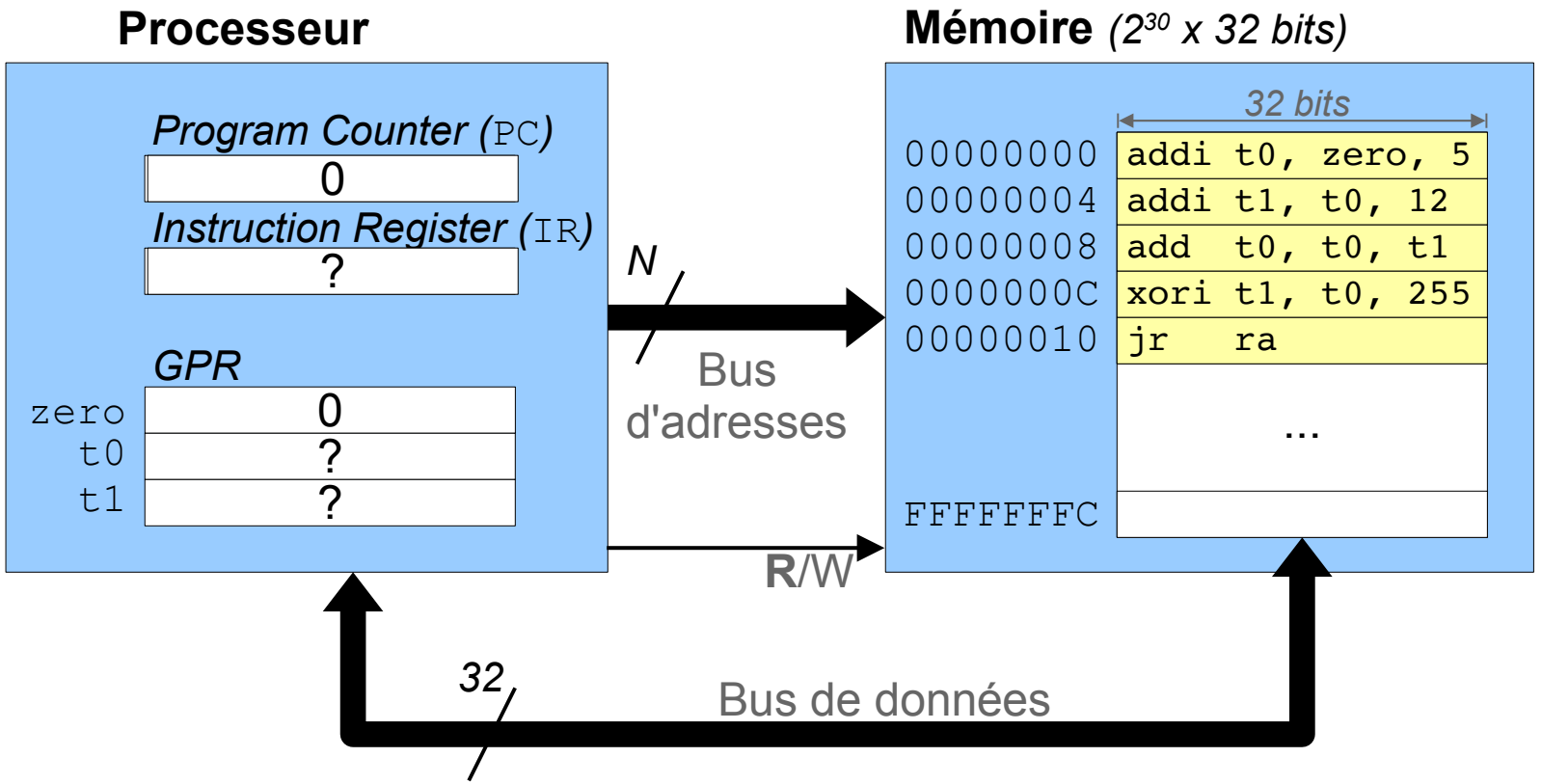
Assembleur


```
0x20080005
0x2109000c
0x01094020
0x390900ff
0x03e00008
```

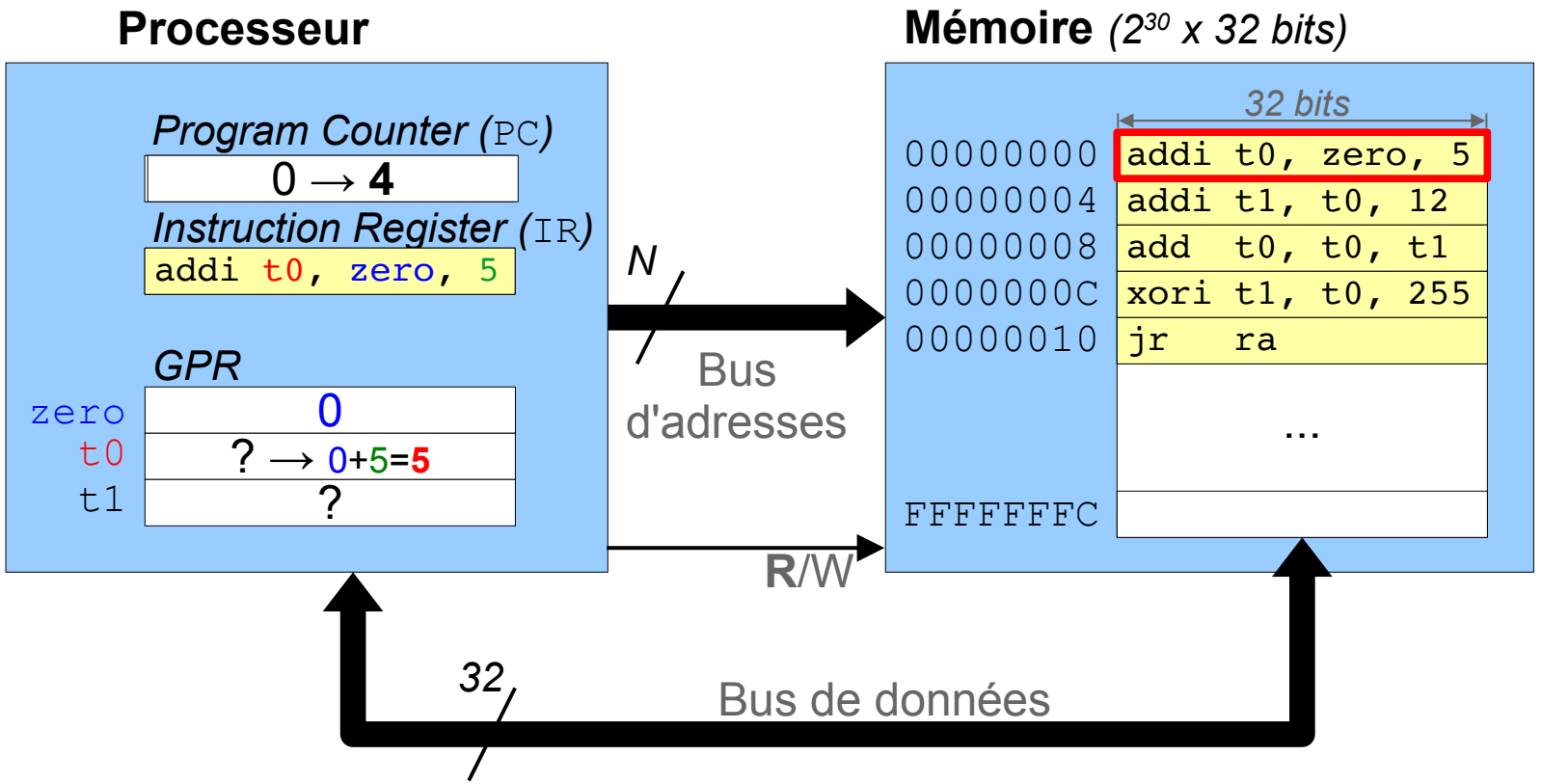
Opérations arithmétiques et logiques



Opérations arithmétiques et logiques



Opérations arithmétiques et logiques

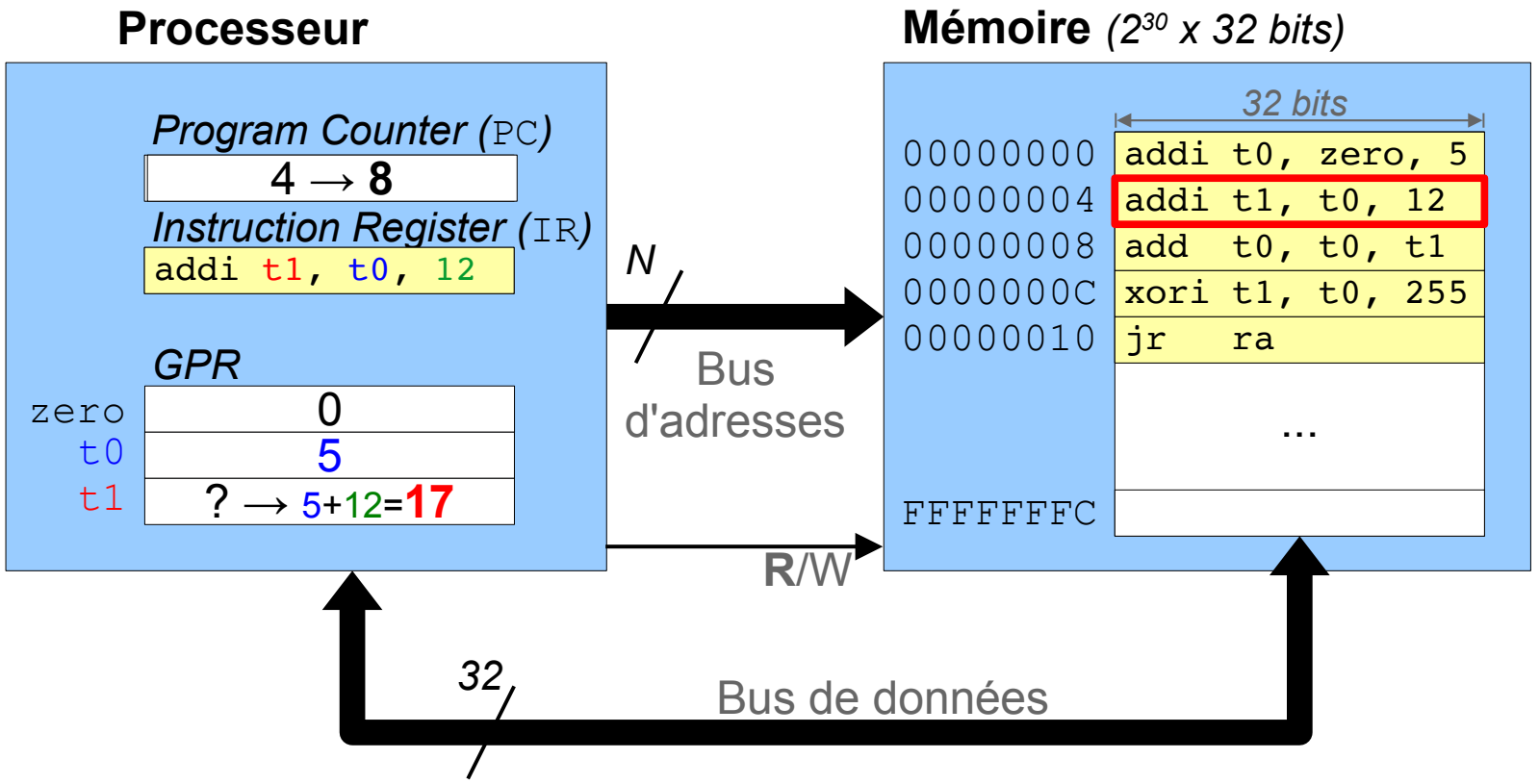


Add Immediate

syntaxe: **addi** rd, rs, imm

description: GPR**[**rd**]** ← GPR[rs] + s-ext(imm)

Opérations arithmétiques et logiques

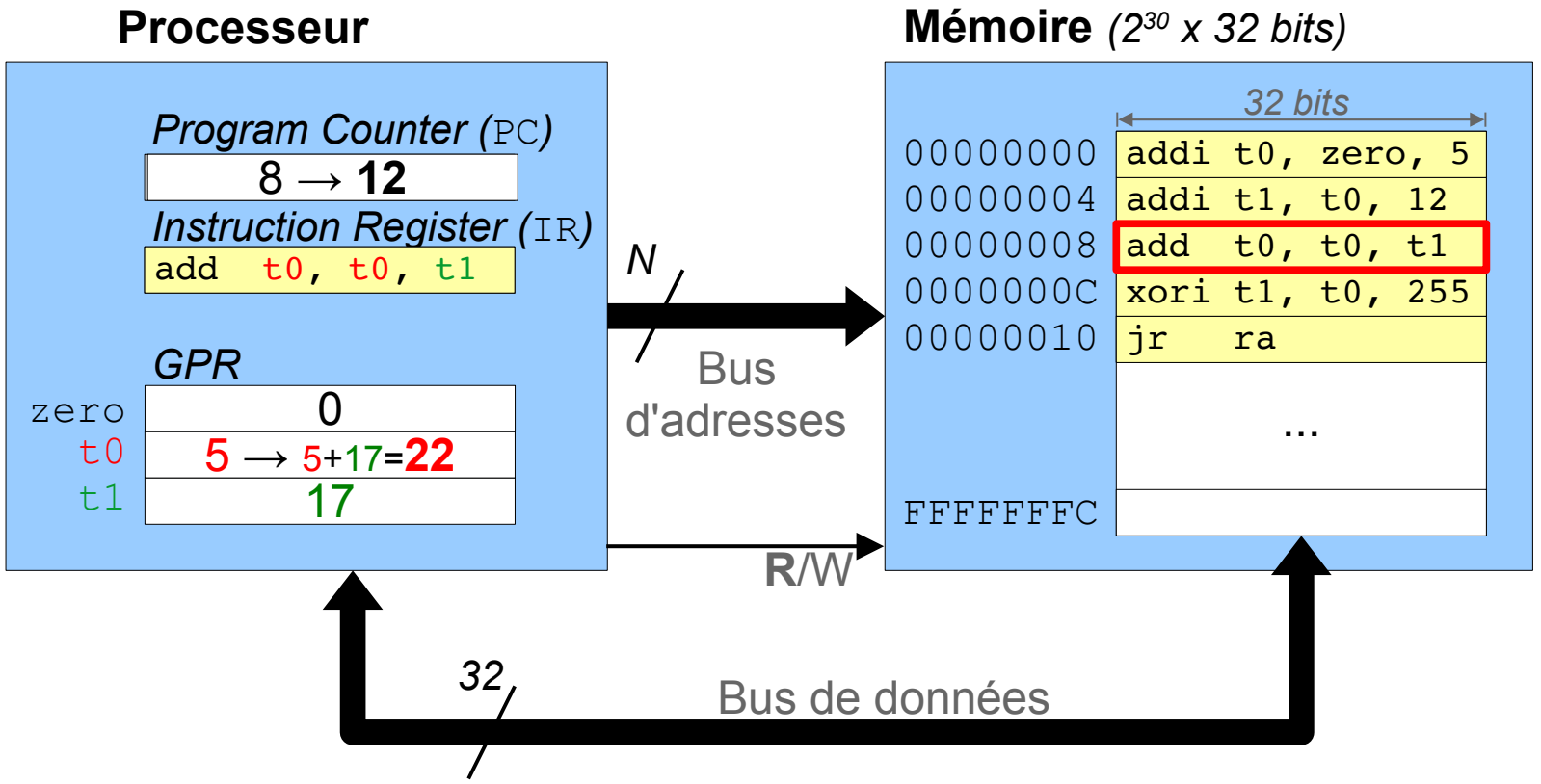


Add Immediate

syntaxe: **addi** rd, rs, imm

description: GPR[rd] ← GPR[rs] + s-ext(imm)

Opérations arithmétiques et logiques

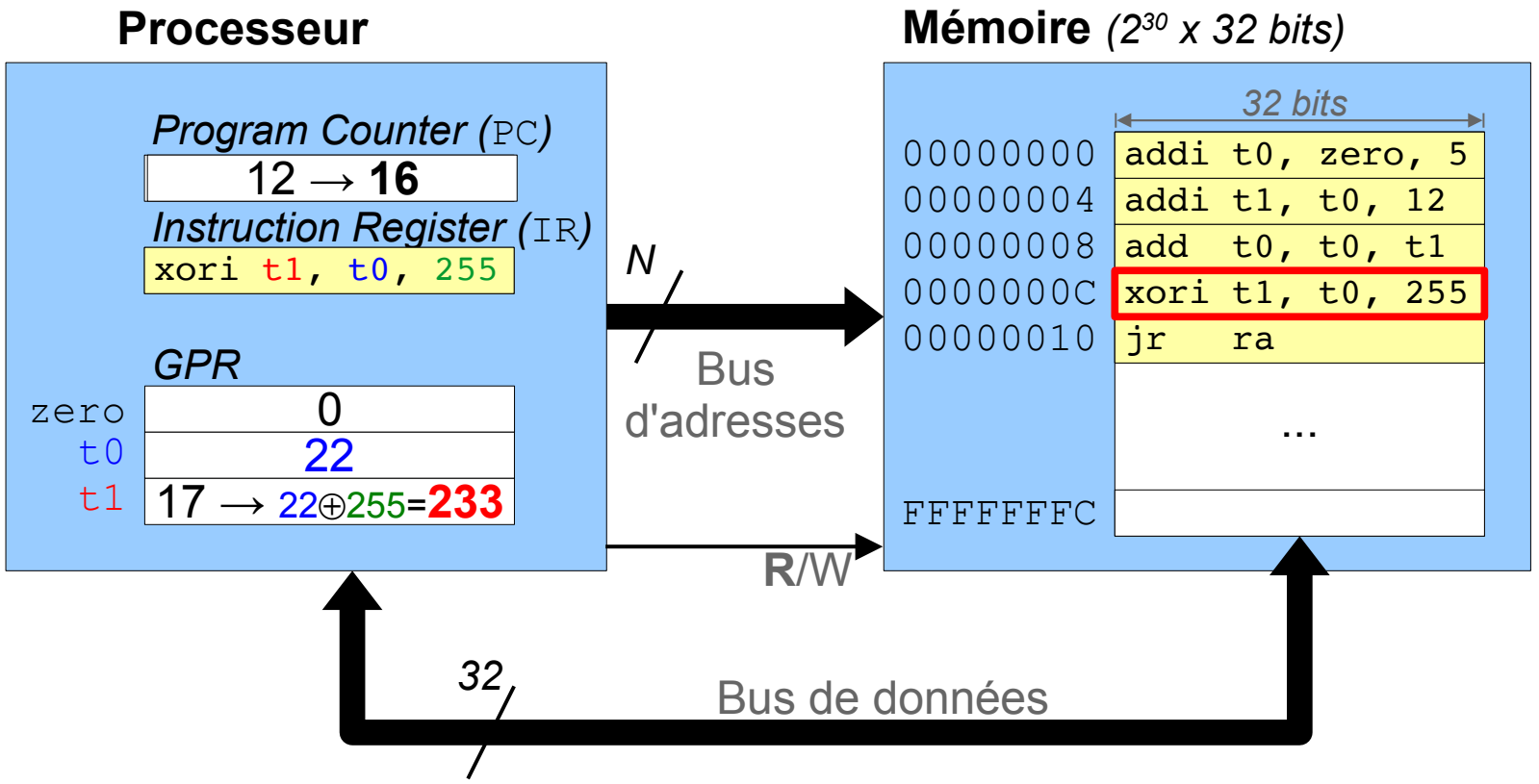


Add

syntaxe: **add** rd, rs, rt

description: GPR[rd] ← GPR[rs] + GPR[rt]

Opérations arithmétiques et logiques



XOR Immediate

syntaxe: **xori** rt, rs, imm

description: GPR**[**rt**]** ← GPR[rs] XOR 0-ext**(**imm**)**

Opérations sur des registres

Exercice

1. Quel est le résultat (`v0`) de la suite d'instructions MIPS suivante ?

```
main:
   li   $a0, 9
   li   $a1, 0xFFFFFFFC
   add  $t0, $a0, $a1
   mult $t0, $t0
   mflo $v0
   jr   $ra
```

2. Comment traduiriez-vous les deux pseudo-instructions `LI` de l'exemple ci-dessus en instructions MIPS ?

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

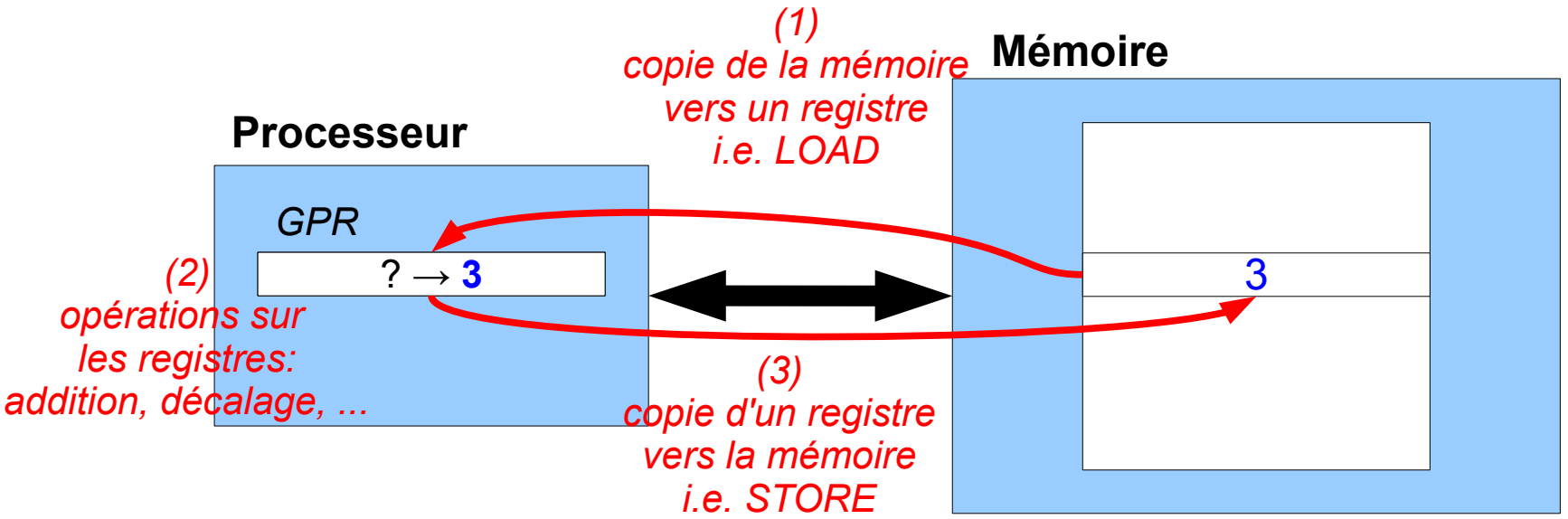
- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Load et Store

Architecture Load-Store

- L'architecture MIPS est une **architecture de type *load-store***. Un processeur MIPS ne travaille pas directement sur les données en mémoire. Celles-ci doivent d'abord être rapatriées dans des registres du processeur avant d'y être modifiées, puis éventuellement recopiées en mémoire.



- Seules deux instructions permettent de charger des données de la mémoire vers un registre interne (*load*) et stocker des données d'un registre interne vers la mémoire (*store*).

Load et Store

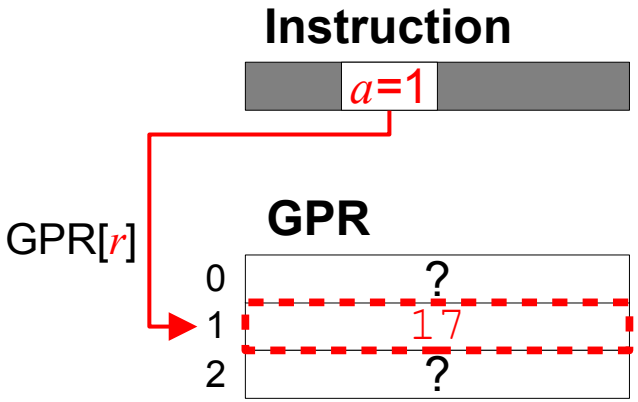
Types d'adressage

- Il existe de nombreuses formes d'adressage. Dans la section précédente, nous avons utilisé, sans les définir, l'adressage de registre et l'adressage immédiat.

Adressage de registre

- La valeur v adressée est celle contenue dans un registre dont l'index/adresse a est stocké dans l'instruction.

$$v = \text{GPR}[a]$$



Adressage immédiat

- La valeur v adressée est stockée dans l'instruction, sous forme d'une valeur immédiate a.

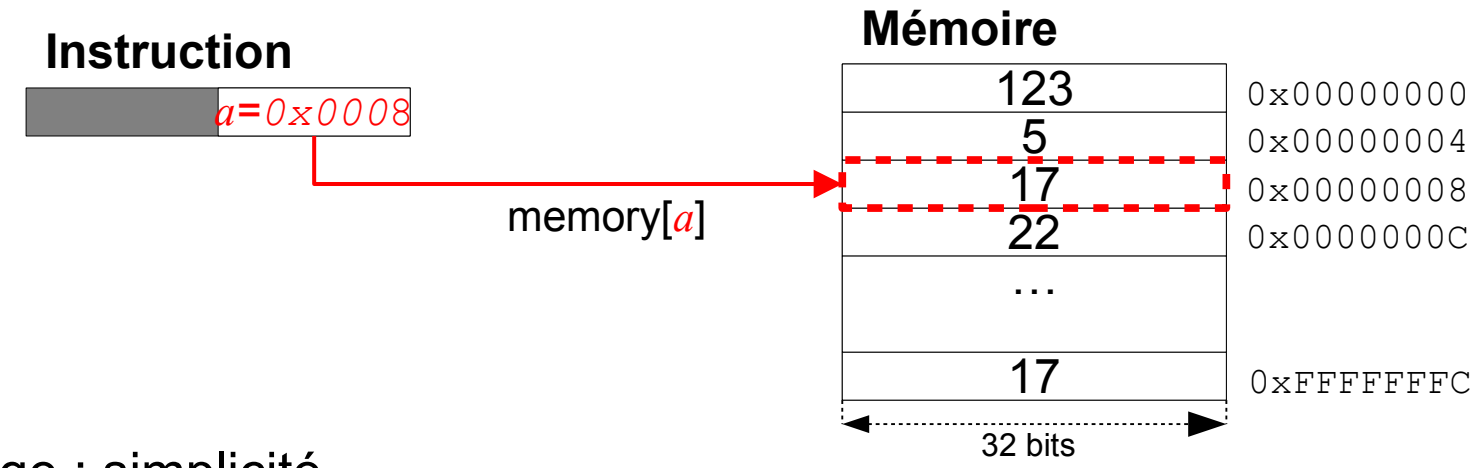
$$v = a$$

Load et Store

Adressage direct

- L'adressage direct est le plus simple adressage de la mémoire. L'adresse mémoire ***a*** est directement contenue dans l'instruction, sous forme d'une valeur immédiate.

v = memory[***a***]



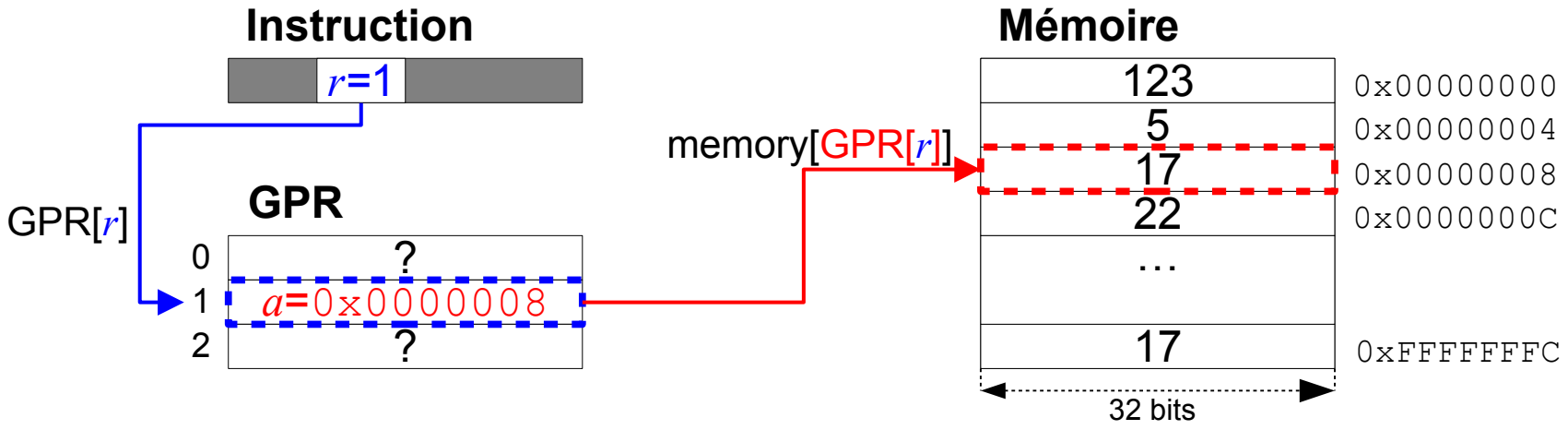
- Avantage : simplicité
- Inconvénient : avec les instructions MIPS de taille fixe (32 bits), l'espace disponible **dans l'instruction** pour représenter une adresse est limité. En enlevant l'*opcode* et une adresse de registre, il reste au maximum 21 bits.

Load et Store

Adressage indirect par registre (*register indirect addressing*)

- Dans cette forme d'adressage, l'adresse ***a*** n'est pas encodée directement dans l'instruction. Elle est placée préalablement dans un registre. L'adresse ***r*** de ce registre est contenue dans l'instruction.

$$v = \text{memory}[\text{GPR}[r]]$$



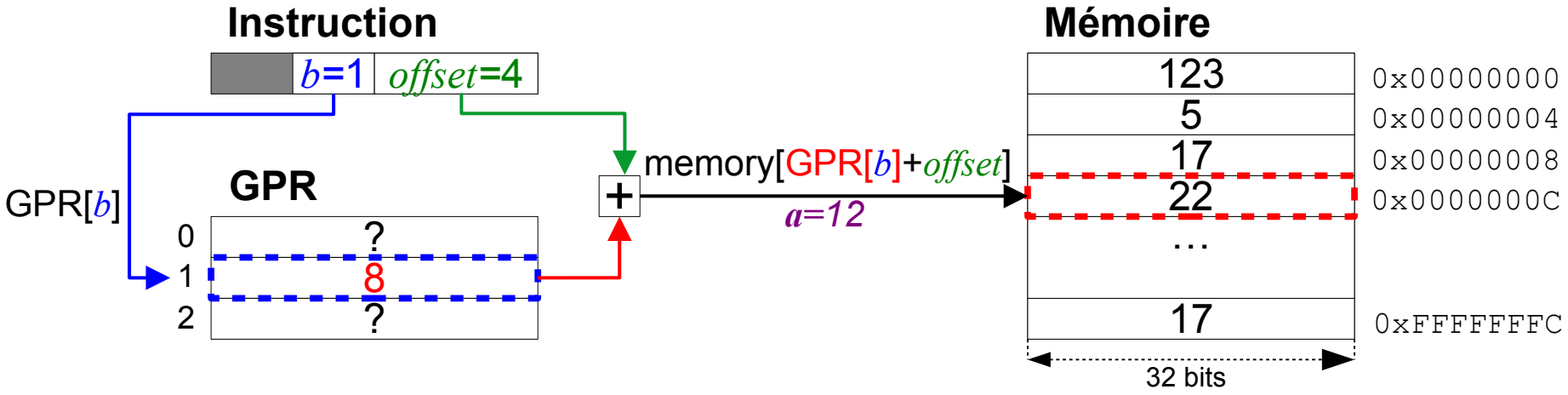
- Avantage : un registre GPR permet de contenir une adresse de taille 32 bits (pour MIPS) suffisante pour adresser toute la mémoire.
- Désavantage : le registre doit être chargé au préalable avec l'adresse, à l'aide d'autres instructions (p.ex. `LA`/`LUI`).

Load et Store

Adressage indirect indexé (base / displacement addressing)

- Cette forme d'adressage est similaire à l'adressage indexé. L'instruction contient l'adresse ***b*** d'un registre et un décalage (*offset*). Le registre b contient une adresse « base ». L'adresse mémoire ***a*** est obtenue en additionnant le contenu du registre d'adresse ***b*** et l'*offset*.

$$v = \text{memory}[\underbrace{\text{GPR}[b] + offset}_{a}]$$



- Avantages : adresse de taille N bits (32 pour MIPS) + permet d'adresser plusieurs cellules mémoire consécutives sans modifier l'adresse de base (contenue dans registre).

Load et Store

Instructions load et store

- Les instructions load et store utilisent un **adressage indirect indexé**. L'adresse d'accès (lecture ou écriture) doit être préalablement chargée dans un registre. L'adresse finale est le contenu de ce registre auquel est ajouté un décalage appelé *offset*.
- Les instructions load et store utilisent l'encodage de **type I**.

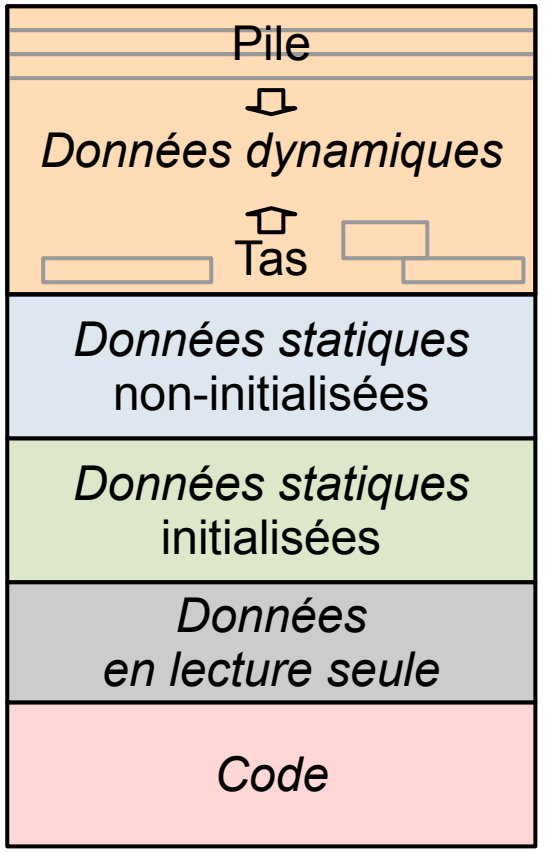
Instruction	Description	OpCode
Load Byte		
LB rt, offset(base)	GPR[rt] ← memory[GPR[base] + *s-ext*(offset)]	100000
Load Half-Word		
LH rt, offset(base)	GPR[rt] ← memory[GPR[base] + *s-ext*(offset)]	100001
Load Word		
LW rt, offset(base)	GPR[rt] ← memory[GPR[base] + *s-ext*(offset)]	100011
Store Byte		
SB rt, offset(base)	memory[GPR[base] + *s-ext*(offset)] ← GPR[rt]	101000
Store Half-Word		
SH rt, offset(base)	memory[GPR[base] + *s-ext*(offset)] ← GPR[rt]	101001
Store Word		
SW rt, offset(base)	memory[GPR[base] + *s-ext*(offset)] ← GPR[rt]	101011

- Les adresses des accès doivent être alignées sur les tailles de mots.

Load et Store

Organisation de la mémoire

- La mémoire d'un programme est typiquement organisée en différentes sections :
  - code (`.text`) – instructions
  - données en lecture seule (`.rodata`)
  - données statiques
    - initialisées (`.data`)
    - non-initialisée (`.bss`)
  - données dynamiques
    - tas (*heap*)
    - pile (*stack*)



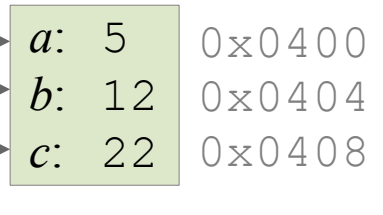
Load et Store

Adressage indirect vs indirect indexé

- Exemple dans lequel on accède à 3 mots contigus en mémoire.

```
    .data
a:  .word 5
b:  .word 12
c:  .word 22
```

```
    .text
main:
    la   $t0, b
    lw   $t1, 0($t0)
    la   $t0, c
    lw   $t2, 0($t0)
    add  $t1, $t1, $t2
    la   $t0, a
    sw   $t1, 0($t0)
    jr   $ra
```

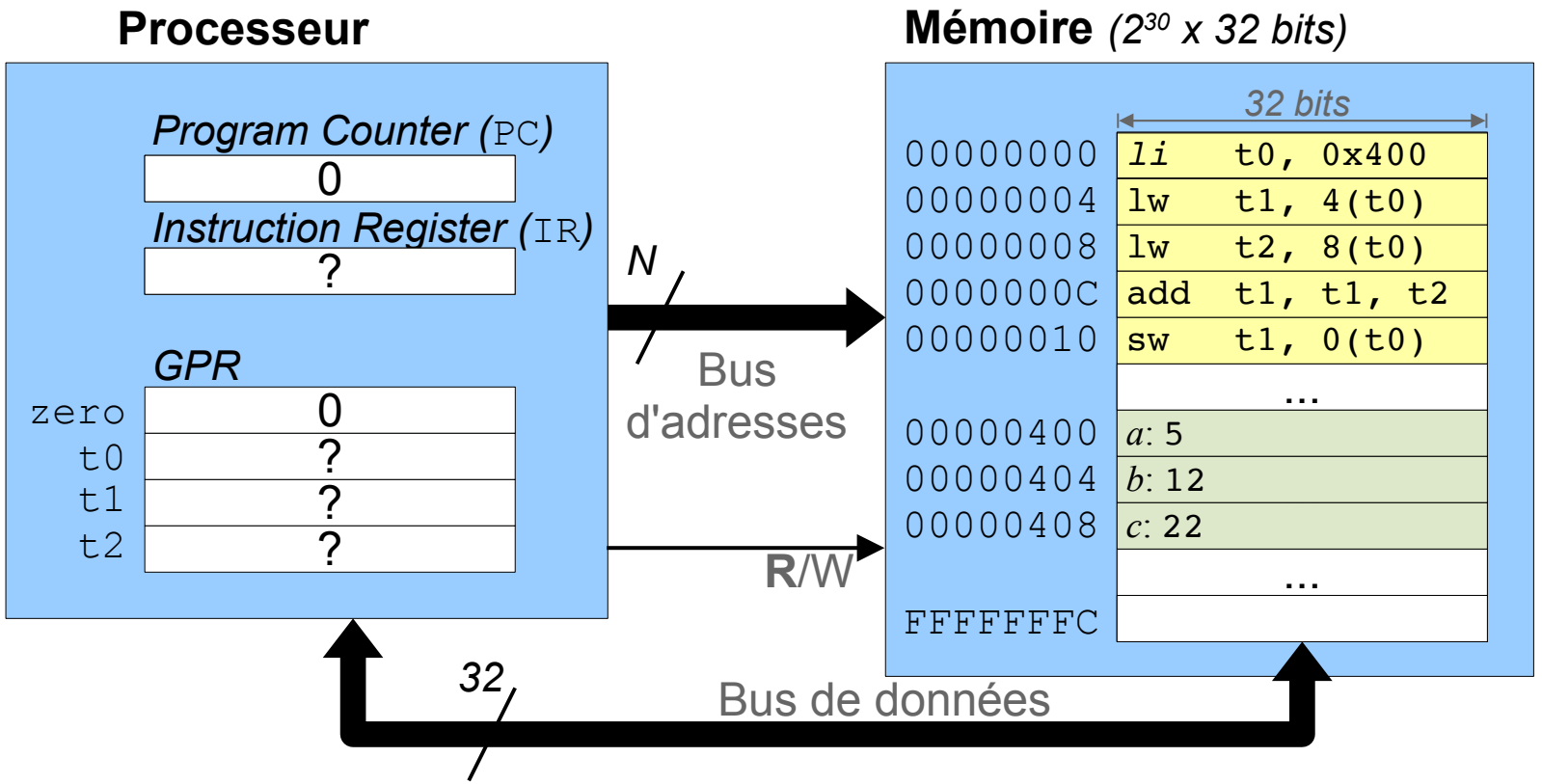



```
    .text
main:
    la   $t0, a
    lw   $t1, 4($t0)
    lw   $t2, 8($t0)
    add  $t1, $t1, $t2
    sw   $t1, 0($t0)
    jr   $ra
```

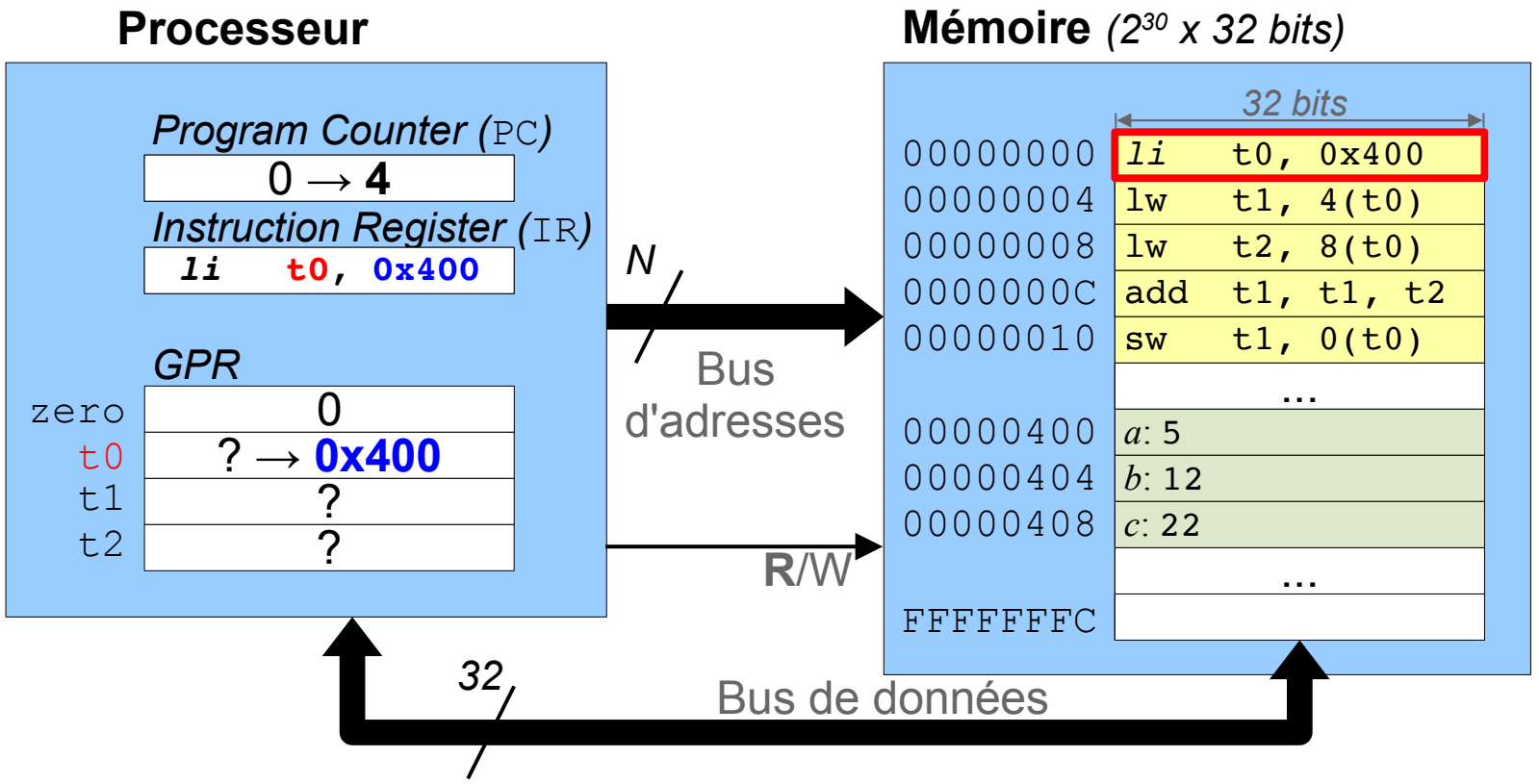
Adressage indirect
nécessité de charger l'adresse de chaque mot

Adressage indirect indexé
chargement de l'adresse du premier mot, puis utilisation de décalages différents → moins d'instructions

Load et Store



Load et Store

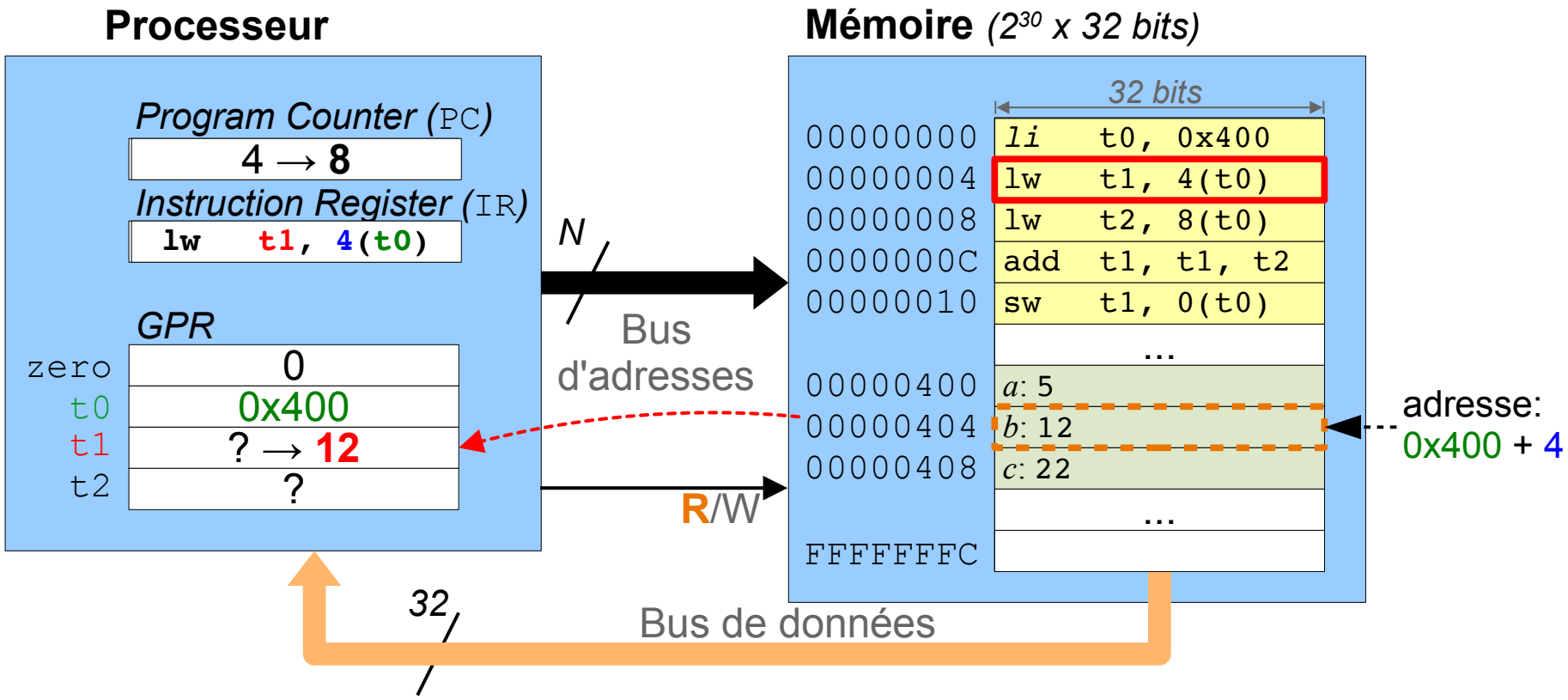


Load Immediate (pseudo-instruction)

syntaxe: `li rdest, imm`

description: GPR[rd] ← imm

Load et Store

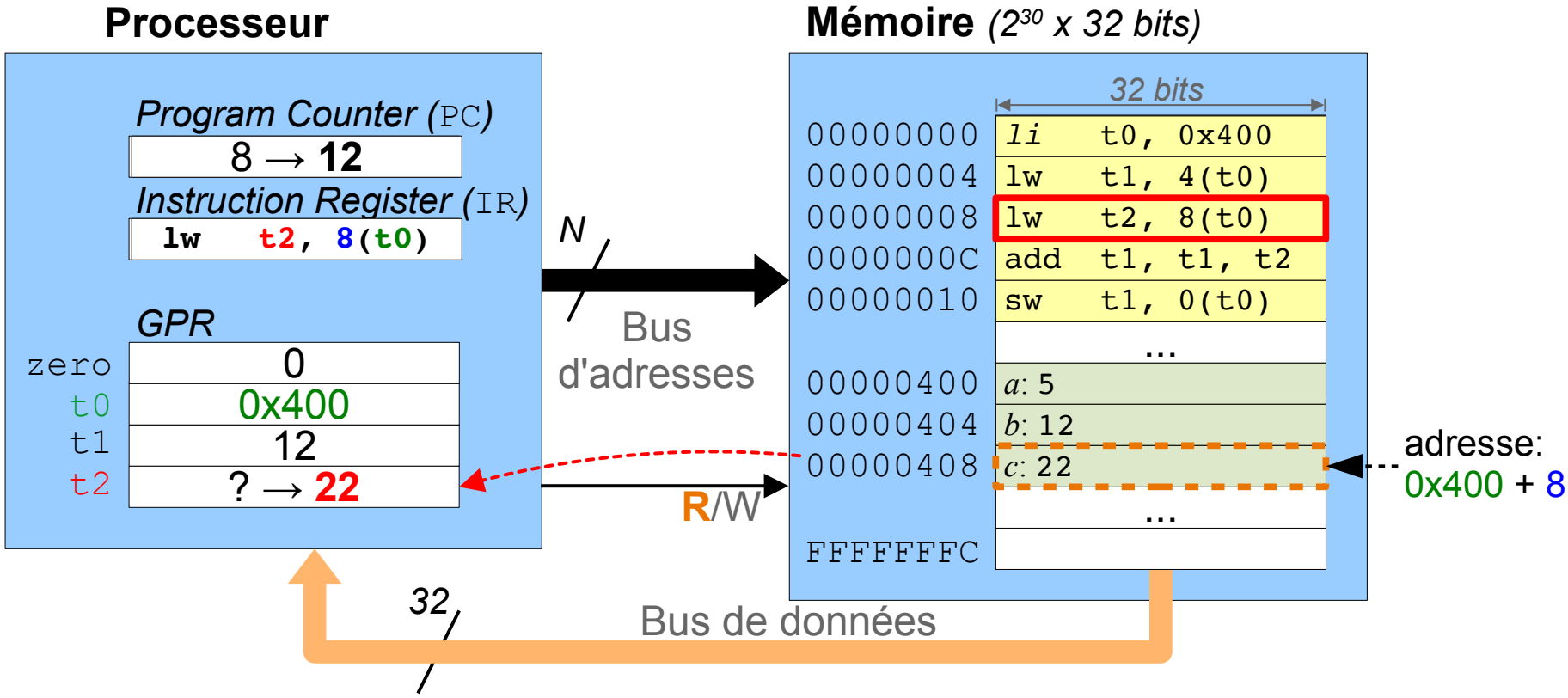


Load Word

syntaxe: `lw rt, offset(base)`

description: GPR[rd] ← memory[GPR[base] + offset]

Load et Store

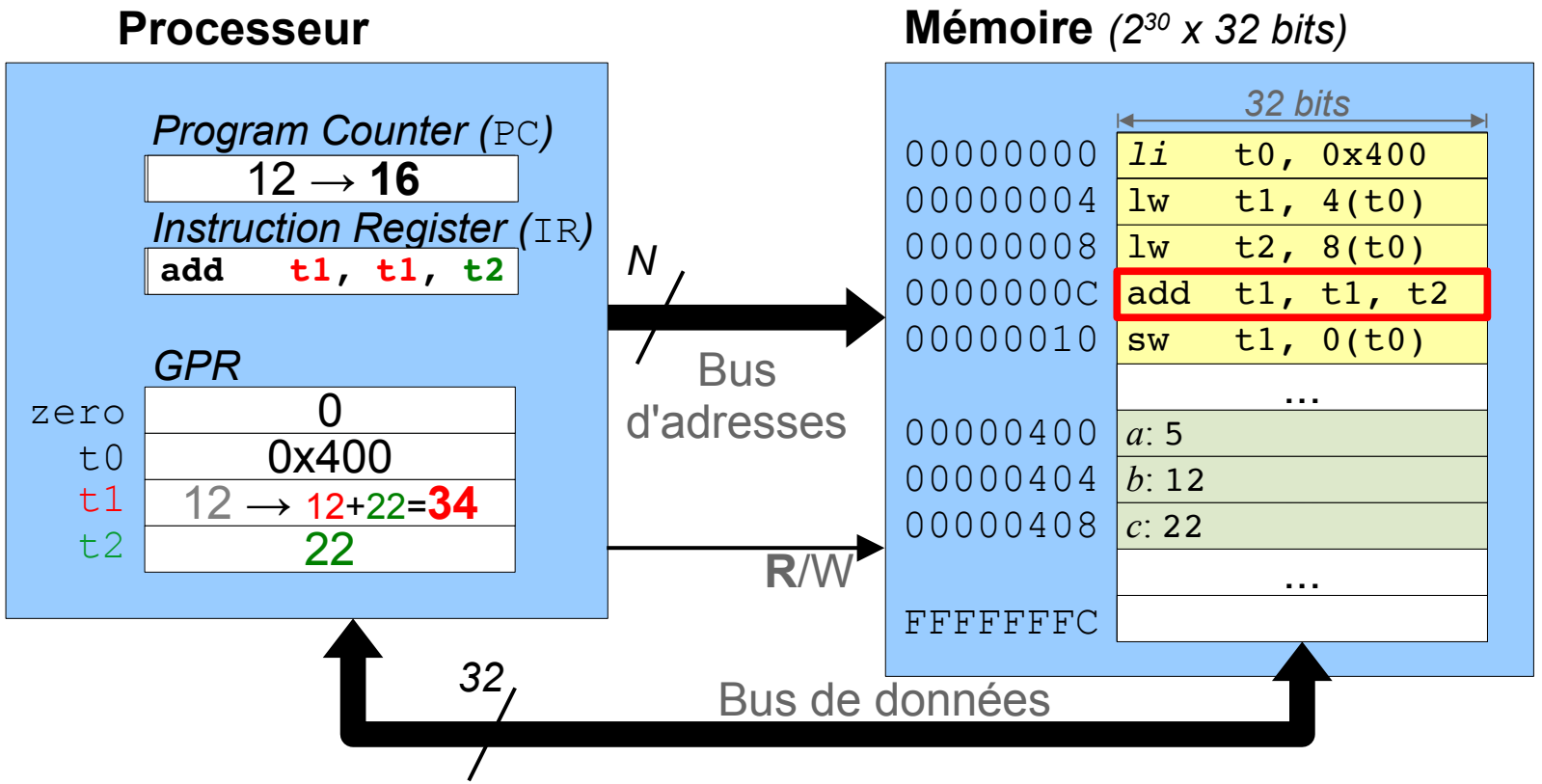


Load Word

syntaxe: `lw rt, offset(base)`

description: GPR[rd] ← memory[GPR[base] + offset]

Load et Store

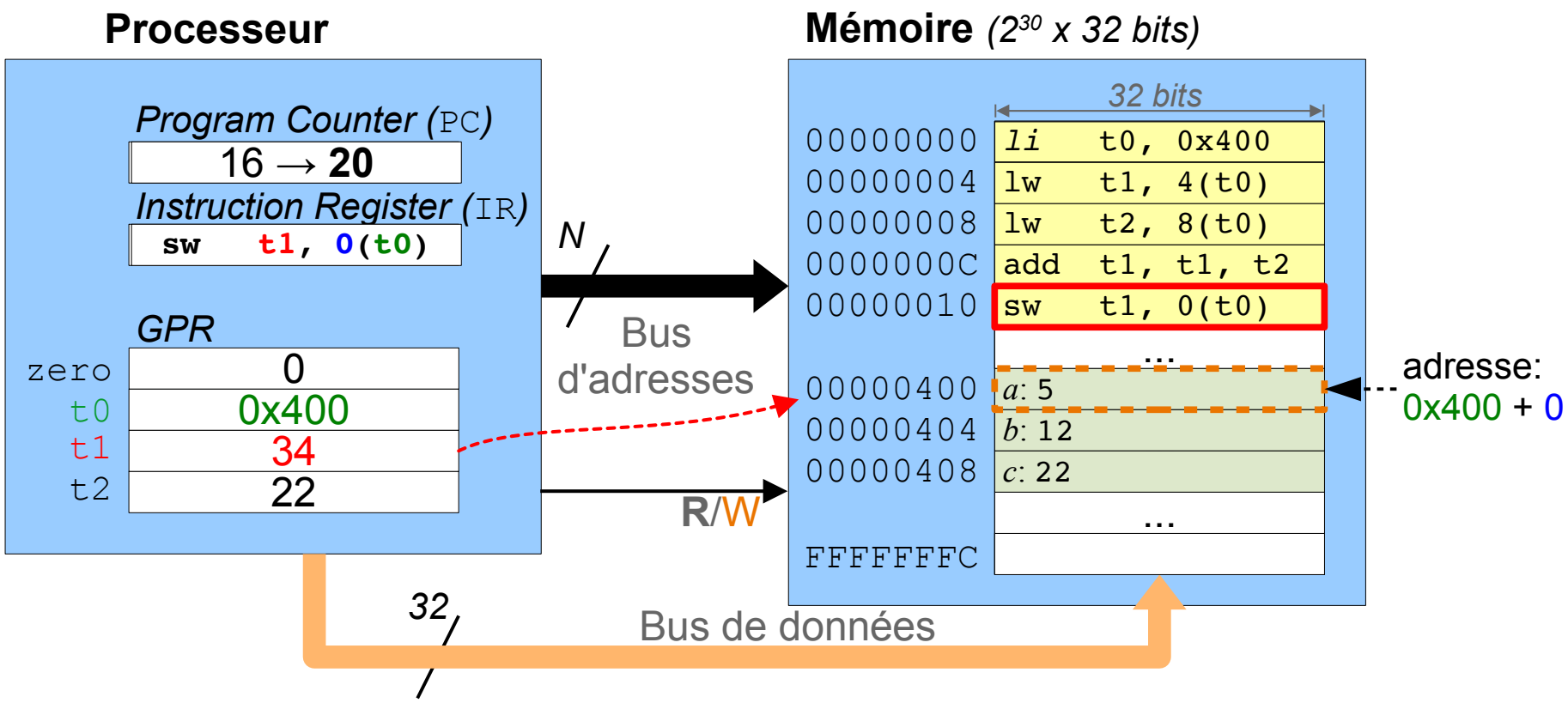


Add

syntaxe: **add** rd, rs, rt

description: GPR[rd] ← GPR[rs] + GPR[rt]

Load et Store



Store Word

syntaxe: `sw rt, offset(base)`

description: memory[GPR[base] + offset] ← GPR[rd]

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Contrôle de flux

Introduction

- Les **instructions de contrôle de flux** permettent de s'écarter du comportement séquentiel d'un programme. Ces instructions consistent à prendre des **décisions** sur base de **tests** et changer la suite des instructions exécutées en effectuant un **branchement** à une nouvelle adresse d'instruction.

- Cette section s'intéresse aux instructions qui permettent les constructions suivantes, disponibles dans la plupart des langages de haut niveau.
  - les branchements inconditionnels (`goto`)
  - l'exécution conditionnelle (`if`)
  - la sélection entre deux cas (`if-then-else`)
  - les boucles (`while`, `for`, ...)
  - la sélection entre N cas (`switch`, `case`)

Contrôle de flux

Branchements inconditionnels

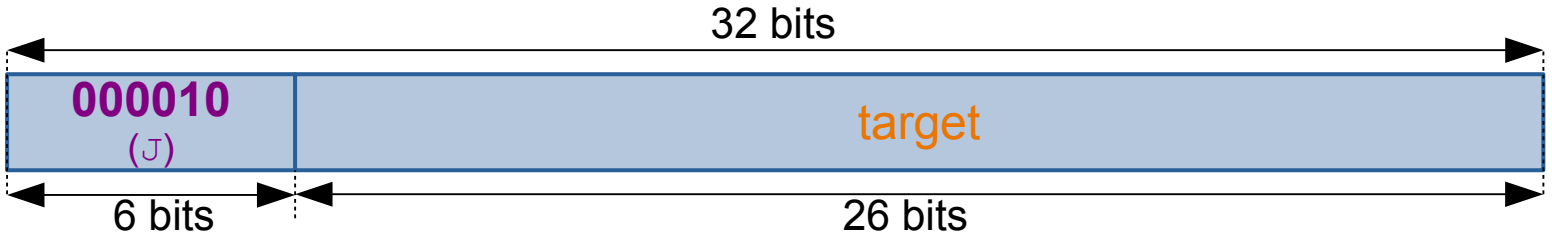
- Un **branchement inconditionnel** ou **saut** (*jump*) est un changement du flux d'exécution du programme sans test préalable de condition. Il consiste à modifier le *Program Counter* (registre `PC`). Le changement est toujours réalisé.
- Le jeu d'instructions MIPS supporte deux instructions de saut : `J` (*jump*) et *Jump Register* (`JR`). Le langage d'assemblage MIPS supporte également une pseudo-instruction de branchement relatif, `B` (*branch*).

Instruction		Description	OpCode
Jump			
J	**target**	PC ← $PC_{31..28}$ || target || 0^2	000010
Jump Register			
JR	**rs**	PC ← GPR[rs]	001000

Sauts

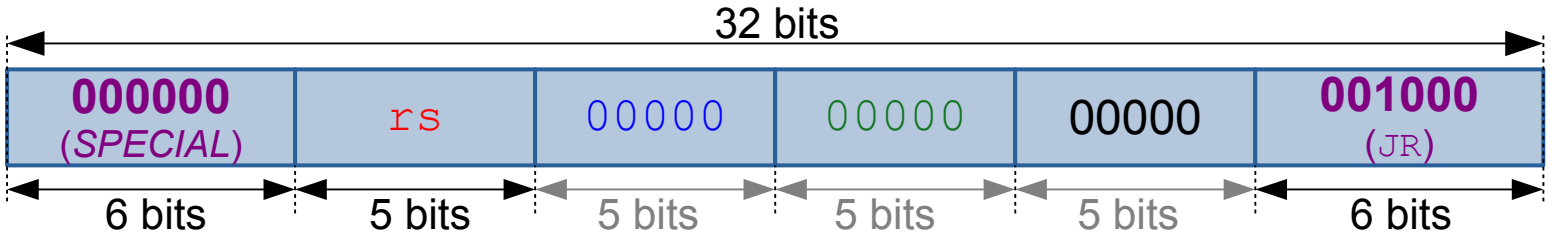
Encodage de *Jump* (`J`)

- L'instruction `J` utilise le format d'encodage **type-J**. Outre l'opcode, l'instruction contient un seul champ `target` occupant la totalité de l'espace restant, soit 26 bits. Il n'est donc pas possible d'encoder des adresses de 32 bits, ce qui contraint les branchements effectués avec `J`.



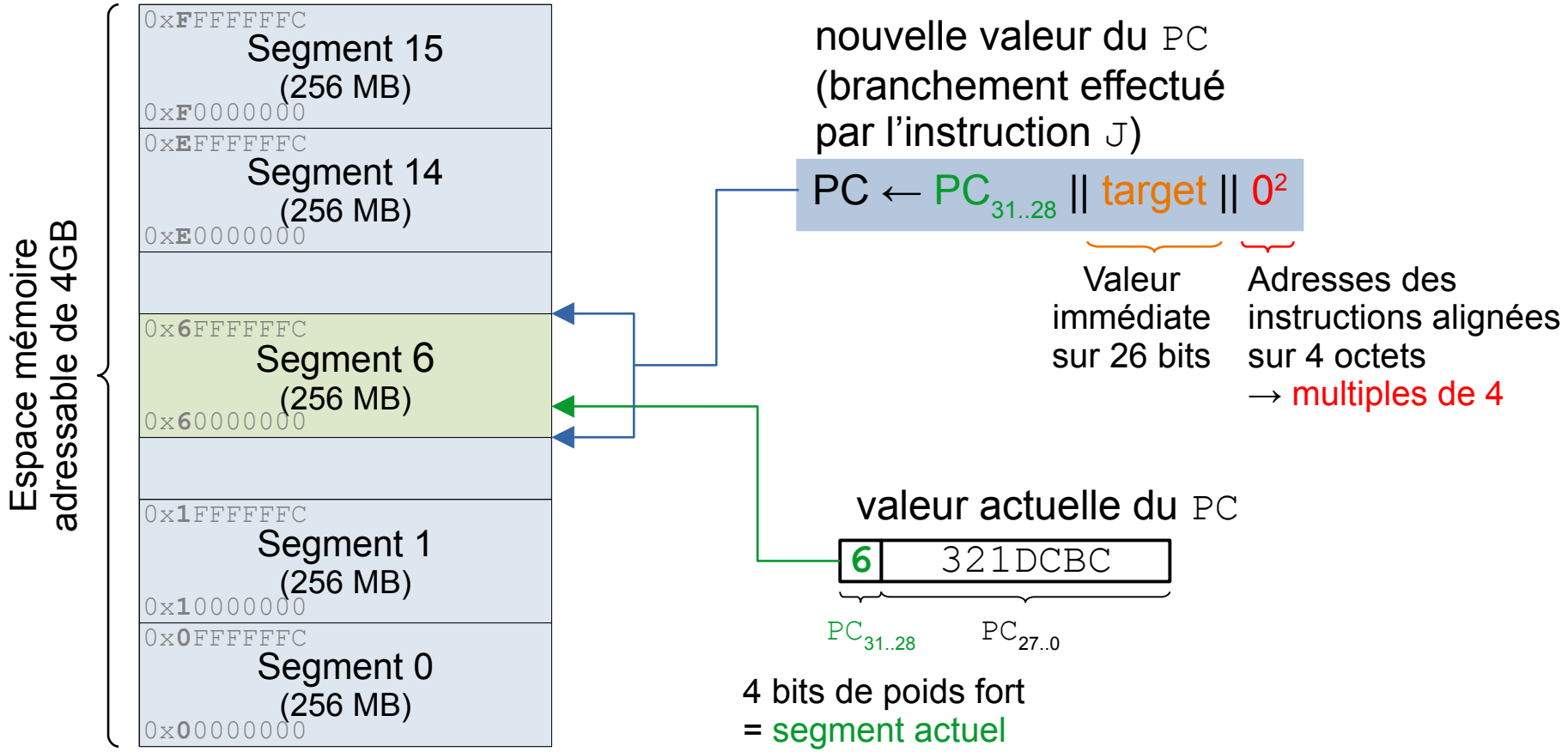
Encodage de *Jump Register* (`JR`)

- L'instruction `JR` utilise l'encodage de **type-R** avec `rs` l'index du registre contenant l'adresse du branchement.



Instruction *Jump* (J)

- Avec l'instruction J, on fait l'hypothèse que l'espace mémoire de 4 GB adressable avec 32 bits d'adresse est découpé en 16 zones disjointes, chacune de 256 MB. L'instruction J ne permet qu'un déplacement à l'intérieur d'une telle zone.



Sauts

Boucle infinie

- Le programme suivant effectue « indéfiniment » l'incrémentation d'une variable préalablement initialisée à 0. La variable est stockée dans le registre `t0`.

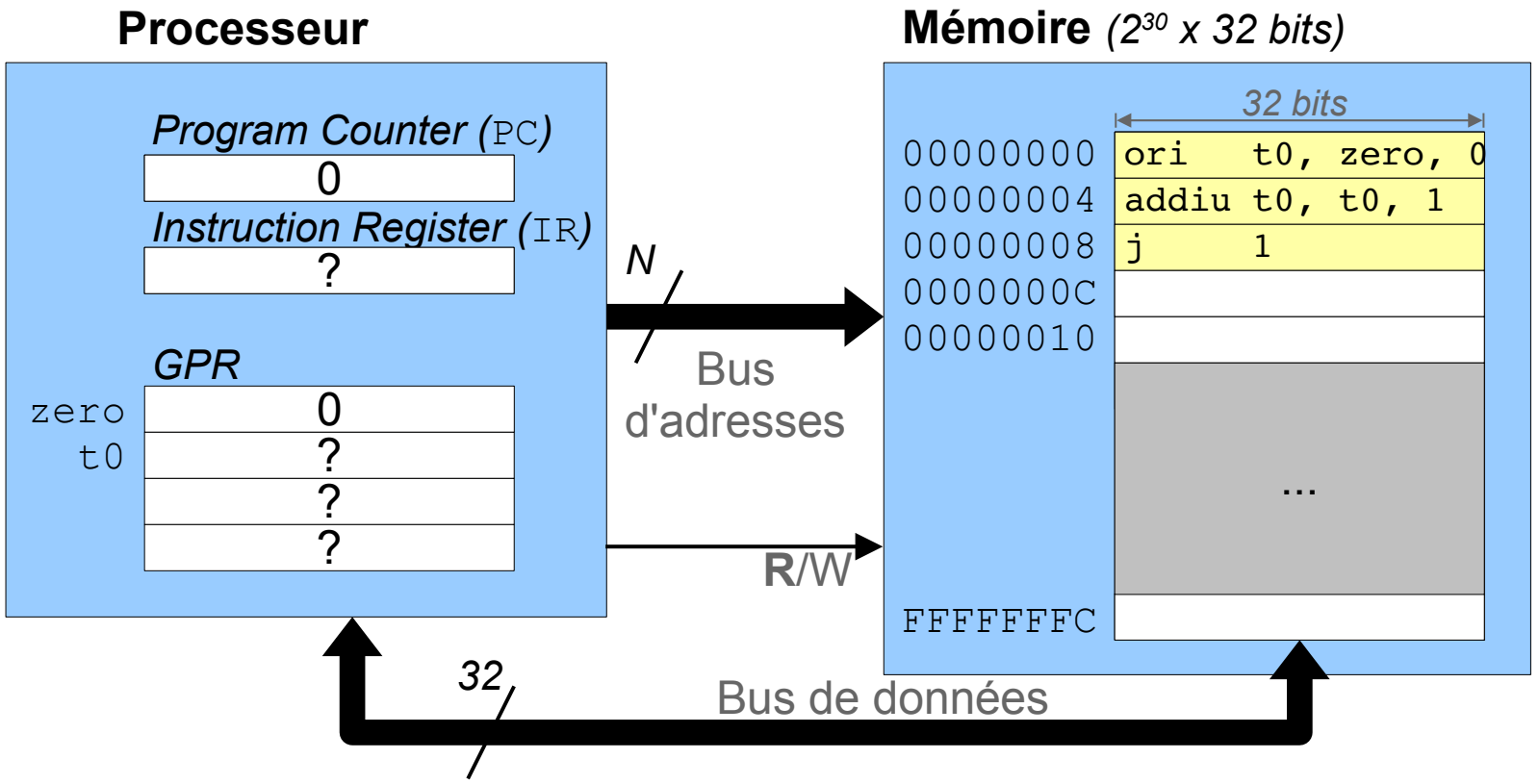
```
0   main:
        ori   $t0, $zero, 0    # initialise valeur à 0
4   loop:
        addiu $t0, $t0, 1      # incrémente variable
        j     loop             # saute à l'adresse loop
```

  - Hypothèse : l'étiquette `main` correspond à l'adresse 0, tandis que l'étiquette `loop` correspond à l'adresse 4.

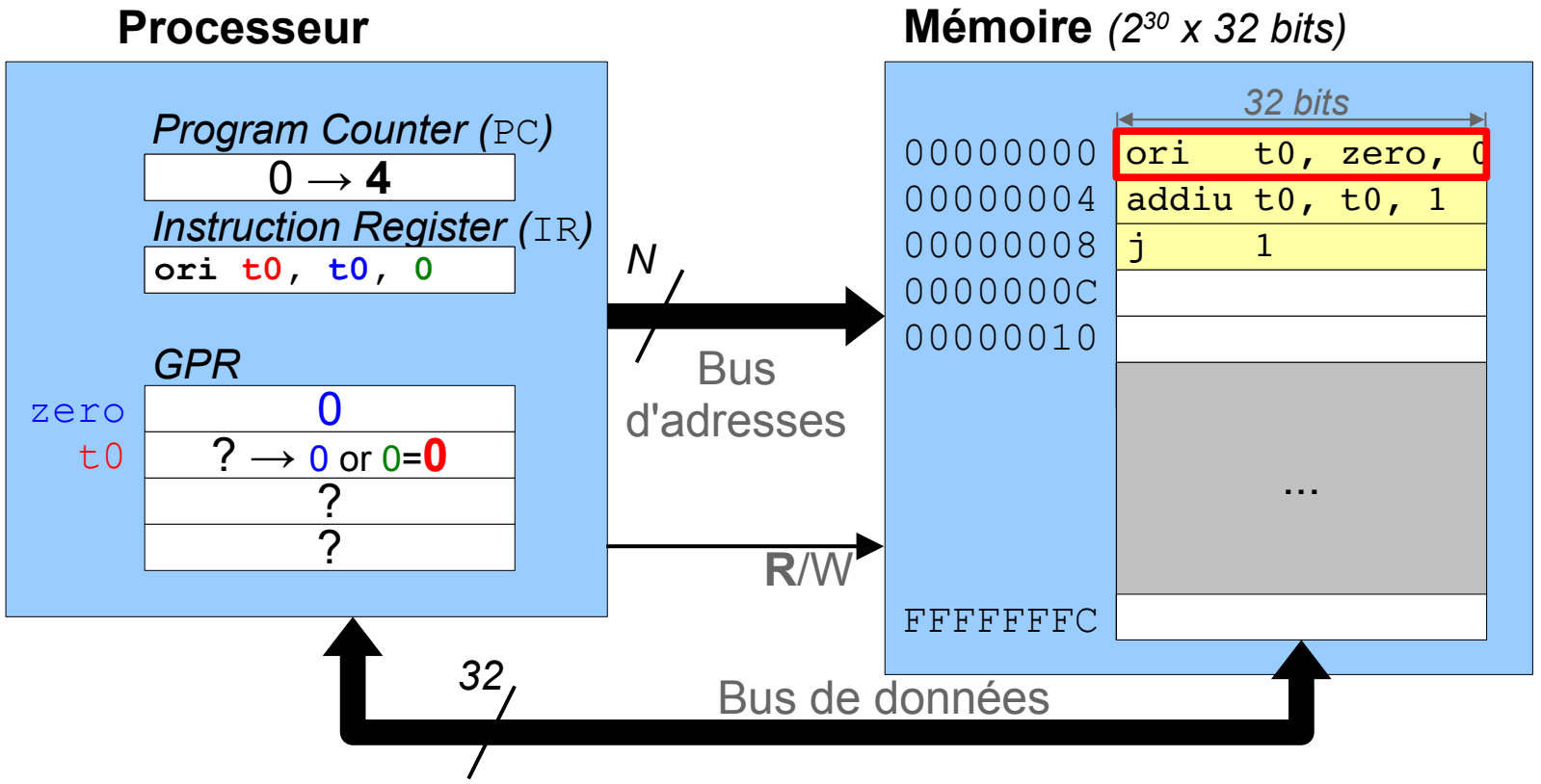
- Pour traduire l'instruction « **`j`** `loop` » en langage machine, l'assembleur doit déterminer la valeur immédiate target de sorte que la nouvelle valeur de `PC` soit égale à la position de l'étiquette `loop` (4). La valeur actuelle de `PC` est l'adresse de l'instruction `J` (8).

$PC \leftarrow PC_{31..28} \,||\, target \,||\, 0^2$ | $4 = 0000 \,||\, 00000000000000000000000001 \,||\, 00$

Sauts


Processeur
Program Counter (PC)
0
Instruction Register (IR)
?
GPR
zero
t0
0
?
?
?
N
Bus
d'adresses
R/W
32
Bus de données
Mémoire (2^30 x 32 bits)
32 bits
00000000
00000004
00000008
0000000C
00000010
FFFFFFFC
ori t0, zero, 0
addiu t0, t0, 1
j 1
...

Sauts

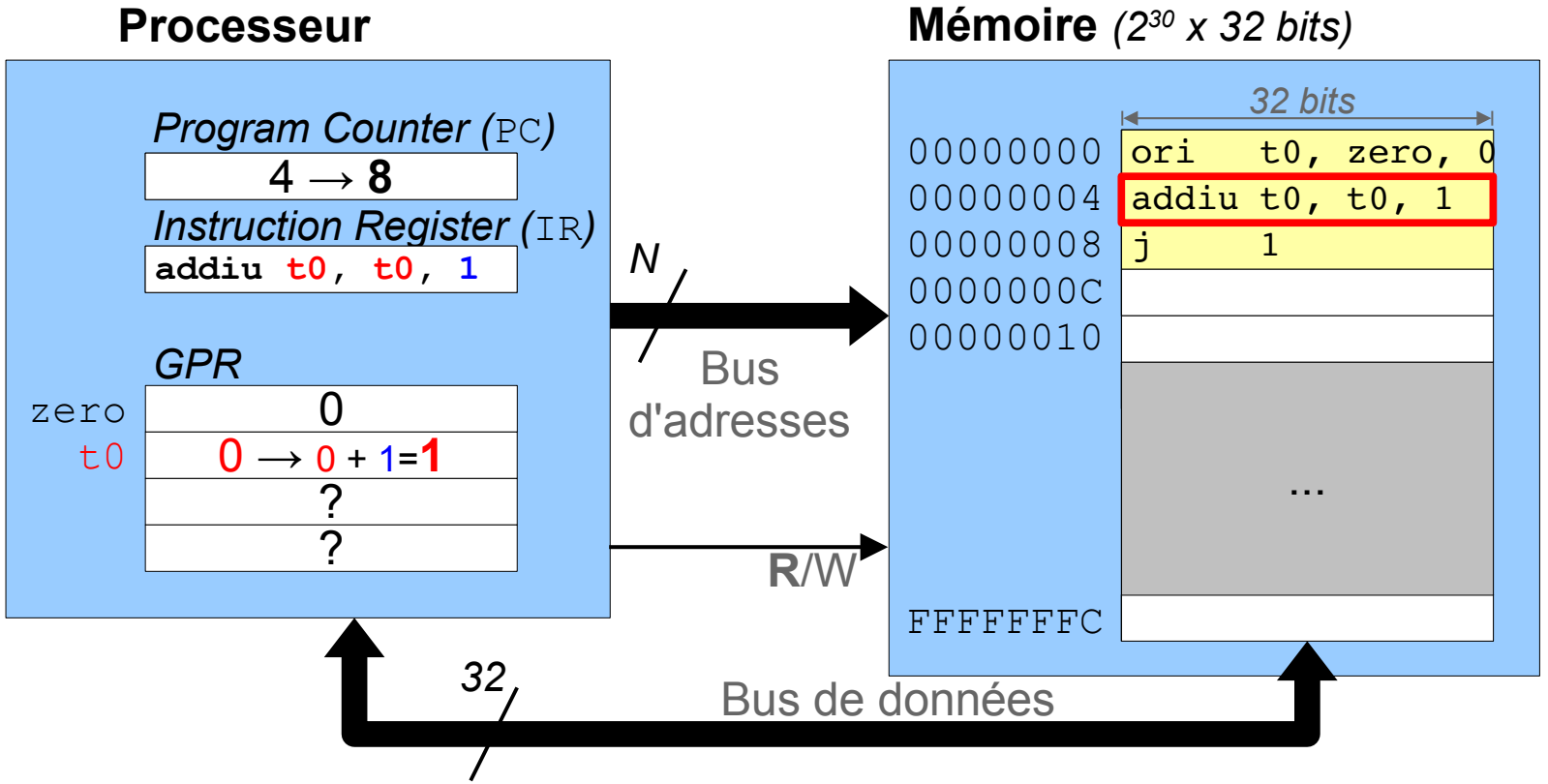


OR Immediate

syntaxe: **ori** rt, rs, imm

description: GPR[**rt**] ← GPR[rs] + 0-ext(imm)

Sauts

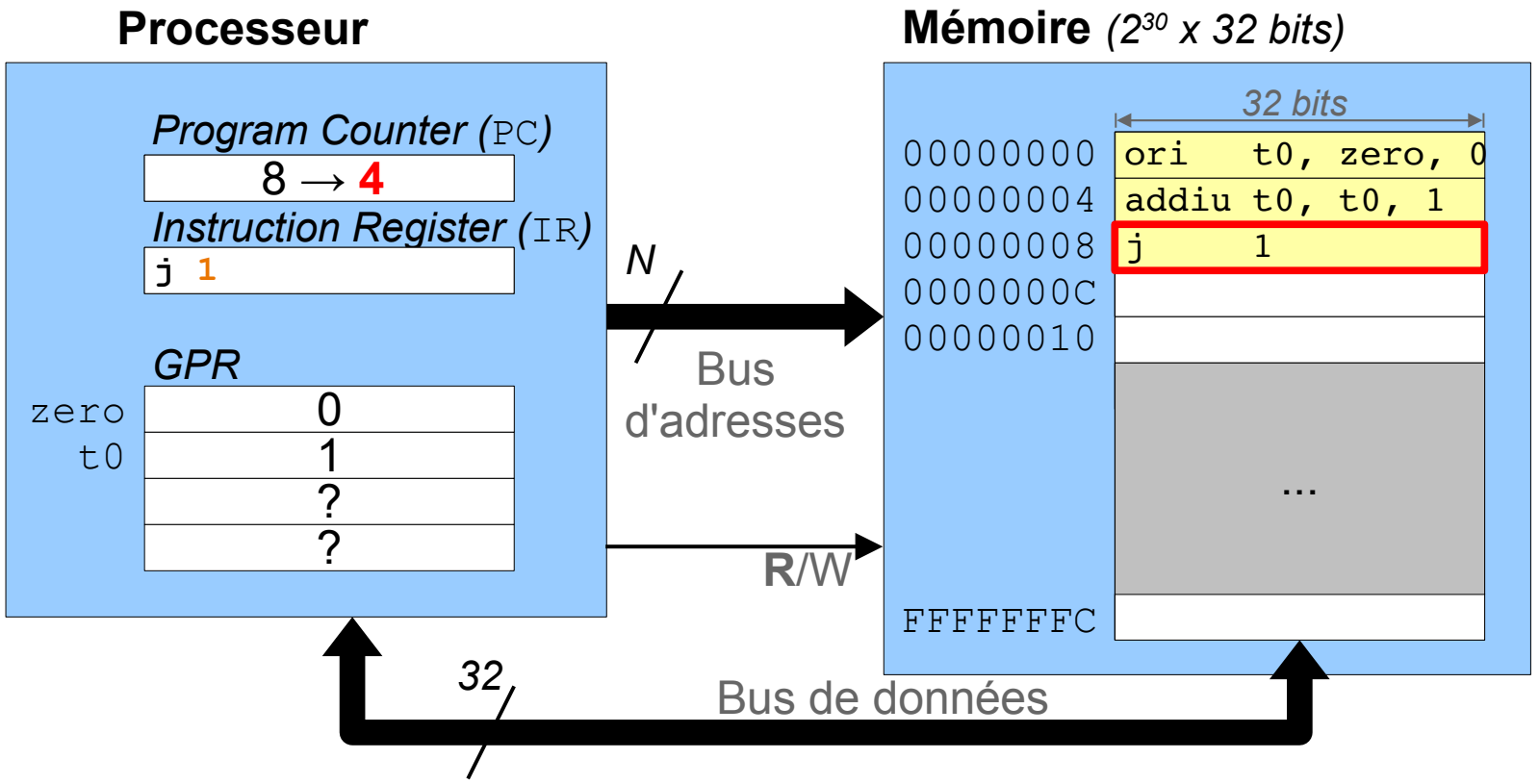


Add Immediate Unsigned

syntaxe: **addiu** rt, rs, imm

description: GPR[rt] ← GPR[rs] + s-ext(imm)

Sauts

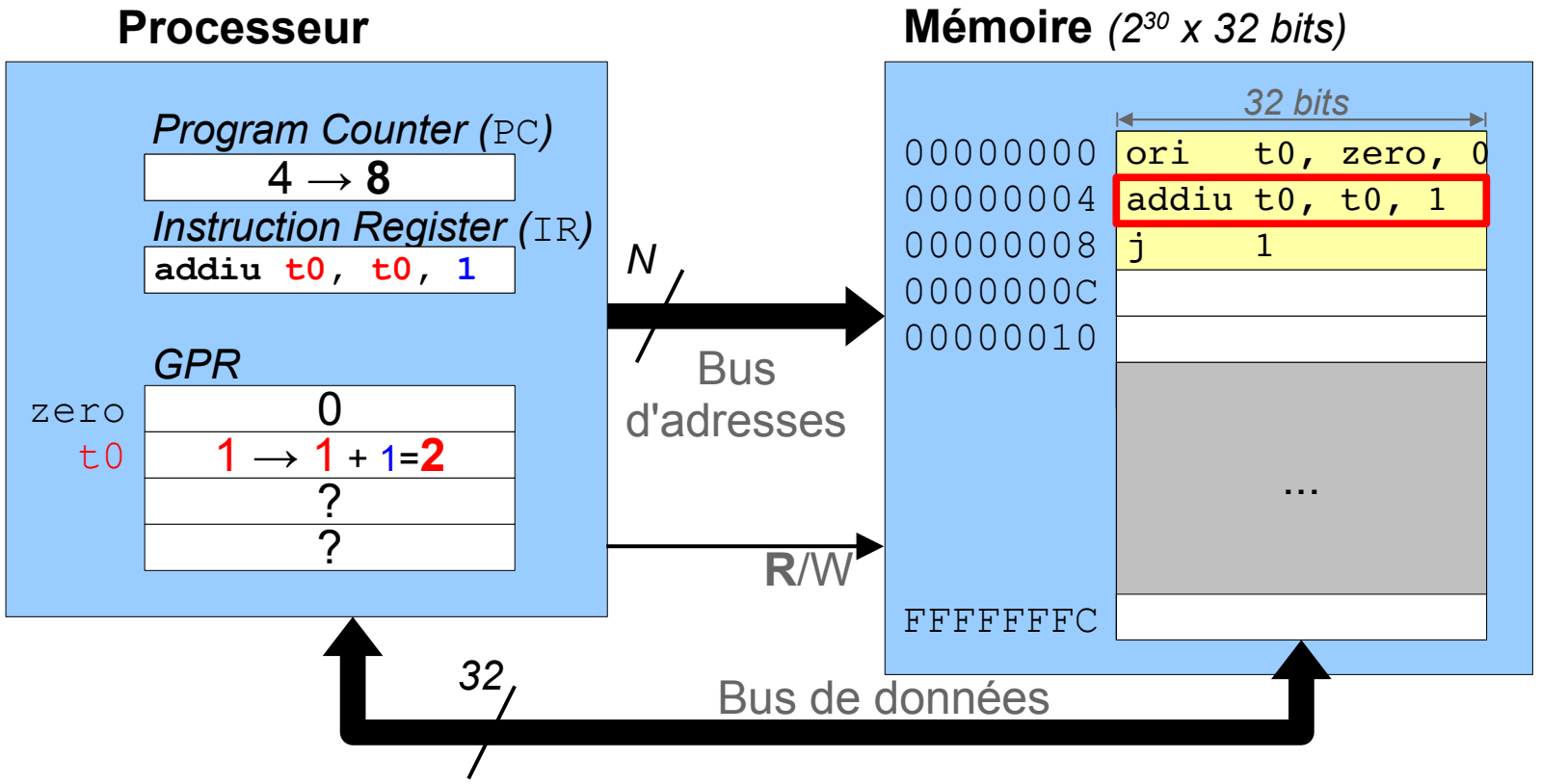


Jump

syntaxe: `j target`

description: $PC \leftarrow PC_{31..28} \,||\, \texttt{target} \,||\, 0^2$

Sauts

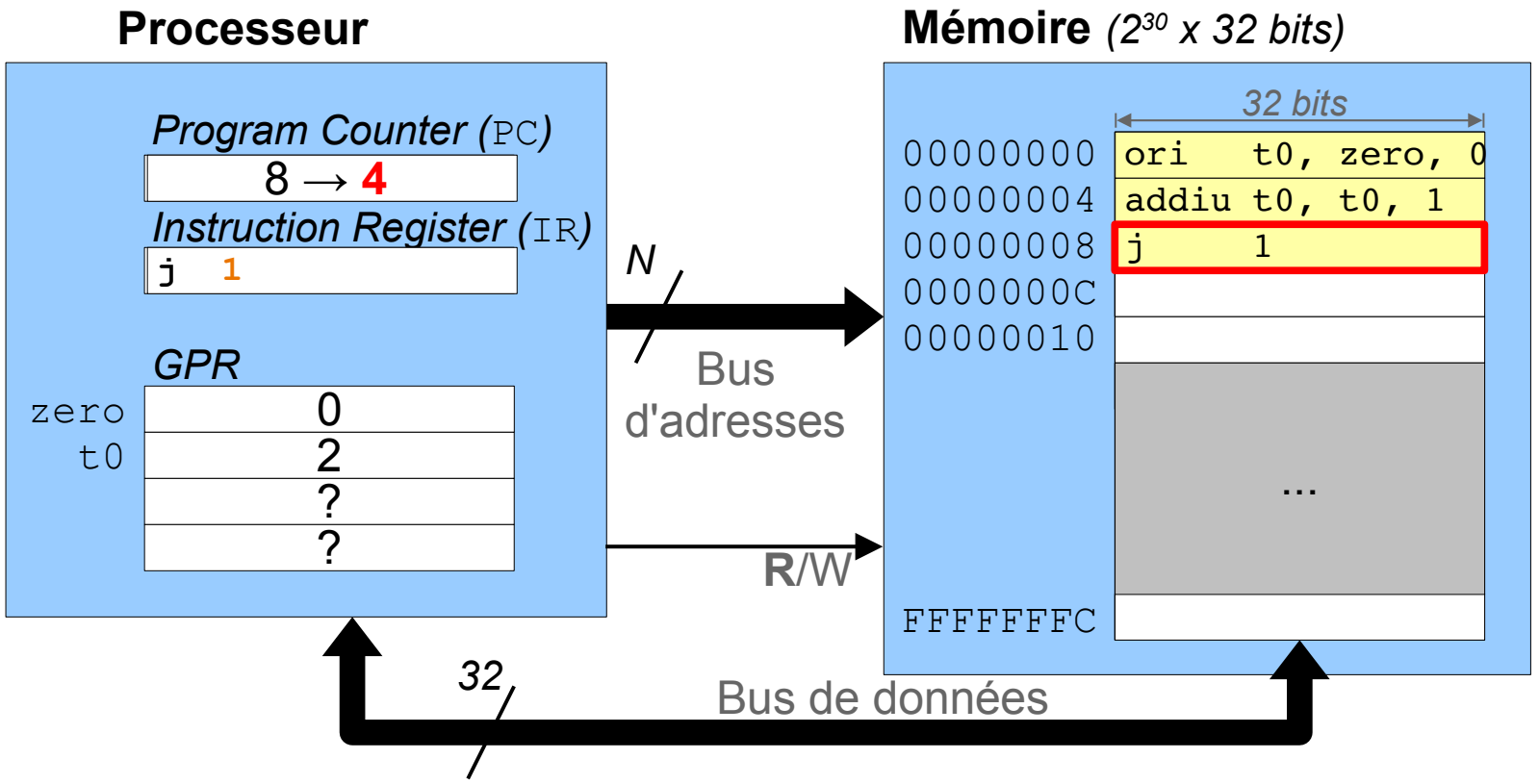


Add Immediate Unsigned

syntaxe: **addi** rt, rs, imm

description: GPR[**rt**] ← GPR[rs] + s-ext(imm)

Sauts



Jump

syntaxe: `j target`

description: $PC \leftarrow PC_{31..28} \,||\, \texttt{target} \,||\, 0^2$

Sauts

Impact du *pipelining*

- L'architecture MIPS est prévue pour fonctionner en *pipeline*. Une particularité de MIPS est que chaque branchement est suivi d'un *branch delay slot*. durant lequel le *Program Counter* (`PC`) est mis à jour.
- Ce choix de conception a pour impact que l'instruction qui suit un branchement ou un saut est toujours exécutée, même s'il y a branchement.
- Dans le simulateur SPIM, ce comportement n'est obtenu que si l'option «*Enable Delayed Branches* » est activée.

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Contrôle de flux

Branchement conditionnel

- Lorsqu'elles sont associées au test d'une condition, les instructions de saut deviennent des **branchements conditionnels**. Le branchement est effectué ou non selon le résultat du test.

- Le jeu d'instructions MIPS supporte quelques instructions permettant d'effectuer des branchements sur base des conditions suivantes
  - égalité ou non-égalité de 2 registres
  - comparaison d'un registre avec 0

- Ces instructions sont de **type I** et effectuent des branchements relatifs.

Branchements conditionnels

Branchement conditionnel relatif

- Un branchement **relatif** est effectué en ajoutant au registre `PC` incrémenté de 4 la valeur immédiate (`offset`) de l'instruction après décalage de 2 vers la gauche et extension signée.

  PC ← PC + 4 + *s-ext*(`offset` || 0^2)

  - Notes :
    - déplacement compris entre -131072 (-2^{17}) et 131068 (2^{17}-4).
    - le décalage de 2 (x 4) assure que `PC` reste un multiple de 4.

- Ainsi une instruction de branchement conditionnel aura le comportement suivant:

  si *condition vraie* **alors**
  PC ← PC + 4 + *s-ext*(`offset` || 0^2)
  sinon
  PC ← PC + 4

Branchements conditionnels

Branchement conditionnel

- Les instructions MIPS de branchement conditionnel sont

Instruction	Description	OpCode	rt
Branch on Equal			
BEQ rs, rt, offset	if GPR[rs] = GPR[rt] then branch	000100	
Branch on Not Equal			
BNE rs, rt, offset	if GPR[rs] ≠ GPR[rt] then branch	000101	
Branch on Greater than or Equal to Zero			
BGEZ rs, offset	if GPR[rs] ≥ 0 then branch	000001	00100
Branch on Greater Than Zero			
BGTZ rs, offset	if GPR[rs] > 0 then branch	000111	
Branch on Less than or Equal to Zero			
BLEZ rs, offset	if GPR[rs] ≤ 0 then branch	000110	
Branch on Less Than Zero			
BLTZ rs, offset	if GPR[rs] < 0 then branch	000001	00000

Note : **BGEZ** et **BLTZ** ont le même *OpCode* et sont distinguées sur base des bits 20:16 (`rt`).

Branchements conditionnels

Pseudo-instructions de branchement conditionnel

- Le langage d'assemblage MIPS fournit plusieurs pseudo-instructions permettant d'exprimer d'autres conditions de branchement. Ces pseudo-instructions sont traduites par l'assembleur en une ou plusieurs instructions réelles.

- Branchement inconditionnel : *Branch* (***b***)

```
b   offset
```
⇨
```
beq   $zero, $zero, offset
```

- Branchement en cas d'égalité avec zéro : *Branch on Equal Zero* (***beqz***)

```
beqz   rsrc, offset
```
⇨
```
beq   rsrc, $zero, offset
```

- comparaison entre registres : *Branch on Greater than Equal* (***bge***)

```
bge   rsrc1, rsrc2, offset
```
⇨
```
slt   $at, rsrc1, rsrc2
beq   $at, $zero, offset
```

Branchements conditionnels

Exemple

- Le programme suivant effectue 2 itérations d'une boucle.
  - une variable dans le registre `t0` est initialisée à la valeur 2
  - à chaque itération, la variable est décrémentée
  - la boucle continue tant que `t0` > 0

```
0   main:
            li          $t0, 2          # initialise variable à 2
4   loop:
            addi        $t0, $t0, -1    # décrémente variable
            bgtz        $t0, loop       # saute à l'adresse loop
                                        # si t0 > 0
```

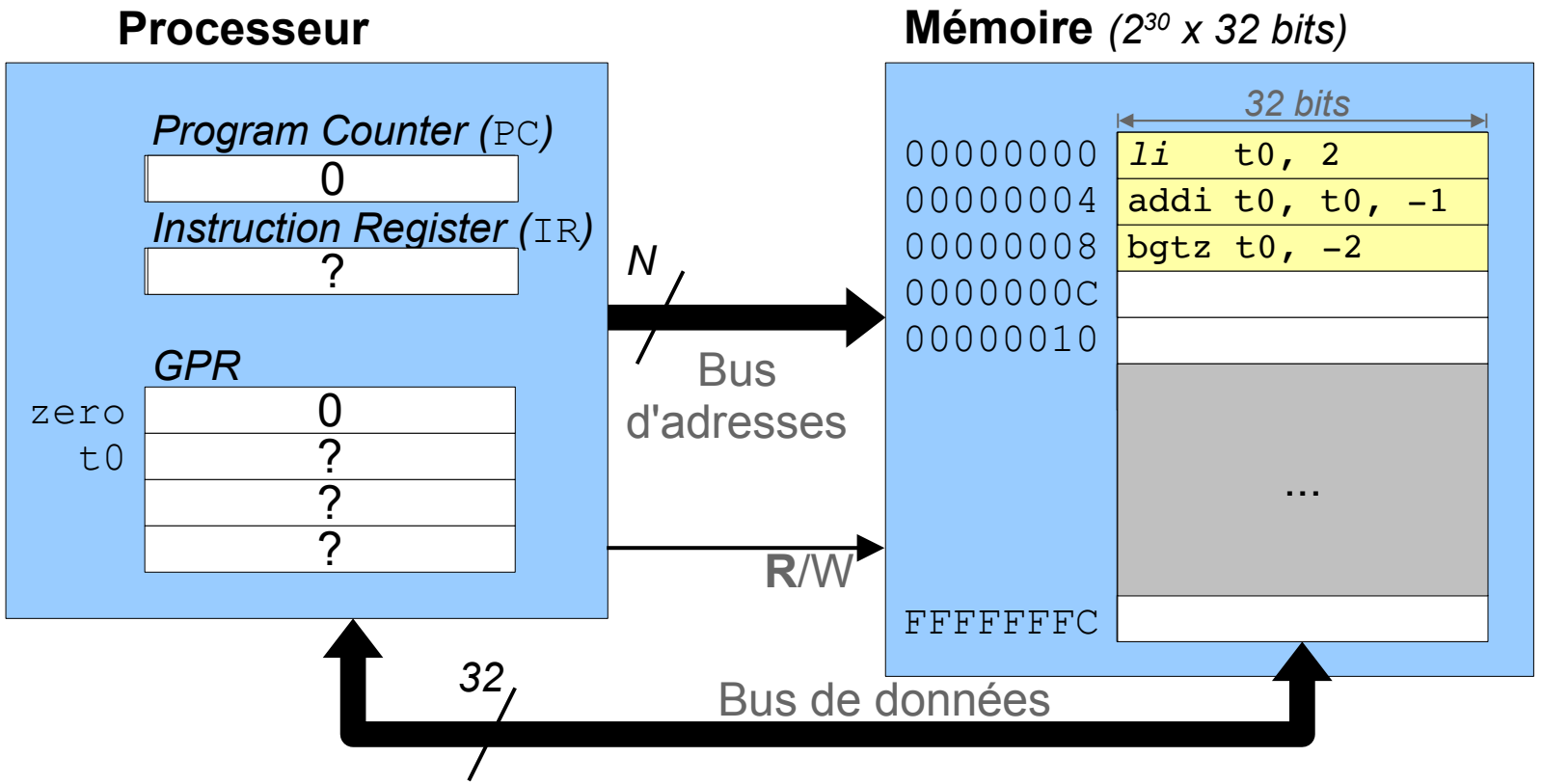
  - Hypothèses : l'étiquette `main` est situé à l'adresse 0 et l'étiquette `loop` à l'adresse 4.

- Comment l'assembleur détermine-t-il la valeur immédiate (offset) de l'instruction `BGTZ` ?

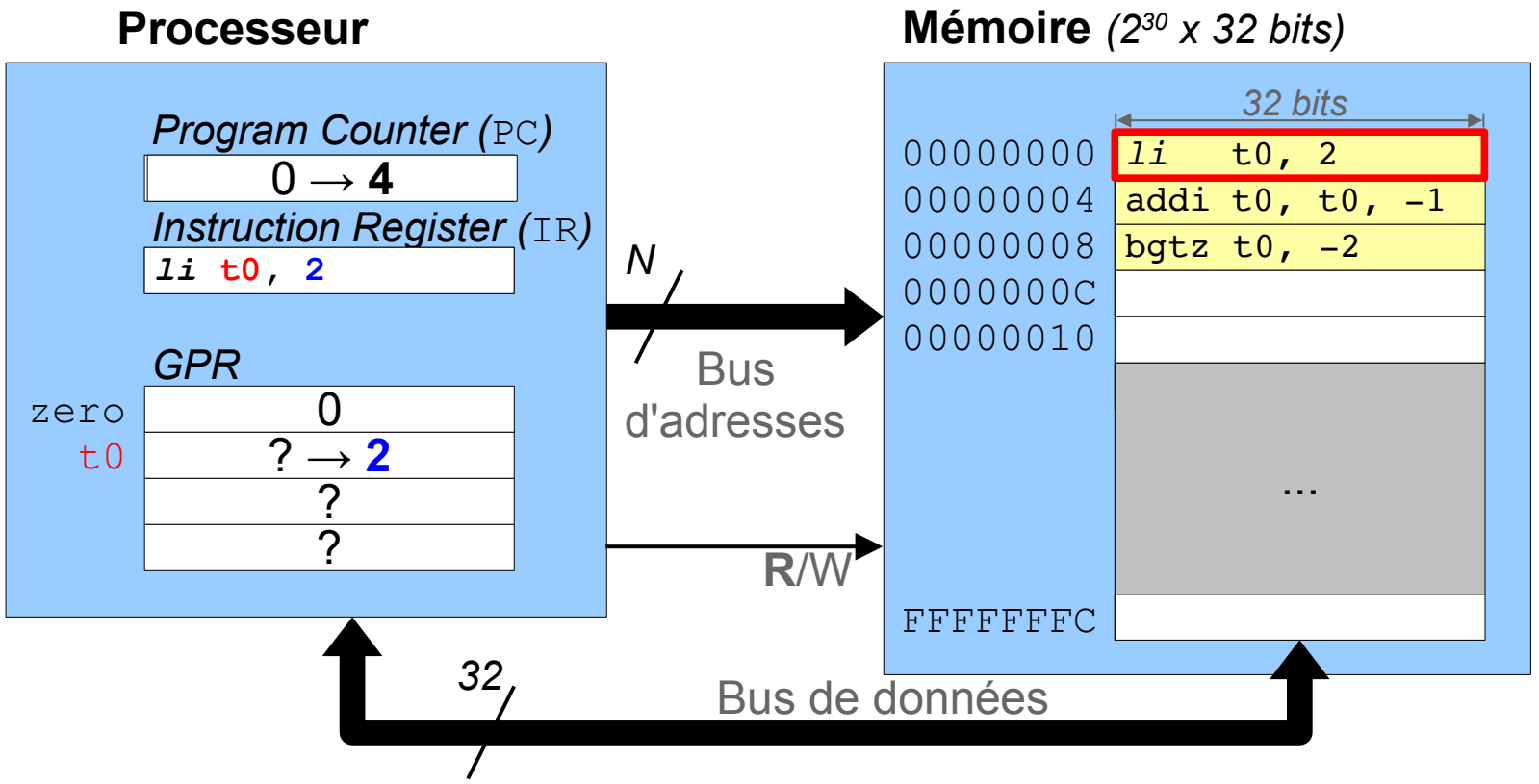
PC ← PC + 4 + *s-ext*(`offset` || 0^2)

4 = 1111111111111110 || 00
-2

Branchements conditionnels



Branchements conditionnels

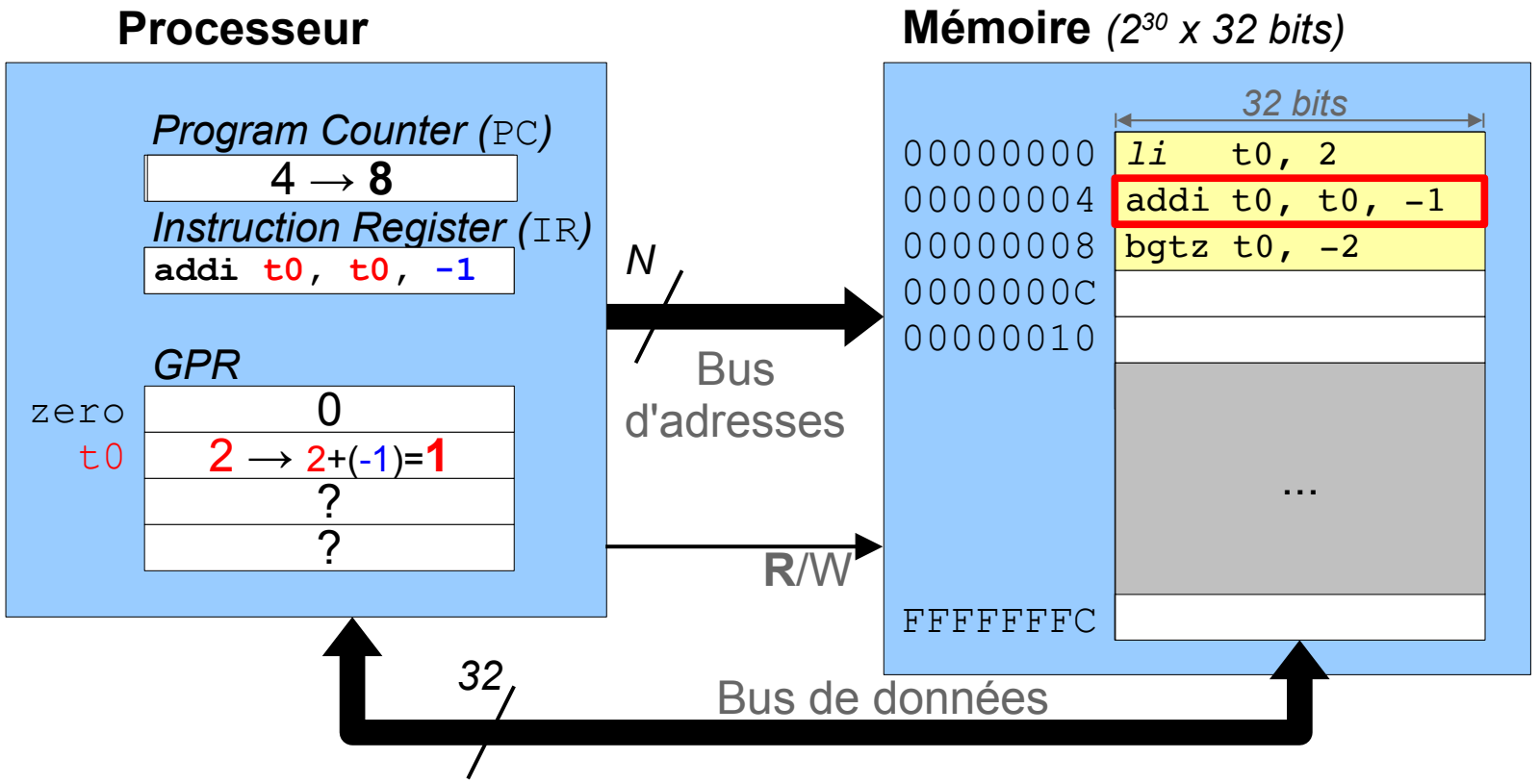


Load Immediate (pseudo-instruction)

syntaxe: `li rdest, imm`

description: GPR[rd] ← imm

Branchements conditionnels

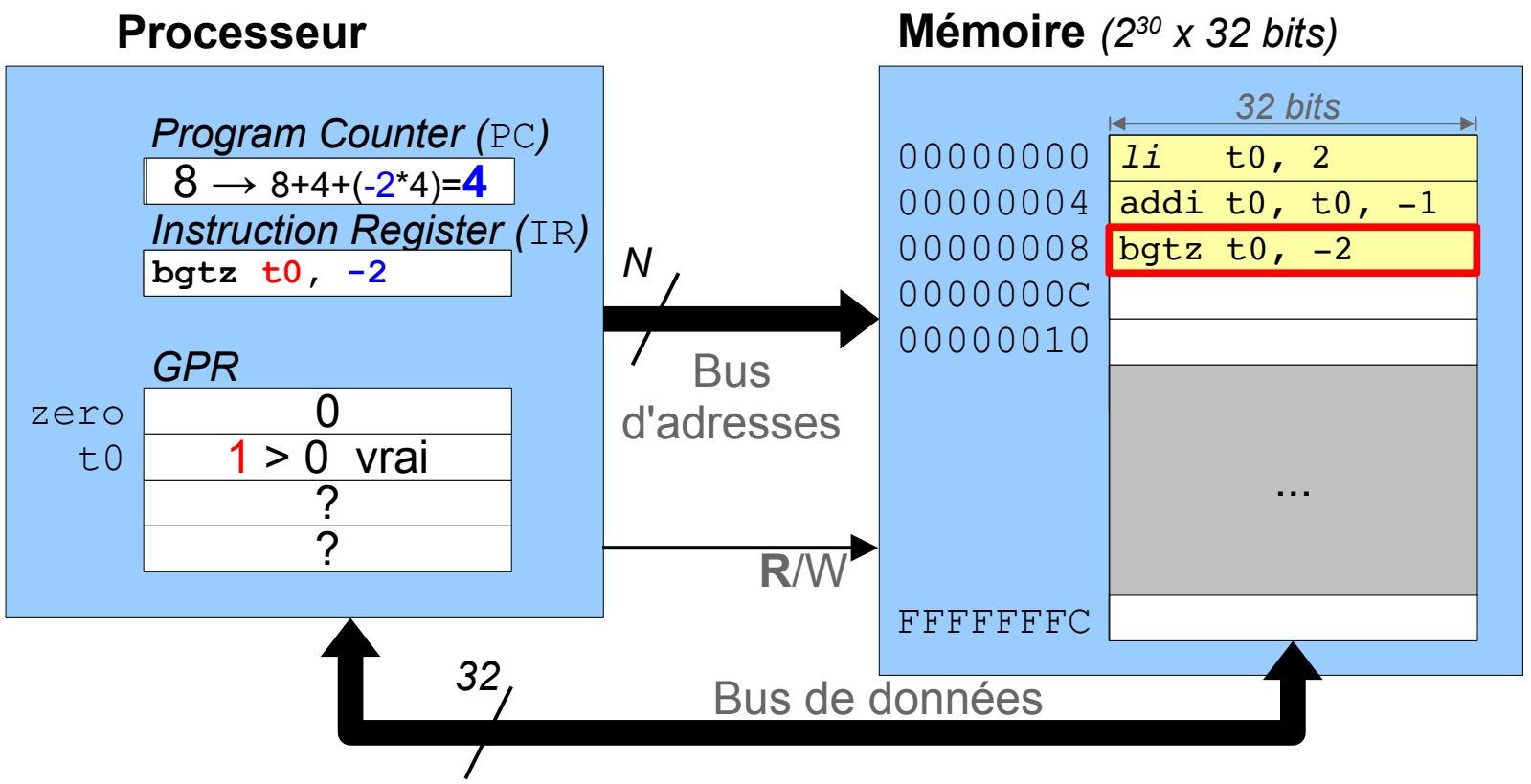


Add Immediate

syntaxe: **addi** rt, rs, imm

description: GPR[**rt**] ← GPR[rs] + *s-ext*(imm)

Branchements conditionnels



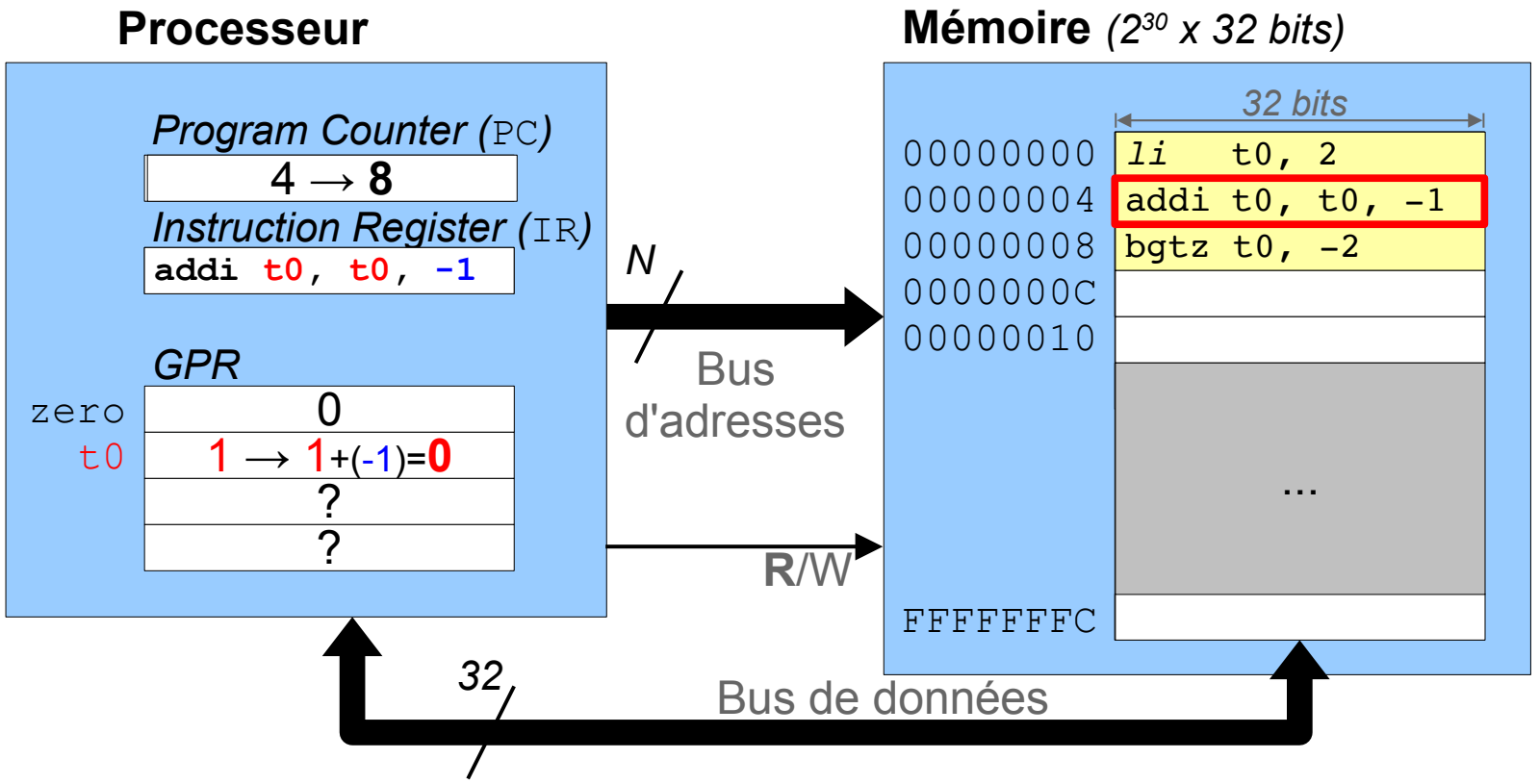
Branch on Greater Than Zero

syntaxe: **bgtz** rs, offset

description: **if** GPR[rs] > 0 **then**

PC ← PC + 4 + *s-ext*(offset || 0^2)

Branchements conditionnels

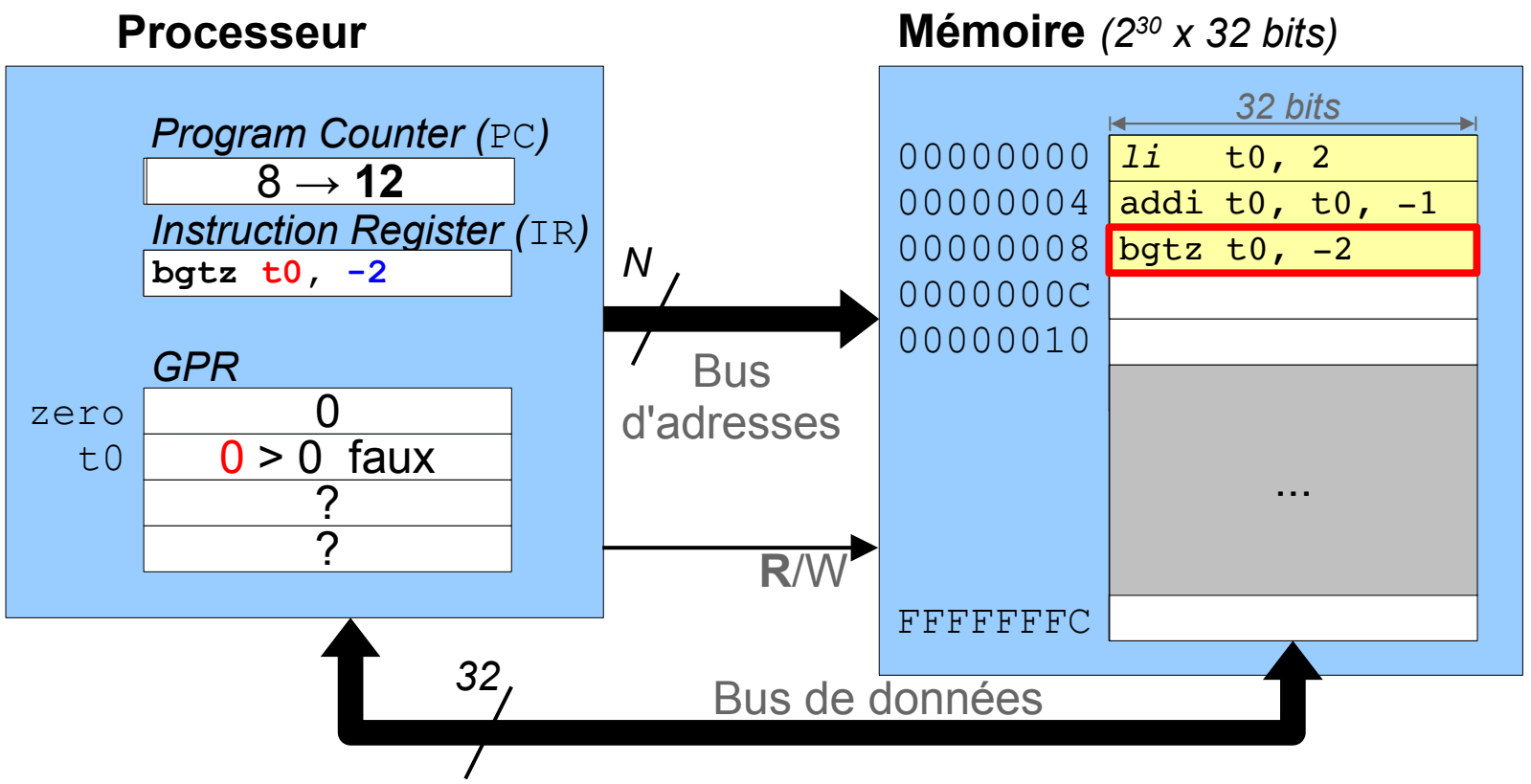


Add Immediate

syntaxe: **addi** rt, rs, imm

description: GPR[**rt**] ← GPR[rs] + *s-ext*(imm)

Branchements conditionnels



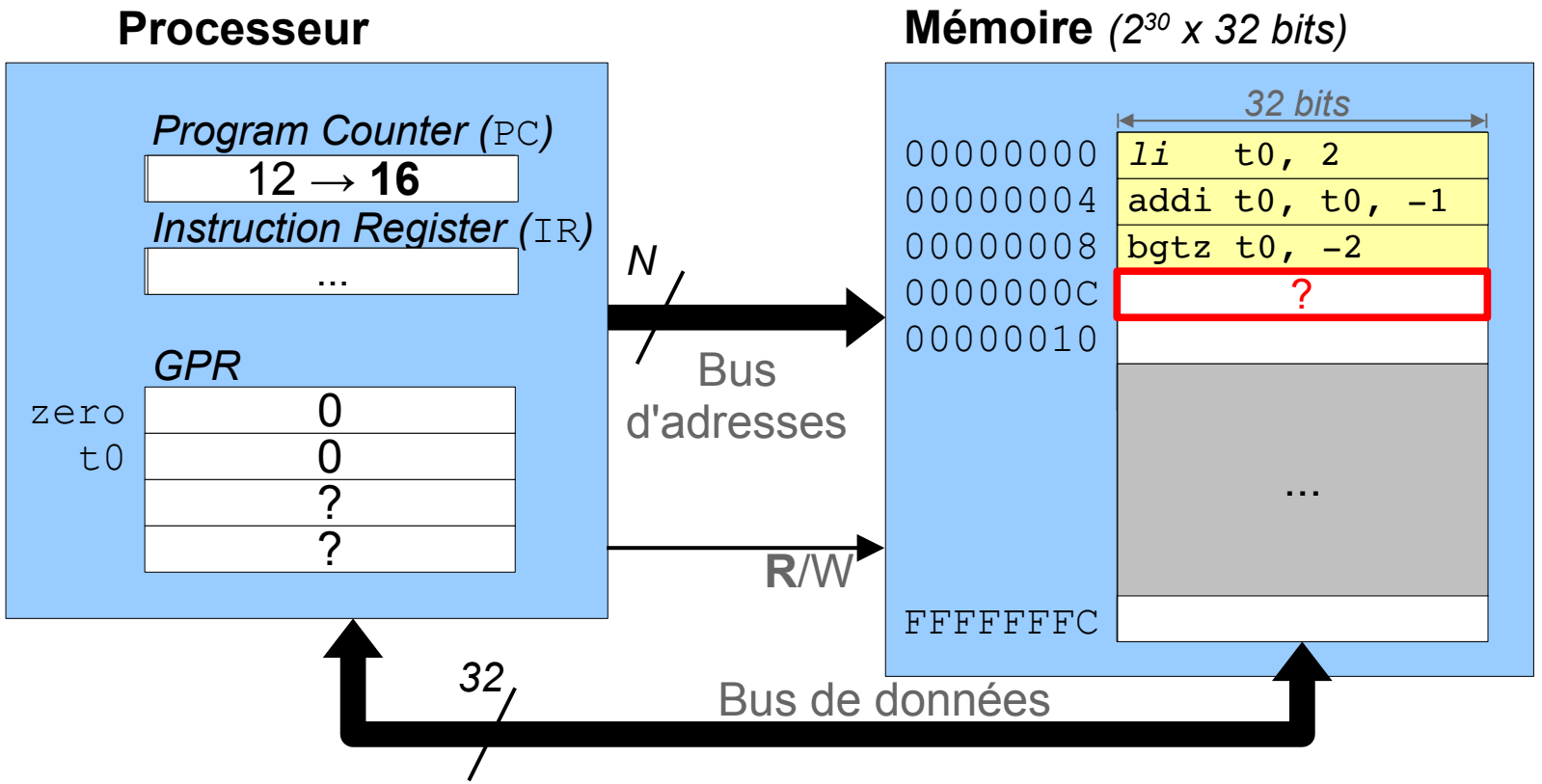
Branch on Greater Than Zero

syntaxe: **bgtz** rs, offset

description: **if** GPR[rs] > 0 **then**

PC ← PC + 4 + *s-ext*(offset || 0^2)

Branchements conditionnels

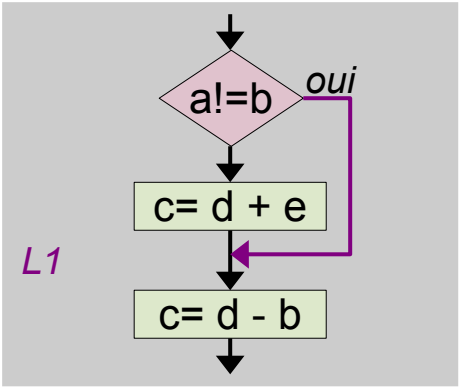


Branchements conditionnels

Instructions conditionnelles

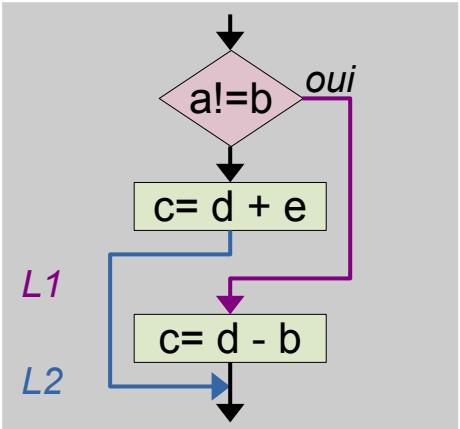
- Comment traduire les blocs d'instructions `if`, `if`/`else` et `switch` suivantes ?
  - Hypothèses : les variables `a` à `e` sont situées respectivement dans les registres `t0` à `t4`.

```
if (a == b)
  c= d + e;
c= d - b;
```




```
     bne    $t0, $t1, L1
     add    $t2, $t3, $t4
L1:
     sub    $t2, $t3, $t1
```

```
if (a == b)
  c= d + e;
else
  c= d - e;
```

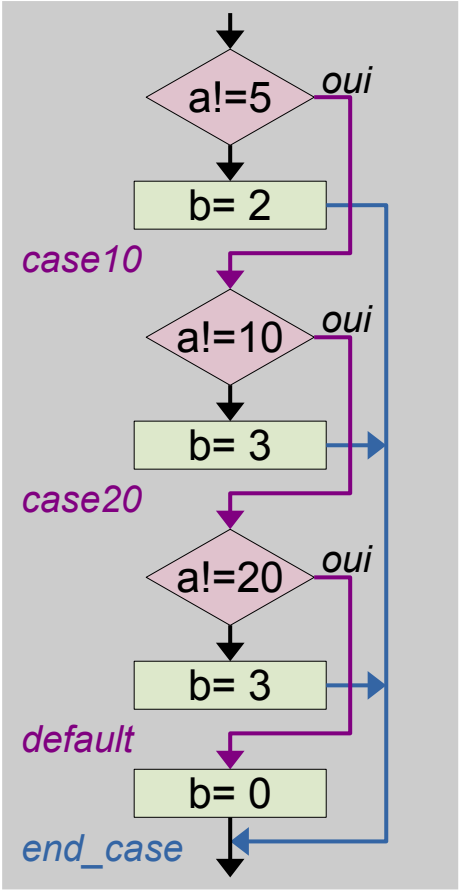



```
     bne    $t0, $t1, L1
     add    $t2, $t3, $t4
     j      L2
L1:
     sub    $t2, $t3, $t4
L2:
```

Branchements conditionnels

Instructions conditionnelles

```
switch (a) {
case 5 :
  b= 2;
  break;
case 10:
  b= 3;
  break;
case 20:
  b= 4;
  break;
default:
  b= 0;
}
```



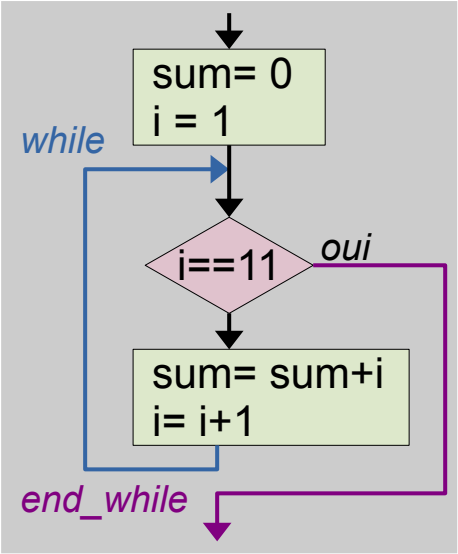

```
# t2 utilisé comme variable
# temporaire
case5:
     li      $t2, 5
     bne     $t0, $t2, case10
     li      $t1, 2
     j       end_case
case10:
     li      $t2, 10
     bne     $t0, $t2, case20
     li      $t1, 3
     j       end_case
case20:
     li      $t2, 20
     bne     $t0, $t2, default
     li      $t1, 4
     j       end_case
default:
     li      $t1, 0
end_case:
```

Branchements conditionnels

Boucles

- Comment traduire les boucles `for` et `while` suivantes ?
  - Hypothèses : les variables `i` et `sum` sont situées respectivement dans les registres `t0` et `t1`.

```
sum= 0;
i= 1;

while (i != 11) {
   sum= sum + i;
   i= i+1;
}
```

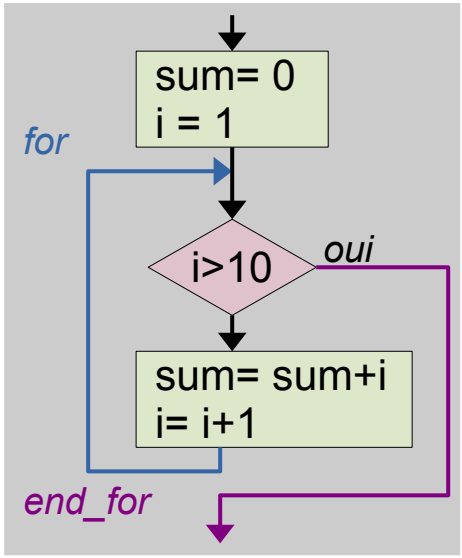



```
     li      $t0, 1    # i
     li      $t1, 0    # sum
     # $t2 variable temporaire
     # limite pour i
     li      $t2, 11

while:
     beq     $t0, $t2, end_while
     add     $t1, $t1, $t0
     addi    $t0, $t0, 1
     j       while
end_while:
```

Branchements conditionnels

Boucles

```
sum= 0;

for (i= 1; i <= 10; i= i+1)
  sum= sum + i;
```




```
    li      $t0, 1      # i
    li      $t1, 0      # sum
    # $t2 variable temporaire
    # nombre d'itérations
    li      $t2, 10

for:
    sub     $at, $t2, $t0
    bltz    $at, end_for
    addi    $t1, $t1, $t0
    addi    $t0, $t0, 1
    j       for
end_for:
```

Branchements conditionnels

Exercice

- Comment traduire la boucle `for` suivante ?
  - les variables `i`, `sum`, `pow` et `num` sont situées respectivement dans les registres `$t0` à `$t3`.

```
sum= 0;
pow= 1;

for (i= 0; i < 5; i++) {
  sum= sum + pow;
  pow= pow * 2;
}
```

Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Appel de fonctions

Introduction

- Une **fonction** (ou **procédure**, **méthode**) est un sous-ensemble d'instructions du programme qui peuvent être **appelées** à partir de plusieurs endroits. Le découpage d'un programme en fonctions offre plusieurs avantages
  - **modularité du code source** : fonction ré-utilisable
  - **lisibilité** : fonction plus petite et donc plus facilement compréhensible; responsabilité limitée
  - **compacité du programme** : les instructions d'une fonction n'existent qu'une fois en mémoire

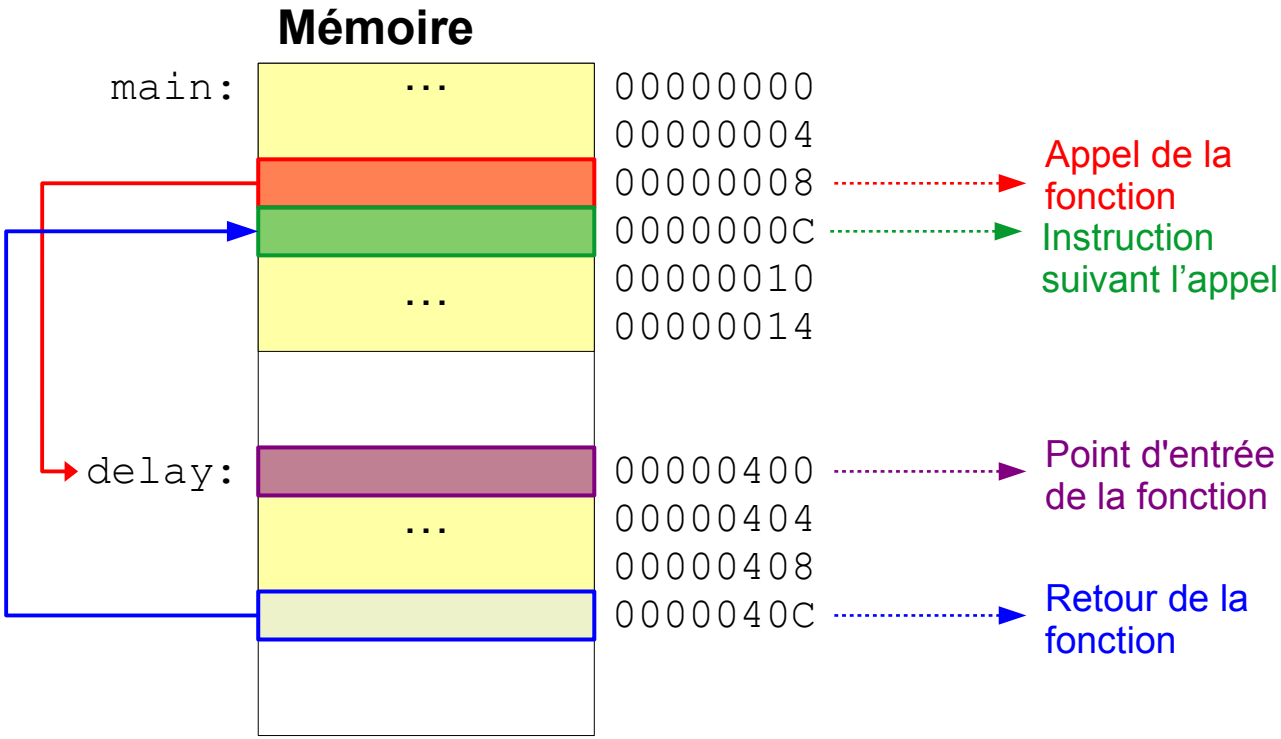
- Cette section étudie les ressources mises à disposition du programmeur par l'architecture MIPS pour l'**appel de fonctions** (*function call*).
  - Comment effectuer l'appel de fonction ?
  - Comment passer des paramètres et retourner des résultats ?
  - Comment stocker les variables locales ?
  - Comment éviter les effets de bord

Appel de fonctions

Principe

- L'appel d'une fonction revient à effectuer un *saut inconditionnel* vers la première instruction de la fonction. Cette dernière est appelée **point d'entrée** (*entry point*) de la fonction.
- Une fois la fonction terminée, il s'agit de retourner à l'instruction qui suit l'instruction d'appel.

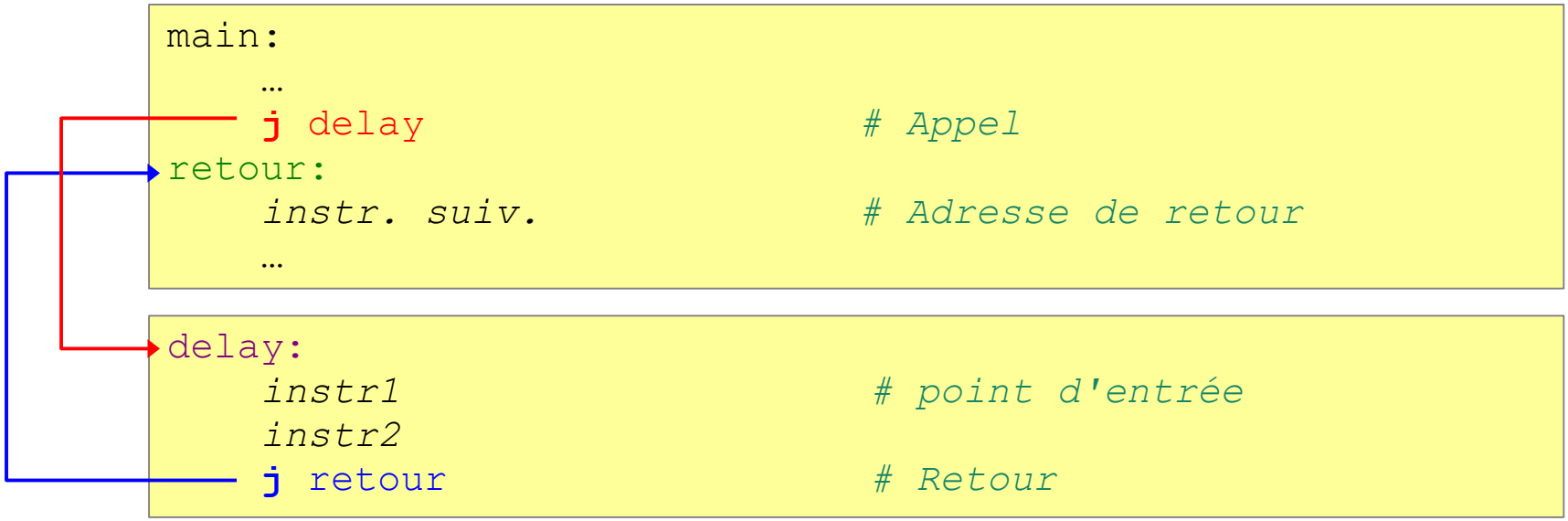
```
void delay()
{
  /* ... */
}

void main()
{
  /* ... */
  delay();
  /* ... */
}
```



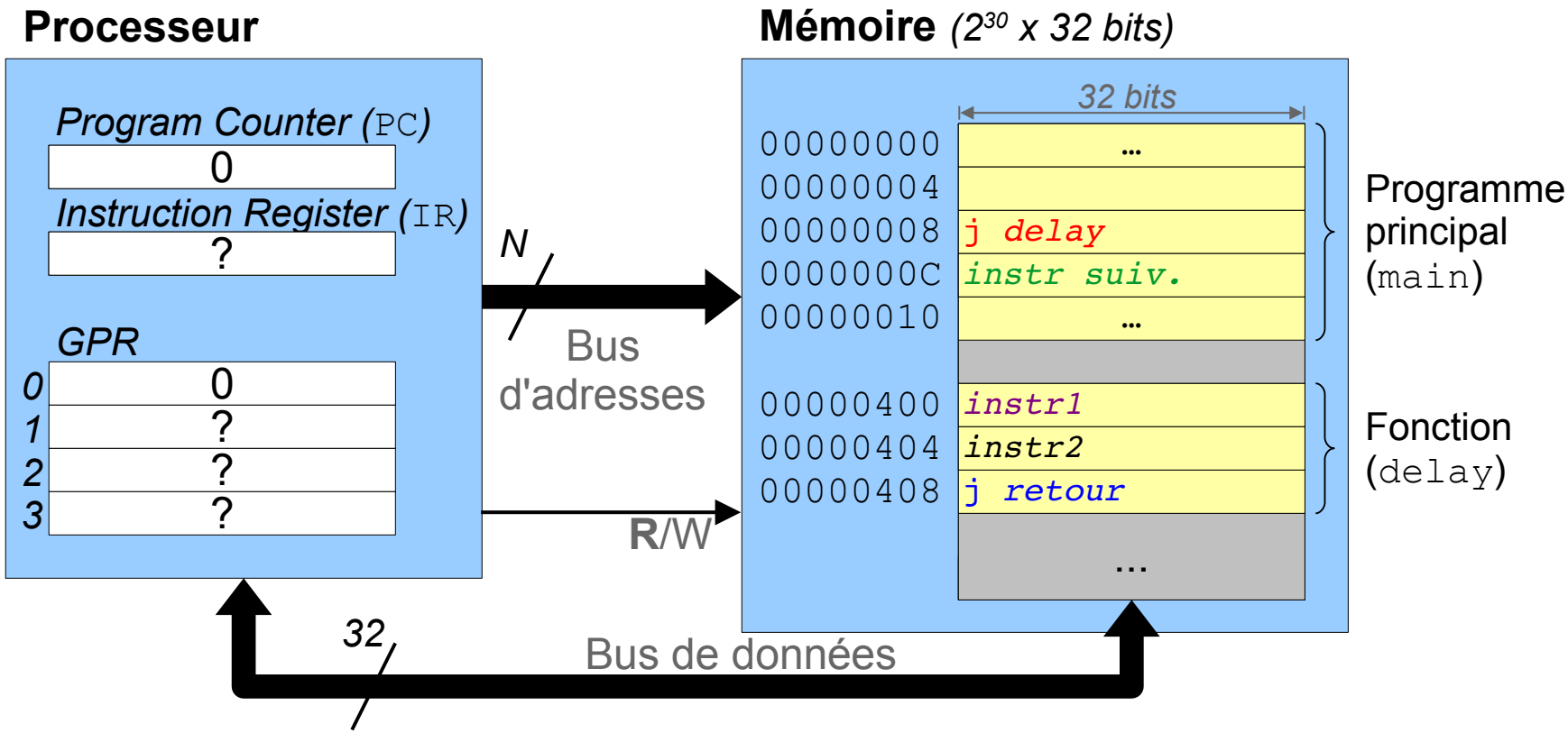
Appel de fonctions

Tentative n°1 - Adressage direct

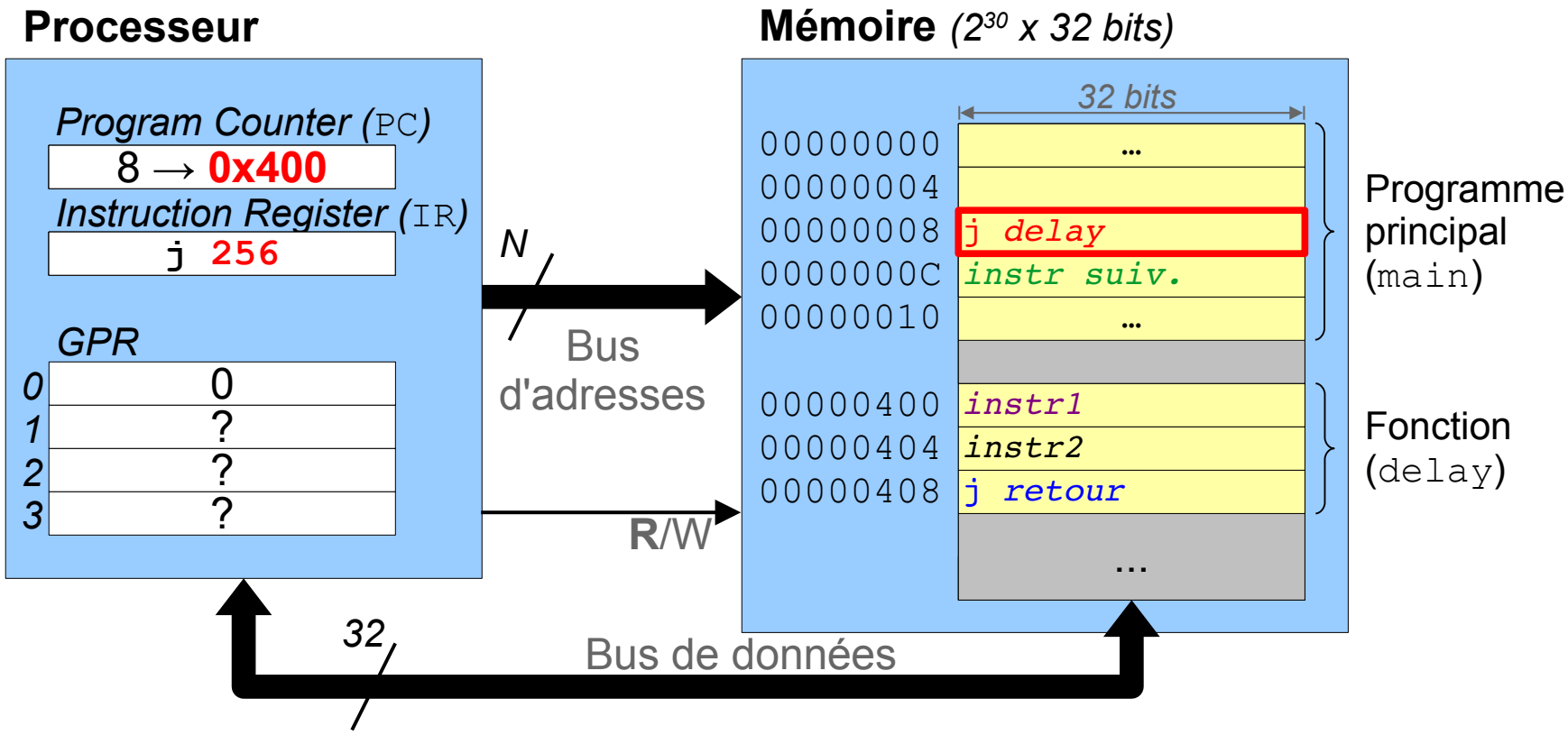
- Une première possibilité pour appeler une procédure serait d'utiliser des instructions de branchement inconditionnel avec un adressage direct.
  - Branchement vers le point d'entrée : instruction *Jump* (`J`).
  - Exécution des instructions de la fonction
  - Retour vers l'instruction qui suit l'appel : instruction *Jump* (`J`).



Appel de fonctions



Appel de fonctions

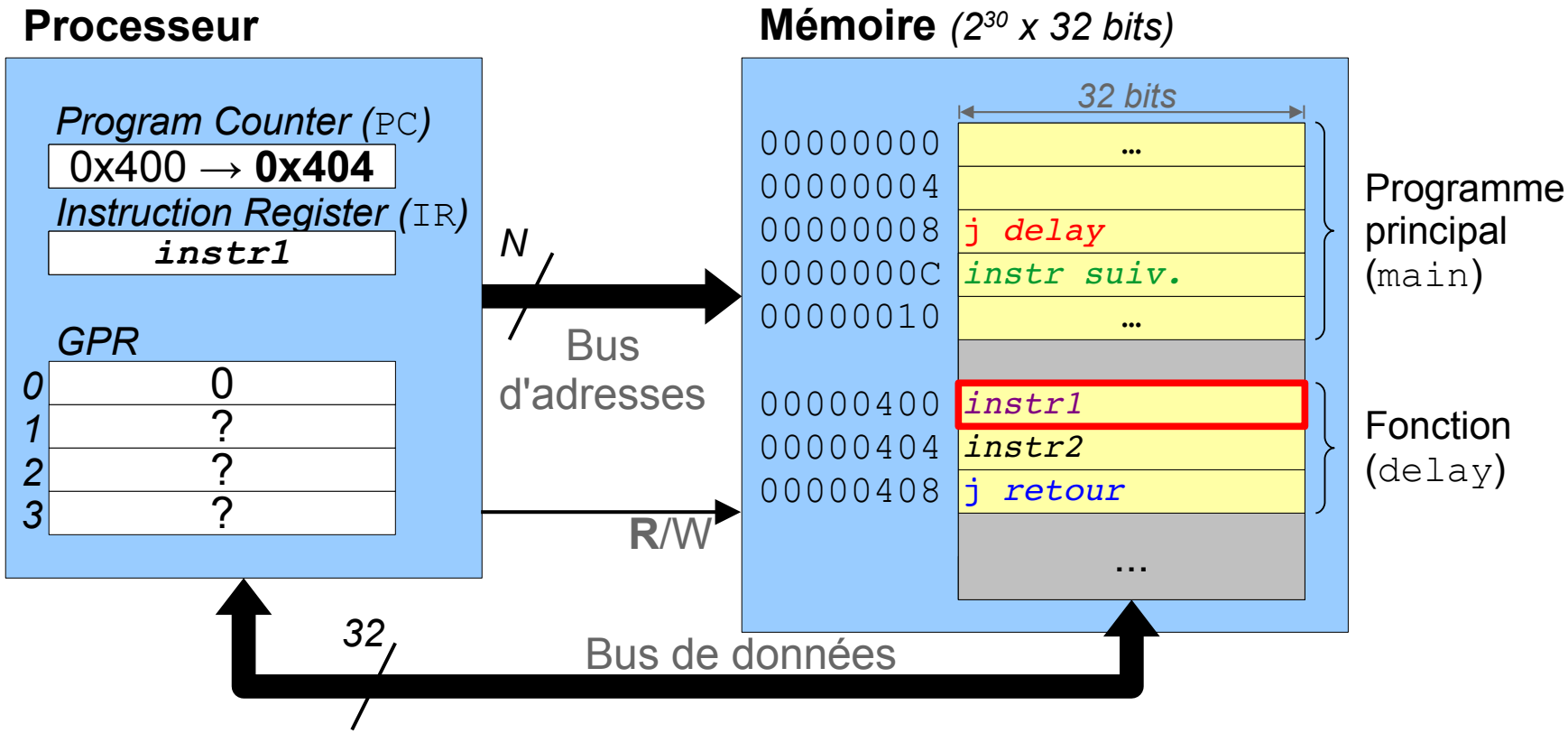


Jump

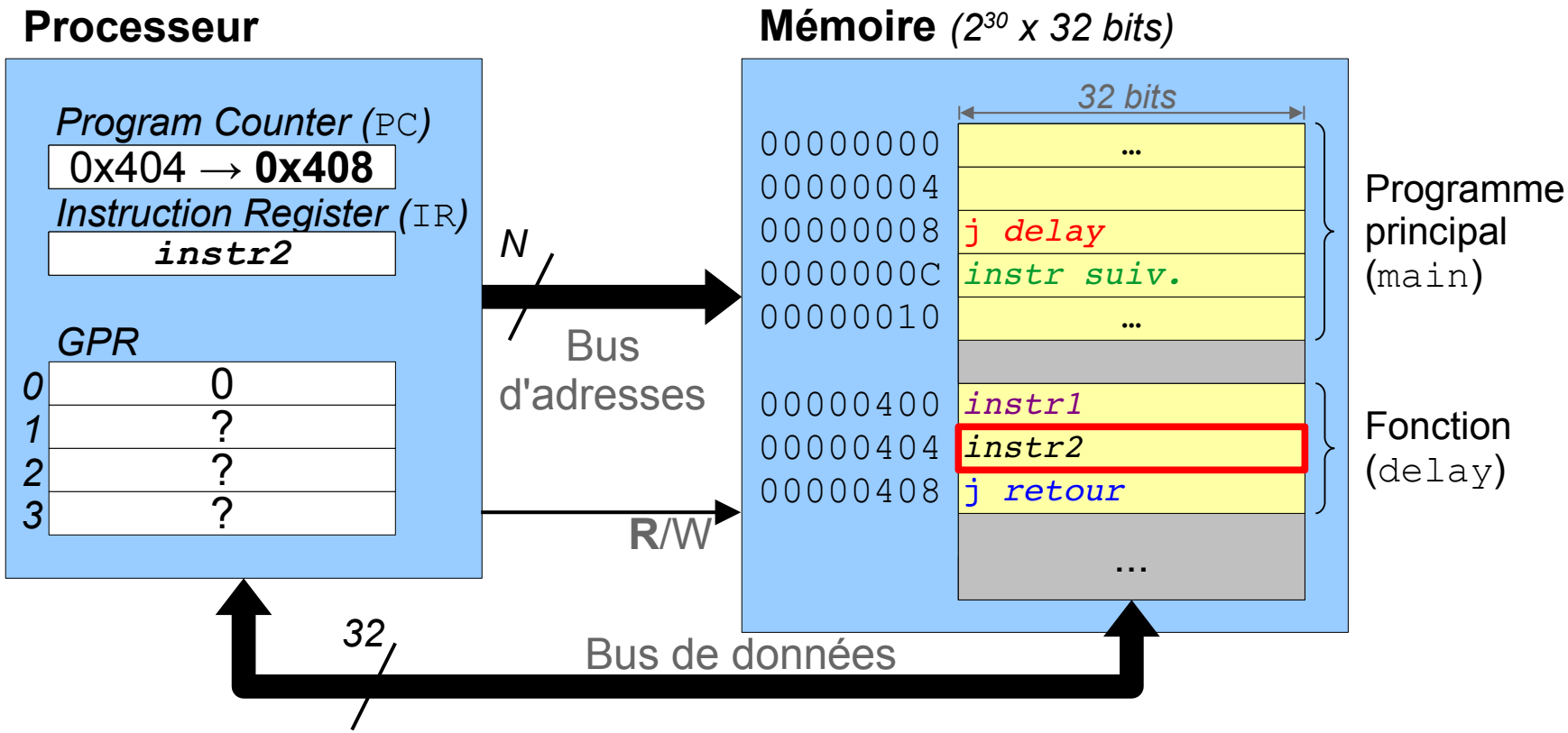
syntaxe: `j target`

description: $PC \leftarrow PC_{31..28} \,||\, \texttt{target} \,||\, 0^2$

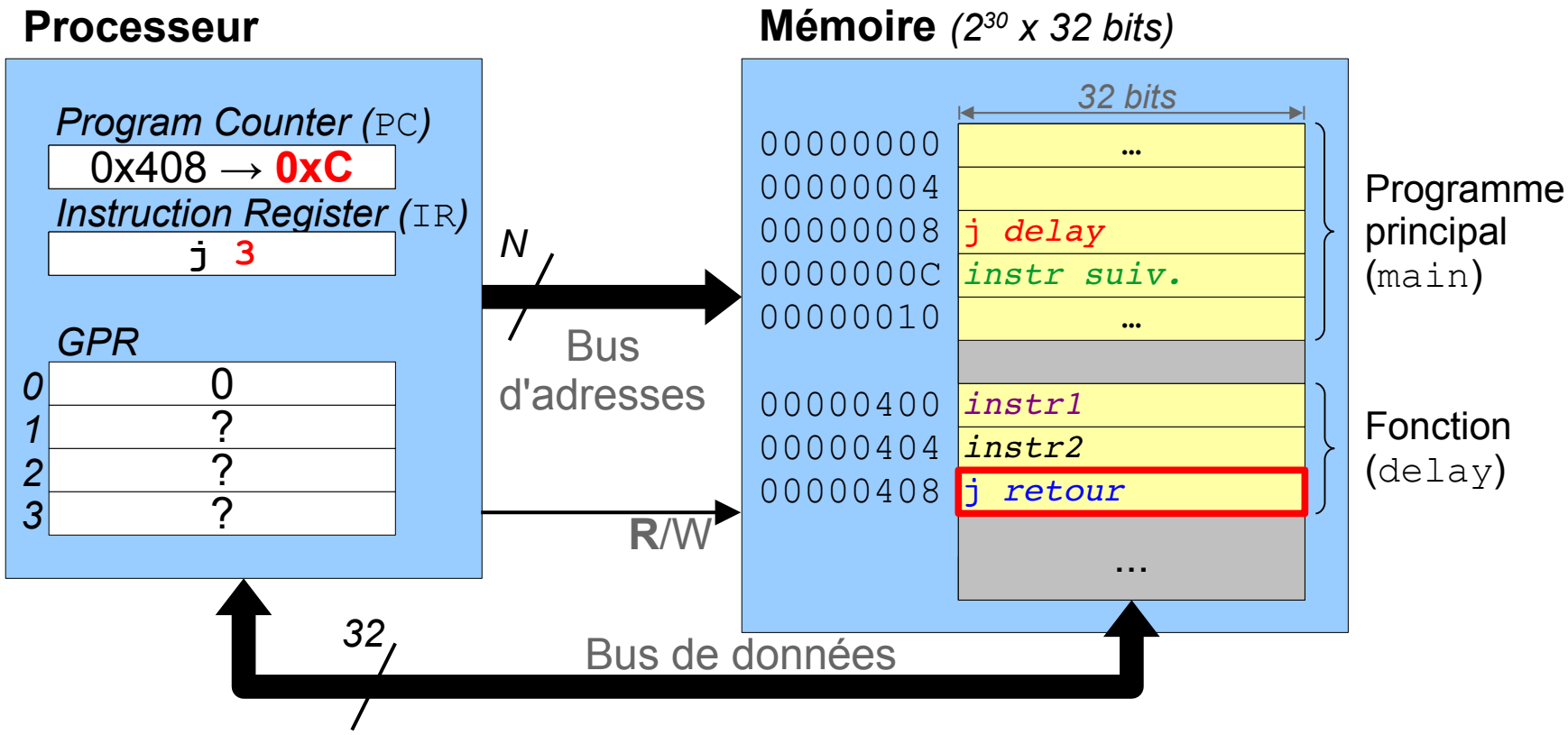
Appel de fonctions



Appel de fonctions



Appel de fonctions

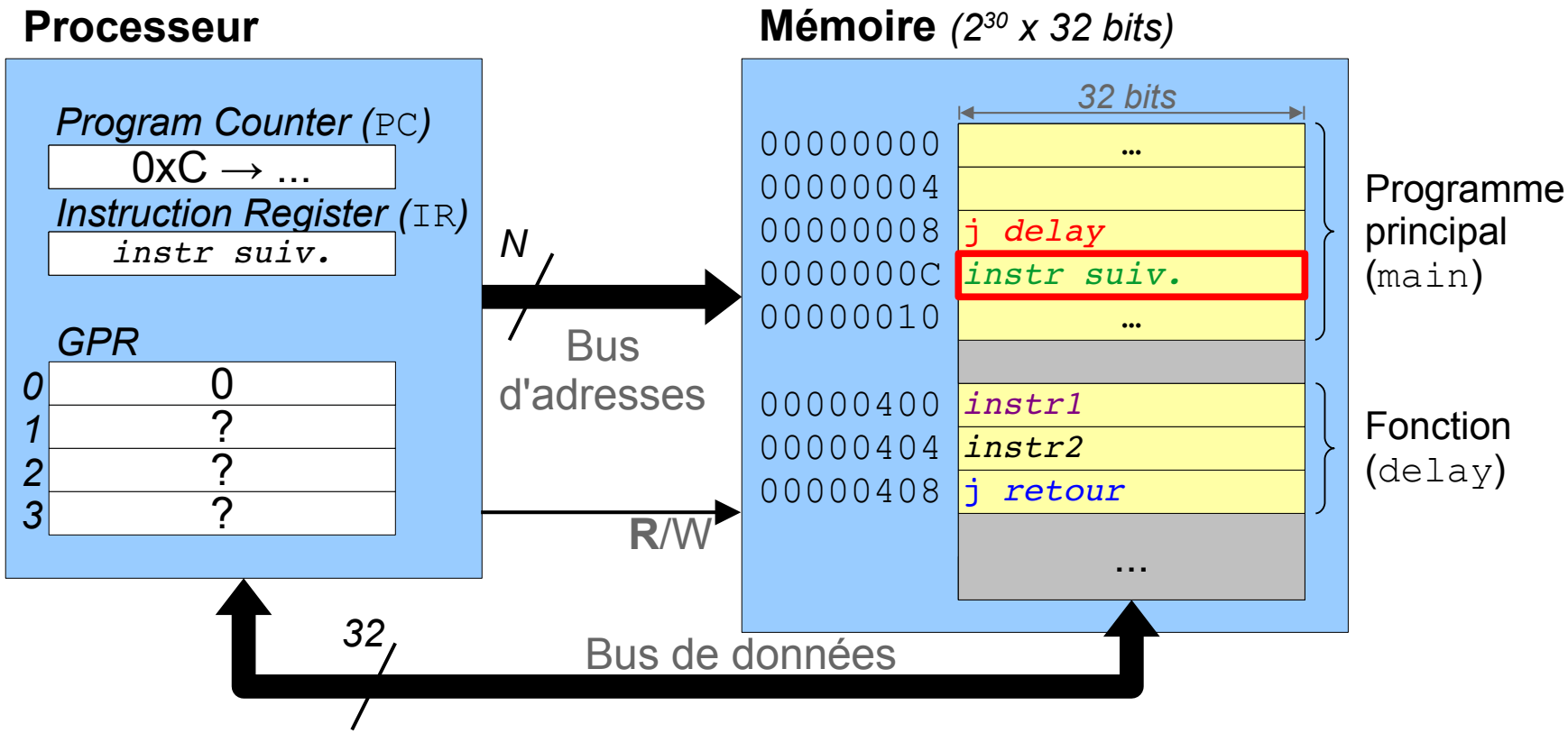


Jump

syntaxe: `j target`

description: $PC \leftarrow PC_{31..28} \,||\, \texttt{target} \,||\, 0^2$

Appel de fonctions



Appel de fonctions

Tentative n°2 – Adressage indirect

- Avec l'adressage direct, l'adresse de retour de la fonction est fixe et encodée dans l'instruction de saut. La fonction ne peut donc être appelée que d'un seul endroit dans le programme.
- Il faudrait disposer d'un moyen de "se rappeler" l'endroit duquel l'appel a été effectué pour pouvoir retourner *juste après* cet endroit (adresse de retour).

- L'instruction *Jump And Link* (`JAL`) est une variante de l'instruction *Jump* (`J`) qui permet de sauvegarder l'adresse de retour dans le registre d'index 31 appelé `ra` (*return address*).

Instruction	Description	OpCode	Funct
Jump And Link **JAL target**	GPR[31] ← PC + 4 PC ← $PC_{31..28}$ || target || 0^2	000011	N/A

- Le retour à l'appelant est réalisé avec l'instruction *Jump Register* (`JR`), en utilisant le registre `ra`.

Appel de fonctions

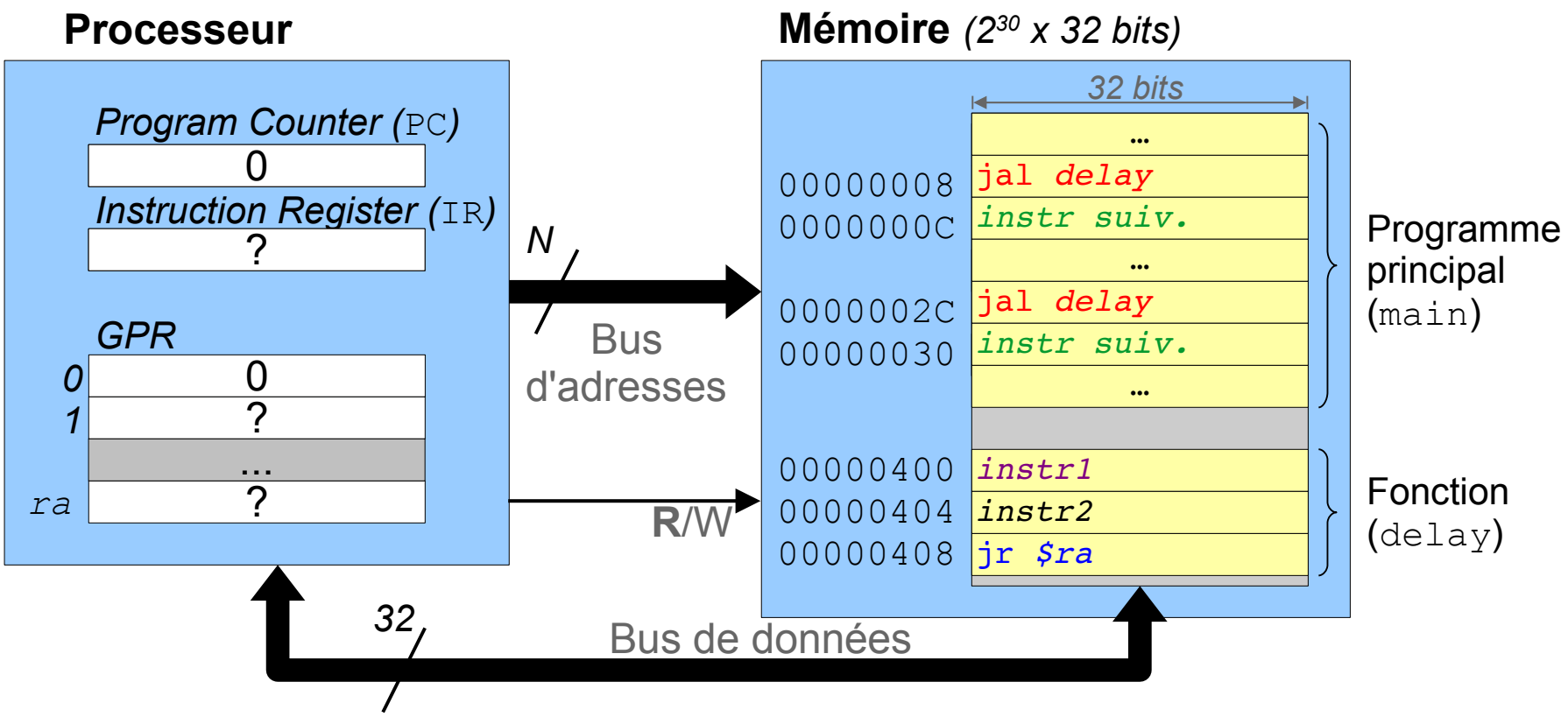
Tentative n°2 – Adressage indirect

- L'exemple suivant illustre l'utilisation de l'instruction *Jump And Link* (`JAL`) pour effectuer un appel de fonction.
- Le retour de la fonction est effectué avec *Jump Register* (`JR`) à partir du registre `ra`.

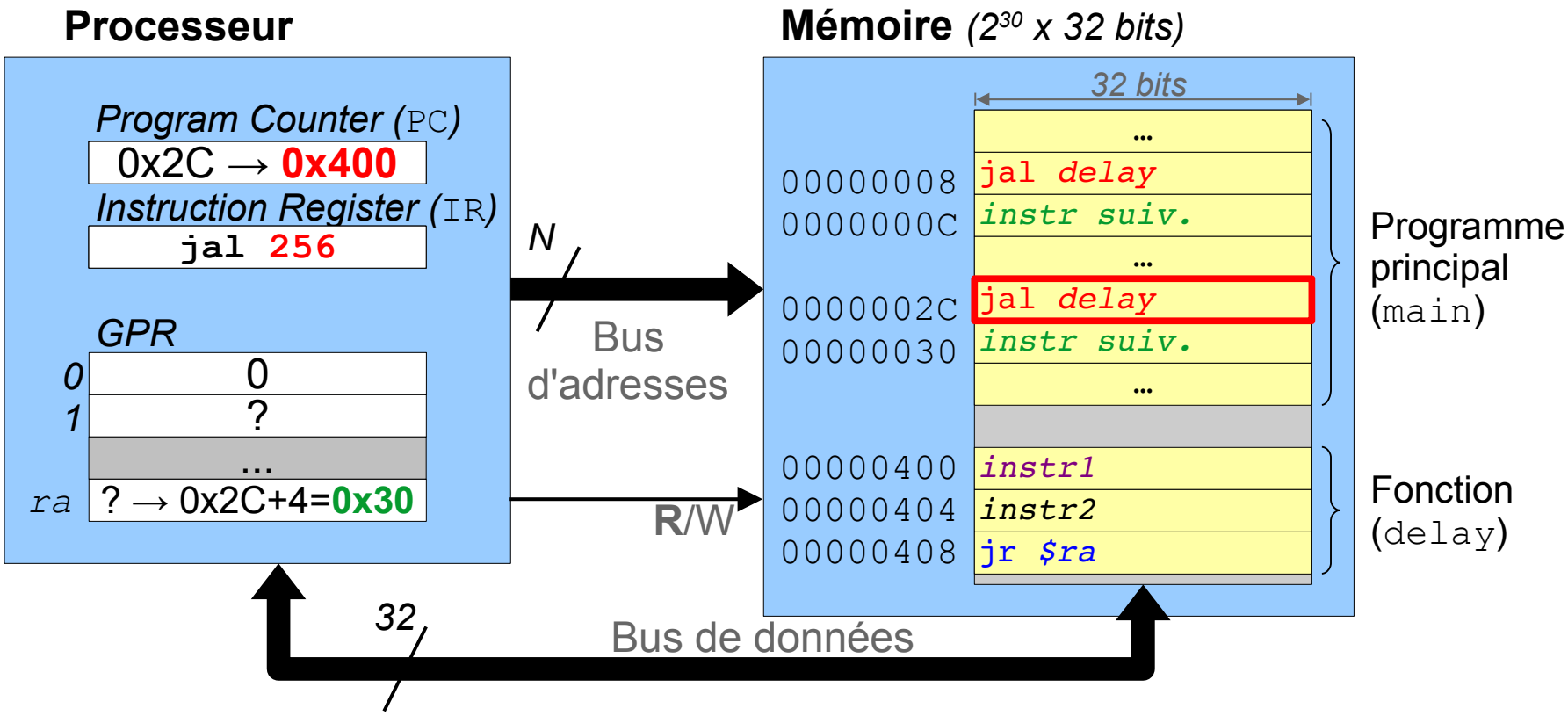
```
main:
    …
    jal delay                  # 1er Appel
    instr. suiv.               # Adresse de retour
    …
    jal delay                  # 2ème Appel
    instr. suiv.               # Adresse de retour
    …
```

```
delay:
    instr1                     # point d'entrée
    instr2
    jr $ra                     # Retour
```

Appel de fonctions



Appel de fonctions



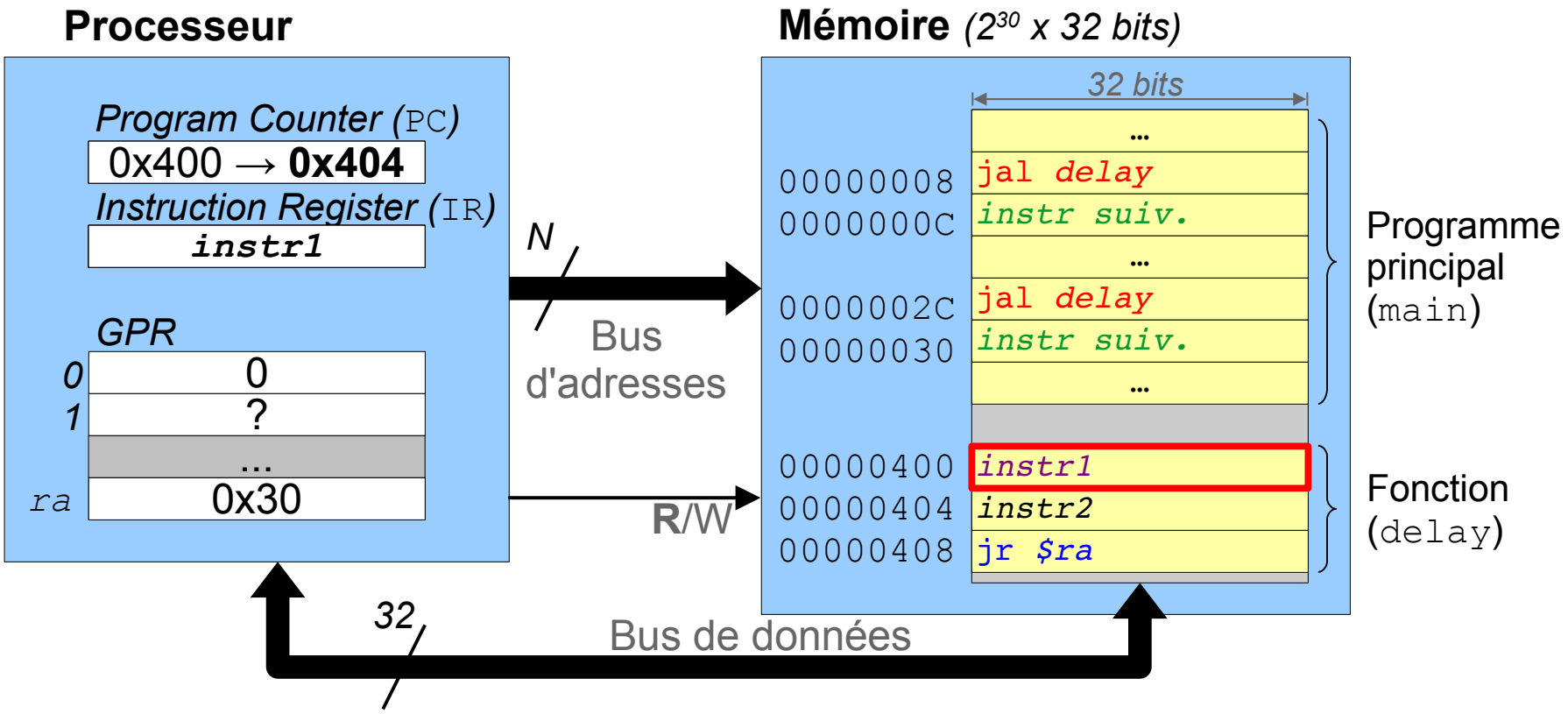
Jump And Link

syntaxe: `jal target`

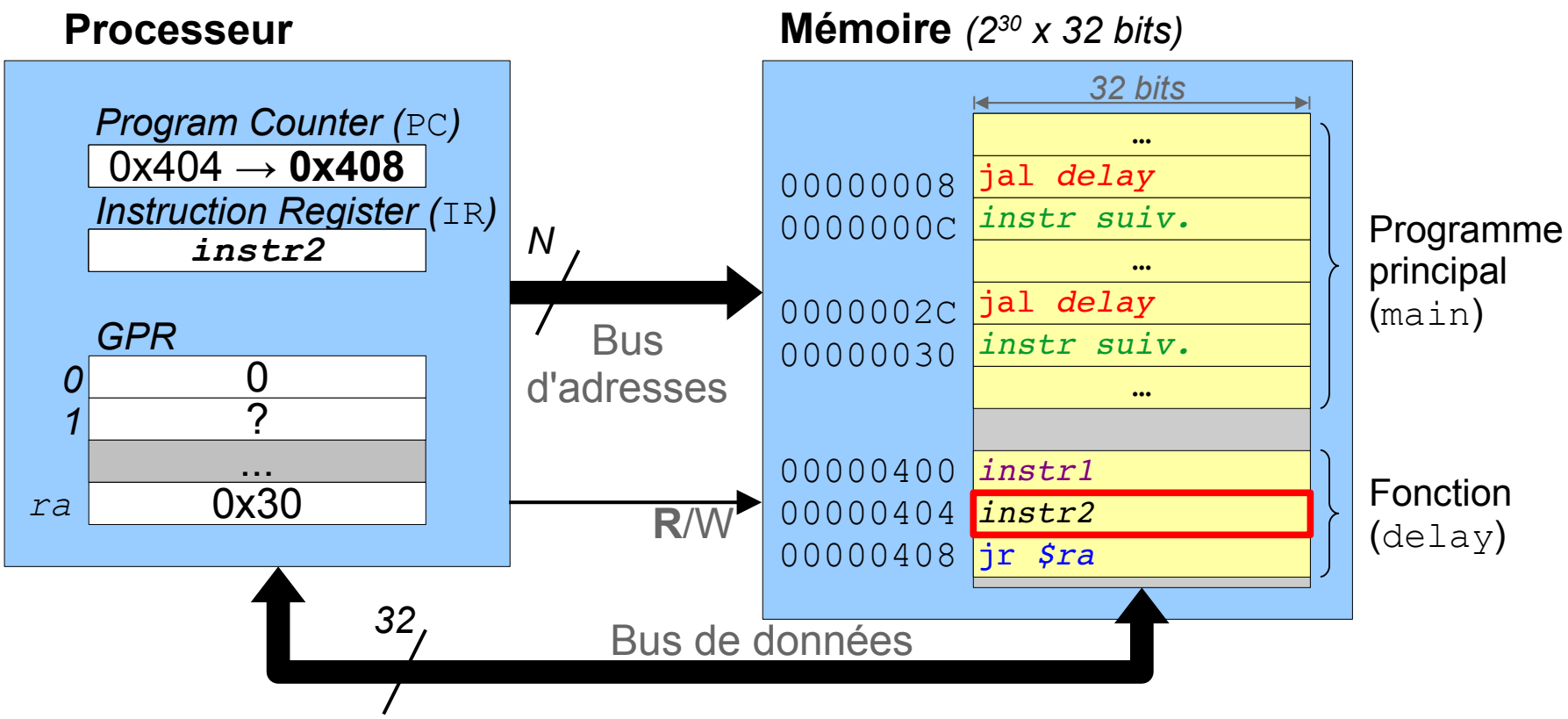
description: GPR[31] ← PC + 4

PC ← $PC_{31..28}$ || target || 0^2

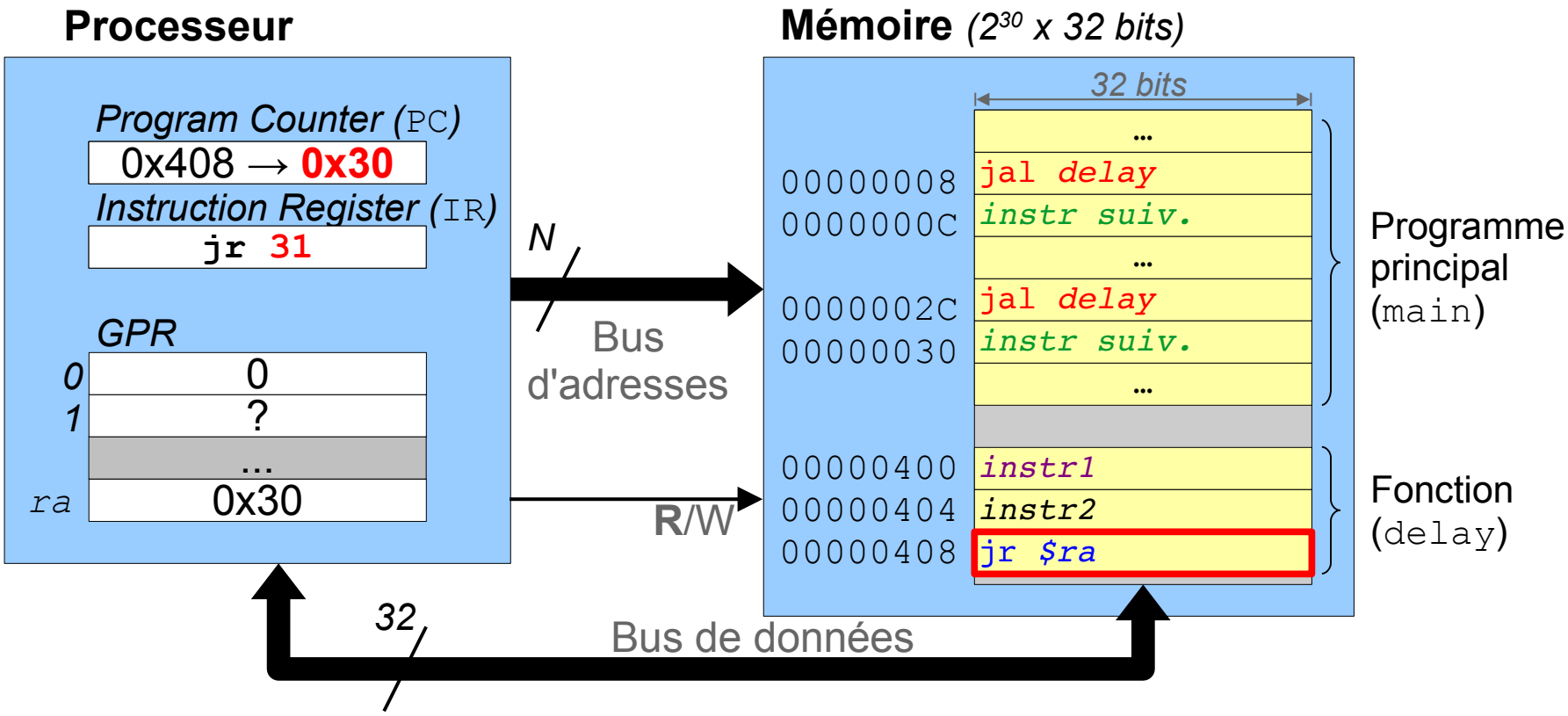
Appel de fonctions



Appel de fonctions



Appel de fonctions

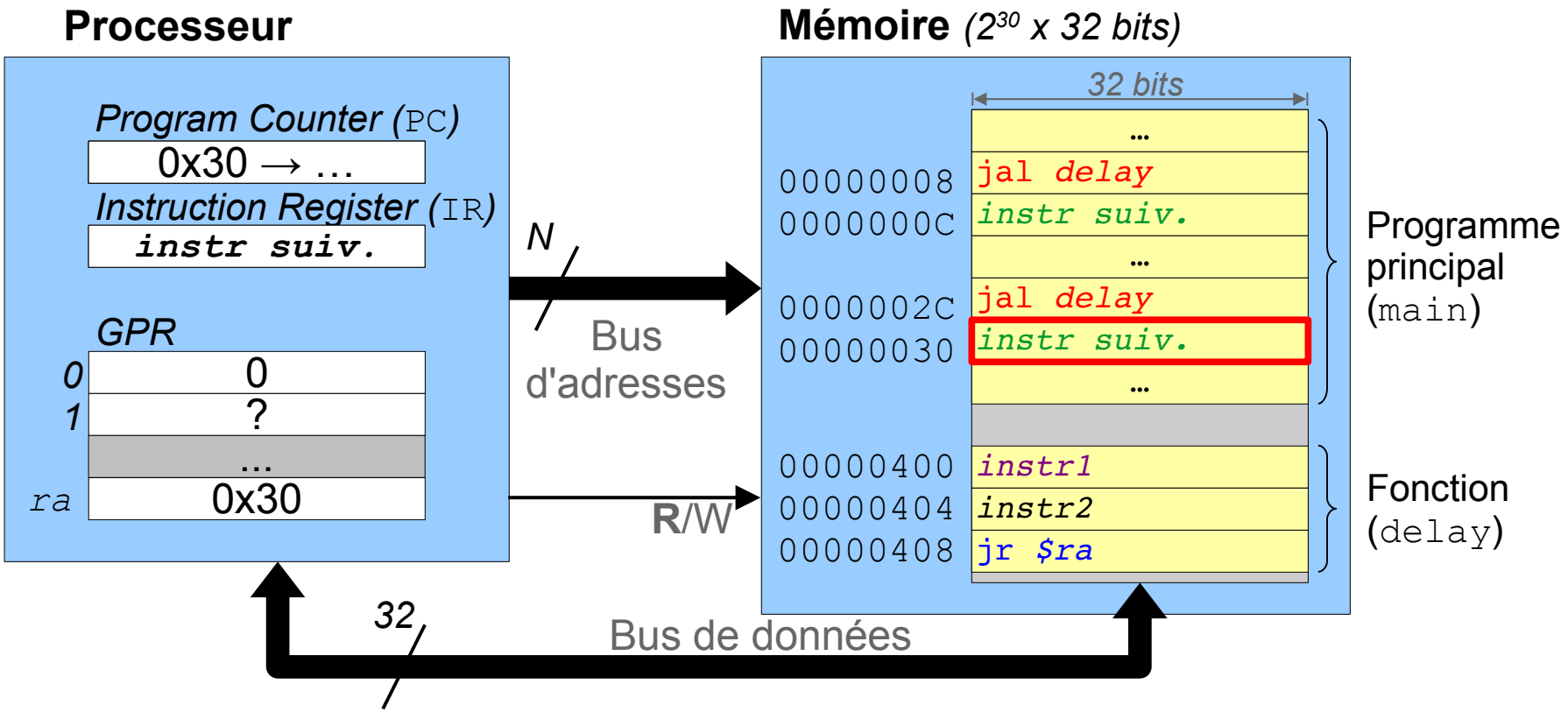


Jump Register

syntaxe: `jr rs`

description: PC ← GPR[rs]

Appel de fonctions

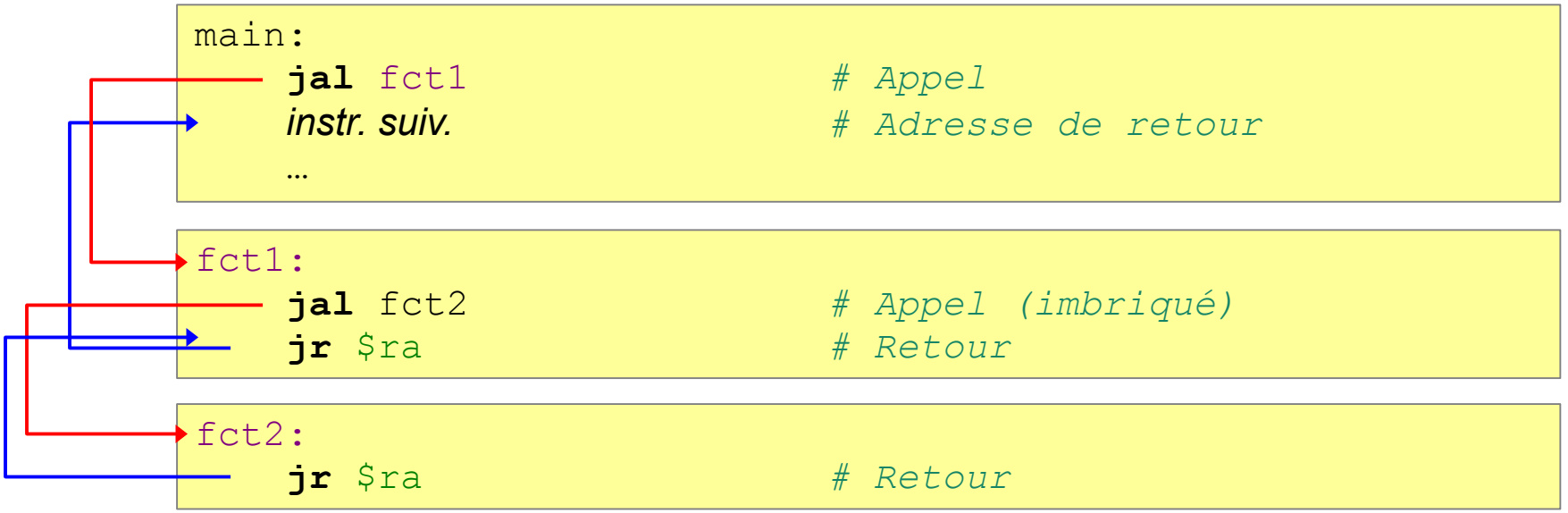


Appel de fonctions

Appels imbriqués/récursifs

- L'adressage indirect et la sauvegarde de l'adresse de retour dans un registre ne fonctionne pas avec les appels imbriqués ou récursifs. En effet, l'adresse sauvegardée dans le registre `ra` lors d'un premier appel sera remplacée par l'adresse de retour du second appel imbriqué.

- Exemple :



Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Appel de fonctions

Tentative n°3 – Sauvegarde de l'adresse de retour

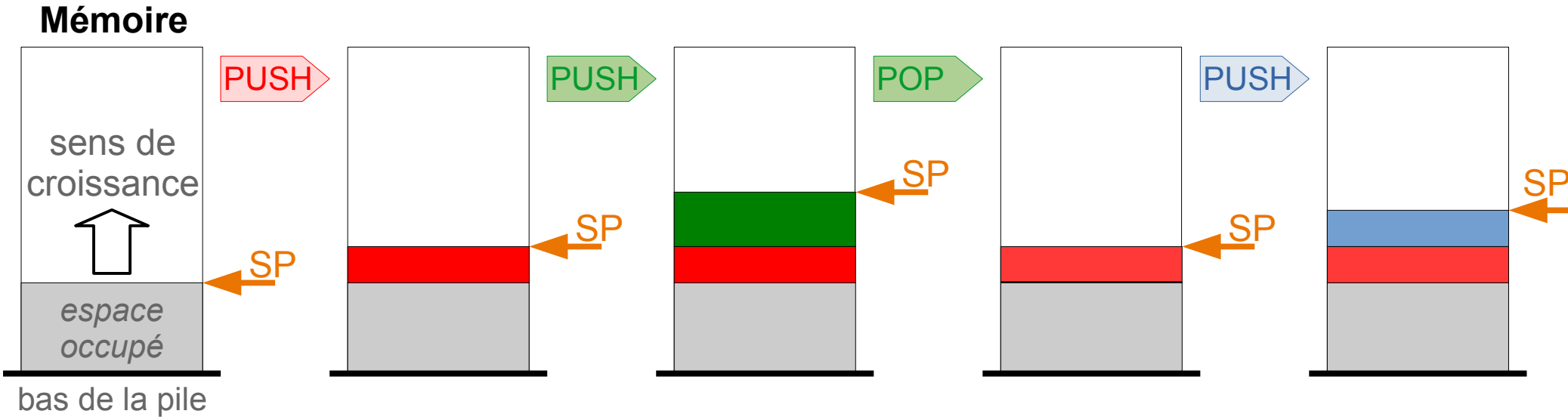
- Lorsqu'une procédure doit elle-même effectuer un appel à une autre procédure, l'adresse de retour qui se trouve actuellement dans le registre `ra` doit être sauvegardée ailleurs.

- Où sauvegarder le contenu de `ra` ?
  - dans un autre registre ? (problème, nombre limité de registres)
  - en mémoire ?

Appel de fonctions

Pile d'appels

- Une **pile** (*stack*) est une structure de données gérée selon la politique d'accès **dernier entré, premier sorti** (*last-in-first-out* ou **LIFO**).
- L'adresse du bas de la pile est fixe. L'adresse du sommet de la pile est variable et désigné par un **pointeur de pile** (*stack pointer*), typiquement gardé dans un registre.
- Deux opérations sont réalisées sur la pile
  - **PUSH** (empiler) : ajoute une donnée sur le sommet de la pile
  - **POP** (dépiler) : retire une donnée du sommet de la pile



Appel de fonctions

Pile d'appels MIPS

- La pile d'appels n'est pas gérée par l'architecture MIPS. C'est le programme(ur) ou le compilateur qui en sont responsables !!
- L'organisation de données sur une pile n'est pas unique. Afin que des programmes écrits par des personnes / outils différents puissent s'appeler les uns les autres, il est nécessaire d'adopter une **convention d'appel** (*calling convention*).
- Dans ce cours, nous suivons l'***Application Binary Interface*** (**ABI**) **System V** pour MIPS définie par la société SCO.

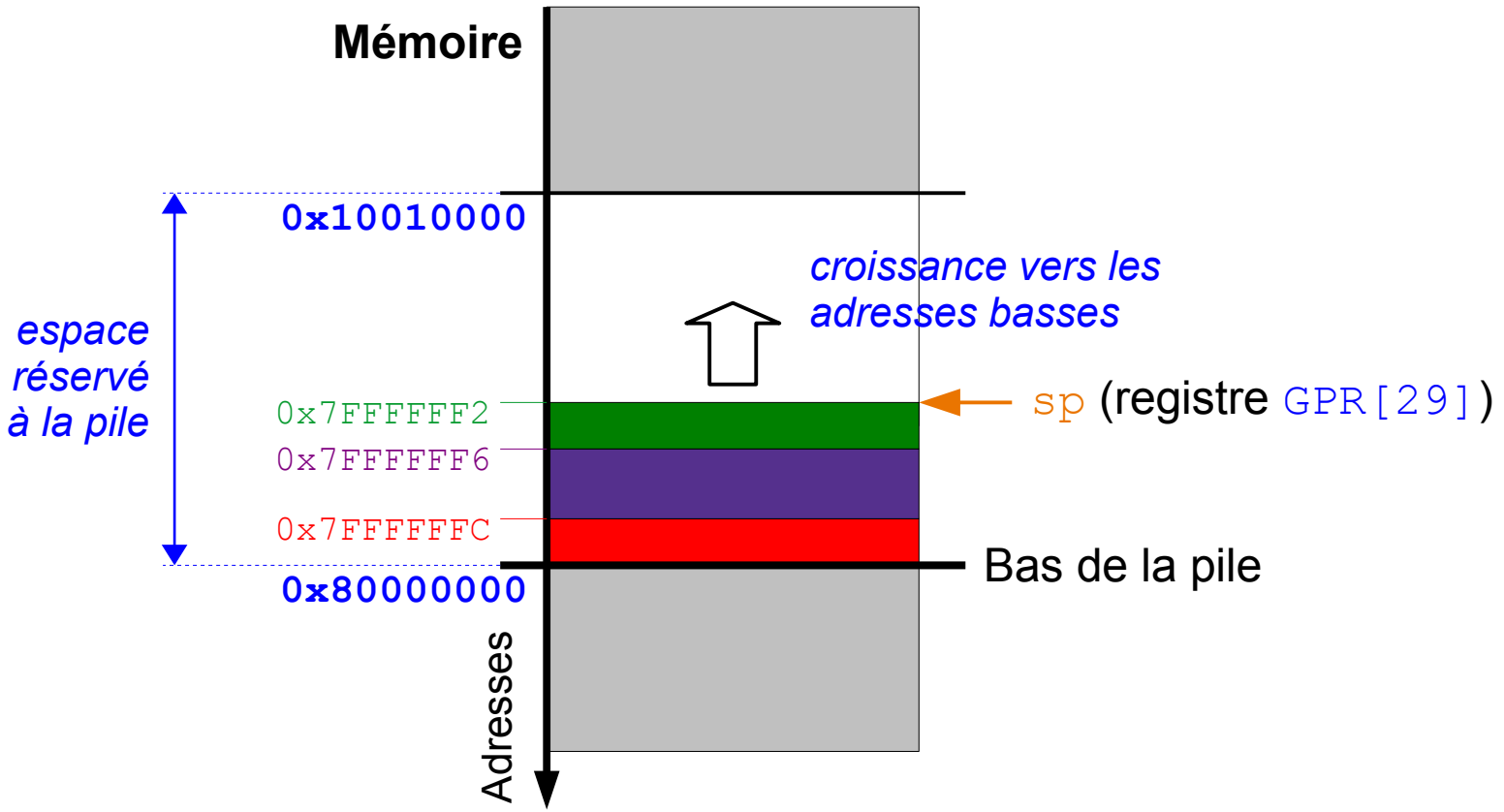
CONVENTION

- La pile se situe dans l'espace mémoire entre les adresses `0x10010000` et `0x7FFFFFFF`. Cet espace est partagé avec le tas.
- Le bas de la pile se situe à l'adresse `0x80000000`.
- La pile croît vers les adresses basses.
- L'adresse du sommet de la pile est conservée dans le registre GPR d'index 29, `sp` (*stack pointer*).

Reference : **SYSTEM V APPLICATION BINARY INTERFACE - MIPS RISC Processor Supplement (3rd Edition)**, The Santa Cruz Operation, Inc, February 1996. Cette ABI est parfois désignée par « o32 ».

Pile d'appels MIPS

- L'exemple ci-dessous montre la structure de la pile en mémoire selon l'ABI MIPS System V.



Appel de fonctions

Pile d'appels MIPS – Opérations

- Initialisation : La pile est initialisée en plaçant la valeur `0x80000000` (bas de la pile) dans le registre `sp`.

- PUSH de N octets
  - Déplacer le pointeur de pile de N octets en soustrayant N de `sp`.
  - Copier les données vers la mémoire à partir de l'adresse `sp`.

```
addiu  $sp, $sp, -4       # Déplacer pointeur pile
sw     $ra, 0($sp)        # Copier données vers pile
```

- POP de N octets
  - Copier les données de la mémoire à partir de l'adresse `sp`.
  - Déplacer le pointeur de pile de N octets vers le bas en ajoutant N à `sp`.

```
lw     $ra, 0($sp)        # Copier données depuis pile
addiu  $sp, $sp, 4        # Déplacer pointeur pile
```

Appel de fonctions

Pile d'appels MIPS

- Une fonction qui souhaite effectuer un appel imbriqué (ou récursif) doit
  - sauvegarder le registre `ra` au sommet de la pile auparavant
  - récuperer le registre `ra` sur la pile avant de retourner

PUSH(ra)
POP(ra)

```
main:
    addiu $sp, $sp, -4
    sw    $ra, 0($sp)
    jal   fct1              # Appel
    lw    $ra, 0($sp)       # Adresse de retour
    addiu $sp, $sp, 4
    jr    $ra
```

PUSH(ra)
POP(ra)

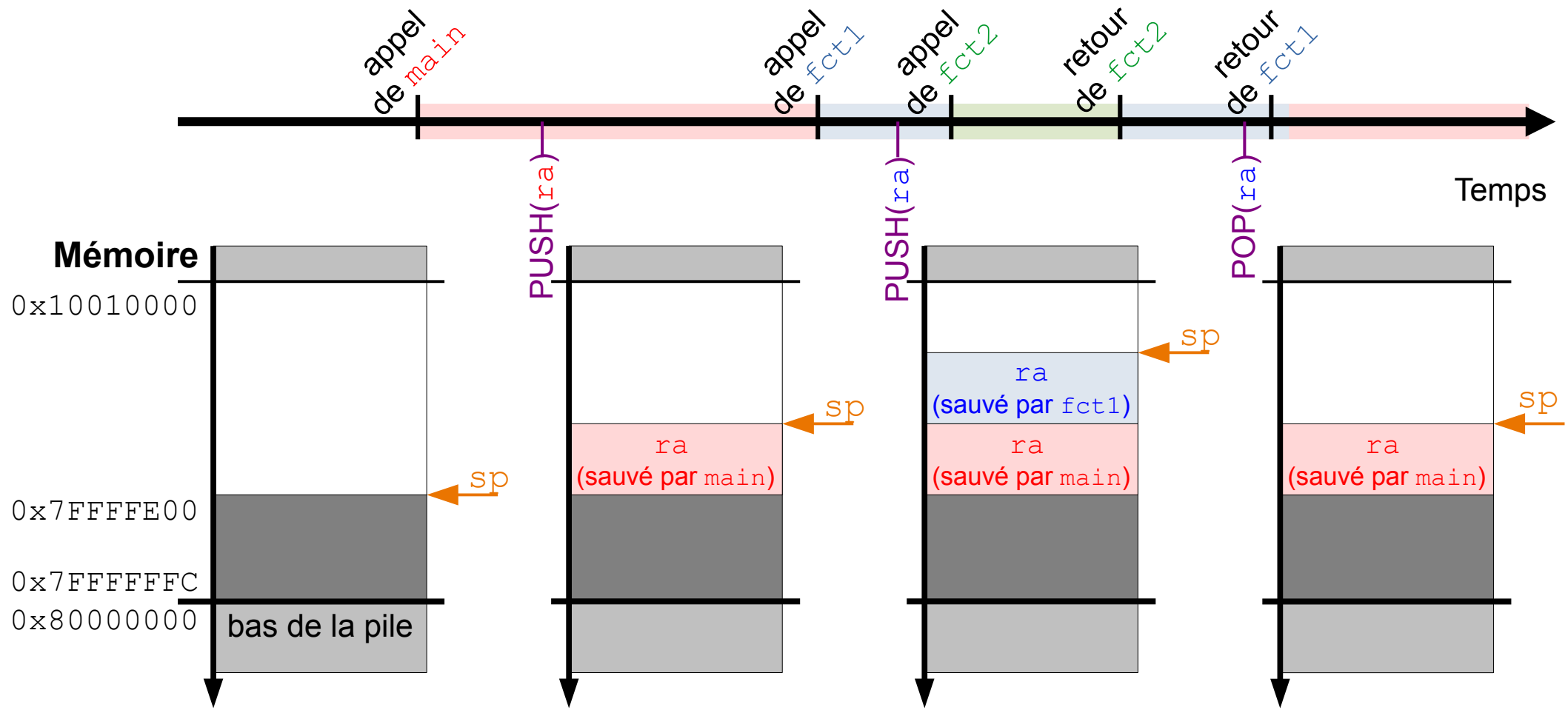
```
fct1:
    addiu $sp, $sp, -4
    sw    $ra, 0($sp)
    jal   fct2              # Appel (imbriqué)
    lw    $ra, 0($sp)
    addiu $sp, $sp, 4
    jr    $ra               # Retour
```

```
fct2:
    jr    $ra               # Retour
```

Appel de fonctions

Pile d'appels MIPS

- Les schémas ci-dessous illustrent l'état de la pile d'appels lors des appels de `main`, `fct1` et `fct2`.



Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Appel de fonctions

Passage de paramètres / valeurs de retour

- Comment l'appelant peut-il **passer des arguments** à une fonction ? Comment la fonction peut-elle **retourner des résultats** à l'appelant ?

- Les deux méthodes les plus répandues consistent à passer les arguments via des registres et/ou via la pile.
  - Via des **registres**
    - Avantage : rapidité d'accès
    - Désavantage : nombre limité de registres
  - Via la **pile**
    - Avantage : zone mémoire très grande
    - Désavantage : accès mémoire nécessaires (lents)
  - **Hybride**
    - Passage d'arguments dans des registres et sur la pile

Appel de fonctions

Convention d'appel

- A nouveau, pour permettre la conception de fonctions par des programmeurs / outils différents, il est nécessaire de se mettre d'accord sur comment les arguments et valeurs de retour sont passés entre fonctions.

- Par exemple, l'**ABI MIPS System V** utilise une approche hybride :

CONVENTION

- Les **registres** `a0` à `a3` sont utilisés pour passer les 4 premiers arguments d'une procédure
- Les autres arguments sont passés sur la **pile**. Le premier argument excédentaire se trouve à l'adresse la plus basse (plus haut sur la pile). L'appelant est responsable de retirer les arguments de la pile après l'appel.
- Les arguments occupent chacun au moins un mot de 32 bits et sont alignés en fonction de leur taille.
- Les **registres** `v0` et `v1` sont utilisés pour les valeurs de retour

Appel de fonctions

Exemple – Addition de 2 entiers

- La fonction `fct_add()` additionne deux nombres entiers. En suivant l'ABI MIPS, les arguments `a` et `b` seront passés dans les registres `a0` et `a1`. La valeur de retour sera transmise via le registre `v0`.

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

```
fct_add(5, 17);
```

- L'implémentation en langage d'assemblage d'une telle fonction peut être réalisée comme suit:

```
fct_add:
    add    $v0, $a0, $a1
    jr     $ra
```

```
main:
    addiu $sp, $sp, -4
    sw    $ra, 0($sp)
    …
    li    $a0, 5
    li    $a1, 17
    jal   fct_add
    …
    lw    $ra, 0($sp)
    addiu $sp, $sp, 4
    jr    $ra
```

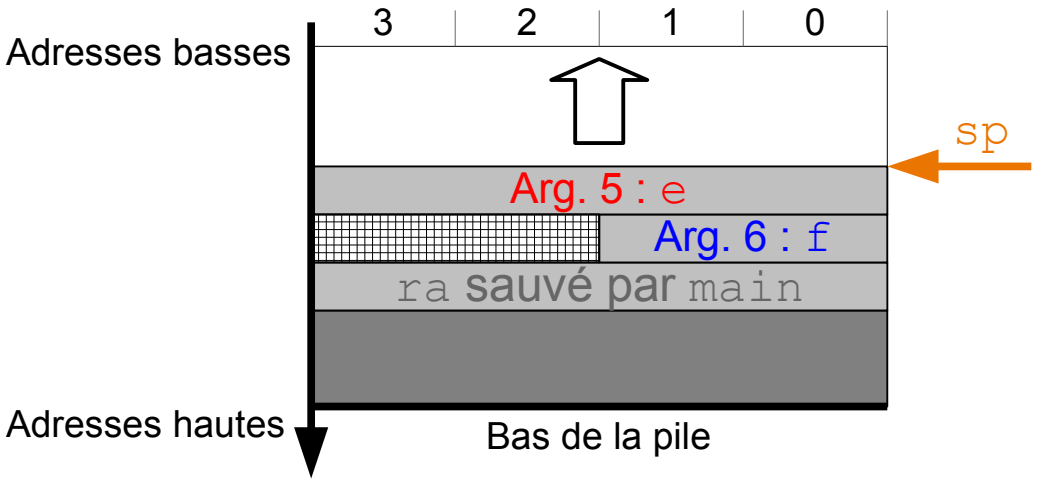
Appel de fonctions

Exemple – Addition de 6 entiers

- La fonction `fct_add6()` effectue l'addition de 6 nombres entiers. Les arguments `a` à `d` sont passés dans les registres `a0` à `a3`. Les arguments **`e`** et **`f`** sont passés sur la pile. La valeur de retour est transmise via le registre `v0`.

```
int fct_add6(int a, int b, int c, int d, int e, short f) {
  return a + b + c + d + e + f;
}
```

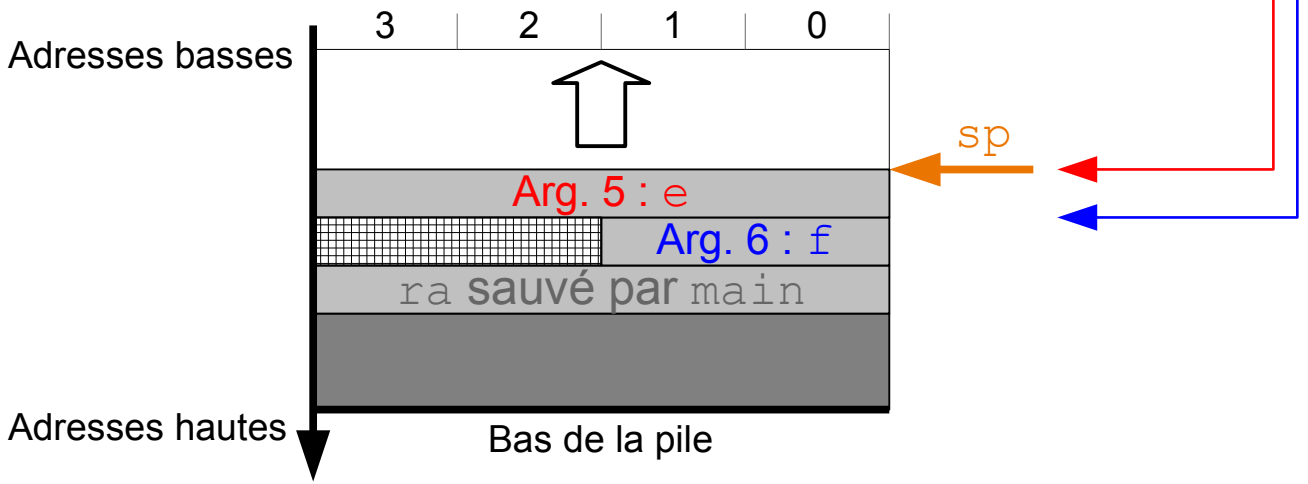
- En suivant l'ABI MIPS System V, la structure de la pile est la suivante :



Exemple – Addition de 6 entiers

- L'implémentation en langage d'assemblage de `fct_add6` peut être réalisée comme suit :

```
fct_add6:
    lw      $v0, 0($sp)         # Charger 'e'
    lh      $t0, 4($sp)         # Charger 'f'
    add     $v0, $v0, $t0       # Additionner 'e' et 'f'
    add     $v0, $v0, $a0       # Ajouter 'a'
    add     $v0, $v0, $a1       # Ajouter 'b'
    add     $v0, $v0, $a2       # Ajouter 'c'
    add     $v0, $v0, $a3       # Ajouter 'd'
    jr      $ra
```

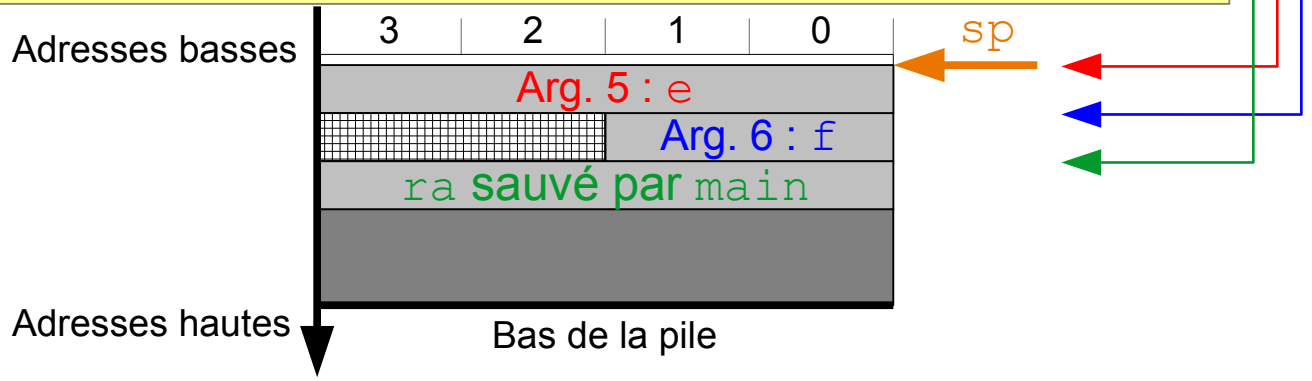


Appel de fonctions

Exemple – Addition de 6 entiers

- Le programme en langage d'assemblage ci-dessous illustre l'appel de la procédure `fct_add6(a=5,b=17,c=-1,d=3,e=0,f=9)`.

```
main:
      li      $a0, 5                  # Valeur de 'a'
      li      $a1, 17                 # Valeur de 'b'
      li      $a2, -1                 # Valeur de 'c'
      li      $a3, 3                  # Valeur de 'd'
      addiu   $sp, $sp, -12           # Allouer espace pile
      sw      $ra, 8($sp)             # Sauver $ra
      li      $t0, 9                  # Valeur de 'f'
      sw      $t0, 4($sp)             # Placer valeur de 'f'
      li      $t0, 0                  # Valeur de 'e'
      sw      $t0, 0($sp)             # Placer valeur de 'e'
      jal     fct_add6
      lw      $ra, 8($sp)             # Récupérer $ra
      addiu   $sp, $sp, 12            # Restaurer pile
      jr      $ra
```



Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- ⇒ Variables locales

Conclusion

Appel de fonctions

Variables locales

- Une fonction peut avoir des **variables locales** (*local data*). Ces variables ont une portée limitée à la fonction et leur durée de vie est au maximum celle de la fonction. Elles sont typiquement placées dans des registres et/ou sur la pile.

- L'ABI MIPS stipule que

CONVENTION

- Les registres `s0` à `s7` sont dédiés aux variables locales des fonctions. Une fonction peut donc librement utiliser ces registres pour stocker ses variables locales.
- Si cela n'est pas suffisant, les variables excédentaires doivent être stockées sur la pile.

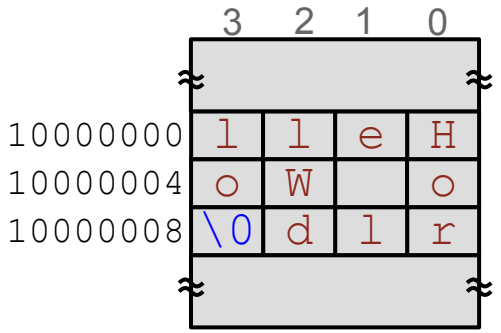
Appel de fonctions

Exemple – Longueur d'une chaîne de caractères

- La fonction `strlen` permet de déterminer la longueur d'une chaîne de caractères (à zéro terminal). L'adresse du premier caractère de la chaîne est passée en argument dans le registre `a0`. La variable locale `len` est stockée dans le registre `s0`.

```
int strlen(char * s) {
  int len= 0;
  while (*(s++) != '\0')
    len++;
  return len;
}
```

```
strlen:
      li      $s0, 0
strlen_loop:
      lb      $t8, 0($a0)
      beq     $t8, $zero, strlen_end
      addi    $a0, $a0, 1
      addi    $s0, $s0, 1
      b       strlen_loop
strlen_end:
      move    $v0, $s0
      jr      $ra
```



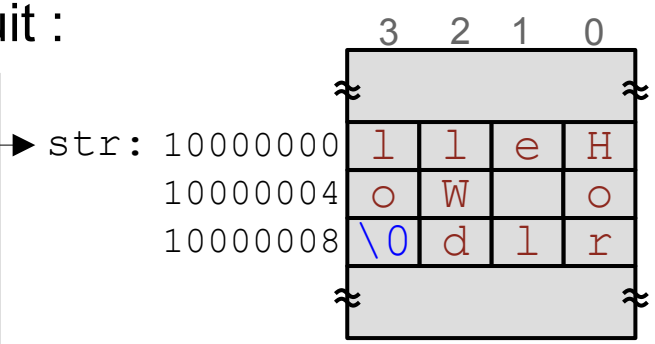
Note : une implémentation plus classique (et plus efficace) évite d'incrémenter `s0` à chaque itération, mais conserve l'adresse initiale de la chaîne dans `s0` et retourne `a0-s0`.

Appel de fonctions

Exemple – Longueur d'une chaîne de caractères

- L'appel de la fonction peut être réalisé comme suit :

```
        .data
str:    .asciiz "Hello World"

        .text
main:
        la      $a0, str
        addiu   $sp, $sp, -4
        sw      $ra, 0($sp)
        jal     strlen
        lw      $ra, 0($sp)
        addiu   $sp, $sp, 4
        # … afficher résultat (contenu dans $v0)
        jr      $ra
```

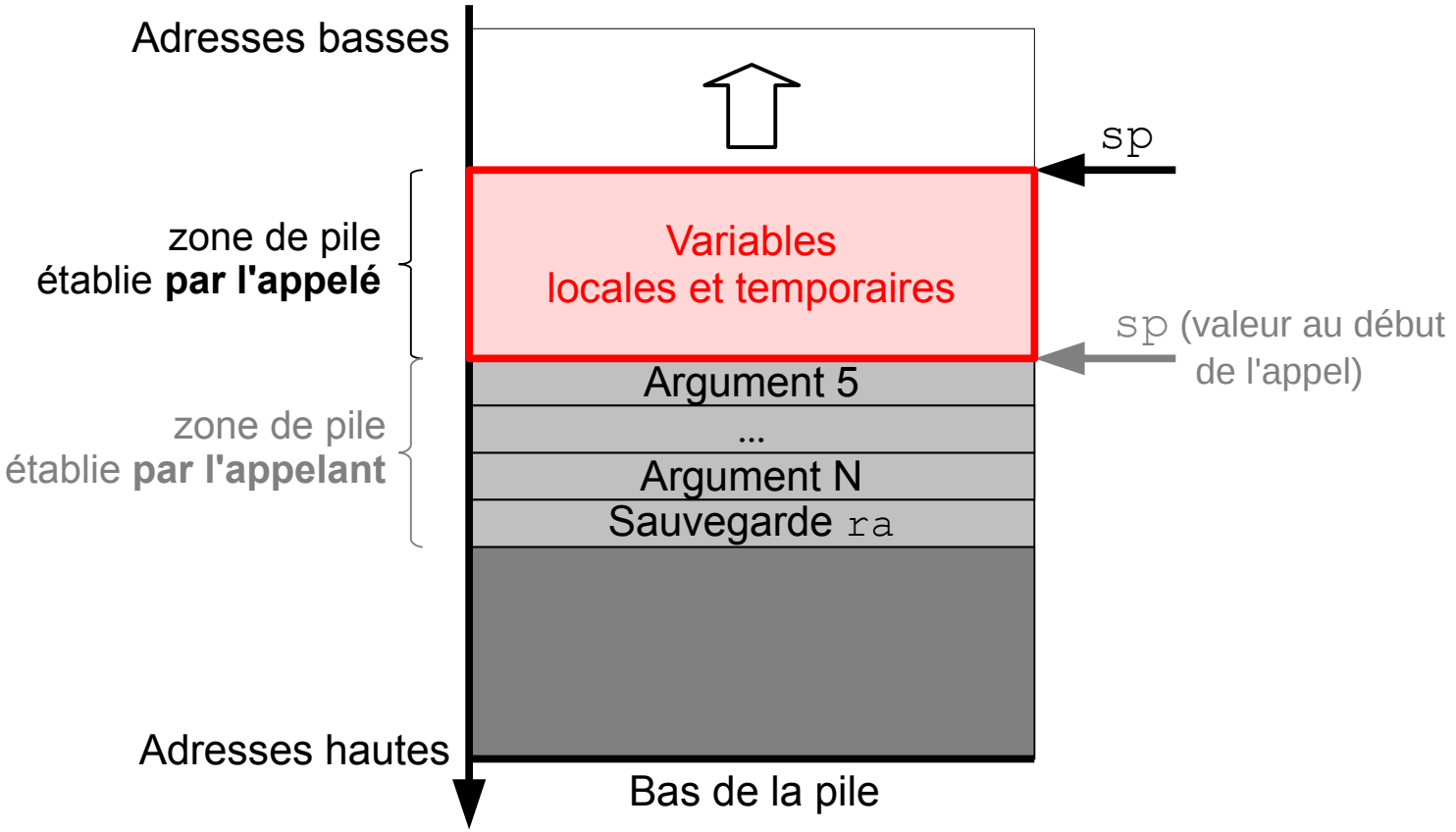


- Pour rappel,
  - Les directives `.data` et `.text` permettent de placer ce qui suit en mémoire respectivement de données et de programme.
  - La directive `.asciiz` place en mémoire les octets d'une chaîne de caractères. Un caractère de code 0 est placé en fin de chaîne.
  - La pseudo-instruction **`la`** charge l'adresse '`str`' dans le registre `a0` (argument de `strlen`).

Appel de fonctions

Variables locales sur la pile

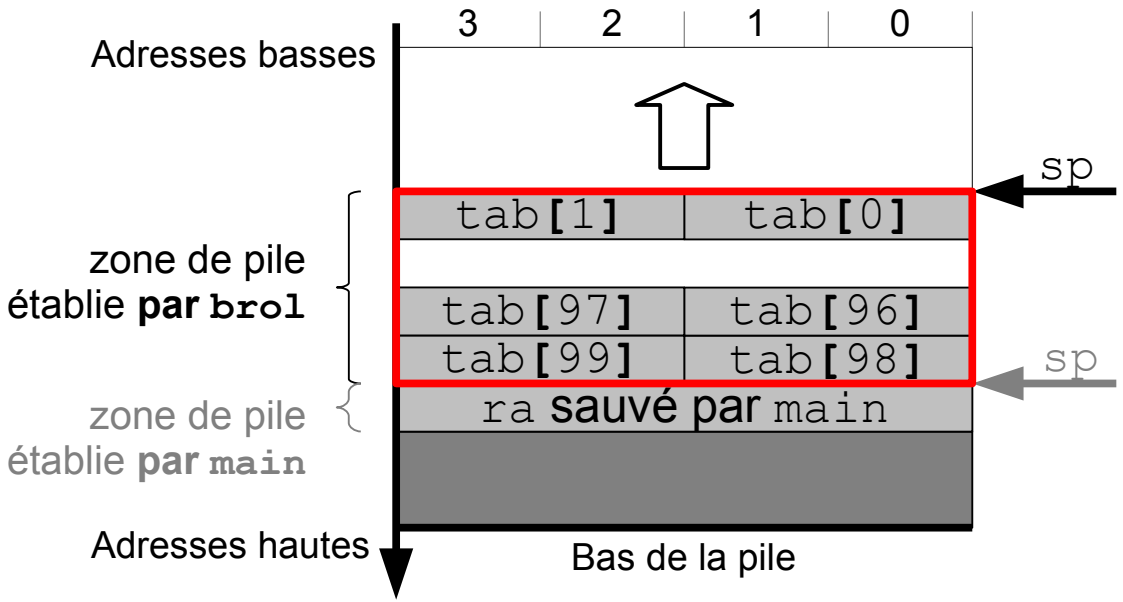
- L'**ABI MIPS** définit une zone de stockage de variables locales sur la pile comme suit



Exemple – Variables locales sur la pile

- La fonction suivante définit un tableau local de 100 entiers courts (`short` – sur 16 bits). Ce tableau sera stocké sur la pile.

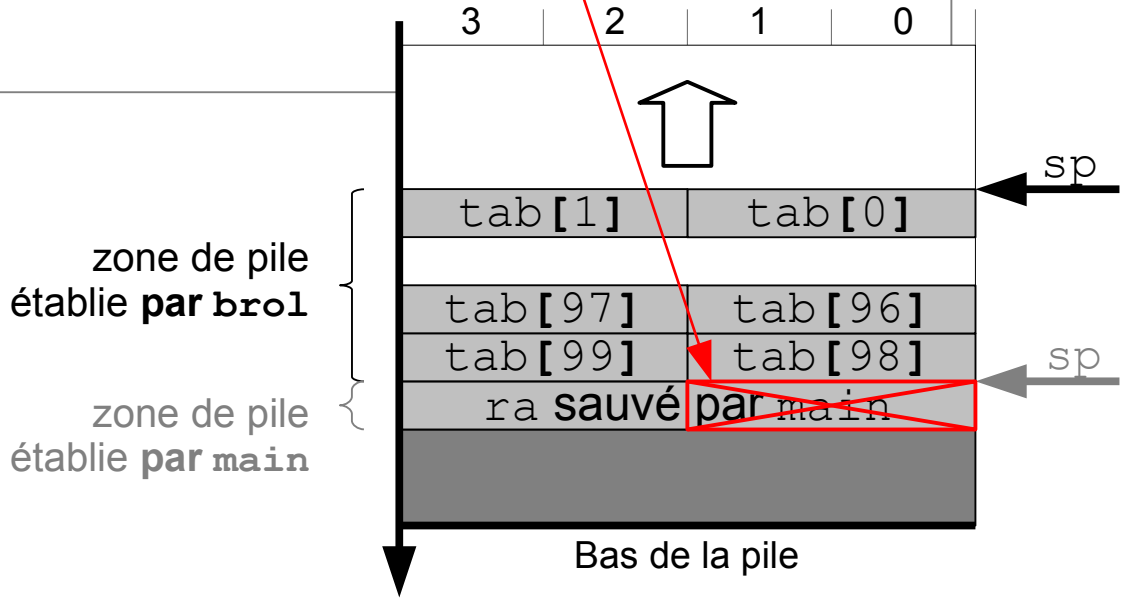
```
unsigned int brol(int a, int b) {
   short tab[100];
   // calcul complexe basé sur tab, a et b
   return ...;
}
```



Exemple – Buffer overflow

- Attention, en langage C, il n'y a pas de vérification de l'accès en dehors de bornes d'un tableau (pas d'exception `ArrayOutOfBoundsException` comme en Java) !
- Que se passe-t-il avec le programme suivant ?

```
unsigned int brol(int a, int b) {
   short tab[100];
   // ...
   tab[100]= 0;
   // ...
   return ...;
}
```



Appel de fonctions

Préservation du contenu des registres

- Avant d'effectuer un appel de fonction, le programme appelant a peut-être stocké des résultats temporaires dans certains registres. La fonction appelée pourrait utiliser ces registres pour son propre compte et en modifier le contenu.
- Par conséquent, il est nécessaire de sauvegarder sur la pile le contenu des registres qui contiennent des valeurs dont on aura encore besoin après l'appel de fonction.

- Qui est responsable d'effectuer la sauvegarde ?
  - L'appelant pourrait sauvegarder tous les registres dont il aura besoin
    (hypothèse : l'appelé peut modifier ces registres)
  - L'appelé pourrait sauvegarder tous les registres qu'il modifie
    (hypothèse : l'appelant aura besoin de ces registres)

- Appliquer naïvement cette façon de procéder peut amener à des gaspillages de ressources.

Appel de fonctions

Préservation du contenu des registres

- Afin de limiter le gaspillage de ressources lors de la sauvegarde de registres, il est possible de diviser les registres temporaires en deux groupes
  - les registres qui doivent être **préservés par l'appelant** si besoin est (*caller-saved registers*)
  - les registres qui doivent être **préservés par l'appelé** si besoin est (*callee-saved registers*).

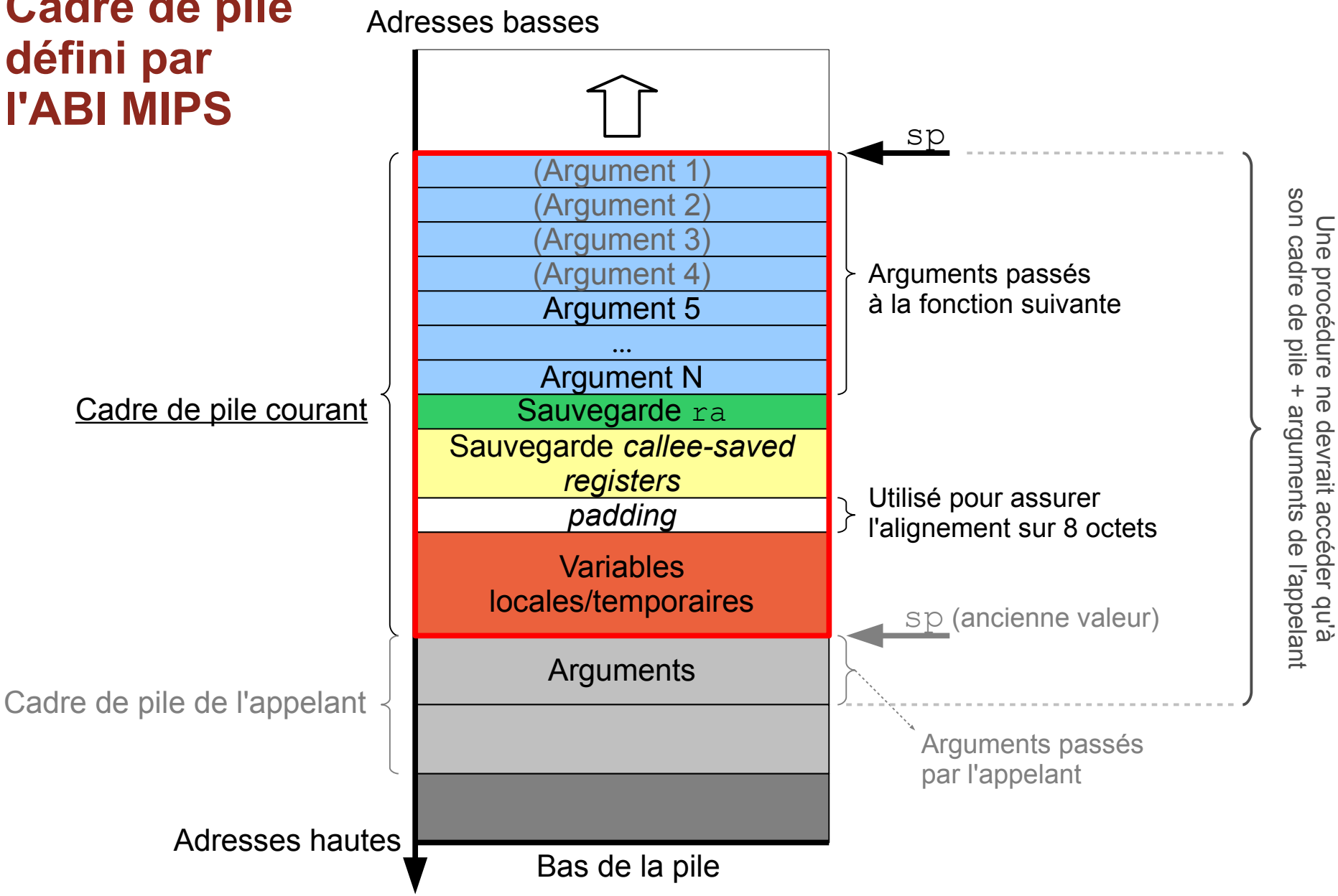
- L'**ABI MIPS** définit que

CONVENTION

- les registres `s0` à `s7` sont, si nécessaire, préservés par l'appelé (*callee-saved registers*).
- les registres `t0` à `t9` sont, si nécessaire, préservés par l'appelant (*caller-saved registers*).

Appel de fonctions

Cadre de pile défini par l'ABI MIPS



Exercice – Fonction récursive : factorielle

- On s'intéresse au calcul récursif de la factorielle

$$n! = \begin{cases} n(n-1)! & si \quad n>1 \\ 1 & si \quad n=0 \quad ou \quad n=1 \end{cases}$$

- Une implémentation récursive en langage C est donnée ci-dessous.

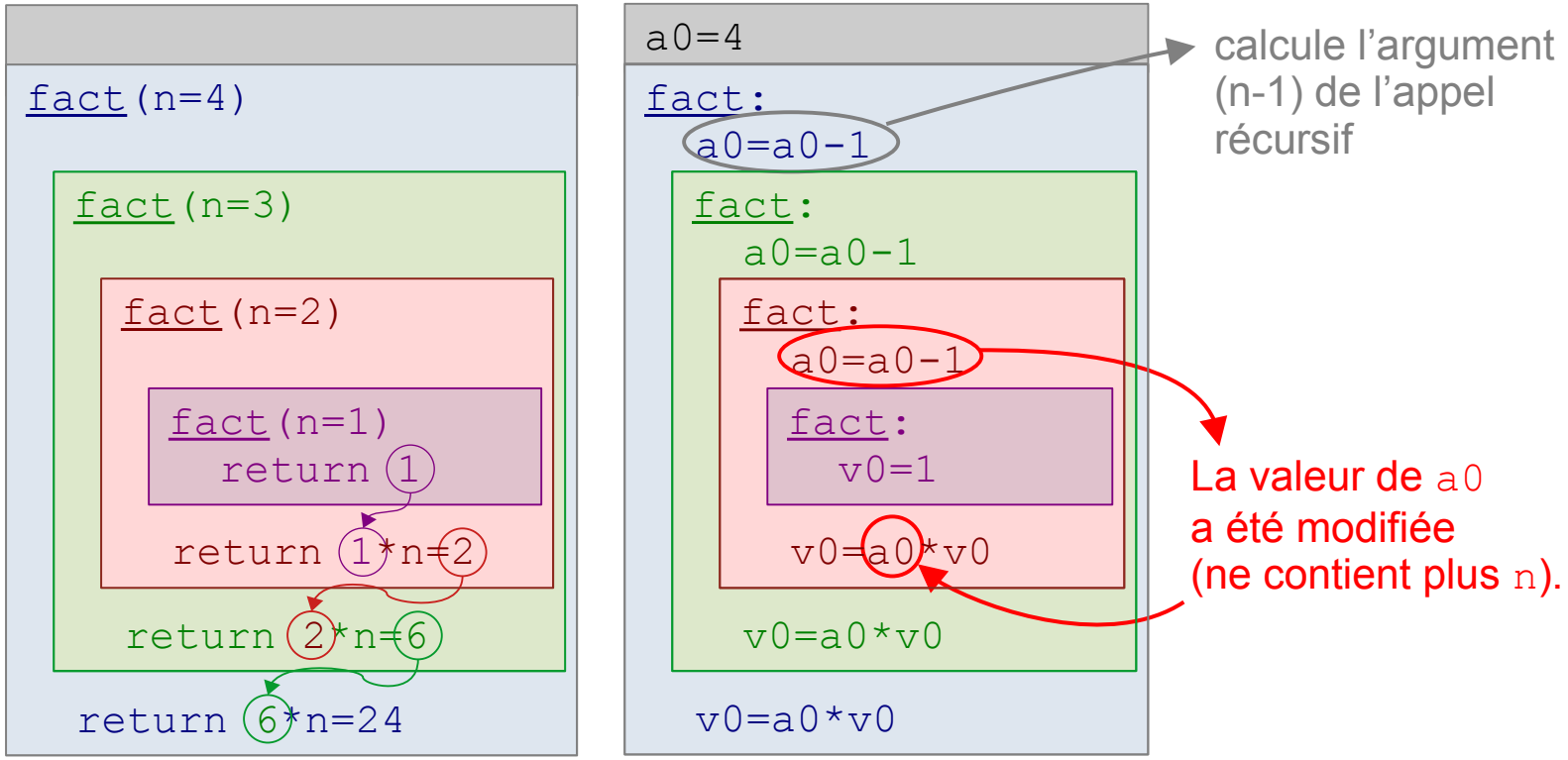
```
// pré-condition: n >= 0
unsigned int fact(unsigned int n) {
  if (n > 1)
    return n * fact(n-1);
  return 1;
}
```

Appel de fonctions

Exercice – Fonction récursive : factorielle

- Supposons que l'argument `n` soit passé dans le registre `a0` et le résultat retourné dans `v0`. Observons le comportement de la fonction pour un l'appel `fact(4)`.

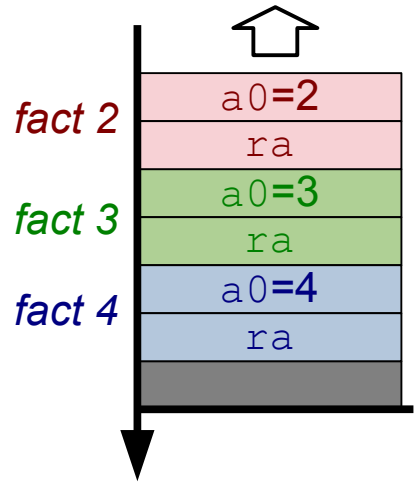
Appels récursifs de `fact()` jusqu'à ce que le cas de base (n ≤ 1) soit atteint.



Exercice – Fonction récursive : factorielle

- Deux approches sont possibles pour éviter de « *perdre* » la valeur de l'argument `n` (registre `a0`) lors des appels récursifs.

- Sauvegarde sur la pile
  - Sauvegarder `a0` sur la pile et le restaurer
  - Avantage : fonctionne dans tous les cas
  - Désavantage : accès mémoires supplémentaires



- Inverser la modification
  - Effectuer la modification inverse de `a0`, i.e. calculer `a0=a0+1`
  - Avantage : pas d'accès mémoire supplémentaire
  - Désavantage : pas toujours applicable/pratique

Appel de fonctions

Exercice – Fonction récursive : factorielle

- Dans cette implémentation de la factorielle, la valeur de `n` (contenue dans `a0`) est sauvegardée sur la pile avant chaque appel récursif.

Cas de base (n ≤ 1)

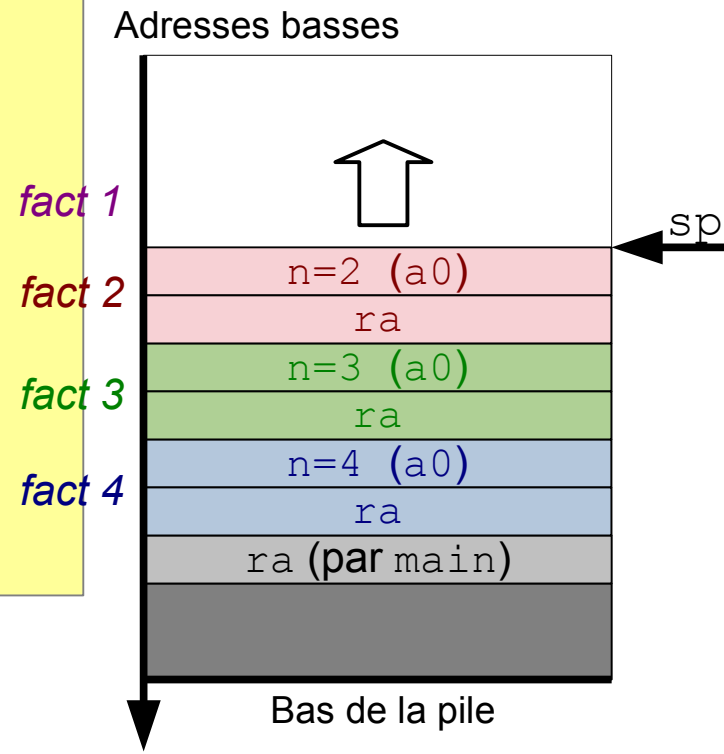
Cas récursif (n > 1)

Sauve `n` et `$ra` sur la pile

Appel récursif (`n`-1)

Récupére `n` et `$ra` sur la pile

```
fact:
        li      $s0, 1
        bgt     $a0, $s0, fact_recurse
        li      $v0, 1
        jr      $ra
fact_recurse:
        addiu   $sp, $sp, -8
        sw      $a0, 0($sp)
        sw      $ra, 4($sp)
        addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 0($sp)
        lw      $ra, 4($sp)
        addiu   $sp, $sp, 8
        mult    $v0, $a0
        mflo    $v0
        jr      $ra
```



Appel de fonctions

Introduction

Registres

- Compteur de programme et registre d'instruction
- Registres généraux (GPR)

Langage d'assemblage

Jeu d'instructions

- Opérations arithmétiques et logiques
- Load et Store
- Sauts
- Branchements conditionnels

Appel de fonctions

- Pile d'appels
- Passage d'arguments et de résultats
- Variables locales

Conclusion

Discussion

Conception d'un jeu d'instructions

- Dans ce chapitre, nous avons volontairement étudié en détail un **jeu d'instructions réel**, celui de l'architecture MIPS. Nous avons également montré comment il pouvait être implémenté.
- La conception d'un jeu d'instruction nécessite souvent d'effectuer **des choix et des compromis**. C'est le cas dans le jeu d'instructions MIPS.

Discussion

Conception d'un jeu d'instructions

- Simplicité = régularité
  - toutes les instructions de type R, par exemple, ont le même encodage avec 3 opérandes.
- Optimiser l'architecture pour les cas les plus communs
  - seule l'instruction `add` a une version immédiate (`addi`) car c'est la plus fréquente pour ce type d'usage
  - chargement de constantes, souvent petites → charger 16 bits poids fort/faible séparément (notamment avec `ori` / `lui`).
- Plus petit = plus rapide
  - load & store, nombre de registres limité à 32.
- Un bon “design” nécessite des compromis
  - choix d'une taille fixe d'instruction et ses implications.

Discussion

Conception d'un jeu d'instructions

- Il est important de noter que d'autres approches et jeux d'instructions ne font pas les mêmes choix que MIPS. On distingue généralement deux grandes familles d'architectures.

- **RISC** (*Reduced Instruction Set Computer*)
  - instructions travaillent uniquement sur des registres (*load & store*)
  - petit nombre d'instructions différentes
  - instructions simples à traiter en *hardware* (souvent taille fixe)
  - nécessite parfois plusieurs instructions pour effectuer une opération → plus de travail pour les compilateurs / assembleurs

- **CISC** (*Complex Instruction Set Computer*)
  - instructions peuvent travailler directement sur la mémoire
  - grand nombre d'instructions différentes (certaines très peu ou jamais utilisées)
  - instructions parfois très complexes à décoder et exécuter
  - généralement beaucoup plus difficile à implémenter en *hardware*

Architecture x86 (CISC)

- Le jeu d'instructions de l'architecture x86 diffère fortement de celui de MIPS. Par exemple
  - x86 utilise des instructions de taille variable.

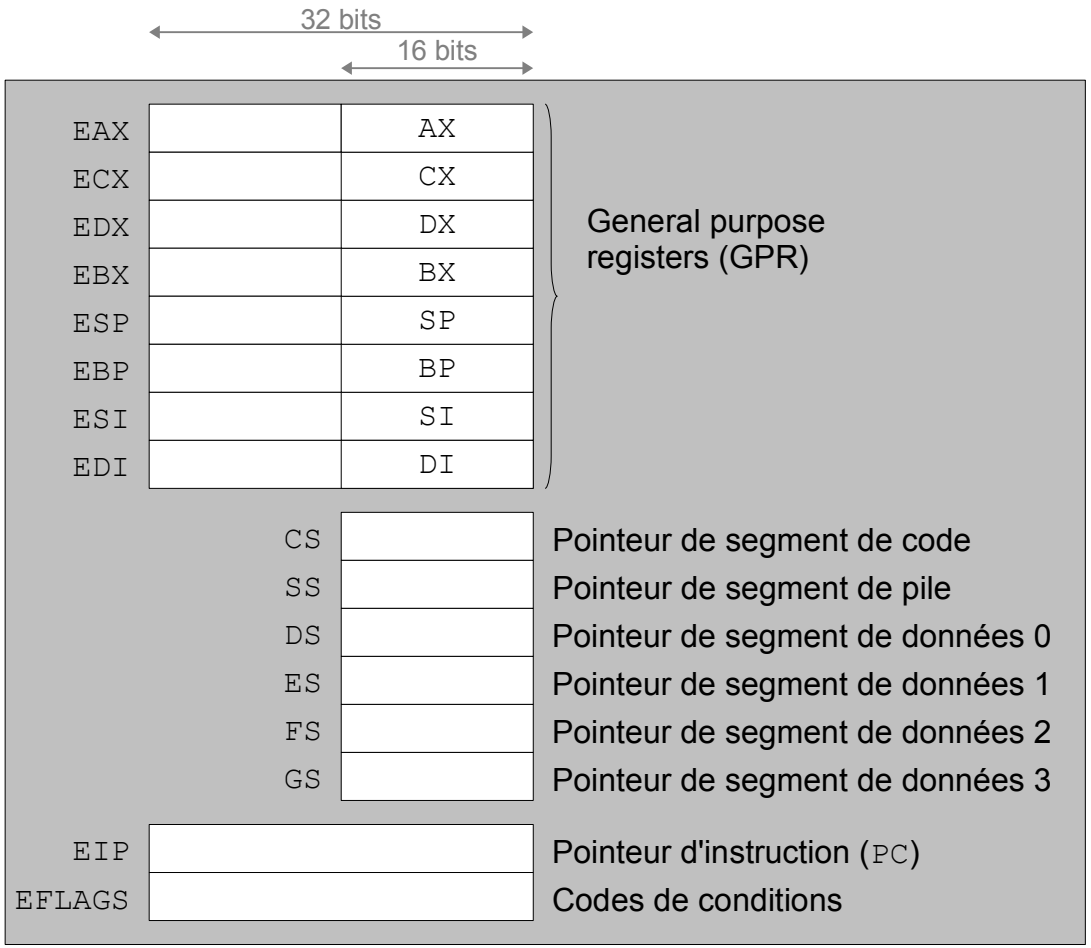
Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
1 octet/préfixe jusqu'à 4 préfixes	1-3 octets	1 octet	1 octet	1, 2 ou 4 octets	1, 2 ou 4 octets

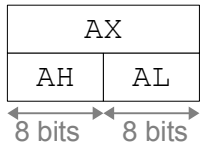
  - Les instructions x86 peuvent adresser des registres mais également directement adresser la mémoire.
  - L'architecture x86 définit un petit nombre de registres (seulement 8 registres GPR!)
  - Le nombre d'instructions est très grand : il y a des instructions très spécialisées comme MMX, SSE, SSE2, ...

Source : Intel 64 and IA-32 Architectures Software Developer's Manual (Vol. 2)

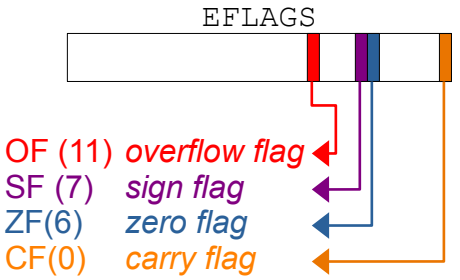
Architecture x86 (CISC)

- Registres des processeurs de la famille IA-32.





Les registres AX, CX, DX et BX sont eux-mêmes découpés en sous registres de 8 bits



Source : Intel 64 and IA-32 Architectures Software Developer's Manual (Vol. 2)

Discussion

Architecture x86 (CISC)

- Les instructions x86 diffèrent fortement de MIPS.
- Exemple : instructions arithmétiques et logiques
  - ont seulement deux opérandes (contre 3 pour MIPS)
  - peuvent adresser directement la mémoire
  - les combinaisons d'opérandes possibles sont les suivantes

Source1/destination	Source2
Registre	Registre
Registre	Immédiat
Registre	Mémoire
Mémoire	Registre
Mémoire	Immédiat

  - il existe 7 modes différents d'adressages mémoire !
    - dont : *direct*, *indirect par registre*, *indirect par registre avec offset*, *indirect par registre avec offset par registre plus multiplicateur immédiat*, ...

Discussion

Architecture x86 (CISC)

- Contrairement à MIPS, le jeu d'instructions de l'architecture x86 offre un support direct pour certaines opérations fréquentes (mais complexes).

- La **gestion de la pile** est supportée en *hardware*
  - Instructions spécifiques permettant d'empiler (**`push`**), dépiler (**`pop`**)

- La **sauvegarde des adresses de retour sur la pile** est supportée en *hardware*
  - L'instruction d'appel de fonction **`call`** place directement l'adresse de retour sur la pile.
  - Il existe également une instruction spéciale de retour de fonction **`ret`** qui dépile l'adresse de retour avant de brancher.

Discussion

Architecture x86 (CISC)

- Voici par exemple comment des appels imbriqués de fonctions sont réalises en x86 (à droite) par rapport à MIPS (à gauche).

MIPS

```
main:
     addiu  $sp, $sp, -4
     sw     $ra, 0($sp)
     jal    fct1
     lw     $ra, 0($sp)
     addiu  $sp, $sp, 4
     ...
```

```
fct1:
     addiu  $sp, $sp, -4
     sw     $ra, 0($sp)
     jal    fct2
     lw     $ra, 0($sp)
     addiu  $sp, $sp, 4
     jr     $ra
```

```
fct2:
     jr     $ra
```

x86

```
main:
     call   fct1
     ...
```

```
fct1:
     call   fct2
     ret
```

```
fct2:
     ret
```

Références

Computer Organization and Design, 4th edition, D. Patterson and J. Hennessy, Morgan-Kauffman, 2009

CODE: The Hidden Language of Computer Hardware and Software, C. Petzold, Microsoft Press, 1999

A Practical Introduction to Computer Architecture, D. Page, Springer-Verlag, 2009

Digital Design and Computer Architecture, D. M. Harris and S. L. Harris, Morgan-Kauffman, 2007

System V Application Binary Interface, MIPS RISC Processor Supplement, 3rd edition, Santa Cruz Operation, Inc. (SCO), 1999-1996

See MIPS Run Linux, 2nd edition, D. Sweetman, Morgan-Kauffman, 2007.

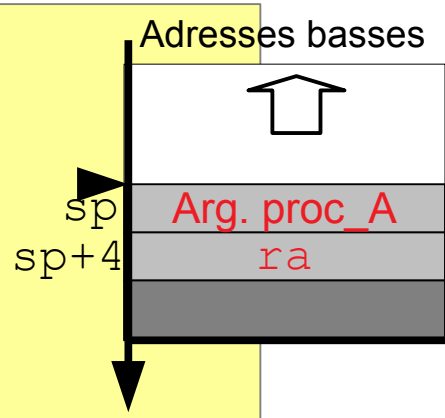
The Elements of Computing Systems: Building a Modern Computer From First Principles, N. Nisan and S. Schocken, MIT Press, 2005.

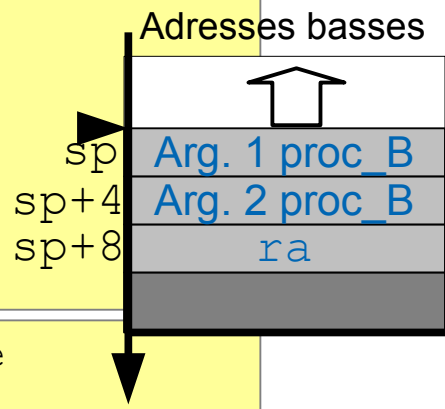
Appel de fonctions

Pile d'appels MIPS

Programme principal

```
main:
        addiu   $sp, $sp, -8          # PUSH
        sw      $ra, 4($sp)           # PUSH
        sw      $t0, 0($sp)           # PUSH  Arg. proc_A
        jal     proc_A                # Appel proc_A
        lw      $ra, 4($sp)           # POP
        addiu   $sp, $sp, 8           # POP
        addiu   $sp, $sp, -12         # PUSH
        sw      $ra, 8($sp)           # PUSH
        sw      $t0, 0($sp)           # PUSH  Arg. 1 proc_B
        sw      $t0, 4($sp)           # PUSH  Arg. 2 proc_B
        jal     proc_B                # Appel proc_B
        lw      $ra, 8($sp)           # POP
        addiu   $sp, $sp, 12          # POP
        jr      $ra                   # Retour
```





Procédure A (1 argument)

```
proc_A:                               # point d'entrée
        blabla
        jr      $ra                   # Retour
```

Procédure B (2 arguments)

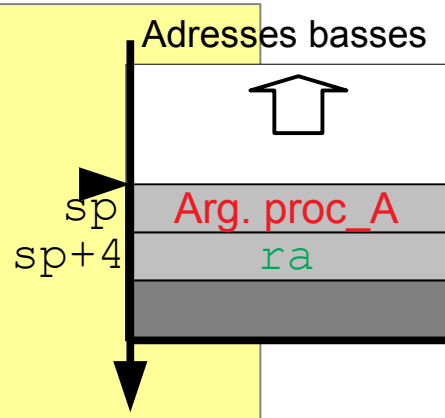
```
proc_B:                               # point d'entrée
        blibli
        jr      $ra                   # Retour
```

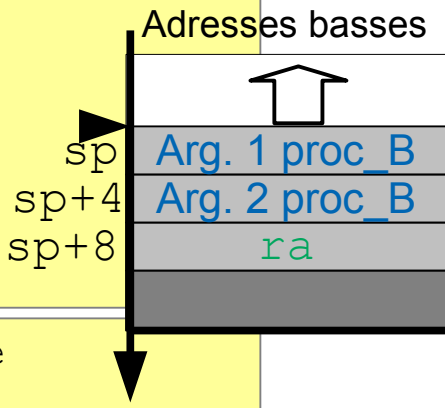
Appel de fonctions

Pile d'appels MIPS

Programme principal

```
main:
        PUSH  addiu   $sp, $sp, -8
              sw      $ra, 4($sp)
              sw      $t0, 0($sp)       # Arg. proc_A
              jal     proc_A            # Appel proc_A
        POP   
              addiu   $sp, $sp, -4
        PUSH
              sw      $t0, 0($sp)       # Arg. 1 proc_B
              sw      $t0, 4($sp)       # Arg. 2 proc_B
              jal     proc_B            # Appel proc_B
              lw      $ra, 8($sp)
        POP   addiu   $sp, $sp, 12
              jr      $ra               # Retour
```





Procédure A (1 argument)

```
proc_A:                               # point d'entrée
        blabla
        jr      $ra                   # Retour
```

Procédure B (2 arguments)

```
proc_B:                               # point d'entrée
        blibli
        jr      $ra                   # Retour
```