Architecture des ordinateurs
Cours 3

Responsable de l'UE : Emmanuelle Encrenaz
Supports de cours : Karine Heydemann

Contact : emmanuelle.encrenaz@lip6.fr

Plan du cours 3

Architecture générale d'un ordinateur

- Le processeur (ou CPU) est l'unité de traitement de l'information (instructions et données). Il exécute des programmes (suite d'instructions qui définissent un traitement à appliquer à des données).
- La mémoire centrale (ou RAM ou mémoire vive) est une unité de stockage temporaire des informations nécessaires à l'exécution d'un programme. Externe au processeur, elle stocke en particulier les intructions du programme en cours d'exécution ou à exécuter et les données du programme (nombre, caractères alphanumériques, adresses mémoire, ...).
- Le bus est le support physique des transferts d'information entre les différentes unités.
- Les périphériques sont des unités connexes permettant de communiquer avec l'ensemble processeur-mémoire : clavier, écran, disque dur, réseau, imprimante/scanner, ...

Instructions, données et programme

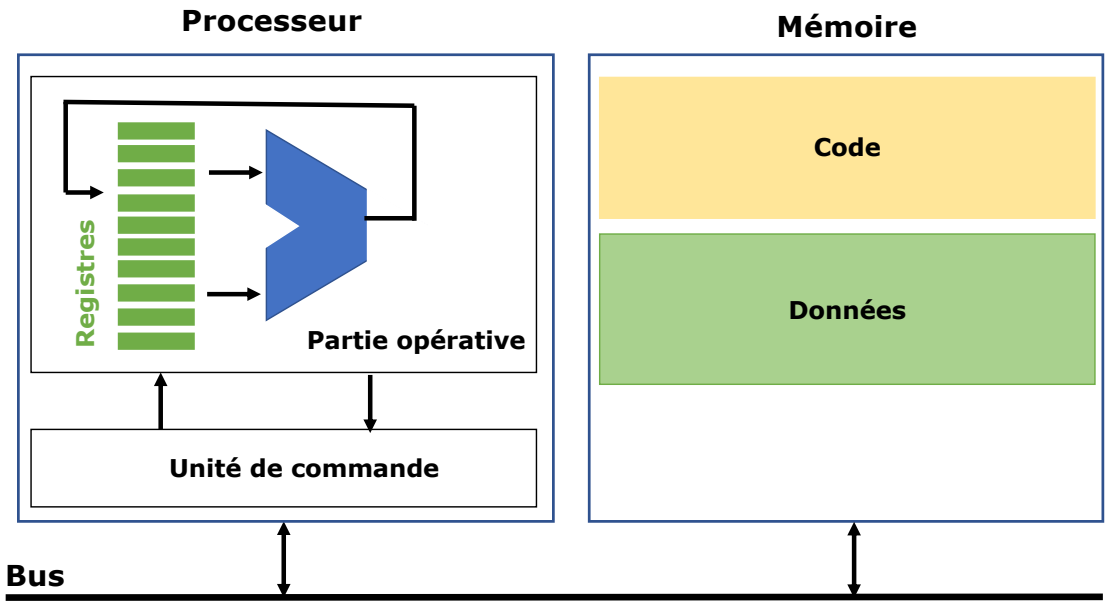
- Un programme définit un traitement à appliquer à des données
- Deux parties :
  - les données
  - le traitement qui est une suite d'opérations sur les données

Représentation en machine

- Les données sont représentées en binaire avec la représentation associée à leur nature (entiers relatifs, caractères,...) et stockées en mémoire
- Le traitement à réaliser est traduit en instructions compréhensibles par le processeur cible : ces instructions sont dites en langage machine
- Elles sont codées en binaire et stockées en mémoire

Stockage des informations

- Dans la mémoire sont stockées les données et les instructions du programme en cours d'exécution
- Dans le processeur, toute donnée (ou information) est stockée dans un registre : instruction en cours d'exécution + des données ou valeurs temporaires
- Transfert des informations entre la mémoire et le processeur via le bus



Les registres

- Un registre *n* bits est un composant capable de mémoriser un mot binaire de *n* bits.
- Changement de valeur possible uniquement lors de front montant/descendant du signal de l'horloge
- Émission de la valeur contenue dans le registre en continu

- Il existe un grand nombre de registres dans un processeur
- Ils stockent toute information nécessaire au processeur

Les registres d'un processeur

Les registres d'un processeur

- L'architecture du processeur définit le nombre, la taille et le nom des registres du processeur
- Certains peuvent être manipulés explicitement par le programmeur via des instructions

Manipulation des registres

- Affectation explicite d'une valeur à un registre par le biais d'instructions
  - Exemple en Mips : \$5 $\leftarrow$ 2, \$6 $\leftarrow$ \$7 + \$8
- Affectation implicite de certains registres lors de l'exécution d'instructions particulières (exemples dans la suite)

Les registres du Mips

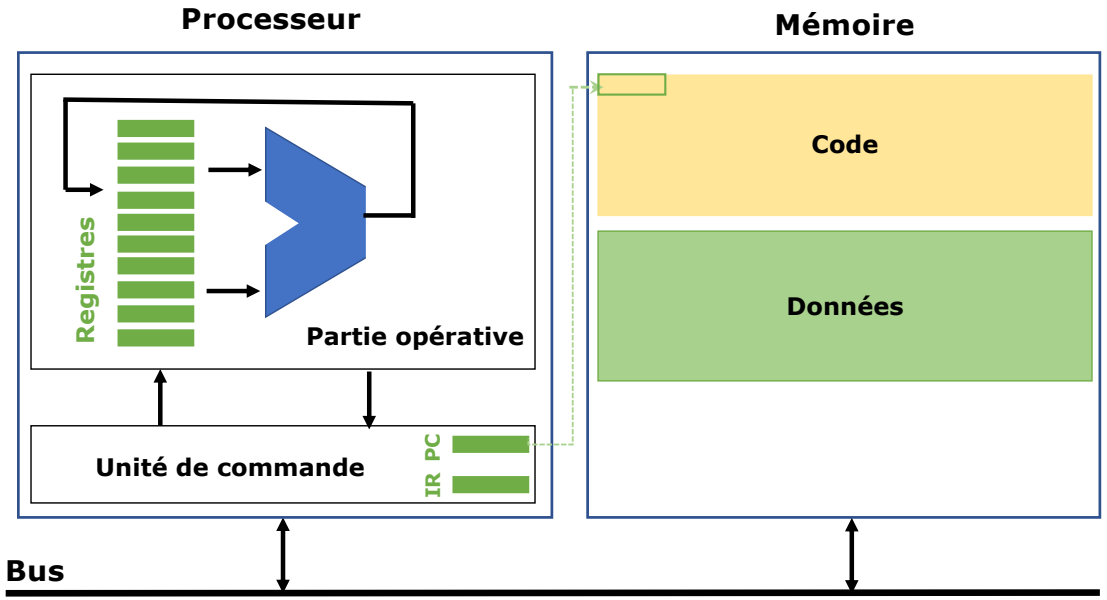
Les registres du Mips

- Les registres du Mips font 32 bits
- Les registres généraux : 32 registres de **`$0`** à **`$31`** ; ce sont des registres de travail, accessibles directement par le logiciel (code assembleur), mais chacun de ces registres a une utilisation prédéfinie
- **`PC`** (Programme Counter) contient l'adresse de l'instruction en cours d'exécution (ou la suivante) ; modifié après l'exécution de chaque instruction
- **`IR`** (Instruction Register) contient l'instruction en cours de traitement
- **`HI`**/**`LO`** (High/Low) sont les registres contenant le résultat d'opérations de multiplication ou de division
- Autres registres non utilisés dans ce cours : EPC, BAR, SR, ...

Boucle d'exécution réalisée par processeur

Deux registres particuliers dans le processeur

- `PC` = Program Counteur
- `IR` = Instruction Register



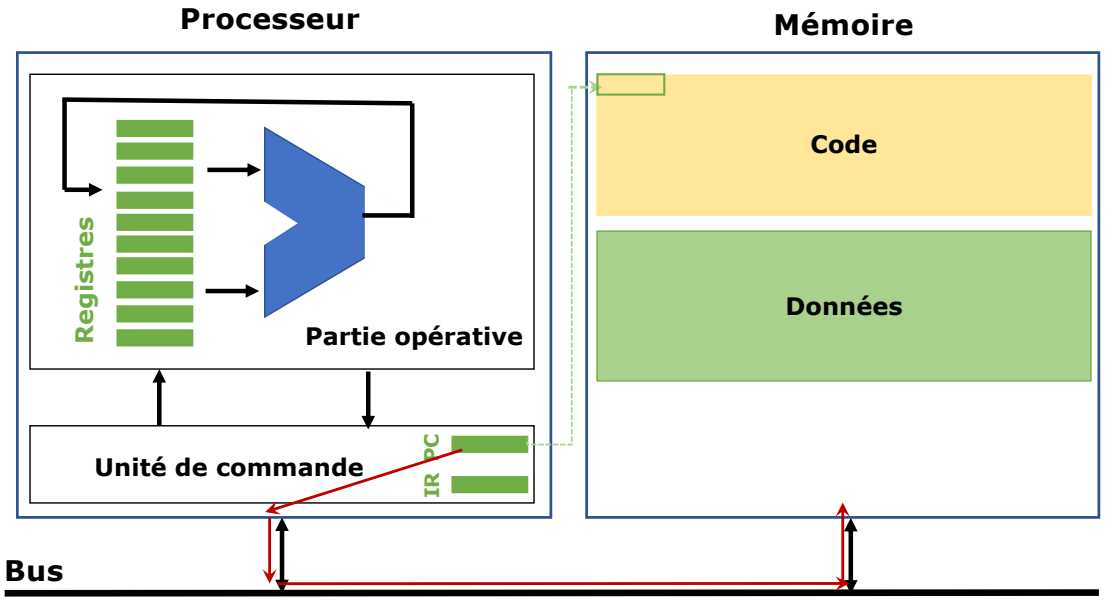
Le processeur exécute sans fin la suite des opérations suivantes

1. Lire une instruction en mémoire (mise dans `IR`)

Boucle d'exécution réalisée par processeur

Deux registres particuliers dans le processeur

- `PC` = Program Counteur
- `IR` = Instruction Register



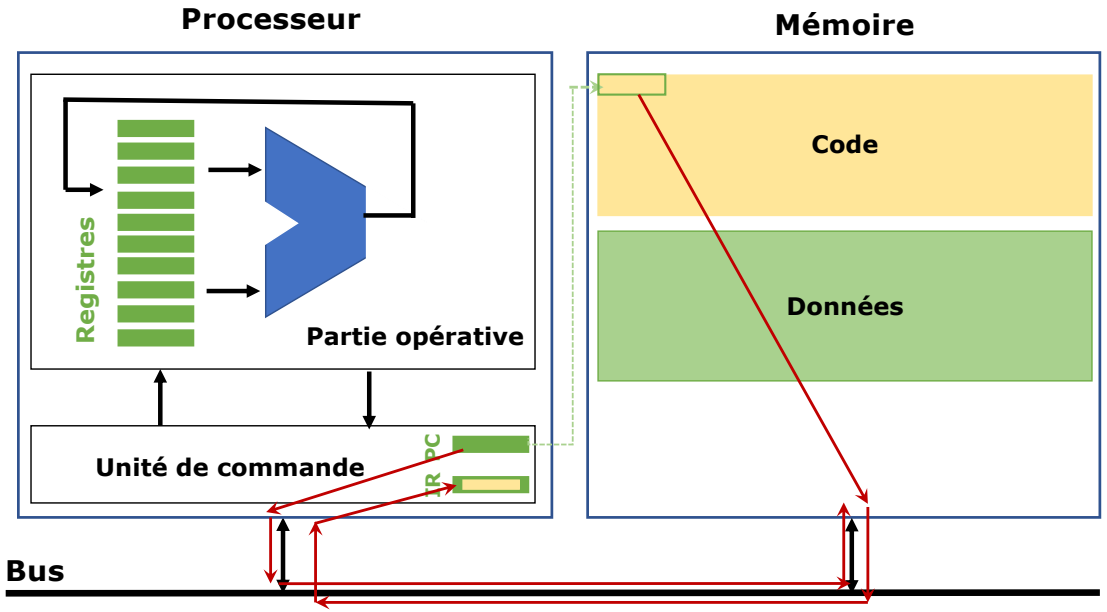
Le processeur exécute sans fin la suite des opérations suivantes

1. Lire une instruction en mémoire (mise dans `IR`)

Boucle d'exécution réalisée par processeur

Deux registres particuliers dans le processeur

- `PC` = Program Counteur
- `IR` = Instruction Register



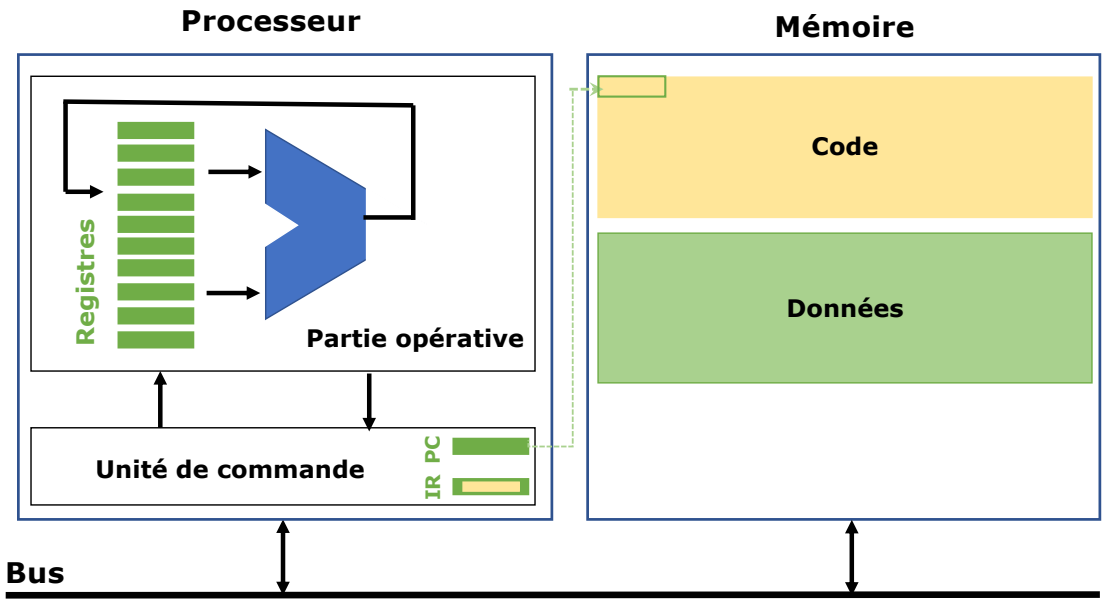
Le processeur exécute sans fin la suite des opérations suivantes

1. Lire une instruction en mémoire (mise dans `IR`)

Boucle d'exécution réalisée par processeur

Deux registres particuliers dans le processeur

- `PC` = Program Counteur
- `IR` = Instruction Register



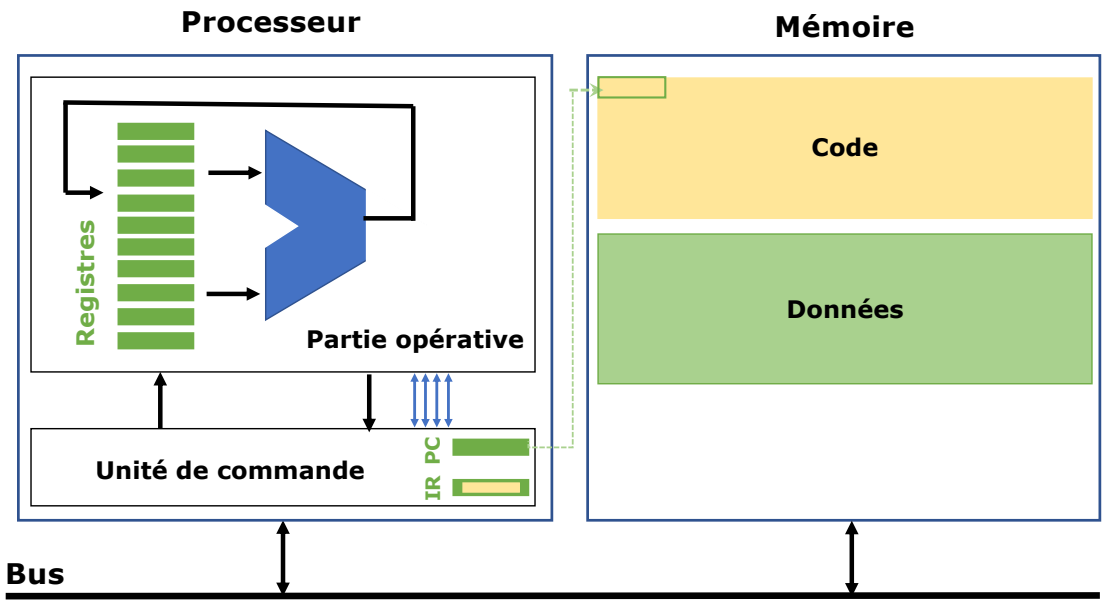
Le processeur exécute sans fin la suite des opérations suivantes

1. Lire une instruction en mémoire (mise dans `IR`)
2. Décoder l'instruction : par exemple le codage de l'instruction `add $4,$3,$2`

Boucle d'exécution réalisée par processeur

Deux registres particuliers dans le processeur

- `PC` = Program Counteur
- `IR` = Instruction Register



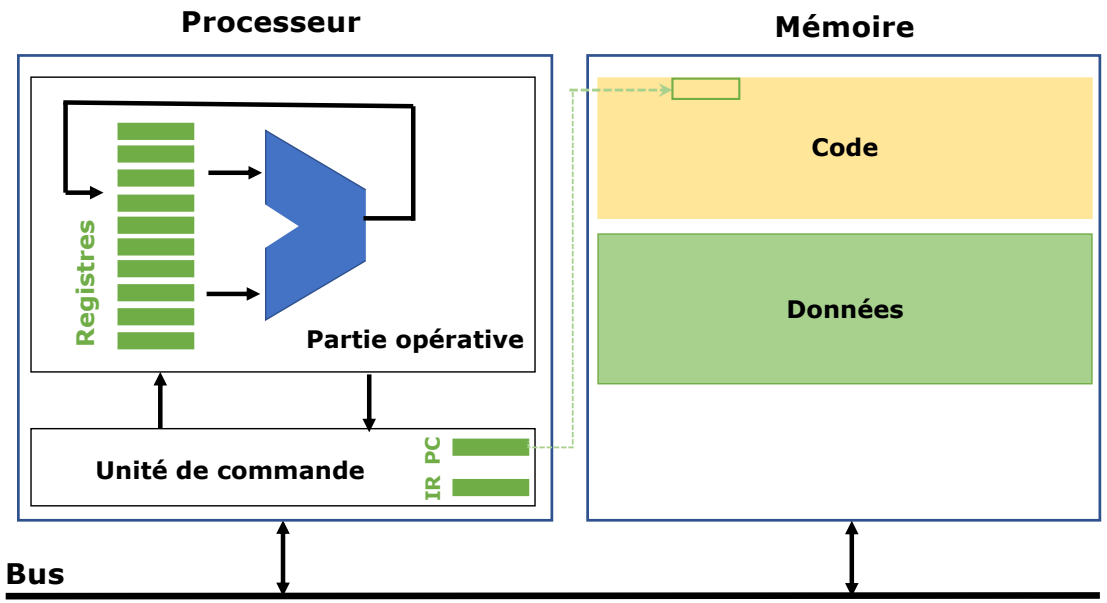
Le processeur exécute sans fin la suite des opérations suivantes

1. Lire une instruction en mémoire (mise dans `IR`)
2. Décoder l'instruction : par exemple le codage de l'instruction `add $4,$3,$2`
3. Exécuter l'instruction : orchestration d'actions pour additionner le contenu des registres `$2` et`$3` et mettre le résultat dans le registre `$4`

Boucle d'exécution réalisée par processeur

Deux registres particuliers dans le processeur

- `PC` = Program Counteur
- `IR` = Instruction Register



Le processeur exécute sans fin la suite des opérations suivantes

1. Lire une instruction en mémoire (mise dans `IR`)
2. Décoder l'instruction : par exemple le codage de l'instruction `add $4,$3,$2`
3. Exécuter l'instruction : orchestration d'actions pour additionner le contenu des registres `$2` et`$3` et mettre le résultat dans le registre `$4`
4. Calculer l'adresse de l'instruction suivante : mise à jour `PC`

Les registres généraux du Mips

Sémantique et utilisation (1)

- **$0** (zero) : contient toujours la valeur 0 ; une écriture dans ce registre ne modifie pas son contenu
- **$1** (at) : registre réservé à l'assembleur (programme qui génère le binaire)
- **$2** (v0) : résultat d'un appel de fonction + numéro d'un appel système
- **$3** (v1) : résultat d'un appel de fonction
- **$4** à **$7** (a0 à a3) : utilisés pour le passage d'arguments lors des appels de fonction ou les appels système (non persistants)
- **$8** à **$15** (t0 à t7), **$24** (t8) et **$25** (t9) : registres non persistants (utilisation libre)
- **$16** à **$23** (s0 à s7) : registres persistants (utilisation libre)

Persistance

- Registres persistants : leur contenu est inchangé au travers des appels systèmes et de fonction.
- Registres non persistants : leur contenu peut changer/n'est pas garanti au travers des appels systèmes et de fonction

Les registres généraux du Mips

Sémantique et utilisation (2)

- **`$26`** et **`$27`** (k0, k1) : registres réservés au système d'exploitation
- **`$28`** (gp) : contient l'adresse de base des variables globales ("global pointer")
- **`$29`** (sp) : contient l'adresse du sommet de la pile ("stack pointer", ou pointeur de pile)
- **`$30`** (fp) : contient l'adresse de base de la fenêtre courante ("frame pointeur")
- **`$31`** (ra) : adresse de retour dans un appel de fonction

Remarques

- En Mips, un registre peut être noté avec un `R`, `r` ou `$` (`R7 = r7 = $7`)
- Les registres en **rouge** ne seront pas utilisés dans le cadre de ce cours
- Vous devez connaître et respecter ces règles d'utilisation

Jeu d'instructions et représentation des instructions

Jeu d'instructions

Vue externe d'un processeur

La vue externe d'un processeur peut être définie par l'ensemble des instructions qu'il est capable de traiter : c'est son jeu d'instructions

Jeu d'instructions : composition

Le jeu d'instructions d'un processeur (ISA) est la donnée :

1. de l'ensemble des instructions qu'il peut effectuer
2. du codage de ces instructions en binaire

Définition d'un jeu d'instructions

Le jeu d'instructions et l'architecture interne du processeur sont définis conjointement

Qu'est ce qu'une instruction ?

Définition d'une instruction

En langage machine c'est une commande donnée au processeur qui définit :

- Le traitement à effectuer maintenant
- Quelle sera la prochaine instruction à exécuter

Traitement à effectuer

- l'opération mise en jeu (ex : addition, opération logique, opération mémoire)
- les opérandes sur lesquelles elle porte : la ou les opérandes sources, l'opérande destination s'il y en a une.

Prochaine instruction à exécuter

Deux cas possibles :

- implicite ET séquentiel : la prochaine instruction à exécuter est celle implantée en mémoire à la suite de l'instruction courante
- explicite dans des instructions spécifiques (instructions dites de saut)

Syntaxe assembleur des instructions

un `CodeOperation` et des `Operandes`

- **CodeOperation** est un mnémonique indiquant l'opération à effectuer : `add, andi, lw, sw, j, beq`
- **Operandes** est une suite d'opérandes potentiellement vide
  - Des constantes dit immédiats (`-1`, `0xFFFF`, `3`, ...)
  - Des registres du processeur nommés ou numérotés (`$0, $1, ..., $31`)

Exemples d'instruction avec différents formats d'opérandes

- OpReg, OpReg, OpReg : `add $4, $2, $5`
- OpReg, OpReg, OpImm_{16} : `ori $4, $2, 0xABCF`
- OpReg, OpReg : `mult $4, $2`
- OpReg, OpImm_{16}(OpReg) : `lw $4, 8($5)`
- OpReg, OpReg, OpLabel : `beq $4,$2, loop`
- OpLabel : `j ma_fonction`

4 différentes classes d'instructions / traitements

Instructions arithmétique et logiques

Ces instructions utilisent l'ALU pour réaliser un calcul sur des données

Instructions de transfert mémoire

Ces instructions lisent ou écrivent des données en mémoire

Instructions de rupture de séquence

Ces instructions permettent de casser l'exécution séquentielle par défaut du code en spécifiant quelle sera la prochaine instruction à exécuter.

Instructions système

Elles demandent un service au système (arrêt du programme, affichage d'une valeur...)

Instruction arithmétiques et logiques

Ces instructions utilisent l'ALU pour réaliser un calcul sur des données

- Le résultat est toujours stocké dans un registre
- Les opérandes sources sont des registres et/ou des constantes entières codées sur 16 bits (et étendues sur 32 à l'exécution)

Instructions arithmétiques

- addition et soustraction : `add`, `addu`, `addi`, `addiu`, `sub`, `subu`
  - entre 2 registres : `add $4,$2,$3`
    `$4 ← $2 + $3`
  - entre un registre et un immédiat : `addi $3,$5,0xFFFF`
    `$3 ← $5 + 0xFFFFFFFF` (extension signée)
- multiplication et division (`mult`, `div`) entre 2 registres avec registres destination implicite :
  - `mult $3,$4`
    `(HI,LO) ← $3 * $4`
  - `div $3,$4`
    `(HI,LO) ← $3 ÷ $4`

Instructions logiques

opérations logiques

- OU logique (bit à bit) : `or`, `ori`
  - entre 2 registres : `or $2,$4,$3`
    `$2 ← $4 | $3`
  - entre un registre et un immédiat : `ori $2,$3,0x00F0`
    `$2 ← $3 | 0x0000FF00` (extension non signée)
- ET logique (bit à bit) : `and`, `andi`
  - entre 2 registres : `and $2,$4,$3`
    `$2 ← $4 & $3`
  - entre un registre et un immédiat : `andi $2,$3,0x00F0`
    `$2 ← $3 & 0x000000F0` (extension non signée)
- OU exclusif (bit à bit) : `xor`
  - entre 2 registres : `xor $2,$2,$2`
    `$2 ← $2 ⊕ $2`

Instructions d'affectation et de décalage

Affectation de registre

- Mettre une valeur sur les 16 bits de poids fort :
  `lui $2,0xABCD` : `$2 ← 0xABCD0000`
- Mettre le contenu du registre `HI` dans `$4` : `mfhi $4`
- Mettre le contenu du registre `LO` dans `$4` : `mflo $4`

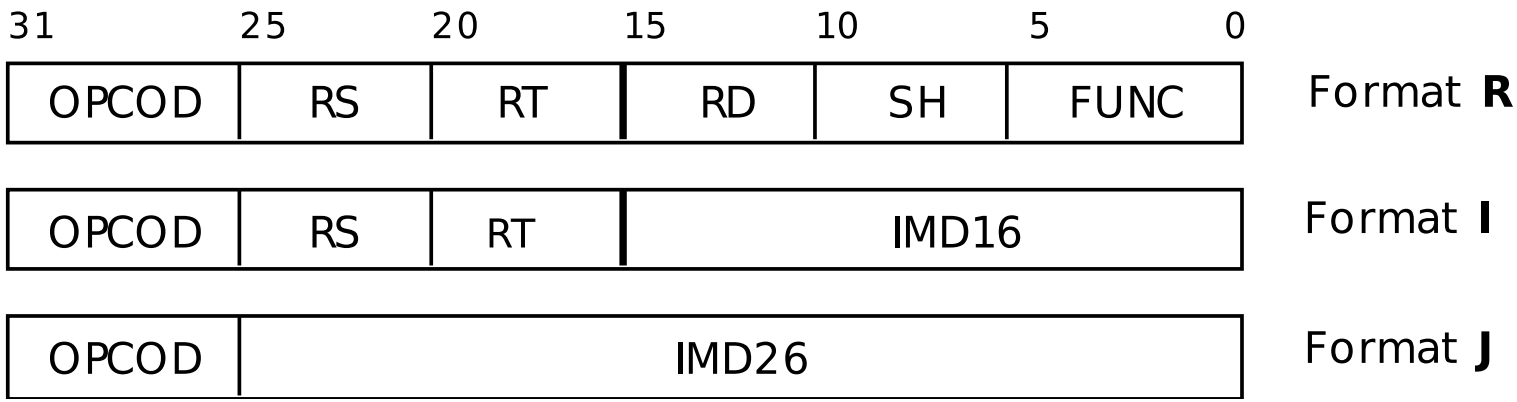
Opération de décalage

- à gauche : `sll`, `sllv`
  - avec 2 opérandes registres `sllv $2,$4,$3` : `$2 ← $4 « $3`
  - avec 1 opérande registre et 1 immédiat `sll $2,$4,16` : `$2←$4 « 16`
- à droite 'signé' dit arithmétique : `sra`, `srav`
  - avec 2 opérandes registres `srav $2,$3,$4` : `$2 ← $3 » $4`
  - avec 1 opérande registre et 1 immédiat `sra $2,$3,2` : `$2 ← $3 » 2`
- à droite 'non signé' dit logique : `srl`, `srlv`
  - avec 2 opérandes registres : `srlv`
  - avec 1 opérande registre et 1 immédiat : `srl`

Codage des instructions MIPS

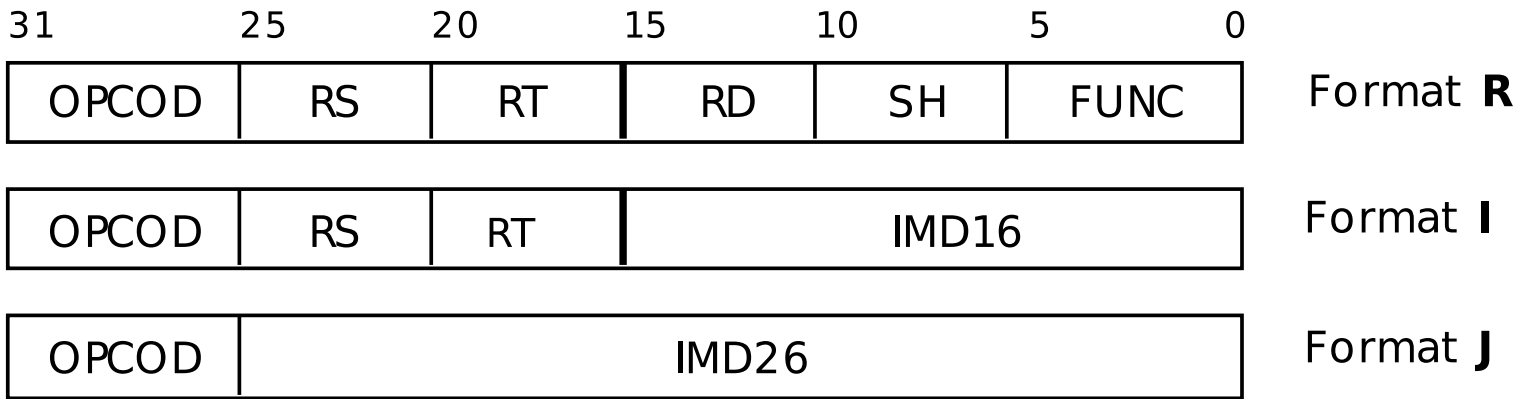
Le jeu d'instructions définit le format de codage binaires des instructions

- Toutes les instructions sont codées sur 32 bits
- 3 formats de codage appelés R, I, J
- Chaque format comporte plusieurs champs
- Un champ contient une information : code opération, numéro de registre, constante, ...



Format de codage des instructions MIPS

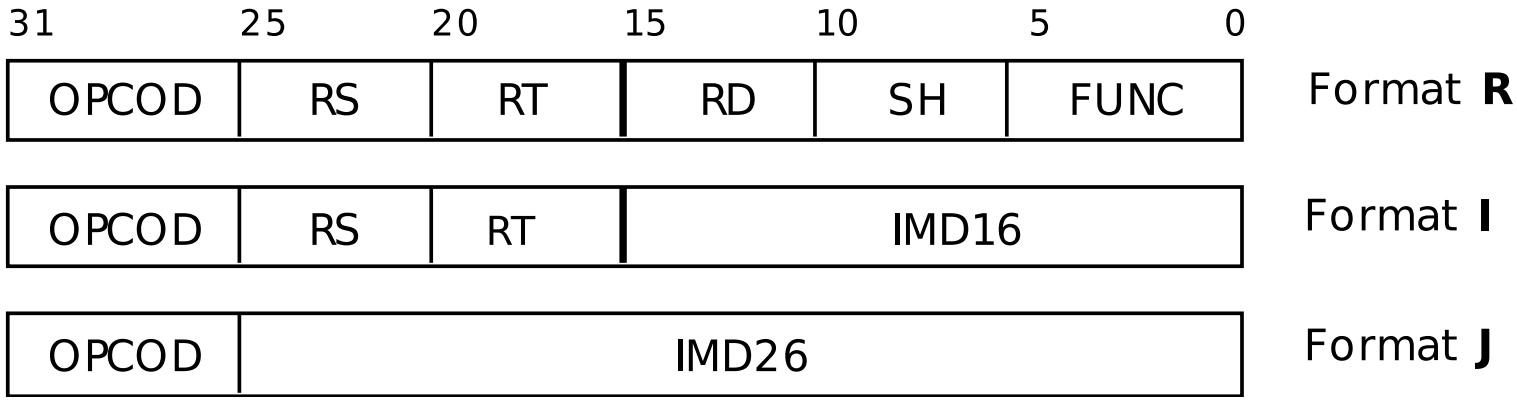
- Format R : instructions arithmétiques et logiques avec 2 registres sources, toutes les instructions de décalage
- Format I : instructions arithmétiques et logiques avec 1 opérande source immédiat (sauf décalage !), accès mémoire et sauts conditionnels
- Format J : instructions de saut inconditionnel avec adressage direct
- Le format est donné pour chaque instruction (nécessaire pour l'encodage d'une instruction)



Format de codage et champ du code opération

Le champ OPCOD de 6 bits (bits 31 à 26) est commun à tous les formats

- L'opcode (ou code opération) code l'opération correspondant à l'instruction
- Sa valeur est donnée par une table de correspondance entre mnémonique et un codage sur 6 bits
- Ce champs permet au processeur de savoir comment décoder les 26 bits restant et l'opération à effectuer



Code opération

- Le mnémonique `addi` a pour code opération 001000.
- Le code 100011 correspond au mnémonique `lw`.

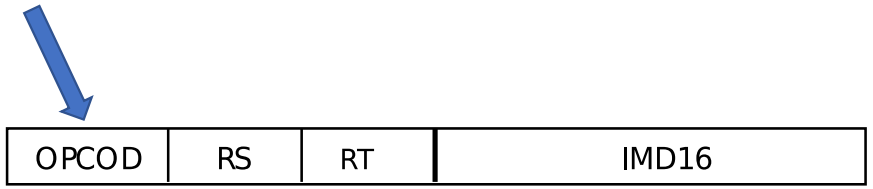
DECODAGE OPCOD

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Codage des instructions : principe

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Code opération donné par une table de correspondance



Format I

Numéro du registre codé en binaire (sur 5 bits)
Exemple :
$3→00011 et $15→01111

Immédiat encodé sur 16 bits
Exemple :
-1→0xFFFF et 3→0x0003

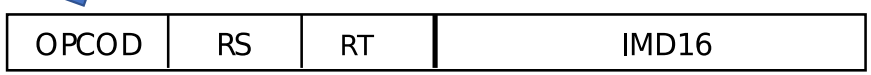
Codage des instructions : exemple

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Exemple : addiu $15, $3, -1

Instruction addiu Rt, Rs, Imm

Code opération donné par une table de correspondance



Format I

Numéro du registre codé en binaire (sur 5 bits)
Exemple :
$3→00011 et $15→01111

Immédiat encodé sur 16 bits
Exemple :
-1→0xFFFF et 3→0x0003

Codage des instructions : exemple

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Exemple : addiu $15, $3, -1

Instruction addiu Rt, Rs, Imm

Codop = addiu →

Rt = $15 →

Rs = $3 →

Imm = -1 →

Code opération donné par une table de correspondance

Format I

Numéro du registre codé en binaire (sur 5

Immédiat encodé sur 16
Exemple :
-1→0xFFFF et 3→0x0003

Codage des instructions : exemple

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Exemple : addiu $15, $3, -1

Instruction addiu Rt, Rs, Imm

Codop → 001001

Rt = $15 → 01111

Rs = $3 → 00011

Imm = -1 → 1111 1111 1111 1111

Code opération donné par une table de correspondance

Format I

Numéro du registre codé en binaire (sur 5

Immédiat encodé sur 16
Exemple :
-1→0xFFFF et 3→0x0003

Codage des instructions : exemple

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Exemple : addiu $15, $3, -1

Instruction **addiu Rt, Rs, Imm**

Codop → 001001

Rt = $15 → **01111**

Rs = $3 → **00011**

Imm = -1 → **1111 1111 1111 1111**

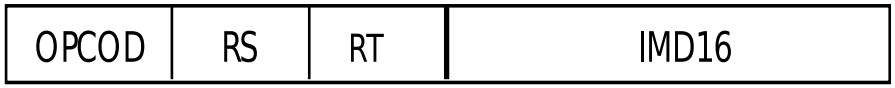
Encodage binaire :

001001 00011 01111 1111 1111 1111 1111

Encodage hexadécimal:

0x246FFFFF

Code opération donné par une table de correspondance



Format I

Numéro du registre codé en binaire (sur 5 bits)
Exemple :
$3→00011 et $15→01111

Immédiat encodé sur 16 bits
Exemple :

Décodage d'une instruction

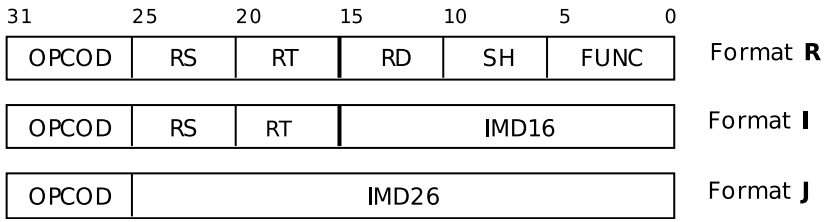
Exemple

- Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101.
- OPCOD =
- Format =
- Champs restants =

DECODAGE OPCOD

INS 28 : 26

INS 31 : 29	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								



Décodage d'une instruction

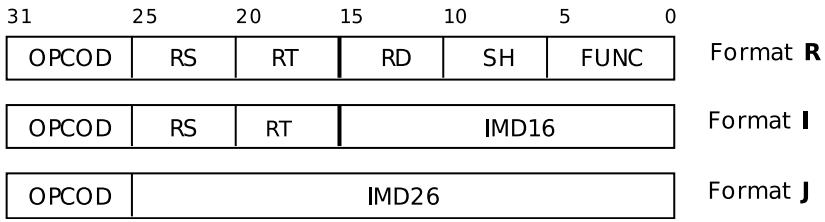
Exemple

- Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101
- OPCOD = 001000 soit un addi
- Format =
- Champs restants =

DECODAGE OPCOD

INS 28 : 26

INS 31 : 29	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								



Décodage d'une instruction

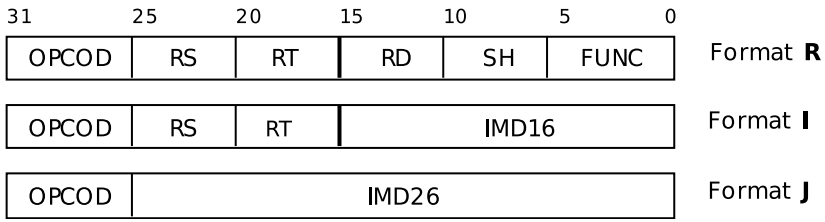
Exemple

- Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101
- OPCOD = 001000 soit un addi
- Format = I, l'instruction est de la forme `addi Rt, Rs, Imd16`.
- Champs restants =

DECODAGE OPCOD

INS 28 : 26

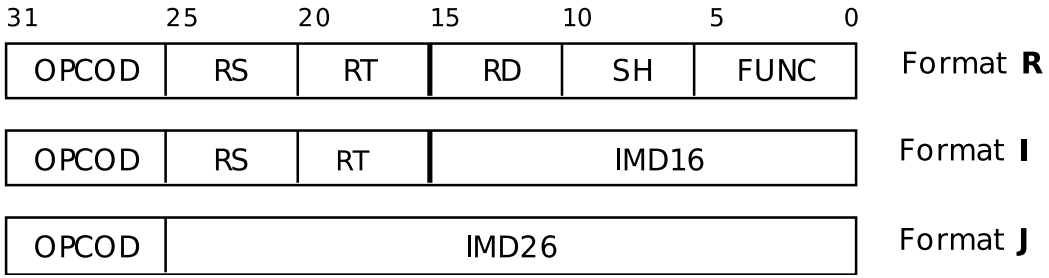
INS 31 : 29	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								



Décodage d'une instruction

Exemple

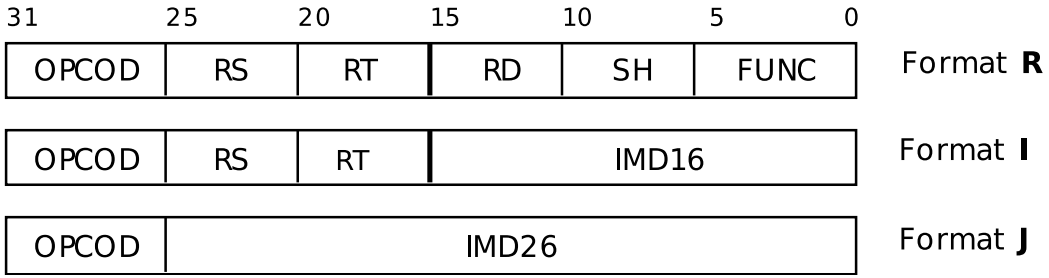
- Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101
- OPCOD = 001000 soit un addi
- Format = I, l'instruction est de la forme `addi Rt, Rs, Imd16`.
- Champs restants =
  - les bits 21 à 25 correspondent à `Rs`,
  - les bits 16 à 20 correspondent à `Rt`
  - les bits 15 à 0 à `Imd16` un immédiat encodé sur 16 bits.



Décodage d'une instruction

Exemple

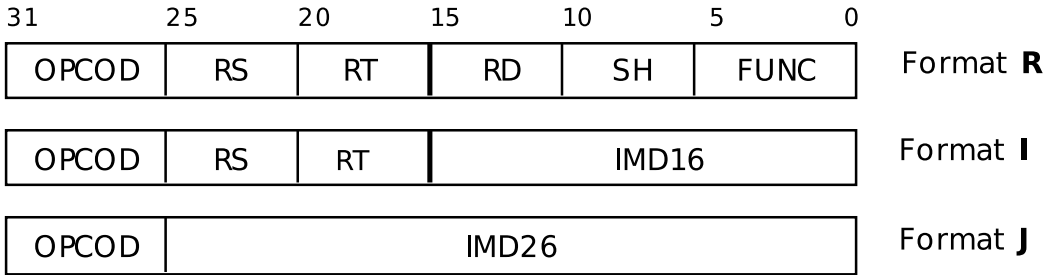
- Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101
- OPCOD = 001000 soit un addi
- Format = I, l'instruction est de la forme `addi Rt, Rs, Imd16`
- Champs restants =
  - les bits 21 à 25 correspondent à `Rs=$3`,
  - les bits 16 à 20 correspondent à `Rt=$2`
  - les bits 15 à 0 correspondent à `Imd16` encodé sur 16 bits qui vaut 0xABCD



Décodage d'une instruction

Exemple

- Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101
- OPCOD = 001000 soit un addi
- Format = I, l'instruction est de la forme `addi Rt, Rs, Imd16`
  - les bits 21 à 25 correspondent à `Rs=$3`,
  - les bits 16 à 20 encodent 2, donc `Rt=$2`
  - les bits 15 à 0 encodent 0x0ABCD donc `Imd16` vaut 0xABCD
- l'instruction est `addi $2, $3, 0xABCD`



Code opération versus code spécial

Codop SPECIAL

- L'opcod SPECIAL correspond au format R et à une famille d'opérations précisées dans le champ FUNC (bits 5 à 0)
- Le codage de l'opération est alors donnée par une seconde table, celle qui donne le codage du champ FUNC
- Par exemple si INS[31:25]=000000 et INS[5:0]=100101, l'opération est un `or`

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5 : 3 \ INS 2 : 0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Décodage d'une instruction

Exemple

- Soit 0x00641020 = 0000 0000 0110 0100 0001 0000 0010 0000.
- OPCOD =
- Format =
- Champs restants =

DECODAGE OPCOD

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Codage d'une instruction

Instruction `or $4, $2, $1`

- OPCOD =
- Format (voir sur le mémento) =
- Champs restants =
- Codage binaire =
- Codage en hexadécimal =

DECODAGE OPCOD

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Codage d'une instruction

Instruction `or $4, $2, $1`

- OPCOD = SPECIAL = 000000 car `or` est dans la 2ème table, valeur 100101_b
- Format (cf. mémento) =
- Champs restants =
- Codage binaire =
- Codage en hexadécimal =

DECODAGE OPCOD

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

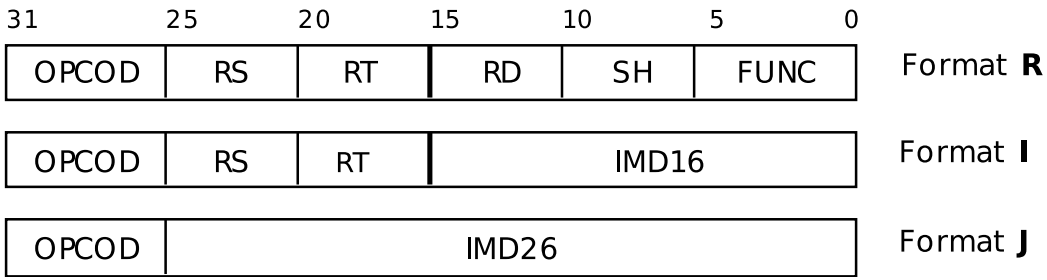
OPCOD = SPECIAL

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Codage d'une instruction

Instruction `or $4, $2, $1`

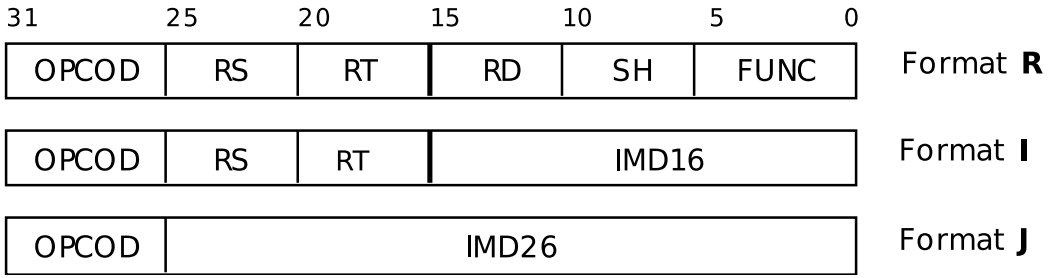
- OPCOD = SPECIAL = 000000 car `or` est dans la 2ème table, valeur 100101_b
- Format (cf. le mémento) : `or Rd, Rs, Rt`
  Format R : OPCODE Rs Rt Rd Sh FUNC
- Champs restants :
- Codage binaire :
- Codage en hexadécimal :



Codage d'une instruction

Instruction `or $4, $2, $1`

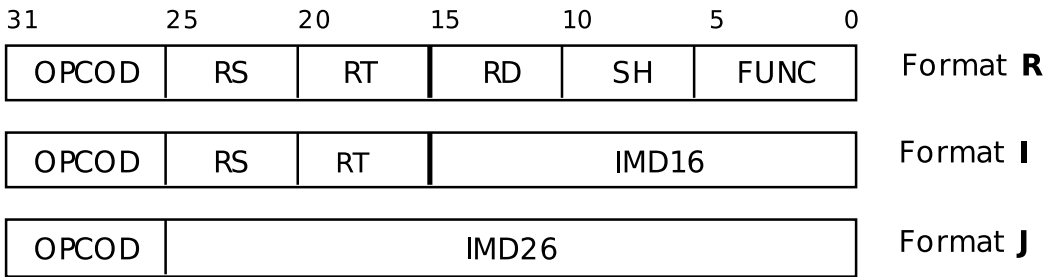
- OPCOD = SPECIAL = 000000 car `or` est dans la 2ème table, valeur 100101_b
- Format (cf. mémento) = `or Rd, Rs, Rt`
  Format R : OPCODE Rs Rt Rd Sh FUNC
- Champs restants : Rd = 4 = 00100_b, Rs = 2 = 00010_b,
  Rt = 1 = 00001_b, Sh = vide = 00000_b, FUNC = 100101_b
- Codage binaire :
- Codage en hexadécimal :



Codage d'une instruction

Instruction `or $4, $2, $1`

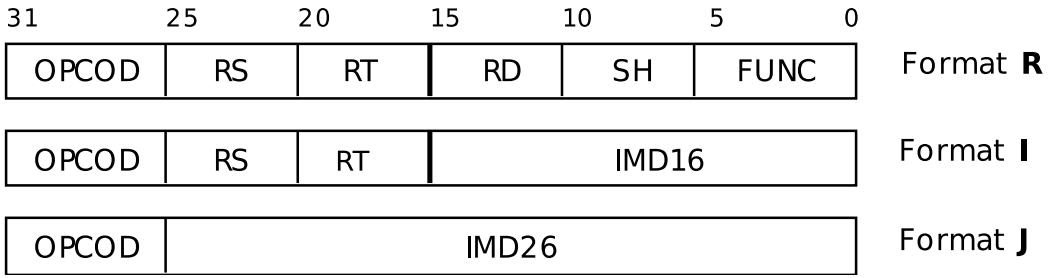
- OPCOD = SPECIAL = 000000 car `or` est dans la 2ème table, valeur 100101_b
- Format (cf. mémento) = `or Rd, Rs, Rt`
  Format R : OPCODE Rs Rt Rd Sh FUNC
- Champs restants : Rd = 4 = 00100_b, Rs = 2 = 00010_b,
  Rt = 1 = 00001_b, Sh = vide = 00000_b, FUNC = 100101_b
- Codage binaire : 000000 00010 00001 00100 00000 100101
- Codage en hexadécimal :



Codage d'une instruction

Instruction `or $4, $2, $1`

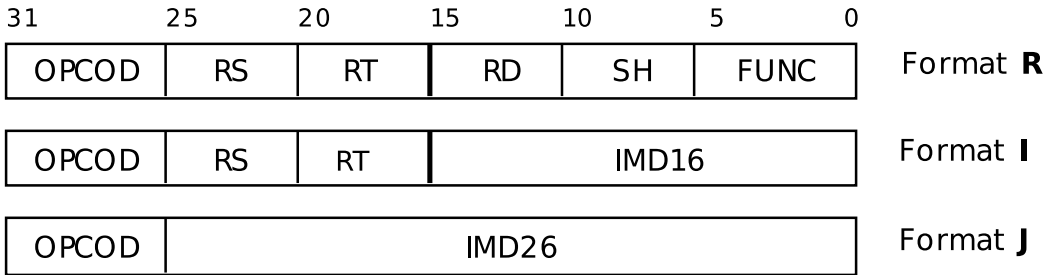
- OPCOD = SPECIAL = 000000 car `or` est dans la 2ème table, valeur 100101_b
- Format (cf. mémento) = `or Rd, Rs, Rt`
  Format R : OPCODE Rs Rt Rd Sh FUNC
- Champs restants : Rd = 4 = 00100_b, Rs = 2 = 00010_b,
  Rt = 1 = 00001_b, Sh = vide = 00000_b, FUNC = 100101_b
- Codage binaire : 000000 00010 00001 00100 00000 100101
  vue quartet = 0000 0000 0100 0001 0010 0000 0010 0101
- Codage en hexadécimal :



Codage d'une instruction

Instruction `or $4, $2, $1`

- OPCOD = SPECIAL = 000000 car `or` est dans la 2ème table, valeur 100101_b
- Format (cf. mémento) = `or Rd, Rs, Rt`
  Format R : OPCODE Rs Rt Rd Sh FUNC
- Champs restants : Rd = 4 = 00100_b, Rs = 2 = 00010_b,
  Rt = 1 = 00001_b, Sh = vide = 00000_b, FUNC = 100101_b
- Codage binaire : 000000 00010 00001 00100 00000 100101
  vue quartet = 0000 0000 0100 0001 0010 0000 0010 0101
- Codage en hexadécimal : 0x00412025



Programme : les différents niveaux

Niveaux de programme

Il existe différents niveaux de programmation qui correspondent aux différents niveaux d'abstraction des traitements à exécuter

- Programme de haut niveau (python, CAML, Java, C, C++)
- Programme en langage d'assemblage/assembleur (MIPS, Armv7, x86, ...)
- Programme binaire

Composition des niveaux de programmes

Haut niveau

- Instruction du langage $\forall$ ISA cible
- Definition d'objets structurés
- Définition de variables nommées utilisables dans les instructions
- Structuration des traitements
- Gestion d'erreurs

Assembleur

- Allocation des données et gestion mémoire
- Suite d'instructions assembleur
- Présence d'étiquettes pour désigner les adresses (données ou instructions)

Binaire

- Suite d'instructions en langage machine

Programme : les différents types d'exécution

Interprétation versus exécution native

Un programme *P* de haut niveau peut être soit :

- compilé en un binaire exécutable : il sera directement exécuté sur la machine cible. On parle d'exécution native.
- interprété par un programme appelé *interpréteur* : c'est l'interpréteur qui s'exécute nativement sur la machine et réalise les traitements dictés par le programme *P* (Python, machine virtuelle Java, ...).
  Le format (source, bytecode, binaire, ...) du programme *P* à donner à l'interpéteur dépend de l'interpréteur.

On s'intéresse aux programmes natifs, exécutables et exécutés par la machine cible, ainsi qu'à la compréhension des principes d'exécution d'un programme natif.

Traduction de programmes natifs

Programme haut niveau $\rightarrow$ programme binaire

- La traduction entre programme de haut niveau et langage binaire est réalisée par un compilateur (par ex. GNU gcc, clang/LLVM, arm-cc, ...).
- Les instructions de haut niveau sont traduites en une suite d'instructions binaires.
- Les registres et les instructions du processeur cibles sont utilisés pour réaliser les traitements.

Programme assembleur $\leftrightarrow$ programme binaire exécutable

- On peut arrêter la compilation au niveau assembleur ou écrire des programme directement en assembleur.
- La traduction assembleur-binaire, appelée *assemblage*, est réalisée par un programme appelé *assembleur*.
- On peut aussi à partir d'un binaire retrouver une représentation assembleur d'un programme, on appelle cette operation *désassemblage* réalisée par un *désassembleur*

Exemple

Programme C	ASM	Binaire
`{`	`...`	
`...`	`ori $4, $0, 2`	`0x34040002`
`a = 2;`	`ori $5, $0, 3`	`0x34050003`
`b = 3;`	`add $6, $5, $4`	`0x00A43020`
`c = (a + b) * 2;`	`sll $6, $6, 1`	`0x00063020`
`...`	`...`	
`}`		

Utilisation des registres

En assembleur, on utilise des registres pour stocker les valeurs des variables (simples) et leur adresse dans le processeur ainsi que pour conserver les résultats de calculs intermédiaires.

Exemple

ASM	Binaire	Desassemblé
Instructions assembleur et présence d'étiquettes	Instructions converties en langage machine	Instructions assembleur. Plus d'étiquettes.

```
      beq  $5, $0, fin      0x10A00003    beq  $5, $0, 3
etiq: andi $4, $4, 0x0FF0   0x30840FF0    andi $4, $4, 4080
      addi $4, $4, -1       0x2084FFFF    addi $4, $4, -1
      j    etiq             0x08100001    j    4194308
fin:  add  $8, $4, $6       0x00864020    add  $8, $4, $6
```

Etiquettes

- Une étiquette est de la forme **etiq :**.
- Dans ce cas, son nom est *etiq*.
- La sémantique d'une étiquette est "adresse de ce qui suit".
- Elle permet de désigner des adresses dans le code assembleur, que ce soit de données ou d'instructions.
- Son nom peut ensuite être utilisé dans certaines instructions.

Programmation assembleur

- Écriture de programmes directement en assembleur.
- Description des traitements à réaliser avec les instructions du jeu d'instructions du processeur cible.
- Allocation et initialisation des données (prochains cours)
- Respect de conventions liées au processeur cible.

Structure d'un programme assembleur MIPS

- En MIPS, tout programme assembleur est constitué de 2 sections :
  - la section de données
  - la section de code

Section de code : directive `.text`

- La directive `.text` désigne la section de code.
- Les instructions doivent se trouver après cette directive dans un programme assembleur.

Section de données : directive `.data`

- La directive `.data` désigne la section de données.
- Les données globales doivent être allouées, et initialisées si besoin, dans cette section du programme (voir cours 4)

Structure d'un code assembleur

```
.data

 # section de données

    # allocation et initialisation d'emplacements mémoire
    # pour les données globales


.text

 # section de code

   # description des traitements àréaliser
   # utilisation des instructions du jeu d'instructions
```

Exemple

Quelques instructions dans la section de code :

```
.data

.text
   addi $3, $0, 10
   addi $4, $0, 0x0F
   add  $4, $4, $3
   ...
```

Assemblage d'un programme

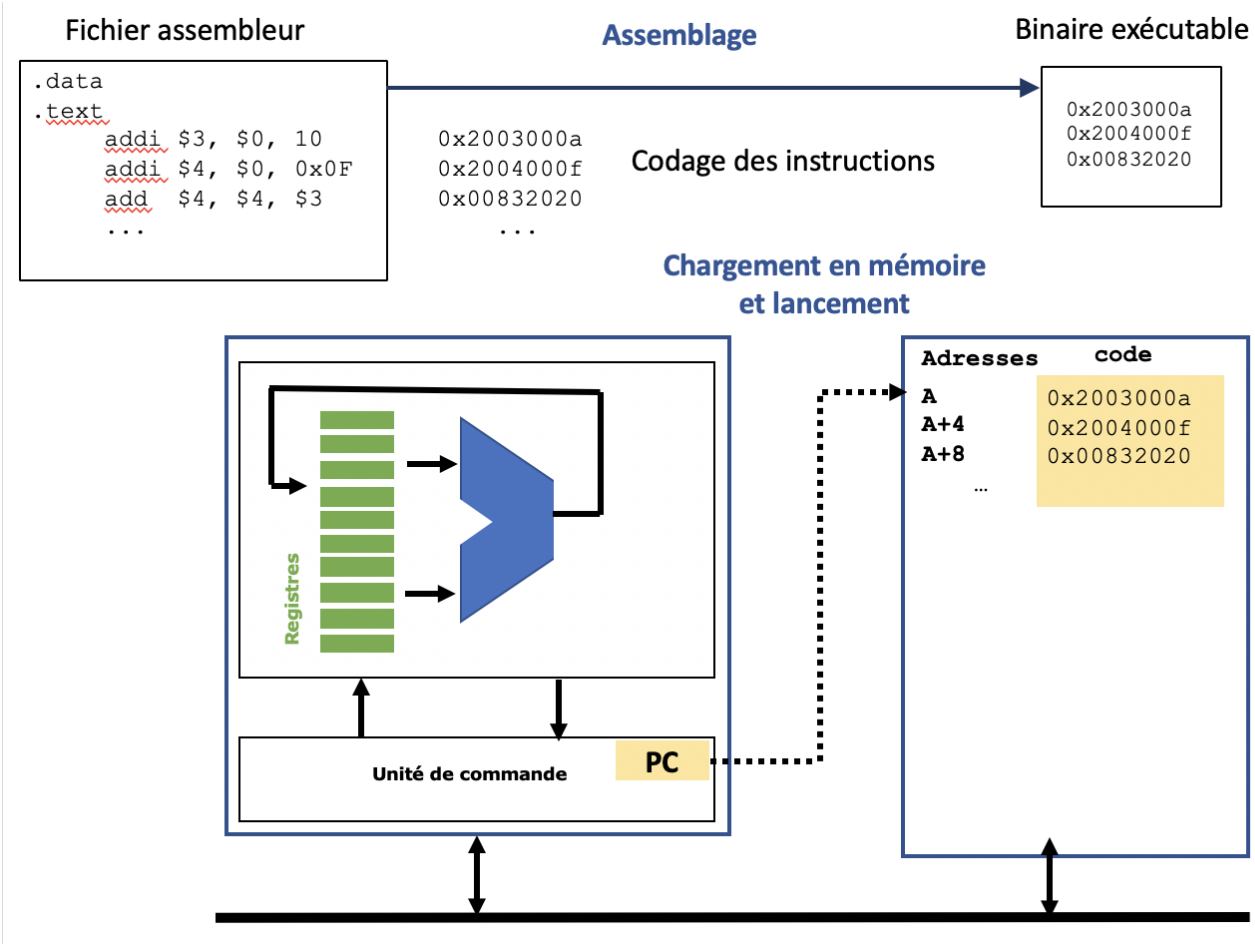
- Une fois un programme assembleur écrit, il faut l'assembler afin de créer un programme binaire exécutable.
- Les instructions sont codées en binaire dans leur ordre d'apparition et rangées consécutivement dans le programme binaire exécutable.

```
.data

.text                  #  codage binaire | adresse
    addi $3, $0, 10    #  0x2003000a     | A = adresse de la première instruction
    addi $4, $0, 0x0F  #  0x2004000f     | A + 4
    add $4, $4, $3     #  0x00832020     | A + 8
    ...
```

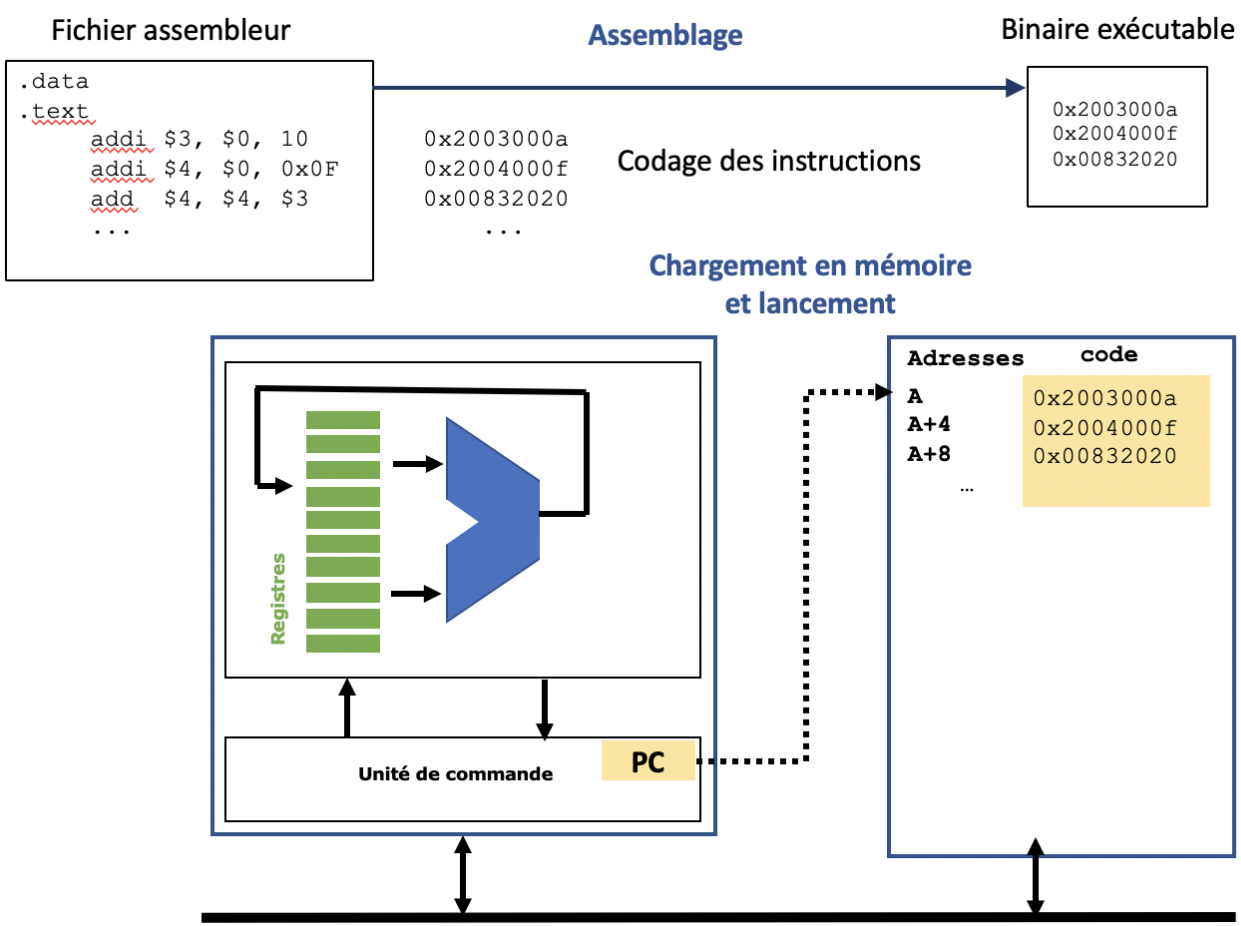
Lancement d'un programme

- Pour exécuter un programme, il faut le "lancer".
- Cette opération, charge le programme en mémoire (en général par un *loader*) :



Lancement d'un programme

- L'adresse du point d'entrée du programme est mise dans le registre PC, ce qui engendre l'exécution de la première instruction... et donc du programme.
- Pour l'instant, dans ce cours, la première adresse de la section de code correspond au point d'entrée du programme.



Exécution et terminaison d'un programme assembleur

- Lors du lancement d'un programme, l'adresse du point d'entrée du programme est mise par le loader dans le registre PC (A sur l'exemple), la première instruction est alors lue en mémoire, puis exécutée.
- L'exécution continue ensuite séquentiellement (avec parfois des sauts lors de l'exécution d'une instruction spécifiant l'adresse de la prochaine instruction différente de celle qui suit dans le programme asm).
- Le programme s'arrête avec un appel système demandant la terminaison du programme.

Appel système (en MARS)

Quoi et comment ?

- Un appel système est une demande de service fourni par le système.
- C'est l'instruction `syscall` qui réalise cette demande de service.
- En MIPS, chaque service a un numéro
- Pour utiliser un service, il faut mettre le numéro correspondant dans le registre `$2` avant d'exécuter l'instruction `syscall`.

Quelques appels système et leur numéro

- Terminaison d'un programme : numéro 10.

```
ori     $2, $0, 10     # mettre 10 dans $2
syscall                # demande d'appel système
```

- Affichage d'un entier : numéro 1.
  Il faut aussi mettre l'entier à afficher dans le registre `$4` avant l'appel.

Exemple précédent avec appel de fin de programme

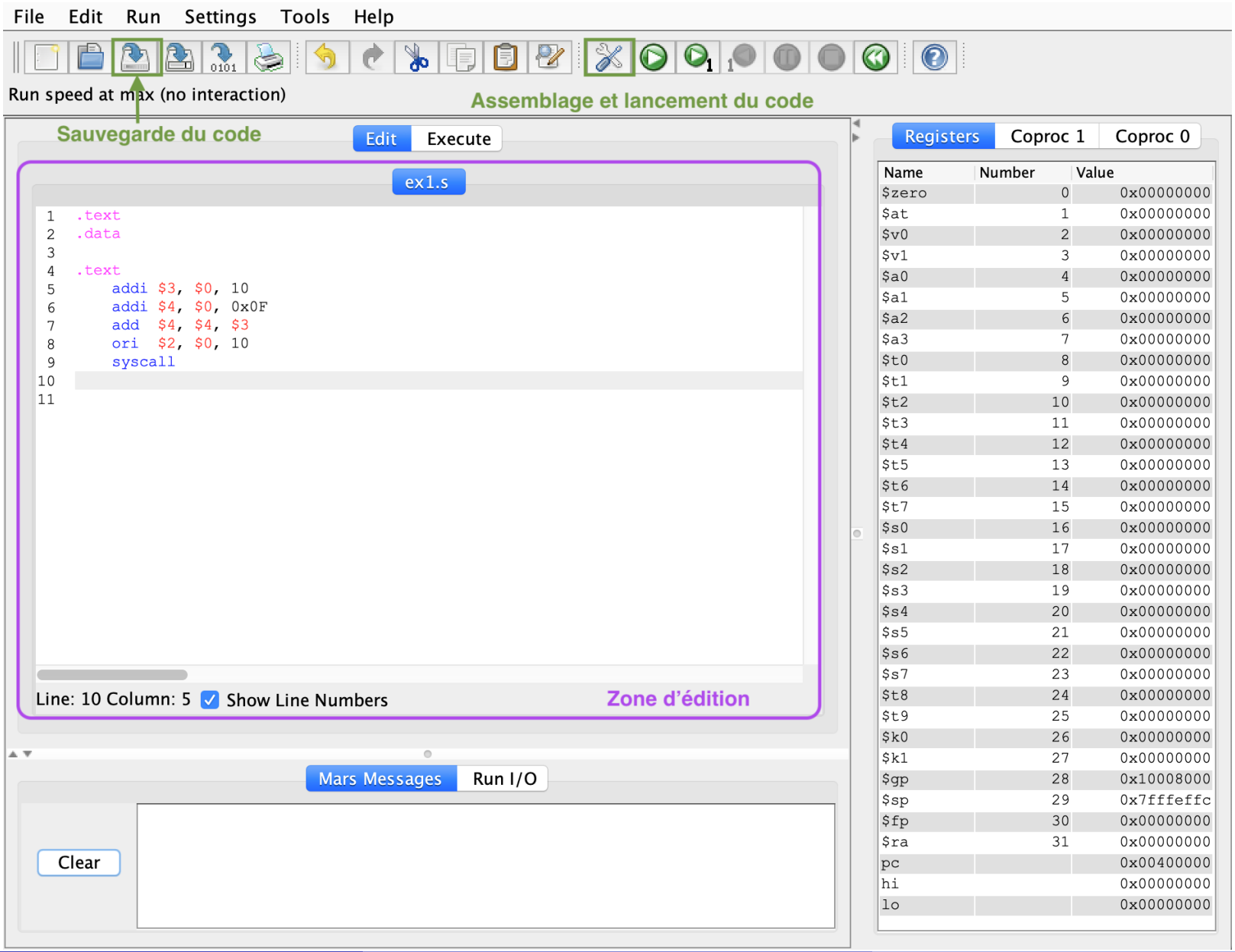
```
.data

.text

    addi $3, $0, 10    # inst 0x2003000a adresse A
    addi $4, $0, 0x0F  # inst 0x2004000f adresse A + 4
    add $4, $4, $3     # inst 0x00832020 adresse A + 8
    ori $2, $0, 10     # inst 0x3402000a adresse A + 12
    syscall            # inst 0x0000000c adresse A + 16
```

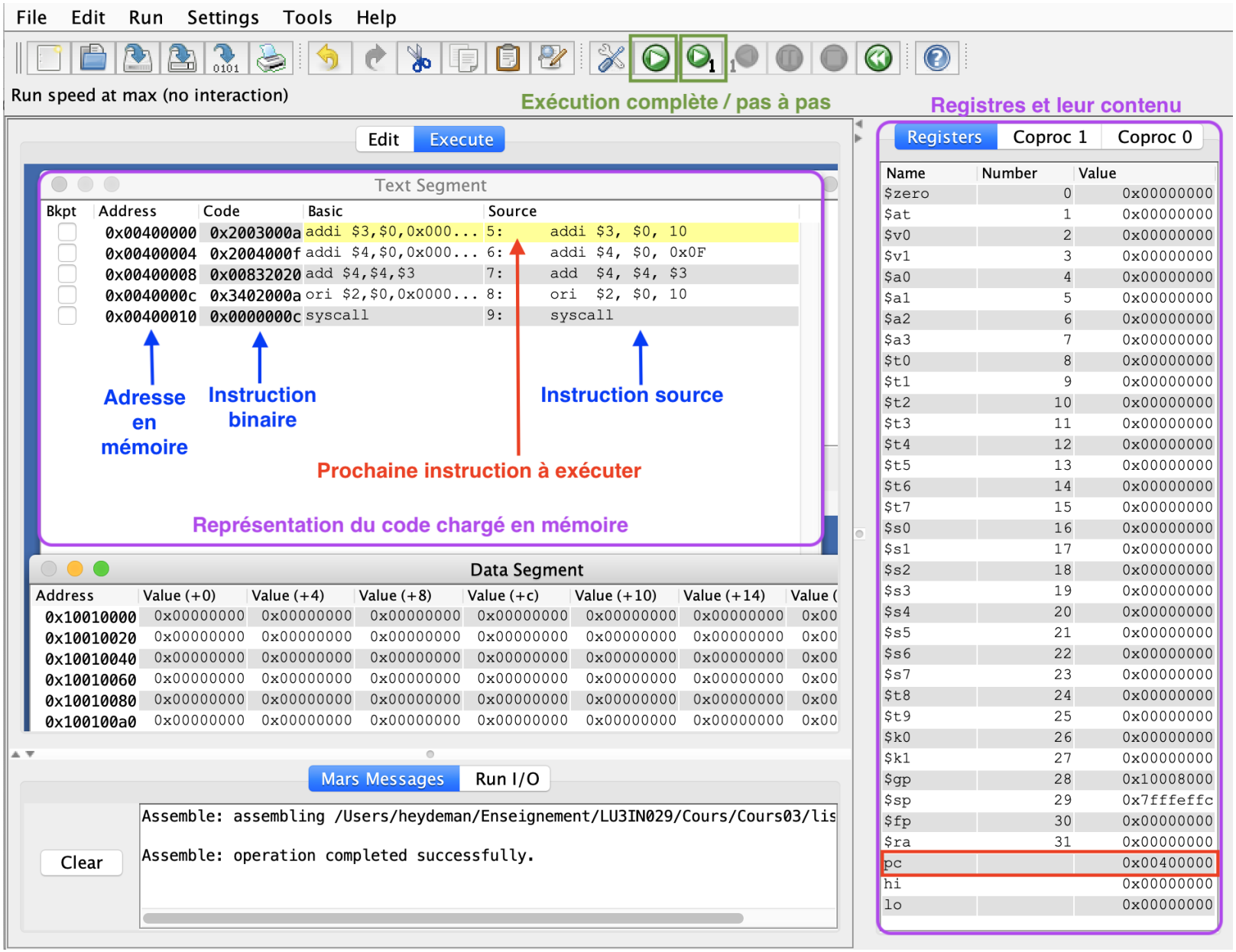
Simulateur Mars

Ecriture du programme exemple



Expérimentations avec Mars

Assemblage et chargement du programme assemblé



Exemple de programmes assembleur

- Programme qui affiche la valeur 2 et termine.
- Programme assembleur qui lit un entier au clavier, lui ajoute 10, l'affiche et termine

Afficher 2 et terminer

- Afficher 2 = service d'affichage d'un entier
- Afficher un entier = appel système numéro 1 + entier à afficher dans `$4`
- Terminer = service de terminaison de programme
- Terminer = appel système numéro 10

```
.data
.text
  # afficher l'entier 2

   ori $2, $0, 1  # mettre 1 dans $2
   ori $4, $0, 2  # mettre l'entier àafficher dans $4
   syscall        # demande d'appel système

  # terminer le programme
   ori $2, $0, 10 # mettre 10 dans $2
   syscall        # demande d'appel systeme
```

Afficher 2 et terminer

- Afficher 2 = service d'affichage d'un entier
- Afficher un entier = appel système numéro 1 + entier à afficher dans `$4`
- Terminer = service de terminaison de programme
- Terminer = appel système numéro 10

```
.data
.text
  # afficher l'entier 2

   ori $2, $0, 1  # mettre 1 dans $2
   ori $4, $0, 2  # mettre l'entier àafficher dans $4
   syscall        # demande d'appel système

  # terminer le programme
   ori $2, $0, 10 # mettre 10 dans $2
   syscall        # demande d'appel systeme
```

Lire un entier, lui ajouter 10, l'afficher et terminer

- Lire un entier = service de lecture d'un entier (numero 5)
  la valeur lue sera dans `$2` juste après l'appel système
- Ajouter 10 = instruction d'addition avec un immédiat (10)
- Afficher un entier = appel système numéro 1 + entier à afficher dans `$4`
- Terminer = service de terminaison de programme (appel système numéro 10)

```
.data
.text
   # lire un entier au clavier
   ori $2, $0, 5 # mettre 5 dans $2
   syscall       # demande d'appel système
                 # entier lu dans $2
   # ajouter 10 àcet entier
   addi $3, $2, 10 # ajouter 10 àl'entier lu

  # afficher le résultat
   ori $2, $0, 1  # mettre 1 dans $2
   or $4, $0, $3  # mettre l'entier àafficher dans $4
   syscall        # demande d'appel système

  # terminer le programme
   ori $2, $0, 10 # mettre 10 dans $2
   syscall        # demande d'appel systeme
```

Lire un entier, lui ajouter 10, l'afficher et terminer

- Lire un entier = service de lecture d'un entier (numero 5)
  la valeur lue sera dans `$2` juste après l'appel système
- Ajouter 10 = instruction d'addition avec un immédiat (10)
- Afficher un entier = appel système numéro 1 + entier à afficher dans `$4`
- Terminer = service de terminaison de programme (appel système numéro 10)

```
.data
.text
   # lire un entier au clavier
   ori $2, $0, 5  # mettre 5 dans $2
   syscall        # demande d'appel système
                  # entier lu dans $2
   # ajouter 10 àcet entier
   addi $3, $2, 10 # ajouter 10 àl'entier lu

  # afficher le résultat
   ori $2, $0, 1   # mettre 1 dans $2
   or $4, $0, $3   # mettre l'entier àafficher dans $4
   syscall         # demande d'appel système

 # terminer le programme
   ori $2, $0, 10 # mettre 10 dans $2
   syscall        # demande d'appel systeme
```

Autres exemples / exercices

- Lire un entier, calculer sa parité et l'afficher
- Lire un entier, calculer son opposé, l'afficher
- Lire un caractère minuscule, calculer sa version en majuscule l'afficher
- Lire deux entiers, les additionner, afficher le résultat
- Lire un entier positif, le multiplier par 3 puis afficher le résultat
- Lire un entier, le multilplier par 4 puis afficher le résultat
- ...

Ce qu'on a vu

- Notion de jeu d'instructions
- Codage / décodage d'instruction
- Tour des instructions arithmétiques et logiques
- Appels systèmes (pour Mars)
- Programmation assembleur de petits programmes
- Simulation avec Mars