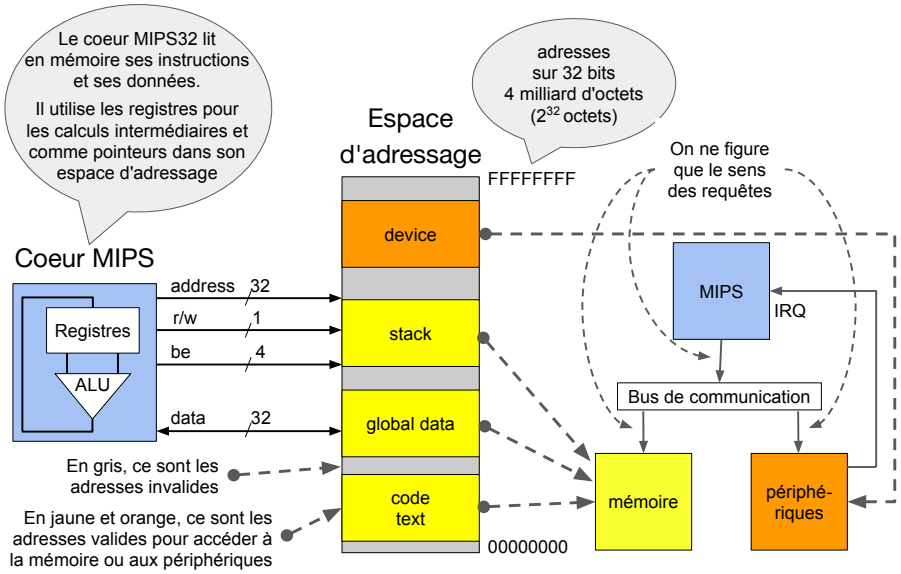
ALMO

Architecture Externe MIPS32
Assembleur

plan

- Modèle d'ordinateur
- Processeur du MIPS
- Registres externes
- Instructions
- Accès à la mémoire (à l'espace d'adressage)
- Principe de l'exécution d'une instruction
- Programmation assembleur
- Présentation de l'application simulateur MARS

Modèle d'ordinateur



Processeur MIPS32

- Type RISC (Reduced Instruction Set Computer) : 57 instructions
- Le MIPS32 manipule des adresses 32 bits, chaque adresse désigne un octet.
- Toutes les instructions sont codées sur 1 mot (4 octets) aligné en mémoire.
- Le MIPS lit jusqu'à une instruction par cycle (Clock Per Instruction CPI = 1)
- Les instructions (arithmétiques et logiques) utilisent seulement les registres.
- L'accès à l'espace d'adressage permet seulement les lectures et écritures.
- Le MIPS dispose de deux modes d'exécution : USER et KERNEL
- En mode USER, certaines instructions et certaines adresses sont interdites
- 6 IRQ (Interrupt ReQuest) en provenance des périphériques matériels
- 1 reset qui démarre le processeur à l'adresse 0xBFC00000

Registres

L'architecture externe d'un processeur correspond
aux registres et aux instructions utilisables par le programmeur.

- Registres externes USER
  - $0—$31 : 32 GPR (General Purpose Register) **$0 est égale à 0**
  - HI / LO : sortie du multiplieur et du diviseur (High / Low)
  - PC : pointeur de programme

- Registres externes@ KERNEL (*nous verrons le détail plus tard*)
  - SR : Status Register (p. ex. un bit de mode USER / KERNEL)
  - CR : Cause Register (contient la cause d'appel au système)
  - EPC : Exception PC (contient l'adresse de retour du système)
  - BAR : Bad Address Register (contient une adresse incorrecte)
  - PROCID : numéro du core (utile lorsqu'il y en a plusieurs)
  - CYCLES : nombre de cycles écoulés depuis le reset

Instructions

- Toutes les instructions sont sur un mot de 4 octets (32 bits)
- Les instructions sont alignées en mémoire

  On dit qu'un objet mémoire (un mot, un tableau, une structure) est alignée en mémoire si son adresse (l'adresse de son premier octet) est un multiple de sa taille.

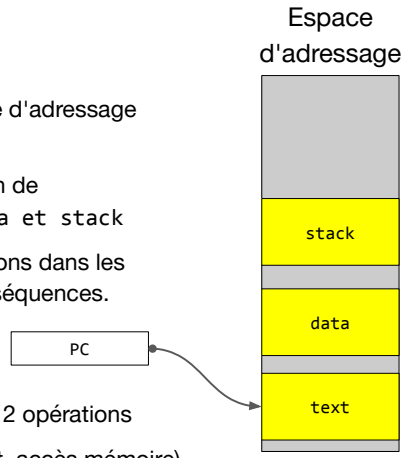
- Une instruction est une opération élémentaire de forme générale

  `coop operands, ...`

- Les opérandes sont des numéros de registres ou des valeurs immédiates (une valeur immédiate est codée dans l'instruction elle-même)
- Il y existe 3 formats d'instruction :
  - Format R : 2 registres source et 1 registre résultat
  - Format I : 1 registre et une valeur immédiate source et 1 registre résultat
  - Format J : branchement inconditionnel, saut à une adresse

Programme

- Un segment est une zone de l'espace d'adressage dont les adresses sont consécutives.
- Un programme qui s'exécute a besoin de 3 segments de mémoire : `text, data et stack`
- Le code est une séquence d'instructions dans les segment `text` avec des ruptures de séquences.
- Le registre PC pointe sur l'instruction en cours d'exécution
- Chaque instruction exécute au moins 2 opérations
  1. Une opération propre (calcul, test, accès mémoire)
  2. Calcul du prochain PC, qui est par défaut PC+4



Les 4 Types d'Instructions du MIPS

1. Instructions arithmétiques et logiques
   - *coop rd, rs, rt* `# rd ← rs coop rt`
   - *coop rd, rs, imm* `# rd ← rs coop imm`
   - **`add $10, $5, $3`** `# $10 ← $5 + $3`
2. Instructions de lecture et d'écriture
   - *load rt, displacement (rs)* `# rt ← memoire( rs + déplacement )`
   - *store rt, displacement (rs)* `# memoire( rs + déplacement ) ← rt`
   - **`lw $10, 8 ($29)`** `# $10 ← MEM [ $29 + 8 ]`
3. Les instructions de branchement
   - *goto adresse* `# PC ← adresse`
   - *goto condition, adresse* `# si (condition) PC ← adresse`
   - **`beq $10, $5, plusloin`** `# if ( $10 == $5) goto plusloin`
4. Instructions système
   - *appel du système et retour*
   - *accès au registres des coprocesseurs*
   - **`mfc0 $4, $14`** `# $4 (des GPR) ← $14 (du copro 0)`

Aide mémoire ALMO
Jeu d'instructions MIPS

Instructions Arithmétiques/Logiques entre registres

Assembleur		Opération		Format
Add	Rd, Rs, Rt	Add (overflow detection)	Rd <- Rs + Rt	R
Sub	Rd, Rs, Rt	Subtract (overflow detection)	Rd <- Rs - Rt	R
Addu	Rd, Rs, Rt	Add (no overflow)	Rd <- Rs + Rt	R
Subu	Rd, Rs, Rt	Subtract (no overflow)	Rd <- Rs - Rt	R
Addi	Rt, Rs,I	Add Immediate (overflow detection)	Rt <- Rs + I	I
Addiu	Rt, Rs,I	Add Immediate (no overflow)	Rt <- Rs + I	I
Or	Rd, Rs, Rt	Logical Or	Rd <- Rs or Rt	R
And	Rd, Rs, Rt	Logical And	Rd <- Rs and Rt	R
Xor	Rd, Rs Rt	Logical Exclusive-Or	Rd <- Rs xor Rt	R
Nor	Rd, Rs, Rt	Logical Not Or	Rd <- Rs nor Rt	R
Ori	Rt, Rs, I	Or Immediate (unsigned immediate)	Rt <- Rs or I	I
Andi	Rt, Rs, I	And Immediate (unsigned immediate)	Rt <- Rs and I	I
Xori	Rt, Rs, I	Exclusive-Or Immediate (unsigned immediate)	Rt <- Rs xor I	I
Sllv	Rd, Rt, Rs	Shift Left Logical Variable (5 lsb of Rs is significant	Rd <- Rt << Rs	R
Srlv	Rd, Rt, Rs	Shift Right Logical Variable (5 lsb of Rs is significant	Rd <- Rt >> Rs	R
Srav	Rd, Rt, Rs	Shift Right Arithmetical Variable (5 lsb of Rs is significant	Rd <- Rt >>* Rs	R
Sll	Rd, Rt, sh	Shift Left Logical	Rd <- Rt << sh	R
Srl	Rd, Rt, sh	Shift Right Logical	Rd <- Rt >> sh	R
Sra	Rd, Rt, sh	Shift Right Arithmetical	Rd <- Rt >>* sh (* : with sign extension)	R
Lui	Rt, I	Load Upper Immediate (16 lower bits of Rt are set to zero	Rt <- I || "0000"	I

Instructions Arithmétiques/Logiques (suite)

Assembleur		Opération		Format
Slt	Rd, Rs, Rt	Set if Less Than	Rd <- 1 if Rs<Rt else 0	R
Sltu	Rd, Rs, Rt	Set if Less Than Unsigned	Rd <- 1 if Rs<Rt else 0	R
Slti	Rt, Rs, I	Set if Less Than Immediate (sign extended Immediate	Rt <- 1 if Rs < I else 0	I
Sltiu	Rt, Rs, I	Set if Less Than Immediate (unsigned immediate	Rt <- 1 if Rs < I else 0	I
Mult	Rs, Rt	Multiply	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Multu	Rs, Rt	Multiply Unsigned	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Div	Rs, Rt	Divide	Rs / Rt LO <- Quotient HI <- Remainder	R
Divu	Rs, Rt	Divide Unsigned	Rs / Rt LO <- Quotient HI <- Remainder	R
Mfhi	Rd	Move From HI	Rd <- HI	R
Mflo	Rd	Move From LO	Rd <- LO	R
Mthi	Rs	Move To HI	HI <- Rs	R
Mtlo	Rs	Move To LO	LO <- Rs	R

Instructions de lecture/écriture mémoire

Assembleur		Opération		Format
Lw	Rt, I (Rs)	Load Word (sign extended Immediate	Rt <- M (Rs + I)	I
Sw	Rt, I (Rs)	Store Word (sign extended Immediate	M (Rs + I) <- Rt	I
Lh	Rt, I (Rs)	Load Half Word (sign extended Immediate. Two bytes from storage is loaded into the 2 less significant bytes of Rt. The sign of these 2 bytes is extended on the 2 most signficant bytes.	Rt <- M (Rs + I)	I
Lhu	Rt, I (Rs)	Load Half Word Unsigned (sign extended Immediate. Two bytes from storage is loaded into the the 2 less significant bytes of Rt, other bytes are set to zero	Rt <- M (Rs + I)	I
Sh	Rt, I (Rs)	Store Half Word (sign extended Immediate. The Two less significant bytes of Rt are stored into storage	M (Rs + I) <- Rt	I
Lb	Rt, I (Rs)	Load Byte (sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt. The sign of this byte is extended on the 3 most signficant bytes.	Rt <- M (Rs + I)	I
Lbu	Rt, I (Rs)	Load Byte Unsigned (sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt, other bytes are set to zero	Rt <- M (Rs + I)	I
Sb	Rt, I (Rs)	Store Byte (sign extended Immediate. The less significant byte of Rt is stored into storage	M (Rs + I) <- Rt	I

Instructions de Branchement

Assembleur		Opération		Format
Beq	Rs, Rt, Label	Branch if Equal	PC <- PC+4+(I*4) if Rs = Rt PC <- PC+4 if Rs ≠ Rt	I
Bne	Rs, Rt, Label	Branch if Not Equal	PC <- PC+4+(I*4) if Rs ≠ Rt PC <- PC+4 if Rs = Rt	I
Bgez	Rs, Label	Branch if Greater or Equal Zero	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0	I
Bgtz	Rs, Label	Branch if Greater Than Zero	PC <- PC+4+(I*4) if Rs > 0 PC <- PC+4 if Rs ≤ 0	I
Blez	Rs, Label	Branch if Less or Equal Zero	PC <- PC+4+(I*4) if Rs ≤ 0 PC <- PC + 4 if Rs > 0	I
Bltz	Rs, Label	Branch if Less Than Zero	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0	I
Bgezal	Rs, Label	Branch if Greater or Equal Zero and link	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0 R31 <-PC+4 in both cases	I
Bltzal	Rs, Label	Branch if Less Than Zero and link	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0 R31 <- PC+4 in both cases	I
J	Label	Jump	PC <- PC 31:28 || I*4	J
Jal	Label	Jump and Link	R31 <- PC+4 PC <- PC 31:28 || I*4	J
Jr	Rs	Jump Register	PC <- Rs	R
Jalr	Rs	Jump and Link Register	R31 <- PC+4 PC <- Rs	R
Jalr	Rd, Rs	Jump and Link Register	Rd <- PC+4 PC <- Rs	R

Instructions Systèmes

Assembleur		Opération		Format
Rfe		Restore From Exception (Privileged instruction. Restore the prevous IT mask and mode	SR <- SR 31:4 || SR 5:2	R
Break	n	Breakpoint Trap (Branch to exception handler. n defines the breakpoint number	SR <- SR 31:6 || SR 3:0 || "00" PC <- "8000 0080" CR <- cause	R
Syscall		System Call Trap (Branch to exception handler	SR <- SR 31:6 || SR 3:0 || "00" PC <- "8000 0080" CR <- cause	R
Mfc0	Rt, Rd	Move From Control Coprocessor (Privileged Instruction . The register Rd of the Control Coprocessor is moved into the integer register Rt	Rt <- Rd	R
Mtc0	Rt, Rd	Move To Control Coprocessor (Privileged Instruction . The integer register Rt is moved into the register Rd of the Control Coprocessor	Rd <- Rt	R

Exécution d'une instruction

Le registre PC contient l'adresse de l'instruction à exécuter

Instruction Fetch (IF)
- Lecture de l'instruction dans le Registre Instruction (registre interne) depuis la mémoire : RI ← MEM [PC]
- PC ← PC + 4 (par défaut)

Decod (DEC)
- Décodage l'instruction
- Lecture du banc de registres et calcul des opérandes

Execute (EXEC)
- Exécution de l'instruction arithmétique et logique
- ou calcul de l'adresse à aller lire en mémoire

Memory access (MEM)
- Lecture ou écriture en mémoire

Write Back (WB)
- Ecriture du résultat dans le banc de registres

⇒ L'exécution d'une instruction se fait en cinq étapes, chacune d'un cycle

IF	DEC	EXEC	MEM	WB

Exécution en pipeline

- Le principe du pipeline consiste à commencer l'exécution de l'instruction suivante avant d'avoir fini l'exécution de l'instruction précédente
- Ainsi, à chaque cycle, le MIPS lit une nouvelle instruction.
  En apparence, le MIPS exécute une instruction à chaque cycle !

i	IF	DEC	EXEC	MEM	WB			
i+1		IF	DEC	EXEC	MEM	WB		
i+2			IF	DEC	EXEC	MEM	WB	
i+3				IF	DEC	EXEC	MEM	WB
i+4					IF	DEC	EXEC	MEM	WB

- La vitesse du MIPS est calculée en CPI (Cycle Par Instruction)
- En pratique, le CPI n'est pas égale à 1 parce qu'il y a des dépendances entre les instructions. Si l'instruction i a besoin du résultat de l'exécution de l'instruction i-1, il faut attendre en introduisant une bulle dans le pipeline, c'est-à-dire que pendant un cycle, on ne lit pas d'instruction.

Accès à la l'espace d'adressage

- Le mode d'adressage définit la manière de calculer l'adresse utilisé pour accéder à l'espace d'adressage
- Le MIPS32 ne dispose que d'un seul mode d'adressage
  l'adresse est calculée en additionnant le contenu d'un registre avec une valeur immédiate (prise dans l'instruction)
- Ce mode est nommé : *register indirect with displacement*

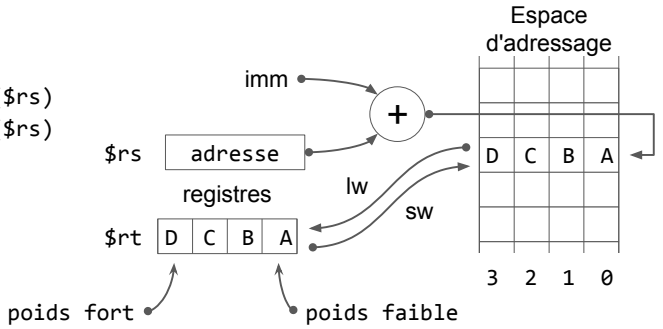
```
lw  $rt, imm ($rs)        # $rt ← MEM ( $rs + imm )
sw  $rt, imm ($rs)        # MEM ( $rs + imm ) ← $rt
```

`imm` est un nombre signé en complément à 2 sur 16 bits
[0xFFFF , 0x7FFF]
[-2^{15} , 2^{15} - 1]
[-32768, +32767]

Accès à l'espace d'adressage : lw et sw

- L'adresse (`$rs + imm`) doit être alignée sur un mot
  sinon c'est une erreur et l'exécution des instructions provoque une exception
- Le placement des octets en mémoire est de type **little endian**
  (poids faible du mot à l'adresse la plus petite)

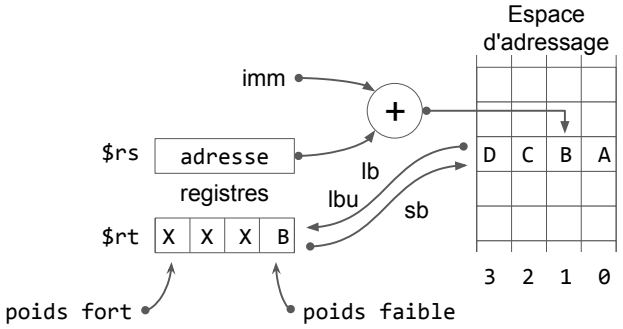
```
lw  $rt, imm ($rs)
sw  $rt, imm ($rs)
```



Accès à l'espace d'adressage : lb, sb et lbu

- On peut lire n'importe quel octet en mémoire.
- Il est rangé dans l'octet de poids faible du registre de destination.
- Les 3 octets de poids forts sont remplis par une valeur X :
  - X est le bit de signe de l'octet lu pour l'instruction `lb`
  - X est le bit `0` pour l'instruction `lbu`

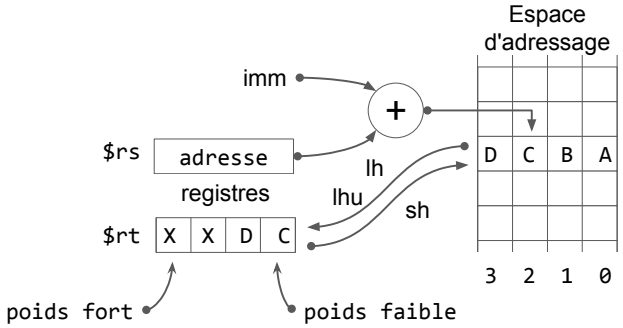
```
lb  $rt, imm ($rs)
sb  $rt, imm ($rs)
lbu $rt, imm ($rs)
```



Accès à l'espace d'adressage : lh, sh et lhu

- On peut lire un demi-mot (2 octets) aligné en mémoire.
- Il est rangé dans les deux octets de poids faible du registre de destination.
- Les 2 octets de poids forts sont remplis par une valeur X :
  - X est le bit de signe de l'octet lu pour `lh`
  - X est le bit `0` pour `lhu`

```
lh  $rt, imm ($rs)
sh  $rt, imm ($rs)
lhu $rt, imm ($rs)
```



Codage des nombres signés en complément à 2

- Les nombres négatifs sont codés en complément à 2 (en fait à 2^n)
- Si un nombre est codés sur n bits,
  le bit de poids fort est à 0 et les n-1 bits de poids faible code la valeur
- Le nombre positif le plus grand est donc $2^{n-1}-1$
  → sur 4 bits : de 0000 à 0111 en binaire donc de 0 à 7 en décimal
- Le codage en complément à 2^n signifie que l'opposé d'un nombre est son complément à 2^n, autrement dit : un nombre + son opposé = 2^n
  → $A + -A = 2^n = 0$ (codé sur n bits)
- En effet, 2^n est un nombre codé sur n+1 bits, en binaire c'est 1 suivi de n fois 0
- Quand un nombre a son bit de poids fort à 0, il est positif et sa valeur se lit directement dans les bits de poids faible
- Quand un nombre a son bit de poids fort à 1, il est négatif et sa valeur est le complément à 2^n
- Pour calculer le complément à 2^n de A = not A + 1 en effet : A + not A + 1 = 2^n

Différence entre add et addu

- Le processeur MIPS teste si le résultat est représentable sur n bits
- En effet, le MIPS dispose de registre de 32 bits si le résultat d'une instruction est trop grand, il est faux.
  - **add** demande au MIPS de réaliser le test de dépassement de capacité et d'appeler le système d'exploitation s'il y a dépassement
  - **addu** ne fait pas ce test
- Dans les deux cas, le résultat du calcul est identique

Directives

Une directive est une commande du programme d'assemblage.
La syntaxe est : *.directive*

Quelques exemples :

.text	demande que la suite soit rangée dans le segment de .text
.data	demande que la suite soit rangée dans le segment de .data
.align *n*	demande que le "pointeur de remplissage" soit aligné sur 2^n (donc les n bits de poid faible de l'adresse seront à 0)
.word *X,Y,...*	alloue autant de mot que d'argument et les initialise à *X, Y*, etc.
.asciiz *"mess"*	alloue autant d'octets que nécessaire pour "mess" + un pour 0 on peut mettre plusieurs messages séparés par une virgule
.space n	demande que le "pointeur de remplissage" soit déplacé de n octets
.section ….	permet de créer des noms de section (on verra plus tard…)

Symbole (label)

Un symbole ou un label ou encore une étiquette est une nom donné à une adresse ou à une valeur dans le programme assembleur

La syntaxe est : `label:`

```
     .data
v1:  .word 42           # v1 est l'adresse du 1er octet du mot contenant 42
v2:  .asciiz "salut"    # v2 est l'adresse du 1er octet de la chaine "salut"
     .align 2
v3:  .word 12           # v3 est l'adresse du 1er octet du mot contenant 12

      .text
lab1: addu $4, $4, -1   # lab1 est l'adresse de l'instruction addu
      bne  $4, $0, lab1 # permet de désigner l'adresse du saut
      la   $4, v1       # demande de charge $4 avec l'adresse v1 (lui+ori)
      lw   $5, ($4)     # lit le mot mémoire présent à l'adresse v1
```


Programme assembleur

```
        .data             # directive demandant que ce qui suit
                          # soit mis dans le segment data

var1:   .word  -12        # allocation d'un mot et initialisation
var2:   .word  42         # allocation du mot juste après

        .text             # directive demandant que ce qui suit
                          # soit mis dans le segment text

ori     $4, $0, -12       # $4 ← -12
ori     $6, $0, 42        # $6 ← 42
addu    $7, $4, $6        # $7 ← $4 + $6
```

macro-instructions : `li` et `la`

une macro-instruction est une pseudo-instruction que l'on peut ajouter au langage assembleur et qui est définie par une séquence d'instructions élémentaires.
Les macro-instructions permettent d'étendre le langage assembleur.

- Le chargement d'un registre avec une valeur prend deux instructions

```
lui     $4, 0x7654          # $4 ← 0x76540000
ori     $4, $4, 0x3210      # $4 ← 0x76543210
```

- `Il existe une macro instruction pour ça`

```
li      $4, 0x76543210      # $4 ← 0x76543210
```

- Les adresses associées aux labels sont connues seulement de l'assembleur.
- Pour les récupérer, il faut utiliser la macro `la : load address.`

```
    .data
v1: word    42
    .text
    la      $4, v1              # $4 ← l'adresse de v1
    lw      $5, ($4)            # $5 ← MEM [ v1 ]
```


Boucles

Un programme comporte toujours des ruptures de séquence permettant de décrire des boucles.

```
int i;
int r;
i = 4;
r = 0;
while (i != 0) {
    r = r + i;
    i = i - 1;
}
```

Assignation des registres
$8 pour i
$9 pour r

```
    .data
i:  .word   0               # int i;
r:  .word   0               # int r;

    .text
    li  $8, 4               # i = 4;
    li  $9, 0               # r = 0;

    j cond_while
while:
    addu    $9, $9, $8      # r = r + i;
    addiu   $8, $8, -1      # i = i - 1;
cond_while:
    bne     $8, $0, while # while (i != 0)
```


Appel système

- Les appels système sont des demande de service au système d'exploitation. Nous les verrons plus en détails plus tard mais vous devez déjà savoir ce que c'est et comment les utiliser dès maintenant.
- Le système d'exploitation propose, ici, 7 services :
  - service n° 1 : écriture d'un nombre entier sur la console
  - service n° 5 : lecture d'un nombre entier depuis la console
  - service n° 11 : écriture d'un caractère sur la console
  - service n° 12 : lecture d'un caractère depuis la console
  - service n° 4 : écriture d'une chaine de caractères sur la console
  - service n° 8 : lecture d'une chaine de caractères depuis la console
  - service n° 10 : terminaison d'un programme
- Pour utiliser un service :
  - placer les argument du service dans les registres $4 à $7
  - placer le numéro du service dans le registre $2
  - exécuter l'instruction `syscall`

par exemple

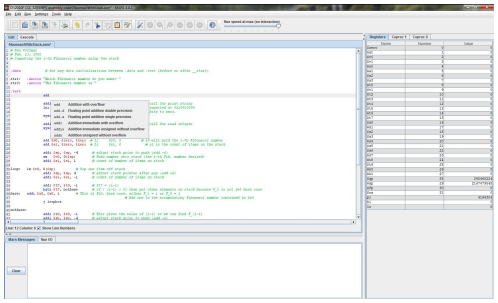
```
li  $4, 42
li  $2, 1
syscall
```


Simulateur MARS

Pour exécuter les programmes assembleur nous allons utiliser MARS `(MIPS Assembler and Runtime Simulator)` de l'université du Missouri

C'est un IDE *(integrated development environment)* comprenant

- un éditeur
- un programme d'assemblage
- un simulateur pour exécuter
  - en pas à pas
  - jusqu'à un point d'arrêt
- permet de voir
  - le contenu des registres
  - l'état de la mémoire
- gère les appels système basiques pour les entrées-sorties avec la console



En résumé, nous avons vu :

- Le modèle d'un ordinateur et le concept d'espace d'adressage
- Les caractéristiques principales du MIPS32
- Le concept de "section" et les 3 sections nécessaires à l'exécution d'un programme.
- Les types d'instructions du MIPS32
- Le principe d'exécution pipelinée
- Les instructions load / store
- Le codage en complément à 2
- La structure d'un programme assembleur
- Les macro-instructions
- Le concept d'appel système
- Le simulateur MARS utilisé en TP

Prochain cours

Les fonctions

- la programmation structurée
- La pile d'exécution (stack)
- Les contextes d'exécution des fonctions
- La convention pour le passage des arguments
- Un exemple