Architecture des ordinateurs
(X31I050)

Frédéric Goualard

Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241
Bureau 112, bât. 11
Frederic.Goualard@univ-nantes.fr

Langage d'assemblage MIPS

De traduction en traduction

Langage C

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

Compilation

Langage d'assemblage MIPS

```
fmax:
        sub $sp, $sp, 24
        sw $ra, 16($sp)

        move $t0, $a0
if:     blt $a1, $a2, else
then:
        move $a0, $a1
        move $a1, $a2
        jalr $t0
        b endif
else:
        move $a0, $a2
        jalr $t0
endif:
        lw $ra, 16($sp)
        add $sp, $sp, 24
        jr $ra
```

Assemblage

Code machine

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

De traduction en traduction

Langage C

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

Compilation

Langage d'assemblage MIPS

```
fmax:
        sub $sp, $sp, 24
        sw $ra, 16($sp)

        move $t0, $a0
if:     blt $a1, $a2, else
then:
        move $a0, $a1
        move $a1, $a2
        jalr $t0
        b endif
else:
        move $a0, $a2
        jalr $t0
endif:
        lw $ra, 16($sp)
        add $sp, $sp, 24
        jr $ra
```

Assemblage

Code machine

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

Intel ix86 *vs.* MIPS

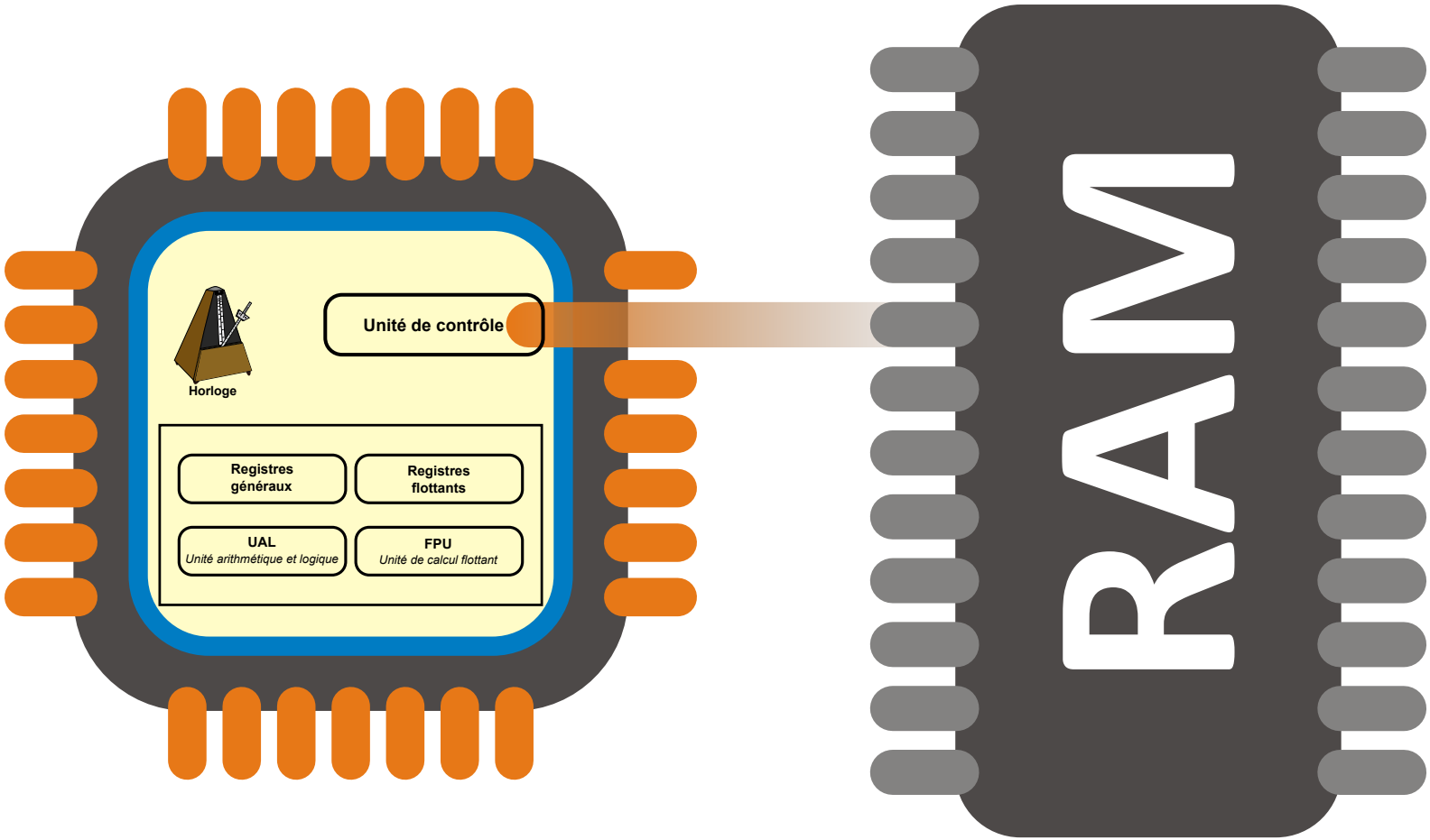
Intel x86

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

MIPS

- Architecture RISC pipelinée (beaucoup de registres, instructions simples)
- Langage d'assemblage simple
- Usage : ordinateurs de bureaux (SGI), lecteurs DVD , appareils photos (Nikon D40), imprimantes (HP Color LaserJet), consoles de jeux (Sony PlayStation PSX), ...
- PIC 32MX250 (Pinguino)
- Précurseur de RISC-V

Processeur / mémoire



Les instructions machine

- Chaque processeur possède son propre jeu d'instructions machine directement compréhensibles par lui
- Chaque instruction machine possède un identifiant numérique : l'*opcode*
- Programmation directe du processeur avec les instructions machines :
  - Difficile et long
  - Compréhension quasi-impossible

➠ Utilisation d'un langage de plus haut niveau associant un mnémonique à chaque instruction machine : le *langage d'assemblage*

Utilisation du langage d'assemblage

Avantages :

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

Inconvénients :

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

Exemple complet

hello.asm

```
        .data
msg:    .asciiz  'Hello, world!\n'

        .text
        .globl __start
__start:
        li $v0, 4
        la $a0, msg
        syscall

        li $v0, 10
        syscall
```

Notion de segment

Déclaration de variable + initialisation

Nom du prog. principal

Début de programme

$v0 = 4

$a0 = @msg

Appel du service print_string

$v0 = 10

Appel du service exit

Structure d'un programme en langage d'assemblage

Organisation du fichier source :

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

32 registres de 32 bits (+ registres flottants)

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

La plupart des usages correspondent à des *conventions*

Utilisation des registres

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

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

Format général d'une instruction ASM

`label: mnémonique` $\texttt{arg}_1$`,` $\texttt{arg}_2$`,` $\texttt{arg}_3$

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

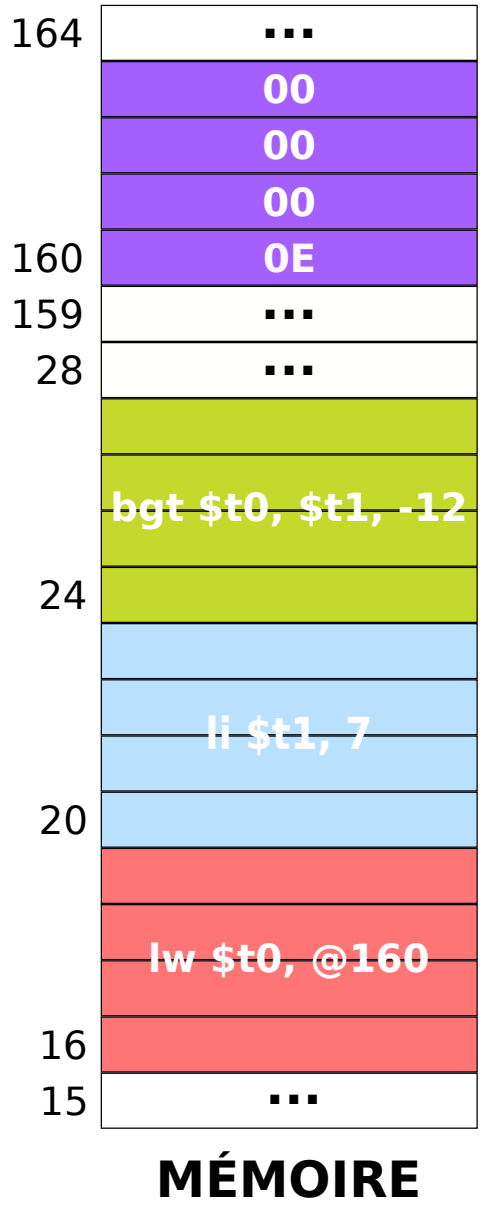
mnémonique. Nom réservé pour une *classe* d'opcodes ayant la même fonction

$\texttt{arg}_i$. Entre 0 et 3 opérandes

- Notation Intel : destination = opérande de gauche

Labels et mémoire

```
        .data
val:    .word 14

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



Types d'opérandes d'une instruction

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

Syntaxe générale

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

```
100  # décimale
0xa2 # hexadécimal
```

- Caractères et chaînes :

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

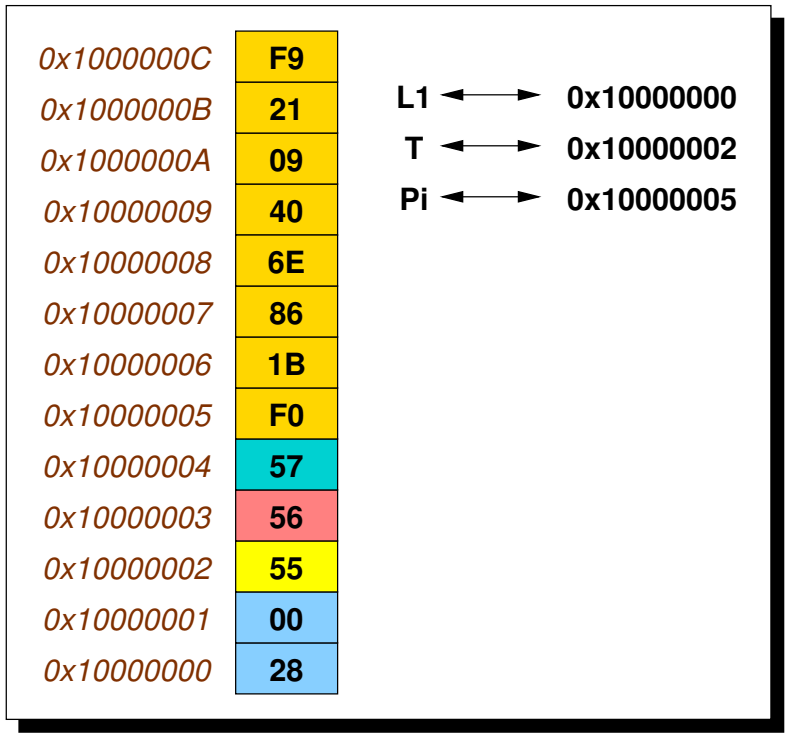
Directives

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

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

La directive donne la taille à réserver *par entrée*

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



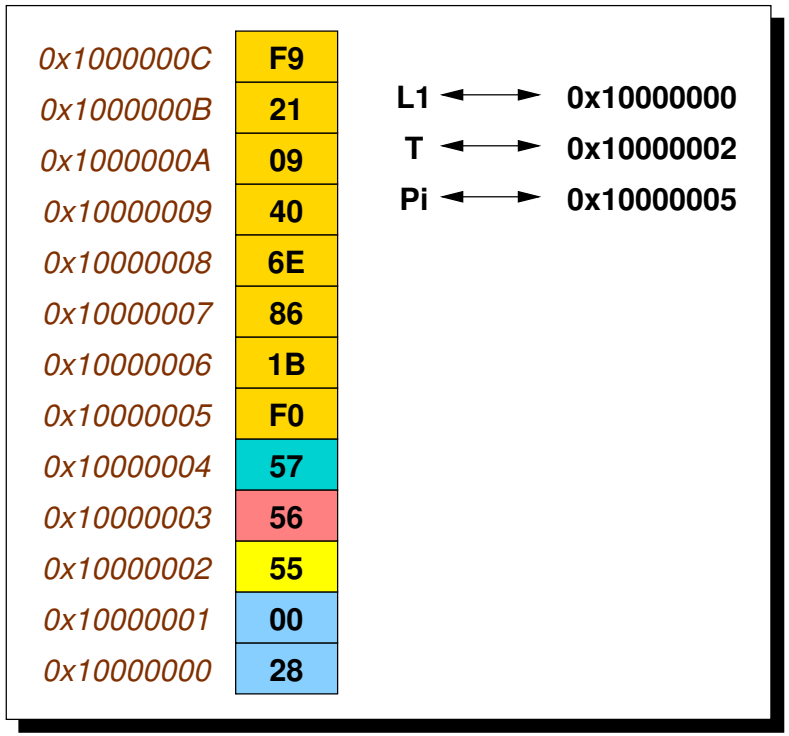
Directives

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

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

La directive donne la taille à réserver *par entrée*

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



Instructions de base

- Chargements (de/vers un registre)

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

- Branchements / Sauts

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

- Instructions logiques & arithmétiques

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

Transferts de données

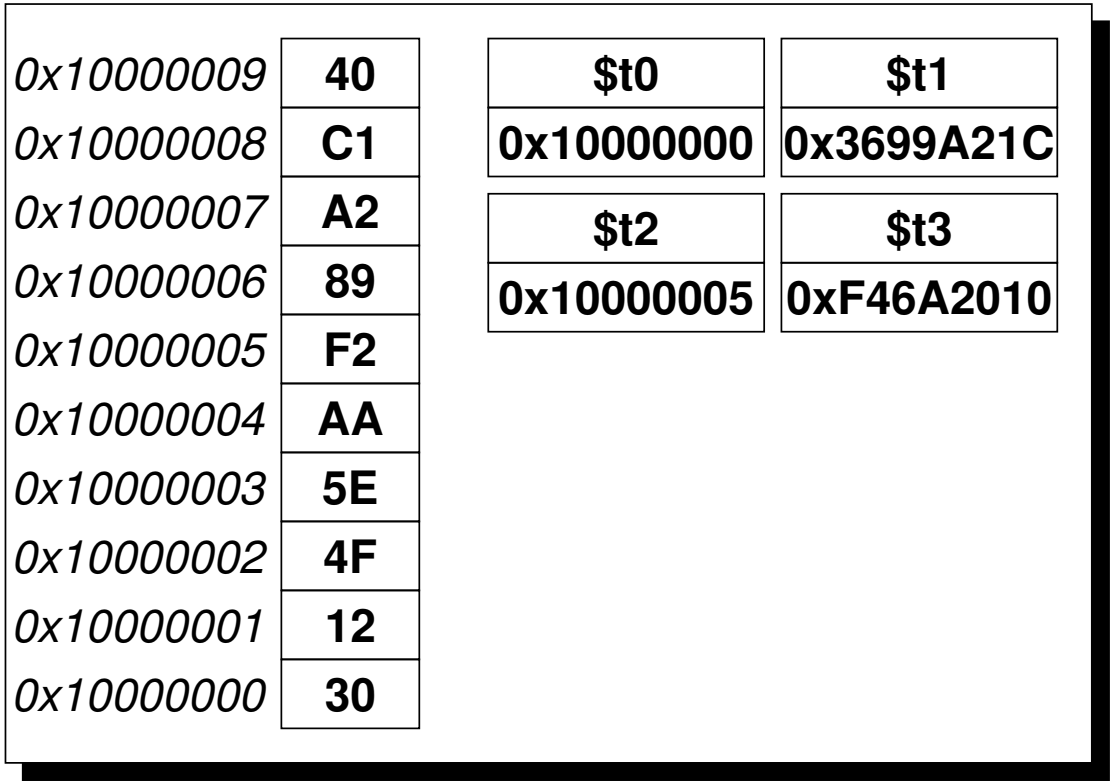
Instructions `move`, `s`X, `l`X

Exemples :

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

Transferts de données

Instructions `move`, `s`X, `l`X

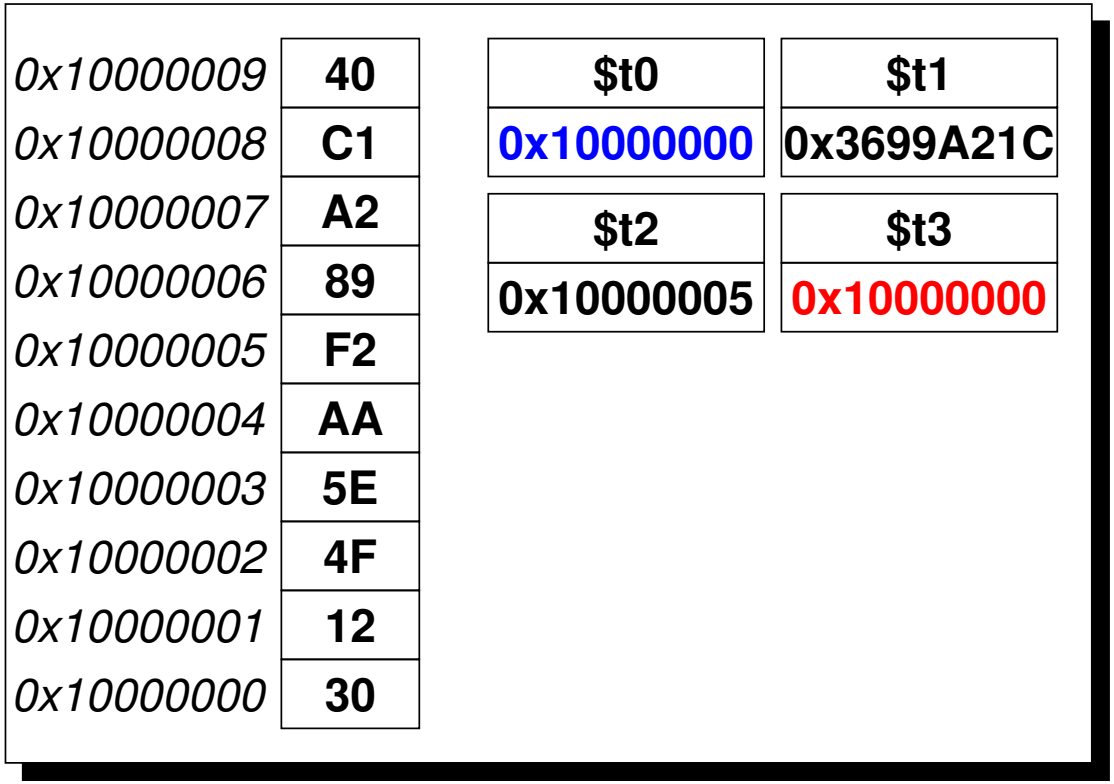


Exemples :

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

Transferts de données

Instructions `move`, `s`*X*, `l`*X*

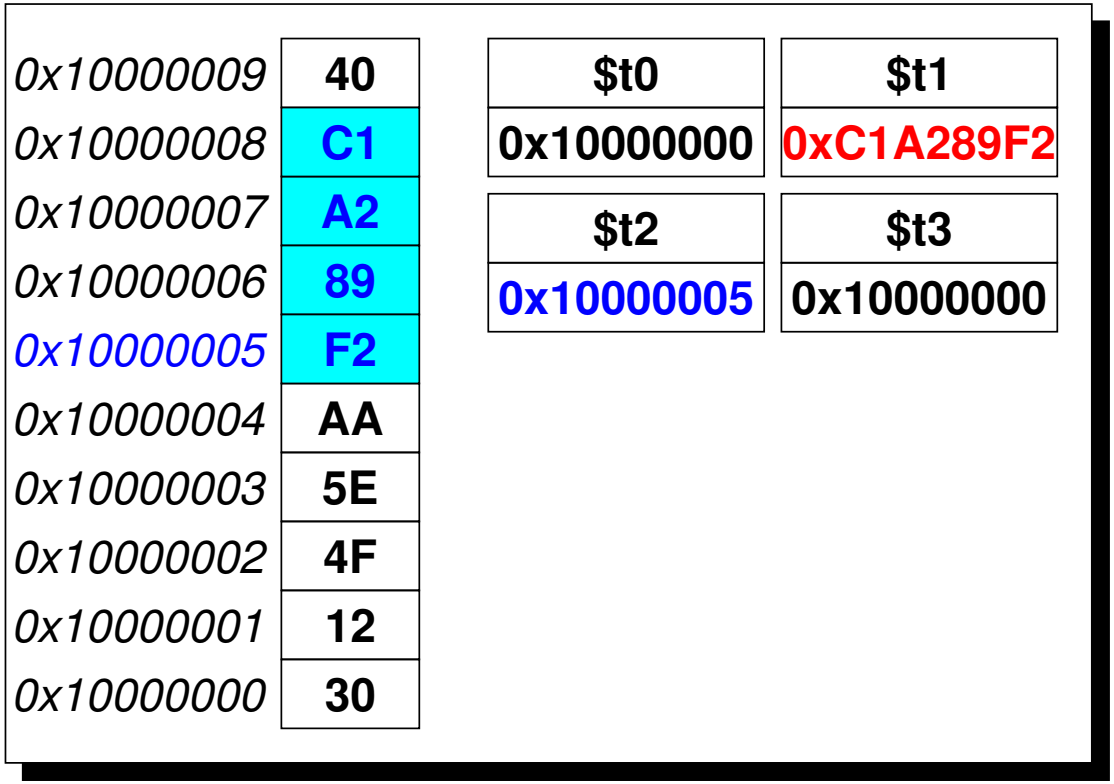


Exemples :

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

Transferts de données

Instructions `move`, `s`X, `l`X

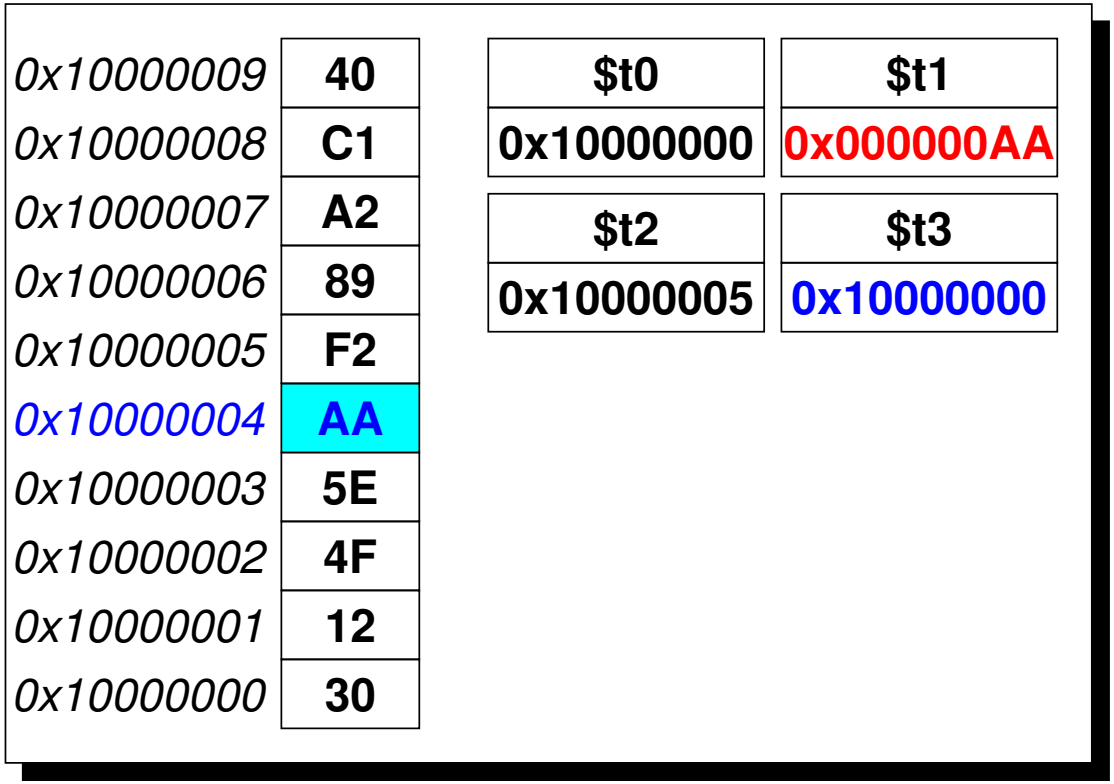


Exemples :

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

Transferts de données

Instructions `move`, `s`X, `l`X

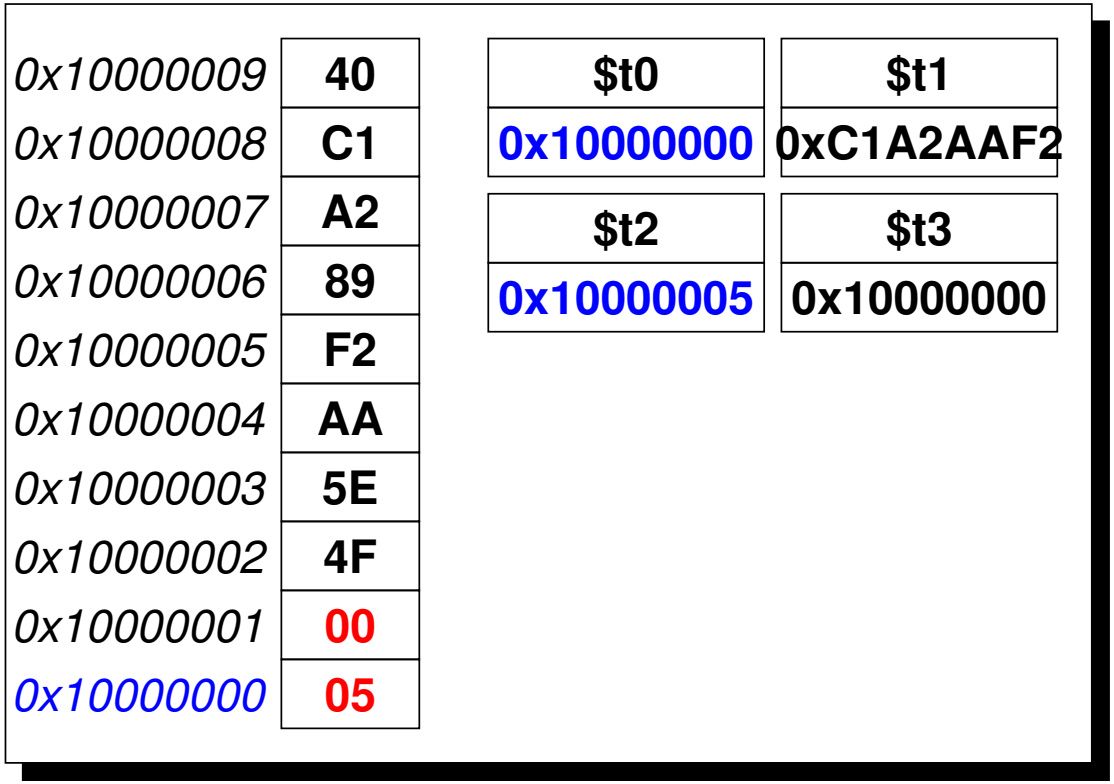


Exemples :

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

Transferts de données

Instructions `move`, `s`*X*, `l`*X*



Exemples :

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

Modes d'adressage

Transfert de ou vers une adresse mémoire :

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

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

Les étiquettes

Étiquettes ou labels :

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

Exemple :

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

Opérations arithmétiques

Opérations sur les entiers :

- `add`, `sub`, `mul`, `div`, ...

Exemple

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

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

Contrôle de flot (1/3)

Sauts relatifs (courts)

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

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

Contrôle de flot (2/3)

```
    .text
    li $t0, 3
    li $t1, -2 # ou 4294967294 (sur 32 bits)
if: ble $t0, $t1, endif
then:
    add $t2, $t2, 3
    b endif
endif:
```

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

Contrôle de flot (2/3)

```
    .text
    li $t0, 3
    li $t1, -2 # ou 4294967294 (sur 32 bits)
if: ble $t0, $t1, endif
then:
    add $t2, $t2, 3
    b endif
endif:
```

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

Tests conditionnels différents suivant le type des opérandes :

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

Contrôle de flot (3/3)

Sauts absolus (longs) :

- Saut inconditionnel : `j finprog`
- Saut sur registre : `jr $ra`
- Saut avec lien : `jal prog`, `jalr $t0`
  - Saut à l'étiquette `prog` après avoir sauvé dans `$ra` l'adresse de retour (adresse de l'instruction suivant le `jal`/`jalr`)

Instructions logiques

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

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

	0	1
NOT	1	0

- Applications :
  - Mise à 0 d'un registre :

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

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

```
  10100101        10100101        10101101
and 00001000    or  00001000    xor 00001000    not 10100101
  00000000        10101101        10100101        01011010
```

Traduction du if then else

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

```
        # test inversé
if:     bne $t0, 5, else
then:
        li $t1, 6
        add $t2, $t2, 1
        b endif
else:
        li  $t1, 9
endif:
```

Traduction du `while do`

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

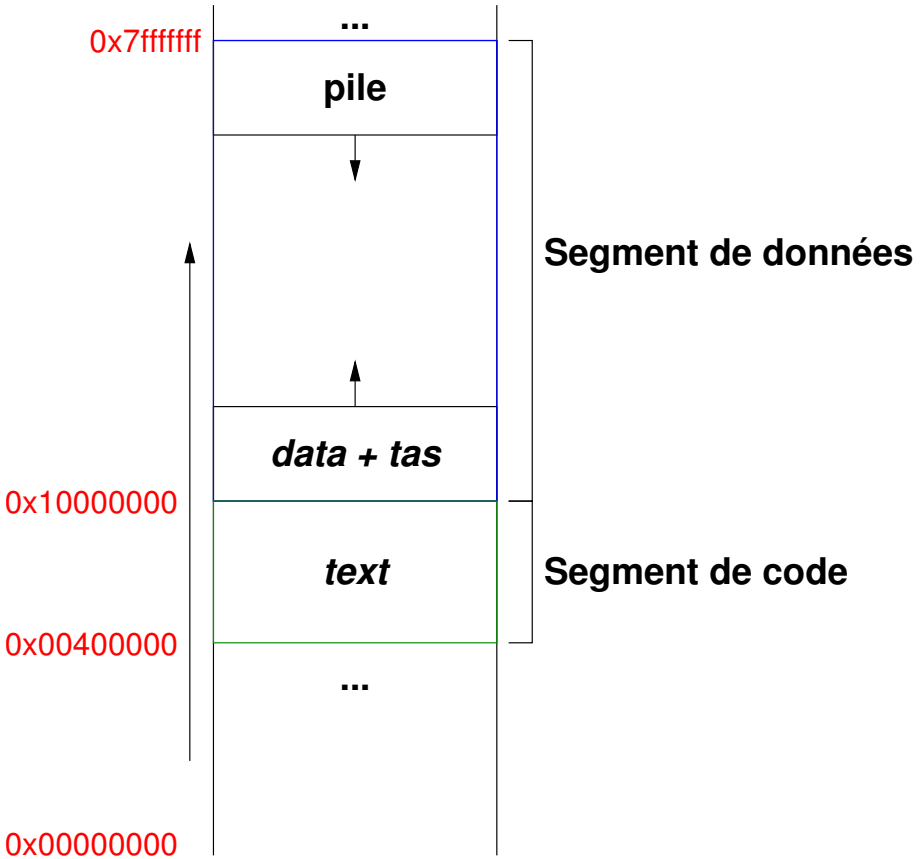
```
            # test inversé
while: ble $t0, 2, endwhile
do:
        sll $t1, $t1, 1
        sub $t0, $t0, 1
        b while
endwhile:
```

Architecture mémoire d'un programme

Utilisation de la mémoire par un programme s'exécutant :

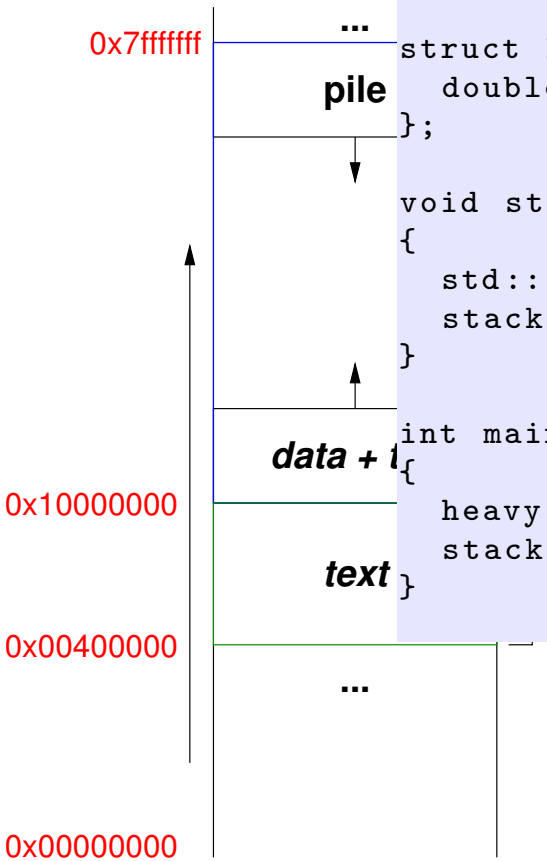
- `.text` : portion de la mémoire occupée par le code du programme
- `.data` : variables statiques + mémoire allouée dynamiquement (*tas*)
- `pile` (*stack*) : variables locales aux procédures/fonctions, paramètres

Positionnement en mémoire



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

Positionnement en mémoire




```
#include <iostream>

struct heavy {
  double a[200000];
};

void stack_overflow(heavy h, unsigned int i)
{
  std::cout << "Appel " << i << "\n";
  stack_overflow(h,i+1);
}

int main(void)
{
  heavy obj;
  stack_overflow(obj,1);
}
```

tas grandissent l'un

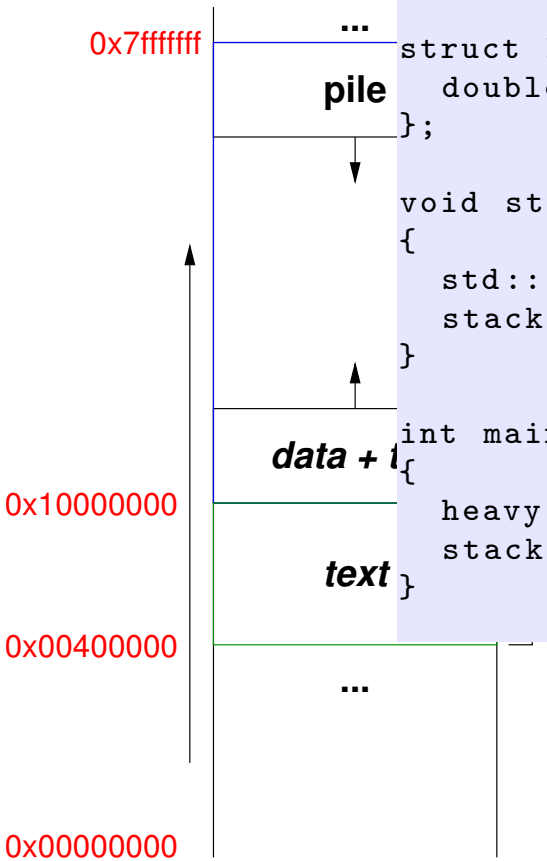
vers les adresses

e/tas :

ow ou

ow

Positionnement en mémoire




```
#include <iostream>

struct heavy {
  double a[200000];
};

void stack_overflow(heavy h, unsigned int i)
{
  std::cout << "Appel " << i << "\n";
  stack_overflow(h,i+1);
}

int main(void)
{
  heavy obj;
  stack_overflow(obj,1);
}
```

```
> ./stack_overflow
Appel 1
Appel 2
Appel 3
Appel 4
Segmentation fault
```

tas grandissent l'un

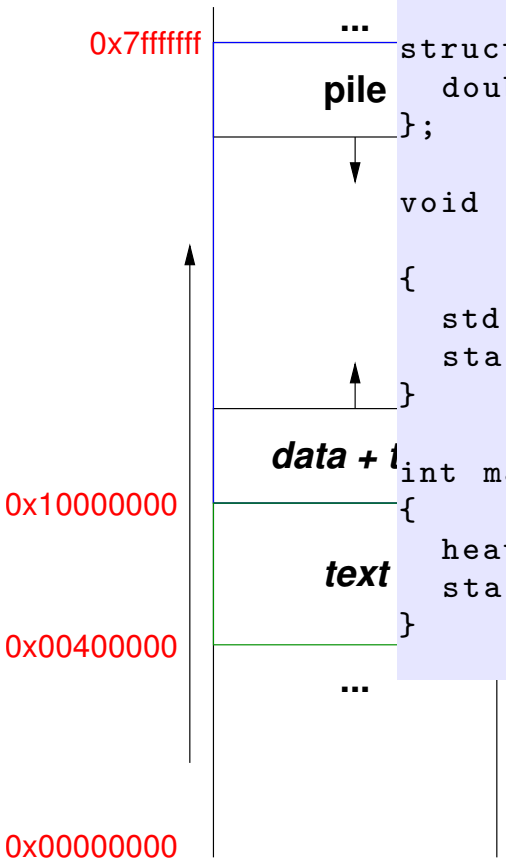
vers les adresses

e/tas :

ow ou

ow

Positionnement en mémoire




```
#include <iostream>

struct heavy {
  double a[200000];
};

void stack_overflow(const heavy &h,
                    unsigned int i)
{
  std::cout << "Appel " << i << "\n";
  stack_overflow(h,i+1);
}

int main(void)
{
  heavy obj;
  stack_overflow(obj,1);
}
```

e tas grandissent l'un

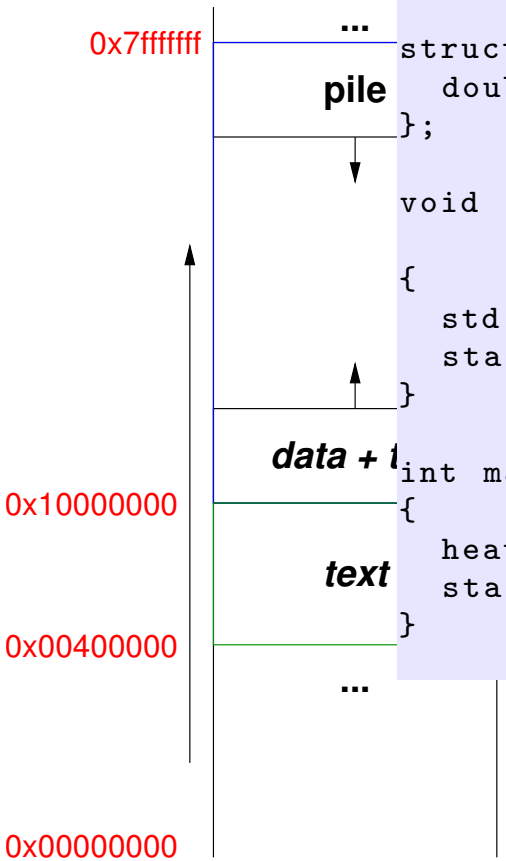
t vers les adresses

e/tas :

ow ou

ow

Positionnement en mémoire




```
#include <iostream>

struct heavy {
  double a[200000];
};

void stack_overflow(const heavy &h,
                    unsigned int i)
{
  std::cout << "Appel " << i << "\n";
  stack_overflow(h,i+1);
}

int main(void)
{
  heavy obj;
  stack_overflow(obj,1);
}
```

```
> ./stack_overflow
Appel 1
Appel 2
...
Appel 277194
Segmentation fault
```

e tas grandissent l'un

t vers les adresses

e/tas :

ow ou

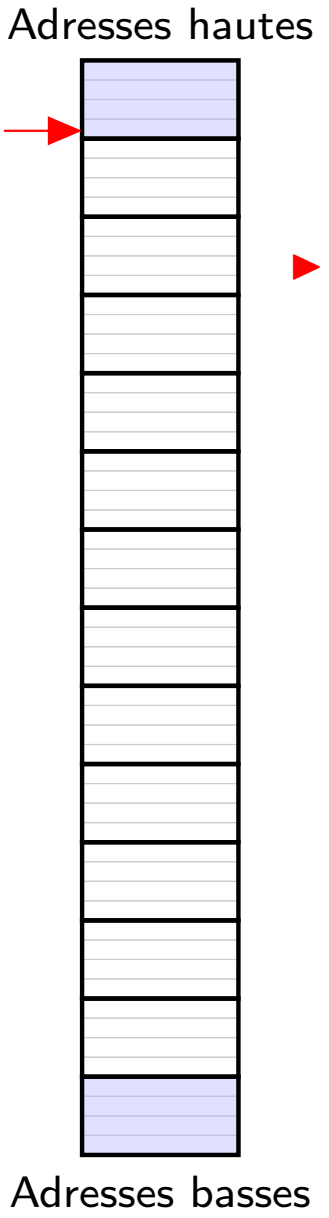
ow

Utilisations de la pile

- Stockage de résultats temporaires si manque de registres
- Sauvegarde de registres modifiés par des appels de fonctions
- Stockage des variables locales aux procédures/fonctions
- Passage de paramètres entre procédures/fonctions
- Convention MIPS : `$sp` pointe sur une adresse divisible par 8

Passage de paramètres : exemple

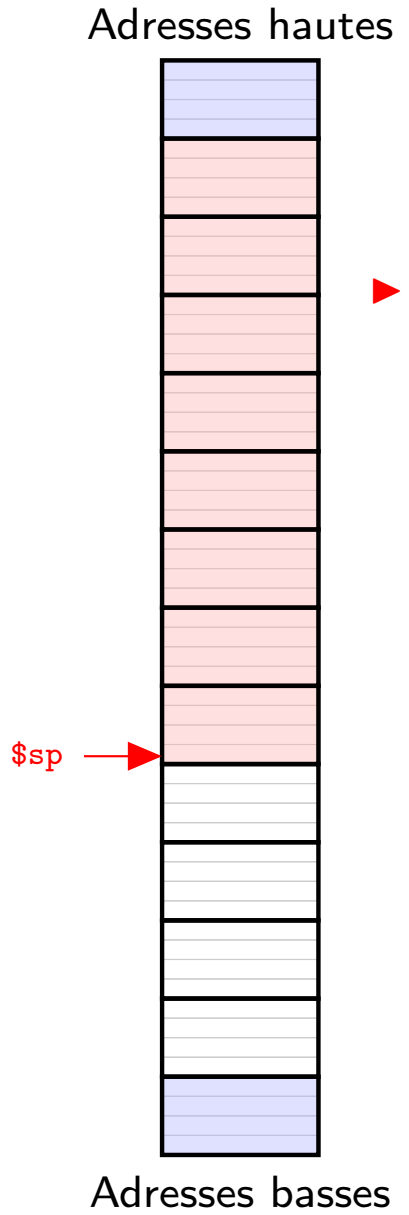
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

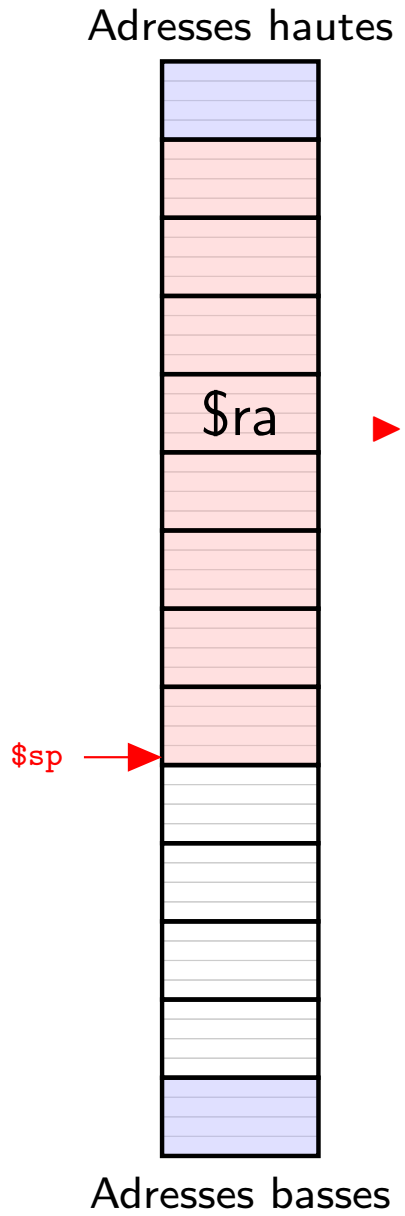
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

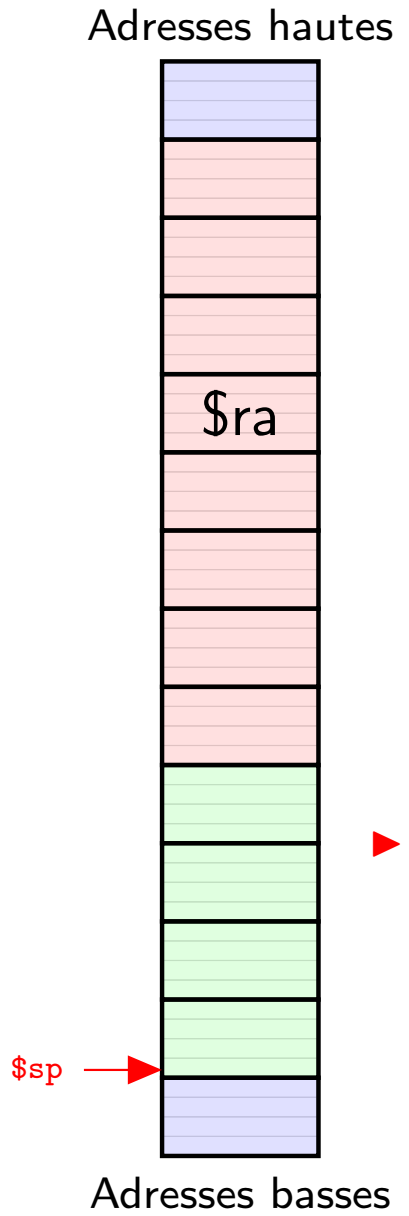
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

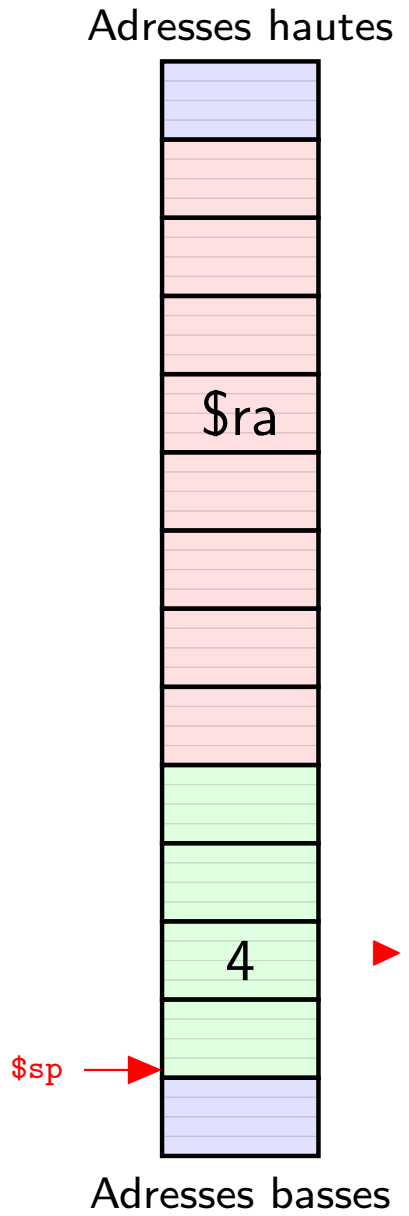
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

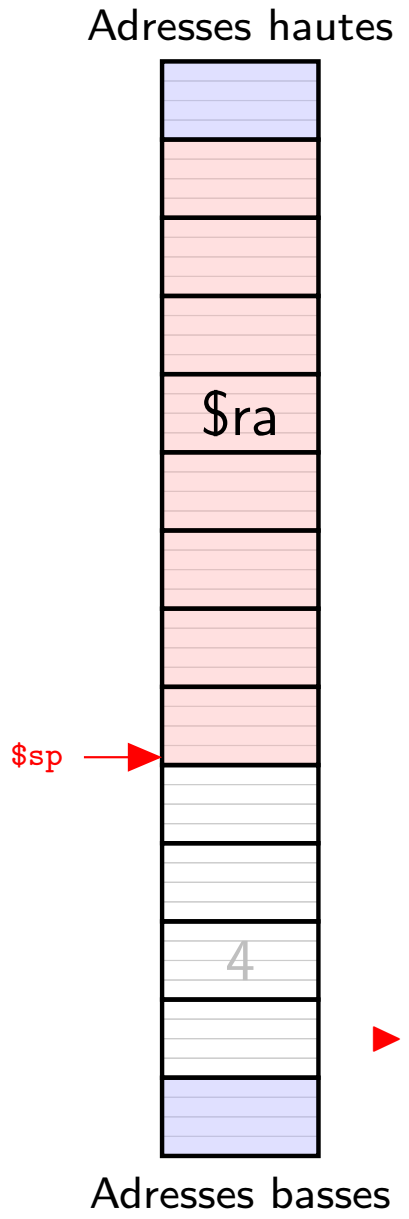
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

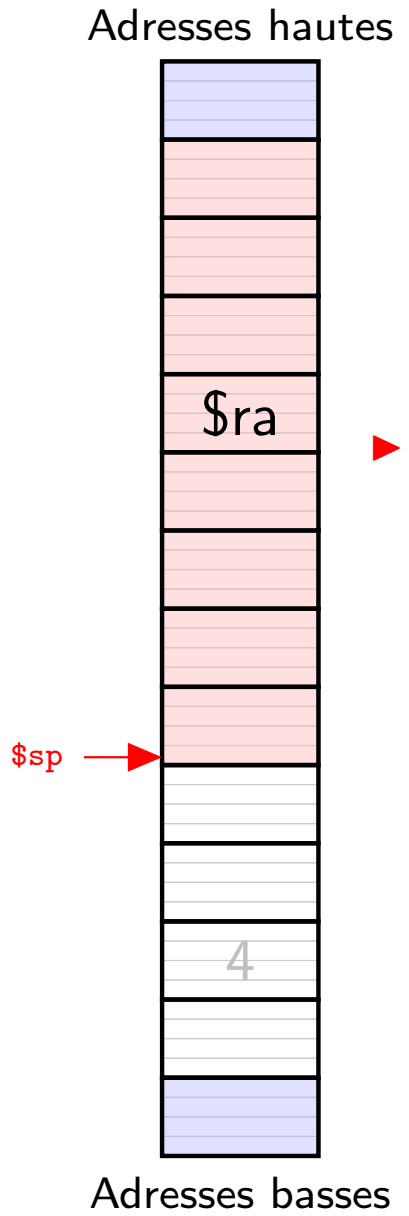
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

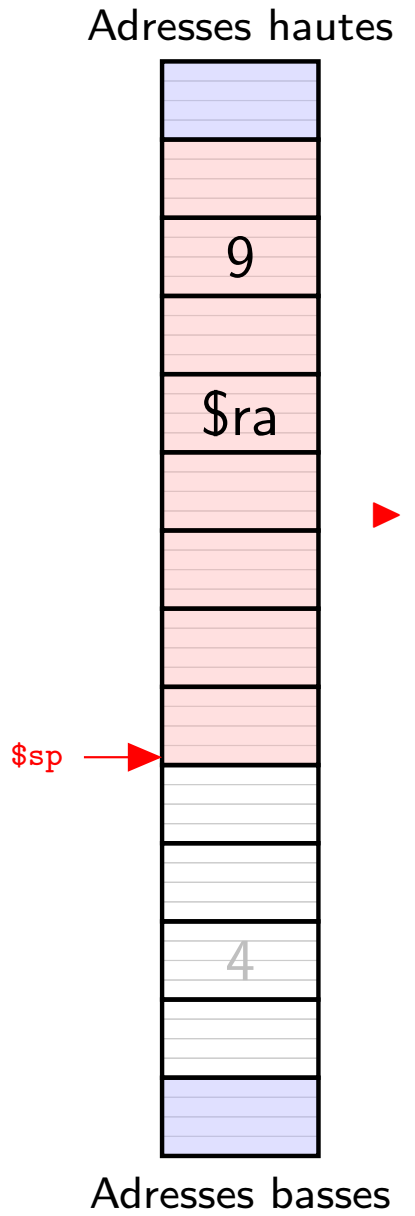
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Passage de paramètres : exemple

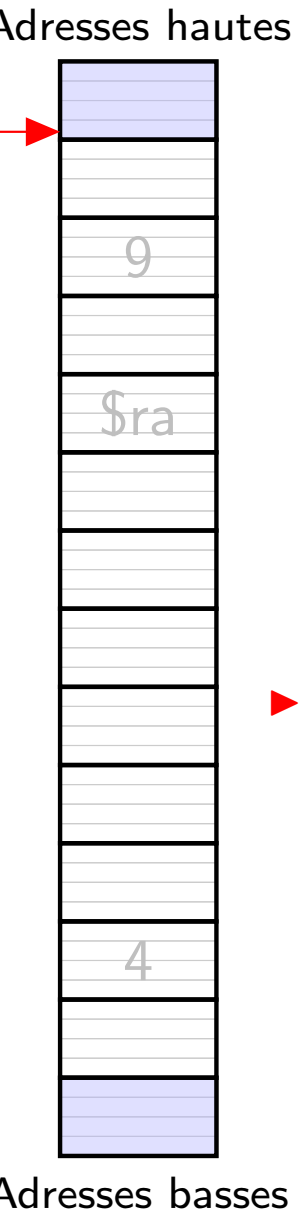
```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

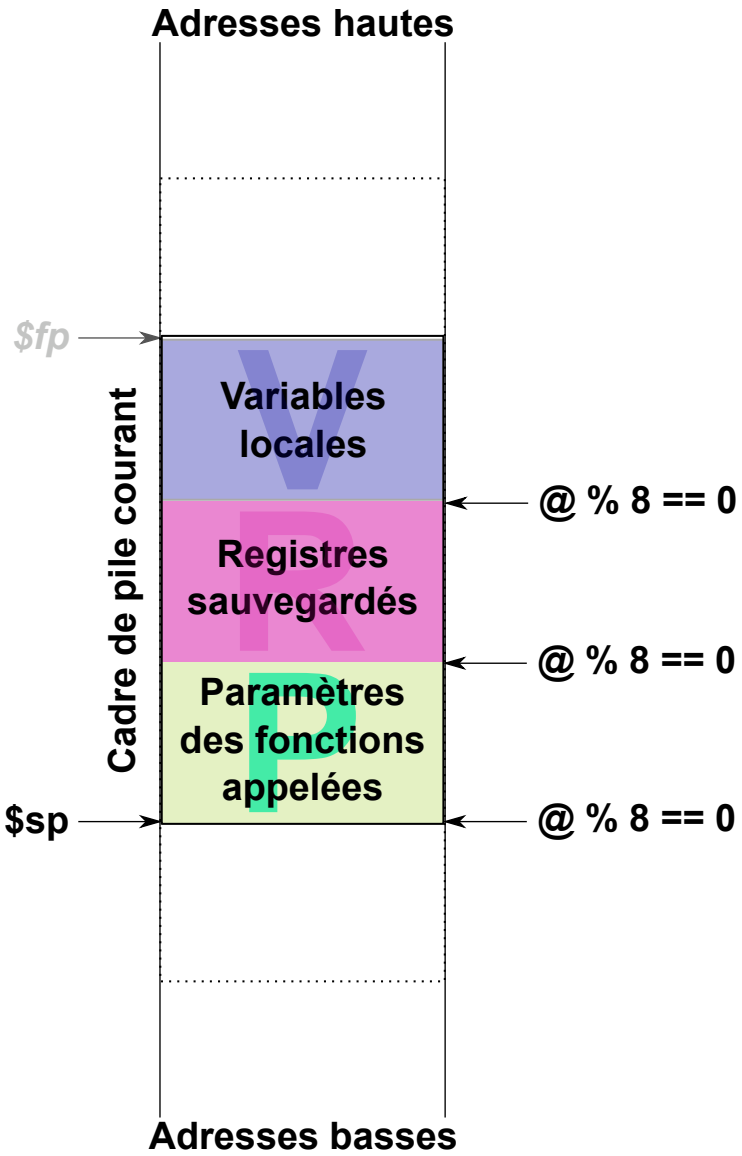
Passage de paramètres : exemple

```
unsigned int addition(unsigned int x,
                      unsigned int y) {
  unsigned int T[4];
  T[1] = x;
  return T[1]+y;
}

int main(void) {
  unsigned int z = addition(4,5);
  printf("%u\n",z);
}
```




```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall

main:
        subu $sp, $sp, 32
        sw $ra, 16($sp)

        li $a0, 4
        li $a1, 5
        jal addition
        sw $v0, 24($sp)

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

        lw $ra, 16($sp)
        addu $sp, $sp, 32
        jr $ra


addition:
        subu $sp, $sp, 16

        sw $a0, 4($sp)
        lw $t0, 4($sp)
        add $v0, $t0, $a1

        addu $sp, $sp, 16
        jr $ra
```

Cadre de pile général



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

Fonction terminale simple

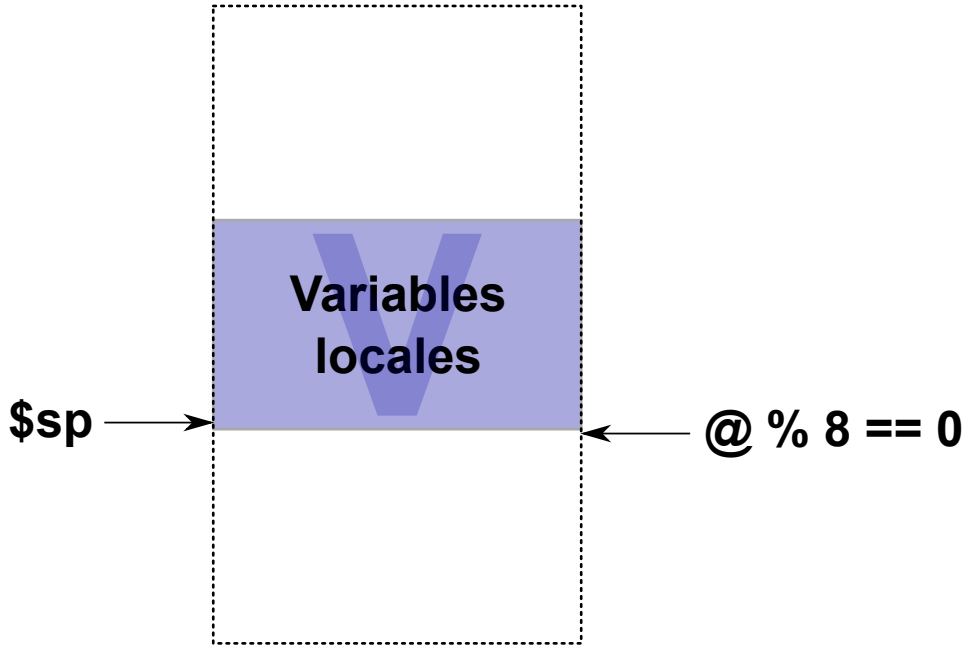
```
int f(int a, int b)
{
  return a+b;
}
```

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

$\Rightarrow$ pas de cadre de pile à créer

Fonction terminale avec variables locales

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

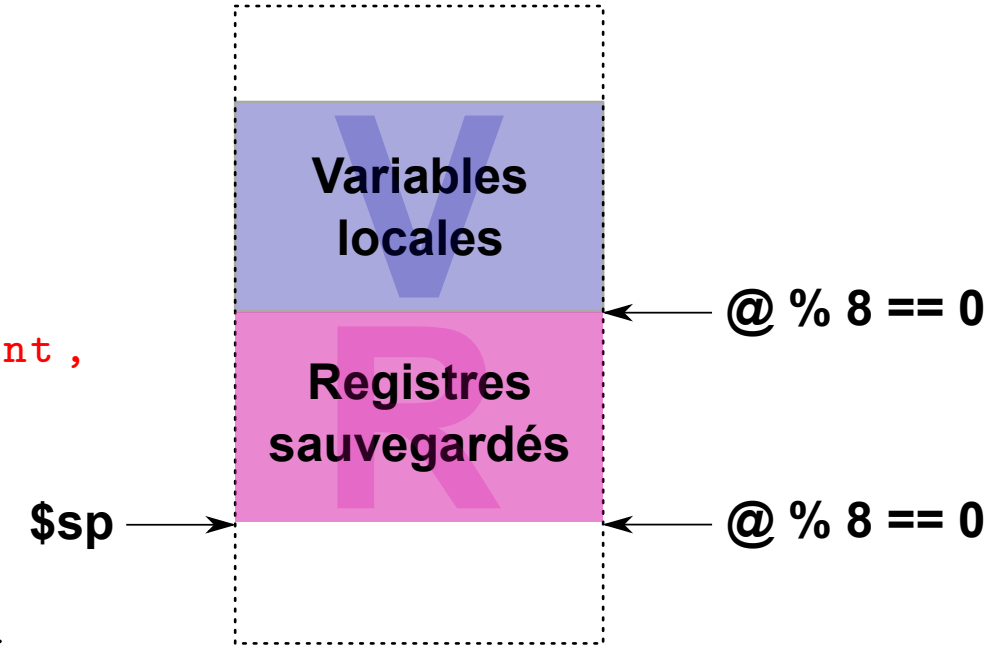


- Pas besoin de sauver `$ra`
- Présence de variables locales
- Pas de modification de registre à sauvegarder

$\Rightarrow$ Création d'un cadre de pile {**V**}

Fonction terminale évoluée

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

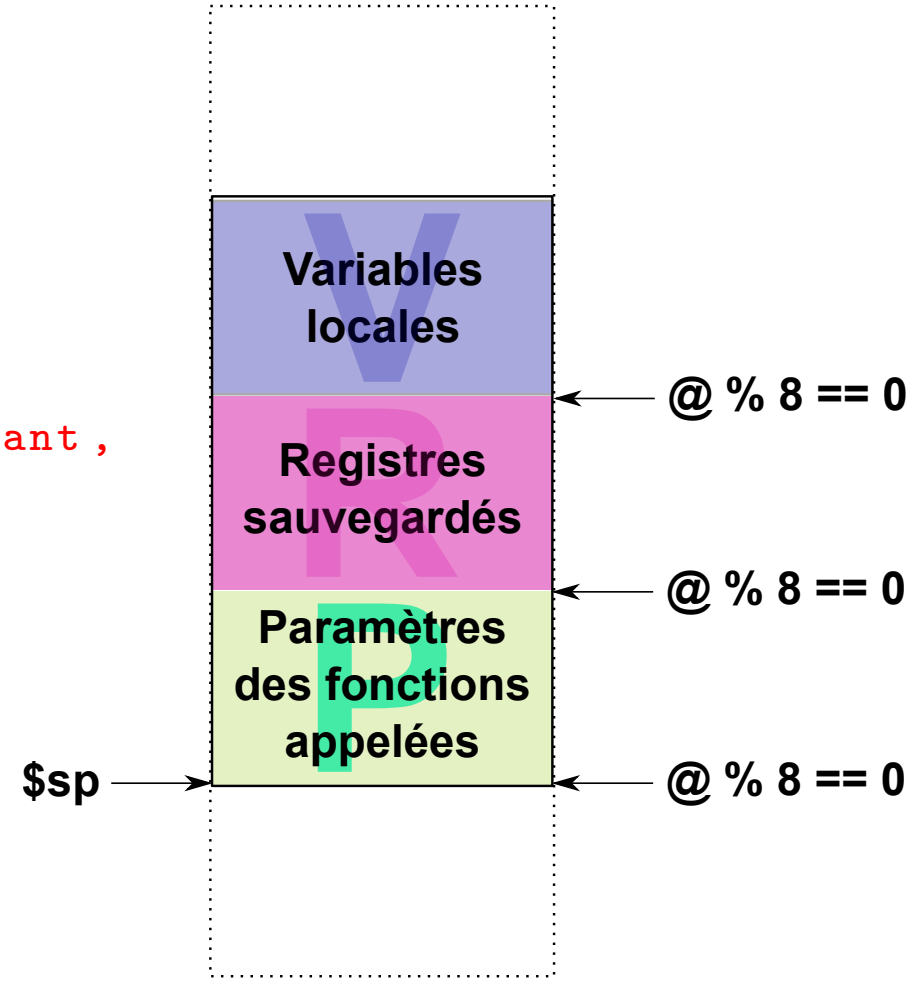


- Pas besoin de sauver `$ra`
- Présence de variables locales
- Registre(s) à sauvegarder

$\Rightarrow$ Création d'un cadre de pile {**R**, **V**}

Fonction non-terminale

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



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

$\Rightarrow$ Création d'un cadre de pile {**P**, **R**, **V**}

Conventions d'appel

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