LU2IN014 : Machine et Représentation

Cours 5 : Pile d'Exécution et Variables Locales

quentin.meunier@lip6.fr

1. Variables locales, contexte d'une fonction et pile d'exécution
2. Exemple de programmes avec variables locales
3. Appels de fonctions

Variables locales

Définitions

- Une variable locale est une variable dont la portée est limitée à une fonction (ou plus généralement à un bloc, mais ne nous intéresse pas trop ici)
- Une variable locale d'une fonction f n'est donc pas visible en dehors de cette fonction : ni dans la fonction appelante, ni dans les fonctions appelées par f
- L'ensemble des variables locales à une fonction constitue le **contexte** de cette fonction (définition approximative)

Variables locales et allocation mémoire

Remarque

- Une variable est un emplacement en mémoire (éventuellement plusieurs)
- $\Rightarrow$ Cela est vrai aussi pour les variables locales

Conséquences

- Il faut pouvoir allouer et désallouer de la mémoire pour les variables locales
  - Allouer de la mémoire au début début d'une fonction pour ses variables locales
  - Désallouer cette mémoire à la fin de la fonction, sinon : fuite mémoire
  - Une fonction qui se rappelle elle-même une fois contient 2 emplacements mémoire pour chacune de ses variables locales

Variables locales et allocation mémoire

Nature de l'allocation mémoire

- On ne connait pas forcément à l'avance le nombre de variables locales dont on va avoir besoin pour un programme (quelles fonctions, nombre d'appels)
- $\Rightarrow$ Besoin d'un mécanisme d'allocation dynamique de mémoire pour les appels de fonction

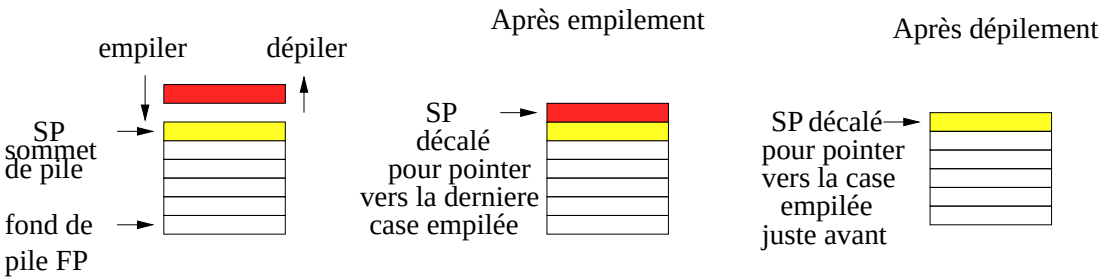
Propriété vis-à-vis des contextes de fonction

- On ne peut faire que deux opérations sur les contextes :
  - Empiler, lors d'un appel de fonction
  - Dépiler, lors de la sortie d'une fonction
- $\Rightarrow$ Notion de pile : la mécanique d'imbrication des fonctions suit celle de la pile
- L'implémentation de l'allocation des contextes de fonction est faite à l'aide d'une pile

Pile et variables locales

Notion de pile

- Structure de données abstraite permettant de stocker/récupérer des informations
- Gestion LIFO (Last In First Out) : la structure de pile conserve l'ordre d'allocation
- Opérations sur la pile :
  - Allouer de la mémoire en sommet de pile
  - Désallouer la mémoire en sommet de pile
  - Accéder (lecture ou écriture) à un élément en pile à partir du sommet de la pile
  - Tester si la pile est vide ou pleine (éventuellement, pas pour nous)
- Mécanisme purement sémantique : concrètement, il s'agit d'un segment en mémoire auquel on accède comme à un autre segment



La pile est vide lorsque SP est après FP, la pile est pleine lorsque SP = FP + taille_max

Pile et variables locales

Fonctionnement de la pile

- La pile est utilisée pour stocker (entre autres) les variables locales
- La pile est gérée dynamiquement :
  - On alloue de la place (des mots) sur la pile lorsque l'on en a besoin
  - On les désalloue lorsque l'on n'en a plus besoin
  - Pas le droit d'accéder à un mot non alloué !
- Allocation dynamique : réalisée avec des instructions du jeu d'instructions
- Comme on alloue/désalloue toujours au sommet, on a simplement besoin de spécifier le nombre d'octets à allouer/désallouer

Implantation de la pile en Mips

Organisation de la pile

- De manière générale, dépend de la machine cible
- En Mips, le registre $29 contient l'adresse du sommet de la pile, et le sommet de la pile est la dernière case occupée (et non la première libre)
- Ce registre est appelé le "pointeur de pile" (SP, Stack Pointer)
- En Mips, la pile grandit vers les adresses décroissantes
  - Allouer dans la pile revient à décrémenter $29
  - Désallouer dans la pile revient à incrémenter $29
- $29 doit toujours contenir l'adresse d'un mot du segment de pile
  - Au chargement il contient l'adresse du fond de pile (0x7FFFFFFC dans la convention Mips)

Contexte d'une fonction

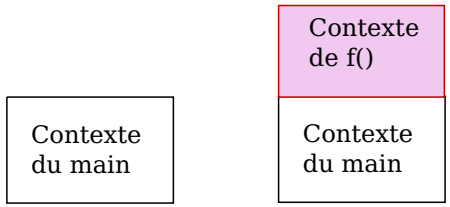
- On peut voir le contexte d'une fonction en pile comme l'ensemble des mots en pile accessibles depuis une fonction
- En réalité, la pile contient :
  - Les variables locales
  - La sauvegarde des registres persistants
  - La sauvegarde des paramètres de fonctions
  - Des mots utilisés lorsque l'on manque de registres pour faire un calcul temporaire (pas vu ici – souvent, c'est un cas pathologique ; dépend du nombre de registres généraux)

Contexte d'une fonction : illustration

Contexte
du main

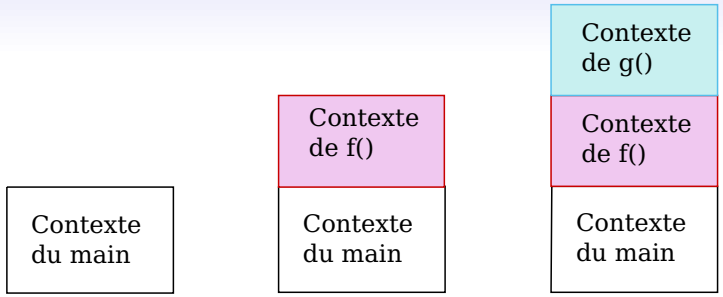
```
  void main() {
→   f();
  }

  void f() {
    g();
  }

  void g() {
    h();
    m();
  }

  void h() {
    ...
  }

  void m() {
    ...
  }
```

Contexte d'une fonction : illustration



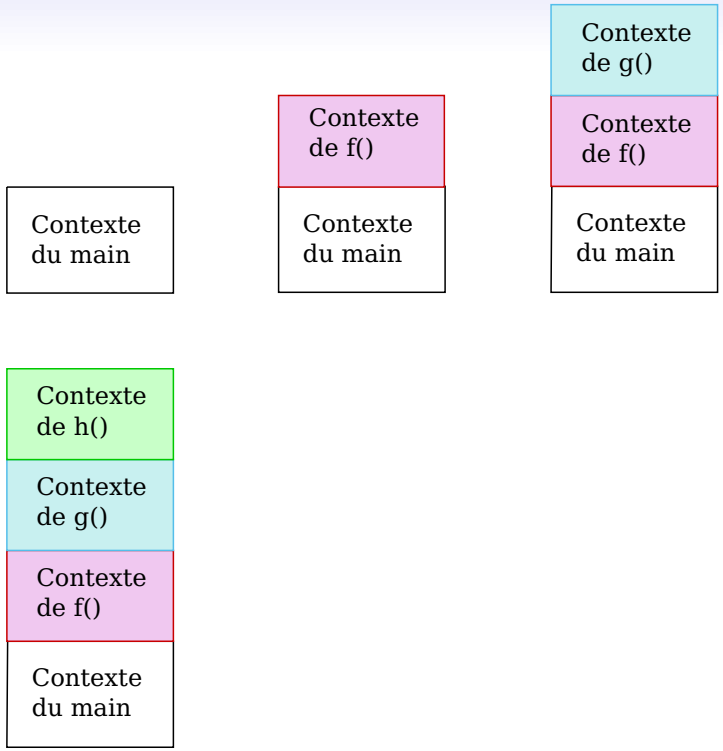

```
void main() {
   f();
}

void f() {
→  g();
}

void g() {
   h();
   m();
}

void h() {
   ...
}

void m() {
   ...
}
```

Contexte d'une fonction : illustration



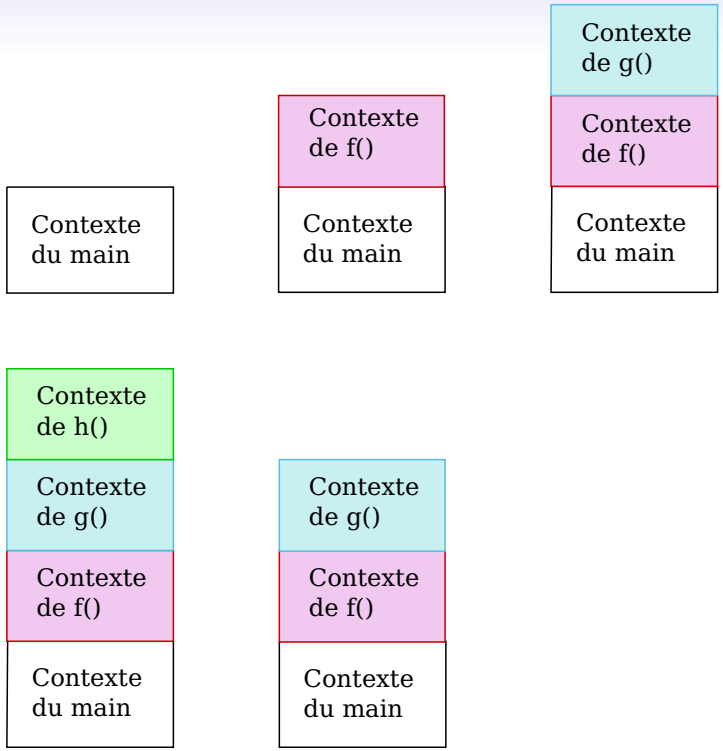

```
void main() {
   f();
}

void f() {
   g();
}

void g() {
→  h();
   m();
}

void h() {
   ...
}

void m() {
   ...
}
```

Contexte d'une fonction : illustration



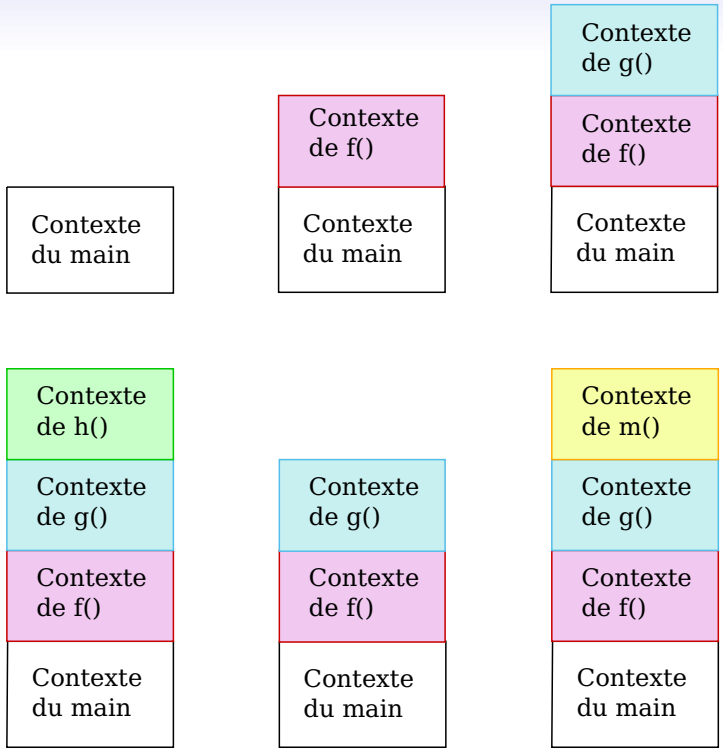

```
void main() {
   f();
}

void f() {
   g();
}

void g() {
   h();
   m();
}

void h() {
→  ...
}

void m() {
   ...
}
```

Contexte d'une fonction : illustration



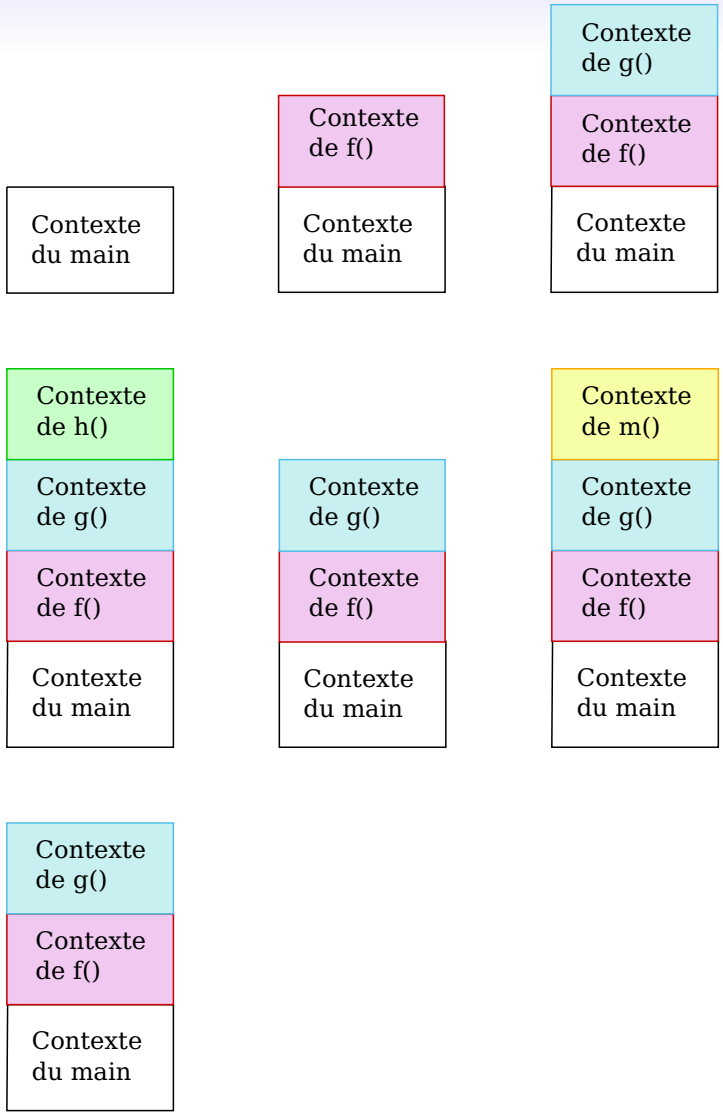

```
void main() {
   f();
}

void f() {
   g();
}

void g() {
   h();
→  m();
}

void h() {
   ...
}

void m() {
   ...
}
```

Contexte d'une fonction : illustration



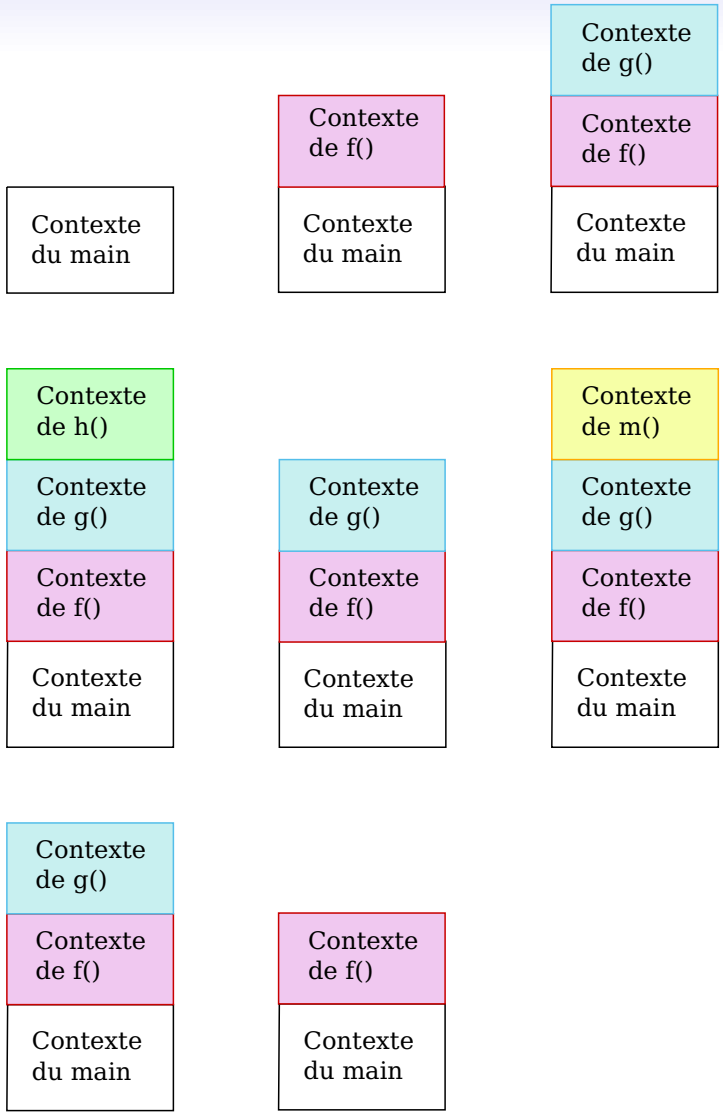

```
void main() {
   f();
}

void f() {
   g();
}

void g() {
   h();
   m();
}

void h() {
   ...
}

void m() {
→  ...
}
```

Contexte d'une fonction : illustration



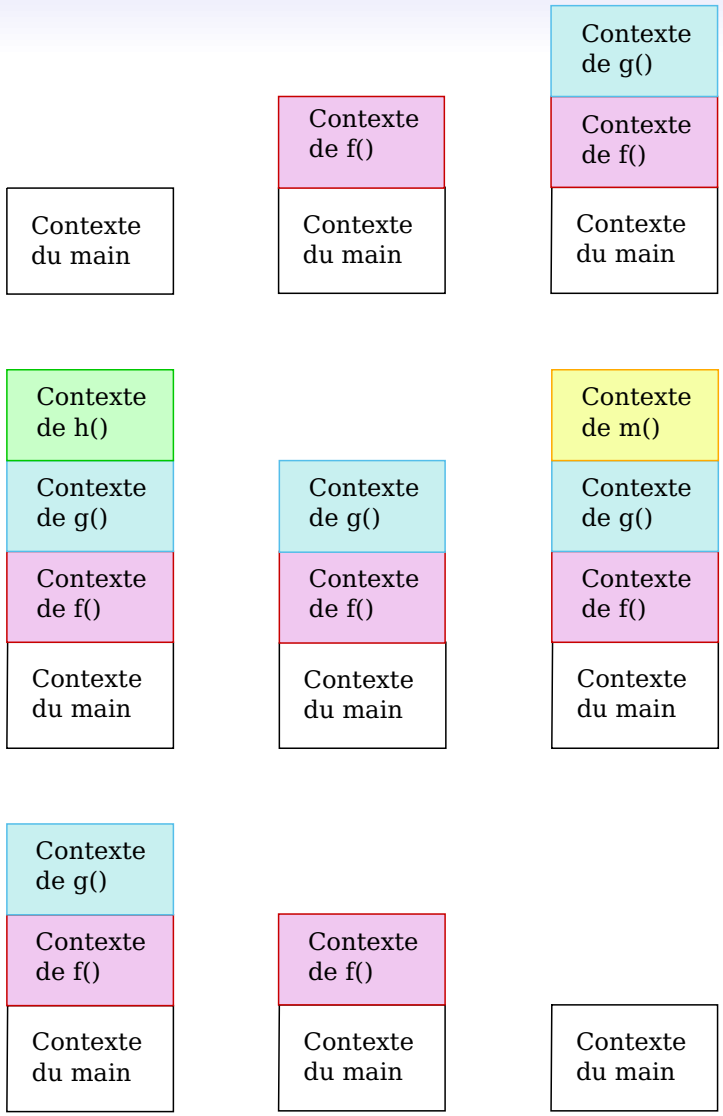

```
void main() {
   f();
}

void f() {
   g();
}

void g() {
   h();
   m();
}

void h() {
   ...
}

void m() {
   ...
}
```

Contexte d'une fonction : illustration




```
void main() {
   f();
}

void f() {
   g();
}

void g() {
   h();
   m();
}

void h() {
   ...
}

void m() {
   ...
}
```

Contexte d'une fonction : illustration




```
void main() {
   f();
}

void f() {
   g();
}

void g() {
   h();
   m();
}

void h() {
   ...
}

void m() {
   ...
}
```

Contexte d'une fonction

En pratique pour la fonction main()

- On alloue dans la pile au début du `main()`
- En particulier, pour les variables locales :
  - L'allocation se fait indépendamment des blocs
  - L'allocation se fait de la même manière que pour les variables globales (alignement, taille des variables)
  - La variable déclarée en dernier se trouve au sommet de la pile
- On désalloue la même quantité de mémoire avant l'appel système `exit` ou le `return`
- Remarque : on ne verra pas de variable locale de type tableau

Exemple avec variables locales

Code C

```
void main() {
    int a = 5;
    int b = 9;
    int c;
    c = a + b;
    exit();
 }
```

Code ASM

```
addiu $29, $29, -12 # alloue 3 mots
ori   $8, $0, 5     # $8 <- 5
sw    $8, 8($29)    # a <- 5
ori   $8, $0, 9     # $8 <- 9
sw    $8, 4($29)    # b <- 9
lw    $8, 8($29)    # $8 <- a (5)
lw    $9, 4($29)    # $9 <- b (9)
addu  $8, $8, $9    # $8 <- 14
sw    $8, 0($29)    # c <- 14
addiu $29, $29, 12  # désallocation
ori   $2, $0, 10
syscall
```

Remarque

- a est à l'adresse ($29 + 8), b à l'adresse ($29 + 4), c à l'adresse $29

Représentation des variables en assembleur

Traduction littérale

- Une **variable** dans un programme C correspond à une (ou plusieurs) **case mémoire**
  - Vrai pour les variables locales, globales, et aussi les paramètres de fonction
- Chaque lecture d'une variable correspond à une lecture en mémoire
- Chaque écriture d'une variable correspond à une écriture en mémoire
- Une variable n'est **pas** associée à un registre : il peut y avoir beaucoup de variables (des centaines) alors qu'il n'y aura toujours que 32 registres
  - Les registres ne sont utilisés que de façon très temporaire
  - Différents `load` de la même variable peuvent avoir des registres de destination différents
  - Un même registre peut contenir les valeurs de variables différentes au cours de l'exécution d'une fonction

Représentation des variables en assembleur

Optimisation

- On peut **optimiser** le code assembleur pour réduire le nombre d'accès mémoire
  - Optimisation manuelle ou *via* des options spécifiques du compilateur
- On peut parfois faire l'association d'une **variable locale** (i.e. dont l'emplacement mémoire est en pile) avec un registre, mais il faut garder en tête que c'est une **optimisation**, et non une approche générale
- Raison : avoir des codes plus petits, qui ont moins d'instructions (et donc plus rapides)
- Mais même dans ce cas, on alloue la place correspondant à la variable locale optimisée en pile

Exemple avec variables locales optimisées en registre

Code C	Code ASM

```c
void main() {
    int a = 5;
    int b = 9;
    int c;
    c = a + b;
    exit();
}
```

```
addiu $29, $29, -12 # alloue 3 mots
ori   $8, $0, 5     # a <- 5
ori   $9, $0, 9     # b <- 9
addu  $10, $8, $9   # c <- 14
addiu $29, $29, 12  # désallocation
ori   $2, $0, 10
syscall
```

Remarque

- a est optimisée dans le registre $8, b est optimisée dans le registre $9, c est optimisée dans le registre $10

Exemple de programme complet

Programme C : passer une chaine en majuscule

```
char str[] = "helloworld";

int main() {
    int i = 0;
    while (str[i] != '\0') {
        str[i] = str[i] - 0x20;
        i = i + 1;
    }
    return 0;
}
```

Exemple de programme complet

Programme Mips

```
.data
str: .asciiz "helloworld"

.text
main:
  addiu $29, $29, -4 # Une variable locale : i
  sw    $0, 0($29)   # i = 0
  lui   $8, 0x1001   # $8 = 0x10010000 = @str
loop:
  lw    $9, 0($29)   # lecture i
  addu  $9, $8, $9   # $9 = &str[i] = @str + i
  lb    $9, 0($9)    # $9 = str[i]
  beq   $9, $0, finloop

  # Corps du while
  lw    $9, 0($29)   # lecture i
  addu  $9, $8, $9   # $9 = &str[i] = @str + i
  lb    $9, 0($9)    # $9 = str[i]
```

Exemple de programme complet

Programme Mips (suite)

```
  addiu $9, $9, -0x20 # str[i] - 0x20
  # Écriture en mémoire
  lw    $10, 0($29)   # Lecture i
  addu  $10, $8, $10  # $10 = &str[i] = @str + i
  sb    $9, 0($10)    # Écriture str[i]
  # i = i + 1
  lw    $9, 0($29)    # Lecture i
  addiu $9, $9, 1     # i + 1
  sw    $9, 0($29)    # Écriture i
  # Retour au début de la boucle
  j     loop
finloop:
  addiu $29, $29, 4   # Désallocation dans la pile

  ori   $2, $0, 0
  jr    $31           # return 0
```

Exemple de programme complet

Reqmarques sur le programme Mips

- Un peu long, mais à peu de chose près, c'est exactement le code généré par gcc
- Traduction facile à automatiser : c'est ça qu'il faut savoir faire
- Néanmoins, on peut largement optimiser ce code (à la main ou avec -O2)
- Exemple d'optimisation "violente" au slide suivant (à ne pas faire en TD/TP/Examen) : on fait disparaitre la variable i
  - Idée que l'on peut s'affranchir des règles quand on les maitrise
- Domaine de la compilation (= 👍)

Exemple de programme complet

Programme Mips optimisé (à ne pas retenir)

```
str: .asciiz "helloworld"

main:
  addiu $29, $29, -4
  lui   $8, 0x1001
loop:
  lb    $9, 0($8)
  beq   $9, $0, fin
  addiu $9, $9, -0x20
  sb    $9, 0($8)
  addiu $8, $8, 1
  j     loop
fin:
  addiu $29, $29, 4
  ori   $2, $0, 0
  jr    $31
```

Appels de fonction

Conventions de liaison en Mips

- Ensemble des règles pour que deux fonctions puissent communiquer entre elles
- Ces deux fonctions ne sont pas forcément écrites par la même personne/produites par le même compilateur
- Importance de les respecter

Rappel

- Un registre **persistant** est un registre dont la valeur est conservée au travers des appels de fonction
- $\Rightarrow$ Une fonction doit sauvegarder (au début) et restaurer (à la fin) tous les registres persistants qu'elle utilise

Communication entre les fonctions appelante/appelée

Passage de paramètres

- Les 4 premiers paramètres sont passés par registre ($4 à $7)
- Les suivants sont passés dans la pile (pas assez de registres) : allocation + écriture des valeurs
- Les paramètres passés par registre ont quand même une place réservée en pile par la fonction appelante
- Idée derrière :
  - Peu de fonctions ont plus de 4 paramètres, donc s'il n'y en a pas plus, la fonction appelée n'est pas obligée de faire d'accès mémoire (couteux)
  - C'est la fonction appelée qui décide si elle optimise les paramètres en registre $\Rightarrow$ La place en pile doit être allouée
  - De plus, si elle a besoin de réutiliser $4 à $7 (par exemple, si elle rappelle une fonction), elle stocke les valeurs des paramètres aux emplacements prévus en pile

Communication entre les fonctions appelante/appelée

Valeur de retour

- La valeur de retour d'une fonction est passée dans le registre $2 + éventuellement $3 (par exemple si mot de 64 bit)

Adresse de retour

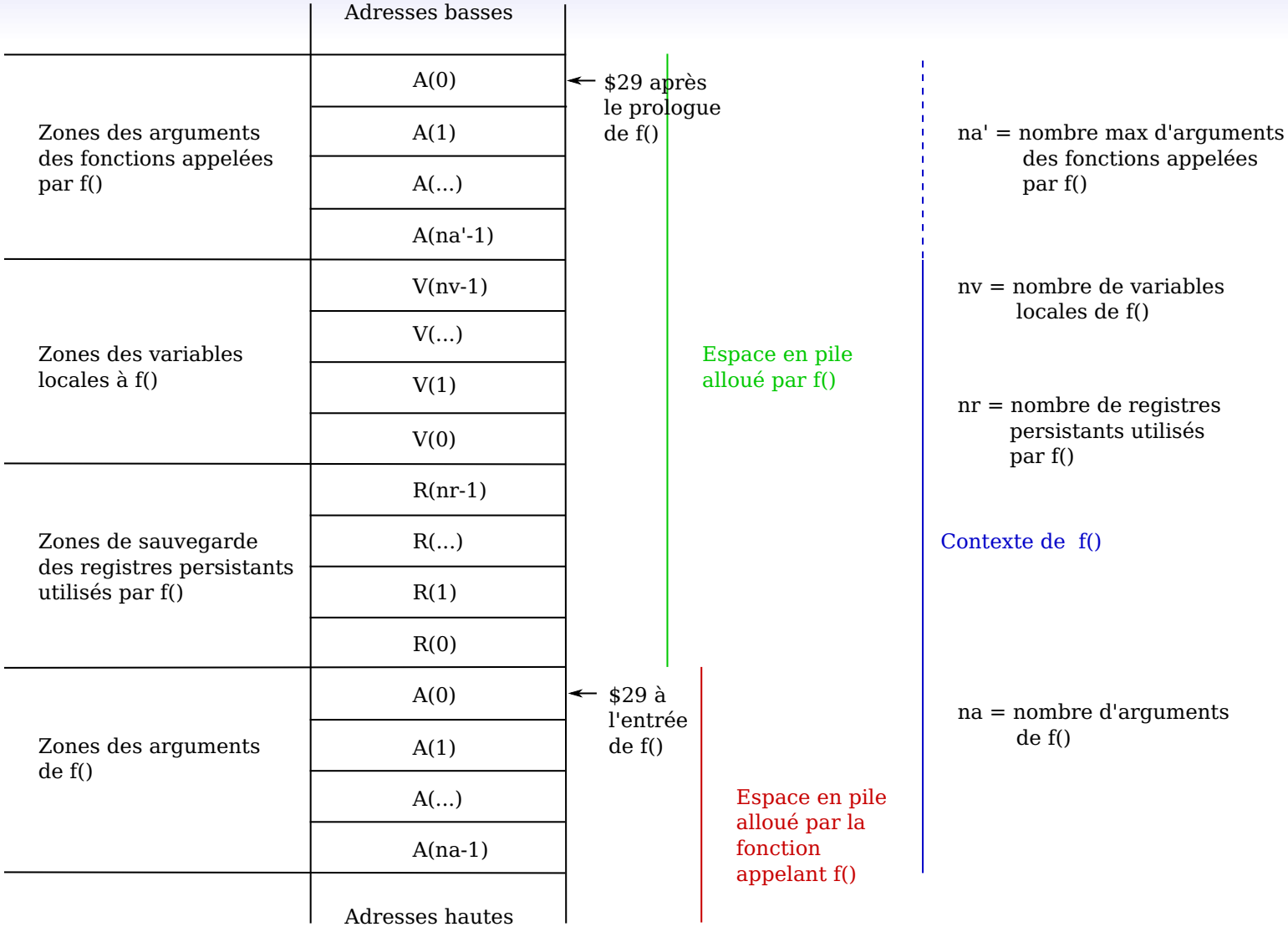
- L'adresse de retour est passée à la fonction appelante dans $31
- L'écriture de ce registre à la bonne valeur est faire directement en matériel, par l'instruction `jal`
  - i.e. l'instruction jal, en plus du saut, écrit dans $31 l'adresse de l'instruction suivant le `jal`
- À la fin d'une fonction, on exécute l'instruction `jr` $31
- Si la fonction n'est pas terminale, il faut sauvegarder et restaurer $31

Appel de fonction

Récapitulatif

- Au début d'une fonction, il faut allouer de la place en pile pour :
  - Sauvegarder les registres persistants utilisés par la fonction (dont $31)
  - Les variables locales
  - Les paramètres des fonctions qu'on va appeler (réutilisation des mots pour les différentes fonctions $\Rightarrow$ Max du nombre de paramètres des fonctions appelées)
- À la fin de la fonction, il faut désallouer la même place en pile
- L'allocation + la sauvegarde des registres constituent le **prologue** de la fonction
- La restauration des registres + la désallocation constituent l'**épilogue** de la fonction

Appel de fonction : vision de la pile



Appel de fonction : exemple (code C)

```c
int x = 2, y = 4, z = 5;

int main() {
    int t1, t2;
    t1 = max2(x, y);
    t2 = max2(t1, z);
    printf("%d", t2);
    return 0;
}

int max2(int a, int b) {
    if (a < b) {
        return b;
    }
    else {
        return a;
    }
}
```

Appel de fonction : code ASM

Code du main

```
.data
x: .word 5 # @0x10010000
y: .word 4 # @0x10010004
z: .word 9 # @0x10010008

.text
.globl main

main:
  # Prologue
  addiu $29, $29, -20 # Allocation de 5 mots : 2 vars locales
                      # + 2 paramètres pour max2 + $31
  sw    $31, 16($29)  # Sauvegarde de $31
  # t1 = max2(x, y)
  lui   $8, 0x1001    #
  lw    $4, 0($8)     # Paramètre a <- x
  lw    $5, 4($8)     # Paramètre b <- y
  jal   max2          # Premier appel à max2
  sw    $2, 12($29)   # Écriture du résultat dans t1
```

Appel de fonction : code ASM

Code du main (fin)

```
# t2 = max(t1, z)
lw     $4, 12($29)    # Paramètre a <- t1
lui    $8, 0x1001     # $8 est non persistant
lw     $5, 8($8)      # Paramètre b <- z
jal    max2           # Second appel à max2
sw     $2, 8($29)     # Écriture du resultat dans t2

# printf("%d", t2)
lw     $4, 8($29)     # $4 <- t2
ori    $2, $0, 1      # Appel système affichage entier
syscall

# Épilogue
lw     $31, 16($29)   # Restauration de $31
addiu  $29, $29, 20   # Désallocation
# return 0
ori    $2, $0, 0      # Valeur de retour du main
jr     $31
```

Appel de fonction : code ASM

Code de max2 : prologue

```
max2:
  # Prologue
  addiu $29, $29, -16   # Allocation de 4 mots :
  sw    $31, 12($29)    # 3 registres persistants
  sw    $18, 8($29)     # + $31 (toujours sauvegardé)
  sw    $17, 4($29)     # sauvegardés dans le
  sw    $16, 0($29)     # prologue
  sw    $4, 16($29)     # écriture a
  sw    $5, 20($29)     # écriture b
```

Appel de fonction : code ASM

Code de max2 : corps et épilogue

```
  # Corps
  lw     $16, 16($29)    # Lecture a
  lw     $17, 20($29)    # Lecture b
  slt    $18, $16, $17   # 1 ssi a < b
  beq    $18, $0, _else  # branch si a >= b
  lw     $2, 20($29)     # $2 <- b
  j      _fin
_else:
  lw     $2, 16($29)     # $2 <- a
_fin:

  # Épilogue
  lw     $31, 12($29)    # Restauration des
  lw     $18, 8($29)     # registres persistants
  lw     $17, 4($29)
  lw     $16, 0($29)
  addiu  $29, $29, 16    # Désallocation en pile
  jr     $31             # return
```