Architecture des microprocesseurs
RISC-V ABI
CRT & édition de liens

Michel Agoyan : michel.agoyan@st.com
Marc Lacruche : marc.lacruche@st.com
Simon Pontie : simon.pontie@cea.fr
Olivier Potin : olivier.otin@emse.fr
Anthony Zgheib : zgheib@emse.fr
Louis Noyez : louis.noyez@emse.fr
Théophile Gousselot : theophile.gousselot@emse.fr
Clément Fanias : clement.fanias@emse.fr

MINES Saint-Étienne

Cache : amélioration des performances

- Average access time = hit time + Miss rate x Miss penality
- 3 voies possibles pour améliorer les performances :
  - Réduire le miss rate (cache de taille plus importante)
  - Réduire le “Miss penality” (utilisation d’un niveau hiérarchique supplémentaire : L2
  - Réduire le “Hit time”

Cache : causes des « Miss »

- Obligatoire : première fois que l'on référence un bloc
  - Cas de "Miss" qui a lieu même si le cache est de capacité infinie
- Capacité : taille du cache trop petite pour contenir toutes les données/instructions nécessaires au programme
- Conflit : Collisions due à la stratégie de remplacement des blocs

Cache : Optimisation logicielle

* Changer l'ordre de parcours de tableau multi-dimensionnel :

```
For(i=0; i< M; i++){
      for(j=0; j <N; j++){
            x[i][j] = 2*x[i][j];
      }
}
```

```
For(j=0; j< N; j++){
      for(i=0; i <M; i++){
            x[i][j] = 2*x[i][j];
      }
}
```

* Améliore la localité spatiale

Cache : Optimisation logicielle

* Fusion de boucles itératives :

```
For(i=0; j< N; i++){
      for(j=0; I <M; j++){
            a[i][j] = b[i][j] * c[i][j] ;
      }
}
For(i=0; j< N; i++){
      for(j=0; I <M; j++){
            d[i][j] = a[i][j] * c[i][j] ;
      }
}
```

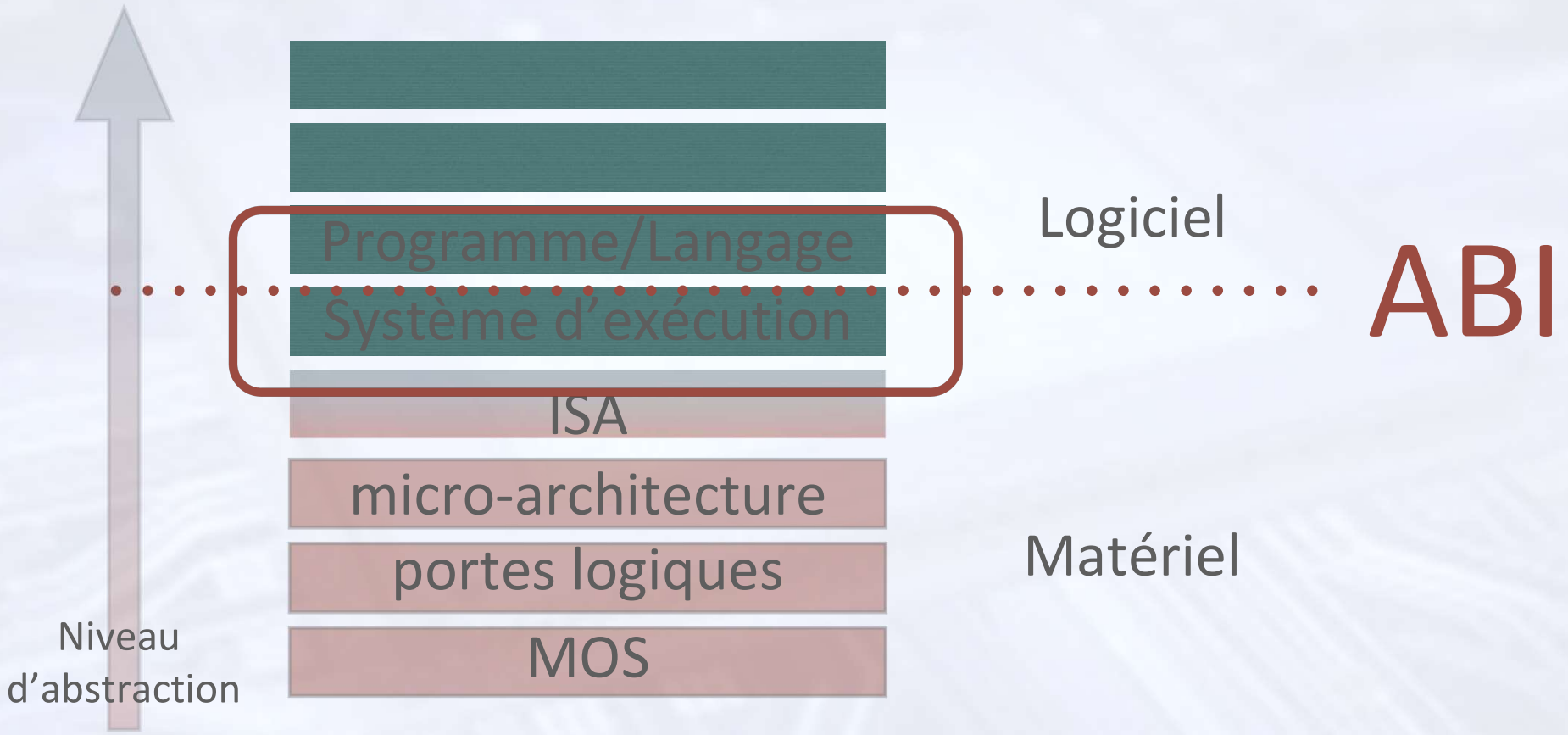
```
For(i=0; i< M; i++){
      for(j=0; j <N; i++){
            a[i][j] = b[i][j] * c[i][j] ;
            d[i][j] = a[i][j] * c[i][j] ;

      }
}
```

* Améliore la localité temporelle

MINES Saint-Étienne

RISC-V ABI (Application Binary Interface)



MINES Saint-Étienne

RISC-V ABI (Application Binary Interface)

- Programmation procédurale est le paradigme de programmation le plus utilisé. (paradigme implémenté dans pratiquement tous les langages)
  - modularité
  - factorisation
  - lisibilité
- Problème : comment assurer l'interopérabilité entre les langages ?

RISC-V ABI (Application Binary Interface)

- ABI définit une convention pour l'appel à une procédure/fonction garantissant qu'une procédure/fonction écrite dans un langage puisse être appelée depuis un programme écrit dans un autre langage
- Cette convention garantit aussi que des chaînes d'outils différentes produisent des codes exécutables compatibles entre eux
- Il peut exister plusieurs convention d'appel. RISC-V : ilp32, ilp32d, ilp32e, ilp32f, lp64, lp64d, lp64f,

RISC-V ABI
Appel de fonction

* Un appel de fonction se décompose en 7 étapes :

1. Transfert des paramètres à la fonction
2. Transfert du flot d'exécution à la fonction
3. Allocation de ressource mémoire
4. Exécution du traitement
5. Transfert du résultat du traitement
6. Libération des ressources mémoires
7. Retour au flot d'exécution principal

RISC-V ABI
la pile :transfert des paramètres

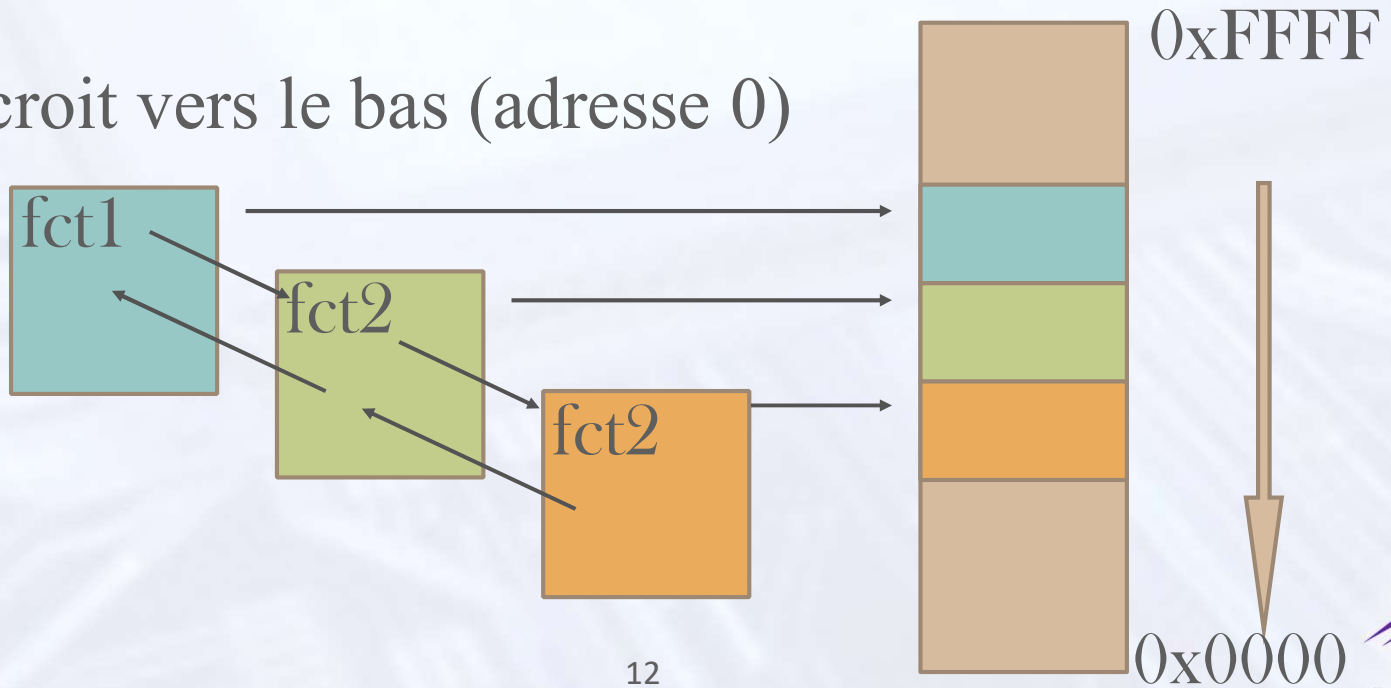
- Les registres du processeur semblent être un choix efficace en terme de performance pour le passage des paramètres à une fonctions :
  - Registres = structure mémoire la plus rapide
  - La plupart des fonctions d'un programme ont moins de 8 paramètres
- Si plus de 8 paramètres . Les registres sont en nombre limité ⇒ utilisation de la RAM

RISC-V ABI
utilisation des registres

N° de registre	nom	description	Responsable de la sauvegarde
0	zero	Câblé à 0	--
1	ra	Adresse de retour	appelant
2	sp	Pointeur de pile	appelant
3	gp	Pointeur global	--
4	tp	Pointeur de “Thread”	--
5..7	t0...t2	Temporaires à sauvegarder	appelant
8	s0/fp	Registre à sauvegarder 0 / pointeur de “frame”	appelé
9	s1	Registre à sauvegarder 0	appelé
10-11	a0,a1	2 premiers paramètres , valeurs de retour	appelant
12-17	a2..a7	Paramètres	appelant
18-27	s2-s7	Registres à sauvegarder	appelé
28-31	t3-t6	Registres temporaires	appelant

RISC-V ABI
la pile : allocation de ressources mémoire

* Allocation se fait dynamiquement à l'aide d'une pile
* On dispose d'un pointeur de pile le registre 2 « sp » qui pointe sur le haut de la pile
* la pile croit vers le bas (adresse 0)

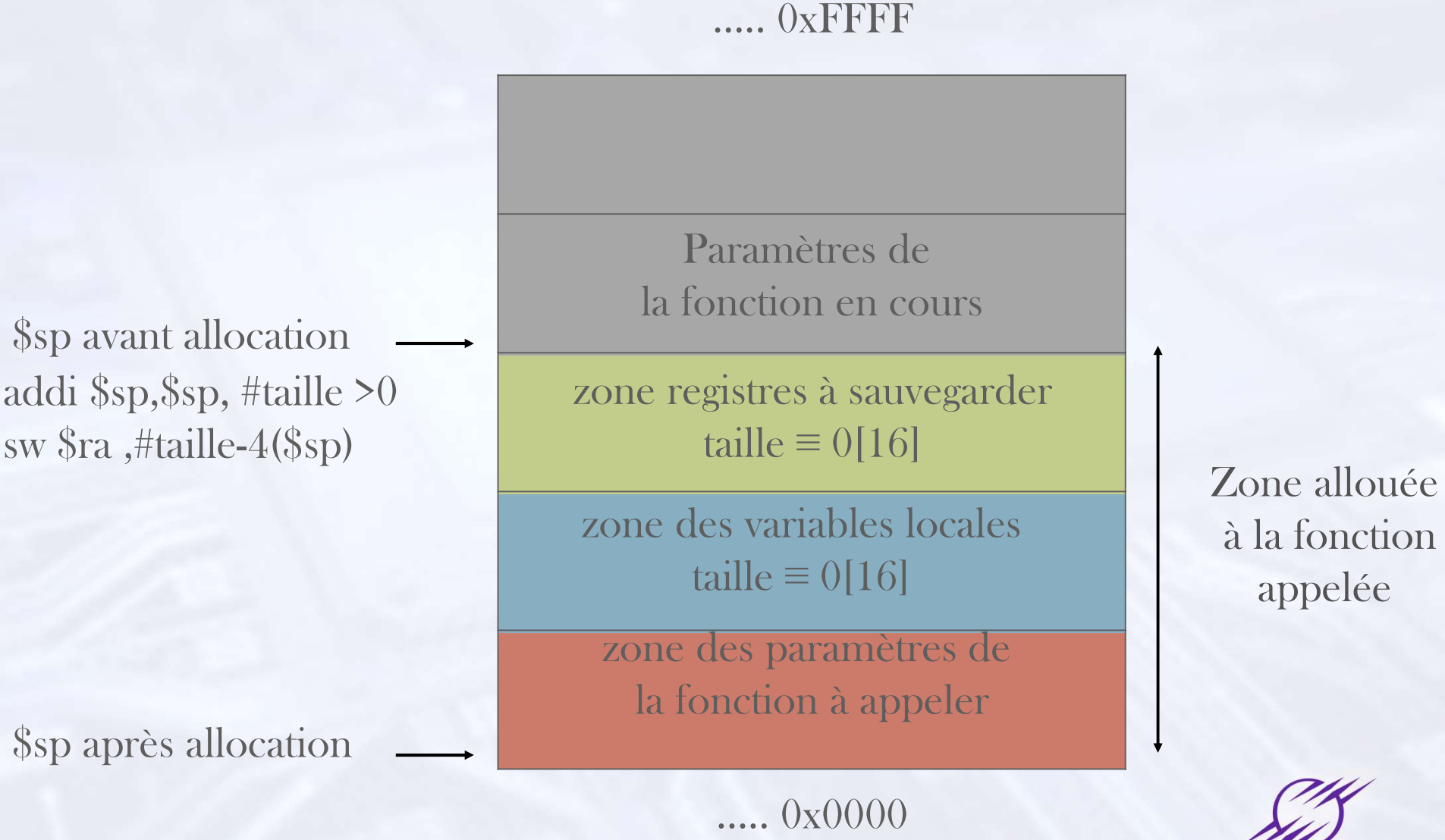


MINES Saint-Étienne

RISC-V ABI
La pile : Allocation/libération de ressources mémoire

- L'espace mémoire alloué pour une fonction sur la pile s'appelle : la «frame»
- Pour allouer de l'espace sur la pile on fait croitre la pile vers le bas:
  - addi $sp,$sp,#taille négative
- Pour libérer l'espace précédemment allouer :
  - addi $sp,$sp,#taille positive
- La taille de la «frame» doit être un multiple de 16 (pour pouvoir empiler les variables scalaires les plus larges = long double)

RISC-V ABI
la pile : structure



MINES
Saint-Étienne

RISC-V ABI

Structure de la pile : zone des paramètres de la fonction à appeler

* Les premiers paramètres (pour le langages C , l'ordre des paramètres d'une fonction est de gauche à droite) sont passés par les 8 registres : $a0,$a1,$a2,$a3,$a4,$a5,$a6,$a7
* Les paramètres supplémentaires sont placés sur la pile.
* Bien que les premiers paramètres sont passés par registres , les 8 premiers mots de la pile de la fonction appelée sont tout de même alloués sur la pile de la fonction appelée
* La fonction appelée est libre d'utiliser la zone de stockage des paramètres comme zone de mémoire de travail
* La taille de la zone de stockage des paramètres doit être suffisamment grande pour accueillir l'ensemble des paramètres le plus important (si plusieurs fonctions appelées)

RISC-V ABI
Structure de la pile : zone des variables locales

- Si la fonction appelée a besoin de variables locales, elles sont allouées sur la pile dans une zone dédiée au dessus de la zone de stockage des paramètres.
- La taille de cette zone doit être un multiple de 16
- L'utilisation de registres est privilégiée (ou peut être demandé par le programmeur)
  - Le débogueur aura du mal a identifié les variables locales
    ⇒ débrailler les optimisations pour le debug

RISC-V ABI
Structure de la pile :
zone des registres à sauvegarder

- Si la fonction utilise un des registre s0 , s1 à s11
- Dans le prologue de la fonction les registres à sauvegarder utilisés par la fonction sont placés sur la pile dans cette zone
- Les registres à sauvegarder seront restaurés dans l'épilogue de la fonction

RISC-V ABI
Transfert du flot d'exécution à la fonction et retour au flot principal

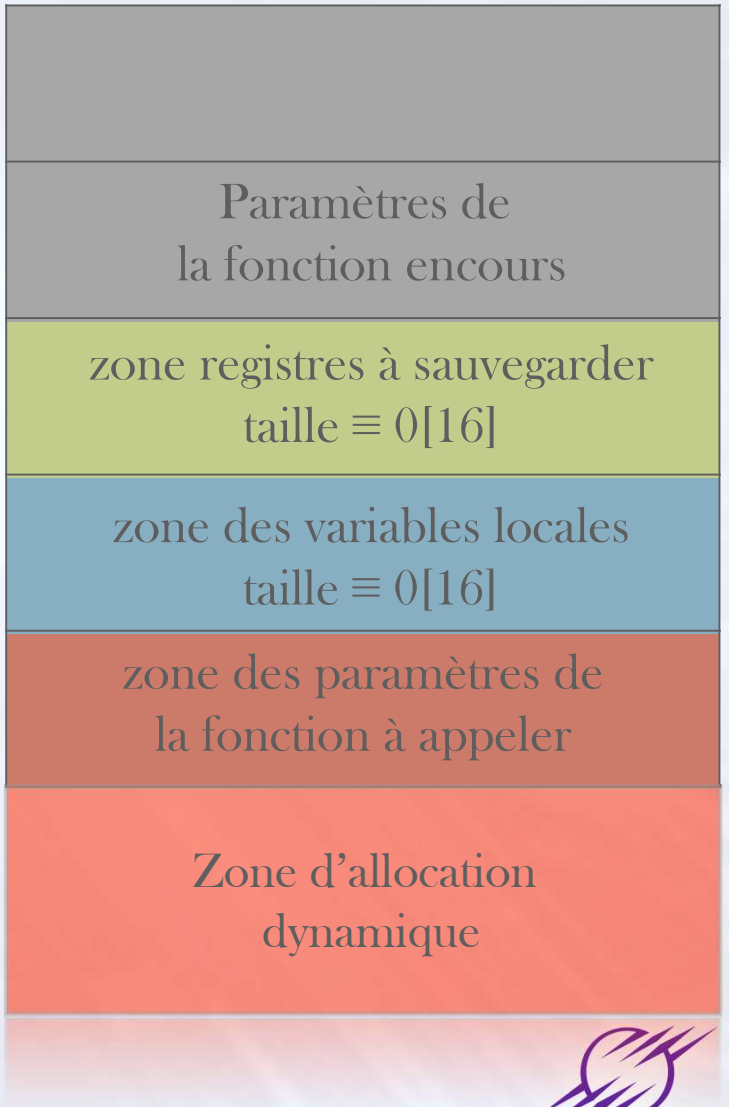
- JAL fct : saut vers fct et sauvegarde de la l'adresse de retour dans $ra
- Si la fonction appelée n'est pas une fonction feuille (elle appelle une ou plusieurs autres fonctions) $ra doit être sauvegardé
- Durant l'épilogue $ra est sauvegardé sur la pile au dessus de la zone des registres à sauvegarder
- $ra est restauré dans la toute fin de l'épilogue juste avant le retour
- JR $ra

RISC-V ABI
retour du résultat à l'appelant

- $a0 et $a1 sont utilisés pour retourner le résultat à l'appelant
- La plupart des fonctions retourne un résultat <32bits , un seul registre est utilisé : $a0
- Le retour de résultat > 64 bits se fait généralement par un pointeur sur une zone mémoire de l'appelant ⇒ ne pas oublier que la zone de mémoire allouée à la fonction est libérée dès le retour de fonction

RISC-V ABI
Allocation dynamique sur la pile

- Taille de variable locale déterminée lors de l'exécution (__builtin_alloca)
- Utilisation d'un registre sauvegardé particulier $s0= $fp
  - En fait toutes les références à la pile sont faites en utilisant $fp (frame pointeur)
  - Pour créer une zone dynamique il suffit de décrémenter $sp
  - $fp et $sp seront restaurés durant l'épilogue de la fonction



MINES
Saint-Étienne

RISC-V ABI
variables globales

- Les variables globales sont accessibles en utilisant $gp (adressage indirect avec déplacement) en une seule instruction :
  - lw $a1, #offset($gp)
  - offset = valeur immédiate sur 12bit = 4KB ,est calculé lors de la construction
  - $gp doit être initialisé au démarrage
- Pour certain compilateur (gcc) on peut régler un seuil (-G —gpsize) pour décider si la variable est accessible en utilisant $gp ou non :
  - lui $t0,%hi(global) (%hi est une macro assembleur prenant les 20bits de poids fort d'un mot de 32bits , ici l'adresse de la variable « global »)
  - addi $t0,%lo(global)
  - lw $a1,($t0)
  - Moins performant que l'accès par $gp seul

RISC-V
Compilation C :1-2

- La compilation : 3 étapes

  1.pré-processeur (inclusion des entêtes , développement des macros)

  2.traduction en assembleur

  3.assemblage ⇒ fichier binaire appelé «fichier objet »

- Fichier objet :
  - Instructions machine
  - Calcul d'adresse incomplet et relatif :
    - certaines références à des fonctions ou variables sont définies
    - certaines références sont externes et sont non résolues

RISC-V
Compilation 2-2

- Les références sont classées dans des sections de sorties :
- .text :
  - le code exécutable = instructions machine
  - les constantes
- .bss, .sbss : les variables globales non itialisées
- .data, .sdata : les variables globales initialisées

RISC-V
édition de lien

* L'éditeur de lien assure deux fonctions :

1. Il résout les références non définies en établissant les liens entre les modules qui définissent les références et les modules qui les utilisent

2. Il collecte les références d'une même section et calcule leurs adresses

* Le fichier de sortie est un fichier binaire (pour gcc format ELF (exécutable and linkable format) contenant les sections « logées » et éventuellement une table de symbole pour le debug

RISC-V
édition de lien : script de commande

* L'édition de lien est guidée par un script de commande :
* Commande MEMORY :

```
MEMORY
{
  rom : o=0x00000000,l=128k
  ram : o=0x10000000,l=64k
}
```

* name : o = origin, l = length

RISC-V
édition de lien : script de commande

Sections d'entrée déclarées par l'utilisateur

Collecte des sections d'entrées

zone mémoire cible

```
SECTIONS
{
 .text :
 {
  __text_start = . ;
  *(.start)
  *(.text)
  *(.rodata)
  __text_end = . ;
 }>rom
 .zero_dat
 {
   __bss_start = . ;
    _global_pointer$= .;
  *(.bss)
   __bss_end  = . ;
 }>ram
....
}
```

Sections de sortie

MINES Saint-Étienne

RISC-V
édition de lien : script de commande

Directive AT :
Calcule les adresses pour placer les sections dans la zone mémoire cible, mais charge les données à l'adresse spécifiée par AT c'est à dire :
___text_end

Allocation de la pile :
stack_size = 1024
directive ALIGN(16)
définition d'un symbole _sp

```
 SECTIONS
{
.....
 .init : AT(__text_end)
 {
   __data_start = . ;
  * (.data)
   __data_end =  . ;
 }>ram
 .stack
  {
     . = ALIGN (16);
    __stack_start = . ;
     . = . + stack_size;
    __stack_end  = . ;
    _sp = . ;
  }>ram
}
```


MINES Saint-Étienne

RISC-V
C startup

- Programme d'initialisation :
- Généralement écrit en assembleur :
- Initialisation du processeur : remise à zéro des registres , gestions des exceptions et interruptions
- Initialisation de l'environnement d'exécution c :
  - init de $sp à l'aide du symbole __stack_end
  - remize à zéro de la section . bss des variables globales non initialisées (__bss_start, __bss_end)
  - Copie des valeurs d'init depuis la rom vers la ram pour les variables globales initialisées , section .data (de __text_end vers __data_start sur une longueur = __data_end - __data_start)
- Branchement vers le main : j main

RISC-V
C startup

```
.section .start;
.globl start;

start:
 //  registers init
li t0,0
li t1,0
...

 // Setup sp and gp
  la  gp, __global_pointer$
  la  sp, _sp
```

```
 // Clear uninitialized (zeroed)
data sections

  la  t1,__bss_start
  la  t2,__bss_end


clr_lp:
  sw    $0,0(t1)
  addi  t1,t1,4
  bltu  t1,t2,clr_lp
  nop
```

RISC-V
C startup

```
// Copy initialized data sections from ROM into RAM
 la      t1,__data_start
  la      t2,__data_end
  la      t3,__tex_end

cp_lp:
  lw      t0,0(t3)
  sw      t0,0(t1)
  addi    t1,t1,4
  addi    t3,t3,4
  bgtu    t2,t1,cp_lp
  nop
 j main
.end start
```

C pour l'embarqué
volatile

* Gestion d'un périphérique :

* écritures successives vers un même registre (FIFO , registre Tx d'une UART ….)

```
Unsigned int *ptr;
ptr*=0x55;
ptr*=0xAA;
```

  * Partage d'un flag de statut d'un périphérique entre une routine d'interruption et le programme principal

```
Unsigned int flag=0;

While(flag !=0);
```

```
Isr ()
{
Flag =1
}
```

* Que fait le compilateur dans ce cas ?

* Comment débrailler les optimisations du compilateur ?

C pour l'embarqué
volatile

- Le qualifieur volatile :
  - Il indique au compilateur de ne faire aucune supposition sur le code qui manipule la variable (rupture du flot d'exécution : interruption, rtos …)
  - Les optimisations sont débraillées
  - Il ne s'applique que sur des variables globales
  - Exemple précédent : Volatile unsigned int * ptr;
    - Ptr pointe sur un entier non signé volatile

C pour l'embarqué
fonction avec un nombre de paramètres variable

- void fct(int param1, …)
- 3 macros C définies dans stdarg.h pour récupérer les paramètres sur la pile :
  - void va_start(va_list ap, last_arg) : initialise ap , last_arg est le dernier paramètre fixe (param1 dans l'exemple plus haut)
  - type va_argv(va_list ap,type) : récupère le paramètre suivant de type « type »
  - void va_end(va_list ap) : épilogue