Conception et exploitation des processeurs

Frédéric Pétrot

GRENOBLE INP UGA Ensimag

Équipe pédagogique :
Julie Dumas, Claire Maiza, Olivier Muller, Frédéric Pétrot, Lionel Rieg (resp.),
Manu Selva et Sebastien Viardot

Année universitaire 2021-2022

C1	Présentation du projet CEP et rappels de VHDL
C2	Chaîne de compilation et assembleur RISC-V
C3	Conventions pour les appels de fonctions
C4	Gestion des interruptions par le logiciel

Sommaire

Introduction

Fonction

Ensemble d'instructions réalisant une tâche donnée pouvant être appelé à divers points d'un programme et retournant après le point d'appel une fois exécuté
Une fonction :

- prend 0, 1 ou *n* paramètre(s) [a]
  retourne potentiellement un résultat
- utilise des variables :
  - *locales* dont la durée de vie est celle de la fonction
  - *globales statiques* dont la durée de vie est celle du programme
  - *globales dynamiques* dont la durée de vie est *déterminée* par le programme

a. Ce nombre étant possiblement variable pour une fonction donnée, par exemple `printf` en C. Nous ne considérerons pas ces fonctions dites « variadiques » dans ce cours.

Introduction

Fonctions : notion de fonction « appelante » et de fonction « appelée »

```
/* Fonction appelante */
uint32_t sos(int32_t x)
{
   /* variables locales */
   uint32_t u, v, w;
   u = square(x);
   /*^point de retour de la
      première invocation */
   v = square(x);
   /*^point de retour de la
      deuxième invocation */
   w = u + v;
   /* valeur de retour */
   return w;
}
```

```
/* Fonction appelée */
uint32_t square(int32_t y)
{
   /* variable locale */
   int32_t u;
   u = y * y;
   /* valeur de retour */
   return u;
}
```

Introduction

En pratique

- toutes les fonctions sont appelées. Il existe un bout de code non appelé : le *boot* du système (mais ce n'est pas une fonction *stricto sensu*)
- certaines fonctions sont appelantes, *i.e.* elles appellent d'autres fonctions
- d'autres non : on les appelle fonctions feuilles

Une fonction étant appelée de plusieurs « endroits », chaque invocation de la fonction s'exécute dans un « contexte » particulier et retourne dans le « contexte » d'où elle a été appelée

Sommaire

Contexte d'exécution

Contexte d'exécution d'une fonction appelée

- paramètres
- variables locales
- registres dont les valeurs doivent être sauvegardées car l'appelante s'attend à les retrouver non modifiés au retour
- adresse de retour dans l'appelante

Principes

- création d'un contexte d'exécution pour chaque invocation
- utilisation du contexte pour l'exécution de la fonction
- destruction du contexte lors du retour de fonction
- contextes crées et détruits dans une *pile*
  - appels imbriqués $\rightarrow$ empilement et dépilement des contextes
  - appels successifs $\rightarrow$ création contextes dans même zone de pile

Contexte d'exécution

Exemple générique de structuration de la pile (*stack frame*)

adresses hautes	...		
	paramètres pour l'appelée	optionnel	appelante
⇓	adresse de retour	optionnel	
	variables locales	optionnel	appelée
⇓	registres à sauver	optionnel	
	paramètres pour l'appelée suivante	optionnel	
adresses basses	...		

Variations suivant les processeurs

- pile descendante ou ascendante [a]
- ordre des éléments dans pile liées aux conventions précises
- inutile d'empiler l'adresse de retour dans les fonctions feuilles
- alignement des éléments dans la pile variable selon les architectures
- ...

a. Parmi les machines supportées par Linux, seul le *HP-Precision Architecture* possède une pile ascendante.

Contexte d'exécution

Programmation des fonctions

Repose sur des conventions spécifiques à chaque processeur :

- utilisation des registres
- support matériel *ad-hoc*

C'est l'*Application Binary Interface* (ABI)

ABI

- différentes conventions possibles pour la même machine
  - généralement définies par les constructeurs, incompatibles
  - x86 : 13 différentes [a]
  - MIPS32 : 3 différentes, Ultrix o32 utilisée par gcc [b]
  - RISC-V : une seule, PSAbi, ouf !
- nécessaire aux systèmes d'exploitation

a. `http://en.wikipedia.org/wiki/X86_calling_conventions`
b. `http://www.linux-mips.org/wiki/MIPS_ABI_History`

Sommaire

ABI RISC-V

Instructions d'appel et retour de fonction

- `jal rd, func` : sauve adresse instruction suivante dans `rd` et saute à adresse étiquette `func`
- `jalr rd, offset(rs)` : sauve adresse instruction suivante dans `rd` puis saute à adresse obtenue en faisant `rs + offset`
- `jr zero, rd` : retourne à l'appelante
- peuvent s'écrire `jal func`, `call func`, et `ret` grâce aux pseudo-instructions `rd` est alors implicitement `ra`

Registres conventionnels

- `ra` : contient l'adresse de retour de la fonction
- `a0` : contient la valeur de retour de la fonction
- `sp` : contient l'adresse de la *dernière case* occupée dans pile

ABI RISC-V

Paramètres

- 8 premiers paramètres dans `a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`, `a7`
- paramètres suivants dans la pile
- remarque : `a0` est *à la fois* le 1er paramètre *et* la valeur de retour

Mise en place du contexte dans la pile

Une responsabilité partagée :

- appelante : positionne les paramètres dans les registres (puis dans la pile si nécessaire), saute à l'appelée
- appelée : réserve la place nécessaire à son propre contexte d'exécution

ABI RISC-V

Convention détaillée d'usage des registres

Nom matériel	Nom logiciel	Signification	Préservé lors des appels ?
x0	zero	Zéro	Oui (Toujours zéro)
x1	ra	Adresse de retour	Non
x2	sp	Pointeur de pile	Oui
x3	gp	Pointeur global	Ne pas utiliser
x4	tp	Pointeur de tâche	Ne pas utiliser
x5-x7	t0-t2	Registres temporaires	Non
x8-x9	s0-s1	Registres préservés	Oui
x10-x17	a0-a7	Registres arguments	Non
x18-x27	s2-s11	Registres préservés	Oui
x28-x31	t3-t6	Registres temporaires	Non

- *préservé* lors des appels :
  appelée doit rendre ces registres inchangés
  doit donc les sauver avant modification et les restaurer avant le retour
- *non préservé* lors des appels :
  appelante doit sauver avant appel et restaurer au retour si veut conserver valeur
  appelée peut les modifier sans précautions

Mise en place du contexte d'exécution

Principe général

Dans la fonction appelée (Rappel : toutes les fonctions sont appelées, …)

- prologue :
  - calculer place nécessaire aux variables locales, n_v, et registres à préserver, n_r
    $\Rightarrow$ faisable car code fonction connu !
  - si fonction appelle une ou plusieurs fonctions : déterminer nombre de paramètres maximum passés à ces fonctions, n_p
  - décrémenter le pointeur de pile en fonction [a] de n_v, n_r et n_p afin qu'il pointe sur la dernière case du contexte
  - sauver les registres qui doivent l'être dans la zone idoine de la pile
- code de la fonction. Il y est souvent nécessaire de sauver et récupérer des registres
- épilogue :
  - restaure les registres qui doivent l'être à partir de la zone idoine de la pile
  - repositionne pointeur de pile sur adresse pile avant appel
  - retourne à l'appelante

a. La « fonction » dépend de l'ABI

ABI RISC-V

Structure de la pile (*stack frame*)

adresses hautes	...		
↓	espace pour paramètres n°10	optionnel	
↓	espace pour paramètres n°9	optionnel	appelante
↓	adresse de retour	optionnel	appelée
↓	registres à sauver	optionnel	
↓	variables locales	optionnel	
↓	espace pour paramètres n°n_{p-1}	optionnel	
↓	...	optionnel	
↓	espace pour paramètres n°10	optionnel	
↓	espace pour paramètres n°9	optionnel	appelante'
↓	adresse de retour	optionnel	appelée'
adresses basses	...		

Remarques sur la pile

- s'il y a plus de 8 paramètres, appelante met dans pile
- `sp` ne change plus entre prologue et épilogue
- `jal` n'empile pas l'adresse de retour
  ⇒ optimisation si fonction feuille
  responsabilité de l'appelée (qui doit faire un `sw ra, n(sp)`)

ABI RISC-V

```
f(...) {
   g(5 params);
   k(2 params);
}
g(...) {
   h(n params);
}
```

déplacement `sp` dans `f()` compatible appel `g()` *et* appel `k()`
mise en place arguments dépendant de déplacement

adresses hautes	variables locales de f	sp dans appelante (f)
↓	valeur de `ra`	adresse de retour
↓	autres registres	
↓	à sauver	
↓	variables locales de g	
↓	paramètre $n-1$	préparation de l'appel
↓	...	de la fonction h
↓	paramètre 8	sp dans appelée (g)
adresses basses	...	

Notes

- pas de paramètres dans la pile de `f`
- adresse de retour est adresse instruction appel à `k`

ABI RISC-V : variables de types entier et pointeurs

`uint32_t, int32_t, uint16_t, int16_t, uint8_t, int8_t`
⇒ 1 registre ou 1 mot de 32 bits en pile
`uint64_t, int64_t`
⇒ 2 registres, ou 1 registre et 1 mot de 32 bits en pile ou 2 mots de 32 bits en pile

Mise en place du contexte

$N = (n_v + n_r + \max(0, n_p - 8)) \times 4$

- n_v nombre de variables en pile nécessaires
- n_r nombre de registres à sauver : temporaires à conserver ou registres à préserver
- n_p nombre maximum de paramètres de l'ensemble des fonctions *appelées*

Prologue

```
 addi sp, sp, -N     réserve N octets dans pile
[sw ra, N-4(sp)]     [empile adresse de retour, si non feuille]
```

Épilogue

```
[lw ra, N-4(sp)]     [dépile adresse de retour, si non feuille]
 addi sp, sp, N      remet pile comme à l'entrée
 ret                 retour appelante
```

ABI RISC-V : agrégats

Dépend de la taille de l'agrégat

- ≤ 32 passé dans un registre, tel qu'en mémoire
- ≤ 64 passé dans paire de registres $a_{i+1} : a_i$, en remplissant a_i des poids faibles vers les poids forts, puis identiquement pour a_{i+1}
- > 64 création d'une zone dans la pile de l'appelante, et passage de l'adresse dans le paramètre correspondant

Sommaire

Exemples

Fonction « feuille » :

```
int32_t plus(void)
{
   int32_t a = 5;
   int32_t b = 4;
   return a + b;
}
```

```
plus:
   li  a0, 5
   li  t0, 4
   add a0, a0, t0
   ret
```

Fonction appelant une fonction sans paramètres :

```
int32_t main(void)
{
   int32_t x;
   x = plus();
   return 0;
}
```

```
main:
   addi sp, sp, -2*4 # place pour x et ra
   sw   ra, 1*4(sp)
   jal  plus
   sw   a0, 0*4(sp)  # maj de x
   mv   a0, zero     # valeur de retour
   lw   ra, 1*4(sp)
   addi sp, sp,  2*4
   ret
```

Exemples

Passage et récupération de paramètres :

```
int32_t
plus(int32_t a, int32_t b)
{
   return a + b;
}

int32_t main(void)
{
   int32_t x;
   x = plus(5, 4);
   return 0;
}
```

```
plus:
   add  a0, a0, a1   # même registre pour valeur
   ret               # retour et 1er paramètre
main:
   addi sp, sp, -2*4 # place pour x et ra
   sw   ra, 1*4(sp)
   li   a0, 5
   li   a1, 4
   jal  plus
   sw   a0, 0*4(sp)  # maj de x
   mv   a0, zero     # valeur de retour
   lw   ra, 1*4(sp)
   addi sp, sp,  2*4
   ret
```

Exemples

Fonctions avec plus de 8 arguments (rare, mais possible)

```
int32_t mad(int p0, int p1, int p2, int p3, int p4, int p5, int p6, int p7, int p8, int p9)
{ dam(p9, p8); return p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9; }
```

- 8 premiers arguments dans les registres : a0 ← p0, ..., a7 ← p7
- 2 arguments suivants dans la pile de l'appelant

adresses hautes	40(sp)	p9	
↓	36(sp)	p8	sp' dans appelante
↓	32(sp)	valeur de ra	adresse de retour de mad
↓	28(sp)	place pour sauvegarder a7	
↓	24(sp)	place pour sauvegarder a6	
↓	20(sp)	place pour sauvegarder a5	
↓	16(sp)	place pour sauvegarder a4	
↓	12(sp)	place pour sauvegarder a3	
↓	8(sp)	place pour sauvegarder a2	
↓	4(sp)	place pour sauvegarder a1	
↓	0(sp)	place pour sauvegarder a0	sp dans mad (sp = sp'-36)
adresses basses		...	

- chaque argument occupe 4 octets, quelque soit son type
- fonction appelée accède le contexte de l'appelante pour récupérer les paramètres 9 et 10
- paramètres utilisés après appel fonction dam ⇒ il faut les sauver tous

Exemples

Fonctions avec plus de 8 arguments (rare, mais possible)

```
int32_t mad(int p0, int p1, int p2, int p3, int p4, int p5, int p6, int p7, int p8, int p9)
{
   dam(p9, p8); return p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9;
}
```

```
addi sp, sp, -9*4 # 8 param + ra
sw   ra, 32(sp)   # sauve ra
sw   a7, 28(sp)   # sauve
sw   a6, 24(sp)   # les
sw   a5, 20(sp)   # 8
sw   a4, 16(sp)   # registres
sw   a3, 12(sp)   # paramètres
sw   a2,  8(sp)   # de la
sw   a1,  4(sp)   # fonction
sw   a0,  0(sp)   # mad
lw   a0, 40(sp)   # prépare p9 et p8
lw   a1, 36(sp)   # params dam
jal  dam          # appelle dam
lw   a0,  0(sp)   # récupère
lw   a1,  4(sp)   # les
lw   a2,  8(sp)   # 8
lw   a3, 12(sp)   # registres
```

```
lw   a4, 16(sp)   # paramètres
lw   a5, 20(sp)   # de la
lw   a6, 24(sp)   # fonction
lw   a7, 28(sp)   # mad
add  a0, a0, a1   # r=p0+p1
add  a0, a0, a2   # r=r+p2
add  a0, a0, a3   # r=r+p3
add  a0, a0, a4   # r=r+p4
add  a0, a0, a5   # r=r+p5
add  a0, a0, a6   # r=r+p6
add  a0, a0, a7   # r=r+p7
lw   t0, 36(sp)   # p8
add  a0, a0, t0   # r=r+p8
lw   t0, 40(sp)   # p9
add  a0, a0, t0   # r=r+p9
lw   ra, 32(sp)   # @ ret
addi sp, sp, 9*4  # maj sp
ret               # back !
```

Exemples

Somme des *n* premiers entiers calculée récursivement :
doit conserver `n (a0)` pour l'addition au retour de l'appel

```
int32_t
s(int32_t n)
{
  return
    n > 0
    ? n + s(n-1)
    : 0;
}
```

```
s: addi  sp, sp, -8 # ra + place pour a0
   sw    ra, 4(sp)  # empile ra, adr retour
   sw    a0, 0(sp)  # sauve a0 pour retour
   bgtz  a0, 1f     # si > 0 va en 1f
   mv    a0, zero   # retourne 0
   j     2f         # sinon
1: addi  a0, a0, -1 # calcule n-1
   jal   s          # appel récursif
   lw    t0, 0(sp)  # récupère n avant appel
   add   a0, a0, t0 # fait somme
2: lw    ra, 4(sp)  # récupère adr retour
   addi  sp, sp, 8  # élimine pile
   ret              # retour appelante
```

Sommaire

ABI en résumé

Usage des registres

- imposé par le matériel (seul `x0 = zero` pour le RISC-V)
- et reposant sur des conventions suivies par les compilateurs

Appels de fonctions

- responsabilité partagée entre fonction appelante et fonction appelée
- structure prédéfinie et acceptée : prix à payer pour la compatibilité
- fonction appelante ne sait pas ce que fait fonction appelée, ne connaît que son prototype
- basée sur des conventions qu'il *faut absolument* respecter

Limitation

- cours suppose scalaires avec taille type 8, 16, ou 32 bits (y compris pointeurs)
- pas de structures, pas d'unions
- nombre d'arguments des fonctions fixes

⇒ Voir document complémentaire au cours pour traiter scalaires 64 bits et structures