Conventions d'appel des fonctions en assembleur RISCV

1 introduction

Les conventions d'appel de fonctions sont essentielles à l'utilisation d'un processeur, puisque tout programme doit pouvoir être appelé à travers sa fonction d'entrée (`main` en langage C), être capable d'appeler des fonctions de bibliothèques, et « raisonnablement » doit être structuré à l'aide de fonctions. Dans la terminologie consacrée, ces conventions font partie de l'*Application Binary Interface* ou *ABI*. Elles doivent être respectées à la lettre pour garantir le fonctionnement de vos propres programmes, qu'ils appellent du code C ou qi'ils soient appelés par lui.

Don't mess with the ABI !

2 Convention de nom pour les registres

Dans le processeur RISCV tous les registres « entiers » hormis le registre `x0` sont identiques du point de vue du matériel. Il faut donc définir des conventions particulières d'usage de ces registres pour leur utilisation dans les programmes. En substance, cela recouvre comment les fonctions sont appelées, et les relations avec le système d'exploitation. Pour refléter cet usage, en assembleur RISCV les registres possèdent un nom matériel et un nom logiciel.

La correspondance est donnée ci-dessous, les explications viennent ensuite.

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

Dans la suite de ce document, nous utiliserons les noms logiciel pour identifier les registres.

3 Conventions relatives aux fonctions

Nous prenons l'exemple du langage C, qui a le mérite d'être complètement explicite, pour la suite. En C, un prototype de fonction ressemble à ceci :

```
type_retour function(type_arg0 arg0, type_arg1 arg1, type_arg2 arg2);
```

Il est impératif d'établir des conventions qu'il faut suivre à la lettre pour implanter les fonctions et faire appels à des fonctions. En effet, lorsque l'on appelle une fonction, on ne connaît d'elle que son prototype, et non la manière dont elle est implantée. On sait juste qu'elle suit les conventions établies.

3.1 Types scalaires

On ne considère dans un premier temps que les types scalaires (entiers et pointeurs de divers types) pour les fonctions dont le nombre d'arguments est connu à la compilation, ce qui couvre une bonne partie des usages et permet de comprendre le principe dans un premier temps.

3.1.1 Valeur de retour

Si la taille, en bits, de `type_retour` est inférieure ou égale à 32, la valeur de retour de la fonction est disponible dans le registre `a0`. Si elle est comprise entre 33 et 64, elle est disponible dans la paire de registre `a1:a0`, poids forts dans `a1` et poids faibles dans `a0`. [1]

Si une fonction retourne un `char`, un `int`, ou un `int32_t`, alors la valeur de retour sera placée dans le registre `a0`. Si elle retourne un `uint64_t`, alors les poids forts seront dans le registre `a1` et les poids faible dans le registre `a0`.

3.1.2 Paramètres

Les paramètres sont passées dans les registres `a0` à `a7`, mais suivant une stratégie qui dépend de leurs tailles respectives. Les registres sont choisis dans l'ordre de 0 à 7, s'il n'y a plus de registres disponibles, les paramètres suivants sont placées dans la pile suivant une stratégie détaillée plus loin.

Si la taille, en bits, de $\texttt{type\_arg}_i$ est inférieure ou égale à 32, la valeur du paramètre est placée dans le prochain registre disponible.

Ainsi, si `arg0` est de type `int32_t`, il sera placé dans le registre `a0`, premier registre disponible. Si `arg1` est de type `char`, il sera placé dans le registre `a1`, prochain registre disponible après `a0`. Si `arg2` est de type `int`, il sera placé dans le registre `a2`, etc.

Si la taille, en bits, de $\texttt{type\_arg}_i$ est comprise entre 33 et 64, la valeur du paramètre est placée dans la prochaine paire de registres disponible. [2]

Ainsi, si `arg0` est de type `int32_t`, il sera placé dans le registre `a0`, premier registre disponible. Si `arg1` est de type `uint64_t`, il sera placé dans la paire de registres `a1:a2`, prochain registre disponible après `a0` (poids forts dans `a1`, poids faibles dans `a2`). Si `arg2` est de type `int`, il sera placé dans le registre `a3`, etc.

3.2 Appel de fonction et retour de fonction

Lors d'un appel à une fonction, qui se fait grâce aux instructions `jal` ou `jalr`, l'adresse de l'instruction qui suit immédiatement le saut est sauvée dans le registre `ra` pour pouvoir reprendre l'exécution lors du retour de fonction. Ces deux instructions existent sous forme de pseudo-instructions qui permettent de ne pas explicitement spécifier le registre dans lequel sauver l'adresse de retour.

Typiquement, le retour à la fonction appelante s'effectue avec l'instruction `jr ra`, qui peut être abrégée en la pseudo-instruction `ret`.

3.3 Gestion de la pile

L'exécution d'une fonction s'effectue dans un « contexte » spécifique, contenant en particulier ses variables locales. Ce contexte est crée lorsque le programme entre dans la fonction, et est détruit lorsque qu'il en sort. Ceci explique en particulier pourquoi on ne peut pas retourner l'adresse d'une variable locale, puisque le contexte de la fonction n'a pas de sens en dehors du moment ou la fonction est en train d'être exécutée.

1. Si la taille du scalaire retourné est supérieure à 64 bits, par ex. `__int128` dans `gcc`, alors on utilise la même stratégie que pour les agrégats, qui est décrite plus loin. Nous ne considérerons pas ces cas dans ce cours.
2. S'il ne reste qu'un registre disponible (forcément `a7`), alors les 32 bits de poids forts sont passés dans `a7` et les 32 bits de poids faibles sont passés par la pile. Si la taille du scalaire est supérieure à 64 bits, la note précédent s'applique également.

Le contexte est crée dans une pile, afin de permettre d'une part d'imbriquer des fonctions, ce qui nécessite de créer des contextes dans des contextes, et d'autre part d'appeler plusieurs fonctions successivement, ce qui permet de réutiliser la même zone de mémoire pour les contextes.

Une vue globale de l'agencement des données dans la pile (ou *stack layout*) est donnée ci-dessous, en supposant que la fonction `f` appelle la fonction `g` qui appelle la fonction `h` :

adresses hautes	paramètres `g` > 8	optionnel	← bas pile `f`
↓	adresse de retour	optionnel	
	registres à sauver	optionnel	
↑	variables locales	optionnel	
adresses basses	paramètres `h` > 8	optionnel	← bas pile `g`

La stratégie générale d'écriture d'une fonction consiste donc à établir un « prologue » dont l'objectif est de réserver la place nécessaire pour stocker l'adresse de retour, les registres de type `s` à sauver, les variables locales, et les paramètres des fonctions appelées qui possèdent plus de 8 paramètres. Cette réservation se fait simplement en déplaçant vers les adresses plus basses le pointeur de pile, registre `sp`. Symétriquement, un « épilogue » doit récupérer les valeurs sauvées dans les registres concernés, déplacer le pointeur de pile vers les adresses hautes de la même quantité que le prologue, et retourner à l'instruction qui suit l'appel, dont l'adresse est stockée dans `ra`.

En faisant l'hypothèse que les types des paramètres ont une taille inférieure ou égale à 32, une fonction ressemblera à ceci :

prologue :
calculer la valeur du déplacement de `sp`, déplacer `sp` de cette quantité, et sauver dans la pile ce qui peut l'être. C'est toujours faisable car cela ne dépend que du code à traduire ;
— soit n_v le nombre de variables locales et n_r le nombre de registres à préserver ;
— si la fonction appelle une ou plusieurs fonctions, on appelle $n_{p+} = \max(0, n_p - 8)$ le nombre maximum de paramètres au delà de 8 passés à ces fonctions ;
— on déplace `sp` : `sp` $\leftarrow$ `sp` $- (n_v + n_r + n_{p+}) \times 4$.
— on sauve les registres qui doivent l'être dans la zone idoine de la pile, *y compris ses propres arguments* si on veut les retrouver au retour.

code de la fonction :
si on appelle une fonction avec plus de 8 arguments, on doit mettre dans la pile les arguments en excès, le 9ème au bas de la pile, le 10ème immédiatement au dessus, etc[3] ;

épilogue :
— on restaure les registres qui doivent l'être à partir de la zone idoine de la pile ;
— on repositionne le pointeur de pile sur l'adresse qu'il avait avant l'appel : `sp` $\leftarrow$ `sp` $+ (n_v + n_r + n_{p+}) \times 4$.
— et on retourne à l'instruction qui suit l'appel d'où l'on vient.

3.3.1 Remarque sur les fonctions variadiques

Les fonctions à nombre variable d'arguments, dites « variadiques », possèdent une convention légèrement différente sur les paramètres (sinon, ça serait trop simple). En effet, les arguments de taille 64 bits doivent être passés dans des paires de registres dont le premier est pair (vous avez bien lu), et lorsqu'il n'y a plus de registres, sur la pile comme classiquement. Ceci peut donc mener à « sauter » un registre impair.

Imaginons l'appel suivant :
`fprintf(stderr, "%ll and %u\n", 0xdeaddeadbeef, 0xc0ffee);`.
Le paramètre `stderr` sera dans `a0`, la chaîne `"%ll and %u\n"` dans `a1`, `0xdead` dans `a2`, `0xdeadbeef` dans `a3` et `0xc0ffee` dans `a4`.

3. Cette phase est rare car peu de fonctions possèdent plus de 8 arguments, mais néanmoins possible. Par exemple la fonction X11 `XWMGeometry` possède 11 arguments (le record parmi les bibliothèques que j'ai parcourue).

Si maintenant c'est un appel à `printf` et non `fprintf`, c.-à-d. :
`printf("%ll and %u\n", 0xdeaddeadbeef, 0xc0ffee);`.
La chaîne `"%ll and %u\n"` sera dans `a0`, il n'y aura rien de significatif dans `a1`, `0xdead` sera dans `a2`, `0xdeadbeef` sera dans `a3` et `0xc0ffee` sera dans `a4`.

3.4 Agrégats

Ces considérations ne sont utiles que pour ceux qui veulent comprendre comment on peut généraliser ces conventions, et sont donc données à titre informatif. Nous n'aurons pas d'exemple de ce type à analyser ou écrire durant les exercices.

On considère maintenant les agrégats, *i.e.* `struct` en C. On ne parle pas ici de pointeurs vers un agrégat, qui sont des scalaires, mais bien de l'agrégat lui même, comme illustré ci-après :

```
struct point {
        uint16_t x, y;
};
struct rect {
        point ll, ur;
};
void draw(screen *s, struct rect r);
rect buid(uint16_t x0, uint16_t x1, int16_t y0, uint16_t y1);
```

3.4.1 Valeur de retour

Si la taille, en bits, de l'agrégat est inférieure ou égale à 32, la valeur de retour de la fonction est disponible dans le registre `a0`. Si elle est comprise entre 33 et 64, elle est disponible dans la paire de registre `a1:a0`, poids forts dans `a1` et poids faibles dans `a0`. Si elle est supérieure à 64, alors une zone pour l'agrégat est allouée par la fonction appelante dans sa pile, typiquement entre les variables locales et les arguments au delà de 8. Lors de l'appel, l'appelant va passer dans le registre `a0` un pointeur sur cette zone, ce qui de fait décale l'ensemble des paramètres des fonctions qui retournent un tel agrégat. Cette stratégie est parfaitement viable, car la zone est allouée dans la pile de l'appelante, et donc est encore valide après le retour de l'appelée.

Si une fonction retourne un `rect`, alors elle met à jour la zone pointée par `a0`. Ainsi `rect buid(uint16_t x0, uint16_t x1, int16_t y0, uint16_t y1)` aura un paramètre implicite `a0`, `a1` contiendra la valeur de `x0`, `a2` contiendra la valeur de `x1`, `a3` contiendra la valeur de `y0`, `a4` contiendra la valeur de `y1`.

Si elle retourne un `point`, alors `a0` contiendra `x` et `a1` contiendra `y`.

3.4.2 Paramètres

La stratégie pour les paramètres est identique à celle de la valeur de retour. Si l'agrégat contient plus de 2 mots de 32 bits, alors une zone est créée dans la pile de l'appelée, et l'adresse de la zone sera mise dans le registre correspondant au paramètre, ou dans la pile si le numéro du paramètre est au delà de 8.

Si nous étudions la fonction `void draw(screen *s, struct rect r)`, nous verrons que `a0` contient la valeur de `s`, en l'occurrence une adresse, et que `a1` contient également une adresse, qui elle pointe sur une zone mémoire placée dans la pile de l'appelant, et qui est initialisée avec les valeurs que contiennent les membres de la structure `r` au moment de l'appel.

3.5 Remarques

Les conventions sont présentées ici de manière quelque peu informelle, ce qui fait que pour certaines combinaisons, par ex. les agrégats dans les fonctions variadiques, il faut inférer ce que l'on doit faire. En toute logique, pour assurer qu'il n'y a aucune ambiguïté, il faudrait énumérer toutes les combinaisons de cas possibles et s'assurer que la stratégie pour chacune est définie et compatible avec toutes les autres.

4 Conventions relatives aux relations avec les librairies et le système d'exploitation

Elles sont fort simples dans les cas qui nous concernent, mais nécessitent néanmoins un nombre de registres spécifiques important.

4.1 Registres à usage général

Le registre `gp`, pointeur global, est positionné lors de l'édition de liens sur une zone permettant d'accéder aux données statiques ou globales définies dans le programme et les librairies. Ce registre est considéré comme une constante et ne doit pas être modifié.

Le registre `tp`, pointeur de tâche, est utilisé par les bibliothèques multi-tâches (genre *pthreads* que vous verrez en 2A) pour pointer sur le stockage local aux tâches. Ceci est forcément obscur puisque vous n'avez pas encore vu la notion de tâches ni de parallélisme. Tout comme le précédent, ce registre est considéré comme une constante et ne doit pas être modifié.

4.2 Registres spécifiques à la gestion des interruptions

Les registres :

— `mtvec` contient l'adresse à laquelle le `pc` est positionné en cas d'interruption ou d'exception.
— `mepc` contient l'adresse de l'instruction (le `pc`) qui était en cours d'exécution lors de l'occurence d'une interruption ou exception.
— `mcause` contient la cause d'un départ en interruption ou exception.
— `mstatus` contient un masque permettant globalement d'être sensible ou non aux interruptions.
— `mie` contient un masque permettant d'être sensible ou non à une ou des interruptions spécifiques.
— `mip` contient les interruptions actuellement en attente de traitement.

Ces registres ne sont pas accessibles avec les instructions que nous avons vues jusqu'à présent. Il faut utiliser `csrr xi, mj` pour lire dans un registre normal `xi` le registre spécial `mj`, et `csrw mj, xi` pour effectuer l'opération inverse.