Un peu de langage d'assemblage du RISC-V sur Linux

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 9 juillet 2023

https://www.bortzmeyer.org/risc-v-assembly.html

Un peu de distraction sans objectif utilitaire à court terme : programmons en langage d'assemblage (ce qui est souvent appelé par abus de langage « assembleur ») sur un processeur RISC-V, dans un environnement Linux.

Cet article vise un public qui sait programmer mais sans avoir fait (ou très peu) de langage d'assemblage. Car oui, un problème terminologique fréquent est de parler d'« assembleur » alors que, en toute rigueur, ce terme désigne le programme qui, comme un compilateur, va traduire le langage d'assemblage en code machine. Ce langage d'assemblage est un langage de bas niveau, ce qui veut dire qu'il est proche de la machine. C'est d'ailleurs un de ses intérêts : regarder un peu comment ça fonctionne sous le capot. On lit parfois que ce langage est tellement proche de la machine qu'il y a une correspondance simple entre une instruction en langage d'assemblage et une instruction de la machine mais ce n'est plus vrai depuis longtemps. Comme un compilateur, l'assembleur ne va pas traduire mot à mot le programme. Il a, par exemple, des « pseudo-instructions » qui vont être assemblées en une ou plusieurs instructions. L'informatique, aujourd'hui, est formée de nombreuses couches d'abstraction empilées les unes sur les autres et on ne peut pas vraiment dire qu'en programmant en langage d'assemblage, on touche directement à la machine. (D'ailleurs, certains processeurs contiennent beaucoup de logiciel, le microcode.)

Pourquoi programmer en langage d'assemblage alors que c'est bien plus facile en, par exemple, C (ou Rust, ou Go, comme vous voulez) ? Cela peut être pour le plaisir d'apprendre quelque chose de nouveau et de très différent, ou cela peut être pour mieux comprendre l'informatique. Par contre, l'argument « parce que cela produit des programmes plus rapides » (ou, version 2023, parce que ça économise des ressources et que l'ADEME sera contente de cet effort pour économiser l'énergie), cet argument est contestable ; outre le risque plus élevé de bogues, le programme en langage d'assemblage ne sera pas automatiquement plus rapide, cela dépendra des compétences du programmeur ou de la programmeuse, et programmer dans ce langage est **difficile**. (Le même argument peut d'ailleurs s'appliquer si on propose de recoder un programme Python en C « pour qu'il soit plus rapide ».)

Chaque famille de processeurs (x86, ARM, etc) a son propre code machine (qu'on appelle l'ISA pour "*Instruction Set Architecture*") et donc son propre langage d'assemblage (une des motivations de C était

de fournir un langage d'assez bas niveau mais quand même portable entre processeurs). Je vais utiliser ici un processeur de la famille RISC-V.

Si on veut programmer avec le langage de cette famille, on peut bien sûr utiliser un émulateur comme QEMU, mais c'est plus réaliste avec un vrai processeur RISC-V. Je vais utiliser celui de ma carte Star 64 <https://www.bortzmeyer.org/star64-first-boot.html>. Un noyau Linux tourne sur celle-ci et on verra qu'on sous-traitera certaines tâches à Linux. Commençons par un programme trivial, éditons `trivial.s` :

```
addi    x10, x0, 42
```

L'instruction `addi` ("*Add Immediate*" car son dernier argument est un littéral, "*immediate data*") fait l'addition, la destination est le registre `x10`. On y met le résultat de l'addition du registre `x0` (qui contient toujours zéro, un peu comme le pseudo-fichier `/dev/zero`) et du nombre 42. La liste complète des instructions que connait le processeur RISC-V figure dans le "*The RISC-V Instruction Set Manual*" <https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications> mais ce texte est difficile à lire, étant plutôt orienté vers les gens qui conçoivent les processeurs ou qui programment les assembleurs. (Et il ne contient pas les pseudo-instructions.) Je me sers plutôt de documents de synthèse comme celui de metheis <https://mark.theis.site/riscv/> ou bien la "*RISC-V Instruction-Set Cheatsheet*" <https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8>.

On va assembler notre petit programme, avec l'assembleur `as` :

```
% as -o trivial.o trivial.s
```

Puis on fabrique un exécutable avec le relieur :

```
% ld -o trivial trivial.o
ld: warning: cannot find entry symbol _start; defaulting to 00000000000100b0
```

Et l'avertissement ? Contrairement aux langages de plus haut niveau, qui s'occupent des détails comme de démarrer et d'arrêter le programme, le langage d'assemblage vous laisse maitre et, ici, je n'ai pas défini l'endroit où démarrer. Bon, ce n'est qu'un avertissement, exécutons le programme :

```
% ./trivial
Illegal instruction (core dumped)
```

Ajoutons explicitement un point d'entrée pour traiter l'avertissement obtenu :

```
.global _start

_start:
        addi    x10, x0, 2
```

On n'a plus l'avertissement, mais le programme plante de la même façon. C'est en fait à la fin du programme qu'il y a un problème : le programme n'a pas d'instruction d'arrêt, l'assembleur n'en a pas ajouté (contrairement à ce qu'aurait fait un compilateur C), et l'« instruction » suivant `addi` est en effet illégale (c'est la fin du code). On va donc ajouter du code pour finir proprement, en utilisant un appel système Linux :

```
_start:
        addi    x10, x0, 2

        # Terminons proprement (ceci est un commentaire)
        li    x10, 0
        li    x17, 93   # 93 = exit
        ecall
```

Cette fois, tout se passe bien, le programme s'exécute (il n'affiche rien mais c'est normal). Il faut maintenant expliquer ce qu'on a fait pour que ça marche.

On veut utiliser l'appel système `exit`. Sur une machine Linux, `man 2 exit` vous donnera sa documentation. Celle-ci nous dit qu'`exit` prend un argument, le code de retour. D'autre part, les conventions de passage d'arguments de RISC-V nous disent qu'on met l'argument dans le registre `x10`, et, lorsqu'on fait un appel système, le numéro de cet appel dans `x17`. Ici, 93 est `exit` (si vous voulez savoir où je l'ai trouvé, `grep _exit /usr/include/asm-generic/unistd.h`). `li` (*"Load Immediate"*) va écrire 0 (le code de retour) dans `x10` puis 93 dans `x17`. Enfin, `ecall` va effectuer l'appel système demandé, et on sort proprement. (Vous noterez que `li x10, 0` est parfaitement équivalent à `addi x10, x0, 0`, et l'assembleur va produire exactement le même code dans les deux cas.) Pour davantage de détails sur l'utilisation des appels système Linux, voir *"Linux System Calls for RISC-V with GNU GCC/Spike/pk"* <https://github.com/scotws/RISC-V-tests/blob/master/docs/riscv_linux_system_calls.md>.

Bon, ce programme n'était pas très passionnant, il additionne 0 + 2 et met le résultat dans un registre mais n'en fait rien ensuite. On va donc passer à un programme plus démonstratif, un *"Hello, world"*. Le processeur n'a pas d'instructions pour les entrées-sorties, on va sous-traiter le travail au système d'exploitation, via l'appel système Linux `write`, qui a le numéro 64 (rappelez-vous, il faut regarder dans `/usr/include/asm-generic/unistd.h`) :

```
.global _start

_start:
        la    x11, helloworld

        # Print
        addi  x10, x0, 1 # 1 = standard output
        addi  x12, x0, 13 # 13 bytes
        addi  x17, x0, 64
        ecall

        # Exit
        addi    x10, x0, 0
        addi    x17, x0, 93
        ecall

.data
helloworld:      .ascii "Hello World!\n"
```

Qu'a-t-on fait ? On a réservé une chaine de caractères dans les données, nommée `helloworld`. On charge son adresse (car `man 2 write` nous a dit que les arguments étaient le descripteur de fichier de la sortie, mis dans `x10`, une adresse et la longueur de la chaine) dans le registre `x11`, la taille à écrire dans `x12` et on appelle `write`. Pour compiler, on va arrêter de se servir de l'assembleur et du relieur directement, on va utiliser gcc qui les appelle comme il faut :

```
% gcc  -nostdlib -static -o hello-world hello-world.s

% ./hello-world
Hello World!
```

C'est parfait, tout marche (normal, je ne l'ai pas écrit moi-même <https://smist08.wordpress.com/2019/09/07/risc-v-assembly-language-hello-world/>). Le `-nostdlib` était pour indiquer à gcc qu'on n'utiliserait pas la bibliothèque C standard, on veut tout faire nous-même (ou presque, puisqu'on utilise les appels système du noyau Linux ; si on écrivait un noyau, sans pouvoir compter sur ces appels système, la tâche serait bien plus difficile).

Si vous lisez du code en langage d'assemblage écrit par d'autres, vous trouverez peut-être d'autres conventions d'écriture. Par exemple, le registre `x10` peut aussi s'écrire `a0`, ce surnom rappelant son rôle pour passer des arguments (alors que `x` préfixe les registres indépendamment de leur fonction). De même, des mnémoniques comme `zero` peuvent être utilisés, ce dernier désignant `x0` (registre qui, rappelons-le, contient toujours zéro). La documentation de ces écritures possibles est la « *"RISC-V ABIs Specification"* <https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications> ». On voit ces variantes dans l'écriture si on demande au désassembleur de désassembler le code trivial :

```
$ objdump -d trivial.o
...
Disassembly of section .text:

0000000000000000 <_start>:
   0:   00200513                li      a0,2
```

Le `addi x10, x0, 2` est rendu par l'équivalent `li a0, 2`.

Faisons maintenant quelque chose d'un tout petit peu plus utile. On va décrémenter un nombre jusqu'à atteindre une certaine valeur. Mais attention : afficher un nombre (pour suivre les progrès de la boucle) n'est pas trivial. `write` ne sait pas faire, il sait juste envoyer vers la sortie les octets. Ça marche bien avec des caractères ASCII comme plus haut, mais pas avec des entiers. On va donc tricher en utilisant des nombres qui correspondent à des caractères ASCII imprimables. Voici le programme :

```
.equ SYS_WRITE, 64
.equ SYS_EXIT, 94
.equ STDOUT, 1
.equ SUCCESS, 0

.global _start

_start:

        # The decrement
        li t2, 1
        # The final value (A)
```

```
        li t1, 65
        # The first value (Z) then the current value
        li t0, 90

loop:
        # Store on the stack
        sb  t0, 0(sp)

        # Print the current value (it is copied on the stack)
        li a0, STDOUT
        add a1, zero, sp
        addi  a2, x0, 1        # 1 byte (parameter count of write() )
        addi  a7, x0, SYS_WRITE
        ecall

        # Decrement
        sub t0, t0, t2

        # End of loop?
        ble t1, t0, loop # BLE : Branch If Lower Or Equal

        # Print the end-of-line
        li a0, STDOUT
        la    a1, eol
        addi  a2, x0, 1
        addi  a7, x0, SYS_WRITE
        ecall

        # Exit
        li    a0, SUCCESS
        addi    a7, x0, SYS_EXIT
        ecall

.data
eol:      .ascii "\n"
```

C'était l'occasion de voir :

— Une nouvelle instruction, BLE ("*Branch if Lower or Equal*") qui teste ses arguments et saute à une étiquette (ici, `loop`) selon le résultat de la comparaison. Grâce à cette instruction, on peut écrire toutes les structures de contrôle (ici, une boucle).
— Une autre instruction, SB ("*Store Byte*") pour écrire dans la mémoire (et non plus dans un registre du processeur).
— Le pointeur de pile, SP ("*Stack Pointer*"), alias du registre `x2`.
— Les directives à l'assembleur comme `.equ`, qui affecte un nom plus lisible à une valeur. La liste de ces directives (on avait déjà vu `.global`) est disponible <`https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md#copyright-and-license-information`>.

Nous avions besoin d'écrire en mémoire car `write` prend comme deuxième argument une adresse. Comme nous n'avons pas encore alloué de mémoire, utiliser la pile est le plus simple. (Sinon, il aurait fallu réserver de la mémoire dans le programme, ou bien utiliser les appels système `brk` ou `mmap` après son lancement.)

Si on veut formater un nombre pour l'imprimer, c'est plus compliqué, donc je copie le code écrit par code4eva <`https://stackoverflow.com/users/7037839/code4eva`> sur StackOverflow <`https://stackoverflow.com/a/67047319/15625`>. Voyons cela avec une fonction qui trouve le plus grand de deux nombres (mis respectivement dans les registres `a0` et `a1`, le résultat, toujours en suivant les conventions d'appel, sera dans `a0`) :

```
_start:
    # Les deux nombres à comparer
    li a0, -1042
```

```
    li a1, 666
    call max
    # On sauvegarde le résultat
    add s0, zero, a0
    ...

max:
blt a0, a1, smallerfirst
ret

smallerfirst:
add a0, zero, a1
ret
```

L'instruction `call` appelle la fonction `max`, qui prendra ses arguments dans `a0` et `a1`. La pseudo-instruction `ret` (”*Return*”, équivalente à `jalr x0, x1, 0`) fait revenir de la fonction, à l'adresse sauvegardée par le `call`. Notez que `call` est en fait une pseudo-instruction (vous en avez la liste <`https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md`>), que l'assembleur traduira en plusieurs instructions du processeur. C'est ce qui explique que vous ne trouverez pas `call` dans la spécification du jeu d'instructions RISC-V. Le programme entier, avec affichage du résultat, est dans .

Souvent on n'écrit pas l'entièreté du programme en langage d'assemblage, mais seulement les parties les plus critiques, du point de vue des performances. La ou les fonctions écrites en langage d'assemblage seront donc appelées par un programme en C ou en Go. D'où l'importance de conventions partagées sur l'appel de fonctions, l'ABI (”*Application Binary Interface*”) qui permettront que les parties en C et celles en langage d'assemblage ne se marcheront pas sur les pieds (par exemple n'écriront pas dans des registres censés être stables), et qui garantiront qu'on pourra lier des codes compilés avec des outils différents. Voici un exemple où le programme principal, qui gère l'analyse et l'affichage des nombres, est en C et le calcul en langage d'assemblage. Il s'agit de la détermination du PGCD par l'algorithme d'Euclide, qui est trivial à traduire en assembleur. Les sources sont et :

```
% gcc -Wall -o pgcd call-pgcd.c pgcd.s

%./pgcd  999 81
pgcd(999, 81) = 27
```

En regardant le source en langage d'assemblage, vous verrez que le respect des conventions (les deux arguments dans `a0` et `a1`, le résultat retourné dans `a0`) fait que, l'ABI étant respectée, les deux parties, en C et en langage d'assemblage, n'ont pas eu de mal à communiquer.

Autre exemple de programme, on compte cette fois de 100 à 1, en appelant la fonction `num_print` pour afficher le nombre. Attention à bien garder ses variables importantes dans des registres sauvegardés (mnémonique commençant par s) sinon la fonction `num_print` va les modifier. Le programme est .

Un dernier truc : le compilateur C peut aussi produire du code en langage d'assemblage, ce qui peut être utile pour apprendre. Ainsi, ce petit programme C :

```
void main() {
  int x = 1;
  x = x + 3;
}
```

Une fois compilé avec `gcc -S minimum.c`, il va donner un fichier `minimum.s` avec, entre autres instructions de gestion du démarrage et de l'arrêt du programme :

```
li a5,1
sw a5,-20(s0)
lw a5,-20(s0)
addiw a5,a5,3
sw a5,-20(s0)
```

On y reconnait l'initialisation de `x` à 1 (`x` est stocké dans le registre `a5` avec un `li` - "*Load Immediate*", charger un littéral, son écriture en mémoire vingt octets sous `s0` (le registre `s0` a été initialisé plus tôt, pointant vers une zone de la pile) avec `sw` ("*Store Word*"), puis son addition avec 3 par `addiw` ("*Add Immediate Word*") et son retour en mémoire avec un `sw`. (L'instruction `lw` me semble inutile puisque `a5` contenait déjà la valeur de `x`.)

Quelques documents et références pour finir :

— Une bonne introduction à la question <https://medium.com/swlh/risc-v-assembly-for-beginners-38
et, du même auteur, des exemples <https://erik-engheim.medium.com/risc-v-assembly-code-exampl
— Très pédagogique, un site qui va vous faire mal aux yeux <https://www.chibialiens.com/riscv/>.
— Les spécifications officielles de RISC-V <https://riscv.org/technical/specifications/> (pas forcément une lecture facile).
— Le programmeur utilisant un assembleur lira avec profit "*RISC-V Assembly Programmer's Manual*" <https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md>.
— Les conventions d'appel (l'ABI, donc), sont dans "*RISC-V ABIs Specification*" <https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications>.
— Pour la liste des directives à l'assembleur (comme `.global`) et pour celle des pseudo-instructions, je recommande le "*RISC-V Assembly Programmer's Manual*" <https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md#copyright-and-license-information>.