Aide assembleur RISC-V

Malo MONIN

Table des matières

Terminologie

On rappelle les trois principales tailles de valeurs dans le tableau suivant.

Nom français	Nom anglais	Taille
Octet	*Byte*	8 bits
Demi-mot	*Half-word*	16 bits
Mot	*Word*	32 bits

Tableau 1. – Tailles de valeurs.

On note `PC` le *program counter* : c'est le registre qui contient l'adresse de la prochaine instruction à exécuter.

Instructions

Opérations arithmétiques et logiques

Les opérations arithmétiques et logiques sont toujours effectuées entre deux registres `rs1` et `rs2`, ou entre un registre `rs1` et une constante immédiate sur 12 bits `imm12`. Le résultat est stocké dans un registre `rd`.

Syntaxe

Les instructions effectuant des opérations entre deux registres acceptent les argument `rd, rs1, rs2`. Celles effectuant des opérations entre un registre et une constante immédiate se terminent par `i` et acceptent les arguments `rd, rs1, imm12`.

Attention. Toutes les instructions effectuant des opérations entre deux registres n'ont pas un équivalent effectuant l'opération avec une constante immédiate. Par exemple, l'instruction `subi` n'existe pas.

Sémantique

Les instructions effectuant des opérations arithmétiques et logiques sont résumées dans les tableaux ci-dessous. Toutes les opérations logiques s'effectuent bit à bit.

Instruction	Valeur stockée dans `rd`
`add`	$\texttt{rs1} + \texttt{rs2}$
`sub`	$\texttt{rs1} - \texttt{rs2}$
`mul`	$\texttt{rs1} \times \texttt{rs2}$
`and`	$\texttt{rs1} \,\&\, \texttt{rs2}$
`or`	$\texttt{rs1} \mid \texttt{rs2}$
`xor`	$\texttt{rs1} \mathbin{\hat{}} \texttt{rs2}$

Tableau 2. – Opération effectuée par chaque instruction arithmétique ou logique impliquant deux registres.

Instruction	Valeur stockée dans `rd`
`addi`	$\texttt{rs1} + \mathrm{sext}(\texttt{imm12})$
`andi`	$\texttt{rs1} \,\&\, \mathrm{sext}(\texttt{imm12})$
`ori`	$\texttt{rs1} \mid \mathrm{sext}(\texttt{imm12})$
`xori`	$\texttt{rs1} \mathbin{\hat{}} \mathrm{sext}(\texttt{imm12})$

Tableau 3. – Opération effectuée par chaque instruction arithmétique ou logique impliquant un registre et une constante immédiate.

Branchements conditionnels

Les instructions permettant de réaliser des branchements conditionnels commencent toutes par `b`.

Syntaxe

Les arguments sont `rs1, rs2, imm12`. Alternativement, il est possible de spécifier un label à la place de `imm12`.

Sémantique

Si la condition est vérifiée, alors $\text{sext}(\texttt{imm12})$ est ajouté à la valeur de `PC`. Sinon, `PC` est incrémenté comme à la normale.

Les instructions de branchement conditionnel sont résumées dans le tableau ci-dessous.

Instruction	Signification	Condition
`beq`	*Branch if equal*	$\texttt{rs1} = \texttt{rs2}$
`bne`	*Branch if not equal*	$\texttt{rs1} \neq \texttt{rs2}$
`blt`	*Branch if less than*	$\texttt{rs1} < \texttt{rs2}$
`bge`	*Branch if greater than or equal*	$\texttt{rs1} \geq \texttt{rs2}$

Tableau 4. – Condition testée par chaque instruction de branchement conditionnel.

Interactions avec la mémoire

Les deux manières d'interagir avec la mémoire sont la lecture et l'écriture.

Syntaxe

Les arguments attendus par les instructions de lecture sont `rd, imm12(rs1)`. Les arguments attendus par les instructions d'écriture sont `rs2, imm12(rs1)`.

Sémantique

Les instructions d'interaction avec la mémoire interragissent systématiquement avec l'emplacement mémoire commençant à l'adresse `rs1` + sext(`imm12`).

Les opérations de lecture lisent la valeur stockée dans l'emplacement mémoire en la copiant dans `rd`, tandis que les instructions d'écriture écrivent la valeur de `rs2` dans l'emplacement mémoire.

Les instructions impliquant des valeurs plus petites que des mots présentent également des versions non-signées, n'effectuant pas d'extension de signe.

Taille	**Instruction de lecture (« *load* »)**		**Instruction d'écriture (« *store* »)**	
	Signée	**Non-signée (« *unsigned* »)**	**Signée**	**Non-signée (« *unsigned* »)**
Byte	`lb`	`lbu`	`sb`	`sbu`
Half-word	`lh`	`lhu`	`sh`	`shu`
Word	`lw`		`sw`	

Tableau 5. – Instructions d'interaction avec la mémoire.

Pseudo-instructions utiles

Les pseudo-instructions sont des instructions qui existent dans l'assembleur, mais qui sont en fait traduites en d'autres instructions lors de l'assemblage. On peut les utiliser de la même manière que des instructions normales.

Syntaxe	**Signification**	**Description**
`nop`	*No operation*	N'effectue aucune opération visible, mais fait avancer `PC`.
`li rd, imm`	*Load immediate*	Écrit la valeur `imm` dans `rd`.
`la rd, label`	*Load address*	Écrit l'adresse du label `label` dans `rd`.
`mv rd, rs1`	*Move*	Copie la valeur de `rs1` dans `rd`.
`b imm`	*Branch*	Ajoute la valeur `imm` à `PC`.
`j imm`	*Jump*	Écrit la valeur `imm` dans `PC`.
`call label`	*Call*	Appelle la fonction débutant en `label` (écrase la valeur de `ra`).
`ret`	*Return*	Rend la main à la fonction appelante.

Tableau 6. – Quelques pseudo-instructions assembleur RISC-V.

Récapitulatif

Instruction	Arguments	Signification
Opérations arithmétiques et logiques		
addi	rd, rs1, imm12	rd $\leftarrow$ rs1 $+$ sext(imm12)
andi		rd $\leftarrow$ rs1 & sext(imm12)
ori		rd $\leftarrow$ rs1 $\mid$ sext(imm12)
xori		rd $\leftarrow$ rs1 $\hat{}$ sext(imm12)
add	rd, rs1, rs2	rd $\leftarrow$ rs1 $+$ rs2
sub		rd $\leftarrow$ rs1 $-$ rs2
mul		rd $\leftarrow$ rs1 $\times$ rs2
and		rd $\leftarrow$ rs1 & rs2
or		rd $\leftarrow$ rs1 $\mid$ rs2
xor		rd $\leftarrow$ rs1 $\hat{}$ rs2
Branchements conditionnels		
beq	rs1, rs2, imm12	Si rs1 $=$ rs2, alors PC est décalé de sext(imm12). Sinon, on passe à l'instruction suivante comme d'habitude.
bne		Idem, avec rs1 $\neq$ rs2.
blt		Idem, avec rs1 $<$ rs2.
bge		Idem, avec rs1 $\geq$ rs2.
Interaction avec la mémoire		
lb	rd, imm12(rs1)	Charge une valeur dans rd depuis l'emplacement mémoire commençant à l'adresse rs1 $+$ sext(imm12).
lh		
lw		
sb	rs1, imm12(rs2)	Stocke une valeur dans l'emplacement mémoire commençant à l'adresse rs2 $+$ sext(imm12) depuis rs1.
sh		
sw		
Pseudo-instructions utiles		
nop	Aucun	N'effectue aucune opération visible, mais fait avancer PC.
li	rd, imm	Écrit la valeur imm dans rd.
la	rd, label	Écrit l'adresse du label label dans rd.
mv	rd, rs1	Copie la valeur de rs1 dans rd.
b	imm	Ajoute la valeur imm à PC.
j	imm	Écrit la valeur imm dans PC.
call	label	Appelle la fonction débutant en label (écrase la valeur de ra).
ret	Aucun	Rend la main à la fonction appelante.

Tableau 7. – Instructions et pseudo-instructions assembleur RISC-V.

Syntaxe

Constantes

Il est possible de déclarer une constante (dont l'identifiant sera essentiellement remplacé par sa valeur lors de l'assemblage, similairement à un `#define` en C) avec la syntaxe `.eqv <identifiant> <valeur>`. Par exemple, `.eqv N 8`.

Constantes, mais différent

Il est possible de déclarer une valeur qui sera stockée dans la mémoire statique (je crois) avec la syntaxe `.byte <valeur>`, `.half <valeur>` ou `.word <valeur>`, en fonction de la taille de la valeur. Il est possible de stocker plusieurs valeurs à la suite dans la mémoire en spécifiant plusieurs valeurs séparées par des espaces. On peut utiliser un label pour donner un nom à une valeur ainsi déclarée : `TABLE: .word 1 2 3 4`.

Registres

L'architecture RISC-V contient 32 registres (apparamment il y en a aussi 32 autres nommés `f{0..31}` mais ils sont chelous). Chacun de ces registres à un nom usuel (« *ABI Name* ») qui correspond à son utilisation en pratique.

Lors d'un appel de fonction, on ne peut pas savoir quels registres la fonction appelée va modifier. Il faut donc définir des convention de sauvegarde des registres : certains registres requièrent d'être restauré à leur état au début de l'appel à la fin d'une fonction (le sauveur est alors la fonction appelée, car la fonction appelante n'a pas besoin de se préoccuper de ces registres lors d'un appel), tandis que d'autres peuvent être utilisés sans avoir à en restaurer le contenu (le sauveur est donc la fonction appelante, qui ne peut pas supposer que l'état de ces registres sera conservé après un appel).

Les registres utilisés dans le cadre du cours d'architecture de L3 sont présentés dans le tableau suivant.

ABI name(s)	**Description**	**Sauveuse**
`zero`	Vaut toujours zéro. Ne supporte pas l'écriture.	Non-applicable.
`ra`	Adresse de retour. Modifiée par `call`, lue par `ret`.	Appelante.
`sp`	Pointeur de pile (« *stack pointer* »). Pointe vers l'emplacement mémoire au dessus de la pile.	Appelée.
`a0`, `a1`	Arguments et valeurs de retour.	Appelante.
`a`i, $i \in [\![2, 7]\!]$	Arguments.	Appelante.
`t`i, $i \in [\![0, 6]\!]$	Valeurs temporaires.	Appelante.
`s`i, $i \in [\![0, 11]\!]$	Registres sauvegardés.	Appelée.

Tableau 8. – Registres dans l'architecture RISC-V.

Autres ressources

Plus d'instructions sont listées sur https://risc-v.guru/instructions/.

Des descriptions plus précises des instructions sont données sur https://lhtin.github.io/01world/app/riscv-isa/?xlen=32.

Plus de registrers et d'informations sur les conventions d'appel à https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf.

La spécification de RISC-V est disponible à https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf.