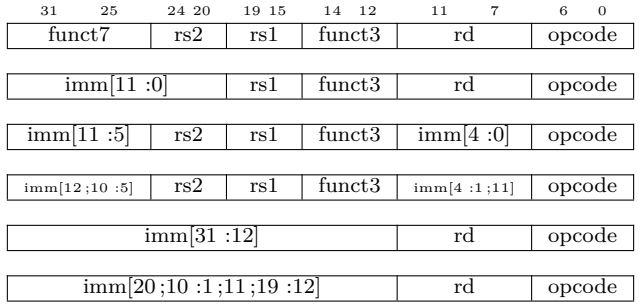
Éléments de langage d'assemblage RISCV

Jeu d'instructions

Le RISCV est un jeu d'instruction de type RISC. Il possède des instructions codées sur 32 bits réparties dans 6 formats différents :



Ci-contre un tableau récapitulatif des principales instructions, la mnémonique associée ainsi qu'un descriptif de l'opération effectuée dans la partie opérative du processeur.

Mnémonique	Format	Opcode/Func3/Func7	Opération	Nom complet
addi rd, rs1, imm	I	0x13/0	R[rd]=R[rs1]+SignExtImm[1]	Addition avec immédiat
add rd, rs1, rs2	R	0x33/0/0	R[rd]=R[rs1]+R[rs2]	Addition non signée entre registres
and rd, rs1, rs2	R	0x33/7/0	R[rd]=R[rs1] & R[rs2]	ET bit à bit entre registres
andi rd, rs1, imm	I	0x13/7	R[rd]=R[rs1] & SignExtImm[1]	ET bit à bit avec immédiat
auipc rd, imm	U	0x17	R[rd]=PC+$IR_{31..12} \| 0^{12}$	Addition PC avec immédiat
beq rs1, rs2, label	B	0x63/0	if(R[rs1]== R[rs2]) PC=PC+BranchAddr[2]	Branchement si égalité
bne rs1, rs2, label	B	0x63/1	if(R[rs1] $\neq$ R[rs2]) PC=PC+BranchAddr[2]	Branchement si inégalité
jal rd, label	J	0x6f	R[rd]=PC+4 ;PC = JumpAddr[3]	Saut inconditionnel avec lien
jalr rd, imm(rs1)	I	0x67/0	R[rd]=PC+4 ;PC = R[rs1]+imm	Saut inconditionnel registre avec lien
lui rd, imm	U	0x37	R[rd] = $IR_{31..12} \| 0^{12}$	Chargement immédiat haut
lw rd, imm(rs1)	I	0x03/2	R[rd] = mem[R[rs1]+SignExtImm[1]]	Chargement registre
or rd, rs1, rs2	R	0x33/6/0	R[rd] = R[rs1] | R[rs2]	OU bit à bit registre à registre
ori rd, rs1, imm	I	0x13/6	R[rd] = R[rs1] | SignExtImm[1]	OU bit à bit avec immédiat
slt rd, rs, rt	R	0x33/2/0	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	Set Less Than registre à registre
slti rd, rs1, imm	I	0x13/2	R[rd] = (R[rs1] < SignExtImm [1]) ? 1 : 0	Set Less Than avec immédiat
slli rd, rs1, sh	I	0x13/1	R[rd] = R[rs1] << SH(=$IR_{24..20}$)	Décalage à gauche
srli rd, rs1, sh	I	0x13/5	R[rd] = R[rs1] >> SH(=$IR_{24..20}$)	Décalage logique à droite
sub rd, rs1, rs2	R	0x33/0/0x20	R[rd]=R[rs1]−R[rs2]	Soustraction
sw rs2, imm(rs1)	S	0x23/2	mem[R[rs1]+StoreAddr[4]] = R[rs2]	Stockage registre
xori rd, rs1, imm	I	0x13/4	R[rd] = R[rs1] XOR SignExtImm[1]	OU exclusif bit à bit avec immédiat

1. $SignExtImm = IR_{31}^{20} \| IR_{31..20}$
2. $BranchAddr = IR_{31}^{20} \| IR_7 \| IR_{30..25} \| IR_{11..8} \| 0$
3. $JumpAddr = IR_{31}^{12} \| IR_{19..12} \| IR_{20} \| IR_{30..25} \| 0$
4. $StoreAddr = IR_{31}^{20} \| IR_{31..25} \| IR_{11..7}$

j label ≡ jal rd, label avec rd=0
jr rs1 ≡ jalr rd, imm(rs1) avec rd=0 et imm=0
ret ≡ jr ra
lb/lh, chargement octet/demi-mot
sb/sh, sauvegarde octet/demi-mot

Autres branchements :
bge, bgeu, blt, bltu
Comparaisons non-signées :
sltu, sltiu

Pseudo-instructions

Les pseudo-instructions facilitent l'écriture des programmes.

Pseudo-instruction	Opération	Expansion possible
li rd, cste	R[rd]=cste	lui rd, %hi(cste) ori rd, rd, %lo(cste)
la rd, symb	R[rd]=adresse de symb	auipc rd, %hi(symb) addi rd rd, %lo(symb)
lw rd, symb	R[rd]=MEM[symb]	auipc rd, %hi(symb) lw rd, %lo(symb)(rd)
⇝ pareil pour lb, lbu, lh, lhu		
sw rs, symb, rt	MEM[symb]=R[rs]	auipc rt, %hi(symb) sw rs, %lo(symb)(rt)
⇝ pareil pour sb, sh		
mv rd, rs	R[rd]=R[rs]	addi rd, rs, 0
nop	ne fait rien	addi x0, x0, 0
not rd, rs	R[rd]=not R[rs]	xori rd, rs, -1
seqz rd, rs	R[rd]=1 if rs=0 else 0	sltu rd, x0, rs
⇝ similaire pour snez, sltz, sgtz		
beqz rs, label	saut si rs = 0	beq rs, zero, label
⇝ similaire pour bnez, blez, bgez, bltz, bgtz		
bgt rs, rt, label	saut si rs > rt	blt rt, rs, label
⇝ similaire pour ble, bgtu, bleu		
j label	saut inconditionnel	jal zero, label
jr rs	saut à l'adresse dans rs	jalr zero, 0(rs)
jal label	saut + maj de ra	jal ra, label
call label	saut + maj de ra	jal ra, label
ret	retour de fonction	jalr zero, 0(ra)

Syntaxe assembleur

Quelques éléments de syntaxe :

1. Un nom de fichier est suffixé par ≪ `.s` ≫.
2. Les commentaires démarrent par un `#` ou un `;` et s'achèvent en fin de ligne. Une bonne pratique consiste à commenter les instructions en fin d'instruction pour faciliter la compréhension du code.

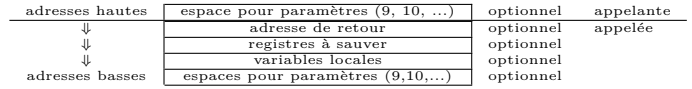
```
# Ceci est une fonction qui fait des choses
 sw ...      ; sauve la valeur copiée dans la mémoire
```

3. Un entier hexadécimal est préfixé par `0x` et un entier décimal est égal à lui-même ($\texttt{250}_{10} = \texttt{0xFA}_{16}$).
4. Une chaîne de caractères est entourée de guillemets et peut contenir des caractères d'échappements. Par exemple `"Chaîne avec retour à la ligne\n"`.
5. Un label est suffixé par `:` et correspond soit à une adresse de variable, soit à une adresse de saut.
6. Un immédiat est une opérande contenue dans une instruction. Cette constante est toujours étendue de signe sur 32 bits, sa taille initiale varie en fonction du type d'instruction (I, S, B, U ou J).
7. Une ligne correspondant à une instruction débute par l'instruction suivie de ses arguments. Ceux-ci sont séparés par des virgules (`add t0, t1, t2`).
8. Les registres sont au nombre de 32 et leur utilisation dépend des conventions suivantes :

Nom matériel	Nom logiciel	Signification	Préservé ?
x0	zero	Zéro	Oui(0)
x1	ra	Adresse de retour	Non
x2	sp	Pointeur de pile	Oui
x3	gp	Pointeur global	X
x4	tp	Pointeur de tâche	X
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

Convention pour les appel de fonctions

- appelante (*caller*) : positionne les paramètres dans les registres (et la pile si nécessaire), saute à l'appelée.
- appelée (*callee*) : réserve la place nécessaire à son propre contexte d'exécution.
- 8 premiers paramètres dans `a0` à `a7` les autres dans la pile.
- `a0` : contient la valeur de retour de la fonction.
- `sp` : contient l'adresse de la *dernière case* occupée dans pile.

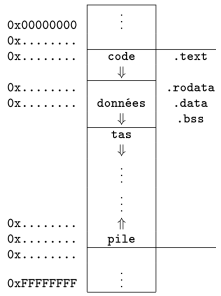


Directives assembleur

Présentes dans le code, ce ne sont pas des instructions à proprement parler mais sont interprétées au moment de la compilation afin de modifier le comportement de l'assembleur.

Directives concernant les sections

- `.text` : indique le début de la section correspondant au code. Lecture uniquement.
- `.data` : indique le début de la section correspondant aux données. En lecture/écriture.
- `.rodata` : section semblable à `.data` mais en lecture seulement.



Autres directives

- `.globl` *symbol* : rend *symbol* visible pour le linker.
- `.comm` *symbol*`,`*length* : déclare le symbole global *symbol* et lui alloue *length* octets non initialisés.
- `.byte` *expression* : crée un octet initialisé à la valeur *expression*.
- `.word` *expression* : crée un mot (4 octets) initialisé à la valeur *expression*.
- `.org` *expression* : place à l'adresse *expression* le code ou les données suivantes.

Infrastructure pour les TP

Étapes de développement d'une fonction

1. Compléter le contexte
2. Écrire la fonction en langage d'assemblage
3. Compiler le programme (`make toto`)
4. Exécuter votre programme avec QEMU
5. Déboguer votre fonction à l'aide de QEMU et GDB
6. Exécuter le script `verif_etud.sh`, corriger si nécessaire
7. Envoyez votre travail sur le dépôt (`git add/commit/push`)

Comportement de l'infrastructure

Pour chaque fonction `toto` à évaluer, seuls deux fichiers sont pris en compte pour l'évaluation à distance :

- le fichier `fct_toto.s` qui contient le programme que vous avez réalisé en langage d'assemblage,
- le fichier `test/toto.sortie` qui doit contenir la sortie de votre programme.

La présence du second fichier *non vide* est nécessaire car c'est elle qui indique à l'infrastructure d'évaluation qu'il faut évaluer la fonction `toto`.
L'évaluation automatique pour la fonction `toto` se fait en 7 étapes :

1. Vérification de l'existence d'un fichier *non vide* `test/toto.sortie`. Si absent, l'évaluation s'arrête.
2. Vérification de la présence et de la syntaxe du contexte de la fonction `toto`.
3. Compilation de la fonction `toto` et de son programme de test.
4. Vérification que la sortie de l'exécutable `toto` donne bien `test/toto.sortie`.
5. Est-ce que la sortie de votre programme est égale à la sortie d'un programme de référence écrit par vos enseignants ?
6. Est-ce que le contexte contient bien les éléments attendus au bon endroit ?
7. Est-ce que l'exécution de votre programme se déroule conformément à ce qui est attendu ?

Ces quatre premières étapes sont vérifiées par le script `../verif_etud.sh` à lancer depuis chaque répertoire de TP. Elles sont également effectuées par votre dépôt à chaque fois que vous poussez un ou plusieurs commit(s) et, si elles réussissent, votre dépôt demande une évaluation au dépôt d'évaluation qui effectue les trois vérifications suivantes :

Une fois toutes ces vérifications effectuées, le dépôt d'évaluation génère le fichier de log et le badge pour la fonction `toto` puis les rend publics.

Que faire si vos badges ne se mettent pas à jour

(a) Avez-vous bien poussé tous les fichiers sur votre dépôt (notamment les sorties) ?
Vérifiez sur l'interface en ligne de votre dépôt le contenu des fichiers.

(b) Est-ce que le script `verif_etud.sh` passe sans erreur ?
Une croix rouge (plutôt qu'une coche verte) sous le bouton bleu "clone" sur la page de votre dépôt indique que `verif_etud.sh` ne passe pas sur votre dépôt.

(c) Quelle est la date du dernier push pris en compte (en haut de la page de votre dépôt) ?
Il peut falloir jusqu'à 10 min pour que les badges se mettent à jour.

Description des contextes

Les contextes doivent suivre la grammaire ci-après. Pour faciliter sa compréhension, les non-terminaux sont **en gras**.

Contexte	→	Fonction : **ident** : **type** Contexte : **liste_contexte**
type	→	feuille | non feuille
liste_contexte	→	ε | **element_contexte** `\n` **liste_contexte**
element_contexte	→	**ident** : **list_localisation**
list_localisation	→	**localisation** | **localisation** ; **list_localisation**
localisation	→	registre **idreg** | registres **idreg** / **idreg**
	|	pile *(**adresse**) | pile *(**adresse**) / *(**adresse**)
	|	mémoire | mémoire, section **segment**
adresse	→	sp | sp + **nb** | sp - **nb**
segment	→	.data | .text | .rodata | .bss | ...

Les non-terminaux **idreg**, **ident** et **nb** représentent respectivement les noms de registres du processeur, les identifiants valides en C (noms de fonctions, de variables) et les entiers naturels ; ε représente le mot vide et `\n` un retour à la ligne. Toute ligne du contexte peut être suivi d'un commentaire commençant par `#` jusqu'à la fin de la ligne.

Déboguer son programme

Pour déboguer un programme, par exemple `pgcd` dans les commandes qui suivent, voici les étapes à suivre :

1. lancer le simulateur sur son programme en mode débogage : `qemu-system-riscv32 -machine cep -bios none -nographic -kernel pgcd -s -S`
2. lancer le débogueur sur son programme : `riscv32-unknown-elf-gdb pgcd`
3. placer un point d'arrêt sur la fonction `main` (ou sur la fonction `pgcd`) et avancer jusqu'à son exécution : `break main` puis `continue`
4. contrôler l'exécution en fonction de vos besoins : `next` pour passer à la ligne suivante et/ou `step` pour rentrer dans un appel de fonction