

IN330 — TP1 : Traitement de chaînes et assemblage pour RISC-V

Grenoble INP – Esisar – année 2024-25 • Version du 5 novembre 2024



Enseignant·e·s :

- Jean-Baptiste Caignaert (jean-baptiste.caignaert@esisar.grenoble-inp.fr)
- Laure Gonnord (laure.gonnord@grenoble-inp.fr)
- Ioannis Parissis (Ioannis.Parissis@grenoble-inp.fr)
- Sébastien Michelland (sebastien.michelland@lcis.grenoble-inp.fr)

◇ Étape 1 — Présentation du projet et de cette séance	1
◇ Étape 2 — Exercices préliminaires	2
◇ Étape 3 — Rendu de test (<i>durant la 1ère séance</i>)	4
◇ Étape 4 — Réalisation de l'assembleur RISC-V (<i>assembler/</i>)	5
◇ Étape 5 — Rendu officiel de l'assembleur (<i>date de rendu sur Chamilo</i>)	5



Étape 1 — Présentation du projet et de cette séance

Objectif

Au cours de son utilisation, un programme assembleur (sous sa forme textuelle) est d'abord *assemblé* sous sa forme binaire, puis *exécuté* par un processeur. Dans ce projet, on implémente ce processus en codant un *assembleur RISC-V* qui encode des instructions du langage assembleur RISC-V sous leur forme binaire, et un *émulateur RISC-V* qui simule l'effet de l'exécution de ces instructions sur un processeur RISC-V.

Les objectifs pédagogiques du projet sont les suivants :

- Mettre en œuvre un programme C “complexe” sur un sujet bas-niveau ;
- Évaluer un programme en concevant des *tests* ;
- Se familiariser avec les opérations bit-à-bit et la notation hexadécimale.

Description des produits

Le projet contient deux produits : un assembleur (avec des tests) et un émulateur (avec des tests), qui seront réalisés lors de **3 séances de 3,5 heures encadrées** et en autonomie dans les créneaux libres de vos emplois du temps.

Assembleur		Émulateur	
# add-simple.s	—————>	# add-simple.hex	—————> # add-simple.state
addi a1, zero, 20		01400593	x0: 0
addi a2, zero, 22		01600613	(...)
add a0, a1, a2		00c58533	x10: 42
			x11: 20
			x12: 22

L'**assembleur** prendra en entrée un fichier assembleur comme `add-simple.s`, et produira un fichier `add-simple.hex` avec les instructions encodées, écrites en hexadécimal.¹ Ce travail occupera la première des trois séances du projet.

¹Traditionnellement ce serait un fichier binaire, mais pour la simplicité ce sera ici un fichier texte.

L'émulateur prendra en entrée le fichier `add-simple.hex` et simulera l'exécution du processeur. Il produira un fichier `add-simple.state` avec la valeur finale de tous les registres à la fin de l'exécution. Écrire l'émulateur occupera les deux dernières séances du projet.

Les tests seront utilisés pour évaluer automatiquement votre projet et sont à développer en continu ; ils accompagneront tous les rendus. Vous devez consigner tous vos essais dans le dossier `tests/`. Les modalités des tests seront expliqués plus tard.

Modalités

- Le projet est à réaliser sous Linux par groupes de 2 élèves.
- Plusieurs rendus sont prévus ; il y a des instructions sur Chamilo.
- Les installations requises sont effectives sur les machines Linux des salles de TP.
- Les tests de ce projet utilisent un compilateur (`clang` ou `gcc`) pour vérifier que l'assembleur est correct. `clang` est installé sur les machines de l'école. Sur vos machines perso :
 - Linux (Debian/Ubuntu/etc): `sudo apt install llvm clang`.
 - Windows ([en utilisant WSL sous Windows](#)): pareil que Linux.
 - Mac OS ([en utilisant homebrew](#)): `brew install riscv64-elf-gcc`.
- **Le document sera mis à jour sur Chamilo au fur et à mesure du projet.**
- Si le sujet est flou, le code fourni ne marche pas, etc : sebastien.michelland@lcis.grenoble-inp.fr avec laure.gonnord@grenoble-inp.fr en Cc.

Ressources

Sur Chamilo, vous trouverez :

- Ce sujet (`IN330_2024_TP1.pdf`) ;
- L'archive du TP1 (`IN330_2024_TP1.zip`), contenant notamment :
 - Le code et les exercices pour l'assembleur (`exos/` et `assembleur/`) ;
- La documentation RISC-V nécessaire pour ce projet (`doc_riscv_projetC.pdf`).

Première séance de TP : l'étape d'assemblage.

L'assembleur doit lire le fichier d'entrée (par exemple `add-simple.s`) ligne-à-ligne, ignorer les lignes vides et les commentaires, et assembler les autres lignes. Chaque ligne contient une instruction RISC-V parmi les 13 listées dans la documentation fournie, et on fait l'hypothèse que le code est toujours correct (pas d'erreur de syntaxe, instruction non reconnue, etc).

Dans chaque instruction, l'assembleur doit séparer le mnémonique de l'instruction et ses arguments, encoder les arguments sous forme entière, et enfin combiner tous ces champs bit-à-bit. Le résultat de l'encodage de l'instruction est un entier de 32 bits, représenté en C par le type `uint32_t`.

Ces entiers de 32 bits sont finalement stockés dans le fichier de sortie (par exemple `add-simple.hex`) en les imprimant avec `fprintf()`.

Les exercices de l'étape 2 vous guident dans la réalisation des étapes importantes de ce programme. Le code que vous allez écrire dans les exercices sera à intégrer dans le code de l'assembleur. Moralement, l'étape 3 où il faut réaliser l'assembleur consiste surtout à connecter ensemble les solutions des exercices.



Étape 2 — Exercices préliminaires

Lecture d'un fichier ligne-à-ligne (`exos/getline.c`)

- Compiler `getline.c` et l'exécuter en spécifiant en paramètre le nom d'un fichier texte, par exemple `./getline getline.c`.

En observant le code source, répondre aux questions suivantes :

- Qu'est-ce qu'on affiche à chaque tour de boucle ?
- Dans le format "[%s]", on a mis des crochets pour délimiter l'affichage. Pourquoi le crochet fermant s'affiche-t-il sur la ligne suivante du terminal ? Expliquer le lien avec la longueur de la chaîne.
- Modifier le code pour supprimer le \n en fin de ligne *s'il existe*.
- Tester, bien sûr.

Séparation d'une instruction en mnémonique et arguments (exos/split_line.c)

Quelques exemples de lignes valides du fichier d'entrée de l'assembleur sont fournis dans `main()`.

- Écrire une fonction `void simplify_punct(char *str)` qui remplace les virgules et parenthèses ouvrantes/fermantes par des espaces dans la chaîne de caractères `str`.
- L'utiliser pour simplifier toutes les chaînes dans le tableau d'exemple, et tester.
- Écrire une fonction `bool is_comment(const char *str)` qui détermine si la ligne fournie en paramètre est un commentaire (i.e. son tout premier caractère est un dièse '#').

On veut maintenant séparer le mnémonique et les arguments de chaque instruction.

- Déclarer une chaîne de caractères `ins` de taille 16 (initialisée à la chaîne vide) et analyser chaque ligne d'exemple 1 (excepté les commentaires) avec `sscanf(l, "%s", ins)`. Observer les contenus de `ins` et la valeur de retour de `sscanf()`. Qu'a-t-on isolé dans chaque ligne ? Que se passe-t-il pour les lignes vides ?
- Ajouter une deuxième chaîne `arg1` (également initialisée à la chaîne vide) et essayer l'appel `sscanf(l, "%s%s", ins, arg1)`. Qu'a-t-on isolé cette fois ?
- Les instructions listées dans la documentation. Proposer et tester un format pour `sscanf()` qui sépare le nom de l'instruction et les arguments en 4 chaînes.
- Programmer cette analyse dans une fonction

```
bool split_line(const char *l, char *ins, char *arg1, char *arg2, char *arg3);
```

- Les chaînes de caractères `ins`, `arg1`, `arg2` et `arg3` seront allouées par l'appelant.
 - Mais `split_line()` les initialisera à la chaîne vide avec `strcpy()` (pour que les arguments non remplis par `scanf()` soient initialisés).
 - `split_line()` renverra `true` si la ligne contenant une instruction (i.e. pas vide, pas un commentaire) et `false` sinon.
- Tester sur les exemples donnés.

Encodage des arguments d'instruction sous forme entière (exos/parse_arg.c)

Ayant précédemment séparé les arguments en chaînes de caractères individuelles, on veut maintenant les encoder pour pouvoir construire les arguments. L'encodage de chaque argument est un nombre entier décrit dans la documentation.

- Catégoriser les types d'arguments possibles des instructions RISC-V considérées dans ce sujet en trois catégories (dont deux sortes de registres).
- Combien de caractères d'un argument est-il nécessaire (et suffisant !) de regarder pour déterminer dans quelle catégorie l'argument se trouve ?

On veut maintenant associer à chaque argument sa valeur entière (numéro de registre ou valeur numérique).

- Consulter le manuel de la fonction `atoi()` ; utiliser cette fonction pour gérer immédiatement un des cas.
- Étant donné une chaîne de caractères `arg` dont les contenus sont le texte "x28", de quel type est `arg+1` ? Vers quoi pointe-t-il ? Est-ce une chaîne de caractères valide ? Utiliser cette observation pour gérer un deuxième cas.
- Gérer le dernier cas en listant les 32 noms de registres RISC-V « jolis » sans faire 32 conditions.
- Écrire une fonction `int parse_arg(const char *arg)` qui prend en paramètre un argument non vide et renvoie sa valeur entière.

Encodage des instructions suivant la spécification RISC-V (`exos/encode.c`)

Soit l’instruction `sd x3, -8(x12)`. En lisant en parallèle la documentation RISC-V et le fichier fourni, répondre aux questions:

- De quel type est cette instruction ? Quel est son opcode, son registre source, l’offset, et son registre destination ?
- Expliquer la signature de la fonction `encode_S_type()` fournie : pourquoi des arguments entiers, pourquoi une valeur de retour entière sur 32 bits ?
- Expliquer comment est calculée cette valeur de retour pour une instruction de type S, à l’aide de vos connaissances sur les opérations booléennes `&`, `|`, `<<`, `>>` vues en TD. Expliquer la relation avec le format donné par la documentation.
- Quelle est la signification de la chaîne de formatage “%08x” utilisée dans le `printf()` du `main` ?
- Sur le même modèle, réaliser et tester les fonctions `encode_I_type()` et `encode_R_type()`.

Il reste maintenant à appeler ces fonctions d’assemblage correctement, c’est l’objet de la fonction `encode_instruction()`.

- Remarquer comment les informations spécifiques pour `sd` sont fournies à la fonction `encode_S_type()`.
- Sur le même modèle, ajouter des cas pour toutes les instructions RISC-V de la documentation, en appelant dans chaque cas l’encodage du type approprié. Faites attention à l’ordre des arguments !

Tester en profondeur !



Étape 3 — Rendu de test (*durant la 1ère séance*)

Dans cette étape, vous rendez sur Chamilo une archive selon une spécification. Votre chargé.e de TP vous fournit ensuite un diagnostic sur le fait que ce rendu est considéré valide ou pas, *durant la séance*.

L’objectif est de tester que vous saurez rendre votre assembleur “Rendu 1 Assemblage” en toute autonomie, à la deadline notée sur Chamilo, puisqu’il n’y aura pas d’autre séance avant ce rendu.

Instructions (dans `assembler/`)

1. À la troisième ligne de `README.md` et à la deuxième de `Makefile`, indiquez votre nom et prénom ainsi que ceux de votre binôme.
2. Compilez le code. Lancez-le manuellement avec `./riscv-assembler tests/add-simple.s tests/add-simple.hex`. Remarquez que `tests/add-simple.hex` a été créé mais est vide.
3. Lancez `make test` ; il doit se plaindre que le fichier `.hex` ne contient aucune instruction.
4. Répondez aux questions dans `README.md` (juste une case à cocher).
5. Tapez `make tar` pour créer une archive de votre dossier de travail. Ouvrez l’archive pour voir quels fichiers sont dedans (et quels fichiers ne sont *pas* dedans).

6. Soumettez sur Chamilo.



Étape 4 — Réalisation de l'assembleur RISC-V (*assembler/*)

Assemblage de l'assembleur

Vous avez maintenant tous les éléments pour réaliser l'assembleur de bout en bout. Dans le dossier *assembler/*, intégrez les fonctions écrites dans les exercices préliminaires et connectez les différents morceaux pour lire le fichier d'entrée et écrire les instructions dans le fichier de sortie. Vous pouvez créer autant de fichier *.c* et *.h* que vous le souhaitez, le *Makefile* les trouvera.

Le code fourni s'occupe de récupérer les noms du fichier d'entrée de sortie sur la ligne de commande. Vous devrez l'exécuter avec une commande du type `./riscv-assembler PROGRAMME.s PROGRAMME.hex`.

Pour l'affichage dans le fichier de sortie, vous ne *devez pas* convertir en hexadécimal à la main. L'un des exercices vous montre comment afficher en hexadécimal avec `printf()` : il vous suffit de remplacer ça par un appel à `fprintf()` en utilisant le `FILE *` du fichier de sortie.

Vous êtes invité-es à faire autant de `printf()` que vous le souhaitez pour surveiller l'exécution de votre code et diagnostiquer les bugs. L'affichage dans le terminal est complètement ignoré par le système de test ainsi que pour la correction du projet.

Les tests

Un système de test vous est fourni ; tapez `make test` (*attention, singulier*) pour lancer les tests.

Chaque fichier assembleur (**.s*) dans le dossier *tests/* est un programme de test. Le système vérifie automatiquement que le résultat de l'assemblage est correct en comparant avec le résultat produit par le compilateur `clang` (qui dispose d'un back-end RISC-V) ou `gcc`.

Attention, `make test` ne remplace pas l'exécution manuelle avec `./riscv-assembler`. En particulier, si vous avez un test qui ne passe pas, la première chose à faire est de le lancer tout seul à la main pour voir vos `printf()` et identifier à quel moment ça se passe mal.

Un exemple *add-simple.s* est fourni. Le bloc de commentaires qui commence par `EXPECTED` à la fin du fichier indique l'état final attendu après l'exécution. C'est utilisé pour l'émulateur uniquement, mais il vous est recommandé de les écrire dès le début du projet en réfléchissant à ce que vos programmes font, ça vous simplifiera le travail des prochaines séances.

Chaque ligne du bloc indique un nom de registre et la valeur que ce registre doit avoir à la fin de l'exécution. *Ne pas lister un registre revient à exiger qu'il ait la valeur 0 à la fin de l'exécution.*

Pour que le système de test fonctionne, il est important que le format des fichiers *.hex* et *.state* soit correct. Vous prendrez soin d'imiter l'exemple au début de ce sujet.



Étape 5 — Rendu officiel de l'assembleur (*date de rendu sur Chamilo*)

Avant de soumettre, vérifiez que :

1. Votre programme *compile* (c'est pas grave s'il ne marche pas mais il doit compiler) ;
2. Vos tests sont dans *tests/* et apparaissent quand on lance `make test` (même s'ils échouent) ;
3. L'archive générée par `make tar` contient bien tous vos fichiers.

Les correcteur.rice.s commenceront toujours par lancer `make test` ; le résultat et la qualité des tests font partie du barème.

Quand vous avez fini, répondez aux questions de ce rendu dans README.md et suivez les instructions de rendu sur Chamilo.