

# IN330 — TP2 et TP3 : Décodage d'instructions et émulation RISC-V

Grenoble INP – Esisar – année 2025-26 • Version du 16 septembre 2025



## Enseignant·e·s :

- Ioannis Parissis ([Ioannis.Parissis@grenoble-inp.fr](mailto:Ioannis.Parissis@grenoble-inp.fr))
- Lydie Terras ([lydie.terras@gmail.com](mailto:lydie.terras@gmail.com))

◇ Étape 1 — Présentation et découpage de ces séances .....	1
◇ Étape 2 — Exercices préliminaires. ....	3
◇ Étape 3 — Faire marcher un programme simple .....	5
◇ Étape 4 — Supporter le reste du jeu d'instructions .....	6
◇ Étape 5 — Étendre l'émulateur (optionnel pour les débutants) .....	6
◇ Rendu TP2 ( <a href="#">date de rendu sur Chamilo</a> ) .....	7
◇ Rendu TP3 ( <a href="#">date de rendu sur Chamilo</a> ) .....	7



## Étape 1 — Présentation et découpage de ces séances

Ce sujet couvre à la fois TP2 et TP3 et contient une partie obligatoire et une partie optionnelle. Pour les débutants, l'intention est d'étaler la partie obligatoire sur les deux séances. Pour les autres, vous pouvez faire la partie obligatoire sur une séance et la partie optionnelle sur l'autre.

Le rendu après TP2 sert d'intermédiaire et aura un coefficient plus faible que les autres.

## Objectif

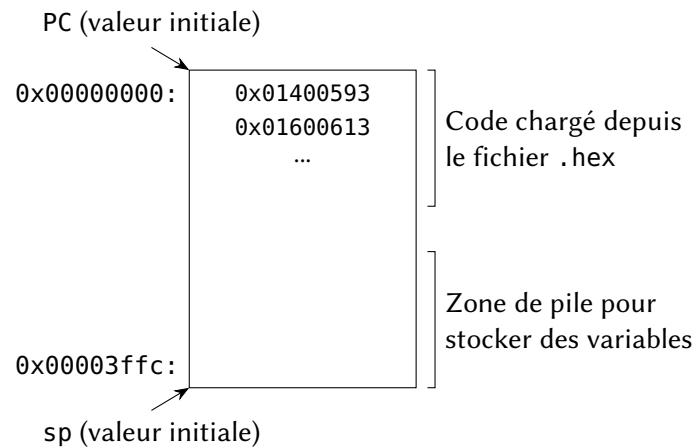
Dans le cycle de vie d'un programme assembleur, nous avons programmé (au TP1) l'étape d'assemblage qui met le code sous une forme binaire. Il nous reste maintenant à exécuter ce code dans un *émulateur* :

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'émulateur est un programme qui imite le comportement d'une machine ; ici on imite un processeur RISC-V 64 bits avec une mémoire de 16 ko. Globalement, le programme qu'on construit va :

1. Charger un fichier .hex dans la mémoire de l'émulateur ;
2. Exécuter les instructions de ce programme jusqu'à arrêt ;
3. Afficher les valeurs finales des registres dans un fichier .state.

La mémoire de la machine, qui fait 16 kio (16384 = 0x4000 octets), sera initialisée à 0 au début de l'exécution. Le code sera chargé au début (i.e. à l'adresse 0) et la pile commencera à la fin (ie. le registre sp sera initialisé à 0x4000).



*Agencement de la mémoire pendant l'exécution.*

Les instructions font 32 bits mais les accès mémoire pour écrire des données en mémoire se feront sur 64 bits (pour stocker des copies des registres) donc la mémoire sera un peu plus subtile qu'un simple tableau pour permettre ces accès de tailles variées.

La majorité du travail sera dans l'émulation de la boucle de lecture-décodage-exécution du processeur, qui va essentiellement :

1. Lire à l'adresse indiquée dans le registre PC une instruction de 32 bits dans la mémoire ;
2. Si cette instruction a la valeur 0, le programme s'arrête (par convention pour ce projet) ;
3. Sinon, décoder l'instruction suivant la documentation RISC-V déjà utilisée pour l'assembleur ;
4. Exécuter l'instruction, ce qui modifie la valeur des registres, de la mémoire, ou PC lui-même ;
5. Recommencer.

À la fin de l'exécution, l'émulateur devra écrire dans un fichier `.state` l'état final des registres.

## Description des produits

**L'émulateur** prendra en entrée un fichier produit par l'assembleur, comme `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.

**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 rappelées plus tard.

## Modalités

- Les réponses aux questions sont à noter dans `emulator/README.md`.
- Pour le reste, pareil que le TP1.
- Si le sujet est flou, le code fourni ne marche pas, etc : [sebastien.michelland@lcis.grenoble-inp.fr](mailto:sebastien.michelland@lcis.grenoble-inp.fr) avec [laure.gonnord@grenoble-inp.fr](mailto:laure.gonnord@grenoble-inp.fr) en Cc.

## Ressources

Sur Chamilo, vous trouverez :

- Ce sujet (`IN330_2024_TP23.pdf`) ;
- L'archive du TP2/3 (`IN330_2024_TP23.zip`), contenant notamment :
  - Le code et les exercices pour l'émulateur (`exos/` et `emulator/`) ;
- La documentation RISC-V déjà utilisée au TP1 (`doc_riscv_projetC.pdf`).



## Étape 2 — Exercices préliminaires

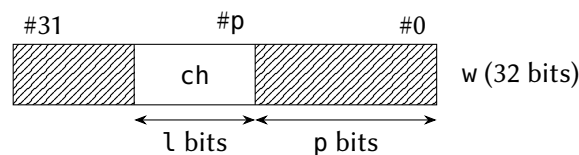
### Extraction de champs de bits et extensions de signe (exos/extract.c)

Le but de cet exercice est de préparer les manipulations de bits nécessaires pour décoder les instructions RISC-V à émuler (i.e. le contraire de ce qu'on a fait au TP1).

On considère l'entier de 32 bits  $w = 0x00b60633$ .

1. Écrire  $w$  en binaire. De quel côté est le bit #0 ? Quelle est la valeur (en décimal) du champ de bits qui commence à la position 0 et est de longueur 7 ?
2. Quelle est de même la valeur (décimale) du champ de bits qui commence à la position 15 et est de longueur 5 ?

On veut faire ce calcul de façon générale pour extraire d'un entier  $w$  de 32 bits un champ quelconque, donné par sa position  $0 \leq p \leq 31$  et sa taille  $1 \leq l \leq 31$ , et le stocker dans un nouvel entier.



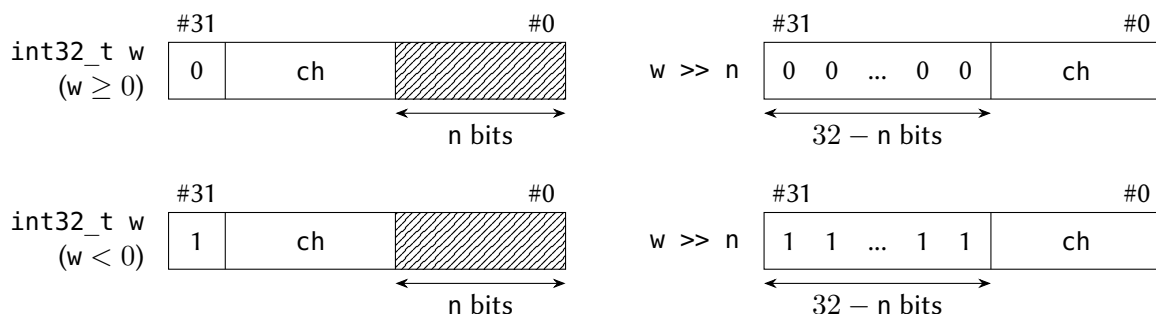
3. (Sur papier) Écrire  $(1 \ll 6)$  et  $(1 \ll 6) - 1$  en binaire. En déduire une formule pour générer un entier dont les  $l$  bits de poids faible valent 1 et les autres 0.
4. Programmer la fonction `extract_field(...)`. La méthode recommandée est de décaler  $w$  vers la droite pour que le champ  $ch$  recherché soit à la position 0, puis de forcer à 0 tous les bits hors de  $ch$  en utilisant le masque trouvé à la question précédente.

La deuxième étape est de reconstruire des entiers signés proprement, car en complément à deux la représentation des nombres négatifs dépend de la taille du champ.

5. (Sur papier) Écrire en binaire l'entier 15 sur 8 bits et 12 bits, puis écrire l'entier -15 sur 8 bits et 12 bits. En déduire une méthode pour déterminer la valeur des 4 nouveaux bits lorsqu'on passe d'une écriture sur 8 bits à une écriture sur 12 bits (on appelle ça une « extension de signe » de 8 vers 12 bits).
6. (Sur papier) On se donne les entiers  $0xffff$  et  $0x3fff$ . Expliquer comment on peut déterminer leur extension de signe de 12 bits vers 32 bits (et la donner) sans calculer le nombre entier qu'ils représentent.

Une façon « naïve » de faire une extension de signe de  $l$  vers 32 bits serait donc de regarder si le bit numéro  $l-1$  est à 1, et si oui, mettre à 1 les bits numéros  $l$  à 31. Mais on peut faire plus rapide en s'appuyant sur l'opérateur de décalage à droite.

L'opérateur de décalage vers la droite, lorsqu'appliqué à un nombre non signé, insère des 0 à gauche. Mais sur les nombres signés, il répète le bit de poids fort (qui reflète le signe) :



Une astuce pour calculer l'extension de signe d'un nombre  $x$  est donc de décaler  $x$  vers la gauche pour que le bit numéro 1-1 devienne le bit de poids fort, de cast le résultat en un type d'entier signé, puis de le re-décaler à droite, répétant ainsi le bit de signe.

7. Programmer ce processus dans la fonction `sign_extend(...)`.
8. Bien sûr, tester tout ça dans `main()`.

### Décodage du format binaire des instructions (exos/decode.c)

Dans cet exercice, on désire “inverser” la fonction d'encodage binaire du TP1, c'est-à-dire, à partir d'un entier non signé, l'interpréter comme une instruction et récupérer les champs associés.

Observons d'abord l'implémentation de la fonction `B_type()` décodant une instruction de type B (remarquer le passage de certains paramètres par adresse):

1. Expliquer la relation entre la formule par laquelle `rs1` est récupéré et l'encodage du type B. Que représente le 15 ? Que représente le `0x1f` (astuce : question 3 de `extract.c`) ?
2. Le calcul de `imm` prend quatre lignes. En observant les décalages vers la gauche, identifier quel fragment de l'immédiat est extrait par chaque ligne.

Vérifier en exécutant les tests fournis que `B_type()` est bien l'inverse de `encode_B_type()`.

3. Sur le même modèle, programmer `R_type()`, `I_type()`, et `S_type()`. Vous pouvez faire la manipulation des bits directement, comme dans `B_type()`, ou bien utiliser les fonctions de l'exercice précédent si vous préférez. Pensez bien à l'**extension de signe** pour les deux immédiats.
4. Ajouter des tests dans `main()` pour vérifier que vos fonctions sont effectivement les inverses de `encode_{R,I,S}_type()`. Vous prendrez soin de tester des immédiats positifs et négatifs ainsi que des valeurs assez grandes pour toucher tous les bits de l'immédiat.

### Analyse des infos machine (emulator/emulator.h)

On veut maintenant analyser le code fourni pour l'émulateur, sans le modifier pour l'instant. Le réflexe à avoir dans cette situation est de regarder en premier les fichiers d'en-tête, qui nous donnent une vue d'ensemble des types de données et des fonctions ; ici on en a un seul, `emulator.h`.

1. L'état de la machine émulée est représenté par la `struct machine`. Décrire brièvement ce que PC, regs et memory sont. Dans un système réel, sur quelles puces électroniques sont ces trois éléments ?
2. Quelle est, implicitement, l'unité de `MACHINE_MEMSIZE` ? Pourquoi la taille du tableau `memory` divise-t-elle cette constante par 8 ?
3. RISC-V propose plusieurs instructions pour lire en mémoire, notamment `lb`, `lh`, `lw` et `ld`. Que veulent dire ces acronymes ? Combien d'octets sont lus par chacune de ces instructions ?
4. En déduire ce que les noms `machine_luw()` et `machine_suw()` veulent dire. À votre avis, quel est l'intérêt d'inclure le préfixe “`machine_`” sur certaines fonctions ? Comparer à une fonctionnalité qui existe en Java et en Python mais pas en C.

### Analyse de l'implémentation (emulator/\*.c)

Le projet contient plusieurs fichiers `.c` ; parcourez rapidement chaque fichier pour vous faire une idée des contenus. Une stratégie classique est de regarder les noms des fonctions et le type des arguments, ce qui donne une idée de l'objectif de la fonction et quelles données elle manipule. On croise aussi avec les commentaires fournis dans le fichier d'en-tête.

1. L'auteur a réparti le code de façon à donner un rôle différent à chaque fichier `.c`. Identifier en quelques mots la tâche dont chaque fichier s'occupe.

Dans un projet C, chaque fichier peut en principe appeler des fonctions de n'importe quel autre fichier. Mais le code est organisé de sorte que seuls *certain*s fichiers `.c` appellent des fonctions venant d'autres fichiers `.c` bien choisis. Par exemple, `main.c` appelle la fonction `emulate()` qui est fournie par `emulate.c`, et donc `main.c` appelle `emulate.c`.

2. Lister tous les cas où un fichier `.c` en appelle un autre.<sup>1</sup>
3. Dans `emulate()`, avant d'appeler la fonction `machine_init()`, le tableau mémoire de 16 ko est alloué avec `malloc()`. Quelles auraient été les autres options ? Pourquoi `malloc()` est-elle préférable ?
4. La fonction `emulate()` a deux paramètres `fp_in` et `fp_out`. Que veulent dire ces noms ?<sup>2</sup>
5. Dans `emulate()`, on commence par charger le code du programme (depuis le fichier `.hex`), et ensuite on fait la boucle de lecture-décodage-exécution. Pourquoi ne peut-on pas exécuter les instructions en même temps qu'on les lit ? Comparer avec le fait que dans l'assembleur on lisait les instructions de haut en bas et on ne les regardait qu'une seule fois.



### Étape 3 — Faire marcher un programme simple

On a maintenant tous les outils nécessaires pour finir l'émulateur. Cette section fait le tour des modifications nécessaires pour faire marcher le programme `add-simple.s` fourni en exemple. Pour rappel, vous pouvez le lancer manuellement avec :

```
% ./riscv-emulator tests/add-simple.ref.hex tests/add-simple.state
```

Le fichier `.ref.hex` est généré automatiquement par `make test` avec `clang` ou `gcc` : pas besoin d'utiliser votre assembleur du TP1.

Un point important du programme est que **vous pouvez afficher avec `printf()` à tout moment** car seul le fichier `.state` est utilisé pour l'évaluation. N'hésitez pas à ajouter des affichages partout où vous le sentez utile pour debugger.

- Complétez l'initialisation des informations machines dans `machine_init()`<sup>3</sup>.
- Ajoutez l'affichage de l'état final des registres dans `emulate()`. L'affichage doit se faire dans le fichier `fp_out` : vous pourrez utiliser `fprintf(fp_out, ...)` qui s'utilise comme `printf()` sauf qu'on spécifie le fichier de sortie en premier paramètre.

À ce stade, vous pouvez tester le projet ; vous devez avoir trois erreurs sur `add-simple.s` indiquant que les registres `a0`, `a1` et `a2` n'ont pas la valeur attendue. Sinon, vous avez probablement écrit le fichier `.state` de façon incorrecte : comparez-le avec l'exemple au début de ce sujet.

- Programmez la boucle de lecture-décodage-exécution dans `emulate()`. Vous devrez lire une instruction à l'adresse PC, si elle vaut 0 arrêter le programme, et sinon appeler `execute_instruction()`.

`execute_instruction()` inspecte les bits des champs `opcode`, `funct3` et `funct7` des instructions pour déterminer leur type ; `bge` et `jal` sont fournis, et les fonctions `do_bge()` et `do_jal()` qui finissent de décoder et exécuter sont fournies également.

- Détectez `addi` dans `execute_instruction()` et créez la fonction `do_addi()` avec juste un `printf()` dedans.

À ce stade, l'émulateur doit afficher le `printf()` quand on l'exécute manuellement (avec `make test` la sortie du programme n'est pas affichée).

<sup>1</sup>Il n'y en a que deux, et l'un est donné ci-dessus.

<sup>2</sup>Internet pourra vous dire ce que `fp` signifie (c'est un nom classique pour une variable de type `FILE *`).

<sup>3</sup>Note de syntaxe `s->field` est un raccourci pour `(*s).field`

- Ajoutez ensuite `I_type()` au programme et utilisez-la dans `do_addi()` pour afficher la valeur des arguments.
- Finir `do_addi()` en modifiant les registres via `mach` (n'oubliez pas `PC += 4`).

Vous devriez maintenant avoir les bonnes valeurs pour `a1` et `a2` dans le `.state`.

A cet étape, les débutant.e.s peuvent rendre le TP2. Voir « Rendu TP2 ». Les non-débutants doivent avancer puis rendre l'étape 4.



## Étape 4 — Supporter le reste du jeu d'instructions

Le reste du jeu d'instruction prévu est à implémenter et tester dans cette étape 4. La méthode est la même pour toutes les instructions ; faites juste attention au fait que dans `execute_instruction()`, pour détecter les instructions il ne faut pas regarder que `opcode` mais aussi `funct3` et `funct7` (s'ils existent).

On rappelle qu'il est *crucial* que vous ayez des tests qui comportent un bloc `EXPECTED` (voir section suivante) et qui se comportent bien à l'exécution (pas de boucles infinies, accès mémoire hors bornes, etc). Il est donc conseillé d'écrire des petits programmes faisant chacun une tâche simple pour simplifier la phase de test/debugging.

Les tests seront à livrer.

Cette étape est la fin du projet pour les débutant.e.s.

### Rappel du fonctionnement du système de test

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 assemble le programme automatiquement avec `clang` ou `gcc` (en remplacement de l'assembleur produit au TP1) et le fait exécuter par l'émulateur. Il vérifie ensuite que le fichier `.state` généré liste les mêmes valeurs de registres que le bloc `EXPECTED` du test.

Attention, `make test` ne remplace pas l'exécution manuelle avec `./riscv-emulator`. 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. 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 du fichier `.state` soit correct. Vous prendrez soin d'imiter l'exemple au début de ce sujet.



## Étape 5 — Étendre l'émulateur (optionnel pour les débutants)

Le but de cette partie est de faire tourner un programme qui vous est fourni sous le nom `blob.hex` et a un comportement non spécifié. Ce programme ne peut pas être émulé directement à la fin de l'étape 4 car il manque quelques instructions et fonctionnalités à l'émulateur. Le code à ajouter est facile à écrire : le défi principal de cette étape est que vous devrez procéder par essai-erreur et explorer la documentation pour trouver ce qu'il faut ajouter.

Vous pourrez trouver la documentation officielle du jeu d'instructions RISC-V sur <https://riscv.org> ; la section technique du site contient la spécification des instructions non-privilégiées (« Volume 1 »). `blob.hex` nécessite 6 instructions supplémentaires qui sont toutes listées dans le Chapitre 2 (“RV32I”) et dont les encodages sont donnés dans le Chapitre 34 (“Instruction Set Listings”).

L'une des 6 instructions manquantes est `ecall`, qui est normalement utilisée pour exécuter des *syscalls*. Dans ce TP, on détourne un *chouille* son usage : on utilise `ecall` pour afficher du texte dans le terminal. Lorsque `ecall` est exécuté, l'émulateur devra lire un caractère dans `a0` et l'afficher sur `stderr`. Afficher sur `stderr` au lieu de `stdout` vous permet de filtrer les messages de debug de votre émulateur et de ne voir que le texte affiché par le programme en lançant l'émulateur comme ceci :

```
% ./riscv-emulator blob.hex blob.state >/dev/null
```

Il est recommandé de raconter au fur et à mesure ce que vous faites dans `README.md`. Bon courage !



### **Rendu TP2** (*date de rendu sur Chamilo*)

Ce rendu intermédiaire permet de voir où vous en êtes et aura un coefficient faible. Les dates de rendu sont sur Chamilo.

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. Dans `README.md`, indiquez si vous planifiez de sauter l'étape 5.
2. Vérifiez que l'émulateur compile (ce sera le cas si vous n'avez pas encore commencé l'étape 3). Notez que vous perdez des points si le code rendu ne compile pas.
3. Répondez aux questions dans `README.md` (selon votre avancement dans le projet).
4. Créez l'archive avec `make tar` et soumettez-la telle quelle sur Chamilo.



### **Rendu TP3** (*date de rendu sur Chamilo*)

Mêmes instructions que TP2. Les dates de rendu sont sur Chamilo.