

# SAÉ S5.A.01

Émulateur processeur RISC-V

Michaël Hauspie

16 septembre 2024



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Description générale de la SAË . . . . .	7
1.1.1	Objectifs . . . . .	7
1.1.2	L'émulation . . . . .	7
1.2	Rappel d'architecture . . . . .	7
1.2.1	Registres . . . . .	8
1.2.2	Unité Arithmétique et Logique (UAL) . . . . .	8
1.2.3	Mémoire et espace d'adressage . . . . .	8
1.2.4	Unité de contrôle . . . . .	9
1.2.5	Entrées/sorties . . . . .	9
1.3	RISC-V . . . . .	9
1.4	Outils à votre disposition . . . . .	10
<b>2</b>	<b>Première étape : découverte du codage du jeu d'instructions</b>	<b>11</b>
2.1	Programme <i>natif</i> et langage assembleur . . . . .	11
2.2	Codage des instructions . . . . .	11
2.2.1	Notations . . . . .	12
2.2.2	Forme générale et <i>opcode</i> . . . . .	12
2.2.3	Registres . . . . .	14
2.2.4	Valeurs immédiates . . . . .	14
2.3	Livrable 1 : un premier décodeur . . . . .	16
2.4	Livrable 2 : un désassembleur complet . . . . .	17
<b>3</b>	<b>À suivre...</b>	<b>21</b>



# Préambule

Ce document présente le travail à réaliser dans le cadre de la SAÉ du semestre 5 du BUT Informatique, Parcours Réalisation d'applications : conception, développement, validation.

De nombreux éléments sont soit directement extrait de la spécification RISC-V, distribuée sous licence Creative Commons CC-BY 4.0, soit retranscrits et adaptés. Notamment, les figures représentant les différents codages (comme la Figure 2.1) sont celles de la spécification reprises à l'identique.

Ce document est lui même placé sous licence Creative Commons CC-BY-NC-SA 4.0





# Chapitre 1

## Introduction

### 1.1 Description générale de la SAÉ

#### 1.1.1 Objectifs

Dans le cadre de cette SAÉ, vous allez être confronté.e.s au développement d'un logiciel qui sort du périmètre dans lequel vous avez l'habitude d'évoluer dans les différentes ressources du BUT. L'objectif de la SAÉ est triple :

1. apprendre un nouveau langage de programmation, voire un nouveau paradigme de programmation (ou au moins faire utiliser un autre langage que Java ou Javascript);
2. (re)découvrir le fonctionnement d'un processeur;
3. écrire un programme respectant une spécification stricte.

Pour répondre à ces objectifs, vous allez développer un logiciel permettant d'émuler un ordinateur basé sur un processeur RISC-V, en utilisant le langage de programmation de votre choix. La seule contrainte est que ce langage ne soit ni Java, ni Javascript. En raison des contraintes de performance liée au développement d'un émulateur, un langage qui se compile vers du code natif serait particulièrement indiqué. On peut par exemple citer C, C++, Rust ou encore Go.

Vous êtes cependant libre d'en choisir un autre. Les exemples de code de ce document seront écrit en C et en Rust.

#### 1.1.2 L'émulation

L'émulation est le fait de simuler le fonctionnement d'une architecture matérielle à l'aide d'un programme informatique. Le programme est en charge de modéliser et de faire évoluer l'état « virtuel » de l'architecture matérielle.

Le logiciel d'émulation (que nous nommerons « émulateur » à partir de maintenant) doit donc proposer une structure de données qui représente le matériel que l'on cherche à simuler et faire évoluer ces données au fur-et-à-mesure de l'avancée de la simulation. Pour en savoir plus sur la notion d'émulation, vous pouvez vous référer à la page Wikipedia qui y est consacrée.

### 1.2 Rappel d'architecture

Afin de comprendre comment nous allons simuler (ou imiter) le fonctionnement d'un processeur, nous allons revenir sur l'architecture générale d'un ordinateur, l'architecture de von Neumann.

Dans sa vision la plus simple, un ordinateur possède :

- une unité de calcul constituée de *registres* et d'une unité arithmétique et logique (UAL, *ALU* en anglais);
- une unité de contrôle constituée principalement d'un compteur de programme;

- une mémoire qui contient à la fois les données manipulées et le programme à exécuter ;
- des périphériques d’entrée/sortie.

### 1.2.1 Registres

Les registres sont les plus petites unités de stockage du processeur. Ils sont intégrés à celui-ci et n’ont pas d’adresse (car il ne sont pas en mémoire). Ils sont directement utilisables dans les instructions machine et servent de stockage de traitement. L’unité arithmétique et logique travaille exclusivement avec eux sur la plupart des architectures. Quelques exemples sur des architectures connues :

- Intel : `rax`, `rbx`, `rsi`, `rdi`...
- ARM : `r0`, `r1`, `r2`...
- RISC-V : `x0`, `x1`, `x2`...

Les registres s’utilisent directement dans le langage assembleur lié à l’architecture. Par exemple, l’instruction RISC-V du programme 1 réalise l’opération *ajouter 10 au contenu du registre x5 et stocker le résultat dans le registre x4*.

```
; x4 = x5 + 10
addi x4, x5, 10
```

Programme 1: Exemple d’instruction RISC-V.

Sur certaines architectures, ces registres peuvent avoir des *alias* (c’est à dire plusieurs noms différents). C’est particulièrement le cas sur RISC-V où `x0` se nomme également `zero`, `x1` se nomme également `ra`... Nous y reviendrons dans la description de l’architecture RISC-V.

Lorsque l’on parle de données manipulées par un processeur, on utilise fréquemment la notion de *mot*. Un *mot* correspond à une unité de donnée *naturelle* pour le processeur. Il s’agit généralement de la taille de ses registres.

On parlera alors d’un *mot* de 4 octets sur un processeur 32 bits, de 8 octets sur un processeur 64 bits. De même, on pourra parler de *demi-mot* pour désigner une valeur de taille 2 octets sur un processeur 32 bits. Nous retrouverons cette notion quand nous décrirons le jeu d’instructions RISC-V.

### 1.2.2 Unité Arithmétique et Logique (UAL)

L’UAL est la partie du processeur qui exécute les intructions. Elle réalise des calculs (additions, soustractions...), des accès mémoires (lecture, écriture en mémoire) ou du contrôle de flot d’exécution (tests et branchements permettant de réaliser des structures *si*, *sinon*, des boucles...). Elle travaille sur les registres.

### 1.2.3 Mémoire et espace d’adressage

La mémoire de travail permet de stocker le programme et les données. Elle est reliée à l’unité arithmétique et logique par un *bus d’adresse* et un *bus de données*. Le bus d’adresse permet au processeur d’indiquer à la mémoire à quelles données il veut accéder (à l’aide d’une *adresse*). Il récupère (pour une lecture) ou fournit (pour une écriture) les données par le bus de données.

On appelle *espace d’adressage* l’ensemble des adresses auquel le processeur peut accéder. Cet espace d’adressage n’est pas forcément égal à la taille de la mémoire disponible sur le système. Par exemple, un processeur ayant des adresses 32 bits a un espace d’adressage de taille  $2^{32}$  octets (soit 4294967296 octets, soit encore 4 Giga octets). Un processeur avec un bus de 64 bits a un espace d’adressage de  $2^{64}$  octets soit... beaucoup trop pour fabriquer une mémoire qui possède autant d’octets.

Un système n’a également pas forcément qu’une mémoire. Par exemple, les micro-contrôleurs – petits ordinateurs intégrés qui équipent nos petits objets (objets connectés, réfrigérateurs, carte bancaire...) – possèdent généralement deux mémoires :

- une mémoire *volatile*, la RAM, dont le contenu est effacé quand on coupe l’alimentation ;



- une mémoire *persistante*, la ROM, dont le contenu persiste même après un arrêt de l'alimentation électrique. De nombreuses technologies existent pour ce type de mémoire avec des possibilités variables (lecture seule complète, ré-inscriptible...);

Ces deux types de mémoire cohabitent dans le même espace d'adressage. On pourrait par exemple imaginer une architecture dans laquelle :

- les adresses de 0x00000000 à 0x00080000 permettent d'accéder aux 512 KB d'une mémoire RAM;
- les adresses de 0x20000000 à 0x20a00000 permettent d'accéder aux 10 MB d'une mémoire ROM.

### 1.2.4 Unité de contrôle

c'est elle qui régit le fonctionnement du processeur. Son composant le plus important est le compteur de programme. Il s'agit d'un registre particulier qui contient l'adresse de l'instruction à exécuter. Sur RISC-V, ce registre se nomme pc. L'algorithme exécuté par l'unité de contrôle (et donc par le processeur) est le suivant :

1. lire en mémoire l'instruction située à l'adresse contenue dans le registre pc ;
2. décode cette instruction
3. l'exécuter et mettre à jour pc pour pointer sur l'instruction suivante à exécuter ;
4. retour en 1

La mise à jour de pc (étape 3) dépend du type d'instruction :

1. pour une instruction *normale* on ajoute à pc la taille de l'instruction courante
2. pour une instruction de branchement sans condition, on modifie pc en lui affectant l'adresse indiquée dans l'instruction de branchement ;
3. pour une instruction de branchement avec condition, on évalue la condition. Si elle est vraie, on effectue l'opération 2., si elle est fausse, l'opération 1.

### 1.2.5 Entrées/sorties

Les périphériques d'entrées/sorties permettent au processeur d'interagir avec le monde extérieur. Il peut s'agir d'un clavier, d'un écran, d'un simple port série, d'une carte son, d'une carte graphique...

Selon l'architecture, ces périphériques sont accessibles via des instructions particulières (in et out sur Intel par exemple<sup>1</sup>) ou rendus visibles dans l'espace d'adressage, comme sur ARM. Il suffit alors d'écrire ou lire à une adresse spécifique pour faire réaliser des opérations à un périphérique.

## 1.3 RISC-V

RISC-V<sup>2</sup> n'est en réalité pas directement un processeur. Il s'agit d'un standard ouvert de jeu d'instructions (ou *ISA*, *Instruction-Set Architecture*). C'est donc avant tout un document décrivant une architecture et un jeu d'instructions. Libre ensuite à chacun de l'implémenter, c'est à dire de concevoir un processeur qui respecte sa spécification et son jeu d'instructions.

D'abord conçu par l'université de Berkeley comme un support à la recherche et à l'éducation, RISC-V est devenu un standard utilisé par l'industrie. De nombreuses implémentations voient le jour, qu'elles soient libre ou propriétaires. L'intérêt de concevoir un processeur respectant le standard étant de pouvoir utiliser l'ensemble des suites logicielles permettant de développer pour ce processeur (notamment assembleurs et compilateurs).

La spécification RISC-V est organisée en plusieurs jeux d'instructions de base auxquels peuvent s'ajouter des extensions. Les jeux de bases définissent l'architecture minimale ne travaillant qu'avec des nombres entiers et se déclinent en plusieurs versions :

- RV32I : jeu d'instructions pour architecture 32 bits ;

1. on peut également avoir accès à des périphériques via des adresses particulières sur Intel.

2. prononcer « risk-five ».

- RV64I : jeu d'instructions pour architecture 64 bits ;
- RV128I : jeu d'instructions pour architecture 128 bits ;
- RV32E et RV64E : jeux d'instructions dédiés aux systèmes embarqués.

Pour ce projet, vous allez émuler le jeu d'instructions RV32I, le plus simple de la spécification. Il est basé sur une architecture 32 bits et de 42 instructions permettant de manipuler des nombres entiers.

Le jeu d'instruction indique les *opérations* que peut réaliser une unité de traitement RISC-V. Le concepteur d'une architecture basée sur RISC-V peut décider de nombreuses choses non imposées par la spécification :

- nombre d'unités de traitement disponibles (mono-coeur ou multi-coeur...);
- organisation de la mémoire ;
- périphériques disponibles et mode d'interaction ;
- ...

Dans ce projet, vous êtes le/la conceptrice de l'architecture. Nous allons faire les choix suivants<sup>3</sup>

- l'architecture contiendra une seule unité de traitement RISC-V fournissant le jeu d'instruction RV32I non privilégié (32 bits, manipulation de nombres entiers, un seul mode de privilège d'exécution) ;
  - elle disposera de 512 KB de mémoire RAM, accessibles de façon contiguë entre les adresses 0x00000000 et 0x00080000 ;
  - au démarrage, l'unité de traitement se met à exécuter le programme à l'adresse 0x100.
- Nous ajouterons d'autres contraintes/fonctionnalités plus loin dans le projet.

Pour toute la durée du projet, en plus de ce sujet, la spécification du jeu d'instruction RISC-V sera votre document de référence. Il est disponible à l'adresse suivante : <https://github.com/riscv/riscv-isa-manual/releases/tag/20240411>

Le fichier qui nous intéresse est `unpriv-isa-asciidoc.html` (ou sa version PDF à votre convenance). Il contient la spécification *non privilégiée* des jeux d'instructions de base (RV32I, RV64I...) ainsi que des extensions non privilégiées. Lorsque le présent document fera référence à cette spécification, il indiquera la section concernée. Par exemple, le jeu d'instructions RV32I est défini à la section 2.

Bien que cela puisse être instructif, vous n'avez pas besoin de lire cette spécification en entier. Le sujet donnera toujours un résumé des informations indispensables et le numéro de section associée. Vous pourrez aller voir la spécification officielle pour avoir plus de précisions.

## 1.4 Outils à votre disposition

Pour vous aider tout au long de cette SAE, vous pouvez utiliser

- quelques programmes d'exemple, en assembleur et en C, avec les outils pour compiler vers un fichier binaire contenant des instructions RISC-V. Pour les compiler, vous devez avoir installé la suite de compilation RISC-V.
  - sous Debian ou Ubuntu : `apt-get install gcc-riscv64-unknown-elf`
  - sous MacOS : `brew install riscv64-elf-gcc`
- Pour la plupart de ces exemples, il suffit de lancer la commande `make` dans le répertoire contenant les sources et un ou plusieurs fichiers ayant pour extension `.bin` seront créés suite aux opérations de compilation et s'assemblent. Ce sont ces fichiers `.bin` qui devront être chargés par votre émulateur.
- <https://github.com/TheThirdOne/rars> : un IDE pour assembleur RISC-V, écrit en Java. Pratique pour tester et assembler les instructions une à une

---

3. au moins dans un premier temps, on pourra imaginer des extensions par la suite si le temps le permet.

## Chapitre 2

# Première étape : découverte du codage du jeu d'instructions

Le premier attendu de ce projet vous permettra de vous familiariser avec le décodage des instructions. Pour cela, vous allez écrire un programme qui prend en entrée un fichier contenant des instructions RISC-V au format **binaire** et qui produira en sortie du **texte**, représentant ces instructions en utilisant leur représentation textuelle en langage assembleur.

### 2.1 Programme *natif* et langage assembleur

Une unité de traitement RISC-V exécute un programme stocké en mémoire en format **binaire** (par opposition à texte). Dans le jeu d'instructions qui nous intéresse (RV32I), toutes les instructions sont codées sur 32 bits. Ce programme est dit *natif* au sens où il est constitué de suites d'octets directement exécutables par l'unité de traitement, sans aucune transformation logicielle.

Un programme est donc une succession de mots de 4 octets, chacun de ces mots représentant une instruction. Par exemple, en RV32I, le mot ayant pour valeur 0x00a10093 peut s'écrire sous la forme texture du programme 2.

```
; Réalise l'opération: x1 = x2 + 10  
addi x1, x2, 10
```

Programme 2: Forme textuelle du mot 0x00a10093.

Cette forme textuelle n'est qu'un code source, elle ne peut pas être exécutée directement par le processeur. Pour cela, il faut *l'assembler*, c'est à dire transformer le texte `addi x1, x2, 10` en un mot de 4 octets valant 0x00a10093.

À la différence des langages que vous manipulez habituellement (comme le Java ou le C), cette traduction est simple est directe, il ne s'agit pas d'une réelle compilation, juste d'un changement de représentation d'une forme compréhensible par un humain à une forme compréhensible par un processeur. Il s'agit par contre bien de la même information : l'instruction qui ajoute 10 à la valeur contenue dans le registre x2 et range le résultat dans le registre x1.

### 2.2 Codage des instructions

Nous allons maintenant détailler la façon dont les instructions sont *codées* dans le jeu d'instructions RV32I. Il nous faut d'abord introduire quelques notations.

### 2.2.1 Notations

Pour toute la suite du document, on considère que, pour un mot de 32 bits, le bit 0 est le bit de poids faible et le bit 31 est le bit de poids fort. Dans toutes les représentations graphiques de codage de nombres, le bit 0 (donc de poids faible) sera sur la **droite** de la représentation.

Les nombres seront représentés en indiquant leur *base* quand cela sera nécessaire. Si la base n'est pas indiquée, il s'agira nécessairement d'une représentation en base 10.

La base sera indiquée soit par sa valeur en indice à droite du nombre, soit en écrivant le nombre dans une représentation usuelle de langage de programmation (*i.e.* utilisant 0x ou 0b pour la base 16 ou la base 2 respectivement) soit explicitement dans le texte. Ainsi les valeurs suivantes représentent le même nombre :

- 0xfe;
- $fe_{16}$ ;
- 0b11111110;
- $11111110_2$ ;
- $254_{10}$ ;
- 254.

Lorsque l'on fera référence à une sous partie d'un nombre, on l'écrira :

- $\text{nombre}[i]$  pour le  $i^{\text{ième}}$  bit de ce nombre;
- $\text{nombre}[j : i]$  pour le nombre représenté par les bits  $i$  à  $j$  avec  $i < j$ ;

Si on écrit  $\text{partie} = \text{nombre}[12 : 10]$ , cela signifie que la valeur *partie* est représentée par les 3 bits 10, 11 et 12 de *nombre*.

On pourra ensuite étendre la notation en indiquant, par exemple,  $\text{partie}[2 : 1]$  pour signifier la valeur représentée par les bits 1 et 2 de *partie*. Dans cet exemple, on a  $\text{partie}[2 : 1] = \text{nombre}[12 : 11]$ .

Pour clarifier, considérons la valeur 32 bits 0xcafe4872. Cette valeur se représente en base deux sous la forme :

```
1100 1010 1111 1110 0100 1000 0111 0010
C    A    F    E    4    8    7    2
```

À partir de cette valeur, voici quelques exemples d'utilisation de la notation introduite précédemment :

- $\text{nombre}[3 : 0] = 0010_2 = 2_{10}$
- $\text{nombre}[11 : 4] = 10000111_2 = 87_{16} = 135_{10}$
- Si  $\text{partie} = \text{nombre}[11 : 4]$  alors  $\text{partie}[7 : 4] = 1000_2 = 8_{10}$

### 2.2.2 Forme générale et opcode

Sauf exception <sup>1</sup>, toutes les instructions sont codées sur 4 octets selon les 4 formes, ou type d'encodage, illustrées par les figures 2.1, 2.2, 2.3 et 2.4 (*c.f.* section 2.2 de la spécification) :

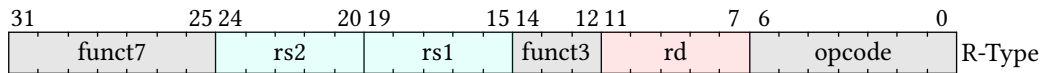


FIGURE 2.1 – Codage de type R

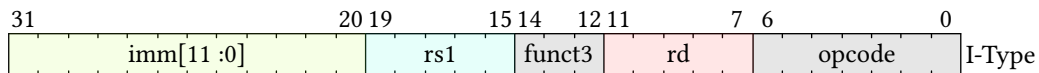


FIGURE 2.2 – Codage de type I

Dans ces figures, le mot 32 bits complet sera nommé *inst* et les nombres indiqués au dessus sont les numéros de bits. On peut également voir indiqués les noms de champs que l'on utilisera par la suite. Par exemple, dans le type R, on peut déduire de la figure correspondante que :

1. qui ne concerne pas ce projet, comme les instructions à taille variable par exemple.

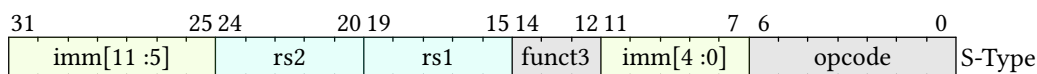


FIGURE 2.3 – Codage de type S

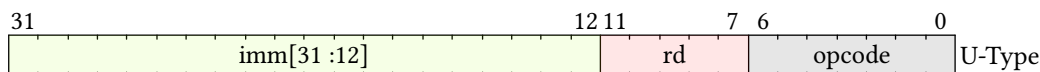


FIGURE 2.4 – Codage de type U

$$rs2 = inst[24 : 20]$$

Si le nom de champ est suivi de numéro de bit, comme dans  $imm[11 : 0]$  du type I, cela signifie que l'on parle des bits 0 à 11 du champ  $imm$ .

Ce qui nous intéresse dans un premier temps est le champ  $opcode$ , qui est une valeur sur 7 bits, représentée par les bits 0 à 6 du mot de 32 bits représentant l'instruction.

Selon la notation que nous avons introduite précédemment, on note

$$opcode = inst[6 : 0]$$

Cet  $opcode$  va nous permettre de distinguer les différents *types* d'instruction. Chaque type d'instruction utilisera l'un des 4 type de codage (R, I, S ou U). Les types d'instruction (et donc les valeurs d'opcode) que nous allons implémenter (ceux du jeu d'instruction RV32I donc) sont les suivants :

- OP-IMM, AUIPC et LUI : opération registre/valeur immédiate (section 2.4.1)
- OP : opération registre/registre (section 2.4.2)
- JAL et JALR : branchement inconditionnel (section 2.5.1)
- BRANCH : branchement (section 2.5.2)
- LOAD et STORE : lecture/écriture mémoire (section 2.6)
- MISC-MEM : action mémoire spéciale (section 2.7)
- SYSTEM : mécanisme d'exceptions (section 2.8)

Le tableau 2.1 indique, pour chaque type d'instruction, la valeur correspondante pour  $opcode$  ainsi que le type de codage employé.

TABLE 2.1 – Type d'instruction et encodage		
Type d'instruction	$opcode$	Type d'encodage
BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>
JALR	1100111 <sub>2</sub>	I
LOAD	0000011 <sub>2</sub>	I
MISC-MEM	0001111 <sub>2</sub>	I
OP-IMM	0010011 <sub>2</sub>	I
SYSTEM	1110011 <sub>2</sub>	I
JAL	1101111 <sub>2</sub>	U <sub>J</sub>
OP	0110011 <sub>2</sub>	R
STORE	0100011 <sub>2</sub>	S
AUIPC	0010111 <sub>2</sub>	U
LUI	0110111 <sub>2</sub>	U

Vous pouvez remarquer que le tableau mentionne 2 types d'encodages supplémentaires, le type U<sub>J</sub> et le type S<sub>B</sub>. Il s'agit en réalité respectivement du type U et du type S pour lesquels le champ  $imm$  s'interprète différemment (voir section 2.3). Nous reviendrons dessus le moment venu.

### 2.2.3 Registres

Dans le codage des instructions, on peut voir les champs *rs1*, *rs2* et *rd*. Ces champs désignent des numéros de registres. En observant plus en détail, on voit que ces champs sont **toujours** codés sur 5 bits. On peut donc représenter des valeurs de 0 à 31. La section 2.1 de la spécification nous apprend qu'un processeur RISC-V supportant le jeu d'instruction RV32I possède 32 registres généraux nommés *x0*, *x1*, ..., *x31*. On peut facilement déduire que le nombre codé dans les instructions correspond à un numéro de registre. Ainsi, si le champ *rs1*, *rs2* ou *rd* vaut 3, cela signifie que l'instruction utilise le registre *x3* pour le champ correspondant.

Considérons l'instruction du programme 3 qui réalise l'opération  $x3 = x5 + x8$ .

```
; syntaxe: add rd, rs1, rs2
add x3, x5, x8
```

Programme 3: Addition de registres

On aura :

- *rd* = 3
- *rs1* = 5
- *rs2* = 8

### 2.2.4 Valeurs immédiates

#### Principe général

On appelle les constantes des *valeurs immédiates* qui apparaissent comme opérande d'une instruction. Dans le programme 4, le nombre **58** est une valeur immédiate.

```
; syntaxe: addi rd, rs1, imm
addi x3, x4, 58
```

Programme 4: Exemple de valeur immédiate.

Dans le codage que l'on considère, si une instruction utilise une valeur immédiate, elle est indiquée par le champ *imm*. Dans le codage de type I, la valeur immédiate est codée sur 12 bits et on a :

$$imm[11 : 0] = inst[31 : 20].$$

L'exemple en C donné par le programme 5 montre comment extraire la valeur immédiate d'une instruction de type I.

```
int immediate_dans_I(uint32_t inst) {
    // 0xffff = 0b1111 1111 1111, 12 bits à 1
    int imm_11_0 = (inst >> 20) & 0xffff;

    return imm_11_0;
}
```

Programme 5: Extraction d'une valeur immédiate depuis un codage de type I.

Dans le codage de type S, c'est un peu plus compliqué et la valeur immédiate est toujours codée sur 12 bits, mais elle est scindée en deux parties. On a alors :

$$imm[11 : 5] = inst[31 : 25]$$

$$imm[4 : 0] = inst[7 : 11].$$

Il faut donc *combiner* ces deux parties pour former les 12 bits de la valeur immédiate. Le programme 6 vous montre comment extraire les 12 bits de la valeur immédiate dans un codage de type S.

```
int immediate_dans_S(uint32_t inst) {
    // 0x7f = 0b111 1111, 7 bits à 1
    int imm_11_5 = (inst >> 25) & 0x7f;
    // 0x1f = 0b1 1111, 5 bits à 1
    int imm_4_0 = (inst >> 7) & 0x1f;

    return imm_11_5 << 5 | imm_4_0;
}
```

Programme 6: Extraction d'une valeur immédiate depuis un codage de type S.

### Extension en 32 bits

Une fois que l'on a extrait la valeur immédiate sur  $n$  bits ( $n = 12$  dans le cas du type I), il faut construire une valeur 32 bits à partir de celle-ci. Ceci est nécessaire car le processeur RISC-V manipule uniquement des valeurs sur 32 bits.

La section 2.3 de la spécification nous explique comment réaliser cette extension pour les différents types de codage. Nous allons détailler les opérations à réaliser pour le type I. À vous de trouver comment réaliser les autres types.

La Figure 2.5 illustre comment l'extension de 12 à 32 bits est réalisée pour le type I.

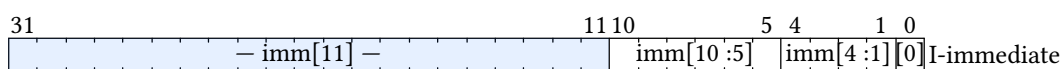


FIGURE 2.5 – Valeur immédiate calculée depuis un codage de type I

Si on note  $imm32$  la valeur immédiate étendue sur 32 bits et  $imm$  la valeur immédiate sur 12 bits extraite du codage de type I, on déduit de la figure que

- $imm32[10 : 0] = imm[10 : 0]$
- $imm32[31 : 11] = imm[11]$  : cela signifie que le bit 11 de  $imm$  est répété sur les bits 11 à 31 de  $imm32$

La raison qui motive ce codage *apparemment* complexe est simplement que la valeur immédiate étendue à 32 bits doit conserver le signe de la valeur immédiate sur 12 bits. Comme vous le savez<sup>2</sup>, les nombres négatifs sont codés en complément à deux. Le bit de poids fort d'un nombre est alors appelé *bit de signe* et permet de savoir si un nombre est positif ou négatif.

Dans la valeur immédiate sur 12 bits, le bit de signe est donc le bit 11. Si on *copie* ce bit du bit 12 au bit 31 pour construire un nombre 32 bits, on aura codé la même valeur, mais cette fois sur 32 bits et non sur 12 bits.

Par exemple l'entier -12 est codé (en complément à deux) :

- 0xff4 sur 12 bits
- 0xffffffff4 sur 32 bits

On voit bien que pour passer de l'un à l'autre, on a simplement recopié le bit 11 sur tous les bits de 11 à 31.

Le programme 7 donne un exemple permettant de réaliser cette extension.

2. en tout cas, vous devriez le savoir.

```

int32_t extension(int immediate) {
    int32_t imm32 = immediate;
    // Test du bit de signe sur 12 bits
    if ((immediate >> 11) & 1) {
        // Extension à 32 bits en mettant à 1 les 20 bits de poids fort
        return imm32 | 0xffff000;
    }
    return imm32;
}

```

Programme 7: Extension du bit de signe.

## 2.3 Livrable 1 : un premier décodeur

Vous pouvez maintenant écrire un premier programme qui doit :

1. prendre en paramètre un chemin vers un fichier binaire
2. pour chaque mot de 32 bits contenu dans le fichier, en extraire les 7 bits d'opcode et en déduire le type instruction correspondant parmi les opérations listées dans le tableau 2.1 (BRANCH, OP-IMM...)
3. de même, pour chaque mot de 32 bits, après avoir extrait l'opcode, déduire le type d'encodage utilisé par l'instruction (voir tableau 2.1)

Votre programme être un programme s'exécutant dans un terminal et respectant le format de ligne de commande suivant :

Un décodeur d'instruction RISC-V RV32I

Utilisation: `decode_riscv [OPTIONS] FICHIER_BIN`

Arguments:

`FICHIER_BIN` Un fichier au format binaire contenant les instructions à décoder

Options:

`-h` Affiche ce message d'aide

La sortie du programme doit respecter **scrupuleusement**<sup>3</sup> le format CSV suivant :

- chaque mot de 32 bits déclenchera l'affichage d'une ligne sur la sortie standard;
- cette ligne contiendra les valeurs suivantes, séparées par une **virgule**
  1. `offset` : à combien d'octets se situe le mot de 32 bits depuis le début du fichier, en hexadécimal,
  2. `valeur` : la valeur du mot de 32 bits, affiché en hexadécimal,
  3. `opcode` : le type d'opcode (BRANCH, OP-IMM... c.f. tableau 2.1),
  4. `encoding` : le type d'encodage à utiliser (I, S<sub>B</sub>, R, S, U<sub>J</sub> ou U, c.f. tableau 2.1)
- la première ligne doit contenir le nom des champs

Un exemple de sortie du programme est donné ci-dessous

```

offset,valeur,opcode,encoding
00000000,d8f56417,AUIPC,U
00000004,580d7b03,LOAD,I
00000008,9a8dca63,BRANCH,S_B
0000000c,43b7a393,OP-IMM,I
00000010,ce9ff513,OP-IMM,I

```

3. Le livrable sera testé à l'aide de scripts automatiques.



```

000000014,0b1f9113,OP-IMM,I
000000018,bacfb383,LOAD,I
00000001c,759cde03,LOAD,I
000000020,89463ee3,BRANCH,S_B
000000024,1ff49b33,OP,R
000000028,f91e1d63,BRANCH,S_B
00000002c,ec1b8ce3,BRANCH,S_B
000000030,60e7a133,OP,R
000000034,142c3fb3,OP,R
000000038,8d5288b7,LUI,U
00000003c,4b0dbb13,OP-IMM,I
000000040,d453dd13,OP-IMM,I
000000044,a0ee89b3,OP,R
000000048,9e574f23,STORE,S
00000004c,e2acf733,OP,R
000000050,dc98d293,OP-IMM,I
000000054,5c941ce7,JALR,I
000000058,93cd598f,MISC-MEM,I
00000005c,3139d333,OP,R
000000060,b45ed1b3,OP,R
000000064,11ce5db3,OP,R
000000068,0bbb25f3,SYSTEM,I
00000006c,a9488d8f,MISC-MEM,I
000000070,3a578a93,OP-IMM,I
000000074,c5e7ceb3,OP,R
000000078,4a15540f,MISC-MEM,I
00000007c,fc377a13,OP-IMM,I
000000080,146d3f33,OP,R
000000084,daf61a63,BRANCH,S_B
000000088,3b982ef3,SYSTEM,I
00000008c,ddd1dfa3,STORE,S
000000090,19db3a93,OP-IMM,I
000000094,614ff3a3,STORE,S
000000098,4729476f,JAL,U_J
00000009c,7412b283,LOAD,I
0000000a0,a2bc3763,BRANCH,S_B
0000000a4,d5884283,LOAD,I

```

## 2.4 Livrable 2 : un désassembleur complet

Fort de l'expérience du programme précédent, vous aller maintenant écrire un programme qui désassemble un fichier contenant des instructions au format binaire.

La spécification de la ligne de commande de votre programme est la suivante :

# Un désassembleur RISC-V pour le jeu d'instruction RV32I

Utilisation: `disas [OPTIONS] FICHIER_BIN`

Arguments:

FICHER\_BIN

Un fichier contenant les instructions à désassembler

Options:

## -h

Affiche ce message d'aide

la sortie devra être un affichage sur la sortie standard, contenant une instruction **complètement** décodée par ligne. Le format de la sortie devra être tel que :

- les registres seront écrits `x0, x1, ..., x31` ;
- les valeurs immédiates seront affichées en décimal (e.g. 10, -12,...);
- les opérandes d'une instructions seront séparées par une virgule.

Voici un exemple de fichier binaire représentant un programme (l'affichage correspond à la sortie de la commande `hexdump` sur le fichier binaire et affiche un mot de 32 bits par ligne, précédé de l'offset dans le fichier)

```
hexdump -e '"%08_ax: %08x\n"' samples/sample.bin
```

```
00000000: 06400093
00000004: 0c800113
00000008: 100105b7
0000000c: 00400513
00000010: 01f01013
00000014: 00100073
00000018: 40705013
0000001c: 0000006f
00000020: 00008067
```

Le résultat du désassemblage d'un tel fichier doit donner le programme 8. Sur cette sortie de programme, les valeurs immédiates sont affichées telles qu'elles seront utilisées par la processeur (c'est à dire avec extension du bit de signe et décalage éventuel, comme dans le cas de l'instruction `lui` par exemple). Pour rendre l'utilisation du désassembleur plus pratique, l'offset de chaque instruction est indiqué en début de chaque ligne et la valeur immédiate est également affichée en hexadécimal en commentaire.

```
00000000: addi          x1, x0, 100          // 0x64
00000004: addi          x2, x0, 200          // 0xc8
00000008: lui           x11, 268500992        // 0x10010000
0000000c: addi          x10, x0, 4           // 0x4
00000010: slli          x0, x0, 31           // 0x1f
00000014: ebreak        x0, x0, 1            // 0x1
00000018: srlai         x0, x0, 1031         // 0x407
0000001c: jal           0                    // 0x0
00000020: jalr          x0, x1, 0            // 0x00000000: addi x1, x0, 100 // 0x64
```

Programme 8: Désassemblage d'un fichier contenant un programme RISC-V.

Pour réussir ce désassemblage, vous allez devoir interpréter correctement tous les types d'encodage. Afin de vous aider, le tableau 2.2 récapitule les instructions et les valeurs attendues pour les différents champs des différents encodage (ces valeurs peuvent se retrouver dans le chapitre 34 de la spécification RISC-V).

La liste des instructions et leur spécification se trouvent essentiellement dans la section 2 de la spécification.

**Attention** Gardez à l'esprit que le but de la SAÉ n'est pas d'écrire un désassembleur, mais un émulateur. Essayez de penser votre code de façon à pouvoir réutiliser le code des deux livrables lorsque vous aller chercher à **exécuter** les instructions.

TABLE 2.2 – Tableau récapitulatif des formes d'encodage des instructions.

Instruction	Nom opcode	Valeur Opcode	Type encodage	funct3	funct7	funct12
BEQ	BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>	000 <sub>2</sub>		
BNE	BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>	001 <sub>2</sub>		
BLT	BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>	100 <sub>2</sub>		
BGE	BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>	101 <sub>2</sub>		
BLTU	BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>	110 <sub>2</sub>		
BGEU	BRANCH	1100011 <sub>2</sub>	S <sub>B</sub>	111 <sub>2</sub>		
LB	LOAD	0000011 <sub>2</sub>	I	000 <sub>2</sub>		
LH	LOAD	0000011 <sub>2</sub>	I	001 <sub>2</sub>		
LW	LOAD	0000011 <sub>2</sub>	I	010 <sub>2</sub>		
LBU	LOAD	0000011 <sub>2</sub>	I	100 <sub>2</sub>		
LHU	LOAD	0000011 <sub>2</sub>	I	101 <sub>2</sub>		
FENCE	MISC-MEM	0001111 <sub>2</sub>	I	000 <sub>2</sub>		
FENCE.TSO	MISC-MEM	0001111 <sub>2</sub>	I	000 <sub>2</sub>		
PAUSE	MISC-MEM	0001111 <sub>2</sub>	I	000 <sub>2</sub>		
ADDI	OP-IMM	0010011 <sub>2</sub>	I	000 <sub>2</sub>		
SLTI	OP-IMM	0010011 <sub>2</sub>	I	010 <sub>2</sub>		
SLTIU	OP-IMM	0010011 <sub>2</sub>	I	011 <sub>2</sub>		
XORI	OP-IMM	0010011 <sub>2</sub>	I	100 <sub>2</sub>		
ORI	OP-IMM	0010011 <sub>2</sub>	I	110 <sub>2</sub>		
ANDI	OP-IMM	0010011 <sub>2</sub>	I	111 <sub>2</sub>		
SLLI	OP-IMM	0010011 <sub>2</sub>	I	001 <sub>2</sub>		
SRLI	OP-IMM	0010011 <sub>2</sub>	I	101 <sub>2</sub>	0000000 <sub>2</sub>	
SRAI	OP-IMM	0010011 <sub>2</sub>	I	101 <sub>2</sub>	0100000 <sub>2</sub>	
JALR	JALR	1100111 <sub>2</sub>	I	-		
ECALL	SYSTEM	1110011 <sub>2</sub>	I	000 <sub>2</sub>		000000000000 <sub>2</sub>
EBREAK	SYSTEM	1110011 <sub>2</sub>	I	000 <sub>2</sub>		000000000001 <sub>2</sub>
JAL	JAL	1101111 <sub>2</sub>	U <sub>J</sub>	-		
ADD	OP	0110011 <sub>2</sub>	R	000 <sub>2</sub>	0000000 <sub>2</sub>	
SUB	OP	0110011 <sub>2</sub>	R	000 <sub>2</sub>	0100000 <sub>2</sub>	
SLL	OP	0110011 <sub>2</sub>	R	001 <sub>2</sub>		
SLT	OP	0110011 <sub>2</sub>	R	010 <sub>2</sub>		
SLTU	OP	0110011 <sub>2</sub>	R	011 <sub>2</sub>		
XOR	OP	0110011 <sub>2</sub>	R	100 <sub>2</sub>		
SRL	OP	0110011 <sub>2</sub>	R	101 <sub>2</sub>	0000000 <sub>2</sub>	
SRA	OP	0110011 <sub>2</sub>	R	101 <sub>2</sub>	0100000 <sub>2</sub>	
OR	OP	0110011 <sub>2</sub>	R	110 <sub>2</sub>		
AND	OP	0110011 <sub>2</sub>	R	111 <sub>2</sub>		
SB	STORE	0100011 <sub>2</sub>	S	000 <sub>2</sub>		
SH	STORE	0100011 <sub>2</sub>	S	001 <sub>2</sub>		
SW	STORE	0100011 <sub>2</sub>	S	010 <sub>2</sub>		
AUIPC	AUIPC	0010111 <sub>2</sub>	U	-		
LUI	LUI	0110111 <sub>2</sub>	U	-		



## **Chapitre 3**

**À suivre...**