SAÉ S5.A.01

Émulateur processeur RISC-V

Michaël Hauspie

26 novembre 2024

Table des matières

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.

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). Il 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 ou `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 forcement 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 forcement 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 nombres 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 [1]) 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 [2] 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 [3]

— 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 RISCV. Pour les compiler, vous devez avoir installé la suite de compilation RISCV.
  — 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 créé suite aux opération de compilation et s'assemblage. Ce sont ces fichiers `.bin` qui devrons être chargés par votre émulateur.
— `https://github.com/TheThirdOne/rars` : un IDE pour assembleur RISCV, 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 celà 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 :

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

Si on écrit $partie = nombre[12 : 10]$, 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, $partie[2 : 1]$ pour signifier la valeur représentée par les bits 1 et 2 de $partie$. Dans cet exemple, on a $partie[2 : 1] = nombre[12 : 11]$.

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 :

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

2.2.2 Forme générale et *opcode*

Sauf exception [1], 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) :

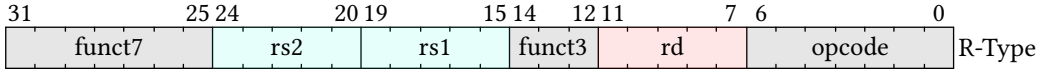

FIGURE 2.1 – Codage de type R

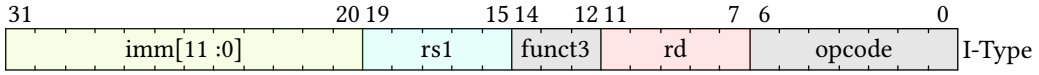

FIGURE 2.2 – Codage de type I

Dans ces figures, le mot 32 bits complet sera nommé $inst$ et les nombres indiqués au dessus sont les numéro 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.

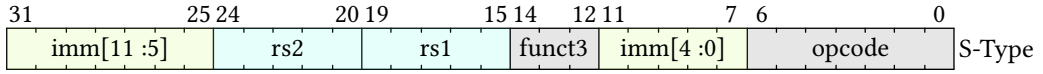

FIGURE 2.3 – Codage de type S

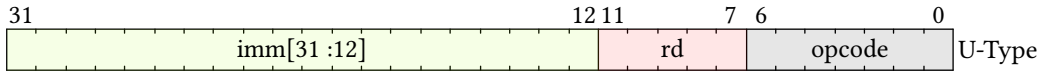

FIGURE 2.4 – Codage de type U

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

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_2	S_B
`JALR`	1100111_2	I
`LOAD`	0000011_2	I
`MISC-MEM`	0001111_2	I
`OP-IMM`	0010011_2	I
`SYSTEM`	1110011_2	I
`JAL`	1101111_2	U_J
`OP`	0110011_2	R
`STORE`	0100011_2	S
`AUIPC`	0010111_2	U
`LUI`	0110111_2	U

Vous pouvez remarquer que le tableau mentionne 2 types d'encodages supplémentaires, le type U_J et le type S_B. 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 apprends 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) {
    // 0xfff = 0b1111 1111 1111, 12 bits à 1
    int imm_11_0 = (inst >> 20) & 0xfff;

    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.

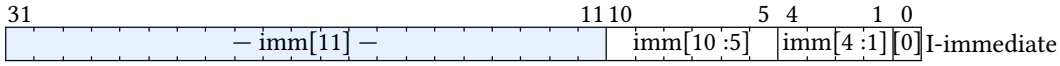

FIGURE 2.5 – Valeur immédiate calculée depuis un codage de type I

Si on note *imm*32 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 *imm*32

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 [2], 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

— `0xfffffff4` 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.

```c
int32_t extension(int32_t 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 immediate | 0xfffff000;
    }
    return immediate;
}
```

Programme 7: Extension du bit de signe.

Les types S_B et U_J

Pour ces deux types, le décodage est globalement le même que pour les types S et U mais l'interprétation de la valeur immédiate est différente. Le détail est donné dans la section 2.3 de la spécification où ces deux types sont nommés B et J, respectivement. L'encodage précis de la valeur immédiate dans ces deux cas est donnée Figures 2.6 et 2.7.

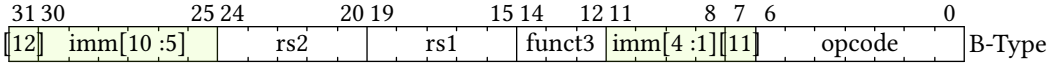

FIGURE 2.6 – Valeur immédiate calculée depuis un codage de type S_B (B).

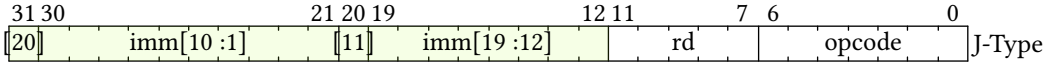

FIGURE 2.7 – Valeur immédiate calculée depuis un codage de type U_J (J).

Par exemple, pour le type B, l'instruction contient les bits 1 à 12 de la valeur immédiate finale. Le bit 0 sera toujours à 0. Ceci est du au fait que les instructions utilisant cet encodage sont des instructions de branchement. La valeur immédiate est donc utilisée pour le calcul de l'adresse qui sera stockée dans le registre pc à la fin de l'exécution de l'instruction. Or, dans le jeu d'instruction RISCV, un processeur ne peut pas exécuter d'intruction située à une adresse impaire. Il est donc inutile de coder le bit 0 car on souhaite que la valeur immédiate soit toujours paire. On gagne alors 1 bit permettant de coder des valeurs plus grande et donc d'effectuer des branchements vers des instructions plus éloignées dans le code du programme.

Le programme 8 donne un exemple de code C qui extrait correctement la valeur immédiate d'une instruction encodée selon le type S_B (B)

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
```

```c
int32_t imm_dans_b(uint32_t icode) {

    // On extrait les différentes parties
    int32_t imm_11  = (icode >> 7) & 0x1;
    int32_t imm_1_4 = (icode >> 8) & 0xf;
    int32_t imm_5_10 = (icode >> 25) & 0x3f;
    int32_t imm_12 = (icode >> 31) & 0x1;

    // On recolle les morceaux
    int32_t imm = imm_1_4 << 1 | imm_5_10 << 5 | imm_11 << 11 | imm_12 << 12;

    // On étend le bit de signe
    return extension_bit12(imm_12);
}
```

Programme 8: Extraction de la valeur immédiate depuis un encodage de type S_B (B)

```
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**[3] 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 hexadecimal,
  2. valeur : la valeur du mot de 32 bits, affiché en hexadecimal,
  3. opcode : le type d'opcode (BRANCH, OP-IMM… *c.f.* tableau 2.1),
  4. encoding : le type d'encodage à utiliser (I, S_B, R, S, U_J 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,d8f56403,LOAD,I
00000004,580d7b63,BRANCH,S_B
00000008,9a8dca13,OP-IMM,I
0000000c,43b7a393,OP-IMM,I
00000010,ce9ff513,OP-IMM,I
00000014,0b1f9103,LOAD,I
00000018,bacfb383,LOAD,I
0000001c,759cde63,BRANCH,S_B
00000020,89463eb3,OP,R
00000024,1ff49b17,AUIPC,U
00000028,f91e1d63,BRANCH,S_B
0000002c,ec1b8ce3,BRANCH,S_B
00000030,60e7a133,OP,R
00000034,142c3fb7,LUI,U
```

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

```
00000038,8d528893,OP-IMM,I
0000003c,4b0dbb13,OP-IMM,I
00000040,d453dd33,OP,R
00000044,a0ee89a3,STORE,S
00000048,9e574f33,OP,R
0000004c,e2acf713,OP-IMM,I
00000050,dc98d2f3,SYSTEM,I
00000054,5c941c93,OP-IMM,I
00000058,93cd59b3,OP,R
0000005c,3139d323,STORE,S
00000060,b45ed1b3,OP,R
00000064,11ce5d93,OP-IMM,I
00000068,0bbb2593,OP-IMM,I
0000006c,a9488df3,SYSTEM,I
00000070,3a578ab3,OP,R
00000074,c5e7cee7,JALR,I
00000078,4a155433,OP,R
0000007c,fc377a63,BRANCH,S_B
00000080,146d3f0f,MISC-MEM,I
00000084,daf61a33,OP,R
00000088,3b982eef,JAL,U_J
0000008c,ddd1dfb3,OP,R
00000090,19db3aa3,STORE,S
00000094,614ff383,LOAD,I
00000098,47294763,BRANCH,S_B
0000009c,7412b283,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:
  FICHIER_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: 010005b7
0000000c: 00400513
00000010: 01f01013
00000014: 00100073
00000018: 40705013
0000001c: 0000006f
00000020: 00008067
```

Le résultat du désassemblage d'un tel fichier doit donner le programme 9. Sur cette sortie de programme, les valeurs immédiates sont affichées telles qu'elle seront utilisées par la processeur (c'est à dire avec extention 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, 16777216        // 0x1000000
0000000c: addi        x10, x0, 4        // 0x4
00000010: slli        x0, x0, 31        // 0x1f
00000014: ebreak
00000018: srai        x0, x0, 1031        // 0x407
0000001c: jal         x0, 0                // 0x0
00000020: jalr        x0, x1, 0        // 0x00
```

Programme 9: 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_2	S_B	000_2		
BNE	BRANCH	1100011_2	S_B	001_2		
BLT	BRANCH	1100011_2	S_B	100_2		
BGE	BRANCH	1100011_2	S_B	101_2		
BLTU	BRANCH	1100011_2	S_B	110_2		
BGEU	BRANCH	1100011_2	S_B	111_2		
LB	LOAD	0000011_2	I	000_2		
LH	LOAD	0000011_2	I	001_2		
LW	LOAD	0000011_2	I	010_2		
LBU	LOAD	0000011_2	I	100_2		
LHU	LOAD	0000011_2	I	101_2		
FENCE	MISC-MEM	0001111_2	I	000_2		
ADDI	OP-IMM	0010011_2	I	000_2		
SLTI	OP-IMM	0010011_2	I	010_2		
SLTIU	OP-IMM	0010011_2	I	011_2		
XORI	OP-IMM	0010011_2	I	100_2		
ORI	OP-IMM	0010011_2	I	110_2		
ANDI	OP-IMM	0010011_2	I	111_2		
SLLI	OP-IMM	0010011_2	I	001_2		
SRLI	OP-IMM	0010011_2	I	101_2	0000000_2	
SRAI	OP-IMM	0010011_2	I	101_2	0100000_2	
JALR	JALR	1100111_2	I	-		
ECALL	SYSTEM	1110011_2	I	000_2		000000000000_2
EBREAK	SYSTEM	1110011_2	I	000_2		000000000001_2
JAL	JAL	1101111_2	U_J	-		
ADD	OP	0110011_2	R	000_2	0000000_2	
SUB	OP	0110011_2	R	000_2	0100000_2	
SLL	OP	0110011_2	R	001_2		
SLT	OP	0110011_2	R	010_2		
SLTU	OP	0110011_2	R	011_2		
XOR	OP	0110011_2	R	100_2		
SRL	OP	0110011_2	R	101_2	0000000_2	
SRA	OP	0110011_2	R	101_2	0100000_2	
OR	OP	0110011_2	R	110_2		
AND	OP	0110011_2	R	111_2		
SB	STORE	0100011_2	S	000_2		
SH	STORE	0100011_2	S	001_2		
SW	STORE	0100011_2	S	010_2		
AUIPC	AUIPC	0010111_2	U	-		
LUI	LUI	0110111_2	U	-		

Chapitre 3

Vers un vrai émulateur : modélisation du processeur et de la mémoire

Maintenant que vous savez décoder des instructions, il ne reste plus qu'à les exécuter. Pour cela, nous devons modéliser l'architecture matérielle que nous allons émuler.

Le modèle que nous allons développer doit être capable de maintenir et de faire évoluer l'état du processeur, de la mémoire et des périphériques.

3.1 Le processeur

Concernant le processeur, la spécification, section 2.1, nous indique ce qui constitue l'état du processeur pour le jeu d'instruction non privilégié RV32I :

— 32 registres de 32 bits, de `x0` à `x31` ;
— 1 registre `pc`, de 32 bits également, pour le compteur de programme.

On nous apprends aussi que le registre `x0` a un comportement particulier : il vaut toujours 0. Ainsi

— lire ce registre retourne toujours 0
— écrire dans ce registre ne change **jamais** l'état du processeur

Ces propriétés permettent de simplifier le jeu d'instruction. Par exemple, pas besoin d'avoir spécifiquement une instruction ne fait rien (comme l'instruction `nop` existant dans la plupart des architectures). Il suffit de faire un calcul avec `x0` comme destination pour faire une instruction qui ne fait rien.

Si on devait proposer une structure de données en C pour stocker l'état du processeur, on pourrait donc utiliser une structure similaire à celle du programme 10.

```
struct riscv_cpu {
    uint32_t regs[32];
    uint32_t pc;
};
```

Programme 10: Un exemple de modélisation en C d'un processeur RISCV supportant RV32I

Avec un tel modèle, l'exécution de l'instruction `add x1, x2, x3` s'écrirait tout simplement [1] :

```
void add_x1_x2_x3(struct riscv_cpu *state) {
    state->regs[1] = state->regs[2] + state->regs[3];
}
```

1. attention, dans votre code il y aurait probablement plus de vérification à effectuer

3.2 La mémoire

La mémoire de notre architecture peut être simplement un tableau d'octets. Une adresse utilisée serait alors simplement l'indice dans ce tableau. En C, on pourrait déclarer une mémoire de 512 KB comme indiqué dans le programme 11

```
char memoire[512 * 1024];
```

Programme 11: une mémoire de 512 KB

En utilisant les deux modèles simples proposés, l'instruction `lb x1, 100` pourrait s'écrire (voir la spécification section 2.6 pour l'explication du calcul d'adresse) :

```
void lb_x1_100(struct riscv_cpu *state, char *memoire) {
    int adresse = 100;
    state->regs[1] = (uint32_t) memoire[adresse];
}
```

Programme 12: Un exemple simpliste d'implémentation de l'instruction lb x1, 100

Attention les modèles proposés ici (en particulier celui de la mémoire) sont simplistes. Pensez que vous aurez de nombreuses vérifications à effectuer qui nécessiterons probablement de revoir les modèles. En particulier, les adresses sont sur 32 bits mais votre mémoire pourra faire moins de 4 GB. Certaines adresses seraient donc invalides ce qui entraînerait une erreur de segmentation dans l'implémentation ne vérifiant pas les adresses.

La boucle principale d'émulation pourrait ressembler à :

```
void emu_loop(struct riscv_cpu *state, char *memoire) {
    // boucle infinie
    for (;;) {
        uint32_t *instruction_ptr = (uint32_t *) &memoire[state->pc];
        uin32_t instruction = *instruction_ptr;
        // en fonction de l'instruction, execute mets à jour state->pc pour pointer vers
        // l'instruction suivante
        execute(instruction, state, memoire);
    }
}
```

Programme 13: Exemple de boucle d'émulation

3.3 Livrable 3 : un émulateur sans entrées/sortie

Vous avez maintenant toutes les clés en main pour implémenter une première version de votre émulateur

1. vous savez décoder des instructions ;
2. vous savez modéliser l'état du processeur et de la mémoire.

Vous devez maintenant écrire un programme qui :

1. crée un processeur et une mémoire ;
2. charge un fichier contenant une version binaire (assemblée) d'un programme au début de la mémoire ;

3. initialise `pc` à l'adresse contenant la première instruction à exécuter ;
4. démarre l'exécution (effectue une boucle d'émulation comme celle du programme 13).

Pour pouvoir utiliser les exemples fournis dans `https://gitlab.univ-lille.fr/michael.hauspie/riscv-samples` votre émulateur doit permettre

1. de choisir une taille de mémoire (avec une taille par défaut à 512 KB) sur la ligne de commande ;
2. de choisir l'adresse de *reset*, c'est à dire la valeur initiale du registre `pc` sur la ligne de commande :
   (a) pour les exemples assembleurs, il faudra utiliser `0`,
   (b) pour les exemples C, il faudra utiliser `0x100` ;
3. de permettre une exécution *pas à pas*, c'est à dire instruction par instruction, en indiquant avant l'exécution de chaque instruction :
   — la valeur de chacun des 32 registres,
   — la valeur du registre `pc` ainsi que le désassemblage de l'instruction située à l'adresse contenue dans ce dernier ;
4. d'exécuter toutes les instructions du tableau 2.2. Pour les instructions `fence*`, `pause`, `ebreak`, `ecall` le comportement doit être le suivant :
   — `fence*`, `pause` et `ecall` : l'instruction ne fait rien et on passe à la suivante,
   — `ebreak` : bascule l'émulateur du mode *exécution continue* au mode *pas à pas* (voir ci-après).

En mode *pas à pas*, avant d'exécuter l'instruction suivante, l'émulateur doit attendre une commande. Vous devez supporter au moins les commandes :

— `step` : exécute l'instruction suivante ;
— `x/COUNT ADDRESS` : qui examine (affiche) `COUNT` octets, en hexadécimal, à partir de l'adresse `ADDRESS` ;
— `reset` : redémarre l'exécution du programme au début (*i.e.* affecte `pc` à la valeur de reset) ;
— `continue` : quitte le mode pas à pas et exécute la boucle d'émulation sans s'arrêter. Le retour en mode pas à pas s'effectue si l'instruction `ebreak` est exécutée ou si une erreur se produit ;
— `exit` : quitte l'émulateur.

Ces commandes vous permettront de facilement comprendre ce qu'est en train d'exécuter votre processeur.

Attention en aucun cas votre émulateur ne doit cesser son exécution de manière non prévue. En particulier, il ne doit pas effectuer d'erreur de segmentation ou équivalent. Si le processeur effectue une opération impossible comme :

— exécuter une instruction qui n'existe pas (erreur de décodage)
— accéder à une adresse qui n'existe pas (en dehors de la mémoire que vous avez définie)

l'émulateur doit afficher un message d'erreur cohérent et revenir en mode *pas à pas* s'il ne l'était plus.

3.4 Tester l'émulateur

N'ayant pas de périphérique, pour tester l'émulateur vous devez observer le comportement des instructions une à une en vérifiant que les registres et la mémoire évoluent correctement.

Une fois toutes les instructions implémentées, vous pouvez utiliser les exemples `crc` et `md5` [2] pour exécuter un programme qui utilise quasiment toutes les instructions.

Dans ces deux exemples, on calcule des fonctions de hachages sur des données connues. À la fin, le programme copie le résultat de la fonction de hachage à l'adresse `0x10000` et execute l'instruction `ebreak` pour arrêter l'émulateur. Il suffit donc d'utiliser la commande `x/4 0x10000` (pour le crc) ou `x/16 0x10000` (pour md5) pour regarder le résultat calculé. Si le résultat est bon, vous pouvez commencer à avoir une confiance relative dans votre implémentation.

— pour `crc`, 36 instructions sont utilisées et les 4 octets calculés sont (en hexadécimal) : `29 af 52 47`
— pour `md5`, 38 instructions sont utilisées les 16 octets calculés sont (en hexadécimal) : `3e 21 bd 0c 3c 5b ab 93 27 c9 e8 a2 69 a3 29 70`

2. dans le dépôt `https://gitlab.univ-lille.fr/michael.hauspie/riscv-samples/`

Chapitre 4

Livrable 4 : Interactions avec le monde extérieur

Pour que notre émulateur puisse être vraiment utilisé, il nous manque la possibilité d'interagir avec l'extérieur. Nous allons mettre en œuvre deux solutions permettant cette interaction. l'extérieur :

— la simulation de périphériques d'entrée/sortie ;
— l'utilisation du *semihosting*.

Dans un premier temps, nous allons nous intéresser aux périphériques.

4.1 Des périphérique de sorties

Le périphérique le plus simple que nous pouvons mettre en œuvre est un équivalent simplifié d'un port série, dans notre cas, un périphérique qui permettra à votre architecture émulée d'envoyer un octet à l'émulateur qui l'affichera sur le terminal. L'accès à un tel périphérique nous permettra de réaliser enfin l'indispensable *« Hello world »*.

4.1.1 Modèles d'accès aux périphériques

Les deux modèles les plus courants pour l'accès aux périphériques sont :

— l'utilisation d'un bus d'entrées/sorties basé sur des *ports* (*Port-based IO*) : l'architecture la plus connue utilisant ce mode est l'architecture intel avec ses instructions `in` et `out` ;
— la mise à disposition des périphériques dans l'espace d'adressage du processeur (*Memory-mapped IO*) : ce mode est privilégié sur la quasi totalité des architectures, y compris sur architecture Intel `x86-64`.

Nous allons utiliser le second modèle pour notre émulateur. Dans ce second modèle, une partie de l'espace d'adressage est utilisée pour associer des périphériques. Ainsi, accéder à une adresse associée à un périphérique permettra au processeur de lui donner des instructions et/ou de lire ou écrire des données à recevoir/envoyer. Ces adresses particulières ne correspondront pas à de la mémoire à proprement parler mais à des *registres* de configuration ou de données des périphériques.

4.1.2 Mise en œuvre des périphériques

Nous allons équiper notre architecture de deux périphériques permettant au processeur d'émettre de l'information :

— un périphérique permettant d'envoyer des octets sur la *sortie standard* de l'émulateur ;
— un périphérique permettant d'envoyer des octets sur la *sortie d'erreur* de l'émulateur.

Le principe de fonctionnement de ces deux périphériques sera équivalent : lorsque le processeur écrira une valeur à une adresse spécifique, l'émulateur écrira cette valeur sur sa sortie standard ou sa sortie d'erreur, réciproquement.

Ainsi :
— écrire une valeur (8, 16 ou 32 bits) à l'adresse `0x4000004` devra déclencher l'écriture de l'octet de poids faible de ce mot sur la sortie standard de l'émulateur ;
— écrire une valeur (8, 16 ou 32 bits) à l'adresse `0x4000008` devra déclencher l'écriture de l'octet de poids faible de ce mot sur la sortie d'erreur de l'émulateur ;
— lire une valeur (8, 16 ou 32 bits) à l'une de ces adresses doit retourner 0.

L'exemple `C/test-serial` contient un programme affichant :
— `Hello world` sur la sortie standard ;
— `Hello error world` sur la sortie d'erreur.

4.2 Un périphérique d'entrée

Dans le même esprit, ajoutez un périphérique qui, lorsque l'adresse `0x40000000` est lue, effectue la lecture bloquante d'un octet depuis l'entrée standard (par exemple en utilisant `getchar()` en C) et sa valeur utilisée comme résultat de la lecture en mémoire. Bien évidemment, si le processeur lit 32 bits, les trois octets de poids fort seront positionnés à zéro. Pour l'instant, aucun programme d'exemple n'est disponible pour tester cette fonctionnalité. Inspirez vous du programme `C/test-serial` pour développer votre propre test.

4.3 Le *semihosting*

Le *semihosting* est un mécanisme introduit par ARM pour permettre à un système embarqué d'interagir avec un hôte par l'intermédiaire d'un debugger. Ce principe peut être étendu à un émulateur et permettre au système émulé d'interagir simplement avec le système hôte.

Le *semihosting* offre un moyen simple, pour le programme cible de faire exécuter des actions ou d'échanger de l'information avec le système hôte. Par exemple, grâce au *semihosting*, le logiciel s'exécutant dans le système embarqué ou dans l'émulateur peut faire des affichages, accéder à des fichiers ou mesurer du temps.

Les opérations possibles via l'utilisation de *semihosting* sont définies par ARM dans leur spécification du *semihosting*. Sur ARM, le déclenchement d'une opération de *semihosting* passe par l'utilisation des instructions d'exception logicielle (`trap`, `svc`, `bkpt`…). Suite à l'exécution d'une de ces instructions utilisant une valeur particulière d'opérande, le debugger connecté (ou l'émulateur dans notre cas) décide d'exécuter une opération de *semihosting*.

Le consortium RISCV a décidé de reprendre la spécification ARM et la plupart des émulateurs RISCV ou des debugger utilisables avec une plateforme RISCV (comme OpenOCD par exemple) supportent tout ou partie des opérations de *semihosting* définies par ARM. Pour supporter le *semihosting* dans l'émulateur, nous devons donc connaître deux choses :

1. comment savoir quand le logiciel invité veut déclencher une opération de *semihosting* ?
2. comment savoir quelle opération il veut déclencher et où aller récupérer les paramètres de cette opération ?

La description de l'instruction `ebreak` dans la section 2.8 de la spécification RISCV nous donne une réponse à la première question.

La spécification indique que le déclenchement d'une opération de *semihosting* se fait lorsque la séquence d'instructions suivante est rencontrée :

```
slli x0, x0, 0x1f   // Entry NOP
ebreak              // Break to debugger
srai x0, x0, 7      // NOP encoding the semihosting call number 7
```

Les deux instructions encadrant l'instruction `ebreak` sont équivalentes à un `nop`, c'est à dire une instruction qui ne fait rien. On pourra donc les exécuter sans problème.

Le commentaire associé à la troisième instruction est cependant faux dans la spécification. Il laisse supposer que la valeur immédiate de la troisième instruction code le numéro d'opération à effectuer. Or, le consortium RISCV publie également des spécifications non liées au jeu d'instructions, notamment une spécification du *semihosting*, disponible à cette adresse. On y apprend que cette troisième instruction, qui utilise `7` comme valeur immédiate, ne code pas le numéro d'opération mais fait partie de la séquence qui déclenche le *semihosting*.

Cette même spécification nous donne la réponse à notre deuxième question :

— l'opération à réaliser est codée dans le registre `a0` ;
— son paramètre est donné dans le registre `a1` ;
— sa valeur de retour est donnée dans le registre `a0` le cas échéant.

Le nom des registres `a0` et `a1` proviennent de l'ABI (*Application Binary Interface*) standard de RISCV (voir la spécification de l'assembleur RISCV à cette adresse). On y apprend qu'`a0` est en fait le registre `x10` et qu'`a1` est le registre `x11`.

4.3.1 Mise en œuvre

Vous allez maintenant devoir supporter quelques opérations de *semihosting* dans votre émulateur. Il faudra donc :

1. détecter l'exécution d'un `ebreak` (ce que vous faites déjà normalement) ;
2. vérifier que l'instruction qui précède est bien `slli x0, x0, 0x1f` et que l'instruction qui suit est bien `srai x0, x0, 7` ;
3. si c'est le cas, exécuter les opérations nécessaires en fonction des valeurs des registres `x10` et `x11`.

Les opérations minimales que vous devrez supporter sont :

— `SYS_WRITEC` : écriture d'un octet sur la sortie standard de l'émulateur ;
— `SYS_WRITE0` : écriture d'une chaîne de caractères `C` (c'est à dire terminée par un `'\0'`) sur la sortie standard.

Les valeurs correspondantes à ces opérations (et donc la valeur du registre `a0`) sont indiquées dans la spécification ARM du *semihosting*.

L'exemple `C/hello_world` utilise le *semihosting* pour afficher :

```
Hello world
0xcafedeca
Done
```

Vous pouvez implémenter d'autre appels *semihosting* si vous le souhaitez.

Chapitre 5

Format de rendu des livrables

Afin d'assurer la bonne reproductibilité de vos livrables, il vous est demandé de construire des images docker permettant d'exécuter vos livrables. Ainsi, vous devez ajouter à votre dépôt git trois fichiers :

- `Dockerfile.l1`
- `Dockerfile.l2`
- `Dockerfile.l3`
- `Dockerfile.l4`

Chacun de ces fichiers doit permettre de construire une image dont le point d'entrée correspond à l'exécutable du livrable. Chaque image doit pouvoir être construite, à partir des sources, en exécutant la commande suivante depuis la racine du dépôt git :

```
docker build -t livrable1 -f Dockerfile.lX .
```

ou `Dockerfile.lX` correspond à l'un des quatres fichiers. Ensuite, le livrable doit pouvoir être exécuté via un `docker run` [1]. Par exemple :

```
$ docker run -it --rm rivemul -h
Rivemul is a terminal simple emulator of the RISC-V RV32I architecture

Usage: rivemul [OPTIONS] [MEMORY_SEGMENT]...

Arguments:
  [MEMORY_SEGMENT]...  The files to load into memory. Format is file:base_addr.  If :base_addr is no

Options:
  -r, --reset-addr <RESET_ADDR>  The reset address [default: 0x0]
  -i, --interactive              Start in Interactive (debug) mode
  -t, --trace-file <TRACE_FILE>  Generate a trace file
  -m, --mem-size <MEM_SIZE>      Memory size (in KB). This memory size will be split in 2 memory ban
  -h, --help                     Print help (see more with '--help')
  -V, --version                  Print version
```

1. N'oubliez pas que si vous voulez que le processus de votre livrable puisse accéder à un fichier d'exemple, celui ci doit être accessible depuis l'intérieur du conteneur, soit car vous l'aurez placé à la construction de l'image soit par un *bind mount*.

Chapitre 6

Extensions

Vous êtes arrivé.e jusqu'ici ? Bravo. Vous avez maintenant le choix d'améliorer votre projet en y ajoutant des extensions et des améliorations parmi les propositions suivantes (non exhaustives) :

- support d'extensions supplémentaires, en particulier les extensions `M` et `F`
- ajout d'une interface graphique
- ajout de périphériques supplémentaires
  - Ports GPIO (en lien avec une interface graphique qui simulerait l'allumage de LED sur certaines *pins* et des interrupteurs sur d'autres)
  - Carte graphique
  - Carte son
  - …

Toutes ces extensions sont facultatives, à faire selon votre envie et vous apporteront un bonus sur la note finale mais ne sont pas nécessaires pour obtenir la note maximale.