lab 2
Simulation d'un RISC-V

M. Briday

1 Objectif

Ce TP utilise un simulateur développé avec Logisim. Ce simulateur (partiel) contient déjà une grosse partie du modèle d'un RISC-V RV32I. L'objectif de ce TP est de comprendre de fonctionnement du flot de contrôle du décodage vers les différents multiplexeurs du système.

Dans un premier temps, une application sera développée en C, puis compilée et simulée sur le modèle.

Dans un deuxième temps, le modèle sera étendu fonctionnellement (ajout d'une instruction) et temporellement (ajout d'un pipeline).

2 Pré-requis

2.1 Installation de Logisim Evolution

La version originale de Logisim n'est plus maintenue, et plusieurs *forks* sont apparus. Nous utiliserons *Logisim Evolution* [1]. On peut trouver une version pré-compilée pour GNU/Linux, MacOS ou Windows sur `https://github.com/logisim-evolution/logisim-evolution/releases`.

2.2 Fonctionnement de base

Logisim est un simulateur de systèmes logiques. L'aide de base pour les différents éléments est bien documentée dans `Help -> User's guide`.

Une fois le fichier de base ouvert, 4 niveaux de description sont définis :

- `alu` est le modèle de l'ALU pour le flot de donnée. Cette partie n'a pas à être modifiée dans un premier temps.
- `registers` définis les 32 registres GPR de l'architecture. Cette partie n'a pas à être modifiée.
- `CPU` est la partie principale ici. Elle contient une instance des bancs de registres et de l'alu. C'est l'organisation complète du fonctionnement interne.

1. `https://github.com/logisim-evolution/logisim-evolution`

— `main` contient une instance du CPU, ainsi que de la mémoire. La ROM contient un programme (qu'on modifiera par la suite)

Pour lancer une simulation, il faut d'abord cliquer sur `'simulate'`, voir figure 1.

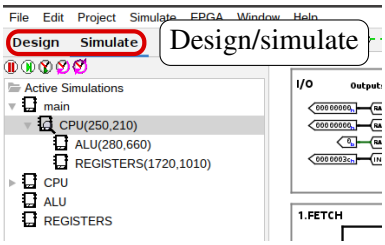

FIGURE 1 – Choix entre conception et simulation

On peut avancer cycle par cycle en appuyant sur `F9`, voir menu `Simulate`. Il est aisé de voir l'évolution des signaux en fonction du temps, avec l'outil `Simulate -> Timing diagram`. Il est recommandé d'utiliser l'hexadécimal plutôt que le binaire directement !

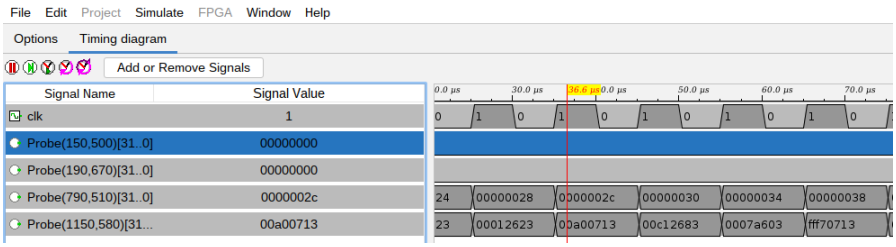

FIGURE 2 – Timing diagram

2.3 Mettre à jour le programme

Le code programme est enregistré dans la partie `ROM` du modèle (partie `main`). Le programme source est en C, et plusieurs étapes sont nécessaires :

- le cross-compilateur compile le programme C pour l'architecture RISC-V RV32I
- l'éditeur de liens (*linker*) va utiliser le mapping mémoire (fichier `linkScript.ld`) pour générer le fichier binaire (format .elf)
- l'outil `objcopy` permet de changer le format de sortie, pour un format `Intel H86` qui donne uniquement les adresses mémoires ainsi que leur contenu
- enfin le script python va transformer ce fichier standard `hex` vers le format d'entrée de logisim (fichier `.rom`).

2.3.1 Outils

Les logiciels suivants sont nécessaires :

le cross-compilateur : la *toolchain* est disponible ici : `https://github.com/riscv-collab/riscv-gnu-toolchain`. L'installation est bien documentée pour GNU/Linux et MacOS. C'est la version `Newlib` qui nous intéresse (développement *baremetal*), et pas la version Linux... (on ne souhaite pas simuler le démarrage de Linux dans Logisim !). Il existe des versions précompilées pour Windows (`https://gnutoolchains.com/risc-v/`), non testées.

Le système de construction `cmake` simplifie la construction de l'ensemble (voir section suivante). *cmake* est un système qui gère les dépendances, mais sous-traite l'ordonnancement des tâches à un outil tiers (Make, Ninja, ...). Sous Linux ou Mac, Make est installé par défaut avec les outils de développements. Sous Windows, l'installation de Ninja est plus simple.

Python est requis pour transformer le fichier binaire au format logisim (extension `.log`)

2.3.2 Mise en œuvre

Toute cette procédure est automatisée en utilisant l'infrastructure `cmake`. Dans le dossier `lab2/code`

- faire un dossier build : `mkdir build`
- aller dans ce dossier : `cd build`
- lancer cmake en précisant le dossier des source : `cmake ..`

```
-> % cmake ..
-- The C compiler identification is GNU 11.1.0
-- Configuring done
-- Generating done
-- Build files have been written to: /home/[..]/build
```

- lancer la compilation : `make`

```
-> % make
[ 16%] Building C object CMakeFiles/lab2.elf.dir/main.c.obj
[ 33%] Building C object CMakeFiles/lab2.elf.dir/home/[..]/sys/startup.c.obj
[ 50%] Linking C executable lab2.elf
[ 50%] Built target lab2.elf
[ 66%] Generate Intel Hex file
[ 83%] Generate Logisim dump file
[ 83%] Built target logisim
[100%] Extract asm dump file
[100%] Built target asm
```

ainsi, les fichiers générés sont :

```
-> % ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  lab2.elf  lab2.hex  lab2.log
lab2.map  Makefile
```

On retrouve, les binaires (`lab2.elf`, `lab2.hex`, `lab2.log`) aux formats respectivement elf, Intel H86 et Logisim. Il y a aussi un fichier `lab2.map` qui donne le mapping mémoire.

On peut alors mettre à jour le programme dans logisim (manuellement...) avec un clic droit sur la `ROM -> Load Images...`.

3 TP

3.1 Démarrage

La partie de décodage est partiellement faite. Elle consiste d'une part à extraire les opérandes (`rs1`, `rs2`, `imm_value`, ...) et d'autre part à trouver quelle est l'instruction est cours de décodage. La sortie de cette première ROM est l'id de l'instruction, comme définie dans le tableau ici : `https://gitlab.univ-nantes.fr/briday-m/itii-aim-lab-etu/-/blob/master/lab2/instructions.md`.

- ▷ connaissant l'`id` de l'instruction et ses opérandes, mettre au point la ROM de décodage pour définir les signaux de contrôles (hors branchement)
- ▷ tester le modèle avec plusieurs programmes (hors branchement, et appel de routine)
- ▷ définissez le signal `branchTaken`. Ce signal utilisera notamment le résultat de l'ALU (signal `NZCV`) et la nature du branchement (encodé dans la partie `func3`).
- ▷ tester avec un programme avec branchements
- ▷ rajouter les instructions d'appel de routine (`jal`, `jalr`).

Note importante : Cette étape nécessite d'être très rigoureux et de tester de manière approfondie chaque instruction. Toute erreur pourra être très pénalisante dans la suite du TP.

Une approche efficace peut être de faire une *double vérification* des signaux de chaque instruction : d'une part avec une analyse signal par signal, et d'autre part, instruction par instruction.

Pour tester une instruction particulière, il est possible de rajouter directement de l'assembleur en ligne dans le code, ou de rajouter un fichier assembleur directement.

3.2 Une LED ! ?

Actuellement, RAM et ROM sont utilisés respectivement pour les données et les instructions. On souhaite rajouter un périphérique de sortie (GPIO) simple avec 2 registres : `DIRA` permet de choisir si la sortie en valide (`1`) ou non (`0`), `PORTA` correspond à l'état de broche de la sortie, suivant la table de vérité 1.

$\texttt{DIRA}_x$	$\texttt{PORTA}_x$	broche x
0	0	Z
0	1	Z
1	0	0
1	1	1

TABLE 1 – Port de sortie GPIO pour la broche x

Les 2 registres `DIRA` et `PORTA` sont mappés en mémoire respoectivement en `0x20000000` et `0x20000004`. Dans le code pour y accéder, vous pouvez soit déclarer cette zone directement dans le *link script* (dans `sys/linkScript.ld`), ou *via* des pointeurs dont la valeurs est définie en dur.

Pour utiliser le périphérique, il est nécessaire d'insérer un décodage d'adresse, pour orienter les accès soit vers la RAM, soit vers le périphérique, en fonction de l'adresse. Vous pouvez vous inspirer du décodage d'adresse pour la RAM.

Attention : actuellement, la RAM est directement sur le signal `RAM_READ_DATA`, il faut supprimer tout conflit en lecture.

Il faut ainsi :

- ▷ insérer un décodage d'adresse en amont du bloc de RAM (écriture), et en aval (lecture)
- ▷ ajouter le périphérique basique avec les 2 registres ;
- ▷ connecter une led à une des broches (pas la 0) de sortie (voir `input/output -> Led` dans Logisim).
- ▷ valider avec un programme permettant de faire clignoter la led, voire même un petit chenillard...

3.3 Ajout d'une instruction

L'architecture RISC-V a été conçue pour permettre de rajouter des instructions pour des application spécifiques. On propose de rajouter une instruction d'addition avec saturate : `adds rd, rs1, rs2`, du type (décodage) `R-type`, dont la sméantique est : $R_d \leftarrow Max(2^{31}-1, rs_1 + rs_2)$.

Il est possible de rajouter une instruction au compilateur. Dans ce cas, il faut indiquer le codage associé, ainsi que la sémantique de l'instruction (pour que le compilateur sache comment utiliser cette instruction). Cette procédure est relativement bien décrite [2], mais suppose de recompiler `gcc`. La procédure est relativement lourde et s'éloigne de l'objet de ce cours. Ici, nous allons simplifier en ajoutant directement le code hexadécimal de l'instruction (à la place de `0xAAAAAAAA`), en utilisant l'ABI.

```
// la fonction ne doit pas etre 'inline',
//car les registres utilises sont fixes.
__attribute__((noinline))
int addSaturate(int a, int b) {
    asm __volatile__ (".word␣0xAAAAAAAA\n"); //TODO
    return a; //
}
```

- ▷ implémenter la nouvelle instruction (matériel) en indiquant clairement les étapes pour l'ajout ;
- ▷ valider le fonctionnement (en indiquant les cas de tests).

2. par exemple `https://nitish2112.github.io/post/adding-instruction-riscv/` ou `https://hsandid.github.io/posts/risc-v-custom-instruction/`)

4 Un pipeline !

Le modèle actuel est purement fonctionnel et permet de simuler un programme instruction par instruction. L'objectif est de rajouter maintenant un pipeline. On pourrait prendre un pipeline de 5 étages classiques : `Fetch`, `Decode`, `Execute`, `Memory`, `Write Back`.

Cependant, l'implémentation risque d'être trop lourde pour un TP. Nous proposons de limiter le pipeline a 2 étages (donc un seul niveau de registres de pipeline) :

— étage 1 : `Fetch`
— étage 2 : `Decode`, `Execute`, `Memory` et `Write Back`.

▷ quel type d'aléa pouvons nous rencontrer ? pourquoi ?
▷ identifiez les signaux qui doivent être sauvegardés dans les registres de pipeline

Pour simplifier l'implémentation, lors de la détection d'une bulle, l'instruction (registre d'instruction) sera remplacée par un `nop` lors du passage à l'étage suivant.

▷ modifiez le schéma pour ajouter le pipeline. Testez votre application en faisant des comparaisons avec la version sans pipeline (purement fonctionnelle).