

# td2 miniriscv

August 2, 2022

## 1 Introduction

Durant ce TD, vous disposez d'un modèle systemC d'un processeur RISCv simple. Ce modèle est incomplet : il manque le shifter, le composant permettant d'effectuer tous les décalages.

(plus d'explication sur le fonctionnement du processeur, sur le debug en utilisant gtkwave, etc)

## 2 Comprendre le mini riscv

### 2.1 Compilation d'un fichier de test

Pour tester le processeur, il faut compiler un exécutable pour l'architecture riscv.

Lorsqu'on exécute un programme avec un système d'exploitation, on a accès à une librairie standard, qui gère quelques initialisations, l'appel vers la fonction *main*, ...

Ici, nous n'avons pas accès à une telle librairie. On utilise donc un petit bout d'assembleur, situé dans *SW/reset.s*, qui définit un segment *reset*, initialise la pile, et saute vers la fonction *main*, qui devra être définie par le programme de test. Ce programme définit aussi deux fonctions, *\_good* et *\_bad*, qui permettent de sortir de la simulation selon le résultat d'un test.

Pour compiler votre programme, il faudra donc commencer par compiler le fichier *reset.o*, en utilisant la commande :

```
riscv32-unknown-elf-gcc -nostdlib -c SW/reset.s -o reset.o
```

Puis, il faut générer le *.o* de votre test, avec la commande

```
riscv32-unknown-elf-gcc -nostdlib -c [votre fichier] -o app.o
```

en remplaçant [votre fichier] par le nom du fichier que vous souhaitez compiler. Enfin, il faut *linker* ces deux fichiers, avec un fichier d'édition de lien spécial. Cela correspond à la commande :

```
riscv32-unknown-elf-gcc -nostdlib reset.o app.o -T SW/link.ld -o app
```

Cela vous génère un exécutable risc-v nommé *app*. Vous pourrez lancer la simulation sur ce fichier de la manière suivante :

```
./run.exe app
```

Pour observer le contenu du programme compilé, et ainsi savoir quelle instruction est présente dans quelle case mémoire, vous pouvez utiliser l'outil `objdump` :

```
riscv32-unknown-elf-objdump -D app
```

## 2.2 Création d'un premier fichier & observation avec GTK-WAVE

Créez un fichier de test contenant une fonction `main`, contenant une unique instruction.

```
.global main
main:
    addi    a0, x0, 23
```

Lancez la simulation. Que se passe-t-il ?

Ouvrez le fichier `tf.vcd` avec GTKWAVE pour essayer d'observer ce qu'il se passe. Vous pouvez par exemple afficher les signaux suivants :

1. *CLK* (signal d'horloge)
2. *decode/PC\_RI* (PC de l'instruction en cours d'exécution dans l'étage de code)
3. *decode/INSTR\_RI* (valeur de l'instruction en cours d'exécution dans de code).

Ajoutez maintenant une instruction

```
j    _good
```

à la fin de votre programme, et testez à nouveau. Maintenant, le programme devrait d'arrêter comme il faut.

## 2.3 écriture d'un véritable test

### 2.3.1 test de l'instruction `addi`

Écrivez un test qui vérifie que l'instruction *add* fonctionne. Voici quelques instructions pour vous aider :

```

add [rd], [rs1], [rs2]      : place rs1 + rs2 dans rd
li  [rd], immediat         : charge l'immédiat (un nombre) dans le registre rd
beg [rs1], [rs2], [label]  : saute au label spécifié si rs1 = rs2
j   _good                  : signale au simulateur que le test réussi
j   _bad                   : signale au simulateur que le test échoue

```

Ces instructions suffisent pour écrire le test demandé, mais vous pouvez en utiliser d'autres.

Votre test doit terminer, et indiquer "Success".

### 2.3.2 test des instructions de shift

On vous demande maintenant d'écrire des tests sur les instructions de shift suivantes :

```

slli [rd], [rs], immediat : effectue un shift gauche logique de rs1 d'une valeur
                           contenue dans l'immédiat.
srli [rd], [rs], immediat : effectue un shift droite logique de rs1 d'une valeur
                           contenue dans l'immédiat
srai [rd], [rs], immediat : effectue un shift droite arithmétique de rs1 d'une valeur
                           contenue dans l'immédiat

```

Ces tests doivent *échouer*, car le shifter n'est pas implémenté. Ce sera à vous de modifier le processeur pour que les tests passent dans la partie suivante.

Petit rappel sur la différence entre décalage arithmétique et logique :

1. Dans un décalage à droite arithmétique, on considère des entiers signés.  
Si on décale l'entier -2 de 1, le résultat est -1.  
Pour ce faire, il faut décaler les bits vers la droite, en copiant le bit de signe : 10000 devient ainsi 11000, et 01111 devient 00111.
2. Dans un décalage droit logique, on considère des entiers non signés, et on insère simplement des zéros à gauche pour compléter après le décalage.  
Ainsi, 10000 devient 01000.  
Notez que -2 est interprété comme 0xFFFFFFE, et devient donc 0X7FFFFFFF, soit aux alentours de 2 milliards.

## 3 Ajout du shifter

Votre objectif est maintenant de compléter le composant *shifter* du processeur. Le fichier `shifter.h` contenant les entrées / sorties du composant vous est fourni. A vous d'écrire les méthodes de la classe `shifter` permettant de calculer les sorties en fonction des entrées.

NB : en `systemC`, vous avez accès à toutes les fonctions classiques du C++, comme par exemple l'opérateur de shift `>>`