Microprocesseur RISC-V 32 bits

Systèmes Numériques 2025-2026

Assim Farsi Félix Landreau Hugo Dischert Séphora Bennaoum

26 janvier 2026

Ce rapport présente l'implémentation d'un microprocesseur RISC-V 32 bits. Le projet comprend un CPU, une tentative de CPU pipelinée, un assembleur/compilateur, et des programmes de démonstration dont une horloge temps réel. Le microprocesseur est simulé grâce à un simulateur de netlists en OCaml. Le code et la mémoire du processus partagent des espaces d'adressage séparés (architecture de Harvard).

1 Introduction

L'objectif de ce projet est de concevoir un microprocesseur capable d'exécuter un sous-ensemble de l'architecture RISC-V. Le projet comprend plusieurs composants :

— CPU : execute une instruction par cycle
— Compiler : assembleur RISC-V vers code machine
— Simulateur de netlist
— Horloge
— CPU_pipeline : tentative d'optimisation du circuit

1.1 Architecture

Nous avons :

— des instructions de base 32 bits (arithmétique, logique, etc.)
— un multiplieur et un diviseur

Le processeur est décrit en Python à l'aide de la bibliothèque `lib_carotte`, qui génère des netlists qui peuvent ensuite être simulées. Le processeur exécute une instruction complète à chaque cycle d'horloge.

2 Composants

2.1 Unité arithmétique et logique

L'**unité arithmétique (arith_unit.py)** implémente l'addition et la soustraction sur n bits. La soustraction $a - b$ est réalisée par $a + \overline{b} + 1$ (CA2). L'**ALU (alu.py)** combine quant à elle les opérations arithmétiques et logiques.

2.2 Drapeaux/Flags

Les drapeaux (flags.py) sont calculés pour les comparaisons et branchements conditionnels : Zero Flag (ZF), Carry/Borrow Flag (CF), Sign Flag (SF) et Negative Flag (NF).

2.3 Multiplieur et diviseur

Le **multiplieur** (multiplier.py) effectue des multiplications signées et non signées sur 32 bits, produisant un résultat sur 64 bits. Le **diviseur** (divider.py) calcule le quotient et le reste pour des opérandes signés ou non signés.

2.4 Registre

Le processeur dispose de 32 registres de 32 bits. Le registre x0 est à zéro. Il y a deux types de registre : un pour le CPU sans pipeline, qui intègre Program Counter (regs.py), et un conçu spécialement pour la pipeline et plus simple (regs_pipeline.py).

3 Assembleur / Compilateur

Le compilateur (compiler.py) traduit le code assembleur RISC-V en code machine. Il effectue deux passes sur le fichier source :
1. Le compilateur parcourt le fichier pour repérer tous les labels et mémoriser leur position (numéro de ligne de code). Les labels peuvent être nommésou numériques. **2 :** Le compilateur génère le code machine instruction par instruction.

3.1 Utilisation du compilateur

Pour compiler un fichier assembleur, il faut modifier la dernière ligne de `compiler.py` pour appeler la fonction `compile` avec le chemin du fichier :

```
1 if __name__ == "__main__":
2     compile("mon_programme.s")
```

Puis exécuter : `python3 compiler.py`
Le code machine est généré dans le fichier `compile.out`, qui est mis dans la ROM du simulateur.

3.2 Instructions

- **Arithmétique :** add, addi, sub, mul, mulh, mulhu, mulhsu, div, divu, rem, remu
- **Logique :** and, andi, or, ori, xor, xori
- **Décalages :** sll, slli, srl, srli, sra, srai
- **Comparaison :** slt, slti, sltu, sltui
- **Mémoire :** lw, sw
- **Branchements :** beq, bne, blt, bge, bltu, bgeu, bgt, ble, bgtu, bleu

— **Sauts :** jal, jalr, j, jr
— lui, auipc, mov, ret

4 Tentative de pipeline (cpu_pipeline.py)

Nous avons essayé de développer une version pipelinée du processeur (cpu_pipeline.py) dans le but d'améliorer les performances du microprocesseur déjà implémenté en exécutant plusieurs instructions en parallèle.

Un pipeline divise l'exécution d'une instruction en plusieurs étages. Pendant qu'une instruction est dans un étage, une autre peut être dans l'étage précédent, et ainsi de suite. Cela permet théoriquement d'exécuter une instruction par cycle une fois le pipeline rempli.

Entre chaque étage, des registres inter-étages (IF/ID, ID/EX, EX/MEM, MEM/WB) mémorisent les données nécessaires pour l'étage suivant.

Le pipeline introduit des problèmes appelés *aléas ou hazards*. Ce sont des situations où une instruction ne peut pas s'exécuter correctement car elle dépend d'une instruction précédente qui n'a pas encore terminé. Cela survient quand une instruction a besoin d'une donnée qui n'est pas encore disponible. Par exemple :

```
1    add x1, x2, x3
2    sub x4, x1, x5
```

Ici, l'instruction `sub` essaie de lire x1 alors que `add` ne l'a pas encore écrit. Sans correction, `sub` lirait une ancienne valeur de x1.

Un aléa de contrôle survient lors d'un branchement. On ne sait pas si le branchement sera pris tant que la condition n'est pas évaluée (étage EX). Entre-temps, des instructions ont déjà été chargées dans le pipeline. Le **forwarding** (`forwarding.py`) permet de résoudre la plupart des aléas de données sans perdre de cycles. L'idée est de transmettre directement le résultat d'un étage vers l'entrée de l'ALU, sans attendre qu'il soit écrit dans les registres. Concrètement, avant d'utiliser une valeur de registre dans l'étage EX, on vérifie si cette valeur est en train d'être calculée par une instruction précédente encore dans le pipeline :

— Si le registre source correspond au registre destination de l'instruction en EX/-MEM, on utilise directement la sortie de l'ALU de cette instruction.
— Sinon, si le registre correspond à celui de l'instruction en MEM/WB, on utilise la donnée qui va être écrite.
— Sinon, on utilise la valeur normale lue depuis les registres.

Le forwarding depuis EX/MEM est prioritaire sur celui depuis MEM/WB, car il contient la valeur la plus récente.

La détection des hazards (`hazards.py`) vérifie si l'instruction en EX est un load et si l'instruction en ID utilise le même registre destination. Quand un branchement est pris, les instructions qui ont été chargées après lui (dans IF et ID) ne doivent pas être

exécutées. On les annule en les remplaçant par des bulles (on met leur registre destination à x0).

4.1 État actuel du pipeline

La version pipelinée génère une netlist. Cependant, elle ne peut pas être lancée avec la clock et rencontre des erreurs. De plus, le simulateur de netlists actuel ne permet pas de visualiser l'exécution parallèle des étages.
Pour générer et simuler le CPU avec pipeline :

1. Compiler le circuit : `make cpu_pipeline`
2. Lancer la simulation : `make sim_pipeline`
3. Quand le simulateur demande le fichier ROM, indiquer le chemin vers le programme compilé (ex : `./compile.out`)

5 Horloge

clock.s incrémente les secondes, minutes et heures. clock_real_time.s pemet d'avoir une horloge en temps réel.

6 Tests et utilisation

Par défaut, les chemins sont correct si ces trois dossiers sont côte à côte :
— `Sysnum_microprocessor`
— `carotte.py (https://github.com/CarottePy/carotte.py)` : fichier carotte.py dans le dossier
— `Netlist_simulator (https://github.com/Acssiohm/Netlist_simulator)` : fichier netlist_simulator.byte dans le dossier
Une fois les bons chemins relatifs mis à jour, on peut :
— compiler un circuit circuit.py avec : make circuit
— lancer la simulation du dernier ciruit compilé avec : make sim
— compiler puis simuler avec circuit.py : make circuit sim
— make cpu : compile le CPU monocycle
— make cpu_pipeline : compile le CPU pipeliné
— pour supprimer les fichiers produits par make : make clean
Concernant le compilateur :
Il faut écrire le code assembleur dans un fichier, puis appeler la fonction python "compile" sur le chemin relatif du fichier, par exemple : compile("compiler_test.ass") Le code produit est mis dans le fichier compile.out.

Le code produit peut alors être exécuté par le CPU, en le lançant avec make cpu sim puis en indiquant l'emplacement du code à exécuter (./compile.out).