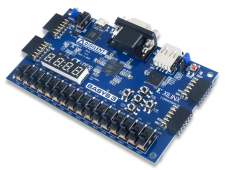
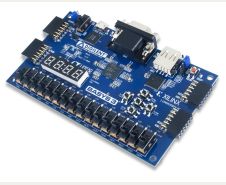
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE



PROJET SYSTÈME INFO

Leandro RODRIGUEZ - Baptiste REBILLARD
Etudiants Ingénieurs de l'INSA Toulouse
Département GEI
Promotion 59
2025-2026

Soutenance le 21/05/2025

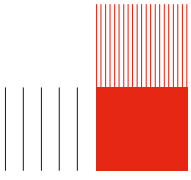


PROJET SYSTÈME INFO

Leandro RODRIGUEZ - Baptiste REBILLARD
Etudiants Ingénieurs de l'INSA Toulouse
Département GEI
Promotion 59
2025-2026

Soutenance le 21/05/2025

SOMMAIRE



1 COMPILATEUR

1.1 Opérateurs supplémentaires

Afin de rester cohérents avec les opérations disponibles dans l'unité arithmétique et logique (ALU) du processeur, nous avons implémenté les opérations bit à bit *AND*, *OR*, *XOR* et *NOT*. Nous avons également ajouté l'opération *DIFF*, absente de l'ALU, qui permet de tester l'inégalité entre deux expressions. Dans l'assembleur final, cette opération est traduite par une combinaison de *EQU* suivie de *NOT*, ce qui la rend donc compréhensible par l'ALU.

1.2 Variables globales

Dans notre implémentation, nous souhaitions introduire la gestion des variables globales. Pour ce faire, il a été nécessaire de restructurer notre analyseur grammatical : initialement, un programme était défini comme une liste de fonctions, mais il est désormais défini comme une liste de déclarations, pouvant être des fonctions, des variables ou des constantes. Les variables globales se voient attribuer un scope égal à 0, afin d'assurer leur visibilité dans l'ensemble du programme.

1.3 Structures de contrôle

1.3.1 IF

Pour implémenter l'instruction *IF* dans notre compilateur, il a fallu que lors de la génération de code, la condition soit d'abord évaluée, puis un saut conditionnel soit inséré à l'aide d'un *JMPF* vers le début du bloc *ELSE* ou la fin de l'instruction si l'*else* est absent. En cas de présence d'un *ELSE*, un saut inconditionnel (*JMP*) est ensuite ajouté pour ignorer le bloc *ELSE* après l'exécution du bloc *IF*.

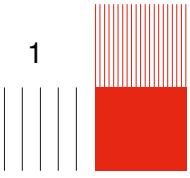
1.3.2 WHILE

Pour l'instruction *WHILE*, nous utilisons une instruction *JMPF* au début, de la même manière que pour le *IF*, afin de sauter le bloc de contrôle si la condition est fausse. Ensuite, à la fin de ce bloc, nous insérons un saut inconditionnel *JMP* vers le test de la condition initiale, ce qui permet de répéter l'exécution tant que la condition reste vraie, réalisant ainsi une boucle.

1.4 Mettre des sauts vers des instructions futures

Lors d'une structure conditionnelle comme *IF* ou *WHILE*, l'adresse de destination du saut conditionnel n'est pas connue au moment où l'on génère l'instruction `JMPF`, car le compilateur n'a pas encore parcouru tout le bloc concerné. Pour gérer cela,nous stockons chaque instruction dans un tableau et nous enregistrons temporairement l'indice de l'instruction où le `JMPF` devra être écrite. Une fois que le compilateur atteint la fin du bloc de contrôle, il peut enfin compléter correctement cette instruction avec l'adresse de saut exacte.

Concrètement, cela se fait à l'aide d'une pile dédiée aux `JMPF`, où l'on empile les positions à compléter. Plus tard, lors de la fermeture de la structure conditionnelle, on dépile cette



position et on la met à jour avec la bonne adresse de destination. Cette technique permet de générer correctement des sauts vers des adresses futures dans le programme, tout en respectant l'ordre séquentiel d'analyse du code source. Cela ressemble à cela :

```
void pushJumpf(int condition) {
    // Sauvegarde l'adresse actuelle
    int index = instruction_counter;
   [...]
    // Empile l'index actuel
    Jumpf_Table[jumpf_counter++] = index;
    // Ecrit une instruction temporaire
    sprintf(Instructions[instruction_counter++], "JMPF TMP 0x%x\n", condition);
}
```

```
int popJumpf() {
    // Récupère l'index du dernier saut et la ligne à patcher
    int jmpf_i = Jumpf_Table[jumpf_counter - 1];
    char *line = Instructions[jmpf_i];
    //Récupère la condition
    int cond;
    sscanf(line, "JMPF TMP 0x%x", &condition);
    // Écriture dans la bonne ligne
    sprintf(Instructions[jmpf_i], "JMPF 0x%x 0x%x\n", instruct_counter, cond);
    // Dépile l'index
    jumpf_counter--;
    return jumpf_index;
}
```

La même solution existe pour les *JMP*.

1.5 Imbrication des structures de contrôle

Grâce à l'utilisation des tables de saut (*JMP* et *JMPF*) combinées à la gestion des scopes, notre compilateur est capable de gérer proprement l'imbrication des structures de contrôle telles que les *IF*, et *WHILE*. Par ailleurs, la notion de scope permet d'isoler les blocs de code imbriqués et d'éviter les interférences entre les structures.

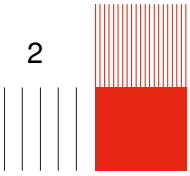
1.6 Pointeurs

Pour gérer les pointeurs, il faut pouvoir déréférencer une adresse mémoire, c'est-à-dire accéder à la valeur stockée à cette adresse. Pour cela, nous avons implémenté deux instructions : `LCOP` (Left COPy) et `RCOP` (Right COPy).

— `RCOP` permet de lire la valeur pointée par une adresse (lecture mémoire).

— `LCOP` permet d'écrire une valeur à l'adresse pointée (écriture mémoire).

Ces instructions permettent d'implémenter des opérations de type `y = *p` (`RCOP`) ou `*p = x` (`LCOP`) dans notre langage intermédiaire.



Quant à l'opérateur `&` (adresse de), il se traduit simplement par une instruction `AFC` (Affectation d'une constante), car on connait l'adresse d'une variable à la compilation.

1.7 Difficultées employées

La phase de rédaction avec `yacc` a été ponctuée de moments d'incompréhension face à certaines erreurs difficiles à diagnostiquer. Pour surmonter ces difficultés, nous avons utilisé plusieurs outils qui se sont révélés très utiles :

— L'option `-Wcounterexamples`, qui permet d'afficher des contre-exemples aux règles conflictuelles. Cela nous a évité de tester à l'aveugle et nous a guidés vers la correction des ambiguïtés.

— Le fichier `y.output`, qui fournit une représentation détaillée des états de l'automate à pile généré par `yacc`, facilitant l'analyse des conflits.

— L'option `yydebug`, qui permet de suivre en temps réel l'empilement et le dépilement des symboles sur la pile de l'analyseur, offrant une meilleure compréhension du comportement du parseur.

2 PROCESSEUR

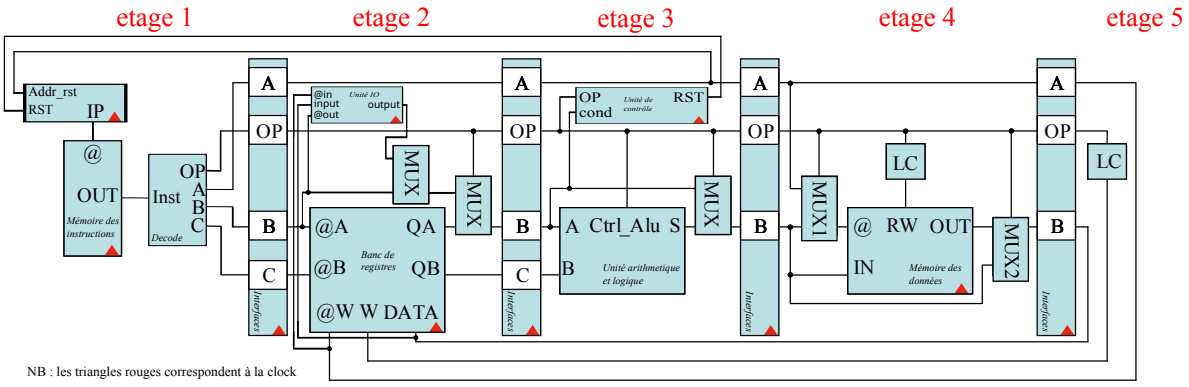


FIGURE 1 – Architecture de notre processeur

2.1 Découpage en Fetch/Decode

Pour commencer, nous avons ajouté un module de décodage de l'instruction. Ce bloc très simple, fonctionnant de manière asynchrone, prend en entrée un signal *Instruction* et renvoie les signaux *OP*, *A*, *B* et *C* correspondants.

2.2 Gestion des aléas

2.2.1 Description du problème

Nous avons remarqué que des aléas de données pouvaient survenir lors de certaines opérations, par exemple lorsque deux instructions utilisant l'UAL (Unité Arithmétique et Logique) étaient exécutées consécutivement. En effet, comme l'UAL est située à l'étage 3 du

pipeline, une instruction en cours de lecture à l'étage 1 peut accéder à un registre dont la valeur n'a pas encore été mise à jour par une instruction précédente encore en cours d'exécution à l'étage 3.

Exemple :

```
ADD 1 2 3      ; R1= R2 + R3
SOU 4 1 5      ; R4= R1 - R5
```

Tick	Ét.1 (Fetch/Decode)	Ét.2 (Lecture registre)	Ét.3 (Exécution UAL)
1	ADD r1 r2 r3		
2	SOU r4 r1 r5	ADD r1 r2 r3	
3		SOU r4 r1 r5	ADD r1 r2 r3
4			SOU r4 r1 r5

On observe qu'au **tick 3**, l'instruction `SOU` lit `r1` alors que la valeur correcte n'a pas encore été écrite par l'instruction précédente. L'écriture du résultat de l'addition dans R1 arrivera au **tick 5**, car le stockage dans le banc de registre se situe à l'**étage 5**.

2.2.2 Résolution

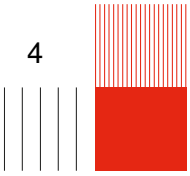
Afin de résoudre ce problème, il suffit d'insérer un certain nombre de `NOP` entre deux instructions, en fonction de l'instruction précédente, et, pour aller plus loin, selon la présence éventuelle de conflits sur les registres et/ou en mémoire.

De notre côté, ne cherchant pas à optimiser le processeur, nous avons simplement inséré 4 instructions `NOP` entre chaque instruction, ce qui réduit la vitesse d'exécution du programme. Cependant, vers la fin de notre projet, nous avons essayé d'implémenter une version plus optimale mais qui n'a pas été finie par manque de temps et donc laissée dans une branche de notre dépot git.

2.2.3 Début de réalisation

Pour gérer les aléas de données, nous avons observé que certaines instructions écrivant dans un registre (instructions arithmétiques, logiques, ainsi que `COP`, `AFC` et `LOAD`) peuvent provoquer des conflits si elles sont immédiatement suivies d'instructions lisant ce même registre (`COP`, `STORE`, opérations, etc.). Afin d'éviter ces situations, notre contrôleur détecte les cas à risque et insère automatiquement des instructions `NOP` en bloquant le compteur de programme (PC), permettant ainsi à l'écriture dans le registre de s'effectuer correctement.

Pour cela, nous avons conçu une unité de contrôle des aléas fonctionnant de manière asynchrone. Elle analyse les données circulant dans chaque étape du pipeline et, en cas de détection d'un aléa, elle lève un drapeau (*flag*). Après l'étape de pipeline LI/DI, nous avons ajouté une seconde étape de pipeline, dotée d'une mémoire tampon. En cas d'aléa, cette mémoire sauvegarde l'instruction en cours et insère des `NOP` dans le pipeline. Une fois l'aléa résolu, le compteur de programme est décrémenté d'une instruction afin de réexécuter celle qui avait été temporairement mise en attente.



3 UNITÉ IO

Lors de la mise en place de la démonstration, nous nous sommes demandé comment interagir simplement et de manière synchrone avec la carte, c'est-à-dire en envoyant de nouvelles instructions depuis le code C. Nous avons observé que d'autres groupes hardcodaient directement les valeurs, mais cette approche ne permettait pas une intégration propre et flexible dans le programme.

Nous avons donc ajouté une `unité IO` aux étages 2 et 5 du processeur, comme illustré sur la figure 1.

— À l'étage 2, cette unité prend une adresse en paramètre et renvoie la valeur qu'elle lit à cette adresse (lecture de périphériques comme les interrupteurs ou boutons).

— À l'étage 5, elle fonctionne différemment : elle reçoit l'adresse via le bus A et la donnée à écrire via le bus B (provenant d'un registre), ce qui lui permet d'écrire la valeur à la bonne adresse — par exemple, pour allumer une LED.

```c
void main() {
   int a,b,c;

   while (1) {
      a = readIO(3);
      b = readIO(4);
      writeIO(1, a+b);
   }
}
```

FIGURE 2 – Exemple de code IO : on lit les entrée IN3 et IN4 et on montre l'addition sur OUT1

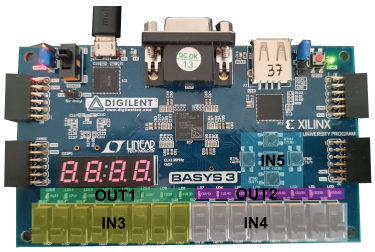


FIGURE 3 – Adresses des IN et OUT

3.1 Synchronisation de la lecture sur la mémoire de données

Etant donné que la mémoire de donnée était synchronisé sur le rising edge de la clock comme les séparateurs,

Pour notre problème, trois solutions sont envisageables :

1. désynchroniser la lecture de la mémoire de données ;

2. désynchroniser l'interface de sortie du 4e étage (celui de la mémoire de données) ;
3. synchroniser la lecture de la mémoire sur le *falling edge*.

L'option 1 est la plus simple à mettre en œuvre sous Vivado, mais reste peu réaliste : en pratique, la mémoire de données est composée de cellules SRAM ou DRAM. Une lecture mémoire implique un décodage d'adresse, un accès à la cellule et souvent un amplificateur de lecture. Ce qui rend la lecture nécessairement synchrone. À l'inverse, un banc de registres, constitué de bascules (flip-flops) très rapides, permet une lecture asynchrone.

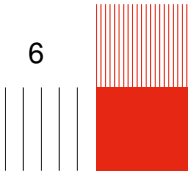
L'option 2 n'est pas compatible avec notre architecture actuelle : synchroniser les deux derniers étages et insérer des NOP entre les instructions sont incompatible, ce qui empêche le système de fonctionner correctement.

Nous retenons donc l'option 3, qui simule un délai de lecture correspondant à un demi-cycle d'horloge, laissant le temps aux données de se stabiliser avant d'être transférées à l'étage suivant sur le *falling edge*.

3.2 OP codes

Pour simplifier la lecture et le décodage des instructions, nous avons opté pour un schéma d'adressage différent (voir Tableau 1). Pourquoi avons-nous choisi une taille de 8 bits ? Théoriquement, 5 bits auraient suffi pour coder l'ensemble des instructions nécessaires. Toutefois, nous avons préféré utiliser 8 bits pour plusieurs raisons : cela permet d'arrondir à l'octet supérieur, de normaliser la taille de toutes les instructions, et de simplifier le traitement matériel en travaillant systématiquement sur des entiers de 8 bits.

Dans ce format, les 4 bits de poids forts indiquent le type d'instruction (par exemple, `0001` désigne une instruction de type `ALU`), tandis que les 4 bits de poids faibles précisent l'instruction spécifique au sein de ce type.



Instr ALU	Binaire	Hexa
NOP	0000 0000	0x0
ADD	0001 0000	0x10
SOU	0001 0001	0x11
MUL	0001 0010	0x12
DIV	0001 0011	0x13
AND	0001 0100	0x14
OR	0001 0101	0x15
XOR	0001 0110	0x16
NOT	0001 0111	0x17
INF	0001 1000	0x18
INFE	0001 1001	0x19
SUP	0001 1010	0x1A
SUPE	0001 1011	0x1B
EQU	0001 1100	0x1C
COP	0010 0000	0x20
AFC	0010 0001	0x21
JMP	0010 0010	0x22
JMPF	0010 0011	0x23
LOAD	0010 0101	0x25
STORE	0010 0110	0x26
LCOP	0010 0111	0x27
RCOP	0010 1000	0x28
PRI	0010 0100	0x24
READ	0010 1001	0x29

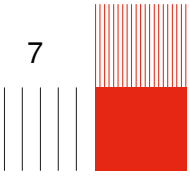
TABLE 1 – Instructions

Vers la fin du projet, nous nous sommes rendu compte que nous aurions pu approfondir cette approche en utilisant, par exemple, des bits supplémentaires parmi les bits de poids fort pour coder des informations spécifiques sur les instructions. L'un de ces bits pourrait indiquer si l'instruction écrit dans le banc de registres, un autre si elle écrit dans la mémoire.

Une telle convention aurait permis une implémentation plus simple et plus lisible, en remplaçant les longues conditions basées sur des séries de `OR` sur les `OPcodes` par une vérification directe d'un bit. Cela rend non seulement la logique plus concise, mais facilite également la compréhension du comportement d'une instruction, en permettant de savoir rapidement si elle effectue une écriture, et vers quelle destination (registre ou mémoire).

3.3 JUMP et JUMPF

Concernant les instructions `jump` et `jumpf`, nous avons ajouté un élément appelé « unité de contrôle », visible sur la Figure 1. Son rôle est d'interpréter l'opération en cours et, s'il s'agit d'un `jump` ou d'un `jumpf`, de mettre à jour l'instruction suivante à exécuter en modifiant la valeur du `Instruction Pointer` (IP). Dans le cas d'un jump, cette mise à jour est effectuée immédiatement. Pour un `jumpf`, l'unité de contrôle vérifie d'abord la sortie de l'`UAL` avant de décider de l'éventuel saut.



4 CROSS-COMPILATEUR

Le cross compilateur a été fait, le fonctionnement est ici expliqué pour une future implémentation

4.1 Choix d'implémentation

- Le script a été écrit en Python pour sa simplicité de manipulation de fichiers et de chaînes.
- La compilation se fait en deux passes :
  - **Pass 1** : construction d'une table `addr_to_index` pour convertir les adresses logiques en indices physiques dans le tableau final.C'est utile pour implémenter ensuite les *JMP* et *JMPF* sans qu'il y ait de décalage dans le saut.
  - **Pass 2** : génération ligne par ligne du code hexadécimal, avec des registres temporaires utilisés pour simuler les opérations. Cela permet de traduire de l'assembleur orienté mémoire vers de l'assembleur orienté registre, même si après discussion avec notre professeur nous n'avons pas exactement compris ce qui était attendu (c.f :4.3)

4.2 Passage de mémoire à registres

Dans notre compilateur, par exemple lors d'une addition, les deux termes sont stockés en mémoire, cependant dans le processeur, ces mêmes termes sont stockés dans des registres. Pour regler cela il faut insèrer des instructions *LOAD* et *STORE* entre la mémoire et les registres avant et après chaque opération. Par exemple, pour une opération d'addition, le second terme est chargé en *R2*, le troisième en *R1*, puis le résultat est calculé et stocké dans *R3*. Enfin, la valeur contenue dans *R3* est sauvegardée en mémoire à l'adresse correspondant au premier terme. On a donc :

```
ADD @3 @2 @1
```

qui devient :

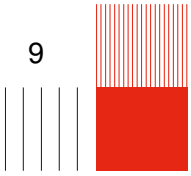
```
LOAD R1 @1
LOAD R2 @2
ADD R3 R2 R1
STORE @3 R3
```

4.3 Problèmes rencontrés et solutions

Au debut nous avons eu des décalages entre adresses logiques et indices d'instructions dans Vivado. Cela a été résolu avec une première passe de comptage des instructions, tenant compte du nombre d'instructions réelles générées pour chaque ligne. De plus un autre soucis a été que nous croyons que nous voulions garder un processeur qui fait les instructions dans les registres, mais qui ensuite `STORE` ces dernières dans la mémoire. La version que nous avons fait n'est pas vraiment similaire au *RISC-V* mais au final cela permet d'avoir plus de place car la mémoire fait 1*Kio* au lieu des 16 registres de 4 octets.

4.4 Résultats obtenus

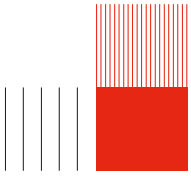
— Génération correcte du code hexadécimal compatible Vivado.
— Conversion réussie de programmes utilisant des instructions arithmétiques, logiques, de saut et de mémoire.



LISTE DES FIGURES ET TABLEAUX

Liste des figures

Liste des tableaux



INSA TOULOUSE
135 avenue de Rangueil
31400 Toulouse

Tel : +33 (0)5 61 55 95 13
www.insa-toulouse.fr