Rapport de stage

Développement de processeurs softcore 8 et 16 bits avec intégration d'un assembleur inline

Lucas GAUVAIN
École Polytechnique Universitaire de Sorbonne Université
Electronique et Informatique - Systèmes Embarqués

2023 - 2024

Stage du 01/07/2024 au 31/08/2024

Maître de stage : Pr. Mario GAZZIRO

Enseignant référent : M. Thibault HILAIRE

Établissement d'origine : École Polytechnique Universitaire de Sorbonne Université, Paris (France)

Université d'accueil : Universidade Federal do ABC, Santo André (Brésil)

Internship Summary

This report presents the internship I completed, which lasted for two months, as part of my 4th year of engineering school, running from July 1, 2024, to August 30, 2024.

The main objectives of this technical internship were to apply and deepen the skills acquired during my training by working on a concrete project. It also provided the opportunity to explore the field of academic research, particularly in the areas of computer science and embedded systems. The goal of my internship was to participate in the development of a processor with dual purposes : for academic use and for running retro video games. This report illustrates the developments achieved over these two months of internship.

The internship took place at Universidade Federal do ABC in Brazil. It also allowed me to practice two foreign languages, English and Portuguese. Additionally, the internship provided me with the opportunity to discover a new country, new cultures, and traditions unique to Brazil.

In appendix A.1 you will find the summary self-assessment form for the internship.

Résumé du stage

Ce rapport présente le stage réalisé, d'une durée de deux mois, s'inscrivant dans le cadre de ma 4ème année d'école d'ingénieurs, et se déroulant du 1er juillet 2024 au 30 août 2024.

Ce stage technique a pour principaux objectifs de mettre en pratique et d'approfondir les compétences acquises durant ma formation, en les appliquant à un projet concret. Il offre également l'opportunité de découvrir le milieu de la recherche universitaire, notamment dans les secteurs de l'informatique et des systèmes embarqués. Le but de mon stage était de participer au développement de processeurs ayant une double utilité : servir à des fins académiques, et permettre l'exécution de jeux vidéo rétro. Ce rapport illustre donc les développements réalisés tout au long de ces deux mois de stage.

Ce stage a été réalisé à l'Universidade Federal do ABC au Brésil. Ainsi, ce dernier me permet également de pratiquer deux langues étrangères, l'anglais et le portugais. Ce stage m'a aussi offert l'opportunité de découvrir un nouveau pays, de nouvelles cultures et de nouvelles traditions qui sont propres au Brésil.

En annexe A.1 se trouve la fiche récapitulative d'auto-évaluation du stage.

Remerciements

Je souhaite tout d'abord exprimer ma gratitude envers toutes les personnes qui m'ont aidé à décrocher ce stage à l'Universidade Federal do ABC (UFABC), sans qui cette mobilité internationale n'aurait pas été possible.

Je tiens à remercier M. Cyril Thabourey pour m'avoir présenté cette université brésilienne et pour son soutien constant lors des démarches administratives.

Je remercie également Mme Mathilde Champeau, enseignante à l'UFABC, pour son accompagnement dans les démarches administratives et pour m'avoir mis en contact avec plusieurs professeurs travaillant dans le domaine des systèmes embarqués.

Enfin, je remercie mon maître de stage, Pr. Mario Gazziro, pour m'avoir proposé un sujet de stage et accueilli dans son laboratoire de recherche. Je remercie tous mes collègues et tous les professeurs du laboratoire avec qui j'ai eu la chance de travailler. Ce fut un réel plaisir de travailler à leurs côtés.

Table des matières

Table des figures

1 Présentation du laboratoire

Le stage s'est déroulé au sein du laboratoire LAOB, un laboratoire matériel et logiciel, spécialisé dans les réseaux, le matériel reconfigurable, la sécurité, l'infographie, la réalité virtuelle et l'embarqué. C'est l'un des laboratoires de recherche pour la formation en ingénierie de l'information de l'Universidade federal do ABC, à Santo André dans la région de São Paulo.

Dans ce laboratoire, différents projets dans différents domaines sont mis en oeuvre. Nous pouvons y retrouver des projets de réalité virtuelle, des projets de scanner 3D, des projets avec processeurs CISC et RISC sur FPGA, des projets d'informatique judiciaire et cybersécurité et de la rétro-informatique et conception de jeux vidéo classiques.

Le laboratoire LAOB est constitué d'une équipe de 4 enseignants et de plusieurs étudiants de l'université. Parmi l'équipe permanente d'enseignants, se trouve :

- Pr. Mario Gazziro, Professeur coordinateur du laboratoire ;
- Pra. Victoria Alejandra Salazar Herrera, Professeur et membre du laboratoire ;
- Pr. Hugo Portes d'Araújo, Professeur membre du laboratoire ;
- Pr. Jecel Mattos de Assumpcao Jr, Professeur d'informatique ;

Une vingtaine d'étudiants de plusieurs niveaux et de plusieurs formations viennent également travailler dans ce laboratoire, pour étudier, travailler sur leurs projets académiques ou pour participer au projets de recherche et de développement en cours.

L'environnement de travail dans le laboratoire est représenté sur la figure ci-dessous :

FIGURE 1 – Laboratoire : Présentation du laboratoire

Mon bureau est le suivant :

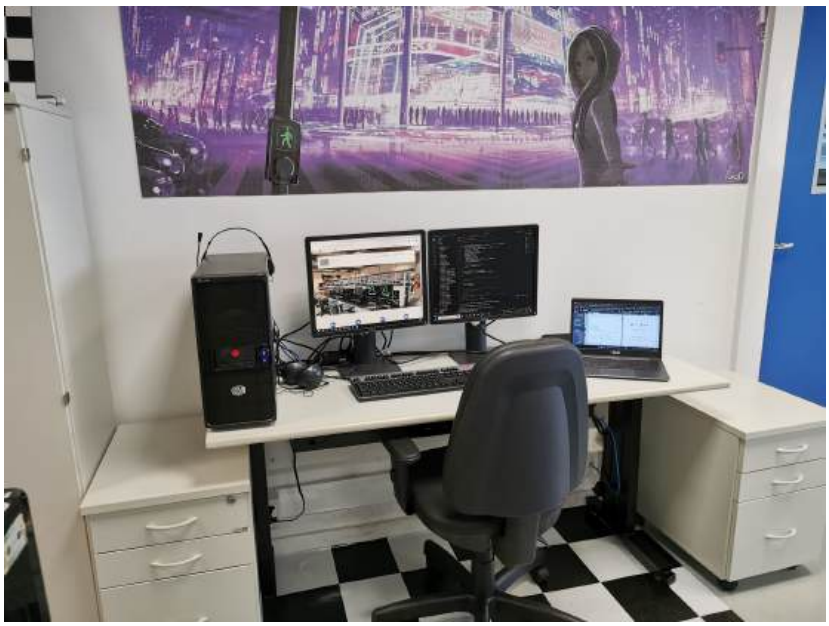


FIGURE 2 – Laboratoire : Bureau de travail

2 Introduction du stage

Comme introduit dans la présentation du laboratoire, deux sujets de recherche sont les processeurs RISC sur FPGA ainsi que la réalisation et l'exécution de jeux vidéo rétro et classiques. Une première borne d'arcade déjà développée dans ce laboratoire est présente et fonctionnelle. Elle embarque un FPGA présent sur une carte TANG NANO 20K, dans laquelle est intégré un processeur 8 bit, permettant l'émulation de jeux conçus pour les plateformes NES (Nintendo Entertainment System). Cette carte est principalement utile pour la conception et l'exécution de jeux vidéo rétro car elle permet de reproduire avec précision l'architecture matérielle des consoles de jeux vidéo classiques.

FIGURE 3 – Laboratoire : Borne d'arcade du laboratoire LAOB

Un des objectifs du laboratoire est de développer un autre processeur et de l'embarquer sur cette carte, afin d'exécuter des jeux plus complexes au niveau sonore et graphique, demandant donc plus de ressources. Pour réaliser cet objectif, les professeurs du laboratoire ont pensé au développement d'un processeur basé sur une architecture *RISC-V*. Ainsi, la mission principale de mon stage sera de participer au développement de nouveaux processeurs, et de développer des compilateurs afin de traduire correctement les futures instructions qui seront renvoyées. Si le temps nous le permet, nous tenterons d'embarquer un nouveau processeur dans une plate-forme d'émulation de jeu vidéos en ligne, 8bitworkshop [1]. Un livre "Livro Computadores e Videogames" [2] a précédemment été écrit par les enseignants du laboratoire.

3 Découverte et prise en main de l'environnement

3.1 Découverte de 8bitworkshop

Pour tester de nouveaux processeurs et compilateurs afin d'exécuter des jeux vidéo rétro, la plateforme en ligne 8bitworkshop permet de programmer, tester et émuler des jeux vidéo rétro. Cette plateforme offre un environnement de développement intégré (IDE) accessible via un navigateur, avec la possibilité de créer une instance locale pour plus de flexibilité. 8bitworkshop est spécialement conçue pour le développement de jeux vidéo rétro et l'apprentissage de la programmation sur des systèmes 8 bits classiques, comme l'Atari 2600 ou la NES (Nintendo Entertainment System). Elle propose des outils de simulation en temps réel et une interface permettant de visualiser chaque signal, rendant cette ressource très intéressante et intuitive pour les passionnés de rétro-gaming, les développeurs amateurs, et ceux intéressés par l'architecture des systèmes de jeu anciens. Il est cependant possible de modifier le CPU 8 bit déjà présent, codé en Verilog, ainsi que le compilateur en langage JSON.

Dans le cadre de notre mission, notre équipe se concentrera sur l'adaptation d'un nouveau CPU en Verilog et d'un compilateur en JSON, une autre équipe cherchera à programmer de nouveaux jeux vidéo.

Actuellement, un processeur « femto8 » est intégré dans 8bitworkshop. J'ai ainsi d'abord commencé par comprendre le fonctionnement de ce processeur et du compilateur déjà présent. Le fichier décrivant le CPU contient trois modules, comme illustré dans l'annexe A.2.

1. L'Unité Arithmétique et Logique (ALU) :

Ce premier composant décrit permet l'exécution des opérations arithmétiques et logiques. Il possède comme entrées :

- A : un opérande d'entrée de N bits
- B : un autre opérande d'entrée de N bits
- carry : une entrée de retenue à un bit utilisée dans certaines opérations
- aluop : un signal de contrôle de 4 bits qui détermine l'opération que l'ALU effectuera

Sa sortie est Y, un résultat de sortie de N+1 bits (comprenant un bit supplémentaire pour la retenue). L'ALU possède également un paramètre N, définissant la largeur des opérandes d'entrée A et B. La valeur par défaut est de 8 bits. Au niveau de la logique interne, le cœur de la fonctionnalité de l'ALU est implémenté à l'aide d'un bloc combinatoire always, qui contient une instruction case. L'instruction case sélectionne l'opération à effectuer en fonction de l'entrée aluop. Plusieurs opérations unaires et binaires sont supportées. Parmi les opérations unaires, agissant seulement sur l'entrée A, nous retrouvons :

- OP_ZERO : définit Y a 0
- OP_LOAD_A : charge A dans Y avec un 0 supplémentaire en tant que bit de retenue.
- OP_INC : incrémente A de 1
- OP_DEC : décrémente A de 1
- OP_ASL : effectue un décalage arithmétique à gauche sur A, en insérant un 0 dans le bit de poids faible
- OP_LSR : effectue un décalage logique à droite, avec le bit de poids faible déplacé dans la position de retenue
- OP_ROL : effectue une rotation à gauche, le bit de retenue étant inséré dans le bit de poids faible
- OP_ROR : effectue une rotation à droite, le bit de retenue étant inséré dans le bit de poids fort

Parmi les opérations binaires, agissant sur les entrées A et B, nous retrouvons :

- OP_OR : effectue un OU bit à bit entre A et B
- OP_AND : effectue un ET bit à bit entre A et B
- OP_XOR : effectue un OU exclusif bit à bit entre A et B
- OP_LOAD_B : charge B dans Y avec un 0 supplémentaire en tant que bit de retenue

Parmi les opérations binaires avec retenue, agissant sur les entrées A et B, nous retrouvons :

- OP_ADD : ajoute A et B
- OP_SUB : soustrait B de A
- OP_ADC : ajoute A, B, et l'entrée de retenue
- OP_SBB : soustrait B et l'entrée de retenue de A

Pour permettre le contrôle de l'ALU, les opérations sont sélectionnées à l'aide d'un signal de contrôle de 4 bits (aluop), défini par des macros qui correspondent à des valeurs de 4 bits.

2. Le Central Processing Unit (CPU)

Il utilise un ensemble de registres, un décodeur d'instructions, et une unité arithmétique et logique (ALU) pour exécuter des instructions. Parmi ses entrées, nous retrouvons :

- clk :le signal d'horloge
- reset : un signal de réinitialisation
- data_in : un bus de données d'entrée de 8 bits

Au niveau des sorties, nous avons :

- address : un bus d'adresses de 8 bits, utilisé pour lire ou écrire des données
- data_out : un bus de données de sortie de 8 bits
- write : un signal de contrôle indiquant une écriture mémoire

Il dispose de plusieurs registres internes :

- IP : le registre du pointeur d'instructions (Instruction Pointer) de 8 bits, qui contient l'adresse de la prochaine instruction à exécuter
- A, B : deux registres de 8 bits utilisés pour stocker des opérandes
- Y : un registre de 9 bits utilisé pour stocker les résultats des opérations de l'ALU (avec un bit supplémentaire pour la retenue)
- state : un registre de 3 bits utilisé pour gérer l'état actuel du processeur dans son cycle d'exécution
- carry : un registre à un bit pour stocker la retenue (carry) des opérations arithmétiques
- zero : un registre à un bit pour stocker l'indicateur de zéro, qui est vrai si le résultat d'une opération est zéro
- flags : un bus de 2 bits combinant les indicateurs carry et zero

Pour permettre le décodage de l'instruction, nous avons le registre opcode, un registre de 8 bits extrait de la mémoire pour déterminer quelle instruction doit être exécutée. Il peut être décomposé en 3 autres registres :

- aluop : les 4 bits de poids faible de l'opcode, utilisés pour sélectionner l'opération à effectuer par l'ALU
- opdest : bits 4 et 5 de l'opcode, utilisés pour sélectionner la destination du résultat de l'ALU
- B_or_data : bit 6 de l'opcode, utilisé pour choisir entre l'utilisation du registre B ou des données immédiates pour l'opérande B de l'ALU

Le processeur traverse plusieurs états dans son cycle d'exécution. À chaque front montant de l'horloge, il évalue son état et effectue les opérations nécessaires. Les différents états sont :

- S_RESET : l'état de réinitialisation du processeur, où il réinitialise le pointeur d'instructions (IP) à `0x80` et se prépare à l'exécution
- S_SELECT : l'état où l'adresse de l'opcode actuel est sélectionnée en fonction de la valeur de IP
- S_DECODE : l'état où l'opcode est lu depuis la mémoire et l'instruction est décodée

- S_COMPUTE : l'état où l'ALU effectue l'opération spécifiée par l'opcode
- S_READ_IP : l'état où le processeur lit une nouvelle valeur pour le pointeur d'instructions depuis la mémoire (mode immédiat)

Ainsi, le CPU est constitué d'une ALU, qui effectue les opérations sur les registres A et B, ou A et les données immédiates, selon l'instruction décodée. Le résultat de l'ALU est ensuite stocké dans le registre Y.

3. Un environnement de test

Cet environnement de test implémente le module CPU et le connecte à une mémoire vive (RAM) de 128 bits et une mémoire morte (ROM) d'également 128 bits. Lorsque le signal write_enable du module de test est actif, les données provenant du CPU sont écrites à une adresse spécifiée dans la RAM.

Pour tester ce processeur, il y a un petit code assembleur qui est utilisé. Dans cet exemple, il permet de calculer la suite de Fibonacci, et de charger le résultat dans la ROM. Ce programme :

1. Initialise le registre A à 0 et B à 1
2. Additionne les valeurs de A et B puis échange leurs contenus
3. Boucle jusqu'à ce que le drapeau carry soit défini, c'est-à-dire jusqu'à ce qu'une valeur décimale sur 8 bits soit dépassée
4. Réinitialise le CPU une fois la séquence terminée

Lorsque l'on simule sur 8bitworkshop, nous pouvons observer que notre programme s'arrête, sur les registres A et B, avel les valeurs 144 et 233. Ceci est normal car, le prochain nombre de la suite de Fibonacci est censé être 377, mais ne peut pas être stocké sur 8 bits. Comme le montre la simulation ci-dessous, le signal de carry passe à 1 et notre boucle s'arrête, reset et recommence.

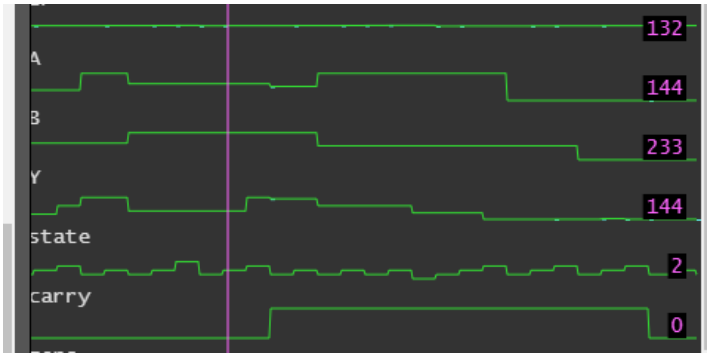


FIGURE 4 – Simulation de la suite de Fibonacci avec femto8

Afin de compiler ce code de test, un compilateur est présent pour traduire les instructions. Ce dernier est écrit en JSON, comme l'illustre l'annexe A.3. Une première section, intitulée 'vars', définit les différentes variables utilisées dans les instructions du processeur :

- reg : représente les registres du processeur, codés sur 2 bits, avec quatre valeurs possibles (a, b, ip, none)
- unop : définit les opérations unaires, codées sur 3 bits
- binop : définit les opérations binaires, également codées sur 3 bits
- const4 : représente une constante codée sur 4 bits
- imm8 : représente une valeur immédiate codée sur 8 bits

Ensuite, une seconde section 'rules' définit les règles de traduction des instruction en code machine. Chaque règle est définie par un format (fmt) décrivant la syntaxe de l'instruction en assembleur, représentée par une séquence de bits (bits) dans le langage machine. Parmi les bits de ces séquences, certains sont fixes et propre à chaque instructions, et d'autres correspondent aux registres ou aux valeurs immédiates utilisées. Cette séquence est ensuite interprétée par l'ALU décrite dans le processeur.

4 Le développement de deux processeurs

4.1 NCPU : Un processeur 8 bits

Le premier processeur que nous avons développé lors de mon stage se nomme NCPU, et a été développé à l'aide du logiciel Digital [3]. Digital est un logiciel permettant de concevoir des circuits logique numériques et de les simuler. Il est possible, une fois les circuits terminés, de les exporter en format Verilog.

Ce premier processeur n'est pas un processeur *RISC-V*. Il s'agit d'un processeur minimaliste de seulement 8 bits inspiré d'un projet github, MCPU [4], principalement utilisé pour des fins pédagogiques. Ce processeur minimaliste ne possède que trois registres (R_0, R_1 et R_2), un compteur de programme (PC) et un registre d'instruction (IR). Pour fonctionner, nous l'avons équipé d'une Unité Arithmétique et Logique (ALU) et d'une unité de contrôle permettant l'exécution des instructions. A l'intérieur de l'Unité Arithmétique et Logique (ALU), nous avons placé un registre de 2 bits permettant de stockés deux flags, un pour la retenue (C) et un pour le zéro (Z). De même, dans l'unité de contrôle, un registre de 2 bits indique le cycle en cours, que ce soit le cycle de récupération (fetch), le cycle d'exécution (execute) ou le cycle de saut (skip).

- Le cycle de récupération, ici, correspondra à la première étape du traitement d'une instruction. Durant ce cycle, le processeur lit l'instruction à exécuter depuis la mémoire, en utilisant le compteur de programme (PC) pour savoir où chercher. Une fois l'instruction récupérée, elle est placée dans le registre d'instruction (IR). Ce cycle sert ainsi à préparer le processeur à comprendre ce qu'il devra faire par la suite.
- Après avoir récupéré l'instruction, le processeur entre dans le cycle d'exécution. C'est à ce moment que l'instruction va être exécutée. Selon le type d'instruction, le processeur va travailler avec l'Unité Arithmétique et Logique (ALU) et les registres pour exécuter une instruction.

- Enfin, le cycle de saut sera utilisé lorsque le processeur doit passer à côté de certaines instructions (notamment lors de l'exécution d'instructions conditionnelles).

Ainsi, ces trois cycles permettent au processeur de décomposer le traitement d'une instruction en étapes claires et ordonnées. Chaque cycle se concentre sur une tâche spécifique, assurant ainsi une exécution plus fiable des instructions.
Nous avons programmé deux formats d'instructions, que l'on peut visualiser dans le tableau ci-dessous :

Bits	7	6	5	4	3	2	1	0
Format 1	s	s	d	d	c	c	a	i
Format 2	1	1	t	t	b	b	b	b

Table 1 – NCPU : Format des instructions

Le premier format sera utilisé pour les opérations. Comme représenté dans le tableau ci-dessus, nous avons pensé que les instructions sur 8 bits pourraient être représentées de la sorte, avec des bits 's', 'd', 'c', 'a' et 'i', où :

- 'd' indique le registre de sauvegarde, qui peut donc être R_0, R_1, R_2 ou PC. Le choix du registre ou du compteur de programme détermine donc où l'instruction va placer le résultat de l'opération qu'elle exécute.
- 's' indique l'origine du deuxième opérande pour l'opération. Le premier opérande est en effet le même que la destination (d), et le deuxième opérande est donc spécifié par s.
- 'c' indique la source de la retenue pour les opérations d'addition. Si ce bit est à 0, cela signifie qu'il n'y a pas de retenue. S'il est à 1, il y a une retenue de 1 ajoutée.
- 'a' indique si le résultat doit être ajouté au contenu actuel du registre de destination plutôt que de simplement le remplacer. Si a est activé, le contenu du registre de destination sera ajouté à l'opérande source.
- 'i' indique si le deuxième opérande source doit être inversé bit par bit avant l'opération. Cela sera principalement utilisé pour les soustractions.

Le second sera pour les branches conditionnelles. Ce second format, avec les bits 4 et 5 (notés 't' dans le tableau ci-dessus), permet de tester les flags de retenue (C) et de zéro (Z) et ainsi d'incrémenter le compteur de programme (PC) à l'aide des bits 0 à 3 (notés 'b' dans le tableau ci-dessus). Les instructions de branchement conditionnel incluent des commandes comme bcc, bcs, bzc et bzs. Elles peuvent aussi être renommées pour refléter des conditions spécifiques comme bne (branch if not equal) et beq (branch if equal).

Afin de mieux interpréter les deux formats d'instructions exposés, notamment le premier format, nous pouvons représenter les instructions dans les deux tableaux suivants :

carry	a=0, i=0	a=0, i=1	a=1, i=0	a=0, i=1
0	B	not B	A+B	A-B-1
1	B+1	0-B	A+B+1	A-B
C	B+C	(1-C)-B	A+B+C	A-B+(1-C)
logic	M	A and B	A or B	A xor B

TABLE 2 – NCPU : Interprétation des instructions

Par la suite, les opérations arithmétiques $B + C$ et $(1 - C) - B$ ont été remplacées par les instructions lda (load) et sta (store). Ces changements rendent le processeur plus flexible et plus facile à comprendre.

- $B + C$ était une opération d'addition entre le registre B et le registre C.
- $(1 - C) - B$: Cette opération était une soustraction modifiée. 1-C correspondait à l'inversion du registre C suivie d'une addition de 1, ce qui constitue un complément à deux. Ensuite, cette valeur est soustraite du contenu de B.

Nous avons donc un nouveau tableau mis à jour :

cc \ ai	00	01	10	11
00	mov	not B	add	
01	inc	neg		sub
10	lda	sda	adc	sbb
11	ldi	and	or	xor

TABLE 3 – NCPU : Interprétation mis à jour des instructions

Afin de visualiser toutes ces instructions qui peuvent être exécutées par le processeur, nous avons commencé par concevoir l'Unité Arithmétique et Logique (ALU) sur le logiciel Digital. Après un peu plus de deux semaines de tests et de conceptions, le modèle final de l'ALU est le suivant :

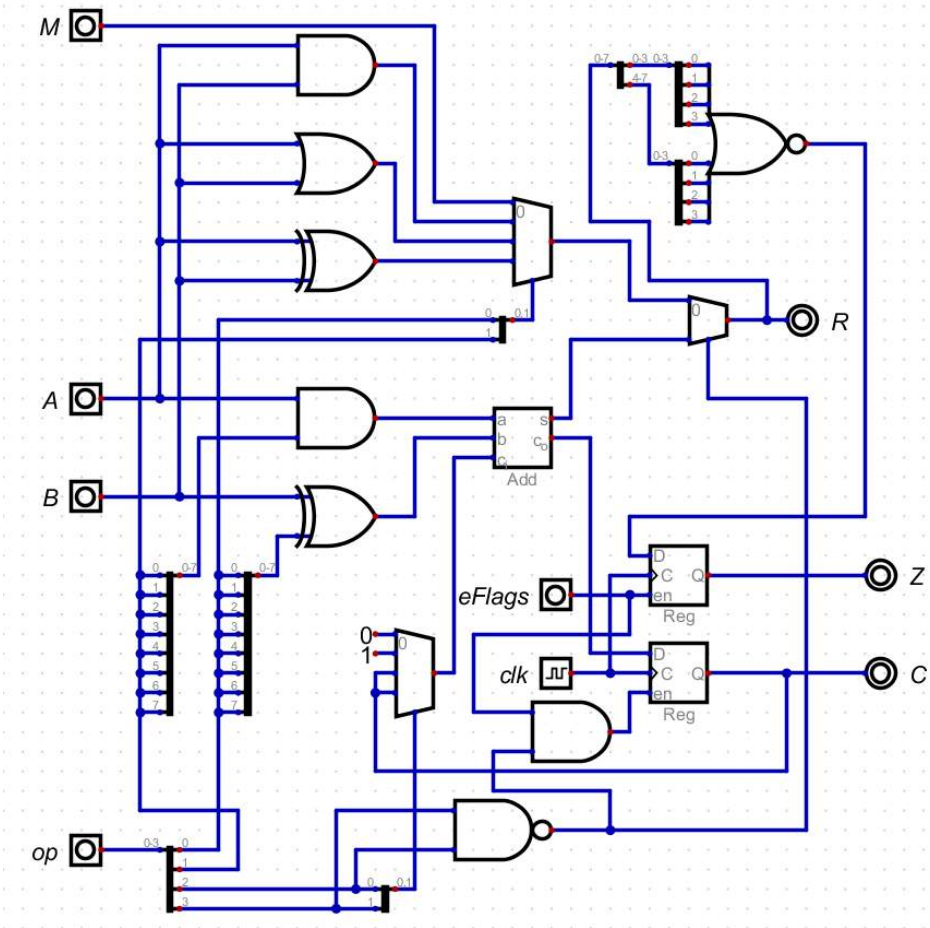


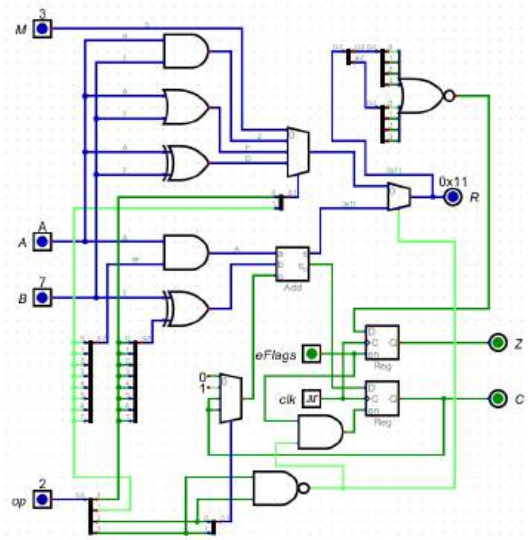
FIGURE 5 – NCPU : Circuit logique de l'ALU du NCPU

Afin de réaliser les instructions, nous avons placé les entrées A, B, OP, et M. Premièrement, pour réaliser les opérations arithmétiques, l'idée était de placer en face des des registres A et B deux portes logiques AND et XOR. En effet, ces portes permettent de forcer une première opérande à 0 et d'inverser la seconde, ce qui permet à une addition de devenir une soustraction. Ainsi, au niveau du composant 'Add', l'entrée A peut avoir la valeur de A ou la valeur 0, et l'entrée B peut avoir la valeur de B ou not(B). De plus, pour la soustraction, nous utilisons également le bit de carry pour effectuer le complément à deux.

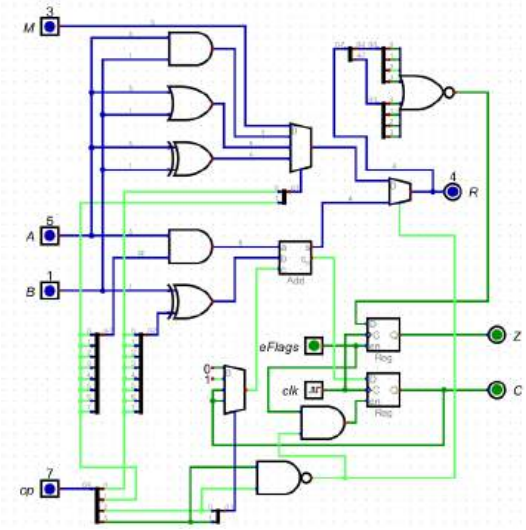
En haut du circuit logique, les 3 portes AND, OR et XOR permettent de réaliser les opérations logiques entre A et B. Ensuite, en fonction des bits 0 et 1 de l'opcode (correspondant donc aux bits a et i de la table 2), nous choisissons à l'aide d'un multiplexeur 4 vers 1 quel résultat envoyer vers la sortie. La première entrée du multiplexeur (M) permet de charger une valeur depuis la mémoire (ce qui correspond à l'opération ldi dans notre jeu d'instructions).

En haut à droite, nous avons utilisé une porte logique NOR de 8 entrées afin de vérifier si la valeur est de 0. Cela permet de mettre le flag Z à 1. De même, les deux registres Z et C permettent de stocker les flags. Ils sont tous les deux synchronisés sur la même clock.

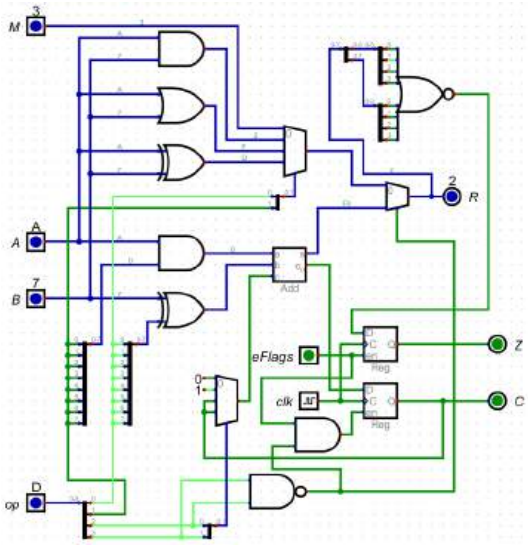
Lorsque nous simulons cet ALU sur digital, nous pouvons observer les signaux partout dans notre circuit. Il est possible de faire des scripts de tests automatiques ou de forcer les valeurs binaires de M, A, B et de l'opcode. Ci-dessous sont représentés 4 simulations des opérations d'additions, de soustractions, de chargement en mémoire et d'un ET logique :



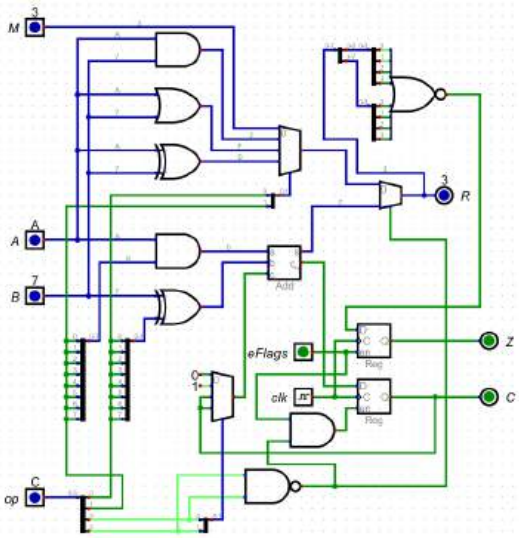
(a) Simulation de l'addition



(b) Simulation de la soustraction



(c) Simulation du ET logique



(d) Simulation de chargement (load)

FIGURE 6 – NCPU : Simulation de quatre instructions avec l'ALU

Nous pouvons vérifier ces résultats en bruts. En effet :

- Pour l'addition : `0xA` + `0x7` = 10 + 7 = 17 = `0x11`
- Pour la soustraction : 5 − 1 = 4
- Pour le ET logique :

$$\begin{array}{rll} & 1\,0\,1\,0 & (10) \\ \text{AND} & 0\,1\,1\,1 & (7) \\ \hline = & 0\,0\,1\,0 & (2) \end{array}$$

- Pour le chargement : $R = M = 3$

Ensuite, une fois ces résultats corrects obtenus, nous avons tester tout les cas d'instructions possibles avec des banc de tests automatiques.Une fois l'ALU terminée et fonctionnelle, nous avons crée le composant correspondant afin de l'intégrer dans le schéma final du processeur :

FIGURE 7 – NCPU : Composant ALU

Un professeur s'est ensuite chargé de la finalisation du processeur NCPU. Le schéma complet du NCPU est le suivant :

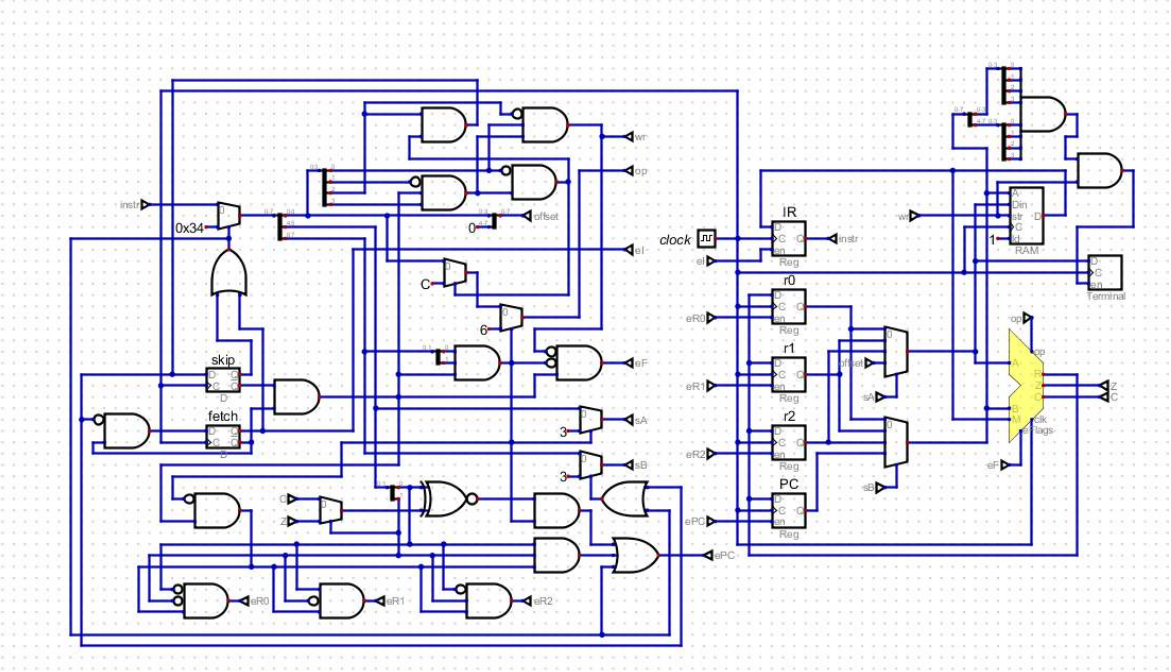


FIGURE 8 – NCPU : Processeur complet

Nous avons sur ce schéma final le chemin de données pour le processeur à droite, une RAM de 256 octets et une interface vers un terminal texte en haut au milieu. L'écriture à l'adresse `0xFF` enregistre en mémoire et envoi le caractère au terminal). Nous pouvons voir l'unité de contrôle à gauche avec notre ALU précédemment étudiée.

4.2 DRV16 : Un processeur 16 bit basé à partir de l'architecture *RISC-V RV32E*

DRV16, le second processeur que nous avons développé, est basé sur une architecture *RISC-V RV32E*. Il s'agit d'une version réduite de l'architecture *RISC-V RV32I*, principalement développée pour répondre aux besoins des systèmes embarqués. Les instructions utilisées par les deux architectures sont identiques (le jeu d'instructions est présenté en annexe A.4) ; la différence réside uniquement dans le nombre de registres. Pour l'architecture *RV32E*, seulement 16 registres, de x0 à x15, sont utilisés, tandis que l'architecture *RV32I* en utilise 32, de x0 à x31.

Ce futur processeur DRV16 a un double objectif. Premièrement, il servira à des fins éducatives pour permettre aux étudiants de découvrir comment fonctionne un processeur et comment le développer avec l'outil Digital. Ensuite, il visera à exécuter des jeux vidéo un peu plus évolués que ceux des bibliothèques NES et SNES, avec des graphismes améliorés. Pour tester cela, il faudra d'abord l'intégrer et travailler avec dans l'outil en ligne 8bitworkshop, présenté au début de ce rapport. Les objectifs du laboratoire sont de terminer ce projet avant la fin de l'année 2024.

Après notre première réunion à propos de ce processeur, nous avons appris d'un professeur que nous allons modifier la manière dont les instructions *RISC-V* sont encodées. Les instructions devront ainsi être codées de la manière suivante :

15 14 13 12	11 10 09 08	07 06 05 04	03 02 01 00
rD	rS1	rS2	opération

Table 4 – Format d'instructions utilisés par DRV16

Comme nous pouvons le constater, le format des instructions est de 16 bits, mais n'est pas celui compatible avec l'extension C de *RISC-V*. Nous devrons nous baser sur ce format pour le développement du processeur.

Comme le précédent processeur, mon travail se concentrera principalement sur le développement de certains composants de ce processeur, et je participerai à la réalisation des tests.

Après avoir compris les instructions qui m'ont été fournies, j'ai commencé par développer l'ALU sur Digital. La première version fonctionnelle est représentée ci-dessous :

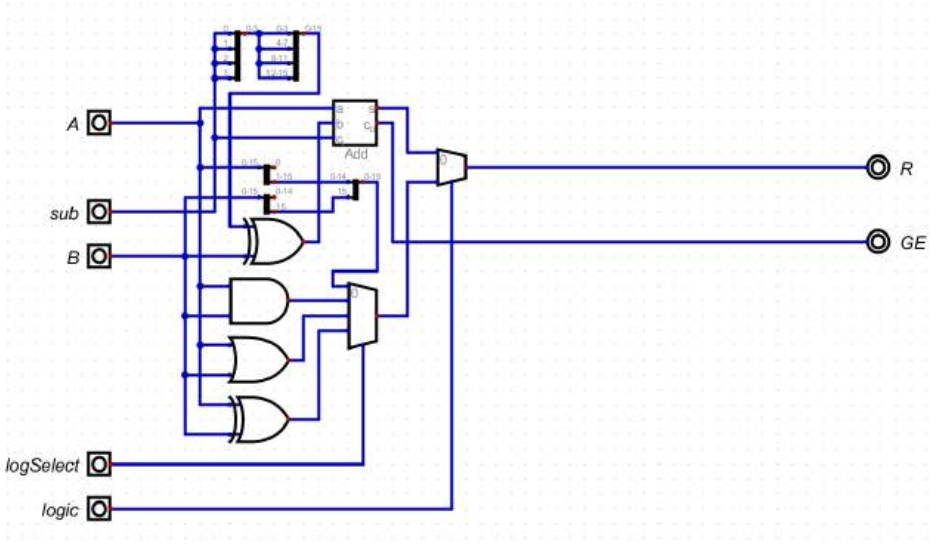


FIGURE 9 – DRV16 : Première version de l'ALU

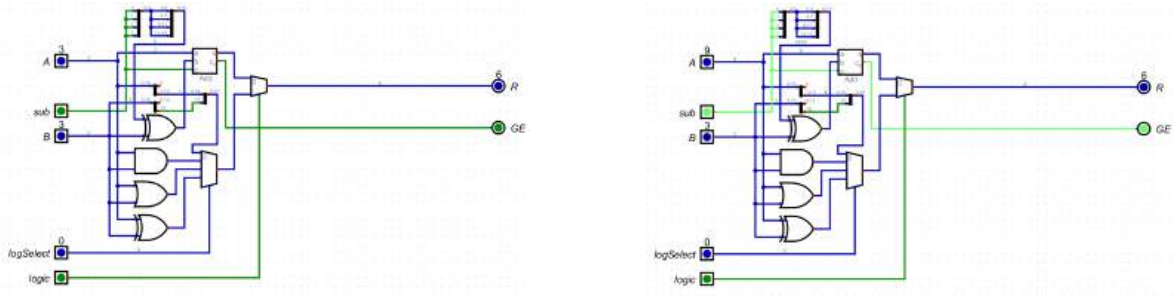
Sur ce bloc, A et B sont des bus de 16 bits qui contiennent les opérandes sur lesquels les opérations seront effectuées. Nous avons aussi comme entrée un signal sub qui contrôle la soustraction à travers l'entrée B. Lorsque sub est actif, tous les bits de B sont inversés via un XOR avec une série de 1 pour permettre une soustraction en utilisant un additionneur. Ainsi, l'inversion des bits et l'ajout du bit de report (sub) permet de réaliser la soustraction de $A - B$. Si sub est à 0, l'ALU effectue une addition classique, donnant A + B. Le signal de dépassement de l'additionneur est utilisé pour générer le signal GE (greater or equal), indiquant que A est supérieur ou égal à B lorsque la soustraction est exécutée, ou si un dépassement de capacité a lieu.

Les opérations logiques sont contrôlées par les signaux logSelect et logic. Si le signal logic est activé, l'ALU sélectionne l'une des quatre opérations logiques : AND, OR, XOR, ou un décalage arithmétique de A vers la droite. Un multiplexeur 4 vers 1 choisit l'opération à appliquer en fonction de la valeur de l'entrée logSelect. Ainsi :

- Si logSelect est à 00, nous effectuons un décalage arithmétique de A
- Si logSelect est à 01, l'opération AND entre A et B est sélectionnée
- Si logSelect est à 10, l'opération OR entre A et B est sélectionnée
- Si logSelect est à 11, l'opération XOR entre A et B est sélectionnée

Ces résultats logiques sont ensuite renvoyés à la sortie de résultat R si le signal logic est actif. Un multiplexeur 2 vers 1 permet de renvoyer soit le résultat de l'opération arithmétique (addition ou soustraction), soit le résultat de l'opération logique.

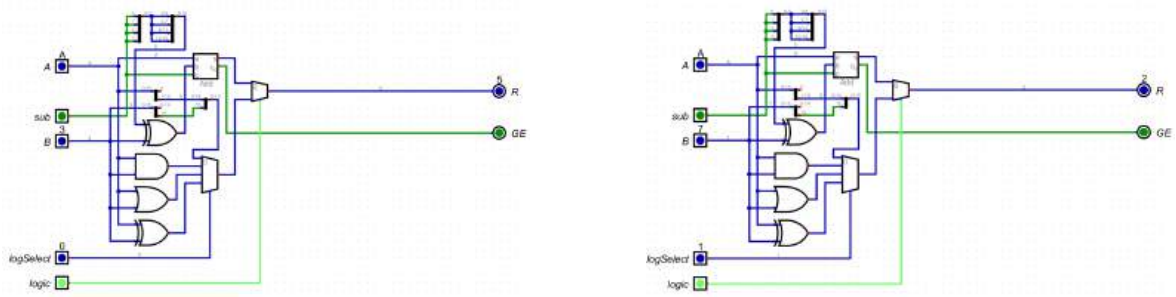
Lorsque nous simulons le circuit en forçant les valeurs, nous obtenons, pour les deux opérations arithmétiques :



(a) Simulation de l'addition (b) Simulation de la soustraction

FIGURE 10 – DRV16 : Simulations arithmétique de l'ALU

De même, pour les opérations logiques nous observons :



(a) Simulation du décalage sur A

(b) Simulation du ET logique

(c) Simulation du OU logique

(d) Simulation du XOR logique

FIGURE 11 – DRV16 : Simulations de logique de l'ALU

Sur ces deux simulations arithmétiques et logiques, nous avons bien :

- Si logic = 0 :
  - $3 + 3 = 6$
  - $9 - 3 = 6$
- Si logic = 1 :
  - Si A = 1010 (10), décalé de 1 vers la droite nous obtenons 0101 (5)
  - Pour le ET logique :

$$\begin{array}{rr} & 1\,0\,1\,0 \quad (10) \\ \text{AND} & 0\,1\,1\,1 \quad (7) \\ \hline = & 0\,0\,1\,0 \quad (2) \end{array}$$

- Pour le OR logique :

$$\begin{array}{rll} & 1\,0\,1\,0 & (10) \\ \text{OR} & 0\,1\,1\,1 & (7) \\ \hline = & 1\,1\,1\,1 & (15) \end{array}$$

- Pour le XOR logique :

$$\begin{array}{rll} & 1\,0\,1\,0 & (10) \\ \text{XOR} & 0\,1\,1\,1 & (7) \\ \hline = & 1\,1\,0\,1 & (13) \end{array}$$

Une fois cette première version terminée, on m'a proposé de rajouter un signal de sortie qui permet de savoir si les deux opérandes d'entrées A et B sont égales. Nous appellerons ce signal NE pour Not Equal. Il est possible de tester si A et B sont égaux en effectuant une soustraction. Si le résultat de la soustraction est 0, cela signifie que $A = B$. Pour réaliser ce contrôle binaire, nous allons récupérer la valeur dans le registre du résultat. Nous allons ensuite comparer les bits 4 par 4 à l'aide d'une porte logique OR. Ainsi, si seulement une des sorties d'une porte logique OR est à 1, cela signifie qu'il y a au moins un bit à 1, donc que le résultat n'est pas 0, et donc que l'opérande A est différente de l'opérande B. Le schéma mis à jour de l'ALU est donc le suivant :

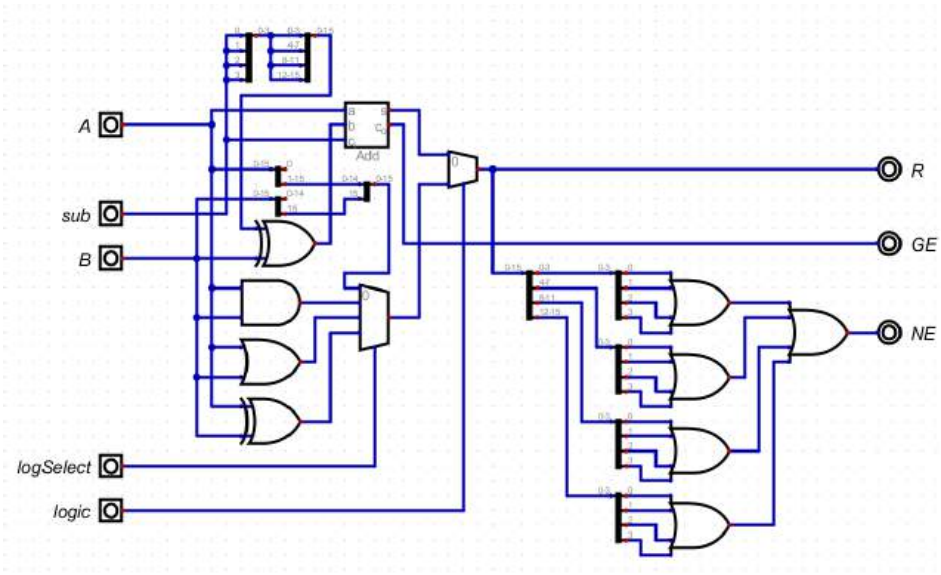
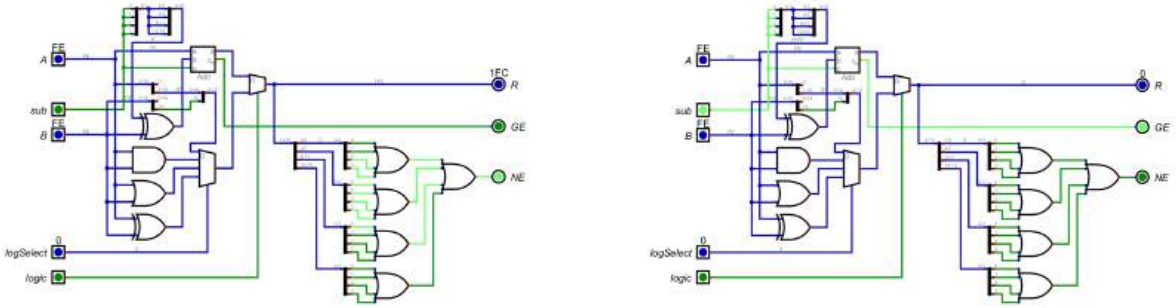


FIGURE 12 – DRV16 : Version finale de l'ALU

Nous pouvons ainsi tester le signal NE en essayant de soustraire deux nombre égaux. Nous visualisons :



(a) Soustraction de deux nombres A et B différents

(b) Soustraction de deux nombres A et B égaux

FIGURE 13 – DRV16 : Simulations du signal NE (Not Equal)

Nous pouvons constater que lorsque nous effectuons la soustraction de `0xFE` − `0xFE`, le signal NE n'est plus activé. De plus le signal GE (Greater or Equal), est quant à lui activé (car les deux entrées sont égales). Comme illustré sur la simulation ci-dessous, ce signal est aussi activé en cas de dépassement de capacité. Si nous essayons d'additionner `0xFFF0` + `0xFE`, nous visualisons :

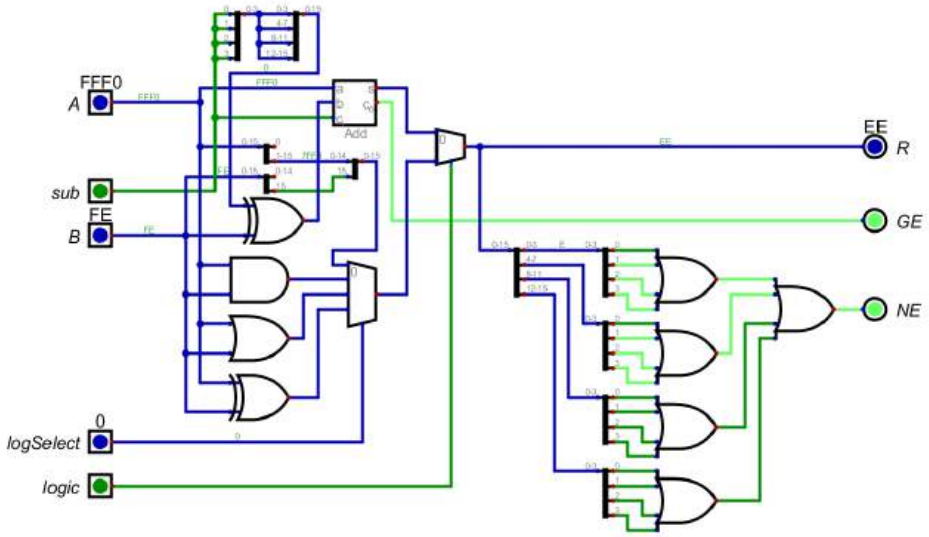


FIGURE 14 – DRV16 : Simulation du signal GE (Greater or Equal)

Le résultat de cette précédente addition ne tient pas sur 16 bits, il y a donc un dépassement de capacité et le signal GE est donc bien activé.

Après avoir terminé l'ALU, j'ai crée un composant très simple qui correspond au registre R_0. Ce dernier contient le compteur de programme (PC). Dans le format d'encodage des instructions, lorsque le champ rD est égal à zéro, aucun registre n'est modifié. De plus, si rS1 ou rS2 valent zéro, la valeur 0 est utilisée à la place de ce qui se trouve dans R_0. Ce registre est donc décrit comme suivant :

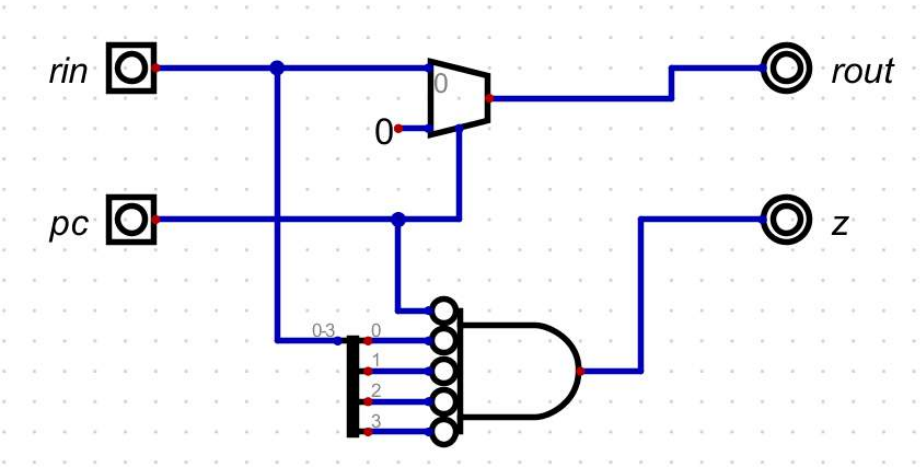
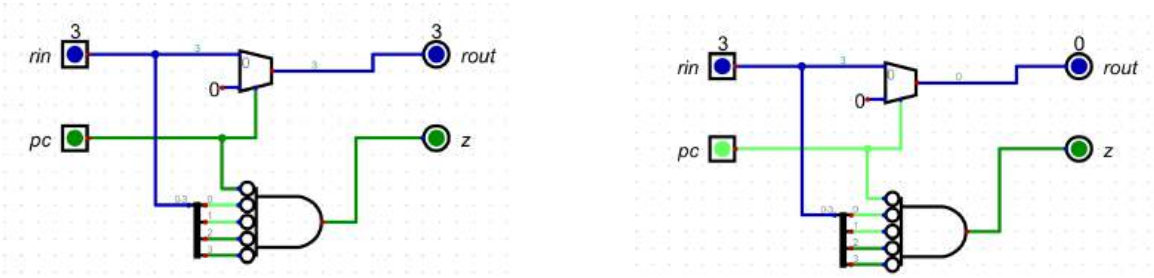


FIGURE 15 – DRV16 : Registre R_0

Ce registre R_0 permet à n'importe quel champs d'instruction d'être ignoré par le PC et indique si un traitement spécial (remplacer par 0 pour les sources et ne pas écrire pour la destination) est nécessaire. Si le registre d'entrée PC est activé, cela signifie que le registre de sortie sera 0. En simulant ce registre, nous visualisons :



(a) R_0 : $R_{out} = R_{in}$ (b) R_0 : $R_{out} = 0$

FIGURE 16 – DRV16 : Simulation du registre R_0 selon le bit PC

Nous visualisons que $R_{out} = R_{in}$ en l'absence du bit PC, sinon nous forçons R_{out} à 0. Nous avons également mis un flag zéro indiquant que le registre d'entrée R_{in} est à 0, comme illustré dans la simulation suivante :

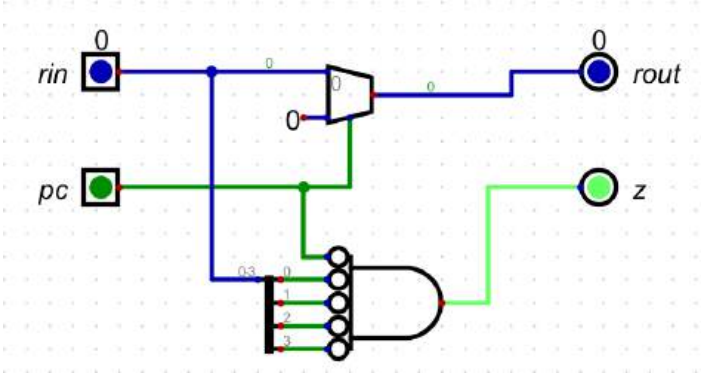


FIGURE 17 – DRV16 : Registre R_0 lorsque $R_{in} = 0$

Par ailleurs, le processeur DRV16 dispose également d'une unité de contrôle des instructions, mais celle-ci à été commencé en parallèle de mon travail par un professeur et n'a donc pas été développé par moi-même. Elle utilise cependant le registre R_0. Sa description est la suivante :

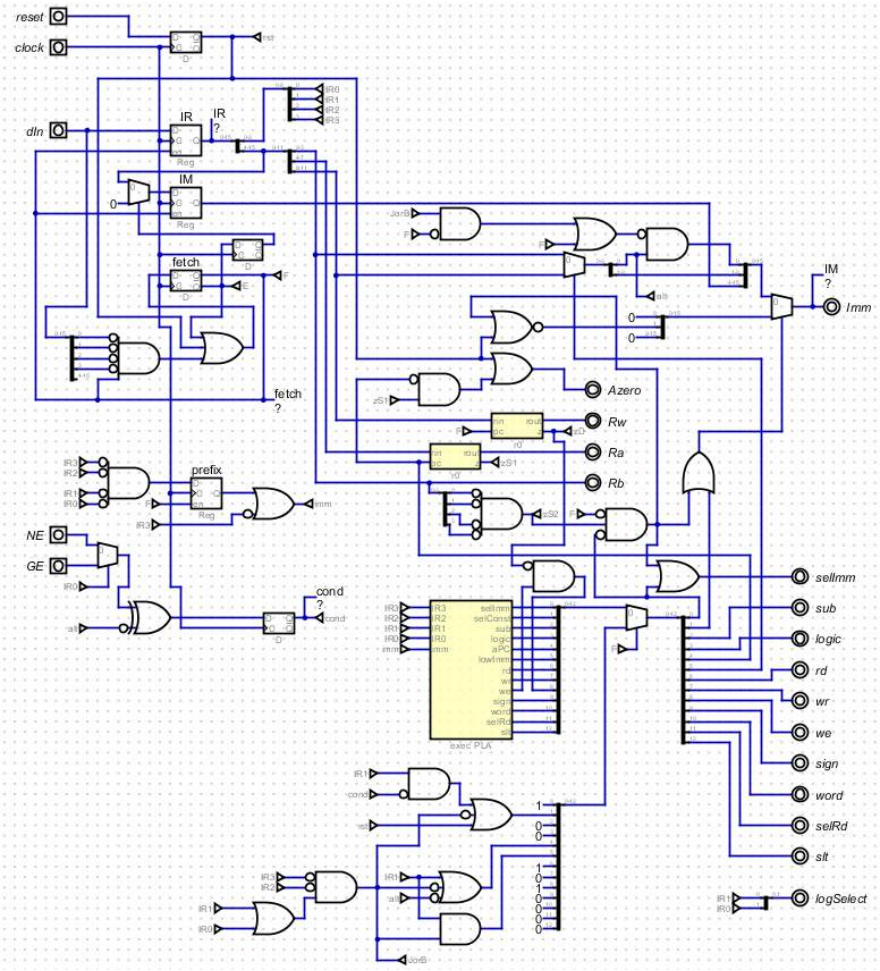


FIGURE 18 – DRV16 : Unité de contrôle

Avec cette unité de contrôle, les instructions selon la base *RISC-V RV32E* que le processeur peut exécuter sont les suivantes :

numéro	instruction	action	information
0		@IM := @IR, @IR := mem[@PC := @PC + 2]	Charge l'instruction suivante en mémoire.
1	JAL	@rD := @PC + 2	Sauvegarde l'adresse de retour (PC + 2) dans le registre destination.
1	JALR	@rD := @PC + 2	Sauvegarde l'adresse de retour (PC + 2) dans le registre destination.
2	BEQ	cond := $@rS1 = @rS2$	Condition : les deux registres sources sont égaux.
2	BNE	cond := $@rS1 \neq @rS2$	Condition : les deux registres sources sont différents.
3	BLT	cond := $@rS1 < @rS2$	Condition : le registre source 1 est inférieur au registre source 2.
3	BGE	cond := $@rS1 \geq @rS2$	Condition : le registre source 1 est supérieur ou égal au registre source 2.
4	LB	@rD := SignExtend(mem[$@rS1 + (@IM\|rS2)$])	Charge un octet signé en mémoire dans le registre destination.
5	LH	@rD := mem[$@rS1 + (@IM\|rS2)$]	Charge un mot signé en mémoire dans le registre destination.
6	SB	mem[$@rS1 + rD$] := 8Bits(@rS2)	Stocke un octet dans la mémoire à l'adresse calculée.
7	SH	mem[$@rS1 + rD$] := @rS2	Stocke un mot dans la mémoire à l'adresse calculée.
8	LBU	@rD := ZeroExtend(mem[$@rS1 + (@IM\|rS2)$])	Charge un octet non signé en mémoire dans le registre destination.
9	ADD	@rD := $@rS1 + @rS2$	Additionne deux registres et stocke le résultat dans le registre destination.
9	ADDI	@rD := $@rS1 + (@IM\|rS2)$	Additionne un registre et un immédiat et stocke le résultat dans le registre destination.

A	SUB	@rD := $@rS1 - @rS2$	Soustrait deux registres et stocke le résultat dans le registre destination.
A	SUBI	@rD := $@rS1 - (@IM\|rS2)$	Soustrait un registre d'un immédiat et stocke le résultat dans le registre destination.
B	SLT	@rD := $@rS1 < @rS2$	Le registre destination reçoit 1 si le premier registre est inférieur au second.
B	SLTI	@rD := $@rS1 < (@IM\|rS2)$	Le registre destination reçoit 1 si le registre est inférieur à l'immédiat.
C	SRS	@rD := $(@rS1 \gg 1)$ | $(@rS2\&\texttt{0x8000})$	Décale à droite et insère un bit de signe.
C	SRSI	@rD := $(@rS1 \gg 1)$ | $(@IM\&\texttt{0x8000})$	Décale à droite et insère un bit de signe à partir de l'immédiat.
D	AND	@rD := $@rS1\&@rS2$	Applique un ET logique entre deux registres.
D	ANDI	@rD := $@rS1\&(@IM\|rS2)$	Applique un ET logique entre un registre et un immédiat.
E	OR	@rD := $@rS1\|@rS2$	Applique un OU logique entre deux registres.
E	ORI	@rD := $@rS1\|(@IM\|rS2)$	Applique un OU logique entre un registre et un immédiat.
F	XOR	@rD := $@rS1 \oplus @rS2$	Applique un XOR logique entre deux registres.
F	XORI	@rD := $@rS1 \oplus (@IM\|rS2)$	Applique un XOR logique entre un registre et un immédiat.

Table 5 – Jeu d'instructions utilisé par DRV16

Nous pouvons noter que le registre PC contient l'adresse de l'instruction en cours pendant la phase d'exécution. Ainsi, l'adresse utilisée pendant le chargement est obtenue en ajoutant 2 au PC. Cela a pour conséquence que le décalage pour les instructions de branchement (comme JAL) se fait à partir de l'instruction actuelle et non de la suivante, contrairement à ce qui est habituel en *RISC-V*.

Cependant, avec un décalage de 16 bits au lieu de 12 bits, cela ne pose pas de problème. Ce problème ne concerne cependant pas JALR. En effet cette instruction n'a pas besoin de modification car le PC n'intervient pas dans son calcul.

Par ailleurs, le jeu d'instructions du DRV16 inclut une instruction SUBI, absente de *RV32E* (qui peut utiliser des constantes négatives avec ADDI). Il manque également certaines comparaisons non signées (comme SLTIU, SLTU, BLTU, BGEU). Les instructions LUI et AUI sont également absentes, car les constantes supérieures à 12 bits sont générées autrement et ne sont pas prises en compte avec notre processeurs.

Les opérations de décalage (comme SLLI, SRLI, SRAI, SLL, SRL, SRA) ne sont également pas implémentées car le matériel nécessaire serait trop volumineux par rapport au reste du processeur. Cependant, le décalage à gauche (SLLI x3,x4,3) peut être réalisé à l'aide de la séquence d'addition suivante :

ADD x3,x4,x4
ADD x3,x3,x3
ADD x3,x3,x3

Les décalages à droite sont implémentés avec l'instruction SRS (shift right step), qui n'est pas présente dans *RV32E*. Par exemple, SRAI x3,x4,3 peut être réalisé avec la séquence :

SRS x3,x4,x4
SRS x3,x3,x3
SRS x3,x3,x3

Tandis que SRLI x3,x4,3 devient :

SRS x3,x4,zéro
SRS x3,x3,zéro
SRS x3,x3,zéro

Enfin, les instructions ECALL et EBREAK, présentes dans la version originale *RV32E*, sont absentes de notre jeu d'instructions.

À la suite du développement de cette unité de contrôle et pour ma dernière semaine de stage, nous étions tous conviés pour concevoir la version finale du CPU. Ayant toujours en tête d'implémenter ce processeur dans l'émulateur en ligne 8bitworkshop, je me suis rapidement tourné vers cet outil afin de commencer la description du compilateur (à l'aide de JSON) comme présenté au début de ce rapport. Bien que le processeur ne soit pas terminé, connaissant le format des instructions et leurs interprétation, je pouvais commencer la description du compilateur.

Ainsi, en parallèle de mon travail, deux jours avant la fin de mon stage, la version finale du CPU m'a été présenté. Elle est illustrée ci-dessous :

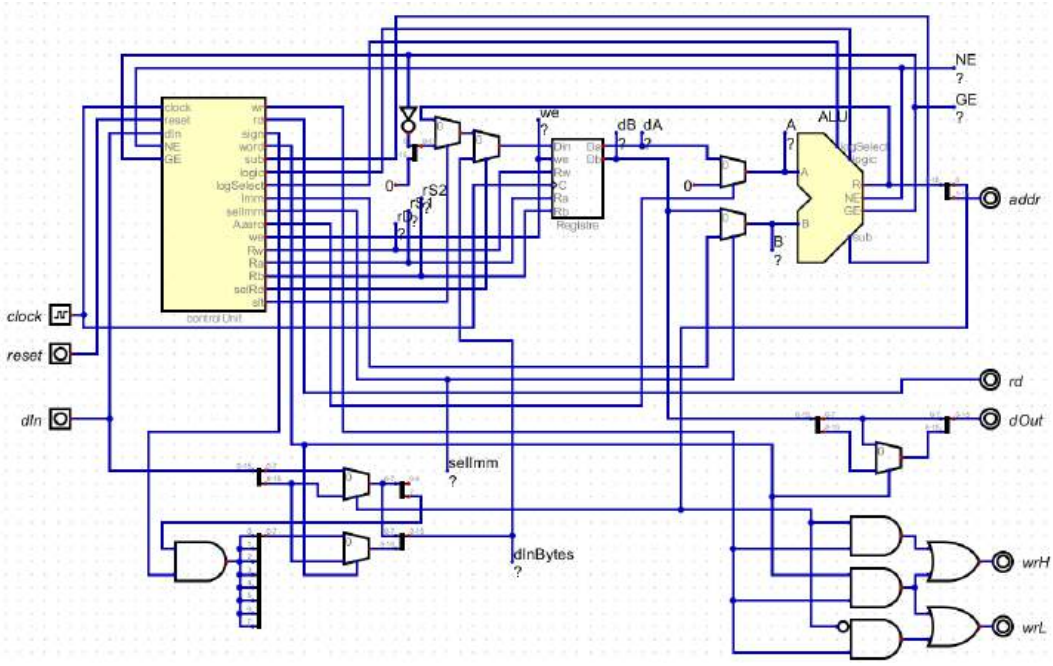


FIGURE 19 – DRV16 : Version complète du CPU

Enfin, pour la rédaction complète du compilateur pour 8bitworkshop, je suis parti d'une base déja présente sur github, permettant de compiler du code *RISC-V*. Cependant, la codification de nos instructions étant assez différente du set *RISC-V RV32E* de base, il a fallu apporter quelques modifications pour rendre ce compilateur compatible avec notre DRV16. En se basant sur le format des instructions décrits dans le tableau 4 et le jeu d'instructions décrit dans le tableau 5, le code adapté que j'ai développé est le suivant :

```
{
  "name":"riscv",
  "vars":{
    "reg":{"bits":4, "toks":["zero","x1","x2","x3","x4","x5","x6","x7",
                             "x8","x9","x10","x11","x12","x13","x14","x15"]},
    "brop":{"bits":3, "toks":["beq","bne","blt","bge"]},
    "imm12":{"bits":12}
  },
  "rules":[
    {"fmt":"jal ~reg,~imm12",        "bits":[0,1,"0001"]},
    {"fmt":"jalr ~reg,~reg,~imm12", "bits":[0,1,2,"0001"]},

    {"fmt":"beq ~reg,~reg,~imm12",  "bits":[0,1,2,"0010"]},
    {"fmt":"bne ~reg,~reg,~imm12",  "bits":[0,1,2,"0010"]},
    {"fmt":"blt ~reg,~reg,~imm12",  "bits":[0,1,2,"0011"]},
    {"fmt":"bge ~reg,~reg,~imm12",  "bits":[0,1,2,"0011"]},

    {"fmt":"lb ~reg,~reg,~imm12",   "bits":[0,1,2,"0100"]},
    {"fmt":"lh ~reg,~reg,~imm12",   "bits":[0,1,2,"0101"]},
    {"fmt":"sb ~reg,~reg,~imm12",   "bits":[0,1,2,"0110"]},
    {"fmt":"sh ~reg,~reg,~imm12",   "bits":[0,1,2,"0111"]},
    {"fmt":"lbu ~reg,~reg,~imm12",  "bits":[0,1,2,"1000"]},

    {"fmt":"add ~reg,~reg,~reg",    "bits":[0,1,2,"1001"]},
    {"fmt":"addi ~reg,~reg,~imm12", "bits":[0,1,2,"1001"]},
    {"fmt":"sub ~reg,~reg,~reg",    "bits":[0,1,2,"1010"]},
    {"fmt":"subi ~reg,~reg,~imm12", "bits":[0,1,2,"1010"]},

    {"fmt":"slt ~reg,~reg,~reg",    "bits":[0,1,2,"1011"]},
    {"fmt":"slti ~reg,~reg,~imm12", "bits":[0,1,2,"1011"]},
    {"fmt":"srs ~reg,~reg,~reg",    "bits":[0,1,2,"1100"]},
    {"fmt":"srsi ~reg,~reg,~reg",   "bits":[0,1,2,"1100"]},

    {"fmt":"and ~reg,~reg,~reg",    "bits":[0,1,2,"1101"]},
    {"fmt":"andi ~reg,~reg,~imm12", "bits":[0,1,2,"1101"]},
    {"fmt":"or ~reg,~reg,~reg",     "bits":[0,1,2,"1110"]},
    {"fmt":"ori ~reg,~reg,~imm12",  "bits":[0,1,2,"1110"]},
    {"fmt":"xor ~reg,~reg,~reg",    "bits":[0,1,2,"1111"]},
    {"fmt":"xori ~reg,~reg,~imm12", "bits":[0,1,2,"1111"]}
  ]
}
```

Dans le code ci-dessus, chaque instruction est spécifiée par un format (fmt) et un encodage binaire associé (bits). Le format utilise des variables comme ~reg pour les registres et ~imm12 pour un immédiat de 12 bits.

Les positions 0, 1 et 2 dans la section bits représentent les emplacements des trois registres utilisés dans l'instruction. Par exemple, pour l'instruction add ~reg,~reg,~reg, les trois registres impliqués (destination, source 1 et source 2) seront encodés dans ces positions. Le dernier champ, comme "1001" pour l'instruction ADD, représente le numéro de l'opération en binaire.

Pour les instructions immédiates comme ADDI et SUBI, la variable ~imm12 remplace l'un des registres, et l'immédiat est également encodé dans les positions correspondantes.

Pour tester ce code, il a donc fallu que j'attende la version finale du CPU afin de pouvoir exporter la schématique en Verilog et créer un projet dans 8bitworkshop. Une fois le CPU terminé, je l'ai intégré dans un nouveau projet. Il a fallu créer un nouveau module *test_CPU_top* et instancier le CPU DRV16 avec des signaux internes pour nos tests, avec les bons noms afin d'avoir une compatibilité dans 8bitworkshop.

À la fin, l'objectif est d'essayer d'exécuter le code assembleur suivant et de vérifier que les instructions fonctionne correctement :

```
1  inicio:
2          xor s0, s1, s1
3          bne s0,zero, inicio
4          addi a0,s0,128
5          la t0, temp
6          lh t1,t0,0
7          jal a1,.+2
8          subi a1,a1,2
9          auipc a0, 512
10         jalr ra,a1,0
11 temp: .word 0
```

Ce programme assembleur est essentiellement un test de base de certaines instructions du processeur. Il ne correspond pas à un algorithme particulier mais il permet d'initialiser des registres et de travailler avec une petite boucle conditionnelle (BNE). Il effectue des sauts relatifs (JAL et JALR), ce qui permet de tester les opérations de branchement et de retour. Il manipule des adresses mémoires (à l'aide de temp) et effectue des lectures et écritures simples. Nous testons aussi des instructions arithmétiques et logiques simples comme ADDI, SUBI, et XOR.

Dans notre cas, nous ne pouvions pas utiliser les registres numérotés par s, mais uniquement les équivalences numérotés par x. De plus, certaines instructions nécessitaient des macros, ce qui est le cas par exemple pour l'instruction la, pour charger une adresse, mais ces dernières ne sont pas prises en charge par 8bitworkshop. Cela donnait donc des erreurs. En retirant les lignes comportant des erreurs dans l'assembleur, nous avons pu visualiser les signaux précédemment décrits. Malheureusement, ces derniers semblaient aléatoire. Avec mes collègues et les professeurs, nous pensons qu'il y a certains signaux et certains réglages propre à 8bitworkshop que nous devons modifier ou instancier afin de pouvoir mieux contrôler nos données à l'intérieur de cet émulateur. Au dernier jour de mon stage, nous ne les avons pas trouvés.

5 Conclusion

Pour conclure ce stage, ce dernier m'a permis de mettre en avant et en application les compétences en développement de processeurs que j'ai acquises lors de ma formation à Polytech Sorbonne. Mes compétences précédentes ont été plus que nécessaire dans ce stage, pour réfléchir de la bonne manière et ainsi pour pouvoir présenter des solutions fonctionnelles. Le projet d'intégration de ce nouveau processeur dans la carte TANG NANO 20K est toujours d'actualité et les objectifs sont de pouvoir émuler des jeux vidéos rétro avant le début de l'année 2025.

Le processeur DRV16 est le processeur qui a demandé le plus de temps de développement, par sa complexité, et par le fait de devoir s'habituer à la description de CPU sur Digital en représentant directement les circuits de logique numérique, sans écrire de VHDL ou de Verilog comme nous avons précédent appris.

Notre processeur DRV16 a également fait l'objet d'une publication scientifique "*drv16 : An educational RISC-V based 16 bit processor*" [5]. Ce fût une fierté de travailler sur ce projet et d'avoir pour la première fois mon nom dans une publication scientifique.

6 Références

[1] **8bitworkshop.** *8bitworkshop - Online Retro Game Development.* Disponible en : https://8bitworkshop.com/.

[2] Jecel Mattos de Assumpção Jr, Chandler Klüser, Victoria Alejandra Salazar Herrera, João Paulo Carmo, et Mario Gazziro. *Computadores e videogames : uma abordagem prática das arquiteturas clássicas.* UFABC, 2024. Disponible en : http://professor.ufabc.edu.br/~mario.gazziro/project30.html.

[3] **hnemann.** *Digital.* Disponible en : https://github.com/hneemann/Digital.

[4] **cpldcpu.** *MCPU.* Disponible en : https://github.com/cpldcpu/MCPU.

[5] Lucas Gauvain, Jecel Mattos de Assumpção Junior, Oswaldo Hideo Ando Junior, Hugo Puertas de Araújo, Marco Roberto Cavallari, João Paulo Carmo, Mario Gazziro *drv16 : An educational RISC-V based 16 bit processor.* MDPI. 2024. Disponible en : https://susy.mdpi.com/user/manuscripts/review_info/17bb1ca9540a29b9a472583109137f56.

A Annexes

A.1 Fiche d'auto-évaluation du stage

Evaluation de stage

Prénom et nom de l'élève ingénieur:	**Lucas Gauvain**
Spécialité de la formation:	**Electronique et Informatique, parcours Systèmes Embarqués (EI-SE)**
Nom et adresse de la société:	**LAOB - UFABC, Santo André, SP**
Prénom et nom du tuteur industriel:	**Mario Gazziro**

*Cette grille d'auto-évaluation renseignée par l'**élève ingénieur** rend compte des compétences démontrées par ce dernier au cours du stage*
Le code couleur ROUGE indique que le niveau requis **n'est pas** atteint pour cette compétence
Le code couleur VERT indique que le niveau requis **est atteint** pour cette compétence

Compétences techniques

Analyser et établir des solutions techniques et économiques pour la réalisation d'un projet	Même avec de l'aide, l'apprenant n'est pas capable de mettre en œuvre des outils et méthodes	Avec de l'aide, l'apprenant met en œuvre des outils et méthodes, sans s'interroger sur leur pertinence	L'apprenant propose des outils et méthodes adaptés. En autonomie, il les met en œuvre	L'apprenant choisit, adapte et met en œuvre des méthodes et outils en justifiant ses choix	Le stage ne mobilise pas cette compétence
Elément(s) de preuve	*Afin de réaliser les missions qui m'ont été confiées, j'ai réfléchis moi même à des méthodes et je les ai présentées à mes collègues.*				
Formaliser un problème en proposant une réflexion approfondie	L'apprenant n'est pas capable de formaliser un problème	Avec de l'aide, l'apprenant est capable de formaliser un problème	En autonomie, l'apprenant est capable de formaliser un problème	L'apprenant est capable de formaliser un problème et d'avoir une réflexion approfondie sur son choix de formalisation	Le stage ne mobilise pas cette compétence
Elément(s) de preuve	*Pour certains problèmes que j'ai rencontrés, j'en ai chercher la cause, je les ai étudiés en profondeur et j'ai essayés de les résoudre.*				
Piloter un projet en utilisant les méthodes et outils de gestion de projet	L'apprenant n'est pas capable de mettre en œuvre une démarche de gestion de projet	L'apprenant met partiellement en œuvre une démarche de gestion de projet	Avec de l'aide, l'apprenant met en œuvre une démarche de gestion de projet	En autonomie, l'apprenant met en œuvre une démarche de gestion de projet	Le stage ne mobilise pas cette compétence
Elément(s) de preuve	*Lors de mes missions, je me fixais des délais pour réaliser certaines tâches afin de pouvoir terminer les projets à temps. Je classais aussi chaque tâche à faire.*				
Capacité à trouver l'information pertinente et à l'exploiter	L'apprenant ne cherche pas d'informations	Avec de l'aide, l'apprenant trouve des informations pertinentes et les exploite	En autonomie, l'apprenant trouve des informations pertinentes. Avec de l'aide, il les exploite	En autonomie, l'apprenant trouve des informations pertinentes et les exploite	Le stage ne mobilise pas cette compétence
Elément(s) de preuve	*J'ai effectuer beaucoup de recherches par moi même, notamment sur internet et sur Github.*				

Intégration dans une équipe

S'intégrer dans un collectif existant	L'apprenant ne respecte pas les règles et les codes (horaires, présentation...)	L'apprenant se contente de respecter les règles et codes (horaires, présentation...)	L'apprenant participe par son action à la dynamique de son service	L'apprenant participe par son action à la diffusion de la culture d'entreprise	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*J'ai veillé à adopter et à promouvoir les valeurs du laboratoire. J'ai participé à toutes les réunions d'équipe sur le projet afin de maintenir une forte collaboration harmonieuse.*				
Interagir avec toutes les parties prenantes et mobiliser les services nécessaires	L'apprenant n'est pas capable d'interagir avec ses encadrants	L'apprenant interagit avec ses encadrants mais ne cherche pas d'aide extérieur	L'apprenant interagit avec ses encadrants et, avec de l'aide, mobilise les services nécessaires	En autonomie, l'apprenant interagit avec toutes les parties prenantes et mobilise les services nécessaires	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*Je n'ai pas hésité à interagir avec tous mes collègues et responsables. J'ai ainsi établi des relations de travail efficaces en sollicitant leur expertise lorsque nécessaire. Cela à permis de mobiliser les ressources nécessaires pour mener à bien ma mission sans avoir une directives spécifique (comme lors d'un TP par exemple).*				
Savoir être responsable, réactif et positif face à une demande	L'apprenant n'est pas responsable et réactif face à une demande	L'apprenant est peu responsable et réactif face à une demande	L'apprenant est responsable et réactif face à une demande	L'apprenant est responsable, réactif et positif face à une demande	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*Si l'on me demande de faire une tâche, un code, ou de me renseigner sur quelque chose, je vais essayer de le faire le plus rapidement possible sans perturber le travail en cours afin de fournir rapidement un bilan.*				
Contribuer à la stratégie de l'entreprise et collaborer à sa mise en œuvre	L'apprenant n'a pas connaissance de la stratégie de l'entreprise	L'apprenant a connaissance de la stratégie de l'entreprise mais ne participe pas à sa mise en œuvre	Lorsqu'il est sollicité, l'apprenant participe à la mise en œuvre de la stratégie de l'entreprise	En pleine autonomie, l'apprenant participe à la mise en œuvre de la stratégie de l'entreprise	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*J'ai pris des initiatives pour aligner mon travail avec la stratégie globale du labo de recherche, en participant à des projets qui compte beaucoup pour les professeurs travaillant dans ce laboratoire. J'ai ainsi su agir de manière autonome dans cette stratégie de réussite.*				

RSE, éthique, réglementation

Gérer les relations au travail en présentiel ou à distance, en termes de responsabilité, de sécurité et de santé	L'apprenant ne sait pas gérer les relations au travail en termes de responsabilité, de sécurité et de santé	L'apprenant sait peu gérer les relations au travail en termes de responsabilité, de sécurité et de santé	Avec de l'aide, l'apprenant gère les relations au travail en termes de responsabilité, de sécurité et de santé	En autonomie, l'apprenant gère les relations au travail en termes de responsabilité, de sécurité et de santé	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*J'ai su gérer de manière autonome les relations professionnelles en veillant à respecter les normes de sécurité, de santé et de responsabilité. J'ai pris soin de suivre les protocoles en place, notamment en portant des équipements de sécurité lorsque c'était nécessaire dans certains laboratoire. Mon approche m'a alors permis de maintenir un environnement de travail optimal.*				
Appliquer une démarche respectant les enjeux environnementaux et les besoins de la société (RSE)	L'apprenant n'a pas conscience de ces enjeux	L'apprenant a conscience de ces enjeux mais ne les prend pas en compte	L'apprenant prend en compte ces enjeux dans sa mission	L'apprenant prend en compte ces enjeux au-delà de sa mission	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*A côté de mon stage, j'ai également agis en pensant à la RSE et au DDRS, en triant les déchets, limitant la consommation de papier inutile, limitant la consommation d'eau abondante le plus possible, etc...*				
Appliquer l'éthique, les normes et les réglementations propres à son secteur d'activités	L'apprenant n'a pas conscience de ces réglementations	L'apprenant a conscience de ces réglementations mais ne les prend pas en compte	L'apprenant prend en compte ces réglementations dans sa mission	L'apprenant prend en compte ces réglementations au-delà de sa mission	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*Je prends en compte ces réglementations dans la vie de tous les jours dans le laboratoire dans lequel j'étais.*				

Communication

Communiquer à l'écrit de façon professionnelle, structurée et synthétique	Les documents de l'apprenant contiennent de nombreuses erreurs. Ils ne sont pas exploitables	Avec de l'aide: l'apprenant produit des documents exploitables en interne	En autonomie: l'apprenant produit des documents exploitables en interne	En autonomie: l'apprenant produit des documents exploitables et diffusables	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*J'ai rédigé des bilans de réunion pour mes collègues. J'ai aussi écrit des notes de fonctionnements de certains outils ou de certaines technologies.*				
Communiquer à l'oral de manière pédagogique, synthétique et adaptée	L'apprenant ne sait pas communiquer à l'oral sur ses missions de manière claire et synthétique	L'apprenant ne sait pas communiquer à l'oral sur ses missions de manière claire et synthétique mais arrive, en discutant, à faire passer ses idées	En autonomie, l'apprenant sait communiquer à l'oral sur ses missions de manière claire et synthétique	L'apprenant sait communiquer à l'oral sur ses missions de manière claire et synthétique, en adoptant un comportement oral et non verbal professionnel	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*J'ai su présenter clairement mes missions, à des étudiants ou des visiteurs. En faisant cela, j'ai pris soin d'adopter un comportement professionnel.*				
Convaincre ou faire passer des idées pour aider à la prise de décision	L'apprenant n'est pas capable de défendre ses idées	Avec de l'aide et l'appui des membre de son équipe, l'apprenant est capable de défendre ses idées	En autonomie, l'apprenant est capable de défendre ses idées	L'apprenant est capable de défendre ses idées et sait conduire une discussion amenant à la prise de décision	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*Lors des réunions ou de discussions, j'ai su proposer mes idées et les expliquer. J'ai aussi donner mon avis sur certaines propositions et justifier leur faisabilité.*				
Donner ses retours, entendre et intégrer ceux des autres	L'apprenant n'est pas capable de donner des retours sur ses missions et d'écouter les conseils des membres de son équipe	Avec de l'aide, l'apprenant est capable de donner des retours sur ses missions et d'écouter les conseils des membres de son équipe	En autonomie, l'apprenant est capable de donner des retours sur ses missions et d'écouter les conseils des membres de son équipe	L'apprenant est capable de donner des retours sur ses missions, d'écouter les conseils des membres de son équipe, et de solliciter des rendez vous lorsque c'est nécessaire	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*J'ai régulièrement donné des retours sur mes missions et sur l'avancement de mon travail. J'ai pris en compte les conseils que l'on me donnait. Il m'est arrivé de prendre l'initiative afin de solliciter des rendez-vous avec mon maitre de stage pour discuter de points, afin de m'assurer que l'avancement se déroulait bien*				

International

Maitriser une ou plusieurs langues étrangères, aussi bien à l'écrit qu'à l'oral	L'apprenant ne maitrise pas de langues étrangères	L'apprernant a un faible niveau en langue étrangère	L'apprenant arrive à échanger dans une langue étrangère	L'apprenant a une communication fluide en langue étrangère	Le stage ne mobilise pas cette compétence
Élément(s) de preuve	*Je n'ai pas eu de problème pour parler avec mes collègues en anglais, et en portugais assez simple avec certains. Je suis toujours compris et je comprends ce que l'on me dis*				
Comprendre et appliquer les méthodes de travail et les réglementations adaptées aux contextes locaux	L'apprenant ne sait pas s'adapter au contexte local	Avec de l'aide, l'apprenant comprend et applique les méthodes de travail et les réglementations adaptées au contexte local	En autonomie, l'apprenant comprend et applique les méthodes de travail et les réglementations adaptées au contexte local	L'apprenant comprend et applique les méthodes de travail et les réglementations adaptées au contexte local avec enthousiasme et curiosité	Le stage ne mobilise pas cette compétence

Élément(s) de preuve	*Les méthodes de travailles qui m'ont été exposées ont été appliquées sans problèmes, pour le travail pour le maintient d'une bonne entente avec mes collègues.*

A.2 Code Verilog du femto8

```
`ifndef ALU_H
`define ALU_H

// ALU operations
`define OP_ZERO          4'h0
`define OP_LOAD_A    4'h1
`define OP_INC       4'h2
`define OP_DEC       4'h3
`define OP_ASL       4'h4
`define OP_LSR       4'h5
`define OP_ROL       4'h6
`define OP_ROR       4'h7
`define OP_OR        4'h8
`define OP_AND       4'h9
`define OP_XOR       4'ha
`define OP_LOAD_B    4'hb
`define OP_ADD       4'hc
`define OP_SUB       4'hd
`define OP_ADC       4'he
`define OP_SBB       4'hf

// ALU module
module ALU(A, B, carry, aluop, Y);

  parameter N = 8;  // default width = 8 bits
  input  [N-1:0] A; // A input
  input  [N-1:0] B; // B input
  input  carry;           // carry input
  input  [3:0] aluop;     // alu operation
  output reg [N:0] Y;     // Y output + carry

  always @(*)
    case (aluop)
      // unary operations
      `OP_ZERO:          Y = 0;
      `OP_LOAD_A:    Y = {1'b0, A};
      `OP_INC:       Y = A + 1;
      `OP_DEC:       Y = A - 1;
      // unary operations that generate and/or use carry
      `OP_ASL:       Y = {A, 1'b0};
      `OP_LSR:       Y = {A[0], 1'b0, A[N-1:1]};
      `OP_ROL:       Y = {A, carry};
      `OP_ROR:       Y = {A[0], carry, A[N-1:1]};
      // binary operations
      `OP_OR:        Y = {1'b0, A | B};
      `OP_AND:       Y = {1'b0, A & B};
      `OP_XOR:       Y = {1'b0, A ^ B};
      `OP_LOAD_B:    Y = {1'b0, B};
      // binary operations that generate and/or use carry
      `OP_ADD:       Y = A + B;
      `OP_SUB:       Y = A - B;
      `OP_ADC:       Y = A + B + (carry?1:0);
      `OP_SBB:       Y = A - B - (carry?1:0);
    endcase

endmodule

// destinations for COMPUTE instructions
`define DEST_A    2'b00
`define DEST_B    2'b01
`define DEST_IP   2'b10
`define DEST_NOP 2'b11

// instruction macros
`define I_COMPUTE(dest,op) { 2'b00, (dest), (op) }
`define I_COMPUTE_IMM(dest,op) { 2'b01, (dest), (op) }
`define I_COMPUTE_READB(dest,op) { 2'b11, (dest), (op) }
`define I_CONST_IMM_A { 2'b01, `DEST_A, `OP_LOAD_B }
`define I_CONST_IMM_B { 2'b01, `DEST_B, `OP_LOAD_B }
`define I_JUMP_IMM { 2'b01, `DEST_IP, `OP_LOAD_B }
`define I_STORE_A(addr) { 4'b1001, (addr) }
`define I_BRANCH_IF(zv,zu,cv,cu) { 4'b1010, (zv), (zu), (cv), (cu) }
`define I_CLEAR_CARRY { 8'b10001000 }
`define I_SWAP_AB { 8'b10000001 }
`define I_RESET { 8'b10111111 }
// convenience macros
`define I_ZERO_A `I_COMPUTE(`DEST_A, `OP_ZERO)
`define I_ZERO_B `I_COMPUTE(`DEST_B, `OP_ZERO)
`define I_BRANCH_IF_CARRY(carry) `I_BRANCH_IF(1'b0, 1'b0, carry, 1'b1)
`define I_BRANCH_IF_ZERO(zero) `I_BRANCH_IF(zero, 1'b1, 1'b0, 1'b0)
`define I_CLEAR_ZERO `I_COMPUTE(`DEST_NOP, `OP_ZERO)

module CPU(clk, reset, address, data_in, data_out, write);

  input            clk;
  input            reset;
  output reg [7:0] address;
  input      [7:0] data_in;
  output reg [7:0] data_out;
  output reg       write;

  reg [7:0] IP;
  reg [7:0] A, B;
  reg [8:0] Y;
  reg [2:0] state;

  reg carry;
  reg zero;
  wire [1:0] flags = { zero, carry };

  reg [7:0] opcode;
  wire [3:0] aluop = opcode[3:0];
  wire [1:0] opdest = opcode[5:4];
  wire B_or_data = opcode[6];

  localparam S_RESET = 0;
  localparam S_SELECT = 1;
  localparam S_DECODE = 2;
  localparam S_COMPUTE = 3;
  localparam S_READ_IP = 4;

  ALU alu(
    .A(A),
    .B(B_or_data ? data_in : B),
    .Y(Y),
    .aluop(aluop),
    .carry(carry));

  always @(posedge clk)
    if (reset) begin
      state <= 0;
      write <= 0;
    end else begin
      case (state)
        // state 0: reset
        S_RESET: begin
          IP <= 8'h80;
          write <= 0;
          state <= S_SELECT;
        end
      // state 1: select opcode address
        S_SELECT: begin
          address <= IP;
          IP <= IP + 1;
          write <= 0;
          state <= S_DECODE;
        end
        // state 2: read/decode opcode
        S_DECODE: begin
          opcode <= data_in; // (only use opcode next cycle)
          casez (data_in)
            // ALU A + B -> dest
            8'b00??????: begin
              state <= S_COMPUTE;
            end
            // ALU A + immediate -> dest
            8'b01??????: begin
            address <= IP;
```

```
                IP <= IP + 1;
              state <= S_COMPUTE;
            end
            // ALU A + read [B] -> dest
            8'b11??????: begin
              address <= B;
              state <= S_COMPUTE;
            end
            // A -> write [nnnn]
            8'b1001????: begin
              address <= {4'b0, data_in[3:0]};
              data_out <= A;
              write <= 1;
              state <= S_SELECT;
            end
            // swap A,B
            8'b10000001: begin
              A <= B;
              B <= A;
              state <= S_SELECT;
            end
            // conditional branch
            8'b1010????: begin
              if (
                (data_in[0] && (data_in[1] == carry)) ||
                (data_in[2] && (data_in[3] == zero)))
              begin
                address <= IP;
                state <= S_READ_IP;
              end else begin
                state <= S_SELECT;
              end
              IP <= IP + 1; // skip immediate
            end
            // fall-through RESET
            default: begin
              state <= S_RESET; // reset
            end
          endcase
        end
        // state 3: compute ALU op and flags
        S_COMPUTE: begin
          // transfer ALU output to destination
          case (opdest)
            `DEST_A: A <= Y[7:0];
            `DEST_B: B <= Y[7:0];
            `DEST_IP: IP <= Y[7:0];
            `DEST_NOP: ;
          endcase
          // set carry for certain operations (4-7,12-15)
          if (aluop[2]) carry <= Y[8];
          // set zero flag
          zero <= ~|Y[7:0];
          // repeat CPU loop
          state <= S_SELECT;
        end
        // state 4: read new IP from memory (immediate mode)
        S_READ_IP: begin
          IP <= data_in;
          state <= S_SELECT;
        end
      endcase
    end

endmodule

`ifdef TOPMOD__test_CPU_top

module test_CPU_top(
  input  clk,
  input  reset,
  output [7:0] address_bus,
  output reg [7:0] to_cpu,
  output [7:0] from_cpu,
  output write_enable,
  output [7:0] IP,
  output [7:0] A,
  output [7:0] B,
  output zero,
  output carry
);

  reg [7:0] ram[0:127];
  reg [7:0] rom[0:127];

  assign IP = cpu.IP;
  assign A = cpu.A;
  assign B = cpu.B;
  assign zero = cpu.zero;
  assign carry = cpu.carry;

  CPU cpu(.clk(clk),
          .reset(reset),
          .address(address_bus),
          .data_in(to_cpu),
          .data_out(from_cpu),
          .write(write_enable));

  always @(posedge clk)
    if (write_enable) begin
      ram[address_bus[6:0]] <= from_cpu;
    end

  always @(*)
    if (address_bus[7] == 0)
      to_cpu = ram[address_bus[6:0]];
    else
      to_cpu = rom[address_bus[6:0]];

  initial begin
`ifdef EXT_INLINE_ASM
    // example code: Fibonacci sequence
    rom = '{
      __asm

.arch femto8
.org 128
.len 128

Start:
      zero A   ; A <= 0
      ldb #1   ; B <= 1
Loop:
      add A,B  ; A <= A + B
      swapab   ; swap A,B
      bcc Loop ; repeat until carry set
      reset    ; end of loop; reset CPU

      __endasm
    };
`endif
  end

endmodule

`endif

`endif
```

A.3 Compilateur du femto8

```
1
2  {
3      "name":"femto8",
4      "width":8,
5      "vars":{
6        "reg":{"bits":2, "toks":["a", "b", "ip", "none"]},
7        "unop":{"bits":3, "toks":["zero","loada","inc","dec","asl",
8                                  "lsr","rol","ror"]},
9        "binop":{"bits":3, "toks":["or","and","xor","mov","add","sub",
10                                  "adc","sbb"]},
11       "const4":{"bits":4},
12       "imm8":{"bits":8}
13     },
14     "rules":[
15       {"fmt":"~binop ~reg,b",       "bits":["00",1,"1",0]},
16       {"fmt":"~binop ~reg,#~imm8",  "bits":["01",1,"1",0,2]},
17       {"fmt":"~binop ~reg,[b]",     "bits":["11",1,"1",0]},
18       {"fmt":"~unop ~reg",          "bits":["00",1,"0",0]},
19       {"fmt":"mov ~reg,[b]",        "bits":["11",0,"1011"]},
20       {"fmt":"zero ~reg",           "bits":["00",0,"1011"]},
21       {"fmt":"lda #~imm8",          "bits":["01","00","1011",0]},
22       {"fmt":"ldb #~imm8",          "bits":["01","01","1011",0]},
23       {"fmt":"jmp ~imm8",           "bits":["01","10","1011",0]},
24       {"fmt":"sta ~const4",         "bits":["1001",0]},
25       {"fmt":"bcc ~imm8",           "bits":["1010","0001",0]},
26       {"fmt":"bcs ~imm8",           "bits":["1010","0011",0]},
27       {"fmt":"bz ~imm8",            "bits":["1010","1100",0]},
28       {"fmt":"bnz ~imm8",           "bits":["1010","0100",0]},
29       {"fmt":"clc",                 "bits":["10001000"]},
30       {"fmt":"swapab",              "bits":["10000001"]},
31       {"fmt":"reset",               "bits":["10001111"]}
32     ]
33   }
```

A.4 Jeu d'instructions *RISC-V RV32I*

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred | succ		00000	000	00000	0001111	FENCE
0000 | 0000 | 0000		00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

La documentation complète *RISC-V* est disponible au lien suivant :
https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf