

---

**UE PSESI MU4IN110**  
**Compréhension, Implémentation, Optimisation du**  
**processeur CVA6 basé sur une architecture**  
**RISC-V**

---



*Auteurs- :*

LEGOUEIX NICOLAS  
QUERIC YANN  
PESHOVA BOJANA  
KANDIAH SUVETHA  
MOHAMMAD WAKEEL

*Enseignants Encadrant :*

M. B.GRANADO  
Mme. N.RAVIDAT

# Table des matières

<b>1 Projet et Architecture CVA6</b>	<b>2</b>
1.1 Concours et organisation du travail . . . . .	2
1.2 Compréhension de l'architecture du CVA6 . . . . .	3
1.2.1 Origines du CVA6 . . . . .	3
1.2.2 Présentation des composants du CVA6 . . . . .	4
<b>2 Environnement de travail</b>	<b>6</b>
2.1 Pré requis et préparation . . . . .	6
2.1.1 Préparation de l'environnement de travail et de compilation . . . . .	6
2.1.2 Problèmes d'installation rencontrés et solutions possibles . . . . .	7
2.2 Génération du bitstream CVA6, chargement sur la carte et exécution des programmes sur le CVA6 . . . . .	8
2.3 Résultats obtenus lors de la synthèse du CVA6 . . . . .	11
2.3.1 Consommation énergétique . . . . .	11
2.3.2 Utilisation des ressources . . . . .	11
2.3.3 Chemins critiques . . . . .	12
2.3.4 Implications réelles : Coremark & Dhrystone . . . . .	12
2.3.5 Conclusion . . . . .	13
<b>3 Description détaillée et optimisations possibles</b>	<b>14</b>
3.1 Front end - Deux étages : PCGEN et IFC - Yann & Bojana . . . . .	14
3.1.1 Objectifs . . . . .	14
3.1.2 Elements clés et fonctionnement . . . . .	15
3.2 Etage 3 : Decode - Suvetha . . . . .	17
3.2.1 Objectifs . . . . .	17
3.2.2 Elements clés et fonctionnement . . . . .	18
3.3 Etage 4 : Issue - Wakeel . . . . .	20
3.3.1 Objectifs . . . . .	20
3.3.2 Elements clés et fonctionnement . . . . .	21
3.4 Etage 5 : Exe - Nicolas . . . . .	23
3.4.1 Objectifs . . . . .	23
3.4.2 Elements clés et fonctionnement . . . . .	23
3.5 Etage 6 : Commit - Yann . . . . .	30
3.5.1 Objectifs . . . . .	30
3.5.2 Element clés et fontionnement . . . . .	30
<b>4 Conclusion et ouverture</b>	<b>31</b>
4.1 Conclusion générale . . . . .	31
4.2 Conclusions personnelles . . . . .	31

# Chapitre 1

## Projet et Architecture CVA6

### 1.1 Concours et organisation du travail

La société Thales, leader mondial dans le domaine de la défense et de l'aérospatiale organise un concours cette année en partenariat avec le GDR SoC2 (Groupement de Recherche sur les System on Chip, Systèmes embarqués et Objets Connectés) et le CNFM (Coordination Nationale de la Formation en Microélectronique et en nanotechnologies).

La première édition de ce concours étudiant d'ampleur nationale porte sur l'optimisation du cœur Ariane CVA6 basé sur l'architecture RISC-V.



L'objectif est d'adapter le code CVA6 modifié par Thales, afin de maximiser ses performances sur les FPGA. Cette architecture est en effet principalement pensée pour fonctionner sur des ASIC (Application Specific Integrated Circuit), et peut peut donc être optimisée si on souhaite en tirer le maximum sur un FPGA.

Pour cela, Thales a mis à notre disposition un dépôt GitHub dans lequel on peut trouver l'architecture à optimiser, des test bench et différentes applications compilables sur le CVA6 qui permettent de tester les performances de l'architecture. Cette dernière est testée sur une carte Zybo Z7-20 de Digilent. Notre optimisation ne doit cependant pas se faire au détriment de la taille occupée par le cœur : nous sommes limités à une augmentation de 50% maximum de surface (mesurée en nombre de LUT : LookUp Table). De préférence, il fallait également rester le plus fidèle possible au code original. Les critères de notation du jury sont les suivants :

- Augmentation de fréquence de fonctionnement (noté sur 5 points / 20)
- Gain de score CoreMark par MHz de fréquence (noté 7 points / 20)
- Augmentation limitée (voire, réduction) de la taille du cœur (noté 4 points / 20)
- "Elégance de la solution" : impact limité sur le code source original (noté 4 points / 20)

Le concours a débuté au mois d'octobre 2020, notre équipe a été constituée sur la plate-forme Piazza le 2 février 2021. Les organisateurs du concours attendaient les résultats de l'optimisation du CVA6 mi-avril. Notre équipe a été engagée hors-concours.

En raison du temps imparti entre le début du projet et la remise des résultats d'optimisation, notre effort s'est porté sur la compréhension de l'architecture CVA6, du langage SystemVerilog et de l'implémentation sur la carte ainsi que de l'analyse de Synthèse. La partie optimisation n'a pas été traitée avec regrets car elle représente le fruit des efforts accomplis pour comprendre le fonctionnement de l'architecture et

le code.

En ce qui concerne la répartition du travail, chacun d'entre nous a eu la charge d'étudier et de comprendre un des 6 étages du pipeline :

- Présentation du projet : Nicolas
- Compréhension de l'architecture du CVA6 : Nicolas, Yann et Wakeel
- Environnement de travail, implantation et tests : Nicolas
- Résultats obtenus en synthèse : Yann et Nicolas
- Etages
  - Front End (PC-GEN et IFC) : Yann et Bojana
  - Decode : Suvetha
  - Issue : Wakeel
  - EXE : Nicolas
  - Commit : Yann
- Conclusion : tout le groupe

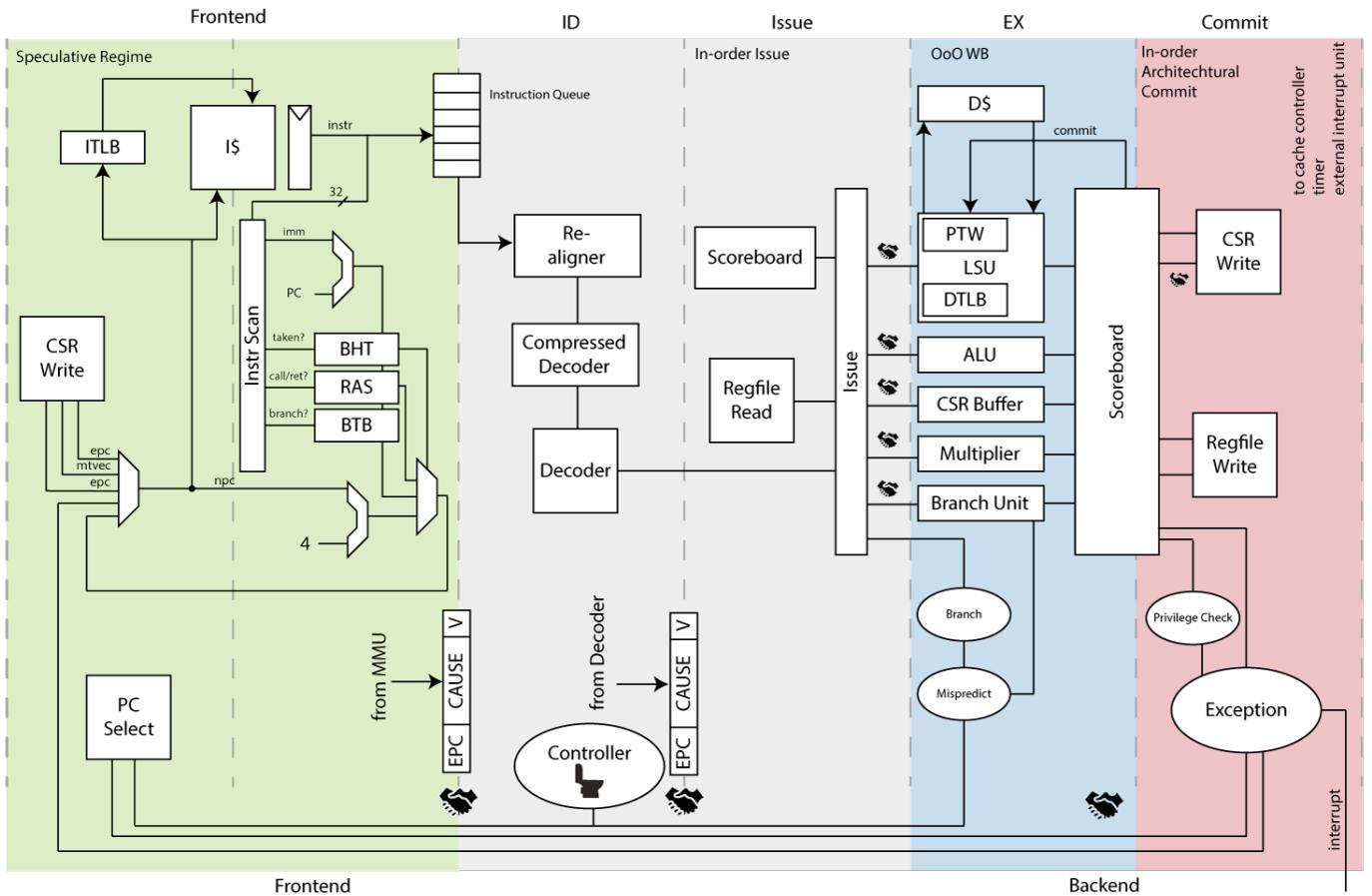
## 1.2 Compréhension de l'architecture du CVA6

### 1.2.1 Origines du CVA6

Le cœur Ariane a initialement été créé par l'ETH Zürich en se basant sur une ISA (Instruction Set Architecture) de plus en plus populaire : le RISC-V. L'équipe de Zürich a démarqué son projet des autres implémentations notamment par l'utilisation d'une MMU (Memory Management Unit) et l'implémentation de plusieurs niveaux de privilèges. Une architecture intégrant plusieurs niveaux de privilèges permet d'accueillir un système d'exploitation de type Linux. Les niveaux de privilèges permettent d'isoler le noyau et les applications.

Par la suite, un nouveau groupe de développeurs en partie constitué de membres de l'équipe de Zurich a créé l'OpenHW Group, une organisation ayant pour objectif de commercialiser des processeurs RISC-V pour des applications industrielles, en incluant entre autres le projet Ariane sous le nom de CVA6 (aux côtés de PULP SoC ou encore CV32E40P). Étant open source, le travail de l'OpenHW group a été réutilisé par plusieurs entreprises, dont Thalès qui souhaite implémenter une version plus compacte sur 32 bits.

## 1.2.2 Présentation des composants du CVA6



Source : Schéma fourni par les concepteurs du CVA6, [lien](#)

Le CVA6 est basé sur une architecture RISCV de type *load-store*, car toutes les opérations s'exécutent dans des registres internes. On ne peut pas manipuler la mémoire directement.

Toutes les architectures basées sur le RISCV diffèrent quant à leur implémentation. Les architectes font des choix architecturaux en fonction des besoins des applications qui tourneront sur l'architecture. Dans le cas du CVA6, des choix ont été faits :

- Un pipeline à 6 étages : l'étage du frontend a été divisé en deux étages, car le temps d'exécution était supérieur à celui des autres étages, en raison de la présence du matériel de prédition de branchement.
- Le scoreboard : il permet principalement de résoudre les dépendances de données (RAW, WAR et WAW) et les branchements non réussis.
- Le découplage entre le chemin de données de l'étage Issue et l'exécution des unités fonctionnelles, permettant la parallélisation des calculs.

Le CVA6 implémente le jeu d'instructions du RISCV, en particulier les extensions I, C et M. Le manuel de référence du RISCV indique les informations suivantes<sup>1</sup> :

**M : Multiplication et Division d'entiers** Cette extension ajoute au RISCV le support pour les multiplications et divisions signées et non signées ainsi que les modulus.

1. Voir ici : [lien](#)

**C : Compression** Cette extension ajoute au RISCV le support pour les instructions compressées, permettant d'obtenir des instructions sur 16 bits contre 32 bits sur la version non compressée de l'architecture. Elle permet, à terme, d'avoir un code assembleur plus compacte. De plus, les instructions compressées permettent de diminuer le nombre de Miss sur le cache d'instructions, étant donné que plus d'instructions peuvent être mises dans un bloc de taille identique.<sup>2</sup>

**I : Entiers** Le set d'instructions de base du RISCV permettant les opérations sur entiers. Il n'a pas été jugé nécessaire d'y ajouter d'extensions gérant les virgules flottantes (F D ou Q).

---

2. Pour plus d'informations à ce sujet : Papier de recherches d'Andrew Waterman sur l'impact des instructions compressées sur l'efficacité du cache et sa consommation énergétique

# Chapitre 2

## Environnement de travail

### 2.1 Pré requis et préparation

#### 2.1.1 Préparation de l'environnement de travail et de compilation

Afin d'obtenir une cohérence entre l'environnement de travail et la compilation, il est préférable de disposer d'un système d'exploitation Linux. Dans le projet, nous avons choisi la plate-forme Linux Ubuntu 20.04 LTS et 20.10. Les éléments à acquérir et à installer sont les suivants :

- Un dépôt GitHub privé dédié au projet
- Une carte Zybo Zynq-7020 de Xilinx recommandée par l'organisation du concours.
- Les sources du CVA6 : <https://github.com/ThalesGroup/cva6-software-contest>
- la chaîne de compilation RISC-V : cross compilation des programmes entre l'ordinateur et la carte : <https://github.com/riscv/riscv-gnu-toolchain>

Les options de compilation sont les suivantes :

```
. / configure --prefix=/path/to/riscv --disable-linux --
               with-cmodel=medany --with-arch=r32ima
```

```
make newlib
```

- Vivado 2020.2 (version recommandée par l'organisation du concours) : génération du bitstream implémenté sur la carte : <https://www.xilinx.com/support/download.html>

Les fichiers de la carte Zybo sont disponibles sur le site de Digilent :

<https://github.com/Digilent/vivado-boards>

- Questa 2019.1 (version sous licence) : simulation du fonctionnement du processeur CVA6
- Minicom ou screen : visualisation des sorties UART des programmes tournant sur la carte Zybo à partir d'un terminal. Les paquets sont disponibles dans la bibliothèque de Ubuntu.
- OpenOCD : communication entre les ports de la carte Zybo et le debugger gdb du RISC-V : fichier fourni dans les sources du CVA6 nommé openocd\_digilent\_hs2.cfg .

<https://github.com/riscv/riscv-openocd>

Il faut compiler OpenOCD avec les commandes suivantes :

```
. / bootstrap
. / configure --enable-ftdi --prefix=/path/to/build --
               exec-prefix=/path/to/build
make && make install
```

OpenOCD fournit le pilote permettant à l'ordinateur de communiquer avec les ports de la carte Zybo. Il s'agit du fichier *60-openocd.rules* qui est à ajouter dans le dossier */etc/udev/rules.d*. Il faut relancer le gestionnaire de pilotes.

```
sudo cp /path/to/openocd/build/share/openocd/contrib
       /60-openocd.rules /etc/udev/rules.d
```

```
sudo udevadm control --reload
```

Il faut noter que cet environnement de travail requiert une centaine de giga octets d'espace disque disponible.

### 2.1.2 Problèmes d'installation rencontrés et solutions possibles

**Vivado** Dans certains cas, l'installation de Vivado peut rester bloquée à l'étape "generating installed devices list". Il s'agit d'un problème de librairies manquantes. Il suffit alors d'installer les deux librairies suivantes, libncurses5 et libtinfo5, puis relancer l'installation.

Les versions de Vivado 2018.3, 2019.1 et 2020.2 permettent d'obtenir le bitstream du CVA6.

**Chaine de compilation RISCV** La chaîne de compilation RISCV renvoie des erreurs de compilation avec la version 10.2.1 de GCC (il s'agit d'un problème de récursivité identifié par la communauté de gcc). Il est recommandé de compiler avec la version 9.2 ou 10.2.0 de GCC. La situation est identique pour OpenOCD.

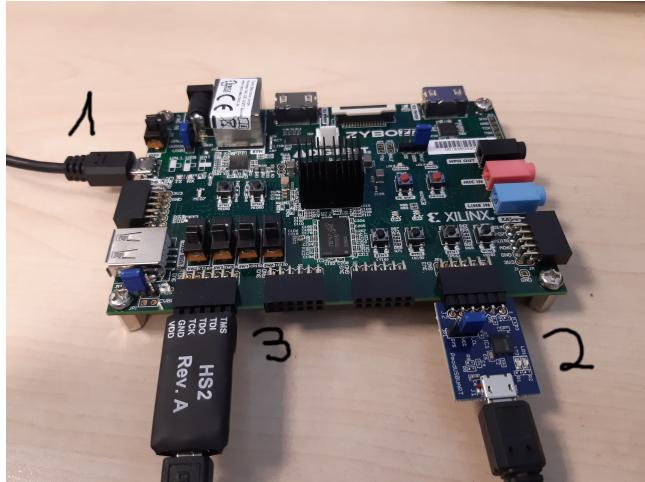
La solution la plus simple est de redéfinir la variable d'environnement.

Des bibliothèques supplémentaires sont nécessaires pour compiler sans erreur. Leur liste est disponible sur le site suivant : <https://github.com/riscv/riscv-gnu-toolchain>

**Branchements de la carte Zybo** Le projet demande 3 types de connection avec la carte :

1. câble USB pour la programmation du bitsream sur la carte et son alimentation
2. câble PMOD pour visualiser les résultats de l'exécution des programmes sur la carte Zybo
3. câble JTAG/HS2 pour éliminer les anomalies de fonctionnement avec gdb RISCV et OpenOCD.

Il est préférable de respecter l'ordre de branchement suivant : câble USB, câble PMOD puis câble JTAG/HS2.



Au moment de l'exécution du programme sur la carte et de sa visualisation dans le terminal minicom ou screen, le tty peut éventuellement renvoyer l'erreur suivante : *connection terminated*. Il suffit simplement de débrancher et de rebrancher la carte entre deux exécutions.

OpenOCD ne peut pas se lancer si l'application Vivado est ouverte, car elle occupe le port HS2.

## 2.2 Génération du bitstream CVA6, chargement sur la carte et exécution des programmes sur le CVA6

Thales propose deux options de gestion de la mémoire différentes pour générer le bitstream. La première utilise des BRAM et la seconde prend en compte la DDR.

Le tutoriel ci-dessous indique toutes les étapes pour créer un bitstream, le charger sur la carte, et exécuter les programmes sur le CVA6.

### Première étape

- source <vivado\_path>/Vivado/2019.1/settings64.sh ou <vivado\_path>/Vitis/2020.2/settings64.sh pour la version la plus récente de Vivado.
- génération du bitstream :

```
make cva6_fpga // pour la version BRAM  
make cva6_fpga_ddr // pour la version DDR
```

### Deuxième étape

- Brancher les 3 cables dans l'ordre indiqué dans le paragraphe Problèmes d'installation et solutions.
- Allumer la carte Zybo.
- Placer le bitstream sur la carte.

```
make program_cva6_fpga
```

Une fois le bitstream sur la carte Zybo, les leds PGOOD et DONE sont allumées.

**Troisième étape** Il faut maintenant compiler les programmes de test fournis par Thales et présents dans le dossier *sw/app* : *coremark* *dhrystone* *helloworld* *helloworld\_printf* *median* *multiply* *qsort* *rsort* *spmv* *towers* *vvad*. Le makefile de Thales permet également de visualiser les instructions RISC-V grâce à un déassemblage. En outre, la version hexadécimale est disponible.

Compilation croisée, génération des fichiers désassemblés et hexadécimaux :

```
make programme.riscv  
make programme.D  
make programme.hex
```

**Quatrième étape** Dans un premier terminal, OpenOCD va permettre de communiquer avec le FPGA grâce au fichier fourni connectant les ports de la carte avec l'ordinateur :

```
openocd -f fpga/openocd_digilent_hs2.cfg
```

Dans un deuxième terminal, on ouvre et on charge un programme de notre choix dans une session gdb RISC-V :

```
riscv32-unknown-elf-gdb sw/app/programme.riscv
```

On connecte gdb (deuxième terminal) avec l'interface OpenOCD (premier terminal) :

```
target remote :3333
```

Enfin, on charge le binaire de l'application sur la carte avec la commande (deuxième terminal) :

```
load
```

Dans un troisième terminal, on appelle soit minicom, soit screen pour visualiser l'exécution du programme. Dans le cas de minicom, il faut connecter le port USB de l'ordinateur avec le cable PMOD. Il faut repérer avec la commande `lsusb` le port USB disponible, et vérifier qu'il est bien reconnu.

```
Bus 005 Device 003: ID 0403:6014 Future Technology Devices  
International , Ltd FT232HSingle HS USB-UART/FIFO IC
```

On exécute ensuite la commande suivante, où la lettre X correspond au numero du tty connecté

```
minicom -D /dev/ttyUSBX
```

Il faut aussi, depuis minicom, modifier la fréquence de communication à 115200 bauds sur 8 bits.

Pour screen :

```
screen /dev/ttyUSBX 115200
```

**Cinquième étape** On démarre l'exécution du programme choisi avec la commande *c* dans le deuxième terminal. Les résultats de l'exécution sont visibles dans le troisième terminal. Les résultats obtenus sont les suivants (exemples des programmes helloworld, dhystone et coremark)



Hello World

Il est possible de modifier l'affichage avec le code suivant :

```
// uint8_t message[12] = "Hello world\n";  
uint8_t message[30] = "Here_comes_the_pain_train_!\n";  
...  
while(1){  
    UART_polled_tx_string(&g_uart_0, message);  
    while(UART_tx_complete(&g_uart_0)==0);  
}
```

```

● File Edit View Search Terminal Help iamgron@X421|AY-M413|A: ~
train !
    Here comes the pain train !
        Here comes the pain train !
            Here comes the pain train !
                Here comes the pain train !
                    Here comes the pain train !
                        Here comes the pain train !
                            Here comes the pain train !
                                Here comes the pain train !
                                    Here comes the pain train !
                                        Here comes the pain train !
                                            Here comes the pain train !
                                                Here comes the pain train !
                                                    Here comes the pain train !
                                                        Here comes the pain train !
                                                            Here comes the pain train !
                                                                Here comes the pain train !
                                                                    Here comes the pain train !
                                                                        Here comes the pain train !
                                                                            Here comes the pain train !
                                                                                Here comes the pain train !
                                                                                    Here comes the pain train !
                                                                                        Here comes the pain train !
                                                                                            Here comes the pain train !
                                                                                                Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
                                                                                                    Here comes the pain train !
................................................................
comes the pain train !
    Here comes the pain train !
        Here comes the pain train !
            Here comes the pain train !
                Here comes the pain train !
                    Here comes the pain train !
                        Here comes the pain train !
                            Here comes the pain train !
                                Here comes the pain train !
                                    Here comes the pain train !
                                        Here comes the pain train !
                                            Here comes the pain train !
                                                Here comes the pain train !
                                                    Here comes the pain train !
                                                        Here comes the pain train !
                                                            Here comes the pain train !
                                                                Here comes the pain train !
                                                                    Here comes the pain train !
................................................................
ain !
    Here comes the pain train !
        Here comes the pain train !
            Here comes the pain train !
                Here comes the pain train !
                    Here comes the pain train !
                        Here comes the pain train !
                            Here comes the pain train !
                                Here comes the pain train !
                                    Here comes the pain train !
................................................................

```

The terminal window shows a continuous loop of the string "Here comes the pain train!" being printed. The window title is "iamgron@X421|AY-M413|A: ~". The status bar at the bottom right shows system information like CPU usage, memory, and date/time.

## Hello World modifié

```

● File Edit View Search Terminal Help iamgron@X4
Microseconds for one run through Dhrystone: 490
Dhrystones per Second: 2038

```

The terminal window displays the results of a Dhrystone benchmark. The title bar shows "iamgron@X4". The output shows the time taken for one run (490 microseconds) and the resulting dhrystones per second (2038).

## Dhrystone

```

● File Edit View Search Terminal Help iamgron@X421|AY-M413|A: ~
2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks       : 1349334
Total time (secs): 0.026987
Iterations/Sec   : 111.165953
Iterations       : 3
Compiler version : GCC10.2.0
Compiler flags
:
Memory location  : BRAM
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x2e87
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 111.165953 / GCC10.2.0      / BRAM

```

The terminal window displays the results of a CoreMark performance run. It shows the core size (666), total ticks (1349334), total time (0.026987 seconds), iterations per second (111.165953), iterations (3), compiler version (GCC10.2.0), and compiler flags. The memory location is specified as BRAM. The output also includes validation information and the CoreMark version.

## Coremark

## 2.3 Résultats obtenus lors de la synthèse du CVA6

Dans cette partie nous présentons les estimations d'utilisation de ressource effectuées par l'outil Vivado pour la carte Zybo7020 que nous utilisons.

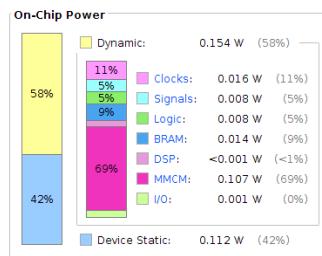
Les fichiers fournis permettent la génération de deux bitstream différents : BRAM et DDR. La différence entre ces deux bitstream réside dans le type de mémoire qui est utilisé.

BRAM signifie Block Random Acces Memory. Il s'agit d'un type de mémoire intégré au sein des FPGA. Chaque Block RAM a une taille limitée (blocs de 18KB et 36KB totalisant 630KB dans notre cas). La mémoire DDR, de taille beaucoup plus importante (dans notre cas 1GB au standard DDR3L) mais est installée en dehors du FPGA, sur la carte Zybo. On y accède par le biais d'un bus dont la bande passante est limitée (dans notre cas, 32 bits à 1066MHz). Nous comparons ici le comportement des deux versions.

### 2.3.1 Consommation énergétique

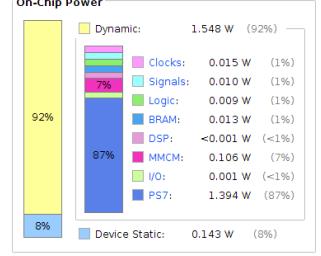
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power: 0.267 W  
Design Power Budget: Not Specified  
Power Budget Margin: N/A  
Junction Temperature: 28.1°C  
Thermal Margin: 56.9°C (4.7 W)  
Effective θJA: 11.5°C/W  
Power supplied to off-chip devices: 0 W  
Confidence level: Medium  
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 1.691 W  
Design Power Budget: Not Specified  
Power Budget Margin: N/A  
Junction Temperature: 44.5°C  
Thermal Margin: 40.5°C (3.4 W)  
Effective θJA: 11.5°C/W  
Power supplied to off-chip devices: 0 W  
Confidence level: Medium  
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

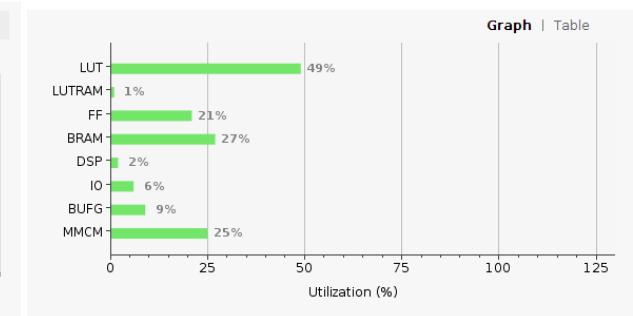
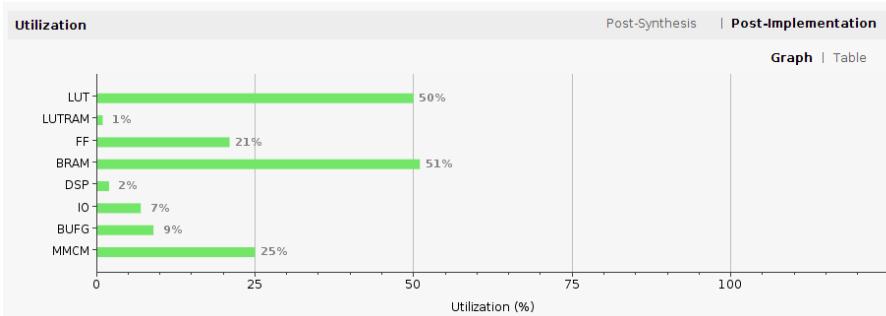


A gauche, bilan énergétique BRAM. A droite, bilan énergétique DDR.

Nous constatons ici une nette différence de consommation énergétique. Il y a en effet ici un facteur 6 en faveur de la BRAM. Cette différence provient, comme on peut le voir dans le détail, de la composante dynamique de l'analyse et plus particulièrement du composant PS7 (Processing System 7). D'après la documentation Vivado, ce composant gère l'interface avec la DDR. En effet, l'utilisation de la DDR mobilise le processeur ARM embarqué dans la carte Zybo car c'est ce dernier qui est interfacé avec la DDR. Cela entraîne donc une augmentation de la consommation considérable.

En raison de cette augmentation de consommation, la température de fonctionnement estimée est également augmentée de plus de 16 degrés celcius.

### 2.3.2 Utilisation des ressources



A gauche, le pourcentage d'utilisation de la BRAM. A droite, le pourcentage d'utilisation de la DDR.

En terme d'utilisation on observe un très nette diminution de la BRAM dans le cas de la DDR. Cette baisse est attendue étant donné que l'on met à profit la DDR. En revanche, on remarque que toutes les cellules mémoires utilisées ne sont pas basculées vers la DDR.

### 2.3.3 Chemins critiques

Timing	Timing
Worst Negative Slack (WNS): 1.428 ns	Worst Negative Slack (WNS): 1.265 ns
Total Negative Slack (TNS): 0 ns	Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 67692	Total Number of Endpoints: 66336
<a href="#">Implemented Timing Report</a>	<a href="#">Implemented Timing Report</a>

A gauche, le timing de la BRAM. A droite, le timing de la DDR.

Dans le rapport de timing de Vivado, deux éléments sont importants :

- Le WNS, ou Worst Negative Slack représente la pire distance séparant le temps mesuré du temps imposé par la contrainte. Par exemple, si le fichier de contraintes impose une fréquence de 100MHz, le pire temps autorisé sera  $1/100\text{MHz} = 10\text{ns}$ . Si le plus long temps de propagation mesuré est de 6.44ns, on aura un WNS de  $10 - 6.44\text{ns} = 3.56\text{ns}$ .
- Cette valeur doit donc idéalement être le plus éloignée possible de 0, et en aucun cas être négative.
- Le TNS, ou Total Negative Slack représente la somme de toutes les chemins ne respectant pas les contraintes. Autrement dit, la somme de tous les points auxquels le WNS est négatif. Un TNS acceptable est donc un TNS nul.

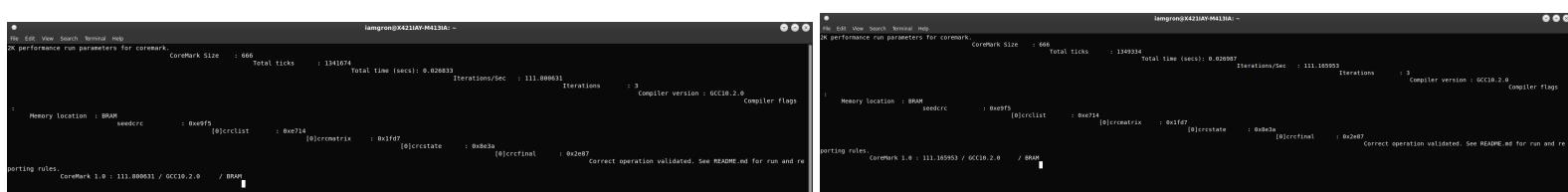
Dans les deux cas, le TNS du CVA6 est nul, ce qui est une bonne chose. On remarque cependant que le WNS en mode DDR est diminué de 0.163ns.

Cela signifie que le temps estimé est plus proche du temps imposé par le fichier de contraintes. Les valeurs sont donc moins bonnes pour la version DDR du CVA6.

### 2.3.4 Implications réelles : Coremark & Dhystone

Nous avons mesuré les performances des deux modes mémoire en exécutant deux des benchmarks fournis par Thales.

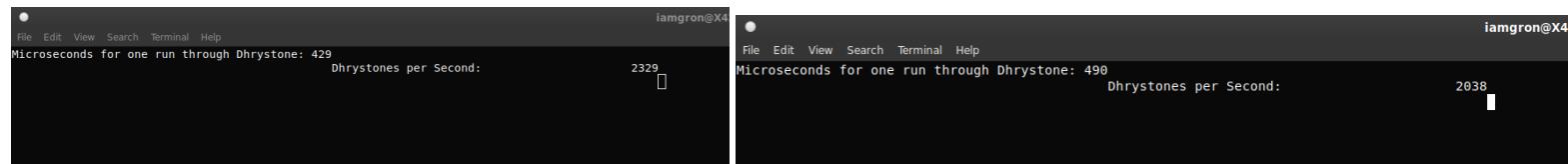
#### Coremark



A gauche, CoreMark de la BRAM. A droite, le CoreMark de la DDR.

Le score Coremark obtenu en mode DDR est légèrement inférieur à celui obtenu en mode DDR. Cela peut s'expliquer par la performance moindre de la DDR.

## Dhystone



A gauche, Dhystone de la BRAM. A droite, le Dhystone de la DDR.

La tendance observée avec Coremark se confirme, avec un résultat DDR plus nettement inférieur à celui de la BRAM. La encore, nous expliquons cette différence par le fait que la DDR est plus lente de par son interface limitée par le bus de communication et sa distance avec le FPGA.

### 2.3.5 Conclusion

L'utilisation de la DDR au lieu de la BRAM ne semble ici pas justifiée. On remarque en effet une dégradation générale des performances, que ce soit sur le plan des estimations de Vivado ou sur le plan pratique avec les benchmarks. Une application utilisant une plus grande quantité de mémoire pourrait justifier le passage en DDR, mais pour notre utilisation le choix se porte sur la BRAM.

# Chapitre 3

## Description détaillée et optimisations possibles

### 3.1 Front end - Deux étages : PCGEN et IFC - Yann & Bojana

#### 3.1.1 Objectifs

**Etage 1 : PC-GEN** L'Étage PC Gen est responsable de la génération du prochain compteur du programme, tout les compteur du programme sont adressés logiquement en cas d'erreur d'adressage l'instruction "fence.vm" vide le pipeline, L'étage a aussi pour tâche de gérer les prédictions sur l'adresse de branchement (Branch prediction). Il comprend un tampon de cible de branchement (BTB) utilisé pour mettre à jour les informations sur les erreurs de branchement et (BHT) implémenté comme un compteur de saturation sur 2 bits, et servant à la prédition. PC Gen communique avec l'étage 2 (IFC) grâce à des signaux de validité : IFC envoie un signal "ready" quand il est prêt à recevoir un signal et pcgen demande une instruction valide en utilisant le signal "fetch\_valid".

Les PC suivant peuvent venir de différentes sources :

- Une affectation par défaut : Récupération du PC+4 (alignement sur 32 bits)
- Prédiction de branchement : Dans le cas où BHT et BTB prédisent un branchement, et informent IF que la prédition est correcte, pour permettre d'avoir une correction sur la prédition cette info est stocké dans branchpredict\_sbe\_t
- Erreur de prédition venant de l'étage EXE : Résultante d'une erreur de prédition de branchement. Demande la récupération de la bonne adresse.
- Un retour d'appel d'environnement : Correction du Pc via recherche dans le registre "[mls]epc" de adresse suivante.
- Une Exception / ou d'une Interruption : Reprise à l'adresse ou a eu lieu l'interruption, l'unité CSR doit déterminer ou a eu lieu l'erreur, et présenter l'adresse correspondante a Pcgén.
- Vidange du pipeline en raison des effets secondaires de CSR : Le pipeline est vidé et reprise à l'adresse de l'instruction suivante.
- Du Débogage : Ordre de priorité le plus élevé, pouvant interrompre toutes les demandes de flux de contrôles.

**Etage 2 : IFC** L'Étage IF reçoit les informations directement de l'étage PC Gen, les adresses suivantes qu'elle soit prédit ou non, et le PC actuel, et si la requête est valide ou non. L'étape IF demande à la MMU d'effectuer une traduction d'adresse sur le PC demandé et contrôle l'interface de la mémoire \$I. L'étage IF ne peut traiter que 2 transactions à la fois dans le cas où on lui fournit plus de 2 l'étage IF ne récupère aucune information supplémentaire de pcgen temps qu'une instruction valide ne sera pas envoyé.

### 3.1.2 Elements clés et fonctionnement

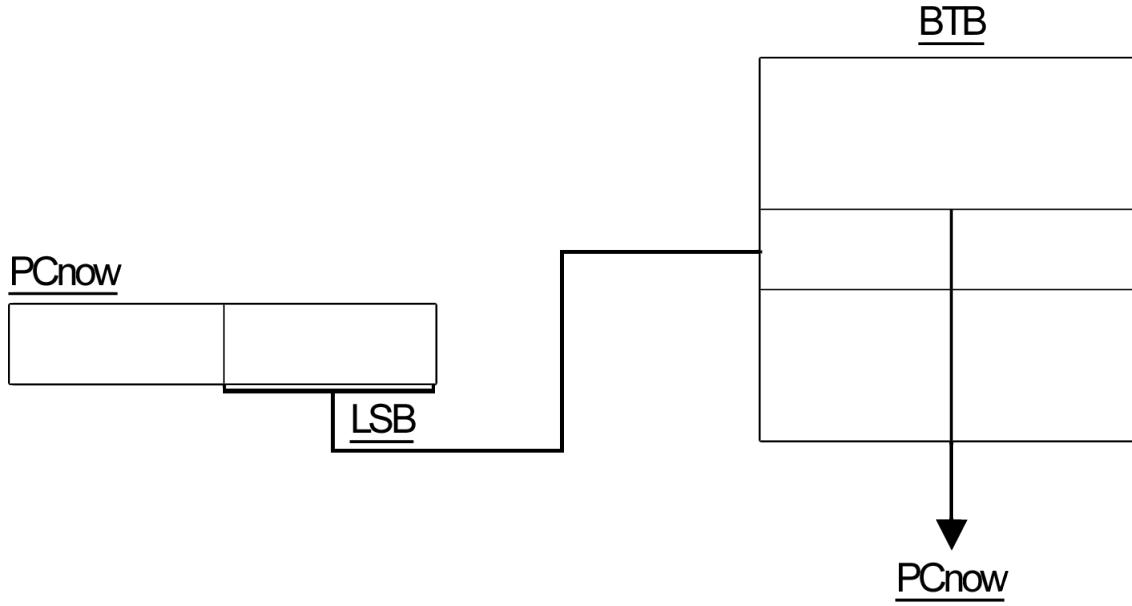


Schéma du BTB

**BTB** Le prédicteur le plus simple qui utilise de l'historique s'appelle Branch Target Buffer-BTB. Le PC courant qui vient de l'instruction de branchemen est utilisé pour indexer une case dans un tableau qui est le BTB. Dans chaque case de ce tableau on a stocké une adresse qui représente la meilleure prédition pour le prochain PC.

A chaque fois qu'on fait une prédition, l'adresse du PC et du prochain PC prédit par le BTB sont transmises à travers le pipeline jusqu'au moment où le PC est vraiment calculé. À ce moment, on compare le PC qui était prédit avec le PC calculé et utilisé. Dans le cas où ils ne sont pas égaux, on aura une erreur de prédition.

Pour que le système fonctionne, il faut ensuite faire la mise à jour dans le BTB. On utilise le PC de l'instruction de branchemen qui était transmis dans le pipeline pour indexer la bonne case dans le BTB et écrire la nouvelle valeur de PC qui était calculée.

Donc, la prochaine fois quand on rencontre cette instruction de branchemen, l'adresse prédite sera la bonne adresse et on n'aura pas une erreur de prédition. Le but de BTB est de réussir à prédire la bonne adresse tout en gardant une latence d'un cycle par instruction. Donc, le BTB doit rester assez petit en taille. Or, on aimerait bien avoir une case pour chaque PC possible. Malheureusement, ça va énormément augmenter la taille de BTB et donc il a été décidé de garder un nombre limité de places dans le BTB. Donc, pour être réaliste, on va sauvegarder dans le BTB que les adresses des instructions qui vont s'exécuter bientôt dans le temps. Par exemple, si on a une boucle, on va remplir le BTB avec toutes les instructions qui se trouvent dans la boucle et on trouvera ce dont on a besoin à chaque itération.

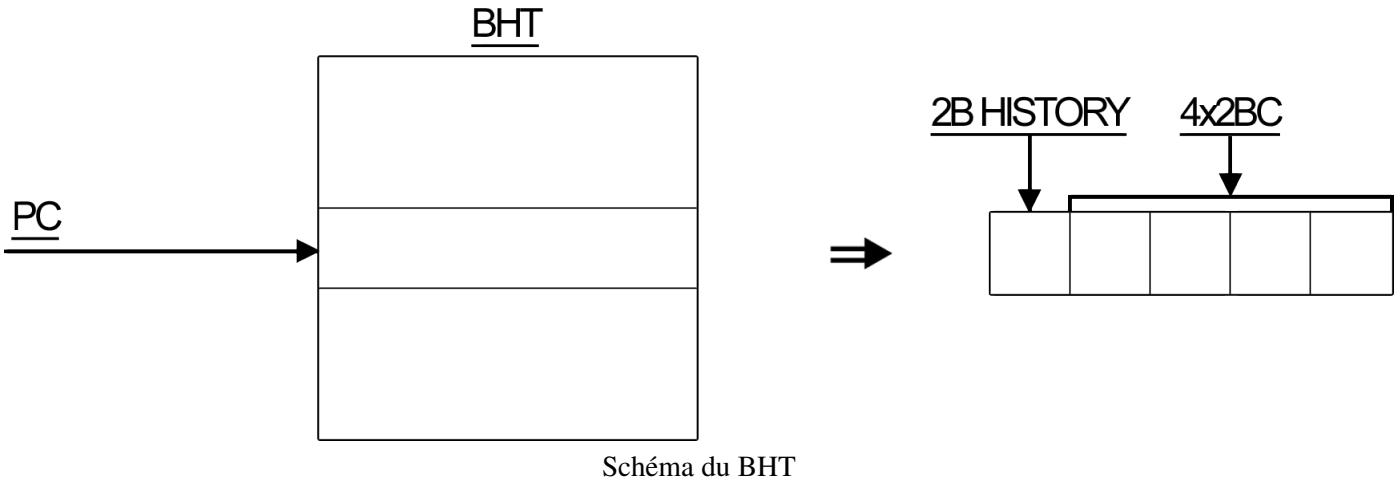


Schéma du BHT

**Interaction BTB-BHT** Le composant BHT va donner la décision sur le branchement, il répond à la question est-ce que le branchement est pris ou non.

Si le branchement est pris, le composant BTB va fournir l'adresse cible de branchement qui était prédictée en utilisant l'historique et par la mise à jour régulière de BTB.

Si le branchement n'est pas pris alors on va juste augmenter PC pour passer à l'adresse de la prochaine instruction qui suit dans le programme.

Une fois que l'adresse est vraiment calculée, on fait la comparaison avec l'adresse prédictée. Comme il était mentionné auparavant, si le branchement prédicté était pris, on fait la mise à jour dans le BTB mais aussi dans le BHT en incrémentant le compteur de saturation.

Si on avait pas un branchement ou bien il n'était pas pris, on décrémente le compteur mais on ne fait pas la mise à jour dans le BTB car on a pas une nouvelle adresse comment le branchement n'était pas pris. Comme le BTB est déjà très petit de taille, on veut le réservé pour les instructions dont on est sûres qu'il s'agit d'un branchement. Par contre le BHT, comme c'est un composant sur 2 BITS, on peut se permettre d'avoir des cases dédiées à beaucoup plus d'instructions et faire la mise à jour tous les temps.

**RAS - Return Address Stack** Il existe plusieurs types de branchement qu'on peut prédire dans un programme. Tout d'abord, on a les branchements conditionnels. Pour les branchements conditionnels, on a besoin de prédire la décision sur le branchement, donc prise ou non prise et on utilise le module BHT dans ce contexte. Enfin, si le branchement est pris, on a aussi besoin de prédire l'adresse et on a le module BTB mis à notre disposition pour cette prédiction.

Ensuite, il existe une autre type d'instruction de branchement très important, il s'agit des branchements de type Jump, Function Call etc. En ce qui concerne la prise de décision de branchement, il s'agit des branchements qui sont toujours prises donc cette partie de la prédiction pose pas de problème. Après, en ce qui concerne l'adresse, la plupart d'entre eux font un saut vers un label ou bien un appel vers une fonction spécifique donc leur destination est toujours très bien définie. On peut en conclure que le BTB et le BHT vont suffrir pour ce genre d'instruction. Par contre, il existe un type d'instruction de branchement où c'est vraiment difficile de prédire l'adresse avec le BTB et il s'agit de Fonction Return. Comme il s'agit d'une décision qui est toujours prise, on aura pas de problème avec le fonctionnement de BHT. Dans le cas où une fonction est toujours appelée du même endroit, le BTB va aussi réussir à faire une bonne prédiction. Mais, il faut prendre en compte la nature des fonctions. Dans la plupart des cas, une fonction est faite pour être utilisée dans plusieurs endroits dans un programme. Donc, l'adresse de retour sera différente pour chaque cas et cela va poser un problème pour la prédiction faite de la part de BTB. Donc, ceci est la raison pour laquelle il faut ajouter un module RAS à cet étage, pour avoir un traitement de prédiction complet dans les différentes circonstances.

Le module RAS est un prédicteur dédié à prédire la return fonction. Il s'agit d'une LIFO (Last In First Out), de petite taille, contrôlée par un pointeur. Le principe de fonctionnement est très simple. Une fois

qu'une fonction est appelée, on transfère l'adresse de l'instruction suivante vers la LIFO en utilisant un signal PUSH et on déplace le pointeur pour qu'il pointe vers une autre case. Ensuite, quand on rencontre finalement le return dans le programme, on envoie un signal POP qui va signaler à la LIFO qu'il faut fournir l'adresse qui a été sauvegardée auparavant. Ensuite on positionne le pointeur sur cette case et même sans supprimer la valeur cette case est déjà libre pour stocker une nouvelle adresse. On peut remarquer que on considère bien le RAS comme un prédicteur et se demander pourquoi ils ont décidé de ne pas utiliser la LIFO de la programme. Alors, la réflexion est toujours faite en termes de nombre de cycles par instruction. La LIFO de la programme peut être assez grande et peut contenir des fonctions enchainées. Dans le cas des branchements, on a besoin d'une LIFO qui sera proche de reste du matériel dédié à la prédiction et aussi une LIFO de petite taille pour garder la latence d'un cycle par instruction. Par contre, avec un nombre limité de cas, le RAS peut se remplir très vite. Dans le cas où il n'y a plus de place, on a deux options, soit interdire de continuer avec les PUSH soit continuer et écraser les valeurs d'avant.

**Instruction Scan** Le problème avec le RAS est dans le fait que on fait la prédiction même avant avoir décodé l'instruction, donc on ne sait pas forcément si c'est une instruction de type return. On peut utiliser encore un prédicteur à un seul bit. Cela suffira car on sait que si une adresse était adresse de retour alors c'est sur que ça sera le cas la prochaine fois aussi. Mais, dans l'architecture RISC-V ils ont utilisé une autre approche dont le nom est Predecoding.

Le processeur contient une mémoire Cache qui sauvegarde les dernières instructions qui étaient déjà cherchés dans la mémoire. Donc seulement si l'instruction n'est pas dans la mémoire cache, le processeur va aller chercher dans la mémoire. Par ailleurs, à chaque fois qu'on récupère une instruction de la mémoire, on va décoder l'instruction et voir si c'est du type return ou pas et ensuite sauvegarder cette information dans le Cache. En général, on peut utiliser le predecoding pour dire s'il s'agit d'un branchement ou pas. Si on sait que ce n'est pas un branchement on peut très facilement décider de ne pas du tout utiliser la prédiction de branchement. Si on a des instructions de taille différente on peut aussi utiliser le predecoding pour savoir c'est quoi la taille de l'instruction. Le principe de predecoding est dans ce cas implémenté avec le composant InstructionScan. On remarque sur le schéma de pipeline que ce composant va permettre de savoir s'il s'agit d'un branchement, d'une fonction ou d'un return.

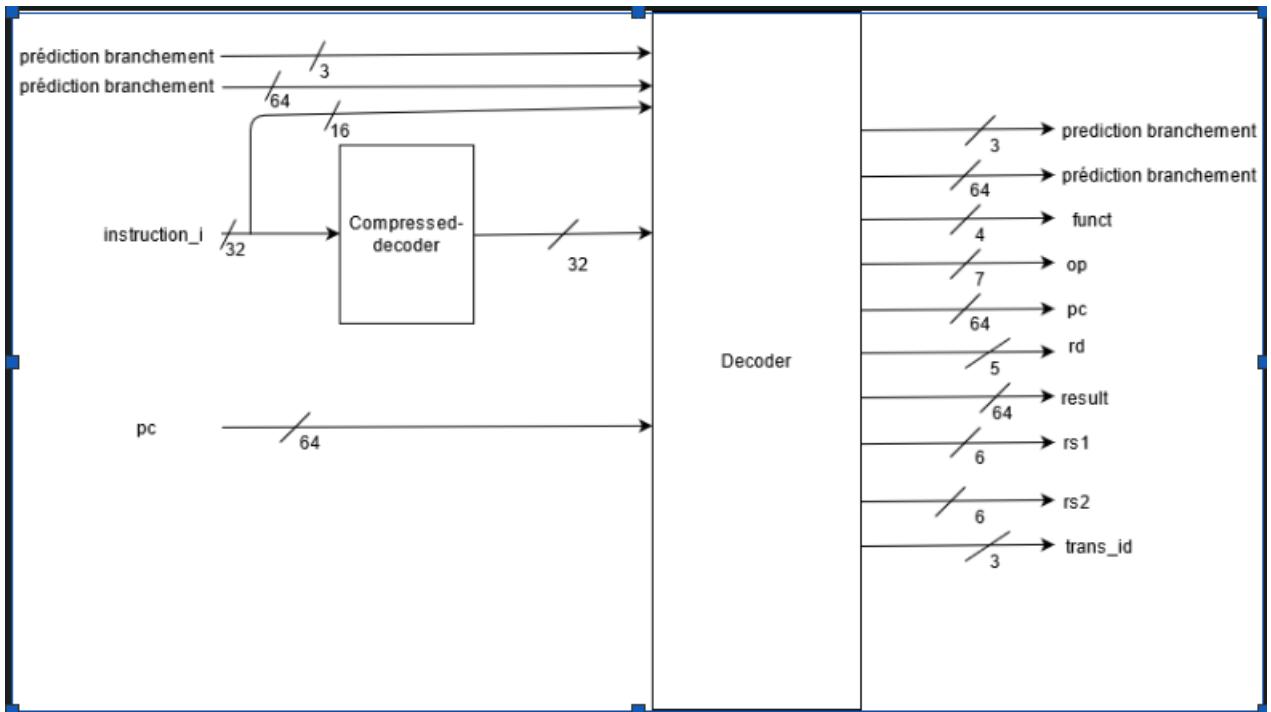
## 3.2 Etage 3 : Decode - Suvetha

### 3.2.1 Objectifs

L'étage Decode a pour objectif de récupérer l'instruction à la sortie du Front End puis la décoder et l'envoyer à l'étage suivant qui est l'étage Issue . Il recherche dans le flux de données entrant des instructions potentielles, les réalignent et les décomposent s'il s'agit d'instructions compressées. Il va transmettre ces informations à l'étage suivant. L'étage ID est principalement constitué de trois composants : le réaligner, compressed decoder et le decoder :

- Le réaligner vérifie le flux de données entrant.
- Le compressed decoder permet de vérifier si toutes les instructions sont sur 32 bits et de faire la conversion si cela est nécessaire.
- Le decoder a pour objectif de décoder l'instruction entrante en identifiant les différentes parties de l'instruction et en les transmettant à l'étage suivant.

Voici le schéma d'ensemble de l'étage ID :



On remarque qu'il y a quatre entrées principales : 2 entrées sur la prédiction de branchement dont une sur 3 bits et l'autre sur 64 bits, puis l'instruction sur 32 bits qui devra être décodée par la suite et enfin une entrée pc sur 64 bits. Comme on peut le voir sur ce schéma, en sortie du decoder on obtient deux instructions de prédiction de branchement que nous avions en entrée ainsi que les différentes parties de l'instruction entrante tel que l'opcode sur 7 bits par exemple.

### 3.2.2 Elements clés et fonctionnement

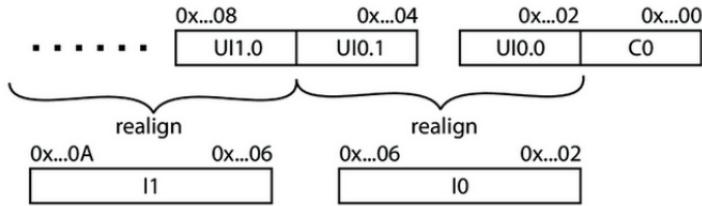
**Re-aligner** On distingue deux types d'instructions :

- Une instruction compressée est définie sur 16 bits. Elle est identifiable à l'aide des deux bits de poids faibles qui sont différents de 11 s'il s'agit d'une instruction compressée.
- Une instruction normale sur 32 bits. Elle peut être identifiée à l'aide des deux derniers bits qui sont toujours à 11.

Le réaligner a un rôle important dans cet étage, car à la compilation une instruction compressée peut rendre une instruction normale non alignée. Il va aligner les instructions sur une adresse multiple de 32 bits. Dans notre cas, il s'agira d'une adresse de 32 bits. Dans un premier temps, le réaligner va lire l'instruction entrante, il va lire bit par bit jusqu'à qu'il aie 32 bits. Une fois qu'il aura chargé cette instruction dans une FIFO, il va faire un mandat d'accès mémoire.

En revanche, si c'est une instruction compressée, il va lire les 16 bits de cette instruction et continuer sa lecture avec les bits de l'instruction suivante. Dans ce cas là, on aura lu une instruction compressée sur 16 bits et 16 bits d'une instruction normale. Sur le schéma ci-dessous, on peut remarquer que l'instruction C0 qui se trouve à droite est une instruction compressée et que la FIFO va compléter les bits restants en récupérant les 16 bits de poids faible de l'instruction suivante UI0. On voit bien que l'instruction UI0 a été séparée en deux.

### Unaligned Instructions:



*Source : Schéma fourni par les concepteurs du CVA6, [lien](#)*

Cela va entraîner le mandat de deux accès mémoire avant même que l'instruction puisse être entièrement décodée. Pour éviter cette situation, le réaligner doit d'abord s'assurer qu'il y a assez de place pour charger l'instruction complète. C'est pour cette raison que le réaligner doit utiliser une mémoire afin de garder la trace de l'instruction précédente pour savoir si elle était compressée ou non et afin de décider que faire de l'instruction à venir.

**Compressed-decoder** Comme nous l'avons vu précédemment, une instruction compressée est sur 16 bits. Le compressed-decoder va nous permettre de décompresser toutes les instructions compressées. Il va récupérer une instruction de 16bits et l'étendre à son équivalent sur 32 bits. Toute instruction compressée a son équivalent en 32 bits. Il a une entrée sur 32 bits et 3 sorties dont l'une est sur 32 bits qui représente l'instruction décompressée. Afin de retrouver l'instruction correspondant sur 32 bits, on va initialiser les sorties de ce composant dans un premier temps. Selon l'opcode identifié et la fonction, on va retrouver l'instruction correspondante sur 32 bits que nous allons ensuite transmettre au decoder.

**Decoder** Enfin le décodeur va récupérer les données d'instructions brutes ou les instructions décompressées et les décoder. Il va transmettre les bits correspondant aux différents opérandes selon les types d'instructions.

Il va prendre cinq entrées : deux entrées de prédiction de branchement (une sur 3 bits et l'autre sur 64 bits), l'instruction de 32 bits sortant du compressed-decoder, une instruction compressée de 16 bits et une entrée de 64 bits correspondant au pc. Et en sortie il générera les bits correspondant aux différentes parties de l'instruction tel que l'opcode, les opérandes etc.

Une instruction peut être de différents types : R, I, S, SB, U, UJ. Selon ces types, les formats de l'instruction sont différents, pour certaines instructions on va trouver une partie immédiat qui correspond à une constante.

**R** Les instructions de type R utilisent trois opérandes (deux registres pour les sources et un registre destinataires qui va récupérer le résultat de l'opération). Ces opérations peuvent être arithmétiques ou des décalages.

**I** Les instructions de type I utilisent deux opérandes (un registre pour la source et l'autre pour stocker le résultat obtenu) et un immédiat de 16 bits. A l'aide de ces instructions, on peut réaliser ces opérations avec une constante tel que des opérations arithmétiques, les load, les branchements etc.

**SB** Les instructions de type SB sont réservées pour les instructions de branchement. Ils seront constitués de deux registres sources et de quatre immédiats (constantes).

**S** Les instructions de type S correspondent aux instructions stores tel que sw et sb.

**U** Les instructions de type U sont des instructions utilisant des immédiats de 20 bits tel que lui.

**UJ** Et enfin les instructions UJ correspondent à des instructions de sauts tel que jal.

	31	30	25 24	21	20	19	15 14	12 11	8	7	6	0
<b>R</b>		funct7		rs2		rs1	funct3		rd		opcode	
<b>I</b>			imm[11:0]			rs1	funct3		rd		opcode	
<b>S</b>		imm[11:5]		rs2		rs1	funct3		imm[4:0]		opcode	
<b>SB</b>	imm[12]	imm[10:5]		rs2		rs1	funct3	imm[4:1]	imm[11]		opcode	
<b>U</b>			imm[31:12]					rd		opcode		
<b>UJ</b>	imm[20]	imm[10:1]	imm[11]		imm[19:12]			rd		opcode		

Source : carte de référence du RISCV, [lien](#)

En regardant ce tableau, on peut comprendre qu'à partir d'une instruction de 32 bits, nous devons tout d'abord identifier le format de l'instruction. Ce format peut être repéré à partir de l'opcode c'est-à-dire des 7 bits de poids faible. En revanche, les mêmes types d'instruction ont le même opcode. Par exemple, toutes les instructions load sont de type I et elles ont toutes le même opcode. Pour pouvoir identifier l'instruction exacte, nous devons regarder l'opcode et le funct3 qui est sur 3 bits et qui va préciser la fonction réaliser. La première étape du decoder serait d'identifier la fonction réalisée par l'instruction. Nous allons donc repérer l'opcode , la funct3 et effectué un and entre les deux.

Ensuite, les différentes parties de l'instruction vont passer par une mémoire RTL qui est l'acronyme de Register Transfer Level. Ces modèles sont utilisés pour faire des tests, évaluer les performances ou encore générer des modèles de bas niveau tel que des Netlist.

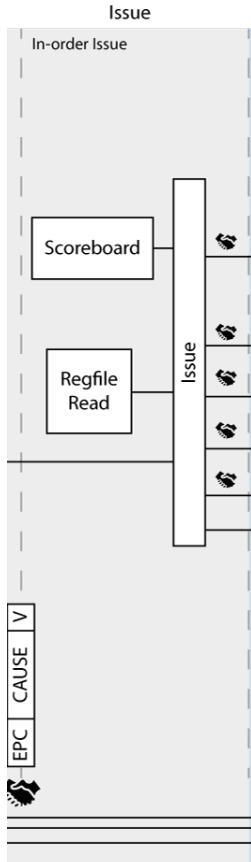
La sortie de ces mémoires RTL vont utiliser des multiplexeurs afin de transmettre les différentes parties de l'instruction en sortie du decoder.

### 3.3 Etage 4 : Issue - Wakeel

#### 3.3.1 Objectifs

L'étage Issue est crucial pour le fonctionnement de l'étage EXE, car il répartit les instructions décodées reçues de l'étage DECOD vers les unités fonctionnelles d'une façon découpée. Autrement dit, le chemin de données de l'étage Issue est découplé du fonctionnement des unités fonctionnelles présentes dans l'étage EXE.

### 3.3.2 Éléments clés et fonctionnement



*Source : extrait du schéma fourni par les concepteurs du CVA6, [lien](#)*

**Issue** Le schéma ci-dessus montre que l'étage Issue est composé de plusieurs éléments, un fichier de registres, un scoreboard et une FIFO.

L'exécution s'effectue en quatre étapes :

1. émission des instructions : attribution de l'identifiant unique à chaque instruction.
2. lecture des opérandes
3. exécution
4. write-back : acquittement de fin d'exécution des unités fonctionnelles

L'étage Issue gère toutes ces étapes sauf l'exécution qui est prise en charge par l'étage EXE.

L'étage Issue supervise toutes les instructions émises. Il connaît aussi l'état de chaque unité fonctionnelle ainsi que le registre dans lequel les instructions sont écrites. Toutes ces informations sont rassemblées dans la partie du scoreboard présente dans l'étage Issue.

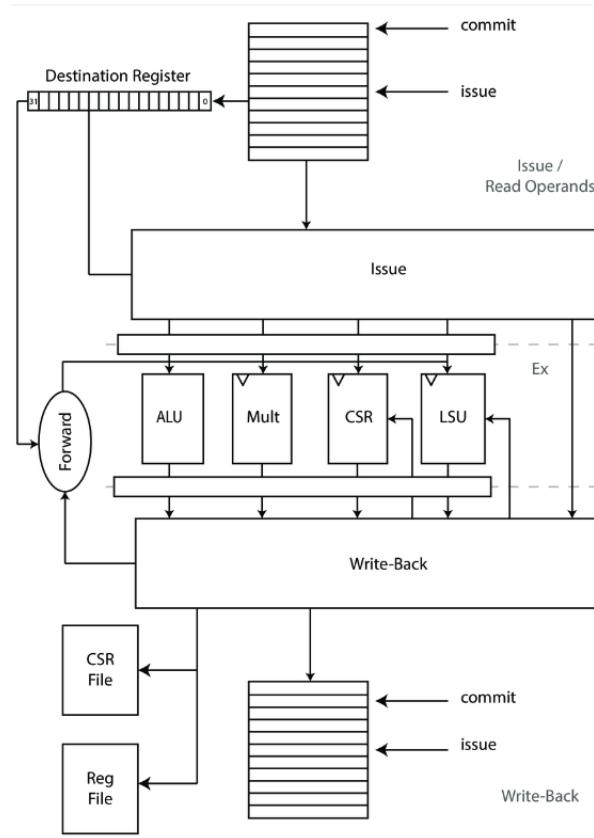
**Fichier de registre** L'étage Issue contient un fichier de registres qui sert à stocker le résultat des calculs des unités fonctionnelles ainsi que les opérandes en attente d'exécution en raison des dépendances de données.

**FIFO** A la réception d'une instruction décodée, la FIFO vérifie si l'unité fonctionnelle qui doit exécuter l'instruction est libre ou si elle le sera dans le prochain cycle. Puis, elle vérifie si les opérandes sources (rs1 et rs2) sont disponibles et qu'aucune autre instruction n'utilise ces mêmes registres (dépendances). La lecture des opérandes se fait dans le même cycle que l'émission de l'instruction.

La lecture des opérandes se fait prioritairement dans le scoreboard puis dans le fichier de registres car, la valeur est mise à jour en premier dans le scoreboard.

**Scoreboard de l'étage Issue** Le scoreboard fonctionne comme une FIFO. Il connaît la répartition des instructions, les redirige correctement afin d'éviter les conflits. Il existe sur la FIFO deux ports, un de lecture et un d'écriture ainsi que des signaux de validité et d'acquittement.

Les instructions décodées de l'étage DECOD sont directement écrites dans le scoreboard. Ces instructions sont ensuite lues par la FIFO Issue. Le scoreboard s'assure qu'une seule instruction soit écrite dans un registre de destination libre. Il comporte un circuit combinatoire qui affiche l'état des 32 registres de destination ainsi que les sorties des unités fonctionnelles qui sont consultables dans le signal rd\_clobber. L'émission des instructions se fait dans l'ordre du programme. Le scoreboard de l'étage Issue crée un identifiant unique pour chaque instruction. Le scoreboard de l'étage EXE va envoyer une confirmation d'exécution ou un message d'erreur à la FIFO de l'étage Issue.



Source : extrait du schéma fourni par les concepteurs du CVA6, étage Issue, [lien](#)

On voit dans ce schéma les différentes composantes de l'étage Issue et EXE avec le sens de transmission des instructions.

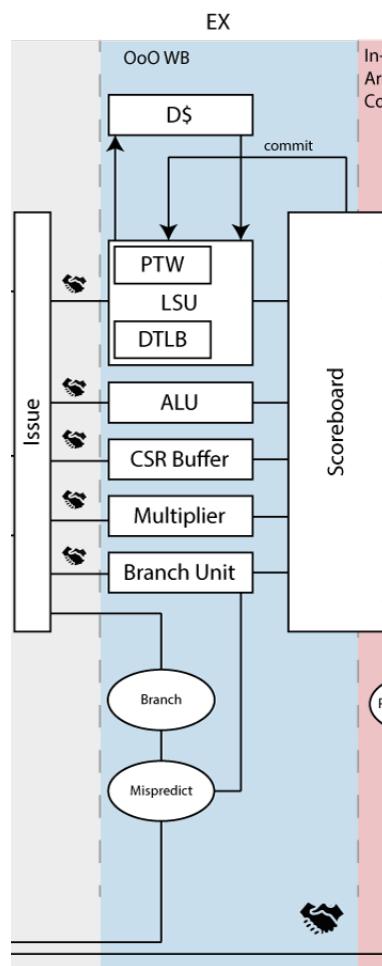
## 3.4 Etage 5 : Exe - Nicolas

### 3.4.1 Objectifs

Cet étage est chargé de réaliser la partie calculatoire débouchant sur le résultat de toutes les instructions transitant par le pipeline. C'est aussi dans cet étage que se trouve l'interface mémoire, ainsi que le matériel de validation des prédictions de branchement.

L'étage EXE matérialise l'originalité des choix architecturaux des concepteurs du CVA6. En effet, c'est à cet étage que se situe le scoreboard qui permet de paralléliser l'exécution des instructions dans les unités fonctionnelles. Le nombre de cycles de gels en est considérablement réduit.

### 3.4.2 Elements clés et fonctionnement



Source : extrait du schéma fourni par les concepteurs du CVA6, [lien](#)

Le schéma ci-dessus montre que l'étage EXE regroupe 5 unités fonctionnelles (ALU, LSU, CSR Buffer, Branch Unit, Multiplicateur), un scoreboard et un gestionnaire de branchements. Nous allons voir leur intérêt et leur fonctionnement successifs.

**Scoreboard** Le scoreboard est un élément-clé de l'étage EXE car il gère les 3 dépendances de données RAW, WAR ET WAW ainsi que la sortie des résultats des instructions des unités fonctionnelles qui arrivent dans le désordre. Ceci est rendu possible par l'attribution d'un identifiant unique à chaque

instruction dans le scoreboard de l'étage Issue.

Pour les dépendances Write After Write, les registres sont renommés avant d'être envoyés dans les unités fonctionnelles. Le nombre de renommages de registres est limité à deux.

Lorsqu'il n'y a plus de dépendances de données et que les calculs ont été validés, le scoreboard de l'étage EXE envoie les instructions par deux à l'étage Commit.

Le travail accompli par le scoreboard permet de réduire le nombre de cycles de gels dans le pipeline en résolvant les dépendances de données, et les instructions qui demandent plusieurs cycles pour s'exécuter comme les *load*, les *store* les branchements et les multiplications et divisions.

Le scoreboard décide du moment opportun pour :

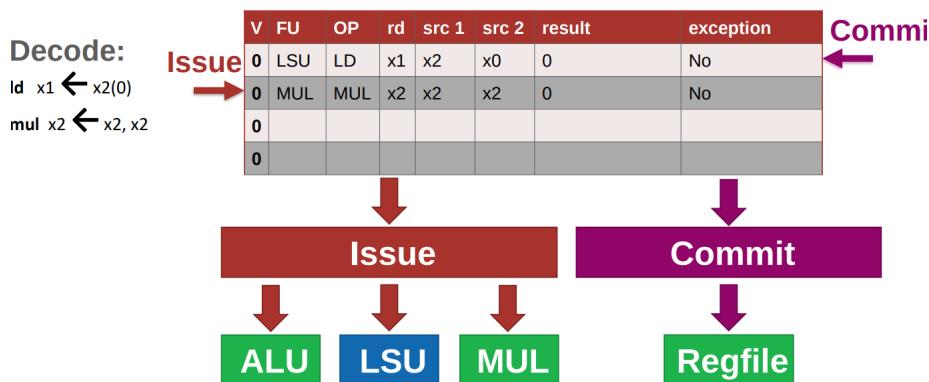
- lire les opérandes d'une instruction multicycle,
- reprendre l'exécution
- écrire les résultats dans les registres cibles

Grâce à l'identifiant unique, le scoreboard envoie les instructions à l'étage commit dans l'ordre du programme exécuté sur le CVA6.

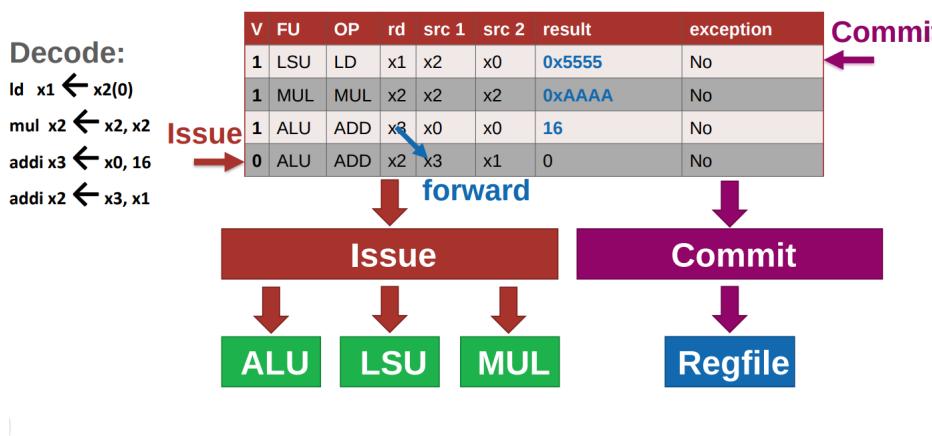
Les images qui suivent explicitent le fonctionnement du scoreboard. Les instructions y sont enregistrées, et le type d'opération, les opérandes ainsi que la FU concernée sont écrits. Quand la FU retourne le résultat du calcul, le champ résultat est rempli, et le bit de validité est mis à jour. Eventuellement, les exceptions sont également renseignées. Si aucune dépendance n'est détectée sur une instruction et que son bit de validité est à 1, l'instruction est envoyée dans l'étage Commit.

*Note : il manque dans ces images le champ correspondant à l'identifiant unique attribué à chaque instruction.*

## Scoreboard – 2 cycle instruction (MUL)

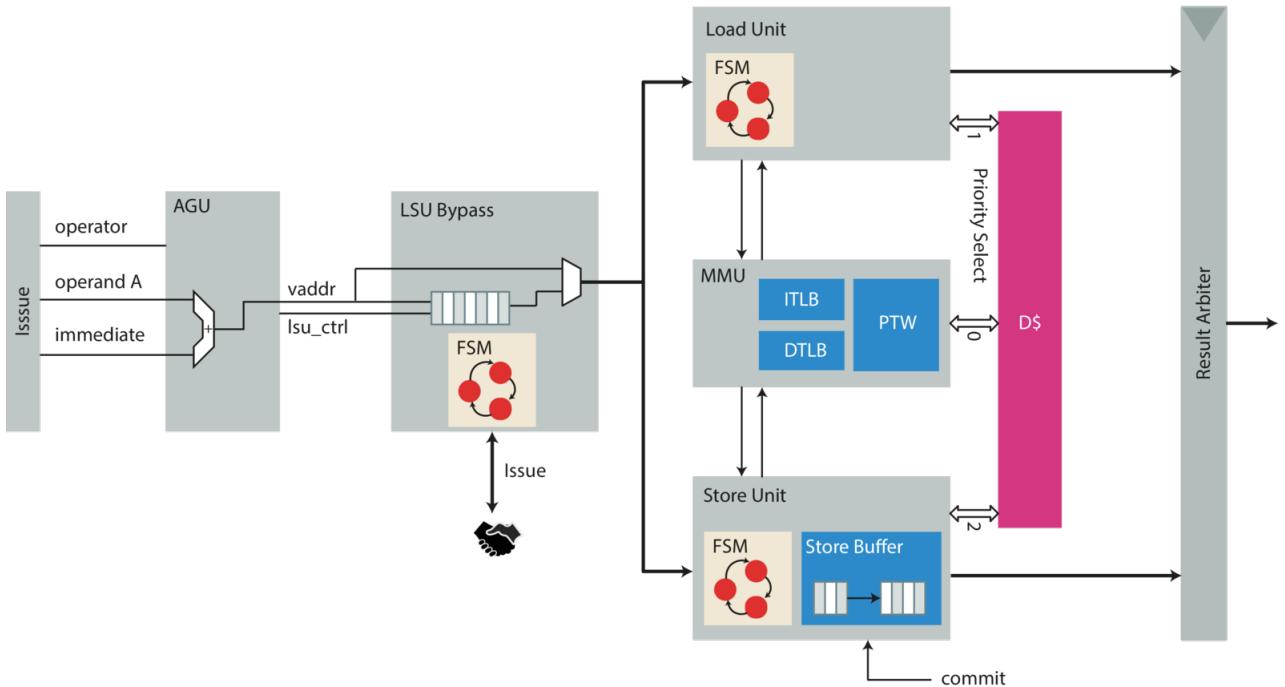


## Scoreboard – Multiple Write Back



Source : Présentation de l'ETH Zürich - Working With RISCV, [lien](#)

## Les 5 unités fonctionnelles (FU)



Source : Schéma de la LSU fourni par les concepteurs du CVA6, [lien](#)

**1 - LSU** La LSU est composée de deux blocs indépendants qui partagent un bloc de traduction d'adresses commun. Le premier correspond à la Load Unit, et l'autre à la Store Unit. Les deux blocs utilisent des signaux différents pour indiquer s'ils sont prêts ou non. Si les blocs ne sont pas prêts, la LSU conserve le dernier signal reçu. Elle peut retarder l'exécution d'une écriture grâce à la FIFO à deux emplacements présente dans le LSU Bypass. Cela est rendu nécessaire par l'arrivée tardive de la réussite soit de la lecture soit de l'écriture.

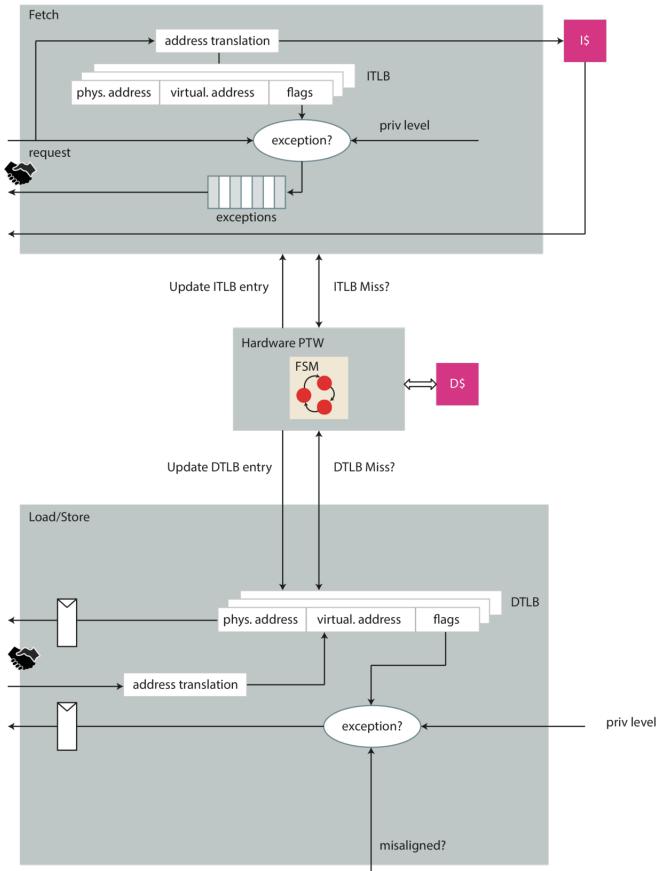
La LSU, ou Load Store Unit sert d'interface avec la mémoire du processeur. Elle implémente pour cela plusieurs blocs distincts :

- L'AGU : Adress Generation Unit. Premier composant de la LSU, ce composant calcule les adresses virtuelles des données.
- La Load Unit (LU) charge les données et les instructions dans leur caches respectifs. Les lectures ont la priorité sur les écritures en raison des accès mémoire. La LU est implémentée sous la forme d'une machine à état qui :
  1. Attend une requête load
  2. Demande à la MMU de traduire les adresses
  3. Effectue la demande de chargement auprès de la MMU et attend son feu vert
  4. Si la donnée n'est pas présente dans le cache un MISS est renvoyé à la LU qui annule la transaction puis attend avant de recommencer, sinon démarre une nouvelle transaction.
- La Store Unit gère les instructions d'écriture en mémoire. Elle utilise également pour ce faire une machine à états et contient un buffer qui permet à la SU de se stocker les opérations de stockage réalisées. La machine à états :
  1. Attends le signal de validité, et vérifie la validité de l'adresse traduite Si ce n'est pas le cas, re demande la traduction de l'adresse. On vérifie également que le buffer d'écriture postées n'est pas plein avant d'avancer. Si c'est le cas, on est obligés d'interrompre le processeur.

## 2. L'instruction est postée dans le buffer d'écriture

Le buffer d'écritures est utilisé par l'étage commit lors qu'il valide les opérations. Il est implémenté comme une FIFO ce qui assure qu'une instruction ne reste pas non traitée trop longtemps.

- La MMU ou Memory Management Unit est responsable de la traduction des adresses physiques en adresses virtuelles et des accès mémoires.



Source : Schéma fourni par les concepteurs du CVA6, [lien](#)

Elle est en mesure de faire ce travail par l'intermédiaire des DTLB et ITLB pour Data/Instruction Translation Lookaside Buffer qui agissent comme des tables liant chaque adresse physique à une adresse virtuelle. Pour chaque couple d'adresse, les TBL disposent de flags permettant par exemple de gérer les exceptions.

En plus des deux TLB, la MMU est composée du PTW pour Page Table Walker. Ce composant est connecté aux deux TLB et fait l'interface avec le cache de données. Lors d'une demande de traduction, si cette dernière ne peut pas être satisfaite par le TLB dont elle émane, le PTW parcourt les pages qui constituent la mémoire à la recherche d'une table contenant l'adresse recherchée. Toutes les pages "visitées" par le PTW sont envoyées dans le ITLB. C'est la MMU qui, à chaque cycle, vérifie si la page contient l'adresse voulue et interrompt le PTW.

**2 - Bloc multiplication et division** Le multiplicateur gère toutes les instructions liées à la division et à la multiplication c'est à dire les instructions : MUL, MULH, MULHU, MULHSU, MULW (multiplication); DIV, DIVU, DIVW, DIVUW (division) et REM, REMU, REMW, REMUW (modulo). Il est constitué d'un module de multiplication, et d'un module de division. La sélection du résultat se fait à ce niveau, à l'aide d'un multiplexeur contrôlé par un signal représentant le type d'opération effectuée (à 1

si l'opération demandée correspond à MUL ou ses dérivés).

**Bloc Multiplication** Le module de multiplication est chargé des opérations de multiplication signée et non signée des opérandes. Il implémente un pipeline d'opérations qui permet l'exécution de la multiplication en deux cycles seulement.

1. Dans un premier temps, le multiplicateur sélectionne le signe à appliquer, en fonction de l'opération véhiculée par le champ operator des données reçues.

Les opérations MUL, MULHU et MULW sont des opérations non signées. Le bit de signe des opérandes est donc ignoré en le combinant avec un 0 via un ET logique ( $x \text{ AND } 0 = 0$ ).

Pour une opération MULHSU, on multiplie un nombre signé (opérande A) et un non signé (opérande B). On conserve donc le bit de signe de la première opérande à l'aide d'un ET logique avec 1 ( $x \text{ AND } 1 = x$ ), et on ignore le bit de signe de la seconde opérande.

De même, pour un MULH on multiplie deux signés, et on conserve donc les deux bits de signes.

2. Le multiplicateur concatène ensuite les opérandes avec les bits de signes ainsi obtenus pour effectuer la multiplication signée des opérandes.

3. Enfin, le résultat en sortie est sélectionné en fonction du type d'opération.

En effet, MULH, MULHU et MULHSU sont des opérations sur les moitiés supérieures des opérandes (MULH : MULtiply upper Half, MULHU MULtiply upper Half Uns et MULHSU MULtiply Half Sign/Uns), on sélectionne donc la moitié supérieure du résultat.

Pour MULW (MULtiply Words), on ne conserve que les 32 premiers bits du résultat. L'opération étant signée, on y ajoute le bit de signe du résultat.

Pour les autres opérations, on peut envoyer les 63 premiers bits du résultat. 63 bits afin d'éviter d'envoyer le bit de signe, qui n'est pas désiré.

**Bloc division** Le Diviseur est construit sous forme d'une machine à état qui fait des additions en série. Elle prends donc jusqu'à 64 cycles maximum pour sortir son résultat.

**3 - CSR Buffer** Lorsqu'une instruction lit ou écrit dans le registre CSR (Control Status Register), le Buffer CSR sauvegarde l'adresse du registre à laquelle l'instruction va accéder. Deux principales raisons expliquent la nécessité de disposer de ce buffer :

1. Une instruction sur le CSR nécessite la rétention de deux données distinctes : le résultat de l'opération, et l'adresse du registre qui va être écrite. Or, cette instruction transite comme toutes les autres par le scoreboard, qui ne dispose que d'un seul champ pour stocker le résultat. On ne peut donc pas y stocker à la fois le résultat et l'adresse. Utiliser le scoreboard pour y stocker les deux données nécessiterait de redessiner ce dernier afin d'accommoder un cas particulier, ce qui ne serait pas optimal. A la place, l'équipe du cva6 a décidé d'implémenter le buffer CSR pour contenir l'adresse. Ainsi, lors du commit de cette instruction, le résultat est obtenu normalement par le biais du scoreboard, et l'adresse est récupérée en sortie du buffer.
2. Etant donné que les instructions sur CSR impactent l'état de l'architecture, cette dernière doit pouvoir être gardée jusqu'à ce que l'étage commit ne décide de la valider.

Lorsque le scoreboard envoie le signal `csr_valid`, le registre considère son contenu comme valide, et sauvegarde l'adresse qui lui est présentée. Le contenu est invalidé dès lors que le signal `csr_commit` est reçu. Il signifie que la donnée a été consommée et que l'instruction relative à l'adresse a été exécutée.

---

*Il est ici important de noter que le registre CSR ne contient qu'un seul emplacement. Deux instructions CSR rapprochées peuvent donc potentiellement bloquer le processeur, le temps que le buffer se soit vidé.*

**4 - unité de branchement (Branch Unit)** La Branch Unit est chargée de gérer toutes les instructions de branchements comme les sauts (conditionnels ou non) dans le code, liés par exemple à un appel de fonction.

Pour cela, elle recalcule le next\_pc à partir du pc actuel, et d'un décalage donné. La branch unit détermine si un branchemet doit effectivement être pris ou non.

Elle intègre pour cela du matériel d'addition et de comparaison. Un des autres objectifs de la branch unit est de déterminer si une prédition de branchemet était correcte ou non, et de le signaler à l'étage PC-GEN où cette dernière se déroule. Elle applique, si nécessaire des corrections au travail de PC-GEN en modifiant la valeur du PC, et en mettant à jour la prédition de branchemets (BHT).

**5 - Unité arithmétique et Logique (ALU)** Les opérations simples sont effectuées par l'ALU, ou Arithmetic and Logic Unit.

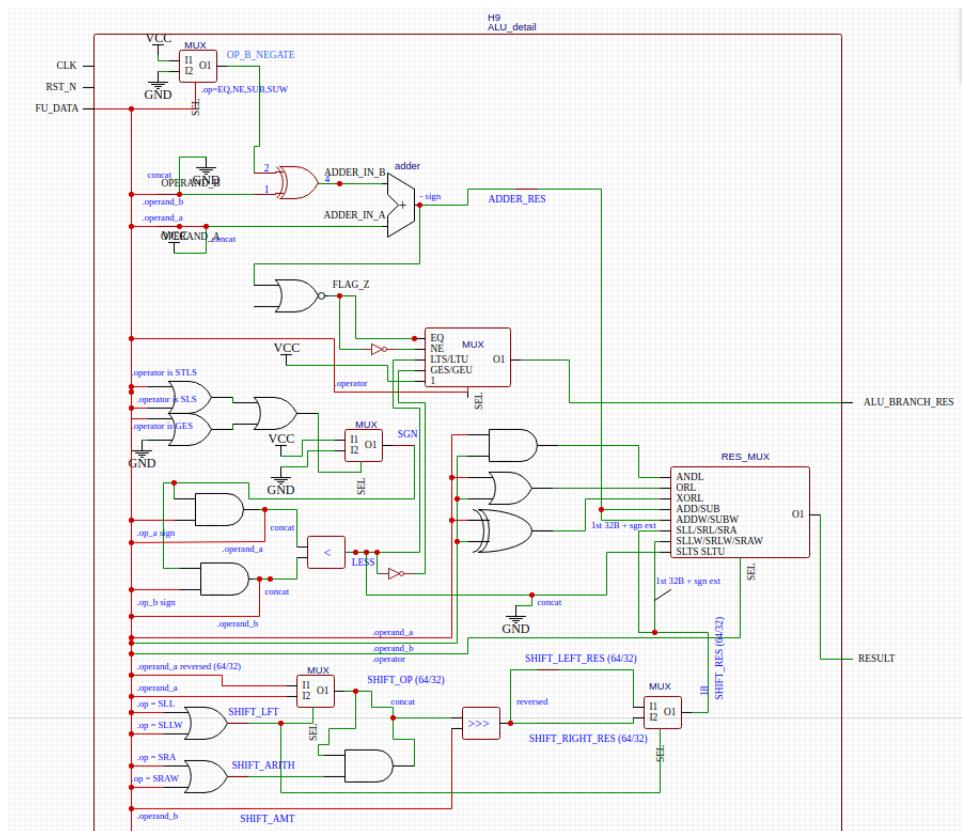


Schéma de l'ALU réalisé sous EasyEDA

Cette unité est capable de faire les additions, soustractions, comparaisons et décalages. Elle intègre en plus de cela de quoi effectuer les AND OR et XOR logiques. L'ALU a pour particularité de fonctionner uniquement avec de la logique combinatoire. Le résultat des calculs est sélectionné à l'aide d'un mulltipleur en sortie, contrôlé avec le champ de l'opcode.

- **Additions, soustractions et égalités :** On retrouve dans l'ALU un premier bloc additionneur. Ce bloc sert à la fois aux additions et aux soustractions. On effectue pour cela un traitement sur la deuxième opérande en amont du bloc : un XOR entre l'opérande B et 0 dans si l'opérateur correspond à une addition, ou 1 si l'opérateur correspond à une soustraction ou un test d'égalité permet de sélectionner une addition ou une soustraction.

La sortie de ce bloc est également utilisée pour les tests d'égalité. En effet, il suffit qu'une

différence soit égale à 0 pour qu'on sache que les deux opérandes sont égales. On obtient ainsi le flag Z.

- **Comparaisons** : Un second bloc permet de comparer deux valeurs. Il fait pour cela passer les deux opérandes dans un test d'infériorité. Une instruction testant l'infériorité est alors satisfaite en renvoyant ce résultat, et un test de supériorité est satisfait en renvoyant son inverse.
- **décalages** : Un dernier bloc permet de réaliser les opérations de décalages logiques et arithmétiques à gauche et à droite. Le type de décalage, la direction de ce dernier sont déterminés en testant directement la valeur de l'opérateur véhiculé par l'instruction. Celon la direction du décalage, l'opérande contenant la valeur à décaller est inversée (en effet, 11001 « 1 équivaut à 10011 » 1, on peut donc de cette façon effectuer à la fois des décalages à gauche et des décalages à droite avec un seul opérateur de décalage à droite). Le bit de signe est sauvegardé dans un bit supplémentaire afin de pouvoir choisir le type de bit qui sera inséré lors du décalage : 0 dans le cas d'un décalage logique (dans ce cas, le flag de décalage arithmétique vaut 0, et ainsi le bit supplémentaire vaut 0 quel que soit le signe de l'opérande) ou bien le bit de signe de l'opérande (dans ce cas, le flag de décalage arithmétique vaut 1 et laisse passer le bit de signe dans le bit supplémentaire.). L'ALU effectue un décalage arithmétique à droite à partir de cette opérande préparée, et de la seconde opérande qui contient le nombre de décalages à effectuer. En sortie, le signal est inversé afin d'obtenir un second signal représentant le décalage à gauche de notre opérande. Enfin, un multiplexeur sélectionne ce signal inversé ou la sortie "brute" du décallleur en fonction de la valeur de l'opérateur.

## 3.5 Etage 6 : Commit - Yann

### 3.5.1 Objectifs

L'étage Commit est le dernier étage du pipeline du processeur servant à la validation des résultats des instructions qu'on lui donne dans le but de mettre à jour l'état de l'architecture, et de réécrire les données présentes dans les différents registres ainsi que le registre CSR (Control/Status Register). L'étage gère aussi 2 types d'instruction de retrait, une qui se contente d'écrire sur le registre l'instruction qu'on lui donne et une autre qui elle aussi écrit mais demande une vérification supplémentaire pour s'exécuter.

Cet étage gère aussi 3 types d'exceptions :

- Une exception s'est produite dans l'un des autres étages sauf PcGen qui ne peut pas en lancer.
- Une autre exception dans le cas où une exception s'est produite dans l'étage commit.
- Une exception qui gère les interruptions peut importe l'étage où elles interviennent.

### 3.5.2 Element clés et fonctionnement

Dans l'architecture du code, cet étage est le plus court en termes de contenu, il ne sert qu'à mettre à jour les registres, mais surtout à gérer les exceptions, soit celle que lui-même renvoie, ou toutes des requêtes d'exception que les autres étages peuvent lui renvoyer. Le Commit étant le dernier étage de l'architecture, il est pertinent de mettre tous les traitements d'exception à cet étage. Avoir un étage ne servant qu'à ça permet de mieux les gérer, mais est aussi un problème dans notre cas. En effet au cours d'une de nos réunions hebdomadaire d'avancement avec nos professeur encadrant, on a nous a expliquer que les exceptions sont l'un des éléments les plus compliqués à optimiser, en plus du fait que l'étude des exceptions est une notions vu au cours de la deuxième année de master. Il est du coup compliqué pour nous, sans les notions abordées dans le cadre de nos études, d'envisager de les optimiser. On a par conséquent pris la décision d'enlever l'optimisation de cet étage de notre projet.

# Chapitre 4

## Conclusion et ouverture

### 4.1 Conclusion générale

Ce projet aura été un défi de taille. Il nous a permis d'approfondir nos connaissances sur le fonctionnement du RISCV et du CVA6, et des méthodes utilisées par les architectes des processeurs afin d'optimiser l'architecture, comme le scoreboarding et la prédition de branchements.

De plus, la réalisation de cette UE nous a permis d'apprendre à travailler sur un projet de grande ampleur au sein d'une équipe de 5 personnes. Bien que nous ayons tous une expérience de travail en équipe, nous n'avions jamais travaillé dans un groupe de cette taille. Nous avons donc appris une nouvelle manière de fonctionner, et à mieux communiquer à la fois entre nous et avec nos professeurs encadrants. Nous avons été très intéressés par les aspects tant théoriques (compréhension du fonctionnement d'une architecture) que pratiques (implémentation du CVA6 et tests sur la carte Zybo).

### 4.2 Conclusions personnelles

**Nicolas** Ce projet a été très intéressant, bien que nous n'ayons pas pu proposer de solutions d'optimisation. J'ai pu approfondir mes connaissances sur l'architecture RISCV et sur le fonctionnement du pipeline.

L'étage EXE s'est révélé bien plus complexe et complet que ce que je pensais. J'ai appris qu'il ne s'agissait pas simplement d'un étage travaillant à obtenir le résultat des instructions. Il interagit avec de nombreux étages (Commit, Issue pour le scoreboard, PC-GEN pour les branchements). Le scoreboard a également été quelque chose que j'ai aimé découvrir. J'estime en effet très impressionnant les gains en cycles qu'il confère grâce à l'exécution dans le désordre.

Ce travail m'a permis de découvrir à quel genre de problématiques un architecte de processeur doit faire face, ainsi que différentes méthodes employées pour augmenter les performances d'un processeur. J'ai également pu me familiariser avec un nouveau langage, le SystemVerilog et apprendre à utiliser des outils de cross compilation (chaîne de compilation du RISCV) et de débogage avec gdb. J'ai également trouvé l'aspect interaction matériel lié à la programmation de la carte et à l'envoi de programmes au CVA6 très intéressants.

Durant ce projet, j'ai également appris à travailler de manière régulière sur un projet de longue durée et de grande ampleur, au sein d'une équipe de 5 personnes.

**Yann** Ce projet aura été enrichissant sur l'approche d'un processeur RISCV et sur le fonctionnement des pipelines. L'étude des étages COMMIT gérant les exceptions et la validation des résultats et le FRONTEND contenant Pc Gen et Instruction Fletch ou j'ai pu en apprendre plus sur les notions de prédition de branchement n'a pas été simple mais cela aura été très constructif. Le fait qu'on n'ai pas pu aller jusqu'à l'étape de l'optimisation est regrettable, mais le projet m'aura permis de travailler sur

un sujet d'une taille beaucoup plus importante que ceux sur lesquels j'ai travaillé durant ma scolarité, ainsi qu'apprendre à travailler au sein d'un groupe de 5 personnes avec le travail à distance.

**Bojana** Ce projet m'a donné l'opportunité de découvrir enfin l'ISA RISC-V dont on a beaucoup entendu parler, mais pas d'une façon approfondie pendant nos cours du premier semestre. La thèse de Waterman qui présente l'idée de création de l'ISA RISC-V m'a permis de comprendre les raisons pour lesquelles ceci est un sujet très important qui est en train de changer le monde de l'architecture en permettant d'avoir des avancements et des solutions innovantes qui peuvent être partagées librement. Au cours du semestre, le travail de compréhension du RISC-V était partagé entre les cinq membres de notre équipe et encouragé par nos deux encadrants. À notre niveau de M1, la compréhension du fonctionnement du RISC-V n'était pas évidente, mais le partage du travail entre nous, la communication et les réunions hebdomadaires m'ont permis de progresser et de vraiment approfondir certaines notions d'architecture comme la prédition de branchements.

En général, je trouve la notion de prédition très étonnante et intéressante, donc le fait de plonger dans le monde des différentes méthodes et le fonctionnement détaillé de cette technique m'a procuré un plaisir intellectuel, et a été le plus grand défi de ce semestre. Je trouve qu'on a beaucoup avancé ensemble comme une équipe et en comparant avec le début de ce semestre je suis sûre que les notions qu'on a apprises vont rester avec nous pendant longtemps.

**Wakeel** Le projet m'a beaucoup apporté sur les plans théoriques et pratiques. J'ai appris à travailler à distance et en équipe. J'ai approfondi mes connaissances sur l'architecture des pipelines d'un processeur RISC-V. La compréhension du système était très complexe, mais grâce aux enseignants encadrants, qui ont pris énormément de temps pour nous expliquer et répondre à toutes nos interrogations, j'ai fini par assimiler le fonctionnement du CVA6. Cela m'a permis de travailler sur un projet de taille importante dans des conditions professionnelles avec en même temps une grande autonomie. Malgré les restrictions dues à la crise sanitaire qui nous ont empêchées de travailler ensemble en présentiel, j'ai beaucoup apprécié travailler avec des gens que je ne connaissais pas avant le début de ce projet. Nous avons fait connaissance sur des salons vocaux puis nous nous sommes entraînés durant tout le semestre.

**Suvetha** Malgré les difficultés rencontrées pour la compréhension de mon étage, j'ai tout de même réussi à avancer dans ce projet. Ce projet m'a permis de découvrir le CVA6 et d'apprendre de nouvelles notions. A travers ce projet, j'ai eu l'occasion de travailler avec un groupe soudé et avec lequel j'ai pu apprendre à travailler à distance. De plus, les deux encadrants du projet m'ont permis de mieux comprendre certaines notions et m'aider à avancer dans les tâches qui m'ont été attribuées. J'ai pris plaisir à participer à ce projet et à travailler avec mon équipe.