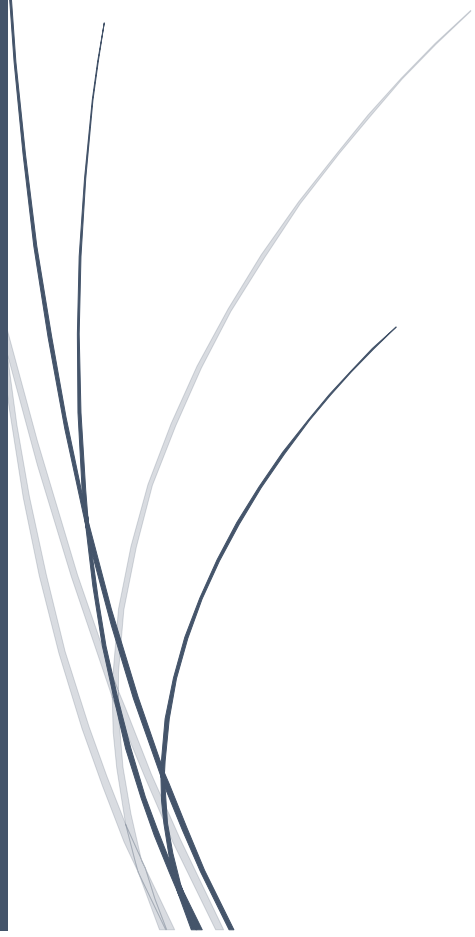
22/01/2018

RAPPORT DE PROJET

Conception d'un cœur RISC-V



Mathieu ESCOUTELOUP
Pierre GACHET
PHELMA – GRENOBLE INP

Table des matières

Table des Illustrations

I. Introduction

Aujourd'hui, les systèmes embarqués ont envahi de nombreux objets électroniques de notre quotidien, tels que les smartphones. Ceux-ci intègrent des systèmes électroniques sur puce (plus couramment appelés System-on-Chip « SoC »), servant à la réalisation de fonctionnalités particulières.

Pour gérer les différents aspects de calculs logiciels de ces systèmes embarqués, des composants bien particuliers ont été développés spécifiquement : il s'agit des processeurs. Un processeur est un composant présent dans de nombreux dispositifs électroniques qui exécute des instructions machine des programmes informatiques. Les processeurs des débuts étaient conçus spécifiquement pour un ordinateur d'un type donné. Cette méthode coûteuse de conception des processeurs pour une application spécifique a conduit au développement de la production de masse de processeurs qui conviennent pour un ou plusieurs usages. Cette tendance à la standardisation qui débuta dans le domaine des ordinateurs centraux (mainframes à transistors discrets et mini-ordinateurs) a connu une accélération rapide avec l'avènement des circuits intégrés. Les circuits intégrés ont permis la miniaturisation des processeurs. La miniaturisation et la standardisation des processeurs ont conduit à leur diffusion dans la vie moderne bien au-delà des usages des machines programmables dédiées. De nos jours, les processeurs sont devenus des microprocesseurs. C'est à dire que tous les composants ont été suffisamment miniaturisés pour être regroupés dans un unique boitier.

L'une des caractéristiques importantes des processeurs est le jeu d'instruction qu'il peut exécuter. Il peut exécuter des instructions comme : additionner deux nombres, comparer deux nombres pour déterminer s'ils sont égaux, comparer deux nombres pour déterminer lequel est le plus grand, multiplier deux nombres... Un processeur peut exécuter plusieurs dizaines, voire centaines ou milliers, d'instructions différentes.

Dans le cadre de notre dernière année en école d'ingénieur à Phelma, nous avons dû réaliser le cœur d'un microprocesseur. Le but de ce projet est d'implémenter un processeur RISC-V.

A. RISC-V

Figure 1 : Logo RISC-V

RISC-V est une architecture de jeu d'instruction 32 ou 64 bits ouverte et libre, c'est-à-dire aux spécifications ouvertes et pouvant être utilisées librement par l'enseignement, la recherche et l'industrie.

Ce projet, créé initialement dans la division informatique de l'Université de Californie à Berkeley, aux États-Unis, avait d'abord une visée d'étude et de recherche, mais est devenu de facto un standard d'architecture ouverte dans l'industrie.

L'objectif de ce jeu d'instructions est de créer un standard ouvert de jeu d'instruction de microprocesseur, à l'image du standard TCP/IP pour les réseaux ou de Linux pour le noyau, l'architecture des processeurs étant pour le moment toujours fermée. En effet, la plupart des jeux utilisés aujourd'hui étant privés (x86-64 d'Intel, ARM…), y accéder est relativement compliqué

B. Cahier des charges

Dans le cadre de notre 3e année en école d'ingénieur à Phelma, nous avons dû réaliser un projet complexe, de la phase de spécification jusqu'à sa conception.

De nos jours, comme on l'a vu avant, les processeurs sont utilisés dans presque tous les produits à base d'électronique. Ils vont permettre de faire des calculs, d'aller rechercher et enregistrer des données en mémoire … Ce composant dirige en général l'ensemble des autres modules.

L'objectif de notre projet va être alors de concevoir un processeur RISC-V pour une cible Internet des Objets. Comme dit précédemment, L'architecture RISC-V est open-source et les différentes spécifications permettant de comprendre son fonctionnement sont accessibles librement sur Internet. L'objectif de ces documents est, en bref, de fournir un ensemble d'instructions pensées et optimisées pour être facilement implémentable. Une des particularités du jeu RISC-V est son extensibilité : en effet, à partir du jeu de base (32 ou 64 bits), il est facilement possible d'ajouter des extensions (calcul de multiplications/ divisions, compression des instructions …) tout en continuant à respecter les règles de base.

Nous allons donc au cours de ce projet chercher à concevoir un processeur RISC-V 32 bits basique extensible (aucune extension ne sera de base intégrée, mais il doit être possible de le faire). Pour cela, la gestion des différentes instructions RV32I devra être gérée.

De notre point de vue, ce projet sera l'occasion de revoir l'ensemble des connaissances acquises dans notre formation dans la conception d'un circuit numérique, particulièrement concernant les architectures numériques.

Pour réaliser ce microprocesseur, nous n'aurons comme éléments que les différents documents fournis sur le site Internet RISC-V. Nous allons à partir de là devoir définir et expliquer l'ensemble de nos choix pour la bonne réalisation de notre processeur.

Voici une liste des différentes étapes qui s'enchaîneront au cours de notre projet :

- Compréhension de la l'ISA RISC-V,
- Choix du jeu d'instructions (32 ou 64 bits) et des extensions,
- Choix de l'architecture du cœur,
- Conception des différents blocs,
- Conception du système global,
- Validation des blocs et du système global,
- Synthèse et post synthèse,
- Optimisation du cœur,
- Création d'une démonstration de test,
- Validation de la plateforme de test.

De plus, nous serons contraints par le domaine d'application visé. Dans notre cas, le processeur devra pouvoir servir dans une application type IoT. Il devra donc pouvoir être embarqué sur des circuits très petits, fonctionnant sur batterie mais ne demandant pas des performances optimales (type serveur ou ordinateur). Ils peuvent par exemple être utilisés comme détecteur de luminosité ou de niveaux, basile GPS, … De ce fait, les contraintes supplémentaires à respecter sont les suivantes :

- Le coût, qui doit être le plus faible possible. A notre échelle, cela peut s'exprimer par la possibilité de réduire au maximum la surface globale du circuit.
- La taille, qui doit être elle aussi réduite au maximum, notamment pour la raison évoquée ci-dessus.
- La consommation, qui doit être réduite elle aussi au maximum pour des circuits sur batterie. Ce sont des éléments comme la fréquence de fonctionnement ou éventuellement l'architecture qui pourront influencer ce paramètre.

A l'inverse, la fréquence de fonctionnement ne sera pas un élément très contraignant ici, les demandes en performances pour des applications IoT n'étant généralement pas très élevées.

En bref, le but final sera donc d'avoir conçu un processeur fonctionnant correctement, et donc capable d'exécuter un programme relativement simple (illustré par la démonstration). Il doit être facilement réutilisable et extensible.

II. Réalisation d'un cœur RISC-V

Le but du projet est de concevoir un processeur RISC-V capable d'exécuter les différentes instructions RV32I. Pour réaliser cela, on va dans un premier temps étudier le document technique de ce jeu d'instructions. L'objectif est de comprendre comment il fonctionne, les choix qui ont été faits (et les raisons), ce qu'il implique et ensuite seulement choisir l'architecture matérielle qui nous semblera la plus adapté à notre processeur. On poursuivra par la conception des différents composants le constituant avant de finir par une validation globale du cœur. Une première synthèse logique sera également effectuée pour valider les choix effectuer au niveau de la description RTL.

A. Choix de l'architecture

Dans un premier temps, nous avons étudié la documentation technique du RISC-V pour connaître le jeu d'instructions RV32I. Celui-ci contient les instructions basiques nécessaires et suffisantes à l'implémentation d'un grand nombre de programmes. L'ISA a également été pensé et conçu pour réduire le matériel requis dans une implémentation minimale. Le jeu RV32I contient 47 instructions uniques, organisées en 6 structures différentes seulement au niveau binaire, et permettant de réaliser de multiples instructions basiques (Opérations arithmétiques, branchement, saut …).

Comme on a pu le voir, un programme va s'exécuter sur notre processeur. Celui-ci devra interpréter les instructions pour réaliser correctement la fonctionnalité souhaitée par le software. Pour exécuter ses instructions, il faut l'architecture matérielle à mettre en place. Dans notre cas, nous avons directement choisi de nous diriger vers une architecture pipeline, pour les gains en performances qu'elle offre mais aussi la facilité à segmenter le travail (les étages pourront être conçus séparément). Un pipeline est l'élément d'un processeur dans lequel l'exécution des instructions est découpée en plusieurs étapes. L'objectif est de maximiser le nombre d'instructions exécutées par cycle tout en réduisant la durée d'une période d'horloge, au détriment de la latence globale du circuit. Pour cela, plusieurs instructions seront donc traitées simultanément par le processeur (Globalement N instructions pour N étages quand celui-ci est rempli). Pour ces différents aspects, nous avons pu nous référer aux cours d'architectures numériques de seconde année.

Le choix du nombre d'étages du pipeline peut selon le matériel que l'on souhaite implémenter (les registres notamment) mais aussi la séparation entre les différentes fonctionnalités à réaliser. On peut donc avoir des pipelines de 3 étages comme de plus de 10 étages. Dans notre cas, justement en vue des contraintes matérielles mais aussi de la complexité de réalisation, notre hésitation s'est portée entre des nombres réduits d'étages. Voici un récapitulatif des avantages et inconvénients des différents pipelines nous paraissant adaptés à notre projet :

	Pipeline 3 étages	Pipeline 4 étages	Pipeline 5 étages
Points positifs	- Moins complexe - Moins de matériel	- Plus facile d'ajouter des extensions (par rapport au 3 étages) - Bon compris matériel/performance	- Plus facile d'ajouter des extensions - Accès mémoire séparé - Validation plus simple sur chaque étage
Points négatifs	- Les extensions peuvent être plus difficiles - Etage plus lent	- Accès mémoire confondu avec le rangement	- Latence et aléa - Coûts supplémentaires

Figure 2: Comparaison des différents pipelines

Notre choix s'est finalement porté vers un pipeline 5 étages. Le premier avantage de celui-ci par rapport aux autres est une séparation des différentes étapes qui nous semblaient plus facile à concevoir. Le test unitaire de chaque étage pourra donc également se faire de manière indépendante. De notre point de vue, cette plus grande séparation des étapes entraîne également une plus grande facilité à implémenter de nouvelles extensions, aspect très important du cahier des charges spécifié en début de projet.

Voici les différents étages que nous implémenteront pour la réalisation de ce pipeline 5 étages :

- IF (Instruction Fetch) charge l'instruction à exécuter dans le pipeline depuis la mémoire d'instructions.
- ID (Instruction Decode) décode l'instruction et récupère les valeurs correspondantes dans les registres.
- EX (Execute) exécute l'instruction (par la ou les unités arithmétiques et logiques), en réalisant les différentes opérations arithmétiques nécessaires (à l'aide d'unité(s) arithmétique et logique).
- MEM (Memory), réalise un transfert depuis un registre vers la mémoire dans le cas d'une instruction du type STORE (accès en écriture) et de la mémoire vers un registre dans le cas d'un LOAD (accès en lecture). Il est traversnt pour les autres instructions.
- WB (Write Back) stocke le résultat dans un registre. La source peut être une donnée mémoire ou le résultat d'une opération.

B. Réalisation du corps du processeur

Comme vu précédemment, nous allons utiliser une architecture basée sur un pipeline 5 étages.

Programme	Cycles d'horloge									
	1	2	3	4	5	6	7	8	9	10
Inst. n° 1	IF	ID	IE	MA	WB					
Inst. n° 2		IF	ID	IE	MA	WB				
Inst. n° 3			IF	ID	IE	MA	WB			
Inst. n° 4				IF	ID	IE	MA	WB		
Inst. n° 5					IF	ID	IE	MA	WB	
Inst. n° 6						IF	ID	IE	MA	WB

Figure 3 : Tableau d'exécution d'instructions

Lorsqu'une instruction arrive sur le processeur, elle va donc normalement s'exécuter en 5 cycles d'horloge (1 cycle par étage du pipeline). Dès qu'un étage a fini de traiter l'instruction, il la transmet à l 'étage suivant. Ce même étage recommence directement à traiter une nouvelle instruction avant même que la précédente ait fini de s'exécuter dans l'ensemble du pipeline. Ceci nous permet donc traiter donc de traiter 5 instructions simultanément à partir du 5ème cycle. Nous allons à présent voir plus en détails le rôle et la conception de chaque étage.

1. Fetch

Le fetch correspond à la lecture de l'instruction (Instruction Fetch) qui arrive à l'entrée du processeur depuis la mémoire d'instructions. Cette instruction est chargée à partir de la case mémoire pointée par le compteur de programme PC disponible en entrée de l'étage :

```
seq : process (i_clk, i_rstn)
    begin
        if (i_rstn = '0') then
            o_pc <= c_PC_INIT;
            o_inst <= c_REG_INIT;
            o_validity <= '0';
        elsif (i_clk'event and i_clk = '1' and i_freeze = '1' and i_load_dependency = '0') then
            o_pc <= i_pc;
            o_inst <= i_data;
            o_validity <= s_validity_inputs;
        end if;
end process seq;
```

Figure 4 : Conception de l'étage FETCH : fetch.vhd

En parallèle de ce chargement de l'instruction dans le pipeline, il est nécessaire de calculer l'adresse de l'instruction à charger dans le cycle suivant.

```
-- Combinatorial Calculation of the Program Counter
s_pc <= i_newpc when (i_jump or i_branch) = '1' else
        s_pc_final + c_PC_STEP;

-- Assignments of the registers
seq : process (i_clk, i_rstn)
    begin
        if (i_rstn = '0') then
            s_pc_final <= c_PC_INIT;
        elsif (i_clk'event and i_clk = '1' and i_freeze = '1' and i_load_dependency = '0') then
            s_pc_final <= s_pc;
        end if;
end process seq;

-- Assignment of the output
o_pc <= s_pc_final;
```

Figure 5 : Conception de l'étage de calcul du PC: pc.vhd

Généralement, la valeur du PC prend s'incrémente d'une valeur de 4. En effet, les adresses étant des adresses octets et nos instructions étant sur 32 bits (4 octets), on se déplace de 4 en 4 dans la mémoire d'instructions. Il arrive cependant qu'un saut ou un branchement soit réalisé et donc l'incrément doit pouvoir être fait avec une valeur différente.

2. Decode

Le Décodage de l'instruction (Instruction Decode) est l'étape suivant du fetch où les données des registres utilisées sont récupérées.

Dans le jeu d'instructions RV32I, il existe plusieurs formats d'instructions, comme indiqué dans la figure suivante. Tous ont bien évidemment une longueur fixe de 32 bits

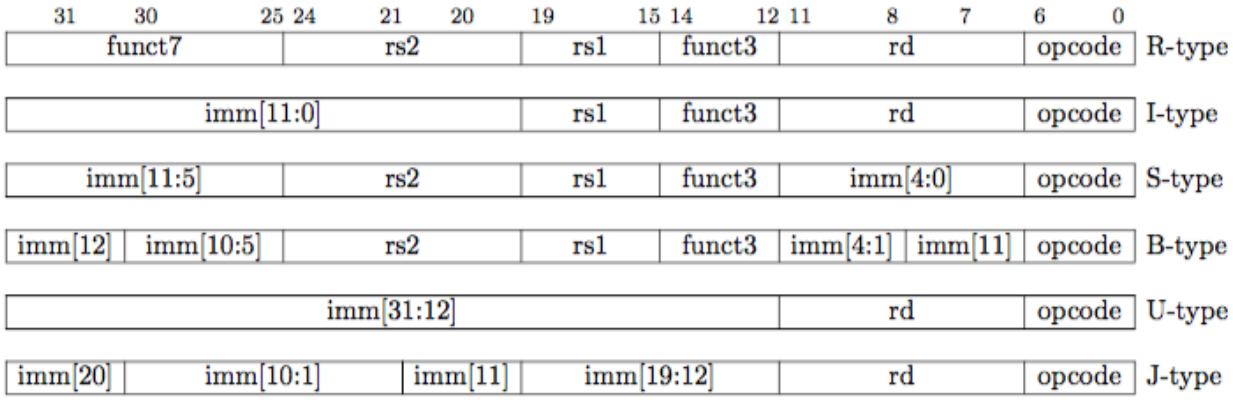


Figure 6 : Formats des instructions RV32I

On peut voir ci-dessus les différentes structures des instructions 32 bits. Ces informations vont nous permettre de réaliser l'étage de décodage, en sachant quels bits sont utilisés pour la sélection de l'opération, des registres sources, du registre de destination etc …. Le rôle de cet étage est justement de récupérer les données correspondantes aux registres spécifiés dans l'instruction. Celles-ci serviront ensuite de paramètres de l'instruction pour les opérations des étapes suivantes.

Détaillons un peu certaines informations fournies par la figure précédente :

- Le registre source 1 est codé des bits 19 à 15,
- Le registre source 2 est codé des bits 24 à 20,
- Le code du type de l'instruction est stocké des bits 6 à 0,
- La fonctionnalité précise est codée dans les bits 14 à 12
- …

Il est essentiel de comprendre l'organisation de ces instructions car c'est elle qui nous permettra de récupérer les bonnes informations. Ainsi, cet étage sera capable d'identifier le type de l'instruction à traiter, d'identifier les registres sources et de récupérer les valeurs correspondantes.

Après avoir compris la façon de décoder une instruction, on a réalisé l'étage de la manière suivante :

```
when c_OPCODE32_OP_IMM |  c_OPCODE32_LOAD | c_OPCODE32_JALR =>  -- I-type Format
    s_rs1select <= i_inst(19 downto 15);
    s_rs2select <= "00000";
    s_rdselect <= i_inst(11 downto 7);
    s_validity_global <= s_validity_inputs;
    if i_inst(6 downto 0) = c_OPCODE32_LOAD then
        s_load <= '1';
    else
        s_load <= '0';
    end if;
when c_OPCODE32_OP =>                                    -- R-type Format
    s_rs1select <= i_inst(19 downto 15);
    s_rs2select <= i_inst(24 downto 20);
    s_rdselect <= i_inst(11 downto 7);
    s_load <= '0';
    s_validity_global <= s_validity_inputs;
```

Figure 7 : Conception de l'étage DECODE : decode.vhd

La première étape a été de regrouper ces différentes instructions par format. Après avoir fait cela, on peut alors retrouver les éléments intéressants présents dans l'instruction selon son format. Sur la figure précédente, on peut voir le décodage de deux formats d'instructions différents : I et R. Le format I n'utilise qu'un seul registre source alors que le R n'en utilise 2. Lors qu'une instruction n'utilise pas l'un des registres rs1, rs2 ou rd, on met celui-ci à 0 par défaut.

Après avoir obtenues les adresses des registres sources et destination, on peut passer à l'étage d'exécution.

3. Execute

1. *Les différentes opérations*

L'exécution de l'instruction (Instruction Execution) est le 3e étage. Cette étape utilise l'unité arithmétique et logique pour combiner les arguments. L'opération précise réalisée par cette étape dépend du type de l'instruction. Dans notre processeur, on veut exécuter seulement les instructions de la norme RV32I.

Il y a plusieurs types d'instructions et donc d'opérations, que nous avons extraits de la spécification RISC-V et que nous allons reprendre ici :

- Instructions de calcul d'entiers sur 32 bits (OR, XOR, …) entre valeur immédiate et registre (format I).

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Figure 8 : Instructions OP-IMM

L'instruction ADDI ajoute la valeur 12-bit signée immédiate au registre source 1. L'instruction SLTI compare deux entiers 32 bits. Les instructions ANDI, ORI, XORI effectuent des opérations logiques basiques bit à bit entre deux données. Le registre est évidemment toujours spécifié.

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Figure 9 : Instructions OP-IMM : les décalages

Les décalages d'une certaine valeur N sont codés dans le format de type I. Ces instructions effectuent donc un décalage de N bits vers la gauche ou la droite.

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Figure 10 : Instructions LUI / AUIPC

LUI et AUIPC permettent de stocker des valeurs immédiates ou la valeur du compteur de programme dans un registre.

- Opérations arithmétiques et logiques sur des entiers contenus dans des registres. C'est ici le format R qui est utilisé. Toutes les opérations lisent les registres sources 1 et 2 en tant qu'opérandes et écrivent le résultat dans le registre de destination. Les champs « funct7 » et « funct3 » sélectionnent les différentes opérations possibles.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Figure 11 : Instructions OP

Ainsi, il est là encore possible de réaliser des additions, soustractions, décalages etc …

- Les instructions de chargement/ rangement en mémoire (LOAD et STORE). Ce sont les seules instructions qui permettent d'accéder à la mémoire de données. Elles sont organisées dans deux formats différents (respectivement I et S). Des calculs sont cependant nécessaires pour ce type d'instructions, pour déterminer notamment l'adresse mémoire à laquelle on cherche à accéder.

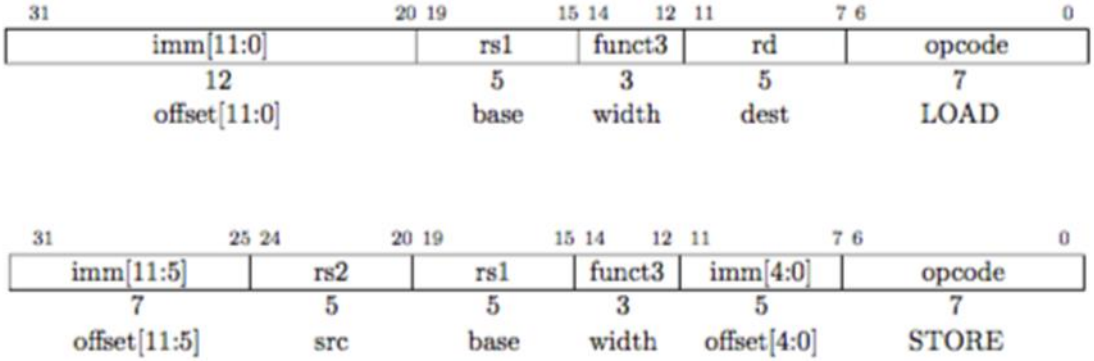


Figure 12: Instructions LOAD & STORE

On notera qu'il est possible de réaliser différents types de LOAD/ STORE : en effet il est possible de charger/ stocker seulement 1 ou 2 octets. On parle alors d'octet (8 bits), de demi-mot (16 bits) et de mot (32 bits).

- Instructions de saut inconditionnel.

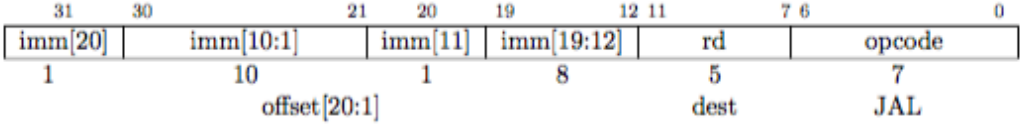


Figure 13 : Instruction JAL

L'instruction JAL stocke l'adresse de l'instruction suivant le saut (pc + 4) dans le registre rd.

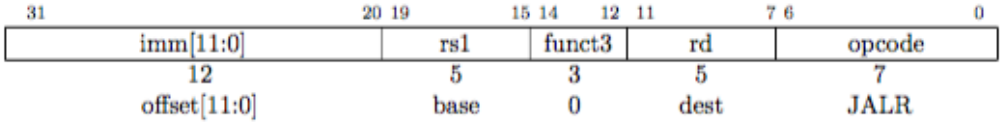


Figure 14 : Instruction JALR

L'instruction de saut indirect JALR utilise le codage de type I. L'adresse cible est obtenue en ajoutant le I-immediate signé 12 bits au registre source 1.

- Instructions de branchement conditionnel

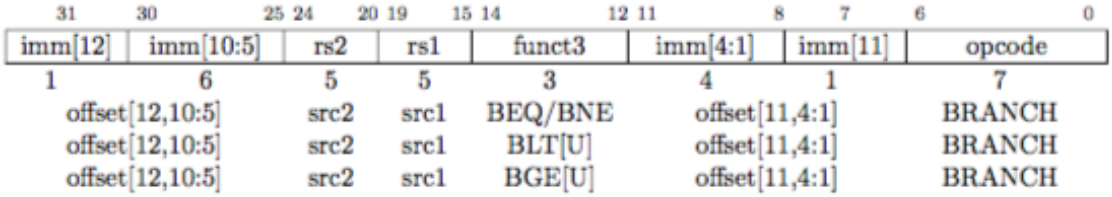


Figure 15 : Instruction de branchement

Les instructions BEQ et BNE prennent l'adresse cible si les registres rs1 et rs2 sont respectivement égaux ou inégaux. BLT et BLTU la prennent si rs1 est inférieure à rs2 et BGE si rs1 est supérieure ou égale à rs2.

C'est donc l'ensemble de ces instructions que nous allons devoir traiter dans notre cœur. Ceci influence donc directement les opérations arithmétiques et logiques que notre étage « execute » (et donc l'ALU) doivent être capables de réaliser. Comprendre ces différentes instructions ainsi que leur organisation (avec le regroupement par format) est également le meilleur moyen d'effectuer un traitement des instructions efficaces et optimisés. Le fait que les bits de codage des opérations (ainsi que la valeur même de ces bits de codage) soient toujours placés à des endroits identiques n'est pas anodin : ceci permet par exemple d'un point de vue matériel de réduire les multiplexeurs à implémenter. Globalement, l'objectif est qu'un bit N servent le plus souvent possible à coder la même chose (un numéro de registre, une puissance 2^N d'un immédiat etc …) pour réduire au maximum les interconnexions.

2. *Conception du bloc*

Après avoir vu en détails l'ensemble des instructions à implémenter dans le processeur, nous allons passer à l'étape de conception de l'étage en lui-même.

Nous avons décidé de réaliser l'ALU séparément dans un sous-bloc, pour une facilité d'écriture du code. Nous nous retrouvons donc avec l'organisation de l'étage suivante :

- L'« execute » qui identifie les différentes parties de l'instructions : les opérandes et l'opération à effectuer,
- L'ALU, qui effectue l'opération en fonction des valeurs de ses entrées,
- et enfin l'étage d'« execute » à nouveau qui récupère et gère le résultat.

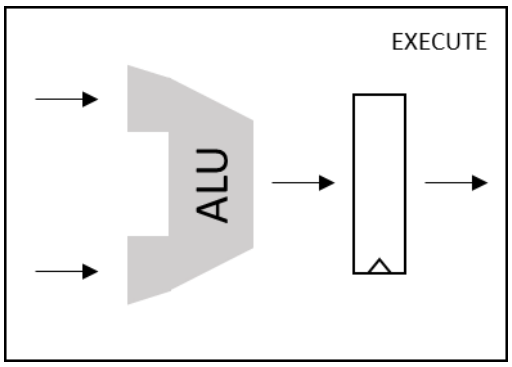


Figure 16: Organisation de l'étage EXECUTE

```
when c_OPCODE32_OP =>                                   -- Operations on Registers
    s_validity_global   <= i_validity;
    s_sel               <= i_inst(14 downto 12);
    s_op1               <= i_rs1;
    s_op2               <= i_rs2;
    s_signed            <= i_inst(30);
    s_amount            <= i_rs2(4 downto 0);
    s_jump              <= '0';
    s_jumpr             <= '0';
    s_branch            <= '0';
```

Figure 17 : Conception de l'étage EXECUTE : execute.vhd

Comme on peut le voir sur l'image ci-dessus, la première étape réalisée dans l'execute est l'analyse de l'opcode. En fonction de la valeur de celui-ci, l'exécute interprète différemment les bits composant l'instruction. Ici, on regarde cette interprétation pour les instructions d'opérations entre registres, caractérisées par l'opcode « c_OPCODE32_OP ».

On constate également que d'autres signaux intermédiaires comme s_jump et s_branch sont utilisés. Ils permettent, en cas d'instructions de branchement ou de saut, de rediriger différemment la sortie. En effet, pour un saut non conditionnel, deux valeurs sont calculées : l'adresse du saut à effectuer mais aussi l'adresse de retour. Et c'est justement cette dernière qui sera stockée dans le registre de destination, alors que l'ALU va elle calculer l'adresse du

saut. Une certaine redirection des données sur les sorties est donc nécessaire dans ce cas particulier. De plus, comme nous l'avons dit précédemment, dans le cas d'une instruction de type saut ou branchement, il est nécessaire d'indiquer au compteur de programme de ne pas effectuer une incrémentation classique, mais plutôt de prendre en compte l'adresse nouvellement calculée. On utilise donc à ce moment-là de nouveau les signaux s_jump et s_branch pour alerter le compteur.

3. *Conception de l'ALU*

Une ALU (Unité Arithmétique et Logique) est l'organe du microprocesseur chargé d'effectuer les calculs. Dans notre projet, l'ALU est un sous-bloc interne de l'execute. Ce dernier a nécessairement besoin d'une ALU pour effectuer les différentes opérations nécessaires à l'exécution des instructions. Cette ALU étant purement combinatoire et intégrée dans l'execute, ils ne représentent donc ensemble qu'un seul étage du pipeline.

L'avantage d'avoir séparé l'execute de l'ALU est qu'ils peuvent être conçu et testé de façon indépendantes : on vérifie dans un premier temps que l'ALU effectue correctement l'ensemble des instructions avant de la connecter au sein de l'étage. Cela facilite la conception de cet étage et réduit la probabilité d'erreur.

L'ALU calcule sur des nombres entiers, et peut effectuer les opérations communes sur des valeurs signées et non-signées, que l'on peut séparer en quatre groupes :

- Les opérations arithmétiques : addition, soustraction, changement de signe, etc,
- Les opérations logiques : compléments à un, à deux, et, ou, ou-exclusif, non, non-et, etc,
- Les comparaisons : inférieur supérieur et/ ou égal,
- Les décalages : de N bits vers la droite ou la gauche, avec $0 \leq N \leq 31$.

```
case i_sel is
        when c_ALU_ADD =>               -- Addition / Substraction
                if i_signed = '0' then
                        s_result <= i_op1 + i_op2;
                else
                        s_result <= i_op1 - i_op2;
                end if;
        when c_ALU_SLL =>               -- Left Shift (Logical)
                for I in 31 downto 0 loop
                        if I < to_integer(unsigned(i_amount)) then
                                s_result(I) <= '0';
                        else
                                s_result(I) <= i_op1(I - to_integer(unsigned(i_amount)));
                        end if;
                end loop;
```

Figure 18 : Conception de l'ALU : alu.vhd

Voici ci-dessus une partie de l'ALU illustrer notre façon de procéder. A partir des valeurs reçues en entrées, l'opération voulue sera effectuée avant de retourner la valeur en sortie.

4. *Intégration de l'ALU dans l'étage*

L'ALU et l'execute sont donc décrits dans deux fichiers séparés, or ils fonctionnent ensemble. Il faut alors les connecter pour qu'ils puissent communiquer et s'envoyer des informations. Une instanciation de l'unité arithmétique et logique est alors nécessaire :

```
alu1 : alu port map (   i_op1       => s_op1,
                        i_op2       => s_op2,
                        i_signed    => s_signed,
                        i_amount    => s_amount,
                        i_sel       => s_sel,
                        o_result    => s_result);
```

Figure 19 : Component ALU

Sur la figure ci-dessus, on peut voir la connexion des ports de l'ALU. On retrouve ici les mêmes signaux contenant les opérandes que ceux vus précédemment.

4. Accès mémoire

Le jeu d'instruction RISC-V impose une architecture de type LOAD - STORE, où seules les instructions de chargement et de stockage peuvent accéder à la mémoire de données. Le reste des instructions utilisent uniquement les différents registres du processeur. Toutes les adresses étant calculées sur 32 bits, le jeu d'instruction fournit un espace d'adressage utilisateur de 2^32 bits adressé par octet. L'environnement d'exécution définira quelles parties de l'espace d'adresse sont autorisées à accéder.

```
when c_OPCODE32_LOAD =>             -- LOAD
    o_write <= '0';
    o_size <= c_MEM_SIZEW;
    case i_inst(14 downto 12) is
        when c_FUNC3_LB =>                  -- Byte
            s_rd(7 downto 0) <= i_data(7 downto 0);
            s_rd(31 downto 8) <= (others => i_data(7));
            s_validity_global <= i_validity;
        when c_FUNC3_LH =>                  -- Half-Word
            s_rd(15 downto 0) <= i_data(15 downto 0);
            s_rd(31 downto 16) <= (others => i_data(15));
            s_validity_global <= i_validity;
        when c_FUNC3_LW =>                  -- Word
            s_rd <= i_data;
            s_validity_global <= i_validity;
            when c_FUNC3_LBU =>             -- Byte Unsigned
            s_rd(7 downto 0) <= i_data(7 downto 0);
            s_rd(31 downto 8) <= (others => '0');
            s_validity_global <= i_validity;
        when c_FUNC3_LHU =>                 -- Half-Word Unsigned
            s_rd(15 downto 0) <= i_data(15 downto 0);
            s_rd(31 downto 16) <= (others => '0');
            s_validity_global <= i_validity;
        when others =>
            s_rd <= i_data;
            s_validity_global <= '0';
```

Figure 20 : Conception de l'étage MEMORY ACCESS (LOAD) : memory_access.vhd

Les instructions de type LOAD sont les seules instructions capables de récupérer des données en mémoire pour les stocker dans un registre. Pour se faire, lorsque le processeur doit exécuter une instruction de chargement, elle accède directement à la mémoire. Suivant la structure de l'instruction les données, à récupérer peuvent être différentes :

- La taille de la donnée peut être de 8, 16 ou 32 bits,
- La donnée peut être considérée comme signée ou non-signée.

Tous ces cas de la spécification RV32I doivent être géré pour que la donnée récupérée dans la mémoire ne soit pas erronée.

Maintenant que nous avons vu comment récupérer une donnée, nous allons regarder comment écrire dans la mémoire de données.

```
        when c_OPCODE32_STORE =>            -- STORE
            s_rd <= i_rd;
            case i_inst(14 downto 12) is
                when c_FUNC3_SB =>          -- Byte
                    o_write <= '1';
                    o_size <= c_MEM_SIZEB;
                    s_validity_global <= i_validity;
                when c_FUNC3_SH =>          -- Half-Word
                    o_write <= '1';
                    o_size <= c_MEM_SIZEH;
                    s_validity_global <= i_validity;
                when c_FUNC3_SW =>          -- Word
                    o_write <= '1';
                    o_size <= c_MEM_SIZEW;
                    s_validity_global <= i_validity;
                when others =>
                    o_write <= '0';
                    o_size <= c_MEM_SIZEW;
                    s_validity_global <= '0';
            end case;
        when c_OPCODE32_LUI | c_OPCODE32_OP | c_OPCODE32_OP_IMM | c_OPCODE32_AUIPC | c_OPCODE32_JAL | c_OPCODE32_JALR | c_OPCODE32_BRANCH =>

            o_write <= '0';
            o_size <= c_MEM_SIZEW;
            s_rd <= i_rd;
            s_validity_global <= i_validity;
        when others =>
            o_write <= '0';
            o_size <= c_MEM_SIZEW;
            s_rd <= i_rd;
            s_validity_global <= '0';
    end case;
```

Figure 21 : Conception de l'étage MEMORY ACCESS (STORE) : memory_access.vhd

Les instructions de type STORE sont les seules à pouvoir écrire dans la mémoire. Un signal de sortie o_write de type std_logic est utilisé pour différencier une demande de lecture d'une demande d'écriture en mémoire (écriture pour o_write = '1'). Tout comme pour le LOAD, il est possible de manipuler différentes tailles de données. Il est donc ici aussi nécessaire de gérer ces différents cas.

Note : Cette différenciation entre la taille des données manipulées se retrouve également lors de l'utilisation de variables en langage de haut niveau. En effet, en langage C par exemple, une variable char (8 bits) n'occupe pas la même place qu'un int (16 bits) ou qu'un long (32 bits). Cette souplesse dans la longueur de données est essentielle pour utiliser au mieux l'ensemble de l'espace mémoire.

5. Write Back

L'étage writeback est le dernier étage de notre pipeline. Le but de cet étage est de stocker le résultat des différentes opérations dans le registre de destination spécifié dans l'instruction.

```
case i_inst(6 downto 0) is          -- Writing in Integer Registers for the different instructions
    when c_OPCODE32_LUI | c_OPCODE32_AUIPC =>                       -- Load Upper Immediate / Add Upper Immediate to Program Counter
        o_write             <= '1';
        o_rdselect          <= i_inst(11 downto 7);
        o_data              <= i_rd;
    when c_OPCODE32_OP_IMM | c_OPCODE32_OP | c_OPCODE32_LOAD =>     -- Operations / Load
        o_write             <= '1';
        o_rdselect          <= i_inst(11 downto 7);
        o_data              <= i_rd;
    when c_OPCODE32_STORE =>                                        -- Store
        o_write             <= '0';
        o_rdselect          <= "00000";
        o_data              <= i_rd;
    when c_OPCODE32_JAL | c_OPCODE32_JALR =>                        -- Jump And Link
        o_write             <= '1';
        o_rdselect          <= i_inst(11 downto 7);
        o_data              <= i_rd;
    when c_OPCODE32_BRANCH =>                                       -- Branch
        o_write             <= '0';
        o_rdselect          <= "00000";
        o_data              <= i_rd;
    when others =>
        o_write             <= '0';
        o_rdselect          <= i_inst(11 downto 7);
        o_data              <= i_rd;
end case;
```

Figure 22 : Conception de l'étage WRITE BACK: writeback.vhd

Note : Bien qu'accessible par une instruction, le registre x0 ne peut être modifié. En effet, la spécification RISC-V prévoît que ce registre contienne tout le temps la valeur 0x00000000.

Un accès en écriture au bloc contenant des registres est alors nécessaire. En effet, nous avons ici aussi décidé de séparé les registres du reste du pipeline. Plusieurs raisons expliquent notre choix :

- Une certaine facilité de conception. En effet, un fichier décrivant uniquement le fonctionnement de ces registres était facilement créé. Ceci nous a donc aidé à valider le fonctionnement de l'étage de decode, qui communique lui aussi avec ces registres pour récupérer les différentes valeurs. A l'inverse, si nous avions fusionné les registres avec l'étage de writeback, nous aurions dû attendre d'avoir deux étages assez fonctionnels pour réaliser de vraies vérifications.
- La spécification RV32I prévoit l'ajout de registres pour l'implémentation de nouvelles extensions : le calcul des flottants et l'ajout de modes privilégiés notamment. En séparant ces registres du reste du pipeline, il est plus facile d'en ajouter des nouveaux sans avoir à modifier en profondeur les étages du pipeline concernés (et donc avec le risque de commettre des erreurs).

La méthode d'écriture dans ce bloc de registres est globalement semblable à celle pour écrire en mémoire : un signal o_write distingue une écriture d'une lecture et le numéro sur 5 bits du registre de destination est comparable à une adresse.

```
architecture reg_integer_arch of reg_integer is
    type regfile is array (0 to c_NREGISTERS - 1) of std_logic_vector(c_NBITS - 1 downto 0);
    signal r_integers : regfile;

    begin
        -- Sequential Logic
        -- Writing and Resetting
        seq : process(i_clk, i_rstn)
            begin
            if (i_rstn = '0') then
                for I in 0 to c_NREGISTERS - 1 loop
                    r_integers(I) <= c_REG_INIT;
                end loop;
            elsif (i_clk'event and i_clk = '1' and i_freeze = '1') then
                if ((i_write = '1') and (to_integer(unsigned(i_rdselect)) /= 0)) then
                    r_integers(to_integer(unsigned(i_rdselect))) <= i_data;
                end if;
            end if;
        end process seq;

        -- Combinatorial Logic
        -- Reading
        o_rs1 <= r_integers(to_integer(unsigned(i_rs1select)));
        o_rs2 <= r_integers(to_integer(unsigned(i_rs2select)));
end reg_integer_arch;
```

Figure 23 : Conception des registres d'entiers REG_INTEGER: reg_integer.vhd

Lorsque que le module writeback met à '1' sa sortie o_write, cela modifie la valeur de l'entrée i_write du bloc de registres. La donnée en entrée est alors écrite dans le registre correspondant.

6. Pipeline

Nous avons à présent détaillé la conception individuelle de chaque étage du pipeline. Il est donc à présent nécessaire de réaliser un fichier top qui comprendra une instance de chaque étage, ainsi que les connexions nécessaires.

```
212         pc1 : pc port map ( i_rstn              => s_rstn,
213                             i_clk               => s_clk,
214                             i_jump              => s_exec_jump,
215                             i_branch            => s_exec_branch,
216                             i_freeze            => s_freeze,
217                             i_load_dependency   => s_dcde_load_dependency,
218                             i_newpc             => s_exec_newpc,
219                             o_pc                => s_pc);
220
221         fetch1 : fetch port map (   i_rstn              => s_rstn,
222                                     i_clk               => s_clk,
223                                     i_pc                => s_pc,
224                                     i_data              => s_imem_idata,
225                                     i_jump              => s_exec_jump,
226                                     i_branch            => s_exec_branch,
227                                     i_freeze            => s_freeze,
228                                     i_load_dependency   => s_dcde_load_dependency,
229                                     o_addr              => s_ftch_addr,
230                                     o_data              => s_ftch_odata,
231                                     o_write             => s_ftch_write,
232                                     o_size              => s_ftch_size,
233                                     o_pc                => s_ftch_pc,
234                                     o_inst              => s_ftch_inst,
235                                     o_validity          => s_ftch_validity);
236
237         reg_integer1 : reg_integer port map (   i_rstn      => s_rstn,
238                                                 i_clk       => s_clk,
239                                                 i_freeze    => s_freeze,
240                                                 i_rs1select => s_regi_rs1select,
241                                                 i_rs2select => s_regi_rs2select,
242                                                 o_rs1       => s_regi_rs1,
243                                                 o_rs2       => s_regi_rs2,
244                                                 i_write     => s_wbck_write,
245                                                 i_rdselect  => s_wbck_rdselect,
246                                                 i_data      => s_wbck_data);
```

Figure 24: Connexion des différents éléments et création du Top

A l'aide de différents signaux instanciés dans ce fichier top (nommé pipeline.vhd), on va connecter les ports des différents étages, mais aussi ceux du top permettant un accès vers l'extérieur et donc les mémoires, entre eux. Ainsi, on arrivera à reformer l'ensemble du pipeline décrit ci-dessous :

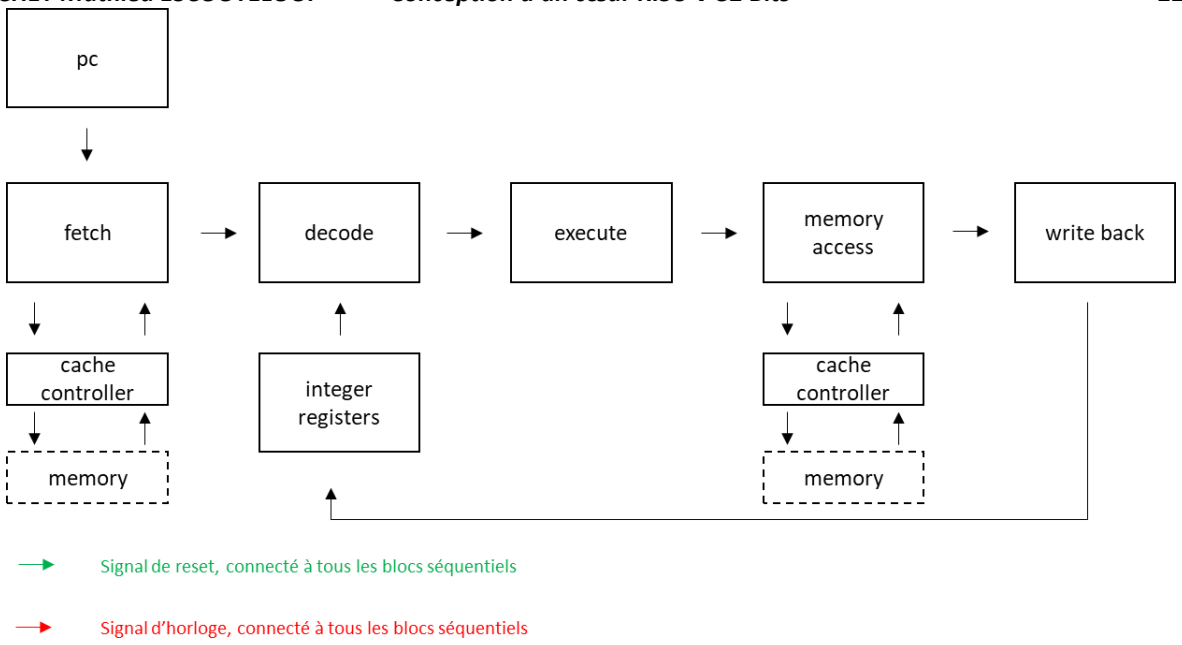


Figure 25: Organisation du fichier pipeline.vhd

L'organisation au niveau top est ainsi détaillée. Il est important de préciser que les mémoires sont représentées différemment car elles n'appartiennent pas au système : ce sont des éléments extérieurs au processeur.

Toutes les briques de base composant notre cœur RISC-V ont ainsi été instanciées et connectées. Après une validation poussée de chaque élément indépendamment, il faudra procéder à une validation du fonctionnement du système global.

7. Dépendances de Données

Bien que ce processeur possiblement fonctionnel dans l'état actuel, il est intéressant, une fois l'architecture globale de base bien établie, de penser à certains cas de fonctionnement particulier en vue d'une augmentation des performances. Voyons notamment le cas de l'exécution de la série de pseudo-instructions suivante :

- inst R3, R1, R2
- inst R6, R5, R4
- inst R7, R3, R6

Pour rappel, voici l'organisation générale d'un pipeline 5 étages basique :

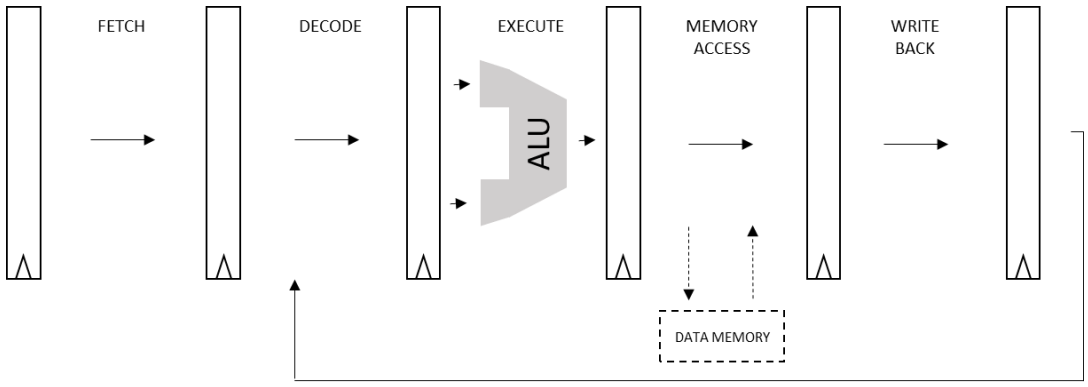


Figure 26: Rappel du Fonctionnement d'un pipeline 5 étages basique

Les résultats des différentes instructions sont respectivement stockés dans les registres R3, R6 et R7. Dans un pipeline comme ci-dessus, les registres sont mis à jour une fois l'instruction en sortie de l'étage « write back ». Les données des registres sont, elles, récupérées dans l'étage de « decode ».

On constate que les instructions 1 et 2 sont complètement indépendantes l'une de l'autre. Par contre, l'instruction 3 utilise elle en paramètre les registres R3 et R6, modifiés par les deux instructions précédentes : on appelle cela une dépendance de données. C'est un cas particulier qui peut s'avérer problématique quand il n'est pas correctement géré. En effet, les instructions s'enchaînant, au moment où la troisième instruction sera dans l'étage de « decode » et aura besoin de R3 et R6, les instructions 1 et 2 n'auront pas encore terminé d'être traitées. Du coup, leurs résultats ne seront pas encore disponibles et les valeurs reçues par le « decode » ne seront pas celles « à jour ». Deux solutions sont alors envisageables :

- La création d'une « bulle » lors de la détection d'une dépendance de données. Concrètement, si on détecte dans l'étage de « decode » qu'une instruction I dépend du résultat d'une instruction I-1, I-2 ou I-3, alors on bloque les étages « fetch » et « decode » en attendant que le résultat soit accessible (dans 1, 2 ou 3 cycles). Bien qu'efficace fonctionnellement, cette solution engendre une baisse des performances : le pipeline ne sera pas complètement rempli durant au moins un cycle.

- La récupération anticipée des données lors de la récupération des données. Les calculs étant effectués par l'ALU, la donnée finale est accessible dès la sortie de l'unité de calcul. Puis elle va se déplacer jusqu'à être rangée dans les registres accessibles par le « decode ». La mise en place de ce système de récupération permet simplement au « decode » de récupérer la donnée la plus « à jour » directement dans l'étage concerné (si dépendance avec l'instruction I-1, alors en sortie de l'ALU, si I-2, alors en sortie d'« execute » ou alors si I-3, en sortie de « memory access »). Une telle solution permet de ne pas avoir de baisse de performances, en contrepartie de l'ajout d'un peu de matériel (mémorisation des registres pas « à jour », multiplexeur entre les différentes valeurs de données possibles etc …).

Nous avons ici choisi d'implémenter la deuxième solution. Voici ci-dessous l'organisation d'un tel pipeline 5 étages :

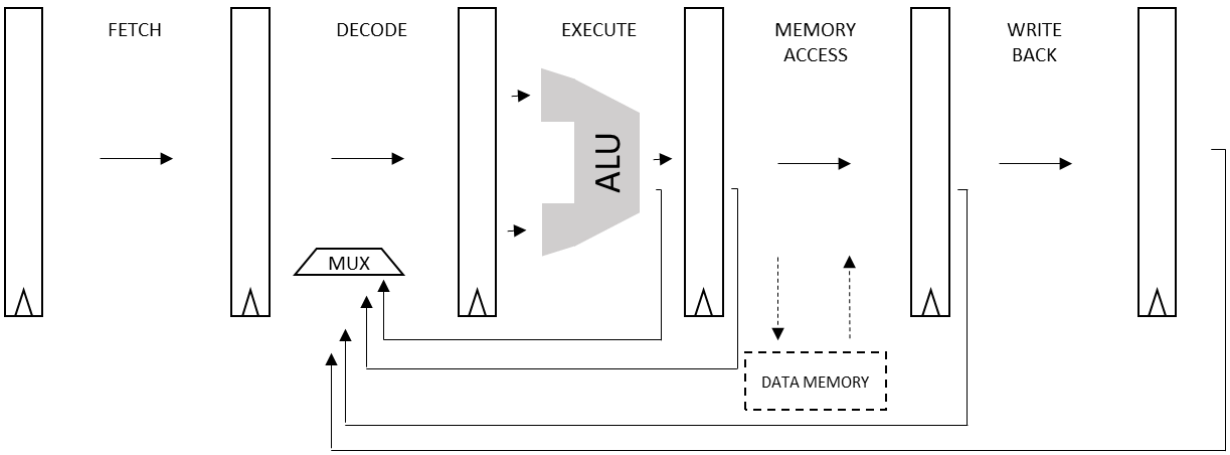


Figure 27: Organisation d'un pipeline 5 étages avec récupération anticipée des données

La différence majeure réside dans la possibilité pour l'étage de « decode » de sélectionner entre plusieurs valeurs d'un même registre. Avec un tel système, la séquence de pseudo-instructions vue précédemment ne génèrera aucun problème et sera bien traité en 5 + 1 + 1 = 7 cycles.

Il existe cependant une autre situation différente de dépendance de données : celle dans le cas de l'exécution d'un chargement mémoire (load). En effet, une telle instruction place dans un registre une donnée située en mémoire. L'accès mémoire s'effectuant dans le 4ème étage, la donnée ainsi récupérée n'est pas accessible par le système de récupération anticipée des données pour les instructions N+1 et N+2. En effet, l'accès mémoire ne sera toujours pas

effectué quand elles passeront dans l'étage « decode » et donc leurs valeurs ne seront pas à jour. Deux solutions semblables à celles vues précédemment nous semblaient alors envisageables :

- La création d'une « bulle » lors de la détection d'une dépendance de données suite à un load. Le « decode » bloque une partie du pipeline en attendant que la donnée lui soit accessible (2 cycles max).

- La récupération anticipée de la donnée depuis l'étage d' « execute ». En détectant une dépendance, on mémoriserait au niveau du « decode » que l'étage suivant doive prendre la donnée en provenance de l'étage « memory access ». Si cette solution nous semblait totalement réalisable, elle supposait la création d'un chemin de données potentiellement très long :

$$T_{chemin} = T_{Logique\ Accès\ Mémoire} + T_{Accès\ Mémoire} + T_{Logique\ Execute} + T_{ALU}$$

En supposant que le chemin critique (dont la fréquence max est déduite) se trouve sur l'accès mémoire (supposition faite en début de projet), cette solution aurait un impact négatif sur la fréquence max du circuit, et donc sur les performances globales du circuit. De ce fait, nous avons choisi d'instaurer la première solution, avec l'hypothèse que des dépendances de données suite à un chargement mémoire ne seront pas effectuées trop régulièrement. Il aurait été intéressant, dans le cas d'une étude complète avec des mémoires définies auparavant, de mesurer réellement les différents timings (notamment celui d'accès mémoire) pour connaître l'impact réel de cette solution.

A présent, ces deux types de dépendance de données sont gérés par notre pipeline, en essayant à chaque fois de mettre en place le meilleur compromis au niveau des performances.

8. Contrôleurs de cache

Un processeur, par nature, a nécessairement besoin d'accéder à des ensembles de mémoires, que cela soit pour récupérer des instructions ou pour lire/ écrire des données. Avec l'organisation hiérarchique des mémoires, le processeur se retrouve plupart du temps connecté à des caches, dans le but de réduire au maximum les temps d'accès.

D'un point de vue organisation, ces différents caches constituent des éléments extérieurs au processeur. Il est donc nécessaire d'anticiper la manière dont le processeur communiquera avec eux. Pour une certaine simplicité d'utilisation, nous avons donc décidé de mettre en place un petit bloc nommé « cache controller », dont le rôle sera de gérer la communication avec un cache donné (un bloc par cache de données/ d'instructions). Un tel bloc, purement combinatoire, se trouve donc à l'interface entre le processeur et les différents caches. L'intérêt est de faciliter la mise en place des connexions entre caches et processeur : en cas de modification du bus d'accès à un cache, il est alors seulement nécessaire de modifier le fichier cache_controller.vhd et non pas les deux étages « fetch » et « memory access ».

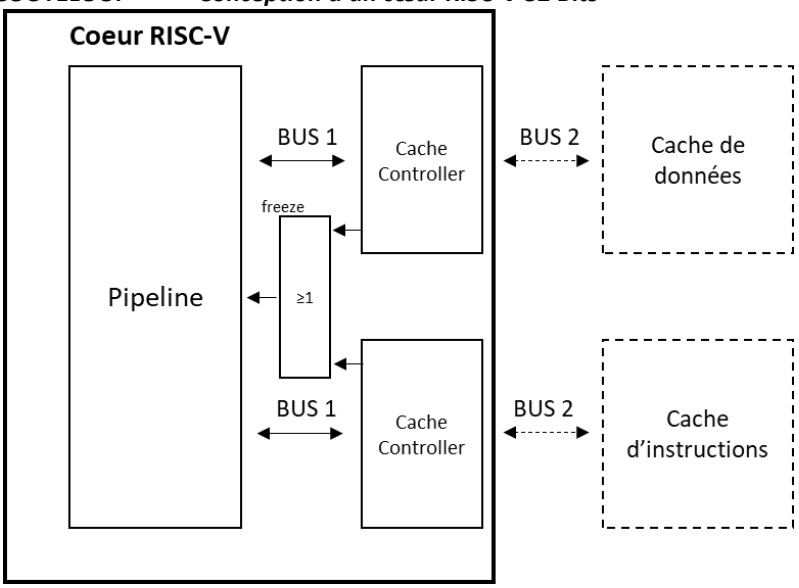


Figure 28: Mise en place des contrôleurs de cache

Dans l'exemple ci-dessus, les deux blocs « cache controller » sont des instances d'un même contrôleur de cache. Ainsi, dans le cas où le BUS 2 changerait, il suffirait simplement d'adapter le contrôleur de cache une seule et unique fois sans toucher au BUS 1 interne.

De plus, avec une telle organisation, il est plus facile au niveau du top du Cœur RISC-V de gérer les « miss » de cache. Pour rappel, les « miss » de cache se déclenchent quand un cache ne dispose pas de la donnée demandée. Il faut alors que celui-ci aille chercher dans un niveau de hiérarchie mémoire supérieur la donnée qui lui manque, ce qui peut demander plusieurs cycles d'attente. Dans notre système, chaque contrôleur cache doit pouvoir détecter un « miss » sur son cache associé. Ainsi, au niveau top, dès qu'un contrôleur a détecté un « miss » il est facile de bloquer l'ensemble du pipeline en attendant que l'ensemble des caches soient prêts. On appellera cette opération de blocage un « freeze ».

C. Validations

1. Test Unitaire : VUnit

VUnit est un framework de tests unitaires open source pour VHDL / SystemVerilog. Ecrit en langage Python, il utilise d'autres outils de simulation, comme ModelSim ici, pour compiler et effectuer les différents tests.

L'une des forces de VUnit est sa facilité d'installation et d'utilisation. Concrètement pour notre projet, sa mise en place a été rapide : nous avons récupéré les différentes librairies Python sur le projet Github associé et puis avons procédé à l'installation en suivant le manuel d'installation. Un script run.py est alors à adapter pour définir différents paramètres comme :

- L'emplacement des fichiers descriptifs du design,
- L'emplacement des différents testbenchs,
- Le nom de la librairie de compilation du design,
- Le nom de la librairie de compilation des benchs.

Une librairie supplémentaire est également à inclure lors de l'écriture des benchs en VHDL, ainsi que 2 brèves commandes dans l'architecture pour indiquer le début et la fin du bench à l'outil. Pour réellement profiter de cet outil, il est cependant nécessaire de réfléchir à des benchs robustes et exhaustifs où les vérifications s'effectuent

automatiquement à l'aide d'assert. Un bench sans assert signifie que l'outil ne détectera aucun problème et donc il sera inutile …

L'intérêt d'un environnement de test unitaire est de pouvoir ré-effectuer facilement l'ensemble des tests mis en place. Dans notre cas, nous avons mis en place côté conception une méthode incrémentale :

- Nous avons dans un premier temps développés les différents blocs en prenant en compte seulement les instructions correspondant à des opérations arithmétiques et logiques.
- Nous avons ensuite ajouté aux différents étages la gestion des instructions de chargement/ stockage en mémoire.
- Nous avons ensuite intégré les instructions de saut inconditionnel.
- Etc …

Une telle méthode de conception implique de modifier plusieurs fois un même étage et donc peut engendrer un risque d'erreur plus important. Cependant, en utilisant un environnement de test unitaire complet, on s'assure que l'ajout d'une nouvelle fonctionnalité ne vient pas perturber le fonctionnement d'une autre. En cas d'ajout d'un nouveau bloc d'un point de vue design, il est conseillé de le vérifier indépendamment de ceux déjà inclus dans VUnit en mettant en place un bench complet. Une fois celui-ci complètement approuvé, il est alors possible de l'intégrer à l'ensemble du panel de benchs gérés par VUnit.

Nous n'avons au cours de ce projet utilisé qu'une infime partie des fonctionnalités qu'offre VUnit : nous nous sommes contentés de paramétrer le script de manière très basique pour qu'il prenne en compte l'ensemble de nos fichiers RTL puis nous exécutions le tout. Le rapport généré en sortie nous indiquait rapidement si des fautes avaient été détectées ou non. Parmi les options offertes en ligne de commande, seule celle nous permettant d'afficher le chronogramme d'un test donné (en plus de la commande de compilation) nous a été nécessaire.

Nous évoquerons plus en détails l'uilisation de VUnit tout au long de cette partie consacrée à la vérification fonctionnelle de notre design.

2. Test de chaque bloc séparément

Dans cette partie, nous allons voir en détails les tests réalisés sur l'ensemble des étages du pipeline. Comme la méthode est la même pour l'ensemble des benchs, on s'intéressera principalement au fichier execute_bench.vhd.

Tout d'abord, comme indiqué dans le paragraphe précédent, pour tester les étages du pipeline, nous avons utilisé le test unitaire VUnit. Pour intégrer VUnit dans un bench, il est notamment nécessaire d'inclure la librairie dédiée :

```
library vunit_lib;
context vunit_lib.vunit_context;
```

Figure 29 : Appel librairie VUnit

Tous les benchs du pipeline utilise le test unitaire VUnit. On retrouve donc l'appel de la librairie Vunit dans tous les fichiers de validation. Il est maintenant nécessaire de mettre en place des vérifications automatiques à l'aide d'assert.

Les asserts sont des instructions qui vérifient qu'une condition spécifiée est vraie et affiche un message si ce n'est pas le cas. L'assert a trois champs :

- Assert : Egalité que l'on cherche à vérifier.
- Report : Message indiqué lorsque la l'égalité spécifiée précédemment est fausse.
- Severity : En cas d'erreur, permet de donner différents niveaux d'importance à celle-ci :

- NOTE peut être utilisé pour transmettre des messages d'information à partir de la simulation
- WARNING peut être utilisé dans une situation inhabituelle dans laquelle la simulation peut être poursuivie, mais les résultats peuvent être imprévisibles.
- ERROR peut être utilisé lorsque la violation d'assertion rend impossible la poursuite de la simulation.
- FAILURE peut être utilisé lorsque la violation d'assertion est une erreur fatale et que la simulation doit être arrêtée immédiatement.

On peut généralement lire un assert de la manière suivante :

« Si tel signal n'est pas égal à telle valeur, alors il faut afficher le message suivant indiquant qu'un résultat inattendu plus ou moins grave est apparu. »

En jouant sur le temps au cours d'un bench, il est ainsi possible de vérifier qu'un même signal prendra les valeurs attendues au cours du temps.

Nous allons à présent voir comment nous avons utilisés ces assert pour mettre en place des benchs les plus robustes possibles.

```
wait for HALF_PERIOD;
dcde_inst <= "0000000" & "10101" & "01111" & c_FUNC3_ADD & "00000" & c_OPCODE32_OP;
dcde_rs1 <= "00000000000000000000000000010010";
dcde_rs2 <= "00000000000000000000000000011100";
wait for HALF_PERIOD;
assert exec_validity = '1' report "Problem signal validaty output  " severity error;
assert exec_jump = '0' report "Problem signal jump output  " severity error;
assert exec_branch = '0' report "Problem signal branch output  " severity error;
assert exec_rd = "00000000000000000000000000101110" report "Problem signal rd output  " severity error;

wait for HALF_PERIOD;
dcde_inst <= "0100000" & "10101" & "01111" & c_FUNC3_SUB & "00001" & c_OPCODE32_OP;
dcde_rs1 <= "00000000000000000000000000011110";
dcde_rs2 <= "00000000000000000000000000010000";
wait for HALF_PERIOD;
assert exec_validity = '1' report "Problem signal validaty output  " severity error;
assert exec_jump = '0' report "Problem signal jump output  " severity error;
assert exec_branch = '0' report "Problem signal branch output  " severity error;
assert exec_rd = "00000000000000000000000000001110" report "Problem signal rd output  " severity error;
```

Figure 30 : Création du bench dédié à un étage : execute_bench.vhd

Comme on peut le voir sur la figure ci-dessus, la première étape des benchs est d'affecter les entrées de l'étage. Dans le cas présent, on vérifie le fonctionnement de l'execute. Pour cela, on affecte l'entrée i_inst avec une instruction connue et respectant le format indiqué dans la spécification RISC-V. On affecte aussi les registres rs1 et rs2 pour effectuer l'opération indiquée par l'instruction.

Après une demi-période (un front montant d'horloge ayant eu lieu), on vérifie la valeur des signaux de sortie. Ici, on se penche surtout sur le signal exec_rd qui correspond au résultat entre les deux registres rs1 et rs2. Si les opérations ne s'exécutent pas correctement, une erreur nous est signalée lors de la simulation du système.

En vérifiant de la même manière un maximum de cas de fonctionnement, on peut ainsi définitivement le bon fonctionnement d'un bloc. En dupliquant cette méthode pour l'ensemble des blocs, on obtient alors une vérification unitaire complète des différents blocs du système. Quand les blocs conçus se multiplient, alors les benchs associés également. C'est aussi dans ce cadre-là qu'un environnement de test comme VUnit devient pertinent, pour vérifier le bon fonctionnement de la totalité des blocs.

3. Test du système global

Nous avons précédemment vérifié le bon fonctionnement de chaque bloc de manière individuel. L'objectif va être à présent de valider le fonctionnement du système global.

Ce système global étant le pipeline, il existe une manière simple de vérifier son bon fonctionnement : simuler l'arrivée d'instruction en entrée de l'étage de fetch, et vérifier les résultats obtenus au niveau des accès à la mémoire de données. Pour faciliter la simulation de lecture d'instructions prédéfinies, nous avons mis en place un système de bench où les valeurs à envoyer en entrer du cœur à tester sont lues dans un fichier ligne par ligne. L'avantage d'une telle solution et qu'il est facile d'ajouter de nouvelles instructions à tester puisqu'il suffit de les coder en binaire dans le fichier texte.

```
114                 while not endfile(f_inst) loop
115                     readline(f_inst,v_inst_line);
116                     read(v_inst_line, v_inst);
117                     s_imem_idata <= v_inst;
118 
119                     wait for PERIOD;
120                     if(not endfile(f_iaddress)) then
121                         readline(f_iaddress,v_iaddress_line);
122                         read(v_iaddress_line, v_iaddress);
123                         assert s_imem_addr = v_iaddress report "Problem in instruction address !" severity error;
124                     end if;
125 
126                     if(not endfile(f_ddataout)) then
127                         readline(f_ddataout,v_ddataout_line);
128                         read(v_ddataout_line, v_ddataout);
129                         assert s_dmem_odata = v_ddataout report "Problem in data out !" severity error;
130                     end if;
131                 end loop;
```

Figure 31: Bench pour la vérification du pipeline

On voit ci-dessous la boucle du bench nous permettant de parcourir l'ensemble du fichier d'instructions : tant que nous ne sommes pas arrivés à la fin de celui-ci, une nouvelle ligne du fichier est lue à chaque cycle puis transformée pour arriver en entrée du pipeline. De la même façon, un système de test automatique du résultat attendu a été mis en place : l'adresse attendue est lue dans le fichier à chaque cycle, puis est comparée à celle réellement présente dans le pipeline. En cas de différence entre ces deux valeurs, un message indiquant une erreur est affiché. De cette manière, on parvient ici à vérifier deux signaux : l'évolution de l'adresse d'instruction et l'évolution de la donnée de sortie.

```
 1 10101010101000100000101110010011
 2 00000000001100110000000010010011
 3 00111000001101110100001010010011
 4 00000000000011011100000100110011
 5 00000000010110111110001000110011
 6 01000000000100101000010100110011
 7 01110000000000000000110000010111
 8 11101001010101100000100000110111
 9 00000000100010000001101010010011
10 01000000010010000101011000010011
11 00000001000001100101011000010011
12 01000000000001010011111110010011
13 10000000000001010010111100010011
14 01011111110111111101010010000011
15 01011010011101010000100100000011
16 01011111110111111101010011000011
17 01011111110111111110110100000011
18 01111111011111111010010100100011
19 11110000001100111111101111101111
20 00000000000000000000000000010011
21 00000000000000000000000000010011
22 00001000000100010111101001100011
```

Figure 32: Exemple de fichier d'instructions fourni au bench

Voici ci-dessus un fichier avec des instructions qui seront lues par le bench. Elles ont été codées manuellement, tout en respectant l'ISA RV32I. On y retrouve à l'intérieur tous les types d'instructions : des opérations entre registres/ avec immédiat, des sauts, des branchements … Les fichiers utilisés pour la vérification des valeurs en sortie sont conçus de manière similaire :

```
1  00000000000000000000000000000100
2  00000000000000000000000000001000
3  00000000000000000000000000001100
4  00000000000000000000000000010000
5  00000000000000000000000000010100
6  00000000000000000000000000011000
7  00000000000000000000000000011100
8  00000000000000000000000000100000
9  00000000000000000000000000100100
10 00000000000000000000000000101000
11 00000000000000000000000000101100
12 00000000000000000000000000110000
13 00000000000000000000000000110100
14 00000000000000000000000000111000
15 00000000000000000000000000111100
16 00000000000000000000000001000000
17 00000000000000000000000001000100
18 00000000000000000000000001001000
19 00000000000000000000000001001100
20 00000000000000000000000001010000
21 11111111111100111111111101001010
22 11111111111100111111111101001110
23 11111111111100111111111101010010
24 11111111111100111111111111011110
```

Figure 33: Valeurs attendues pour l'adresse d'instruction

On retrouve ci-dessus l'évolution classique de l'adresse d'instruction, qui s'incrémente à chaque cycle de 4 (les instructions étant sur 32 bits, on se déplace de 4 en 4 octets). On remarque sur la fin une incrémentation différente de l'adresse : cela correspond à la vérification du bon fonctionnement d'une instruction de saut.

Avec un tel système de vérification automatisé à l'intérieur du bench, la plupart des erreurs de fonctionnement sont détectées. Il peut cependant s'avérer nécessaire de regarder en détail l'évolution de certains signaux :

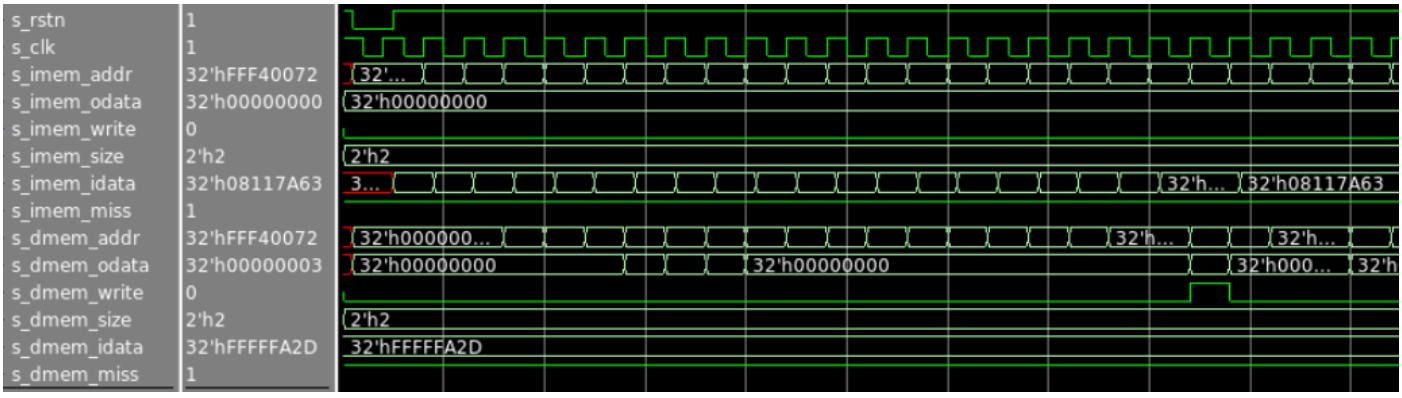


Figure 34: Chronogramme pour la vérification du fonctionnement du pipeline

On visualise ci-dessus l'évolution des signaux externes du pipeline. Cette méthode visuelle n'est cependant viable que pour analyser des cas bien précis : les instructions s'enchaînant à chaque cycle, il serait bien trop facile d'oublier de vérifier certains points.

Nous avons à présent vu la méthode de vérification du pipeline entier. Cette méthode se basant sur un simple bench, celui-ci vient donc simplement s'ajouter à la batterie de tests unitaires mise en place avec VUnit. Plus tard, si nous souhaitons ajouter de nouvelles fonctionnalités au système, nous n'aurons donc qu'à relancer l'ensemble de la batterie de tests pour être certain que nous n'avons défaits des éléments qui fonctionnait auparavant.

```
==== Summary ==========================================================
pass LIB_PIPELINE_BENCH.tb_execute.all        (2.5 seconds)
pass LIB_PIPELINE_BENCH.tb_decode.all         (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_pipeline.all       (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_writeback.all      (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_fetch.all          (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_reg_integer.all    (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_memory_access.all  (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_alu.all            (1.6 seconds)
pass LIB_PIPELINE_BENCH.tb_finaldemo.all      (2.0 seconds)
pass LIB_PIPELINE_BENCH.tb_demo.all           (1.6 seconds)
==============================================================================
pass 10 of 10
==============================================================================
Total time was 17.5 seconds
Elapsed time was 17.8 seconds
==============================================================================
All passed!
```

Figure 35: Ensemble des benchs mis en place avec VUnit

D. Synthèse

Maintenant que la description RTL a été validée en simulation, nous allons chercher à vérifier si notre design est synthétisable.

1. Spyglass

Avant la synthèse, il est intéressant, mais non obligatoire, de réaliser une vérification poussée du code à l'aide d'un outil spécialisé : Spyglass. Si certaines vérifications de syntaxes ont été analysées par le biais des compilations ModelSim, Spyglass va effectuer une analyse plus poussée, en cherchant notamment à interpréter le code d'un point de vue matériel. L'un des objectifs, en utilisant cet outil, est d'essayer d'anticiper la génération de matériel ne correspondant pas à ce que l'on voulait écrire auparavant dans le code. L'exemple typique de génération de matériel non-maîtrisée est la création de cellules latchs involontaires. Ce genre de situation arrive par exemple lorsque dans un bout de code de type « case … when … » ou « if … then … elsif … », tous les cas ne sont pas pris en compte. Ceci peut engendrer une non-modification du signal et donc, par déduction, une mémorisation de l'ancienne valeur (d'où l'apparition de latchs pour la mémorisation).

Après avoir indiqué à l'outil les différents fichiers VHDL à vérifier, nous pouvons alors lancer l'analyse.

Note : Très peu de licences étant disponibles, seules 2 vérifications différentes seront réalisées, les autres générant par défaut une erreur fatale.

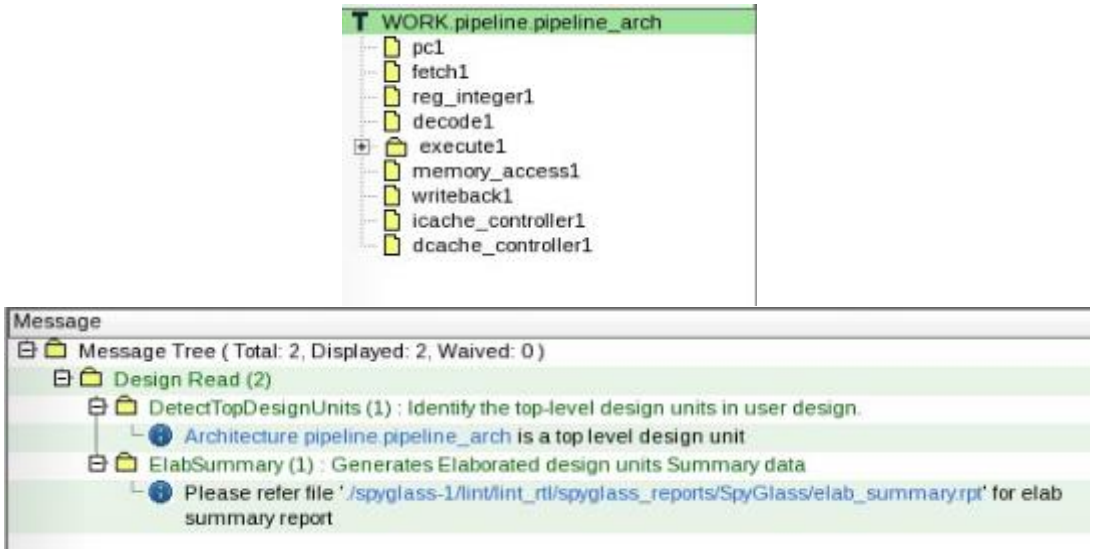


Figure 36: Détection du design par Spyglass

Figure 37: Résultat des différentes vérifications via Spyglass

Les deux premières figures nous montrent que Spyglass a bien retrouvé et interprété le design : la détection des différents blocs et l'élaboration du design se sont déroulées sans erreur.

La troisième figure nous montre le résultat des différentes vérifications effectuées (F : Fatal Error, E : Error, W : Warning, I : Information). Sur les deux premiers tests, aucun problème n'a été détecté (que des messages

d'informations). Les deux suivants se sont conclus par une erreur fatale comme attendu, les licences correspondantes n'étant pas disponibles.

Les vérifications sous Spyglass sont à présent terminées. Aucun problème n'ayant été détecté, nous allons pouvoir passer à l'étape de synthèse en elle-même.

2. Synthèse

Nous allons à présent détailler l'étape de synthèse. Elle consiste à générer un ensemble de portes logiques réalisant la même fonctionnalité que celle décrite dans le code RTL. La liste de l'ensemble des portes logiques et des signaux ainsi générés s'appelle la netlist. A partir de ce fichier, il est ensuite possible d'avoir de premières estimations sur les performances globales du circuit :

- On peut déterminer le chemin critique du design. En fonction du temps de traversé de ce chemin, il sera possible de déterminer la fréquence maximale de fonctionnement de notre circuit. Pour améliorer cette dernière, il faudra donc à l'inverse chercher à diminuer le chemin critique.
- On peut également faire une première estimation de la consommation du circuit,
- Et enfin, on peut également connaître la surface du circuit, en s'appuyant sur la surface individuelle de chaque cellule notamment.

Dans le cadre de ce projet, ces différentes caractéristiques n'étaient pas spécifiées. De plus, toutes ces caractéristiques varient grandement en fonction de la technologie utilisée mais aussi des contraintes imposées à l'outil de synthèse. Ici, nous sommes partis sur une technologie 350nm et une fréquence spécifiée à 100MHz. L'objectif de cette synthèse est uniquement de vérifier que notre design est correctement synthétisable, et que donc la netlist générée est fonctionnelle. Concernant l'outil de synthèse, nous avons utilisé DesignCompiler de Synopsys, un outil que nous avions en partie utilisé précédemment durant la formation.

Voici ci-dessous l'un des scripts utilisés pour automatiser la génération d'une netlist. Celle qui est générée par ce script est dite « à plat » : l'outil va supprimer la hiérarchie entre les différents blocs en entrée (avec par exemple l'ALU contenue dans l'étage « execute », lui-même contenu dans le top) et tout mettre au même niveau dans un seul et unique bloc. Ceci a pour effet de lui permettre d'optimiser certains chemins logiques se trouvant découpés sur plusieurs blocs.

Pour avoir des éléments de comparaison à notre disposition, une netlist hiérarchique est également générée par un autre script semblable à celui-ci :

```
source ../scripts/define_variables.tcl

define_design_lib $LIBRARY_NAME -path ../libs_synth/$LIBRARY_NAME

analyze -library $LIBRARY_NAME -format vhdl $VHDL_FILES

elaborate ${DESIGN_NAME} -architecture $ARCHI_TOP_NAME -library $LIBRARY_NAME

source ../scripts/top_num.sdc

set_operating_conditions -library c35_CORELIB_TYP WORST

compile -exact_map -area_effort high -ungroup_all

write_sdf ../results/${DESIGN_NAME}_flat.sdf
write -format ddc -hierarchy -output ../results/${DESIGN_NAME}_flat.ddc

#Genere la netlist au format VHDL
write -hierarchy -format vhdl -output ../results/${DESIGN_NAME}_flat.vhd
write -hierarchy -format verilog -output ../results/${DESIGN_NAME}_flat.v

report_timing -nworst 10    > ../log/${DESIGN_NAME}_flat.timing.log
report_area                 > ../log/${DESIGN_NAME}_flat.area.log
report_power                > ../log/${DESIGN_NAME}_flat.power.log
report_cell                 > ../log/${DESIGN_NAME}_flat.cell.log
report_clock                > ../log/${DESIGN_NAME}_flat.clock.log
report_qor                  > ../log/${DESIGN_NAME}_flat.qor.log

quit
```

Figure 38: Script pour l'automatisation de la génération d'une netlist à plat

Détaillons les principales étapes de cette synthèse :

- Création d'une librairie pour la synthèse,
- Analyse des différents fichiers VHDL du projet,
- Elaboration de l'ensemble du design,
- Lecture des contraintes de timing, et aussi des différents corners de fonctionnement,
- Compilation de l'ensemble du design. On a ici spécifié à l'outil de gros efforts en termes de surface, ainsi que la volonté de générer une netlist « à plat »,
- Ecriture des différents fichiers de résultats, dont des netlist aux formats Verilog et VHDL, ainsi que des fichiers de description des timings et des connexions (nécessaires pour la future simulation post-synthèse),
- Ecriture de différents rapports concernant le circuit (timing, consommation, surface …).

```
Timing Path Group 'HCLK'
-----------------------------------
Levels of Logic:               23.00
Critical Path Length:           9.36
Critical Path Slack:            0.00
Critical Path Clk Period:      10.00
Total Negative Slack:           0.00
No. of Violating Paths:         0.00
Worst Hold Violation:           0.00
Total Hold Violation:           0.00
No. of Hold Violations:         0.00
-----------------------------------
```

```
Timing Path Group 'HCLK'
-----------------------------------
Levels of Logic:               30.00
Critical Path Length:           9.49
Critical Path Slack:            0.00
Critical Path Clk Period:      10.00
Total Negative Slack:           0.00
No. of Violating Paths:         0.00
Worst Hold Violation:           0.00
Total Hold Violation:           0.00
No. of Hold Violations:         0.00
-----------------------------------
```

Figure 39: Comparaison en Timing pour des netlists à plat (gauche) et hiérarchique (droite)

```
Cell Count
-----------------------------------
Hierarchical Cell Count:           0
Hierarchical Port Count:           0
Leaf Cell Count:                7031
Buf/Inv Cell Count:             1623
Buf Cell Count:                  566
Inv Cell Count:                 1057
CT Buf/Inv Cell Count:             0
Combinational Cell Count:       5671
Sequential Cell Count:          1360
Macro Count:                       0
-----------------------------------
```

```
Cell Count
-----------------------------------
Hierarchical Cell Count:          22
Hierarchical Port Count:        2796
Leaf Cell Count:                7913
Buf/Inv Cell Count:             2293
Buf Cell Count:                  856
Inv Cell Count:                 1437
CT Buf/Inv Cell Count:             0
Combinational Cell Count:       6553
Sequential Cell Count:          1360
Macro Count:                       0
-----------------------------------
```

Figure 40: Comparaison des cellules logiques pour des netlists à plat (gauche) et hiérarchique (droite)

```
Area
-----------------------------------
Combinational Area:   423605.001656
Noncombinational Area:
                      468267.784027
Buf/Inv Area:          82810.000381
Total Buffer Area:         35089.60
Total Inverter Area:       47720.40
Macro/Black Box Area:      0.000000
Net Area:             179514.000000
-----------------------------------
Cell Area:            891872.785683
Design Area:         1071386.785683
```

```
Area
-----------------------------------
Combinational Area:   530165.997498
Noncombinational Area:
                      468122.183929
Buf/Inv Area:         117881.400742
Total Buffer Area:         56074.20
Total Inverter Area:       61807.20
Macro/Black Box Area:      0.000000
Net Area:             188343.000000
-----------------------------------
Cell Area:            998288.181427
Design Area:         1186631.181427
```

Figure 41: Comparaison de la surface pour une netlist à plat (gauche) et hiérarchique (droite)

```
Logic Optimization:                58.03
```

```
Logic Optimization:                26.39
```

Figure 42: Quantification des optimisations réalisée pour une netlist à plat (gauche) et hiérarchique (droite)

Globalement, on constate que comme attendu, DesignCompiler a réussi à mieux optimiser la netlist à plat plutôt que la netlist hiérarchique. On notera notamment des gains au niveau du nombre de cellules logiques combinatoires utilisées : 5671 contre 6553 (-13%) et donc également au niveau de la surface globale du circuit : 1.071mm² contre 1.187mm² (-10%). Chose rassurante, le même nombre de buffers a été créé pour les deux netlists : 1360.

Note : Ce nombre de registres aurait pu facilement être calculé dès l'étude architecturale du système. En effet, c'est un des éléments clés qui peut être quasiment défini à l'unité près dès que le choix du pipeline a été établi.

D'un point de vue timing, on notera que l'outil a réussi à effectuer les optimisations nécessaires pour tenir la fréquence de 100 MHz à laquelle il était contraint.

```
Point                                              Incr       Path
-----------------------------------------------------------------
clock HCLK (rise edge)                             0.00       0.00
clock network delay (ideal)                        0.00       0.00
decode1/o_validity_reg/C (DFC3)                    0.00 #     0.00 r
decode1/o_validity_reg/Q (DFC3)                    1.25       1.25 f
decode1/o_validity (decode)                        0.00       1.25 f
execute1/i_validity (execute)                      0.00       1.25 f
execute1/U282/Q (NAND24)                           0.33       1.58 r
execute1/U284/Q (INV6)                             0.14       1.72 f
execute1/U283/Q (NAND28)                           0.45       2.17 r
execute1/U123/Q (INV12)                            0.07       2.24 f
execute1/U262/Q (NAND23)                           0.27       2.51 r
execute1/U288/Q (INV6)                             0.10       2.61 f
execute1/U200/Q (NAND24)                           0.30       2.91 r
execute1/U196/Q (BUF8)                             0.40       3.31 r
execute1/U169/Q (NOR24)                            0.21       3.52 f
execute1/U78/Q (CLKIN6)                            0.14       3.66 r
execute1/U417/Q (OAI212)                           0.31       3.97 f
execute1/alu1/i_amount[4] (alu)                    0.00       3.97 f
execute1/alu1/U709/Q (CLKBU15)                     0.61       4.57 f
execute1/alu1/U303/Q (NAND24)                      0.42       5.00 r
execute1/alu1/U15/Q (BUF12)                        0.44       5.44 r
execute1/alu1/U190/Q (OAI221)                      0.27       5.71 f
execute1/alu1/U440/Q (AOI212)                      0.34       6.05 r
execute1/alu1/U300/Q (IMUX23)                      0.33       6.38 f
execute1/alu1/U337/Q (CLKIN6)                      0.24       6.62 r
execute1/alu1/U565/Q (AOI222)                      0.14       6.76 f
execute1/alu1/U960/Q (NAND22)                      0.45       7.21 r
execute1/alu1/U85/Q (NAND22)                       0.13       7.34 f
execute1/alu1/U500/Q (NAND24)                      0.30       7.64 r
execute1/alu1/U504/Q (NAND24)                      0.08       7.72 f
execute1/alu1/U718/Q (CLKIN6)                      0.17       7.89 r
execute1/alu1/U727/Q (NOR24)                       0.19       8.08 f
execute1/alu1/U639/Q (NAND24)                      0.32       8.40 r
execute1/alu1/o_result[8] (alu)                    0.00       8.40 r
execute1/U278/Q (NAND26)                           0.08       8.49 f
execute1/U279/Q (NAND28)                           0.25       8.74 r
execute1/o_rd_alu[8] (execute)                     0.00       8.74 r
decode1/i_rd_alu[8] (decode)                       0.00       8.74 r
decode1/U110/Q (AOI222)                            0.24       8.98 f
decode1/U218/Q (OAI2112)                           0.50       9.49 r
decode1/o_rs1_reg[8]/D (DFC3)                      0.00       9.49 r
data arrival time                                             9.49

clock HCLK (rise edge)                            10.00      10.00
clock network delay (ideal)                        0.00      10.00
clock uncertainty                                 -0.50       9.50
decode1/o_rs1_reg[8]/C (DFC3)                      0.00       9.50 r
library setup time                                -0.01       9.49
data required time                                            9.49
-----------------------------------------------------------------
data required time                                            9.49
data arrival time                                            -9.49
-----------------------------------------------------------------
slack (MET)                                                   0.00
```

Figure 43: Chemin Critique en synthèse

Les chemins critiques, limitants théoriquement notre fréquence, ont été identifiés par DesignCompiler d'ans l'étage « execute ». C'est un résultat logique, quand on sait que c'est dans cet étage que la plupart des calculs (et donc les portes logiques nécessaires) sont implémentés. On retrouve par exemple ci-dessus le chemin démarrant en sortie du registre o_validity (sortie de l'étage « decode ») et se terminant en entrée du registre o_rs1 (sortie de l'étage

« decode »). C'est donc un chemin lié à la gestion de la dépendance de donnée, qui traverse l'étage « execute » et l'ALU avant de revenir dans l'étage de « decode ».

Pour faire le lien avec ce que nous avons vu précédemment entre synthèse hiérarchique et synthèse à plat, on peut constater que le chemin ci-dessus est un exemple typique de chemin optimisable avec une synthèse à plat. En effet, en traversant 3 blocs hiérarchiques différents, l'outil va être capable de réaliser bien plus d'optimisation avec une synthèse à plat. On ne retrouve d'ailleurs ce chemin que dans la liste des chemins critiques pour la synthèse hiérarchique.

Enfin, une dernière donnée intéressante liée aux deux types de synthèse est la quantification des optimisations réalisées par l'outil sur chaque version du design : ainsi, comme on pouvait s'y attendre, l'outil a réalisé deux fois plus d'optimisations sur la netlist à plat que sur la netlist hiérarchique.

Nous avons à présent des netlists qui semblent avoir été correctement générées. La dernière étape va consister à simuler la fonctionnalité de ces netlists générées.

3. Simulation Post-Synthèse

Comme nous avions pu le faire lors de la conception RTL du design, nous allons effectuer des simulations du cœur RISC-V en utilisant cette fois la netlist VHDL générée précédemment. L'outil utilisé pour cela sera encore une fois ModelSim, le bench sera pipeline_bench.vhd (le même que celui présenté auparavant dans la partie vérification RTL). En plus de cela, il faudra bien spécifier à ModelSim les librairies à utiliser pour compiler et simuler la netlist, ainsi que le fichier des timings correspondants.

Nous ne reviendrons pas ici en détail sur le bench utilisé. Cependant, plusieurs assertions ayant été utilisées pour indiquer des erreurs, il va être rapide de savoir si la simulation correspond au résultat attendu ou non.

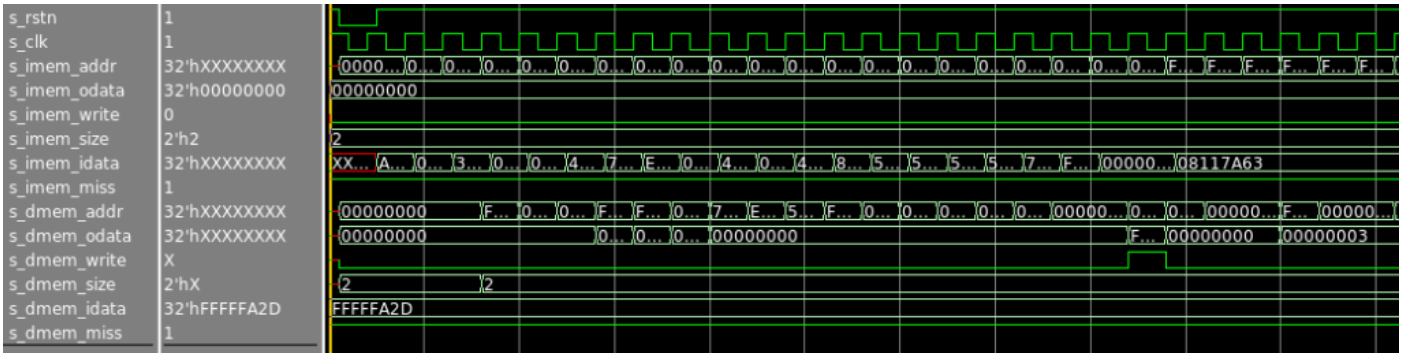


Figure 44: Chronogramme de la simulation post-synthèse

Finalement, aucune erreur n'a été indiquée par le bench. En visualisant plus en détail le chronogramme, on retrouve la même évolution des signaux que pour la simulation RTL. La fonctionnalité semble donc avoir été conservé et l'on peut conclure que notre netlist décrit bien le fonctionnement voulu.

Finalement, on peut conclure que la version actuelle de notre design est synthétisable : aucune erreur n'est générée durant cette étape du flot de conception et le résultat obtenu en sortie est fonctionnellement correct.

III. Utilisation et démonstration du fonctionnement du Processeur

Nous avons vu précédemment les différentes étapes de conception et de vérification que nous avons mises en place dans le but d'obtenir un cœur RISC-V 32 bits fonctionnel. Nous allons à présent voir comment un potentiel utilisateur pourrait récupérer nos codes pour une utilisation réelle, ainsi qu'illustrer cela avec une démonstration réelle.

A. Manuel d'utilisation

Il est possible de simplifier très simplement du point de vue

Du point de vue d'un utilisateur extérieur, ce processeur RISC-V fonctionne, comme un processeur classique, très simplement :

- A chaque cycle, le cœur va lire en mémoire d'instructions une nouvelle instruction, qu'il mettra au total 5 cycles à traiter.
- Ces instructions consistent, pour la plupart, en des opérations sur des données non-comprises dans le code de l'instruction. Il faut donc aller les chercher ailleurs : en mémoire de données.

En conclusion, en plus du cœur lui-même, des mémoires d'instructions et de données sont nécessaires pour obtenir un système fonctionnel. Ce sont des éléments que l'utilisateur devra connecter au cœur RISC-V.

Note : Ce cœur RISC-V utilise des adresses octets et des données codées sur 32 bits. Cela donc doit être également le cas des mémoires utilisées. Si le protocole d'échange entre les caches et le cœur doivent être changés, c'est alors sur le fonctionnement du bloc ***cache_controller*** *qu'il faudra intervenir. On notera également, que s'il existe des signaux d'écriture et taille en sortie du cœur pour accéder à la mémoire instruction, ceci est uniquement dans le but de n'avoir qu'un unique protocole d'accès aux caches. Ainsi, du point de vue du processeur, la mémoire instruction ne sera accédée en lecture 32 bits uniquement.*

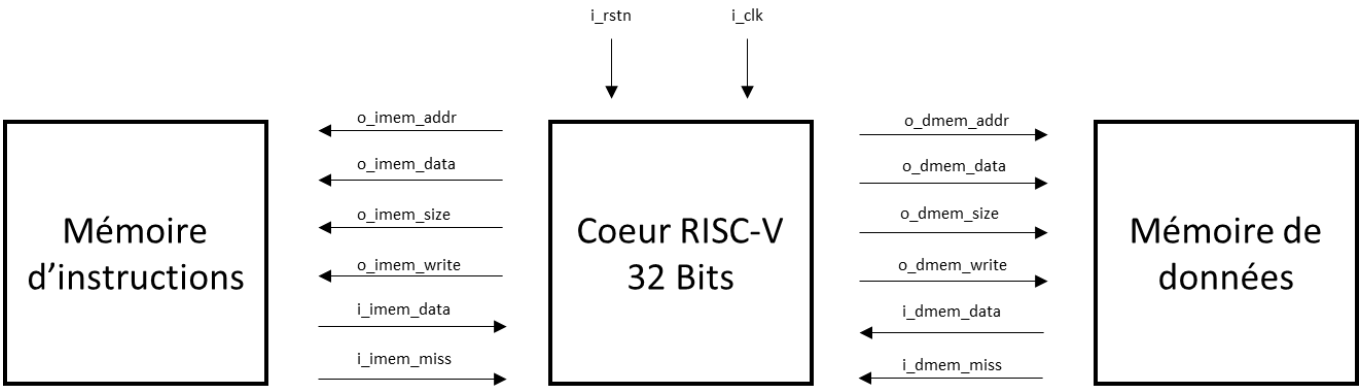


Figure 45: Intégration du Cœur dans un système utilisateur

En plus des mémoires, il est évidemment nécessaire de connecter un signal de reset négatif et une horloge en entrée du cœur. A la suite de cela, nous aurons donc un système complet du point de vue matériel. Côté logiciel, il faudra charger ces différentes mémoires avant de lancer l'exécution.

Note : Nous ne verrons pas ici la méthode de remplissage de ces mémoires, mais plutôt le contenu en lui-même.

Côté mémoire de données, il faut donc qu'elle contienne les différentes valeurs nécessaires aux opérations du processeur. Deux situations sont possibles ici :

- Aucune valeur n'est initialisée en mémoire car le processeur n'aura besoin d'en lire aucune pour le bon déroulement du programme.

- Certaines valeurs ont été initialisées en vue d'une future lecture du processeur.

L'objectif est ici d'éviter que le processeur aille lire une donnée non initialisée à une adresse spécifiée dans une instruction. Sinon, on peut alors s'attendre à des résultats incohérents de calculs pour l'utilisateur.

Note : Pour ce manuel d'utilisation, on a supposé que la mémoire de données occupée l'ensemble de la plage d'adresse de données du système. Dans un environnement plus complexe, on pourrait supposer que le cœur ait à lire et à écrire des données à des adresses correspondant à des périphériques, des I/Os etc …

Côté mémoire d'instruction, cette version du cœur ne sachant exécuter que les instructions RISC-V RV32I, il est nécessaire que celles stockées en mémoire correspondent à ce format-là. Pour rappel, en annexes, l'ensemble des instructions gérées ont été listées, en plus de la convention d'utilisation des différents registres par les instructions.

Après reset (signal i_rstn à '0'), le processeur recommence sa lecture des instructions à partir de l'adresse 0x00000000 de la mémoire d'instructions. Il est donc nécessaire que la première instruction à exécuter se trouve à cette adresse, afin d'éventuellement effectuer un saut vers une autre adresse.

B. Cas Concret d'utilisation

Nous avons vu précédemment quels éléments devait mettre en place l'utilisateur pour obtenir un système complet fonctionnel. Nous allons à présent voire la mise en place d'un exemple, avec simulation sous ModelSim.

Nous allons dans un premier temps nous intéresser à la conception d'un simple programme écrit en assembleur RISC-V. En voici le détail ci-dessous :

```
1  -- CALCUL DU PGCD DE A ET B
2  -- A = 17 et B = 3
3
4  000000000000 00000 000 00010 0010011    -- 00   addi sp, zero, 0
5  000000111100 00000 000 00011 0010011    -- 04   addi gp, zero, 60
6  000000010001 00000 000 01010 0010011    -- 08   addi a1, zero, 17
7  000000000011 00000 000 01011 0010011    -- 0C   addi a2, zero, 3
8  0000000 01010 00010 010 00000 0100011   -- 10   sw a1, sp(0)
9  0000000 01011 00010 010 00100 0100011   -- 14   sw a2, sp(4)
10 000000001000 00010 000 00010 0010011    -- 18   addi sp, sp, 8
11 000000000000 00011 000 00001 1100111    -- 1C   jalr ra, gp, 0
12 111111111100 00010 000 00010 0010011    -- 20   addi sp, sp, -4
13 000000000000 00010 010 01100 0000011    -- 24   lw a3, sp(0)
14 0001000 01100 00000 010 00000 0100011   -- 28   sw a3, zero, 256
15 000000000000 00000 000 00000 0010011    -- 2C   nop
16 000000000000 00000 000 00000 0010011    -- 30   nop
17 00001011100000000000 00000 1101111      -- 34   jal zero, 92
18 00000000000000000000000000000000        -- 38
19 111111111000 00010 000 00010 0010011    -- 3C   addi sp, sp, -8
20 000000000000 00010 010 01101 0000011    -- 40   lw a4, sp(0)
21 000000000100 00010 010 01110 0000011    -- 44   lw a5, sp(4)
22 000000000000 00000 000 00000 0010011    -- 48   nop
23 000000000000 00000 000 00000 0010011    -- 4C   nop
24 0000000 01110 01101 001 10000 1100011   -- 50   bne a4, a5, 16
25 0000000 01101 00010 010 00000 0100011   -- 54   sw a4, sp(0)
26 000000000100 00010 000 00010 0010011    -- 58   addi sp, sp, 4
27 000000000000 00001 000 00000 1100111    -- 5C   jalr zero, ra, 0
28 0000000 01110 01101 100 11100 1100011   -- 60   blt a4, a5, 24
29 0100000 01110 01101 000 01111 0110011   -- 64   sub a6, a4, a5
30 0000000 01110 00010 010 00000 0100011   -- 68   sw a5, sp(0)
31 0000000 01111 00010 010 00100 0100011   -- 6C   sw a6, sp(4)
32 000000001000 00010 000 00010 0010011    -- 70   addi sp, sp, 8
33 000000000000 00011 000 00000 1100111    -- 74   jalr zero, gp, 0
34 0100000 01101 01110 000 01111 0110011   -- 78   sub a6, a5, a4
35 0000000 01101 00010 010 00000 0100011   -- 7C   sw a4, sp(0)
36 0000000 01111 00010 010 00100 0100011   -- 80   sw a6, sp(4)
37 000000001000 00010 000 00010 0010011    -- 84   addi sp, sp, 8
38 000000000000 00011 000 00000 1100111    -- 88   jalr zero, gp, 0
39 00000000000000000000000000000000        -- 8C
40 00000000000000000000000000000000        -- 90
```

Figure 46: Programme de calcul du PGCD de 17 et 3

Ce programme calcule le PGCD (Plus Grand Commun Diviseur) de deux nombres entiers (3 et 17 ici). Ce calcul est facilement implémentable en appliquant l'algorithme des différences :

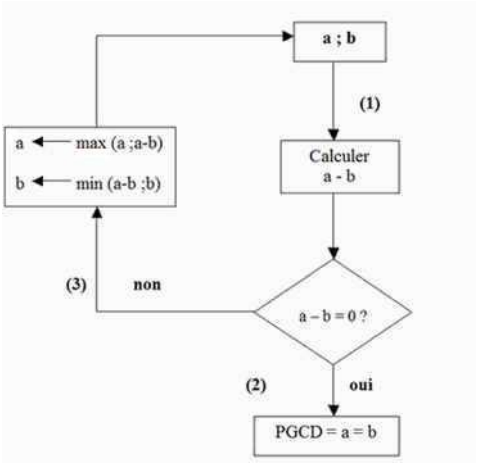


Figure 47: Algorithme des différences pour le calcul du PGCD

Le programme peut être divisé en trois parties :

- 0x00 – 0x04 : On commence par effectuer une configuration du système avant l'exécution réelle du programme. Pour cela, on initialisée la valeur du pointeur de pile, puis on récupère l'adresse de la fonction PGCD qui sera utilisée ultérieurement.
- 0x08 – 0x1C ;0x20 – 0x34 : On se trouve ici dans la partie principale du programme. Elle se divise elle-même en deux sous-parties. Dans la première, on va y définir les valeurs des arguments de la fonction, les stocker dans la pile, incrémenter celle-ci en conséquence puis effectuer un saut vers la fonction (tout en stockant l'adresse de retour de fonction). La seconde sous-partie récupérera le résultat dans la pile, décrémentera le pointeur de pile en conséquence et ira stocker la valeur en mémoire à son adresse finale.
- 0x3C – 0x88 : C'est ici que l'algorithme des différences a été implémenté. On récupère dans la pile les arguments puis on vérifie s'ils sont égaux. Si oui, via la pile, on retourne alors directement l'un d'eux comme résultat. Si non, alors on effectue une soustrait le plus petit des arguments au plus grand, puis on effectue un nouvel appel de la fonction, en passant là encore les arguments par la pile.

Les différentes instructions de ce programme ont été écrites et codées à la main, pour pouvoir être lues par le simulateur ModelSim. Dans un cas réel, cette traduction est le rôle du compilateur. Il en existe plusieurs spécialement dédiés au RISC-V.

Nous avons donc à présent un programme en binaire. Nous allons à présent, à l'aide d'un bench, simuler la connexion de notre cœur RISC-V à une mémoire d'instructions contenant ce programme. Pour la mémoire de données, aucune initialisation ne sera nécessaire ici, les variables nécessaires étant directement codées dans les instructions.

Avant la simulation, effectuons d'abord séparément le calcul du PGCD de 17 et 3 selon l'algorithme des différences :

- A = 17 et B = 3 A > B donc : A – B = 17 – 3 = 14.
- A = 14 et B = 3 A > B donc : A – B = 14 – 3 = 11.
- A = 11 et B = 3 A > B donc : A – B = 11 – 3 = 8.
- A = 8 et B = 3 A > B donc : A – B = 8 – 3 = 5.
- A = 5 et B = 3 A > B donc : A – B = 5 – 3 = 2.
- A = 2 et B = 3 A < B donc : B – A = 3 - 2 = 1.
- A = 2 et B = 1 A > B donc : A – B = 2 – 1 = 1.
- A = 1 et B = 1 A = B donc : PGCD = A = B = 1

On s'attend donc à ce que notre programme effectue ces différentes étapes, jusqu'à stocker à l'adresse 256 de la mémoire de données le résultat final PGCD = 1.

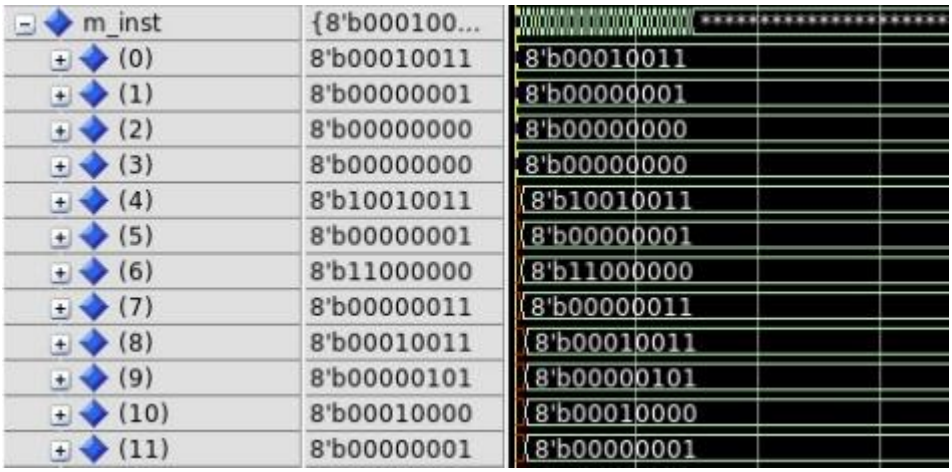


Figure 48: Stockage des instructions en mémoire

Voici ci-dessous les valeurs contenues en mémoire instructions durant la simulation. Pour ces 12 octets, on retrouve bien le code binaire des trois premières instructions 32 bits vues précédemment. L'initialisation de la mémoire instruction s'est donc bien déroulée. La mémoire de données n'étant pas initialisées ici, passons directement à l'exécution du programme.

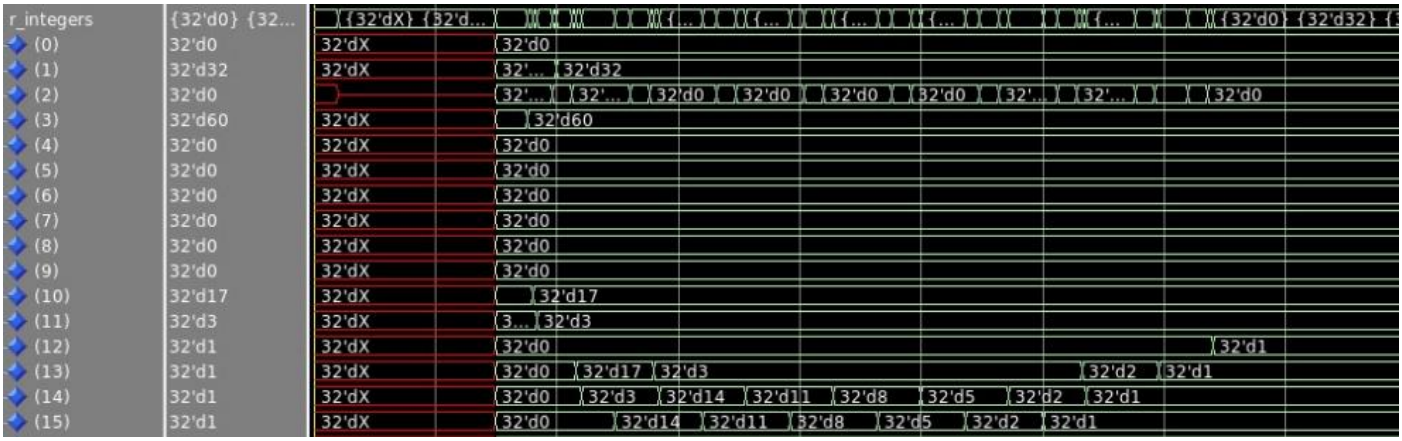


Figure 49: Evolution des registres durant l'exécution du programme

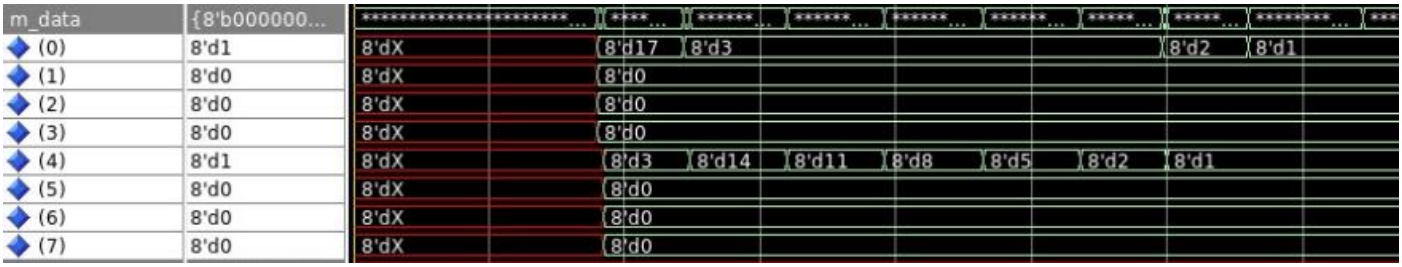


Figure 50: Evolution de la pile durant l'exécution du programme

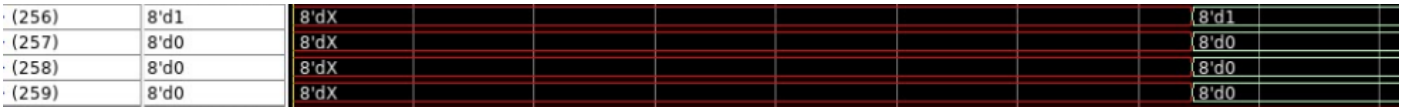


Figure 51: Stockage du résultat final dans la mémoire

Intéressons-nous tout d'abord à l'évolution des registres du processeur. Dans le registre[1], on retrouve bien l'adresse de retour de la fonction : 32 = 0x20. Dans le registre[2], on retrouve l'évolution du pointeur de pile, initialement égal à 0x00, et qui s'incrémente/ décrémente en multiple de 4 au cours des différents appels de la fonction PGCD. Le registre[3] contient lui l'adresse de la fonction : 60 = 0x3C. Enfin, les registres[10 – 11], contiennent les valeurs de départ, le registre[12] le résultat final et les registres[13 – 15] les différentes valeurs utilisées par la fonction PGCD.

Observons maintenant la zone mémoire correspondant à la pile. On y retrouve les éléments passés en arguments lors des différentes itérations de la fonction. Si l'on se réfère au calcul effectué précédemment à la main, on retrouve bien l'évolution attendue des arguments avec pour finalité les deux éléments égaux à 1.

Enfin, si l'on observe la case mémoire où doit être stocké le résultat, on constate qu'après une longue période sans valeur définie (car pas d'initialisation), le résultat attendu y est finalement écrit.

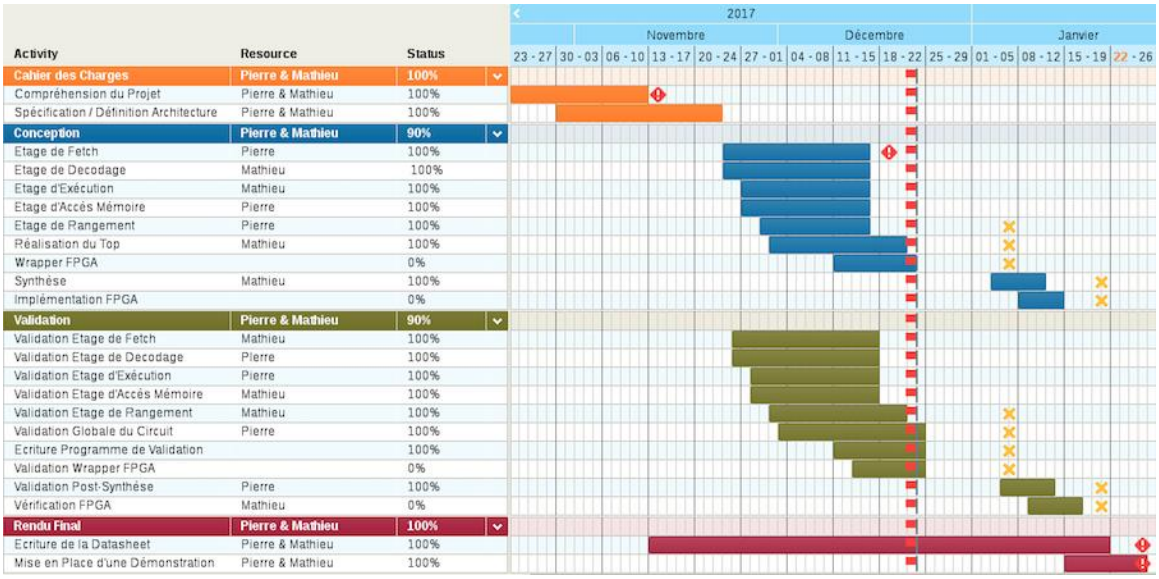
Ce programme relativement simple, en plus d'illustrer la partie « Manuel d'utilisation », nous permet de montrer la présence des différentes fonctionnalités attendues :

- Gestion des différentes instructions de calcul,
- Gestion des différentes instructions d'accès mémoire,
- Gestion des différentes instructions de branchement et de saut,
- Optimisation pour la gestion des différentes dépendances de données
- Utilisation d'une pile logicielle pour le passage des arguments.

IV. Planning

Dans le cadre de ce projet, comme détaillé précédemment, nous avions à concevoir un processeur RISC-V pour une application IoT.

La première chose à faire après avoir pris en main le projet était de définir un plan d'action. Voici celui que nous avions mis en place en début de projet :



Activity	Resource	Status
Cahier des Charges	**Pierre & Mathieu**	**100%**
Compréhension du Projet	Pierre & Mathieu	100%
Spécification / Définition Architecture	Pierre & Mathieu	100%
Conception	**Pierre & Mathieu**	**90%**
Etage de Fetch	Pierre	100%
Etage de Decodage	Mathieu	100%
Etage d'Exécution	Mathieu	100%
Etage d'Accès Mémoire	Pierre	100%
Etage de Rangement	Pierre	100%
Réalisation du Top	Mathieu	100%
Wrapper FPGA		0%
Synthèse	Mathieu	100%
Implémentation FPGA		0%
Validation	**Pierre & Mathieu**	**90%**
Validation Etage de Fetch	Mathieu	100%
Validation Etage de Decodage	Pierre	100%
Validation Etage d'Exécution	Pierre	100%
Validation Etage d'Accès Mémoire	Mathieu	100%
Validation Etage de Rangement	Mathieu	100%
Validation Globale du Circuit	Pierre	100%
Ecriture Programme de Validation		100%
Validation Wrapper FPGA		0%
Validation Post-Synthèse	Pierre	100%
Vérification FPGA	Mathieu	0%
Rendu Final	**Pierre & Mathieu**	**100%**
Ecriture de la Datasheet	Pierre & Mathieu	100%
Mise en Place d'une Démonstration	Pierre & Mathieu	100%

Figure 52: Planning Prévisionnel de début de projet

Pour mener à bien notre projet, nous avons coupé notre temps en 3 grandes étapes. La première est l'écriture du cahier des charges. Pour réaliser cela, nous avons commencé par l'étude de la documentation technique du RISC-V. Après une étude approfondie du système, nous avons dû dans un premier temps choisir les spécifications et l'architecture de notre processeur. Cette étape est cruciale dans la réalisation d'un projet et ne doit pas être négligée. Cette phase n'est pas une phase de production mais elle est cruciale pour le bon déroulement de la suite du projet : une mauvaise compréhension pourrait engendrer des erreurs de conceptions qui se concluraient par des pertes de temps voire un produit non fonctionnel. Dans notre cas, une grande partie des questions concernant l'architecture de notre processeur ont été posées dès les premières semaines du projet, ce qui nous a permis de pouvoir avancé plus tard dans la conception de manière relativement fluide.

Après avoir défini l'architecture choisie, la deuxième étape était la conception du cœur. Cette étape est en lien direct avec la troisième partie qui est la validation du pipeline. Pour réaliser ces étapes, chacun des étages du pipeline ont été répartis entre les deux membres du groupe. Dans un souci d'efficacité et de bonne compréhension de la spécification RISC-V, la validation des étages a été répartie dans le sens contraire de la conception. Par exemple, la personne responsable de la conception de l'étage de fetch ne devait pas concevoir le bench associé. L'objectif étant de valider au mieux le fonctionnement de notre processeur, si l'étage et la validation sont réalisés par la même personne, la personne ne traitera que les cas auxquels elle aura pensé. En multipliant les points de vue sur un bloc et sa vérification, on diminue donc les risques d'oubli mais aussi d'erreur.

A la fin, il était normalement prévu que le processeur soit implémenté sur FPGA. Finalement, par manque de temps en vue d'une implémentation FPGA et sa validation (prise en main des outils, création d'un wrapper pour le processeur etc …), il a été décidé en accord avec les enseignants encadrants de nous focaliser sur une démonstration fonctionnelle en simulation.

V. Conclusion

Dans le cadre de notre dernière année au sein de PHELMA, nous avons été amenés à réaliser par groupe de deux un projet de A à Z. Nous avons dû pour notre part réaliser la conception d'un processeur RISC-V en vue d'une intégration dans un système IoT.

Pour réaliser ce projet, nous avons dû procéder par étape. Dans un premier temps, nous avons étudié la documentation technique. Après une analyse complète de cette dernière, nous avons commencé à réaliser l'architecture de notre processeur. Nous avons ensuite validé séparément l'ensemble des étages du pipeline, puis nous avons réalisé un test complet du processeur à l'aide d'instruction simple. Nous avons pu valider le bon fonctionnement de notre processeur et l'illustrer par la mise en place d'une démonstration complète. Nous avons par la suite également pu procéder à une première synthèse du circuit.

Après une validation unitaire suffisante, l'objectif était de simuler le bon fonctionnement de notre processeur dans un cadre aussi proche que possible de la réalité. Un programme capable de calculer le PGCD (plus grand diviseur commun) entre deux nombres de manière récursive a été écrit en binaire et exécuté par notre processeur. Un tel programme permettait d'illustrer le fonctionnement de plusieurs points essentiels dans un processeur :

- La gestion d'instructions basiques mais nécessaires à la réalisation d'un programme,
- La gestion des accès mémoires,
- La gestion des appels et retour de fonctions logicielles (saut inconditionnel et branchement conditionnel),
- La gestion de différents cas d'aléas de donnés et d'exécution.

Dans l'ensemble, nous avons pu mener au point espéré, avec un design basique mais fonctionnel, et donc capable d'exécuter de programmes très simples. L'une des étapes clés de ce projet a vraiment été la compréhension du projet et des spécifications RISC-V : poser sur papier l'ensemble de nos idées nous a permis de nous poser les bonnes questions et de ne pas nous perdre lors de la conception. La mise en place de certaines optimisations supplémentaires (comme la suppression des bulles en cas de dépendance de données suite à un chargement en mémoire), l'intégration sur FPGA ou la mise en place de différents niveaux de privilège d'exécution auraient pu être d'autres aspects très intéressants à aborder

Ce projet a été vraiment très intéressant et enrichissant sur le plan personnel. Le fait de nous avoir fait réaliser un projet en entier nous a permis de mieux comprendre l'ensemble des étapes à réaliser, mais aussi de comprendre l'importance de l'étape de compréhension du système pour ne pas se diriger dans de mauvaises directions. De plus, ce fut l'occasion de non seulement mettre en pratique de nombreux concepts vus en cours, mais surtout de pousser encore un peu plus loin notre compréhension de tels systèmes numériques. Bien évidemment, avoir travaillé sur l'implémentation d'un cœur RISC-V nous a permis d'acquérir des spécifiques au sujet qui pourront nous servir plus tard dans notre vie professionnelle.

Nous tenons à remercier Régis LEVEUGLE et Michele PORTOLAN pour nous avoir proposer ce sujet de projet fort enrichissant, pour leurs conseils ainsi que leur aide tout au long de ce projet.

VI. Annexes

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

Figure 53: Ensemble des instructions RV32I gérées par le Cœur

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Figure 54: Convention d'utilisation des registres d'entiers 32 Bits