Sorbonne Université

Master SESI

Rapport PSESI

Réalisation d'un processeur RVUI32 RISC-V

Mai 2019

Quentin CHASSET, William FABRE, Adrian SATIN

Encadrant

Jean-Lou Desbarbieux

Table des matières

Situation du projet

1.1 Objectif du projet

Le but de ce projet est la réalisation VLSI d'un processeur de type RISC-V en langage VHDL ainsi que sa synthèse, son placement et son routage avec des outils libres. L'outil GHDL [1], la suite Alliance [2] et la suite Coriolis [3] ont été utilisés pendant le projet.

1.2 Qu-est-ce que RISC-V ?

Le RISC-V est un jeu de d'instructions de type RISC (Reduced Instruction Set Computer) libre et ouvert. Il est développé à l'université de Berkeley depuis 2010. Depuis 2015 une fondation a été créée pour améliorer le jeu d'instructions et s'ouvrir à l'industrie. Aujourd'hui la fondation RISC-V [4] compte plus de 200 membres et s'occupe de la définition de l'ISA [5]. À ses débuts le RISC-V était développé dans un but de recherche et pédagogique, aujourd'hui avec l'aide de la fondation, les visées sont plus industrielles, tout en ne délaissant pas le côté universitaire.
Seul le jeu d'instructions est défini, aucun choix architectural n'est fait, ainsi tout type d'architecture peut être utilisé. Par exemple pipelinée ou non, ou bien in-order ou out-of-order. Néanmoins, le jeu d'instructions est conçu pour pouvoir implémenter ces différentes architectures, et des composants modernes des processeurs par exemple des prédicteurs de branchements ou des pré-fetcheur d'instructions. La partie USER du jeu d'instructions est partiellement fixée, notamment les extensions communes. La spécification définie aussi des extensions du jeu d'instructions qui seront développées ultérieurement. La partie PRIVILÉGIER du jeu d'instructions est encore en réflexion, mais elle est d'ores et déjà implémentée par quelques processeurs RISC-V existants.

1.3 Les Enjeux

L'une des ambitions de RISC-V est de définir un jeu d'instructions moderne, en tenant compte des erreurs du passé. Par exemple, il supporte l'adressage sur 32 ou 64 bits, mais il est prévu une possible extension sur 128 bits pour ne pas manquer de bits pour adresser encore plus de mémoire qu'aujourd'hui [4]. Le but de ce jeu d'instruction est aussi d'être universel, pouvant être utilisé pour implémenter d'un micro-controleur à un supercalculateur. Le jeu d'instructions est sous licence Creatives Commons 4.0 avec obligation d'attribution, cela permet à toutes personnes, physiques ou morales, de créer des processeurs RISC-V sans avoir à signer d'accord de non-divulgation ou même à payer des redevances. Seul le nom « RISC-V »est légèrement protégé : en cas d'usage commercial, il faut être membre de la fondation RISC-V pour l'utiliser. L'idée d'être libre et ouvert à la contribution fait partie intégrante de la façon de fonctionner du RISC-V [3], par exemple l'élaboration de la spécification est ouverte à toutes contributions sur la plateforme de partage de code github [6].

1. http ://ghdl.free.fr/
2. https ://www-soc.lip6.fr/equipe-cian/logiciels/alliance/
3. https ://www-soc.lip6.fr/equipe-cian/logiciels/coriolis/
4. https ://riscv.org/risc-v-foundation/
5. ISA : Instruction Set Architecture, c'est l'architecture externe du processeur
6. `https://github.com/riscv/`

Cadre du Projet

2.1 Une architecture modulaire

Le jeu d'instructions d'un processeur RISC-V est composé d'une base à laquelle on peut adjoindre zéro, une ou plusieurs extensions. La base I est le minimum nécessaire pour réaliser un processeur. Elle est déclinée en plusieurs tailles selon la taille des registres et de l'espace d'adressage : 32, 64, 128 bits (resp. RVI32, RVI64, RVI128). Les différentes extensions fixées par la fondation RISC-V sont décrites dans le Instruction Set Manuel Volume 1 [1] ou Instruction Set Manual Volume II : Privileged Architecture [2] qui correspond aux extensions avec privilèges. Les bases et les extensions sont à choisir en fonction de la plateforme supportée.
Par exemple, la base E, pour embedded, ne supporte que 16 registres et n'implémente pas les extensions ajoutant le support des nombres à virgules flottantes, ce qui en fait un bon candidat pour des applications dans l'informatique enfouie. À ce jour, il existe 13 extensions au jeu d'instruction de base. Les bases et extensions peuvent être "fixées" ce qui signifie que la composante ne sera (très certainement) pas modifiée durant le processus de ratification de la fondation.

Base	Implémente	Fixée
RVI32	Base entière (32)	Y
RVI64	Base entière (64)	Y
RVI28	Base entière (128)	Y
RVE32	BE réduite pour l'embarqué	N

TABLE 2.1 – Tableau des bases

Extension	Implémente	Fixée
M	Mult Div	Y
A	Opérations Atomique	Y
F	Virgule flottante(64)	Y
D	Virgule flottante(128)	Y

TABLE 2.2 – Tableau des extensions dans G

Quatre extensions (M,A,F et D) sont regroupées sous l'abréviation "G". Elles correspondent aux extensions attendues pour des programmes généraux. Elles ne sont pas obligatoires car elles peuvent être émulées de façon logicielle (à l'exception de l'atomicité (extension A)). Les autres extensions sont plus spécialisées et permettent d'accélérer certaines opérations en fournissant des implémentations de façon matérielle. Par exemple, l'extension V, pour Vecteur, permet de vectoriser les opérations. Cela accélère considérablement ces traitements.
Le coût à payer est l'implémentation de plus de matériel et donc un coût et une consommation supérieure. Comme cette extension est complètement facultative, son support ou non est à la discrétion des architectes et uniquement présente si elle est utile au but poursuivi par le processeur.
Une autre extension, C pour compressed, permet d'exécuter des instructions de taille 16 bits, ce qui permet de réduire la taille du code de 25 à 30%. Les instructions compressées sont des raccourcis des instructions les plus populaires de la base I. Les instructions compressées et standards sur 32 bits peuvent être mixées, toutefois les instructions compressées ne permettent d'adresser que 16 registres (au lieu de 32).
C'est cet ensemble de bases et extensions facultatives qui permet à RISC-V d'être extrêmement modulaire.

Identification des tâches à réaliser

Nous avons identifié 3 grandes étapes de conception : la conception de l'architecture du processeur, sa réalisation en VHDL synthétisable, et la phase de synthèse logique et de placement/routage pour obtenir un schéma électronique ainsi que le dessin des masques qui permettrait d'envoyer le processeur en fonderie.

3.1 Conception de l'architecture interne d'un RISC-V RV32I USER-only

L'architecture externe est l'ensemble des descriptions nécessaires à la compréhension du programmeur pour utiliser un objet. Elle nous est entièrement décrite dans la spécification (version 2.2 [1]). Notre rôle ici est de définir l'architecture interne permettant la réalisation du processeur.

Pour cette tâche, nous avons suivi une approche du haut vers le bas pour définir l'ensemble des composants nécessaires :

- Définition de l'utilisation du processeur ainsi que ses limites.
- Définition de l'interface du processeur. C'est-à-dire quels sont les signaux qui permettent d'interagir avec le processeur.
- Choix sur la structure interne.
- Définition des interfaces et interactions des composants du processeur.

Cette analyse a permis d'obtenir un schéma de l'ensemble des composants ainsi que leurs interactions.

La description de la réalisation fut une étape critique. Pour éviter au maximum de revenir sur le travail déjà fourni, nous avons vérifié que l'architecture définie respectait bien l'ensemble de la spécification d'un RV32I à chaque étape et pour chaque composant. Une description est valide si et seulement si la somme des composantes qui la compose est valide, que leur interaction est valide et qu'elle possède elle-même une définition valide.

Procédure de recette

La procédure de validation est complexe à traiter, nous entendons par là qu'aucun outil ne permet une validation à part l'essai/erreur. Cependant, nous avons trouvé la comparaison avec d'autres architectures un bon complément.

Une autre méthode, qui n'a pas été utilisée par manque de connaissances et de temps, est la définition de nos interfaces à travers un langage du type LUSTRE [1] afin de démontrer la validité par rapport à une spécification.

1. https://fr.wikipedia.org/wiki/Lustre_(langage)

3.2 Réalisation VLSI synthétisable d'un RISC-V RV32I USER-only

Nous avons implémenté, en langage VHDL, l'architecture interne conçue grâce à la méthode précédente. Pour ce faire nous avons créé de petits composants VHDL que nous avons instanciés pour créer l'ensemble du processeur.

Le langage VHDL permettant aussi de décrire des constructions qui ne sont pas synthétisables, c'est-à-dire ne pouvant ensuite être transformé en porte logique de façon automatique, une attention particulière a été mise pour réaliser des composants qui sont synthétisables et donc réalistes pour une implémentation matérielle.
Dans la réalisation plusieurs instructions de la spec 2.2 de RV32I ne sont pas implémentées :

- L'instruction FENCE car elle dépend fortement de l'architecture externe des composants gérant les accès mémoires, ce qui n'est pas défini, ni même existant dans le cadre de notre projet.
- Les instructions permettant l'accès aux compteurs utilisateurs ainsi que les instructions permettant de lire le CSR (Control Status Register) ne seront également pas implémentées non plus. Ces instructions, dans les spécifications futures, deviendront des extensions et seront donc optionnelles.
- L'instruction `ecall` permettant de réaliser un appel vers l'environnement d'exécution. En effet, notre réalisation n'ayant pas de mode privilégié, cette instruction ne peut être implémentée.
- L'instruction `ebreak`, car cette instruction branche également vers l'environnement d'exécution.

Procédure de recette

Pour valider un composant nous avons utilisé des jeux de tests contenant les valeurs d'entrée ainsi que les valeurs de sortie attendues. Leur exécution sur nos composants a permis de valider ou non le fonctionnement du composant. Cette solution n'est pas adaptée à tous les composants et ne passe pas bien à l'échelle, mais elle est très pratique pour de petits composants.

La fondation RISC-V fournit un ensemble de tests unitaires [2], permettant de tester individuellement chaque instruction du jeu d'instructions du RISC-V. Dans notre cas seules les instructions appartenant à la base I ont été testées. Celles-ci sont regroupées dans le jeu de test `rv32ui`. Plusieurs tests ne seront pas implémentés : par exemple le test de l'instruction `FENCE`, les autres instructions non implémentées n'étant pas testées dans le jeu `rv32ui`.
Pour lancer ces jeux de tests sur notre implémentation nous avons utilisé le simulateur `GHDL` [3].

Le processeur est instancié dans un composant VHDL qui permet de faire l'interface avec le simulateur RISC-V développé dans un autre projet PSESI. Ceci est réalisé grâce à de l'interopérabilité entre VHDL et le langage C utilisé pour le simulateur. Cette interface permet de fournir l'abstraction mémoire permettant de lancer un binaire au format `ELF` dans le simulateur GHDL sur notre implémentation. L'évaluation de cette tâche s'est donc faite sur les tests unitaires fournis par la fondation RISC-V, exécutés sur la réalisation VHDL, à l'aide du simulateur GHDL.

2. `https://github.com/riscv/riscv-tests`
3. `http://ghdl.free.fr/`

3.3 Synthèse et placement/routage d'un RISC-V RV32I USER-only

Après la réalisation en VHDL, une phase de synthèse intervient. Cette phase est essentielle puisqu'elle permet de vérifier que la description effectuée en VHDL peut être traduite en portes logiques exitantes dans des bibliothèques de cellules.
A l'issue de cette étape de synthèse, nous obtenons une description en porte logique appelée netlist, qui donne le détail de connexion de chaque porte logique entre elles. Cette étape est réalisée avec la suite Alliance (outils VASY, BOOM, BOOG et LOON) développé au LIP6.

L'étape suivante est une phase de placement qui permet d'interpréter le schéma électronique établi par la netlist en réalisant un placement sur une grille de chaque portes logiques que comporte la netlist. Le routage va par la suite connecter les portes logiques entre elles et les alimenter.
Ces deux dernières étapes sont réalisées grâce au logiciel Coriolis, aussi développé au LIP6, et nous permet d'obtenir un dessin des masques.

Procédure de recette

Le résultat de cette tâche est un dessin de masque pouvant être envoyé en fonderie pour intégration. Il s'agit de créer ce dessin, mais nous ne pouvons pas vérifier sa validité en raison du manque de connaissance et de temps dans ce projet. Le dessin des masques nous permet néanmoins de valider que le design choisi est, a priori, réalisable (ce qui ne garanti pas que le composant post-synthèse est bien celui que nous avons décrit, pensé).

Conception de l'architecture

4.1 Définition de l'utilisation du processeur ainsi que ses limites

Nous avons choisir de réaliser un processeur RISC-V, en version 32 bits. Nous supportons uniquement les instructions USER et aucune extension. Le support des exceptions et interruptions n'est pas présent. Comme il n'y a pas de mode privilégié, les instructions `SYSTEM` (`CSR`, les compteurs et les instructions `ECALL`/`EBREAK`) ne sont pas supportées.

4.2 Définition de l'interface du processeur

L'interface de notre cœur de processeur est la suivante :

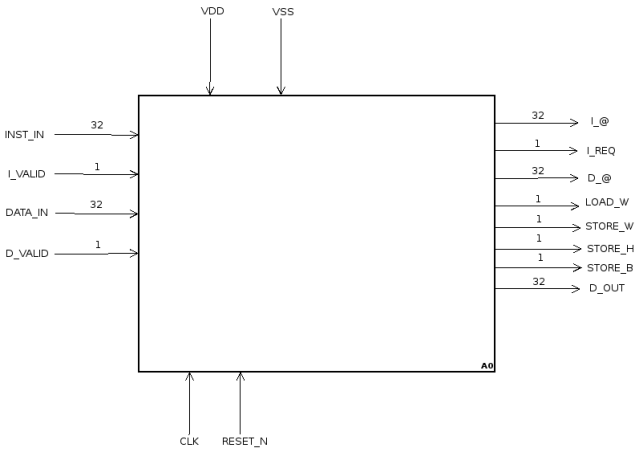


FIGURE 4.1 – Illustration de l'interface du cœur

Elle est principalement composée d'entrées et sorties permettant d'interagir avec la mémoire. Nous avons choisi un codage one-hot pour les accès mémoire (1 bit par taille, plus 1 bit pour la lecture et 1 bit pour l'écriture). Cela n'augmente pas les nombres de plots et est plus explicite par rapport à un codage compressé et évite ainsi à la mémoire située en sortie de MEM d'effectuer le décodage d'un vecteur de bit encodé pour diminuer le nombre de fils.

4.3 Architecture interne et interactions des composants du processeur

L'architecture interne correspond à l'implémentation matérielle de l'architecture externe qui est abstraite pour uniquement permettre de l'utiliser sans présupposer de choix d'implémentation. L'architecture interne n'est pas définie ni même induite par la spécification du RISC-V. Nous avons choisi une architecture interne à 5 étage pipeliné. L'exécution des instructions se fait de façon ordonnée. Les étages sont les suivants :

- `IFETCH` : Cet étage s'occupe de récupérer une instruction en mémoire à partir de son adresse.
- `DEC` : `DECODE` récupère les instructions en provenance de IFETCH pour les décoder. En parallèle, il s'occupe de gérer le pipeline, c'est à dire c'est-à-dire d'envoyer de nouvelles instructions à exécuter. Il gère le flot d'instructions, notamment les sauts et branchements en s'occupant d'envoyer les adresses des instructions suivantes vers `IFETCH`.
- `EXE` : Cet étage réalise l'exécution des instructions : les opérations logiques et arithmétiques ainsi que le calcul l'adresse d'un accès mémoire avec son offset.

- `MEM` : Cet étage réalise un accès mémoire si nécessaire et transmet le résultat de l'accès mémoire ainsi que le résultat de `EXE` vers l'étage `WB`.
- `WB` : Cet étage s'occupe de l'écriture dans le banc de registre, ainsi que dans le cas d'un accès mémoire des extensions de signes.

Les étages sont séparés par des FIFO de profondeur 1, qui permettent de sychroniser l'ensemble du pipeline. La profondeur est de 1 pour faciliter les bypass.

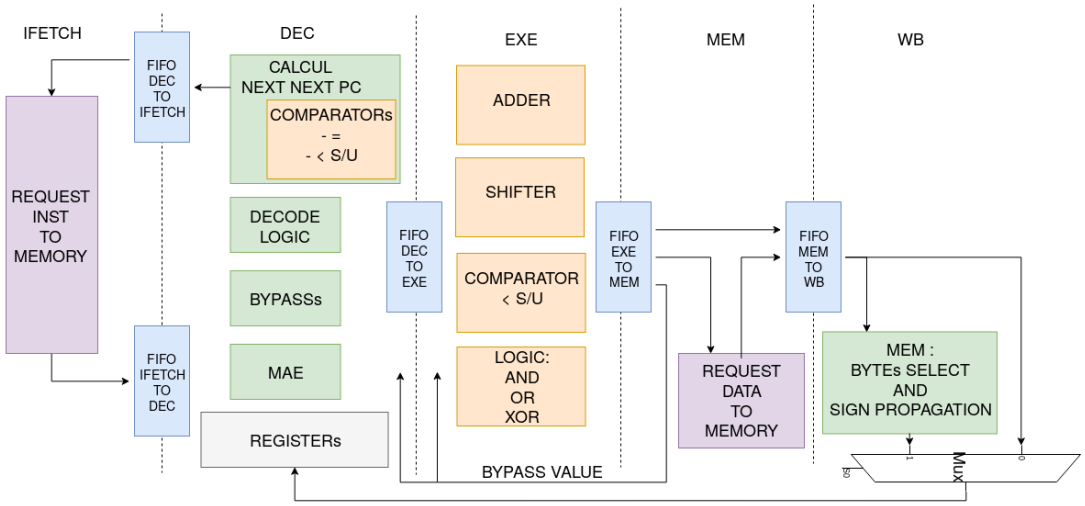


FIGURE 4.2 – Illustration de l'architecuture interne du cœur

De plus, une documentation explicative de l'architecture détaillée du processeur ainsi que d'une partie de sa réalisation a été réalisée, en anglais, elle est ci-joint en annexe.

4.4 Optimisations et choix architecturaux

Pour accélérer l'exécution des instructions ayant des dépendances de données en elles, deux bypass ont été ajoutés. Il s'agit d'un bypass de la sortie de l'étage `EXE` vers l'entrée de l'étage `DEC` qui est utilisé pour le calcul des conditions de branchement et pour l'exécution d'instruction en provenance de registres. Un second bypass est présent entre la sortie de l'étage `EXE` et l'entrée de l'étage `EXE`.
Un questionnement est survenu sur la pertinance d'avoir deux étages séparés pour `MEM` et `WB`, après discussion avec Pirouz BAZARGAN SABET, il ressort que dans un processeur avec ces caches, l'étage critique, qui détermine le temps de cycle est l'étage `MEM` puisqu'il faut chercher une adresse dans le répertoire du cache. Nous avons donc décidé de conserver deux étages distincts.

4.5 Évaluation de notre architecture - Procédure de recette

L'évaluation d'une architecture est compliquée à réaliser. Nous avons choisi de prendre en compte le CPI, le nombre de cycles moyen nécessaire à l'exécution d'une instruction. Il serait aussi important de pouvoir valider une fréquence de fonctionnement, mais cela nécessite de réaliser une étude après placement et routage. Il a été envisagé de comparer les CPI entre plusieurs réalisations d'un RISC-V existant sur une même exécutable pour évaluer l'architecture. Néanmoins, nous n'avons pas trouvé d'implémentation calculant leurs CPI. Seul le RISCV-SODOR [1] de l'université de Berkeley proposait cela, mais il nous a été impossible d'obtenir des mesures, la documentation n'étant pas à jour (le calcul du CPI semble avoir été rétiré).

Sur l'ensemble des `riscv-tests`, la moyenne des CPI est de 1.65 cycles par instructions, avec des caches parfaits répondant dans le cycle. Notons l'importance des bypass, la moyenne des CPI passant à 2.41 cycles par instruction sans les bypass. Une piste pour améliorer le CPI serait d'ajouter un bypass en sortie de l'étage `MEM` vers `DEC` et `EXE` pour les operations à base de registre (et pas les accès mémoire).

1. https ://github.com/ucb-bar/riscv-sodor

Réalisation VLSI synthétisable

La réalisation en VHDL n'a pas posé de problème particulier, l'ensemble de notre architecture ayant été pensé et détaillé au préalable.

5.1 Réalisation VHDL

Les signaux de données sont paramétrés par une valeur générique permettant de modifier plus facilement une partie du processeur pour pouvoir passer d'une architecture 32 bits vers une architecture 64 bits, ce qui nécessiterait néanmoins le support de nouvelles instructions.

5.1.1 Décode

`DECODE` effectue le décodage de l'instruction qui arrive de `IFETCH`, calcule l'adresse de PC et gère l'avancement des instructions. Son fonctionnement détaillé est fourni en annexe, de même qu'une explication sur la gestion des bypass.

Etats

L'étage `DECODE` peut être dans 2 principaux états :

- `RUN` : Dans ce cas `DECODE` s'occupe d'une instruction : il la décode, l'envoie vers EXE si nécessaire et calcul l'adresse des instructions suivantes (l'instruction immédiate dans le cas d'un branchement pris ou d'un jump, 1 instruction plus tard dans le cas général).
- `BRANCH` : Dans ce cas `DECODE` récupère une instruction inutile après un branchement pris ou un jump, et la jette. Il envoie également l'adresse de la seconde instruction à suivre vers `IFETCH`.

5.1.2 Comparateur et Shifter

Comparateur

Le comparateur doit pouvoir comparer 2 nombres sur 32 bits interprétés tout deux comme des nombres signés ou des nombres non-signés. Ce comparateur est de complexité logarithmique et est réalisé à l'aide de signaux de propagation et de génération. Son architecture, présentée dans l'UE d'Architecture des processeurs par Pirouz Bazagan-Sabet en L3 est reprise en annexe. L'ajout du support des nombres signés et non-signés est réalisé uniquement en modifiant le signal de propagation du bit 31 (bit de signe). Si les nombres doivent être interprétés comme signés, alors "0" est supérieur à "1", en revanche si les nombres sont interprétés comme non-signés, alors "0" est bien inférieur à "1". Un test-bench a été réalisé pour ce composant.

Shifter

Le Shifter permet de faire des opérations de décalage logique ou arithmétique, vers la gauche ou la droite. Il est implémenté en couche de multiplexeur ($log_2(32) = 5$ couches). Son architecture détaillée peut être trouvée dans la documentation en annexe. Un test-bench a été réalisé pour ce composant.

5.2 Simulation

5.2.1 Plateforme logiciel

La plateforme logicielle nous permet d'exécuter des binaires RISC-V sur notre modèle VHDL en simulation à l'aide de GHDL. Pour cela, le simulateur instancie notre cœur, ainsi qu'un cache de données et un cache d'instructions. Ces deux caches sont réalisé en VHDL et en C, répondent aux requêtes d'accès mémoire via le C qui modélise une mémoire et permet de lire le binaire au format ELF.

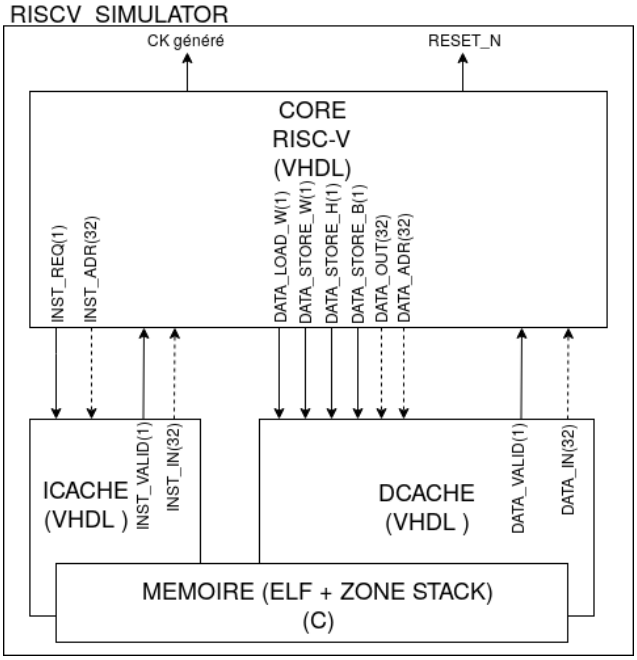


FIGURE 5.1 – Illustration de l'architecture du simulateur

La simulation consiste à exécuter le processeur, en simulant l'horloge tant que le processeur n'est pas arrivé à une étiquette `good` ou `bad` provenant du binaire ELF. À la fin de la simulation le VHDL appelle une fonction en C pour enregister s'il réussi ou échoue. Cela permet au simulateur de renvoyer un code de retour non nul en cas d'échec.
Les caches sont paramétrables, par défaut, ils répondent dans le cycle, mais il est possible des les configurer pour répondre avec une latence indépendante en cas de lecture d'instruction, de lecture de données, ou d'écriture de données. Il également possible de configurer une latence aléatoire bornée permettant de modéliser plus fidèlement le comportement réel d'un processeur+caches.

5.2.2 Calcul du CPI

Le CPI, cycles par instruction est calculé à partir de la trace de simulation au format GHW, le format de GHDL pour les traces de simulation. Ce programme prendre en argument une trace de simulation et affiche le CPI ainsi que le nombre d'instructions exécutés et le nombre de cycles de simulation.

5.2.3 riscv-test

Les `riscv-test` sont un ensemble de tests unitaires d'instructions du RISC-V fournis par la fondation RISC-V. Il est nécessaire de fournir un environnement d'exécution à ces tests permettant d'indiquer sur notre modèle de processeur, comment initialiser le processeur et comment les tests se finissent. Dans notre cas, il n'y a pas d'initialisation de registre nécessaire, mais il faut ajouter les étiquettes `good` et `bad` pour signaler la réussite ou l'échec d'une exécution. Nous avons décrit cet environnement et exécuté les tests correspondant aux instructions supportées par notre processeur. L'ensemble de ces 38 tests sont corrects.

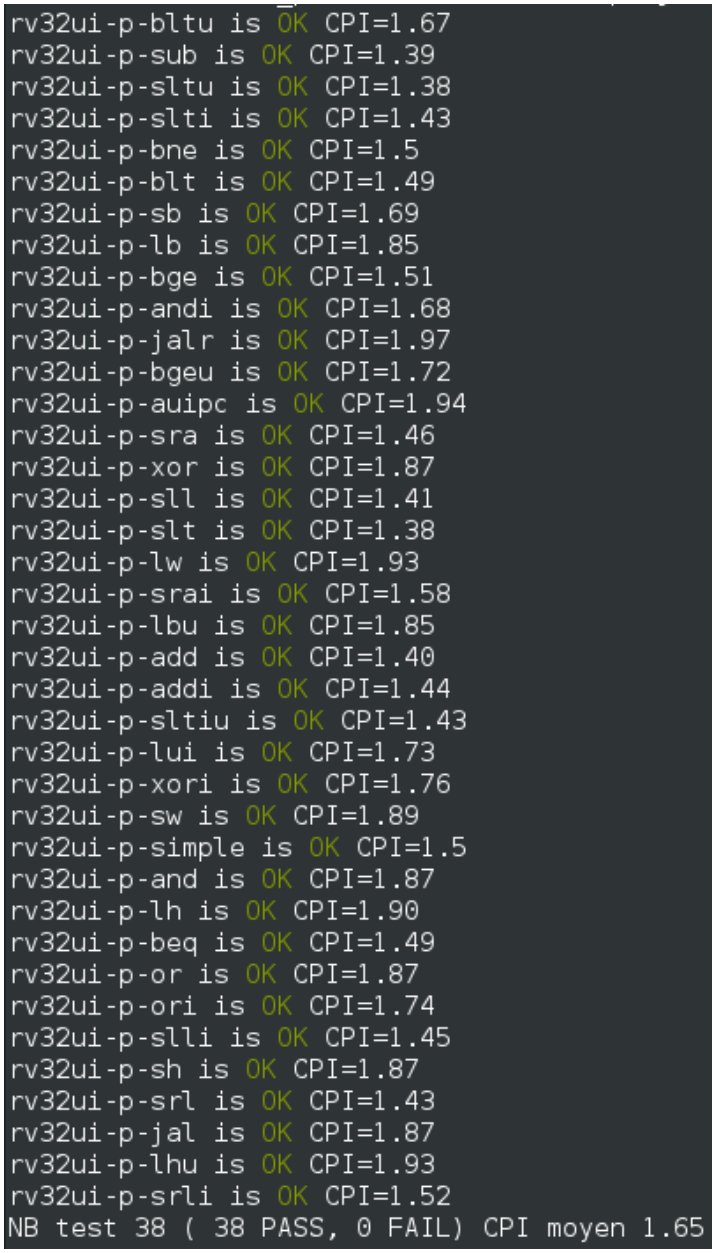


FIGURE 5.2 – riscv-tests sur notre processeur

Synthèse et placement routage

6.1 Synthèse

La synthèse a été réalisée à l'aide de la suite Alliance :

- `VASY` : VASY transforme un fichier VHDL vers le sous ensemble VBE utilisé par la suite Alliance.
- `BOOM` : BOOM utilise une description VBE pour optimiser en surface et/ou en délai.
- `BOOG` : BOOG utilise une description VBE pour transformer cette description comportementale en netlist, utilisant les cellules logiques de base utilisées par Alliance, comme celles de la bibliothèque sxlib.
- `LOON` : Ce logiciel optimise les nets.

6.2 Placement et routage

Pour réaliser le placement et le routage, il est nécessaire d'avoir une description avec des sorties/entrées sur des plots. Pour cela, nous avons instancié notre description de processeur en VHDL dans un ensemble VHDL représentant les plots et associé les sorties/entrées de notre processeur aux plots. Le placement a pu être réalisé avec la suite Coriolis, néanmoins le routage n'a pas pu être correctement effectué en raison d'un bug. Nous n'avons donc pas pu réaliser le dessin des masques. Notre design est « pad-limited », c'est-à-dire que la taille de notre design est determinée par la taille de nos plots.

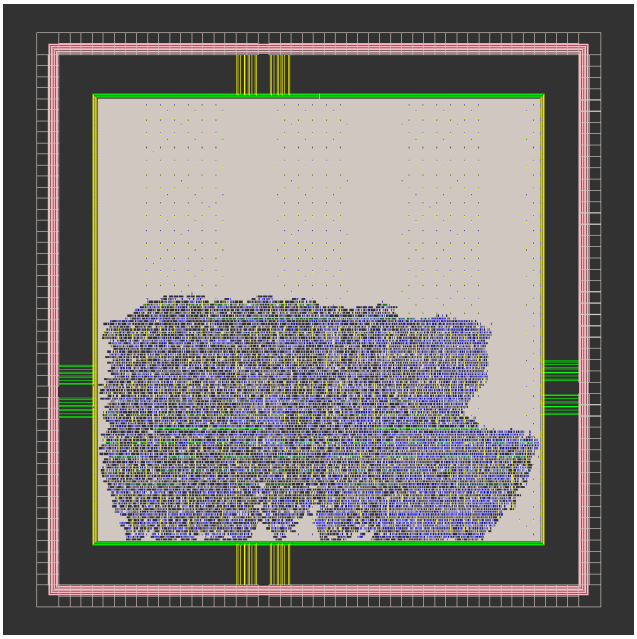
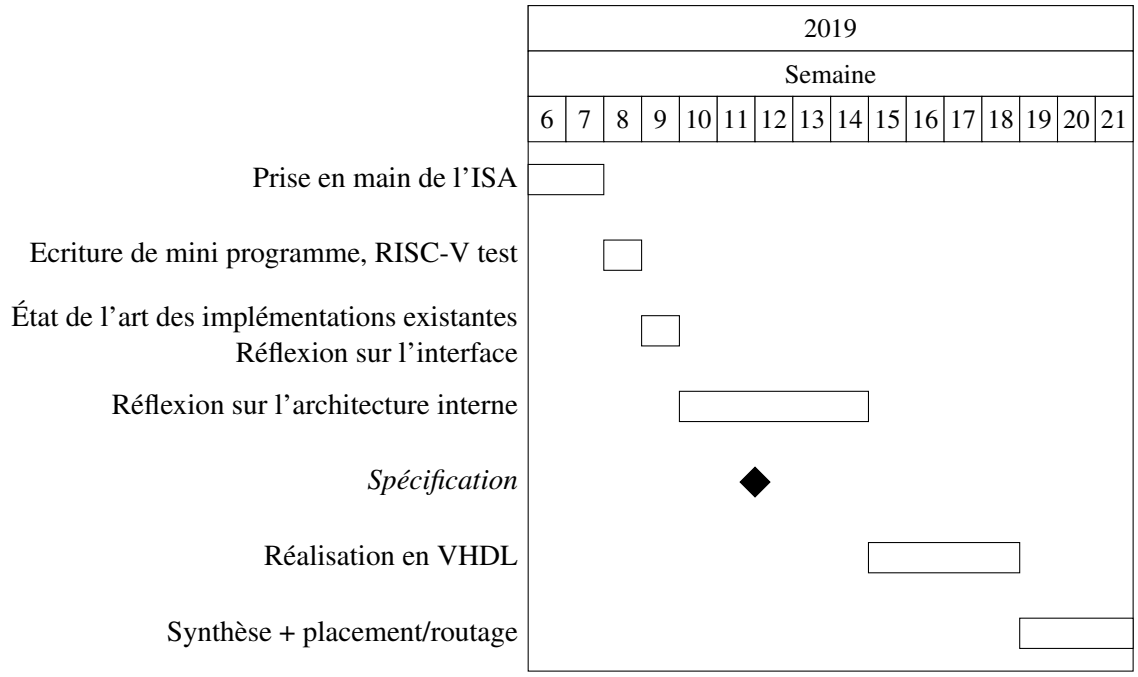


FIGURE 6.1 – Cœur avec ses plots après placement dans Coriolis (cgt)

Planning et répartition des tâches

Planning

Nous avons choisi comme mode de fonctionnement d'utiliser un gestionnaire de version, git, qui nous permet de centraliser le projet. De plus, nous utilisons également un wiki qui permet de suivre l'ensemble des éléments importants du projet, de notre compréhension du jeu d'instructions à notre bibliographie d'articles et papiers de recherche nous intéressant sur le RISC-V. Nous avons établi un planning que nous nous avons globalement respecté.



Répartition des tâches

Nous avons choisi de répartir les tâches au fur et à mesure du projet, car avant la conception de l'architecture nous n'avions pas connaissance détaillée des tâches à réaliser. Nous avons réalisé ensemble l'architecture interne globale, en réfléchissant sur les éléments et en mettant en commun régulièrement nos idées et avancements.

Voici la répartition des tâches :

- Architecture interne global : Tous
- Schéma détaillé (en annexe) : Adrian
- Squelette de base VHDL : Quentin
- Conception et réalisation du comparateur signé/non-signé : Adrian
- Étages du pipeline : Adrian
- Connexion de l'ensemble des étages : Adrian
- FIFO générique : Quentin
- Taille générique signaux de données : Quentin
- Shifter : William

- Bypass : Adrian
- Synthèse : Quentin
- Amélioration plateforme de simulation (reprise en partie de VLSI1) : Adrian
- Intégration des RISC-V tests : Adrian
- Ajout du support de la latence dans les caches : Adrian
- Documentation : Adrian
- Calcul du CPI à partir de la trace GHW : Adrian
- Interface du cœur avec les plots, Placement/Routage : Adrian

Bibliographie

[1] The risc-v instruction set manual, volume 1 : User-level isa, document version 2.2. `https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`, May 2017. 3, 4

[2] The risc-v instruction set manual, volume 2 : Privileged architecture, version 1.10. `https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf`, May 2017. 3

[3] Krste Asanović and David A. Patterson. Instruction sets should be free : The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014. 2

[4] Gordon Bell and William D. Strecker. Computer structures : What have we learned from the pdp-11 ? In *Proceedings of the 3rd Annual Symposium on Computer Architecture, Clearwater, FL, USA, January 1976*, pages 1–14, 1976. :conf/isca/BellS76

[5] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016.

Annexe

R5

Manual

https://framagit.org/psesi-riscv-2019/projet/

Contents

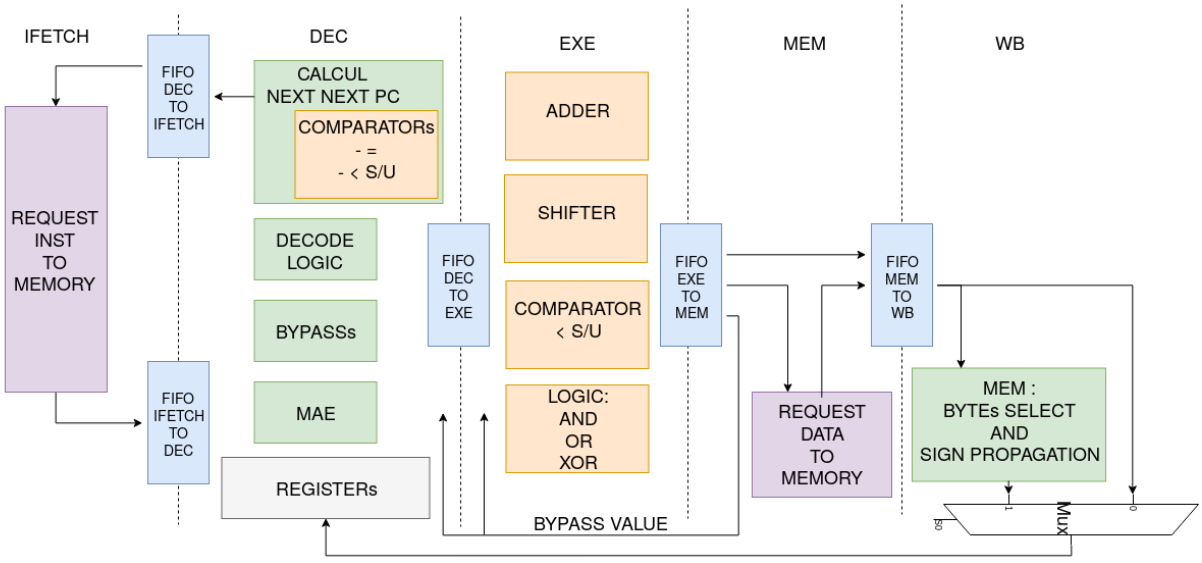
Document revisions

Rev	Date	Description
0.1	May 2019	Initial documentation of R5

1 Introduction

R5 is an 5 stage RISC-V[1] 32 bits processor. ISA supported by R5 is the base integer, RV32UI from 2.2 reference [2] of RISC-V Foundation. All user instructions are supported except for `SYSTEM` instructions, since R5 does not have support for the privileged mode, and some instructions (FENCE.I, counters, and CSR) that will be implemented in the next version of the specification. The `fence` instruction is implemented as a `nop`, since R5 only has one hart[1]. No exceptions nor interruptions are supported. Extensions are not supported.

Here is an overview of our pipeline, a schematic with all the details can be found as annexe.



PC at reset is `0x08000000`. We have tested our design with riscv-tests. All is synthesizable with Alliance. More details on how to use it can be found in `README.md` in the code

2 Interface of the core

The R5 core has the following pads interfaces :

- ck (IN - 1 bit) : Clock
- reset_n (IN - 1 bit) : Synchronous reset
- VDD (IN - 1 bit) : Power Supply VDD Voltage
- VSS (IN - 1 bit) : Power Supply VSS Voltage
- inst_adr (OUT - 32 bits) : Instruction address
- inst_req (OUT - 1 bit) : Instruction request
- inst_in (IN - 32 bits) : Instruction code requested
- inst_valid (IN - 1 bit) : Instruction is valid
- data_adr (OUT - 32 bits) : Data address

[1] Hardware Thread

- data_load_w (OUT - 1 bit) : Request a word load
- data_store_b (OUT - 1 bit) : Request a byte (8 bits) store
- data_store_h (OUT - 1 bit) : Request a half (16 bits) store
- data_store_w (OUT - 1 bit) : Request a word (32 bits) store
- data_out (OUT - 32 bits) : Data to store
- data_in (IN - 32 bits) : Data loaded
- data_valid (IN - 1 bit) : Load is valid

3 Pipeline

The R5 Pipeline is a synchronous pipeline. Each stage is separated with a FIFO of depth 1. It is possible to consume and produce in the same cycle. When the stage has done his work and can commit in the next FIFO, it will produce to the next FIFO and consume inside the previous FIFO at the rising edge of clock. All stage answers in 1 clock cycle, except for `IFETCH` and `MEM` that can take more clock cycles depending of number of clock cycles memory takes to answer. We also have bypass from `EXE` output to `EXE` and `DEC` input.

4 IFETCH

`IFETCH` consumes an address to fetch an instruction from `DEC` and get it from memory. `IFETCH` will forward the instruction and it's address (`PC`) to the `DEC` stage.

5 DEC

`DEC` is responsible for managing the flow of instructions (when either branch/jump or just go to pipeline) and decoding instructions to expend them to control `EXE`, `MEM` et `WB` stage. `DEC` also has a register bench with 32 registers (reg0 hardwire to 0) which have 2 read ports, 1 write port and 1 invalid port.

The `DEC` push to `IFETCH` address of instruction is required not just after the one that is currently decoded, but two in advance. With this we only have one instruction fetch lost on a branch/jump (but there is no delayed slot in RISC-V).

5.1 How the next next PC is calculated

To handle the different PCs, depending on the value of registers comparison, we calculate 3 different PC and choose later at the end of the stage (when comparison is available). The 3 different possibilities are:

- PC + 8 (or +4): Instruction is not a branch nor a jump. We push PC+8 (or +4 after a branch taken/jump) to `IFETCH`, or when it's a branch not taken.
- PC + offset : Instruction is a `JAL` or a branch taken.
- Register + offset : Instruction is a `JALR`

We choose between the possibilities with a multiplexer controlled by the result of the comparison (equal or < (signed or unsigned)). If it's not a branch, the result of the comparison is pointless, so we put the right next next PC on both inputs of this multiplexer.

5.2 How we push instructions to EXE

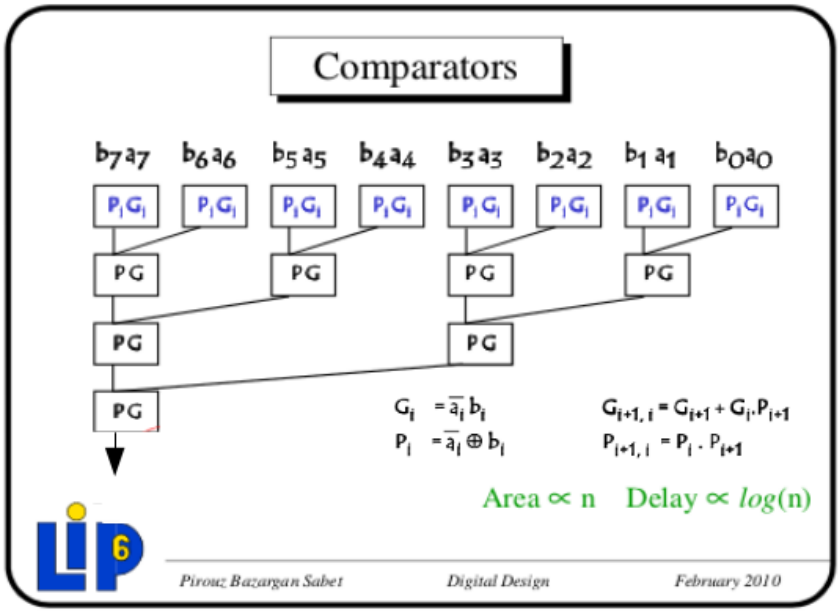
Instructions are pushed to EXE only if we have OP1 and OP2 valid (from register or from bypass) and if we can invalidate register destination.

5.3 Bypass

We have two bypasses, from the output of `EXE` to the input of `EXE` and the input of `DEC`. These bypasses are managed in `DEC` with two 5 bits registers (to store `RD`) and 1 bit validity by register. `BYPASS_RD_ON_DEC` contains RD that can be used to bypass in `DEC` stage, and `BYPASS_RD_WHEN_EXE` contains RD that will be in the output of `EXE` when the current instruction in `DEC` will be in `EXE` stage. We keep track of this by recording the RD we push to `EXE` stage and watching if `EXE` stage is empty or not (to update `BYPASS_RD_ON_DEC` if the instruction in `DEC` is waiting on a condition, and the instruction in `EXE` has gotten to `MEM`)

5.4 Comparator

The comparator is able to handle signed and unsigned comparison between two 32 bits values, interpreted as signed or unsigned values. The comparator is logarithmic, here is an overview with an 8 bit example :



Thanks to Pirouz Bazargan-Sabet for explanation.

To do signed and unsigned comparison, we only change the Generate of the sign bit (here 31). When the value is signed 1 is smaller than 0, else unsigned 0 is smaller than 1. So the Generate of bit sign become : `(UNSIGNED .` $\overline{A_i}$ `.` B_i`) + (SIGNED .` A_i `.` $\overline{B_i}$`)`. The rest of comparison is still the same.

6 EXE

The EXE stage starts by updating the OP1, OP2 and STORE values with the values of bypass if neccessary and then do these operations in parallel :

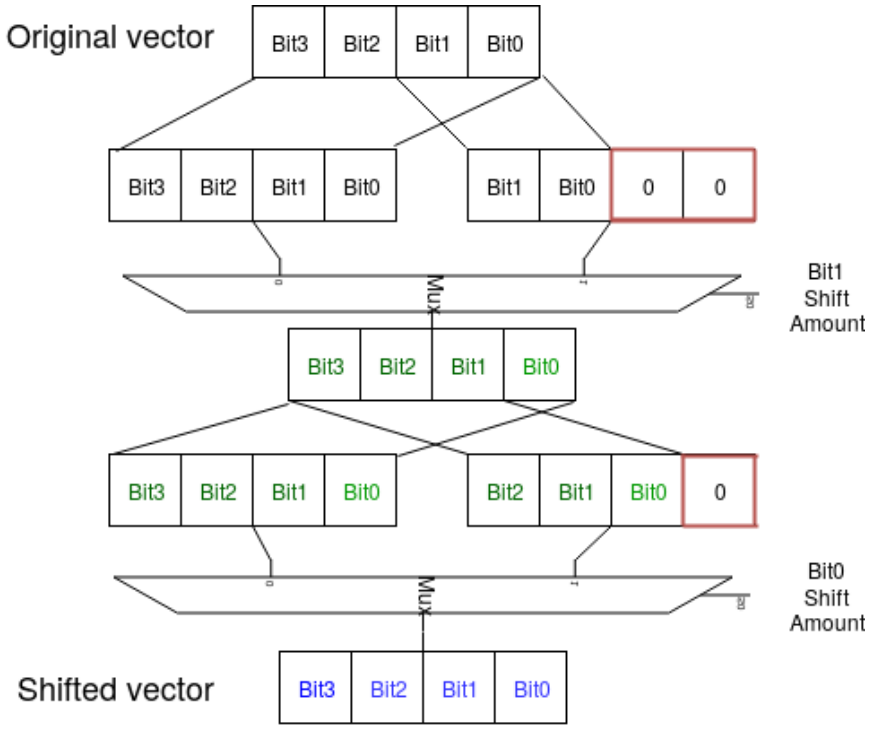
- OP1 + reversed_or_not + CIN : reversed_or_not is OP2 or NOT OP2 if it's a SUB instruction (in this case CIN = 1)
- Shifter with OP1, OP2, SHAMT
- Comparator < with OP1, OP2, signed or unsigned, the result is extended to 32 bit with a 0. Here it works in the same way than in the DEC stage.
- OP1 AND OP2
- OP1 OR OP2
- OP1 XOR OP2

The relevant result is chosen with a multiplexer cascade.

6.1 Shifter

The shifter is a logarithmic shifter, we do shift right and shift left in parallel, then we chose the result at the end.

Take as an example shift left, if bit 4 of the shift amount is 1, this means that we will shitf at least 16 (2^4) positions, what will be done is to combine 16 LSB of input and 16 bit at 0 or the sign depending of arithmetic or logic shift, for 16 LSB of output. Then we do the same but with bit 3 of the shift amount, a shift of 8 positions, and the result of the previous shift as input. As example this a logical shift left with a 4 bits input and a 2 bit shift amount:



7 MEM

`MEM` only make requests to memory if required and produces two results to `WB` : the output of `EXE` and the result from memory.

8 WB

`WB` writes back values to the register. If the instructions are a memory load, this stage also does the following (since all load requests are a request to memory to load a word).

- Select which part of the word is useful in case of a `LH` or `LB` depending on the address of the request (result from `EXE`)
- Extend or not the bit sign in case of a `LH` or `LB`

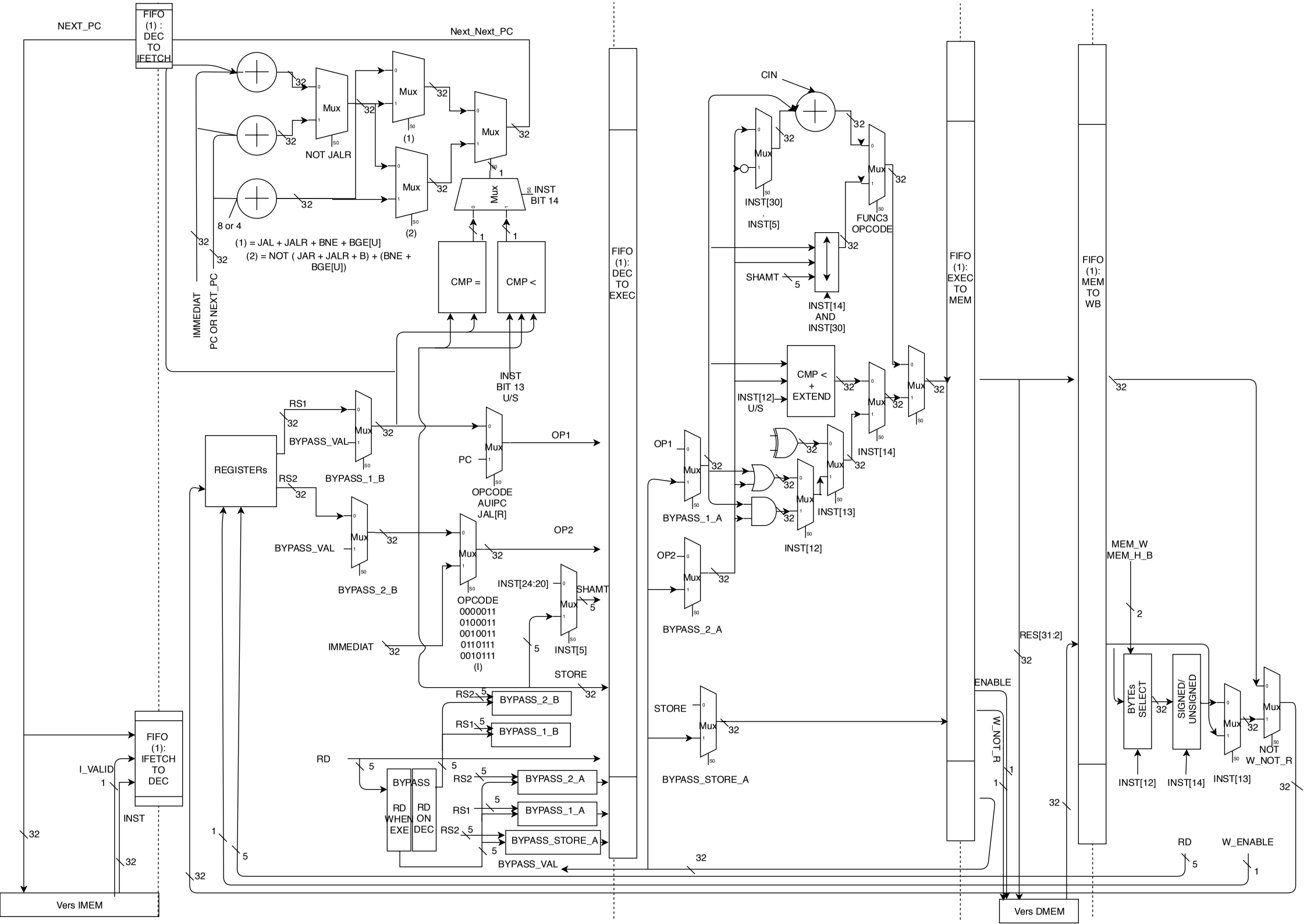
Bibliography

References

[1] Risc-v foundation | instruction set architecture (isa).

[2] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10. Technical report, CS Division, EECS Department, University of California, Berkeley, May 2017.

Annex


NEXT_PC
FIFO (1) : DEC TO IFETCH
Next_Next_PC
Mux
NOT JALR
(1)
(2)
INST BIT 14
8 or 4
(1) = JAL + JALR + BNE + BGE[U]
(2) = NOT (JAR + JALR + B) + (BNE + BGE[U])
IMMEDIAT
PC OR NEXT_PC
CMP =
CMP <
FIFO (1): DEC TO EXEC
INST BIT 13 U/S
RS1
BYPASS_VAL
BYPASS_1_B
REGISTERs
RS2
BYPASS_2_B
PC
OP1
OP2
OPCODE AUIPC JAL[R]
OPCODE 0000011 0100011 0010011 0110111 0010111 (I)
INST[24:20]
SHAMT
INST[5]
IMMEDIAT
STORE
RD
BYPASS
RD WHEN EXE
RD ON DEC
BYPASS_2_A
BYPASS_1_A
BYPASS_STORE_A
BYPASS_VAL
CIN
INST[30] . INST[5]
FUNC3 OPCODE
SHAMT
INST[14] AND INST[30]
CMP < + EXTEND
INST[12] U/S
INST[14]
INST[13]
INST[12]
BYPASS_1_A
BYPASS_2_A
BYPASS_STORE_A
FIFO (1): EXEC TO MEM
ENABLE
W_NOT_R
FIFO (1): MEM TO WB
MEM_W MEM_H_B
RES[31:2]
BYTEs SELECT
SIGNED/ UNSIGNED
NOT W_NOT_R
INST[12]
INST[14]
INST[13]
RD
W_ENABLE
FIFO (1): IFETCH TO DEC
I_VALID
INST
Vers IMEM
Vers DMEM