Rapport de Projet Numérique

Asmodee
Core RISC V Asynchrone

THIRION Nathan, LESAGE Xavier

PHELMA 2019-2020

Résumé

Ce projet à pour but de mettre en évidence les possibilités apportées par les processeurs RISC V, ainsi que l'étude des processeurs asynchrones. La problématique est donc de partir du processeur asynchrone Ibex [low] ainsi que de la documentation de M. Sparsø [Spa06] pour concevoir un core de processeur RISC V asynchrone.

Pour ce faire, il a fallu tout d'abord concevoir les blocs essentiels ainsi que des bancs de tests pour leur vérification, puis la conception d'un circuit de contrôle asynchrone comme établi dans le document qui nous a été fourni [FES+20].

Nous avons réussi à obtenir une preuve de concept comme il sera détaillé plus bas dans ce rapport.

Sommaire

Table des figures

I Introduction

I.1 Sujet

Le sujet est comme énoncé plus haut la conception d'un core de processeur asynchrone RISC V à partir de la documentation du processeur Ibex [low] et du document fourni exposant la fonctionnalité d'un circuit asynchrone [FES+20]. Pour ce faire, nous avons tout d'abord du trouver et comprendre la documentation nécessaire pour comprendre les deux notions principales du projet : RISC V et asynchrone. Ensuite nous avons conçu les blocs d'un processeur synchrone RISC avec en tête son implémentation asynchrone (cf. Chap. II). Pour enfin établir un contrôleur asynchrone et piloter les blocs précédemment définis (cf Sec. II.1.7).

Dans cette partie nous allons découper la problématique autour des notions clés.

I.2 RISC V

La notion de RISC V veut dire Reduced instruction set computer version 5 qui se traduis par processeur à jeu d'instructions réduit. C'est à dire que RISC V établi une norme pour les instructions du processeur sur leurs taille, opérandes, etc. Mais dont l'objectif est d'avoir un nombre d'instruction assez réduits, contrairement aux CISC, ce qui les rends facilement implémentable sous formes de petits processeurs pour de l'embarqué ou comme processeur secondaire dans des systèmes complexes.

I.2.1 Jeu d'instructions

Nous nous sommes restreint à un seul jeu d'instructions parmi les nombreux possibles de la norme RISC V, le set RV32I (Base Integer Instruction Set). Ce set se réduit à des opérations qui prennent un ou deux registres en entrée ou un immédiat vers un registre en sortie. Ces opérations qui sont elles-même découpées en types comme illustré dans la Fig. I.1 issu du manuel RISC V [WA19]. On voit différents types d'immédiats qu'il faudra récupérer et traités différemment. De plus, sauf les branchements type B et les écritures mémoires type S, on voit un registre de destination.

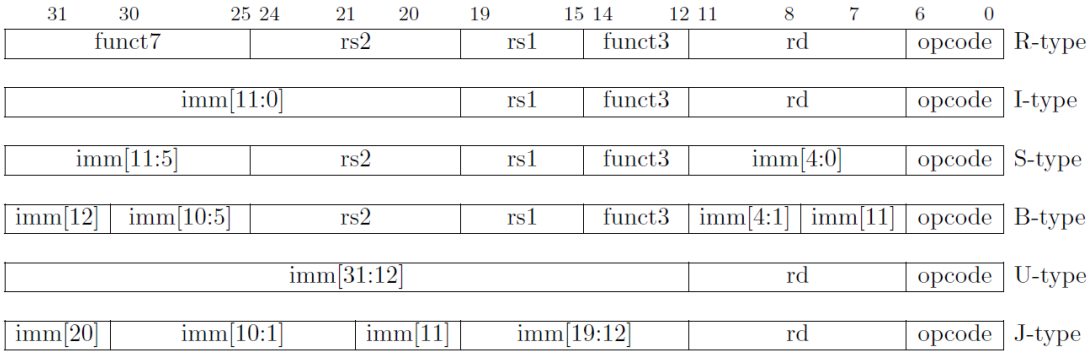


FIGURE I.1 – RISC V - Instructions

I.3 Asynchrone

Ce processeur fonctionne de manière asynchrone, chacun des blocs possèdent donc un bloc de contrôle asynchrone leur envoyant un signal d'enable au bon moment pour mettre à jours leurs entrées, remplaçant donc l'horloge. Ces blocs de contrôles communiquent entre eux par un protocole de handshake en 4 phases leur permettant de se synchroniser.

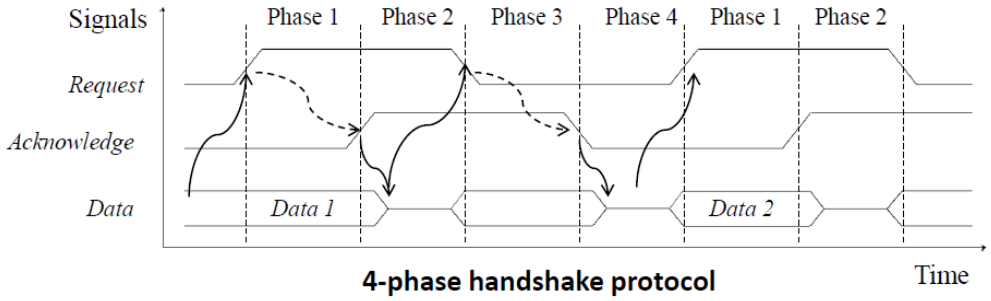


FIGURE I.2 – Protocole de handshake en 4 phases

Phase 1 : Quand un bloc envoi un bus de donnée à un autre, son contrôleur envoi un signal de requête req au contrôleur du bloc suivant.

Phase 2 : Quand le bloc suivant reçoit cette requête, si il est disposé à recevoir les données, il les enregistre et renvois un signal d'acquittement ack.

Phase 3 : Quand le bloc émetteur reçoit ce signal d'acquittement, il peut invalider les données qu'il envoi et éteindre son signal de requête.

Phase 4 : Quand le bloc récepteur ne reçoit plus le signal de requête entrant, il éteint à son tour son signal d'acquittement et le Handshake est terminé.

Le fonctionnement des contrôleurs est détaillé dans la section II.1.7.

II Architecture

II.1 Définition

II.1.1 Accès Mémoire (LSU et Fetch)

Dans la Fig. A.1 on voit la globalité de l'architecture du core Ibex avec deux accès mémoires à ses extrémités : d'une part le bloc Fetch qui lit une instruction dans la mémoire d'instruction à l'adresse correspondant à PC (programm counter) ; et d'autre part la LSU (Load Store Unit) qui est le contrôleur de la mémoire de données. Ces deux blocs sont très similaires et on peut établir un protocole d'écriture et lecture mémoire similaire pour les deux blocs, bien que plus simple pour Fetch. Pour le protocole nous avons récupéré le protocole proposé dans l'architecture Ibex [low], cependant nous avons dû adapter ce protocole illustré dans la Fig. II.1 pour une utilisation asynchrone. Pour ce faire, nous différencions les signaux de request et acknowledge du circuit asynchrone (cf. Sec. II.1.7) et les bus de données.

Dans la Fig. II.2 on voit l'architecture mise en place. l'unité LSU est découpé en deux pour l'entrée et la sortie de la mémoire. On retrouve le nom des signaux de contrôles avec leur correspondance sur les contrôleurs asynchrones.

Pour l'étage de Fetch on enlève juste la partie écriture mais le reste est équivalent.

Dans la Sec. IV.1.1 est expliqué son test avec une mémoire artificielle.

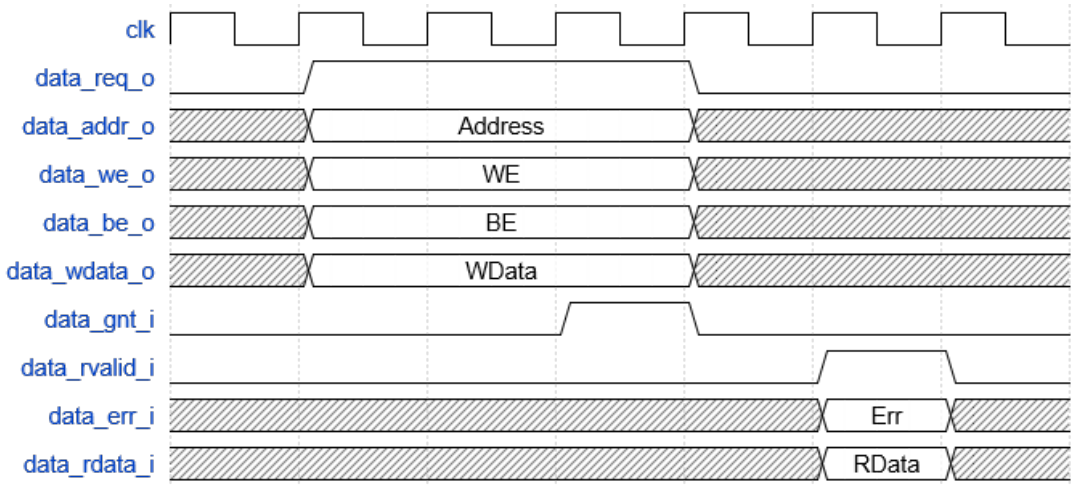


FIGURE II.1 – Protocole d'accès mémoire

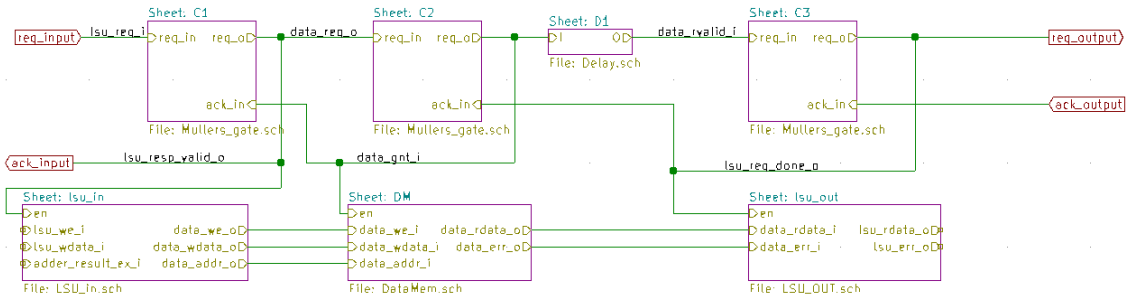


FIGURE II.2 – Diagramme de blocs accès mémoire

II.1.2 Gestion des GP Registre

A l'instar de la mémoire, les registres sont une mémoire accessible dans le core, 32 registres de 32 bits chacun. On peut y écrire ou lire une information et ils sont utilisés pour les calculs dans le processeur. On a découpé les accès registre en lecture et écriture.

La lecture est nécessaire pour la définition des opérandes entre Decode et Issue (cf Sec. II.1.3 et Sec. II.1.4). On prend en entrée une ou deux requêtes car le core peut lire jusqu'à deux registres en même temps, ainsi que les adresses correspondantes sur cinq bits (32 GPR de 32 bits).

L'écriture est faite en fin de boucle soit pour enregistrer les résultats d'une opération (cf. Sec. I.2.1) ou le chargement d'une donnée de la mémoire vers un registre.

II.1.3 Le bloc decode

Le bloc decode reçoit une instruction de 32 bit provenant du bloc fetch. Selon le type de l'instruction, il envoi les instructions et les opérandes nécessaires aux différents blocs d'exécution. Le type de l'instruction est définit à la fin, dans son opcode, les immédiats ont des positions différentes selon leurs types, et les registres de sources et destination sont tout le temps au même endroit. La norme à suivre vient du jeu d'instruction sélectionné comme exposé dans la Sec. I.2.1.

Certaines instructions ont besoin de lire des valeurs contenues dans des registres, le bloc décode envoi donc toujours deux adresses de lecture et deux requêtes de lecture, positives ou non, au bloc reg file.

Le reste des informations extraites de l'instruction, telles que tout les immédiats lus, le type d'immédiat à utiliser, les types d'opérandes et opérateurs à envoyer aux ALUs et les requêtes de lecture/écriture, sont envoyées au bloc Issue qui se chargera de la suite du

décodage.

II.1.4 Le bloc Issue

Le bloc Issue communique directement avec les ALUs, la LSU, et le bloc d'écriture en registres, il envoi constamment les données nécessaires aux fonctionnements de ces blocs ainsi que les signaux de requêtes correspondants. Il reçoit ces données directement du bloc decode mais aussi les données lues dans les registres venant du bloc regfile et la valeur du PC actuel venant de *pc_alu*.

Pour l'ALU, le bloc Issue envoi deux opérandes, un opérateur et un signal de requête. la nature d'un opérande dépend du type de l'instruction, il peut être un des immédiats hérités du bloc décode ou le résultat de la lecture d'un registre. Cette information fait aussi partie des informations envoyées par le bloc décode au bloc Issue.

Pour le bloc *pc_alu*, les opérandes peuvent être des immédiats, des valeurs lues en registres ou la valeur actuelle du PC. Tout comme pour l'alu, la nature des ces opérandes est identifiée par decode, qui la communique à Issue. Contrairement à l'alu cependant, il n'y a pas d'opérateur à communiquer car l'opération est toujours une addition. Il existe aussi le cas des instructions de branchements conditionnels, qui nécessite que le résultat de l'alu soit pris en compte par *pc_alu*. Issue envoi donc aussi à *pc_alu* un signal lui indiquant si l'opération est conditionnelle, donc si le résultat de l'alu doit être pris en compte.

Pour la LSU, Issue envoi en permanence le résultat de la lecture d'un des registres qu'il reçoit en entrée, car c'est la donnée à écrire en mémoire en cas de store (l'adresse de lecture ou d'écriture en mémoire provient toujours de l'ALU). De plus, Issue envoi une requête d'utilisation à la LSU et un signal indiquant si c'est une lecture ou une écriture.

Pour le bloc d'écriture en registre, Issue envoi un signal de requête d'écriture ou non, une adresse registre où effectuer cette écriture et un signal permettant au regfile de savoir quelles données d'écriture il doit considérer, entre celles provenant de l'ALU et celles provenant de la LSU.

II.1.5 L'ALU

L'ALU reçoit en entrée un signal d'activation, deux opérandes de 32 bits et un opérateur pouvant être une addition, soustraction, un décalage de bits, une opération logique ou une comparaison. Le résultat en sortie est envoyé sur le bloc d'écriture en registre, mais aussi sur l'entrée de la LSU car l'adresse de lecture/écriture de cette dernière est toujours calculée

dans l'ALU. Enfin, la sortie de l'ALU est aussi envoyée en entrée du bloc *pc_alu* car dans le cas d'un branchement conditionnel, la condition de branchement est calculée dans l'ALU et doit donc être renseignée au bloc *pc_alu*. La liste des opérateurs de ALU peut être trouvé dans l'annexe B.1, on y trouve la liste suivante :

1. opérateurs arithmétiques
   (a) ADD
   (b) SUB
2. opérateurs logiques
   (a) XOR
   (b) OR
   (c) AND
3. opérateurs shift
   (a) SRA (Shift Right Arithmetic)
   (b) SRL (Shift Right Logic)
   (c) SLL (Shift Left Logic)
4. opérateurs comparaisons
   (a) LT (Less Than)
   (b) LTU (Less Than Unsigned)
   (c) GE (Greater Equal)
   (d) GEU (Greater Equal Unsigned)
   (e) EQ (Equal)
   (f) NE (Not Equal)
5. opérateurs set (traités comme comparaisons)
   (a) SLT (Set Less Than)
   (b) SLTU (Set Less Than Unsigned)

II.1.6 Le bloc *pc_alu*

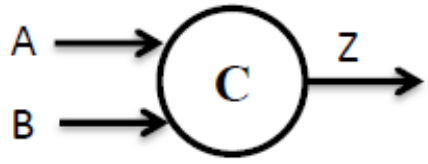
Cette ALU sert uniquement à calculer l'adresse de la prochaine instruction à lire. Il effectue en permanence la somme de ses opérandes provenant du bloc Issue sauf en cas de

branchement conditionnel. Si le signal de branchement conditionnel provenant de Issue est activé, le *pc_alu* doit évaluer la valeur en sortie de l'ALU qui représente le résultat de la condition. Donc si il y a branchement conditionnel et que le résultat de l'ALU est nul, le branchement n'est pas effectué, *pc_alu* remplace la valeur de l'opérande b (qui était la valeur du saut) par 4 (qui est la longueur d'un mot en octet), et l'additionne à l'opérande a (qui dans ce cas là est la valeur actuelle du PC) pour son résultat. Le résultat est envoyé au bloc de fetch pour lire l'instruction suivante mais aussi au bloc Issue pour lui renseigner la nouvelle valeur du PC actuel.

II.1.7 Les contrôleurs asynchrones

Les contrôleurs asynchrones sont identiques pour tout les blocs définis précédemment, ils envoient un signal d'enable à leurs blocs respectifs au moment approprié pour imiter une horloge mettant à jours leurs entrées. Ils sont conçus pour communiquer entre eux en suivant le protocole décrit dans la section I.3.

Ils utilisent une porte appelée élément C ou porte de Muller dont la table de vérité est décrite dans la Fig. II.3.



F	R	Z
0	0	*0*
0	1	*Memory*
1	0	*Memory*
1	1	*1*

FIGURE II.3 – Porte de Muller

Étant donné que la sortie d'une porte de Muller dépend de sa valeur précédente, cette sortie doit être initialisée pour que le contrôleur puisse fonctionner. Nous avons donc ajouté une troisième entrée à nos portes de Muller appelée reset initialisant la sortie à zéro. Ce signal de reset est injecté dans toutes les portes de Muller du circuit au démarrage du processeur.

Un contrôleur asynchrone a la structure illustrée dans la Fig. II.4.

Lorsqu'un contrôleur reçoit la requête *req_in* et un bus de données, il n'envoie le signal d'enable (et donc n'enregistre ces entrées) que si il ne reçoit pas d'acquittement venant du bloc suivant (*ack_out* entrée inversée de la porte de Muller), ce qui signifie qu'il est prêt à

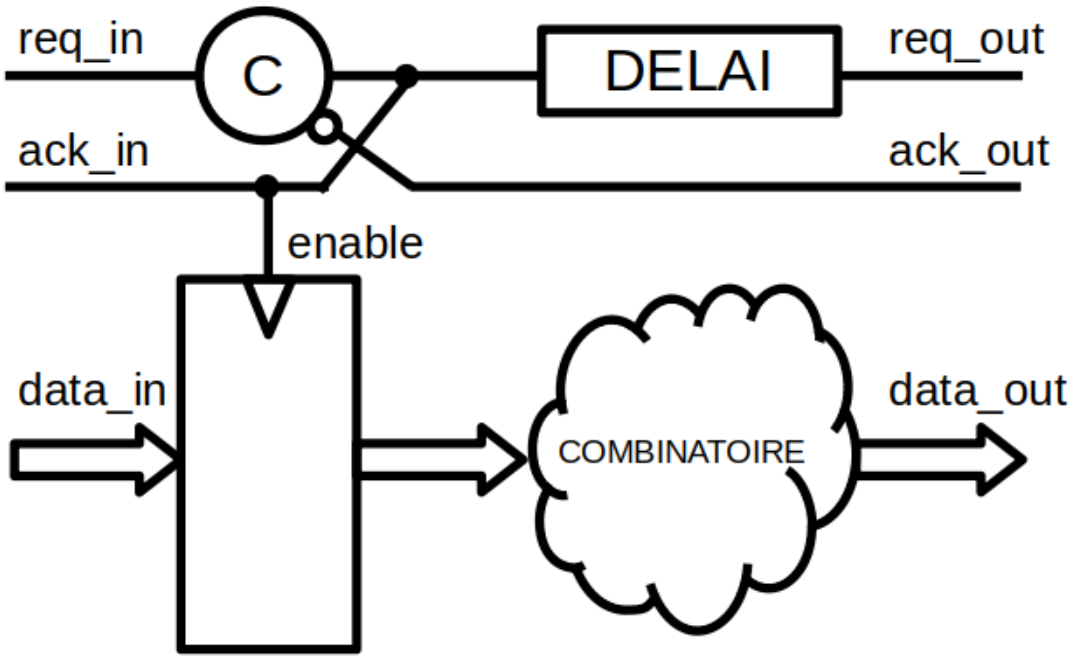


FIGURE II.4 – Contrôleur asynchrone

les recevoir.

Une fois les nouvelles données reçues, il envoi son signal d'acquittement *ack_in* conformément au protocole. Il envois aussi un signal de requête au bloc suivant *req_out* qui doit traverser un bloc de délai afin que le bloc suivant ne reçoive la requête qu'une fois les opérations combinatoires terminées. Grâce à la porte de Muller, ce signal reste actif tant que le signal d'acquittement du bloc suivant *ack_out* n'est pas activé, conformément au protocole.

II.1.8 Les forks et les joins

Le système de contrôleurs décrit précédemment permet une propagation linéaire des données, d'un bloc à l'autre, mais certains blocs doivent envoyer des informations à plusieurs blocs différents ou en recevoir de plusieurs blocs en même temps. On a donc besoin des blocs fork et join.

Le bloc fork permet de dupliquer un signal de requête sortant d'un bloc vers plusieurs autres blocs, et de renvoyer l'acquittement à l'émetteur qu'une fois tout les signaux d'acquittement des récepteurs activés.

Inversement, le bloc join permet d'envoyer un signal de requête au récepteur qu'une fois

tout les signaux de requête des émetteurs activés et de dupliquer le signal d'acquittement sortant du récepteur vers les émetteurs.

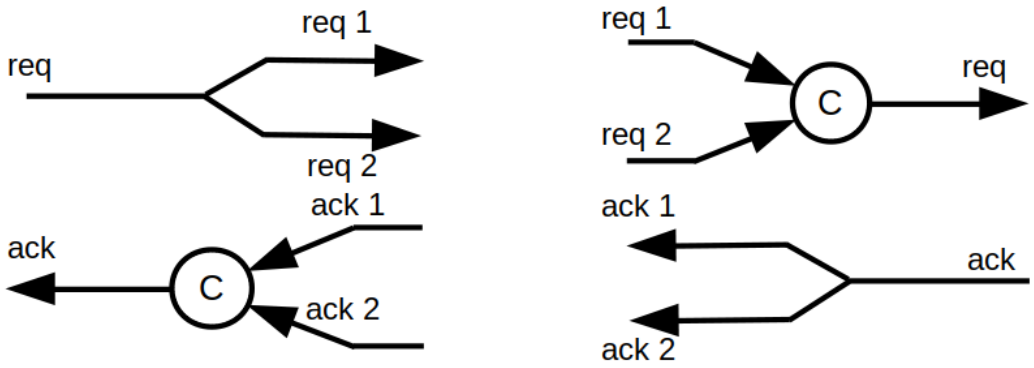


FIGURE II.5 – Fork et Join

On peut voir sur la figure II.5 les schémas des blocs fork et join à deux entrées ou sorties, mais ce schéma est facilement généralisable pour un nombre d'entrées ou de sorties quelconque.

II.2 Difficultés de mise en œuvre

II.2.1 Boot

Le bloc boot recopie son entrée sur sa sortie quand il reçoit son signal d'enable. Il peut sembler inutile mais il remplie en réalité 2 fonctions :

Premièrement, comme nous le décrirons dans la section II.3, la structure globale du core est une grande boucle de contrôleurs, mais il y a aussi une plus petite boucle entre *pc_alu* et Issue, car Issue renseigne la valeur actuelle du pc à *pc_alu*, qui en retour communique directement la nouvelle valeur calculée. On aurait donc une boucle de deux contrôleurs communiquant directement l'un à l'autre, ce qui ne peut pas fonctionner avec un protocole de handshake à 4 phases. Nous avons donc introduit le bloc boot dans cette boucle, maintenant de longueur 3, la rendant fonctionnelle.

Secondement, pour démarrer l'ensemble du core, la valeur du PC doit être initialisée et une première requête doit être émise.

Pour initialiser la valeur du PC, nous devons injecter une adresse de boot dans le bloc *pc_alu*, l'envoyant en sortie si le signal de reset est activé.

En suite, la première requête doit prendre la forme d'une impulsion envoyée en entrée du bloc précédant le fork vers Fetch et Issue, afin que ce bloc maintienne cette requête vers Fetch et Issue jusqu'à leurs acquittements.

Cependant, envoyer une requête en entrée de *pc_alu* effaçait instantanément la valeur initiale de boot. Grâce à l'ajout du bloc boot, l'impulsion de démarrage peut être envoyée en entrée de ce dernier pour lancer les requêtes suivantes, tout en gardant la valeur de boot injectée dans *pc_alu* par le reset.

II.3 Structure globale

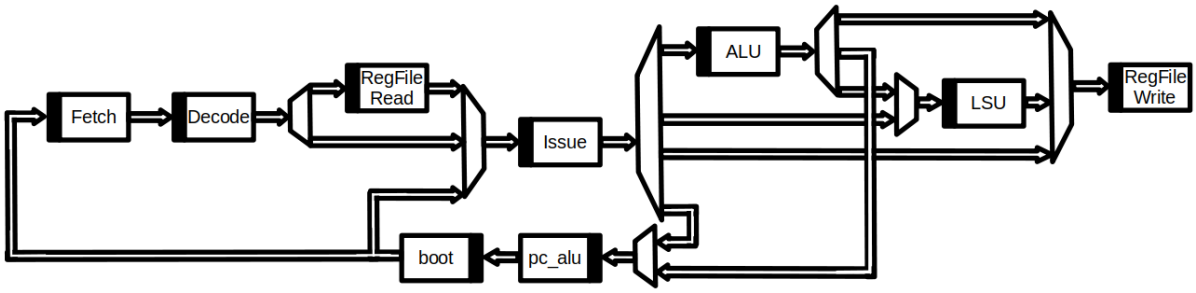


FIGURE II.6 – Chemin de donnée du processeur

La figure II.6 montre le chemin effectué par les données, et donc aussi le chemin des requêtes (et acquittements en sens inverse) entre les contrôleurs. On y retrouve tous les blocs décrits individuellement dans la section II.1. Ce chemin est donc une boucle entre la recherche d'une instruction, son décodage, son exécution et la mise à jour du PC. La valeur calculée du nouveau pc est aussi directement bouclée sur le bloc Issue. On peut voir qu'une instruction passe par 7 blocs pour être effectuée, c'est donc un pipeline à 7 étages.

III Conception

III.1 Flot de conception

Une conception de cette taille requiert un flot défini qui a déjà été discuté par le passé en cours. Nous n'avons pas réussi à aller jusqu'à la synthèse mais nous pouvons cependant suggérer une démarche à partir de ce qu'il reste à faire.

Comme pour tout projet, la première partie est destinée à comprendre le sujet et sa problématique comme expliqué dans le chapitre I d'introduction. C'est au cours de ces recherches que nous avons pris connaissance des documents de la bibliographie.

Nous avons ensuite découper le processeur en blocs fonctionnels, à peu près indépendants, pour pouvoir les définir en System Verilog. Ces blocs ont tous été testés séparément avec des bancs de test personnalisés.

Une fois la vérification des blocs passée, nous avons commencé la partie asynchrone du projet en créant d'abord les briques puis les blocs nécessaires. Ce sont la Muller's Gate, les forks et les joins ainsi que le contrôleur asynchrone exposé Sec. II.1.7.

Puis nous avons conçu un circuit de contrôle fonctionnel pour tout les blocs avec une gestion du démarrage. Le problème rencontré dans cette partie est discuté dans la Sec. II.2.1. l'objectif étant ici de tracer le chemin des requêtes pour retrouver le circuit établi.

Enfin nous avons accroché les contrôleurs aux blocs. Pour vérifier le fonctionnement du core dans sa globalité nous avons créé une mémoire d'instruction artificielle avec un programme à exécuter.

Voila tous ce que nous avons réussi à faire, cependant nous pouvons prévoir l'adaptation du code pour la synthèse. La synthèse devrait probablement être faite par étape.

Après la synthèse, nous devrions avoir une idée de la surface, consommation, mais surtout du chemin critique. Dans les contrôleurs, pour le fonctionnement du core il faut régler le délai pour les requêtes de sorties. Ce réglage doit être fait à partir du résultat de la synthèse.

IV Résultats

IV.1 Évaluation du circuit

Comme exposé dans la Sec. III.1, nous avons effectué des tests sur tout les blocs séparément avant de les assembler pour construire le circuit final.

IV.1.1 Banc de test accès mémoire

Pour tester la LSU et la mémoire on utilise un banc de test simpliste qui veut charger en mémoire un mot et le lire par la suite. Pour ce faire nous avons créer une mémoire de test qui répond aux requêtes par des acquittements. Dans le rendu vous trouverez le code du banc de test en entier et on peut voir en annexe B.3 la structure de la mémoire de test.

On voit dans la Fig. IV.1 le chronogramme du test décris ci-dessus avec en sortie 1 qui est la valeur d'entrée. Les entrées data sont pilotés par lsu, on met ceci en avant avec un délai dans les requêtes et idem pour les sorties.

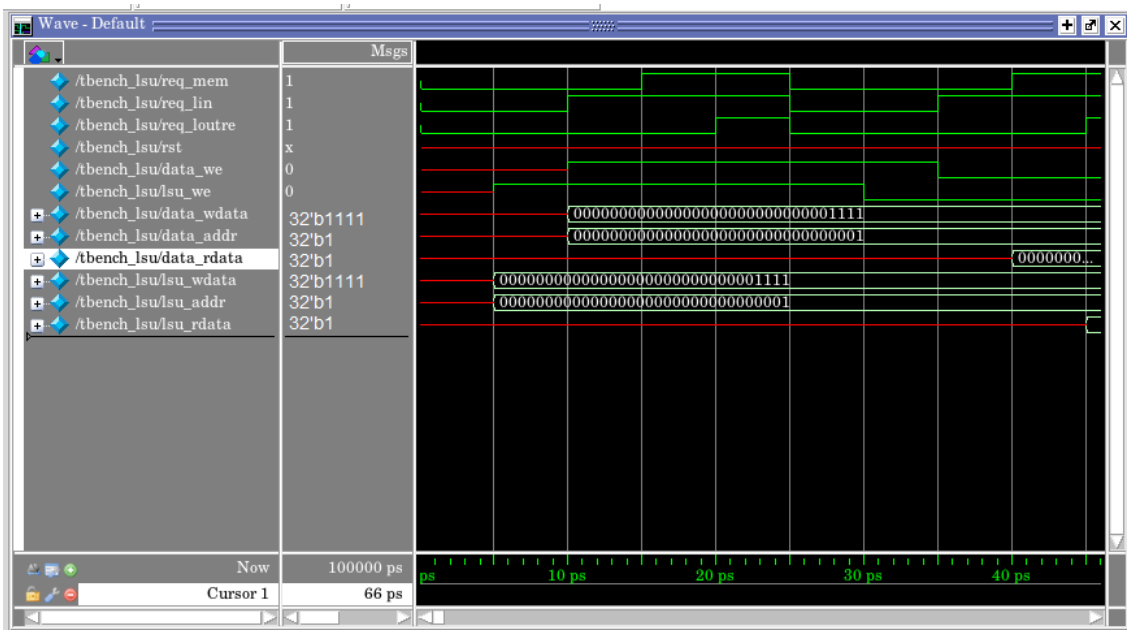


FIGURE IV.1 – Chronogramme LSU

IV.1.2 Banc de test RF

Pour tester l'écriture et la lecture dans les registres, on initialise tout les registres avec zéro et on écris deux valeurs dans deux registres différents. On peut ensuite lire les valeurs en simultané.

IV.1.3 Banc de test Decode et Issue

Après avoir vérifié le fonctionnement de Decode avec quelques instructions de type différents, nous avons décidé d'accrocher la sortie du Decode directement à issue en ajoutant les entrées supplémentaires (RF, PC). Nous pouvons donc directement envoyer des instructions en entrée et observer la sortie de Issue.

Nous avons décider de faire de cette façon car issue prend trop d'entrée pour les contrôler manuellement avec moins d'erreurs que decode. Le code utilisé est dans l'annexe B.4 et le chronogramme obtenu est en annexe C.1.

IV.1.4 ALU et PC ALU

Pour l'ALU et PC ALU, nous avons fait un banc de test exhaustif avec les opérateurs et types d'opérandes possibles. Les fichiers des bancs de test sont inclus dans le rendu et le chronogramme résultants du test ALU est en annexe C.2.

IV.1.5 Contrôleur Asynchrone

Nous avons tout d'abord testé un élément C simpliste sans reset (cf. Sec. II.1.7).

Puis nous avons essayé la communication entre deux contrôleurs dans un test lui aussi simplifié. On peut voir dans la Fig. IV.2 le résultat du banc de test, dont le code est en annexe B.5. le test consiste en l'observation de la réaction de la chaîne non-bouclée à une requête en entrée.

Ensuite, nous avons fait tourner le circuit asynchrone seul mais bouclé avec une séquence de démarrage simplifiée. Le résultat est un enchaînement de requêtes rapides qui montre le système bouclé sans soucis. Pour pouvoir mieux observer les cycles un contrôleur à un très long délai devant les autres (150 ns pour PC ALU contre 5 ns pour les autres).

Le chronogramme n'est pas vraiment informatif donc pas inclus dans ce document.

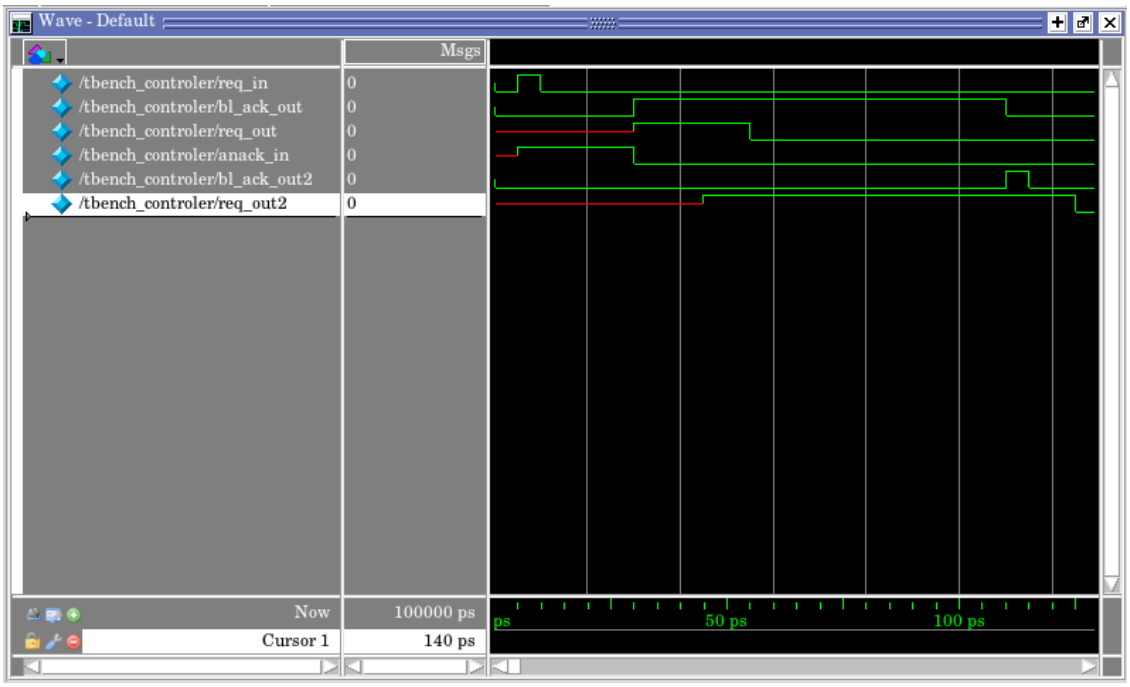


FIGURE IV.2 – Chronogramme deux Contrôleurs

IV.1.6 Asmodee

Nous avons finalement pu effectuer un test du core en entier avec un programme en mémoire. Le code de la mémoire artificielle est en annexe B.6 avec un offset de boot = 256. Le programme fait successivement deux additions avec 0 sur 2 registres différents pour charger une valeur dans le registre, c'est donc une opération registre immédiat ; puis un XOR entre les deux registres. Le chronogramme est illustré dans la Fig. IV.3. On trouve dans la mémoire des registres 1, 2 et 3 respectivement les valeurs 16, 21 et 5, ce qui correspond bien aux résultats attendus.

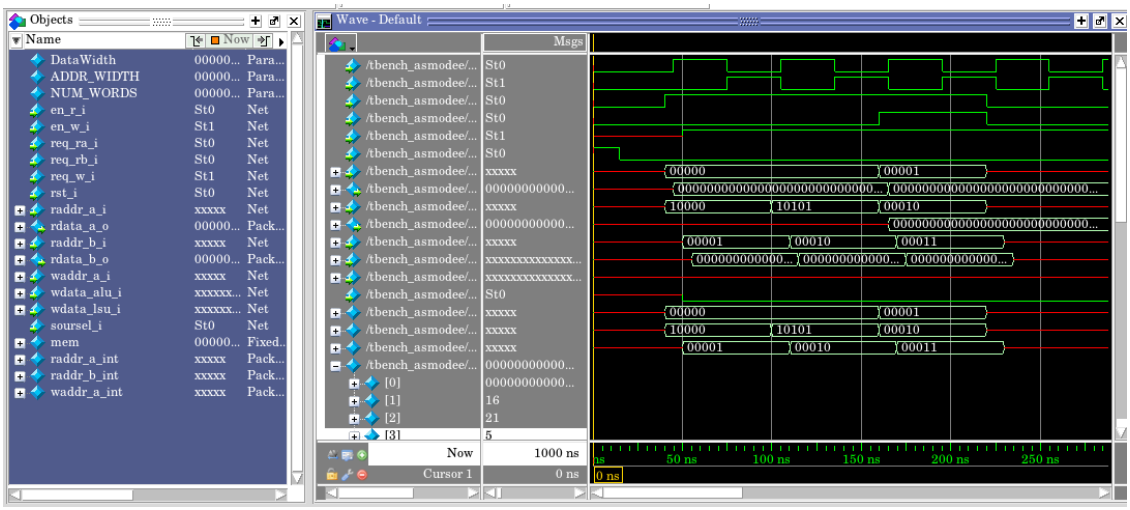


FIGURE IV.3 – Chronogramme programme

V Conclusion

Pour conclure ce document, nous allons mettre en avant ce que nous avons réussi à faire par rapport au sujet présenté dans l'introduction, mais aussi les perspectives pour améliorer ce projet et en faire un produit fini.

Comme nous l'avions exposé dans l'introduction, l'objectif du projet était de mettre en avant un concept. Nous avons une preuve de concept avec ASModee. Nous pouvons a peu près exécuter les instructions de RV32I et un programme qui les utilisent, c'est donc un processeur RISC V. De plus aucune horloge n'est nécessaire pour l'exécution du programme, le circuit de contrôle asynchrone suffit pour l'exécution après la séquence de démarrage, c'est donc un processeur asynchrone. Cependant, le core est loin d'être fini.

En plus de toute les extensions possibles sur un RISC V, comme des ALUs supplémentaires pour des opérations plus diverses (p. ex. MULT/DIV, Float) ou la gestion d'autres données que des mots, nous avons encore beaucoup à faire pour avoir un core véritablement utilisable. Le core est limité à l'exécution seule, ce qui le rend difficilement debugable (Debug mode) et peu transparent (CSR). De plus, pour optimiser le core on pourrait traiter les informations comprimées et avoir une queue FIFO dans Fetch, comme implémentée par l'Ibex.

Pour le circuit asynchrone nous avons décidé d'utiliser des Forks et des Joins, ce qui est plus simple à implémenter mais consomme plus que des Splits et Merges.

N'ayant pas fait la synthèse, nous n'avons pas vraiment d'idée de la consommation et ni de la surface du core.

Malgré tout nous avons montré la simplicité et la flexibilité des core asynchrone et RISC. Nous avons aussi appris à mieux connaître le flot de conception numérique et la vérification avec ModelSim. Nous avons sans doutes développés des compétences utiles pour notre futur d'ingénieur numéricien et la suite de nos études.

Annexes

A Ibex

A.1 Architecture

La Fig. A.1 vient du manuel d'utilisateur de l'Ibex [low] et représente la globalité du circuit Ibex.

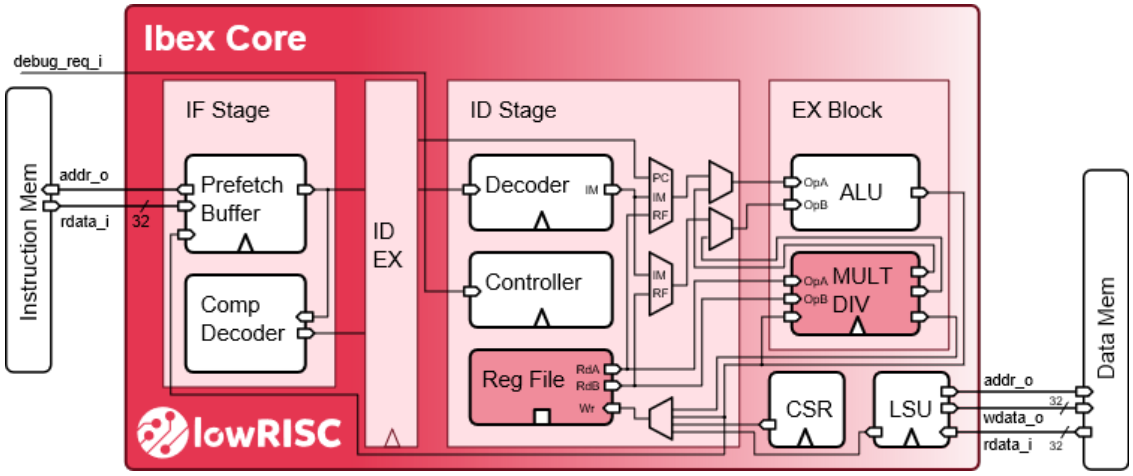


FIGURE A.1 – Diagramme de blocs du core Ibex

B Code source

B.1 Package

```
//define constqnt used in core

package pkg;

int unsigned offset = 256;

typedef enum logic [4:0] {
        // Arithmetics
         ADD,
         SUB,

        // Logics
         XOR,
         OR,
         AND,
        // RV32B
        /*
         XNOR,
         ORN,
         ANDN,
        */

        // Shifts
         SRA,
         SRL,
         SLL,
        // RV32B
        /*
         SRO,
         SLO,
```

```
 ROR,
 ROL,
 REV,
 REV8,
 ORCB,
*/

// Comparisons
 LT,
 LTU,
 GE,
 GEU,
 EQ,
 NE,
// RV32B
/*
 MIN,
 MINU,
 MAX,
 MAXU,
*/

/*
// Pack
// RV32B
 PACK,
 PACKU,
 PACKH,


// Bitcounting
// RV32B
 CLZ,
 CTZ,
```

```
        PCNT,

        */
        // Set lower than
        SLT,
        SLTU
        /*

        // Ternary Bitmanip Operations
        // RV32B
        CMOV,
        CMIX,
        FSL,
        FSR,

        // Single-Bit Operations
        // RV32B
        SBSET,
        SBCLR,
        SBINV,
        SBEXT
        */
} alu_op;

typedef enum logic [1:0] {
        JAL,
        JALR,
        BRANCH
} pc_op;

typedef enum logic [6:0] {
        OPCODE_LOAD     = 7'h03,
        OPCODE_MISC_MEM = 7'h0f,
        OPCODE_OP_IMM   = 7'h13,
```

```
        OPCODE_AUIPC    = 7'h17, //PC ALU
        OPCODE_STORE    = 7'h23,
        OPCODE_OP       = 7'h33,
        OPCODE_LUI      = 7'h37,
        OPCODE_BRANCH   = 7'h63, //PC ALU
        OPCODE_JALR     = 7'h67, //PC ALU
        OPCODE_JAL      = 7'h6f, //PC ALU
        OPCODE_SYSTEM   = 7'h73
} opcode;

// Regfile write data selection
//used with CSR
/*
typedef enum logic {
  RF_WD_EX,
  RF_WD_CSR
} rf_wd_sel_e;
*/

typedef enum logic [1:0] {
        OP_A_REG,
        OP_A_IMM,
        OP_A_CURRPC
} op_a_sel;

typedef enum logic {
        OP_B_REG,
        OP_B_IMM
} op_b_sel;

typedef enum logic [2:0] {
        IMM_B_I,
        IMM_B_S,
        IMM_B_B,
```

```
        IMM_B_U, //upper
        IMM_B_J, //jump
        IMM_B_O, //offset (boot address)
        IMM_B_N  //incr (length(instr)=4)
} imm_b_sel;

endpackage
```

B.2 Controleur Asynchrone

```
module controler #(
        parameter DELAY = 5
) (
        input  logic rst_i,

        input  logic req_in_i,
        input  logic ack_out_i,

        output logic req_out_o,
        output logic ack_in_o
);

timeunit  1ns;
timeprecision 1ns;

        c_element muller(
         .rst(rst_i),
         .a(req_in_i),
         .b(~ack_out_i),
         .c(ack_in_o)
        );

        assign #DELAY req_out_o =  ack_in_o;
```

```
endmodule
```

B.3 Mémoire de test

```
module tmem(
        //control signal
        input  logic                req_i,

        input  logic             data_we_i,
        input  logic [31:0] data_wdata_i,
        input  logic [31:0] data_addr_i,

        output logic [31:0] data_rdata_o
);

reg [31:0] mem [7999:0];

always_ff @(posedge(req_i)) begin
        if(data_we_i) begin
                mem[data_addr_i] <= data_wdata_i;
        end
        else begin
                data_rdata_o <= mem[data_addr_i];
        end
end
endmodule
```

B.4 Testbench - Issue

```
//OUTDATED avec l'ajout des controleurs asynchrone
// I / O plus correspondant, signaux enable

//Test de issue qui redirige les flux sortant de decode vers les unités suivantes

module tbench_issue;
```

```
    import pkg::*;

    logic [31:0] rf_rdata_a, rf_rdata_b, pc_rdata, instruction;
    logic [31:0] imm_i_type, imm_s_type, imm_b_type, imm_u_type,
    imm_j_type, imm_o_type, imm_n_type;
    logic req_issue, req_decode, rst, req_pc_alu, req_alu_i,
    req_alu_o, bb, req_data, data_req, we_data, data_we, ss, req_w;
    logic req_rf_ra, req_rf_rb;
    pkg::op_a_sel type_operand_a; pkg::op_b_sel type_operand_b;
    pkg::imm_b_sel type_imm_b;
    pkg::alu_op operateur_alu, operateur_alu_o;
    pkg::pc_op operateur_pc_alu;

    logic [4:0] waddr_i, waddr_o, rf_raddr_a, rf_raddr_b;

    logic [31:0] operand_alu_a, operand_alu_b, operand_pc_alu_a,
    operand_pc_alu_b, lsu_wdata;

issue issue_m (
  .req_i            (req_issue),
  .rst_ni           (rst),

  .rf_rdata_a_i     (rf_rdata_a),
  .rf_rdata_b_i     (rf_rdata_b),
  .pc_rdata_i       (pc_rdata),

  .imm_i_type_i     (imm_i_type),
  .imm_s_type_i     (imm_s_type),
  .imm_b_type_i     (imm_b_type),
  .imm_u_type_i     (imm_u_type),
  .imm_j_type_i     (imm_j_type),
  .imm_o_type_i     (imm_o_type),
  .imm_n_type_i     (imm_n_type),
```

```
.type_operand_a_i (type_operand_a),
.type_operand_b_i (type_operand_b),
.type_imm_b_i     (type_imm_b),

.req_alu_i        (req_alu_i),
.req_alu_o        (req_alu_o),

.operateur_alu_i(operateur_alu),
.operand_alu_a_o(operand_alu_a),
.operand_alu_b_o(operand_alu_b),
.operateur_alu_o(operateur_alu_o),

        //To PC_ALU
.req_pc_alu_i(req_pc_alu),
.operateur_pc_alu_i(operateur_pc_alu),



.operand_pc_alu_a_o(operand_pc_alu_a),
.operand_pc_alu_b_o(operand_pc_alu_b),
.branch_bool_o(bb),

        //To LSU
.data_req_i(req_data),
.data_req_o(data_req),// start transaction to data memory

.data_we_i(we_data),
.data_we_o(data_we),// write enable

.lsu_wdata_o(lsu_wdata),
    //output logic [31:0], lsu_addr_o, //adresse caluculer dans l'ALU

    //To RF write
.rf_waddr_i(waddr_i),
```

```
   .rf_waddr_o(waddr_o),
   .rf_soursel_o(ss),//0 for ALU and 1 for LSU
   .req_rf_w_o(req_w)
);

decode deco(
        .req_i(req_decode),
  .rst_ni(rst),

  //from IF
 .instr_rdata_i(instruction),

  //immediates
  .imm_i_type_o(imm_i_type),
  .imm_s_type_o(imm_s_type),
  .imm_b_type_o(imm_b_type),
  .imm_u_type_o(imm_u_type),
  .imm_j_type_o(imm_j_type),
  .imm_o_type_o(imm_o_type),
  .imm_n_type_o(imm_n_type),

      //Decode / RF
      //req
      .req_rf_ra_o(req_rf_ra),
      .req_rf_rb_o(req_rf_rb),
      //source register address
  .rf_raddr_a_o(rf_raddr_a),
  .rf_raddr_b_o(rf_raddr_b),

      //destination register addr
      .rf_waddr_o(waddr_i),

  // LSU
  .req_data_o(req_data),// start transaction to data memory
```

```
    .we_data_o(we_data),// write enable

        .req_pc_alu_o(req_pc_alu),
        .operateur_pc_alu_o(operateur_pc_alu),

        .req_alu_o(req_alu_i),
        .operateur_alu_o(operateur_alu),

        .type_operand_a_o(type_operand_a),
        .type_operand_b_o(type_operand_b),
        .type_imm_b_o(type_imm_b)
  );

//req_decode, rst, instruction, rf_rdata_a, rf_rdata_b, pc_rdata, req_issue

        initial begin
                #5 req_decode = 0; rst = 0; req_issue = 0;
                instruction = 32'b0000000_00001_10000_000_01010_0110011;
                #5 req_decode = 1;
                rf_rdata_a  = 32'b00000000000011000000010100110011;
                rf_rdata_b  = 32'b00000000000011000000010100000000;
                pc_rdata        = 32'b00000000000000000000000100000000;
                #5 req_issue = 1;

                #5 req_decode = 0; rst = 0; req_issue = 0;
                instruction = 32'b0000000_00001_10000_100_01010_0110011;
                #5 req_decode = 1;
                rf_rdata_a  = 32'b00000000000011000000010100110011;
                rf_rdata_b  = 32'b00000000000011000000010100000000;
                pc_rdata        = 32'b00000000000000000000000100000000;
                #5 req_issue = 1;

                #5 req_decode = 0; rst = 0; req_issue = 0;
                instruction = 32'b???????_?????_?????_???_?????_1101111;
```

```
                #5 req_decode = 1;
                rf_rdata_a  = 32'b00000000000011000000010100110011;
                rf_rdata_b  = 32'b00000000000011000000010100000000;
                pc_rdata        = 32'b00000000000000000000000100000000;
                #5 req_issue = 1;

                #5 req_decode = 0; rst = 0; req_issue = 0;
                instruction = 32'b???????_?????_?????_010_?????_0100011;
                #5 req_decode = 1;
                rf_rdata_a  = 32'b00000000000011000000010100110011;
                rf_rdata_b  = 32'b00000000000011000000010100000000;
                pc_rdata        = 32'b00000000000000000000000100000000;
                #5 req_issue = 1;

                #5 req_decode = 0; rst = 0; req_issue = 0;
                instruction = 32'b0100000_?????_?????_101_?????_0110011;
                #5 req_decode = 1;
                rf_rdata_a  = 32'b00000000000011000000010100110011;
                rf_rdata_b  = 32'b00000000000011000000010100000000;
                pc_rdata        = 32'b00000000000000000000000100000000;
                #5 req_issue = 1;

                #5 req_decode = 0; rst = 0; req_issue = 0;
                instruction = 32'b???????_?????_?????_000_?????_1100011;
                #5 req_decode = 1;
                rf_rdata_a  = 32'b00000000000011000000010100110011;
                rf_rdata_b  = 32'b00000000000011000000010100000000;
                pc_rdata        = 32'b00000000000000000000000100000000;
                #5 req_issue = 1;
        end
endmodule
```

B.5 Testbench - 2 Contrôleurs

```
//Test communication entre 2 controler asynchrone et reglage des délais

module tbench_controler;

timeunit  1ns;
timeprecision 1ns;
        logic rst;
        logic req_in, ack_out, req_out, ack_in, ack_out2, req_out2;

        controler ctl (
                .rst_i(rst),

                .req_in_i(req_in),
                .ack_out_i(ack_out),

                .req_out_o(req_out),
                .ack_in_o(ack_in)
        );

        defparam ctl.DELAY = 25;

        controler ctl2 (
                .rst_i(rst),

                .req_in_i(req_out),
                .ack_out_i(ack_out2),

                .req_out_o(req_out2),
                .ack_in_o(ack_out)
        );

        defparam ctl2.DELAY = 15;
```

```
        initial begin
                rst = 1;

                req_in = 0; ack_out2 = 0; ack_out = 0;
                #5 req_in = 1;

                #5 req_in = 0;
                #100 ack_out2 = 1;

                #5 ack_out2 = 0;

                rst = 0;

                req_in = 0; ack_out2 = 0; ack_out = 0;
                #5 req_in = 1;

                #5 req_in = 0;
                #100 ack_out2 = 1;

                #5 ack_out2 = 0;
        end

endmodule
```

B.6 Programme de test

```
module tinsmem(
        //control signal
        input  logic                 req_i,

        input  logic [31:0] instr_addr_i,

        output logic [31:0] instr_rdata_o
```

```
);

reg [31:0] mem [127:0];

always_ff @(posedge(req_i)) begin
        instr_rdata_o <= mem[instr_addr_i/4];
end

// DEBUG //
//Chargement en mémoire
assign mem[64] = 32'b0000000_10000_00000_000_00001_0010011; //ADD r0+16 -> r1
assign mem[65] = 32'b0000000_10101_00000_000_00010_0010011; //ADD r0+21 -> r2
assign mem[66] = 32'b0000000_00010_00001_100_00011_0110011; //XOR r1,r2 -> r3
endmodule
```

C Chronogrammes

C.1 Issue

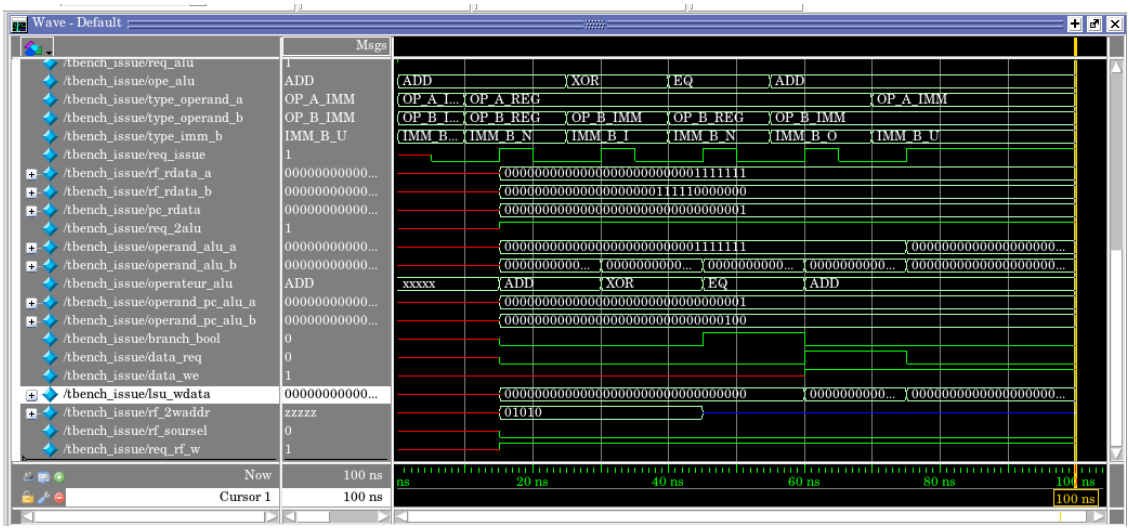


FIGURE C.1 – Chronogramme du testbench Issue

C.2 ALU

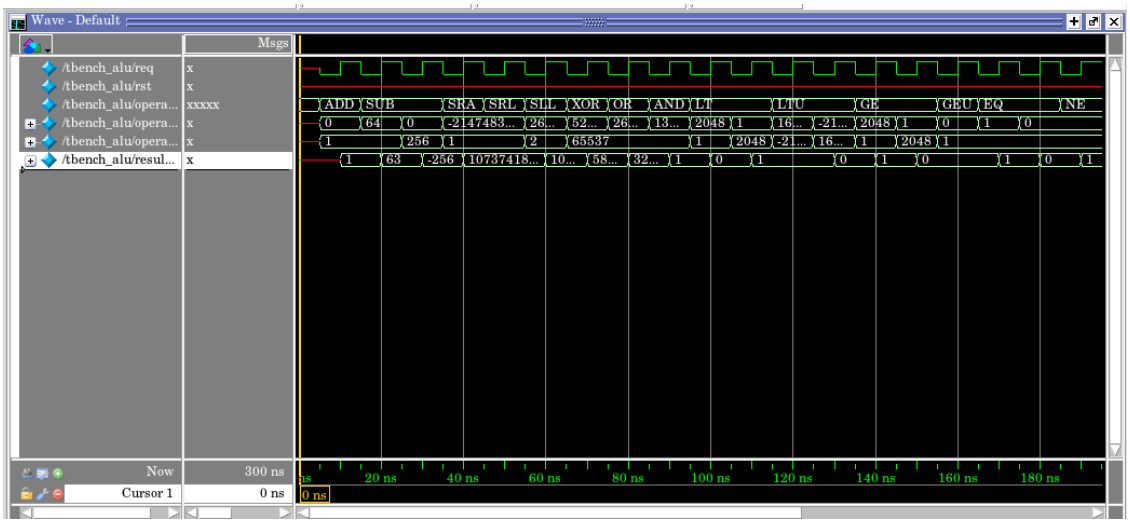


FIGURE C.2 – Chronogramme du testbench ALU

Biblographie

[FES+20] L. FESQUET, Y. DECOUDU, R. IGA et R. A. GUAZZELLI. *For the Asynchronologists. (Fr) [Circuit, Device and System Integration Group - TIMA Laboratory]*. Rapp. tech. 2020.

[low] lowRISC. *Ibex User Manual*. URL : `https://ibex-core.readthedocs.io/en/latest/index.html`.

[Spa06] J. Sparsø. *Asynchronous Circuit Design. A Tutorial*. 2006. URL : `http://www.imm.dtu.dk/pubdb/views/publication_details.php?id=855`.

[WA19] A. Waterman et K. Asanovi´c. *The RISC-V Instruction Set Manual. Volume I : Unprivileged ISA*. Version 20191213. December 13, 2019. URL : `https://riscv.org/specifications/isa-spec-pdf/`.