SETR-ME2 Langages matériels, réutilisation et intégration

Compte rendu mini-projet

Louison Gouy et Téo Biton
November 25, 2022

École Polytech de Nantes
Electronique et Technologies Numériques

Enseignants référents : Sébastien PILLEMENT et Maria MENDEZ REAL

Abstract

En se basant sur une IP de cellule RAM en VHDL, il s'agit de construire un bloc mémoire générique avec un décodeur pour accéder à un nombre de RAM à définir. Ensuite, ce bloc mémoire doit être connecté à un modèle de processeur simple dans le but d'exécuter un programme assembleur. L'objectif final est d'avoir un top module le plus générique possible et qu'il soit synthétisable.

Contents

List of Figures

1 Introduction

L'objectif de ce mini-projet est de concevoir un système à base de microprocesseur en réutilisant des blocs de composants de base. La première partie du TP consiste à reprendre le code d'une RAM simple décrite en VHDL en la rendant générique. Ensuite, il faut créer un bloc contenant plusieurs instances de cette RAM et un décodeur pour y accéder. Enfin, la dernière partie du projet consiste à intégrer ce bloc mémoire à un système comprenant un processeur décrit en Verilog.

Ce TP a pour objectifs d'initier les étudiants aux notions de généricité et de réutilisation d'Intereltual Property (IP), tout en leur permettant de renouer avec le VHDL et de découvrir le Verilog. De plus, ils seront amenés à intégrer les différents composants entre eux de sorte à produire un module global utilisable.

Pour mener à bien ce TP, nous avons fait le choix d'une approche atypique : pour limiter les contraintes liées à l'utilisation de *ModelSim* sur les machines de l'école, nous avons travaillé avec des outils de compilation et de simulation Open Source disponibles sur nos machines. Ainsi, nous avons utilisé *ghdl 2.0.0* pour la compilation et la simulation des codes VHDL et *gtkwave 3.3.104* pour l'affichage. De plus, pour élargir notre champs de compétences au delà des attendus du TP, nous avons défini dès le départ des modules ou langages à utiliser pour les bancs de tests. Pour compiler et simuler les codes VHDL, nous avons utilisé *Cocotb*, librairie Python permettant de tester et simuler des codes en langages de description matérielle. Pour simuler le code en *Verilog* du processeur, nous avons défini des bancs de test en C++. Enfin, pour simuler le top module intégrant le bloc processeur et le bloc mémoire, nous avons utilisé *ModelSim*, moins contraignant pour simuler un système à langages mixtes. L'élaboration RTL à quant à elle été réalisée sous *Vivado*. Cette approche peut paraitre contreproductive dans un cadre industriel mais, dans ce contexte pédagogique, se familiariser avec différents outils prend plus de sens.

Pour chaque étape, les choix technologiques ainsi que des preuves de fonctionnement en simulation seront montrées. Les codes sources sont fournis dans une archive annexe et dans le projet GitHub https://github.com/loulou4418/Reuse_indegration.

2 Bloc mémoire

Cette première partie traite de la génération du bloc mémoire fonctionnel. L'objectif est d'obtenir un bloc contenant un nombre générique de mémoires ainsi qu'un décodeur pour en gérer l'accès, via des signaux `Chip Select`.

Comme indiqué lors de l'introduction, pour chaque sous module du bloc mémoire, des bancs de test ont été réalisé grâce à *Cocotb*, *"COroutine based COsimulation TestBench"*, un environnement de vérification VHDL et SystemVerilog basé sur le langage Python. C'est un outil open source que nous avons couplé à ghdl pour la compilation du VHDL et à gtkwave pour afficher les résultats. Il a été spécifiquement pensé pour réduire le temps d'écriture de bancs de tests et prône la réutilisation de designs, philosophie appropriée pour le module ME2.

Note: Pour la vue Register Transfer Logic (RTL), les paramètres génériques définis étaient les suivants:

$$\begin{cases} ABUSSIZE = 12 \\ DBUSSIZE = 32 \\ NBMSB = 2 \end{cases}$$

Les figures montrant les schémas RTL auront donc des valeurs en accordance avec ces paramètres.

2.1 Récupération d'une IP de RAM

Il nous a été fourni un modèle simple de mémoire Random Access Memory (RAM). La première tâche vise à comprendre son fonctionnement puis à trouver une solution afin de rendre son implémentation générique tant sur la taille des données que sur l'adresse. Le modèle récupéré génère une RAM par un tableau de vecteurs d'une taille variable selon la taille du bus d'adresses.

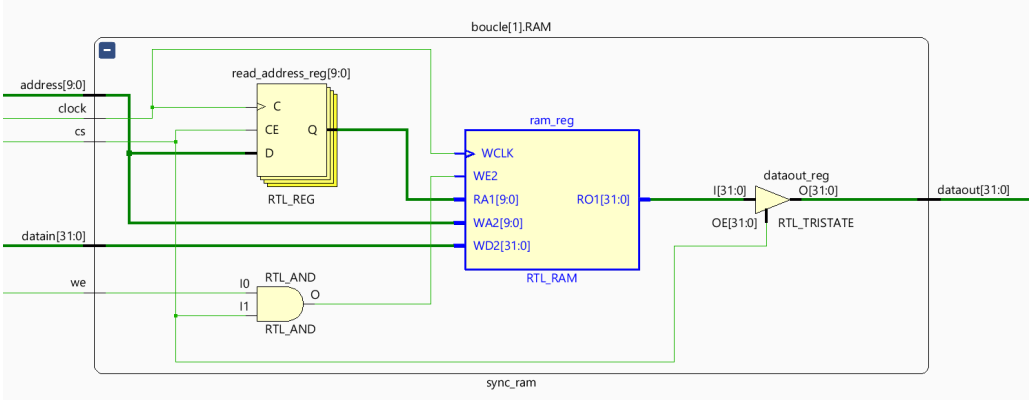

Figure 1: Bloc RAM niveau RTL

En figure 1, nous pouvons voir le bloc RAM synchrone avec les bus de données et d'adresse en entrée. De plus, des bascules D en parallèle permettent de sauvegarder l'adresse de lecture correspondant au signal `read_address`. Une porte logique AND permet de vérifier la condition sur `cs` et `we` pour l'écriture en mémoire. Enfin, une porte tri-state affecte la valeur au bus de sortie selon l'état dy signal `cs`. Après modification du modèle, la lecture et l'écriture sur le bus de données en sortie n'ont lieu que lorsque le signal `Chip Select (cs)` est à l'état haut. Nous avons pu générer un banc de test avec *Cocotb* qui consistent à tester l'écriture via le signal `Write Enable (we)` et l'activation selon le signal `cs`.

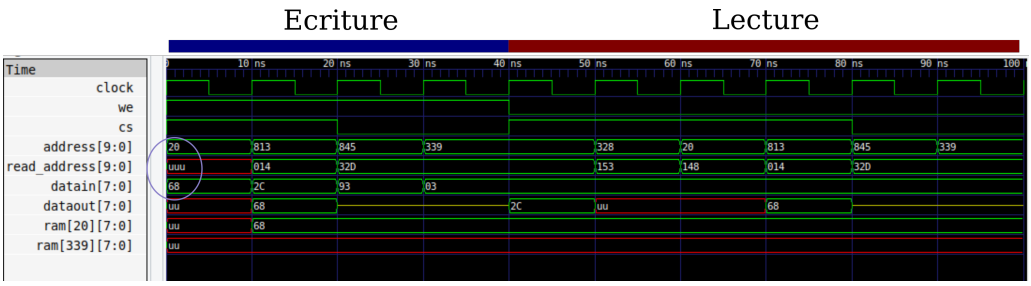

Figure 2: Simulation du bloc mémoire

Sur la figure ci-dessus, nous pouvons vérifier le comportement de la mémoire. Dans un premier temps, des valeurs sont écrites en mémoire (`we` = 1) avec des variations du signal `cs` pour vérifier si les écritures ont lieu ou non. Dans un deuxième temps, des cycles de lecture s'enchaînent aux mêmes adresses pour vérifier le contenu de la RAM. En jouant également sur la valeur du signal `cs`, nous pouvons voir que la sortie est indéfinie lorsque `cs` = 0, que la valeur contenue en adresse 20 de la RAM est correctement lue, et que l'adress 339 est indéfinie puisque `cs` n'était pas actif lors de l'écriture.

2.2 Décodeur d'adresse

En vue de pouvoir intégrer plusieurs instances de la RAM, il est nécessaire de décrire un décodeur d'adresses qui analysera les n-bits de poids fort (n pour la généricité) pour accéder aux différentes RAM en lecture ou en écriture.

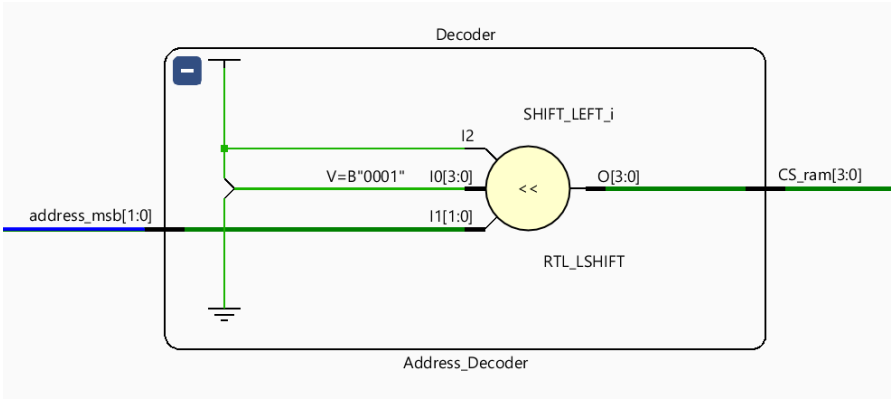

Figure 3: Décodeur au niveau RTL

Le décodeur fonctionne de façon purement combinatoire : le fonctionnement interne est simplement un décalage à gauche d'un bus de taille 2^n et de valeur initiale 1; le nombre de décalages effectués est égal à la valeur des bits d'entrée et il y a 2^n valeurs possibles. Le banc de test est élémentaire : il suffit de tester les 2^n entrées possibles, n étant le nombre de bits de poids fort pris en considération. Via l'utilisation de *Cocotb*, il est aisé de vérifier les sorties pour des n différents (ci-dessous, les tests pour n = 2 et n = 3). La sortie est affichée directement dans le terminal.

```
Decoder entry size: 2
addr_msb: 00 ; cs_ram = 0001
addr_msb: 01 ; cs_ram = 0010
addr_msb: 10 ; cs_ram = 0100
addr_msb: 11 ; cs_ram = 1000
```

```
Decoder entry size: 3
addr_msb: 000 ; cs_ram = 00000001
addr_msb: 001 ; cs_ram = 00000010
addr_msb: 010 ; cs_ram = 00000100
addr_msb: 011 ; cs_ram = 00001000
addr_msb: 100 ; cs_ram = 00010000
addr_msb: 101 ; cs_ram = 00100000
addr_msb: 110 ; cs_ram = 01000000
addr_msb: 111 ; cs_ram = 10000000
```

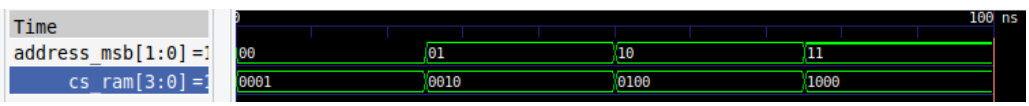

Figure 4: Simulation du décodeur pour 2 bits de poids fort en entrée

2.3 Bloc Quad-ram

Le bloc Quadram est le top module du bloc mémoire. Il ne contient en réalité que des générations d'instances de RAM et du décodeur, toujours d'une façon générique suivant la taille du bus d'adresse et du bus de données. Il n'est pas possible de déterminer manuellement le nombre de blocs RAM instancié, celui-ci est en effet calculé suivant le nombre de bits de poids forts pris en compte : pour n-bits, il y aura 2^n cellules RAM. Cette implémentation est choisie car calculer un log de 2 avec des composants matériels rajoutent une complexité non souhaitée à l'application. Les bits de poids fort pris en entrée du décodeur ne sont pas redirigés vers la cellule RAM par la suite, ainsi le bus d'adresse réel pris en entrée par la RAM est de `abus_size - nb_msb`.

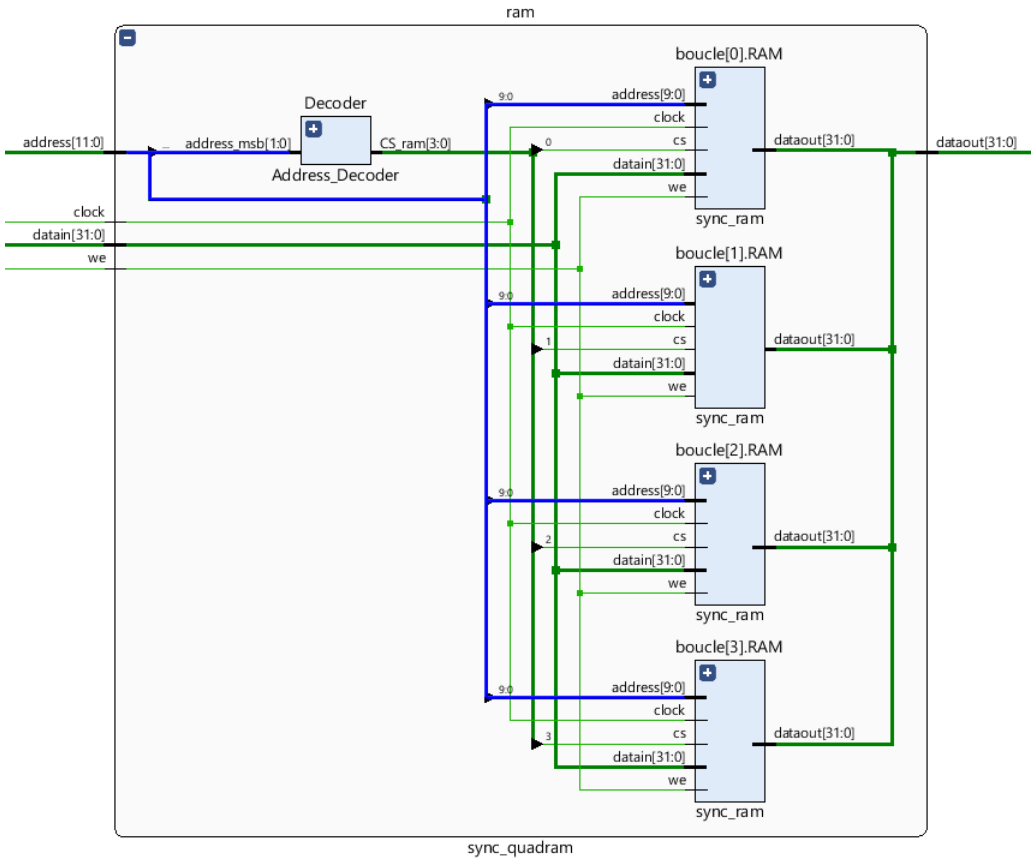

Figure 5: Bloc mémoire complet niveau RTL

Le décodeur prend en entrée un nombre spécifié en paramètres de bits; ici, 2 bits en entrée. Il y a donc un bus `CS_ram` en sortie qui est décomposé vers 4 cellules RAM. Le reste des bits du bus d'adresses ainsi que le bus de données sont en entrées des cellules RAM. Nous avons donc pu générer un banc de tests avec *Cocotb* qui consiste à faire un cycle de 5 écritures à des adresses aléatoires en mémoire puis de faire 5 cycles de lecture à ces mêmes adresses pour vérifier le contenu des blocs RAM et le fonctionnement des signaux `Chip Select`.

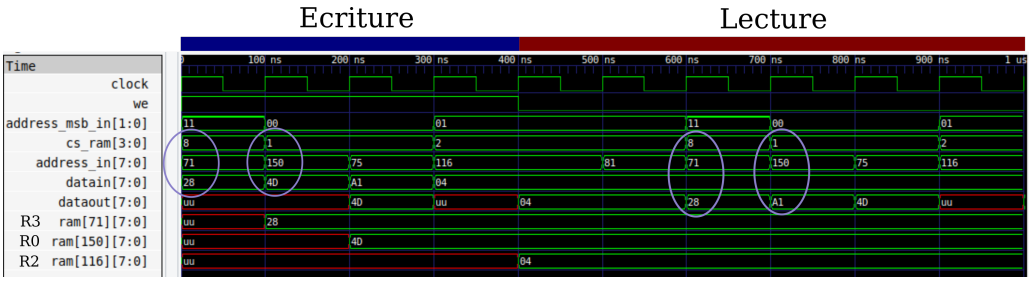

Figure 6: Simulation du bloc mémoire complet

Comme pour tester le comportement d'une cellule RAM, 5 cycles d'écritures sont effectués puis 5 cycles de lecture. Sur la figure ci-dessus, les valeurs contenues aux adresses des RAMs correspondent à l'écriture qui a eu lieu au cycle d'horloge précédent. Pour les lecture, le bon fonctionnement est moins évident puisqu'en effet, à 700 ns, la valeur lue à l'adresse 150 de la RAM0 n'est pas 4D comme attendu, mais A1; ceci est dû à la sauvegarde de `read_address` qui lors de la dernière activation de la RAM0, avait pour valeur 75. Ainsi, au cycle d'horloge suivant la valeur 4D est lue à l'adresse de lecture 150. Ce cycle d'horloge supplémentaire vient de notre décision d'affecter la valeur `address` à `read_address` uniquement lorsque le signal `cs` est à l'état haut; nous trouvions illogique d'avoir une affectation permanente alors que les adresses peuvent être destinées qu'à une seule RAM à chaque cycle. Les valeurs indéfinies sur le bus de sortie correspondent à une lecture à une adresse où aucune valeur n'a été écrite.

Le nom donné par défaut au bloc est "Quadram" mais en réalité le modèle pourrait en contenir 1, 2, 8, 16 ...

3 Modèle de processeur

Cette partie vise à présenter les modifications du processeur afin d'assembler un système complet avec les blocs RAM. Pour tester la mémoire réalisée dans la partie précédente, le sujet fourni imagine un modèle de processeur élémentaire. Cela permet à la fois de contextualiser le mini-projet tout en faisant appel aux connaissances d'architecture des ordinateurs vue les années passées. Un modèle de processeur appelé `simple proc` est donné aux étudiants sous forme d'un code *Verilog*. Il est composé d'une mémoire interne pour stocker les instructions et données, mais ne comporte aucune entrée sortie. Il n'est donc pas synthétisable, mais uniquement simulable. L'objectif est de faire les modifications nécessaires pour assurer la compatibilité avec les blocs RAM. Cela passe par l'ajout d'un certain nombre de signaux et par la modification de l'Instruction Set Architecture (ISA).

Comme indiqué dans l'introduction, le logiciel *Verilator* a été utilisé pour la simulation. Sa particularité est de transformer la description *Verilog* en *C++*. Le testbench peut être écrit en *C++* offrant ainsi une vue plus haut niveau appréciée pour le test. Le projet est largement utilisé dans le monde académique comme dans l'industrie. On peut ainsi voir RISC-V Foundation, Microchip, NXP, Intel parmi ses utilisateurs.

3.1 Modification de l'architecture

La première modification à apporter est l'ajout d'entrées/sorties au processeur. Cela doit permettre la connexion avec le bloc quadram. Comme une entité (entity) en VHDL, le mot clé `module` permet d'adopter une vue externe. On ajoute alors deux entrées clock (clk) et reset (nRst) pour la synchronisation. Nous disposons dès lors d'une horloge, les instructions non synthétisable `step` peuvent être retirées après avoir rendu synchrone le process principal. La ligne *Verilog* suivante permet cela.

```
1    always @ (posedge clk)
```

Il a été choisi précédemment de dupliquer le bus de données selon le sens de communication pour ne pas gérer la complexité induite par la bidirectionalité. Deux tableaux de bits `datain`, `dataout` sont définis un en entrée et l'autre en sortie. Leur taille est passée en paramètre pour garantir la généricité. Finalement, le bus d'adresse en déclaré en sortie car, seul le processeur en a le contrôle. Sa taille est, elle aussi, variable. Le signal `we` pour "write enable" est utile pour contrôler la RAM est déclarée comme sortie.

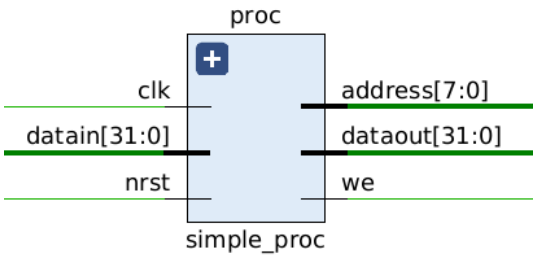

Figure 7: Entité simple proc

Après l'analyse RTL, la vue externe obtenue figure 7 est bien celle attendue. Tous les signaux définis sont présents. Les paramètres pour la longueur des bus de données et d'adresse sont respectivement 32 et 12 dans cet exemple.

La seconde modification est nécessaire en raison du changement de mémoire. Le processeur est initialement conçu avec une mémoire interne. La contrainte concernant l'usage d'une mémoire externe rend l'accès à la donnée impossible pour les opérations arithmétiques. Pour palier à cela, la solution est l'ajout de registres tampons d'une longueur égale à celle du bus de données. Leur politique de gestion est explicitement laissée libre dans le sujet. Les choix effectués par les étudiants concepteurs ne concernent pas la déclaration des registres, mais plutôt l'ISA. C'est pourquoi cette question sera abordée dans la partie suivante.

Le signal reset est choisi comme étant actif à l'état bas, son activation a pour effet de mettre à zéro les registres PC, IR, SR, reg_A, reg_B et le signal we.

3.2 Modification de l'ISA

Une ISA est définie comme étant un modèle abstrait d'un ordinateur. Elle spécifie les instructions, les registres, les types de données et les fondamentaux de la gestion mémoire. L'ISA ne décrite pas l'architecture d'un Central Processing Unit (CPU) mais sert à sa conception. Elle garantit la compatibilité au niveau binaire de deux implémentations [3]. Dans notre cas, l'ISA n'est pas proprement définit, c'est à nous de l'adapter pour répondre au comportement demandé.
Précédemment, une solution a été implémentée pour palier à l'inaccessibilité des données par les instructions arithmétiques. Elle consiste à ajouter deux registres tampons A et B. Il existe différentes philosophies de processeur parmi elles les plus connues, Reduce Instruction Set Computer (RISC) et Complex Instruction Set Computer (CISC). Elles sont différentes sur plusieurs points. Pour ce qui nous intéresse ici, les architecture RISC sont plus simple et exécute les instructions en un seul coup d'horloge. Les processeurs CISC sont plus complexes et disposent d'instructions plus puissantes. Étudions cela à travers l'addition. En x86 (CISC), l'instruction `add <reg>,<mem>` peut prendre en opérande un registre et une adresse mémoire [2]. En langage ARM assembleur, `add Rd, Rm` ne peut se faire qu'entre deux registres [1]. Il faudra donc charger (LDR) les deux valeurs dans des registres avant d'effectuer une addition. Pour des questions de simplicité, mais aussi car l'architecture RISC sont particulièrement intéressantes pour l'embarqué, comme le montre la popularité de ARM et RISC-V, nous choisirons la philosophie RISC.
Il convient alors de modifier les instructions arithmétiques. Elles accèderont seulement aux registres. La règle générale est alors $A = A \quad o \quad B$ ou les lettres A et B représentent les registres et le symbole o les opérations arithmétiques telles que l'addition, la soustraction et la multiplication. Le résultat du complément est également stocké dans A. Le code *Verilog* dans le cas de l'addition est le suivant :

```
1 `ADD:
2 begin
3     reg_A = reg_A + reg_B;
4     setcondcode(reg_A);
5 end
```

L'appel à la fonction `setcondcode()` met à jour l'état du registre SR avec le résultat du calcul.

Dans l'état actuel du processeur, il n'est pas possible d'affecter une valeur aux registres A et B. Pour palier à cela, deux instructions "load" sont ajoutées LDA et LDB. Il existe une subtilité quant à leur mise en place. Contrairement à la mémoire interne, la RAM n'est pas accessible immédiatement. D'abord l'adresse est placée sur son bus puis, au cycle suivant, la valeur de la case mémoire est placée sur le bus de données. Cela écarte une solution naïve qui viserait à écrire l'adresse et récolter la donnée de manière synchrone au premier front. Il faudrait plutôt affecter la valeur au registre au front d'horloge suivant. Nous avons choisi, en nous basant sur le processeur précédemment étudié le Z80, d'utiliser un signal `t_wait`. Il

permet d'ajouter un cycle horloge au cycle machine qui était initialement systématiquement égaux. Lorsque `t_wait` vaut 1, le registre `PC` n'est pas incrémenté, et `IR` maintenu à sa valeur précédente. Le chronogramme commenté figure 8 présente les résultats obtenus après

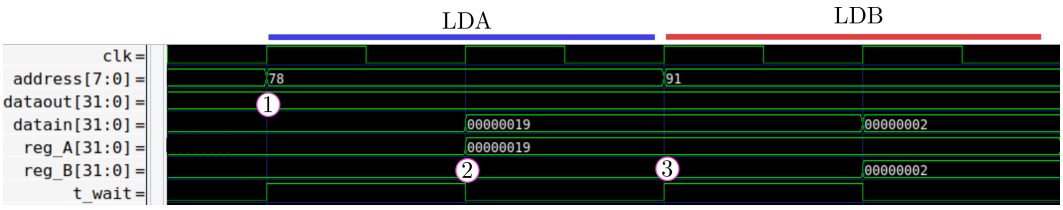

Figure 8: Chronogramme test LDA et LDB avec `t_wait`

simulation. Un testbench permet de charger la mémoire avec les instructions `LDA 0x78` et `LDB 0x91`. Il fournit ensuite une réponse semblable au comportement de la RAM en plaçant au second front une valeur sur le bus de données. A l'instant du label 1, l'instruction `LDA 0x78` est exécutée, 0x78 est placé sur le bus d'adresse et `t_wait` passe à l'état haut. Puis à l'instant 2, lors du second front, la donnée 0x19 est fournie et affectée au registre A. Enfin, à l'instant 3, la valeur 0x19 est maintenue dans le registre. L'instruction de chargement (load) a fonctionné. Le même raisonnement s'applique pour `LDB`. La solution présentée, visant à allonger les instructions de chargement d'un cycle, est simple à mettre en œuvre. Elle a cependant le désavantage évident de multiplier par deux le temps d'exécution.
Cette stratégie peut être appliquée de la même façon à l'instruction de stockage (`STR`). Le même signal `t_wait` est mis à l'état haut afin de figer le processeur pendant un coup d'horloge. Les bus conservent alors leur état et la mémoire peut assimiler la valeur. On propose de présenter la validation par le résultat du testbench. Dans le testbench en *C++*, une structure est définie correspondant à chaque instruction. La fine séparation fournie par les *bit field* du langage C permet d'assigner une taille à chauqe champs.

```
typedef struct {
    uint16_t BB : 12;
    uint16_t AA : 12;
    uint8_t RESERVED : 3;
    uint8_t IM : 1;
    uint8_t OPCODE : 4;
} _STR_t;
```

La ligne `STR.field.IM = 1;` ci-dessous, correspond ainsi à l'affectation d'un seul et unique bit à l'emplacement souhaité. Cela facilite l'écriture et évite de passer par la notation binaire ou hexadécimale.

```
STR.field.IM = 1;
STR.field.AA = 0xAF; /* idenfiable random data value */
STR.field.BB = 0x81; /* address */
dut->simple_proc__DOT__MEM[0] = STR.val; /* set memory */
```

La dernière ligne de l'extrait ci-dessus réalise l'initiation de la mémoire interne. La première instruction, à l'index [0] est donc un `STR`.

```
/* assert STR test when IR reg == STR */
if (dut->simple_proc__DOT__IR == STR.val && dut->clk == 1){
    if ((dut->address == 0x81) && (dut->dataout == 0xAF)){
      printf("Success asserting STR test 1 \n");
    }
    else{
      printf("STR failed. Addr:%d data:%d \n", dut->address, dut->dataout);
    }
}
```

Le test pour vérifier la conformité de l'instruction s'attache à contrôler l'état du bus d'adresse et de donnée sachant l'instruction encodée. Le résultat du terminal est alors :

```
    Success asserting STR test 1
```

Une seconde confirmation avec les chronogrammes est effectuée. Elle confirme le premier diagnostique.

Un testbench en *C++* est écrit pour chacune des instructions LDA, LDB, STR et ADD. Cela peut paraitre redondant, mais on s'assure ainsi qu'une modification sur une instruction n'impacte pas les autres. Nous sommes satisfaits d'avoir suivi cette méthode, car cela nous a gagné un temps précieux dans l'étape suivante combinant RAM et processeur.

4 Système complet: processeur et mémoires

Le système complet, ou top module, instancie une Quadram décrite en VHDL et le processeur décrit en Verilog. Le top module a une horloge et le signal de reset en logique négative et en sortie, le bus de données. Son comportement est décrit en VHDL; un composant est défini pour faire appel au module *simple_proc* correspondant au processeur. Pour conserver la généricité, il est possible de définir la taille du bus de données et du bus d'adresses, ainsi que le nombre de bits de poids fort à isoler pour le décodeur d'adresses.

4.1 Simulation

Via Modelsim, il est possible de charger des valeurs hexadécimales dans la mémoire *MEM* du processeur: ces valeurs sur 4 octets sont des instructions définies dans le jeu d'instructions du processeur. En simulation, à chaque coup d'horloge, une instruction est chargée, décodée et executée. L'exécution séquentielle de ces instructions permet de tester le fonctionnement global du système. L'approche choisie est de stocker des valeurs *immediate* contenue dans un champs de l'instruction *Store* en mémoire puis de charger ces valeurs dans les registres tampons du processeur. Pour le test, des opérations mathématiques sont effectuées sur ces valeurs: une addition puis une multiplication, le tout en faisant intervenir différentes cellules RAM de la mémoire. Pour les tests, les paramètres ont été définis de la façon suivante:

$$\begin{cases} ABUSSIZE = 10 \\ DBUSSIZE = 32 \\ NBMSB = 2 \end{cases}$$

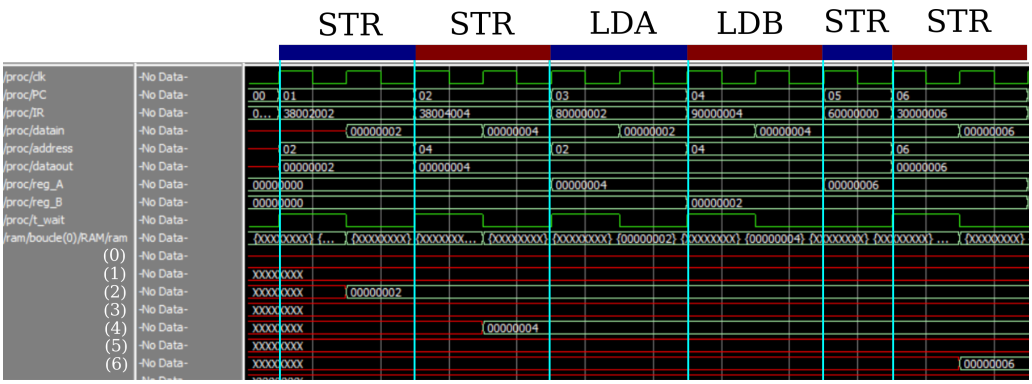

Figure 9: Simulation du top module

Ce chronogramme correspond aux instructions exécutées pour obtenir le résultat de l'addition 2 + 4 en mémoire. Nous pouvons observer les deux coups d'horloge nécessaires pour les instructions *Load* et *Store*, permis par l'activation du signal *wait*. En mémoire, les valeurs apparaissent de manière séquentielle au fur et à mesure que les instructions s'exécutent. Le résultat attendu (2 + 4 = 6) est retrouvé en mémoire à l'adresse souhaitée. Dans le cas du test ci-dessus, une seule mémoire est sollicitée. D'autres tests ont été effectués et valident le comportement du système pour une multiplication de deux valeurs contenues dans des mémoires différentes. Ci-dessous le fichier .txt des instructions chargées en mémoire.

```
38002002 // STR immediate value 2 at address 2 (ram 0)
38004004 // STR immediate value 4 at address 4 (ram 0)
80000002 // LDA value from address 2
90000004 // LDB value from address 4
60000000 // ADD values from registers A and B
30000006 // STR value from register A at address 6 (ram 0)
38003003 // STR immediate value 3 at address 3 (ram 0)
38005084 // STR immediate value 5 at address 4 (ram 2)
80000003 // LDA value from address 3
90000084 // LDB value from address 4
70000000 // MUL values from registers A and B
30000086 // STR value from register A at address 6 (ram 2)
00000000 // HLT
```

4.2 Analyse RTL du module

Tout au long du rapport, des blocs au niveau RTL des différents blocs ont été utilisé à des fins d'illustration. Ces blocs étaient issus de l'analyse RTL du top module réalisée sous Vivado. Avant d'effectuer cette étape, nous avons retiré certains éléments non synthétisables du processeur notamment les commandes *step* qui ajoutaient un offset de 10 ns ou encore les commandes ModelSim. Le résultat montre bien, les différentes entités du projet. On

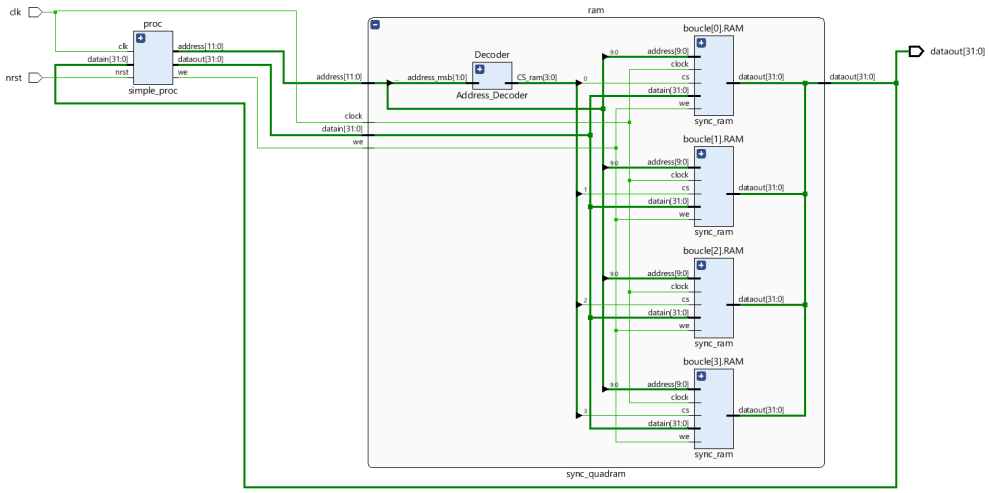

Figure 10: Synthèse du top module

observe le processeur à gauche, le décodeur au centre et les 4 blocs RAM à droite. Les interconnexions et notamment les bus sont particulièrement intéressants à visualiser.

4.3 Vue RTL du processeur

Lorsqu'on double clic sur le processeur de la figure 10 une vue détaillé apparaît. Elle représente, les circuits logiques de ce dernier. On peut alors identifier rapidement, les différents registres matérialisés par des vecteurs de bascules. On constate également un certain nombre de multiplexeurs pour décoder les instructions. Les éléments sont agencés de manière désorganisée il est difficile d'insérer une capture globale pertinente. Il est toutefois possible d'extraire les blocs un à un.

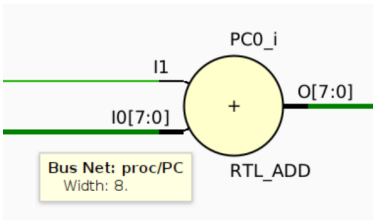

Figure 11: Additionneur de PC vue RTL

La figure 11 présente par exemple, l'additionneur de PC à chacun coup d'horloge. Le registre PC est incrémenté de la constante 1.

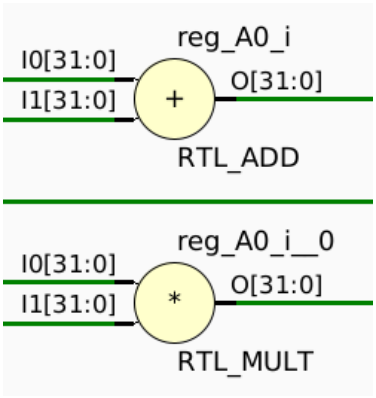

Figure 12: Additionneur et multiplieur vue RTL

Les opérations de l'Arithmetic and Logic Unit (ALU) sont également visibles comme le montre la figure 12. Ils sont connectés aux registres et A est présent à la fois en entrée et en sortie comme souhaité.

5 Conclusion

Pour conclure, ce mini-projet nous a permis de fortement consolider nos compétences en langages de description matérielle, principalement par la découverte du langage Verilog. De plus, c'était un premier pas vers des projets d'intégration à grande échelle où la réutilisation et la généricité des composants est de mise. L'utilisation d'outils open source nous a poussé à être curieux et surtout confronté à des problèmes classiques lors la mise en place d'environnement de développement partagés: gérer correctement les versions des logiciels utilisés et passer par un outil de gestion de versions comme Git.

Valider le fonctionnement global du sytème a été d'une grande satisfaction tout comme parvenir à synthétiser le projet. Le temps investi à formaliser les bancs de test a permis de gagner un temps considérable, notamment lors des phases d'intégration qui n'ont généré que peu de bugs. La prochaine étape aurait été de configurer un FPGA avec un modèle synthétisé et d'y charger des instructions.

Acronyms

ALU Arithmetic and Logic Unit 14

CISC Complex Instruction Set Computer 9

CPU Central Processing Unit 9

IP Intereltual Property 2

ISA Instruction Set Architecture 8, 9

RAM Random Access Memory 3–8, 10, 12, 13

RISC Reduce Instruction Set Computer 9

RTL Register Transfer Logic 3, 13

References

[1] ARM. Developer suite. Technical Report DUI 0068B, 2001.

[2] c9x.me. x86 instruction set reference. 2022.

[3] Wikipedia. Instruction set architecture, November 2022.