Conception et développement d'un processeur à jeu d'instruction réduit RV32I

S. Jovanovic, Y. Berviller, S. Weber
Pôle CNFM MIGREST - Nancy
Institut Jean Lamour (UMR7198), Université de Lorraine
Contact email : slavisa.jovanovic@univ-lorraine.fr

Nous présentons un ensemble de travaux pratiques qui seront dispensés au sein du Master EEA - Électronique Embarquée à l'Université de Lorraine et Télécom Nancy - parcours Logiciel Embarqué, dans le cadre des modules *Conception VLSI* (Master EEA) et *Conception et Développement d'un Système sur Puce* (Télécom Nancy). Ces travaux pratiques portent sur la modélisation de systèmes et circuits en VHDL, et se focalisent en particulier sur le développement incrémental d'une architecture de processeur à jeu d'instructions réduit de type RISC-V (le jeu d'instructions RV32I) en utilisant des suites logicielles de **Mentor Graphics** (*Modelsim* pour la modélisation et simulation VHDL), et de **Intel-Altera** (*Quartus Prime*) pour la validation expérimentale.

I. Introduction

Pour initier les étudiants des formations Master EEA - Électronique Embarquée à l'université de Lorraine et Télécom Nancy - parcours Logiciel Embarqué à la modélisation de systèmes et circuits en VHDL, nous mettons en place une série de travaux pratiques autour des suites logicielles de Mentor Graphics (Modelsim pour la modélisation et simulation VHDL), et de Intel-Altera (Quartus Prime) permettant de simuler, tester, synthétiser et vérifier de manière expérimentale le fonctionnement d'un processeur à jeu d'instructions réduit (le processeur RISC-V à jeu d'instructions ouvert, plus précisément l'ensemble RV32I).

La modélisation de l'architecture RV32I au niveau RTL s'inscrit dans le cadre de deux modules : le module Conception VLSI (Master EEA), où l'accent est mis sur la conception de circuits numériques de calcul au niveau RTL (opérateurs arithmétiques, synthèse logique, placement et routage) de 30 heures, et le module Conception et Développement d'un Système sur Puce (Télécom Nancy) de 44h, où l'objectif principal est de familiariser les étudiants avec les mécanismes de fonctionnement employés au sein d'un processeur et de les mettre en pratique dans le langage de description de matériel VHDL.

Ces travaux pratiques s'adressent aux étudiants de niveau Master 2 et aux élèves de troisième année d'école ayant déjà un bagage de connaissances solide en conception numérique (VHDL, FPGA - au moins 30h) et en conception de circuits microélectroniques de base (60h) avec la suite logicielle Cadence (30h) (pour les étudiants de Master EEA). De plus, les étudiants ont des connaissances en programmation orientée objet (Java, Python, C/C++), en programmation structurelle sur microcontrôleur (langage C) et en programmation C sous Linux (compilation croisée) leur permettant d'assimiler les concepts liés à la modélisation de haut niveau de manière plus aisée. Par conséquent, on peut considérer que les étudiants ont une expérience suffisante et non négligeable pour mener à bien le projet proposé.

Article publié par EDP Sciences et disponible sur le site https://www.j3ea.org ou https://doi.org/10.1051/j3ea/20241012

II. Démarche pédagogique

Les étapes de modélisation, simulation, vérification et test d'un processeur RV32I passent par les étapes suivantes :

- Rappel des éléments combinatoires et séquentiels et l'introduction au jeu d'instructions réduits RISC-V (l'ensemble d'instructions RV32I est uniquement considéré),
- Modélisation, simulation et vérification du jeu d'instruction RV32I de manière progressive : en partant des instructions de type R ; en passant par les instructions de type I (y compris les instructions de type *load*) ; en rajoutant les instructions de stockage dans la mémoire de données (de type S) ; de changement de flot de contrôle (de type B) et en terminant par les instructions de type J et U,
- Vérification de l'architecture globale incorporant l'ensemble d'instructions du jeu RV32I sur un ensemble de tests évolués,
- Test sur FPGA et amélioration des performances de l'architecture initiale en jouant notamment sur le nombre de cycles d'horloge utilisés pour l'exécution de chaque instruction (réduction des chemins critiques identifiés).

Rappel des blocs logiques de base et introduction au jeu d'instructions RV32I

Dans la première étape, l'étudiant est amené à se familiariser avec les mécanismes principaux de fonctionnement d'un processeur et en particulier avec l'architecture RISC-V et son jeu d'instructions réduit.

Dans cette étape, une introduction brève des éléments de base qui seront utilisés pour la conception d'un processeur est effectuée. Il s'agit en particulier de rappeler les éléments de base (combinatoires et séquentielles) nécessaires pour la conception d'un processeur : multiplexeur/démultiplexeur, opérateurs arithmétiques et logiques de base, compteurs/décompteurs, registres (normaux et à décalage), blocs mémoire (ROM et RAM) et machine à états finis.

Dans cette étape, le jeu d'instructions ouvert RISC-V et notamment son sous-ensemble RV32I est introduit. L'étudiant se familiarise avec les différents types d'instructions: les instructions de type R ne travaillant qu'avec des valeurs stockées dans la banque de registres; les instructions de type I où le résultat fourni par l'unité arithmétique et logique (ALU) est calculé en utilisant une donnée issue de la banque de registres et une constante (nommée immédiate et codée sur 12 bits) incorporée directement dans l'instruction programme ; les instructions de chargement ou de type *load* permettant de stocker le résultat de l'ALU dans une case de la mémoire de données (le bloc RAM) - il s'agit du même format que les instructions de type I (considérées comme telles) avec une finalité complètement différente où l'ALU calcule l'adresse de la case mémoire qui sera utilisée (au lieu d'un des registres de la banque de registres) pour stocker le contenu d'un registre de la banque des registres ; les instructions de type S ou de stockage qui permettent de charger une valeur de la mémoire de données dans un des registres de la banque de registres et cela au niveau d'un octet, d'un demi-mot ou de la valeur entière sur 32 bits (en mode signé ou non-signé) ; les instructions de type B qui permettent de gérer le flot d'exécution d'un programme en introduisant les instructions de branchement basées sur la comparaison de deux valeurs stockées dans les registres de la banque des registres ; les instructions de saut ou de type J permettant de sauter sur une adresse calculée ; et finalement les instructions de type U permettant de construire des adresses de saut en se basant sur une constante de 20 bits.

Modélisation, simulation et vérification du jeu d'instruction RV32I de manière progressive

Dans cette deuxième étape, l'étudiant commence à modéliser une première architecture comportant les blocs combinatoires et séquentielles de base nécessaires pour faire exécuter les instructions de type R (1). Dans le jeu d'instructions R, il existe 10 différentes instructions effectuant essentiellement les opérations arithmétiques et logiques (Fig.2.). Parmi les blocs combinatoires et séquentielles, l'étudiant est amené à proposer un compteur programme incrémentant de 4 unités (présenté par le bloc PC), un bloc mémoire de type ROM permettant de stocker les instructions du programme (nommé IMEM en figure 1); une banque de registres comportant 32 registres avec deux entrées en lecture (les entrées R_A et R_B) et une en écriture (l'entrée R_W); ainsi qu'un bloc ALU permettant d'exécuter les opérations arithmétiques et logiques correspondant aux instructions de type R (Fig.1.).

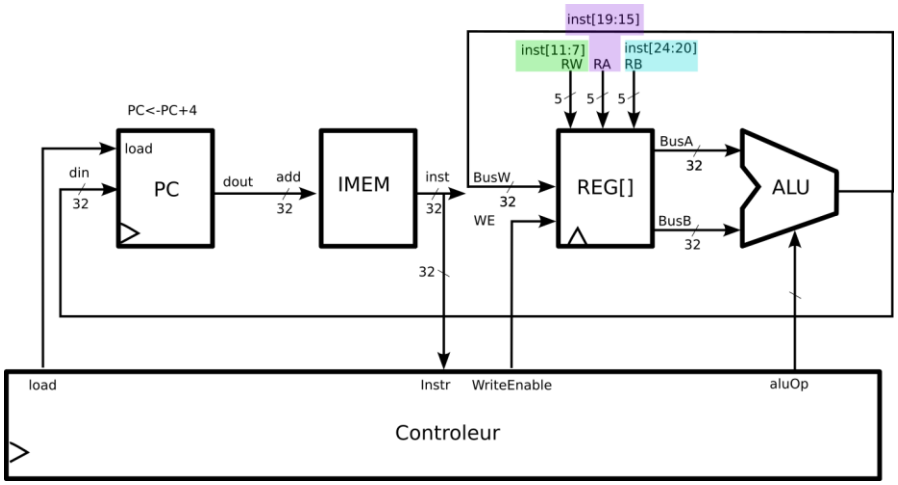

Fig.1. Architecture de base permettant de réaliser le jeu d'instruction de type R

L'exemple d'une description VHDL des blocs combinatoires est donné dans les figures 2 et 3. Sur l'exemple de la banque de registres de taille 32, la figure 2 montre les trois entrées d'adresses RA, RB en lecture avec les sorties correspondantes BusA et BusW, et RW en écriture avec son entrée BusW. L'architecture de cette banque de registres est donnée figure 3. Pour simplifier l'utilisation de cette banque de registres, tous les registres sont initialisés par leurs propres adresses (le registre 1 par la valeur 1, le registre 2 par la valeur 2, et ainsi de suite).

L'étudiant décrit également un bloc nommé Contrôleur représentant le décodeur d'instruction qui dans un premier temps indiquera à l'ALU le type d'opération à exécuter (via la sortie *aluOp*). Ce bloc se chargera également de positionner les valeurs des signaux de contrôle du compteur programme et de l'écriture dans la banque de registre. Ces deux signaux sont positionnés à '0' et à '1' de manière respective puisque dans les instructions de type R l'écriture dans un registre est systématique et le contenu du compteur programme PC reste inchangé par l'extérieur. Il faut également préciser que dans l'architecture proposée figure 1, le bloc mémoire programme IMEM est en lecture seule asynchrone (sans horloge) pour éviter le retard d'un cycle d'horloge au moment de la lecture. La lecture en prenant en compte l'horloge sera introduite dans la dernière étape de conception du processeur.

```
entity regbank is
   generic (
      dataWidth      : integer:=32
   );
   port (
      RA             :  in     std_logic_vector(4 downto 0);
      RB             :  in     std_logic_vector(4 downto 0);
      RW             :  in     std_logic_vector(4 downto 0);
      BusW           :  in     std_logic_vector(dataWidth - 1 downto 0);
      BusA           :  out    std_logic_vector(dataWidth - 1 downto 0);
      BusB           :  out    std_logic_vector(dataWidth - 1 downto 0);
      WE             :  in     std_logic;
      clk            :  in     std_logic;
      reset          :  in     std_logic
   );
end entity regbank;
```

Fig.2. Interface VHDL de la banque de registres de 32 bits.

```
architecture behav of regbank is
   type   regType is array(0to31) of std_logic_vector(dataWidth-1downto 0);
   signal regBank32:  regType;
   constant zero  :std_logic_vector(4 downto 0):=(others=>'0');
begin
   process(clk,reset)
   begin
      if (reset='1') then
         for i in 0 to 31 loop
            regBank32(i)   <=std_logic_vector(to_unsigned(i,dataWidth));
         end loop;
      elsif rising_edge(clk) then
         if WE='1' then
            if RW/=zero then
               regBank32(to_integer(unsigned(RW)))<= BusW;
            end if;
         end if;
      end if;
   end process;
   BusA <= regBank32(to_integer(unsigned(RA)));
   BusB <= regBank32(to_integer(unsigned(RB)));
end behav;
```

Fig.3. L'architecture d'une banque de registres de taille 32 initialisée par les adresses de registres.

31 25	24 20	19 15	14 12	11 7	6 0	bit
funct7	rs2	rs1	funct3	rd	OpCode	R
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

Fig.4. Liste complète des instructions de type R comportant les opérations arithmétiques (addition (*add*), soustraction (*sub*), comparaison (slt et sltu)), logiques (*and*, or, *xor*) et de décalage (sll, slr, sla).

L'architecture proposée figure 1 est essentielle puisqu'elle constitue la base pour tous les autres types d'instructions. Sur cette architecture, l'étudiant effectue les premiers tests en générant le code à charger dans la mémoire programme en utilisant des générateurs automatiques disponibles en ligne (2-3) ou en chargeant dans la mémoire le contenu issu de la compilation croisée d'un code écrit en assembleur RV32I et converti en code machine à charger dans la mémoire ROM. Un exemple de code assembleur utilisé pour tester l'instruction *add* est donné en figure 3 avec le code machine correspondant.

```
1  add  x1,x1,x1        1 001080b3
2  add  x1,x1,x31       2 01f080b3
3  add  x2,x2,x1        3 00110133
4  add  x2,x2,x31       4 01f10133
5  add  x3,x3,x1        5 001181b3
6  add  x3,x3,x31       6 01f181b3
7  add  x4,x4,x1        7 00120233
8  add  x4,x4,x31       8 01f20233
9  add  x5,x5,x1        9 001282b3
10 add  x5,x5,x31      10 01f282b3
11 add  x6,x6,x1       11 00130333
12 add  x6,x6,x31      12 01f30333
13 add  x7,x7,x1       13 001383b3
14 add  x7,x7,x31      14 01f383b3
15 add  x8,x8,x1       15 00140433
```

Fig.5. Exemple de code assembleur et code machine correspondant pour tester l'instruction *add*.

Les résultats de simulation du code présenté figure 5 sont donnés figures 6 et 7 sur l'exemple d'exécution d'une instruction de type R. L'instruction de type R annotée est l'instruction *addi* x2, x2, x1 qui prend les contenus des registres x2 et x1, les additionne et stocke le résultat à nouveau dans le registre x2.

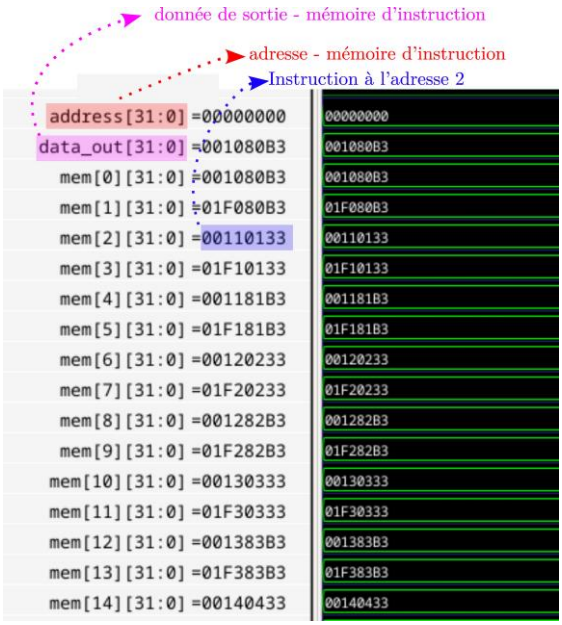

Fig.6. Contenu de la mémoire programme pour le test de l'instruction *add* de type R.

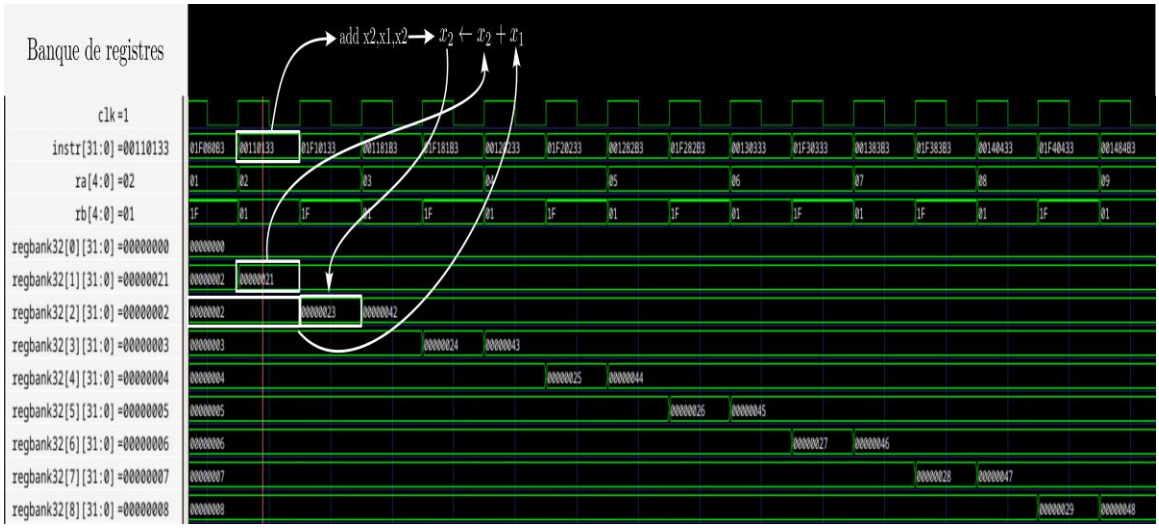

Fig.7. Un exemple d'exécution d'une instruction de type R (*add*) avec le contenu de la banque de registres.

En se basant sur l'architecture présentée figure 1, l'étudiant introduit progressivement les modifications nécessaires à l'architecture initiale pour pouvoir exécuter d'autres types d'instructions. La figure 8 montre l'évolution progressive de l'architecture avant d'arriver à l'architecture finale permettant d'exécuter toutes les instructions de l'ensemble RV32I.

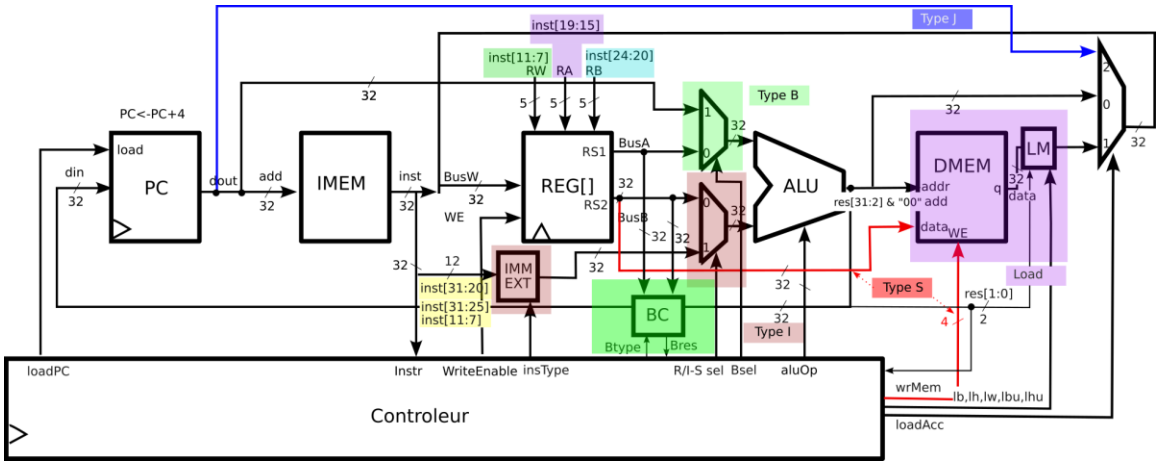

Fig.8. Architecture de base (en noir) augmentée (en couleurs) pour implémenter le jeu d'instruction RV32I.

Vérification d'architecture globale

La vérification de l'architecture se fait également de manière progressive, en fonction de l'évolution de l'architecture initiale présentée figure 1. Ainsi, l'étudiant vérifie dans un premier temps l'architecture au niveau du type d'instruction observé (R, I ou autre) et en combinant les types d'instructions déjà rajoutés dans les phases précédentes avec le tout dernier. L'objectif est non seulement de valider les nouvelles instructions mais également de montrer la validité des instructions déjà rajoutées dans les étapes précédentes. Avec un jeu d'instructions de plus en plus complet, il est également possible de créer des vecteurs de tests de plus en plus évolués. A titre d'exemple, avec les instructions de type I, il est possible de modifier très facilement le contenu d'un registre (utilisation de l'instruction *addi* - *addi* x10, x0,13 pour écrire par exemple la valeur 13 dans le registre x10) sans être obligé d'initialiser la banque de registres dans le code VHDL. Des exemples d'instructions de type I (*load*) sont donnés figure 9.

```
1 addi x5,x0,0
2 addi x7,x0,8
3 lw   x1,0(x5)
4 lw   x2,0(x5)
5 lw   x3,0(x5)
6 lw   x4,0(x5)
7 lw   x1,0(x7)
8 lw   x2,0(x7)
9 lw   x3,0(x7)
10 lw  x4,0(x7)
11 lw  x5,0(x7)
```

Fig.9. Les instructions de type I permettant de charger des valeurs arbitraires sur 32 bits sans passer par l'initialisation dans le code VHDL.

Dans cette étape de vérification, les étudiants se familiarisent non seulement avec l'écriture des premiers tests de base, mais également avec des tests beaucoup plus élaborés en assembleur RV32I. Une fois un code en assembleur écrit, il est très facile de le convertir en code machine en utilisant la chaîne de compilation croisée RV32I :

```
riscv32-unknown-elf-as -march=rv32i add.s
riscv32-unknown-elf-objdump -l -M numeric,no-aliases -d -EL a.out > add.hex
```

où la première commande compile le fichier assembleur *add.s* et génère un fichier de sortie (*a.out*), et la seconde le transforme en un fichier texte (nommé *add.hex)* contenant le code machine à charger dans la mémoire ROM. Le contenu à charger dans la mémoire ROM doit être extrait du fichier *add.hex*, car il contient également d'autres informations comme illustré figure 10.

```
1
2 a.out:     file format elf32-littleriscv
3
4
5 Disassembly of section .text:
6
7 00000000 <.text>:
8    0:   001080b3                add     x1,x1,x1
9    4:   01f080b3                add     x1,x1,x31
10   8:   00110133                add     x2,x2,x1
11   c:   01f10133                add     x2,x2,x31
12  10:   001181b3                add     x3,x3,x1
13  14:   01f181b3                add     x3,x3,x31
14  18:   00120233                add     x4,x4,x1
15  1c:   01f20233                add     x4,x4,x31
16  20:   001282b3                add     x5,x5,x1
17  24:   01f282b3                add     x5,x5,x31
18  28:   00130333                add     x6,x6,x1
19  2c:   01f30333                add     x6,x6,x31
20  30:   001383b3                add     x7,x7,x1
21  34:   01f383b3                add     x7,x7,x31
22  38:   00140433                add     x8,x8,x1
23  3c:   01f40433                add     x8,x8,x31
24  40:   001484b3                add     x9,x9,x1
25  44:   01f484b3                add     x9,x9,x31
26  48:   00150533                add     x10,x10,x1
27  4c:   01f50533                add     x10,x10,x31
28  50:   001585b3                add     x11,x11,x1
29  54:   01f585b3                add     x11,x11,x31
30  58:   00160633                add     x12,x12,x1
31  5c:   01f60633                add     x12,x12,x31
32  60:   001686b3                add     x13,x13,x1
33  64:   01f686b3                add     x13,x13,x31
34  68:   00170733                add     x14,x14,x1
35  6c:   01f70733                add     x14,x14,x31
36  70:   001787b3                add     x15,x15,x1
37  74:   01f787b3                add     x15,x15,x31
38  78:   00180833                add     x16,x16,x1
39  7c:   01f80833                add     x16,x16,x31
40  80:   001888b3                add     x17,x17,x1
41  84:   01f888b3                add     x17,x17,x31
42  88:   00190933                add     x18,x18,x1
43  8c:   01f90933                add     x18,x18,x31
44  90:   001989b3                add     x19,x19,x1
45  94:   001a0a33                add     x20,x20,x1
46  98:   01fa0a33                add     x20,x20,x31
47  9c:   001a8ab3                add     x21,x21,x1
48  a0:   01fa8ab3                add     x21,x21,x31
49  a4:   001b0b33                add     x22,x22,x1
50  a8:   01fb0b33                add     x22,x22,x31
51  ac:   001b8bb3                add     x23,x23,x1
```


Fig.10. Le fichier *add.hex* généré par les commandes précédentes. La partie entourée en rouge doit être extraite et chargée dans la mémoire programme.

Dans la dernière partie de vérification, la même chaîne de compilation croisée est utilisée avec des vecteurs de test écrits en C et en précisant uniquement l'utilisation de l'ensemble RV32I. Ainsi, des tests beaucoup plus élaborés peuvent être créés pour la validation de l'architecture proposée. Un exemple de compilation d'un programme en langage C décrivant la suite de Fibonacci pour l'architecture RV32I est présenté ci-dessous.

```
riscv32-unknown-elf-gcc -I. -O2 -fno-pic -march=rv32i -mabi=ilp32 -fno-stack-protector -w -Wl,--no-relax -c fib.c
```

On voit clairement l'argument *-march=rv32i* précisant l'architecture RV32I. Comme pour les tests écrits en assembleur, cette commande génère un exécutable *a.out* qui doit être converti en code mémoire pour être chargé dans la mémoire ROM.

Test sur FPGA et amélioration des performances

Dans la dernière étape, l'architecture RV32I validée sur la totalité des instructions de l'ensemble RV32I au niveau comportemental, doit être adaptée pour un fonctionnement en plusieurs cycles d'horloge. Rappelons que le développement de l'architecture a été effectué en utilisant les blocs mémoire programme (la ROM) et de données (la RAM) totalement asynchrones. La lecture asynchrone permet de ne pas gérer dans un premier temps le délai d'un cycle d'horloge pour accéder à la donnée issue de la mémoire. Par contre, pour une validation expérimentale sur les plateformes de type FPGA comme Intel Altera DE10 ou autre, la lecture asynchrone n'est plus possible. C'est pourquoi la première optimisation à faire consiste à utiliser des blocs mémoire où la lecture ou l'écriture doivent être synchrones.

Dans cette première phase, les étudiants sont amenés à modifier la gestion des étapes de décodage d'instructions en prenant en compte les retards engendrés par une lecture synchrone des blocs mémoires. Les modifications seront à apporter au niveau du bloc *Contrôleur* où une machine à états finis doit être décrite pour cette gestion.

Les modifications au niveau du contrôleur ne doivent pas perturber le fonctionnement de l'architecture initiale. La validation doit donc utiliser la même batterie de tests que dans les étapes précédentes. Une fois l'architecture validée, une caractérisation des performances de l'architecture globale en termes de fréquence de fonctionnement maximale et en nombre d'instructions par seconde (en MIPS par exemple) pourra avoir lieu.

III. Conclusion

Nous avons présenté un ensemble de travaux pratiques qui seront dispensés au sein du Master EEA - Électronique Embarquée à l'université de Lorraine et Télécom Nancy - parcours Logiciel Embarqué, dans le cadre des modules Conception VLSI (Master EEA) et Conception et Développement d'un Système sur Puce (Télécom Nancy). Ces travaux pratiques portent sur la modélisation de systèmes et circuits en VHDL, et se focalisent en particulier sur le développement d'une architecture de processeur à jeu d'instructions réduit de type RISCV (le jeu d'instruction RV32I) en utilisant des suites logicielles de Mentor Graphics (Modelsim pour la modélisation et simulation VHDL), et de Intel Altera (Quartus Prime) pour la validation expérimentale. La mise en place de ces TP est déjà en cours dans la formation Logiciel Embarqué à Télécom Nancy et est prévue pour la rentrée 2023/24 au sein du Master EEA.

Remerciements

Nous remercions, le GIP CNFM (4), porteur du projet INFORISM (ANR-23-CMAS-0024) (5) ainsi que le pôle MIGREST pour la participation financière à l'acquisition du matériel nécessaire à la réalisation de cette série de travaux pratiques.

Références

1. Harris, Sarah, and David Harris. 2021. Digital Design and Computer Architecture, RISC-v Edition. Morgan Kaufmann.
2. "RISC-v Instruction Encoder/Decoder." n.d. https://luplab.gitlab.io/rvcodecjs/.
3. "RISC-v Online Assembler." n.d. https://riscvasm.lucasteske.dev/#.
4. GIP-CNFM : Groupement d'Intérêt Public - Coordination Nationale pour la formation en Microélectronique et en nanotechnologies. Website : http://wwww.cnfm.fr
5. INFORISM, INgénierie de FORmations Innovantes et Stratégiques en Microélectronique, projet ANR-23-CMAS-0024-INFORISM au titre du programme France 2030.