MIPS R2000

Projet microélectronique

de Sainte Marie Nils, Edde Jean-Baptiste

4 septembre 2020

Table des matières

Table des figures

Introduction

Le but de ce projet est de dérouler le flow de conception d'un projet micro-électronique à travers la réalisation d'un coeur de microprocesseur, celui du MIPS R2000.
De la recherche bibliographique au placement routage, les objectifs sont multiples :

- Réaliser la description comportementale en SystemVerilog d'une architecture existante
- En réaliser la synthèse RTL et analyser les rapports de puissance, surface et timing
- Faire le placement routage et les vérifications électriques

Pour présenter au mieux le travail réalisé au cours de ces quelques mois, ce rapport est découpé en quatre parties :

- L'architecture générale du MIPS
- Flow de conception du coeur
- Tests et simulations comportementales de la description SystemVerilog
- Synthèse de l'architecture
- Placement et routage du circuit

La première partie se focalise sur les recherches effectuées autour de l'architecture du MIPS (écriture des instructions, fonctionnement de la mémoire et des registres, etc.) pour aboutir à un exemple de design simplifié qui sera complété pour tendre vers l'implémentation réel du MIPS R2000.
Ensuite, tout le découpage du problème ainsi que la description en SystemVerilog jusqu'aux tests sur Modelsim sont abordés pour finalement aboutir à la synthèse sur cible ASIC avec l'extraction des différents rapports quantitatifs sur le circuit tel que la puissance consommée, la place occupée ainsi que détails de timing.
Puis, nous allons finir par le placement routage qui s'est limité - on l'expliquera pourquoi - au placement des standard cells et à la génération de l'arbre d'horloge.

Avant de rentrer dans le coeur de notre implémentation du MIPS R2000, commençons donc par présenter une brève histoire des microprocesseurs MIPS
En 1981, sous le soleil californien, John Leroy Hennessy alors professeur à Stanford University, lance un projet de recherche nommé "Stanford MIPS" dans le but d'étudier un type d'ISA qui s'appellera plus tard RISC [9].
L'année 1984 sonna la fin du projet de recherche et la naissance de "MIPS Computer Systems, Inc" avec la commercialisation du premier microprocesseur MIPS deux ans plus tard, le MIPS R2000 basé sur une architecture 32 bits et cadencé à 8 MHz [6]. S'en suivi une (longue) série de microprocesseurs du R3000 au R10000, utilisés dans divers domaines comme l'embarqué ou encore la PlaySation de Sony. Aujourd'hui MIPS est encore en Californie, racheté en 2018 par la société Wave Computing qui propose des solutions à base d'IA aux entreprises [7].

A Architecture

A.1 Mémoire et registres du MIPS

La taille maximale d'une donnée adressable en mémoire est de **32 bits** ce qui constitue un mot mémoire ou **Word** en anglais. Ensuite, il est possible d'adresser 16 bits **Half Word** ou 8 bits soit un octet ou **Byte** en anglais.
Cela fait que la mémoire du MIPS est **découpée en octets**.

Byte (B) : 8 bits | Half Word (HW) : 16 bits | Word (W) : 32 bits

Au niveau des **registres**, le MIPS possède **32 registres** généraux de **32 bits**, avec le registre ***$zero*** qui à toujours la valeur zéro.
Voici un extrait de *Computer Organization And Design 5th Edition*[5] qui liste les 32 registres avec leur mnémoniques et leur **rôle** :

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

FIGURE 1 – Liste des Registres

A.2 Instructions du MIPS

Trois types d'instructions sont présentes dans le MIPS : **Type R**, **I** et **J**.
Elles sont toutes codées sur **32 bits** et se différencient par leur ***opcode*** correspondant aux **6 premiers bits** de l'instruction.

Pour les **Type R**, leur *opcode* est toujours nulle et ce sont les **6 derniers bits**, correspondant au champ ***function*** qui servent d'identification de l'opération.
Les champs **rs**, **rt**, **rd**, **sa**, sont respectivement, *register source, register target, register destination* et *sham* ou *shift*.

Type	opcode	rs	rt	rd	sa	fnc
R	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Exemple d'instruction (addition de deux registre dans un registre) :

add $rd, $rs, $vrt
rd <- rs + rt

Pour les **Type I**, les calculs sont fait avec des valeurs immédiates (**imediate**) sur **16 bits**, présentes ni dans les registre, ni dans la mémoire, mais codé directement dans l'instruction.
Cette fois-ci, c'est ***rt*** qui est écrit.

Type	opcode	rs	rt	immediate
I	6 bits	5 bits	5 bits	16 bits

Exemple d'instruction (addition d'un registre et d'un imediate dans un registre) :

addi $rs, $rt, 0x15

rt <- rs + 0x15

Pour les **Type J**, le compteur ordinale ***pc*** prend la valeur de l'adresse spécifiée multiplié par 4, ce qui effectue un **saut** dans le programme.

Type	opcode	adresse
J	6 bits	26 bits

Exemple d'instruction (saut à une adresse) :

J 0x21

pc <- 0x21 « 2

Pour consulter l'ensemble des instructions, voir **Annexe F.1**

A.3 Pipeline

Une implémentation un peu "grossière" ou disons "incomplète" du MIPS serait la suivante :

- Un **compteur ordinale** qui indique l'adresse de la prochaine instruction a exécuter et qui s'incrémente de 4 en 4 (registre pc).
- Un **banc de registre** de 32x32
- Une **UAL** (ou ALU en anglais) pour réaliser les opérations
- Un **accès** à la **mémoire**
- **Écriture** dans les registre

Bien sur, il y a un peu plus de logique autour et la figure ci-dessous montre une **implémentation basique** de ce MIPS.

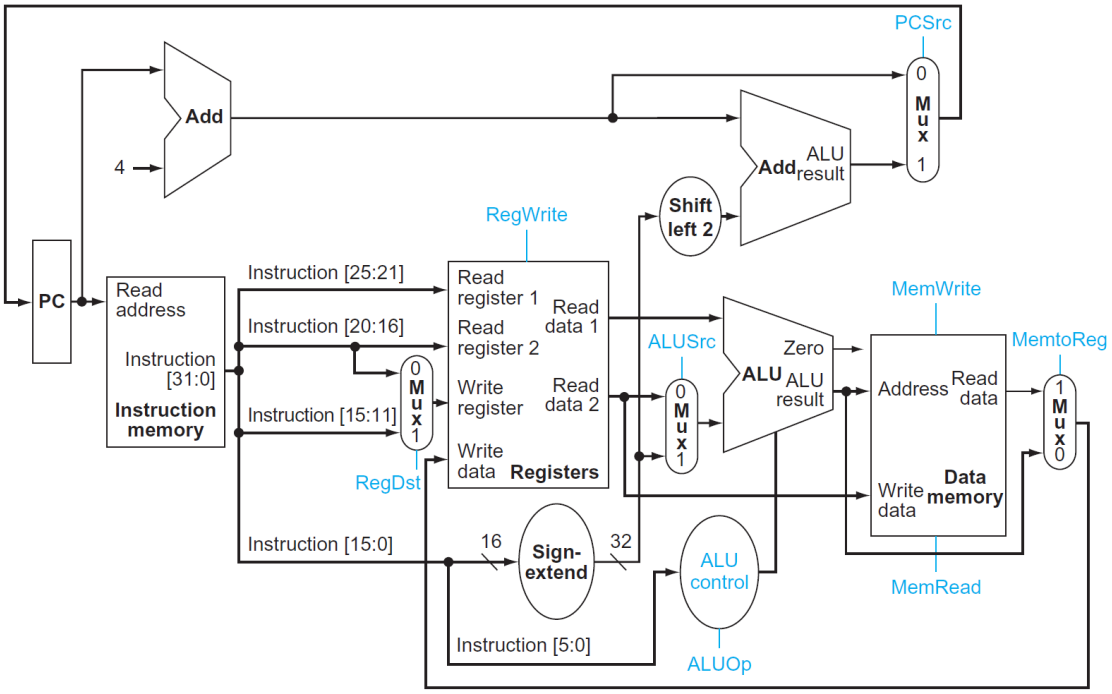


FIGURE 2 – Vue simplifier du single-cycle MIPS

Cette implémentation est appelé ***single-cycle*** car à **chaque cycle d'horloge** donc à chaque nouvelle valeur prise par le registre **pc**, une instruction est exécutée.
Cependant, cette architecture en "***un cycle***" n'est pas utilisé de nos jour et ça n'est pas non plus celle que nous avons réalisé pour plusieurs raisons.
En effet, même si ce design est parfaitement fonctionnel, sa vitesse et donc la **fréquence d'horloge** est inévitablement **limité** par l'instruction la plus lente en l'occurrence, certainement une instruction de **load** qui aurait besoin de passer par toutes les étapes jusqu'à la écriture dans les registres pour être exécutée.
Dans les premiers ordinateurs, cette implémentation était utilisé et fonctionnait bien car le jeu **d'instructions** était très **limité**. Cependant si l'on veux rajouter des instructions un peu plus complexes ou encore la gestion de calcul flottant, le ***single-cycle*** devient inutilisable et **limite** beaucoup trop la **fréquence d'horloge** du circuit à mesure que les instructions se complexifient.

Une **solution** est la technique d'implémentation d'un ***pipeline***.
L'idée est simple, entre chaque étape/étage, à savoir, lecture de l'instruction, lecture des registres et décodage de l'instruction, calcul(s), accès à la mémoire et écriture dans le registre, les **sorties** des étages sont **enregistrés** avant d'être transmises à l'étage d'après.
Cela permet **d'exécuter plusieurs instructions** à la fois ; se trouvant à des phases différentes d'exécution.
La figure ci-dessous montre ce que cela donne si l'on "***pipline***" le ***single-cycle*** présenté plus haut.

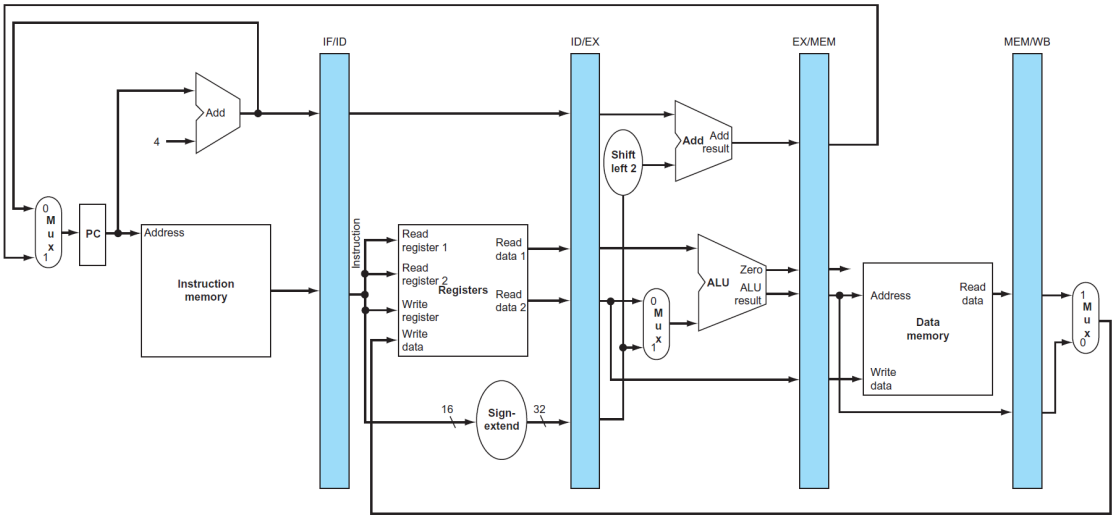


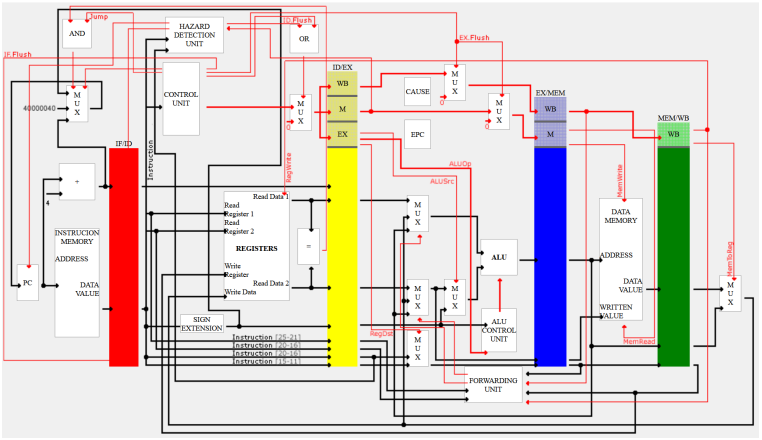
FIGURE 3 – Vue simplifier du Pipline

Bien que cette technique se présente comme plus performante par rapport au single-cycle, il y a quelques **hypothèses** qu'il est nécessaire de respecter.
Le pipeline doit être **équilibré**, c'est à dire que chaque étage doit prendre à peu près le même temps à faire ses opération et c'est ce temps qui fixe la fréquence d'horloge d'où l'importance d'un équilibre.
Enfin, le ***pipline*** doit être (toujours) le lus souvent **plein** au quel cas, on peu rapidement se retrouver dans le cas du ***single-cycle*** ce qui n'est pas le but...
Cette technique ajoute cependant quelque subtilité qu'il n'y avait pas dans le single-cycle comme des ***aléas***, mais cela est traité plus loin dans le rapport.

B Flow de conception

B.1 Recherches

Nous avons commencé nos recherches est nous sommes arrêté sur l'architecture du WebMips [3] que l'on a jugé plutôt claire tout en étant complète et que l'on montre ci-dessous :



C'est un peu plus tard, que l'on est tombé sur le fameux *Computer Organization And Design 5th Edition* qu'on a déjà cité plus tôt puisque notre projet se repose en grande partie dessus.

B.2 Découpage du problème

B.2.1 Premiers étapes : construction d'un MIPS "simple"

Au vue du schéma du WebMips, il nous est naturellement venu l'idée de concevoir les étages du pipeline un à un en commençant bien évidemment par l'étage de lecture.
Après avoir bouclé avec l'étage d'écriture, nous avions remarqué que nos données n'étaient pas enregistrées d'un étage à l'autre. Notre architecture se rapprochait plus alors du MIPS **single-cycle**.

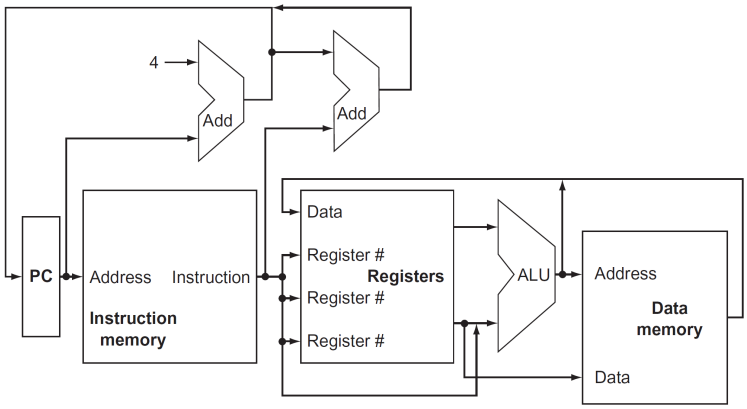


FIGURE 4 – Première étape

Nous avons donc ajouté des bascules pour réaliser le pipeline fonctionnel, et à la fin nous avons fini par obtenir un circuit suivant ce schéma (bien qu'en réalité il y a un peu plus de bloc car nous avons rajouté beaucoup plus d'instructions (**voir Annexe**) que celles décrites dans dans *Computer Organization*) :

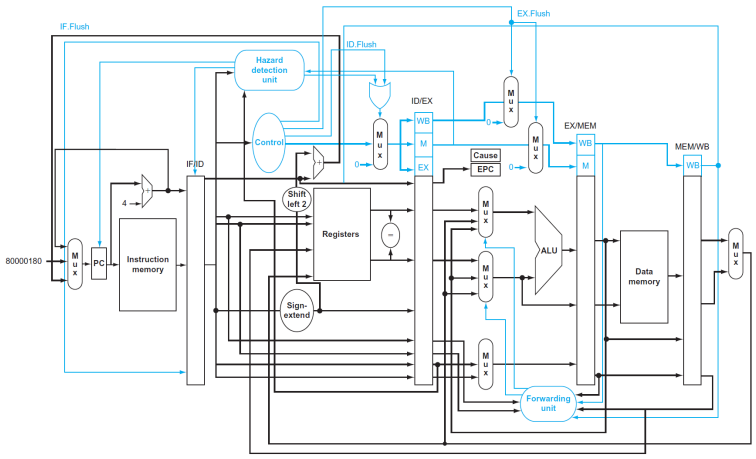


FIGURE 5 – Design "complet"

Fetch

Dans cet étage de **lecture**, il y a trois éléments importants : le registre *pc*, la ROM qui contient les instructions et les deux additionneurs pour passer à l'instruction suivante et prendre les branchements.

Decode

L'étage de décodage de l'instruction comporte, quant à lui, trois modules majeurs : le **banc de registres**, qui nous le allons voir plus tard, occupe une grande place dans le processeur ; l'unité de contrôle des **instructions**, qui est "l'essence" de ce module puisque c'est lui qui effectue le décodage ; et enfin l'unité de contrôle des **aléas** dont on parlera en détail juste après.

Execute

Une fois le décodage terminé, les sorties de l'unité de **contrôle des instructions** rentre dans l'unité de **contrôle de l'ALU** pour que ce soit la bonne opération qui soit effectué dans l'ALU lui-même pendant que les opérandes sont déterminées par une série de MUX couplés à une *forwarding unit*.

Memory

Il n'y a que deux types d'accès mémoire, soit en écriture soit en lecture (instructions **store** et **load**).
Lors d'un **store**, la donné à écrire est présente dans le registre *rt* et l'adresse à laquelle écrire est directement calculé en sortie de l'ALU (*immediate*+*rs*).
Lors d'un **load**, cette fois-ci, c'est le registre *rt* qui va recevoir le contenu de la mémoire à l'adresse (*immediate*+*rs*).

Write Back

Dans cet étage très simple, seulement deux types de données peuvent être écrite dans les registres. Soit à l'issue d'un calcul donc directement à la sortie de l'ALU soit à l'issue d'un load donc directement en sortie de la mémoire.

B.2.2 Contrôle des aléas

Une chose qui a été évoqué à la fin du **A.3**, sont les aléas que génère la mise en place du pipline.

Forwarding unit

Prenons le programme suivant :

I1 : add $at, $v0, $v1 ; at<-v0+v1
I2 : sub $v0, $at, $v1 ; v0<-at-v1
I3 : sub $a0, $at, $a1 ; a0<-at-a1

Ici on écrit d'abord dans le registre *$at* puis on utilise sa valeur dans I2 et I3. Or, quand I2 sera en phase de décode, le registre *$at* n'aura toujours pas pris sa nouvelle valeur. Il faut donc la "forwarder" depuis la sortie de l'étage d'execute. Pareil, quand I3 est décodé, la nouvelle valeur de *$at* doit être "forwardé" depuis la sortie du Write Back. Ce même soucis se pose pour tous les opérandes.

Hazard detection unit

Prenons le programme suivant :

I1 : lw $at, 0x12($v0) ; at <- memory[v0+0x12]
I2 : sub $at, $v0, $v1 ; at<-v0-v1

Ici, on charge un mot mémoire dans *$at* puis on veut y accéder à I2 or cette valeur ne sera accèssible que dans l'étage de Write Back, ce qui implique d'ajouter une bulle et lire deux fois I2.
Le même problème se pose avec les branchements mais dans l'autre sens, il faut d'abord calculer effectuer le calcul pour évaluer le branchement.

I2 : sub $at, $v0, $v1 ; at<-v0-v1
I1 : beq $at, $v0, 0x12 ; branch si at==v0

Pour voir l'implémentation en SystemVerilog de la Hazard et Forwarding unit, voir **Annexe F.2**

B.2.3 Implémentation des modules externes : les mémoires

ROM

Nous avons commencé par l'implémenter au sein même du *fetch* mais cette méthode n'avait aucun sens matériellement. Nous avons donc fini par la placer à l'extérieur du MIPS dans le banc de test.

RAM

De la même manière, la RAM a migré du module mémoire au banc de test. Cependant, cette première RAM était une colonne de 32bits. Lors de l'ajout des fonctions *load byte, store byte, load halfword*, etc. nous avons dû la changer en une structure en 4 colonnes de 8bits se rapprochant de la réalité par la même occasion. (Voir Annexe)

C Test du circuit

C.1 Test des étages

Comme il est dit dans la partie **Flow de conception**, on a commencé par réaliser les étages un par un que l'on testait au fur et à mesure. On initialisait alors toutes les entrées dans le fichier "testbench.sv".

C.2 Test des instructions

Puis vint le moment où nous avons implémenté la ROM, nous permettant ainsi de charger une variable d'un tableau comportant plusieurs mots, donc instructions, pour pouvoir ensuite écrire dans un document texte à part les suites d'instructions. Comme nous ajoutions les instructions petit à petit, nous les testions afin de finaliser cet ajout. Nous avons maintenant un fichier comportant toutes les instructions que l'on utilisent pour valider chaque modification que l'on apporte sur notre code RTL.

```
000733C0 //sll $a2 $a3 0x10 a2 <- ?
0022282A //slt $a1 $at $v0 a1 <- at<v0 ? =1
0041282A //slt $a1 $v0 $at a1 <- at<v0 ? =0
00063402 //srl $a2 $a2 0x10 a2 <- 1

00E63004 //sllv $a2 $a2 $a3 a2 <- 2
00E63006 //srlv $a2 $a2 $a3 a2 <- 1

00620820 //add $at $v1 $v0 at <- v1+v0=5
00222022 //sub $a0 $at $v0 a0 <- at-v0=3
00622825 //or $a1 $v1 $v0  a1 <- v1|v0=3
00622824 //and $a1 $v1 $v0 a1 <- v1&v0=2
00413026 //xor $a2 $v0 $at a2 <- v0^at=7
00622827 //nor $a1 $v1 $v0 a1 <- -4
```

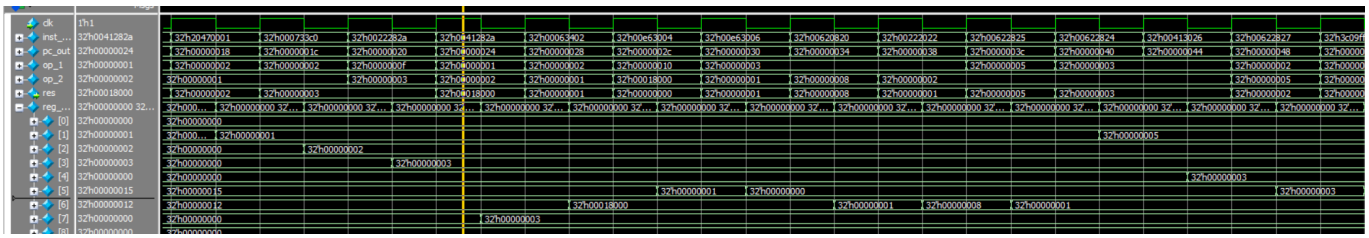


FIGURE 6 – Exemple d'une suite d'instruction dans notre programme de tes exhaustif

C.3 Test avec programmes

Seulement, après l'ajout de la *forwarding unit* et la *hazard detection unit* le simple test de nos instructions ne suffisait pas à vérifier le bon fonctionnement du processeur. Nous avons donc maintenant un programme de "data dependency" ainsi qu'un "vrai" programme complet calculant les 2*N premiers termes de la suite de Fibonacci entre deux registres.

```
//Fibonacci
8C030027 //lw v1 A(zero) v1<-A
8C010007 //lw at 1(zero) at<-1
AC01000B //sw at 2 zero  mem[2]<-at
8C020003 //lw v0 2(zero) v0<-at

00410820 //add at at v0 at <- at + v0
AC01000B //sw at 2 zero  mem[2]<-at

20840001 //addi a0 a0 0x1  a0++
00221020 //add v0 at v0 at <- at + v0
00410820 //add at at v0 at <- at + v0
1464FFFD //BNE v1 a0 0xFFFD
```

On voit ci-dessous le résultat de la simulation avec l'initialisation des valeur dans la mémoire (en bleu) puis le calcul dans les registres (reg_file[1] et reg_file[2]) jusqu'à la fin de la boucle (au niveau de la flèche rouge) s'effectuant après avoir bien calculé les 2*N premiers termes avec N=10.

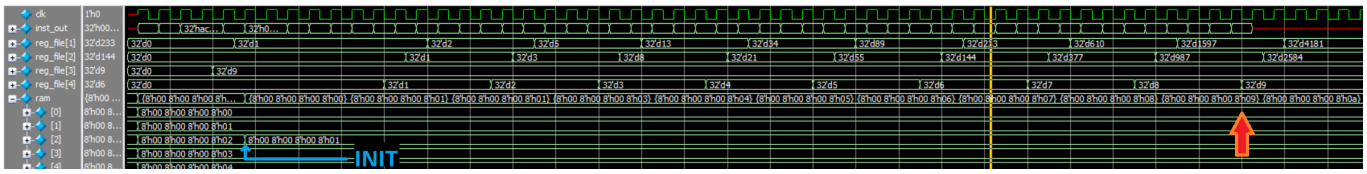


FIGURE 7 – Exemple de programme complet

D Synthèse

Une fois les validations effectuées, il était temps de faire la synthèse et pour nous c'était sur cible ASIC :

D.1 Première synthèse avec la RAM/ROM intégrée

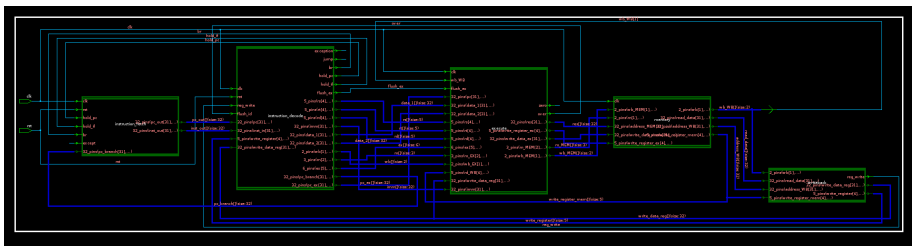
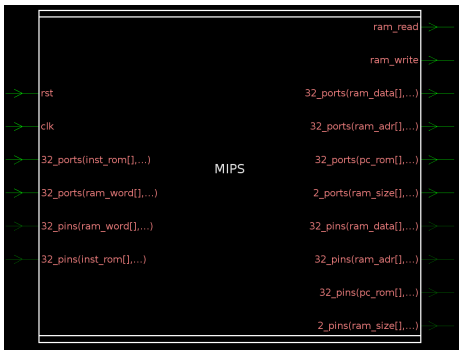


FIGURE 8 – Premier résultat de la première synthèse

Voici, ci-dessus, le résultat de notre première synthèse, cependant il a fallut la refaire à cause de nos choix discutables sur l'implémentation de la RAM et de la ROM. On voit que notre processeur ne contient uniquement que deux entrées : **clk** et **rst**.

D.2 "Vrai" synthèse

Nous avons donc réalisé une nouvelle synthèse après avoir modifié le système en rajoutant les entrées/sorties nécessaires pour discuter avec la mémoire (voir **Architecture**) pour obtenir le "boitier ci-dessous" :



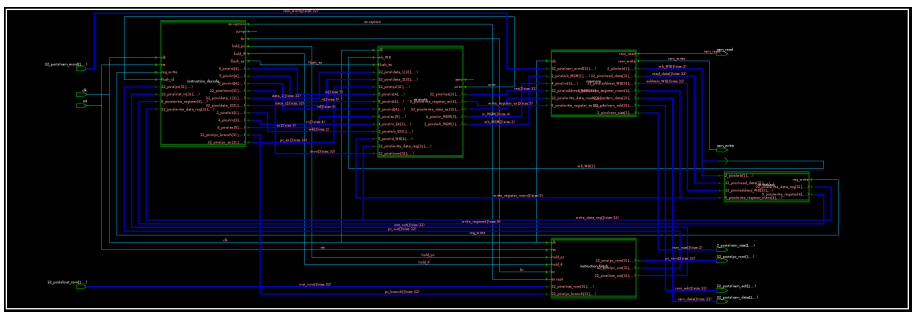


FIGURE 9 – Module de la synthèse finale

Ci-dessous, le rapport de hiérarchie comportant tout nos modules ainsi que ceux rajouté par la synthèse :

```
MIPS
    EX
        ALU
            ALU_DW01_add_2
            ALU_DW01_ash_1
            ALU_DW01_cmp2_1
            ALU_DW01_sub_3
            ALU_DW_rash_1
            ALU_DW_rash_2
        ALU_ctrl_unit
        execute_MUX_RTRD
        forwarding_unit
    ID
        ID_DW01_add_0
        ID_DW01_cmp6_1
        decode_CONTROL_UNIT
        decode_HAZARD_UNIT
        decode_REG_MAPP
    IF
        fetch_MUX
            fetch_MUX_DW01_add_2
        fetch_PC_REG
    MEM
    WB
```

FIGURE 10 – Hiérarchie finale

D.2.1 Puissance

```
execute                  4.6321       N/A       N/A (N/A)   1730.8846  h
instruction_decode      38.6034       N/A       N/A (N/A)   4912.5225  h
instruction_fetch        3.5579       N/A       N/A (N/A)    431.8616  h
memory                   2.6145       N/A       N/A (N/A)    197.6619  h
writeback                0.0698       N/A       N/A (N/A)     37.4527  h
------------------------------------------------------------------------------
Totals (39 cells)       49.518mW     N/A       N/A (N/A)      7.329nW
```

Nous avons donc un circuit qui consomment la majorité de sa puissance du bac des 32 registres : 78% de la puissance du circuit.
A noter que le circuit possède également un puissance consommé de 770 µW. Soit 10 fois plus que la puissance consommé par le *writeback*, on peut donc se demander s'il est vraiment nécessaire de le désigner en tant qu'un top module au même titre que le decode, et si l'on ne pourrait pas simplement l'inclure dans le module *memory*.

D.2.2 Surface

```
Number of ports:                          166
Number of nets:                           623
Number of cells:                           39
Number of combinational cells:             34
Number of sequential cells:                 0
Number of macros/black boxes:               0
Number of buf/inv:                         33
Number of references:                       7

Combinational area:             424878.998978
Buf/Inv area:                    77459.200714
Noncombinational area:          462298.187378
Macro/Black Box area:                0.000000
Net Interconnect area:          177399.000000

Total cell area:                887177.186356
Total area:                    1064576.186356
```

Ci-dessus, le rapport de surface montre les généralités du circuit. Comme le nombre de port de 166 très élevé dû aux entrées/sorties de mots de 32 bits pour la mémoire RAM et la ROM. Il nous montre également la proportion de cellule combinatoire dans le circuit soit : 34 pour 39 cellules. Cependant, même si la plupart des cellules sont combinatoires, ça ne veut pas dire que ce sont elles qui occupent la majorité du circuit. En effet, on voit un plus bas dans le rapport que la surface occupée par les circuits combinatoires (424878) est inférieure à celle occupée par les non-combinatoires (4622298). De ces deux surfaces sont ajouté les inter-connections ce qui nous ramène à une utilisation de la surface par les cellule de 83%.

Hierarchical area distribution

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combinational	Local cell area: Noncombinational	Local cell area: Black-boxes	Design
MIPS	887177.1864	100.0	1838.2000	0.0000	0.0000	MIPS
execute	222021.8001	25.0	24824.8002	20748.0000	0.0000	EX
execute/alu	167530.9999	18.9	35908.6002	0.0000	0.0000	ALU
execute/alu/add_170	22204.0000	2.5	22204.0000	0.0000	0.0000	ALU_DW01_add_2
execute/alu/lt_174	8408.4000	0.9	8408.4000	0.0000	0.0000	ALU_DW01_cmp2_1
execute/alu/sll_175	27627.5997	3.1	27627.5997	0.0000	0.0000	ALU_DW01_ash_1
execute/alu/sra_176	21221.2002	2.4	21221.2002	0.0000	0.0000	ALU_DW_rash_1
execute/alu/srl_177	27390.9999	3.1	27390.9999	0.0000	0.0000	ALU_DW_rash_2
execute/alu/sub_171	24770.2000	2.8	24770.2000	0.0000	0.0000	ALU_DW01_sub_3
execute/alu_ctrl_unit	2293.2000	0.3	2293.2000	0.0000	0.0000	ALU_ctrl_unit
execute/fw_unit	5714.8000	0.6	5714.8000	0.0000	0.0000	forwarding_unit
execute/mux_RTRD	910.0000	0.1	910.0000	0.0000	0.0000	execute_MUX_RTRD
instruction_decode	593119.7865	66.9	5605.6000	37674.0000	0.0000	ID
instruction_decode/add_358	3731.0000	0.4	3731.0000	0.0000	0.0000	ID_DW01_add_0
instruction_decode/control_UNIT	4550.0001	0.5	4550.0001	0.0000	0.0000	decode_CONTROL_UNIT
instruction_decode/hazard_unit	1565.2000	0.2	1565.2000	0.0000	0.0000	decode_HAZARD_UNIT
instruction_decode/r60	8936.2000	1.0	8936.2000	0.0000	0.0000	ID_DW01_cmp6_1
instruction_decode/reg_MAPP	531057.7865	59.9	176885.7989	354171.9875	0.0000	decode_REG_MAPP
instruction_fetch	47192.5999	5.3	3403.4000	19219.2002	0.0000	IF
instruction_fetch/mux	13122.2001	1.5	5514.6001	0.0000	0.0000	fetch_MUX
instruction_fetch/mux/r54	7607.6000	0.9	7607.6000	0.0000	0.0000	fetch_MUX_DW01_add_2
instruction_fetch/pc_REG	11447.7996	1.3	345.8000	11101.9996	0.0000	fetch_PC_REG
memory	19383.0000	2.2	0.0000	19383.0000	0.0000	MEM
writeback	3621.7999	0.4	3621.7999	0.0000	0.0000	WB
Total			424878.9990	462298.1874	0.0000	

FIGURE 11 – Rapport de surface

Maintenant, au sein des cellules, on retrouve une répartition de la surface totale similaire à la répartition de la puissance entre les modules, avec le module **instruction_decode** occupant plus de la moitié de la surface du module **MIPS** puisqu'il possède le bac des 32 registres de 32 bits

D.2.3 Vitesse

Pour le test de vitesse, nous avons choisit de commencer avec une fréquence d'horloge relativement faible par rapport aux standard d'aujourd'hui (100MHz) car nous travaillons avec une techno. relativement grosse (0,35 μm) et que notre architecture est celle du premier MIPS qui au moment de sa sortie possédait une horloge à 8 MHz.

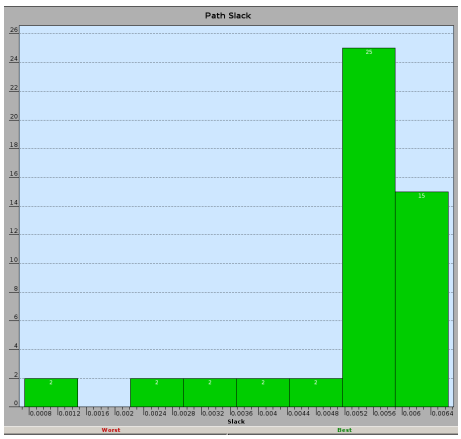


FIGURE 12 – Test du chemin critique pour 10ns soit 100MHz

Nous sommes tout de même parvenue à monter jusqu'à 200 MHz - comme les derniers R2000 en production - sans soucis mais n'avons pas voulu monter plus haut car nous avions certaines difficultés à rentrer dans design_vision le chemin que l'on jugeais être le chemin critique à savoir le calcul dans l'étage d'exécution.

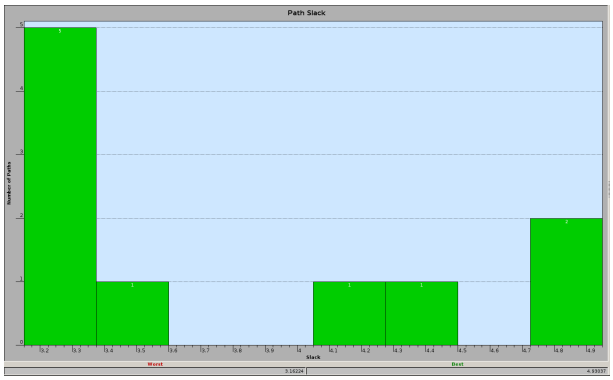


FIGURE 13 – Test du chemin critique pour 5ns soit 200MHz

D.2.4 Autre

Net	Fanout	Fanin	Load	Resistance	Pins
Total 623 nets	1791	623	28.18	12.54	2414
Maximum	382	1	4.97	2.67	383
Average	2.87	1.00	0.05	0.02	3.87

FIGURE 14 – Rapport des noeuds

On observe que le fanout moyen est de l'ordre de 1 à 2 si l'on omet la clock qui est à 382 (augmentant beaucoup la moyenne) ainsi que divers autres signal pouvant se rapprocher de la dizaine.
Dans le rapport des ports, nous avons remarqué que les entrées et sorties étaient indiquées avec des impédances nulles. Cette approximation peut probablement entrainer une consommation du circuit réduite par rapport à la réalité.

E Placement routage

Comme pour la synthèse, nous avons fait un premier placement/rootage "approximatif". Nous n'avions pas à disposition des fichiers "io" représentant la couronnes d'entrées/sortie. Mais nous avons quand même pu appréhender l'outil **Innovus** pour obtenir le résultat suivant :

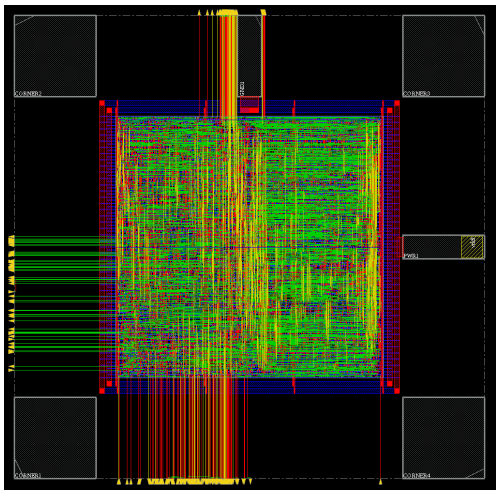


FIGURE 15 – Première tentative de rootage

Après avoir donc crées les fichiers "mips_io.v", "mips_pads.io" et modifié le script "init.tcl" et suivi l'énoncé de "TP_Filtre" jusqu'à la génération de l'arbre d'horloge et au remplissage, nous avons pu obtenir ce placement plus "propre" :

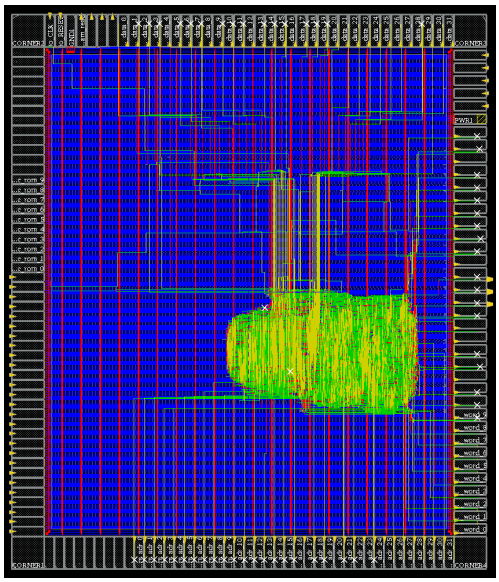


FIGURE 16 – Rootage finale

Or, comme on l'a évoqué plus haut, nous avons un grand nombre d'entrée/sortie bien trop élevé. Si bien que nous avons une occupations des cellules de moins de 4%.

Conclusion

Le bilan de l'avancement du flow de conception peut donc se résumer ainsi :

— Nous avons fini la partie description comportementale, même s'il nous reste tout de même la fonction *jump register* et peut-être une poignée d'autres instructions.

— Nous avons bien entamé la partie synthèse RTL, il y a certains rapport qui mériterait d'être plus approfondie (paramètres/datapath à choisir).

— Enfin, nous avons pu goûter au placement/rootage, mais il s'est limité au placement des cellules et à la génération de l'arbre d'horloge. Nous avons que peu étudié les rapports en partie car notre placement/rootage est irréalisable.

S'il nous restait un peu plus de temps nous aurions essayé d'implémenter les fonctions **mult** et **div** possédant des résultats sur 64bits sur deux nouveaux registres *hi* et *lo* ainsi que les fonctions écrivant sur ces registres **mfhi** et **mflo**. On aurait également pu essayé de réduire le nombre de ports du module MIPS essayant de parallélisation ou autre stratégie.

F Annexes

F.1 Liste des instructions

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

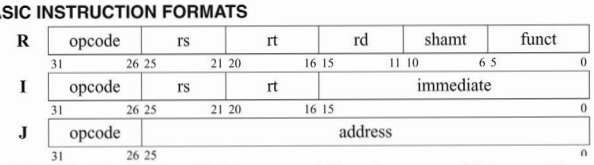
MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS



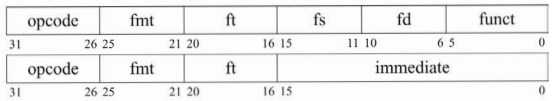
ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FORMAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

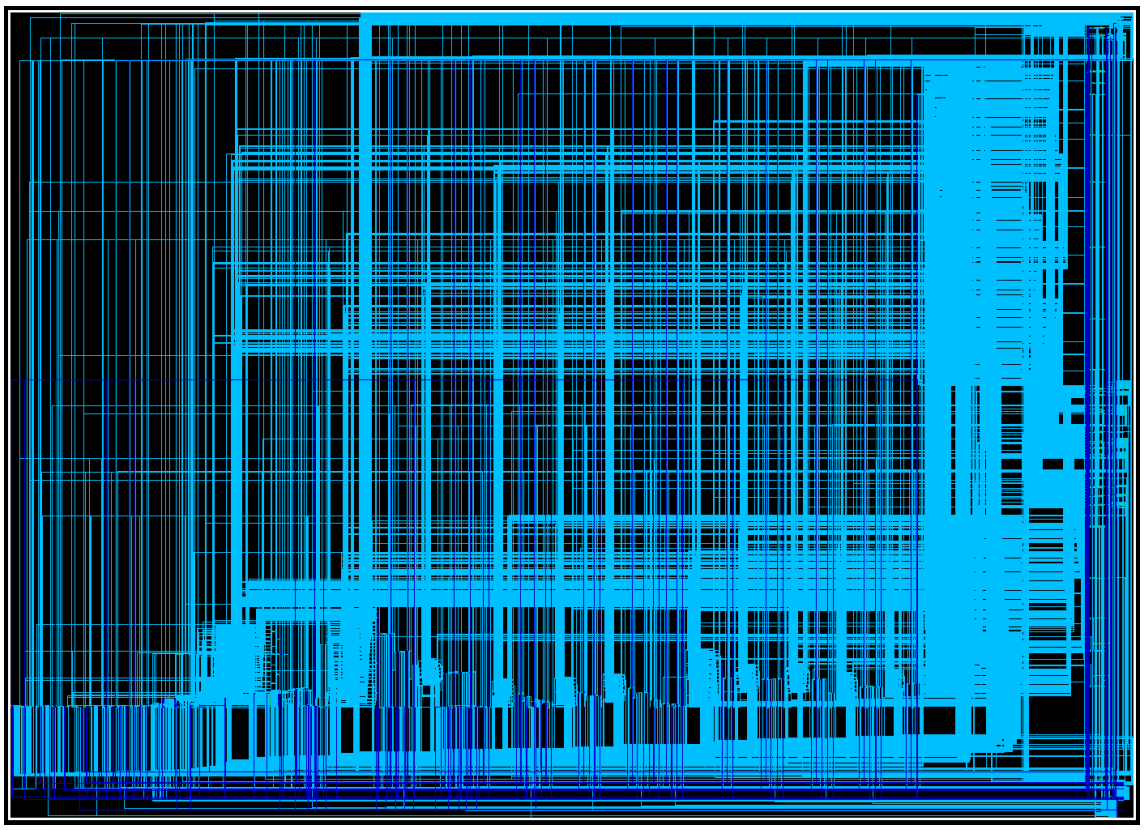
F.2 Hazard et Forwarding unit

```
module forwarding_unit ( rs_id, rt_id, rd_ex, reg_write_ex, rd_wb, reg_write_wb, forward_a, forward_b );

  //Inputs declaration
  input [4:0] rs_id, rt_id, rd_ex, rd_wb;
  input reg_write_ex, reg_write_wb;

  //Outputs declaration
  output [1:0] forward_a, forward_b;

  //Variables declaration
  reg [1:0] fw_a, fw_b;


  //------Code starts Here------//
  always_comb
    begin
      fw_a = 0;
      fw_b = 0;

      if ( reg_write_ex && (rd_ex!=0) && (rd_ex==rs_id) )
        fw_a = 2;
          if ( (reg_write_wb && (rd_wb!=0)) && !(reg_write_ex && rd_ex!=0 && rd_ex==rs_id) && (rd_wb==rs_id) )
        fw_a = 1;
      if ( reg_write_ex && (rd_ex!=0) && (rd_ex==rt_id) )
        fw_b = 2;
          if ( (reg_write_wb && (rd_wb!=0)) && !(reg_write_ex && rd_ex!=0 && rd_ex==rt_id) && (rd_wb==rt_id) )
        fw_b = 1;
    end

  assign forward_a = fw_a;
  assign forward_b = fw_b;

endmodule // End of module forwarding_unit
```

Code de la Forwarding unit

```
module decode_HAZARD_UNIT ( rt_id, rs_id, rt_ex, mem_Read, mux_ctrl_unit, hold_pc, hold_if );

  //Inputs declaration
  input [4:0] rt_id, rs_id, rt_ex;
  input mem_Read;

  //Outputs declaration
  output reg mux_ctrl_unit;
  output reg hold_pc, hold_if;

  //------Code starts Here------//
  always_comb
    begin
      hold_pc = 0;
      hold_if = 0;
      mux_ctrl_unit = 0;

      if ( mem_Read && (rt_ex==rs_id || rt_ex==rt_id) )
        begin
          hold_pc = 1;
          hold_if = 1;
          mux_ctrl_unit = 1;
        end
    end

endmodule // End of module decode_HAZARD_UNIT
```

Code de la Hazard detection unit



Premier résultat de la première synthèse pour le plaisir de vos yeux

Pour ce projet nous avons utilisé d'autres sources non cités, que ce soit sur le langage SystemVerilog [8], sur les instructions du MIPS [2] ou encore des outils comme un convertisseur d'instructions assembleur/hex/binaire [10].
Pour la compilation du code, les simulations et génération des chronogramme, nous avons utilisé Modelsim, mais aussi EDA playground [1] qui est une sorte d'équivalent mais en ligne. Ensuite, pour la production du code SV en lui même, nous avons utilisé le package "teletype" d'Atom, pour partager en directe le code. Finalement, tous le projet était gérer avec Github [4] ce qui était plus simple pour nous surtout dans ces conditions de projet à distance, même si on l'avait commencé avant le confinement.

Listes des instructions de notre microprocesseur

J, JAL, BEQ, BNE : **Branchements et Sauts** (4)
ADDI, ADDIU, SLTIU, ANDIU, ANDI, ORI, XORI : **Opérations Imediate** (7)
LB, LW, LHU, SB, SH, SW, LBU : **Load et Store** (7)
ADD, SUB, AND, OR, XOR, NOR, SLT, SLL, SRL, SRA, SRAV, LUI, SRLV, SLLV : **Type R** (15)
Total = 33 instructions

Références

[1] Doulos. *EDA playground*. https://www.edaplayground.com/. Pour les simulations en parallèle de Modelsim. 2015.

[2] James F. Frenzel. *MIPS Instruction Reference*. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html. Pour les instructions. 1998.

[3] Roberto Giorgi. *WebMips*. http://bellerofonte.dii.unisi.it/index.asp. Accessed on 2012-11-11. 1988.

[4] Nils de Sainte Marie Jean-Baptiste Edde. *MIPS-R2000*. https://github.com/JBtoriel/MIPS-R2000. GitHub du projet. 2020.

[5] David A. Patterson John L. Hennessy. "Computer Organisation and Design 5th edition". In : 4 (2014), p. 242-344.

[6] *MIPS architecture*. https://en.wikipedia.org/wiki/MIPS_architecture. 2020.

[7] *MIPS Technologies*. https://en.wikipedia.org/wiki/MIPS_Technologies. 2020.

[8] Alexis Polti. *Cours en ligne les HDL*. https://hdl.telecom-paristech.fr/verilog_structurel.html. Pour les petites piqûres de rappel. 2014.

[9] *Stanford MIPS*. https://en.wikipedia.org/wiki/Stanford_MIPS. 2019.

[10] Bucknell University. *MIPS Converter*. https://www.eg.bucknell.edu/~csci320/mips_web/. Convertisseur MIPS assembleur, hexa/binaire. 2014.