

Architecture des ordinateurs
Processeur MIPS



Alok Menghrajani
Alexandre Lopes

Mai 2002

Processeur MIPS

[Logo fortement inspiré par Intel]

Table des matières

1	Introduction	1
2	But	2
3	Spécification de notre processeur MIPS	2
3.1	Rappel historique	2
3.2	Généralités	2
3.3	Instructions de type-R	3
3.3.1	L'addition	3
3.3.2	La soustraction	3
3.3.3	Le et-logique	3
3.3.4	Le ou-logique	4
3.4	Instructions de type-I	4
3.4.1	L'addition immédiate	4
3.4.2	Le saut conditionnel	4
3.4.3	Load Word	5
3.4.4	Store Word	5
4	Méthodologie	5
4.1	Logiciels utilisés	5
4.2	Les composants de base.	6
4.2.1	ALU	6
4.2.2	Register File	6
4.2.3	Multiplexeurs	6
4.2.4	Registre	7
4.2.5	Portes Logiques	7
4.2.6	Sign Extended	7
4.2.7	Unités de mémoire	7
4.3	L'unité de contrôle	9
4.4	Le test	11
5	Analyse	11
5.1	Problèmes rencontrés	11
5.2	Optimisation	11
5.3	La réalité	12
6	Conclusion	12
7	Bibliographie	14

8	Annexes	i
8.1	Plans	i
8.2	Codes sources	vii
8.2.1	VHDL	vii
8.2.2	Assembleur	xxiii
8.3	Copies d'écran	xxiv

1 Introduction

Pour manipuler des données, il faut réunir deux éléments : un algorithme (une séquence d'opérations) et des ressources matérielles (pour effectuer les opérations). Un processeur est un système qui réunit les ressources matérielles et qui possède un moyen d'accès vers l'algorithme et les données. En général, un processeur va prendre des instructions se trouvant dans une ROM¹ et les exécuter de façon séquentielle. Certaines instructions permettent de rompre l'ordre séquentiel et d'effectuer des sauts.

Les instructions que peut accepter un processeur sont très simples. Un processeur peut par exemple additionner deux nombres, aller chercher ou enregistrer des données, comparer deux nombres afin de choisir quelle va être la prochaine instruction.

D'un point de vue matériel, le processeur est une entité composée de plusieurs sous-unités :

- ALU (Unité arithmétique et logique)
Unité qui permet d'effectuer les opérations mathématiques de base ; l'addition, la soustraction, le et-logique, le ou-logique.
- Register File (Registres)
Ensemble des registres qui sont utilisés comme intermédiaires dans les calculs. Les registres sont directement situés sur le processeur et on peut donc y accéder beaucoup plus rapidement.
- Trois bus de données et deux bus d'adresses
Utilisés pour les composantes externes : dans notre cas pour la RAM² et la ROM.
- PC (Program Counter)
Registre qui pointe vers l'instruction à exécuter.
- Unité de contrôle
Unité qui va gérer l'utilisation des ressources ci-dessus. Cette unité se représente comme une machine d'états.
- Datapath
Circuit qui relie les différentes ressources.

¹ "Read-Only Memory", mémoire en lecture uniquement

² "Random-Access Memory", mémoire volatile utilisée pour lire et écrire des données

2 But

Lors du semestre d'hiver nous avons effectué des laboratoires qui nous ont permis de nous familiariser avec les composants de base d'un processeur. L'objectif de ce TP est de concevoir et simuler un processeur simple mais complet. Nous avons donc dû réunir les composants vus lors du semestre précédent ainsi qu'implémenter l'unité de contrôle.

3 Spécification de notre processeur MIPS

3.1 Rappel historique

En 1981, le Dr. John Hennessy (co-auteur du "Computer Organization & Design" [1]) développa le premier processeur RISC³ commercialisé, dont l'architecture devint le standard ISA⁴ appelé MIPS.

En 2000, l'utilisation de l'architecture MIPS est très répandue. Plusieurs processeurs embarqués l'utilisent et Sony s'en servi pour sa plateforme de jeux PlayStation.

3.2 Généralités

Nous allons juste implémenter une petite partie de la spécification MIPS. Nous allons considérer le 'mot' comme étant 32 bits, et le 'byte' comme 8 bits.

Les codes sources sont généralement commentés en anglais, principalement pour des raisons de préférences personnelles, mais aussi à cause du gain de place que cela procure (les phrases en anglais sont très souvent plus courtes).

Voici un survol du processeur :

- Toutes les instructions ont une longueur fixe de *un mot* et sont codées régulièrement.
- Nous utilisons un modèle load/store, donc les opérandes sont toujours des registres.
- Nous avons 32 registres de 32 bits.
- Le système de mémoire est basé sur l'architecture de "Harvard".
- Le PC pointe toujours vers une adresse multiple de quatre ("quad aligned"). Il n'est pas possible d'accéder directement à ce registre.
- Le processeur saute les instructions inconnues. Il y a cependant une exception (voir section 3.3, page 3).

Dans les sections qui suivent, nous allons introduire le jeu d'instructions que nous avons implémenté, ainsi qu'aborder les détails du processeur.

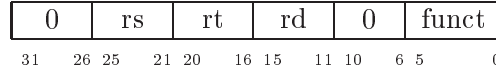
³"Reduced Instruction Set Computer", architecture qui prône la simplicité et la rapidité des instructions.

⁴Instruction Set Architecture

3.3 Instructions de type-R

Les instructions de type-R sont traitées par l'ALU. Nous en avons implémenté quatre : add, sub, and et or. Ces instructions sont exécutées en trois coups d'horloge.

Voici leur format :



rs, *rt*, *rd* sont les registres. Attention : l'ordre de ces registres n'est pas le même au niveau du codage de l'instruction et de l'assembleur.

funct est la fonction à effectuer. Nous nous sommes basé sur le cahier des charges et non sur la spécification MIPS. Par exemple, le 'add' que nous avons implémenté est en fait un 'addu' (addition sans *overflow*) et de même pour 'sub' qui est en fait un 'subu' (soustraction sans *overflow*). Les bits de *funct* sont utilisés directement par l'ALU :

```
88 ALUCtrl<=Funct(1)&not(Funct(2))&Funct(0);
```

(Controlunit, source complète : annexes, page xvii)

Lors du décodage des instructions r-type, l'unité de contrôle n'effectue aucune vérification sur la valeur de *funct*. Il est donc possible d'exécuter une instruction non-définie et de se retrouver avec une valeur abhérante. Il faut donc notamment faire attention d'avoir à la fin du code source quelque chose de la forme :

```
34 end_prog:      beq $zero, $zero, end_prog;
```

(Programme de test 2, source complète : annexes, page xviii)

3.3.1 L'addition

add rd, rs, rt

Effectue une addition. $rd = rs + rt$. La valeur de *funct* doit être 0x20.

3.3.2 La soustraction

sub rd, rs, rt

Effectue une soustraction. $rd = rs - rt$. La valeur de *funct* doit être 0x22.

3.3.3 Le et-logique

and rd, rs, rt

Effectue un et-logique. $rd = rs$ ET rt . La valeur de *funct* doit être 0x24.

3.3.4 Le ou-logique

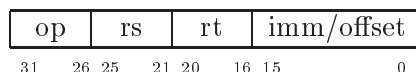
or rd, rs, rt

Effectue un ou-logique. $rd = rs$ OU rt . La valeur de *funct* doit être 0x25.

3.4 Instructions de type-I

Les instructions de type-I possèdent un opérande qui est une valeur constante sur 16 bits (valeur immédiate). Nous avons implémenté les quatres instructions suivantes : *addi*, *beq*, *lw*, *sw*. Ces instructions prennent trois coups d'horloge, sauf *sw* qui en prend un de plus.

Voici leur format :



op identifie l'opération à effectuer.

rs et *rt* sont les registres. A nouveau, attention à l'ordre, qui n'est pas le même au niveau du codage de l'instruction et de l'assembleur.

imm/offset est un nombre sur 16 bits. Ce nombre est signé et son signe est préservé lors de la conversion en 32 bits.

3.4.1 L'addition immédiate

addi rt, rs, imm

Effectue une addition. $rt = rs + imm$. La valeur de *op* doit être 0x08. Pour effectuer une addition avec une valeur immédiate plus grande que 16 bits, il faut garder le nombre en RAM, le charger dans un registre, puis effectuer l'addition simple :

```

1      addi $s0, $zero, 0x1234ABCD; # This is illegal !
2      lw   $t0, 0($zero);         # Assuming mem[0]=0
                                     x1234ABCD
3      add  $s0, $zero, $t0;

```

3.4.2 Le saut conditionnel

beq rs, rt, offset

Effectue un saut si *rs* et *rt* sont égaux. La valeur de *op* doit être 0x04. L'adresse du saut vaut $PC + 4 + 4 * offset$. Le saut est relatif au $PC + 4$, car ce dernier a été incrémenté avant d'exécuter le saut. L'offset est multiplié par 4, car toutes les instructions font *un mot*; ainsi il est possible d'effectuer des sauts plus longs, et garantir l'alignement du PC sur les instructions.

Au niveau matériel, la multiplication par quatre se fait à l'aide d'un décalage de deux bits vers la gauche.

3.4.3 Load Word

lw *rt*, *offset*(*rs*)

Charge le mot situé à l'adresse $rs + offset$ dans le registre rt . La valeur de op doit être 0x23. Le mot est chargé depuis la RAM. Nous ne respectons pas de délai d'attente ni ne détectons de signaux de confirmation. Nous partons du principe que la RAM est assez rapide pour le processeur.

3.4.4 Store Word

sw *rt*, *offset*(*rs*)

Enregistre la valeur de rt à l'adresse $rs + offset$. La valeur de op doit être 0x2B. Le mot est enregistré dans la RAM. Nous ne respectons pas de délai d'attente, ni ne détectons de signaux de confirmation, mais nous décalons le chargement de l'adresse de destination et le signal *chip select* afin que l'adresse soit stable avant l'écriture des données. Ceci explique que cette instruction prend un coup d'horloge de plus que les autres (voir section 5.2, page 11 pour de plus amples informations sur comment nous avons optimisé cela).

4 Méthodologie

4.1 Logiciels utilisés

Nous avons utilisé les logiciels suivant, dans l'ordre d'importance :

- ModelSim, Mentor Graphics.
Simulateur pour tester les différents composants et le processeur.
- HDL Designer, Mentor Graphics.
*Programme dans lequel nous avons écrit le code VHDL de chaque composant. Ce programme permet aussi d'écrire des composants supplémentaires nommés *tb*, pour faciliter les tests des composants de base.*
- Mips Assembleur, Xavier Perseguers.
Compilateur et simulateur de l'assembleur MIPS.
- Leonardo, Mentor Graphics.
Synthétiseur utilisé pour optimiser le code vhdl afin d'améliorer les performances du processeur.
- L^AT_EX.
Utilisé pour la mise en page du présent document.

4.2 Les composants de base.

La première tâche a été de récupérer les composants que nous avons conçus lors de semestre précédent et de les importer dans ce projet. Certains composants de base étaient nouveaux. Nous avons vérifié le bon fonctionnement de chaque composant.

4.2.1 ALU

(voir annexes, page x pour la source.)

C'est l'unité arithmétique et logique. Cette unité possède deux signaux d'entrées (a et b) de 32 bits, et un signal de contrôle (op) de 3 bits. Nous aurions pu n'utiliser que 2 bits pour ce signal, mais l'utilisation d'un bit supplémentaire permet d'optimiser le circuit. La sortie ($result$) est sur 32 bits. Quand cette sortie est nulle, le signal *zero* est activé. Nous ne générons pas de signal *overflow*, car nous n'en avons pas besoin, puisque la seule instruction de comparaison que nous avons implémenter est 'beq'.

<i>op</i>	<i>result</i>
000	a ET b
001	a OU b
010	$a + b$
110	$a - b$

(En fait c'est $a + \text{NON}(b) + 1$, mais c'est équivalent.)

4.2.2 Register File

(voir annexes, page xv pour la source.)

Le Register File est un ensemble de registres. Il est composé de 32 registres de 32 bits chacun. Ces registres sont d'utilisation générale, à l'exception du registre \$zero, dont la valeur est toujours nulle. Il y a cependant une convention au niveau de l'utilisation de ces registres ([1] p. 140), mais ceci n'est qu'une convention !

4.2.3 Multiplexeurs

(voir annexes, page xii pour la source.)

Nous utilisons les multiplexeurs pour sélectionner les signaux qui nous intéressent. Les quatre que nous utilisons dans notre processeur nous permettent de :

1. Sélectionner la source du PC avec le signal de contrôle *PCSrc* en cas de saut.

2. Sélectionner la source de l'ALU (le Register File ou la ROM dans le cas d'un immediate) avec le signal *ALUSrc*.
3. Décider si les informations viennent de la mémoire ou non, avec le signal *MemtoReg*.
4. Décider de la partie de l'instruction qu'on utilise comme adresse avec le signal *RegDst*.

4.2.4 Registre

(voir annexes, page viii pour la source.)

Ce sont de simples registres, ils sont utilisés par le IR et le PC. Le IR utilise le signal *clk* et prend les instructions dans la ROM en entrée, il est contrôlé par *IRWrite*. Le PC utilise les signaux *reset* et *clk* et prend en entrée soit $PC + 4$ soit l'adresse de saut, ceci étant décidé par le signal qui contrôle le saut (*PCWriteCond*).

4.2.5 Portes Logiques

(voir annexes, page vii pour la source.)

A l'exception des portes incluses dans la structure des autres composants, nous n'utilisons que deux portes logiques, une porte ET et une porte OU. Nous utilisons la porte ET entre le signal *zero* de l'ALU et le signal *PCWriteCond*. Nous utilisons la porte OU avec le signal de sortie de la porte ET et *PCWrite*.

4.2.6 Sign Extended

(voir annexes, page xvi pour la source.)

Ce composant permet l'extension du signe d'un nombre de 16-bits sur 32-bits. Comme nous l'avons vu précédemment, les champs "offset" et "imm" des instructions *addi*, *beq*, *lw* et *addi* sont sur 16-bits signés. On effectue une extension du bit de signe sur les 16-bits de poids fort pour obtenir une valeur sur 32-bits dont la valeur se situe entre $0xFFFF8000$ et $0x00007FFF$. Il existe d'autres instructions MIPS qui utilisent des valeurs non-signées sur 16-bits, ce qui permet de travailler avec des valeurs allant de $0x0000$ à $0xFFFF$ mais ces instructions ne seront pas implémentées dans le cadre de notre processeur.

4.2.7 Unités de mémoire

(voir annexes, page xx pour la source.)

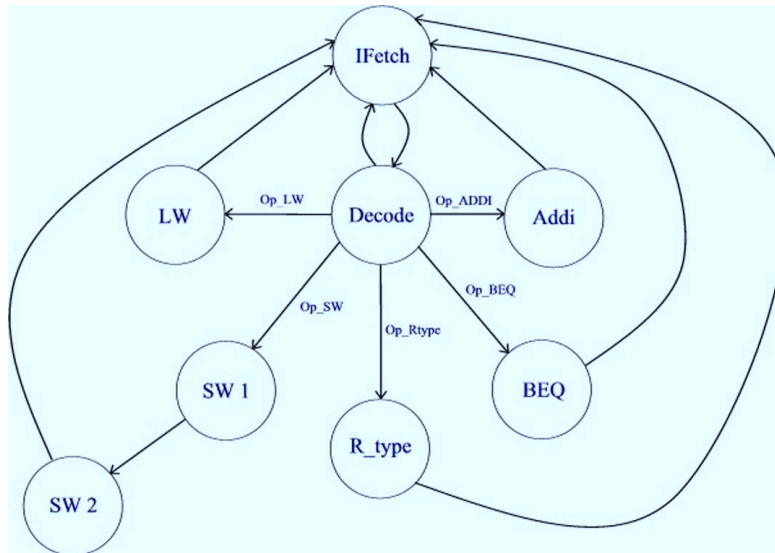
Comme dit dans les généralités, nous utilisons une architecture de “Harvard”, c’est-à-dire que les instructions sont stockées en ROM et les données en RAM. Il est impossible de briser cette structure, donc il est impossible de faire du code auto-modifiable. Nos mémoires peuvent contenir 256 mots chacun. Afin d’éviter l’accès aux adresses mal alignées, les deux bits de poids faible sont mis à zéro. Ainsi toutes les adresses sont multiples de 4. Etant donné que notre processeur travaille tout le temps en 32 bits (il n’y a pas de possibilité de travailler sur une unité plus petite), le problème de l’encodage au niveau de la RAM / ROM ne se pose pas (Little endian, Big endian).

La ROM est contrôlée par deux signaux : *cs* et *oe*, ils permettent d’activer la lecture d’instructions dans la mémoire. Il n’y a pas besoin de signal *we* car il est impossible d’écrire dans la ROM. Deux bus sont reliés à cette mémoire : le bus d’adresse et le bus de données. La RAM quant à elle est contrôlée par trois signaux, les deux premiers étant les mêmes que pour la ROM et le troisième *we* qui permet l’écriture de données. Ici trois bus sont reliés à la mémoire : deux bus de données (un pour l’entrée et l’autre pour la sortie) et un bus d’adresse. Ceci est inhabituel car d’habitude, le bus de données est utilisé dans les deux sens. Mais la simplicité de ce processeur permet de séparer le bus utilisé pour l’entrée du bus utilisé pour la sortie.

4.3 L'unité de contrôle

(voir annexes, page xvii pour la source.)

La figure ci-dessous représente le graphe des états que doit gérer notre unité de contrôle. Les flèches rouges représentent le retour à l'état *ifetch*, qui est effectué lorsque l'*op code* est invalide. Il est impossible de spécifier dans une instruction un registre qui n'existe pas, car la plus grande valeur possible correspond à 11111, donc 31 qui est le dernier registre. Cela peut paraître évident mais il est possible que pour certains processeurs (par exemple le 86x86) le problème se soit posé.



L'état 'IFetch' correspond la recherche de l'instruction en ROM. Le PC est incrémenté après cet état. Ensuite, l'état 'decode' va déterminer de quelle instruction il s'agit, en se servant de l'*op code* (ainsi que *funct* s'il s'agit d'un type-R). Sur la page suivante, nous pouvons voir un tableau récapitulatif de tous les signaux que doit gérer l'unité de contrôle pour chaque état du graphe.

	ALUCtrl	ALUSrc	IRWrite	MemdCS	MemdWE	MemiCS	MemtoReg	PCSrc	PCWrite	PCWriteCond	RegDst	RegWrite
ifetch			1		0	1		0	1	0		0
decode			0	0	0	0			0	0		0
addi	010	1	0		0	0	0		0	0	0	1
beq	110	0	0		0	0		1	0	1		0
r_type	*	0	0		0	0	0		0	0	1	1
lw	010	1	0	1	0	0	1		0	0	0	1
sw1	010	1	0		0	0			0	0		0
sw2			0	1	1	0			0	0		

* dépend de la valeur de *funct*.

4.4 Le test

Finalement nous avons testé notre processeur à l'aide de deux programmes. Le premier à été fourni avec la donnée de ce laboratoire. Il contient toutes les instructions sauf le *sw* ; mais ne fait rien de particulier. Le deuxième (voir annexes, page xxiii) que nous avons écrit nous-même calcule la liste des nombres premiers entre 2 et N. Ce programme un peu plus long utilise l'instruction *sw*.

5 Analyse

5.1 Problèmes rencontrés

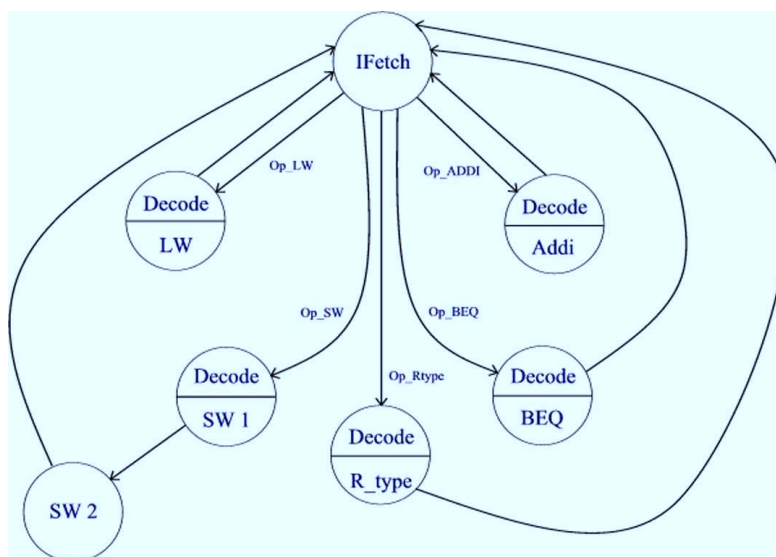
Nous n'avons pas rencontré de réel problème au niveau de la conception de ce processeur. Les principaux soucis nous ont été causé par les nouveaux outils. En effet, il nous a fallu un moment pour que nous puissions les maîtriser entièrement.

5.2 Optimisation

Une discussion complète sur les possibilités d'optimisation sortirait du cadre de ce document. Nous allons cependant faire un bref survol de ce sujet si important.

Il y a plusieurs moyens d'augmenter la vitesse d'un processeur. On peut optimiser l'utilisation des différentes ressources, en ajoutant par exemple un 'pipeline' qui va permettre d'utiliser les multiples ressources en même temps sur des données différentes. Une solution plus simple pour augmenter le *through output* consiste à utiliser des instructions plus longues mais qui demandent moins de coups d'horloge, ou d'avoir des instructions plus courtes mais qui demandent plus de coups d'horloge. Nous avons scinder l'état 'decode' qui n'utilise que très peu de ressources matérielles avec les états qui le suivent. Ceci a permis de réduire le nombre de coups d'horloge nécessaires.

Nous obtenons ainsi le graph des états suivant :



Il est possible d'optimiser encore plus et de n'avoir qu'un seul état 'SW'.

5.3 La réalité

Nous avons conçu un processeur qui fonctionne parfaitement. Cependant il est plutôt inutilisable, car il manque certaines instructions importantes :

Tout d'abord il faudrait implémenter les instructions *jal* et *jr*, qui permettraient l'utilisation des procédures. Sans les exceptions, il est très difficile de concevoir un système d'exploitation avec une interface utilisateur. On peut cependant se passer de certaines instructions; voici une alternative à l'instruction 'bgt' :

```

1      # bgt $s0, $s1, end_prog;
2      lw   $t0, 0($zero);           # mem[0] must contain
3                                          # the value 0x80000000
4      sub  $t1, $s1, $s0;
5      and  $t1, $t1, $t0;
6      beq  $t1, $t0, end_prog;

```

Mais il serait quand même utile d'implémenter un jeu complet d'instructions qui contiendraient entre autres : le non-logique (ou le nand), les instructions de manipulation sur les bits (rotations, décalage), les autres instructions de saut conditionnel ('bgt', 'bge', 'blt', 'ble'), etc... Il faudrait éventuellement plus de possibilités de calcul arithmétique (par exemple les instructions 'div', 'mult', ou une unité de virgule flottante.

6 Conclusion

Notre processeur n'a (malheureusement) aucune valeur commerciale, mais on pourrait cependant lui donner une particularité. Imaginons de char-

ger notre processeur sur une carte FPGA. Une FPGA est une carte reprogrammable. C'est assez inhabituel qu'une telle carte accueille un processeur, puisque ces cartes sont programmées pour des utilisations spécifiques à leur environnement (c'est l'intérêt même des cartes reprogrammables). Mais le fait d'avoir un mini-processeur embarqué, tout en gardant assez de place pour des ressources spécifiques aux besoins, peut être une approche intéressante. On arriverait donc à des cartes hybrides (une partie générique et une partie spécialisée). Nous pensons que les applications de telles cartes peuvent être assez larges.

Pour conclure, nous sommes très heureux d'avoir travaillé sur ce projet. La partie pratique a été grandement bénéfique, elle nous a permis de mieux assimiler les concepts vus pendant le cours. La rédaction du rapport nous a permis de nous familiariser avec l'environnement \LaTeX .

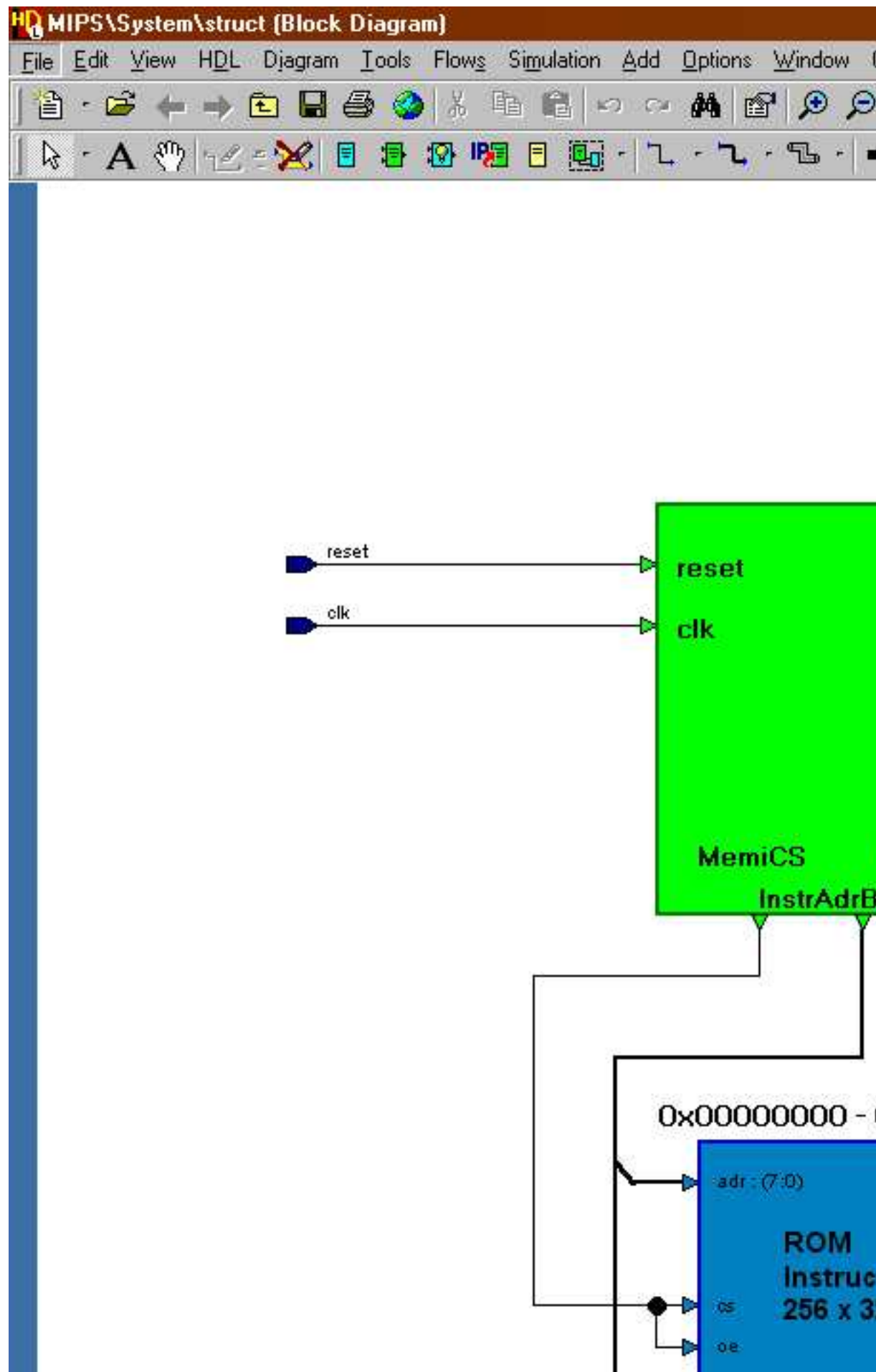
7 Bibliographie

Références

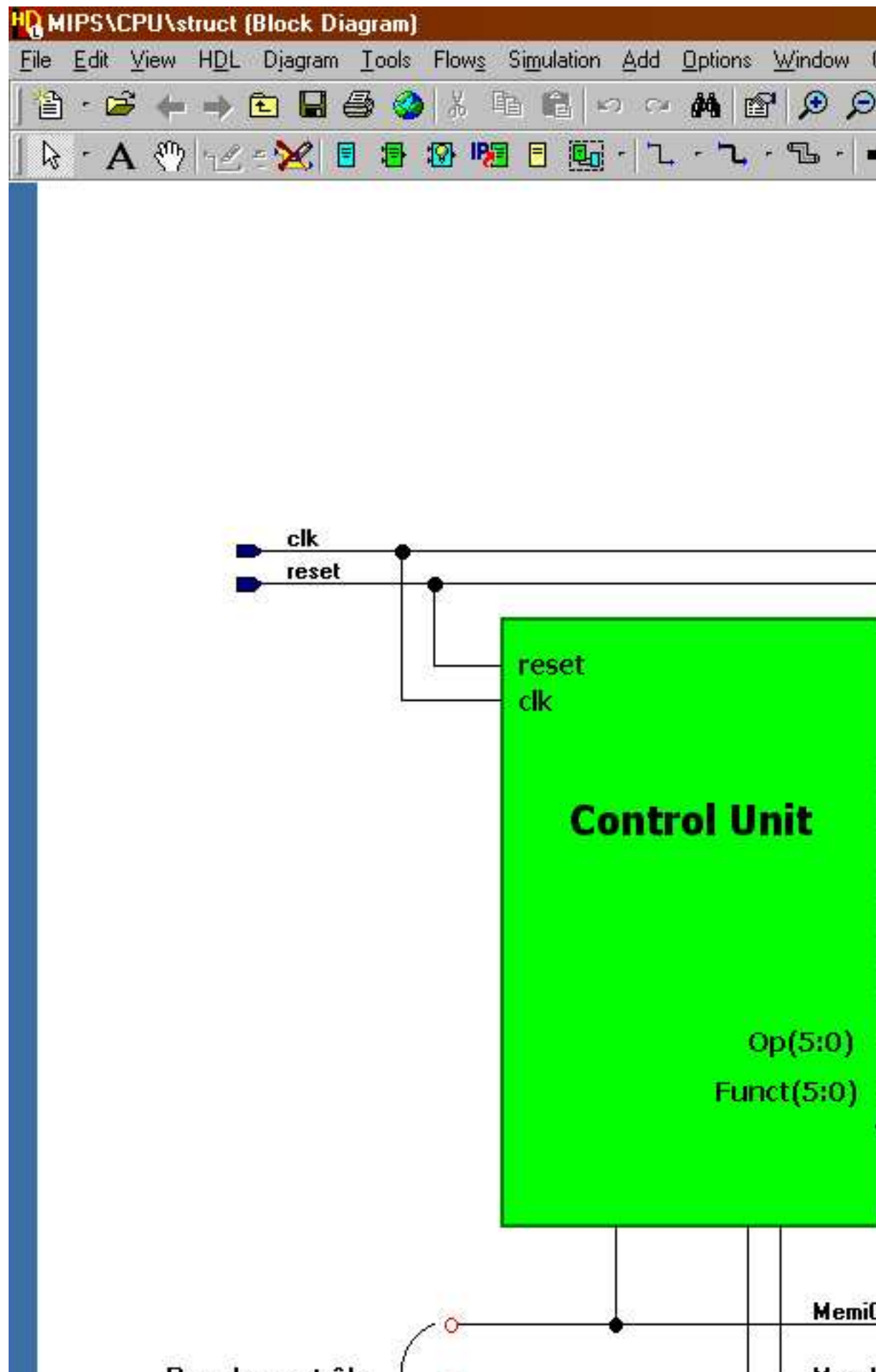
- [1] David A. Patterson and John L. Hennesy, *Computer Organization & Design*, Morgan Kaufmann, San Fransisco, second edition, 1998.
- [2] John F. Wakerly, *Digital Design*, Prentice Hall, New Jersey, third edition updated, 2001.
- [3] Prof. E. Sanchez et Prof. P. Ienne, *Cours de conception des processeurs et d'architecture des ordinateurs*, 2001-2002.

8 Annexes

8.1 Plans



Vue générale du processeur et des ses bus.

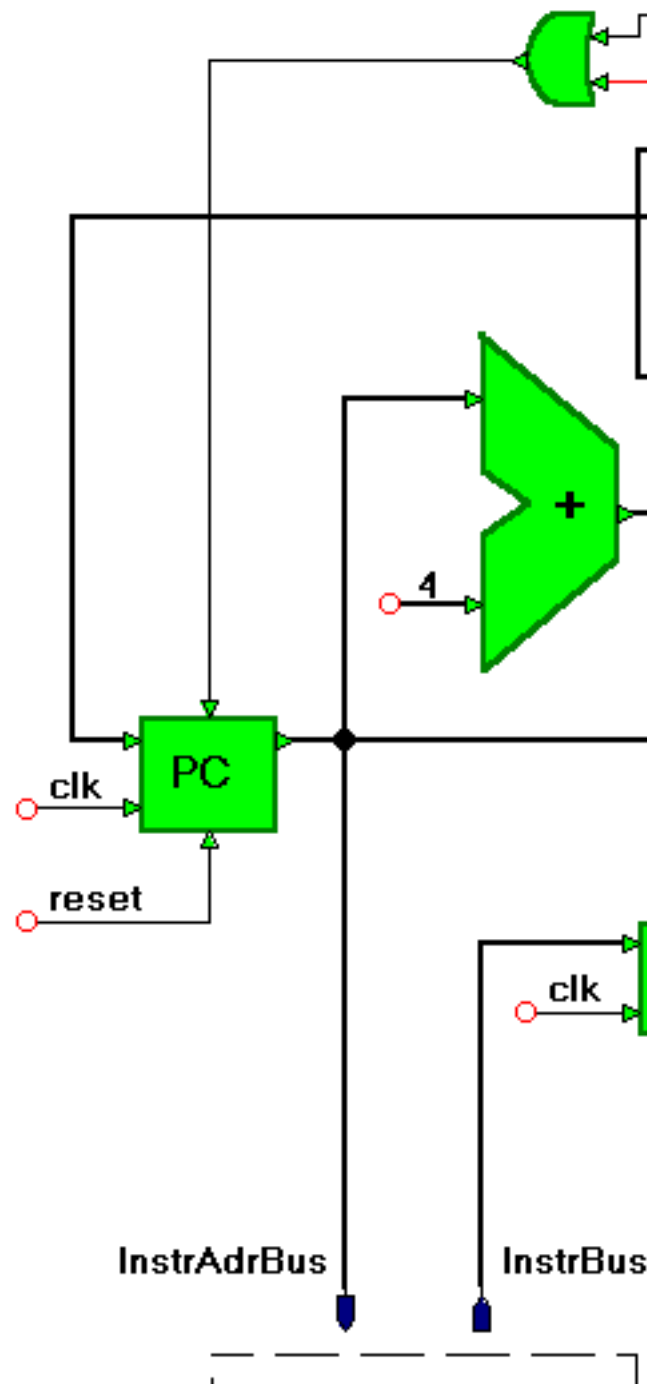
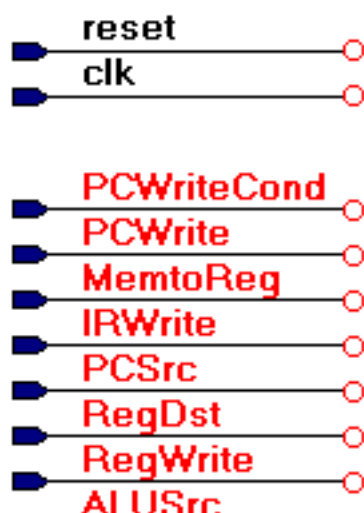


L'unité de contrôle qui contrôle les ressources.

```

Package List
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
    
```

Declarations



Les ressources ainsi que leurs interconnexions.

8.2 Codes sources

8.2.1 VHDL

Listing 1 – Add source

```
1 -- Simple 32 bits Adder.
2 -- Used to increment the PC.

4 LIBRARY ieee ;
5 USE ieee.std_logic_1164.all;
6 USE ieee.std_logic_arith.all;
7 USE ieee.std_logic_unsigned.all;

9 ENTITY add IS
10     PORT(
11         a      : IN      std_logic_vector (31 downto 0) ;
12         b      : IN      std_logic_vector (31 downto 0) ;
13         result : OUT     std_logic_vector (31 downto 0)
14     );
15 END add ;

17 ARCHITECTURE synth OF add IS
18 BEGIN
19     result <= a+b;    -- Use IEEE's math lib.
20 END synth;
```

Listing 2 – And2 source

```
1 -- Simple logical and (32 bits)
2 -- Used to determine wheter the conditional
3 -- branch is to be taken or not.

5 LIBRARY ieee ;
6 USE ieee.std_logic_1164.all;
7 USE ieee.std_logic_arith.all;

9 ENTITY And_2 IS
10     PORT(
11         In0  : IN      std_logic ;
12         In1  : IN      std_logic ;
13         Out0 : OUT     std_logic
14     );
15 END And_2 ;

17 ARCHITECTURE synth OF And_2 IS
18 BEGIN
19     Out0 <= In0 and In1;
20 END synth;
```

Listing 3 – Or2 source

```

1 -- Simple 32 bits OR. Used with the And2.vhdl.

3 LIBRARY ieee ;
4 USE ieee.std_logic_1164.all;
5 USE ieee.std_logic_arith.all;

7 ENTITY or_2 IS
8   PORT(
9     In0  : IN    std_logic ;
10    In1  : IN    std_logic ;
11    Out0 : OUT   std_logic
12  );
13 END or_2 ;

15 ARCHITECTURE synth OF or_2 IS
16 BEGIN
17   Out0 <= In0 or In1;    -- use IEEE's logical lib.
18 END synth;

```

Listing 4 – IR Source

```

1 -- IR.vhdl
2 -- Used to fetch instructions from ROM.

4 LIBRARY ieee ;
5 USE ieee.std_logic_1164.all;
6 USE ieee.std_logic_arith.all;

8 ENTITY IR IS
9   PORT(
10    a   : IN    std_logic_vector (31 DOWNTO 0) ;
11    q   : OUT   std_logic_vector (31 DOWNTO 0) ;
12    we  : IN    std_logic ;
13    clk : IN    std_logic
14  );
15 END IR ;

17 ARCHITECTURE synth OF IR IS
18 BEGIN
19   process(clk, we)
20   begin
21     if (clk='1' and clk'event) then
22       if (we='1') then
23         q<=a;    -- this is a pretty simple register, needs the
                   'we' signal to be activated
24                 -- in order to fetch the fresh data.
25       end if;
26     end if;
27   end process;
28 END synth;

```

Listing 5 – PC source

```

1  -- PC.vhdl
2  -- Program Counter. Register with a 'we' signal and a reset.

4  LIBRARY ieee ;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;

8  ENTITY pc IS
9      PORT(
10         clk      : IN      std_logic ;
11         a        : IN      std_logic_vector (31 downto 0) ;
12         q        : OUT     std_logic_vector (31 downto 0) ;
13         we       : IN      std_logic ;
14         reset    : IN      std_logic
15     );
16 END pc ;

18 ARCHITECTURE synth OF pc IS
19 BEGIN
20     process(clk , reset)
21     begin
22         if (reset='1') then
23             q<=(others=>'0');
24         elsif (clk'Event and clk='1') then
25             if (we='1') then
26                 q<=a;
27             end if;
28         end if;
29     end process;
30 END synth;

```

Listing 6 – Reg32 source

```

1  -- reg32.vhdl
2  -- This component is really STRANGE !!!
3  -- We think it's used nowhere...

5  LIBRARY ieee ;
6  USE ieee.std_logic_1164.all;
7  USE ieee.std_logic_arith.all;

9  ENTITY reg_32 IS
10     PORT(
11         clk : IN      std_logic ;
12         a   : IN      std_logic_vector (31 downto 0) ;
13         q   : OUT     std_logic_vector (31 downto 0)
14     );
15 END reg_32 ;

17 ARCHITECTURE synth OF reg_32 IS
18 BEGIN
19     process (clk)
20     begin
21         if (clk'Event and clk='1') then
22             q<=a;
23         end if;
24     end process;
25 END synth;

```

Listing 7 – ALU source

```

1  -- ALU.vhdl
2  -- Arithmetic Logic Unit.
3  -- This unit can perform logical operations (and, or)
4  -- and arithmetics ones (plus, minus).
5  --
6  -- The operation performed depends on op:
7  -- op -> result
8  -- 000 -> A and B
9  -- 001 -> A or B
10 -- 010 -> A + B
11 -- 110 -> A - B (Actually A+not(B)+1)
12 -- Since negative numbers are represented as a two's complement we
13 -- can perform the
14 -- subtraction as an addition. This really decreases the circuit
15 -- needed.

16 LIBRARY ieee ;
17 USE ieee.std_logic_1164.all;
18 USE ieee.std_logic_arith.all;
19 USE ieee.std_logic_unsigned.all;

21 ENTITY alu IS
22     PORT(
23         a      : IN    std_logic_vector (31 downto 0);
24         b      : IN    std_logic_vector (31 downto 0);
25         result : OUT   std_logic_vector (31 downto 0);
26         zero   : OUT   std_logic;
27         op     : IN    std_logic_vector (2  downto 0)
28     );
29 END alu ;

31 ARCHITECTURE synth OF alu IS
32     SIGNAL sig_b : std_logic_vector(31 downto 0);
33     SIGNAL sig_cin : std_logic;
34     SIGNAL sig_and, sig_or, sig_arith : std_logic_vec
35         tor(31 downto 0);
36     SIGNAL sig_result : std_logic_vector(31 downto 0);
37 BEGIN
38     process(op, b)
39     begin
40         -- sig_b is an intermediate signal, that will either contain the
41         -- value of b
42         -- or not(b) in case of a subtraction.
43         -- sig_cin allows us to have the +1 calculated directly, since it
44         -- can be interpreted
45         -- as the first carry.
46         if (op="110") then
47             sig_b <= not b;
48             sig_cin <= '1';
49         else
50             sig_b <= b;
51             sig_cin <= '0';
52         end if;
53     end process;
54     -- sig_and, sig_or and sig_arith are the intermediate results. We
55     -- could have avoided their use
56     -- by declaring result as type: buffer...
57     sig_and <= A and sig_b;
58     sig_or  <= A or  sig_b;
59     sig_arith <= A + sig_b + (("00000000000000000000000000000000") &

```

```
    sig_cin);
57 process(op, sig_and, sig_or, sig_arith)
58 begin
59     case op is
60         when "000" =>
61             sig_result <= sig_and;
62         when "001" =>
63             sig_result <= sig_or;
64         when "010" | "110" =>
65             sig_result <= sig_arith;
66         when others =>
67             sig_result <= "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU";
68     end case;
69 end process;
70
71                                     -- ... but we need the
72                                     result to set the zero
73                                     flag !
74
75 process(sig_result)
76 begin
77     if (sig_result="00000000000000000000000000000000") then
78         zero <= '1';
79     else
80         zero <= '0';
81     end if;
82 end process;
83 result <= sig_result;
84 END synth;
```

Listing 8 – Mux2 source

```

1  -- Simple multiplexor (32 bits).
2  -- the sel signal selects either e0 or e1 as
3  -- output signal.

4  LIBRARY ieee ;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;

9  ENTITY mux_2 IS
10     PORT(
11         e0  : IN      std_logic_vector (31 downto 0) ;
12         e1  : IN      std_logic_vector (31 downto 0) ;
13         o   : OUT     std_logic_vector (31 downto 0) ;
14         sel : IN      std_logic
15     );
16 END mux_2 ;

18 ARCHITECTURE synth OF mux_2 IS
19 BEGIN
20     process (sel, e0, e1)
21     begin
22         if (sel='0') then
23             o<=e0;
24         elsif (sel='1') then
25             o<=e1;
26         else
27             o<=(others=>'U');
28         end if;
29     end process;
30 END synth ;

```

Listing 9 – Mux2_5b source

```

1  -- mux2_5b.vhdl
2  -- Another multiplexor, this time with 5 bits.

4  LIBRARY ieee ;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;

8  ENTITY mux2_5b IS
9     PORT(
10         In0  : IN      std_logic_vector (4 DOWNTO 0) ;
11         In1  : IN      std_logic_vector (4 DOWNTO 0) ;
12         sel  : IN      std_logic ;
13         Out0 : OUT     std_logic_vector (4 DOWNTO 0)
14     );
15 END mux2_5b ;

17 ARCHITECTURE synth OF mux2_5b IS
18 BEGIN
19     process(sel, In0, In1)
20     begin
21         if (sel='0') then
22             Out0<=In0;
23         elsif (sel='1') then
24             Out0<=In1;
25         else
26             Out0<=(others=>'U');
27         end if;

```

```
28     end process ;  
29 END synth ;
```

Listing 10 – Mux4 source

```
1 -- mux4.vhdl
2 -- A more complex multiplexor that selects between
3 -- 4 signals. The code is pretty straight forward.
4 -- Another strange thing about this file: We don't know
5 -- if it's used in the final processor ??? Have we
6 -- missed out something ??? Was this a test for us ?

8 LIBRARY ieee ;
9 USE ieee.std_logic_1164.all;
10 USE ieee.std_logic_arith.all;
11 USE ieee.std_logic_unsigned.all;

13 ENTITY mux4 IS
14     PORT(
15         i0  : IN    std_logic_vector (31 downto 0) ;
16         i1  : IN    std_logic_vector (31 downto 0) ;
17         i2  : IN    std_logic_vector (31 downto 0) ;
18         i3  : IN    std_logic_vector (31 downto 0) ;
19         o   : OUT   std_logic_vector (31 downto 0) ;
20         sel : IN    std_logic_vector (1  downto 0) ;
21     );
22 END mux4 ;

24 ARCHITECTURE synth OF mux4 IS
25 BEGIN
26     process(sel, i0, i1, i2, i3)
27     begin
28         case sel is
29             when "00" => o<=i0 ;
30             when "01" => o<=i1 ;
31             when "10" => o<=i2 ;
32             when "11" => o<=i3 ;
33             when others => o<=(others=>'U') ;
34         end case ;
35     end process ;
36 END synth ;
```

Listing 11 – Regfile source

```

1  -- regfile
2  -- This is where the processor's register will be kept.

4  LIBRARY ieee ;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;
7  USE ieee.std_logic_unsigned.all;

9  ENTITY RegFile_32 IS
10     PORT(
11         clk      : IN      std_logic   ;
12         aa       : IN      std_logic_vector (4 downto 0) ;
13         ab       : IN      std_logic_vector (4 downto 0) ;
14         aw       : IN      std_logic_vector (4 downto 0) ;
15         a        : OUT     std_logic_vector (31 downto 0) ;
16         b        : OUT     std_logic_vector (31 downto 0) ;
17         WData    : IN      std_logic_vector (31 downto 0) ;
18         RegWrite : IN      std_logic
19     );
20 END RegFile_32 ;

22 ARCHITECTURE synth OF RegFile_32 IS
23     type mem_array is array(0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
24     signal reg : mem_array;
25 BEGIN
26     process (clk)
27     begin
28         if (clk'Event and Clk='1') then
29             if (RegWrite='1') then
30                 reg(conv_integer(aw))<=WData;           -- a little subtle, we
31                                                         -- vector to an int to
32                                                         -- access the array
33                                                         -- (which
34                                                         -- is in the native
35                                                         -- format).
36             end if;
37             reg(0)<=(others=>'0');                       -- Register zero is
38                                                         -- always nil.
39         end if;
40     end process;

41     a<=reg(conv_integer(aa));
42     b<=reg(conv_integer(ab));
43 END synth;

```

Listing 12 – Shiftleft2 source

```
1 -- Shift left by 2
2 -- Simple component to multiply
3 -- the input by four. We perform
4 -- a shift on the bits.

6 LIBRARY ieee ;
7 USE ieee.std_logic_1164.all;
8 USE ieee.std_logic_arith.all;

10 ENTITY shiftleft2 IS
11     PORT(
12         entree : IN      std_logic_vector (31 DOWNTO 0) ;
13         sortie : OUT     std_logic_vector (31 DOWNTO 0)
14     );
15 END shiftleft2 ;

17 ARCHITECTURE synth OF shiftleft2 IS
18 BEGIN
19     sortie<=entree(29 downto 0) & "00";
20 END synth;
```

Listing 13 – Signextend source

```
1 -- Sign extend
2 -- Component used to sign extend 16 bits (immediates)
3 -- to 32 bits (for the registers).
4 -- All we need to do is copy the MSB to the left.

6 LIBRARY ieee ;
7 USE ieee.std_logic_1164.all;
8 USE ieee.std_logic_arith.all;

10 ENTITY SignExtend IS
11     PORT(
12         entree : IN      std_logic_vector (15 downto 0) ;
13         sortie : OUT     std_logic_vector (31 downto 0)
14     );
15 END SignExtend ;

17 ARCHITECTURE synth OF SignExtend IS
18 BEGIN
19     g1: for b in 0 to 15 generate
20         sortie(b)<=entree(b);
21         sortie(b+16)<=entree(15);
22     end generate;
23 END synth;
```

Listing 14 – Control Unit source

```

1  --- Control unit.vhdl
2  --- THE INTERESTING PART...

4  LIBRARY ieee;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;

8  ENTITY ControlUnit IS
9      PORT(
10         Funct      : IN      std_logic_vector (5 DOWNTO 0);
11         Op         : IN      std_logic_vector (5 DOWNTO 0);
12         clk        : IN      std_logic;
13         reset      : IN      std_logic;
14         ALUctrl    : OUT     std_logic_vector (2 DOWNTO 0);
15         ALUSrc     : OUT     std_logic;
16         IRWrite    : OUT     std_logic;
17         MemdCS     : OUT     std_logic;
18         MemdWE     : OUT     std_logic;
19         MemiCS     : OUT     std_logic;
20         MemtoReg   : OUT     std_logic;
21         PCSrc      : OUT     std_logic;
22         PCWrite    : OUT     std_logic;
23         PCWriteCond : OUT    std_logic;
24         RegDst     : OUT     std_logic;
25         RegWrite   : OUT     std_logic
26     );
27 END ControlUnit ;

29 ARCHITECTURE synth OF ControlUnit IS
30     type state is (ifetch, decode, addi, beq, r_type, sw, sw2, lw);
31     SIGNAL next_state, current_state: state;
32 BEGIN
33     process(reset, next_state, current_state, Op, Funct)
34     begin
35         if (reset = '1') then
36             next_state<=ifetch;    -- The processor must start in this
37                 state.
38             PCWrite<='0';
39             IRWrite<='0';
40             MemiCS<='0';
41         else
42             IRWrite<='0';
43             MemiCS<='0';
44             MemdWE<='0';
45
46             PCWrite<='0';
47             PCWriteCond<='0';
48
49             case current_state is
50                 when ifetch =>    -- prepatate the processor to fetch the
51                     next
52                         IRWrite<='1';    -- instruction.
53                         MemiCS<='1';
54                         RegWrite<='0';
55                         PCWrite<='1';
56                         PCSrc<='0';
57                         next_state<=decode;
58                 when decode =>    -- using the instruction op code find
59                     the
60                         RegWrite<='0';    -- instruction to perform.
61                         MemdCS<='0';

```

```

59         case Op is
60             when "100011" =>
61                 next_state<=lw;
62             when "101011" =>
63                 next_state<=sw;
64             when "001000" =>
65                 next_state<=addi;
66             when "000100" =>
67                 next_state<=beq;
68             when "000000" =>
69                 next_state<=r_type;  -- all r-type instructions
60                                     are treated
70             when others =>          -- the same way.
71                 next_state<=ifetch; -- loop back to ifetch if
60                                     unknown...
72         end case;
73     when addi =>
74         ALUCtrl<="010";           -- Each instruction will
75         ALUSrc<='1';             -- set different signals
76         MemToReg<='0';          -- depending on the
60                                     resources
77         RegWrite<='1';          -- used.
78         RegDst<='0';
79         next_state<=ifetch;
80     when beq =>
81         ALUCtrl<="110";
82         ALUSrc<='0';
83         RegWrite<='0';
84         PCWriteCond<='1';
85         PCSrc<='1';
86         next_state<=ifetch;
87     when r_type =>
88         ALUCtrl<=Funct(1)&not(Funct(2))&Funct(0);
89         ALUSrc<='0';
90         MemToReg<='0';
91         RegWrite<='1';
92         RegDst<='1';
93         next_state<=ifetch;
94     when sw =>
95         RegWrite<='0';
96         ALUSrc<='1';
97         ALUCtrl<="010";
98         next_state<=sw2;
99     when sw2 =>
100         MemdWE<='1';
101         MemdCS<='1';
102         next_state<=ifetch;
103     when lw =>
104         MemToReg<='1';
105         RegWrite<='1';
106         RegDst<='0';
107         MemdCS<='1';
108         ALUSrc<='1';
109         ALUCtrl<="010";
110         MemdWE<='0';
111         next_state<=ifetch;
112     end case;
113 end if;
114 end process;
115 process (clk)
116 begin
117     if (clk='1' and clk'Event and reset='0') then

```

```
118         current_state<=next_state;      --- Refresh the state machine
119     end if;
120 end process;
121 END synth;
```

Listing 15 – RAM source

```
1 -- Mem data.vhdl
2 -- Simple array of 32 bits used to store the data.
3 -- The first two words are set to specific values
4 -- for the test program.

6 LIBRARY ieee ;
7 USE ieee.std_logic_1164.all;
8 USE ieee.std_logic_arith.all;
9 USE ieee.std_logic_unsigned.all;

11 ENTITY memdata IS
12     PORT(
13         adr       : IN       std_logic_vector (7 DOWNTO 0) ;
14         Data_Out  : OUT      std_logic_vector (31 DOWNTO 0) ;
15         cs        : IN       std_logic       ;
16         we        : IN       std_logic       ;
17         Data_In   : IN       std_logic_vector (31 DOWNTO 0) ;
18         oe        : IN       std_logic
19     );
20 END memdata ;

22 ARCHITECTURE synth OF memdata IS
23     type memdtype is array (((2**8)/4)-1 downto 0) of std_logic_vector
24         (31 downto 0);
25     signal memoired : memdtype := (0 => "
26         000000000000000000000000000000100",
27         1 => "0000000000000000000000000000000011",
28         others => "0000000000000000000000000000000000000000"
29     );
30 BEGIN
31     process(cs, oe, we, adr, Data_In)
32     begin
33         if (oe='0') then
34             Data_Out<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
35         end if;
36         if (we='0') then
37             -- Cycle de lecture
38             if (oe='1' and cs='1') then
39                 Data_Out<=memoired(conv_integer(adr));
40             end if;
41         else
42             -- Cycle d'écriture
43             if (cs='1') then
44                 memoired(conv_integer(adr))<=Data_In;
45             end if;
46         end if;
47     end process;
48 END synth;
```

Listing 16 – ROM source

```

1  --- meminstr.vhdl
2  --- Same as RAM (memdata.vhdl)
3  --- Except is read only.

5  LIBRARY ieee ;
6  USE ieee.std_logic_1164.all;
7  USE ieee.std_logic_arith.all;
8  USE ieee.std_logic_unsigned.all;

11 ENTITY meminstr IS
12   PORT(
13     adr   : IN    std_logic_vector (7 DOWNTO 0);
14     data  : OUT   std_logic_vector (31 DOWNTO 0);
15     cs    : IN    std_logic;
16     oe    : IN    std_logic
17   );
18 END meminstr ;

20 ARCHITECTURE synth OF meminstr IS
21   type memitype is array(((2**8)/4)-1 downto 0) of std_logic_vector
22     (31 downto 0);
23   signal memoirei : memitype := (
24     -- add $s0, $zero, $zero
25     -- R-type:000000 i $zero:00000 i $zero:00000 i $s0:10000 i shamt
26     :00000 i funct:100000
27     0 => "00000000000000001000000000100000", -- 00008020H
28     -- addi $s0, $s0, 0x0000
29     -- Addi:001000 i $s0:10000 i $s0:10000 i imm:0000000000000000
30     1 => "00100010000100000000000000000000", -- 22100000H
31     -- lw $s1, 0x0000($s0) ; $s1 <- 4
32     -- lw:100011 i $s0:10000 i $s1:10001 i address:0000000000000000
33     2 => "10001110000100010000000000000000", -- 8E110000H
34     -- lw $s2, 0x0004($s0) ; $s2 <- 3
35     -- lw:100011 i $s0:10000 i $s2:10010 i address:0000000000000100
36     3 => "10001110000100100000000000000100", -- 8E120004H
37     -- add $s3, $s1, $s2 ; $s3 <- 7
38     -- R-type:000000 i $s1:10001 i $s2:10010 i $s3:10011 i shamt
39     :00000 i funct:100000
40     4 => "00000010001100101001100000100000", -- 02329820H
41     -- sub $s0, $s1, $s2 ; $s0 <- 1
42     -- R-type:000000 i $s1:10001 i $s2:10010 i $s0:10000 i shamt
43     :00000 i funct:100010
44     5 => "00000010001100101000000000100010", -- 02328022H
45     -- L1:
46     -- sub $s3, $s3, $s0 ; $s3 <- $s3 - 1
47     -- R-type:000000 i $s3:10011 i $s0:10000 i $s3:10011 i shamt
48     :00000 i funct:100010
49     6 => "00000010011100001001100000100010", -- 02709822H
50     -- beq $s3, $zero, L2
51     -- Beq:000100 i $s3:10011 i $zero:00000 i address
52     :0000000000000001
53     7 => "00010010011000000000000000000001", -- 22600001H
54     -- beq $zero, $zero, L1
55     -- beq:000100 i $zero:00000 i $zero:00000 i address
56     :1111111111111101
57     8 => "00010000000000000111111111111101", -- 1000FFFDH
58     --L2:
59     -- beq $zero, $zero, L2
60     -- beq:000100 i $zero:00000 i $zero:00000 i address
61     :1111111111111111

```


8.2.2 Assembleur

Listing 17 – Programme de test 2

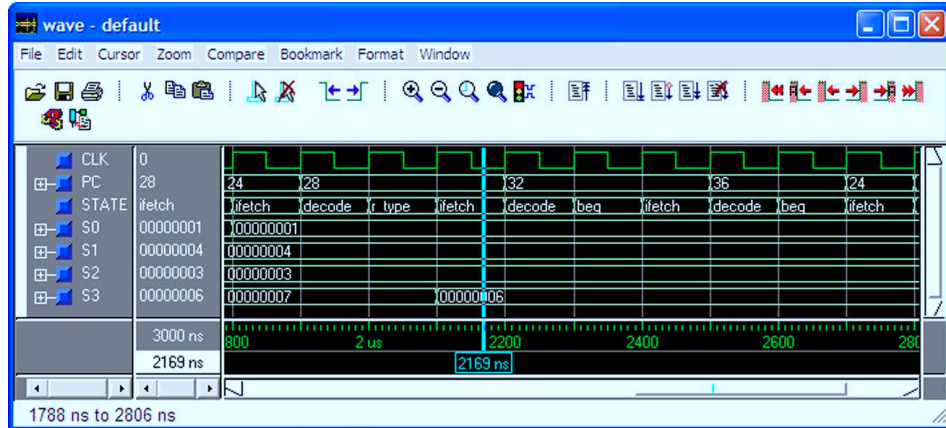
```

1 # Find all the prime numbers between [2 and N[
2 #
3 # Programmed by Alok Menghrajani & Alex Lopes
4 # Written for our MIPS processor (only 8 instructions available)
5 #
6 # The algorithm used is based on the Sieve of Eratosthenes.
7 # No assumptions made about initial values in registers,
8 # except that $zero=0.
9 # Memory assumed to be all 00's.
10 # N is stored in $a0.

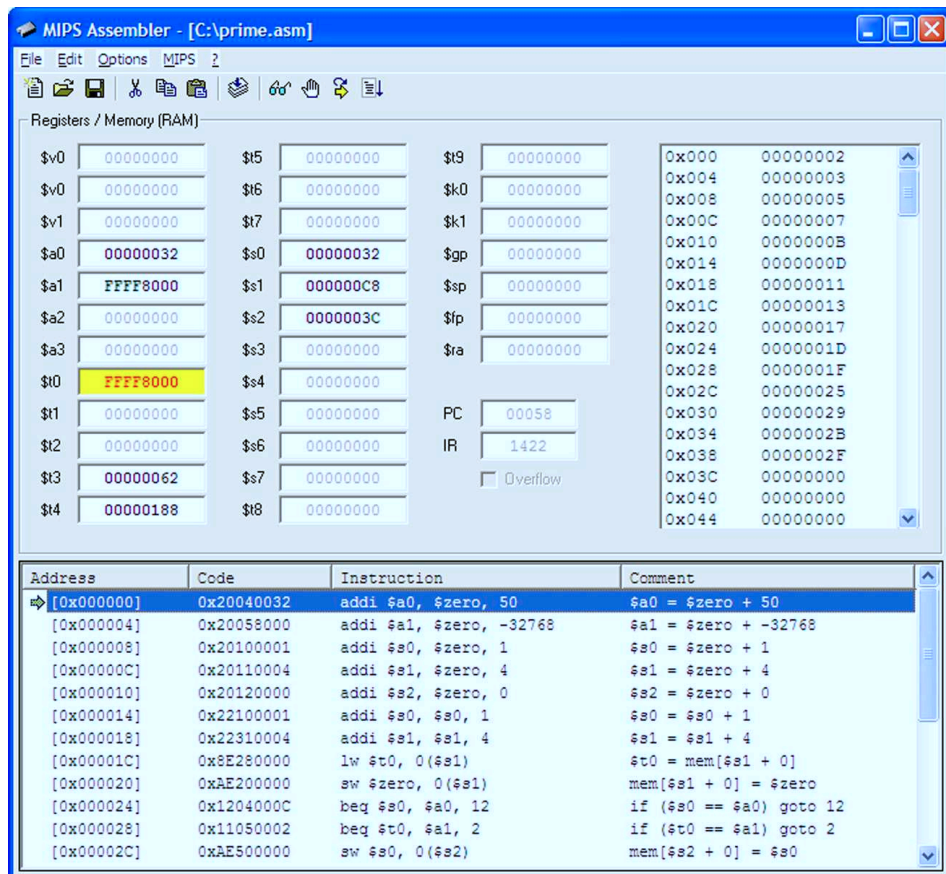
12         addi $a0, $zero, 50;           # Place N in $a0
13         addi $a1, $zero, 0x8000; # BGT and marker
14         addi $s0, $zero, 1;
15         addi $s1, $zero, 4;           # Table PTR
16         addi $s2, $zero, 0;           # Final PTR
17 seek:    addi $s0, $s0, 1;
18         addi $s1, $s1, 4;
19         lw   $t0, 0($s1);             # Find prime
20         sw   $zero, 0($s1);          # No garbage
21         beq  $s0, $a0, end_prog;
22         beq  $t0, $a1, not_prime;    # S0 is prime
23         sw   $s0, 0($s2);
24         addi $s2, $s2, 4;
25 not_prime: addi $t3, $s0, 0;
26         addi $t4, $s1, 0;
27 mark_loop: add $t3, $t3, $s0;
28         add  $t4, $t4, $s1;
29         sub  $t0, $a0, $t3;          # BGT
30         and  $t0, $t0, $a1;
31         beq  $t0, $a1, seek;
32         sw   $a1, 0($t4);
33         beq  $zero, $zero, mark_loop;
34 end_prog: beq  $zero, $zero, end_prog;

```

8.3 Copies d'écran



Exécution du premier programme de test avec ModelSim.



Exécution du programme qui trouve les nombres premiers sur Mips Assembleur. On voit en haut à droite, les nombres premiers compris entre 2 et 50.