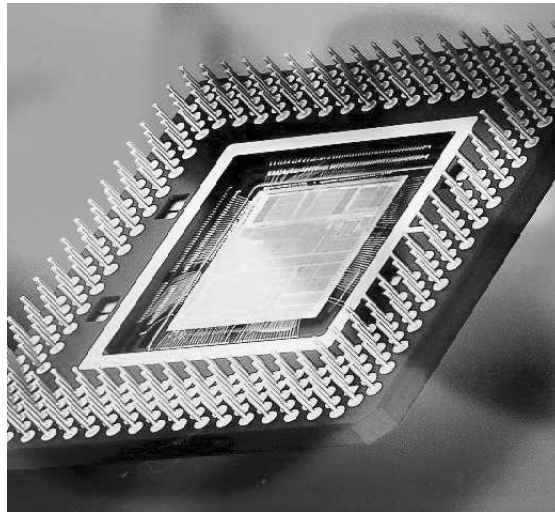


Architecture des Ordinateurs

Processeur MIPS

Rapport



Xavier Perseguers, xavier.perseguers@epfl.ch
Tadeusz Senn, jeantadeusz.senn@epfl.ch

1er mai 2002



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Processeur MIPS – BB

Introduction

Ce rapport présente la réalisation d'un processeur MIPS 32 bits en VHDL. Le processeur, surnommé BB, a été créé avec un petit nombre d'instructions (arithmétiques : *ADD*, *SUB* et *ADDI* ; logiques : *AND* et *OR* ; de saut conditionnel : *BEQ* ; et d'accès à la mémoire : *SW* et *LW*). La technique du pipelining n'a pas été abordée.

Il décrit la méthodologie utilisée pour implémenter le processeur, les tests effectués pour vérifier son bon fonctionnement, les problèmes rencontrés et leurs solutions, ainsi que certaines améliorations possibles.

Méthodologie

La méthodologie choisie a été de créer les composants et de les tester au fur et à mesure.

Nous utilisons au maximum la bibliothèque arithmétique. De cette façon, notre approche est plus comportementale que structurelle. C'est aussi pourquoi nous préférons ne pas réutiliser les composants développés séparément.

Unité de contrôle

L'unité de contrôle est une machine de Moore, *i.e.* une machine dont l'état futur ne dépend que de l'état présent. Elle est composée de sept états organisés selon la figure 1 :

Fetch : lecture de l'instruction suivante (*IRWrite* et *MemiCS* à 1), incrémentation du *Program Counter* (PC) (*PCWrite* activé, et *PCSrc* désactivé pour que le PC soit incrémenté de la valeur par défaut 4, *i.e.* $PC = PC + 4$ comme dans tous les états où il est désactivé).

Decode : détermination de l'état futur en fonction de l'instruction lue dans l'état *Fetch*. De plus, si l'instruction est un *SW*, préparation de l'adresse en désactivant *ALUSrc* (explications dans la section Problèmes et Solutions, en page 6).

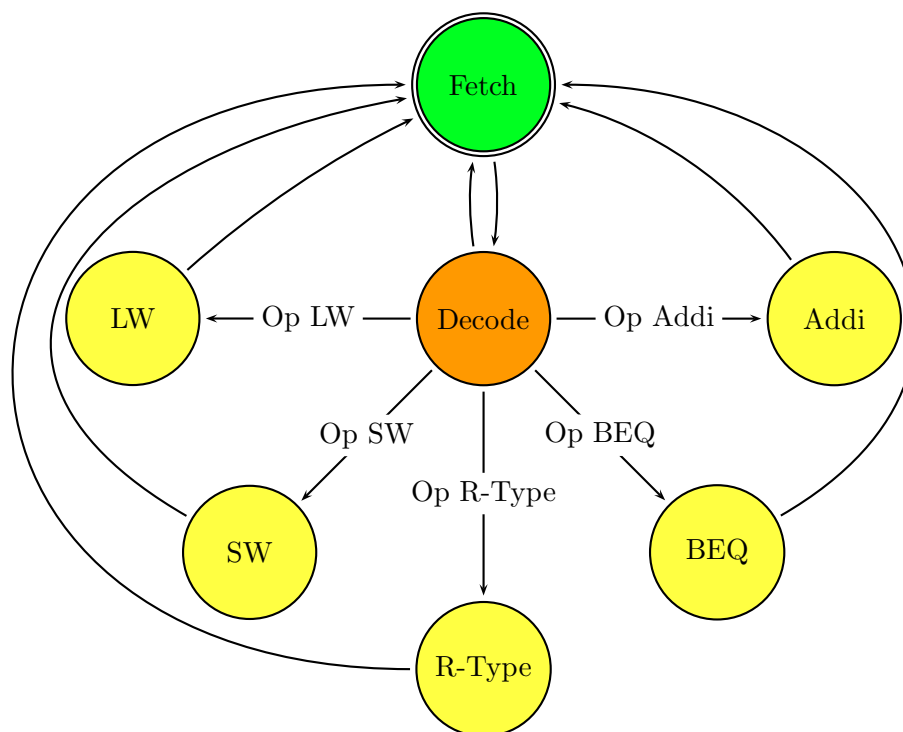


FIG. 1 – Graphe des états de l'unité de contrôle

Les états suivants sont des opérations :

Addi

ALUCtrl : signal à *010* pour effectuer l'addition avec l'ALU.

ALUSrc : activé (signal à *1*) pour sélectionner la valeur immédiate comme l'une des sources de l'addition.

RegWrite : activé pour enregistrer le résultat dans un registre.

RegDst : désactivé (signal à *0*) pour prendre l'adresse du registre destination *RT* venant des bits 20 à 16 de l'instruction.

BEQ

ALUCtrl : signal à *110* pour effectuer une soustraction avec l'ALU.

La soustraction permet de comparer la valeur des deux registres.

Si le résultat est nul, le flag *zero* de l'ALU sera activé et provoquera la mise à jour du PC.

PCSrc : activé pour remplacer la valeur du PC par la sortie de l'additionneur calculant le *Branch Target*.

PCWriteCond : activé pour écrire dans le PC pour autant que le bit *zero* de l'ALU soit aussi activé.

ALUSrc : désactivé pour que la seconde sortie du *Register File* arrive sur l'ALU.

RegDst : désactivé pour prendre l'adresse du registre destination *RT* venant des bits 20 à 16 de l'instruction.

SW

ALUCtrl : signal à *010* pour effectuer une addition avec l'ALU. L'addition permet de calculer l'adresse mémoire en combinant l'adresse stockée et l'offset.

ALUSrc : activé pour prendre comme seconde opérande de l'ALU les 16 bits de poids faible (*sign extended*), c'est-à-dire la partie *offset* de l'instruction.

MemdCS : activé pour sélectionner la mémoire.

MemdWE : activé pour remplacer le contenu de la mémoire désigné par l'adresse par le contenu de *DataIn*.

RegDst : désactivé pour prendre l'adresse du registre destination *RT* venant des bits 20 à 16 de l'instruction.

LW

ALUCtrl : signal à *010* pour calculer l'adresse mémoire comme dans l'état **SW**.

ALUSrc : activé comme dans l'état **SW**.

MemdCS : activé comme dans l'état **SW**.

MemtoReg : activé pour copier le contenu de la mémoire dans un registre.

RegWrite : activé pour autoriser l'écriture dans le registre.

RegDst : désactivé pour prendre l'adresse du registre destination *RT* venant des bits 20 à 16 de l'instruction.

R-Type

ALUCtrl : déterminé par les bits *Funct* de l'instruction. Permet de choisir l'opération arithmétique ou logique à effectuer.

RegDst : activé pour signifier que le registre de destination du résultat de l'opération est déterminé par les bits 15 à 11 de l'instruction (partie *RD*).

RegWrite : activé comme dans l'état **LW**.

ALUSrc : désactivé pour que la seconde sortie du *Register File* arrive sur l'ALU.

Donner une valeur initiale à tous les signaux sortants, y compris l'état futur, empêche la synthèse de latches et simplifie le code. De plus, cela permet au circuit d'être dans un état stable par défaut.

	Decode							R-Type (ALUCtrl[2..0]/Funct[5..0])			
	Fetch	SW	Autres	Addi	Beq	SW	LW	Add	Sub	And	Or
ALUCtrl[2..0]				010	010	010	110	010/100000	110/100010	000/100100	001/100101
ALUSrc		1	0	1	0	1	1	0	0	0	0
IRWrite	1										
MemdCS						1	1				
MemdWE						1					
MemiCS	1										
MemtoReg							1				
PCSrc					1						
PCWrite	1										
PCWriteCond					1						
RegDst				0	0	0	0	1	1	1	1
RegWrite				1			1	1	1	1	1
Etat suivant	Decode				Fetch						

TAB. 1 – Synthèse des signaux de l'unité de contrôle — inspiré de la figure 5.20 p. 361 du COD.

Tests Effectués

La première vérification du bon fonctionnement a été faite avec le programme proposé dans la mémoire d'instructions. Ensuite, les opérations logiques (*AND* et *OR*) ainsi que l'accès en écriture à la mémoire (*SW*) ont été rajoutés parce qu'ils n'étaient pas testés dans le programme original.

Enfin, nous avons écrit un programme calculant le produit de deux facteurs initialement stockés en mémoire (cf. listing 1). Le programme est structuré comme une procédure. Les adresses des deux facteurs sont stockées respectivement dans *\$a0* et *\$a1*, et celle de retour, dans *\$v0*. La première partie permet d'initialiser le contenu de la mémoire (*memdata*); la seconde effectue la multiplication, basée sur sa définition mathématique :

$$a \cdot b = \sum_1^a b$$

Listing 1: *Programme de multiplication : 6 · 8*

```

1  # Initialization

3  addi  $s0, $zero, 6           # $s0 = 6
4  addi  $s1, $zero, 8           # $s1 = 8
5  add   $a0, $zero, $zero       # $a0 = 0
6  addi  $a1, $a0, 4             # $a1 = 4
7  addi  $v0, $a1, 4             # $v0 = 8
8  sw    $s0, 0($a0)             # mem[$a0] = $s0
9  sw    $s1, 0($a1)             # mem[$a1] = $s1

11 # Multiplication

13 lw    $s0, 0($a0)             # $s0 = operand #1
14 lw    $s1, 0($a1)             # $s1 = operand #2
15 add   $s2, $zero, $zero       # $s2 : temp result
16 add   $s3, $zero, $zero       # $s3 : loop counter
17 Loop:
18 add   $s2, $s2, $s1           # $s2 = $s2 + $s1
19 addi  $s3, $s3, 1             # $s3 = $s3 + 1
20 beq   $s3, $s0, End           # if $s3=$s1 GOTO End
21 beq   $zero, $zero, Loop      # GOTO Loop
22 End:
23 sw    $s2, 0($v0)            # mem[$v1] <= result

25 Eternity:
26 beq   $zero, $zero, Eternity  # infinite loop

```

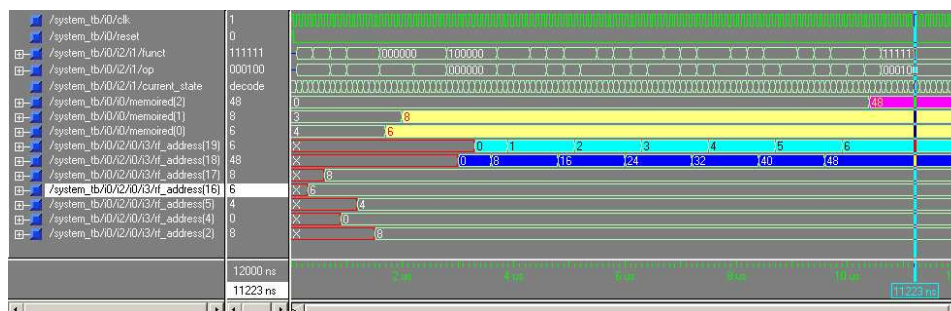


FIG. 2 – Déroulement du programme

Problèmes et Solutions

Le premier problème est apparu lors de la simulation : un signal avait une valeur 'X' alors que le programme n'utilisait que des '0' et des '1'. Nous avons initialisé les signaux dans le processus Reset de l'unité de contrôle (processus propre) et au début du processus principal. Or l'un des signaux avait une autre assignation dans l'état *Fetch*. Comme solution, seul l'état est remis à *Fetch* dans le Reset, et tous les signaux sont initialisés à chaque coup d'horloge dans le processus principal.

Dans l'ALU, le flag *zero* doit être activé quand un résultat est nul. Comme le résultat temporaire (dans notre cas *tp_result*) est défini comme un signal, il est indispensable de définir la valeur de ce flag à l'extérieur du processus de calcul sous peine d'avoir un décalage d'un coup d'horloge entre le nouveau résultat et son activation éventuelle.

L'opération de saut conditionnel *BEQ* peut être effectuée, théoriquement, par les opérations *AND* ou *SUB* de l'ALU. En pratique, le signal *zero* est indispensable pour que la valeur du PC puisse être différente de $PC = PC + 4$; d'où le choix de la soustraction qui active ce signal si les opérandes sont égales.

Le dernier problème s'est posé avec l'instruction *SW*. En effet, lors des premiers tests et comme nous nous sommes basés tout d'abord sur le graphe des états du cours (cf. figure 1, p. 2), l'instruction commençait par écrire n'importe où en mémoire avant de le faire à la bonne adresse. En examinant le schéma de l'unité de traitement, il est apparu que l'adresse n'est valide qu'après son calcul dans l'ALU, lequel requière un coup d'horloge supplémentaire par rapport au signal de sélection en écriture de la mémoire (*MemdWE*). Ainsi, il était envisageable d'ajouter un état intermédiaire, entre les états *Decode* et *SW*, pour laisser du temps à l'ALU. Toutefois nous avons trouvé plus astucieux de préparer l'adresse dans l'état *Decode*.

Améliorations

Le design du processeur mis à part, des améliorations au niveau des performances et des possibilités intrinsèques sont envisageables.

Actuellement, toutes les instructions demandent, pour s'exécuter, trois coups d'horloge pendant lesquels une partie du processeur n'est pas utilisée. Il serait intéressant d'implémenter le pipelining pour réduire le temps total d'exécution et avoir une meilleure gestion des ressources du processeur. Le pipelining permet de commencer à exécuter une nouvelle instruction alors que la précédente est encore traitée. Il faut évidemment remarquer qu'une instruction de saut ou un test empêchent l'application de cette technique. Le nombre de coups d'horloge pour une instruction n'est pas réduit, mais comme les opérations se chevauchent, le temps total d'exécution d'une séquence d'instructions est diminué.

Le Processeur MIPS – \mathbb{B} a un jeu d'instructions quelque peu restreint. Des instructions telles que des décalages (*SLL* et *SRL*), un autre comparateur (*SLT*) et une nouvelle opération logique (*NOR*) augmenteraient de façon significative les possibilités de programmation. Les décalages permettent des multiplications et divisions par 2; le comparateur '*<*' complète les instructions de sauts conditionnels; et le *NOR* permet de calculer toutes les autres fonctions logiques, en particulier la fonction *NOT*.

Enfin, la gestion des exceptions permettrait de détecter les dépassements de capacité dans les opérations arithmétiques ainsi que les instructions non reconnues ou incorrectes.

Conclusion

17.04.2002 11h49: ÇA MARCHE!!!

Pour réaliser le *Processeur MIPS – \mathbb{B}* , les solutions choisies — par exemple les bibliothèques arithmétiques — nous sont apparues comme évidentes du fait de leur simplicité et de leur logique. De plus, plutôt que de coder les instructions utilisées pour la multiplication en binaire, nous avons préféré créer, en informaticiens, le programme MIPS Assembler¹.

Finalement, ce projet nous a permis de nous familiariser avec la logique d'un processeur, son architecture et son fonctionnement. Nous espérons garder en mémoire les quelques subtilités découvertes pour pouvoir les exploiter dans notre avenir en ingénierie informatique.

1. Disponible sur <http://diwww.epfl.ch/~persegue/mips/>

Références

- [1] D.A. PATTERSON et J.L. HENNESSY. *Computer Organization & Design, 2nd Edition*. Morgan Kaufmann Publishers, 1998.
- [2] E. SANCHEZ et P. IENNE. Notes du cours *Conception des Processeurs*, 2001-2002.
- [3] E. SANCHEZ et P. IENNE. Notes du cours *Architecture des Ordinateurs*, 2002.
- [4] B. ORWELL. *Nineteen Eighty-Four*. Penguin Books, 1948.

Annexes

Fichiers VHDL

add_synth.vhd	10
alu_synth.vhd	11
and_2_synth.vhd	13
controlunit_synth.vhd	14
ir_synth.vhd	17
memdata_synth.vhd	18
meminstr_synth.vhd	20
mux_2_synth.vhd	22
or_2_synth.vhd	23
pc_synth.vhd	24
reg_32_synth.vhd	25
regfile_32_synth.vhd	26
shiftright2_synth.vhd	28
signextend_synth.vhd	29

Listing 2: *add_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;
3  USE ieee.std_logic_arith.all ;
4  USE ieee.std_logic_unsigned.all ;

6  ENTITY add IS
7      PORT(
8          a      : IN      std_logic_vector (31 downto 0);
9          b      : IN      std_logic_vector (31 downto 0);
10         result : OUT     std_logic_vector (31 downto 0)
11     );
12 END add;

14 — Simple opération arithmétique (addition)
15 — utilisant la bibliothèque arithmétique

17 ARCHITECTURE synth OF add IS
18 BEGIN
19     result <= a + b;
20 END synth;
```

Listing 3: *alu_synth.vhd*

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_arith.all;
4  USE ieee.std_logic_unsigned.all;

6  ENTITY alu IS
7      PORT(
8          a      : IN      std_logic_vector (31 downto 0);
9          b      : IN      std_logic_vector (31 downto 0);
10         result : OUT     std_logic_vector (31 downto 0);
11         zero   : OUT     std_logic;
12         op     : IN      std_logic_vector (2  downto 0)
13     );
14 END alu;

16 ARCHITECTURE synth OF alu IS
17     SIGNAL tp_result : std_logic_vector(31 downto 0);
18 BEGIN

20     -- Processus principal décodant le signal op.
21     -- Choix de 'U' pour des opérations non reconnues

23     PROCESS (op, a, b)
24     BEGIN
25         CASE op IS
26             WHEN "000" => tp_result <= a AND b;
27             WHEN "001" => tp_result <= a OR b;
28             WHEN "010" => tp_result <= a + b;
29             WHEN "110" => tp_result <= a - b;
30             WHEN others => tp_result <= (OTHERS => 'U');
31         END CASE;
32     END PROCESS;

34     -- Gestion du flag zero

36     PROCESS (tp_result)
37     BEGIN
38         IF (tp_result = (31 downto 0 => '0')) THEN
39             zero <= '1';
40         ELSE
41             zero <= '0';
42         END IF;
```

```
43  END PROCESS;

45  -- Passage par un signal temporaire pour que la
46  -- sortie result ait toujours une valeur

48  result <= tp_result;

50 END synth;
```

Listing 4: *and_2_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;
3  USE ieee .std_logic_arith .all ;

5  ENTITY And_2 IS
6      PORT(
7          In0   : IN   std_logic ;
8          In1   : IN   std_logic ;
9          Out0  : OUT  std_logic
10         );
11 END And_2 ;

13 — Simple opération logique (ET logique)
14 — utilisant la bibliothèque arithmétique

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

Listing 5: *controlunit_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;

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

25 ARCHITECTURE synth OF ControlUnit IS

27     — déclaration des 7 états
28     type State_Type is (Fetch , Decode , Addi , Beq ,
29         RType , SW , LW);
30     signal current_state : State_Type;
31     signal next_state : State_Type;

33 BEGIN

35     PROCESS ( current_state , op , Funct)
36     BEGIN

38         — Initialisation de tous les signaux
39         — à chaque changement d'état

41         ALUctrl    <= (OTHERS => '0');
42         ALUSrc     <= '0';
```

```

43     IRWrite      <= '0';
44     MemdCS       <= '0';
45     MemdWE       <= '0';
46     MemiCS       <= '0';
47     MemtoReg     <= '0';
48     PCSrc        <= '0';
49     PCWrite      <= '0';
50     PCWriteCond  <= '0';
51     RegDst       <= '0';
52     RegWrite     <= '0';
53     next_state   <= Fetch;

55     CASE current_state IS

57         -- Fetch : Charger l'instruction suivante

59         WHEN Fetch =>
60             IRWrite <= '1';
61             PCWrite <= '1';
62             MemiCS <= '1';
63             next_state <= Decode;

65         -- Decode : Décoder l'instruction
66         --           Pour SW, calculer l'adresse en plus

68         WHEN Decode =>
69             CASE Op IS
70                 WHEN "001000" => next_state <= Addi;
71                 WHEN "000100" => next_state <= Beq;
72                 WHEN "000000" => next_state <= RType;
73                 WHEN "101011" =>
74                     ALUctrl <= "010";
75                     ALUSrc <= '1';
76                     MemdCS <= '1';
77                     next_state <= SW;
78                 WHEN "100011" => next_state <= LW;
79                 WHEN OTHERS => next_state <= Fetch;
80             END CASE;

82         -- Opérations

84         WHEN Addi =>
85             ALUctrl <= "010";
86             ALUSrc <= '1';

```

```

87         RegWrite <= '1';
88     WHEN Beq =>
89         -- Soustraction pour un saut
90         ALUctrl <= "110";
91         PCSrc <= '1';
92         PCWriteCond <= '1';
93     WHEN RType =>
94         -- Choix de l'opération de type R-type
95         CASE Funct IS
96             WHEN "100000" => ALUctrl <= "010";
97             WHEN "100010" => ALUctrl <= "110";
98             WHEN "100100" => ALUctrl <= "000";
99             WHEN "100101" => ALUctrl <= "001";
100            WHEN OTHERS => ALUctrl <= "UUU";
101         END CASE;
102         RegDst <= '1';
103         RegWrite <= '1';
104     WHEN SW =>
105         ALUctrl <= "010";
106         ALUSrc <= '1';
107         MemdCS <= '1';
108         MemdWE <= '1';
109     WHEN LW =>
110         ALUctrl <= "010";
111         ALUSrc <= '1';
112         MemdCS <= '1';
113         MemtoReg <= '1';
114         RegWrite <= '1';
115     END CASE;
116 END PROCESS;

118 -- Processus Reset :
119 -- Initialise uniquement l'état courant

121 PROCESS (clk , reset)
122 BEGIN
123     IF (reset = '1') THEN
124         current_state <= fetch;
125     ELSIF (clk'event AND clk = '1') THEN
126         current_state <= next_state;
127     END IF;
128 END PROCESS;

130 END synth;
```

Listing 6: *ir_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;

4  ENTITY IR IS
5      PORT(
6          a      : IN      std_logic_vector (31 downto 0);
7          q      : OUT    std_logic_vector (31 downto 0);
8          we     : IN      std_logic ;
9          clk    : IN      std_logic
10     );
11 END IR ;

13 ARCHITECTURE synth OF IR IS
14 BEGIN

16     — Registre d'instruction (32 bits)
17     — activé en écriture avec un signal
18     — supplémentaire ('we')

20     PROCESS (clk , a , we)
21     BEGIN
22         IF (clk 'event AND clk = '1') THEN
23             IF (we = '1') THEN
24                 q <= a ;
25             END IF ;
26         END IF ;
27     END PROCESS ;

29 END synth ;
```

Listing 7: *memdata_synth.vhd*

```

1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;
3  USE ieee.std_logic_unsigned.all ;

5  ENTITY memdata IS
6    PORT(
7      adr      : IN      std_logic_vector (7 downto 0);
8      Data_Out : OUT    std_logic_vector (31 downto 0);
9      cs       : IN      std_logic ;
10     we       : IN      std_logic ;
11     Data_In  : IN      std_logic_vector (31 downto 0);
12     oe       : IN      std_logic
13   );
14 END memdata ;

16 ARCHITECTURE synth OF memdata IS
17   TYPE memdtype IS ARRAY((2*8)/4) -1 downto 0) OF
18     std_logic_vector(31 downto 0);
19   SIGNAL memoired : memdtype :=
20     (0 =>      "00000000000000000000000000000000100",
21     1 =>      "00000000000000000000000000000000011",
22     OTHERS => "000000000000000000000000000000000");
23 BEGIN

25   -- Memoire de données :
26   -- Doit être sélectionnée ('cs')
27   -- puis deux modes : lecture et écriture
28   -- en fonction de 'we'
29   -- (conv_integer nécessaire parce que l'index de la
30   -- mémoire est de type integer)

32   PROCESS (cs , oe , we , adr , Data_In)
33   BEGIN
34     Data_Out <= (OTHERS => 'Z');

36     IF (cs = '1') THEN
37       IF (we = '1') THEN

39         -- Mode : Ecriture
40         memoired(CONV_INTEGER(adr)) <= Data_In;

42     ELSE

```

```
44      -- Mode : Lecture
45      IF (oe = '1') THEN
46          Data_Out <= memoired(CONV_INTEGER(adr));
47      END IF;

49      END IF;
50  END IF;
51  END PROCESS;

53  END synth;
```

Listing 8: *meminstr_synth.vhd*

```

1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;

5  ENTITY meminstr IS
6    PORT(
7      adr   : IN      std_logic_vector (7 downto 0);
8      data  : OUT     std_logic_vector (31 downto 0);
9      cs    : IN      std_logic;
10     oe    : IN      std_logic
11   );
12 END meminstr ;

14 ARCHITECTURE synth OF meminstr IS
15   TYPE memitype IS ARRAY((2**8)/4)-1 downto 0) OF
16     std_logic_vector(31 downto 0);

18   signal memoirei : memitype := (
19     — addi $s0, $zero, 6
20     0 => "00100000000100000000000000000110",
21     — addi $s1, $zero, 8
22     1 => "00100000000100010000000000000100",
23     — add $a0, $zero, $zero
24     2 => "0000000000000000000100000000100000",
25     — addi $a1, $a0, 4
26     3 => "00100000100001010000000000000100",
27     — addi $v0, $a1, 4
28     4 => "00100000101000100000000000000100",
29     — sw $s0, 0($a0)
30     5 => "10101100100100000000000000000000",
31     — sw $s1, 0($a1)
32     6 => "10101100101100010000000000000000",
33     — lw $s0, 0($a0)
34     7 => "10001100100100000000000000000000",
35     — lw $s1, 0($a1)
36     8 => "10001100101100010000000000000000",
37     — add $s2, $zero, $zero
38     9 => "00000000000000000001001000000100000",
39     — add $s3, $zero, $zero
40     10 => "00000000000000000001001100000100000",
41     — add $s2, $s2, $s1
42     11 => "00000010010100011001000000100000",

```

```
43      -- addi $s3, $s3, 1
44      12 => "00100010011100110000000000000001",
45      -- beq $s0, $s3, 1
46      13 => "00010010011100000000000000000001",
47      -- beq $zero, $zero, -4
48      14 => "00010000000000001111111111111100",
49      -- sw $s2, 0($v0)
50      15 => "10101100010100100000000000000000",
51      -- beq $zero, $zero, -1
52      16 => "00010000000000001111111111111111",
53
54      OTHERS => "00000000000000000000000000000000");
55
56 BEGIN
57
58      -- Mémoire d'instruction contenant le programme
59      -- de multiplication
60
61      PROCESS (cs, oe, adr)
62      BEGIN
63          IF (cs = '1' AND oe = '1') THEN
64              data <= memoirei(CONV_INTEGER(adr));
65          ELSE
66              -- inactive => 'Z' en sortie
67              data <= (OTHERS => 'Z');
68          END IF;
69      END PROCESS;
70
71 END synth;
```

Listing 9: *mux_2_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;

4  ENTITY mux_2 IS
5      PORT(
6          e0  : IN      std_logic_vector (31 downto 0);
7          e1  : IN      std_logic_vector (31 downto 0);
8          o   : OUT     std_logic_vector (31 downto 0);
9          sel : IN      std_logic
10     );
11 END mux_2;

13 ARCHITECTURE synth OF mux_2 IS
14 BEGIN

16     -- Multiplexeur 2 entrées de 32 bits
17     -- avec 1 bit de contrôle.
18     -- Si le signal 'sel' diffère de '0' ou de '1' ,
19     -- envoi de 'U' en sortie

21     PROCESS (e0 , e1 , sel)
22     BEGIN
23         CASE sel IS
24             WHEN '0'      => o <= e0 ;
25             WHEN '1'      => o <= e1 ;
26             WHEN OTHERS => o <= (OTHERS => 'U') ;
27         END CASE;
28     END PROCESS;

30 END synth;
```

Listing 10: *or_2_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;

4  ENTITY or_2 IS
5      PORT(
6          In0  : IN    std_logic ;
7          In1  : IN    std_logic ;
8          Out0 : OUT   std_logic
9      );
10 END or_2 ;

12 — Simple opération logique (OU logique)
13 — utilisant la bibliothèque arithmétique

15 ARCHITECTURE synth OF or_2 IS
16 BEGIN
17     Out0 <= In0 OR In1 ;
18 END synth ;
```

Listing 11: *pc_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;

4  ENTITY pc IS
5      PORT(
6          clk      : IN  std_logic ;
7          a        : IN  std_logic_vector (31 downto 0) ;
8          q        : OUT std_logic_vector (31 downto 0) ;
9          we       : IN  std_logic ;
10         reset    : IN  std_logic
11     );
12 END pc ;

14 ARCHITECTURE synth OF pc IS
15 BEGIN

17     -- Registre PC 32 bits
18     -- Processus Reset et principal

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 ;

31 END synth ;
```

Listing 12: *reg_32_synth.vhd*

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;

4  ENTITY reg_32 IS
5      PORT(
6          clk : IN      std_logic;
7          a   : IN      std_logic_vector (31 downto 0);
8          q   : OUT     std_logic_vector (31 downto 0)
9      );
10 END reg_32;

12 ARCHITECTURE synth OF reg_32 IS
13     SIGNAL reg_signal : std_logic_vector (31 downto 0);
14 BEGIN

16     -- Registre 32 bits séquentiel
17     -- sans signal Reset

19     PROCESS (clk)
20     BEGIN
21         IF (clk'event AND clk = '1') THEN
22             reg_signal <= a;
23         END IF;
24     END PROCESS;

26     q <= reg_signal;

28 END synth;
```

Listing 13: *regfile_32_synth.vhd*

```

1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;
3  USE ieee .std_logic_unsigned .all ;

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

18 ARCHITECTURE synth OF RegFile_32 IS
19   TYPE address_type is array (31 downto 0) of
20     std_logic_vector (31 downto 0);
21   SIGNAL rf_address : address_type;
22 BEGIN

24   -- Registre Principale

26   PROCESS (aa , ab , clk , RegWrite , aw)
27   BEGIN

29     -- Sorties constantes
30     a <= rf_address (CONV_INTEGER (aa));
31     b <= rf_address (CONV_INTEGER (ab));

33     -- Ecriture séquentielle
34     IF (clk 'event AND clk = '1') THEN

36       -- Adresse '000' interdite
37       IF (RegWrite = '1' AND aw /= "000") THEN
38         rf_address (CONV_INTEGER (aw)) <= WData;
39       END IF;
40     END IF;

42   END PROCESS;

```

```
44    -- Registre zero
45    rf_address(0) <= (others => '0');

47 END synth;
```

Listing 14: *shiftright2_synth.vhd*

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;

4 ENTITY shiftright2 IS
5   PORT(
6     entree : IN    std_logic_vector (31 downto 0);
7     sortie : OUT  std_logic_vector (31 downto 0)
8   );
9 END shiftright2;

11 ARCHITECTURE synth OF shiftright2 IS
12 BEGIN

14   -- Décalage vers la gauche de 2 bits
15   -- en ajoutant des '0'

17   sortie <= entree(29 downto 0) & "00";

19 END synth;
```

Listing 15: *signextend_synth.vhd*

```
1  LIBRARY ieee ;
2  USE ieee .std_logic_1164 .all ;

4  ENTITY SignExtend IS
5      PORT(
6          entree : IN      std_logic_vector (15 downto 0);
7          sortie : OUT    std_logic_vector (31 downto 0)
8      );
9  END SignExtend ;

11 ARCHITECTURE synth OF SignExtend IS
12 BEGIN

14     — Extension du bit 15 aux bits 31–16 de l’entrée

16     sortie <= (31 downto 16 => entree(15)) & entree ;

18 END synth ;
```