SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

POLYTECH SORBONNE

Processeur Monocycle:

Simulation VHDL

YANG Liyun
JIN Qianhui

Année: 2019-2020
Enseignant: M.Douze

Sommaire

I. Introduction

Pour ce projet du processeur mono-cycle, nous avons séparé en 4 parties. La première partie décrit l'unité de traitement du processeur. La deuxième partie parle du unité de gestion des instructions du processeur. Ensuite, c'est l'unité de contrôle du processeur. Et dans la dernière partie, nous assemblons ces trois unité et faire quelque modifications.

Pour être facile à comprendre, nous avons nommés la même nom du fichier pour le titre de chaque petite partie. En plus, nous avons fait les test bench pour chaque composant pour éviter de mauvaise fonctionnement en cas d'assemblage. Pour les schémas bloc, nous les obtenons à partir du Netlist Viewer de Quartus.

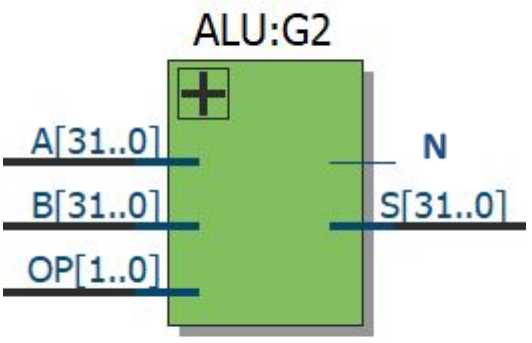
II. Unité de traitement

L'unité de traitement du processeur est principalement composé d'une unité arithmétique et logique, d'un banc de registre et d'une mémoire de données, en plus des multiplexeurs et extension.

L'unité arithmétique et logique - ALU.vhd

Selon le schéma du bloc de l'ALU, nous le définie en vhdl par la manière suivante:

```
Entity ALU is    --Unité arithmétique log
port(
  OP:in std_logic_vector(1 downto 0);
  A,B:in std_logic_vector(31 downto 0);
  S:out std_logic_vector(31 downto 0);
  N:out std_logic);
end entity;
```

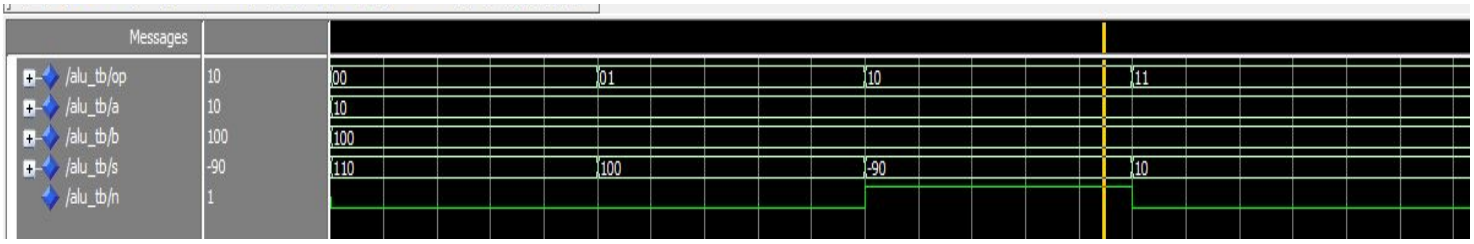


Selon la valeur du signal OP, l'ALU va effectuer les différentes opérations avec A et B, 2 bus 32 bits en entrée:

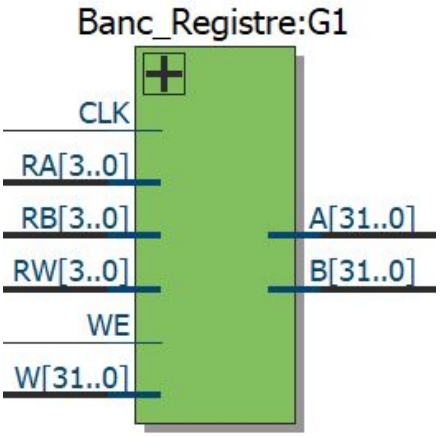
-quand OP= "00", S=A+B
-quand OP= "01", S=B
-quand OP= "10", S=A-B
-quand OP= "11", S=A

Comme la signal N prend la valeur 1 quand S est strictement négatif et 0 sinon. Donc pour déterminer le signal N, on prend simplement la valeur du poid fort du signal S, bus 32 bits en sortie.

Voici la simulation:



Banc de registres - Banc_Registre.vhd



Le banc possède un tableau de 16 registres de 32 bits.

La lecture ffectue de manière combinatoire et simultanée, les bus de 32 bits en sortie A et B prennent la valeur du RA_ème et Rb_ème registres.

L'écriture se fait de manière synchrone. Lorsque la valeur du signal WE=1 et front montant ,on peut écrit la valeur du signal W dans RW_ème registre.

Voici son définition en vhdl:

```
Entity Banc_Registre is
port(
  CLK:in std_logic;
  W: in std_logic_vector(31 downto 0);   --bus de données
  RA: in std_logic_vector(3 downto 0);   --bus d'adresse en lecture du port A
  RB: in std_logic_vector(3 downto 0);   --bus d'adresse en lecture du port B
  RW: in std_logic_vector(3 downto 0);   --bus d'adresse en écriture
  WE: in std_logic;                      --commande pour ecire dans le registre
  A: out std_logic_vector(31 downto 0);  --Bus de données en lecture du port A
  B: out std_logic_vector(31 downto 0)   --Bus de données en lecture du port B
);
end entity;
```

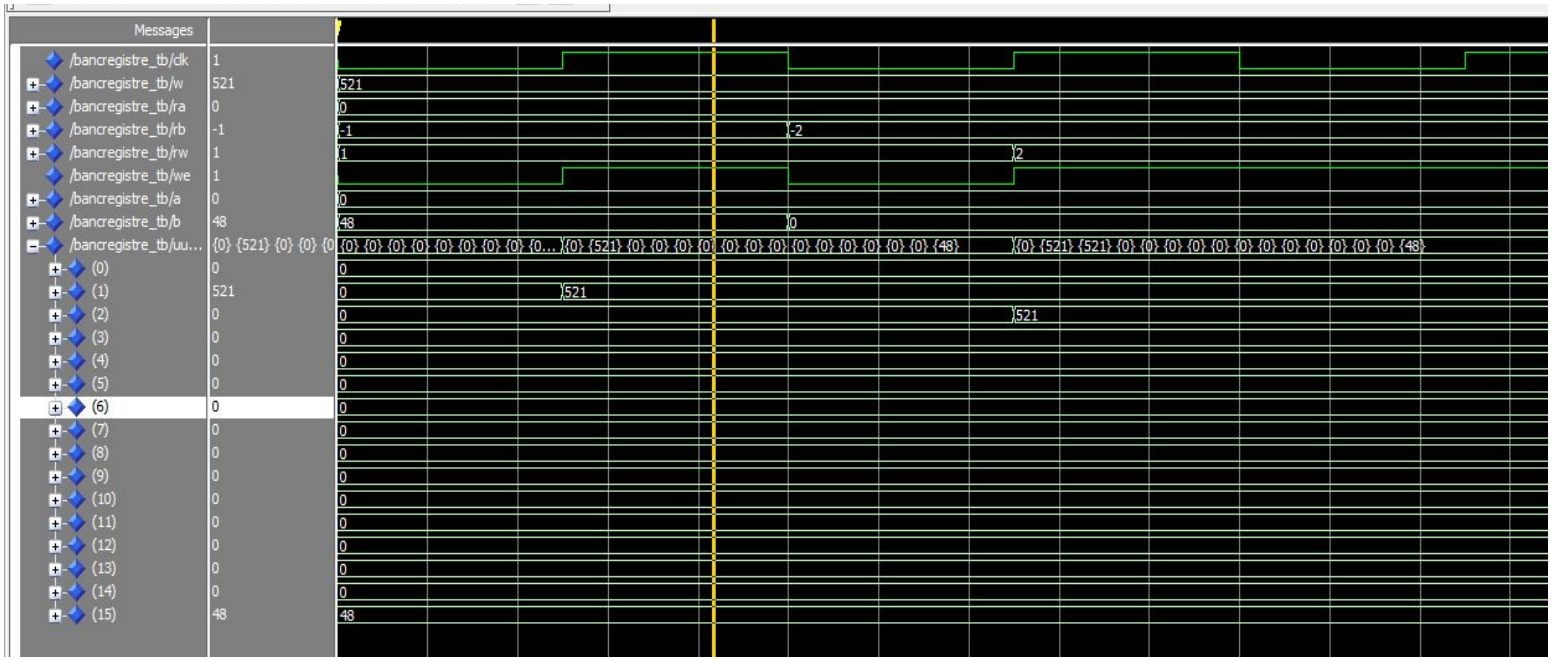
Pour le tableau du registre, on a fait une fonction pour l'initialiser:

```
Architecture rtl of Banc_Registre is
  --definie un tableau de 16 lignes et chaque ligne contient 32 bits
  type table is array (0 to 15) of std_logic_vector(31 downto 0);

  --fonction d'initialiser du tableau
  function init_banc return table is

  variable result : table;

begin
  for i in 14 downto 0 loop
    result(i):= (others => '0');
  end loop;

  result(15):= x"00000030";      --R(15)=48
  return result;
end init_banc;


signal Banc: table:= init_banc;
```

Dans le tableau du registre, tous les registres sont en valeur 0 sauf la 15ème qui est en valeur 48.

Après l'initialisation, on crée un signal interne Banc qui est de type table.

Voici son simulation:



Assemblage de l'unité de traitement - Unite_traitement.vhd

Pour valider l'unité de traitement, nous assemblons l'ALU et le banc de registre.

Voici la déclaration:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Entity Unite_Traitement is
port(
    Reset,Clk,WE: in std_logic;
    OP:in std_logic_vector(1 downto 0);
    RW,RA,RB: in std_logic_vector(3 downto 0);
    busW: inout std_logic_vector(31 downto 0));
end entity;

Architecture rtl of Unite_Traitement is
  signal busA,busB: std_logic_vector(31 downto 0);
  signal N:std_logic;

begin

G1: entity work.Banc_Registre port map(CLK=>Clk ,W=>busW ,RA=>RA ,RB=>RB ,Rw=>RW ,WE=>WE ,A=>busA ,B=>busB);
G2: entity work.ALU port map(OP=>OP ,A=>busA ,B=>busB ,S=>busW ,N=>N);

end Architecture;
```

Les entrés du bloc sont les même entrés dans le banc registre, et les deux sorties du banc de registre deviennent des signals interne: busA et busB. Et ces deux signals interne sont considérés comme les entrés de l'ALU. Les sorties du bloc sont N et busW qui est aussi l'entrée du bloc. Donc nous avons mis inout pour le signal busW.

Puis nous avons fait une simulation pour cette assemblage qui tester les différentes opérations.

Dans le test bench, nous avons fait deux processus, un pour l'horloge et d'autre pour la simulation principale. Nous avons fait le changement d'état de l'horloge toutes les 5 ns. Dans le deuxième processus, nous avons initialisé en début par le reset qui est actif en état haut, puis nous avons effectuons les opérations suivantes:

-RA = "1111", RW = "0001", OP = "11" : R(1) = R(15) = 48

-RA = "0001", RB= "1111", RW = "0001", OP = "00" : R(1) = R(1) + R(15) = 48+48=96

-RW= "0010": R(2) = R(1) + R(15) = 96+48=144

-RW= "0011", OP= "10": R(3) = R(1) - R(15) = 96-48=48

-RA= "0111", RW= "0101": R(5) = R(7) - R(15) = 0-48=-48

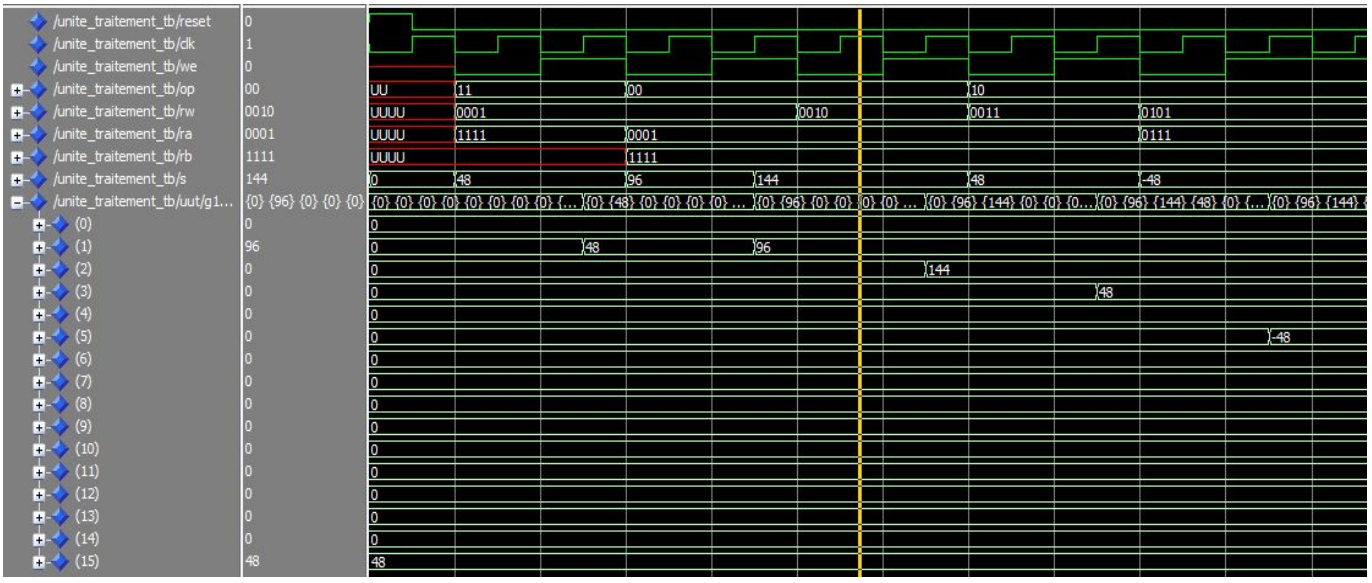
En plus, pour obtenir les résultats clairement, nous avons changé la signal WE tous les 10 ns.

```
Clock: process
  begin
    Clk <= '0';
    wait for 5 ns;
    Clk <= '1';
    wait for 5 ns;
end process;
```

```
simulate:process

  begin
  --initialisation
  Reset <= '1';
  wait for 5 ns;
  Reset <= '0';
  wait for 5 ns;
```

```
-- R(1) = R(15)
WE <= '0';      --pas de l'ecriture au début
RA <= "1111";   --busA = R(15)
RW <= "0001";   --S=R(1)
OP <= "11";     --S=busA;
wait for 10 ns;
WE <= '1';
wait for 10 ns;
```

Nous avons obtenu la solution correcte:

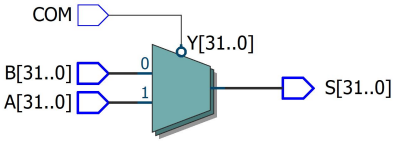


Alors pour améliorer l'unité de traitement, nous décidons d'ajouter le composant multiplexeur et extension.

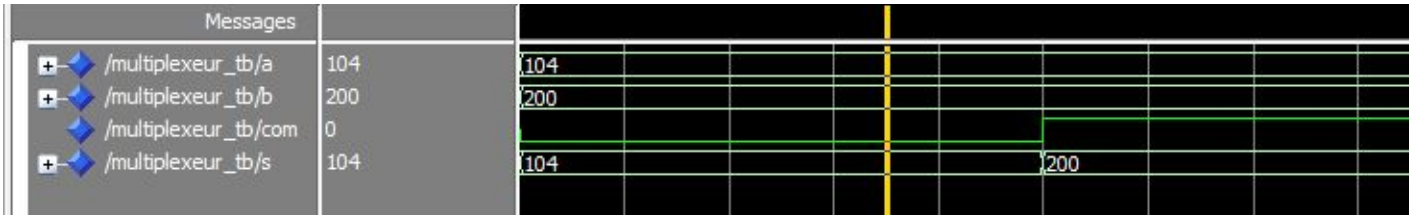
Multiplexeur 2 - Multiplexeur.vhd

La multiplexeur 2 est un élément qui a 2 entrées A et B qui sont sélectionnés par la commande entrée COM, alors si COM= ‘0’, la sortie S va prendre la valeur du A, si COM= ‘1’, S=B. Pour être utilisable, nous avons utilisé un paramètre générique N pour modifier la taille du bits pour les entrées et la sortie.

Voici le schéma du bloc, la déclaration et la simulation:




```
Entity Multiplexeur is
generic (N: positive :=32);
port(
  A,B: in std_logic_vector(N-1 downto 0);
  COM: in std_logic;        --commande de selection
  S: out std_logic_vector(N-1 downto 0));
end entity;
```



Extension de signe - Extension.vhd

L’extension sert à étendre une entrée de N bits en sortie de 32 bits. N est un paramètre générique. Les N-1 dernière du sortie prend la même valeur que l’entrée et les 32-N+1 bits resté va prend la même valeur de la Nème bit de l’entré qui est la bit du signe.

Voici la déclaration et la simulation:

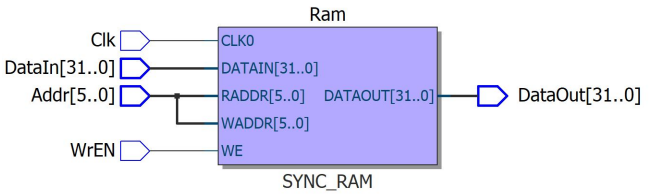
```
Entity Extension is
generic (N: positive :=8);
port(
  E:in std_logic_vector(N-1 downto 0);
  S: out std_logic_vector(31 downto 0));
end entity;
```

Mémoire de données - Memoir.vhd

La mémoire de données est similaire de la banc registre. C’est un tableau qui sert à charger et stocker 64 mots de 32 bits. L’écriture est aussi activée

par un signal WE= ‘1’ et en front montant. Quant à la lecture, le bus de sortie DataOut porte la valeur de la Addr_ème donnée.

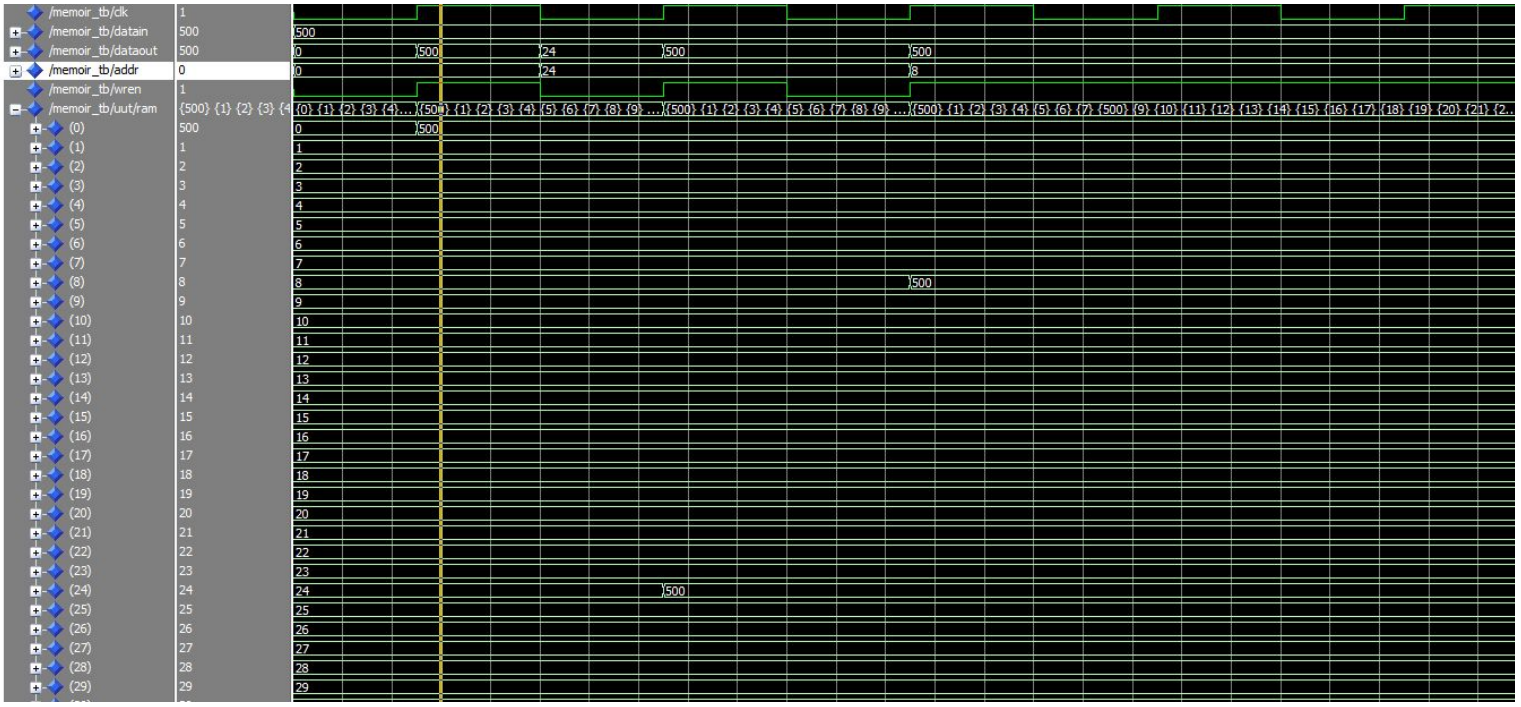
Voici le schéma bloc, la déclaration du module, l’initialisation du tableau et la simulation:




```
entity Memoir is
port(
  Clk:in std_logic;
  DataIn:in std_logic_vector(31 downto 0);   --bus donnée en ecriture
  DataOut: out  std_logic_vector(31 downto 0);  --bus donnée en lecture
  Addr: in std_logic_vector(5 downto 0);  --bus d'adresses en lecture et écriture
  WrEN: in std_logic    --Write enable
);
end entity;
```

```
type Ramtype is array(0 to 63) of std_logic_vector(31 downto 0);

--fonction d'initialisation du tableau
function init_Banc return Ramtype is

variable result : Ramtype;

begin
  for i in 63 downto 0 loop
    result(i):= Std_logic_vector(To_unsigned(i,32));
  end loop;
```

Dans l’initialisation du tableau, pour être plus évident, nous avons intégré tous les mots une valeur qui est différent que les autres.

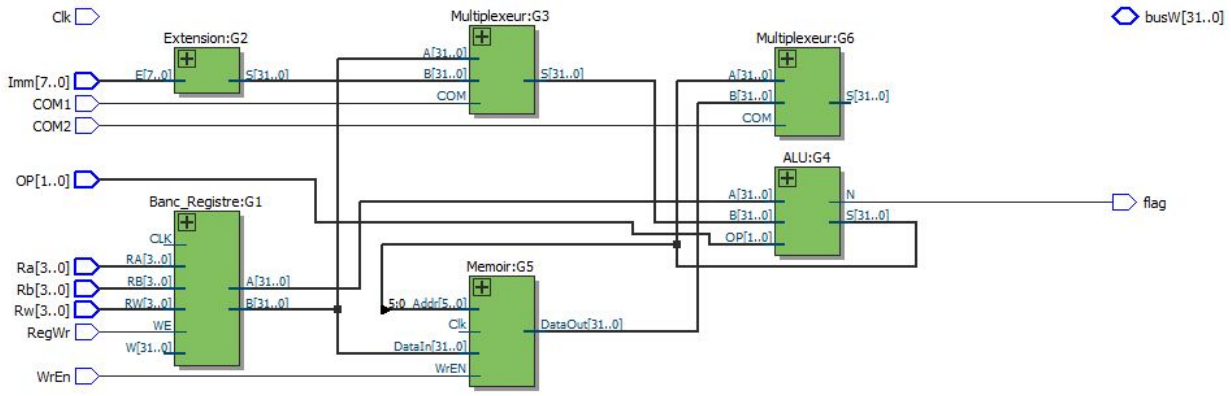


Assemblage unité de traitement - AUI.vhd

Nous assemblons tous les modules décrit avant en un bloc.

```
Entity AUT is --assembleur de unite de traitement
port(
       Clk:in std_logic;
       RegWr:in std_logic;       --commande pour ecrire dans le registre
       Rw,Ra,Rb:in std_logic_vector(3 downto 0);   --bus d'adresse
       Imm:in std_logic_vector(7 downto 0);      --valeur immediate
       COM1,COM2: in std_logic;
       OP: in std_logic_vector(1 downto 0);
       WrEn: in std_logic;     --commande pour ecrire dans le memoir
       busW: inout std_logic_vector(31 downto 0);
       flag: out std_logic   --N dans ALU
);
end entity;

Architecture rtl of AUT is
  signal busA,busB:std_logic_vector(31 downto 0);
  signal ALUOut:std_logic_vector(31 downto 0);
  signal ExOut:std_logic_vector(31 downto 0);
  signal multiOut: std_logic_vector(31 downto 0);
  signal DataOut: std_logic_vector(31 downto 0);

begin

G1:Entity work.Banc_Registre port map(CLK=>Clk ,W=>busW ,RA=>Ra ,RB=>Rb ,Rw=>Rw ,WE=>RegWr ,A=>busA ,B=>busB);
G2:Entity work.Extension generic map(8) port map(E=>Imm ,S=>ExOut);
G3:Entity work.Multiplexeur generic map(32) port map(A=>busB ,B=>ExOut ,COM=>COM1,S=>multiOut);
G4:Entity work.ALU port map(OP=>OP ,A=>busA ,B=>multiOut ,S=>ALUOut ,N=>flag);
G5:Entity work.Memoir port map(Clk=>Clk ,DataIn=>busB ,DataOut=>DataOut ,Addr=>ALUOut(5 downto 0) ,WrEn=>WrEN);
G6:Entity work.Multiplexeur generic map(32) port map(A=>ALUOut ,B=>DataOut ,COM=>COM2,S=>busW);

end Architecture;
```

Le COM1 sert à choisir entre la valeur immédiate et busB, la COM2 permet de choisir entre la sortie de l'ALU et la sortie de mémoir.



Puis pour vérifier son bonne fonctionnement, nous avons effectué ces opérations suivant:

-L'addition de 2 registres

```
--Addition de 2 registres
RegWr <= '0';   --on n'ecrit rien dans le registre
wait for 10 ns;
OP <= "00";     --ALUOut=busA+busB
Ra <= "0001";   --busA=R(1)
Rb <= "1111";   --busB=R(15)
Rw <= "0010";   --R(2)=busW
COM1 <= '0';    --multiOut=busB
COM2 <= '0';    --busW=ALUOut
WrEn <= '0';    --on n'ecrit rien dans le memoir
wait for 10 ns;
```

busW = busA+busB
= R(1)+R(15)
= 0+48
=48

-L'addition d'un registre avec une valeur immédiate

```
--Addition d'un registre avec une valeur immediate( ALUOut=busA+ExOut )
Imm <= "01100101";   --valeur immediate
COM1 <= '1';         --multiOut=ExOut
Rw <= "0011";        --R(3)=busW
wait for 10 ns;
```

busW = busA+Exout
= R(1)+101
= 0+101
=101

-La soustraction de deux registres

```
--Soustraction de 2 registres
OP <= "10";       --AluOut= busA-busB
Rw <= "0100";     --R(4)=busW
Ra <= "0101";     --busA=R(5)
COM1 <= '0';      --multiOut=busB
wait for 10 ns;
```

busW = busA-busB
= R(5)-R(15)
= 0-48
= -48

-La soustraction d'une valeur immédiate à un registre

```
--Soustraction d'un valeur immediate a 1 registre (AluOut=busA-ExOut)
Imm <= "00000111";
Rw <= "0110";     --R(6)=busW
COM1 <= '1';      --multiOut=ExOut
wait for  10 ns;
```

busW = busA-Exout
= R(5) - 7
= 0-7 = -7

-La copie de la valeur d'un registre dans un autre registre

```
--copie de la valeur d'un registre dans un autre registre
Ra <= "1111";       --busA= R(15)
Rw <= "0111";       --R(7)=busW
OP <= "11";         --ALUOut= busA
COM1 <= '0';        --multiOut= busB
RegWr <= '1';       --copie la valeur busA(R(15)) dans R(7)
wait for 10 ns;
```

R(7) = busW = busA
= R(15)
= 48

-La lecture d'un mot de la mémoire dans un registre

```
--Lecture d'un mot de la memoire dans un registre
RegWr <= '0';
wait for 10 ns;
Imm <= "10000111";
COM1 <= '1';      --multiOut=ExOut
COM2 <= '1';      --busW=DataOut (un mot)
Rw <= "1000";     --R(8)=busW
Rb <= "1110";     --busB=R(14)
OP <= "01";       --ALUOut = busB
wait for 10 ns;
```

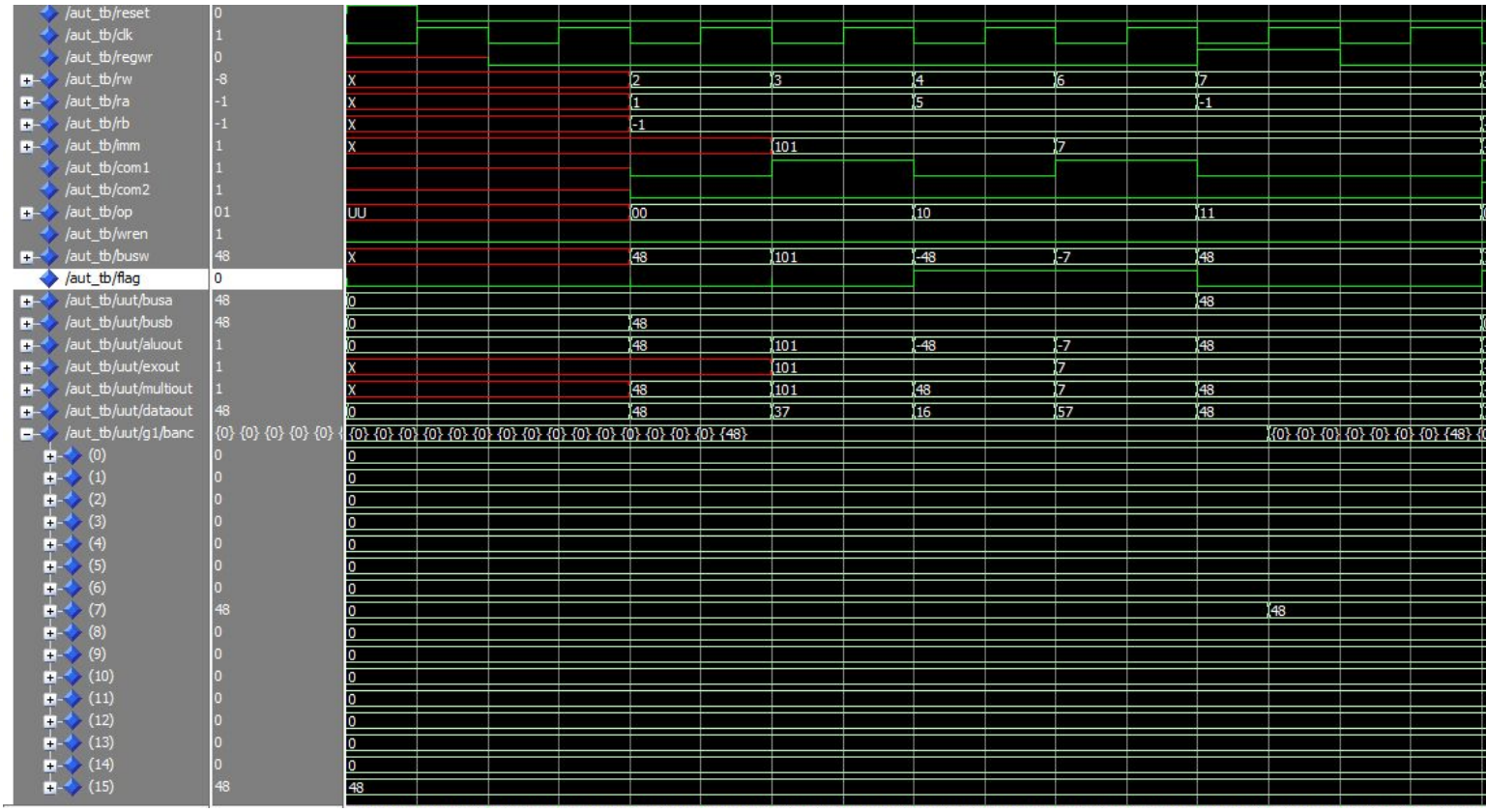
Nous lisons un mot qui sortie de la mémoir dans le registre, c'est-à-dire
busW = DataOut = 48

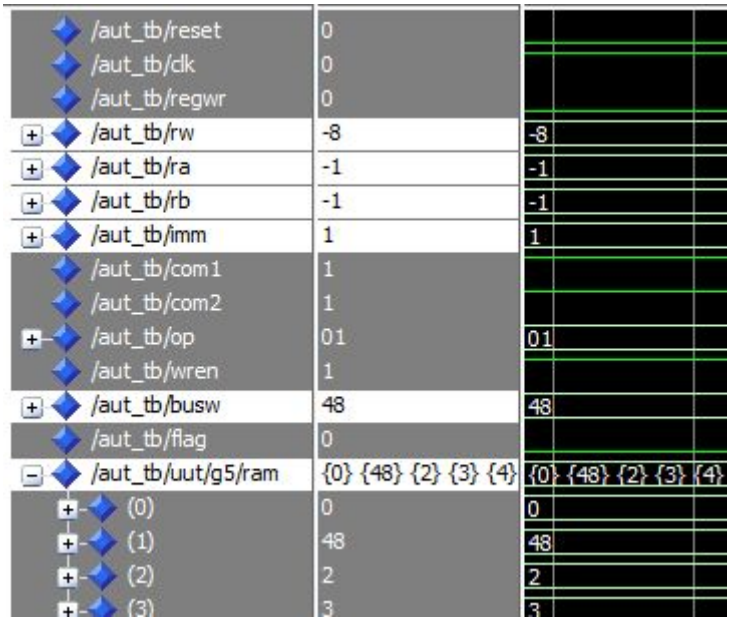
-L'écriture d'un registre dans un mot de mémoire

```
--L'ecriture d'un registre dans un mot de la memoire
Rb <= "1111";       --busB=R(15)
Imm <= "00000001";
WrEn <= '1';        --Ecriture dans la memoire
wait for 10 ns;
```

Cette fois nous devons écrit la valeur de registre dans la tableau de mémoir.
Mem(AluOut) = Mem(1)= busW = 48

Dans la simulation, on a obtenu les bon résultats, donc ce unité de traitement marche bien:

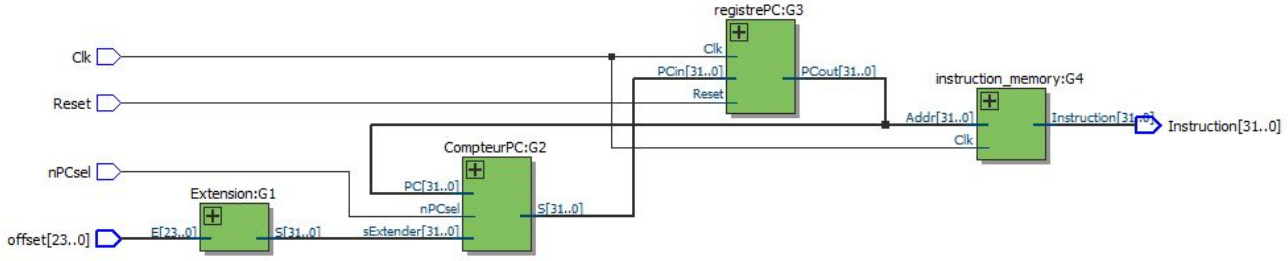




III. Unité de gestion des instructions

Unité de gestion des instructions - UGI.vhd

L'unité de gestion des instructions est une mémoire qui envoie le mot choisi en sortie, l'adresse du mot choisi est déterminée par un registre PC qui s'incrémenter à front montant d'horloge grâce à un module de compteur. Pour être plus claire et correct, nous décidons de traiter ces modules séparément, puis les assembler ensemble à la fin dans fichier UGI.vhd.



Pour l'UGI, nous l'avons séparé en 4 parties: la mémoire d'instruction, le registre PC, l'unité d'extension et un compteur. Pour l'unité d'extension, nous avons pris ce que nous avons écrit dans la première partie en changeant le paramètre générique N=24.

Mémoire d'instruction - instruction_memory.vhd

Ce module est similaire à celui de l'unité de traitement, mais sans la commande de écrite et les bus de données en écriture.

Voici la déclaration:

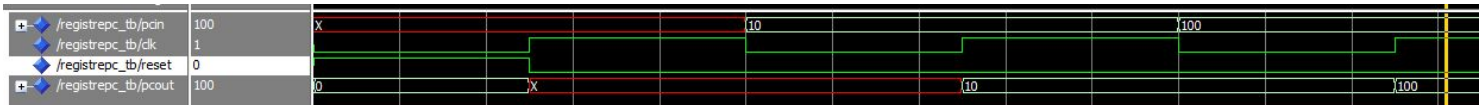
```
entity instruction_memory is
  port(
    Clk: in std_logic;
    Addr: in std_logic_vector (31 downto 0);
    Instruction: out std_logic_vector (31 downto 0)
    );
end entity;
```

Registre PC - RegitsrePC.vhd

C'est une simple redirection du signal entrant vers la signal sortant mais synchrone, il est marche que en front montant. Dans UGI, cela est la signal de compteur. Il y a un Reset dans ce composant qui sert à initialiser la valeur du signal en 0.

Voici la déclaration et la simulation:

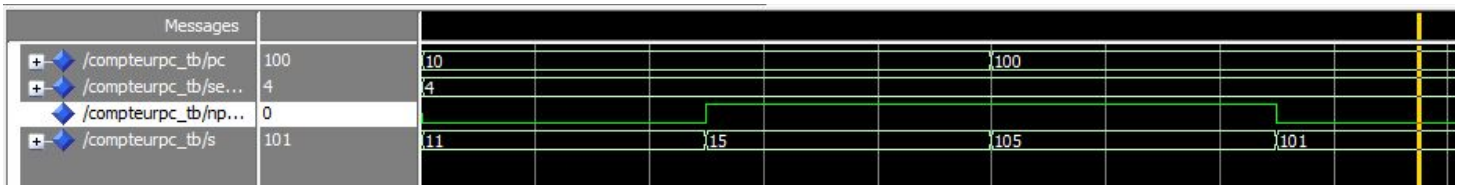
```
entity registrePC is
port(
  PCin: in std_logic_vector (31 downto 0);
  Clk: in std_logic;
  Reset: in std_logic;                    --
  PCout: out std_logic_vector(31 downto 0)
);
end entity;
```



Compteur - CompteurPC.vhd

Ce module sert à incrémenter le signal entré +1 quand nPCsel= ‘0’ ou +1+valeur offset quand nPCsel= ‘1’. Et il reprend la sortie pour incrémenter à nouveau. Dans l’architecture de cette composant, nous avons choisit de calculer ces deux résultat et les stocker dans deux signaux internes. Puis déterminer la sortie par la commande de sélection nPCsel qui prend l’un des deux signal pour la signal sortie.

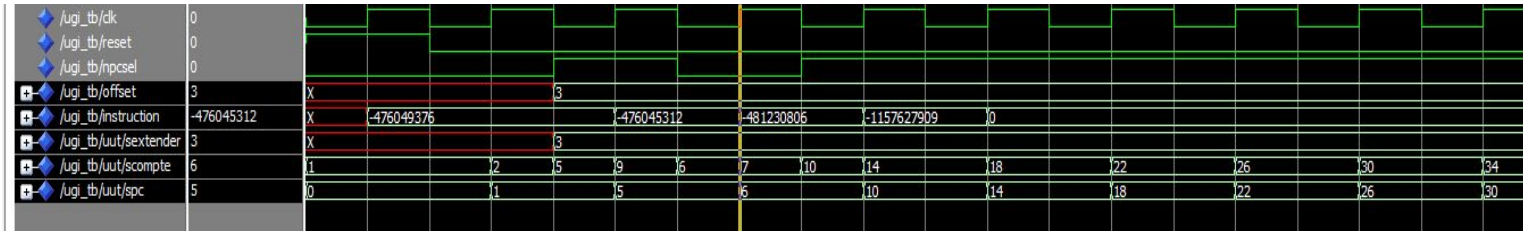
```
Entity CompteurPC is
port(
  PC,sExtender: in std_logic_vector(31 downto 0);
  nPCsel: in std_logic;          --commande de selection
  S: out std_logic_vector(31 downto 0));
end entity;

Architecture rtl of CompteurPC is

  signal Y1: std_logic_vector(31 downto 0);
  signal Y0: std_logic_vector(31 downto 0);

begin

  Y1<=std_logic_vector(unsigned(PC) +1);     --signal stock la valeur de PC+1
  Y0<=std_logic_vector(signed(Y1) + signed(sExtender));    --signal stock la valeur de PC+1+offset

  S<= Y1 when (nPCsel='0') else
      Y0 when (nPCsel='1');
```



Simulation de l'UGI - UGI_tb.vhd

Voici le contenu de test bench et la simulation:

```
simulate: process
begin
  Reset <= '1';    --initialisation
  wait for 10 ns;
  Reset <= '0';
  wait for 10 ns;
  offset <= "00000000000000000000011";    --offset=3
  nPCsel <= '1';                          --Instruction=mem(5)
  wait for 10 ns;
  nPCsel <= '0';     --Instruction=mem(6)
  wait for 10 ns;
  nPCsel <= '1';     --Instruction=mem(10)
  wait for 10 ns;
  wait;
end process;
```



Nous pouvons voir que en fil jaune, la valeur affichée du sortie de compteur PC est 6 et après le front montant et nPCsel égale 0, on a bien la sortie du compteur égale 7. Et les instructions correspondent bien la valeur initialisée dans le tableau de mémoir. Et au cas de nPCsel égale 1, la sortie de compteur égale 5, après l'horloge, on a bien obtenue 9 quant à la sortie.

Donc l'UGI fonctionne bien.

IV. Unité de contrôle

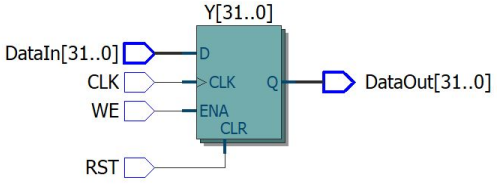
L'unité de contrôle est composé par un registre 32 bits et d'un décodeur combinatoire.

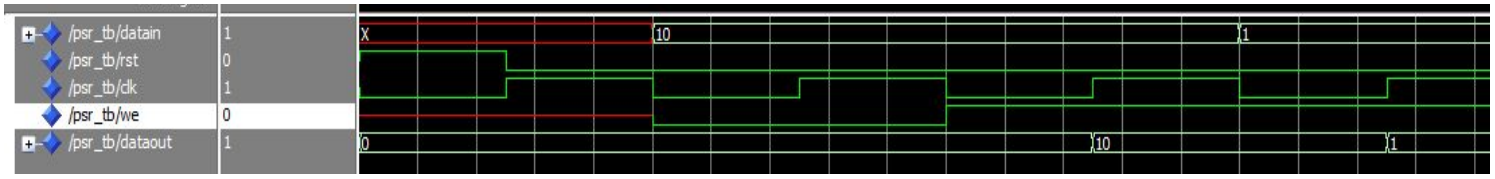
Registre 32 bit avec commande de chargement - PSR.vhd

Ce registre sert à stocker l'état du processeur. Dans notre processeur, le PSR est acquise par le drapeau N de l'ALU, mais nous l'avons décrit dans la fin, dans l'assemblage des trois unités. Dans ce composant, nous avons décrit juste une chargement qui est activée par une commande de charge WE, lorsque WE= '1' la sortie va prendre la valeur de bus d'entrée, sinon aucune action. En plus il est synchrone et il y a aussi un reset asynchrone qui sert à initialiser la signal de sortie en 0.

Voici la déclaration, le schéma de bloc et la simulation:

```
Entity PSR is   --Processor State Registre
port(
  DataIn: in std_logic_vector(31 downto 0);     --bus de chargement
  RST: in std_logic;          --reset asynchrone, actif a l'etat haut
  CLK: in std_logic;
  WE: in std_logic;           -- commande de chargement
  DataOut: out std_logic_vector(31 downto 0)    --bus de sortie
  );
end entity;
```





Décodeur d'instructions - Decodeur.vhd

Ce module contrôle tous les signaux de l'unité de traitement, l'unité de gestion des instructions et le registre PSR. Voici sa déclaration:

```
Entity Decodeur is
port(
  nPCsel: out std_logic;
  PSR: in std_logic_vector(31 downto 0);
  Instruction: in std_logic_vector(31 downto 0);
  RegWr: out std_logic;
  RegSel: out std_logic;
  ALUctr: out std_logic_vector(1 downto 0);
  ALUsrc: out std_logic;
  PSREn: out std_logic;
  MemWr: out std_logic;
  WrSrc: out std_logic;
  Rn,Rd,Rm: out std_logic_vector(3 downto 0);
  Offset: out std_logic_vector(23 downto 0);
  Imm: out std_logic_vector(7 downto 0)
  );
end Entity;
```

Ce module se compose en deux process. Premièrement, nous créonsun type énuméré qui a 8 état différents: MOV, ADDi, ADDr, CMP, LDR, STR, BAL, BLT. Et nous créons un signal instr_courante de ce type. Puis nous faitons un processus sensible à la valeur de la signal sortie de la mémoire instruction. Nous effectuons une analyse sur les bits de poids forts de l'instruction, comme nous les indiquons au-dessous:

-quand le début du Instruction= "111000101000", alors instr_courant=Addi
-quand le début du Instruction= "111000001000", alors instr_courant=Addr
-quand le début du Instruction= "11101010", alors instr_courant=BAL
-quand le début du Instruction= "10111010", alors instr_courant=BLT
-quand le début du Instruction= "111000110101",alors instr_courant=CMP
-quand le début du Instruction= "111001100001", alors instr_courant=LDR
-quand le début du Instruction= "1110001110100000", alors instr_courant=MOV
-quand le début du Instruction= "111001100000", alors instr_courant=STR
-sinon, instr_courante=NULL

```
process(Instruction)    --processeur sensible par Instruction pour determiner l'etat
begin
  if (Instruction(31 downto 20)= "111000101000") then
    instr_courante <= ADDi;
  elsif (Instruction(31 downto 20)= "111000001000")then
    instr_courante <= ADDr;
  elsif (Instruction(31 downto 24)= "11101010") then
    instr_courante <= BAL;
  elsif (Instruction(31 downto 24)= "10111010") then
    instr_courante <= BLT;
  elsif (Instruction(31 downto 20)= "111000110101") then
    instr_courante <= CMP;
  elsif (Instruction(31 downto 20)= "111001100001") then
    instr_courante <= LDR;
  elsif (Instruction(31 downto 16)= "1110001110100000") then
    instr_courante <= MOV;
  elsif (Instruction(31 downto 20)= "111001100000") then
    instr_courante <= STR;
  else NULL;
  end if;
```

Deuxièmement, nous allons faire un processus sensible sur la signal instr_courante et instruction. Au début de ce processus, nous initialisons les signaux offset , imm, Rd, Rn et Rm en 0. Puis nous donnons des valeurs pour les commandes des registres et les opérations du processeur. Et nous donnons aussi des valeurs pour les signaux qui initialisent en début d'après la valeur de l'instruction.

Voici le tableau des valeurs des commandes:

Instruction	nPCSel	RegWR	ALUSrc	ALUCtr	PSREn	MemWr	WrSrc	RegSel
ADDi	0	1	1	00	0	0	0	0
ADDr	0	1	0	00	0	0	0	0
BAL	1	0	0	00	0	0	0	0
BLT	flag(0)	0	0	00	1	0	0	0
CMP	0	0	1	10	1	0	0	0
LDR	0	1	1	00	0	0	1	0
MOV	0	1	1	01	0	0	0	0
STR	0	0	1	00	0	1	0	1

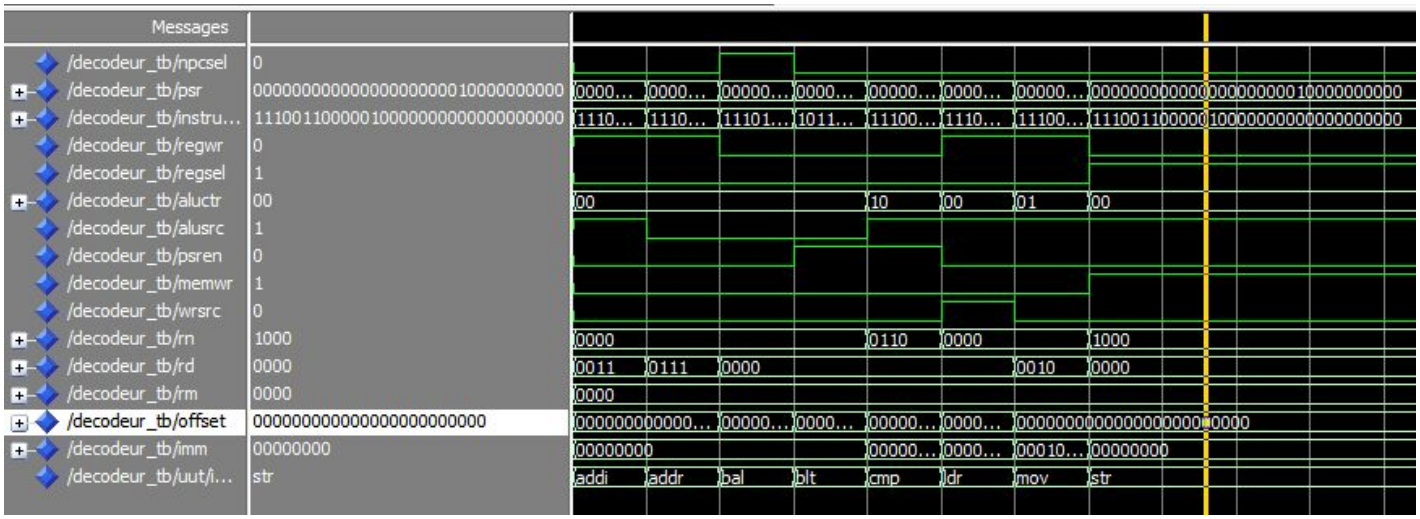
Il y a une exception dans le tableau est que pour la valeur de nPCSel pour état BLT est dépend de la valeur de la drapeau N de l'ALU qui est la première bit du l'entrée de module PSR = flag dans notre assemblage.

Et voici la déclaration du process:

```
process(instr_courante,Instruction)    --proc
                                       --vale
begin

    offset  <= "00000000000000000000000000";
    imm     <= "00000000";
    Rd      <= "0000";
    Rn      <= "0000";
    Rm      <= "0000";

  if (instr_courante=ADDi) then
    nPCsel <= '0';
    RegWr <= '1';
    ALUSrc <= '1';
    ALUCtr <= "00";
    PSREn <= '0';
    MemWr <= '0';
    WrSrc <= '0';
    RegSel <= '0';
    Rn <= Instruction(19 downto 16);
    Rd <= Instruction(15 downto 12);
    Imm <= Instruction(7 downto 0);
```

```
  elsif (instr_courante=BLT) then
    nPCsel <= PSR(0);
    RegWr <= '0';
    ALUSrc <= '0';
    ALUCtr <= "00";
    PSREn <= '1';
    MemWr <= '0';
    WrSrc <= '0';
    RegSel <= '0';
    Offset <= Instruction(23 downto 0);

  elsif (instr_courante=CMP) then
    nPCsel <= '0';
    RegWr <= '0';
    ALUSrc <= '1';
    ALUCtr <= "10";
    PSREn <= '1';
    MemWr <= '0';
    WrSrc <= '0';
    RegSel <= '0';
    Rn <= Instruction(19 downto 16);
    Imm <= Instruction(7 downto 0);
```

Voici la simulation du décodeur:



Nous vérifions tous les valeurs de commande de registre correspondent bien que l'état affiché dans la simulation, et celui de la valeur de instr_courant est bien correspond que la valeur de instruction. Donc le module décodeur fonctionne bien.

V. Assemblage et validation du processeur

Dans cette partie, nous allons faire quelque modification dans certaines modules afin de valider l'assemblage de trois unités. Et nous allons former un processeur monocycle.

Complétion sur l'AUT par un multiplexeur - AUT_modif.vhd

L'ajout de multiplexeur dans AUT sert à choisir l'entré de l'adresse de busB qui est utile dans l'état STR. En effet la structure de multiplexeur est la même que celui dans AUT, donc nous avons choisi de changer un peu la déclaration et ajouter un multiplexeur en plus dans l'architecture de l'AUT.

```
Entity AUT_modif is --assembleur de unite de traitement apres la modification
port(
       Clk:in std_logic;
       RegWr,RegSel:in std_logic;
       Rn,Rd,Rm:in std_logic_vector(3 downto 0);
       Imm:in std_logic_vector(7 downto 0);
       COM1,COM2: in std_logic;
       OP: in std_logic_vector(1 downto 0);
       WrEn: in std_logic;
       busW: inout std_logic_vector(31 downto 0);
       flag: out std_logic   --N dans ALU
);
end entity;

Architecture rtl of AUT_modif is
  signal busA,busB:std_logic_vector(31 downto 0);
  signal ALUOut:std_logic_vector(31 downto 0);
  signal ExOut:std_logic_vector(31 downto 0);
  signal multiOut: std_logic_vector(31 downto 0);
  signal DataOut: std_logic_vector(31 downto 0);
  signal Rb: std_logic_vector(3 downto 0);

begin

G1:Entity work.Multiplexeur generic map(4) port map(A=>Rm ,B=>Rd ,COM=>RegSel,S=>Rb);
G2:Entity work.Banc_Registre port map(CLK=>Clk ,W=>busW ,RA=>Rn ,RB=>Rb ,Rw=>Rd ,WE=>RegWr ,A=>busA ,B=>busB);
G3:Entity work.Extension generic map(8) port map(E=>Imm ,S=>ExOut);
G4:Entity work.Multiplexeur generic map(32) port map(A=>busB ,B=>ExOut ,COM=>COM1,S=>multiOut);
G5:Entity work.ALU port map(OP=>OP ,A=>busA ,B=>multiOut ,S=>ALUOut ,N=>flag);
G6:Entity work.Memoir port map(Clk=>Clk ,DataIn=>busB ,DataOut=>DataOut ,Addr=>ALUOut(5 downto 0) ,WrEn=>WrEN);
G7:Entity work.Multiplexeur generic map(32) port map(A=>ALUOut ,B=>DataOut ,COM=>COM2,S=>busW);
```

Modification de l'initialisation de la mémoire instruction

Nous modifions la fonction de l'initialisation de la mémoire instruction dans le instruction_memory.vhd afin de simuler la code MIPS codé en binaire sur la tableau d'instruction.

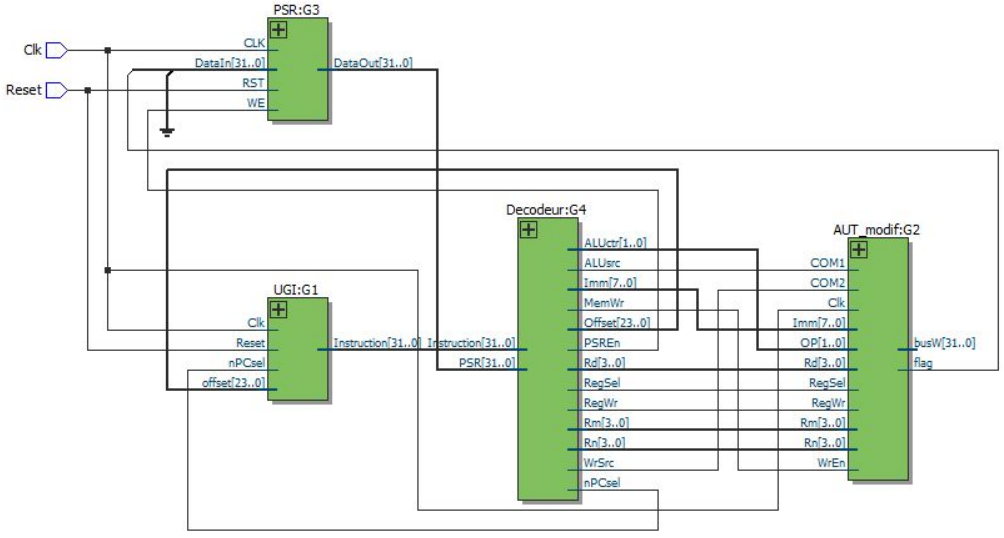
Voici la modification:

```
begin
  for i in 63 downto 0 loop
    result(i):= (others => '0');
  end loop;                    -- Addr      -- INSTRUCTION  -- COMMENTAIRE
    result (0):=x"E3A01020";   -- 0x0 _main -- MOV R1,#0x20 -- R1 = 0x20
    result (1):=x"E3A02000";   -- 0x1       -- MOV R2,#0x00 -- R2 = 0
    result (2):=x"E6110000";   -- 0x2 _loop -- LDR R0,0(R1) -- R0 = DATAMEM[R1]
    result (3):=x"E0822000";   -- 0x3       -- ADD R2,R2,R0 -- R2 = R2 + R0
    result (4):=x"E2811001";   -- 0x4       -- ADD R1,R1,#1 -- R1 = R1 + 1
    result (5):=x"E351002A";   -- 0x5       -- CMP R1,0x2A  -- si R1 >= 0x2A
    result (6):=x"BAFFFFFB";   -- 0x6       -- BLT loop     -- PC = PC + (-5) si N = 1
    result (7):=x"E6012000";   -- 0x7       -- STR R2,0(R1) -- DATAMEM[R1] = R2
    result (8):=x"EAFFFFF7";   -- 0x8       -- BAL main     -- PC = PC + (-7)
  return result;
end init_mem;
```

Assemblage - Monocycle.vhd

Nous allons assembler AUT, UGI PSR et décodeur ensemble dans le module final: Monocycle. Pour la déclaration de monocycle, il nous reste que deux entrés qui doivent être déclaré, ce sont la horloge et le reset. Et il y a beaucoup de signaux internes qui vont être déclarées dans l'architecture. Voici la déclaration et le schéma de bloc:

```
Entity Monocycle is
port(
  Clk,Reset: in std_logic
  );
end Entity;

Architecture rtl of Monocycle is
  signal Instruction: std_logic_vector(31 downto 0);
  signal nPCsel: std_logic;
  signal Rn,Rd,Rm: std_logic_vector(3 downto 0);
  signal RegWr,RegSel: std_logic;
  signal ALUsrc,WrSrc: std_logic;
  signal ALUctr: std_logic_vector(1 downto 0);
  signal PSREn: std_logic;
  signal MemWr: std_logic;
  signal N: std_logic;
  signal flag: std_logic_vector(31 downto 0);
  signal PSR: std_logic_vector(31 downto 0);
  signal busW: std_logic_vector(31 downto 0);
  signal Offset: std_logic_vector(23 downto 0);
  signal Imm: std_logic_vector(7 downto 0);
```

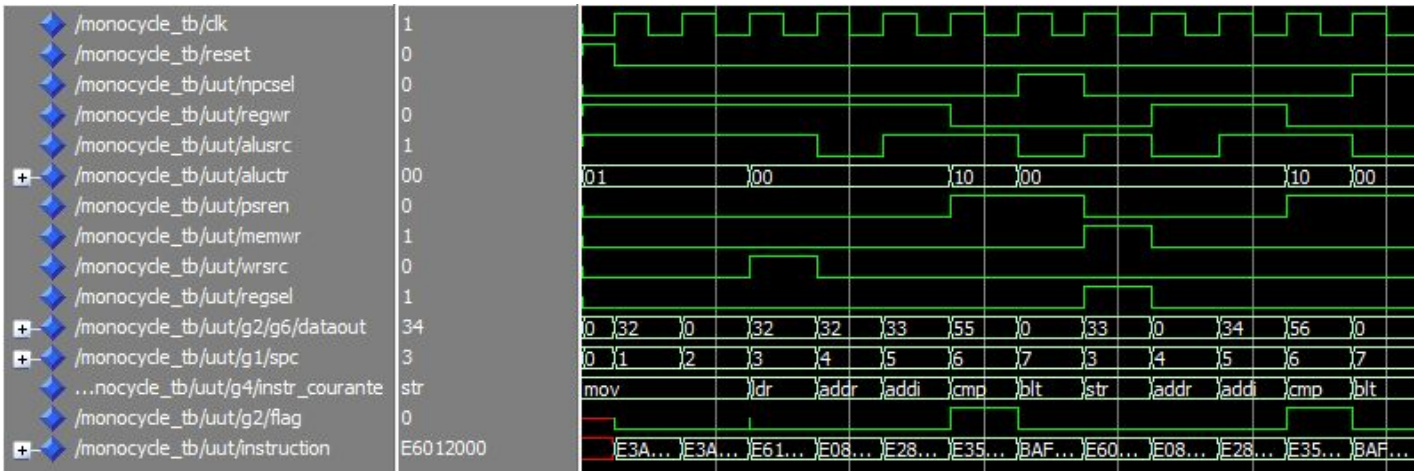


En plus dans le monocycle, nous obtenons la signal flag par l'acquisition de la drapeau N de l'ALU sur son poid faible, et les autres bits nous les mettons en 0.

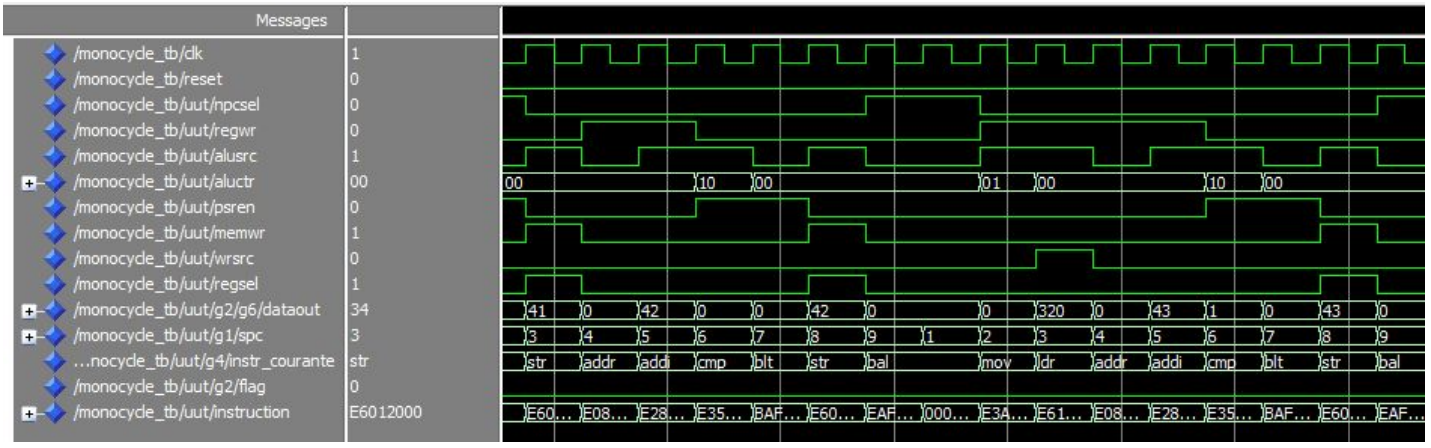
```
flag <="00000000000000000000000000000000"&N;   --le poid faible de flag prend la valeur de N(signe)
```

Nous avons aussi fait une simulation pour le monocycle pour tester son fonctionnement.

Voici la simulation:



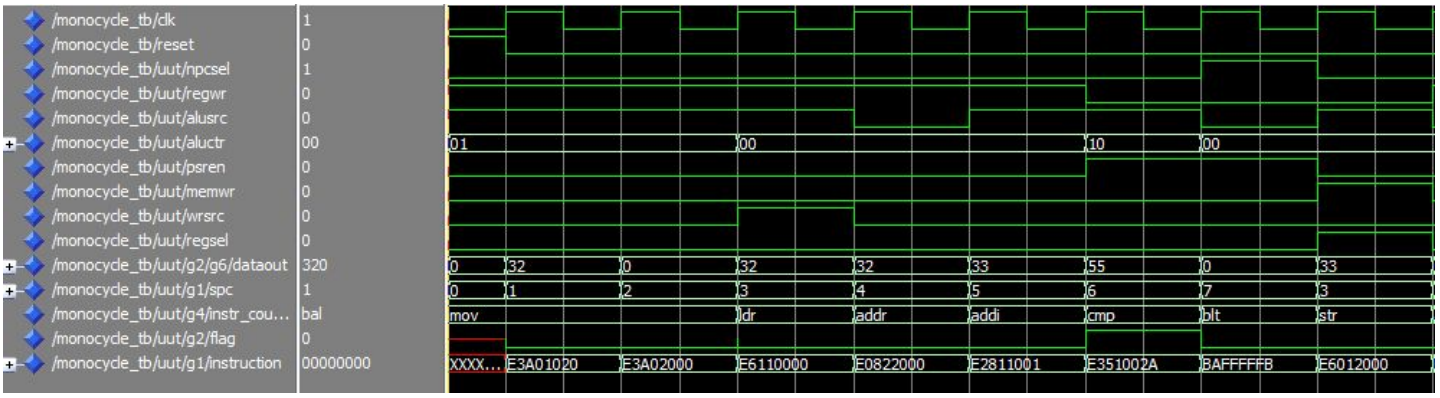
C’est la simulation au début, nous pouvons voir qu’il y a une bonne roulement de boucle qui est contrôlé par la signal spc, mais quand nous vérifions sur la signal instr_courant, il y a un des inconvénients. Quand la boucle atteint l’état BLT ,il ne va pas brancher en LDR, il passe d’abord en STR et après tout de suite en ADDr.



Mais après 10 fois la même boucle, nous avons bien obtenu la bonne branchement vers STR, BAL et MOV. Mais le la valeur de la signal spc égale 9 et 1 lié l’état BAL pas 8, et après l’état BAL, il branche vers l’état MOV en 2 pas en 0 et 1, il saut une action de MOV. Et après la 10 boucle au début, il reste bouclé de 1 à 9, de MOV à BAL.

VI. Conclusion

En conclusion, d'après les problèmes apparaissent dans la simulation monocycle. Nous pensons qu'il y a un contrainte de temps. Cela peut être vérifié par la valeur de l'instruction et celui de instr_courante. Nous pouvons voir l'adresse correspond à l'action est un peu décalée. Par exemple, pour la première action MOV, normalement, elle doit avoir la valeur E3A01020 qui apparaît dans l'instruction, mais dans notre cas, elle apparaît dans la deuxième action de MOV. Nous pensons c'est à cause de cela qui fait apparaître l'action STR à cause un compteur en trop. Et cela rend aussi une saute de la partie LDR.




```
end loop;                       -- Addr      -- INSTRUCTION
  result (0):=x"E3A01020";      -- 0x0 _main -- MOV R1,#0x20
  result (1):=x"E3A02000";      -- 0x1       -- MOV R2,#0x00
  result (2):=x"E6110000";      -- 0x2 _loop -- LDR R0,0(R1)
  result (3):=x"E0822000";      -- 0x3       -- ADD R2,R2,R0
  result (4):=x"E2811001";      -- 0x4       -- ADD R1,R1,#1
  result (5):=x"E351002A";      -- 0x5       -- CMP R1,0x2A
  result (6):=x"BAFFFFFB";      -- 0x6       -- BLT loop
  result (7):=x"E6012000";      -- 0x7       -- STR R2,0(R1)
  result (8):=x"EAFFFFF7";      -- 0x8       -- BAL main
```

En plus, pour les problèmes apparus après l'action BLT, c'est aussi à cause de problème en début. Mais nous ne savons pas comment résoudre cette problème.