

# PROCESSEUR MONO-CYCLE: SIMULATION VHDL

## Un compte-rendu global est demandé pour ce projet

L'objectif de ces séances est de concevoir et de simuler un cœur de processeur. Ce processeur sera conçu à partir de briques de base (registres, multiplexeurs, bancs mémoire, UAL...) qui seront combinés pour réaliser les différents blocs du système (unité de traitement, unité de gestion des instructions, unité de contrôle). Le modèle du processeur sera ensuite validé en simulant l'exécution d'un programme simple de test.

Pour chaque modèle, l'électronique sera décrite en VHDL comportemental et simulée avec Modelsim à l'aide d'un banc de test que vous développerez pour cela.

## PARTIE 1 – UNITE DE TRAITEMENT

L'unité de traitement du processeur est principalement composée d'une Unité Arithmétique et Logique, d'un banc de registres et d'une mémoire de données, ainsi que de composants annexes (multiplexeurs, extension de signe...)

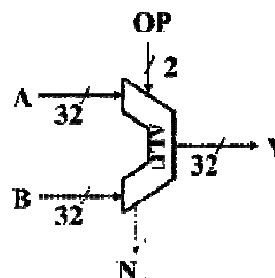
### Unité Arithmétique Logique - ALU

L'Unité Arithmétique et Logique (UAL) 32 bits possède:

- **OP** : Signal de commande sur 2 bits
- **A, B** : Deux bus 32 bits en entrée
- **S** : Bus 32 bits en sortie
- **N** : Drapeau de sortie sur 1 bit

Les valeurs du signal de commande **OP** déterminent les opérations implémentées dans l'UAL

- |   |              |              |            |
|---|--------------|--------------|------------|
| - | <b>ADD</b> : | $Y = A + B;$ | OP = "00"; |
| - | <b>B</b> :   | $Y = B;$     | OP = "01"; |
| - | <b>SUB</b> : | $Y = A - B;$ | OP = "10"; |
| - | <b>A</b> :   | $Y = A;$     | OP = "11"; |



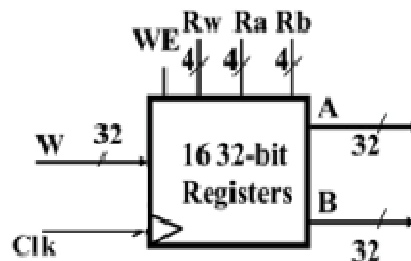
A l'issue de chaque opération, le drapeau **N** prend la valeur 1 si le résultat est strictement négatif, et 0 sinon.

*Conseil : La librairie ieee.numeric\_std.all permet les opérateurs '+' et '-'*

## Banc de Registres

Le banc possède 16 registres de 32 bits. Il comporte les entrées sorties suivantes :

- **CLK**: Horloge,
- **Reset** : reset asynchrone (actif à l'état haut) non représentés sur le schéma
- **W**: Bus de données en écriture sur 32 bits
- **RA**: Bus d'adresses en lecture du port A sur 4 bits
- **RB**: Bus d'adresses en lecture du port B sur 4 bits
- **RW**: Bus d'adresses en écriture sur 4 bits
- **WE**: Write Enable sur 1 bit
- **A**: Bus de données en lecture du port A
- **B**: Bus de données en lecture du port B



La **lecture** des registres se fait de manière **combinatoire et simultanée** :

- Le bus de sortie **A** porte la valeur du registre N° **RA**
- Le bus de sortie **B** porte la valeur du registre N° **RB**

L'**écriture** des registres se fait de manière **synchrone**, sur le front montant de l'horloge. Elle est commandée par le signal **WE** :

- Si **WE** = 1, au front montant on recopie la valeur du bus **W** dans le registre n° **RW**
- Si **WE** = 0, au front montant il n'y a pas d'écriture effectuée

### A FAIRE

#### 1) Décrire et simuler ces modules en VHDL comportemental

*Le banc de registres sera déclaré comme étant un tableau de std\_logic\_vector. Son initialisation sera réalisée à l'aide d'une fonction (voir code ci-dessous)*

```

-- Declaration Type Tableau Memoire
type table is array(15 downto 0) of std_logic_vector(31 downto 0);

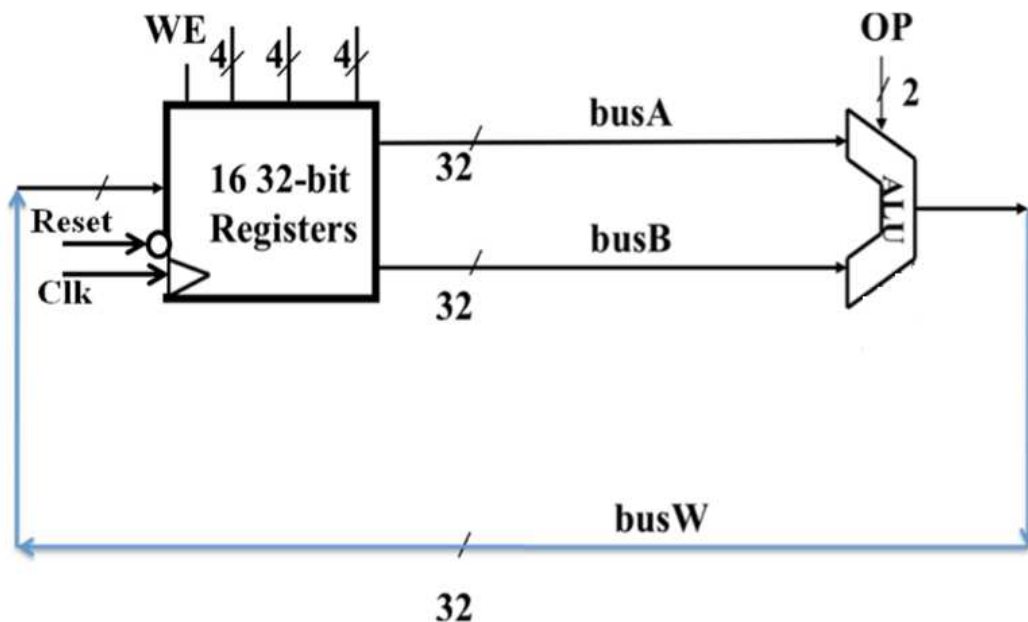
-- Fonction d'Initialisation du Banc de Registres
function init_banc return table is
variable result : table;
begin
    for i in 14 downto 0 loop
        result(i) := (others=>'0');
    end loop;
    result(15) := X"00000030";
    return result;
end init_banc;

-- Déclaration et Initialisation du Banc de Registres 16x32 bits
signal Banc: table:=init_banc;

```

*Conseil : Il faut impérativement éviter que la variable d'indexation du tableau soit un entier négatif, non initialisé, ou supérieur à la taille du tableau. Si un de ces cas se produit, la simulation s'arrêtera sur une erreur fatale.*

- 2) **Assembler l'UAL et le banc comme sur le schéma ci-dessous pour valider l'unité de traitement.**



- 3) **Ecrire un banc de test pour valider par simulation le bon fonctionnement des opérations suivantes :**

- $R(1) = R(15)$
- $R(1) = R(1) + R(15)$
- $R(2) = R(1) + R(15)$
- $R(3) = R(1) - R(15)$
- $R(5) = R(7) - R(15)$

### Multiplexeur 2 vers 1

Ce multiplexeur dispose d'un paramètre générique **N** fixant la taille des données en entrée et en sortie

- **A, B** : Deux bus sur **N** bits en entrée
- **COM**: Commande de sélection sur 1 bit
- **S** : Bus sur **N** bits en sortie

Les valeurs du signal **COM** fixent le comportement de la sortie

COM	Opération Effectuée
0	S = A
1	S = B

### Extension de signe

Ce module permet d'étendre en sortie sur 32 bits le signe d'une entrée codée sur **N** bits. Il possède donc un paramètre générique fixant la valeur de **N**. Il comporte également

- **E**: Bus sur **N** bits en entrée
- **S** : Bus sur 32 bits en sortie

## Mémoire de données

Cette mémoire permet de charger et stocker 64 mots de 32 bits. . Il comporte les entrées sorties suivantes :

- **CLK**: Horloge
- **DataIn**: Bus de donnés en écriture sur 32 bits
- **DataOut**: Bus de données en lecture sur 32 bits
- **Addr**: Bus d'adresses en lecture et écriture sur 6 bits
- **WrEn**: Write Enable sur 1 bit

La **lecture** des registres se fait de manière **combinatoire et simultanée** :

- Le bus de sortie **DataOut** porte la valeur de la données N° **Addr**

L'**écriture** des registres se fait de manière **synchrone**, sur le front montant de l'horloge. Elle est commandée par le signal **WE** :

- Si **WE** = 1, au front montant on recopie la valeur du bus **DataIn** dans le registre n° **Addr**
- Si **WE** = 0, au front montant il n'y a pas d'écriture effectuée

*Conseil:*

*\_ Ce module est très proche du banc de registre et peut être conçu à partir de ce dernier.*

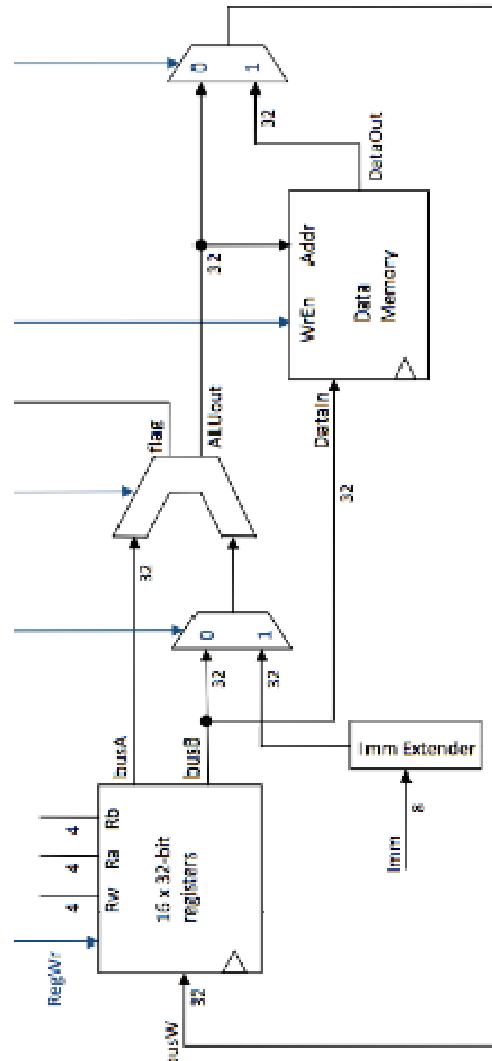
*\_ Le signal provenant de l'Alu étant sur 32 bits, la réduction avec les 6 bits de Addr peut être réalisée au niveau du **Port Map** (Addr => SignalAluOut (5 downto 0), ...)*

## A FAIRE

- 1) *Donner le schéma bloc de ces modules*
- 2) *Décrire et simuler ces modules en VHDL comportemental*

## Assemblage Unité de Traitement

Les modules décrits précédemment doivent être assemblés conformément au schéma ci-dessous:



### A FAIRE

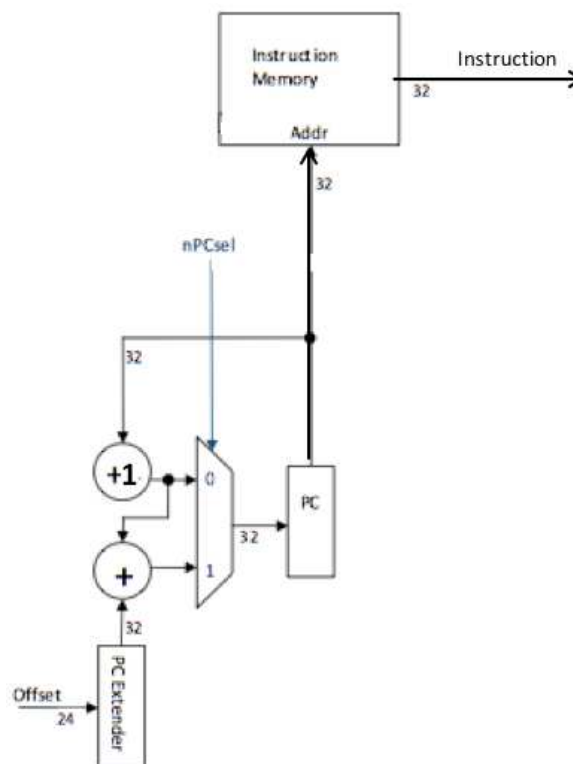
- 1) *Ecrire un module VHDL effectuant l'assemblage de l'unité de traitement.*
- 2) *Ecrire un testbench pour valider par simulation le bon fonctionnement de :*
  - *L'addition de 2 registres*
  - *L'addition d'1 registre avec une valeur immédiate*
  - *La soustraction de 2 registres*
  - *La soustraction d'1 valeur immédiate à 1 registre*
  - *La copie de la valeur d'un registre dans un autre registre*
  - *L'écriture d'un registre dans un mot de la mémoire.*
  - *La lecture d'un mot de la mémoire dans un registre.*

## PARTIE 2 – UNITE DE GESTION DES INSTRUCTIONS

### Unité de gestion des instructions

L'unité de gestion des instructions 32 bits, représentée sur la figure ci-dessous, possède :

- Une mémoire d'instruction de 64 mots de 32 bits similaire à celle de l'unité de traitement, celle-ci est fournie en annexe dans le fichier **instruction\_memory.vhd**
- Il n'y a pas de bus de données en écriture ni de Write Enable.
- Un registre 32 bits (Registre **PC**). Ce registre possède une horloge et un reset asynchrone (actif à l'état haut) non représentés sur le schéma.
- Une unité d'extension de 24 à 32 bits signés similaire au module décrit précédemment
- Une unité de mise à jour du compteur de programme PC suivant le signal de contrôle **nPCsel** :
  - Si **nPCsel** = 0 alors **PC** = **PC** + 1
  - Si **nPCsel** = 1 alors **PC** = **PC** + 1 + SignExt(offset)



### A FAIRE

#### 1) Décrire et simuler l'unité de gestion des instructions

## **PARTIE 3 – UNITE DE CONTROLE**

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

### **Registre 32 bit avec commande de chargement**

Ce registre servira à stocker l'état du processeur (Processor State Register ou **PSR**). Dans notre cas, l'état se limitera à la valeur du drapeau **N** de l'**UAL**. Il possède les entrées/sorties suivantes :

- **DATAIN** : Bus de chargement sur 32 bits
- **RST** : Reset asynchrone, actif à l'état haut
- **CLK** Horloge
- **WE** : Commande de chargement
- **DATAOUT** : Bus de sortie sur 32 bits

Si **WE**=1, le registre mémorise la valeur placée sur le bus **DATAIN**. Si **WE**=0, le registre conserve sa valeur précédente



### Decodeur d'Instructions

Ce module combinatoire génère les signaux de contrôle de l'unité de traitement, de l'unité de gestion des instructions, ainsi que du registre **PSR**, tous décrits précédemment.

Les valeurs de ces commandes dépendent de l'instruction récupérée dans la mémoire instructions, et éventuellement de l'état du registre **PSR**. La structure binaire des différentes instructions est décrite en Annexe page 15, 16 et 17.

Les commandes à générer concernent :

- Le multiplexeur d'entrée du registre **PC**
- La commande de chargement du registre **PSR**
- Les bus d'adresse **RA**, **RB**, **RW**, ainsi que le **WE** du banc de registres
- La commande **RegSel** du multiplexeur situé en amont du bus d'adresse **RB** de ce banc.
- La commande **OP** de l'**UAL**
- La commande **UALSrc** du multiplexeur situé en amont de l'entrée B de l'**UAL**.
- La commande **WrSrc** du multiplexeur situé en sortie de l'**UAL** et de la mémoire de données.
- Le Write Enable de la mémoire de données

Pour la conception de ce décodeur, on s'appuiera sur le schéma d'ensemble du processeur, du tableau de commande ainsi que sur le descriptif du jeu d'instructions fournis en annexe..

Pour décrire ce décodeur d'instructions, on déclarera tout d'abord le type énuméré suivant

```
type enum_instruction is (MOV, ADDi, ADDr, CMP, LDR,
STR, BAL, BLT);
```

```
signal instr_courante: enum_instruction;
```

On pourra ensuite utiliser deux process dans l'architecture

- Un process sensible sur la sortie de la mémoire instructions (nommé ***instruction*** sur le schéma en annexe) dont le rôle est de fixer la valeur du signal ***instr\_courante***
- Un process sensible sur le signal ***instruction*** qui donnera la valeur des commandes des registres et opérateurs du processeur.

## A FAIRE

- 1) ***Compléter le tableau « valeurs des commandes » se trouvant en pièce jointe, qui synthétisera les actions du décodeur en fonction des instructions.***
- 2) ***Décrire en VHDL les deux modules de l'unité de contrôle. Le registre PSR sur 32 bits, s'il reçoit une commande de chargement, fera l'acquisition du drapeau N de l'ALU sur son poids faible, et 31 bits à 0 sur ses poids forts.***

## **PARTIE 4 – ASSEMBLAGE ET VALIDATION DU PROCESSEUR**

Il faut à présent finaliser la modélisation du processeur en assemblant ses trois unités principales.

- L'unité de gestion des instructions
- L'unité de traitement
- L'unité de contrôle

### **A FAIRE**

- 1) *Compléter l'unité de traitement conçue précédemment en ajoutant un multiplexeur à 2 entrées sur 4 bits piloté par le signal de commande RegSel généré par l'unité de contrôle. Ce multiplexeur sera placé en entrée de l'adresse RB du banc de registres, comme cela est représenté sur le schéma du processeur en annexe.*
- 2) *Assembler le processeur à partir de ses trois unités.*
- 3) *Simuler l'exécution du programme de test par le processeur et vérifier son bon fonctionnement. Il sera peut-être nécessaire d'initialiser quelques données dans la Mémoire Data (de 0x20 à 0x2A)*



## Valeurs des Commandes

INSTRUCTION	nPCSel	RegWr	ALUSrc	ALUCtr	PSREn	MemWr	WrSrc	RegSel
ADDi								
ADDr								
BAL								
BLT								
CMP								
LDR								
MOV								
STR								

## **Sous- ensemble du jeu d'instructions**

Type d'Instruction	Code Assembleur	Actions
Traitement de Données	ADD Rd, Rn, Rm	Rd := Rn + Rm
	ADD Rd, Rn, #Imm	Rd := Rn + Imm
	MOV Rd, #Imm	Rd := Imm
	CMP Rn, #Imm	Flag := Rn - Imm
Accès Mémoire	LDR Rd, [Rn, #Offset]	Rd := Mem[Rn + Offset]
	STR Rd, [Rn, #Offset]	Mem[Rn + Offset] := Rd
Branchements	B{AL} label	PC := PC + Offset
	BLT	Flag => PC := PC + Offset

## **Programme de test**

```

0x0  _main :    MOV R1, #0x20      ; --R1 <= 0x20
0x1                      MOV R2, #0      ; --R2 <= 0
0x2  _loop :    LDR R0, 0(R1)      ; --R0 <= DATA_MEM[R1]
0x3                      ADD R2, R2, R0   ; --R2 <= R2 + R0
0x4                      ADD R1, R1, #1   ; --R1 <= R1 + 1
0x5                      CMP R1, 0x2A     ; --? R1 = 0x2A
0x6                      BLT loop        ; --branchement à _loop si
R1 inferieur a 0x2A
    _end :
0x7                      STR R2, 0(R1)    ; --DATA_MEM[R1] <= R2
0x8                      BAL main        ; --branchement à _main

```

## **Mémoire d'instruction : instruction\_memory.vhd**

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity instruction_memory is
    port(
        PC: in std_logic_vector (31 downto 0);
        Instruction: out std_logic_vector (31 downto 0)
    );
end entity;

architecture RTL of instruction_memory is
    type RAM64x32 is array (0 to 63) of std_logic_vector (31 downto 0);

    function init_mem return RAM64x32 is
        variable result : RAM64x32;
    begin
        for i in 63 downto 0 loop
            result (i) := (others => '0');
        end loop;
        -- PC          -- INSTRUCTION  -- COMMENTAIRE
        result (0) := "E3A01020"; -- 0x0 _main -- MOV R1,#0x20 -- R1 = 0x20
        result (1) := "E3A02000"; -- 0x1      -- MOV R2,#0x00 -- R2 = 0
        result (2) := "E6110000"; -- 0x2 _loop -- LDR R0,0(R1) -- R0 = DATAMEM[R1]
        result (3) := "E0822000"; -- 0x3      -- ADD R2,R2,R0 -- R2 = R2 + R0
        result (4) := "E2811001"; -- 0x4      -- ADD R1,R1,#1 -- R1 = R1 + 1
        result (5) := "E351002A"; -- 0x5      -- CMP R1,0x2A  -- si R1 >= 0x2A
        result (6) := "BAFFFFFFB"; -- 0x6      -- BLT loop -- PC = PC + (-5) si N = 1
        result (7) := "E6012000"; -- 0x7      -- STR R2,0(R1) -- DATAMEM[R1] = R2
        result (8) := "EAF7FFF7"; -- 0x8      -- BAL main    -- PC = PC + (-7)
        return result;
    end init_mem;

    signal mem: RAM64x32 := init_mem;

begin
    Instruction <= mem(to_integer (unsigned (PC)));
end architecture;

```





Condition Field

Op. Code (Data Processing)

AL : Always : "1110"

ADD : "0100"

LT : Less Than : "1011"

MOV : "1101"

NV : Never : "1111"

CMP : "1010"

1.1. ADD (Immediate)

ADD Rd, Rn, #imm.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	0	1	0	0	0	Rn				Rd				0	0	0	0	imm.							

1.2. ADD (Register)

ADD Rd, Rn, Rm																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	0	1	0	0	Rn				Rd				0	0	0	0	0	0	0	0	Rm			

1.3. MOV (Immediate)

MOV Rd, #imm.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0	Rd				0	0	0	X	imm.							

X [bits] : Immediate Extensor

1.4. CMP (Immediate)

CMP Rn, #imm.																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	1	1	0	1	0	Rn				0	0	0	0	0	0	0	0	imm.							

2.1. LDR (Immediate)

LDR		Rd, [Rn, #Offset]																Offset															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	0	1	1	0	0	0	0	1	Rn				Rd				Offset													

2.2 STR (Immediate)

STR		Rd, [Rn, #Offset]																Offset															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	0	1	1	0	0	0	0	0	Rn				Rd				Offset													

3.1. B (Always)

B(AL)				label																Offset															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	0	1	0																												

3.2 B (If Less Than)

BLT		label																Offset															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	1	1	0	1	0	Offset																									