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.

Compétences :

- **Rendre compte de votre travail par la rédaction d'un compte rendu de projet**
- **Concevoir, synthétiser et valider une IP numérique décrite en VHDL RTL**

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…)

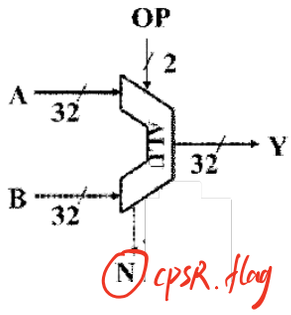
Unité Arithmétique Logique - ALU

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

- ADD : Y = A + B; OP = "00";
- B : Y = B; OP = "01";
- SUB : Y = A - B; OP = "10";
- A : Y = A; OP = "11";



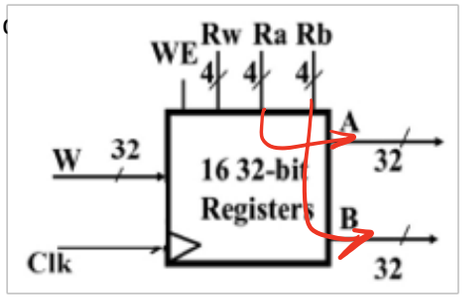
A l'issue de chaque opération, le drapeau N prend la valeur 1 si le résultat est strictement négatif, et 0 sinon.

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 donnés 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 d



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

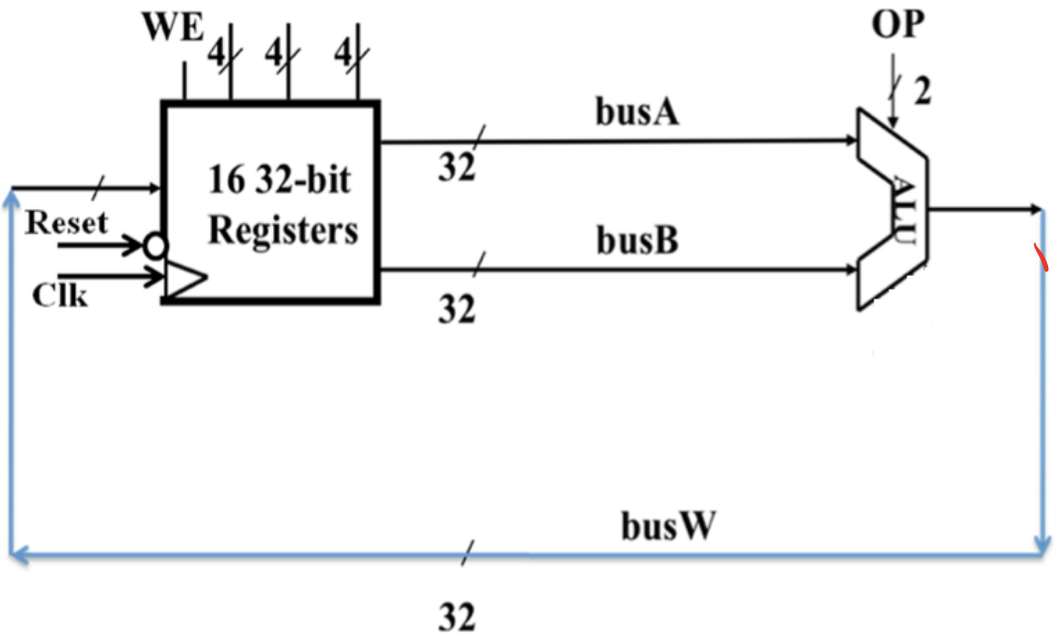
Le banc de registres sera déclaré comme étant un tableau de std_logic_vector. Son initialisation sera réalisée à l'aide d'une fonction (voir code ci-dessous)

```
-- 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;
```

⇒ pour faire la simulation

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 :

Au début R(15)=0x30 Rw Ra = 0x30 Rb

- 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)

ADD :	Y = A + B;	OP = "00";
B :	Y = B;	OP = "01";
SUB :	Y = A - B;	OP = "10";
A :	Y = A;	OP = "11";

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

Reset : reset asynchrone (actif à l'état haut) non représentés sur le schéma

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

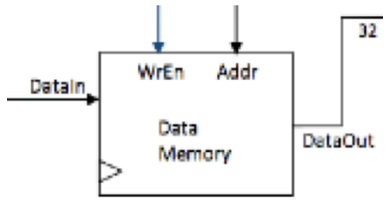
Si WE = 0, au front montant il n'y a pas d'écriture effectuée

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), …)*

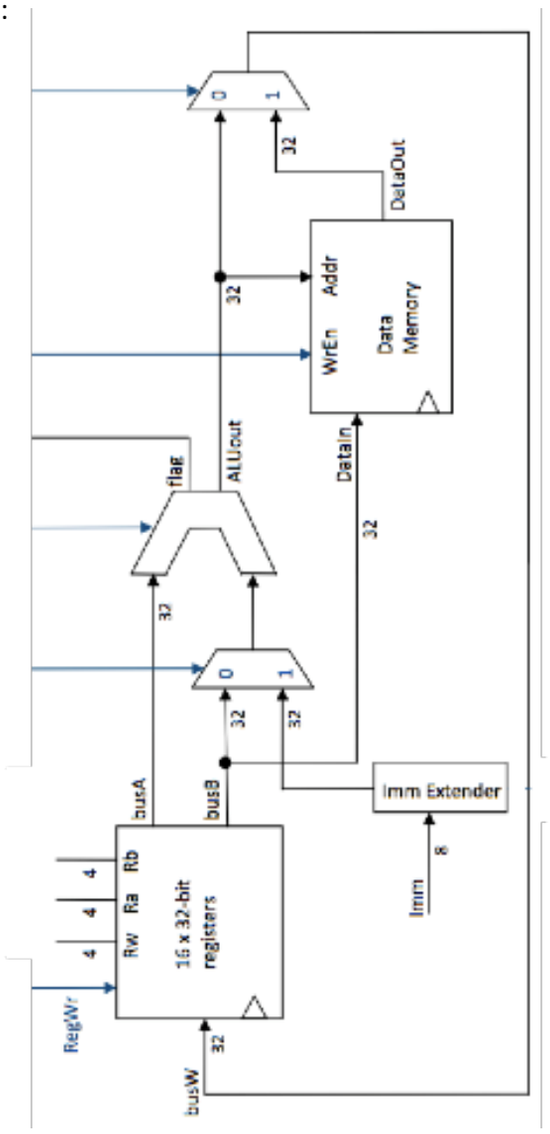
A FAIRE



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

Assemblage Unité de Traitement

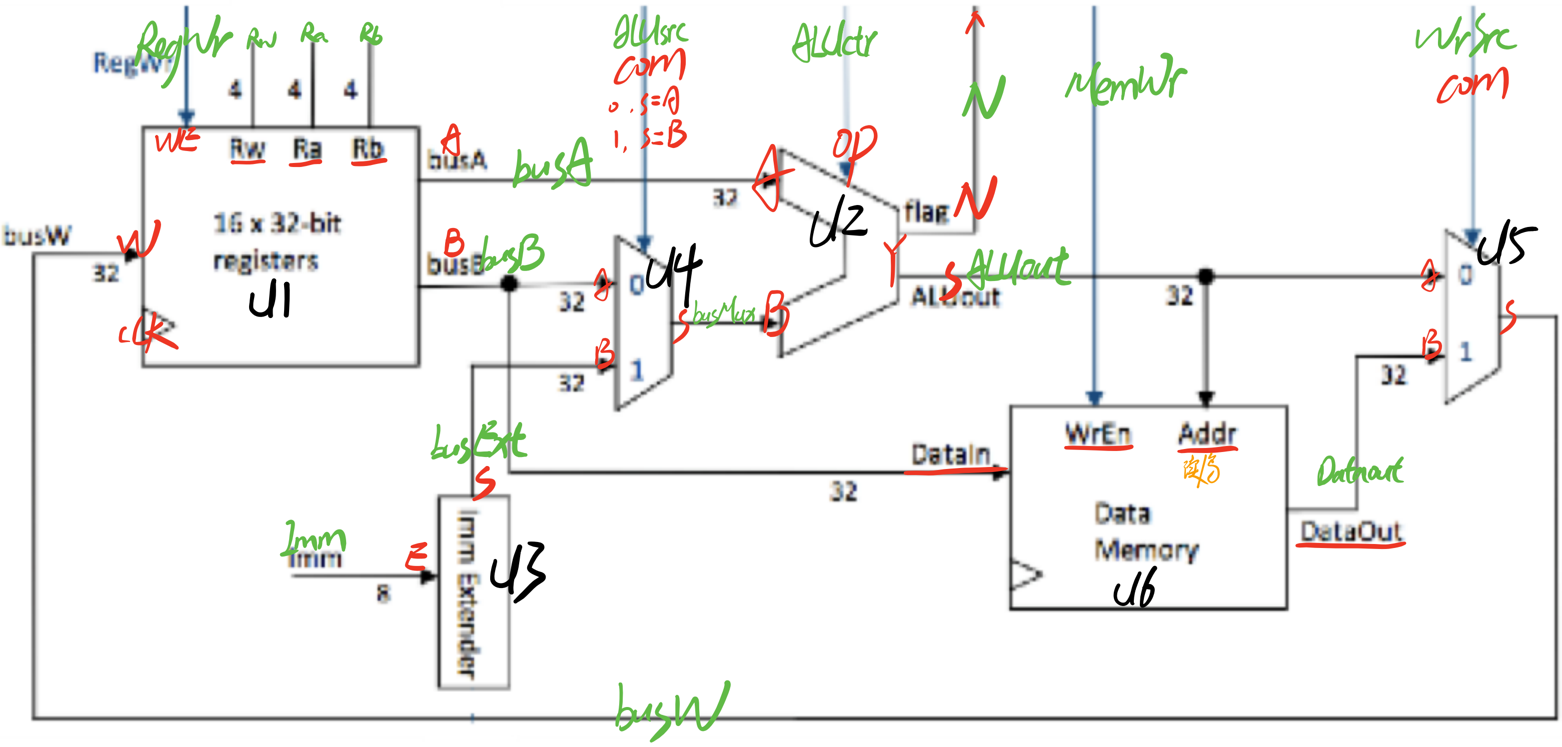
Les modules décrits précédemment doivent être assemblés conformément au schéma ci-dessous :



A FAIRE

1) Ecrire un module VHDL effectuant l'assemblage de l'unité de traitement.
2) Ecrire un banc de test 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.

ADD :	Y = A + B;	OP = "00";
B :	Y = B;	OP = "01";
SUB :	Y = A - B;	OP = "10";
A :	Y = A;	OP = "11";



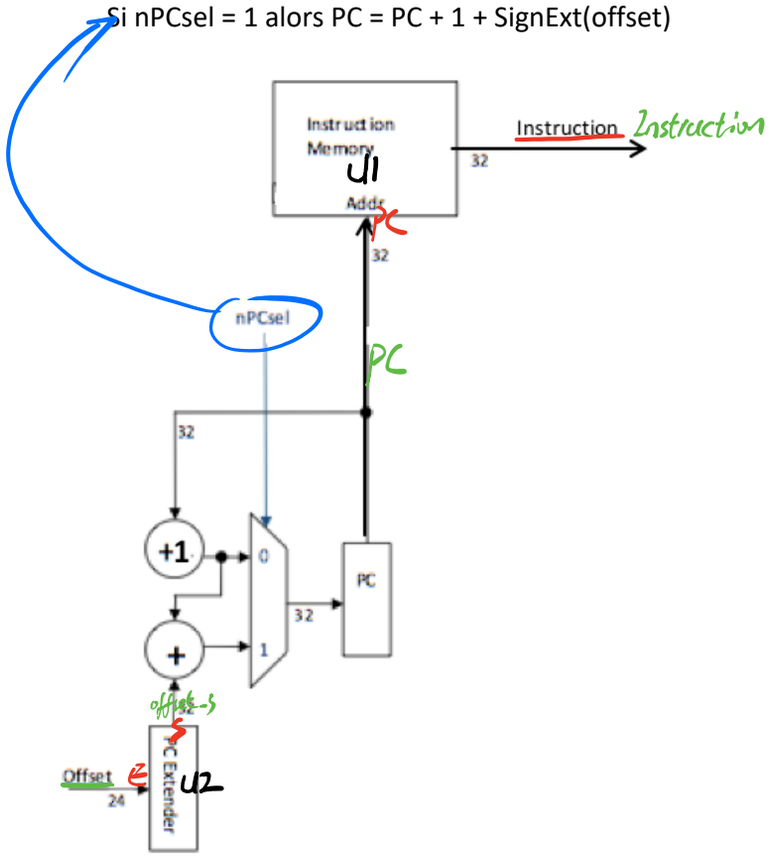
RegWr
4
4
4
Rw
Ra
Rb
16 x 32-bit
registers
busA
busB
busW
32
0
1
flag
ALUout
DataIn
WrEn
Addr
Data
Memory
DataOut
Imm
8
Imm Extender
aluSrc
com
ALUctr
MemWr
WrSrc
WE
W
clk
U1
U2
U3
U4
U5
U6
op
N
Y
busMux

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 fourni 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.

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 0x10 à 0x1A)

PARTIE 5 – TEST COMPLET DU PROCESSEUR

Afin de tester l'ensemble des instructions du processeur monocycle, on propose de simuler l'exécution d'un second petit programme en assembleur qui fait intervenir des instructions LOAD et STORE avec Offset.

```
main: MOV   R0,#0x10    @ R1 <= 0x10, adresse de TAB
      MOV   R1, #1              @ I=1
FOR:  LDR   R3, [R0]        @ TAB[i]
      LDR   R4, [R0,#1]           @ TAB[i+1]
      STR   R4, [R0]        @ TAB[i] <- R4
      STR   R3, [R0,#1]           @ TAB[i+1] <- R3
      ADD   R0, R0, #1      @ on incrémente l'adresse du Tab
      ADD   R1, R1, #1      @ i++
      CMP   R1, #0xA        @ si i<10, on saute au label FOR
      BLT   FOR
wait: BAL   wait
```

A FAIRE

1) Retrouvez le code binaire de ce petit programme à partir du code opératoire des instructions ARM7TDMI données en Annexe

2) Modifier le code VHDL de la mémoire d'instruction en créant un nouveau composant appelé instruction_memory2.vhd

3) Simuler l'exécution de ce deuxième programme de test par le processeur et vérifier son bon fonctionnement. Il sera nécessaire d'initialiser quelques données dans la Mémoire Data (de 0x10 à 0x1A)

PARTIE 6 – AUGMENTATION DU JEU D'INSTRUCTION

Dans la description de ce processeur monocycle, nous n'avons implémenté qu'une petite partie du jeu d'instruction du processeur ARM7TDMI.

Pour cette partie, nous proposons d'étendre ce jeu d'instruction avec les instructions BNE (Branch if Not Equal), STRGT (Store if Greater Then) et ADDGT afin de pouvoir tester un algorithme de tri à bulles.

Instruction BNE :

Pour rappel, l'instruction BLT teste la variable d'état N du PSR suite à une instruction CMP Rn, #Imm pour savoir si il faut continuer la séquence (N=0) ou sauter sur l'adresse du branchement (N=1).

L'instruction BNE ressemble beaucoup à l'instruction BLT sauf qu'elle va tester la variable d'état Z (Zero). Cette variable d'état permet d'indiquer si le résultat d'une opération arithmétique et logique est égal à zéro en sortie de l'UAL.

- Si le résultat est égal à zéro => Z= 1
- Si le résultat est différent de zéro => Z=0

Cette variable d'état Z est mise à jour dans le PSR (Program Status Register) au même titre que la variable N après chaque exécution de l'instruction CMP. Vous pouvez sélectionner un des 31 bit restant au PSR pour la position de Z.

Finalement, l'instruction BNE teste la variable d'état Z du PSR suite à une instruction CMP Rn, #Imm pour savoir si il faut continuer la séquence (Z=1) ou sauter sur l'adresse du branchement (Z=0).

Instruction STRGT et ADDGT :

Une des particularités des processeurs ARM, toutes les instructions peuvent s'exécuter de façon conditionnelle: une instruction sera exécutée ou non en fonction de l'état des bits N, Z, C, V

En assembleur ARM, il suffit d'ajouter au mot-clé de l'instruction un suffixe qui représente sa condition d'exécution.

Extension Mnémonique	Interprétation	Etat indicateur
EQ	Equal/equals zero	Z = 1
NE	Not Equal	Z = 0
LT	Lesser Then	N=1
GT	Greater Then	N = 0

Exemple d'exécution conditionnelle en fonction du résultat d'une comparaison

```
CMP r4, r5          @ comparer r4 et r5
SUBGT r4, r4, r5    @ si >, alors r4 ← r4 – r5
SUBLT r5, r5, r4    @ si <, alors r5 ← r5 – r4
```

Pour les instructions STRGT et ADDGT, il suffit de tester l'état de N pour savoir si l'instruction doit être exécutée ou pas.

Finalement l'instruction STRGT teste la variable d'état N du PSR suite à une instruction CMP Rn, Rm pour savoir si il faut l'exécuter (N=0) ou pas (N=1).

Algorithme du tri à bulles

Le principe du tri à bulles est de comparer deux valeurs adjacentes et d'inverser leur position si elles sont mal placées. Ainsi pour trier une liste dans l'ordre croissant, si un premier nombre x est plus grand qu'un deuxième nombre y, alors x et y sont mal placés et il faut les inverser. Si, au contraire, x est plus petit que y, alors on ne fait rien et l'on compare y à z, l'élément suivant. On parcourt ainsi entièrement la liste, puis on recommence jusqu'à ce qu'il n'y ait plus aucun élément à inverser.

Voici un programme assembleur qui trie les éléments d'un tableau TAB = {3, 107, 27, 12, 322, 155, 63}.

```
@ Mémoire de données
0x20: 3
0x21:  107
0x22:  27
0x23:  12
0x24: 322
0x25: 155
0x27:  63
@mémoire de programme
start:  MOV   R0, #0x20          @ R0 <- 0x20, adresse de TAB
        MOV   R2, #1             @ nombre de permutations
WHILE:  MOV   R2, #0             @ permut = 0
        MOV   R1, #1             @ I=1
FOR:    LDR   R3, [R0]           @ TAB[i]
        LDR   R4, [R0,#1]        @ TAB[i+1]
        CMP   R3, R4
        STRGT R4, [R0]           @ TAB[i] <- R4
        STRGT R3, [R0,#1]        @ TAB[i+1] <- R3
        ADDGT R2, R2, #1         @ NB_PERMUT = NB_PERMUT + 1
        ADD   R0, R0, #1         @ on incrémente l'adresse du Tab
        ADD   R1, R1, #1          @ i++
        CMP   R1, #0x7           @ si i<7, on saute sur la boucle FOR
        BLT   FOR
        CMP   R2, #0             @ test permut
        MOV   R0, #0x20          @ on remet l'adresse de base du tableau dans R0
        BNE   WHILE              @ si permut est diffférent de 0 on saute à WHILE
@ Pour finir, on bloque le processeur dans une boucle infinie vide
wait:   BAL   wait
```

A FAIRE

1) Retrouvez le code binaire du programme assembleur permettant de réaliser l'algorithme de tri à bulles à partir du code opératoire des instructions ARM7TDMI donnés en Annexe.

2) Modifier le code VHDL de la mémoire d'instruction en créant un nouveau composant appelé instruction_memory3.vhd

3) Simuler l'exécution de ce troisième programme de test par le processeur et vérifier son bon fonctionnement. Il sera nécessaire d'initialiser quelques données dans la Mémoire Data (de 0x20 à 0x27)

ANNEXES

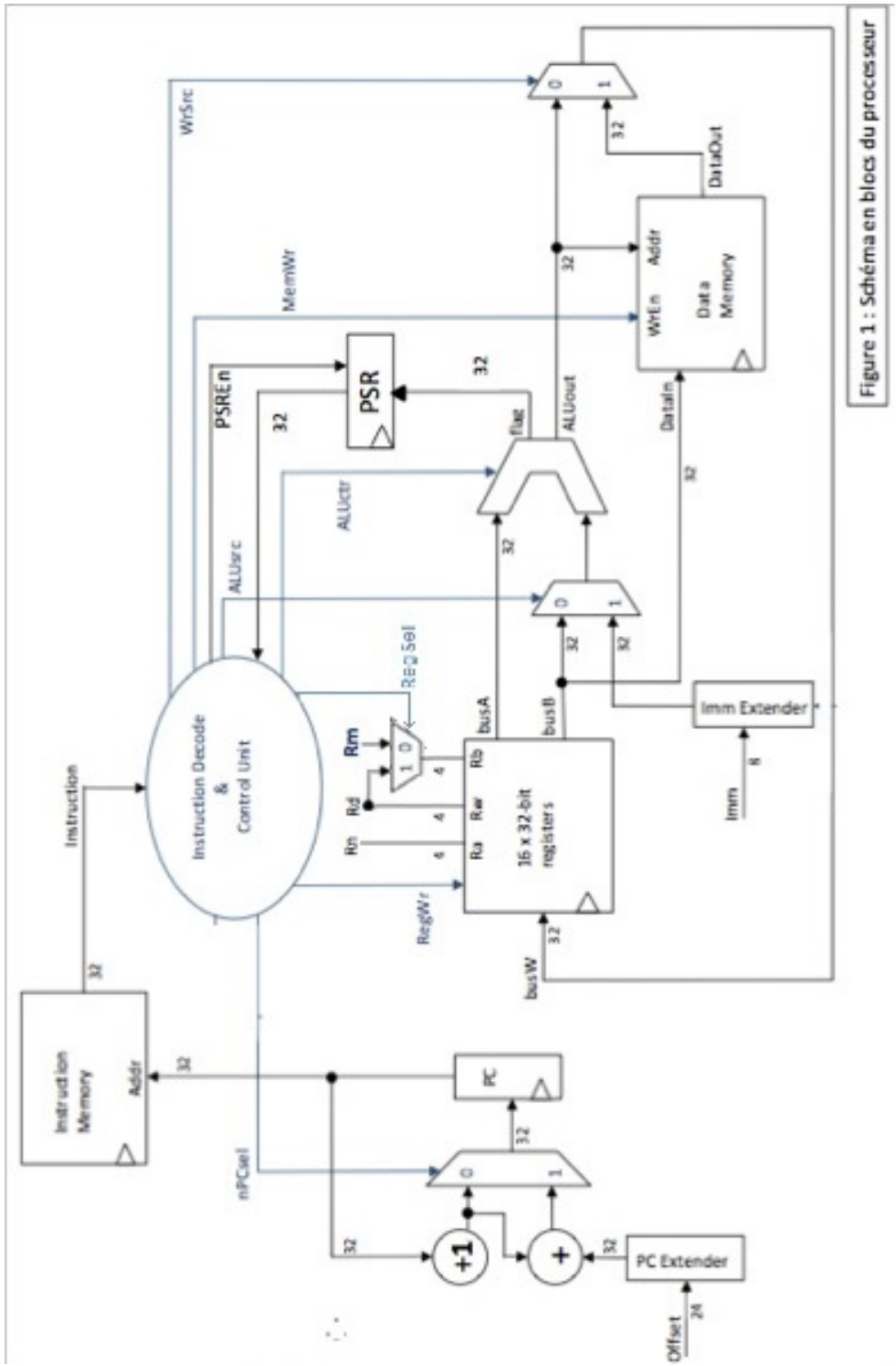


Figure 1 : Schéma en blocs du processeur

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

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):=x"E3A01010";-- 0x0 _main -- MOV R1,#0x10 -- R1 = 0x10
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;

signal mem: RAM64x32 := init_mem;

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

tement de Données (ADD, MOV, CMP)

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
ond	0	0	I	Op. Code				S	Rn				Rd				Operand 2											

Immediate Selection
Set Status Flags

I (bit25) = 0

11	10	9	8	7	6	5	4	3	2	1	0
Shift								Rm			

I (bit25) = 1

11	10	9	8	7	6	5	4	3	2	1	0
Rot.				Imm.							

s Mémoire (LDR, STR)

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
ond	0	1	I	P	U	B	W	L	Rn				Rd				Offset											

: Pre / Post Indexing
: Up / Down Displacement
Byte / Word Access
: Write Back (updates Rn)
Load / Store

I (bit25) = 0

11	10	9	8	7	6	5	4	3	2	1	0
Shift								Rm			

I (bit25) = 1

11	10	9	8	7	6	5	4	3	2	1	0
Rot.				Imm.							

achements (B, BLT)

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
ond	1	0	1	L	Offset																							

Link Bit (updates Link Register)

指令的编码 decodeur (在cur.pdf里也有)

Condition Field
AL : Always : "1110"
LT : Less Than : "1011"
NV : Never : "1111"

Op. Code (Data Processing)
ADD : "0100"
MOV : "1101"
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	1	0	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 (bit8) : Immediate Extendor

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	0	1	0	1	Rn				0	0	0	0	0	0	0	0	Imm.							

2.1. LDR (I ediate)

LDR Rd, #Offset]

31	30	29	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	1	0	0	0	0	1	Rn				Rd				Offset											

2.2 STR (I ediate)

STR Rd, #Offset]

31	30	29	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	1	0	0	0	0	0	Rn				Rd				Offset											

3.1. B (Alw)

B(AL) la

31	30	29	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	1	0	1	0	Offset																							

3.2. B (If L Than)

BLT lab

31	30	29	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	0	1	0	Offset																							