FLEISER
Victor
EISE3

Projet VHDL Processeur MonoCycle

COMPTE-RENDU

Sommaire

Description du rendu....2
Contenu du dossier rendu....2
Partie 1 – Unité de Traitement....3
Objectif....3
Fonctionnement....3
TEST-BENCH....4
Contenu du code....4
Simulation....4
Partie 2 – Unité de Gestion des instructions....6
Objectif....6
Fonctionnement....6
TEST-BENCH....7
Contenu du code....7
Simulation....7
Partie 3 – Unité de contrôle....8
Objectif....8
Fonctionnement....8
Contenu du code....9
Tableau des commandes....10
Partie 4 – Assemblage et validation du Processeur....11
Objectif....11
Fonctionnement....11
Programme test du Processeur....12
TEST-BENCH....12
Contenu du code....12
Simulation....13
Partie 5 – Test du processeur sur carte FPGA....14
Objectif....14
Fonctionnement....14
Implémentation sur carte FPGA....15
Partie 6 – Gestion des interruptions externes....16
Objectif....16

Description du rendu

L’objectif du projet est de créer un processeur en VHDL (Parties 1 à 4) pour ensuite l’implémenter sur carte FPGA (Partie 5) et ajouter des éléments supplémentaires tels que la gestion des interruptions (Partie 6) ainsi que un périphérique UART (Partie 7).

J’ai terminé les parties 1 à 5, avec le code VHDL de la partie 6 commencé.

Contenu du dossier rendu

- COMPTE-RENDU.pdf (ce document)

- Dossier ProjetProcesseurMonocycle_docs :

    - Images des test bench principaux (inclus dans le compte rendu)

    - Tableau de la partie 3 (inclus dans le compte rendu)

    - Images des schémas utilisés (inclus dans le compte rendu)

- Dossier ProjectProcesseurMonocycle_Quartus :

    - Dossier Fit :

        - Projet Quartus (top_level.qpf)

    - Dossier Simu :

        - Code VHDL des Test-Bench

        - Script de simulation du Test-Bench de l’unité de traitement (simu_unite_traitement.do)

        - Script de simulation du Test-Bench de l’unité de gestion des instructions (simu_unite_gestion_instruction.do)

        - Script de simulation du Test-Bench du processeur entier (simu_processeur.do)

    - Dossier Src :

        - Code VHDL de toutes les entités

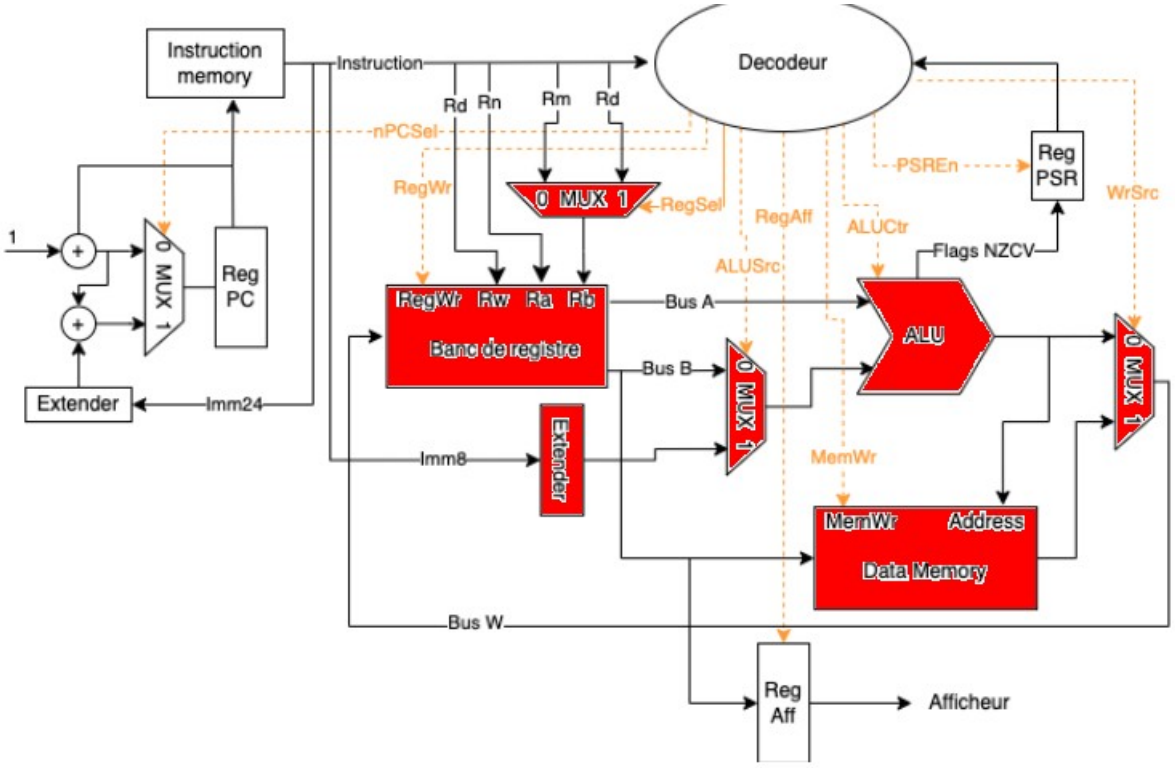
Partie 1 – Unité de Traitement

Objectif

L'objectif de cette partie est de créer l'unité de traitement, elle permet de faire les opérations et lectures/écritures, entre autres elle permet de :

- Lire/écrire sur les registre stockés
- Lire/écrire sur la mémoire stockée
- Faire des opérations à partir de registres et/ou immédiats
- Renvoyer des Flags à l'unité de contrôle

Fonctionnement



Éléments :

- Banc de registre : Entité stockant un tableau de 16 registres 32 bits, elle possède des adresses en entrées :
    - Rw : Adresse du registre sur lequel écrire (si le signal RegWr est '1')

    - Ra : Adresse du registre du premier opérande de l'instruction courante, la valeur du registre est renvoyée sur BusA vers l'ALU

    - Rb : Adresse du registre du deuxième opérande de l'instruction courante OU de la destination de l'instruction courante (en fonction du signal RegSel grâce à un multiplexeur 2v1), la valeur du registre est renvoyée sur BusB vers le reste de l'unité de traitement

Le contenu écrit provient de l'ALU OU de la mémoire (en fonction du signal WrSrc grâce à un multiplexeur 2v1)

- Data Memory : Entité stockant un tableau de 64 registres 32 bits, on peut lire ou écrire le registre d'adresse donné par l'ALU en fonction du signal MemWr

- Extender : Entité prenant une partie immédiate de 8 bits depuis l'instruction courante et l'étendant à 32 bits pour pouvoir faire des opérations dessus

- ALU (Arithmetic-Logic Unit) : Entité faisant les opérations (somme, opérande 2, soustraction, opérande 1, OR, AND, XOR, NOT), les opérandes sont :
    - Le contenu du registre pointé par l'opérande 1 de l'instruction courante
    - Le contenu du registre pointé par l'opérande 2 de l'instruction courante OU la partie immédiate étendue de l'instruction courante (en fonction du signal ALUSrc grâce à un multiplexeur 2v1)

TEST-BENCH

Contenu du code

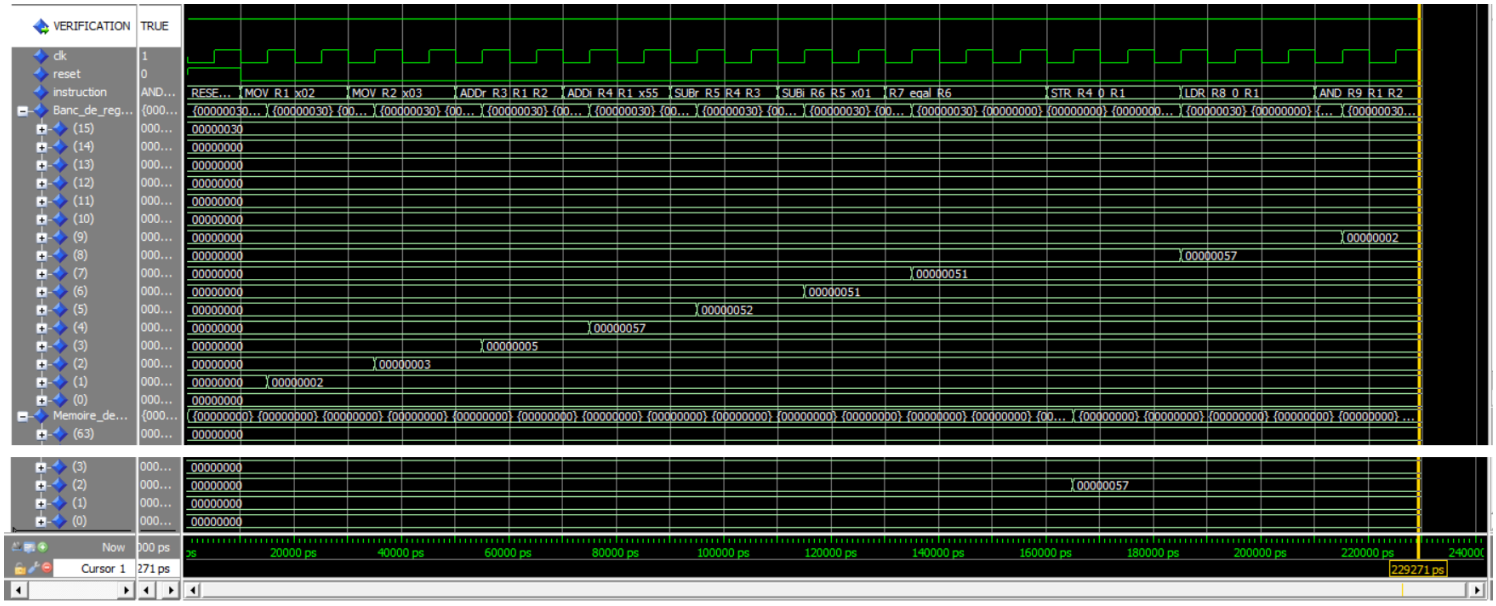
- Signal de vérification OK (appelé VERIFICATION dans la simulation)

- Des déclarations de types pour récupérer des array de l'UUT (memory et registers) ainsi que un type possédant le nom des instructions pour pouvoir l'afficher sur le test bench.

- Process pour la Clock

- Process de vérification

Simulation



Set d’instructions testé : (également affiché dans le test bench de la simulation)
RESET initial

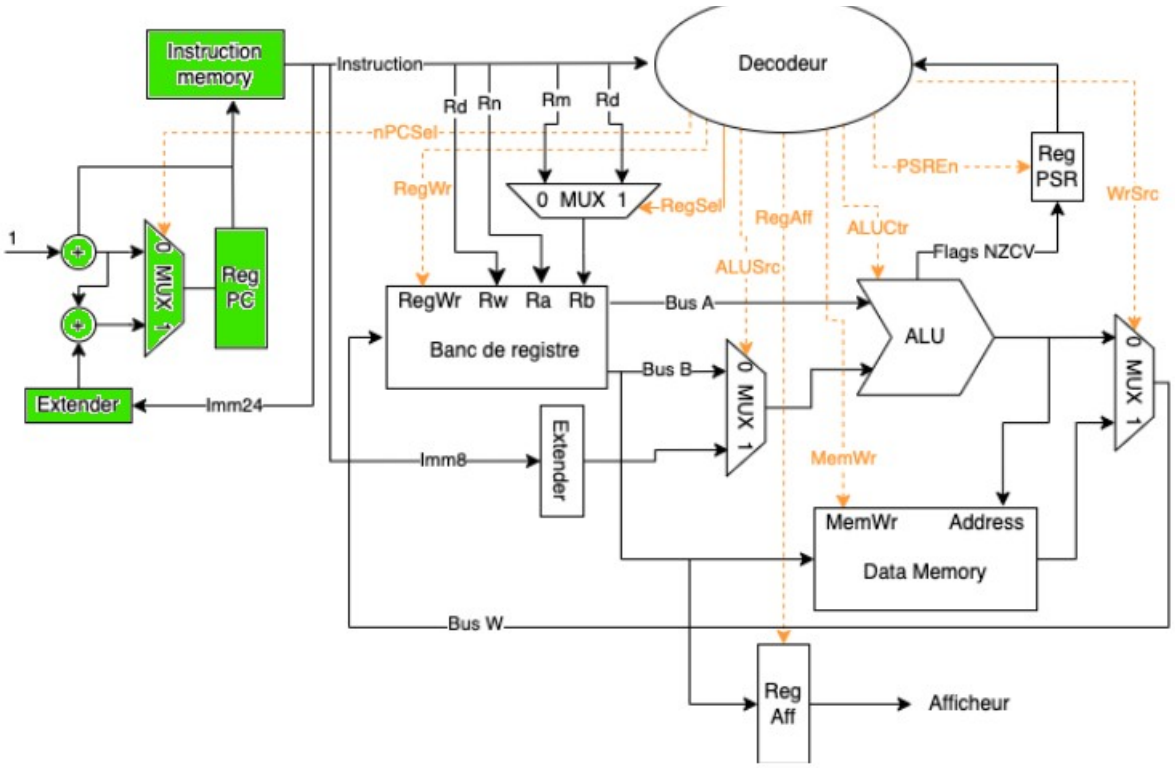
```
MOV R1, 0x02          -- R1 = 0x2
MOV R2, 0x03          -- R2 = 0x3
ADDr R3, R1, R2       -- R3 = R1 + R2 = 0x5
ADDi R4, R1, 0x55     -- R4 = R1 + 0x55 = 0x57
SUBr R5, R4, R3       -- R5 = R4 – R3 = 0x52
SUBi R6, R5, 0x01     -- R6 = R5 – 0x1 = 0x51
R7 egal R6            -- R7 = R6 = 0x51
STR R4, 0 (R1)        -- mem(R1) = R4 = 0x57
LDR R8, 0 (R1)        -- R8 = mem(R1) = 0x57
AND R9, R1, R2        -- R9 = R1 AND R2 = 0x2
```

Partie 2 – Unité de Gestion des instructions

Objectif

L'objectif de cette partie est de créer l'unité de gestion des instructions, elle permet de stocker le PC et de faire des opérations dessus, elle permet également de stocker les instructions du programme et de fournir l'instruction courante.

Fonctionnement



Éléments :

- RegPC : Registre stockant le PC actuel

- Extender : Entité prenant une partie immédiate de 24 bits et l'étendant à 32 bits pour faire des opérations de décalage du PC

- Instruction memory : Entité stockant un tableau d'instructions 32 bits et fournissant l'instruction courante correspondant au PC au reste du processeur

- Multiplexeur 2v1 et opérations : Permet soit d'incrémenter le PC, soit de le décaler par la valeur immédiate (étendue) en fonction de nPCSel provenant de l'unité de contrôle

TEST-BENCH

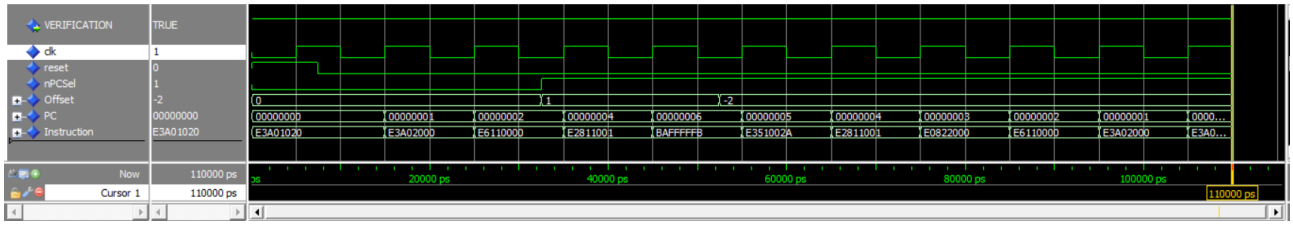
Contenu du code

- Signal de vérification OK (appelé VERIFICATION dans la simulation)

- Process pour la Clock

- Process de vérification de l'instruction courante sortant de l'UUT.

Simulation



L'instruction courante évolue correctement :

- En premier temps on incrémente normalement le PC en désactivant nPCSel

- Ensuite on teste le décalage du PC de 1+1=2 instructions pour tester les décalages positifs

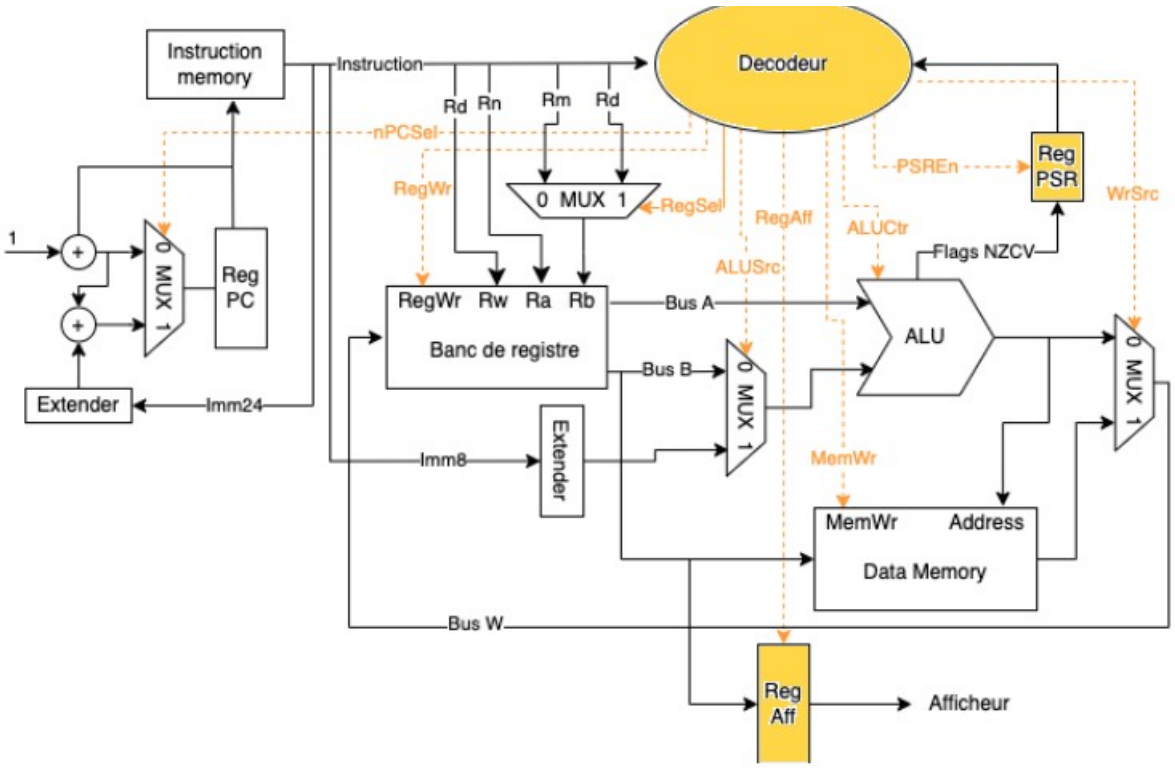
- Enfin on teste le décalage du PC de -2+1=-1 instructions pour tester les décalages négatifs

Partie 3 – Unité de contrôle

Objectif

L’objectif de cette partie est de créer l’unité de contrôle, elle permet de décoder l’instruction courante ainsi que les flags de l’ALU afin de contrôler ce que les autres éléments de l’unité de traitement et de gestion des instructions devraient faire, ainsi que les données passée à l’afficheur.

Fonctionnement



Éléments :

- Décodeur : Décode l’instruction courante ainsi que les flags de l’ALU afin de gérer les signaux contrôlant le reste du processeur (unité de traitement, unité de gestion des instructions, registre d’affichage et registre PSR)

- Registre PSR : registre stockant les Flags de l’ALU sur les 4 bits de poids fort, activé par PSREn

- Registre Affichage : registre stockant les données 32 bits à afficher, activé par RegAff

Contenu du code

```
if (Instruction(27 downto 26) = "00") then              --Instruction de traitement :
    case Instruction(24 downto 21) is
        when "1101" => instr_courante <= MOV;                                   --MOV
        when "1010" => instr_courante <= CMP;                                   --CMP
        when "0100" => instr_courante <= ADDi when (Instruction(25) = '1') else --ADDi
        ADDr when Instruction(25) = '0' else                                    --ADDr
        UNDEFINED;
        when others => instr_courante <= UNDEFINED;
    end case;

elsif (Instruction(27 downto 26) = "01") then           --Instruction de transfert :
    if (Instruction(20) = '0') then                     -- Link = 0 = Store
        instr_courante <= STR;                                                  --STR
    elsif (Instruction(20) = '1') then                  -- Link = 1 = Load
        instr_courante <= LDR;                                                  --LDR
    else
        instr_courante <= UNDEFINED;
    end if;

elsif (Instruction(27 downto 25) = "101") then          --Instruction de branchement relatif :

    if Instruction(31 downto 28) = "1011" then          --cond = LT
        instr_courante <= BLT;                                                  --BLT

    elsif Instruction(31 downto 28) = "1110" then       --cond = AL

        if (Instruction(24) = '0') then                 -- Link = 0 = Branch
            instr_courante <= B;                                                --B
        elsif (Instruction(24) = '1') then              -- Link = 1 = Branch and link
            instr_courante <= UNDEFINED;                                        --BAL (NOT IMPLEMENTED)
        else
            instr_courante <= UNDEFINED;
        end if;
    else
        instr_courante <= UNDEFINED;
    end if;

else
    instr_courante <= UNDEFINED;
end if;
```

Le décodeur d'instructions décode les instructions de la manière suivante :

- Choix du type d'instructions par les bits 25-27 (instruction de traitement/transfert/branchement relatif)
    1) Instruction de traitement : Choix de l'instruction par les bits 24-21
        - Dans le cas de ADD : Choix registre/immédiat par le bit 25
    2) Instruction de transfert : Choix de l'instruction Store/Load par le bit 20
    3) Instruction de branchement relatif : Choix de l'instruction par les bits 31-28

Note : Il y a également un type UNDEFINED pour toute instruction non implémentée.

Tableau des commandes

Le décodeur d'instructions met ensuite les signaux contrôlant le reste du processeur à jour suivant le tableau suivant :

INSTRUCTION	nPCSel	RegWr	ALUSrc	ALUCtr	PSREn	MemWr	WrSrc	RegSel	RegAff
ADDi ADD Rd,Rn,imm; --Rd <= Rn + imm	0	1	1	00	- 0	0	0	- 0	0
ADDr ADD Rd,Rn,Rb; --Rd <= Rn + Rb	0	1	0	00	- 0	0	0	0	0
BAL BAL label	1	0	- 0	- 00	- 0	0	- 0	- 0	0
BLT BLT label; --branchmnt si FLAG N = 1	1 ou 0 FLAG N	0	- 0	- 00	0	0	- 0	- 0	0
CMP CMP Rn,imm; --Rn ?= imm -> FLAGS	0	0	1	010	1	0	- 0	0	0
LDR LDR Rd,imm(Rn); --Rd <= MEM[Rn+imm]	0	1	1	000	- 0	0	1	- 0	0
MOV MOV Rd,imm;--Rd <= imm	0	1	1	001	- 0	0	0	- 0	0
STR STR Rd,imm(Rn); --MEM[Rd+imm] <= Rn	0	0	1	000	- 0	1	- 0	1	1

Note : Ne pas prendre en compte les signes '-' dans le tableau (ils correspondent aux signaux dont la valeur n'importe pas en théorie, cela dit je n'ai pas revérifié chaque case, il y a donc potentiellement des '-' en trop ou manquant).

Note : Il aurait été possible et plus logique de mettre les signaux de contrôle aux bonnes valeurs au fur et à mesure de l'interprétation de l'instruction (cf code plus haut) afin de pouvoir implémenter des nouvelles instructions plus facilement et logiquement.

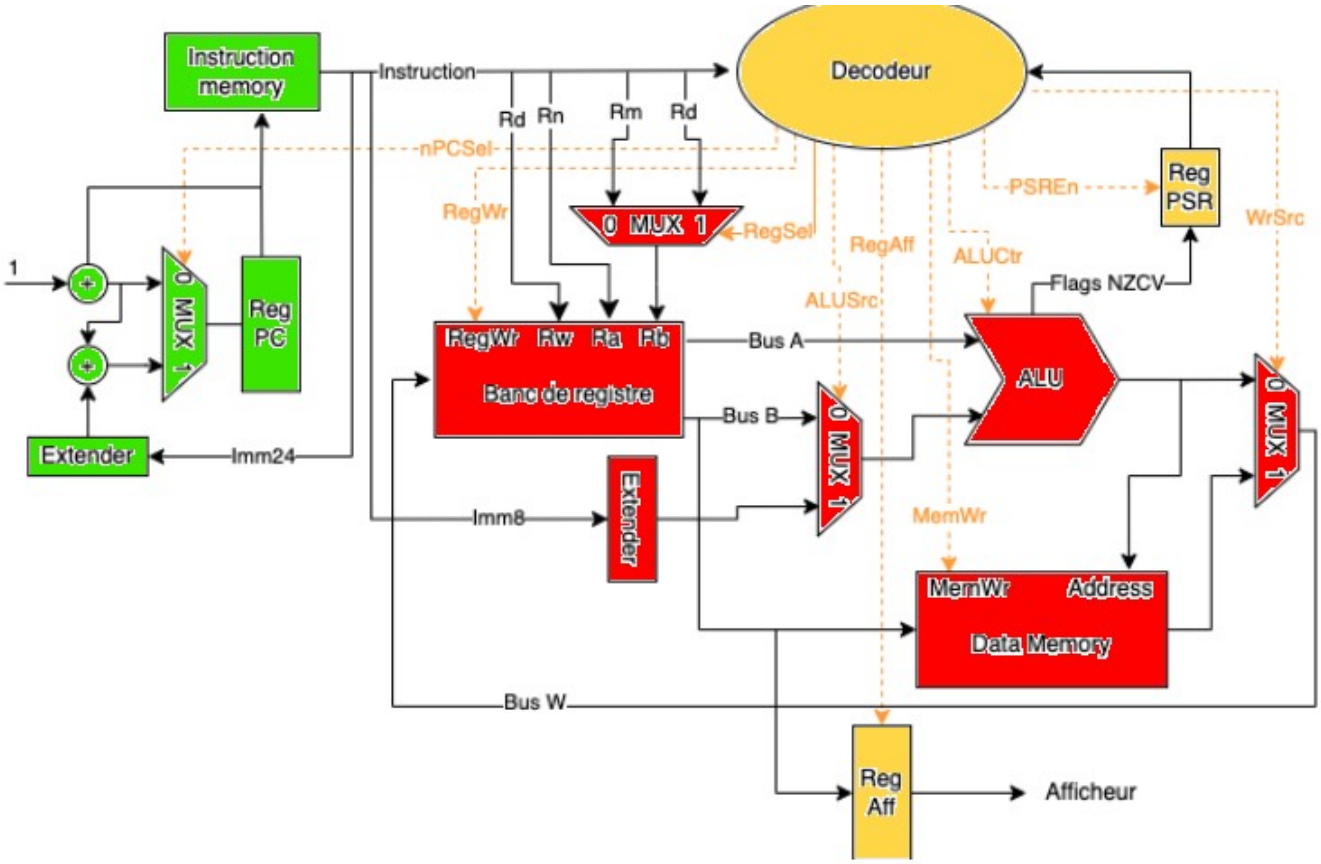
Note : Je n'ai pas fait de test-bench pour cette partie car les erreurs possibles sont principalement des erreurs sur le tableau précédent, et le code sera testé juste après dans la partie 4 lors de l'assemblage du processeur.

Partie 4 – Assemblage et validation du Processeur

Objectif

L’objectif de cette partie est de combiner le processeur en utilisant les 3 parties précédentes.

Fonctionnement



Éléments :

- Rouge : Unité de Traitement : cf Partie 1

- Vert : Unité de gestion d’Instructions : cf Partie 2

- Jaune : Unité de Contrôle : cf Partie 3

Programme test du Processeur

Pour tester le processeur on utilise le code suivant dans l'instruction_memory :

```
end loop;                  -- PC        -- 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)
    -- fonction additionne les elements de la memoire de @0x20 à @0x2A (inclus) puis le stocke dans la case mémoire suivante (@0x2B). ensuite relance la fonction
    -- label : main                          //debut de la fonction
    -- r1 = @0x20                            //adresse du debut du tableau
    -- r2 = 0                                //variable stockant le resultat de la somme
    -- while (r1 < 0x2A)                     //itere sur le tableau entre 0x20 et 0x2A
    --      r0 = mem[r1]                     //recupere le nouvel element
    --      r2 += r0                         //additionne le nouvel element
    --      r1++                             //incremente l'adresse
    -- mem[r1] = r2                          //ecrit le resultat dans la case 0x2A (ou 0x2B?)
    -- goto : main                           //relance la fonction
```

TEST-BENCH

Contenu du code

- Signal de vérification OK (appelé VERIFICATION dans la simulation)

- Des déclarations de types pour récupérer des array de l'UUT (memory et registers) ainsi que pour les tests

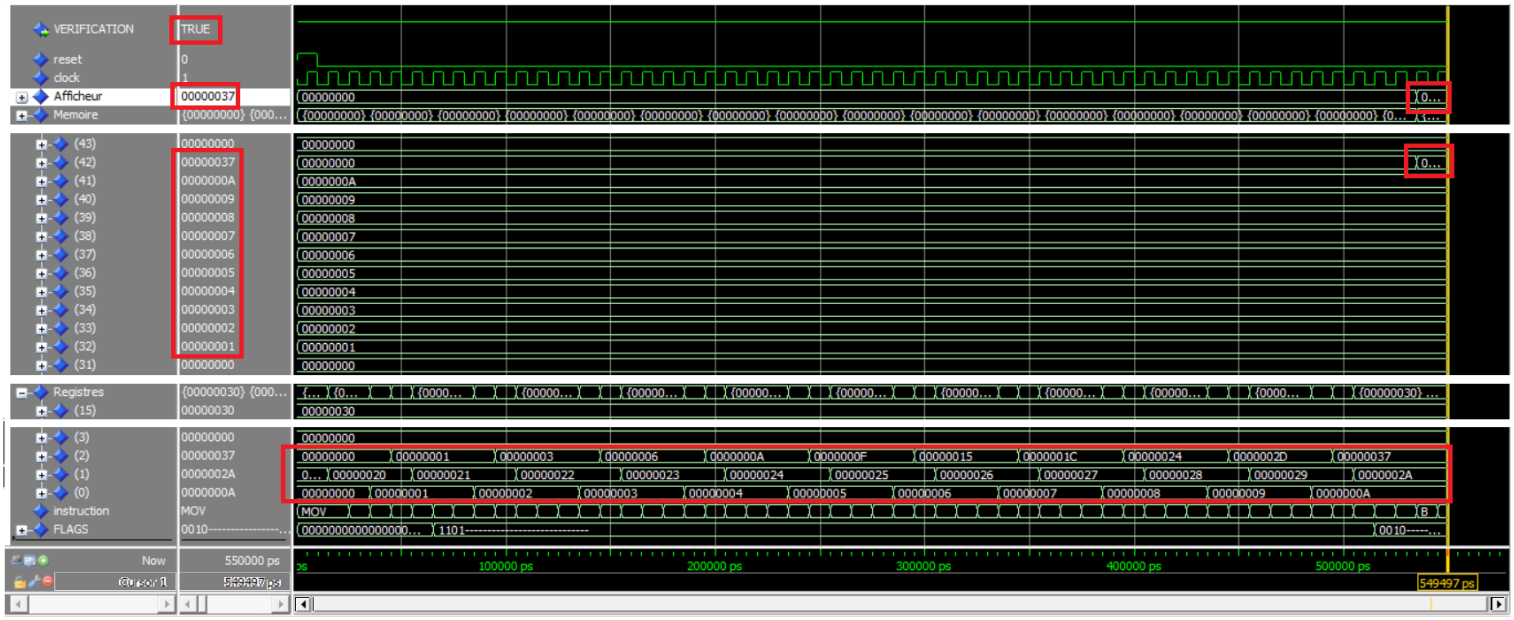
- Process pour la Clock

- Process de vérification des registres/mémoire affecté après chaque instruction.

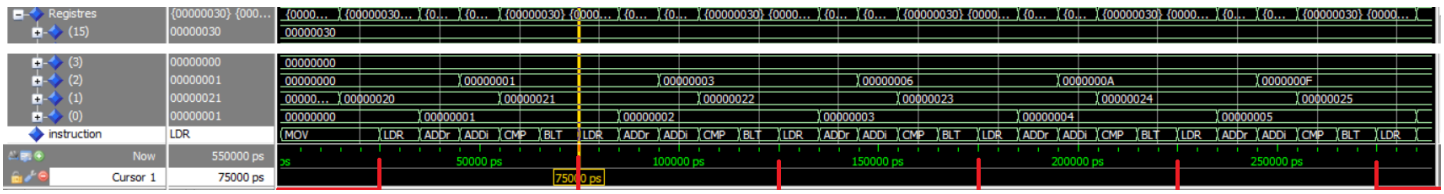
```
94      --2EME INSTRUCTION FINIE
95      --MOV R2,#0x00
96      TESTS_registers(2) <= << signal.processeur_tb.UUT.unite_traitement.registers.Banc : table >>(2);--load register 2
97      wait for 5 ns;  --32.5 NS
98      if (TESTS_registers(2) /= x"00000000") then                         --test si c'est la bonne valeur
99          OK <= False;
100     end if;
101     wait for 5 ns;  --37.5 NS
102
103     --Il y a 10 itérations de la boucle
104     for i in 0 to 9 loop
105
106         --3EME INSTRUCTION FINIE
107         --LDR R0,0(R1)
108         TESTS_registers(0) <= << signal.processeur_tb.UUT.unite_traitement.registers.Banc : table >>(0);--load register 0
109         wait for 5 ns;
110         if (TESTS_registers(0) /= << signal.processeur_tb.UUT.unite_traitement.memory.Memory : mem >>(to_integer(unsigned(TESTS_registers(1)))
111             OK <= False;
112         end if;
113
114         TESTS_registre_tempo <= << signal.processeur_tb.UUT.unite_traitement.registers.Banc : table >>(2);--load register 2
115         wait for 5 ns;
116
117         --4EME INSTRUCTION FINIE
118         --ADD R2,R2,R0
119         TESTS_registers(2) <= << signal.processeur_tb.UUT.unite_traitement.registers.Banc : table >>(2);--load register 2
120         wait for 5 ns;
121         if (TESTS_registers(2) /= std_logic_vector(to_unsigned(to_integer(unsigned(TESTS_registre_tempo))+to_integer(unsigned(TESTS_registers(
122             OK <= False;
123         end if;
124
125         TESTS_registre_tempo <= << signal.processeur_tb.UUT.unite_traitement.registers.Banc : table >>(1);--load register 1
126         wait for 5 ns;
127
128         --5EME INSTRUCTION FINIE
129         --ADD R1,R1,#1
130         TESTS_registers(1) <= << signal.processeur_tb.UUT.unite_traitement.registers.Banc : table >>(1);--load register 1
131         wait for 5 ns;
```

Méthode pour vérifier après chaque instruction si le code est correct : boucle for pour relancer cette partie du test bench, on cherche directement dans l'UUT les logic_vector qui nous intérèssent pour les stocker dans des signaux de tests, et enfin une vérification invalidant le booléen OK si il y a un problème

Simulation



Les registres et la mémoire évoluent correctement :
- La mémoire 0x20 à 0x2A contient bien les valeurs initialisées
- La mémoire 0x2B récupère bien la somme des 10 valeurs précédentes (1+2+...+10 = 55 = 0x37)
- Le registre 2 contient bien la somme après chaque itération
- Le registre 1 contient bien l'adresse de la case itérée
- Le registre 0 contient bien la valeur de la case itérée
- L'afficheur est bien mis à jour à la fin de l'instruction STR



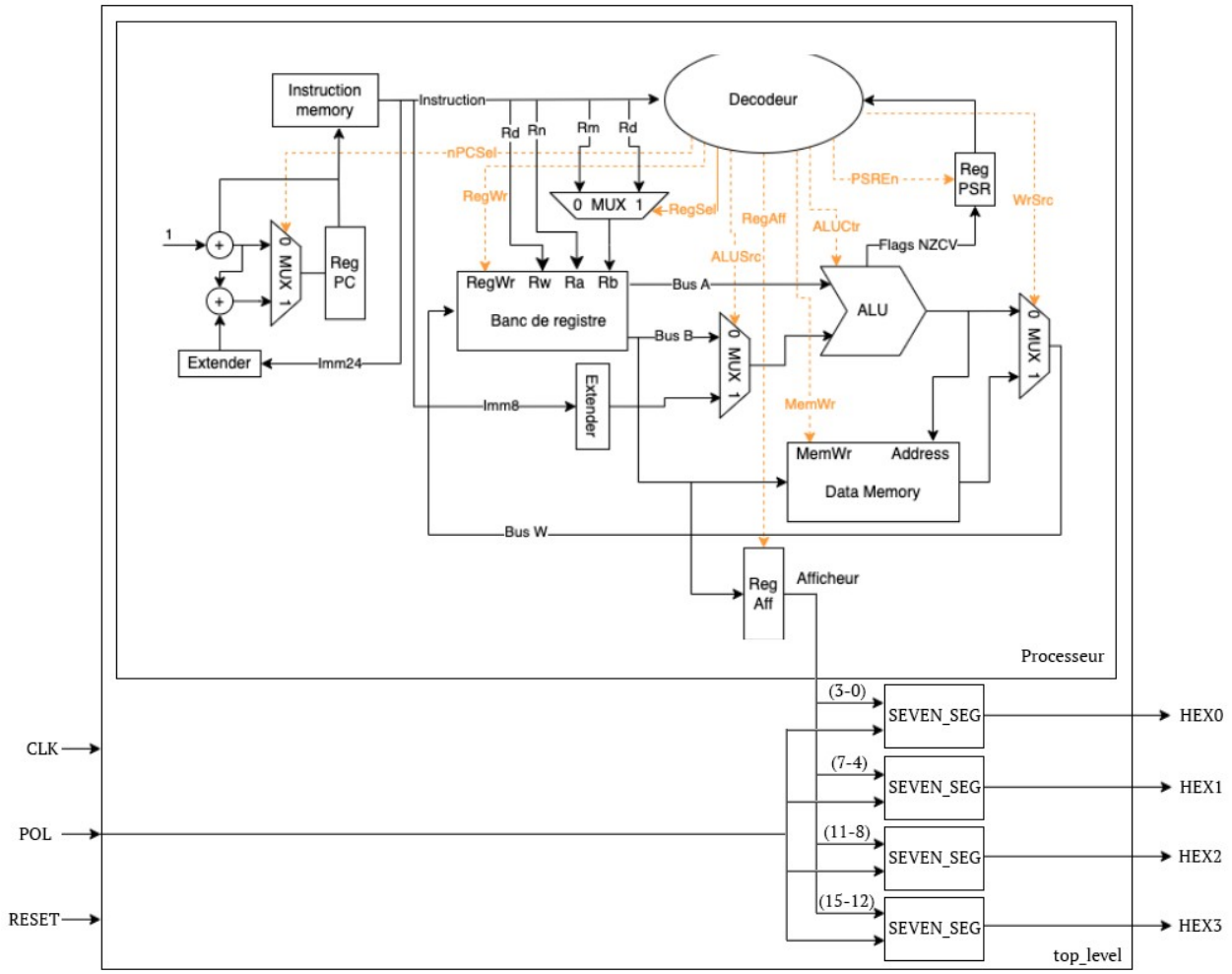
Zoom sur les 5 premières itérations.

Partie 5 – Test du processeur sur carte FPGA

Objectif

L'objectif de cette partie est d'implémenter le processeur sur la carte FPGA afin d'afficher les 16 derniers bits en hexadécimal sur 4 afficheurs sept segments de la carte.

Fonctionnement



Éléments :

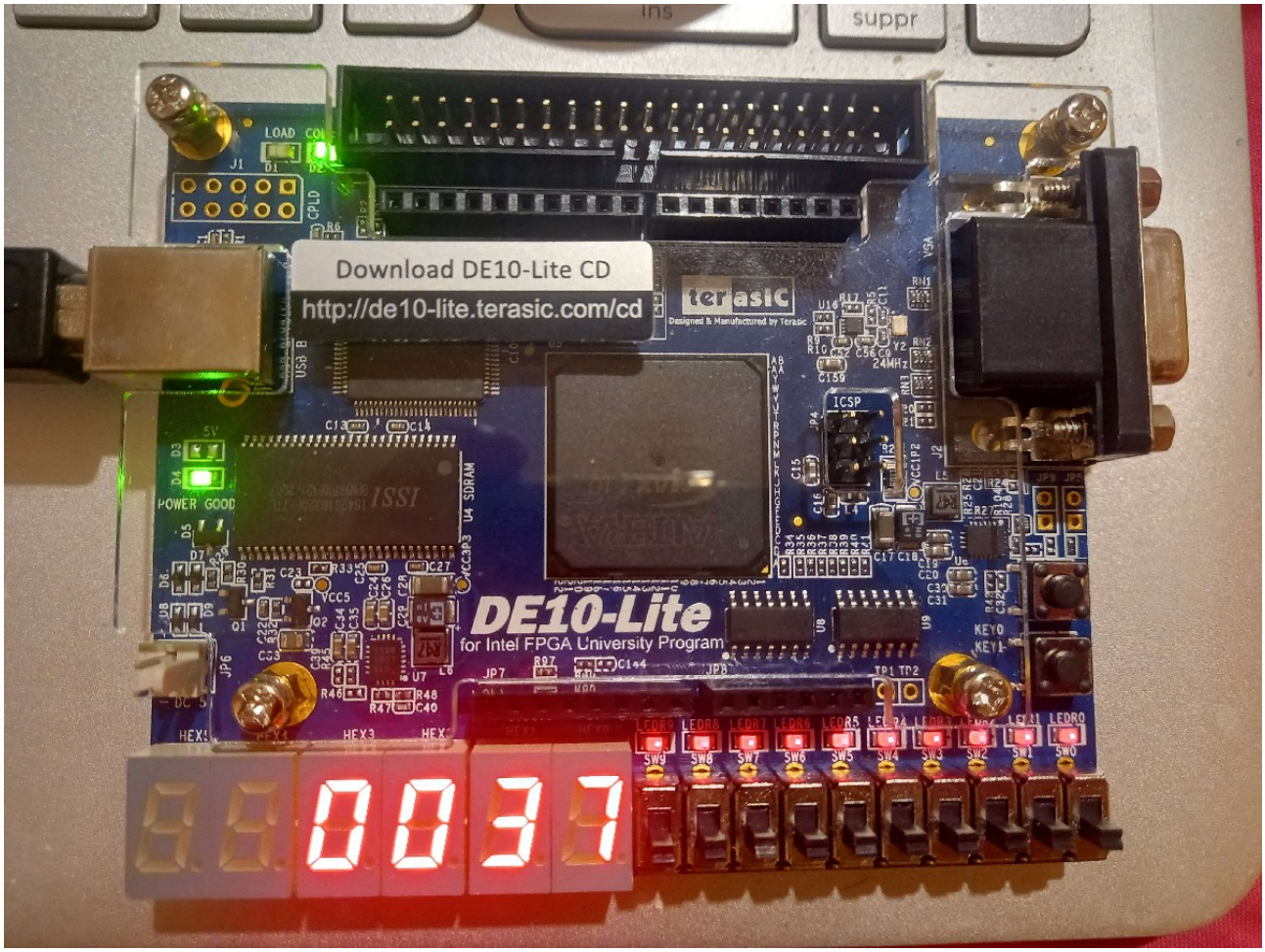
- Processeur : cf Partie 4

- SEVEN_SEG : Cette entité permet de convertir un nombre BCD de 4 bits en un signal correspondant aux sept segments d'affichage, pour représenter visuellement le nombre BCD.

- top_level : Cette entité combine le Processeur et les SEVEN_SEG, il est le niveau le plus haut du projet sur Quartus et renvoie à la carte les 4 signaux pour l'affichage sept segments. Il reçoit de la carte la clock, le reset (bouton à droite de la carte) ainsi qu'un signal Pol pour inverser la polarité de l'affichage sept segments (interrupteur le plus à gauche)

Note : Le code SEVEN_SEG.vhd provient directement du fichier que j'ai complété lors du TP1-Compteur-BCD

Implémentation sur carte FPGA



Le nombre hexadécimal affiché est bien celui calculé dans la partie 4 (0x37).

Note : Il est possible d'inverser la polarité avec l'interrupteur le plus à gauche, et de reset avec le bouton de droite

Partie 6 – Gestion des interruptions externes

Objectif

L’objectif de cette partie est d’implémenter le contrôleur d'interruption vectorisé afin de gérer les interruptions.

Je n’ai pas fini cette partie, le code commencé est disponible dans le fichier ProjectProcesseurMonocycle_Quartus/src/VIC.vhd