République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
ECOLE NATIONALE POLYTECHNIQUE

DÉPARTEMENT D'ÉLECTRONIQUE

Mémoire de Master en Electronique

Thème :

Description et réalisation d'une Machine MIC-1

Amine OUDJEHANE

Sous la direction de : M. Rabah SADOUN

Présenté et soutenu publiquement le (23/06/2016)

Composition du Jury :

Président	M. Mohamed TRABELSI.	PR	ENP
Promoteur	M. Rabah SADOUN.	PR	ENP
Examinateur	M. Mohamed TAGHI.	PR	ENP

Promotion : Juin 2016

REMERCIEMENTS

Aucun travail ne s'accomplit dans la solitude

J'ai tout au long de mes années d'études eu le soutien de plusieurs personnes et je tiens à leur exprimer ma gratitude.

Mes vifs remerciements s'adressent en premier lieu à Monsieur SADOUN pour ses précieux conseils, son soutien, sa disponibilité ainsi que ses qualités relationnelles indéniables.

Je tiens aussi à remercier tous les enseignants du Département Electronique à l'Ecole Nationale Polytechnique auxquels je dois ma formation d'ingénieur.

Enfin, je remercie les membres du jury qui me font l'honneur d'évaluer mon travail.

- Monsieur TRABELSI (Président du jury).
- Monsieur TAGHI (Examinateur).

ملخص

الهدف من هذا المشروع هو تحقيق آلة ميك1 بالإتش دي إل. الآلة تتكون من عدة اجزاء، تم تعريف كل جزء على حدة ثم قمنا بجمع كل القطع و تشكيل تطبيق أعلى مستوى في واحد وصف واحد و ذلك باستعمال أداة كوارتس

الكلمات الدالة

ميك1، معالج، كوارتس، اجزاء شغالة.

Abstract

The main goal of this project is to describe the MIC1 machine on VHDL. The machine consists of several blocks, each was described individually and then assembled in a top level VHDL description VHDL by instantiating all the blocks in one description, and simulating it with Quartus by Altera.

Key Words: MIC-1, Processor, Quartus, Functional blocs.

Résumé

Le but de ce projet est de réaliser la machine MIC1 en VHDL, la machine se compose de plusieurs blocs, chacun a été réalisé individuellement et puis on a assemblé tous les blocs en une description top level en VHDL en instanciant tous les blocs en une seule description, et en les simulant avec l'outil Quartus.

Mots clés : MIC-1, Processeur, Quartus, Blocs foctionnels.

Table des matières

Liste des figures

Liste des abréviations

CISC	Complex Instruction Set Computer
CPU	Dynamic Random Access Memory
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
MAC	Multiply Accumulate
MCO	Micro Compteur Ordinal
MMUX	Micro Multiplexeur
RAD	Registre d'ADresse
RAM	Random Access Memory
RDO	Registre de DOnnées
RISC	Reduced Instruction Set Computer
RMI	Registre Micro-Instruction
SOC	System On Chip
UMA	Uniform Memory Access
UAL	Unité Arithmétique et Logique
VLIW	Very Long Instruction Word

INTRODUCTION GENERALE

La machine MIC-1 est une architecture de processeur inventé par Andrew S. Tanenbaum pour être utilisée comme un exemple simple mais complet dans son livre d'enseignement Organisation informatique Structurée.

Elle se compose d'une unité de commande très simple qui fonctionne à base de microcodes à partir d'une bibliothèque.

Le présent travail consiste en une mise en œuvre de cette architecture. Pour ce faire, le document est organisé comme suit :

Le premier chapitre concerne la modélisation des architectures distribuées hétérogènes. Après avoir réalisé un état de l'art des différents modèles d'architecture, nous présenterons une classification des différents processeurs (du processeur CISC aux systèmes sur puces SOC). On abordera dans une autre section la classification des machines ou nous utiliserons deux notions pour y parvenir. La première identifie le type de parallélisme offert par la machine, il est basé sur la relation entre le nombre de séquenceurs et le nombre d'unités de traitements arithmétiques et logiques. Le second critère repose sur l'organisation de la mémoire dans la machine.

Le second chapitre est consacré à la description de l'architecture de la machine MIC-1. Chaque bloc est décrit séparément et validé par simulation fonctionnelle.

Le troisième chapitre présente la mise en œuvre de notre machine MIC-1 en interconnectant les différents blocs déjà décrits dans le chapitre II ainsi que la description Top-level.

CHAPITRE I : MODELE D'ARCHITECTURE

I.1 Etat de l'art

En 1946, Von Neumann a posé les bases d'un modèle d'architecture encore largement utilisé aujourd'hui [16].

Ce modèle repose sur la coopération de cinq unités (Cf. figure I-1). L'unité mémoire contient des instructions et des données. L'unité de traitement transforme (effectue des calculs) les données stockées en mémoire. Les unités d'entrée et de sortie permettent de transférer des données entre la mémoire et l'environnement.

L'unité de commande contrôle l'ensemble de ces unités, elle repose sur un séquenceur d'instructions qui lit ses instructions dans la mémoire et les applique à l'unité de traitement. L'ensemble unité de commande-unité de traitement est souvent appelé processeur ou CPU (Central Processing Unit) [16].

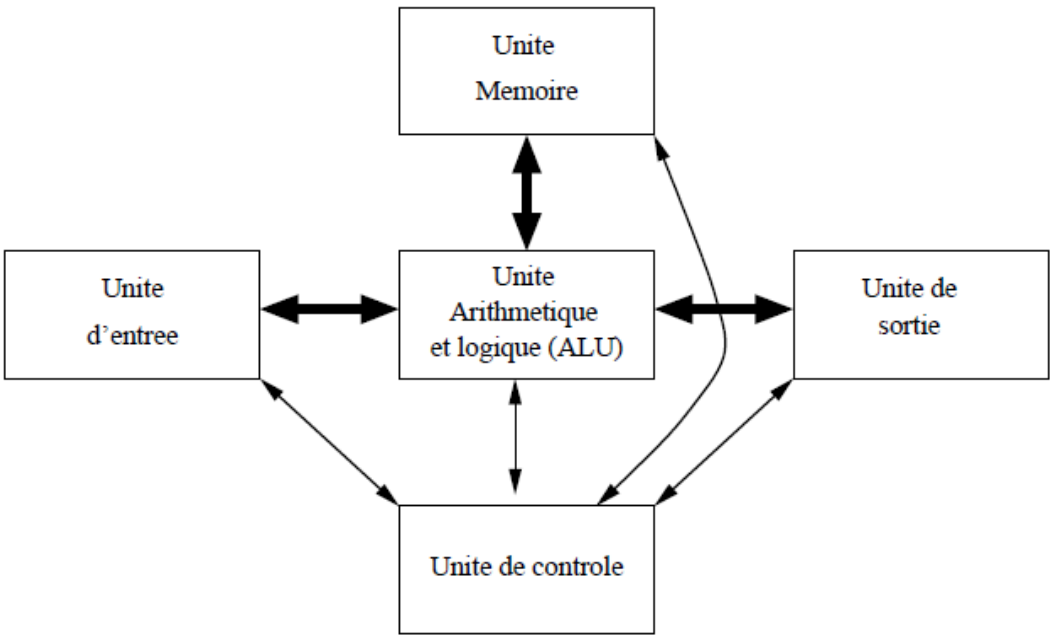


Figure I-1 : Modèle de Von Neumann (les flèches en gras représentent les chemins de données, les flèches fines les chemins de contrôle) [16]

I.2 Classification des processeurs

Bien que toujours basés sur le principe de Von Neumann, les processeurs ont bénéficié de nombreuses modifications architecturales afin d'améliorer leur puissance de calcul. Il est ainsi possible de les classer selon leur architecture interne [1].

I.2.1 Processeurs CISC

L'unité de traitement d'un processeur était initialement capable de réaliser un petit nombre d'opérations arithmétiques et logiques (ALU). Pour accroitre les performances des processeurs, les concepteurs ont modifié l'unité de contrôle et l'unité de traitement de façon à pouvoir supporter des opérations et des modes d'adressage (calcul d'adresses mémoire à partir de valeurs de registres) de plus en plus complexes, d'où le nom de **C**omplex **I**nstruction **S**et **C**omputer. Le séquenceur d'instructions ne pouvant plus être uniquement câblé, étant donné sa complexité [48], a rapidement nécessité un fonctionnement basé sur des microprogrammes. Basé sur ce fonctionnement complexe, les instructions complexes nécessitent en moyenne une dizaine de cycles d'horloge.

I.2.2 Processeurs RISC

Dans les années 1980 [2, 3], il a été montré que les processeurs CISC passaient 80 pour cent de leur temps à exécuter 20 pour cent du jeu d'instructions offert et que la réduction [3] du nombre d'instructions (**R**educed **I**nstruction **S**et **C**omputer [4]) accompagnée d'une simplification de leur format (taille fixe des instructions, emplacement fixe des "codes opérations" et des opérandes) permettait de simplifier leur encodage et de le câbler complétement (sans utilisation de microprogramme). Ainsi, en apportant simplicité et régularité il a été possible de pipeliner le décodage et l'exécution des instructions, permettant finalement de réduire leur durée d'exécution.

Souvent couplés à une architecture HARVARD (ref harvard) (dans laquelle les instructions et les données sont lues dans des mémoires caches ou des mémoires physiques différentes, connectées au processeur par des bus séparés) les processeurs RISC permettent d'exécuter une instruction par cycle d'horloge.

L'utilisation efficace de ces processeurs repose en grande partie sur la capacité de leurs compilateurs à bien gérer les registres internes. Le travail qui était effectué par les microprogrammes des processeurs CISC est effectué en grande partie par les compilateurs des processeurs RISC.

I.2.3 Processeurs VLIW

Ces processeurs sont basés sur un séquenceur d'instructions et plusieurs unités de traitement qui peuvent travailler en parallèle pour effectuer des opérations très différentes sur des données différentes. Pour cela, chaque instruction du processeur contient une instruction spécifique pour chaque unité de traitement. Les instructions de ces processeurs sont donc stockées dans des mots de grande longueur, d'où leur nom de **V**ery **L**ong **I**nstruction **W**ord [5]. Ces processeurs offrent un parallélisme de calcul qualifié de parallélisme à grain fin car il est observé au niveau des instructions du processeur. Ce parallélisme est si fin que c'est essentiellement sur les compilateurs que repose l'exploitation des unités parallèles [6]. Ces processeurs sont rarement classés parmi les machines SIMD [7], car bien que n'ayant qu'un seul séquenceur d'instructions, les opérations effectuées par les unités de traitement ne sont pas nécessairement identiques comme c'est le cas pour les machines SIMD dont la structure est régulière.

I.2.4 Processeurs de Traitement du Signal - DSP

Dans le but d'accélérer les calculs spécifiques au domaine du traitement du signal (filtrage, fft), un autre type de processeur a été créé. Il s'agit des processeurs de traitement du signal (DSP - Digital Signal Processor), ils possèdent pour la plupart [8, 1] :

- Des unités capable(s) d'effectuer en parallèle la lecture de deux opérandes, une multiplication, et une opération arithmétique (cela permet d'accélérer par exemple la somme de produits - MAC Multiply ACcumulate - utilisée dans le produit de matrices pour les calculs de filtres).
- Des modes d'adressage spécialisés (par exemple pré et post modification des pointeurs d'adresse, adressage circulaire, adressage bit-reverse, pour accélérer le calcul des FFT).
- Des instructions de contrôles particulières (grâce à une architecture adaptée) qui permettent l'exécution de boucles sans surcoût temporel (i.e. sans gaspiller de cycle à incrémenter, puis tester le compteur de boucles et à effectuer le saut conditionnel arrière vers le début de la boucle).
- l'architecture des DSP est souvent faite de façon à permettre plusieurs accès simultanés à la mémoire pendant l'exécution d'une instruction arithmétique. Ainsi ils ont souvent de la mémoire interne pour les données et une mémoire cache instruction pour faire de "l'optimisation 1 cycle" : lecture 2 opérandes, calcul, écriture dans le même cycle.

Ces processeurs offrent donc un parallélisme à grain fin qui ne peut être exploité que par la connaissance précise et spécifique de l'architecture du processeur. Comme pour les processeurs

RISC, les performances de ces processeurs dépendent fortement de la capacité de leurs compilateurs à bien placer les données [8].

I.2.5 Microcontrôleurs

Basés sur les modèles de processeurs CISC ou RISC, ils sont conçus pour fonctionner avec un minimum de composants extérieurs. Ainsi leur mémoire est souvent embarquée sur le chip ainsi que de nombreuses unités d'entrée-sortie (CAN [9], UART RS232, I2C [10], pour échanger des données avec des périphériques (claviers, afficheurs) et processeurs, convertisseurs analogique/numérique ou numérique/analogique pour appréhender le monde physique extérieur).

En contrepartie leur puissance de calcul est souvent relativement faible étant donné la surface de silicium restante et les contraintes d'embarquabilité et de consommation minimale (puisque la fréquence de fonctionnement est souvent diminuée). Dans le contexte des systèmes embarqués, ils sont destinés à être implantés géographiquement près des capteurs et actionneurs, où ils réalisent alors souvent des calculs simples liés à l'acquisition ou à la commande.

I.2.6 Les Systèmes "On Chip" (SOC)

Avec l'´émergence des Socs (Système sur un même morceau de silicium, "System On a Chip"), qui incluent diverses unités (processeurs, mémoires, systèmes d'entrée-sortie), la définition d'un processeur devient plus floue : par exemple, le TMS320C80 [11] de Texas Instrument est qualifié de "processeur" alors qu'il inclut quatre processeurs DSPs, un processeur RISC, cinq bancs mémoire, un crossbar et un DMA sur un seul chip.

I.3 Classification des machines

A partir d'un seul processeur il est possible de construire des machines relativement simples, de puissance égale à celle du processeur, accroître la puissance de calcul de la machine équivaut à accroître celle du processeur. Une autre technique que celle de l'utilisation de processeur unique, consiste à construire des machines parallèles par connexion de processeurs à l'aide de mémoires et de média de communication.

Dans ces machines, les unités de traitement de chaque processeur fonctionnent en parallèle de façon à résoudre un problème commun. Pour cela ils doivent coopérer, c'est à dire communiquer pour s'échanger des données mais aussi se synchroniser. Il est possible de connecter ensemble de telles machines pour en créer de plus puissantes encore. Ainsi, nous

constatons que toutes les machines qu'il est possible de construire, sont basées sur deux notions génériques : le traitement (basé sur le séquencement d'instructions réalisant des opérations arithmétiques et logiques) et la mémoire. Nous allons utiliser ces deux notions pour classer les machines. La première identifie le type de parallélisme offert par la machine, il est basé sur la relation entre le nombre de séquenceurs et le nombre d'unités de traitements arithmétiques et logiques.
Le second critère repose sur l'organisation de la mémoire dans la machine.

I.3.1 Séquenceurs

Dans la classification de Flynn [12], qui a maintenant 30 ans, les machines sont organisées en quatre groupes correspondant au parallélisme d'instruction (un ou plusieurs séquenceurs qui permettent un ou plusieurs flux d'instructions) et de données (une ou plusieurs unités de traitement qui permettent un ou plusieurs flux de données).
Les machines SISD sont basées sur un unique séquenceur d'instructions et une unique unité de traitement.
Ce sont les machines les plus simples, elles n'offrent aucun parallélisme, leur architecture est comparable à celle de Von Neumann. Les machines SIMD n'ont toujours qu'un séquenceur d'instructions mais renferment plusieurs unités de traitement fonctionnant en parallèle de manière synchrone, multipliant d'autant leur capacité de calcul. Les machines MISD sont des machines possédant plusieurs séquenceurs d'instructions mais une seule unité de traitement (il n'existe pas encore de machine commerciale de ce type [7] particulier, les machines pipeline sont parfois classées dans cette catégorie). La dernière catégorie, très en vogue ces dernières années grâce aux progrès technologiques, correspond aux machines MIMD. Elles possèdent plusieurs séquenceurs d'instructions indépendants et plusieurs unités de traitements et fonctionnent de ce fait de manière principalement asynchrone. Les MIMD sont homogènes si les séquenceurs et les unités de traitements qui les composent sont identiques, et hétérogènes s'ils sont différents. Il existe une très grande diversité de machines MIMD aux caractéristiques architecturales très différentes (selon par exemple, l'organisation de leurs mémoires ou de leurs modes de communication), montrant ainsi la limite de la classification de Flynn.

I.3.2 Mémoire(s)

L'organisation de la mémoire et les méthodes d'accès à la mémoire par les processeurs sont la base d'une seconde classification [13] qui organise les machines MIMD en trois catégories : UMA, NUMA et NORMA.

La plus connue reste UMA (Uniform Memory Access). Dans cette dernière tous les processeurs ont accès à toute la mémoire de la machine de façon uniforme. C'est le cas des machines à mémoire centralisée et partagée (Centralized Shared Memory) entre tous les processeurs au moyen d'un bus. Ce bus devient rapidement le goulet d'´étranglement lorsque les requêtes des processeurs à la mémoire sont nombreuses et simultanées, il engendre donc un phénomène de contentions des processeurs qui est proportionnel à leur nombre dans l'architecture. Ce nombre est donc généralement limité, selon la technologie employée, à moins d'une trentaine de processeurs [14]. Pour tenter de résoudre ce problème, deux types de variante architecturale ont été introduites. La première repose sur l'ajout de mémoire cache au niveau de chaque processeur. La seconde consiste à diviser la mémoire en bancs physiques distincts. Chaque banc peut être mis en relation avec n'importe quel processeur au moyen d'un réseau d'interconnexion dynamique (capable de supporter de forts débits et des temps d'accès très courts : bus hiérarchisés, multiétage, crossbar). Bien que physiquement découpée, la mémoire reste logiquement continue et accessible de façon uniforme par tous les processeurs qui peuvent aussi être connectés à une mémoire cache pour diminuer d'avantage la latence d'accès à la mémoire.

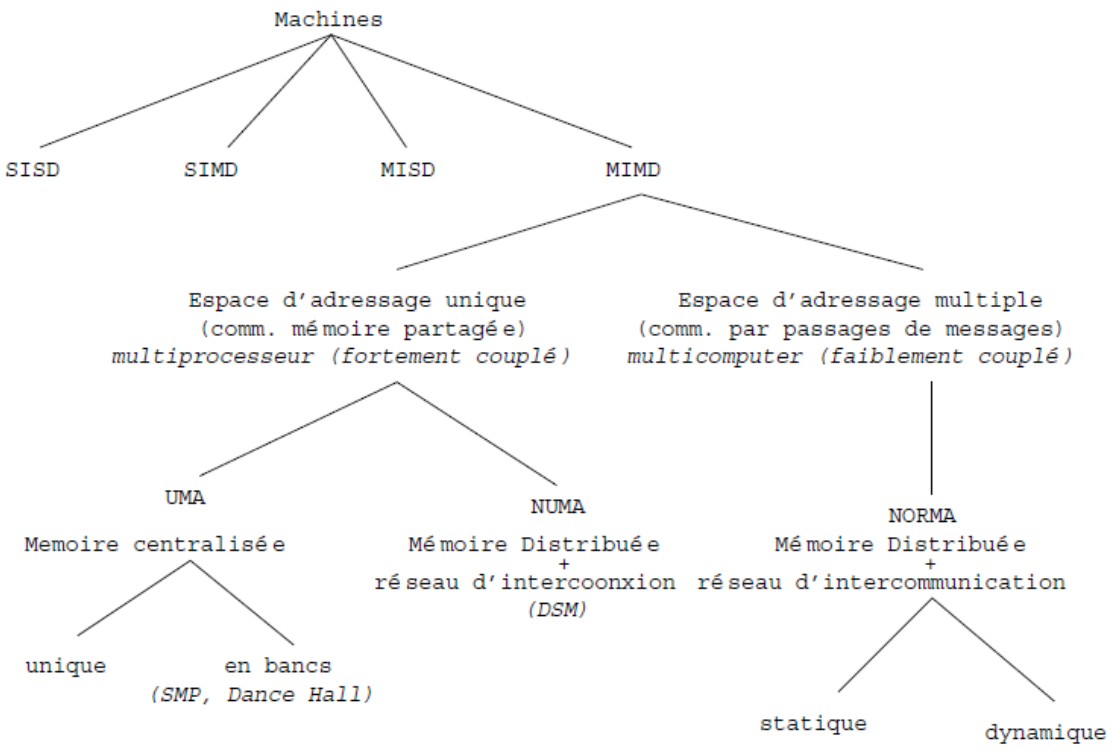


Figure I-2 : Classification des différents types de machines

CHAPITRE II :
DESCRIPTION DE LA MACHINE MIC-1 ET LES DIFFERENTS BLOCS

Dans ce chapitre, on abordera le rôle de chaque bloc intervenant dans la machine MIC-1, sa description matérielle ainsi qu'une simulation fonctionnelle.

II.1 Description de l'architecture MIC-1

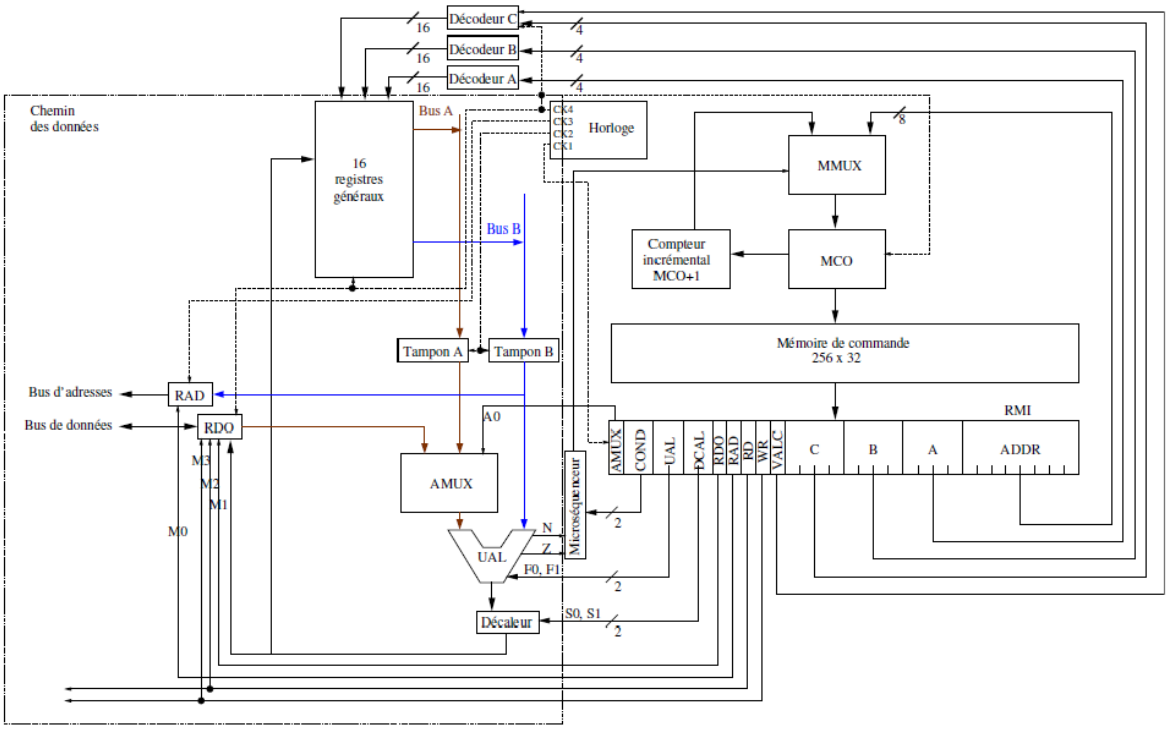


Figure II-1 : Architecture d'une machine MIC-1

II.1.1 Les cycles fonctionnels

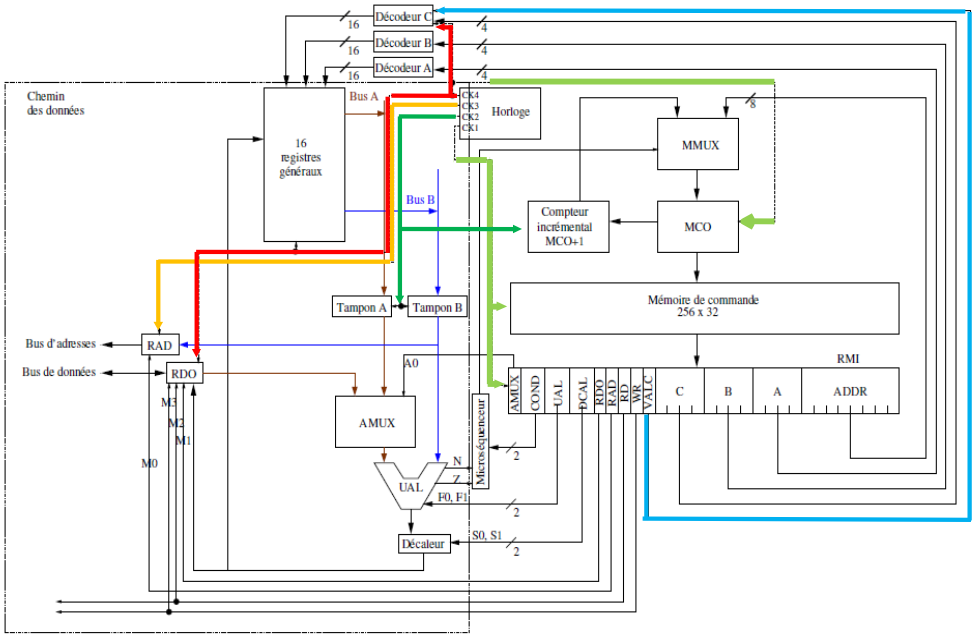


Figure II-2 : Architecture d'une machine MIC-1 avec la mise en évidence des 4 sous cycles

L'exécution d'une micro-instruction est divisée en quatre étapes, correspondant chacune a un des quatre sous-cycles de l'horloge : CLK1 à CLK4

Sous cycle CLK1 (couleur rouge sur la figure)

La micro-instruction est transférée de la mémoire de commande vers le registre de micro-instruction (RMI). Ainsi, pour la mémoire de commande, le registre RMI joue le rôle d'un registre de donnée, tandis que le registre MCO (micro-compteur ordinal) joue le rôle d'un registre d'adresse.

Sous cycle CLK2 (couleur orange)

Le compteur incrémental ajoute 1 au contenu de MCO

Les valeurs des champs A et B sélectionnent, par l'intermédiaire des décodeurs, les registres généraux dont les contenus sont transférés sur les bus, puis sur les tampons A et B.

Sous cycle CLK3 (couleur vert foncé)

Les opérations sont exécutées par l'UAL et le décaleur, avec comme opérandes :

- Le tampon B

- La sortie du multiplexeur qui est soit le tampon A, soit le contenu du registre de donnée RDO.
- La donnée du tampon B est transférée dans le registre d'adresse RAD si champ RAD est a 1 .

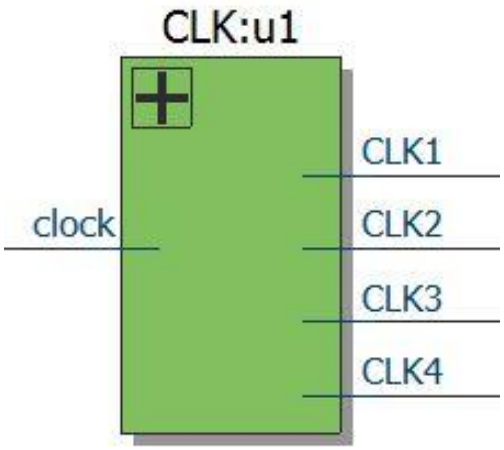
Sous cycle CLK4 (couleur vert clair)

La donnée du bus C en sortie du décaleur peut être déchargée dans un des registres généraux (désigné par le champ C), lorsque VALC vaut 1, et dans RDO, si le champ RDO vaut 1.
Le décodeur C est donc relié à la fois aux champs C et VALC de la micro-instruction et au quatrième signal d'horloge CK4.

II.2 Description des différents blocs

II.2.1 Clock

En entrée, on reçoit un seul signal d'horloge qui est déphasé pour créer 4 signaux d'horloges qu'on appelle sous cycles d'instructions (CLK1 => CLK 4). Chacun de ces sous cycles est réservé à une fonction bien précise :



Fetch : Recherche micro instruction
Read registers : Lecture de registres
Execute : UAL ou/et sélection mémoire
Write : Registre ou lecture/écriture mémoire

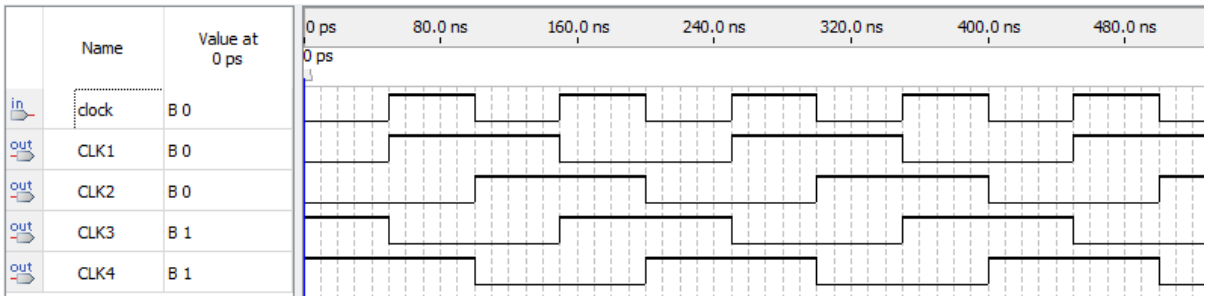


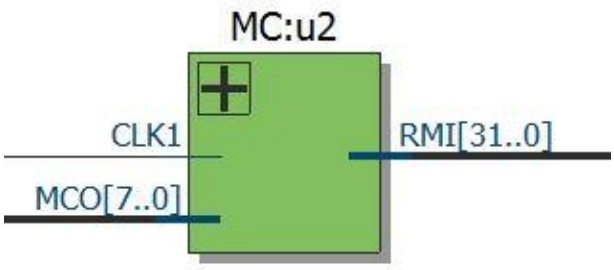
Figure II-3 : Simulation fonctionnelle de CLK1 à CLK4

Description matérielle

```vhdl
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

entity CLK is
    Port(clock : in std_logic;
         CLK1, CLK2, CLK3, CLK4: out std_logic);

end CLK;

architecture Behav of CLK is
    signal clockInv : std_logic;

    signal sig1 : std_logic := '0';
    signal sig2 : std_logic := '0';
    signal sig3 : std_logic := '1';
    signal sig4 : std_logic := '1';
begin
    clockInv <= not clock;

    process(clock)
    begin

        if clock'event and clock='1' then
            sig1 <= not sig1;
        else
            sig1 <= sig1;
        end if;
    end process;
    CLK1 <= sig1;

    process(clock)
    begin

        if clock'event and clock = '0' then
            sig2 <= not sig2;
        else
            sig2 <= sig2;
        end if;
    end process;
    CLK2 <= sig2;

    process(clockInv)
    begin

        if clockInv'event and clockInv = '0' then
            sig3 <= not sig3;
        else
            sig3 <= sig3;
        end if;
    end process;
    CLK3 <= sig3;

    process(clockInv)
    begin

        if clockInv'event and clockInv='1' then
            sig4 <= not sig4;
        else
            sig4 <= sig4;
        end if;

    end process;
    CLK4 <= sig4;

end Behav;
```

II.2.2 Mémoire de Commande

Mémoire micro programmée, contient le micro programme sur 32bits. Permet d'associer les instructions de la RAM en des instructions microprogrammes.

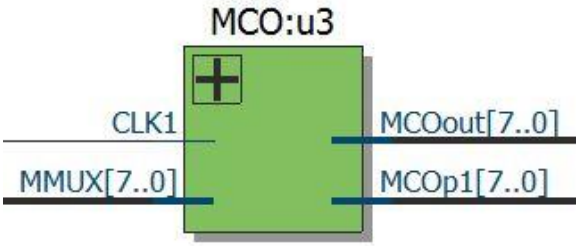


Description matérielle

```
1   library IEEE;
2   use ieee.std_logic_1164.ALL;
3   Use ieee.std_logic_unsigned.ALL;
4   use ieee.numeric_std.ALL;
5
6
7   entity MCOp1 is
8       port(CLK2 : in std_logic;
9            MCO  : in std_logic_vector(7 downto 0);
10           MMUX :  out std_logic_vector(7 downto 0)
11  );
12  end MCOp1;
13
14  architecture Behav of MCOp1 is
15
16      begin
17      process(CLK2)
18      begin
19          if (CLK2'event and CLK2='1') then
20              MMUX <= MCO + '1';
21          end if;
22      end process;
23      end Behav;
24
```

II.2.3 MCO

Micro compteur ordinal, Il pointe à la mémoire de l'instruction à charger dans le RMI.



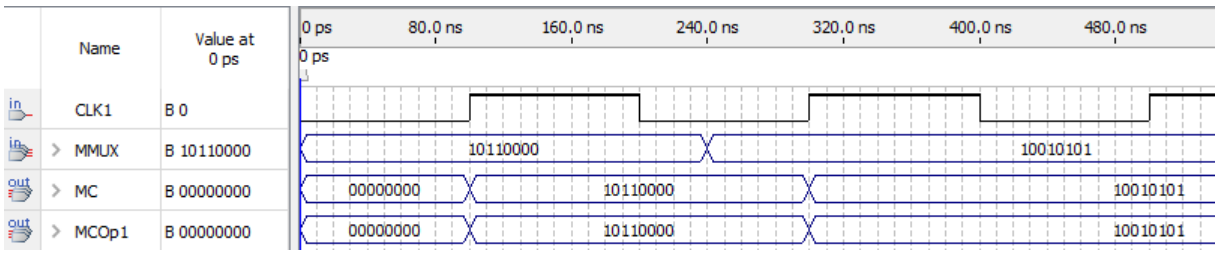
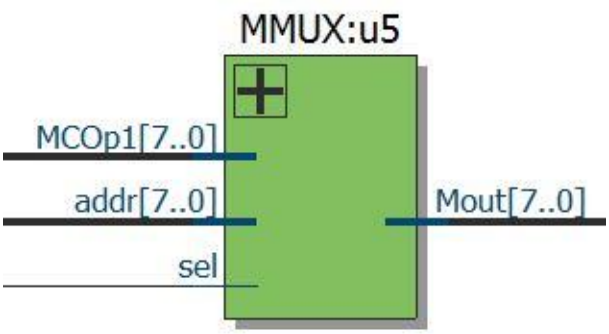


Figure II-4 : Simulation fonctionnelle de MCO

```
1  library IEEE;
2  use ieee.std_logic_1164.ALL;
3  Use ieee.std_logic_unsigned.ALL;
4  use ieee.numeric_std.ALL;
5
6
7  entity MCO is
8      port( CLK1 : in std_logic;
9      MMUX : in std_logic_vector(7 downto 0);
10     MCOout, MCOp1 : out std_logic_vector(7 downto 0)
11     );
12     end MCO;
13
14 architecture Behav of MCO is
15
16     begin
17 --MCOout=> "00000000";
18     process(CLK1)
19     begin
20         if (CLK1'event and CLK1='1') then
21             MCOp1 <= MMUX;
22             MCOout <= MMUX;
23         end if;
24     end process;
25     end Behav;
26
```

II.2.4 MMUX

Micro Multiplexeur, il sélectionne d'adresse de la prochaine instruction entre une incrémentation du MCO ou directement du RMI selon un signal de commande du séquenceur.



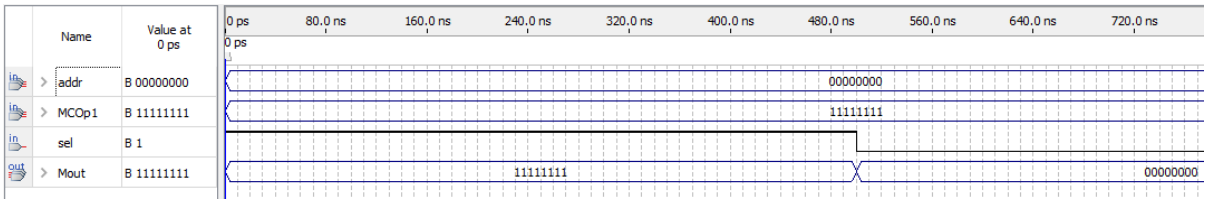
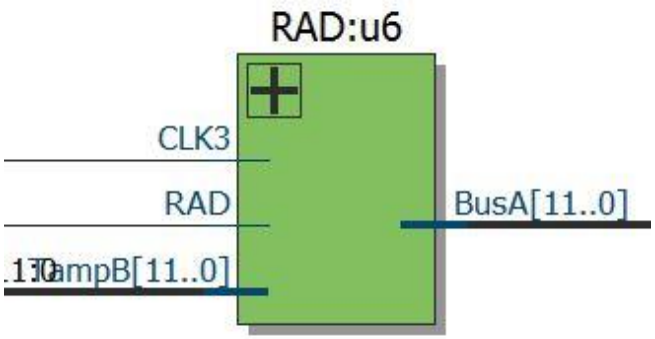


Figure II-5 : Simulation fonctionnelle de MMUX

```
1  library IEEE;
2  use ieee.std_logic_1164.ALL;
3  Use ieee.std_logic_unsigned.ALL;
4  use ieee.numeric_std.ALL;
5
6
7  entity MMUX is port(
8       addr, MCOp1 : in std_logic_vector(7 downto 0);
9       sel : in std_logic;
10      Mout :  out std_logic_vector(7 downto 0)
11      );
12      end MMUX;
13
14 architecture Behav of MMUX is
15
16      begin
17
18
19      Mout <= addr when sel = '0'
20      else MCOp1;
21
22      end Behav;
```

II.2.5 RAD

Registre d'adresse, pointe sur une adresse dans la RAM ou contient l'adresse à la transmettre.



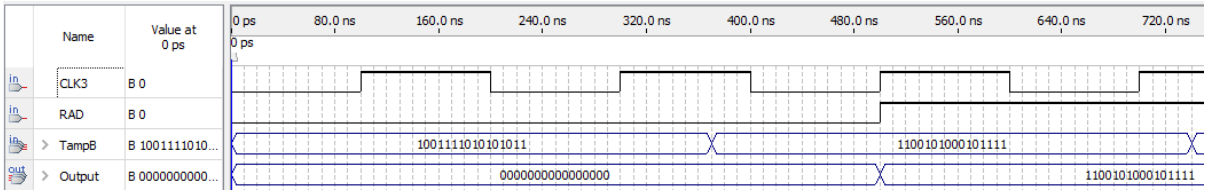


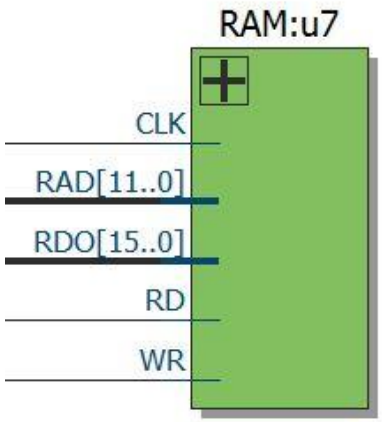
Figure II-6 : Simulation fonctionnelle de RAD

```vhdl
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;


entity RAD is
    port(
    CLK3, RAD : in std_logic;
    TampB : in std_logic_vector(11 downto 0);
    BusA : out std_logic_vector(11 downto 0)
    );
end RAD;

architecture Behav of RAD is

    signal Sig : std_logic_vector(11 downto 0);
    begin
    Sig<= TampB;

    process(CLK3, RAD)
    begin
        if (CLK3 = '1' and CLK3'event) then
            if(RAD='1')then
                BusA <= Sig;
            end if;
        end if;
    end process;

end Behav;
```

II.2.6 RAM :

Elle est décrite comme étant une matrice (array) de 4096 vecteurs (adresses) de 16 bits.

Les données de RDO sont soit stockées dans l'adresse indiquée dans RAD ou l'inverse selon l'état des signaux WR et RD.

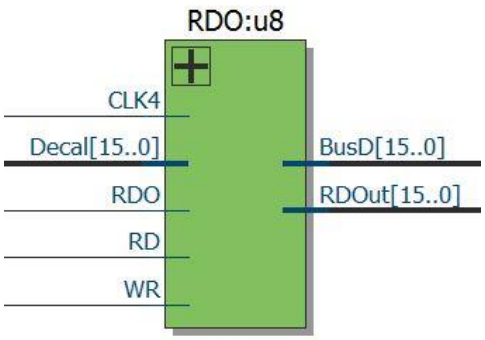


Description matérielle

```
1  library IEEE;
2  C:\Users\Amine\Downloads\VHDL\MC.vhd
3  Use ieee.std_logic_unsigned.ALL;
4  use ieee.numeric_std.ALL;
5
6  entity RAM is
7      port
8      (CLK,WR,RD : in std_logic;
9       RAD : in std_logic_vector(11 downto 0);
10      RDO : inout std_logic_vector(15 downto 0)
11     );
12 end RAM;
13
14 architecture Behav of RAM is
15     type RAM  is array(4095 downto 0) of std_logic_vector(15 downto 0); --
16     signal Sig :RAM:=  ( 40 => "0000000000000100", others => (others=>'0'))
17 begin
18
19 --Sig(40)<= x"AAAA";
20
21     process(CLK)
22     begin
23         if(CLK'event and CLK='1') then
24             if(RD='1' and WR='0') then
25                 Sig(to_integer(unsigned(RAD)))<= RDO;
26             else if(WR='1' and RD='0') then
27                 RDO <= Sig(to_integer(unsigned(RAD)));
28             else
29                 RDO <= "ZZZZZZZZZZZZZZZZ";
30             end if;
31             end if;
32         end if;
33     end process;
34
35 end Behav;
```

II.2.7 RDO :

Registre de données, contient les données à échanger entre l'UAL et la RAM, il est commandé par les signaux WR et RD, on a utilisé un signal INOUT pour l'échange des données avec la RAM.



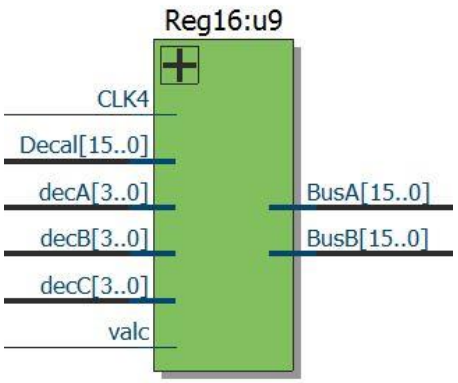
Description matérielle

```
1  library IEEE;
2  use ieee.std_logic_1164.ALL;
3  Use ieee.std_logic_unsigned.ALL;
4  use ieee.numeric_std.ALL;
5
6
7  entity RDO is port(
8
9      CLK4, RDO, RD, WR : in std_logic;
10     Decal : in std_logic_vector(15 downto 0);
11     BusD : inout std_logic_vector(15 downto 0):="0000000000000000";
12     RDOut : out std_logic_vector(15 downto 0)
13     );
14 end RDO;
15
16 architecture Behav of RDO is
17 begin
18
19 process(CLK4)
20 begin
21     if (CLK4 = '1' and CLK4'event) then
22         if(RDO ='1') then
23             if(RD = '1')then
24                 RDOut<=BusD;
25             else if(WR = '1') then
26                 BusD<=Decal;
27             end if;
28             end if;
29         end if;
30     end if;
31 end process;
32 end Behav;
```

II.2.8 Les 16 registres

Boite de 16 registres à usage spécifique décrite de la même façon que la RAM.

- Registres A, B, C, D, E, F à usage général.
- Registre CO contient la valeur du Compteur Ordinal.
- Registre AC (Accumulateur).
- Registre PP contient le Pointeur de Pile.
- Registre RI (Registre d'Instruction).
- Registre RIT contient une copie de RI.
- Registres 0, +1, et -1 contiennent les valeurs constantes.



- Registre AMASQ est un masque d'adresse (contient 0x0fff).
- Registre PMASQ est un masque de pile (contient 0x00ff).

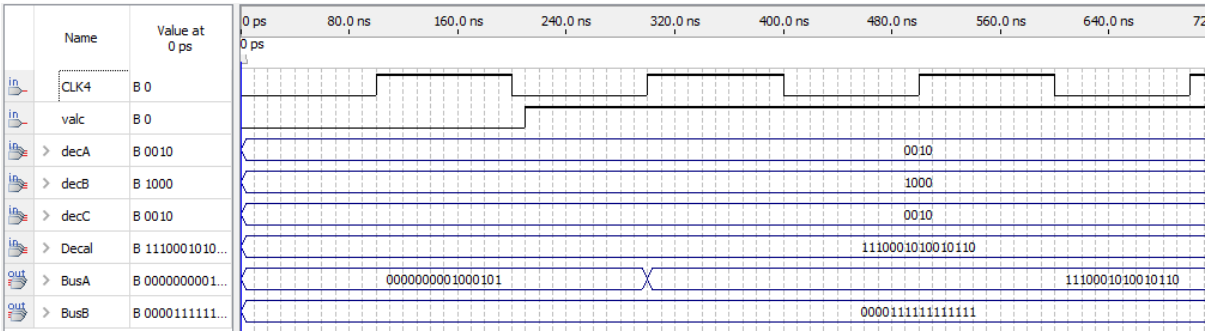


Figure II-7 : Simulation fonctionnelle des différents registres

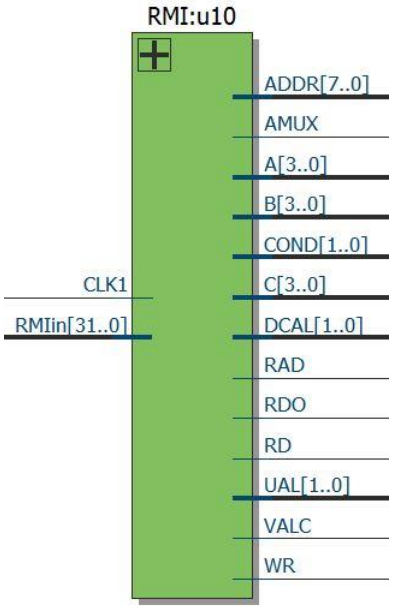
Description matérielle

```
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

entity Reg16 is
    port
    (valc, CLK4      : in std_logic;
     Decal           : in std_logic_vector(15 downto 0);
     decA,decB,decC  : in std_logic_vector(3 downto 0);
     BusA,BusB       : out std_logic_vector(15 downto 0)
    );
end Reg16;

architecture Behav of Reg16 is
    type REG  is array(15 downto 0) of std_logic_vector(15 downto 0);
    signal Sig : REG:=(x"0000",x"0000",x"0000",x"0000",x"0000",x"0000'
begin

    process(CLK4)
    begin
    if(CLK4'event and CLK4='1') then
        if(valc = '1') then
            Sig(to_integer(unsigned(decC))) <= Decal;
        end if;
    end if;
    end process;

BusA <= Sig(to_integer(unsigned(DecA)));
BusB <= Sig(to_integer(unsigned(DecB)));

```

II.2.9 RMI

Registre micro instruction, il charge l'instruction en cours d'exécution et génère les signaux de commande des différents blocs.

Il reçoit l'adresse de la mémoire de commande et l'interprète en plusieurs signaux de commande responsable à l'exécution de l'instruction actuelle par les autres blocs.



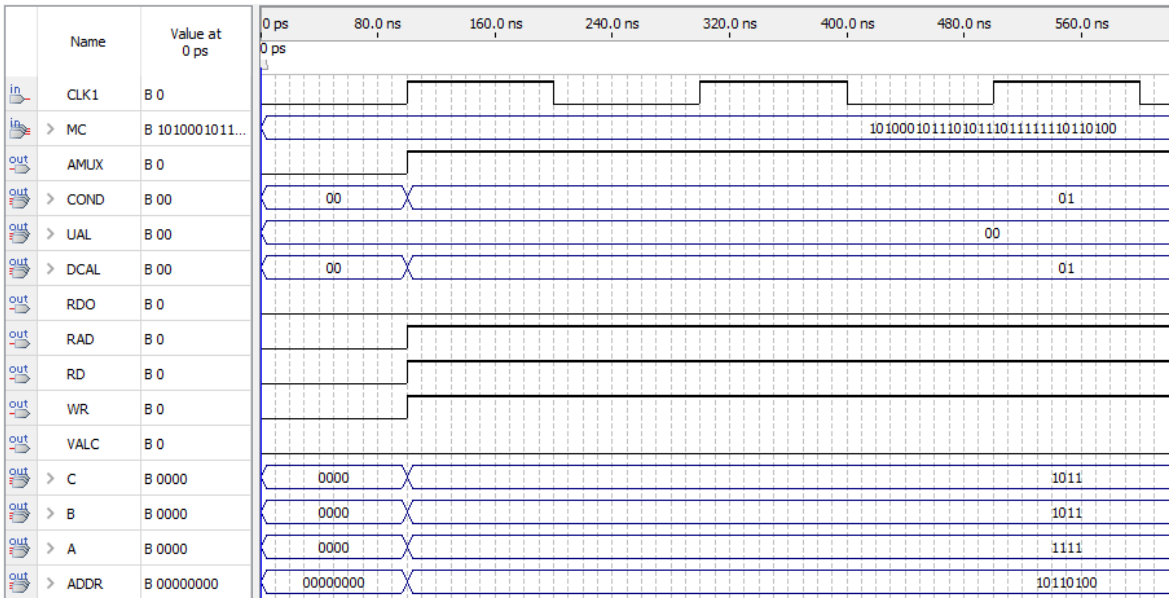
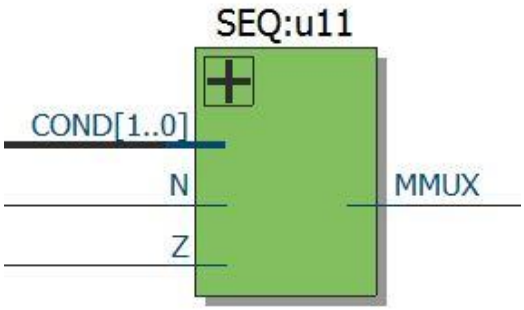


Figure II-8 : Simulation fonctionnelle de la RMI

```
1  library IEEE;
2  use ieee.std_logic_1164.ALL;
3  Use ieee.std_logic_unsigned.ALL;
4  use ieee.numeric_std.ALL;
5
6  entity RMI is
7          port (  CLK1 : in std_logic;
8                  RMIin  : in std_logic_vector(31 downto 0);
9                  AMUX,RDO, RAD, RD, WR, VALC : out std_logic;
10                 COND, UAL, DCAL : out std_logic_vector(1 downto 0);
11                 A, B, C     : out std_logic_vector(3 downto 0);
12                 ADDR : out std_logic_vector(7 downto 0)
13         );
14         end RMI;
15
16 architecture Behav of RMI is
17
18     signal MCsig : std_logic_vector(31 downto 0);
19     begin
20     MCsig <= RMIin;
21
22         process (CLK1)
23         begin
24             if (CLK1'event and CLK1='1') then
25             AMUX    <= MCsig(31);
26             COND    <= MCsig(30 downto 29);
27             UAL     <= MCsig(28 downto 27);
28             DCAL    <= MCsig(26 downto 25);
29             RDO     <= MCsig(24);
30             RAD     <= MCsig(23);
31             RD      <= MCsig(22);
32             WR      <= MCsig(21);
33             VALC    <= MCsig(20);
34             C       <= MCsig(19 downto 16);
35             B       <= MCsig(15 downto 12);
36             A       <= MCsig(11 downto 8);
37             ADDR    <= MCsig(7 downto 0);
38
39             end if;
40         end process;
41     end Behav;
```

II.2.10 Séquenceur

Détermine la fonction pour chaque sous cycle et enclenche un jump vers la prochaine micro instruction selon les signaux de commandes (Cond et les flags N et Z) pour chaque sous cycle.



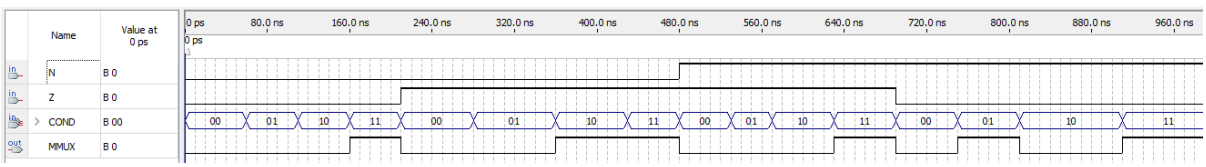


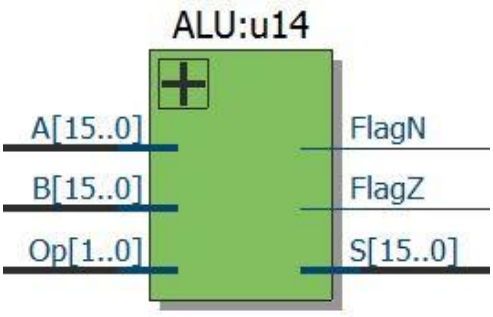
Figure II-9 : Simulation fonctionnelle du séquenceur

```
1  library IEEE;
2  use ieee.std_logic_1164.ALL;
3  Use ieee.std_logic_unsigned.ALL;
4  use ieee.numeric_std.ALL;
5
6
7  entity SEQ is
8      port(
9      N : in std_logic;
10     Z : in std_logic;
11     COND : in std_logic_vector(1 downto 0);
12     MMUX : out std_logic
13     );
14     end SEQ;
15
16 architecture Behav of SEQ is
17     signal Sig: std_logic_vector(3 downto 0);
18
19     begin
20     Sig<= COND & Z & N;
21
22     with Sig select
23     MMUX <= '1' when "0101",
24             '1' when "1010",
25             '1' when "1100",
26             '1' when "1101",
27             '1' when "1110",
28             '1' when "1111",
29             '0' when others;
30
31
32     end Behav;
```

II.2.11 ALU :

Unité arithmétique et logique avec 4 opérations possibles (sur des mots de 16 bits). Une opération sur les entrées A (en sortie de AMUX) et B (provenant du tampon B) est sélectionnée par Op :

0-> A + B, 1-> A ET B, 2-> A, 3-> NON A.



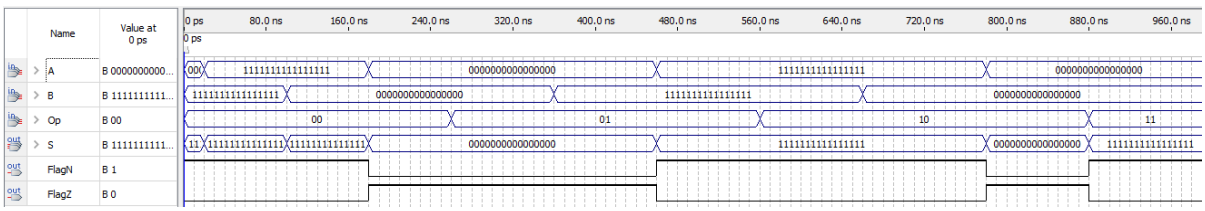
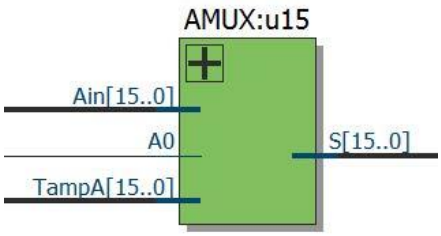


Figure II-10 : Simulation fonctionnelle de l'ALU

```
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;


entity ALU is
port(
        A,B : in std_logic_vector(15 downto 0);
        Op : in std_logic_vector(1 downto 0);
        S : out std_logic_vector(15 downto 0);
        FlagN : out std_logic;
        FlagZ : out std_logic
        );
end ALU;

architecture Behavioral of ALU is

signal RegA,RegB,RegS : std_logic_vector(15 downto 0) := (others => '0');
signal Flag : std_logic_vector(1 downto 0);

begin

S <= RegS;
RegA <= A;
RegB <= B;


with Op select
    RegS <=(RegA + RegB) When "00",
        (RegA and RegB) When "01",
        RegA When "10",
        not(RegA) When others ;

Process(RegS)
begin
    if (RegS(15)='1') then Flag <="01";
    elsif (RegS ="0000000000000000") then Flag <="10";
    else Flag<="00";
    end if;

end process;

FlagZ<=Flag(1);
FlagN<=Flag(0);

end Behavioral;
```

II.2.12 AMUX

C'est un multiplexeur qui sa principale fonction est de choisir entre le bus A ou le registre de données RDO.



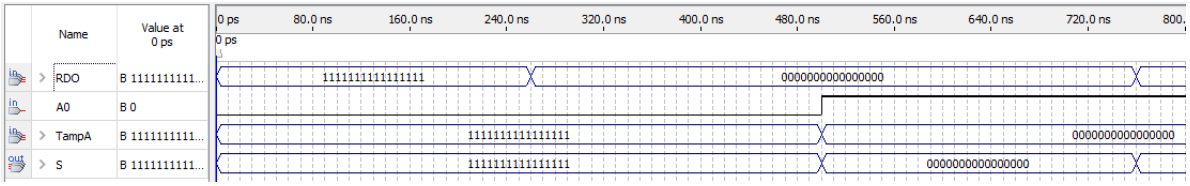


Figure II-11 : Simulation fonctionnelle de AMUX

Description matérielle

```
1   library IEEE;
2   use ieee.std_logic_1164.ALL;
3   Use ieee.std_logic_unsigned.ALL;
4   use ieee.numeric_std.ALL;
5
6
7   entity AMUX is port(
8       A0 : in std_logic;
9       TampA : in std_logic_vector(15 downto 0);
10      Ain : in std_logic_vector(15 downto 0);
11      S : out std_logic_vector(15 downto 0)
12      );
13      end AMUX;
14
15  architecture Behav of AMUX is
16
17      begin
18      S <= TampA when A0 = '0'
19      else Ain;
20
21      end Behav;
22
```

II.2.13 Décaleur :

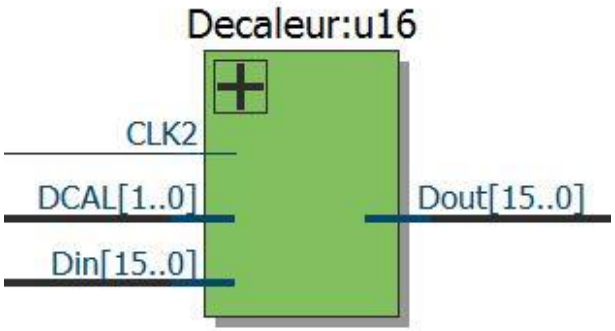
Commandé avec un signal DCAL dont la configuration indique le sens du décalage :

00 -> pas de décalage

10 -> décalage à droite

01 -> décalage à gauche

11 -> ne rien faire.



Description matérielle

```
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

entity Decaleur is

   port( CLK3: in std_logic;
         DCAL: in std_logic_vector(1 downto 0);
         Din: in std_logic_vector( 15 downto 0);
         Dout: out std_logic_vector(15 downto 0)
        );
end Decaleur;

architecture Behav of Decaleur is
    begin
    process
    variable SigOut: std_logic_vector(15 downto 0);
    begin
        wait until rising_edge (CLK3);
        SigOut := Din;
        if DCAL="01" then
            for i in 0 to 14 loop
                SigOut(i) := SigOut(i+1);
            end loop;
        SigOut(15) := '0';
        elsif DCAL="10" then
            for i in 0 to 14 loop
                SigOut(i+1) := SigOut(i);
            end loop;
        SigOut(0) := '0';
        elsif DCAL="00" then
        SigOut := Din;
        end if;
    Dout <= SigOut;
    end process;
end Behav;
```

II.2.14 Tampon

Description matérielle

```
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;


entity Tampon is port(
     Sinput : in std_logic_vector(15 downto 0);
     CLK2 : in std_logic;
     Soutput :   out std_logic_vector(15 downto 0)
     );
end Tampon;

architecture Behav of Tampon is

--signal Sigout: std_logic_vector(15 downto 0);
    begin


    process(CLK2)
    begin
        if (CLK2 = '1' and CLK2'event) then
        Soutput <= Sinput;
        end if;
    end process;

end Behav;
```

CHAPITRE III : ASSEMBLAGE DE LA MACHINE MIC-1

III.1 Introduction

Après l'assemblage des différents blocs, on obtient la machine MIC1 qui a pour but d'interpréter les instructions et les exécuter. Pour simplifier, on peut soit créer un programme (appelé macro-programme), le compiler et voir ce que donne son interprétation avec les micro instructions (appelés microprogramme), soit créer nous-mêmes ces propres micro-instructions et voir comment elles interprètent un macro-programme.

L'exécution est divisée en quatre étapes, correspondant chacune à un des quatre sous-cycles de l'horloge :

- Sous cycle CLK1 (Fetch) : Recherche micro Instruction.
- Sous cycle CLK2 (Read) : Lecture registres.
- Sous cycle CLK3 (Execute) : UAL ou sélection mémoire.
- Sous cycle CLK4 (Write) : Registre ou lecture/écriture.

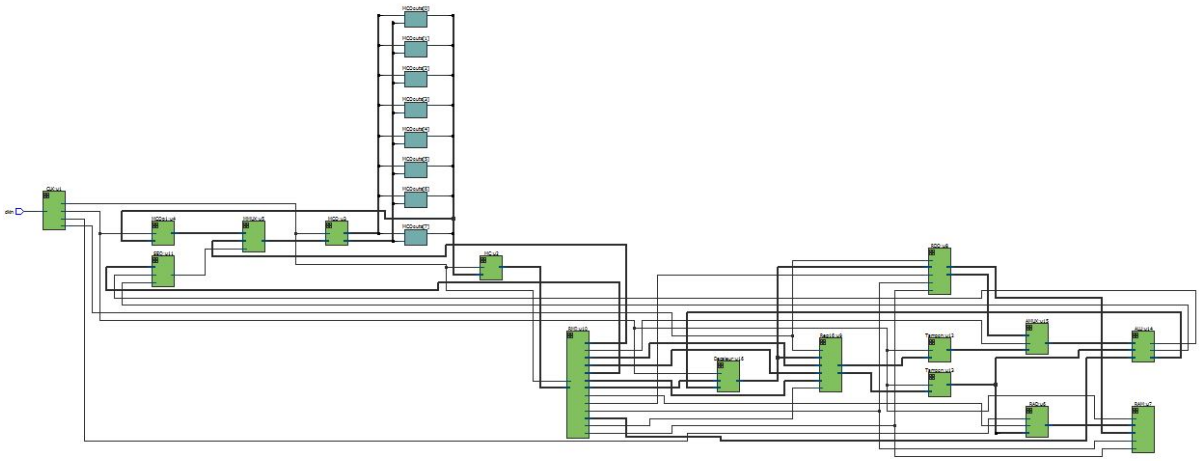


Figure III-1 : Schéma global de l'assemblage des différents blocs

III.2 Description matérielle

```
library IEEE;
use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

entity MIC1 is
    port(clkin: in std_logic
      );
end MIC1;

architecture Behav of MIC1 is

component CLK is
    Port(clock : in std_logic;
         CLK1, CLK2, CLK3, CLK4: out std_logic);
end component;

component MC is
    port(CLK1 : in std_logic;
         MCO: in std_logic_vector(7 downto 0);
         RMI: out std_logic_vector(31 downto 0)
        );
end component;

component MCO is
    port( CLK1 : in std_logic;
    MMUX : in std_logic_vector(7 downto 0);
    MCOout, MCOp1 : out std_logic_vector(7 downto 0)
    );
end component;

component MCOp1 is
    port(CLK2 : in std_logic;
        MCO  : in std_logic_vector(7 downto 0);
        MMUX :  out std_logic_vector(7 downto 0)
);
end component;

component MMUX is port(
    addr, MCOp1 : in std_logic_vector(7 downto 0);
    sel : in std_logic;
    Mout :  out std_logic_vector(7 downto 0)
    );
end component;

component RAD is
    port(
    CLK3, RAD : in std_logic;
    TampB : in std_logic_vector(11 downto 0);
    BusA : out std_logic_vector(11 downto 0)
    );
end component;

component RAM is
    port
    (CLK,WR,RD : in std_logic;
     RAD : in std_logic_vector(11 downto 0);
     RDO : inout std_logic_vector(15 downto 0)
    );
end component;

component RDO is port(
```

```
63
64      CLK4, RDO, RD, WR : in std_logic;
65      Decal : in std_logic_vector(15 downto 0);
66      BusD : inout std_logic_vector(15 downto 0):="0000000000000000";
67      RDOut : out std_logic_vector(15 downto 0)
68      );
69  end component;
70
71  component Reg16 is
72      port
73      (valc, CLK4      : in std_logic;
74       Decal           : in std_logic_vector(15 downto 0);
75       decA,decB,decC  : in std_logic_vector(3 downto 0);
76       BusA,BusB       : out std_logic_vector(15 downto 0)
77      );
78  end component;
79
80  component RMI is
81          port (  CLK1 : in std_logic;
82                  RMIin  : in std_logic_vector(31 downto 0);
83                  AMUX,RDO, RAD, RD, WR, VALC : out std_logic;
84                  COND, UAL, DCAL : out std_logic_vector(1 downto 0);
85                  A, B, C     : out std_logic_vector(3 downto 0);
86                  ADDR : out std_logic_vector(7 downto 0)
87          );
88  end component;
89
90  component SEQ is
91      port(
92      N : in std_logic;
93      Z : in std_logic;
94      COND : in std_logic_vector(1 downto 0);
95      MMUX : out std_logic
96      );
97  end component;
98
99  component Tampon is port(
100     Sinput : in std_logic_vector(15 downto 0);
101     CLK2 : in std_logic;
102     Soutput :   out std_logic_vector(15 downto 0)
103     );
104 end component;
105
106 component ALU is
107 port(
108         A,B : in std_logic_vector(15 downto 0);
109         Op : in std_logic_vector(1 downto 0);
110         S : out std_logic_vector(15 downto 0);
111         FlagN : out std_logic;
112         FlagZ : out std_logic
113         );
114 end component;
115
116 component AMUX is port(
117     A0 : in std_logic;
118     TampA : in std_logic_vector(15 downto 0);
119     Ain : in std_logic_vector(15 downto 0);
120     S : out std_logic_vector(15 downto 0)
121     );
122 end component;
123
124 component Decaleur is
```

```
125     port( CLK2: in std_logic;
126           DCAL: in std_logic_vector(1 downto 0);
127           Din: in std_logic_vector( 15 downto 0);
128           Dout: out std_logic_vector(15 downto 0)
129          );
130 end component;
131
132
133 signal clock1, clock2, clock3, clock4, MMUXs, AMUXs, VALCs, RDs, WRs, RADs, RDOs, I
134 signal DCALs, CONDs, UALs : std_logic_vector(1 downto 0);
135 signal AA, BB, CC: std_logic_vector(3 downto 0);
136 signal MMout, ADDRs, MCOouts, MCOp1s: std_logic_vector(7 downto 0);
137 signal BusAs : std_logic_vector(11 downto 0);
138 signal TIA, TIB, TOA, TOB, BusDs, UALin, UALout, BusCC, DECALs, RDOouts, Ss, Ds: st
139 signal MCouts: std_logic_vector(31 downto 0);
140
141
142 begin
143
144 u1: CLK port map (clock=>clkin, CLK1=>clock1, CLK2=>clock2, CLK3=>clock3, CLK4=>cl
145 u3: MCO port map (CLK1=>clock1, MMUX=>MMout, MCOout=>MCOouts, MCOp1=>MCOouts  );
146 u4: MCOp1 port map (CLK2=>clock2, MCO=>MCOouts, MMUX=>MCOp1s );
147 u2: MC port map (CLK1=>clock1, MCO=>MCOouts, RMI=>MCouts );
148 u10: RMI port map (CLK1=>clock1, RMIin=>MCouts, AMUX=>AMUXs,RDO=>RDOs, RAD=>RADs, I
149 u9: Reg16 port map (valc=>VALCs, CLK4=>clock4, Decal=>DECALs, decA=>AA,decB=>BB,dec
150 u12: Tampon port map (Sinput=>TIA, CLK2=>clock2, Soutput=>TOA );
151 u13: Tampon port map (Sinput=>TIB, CLK2=>clock2, Soutput=>TOB );
152 u6: RAD port map (CLK3=>clock3, RAD=>RADs, TampB=>TOB(11 downto 0), BusA=>BusAs );
153 u7: RAM port map (CLK=>clock2, WR=>WRs,RD=>RDs, RAD=>BusAs, RDO=>BusDs );
154 u8: RDO port map (CLK4=>clock4, RDO=>RDOs, RD=>RDs, WR=>WRs,Decal=>DECALs,BusD=>Bus
155 u15: AMUX port map (A0=>AMUXs, TampA=>TOA, Ain=>RDOouts, S=>Ss );
156 u14: ALU port map (A=>Ss, B=>TOB, Op=>UALs, S=>Ds, FlagN=>Ns, FlagZ=>Zs );
157 u11: SEQ port map (N=>Ns, Z=>Zs, COND=>CONDs, MMUX=>MMUXs );
158 u5: MMUX port map (addr=>ADDRs, MCOp1=>MCOp1s, sel=>MMUXs, Mout=>MMout);
159 u16: Decaleur port map (CLK2=>clock2, DCAL=>DCALs, Din=>Ds, Dout=>DECALs );
160
161
162 end Behav;
```

III.3 Résultat de compilation (utilisation des ressources)

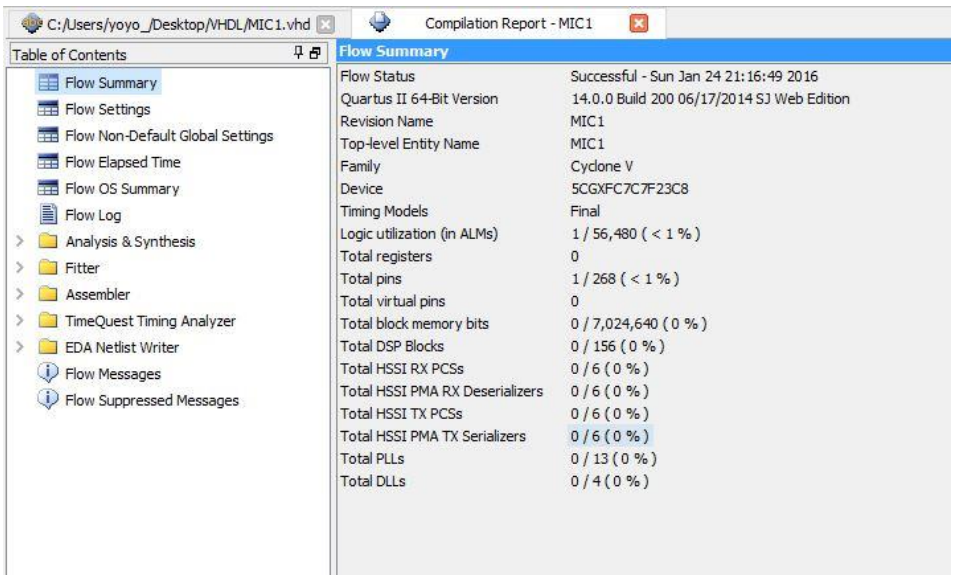


Figure III-2 : Rapport d'analyse de l'utilisation des ressources

III.4 Conclusion

Dans ce projet on a utilisé le langage VHDL et l'outil Quartus d'Altera afin de décrire et concevoir la Machine MIC1, on a commencé par décrire chaque bloc individuellement et ensuite on a connecté l'ensemble ce ces blocs dans une seule description Top-Level.

CONCLUSION GENERALE

Nous avons commencé cette étude en réalisant un état de l'art sur les différents processeurs à travers le chapitre I. Pour y parvenir, nous avons identifié une certaine classification des processeurs ainsi que celle des machines.

Par la suite, nous avons décrit l'architecture de la machine MIC-1 et ses cycles fonctionnels. Chacun des blocs composant cette machine a été décrit en Vhdl et validé par simulation fonctionnelle.

Ainsi dans le dernier chapitre, nous avons en premier lieu, réuni tous les blocs décrits auparavant en un seul schéma de blocs en les connectant entre eux tout en respectant l'architecture de la machine MIC-1. Par la suite, nous avons généré une description Top-Level grâce à l'outil Quartus de Altéra pour valider l'architecture.

Bibliographie

[1]. S. Heath. Microprocessor architectures RISC, CISC and DSP. Newnes-Butterworths, 2nd edition, 1995.

[2]. J.L. Hennessy and D.A. Patterson. Organisation et conception des ordinateurs, l'interface materiellogiciel. International Thomson Publishing France, 1994.

[3]. D.A. Patterson and C.H. Sequin. A VLSI RISC. IEEE Computer Magazine, 15(9):8–21, 1982.

[4]. G. Radin. The 801 minicomputer. In ACM SIGARCH-10.2 SIGPLAN-17.4, editor, Symposium on Architectural Support for Programming Languages ans Operating Systems (ASPLOS-I), pages 39–47, 1982.

[5]. J.L. Hennessy and D.A. Patterson. Architecture des ordinateurs, approche quantitative. International Thomson Publishing France, 2nd edition, 1996.

[6]. A.Y Zomaya. Parallel and distributed computing handbook. McGraw-Hill, 1996.

[7]. J.R. Ellis. BULLDOG: a Compiler for VLIW Architectures. MIT Press, Cambridge, 1986.

[8]. J.L. Jacquemin. Informatique parallele et systemes multiprocesseurs. Hermes, 1993.

[9]. R. Leupers. Retargetable code generation for digital signal processing. Kluwer Academic Publ.,1997.

[10]. H. Zeltwanger. An inside look at the fundamentals of CAN. Control Engineering, pages 51–56, January 1995.

[11]. D. Paret. Le bus I2C. Dunod, 1994.

[12]. Texas Instruments. TMS320C80 User's guide, 1995.

[13]. M.J. Flynn. Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21(9):948–960, Septembre 1972.

[14]. J. Blazewicz, K. Ecker, and D. Trystram B. Plateau. Handbook on parallel and distributed processing. Springer, 2000.

[15]. J.L. Hennessy and D.A. Patterson. Organisation et conception des ordinateurs, l'interface materiellogiciel. International Thomson Publishing France, 1994.

[16]. T. Grandpierre. Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés. France. 2000.