ENSEIRB-MATMECA

EN 216

Processeur

Conception d'un processeur avec jeu d'instructions élémentaires

Auteurs :
Wilfried Baradat
Maxime Descos

Table des matières

1 Introduction

L'objectif de ce module est d'assembler des fonctions élémentaires (combinatoires et séquentielles) pour réaliser un processeur programmable avec un jeu d'instructions élémentaires. L'architecture conçue sera intégrée sur une carte de prototypage FPGA.

2 Processeur avec jeu d'instructions élémentaires

Le processeur à concevoir est un processeur 8-bits à usage universel. Il est capable d'exécuter 4 types d'instructions. Ce processeur est basé sur un registre accumulateur appelé ACCU de taille 8 bits. Chaque instruction est codée sur 8 bits. Deux bits pour coder le type de l'opération (code.op) et 6 bits pour coder l'opérande ou l'adresse de l'opérande dans la mémoire selon le type de l'instruction.

Les quatre instructions présentes sont les suivantes (ACCU représente un des registres d'entrée de l'ALU et le registre qui écrit dans la RAM) :

- NOR : 00 AAAAAA, i.e. ACCU = ACCU NOR Mem[AAAAAA]
- ADD : 01 AAAAAA, i.e. ACCU = ACCUR + Mem[AAAAAA]
- STA : 10 AAAAAA, i.e. Mem[AAAAAA] = ACCU
- JCC : 11 DDDDDD, i.e. Si Carry = 0, alors PC DDDDDD, sinon effacer la retenue (Carry = 0)

De plus, la mémoire contient à la fois le programme et les données, on a donc une architecture de type Von Neumann. Étant donné que l'on a 6 bits pour l'adresse, il nous faut une mémoire de taille $2^6 = 64$ octets.

Passons à l'architecture du processeur. Il est découpé en trois blocs :

- L'unité de traitement composée de l'ALU (Unité Arithmétique et Logique, du registre ACCU (registre de base du processeur), d'une retenue (carry) après l'ALU et d'un registre (R1) stockant la deuxième donnée d'entrée de l'ALU.
- L'unité de mémorisation composée d'une mémoire RAM de 64 octets contenant le programme à exécuter. Cette mémoire permet une écriture synchrone et une lecture asynchrone.
- L'unité de contrôlé composée d'une machine d'état (FSM), d'un registre (RI) contenant l'instruction à exécuter, d'un compteur programme (PC) et d'un multiplexeur permettant de choisir si l'adresse à envoyer à la RAM vient de la FSM ou du PC.

On a alors le schéma bloc suivant :

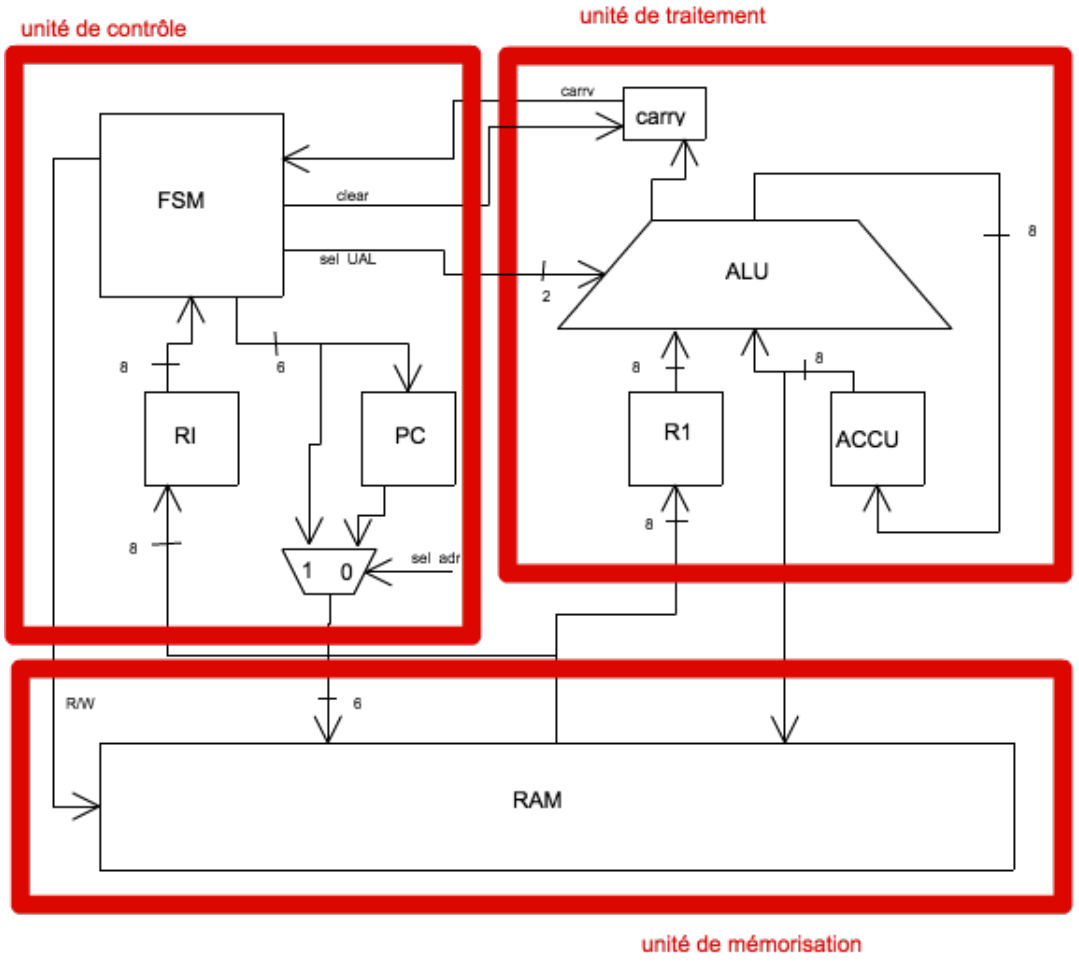

FIGURE 1 – Schéma bloc du processeur

2.1 Fonctionnement de l'unité de traitement

Cette unité est le coeur de la partie opérative du processeur. Elle fonctionne suivant le principe suivant : les registres R1 et ACCU envoient leurs données à l'ALU. L'ALU décode le signal *sel_UAL* venant de la FSM qui indique l'opération à effectuer (NOR ou ADD). Après l'opération, le résultat est stocké de nouveau dans l'ACCU. De plus, la retenue de l'opération est stockée dans Carry. Ce bit permet d'effectuer des opérations de comparaison qui permettent les "jump".

2.2 Fonctionnement de l'unité de mémorisation

Cette unité est basique et n'a pas ici de nouveautés particulières. La FSM indique si l'accès est en lecture ou en écriture.

— en lecture : de manière asynchrone, la RAM renvoie constamment la donnée stockée à l'adresse spécifiée. Cette donnée est alors envoyée au registre instruction de la partie contrôle (afin de lire l'instruction), et au registre de l'unité de traitement (afin d'effectuer une opération entre une donnée de la mémoire et le registre ACCU).

— en écriture : de manière synchrone, la RAM lit l'adresse qu'on lui envoie, va à la case mémoire correspondante et copie la donnée venant du registre ACCU dans cette case.

2.3 Fonctionnement de l'unité de contrôle

Cette unité est le coeur du processeur mais également la partie qui peut être à l'origine de la plupart des problèmes. En effet, elle doit gérer tous les autres blocs et s'assurer que le processeur effectue correctement les étapes. Pour cela, une machine d'états est implantée. Elle a 7 états :

— INIT : état d'initialisation au démarrage pour effacer tous les registres.

— FETCH_IN : aller chercher l'instruction à exécuter dans la mémoire.

— DECODE : décoder l'instruction.

— JCC : aller à une instruction particulière suivante la retenue.

— STORE : écrire dans la mémoire la donnée stockée dans ACCU.

— FETCH_OP : aller chercher la donnée nécessaire pour l'opération à effectuée si c'est une opération de l'unité de traitement.

— EXE_NOR_ADD : effectuer l'opération arithmétique ou logique.

Voici ci-dessous le tableau montrant l'état des signaux de contrôle de chaque bloc suivant l'état de la FSM :

state	UAL	R1	ACCU	CARRY	MEM	PC	RI	MU_ADDR
INIT	∅	clear	clear	clear	R/W = 0	clear	clear	sel_adr = 0
FETCH_IN	∅	$\overline{load}$	$\overline{load}$	$\overline{load}$	R/W = 0	incr	load	sel_adr = 0
DECODE	∅	$\overline{load}$	$\overline{load}$	$\overline{load}$	R/W = 0	$\overline{incr}$	$\overline{load}$	sel_adr = 1
JCC	111	$\overline{load}$	$\overline{load}$	load	R/W = 0	$\overline{incr}$	$\overline{load}$	dépend de la carry
STORE	111	$\overline{load}$	$\overline{load}$	$\overline{load}$	R/W = 1	$\overline{incr}$	$\overline{load}$	1
FETCH_OP	111	load	$\overline{load}$	$\overline{load}$	R/W = 0	$\overline{incr}$	$\overline{load}$	sel_adr = 1
EXE_NOR_ADD	RI[0-1]	$\overline{load}$	load	load	R/W = 0	$\overline{incr}$	$\overline{load}$	sel_adr = 1

3 Algorithme du PGCD

Pour tester notre processeur, nous allons utiliser un programme qui calcule le plus grand commun diviseur de deux nombres. On utilise la propriété suivante pour construire l'algorithme :

> Si un nombre est diviseur de 2 nombres a et b, alors il est aussi diviseur de leur différence. Si cette différence est positive alors on remplace a par la valeur de la différence sinon on remplace b par la valeur de la différence.
>
> On va alors calculer en boucle la différence de a et b et effectuer deux actions différentes suivant le signe de cette différence. Pour calculer la différence, on utilise le complément à deux : pour avoir $-b$, on fait $-b = non(b) + 1$. On ajoute alors a et $-b$ et on analyse la retenue. Si la retenue est égale à 0, alors la différence est négative. Dans ce cas-là, on change le signe de la différence, on la met dans b et on recommence l'algorithme.
>
> Sinon, on met la différence dans a, on vérifie si a est nul en lui ajoutant 11111111. On a alors la retenue qui est positive et on recommence l'algorithme, sauf si $a = 0$, dans ce cas, on est arrivé au bout et le PGCD est b.
>
> Pour cet algorithme, on utilise seulement les quatre opérations existantes dans le processeur. Cet algorithme est stocké dans les premières cases mémoires de la RAM, le compteur programme (PC) va se charger de l'ordre avec lequel les instructions seront lues.

4 Implémentation des trois blocs

4.1 Partie opérative

La partie opérative est composée des blocs suivants :
— l'ALU (pour Arithmetic and Logic Unit, UAL en français) ;
— la bascule CAR (pour Carry) ;
— le registre R1 ;
— le registre ACCU.

4.1.1 L'ALU

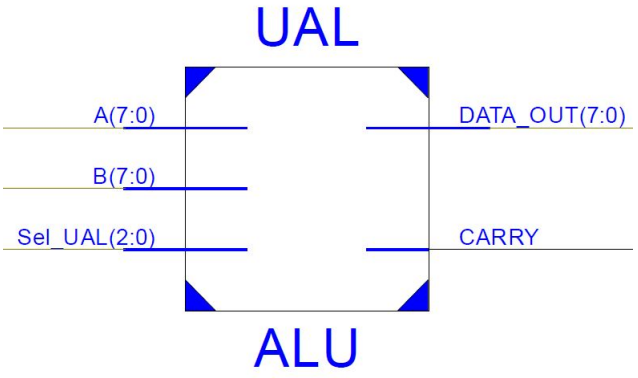

FIGURE 2 – Symbole du bloc ALU

L'ALU est décrite à l'aide d'un processus combinatoire. Celui-ci est déclenché lorsque un des opérandes A ou B change, ou lorsque le signal *Sel_UAL* contrôlé par la FSM change.

On créé ici un signal intermédiaire *data_result* sur 9 bits.

Le signal *Sel_UAL* contient l'information d'exécution d'un NOR ou d'un ADD. Lorsque celui-ci demande un NOR, on insère le résultat du NOR sur les 8 LSB du signal intermédiaire et on rajoute un '0' en MSB. Lorsque celui-ci demande un ADD, on insère le résultat de l'ADD sur les 9 bits du signal intermédiaire, permettant de prendre en compte une retenue en MSB.

En dehors du processus, on attribue alors les 8 LSB du signal *data_result* à la sortie *DATA_OUT* et le MSB de ce signal au signal *CARRY*.

4.1.2 La bascule CAR (Carry)

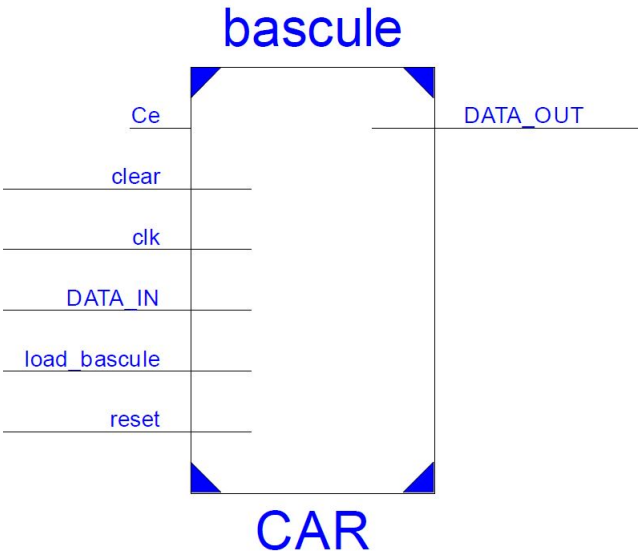

FIGURE 3 – Symbole de la bascule Carry

Le bloc Carry est une simple bascule générique, c'est-à-dire qu'elle maintient la valeur de la sortie $DATA\_OUT$ même lorsque la valeur de $DATA\_IN$ change. En effet, le basculement de la valeur de $DATA\_IN$ à $DATA\_OUT$ se fait uniquement lorsque le signa $load\_bascule$ est égale à 1.

4.1.3 Les registres R1 et ACCU

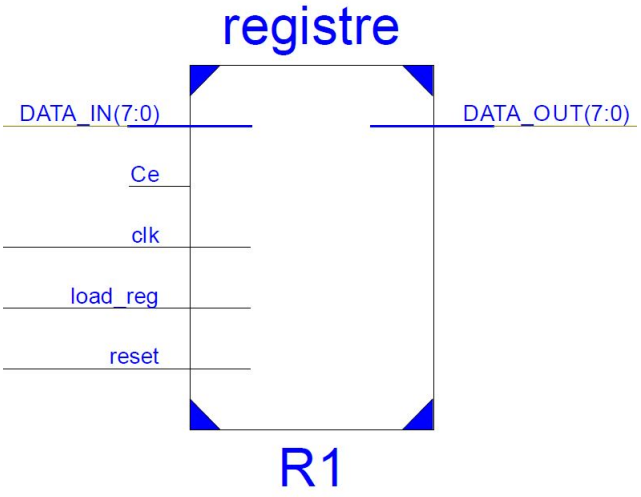

FIGURE 4 – Symbole de la bascule R1

Les registres R1 et ACCU sont les mêmes composants : un registre générique. Une seule petite nuance apparaît dans le code par rapport à la bascule : on recopie le signal d'entrée sur un signal intermédiaire (appelé ici *data*) lors d'un front d'horloge et lorsque le signal *load_reg* vaut 1, mais le signal de sortie $DATA\_OUT$ est assigné avec le signal *data* de manière asynchrone.

4.2 Partie mémorisation

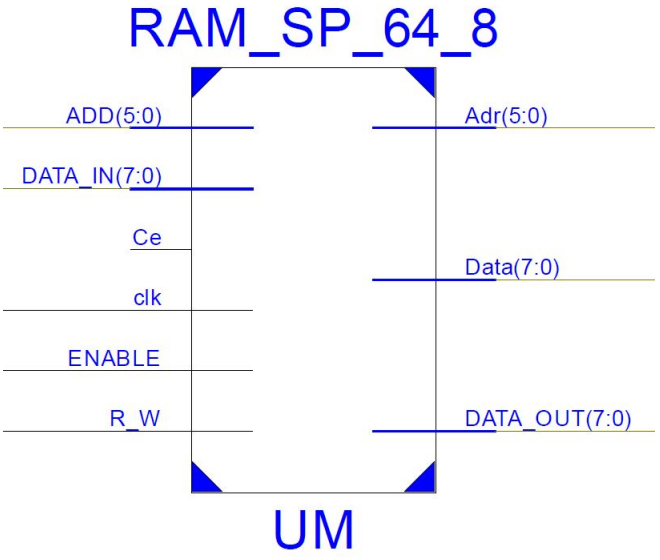

FIGURE 5 – Symbole du bloc RAM

La partie mémorisation est constituée uniquement du bloc RAM_SP_64_8. C'est une RAM de 64 octets à écriture synchrone et lecture asynchrone.

Celle-ci est constituée d'un seul processus déclenché par les fronts d'horloge. Dans un premier temps, lorsque le signal *ENABLE* est à 1, on initialise le tableau correspondant à la RAM. Dans notre cas, on insère dans les 22 premières cases l'algorithme retranscrit en octets ainsi que des données. Les 42 autres cases n'étant pas utilisés dans cet algorithme, on les remplit de zéros à l'aide d'un *for loop*. La RAM est maintenant initialisée. Lorsqu'une écriture sur la RAM est demandée, on écrit à l'adresse spécifiée par le bloc *Control_unit* l'octet délivré par le bloc *Processing_unit*.

Enfin, les signaux de sortie sont affectés hors du processus et sont mis à jour dès qu'un changement dans le tableau *ram* ou du signal *ADD* a lieu.

4.3 Partie contrôle

La partie contrôle est composée des blocs suivants :
— la FSM (pour Finite State Machine ou machine d'états en français) ;
— le compteur programme PC ;
— le multiplexeur mux_2to1_6bit ;
— le registre d'instructions RI.

4.3.1 La FSM

FIGURE 6 – Symbole du bloc RAM

La machine d'états ou FSM est décomposée en deux processus :
— un processus séquentiel permettant le passage de l'état présent à l'état futur à chaque coup d'horloge ;
— un processus combinatoire constitué d'un *case* permettant d'assigner chaque signal de contrôle pour chaque état.

A l'aide du tableau vu précédemment (référençant les signaux de contrôle pour chaque état), il est alors simple d'attribuer chaque signaux de contrôle dans le second processus.

4.3.2 Le compteur programme PC

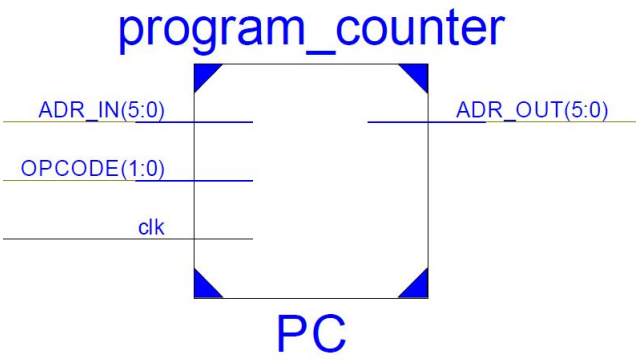

FIGURE 7 – Symbole du bloc RAM

Le compteur de programme PC a été implémenté sous la forme de deux processus :

- un processus séquentiel permettant le changement du compteur programme futur au compteur programme présent (signaux *next_pc* et *current_pc*) ;
- un processus combinatoire effectuant des actions différentes en fonction du signal *OPCODE*.

Suivant la valeur du signal *OPCODE*, on effectue les actions suivantes :

- Quand *OPCODE* vaut *PC_OP_NOTHING* (i.e. "00"), on ne fait rien ;
- Quand *OPCODE* vaut *PC_OP_INCREMENT* (i.e. "01"), on incrémente le compteur programme de 1 ;
- Quand *OPCODE* vaut *PC_OP_ASSIGN* (i.e. "10"), on assigne une valeur spécifiée en entrée (signal *ADR_IN*) au compteur programme. Cette assignation permet d'effectuer un jump ;
- Quand *OPCODE* vaut *PC_OP_RESET* (i.e. "11"), on remet le signal intermédiaire *next_pc* à 0.

4.3.3 Le multiplexeur mux_2to1_6bit

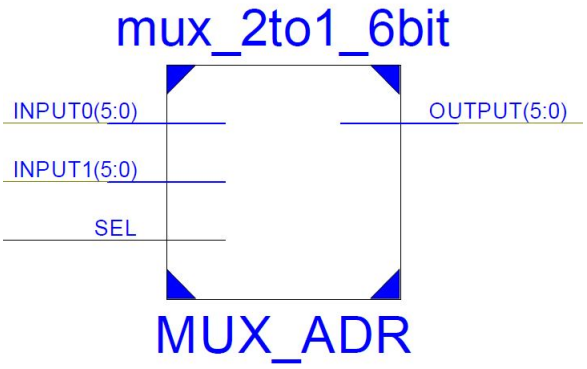

FIGURE 8 – Symbole du bloc RAM

Le multiplexeur mux_2to1_6bit est, comme son nom l'indique, un multiplexeur permettant de sélectionner un signal parmi deux signaux, tous les signaux étant de 6 bits.

Cette sélection se fait de manière combinatoire à l'aide du signal de sélection *SEL*.

4.3.4 Le registre d'instructions RI

Le registre d'instructions RI est un registre du même type que les registres ACCU et R1, on utilise donc le même composant.

5 Prototypage

Pour pouvoir mettre en évidence le fonctionnement du programme, on utilise le bloc acces_carte permettant de gérer les afficheurs 7 segments. Ce dernier affiche sur 4 afficheurs l'adresse et le contenu de la mémoire à cet adresse, tout cela en hexadécimal. Notre processeur fonctionne effectivement comme on peut le voir sur l'image ci-dessous. Les afficheurs affichent : 15 0C, 15 correspondant à l'adresse 21 en décimal, 0C correspondant à 12 en décimal. On obtient bien le PGCD de 60 et 36.

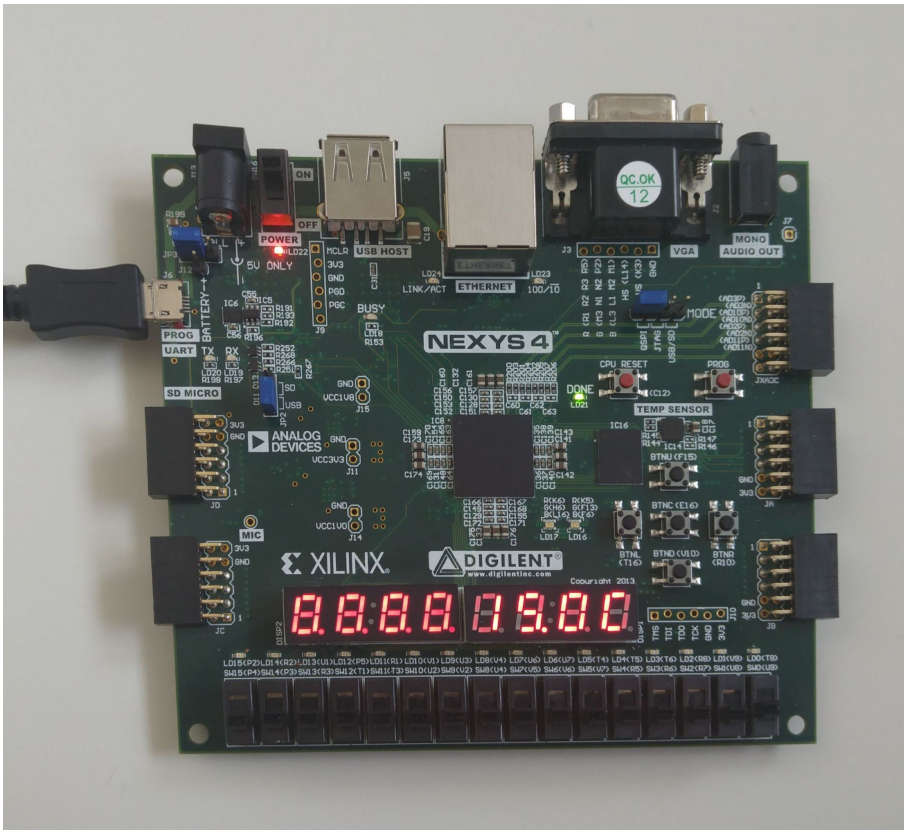

FIGURE 9 – Photographie de la Nexys4 exécutant le processeur élémentaire

6 Conclusion

Ce projet, outre le fait de commencer à nous montrer le fonctionnement d'un processeur de la théorie jusqu'à l'implémentation, nous a permis de travailler et d'apprendre en autonomie pendant plusieurs mois. D'un point de vue théorique, ce sujet a demandé une recherche et une compréhension approfondie d'algorithme, de technique de conception que nous n'avions jamais vu.

D'un point de vue de la théorie, nous avons pu voir comment est découpé un processeur et quelle est la logique derrière chacune de ces parties..

Par ces différents aspects, et par la rédaction de ce rapport, les connaissances apportées par la réalisation de ce projet sont nombreuses.