CentraleSupélec | université PARIS-SACLAY

Projet - Architecture de processeur

Architecture des Systèmes Numériques

Téo Lefebvre, Jules Pigasse

Février 2024

Table des matières

Introduction

Pour ce projet, on se fixe l'objectif de créer un processeur capable d'exécuter le programme 1, écrit en langage C. Dans l'idéal, il faudrait pouvoir charger les données de la mémoire principale, effectuer le traitement et écrire le résultat dans la mémoire principale également.

```c
void gradient(int *inVect, int *outVect, int vectSize) {
  for (int i = 1; i < vectSize; i++)
    outVect[i] = 3 * inVect[i] - 2 * inVect[i-1];
}

int main(int argc, char **argv) {
  int inVect[8] = {16, 16, 0, 0, 0, 16, 16, 16};
  int outVect[8] = {0};
  gradient(inVect, outVect, 8);
  return 0;
}
```

Listing 1 – Programme à exécuter

1 Instruction Set Architecture

Dans un premier temps, avant de concevoir notre processeur, il faut définir notre ISA. Dans cette partie, on va justifier les choix que l'on a fait pour notre ISA.

Pour la taille des données, on voit dans le programme 1 que les données vont dans le pire des cas jusqu'à 48 ($3 * 16$) et peuvent être négatives. On a donc choisit de stocker uniquement des entiers signés, et ce sur un octet (donc on peut coder des nombres de -128 à 127). On aurait pu se limiter à 7 bits, mais il est plus logique de stocker des données sur des octets et de cette manière, on a un peu de marge.

Puisqu'on code les nombres sur un octet, on va se limiter à une mémoire principale (RAM) avec $2^8 = 256$ adresses.

Étant donné que l'on va stocker les données d'entrée et de sortie en mémoire principale, on n'a pas besoin de beaucoup de registres. On se limitera donc à 8 registres, dont le premier (R0) servira à stocker les valeurs inutiles des calculs pour les conditions. On peut ainsi coder l'adresse d'un registre sur 3 bits.

Il faut maintenant déterminer les instructions dont on va avoir besoin. Naturellement, il va nous falloir les opérations ADD et SUB, ainsi que la possibilité d'écrire une valeur dans un registre avec MOV.

On veut être capable de faire des boucles et des appels à des fonctions, il va donc nous falloir une instruction de saut JUMP et une instruction de saut conditionnelle ZJUMP, ainsi qu'une pile que l'on va manipuler avec les instructions PUSH et POP. Les instructions précédentes sont suffisantes pour faire des appels de fonction, donc

on n'a pas besoin d'implémenter spécifiquement les instructions CALL et RET. Si on veut les utiliser, on peut les traduire avec le compilateur.

Enfin pour charger des données depuis la mémoire principale vers les registres et en écrire, il va nous falloir les instructions LOAD et WRITE.

On ajoute également une instruction NOP qui ne fait rien et arrête le programme en désactivant tous les composants.

Pour effectuer une multiplication, on a toutes les instructions pour coder une fonction permettant de multiplier deux nombres. Cependant par soucis de simplicité, on va ajouter une instruction MUL à notre ISA.

On obtient alors l'ISA décrite dans le tableau 1. Pour les formats 00 et 01, les sources RA et RB sont sur 4 bits mais on n'utilise que les 3 bits de poids faible étant donné que ce sont les adresses de registres.

2	3	3	4	4		
OpCode		RDestination	Source RA	Source RB	Mnemonic	Fonction
Format	Opérateur					
00	000	RID	RID	RID	ADD	RDest = RA + RB
00	001	RID	RID	RID	SUB	RDest = RA - RB
00	010	RID	RID	RID	MUL	RDest = RA * RB
00	011	RID	RID	NA	MOV	RDest = RA
00	100	NA	NA	NA	NOP	Ne fait rien et arrête l'execution du programme (enable = 0)
00	101	NA	NA	NA		
00	110	RID	RID	NA	LOAD	Charge dans RDest la donnée de la RAM à l'adresse indiquée dans RA
00	111	RID	RID	NA	WRITE	Ecrit la valeur de RDest dans la RAM à l'adresse indiquée dans RA

2	3	3	4	4		
OpCode		RDestination	Source RA	Immédiat	Mnemonic	Fonction
Format	Opérateur					
01	000	RID	RID	Unsigned Int 4 bits	ADD	RDest = RA + Imm
01	001	RID	RID	Unsigned Int 4 bits	SUB	RDest = RA - Imm
01	010	RID	RID	Unsigned Int 4 bits	MUL	RDest = RA * Imm
01	011	NA	NA	NA		
01	100	RID	NA	NA	PUSH	Ajouter la valeur du registre RDest à la stack
01	101	NA	NA	NA		
01	110	NA	NA	NA		
01	111	NA	NA	NA		

2	3	3	8		
OpCode		RDestination	Immédiat long	Mnemonic	Fonction
Format	Opérateur				
10	000	NA	Int 8 bits	JUMP	Aller à la ligne Imm
10	001	NA	Int 8 bits	ZJUMP	Aller à la ligne Imm si le résultat du calcul précédent vaut 0
10	010	RID	NA	JUMP	Aller au numéro de ligne contenu dans RDest
10	011	RID	Int 8 bits	MOV	RDest = Imm
10	100	NA	Int 8 bits	PUSH	Ajouter la valeur Imm à la stack
10	101	RID	Int 8 bits	POP	Enlève la valeur de la stack à l'index Imm et la sauvegarde dans RDest
10	110	RID	Int 8 bits	LOAD	Charge dans RDest la donnée de la RAM à l'adresse Imm
10	111	RID	Int 8 bits	WRITE	Ecrit la valeur de RDest dans la RAM à l'adresse Imm

FIGURE 1 – Tableau de définition de l'ISA

2 Architecture du processeur

Pour notre CPU, on a donc besoin des éléments suivants :

— Decode
— ALU
— Register File
— Fetch
— ROM
— RAM

— Stack

On peut voir l'architecture du CPU et les signaux qui relient les différents composants (qu'on détaille ci-dessous) sur la figure 2.

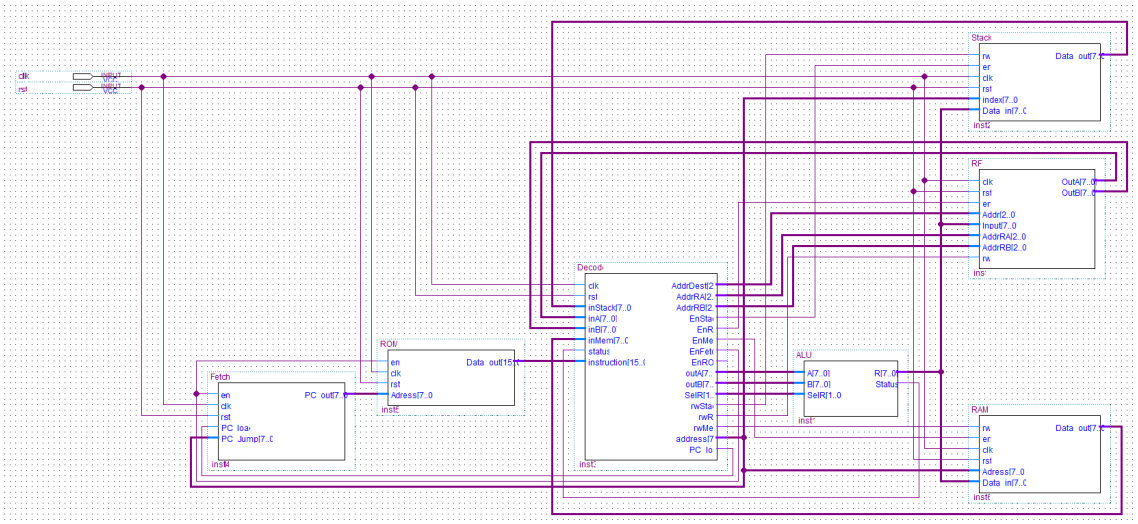

FIGURE 2 – Schéma du CPU

On connecte ces composants par les signaux suivants.

Les entrées du CPU :
— clk (std_logic) : horloge (tous les composants sont synchronisés sauf l'ALU)
— rst (std_logic) : le signal de reset pour initialiser l'état de tous les composants

Les signaux internes entre les composants :
— EnStack (std_logic) : pour activer ou non la Stack
— EnRF (std_logic) : pour activer ou non le Register File
— EnRAM (std_logic) : pour activer ou non la RAM
— EnFetch (std_logic) : pour activer ou non le Fetch
— EnROM (std_logic) : pour activer ou non la ROM
— rwStack (std_logic) : si PUSH alors rw = '1' / si POP alors rw = '0'
— rwRF (std_logic) : '1' si on veut écrire / '0' si on veut lire
— rwRAM (std_logic) : '1' si on veut écrire / '0' si on veut lire
— A (std_logic_vector(7 downto 0)) : première entrée de l'ALU
— B (std_logic_vector(7 downto 0)) : deuxième entrée de l'ALU
— SelR (std_logic_vector(1 downto 0)) : pour sélectionner l'opération à effectuer ("00" = add / "01" = sub / "10" = mul)
— Result (std_logic_vector(7 downto 0)) : le résultat du calcul de l'ALU
— Status (std_logic) : si le résultat est nul alors '1' sinon '0'
— AddrDest (std_logic_vector(2 downto 0)) : adresse du registre dans lequel écrire
— AddrRA (std_logic_vector(2 downto 0)) : adresse du premier registre à lire
— AddrRB (std_logic_vector(2 downto 0)) : adresse du second registre à lire
— address (std_logic_vector(7 downto 0)) : index dans la pile pour la Stack, adresse d'une donnée pour la RAM, numéro de ligne pour le Fetch

— PC_load (std_logic)) : si '1' alors le Fetch doit aller à la ligne indiquée par address sinon il doit incrémenter d'une ligne.

Le fonctionnement des différents composants est celui que l'on pourrait attendre de chacun d'eux et se passe d'explications excepté pour le Decode qui présente quelques spécificités. Etant donné que tous les composants sont synchrones (sauf l'ALU), si une instruction a besoin de données de la Stack, du Register File ou de la RAM, il faut que le Decode les demande avant de pouvoir les utiliser deux fronts d'horloge plus tard. Un processeur pipeliné permettrait peut-être de contourner ce problème.

Il faut noter qu'il n'est pas prévu que le processeur gère les erreurs. Par exemple, si on tente de POP la valeur d'indice 10 de la Stack alors qu'elle n'est que de longueur 5, le programme va planter. La précaution à avoir est laissée au développeur.

Étant donné qu'il y a beaucoup d'instructions, il est très fastidieux de réaliser le testbench du Decode, d'autant plus que celui ci n'a pas tellement plus d'intérêt que de simplement relire le code. Comme le programme 1 va utiliser la quasi-totalité des instructions, si on arrive à le faire fonctionner, on pourra considérer que le Décode fonctionne correctement (ainsi que les autres composants, mais eux ont tous été testés individuellement).

3 Programme et simulation

La traduction en langage assembleur du programme 1 est donnée dans le programme 2. On charge le programme dans la ROM au moment du reset. On charge les 8 valeurs de inVect dans la RAM dans les adresses de 0 à 7 au moment du reset également. Les valeurs du outVect sont stockées dans la RAM aux adresses 8 à 15.

```
1  MOV R1 1
2  SUB R0 R1 8
3  ZJUMP 30 # 29 en binaire car les adresses de la ROM commencent par
       0
4  PUSH R1 # argument i pour GRADIENT
5  PUSH 7 # ligne a executer au retour de la fonction, 6 en binaire
6  JUMP GRADIENT
7  ADD R1 R1 1
8  JUMP 2
9
10 GRADIENT:
11 PUSH R1 # push pour sauvegarder les valeurs des registres qu'on va
       utiliser
12 PUSH R2
13 PUSH R3
14 POP R1 4 # argument i
15 LOAD R2 R1 # inVect[i]
16 SUB R1 R1 1 # i-1
17 LOAD R3 R1 # inVect[i-1]
18 MUL R2 R2 3
```

```
19 MUL R3 R3 2
20 SUB R2 R2 R3
21 ADD R1 R1 9 # i+8 pour obtenir l'adresse en RAM de outVect[i]
22 WRITE R2 R1 # ecrire en RAM la valeur de R2 a l'adresse contenue
       dans R1
23 POP R3 # pop pour reinitialiser les registres aux valeurs
       sauvagardees
24 POP R2
25 POP R1
26 POP R0 # ligne ou retourner apres la fonction
27 JUMP R0 # retourne a la ligne dans R0
28
29 PRINT:
30 LOAD R1 8 # pour voir passer les valeurs dans les chronogrammes
31 LOAD R1 9
32 LOAD R1 10
33 LOAD R1 11
34 LOAD R1 12
35 LOAD R1 13
36 LOAD R1 14
37 LOAD R1 15
```

Listing 2 – Programme en assembleur

On réalise une simulation du CPU [**2**]. On obtient alors le chronogramme suivant avec tous les signaux mentionnés plus haut. On peut voir à la fin de la simulation, dans le chronogramme figure 3 les valeurs calculées de outVect grâce au signal inMem (qui correspond aux valeurs lues dans la mémoire par les instructions LOAD). Les valeurs correspondent bien à celle attendues (c'est à dire $[0, 16, -32, 0, 0, 48, 16, 16]$).

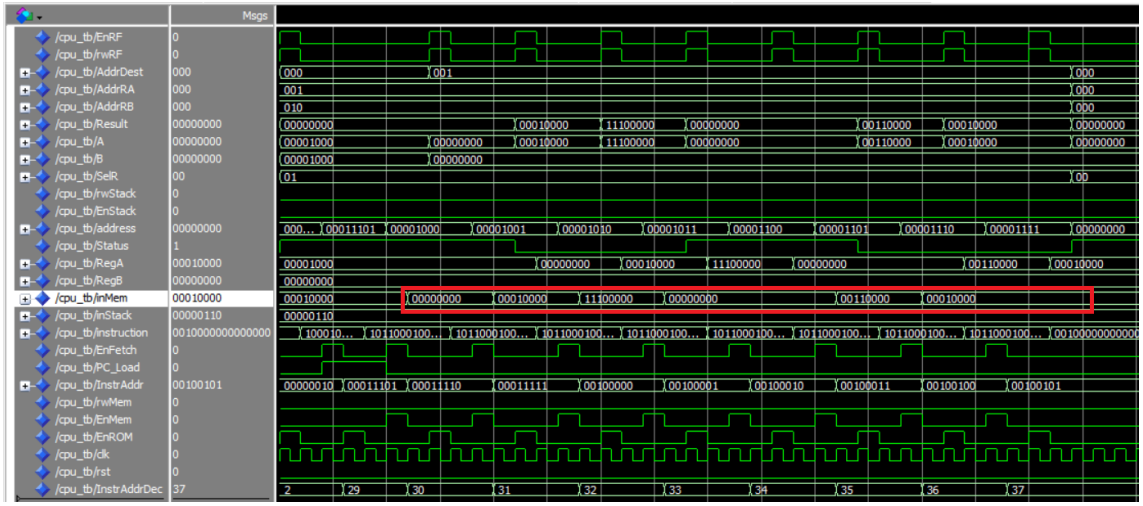

FIGURE 3 – Fin du chronogramme de la simulation du CPU

Nous tenons également à mentionner le site FPGATutoriel [**1**] qui nous a bien aidé avec les subtilités du VHDL.

Bibliographie

[1] *Formation VHDL*. URL : https://fpgatutorial.com/vhdl/.

[2] Suman SAMUI. *Intel Quartus Prime Lite edition | Behaviourial Simulation using VHDL Testbench code*. URL : https://www.youtube.com/watch?v=76cNeZWTjc0.