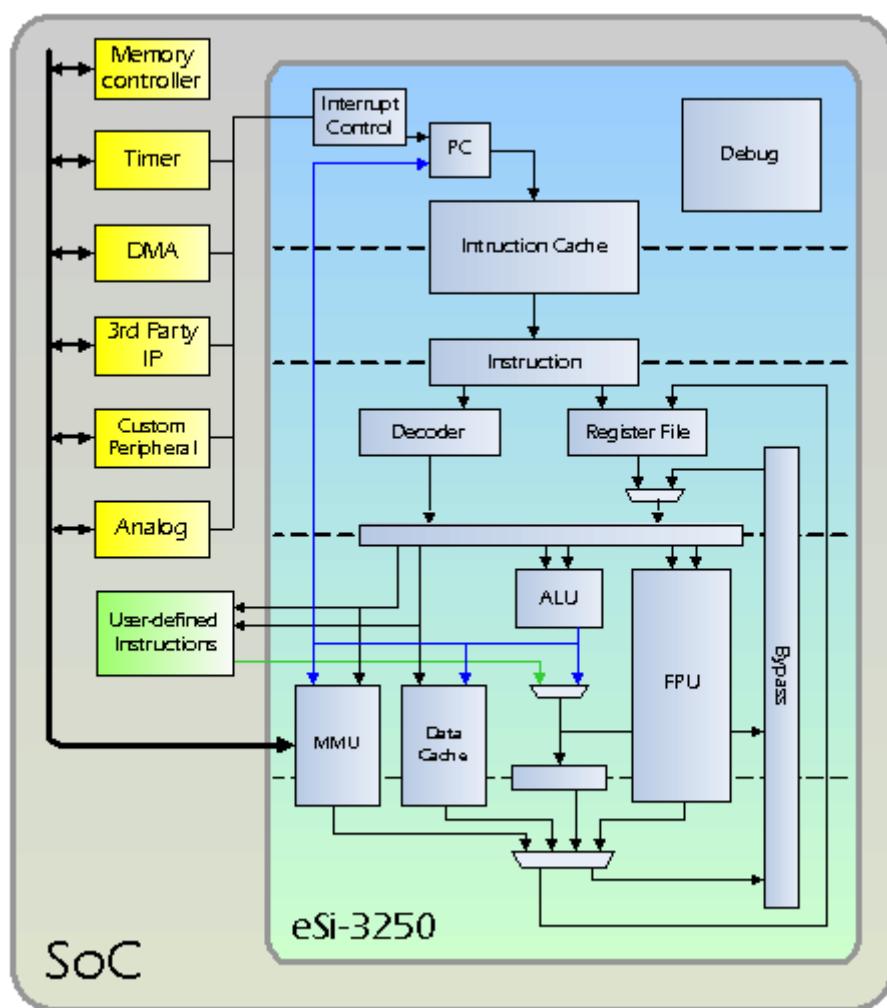


COMPTE-RENDU PROCESSEUR MONO-CYCLE



Sommaire :

<u>Introduction</u>	3
<u>I - Unité de traitement</u>	4
i - ALU.....	4
ii - Banc de registres.....	5
iii - Assemblage.....	7
iv - Multiplexeur 2 vers 1.....	8
v - Extension de signe.....	9
vi - Mémoire de données.....	9
vii - Assemblage.....	10
<u>II - Unité de gestion des instructions</u>	12
i - Mémoire d'instructions.....	12
ii - Registre 32 bits PC.....	12
iii - Unité de mise à jour du compteur PC.....	13
iv - Assemblage.....	13
<u>III - Unité de contrôle</u>	14
i - Registres 32 bits PSR et AFFICHAGE.....	14
ii - Décodeur d'instructions.....	14
<u>IV - Assemblage complet du processeur</u>	15
i - Assemblage.....	15
ii - Test de fonctionnement.....	15
<u>V - Implémentation sur carte FPGA</u>	17
<u>VI - Gestion des interruptions externes</u>	18
i - VIC.....	18
ii - Explication des instructions.....	18
iii - Modification et assemblage dans le processeur.....	20
<u>VII - Périphérique UART</u>	23
i - Emission.....	23
ii - Réception.....	25

Introduction

Un processeur est un composant présent dans de nombreux dispositifs électroniques qui exécute les instructions machines des programmes informatiques. Ce n'est pas qu'une unité de calcul. Cette dernière est incluse dans le processeur mais il fait aussi appel à une unité de contrôle, une unité d'entrée-sortie, à une horloge et à des registres.

Le séquenceur, ou unité de contrôle, se charge de gérer le processeur. Il peut décoder les instructions, choisir les registres à utiliser, gérer les interruptions ou initialiser les registres au démarrage. Il fait appel à l'unité d'entrée-sortie pour communiquer avec la mémoire ou les périphériques. L'horloge doit fournir un signal régulier pour synchroniser tout le fonctionnement du processeur. Les registres sont des petites mémoires internes très rapides, pouvant être accédées facilement.

L'objectif de ce projet va être de concevoir un processeur mono-cycle composant par composant puis de les assembler pour obtenir le processeur final. Tout sera écrit en VHDL, les simulations seront faites sur ModelSim. Outre demande explicite d'un banc de test, la majorité des composants individuels seront simulés via stimulis sur ModelSim.

Unité de traitement

L'unité de traitement sert principalement à faire des opérations sur les registres d'un banc de registres ou de la mémoire des données (dans ce cas là, on les qualifiera de cases mémoire).

i – UAL ou Unité Arithmétique et Logique :

L'UAL prend en entrée 2 bus de 32 bits et fait une opération sur ces 2 derniers selon un signal de commande sur 3 bits (lui aussi étant un signal d'entrée). En sortie, il y a le résultat de l'opération faite dans un bus de 32 bits ainsi que 4 drapeaux sur 1 bit chacun indiquant certaines caractéristiques du résultat (si il est strictement inférieur à zéro, égal à zéro, présence d'une retenue ou encore overflow).

Le cours sur le processeur nous indique qu'en représentation signée le drapeau de retenue est inutile, par conséquent il sera mis à zéro par défaut dans l'écriture du composant.

On rappelle les opérations faite selon le signal de commande :

Signal de commande OP	000	001	010	011	100	101	110	111
Signaux entrée A et B	A+B	B	A-B	A	A OU B	A ET B	A XOR B	NON A

Après avoir écrit le composant, on le simule pour vérifier son bon fonctionnement. On test si les opérations s'effectuent correctement pour les différentes versions du signal de commande OP.

Opérations arithmétiques :

/alu/OP	011	000	001	010	011
/alu/A	3	3			
/alu/B	15	3			15
/alu/SUM	3	6	3	0	3
/alu/N	0				
/alu/Z	0				
/alu/C	0				
/alu/V	0				

Représentation décimale. On obtient le bon résultat dans SUM pour les différentes opérations :

Pour OP = 000 on a SUM = 6 car A + B = 3 + 3 = 6

Pour OP = 001 on a SUM = 3 car B = 3

Pour OP = 010 on a SUM = 0 car A - B = 3 - 3 = 0

Pour OP = 011 on a SUM = 3 car A = 3

Opérations logiques :

/alu/OP	100	100	101	110	111	
/alu/A	000...	00000000		FFFFFF		
/alu/B	FFF...	FFFFFF				
/alu/SUM	FFF...	FFFFFF	00000000			
/alu/N	1					
/alu/Z	0					
/alu/C	0					
/alu/V	0					

Représentation hexadécimale. Les résultats dans SUM correspondent bien aux résultats des opérations logiques sur A et B.

Pour OP = 100 on a SUM = 11...11 car A OU B = 00...00 OU 11...11 = 11...11

Pour OP = 101 on a SUM = 0...0 car A ET B = 0...0 ET 1...1 = 0...0

Pour OP = 110 on a SUM = 0...0 car A XOR B = 1...1 XOR 1...1 = 0...0

Pour OP = 111 on a SUM = 0...0 car NON A = NON 1...1 = 0...0

ii – Banc de registres :

Le banc de registres est un composant contenant 16 registres sur lesquels on peut écrire en entrée ou prendre leur contenu en sortie. On écrit dans un registre (d'adresse Rd ou RW) pendant un process synchrone selon un signal de commande et on prendra toujours en sortie les registres d'adresses entrées (d'adresse Rn et Rm ou RA et RB).

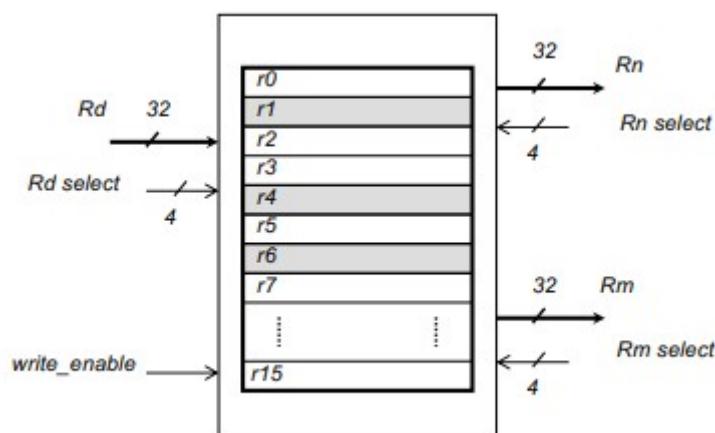


Schéma provenant du cours archi mono-cycle

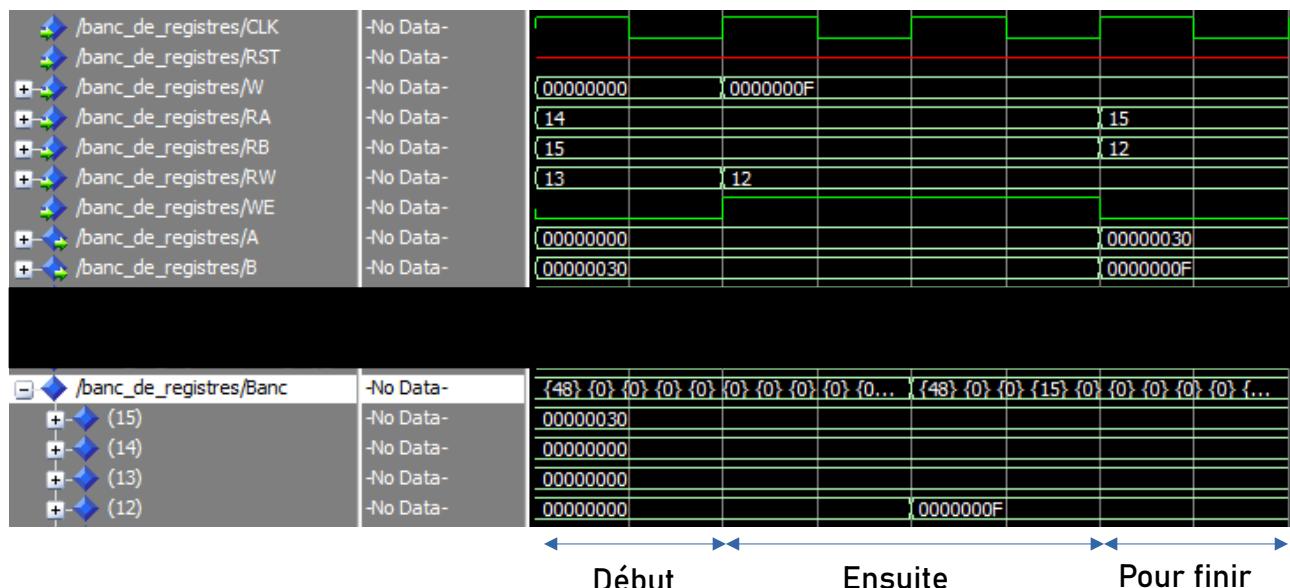
Dans la fonction donnée dans le projet pour créer le banc, tout les registres sont initialisés à 0 sauf le registre numéro 15 qui est à 0x00000030.

Ainsi lors de la simulation, on obtient :

W est le bus d'écriture, RW l'adresse du registre dans lequel on écrit.

A,B les bus de sortie, RA,RB les adresses des registres dont le contenu sera dans A et B.

WE le signal de commande d'écriture.



Au début, on n'écrit pas dans le banc de registre, A et B sortent le contenu des registre 14 et 15, d'où A=0x0 et B=0x30. Ensuite, on écrit dans le registre 12 la valeur 0x0F en passant le signal de commande WE à 1 puis on le repasse à 0 pour ne plus écrire dedans. Pour finir, on sort le contenu de ce registre dans B et dans A on sort le contenu du registre 15.

iii – Assemblage :

On va assembler les 2 composants précédents comme dans le schéma donné dans l'énoncé. Puis on va tester le bon fonctionnement de l'assemblage par un test bench.

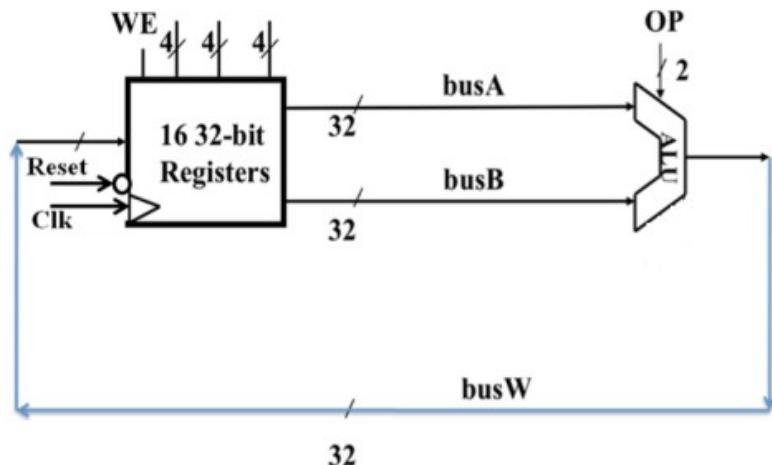


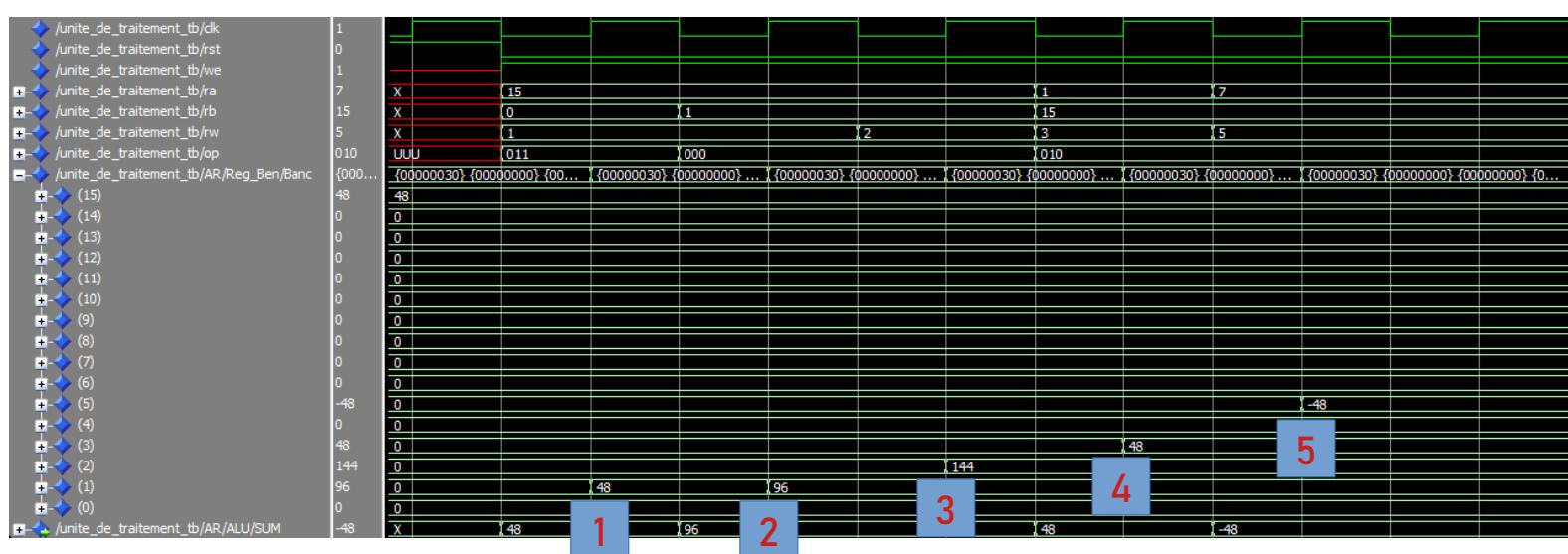
Schéma provenant de l'énoncé

Comme demandé, le test bench va réaliser les opérations suivantes :

- $R(1) = R(15)$
- $R(1) = R(1) + R(15)$
- $R(2) = R(1) + R(15)$
- $R(3) = R(1) - R(15)$
- $R(5) = R(7) - R(15)$

Pour cela, on assigne les valeurs nécessaires à WE,RA,RB,RW et OP (signaux décrits dans les composants précédents).

Une fois sur ModelSim, on obtient le graphe suivant :

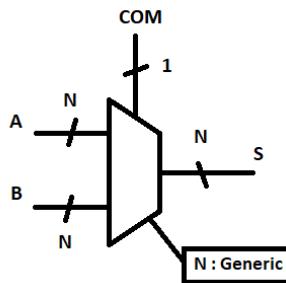


Représentation décimale. Sachant que le registre 15 est initialisé avec la valeur 0x30 (48 en décimal), on constate le bon fonctionnement des opérations.

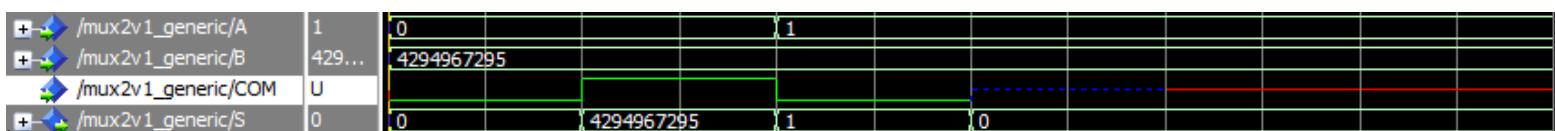
- $R(1) = R(15) = 48$ 1
- $R(1) = R(1) + R(15) = 48 + 48 = 96$ 2
- $R(2) = R(1) + R(15) = 96 + 48 = 144$ 3
- $R(3) = R(1) - R(15) = 96 - 48 = 48$ 4
- $R(5) = R(7) - R(15) = 0 - 48 = -48$ 5

iv – Multiplexeur 2 vers 1 :

Un multiplexeur 2 vers 1 est un composant dont la valeur d'une des deux entrées sera propagée sur la sortie S suivant la valeur d'un signal de commande.



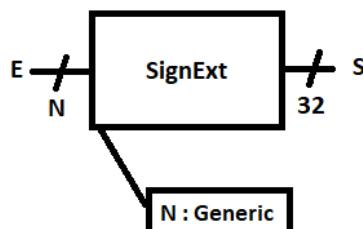
COM	0	1	Autre (-,X,U,...)
Opération	$S = A$	$S = B$	$S = 0x00..00$



On observe bien que S prend la valeur de A ou B selon la valeur de COM.

v – Extension de signe :

Ce composant va prendre en entrée un bus de taille N et en sortira un bus de 32 bits étendu selon le bit de signe. Comme le multiplexeur, c'est un composant générique. En voici le schéma bloc :

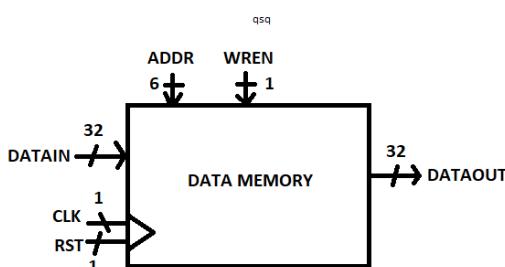


[+]	/extende_generic/E	10101011	00001111	00001000	10001000	10101011
[+]	/extende_generic/S	11111111111111...	0000000000000000000000001111	000000000000000000000000000000001000	111111111111111111111111111100001000	1111111111111111111111111111101010111

S apparaît bien comme la version étendue de E.

vi – Mémoire de données :

Ce composant est très similaire au banc de registre par conséquent on reprend le code de ce dernier en modifiant seulement quelques paramètres : un seul contenu de registre en sortie et le registre en écriture est aussi celui en lecture.



[+]	/data_memory/CLK	1	00000000	00000000	00000000	00000000
[+]	/data_memory/RST	U	00000000	00000000	00000000	00000000
[+]	/data_memory/DATAIN	0000FFFF	0000FFFF	0000FFFF	0000FFFF	0000FFFF
[+]	/data_memory/ADDR	60	60	60	61	61
[+]	/data_memory/WREN	1	00000000	00000000	00000000	00000000
[+]	/data_memory/DATAOUT	0000003C	0000003C	0000003C	0000003D	0000003D
-	/data_memory/Banc	{000000...}	{00000030}	{00000030}	{0000003E}	{0000003D}
+/-	(63)	00000030	00000030	00000030	00000030	00000030
+/-	(62)	0000003E	0000003E	0000003E	0000003E	0000003E
+/-	(61)	0000003D	0000003D	0000003D	0000003D	0000003D
+/-	(60)	0000003C	0000003C	0000003C	0000003C	0000003C
+/-	(59)	0000003B	0000003B	0000003B	0000003B	0000003B

On constate le bon fonctionnement du composant en simulation. D'abord, avec l'écriture de 0xFFFF dans le registre 60 grâce à l'activation du signal de commande d'écriture WREN ; et le contenu du registre 60 apparaît aussi en sortie. Puis avec la simple lecture du registre 61 qui contient la valeur 0x3D Pour ce composant et le futur du projet, les cases mémoires seront initialisées avec la valeur de leur adresse (ex : Mem(5) = 5, Mem(49) = 49, etc).

vii – Assemblage :

On assemble tout les précédents composants dans un seul programme pour obtenir l'unité de traitement schématisée dans l'énoncé :

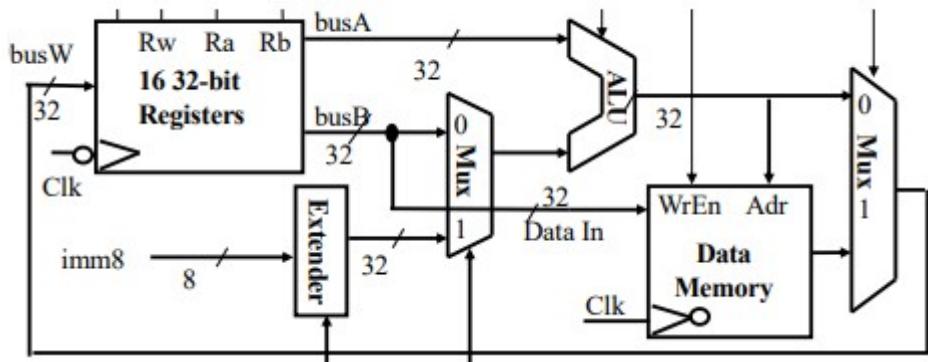


Schéma provenant du cours mono-cycle

Un test bench est demandé pour valider le bon fonctionnement des opérations suivantes :

- L'addition de 2 registres 1
- L'addition d'un registre avec une valeur immédiate 2
- La soustraction de 2 registres 3
- La soustraction d'une valeur immédiate à 1 registre 4
- La copie de la valeur d'un registre dans un autre registre 5
- L'écriture d'un registre dans un mot de la mémoire 6
- La lecture d'un mot de la mémoire dans un registre 7

Comme dit précédemment, les cases mémoires sont initialisées avec leur adresse et le registre 15 du banc de registre contient la valeur 48

$$1 \rightarrow R(14) = R(15) + R(15) = 48 + 48 = 96$$

$$2 \rightarrow R(13) = R(14) + 3 = 96 + 3 = 99$$

$$3 \rightarrow R(12) = R(14) - R(15) = 96 - 48 = 48$$

$$4 \rightarrow R(15) = R(15) - 8 = 48 - 8 = 40$$

$$5 \rightarrow R(15) = R(13) = 99$$

$$6 \rightarrow \text{Mem}(35) = R(13) = 99 \quad (\text{On prend les 6 bits de poids faible de } R(15) \text{ ce qui donne 35})$$

$$7 \rightarrow R(15) = \text{Mem}(0) = 0$$

Signaux principaux ainsi que quelques registres du banc

..._tb/dk	-No ...														
..._tb/rst	-No ...														
..._tb/ra	-No ...		15	14		15		13							0
..._tb/rb	-No ...		15	0		15	0	13							0
..._tb/rw	-No ...		14	13		12		15							15
..._tb/we	-No ...														
..._tb/op	-No ...		000		010			001				011			
..._tb/jmn	-No ...		0	3	0	8		0							
.../com1	-No ...														
.../com2	-No ...														
...b/wren	-No ...														
...tb/flag	-No ...		0000			0000						0100			
.../Banc	-No ...		{00000030}	{00000...}	{00000030}	{00000...}	{00000030}	{00000...}	{00000028}	{00000...}	{00000063}	{00000060}	{00000063}	{00000030}	{00000000}
◆ (15)	-No ...		48					40		99					0
◆ (14)	-No ...		0	96											
◆ (13)	-No ...		0		99										
◆ (12)	-No ...		0			48									
◆ (11)	-No ...		0				4			5					7
		1	2	3											

Quelques registres de la mémoire de données

(37)	-No ...	00000025													6
(36)	-No ...	00000024													
(35)	-No ...	35													99
(34)	-No ...	00000022													
(33)	-No ...	00000021													

Unité de gestion des instructions

L'unité de gestion des instructions sert à envoyer les instructions que le processeur va exécuter. Ces dernières sont contenues dans la mémoire d'instructions et elles sont choisies grâce à un compteur qui est incrémenté de 1 à chaque front montant de l'horloge, mais il se peut que pour certaines instructions le compteur soit additionné ou soustrait d'une valeur immédiate différente de +1.

i – Mémoire d'instructions :

Le composant est directement donné via moodle et contient une fonction de création de liste de registre similaire à celle du banc de registre et de la mémoire de données. Une suite d'instructions y est déjà initialisée :

0x0 _main : MOV R1,#0x20 ;	--R1 <= 0x20
0x1 MOV R2,#0 ;	--R2 <= 0
0x2 _loop : LDR R0,0(R1) ;	--R0 <= DATA_MEM[R1]
0x3 ADD R2,R2,R0 ;	--R2 <= R2 + R0
0x4 ADD R1,R1,#1 ;	--R1 <= R1 + 1
0x5 CMP R1,0x2A ;	-- R1 - 0x2A, mise à jour de N
0x6 BLT loop ;	--branchement à loop si R1 inf à 0x2A
0x7 _end : STR R2,0(R1) ;	--DATA_MEM[R1] <= R2
0x8 BAL main ;	--branchement à _main

Ces instructions reviennent à faire l'addition de 10 cases mémoires (Mem(32) jusqu'à Mem(41)) puis de stocker le résultat dans Mem(42) et ensuite la séquence d'instructions recommence depuis le début. Dans notre cas, le résultat stocké est 365 qui est la somme des i pour i allant de 32 à 41. Le résultat varie selon l'initialisation des cases mémoires.

ii – Registre 32 bits PC :

Ce registre va « laisser passer » la donnée d'entrée contrairement à un registre classique. Il n'y a pas de signal d'écriture dans le registre. On écrit le composant à part plutôt que de faire toutes les étapes dans le fichier d'assemblage.

iii – Unité de mise à jour du compteur PC :

Le compteur PC peut soit être incrémenté de 1 soit incrémenté/soustrait d'une valeur nommée offset qu'on retrouve dans les instructions de branchement. Ainsi, la présence d'un multiplexeur pour choisir l'un de ces deux cas est indispensable. L'offset étant sur 24 bits, il est nécessaire de réutiliser le composant d'extension de signe avant de l'additionner à PC et ensuite de passer par le multiplexeur.

iv – Assemblage :

On assemble tout les précédents composants dans un seul programme pour obtenir l'unité de gestion des instructions schématisée dans l'énoncé :

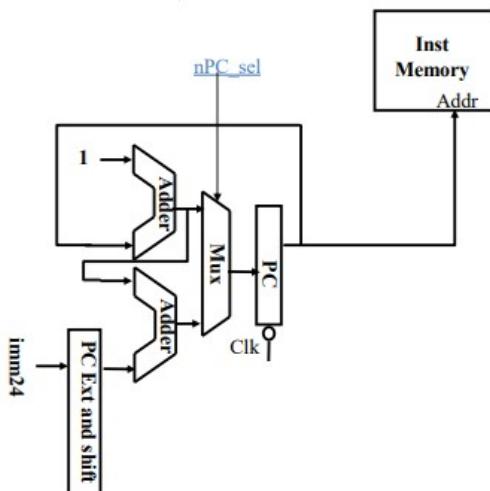
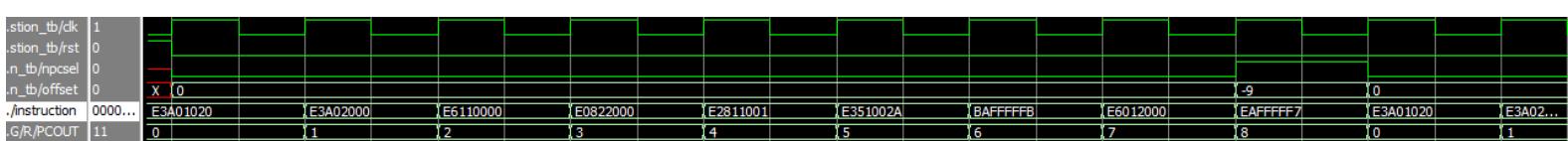


Schéma provenant du cours archi mono-cycle

On laisse PC s'incrémenter de 1 pendant les premières instructions, on constate le bon fonctionnement de l'unité avec les différentes instructions en sortie. Quand on ajoute un offset, on constate bien qu'on renvoie les instructions précédemment envoyées.



Unité de contrôle

L'unité de contrôle est composé du décodeur d'instructions ainsi que d'un registre utilisé pour stocker les flag de l'UAL.

i – Registres 32 bits PSR et AFFICHAGE :

Pour stocker l'état du processeur ainsi que la valeur à afficher sur les LEDs, on crée deux registres respectivement Reg_PSR et Reg_Aff. Le premier prendra les flag de l'UAL et les enverra vers le décodeur, le second prendra la valeur du registre Rm du banc de registres seulement lorsqu'une instruction de type store sera traitée

ii – Décodeur d'instructions :

Le décodeur prend en entrée l'instruction envoyée par l'unité précédemment écrite et selon cette dernière envoie différentes valeurs à différents signaux de contrôle dans l'unité de traitement et l'unité de gestion des instructions.

Les commandes à générer concernent :

- La commande nPCsel du multiplexeur situé en amont du registre PC
- La commande de chargement du registre PSR (PSRen)
- Le write enable du banc de registres (RegWr)
- La commande RegSel du multiplexeur situé en amont du banc pour RB
- La commande OP de l'UAL (ALUCtrl)
- La commande ALUSrc du multiplexeur situé en amont de l'entrée B de l'UAL
- La commande WrSrc du multiplexeur situé en sortie de l'UAL et de la mémoire de données
- Le Write Enable de la mémoire de données (MemWr)
- La commande de chargement du registre Aff(RegAff)

Voici le tableau de valeurs des commandes :

Instruction	nPCsel	RegWr	ALUSrc	ALUCtr	PSRen	MemWr	WrSrc	RegSel	RegAff
ADDi	0	1	1	000	1	0	0	0	0
ADDR	0	1	0	000	1	0	0	0	0
BAL	1	0	0	000	0	0	0	0	0
BLT	1	0	0	000	0	0	0	0	0
CMP	0	0	1	010	0	0	0	0	0
LDR	0	1	0	011	0	0	1	0	0
MOV	0	1	1	001	0	0	0	0	0
STR	0	0	0	011	0	1	0	1	1

Assemblage complet du processeur

Avant de tout assembler, on ajoute un multiplexeur en amont de l'entrée du banc de registres pour choisir l'adresse de RB.

i - Assemblage :

On assemble les différentes unités et composants de sorte à obtenir un processeur comme sur le schéma suivant :

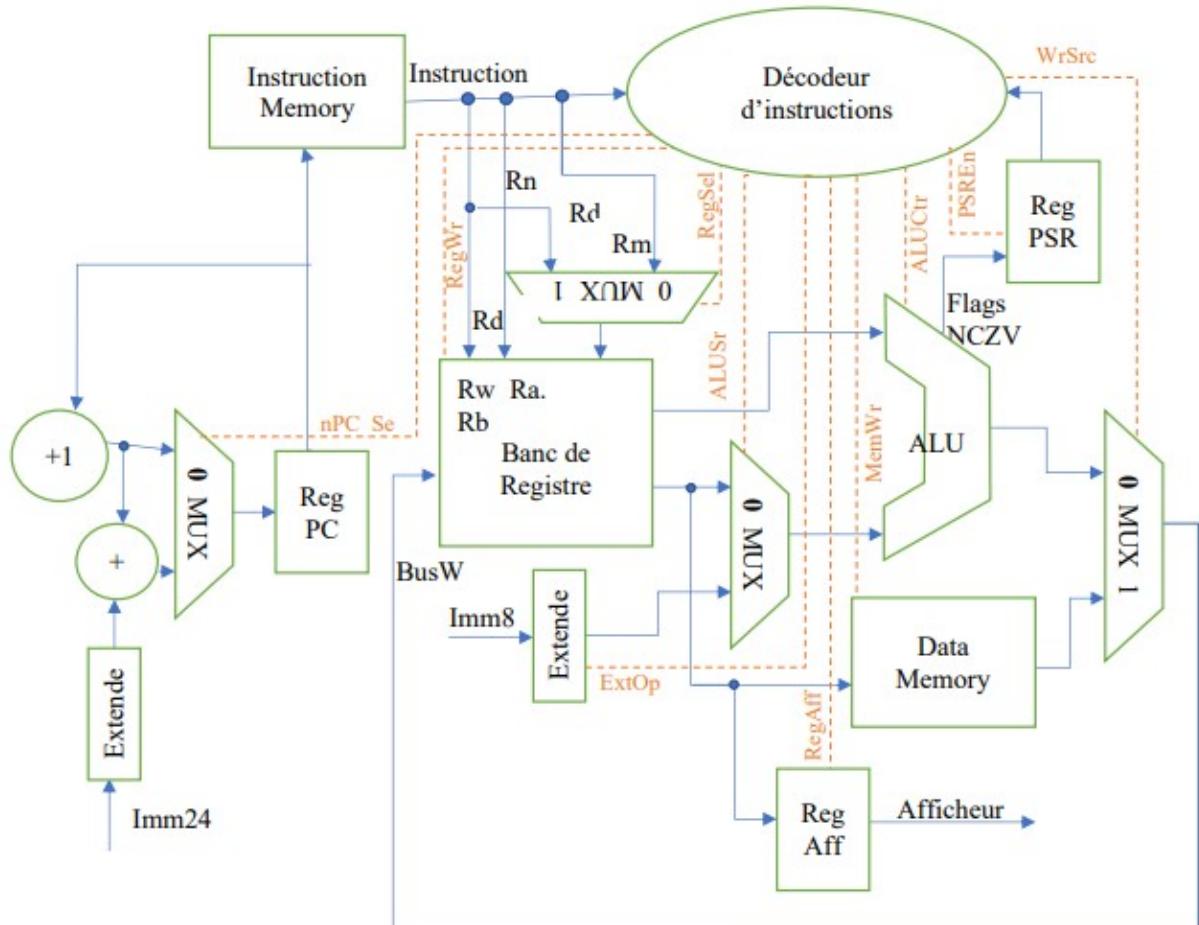


Schéma provenant de l'énoncé

ii - Test de fonctionnement :

Etant donné qu'il n'y qu'un signal de sortie pour tout le processeur, il suffit de vérifier la valeur du signal de sortie de **Reg_Aff** pour vérifier le bon fonctionnement de ce dernier. D'après la partie sur la mémoire d'instructions, le signal de sortie devrait afficher le contenu de **Mem(42)** qui est égal à : $\text{Mem}(42) = 32 + 33 + \dots + 41 = 365 = 0x16D$

Après ce calcul, la séquence d'instructions redémarre et ainsi refait la même chose. Le signal de sortie restera donc toujours le même après la première assignation de 365

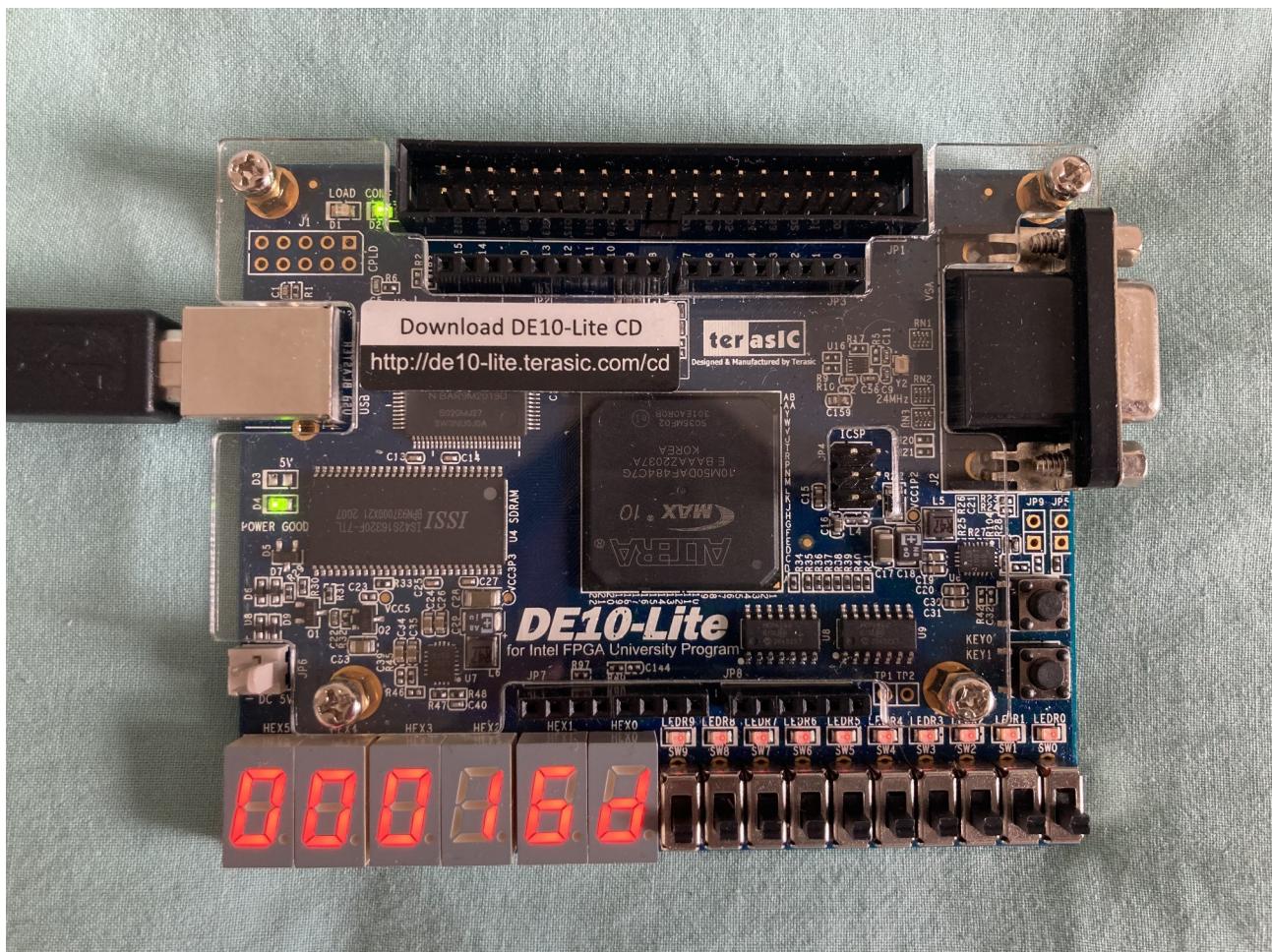
P/instruction	E35...	E0822000	E2811001	E351002A	BAFFFFFB	E6012000	EAFFFFF7	E3A01020	E3A02000	E6110000	E0822000
str_courante	CMP	ADD	CMP	BLT	STR	BAL	MOV		LDR	ADD	
/UT/BR/Banc	{000...	000...	{00000030}	{00000030}	{00000000}	{00000000}	{00000000}	{00000000}	{00000000}	{00000000}	{00000000}
15)	000...	00000030									
14)	000...	00000000									
13)	000...	00000000									
12)	000...	00000000									
11)	000...	00000000									
10)	000...	00000000									
9)	000...	00000000									
8)	000...	00000000									
7)	000...	00000000									
6)	000...	00000000									
5)	000...	00000000									
4)	000...	00000000									
3)	000...	00000000									
2)	000...	00000144	0000016D							00000000	
1)	000...	0000029		0000002A					00000020		
0)	000...	0000029								00000020	
b/P/Afficheur	365	0					365				

On observe ainsi le bon fonctionnement du processeur :

Implémentation sur carte FPGA

Pour afficher le signal de sortie du processeur, on va avoir besoin d'afficheurs sept segments qui afficheront ce signal en représentation hexadécimale. On récupère le code de l'afficheur donné dans le cours et on y rajoute les cas pour des valeurs supérieures à 9 pour une représentation hexadécimale. Ensuite, on assemble le processeur avec 6 afficheurs 7 segments.

Après implémentation sur carte via Quartus, on obtient bien la valeur hexadécimale correspondant à 365 qui est 16D :

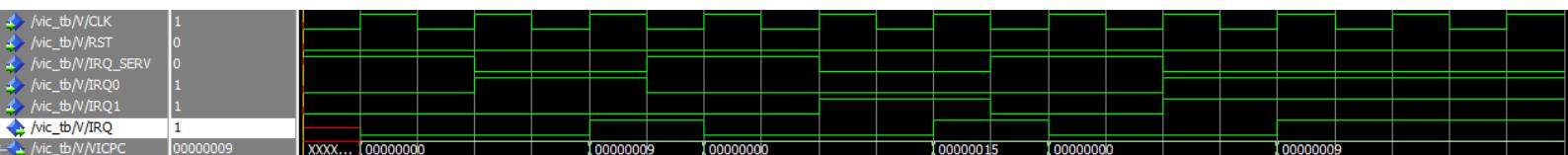


Gestion des interruptions externes

Une interruption est une suspension temporaire de l'exécution d'un programme informatique par le microprocesseur afin d'exécuter un programme prioritaire (appelé service d'interruption). On va rajouter un composant pour gérer ces interruptions qui s'appelle un contrôleur d'interruptions vectorisé ou VIC. Une interruption étant une instruction, il faudra aussi modifier le décodeur d'instructions puis toutes les unités/composants reliées à ces composants. La mémoire des instructions a été modifiée et contient désormais des séquences d'instructions liées aux interruptions.

i – VIC :

Le composant prend en entrée deux requêtes d'interruptions externes différentes : IRQ0 et IRQ1, ainsi que IRQ_SERV qui est l'acquittement de l'interruption. En sortie, il y a la requête d'interruption à envoyer vers l'unité de gestion des instructions : IRQ qui indique donc si une requête d'interruption a été émise. Ainsi que l'adresse du début du sous-programme d'interruption qui sera donc égale à 0 ou 9 ou 21 (correspondant aux débuts des différentes séquences d'instructions). Grâce au test bench, on constate le bon fonctionnement du composant pour les différents cas :



On commence par acquitter pour que tout soit à 0, puis on envoie l'IRQ0 qui va mettre IRQ à 1 et VICPC à 0x09. Tout repasse à 0 car on a renvoyé un acquittement. On réitère le processus pour IRQ1 puis on test le composant pour quand IRQ0 et IRQ1 sont envoyé simultanément, et on remarque bien que c'est IRQ0 qui a la priorité avec VICPC = 0x09.

ii – Explication des instructions :

Une nouvelle mémoire d'instructions est donnée dans l'énoncé : la séquence d'instructions principale (hors interruption) est quasiment identique à la précédente à la seule différence qu'on fait la somme des cases mémoires 16 à 25 ce qui est donc égale à 205 ou 0x0CD en hexadécimal. Puis on stockera ce résultat dans Mem(26).

L'interruption 0 a pour instructions les suivantes :

ISR 0 : interruption 0

sauvegarde du contexte

STR R1,0(R15) ;	--MEM[R15] <= R1
ADD R15,R15,1 ;	--R15 <= R15 + 1
STR R3,0(R15) ;	--MEM[R15] <= R3

traitement

MOV R3,0x10 ;	--R3 <= 0x10
LDR R1,0(R3) ;	--R1 <= MEM[R3]
ADD R1,R1,1 ;	--R1 <= R1 + 1
STR R1,0(R3) ;	--MEM[R3] <= R1

restauration du contexte

LDR R3,0(R15) ;	--R3 <= MEM[R15]
ADD R15,R15,-1 ;	--R15 <= R15 - 1
LDR R1,0(R15) ;	--R1 <= MEM[R15]
BX ;	-- instruction de fin d'interruption

Etant donné qu'on utilise les registres 1 et 3 du Banc dans la partie traitement, on sauvegarde leur valeur actuelle dans sauvegarde du contexte pour ensuite les « reprendre » dans restauration du contexte avant de finir la séquence d'interruption. Dans la partie traitement, en simplifiant en une ligne, on a $\text{Mem}(16) = \text{Mem}(16) + 1$; or dans la séquence principale on additionne cette case mémoire aux autres donc l'interruption 0 permet d'ajouter 1 à la somme actuelle des cases mémoires. Exemple : en faisant 2 requêtes d'interruptions 0 à la suite, la séquence principale stockera dans $\text{Mem}(26)$ la valeur 207 (0xCF).

L'interruption 1 a pour instructions les suivantes :

ISR 1 : interruption 1

sauvegarde du contexte

STR R4,0(R15) ;	--MEM[R15] <= R4
ADD R15,R15,1 ;	--R15 <= R15 + 1
STR R5,0(R15) ;	--MEM[R15] <= R5

traitement

MOV R5,0x10 ;	--R5 <= 0x10
LDR R4,0(R5) ;	--R4 <= MEM[R5]

ADD R4,R4,2 ;	--R4 <= R4 + 2
STR R4,0(R5) ;	--MEM[R5] <= R4

restauration du contexte

LDR R5,0(R15) ;	--R5 <= MEM[R15]
ADD R15,R15,-1 ;	--R15 <= R15 - 1
LDR R4,0(R15) ;	--R4 <= MEM[R15]
BX ;	-- instruction de fin d'interruption

On remarque que cette séquence est quasiment identique à celle de l'interruption 0, les différences étant que les registres 4 et 5 sont utilisés pour faire le traitement et aussi que l'on fait l'opération suivante : $\text{Mem}(16) = \text{Mem}(16) + 2$.

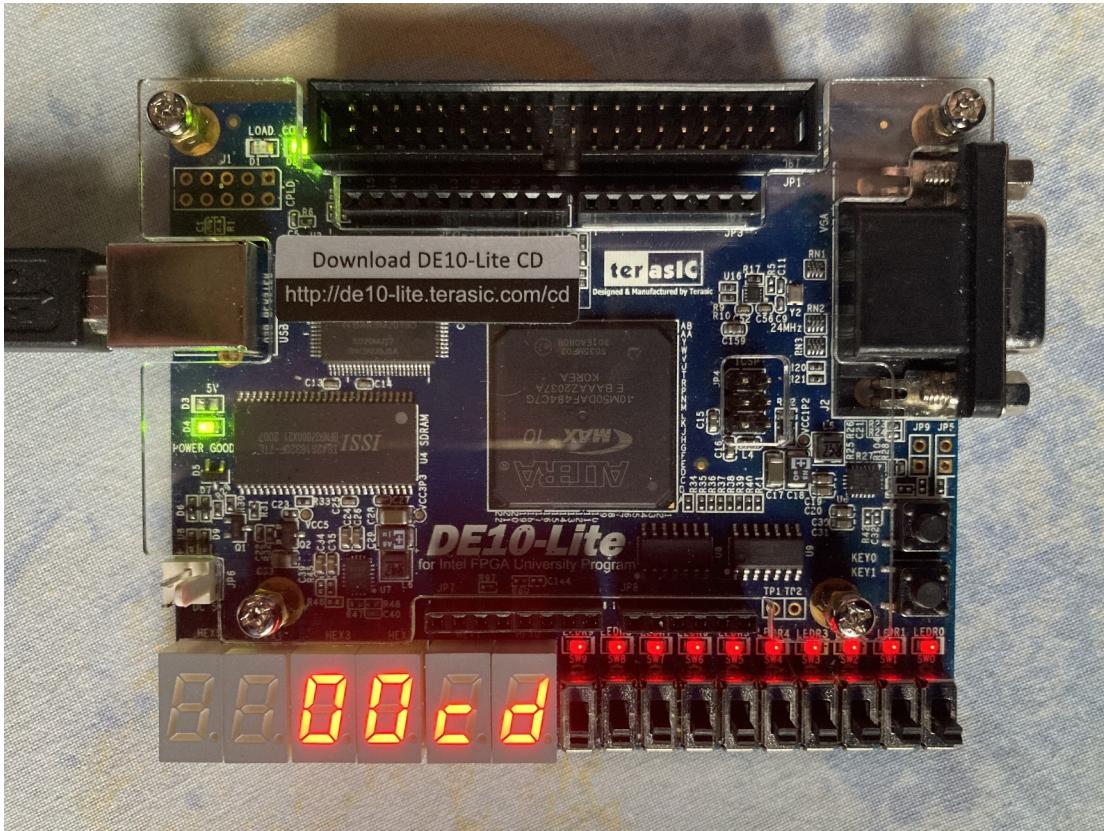
iii – Modification et assemblage dans le processeur :

Comme dit précédemment, on modifie le décodeur d'instructions pour qu'il puisse traiter les instructions BX d'interruption et qu'il renvoie un nouveau signal associé à ces dernières : irq_end qui signifie que la séquence d'instructions d'interruption est terminée. L'unité de gestion des instructions va aussi être modifiée pour pouvoir sauvegarder l'état du compteur PC lorsqu'une requête d'interruption est émise et envoyer VICPC vers la mémoire d'instructions. Pour tout cela on rajoute un nouveau registre : le registre LR. C'est lui qui va sauvegarder le compteur PC.

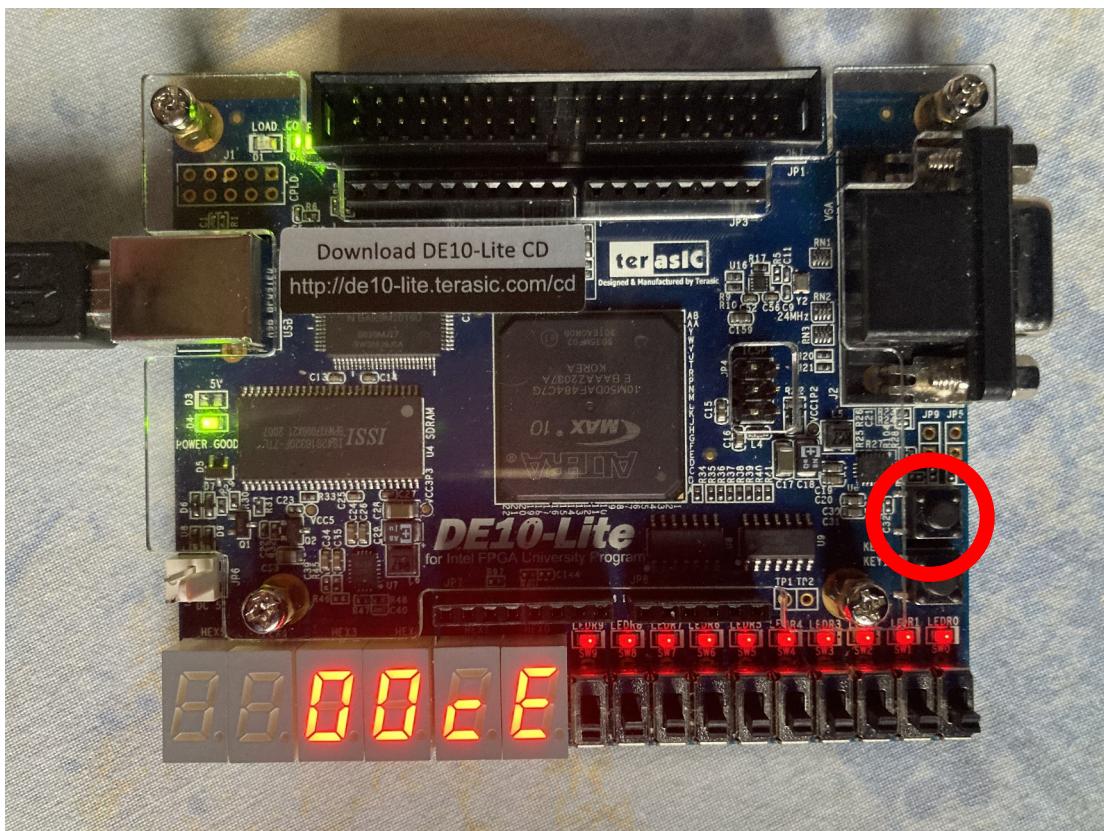
Après simulation via test bench et sur carte, j'ai remarqué qu'il arrivait très souvent qu'en envoyant une requête d'interruption, la valeur affichée ne correspondait pas à ce qu'elle devrait être. Ex : Passer de 0xCD à 0x128 en envoyant une interruption 0 alors qu'on devrait passer à 0xCE. Cela est probablement dû au fait qu'à la fin de l'interruption la reprise des instructions principale selon l'instruction sauvegardée se fait mal et une coquille s'insert. N'ayant pas réussi à résoudre cette coquille, j'ai trouvé une alternative : à chaque fin d'interruption, la séquence principale est exécutée depuis le début. Pour faire cela, j'ai mis le signal reset du registre PC comme un OU logique entre le signal reset classique et le signal IRQ_END qui signifie la fin de l'interruption afin que le bus de sortie soit égal à 0x0..0 dans ce dernier cas. En conséquence de cela je n'ai plus besoin du registre LR et je l'ai donc supprimé.

Une fois tout cela fait, on rajoute dans le top_level la possibilité d'envoyer des interruptions via les boutons KEY 0 et 1 et le reset est sur le

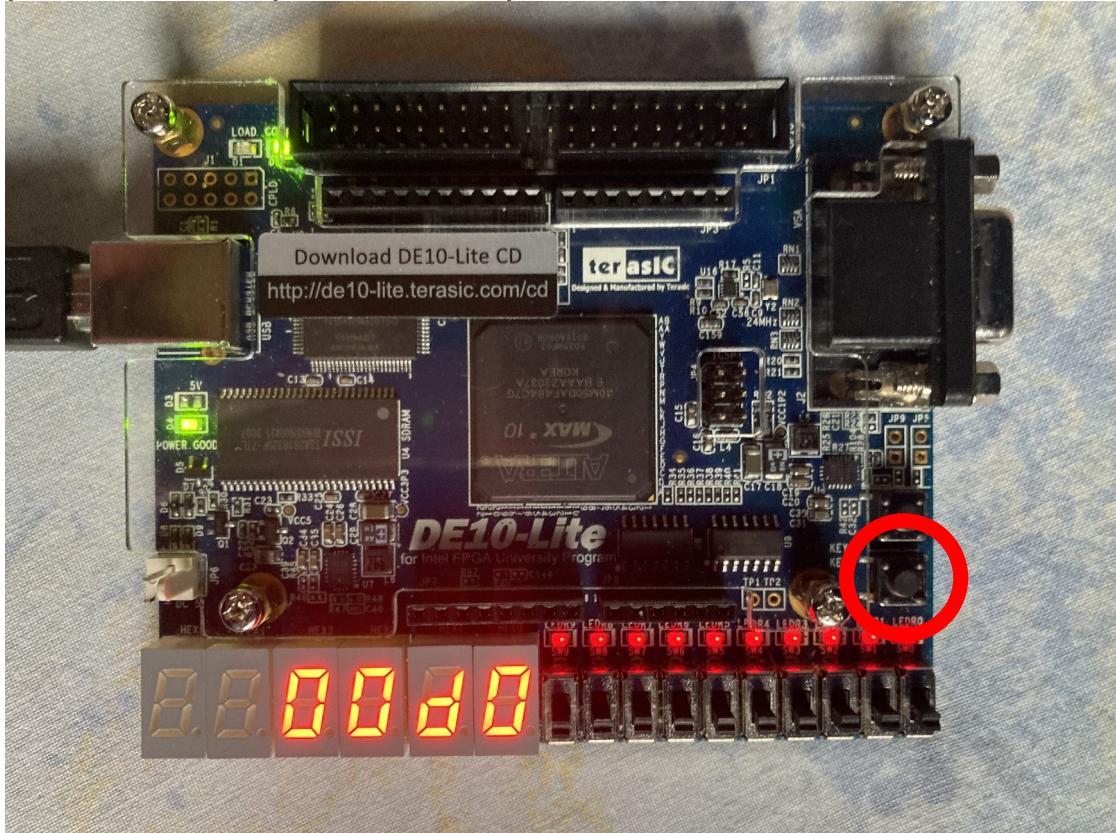
SWITCH 9. Les cases mémoires sont toujours initialisées de la même manière qu'avant. Une fois implémenté sur la carte, on obtient donc ceci : Le résultat de la séquence d'instructions classique :



Lorsqu'on fait une requête d'interruption 0 ($0xCD + 0x01 = 0xCE$) :



Lorsqu'on fait une requête d'interruption 1 ($0xCE + 0x02 = 0xD0$) :



Périphérique UART

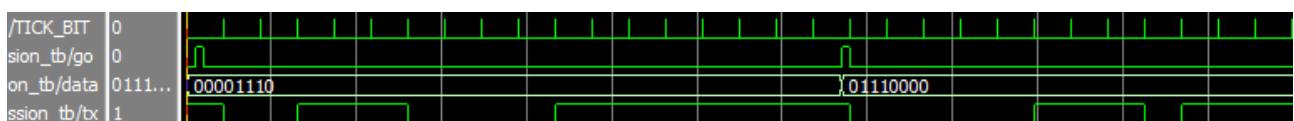
Un UART, pour Universal Asynchronous Receiver Transmitter, est un émetteur-récepteur asynchrone universel. c'est le composant utilisé pour faire la liaison entre l'ordinateur et le port série. L'ordinateur envoie les données en parallèle (autant de fils que de bits de données). Il faut donc transformer ces données pour les faire passer à travers une liaison série qui utilise un seul fil pour faire passer tous les bits de données.

i – Emission :

Le périphérique d'émission de l'UART prend en entrée un octet (DATA) et un signal de commande pour émettre cet octet ou non. Il comporte deux composants :

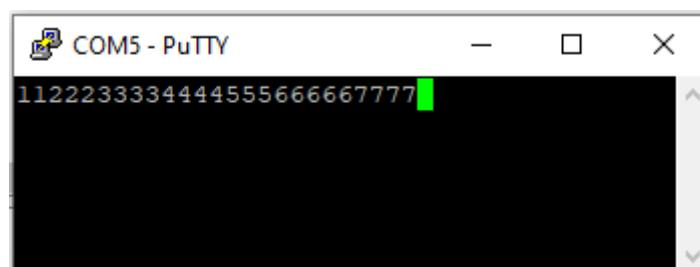
- Un composant nommé FDIV qui envoie sur front montant un signal d'horloge respectant le baud rate de 115200 et la CLOCK_50 de la carte FPGA de fréquence 50M Hz. Donc il faut compter jusqu'à $50\ 000\ 000/115200 -1 = 433$ fronts montants de la CLOCK_50 pour que le signal envoyé par le FDIV passe de montant à descendant ou inversement.
- Un autre composant qui est une machine à état permettant d'envoyer l'octet contenu dans DATA si le signal de commande GO est activé et selon l'horloge générée par FDIV. L'octet est envoyé bit par bit via le signal TX avec un bit de start à 0 rajouté ainsi qu'un bit de stop à 1.

Après avoir écrit ce périphérique, on le simule sur ModelSim :

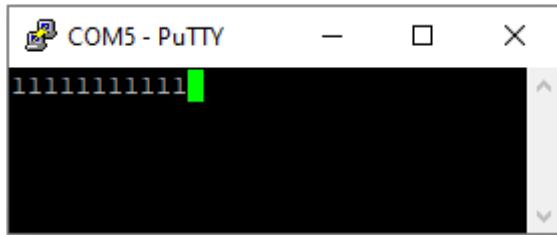


On constate que TX émet bien les octets que l'on veut en comparant les formes qu'il fait avec les chaînes de bits.

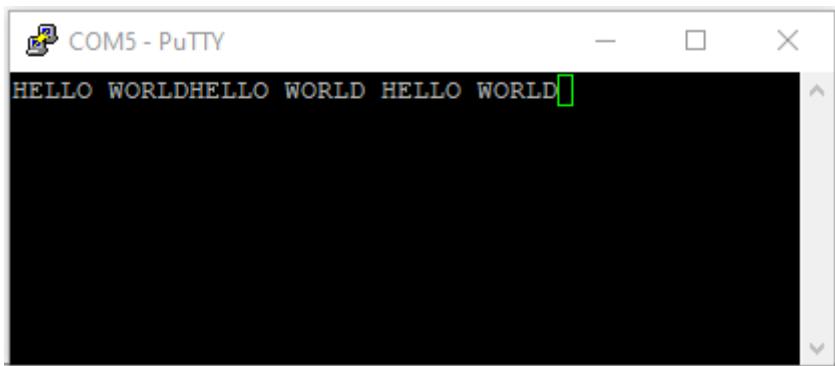
On test ensuite ce périphérique grâce à Putty en connectant un convertisseur série/USB à la carte FPGA et au PC. Avec les bits de l'octet reliés aux switchs, on arrive à écrire différents caractères :



Suite à ce premier test on rajoute un registre 32 bits dans le périphérique actif grâce à GO. Puis on incorpore le périphérique au processeur. Le signal de commande GO ne sera plus gérer par moi mais par le décodeur d'instruction qui l'activera uniquement pour les instructions STR d'adresse 0x40. Ensuite, on veut pouvoir écrire 1 sur Putty à chaque fois que l'on rentre dans l'irq1 : pour cela, on rajoute au VIC un signal de sortie DATA qui sera l'octet à émettre grâce au périphérique UART. DATA prend donc la valeur 0x31 à chaque fois qu'une requête d'interruption 1 est détectée. On test à nouveau sur Putty et on arrive bien à écrire 1 à chaque fois que l'on envoie une IRQ1 :



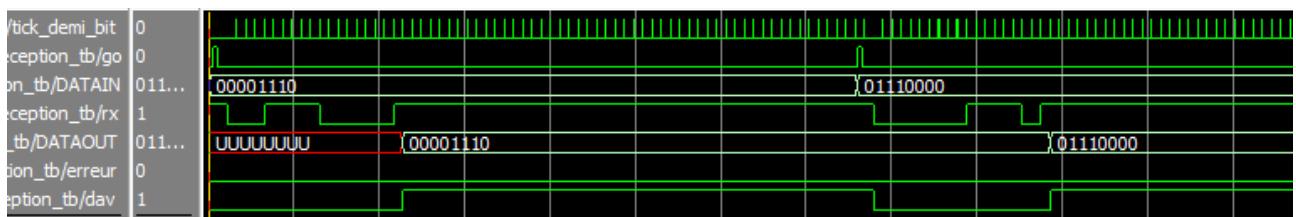
Dorénavant, on cherche à envoyer une chaîne de caractères : « HELLO WORLD ». Pour cela, à chaque caractère émis, on va envoyer un nouveau type de requête d'interruption via l'UART : txlrq. Le VIC est modifié pour recevoir et traiter cette nouvelle interruption. Afin d'envoyer une chaîne de caractères, on modifie d'abord la mémoire d'instructions pour que l'interruption 1 ne contienne qu'une instruction STR (d'adresse 0x40) et une instruction BX. Le VIC est modifié encore une fois pour qu'il compte chaque caractère émis et modifie DATA en conséquence. Une fois la chaîne complètement envoyée, on remet VICPC à 0x00000000. Une fois testé sur Putty, on obtient le résultat escompté :



Si on fait un reset après avoir écrit la chaîne, la prochaine chaîne s'écrira à la suite sans espace de séparation (1ère et 2ème chaîne). Si on envoie irq1 après irq1 sans reset, alors les chaînes seront espacées (2ème et 3ème chaîne).

ii – Réception:

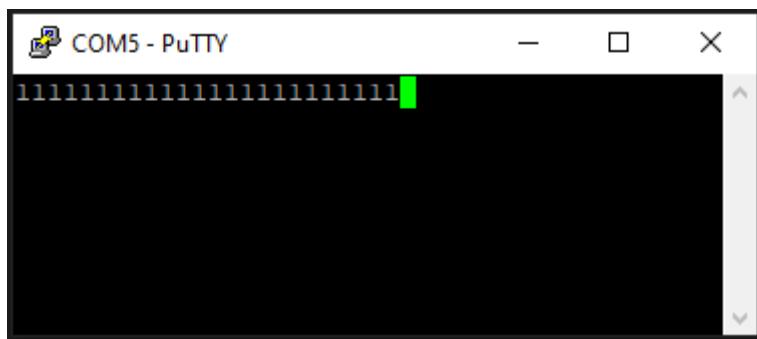
Le périphérique de réception de l'UART prend en entrée un signal RX dont le comportement est similaire au signal TX du périphérique d'émission. En sortie, il y a donc l'octet reçu ainsi qu'un signal indiquant la présence d'une erreur dans la réception et un autre signal indiquant la bonne réception de l'octet. Le périphérique comporte un FDIV ayant un signal d'horloge deux fois plus rapide que celui du FDIV du périphérique d'émission. La machine à état quand à elle écrit l'octet selon le comportement de RX. On simule ce composant en utilisant le périphérique d'émission pour envoyer le signal RX car le faire manuellement dans le test bench est trop compliqué :



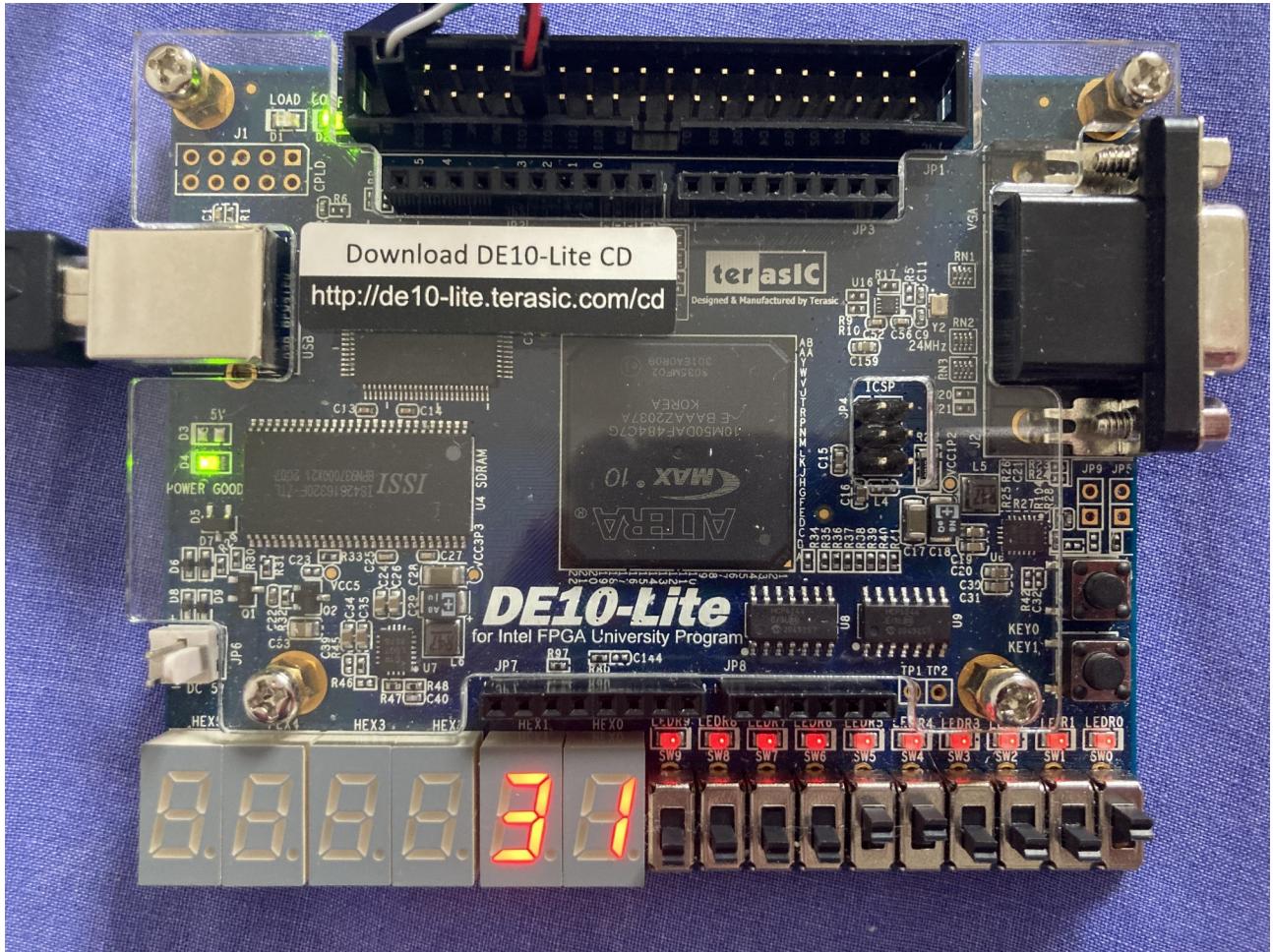
DATAIN est l'octet que l'on envoie dans UART_Emission et on constate la bonne réception de ce dernier avec le signal DATAOUT. Le signal d'erreur reste bas et le signal d'acquisition reste haut après réception des octets.

Pour simuler cela sur la carte FPGA on refait le même fonctionnement que pour le test bench et pour vérifier que cela fonctionne, on écrira sur les LEDs le code hexadécimal du caractère émis grâce à au périphérique d'émission.

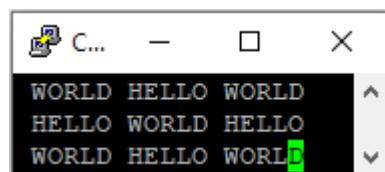
En envoyant le caractère '1' de code ASCII 0x31 :



On obtient ceci :



On suit le même chemin que pour la première partie : on incorpore le périphérique de réception au processeur et on rajoute un registre 32 bits dans l'UART_Réception pour stocker l'octet, le signal de commande étant le signal DAV. On réutilise la même technique pour vérifier si ça fonctionne en implémentant sur la carte : l'affichage du code ASCII du caractère émis sur les LEDs. Le VIC n'ayant pas changé, en envoyant l'irq1 il y a toujours « HELLO WORLD » qui est émis. Donc, les LEDs afficheront 0x44 soit le code ASCII du caractère D.



On obtient ceci sur la carte :

