

# Rapport de Projet - Processeur mono-cycle en VHDL

## Version complète

Polytech Sorbonne

Formation Ingénieur Electronique - Informatique 3ème année

**Ugo LUCCHI**

Encadrants : Yann Douze, Mouad Abrini

Mai - Juin 2024



[https://github.com/LuckyMoy/Processeur\\_Mono-Cycle](https://github.com/LuckyMoy/Processeur_Mono-Cycle)  
(en privé avant le rendu)



Malgré la longueur de ce rapport, un effort a été fait pour le maintenir le plus concis possible tout en étant clair et exhaustif sur le travail important fourni pour ce projet. Si celui-ci venait à être vraiment trop long, Une version condensée et donc potentiellement incomplète est fournie dans le répertoire "autre" de l'archive.

## Introduction

Ce projet a pour objectif de concevoir et de simuler le cœur d'un processeur monocycle 32 bits de type **ARM7 TDMI** en utilisant le langage de description matérielle VHDL. L'essentiel du travail consiste à assembler des composants de base tels que des registres, des multiplexeurs, des bancs de mémoire, et une unité arithmétique et logique (UAL) pour créer les différents blocs du processeur. Ces blocs incluent l'unité de traitement, l'unité de gestion des instructions, et l'unité de contrôle. Le bon fonctionnement du processeur est vérifié par la simulation de l'exécution d'un programme test simple, puis testé sur un FPGA.

Chaque composant est conçu en VHDL comportemental RTL et simulé à l'aide de ModelSim. Des bancs de test spécifiques ont été développés pour valider chaque module. Le projet permet de mettre en pratique des compétences cruciales en conception de circuits numériques, en rédaction de rapports techniques, et en validation de composants IP (Intellectual Property) numériques. Ce document présente les différentes étapes de conception, de simulation et de validation de chaque composant, ainsi que leur intégration finale dans un processeur monocycle fonctionnel.

## Méthodologie

Lors de ce projet, j'ai porté une attention particulière à rendre mon travail propre et clair. J'ai ainsi réparti le plus possible le projet en petits composants en adoptant une approche structurelle.

Chaque partie du sujet correspond à un composant décrit dont la source est dans le répertoire /src de l'archive rendue avec ce projet.

Chaque composant décrit dans ce projet (hormis certains composants des parties 6 et 7 qui ne sont que des modifications de composants déjà testés) a été vérifié et validé à l'aide d'un test bench auto-testant. Chaque fichier Testbench est nommé par le nom du composant suivi de "*\_TB.vhd*". A chaque fichier est associé un script nommé "*simu\_[nom du composant].do*" destiné à permettre leur exécution dans ModelSim. Ces fichiers sont dans le répertoire /simu de l'archive.

Les chronogrammes des simulations sont présents dans ce rapport à la partie correspondant au composant.

La **liste des sources en annexe** fournit plus d'informations sur les fichiers présents dans l'archive et renseigne la page où elle est présentée.

Ce rapport reprend la structure du sujet du projet en ajoutant quelques sous-parties. Pour chaque composant, une présentation de la solution proposée est fournie ainsi que la description des tests mis en œuvre dans le Test Bench suivie de la capture du chronogramme associé.

Concernant la plagiat, j'ai été très rigoureux lors de ce projet et de l'élaboration de ce rapport : Aucune source ou simulation n'a été copiée ou inspirée du travail de quelqu'un d'autre que ce soit un camarade de classe, des projets d'autres années, d'internet ou d'une intelligence artificielle.

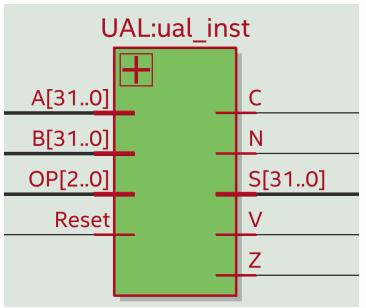
# Sommaire

Introduction.....	1
Méthodologie.....	2
<b>Partie 1 - Unité de Traitement.....</b>	<b>4</b>
1.1.1 : Unité Arithmétique et Logique.....	4
1.1.1 : Banc de registres.....	5
1.2.2/3 : Assemblage UAL et Banc de registres.....	7
1.2 : Mux 2 vers 1.....	8
1.2 : Extension de signe.....	9
1.2 : Mémoire de données.....	10
1.3 Assemblage de l'Unité de Traitement.....	11
<b>Partie 2 - Unité de gestion des instructions.....</b>	<b>13</b>
Instruction Memory.....	13
Registre PC.....	14
PC Update Unit.....	14
Assemblage Unité d'instruction.....	16
<b>Partie 3 - Unité de Contrôle.....</b>	<b>18</b>
3.1 : Tableau "valeurs des commandes" complété.....	18
3.2 : Registre PSR.....	18
3.2 : Décodeur d'Instructions.....	19
3.3 : Assemblage de l'Unité de Contrôle.....	21
<b>Partie 4 - Assemblage et validation du processeur.....</b>	<b>23</b>
4.1 : Modification de l'Unité de Traitement.....	23
4.2/3 : Assemblage du Processeur.....	24
<b>Partie 5 - Test du Processeur sur carte FPGA.....</b>	<b>26</b>
5.1 : Top Level & Projet Quartus.....	26
5.2 : Test sur Carte.....	27
<b>Partie 6 - Gestion des interruptions externes.....</b>	<b>28</b>
6.1 : Vectorized Interruption Controller.....	28
6.2 : Modification du décodeur d'instruction.....	29
6.3 : Modification de l'Unité de gestion des Instructions.....	30
6.4 : Modification du processeur & simulation.....	32
6.4 : Test du VIC sur FPGA.....	33
<b>Partie 7.A - Périphérique UART TX.....</b>	<b>35</b>
Composants Uart.....	35
Registre UART_Conf.....	35
Modification de l'UT : Ajout d'une sortie UART_TX.....	36
Modification de l'Instruction Memory : envoi d'un caractère sur IRQ.....	38
Test sur carte intermédiaire.....	38
Modification du VIC.....	39
Modification de l'Instruction Memory : envoi d'une chaîne de caractères.....	40
Test sur carte complet.....	41
<b>Partie 7.B - Périphérique UART RX.....</b>	<b>44</b>
Modification de l'UT : Ajout d'une entrée UART_RX.....	44
Modification du VIC.....	45
Modification de l'Instruction Memory : Réception d'un caractère.....	46
Test sur carte.....	47
<b>Conclusion.....</b>	<b>49</b>
<b>Annexe : Table des fichiers.....</b>	<b>50</b>

# Partie 1 - Unité de Traitement

## 1.1.1 : Unité Arithmétique et Logique

### Description

Fichier : src/UAL.vhd	Entité : UAL
<pre>entity UAL is   port (     Reset      : in std_logic;     OP         : in std_logic_vector(2 downto 0);     A          : in std_logic_vector(31 downto 0);     B          : in std_logic_vector(31 downto 0);     S          : out std_logic_vector(31 downto 0); -- registre de sortie su 32 bits     N, Z, C, V  : out std_logic -- drapeaux (neg, zero, carry, overflow)   ); end UAL;</pre>	

Pour concevoir l'UAL, j'ai opté pour une description combinatoire en VHDL. J'ai utilisé des variables plutôt que des signaux afin de pouvoir modifier directement la sortie après les calculs et les drapeaux.

Pour les opérations d'addition et de soustraction, j'ai étendu les valeurs à 33 bits en ajoutant un bit de signe, ce qui permet de récupérer la retenue (carry) et de détecter les dépassesments de capacité (overflow). Plus précisément, l'opération est effectuée en utilisant le type signed pour gérer correctement les valeurs signées, et l'extension permet de capter les éventuelles retenues sur le bit le plus significatif.

Pour les autres opérations logiques (comme OR, AND, XOR, NOT), les opérations sont réalisées directement sur les 32 bits des entrées, sans extension, car elles ne nécessitent pas de gestion de la retenue ou du dépassement de capacité.

Ensuite, j'ai mis à jour les drapeaux de statut (N pour négatif, Z pour zéro, C pour carry, V pour overflow) après chaque opération :

- Le drapeau N est mis à jour en fonction du bit le plus significatif du résultat.
- Le drapeau Z est mis à 1 si le résultat est zéro.
- Le drapeau C est mis à jour en fonction de la retenue pour les opérations arithmétiques.
- Le drapeau V est mis à jour pour détecter les dépassesments de capacité dans les opérations arithmétiques.

Pour la réinitialisation, tous les signaux de sortie sont remis à zéro lorsque le signal de réinitialisation est activé.

### Simulation

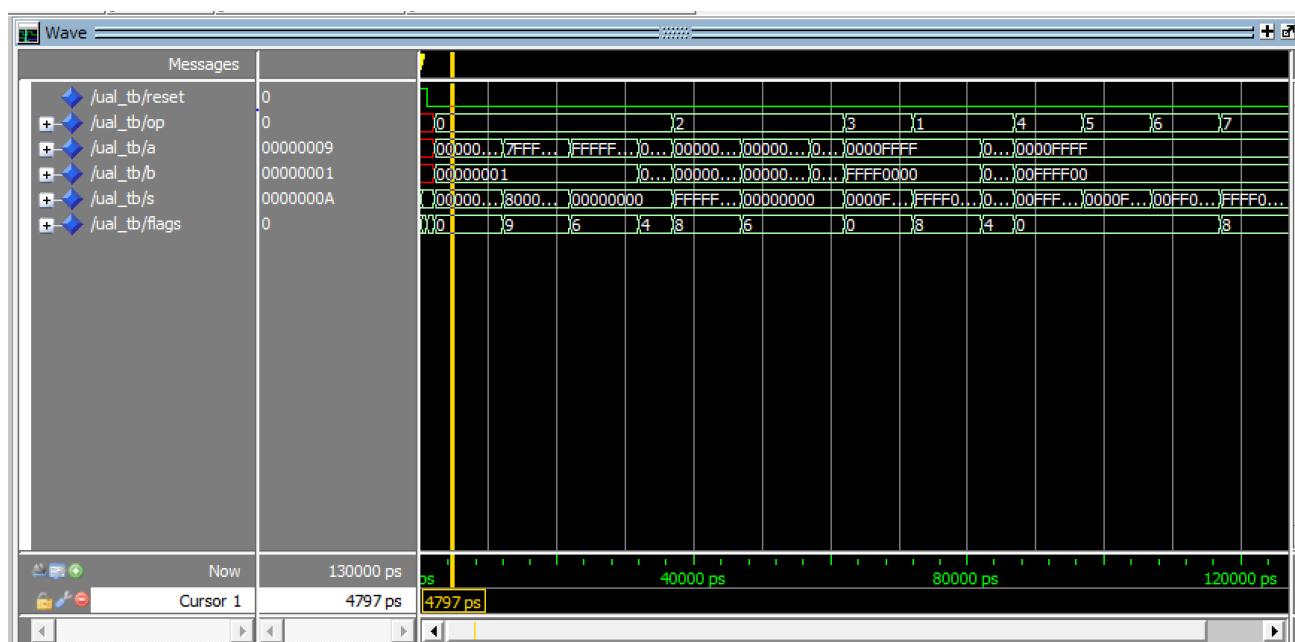
Pour la simulation j'instancie mon composant dans un TB et j'y branche tous les ports à des signaux que je vais pouvoir observer. Pour chaque test de mon scénario, je mets des asserts sur les valeurs de S et des Flags pour remonter une erreur en cas de fonctionnement non prévu.

Mon scénario se compose des tests suivants :

1. Addition simple sans carry ni overflow
2. Addition avec overflow type  $127 + 1 \rightarrow -128$
3. Addition avec carry type  $255 + 1 \rightarrow 0$
4. Soustraction résultat négatif
5. Soustraction résultat 0
6. S = A
7. S = B
8. Or
9. And
10. Xor
11. Not

Ce scénario permet de tester toutes les opérations et tous les flags.

#### Chronogramme modelsim :



#### 1.1.1 : Banc de registres

##### Description

**Fichier:** src/register\_bench.vhd

**Entité:** REG\_BENCH



Ce composant est décrit très simplement comme proposé par la consigne, combinatoire en lecture, synchronique en écriture en utilisant un tableau de 16 fois 32 bits associé à une fonction d'initialisation.

Le registre 0xF est pré-chargé avec la valeur 0x30 (futur pointeur de pile)

Pour visualiser plus facilement ce qu'il se passe, j'ai fait également sortir un signal de test correspondant à la valeur du registre pointé par RW en asynchrone.

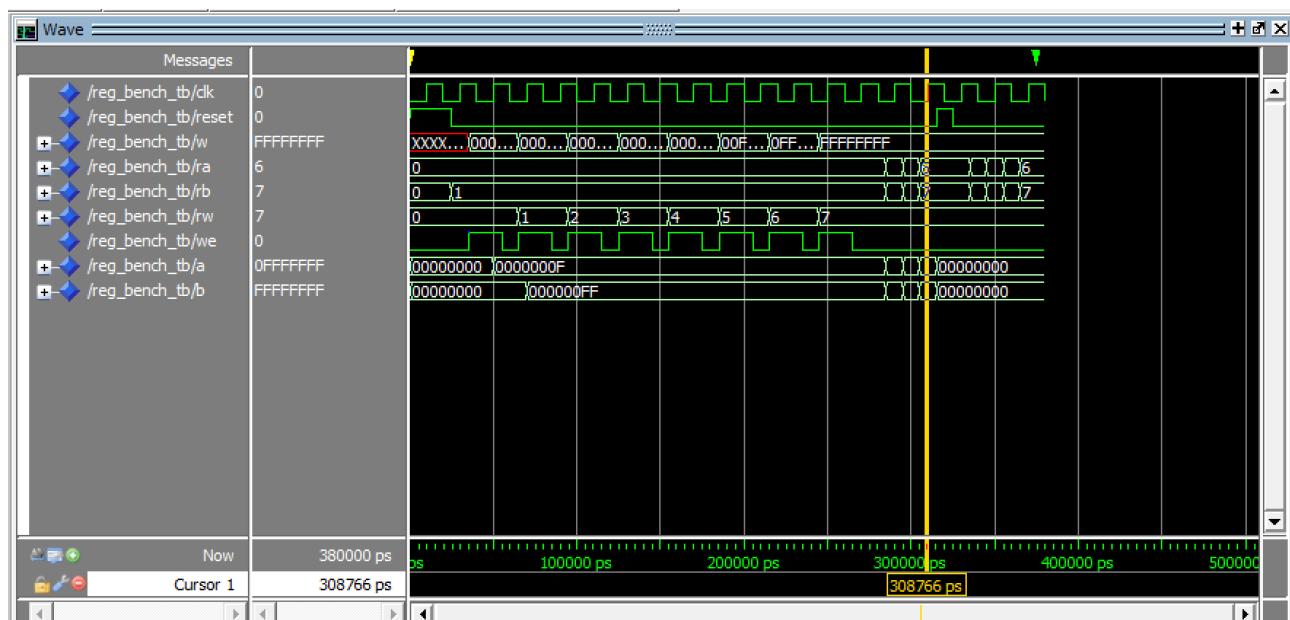
## Simulation

Pour ce Test Bench, j'écris des valeurs différentes sur les 8 premiers registres puis je fais défiler les ports A et B sur ces mêmes registres en vérifiant les valeurs avec des asserts. A sur le registre 0x0, B sur le registre 0x1 puis A sur le registre 0x2, B sur 0x3 jusqu'à 8.

Cela me permet de tester le bon fonctionnement de l'écriture et de la lecture.

Ensuite, je réinitialise mon composant via son port *Reset* le composant puis je parcours à nouveau mes registre de la même manière pour m'assurer que tous sont à zéro. Je teste ainsi le bon fonctionnement du *Reset*. Je constate sur le chronogramme qu'il est bien asynchrone.

## Chronogramme modelsim :



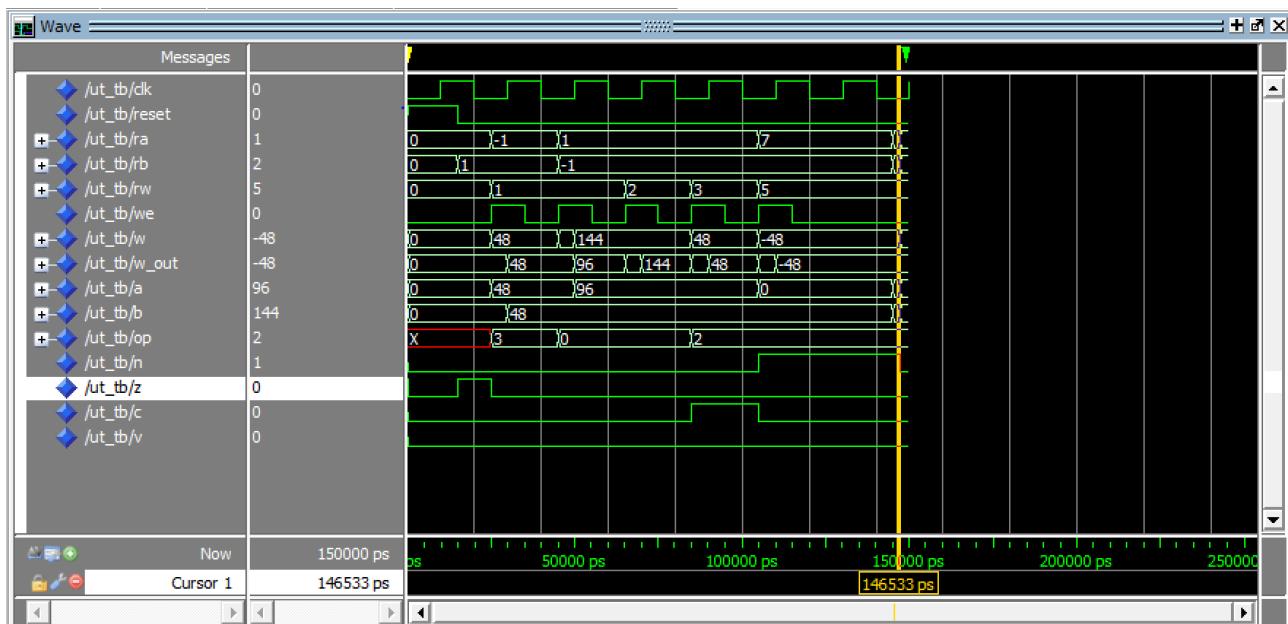
## 1.2.2/3 : Assemblage UAL et Banc de registres

Pour cet assemblage provisoire, j'ai choisi d'instancier et de relier directement mes composants dans un Test Bench. Pour les tests, j'ai configuré une horloge. Je donne ensuite aux ports RA, RB, RW, OP et WE les valeurs correspondant aux opérations demandées dans la consigne :

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

Je peux ainsi contrôler la bonne exécution des opérations directement sur le chronogramme. Pour rendre le Test Bench plus efficace, je parcours ensuite les registre utilisés comme précédemment pour vérifier avec des asserts leur valeur.

### Chronogramme modelsim :



## 1.2 : Mux 2 vers 1

### Description

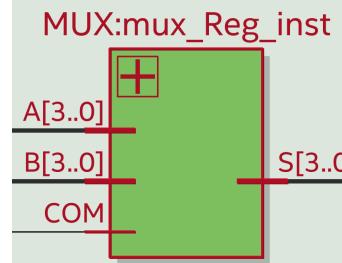
Fichier: src/MUX.vhd

Entité: MUX

```

entity MUX is
  generic (
    N : integer := 32 -- Nombre de bits par défaut pour les vecteurs logiques
  );
  port (
    COM      : in std_logic;
    A        : in std_logic_vector((N-1) downto 0);
    B        : in std_logic_vector((N-1) downto 0);
    S        : out std_logic_vector((N-1) downto 0) -- registre de sortie su N
  );
end MUX;

```



J'ai décrit ce composant très simplement en utilisant un *Process combinatoire* sensible sur les entrées A et B ainsi que la commande COM. J'ai également introduit un *Generic N* correspondant au nombre de bits des vecteurs en entrée/sortie.

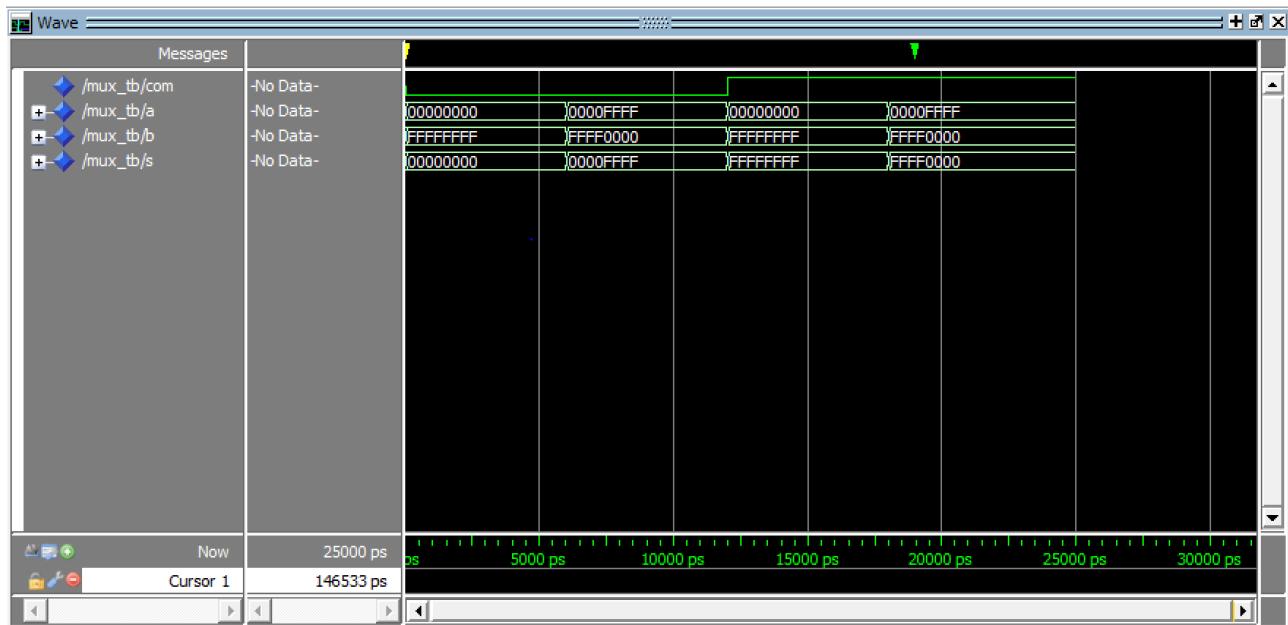
## Simulation

Pour le Test Bench j'ai choisi de tester la valeur de la sortie avec des asserts dans 4 cas :

1. Commutation sur A
2. Changement de la valeur de A
3. Commutation sur B
4. Changement de la valeur de B

Cela me permet de m'assurer du bon comportement de mon composant tant sur commutation que sur changement de la valeur de la voie en entrée.

### Chronogramme modelsim :



## 1.2 : Extension de signe

### Description

**Fichier:** src/sign\_ext.vhd

**Entité:** SIGN\_EXT

```

entity SIGN_EXT is
  generic (
    N : integer := 32 -- Nombre de bits par défaut pour les vecteurs logiques
  );
  port (
    E      : in std_logic_vector((N-1) downto 0);
    S      : out std_logic_vector(31 downto 0) -- registre de sortie su N bits
  );
end SIGN_EXT;

```



J'ai décrit ce composant très simplement en utilisant un *Process combinatoire* sensible sur l'entrée E. J'ai également introduit un *Generic N* correspondant au nombre de bits du vecteurs en entrée. La sortie est toujours sur 32 bits.

Une amélioration possible serait de Passer le nombre de bits en sortie en *Generic* également mais cela n'est pas nécessaire ni demandé dans ce projet

## Simulation

Mon Test Bench instancie ce composant avec 16 bit en entrée et observe la sortie. Je teste très simplement mon extenseur de signe en lui appliquant une valeur positive puis négative en entrée et en vérifiant à chaque fois la valeur en sortie avec un assert. Cela permet de tester les deux cas de figure.

### Chronogramme modelsim :

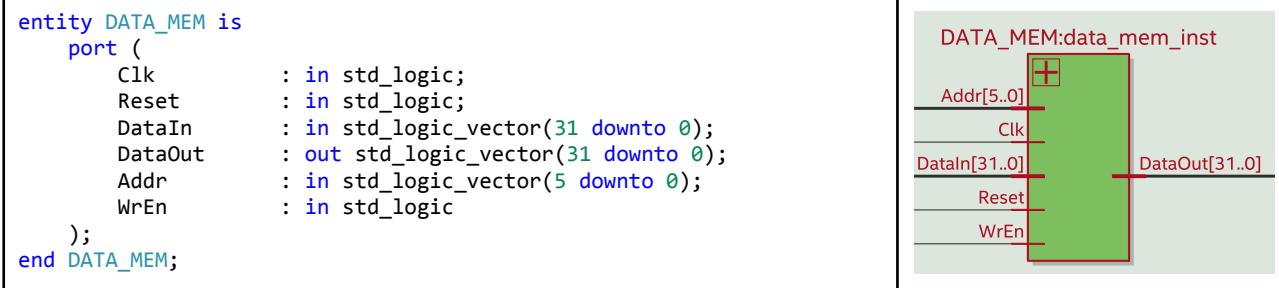


## 1.2 : Mémoire de données

### Description

**Fichier:** src/data\_mem.vhd

**Entité:** DATA\_MEM



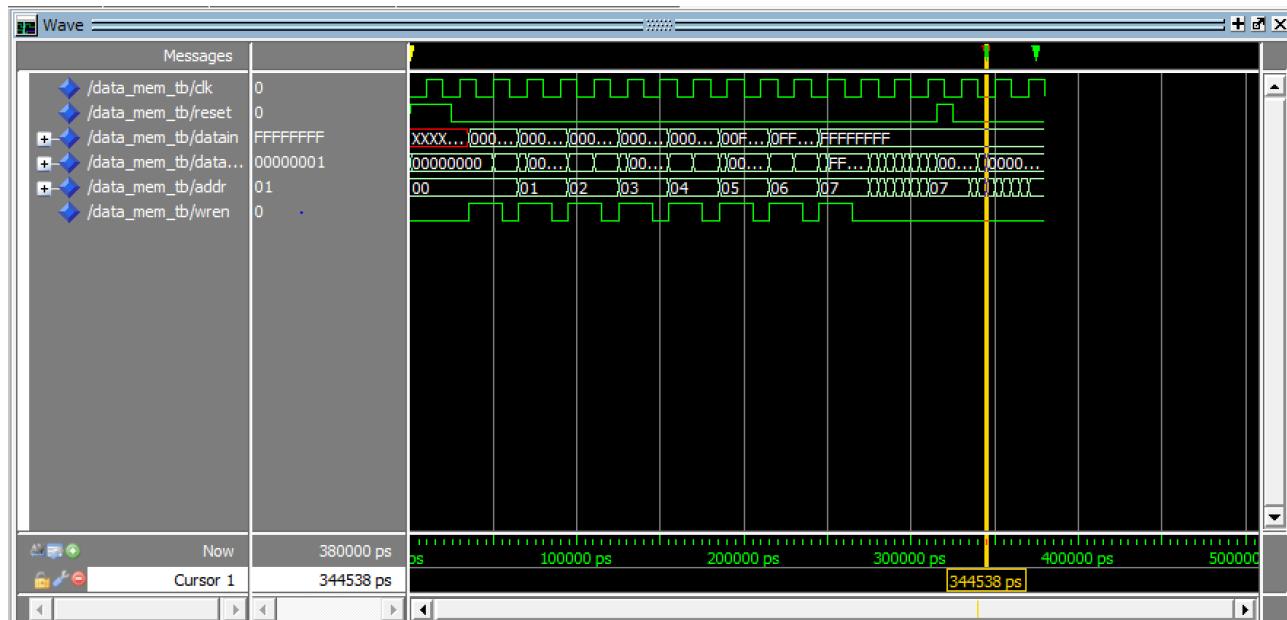
Pour décrire ce composant, j'ai repris le code du banc de registre en y appliquant les modifications suivantes :

- Les bus d'adresse RA, RB et RW sont remplacés par *Addr*,
- Les 2 bus de sortie A et B sont remplacé par 1 seul bus *DataOut* pointé par *Addr* toujours en combinatoire,
- L'écriture de fait sur l'emplacement pointé par *Addr* toujours en synchrone sur *WrEn*,
- La taille du tableau de vecteurs passe à 64 fois 32 bits
- La fonction d'initialisation est modifiée pour initialiser toutes les cases mémoires et pré charger des données.

## Simulation

Pour le test bench de ce composant, j'ai repris le scénario de test du banc de registre : J'écris en mémoire dans les emplacements 0 à 7 puis je parcours ces emplacements en vérifiant la valeur sur *DataOut* avec des asserts puis je reset et je parcours à nouveau les emplacements pour contrôler le bon effacement.

### Chronogramme modelsim :



On note ici une erreur sur la valeur de l'emplacement 1 après le reset. Cela est dû au fait que la mémoire a été modifiée par la suite et que la valeur 0X01 est ajoutée dans ce mot à la place de 0.

## 1.3 Assemblage de l'Unité de Traitement

### Description

Fichier:	src/UT.vhd	Entité:	UT
<pre>ENTITY UT is   PORT   (     Clk      : IN STD_LOGIC;     Reset    : IN STD_LOGIC;     RegWr   : IN STD_LOGIC;     RW, RA, RB : IN STD_LOGIC_VECTOR(3 downto 0);     COM_Mux_im : IN STD_LOGIC;     COM_Mux_reg : IN STD_LOGIC;     COM_Mux_out : IN STD_LOGIC;     OP       : IN STD_LOGIC_VECTOR(2 downto 0);     Im      : IN STD_LOGIC_VECTOR(7 downto 0);     WrEn   : IN STD_LOGIC;     Flags   : OUT STD_LOGIC_VECTOR(3 downto 0);      busA, busB, busW : OUT STD_LOGIC_VECTOR(31 downto 0) -- debug   ); END entity;</pre>			

Pour l'unité de commande, j'ai choisi de décrire un composant qui instancie tous les composants décrits précédemment selon le schéma proposé dans le sujet. Les noms des signaux sont ceux des schémas. Les drapeaux N, Z, C et V sont en sortie sur un vecteur sur 4 bit dans cet ordre (N sur le bit de poids fort).

Pour permettre de mieux comprendre ce qu'il se passe lors de la simulation, je fais également sortir la valeur des bus A, B et W. (le bus B sera plus tard utilisé pour l'afficheur notamment).

Le code de ce composant a été légèrement modifié dans la partie 4 mais cela n'a pas d'incidence sur son fonctionnement ici et le test bench a été adapté pour fonctionner sans être influencé par la modification. Le chronogramme présenté ci-dessous est donc fidèle à celui obtenu lors du développement de l'UT.

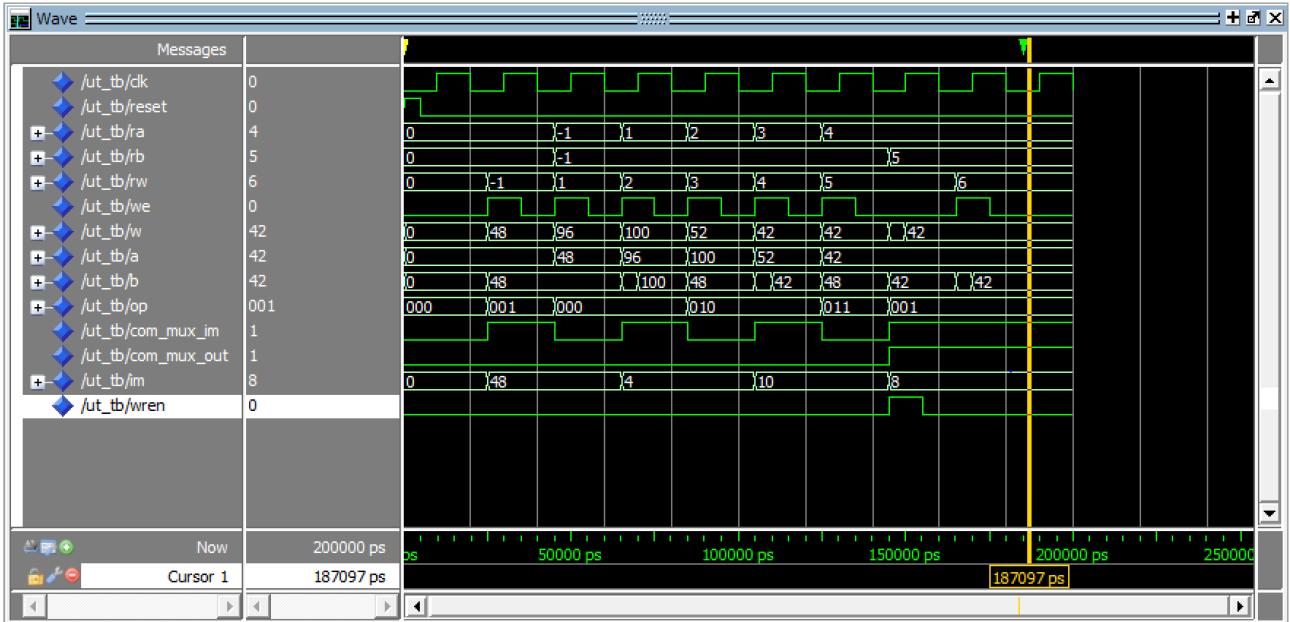
### Simulation

Le test Bench de ce composant est crucial car il permet de valider le bon fonctionnement de cette partie centrale du processeur. Pour cela, j'ai instancié mon unité de traitement et relié ses ports à des signaux observables sur le chronogramme.

Pour le scénario de test, j'ai repris les instructions proposées dans le sujet. Pour chacune de ces instructions, j'ai réfléchi aux valeurs à donner aux signaux de commandes et aux bus d'adresse tel que le fera l'unité de Contrôle. Je donne donc sa valeur à chaque signal, j'attends le prochain front montant d'horloge et je contrôle le résultat sur le bus W avec un assert.

Pour l'opération d'écriture en mémoire, l'ai volontairement commuté le mux en amont du bus W pour pouvoir observer en direct l'écriture sur front montant de l'horloge.

### Chronogramme modelsim :



# Partie 2 - Unité de gestion des instructions

## Instruction Memory

### Description

Fichier:	Entité:
<pre>entity INSTRUCTION_MEMORY is   port(     PC:      in std_logic_vector (31 downto 0);     Instruction:  out std_logic_vector (31 downto 0)   ); end entity;</pre>	<b>INSTRUCTION_MEMORY:IM_inst</b> PC[31..0] + Instruction[31..0]

Le code de ce composant nous est déjà fourni. Il contient une liste d'instructions en dur qui constituent le programme en assembleur que l'on exécutera plus tard.

L'analyse de ce code révèle que celui-ci contient une boucle qui parcourt et somme tous les emplacements en mémoire de 0x10 à 0x19 puis enregistre le résultat dans la mémoire (ce qui permettra de l'afficher) avant de reboucler à nouveau.

### Simulation

La simulation n'a pas grand intérêt car le code fourni est déjà fonctionnel. On se contente ici de faire défiler PC et d'observer que l'instruction en sortie défiler également.

### Chronogramme modelsim :



# Registre PC

## Description

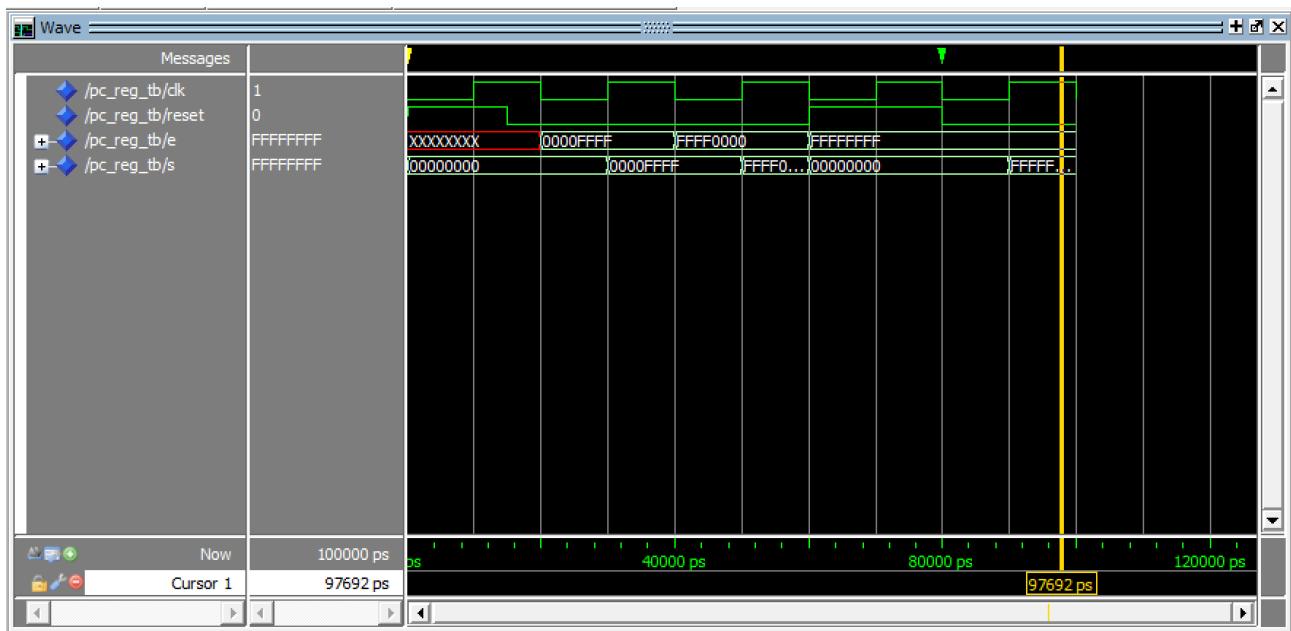
Fichier:	Entité:
<pre>entity PC_REG is   port(     Clk : in STD_LOGIC;     Reset : in STD_LOGIC;     E: in std_logic_vector (31 downto 0);     S: out std_logic_vector (31 downto 0)   ); end entity;</pre>	<b>PC_REG: PC_REG</b>

Le registre PC est décrit en synchrone avec un reset asynchrone. A chaque front montant de l'horloge, on applique l'entrée sur la sortie.

## Simulation

On teste à l'aide d'asserts que la sortie prend bien la valeur de l'entrée au front montant de l'horloge et pas avant. On teste aussi le reset et en dehors d'un front montant pour vérifier qu'il est bien asynchrone.

### Chronogramme modelsim :



# PC Update Unit

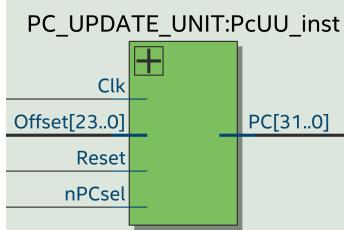
## Description

Fichier:	Entité:
<code>src/PC_update_unit.vhd</code>	<code>PC_UPDATE_UNIT</code>

```

entity PC_UPDATE_UNIT is
  port(
    Clk : in STD_LOGIC;
    Reset : in STD_LOGIC;
    nPCsel : in STD_LOGIC;
    Offset: in std_logic_vector (23 downto 0);
    PC : out std_logic_vector (31 downto 0)
  );
end entity;

```

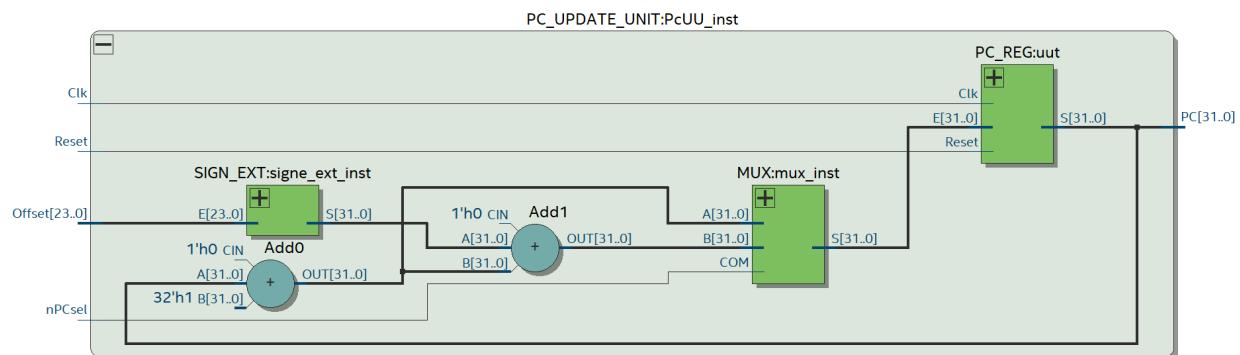


Le PC Update Unit est constitué du registre PC est relié en sortie à un signal lui même donnant sur le port de sortie PC. Ce signal est ensuite transformé en integer, incrémenté de 1 puis transformé à nouveau en signal sur 32 bits.

L'entrée Offset passe par un signe extender (voir partie 1) puis est également transformée en integer, sommée avec l'integer du PC+1 puis transformée à nouveau en signal sur 32 bits.

Ces deux signaux obtenus sont en entrée d'un mux commandé par nPCsel dont la sortie est reliée au registre PC.

Schéma interne du composant :

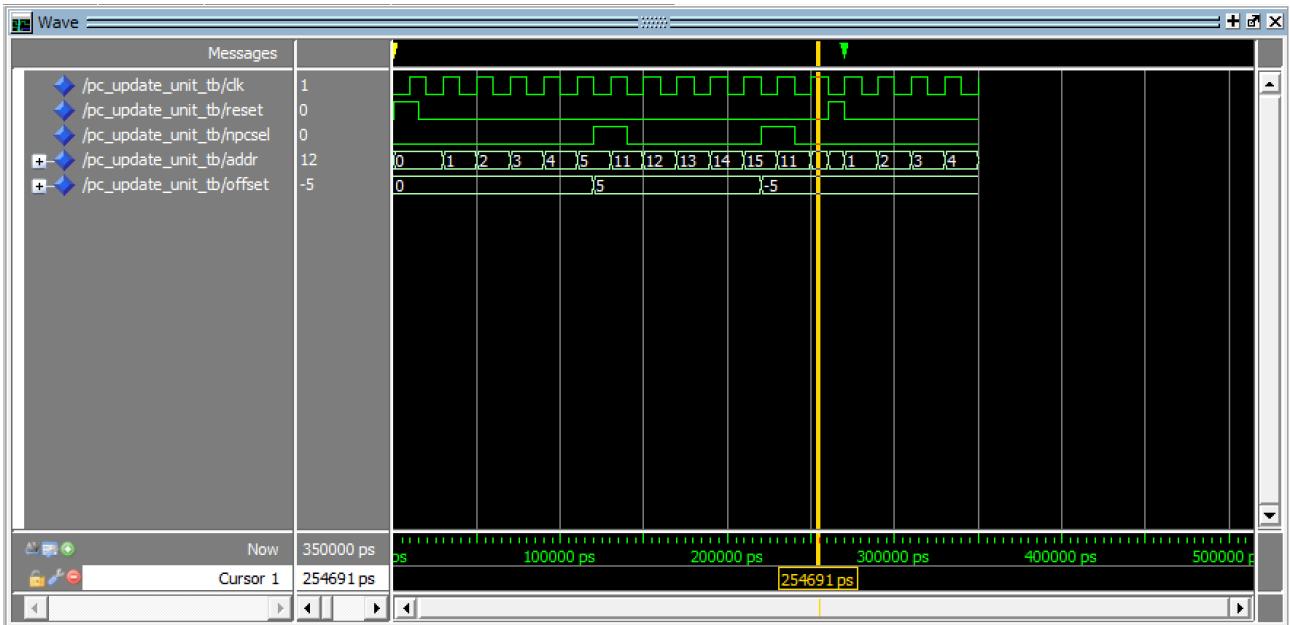


## Simulation

On laisse le composant tourner quelques instants pour vérifier que Addr s'incrémente correctement puis on applique la valeur 5 sur l'offset en passant nPCsel à 1. Addr doit sauter 5 valeurs. On le laisse à nouveau défiler 5 tours puis on recommence cette fois-ci avec offset à -5. Addr doit repartir de là où on avait sauté la première fois.

On teste également le reset : Addr doit repartir de zéro.

## Chronogramme modelsim :



## Assemblage Unité d'instruction

### Description

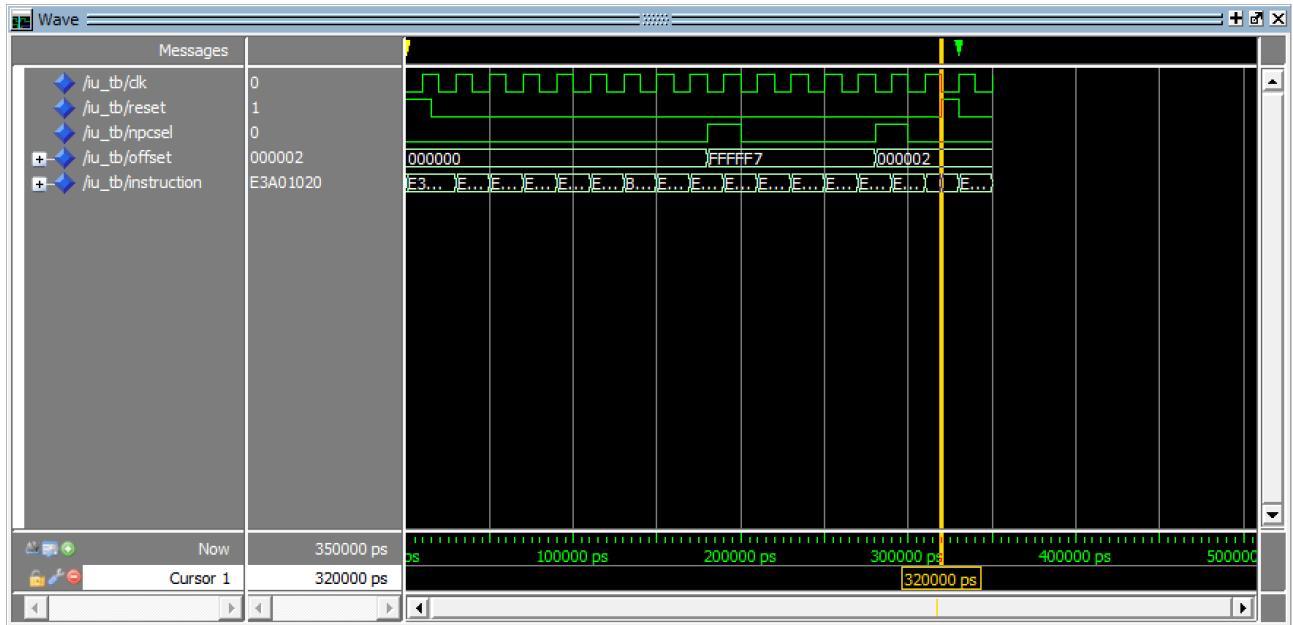
Fichier:	src/Instructions_Unit.vhd	Entité:	IU
<pre>ENTITY IU is   PORT   (     Clk      : IN STD_LOGIC;     Reset    : IN STD_LOGIC;     nPCsel   : IN STD_LOGIC;      Offset   : IN STD_LOGIC_VECTOR(23 downto 0);     Instruction : OUT STD_LOGIC_VECTOR(31 downto 0)   ); END entity;</pre>			

Ce composant est simplement l'assemblage de l'update unit et de la mémoire d'instruction et permet de simplifier l'assemblage plus tard du processeur.

### Simulation

Le scénario de test est identique à celui du PC Update Unit, mise à part que l'on observe ici l'évolution de l'instruction au lieu de l'adresse. On vérifie tout de même que les instructions observées sont cohérentes avec la mémoire en fonction des sauts qui sont faits.

### Chronogramme modelsim :



# Partie 3 - Unité de Contrôle

## 3.1 : Tableau "valeurs des commandes" complété

INSTRUCTION	nPCSel	RegWr	ALUSrc	ALUCtr	PSREn	MemWr	WrSrc	RegSel	RegAff
ADDi	0	1	1	0	(s) 0	0	0	-	0
ADDr	0	1	0	0	(s) 0	0	0	0	0
BAL	1	0	-	---	0	0	-	-	0
BLT	flg N	0	-	---	0	0	-	-	0
CMP	0	0	bit# 1	10	1	0	-	-	0
LDR	0	1	1	0	0	0	1	-	0
MOV	0	1	bit# 1	1	0	0	0	-	0
STR	0	0	1	0	0	1	-	1	1

## 3.2 : Registre PSR

### Description

Fichier:	Entité:
<pre>entity REG32 is   port(     Clk : in STD_LOGIC;     Reset : in STD_LOGIC;     WrEn : in STD_LOGIC;     E: in std_logic_vector (31 downto 0);     S: out std_logic_vector (31 downto 0)   ); end entity;</pre>	REG32:inst_reg

Pour réaliser ce composant que j'ai nommé *reg32*, j'ai repris le code composant *regPC* défini dans la partie 2. J'y ai ajouté une entrée WE et conditionné l'écriture sur la sortie à WE = '1'. Voici le code de l'écriture synchrone avec le reset asynchrone :

```
if Reset = '1' then
  S <= (others => '0');
elsif rising_edge(Clk) and WrEn = '1' then
  S <= E;
end if;
```

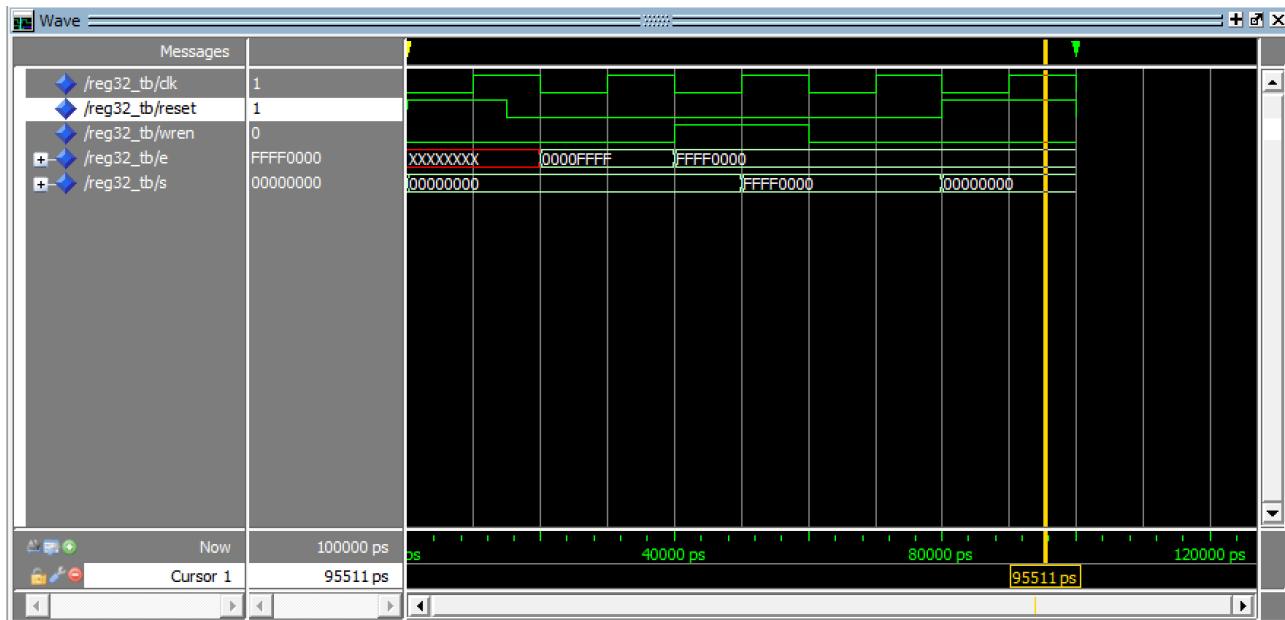
### Simulation

Pour la simulation, j'ai effectué les test ci-dessous en laissant passer un tour d'horloge puis en contrôlant la sortie avec un assert :

1. Changement de la valeur en entrée sans WE : la sortie ne doit pas changer
2. Changement de la valeur en entrée avec WE : la sortie doit changer
3. Reset : la sortie doit repasser immédiatement à zéro (reset asynchrone)

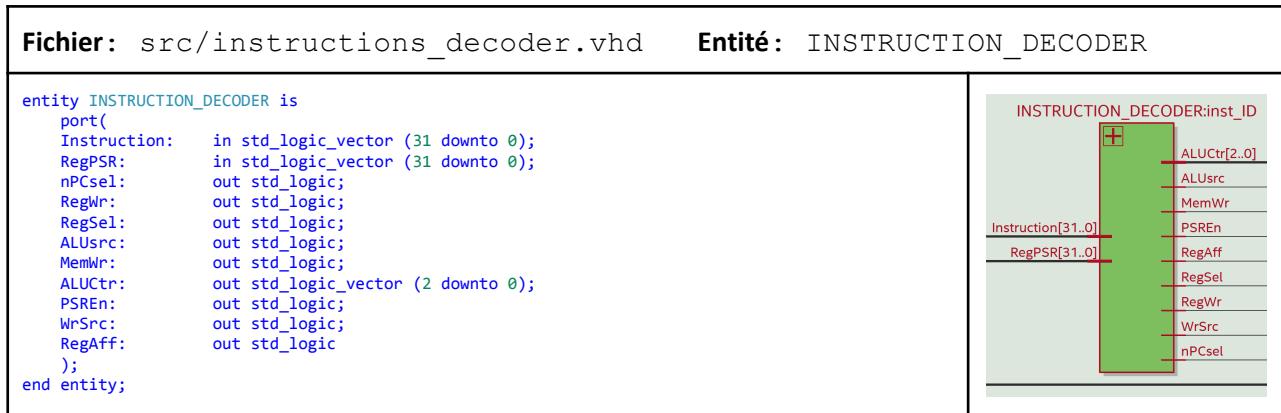
Cela permet de bien tester l'écriture uniquement quand WE est à '1' ainsi que le reset asynchrone.

### Chronogramme modelsim :



## 3.2 : Décodeur d'Instructions

### Description



Le but de ce composant est de décoder le bus *Instruction* pour en déduire l'instruction assembleur correspondante et d'appliquer les bonnes valeurs sur les signaux de contrôle. Ce composant fonctionne entièrement en combinatoire

Comme proposé par la consigne, on utilise un process sensible sur l'entrée *Instruction* pour définir un signal *Insrt\_courrente* qui représente le type d'instruction (type *enum* : MOV, ADDi, ADDR, CMP, LDR, STR, BAL, BLT, NONE). Un deuxième process sensible sur *Insrt\_courrente* permet d'appliquer les bonnes valeurs aux signaux de commande (cf. tableau partie 3.1).

Après étude de la documentation, on remarque que pour les instructions de traitement et de transfert, les 12 bits de poids fort permettent de décoder l'instruction. Pour les instructions de branchement BLT et BAL, seul les 8 bits de poids fort permettent de déduire l'instruction utilisée. On obtient le code suivant pour le décodage de l'instruction :

```

instr_courante <= NONE;
  case instruction(31 downto 20) is
    when x"E3A" =>
      instr_courante <= MOV;
    when x"E28" =>
      instr_courante <= ADDi;
    when x"E08" =>
      instr_courante <= ADDR;
    when x"E35" =>
      instr_courante <= CMP;
    when x"E61" =>
      instr_courante <= LDR;
    when x"E60" =>
      instr_courante <= STR;
    when others => null;
  end case;
  -- Cas des instructions de branchement
  case instruction(31 downto 24) is
    when x"EA" =>
      instr_courante <= BAL;
    when x"BA" =>
      instr_courante <= BLT;
    when others => null;
  end case;

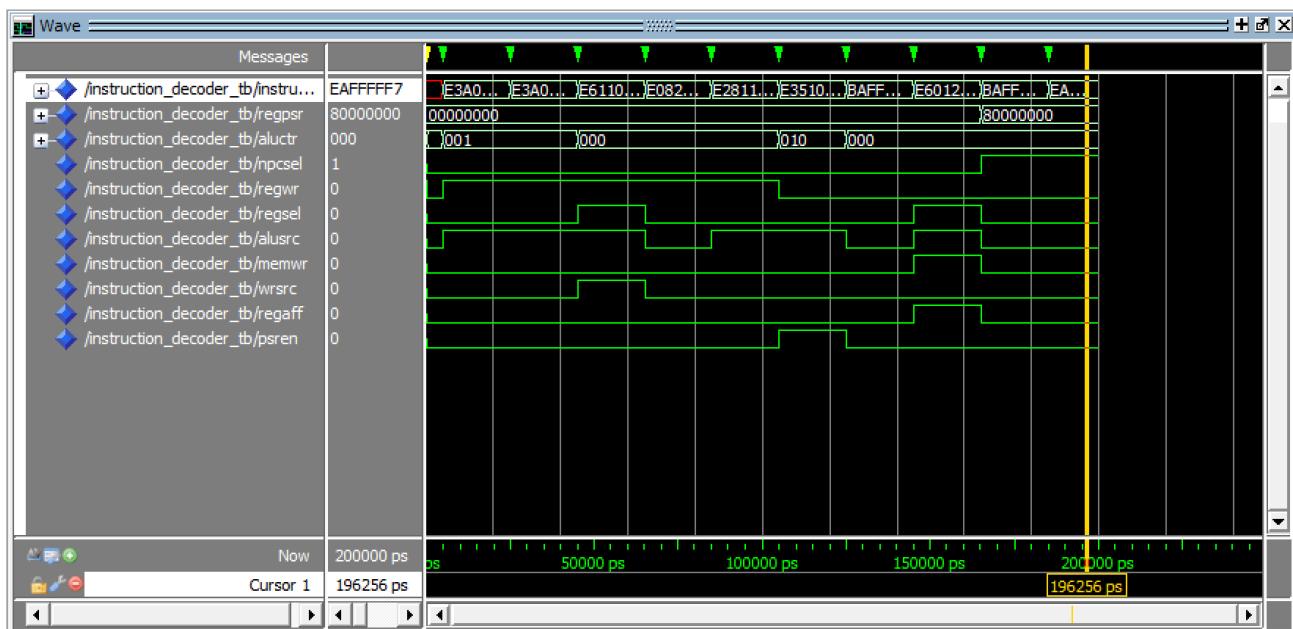
```

## Simulation

Pour le test de ce composant, je reprend les instructions assemblées de la mémoire d'instruction et je les applique unes à unes en entrée du décodeur. J'ai également doublé l'instruction BLT afin de pouvoir observer la dépendance de *nPCsel* au bit 31 du PSR.

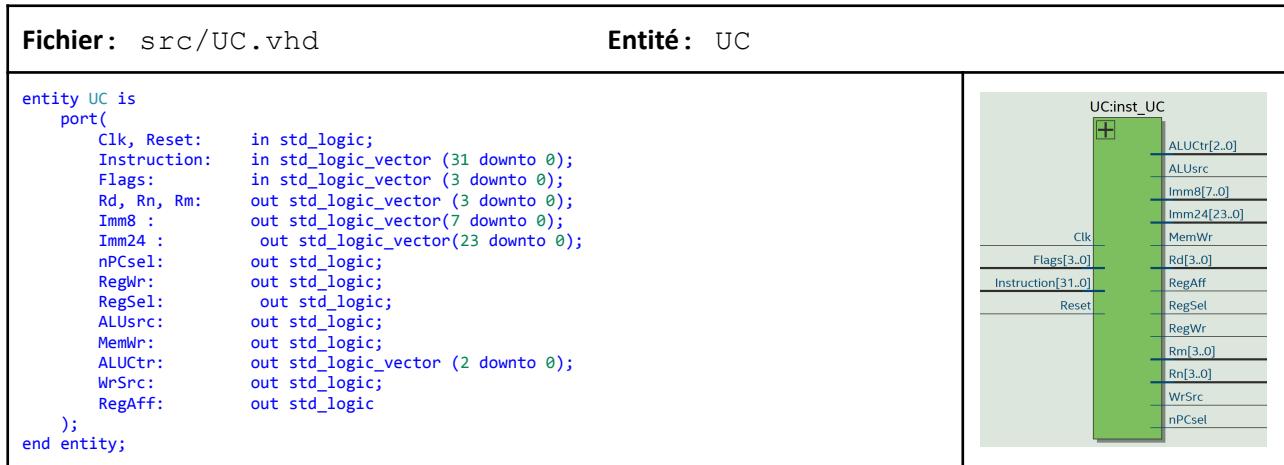
Il n'est pas nécessaire de rendre ce banc de test auto-vérifiant car les sorties dépendent du tableau présenté plus tôt. Le test bench sert davantage à vérifier le bon fonctionnement du composant et le bon changement des valeurs des signaux.

### Chronogramme modelsim :



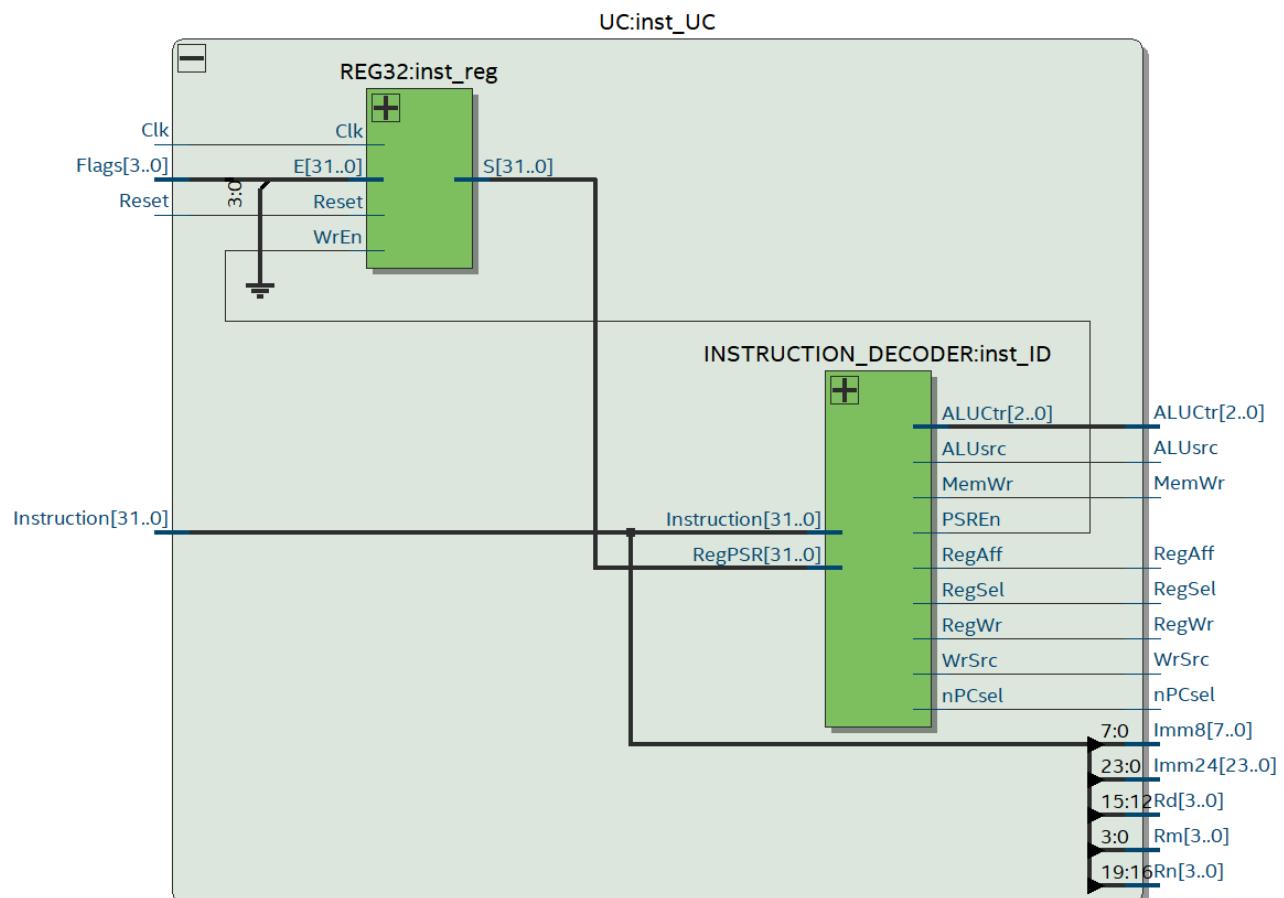
### 3.3 : Assemblage de l'Unité de Contrôle

#### Description



L'unité de contrôle est l'assemblage du registre PSR et du décodeur d'instructions. Elle prend en entrée l'instruction ainsi que les flags en provenance de l'unité de traitement. En sortie, on retrouve les différents signaux de contrôle du tableau de la partie 3.1 (sauf *PSRen* qui est interne au composant) ainsi que les bus d'adresse *Rd*, *Rn* et *Rm* et les immédiats sur 8 et 24 bits *Imm8* et *Imm24* provenant directement de l'instruction.

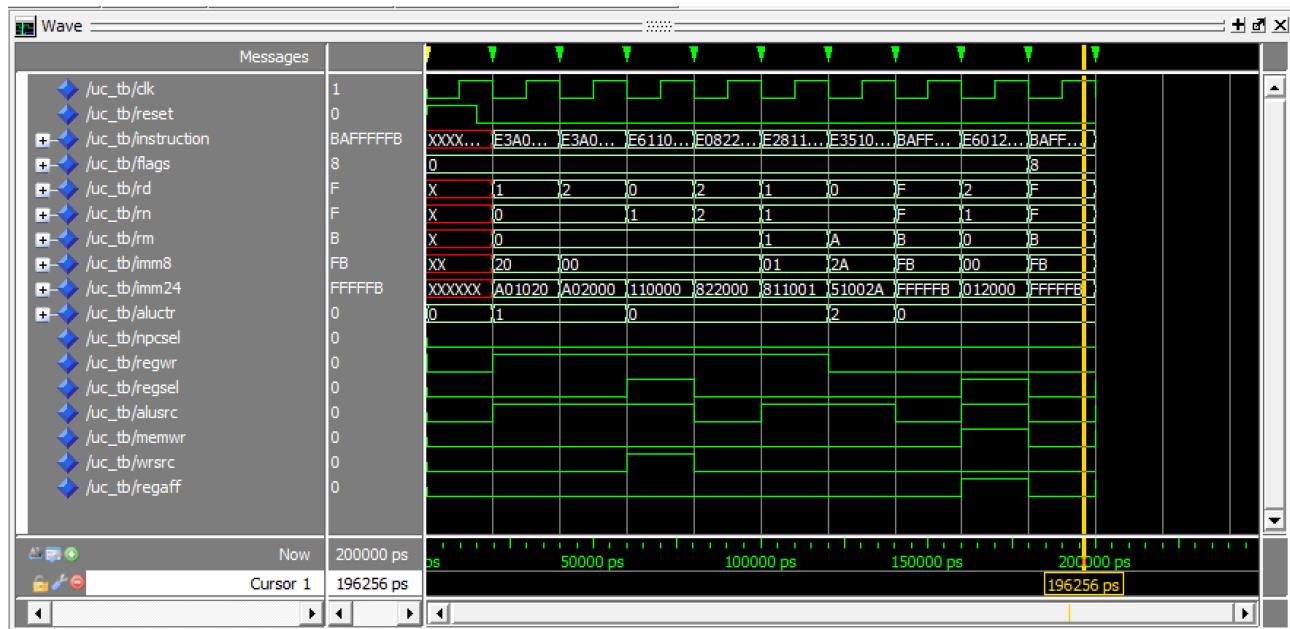
Schéma interne du composant :



## Simulation

Le test bench de ce composant vise principalement à vérifier le bon fonctionnement de l'assemblage du PSR et du décodeur. Il reprend exactement le scénario du décodeur d'instruction. On vérifie également tout de même visuellement les valeurs des sorties *Rd*, *Rn*, *Rm*, *Imm8* et *Imm24*.

### Chronogramme modelsim :



# Partie 4 - Assemblage et validation du processeur

## 4.1 : Modification de l'Unité de Traitement

### Description

Pour adapter l'Unité de Traitement, j'ai instancié un mux supplémentaire que j'ai relié comme sur le schéma proposé (RB sur A, RW sur B).

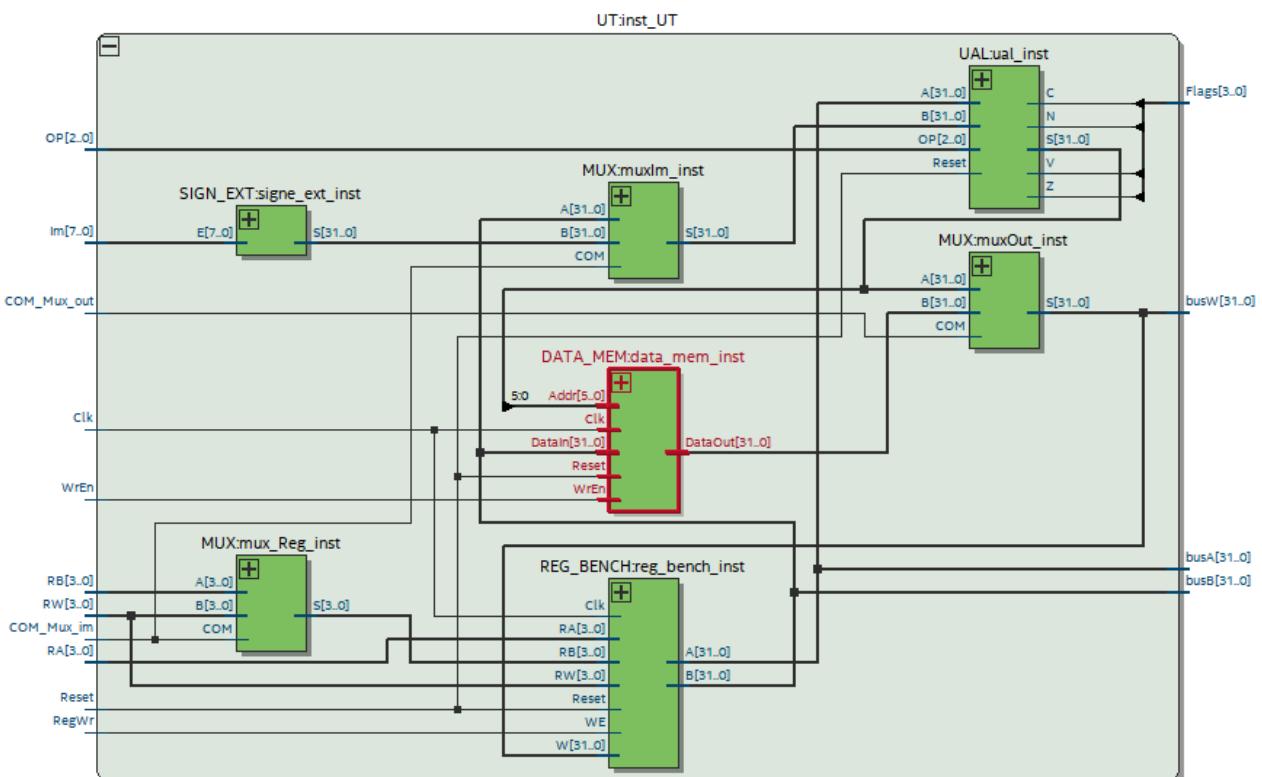
J'ai choisi de conserver les noms des bus d'entrée RA, RB et RW afin de limiter au maximum les modifications sur le composant. Concernant la sortie de l'Unité de Contrôle, Rw correspond à Rd, RA à Rn et RB à Rm.

J'ai également ajouté un port d'entrée *RegSel* pour la commande de ce nouveau multiplexeur.

Pour la sortie de l'afficheur, j'ai utilisé le bus B qui était déjà en sortie précédemment pour mes tests.

En raison de la modification assez simple de ce composant, je n'ai pas créé de Test Bench spécifique. J'ai simplement mis à jour le test bench précédent pour qu'il s'exécute normalement en forçant *RegSel* à '0'. J'ai tout de même vérifié le bon fonctionnement "sur le tas" lors de la simulation du processeur complet à l'étape suivante.

Voici le schéma du composant :



## 4.2/3 : Assemblage du Processeur

### Description

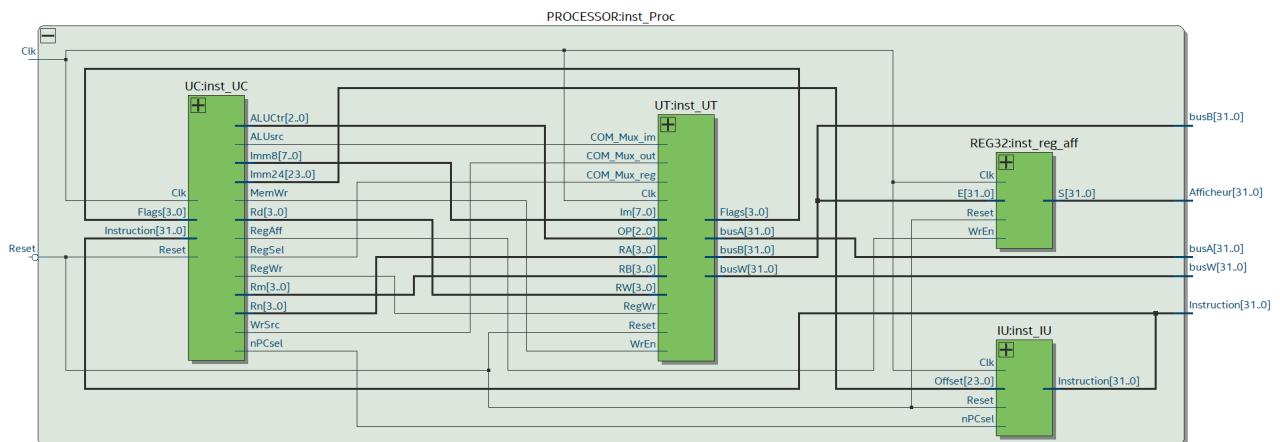
Fichier: src/Processor.vhd	Entité: PROCESSOR
<pre>entity PROCESSOR is   port(     Clk, Reset:      in std_logic;     busA, busB, busW: out std_logic_vector(31 downto 0); -- debug     Instruction:     out std_logic_vector(31 downto 0); -- debug     Afficheur:        out std_logic_vector(31 downto 0)   ); end entity;</pre>	

En raison de la répartition des 3 parties précédentes en 3 grands composants, l'assemblage du processeur est grandement simplifié.

J'ai donc créé un composant *PROCESSOR* qui instancie les 3 parties précédentes comme sur le schéma proposé. J'ai également instancié un registre 32 bits (cf : 3.2 : Registre PSR) prenant en entrée la sortie busB de l'Unité de Traitement et ayant le bus *Afficheur* en sortie.

Mon composant processeur prend en entrée une *Clock* et un *Reset* distribué en cascade au sous-composant en ayant besoin. Son unique sortie est le bus *Afficheur*. Pour faciliter la compréhension lors de la simulation, j'ai ajouté des sorties provisoires pour les bus A, B et W de l'Unité de Traitement ainsi que pour l'instruction.

Schéma interne du composant :



### Simulation

Le Test bench de ce composant est relativement simple. Après instantiation du processeur. Il suffit de le laisser tourner suffisamment longtemps pour que programme décrit précédemment ai le temps de finir de s'exécuter et applique la bonne valeur sur la sortie *Afficheur* avant de reboucler.

J'ai modifié la mémoire de donnée pour y mettre différentes valeurs sur les emplacements visités par la programme dont on doit voir la somme en sortie sur *Afficheur*. Le Test Bench contrôle cette valeur avec un assert. Voici la modification faite dans la mémoire de donnée :

```
-- Fonction d'initialisation du Mem
function init_Mem return table is
  variable result : table;
```

```

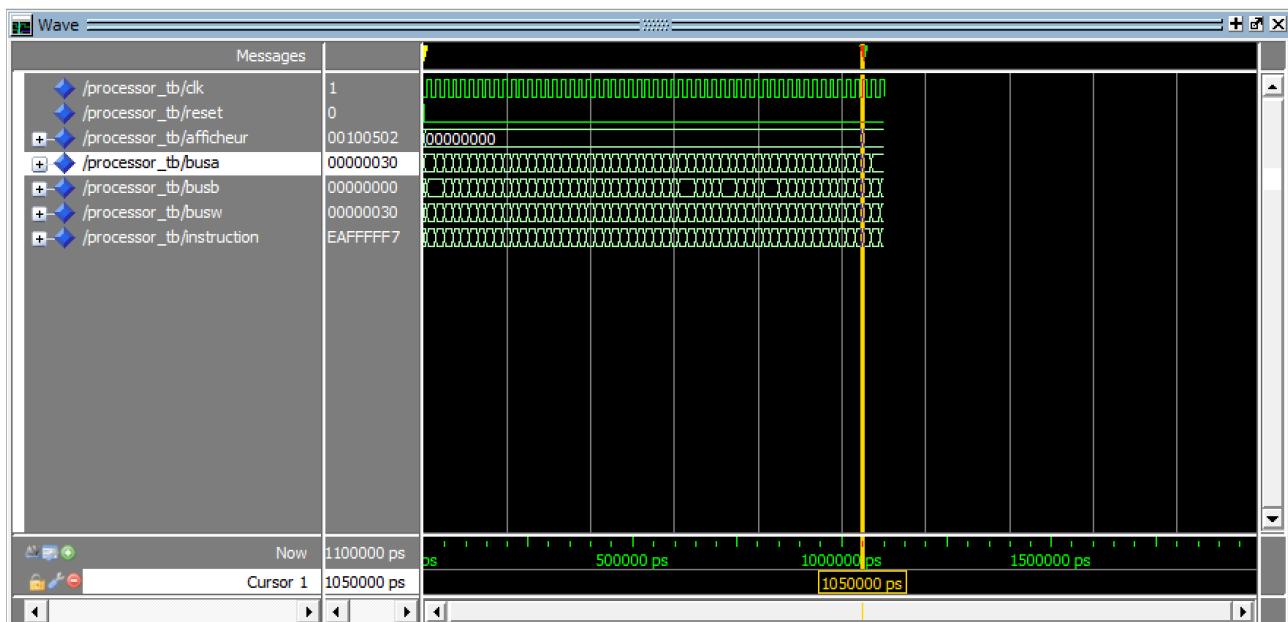
begin
  for i in 31 downto 0 loop
    result(i) := (others=>'0');
  end loop;
  for i in 63 downto 54 loop
    result(i) := (others=>'0');
  end loop;
  -- Mémoire Chargée pour test
  result (1):=x"00000001";

  result (32):=x"00000001";
  result (33):=x"00000100";
  result (34):=x"00000200";
  result (35):=x"00000200";
  result (36):=x"00080000";
  result (37):=x"00000000";
  result (38):=x"00000000";
  result (39):=x"00000000";
  result (40):=x"00000001";
  result (41):=x"00080000";

  return result;
end;

```

### Chronogramme modelsim :



# Partie 5 - Test du Processeur sur carte FPGA

## 5.1 : Top Level & Projet Quartus

### Description du Top Level

**Fichiers :** src/Quartus/TopLevel.vhd  
src/Quartus/SEVEN\_SEG.vhd

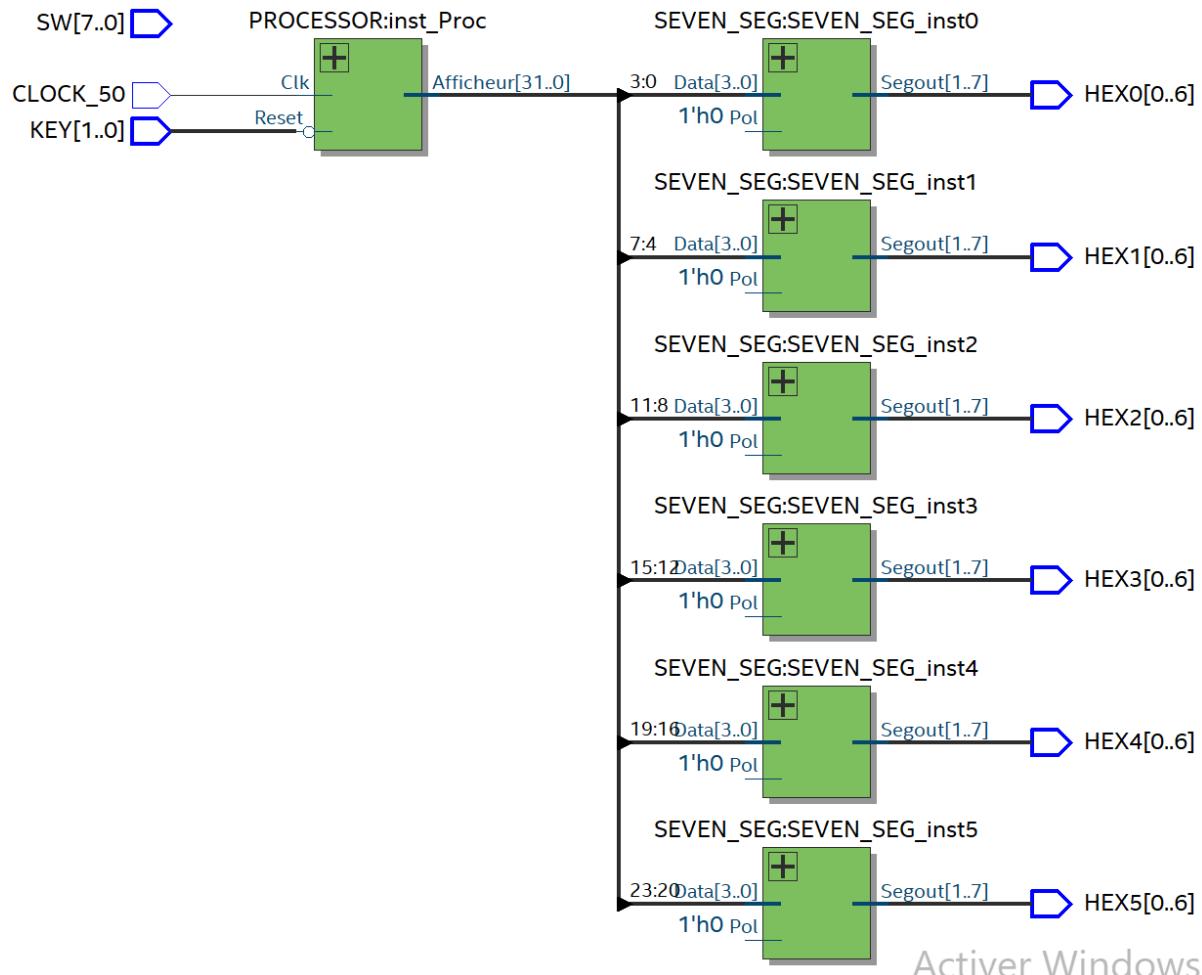
**Entité :** TopLevel

Le but du Top Level est d'instancier le processeur et de le relier aux afficheurs 7-segments de la carte DE10 lite. Pour cela, on utilise le composant SEVEN\_SEG des TP précédents. Ce composant affiche sur son bus de sortie (rélié à l'afficheur) le chiffre correspondant à la valeur sur son bus d'entrée *Data*.

On en instancie 6 pour les 6 afficheurs de la carte et on relie leurs entrées au 6 premier quartets du bus *Afficheur* (0-3, 4-7, 8-11, 12-15, 16-19 et 20-23) en sortie du processeur.

On utilise l'horloge à 50MHz de la carte et on relie le *Reset* à un des boutons.

Schéma :



## 5.2 : Test sur Carte

Une fois le projet Quartus créé, compilé et les pins de la carte affectés correctement, on peut charger le projet dans la carte. On observe alors sur les afficheurs la somme des valeurs que l'on a entré dans la mémoire de donnée sur les emplacements 0x20 à 0x29.

### Validation par le professeur en classe

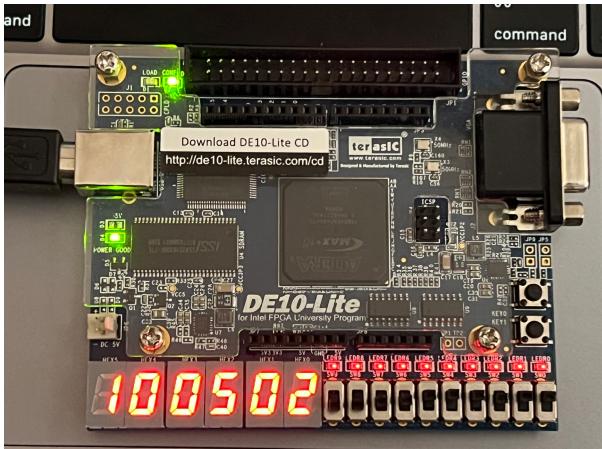
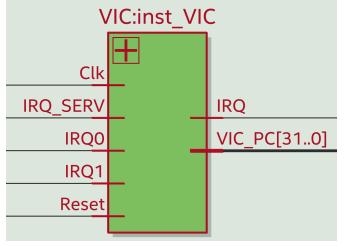


Photo de la carte avec la somme correcte sur les afficheurs

# Partie 6 - Gestion des interruptions externes

## 6.1 : Vectorized Interruption Controller

### Description

Fichier:	src/VIC.vhd	Entité:	VIC
	<pre>entity VIC is   port (     Clk      : in std_logic;     Reset    : in std_logic;     IRQ_SERV : in std_logic;     IRQ0, IRQ1 : in std_logic;      IRQ      : out std_logic;     VIC_PC   : out std_logic_vector(31 downto 0)   ); end VIC;</pre>		

Le VIC est décrit en synchrone sur l'horloge avec un reset asynchrone. On utilise des signaux *IRQ0/1\_last* pour détecter les fronts montants. Des variables *IRQ0/1\_memo* permettent de connaître l'état des IRQ.

Un front montant sur un port irq active une irq et seul *irq\_serv* peut la désactiver (sauf en cas de reset).

Une succession de if permet de donner la bonne valeur à VICPC tout en gérant la priorité.

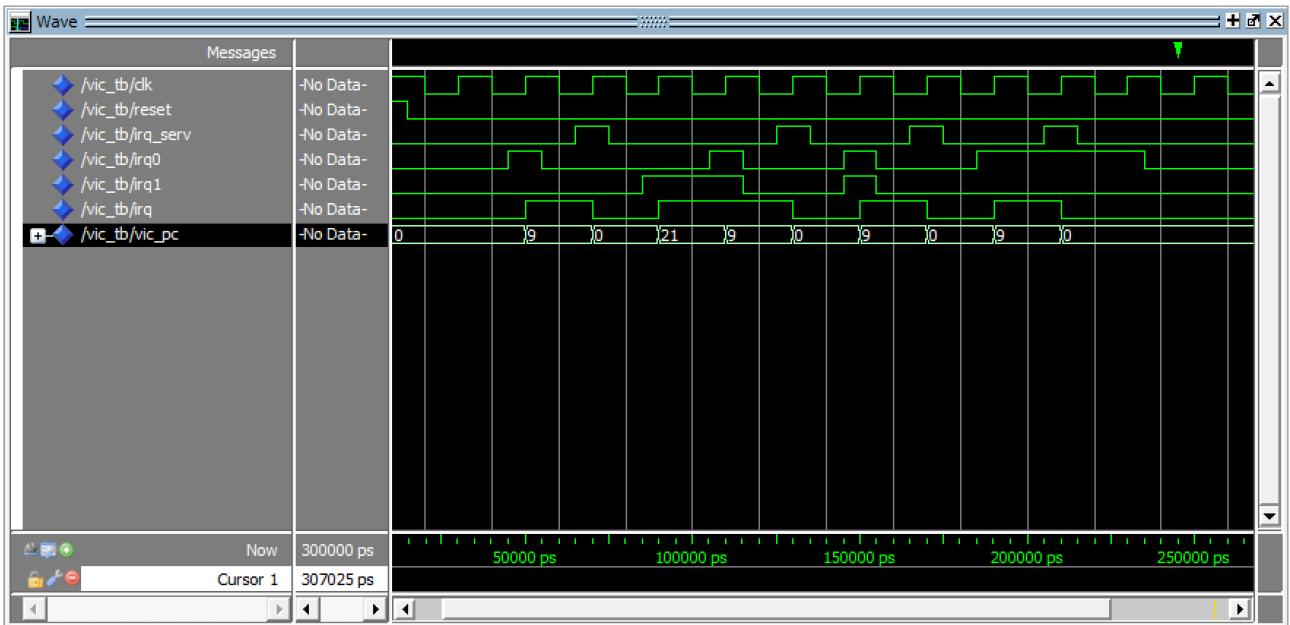
### Simulation

Le scénario de test comporte les tests suivant :

1. état initial après reset
2. IRQ sur voie 0 et acquittement
3. IRQ sur voie 1 puis sur voie 0 et acquittement (priorité en différencié)
4. IRQ sur voie 1 et 0 et acquittement (priorité en simultané)
5. IRQ sur une voie puis acquittement avec la voie toujours active puis passage à l'état bas de la voie (test acquittement et de l'irq uniquement sur front montant)

Pour chaque test, des asserts contrôlent les valeurs de VICPC et IRQ.

**Chronogramme modelsim :**



VICPC est affiché en décimal.

## 6.2 : Modification du décodeur d'instruction

### Description

Fichier:	src/instructions_decoder_VIC.vhd	Entité:	INSTRUCTION_DECODER_VIC
<pre>entity INSTRUCTION_DECODER_VIC is   port(     Instruction:  in std_logic_vector (31 downto 0);     RegPSR:      in std_logic_vector (31 downto 0);      nPCsel:      out std_logic;     RegWr:       out std_logic;     RegSel:      out std_logic;     ALUsrc:      out std_logic;     MemWr:       out std_logic;     ALUCtr:      out std_logic_vector (2 downto 0);     PSREN:       out std_logic;     WrSrc:       out std_logic;     LRen:        out std_logic;     IRQ_END:     out std_logic;     RegAff:      out std_logic   ); end entity;</pre>		<b>INSTRUCTION_DECODER_VIC:inst_ID</b> <ul style="list-style-type: none"> <li>+ ALUCtr[2..0]</li> <li>- ALUsrc</li> <li>- IRQ_END</li> <li>- LRen</li> <li>- MemWr</li> <li>- PSREN</li> <li>- RegAff</li> <li>- RegSel</li> <li>- RegWr</li> <li>- WrSrc</li> <li>- nPCsel</li> </ul>	

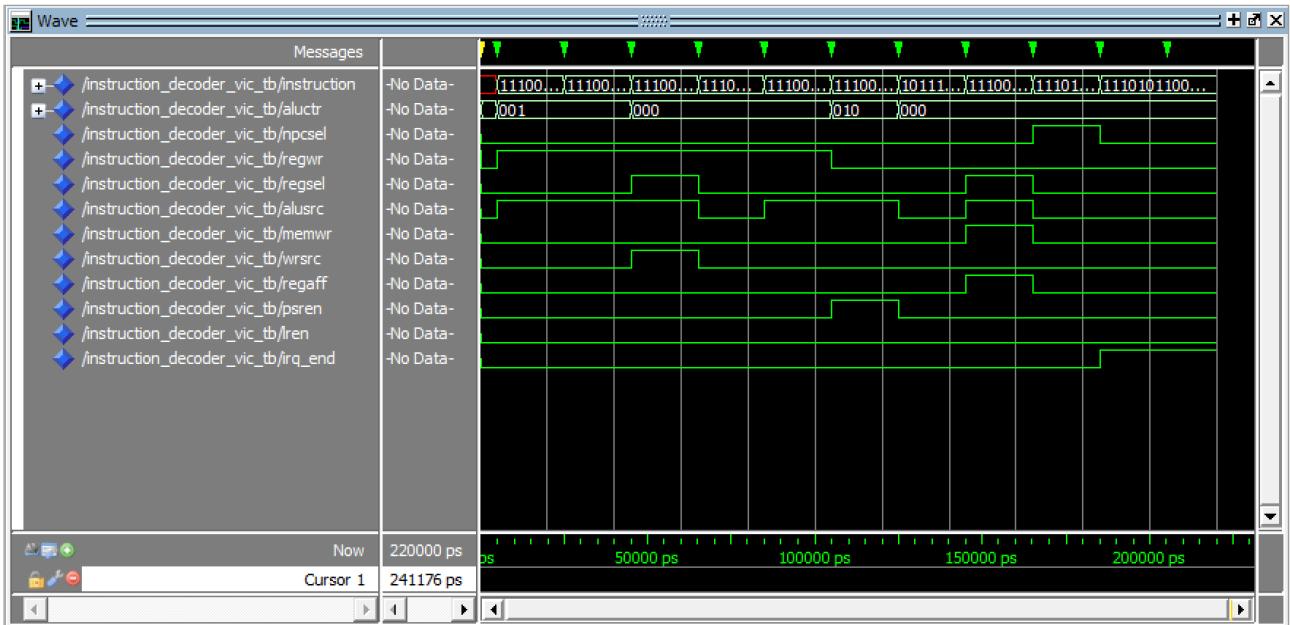
Le décodeur d'instruction est modifié pour permettre la gestion des fin d'instructions. Pour cela on ajoute le cas BX pour l'instruction "EB000000". Pour les instructions BX, toutes les commandes sont à zéro hormis IRQ-END. IRQ-END est à zéro pour toutes les autres instructions.

Une sortie IRQ-END est ajoutée au composant.

### Simulation

Le test\_bench est identique au précédent et ajoute simplement une instruction BX à la fin.

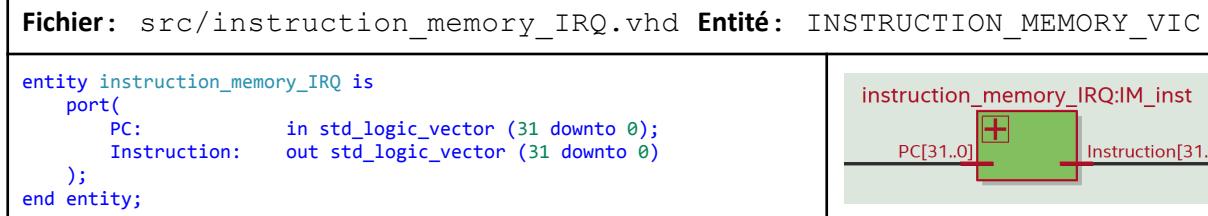
### Chronogramme modelsim :



## 6.3 : Modification de l'Unité de gestion des Instructions

### Instruction Memory

#### Description



Le code de ce composant est également fourni. Un test bench n'est pas nécessaire. L'analyse du code assembleur nous apprend que ce programme possède un main identique au programme précédent et possède 2 interrupts handler.

Les interruptions sauvegardent tout d'abord le contexte (R4 et R5) dans la mémoire au pointeur de pile (R15) en l'incrémentant.

Ensuite ils chargent la valeur du premier emplacement parcouru par le main et l'incrémentent de 1 pour l'IRQ 0 ou 2 pour l'IRQ 1.

Le contexte est ensuite restauré et le pointeur de pile décrémenté.

J'ai modifié l'IRQ1 pour faire -1 au lieu de +2.

### PC Update Unit

#### Description

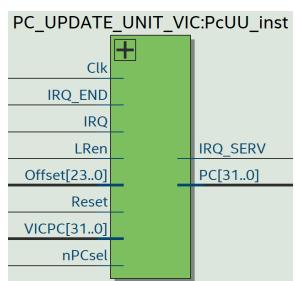
<b>Fichier:</b> src/PC_update_unit_VIC.vhd	<b>Entité:</b> PC_UPDATE_UNIT_VIC
--	-----------------------------------

```

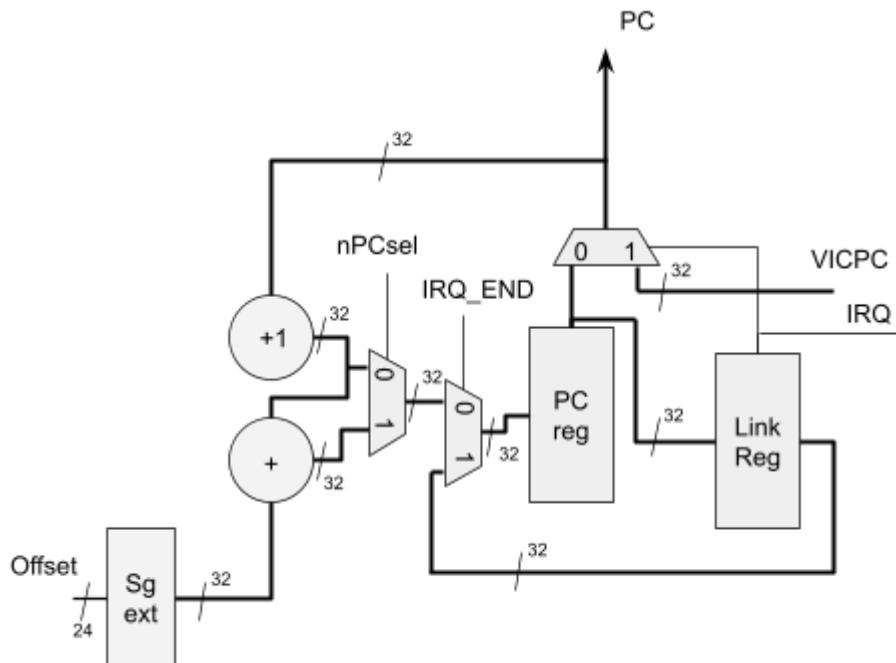
entity PC_UPDATE_UNIT_VIC is
  port(
    Clk : in STD_LOGIC;
    Reset : in STD_LOGIC;
    nPCsel : in STD_LOGIC;
    Offset : in std_logic_vector (23 downto 0);
    LRen : in STD_LOGIC;
    VICPC: in std_logic_vector (31 downto 0);
    IRQ : in STD_LOGIC;
    IRQ_END : in STD_LOGIC;

    PC : out std_logic_vector (31 downto 0);
    IRQ_SERV : out STD_LOGIC
  );
end entity;

```



Pour implémenter les branchements de la PC update unit, on utilise des multiplexeurs et un registre 32 bits (voir partie 3 reg PSR) pour le Link Register. Ces éléments sont agencés comme sur le schéma ci-dessous.

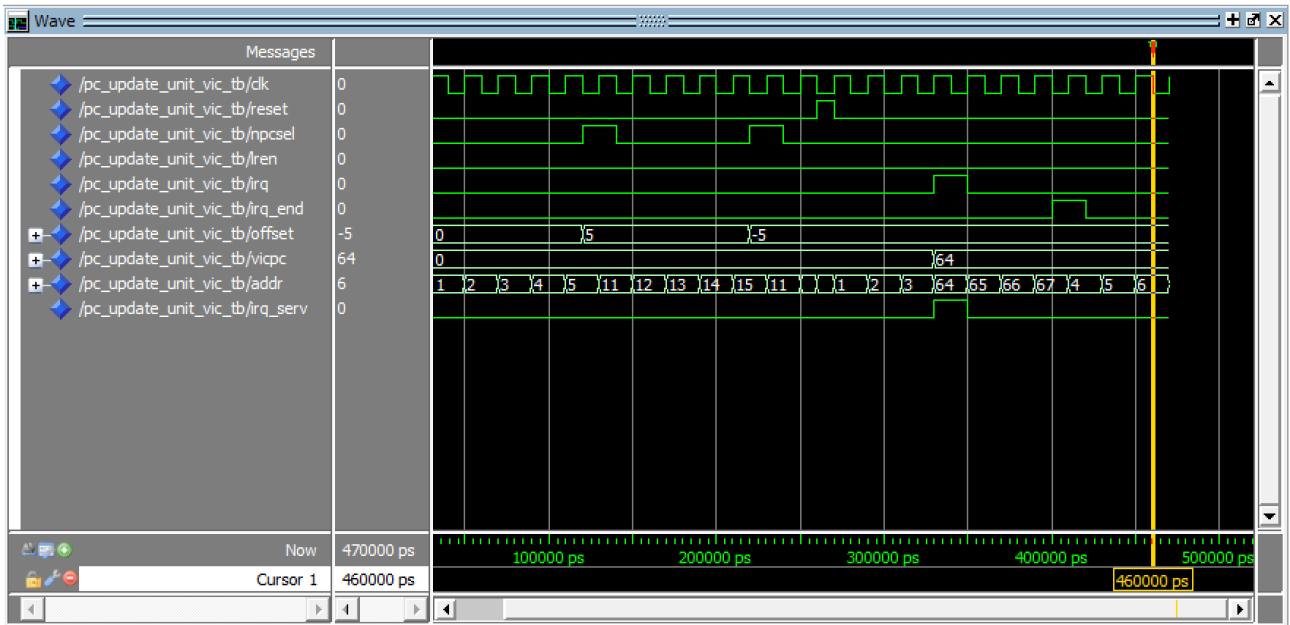


Lorsque IRQ et IRQ-END sont à l'état bas, le fonctionnement est inchangé. Lorsque IRQ passe à 1, PC prend immédiatement la valeur de VICPC. La valeur en sortie du registre PC est stockée dans le Link Register. Dès que IRQ-END passe à un, l'entrée du registre PC reçoit alors le contenu du LR. Ainsi, on repartira de ce point à la prochaine instruction.

### Simulation

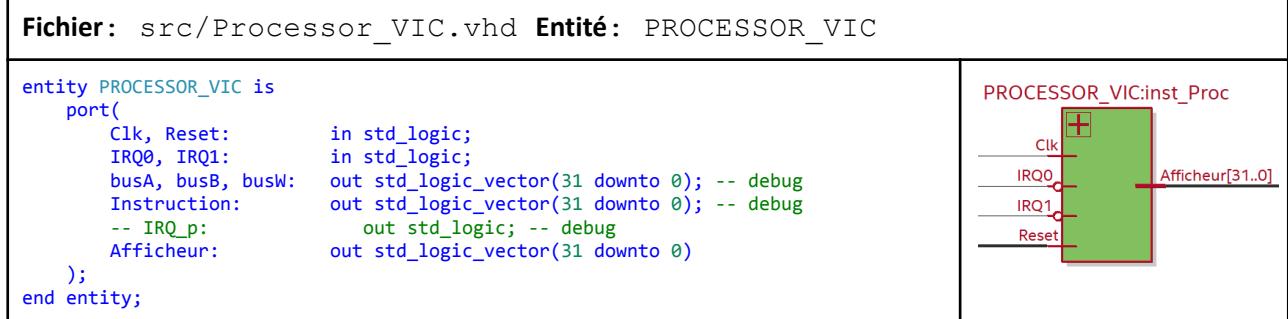
Pour ce test, on reprend le test bench du VIC et on y ajoute un test de l'IRQ : un branchement à 64 puis un retour à la valeur avant l'appel.

### Chronogramme modelsim :



## 6.4 : Modification du processeur & simulation

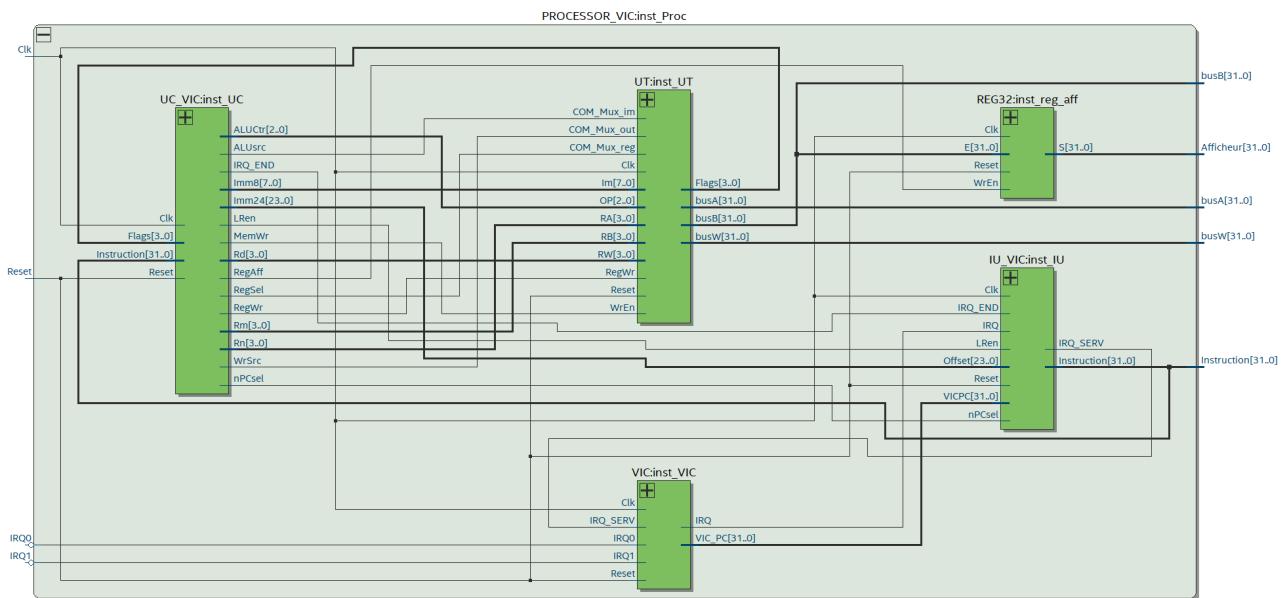
### Description



Préalablement, un composant unité d'instruction VIC a été créé pour rassembler le PC update unit et la mémoire d'instruction modifiée.

L'assemblage du processeur avec VIC est la modification du processeur de la partie 4 en y ajoutant les VIC, en remplaçant les composants modifiés par leur nouvelle version et en ajoutant les nouveaux signaux. Les ports pour les entrées *IRQ0* et *IRQ1* sont également ajoutés.

Schéma interne du composant :



## Simulation

Le test bench du nouveau processeur consiste à déclencher une IRQ puis de laisser le processeur tourner jusqu'à ce que la nouvelle valeur effectivement incrémentée s'affiche sur la sortie *Afficheur*.

**Chronogramme modelsim :**

## 6.4 : Test du VIC sur FPGA

### Top Level

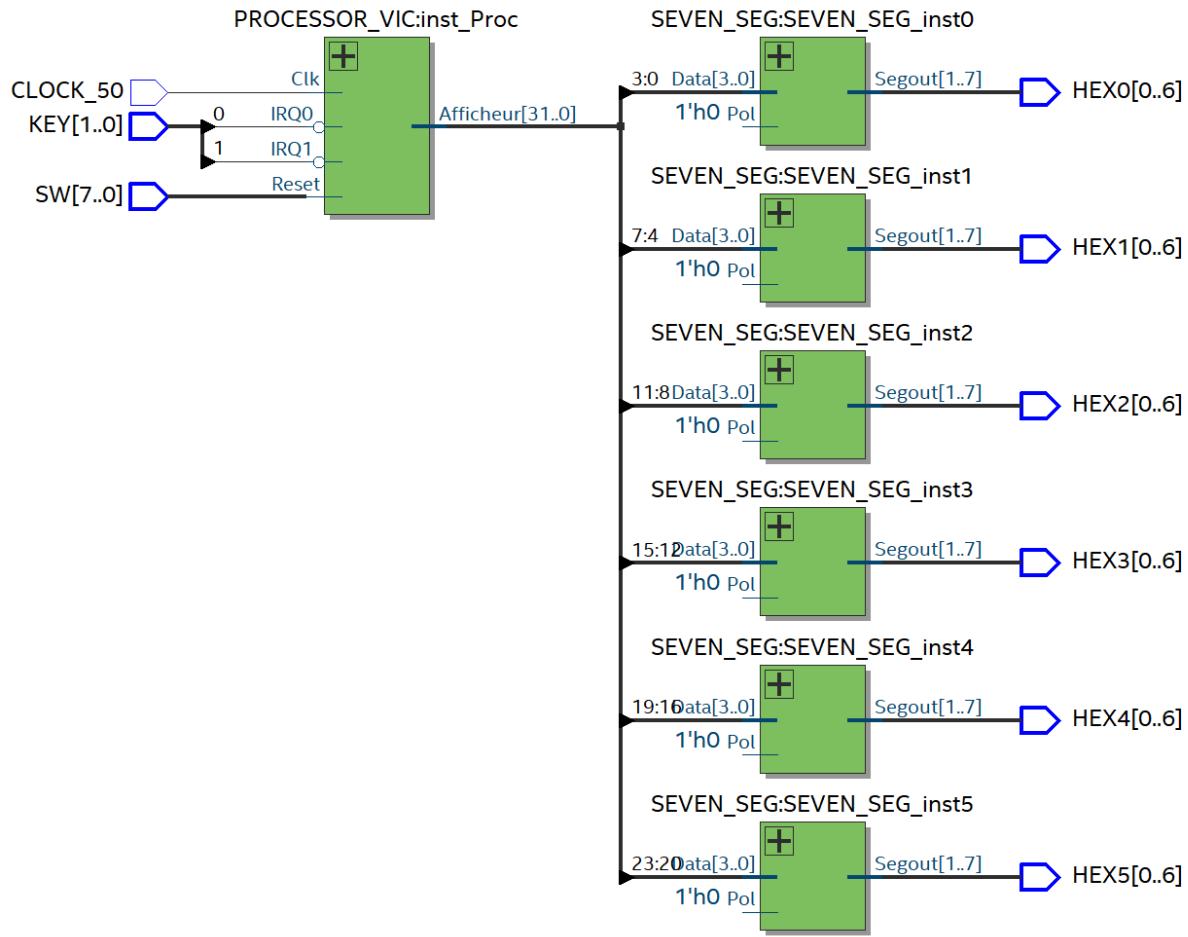
**Fichiers:** src/Quartus/TopLevel\_VIC.vhd

**Entité:** TopLevel\_VIC

Pour le Top Level du processeur avec VIC, on reprend exactement le fichier du Top Level précédent.

Pour pouvoir déclencher les interruptions, on relie les entrées *IRQ* du Processeur aux boutons de la carte, on doit alors déplacer le *reset*. J'ai fait le choix de placer le *reset* sur le switch 0. Cela n'est pas idéal car le *reset* n'a pas vocation à rester enfoncé mais permet néanmoins de garder les 2 *irq* sur les boutons poussoirs, la carte n'en ayant que 2.

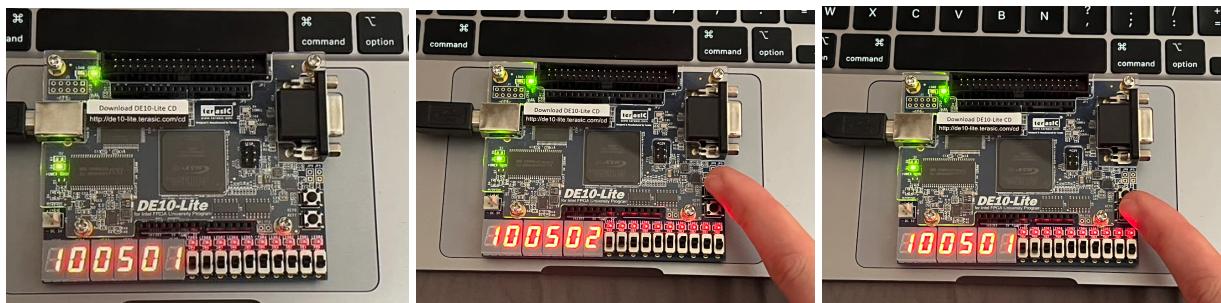
Schéma :



## Test sur Carte FPGA

Une fois le projet compilé, on peut le charger sur la carte. On voit alors s'afficher sur les 7-segment la somme des registres parcourus par le main. Lorsque l'on appuie sur le bouton 0 (**IRQ0**) on observe la somme s'incrémenter de 1. Lorsque l'on appuie sur le bouton 1 (**IRQ1**) on observe la somme se décrémenter de 1 (modification faite de la mémoire d'instruction).

### Validation par le professeur en classe



# Partie 7.A - Périphérique UART TX

## Composants Uart

**Fichiers :** src/UART/uart\_tx.vhd  
src/UART/fdiv.vhd

**Entité :** UART\_TX  
FDIV

Les composants utilisés pour l'uart proviennent du TP précédent ou ils ont été décrits et présentés plus en détail. On ne fera ici qu'une présentation sommaire et on les utilisera en l'état.

### UART\_TX :

Ce module envoie des données en série via une interface UART. Il prend des données parallèles de 8 bits (*Data*), commence la transmission avec le signal *Go*, et sort les données série sur *Tx*. Le signal *tx\_busy* indique si le module est en train de transmettre. Le signal *IRQ\_TX* indique que la transmission est terminée. Les signaux *Parity* et *Even* permettent de gérer le bit de parité mais ne sont pas utilisés ici.

### FDIV :

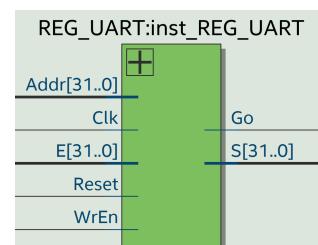
Ce module divise une fréquence d'horloge d'entrée (*clk*) pour produire une fréquence d'horloge de sortie plus basse (*Tick*). Il est utilisé pour ajuster les fréquences d'horloge pour l'UART. Le signal *reset* permet de réinitialiser le diviseur de fréquence. Les paramètres génériques permettent de choisir la fréquence d'horloge en entrée (50 MHz ici) et deux bauds rate pour la sortie. Le signal *bauds\_rate* permet de passer d'un bauds rate à l'autre simplement.

## Registre UART\_Conf

### Description

**Fichier :** src/UART\_reg.vhd **Entité :** INSTRUCTION\_DECODER\_VIC

```
entity REG_UART is
  port(
    Clk : in STD_LOGIC;
    Reset : in STD_LOGIC;
    WrEn : in STD_LOGIC;
    Addr : in std_logic_vector (31 downto 0);
    E: in std_logic_vector (31 downto 0);
    S: out std_logic_vector (31 downto 0);
    Go : out STD_LOGIC
  );
end entity;
```



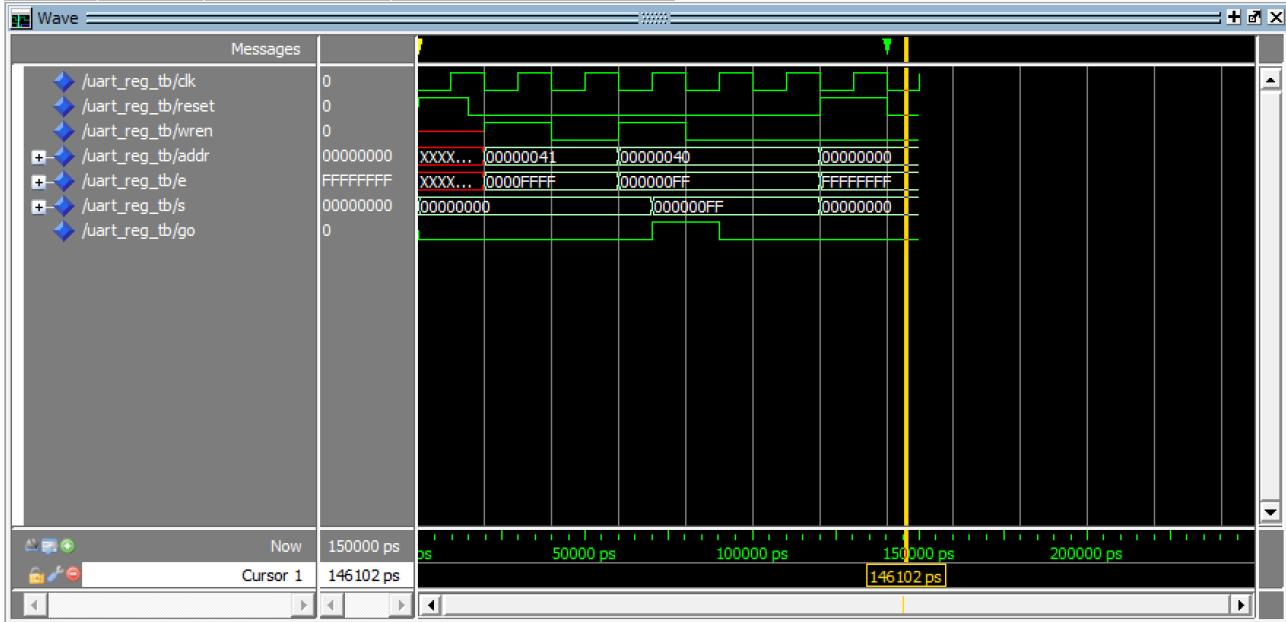
Le registre UART\_Conf est un registre 32 bits (voir partie 3 reg PSR) modifié pour :

- Avoir en entrée le bus d'adresse sur 32 bit
- N'écrire que quand *WrEn* est à l'état haut ET que le bus d'adresse est à 0x40
- Avoir en sortie un signal *go* qui passa à l'état haut lors de l'écriture

## Simulation

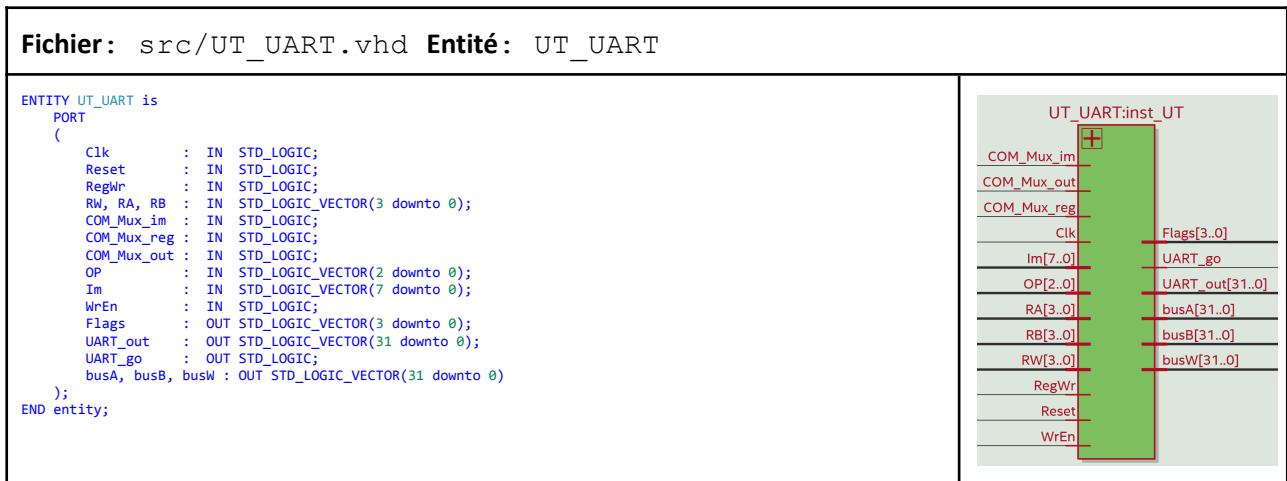
Le test bench teste après le *reset* une tentative d'écriture avec le bus d'adresse différent de 0x40 et une écriture avec le bus d'adresse à 0x40 puis un reset. A chaque étape, les valeurs de *go* et *s* sont contrôlées avec des asserts.

### Chronogramme modelsim :



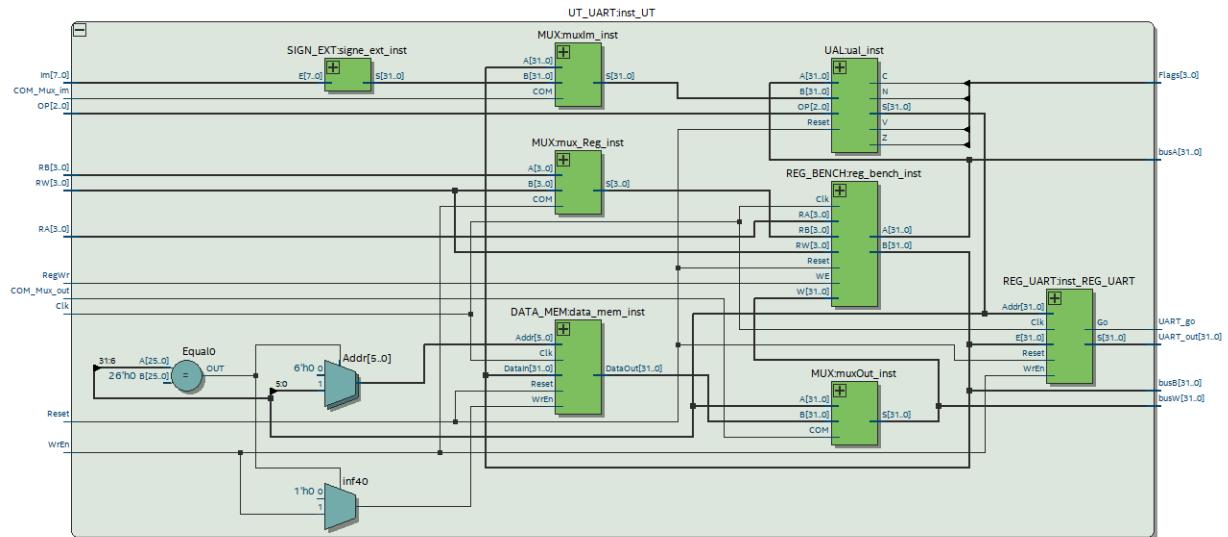
## Modification de l'UT : Ajout d'une sortie UART\_TX

### Description



Les ports pour la sortie de l'uart et le go sont ajoutés et reliés à à un registre *UART\_Conf* instancié dans l'ut. Pour éviter de déclencher l'écriture en mémoire non désirée, un décodage d'adresse a été mis en place. Un process sensible sur *WrEn* et le bus d'adresse de la mémoire sur 32 bits (*ALUout*) est ajouté ainsi qu'un signal intermédiaire remplaçant *WrEn* en entrée de la mémoire. Lorsque la valeur de l'adresse est inférieure à 40 (bits 31 à 6 à '0'), *WrEn* est transmis à la mémoire (signal intermédiaire à *WrEn*). Sinon, *WrEn* est bloqué (signal intermédiaire à '0')

Schéma interne du composant :

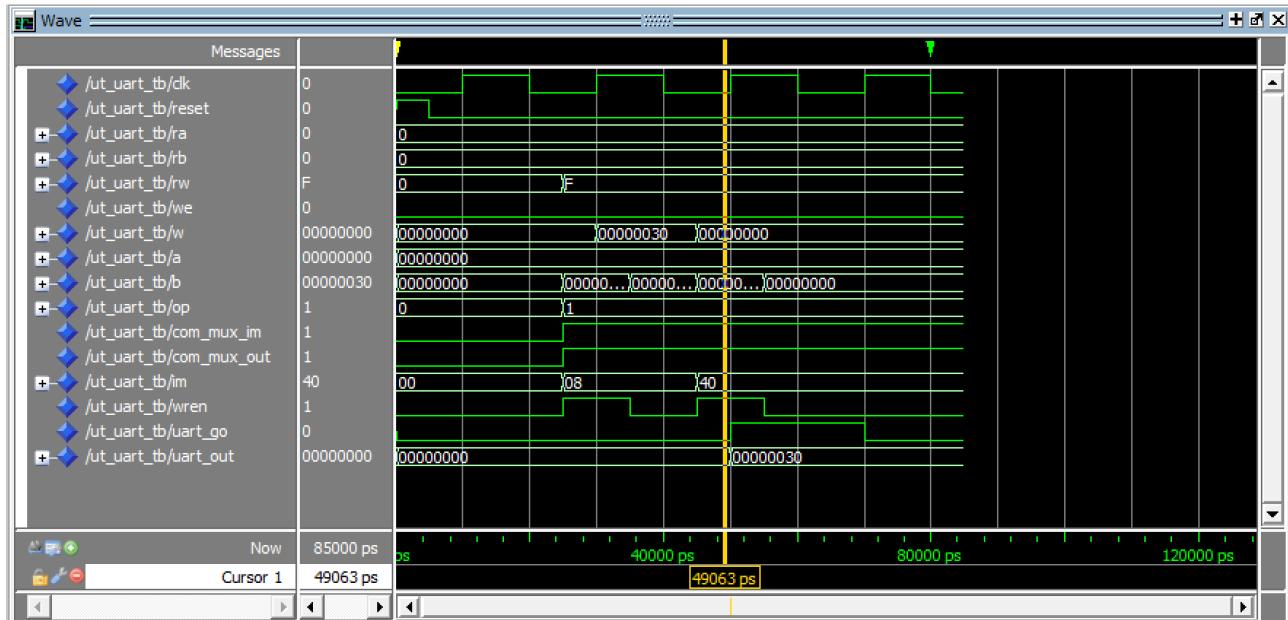


## Simulation

Le test écrit une valeur en mémoire comme précédemment (visualisable sur le bus W) puis à l'emplacement 0x40.

On vérifie avec des asserts que l'écriture en mémoire se fait toujours correctement et que l'écriture sur l'UART et le go se font également correctement sans écrire dans la mémoire.

## Chronogramme modelsim :



## Modification de l'Instruction Memory : envoi d'un caractère sur IRQ

### Description

**Fichier:** src/instruction\_memory\_UART1.vhd **Entité:** instruction\_memory\_UART1

<pre>entity instruction_memory_UART1 is   port(     PC:          in std_logic_vector (31 downto 0);     Instruction: out std_logic_vector (31 downto 0)   ); end entity;</pre>	
--	--

Le code assembleur de la mémoire d'instruction est modifié au niveau de l'IRQ 0 pour pouvoir charger la valeur d'un caractère depuis l'emplacement 0x10 de la mémoire puis l'écrire sur l'uart (emplacement 0x40).

```
-- ISR 0 : interruption 0 : envoi d'un caractère
-- sauvegarde du contexte
ram_block(9) := x"E60F1000"; -- STR R1,0(R15) ; --MEM[R15] <= R1
ram_block(10) := x"E28FF001"; -- ADD R15,R15,1 ; --R15 <= R15 + 1
ram_block(11) := x"E60F3000"; -- STR R3,0(R15) ; --MEM[R15] <= R3
--traitement
ram_block(12) := x"E3A05010"; -- MOV R5,0x10 ; --R5 <= 0x10
ram_block(13) := x"E6151000"; -- LDR R1,0(R5) ; --R1 <= MEM[R5]
ram_block(14) := x"E3A03040"; -- MOV R3,0x40 ; --R3 <= 0x40
ram_block(15) := x"E6031000"; -- STR R1,0(R3) ; --MEM[R3] <= R1
-- restauration du contexte
ram_block(16) := x"E61F3000"; -- LDR R3,0(R15) ; --R3 <= MEM[R15]
ram_block(17) := x"E28FF0FF"; -- ADD R15,R15,-1 ; --R15 <= R15 - 1
ram_block(18) := x"E61F1000"; -- LDR R1,0(R15) ; --R1 <= MEM[R15]
ram_block(19) := x"EB000000"; -- BX ; -- instruction de fin d'interruption
ram_block(20) := x"00000000";
```

## Test sur carte intermédiaire

### Description Processeur

Fichier:	src/Processor_UART1.vhd	Entité:	PROCESSOR_UART1
<pre>entity PROCESSOR_UART1 is   port(     Clk, Reset:      in std_logic;     IRQ0, IRQ1:      in std_logic;     UART_go:         out std_logic;     UART_out:        out std_logic_vector(31 downto 0);     busA, busB, busW: out std_logic_vector(31 downto 0); -- debug     Instruction:     out std_logic_vector(31 downto 0); -- debug      Afficheur:        out std_logic_vector(31 downto 0)   ); end entity;</pre>			

L'assemblage du processeur UART\_TX1 est la modification du processeur précédent en remplaçant les composants modifiés par leur nouvelle version et en ajoutant les nouveaux signaux. Les ports pour l'entrée *IRQ\_TX* ainsi que les sorties *IRQ\_TX* et *UART\_out* sont également ajoutés.

### Simulation

Pour tester ce processeur, on déclenche son *IRQ0* puis on le laisse tourner une dizaine de tours d'horloge pour voir le caractère en sortie accompagné du go.

### Chronogramme modelsim :

## Top Level

Fichier: src/Quartus/TopLevel\_UART1.vhd

Entité: TopLevel\_UART2

On reprend le Top Level précédent en y ajoutant le FDIV et le composant de transmission *UART\_TX*. La sortie *TX* est reliée au pin *GPIO\_0* de la carte.

### Test sur carte

Une fois le projet Quartus compilé, on le charge sur la carte. Le programme précédent est toujours fonctionnel. Pour observer la transmission du caractère, on dispose d'une interface série/USB. En utilisant le logiciel PUTTY, on observe effectivement que des caractères sont envoyés à chaque appuis du bouton 0 (*IRQ\_0*).



### Validation par le professeur en classe

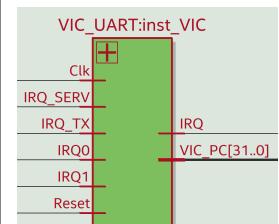
## Modification du VIC

### Description

Fichier: src/VIC\_UART.vhd Entité: VIC\_UART

```
entity VIC_UART is
  port (
    Clk      : in std_logic;
    Reset    : in std_logic;
    IRQ_SERV : in std_logic;
    IRQ0, IRQ1 : in std_logic;
    IRQ_TX   : in std_logic;

    IRQ      : out std_logic;
    VIC_PC  : out std_logic_vector(31 downto 0)
  );
end VIC_UART;
```

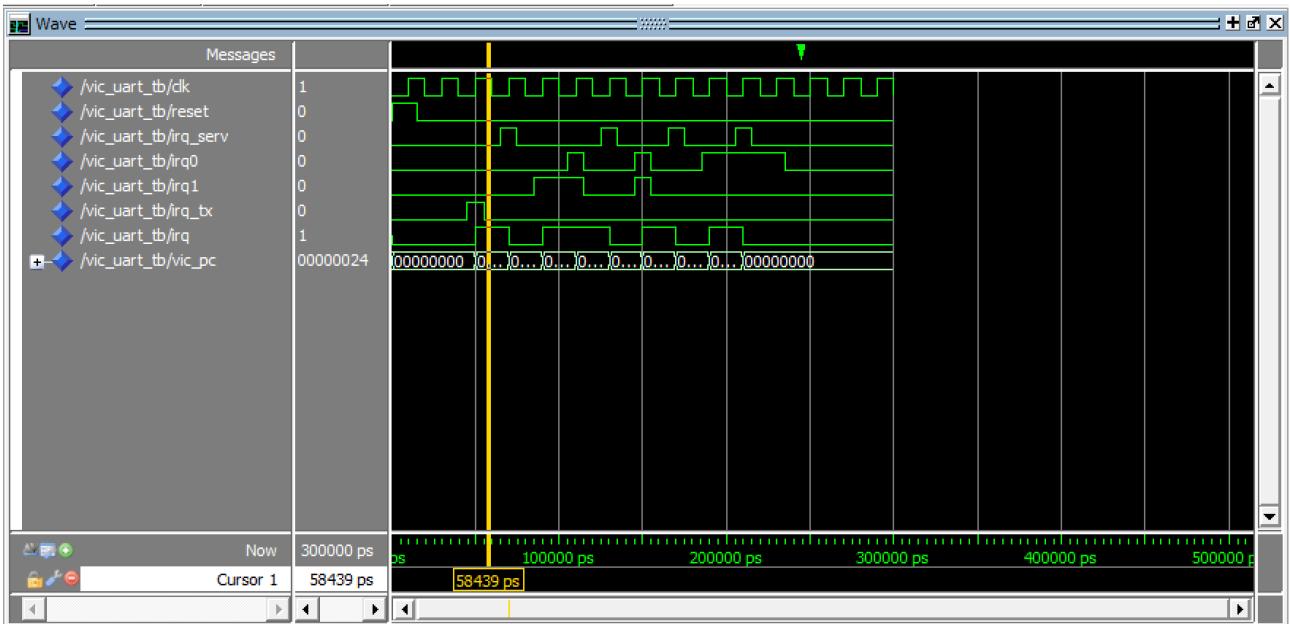


Le VIC a été modifié pour ajouter une IRQ pour la transmission. Le fonctionnement est identique au précédent, la priorité de la nouvelle IRQ est maximale.

### Simulation

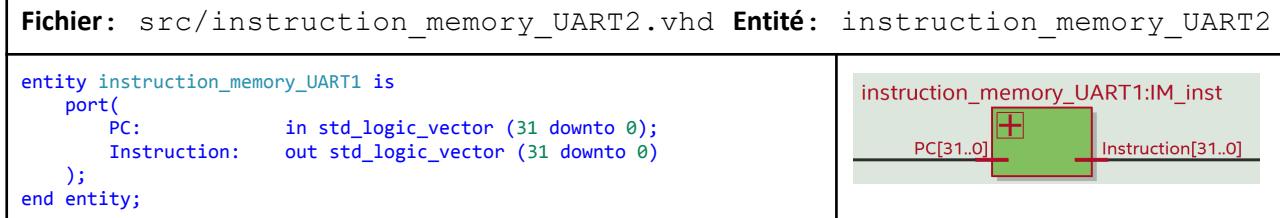
La simulation de ce composant est identique mais avec un test sur *IRQ\_TX*.

### Chronogramme modelsim :



## Modification de l'Instruction Memory : envoi d'une chaîne de caractères

### Description



```
--sauvegarde du contexte
ram_block(9) := x"E60F1000"; -- STR R1,0(R15) ; --MEM[R15] <= R1
ram_block(10) := x"E28FF001"; -- ADD R15,R15,1 ; --R15 <= R15 + 1
ram_block(11) := x"E60F3000"; -- STR R3,0(R15) ; --MEM[R15] <= R3
--traitement
ram_block(12) := x"E3A08008"; -- MOV R8,0x08 ; --R8 <= 0x08
ram_block(13) := x"E3A07001"; -- MOV R7,0x01 ; --R7 <= 0x01
ram_block(14) := x"E3A0A040"; -- MOV R10,0x40 ; --R10 <= 0x40
ram_block(15) := x"E617900F"; -- LDR R9,15(R7) ; --R9 <= MEM[15(R7)]
ram_block(16) := x"E60A9000"; -- STR R9,0(R10) ; --MEM[R10] <= R9
ram_block(17) := x"E6087000"; -- STR R7,0(R8) ; --MEM[R8] <= R7
-- restauration du contexte
ram_block(18) := x"E61F3000"; -- LDR R3,0(R15) ; --R3 <= MEM[R15]
ram_block(19) := x"E28FF0FF"; -- ADD R15,R15,-1 ; --R15 <= R15 - 1
ram_block(20) := x"E61F1000"; -- LDR R1,0(R15) ; --R1 <= MEM[R15]
ram_block(21) := x"EB000000"; -- BX ; -- instruction de fin d'interruption
ram_block(22) := x"00000000";
```

```
-- ISR_TX : interruption TX
--sauvegarde du contexte - R15 correspond au pointeur de pile
ram_block(36) := x"E60F4000"; -- STR R4,0(R15) ; --MEM[R15] <= R4
ram_block(37) := x"E28FF001"; -- ADD R15,R15,1 ; --R15 <= R15 + 1
ram_block(38) := x"E60F5000"; -- STR R5,0(R15) ; --MEM[R15] <= R5
--traitement
ram_block(39) := x"E3A08008"; -- MOV R8, 0x08
```

```

ram_block(40) := x"E3A09040"; -- MOV R9, 0x40
ram_block(41) := x"E6187000"; -- LDR R7, 0(R8)
ram_block(42) := x"E357000C"; -- CMP R7, 0xC
ram_block(43) := x"BA000001"; -- BLT -tx
ram_block(44) := x"EA000003"; -- BAL -end
ram_block(45) := x"E617A010"; -- -tx: LDR R10, 0x10(R7)
ram_block(46) := x"E609A000"; -- STR R10, 0(R9)
ram_block(47) := x"E2877001"; -- Addi R7, R7, 0x1
ram_block(48) := x"E6087000"; -- - end : STR R7, 0(R8)
-- restauration du contexte
ram_block(49) := x"E61F5000"; -- LDR R5,0(R15) ; --R5 <= MEM[R15]
ram_block(50) := x"E28FF0FF"; -- ADD R15,R15,-1 ; --R15 <= R15 - 1
ram_block(51) := x"E61F4000"; -- LDR R4,0(R15) ; --R4 <= MEM[R15]
ram_block(52) := x"EB000000"; -- BX ; -- instruction de fin d'interruption

```

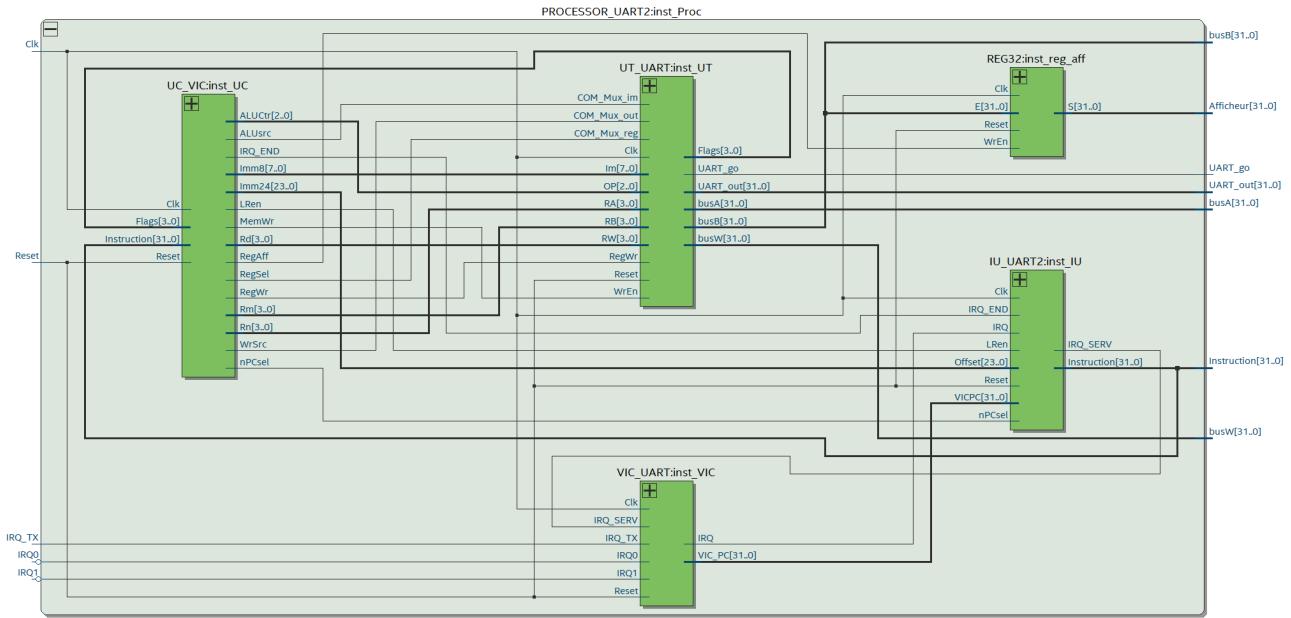
## Test sur carte complet

### Description Processeur

Fichier:	Entité:
<pre> entity PROCESSOR_UART2 is   port(     Clk, Reset:      in std_logic;     IRQ0, IRQ1:      in std_logic;     IRQ_TX :         in std_logic;     UART_go:         out std_logic;     UART_out:        out std_logic_vector(31 downto 0);     busA, busB, busW: out std_logic_vector(31 downto 0); -- debug     Instruction:    out std_logic_vector(31 downto 0); -- debug     -- IRQ_p:          out std_logic; -- debug     Afficheur:       out std_logic_vector(31 downto 0)   ); end entity; </pre>	<b>PROCESSOR_UART2:inst_Proc</b>

L'assemblage du processeur UART\_TX2 est la modification du processeur précédent en remplaçant les composants modifiés par leur nouvelle version et en ajoutant les nouveaux signaux.

Schéma interne du composant :

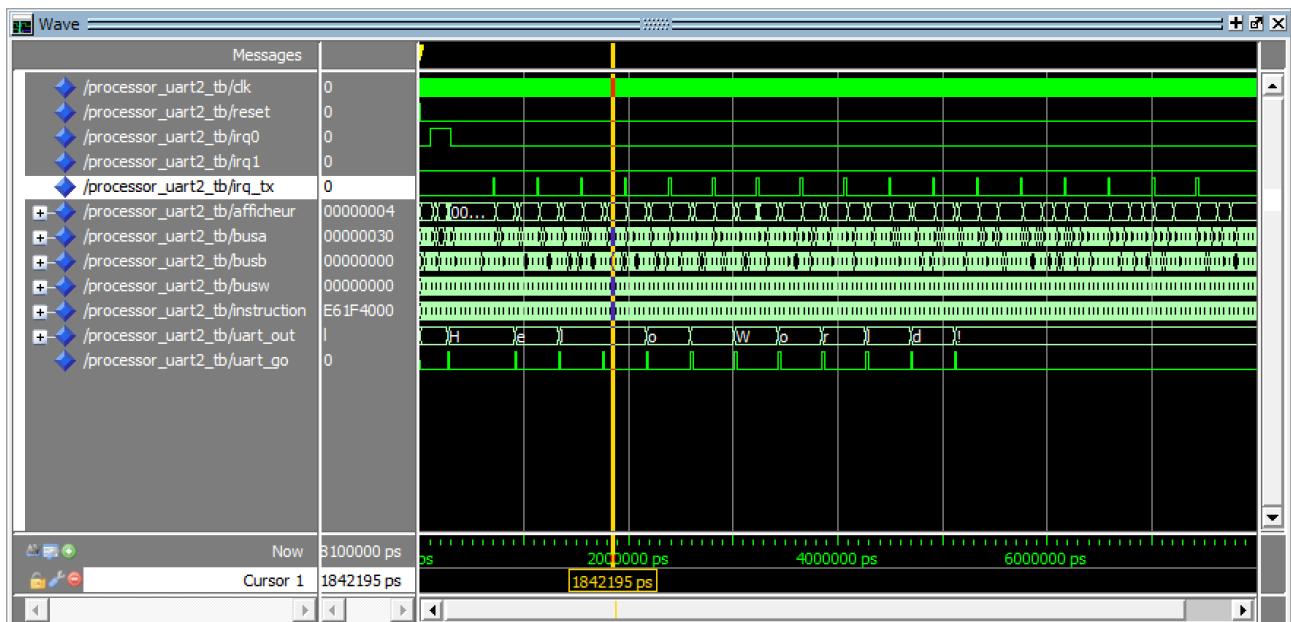


## Simulation

Pour tester ce processeur, on déclenche son IRQ0 puis on le laisse tourner une dizaine de tours d'horloge pour voir le premier caractère en sortie accompagné du go.

On simule ensuite la fin d'émission en levant l'IRQ\_TX à intervalles de temps suffisamment espacées pour simuler les fonctionnement de l'UART. On observe alors les caractères s'envoyer à la suite jusqu'au dernier, qui ne déclenche plus d'envoi.

### Chronogramme modelsim :



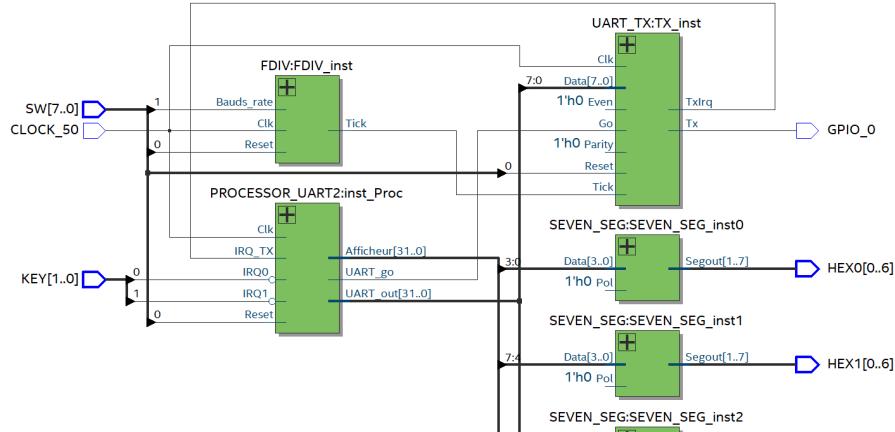
## Top Level

**Fichier:** src/Quartus/TopLevel\_UART2.vhd

**Entité:** TopLevel\_UART2

On reprend à nouveau le code du Top Level précédent en reliant cette fois le signal de fin de transmission à l'*irq\_tx* du processeur.

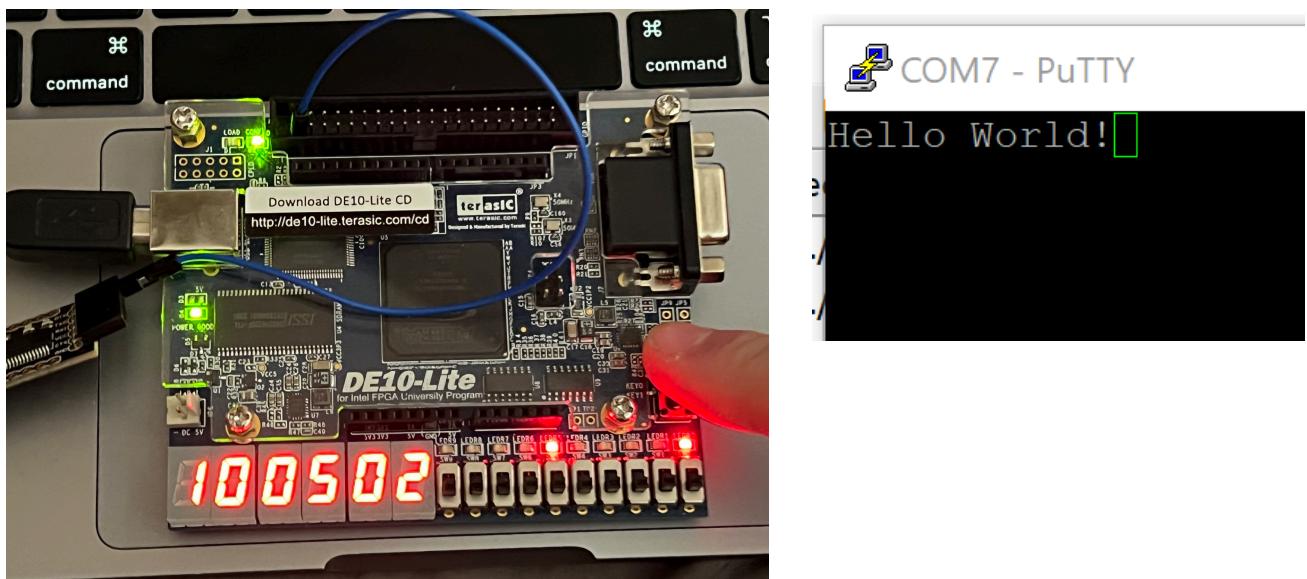
Schéma :



## Test sur carte

On utilise à nouveau PUTTY pour visualiser la sortie sur le bus UART. A chaque déclenchement de l'IRQ\_0, on observe bien l'envoi de la chaîne de caractère "Hello World!".

### Validation par le professeur en classe

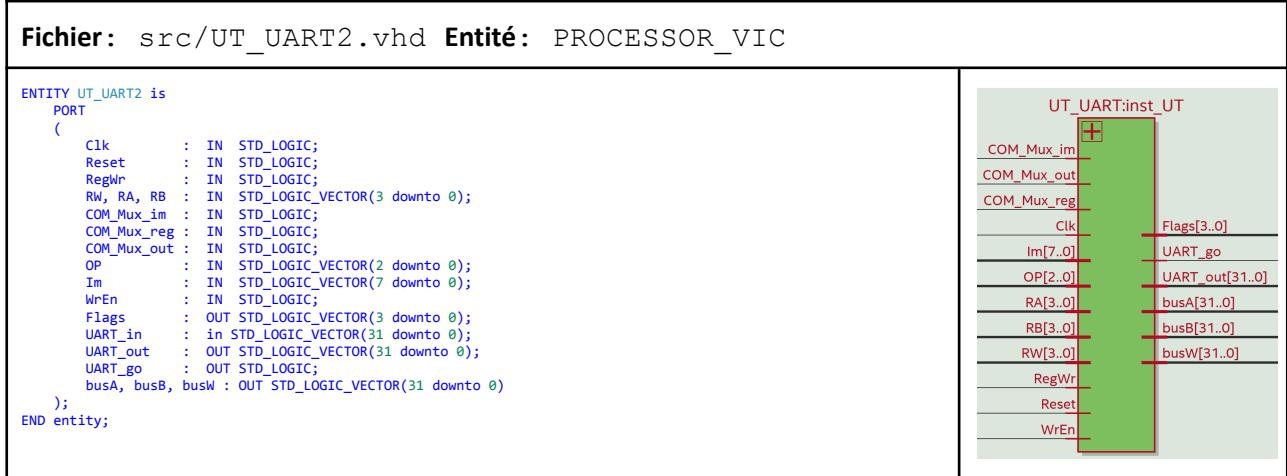


Test de l'envoi d'une chaîne de caractère et réception sur PUTTY

# Partie 7.B - Périphérique UART RX

## Modification de l'UT : Ajout d'une entrée UART\_RX

### Description



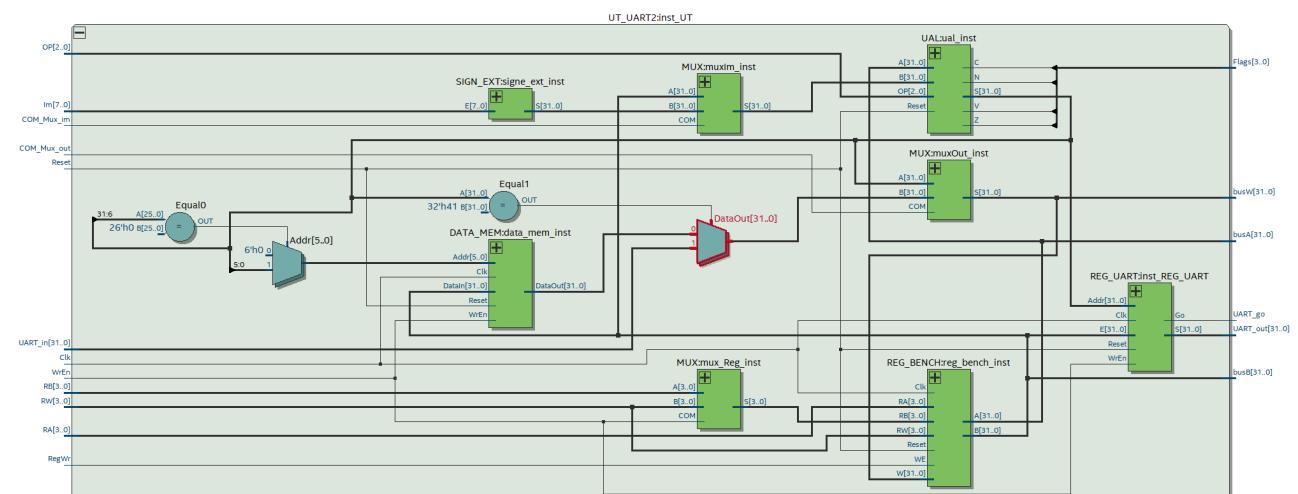
Pour adapter l'UT à la réception de l'UART, on procède de manière similaire à la transmission. On commence par ajouter un bus *uart\_in* pour la valeur reçue.

La lecture se fait de manière similaire à la transmission : Pour accéder aux données reçues, on peut lire l'emplacement mémoire 0x41. On ajoute un multiplexeur en sortie de la mémoire de données. Ce mux est commandé par un process qui décode l'adresse :

Quand le bus d'adresse est différent de 0x41, le mux laisse passer la sortie de la mémoire, le fonctionnement est nominal.

Quand le bus d'adresse est à 0x41, le mux laisse alors passer le bus *uart\_in*, l'UT se comporte alors comme si la valeur provenait de la mémoire de données.

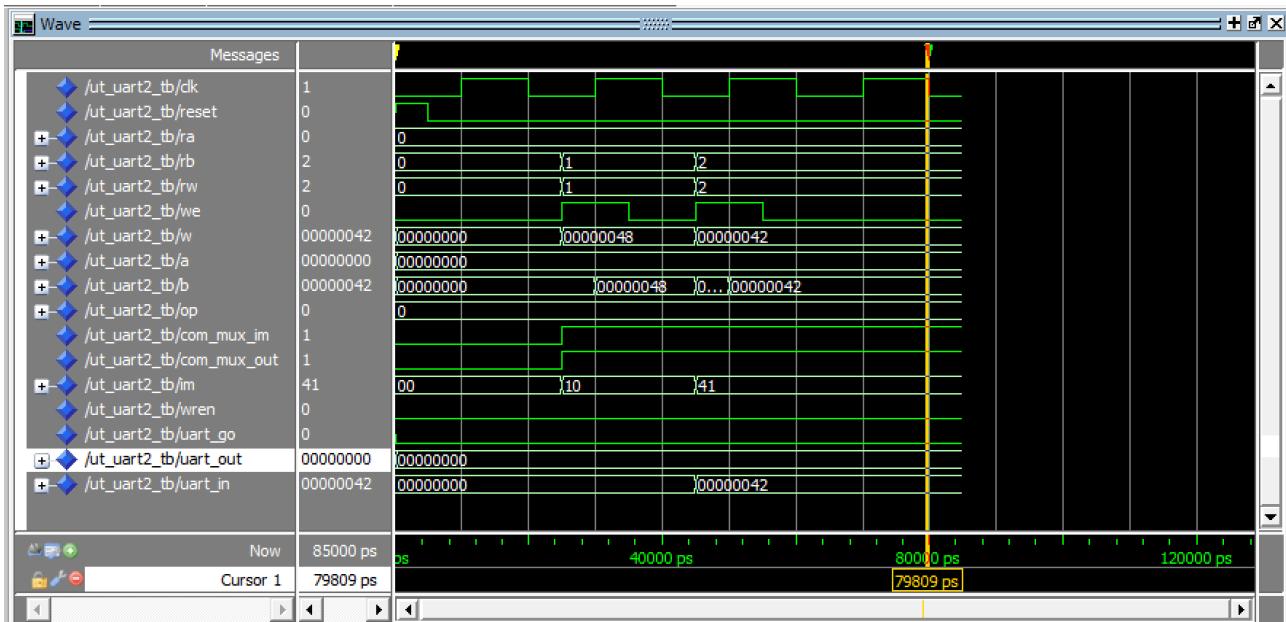
Schéma interne du composant :



## Simulation

Pour tester cette modification, on fourni une donnée sur l'entrée UART puis on fait un STR à l'emplacement 0x41. On vérifie ensuite que la donnée est bien lue sur le bus W (On peut la voir s'écrire en direct dans le banc de registre sur le bus B).

### Chronogramme modelsim :



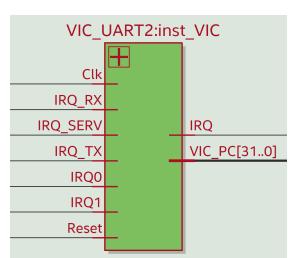
## Modification du VIC

### Description

Fichier: src/VIC\_UART2.vhd Entité: VIC\_UART2

```
entity VIC_UART2 is
  port (
    Clk      : in std_logic;
    Reset    : in std_logic;
    IRQ_SERV : in std_logic;
    IRQ0, IRQ1 : in std_logic;
    IRQ_TX   : in std_logic;
    IRQ_RX   : in std_logic;

    IRQ      : out std_logic;
    VIC_PC  : out std_logic_vector(31 downto 0)
  );
end VIC_UART2;
```

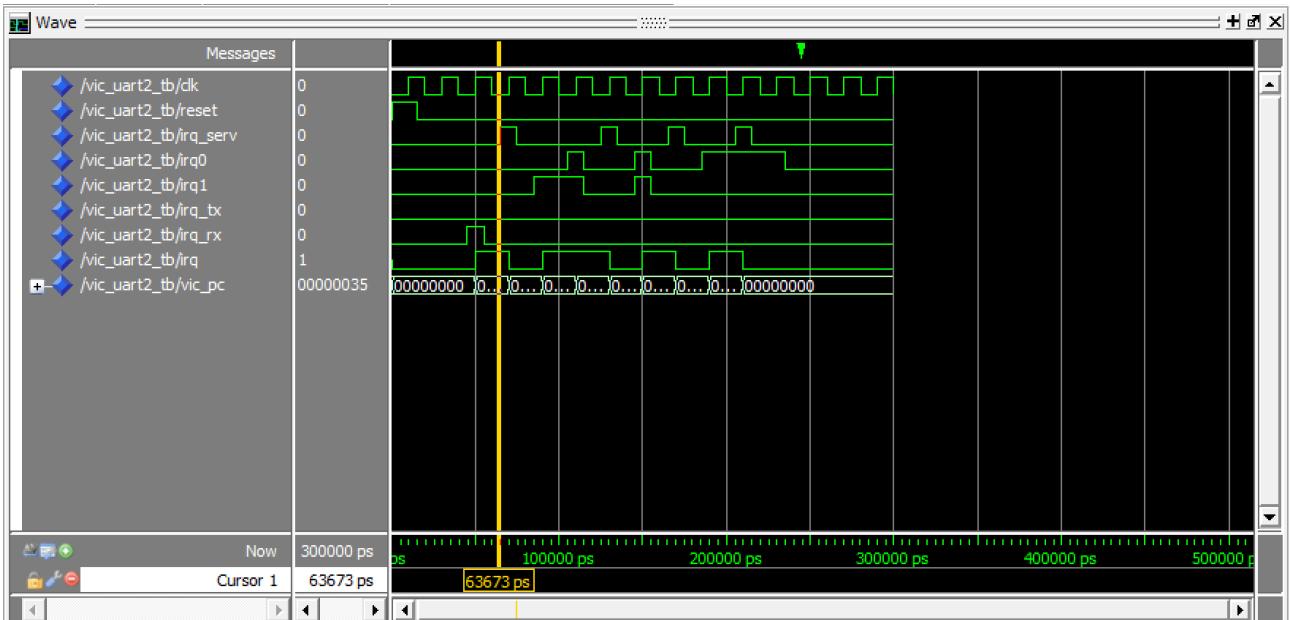


Le VIC a été modifié pour ajouter une IRQ pour la transmission. Le fonctionnement est identique au précédent, la priorité de la nouvelle IRQ est maximale.

## Simulation

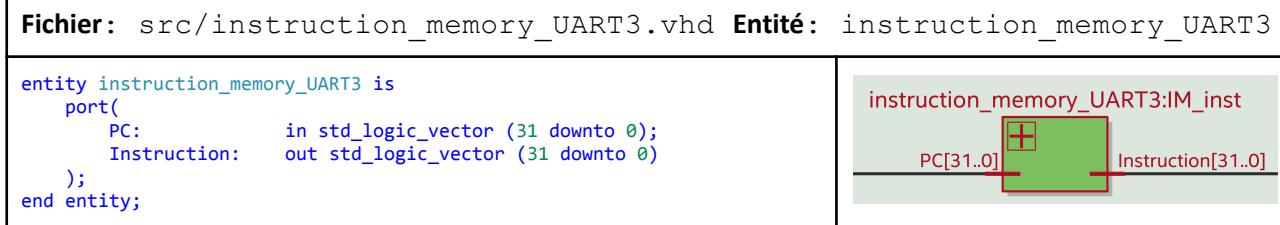
La simulation de ce composant est identique mais avec un test sur *IRQ\_RX*.

### Chronogramme modelsim :



## Modification de l'Instruction Memory : Réception d'un caractère

### Description



Le code assembleur est modifié pour ajouter une interruption lors de la réception d'un caractère sur l'UART :

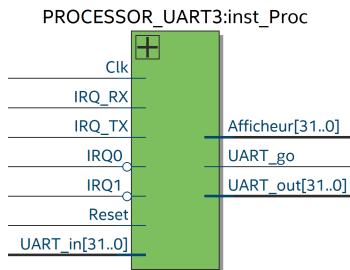
Le main ne boucle plus que sur une valeur. L'IRQ de réception déclenche la lecture du caractère reçu puis son écriture à l'emplacement sur lequel boucle le main. Ainsi, le caractère reçu s'affichera sur le port *Afficheur*.

Voici le code de l'IRQ handler :

```
-- ISR_RX : interruption RX
-- sauvegarde du contexte - R15 correspond au pointeur de pile
ram_block(53) := x"E60F4000"; -- STR R4,0(R15) ; --MEM[R15] <= R4
ram_block(54) := x"E28FF001"; -- ADD R15,R15,1 ; --R15 <= R15 + 1
ram_block(55) := x"E60F5000"; -- STR R5,0(R15) ; --MEM[R15] <= R5
-- traitement
ram_block(56) := x"E3A05041"; -- MOV R5,0x41 ; --R5 <= 0x41
ram_block(57) := x"E6154000"; -- LDR R4,0(R5) ; --R4 <= MEM[R5]
ram_block(58) := x"E3A05001"; -- MOV R5,0x01 ; --R5 <= 0x01
ram_block(59) := x"E6054000"; -- STR R4,0(R5) ; --MEM[R5] <= R4
-- restauration du contexte
ram_block(60) := x"E61F5000"; -- LDR R5,0(R15) ; --R5 <= MEM[R15]
ram_block(61) := x"E28FF0FF"; -- ADD R15,R15,-1 ; --R15 <= R15 - 1
ram_block(62) := x"E61F4000"; -- LDR R4,0(R15) ; --R4 <= MEM[R15]
ram_block(63) := x"EB000000"; -- BX ; -- instruction de fin d'interruption
```

# Test sur carte

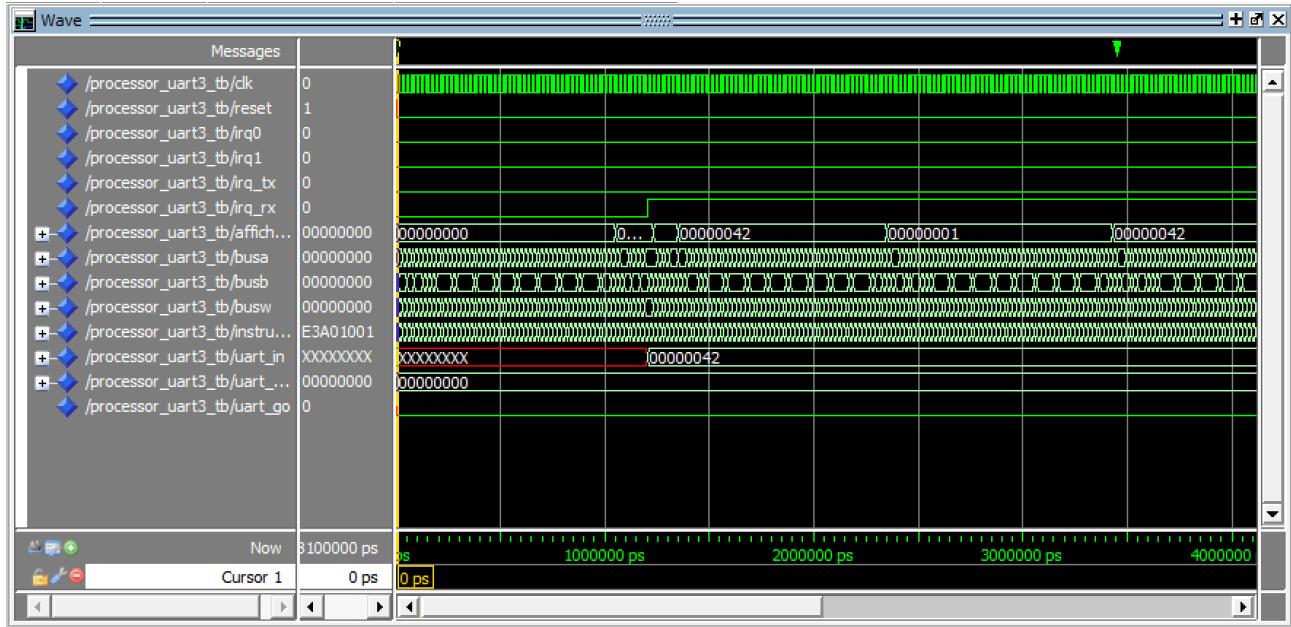
## Description Processeur

Fichier:	src/Processor_UART3.vhd	Entité:	PROCESSOR_UART3
<pre>entity PROCESSOR_UART3 is   port(     Clk, Reset:      in std_logic;     IRQ0, IRQ1:      in std_logic;     IRQ_RX, IRQ_TX : in std_logic;     UART_go:         out std_logic;     UART_in:         in std_logic_vector(31 downto 0);     UART_out:        out std_logic_vector(31 downto 0);     busA, busB, busW: out std_logic_vector(31 downto 0); -- debug     Instruction:    out std_logic_vector(31 downto 0); -- debug     -- IRQ_p:          out std_logic; -- debug     Afficheur:       out std_logic_vector(31 downto 0)   ); end entity;</pre>			

L'assemblage du processeur *UART\_RX* est la modification du processeur précédent en remplaçant les composants modifiés par leur nouvelle version et en ajoutant les nouveaux signaux. Les ports pour les entrées et *IRQ\_RX* et *UART\_in* sont également ajoutés.

## Simulation

### Chronogramme modelsim :

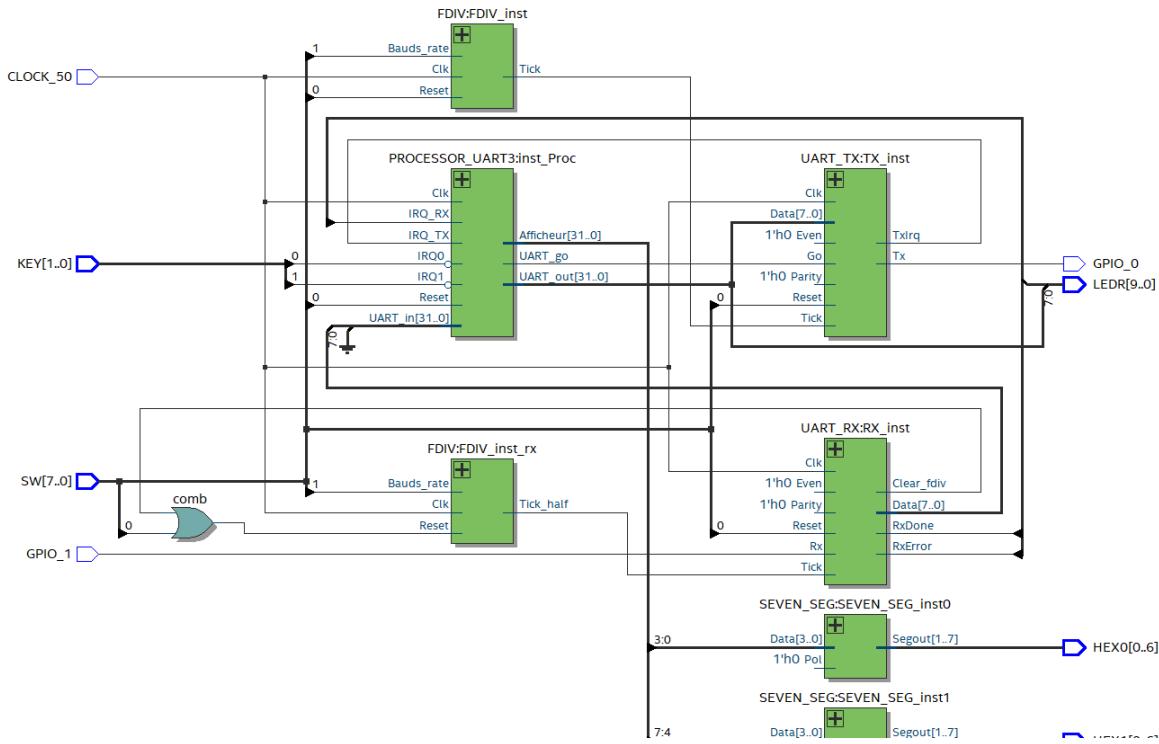


## Top Level

Fichier:	src/Quartus/TopLevel_UART3.vhd	Entité:	TopLevel_UART3
----------	--------------------------------	---------	----------------

La modification de ce Top Level est identique à la modification précédente en ajoutant en plus le composant de réception *UART\_RX* ainsi que son *FDIV* dont le *Reset* est également commandé par le signal de début de réception provenant de *UART\_RX*. Le signal de fin de réception est également relié à l'entrée *IRQ\_rx* du processeur. L'entrée Rx est reliée au pin *GPIO\_1* de la carte.

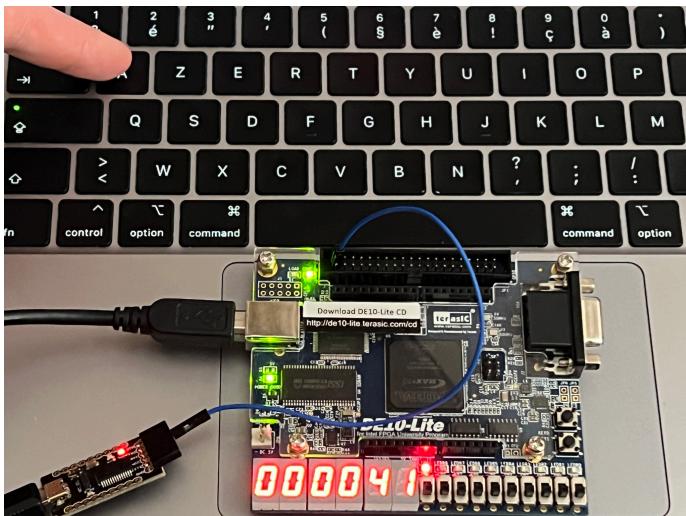
## Schéma:



## Test sur carte

Une fois le projet compilé, on le charge sur la carte. On observe alors que toutes les fonctions précédemment implémentées sont toujours fonctionnelles. De plus, lorsque l'on envoie un caractère depuis PUTTY, son code ASCII hexadécimal s'affiche sur l'afficheur.

## **Validation par le professeur en classe**



## Test de l'envoi d'un 'A' (code ASCII : 0x41) depuis PUTTY

# Conclusion

Le projet de développement d'un processeur mono-cycle en VHDL a été une expérience longue mais très enrichissante. Au cours des mois de mai et juin 2024, j'ai travaillé sur ce projet dans le cadre de ma formation d'ingénieur en électronique et informatique à Polytech Sorbonne, sous la supervision de Yann Douze et Mouad Abrini. Ce projet m'a permis d'apprendre beaucoup de choses, autant sur le plan technique que pratique.

## Apprentissage Technique

Tout d'abord, la conception et la réalisation du processeur mono-cycle m'ont permis de mieux comprendre les architectures informatiques. En décomposant le projet en petits composants, j'ai pu apprendre comment chaque partie interagit avec les autres. Chaque composant a été vérifié et validé à l'aide de bancs de test auto-testants, ce qui a été crucial pour s'assurer de leur bon fonctionnement. Cette méthodologie m'a aidé à structurer mon travail de manière claire et efficace.

## Défis et Solutions

L'un des principaux défis a été de garantir que chaque composant fonctionne correctement et s'intègre bien avec les autres. La mise en œuvre de l'unité arithmétique et logique (UAL), du banc de registres et de l'unité de contrôle a demandé beaucoup de temps et d'efforts. Les tests rigoureux et le débogage ont été essentiels pour résoudre les problèmes rencontrés.

L'intégration de tous les composants pour former un système fonctionnel a également été complexe. Tester le processeur sur une carte FPGA a révélé des problèmes d'intégration qui ont nécessité des ajustements minutieux. Cela m'a appris l'importance de la patience et de la persévérance dans le développement de projets techniques.

## Compétences Pratiques

Ce projet m'a aussi permis de me familiariser avec des outils professionnels tels que ModelSim et Quartus. J'ai appris à rédiger des scripts de simulation et à interpréter des chronogrammes, des compétences qui seront utiles dans ma future carrière. En intégrant des fonctionnalités avancées comme la gestion des interruptions et la communication UART, j'ai pu enrichir le processeur de base et le rendre plus versatile.

L'ajout de la gestion des interruptions et de la communication UART a nécessité des modifications et des extensions du processeur, ce qui m'a permis de comprendre comment ces fonctionnalités peuvent être mises en œuvre dans un système existant.

## Réflexions Finales

En conclusion, ce projet a été une expérience précieuse qui m'a permis de développer des compétences techniques avancées et de mieux comprendre les systèmes numériques. Bien que le projet ait été long et parfois difficile, il a été très intéressant et instructif.

Je remercie mes encadrants, Yann Douze et Mouad Abrini, pour leur soutien et leurs conseils tout au long de ce projet. Leur expertise a été essentielle pour surmonter les défis rencontrés. Je suis satisfait du travail accompli et des compétences que j'ai acquises, qui me seront utiles tout au long de ma carrière professionnelle.

# Annexe : Table des fichiers

Répertoire parent	Nom du fichier	Description	Page du rapport
./src/UART	fdiv.vhd	Diviseur de fréquence	35
./src/UART	uart_rx.vhd	Récepteur UART	35
./src/UART	uart_tx.vhd	Émetteur UART	47
./src/Quartus	SEVEN_SEG.vhd	Décodeur 7 segments	26
./src/Quartus	TopLevel_UART1.vhd	Niveau supérieur avec UART1	38
./src/Quartus	TopLevel_UART2.vhd	Niveau supérieur avec UART2	41
./src/Quartus	TopLevel_UART3.vhd	Niveau supérieur avec UART3	47
./src/Quartus	TopLevel_VIC.vhd	Niveau supérieur avec VIC	33
./src/Quartus	TopLevel.vhd	Niveau supérieur	26
./src	data_mem.vhd	Mémoire de données	10
./src	instruction_memory_IRQ.vhd	Mémoire d'instructions avec IRQ	30
./src	instruction_memory_UART1.vhd	Mémoire d'instructions avec UART1	37
./src	instruction_memory_UART2.vhd	Mémoire d'instructions avec UART2	40
./src	instruction_memory_UART3.vhd	Mémoire d'instructions avec UART3	46
./src	instruction_memory.vhd	Mémoire d'instructions	13
./src	instructions_decoder_VIC.vhd	Décodeur d'instructions avec VIC	29
./src	instructions_decoder.vhd	Décodeur d'instructions	19
./src	Instructions_Unit_UART.vhd	Unité d'instructions avec UART	37
./src	Instructions_Unit_UART2.vhd	Unité d'instructions avec UART2	40
./src	Instructions_Unit_UART3.vhd	Unité d'instructions avec UART3	46
./src	Instructions_Unit_VIC.vhd	Unité d'instructions avec VIC	30
./src	Instructions_Unit.vhd	Unité d'instructions	16
./src	MUX.vhd	Multiplexeur	8
./src	PC_reg.vhd	Registre de compteur de programme	14
./src	PC_update_unit_VIC.vhd	Unité de mise à jour du PC avec VIC	30
./src	PC_update_unit.vhd	Unité de mise à jour du PC	14
./src	Processor_UART1.vhd	Processeur avec UART1	38
./src	Processor_UART2.vhd	Processeur avec UART2	41
./src	Processor_UART3.vhd	Processeur avec UART3	47
./src	Processor_VIC.vhd	Processeur avec VIC	32
./src	Processor.vhd	Processeur seul	24
./src	reg32.vhd	Registre 32 bits	18
./src	register_bench.vhd	Banc de registres	5
./src	sign_ext.vhd	Extension de signe	9
./src	UAL.vhd	Unité Arithmétique et Logique	4
./src	UART_reg.vhd	Registre UART	36
./src	UC_VIC.vhd	Unité de contrôle avec VIC	29
./src	UC.vhd	Unité de contrôle	22
./src	UT_UART.vhd	Unité de traitement avec UART	36
./src	UT_UART2.vhd	Unité de traitement avec UART2	44
./src	UT.vhd	Unité de traitement	11
./src	VIC_UART.vhd	VIC avec UART	39
./src	VIC_UART2.vhd	VIC avec UART2	45
./src	VIC.vhd	Contrôleur d'interruptions vectorisé	28
./simu	data_mem_TB.vhd	Banc de test pour la mémoire de données	10
./simu	instruction_memory_TB.vhd	Banc de test pour la mémoire d'instructions	13
./simu	instructions_decoder_TB.vhd	Banc de test pour le décodeur d'instructions	20
./simu	instructions_decoder_VIC_TB.vhd	Banc de test pour le décodeur d'instructions avec VIC	29
./simu	Instructions_Unit_TB.vhd	Banc de test pour l'unité d'instructions	16
./simu	MUX_TB.vhd	Banc de test pour le multiplexeur	8
./simu	PC_reg_TB.vhd	Banc de test pour le registre de compteur de programme	14
./simu	PC_update_unit_TB.vhd	Banc de test pour l'unité de mise à jour du PC	15

<b>./simu</b>	PC_update_unit_VIC_TB.vhd	Banc de test pour l'unité de mise à jour du PC avec VIC	31
<b>./simu</b>	Processor_TB.vhd	Banc de test pour le processeur	24
<b>./simu</b>	Processor_UART1_TB.vhd	Banc de test pour le processeur avec UART1	38
<b>./simu</b>	Processor_UART2_TB.vhd	Banc de test pour le processeur avec UART2	42
<b>./simu</b>	Processor_UART3_TB.vhd	Banc de test pour le processeur avec UART3	47
<b>./simu</b>	Processor_VIC_TB.vhd	Banc de test pour le processeur avec VIC	33
<b>./simu</b>	Reg_Bench_TB.vhd	Banc de test pour le banc de registres	6
<b>./simu</b>	REG32_TB.vhd	Banc de test pour le registre 32 bits (PSR et autres)	18
<b>./simu</b>	sign_ext_TB.vhd	Banc de test pour l'extension de signe	9
<b>./simu</b>	UAL_tb.vhd	Banc de test pour UAL	4
<b>./simu</b>	UART_Reg_TB.vhd	Banc de test pour le registre UART	36
<b>./simu</b>	UC_TB.vhd	Banc de test pour l'unité de contrôle	22
<b>./simu</b>	UT_TB.vhd	Banc de test pour l'unité de traitement	12
<b>./simu</b>	UT_UAL_RegBen_TB.vhd	Banc de test pour UT, UAL, RegBench	7
<b>./simu</b>	UT_UART_TB.vhd	Banc de test pour l'unité de traitement avec UART	37
<b>./simu</b>	UT_UART2_TB.vhd	Banc de test pour l'unité de traitement avec UART2	45
<b>./simu</b>	VIC_TB.vhd	Banc de test pour le VIC	28
<b>./simu</b>	VIC_UART_TB.vhd	Banc de test pour le VIC avec UART	39
<b>./simu</b>	VIC_UART2_TB.vhd	Banc de test pour le VIC avec UART2	45
<b>./simu</b>	simu_data_mem.do	Script de simulation pour la mémoire de données	10
<b>./simu</b>	simu_instruction_memory.do	Script de simulation pour la mémoire d'instructions	13
<b>./simu</b>	simu_Instructions_Decoder_VIC.do	Script de simulation pour le décodeur avec VIC	29
<b>./simu</b>	simu_Instructions_Decoder.do	Script de simulation pour le décodeur d'instructions	20
<b>./simu</b>	simu_IU.do	Script de simulation pour l'unité d'instructions	16
<b>./simu</b>	simu_MUX.do	Script de simulation pour le multiplexeur	8
<b>./simu</b>	simu_PC_REG.do	Script de simulation pour le registre PC	14
<b>./simu</b>	simu_PC_update_unit_VIC.do	Script de simulation pour l'unité de mise à jour avec VIC	31
<b>./simu</b>	simu_PC_update_unit.do	Script de simulation pour l'unité de mise à jour du PC	15
<b>./simu</b>	simu_Processor_UART1.do	Script de simulation pour le processeur avec UART1	38
<b>./simu</b>	simu_Processor_UART2.do	Script de simulation pour le processeur avec UART2	42
<b>./simu</b>	simu_Processor_UART3.do	Script de simulation pour le processeur avec UART3	47
<b>./simu</b>	simu_Processor_VIC.do	Script de simulation pour le processeur avec VIC	33
<b>./simu</b>	simu_Processor.do	Script de simulation pour le processeur seul	24
<b>./simu</b>	simu_REG32.do	Script de simulation pour le registre 32 bits	18
<b>./simu</b>	simu_RegBen.do	Script de simulation pour le banc de registres	6
<b>./simu</b>	simu_sign_ext.do	Script de simulation pour l'extension de signe	9
<b>./simu</b>	simu_UAL_RegBen.do	Script de simulation pour UAL et banc de registres	7
<b>./simu</b>	simu_UAL.do	Script de simulation pour UAL	4
<b>./simu</b>	simu_UART_REG.do	Script de simulation pour le registre UART	36
<b>./simu</b>	simu_UC.do	Script de simulation pour l'unité de contrôle	22
<b>./simu</b>	simu_UT_UART.do	Script de simulation pour l'unité de traitement avec UART	37
<b>./simu</b>	simu_UT_UART2.do	Script de simulation pour l'unité de traitement avec UART2	45
<b>./simu</b>	simu_UT.do	Script de simulation pour l'unité de traitement	12
<b>./simu</b>	simu_VIC_UART.do	Script de simulation pour le VIC avec UART	39
<b>./simu</b>	simu_VIC_UART2.do	Script de simulation pour le VIC avec UART2	45
<b>./simu</b>	simu_VIC.do	Script de simulation pour le VIC	28