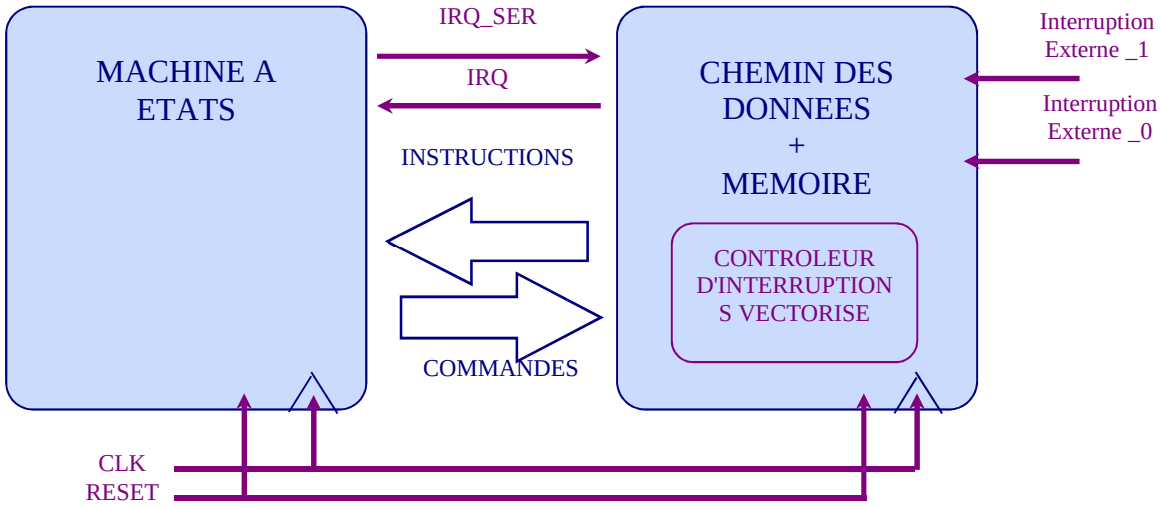
PROCESSEUR MULTI-CYCLES: IMPLEMENTATION FPGA

Nous allons à présent faire évoluer l'architecture du processeur mono-cycle modélisé précédemment afin que les instructions soient exécutées sur plusieurs cycles. Le synopsis de son architecture est donné ci-dessous :



- Le bloc « Chemin de Données + Mémoire » est inspiré de la version mono-cycle du processeur avec quelques ajustements. Il inclut un contrôleur d'interruptions qui permet de gérer le traitement de deux interruptions externes (qui seront connectées à deux boutons poussoirs de la carte FPGA).
- Le bloc « Machine à Etats » réalise le décodage des instructions et permet leur exécution en configurant les différents registres et opérateurs du chemin de données.

Le but de ce TP sera:

1) ***De reprendre le chemin de données du processeur monocycle pour l'adapter au multi-cycles.***
2) ***De décrire en VHDL la machine à états du processeur et de l'interfacer au chemin de données dont le code vous sera donné.***
3) ***D'implémenter le processeur complet sur la carte ALTERA DE0.***

PARTIE 1 – PRISE EN MAIN DU PROJET SUR QUARTUS

A FAIRE

- Copier sur votre compte le projet Quartus du processeur multi-cycles qui se trouve sur le réseau.
- Ouvrir le projet. Il contient pour le moment 3 fichiers source VHDL :
    - **DE0_TOP** : le fichier principal qui instancie le processeur et le connecte aux entrées/sorties de la carte DE0
    - **ARM** : le modèle du processeur. On y instancie le chemin de données et la MAE.
    - **DATAPATH** : le chemin de données du processeur multi-cycles.
- Ajouter aux sources du projet (Menu Project → Add/Remove Files in Project) les modules de base développés pour le processeur mono-cycle, à savoir :
    - *UAL (ou ALU)*
    - *Banc de registres*
    - *Mux 2→1*
    - *Extension de signe*
    - *Registre 32 bits avec commande de chargement*

Modules de base additionnels pour le chemin de données:

Multiplexeur 4 → 1

Entrées/Sorties :

- **A, B, C, D**: Entrées sur 32 bits
- **COM**: Commande sur 2 bits.
- **S**: Sortie sur 32 bits.

Fonctionnement:

Ce multiplexeur 4→1 laisse passer en sortie l'entrée selon la valeur de la commande **COM**

COM	Sortie du Multiplexeur
00	Entrée A
01	Entrée B
10	Entrée C
11	Entrée D

Registre 32 bits

Entrées/Sorties :

- **DATAIN**: Entrée sur 32 bits
- **CLK**: Horloge
- **RST**: Reset asynchrone, actif à l'état haut
- **DATAOUT**: Sortie sur 32 bits.

Memoire 64 x 32 bits

La mémoire du processeur sera réalisée grâce à l'outil **Mega Wizard Plug-In Manager** (accessible depuis le menu **Tools** de Quartus.

- Sélectionner **Create a New Custom Megafunction**.
- Choisir **Memory Compiler** puis **RAM 2-PORT** avec une sortie de type VHDL. Donner un nom à ce fichier.
- Dans les onglets successifs:
  - Cocher 1 port de lecture et 1 port d'écriture.
  - Configurer la taille de la mémoire pour disposer de 64 mots de 32 bits.
  - Cocher les options **Single Clock** et **Read Enable**. Ne pas cocher les autres.
  - Dans les **Registered Ports**, cocher **Write Input Ports** et **Read Input Ports**. Ne pas cocher le reste.
  - Sélectionner **I Don't Care**
  - Donner un fichier d'initialisation du tableau. Ce fichier **mem_init.mif** est disponible sur le disque Commun des machines (à recopier sur votre compte)

A FAIRE

1) ***Créer de nouvelles sources VHDL pour décrire les modules additionnels (Mux 4→1, Registre 32 bits).***

2) ***Créer la Megafunction de la mémoire 64x32 bits***

3) ***Ajouter ces sources au projet.***

FONCTIONNEMENT DU CHEMIN DE DONNES + MEMOIRE

L'architecture du chemin de données est donnée en Annexe 1. On y retrouve les principaux modules vus dans le processeur monocycle (**UAL**, banc de registres, extenseurs de données, registre **PC**…)

Les principales différences et ajouts sont :

- Les mémoires données et instructions du processeur monocycle sont rassemblées en une seule mémoire.

- On trouve des registres (**IR, DR, A, B, ALUOUT**) entre les principaux modules du chemin de données (Mémoire, banc de registres, **ALU**…). Cela permet de gérer l'exécution d'une instruction sur plusieurs cycles.

- Un contrôleur d'interruptions vectorisé (**VIC**) permet d'interrompre le programme principal si l'une des deux requêtes (**IRQ0** et **IRQ1**) est activée.

- Deux registres (**LR, SPSR**) permettent en cas d'interruption de faire une sauvegarde de contexte (du registre **PC** et du registre de statut **CPSR**).

Voici à présent le détail de fonctionnement des modules du chemin de données. Les blocs sont décrits selon le schéma de l'Annexe 1, en partant de la gauche vers la droite. On prêtera notamment attention aux noms et aux valeurs des commandes qui seront à traiter par la **MAE**.

VIC : Controleur d'interruption vectorisé

Entrées :

- **CLK**: Horloge
- **RESET** : Reset asynchrone, actif à l'état haut
- **IRQ_SERV** : Acquittement de l'interruption venant de la MAE
- **IRQ0, IRQ1** : Requêtes d'interruption

Sorties :

- **IRQ**: Requête d'interruption envoyée à la MAE
- **VICPC** : Adresse de début du sous-programme d'interruption (sur 32 bits)

Fonctionnement :

- Ce module échantillonne les valeurs de **IRQ0** et **IRQ1** afin de connaître la dernière (**IRQi(n)**) et l'avant dernière (**IRQi(n-1)**) valeur de ces deux signaux.
- Si on détecte une transition montante (**IRQi(n)**=1 ET **IRQi(n-1)**=0) sur l'un de ces signaux, on force à l'état haut un signal **IRQ0_memo** ou **IRQ1_memo,** signalant une requête d'interruption.
- **IRQ1_memo** et **IRQ0_memo** restent à l'état haut tant que l'on n'a pas reçu d'acquittement sur l'entrée **IRQ_SERV**.
- S'il n'y a aucune requête d'interruption, **VICPC** est forcé à 0.
  - Si **IRQ0_memo** = 1, on force la sortie **VICPC** à la valeur 0x9 (adresse de début de ce sous-programme d'interruption)
  - Si **IRQ1_memo** = 1, on force la sortie **VICPC** à la valeur 0x15 (adresse de début de ce sous-programme d'interruption)
  - L'**IRQ0** est prioritaire sur l'**IRQ1**.
  - La sortie **IRQ** envoyée à la **MAE** est un OU logique entre **IRQ1_memo** et **IRQ0_memo**.

Signal envoyé à la MAE : ***IRQ***
Commande à générer par la MAE: ***IRQ_SERV.***

A FAIRE

1) ***Créer une nouvelle source VHDL pour décrire le VIC. Ajouter cette source au projet.***

MUX_PC: Multiplexeur d'entrée du registre PC

Ce bloc est un multiplexeur 4→1. Les entrées et la sortie sont sur 32 bits.

Entrées :

- **A**: Sortie de l'**ALU**
- **B**: Sortie du registre **ALUOUT**
- **C**: Sortie du registre **LR**
- **D**: Sortie du **VIC** (Adresse du sous-programme d'interruption)
- **PCSel**: Commande sur 2 bits.

Sorties :

- **S**: Connectée à l'entrée du registre **PC**

Fonctionnement:

Ce multiplexeur 4→1 permet de sélectionner la donnée à charger dans le registre **PC**. Il est commandé par un signal **PCSel** sur 2 bits

PCSel	Sortie du Multiplexeur
00	Sortie de l'**ALU**
01	Sortie du registre **ALUOUT**
10	Sortie du registre **LR**
11	Sortie du **VIC**

Commande à générer par la MAE: ***PCSel (2 bits)***

PC: Registre PC

Fonctionnement

Ce registre 32 bits avec commande de chargement contient l'adresse de la prochaine instruction. Il charge une donnée en entrée si la commande **PCWrEn** est mise à 1, et conserve sa valeur précédente si la commande est à 0.

Commande à générer par la MAE: ***PCWrEn***

LR: Link Register

Fonctionnement :
Ce registre 32 bits avec commande de chargement sauvegarde la valeur du registre PC lors du traitement d'une interruption. Le chargement de l'adresse de retour au programme principal est commandé par le signal **LRWrEn**, actif à 1. Le registre conserve sa valeur sinon.

Commande à générer par la MAE: ***LRWrEn***

Mux_MEM : Multipkexeur du Bus d'adresses de la mémoire

Fonctionnement :
Ce multiplexeur 2→1 sur 6 bits permet de sélectionner la valeur à positionner sur le bus d'adresses de la mémoire. Il est commandé par le signal **AdrSel.**

AdrSel	Sortie du Multiplexeur
0	Sortie du registre **PC** (5 downto 0)
1	Sortie du registre **ALUOUT** (5 downto 0)

Commande à générer par la MAE: ***AdrSel***

MEMORY: Memoire interne du processeur

Fonctionnement :
La mémoire contient le programme à exécuter ainsi que toutes les données nécessaires. D'une capacité de 64x32 bits, elle possède un port en lecture et un port en écriture. Le bus d'adresses est commun à ces deux ports.
La lecture est commandée par le signal **MemRdEn** actif à l'état haut.
L'écriture est commandée par le signal **MemWrEn**, actif à l'état haut

Commande à générer par la MAE: ***MemRdEn, MemWrEn***

IR : Registre Instruction

Fonctionnement :
Ce registre 32 bits avec commande de chargement contient l'instruction courante du programme. Il charge une donnée provenant de la mémoire si la commande **IRWrEn** est à l'état haut, et conserve sa valeur sinon.

Commande à générer par la MAE: ***IRWrEn***

DR : Registre de Données Mémoire

Fonctionnement :
Ce registre 32 bits stocke une donnée issue de la mémoire. Le chargement se fait à chaque cycle d'horloge.

Commande à générer par la MAE: ***Aucune***

MUX_REG_RB : Multiplexeur du Bus d'adresses RB du banc de registres

Fonctionnement :
Ce multiplexeur 2→1 sur 4 bits permet de sélectionner la valeur à positionner sur le bus d'adresses **RB** du banc de registres. Sa commande est générée par l'instruction stockée dans le registre **IR** selon l'équation suivante :

COM = NOT [IR(27) OR IR(20)] AND IR(26).

COM	Sortie du Multiplexeur
0	Sortie du registre **IR** (3 downto 0)
1	Sortie du registre **IR** (15 downto 12)

Commande à générer par la MAE: ***Aucune***

MUX_REG_BUSW : Multiplexeur du Bus des données W du banc des registres

Fonctionnement :
Ce multiplexeur 2→1 sur 32 bits permet de sélectionner la valeur à positionner sur le bus de données **W** du port d'écriture du banc de registres. Il est commandé par le signal **WSel**.

WSel	Sortie du Multiplexeur
0	Sortie du registre **DR**
1	Sortie du registre **ALUOUT**

Commande à générer par la MAE: ***WSel***

Register File : Banc de registres

Fonctionnement :
Le banc de registres contient 16 registres de 32 bits, accessibles via deux ports de lecture et un port d'écriture. L'écriture est commandée par le signal **RegWrEn**.

Commande à générer par la MAE : ***RegWrEn***

Ext_8 : Extension de données 8 → 32

Fonctionnement :
Extension signée des 8 bits de poids faible du registre **IR** sur 32 bits

Commande à générer par la MAE : ***Aucune***

Ext_24 : Extension de données 24 → 32

Fonctionnement :
Extension signée des 24 bits de poids faible du registre **IR** sur 32 bits

Commande à générer par la MAE : ***Aucune***

A : Registre de sortie du port A du banc de registres

Fonctionnement :
Ce registre 32 bits stocke à chaque cycle la valeur du bus de données du port de lecture A du banc de registres.

Commande à générer par la MAE : ***Aucune***

B : Registre de sortie du port B du banc de registres

Fonctionnement :
Ce registre 32 bits stocke à chaque cycle la valeur du bus de données du port de lecture B du banc de registres.

Commande à générer par la MAE : ***Aucune***

Mux_ALU_A : Multiplexeur entrée A de l'ALU

Fonctionnement :

Ce multiplexeur 2→1 sur 32 bits permet de sélectionner la valeur à positionner sur l'entrée A de **l'ALU**. Il est commandé par le signal **ALUSelA**.

ALUSelA	Sortie du Multiplexeur
0	Sortie du registre **PC**
1	Sortie du registre **A**

Commande à générer par la MAE: ***ALUSelA***

Mux_ALU_B : Multiplexeur entrée B de l'ALU

Fonctionnement :

Ce multiplexeur 4→1 sur 32 bits permet de sélectionner la valeur à positionner sur l'entrée B de l'**ALU**. Il est commandé par le signal **ALUSelB**.

ALUSelB	Sortie du Multiplexeur
00	Sortie du registre **B**
01	Sortie du module **EXT8**
10	Sortie du module **EXT24**
11	1 (valeur constante)

Commande à générer par la MAE: ***ALUSelB (2 bits)***

ALU : Unité Arithmetique et Logique

Fontionnement :
L'**ALU** permet d'effectuer une opération, en fonction de la commande **ALUOP** sur 2 bits. En plus du résultat, il fournit des drapeaux indicateurs sur la nature du résultat (en particulier le flag **N**).

ALUOP	Opération
00	A + B
01	B
10	A – B
11	A

Commande à générer par la MAE: ***ALUOP (2 bits)***

ALU_Out : Registre de sortie de l'ALU

Fonctionnement :
Ce registre 32 bits stocke à chaque cycle d'horloge la sortie de l'**ALU**.

Commande à générer par la MAE: ***Aucune***

Mux_CPSR : Multiplexeur d'entrée du Registre CPSR

Fonctionnement :
Ce multiplexeur 2→1 sur 32 bits permet de sélectionner la valeur à positionner en entrée du registre **CPSR**. Il est commandé par le signal **CPSRSel**.

CPSRSel	Sortie du Multiplexeur
0	32 bits comprenant le drapeau **N** de l'**ALU** (MSB) et les bits 30 à 0 du registre **CPSR** (LSB)
1	Sortie du registre **SPSR**

Commande à générer par la MAE: ***CPSRSel***

CPSR : Current Processor Status Register

Fonctionnement :
Ce registre 32 bits avec commande de chargement stocke l'état courant du processeur. Le chargement est commandé par le signal **CPSRWrEn**, actif à l'état haut.

Commande à générer par la MAE: ***CPSRWrEn***

SPSR : Saved Processor Status Register

Fonctionnement :
Ce registre 32 bits à commande de chargement sauvegarde l'état du processeur en cas de traitement d'une interruption. Le chargement est commandé par le signal **SPSRWrEn**, actif à l'état haut.

Commande à générer par la MAE: ***SPSRWrEn***

RESULTAT: Registre Résultat

Fonctionnement :
Ce registre 32 bits à commande de chargement a pour but de sauvegarder le résultat du calcul effectué par le programme principal une fois que celui-ci est terminé. Le chargement est validé par la commande **ResWrEn** active à l'état haut.

La sortie du registre est connectée aux afficheurs 7 segments de la carte DE2-70 pour visualisation.

Commande à générer par la MAE: ***ResWrEn***

A FAIRE

1) ***Compléter le fichier VHDL DATAPATH pour y instancier tous les blocs de base du chemin de données, conformément au cahier des charges ci-dessus.***

CODAGE VHDL DE LA MACHINE A ETATS

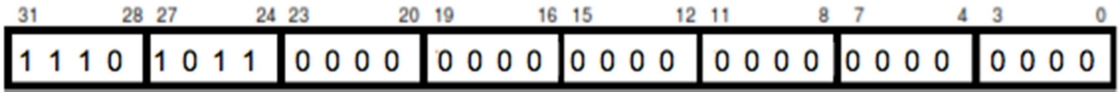
Le rôle de la **MAE** est donc d'effectuer le décodage des instructions envoyées par la mémoire du processeur et d'en mettre en œuvre l'exécution. On trouvera ci-dessous une vue externe de ce composant.

Descriptif des signaux d'E/S :

- ***CLK*** : Horloge
- ***RST*** : Reset Asynchrone
- ***IRQ*** : Requête d'interruption provenant du VIC
- ***IRQ_SERV*** : Drapeau d'interruption
- I***NST_MEM***: Instruction (32 bits) à décoder. Vient de la sortie de la mémoire
- ***INST_REG*** : Instruction (32 bits) à décoder. Vient de la sortie du registre **IR** (*)
- ***N***: Drapeau **N** de l'**ALU** (pour les instructions de branchement conditionnel)
- ***COMMANDES*** : Commandes des registres et des opérateurs du chemin de données (cf. plus haut)

(*) En raison de l'exécution sur plusieurs cycles des instructions, il est nécessaire d'avoir accès à l'instruction depuis ces deux points du chemin de données.

Le sous-ensemble du jeu d'instruction est le même que celui du processeur mono-cycle. On y ajoute simplement l'instruction **BX** qui permet la restauration du contexte à la fin du traitement d'une interruption. Son code binaire est le suivant :



Fonctionnement

Le fonctionnement général de la **MAE** est donné par les graphes d'état des annexes 2 et 3. L'Annexe 2 donne le séquencement sans tenir compte des interruptions. L'Annexe 3 montre comment implémenter la gestion des interruptions en rajoutant trois états et des conditions supplémentaires sur **IRQ** et un signal interne **ISR** dont le comportement est le suivant :

- **ISR** est un signal synchrone sur l'horloge
- **ISR** passe à 1 lorsque la **MAE** provoque une sauvegarde de contexte.
- **ISR** passe à 0 lorsque la **MAE** provoque une restauration de contexte.

Chaque état implique un positionnement bien spécifique des signaux de commande à envoyer au chemin de données.

Programme de Test

Le programme de test est le suivant. Le programme principal fait la somme de 10 cases mémoires qui contiennent des valeurs allant de 1 à 10. Le résultat attendu est donc 55.

Les sous-programmes d'interruption incrémentent la valeur de l'une des cases mémoire, modifiant ainsi le total en sortie.

Programme principal:

```
0x0   _main :    MOV R1,#0x20     ; --R1 <= 0x20
0x1              MOV R2,#0        ; --R2 <= 0
0x2   _loop :    LDR R0,0(R1)     ; --R0 <= MEM[R1]
0x3              ADD R2,R2,R0     ; --R2 <= R2 + R0
0x4              ADD R1,R1,#1     ; --R1 <= R1 + 1
0x5              CMP R1,0x2A      ; --? R1 = 0x2A
0x6              BLT loop         ; --branchement à _loop si R1
inferieur a 0x2A
       _end  :
0x7              STR R2,0(R1)     ; --MEM[R1] <= R2
0x8              BAL main         ; --branchement à _main
```

ISR interruption 0 :

--sauvegarde du contexte - R15 correspond au pointeur de pile

```
0x9              STR R1,0(R15)    ; --MEM[R15] <= R1
0xA              ADD R15,R15,1    ; --R15 <= R15 + 1
0xB              STR R3,0(R15)    ; --MEM[R15] <= R3
```

--traitement

```
0xC              MOV R3,0x20      ; --R3 <= 0x20
0xD              LDR R1,0(R3)     ; --R1 <= MEM[R3]
0xE              ADD R1,R1,1      ; --R1 <= R1 + 1
0xF              STR R1,0(R3)     ; --MEM[R3] <= R1
```

--chargement du context

```
0x10             LDR R3,0(R15)    ; --R3 <= MEM[R15]
0x11             ADD R15,R15,-1   ; --R15 <= R15 - 1
0x12             LDR R1,0(R15)    ; --R1 <= MEM[R15]
0x13             BX               ; -- instruction de fin
d'interruption
```

ISR interruption 1 :

--sauvegarde du contexte - R15 correspond au pointeur de pile

```
0x15             STR R4,0(R15)    ; --MEM[R15] <= R4
0x16             ADD R15,R15,1    ; --R15 <= R15 + 1
0x17             STR R5,0(R15)    ; --MEM[R15] <= R5
```

--traitement

```
0x18             MOV R5,0x20      ; --R5 <= 0x20
0x19             LDR R4,0(R5)     ; --R4 <= MEM[R5]
0x1A             ADD R4,R4,2      ; --R4 <= R1 + 2
0x1B             STR R4,0(R5)     ; --MEM[R5] <= R4
```

--chargement du contexte

```
0x1C             LDR R5,0(R15)    ; --R5 <= MEM[R15]
0x1D             ADD R15,R15,-1   ; --R15 <= R15 - 1
0x1E             LDR R4,0(R15)    ; --R4 <= MEM[R15]
0x1F             BX               ; -- instruction de fin
d'interruption
```

Travail à Effectuer

1) ***Décrire dans une nouvelle source VHDL la MAE du processeur, conformément aux graphes d'état des annexes 2 et 3. Ajouter cette source au projet. Instancier la MAE dans le fichier ARM.***

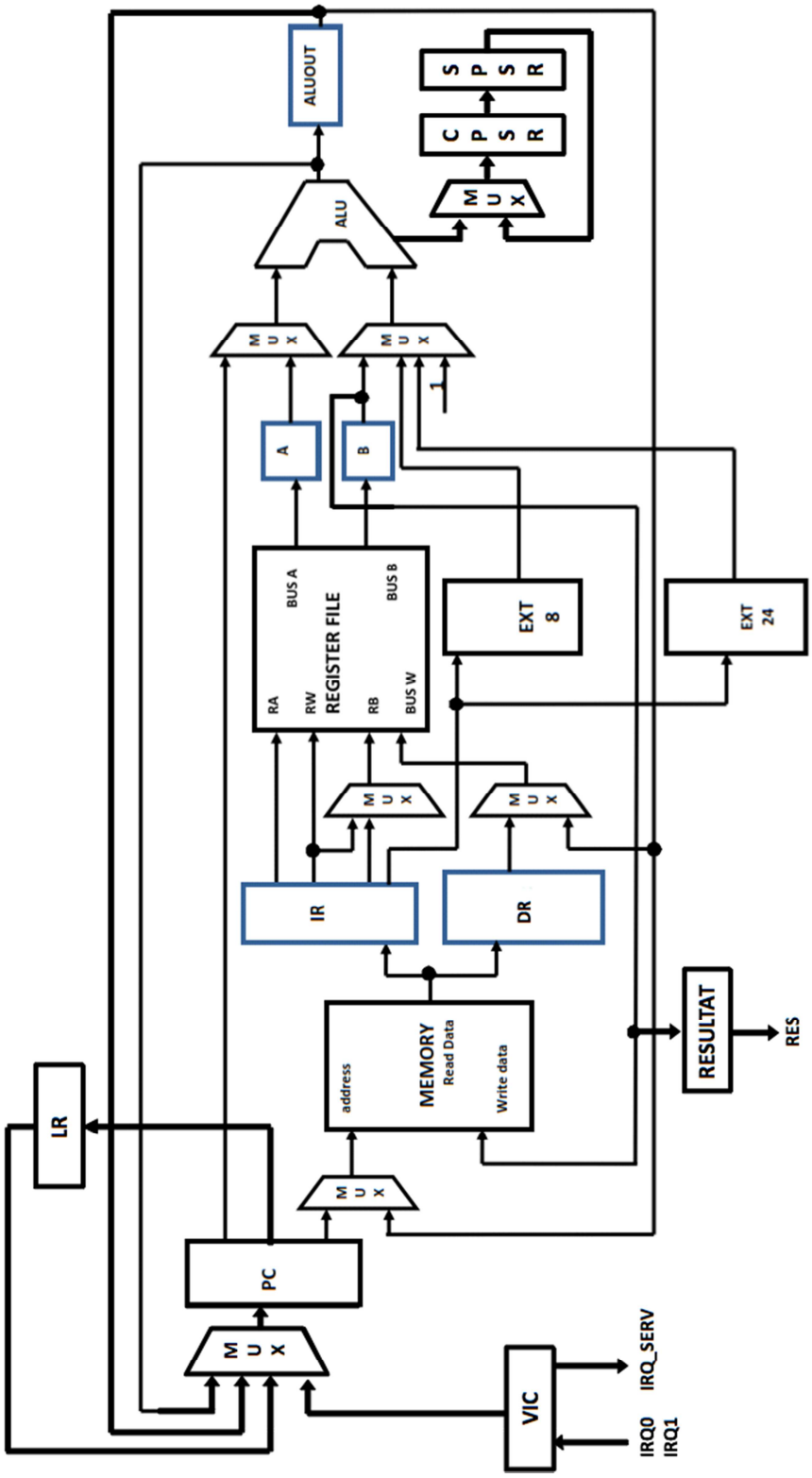
2) ***Vérifier avec l'Assignment Editor (menu Assignments) que les entrées/sorties du projet sont bien mappées sur la carte. Au besoin, importer le fichier qsf de la DE0 (on le trouvera sur le disque commun).***

3) ***Compiler le projet (Menu Processing → Start Compiler). Noter sur le rapport de synthèse le taux d'occupation des ressources du FPGA.***

4) ***A l'aide du testbench TEST_ARM.VHD, simuler le module ARM avec Modelsim Altera. Vérifier qu'entre 6,0 et 6,1 µs, la sortie résultat passe à la valeur 55 (en décimal).***

5) ***Etudier le code du fichier DE0_TOP pour comprendre comment piloter le processeur à l'aide des boutons, switches, LEDs et afficheurs de la carte. Implémenter le processeur sur la carte DE0 et vérifier le bon fonctionnement du processeur.***

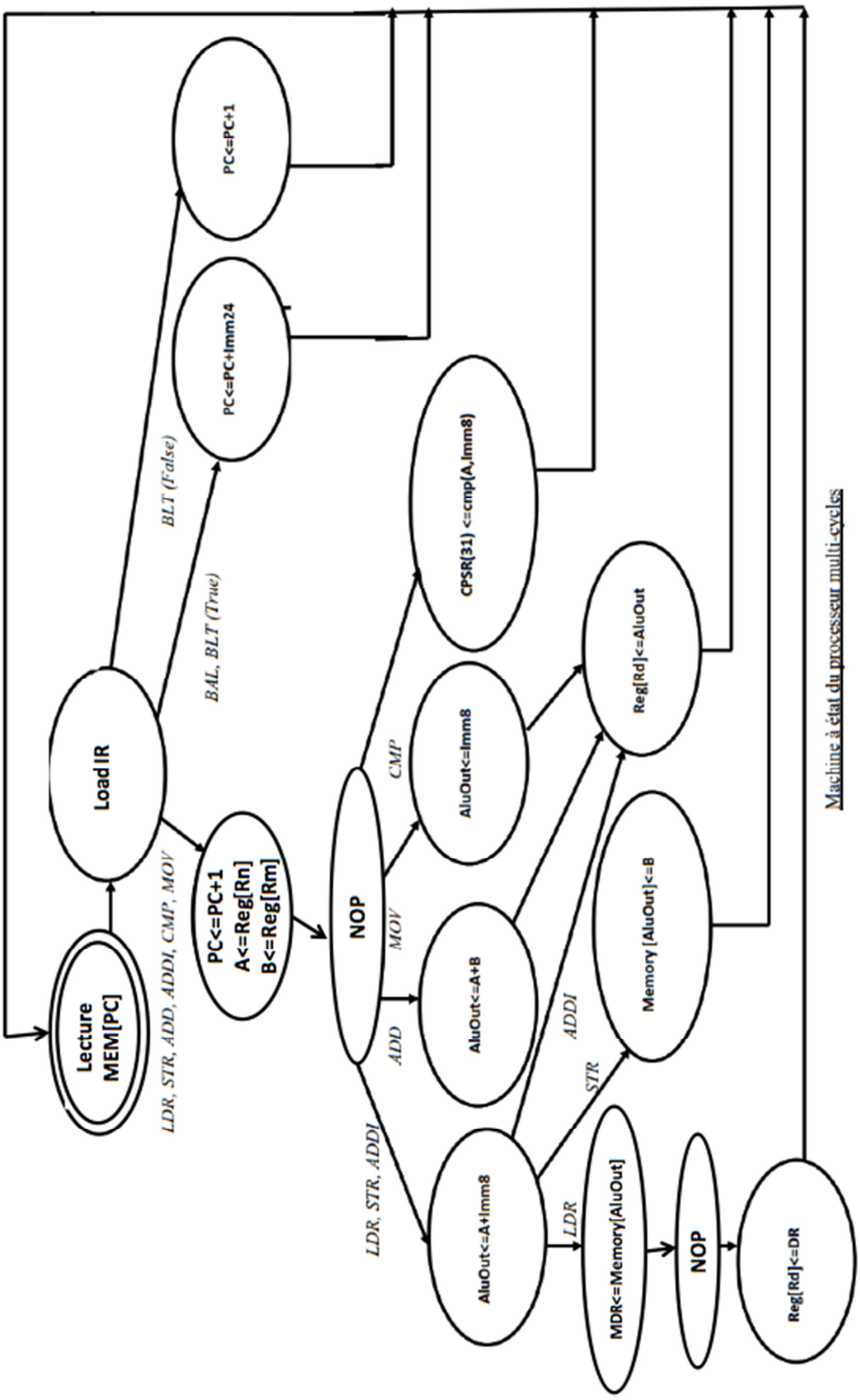
ANNEXE 1 – CHEMIN DE DONNEES et TABLEAU DES COMMANDES



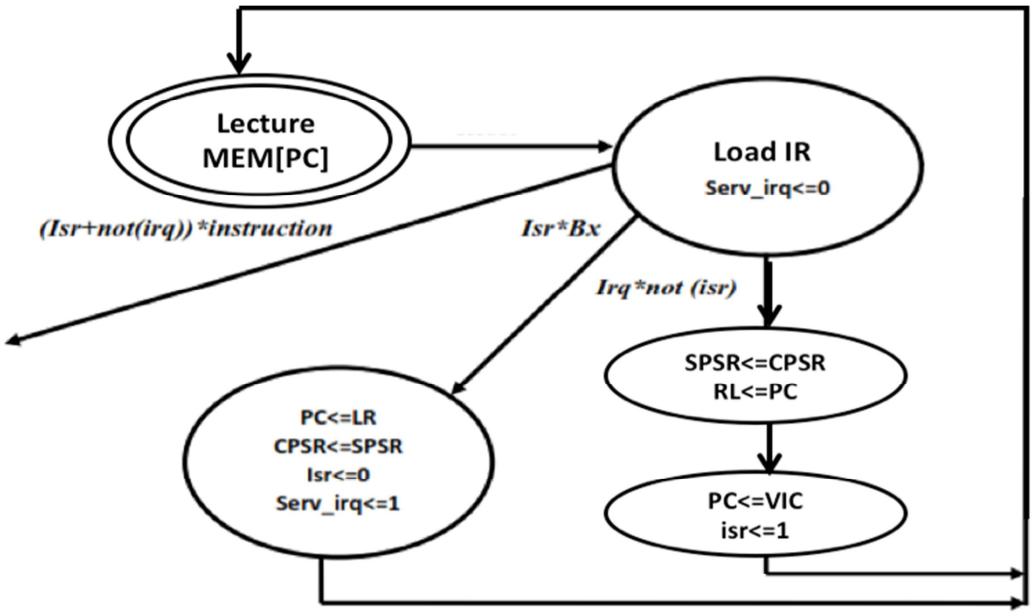
ETAT MAE	IRQ Serv	PC Sel	PC Wr En	LR Wr En	Adr Sel	Mem RdEn	Mem WrEn	IR Wr En	W Sel	Reg Wr En	ALU SelA	ALU SelB	ALU OP	CPSR Sel	CPSR WrEn	SPSR WrEn	Res WrEn

ETAT MAE	IRQ Serv	PC Sel	PC Wr En	LR Wr En	Adr Sel	Mem RdEn	Mem WrEn	IR Wr En	W Sel	Reg Wr En	ALU SelA	ALU SelB	ALU OP	CPSR Sel	CPSR WrEn	SPSR WrEn	Res WrEn

ANNEXE 2 – GRAPHE D'ETATS MAE (sans les interruptions)



ANNEXE 3 – AJOUT DES INTERRUPTIONS



États ajoutés pour l'interfaçage