

Le système MUO

et ses outils

Jean-Pierre Miceli
Décembre 2005
Version 2.1



Mise à jour de ce manuel

Je remercie tous les utilisateurs de ce manuel de m'indiquer les erreurs qu'il comporte, de même, que les informations manquantes ou incomplètes. Cela permettra une mise à jour régulière de ce manuel.

version 1: Le système à Processeur MU0 au labo et ses outils
Bernard Perrin, mars 2003

Contact

Auteur: Jean-Pierre Miceli
e-mail : reds@heig-vd.ch
Tél: +41 (0)24 557 63 02

Adresse: Institut REDS, Reconfigurable & Embedded Digital Systems
HEIG-VD (Haute Ecole d'Ingénierie et Gestion du Canton de Vaud)
Route de Cheseaux 1
CH-1400 Yverdon-les-Bains
Tél : ++41 (0)24 55 76 330 (central)
Fax : ++41 (0)24 55 76 404
E-mail : reds@heig-vd.ch
Internet : <http://reds.heig-vd.ch/>

Autres personnes à contacter en cas d'absence:

M. Boada Serge	e-mail: Serge.Boada@heig-vd.ch	Tél. direct +41 (0)24 55 76 269
M. Messerli Etienne	e-mail: Etienne.Messerli@heig-vd.ch	Tél. direct +41 (0)24 55 76 302
M. Boutillier Guillaume	e-mail: guillaume.boutillier@heig-vd.ch	Tél. direct +41 (0)24 55 76 273
M. Primault Gilles	e-mail: gilles.primault@heig-vd.ch	Tél. direct +41 (0)24 55 76 259

Table des matières

Chapitre 1 Le système MU0	1
1-1. Spécification du système à processeur MU0	2
1-1.1. Spécification du bus système	2
1-1.2. Plan d'adressage du système à processeur MU0	2
<i>Description du fonctionnement des ROR, ROL, LSR et LSL</i>	3
1-2. Les objectifs didactiques	4
Chapitre 2 Le processeur MU0	5
2-1. Modèle du programmeur du MU0	6
2-2. Structure du système à processeur MU0	7
2-2.1. Schéma bloc général	7
2-2.2. Etude du niveau RTL (Register Transfer Level)	8
2-2.3. L'unité de commande (Control Logic)	9
2-2.4. L'initialisation	10
2-2.5. L'ALU	10
2-3. Informations supplémentaires	13
Chapitre 3 Réalisation du système	15
3-1. La bibliothèque MU0	16
3-1.1. Le MU0_top	16
3-1.2. Le MU0 Std	17
<i>Version synthétisable</i>	17
<i>Gestion du timing interne et externe</i>	18
<i>Version de spécification</i>	20
3-2. La bibliothèque BlocIO	21
3-3. La bibliothèque SysMU0	21
3-4. La bibliothèque FPGA_CPU et la bibliothèque FPGA_IO	22

Chapitre 4 Les outils	23
4-1. Liste des instructions disponibles	23
4-2. L'assembleur AsmMu0	24
4-2.1. Syntaxes acceptées par AsmMu0	25
<i>La mise en page</i>	25
<i>Les étiquettes (labels)</i>	25
<i>Les commentaires</i>	25
<i>Les équivalences</i>	25
<i>La définition de constantes stockés en mémoire ROM</i>	26
<i>Les constantes référant une étiquette</i>	26
<i>Les nombres</i>	26
<i>Encodage lors d'un LDA #</i>	27
4-2.2. Formats de sortie	27
4-2.3. Invocation de l'assembleur	28
<i>Invocation par commande DOS</i>	28
<i>Invocation depuis MU0Sim</i>	29
4-2.4. Numéro de version	29
4-3. Le Simulateur SimMu0	29
4-3.1. Modes de fonctionnement de SimMU0	31
4-3.2. Fonctionnalités de base	31
4-3.3. Valeurs affichées	32
4-3.4. Formats d'entrée/sortie du simulateur	32
4-3.5. Ajout d'instruction	33
4-3.6. Mode Target	33
4-3.7. Limitations	33
 Chapitre 5 Utilisation	 35
5-1. Informations générales	35
5-1.1. Lancement de SimMU0	35
5-2. Mise en route du système	36
5-2.1. Mode Simulation	36
5-2.2. Mode intégration	36
<i>Création du fichier de configuration</i>	37
<i>Intégration dans la cible</i>	38

Chapitre 1

Le système MU0

Ce document donne une description des différentes parties du système à processeur MU0 utilisé dans le laboratoire de systèmes numériques. Celui-ci est utilisé dans le cadre de l'enseignement des systèmes à processeur (voir § 1-2, Les objectifs didactiques).

Ce système est composé:

- D'un processeur simple, le MU0
- D'un bloc contenant les entrées/sorties. Il contient des entrées/sorties de base. Il a été réalisé de façon à pouvoir ajouter de nouvelles interfaces.
- D'un bloc contenant une mémoire ROM pour le programme et une mémoire RAM pour les données

Ce système est disponible sous forme de descriptions VHDL. Ceci permet de faire une simulation matérielle de l'ensemble avec le simulateur *ModelSim* ou d'intégrer le processeur et le bloc d'entrées/sorties dans des circuits programmables.

Les outils disponibles pour utiliser ce système sont:

- Un simulateur, *SimMU0*, logiciel de mise au point d'applications basées sur le processeur MU0
- Un assembleur, *AsmMu0*. Il permet de vérifier un code assem-

bleur *MU0* (parsing) et de générer le fichier pour charger la mémoire ROM (fichier *.hex), ainsi que celui utilisé pour la simultaion (*.tcl).

- Une implémentation matérielle, la carte *Carte système MU0*

1-1 Spécification du système à processeur MU0

Le système à processeur MU0 utilise une architecture de Von-Neuman. Il a un seul espace d'adressage. La mémoire de programme, la mémoire de données et les entrées/sorties sont dans le même plan d'adressage. La figure 1-1 donne une vue de la structure du système.

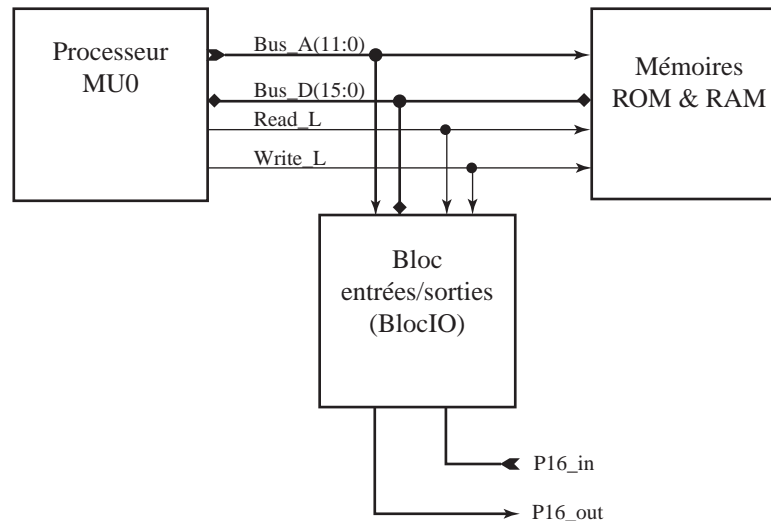


Figure 1- 1 : Structure du système à processeur MU0

1-1.1 Spécification du bus système

Le bus du système MU0 définit les signaux suivants:

- Bus d'adresses de 12 bits
- Bus de données de 16 bits
- Bus de commandes: nRead et nWrite
nRead = '0' accès en lecture
nWrite = '0' accès en écriture

1-1.2 Plan d'adressage du système à processeur MU0

Les deux tableaux suivants donnent le plan d'adressage du système. Des plages d'adresse ont été prévues pour des évolutions futures de ce système:

- Adresses 806 à 80F pour des entrées/sorties de base
- Adresses 810 à BFF pour des interfaces particulières

Adresse		Utilisation
Hex	Déc.	
000 à 7FF	0 à 2047	Mémoire de programme ROM 2Kx16 bits (seulement lecture)
800 à 80F	2048 à 2063	Réservé pour les entrées-sorties de base (voir description ci-après)
810 à BFF	2064 à 3071	Libre pour d'autres entrées-sorties
C00 à FFF	3072 à 4095	Mémoire de donnée RAM 1K x 16 bits Accès en lecture et en écriture

Tableau 1-1 : Plan d'adressage du système MU0

Adresse		Utilisé pour ..	
Hex	Déc.	Accès en lecture	Accès en écriture
800	2048	Lecture état 16OUT	Ecriture des sorties 16OUT
801	2049	Lecture état 16IN	pas utilisé
802	2050	ROR: rotation à droite	
803	2051	ROL: rotation à gauche	
804	2052	LSR: décalage logique à droite (le MSB est mis à '0')	
805	5053	LSL: décalage logique à gauche (le LSB est mis à '0')	
806 à 80F	2054 à 2050	Réservé pour de futures entrées-sorties de base	

Tableau 1-2 : Description des fonctionnalités du module E/S de base

Description du fonctionnement des ROR, ROL, LSR et LSL

- En écriture: exécute l'opération et mémorise le résultat dans un registre
- En lecture: lit le contenu du registre, donc le résultat de l'opération choisie

L'exemple suivant illustre l'utilisation du périphérique LSR:

```
LDA val ; chargement de la valeur à décaler
; ACC = LSR(ACC)
STO 0x804 ; décalage logique à droite
LDA 0x804 ; lecture du resultat mémorisé
```

Exemple 1- 1 : Utilisation d'une entrée/sorties de base

1-2 Les objectifs didactiques

Ce système est utilisé pour l'enseignement des *systèmes à processeur*. Il permet de mettre en évidence les points suivants:

- Fonctionnement d'un système à processeur
 - Vue du programmeur (simulateur seul)
 - Fonctionnement des bus, plan d'adressage
 - Ecriture de programme assembleur simple
- Fonctionnement interne d'un processeur
 - Registres internes (PC, IR, ACC)
 - Décodeur d'instructions
 - Séquencement des instructions
- Les entrées/sorties, indispensables pour interfacer le système au monde extérieur.

Pour ce faire, diverses possibilités de conception sont prévues:

- Modification du processeur
Ajout d'instructions ou de fonctionnalités pour étendre les performances, par exemple:
 - Ajout d'un nouveau mode d'adressage
 - Ajout d'une ligne d'interruption
- Conception d'interface
 - Liaison série
 - CO-processeur; multiplication, division, ...

Ce système permet aussi d'effectuer des vérifications à différents niveaux:

- Exécution d'un code assembleur dans le simulateur du CPU
- Simulation VHDL de la partie réalisée en utilisant un test-bench
- Simulation VHDL liée avec le simulateur et un programme assembleur. Co-vérification entre la description VHDL (partie matérielle) et un code assembleur (partie logicielle)
- Intégration dans une cible et utilisation d'un émulateur

Chapitre 2

Le processeur MU0

Le processeur MU0 a été développé dans un but pédagogique à l'université de Manchester. Il est présenté en introduction du livre "ARM, system-on-chip architecture" de Steve Furber. La description donnée ci-dessous correspond à la version de base de ce CPU telle que donnée par Steve Furber.

Le MU0 est constitué des blocs fonctionnels suivants:

- Un compteur de programme (PC pour Program Counter). Il s'agit d'un registre utilisé pour contenir l'adresse de la prochaine instruction à lire (exécuter)
- Un registre d'instruction (IR, Instruction Register). Ce registre contient l'instruction qui est en cours d'exécution
- Un registre nommé accumulateur (ACC pour Accumulator) qui stocke la donnée que le processeur traite
- Une unité de calcul (ALU pour Arithmetic-Logic Unit) qui peut exécuter un certain nombre d'opérations telles que l'addition, la soustraction, etc...
- Une unité de séquençement

Nous allons étudier ce processeur par étapes. Nous commencerons par le fonctionnement vu par le programmeur (vue externe). Puis nous entrerons de plus en plus dans les détails du fonctionnement (vue interne).

2-1 Modèle du programmeur du MU0

Le MU0 est un processeur RISC 16 bits. Il génère 12 bits d'adresse, lui permettant d'atteindre directement 4096 mots de 16 bits (mémoire et entrées/sorties confondues). Les instructions ont une longueur de 16 bits (une instruction = un mot mémoire de 16 bits), soit 4 bits pour le code d'opération (*opcode*) et 12 bits pour l'opérande. La figure 2-1 donne une représentation du format d'une instruction. Il est à remarquer que l'opérande à la même taille que le bus d'adresse, soit 12 bits.

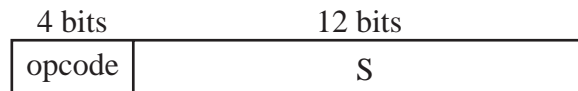


Figure 2- 1 : Format d'une instruction

Le jeu d'instructions disponible utilise seulement 8 opcodes sur les 16 disponibles (version standard du MU0). Ces instructions sont résumées dans le tableau suivant:

Instructions	Opcode	Effet
LDA S	0000	ACC:= mem ₁₆ [S]
STO S	0001	mem ₁₆ [S]:= ACC
ADD S	0010	ACC:= ACC + mem ₁₆ [S]
SUB S	0011	ACC:= ACC - mem ₁₆ [S]
JMP S	0100	PC:= S
JGE S	0101	if ACC >=0, PC:= S else PC := PC + 1
JNE S	0110	if ACC != 0, PC := S else PC := PC + 1
STP	0111	stop

Tableau 2-1 : Les instructions disponibles dans le MU0

Remarque: La notation $ACC := ACC + \text{mem}_{16}[S]$ se traduit par ajouter à l'accumulateur le contenu de la case mémoire (sur 16 bits) se trouvant à l'adresse S. Le résultat se trouvera dans l'accumulateur.

Les instructions sont lues (*fetch*) dans la mémoire à des adresses consécutives, l'adresse de départ étant l'adresse zéro. Lors de l'exécution d'une instruction de saut, le déroulement du programme continue depuis l'adresse fournie par l'instruction de saut.

2-2 Structure du système à processeur MU0

Un processeur est une machine séquentielle complexe. Il peut donc être décomposé en une unité de commande (au moins) et une unité de traitement (au moins)

L'unité de commande fournit les commandes au système afin de pouvoir exécuter l'instruction en cours. Dans le MU0, elle est réalisée sous la forme d'une machine séquentielle simple.

L'unité de traitement est de type universel. Elle est donc basée sur une ALU entourée de divers registres et aiguillages.

2-2.1 Schéma bloc général

On peut remarquer que les quatre premières instructions du tableau 2-1 nécessitent deux accès sur le bus (un pour lire l'instruction en mémoire et un pour prendre ou stocker une donnée), les quatre autres instructions peuvent s'exécuter en un cycle d'horloge, avec une UT appropriée, car elles n'effectuent qu'un accès sur le bus pour aller chercher la prochaine instruction. Une unité de traitement, possédant suffisamment de ressources pour que ces instructions soient exécutées en un seul cycle d'horloge par accès bus, est représentée à la figure suivante:

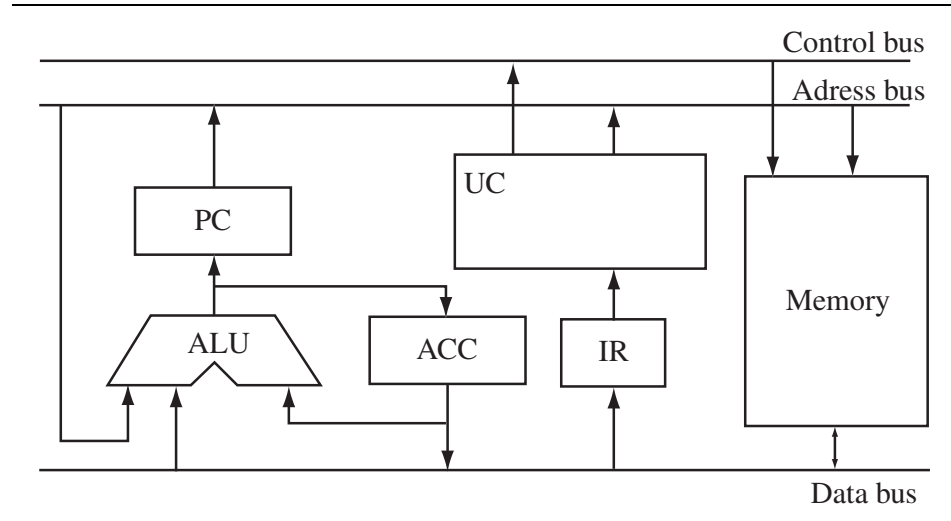


Figure 2- 2 : MU0 datapath

Avec ce design, il n'est pas nécessaire d'implémenter un incrémenteur pour le *PC*. Durant un cycle *Fetch*, l'*ALU* n'est pas utilisée. Elle est donc disponible pour effectuer l'incrémentation du *PC*.

Les instructions s'exécutent en une ou deux étapes (deux cycles), Cycle = '0' et Cycle = '1'. Certaines instructions n'ont que le Cycle = '0'.

- Le transfert d'un opérande à travers le bus et l'exécution de l'opération désirée (Execute) se déroulent dans le cycle 0
- La lecture de la prochaine instruction (Fetch) dont l'adresse est fournie par le *PC* ou le registre d'instruction se déroule dans le cycle 0 si l'instruction exécutée ne requiert pas de transfert d'un opérande à travers le bus (c'est le cas de *JMP S*, *JGE S* et *JNE S*). Si le transfert d'un opérande à travers le bus est nécessaire, la lecture de la prochaine instruction se déroule dans le cycle 1. Dans le cas des instructions de saut, le *Fetch* constitue l'exécution de l'instruction. L'exécution de l'instruction et le fetch de l'instruction suivante s'effectuent simultanément en un seul et même cycle.

2-2.2 Etude du niveau RTL (Register Transfer Level)

Il faut maintenant déterminer les signaux de commande nécessaires pour que l'unité de traitement conçue puisse exécuter le jeu d'instructions en son entier.

Un registre charge une nouvelle valeur au flanc actif de l'horloge, si la commande de permission (*Enable*) est activée. La valeur du *PC*, par exemple, sera mise à jour uniquement si le signal *PCce* est à '1' sinon il gardera son état actuel.

La figure 2-3 donne le schéma au niveau RTL du processeur. On peut y voir les *Enable* pour tous les registres, les signaux de contrôle pour l'*ALU*

ainsi que les signaux de sélection des deux multiplexeurs, le contrôle de la porte trois états qui permet d'envoyer la valeur de l'ACC sur le bus et les signaux de demande de lecture (*Read*) et d'écriture (*Write*). Les autres signaux représentés sont des sorties de l'unité de traitement vers l'unité de commande, soit le opcode et le signal indiquant si l'ACC a une valeur nulle ou négative, pour permettre le contrôle des instructions de saut.

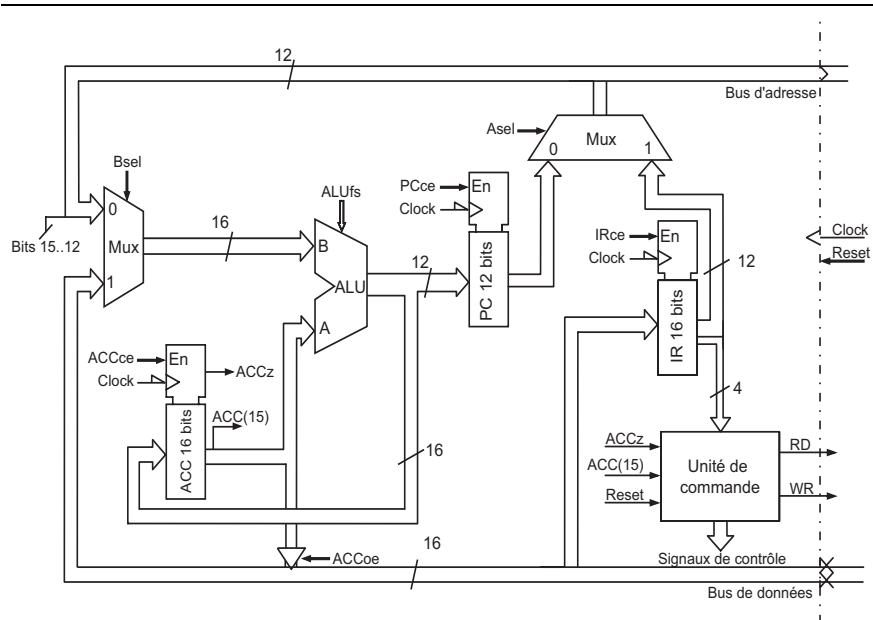


Figure 2- 3 : Organisation du *register transfer level* du MU0

2-2.3 L'unité de commande (Control Logic)

L'unité de commande décode l'instruction courante et active les bons signaux de commande de l'unité de traitement. L'unité de commande est une machine d'état donc son fonctionnement pourrait être représenté par un graphe d'états ou un organigramme. Dans le MU0, la machine d'état est triviale. Elle n'a besoin que de deux états (nombre maximum de cycles pour les instructions), et un seul bit d'état (*Cycle*) est donc suffisant.

Le décodeur d'état futur et le décodeur de sortie sont représentés par le tableau 2. Dans ce tableau le x indique une condition "don't care".

Input						Output									
Opcode	Cycle	ACC15				Bsel		PCce		ACCoe		RD		Cycle+	
Instruction	Reset	ACCz				Asel	ACCce		IRce		ALUfs		WR		
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	=0	1	0	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	=B	1	0	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	0	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	0	1	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	0	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	0	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	0	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A-B	1	0	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	0	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	0	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	0	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	0	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	0	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	0	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	0	0

Tableau 2-2 : Logique de contrôle du MU0

2-2.4 L'initialisation

Le processeur doit démarrer dans un état connu. Une entrée nommée *Reset* permet de démarrer l'exécution des instructions depuis une adresse connue. Le *MU0* commence l'exécution depuis l'adresse 000_{16} . Il faut noter que le *Reset* a une action synchrone sur les éléments du *MU0*. Pour ce faire, le signal *Reset* force à zéro la sortie de l'*ALU* et cette valeur est chargée dans le *PC* au prochain flanc actif de l'horloge. Au cycle suivant le *MU0* va chercher l'instruction pointée par le *PC* (qui vaut alors 0). Il faut donc que le *Reset* dure deux cycles d'horloge pour initialiser le processeur.

2-2.5 L'ALU

L'*ALU* doit pouvoir effectuer cinq opérations: $A + B$, $A - B$, B , $B + 1$, 0 ; la dernière opération étant utilisée lors de l'activation du *Reset*. Toutes les opérations peuvent être réalisées en utilisant un additionneur:

- $A + B$ est le résultat standard d'un additionneur (avec l'entrée du *Carry* à zéro)
- $A - B$ peut être implémenter comme $A + \text{not } B + 1$. Il faut donc

inverser l'entrée Q ($Q = C_1(B)$) et forcer l'entrée du *Carry* à 1

- B est obtenu en forçant l'entrée P à zéro et l'entrée *Carry* à zéro
- B + 1 est obtenu en forçant P à zéro et le *Carry* à 1

La figure suivante propose une représentation de l'ALU telle qu'elle a été réalisée au laboratoire. Dans cette configuration, l'ALU permet d'effectuer plus d'opération que nécessaire au fonctionnement du MU0 décrit ici. Mais cette configuration facilite de futures extensions.

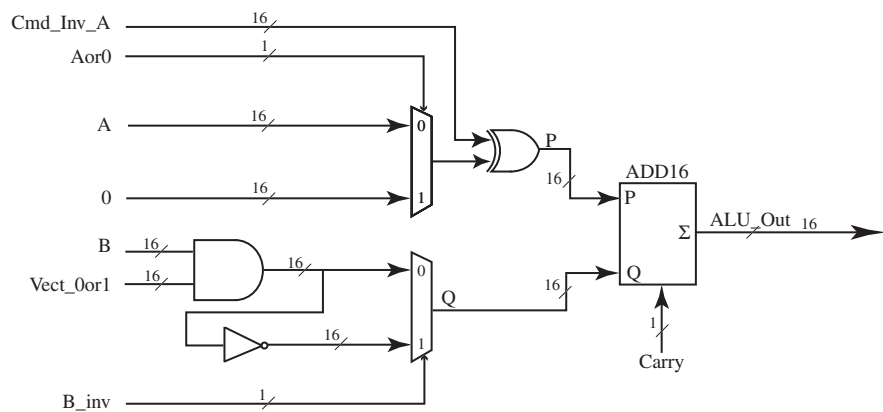


Figure 2- 4 : Illustration de l'implémentation de l'ALU effectuée au laboratoire

Le tableau suivant représente la table de fonctionnement de cette ALU

Cmd_A_inv	Aor0	B_inv	Vect_0or1	Carry	fonction ALU
0	0	0	0	0	A
0	0	0	0	1	A+1
0	0	0	1	0	A+B
0	0	0	1	1	A+B+1
0	0	1	0	0	A - 1
0	0	1	0	1	A
0	0	1	1	0	A + not (B)
0	0	1	1	1	A-B
0	1	0	0	0	0
0	1	0	0	1	1
0	1	0	1	0	B
0	1	0	1	1	B+1
0	1	1	0	0	- 1
0	1	1	0	1	0
0	1	1	1	0	not (B)
0	1	1	1	1	-B
1	0	0	0	0	not (A)
1	0	0	0	1	- A
1	0	0	1	0	not (A) + B
1	0	0	1	1	B - A
1	0	1	0	0	not (A) -1
1	0	1	0	1	not (A)
1	0	1	1	0	not (A) + not (B)
1	0	1	1	1	not (A) + not (B) -1
1	1	0	0	0	- 1
1	1	0	0	1	0
1	1	0	1	0	B - 1
1	1	0	1	1	- B
1	1	1	0	0	-2
1	1	1	0	1	-1
1	1	1	1	0	not (B) - 1
1	1	1	1	1	not (B)

Tableau 2-3 : table de fonctionnement de l'ALU du laboratoire

2-3 Informations supplémentaires

Pour des informations supplémentaires, un didacticiel est installé sur les ordinateurs du laboratoire (salle A07 et A09). Celui-ci présente le processeur simple MU0 ainsi que le processeur ARM. Vous le trouverez dans le menu:

Démarrer → Laboratoire Numérique → Embedded → ARM_CBD.

Ces informations sont tirées du livre «ARM, system-on-chip architecture», de Steve Fruber.

Chapitre 3

Réalisation du système

Le système MU0 est constitué d'un certain nombre de bibliothèques VHDL. Elles ont été réalisées avec le programme *HDL Designer*. On trouve les Bibliothèques suivantes:

- **MU0**: Il s'agit des descriptions du processeur en lui-même. Utilisée pour la simulation ou pour l'intégration.
- **BlocIO**: Elle contient le décodeur d'adresses pour les mémoires (CS_RAM et CS_ROM) et les entrées/sorties de base du système. Utilisée en simulation ou pour l'intégration.
- **SysMU0**: Elle décrit le système complet. Elle instancie le MU0 et le BlocIO. Elle contient aussi un modèle de ROM et un modèle de RAM. Utilisée en simulation uniquement.
- **FPGA_IO**: Elle sert pour l'intégration du BlocIO dans la FPGA_IO de la cible.
- **FPGA_CPU**: Elle sert pour l'intégration du processeur dans la FPGA_CPU de la cible.

Les paragraphes suivants donnent une description de ces différentes bibliothèques.

3-1 La bibliothèque MU0

Cette bibliothèque contient la description du processeur MU0. Il s'agit du MU0 standard auquel ont été ajoutées des fonctions de debug. Le séquençement a aussi été modifié. Le découpage de cette bibliothèque est le suivant:

- MU0_Std, qui correspond au MU0 standard avec une gestion du timing du bus. Deux descriptions de ce processeur ont été réalisées
- Une description synthétisable, comprenant les huit instructions de base
- Une description de spécification. Elle comprend les huit instructions de base ainsi que les instructions supplémentaires décrites dans le tableau 4-2
- MU0_top qui comprend le MU0_Std ainsi que de la logique pour le débogage.

3-1.1 Le MU0_top

Le composant MU0_top est le top de la bibliothèque MU0. Il contient une instance du MU0 standard et une partie de logique pour le débogage. La figure 3-1 représente le MU0 top.

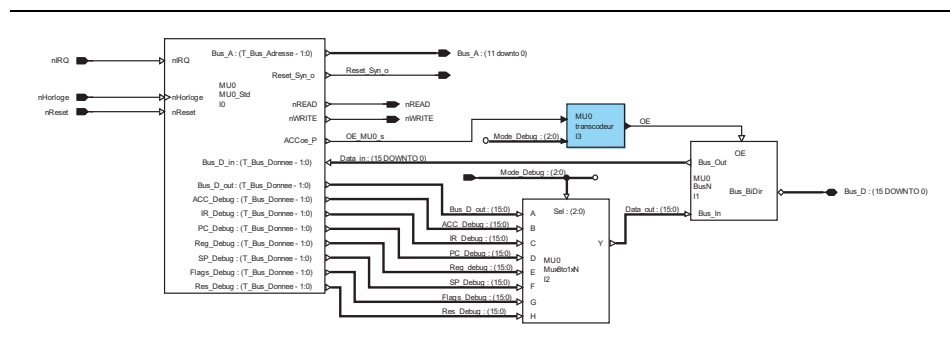


Figure 3- 1 : Structure du composant MU0_top

La logique de débogage consiste en un multiplexeur et un transcodeur. Le but est de transmettre les valeurs des registres internes du processeur (ACC, PC, IR) sur le bus de données du système. Le but est de pouvoir disposer de l'état interne du processeur à l'extérieur de celui-ci. Le signal *Mode_debug* permet de sélectionner quelle valeur est à transmettre sur le bus. Le transcodeur permet de commander la porte trois états selon le mode de fonctionnement du système (mode normal ou mode debug). L'exemple suivant donne l'architecture de ce transcodeur.

```
architecture comport of transcodeur is
begin -- comport
```

```

OE <= OE_MU0_s when Mode_Debug = "000" else -- Mode normal
'1'      when Mode_Debug = "001" else -- Lecture ACC
'1'      when Mode_Debug = "010" else -- Lecture IR
'1'      when Mode_Debug = "011" else -- Lecture PC
'1'      when Mode_Debug = "100" else -- Lecture REG
'1'      when Mode_Debug = "101" else -- Lecture SP
'1'      when Mode_Debug = "110" else -- Lecture Flags
'1'      when Mode_Debug = "111" else -- Lecture
Reserve
'0';

end comport;

```

Exemple 3- 1 : Architecture du transcodeur

3-1.2 Le MU0 Std

Le MU0 standard correspond au MU0 de base auquel il a été ajouté une gestion du timing du bus. Deux descriptions ont été réalisées: une version synthétisable et une version de spécification.

Version synthétisable

La figure 3-2 donne une vue du composant MU0_Std tel qu'il a été décrit sous HDL Designer. Il s'agit du MU0 de base (cf. figure 2-3) avec une gestion des cycles du système. Cette gestion est réalisée par les composants Gen_Ph et Dec_Ph.

D'autre part, les portes trois états ont été déplacées au niveau supérieur afin de pouvoir gérer les différents mode de debug.

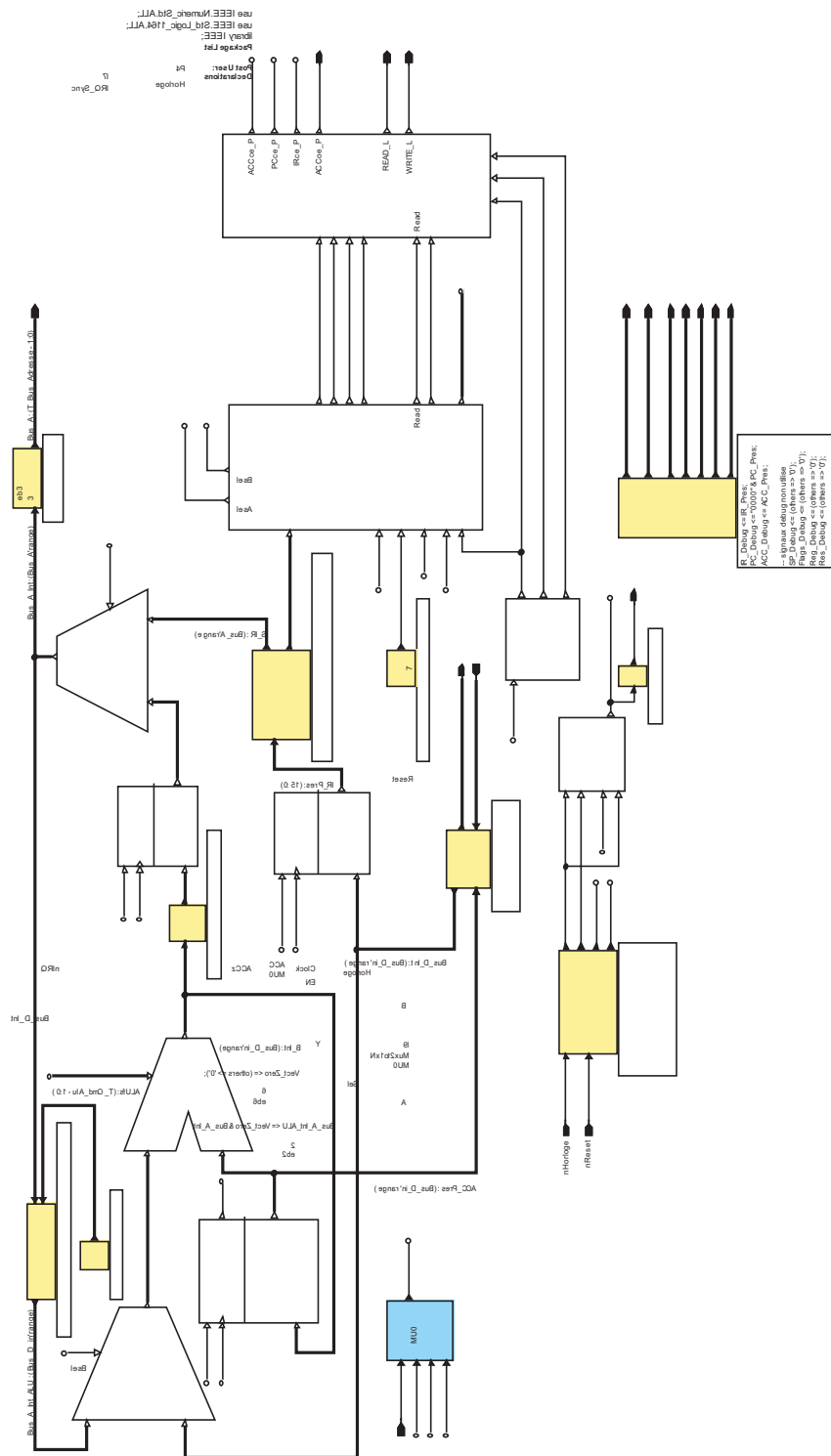


Figure 3- 2 : Structure du MU0_std (version synthétisable)

Gestion du timing interne et externe

Le but de cette gestion est d'assurer:

- Que le temps de préaffichage et de maintien des registres du CPU sont respectés lors d'une lecture

- Que le temps de préaffichage et de maintien des latches de la mémoire RAM sont respectés lors d'une écriture
- Qu'il y a un temps mort entre 2 transferts sur le bus de données, de façon à éviter des court-circuits temporaires lors des changements des sens.

Le composant Gen_Ph génère un signal d'horloge à 3 phases. La figure suivante donne un chronogramme du fonctionnement de ce composant.

Un cycle système dure quatre périodes de l'horloge du MU0.

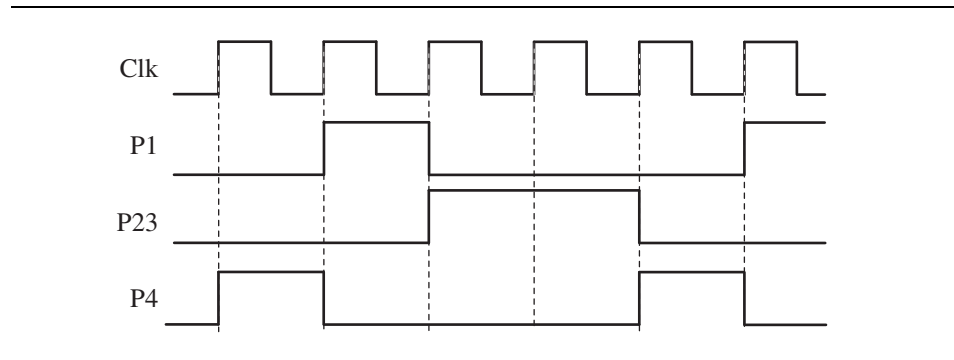


Figure 3-3 : Chronogramme du composant Gen_Phase

Le composant Dec_Ph permet d'activer les signaux de commande du système au moment opportun.

```
architecture Equations of Dec_Ph is
```

```
    signal READ_s  : Std_Logic;
    signal WRITE_s : Std_Logic;
```

```
begin
```

```
    -- Interne
```

```
    ACCce_P <= ACCce and P4;
    PCce_P  <= PCce  and P4;
    IRce_P  <= IRce  and P4;
    ACCoe_P <= ACCoe and not(P1);
```

```
    -- Externe
```

```
    READ_s  <= Read and (not(P1));
    WRITE_s <= Write and P23;
```

```
    -- Sorties actives a zero
```

```
    READ_L  <= not(READ_s);
    WRITE_L <= not(WRITE_s);
```

```
end Equations;
```

Exemple 3-2 : Architecture du composant Dec_Phase

Le timing des deux cycles du bus, lecture et écriture sont représentés aux figures 3-4 et 3-5.

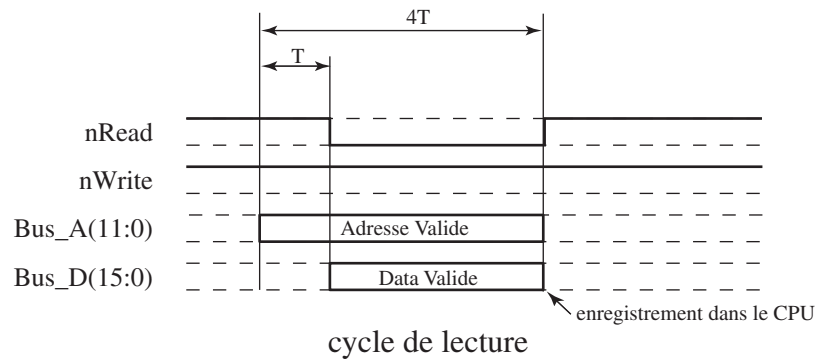


Figure 3- 4 : représentation d'un cycle de lecture

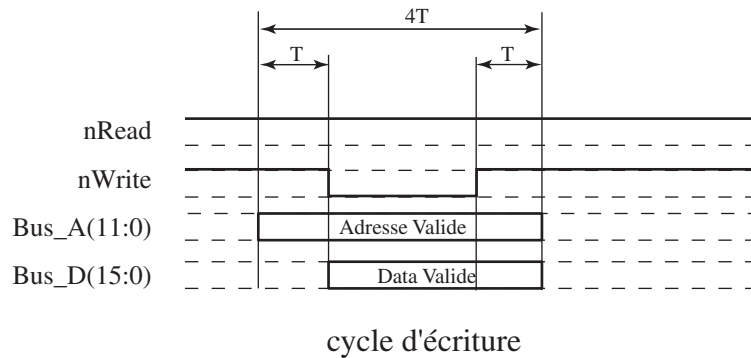


Figure 3- 5 : Représentation d'un cycle d'écriture

Version de spécification

La version de spécification du processeur permet de rapidement ajouter des instructions et de les tester dans le système. Il permet aussi de faire des tests entre cette version et la version synthétisable du MU0.

L'exemple 3-1 donne la description d'une instruction (LDA) telle qu'elle est décrite dans le modèle de spécification.

```

case Instr_Code_v is
  when LDA_S =>
    if Cycle_Pres = 0 then --cycle execute LDA
      Acc_Fut <= Signed(Bus_D); --ACC := mem[S]
      Cycle_Fut <= "1";
      Bus_A <= Operand_v;
      READ <= '1';
      WRITE <= '0';

```

Exemple 3- 3 : Code VHDL d'une instruction dans la version de spécification du MU0

3-2 La bibliothèque BlocIO

La bibliothèque BlocIO contient les entrées/sorties de base ainsi que le décodeur d'adresses pour la ROM et la RAM. La figure 3-2 donne une représentation du top de cette bibliothèque.

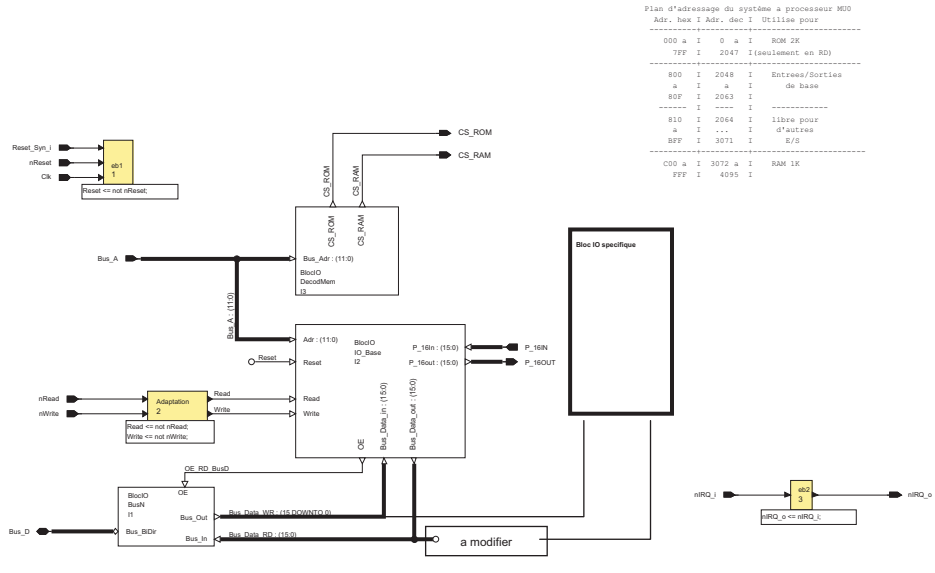


Figure 3- 6 : Représentation du BlocIO_top

Cette description est synthétisable.

3-3 La bibliothèque SysMU0

SysMU0_top est une représentation du système complet. Il est constitué d'une instance du MU0, d'une instance du blocIO et de modèles de mémoires RAM et ROM. La figure 3-3 donne une représentation de SysMU0_top.

On y trouve encore deux générateurs d'horloge, un pour le CPU et un pour les périphériques. Ceci permet de faire des simulations du système avec des horloges différentes.

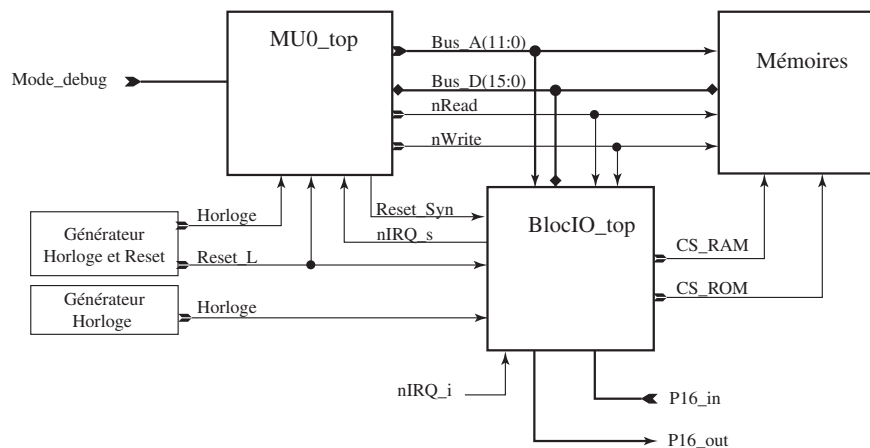


Figure 3- 7 : Représentation de SysMU0_top

Le bloc Mémoires contient un modèle pour une ROM et une RAM. Le modèle de ROM est initialisé avec un programme. Ce programme est contenu dans un fichier *.hex. Le chemin d'accès de ce programme est spécifié dans l'architecture de la ROM (ligne 46).

```
file hex_file : text open Read_mode is
    "../ASM/Prog.hex" ;
```

Exemple 3- 4 : Spécification du chemin du programme contenu en ROM

Cette bibliothèque contient des descriptions qui ne sont pas synthétisables. Elle est utilisée uniquement pour la simulation.

3-4 La bibliothèque FPGA_CPU et la bibliothèque FPGA_IO

Le rôle de ces deux bibliothèques est de faire une adaptation de signaux entre les composants de base du système (MU0 et BlocIO) et les signaux disponibles sur les FPGAs. La carte *système MU0* a été réalisée afin de pouvoir intégrer de futures évolutions de ce système. De ce fait, cette version n'utilise pas toutes les ressources de cette carte (par exemple le bus d'adresse disponible sur la carte à une largeur de 16 bits).

Chapitre 4

Les outils

Pour le développement et la simulation d'une application du système MU0, Nous disposons de 2 outils:

- Un assembleur, *AsmMu0*
- Un simulateur, *SimMU0*

4-1 Liste des instructions disponibles

Le MU0 standart (telque décrit dans le livre de Steve Furber) dispose de huit instructions, données dans le tableau 4-1. Des instructions ont été ajoutées au simulateur *SimMU0* (en mode *Internal* et en mode *Tutorial*, cf. §suivant) et à l'assembleur afin d'étendre les possibilités du système. Ces instructions supplémentaires sont données dans le tableau 4-2.

Instructions	Opcode	Effet
LDA S	0000	ACC:= mem ₁₆ (S)
STO S	0001	mem ₁₆ (S):= ACC
ADD S	0010	ACC:= ACC + mem ₁₆ (S)
SUB S	0011	ACC:= ACC - mem ₁₆ (S)
JMP S	0100	PC:= S
JGE S	0101	if ACC >=0, PC:= S else PC := PC + 1
JNE S	0110	if ACC != 0, PC := S else PC := PC + 1
STP	0111	stop

Tableau 4-1 : Instructions standards du MU0

Instructions	Opcode	Effet
LDA[R]	1000	ACC := mem ₁₆ [R]
STO[R]	1001	mem ₁₆ [R] := ACC
LDR S	1010	R := mem ₁₆ (S)
LDA#	1011	ACC := #val (uniquement dans simulateur)
CALL S	1100	PC := S, Sauve PC
RET	1101	Restaure PC
RETI	1111	Restaure PC

Tableau 4-2 : Instructions supplémentaires disponibles dans le simulateur SimMU0

4-2 L'assembleur *AsmMu0*

Cet assembleur permet de réaliser la vérification syntaxique d'un fichier source en assembleur MU0 et de le convertir en 3 fichiers de sortie pour la simulation avec *ModelSim* et avec *SimMU0* ainsi que pour la programmation d'une Flash.

4-2.1 Syntaxes acceptées par AsmMu0

La mise en page

Chaque instruction se trouvera sur une ligne différente. Les espaces et tabulations peuvent être utilisés indifféremment, dans un but de mise en page. Les lignes vides sont autorisées.

```

; Liste des equivalences permettant de
; rendre le programme plus lisible
Cpt          EQU 0xC04

; Programme principal: utilise et teste la
; sous-routine Mult16 en l'appelant TestCnt fois
Test:        LDA #4           ; Chargement de 4 dans
                                   ; l'accumulateur
                                   ; Cpt = 4
                                   STO Cpt

```

Exemple 4- 1 : Exemple de mise en page

Remarque: Chaque instruction est terminée par un retour à la ligne. Il faut prendre garde de ne pas oublier le retour à la ligne sur la dernière instruction.

Les étiquettes (labels)

Une instruction peut être référencée par une étiquette. Celle-ci doit obligatoirement commencer par une lettre. Elle ne sera suivie que par des lettres, des chiffres ou un underscore ('_'). Le caractère ':' sépare le label de l'instruction.

```

Test_1 :    LDA #4

```

Exemple 4- 2 : Les étiquettes

Les commentaires

Tout ce qui se trouve à droite d'un ';' est considéré comme commentaire. Une ligne peut n'être constituée que d'un commentaire.

```

; Programme principal: utilise et teste la
; sous-routine Mult16 en l'appelant TestCnt fois
Test:        LDA #4
                                   CALL Mult16 ; Appel de la multiplication

```

Exemple 4- 3 : Les commentaires

Les équivalences

Afin de rendre le code plus lisible, des équivalences peuvent être définies afin d'associer à une position mémoire ou E/S le nom d'une variable. Bien qu'il soit préférable de placer les équivalences en début de program-

me, elles peuvent apparaître n'importe où dans le programme (même après leur utilisation).

```
IoOut EQU 0x800 ; Definition des IOs de base

STO IoOut
```

Exemple 4- 4 : Les équivalences

La définition de constantes stockés en mémoire ROM

Il est possible de définir des constantes sur 16 bits en ROM. Il est pour cela nécessaire d'utiliser la directive `'.DATA'`. En général elle est employée conjointement avec une étiquette `y` faisant référence. Les constantes sont en principe définies en fin de programme et dans tous les cas à un endroit où le PC ne pointera jamais.

```
                LDA NbTest

NbTest:         .DATA 4
```

Exemple 4- 5 : Les constantes

Les constantes référant une étiquette

Lors de la définition de constantes, il est possible de faire référence à une adresse représentée par une étiquette (dont la valeur sera déterminée lors de l'assemblage). Le caractère '@' est utilisé à de telles fins.

```
LDR PtrTable   ; R pointe sur la constante Table
LDA [R]        ; Chargera 0x1234 dans l'accumulateur

Table:         .DATA 0x1234
PtrTable:      .DATA @Table
```

Exemple 4- 6 : Constantes référant une étiquette

Les nombres

Un nombre doit commencer par un chiffre ('0'-'9') ou éventuellement par le signe '-'. Lorsqu'il est suivi d'un `x`, cela signifie que la valeur est donnée en hexadécimal et peut par conséquent contenir les lettres ('A'-'F') en plus des chiffres.

Les équivalences et les références à une position mémoire utiliseront des nombres non signés. Les nombres signés peuvent être employé avec un `'LDA #'` ou un `'.DATA'`. La représentation d'un nombre négatif se fait par complément à 2.

Encodage lors d'un LDA #

Toute valeur sur 16 bits n'étant pas automatiquement encodable sur 12 bits, l'assembleur générera une erreur pour celles qui ne le sont pas. La règle est la suivante: le nombre à encoder, exprimé en valeur absolue, est divisé r fois par 2 jusqu'à obtenir un nombre v impair ou jusqu'à concurrence d'un maximum de 7 fois. Le résultat obtenu doit alors être inférieur à 256.

Les valeurs suivantes sont encodables:

V	0x4	0x5000	-0x4	-0x5000
Encode(V)	0x201	0x7A0	0xA01	0xFA0

Les valeurs suivantes ne le sont pas:
257, 1025, ...

4-2.2 Formats de sortie

L'assembleur peut générer 3 formats de sortie différents:

- a. hexadécimal (-hex): utilisé lors de simulation sous ModelSim par un modèle de ROM adéquat.

```
000 000c
001 1c06
002 000d
003 600a
```

Exemple 4- 7 : Fichier au format hexadécimal

- b. Intel hexadécimal (-int): utilisé par le programmeur d'EEPROM

```
:2000000001010F00C80019E30111022101310FD00640065006600D000070009000A000020014
:00020010008000006E
:00000001FF
```

Exemple 4- 8 : Fichier au format Intel

- c. Tcl (-tcl): Format utilisé en entrée par le simulateur Mu0Sim. Il est à noter que les équivalences seront résolues et non présentes dans ces données. Le nombre d'éléments de la liste Program est strictement égale au nombre d'adresses utilisées en ROM.

```

set Program { \
  {begin : LDA_A 257 } \
  {LDA_L L200 } \
  {LDA_A 200 } \
  {LDA# 793 } \
  {STO 257 } \
  {ADD_A 258 {;} { other comment}} \
  {loop : SUB_A 257 {;} { loop branch }} \
  {LDR_L L200 } \
  {JMP loop } \
  {JGE loop } \
  {JNE loop } \
  {CALL subr } \
  {STP } \
  {subr : RET } \
  {RETI {;} { comment with multiple words}} \
  {L200 : .DATA 512 } \
  {L128 : .DATA 128 } \
}

```

Exemple 4- 9 : Fichier au format TCL

4-2.3 Invocation de l'assembleur

L'assembleur étant écrit en java, il peut tourner sur un grand nombre de plate-formes: PC sous Windows ou Linux, Sun workstations ou Mac. Il nécessite par contre l'installation d'une JVM (Java Virtual Machine). Les exemples ci-dessous présupposent la mise en place du fichier classes.jar dans le répertoire courant.

L'invocation de l'assembleur peut se faire de deux manières différentes: par des commandes *DOS* ou depuis le simulateur *SimMU0*.

Invocation par commande DOS

L'invocation se fait directement depuis l'interpréteur DOS. La syntaxe de commande est la suivante:

```
$ java -cp classes.jar AsmMu0 [-hex | -tcl | -int] [filename]
```

Le type de format de sortie est donné par le premier argument optionnel. Lorsque aucun format de sortie n'est spécifié, le format hexadécimal est utilisé.

Le nom du fichier assembleur est optionnel. Lorsqu'il n'est pas présent, stdin est utilisé. L'invocation suivante est donc correcte:

```
$ cat filename | java -cp classes.jar AsmMu0
```

Cette possibilité est utile lorsqu'un pré-processing du fichier assembleur est nécessaire. Il est alors possible de chaîner les commandes dans des pi-

pes ()), donc en évitant de devoir passer par la création de fichiers temporaires.

De manière identique, les résultats de compilation se trouvent sur stdout et peuvent être redirigés dans un fichier.

```
1. Création d'un fichier pour Mu0Sim
$ java -cp classes.jar AsmMu0 -tcl test.s > test.tcl
2. Génération de données en vue de créer une EEPROM
$ java -cp classes.jar AsmMu0 -int test.s > test.int
```

Exemple 4- 10 : Ligne de commande pour génération de fichier aux formats TCL et Intel

Invocation depuis MU0Sim

Cet assembleur est appelé automatiquement lors du chargement (bouton *Load*) d'un programme assembleur depuis le simulateur *SimMU0*.

4-2.4 Numéro de version

Il est possible d'obtenir le numéro de version de l'assembleur en tapant la commande suivante

```
$java -cp classes.jar AsmMu0 -version
```

4-3 Le Simulateur SimMu0

Le simulateur *SimMU0*, écrit en Tcl/Tk, est, tout comme *AsmMu0*, utilisable sur un grand nombre de plateformes. Pour fonctionner, il nécessite l'installation d'un interpréteur Tcl/Tk comme wish (version 8.0 ou ultérieure). La fenêtre principale de *SimMU0* est représentée à la figure suivante.

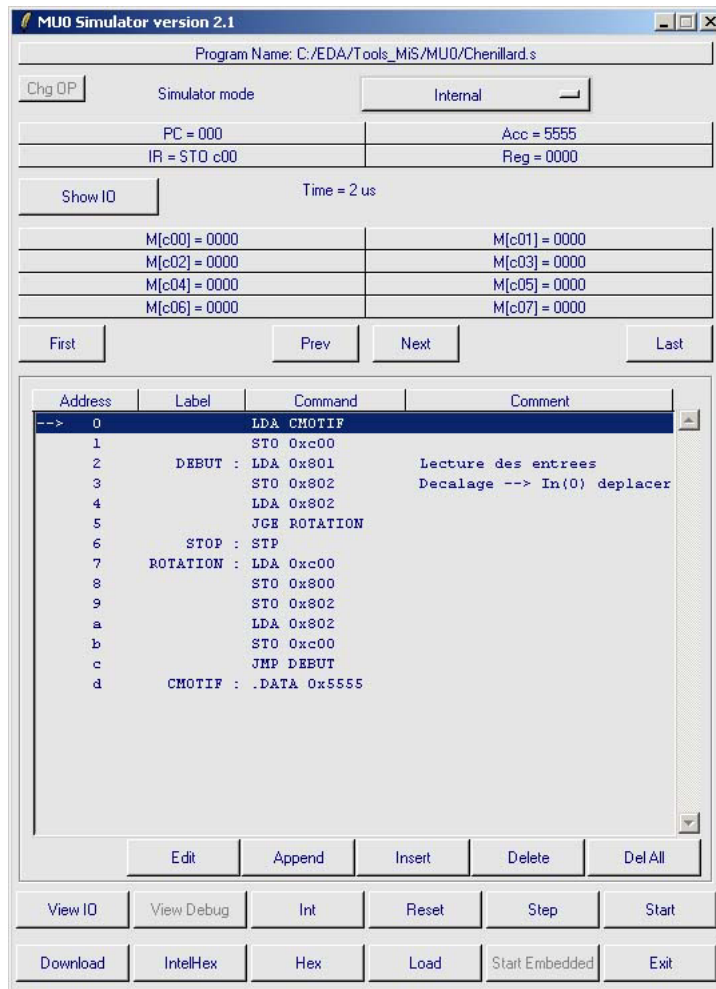


Figure 4- 1 : Fenêtre principale de simulateur pour MU0, *SimMU0*

La figure suivante donne une vue de la fenêtre représentant les ports d’entrée et de sortie du système.

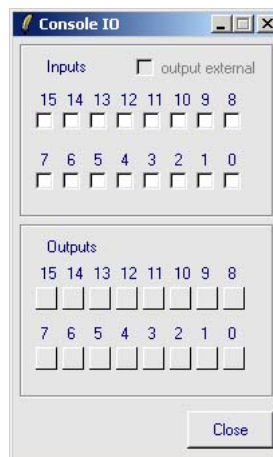


Figure 4- 2 : Fenêtre représentant les IOs du système

4-3.1 Modes de fonctionnement de *SimMU0*

Le simulateur possède 4 modes de fonctionnement:

- Internal : le simulateur fonctionne de façon autonome
- Tutorial: le simulateur fonctionne de façon autonome. De plus il affiche une représentation du processeur affichant la valeur du flot de données
- ModelSim : le simulateur ne se comporte plus qu'en interface utilisateur de ModelSim en contrôlant le déroulement de la simulation et en mettant à jour les valeurs affichées.
- Target: le simulateur affiche les valeurs lues dans la cible matériel.

Une auto-détection est faite dans *SimMU0* de sorte à vérifier s'il a été lancé sous ModelSim ou si la cible est connectée. Le choix possible entre les différents modes est déterminé en fonction de cette détection.

4-3.2 Fonctionnalités de base

Le simulateur est à la base un éditeur de programmes orienté sur le jeu d'instructions du MU0. La rangée de boutons sous l'affichage du programme est utilisée pour cela :

- Edit : édite l'instruction sélectionnée (équivalent à un double-click sur l'instruction)
- Append : ajoute une instruction en fin de programme
- Insert : insère une instruction immédiatement après l'instruction sélectionnée
- Delete : efface l'instruction sélectionnée
- Del All : efface tout le programme

Il est également possible d'utiliser le bouton de droite de la souris afin d'insérer une instruction. Les choix *Insert before* et *Insert after* permettent de définir si l'instruction s'insère immédiatement avant ou après l'instruction courante.

Lors de l'édition ou de l'ajout d'une instruction, le choix de LDA, ADD, SUB ou LDR peut se faire de deux manières :

- Avec *_L* : la boîte de dialogue suivante présentera tous les labels référencables
- Avec *_A* : la boîte de dialogue suivante donnera la possibilité de spécifier une adresse à l'aide de 3 digits hexadécimaux.

L'opérande d'un LDA immédiat est également donné à l'aide de 3 digits hexadécimaux. Elle est donc donné sous forme encodée. Le décodage se fera automatiquement lors de la simulation.

Le bouton de droite permet aussi de gérer l'utilisation de points d'arrêts (breakpoints) dans le programme :

- Add breakpoint :
place un point d'arrêt sur l'instruction sélectionnée
- Clear breakpoint :
efface l'éventuel point d'arrêt placé sur l'instruction sélectionnée
- Clear all breakpoints :
Efface tous les points d'arrêts.

Il est possible d'exécuter le programme en pas par pas (bouton *Step*) ou de manière continue (bouton *Start*). Dans ce dernier cas, le programme tournera à sa vitesse maximale dès qu'un point d'arrêt est mis en place dans le programme (même placé à un endroit bidon). Le bouton *Start* est converti en bouton *Stop* lorsque le programme tourne. Il permet à l'utilisateur de reprendre la main et de voir où le programme se trouve. Lorsque aucun point d'arrêt n'est utilisé, le programme est artificiellement ralenti de sorte à pouvoir suivre visuellement son exécution.

Le bouton *Int* simule l'apparition d'une interruption (branchement à l'adresse 1). Le bouton est désactivé (en grisé) tant que le retour d'interruption n'est pas effectué (instruction *RETI*) ou qu'un redémarrage n'est pas demandé (bouton *Reset*).

Lors de redémarrage (bouton *Reset*) il est possible de remettre à zéro la mémoire ou/et les entrées/sorties. La visualisation du port de sortie 0x800 peut être faite sous forme de LEDs et le contrôle des pins du port d'entrée placé en 0x801 peut se faire à l'aide de switches. Pour cela utiliser le bouton *View IO*.

4-3.3 Valeurs affichées

SimMu0 affiche les valeurs suivantes :

- PC : compteur de programme
- Acc : contenu de l'accumulateur
- Reg : contenu du registre d'index, le registre est disponible dans le simulateur en mode Internal et Tutorial mais pas dans les autres modes.
- IR : contenu du registre d'instruction
- Time : temps courant [ms]
- Mémoire : par 128 banques de 8 mémoires (de 0xC00 à 0xFFF)
- IO : par 128 banques de 8 IOs (de 0x800 à 0xBFF).

Toutes ces valeurs, à l'exception de IR, sont modifiables par l'utilisateur en mode Internal.

4-3.4 Formats d'entrée/sortie du simulateur

Le simulateur peut lire des fichiers aux formats :

- tcl : format natif. Lecture se faisant sans l'intervention d'un programme tiers.

- s : format assembleur. L'assembleur *AsmMu0* est implicitement appelé. Il est par conséquent nécessaire de disposer du fichier *classes.jar* dans le répertoire courant ou dans le répertoire prévu, soit *C:\EDA\Tool_REDS\MU0*.

De manière identique, il est possible de sauver le programme dans ces 2 formats. De plus, *SimMu0* peut générer les formats Hex (bouton *Hex*) et Intel Hex (bouton *IntelHex*). Ceci se fait sans l'utilisation de *AsmMu0*.

4-3.5 Ajout d'instruction

Dans les modes d'utilisation *Target* et *ModelSim*, il est possible de modifier la liste des instructions de base. Pour ce faire, on dispose du bouton *Chg OP*.

4-3.6 Mode *Target*

Dans le mode *Target*, il est possible de programmer la ROM du système, représentée par une mémoire de type flash, en pressant le bouton *Download*.

Il est aussi possible de rendre le système totalement embarqué à l'aide du bouton *Start Embedded*.

Pour la gestion des entrées/sorties, les commandes se font, par défaut, depuis l'extérieur. Pour pouvoir commander ces entrées/sorties depuis le simulateur, il faut décocher la case *output external* (dans la fenêtre IO).

Il est aussi possible de visualiser les registres internes type le pointeur de pille à l'aide du bouton *View Debug*. Il ne faut pas oublier de faire les connexions au niveau de *HDL Desinger* !

4-3.7 Limitations

Lors de la reprise de programmes édités en dehors de *SimMU0* (par exemple à l'aide de *UltraEdit*), les équivalences seront perdues. Par conséquent il est déconseillé de modifier un programme sous *SimMU0* et de le sauver ensuite en *.s*, dès que des équivalences sont utilisées. De même, toute ligne ne comportant qu'un commentaire (ou une ligne blanche de mise en page) sera perdue.

Chapitre 5

Utilisation

Ce chapitre décrit les marches à suivre pour l'utilisation du système MU0 dans ces divers modes de fonctionnement, soit:

- Internal: Mode autonome (simulation logiciel)
- Tutorial: fonctionne comme le mode Internal, mais affiche en plus une représentation schématique du MU0 en donnant les signaux de commandes qui sont actifs.
- ModelSim: Mode de simulation avec les descriptions VHDL
- Target: Simulation avec la carte cible (intégration)

5-4 Informations générales

5-4.1 Lancement de *SimMU0*

Il y a plusieurs possibilités pour lancer le simulateur MU0.

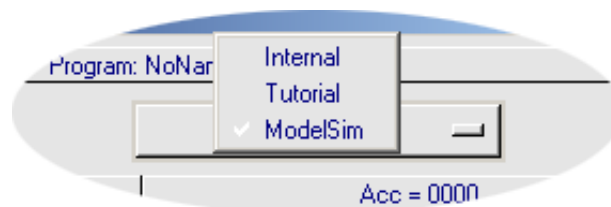
- Depuis le menu Démarrer Labo → numérique embedded → MU0 → SimMU0. Lancer de cette manière, *SimMU0* propose les modes Internal et Tutorial. Si la carte *système MU0* est branchée, le mode Target est aussi disponible.

- Depuis *ModelSim*. Dans la barre d'outils de la fenêtre principale de ModelSim se trouve un bouton se nommant *SimMU0*. Ce bouton lance *SimMU0* et ajoute à la fenêtre Wave les signaux de base du système.

5-5 Mise en route du système

5-5.1 Mode Simulation

- Copiez le dossier *SysMU0_proj* contenant les différentes bibliothèques du projet SysMU0 dans le dossier de travail
Chemin conventionnel: D:\Classe\Groupe\SysMU0\...
- Lancez *HDL Designer* et ouvrez le projet *sysMU0.hdp*
- Compilez le projet dans *HDL Designer*, *tasks compile*, en sélectionnant *SysMU0_top_tb*.
- Lancez la simulation depuis *HDL Designer*, *tasks simulate ModelSim* démarre et le projet est chargé dans le simulateur.
- Lancement du simulateur *SimMU0*
Le lancement du simulateur se fait grâce au bouton se trouvant dans la fenêtre principale de *ModelSim*. Lors du chargement de *SimMU0*, il vous sera demandé de faire un Restart du *ModelSim* et les signaux de base seront ajoutés dans la fenêtre de *Wave*.
- Vérifiez le mode de fonctionnement de *SimMU0*. Dans notre cas, il faut sélectionner le mode ModelSim comme présenté dans la figure suivante



- Pour charger un programme, sélectionnez le bouton *Load* de *SimMU0*. Sélectionnez le programme (fichier *.tcl ou *.s) et validez par *ouvrir*. Confirmez le redémarrage du simulateur en validant *Restart*.
- Vous pouvez démarrer la simulation soit pas à pas en pressant sur le bouton *Step*, soit en continu en pressant sur le bouton *Start*, la simulation est stoppée en pressant sur le bouton *Stop*.

5-5.2 Mode intégration

Dans cette section, il est décrit, dans un premier temps, les différentes étapes pour générer les fichiers de configuration. Ensuite on trouve les étapes pour l'intégration proprement dite.

Création du fichier de configuration

Afin de pouvoir intégrer le MU0 ou le bloc de périphériques dans les FPGAs, il existe deux bibliothèques qui font le lien entre les FPGAs et les instances des modules de base. Ce sont `FPGA_CPU` et `FPGA_IO`.

a. HDL Designer


1. Ouvrir la bibliothèque désirée depuis le *Design Manager* de HDL Designer.
2. Générez le fichier d'entrée pour le synthétiseur *Precision* (*_concat.vhd). Pour ce faire, sélectionnez soit `FPGA_IO_top` ou `FPGA_CPU_top` dans la librairie créée au point 2., puis lancez le tasks *synthesis*.

b. Synthèse, outil: Precision

1. Lancez le programme *Presicion* en lançant le tasks *Synthesis* depuis HDL Designer. Attention à bien sélectionner le composant à synthétiser (`FPGA_IO_top` ou `FPGA_CPU_top`)
2. Onglet *Design*, bouton *Setup Design*
Dans la fenêtre qui apparaît, choisissez: **Altera Acex EP1K30QC208-3** → Validez avec le bouton *OK*.
3. Lancez la compilation avec le bouton *Compile*, onglet *Design*
4. Dans le menu *File* → *Run Script...* Sélectionnez le fichier `FPGA_IO.TCL` ou `FPGA_CPU.tcl`
5. Lancez la synthèse avec le bouton *Synthesize*, onglet *Design*

Remarque: Pour plus d'information sur l'utilisation de *Precision*, veuillez vous référer au document "*Introduction aux logiciels de Mentor Graphics*".

c. Quartus II

1. Créez un nouveau projet en cliquant sur le bouton  1, une fenêtre s'ouvre.
Sélectionnez le répertoire `P_R` de la bibliothèque contenant le composant (répertoire `..\SysMU0_proj\FPGA_IO\P_R`) et validez en cliquant sur le bouton "*Créer*".
2. Ouvrez la fenêtre du compilateur, Menu *Quartus II* → *Compiler Tool*.
3. Choisissez les options pour la configuration du circuit.
Choisissez *Assignments* → *Device...* . Dans la fenêtre qui s'ouvre, cliquez sur le bouton *Device & Pin Options...* Dans la nouvelle fenêtre, sous l'onglet *General*, sélectionnez (cocher) les options suivantes:
 - **Enable device-wide output enable (DEV_OE)**
 - **Enable INIT_DONE Output**
Validez en cliquez *OK* (deux fois)
4. Cliquez sur le bouton *Start*

Remarque: Il est important de sélectionner les options de configuration de la FPGA. Ceci évite des courts-circuits sur la carte.

Intégration dans la cible

- a. Branchez le câble USB sur la carte.
- b. Dans le menu *Démarrer->Labo numérique->embedded->MU0*, choisissez *SimMU0*.
Cette action permet de configurer les FPGAs dans un mode d'attente et de lancer le *SimMU0* en mode *Target*
- c. Câblez le circuit avec un câble Byteblaster MV. Attention à bien choisir le connecteur relatif à la FPGA que l'on veut modifier la configuration (FPGA_IO ou FPGA_CPU)
- d. Ouvrez la fenêtre du programmeur (menu MaxPlus+II --> programmer). Configurez le circuit avec le bouton Configure.
- e. Dans le simulateur, cliquez sur Load et chargez le fichier *Votre_Programme.s*.
- f. Cliquez sur Download. Chargement du programme dans la mémoire Flash de la cible
- g. Système prêt au fonctionnement.
Vous pouvez cliquer à choix sur Step ou Start pour exécuter le code.