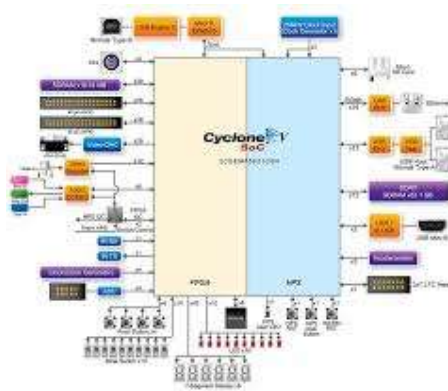


IP AXI4-lite avec I/O de la FPGA

SYSTÈME SOC INTÉGRÉ AVEC FPGA (SOCF)



Auteur : Spinelli Isaia
Prof : Etienne Messerli
Ing : Sébastien Masle
Date : 03.04.2020
Salle : A09 (maison) – HEIG-VD
Classe : SOCF

Table des matières

Introduction.....	- 3 -
Première partie : sans interruption.....	- 3 -
Plan d'adressage.....	- 3 -
Conception	- 4 -
Ecriture	- 5 -
Lecture.....	- 6 -
Description VHDL.....	- 7 -
Test et validation de l'IP	- 8 -
Écriture	- 8 -
Lecture.....	- 9 -
Création du composant.....	- 9 -
Ajout du composant	- 10 -
Modification du top.....	- 11 -
Validation pratique.....	- 11 -
Réalisation de la spécification	- 12 -
Deuxième partie : avec interruption	- 13 -
Plan d'adressage.....	- 13 -
Conception	- 14 -
Test de l'IP	- 15 -
Mise à jour dans Qsys.....	- 15 -
Test de l'IP avec le code C	- 16 -
Réalisation de la spécification	- 16 -
Compléter le code C	- 16 -
Modifier la configuration mémoire.....	- 17 -
Compiler et tester.....	- 17 -
Fonctionnalité de strobe	- 18 -
Description VHDL.....	- 19 -
Test de la fonctionnalité strobe (test bench).....	- 19 -
Test de la fonctionnalité strobe (code)	- 20 -
Supplémentaire : Gestion Edge	- 21 -
Plan d'adressage.....	- 21 -
Description VHDL.....	- 21 -
Mise à jour du projet	- 22 -
Test de fonctionnalité	- 22 -
Annexes	- 23 -

Conclusion	- 23 -
Difficultés rencontrées	- 23 -
Compétences acquises	- 23 -
Résultats obtenus	- 23 -

Introduction

Ce laboratoire a pour but de réaliser une IP avec une interface AXI4-lite et connectée sur le bus Lightweight HPS-to-FPGA. Cette IP doit permettre d'accéder à des I/O câblées sur la partie FPGA via des registres. Je dois analyser le fonctionnement du bus AXI4-lite afin de concevoir une IP personnalisée pour les besoins du laboratoire

Première partie : sans interruption

L'objectif est d'interfacer à l'aide d'une IP AXI4-lite tous les I/O disponibles sur la FPGA, sans utiliser des composants PIO, soit les boutons (KEYs), les switchs (SW), les LEDs et les afficheurs 7 segments.

Mon IP AXI4-lite comprend une constante 32 bits à l'offset 0x0 ainsi qu'un registre de test R/W à l'offset 0x4. Les offsets sont relatifs à l'adresse de base donnée à l'instance de l'IP dans Qsys.

Plan d'adressage

Pour commencer, j'ai conçu un plan d'adressage afin mettre au claire les différents aspects de mon interface.

<i>N</i>	<i>Offset</i>	<i>D32</i> <i>Read</i> <i>0</i>	<i>D32</i> <i>Write</i> <i>0</i>	<i>I / O</i>
0	0x0000 0000	Constante (0xDEADBEEF)	not used	Test
1	0x0000 0004	[31..0] regTest	[31..0] regTest	
2	0x0000 0008	Reserved	Reserved	
3	0x0000 000C	Reserved	Reserved	
4	0x0000 0010	Reserved	Reserved	
5	0x0000 0014	Reserved	Reserved	
...	...	Reserved	Reserved	
64	0x0000 0100	[31..10] '0..0' - [9..0] dataLEDs (9..0)	[31..10] reserved - [9..0] dataLEDs (9..0)	Leds
...	...	Reserved	Reserved	
128	0x0000 0200	[31..4] '0..0' - [3..0] dataKeys (3..0)	not used	Keys
129	0x0000 0204			
...	...	Reserved	Reserved	
192	0x0000 0300	[31..10] '0..0' - [9..0] dataSwitchs (9..0)	not used	Switchs
...	...	Reserved	Reserved	
256	0x0000 0400	[31..7] '0..0' - [6..0] dataHEX0 (6..0)	[31..7] reserved - [6..0] dataHEX0 (6..0)	7seg (Le point n'est pas connecté)
260	0x0000 0410	[31..7] '0..0' - [6..0] dataHEX1 (6..0)	[31..7] reserved - [6..0] dataHEX1 (6..0)	
264	0x0000 0420	[31..7] '0..0' - [6..0] dataHEX2 (6..0)	[31..7] reserved - [6..0] dataHEX2 (6..0)	
268	0x0000 0430	[31..7] '0..0' - [6..0] dataHEX3 (6..0)	[31..7] reserved - [6..0] dataHEX3 (6..0)	
272	0x0000 0440	[31..7] '0..0' - [6..0] dataHEX4 (6..0)	[31..7] reserved - [6..0] dataHEX4 (6..0)	
276	0x0000 0450	[31..7] '0..0' - [6..0] dataHEX5 (6..0)	[31..7] reserved - [6..0] dataHEX5 (6..0)	
...	...	Reserved	Reserved	
1023	0x0000 0FFF	Reserved	Reserved	

Figure 0-1 : Plan d'adressage (partie 1)

L'interface dispose de 12 bits adressables ce qui représente 4Ko avec un bus de 32bits d'adresse et de donnée.

On peut voir à l'offset 0 une constante d'une valeur de 0xDEADBEEF afin quel la valeur soit facilement reconnaissable. Cette constant sera disponible seulement en lecture et non pas en écriture.

À l'offset 0x4 il y a un registre de test accessible en écriture et lecture afin de tester facilement l'interface. Étant donné que je dispose d'une grande plage d'adresse, je me suis permis afin de facilité le décodage d'adresse de laisser un offset de 0x100 entre chaque I/O de mon interface.

Comme on peut le voir, à l'offset 0x100, il y a les leds accessible en écriture ainsi qu'en lecture. Comme il y a que 10 leds, uniquement les 10 premiers bits sont utilisés et les autres (31 à 10) sont réservés en cas d'écriture et une valeur de 0 sera retourné en cas de lecture. **Cela signifie qu'une écriture sur ces bits réservés n'aura aucun effet.**

Ensuite, à l'offset 0x200, il y a les inputs des 4 Keys qui sont accessible uniquement en lecture. En cas d'écriture à cette adresse, il n'y aura aucun effet. On peut remarquer une ligne noire en dessous car il est demandé plus tard de gérer les interruptions et donc une ou plusieurs adresses sera nécessaires pour la gestion de ces interruptions.

À l'offset 0x300 il y a les 10 switches accessibles uniquement en lecture comme les Keys. En cas d'écriture à cette adresse, il n'y aura aucun effet.

A partir de l'offset 0x400, il y a les 6 afficheurs 7 seg décaler avec un offset de 0x10. Par exemple, le premier afficheur est à l'offset 0x400 et le seconde à 0x410. Ces différents afficheurs sont accessibles en lecture ainsi qu'en écriture. Uniquement les 7 premiers bits sont utilisés pour les 7 segments étant donné que le point n'est pas branché.

La plage d'adresse s'étend jusqu'à un offset de 0xffff car l'interface dispose de 12bits. Toutes les adresses non utilisées sont pour l'instant réservées et sera peut-être utilisées plus tard. J'ai décidé qu'en cas de lecture à une adresse non utilisée, cela n'aura aucun effet.

Conception

Je dois dire qu'au début de ce laboratoire j'étais perdu, je ne savais pas par quoi commencer. De ce fait, comme cela me faisait penser à IFS, j'ai commencé faire un petit schéma pour représenter grossièrement mon interface :

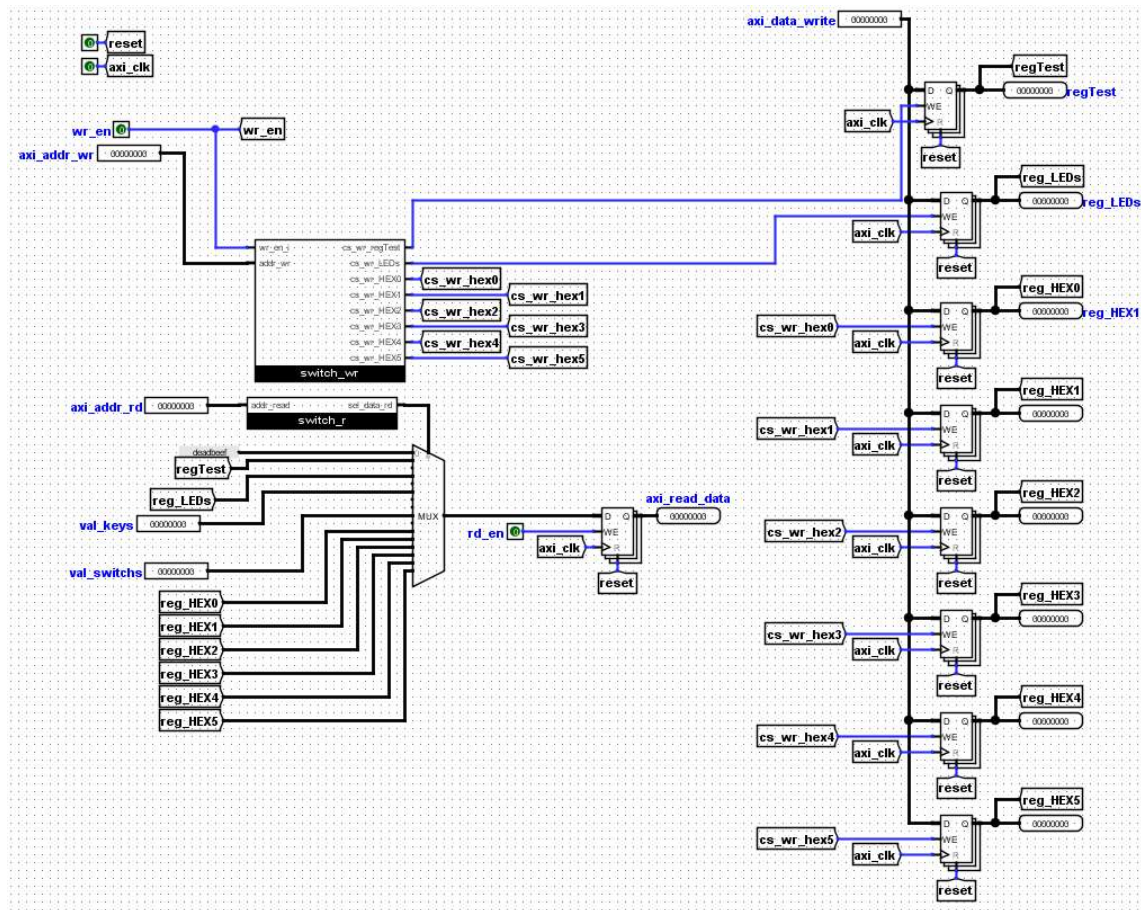


Figure 0-2 : Schéma grossier de l'interface

Au lieu d'avoir des décodeurs, des simples switches seront utilisé pour décoder l'adresse pour une lecture et une écriture. Cette étape m'a éclairé afin de mieux percevoir le système général.

Une fois que le concept m'est paru plus claire, j'ai commencé par lire le document fourni « designing_a_custom_axi_slave_rev1.pdf ». Ce document m'as fait comprendre plus exactement les étapes à réaliser.

Ensuite, j'ai analysé le code fourni, ce qui m'a encore aidé pour commencer à implémenter l'interface AXI4-lite.

Ecriture

Une bonne partie du code VHDL pour l'écriture d'une adresse et des données été déjà écrite. J'ai pu compléter le reste en m'inspirant du code déjà fourni et surtout en analysant le document donné qui explique comment designer un bus axi slave.

Après avoir lus entièrement le chapitre sur la transition d'une écriture, j'ai pris connaissance de chaque signal du bus et des différents canaux. Ce qui m'a le plus aidé à finir l'implémentation est le chronogramme dans le document fourni :



Figure 0-3 : Chronogramme d'écriture sur un bus AXI light

Dans ce chronogramme, on peut voir tous les signaux utiles pour une transaction d'écriture du master au slave. De plus, on peut voir les différents timings ainsi que les 3 canaux utilisés :

1. Le canal d'adresse et de contrôle
2. Le canal des données et de paramètre (strobe)
3. Le canal de réponse

C'est trois canaux sont indiqués par les bandes bleus sur le chronogramme. De plus, on peut voir qu'il est possible d'utiliser deux canaux simultanément. Ici, on écrit l'adresse et les données en même temps.

Le paramètre strobe, envoyé en même temps que les datas, indique quel octet nous souhaitons écrire.

Lecture

Le canal de l'adresse de lecture était déjà implémenté. Cependant, celui des données ne l'était pas du tout. Comme pour la partie écriture, je me suis grandement aidé du document fourni « designing_a_custom_axi_slave_rev1.pdf ». Celui m'a permis de connaître les deux canaux de lecture et tous les signaux utiles à une lecture.

Un chronogramme pour la lecture est documenté. Celui aussi m'a beaucoup aidé pour les timings de la transaction :



Figure 0-4: Chronogramme de lecture sur un bus AXI light

On peut voir les deux différents canaux :

1. Le canal d'adresse et de contrôle
2. Le canal de donnée et de réponse

On indique l'adresse qu'on souhaite lire et au flanc montant suivant, la donnée est prête à être lue.

Finalement, après avoir réalisé un petit schéma avec logisim, étudier les documents fournis, analyser le code déjà écrit et surtout m'inspirer des chronogrammes, j'ai pu concevoir l'IP demandée avec une interface AXI4-lite

Description VHDL

La description VHDL de l'interface du bus AXI4-lite est en annexe.

Test et validation de l'IP

Pour commencer, j'ai testé l'IP afin de valider son fonctionnement. Pour ce faire, j'ai utilisé le test Bench fourni qui teste la validité des accès en lecture et écriture.

Écriture

J'ai commencé par tester l'accès en écriture. Au début quelques timing n'était pas respecté, j'ai donc dû modifier un peu mon IP. Après un certain nombre de correction, j'ai obtenu le chronogramme suivant :

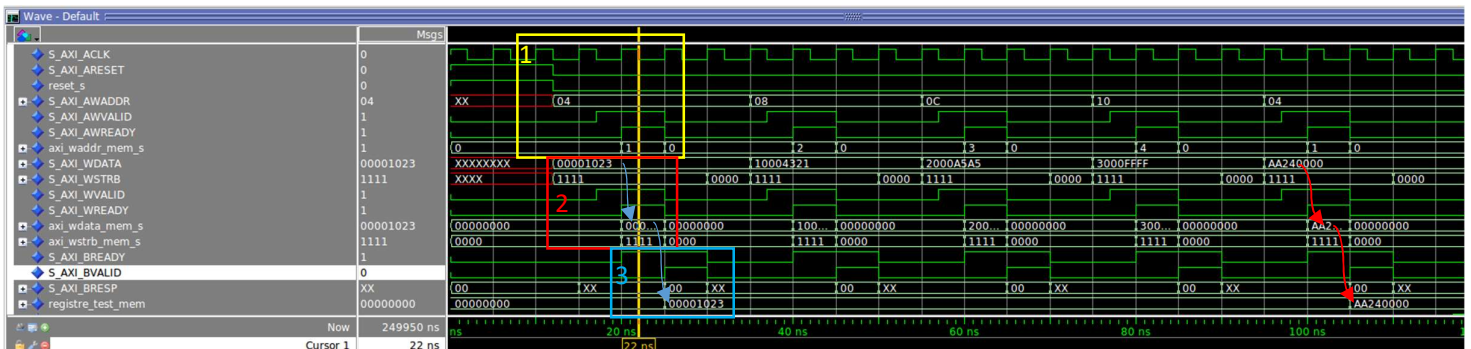


Figure 0-5 : Écriture avec le test Bench

1. Après le reset du début, on peut voir qu'un accès d'écriture va être effectué à l'adresse 0x04. Une fois que le master a levé le signal AWValid qui indique que l'adresse est valide, et que le slave est prêt (AWReady à 1) on voit le signal « axi_waddr_mem_s » qui enregistre la valeur de l'adresse. L'adresse 0x4 correspond à l'adresse 0x1 car les 2 bits de poids faible sont ignorés car nous travaillons avec des mots de 32 bits.
2. Simultanément, les données à écrire ainsi que le paramètre strobe est envoyé. Après que le master est indiqué que les données et le paramètre strobe sont valide (WValid à 1), et que le slave est prêt à les lire (WReady à 1), les données et le paramètre strobe sont enregistrés dans les signaux correspondant (axi_wdata_mem_s et axi_wstrb_mem_s).
3. Lorsque le master est prêt à lire la réponse (BReady à 1) et ensuite que le slave à une réponse valide (BValid à 1), le slave envoie la réponse et effectue l'écriture.

Les trois prochaines écritures se passent correctement mais elles sont faites à des adresses pas prises en compte par mon IP. Cependant la dernière écriture s'effectue aussi à l'adresse 0x4, donc le registre de test est de nouveau affecté par la nouvelle valeur donnée.

Après chaque transaction, j'ai décidé de remettre des valeurs par défaut afin de bien voir les transitions. Par exemple, BResp passe à chaque fois à XX après les transactions et il en va de même pour les signaux internes (axi_waddr_mem_s, axi_wdata_mem_s et axi_wstrb_mem_s passe à 0).

Lecture

Après avoir testé et validé la partie écriture de mon IP, j'ai commencé à tester la partie lecture. Une fois avoir obtenu le chronogramme ci-dessous, j'en ai déduit que la partie lecture était correcte.

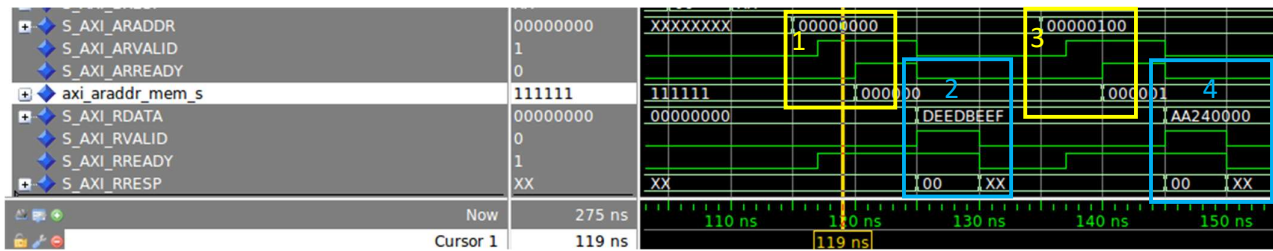


Figure 0-6 : Lecture avec le test Bench

1. Le master commence par indiquer quelle adresse il souhaite lire. Dans le premier l'adresse est 0x0, ce qui correspond à ma constante (0xdeedbeef). Une fois que le master indique que l'adresse est valide et que le slave est ready, l'adresse est lue et enregistré dans le signal interne axi_araddr_mem_s.
2. Une fois qu'une adresse a été enregistrée par le slave et que le master est prêt à recevoir la réponse, le slave peut envoyer sur le bus de lecteur (S_AXI_RDATA) les données à l'adresse souhaité ainsi que le signal de réponse (S_AXI_RRESP).
3. L'étape est la même qu'au point 1, mais l'adresse souhaitée est « 100 » (0x4) ce qui correspond à un offset de 1 étant donné que nous travaillons par mot de 32 bits.
4. L'étape est la même qu'au point 2. La valeur 0xAA240000 écrite précédemment dans la partie écriture (Figure 0-7) est maintenant relue.

On peut voir que la chaîne complète fonctionne, écriture suivie d'une lecture grâce au registre de test à l'offset 0x4. Maintenant que d'après le test bench mon IP fonctionne correctement je souhaite le vérifier à l'aide d'un petit code C qui permettra d'écrire simplement les switches sur les leds. Pour cela, je dois maintenant créer et ajouter mon IP dans mon projet VHDL.

Création du composant

Comme indiqué dans la donnée du laboratoire, j'ai créé un composant dans mon projet de Qsys afin d'ajouter mon IP à Qsys. Malheureusement j'ai perdu du temps à cause d'une petite erreur stupide. Je n'ai pas tout de suite cliqué sur le bouton « Analyze Syntheses Files », j'ai donc ajouté les signaux manuellement et les noms correspondaient pas.

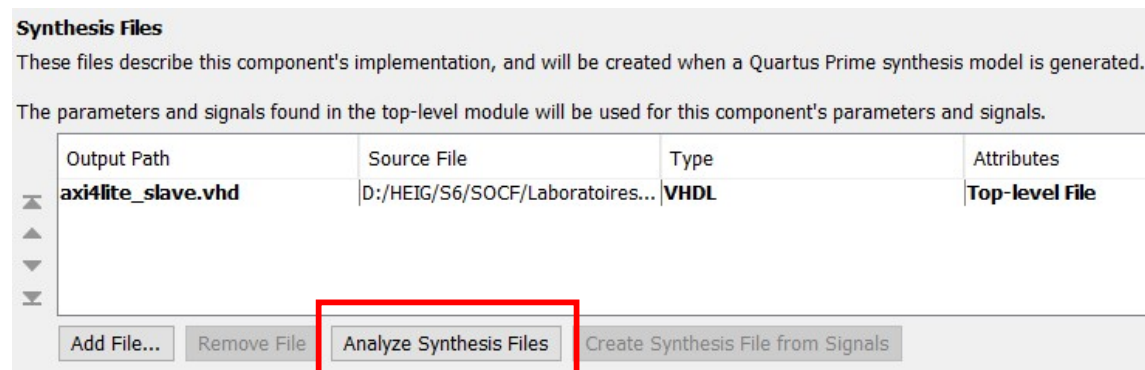


Figure 0-8 : Bouton oublié lors de la création de l'IP

Grâce à l'aide de l'assistant M. Masle, j'ai pu résoudre ce problème.

Voici à quoi doit ressembler les signaux et les interfaces :

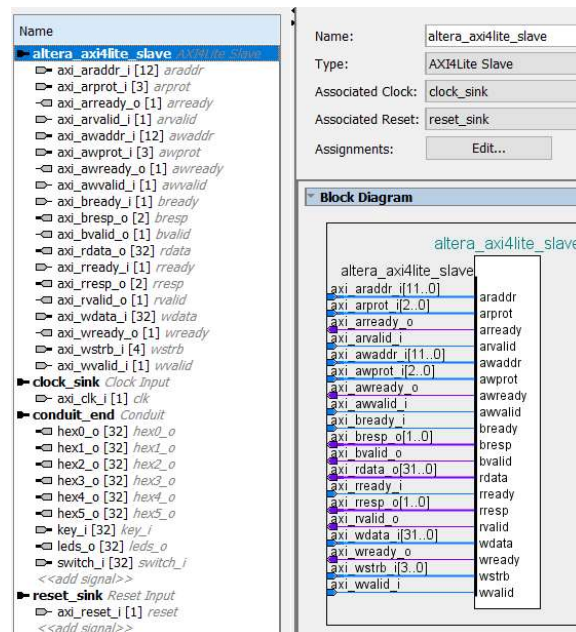


Figure 0-9 : Signaux et interfaces

Après avoir ajouté les composants AXI4Lite Slave, Clock Input, Reset Input et Conduit, j'ai pu glisser les signaux dans les interfaces correspondante. Il a aussi fallu lier la clock et le reset à l'interface AXI4Lite slave.

Ajout du composant

Après avoir créer mon nouveau composant, je l'ai ajouté dans le système Qsys. Ensuite, j'ai effectué les connexions ainsi que les exports de memory, hps_io et du conduit de mon IP. Finalement, j'ai ajouté l'adressage du composant. Voici à quoi cela doit ressembler :

Use	Connections	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>		clk_0 clk_in clk_in_reset clk clk_reset	Clock Source Clock Input Reset Input Clock Output Reset Output	clk <i>Double-click to export</i> <i>Double-click to export</i> <i>Double-click to export</i>	exported clk_0		
<input checked="" type="checkbox"/>		hps_0 memory hps_io h2f_reset h2f_lw_axi_clock h2f_lw_axi_master	Arria V/Cyclone V Hard Process... Conduit Conduit Reset Output Clock Input AXI Master	memory hps_io <i>Double-click to export</i> <i>Double-click to export</i> <i>Double-click to export</i>			
<input checked="" type="checkbox"/>		AXI4_lite_perso_0 altera_axi4lite_slave clock_sink reset_sink conduit_end	AXI Spinelli AXI4Lite Slave Clock Input Reset Input Conduit	<i>Double-click to export</i> <i>Double-click to export</i> <i>Double-click to export</i> conduit_export	clk_0 [clock_sink] [clock_sink] [clock_sink]	0x0000_0000	0x0000_0fff

Figure 0-10 : Système Qsys

Après avoir fait tout cela, j'ai pu générer les fichiers HDL du projet Qsys.

Remarque : La génération des fichiers HDL doit être refaite à chaque modification de l'IP.

Modification du top

Grâce au menu Generate -> Show Instantiation Template dans Qsys, j'ai pu apporter les modifications nécessaires au top du projet dans le fichier DE1_SoC_top.vhd. J'ai donc ajouté les nouveaux signaux « conduit » dans le composant « qsys_system ». Ensuite, j'ai mappé les conduits avec les I/Os de la FPGA comme ci-dessous :

conduit_export_switch_i (9 downto 0)	=> SW_i ,	-- switch_i
conduit_export_switch_i (31 downto 10)	=> (others => '0'),	
conduit_export_key_i (3 downto 0)	=> KEY_i ,	-- key_i
conduit_export_key_i (31 downto 4)	=> (others => '0'),	
conduit_export_leds_o (9 downto 0)	=> LEDR_o ,	-- leds_o
conduit_export_hex0_o (6 downto 0)	=> HEX0_o ,	-- hex0_o
conduit_export_hex1_o (6 downto 0)	=> HEX1_o ,	-- hex1_o
conduit_export_hex2_o (6 downto 0)	=> HEX2_o ,	-- hex2_o
conduit_export_hex3_o (6 downto 0)	=> HEX3_o ,	-- hex3_o
conduit_export_hex4_o (6 downto 0)	=> HEX4_o ,	-- hex4_o
conduit_export_hex5_o (6 downto 0)	=> HEX5_o ,	-- hex5_o

Figure 0-11 : Mapping du Top

On peut voir ici que j'ai décidé de créer une sortie pour chaque afficheur 7 segments. Il aurait été possible de combiner les afficheurs 0 à 3 et 4 à 5. Cependant, j'ai préféré avoir accès à chaque afficheur indépendamment. De plus, il a fallu mettre à 0 tous les bits non utilisés des entrées keys et switch.

Maintenant que tout est prêt, j'ai pu synthétiser et faire le placement routage du projet.

Validation pratique

Avant de me lancer dans les spécifications, je voulais m'assurer du bon fonctionnement réel de mon interface grâce à un test pratique. J'ai donc écrit quelques lignes de code C afin de tester que l'écriture ainsi que la lecture se déroulent correctement. Voici le code que j'ai testé :

```

93
94  AXI_HEX5 = 0x40;
95  AXI_HEX4 = 0xF9;
96  AXI_HEX3 = 0x24;
97  AXI_HEX2 = 0x30;
98  AXI_HEX1 = 0x19;
99  AXI_HEX0 = 0x12;
100
101  AXI_LEDS = AXI_SWITCHES;
102

```

Figure 0-12 : Code de test

Ce code affiche de 0 à 5 sur les afficheurs 7 segments dans gauche à droite et copie les valeurs des switches sur les leds. Une fois le projet lancé, voici ce que j'ai pu voir sur ma carte DE1-SoC :

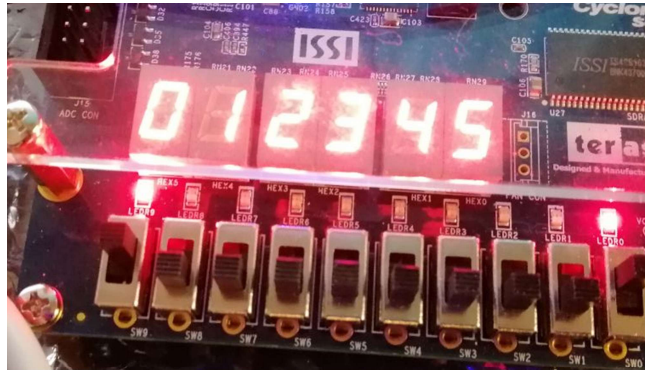


Figure 0-13 : Test sur la DE1

Grâce à ce test, j'ai pu m'assurer que mon interface fonctionne aussi dans la pratique.

Réalisation de la spécification

Maintenant que tout est prêt, j'ai pu réaliser la spécification de la partie 1. J'ai commencé par créer un projet Altera Monitor Program. Ensuite, j'ai repris les fichiers C du laboratoire précédent et j'ai adapté le code afin de répondre aux spécifications demandées dans ce laboratoire.

Ceci était facile car le 90% de la spécification est la même que le labo précédent.

Vous pouvez voir le code final de la partie 1 dans le répertoire

« `axi4lite/axi4lite/soft/src/lab05_partie1.c` ».

Remarque : J'ai perdu énormément de temps car j'avais d'étranges problèmes pour charger mon code C sur la carte DE1-SoC. Heureusement, M. Masle a mis à disposition sur switch tous les softs nécessaires pour Windows. J'ai donc pu télécharger, dézipper et faire les installations des différents programmes sur ma machine native. Grâce à cela, j'ai gagné beaucoup de temps pour chaque action de plus il était maintenant possible de programmer la DE1-SoC.

Deuxième partie : avec interruption

Pour la deuxième partie, il est demandé de gérer l'appui sur les boutons KEY 2 et 3 à l'aide d'interruption vers le HPS. Le design doit générer une interruption lors d'une détection d'un flanc d'un des 4 boutons. Il doit être possible de masquer/démasquer l'interruption pour chaque bouton.

Plan d'adressage

Afin de répondre à la deuxième partie, j'ai complété mon plan d'adressage afin de gérer les interruptions.

N	Offset	D32	Read	0	D32	Write	0	I/O
0	0x0000 0000		Constante (0xDEADBEEF)			not used		
1	0x0000 0004		[31..0] regTest			[31..0] regTest		Test
2	0x0000 0008		Reserved			Reserved		
3	0x0000 000C		Reserved			Reserved		
4	0x0000 0010		Reserved			Reserved		
5	0x0000 0014		Reserved			Reserved		
...	...		Reserved			Reserved		
64	0x0000 0100		[31..10] '0..0' - [9..0] dataLEDs (9..0)			[31..10] reserved - [9..0] dataLEDs (9..0)		Leds
...	...		Reserved			Reserved		
128	0x0000 0200		[31..4] '0..0' - [3..0] dataKeys (3..0)			not used		
129	0x0000 0204		[31..4] '0..0' - [3..0] sourceIRQ (3..0)			not used		Keys
130	0x0000 0208		[31..4] '0..0' - [3..0] maskIRQ (3..0)			[31..4] reserved - [3..0] maskIRQ (3..0)		
...	...		Reserved			Reserved		
192	0x0000 0300		[31..10] '0..0' - [9..0] dataSwitchs (9..0)			not used		Switchs
...	...		Reserved			Reserved		
256	0x0000 0400		[31..7] '0..0' - [6..0] dataHEX0 (6..0)			[31..7] reserved - [6..0] dataHEX0 (6..0)		7seg (Le point n'est pas connecté)
260	0x0000 0410		[31..7] '0..0' - [6..0] dataHEX1 (6..0)			[31..7] reserved - [6..0] dataHEX1 (6..0)		
264	0x0000 0420		[31..7] '0..0' - [6..0] dataHEX2 (6..0)			[31..7] reserved - [6..0] dataHEX2 (6..0)		
268	0x0000 0430		[31..7] '0..0' - [6..0] dataHEX3 (6..0)			[31..7] reserved - [6..0] dataHEX3 (6..0)		
272	0x0000 0440		[31..7] '0..0' - [6..0] dataHEX4 (6..0)			[31..7] reserved - [6..0] dataHEX4 (6..0)		
276	0x0000 0450		[31..7] '0..0' - [6..0] dataHEX5 (6..0)			[31..7] reserved - [6..0] dataHEX5 (6..0)		
...	...		Reserved			Reserved		
1023	0x0000 0FFF		Reserved			Reserved		

Figure 0-1 : Plan d'adressage (Partie 2)

Mon plan d'adressage est resté globalement identique mais j'ai rajouté 2 I/Os. Pour commencer, à l'offset 0x204, j'ai ajouté un champ afin de lire la source d'interruption. Chaque bit correspond à chaque bouton. Par exemple, Si le bit 0 du champs « sourceIRQ » est à 1, cela signifie qu'il y a eu une interruption sur la KEY0. J'ai décidé de faire un acquittement lors de la lecture de la source comme ça cela est fait automatiquement.

Le deuxième champ est « maskIRQ » qui est accessible en lecture et écriture. Il permet, comme son nom l'indique, de masquer ou pas une interruption. Par défaut, les 4 bits sont à '0' ce qui signifie que les quatre interruptions sont actives (non masquée).

Conception

Afin de gérer les interruptions, j'ai commencé par ajouté une sortie à mon interface qui sera directement connecté sur une ligne d'interruption du HPS.

```
-- Interruption
irq_o      : out std_logic
```

Figure 0-2 : Déclaration de la sortie irq

Ensuite, j'ai ajouté quelques nouveaux signaux afin de gérer les interruptions :

```
----- SIGNAUX GESTION IRQ -----
signal irq_s      : std_logic;
signal irq_source  : std_logic_vector(3 downto 0) := (others => '0');
signal key_val_save : std_logic_vector(3 downto 0) := (others => '1');
-- par défaut, toutes les irq actives
signal key_irq_mask : std_logic_vector(3 downto 0) := (others => '0');
```

Figure 0-3 : Signaux pour la gestion des interruptions

- Le signal « irq_s » est simplement le signal lié à la sortie irq_o.
- Le signal « irq_source » représente le champ « sourceIRQ » dans mon plan d'adressage. Il permet d'indiquer la source de l'interruption. Par défaut, l'état des bits est à '0', signifiant qu'il n'y a pas eu d'interruption.
- Le signal « key_val_save » permet d'enregistrer la valeur des KEYS afin de pouvoir le comparer avec la valeur réelle pour détecter un flanc. Par défaut, l'état des bits est à '1', car les boutons sont actifs bas.
- Le signal « key_irq_mask » représente le champ « maskIRQ » dans mon plan d'adressage. Il permet de gérer le masquage/démasquage de l'interruption de chaque bouton.

Afin de gérer les interruptions, je suis vite parti sur une solution de créer un process et d'utiliser la fonction « rising_edge » sur chaque bit des entrées « key_i ». Malheureusement, ce n'était pas aussi facile. En effet, il m'était impossible d'utiliser la fonction « rising_edge » sur l'entrée « key_i ». De plus, il est impossible de changer l'état d'un signal dans deux process différents. Étant donné que je devais gérer l'acquittement lors d'une lecture, il était plus simple de tout faire dans le process de lecture. Cependant, il aurait été possible de faire un signal de synchronisation entre les deux process. Voici mon process de lecture de données dans lequel j'ai ajouté la gestion des interruptions :

```

-- Read data channel
-- Implement axi_wready generation
process (reset_s, axi_clk_i)
--number address to access 32 or 64 bits data
variable int_raddr_v : natural;
begin

    if reset_s = '1' then
        --axi_waddr_done_s <= '0';
        axi_rvalid_s <= '0';
        axi_rdata_mem_s <= (others => '0');
        axi_rresp_s <= "00";

        1 irq_source <= "0000";
        irq_s <= '0';

    elsif rising_edge(axi_clk_i) then
        -- Gestion des interruptions
        2 if (key_val_save(0) /= registre_key_mem(0) and registre_key_mem(0) = '0' and key_irq_mask(0) = '0') then
            irq_source(0) <= '1';
            irq_s <= '1';
        elsif (key_val_save(1) /= registre_key_mem(1) and registre_key_mem(1) = '0' and key_irq_mask(1) = '0') then
            irq_source(1) <= '1';
            irq_s <= '1';
        elsif (key_val_save(2) /= registre_key_mem(2) and registre_key_mem(2) = '0' and key_irq_mask(2) = '0') then
            irq_source(2) <= '1';
            irq_s <= '1';
        elsif (key_val_save(3) /= registre_key_mem(3) and registre_key_mem(3) = '0' and key_irq_mask(3) = '0') then
            irq_source(3) <= '1';
            irq_s <= '1';
        end if;
        3 -- Met à jour l'ancienne valeur des keys
        key_val_save <= registre_key_mem;
    end if;
end process;

```

Figure 0-4 : Code pour la gestion des interruptions

1. Remise à '0' des signaux en cas de reset
2. Détection de flanc et test du masque. Si oui, mise à '1' de la source et de l'interruption.
3. Mise à jour des valeurs des boutons dans le signal de sauvegarde.

J'ai décidé de faire une détection sur flanc descendant car les boutons sont actifs bas.

Comme l'indique le plan d'adressage, j'ai ajouté le signal « irq_source » en lecture et « key_irq_mask » en lecture et écriture.

Remarque : Le code complet est en annexe.

Test de l'IP

Étant donné que j'ai trouvé cette partie relativement simple et que le code C du laboratoire précédent permet déjà de tester si une interruption est générée, je n'ai pas voulu perdre du temps à modifier le test Bench afin de tester la fonctionnalité d'interruption.

Mise à jour dans Qsys

Il est maintenant nécessaire de modifier mon composant dans Qsys afin d'ajouter une ligne d'interruption. Pour ce faire, j'ai ajouté une interface « Interrupt Sender » à mon composant en y ajoutant le signal de sortie « irq_o » correspondant :

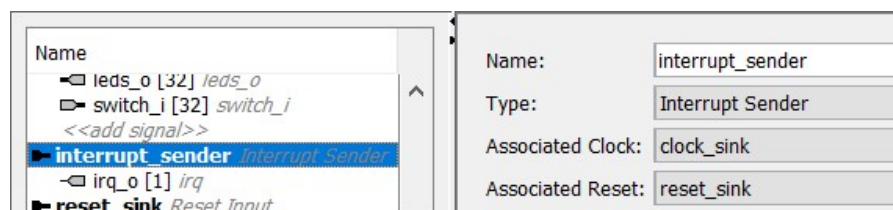


Figure 0-5 : Interface de l'interruption

Ensuite, comme pour le laboratoire précédent, j'ai activé les interruptions FPGA to HPS dans le composant HPS. Puis, j'ai connecté la ligne d'interruption sur le composant HPS sur la même ligne que le laboratoire précédent afin de garder le même numéro d'interruption (72).



Figure 0-6 : Connexion dans Qsys

Postérieurement, j'ai pu générer les fichiers HDL. Et finalement, synthétiser et faire le placement routage du projet.

Test de l'IP avec le code C

Maintenant que tout est prêt, j'ai pu reprendre les code C afin d'activer les interruptions du laboratoire précédent. Ensuite, afin de m'assurer que cela fonctionne, j'ai mis du code C qui affiche sur des afficheurs 7 segments des informations dans la routine d'interruption :

```
void pushbutton_ISR(void) {
    static int i = 0;
    int src_irq = AXI_INT_SRC;

    AXI_HEX0 = src_irq;
    AXI_HEX1 = i++;
}
```

Figure 0-7 : Code de test des interruptions

J'ai facilement pu constater grâce aux afficheurs 7 segments que les interruptions étaient bien générées et acquittées.

Réalisation de la spécification

Maintenant que j'ai testé le bon fonctionnement de mon interface, j'ai commencé par réaliser la spécification de la partie 2 du code qui consiste à utiliser une interruption pour les actions sur les boutons KEY2 et KEY3.

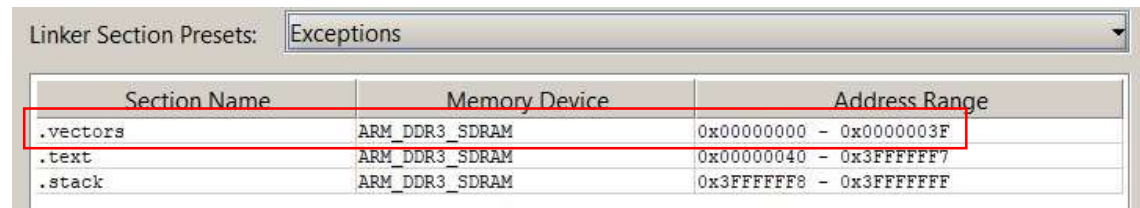
Compléter le code C

Cette étape était simple car le changement dans le code nécessitait de reproduire le code du laboratoire précédent afin de traiter des interruptions. Cependant, j'ai tout de même apporté une modification. Tous les traitements sont faits dans le « main » et non plus dans la routine d'interruption qui ne devrait contenir uniquement le strict minimum de traitement.

Remarque : Le code complet du laboratoire est en annexe.

Modifier la configuration mémoire

Afin d'allouer une portion mémoire pour les vecteurs d'interruptions, il a fallu modifier la configuration mémoire du projet « Altera Monitor Program » :



Section Name	Memory Device	Address Range
.vectors	ARM_DDR3_SDRAM	0x00000000 - 0x0000003F
.text	ARM_DDR3_SDRAM	0x00000040 - 0x3FFFFFF7
.stack	ARM_DDR3_SDRAM	0x3FFFFFF8 - 0x3FFFFFFF

Figure 0-8 : Modification de la configuration mémoire

On peut voir qu'une section a été ajoutée pour les vecteurs.

Compiler et tester

Après avoir compiler et charger mon programme dans la DE1-SoC, j'ai pu tester le bon fonctionnement des spécifications demandées.

Test des spécifications

Pour ce faire, j'ai appuyé plusieurs fois sur KEY0 avec des valeurs de switch différents afin de m'assurer que les leds aient toujours la même valeur des switches après l'appui. J'ai aussi vérifié que les afficheurs HEX5 à HEX0 affichent en hexadécimal les bits 23 à 0 de la constante définie dans l'IP.

Ensuite, j'ai fait les mêmes tests pour KEY1. Je me suis assuré que l'états inverses des switches est copiés sur les LEDs et que les afficheurs HEX5 à HEX0 affichent en hexadécimal l'inverse des bits 23 à 0 de la constante définie dans l'IP.

J'ai aussi testé le bon fonctionnement de ces deux boutons en les testant consécutivement l'un après l'autre.

Ultérieurement, j'ai testé KEY2 plusieurs fois et l'effet sur les bords. L'affichage des LEDs et des afficheurs 7 segments ont bien subi une rotation à droite. Rotation d'un bit pour les LEDs, rotation d'un afficheur complet pour les afficheurs 7 segments.

Finalement, j'ai testé KEY3 plusieurs fois et l'effet sur les bords. L'affichage des LEDs et des afficheurs 7 segments ont bien subi une rotation à gauche. Rotation d'un bit pour les LEDs, rotation d'un afficheur complet pour les afficheurs 7 segments.

Test du masquage

Afin de tester le masquage des interruptions, j'ai initialisé la valeur du masque à 0x08 afin de masquer le bouton KEY3. Puis, j'ai ajouté ce code dans la routine d'interruption :

```
// Tous les 3 interruptions de KEY0 et KEY1, change le masque de key 2 et 3
if (src_irq & KEY0 || src_irq & KEY1) {
    cpt_int++;

    if (cpt_int % 3 == 0)
        AXI_INT_MASK = AXI_INT_MASK ^ (KEY3 | KEY2);
}
```

Figure 0-9 : Code de test du masquage

Il permet d'intervertir le masque de KEY2 et KEY3 après 3 interruptions sur KEY0 ou KEY1. Donc, au démarrage l'appui sur la KEY3 n'avait aucun effet contrairement à KEY2. Après 3 appuis sur KEY1/KEY0, c'était le contraire comme attendu. L'appui sur KEY2 n'avait aucun effet contrairement à KEY3.

Fonctionnalité de strobe

Cette fonctionnalité permet de choisir quelle partie des bits du bus AXI_WDATA vont être pris en compte. Voici un schéma dans la documentation qui l'explique bien :

S_AXI_WSTRB signals		
S_AXI_WSTRB [3:0]	S_AXI_WDATA active bits [31:0]	Description
1111	11111111111111111111111111111111	All bits active
0011	00000000000000001111111111111111	Least significant 16 bits active
0001	00000000000000000000000011111111	Least significant byte (8 bits) active.
1100	11111111111111111000000000000000	Most significant 16 bits active

Figure 0-10 : Tableau du la fonction strobe

Je pensais ne pas avoir assez de temps pour réaliser ce laboratoire donc j'ai décidé par ne pas gérer cette fonctionnalité au début. Étant donné que du temps supplémentaire nous a été donné, j'en ai profité pour réaliser cette fonctionnalité.

J'ai commencé par modifier le test Bench afin de tester ce paramètre :

```
constant TAB_STI_AXI_WRITE : Type_Tab_Stimuli_AXI_WRITE :=
--REM: table fixe pour 32 bits de data
(
    ( 4, x"01234567", "1111"),    --écriture adresse 0x04
    ( 8, x"01234567", "0111"),    --écriture adresse 0x08
    (12, x"01234567", "0011"),    --écriture adresse 0x0C
    (16, x"01234567", "0001"),    --écriture adresse 0x10
    ( 4, x"01234567", "1100")    --écriture adresse 0x04
);
```

Figure 0-11 : Modification du test Bench

Grâce aux nouvelles valeurs de ce tableau d'écriture, il sera facile de voir le bon fonctionnement du paramètre strobe.

J'ai relancé le nouveau test bench pour voir que la fonctionnalité n'est pas réalisée :



Figure 0-12 : Test 1 du paramètre strobe

On peut voir que la donnée à écrire est bien 0x01234567 et que le paramètre strobe est à « 0011 ». On peut donc s'attendre à que les deux octets de poids fort de la donnée à écrire seront pas actif. Ce qui donnerait 0x00004567. Cependant, les données enregistrées restent 0x01234567 dans le signal « axi_wdata_mem_s ». Le but maintenant est de réaliser cette fonctionnalité.

Description VHDL

Afin de prendre en compte le paramètre strobe, j'ai modifié le process qui s'occupe du canal des données d'écriture afin d'enregistrer uniquement les octets souhaitée par le paramètre strobe. Voici à quoi ressemble le code :

```

1 axi_wdata_mem_s <= (others => '0');

2 if (axi_wstrb_i(0) = '1') then
    axi_wdata_mem_s(7 downto 0) <= axi_wdata_i(7 downto 0);
end if;
if (axi_wstrb_i(1) = '1') then
    axi_wdata_mem_s(15 downto 8) <= axi_wdata_i(15 downto 8);
end if;
if (axi_wstrb_i(2) = '1') then
    axi_wdata_mem_s(23 downto 16) <= axi_wdata_i(23 downto 16);
end if;
if (axi_wstrb_i(3) = '1') then
    axi_wdata_mem_s(31 downto 24) <= axi_wdata_i(31 downto 24);
end if;

```

Figure 0-13 : Code gestion de strobe

1. J'ai commencé par mettre tous les bits à '0' comme si aucun octet était actif.
2. Ensuite, j'ai testé chaque bit du paramètre strobe afin d'assigner strictement les octets actifs.

Test de la fonctionnalité strobe (test bench)

Maintenant que le test bench est déjà prêt, il suffit de compiler le nouveau code de mon IP et de lancer le test bench.

Remarque : Afin de voir plus facilement les valeurs dans le signal « axi_wdata_mem_s » j'ai commenté se remise à 0 à chaque fin de lecture des données.



Figure 0-14 : Chronogramme de test de strobe

Comme indiqué par les flèches orange, les données d'écriture enregistrées correspondent bien aux paramètre strobe (S_AXI_WSTRB). Par exemple, pour la troisième flèche orange, le paramètre strobe vaut « 0011 » et on peut voir comme attendu que les données enregistrées sont bien « 0x00004567 ». On peut aussi bien voir dans le dernier cas, le paramètre vaut « 1100 », donc on souhaite activer seulement les 2 octets de poids fort. Comme attendu, les données enregistrées sont bien « 0x01230000 ».

Test de la fonctionnalité strobe (code)

Afin de m'assurer du bon fonctionnement de l'implémentation du paramètre strobe, j'ai voulu le tester en situation réel. J'ai donc écrit du code pour tester ce paramètre. Puis grâce au débogueur j'ai avancé pas à pas dans le code en assembleur afin de m'assurer du bon fonctionnement.

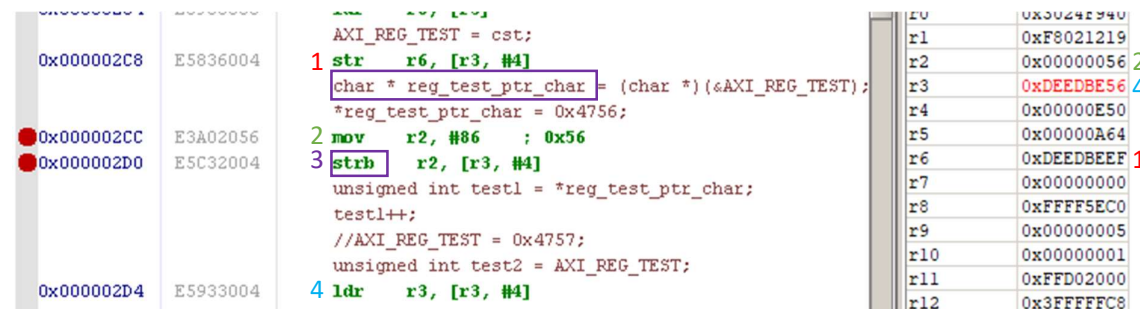


Figure 0-15 : Débogage pour tester le paramètre strobe

On début du programme, on lit l'adresse de la constante (0xDEADBEEF) et la place dans le registre de test qui est à l'offset 0x4. A la première instruction, le registre r3 comprend l'adresse de base (0xFF200000) donc l'adresse de la constante. De ce fait, « r3, #4 » indique l'adresse du registre de test.

1. **Str** : la première instruction permet d'écrire la valeur du registre r6 (qui contient la valeur de la constante) dans le registre de test à l'offset 0x4.
2. **Mov** : Place la valeur de 8 bits (0x56) dans le registre r2.
3. **Strb** : Cette instruction permet de placer un byte de la valeur du registre r2 (0x56) à l'adresse du registre de test. Un pointeur de char est utilisé afin de forcer le programme à utiliser l'instruction « strb » pour s'assurer qu'une écriture d'uniquement 8 bits se fait et pas 32 bits. Ce qui permet de tester la fonctionnalité strobe implémenté.
4. **Ldr** : Cette instruction permet de lire 32 bits à l'adresse du registre de test et la place dans le registre r3. On peut voir que la valeur vaut 0xDEADBEE5. Ce qui confirme que mon IP n'est pas écrasé 0xDEADBE par 0x000000 grâce à l'implémentation du strobe.

Finalement, j'ai pu confirmer que mon implémentation du paramètre strobe dans mon IP est bien fonctionnel même sur le matériel.

Supplémentaire : Gestion Edge

Le paramètre Edge permet de choisir sur quel flanc l'interruption sera levée. Par défaut, j'ai fixé ce paramètre au flanc descendant afin d'avoir une interruption dès qu'un bouton est pressé, car ils sont actifs bas.

Plan d'adressage

Pour gérer ce nouveau paramètre, j'ai ajouté un champ dans mon plan d'adressage. Voici la modification que j'ai effectuée :

128	0x0000 0200	[31..4] '0..0' - [3..0] dataKeys (3..0)	not used	Keys
129	0x0000 0204	[31..4] '0..0' - [3..0] sourceIRQ (3..0)	not used	
130	0x0000 0208	[31..4] '0..0' - [3..0] maskIRQ (3..0)	[31..4] reserved - [3..0] maskIRQ (3..0)	
131	0x0000 020C	[31..4] '0..0' - [3..0] edgeIRQ (3..0)	[31..4] reserved - [3..0] edgeIRQ (3..0)	

Figure 0-1 : Modification du plan d'adressage

Toute la tables reste identiques mise à part le nouveau champ « edgeIRQ » à l'offset 0x20C (131). Ce paramètre est évidemment accessible en lecture ainsi qu'en écriture. Il permettra de choisir sur quel flanc générée l'interruption.

Description VHDL

Afin de réaliser cette nouvelle fonctionnalité, j'ai ajouté un signal « key_irq_edge » de 4 bits initialisé à 0 afin d'activer l'interruption sur un flanc descendant par défaut.

```
-- par défaut, toutes les irq sur flanc descendant
signal key_irq_edge      : std_logic_vector(3 downto 0) := (others => '0');
```

Figure 0-2 : Déclaration du signal edge

Ensuite, dans les « switch » des canaux de lecture et d'écriture, j'ai ajouté le « case » pour ce nouveau champ afin d'y accéder en lecture et en écriture.

Finalement, j'ai modifié la gestion des interruptions afin de détecter une interruption sur un flanc en fonction du signal « key_irq_edge » :

```
elsif rising_edge(axi_clk_i) then
-- Gestion des interruptions
if (key_val_save(0) /= registre_key_mem(0) and registre_key_mem(0) = key_irq_edge(0) and key_irq_mask(0) = '0') then
    irq_source(0) <= '1';
    irq_s <= '1';
elsif (key_val_save(1) /= registre_key_mem(1) and registre_key_mem(1) = key_irq_edge(1) and key_irq_mask(1) = '0') then
    irq_source(1) <= '1';
    irq_s <= '1';
elsif (key_val_save(2) /= registre_key_mem(2) and registre_key_mem(2) = key_irq_edge(2) and key_irq_mask(2) = '0') then
    irq_source(2) <= '1';
    irq_s <= '1';
elsif (key_val_save(3) /= registre_key_mem(3) and registre_key_mem(3) = key_irq_edge(3) and key_irq_mask(3) = '0') then
    irq_source(3) <= '1';
    irq_s <= '1';
end if;
-- Met à jour l'ancienne valeur des keys
key_val_save <= registre_key_mem;
```

Figure 0-3 : Gestion des interruptions avec le paramètre edge

Avant, je testais l'égalité des valeurs des boutons « registre_key_mem » avec la constante '0' afin de détecter un flanc descendant. Maintenant, je test l'égalité avec le paramètre edge des boutons « key_irq_edge ».

Mise à jour du projet

Une fois toutes ces modifications apportées, j'ai compilé ma nouvelle description VHDL de mon IP. Ensuite j'ai mise à jour mon projet en mettant à jour mon composant dans Qsys, puis, j'ai généré les fichiers HDL, ensuite, j'ai synthétisé et fait le placement routage du projet.

Test de fonctionnalité

Afin de tester cette nouvelle fonctionnalité, j'ai modifié le code C. J'ai commencé par ajouter une définition qui permet de lire et écrire facilement dans le champ Edge des keys :

```
// 0 = interruption flanc descendant (défaut)
#define AXI_INT_EDGE          *(vuint *) (AXI_LIGHT_BASE_ADDR + 0x20C)
```

Figure 0-4 : Définition du paramètre edge

Ensuite, j'ai modifié la valeur par défaut de ce champ afin de tester si un flanc montant était bien géré. J'ai donc mis à 1 le bit correspondant à la KEY3 afin de lever une interruption uniquement au flanc montant de la KEY3 :

```
// KEY 3 sur flanc montant
AXI_INT_EDGE = KEY3;
```

Figure 0-5 : Initialisation du paramètre edge

Après ces modifications, j'ai pu tester cette nouvelle fonctionnalité directement sur le matériel. J'ai commencé par télécharger le nouveau système sur la carte DE1 grâce au programme « Altera Monitor Program ». Ensuite, j'ai compilé et loadé le nouveau programme C. Finalement, en lançant le nouveau programme, j'ai pu constater que la rotation à gauche était bien effectuée seulement lors du relâchement du bouton KEY3.

J'ai décidé de laisser cette modification au programme final rendu car il respecte toujours les spécifications demandées et permet de prouver, sans changement de code, cette fonctionnalité.

Annexes

Voici la liste dans l'ordre des annexes :

1. Code VHDL de mon IP (axi4lite_slave.vhd)
2. Code VHDL du top (DE1_SoC_top.vhd)
3. Code du programme principal (labo5.c)
4. Définitions du code (defines.h)
5. Point H du fichier exception (exceptions.h)
6. Code de fonctions utiles (exceptions.c)
7. Définitions d'adresse (address_map_arm.h)

Conclusion

Je dois avouer qu'au début du laboratoire j'étais perdu et avais peur de tout le travail demandé. Finalement, sans prendre en compte les temps de compilations extrêmement long, j'ai beaucoup apprécié ce laboratoire. C'est pourquoi, c'est avec plaisir que j'ai ajouté la fonctionnalité edge sur les interruptions.

Difficultés rencontrées

- Le bon fonctionnement de tous les programmes fut difficile. En effet, une mise en place sur Windows fût nécessaire.
- La compréhension du fonctionnement complet du bus AXI 4 Lite.

Compétences acquises

- Installation complète de l'environnement sur Windows
- Perfectionnement de la méthodologie
- Perfectionnement des logiciels (Quartus Prime, Qsys, altera monitor program et Questasim)

Résultats obtenus

J'ai réussi à mettre en place toutes les étapes qui m'était demandé dans ce laboratoire. Les description VHDL sont synthétisable et intégrable. Je suis particulièrement fier d'avoir réussi à faire complètement le travail demandé. Je me dois remercier les professeurs d'avoir repoussé la date du rendu et surtout l'assistant M. Masle qui a passé environ 2 heures afin de m'aider en partie pour résoudre ces gros problèmes de logiciel.

Date : 08.05.20

Nom de l'étudiant : Spinelli Isaia