Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Domaine Sciences de l'ingénieur
Rte du Rawyl 47
CH- 1950 Sion 2
Tél. +41 27 606 85 11
Fax +41 27 606 85 75
info@hevs.ch

www.hevs.ch

Filière Systèmes industriels

Orientation Infotronics

Diplôme 2007

Samuel Valentini

Pont AMBA - Wishbone

Professeur	François Corthay
Expert	Jacques Tinembart

Sion, le 23 novembre 2007

HES-SO Valais

SI	TV	EE	IG	EST
X	X	X	X	

Données du travail de diplôme
Daten der Diplomarbeit

FO.0.2.02.07.DB
DD/18/05/2006

Filière / *Studiengang* : Systèmes industriels

Confidentiel / *Vertraulich* ☐

Etudiant / *Student* Samuel Valentini	**Année scolaire / *Schuljahr*** 2006/07	**No TD / *Nr. DA*** SI/2007/12
Proposé par / *vorgeschlagen von* HES-SO Valais, UIT		**Lieu d'exécution / *Ausführungsort*** HES-SO Valais, DSI **Expert / *Experte***

Titre / *Titel*:

Pont AMBA - Wishbone

Description / *Beschreibung*:

Le bus AMBA est largement utilisé par l'unité Infotronics dans la réalisation de System-on-Chip (SoC). Le bus Whisbone est un standard OpenCore <http://www.opencores.org/> pour lequel beaucoup de blocs sont en accès libre.

Le but de ce projet est de réaliser un pont (bridge) entre le bus AMBA et le bus Whisbone pour permettre d'intégrer des périphériques Whisbone à un système AMBA. Le travail de semestre a permis la réalisation d'un pont AMBA-APB <--> Wishbone.

Le travail de diplôme consiste en la réalisation d'un pont AMBA-AHB <--> Wishbone. Dans une première phase, le maître (ou les maîtres) du bus seront du côté AMBA. Ensuite, la possibilité de gérer des maîtres de chaque côté est à étudier.

Objectifs / *Ziele:*

- Réalisation d'un pont AMBA-AHB <--> Wishbone, avec maître du côté AMBA
- Périphérique rapide : étude de la possibilité de gérer des registres par l'APB et un flot de données via l'AHB
- Démonstrateur SPDIF
- Gestion d'un système multi-maîtres.

Signature ou visa / *Unterschrift oder Visum*	**Délais / *Termine***
Resp. de l'orientation infotronics	Attribution du thème / Ausgabe des Auftrags: 03.09.2007
Professeur/*Dozent*: François Corthay	Remise du rapport / Abgabe des Schlussberichts: 23.11.2007
Etudiant/*Student*:	Exposition publique / Ausstellung Diplomarbeiten: 30.11.2007
	Défenses orales / Mündliche Verfechtungen Semaine 49

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats:* .....

Valentini, Samuel 2007

Pont AMBA - Wishbone

Brücke AMBA - Wishbone

Objectifs

Le but de ce projet basé sur les FPGA est de réaliser un pont (bridge) entre le bus AMBA et le bus Wishbone pour permettre d'intégrer des périphériques Wishbone à un système AMBA.
Dans un but de démonstration, il est demandé qu'un émetteur SPDIF soit connecté au bus Wishbone et qu'il soit possible d'écouter un fichier musical.

Résultats

Le pont et la lecture du fichier musical fonctionnent correctement.
Un contrôleur Ethernet a même été rajouté dans l'optique de télécharger plus rapidement le fichier sur la mémoire.

Mots-clés

FPGA, AMBA, Wishbone, Gaisler, SPDIF, SDRAM, Ethernet

Ziel

Das Ziel des Projektes, welches auf einer FPGA aufbaut, ist eine Brücke (bridge) zwischen dem Bus AMBA und dem Bus Wishbone zu realisieren, um Wishbone Peripherie in ein System AMBA zu integrieren.
Als Demonstrations-Ziel soll ein Sender SPDIF an den Bus Wishbone angeschlossen werden und es soll möglich sein ein Stück Musik zu hören.

Resultate

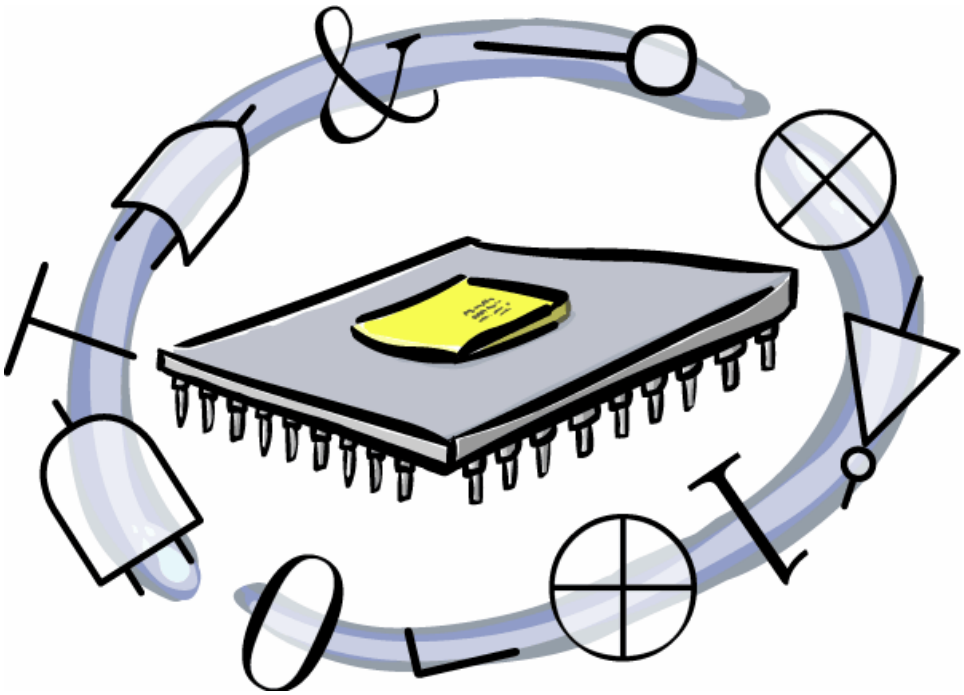
Die Brücke und das Lesen der Musikdatei funktionieren korrekt.
Es wurde sogar ein Ethernet-Kontroller hinzugefügt, um eine Datei schneller in den Speicher zu laden.

Schlüsselwörter

FPGA, AMBA, Wishbone, Gaisler, SPDIF, SDRAM, Ethernet

Travail de Diplôme

PONT AMBA – WISHBONE



Valentini Samuel
Sion, le 23.11.2007

Table des matières

1. Introduction

Après avoir terminé le cursus de formation d'ingénieur HES au sein de la HES-SO Valais, les étudiants sont amenés à accomplir un travail de diplôme. Ce travail individuel d'une durée de douze semaines invite l'étudiant à réaliser un projet parmi ceux présenté par des professeurs de l'école. Ce travail est déjà amorcé par le projet de semestre fait six mois auparavant.

1.1 FPGA

Les FPGA (*field-programmable gate array)* sont des puces électroniques contenant des millions de portes logiques (réseau logique) pouvant être programmées par un développeur de telle sorte à avoir une ou plusieurs fonctionnalités voulues.
Les FPGA sont utilisés dans diverses applications nécessitant de l'électronique numérique (télécommunications, aéronautique, transports par exemple). Les FPGA modernes sont assez vastes et contiennent suffisamment de mémoire pour être configurés afin d'héberger un cœur de processeur ou un DSP (*digital signal processor*), dans le but d'exécuter un logiciel. On parle dans ce cas de processeur « softcore », par opposition aux microprocesseurs « hardcore » enfouis dans le silicium.
Les FPGA peuvent donc contenir par exemple des processeurs, des mémoires, des convertisseurs, des contrôleurs, … Et ces fonctionnalités travaillent en parallèle les unes avec les autres.
Afin de pouvoir finaliser une FPGA, il est nécessaire d'utiliser un langage de description matériel (HDL) ou bien un outil de saisie graphique.

Un des langages de programmation utilisé fréquemment pour les FPGA est le VHDL (*VHSIC (Very-High Speed Integrated Circuits) Hardware Description Language*). Le VHDL a été développé à l'origine par le Département de la Défense des Etats-Unis pour documenter le comportement des ASICs (*Application-Specific Integrated Circuit*). En clair, c'est un langage de description matériel voué à décrire le comportement et l'architecture de systèmes électroniques numériques. L'une des principales particularités de ce langage est que les instructions se font en parallèle.
Les fichiers VHDL sont regroupés en librairies. Il est naturellement avantageux d'utiliser ces librairies dans le but de gagner du temps lorsqu'on doit faire la conception d'un projet contenant une FPGA.

Verilog est un langage similaire au VHDL, qui a été développé par la société Cadence Design Systems.

1.2 Situation

La plupart des fonctionnalités des FPGA communiquent entre elles aux moyens de bus. Chaque bloc est conçu pour un bus spécifique.
Les bus en relation avec ce projet sont le bus AMBA et le bus Wishbone. Les deux bus fonctionnent avec l'architecture **master/slave**.

Le bus AMBA est beaucoup utilisé dans l'unité infotronique de l'école grâce à aux nombreuses librairies de Gaisler Research qui lui sont associées. Ces librairies offrent de larges possibilités pour les systèmes à FPGA. Gaisler Research est une entreprise privée dont le fondateur est Jiri Gaisler. Elle fournit donc des « blocs » synthétisable pour les FPGA, mais son produit principal est le processeur LEON.
Le processeur LEON3 est un processeur dit « softcore ». Il utilise le bus AMBA pour communiquer avec ses slaves. Jusqu'ici, l'unité infotronique a utilisé le processeur LEON2.

Le bus Wishbone fait parti du monde libre et les composants qui lui sont associés sont donc entièrement gratuits. Il sera donc probablement de plus en plus utilisé dans les années à venir.

Deux cartes à FPGA sont à disposition pour l'implémentation physique du circuit. La première est une petite carte, mais contenant suffisamment de mémoire et de puissance pour faire tourner un Linux. Son nom est Suzaku.

Figure 1 : Carte Suzaku

La deuxième est une carte complète de test et contient quatre sortes de mémoires (Flash, EEPROM, RAM, SDRAM) et une FPGA avec plus de capacité.

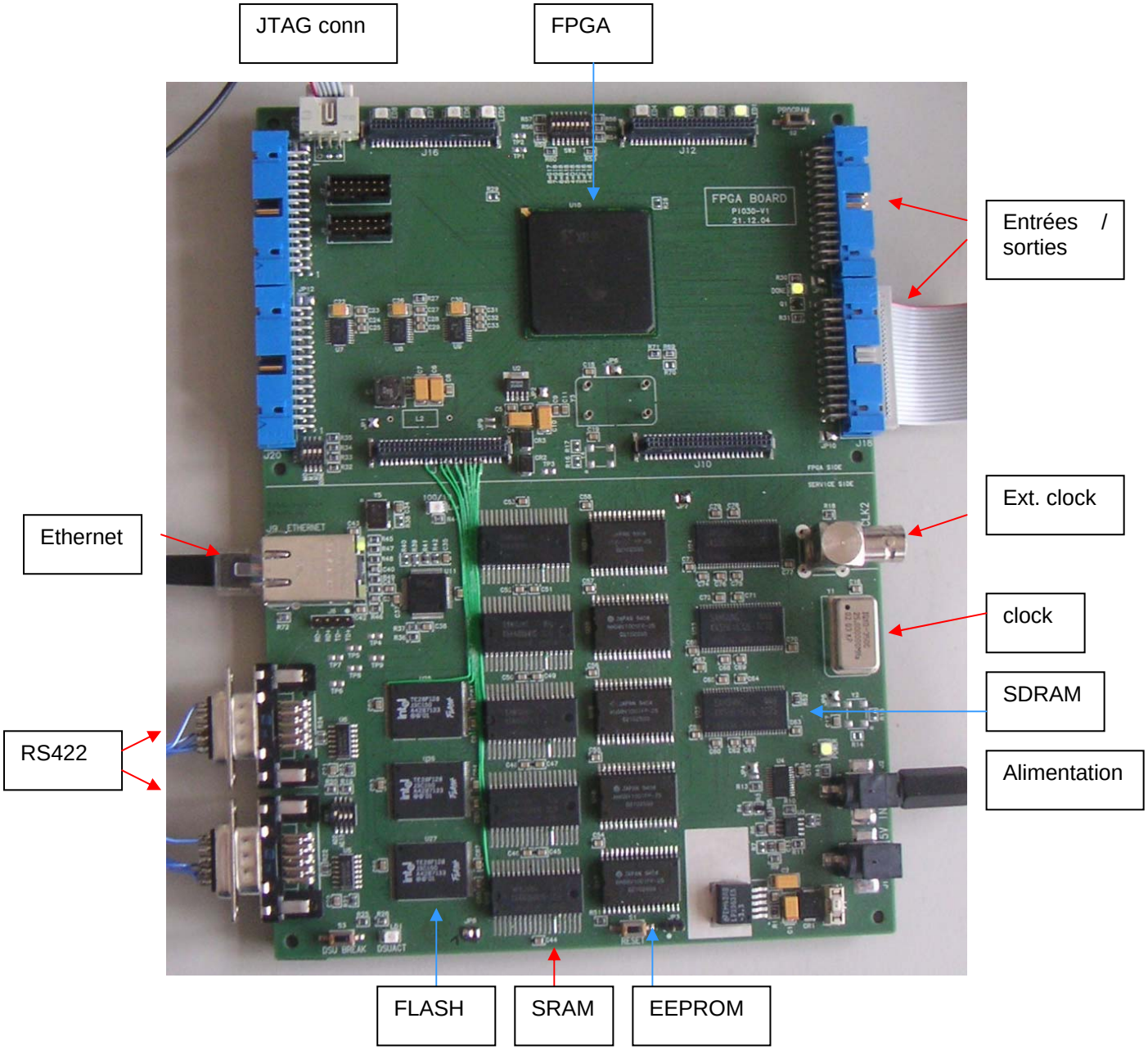


Figure 2 : Carte

Lors du projet de semestre, un pont AMBA-APB / Wishbone a été réalisé. La simulation du pont arrivait à lire et écrire sur un émetteur SPDIF et la synthèse du circuit a montré qu'il était possible de l'implémenter dans la première FPGA.

L'école fera une journée porte ouverte le 30 novembre 2007 avec comme principal intérêt les projets de diplôme des futurs ingénieurs.

1.3 Objectifs

A partir de la situation donnée, les objectifs du projet sont formulés de manière assez générale et non-exhaustive. Il est aussi évident que certains objectifs se révèlent en cours de route.

1.3.1 Objectifs de la donnée

- Réalisation d'un pont AMBA-AHB <--> Wishbone, avec maître du côté AMBA
- Périphérique rapide : étude de la possibilité de gérer des registres par l'APB et un flot de données via l'AHB
- Démonstrateur SPDIF
- Gestion d'un système multi-maître (annulé)

1.3.2 Objectifs découlant de la donnée

Une partie de ces objectifs ont été établis pendant le projet.

- Le pont doit gérer différentes vitesse de périphérique Wishbone
- Implémentation du circuit sur la carte choisie
- Etablir une communication série avec la carte
- Charger un programme capable d'écrire et lire un fichier musical sur une mémoire
- Ecrire le fichier sur un émetteur SPDIF de telle manière qu'il le joue
- Développement d'une carte d'adaptation SPDIF

1.4 Logiciels utilisés

Différents logiciels sont utilisés durant ce projet. Ceux-ci sont brièvement présentés ici et sont plus développé dans les chapitres qui les concernent.

Programmes liés aux FPGA :
- **HDL Designer** : Outils de conception graphique pour les FPGA.
- **ModelSim** : Programme de simulation
- **Synplify** : Outils de synthèse de code VHDL (transformation de code en portes logiques)
- **Xilinx ISE** : Outils de synthèse, mapping, placement et routage pour les FPGA du fabricant Xilinx

Programmes généraux :
- **Ultra Edit** : Editeur de texte, qui facilite la programmation pour de nombreux langages
- **Cygwin** : Shell qui permet l'utilisation de programme qui fonctionnent sous UNIX
- **SuPTerminal** : Terminal polyvalent

Programmes de Gaisler Research :

- **Grmon** : Programme qui permet la communication et le debug avec le processeur LEON
- **BCC** : Cross-compilateur pour les processeurs LEON2 et 3, il est basé sur un compilateur GNU

Programmes de développement de circuits imprimés P-CAD :

- **Schematic** : Outil de création de schémas électrique
- **PCB** : Outil de routage pour la création de circuits imprimés à partir du schéma du programme Schematic.

2. Etude du projet

2.1 Etude de faisabilité

2.1.1 Introduction

L'étude de faisabilité doit montrer qu'il est possible de réaliser les principaux objectifs du projet. Pour cela, il faut donc étudier les bus AMBA et Wishbone et révéler qu'il est admissible ou non de pouvoir rendre un périphérique Wishbone utilisable pour un processeur travaillant sur le bus AMBA.
Le but est donc de prendre une vue d'ensemble des différents thèmes du projet. La vue d'ensemble doit être suffisamment claire et approfondie pour se donner une idée du fonctionnement général, sans pour autant entrer trop dans les détails. Ceux-ci peuvent se trouver dans les documents officiels.

Comme la consigne du projet demande d'avoir un processeur LEON3 qui appartient aux librairies de Gaisler Research, le projet utilisera donc les outils mis à disposition par Gaisler Research. Ce qui veut dire que le pont sera entièrement compatible avec les librairies de Gaisler, mais cela ne garantit en aucun cas qu'il fonctionne pleinement avec un bus AMBA pur.

Figure 3 : logo Gaisler Research

L'utilisation des outils de Gaisler donne, entre autre, une capacité de « plug and play ». Ce document distingue en général ce qui est spécifique aux librairies Gaisler.

2.1.2 Le bus AMBA

Le bus AMBA (Advanced Microcontroller Bus Architecture) a été développé en 1995 par la société ARM pour ses propres processeurs et il est largement utilisé comme bus « on-chip ».
Il y a trois catégories de bus AMBA : AHB, ASB, APB.

Le bus AHB (Advanced High-performance Bus) est caractérisé par sa grande vitesse et ses hautes performances. Il est encore divisé en deux catégories, le bus des masters (AHB master) et celui des slaves (AHB slave). C'est le bus qui est relié aux masters et aux slaves rapides.

Le bus ASB (Advanced System Bus) est un bus qui peut remplacer le bus AHB, mais il est moins performant. Il est donc peu utilisé, et d'ailleurs il n'est pas utilisé dans ce projet.

Le bus APB (Advanced Peripheral Bus) est optimisé pour une consommation minimale de puissance, une simplicité de conception du slave et afin d'assurer le bon fonctionnement des périphériques. C'est le bus qui est relié aux slaves lents.

La figure suivante présente la vue général d'un circuit à bus AMBA :

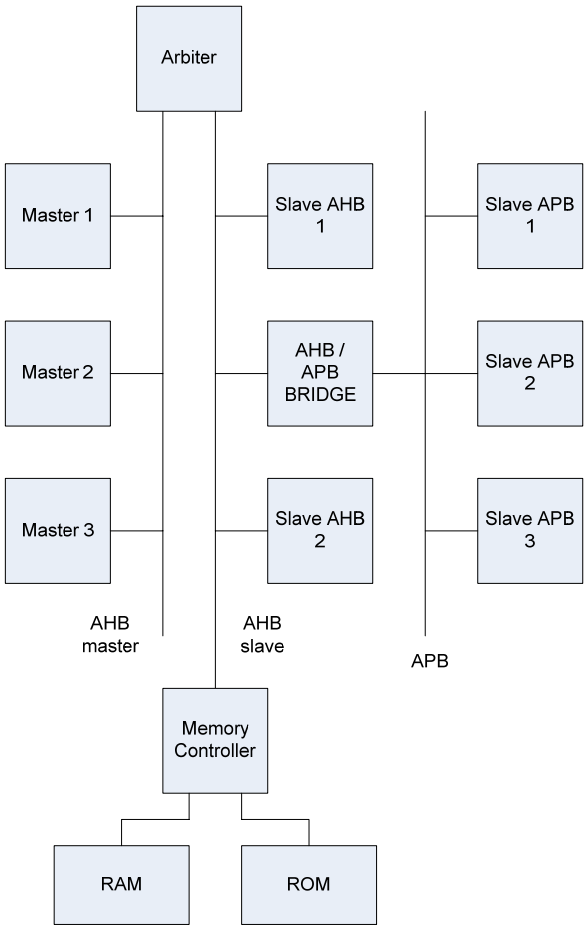


Figure 4 : Circuit général à bus AMBA

Les masters sont généralement des processeurs. Ils communiquent via l'arbitre en spécifiant l'adresse du slave avec lequel ils souhaitent échanger des données.
Chaque slave peut donner une réponse sur son propre bus qui retourne aux masters.
Comme son nom l'indique, l'arbitre gère les transferts et priorités des masters. De plus, il fait office de pont entre le bus AHB master et AHB slave.
Le contrôleur de mémoire s'occupe des mémoires RAM et ROM, mais il peut également se charger des IO.

2.1.2.1 Le bus AMBA – AHB

La figure suivante présente un exemple de circuit qui montre le fonctionnement des bus AHB master et AHB slave :

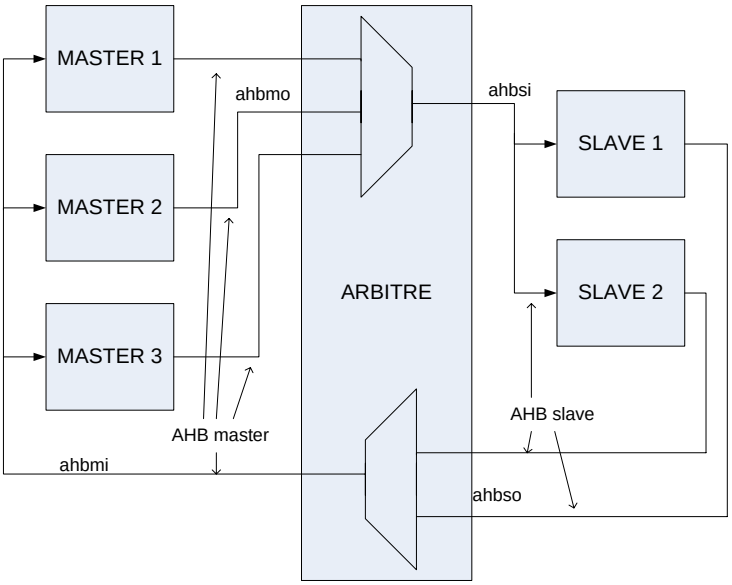


Figure 5 : Arbitre AMBA

Du côté master, l'arbitre a comme sortie un unique bus (contenant les données et bits de contrôles) qui sera l'entrée des masters (ahbmi) et a en entrée tous les bus de sortie (ahbmo) de chaque master (contenant chacun ses propres adresses, données, …). Le même principe se trouve côté slave, mais avec des signaux différents.
En général, le master fait une requête sur le bus ahbmo pour lire ou écrire sur un slave. L'arbitre s'occupe de l'envoyer au bon slave via le bus ahbsi. Le slave répond sur le ahbso en spécifiant le numéro du master. Finalement, le master reçoit la réponse grâce à l'arbitre via le bus ahbmi.

Il y a donc une partie « In » et « Out » pour chaque bus AHB.

Dans le cadre de ce projet, le bus AHB master n'est pas nécessaire d'être étudié en profondeur, par contre le bus AHB slave doit être bien détaillé.
Le tableau suivant présente les signaux du bus ahbsi :

Nom du signal	Nbr de bits	gaisler	Fonction
haddr	32	non	adresse du transfert en cour
hwdata	32	non	données pour une opération d'écriture
hsel	16	oui	sélection du slave (un bit par slave)
hwrite	1	non	0 : lecture, 1 : écriture
hburst	3	non	indique s'il y a un « burst transfer » et ses caractéristiques, détaillé plus loin
hsize	3	non	taille des données du cycle en cour (0 : 8bits, 1 : 16 bits, 2 : 32 bits, …, 7 : 1024 bits)
htrans	2	non	type de transfert en cour, détaillé plus loin
hprot	4	non	signaux de protection, détaillé plus loin
hready	1	non	0 : wait state, 1 : normal

hmaster	4	non	index du master qui fait l'opération
hmastlock	1	non	0 : normal, 1 : opération indivisible
hmbsel	4	oui	sélection de banque
hcache	1	oui	données "cachable"
hirq	32	oui	vecteur d'interruption

La colonne « gaisler » indique si le signal provient des librairies Gaisler ou s'il était déjà donné par la spécification de ARM.

Le tableau suivant présente les signaux du bus ahbso :

Nom du signal	Nbr de bits	gaisler	Fonction
hready	1	non	0 : wait state, 1 : normal
hresp	2	non	indique si le transfert s'est bien passé (0 : ok, 1 : error, 2 : retry, 3 : split)
hrdata	32	non	données pour une opération de lecture
hsplit	16	non	donne le numéro du master pour terminer un « split transfer »
hcache	1	oui	données "cachable"
hirq	32	oui	vecteur d'interruption
hconfig	8 * 32	oui	données pour le "plug and play", détaillé plus loin
hindex	4	oui	numéro d'index du slave

Les signaux clock et reset (actif bas) sont externes au bus AMBA.

Détails du signal htrans :

Le signal htrans signal quel type de transfert est en cour. Le tableau suivant décrit les quatre différents types :

Valeur	Type	Description
0	IDLE	Aucun transfert est en cour, le maître a le contrôle du bus mais n'a pas besoin d'effectuer de transfert
1	BUSY	BUSY indique que le maître du bus continue un burst transfer, mais le prochain transfert ne peut pas avoir lieu immédiatement.
2	NONSEQ	Indique un transfert normal ou le premier transfert d'un burst.
3	SEQ	transfert burst

Détails du signal hprot :

Le signal de protection hprot procure des informations qui sont sur l'accès du bus et qui sont principalement destinées pour les modules qui souhaitent implémenter des niveaux de protections.

hprot(3)	hprot(2)	hprot(1)	hprot(0)	Description
-	-	-	0	opcode fetch
-	-	-	1	data access
-	-	0	-	user access
-	-	1	-	privileged access
-	0	-	-	not bufferable
-	1	-	-	bufferable
0	-	-	-	not cacheable
1	-	-	-	cacheable

Il est à noter que tous les masters ne sont pas capables de fournir les informations de protection. C'est pourquoi il est recommandé que les slaves n'utilise le signal hprot uniquement s'il est strictement nécessaire.

Détails des signaux hconfig :

La compréhension des signaux hconfig nécessite une bonne connaissance du fonctionnement du « plug and play » des librairies de Gaisler. Chaque module master / slave doit avoir ses signaux hconfig correctement configurés pour qu'il soit reconnu par les autres périphériques. L'annexe A présente ceci dans les détails.

Détails du signal hburst :

Un « burst transfer » est une lecture / écriture qui se fait sur plusieurs adresses qui se suivent. Le bus AMBA AHB définit le « wrapping burst » et l'« incrementing burst » pouvant chacun effectuer 4, 8 et 16 transferts. De plus, l'« incrementing burst » peut avoir un nombre indéfini de transferts.
L'incrementing burst signifie que la prochaine adresse est une de plus que l'actuelle. Par exemple 0x1C, 0x20, 0x24, 0x28, 0x2C, 0x30, 0x34, …
Pour le wrapping burst, si l'adresse de départ n'est pas alignée sur le nombre total de bytes dans le burst (taille x nbr de transferts) alors l'adresse des transferts dans le burst va sauter quand la limite du nombre total de bytes, alignée dans la mémoire, est atteinte. De ce fait, si l'adresse de départ est 28,

Autre exemple, pour un wrap 8 avec une taille de transfert de 32 bits qui commence à l'adresse 0x34. Le nombre total de bytes du burst est 8 * 32 bits = 32 bytes. Les adresses s'incrémentent jusqu'à ce qu'elles atteignent la fin du « bloque de mémoire ». A ce moment elles sautent au début de ce bloque. Ainsi, les adresses seront dans l'ordre : 0x34, 0x38, 0x3C, 0x20, 0x24, 0x28, 0x2C, 0x30.

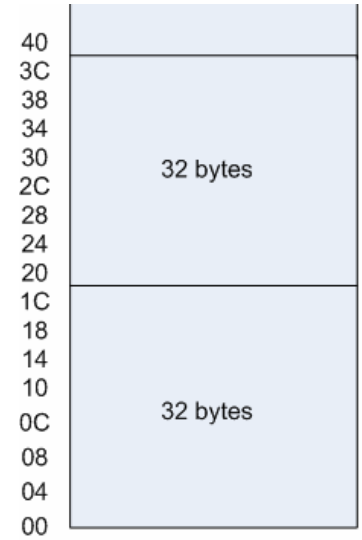


Figure 6 : Illustration Wrapping burst

Le signal hburst défini les caractéristiques du transfert en cour, le tableau suivant présente ses différentes valeurs et significations :

Valeur	type	Nombre de cycle
0	-	1
1	INCR	inconnu
2	WRAP	4
3	INCR	4
4	WRAP	8
5	INCR	8
6	WRAP	16
7	INCR	16

Il est à noter qu'un transfert unique peut avoir la valeur 1 en plus de la valeur 0.
La taille du burst indique le nombre de cycle du burst et pas le nombre de bytes transférés. Le nombre total de données transférées dans un burst se calcul en multipliant le nombre de cycle par le nombre de byte par cycle indiqué par le signal HSIZE.
Tous les transferts d'un burst doivent être alignés sur l'adresse dont il est la cible, par rapport à la taille des données du transfert. Par exemple, un transfert de word (32 bits) doit avoir les 2 LSB des adresses à 0, pour un halfword le LSD doit être à 0.

Opération simple sur le bus AMBA :

La figure suivante montre le transfert le plus simple sur le bus AMBA - AHB, sans wait-state :

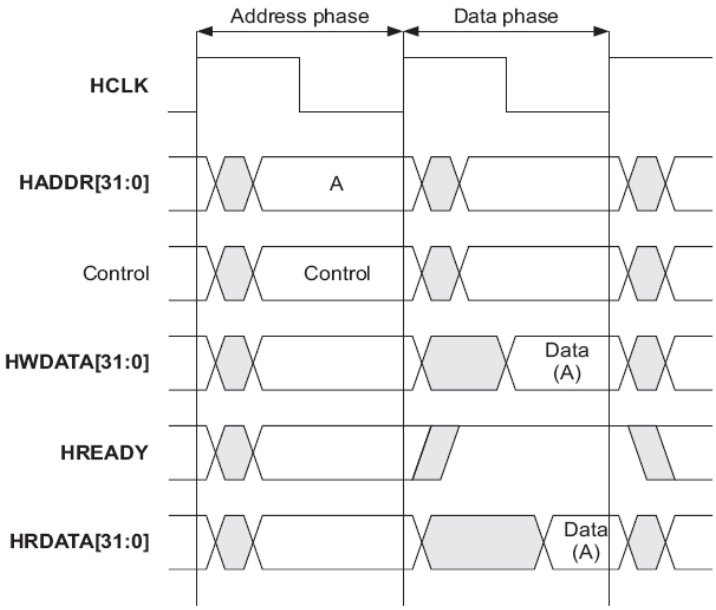


Figure 7 : Transfert simple sur bus AMBA - AHB

Un transfert se fait en deux temps : la phase d'adressage et la phase de donnée.
Le master met les adresses et les bits de contrôle stable pour un premier flanc montant du clock. Dans ce cas, le slave est prêt à recevoir un transfert et met HREADY à 1. Si hwrite est à 1 (écriture) les données HWDATA devront être stable pour le flanc montant suivant, sinon c'est une lecture et le slave devra mettre la donnée sur HRDATA pour ce même flanc montant.

Les adresses et les bits de contrôles peuvent changer lors de la phase des données. En effet, ceci est dû à la structure « pipeline » du bus AMBA. Un nouveau transfert a débuté avec de nouvelles adresses et de nouveaux bits de contrôle lors de la phase des données du cycle précédent.

La figure suivante montre un transfert simple sur le bus AMBA - AHB, avec des wait-states :

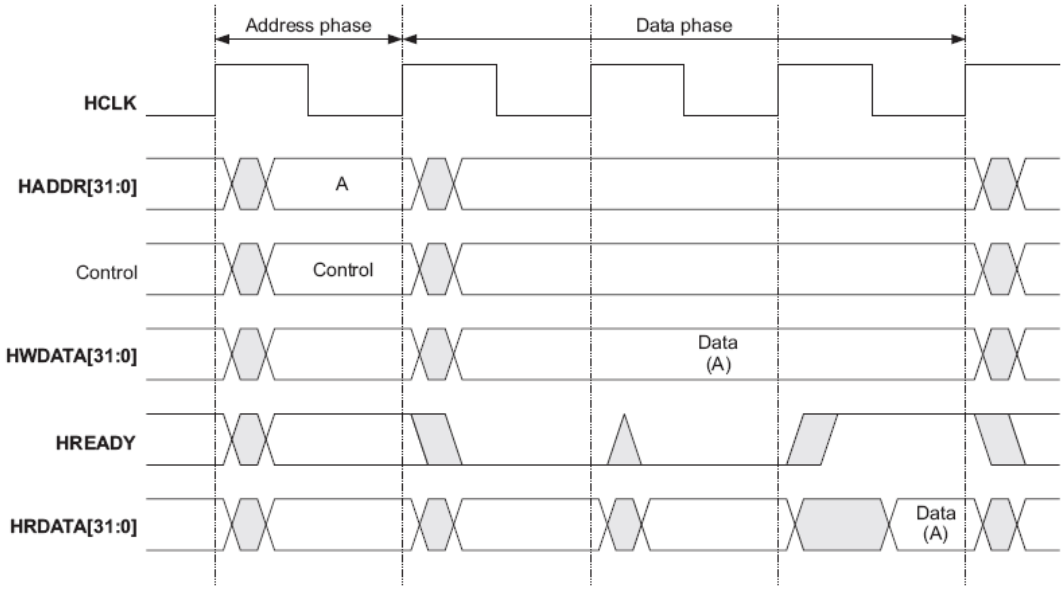


Figure 8 : Wait-state sur le bus AMBA - AHB

Lorsque un slave n'est pas prêt à exécuter un transfert, il a la possibilité d'insérer des wait-states. Un master a également cette liberté, mais il le fera plutôt sur des burst transfers. Le wait-state se fait donc en forçant HREADY à 0.
Pour une opération d'écriture, les données doivent rester stable le temps que le slave ait remis HREADY à 1. Pour une opération de lecture, les données seront stables que pour le dernier flanc montant du clock.
Les adresses et les bits de contrôle du prochain transfert seront ainsi rallongés jusqu'à ce que le slave / master lent soit prêt.

La figure suivante présente un incremental burst sur le bus AMBA – AHB :

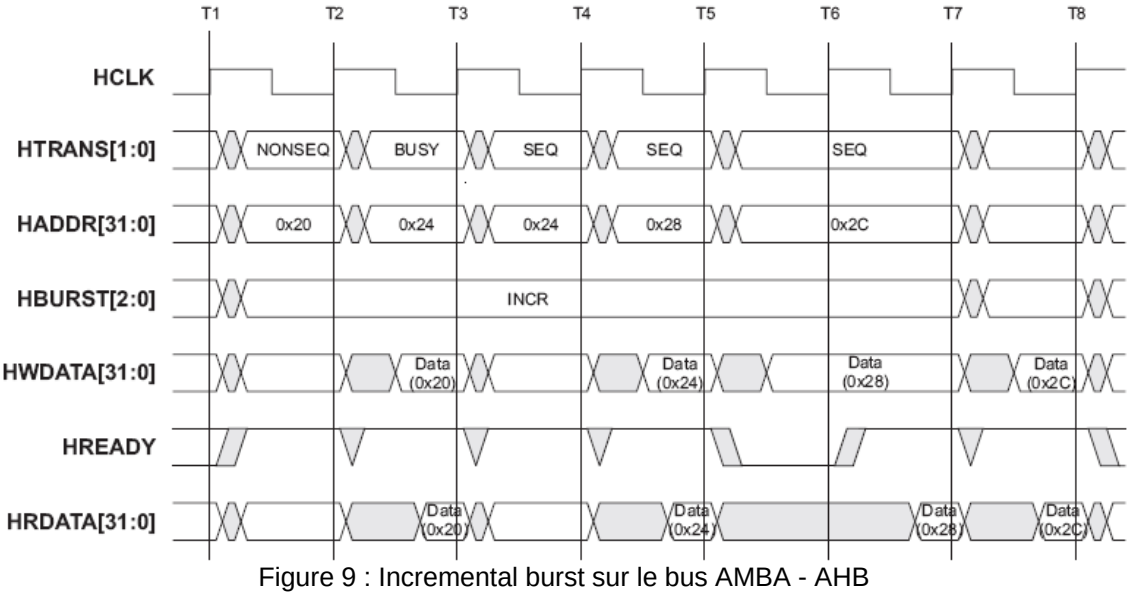


Figure 9 : Incremental burst sur le bus AMBA - AHB

Un burst transfer commence toujours avec HTRANS qui a la valeur NONSEQ, ce qui peut aussi signifier que c'est un transfert unique. Dans cet exemple, le burst est incrémental, donc les adresses se suivent. Dans le 2ème cycle, le master est BUSY, c'est pourquoi il remet la même adresse au temps T4 qu'au temps T3.
Le master exécute le 3ème transfert immédiatement, mais cette fois, le slave est incapable d'achever l'opération et utilise HREADY pour insérer un wait-state. Le dernier transfert du burst s'exécute sans wait-state.

La figure suivante présente un wrapping burst sur le bus AMBA – AHB :

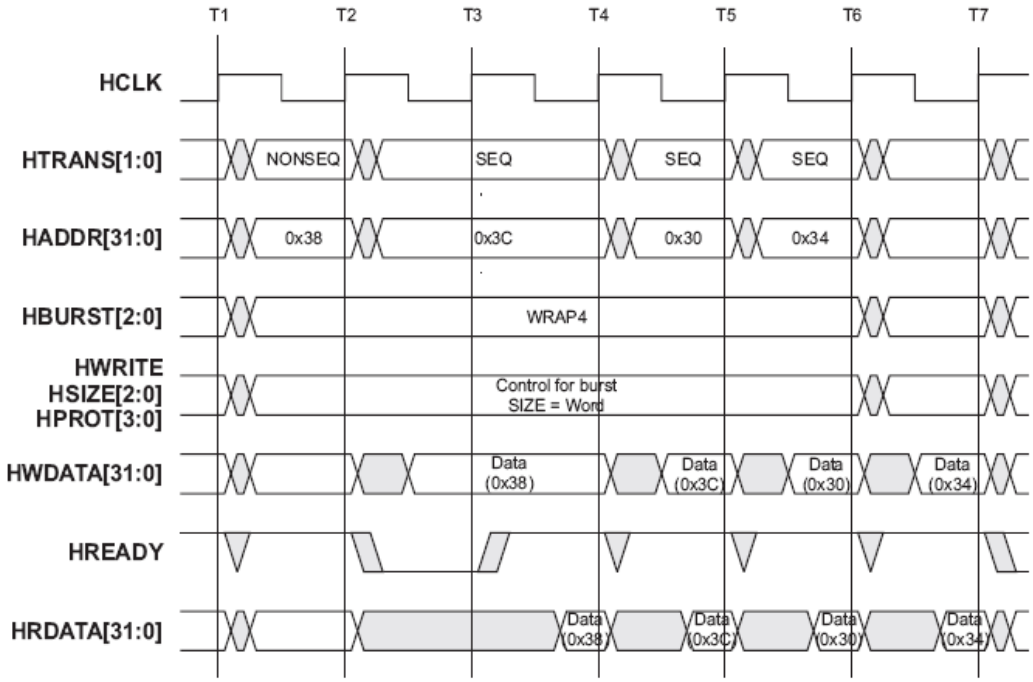


Figure 10 : Wrapping burst sur le bus AMBA - AHB

Un wrapping burst se déroule de la même manière qu'un incremental burst. Seules les adresses changent.

2.1.2.2 Le bus AMBA – APB

Le bus APB est généré à partir du pont AHB-APB. Le pont possède une certaine plage d'adresse par rapport aux masters, il traduit donc les transferts de données qui correspondent à cette plage d'adresse.

La figure suivante présente la vue générale du bus APB :

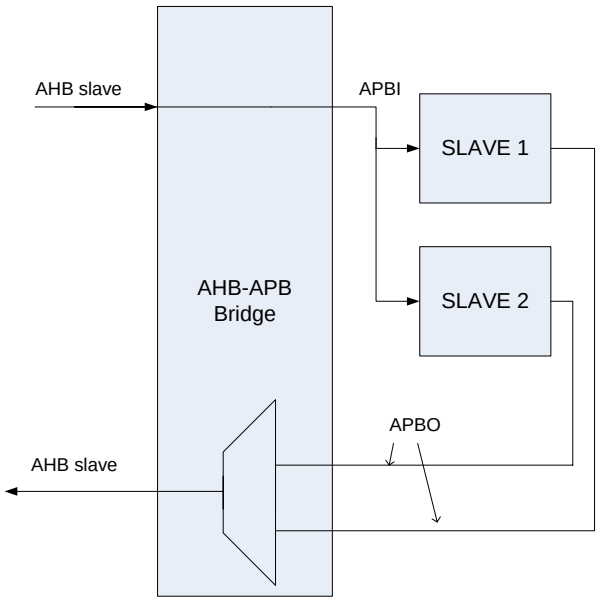


Figure 11 : Bus APB

Comme pour le bus AHB, tous les slaves ont le même bus en entrée et chacun a le sien en sortie.

Le tableau suivant liste les signaux du bus apbi :

Nom du signal	Nbr de bits	gaisler	Fonction
psel	16	non	sélection du slave (un bit par slave)
penable	1	non	indique une opération en coure
paddr	32	non	adresse du transfert en coure
pwrite	1	non	0 : lecture, 1 : écriture
pwdata	32	non	données pour une opération d'écriture
pirq	32	oui	vecteur d'interruption

Le tableau suivant liste les signaux du bus apbo :

Nom du signal	Nbr de bits	gaisler	Fonction
prdata	32	non	données pour une opération de lecture
pirq	32	oui	vecteur d'interruption
pconfig	2 * 32	oui	données pour le "plug and play", détaillé plus loin
pindex	4	oui	numéro d'index du slave

La colonne « gaisler » indique si le signal provient des librairies Gaisler ou s'il était déjà donné par la spécification de ARM.
Le bus APB ne peut donc pas faire de burst transfert et un slave APB n'a aucun moyen d'avertir le master qu'il n'est pas prêt pour un transfert. Tous les transferts ont donc une durée fixe.

Détails du pconfig :

La compréhension des signaux pconfig nécessite une bonne connaissance du fonctionnement du « plug and play » des librairies de Gaisler. Chaque slave APB doit avoir ses signaux pconfig

correctement configurés pour qu'il soit reconnu par le pont AHB / APB. L'annexe A présente ceci dans les détails.

La figure suivante présente une opération d'écriture sur le bus APB :

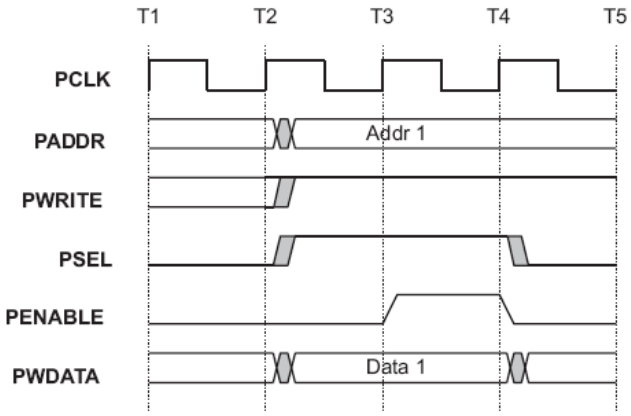


Figure 12 : Ecriture sur le bus AMBA - AP

Les adresses, les données, la sélection et un état haut pour le write enable sont placés par le pont après un flanc montant du clock. Le enable est mis à 1 au coup d'horloge suivant et le slave doit avoir exécuté l'opération au 3ème flanc montant du clock.

La figure suivante présente une opération de lecture sur le bus APB :

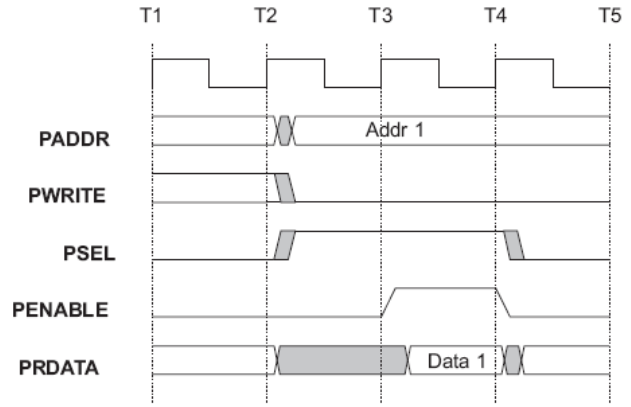


Figure 13 : Lecture sur le bus AMBA - APB

Les adresses, la sélection et un état bas pour le write enable sont placés par le pont après un flanc montant du clock. Le enable est mis à 1 au coup d'horloge suivant et le slave doit avoir exécuté l'opération au 3ème flanc montant du clock.

2.1.3 Le bus Wishbone

Le bus Wishbone a été développé par OpenCores qui a été fondé en 1999 par Damjan Lanpret. OpenCores est une communauté qui œuvre dans le but du développement et de la distribution de composants en VHDL et Verilog en open source. Ces composants utilisent le bus Wishbone comme moyen de communication et ils peuvent être téléchargés gratuitement via le site officiel OpenCores.org.

Figure 14 : Logo Opencores.org

Le bus Wishbone n'a pas d'interconnexion strict entre les masters et les slaves. La spécification propose quelques solutions d'architecture. Le développeur doit décider quelle solution est la plus adaptée pour son système. Et si les contraintes de son projet change, il pourra passer d'une solution à une autre sans changer les composants.

Les 4 types d'interconnexions définies par Wishbone sont :
- Point-to-point
- Data flow
- Shared bus
- Crossbar switch

Malgré ces possibilités, rien n'empêche un développeur de créer sa propre interconnexion.

Point-to-point

L'interconnexion point-to-point est la voie la plus simple pour connecter deux modules Wishbone ensemble. Cela permet à un seul master de se connecter à un seul slave. Cette option est donc très limitée.
La figure suivante est le schéma du modèle point-to-point :

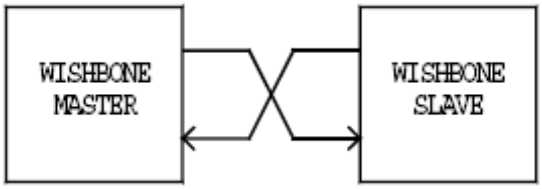


Figure 15 : Interconnexion Wishbone Point-to-point

Data flow

L'interconnexion Data flow est utilisée quand les données procèdent de manière séquentiel.
Comme il est visible dans la figure suivante, l'architecture Data flow a à la fois une interface master et une interface slave :

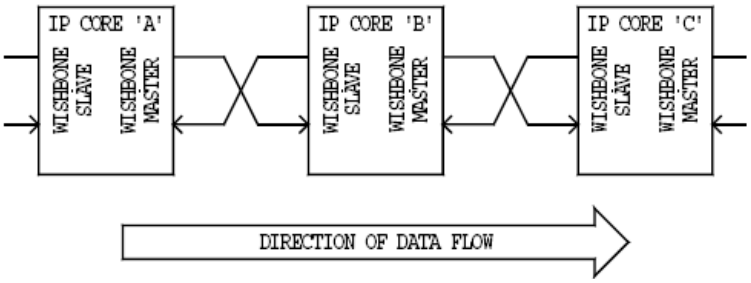


Figure 16 : Interconnexion Wishbone Data Flow

Shared bus

L'architecture shared bus est utilisée quand plus d'un master et / ou plus d'un slave doivent être connectés ensemble. Tous les masters et les slaves partagent le même bus. Un arbitre (pas présent sur la figure) commande les masters et choisi lequel peut avoir le bus. La spécification Wishbone ne dit pas comment le « shared bus » doit être implémenté.
La figure suivante présente la vue de l'architecture Shared bus :

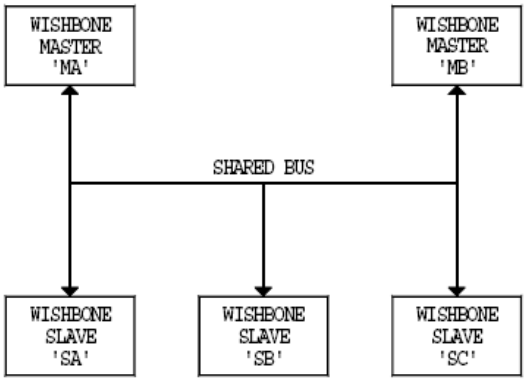


Figure 17 : Interconnexion Wishbone Shared bus

Crossbar switch

L'architecture Crossbar switch est utilisée quand on a à connecter plus d'un master et / ou plus d'un slave. Chaque master et slave a sa propre connexion à l'arbitre. L'arbitre commande les masters et s'occupe de faire suivre les informations entre les slaves et les masters.
La figure suivante montre un exemple de l'architecture Crossbar switch :

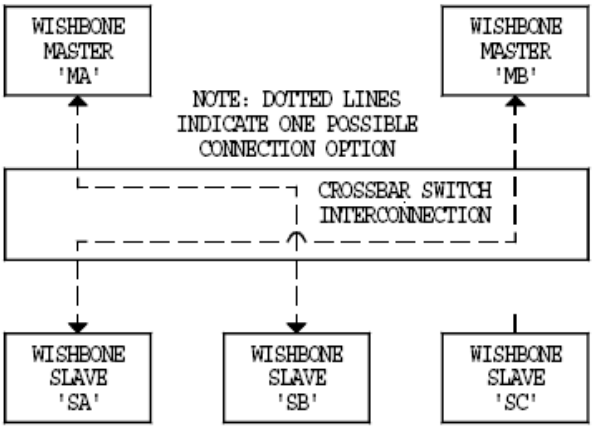


Figure 18 : Interconnexion Wishbone Crossbar switch

Le tableau suivant présente les signaux qui vont du master Wishbone au slave :

Nom du signal	Nombre de bits	Fonction
clk	1	signal d'horloge
rst	1	signal de reset
adr	variable	adresse du transfert en cour
dat	variable	données pour une opération d'écriture
sel	16	sélection du slave
we	1	0 : lecture, 1 : écriture
bte	2	type de burst (détaillé plus loin)
cti	3	type de transfert (détaillé plus loin)
cyc	1	indique un cycle (burst ou transfert simple) en cour, il reste actif tout le long d'un burst
stb	1	indique un transfert (un transfert dans un burst ou un transfert simple) en cour
lock	1	indique un transfert indivisible

Le tableau suivant présente les signaux qui vont du slave au master :

Nom du signal	Nombre de bits	Fonction
dat	variable	données pour une opération de lecture
ack	1	indique la terminaison d'un transfert valide
err	1	indique une terminaison d'un transfert anormal
rty	1	indique que le slave n'est pas prêt à recevoir une opération (wait-state)

Tous les signaux Wishbone, sans exception, sont actifs haut.

Détails du signal cti :

Le signal cti (Cycle Type Identifier) fournit des informations sur le cycle courant. Le master envoie ces informations au slave. Le slave peut utiliser ces informations pour préparer la réponse pour le prochain cycle.

Le tableau suivant décrit la signification des différentes valeurs de cti :

Valeur	Description
0	cycle classique
1	burst à une adresse constante
2	burst incrémental
3	réservé
4	réservé
5	réservé
6	réservé
7	fin de burst

Tous les masters et les slaves ne sont pas obligés de supporter un transfert de burst et donc ils peuvent ne pas avoir le signal cti.
Lors du dernier transfert d'un burst, le signal cti doit être à 7.

Détails du signal bte :

Le signal bte (Burst Type Extension) est envoyé par le master au slave pour fournir des informations additionnels sur le burst en cour. Ces informations sont viables uniquement pour le burst incrémental.

Le tableau suivant décrit la signification des différentes valeurs de bte :

Valeur	Description
0	burst linéaire
1	wapping burst - 4 coups
2	wapping burst - 8 coups
3	wapping burst - 16 coups

Les masters et slaves qui peuvent supporter un burst doivent avoir le signal bte.

Le Wishbone défini le wrap burst ainsi :

LSB de l'adresse de départ	linéaire	wrapping 4	wrapping 8
000	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
001	1-2-3-4-5-6-7-8	1-2-3-0-5-6-7-4	1-2-3-4-5-6-7-0
010	2-3-4-5-6-7-8-9	2-3-0-1-6-7-4-5	2-3-4-5-6-7-0-1
011	3-4-5-6-7-8-9-A	3-0-1-2-7-4-5-6	3-4-5-6-7-0-1-2
100	4-5-6-7-8-9-A-B	4-5-6-7-8-9-A-B	4-5-6-7-0-1-2-3
101	5-6-7-8-9-A-B-C	5-6-7-4-9-A-B-8	5-6-7-0-1-2-3-4
110	6-7-8-9-A-B-C-D	6-7-4-5-A-B-8-9	6-7-0-1-2-3-4-5
111	7-8-9-A-B-C-D-E	7-4-5-6-B-8-9-A	7-0-1-2-3-4-5-6

La figure suivante présente deux transferts simples sur le bus Wishbone :

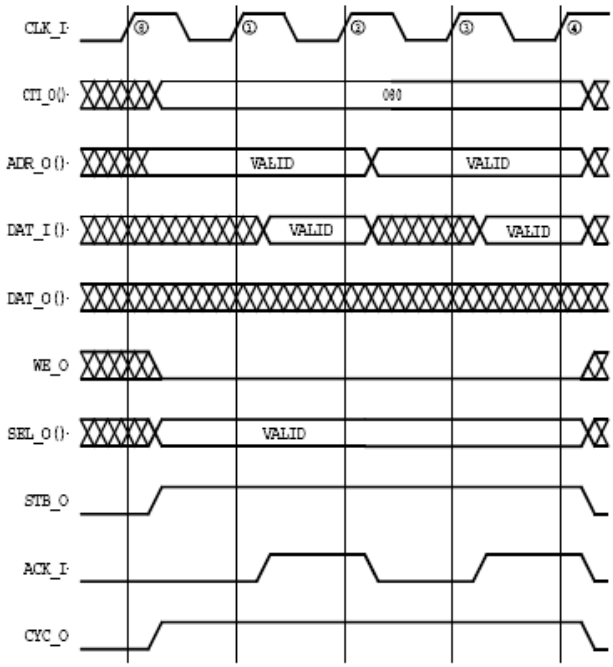


Figure 19 : Transfert simple sur le bus Wishbone

La convention de nomenclature de Wishbone ajoute des _I et _O à la fin du nom d'un signal pour indiquer si c'est une entrée ou une sortie. Ici, ces suffixes sont en rapport au master.
Le master met l'adresse, le write enable, la sélection, le strobe et cycle stable pour un premier flanc montant du clock. Dans ce cas, c'est une opération de lecture et le slave retourne donc la donnée sur son bus dat. Il rend aussi actif le signal ack pour indiquer que l'opération s'est bien passée. Cette opération se répète une nouvelle fois à partir du 3ème flanc montant. Le signal strobe n'a pas eu besoin d'être inactif entre les deux opérations, comme elles se succèdent.

Le signal strobe permet au slave de savoir qu'une opération est en cour, lorsque le slave est sélectionné, il doit mettre dès que possible son signal ack à 1. Cela peut prendre 0 à plusieurs clock suivant la rapidité du slave. Le signal ack peut donc être asynchrone pour avoir plus de rapidité de réaction. Quand le master reçoit le signal ack à 1, il sait que le slave a exécuté l'opération au flanc montant et peut donc s'il n'a pas d'autre opération à faire, mettre le signal strobe à 0, comme le montre la figure suivante.

Figure 20 : Signal ACK sur le bus Wishbone

La figure suivante présente un burst wrap de 4 lectures sur le bus Wishbone :

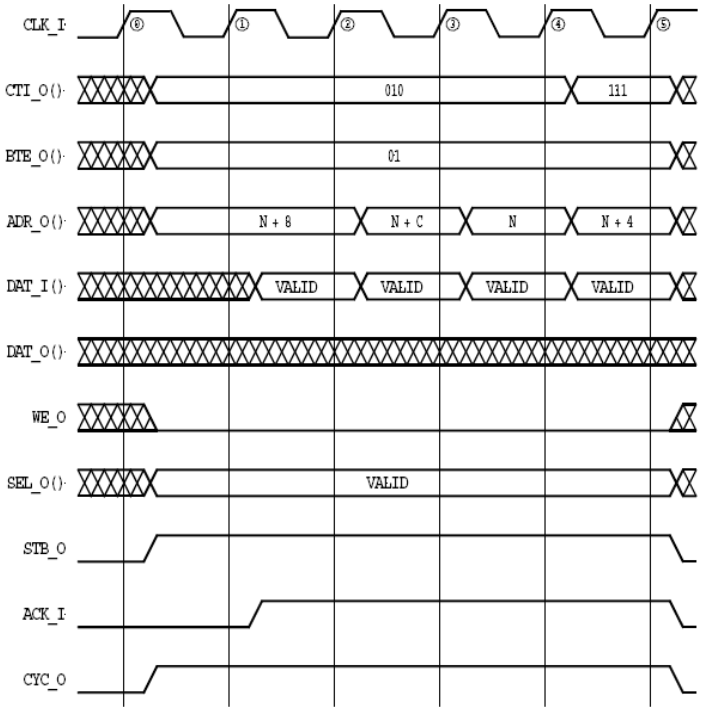


Figure 21 : Wrapping burst sur le bus Wishbone

Le master met le cti à 2, le bte à 1, l'adresse, le write enable, la sélection, le strobe et le cycle pour le premier flanc montant. Il garde ces valeurs pour le prochain flanc montant où le slave aura répondu avec son bus dat et le signal ack. Lors des transferts suivants, les opérations se font en un coup d'horloge. Le slave s'attend à avoir un autre transfert et prépare sa réponse, tandis que le master regarde le signal ack pour terminer la phase des données. Le dernier transfert se termine par un cti à 7 pour signaler que c'est la fin du burst. Le signal stb a été actif tout le long du burst, comme celui-ci n'a pas été interrompu.
La figure suivante présente un burst à adresse constante avec 4 écritures sur le bus Wishbone :

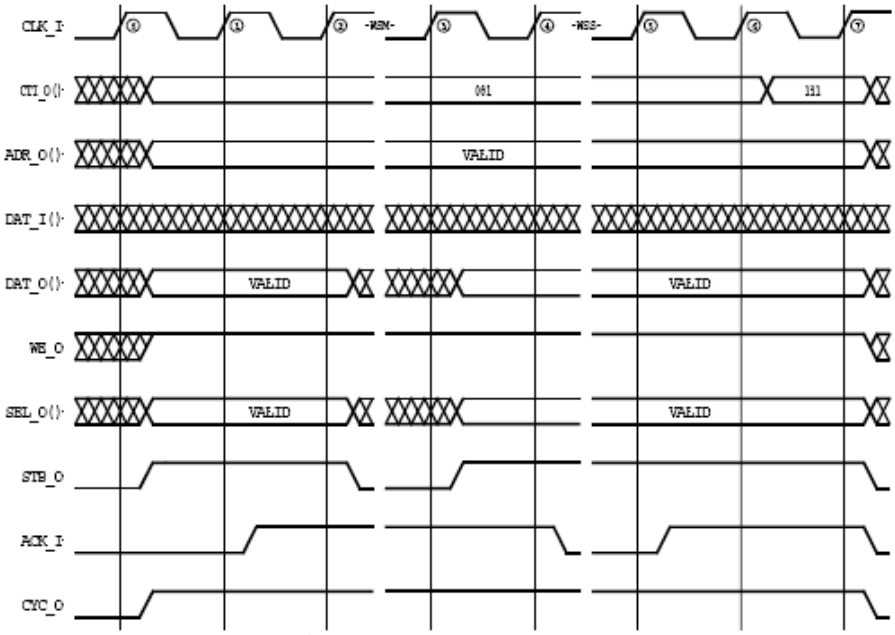


Figure 22 : Burst à adresse constante sur le bus Wishbone

Le master place l'adresse, la valeur 1 à cti, les données, la sélection, le strobe et le cycle sur le bus pour un premier flanc montant du clock. Il garde ces valeurs pour le prochain flanc montant où le slave aura répondu avec le signal ack. Le transfert est interrompu après le flanc montant n° 2, le signal strobe se met à 0 alors que le cycle reste à 1.
Lors du flanc montant n° 5, le slave n'est pas prêt et a donc mis le signal ack à 0.

2.2 Cahier des charges

Le cahier des charges détermine précisément les contraintes que le résultat final du projet doit maîtriser.

Pont AMBA - AHB / Wishbone

Ce pont doit faire la conversion entre les deux bus cités. Il doit donner toutes les informations qu'ils ont en commun d'un bus à l'autre. N'importe quel slave Wishbone doit pouvoir se connecter au pont et recevoir des opérations de lecture et d'écriture. C'est-à-dire qu'un slave rapide autant qu'un slave lent reste compatible avec le pont.

Burst et registres

Le pont AHB – Wishbone doit pouvoir gérer les transferts dit de « burst ». Des registres Wishbone sont placés pour tester les ponts et contrôler quel émetteur SPDIF est en sortie.

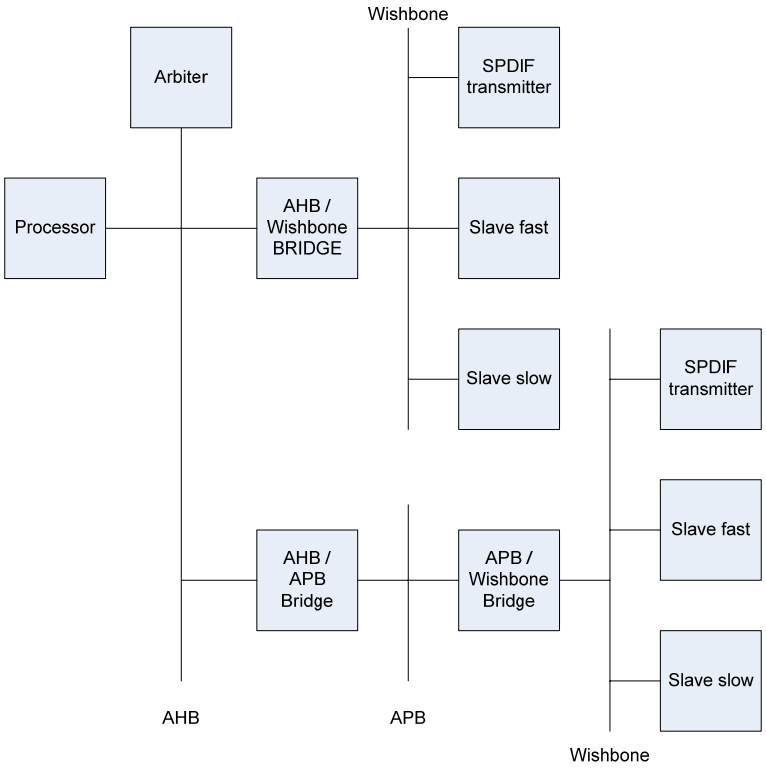


Figure 23 : Vue générale du cahier des charges

Démonstrateur SPDIF

Le circuit doit être implémenté en physique sur la FPGA de la carte choisie. Il doit être possible de charger un fichier sur une mémoire, puis de l'écrire sur un émetteur SPDIF branché à des haut-parleurs.

2.3 Déroulement et planning

Il est vain de décrire un déroulement daté et précis au début d'un projet sur les FPGA. En effet, un petit problème peut parfois prendre plusieurs jours, voir semaines à être résolu. De même qu'on peut avoir beaucoup de chance et réussir un objectif du premier coup.

Néanmoins, il est possible de décrire une liste de tâche à suivre pour arriver au but.

- Terminer l'implémentation du pont AMBA – APB / Wishbone
- Chargement d'un fichier en mémoire
- Lecture du fichier sur les haut-parleurs
- Création du pont AMBA AHB / Wishbone
- Gestion des bursts avec le pont et développement de registres Wishbone
- Implémenter le nouveau pont sur la FPGA

Le projet se déroule du 3 septembre au 23 novembre, date de remise du rapport.

3. Création du schéma de base

3.1 Introduction

Cette partie consiste à assembler les blocs qui environnent le processeur LEON3 de manière à ce qu'il puisse exécuter un programme. Cela doit être fait avant de créer le pont pour avoir un système avec le bus AMBA qui tourne correctement.
La majorité du schéma de base a été développée lors du projet de semestre sur la base des librairies de Gaisler. Cependant, la partie DSU et les UARTs ont été rajoutés par après.

La figure suivante présente les blocs de base qui sont connectés au bus AMBA :

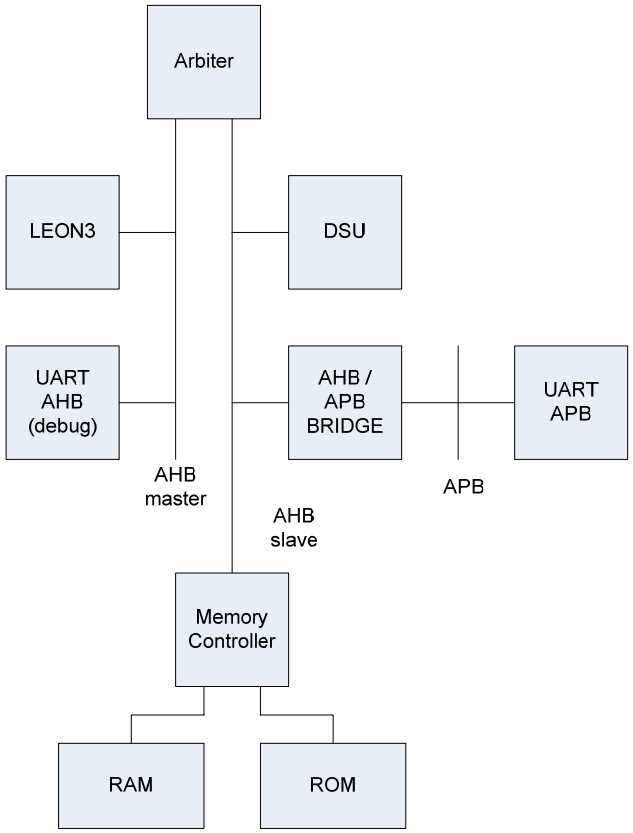


Figure 24 : Vue générale du schéma de base

Chaque périphérique a des bits de configuration appelés génériques. Toutes ces variables sont répertoriées dans le fichier config.vhd fourni en annexe B. Tous les slaves possèdent des génériques qui permettent, entre autre, de spécifier leur plage d'adresse.

Le document officiel grip.pdf décrit dans les détails chacun des composants utilisés dans ce circuit. Ceci n'est donc qu'un résumé des blocs utilisé de ce document.

3.2 Le processeur LEON3

Le modèle VHDL du LEON implémente un processeur 32 bits qui fonctionne conformément à l'architecture SPARC V8. Il a été développé pour des applications embarquées avec les caractéristiques on-chip suivantes : pipeline de 7 étages avec une architecture Harvard, instructions et cache de données séparées, multiplieur et diviseur hardware, unité de debug et des extensions multiprocesseur.
Il est possible d'en synthétiser plusieurs en parallèle.

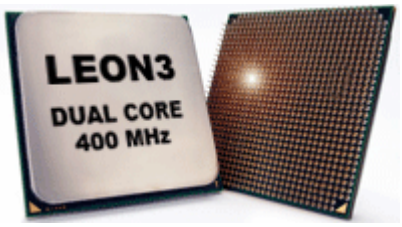


Figure 25 : Illustration du processeur LEON

Le LEON3 est fournit dans la librairie gaisler. Un certain nombre de génériques sont à configurer en fonction du circuit dans lequel il sera implémenté. L'annexe B contient les génériques de tous les périphériques du circuit.

Figure 26 : Illustration du processeur LEON

Lors du démarrage, le processeur va chercher les instructions du programme à l'adresse 0x0.

Le processeur LEON3 ne fait des transferts de burst uniquement pour chercher ses instructions dans la mémoire, à l'exception d'une écriture d'**une** variable de grandeur supérieur à 32 bits où il exécute une burst incrémental. Par exemple, une boucle for ou while ne crée pas de burst.

3.3 Arbitre

L'arbitre s'occupe non seulement de gérer les masters, mais il sert aussi de pont AMBA – AHB master / AHB slave. Comme expliqué dans l'annexe A, il crée aussi une zone mémoire où sont stockées les informations des différents slaves et masters avec, entre autre, le mapping de ceux-ci.

Ce contrôleur peut supporter jusqu'à 16 masters et 16 slaves. Deux algorithmes d'arbitration sont à disposition et peuvent être choisi à l'aide du générique rrobin : priorité fixe, la priorité est donnée par le numéro d'index du master (rrobin = 0) et round robin où l'accès au bus change à chaque transfert.

3.4 DSU et UART AHB

Comme il est expliqué dans le chapitre 4.3, un DSU (Debug Support Unit) doit être présent dans le circuit pour charger un programme dans la mémoire que le processeur peut exécuter. Il est aussi naturellement d'une grande aide pour le debug de ce programme.

Le DSU est considéré comme un slave AHB. Il a malgré cela une connexion directe de debug (dbgi et dbgo) avec les masters. Il peut donc, grâce à cette connexion, contrôler les processeurs. Un programme informatique peut donc piloter le DSU via plusieurs possibilités d'interface.
La carte utilisée dans le projet a deux interfaces RS422 avec un convertisseur RS232 / RS422. Cette solution a donc été choisie.
L'UART AHB agi comme un AHB master contrôlé par le programme depuis le PC et à la fois un slave APB d'où un master peut renvoyer des informations.

Il devient ainsi possible de placer des breakpoints, d'avoir des watchpoints, exécuter des opérations single-step, …

La figure suivante présente le fonctionnement du DSU :

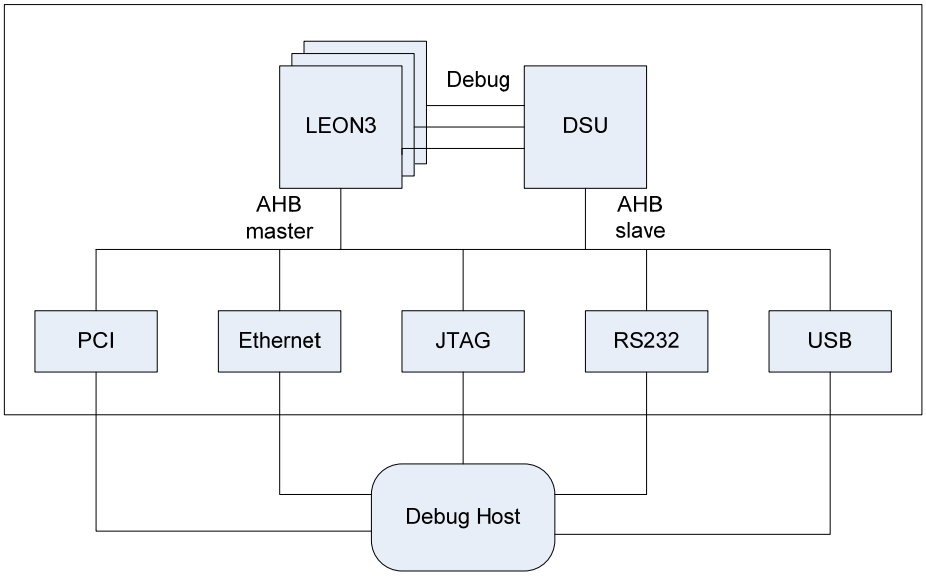


Figure 27 : Fonctionnement du DSU

3.5 Pont AHB / APB

Le pont AHB / APB est le « master » du bus APB. Ce contrôleur supporte jusqu'à 16 slaves APB. Il reconnaît l'adresse des slaves avec le même principe plug and play que l'arbitre le fait avec le bus AHB. Les slaves ont chacun deux mots de 32 bits nommés pconfig qui contiennent leurs informations (adresses, masque, Vendor ID, Device ID, …). Si le pont AHB / APB est mappé à l'adresse 0x80000000, ces informations plug and play sont enregistrées à 0x800FF000.
Toujours de la même manière que l'arbitre, le pont conduit les informations du slave APB qui a une opération en cours sur le bus AHB à l'image d'un multiplexeur.

3.6 Contrôleur de mémoire

Le contrôleur de mémoire gère un bus de mémoire (memi et memo) ayant la capacité de contrôler des PROM, I/O, ram statique asynchrone SRAM et ram dynamique synchrone SDRAM. Le contrôleur est un slave du bus AHB, mais ses registres de configuration s'accèdent depuis le bus APB.
Le décodage du chip-select est fait pour deux banques PROM, une banque I/O, cinq banques SRAM et deux banques SDRAM. Mais trois espaces d'adresses (PROM, I/O et RAM) sont déterminés à partir des génériques.

La SDRAM est gérée par un contrôleur qui est spécialement crée à cet effet, le SDCTRL, SDRAM controller de Gaisler Research, qui est instancié optionnellement dans le contrôleur de mémoire. Le générique sden du contrôleur de mémoire permet de l'inclure. Le SDCTRL supporte des mémoires de 64M, 256M et 512M avec 8 à 12 bits de colonnes d'adresses et jusqu'à 13 bits de lignes d'adresses. Les opérations du SDCTRL sont configurées à l'aide des registres MCFG2 et MCFG3 (voir plus loin). La plage d'adresse de la SDRAM se trouve dans la moitié supérieure de celle de la RAM. La plage d'adresse de la RAM doit donc être choisie suffisamment grande pour y placer celle de la SDRAM.
Quand le SDCTRL est enclenché dans le MCFG2, il crée automatiquement les séquences de PRECHARGE, 2 x AUTO-REFRESH et LOAD-MODE-REG sur les deux banques simultanément.
Le contrôleur de SDRAM contient une fonction de refresh qui exécute périodiquement la commande AUTO-REFRESH sur chaque banque de la SDRAM. La période entre deux commandes (en période de clock) est programmée dans le compteur de recharge du registre MCFG3. En fonction du type de SDRAM, la période requise est typiquement 7.8 ou 15.6us, ce qui correspond à 780 ou 1560 clock à 100MHz.
La SDRAM requiert une synchronisation spéciale. Un clock inversé peut être produit.

La figure suivante présente un exemple d'une vue mémoire que le contrôleur peut gérer :

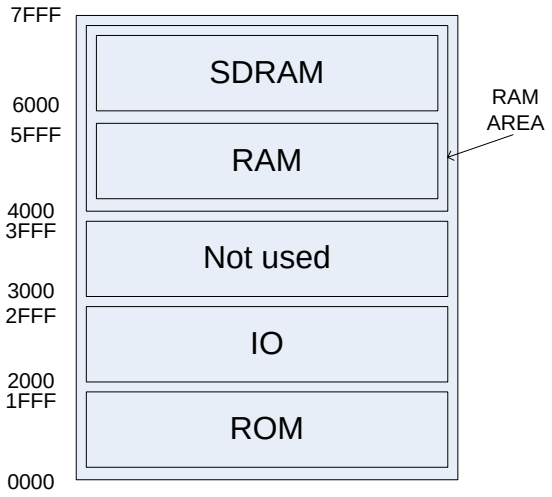


Figure 28 : Memory mapping du contrôleur de mémoire

Le tableau suivant présente les registres du contrôleur de mémoire :

Register	APB address offset
MCFG1	0x0
MCFG2	0x4
MCFG3	0x8

Le Memory Configuration register 1, MCFG1, est utilisé pour programmer les timings et accès de la rom et des I/O.

Bits	Description
31 : 29	réservés
28 : 27	largeur du bus I/O : "00" : 8, "01" : 16, "10" : 32 bits
26	Enable du ready du bus I/O
25	Bus error enable
24	réservé
23 : 20	Nbre de wait-states du bus I/O : "0000" = 0, "0001" = 1, "0010" = 2,..., "1111" = 15
19	Enable du bus I/O
18 : 12	réservés
11	Mettre ce bit à 1 permet d'écrir sur la PROM
10	réservé
9 : 8	largeur du bus PROM : "00" : 8, "01" : 16, "10" : 32 bits
7 : 4	Nbre de wait-states du bus de la PROM en écriture : "0000" = 0, "0001" = 1, "0010" = 2,..., "1111" = 15
3 : 0	Nbre de wait-states du bus de la PROM en lecture : "0000" = 0, "0001" = 1, "0010" = 2,..., "1111" = 15

Le registre MCFG2 est utilisé pour contrôler les timings de la SRAM et la SDRAM.

Bits	Description
31	Enable du refresh de la SDRAM
30	Timing Trp, 0 : 2 clock, 1 : 3 clock
29 : 27	Timing Trfg, 3 + valeur clock
26	Timing Tcas, 0 : 2 cycles CAS, 1 : 3 cycles CAS
25 : 23	Taille des banques SDRAM, "000" = 4Mbyte, "001" = 8 Mbyte, "010" = 16 Mbyte.... "111" = 512Mbyte
22 : 21	Taille des colonnes de la SDRAM, "00" = 256, "01" = 512, "10" = 1024, "11" = 4096 quand les bits 25:23 = "111" sinon 2048
20 : 19	Génère une commande SDRAM, "01" = PRECHARGE, "10" = AUTO-REFRESH, "11" = LOAD-COMMAND-REGISTER
18	Donne la taille du bus SDRAM, 0 : 32 bits, 1 : 64 bits, lecture seulement
17 : 15	réservés
14	Enable du SDCTRL
13	Disable de la SRAM
12 : 9	Taille d'une banque de la RAM, "0000"=8 kbyte, "0001"=16kbyte, ..., "1111"=256 Mbyte
8	réservé
7	Enable du ready de la RAM
6	Enable du RMW read-modify-write cycles
5 : 4	Largeur de la RAM, "00"=8, "01"=16, "1X"=32
3 : 2	Nbre de wait-states du bus de la RAM en écriture : "00"=0, "01"=1, "10"=2, "11"=3
1 : 0	Nbre de wait-states du bus de la RAM en lecture : "00"=0, "01"=1, "10"=2, "11"=3

Le registre MCFG3 contient la valeur de recharge du refresh de la SDRAM.

Bits	Description
31 : 27	réservés
26 : 12	Valeur de recharge du refresh de la SDRAM
11 : 0	réservés

Cette valeur doit être calculée ainsi :

$$t_{REFRESH} = \frac{reload\_value + 1}{SYSCLK}$$

3.7 UART APB

Cet UART fournit une communication série qui permet d'apporter
- une entrée et sortie utilisateur sur une console du PC pour les programmes,
- permet de télécharger des données (fichier musical)
- procure également un outil de debug supplémentaire en fournissant par exemple les valeurs des variables à l'écran.

L'UART contient un décompteur de 12 bits pour générer le baud-rate désiré. Le décompteur fonctionne à la vitesse du clock et génère un « tick » à chaque fois qu'il y a un débordement. Il est ensuite rechargé avec la valeur du registre correspondant. La fréquence du « tick » doit être huit fois supérieure à la valeur du baud-rate désiré.

Deux FIFO stockent les données des transferts. Celles-ci peuvent avoir une taille de 1 à 32 * 2bytes.

Register	APB address offset
Data register	0x0
Status register	0x4
Control register	0x8
Scaler register	0xC

Le registre des données est soit la valeur reçu en cas de lecture, soit la valeur envoyée en écriture. Seuls les huit LSB sont actifs.

Le tableau suivant décrit les différents bits du registre de Status :

Bits	Description
31 : 26	Nbre de données dans la FIFO du récepteur
25 : 20	Nbre de données dans la FIFO du transmetteur
19 : 11	réservés
10	Indique que la FIFO du récepteur est pleine
9	Indique que la FIFO du transmetteur est pleine
8	Indique que la FIFO du récepteur est au moins à moitié pleine
7	Indique que la FIFO du transmetteur est moins qu'à moitié pleine
6	Indique qu'une erreur a été détectée dans une frame
5	Indique une erreur de parité
4	Indique qu'au moins un caractère a été perdu à cause d'un dépassement
3	Indique qu'un BREAK a été reçu
2	Indique que la FIFO du transmetteur est vide
1	Indique que le registre à décalage du transmetteur est vide
0	Indique qu'une nouvelle donnée a été reçue

Le tableau suivant décrit les différents bits du registre de Control :

Bits	Description
31 : 11	réservés
10	Enable de l'interruption de la FIFO du récepteur
9	Enable de l'interruption de la FIFO du transmetteur
8	En mettant ce bit à 1, le décompteur utilisera un clock externe au lieu du normal
7	Enable du loop-back mode
6	Enable du flow control
5	Enable de la parité
4	Sélection de la parité, 0 : pair, 1 : impair
3	Si ce bit est à 1, une interruption sera générée quand une frame est transmise
2	Si ce bit est à 1, une interruption sera générée quand une frame est reçue
1	Enable du transmetteur
0	Enable du récepteur

Le tableau suivant décrit les différents bits du registre du décompteur :

Bits	Description
31 : 12	réservés
11 : 0	Valeur de recharge

3.8 Emetteur SPDIF

L'émetteur SPDIF est le slave Wishbone qui a été choisi d'utiliser pour rendre le projet un peu plus ludique et concret.
SPDIF (EC-958) est un acronyme de Sony/Philips Digital InterFace (Interface numérique Sony/Philips). L'interface SPDIF est donc un standard numérique international et qui a été conçu par les sociétés Sony et Philips. Le SPDIF permet de transférer des données audio numériques. L'intérêt principal du SPDIF réside dans sa capacité à permettre, en numérique, le transfert du son entre deux équipements audio numériques sans perte de qualité puisqu'il évite une double conversion DA/AD. De plus, le transfert analogique peut impliquer une altération du signal d'origine (souffle, rapport signal/bruit dégradé, atténuation, distorsion).
Les fréquences d'échantillonnage autorisées sont : 44.1kHz (CD), 48kHz (DAT, Digital Audio Tape), 32kHz (DSR). Le nombre de bits par échantillons peut aller de 16 à 24 bits.

L'utilisation de l'émetteur SPDIF se fait en remplissant le « sample buffer », qui se trouve dans la moitié supérieure de la plage d'adresse de l'émetteur, et en mettant à jour ses registres. Un UserData buffer et un ChannelStatus buffer peuvent être activés à l'aide des génériques de l'émetteur. En plus de la sortie SPDIF, une sortie d'interruption est disponible.

Le tableau suivant liste les registres et buffers optionnels de l'émetteur SPDIF :

Register	APB address offset
TxVersion	0x00
TxConfig	0x01
TxChStat	0x02
TxIntMask	0x03
TxIntStat	0x04
UserData	0x20 - 0x37
ChStatus	0x40 - 0x57

Le registre TxVersion permet de lire les valeurs des génériques de l'émetteur et sa version (= 1).

Le registre TxConfig permet, comme son nom l'indique, de configurer l'émetteur. Le tableau suivant liste les variables de ce registre :

Bits	Nom	Description
31 : 24		Pas utilisé
23 : 20	MODE	Format des échantillons : 0 : 16bits, 1 : 17bits, …, 8 : 24 bits, 9-15 : pas utilisé
19 : 16		Pas utilisé
15 : 8	RATIO	Diviseur de clock pour générer la fréquence de transmission, détaillé plus loin
7 : 6	UDATEN	0 : User Data A&B mis à 0, 1 : User Data A&B généré depuis les bits 7-0 du User Data, 2 : User Data A généré depuis les bits 7-0 du User Data et B généré depuis les bits 15-8 du User Data, 3 : réservé
5 : 4	CHSTEN	0 : Channel Status A&B généré depuis TxChStat, 1 : Channel Status généré depuis les bits 7-0 du ChStat, 2 : Channel Status A généré depuis les bits 7-0 de ChStat et B généré depuis les bits 15-8 de ChStat, 3 : réservé
3		Pas utilisé
2	TINTEN	Enable de la sortie d'interruption
1	TXDATA	Si ce bit est à 0, les données transmises seront que des zéros
0	TXEN	Enable de l'émetteur

Le taux de transmission de données du signal SPDIF est une fonction du clock du bus Wishbone et des bits RATIO. Le taux de transmission est 64 fois la fréquence d'échantillonnage, chaque échantillon est codé en 32 bits et il y a 2 canaux. La fréquence d'échantillonnage est donnée par l'équation suivante :

$$S_{freq} = \frac{Wishbone\_clock\_frequency}{128 \cdot (RATIO + 1)}$$

Exemple : La fréquence d'échantillonnage est 48kHz et le clock du Wishbone vaut 12.288MHz. La valeur RATIO doit être mise à 1. Le taux de transmission sera 3.072 Mbps.

Le registre TxChStat définit les caractéristiques des données envoyées sur le SPDIF. Le tableau suivant liste les variables de ce registre :

Bits	Nom	Description
31 : 8		Pas utilisé
7 : 6	FREQ	00 : 44.1kHz, 01 : 48kHz, 10 : 32kHz, 11 : convertisseur de taux d'échantillons
5 : 4		Pas utilisé
3	GSTAT	Statuts de génération, 0 : pas d'indication, 1 : Original / commercial
2	PREEM	Pre-emphasis, 0 : aucun, 1 : 50/15 us
1	COPY	Copyright, 0 : copie interdite, 1 : copies permises
0	AUDIO	Format de donnée, 0 : audio, 1 : autre

Le registre TxIntMask contient les enables des différentes sources d'interruption.

Le tableau suivant liste les variables de ce registre :

Bits	Nom	Description
31 : 5		Pas utilisé
4	HCSBF	La moitié supérieure du channel status ou du User Data buffer est vide
3	LCSBF	La moitié inférieure du channel status ou du User Data buffer est vide
2	HSBF	La moitié supérieure du sample buffer est vide
1	LSBF	La moitié inférieure du sample buffer est vide
0		Pas utilisé

Le registre TxIntStat contient les flags des interruptions du TxIntMask. Si le bit correspondant dans le TxIntMask est à 1, la sortie d'interruption devient active. Le fait d'écrire un 1 sur un flag actif efface l'événement. Le signal d'interruption devient inactif lorsque tous les événements ont été effacés.
Le tableau suivant liste les variables du registre TxIntStat:

Bits	Nom	Description
31 : 5		Pas utilisé
4	HCSBF	La moitié supérieure du channel status ou du User Data buffer est vide
3	LCSBF	La moitié inférieure du channel status ou du User Data buffer est vide
2	HSBF	La moitié supérieure du sample buffer est vide
1	LSBF	La moitié inférieure du sample buffer est vide
0		Pas utilisé

3.9 Memory Map

La figure suivante représente une vue provisoire de la mémoire des bases du schéma AMBA ainsi posées :

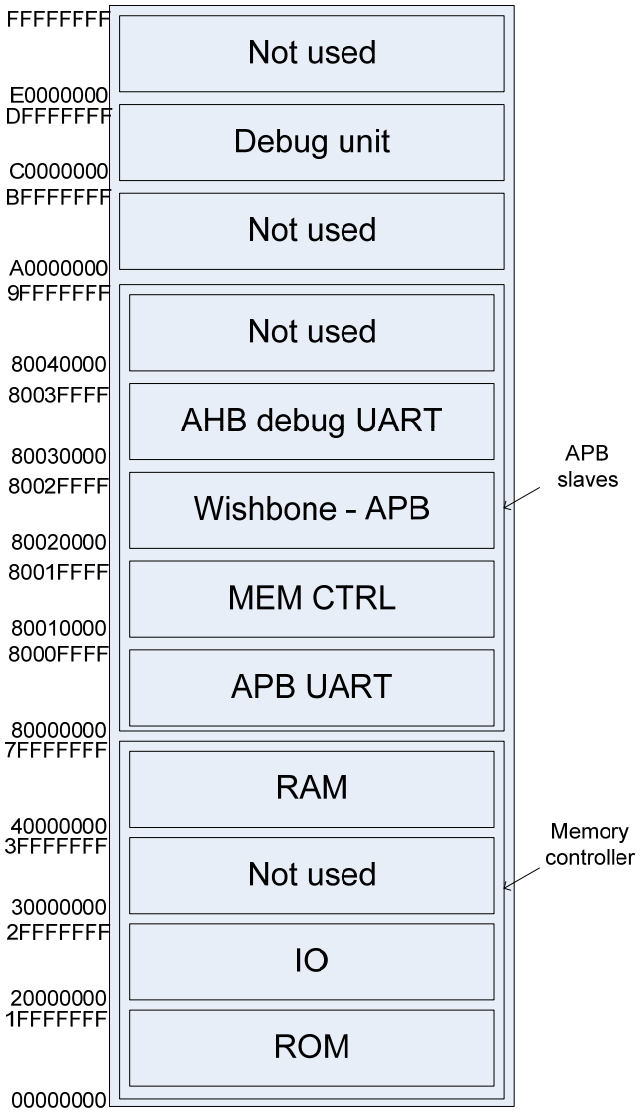


Figure 29 : Memory mapping du schéma de base

La ROM est placée à l'adresse 0x0, car c'est à cette adresse que le processeur ira chercher les premières instructions. Le contrôleur de mémoire se trouve donc aussi à l'adresse 0 jusqu'à une adresse arbitrairement choisie. La plage d'adresse de la RAM doit être suffisamment grande pour contenir aussi la SDRAM. Les périphériques APB n'ont pas énormément de place mémoire, mais ce ne sont souvent que quelques registres.

4. Création du pont AMBA – AHB / Wishbone

4.1 Introduction

Le développement du pont se fait avec le programme HDL Designer.
Le pont se comporte comme un slave AHB par rapport au bus AMBA et comme un master par rapport au bus Wishbone.

4.2 Comparaison des deux bus

Le pont doit être compatible avec tous les slaves Wishbone possible. Les slaves peuvent avoir différentes vitesse de réaction et de réponse. La figure suivante présente trois vitesses possibles :

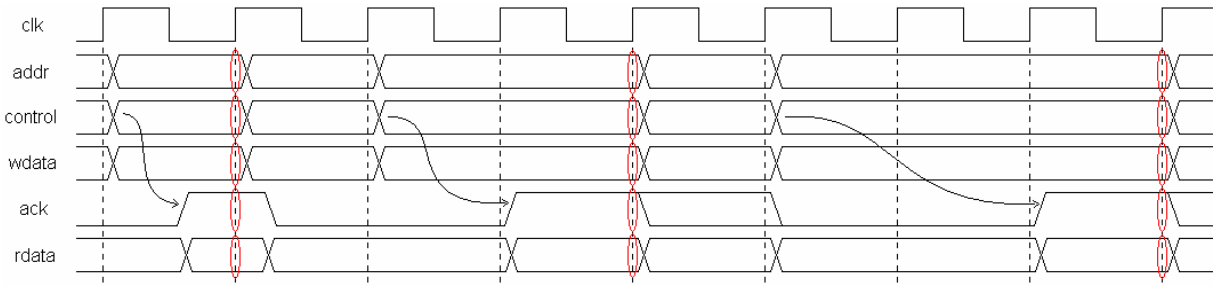


Figure 30 : Vitesses des slaves

Le premier transfert est rapide, le slave réagit avant qu'il y ait eu un flanc montant. Avec un tel slave, on peut donc effectuer un transfert à chaque coup d'horloge.
Le deuxième transfert est de vitesse moyenne, le slave met un clock pour placer le ack et un autre pour effectuer le transfert.
Le dernier transfert est lent, le slave met deux coups d'horloge pour placer le ack et un autre pour exécuter l'opération.

Afin de s'adapter à tous les slaves, le pont doit pouvoir placer des wait-states lorsqu'il est nécessaire.

La figure suivante présente la conversion du bus AMBA au bus Wishbone avec un transfert de vitesse moyenne (1 wait-state) :

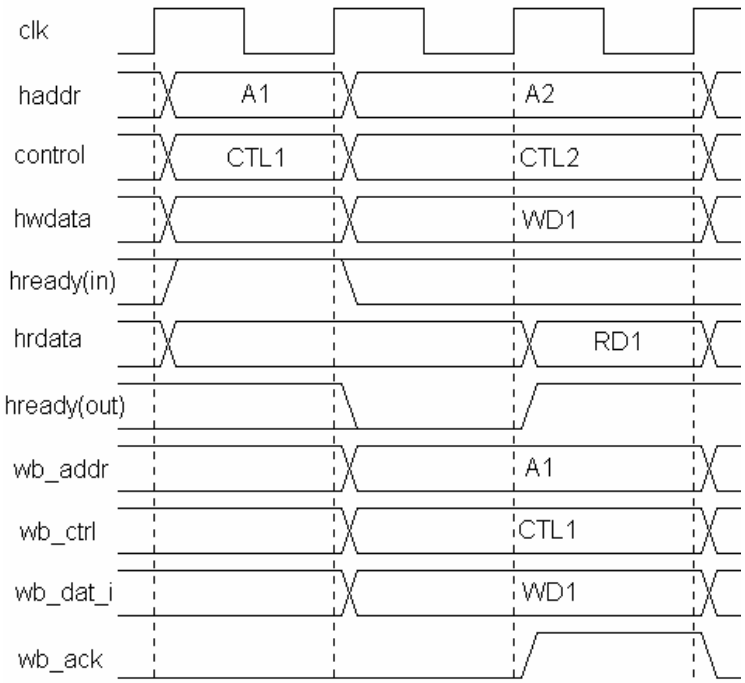


Figure 31 : Conversion AHB / Wishbone

Cette figure est très importante à comprendre, elle est l'objectif principal que le pont doit atteindre.
Le master commence le transfert après un premier flanc montant en posant les adresses et les signaux de contrôle (address phase). Ceux-ci seront stables lors du prochain flanc montant. Les données sont ensuite placées sur le bus (data phase).
Les adresses et les bits de contrôle sont mémorisés lors du flanc montant où ils sont stables. Ceux-ci sont ensuite dirigés vers le bus Wishbone. Les données sont directement dirigées vers le bus Wishbone et sont aussi mémorisées lors du flanc montant où elles sont stables.
Le périphérique Wishbone répond par un ack selon sa vitesse. Le pont doit ajouter des wait-states tant que le slave ne répond pas.
L'opération de lecture / écriture est exécutée lorsqu'un flanc montant apparaît avec un état haut sur le signal ack. Le bus est ensuite prêt à réaliser un nouveau transfert.

4.3 Solution choisie

4.3.1 Introduction

Il existe sûrement plusieurs manières de convertir le bus AMBA - AHB au bus Wishbone. Cette solution se veut élégante et fonctionnelle, mais elle n'est compatible avec le bus AMBA pur, car le signal de sélection, ajouté par les librairies de Gaisler, est une clé de son fonctionnement.
La conception se fait sur une architecture de bloc, et non pas en vhdl pur, pour une facilité de conception, mais aussi de compréhension du fonctionnement pour des usages et modifications futurs.

4.3.2 Généralités

Le bus Wishbone, contrairement au bus AMBA des librairies de Gaisler, est décomposé en chaque signal interne du bus. Ceci n'est pas esthétique et un type de bus Wishbone est donc crée :

```
type wb_i_type is record
  clk     : std_logic;
  rst     : std_logic;
  adr     : std_logic_vector(ADDR_WIDTH - 1 downto 0);
  dat     : std_logic_vector(DATA_WIDTH - 1 downto 0);
  sel     : std_logic_vector(15 downto 0);-- selection
  we      : std_logic;                     -- write enable
  bte     : std_logic_vector(1 downto 0);  -- burst type extension
  cti     : std_logic_vector(2 downto 0);  -- cycle type idenfier
  stb     : std_logic;                     -- strobe (valid data transfer)
  cyc     : std_logic;                     -- cycle (valid bus cycle)
  lock    : std_logic;                     -- transfer is uninterruptible
end record;

type wb_o_type is record
  dat     : std_logic_vector(31 downto 0); -- data
  ack     : std_logic;                      -- acknowledge
  err     : std_logic;                      -- error during last transfer
  rty     : std_logic;                      -- retry (slave not ready)
end record;
type wb_o_vector is array (15 downto 0) of wb_o_type;
```

Le signal de clock de la partie AMBA est le même que celui du bus Wishbone.
Le signal de reset du bus AMBA (actif bas) est inversé pour créer celui du bus Wishbone (actif haut).
Un signal sel est créé à partir de hsel du bus AHB :

```
sel <= hsel(NAHBSLV – 1 - WB_HINDEX);
```

L'inversion est due à la déclaration du signal `hsel : std_logic_vector(0 to NAHBSLV-1)`
La constant NAHBSLV est définie dans le fichier amba de la librairie grlib et peut être changé pour augmenter ou diminuer le nombre maximal de slave AHB. Le pont s'adapte ainsi aux changements.
Le signal sel indique donc quand le pont est sélectionné par un master.

4.3.3 Machine d'état

La machine d'état a pour objectif de :

- détecter un nouveau transfert et gérer certains signaux de contrôle du Wishbone en conséquence
- initialiser ces signaux de contrôle
- insérer des wait-states si nécessaire
- charger les données au bon moment

Les signaux de contrôles sont stb et cyc, les deux enable du Wishbone.

La machine d'état a les entrées suivantes :

- clock, reset
- sel
- hready, htrans

- ack

et les sorties :
- hready
- load_data (signal interne au pont qui indique quand les données peuvent être mémorisées)
- cyc, stb

La machine d'état est composée de trois états. Un état d'attente, un état où un transfert se réalise et un état d'initialisation.
La figure suivante est la machine d'état du pont :

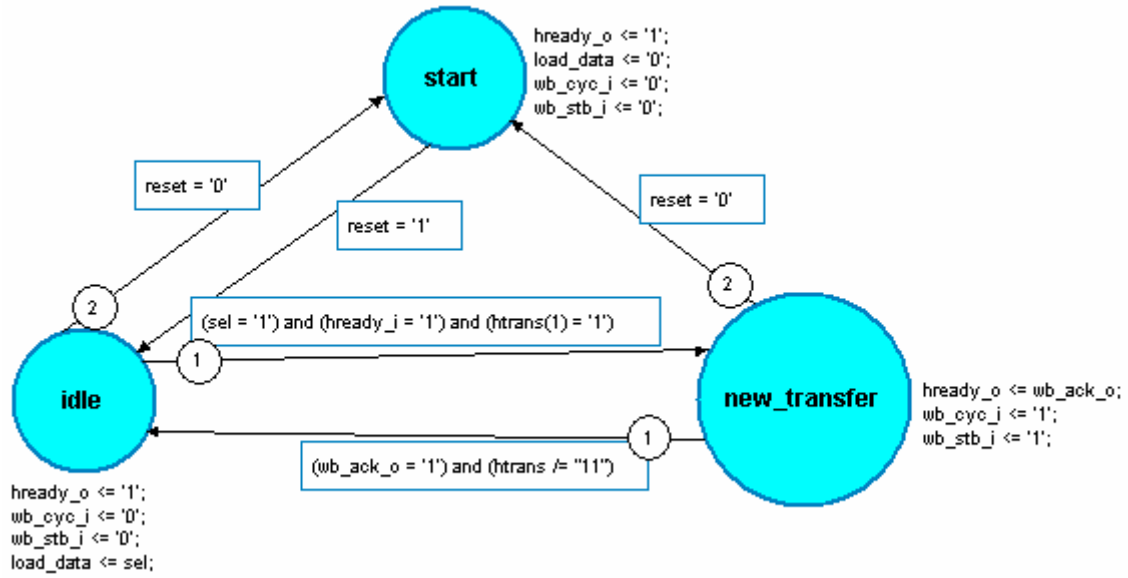


Figure 32 : Machine d'état

L'état initial est l'état « start » , il initialise tous les signaux que gère la machine d'état. Dès que le reset est terminé, la machine d'état bascule dans l'état « idle ». Les sorties sont alors :
- 0 pour hready (pas de wait-state)
- 0 pour cyc et stb (enable inactif)
- load_data vaut la même valeur que sel. En effet, dès que le pont est sélectionné, la valeur des données peut être mémorisée. Quand il y a un transfert, cette valeur ne change pas et donc reste à 1, car les données en entrée restent les mêmes.

Quand un transfert destiné à un slave Wishbone est détecté (sel = 1, hready = 1, htrans(1) = 1), la machine d'état entre dans l'état « new transfer ». Les sorties sont alors :
- hready prend la valeur de ack. Ainsi, quand il n'y a pas eu de ack lors d'un flanc montant du clock, hready vaudra 0 et crée donc un wait-state.
- cyc et stb valent 1. (enable actif)

A l'instant où le ack est reçu et htrans ne vaut pas 11 (pas de burst), la machine d'état retourne dans l'état « idle ».

4.3.4 Adresse et WE

Les adresses et le WE (write enable) ont un coup de clock pour être mémorisé, comme ils arrivent avant les données. La mémorisation se fait à l'aide de bascule D avec un signal load ou E (enable).

Ce signal est actif quand sel et hready valent 1. En effet, les adresses et le WE doivent être chargé quand le pont est sélectionné et quand il n'y a pas de wait-state. Par exemple, si le slave est rapide et qu'il n'y a pas de wait-state, les signaux sont mémorisés à chaque clock. S'il est lent, ils sont mémorisés juste quand le pont est sélectionné.
La figure suivante illustre ce fonctionnement :

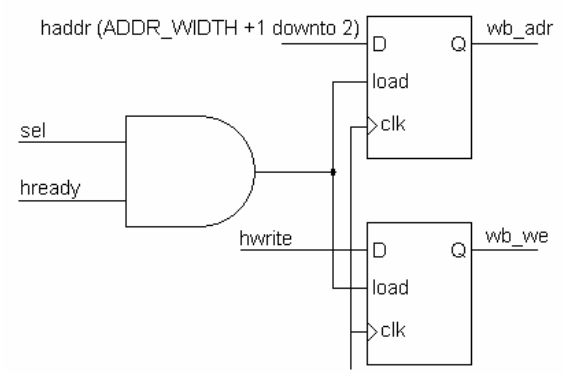


Figure 33 : Adresse et Write enable

4.3.5 Données

La machine d'état crée un signal load_data, qui indique quand les données peuvent être mémorisées. Cependant, pour un slave rapide, les données doivent être placées directement sur le bus Wishbone. Ainsi, lors du clock où les données sont placées par le bus AHB, c'est celles-ci qui doivent être mises sur le bus Wishbone et pour les clock suivants, c'est les données mémorisées qui doivent être sur le bus. Il faut donc un multiplexeur qui est commandé par le signal sel.
Lors du dernier transfert d'un burst, le pont doit envoyer, comme les autres transferts d'un burst, les données qui sont en entrées et pas encore mémorisées. Or, le signal sel n'est plus actif et dirigerait donc les données précédemment mémorisées sur le bus Wishbone. Il faut donc ajouter une condition au signal de sélection.
La figure suivante illustre ce fonctionnement :

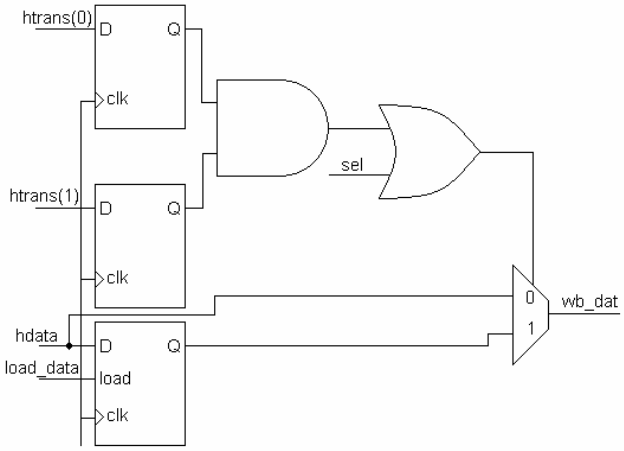


Figure 34 : Données

Les 2 bits du signal htrans indiquent qu'il s'agit d'un burst. Le fait de les retarder d'un clock avec des bascules D et de vérifier que les 2 bits sont à 1 permet de connaître les transferts de burst et le dernier y compris.

4.3.6 Sélection du slave Wishbone

Le pont doit savoir à quel slave est destiné un transfert pour rendre actif le bon signal de sélection.
La figure suivante présente la situation du masque des slaves Wishbone :

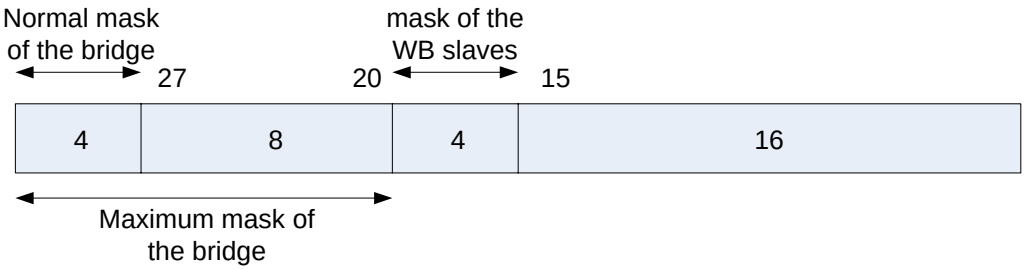


Figure 35 : Masque des slaves AHB

Les bits 19 à 16 des adresses indiquent donc quel slave doit être sélectionné. Et la taille de la plage d'adresse d'un slave est donc de 2^16 bytes, soit 64kBytes.
Un tableau en VHDL déclare les différentes adresses de chaque slave Wishbone.

```
CONSTANT addr_WB_Slave0 : std_logic_vector(3 downto 0) := "0000";
CONSTANT addr_WB_Slave1 : std_logic_vector(3 downto 0) := "0001";
CONSTANT addr_WB_Slave2 : std_logic_vector(3 downto 0) := "0010";
CONSTANT addr_WB_Slave3 : std_logic_vector(3 downto 0) := "0011";
…
CONSTANT addr_WB_Slave15 : std_logic_vector(3 downto 0) := "1111";

type addr_WB_SlaveVector is array (0 to 15) of std_logic_vector (3 downto 0);
CONSTANT tabAddr_WB_Vector : addr_WB_SlaveVector := (
0 => addr_WB_Slave0,
1 => addr_WB_Slave1,
2 => addr_WB_Slave2,
3 => addr_WB_Slave3,
…
15 => addr_WB_Slave15 );
```

Un processus vérifie à chaque changement des adresses et de sel à quel slave correspond la nouvelle adresse.

```
sele: process(sel, haddr)
begin
   for i in 0 to NAHBSLV - 1 loop
      if (sel = '1') and (haddr(19 downto 16) = tabAddr_WB_Vector(i)) then
         wb_sel(i) <= '1';
      else
         wb_sel(i) <= '0';
      end if;
   end loop;
end process sele;
```

Le processus vérifie pour chaque ligne du tableau si elle est identique aux bits 19 à 16 des adresses. Le signal sel doit être actif pour avoir un 1 en sortie.

Le signal de sortie du processus est ensuite mémorisé grâce à une bascule D commandée par sel.

4.3.7 Signaux de burst

Le pont doit créer les deux signaux de burst Wishbone cti et bte à partir des signaux AMBA htrans et hburst.
Le bus AMBA ne possède pas de burst à une adresse constante. Donc lorsqu'il y a un burst, la valeur de cti sera soit 2, soit 7. Selon la règle 4.30 de la spécification Wishbone, un master doit placer la valeur End-of-burst (7) lors du dernier transfert du burst courant. Or, cela n'est malheureusement pas possible à réaliser pour le pont, car le bus AMBA n'indique pas le dernier transfert d'un burst. Il serait possible de compter le nombre de transfert réalisé et déduire le dernier transfert, mais le bus AMBA a la particularité de faire des bursts de longueur indéfinie et c'est ce type de burst qui est le plus fréquemment utilisé.
La valeur de cti est donc soit 0 pour un transfert simple, soit 2 pour un burst :

```
burst: process(hburst, htrans)
begin
  if (htrans(1) = '1') and (hburst /= "000") then
    cti <= "010";
  else
    cti <= "000";
  end if;
end process burst;
```

Le signal bte fournit des informations supplémentaires sur le burst. Toutes ces informations se retrouvent dans hburst.

```
burst1: process(hburst)
begin
 case hburst is
  when "010" => bte <= "01";
  when "100" => bte <= "10";
  when "110" => bte <= "11";
  when others => bte <= "00";
 end case;
end process burst1;
```

Les deux signaux sont ensuite mémorisés à l'aide d'une bascule D. En fonction de sel, ce sont soit les signaux créé par les processus, soit les signaux mémorisés qui sont placé sur le bus Wishbone.

4.3.8 Signaux de configuration

Les signaux de configuration hconfig sont des constantes à déclarer dans un fichier vhdl. Un développeur qui utilise le pont doit changer les valeurs des constantes en fonction de son design.

```
CONSTANT WB_VENDORID : std_logic_vector(7 downto 0) := "00001000";
CONSTANT WB_DEVICEID : std_logic_vector(11 downto 0) := "000000000101";
CONSTANT WB_VERSION : std_logic_vector(4 downto 0) := "00001";
CONSTANT WB_IRQ : std_logic_vector(4 downto 0) := "00000";
CONSTANT WB_TYPE : std_logic_vector(3 downto 0) := "0010";
CONSTANT WB_HINDEX : integer := 10;
CONSTANT WB_HADDR : std_logic_vector(11 downto 0):=
                                std_logic_vector(TO_UNSIGNED(16#A00#, 12));
CONSTANT WB_HMASK : std_logic_vector(11 downto 0) :=
                                std_logic_vector(TO_UNSIGNED(16#E00#, 12));

CONSTANT WB_HCONFIG0 : amba_config_word := WB_VENDORID & WB_DEVICEID & "00"
& WB_VERSION & WB_IRQ;
CONSTANT WB_HCONFIG4 : amba_config_word := WB_HADDR & "0000" & WB_HMASK &
WB_TYPE;
```

Les constantes WB_HCONFIG0 et WB_HCONFIG4 sont ensuite placées à leur index respectif des signaux hconfig. Les autres index sont remplis avec des 0.

4.3.9 Multiplexage des slaves

Le pont se veut flexible, il doit donc pouvoir gérer plusieurs slaves en même temps tout en retournant les informations d'un slave à la fois au master. Les sorties des slaves doivent donc être multiplexées. Cela est fait avec le code vhdl suivant :

```
sel_slave: process(wb_sel_i, wb_o)
begin
 case wb_sel_i is
  when "0000000000000001" =>  wb_dat_o <= wb_o(0).dat;
                              wb_ack_o <= wb_o(0).ack;
                              wb_err_o <= wb_o(0).err;
                              wb_rty_o <= wb_o(0).rty;
  when "0000000000000010" =>  wb_dat_o <= wb_o(1).dat;
                              wb_ack_o <= wb_o(1).ack;
                              wb_err_o <= wb_o(1).err;
                              wb_rty_o <= wb_o(1).rty;
  …
  when "1000000000000000" =>  wb_dat_o <= wb_o(15).dat;
                              wb_ack_o <= wb_o(15).ack;
                              wb_err_o <= wb_o(15).err;
                              wb_rty_o <= wb_o(15).rty;
  when others =>              wb_dat_o <= (others => '0');
                              wb_ack_o <= '0';
                              wb_err_o <= '0';
                              wb_rty_o <= '0';
 end case;
end process sel_slave;
```

4.3.10 Signaux de retour

En plus des signaux de configuration (hconfig), des données et du signal de wait-state, le bus AHB demande le signal hresp qui indique comment s'est passé le transfert pour le slave (OK, error, retry, split). Ceci est fait avec les signaux err (error) et rty (retry) fournis par le bus Wishbone.

```
ahbso.hready <= hready_o;

ahbso.hresp(0) <= wb_err_o;
ahbso.hresp(1) <= wb_rty_o;

ahbso.hrdata <= wb_dat_o;

ahbso.hsplit <= (others => '0');

ahbso.hcache <= '0';

ahbso.hirq <= (others => '0');

ahbso.hconfig(0) <= WB_HCONFIG0;
ahbso.hconfig(1) <= (others => '0');
ahbso.hconfig(2) <= (others => '0');
ahbso.hconfig(3) <= (others => '0');
ahbso.hconfig(4) <= WB_HCONFIG4;
ahbso.hconfig(5) <= (others => '0');
ahbso.hconfig(6) <= (others => '0');
ahbso.hconfig(7) <= (others => '0');

ahbso.hindex <= WB_HINDEX;
```

4.4 Utilisation du pont

Les étapes suivantes décrivent les différentes phases nécessaires pour inclure le pont dans un projet :

- inclure la librairie dans le projet.
- Placer le composant dans un schéma ou l'utiliser dans un code vhdl
- Changer les variables du fichier config en fonction du design
- Connecter le pont au bus AHB et les slaves Wishbone au pont, chacun doit avoir son signal de sélection et son canal de sortie.
- Si un slave Wishbone a une sortie d'interruption, il faut la router manuellement jusqu'à un bit du vecteur hirq du port ahbso.

5. Pont AMBA – APB / Wishbone

5.1 Introduction

Le pont APB / Wishbone a été fait en grande partie durant le projet de semestre, mais il est intéressant de le revisiter et améliorer son esthétique.
La figure suivante présente la vue la plus simple pour l'utilisation du pont APB / Wishbone :

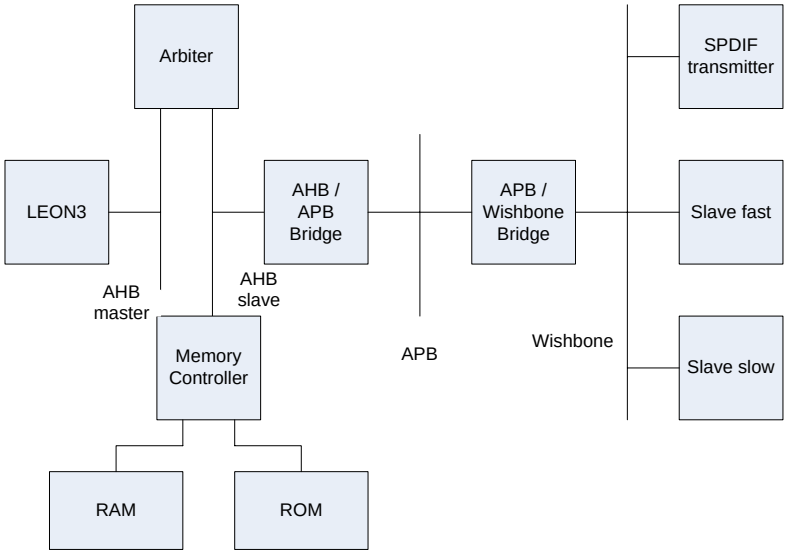


Figure 36 : Pont AMBA – APB / Wishbone

5.2 Comparaison des deux bus et Solution choisie

Un slave APB ne peut pas indiquer au master qu'il n'est pas prêt ou qu'il est lent et qu'il lui faut plus de temps pour un transfert. Une écriture ou lecture sur le pont APB a donc un nombre de coup de clock fixe.

De ce fait, les adresses, données, clock, enable sont connecté directement du bus APB au Wishbone. Le reset est quand à lui inversé comme pour le pont AHB / Wishbone.

Un signal sel est créé de la même manière que pour le pont AHB / Wishbone :

```
sel <= psel(WB_PINDEX);
```

Le bus APB ne peut pas faire de transfert de burst, les signaux Wishbone de burst cti et bte sont donc mis à 0.

La figure suivante présente la vue d'un transfert du bus AHB au bus Wishbone en passant par le APB :

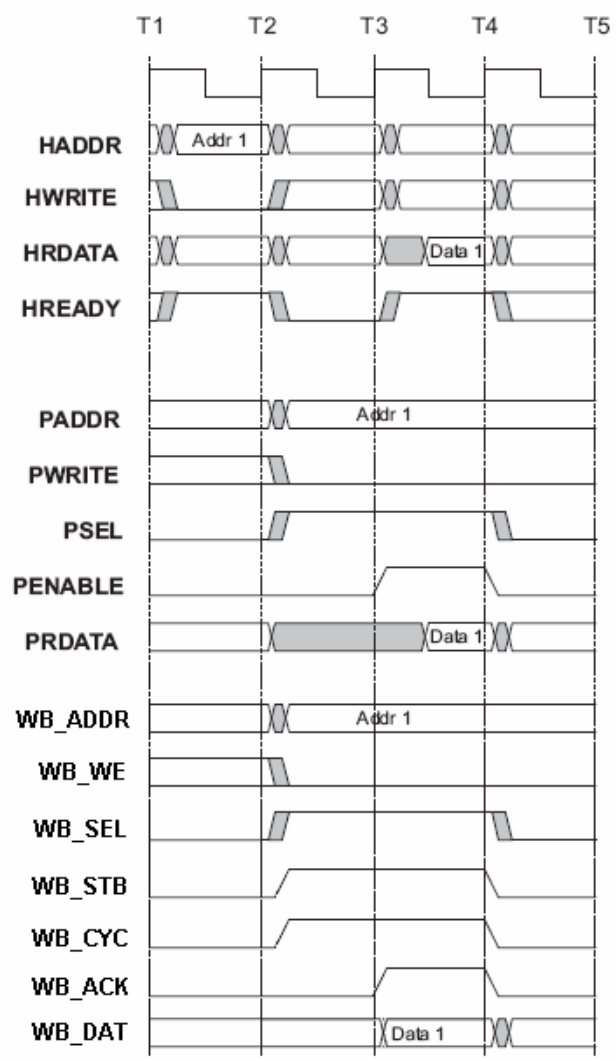


Figure 37 : Transfert de AHB – APB – Wishbone

On remarque que les signaux stb et cyc doivent apparaître un coup d'horloge avant le signal enable du bus APB. Cela se fait avec sel et le signal pwrite inversé dans une porte ET. Le résultat avec le penable dans une porte OU.

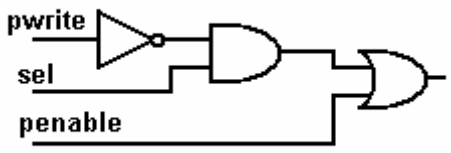


Figure 38 : Enable du pont APB / Wishbone

La sélection du slave Wishbone se fait de la même manière que pour le pont AHB / Wishbone, il utilise le même vecteur d'adresses des slaves, mais ils sont comparés avec les bits 15 à 12 de paddr.

```
sele: process(sel, paddr)
begin
   for i in 0 to NAPBSLV - 1 loop
      if (sel = '1') and (paddr(15 downto 12) = tabAddr_WB_Vector(i)) then
         wb_sel_i(i) <= '1';
      else
         wb_sel_i(i) <= '0';
      end if;
   end loop;
end process sele;
```

Le multiplexage des slaves Wishbone se fait exactement de la même manière que pour le pont AHB / Wishbone.

Les constantes VendorID, DeviceID, … sont les mêmes que pour le pont AHB / Wishbone, seuls les constantes PADDR et PMASK sont ajoutées. Les constantes WB_PCONFIG0 et WB_PCONFIG1 sont créés à partir de toutes ces constantes et sont placée dans leurs signaux de configuration pconfig respectifs. Un développeur qui utilise le pont doit changer les valeurs des constantes en fonction de son design.

```
CONSTANT WB_PADDR : std_logic_vector(11 downto 0) :=
                                 std_logic_vector(TO_UNSIGNED(16#200#, 12));
CONSTANT WB_PMASK : std_logic_vector(11 downto 0) :=
                                 std_logic_vector(TO_UNSIGNED(16#F00#, 12));

CONSTANT WB_PCONFIG0 : amba_config_word := WB_VENDORID & WB_DEVICEID & "00"
                                                    & WB_VERSION & WB_IRQ;
CONSTANT WB_PCONFIG1 : amba_config_word := WB_PADDR & "0000" & WB_PMASK &
                                                                  "0001";
```

```
apbo.pconfig(0) <= WB_PCONFIG0;
apbo.pconfig(1) <= WB_PCONFIG1;
```

La procédure d'utilisation de ce pont est la même que celle du pont AHB / Wishbone.

6. Implémentation

6.1 Introduction

Le but de l'implémentation est de vérifier que le circuit réalisé et simulé fonctionne réellement sur une carte, mais aussi d'illustrer le projet lors de la journée porte ouverte.
A la fin de l'implémentation, il sera donc possible d'écouter un fichier musical grâce à la sortie SPDIF.

6.2 Programmation de la FPGA

Le premier objectif consiste à programmer la FPGA avec le circuit de base. L'annexe C en décrit la procédure.

La carte Suzaku a d'abord été choisie pour son originalité.

Caractéristiques de la carte :

FPGA	Xilinx Spartan-3 (XC3S400 FT256)
Soft Processor	MicroBlaze
Crystal Oscillator	3.6864MHz (frequency multiplied by FPGA's internal DCM)
Memory	BRAM 8Kbyte FLASH Memory 4Mbyte SDRAM 16Mbyte
Configuration	Stored in FLASH Memory, Controller TE7720
JTAG	2 ports (FPGA, TE7720)
Ethernet	10Base-T/100Base-Tx
Serial	UART 115.2kbps
Timer	2ch (1ch for OS)
Free I/O Pin	86-pin
Reset Function	Software Reset
Power Supply	Voltage: 3.3V±3% Consumption current: 350mA typ (while processor is operating)
Dimensions	72×47mm

Les mémoires n'ayant que 16 bits de données alors qu'il en avait été prévu 32 sur le circuit, il a donc fallu adapter le bus de données. L'annexe D présente comment passer de 32 à 16 bits de données.
La carte a 2 connexions JTAG. Une pour la programmation de la FPGA et une pour accéder aux mémoires.
Il n'a malheureusement pas été possible de charger le circuit sur la FPGA, même en passant par les mémoires, pour une raison inconnue.
Par contre, un schéma contenant uniquement un inverseur a été mis sur la FPGA. Le fonctionnement a été vérifié grâce à une carte branchée à celle de la FPGA possédant des interrupteurs et des LED.

Après une semaine, il a été décidé de passer à la carte « FPGA BOARD ». C'est une carte de teste pour FPGA avec de nombreuses mémoires, entrées / sorties de toutes sortes et une FPGA de grande capacité.

FPGA	Xilinx Virtex-2 (XC2V3000 BG728 -6)
Crystal Oscillator	6 differents clock (25MHz, extern clock and not mounted clocks)
Memory	SRAM 2 Mbyte EEPROM 512 Kbyte FLASH Memory 256 Mbyte SDRAM 64 Mbyte
Configuration	Stored in FLASH Memory, Controller TE7720
JTAG	3 ports
Ethernet	10Base-T/100Base-Tx
Serial	2 ports RS422
Free I/O Pin	64-pin + mezzanine pins
Reset Function	Software Reset
Power Supply	Voltage: 5V Consumption current: 350mA typ (while processor is operating)
Dimensions	72×47mm

Le chargement a été réussi dès la première tentative sur cette carte.

6.3 Téléchargement d'un fichier musical sur la SDRAM

Le port série à disposition n'a pas une vitesse suffisante pour lire en direct un fichier musical depuis le PC, il faut donc d'abord l'enregistrer sur une mémoire à disposition sur la carte. Le format de fichier qui peut être lu avec le décodeur SPDIF est le WAV. Ce format qui n'a pas de compression est rapidement volumineux. Le choix se limite donc à la mémoire FLASH et la SDRAM. La FLASH étant relativement compliquée d'utilisation et le contrôleur de mémoire présent dans le circuit de la FPGA ayant un contrôleur de SDRAM intégré, le choix se porte donc sur la SDRAM.

Pour charger des données sur la SDRAM, le moyen le plus simple consiste à communiquer avec le DSU du circuit via l'UART AHB et un programme depuis le PC. Grmon est un programme qui peut être téléchargé sur le site de Gaisler Research et sa version de démonstration fonctionne pendant 21 jours. Il permet de débugger un programme tournant sur un processeur LEON et des circuits basé sur les librairies grlib.
L'annexe E décrit les commandes principales du Grmon.

Programmation

Pour programmer la SDRAM, il faut donc développer un programme en langage C qui doit configurer le contrôleur de mémoire pour qu'il communique correctement avec la SDRAM (registres MCFG2 et MCFG3), télécharger les données depuis l'UART APB et écrire sur la SDRAM.

Contrôleur de mémoire

Le tableau suivant présente les valeurs données pour le registre MCFG2 du contrôleur de mémoire.

Bits	Description	Valeur
31	Enable du refresh de la SDRAM	1
30	Timing Trp, 0 : 2 clock, 1 : 3 clock	0
29 : 27	Timing Trfg, 3 + valeur clock	0
26	Timing Tcas, 0 : 2 cycles CAS, 1 : 3 cycles CAS	1
25 : 23	Taille des banques SDRAM, “000”=4Mbyte, “001”=8 Mbyte, “010”=16 Mbyte.... "111”=512Mbyte	1
22 : 21	Taille des colonnes de la SDRAM, “00” = 256, “01” = 512, “10” = 1024, “11” = 4096 quand les bits 25:23 = ”111” sinon 2048	100
20 : 19	Génère une commande SDRAM, “01”=PRECHARGE, “10”=AUTO-REFRESH, “11”=LOAD-COMMAND-REGISTER	11
18	Donne la taille du bus SDRAM, 0 : 32 bits, 1 : 64 bits, lecture seulement	0
17 : 15	réservés	
14	Enable du SDCTRL	1
13	Disable de la SRAM	0
12 : 9	Taille d'une banque de la RAM, “0000”=8 kbyte, “0001”=16kbyte, ..., “1111”=256 Mbyte	1000
8	réservé	
7	Enable du ready de la RAM	0
6	Enable du RMW read-modify-write cycles	1
5 : 4	Largeur de la RAM, “00”=8, “01”=16, “1X”=32	10
3 : 2	Nbre de wait-states du bus de la RAM en écriture : “00”=0, “01”=1, “10”=2, “11”=3	00
1 : 0	Nbre de wait-states du bus de la RAM en lecture : “00”=0, “01”=1, “10”=2, “11”=3	00

La valeur du registre MCFG2 est ainsi 0x96385060. A chaque écriture sur la SDRAM, la commande LOAD-COMMAND-REGISTER doit être exécutée et donc cette valeur doit être à chaque fois réécrite.

La SDRAM de la carte a besoin d’un taux de refresh de 7.8us.

$$t_{REFRESH} = \frac{reload\_value + 1}{SYSCLK} \Rightarrow reload\_value = t_{REFRESH} \cdot SYSCLK - 1 = 7.8\mu s \cdot 25MHz - 1 = 194$$

La valeur du registre MCFG3 est donc 0xC2000.

UART

Comme les fichiers WAV sont volumineux, il est intéressant d’avoir un débit maximal sur le port série. Dans cette optique, le bit de parité n’est pas actif et le baud-rate est fixé à 115'200.

Aucune option n’est activée, le registre de contrôle n’active que l’émetteur et le récepteur et a donc la valeur de son Control Register est 0x3.

$$F_{tick} = 8 \cdot 115'200 = 921'600$$

$$reload\_value = \frac{25MHz}{921'600} = 27$$

Le registre du Scaler doit avoir la valeur 0x1B pour un débit de 115'200 baud.

Le bit 0 du registre de Status de l'UART indique qu'une valeur a été reçue. Ce bit doit donc être contrôlé quand un nouveau paquet est attendu.

Fichiers WAV

WAV (ou **WAVE**), une contraction de *WAVEform audio format*, est un standard pour stocker l'audio digitale. Le format WAV ne correspond à aucun format d'encodage spécifique, il s'agit d'un conteneur capable de recevoir des formats aussi variés que le MP3, le WMA, l'ATRAC3, l'ADPCM, le PCM. C'est ce dernier qui est cependant le plus courant, et c'est pour cela que l'extension .wav est souvent, et donc à tort, considérée comme correspondant à des fichiers "sans pertes".
Les fichiers WAV contiennent un header de 44 bytes. Le tableau suivant présente le contenu du header :

Nom du champ	Nbre de byte	Description
TAG1	4	Constante "RIFF" *(0x52,0x49,0x46,0x46)*
SIZE1	4	Taille du fichier moins 8 octets
FORMAT	4	Format = "WAVE" *(0x57,0x41,0x56,0x45)*
TAG2	4	Identifiant "fmt " *(0x66,0x6D,0x74,0x20)*
LGDEF	4	Nombre d'octets utilisés pour définir en détail le contenu
FORMAT	2	Format de fichier (1: PCM, ...)
NBCANAUX	2	Nombre de canaux (*1* pour mono ou *2* pour stéréo)
FREQ	4	Fréquence d'échantillonnage (en Hertz)
BYTEPERSEC	4	Nombre d'octets par seconde de musique
NBRBYTE	2	Nombre d'octets par échantillon
NBBITS	2	Nombre de bits par donnée
TAG3	4	Constante "data" *(0x64,0x61,0x74,0x61)*
SIZE2	4	Taille du fichier moins 44 octets

Le dernier champ donne la taille des données dans le header. Il est utile de le connaître pour savoir combien d'octets vont être transmis par l'UART.

Le programme initialise l'UART et le contrôleur de mémoire avec les valeurs vues précédemment. Lorsque le fichier est envoyé depuis le PC, il calcule la taille du fichier et l'enregistre à la première position de la SDRAM. Le programme attend ensuite chaque fois de recevoir 4 bytes pour les enregistrer en 32bits sur la SDRAM jusqu'à ce que le nombre de bytes reçus ait atteint celui de la taille.

Il est à noter que le fichier WAV est en little endian et que le protocole SPDIF fonctionne en big endian, les bytes doivent donc être inversés.

Le diagramme suivant décrit le fonctionnement de l'écriture du fichier WAV :

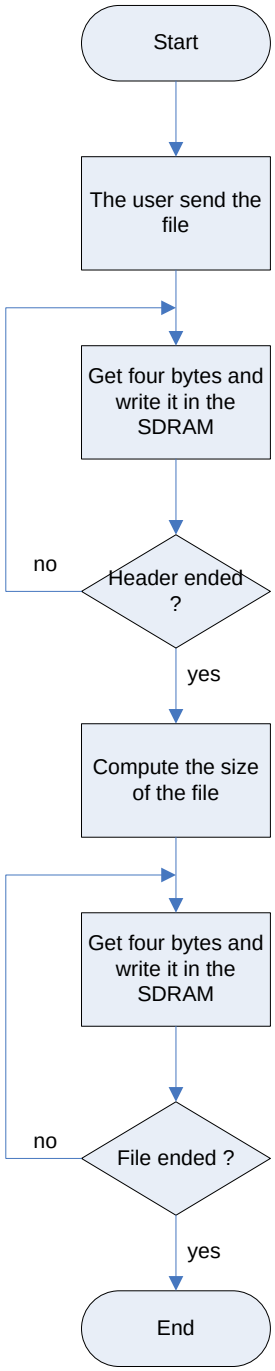


Figure 39 : Diagramme de l'écriture du fichier WAV

6.4 Lecture du fichier

Le fichier doit être lu sur la SDRAM pour être transmis sur l'émetteur SPDIF. Celui-ci doit être configuré en fonction des caractéristiques du fichier WAV (fréquence d'échantillonnage, nombre de bits par échantillons, …).
Deux émetteurs SPDIF sont à disposition dans le circuit, un du pont AHB / WB et un du pont APB / WB. Des slaves Wishbone de test ont été créé (voire le chapitre de test) possédant des registres dont un qui est en sortie. Le bit 0 du slave pilote un multiplexeur ayant les deux signaux SPDIF en entrée. L'écriture d'un 1 ou d'un 0 sur le slave permet donc de choisir quel signal SPIDF est en sortie.

L'émetteur SPDIF a un buffer de la taille de la moitié de sa plage d'adresse. Le buffer est divisé en deux : le buffer supérieur et inférieur. Avant de commencer à émettre, le buffer doit être rempli et configuré en fonction du fichier qui est lu. L'émetteur commence à lire le buffer inférieur, et quand il l'a terminé, un flag correspondant est mis à 1. Il continue à lire le buffer supérieur et pendant ce temps, le buffer inférieur peut être rempli de nouvelles données. Un autre flag est mis à 1 lorsque le buffer supérieur est fini d'être lu. L'émetteur recommence à lire le buffer inférieur et pendant ce temps, le buffer supérieur peut être réécrit. Et ainsi de suite jusqu'à ce que le fichier soit terminé.

La figure suivante illustre la zone mémoire de l'émetteur SPDIF :

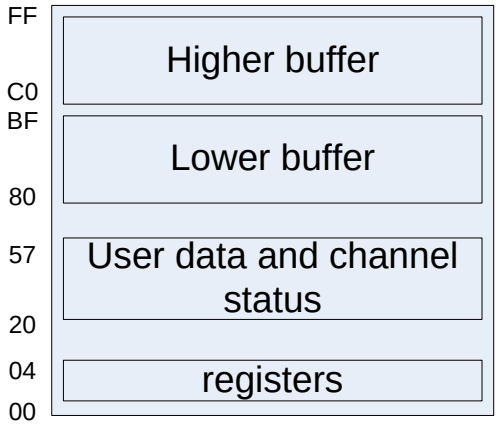


Figure 40 : Memory mapping de l'émetteur SPDIF

Le diagramme suivant présente le fonctionnement de la lecture du fichier wav :

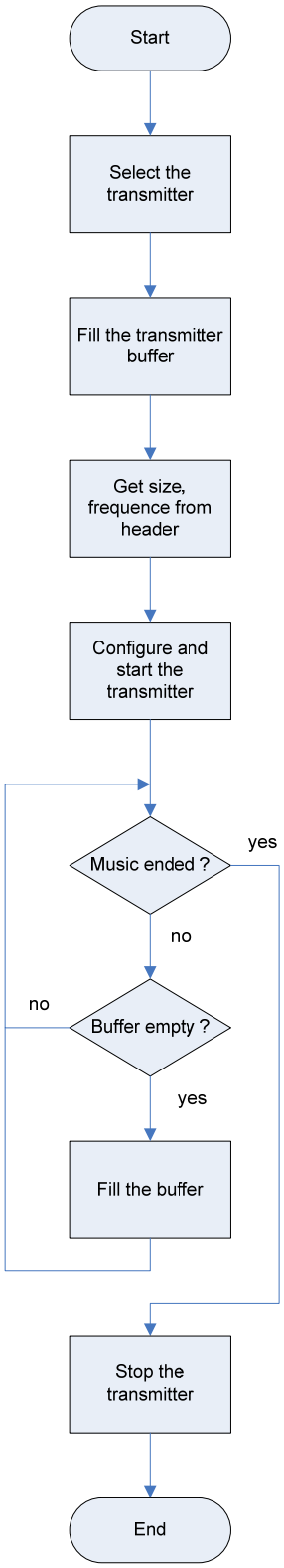


Figure 41 : Diagramme de la lecture du fichier WAV

Le programme en langage C doit être compilé en fichier exécutable pour le processeur LEON. Cela se fait avec un compilateur BCC (Bare-C Cross-Compiler System for LEON) pouvant être téléchargé depuis le site de Gaisler. Ce compilateur peut fonctionner sur Linux ou Windows avec le programme Cygwin.
La commande suivante permet d'obtenir un fichier exécutable pour le LEON :

```
sparc-elf-gcc -msoft-float -g -O2 nom_du_fichier.c -o nom_du_fichier.exe
```

La commande « load » du moniteur Grmon permet de charger le fichier en mémoire et « run » de l'exécuter.

6.5 Adaptateur SPDIF

Le signal de sortie SPDIF de l'émetteur est encore un signal TTL (0 – 3.3V). Or, l'entrée numérique du haut-parleur s'attend à recevoir un signal entre ± 0.5V et ± 1V. La création d'une carte d'adaptation est donc nécessaire.
La figure suivante montre le fonctionnement de l'adaptateur SPDIF, d'autres possibilités sont possible pour avoir le même résultat :

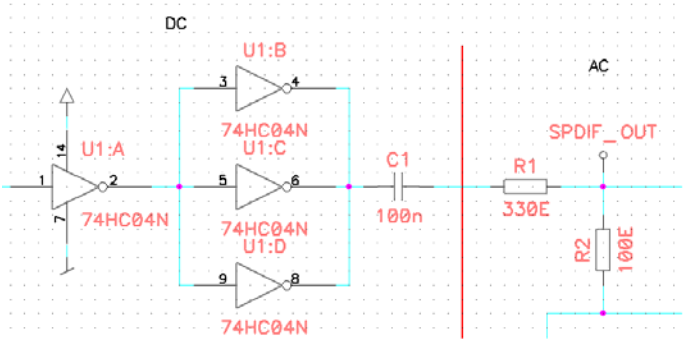


Figure 42 : Conversion TTL en SPDIF

Le but des deux inversions est de donner suffisamment de courant au haut-parleur, la deuxième inversion est donc faite avec 3 inverseurs en parallèle. Cela est suivit d'un condensateur qui enlève la composante continue du signal, il y aura donc un signal ± 2.5V. Puis un diviseur de tension réduit le niveau de tension pour atteindre ± 0.75V.
La carte crée contient également un circuit qui a la fonctionnalité inverse.

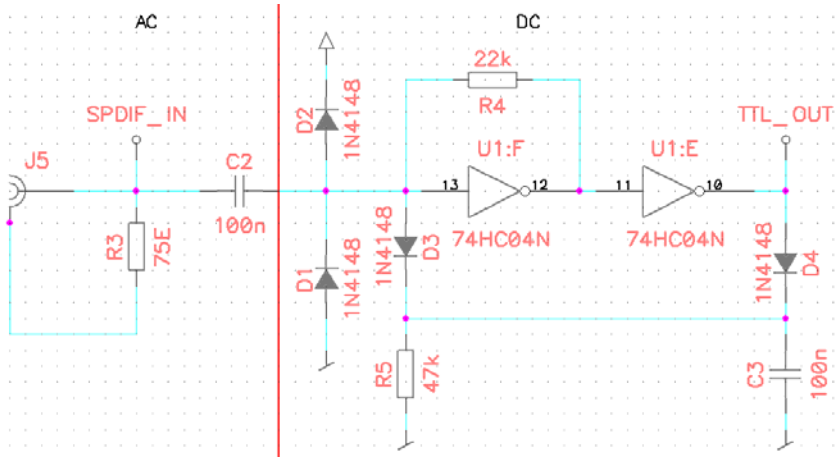


Figure 43 : Conversion SPDIF en TTL

Le circuit amplifie et ajoute une composante continue au signal SPDIF afin qu'il corresponde aux niveaux TTL.

Des pointes de tests permettent de mesurer facilement des signaux à l'oscilloscope. Deux signaux au choix peuvent ainsi être mesurés.
La carte a été tirée, percée, limée et soudée par mes soins à l'atelier électronique de l'école.

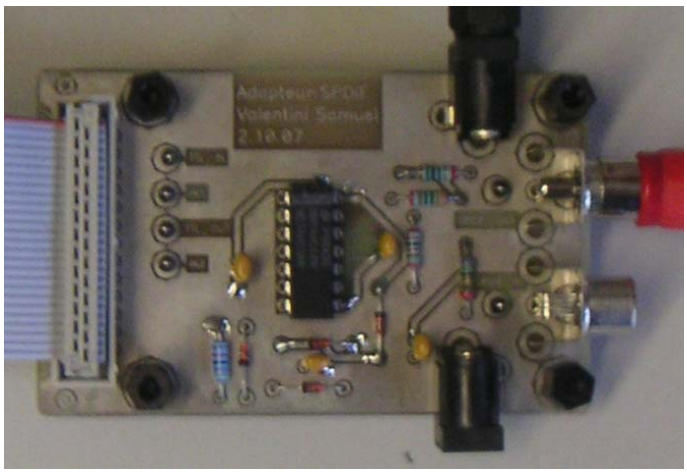


Figure 44 : Carte Adaptateur SPDIF

Les schémas PCAD schématique et PCB se trouvent en annexe F1 et F2.

7. Tests

7.1 Introduction

Les tests sont une étape importante de la création d'un circuit, puisqu'ils permettent de déceler les erreurs et aident souvent à la compréhension des parties les plus compliquées.
Une grande partie des tests consiste à simuler les différentes parties du système (ponts et schéma de base). Des tests peuvent aussi être faits avec le moniteur Grmon et ses fonctions wmem et mem. Le test final est l'écoute de musique sur les haut-parleurs.

La procédure de test se fait donc en commençant par le niveau le plus bas pour arriver jusqu'au système complet.

7.2 Simulations

7.2.1 Introduction

Une librairie de test doit être crée pour simuler le circuit.
Un banc de test créé les signaux de clock, de reset, et autres signaux d'entrées requis.
Les librairies de Gaisler fournissent des mémoires SRAM qui sont utilisées ici pour simuler la RAM et la ROM de la carte. Le composant SRAM possède 8 bits de données, il y en a donc 4 en parallèle afin d'avoir 32 bits de données.
La figure suivante présente la vue générale de la librairie de test :

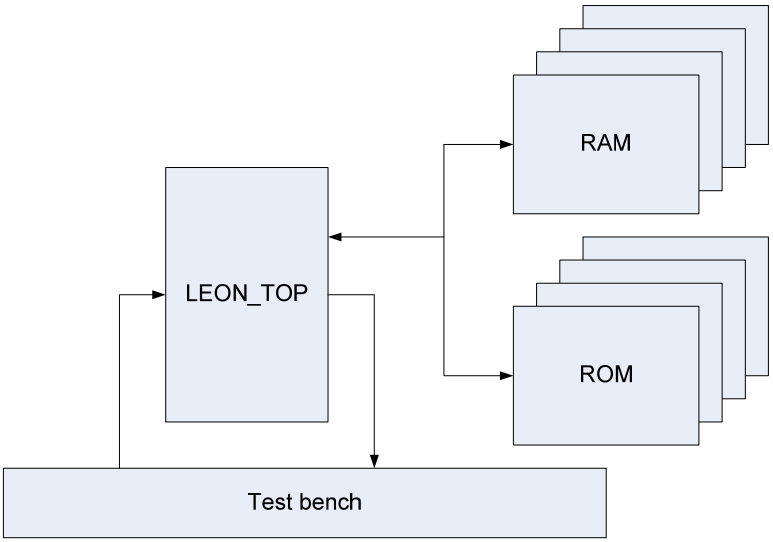


Figure 45 : Librairie de test

Le composant LEON_TOP étant évidemment le circuit à tester.

7.2.2 Connexion des RAM et ROM

Le contrôleur de mémoire fournit les bus memi et memo contenant toutes les entrées / sorties nécessaires aux mémoires.
Le composant SRAM a les entrées / sorties suivantes :

- a (adresses)
- ce1 (chip select)
- we (write enable)
- oe (output enable)
- d (data in et out)

Et il a les génériques suivants :

- index : numéro du byte dans le word (0, 1, 2 et 3)
- abits : nombre de bits d'adresse, défini aussi la grandeur de la mémoire
- tacc : temps d'accès en ns
- fname : fichier qui est chargé dans la mémoire

Une boucle for crée les quatre RAM. La variable d'itération distingue chaque SRAM avec le numéro d'index. Les constantes RAMS_DETH et ROMS_DETH déterminent la valeur de abits de la RAM respectivement la ROM.

Le tableau suivant décrit comment les entrées des RAM et des ROM sont connectées :

SRAM input	RAM	ROM
a	address(RAMS_DEPTH+1 downto 2)	address(ROMS_DEPTH+1 downto 2)
ce1	ramsn(0)	romsn(0)
we	rwen(0)	1
oe	ramoen(0)	oen

Chaque byte des données doit être assigné à un bloc de SRAM. La variable d'itération de la boucle for est utilisée à cet effet.
La figure suivante décrit comment sont assignés les bytes des données :

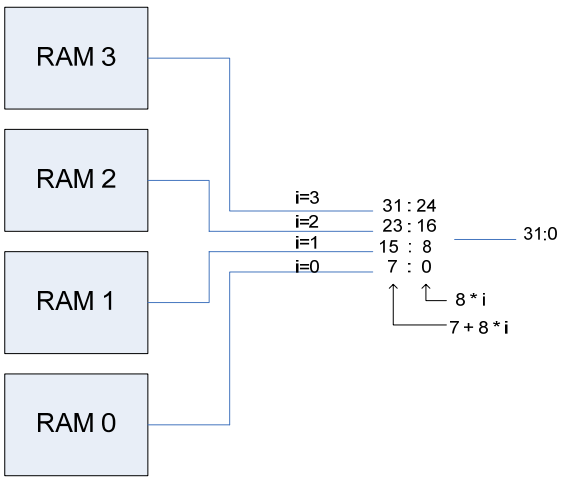


Figure 46 : Données des mémoires de la simulation

i représente la variable d'itération de la boucle for qui sont RAMS_INDEX et ROMS_INDEX. La sélection des bytes des données se fait donc avec le code vhdl ci-dessous :

```
data(7+8*i downto 8*i) : (31 downto 0)
```

7.2.3 Banc de test

Le banc de test sert à produire les signaux d'entrée nécessaires au fonctionnement du circuit. Dans le cas simple d'une simulation sans DSU, UART, … seuls le clock et le reset sont requis. Ils peuvent aussi être utilisé pour analyser les sorties pour en vérifier la fonctionnalité.
Les bancs de test sont écrits en vhdl et permettent l'utilisation de commandes spéciales, telles que des références au temps.
Le code vhdl suivant présente le banc de test utilisé :

```
ARCHITECTURE leon_tester OF leon_tester IS
      constant    clockFrequency    : real := 25.0E6;
      constant    clockPeriod       : time := (1.0/clockFrequency) * 1 sec;
      signal      sClock            : std_uLogic := '1';

BEGIN

      sClock <= not sClock after clockPeriod/2;
      clock <= transport sClock after clockPeriod*9/10;
      reset <= '0', '1' after 10*clockPeriod;

      data <= (others => 'Z');
      rwen <= (others => 'Z');

END ARCHITECTURE leon_tester;
```

Le signal de clock est décalé de 9/10 de période pour que ses flancs montants n'apparaissent pas au moment où d'autres signaux d'entrée changent.

7.2.4 Programmation

Un programme doit se trouver en mémoire ROM pour que le processeur puisse exécuter des transferts sur les ponts. Ce programme doit donc avoir des accès sur la plage mémoire dédiée aux ponts.
La programmation se fait en langage C et la compilation avec un compilateur BCC (Bare-C Cross-Compiler System for LEON) pouvant être téléchargé depuis le site de Gaisler. Ce compilateur peut fonctionner sur Linux ou Windows avec le programme Cygwin.

La SRAM utilisée pour simuler les mémoires doit recevoir un fichier de type SREC. Les fichiers SREC (Motorola S-records) est un format encodé en ASCII pour les données binaires. Le site suivant décrit dans les détails les particularités de ce format :

http://www.die.net/doc/linux/man/man5/srec.5.html

Le programme en C suivant écrit des valeurs dans la mémoire RAM :

```
#define RAMAREA (0x40000000)
int main(int argc, char *argv[])
{
      volatile unsigned char *ram_char=(volatile unsigned char *)RAMAREA;
      volatile unsigned int *ram_int=(volatile unsigned int *)RAMAREA;
      ram_char[1]=9;
      ram_int[4]=7;
      return 0;
}
```

La constante RAMAREA correspond à l'adresse de la RAM. Les variables ram_char et ram_int sont des pointeurs de tailles différentes sur la RAM. Des instructions font écrire ensuite dans la RAM à des endroits différents. Ce petit programme permet donc de vérifier le fonctionnement des blocs principaux (LEON3, arbitre, contrôleur de mémoire) du schéma de base AMBA. Les autres tests en simulation se font de la même manière, avec de nouvelles constantes d'adresse et de nouveaux pointeurs sur les endroits désirés.

Les commandes suivantes permettent de compiler le programme en C pour l'obtenir en fichier SREC :

```
sparc-elf-gcc -msoft-float -c -g -O2 nom_du_fichier.c
sparc-elf-mkprom -freq 25 -rmw nom_du_fichier.o -msoft-float
sparc-elf-objcopy -O srec prom.out nom_du_fichier.srec
```

Le paramètre freq est en MHz.
Note : le compilateur doit se trouver dans le PATH :

```
export PATH=/opt/sparc-elf-3.4.4/bin:$PATH
```

Le générique fname de la ROM doit correspondre au fichier SREC obtenu.

7.2.5 Slave Wishbone rapide et lent

L'émetteur SPDIF est le seul slave utilisé dans le projet et il est de vitesse moyenne (un wait-state requis). Des slaves rapide (0 wait-state) et lent(4 wait-states) doivent donc être développé pour vérifier si les ponts peuvent les supporter.
Ces nouveaux slaves contiennent quatre registres de 32 bits où le processeur peut y écrire et lire des données. Le registre 0 est mis en sortie.

Ces slaves permettent également de tester si les ponts peuvent gérer plusieurs slaves Wishbone en même temps.

Le code vhdl de ces slaves est en annexe G1 et G2.

7.2.6 Résultats

Enormément de simulation ont été faite pour tester les ponts, il n'est donc pas possible de toutes les présenter ici et cela serait également fastidieux à lire. Seules les simulations de lecture et écriture sur les différents périphériques sont donc mentionné dans ce chapitre.

Lors du lancement de la simulation, il est déjà intéressant de voir la liste des périphériques AMBA avec leur numéro d'index, mapping dans la mémoire, …

Les premiers tests en simulation se font donc sur la RAM.
Ecriture d'un **caractère** sur la RAM :

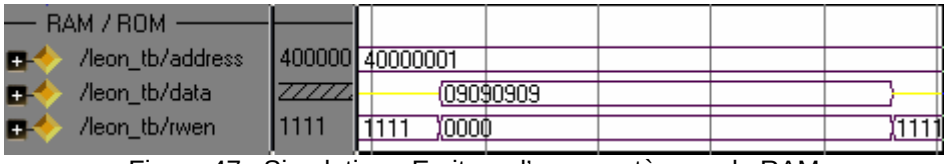


Figure 47 : Simulation - Ecriture d'un caractère sur la RAM

On remarque que la donnée 0x09 est envoyée aux quatre RAM, puisque il y a une écriture 0x09090909 et tous les write enable sont à 0.

Ecriture puis lecture d'un **entier** sur la RAM :

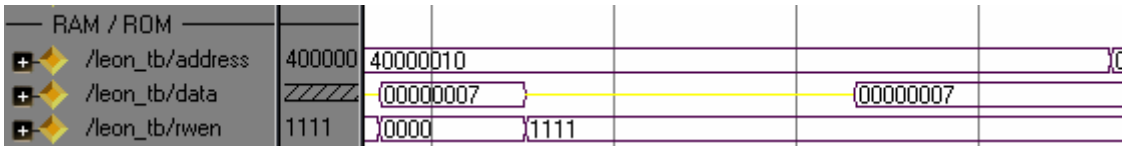


Figure 48 : Simulation – Ecriture d'un entier sur la RAM

Un programme peut donc avoir des variables stockées dans la RAM.

La prochaine étape consiste à tester le pont APB.
Ecriture puis lecture sur l'émetteur SPDIF du pont APB / Wishbone :

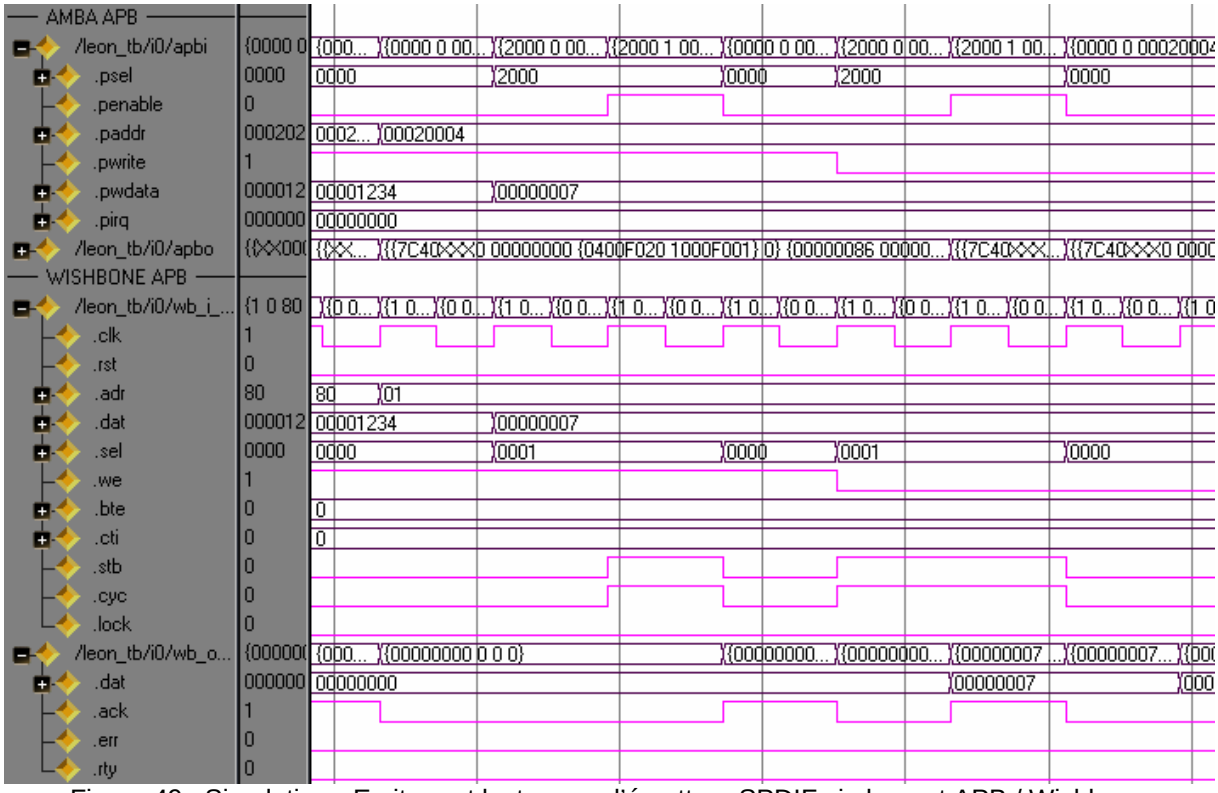


Figure 49 : Simulation - Ecriture et lecture sur l'émetteur SPDIF via le pont APB / Wishbone

Le bus AMBA APB place les adresses (0x20004 (2 pour le pont et 4 pour le 2ème mot de 32 bits)), le write enable à 1 et les données, puis met l'enable à 1.

Le bus Wishbone change immédiatement ses entrées en fonction de celles du bus APB.
Pour la lecture, les deux enable du bus Wishbone un coup d'horloge plus tôt.

Le bus AMBA APB ne peut pas inclure de wait-state dans un transfert et a donc une durée fixe. De ce fait, un slave Wishbone lent ne peut pas fonctionner avec le pont APB / Wishbone.

Ecriture et lecture sur le slave rapide du pont APB / Wishbone :

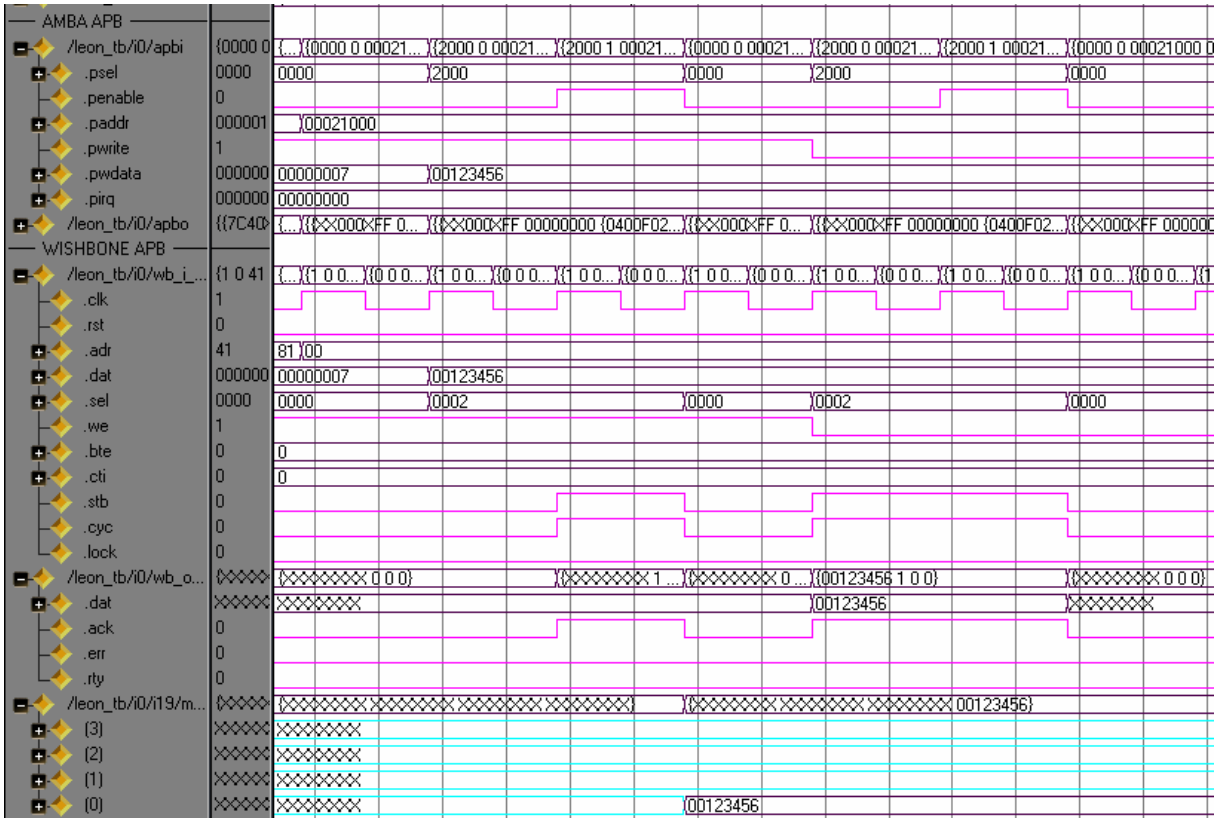


Figure 50 : Simulation - Ecriture et lecture sur le slave rapide via le pont APB / Wishbone

Les derniers signaux sont l'état des registres du slave rapide.
On remarque que le signal ACK arrive plus rapidement que pour l'émetteur SPDIF.

Les mêmes tests sont effectués pour le pont AHB / Wishbone.
La figure suivante présente une écriture sur l'émetteur SPDIF sur le pont AHB / Wishbone :

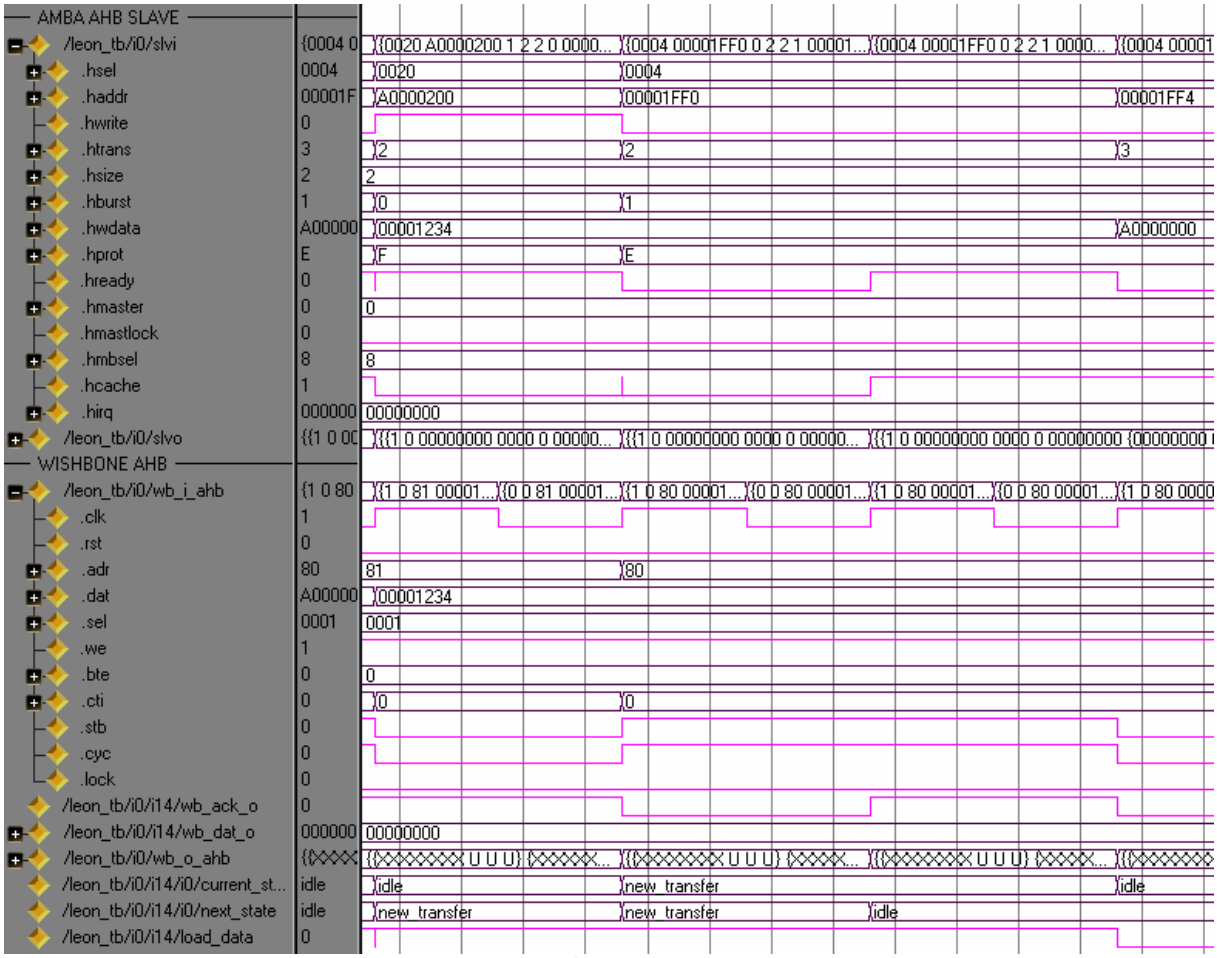


Figure 51 : Simulation - Ecriture sur l'émetteur SPDIF via le pont AHB / Wishbone

Le processeur place les adresses et les bits de contrôle dans un premier flanc montant du clock et aussi les données qui pourraient arriver au second flanc montant. Le pont place les informations sur le bus Wishbone et un wait-state sur le bus AMBA. Lorsqu'il reçoit le ACK du slave, le pont se met en ready et le transfert est exécuté.

Lecture sur l'émetteur SPDIF du pont AHB / Wishbone :

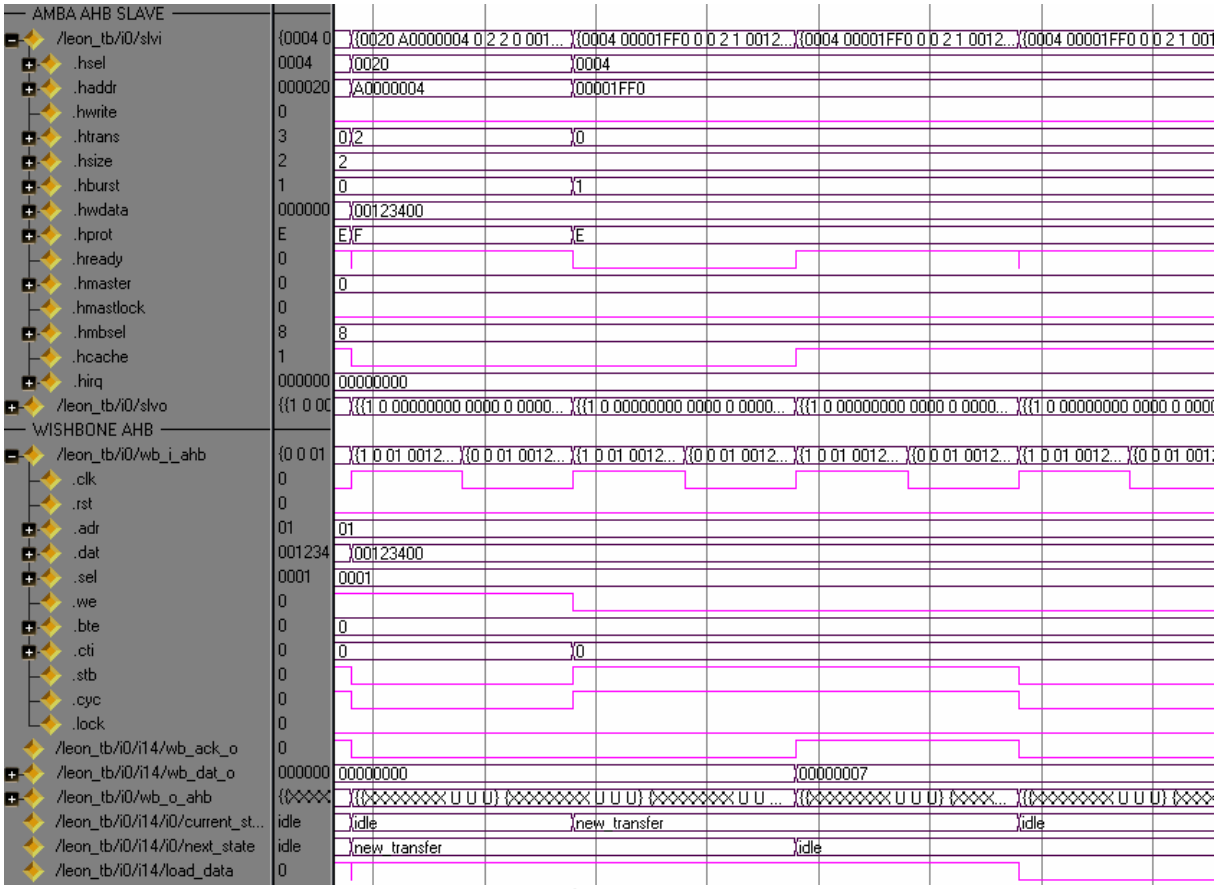


Figure 52 : Simulation - Lecture sur l'émetteur SPDIF via le pont AHB / Wishbone

Le processeur place les adresses et les bits de contrôle dans un premier flanc montant du clock. Le pont place les informations sur le bus Wishbone et un wait-state sur le bus AMBA. Lorsqu'il reçoit le ACK du slave, le pont se met en ready et le transfert est exécuté.

Ecriture sur le slave rapide du pont AHB / Wishbone :

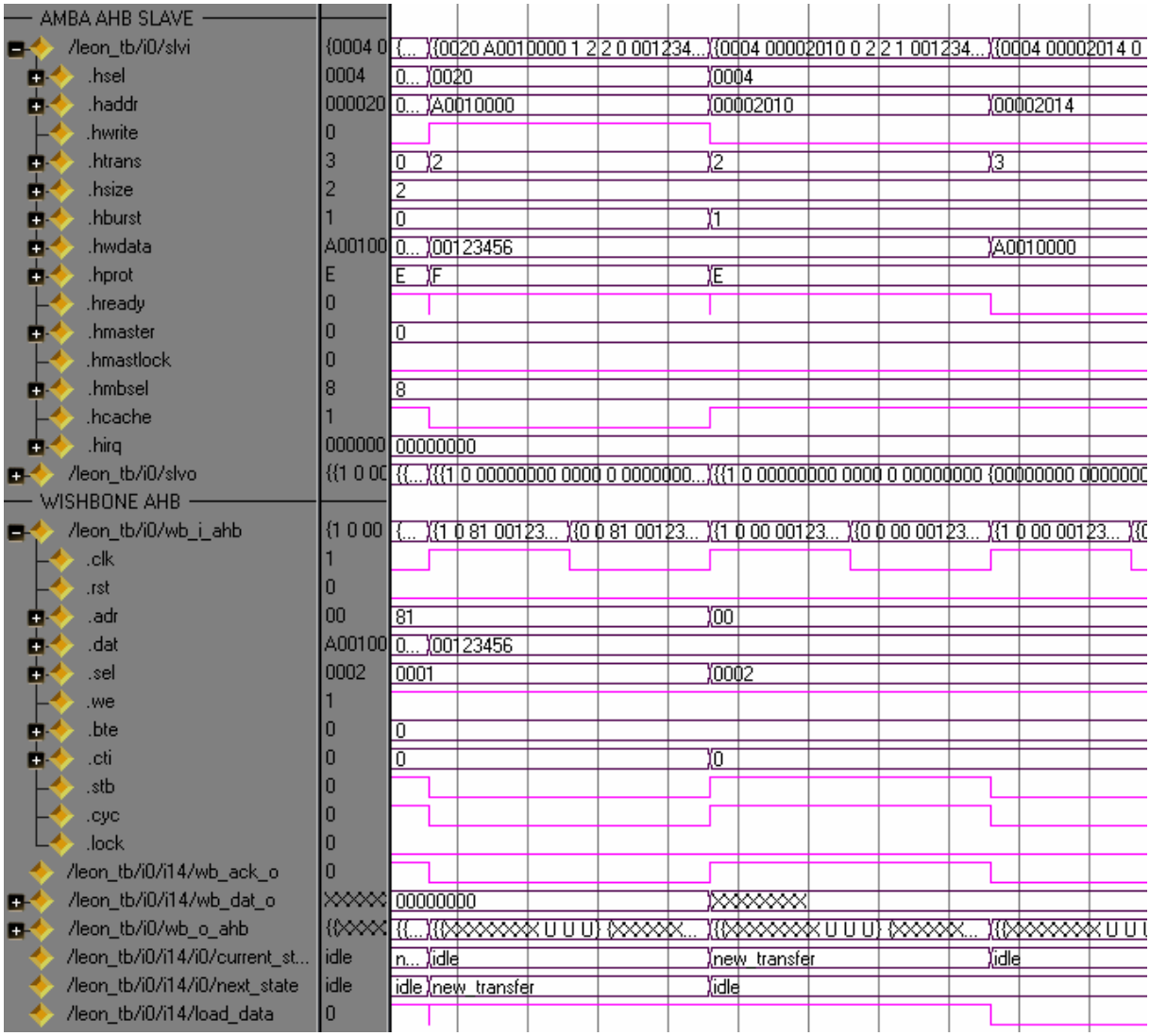


Figure 53 : Simulation - Lecture sur le slave rapide via le pont AHB / Wishbone

Le processeur place les adresses et les bits de contrôle dans un premier flanc montant du clock et aussi les données qui pourraient arriver au second flanc montant. Au second coup d'horloge, le pont place les informations sur le bus Wishbone et pas de wait-state sur le bus AMBA parce qu'il reçoit directement le ACK du slave, le transfert est donc exécuté au 3ème coup d'horloge.

Lecture sur le slave rapide du pont AHB / Wishbone :

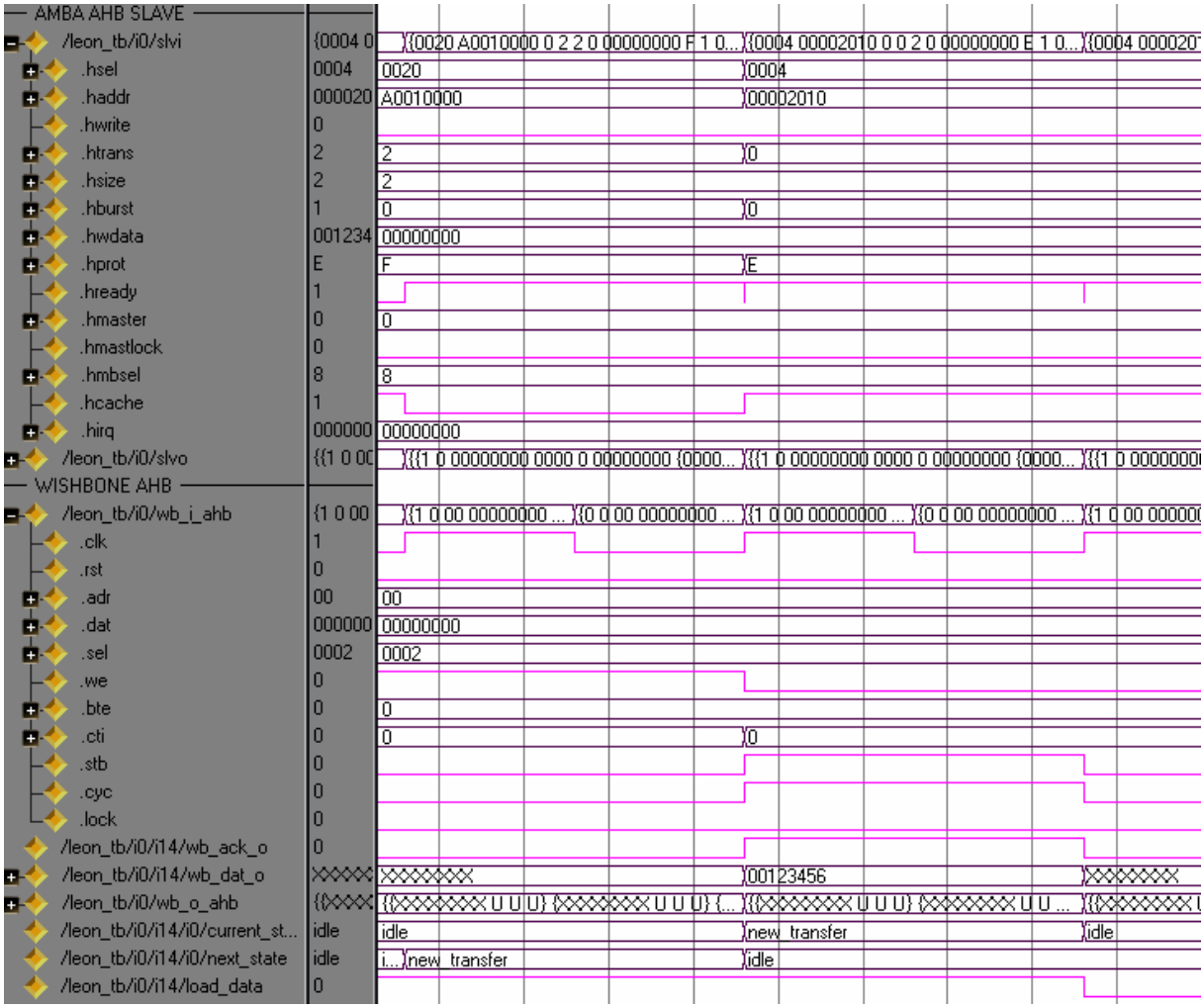


Figure 54 : Simulation - Lecture sur le slave rapide via le pont AHB / Wishbone

Le processeur place les adresses et les bits de contrôle dans un premier flanc montant du clock. Au second coup d'horloge, le pont place les informations sur le bus Wishbone et pas de wait-state sur le bus AMBA parce qu'il reçoit directement le ACK du slave, le transfert est donc exécuté au 3ème coup d'horloge.

Ecriture sur le slave lent du pont AHB / Wishbone :

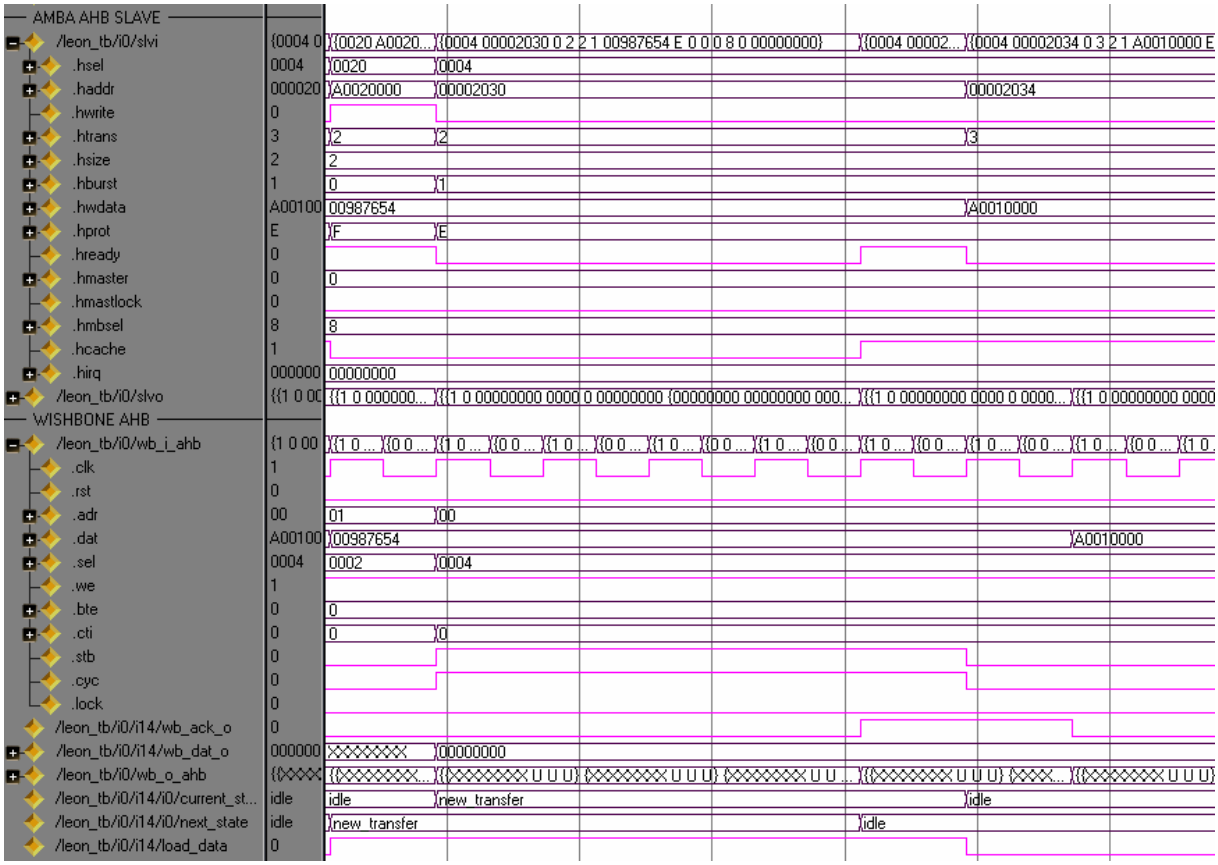


Figure 55 : Simulation - Ecriture sur le slave lent via le pont AHB / Wishbone

Le processeur place les adresses et les bits de contrôle dans un premier flanc montant du clock et aussi les données qui pourraient arriver au second flanc montant. Au second coup d'horloge, le pont place les informations sur le bus Wishbone et des wait-states sur le bus AMBA jusqu'à ce qu'il reçoive le ACK du slave, le transfert est donc exécuté au 7ème coup d'horloge.

Lecture sur le slave lent du pont AHB / Wishbone :

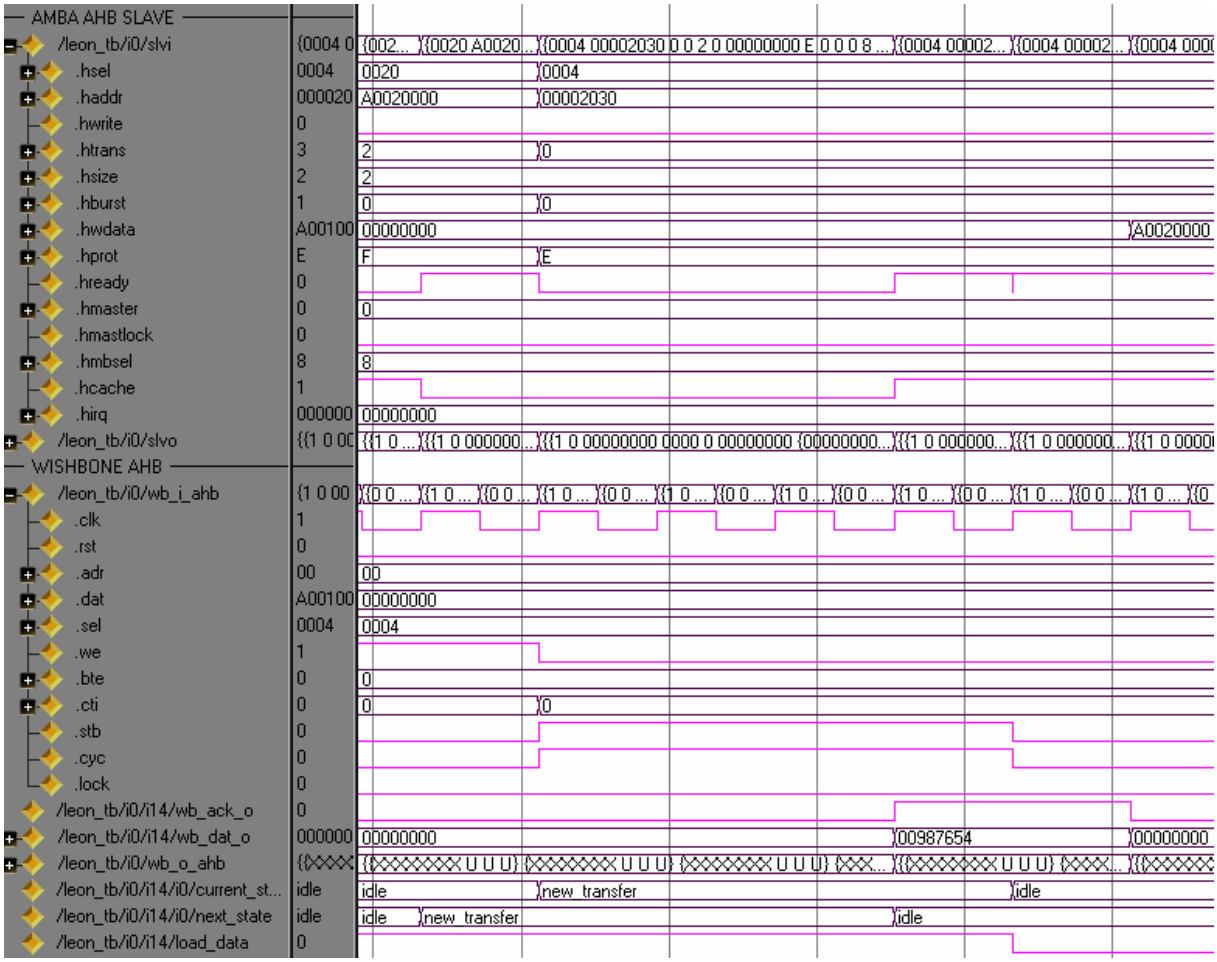


Figure 56 : Simulation - Lecture sur le slave lent via le pont AHB / Wishbone

Le processeur place les adresses et les bits de contrôle dans un premier flanc montant du clock. Au second coup d'horloge, le pont place les informations sur le bus Wishbone et des wait-states sur le bus AMBA jusqu'à ce qu'il reçoive le ACK du slave, le transfert est donc exécuté au 7ème coup d'horloge.

Les burst transfers sont compatible avec une vitesse de slave rapide et moyenne. En effet, le slave lent met son signal ACK à 0 trop lentement et le master croit que le slave est prêt à recevoir un nouveau transfert. Pour faire du burst, le slave doit donc être capable de faire des transferts en un coup de clock ou pouvoir placer son signal ACK à 0 suffisamment rapidement.
Il aurait été possible de faire du burst sur un slave lent en utilisant par exemple le signal RTY du bus Wishbone pour placer des wait-states. Cela dit, il n'y a pas un grand intérêt à faire du burst sur un slave lent.

Burst transfer sur le slave rapide :

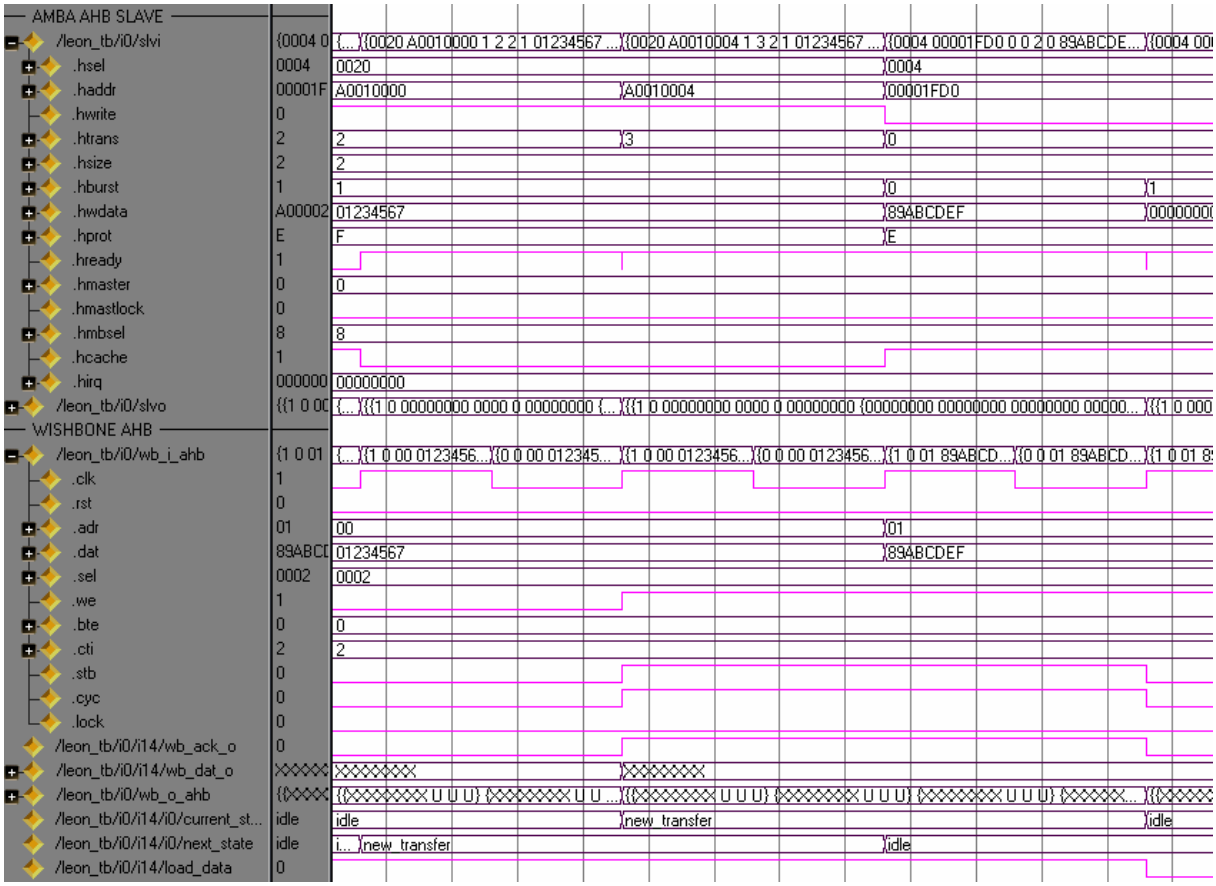


Figure 57 : Simulation – Burst transfer sur le slave rapide via le pont AHB / Wishbone

Le processeur place les adresses, les bits de contrôle et les données sur le bus AHB, quand il n'y a plus de wait-state (hready = 1) le pont met les signaux d'enable du bus Wishbone à 1.
Au coup d'horloge suivant, le transfert est exécuté, puis les adresses sont incrémentées et les données sont actualisées.
Au dernier flanc montant du clock, le 2ème transfert est exécuté.

Burst transfer sur l'émetteur SPDIF :

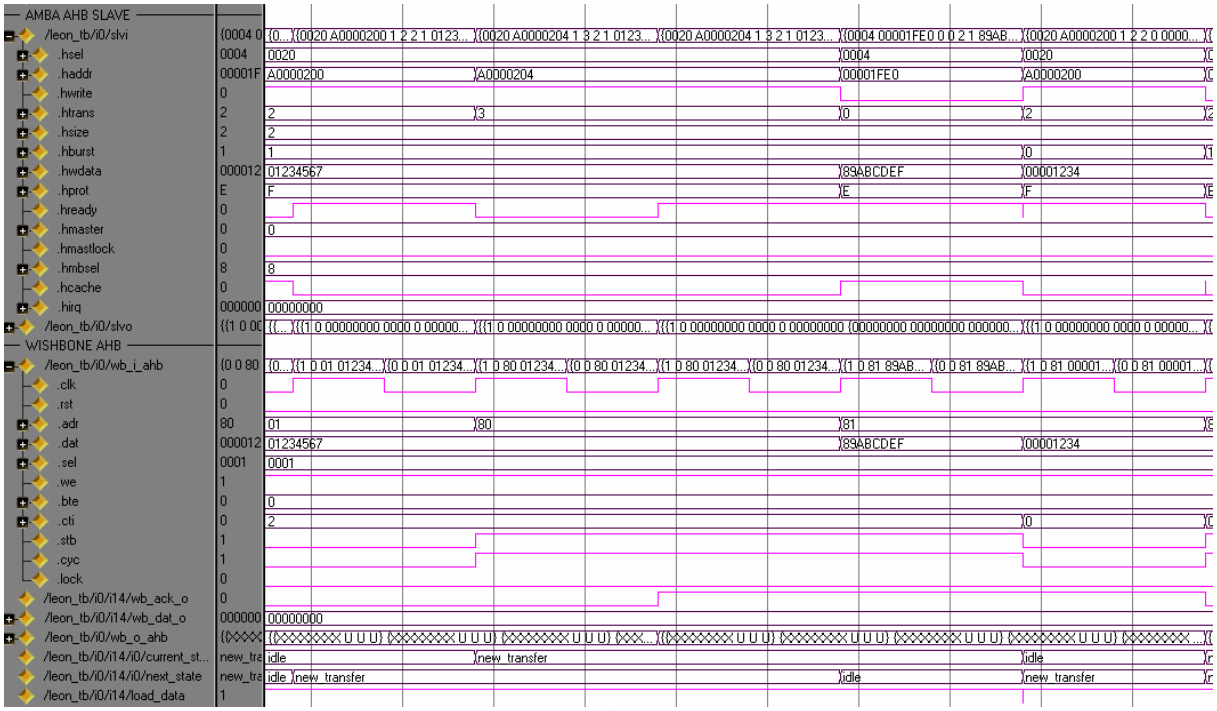


Figure 58 : Simulation – Burst transfer sur l'émetteur SPDIF via le pont AHB / Wishbone

Le processeur place les adresses, les bits de contrôle et les données sur le bus AHB, quand il n'y a plus de wait-state (hready = 1) le pont met les signaux d'enable du bus Wishbone à 1 et place un wait-state.
Au coup d'horloge suivant, le pont reçoit un ACK du slave et enlève le wait-state.
Au 3ème coup d'horloge, le transfert est exécuté, puis les adresses sont incrémentées et les données sont actualisées.
Au dernier flanc montant du clock, le 2ème transfert est exécuté.

On remarque que l'émetteur SPDIF a besoin d'un wait-state au début, puis il peut recevoir un transfert par clock.

Signal wb_burst

Un signal wb_burst a été créé dans le pont AHB / Wishbone, il indique quand le pont fait un burst transfer. Le code suivant crée ce signal :

```
burst2: process(hburst, clk)
begin
if rising_edge(clk) then
   wb_burst <= sel and hready_i and (hburst(0) or hburst(1) or hburst(2));
end if;
end process burst2;
```

La figure suivante montre le signal wb_burst lors des précédents tests :

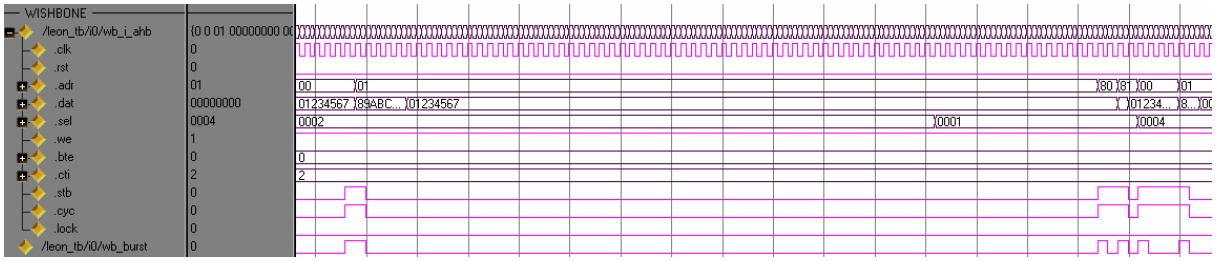


Figure 59 : Simulation – Signal wb_burst

L'impulsion de gauche est le burst du slave rapide, elle dure deux coups de clock et il n'y a donc pas de wait-state.
Les deux impulsions suivantes sont celui du burst de l'émetteur SPDIF, l'espace entre les deux impulsions est du au wait-state.
Les deux dernières impulsions sont celui du burst sur le slave lent, on remarque qu'il y a quatre wait-state pour le premier transfert, mais aucun pour le deuxième.

7.2.7 Résumé

Le tableau ci-dessous résume les tests de simulation :

		transfert normal		burst transfer	
		écriture	lecture	écriture	lecture
APB	slave rapide	ok	ok		
	émetteur SPDIF	ok	ok		
AHB	slave rapide	ok	ok	ok	ok
	émetteur SPDIF	ok	ok	ok	ok
	slave lent	ok	ok		
RAM		ok	ok		

7.2.8 Conclusion

La simulation offre des résultats satisfaisants. Les conversions AMBA AHB / Wishbone et APB / Wishbone se passent comme il avait été prévu. Cela permet d'espérer de bons résultats avec la FPGA.

7.3 Moniteur Grmon

Le chapitre « Implémentation » décrit la procédure pour faire fonctionner le circuit dans la FPGA. Cette partie ne fait donc que rappeler divers manières de tester le circuit. Afficher des captures d'écran du moniteur n'aurait pas apporté de nouvelles informations.

La connexion avec la carte depuis le moniteur Grmon détermine déjà si l'UART AHB et le DSU fonctionnent. Si tel n'est pas le cas, il est quand même possible de vérifier la présence du DSU avec la LED DSUACT.

La commande « info sys » du Grmon liste les différents périphériques du circuit détectés.
Les commandes « mem » et « wmem » permettent de lire et écrire sur les registres et mémoire du circuit.
Inclure des « printf » dans le programme avec notamment les valeurs des variables est un bon outil de debug du programme.

7.4 Lecture d'un fichier musical

Divers fichiers ont été testés et tous ont fonctionné avec une bonne qualité de son. En effet, le bus SPDIF étant numérique, il n'y a pas de perte dans les câbles et sur la carte d'adaptation.

Le signal SPDIF et deux autres signaux peuvent être mesuré grâce à des pointes de test sur la carte.
Le signal wb_burst a été mis sur une de ces pointes de test et le même programme de test de la simulation vue précédemment a été exécuté. La figure suivante présente le résultat :

Figure 60 : Mesure – Signal wb_burst

On remarque que les timings sont les même que lors de la simulation.

7.5 Conclusion

Les objectifs sont remplis, les ponts fonctionnent pleinement et ils sont configurables pour s'adapter à toutes les situations où il est techniquement possible d'y avoir des slaves Wishbone.

8. Travail supplémentaire : Contrôleur Ethernet

8.1 Introduction

Ayant terminé les ponts deux semaines à l'avance il a été décidé d'ajouter un contrôleur Ethernet au circuit pour augmenter la vitesse de téléchargement du fichier musical. En effet, celle-ci prend avec le port série à 115'200 baud 30 min à 1 heur suivant la taille du fichier.

Les librairies de Gaisler fournissent un contrôleur Ethernet nommé GRETH (Gaisler Research's Ethernet Media Access Controller) qui se place comme un master AHB avec des registres accessible sur le bus APB.

8.2 Caractéristiques du GRETH

Le GRETH fournit une interface entre un bus AMBA – AHB et un réseau Ethernet. Il supporte une vitesse de 10/100 Mbit dans les deux modes full et half duplex et dans les deux interfaces MII et RMII qui devraient être connectés à une couche physique.

Une interface MDIO est utilisée pour accéder à la configuration et aux registres de status dans une ou plusieurs couches physiques connectées au MAC. Cette interface peut aussi être contrôlée depuis le bus APB.

Un support optionnel de debug EDCL (Ethernet Debug Communication Link) est aussi fournit. L'EDCL permet de se connecter au DSU avec le Grmon par Ethernet au lieu du port série. Il utilise les protocoles ARP, IP et UDP ensemble et une couche d'application propre.

Le GRETH contient évidemment un émetteur et un récepteur, mais seul le récepteur est étudié ici, bien que leur fonctionnement soit similaire.

Toutes les informations du GRETH peuvent se trouver sur le document grip.pdf.

8.3 Utilisation du GRETH

Le contrôleur copie les paquets Ethernet qui correspondent à son adresse Ethernet à un endroit de la mémoire pointé par un descripteur aussi dans la mémoire, pointé par un registre du GRETH. A chaque paquet reçu, la valeur du registre s'incrémente automatiquement et il va lire le descripteur suivant. Un descripteur a la taille de 2 bytes et la grandeur de la plage de mémoire des descripteurs doit être 1kBytes. Le pointeur va automatiquement retourner à l'adresse de base des descripteurs quand la limite des 1kBytes est atteinte.

La figure ci-dessous donne une vue de la mémoire avec un exemple d'adressage :

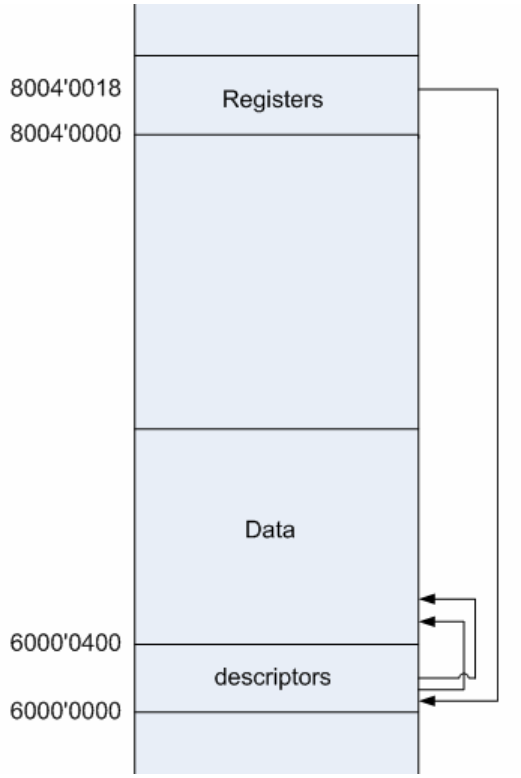


Figure 61 : Memory mapping du GRETH

Le descripteur doit être d'abord configuré avant de commencer la réception.
Lorsqu'un paquet est reçu, les bits de configuration du descripteur sont mis à 0 et les bits d'indication d'erreurs sont actualisés.
Le tableau suivant liste les bits du byte 0 d'un descripteur :

Bits	Description
0 - 10	Longueur du paquet reçu en byte
11	enable du descripteur (doit être mis en dernier)
12	wrap (WR) retourne le pointeur des descripteurs au premier descripteur (1) sinon augmente le pointeur de 8 (0)
13	enable des interruptions, si ce bit est mis à 1 une interruption est générée lorsque le paquet est reçu
14	Alignement Error (AE), le paquet reçu n'est pas aligné sur 8 bits
15	Frame Too long (FT), le paquet reçu est trop long et l'excédant est tronqué
16	CRC Error (CE), indique une erreur de CRC dans le paquet reçu
17	Overrun Error (OE), indique que le paquet reçu a subi un dépassement dans la FIFO
18-31	réservés

Le tableau suivant liste les bits du byte 1 d'un descripteur :

Bits	Description
0-1	réservés
2-31	Adresses du paquet qui va être reçu

Le tableau suivant liste les registres du GRETH :

Register	APB Address offset
Control register	0x0
Status regiter	0x4
MAC Address MSB	0x8
MAC Address LSB	0xC
MDIO Control / Status	0x10
Transmitor descriptor pointer	0x14
Receiver descriptor pointer	0x18
EDCL IP	0x1C

Le tableau suivant liste les bits du Control register :

Bits	Description
0	Enable de l'émetteur
1	Enable du récepteur
2	Enable de l'interruption de l'émetteur, envoie une interruption quand un paquet est envoyé
3	Enable de l'interruption du récepteur, envoie une interruption quand un paquet est reçu
4	Full Duplex (FD), 1 : full duplex, 0 : half duplex
5	Promiscus Mode (PM), 1 : reçoit tous les paquets sans regarder l'adresse Ethernet, 0 : reçoit que les paquets dont l'adresse de destination est la même que celle du GRETH et les broadcasts
6	Reset, 1 : reset le GRETH
7	Speed (SP), 0 : 10Mbit, 1 : 100Mbit
8-27	réservés
28-30	EDCL buffer size (BS), taille du buffer de l'EDCL, 0 : 1kB, 1 : 2kB, …, 6 : 64kB
31	EDCL enable (ED), à 1 si l'EDCL a été activé dans les génériques

Le tableau suivant liste les bits du Status register, tous ces bits peuvent être remis à 0 en y écrivant un 1 :

Bits	Description
0	Receiver error (RE), indique si un paquet a été reçu avec une erreur
1	Transmittor error (TE), indique si un paquet a été transmis avec une erreur
2	Receiver Interrupt (RI), indique si un paquet a été reçu sans erreur
3	Transmittor Interrupt (TI), indique si un paquet a été transmis sans erreur
4	Receiver AHB error (RA), indique si le récepteur a eu une erreur sur le bus AHB
5	Transmittor AHB error (TA), indique si le émtteur a eu une erreur sur le bus AHB
6	Too Small(TS), indique si le paquet reçu est plus petit que la taille minimum
7	Invalid Address (IA), indique si un paquet pas accepté par l'adresse MAC a été reçu

L'adresse MAC doit être écrite sur les bits 15 à 0 du registre MAC Address MSB et sur tous les bits du registre MAC Address LSB.

Le registre Receiver descriptor pointer représente l'adresse du descripteur qui est pointé.
Les bits 31 à 10 du registre Receiver descriptor pointer sont l'adresse de base des descripteurs, les bits 9 à 3 pointent sur le descripteur actif et les bits 2 à 0 ne peuvent pas être mis à 1 du fait qu'un descripteur tient sur 2 bytes.

Donc, pour un paquet ordinaire, la procédure suivante doit être suivie :
- Ecrire l'adresse du paquet dans un premier descripteur aligné sur 1kByte
- Ecrire 0x800 dans le byte 0 du descripteur

- Ecrire l'adresse du descripteur dans le registre 6 du GRETH
- Commencer la réception en mettant à 1 le receiveur enable du Control register du GRETH
- Contrôler le flag 4 du status register du GRETH jusqu'à ce qu'il soit à 1
- Le descripteur doit avoir contenir la longueur du paquet reçu et d'éventuels bits d'erreur
- Créer un nouveau descripteur
- …

8.4 Programmation

Pour simplifier au maximum la programmation en C, le protocole UDP a été choisi, le GRETH ne fera donc que recevoir des paquets et n'en enverra aucun.

Lorsque le GRETH reçoit un paquet, il l'écrit tel quel à l'adresse qui est pointée par le descripteur. C'est-à-dire qu'il y aura aussi les headers de la couche Ethernet et IP. Les paquets ne sont donc pas lisible de suite par l'émetteur SPDIF et les données doivent donc être copiée pendant le téléchargement du fichier. Dans cet objectif et dans le but de perdre le moins de paquets possible, il a été décidé de créer de la place en mémoire pour 8 paquets en plus de la plage de mémoire des descripteurs. Le reste de la SDRAM est réservée pour les données qui seront copiées.
Il y a donc huit descripteurs qui pointent chacun sur une zone mémoire des paquets. La grandeur de cette zone est fixée à 2kbyte, car la grandeur maximale d'un paquet est 1.5kbytes.
La figure suivante présente la vue mémoire de la stratégie choisie :

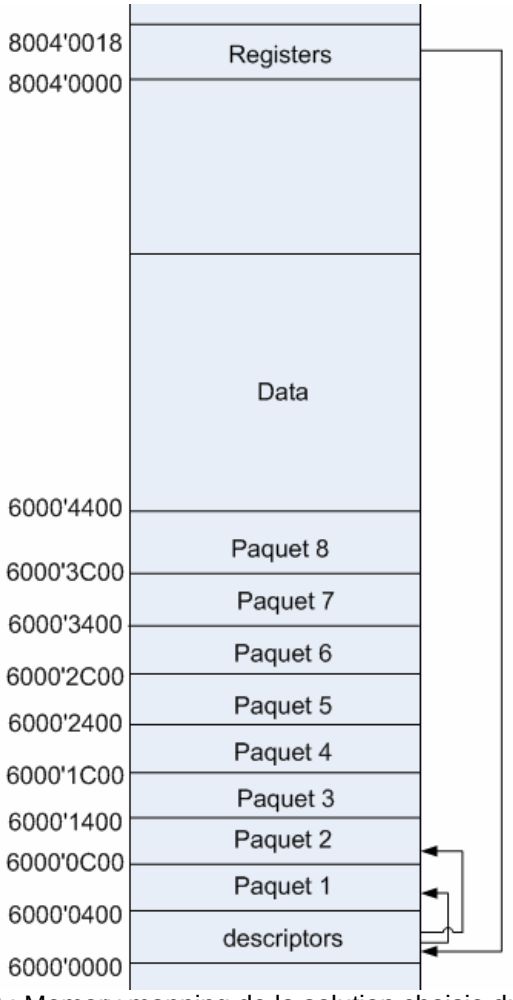


Figure 62 : Memory mapping de la solution choisie du GRETH

La figure suivante présente l'algorithme de chargement d'un fichier sur la SDRAM par Ethernet :

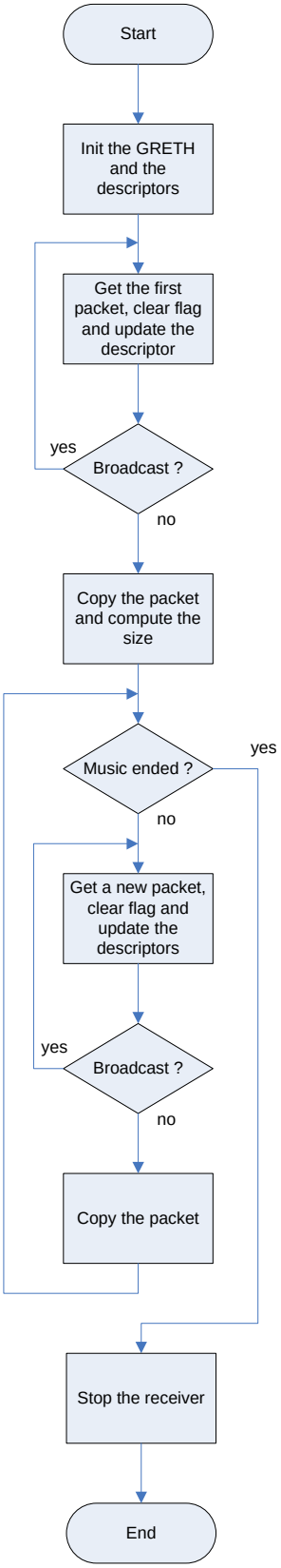


Figure 63 : Diagramme d'écriture du fichier WAV en SDRAM avec le GRETH

Les headers des paquets Ethernet et IP cumulés ne sont pas alignés sur 32 bits, mais sur 16 bits. Il est donc conseillé d'exécuter des accès sur 16 bits lors de la copie des données. Trois accès sont donc nécessaires pour la copie de données.

Lors de la copie du fichier WAV, les données doivent toujours être inversées pour changer entre big endian et little endian.

8.5 Envoi du fichier depuis le PC

Le GRETH n'a pas d'adresse IP mais seulement une MAC, or le protocole UDP a besoin d'une adresse IP. Une adresse IP fictive doit donc lui être attribuée. Cette adresse ne doit évidemment pas être utilisée par une autre station dans le réseau Ethernet.

La commande « arp -a» permet de voir les adresses MAC attribuée à une adresse IP connue par le PC.
La commande « arp –s ADDR_IP ADDR_ETH » permet d'ajouter une adresse MAC attribuée à une adresse IP dans la liste vue ci-dessus.
Un programme qui envoie un paquet à une adresse IP listée dans « arp –a » a donc l'adresse MAC attribuée dans le paquet Ethernet. Si tel n'est pas le cas, une requête ARP aurait eu lieu et la station à l'adresse IP aurait du répondre sont adresse MAC. Comme il a été décidé de ne pas envoyer de paquet avec le GRETH pour simplifier la programmation, cette solution plus évidente a été choisie.

Le protocole Ethernet est rapide et a des vitesses de pointes de 10Mbit à 100Mbit suivant ses caractéristiques. L'horloge du bus AMBA de la carte utilisée est de 25MHz.
Comme plusieurs transferts doivent être exécutés pour chaque mot de 32 bits d'un paquet, le protocole Ethernet est trop rapide pour le bus AMBA et le programme utilisée. Le programme qui envoie le fichier ne doit donc pas utiliser la pleine capacité du protocole Ethernet et a besoin d'être ralenti.

Le programme netcat aurait pu être employé pour envoyer le fichier s'il n'y avait pas besoin de ralentir la vitesse d'envoi.
La commande aurait été :

```
nc –vvn –u ADDR_IP PORT < nom_du_fichier
```

Un programme a donc été écrit dans le langage de programmation Perl sous le nom de udpSend.pl. Celui-ci se trouve en annexe H et a été développé par le professeur et superviseur du projet François Corthay.
La commande pour l'utiliser est :

```
./udpSend.pl ADDR_IP PORT nom_du_fichier
```

8.6 Conclusion

Ce travail supplémentaire a permis d'améliorer l'utilisation et la présentation de l'application choisi du pont. Il a également amené de nouveaux domaines de l'électronique dans le projet.
Le résultat n'est cependant pas bon à 100% : il arrive que certains paquets ne soit pas reçu (de l'ordre de 1 sur 10'000) et que le contrôleur plante sans raison apparente.
Il a été essayé de se connecter au GRETH avec le Grmon et de passer le programme contenant le fichier son, mais il était impossible de compiler un fichier de cette taille.

9. Bilan

Il est intéressant de regarder une vue du schéma final :

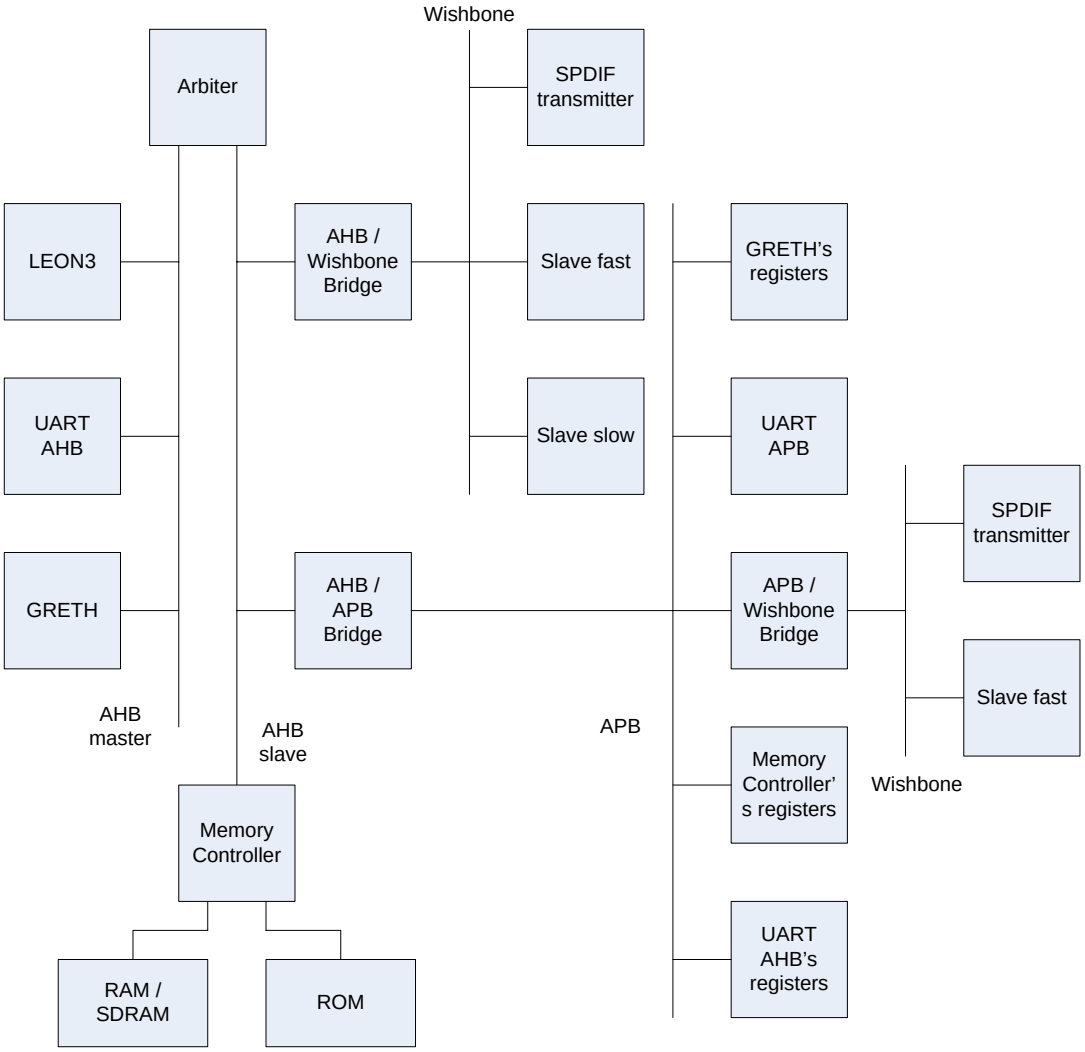


Figure 64 : Vue final du schéma

La figure suivante est le memory mapping complet du circuit final :

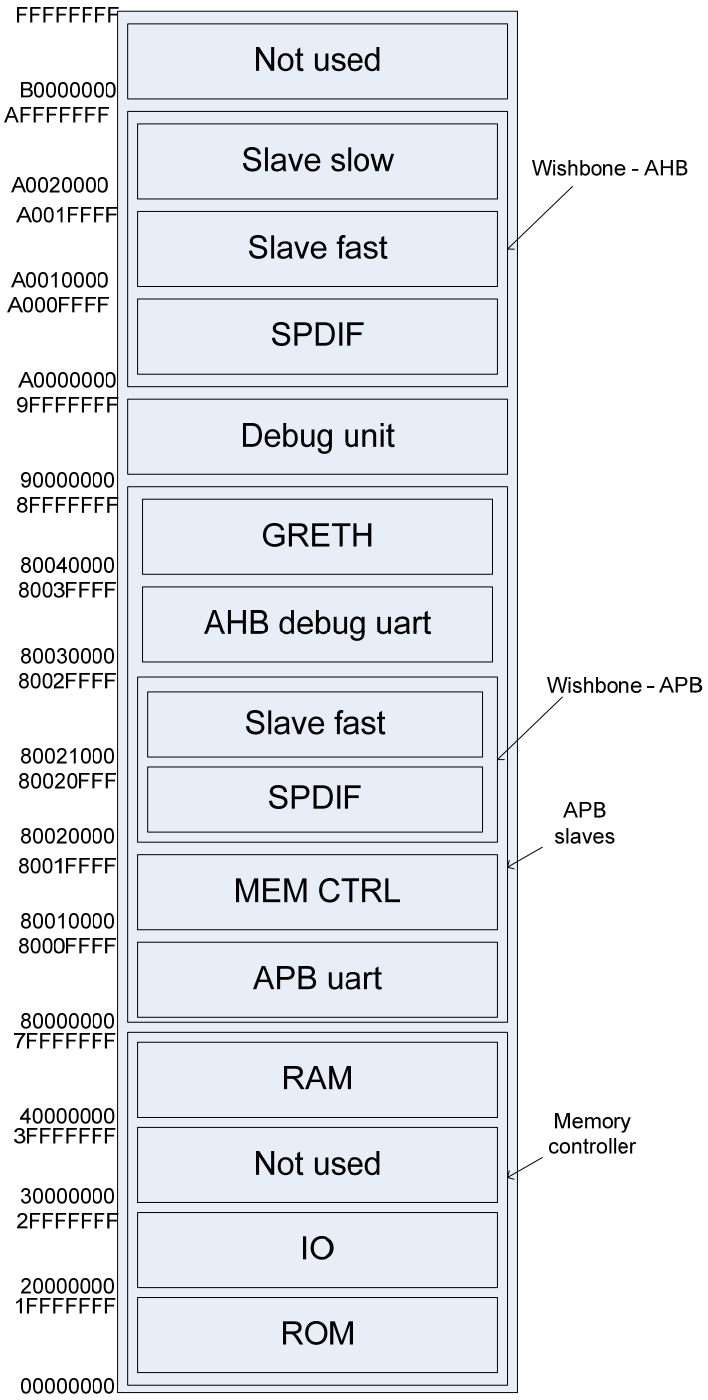


Figure 65 : Memory mapping final du schéma

Le circuit final donne le rapport d'implémentation présent dans l'annexe I. On voit que la FPGA contient encore de la place.

Le tableau suivant présente le déroulement final du travail de diplôme :

		3-7 sept	10-14 sept	17-21 sept	24-28 sept	1-5 oct	8-12 oct	15-19 oct	22-26 oct	29-2 nov	5-9 nov	12-16 nov	19-23 nov
1	Implémentation suzaku	X											
2	Implémentation FPGA Board		X	X									
3	Programmation SDRAM			X	X								
4	Lecture de la musique				X	X							
5	Pont AHB / Wishbone					X	X	X					
6	Slave rapide et lent, Pont APB / WB								X				
7	Différents tests									X			
8	Contrôleur Ethernet									X	X	X	
9	Rapport			X	X	X	X	X	X	X	X	X	X

Il est à noter qu'il y a eu un jour férié le 1er novembre et une journée au forum HES-SO 07 le 7 novembre.

10. Conclusion

Les objectifs sont atteints et un travail supplémentaire a été réalisé. En effet, le pont AMBA-AHB / Wishbone a été conçu et le pont AMBA-APB / Wishbone a été amélioré. Leur fonctionnement a été prouvé par différents tests et le résultat final a été la lecture d'un fichier musical sur des haut-parleurs. Je garde une vue clairement positive sur le bilan de ce projet, tant sur le plan technique que sur celui de la gestion d'un projet.
Néanmoins, des améliorations mineures sont encore possible, entre autre :

- lire directement un fichier depuis le PC en passant par Ethernet sans le stocker sur la SDRAM
- développer une solution pour les interruptions sur le pont AMBA – AHB / Wishbone

Les principales difficultés du projet étaient l'implémentation sur FPGA, qui a pris plusieurs semaines, et la compréhension de la conversion AHB / Wishbone.

L'unité infotronique de l'école possède désormais les ponts AMBA – AHB / Wishbone et AMBA – APB / Wishbone et les possibilités offertes par les produits de Opencores.org sont donc ouvertes à leurs projets basées sur les librairies de Gaisler Research.

Pour terminer, je tiens à remercier les personnes suivantes pour leurs aides, conseils et soutiens durant ce travail de diplôme :
- François Corthay, professeur et superviseur de ce travail de diplôme
- Christophe Bianchi, professeur
- Oliver Gubler, adjoint scientifique
- Marc Pignat, adjoint scientifique
- Sébastien Farquet, adjoint scientifique
- Mes collèges diplômant de la salle A309

Sion, le 23 novembre 2007 Samuel Valentini

11. Liste des Annexes

A : Fonctionnement du plug and play de Gaisler et les signaux hconfig et pconfig
B : fichier de configuration du circuit
C : Tutorial : Implémenter un circuit dans une FPGA
D : Tutorial : passer de 32 à 16 bits de données de mémoire
E : Tutorial : Grmon
F1 : Schéma de l'adaptateur SPDIF
F2 : Routage de l'adaptateur SPDIF
G1 : Code VHDL du slave rapide
G2 : Code VHDL du slave lent
H : Code Perl du programme d'upload d'un fichier en protocole UDP
I : Rapport d'implémentation du circuit final
J : Schéma du pont AHB / Wishbone
K : Code VHDL du pont APB / Wishbone
L : Code C du programme qui reçoit un fichier par le port série et le lit
M : Code C du programme qui reçoit un fichier par le port Ethernet et le lit
N : Code C de simulation
O : Tutorial : Compiler et implémenter un design préfabriqué des librairies Gaisler
P : Tutorial : Utilisation de la SDRAM avec le contrôleur de mémoire des librairies Gaisler

12. Liens, Références

Comme dans la plupart des projets technique, une grande partie des sources d'informations se trouvent sur Internet.

L'encyclopédie en ligne Wikipédia contient énormément d'informations. Dans le cadre de ce projet, on peut y trouver notamment sur le VHDL, les FPGA, SPDIF, fichiers WAV, …
http://fr.wikipedia.org/wiki/Accueil

Toutes les informations des librairies de Gaisler se trouvent le site officiel de Gaisler Research.
http://www.gaisler.com/cms/

Forum actif sur le processeur LEON et les librairies de Gaisler Research en général :
http://tech.groups.yahoo.com/group/leon_sparc/

Site officiel du bus AMBA :
http://www.arm.com/products/solutions/AMBAHomePage.html

Site de OpenCores, communauté qui a créé le bus Wishbone. Les blocs certifiés Wishbone se trouvent donc à cette adresse :
http://www.opencores.org/

Explication des fichiers srec :
http://www.die.net/doc/linux/man/man5/srec.5.html

13. Contenu du CD

La liste ci-dessous décrit les différents dossiers présents dans le CD :

- Adaptateur : Fichiers PCAD de l'adaptateur
- doc : Tous les documents utilisés dans le projet
- gaisler : Librairies de Gaisler et un fichier pour créer un projet à partir de celles-là
- grlib-gpl : Designs complet de Gaisler
- LEON : Contient le projet HDL Designer, les synthèses et les fichiers .bit
  - hdl : les fichier vhdl du pont et synthèse dans le dossier synplify
- locigiels : Petits programmes utilisés dans le projet
- Rapport : Le rapport et les annexes
- prog : Programmes en C

ANNEXE A

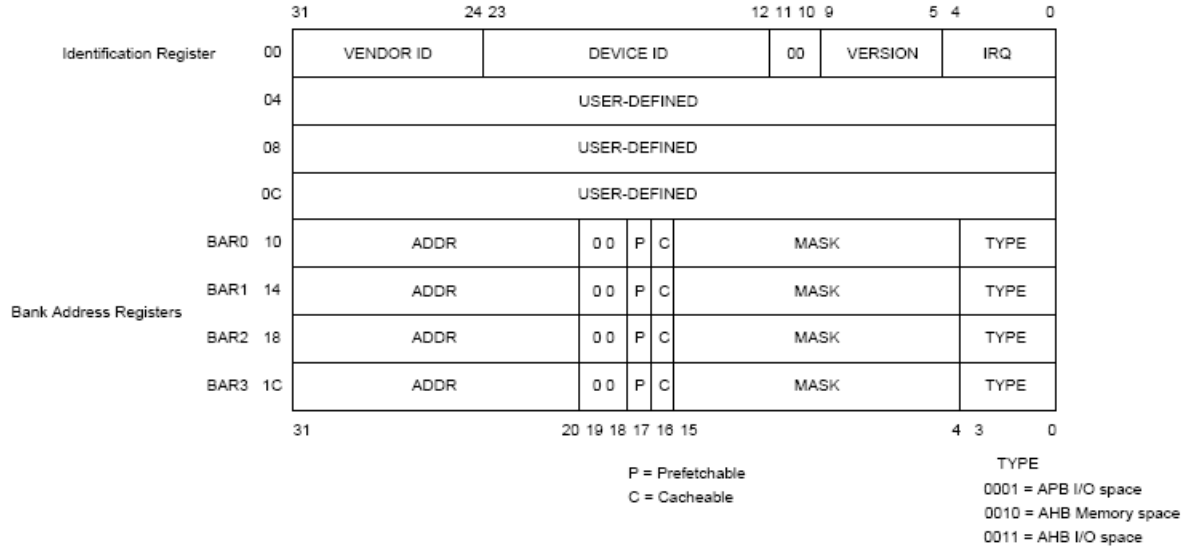
Fonctionnement du plug and play de Gaisler et les signaux hconfig et pconfig

Fonctionnement du « plug and play » de Gaisler et détails du hconfig

Le plug and play de Gaisler est constitué de trois parties : **identification des modules rattachés au bus, un plan d'adressage des slaves et un routage d'interruption**. Ces informations sont contenues dans les signaux du hconfig qui est en sortie de chaque master et chaque slave dans le bus AHB. Le hconfig est divisé en 8 mots de 32 bits.

Toutes les informations sont rassemblées dans une table « read-only » qui se trouve dans une adresse fixe. Cette table, crée par le AHB controller (arbitre), se trouve normalement à l'adresse 0xFFFFF000 – 0xFFFFF800 pour les maître et 0xFFFFF800 – 0xFFFFFFFC pour les slaves. La table a ainsi de la place pour 64 maîtres et 64 slaves. Un système d'exploitation ou une autre application peut donc scanner cette table de configuration et détecter automatiquement quels modules sont présents sur le bus AHB, comment ils sont configurés et où ils sont localisés (slaves).

Le tableau suivant présente le format des signaux hconfig :



Quand on utilise un composant des librairies de Gaisler, la plupart des variables qui composent le hconfig sont déjà prêtes dans le composant. Il ne reste qu'à définir les variables ADDR et MASK (détaillé plus bas) dans les génériques du composants et qui se nomment respectivement HADDR et HMASK.

Identification

L'identification se fait à l'aide de trois variables : Vendor ID (fabricant), Device ID (type de module), Version.

Le Vendor ID est un nombre donné pour un fabricant ou une organisation par Gaisler Research. Le tableau suivant présente les sept différents Vendor ID :

Vendor	ID
Gaisler Research	0x01
Pender Electronic Design	0x02
European Space Agency	0x04
Astrium EADS	0x06
OpenChip.org	0x07
OpenCores.org	0x08
Gleichmann Electronics	0x10

Le numéro 0x0 est réservé pour indiquer qu'aucun module n'est présent.

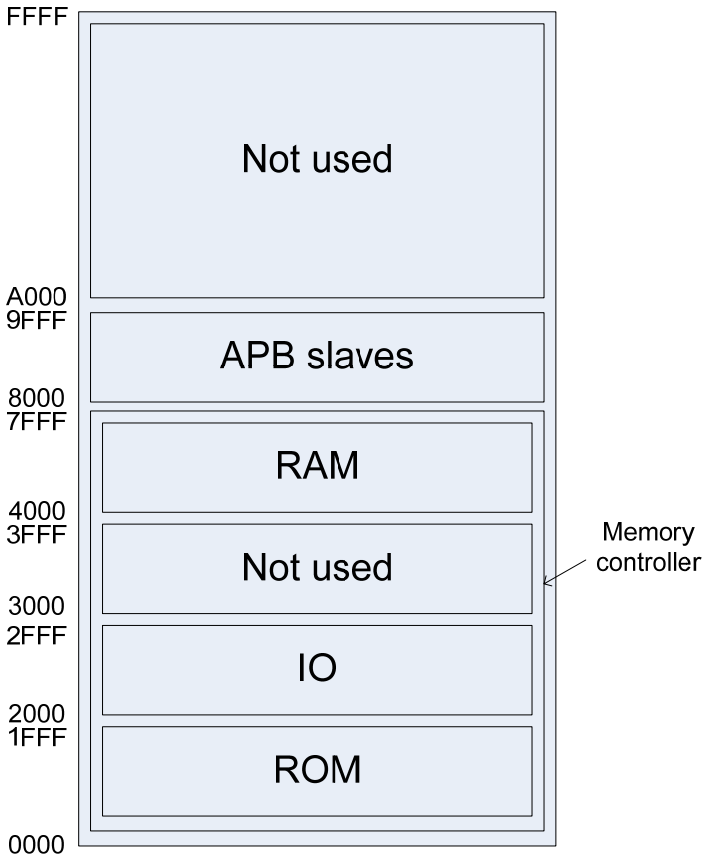
Le tableau suivant présente quelques Device ID :

Gaisler Research device id's	
GAISLER_LEON3	0x003
GAISLER_LEON3DSU	0x004
GAISLER_ETHAHB	0x005
GAISLER_APBMST	0x006
GAISLER_AHBUART	0x007
GAISLER_SRCTRL	0x008
GAISLER_SDCTRL	0x009
GAISLER_APBUART	0x00C
GAISLER_IRQMP	0x00D
GAISLER_AHBRAM	0x00E
GAISLER_GPTIMER	0x011
GAISLER_PCITRG	0x012
GAISLER_PCISBRG	0x013
GAISLER_PCIFBRG	0x014
GAISLER_PCITRACE	0x015
GAISLER_PCIDMA	0x016
GAISLER_AHBTRACE	0x017
GAISLER_ETHDSU	0x018
GAISLER_PIOPORT	0x01A
GAISLER_AHBROM	0x01B
GAISLER_AHBJTAG	0x01C
GAISLER_ATACTRL	0x024
GAISLER_DDRSPA	0x025
GAISLER_USBEHC	0x026
GAISLER_USBUHC	0x027
GAISLER_KBD	0x060
GAISLER_VGA	0x061
GAISLER_SVGA	0x063
GAISLER_ETHMAC	0x01D
GAISLER_L2TIME	0xFFD
GAISLER_L2C	0xFFE
GAISLER_PLUGPLAY	0xFFF

European Space Agency device id's	
ESA_LEON2	0x2
ESA_MCTRL	0xF
Opencores device id's	
OPENCORES_PCIBR	0x4
OPENCORES_ETHMAC	0x5

Adressage

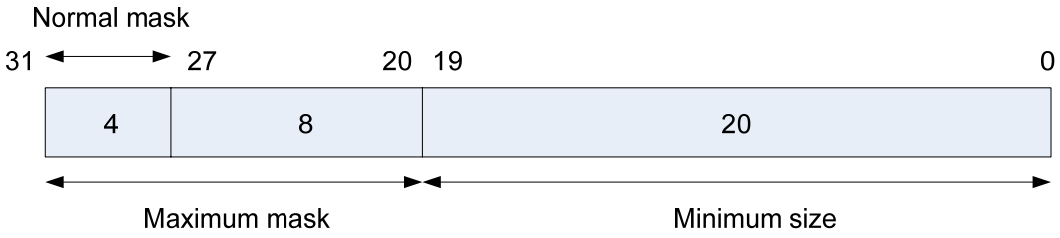
La figure suivante présente un **exemple** de la vue de la mémoire d'un système classique à bus AMBA :



Dans le système de Gaisler, la méthode pour décrire la plage d'adresse d'un slave se fait avec l'adresse de début et le masque. Le masque définit la grandeur de la plage d'adresses. Les bits 0 du masque sont les bits qui peuvent changer par rapport à l'adresse du début. Par exemple pour la mémoire ROM :
Adresse du début : 0x000
Masque : 0xE00
Plage d'adresses : 0x000 à 0x1FFF

L'adresse de début et le masque doivent être donnés dans les Bank Address Register 0 à 3 (BAR). Un slave peut donc avoir 4 adresses différentes.

L'adressage sur le bus AMBA AHB se fait sur 32 bits. Un slave peut donner un masque sur 12 bits. Ce qui veut dire que sa taille minimal est de 1 MBytes (2^20). La grandeur du masque habituel est de 4 bits (0xF000). La figure suivante explique le fonctionnement de l'adressage sur le bus AMBA – AHB :



Pour que l'arbitre sache à quel slave est destiné une opération de lecture / écriture et qu'il puisse mettre à 1 le bon bit de sélection (hsel), il doit vérifier pour chaque slave que l'équation ci-dessous soit juste :

```
((BAR.ADDR xor HADDR[31:20]) and BAR.MASK) = 0
```

Le champ TYPE (bits 0-3 du BAR) défini quel type de périphérique est connecté (voir schéma). La valeur utilisée par défaut est 2 et 0 pour les BAR ou module non-utilisés.

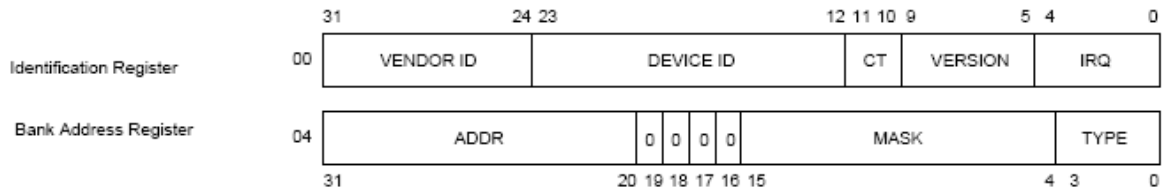
Interruptions

Les librairies de Gaisler ont ajouté un bus d'interruptions HIRQ de 32 bits en entrée et en sortie de chaque slave et master. Un master ou un slave peut donc lire et écrire sur ce vecteur d'interruption. Pour chaque élément de ce vecteur, les signaux des masters et ceux des slaves sont mis dans une porte OU. En conséquence, les masters et slaves AHB partagent le même vecteur de 32 bits d'interruptions. Le numéro IRQ du hconfig définit donc sur quel bit du vecteur le slave / master peut écrire.

Plug and play sur le bus APB

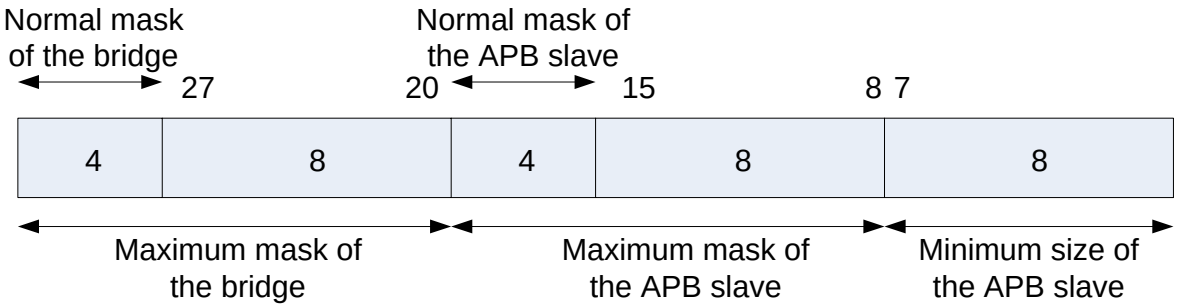
L'implémentation GRLIB du bus APB inclut le même type de mécanisme que le bus AHB pour fournir un support plug and play. Les informations plug and play pour tous les slaves APB sont référencées dans une table « read-only » qui se trouve typiquement à l'adresse 0xXXXFF000 à 0xXXXFFF00. Les X sont l'adresse du pont AHB / APB. Cela donne donc suffisamment de place pour 512 slaves.

La figure suivante présente les signaux pconfig :



Les signaux pconfig sont donc une version allégée du hconfig. Il comprend les mêmes Vendor ID, Device ID, Version et IRQ dans le registre d'indentification. Les valeurs ADDR et MASK ont la même signification que ceux du hconfig, mais ils ne sont pas placés au même endroit dans le vecteur d'adresse.

La figure suivante explique l'adressage sur le bus APB :



Les 12 bits d'adresse et de masque dans le pconfig sont donc les bits 19 à 8 des adresses des masters. Ainsi, avec 16 slaves APB qui ont une plage d'adresse égal, les bits d'adresses qui leur sont dédié sont les bits 0 à 15.
La taille minimale d'un slave APB est donc de 2^8 = 256 bytes.

Le champ TYPE (bits 0-3 du BAR) défini quel type de périphérique est connecté. La valeur utilisée par défaut est 1, 0 est réservé pour modules non-utilisés.

ANNEXE B

Fichier de configuration du circuit

```
--
-- VHDL Package Header Leon.config
--
-- Created:
--          by - valesamu.UNKNOWN (WE2783)
--          at - 09:00:58 09.03.2007
--
-- using Mentor Graphics HDL Designer(TM) 2006.1 (Build 72)
--

LIBRARY techmap;
  USE techmap.gencomp.all;

LIBRARY grlib;
  USE grlib.amba.all;

LIBRARY ieee;
  USE ieee.std_logic_1164.all;
  USE ieee.numeric_std.all;

PACKAGE config IS
    -- GENERAL CONSTANTS
    CONSTANT TECH : integer := DEFFABTECH;
    CONSTANT MEMTECH : integer := DEFMEMTECH;


    -- LEON CONSTANTS
    CONSTANT LEON_HINDEX : integer := 0;
    CONSTANT LEON_NWINDOWS : integer := 8;
    CONSTANT LEON_DSU : integer := 1;
    CONSTANT LEON_FPU : integer := 0;
    CONSTANT LEON_V8 : integer := 2;
    CONSTANT LEON_CP : integer := 0;
    CONSTANT LEON_MAC : integer := 0;
    CONSTANT LEON_PCLOW : integer := 2;
    CONSTANT LEON_NWP : integer := 0;
    CONSTANT LEON_ICEN : integer := 1;
    CONSTANT LEON_IREPL : integer := 0;
    CONSTANT LEON_ISETS : integer := 1;
    CONSTANT LEON_ILINESIZE : integer := 4;
    CONSTANT LEON_ISETSIZE : integer := 1;
    CONSTANT LEON_ISETLOCK : integer := 0;
    CONSTANT LEON_DCEN : integer := 1;
    CONSTANT LEON_DREPL : integer := 2;
    CONSTANT LEON_DSETS : integer := 1;
    CONSTANT LEON_DLINESIZE : integer := 4;
    CONSTANT LEON_DSETSIZE : integer := 1;
    CONSTANT LEON_DSETLOCK : integer := 0;
    CONSTANT LEON_DSNOOP : integer := 0;
    CONSTANT LEON_ILRAM : integer := 0;
    CONSTANT LEON_ILRAMSIZE : integer := 1;
    CONSTANT LEON_ILRAMSTART : integer := 16#8e#;
    CONSTANT LEON_DLRAM : integer := 0;
    CONSTANT LEON_DLRAMSIZE : integer := 1;
    CONSTANT LEON_DLRAMSTART : integer := 16#8f#;
    CONSTANT LEON_MMUEN : integer := 0;
    CONSTANT LEON_ITLBNUM : integer := 8;
    CONSTANT LEON_DTLBNUM : integer := 8;
    CONSTANT LEON_TLB_TYPE : integer := 1;
    CONSTANT LEON_TLB_REP : integer := 0;
    CONSTANT LEON_IDDEL : integer := 2;
    CONSTANT LEON_DISAS : integer := 0;
    CONSTANT LEON_TBUF : integer := 0;
    CONSTANT LEON_PWD : integer := 2;
    CONSTANT LEON_SVT : integer := 1;
    CONSTANT LEON_RSTADDR : integer := 0;
    CONSTANT LEON_SMP : integer := 0;
    CONSTANT LEON_CACHED : integer := 0;


    -- AHBCTRL CONSTANTS (arbitre)
    CONSTANT AHB_DEFMAST : integer := LEON_HINDEX; -- 0
    CONSTANT AHB_SPLIT : integer := 0;
    CONSTANT AHB_RROBIN : integer := 0;
```

```
    CONSTANT AHB_TIMEOUT : integer := 0;
    CONSTANT AHB_IOADDR : integer := 16#fff#;
    CONSTANT AHB_IOMASK : integer := 16#fff#;
    CONSTANT AHB_CFGADDR : integer := 16#ff0#;
    CONSTANT AHB_CFGMASK : integer := 16#ff0#;
    CONSTANT AHB_NAHBM : integer := NAHBMST;
    CONSTANT AHB_NAHBS : integer := NAHBSLV;
    CONSTANT AHB_IOEN : integer := 1;
    CONSTANT AHB_DISIRQ : integer := 0;
    CONSTANT AHB_FIXBRST : integer := 0;
    CONSTANT AHB_DEBUG : integer := 2;
    CONSTANT AHB_FPNPEN : integer := 0;
    CONSTANT AHB_ICHECK : integer := 1;
    CONSTANT AHB_DEVID : integer := 0;
    CONSTANT AHB_ENBUSMON : integer := 1; -- 0
    CONSTANT AHB_ASSERTWARN : integer := 0;
    CONSTANT AHB_ASSERTERR : integer := 0;
    CONSTANT AHB_HMSTDISABLE : integer := 0;
    CONSTANT AHB_HSLVDISABLE : integer := 0;
    CONSTANT AHB_ARBDISABLE : integer := 0;


    -- DEBUG SUPPORT UNIT CONSTANTS (dsu3)
    CONSTANT DSU_HINDEX : integer := 2;
    CONSTANT DSU_HADDR : integer := 16#C00#; -- 16#900#
    CONSTANT DSU_HMASK : integer := 16#E00#; -- 16#F00#
    CONSTANT NCPU : integer := 1;
    CONSTANT DSU_TBITS : integer := 30;
    CONSTANT DSU_TECH : integer := 0;
    CONSTANT DSU_IRQ : integer := 0;
    CONSTANT DSU_KBYTES : integer := 0;


    -- AHBUART CONSTANTS
    CONSTANT UART1_HINDEX : integer := 1;
    CONSTANT UART1_PINDEX : integer := 3;
    CONSTANT UART1_PADDR : integer := 16#300#;  -- 7
    CONSTANT UART1_PMASK : integer := 16#f00#;  -- 0


    -- APBUART CONSTANTS
    CONSTANT UART2_PINDEX : integer := 1;
    CONSTANT UART2_PADDR : integer := 1; -- 16#200#
    CONSTANT UART2_PMASK : integer := 16#fff#; -- 16#f00#
    CONSTANT UART2_CONSOLE : integer := 0;
    CONSTANT UART2_PIRQ : integer := 1;
    CONSTANT UART2_PARITY : integer := 0;
    CONSTANT UART2_FLOW : integer := 0;
    CONSTANT UART2_FIFOSIZE : integer := 4;


    -- APBCTRL CONSTANTS (pont ahb/apb)
    CONSTANT APB_HINDEX : integer := 12;
    CONSTANT APB_HADDR : integer := 16#800#; -- 0
    CONSTANT APB_HMASK : integer := 16#F00#; -- fff
    CONSTANT APB_NSLAVES : integer := NAPBSLV;
    CONSTANT APB_DEBUG : integer := 2;
    CONSTANT APB_ICHECK : integer := 1;
    CONSTANT APB_ENBUSMON : integer := 1;
    CONSTANT APB_ASSERTERR : integer := 0;
    CONSTANT APB_ASSERTWARN : integer := 0;
    CONSTANT APB_PSLVDISASBLE : integer := 0;


    -- MCTRL CONSTANTS (memory controller)
    CONSTANT MEM_HINDEX : integer := 13;
    CONSTANT MEM_PINDEX : integer := 0;
    CONSTANT MEM_ROMADDR : integer := 16#000#;
    CONSTANT MEM_ROMMASK : integer := 16#E00#;
    CONSTANT MEM_IOADDR : integer := 16#200#;
    CONSTANT MEM_IOMASK : integer := 16#E00#;
    CONSTANT MEM_RAMADDR : integer := 16#400#;
    CONSTANT MEM_RAMMASK : integer := 16#C00#;
    CONSTANT MEM_PADDR : integer := 16#100#;    -- 0
```

```
    CONSTANT MEM_PMASK : integer := 16#f00#;    -- fff
    CONSTANT MEM_WPROT : integer := 0;
    CONSTANT MEM_INVCLK : integer := 1;
    CONSTANT MEM_FAST : integer := 1;
    CONSTANT MEM_ROMASEL : integer := 28;
    CONSTANT MEM_SDRASEL : integer := 29;
    CONSTANT MEM_SRBANKS : integer := 1;        -- 4
    CONSTANT MEM_RAM8 : integer := 0;
    CONSTANT MEM_RAM16 : integer := 0;          -- 0
    CONSTANT MEM_SDEN : integer := 1;
    CONSTANT MEM_SEPBUS : integer := 0;
    CONSTANT MEM_SDBITS : integer := 32;
    CONSTANT MEM_SDLSB : integer := 2;
    CONSTANT MEM_OEPOL : integer := 0;
    CONSTANT MEM_SYNCRST : integer := 0;


    -- EXTERNAL RAMS CONSTANTS
    --CONSTANT RAMS_INDEX : integer := 14;
    CONSTANT RAMS_TACC : integer := 10; -- access time (10 ns)
    CONSTANT RAMS_FNAME : string := "..\..\software\test.srec";--"..\..\software\ram.dat";
    CONSTANT RAMS_DEPTH : integer := 15; -- ram address depth


    -- EXTERNAL ROMS CONSTANTS
    --CONSTANT ROMS_INDEX : integer := 6;
    CONSTANT ROMS_TACC : integer := 10; -- access time (10 ns)
    CONSTANT ROMS_FNAME : string := "..\..\software\test.srec"; --
    CONSTANT ROMS_DEPTH : integer := 16; -- rom address depth  13


    -- Ethernet Media Access Controller GRETH
    CONSTANT ETH_HINDEX : integer := 2;
    CONSTANT ETH_PINDEX : integer := 4;
    CONSTANT ETH_PADDR : integer := 16#400#;
    CONSTANT ETH_PMASK : integer := 16#f00#;
    CONSTANT ETH_PIRQ : integer := 2;
    CONSTANT ETH_MEMTECH : integer := MEMTECH;
    CONSTANT ETH_IFG_GAP : integer := 24;
    CONSTANT ETH_ATTEMPT_LIMIT : integer := 16;
    CONSTANT ETH_BACKOFF_LIMIT : integer := 10;
    CONSTANT ETH_SLOT_TIME : integer := 128;
    CONSTANT ETH_MDSCALER : integer := 25;
    CONSTANT ETH_ENABLE_MDIO : integer := 0;
    CONSTANT ETH_FIFOSIZE : integer := 8;
    CONSTANT ETH_NSYNC : integer := 2;
    CONSTANT ETH_EDCL : integer := 1;
    CONSTANT ETH_EDCLBUFSIZE : integer := 2;
    CONSTANT ETH_MACADDRH : integer := 16#00005e#;
    CONSTANT ETH_MACADDRL : integer := 16#000000#;
    CONSTANT ETH_IPADDRH : integer := 16#C0A8#;  --16#996D#;
    CONSTANT ETH_IPADDRL : integer := 16#0035#;  --16#05F5#;
    CONSTANT ETH_PHYRSTADR : integer := 0;
    CONSTANT ETH_RMII : integer := 0;
    CONSTANT ETH_OEPOL : integer := 0;




    ------------------------------------------------------------------------
    --                      WISHBONE CONSTANTS                            --
    ------------------------------------------------------------------------

    -- GENERAL CONSTANTS

    CONSTANT DATA_WIDTH : integer := 32;    --
    CONSTANT ADDR_WIDTH : integer := 8;   --

    CONSTANT WB_VENDORID : std_logic_vector(7 downto 0) := "00001000";
    CONSTANT WB_DEVICEID : std_logic_vector(11 downto 0) := "000000000110";
    CONSTANT WB_VERSION : std_logic_vector(4 downto 0) := "00001";
    CONSTANT WB_IRQ : std_logic_vector(4 downto 0) := "00000";
    CONSTANT WB_TYPE : std_logic_vector(3 downto 0) := "0010";
```

```
    type wb_i_type is record
      clk           : std_logic;
      rst           : std_logic;
      adr           : std_logic_vector(ADDR_WIDTH - 1 downto 0);
      dat           : std_logic_vector(DATA_WIDTH - 1 downto 0);
      sel           : std_logic_vector(15 downto 0);        -- selection
      we            : std_logic;                            -- write enable
      bte           : std_logic_vector(1 downto 0);         -- burst type extension
      cti           : std_logic_vector(2 downto 0);         -- cycle type idenfier
      stb           : std_logic;                            -- strobe (valid data
transfer)
      cyc           : std_logic;                            -- cycle (valid bus cycle)
      lock          : std_logic;                            -- transfer is uninterruptible
    end record;

    type wb_o_type is record
      dat           : std_logic_vector(31 downto 0);        -- data
      ack           : std_logic;                            -- acknowledge
      err           : std_logic;                            -- error during last transfer
      rty           : std_logic;                            -- retry (slave not ready)
    end record;
    type wb_o_vector is array (15 downto 0) of wb_o_type;

    CONSTANT FAST_WBINDEX : integer := 1;
    CONSTANT SLOW_WBINDEX : integer := 2;


    CONSTANT addr_WB_Slave0 : std_logic_vector(3 downto 0) := "0000";
    CONSTANT addr_WB_Slave1 : std_logic_vector(3 downto 0) := "0001";
    CONSTANT addr_WB_Slave2 : std_logic_vector(3 downto 0) := "0010";
    CONSTANT addr_WB_Slave3 : std_logic_vector(3 downto 0) := "0011";
    CONSTANT addr_WB_Slave4 : std_logic_vector(3 downto 0) := "0100";
    CONSTANT addr_WB_Slave5 : std_logic_vector(3 downto 0) := "0101";
    CONSTANT addr_WB_Slave6 : std_logic_vector(3 downto 0) := "0110";
    CONSTANT addr_WB_Slave7 : std_logic_vector(3 downto 0) := "0111";
    CONSTANT addr_WB_Slave8 : std_logic_vector(3 downto 0) := "1000";
    CONSTANT addr_WB_Slave9 : std_logic_vector(3 downto 0) := "1001";
    CONSTANT addr_WB_Slave10 : std_logic_vector(3 downto 0) := "1010";
    CONSTANT addr_WB_Slave11 : std_logic_vector(3 downto 0) := "1011";
    CONSTANT addr_WB_Slave12 : std_logic_vector(3 downto 0) := "1100";
    CONSTANT addr_WB_Slave13 : std_logic_vector(3 downto 0) := "1101";
    CONSTANT addr_WB_Slave14 : std_logic_vector(3 downto 0) := "1110";
    CONSTANT addr_WB_Slave15 : std_logic_vector(3 downto 0) := "1111";

    type addr_WB_SlaveVector is array (0 to 15) of std_logic_vector (3 downto 0);
    CONSTANT tabAddr_WB_Vector : addr_WB_SlaveVector := (
    0 => addr_WB_Slave0,
    1 => addr_WB_Slave1,
    2 => addr_WB_Slave2,
    3 => addr_WB_Slave3,
    4 => addr_WB_Slave4,
    5 => addr_WB_Slave5,
    6 => addr_WB_Slave6,
    7 => addr_WB_Slave7,
    8 => addr_WB_Slave8,
    9 => addr_WB_Slave9,
    10 => addr_WB_Slave10,
    11 => addr_WB_Slave11,
    12 => addr_WB_Slave12,
    13 => addr_WB_Slave13,
    14 => addr_WB_Slave14,
    15 => addr_WB_Slave15 );


    -- APB BRIDGE CONSTANTS

    CONSTANT WB_PINDEX : integer := 2;
    CONSTANT WB_PADDR : std_logic_vector(11 downto 0) :=
std_logic_vector(TO_UNSIGNED(16#200#, 12));
    CONSTANT WB_PMASK : std_logic_vector(11 downto 0) :=
std_logic_vector(TO_UNSIGNED(16#F00#, 12));

    CONSTANT WB_PCONFIG0 : amba_config_word := WB_VENDORID & WB_DEVICEID & "00" &
```

```
WB_VERSION & WB_IRQ;
    CONSTANT WB_PCONFIG1 : amba_config_word := WB_PADDR & "0000" & WB_PMASK & "0001";

    -- AHB SLAVE BRIDGE CONSTANTS


    CONSTANT WB_HINDEX : integer := 10;
    CONSTANT WB_HADDR : std_logic_vector(11 downto 0) :=
std_logic_vector(TO_UNSIGNED(16#A00#, 12));
    CONSTANT WB_HMASK : std_logic_vector(11 downto 0) :=
std_logic_vector(TO_UNSIGNED(16#F00#, 12));

    CONSTANT WB_HCONFIG0 : amba_config_word := WB_VENDORID & WB_DEVICEID & "00" &
WB_VERSION & WB_IRQ;
    CONSTANT WB_HCONFIG4 : amba_config_word := WB_HADDR & "0000" & WB_HMASK & WB_TYPE;


    -- Gaisler Research device id's
    -- GAISLER_LEON3    0x003
    -- GAISLER_LEON3DSU 0x004
    -- GAISLER_ETHAHB   0x005
    -- GAISLER_APBMST   0x006
    -- GAISLER_AHBUART  0x007
    -- GAISLER_SRCTRL   0x008
    -- GAISLER_SDCTRL   0x009
    -- GAISLER_APBUART  0x00C
    -- GAISLER_IRQMP    0x00D
    -- GAISLER_AHBRAM   0x00E
    -- GAISLER_GPTIMER  0x011
    -- GAISLER_PCITRG   0x012
    -- GAISLER_PCISBRG  0x013
    -- GAISLER_PCIFBRG  0x014
    -- GAISLER_PCITRACE 0x015
    -- GAISLER_PCIDMA   0x016
    -- GAISLER_AHBTRACE 0x017
    -- GAISLER_ETHDSU   0x018
    -- GAISLER_PIOPORT  0x01A
    -- GAISLER_AHBROM   0x01B
    -- GAISLER_AHBJTAG  0x01C
    -- GAISLER_ATACTRL  0x024
    -- GAISLER_DDRSPA   0x025
    -- GAISLER_USBEHC   0x026
    -- GAISLER_USBUHC   0x027
    -- GAISLER_KBD      0X060
    -- GAISLER_VGA      0x061
    -- GAISLER_SVGA     0x063
    -- GAISLER_ETHMAC   0x01D
    -- GAISLER_L2TIME   0xffd   internal device: leon2 timer
    -- GAISLER_L2C      0xffe   internal device: leon2compat
    -- GAISLER_PLUGPLAY 0xfff   internal device: plug & play configarea
    --
    -- European Space Agency device id's
    -- ESA_LEON2        0x2
    -- ESA_MCTRL        0xF
    --
    -- Opencores device id's
    -- OPENCORES_PCIBR  0x4
    -- OPENCORES_ETHMAC 0x5


    -- Vendor                   ID
    -- no core                  0x00
    -- Gaisler Research         0x01
    -- Pender Electronc Design  0x02
    -- European Space Agency    0x04
    -- Astrium EADS             0x06
    -- OpenChip.org             0x07
    -- OpenCores.org            0x08
    -- Gleichmann Electronics   0x10

    -- Prefetchable memory works by having blocks of memory copied out of the main memory
into a small buffer
    -- controlled by the memory chipset. This makes repeated read operations from the same
segment of memory faster.
```

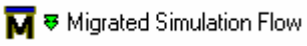
```
    -- The buffer has to be flushed if a different segment of memory is read, of course,
or if memory writes are
    -- being forcibly committed back to main memory, but it does speed read operations.
Non-prefetchable memory
    -- does not use this mechanism; all reads and writes are done directly to memory.

    -- TYPE
    -- 0001 = APB I/O space
    -- 0010 = AHB Memory space
    -- 0011 = AHB I/O space

END config;
```

ANNEXE C

Tutorial : Implémenter un circuit dans une FPGA

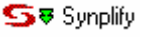
Procédure pour implémenter un circuit dans une FPGA

1. Compilation

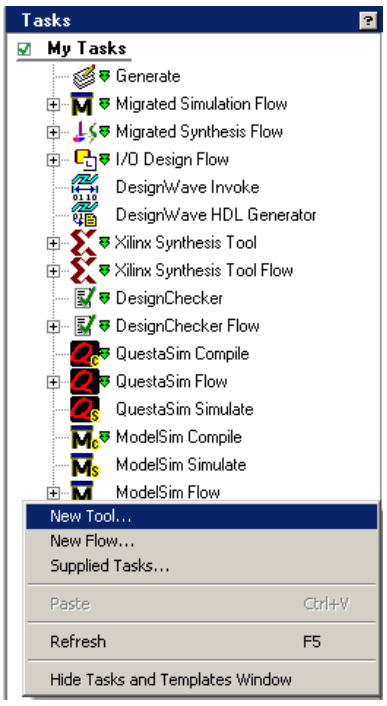
Si cela n'est pas déjà fait, compiler le top level à l'aide de l'outil 

dans HDL Designer. La compilation génère les fichiers struct et entity du top level.

2. Synthèse

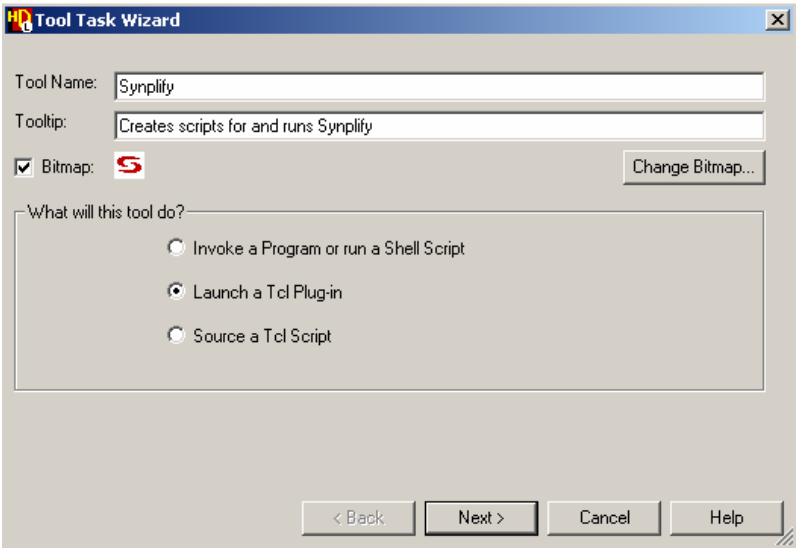
Toujours dans HDL Designer, utiliser l'outil 

pour faire la synthèse du circuit.
La synthèse sert à savoir si le circuit peut passer dans la FPGA choisie et à transformer le circuit en portes logiques. Synplify Pro est un programme efficace pour la synthèse.
Si l'outil n'est pas présent, créer un nouvel outil :

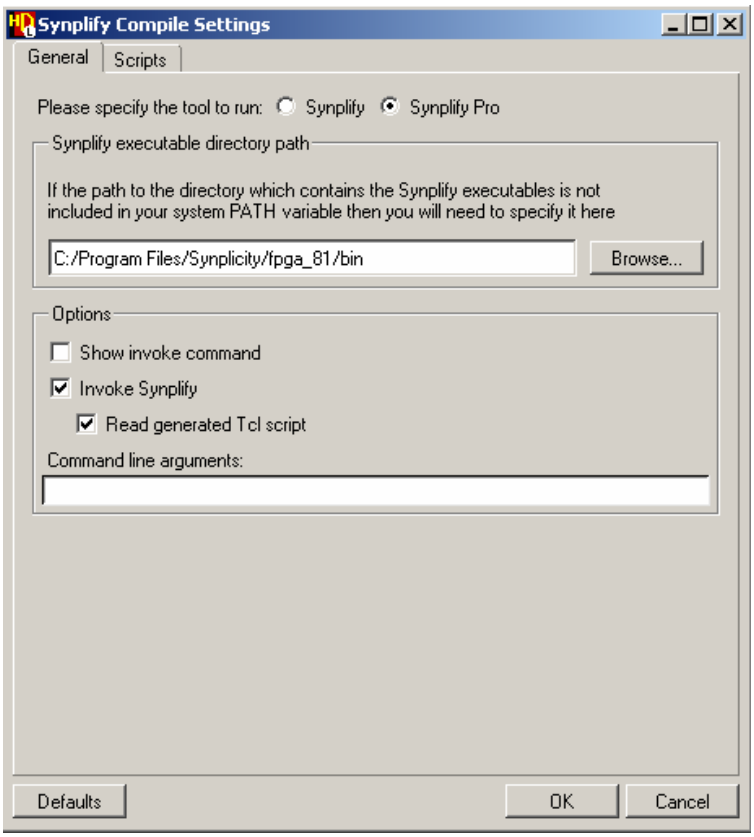


Remplir les champs comme indiqué ici :

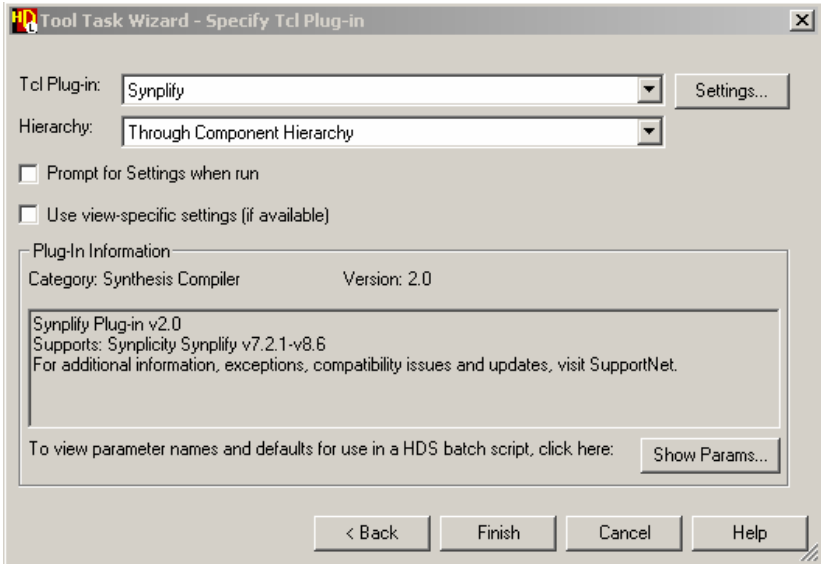
Valentini Samuel
Sion, le 23 novembre 2007



Cliquer sur Next et sur Settings pour arriver à la fenêtre qui suit :

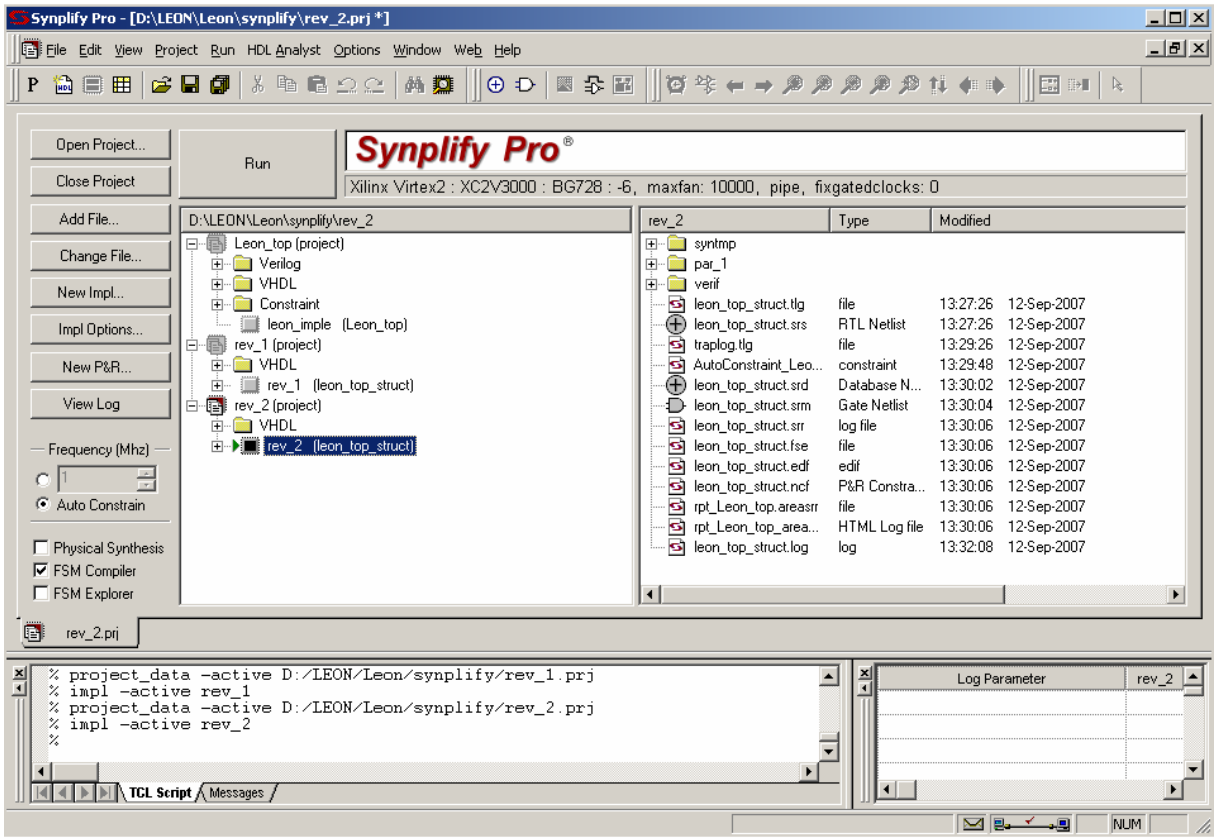


Vérifier que l'exécutable soit au chemin indiqué et valider.

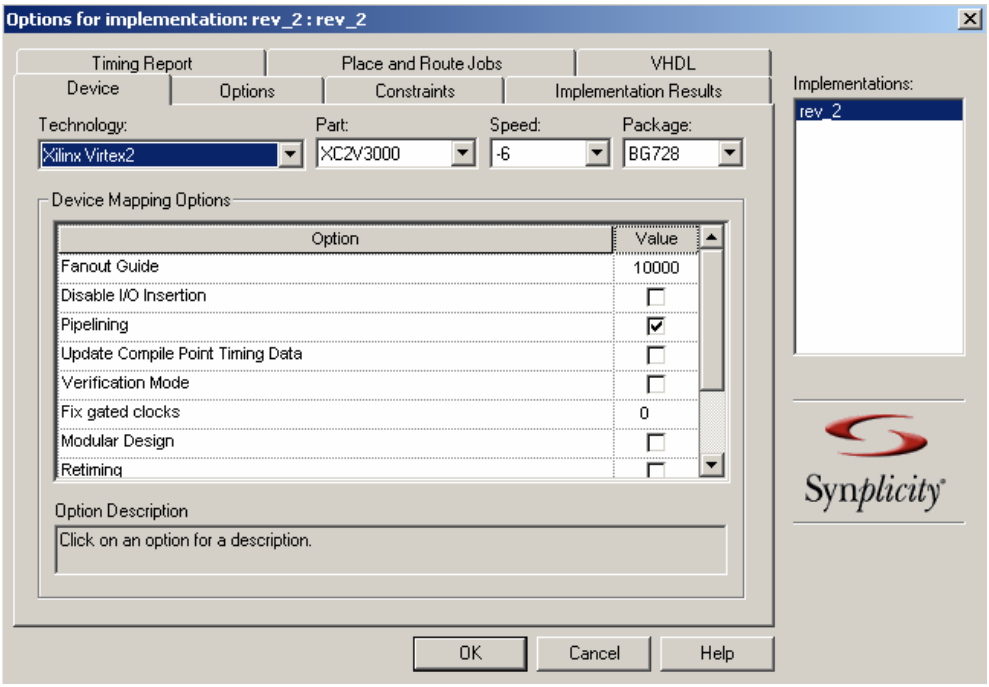


Cliquer sur Finish pour terminer.

La fenêtre du programme s'ouvre après avoir lancé Synplify.



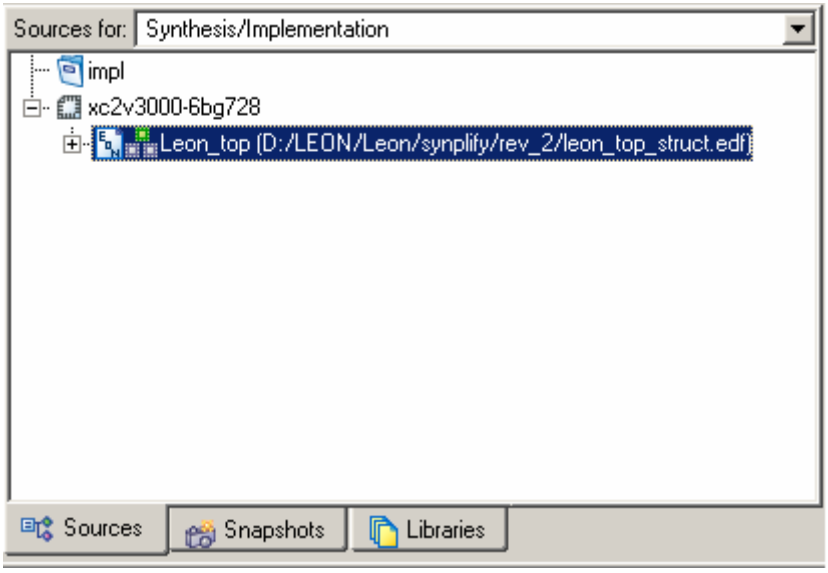
Il faut ensuite entrer le type de FPGA où sera implémenté le circuit. Pour cela, double-cliquer sur rev_x.



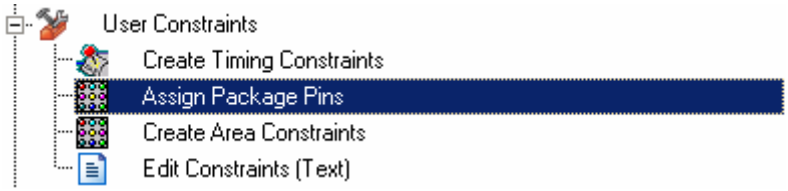
Puis, commencer la synthèse en appuyant sur le bouton Run. Si tout se passe bien, le programme a généré un fichier .edf dans le répertoire du projet.

3. Génération du fichier .bit

L'étape suivante se passe sur le programme Xilinx – ISE. Il faut en premier créer un projet dans le même répertoire que précédemment et avec la même FPGA. Il suffit ensuite d'inclure le fichier .edf créé par Synplify. Le fichier s'ajoute dans la hiérarchie des fichiers source.



Double-cliquer sur « Assign Package Pins » pour choisir quelle entrée/sortie du circuit ira sur quelle pin de la FPGA.

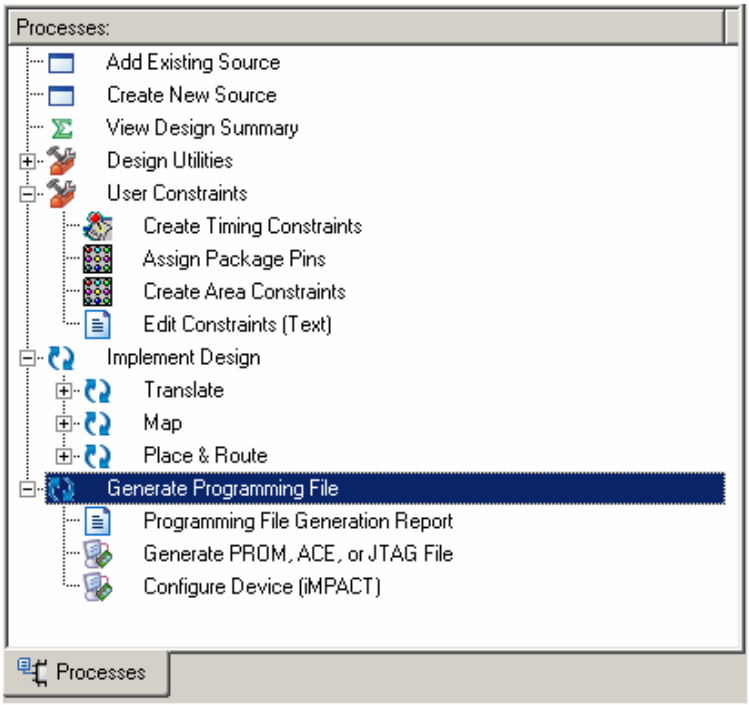


Le programme demandera alors la permission de créer un fichier .ucf où seront sauvegardées ces contraintes.

Design Object List - I/O Pins

	I/O Name	I/O Direction	Loc	Bank	I/O Std.
	address(0)	Output	C27	BANK2	
	address(1)	Output	D27	BANK2	
	address(2)	Output	D25	BANK2	
	address(3)	Output	D26	BANK2	
	address(4)	Output	E24	BANK2	
	address(5)	Output	E25	BANK2	
	address(6)	Output	E26	BANK2	
	address(7)	Output	E27	BANK2	
	address(8)	Output	F23	BANK2	
	address(9)	Output	F24	BANK2	
	address(10)	Output	F25	BANK2	
	address(11)	Output	F26	BANK2	
	address(12)	Output	F27	BANK2	
	address(13)	Output	G27	BANK2	

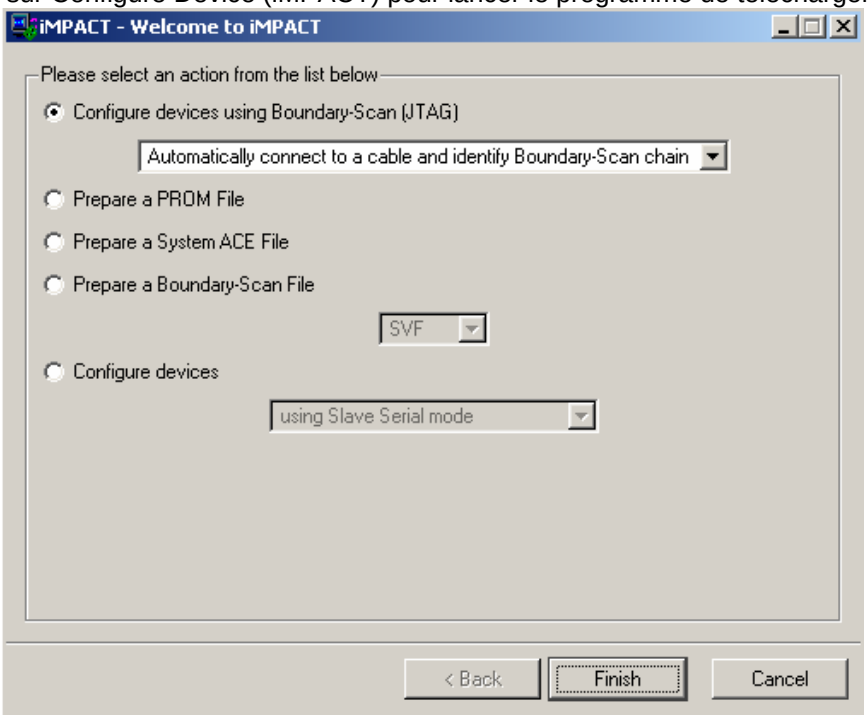
Entrer le numéro de la pin dans « Loc » pour chaque entrée/sortie.
Sauvegarder les modifications et quitter.
Double-cliquer enfin sur « Generate Programming File » pour créer le fichier .bit.



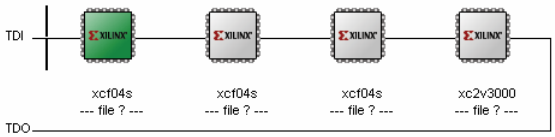
Le programme fera automatiquement les étapes intermédiaires.

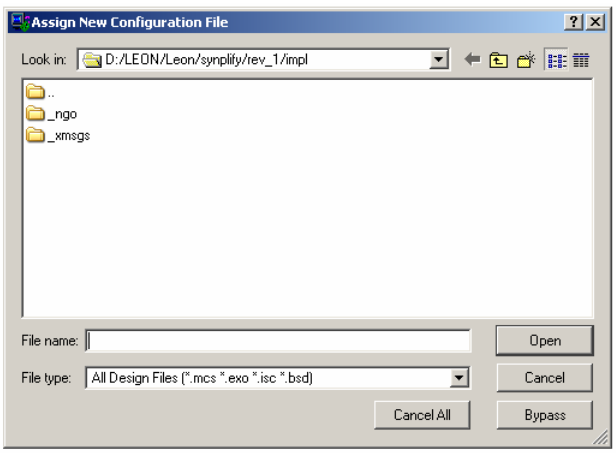
4. Chargement du fichier

Pour charger le fichier, il faut avoir correctement branché un câble JTAG possédant un Chameleon à la carte de la FPGA.
Double-cliquer sur Configure Device (iMPACT) pour lancer le programme de téléchargement.

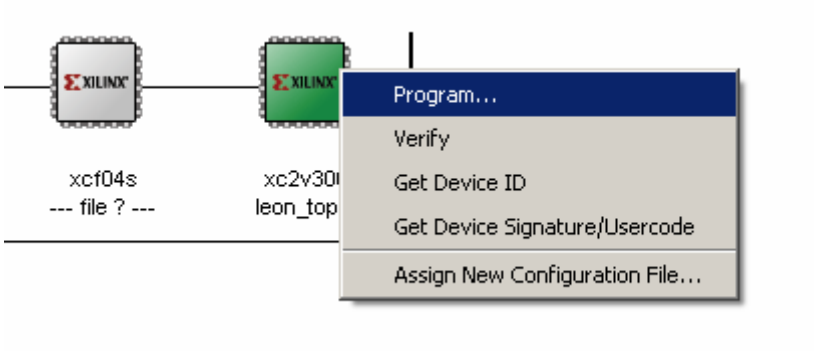


Cliquer sur Finish pour commencer la détection des composants.



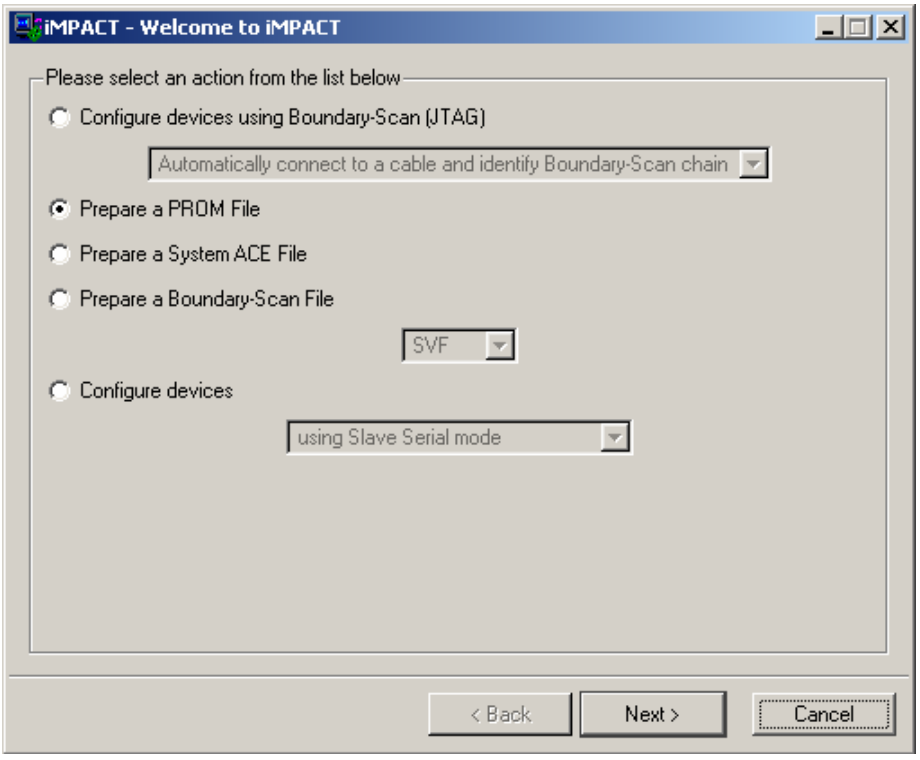


Le programme affiche les FPGA et/ou des mémoires et demande pour chacun quel fichier va y être chargé. Cliquer donc sur Cancel quand il s'agit de mémoire et sélectionner le fichier .bit pour la FPGA.
Pour commencer le téléchargement sur la FPGA, cliquer droit sur la FPGA et cliquer sur Program

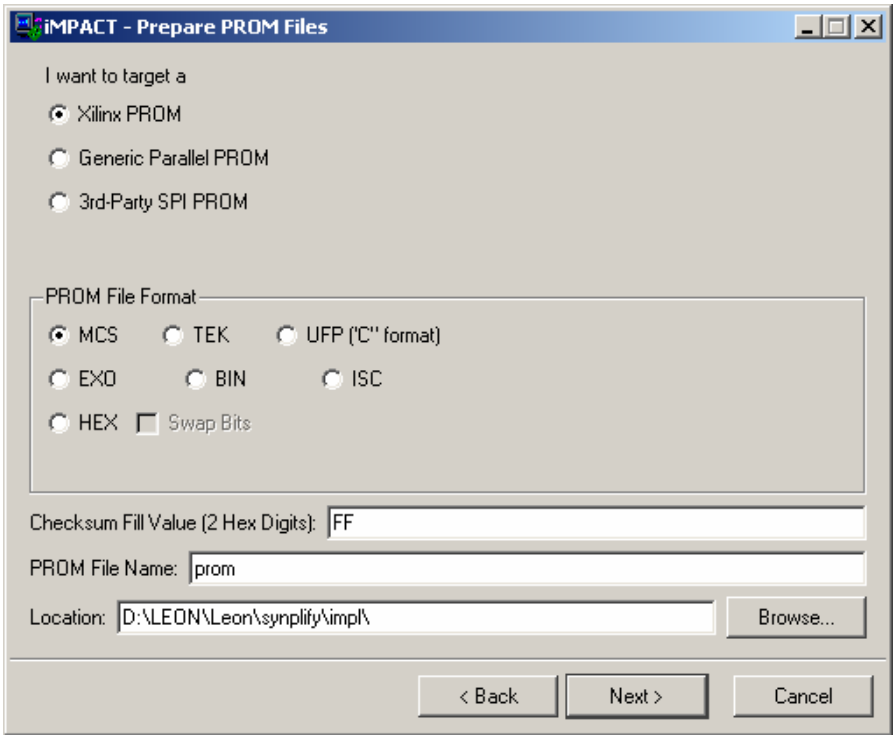


Cliquer sur OK dans la fenêtre qui s'ouvre.
Une fois le téléchargement terminé, le circuit devrait normalement tourner sur la carte.

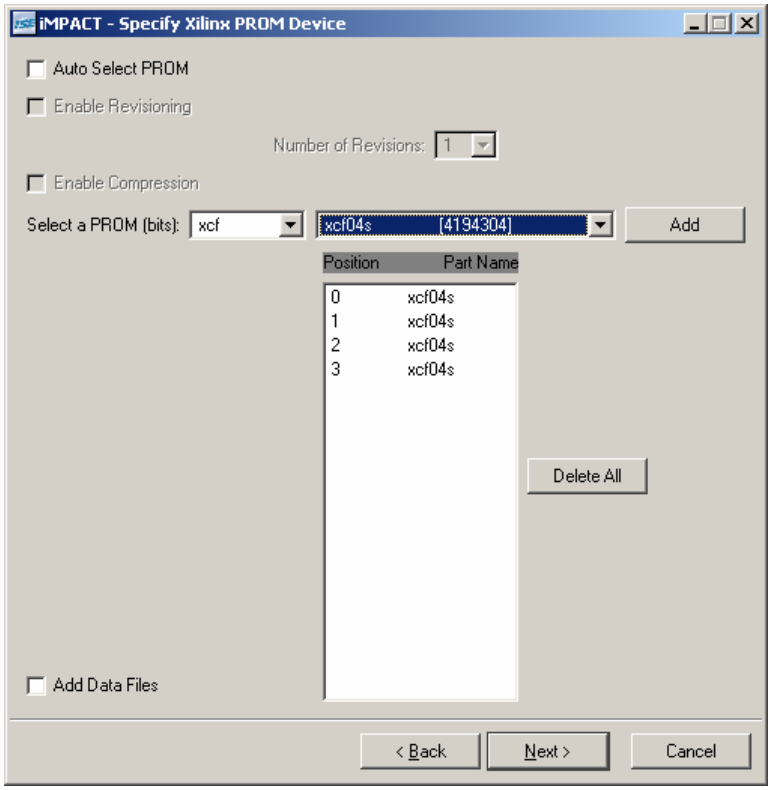
Une autre méthode consiste à programmer la FPGA via les mémoires. Avec cette méthode, la FPGA se programmera automatiquement à chaque reboot de la carte.
Il faut pour cela créer un fichier PROM

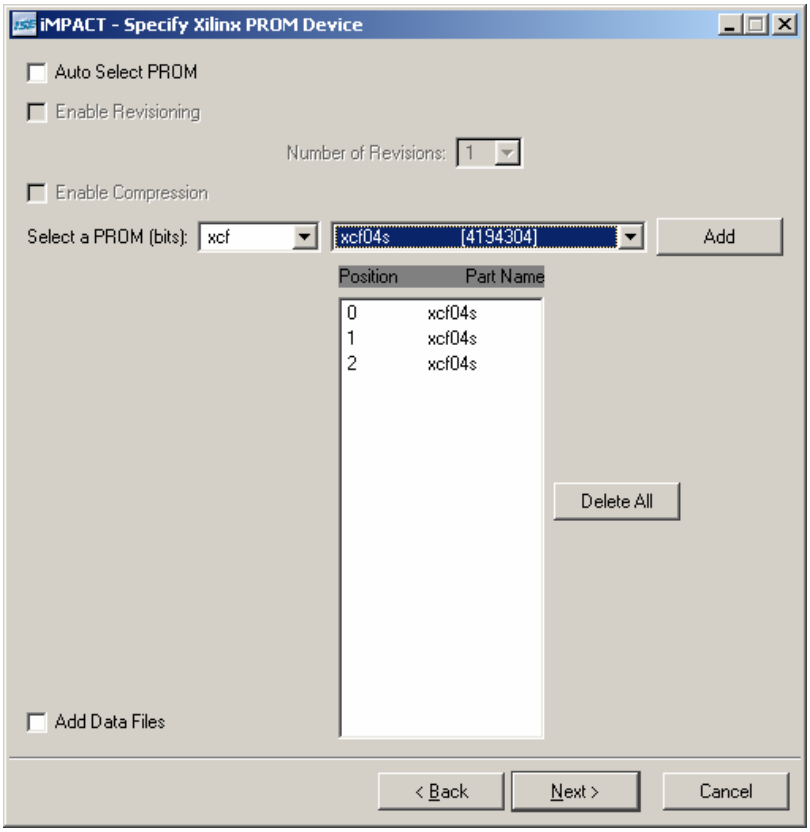


Choisir le type de FPGA et le format du fichier.

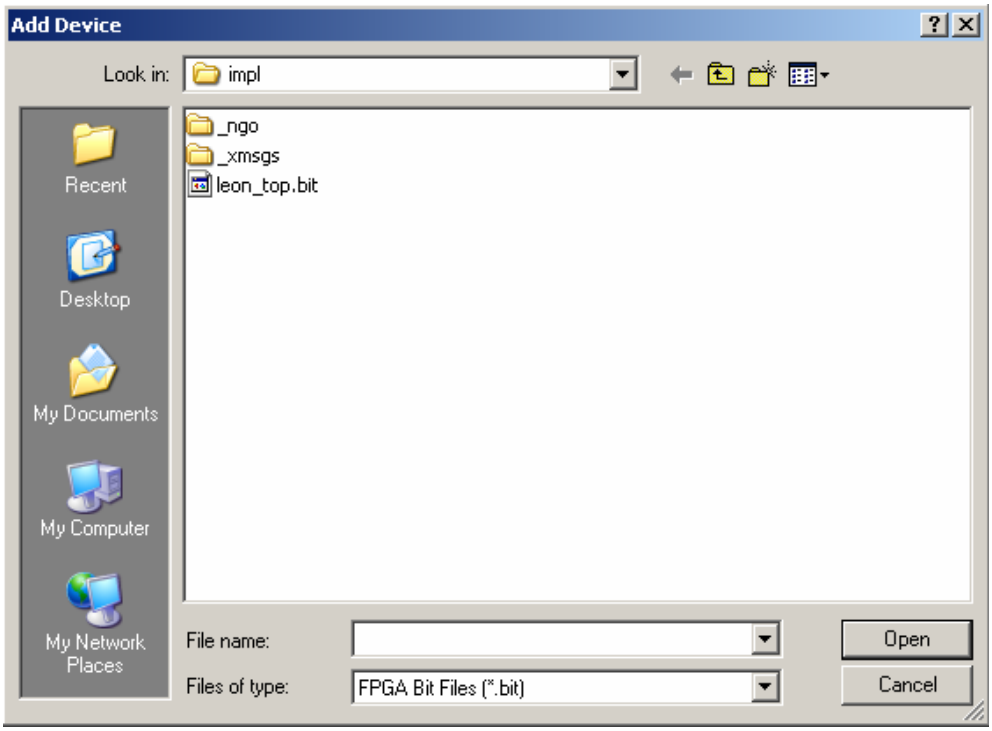


Ensuite choisir le type de mémoire.

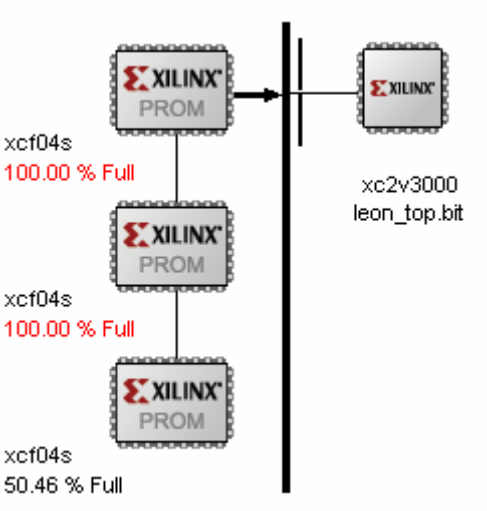




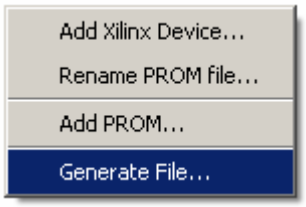
Charger le fichier .bit préalablement créé.



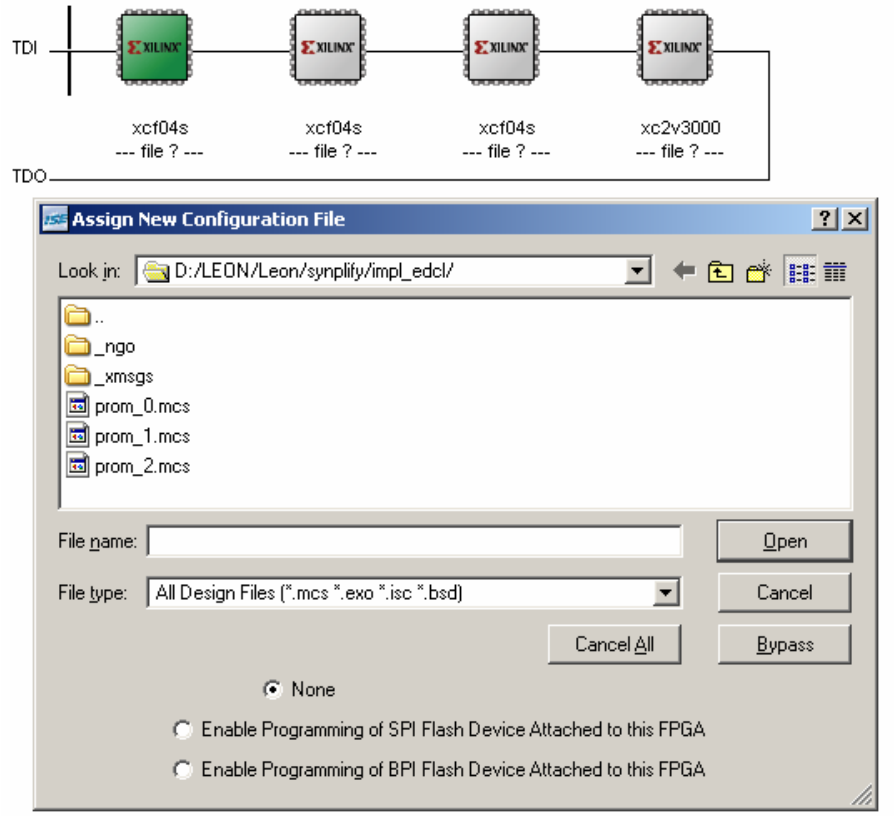
Ensuite, le programme affiche l'utilisation des mémoires :



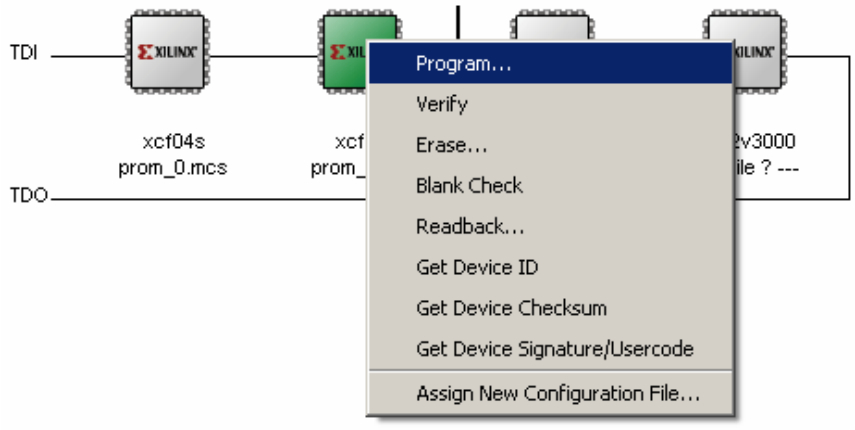
Faire un clique droit sans sélectionner de mémoire et cliquer sur Generate File :



Charger les fichiers prom généré dans les mémoires :



Pour finir, programmer les mémoires :



La FPGA devrait maintenant aller lire les mémoires à chaque démarrage de la carte et se programmer automatiquement.

ANNEXE D

Tutorial : passer de 32 à 16 bits de données de mémoire

Marche à suivre pour le changement du nombre de bits de données de la ROM de 32 à 16 bits.

Changer la génération de SRAM en 4 à 5. Par exemple :

```
rom_gen: FOR ROMS_INDEX IN 4 TO 5 GENERATE
```

Changer la largeur du bus de donnée en 15 downto 0.

Sortie de la SRAM :

```
data(7+8*(1-ROMS_INDEX+4) DOWNTO 8*(1-ROMS_INDEX+4)) :
std_logic_vector(15 DOWNTO 0)
```

Passage du bus data au bus memi :

```
memi.data (31 downto 16) <= data;
```

Passage du bus memo au bus data :

```
d_pads: for i in 0 to 15 generate
  data(i) <= memo.data(i)
  when memo.bdrive((15-i)/8) = '0' else 'Z';
end generate;
```

Signaler le changement au registre du memory controller.

```
memi.bwidth <= "01";
```

Changement des constantes du memory controller :

```
CONSTANT MEM_SRBANKS : integer := 2;
CONSTANT MEM_RAM16 : integer := 1;
```


Valentini Samuel
Sion, le 23.11.2007

ANNEXE E

Tutorial : Grmon

Tutorial : Grmon

Grmon est un programme qui peut être téléchargé sur le site de Gaisler Research et sa version de démonstration fonctionne pendant 21 jours. Il permet de débugger un programme tournant sur un processeur LEON et des circuits basé sur les librairies grlib.

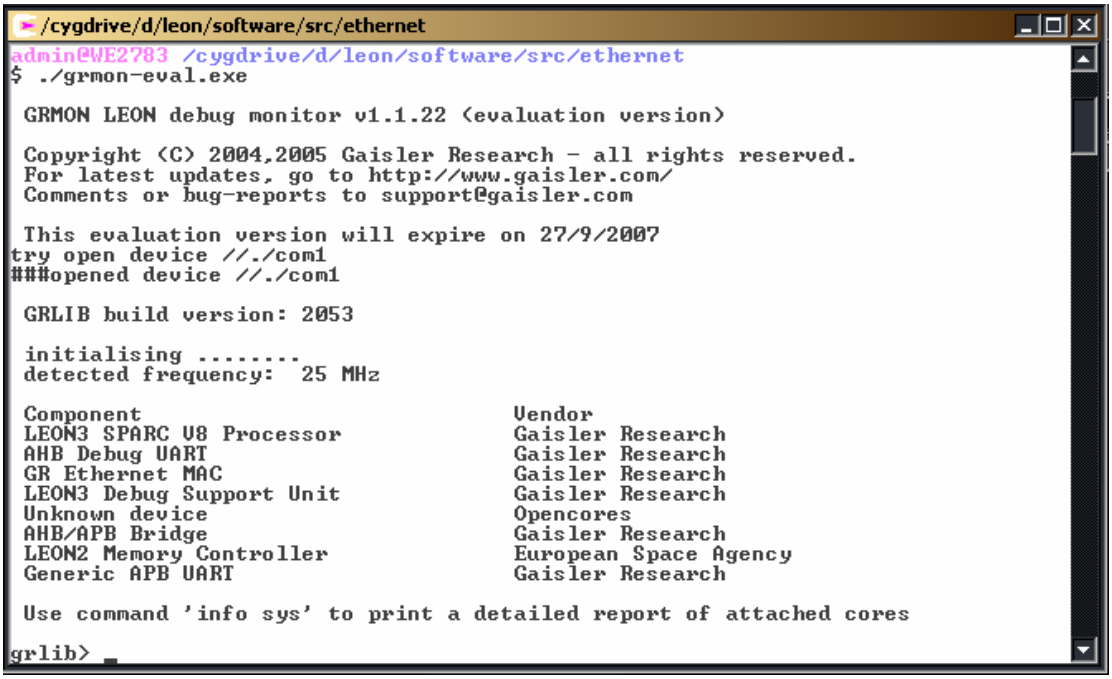
Avec ce moniteur, on peut donc :

- lire / écrire sur tous les registres et emplacements de la mémoire,
- télécharger une application pour un processeur LEON et l'exécuter,
- gestion des breakpoints et des watchpoints,
- fonctionne avec les ports USB, JTAG, RS232, PCI, Ethernet et spacewire.

Il est fournit pour les plateformes Linux-x86, Solaris-2.x, Windows (2000 et XP) et Windows avec cygwin.
GrmonRCP est une interface graphique du Grmon, basé sur l'Eclipse Rich Client Platform.

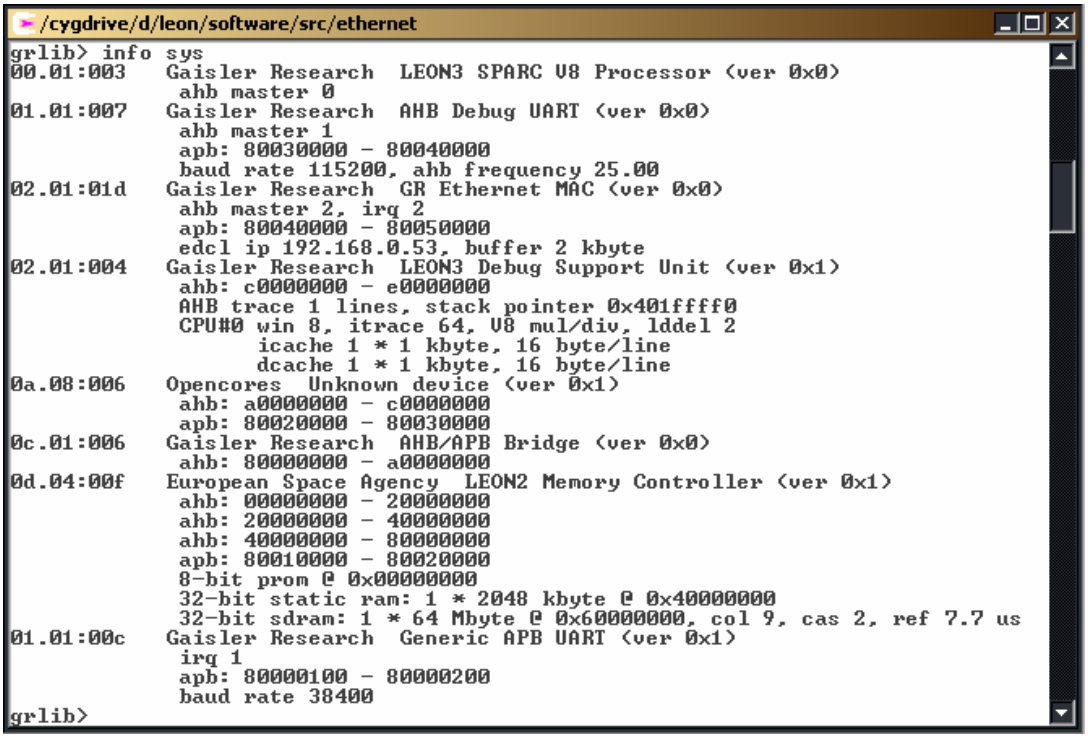
Ce document ne présente que les fonctions basiques de Grmon, le document grmon.pdf fournit avec le programme entre dans les détails de chaque commande.

La capture d'écran suivante présente le lancement du programme Grmon :



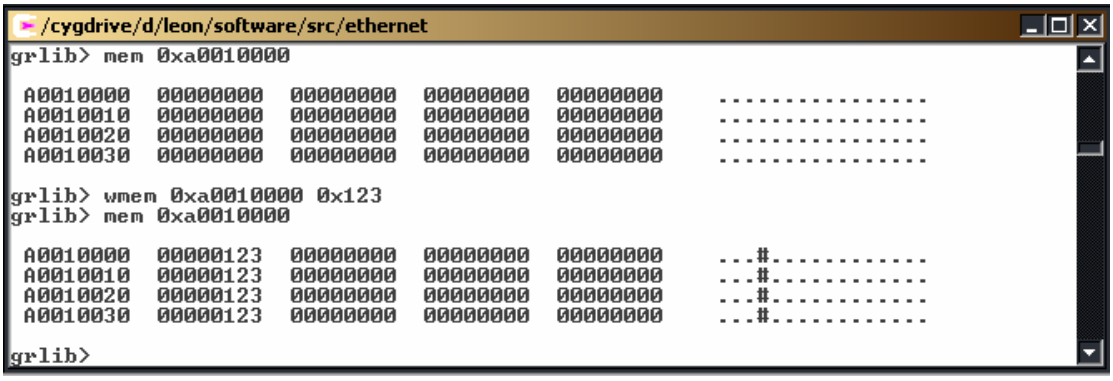
Le programme détecte donc les composants du circuit grâce aux fonctions plug and play des librairies de Gaisler. La commande info sys permet d'avoir plus d'informations sur les composants, avec notamment les plages d'adresses de chacun.

Valentini Samuel
Sion, le 23.11.2007

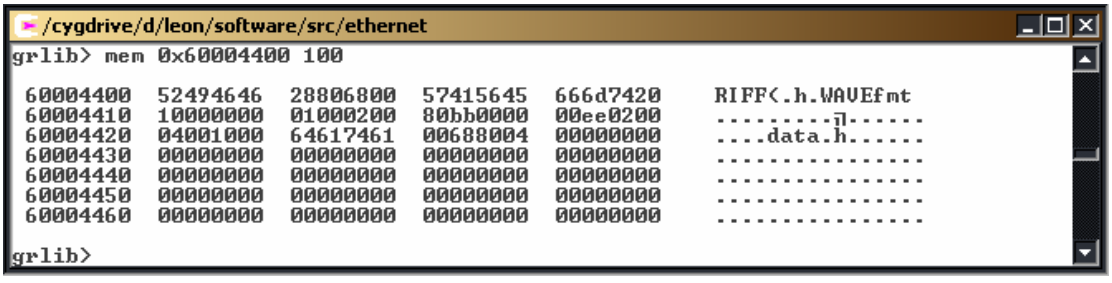


La commande help retourne la liste des commandes disponible.

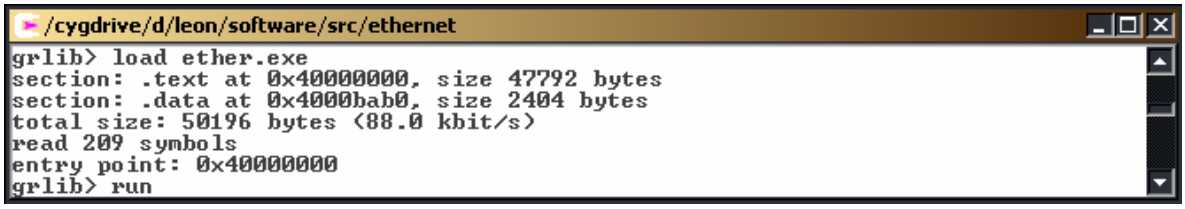
Les commandes mem et wmem sont très utiles pour vérifier qu'un composant fonctionne correctement. On peut en effet lire et écrire sur n'importe quelle adresse du circuit.



Il est possible d'afficher un nombre d'adresse souhaité avec la commande mem en spécifiant cette longueur.



Les commandes load et run permettent de charger respectivement exécuter un programme.



Comme pour tous les programmes, la combinaison Ctrl-C permet de stopper le programme. La commande cont fait reprendre le programme où il s'était arrêté.

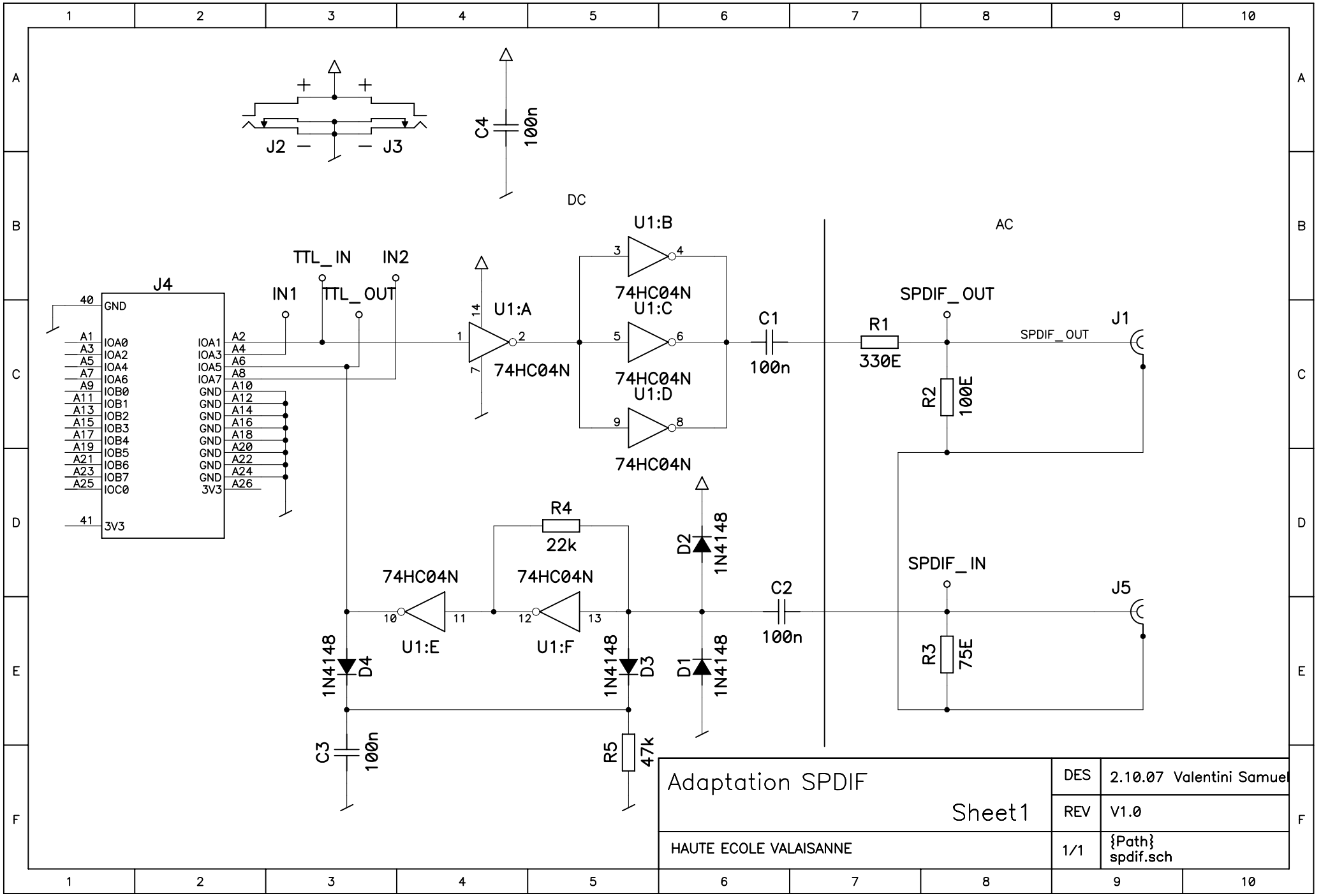
La commande break permet d'insérer un breakpoint à l'adresse de la mémoire des instructions choisies.

Et la commande delete permet de les supprimer.

Valentini Samuel
Sion, le 23.11.2007

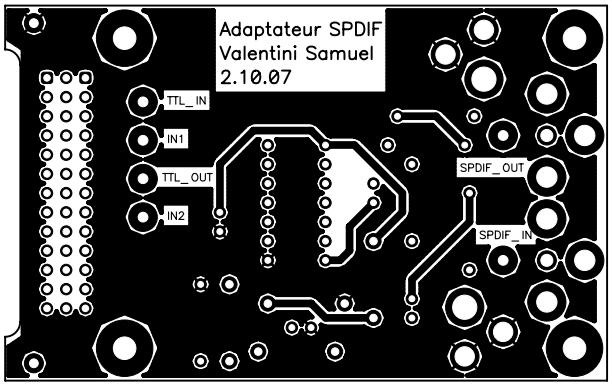
ANNEXE F1

Schéma de l’adaptateur SPDIF

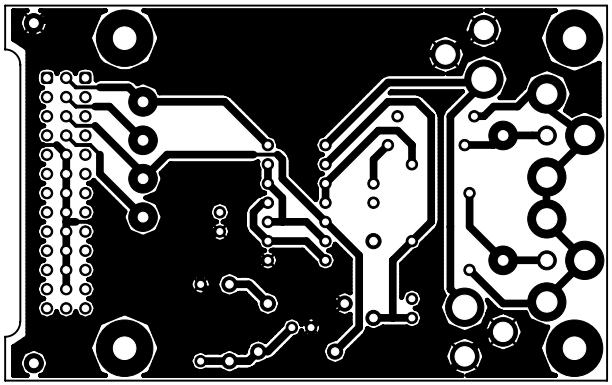

J2
J3
C4
100n
DC
AC
U1:B
74HC04N
U1:C
U1:A
74HC04N
74HC04N
U1:D
74HC04N
TTL_IN
IN2
IN1
TTL_OUT
J4
GND
IOA0
IOA1
IOA2
IOA3
IOA4
IOA5
IOA6
IOA7
IOB0
IOB1
IOB2
IOB3
IOB4
IOB5
IOB6
IOB7
IOC0
3V3
C1
100n
R1
330E
SPDIF_OUT
J1
R2
100E
R4
22k
74HC04N
74HC04N
U1:E
U1:F
D2
1N4148
D1
1N4148
D4
1N4148
D3
1N4148
C2
100n
SPDIF_IN
J5
R3
75E
C3
100n
R5
47k
Adaptation SPDIF
Sheet1
HAUTE ECOLE VALAISANNE
DES
2.10.07 Valentini Samuel
REV
V1.0
1/1
{Path}
spdif.sch

ANNEXE F2

Routage de l’adaptateur SPDIF



Adaptator SPDIF	DES	23.11.2007
	REV	V1.0
HAUTE ECOLE VALAISANNE	PCB/ spdif.pcb	



Adaptator SPDIF	DES	23.11.2007
	REV	V1.0
HAUTE ECOLE VALAISANNE	PCB/ spdif.pcb	

ANNEXE G1

Code VHDL du slave rapide

```
--
-- VHDL Architecture Leon.wb_slave_fast.arch
--
-- Created:
--          by - Valentini Samuel
--          at - 23.11.2007
--
-- using Mentor Graphics HDL Designer(TM) 2006.1 (Build 72)
--
ARCHITECTURE arch OF wb_slave_fast IS
    type memory_type is array (3 downto 0) of std_logic_vector(DATA_WIDTH - 1 downto 0);
    signal mem_fast: memory_type;

BEGIN

  read_ack: process(wb_i.stb, wb_i.cyc, wb_i.sel(WBINDEX))
  begin
    if (wb_i.sel(WBINDEX) = '1') and (wb_i.cyc = '1') and (wb_i.stb = '1') then
      wb_o.ack <= '1';
    else
      wb_o.ack <= '0';
    end if;
  end process read_ack;

  read_data: process(wb_i.stb, wb_i.adr, wb_i.cyc, wb_i.sel(WBINDEX), wb_i.we)
  begin
    if (wb_i.sel(WBINDEX) = '1') and (wb_i.cyc = '1') and (wb_i.stb = '1') and (wb_i.we =
'0') then
      wb_o.dat <= mem_fast(to_integer(unsigned(wb_i.adr)));
    else
      wb_o.dat <= (others => '-');
    end if;
  end process read_data;

  write: process(wb_i.clk)
  begin
    if rising_edge(wb_i.clk) then
      if (wb_i.sel(WBINDEX) = '1') and (wb_i.cyc = '1') and (wb_i.stb = '1') and (wb_i.we
= '1') then
        mem_fast(to_integer(unsigned(wb_i.adr))) <= wb_i.dat;
      end if;
    end if;
  end process write;

wb_o.err <= '0';
wb_o.rty <= '0';
reg0 <= mem_fast(0);

END ARCHITECTURE arch;
```

ANNEXE G2

Code VHDL du slave lent

```
--
-- VHDL Architecture Leon.wb_slave_slow.arch
--
-- Created:
--          by - admin.UNKNOWN (WE2783)
--          at - 16:17:52 19.10.2007
--
-- using Mentor Graphics HDL Designer(TM) 2006.1 (Build 72)
--
ARCHITECTURE arch OF wb_slave_slow IS
    type memory_type is array (3 downto 0) of std_logic_vector(DATA_WIDTH - 1 downto 0);

    signal mem_slow: memory_type;

    signal wait_counter: std_logic_vector(1 downto 0);

BEGIN

  slow: process(wb_i.clk, wb_i.rst)
  begin
    if wb_i.rst = '1' then
      wb_o.ack <= '0';
      wb_o.dat <= (others => '0');
      wait_counter <= (others => '0');
    elsif rising_edge(wb_i.clk) then
      if (wb_i.sel(WBINDEX) = '1') and (wb_i.cyc = '1') and (wb_i.stb = '1') then
        if wait_counter = "11" then
          wb_o.ack <= '1';
          wait_counter <= "00";
          if wb_i.we = '0' then
            wb_o.dat <= mem_slow(to_integer(unsigned(wb_i.adr)));
          else
            mem_slow(to_integer(unsigned(wb_i.adr))) <= wb_i.dat;
          end if;
        else
          wait_counter <= wait_counter + 1;
        end if;
      else
        wb_o.ack <= '0';
        wb_o.dat <= (others => '0');
      end if;
    end if;
  end process slow;

wb_o.err <= '0';
wb_o.rty <= '0';
reg0 <= mem_slow(0);

END ARCHITECTURE arch;
```

ANNEXE H

Code Perl du programme d’upload d’un fichier en protocole UDP

```
#!/usr/bin/perl

use IO::Socket;
use Time::HiRes qw(usleep);

$separator = '-' x 80;
$indent = ' ' x 2;

#-------------------------------------------------------------------------
----
# Input arguments
#
use Getopt::Std;
my %opts;
getopts('hv', \%opts);

die("\n".
    "Usage: $0 [server] [portId] [file to send]\n".
    "\n".
    "Parameters:\n".
    "${indent}-h  display this help message\n".
    "${indent}-v  verbose\n".
    "\n".
    "The message is sent to machine <server>, UDP port <portId>.\n".
    "\n".
    "More information with: perldoc $0\n".
    "\n".
    ""
   ) if ($opts{h});
$verbose = $opts{v};

my $hostName = shift(@ARGV) || 'localhost';
my $portId = shift(@ARGV) || 16000;
my $dataFileSpec = join(' ', @ARGV) || $0;


#-------------------------------------------------------------------------
----
# send UDP frame and read result with a timeout
#
sub sendUDP {

  my ($hostName, $portId, $data) = @_;

  if (length($hostName) > 0) {
                                                                  # open UDP
socket
    my $socket = IO::Socket::INET->new(Proto => 'udp')
      or return('Socket open failed.');
    my $ipAddress = inet_aton($hostName);
    my $portAddress = sockaddr_in($portId, $ipAddress);
                                                                     # send
data
    send($socket, $data, 0, $portAddress);
                                                                   # close
socket
    close($socket);
  }

  return($reply);
}
```

```
################################################################################
####
# Main program
#

print "$separator\n";
print
  "Sending file \"$dataFileSpec\" on port $portId of \"$hostName\".\n";

open(DATA, $dataFileSpec) or die "can't open $dataFileSpec: $!";
binmode(DATA);                                                    # for
DOS

while (read(DATA, my $buff, 2**10)) {
  sendUDP($hostName, $portId, $buff);
  usleep (100);  # us
}



################################################################################
####
# Documentation (access it with: perldoc <scriptname>)
#
__END__

=head1 NAME

udpSend.pl - Sends a message via UDP and awaits a reply

=head1 SYNOPSIS

udpSend.pl [server] [portId] [file to send]

=head1 DESCRIPTION

This script sends a file in the form of UDP messages.
The messages are sent to C<server> on port C<portId>.

=head1 OPTIONS

=over 8

=item B<-h>

Display a help message.

=item B<-v>

Be verbose.

=back

=head1 AUTHOR

Francois Corthay, HEVs, 2007

=cut
```

ANNEXE I

Rapport d’implémentation du circuit final

IMPL_EDCL Project Status			
Project File:	impl_edcl.ise	**Current State:**	Programming File Generated
Module Name:	Leon_top	• **Errors:**	No Errors
Target Device:	xc2v3000-6bg728	• **Warnings:**	10 Warnings
Product Version:	ISE 9.2i	• **Updated:**	mer. 21. nov. 07:47:25 2007

IMPL_EDCL Partition Summary
No partition information was found.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	3,563	28,672	12%	
Number used as Flip Flops	3,562			
Number used as Latches	1			
Number of 4 input LUTs	9,506	28,672	33%	
Logic Distribution				
Number of occupied Slices	6,024	14,336	42%	
Number of Slices containing only related logic	6,024	6,024	100%	
Number of Slices containing unrelated logic	0	6,024	0%	
Total Number of 4 input LUTs	10,324	28,672	36%	
Number used as logic	9,506			
Number used as a route-thru	201			
Number used for Dual Port RAMs	514			
Number used as 16x1 RAMs	96			
Number used as Shift registers	7			
Number of bonded IOBs	151	516	29%	
IOB Flip Flops	138			
Number of Block RAMs	11	96	11%	
Number of MULT18X18s	1	96	1%	
Number of GCLKs	3	16	18%	
Total equivalent gate count for design	897,827			
Additional JTAG gate count for IOBs	7,248			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report					
Translation Report	Current	mer. 12. sept. 10:58:02 2007	0	0	0
Map Report	Current	mer. 12. sept. 10:58:48 2007	0	2 Warnings	3 Infos
Place and Route Report	Current	mer. 12. sept. 11:07:34 2007	0	8 Warnings	4 Infos
Static Timing Report	Current	mer. 12. sept. 11:08:10 2007	0	0	3 Infos
Bitgen Report	Current	mer. 12. sept. 11:09:04 2007	0	0	0

ANNEXE J

Schéma du pont AHB / Wishbone

Leon/bridge_ahb_slv/struct1

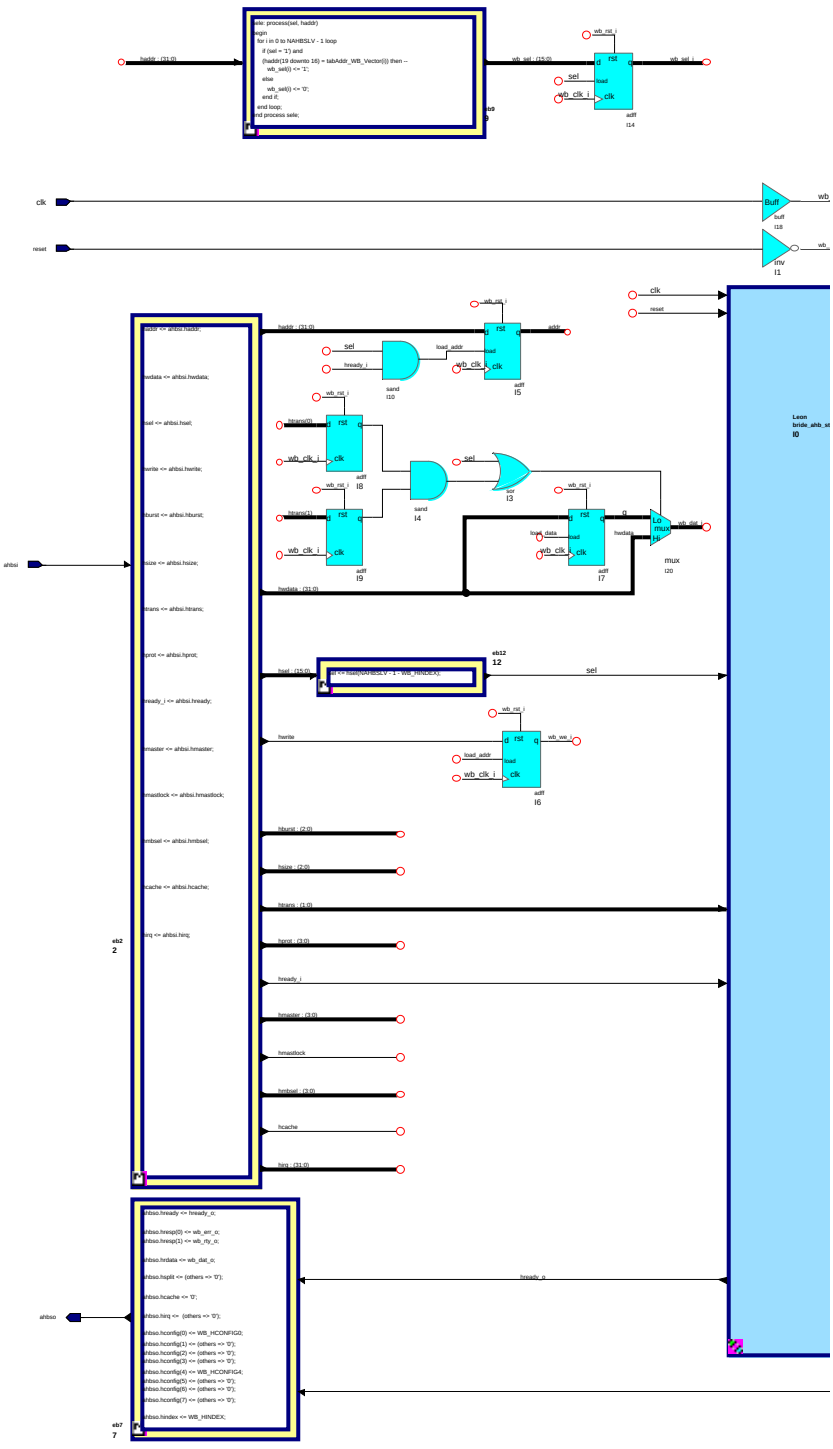
Package List
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY Leon;
USE Leon.config.all;

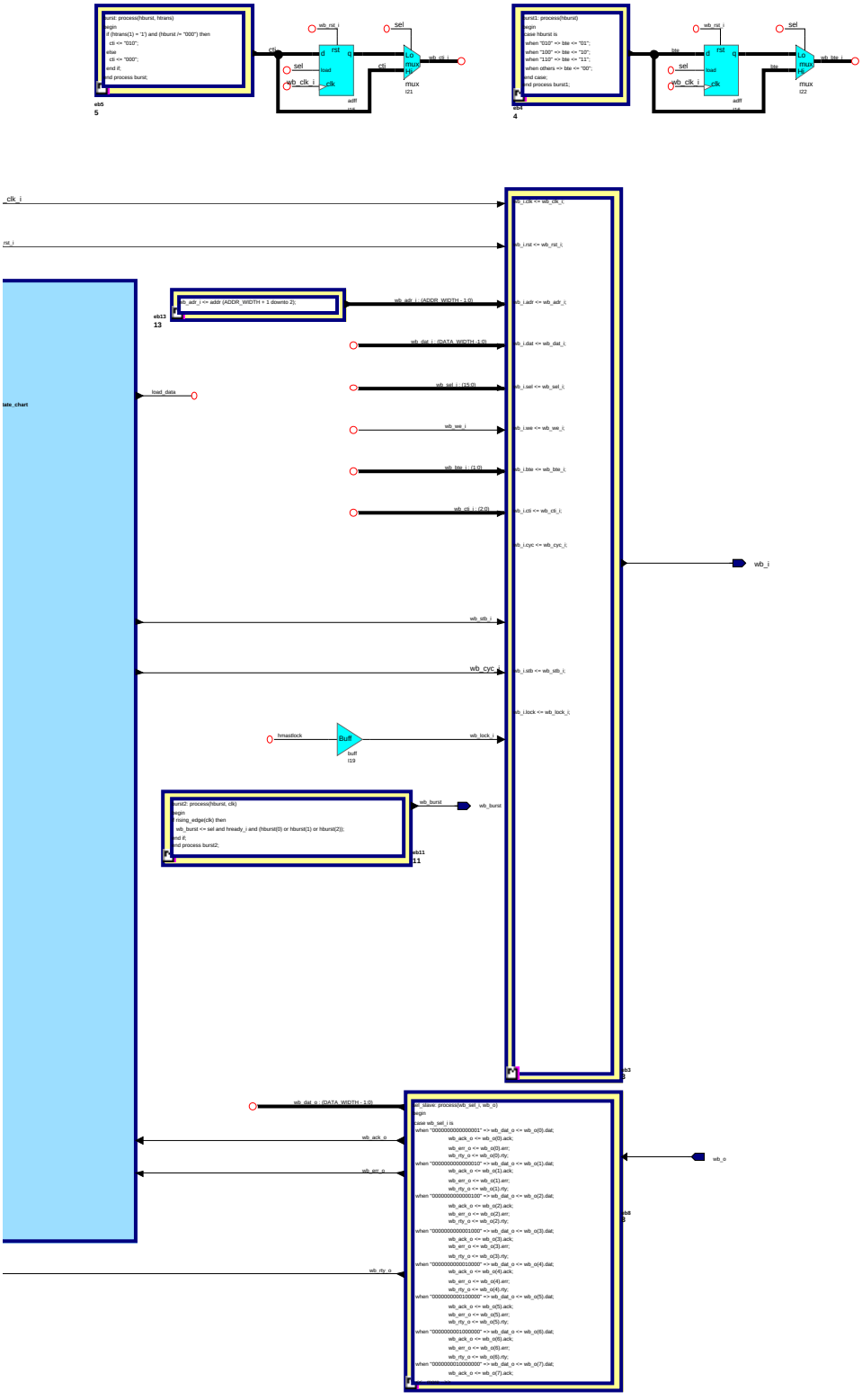
Declarations
Ports:

```
ahbsi     : ahb_slv_in_type
clk       : std_logic
reset     : std_ulogic
wb_o      : wb_o_vector
ahbso     : ahb_slv_out_type
wb_burst  : std_logic
wb_i      : wb_i_type
```

Diagram Signals:

```
SIGNAL addr      : std_logic_vector(31 DOWNTO 0)
SIGNAL bte       : std_logic_vector(1 DOWNTO 0)
SIGNAL cti       : std_logic_vector(2 DOWNTO 0)
SIGNAL din0      : std_logic_vector(1 DOWNTO 0)
SIGNAL din1      : std_logic
SIGNAL dout      : std_logic
SIGNAL haddr     : std_logic_vector(31 DOWNTO 0)
SIGNAL hburst    : std_logic_vector(2 DOWNTO 0)
SIGNAL hcache    : std_ulogic
SIGNAL hirq      : std_logic_vector(31 DOWNTO 0)
SIGNAL hmaster   : std_logic_vector(3 DOWNTO 0)
SIGNAL hmastlock : std_ulogic
SIGNAL hmbsel    : std_logic_vector(3 DOWNTO 0)
SIGNAL hprot     : std_logic_vector(3 DOWNTO 0)
SIGNAL hready_i  : std_ulogic
SIGNAL hready_o  : std_ulogic
SIGNAL hsel      : std_logic_vector(15 DOWNTO 0)
SIGNAL hsize     : std_logic_vector(2 DOWNTO 0)
SIGNAL htrans    : std_logic_vector(1 DOWNTO 0)
SIGNAL hwdata    : std_logic_vector(31 DOWNTO 0)
SIGNAL hwrite    : std_ulogic
SIGNAL load_addr : std_logic
SIGNAL load_data : std_logic
SIGNAL q         : std_logic_vector(31 DOWNTO 0)
SIGNAL q1        : std_logic
SIGNAL q2        : std_logic
SIGNAL q3        : std_logic_vector(2 DOWNTO 0)
SIGNAL sel       : std_logic
SIGNAL wb_ack_o  : std_logic
SIGNAL wb_adr_i  : std_logic_vector(ADDR_WIDTH - 1 DOWNTO 0)
SIGNAL wb_bte_i  : std_logic_vector(1 DOWNTO 0)
SIGNAL wb_clk_i  : std_logic
SIGNAL wb_cti_i  : std_logic_vector(2 DOWNTO 0)
SIGNAL wb_cyc_i  : std_logic
SIGNAL wb_dat_i  : std_logic_vector(DATA_WIDTH -1 DOWNTO 0)
SIGNAL wb_dat_o  : std_logic_vector(DATA_WIDTH - 1 DOWNTO 0)
SIGNAL wb_err_o  : std_logic
SIGNAL wb_lock_i : std_logic
SIGNAL wb_rst_i  : std_logic
SIGNAL wb_rty_o  : std_logic
SIGNAL wb_sel    : std_logic_vector(15 DOWNTO 0)
SIGNAL wb_sel_i  : std_logic_vector(15 DOWNTO 0)
SIGNAL wb_stb_i  : std_logic
SIGNAL wb_we_i   : std_logic
```





<company name>		Project:	LEON
Title:	<enter diagram title here>	<enter comments here>	
Path:	Leon/bridge_ahb_slv/struct1		
Edited:	by admin on 20 nov. 2007		

ANNEXE K

Code VHDL du pont APB / Wishbone

```
--
-- VHDL Architecture Leon.bridge_apb.struct
--
-- Created:
--          by - Valentini Samuel
--          at - 23.11.2007
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY grlib;
USE grlib.amba.all;
USE grlib.stdlib.all;
USE grlib.devices.all;
LIBRARY techmap;
  USE techmap.gencomp.all;
LIBRARY fpu;
  USE fpu.libfpu.all;
LIBRARY gaisler;
USE gaisler.memctrl.all;
USE gaisler.leon3.all;
USE gaisler.libiu.all;
USE gaisler.libcache.all;
USE gaisler.libproc3.all;
USE gaisler.arith.all;

LIBRARY std;
USE std.textio.all;

LIBRARY leon;
  USE leon.config.ALL;
LIBRARY esa;
USE esa.memoryctrl.all;
LIBRARY spdif;
USE spdif.tx_package.all;


ARCHITECTURE struct OF bridge_apb IS

   -- Architecture declarations

   -- Internal signal declarations
   SIGNAL din1        : std_logic;
   SIGNAL dout1       : std_logic;
   SIGNAL paddr       : std_logic_vector(31 DOWNTO 0);
   SIGNAL penable     : std_logic;
   SIGNAL pirq        : std_logic_vector(31 DOWNTO 0);
   SIGNAL prdata      : std_logic_vector(31 DOWNTO 0);
   SIGNAL psel        : std_logic_vector(0 TO 15);
   SIGNAL pwdata      : std_logic_vector(31 DOWNTO 0);
   SIGNAL pwrite      : std_logic;
   SIGNAL sel         : std_logic;
   SIGNAL wb_ack_o    : std_logic;
   SIGNAL wb_adr_i    : std_logic_vector(ADDR_WIDTH - 1 DOWNTO 0);
   SIGNAL wb_bte_i    : std_logic_vector(1 DOWNTO 0);
   SIGNAL wb_clk_i    : std_logic;
   SIGNAL wb_cti_i    : std_logic_vector(2 DOWNTO 0);
   SIGNAL wb_cyc_i    : std_logic;
   SIGNAL wb_dat_i    : std_logic_vector(DATA_WIDTH -1 DOWNTO 0);
   SIGNAL wb_dat_o    : std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
   SIGNAL wb_rst_i    : std_logic;
```

```
   SIGNAL wb_sel_i    : std_logic_vector(15 DOWNTO 0);
   SIGNAL wb_stb_i    : std_logic;
   SIGNAL wb_we_i     : std_logic;
   SIGNAL wish_enable : std_logic;


BEGIN

   paddr <= apbi.paddr;
   pwdata <= apbi.pwdata;
   psel <= apbi.psel;
   pwrite <= apbi.pwrite;
   penable <= apbi.penable;
   pirq <= apbi.pirq;
   sel <= psel(WB_PINDEX);
   wb_cti_i <= (others => '0');
   wb_bte_i <= (others => '0');
   wb_i.clk <= wb_clk_i;
   wb_i.rst <= wb_rst_i;
   wb_i.adr <= wb_adr_i;
   wb_i.dat <= wb_dat_i;
   wb_i.sel <= wb_sel_i;
   wb_i.we <= wb_we_i;
   wb_i.bte <= wb_bte_i;
   wb_i.cti <= wb_cti_i;
   wb_i.cyc <= wb_cyc_i;
   wb_i.stb <= wb_stb_i;
   wb_i.lock <= '0';

   sel_slave: process(wb_sel_i, wb_o)
   begin
    case wb_sel_i is
     when "0000000000000001" => wb_dat_o <= wb_o(0).dat;
                           wb_ack_o <= wb_o(0).ack;
     when "0000000000000010" => wb_dat_o <= wb_o(1).dat;
                           wb_ack_o <= wb_o(1).ack;
     when "0000000000000100" => wb_dat_o <= wb_o(2).dat;
                           wb_ack_o <= wb_o(2).ack;
     when "0000000000001000" => wb_dat_o <= wb_o(3).dat;
                           wb_ack_o <= wb_o(3).ack;
     when "0000000000010000" => wb_dat_o <= wb_o(4).dat;
                           wb_ack_o <= wb_o(4).ack;
     when "0000000000100000" => wb_dat_o <= wb_o(5).dat;
                           wb_ack_o <= wb_o(5).ack;
     when "0000000001000000" => wb_dat_o <= wb_o(6).dat;
                           wb_ack_o <= wb_o(6).ack;
     when "0000000010000000" => wb_dat_o <= wb_o(7).dat;
                           wb_ack_o <= wb_o(7).ack;
     when "0000000100000000" => wb_dat_o <= wb_o(8).dat;
                           wb_ack_o <= wb_o(8).ack;
     when "0000001000000000" => wb_dat_o <= wb_o(9).dat;
                           wb_ack_o <= wb_o(9).ack;
     when "0000010000000000" => wb_dat_o <= wb_o(10).dat;
                           wb_ack_o <= wb_o(10).ack;
     when "0000100000000000" => wb_dat_o <= wb_o(11).dat;
                           wb_ack_o <= wb_o(11).ack;
     when "0001000000000000" => wb_dat_o <= wb_o(12).dat;
                           wb_ack_o <= wb_o(12).ack;
     when "0010000000000000" => wb_dat_o <= wb_o(13).dat;
                           wb_ack_o <= wb_o(13).ack;
```

```
     when "0100000000000000" => wb_dat_o <= wb_o(14).dat;
                                wb_ack_o <= wb_o(14).ack;
     when "1000000000000000" => wb_dat_o <= wb_o(15).dat;
                                wb_ack_o <= wb_o(15).ack;
     when others =>             wb_dat_o <= (others => '0');
                                wb_ack_o <= '0';
    end case;
   end process sel_slave;

   sele: process(sel, paddr)
   begin
      for i in 0 to 15 loop
         if (sel = '1') and (paddr(15 downto 12) = tabAddr_WB_Vector(i))
then
            wb_sel_i(i) <= '1';
         else
            wb_sel_i(i) <= '0';
         end if;
      end loop;
   end process sele;

   apbo.prdata <= prdata;
   apbo.pindex <= WB_PINDEX;
   apbo.pconfig(0) <= WB_PCONFIG0;
   apbo.pconfig(1) <= WB_PCONFIG1;
   apbo.pirq <= (others => '0');

   dout1 <= sel AND din1;
   wb_clk_i <= clk;
   wb_we_i <= pwrite;
   wb_cyc_i <= wish_enable;
   wb_stb_i <= wish_enable;
   wb_adr_i <= (OTHERS => 'Z');
   wb_adr_i(7 DOWNTO 0) <= paddr(9 DOWNTO 2);
   wb_dat_i <= (OTHERS => 'Z');
   wb_dat_i(31 DOWNTO 0) <= pwdata;
   prdata <= (OTHERS => 'Z');
   prdata(31 DOWNTO 0) <= wb_dat_o;
   wb_rst_i <= NOT(reset);
   din1 <= NOT(pwrite);
   wish_enable <= penable OR dout1;

END struct;
```

ANNEXE L

Code C du programme qui reçoit un fichier par le port série et le lit

```
/*  this programm can load a music file in a SDRAM tought a serial  */
/*  connection and can play it.                                     */
/*  Valentini Samuel 23.11.07                                       */

#include <stdio.h>

#define UARTAREA (0x80000100)
#define SPDIFAHBAREA (0xA0000000)
#define SPDIFAPBAREA (0x80020000)
#define MEMCTRLAREA (0x80010000)
#define SDRAMAREA (0x60000000)
#define WB_SLOW_AREA (0xA0020000)

#define MODE (0x0)
//#define RATIO (0x3)
#define UDATEN (0x0)
#define CHSTEN (0x0)
#define TINTEN (0x0)
//#define FREQ (0x01)
#define GSTAT (0x0)
#define PREEM (0x0)
#define COPY (0x1)
#define AUDIO (0x0)

int main(int argc, char *argv[])
{

  /*********************************************************************
  *                          INITIALISATION                            *
  *********************************************************************/

  int i = 0;                // iteration variable
  int j = 0;                // iteration variable
  unsigned int perc;        // percentage of music read
  unsigned int old_perc;    // old percentage of music read
  int s [4];                // the four char get to have 32 bytes to copy
  char fourchar = 0;        // number of char got yet
  unsigned int size = 0;    // size of the music file
  unsigned int done = 0;    // size done (download or read)
  int choice;               // choice of the user
  char bufferId;            // to know wich SPDIF buffer to fill
  int mask [2] = {2, 4};    // to know wich SPDIF buffer to fill

  int freq;                 // freq of music in Hz
  char spdif_freq;          // number to put in the SPDIF register for the freq
  int ratio = 1;            // number to put in the SPDIF register for the freq
                            // and the clock

  volatile unsigned int *spdif_ahb = (volatile unsigned int *)SPDIFAHBAREA;
  volatile unsigned int *spdif_apb = (volatile unsigned int *)SPDIFAPBAREA;
  volatile unsigned int *memctrl = (volatile unsigned int *)MEMCTRLAREA;
  volatile unsigned int *sdram = (volatile unsigned int *)SDRAMAREA;
  volatile unsigned int *uart = (volatile unsigned int *)UARTAREA;
  volatile unsigned int *slow_ahb = (volatile unsigned int *)WB_SLOW_AREA;

  uart[3] = 27;                 // set uart baud rate : 115200
  uart[2] = 3;                  // set uart receiver and transmitter enable

  spdif_ahb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
                + (TINTEN << 2) + (0 << 1) + 0;    // TxConfig
  spdif_ahb[3] = 0x0;   // TxIntMask

  spdif_apb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
               + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
  spdif_apb[3] = 0x0;   // TxIntMask

  memctrl[1] = 0x96385060;      // MCFG2, set sdram options     0x95B85060
  memctrl[2] =  0xC2000;//0x185000

  while(1)
  {
    i = 0;
    printf("Welcome\n1: Load the file\n2: Play music (ahb)\n3: Play music (apb)\n");
    while(i == 0)
```

```
    {
      if((uart[1] & 0x01) == 1)
      {
        choice = uart[0];
        i = 1;
      }
    }

    if(choice == 49)
    {
      /*********************************************************************
      *                              HEADER AND SIZE                       *
      *********************************************************************/

      printf("\nPlease send the file\n");

      size = 0;
      for(i = 0; i < 11; i++)   // header (44 bytes = 11 words)
      {
        fourchar = 0;
        while(fourchar < 4)
        {
          if((uart[1] & 0x01) == 1)
          {
            s[fourchar] = uart[0];
            fourchar++;
          }
        }
        sdram[i] = ((s[0] & 0xFF) << 24) + ((s[1] & 0xFF) << 16)
                     + ((s[2] & 0xFF) << 8) + (s[3] & 0xFF);
        memctrl[1] = 0x96285060;        // MCFG2, do a LOAD-COMMAND-REGISTER
      }

      size += s[0];
      size += s[1] << 8;
      size += s[2] << 16;
      size += s[3] << 24;

      sdram[10] = size;
      memctrl[1] = 0x96285060;      // MCFG2, do a LOAD-COMMAND-REGISTER

      /*********************************************************************
      *                              FILL THE SDRAM                        *
      *********************************************************************/

      for(i = 0; i < ((size)/4-1); i++) // make the sample buffer full
      {
        fourchar = 0;
        while(fourchar < 4)
        {
          if((uart[1] & 0x01) == 1)
          {
            s[fourchar] = uart[0];      // get 32 bits
            fourchar++;
          }
        }

        sdram[i + 44] = ((s[1] & 0xFF) << 24) + ((s[0] & 0xFF) << 16)
                        + ((s[3] & 0xFF) << 8) + (s[2] & 0xFF);
        memctrl[1] = 0x96285060;    // MCFG2, do a LOAD-COMMAND-REGISTER

        if ((i % 5000) == 0)
        {
            uart[0] = 0x2E;         // print a point
        }
      }
      printf("Programming done\n");
    }

    else if(choice == 50)           // play with the ahb
    {

      slow_ahb[0] = 0x0;            // select the ahb transmitter in output
      /*********************************************************************
```

```
      *                 MAKE THE BUFFER FULL BEFORE START                *
      ******************************************************************/

      for(i = 0; i < 64; i++)   // make the sample buffer full
      {
        spdif_ahb[i * 2 + 128] = (sdram[i + 11] & 0xFFFF0000) >> 16;
        spdif_ahb[i * 2 + 129] = (sdram[i + 11] & 0x0000FFFF);
      }

      /*****************************************************************
      *                         START PLAY MUSIC                       *
      ******************************************************************/

      size = sdram[10];         // load the size
      freq = sdram[6];          // load the frequence
      s[0] = freq & 0x000000FF; // change from big endian to little endian
      s[1] = (freq & 0x0000FF00) >> 8;
      s[2] = (freq & 0x00FF0000) >> 16;
      s[3] = (freq & 0xFF000000) >> 24;

      freq = s[3] + (s[2] << 8) + (s[1] << 16) + (s[0] << 24);

      if(freq == 44100) // check the wich frequence to send to the transmitter
        spdif_freq = 0;
      else if(freq == 48000)
        spdif_freq = 1;
      else if(freq == 32000)
        spdif_freq = 2;
      else
        spdif_freq = 3;

      spdif_ahb[2] = (spdif_freq << 6) + (GSTAT << 3) + (PREEM << 2)
                       + (COPY << 1) + AUDIO;          // TxChStat
      ratio = ((25000000)/(128 * freq)) - 1;    // compute the ratio variable
      printf("\nMusic is starting\n");
      spdif_ahb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6)
                + (CHSTEN << 4) + (TINTEN << 2) + (1 << 1) + 1; // TxConfig

      perc = 0;
      old_perc = 0;
      done = 128;                   // 128 byte are already in the buffer
      bufferId = 0;
      while((done * 4) < (size))    // done is in word and size in byte
      {     // check if lower/higher buffer is empty
        if((spdif_ahb[4] & mask[bufferId]) == mask[bufferId])
        {
            spdif_ahb[4] = mask[bufferId];  // clear the flag
            for(i = 0; i < 32; i++, done++) // make the sample buffer full
            {
              spdif_ahb[i * 2 + 128 + bufferId * 64] =
                                  (sdram[done + 11] & 0xFFFF0000) >> 16;
              spdif_ahb[i * 2 + 129 + bufferId * 64] =
                                  (sdram[done + 11] & 0x0000FFFF);
            }
            bufferId = bufferId ^ 1;            // change the selected buffer

            perc = (int) (100 * (done * 4)/size);// compute the actual percentage
        }
        if(perc != old_perc)            // check if percentage has changed
        {
            printf(" %i%%", perc);
            uart[0] = 0x0D;
            old_perc = perc;
        }
      }
      printf("\nMusic Ended\n");
      spdif_ahb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6)
            + (CHSTEN << 4) + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
    }

    else if(choice == 51)
    {

      slow_ahb[0] = 0x1;            // select the apb transmitter in output
```

```
      /*****************************************************************
      *                  MAKE THE BUFFER FULL BEFORE START                *
      *****************************************************************/

      for(i = 0; i < 64; i++)   // make the sample buffer full
      {
        spdif_apb[i * 2 + 128] = (sdram[i + 11] & 0xFFFF0000) >> 16;
        spdif_apb[i * 2 + 129] = (sdram[i + 11] & 0x0000FFFF);
      }

      /*****************************************************************
      *                          START PLAY MUSIC                       *
      *****************************************************************/

      size = sdram[10];         // load the size
      freq = sdram[6];          // load the frequence
      s[0] = freq & 0x000000FF;     // change from big endian to little endian
      s[1] = (freq & 0x0000FF00) >> 8;
      s[2] = (freq & 0x00FF0000) >> 16;
      s[3] = (freq & 0xFF000000) >> 24;

      freq = s[3] + (s[2] << 8) + (s[1] << 16) + (s[0] << 24);

      if(freq == 44100)     // check the wich frequence to send to the transmitter
        spdif_freq = 0;
      else if(freq == 48000)
        spdif_freq = 1;
      else if(freq == 32000)
        spdif_freq = 2;
      else
        spdif_freq = 3;

      spdif_apb[2] = (spdif_freq << 6) + (GSTAT << 3) + (PREEM << 2)
                   + (COPY << 1) + AUDIO;      // TxChStat
      ratio = ((25000000)/(128 * freq)) - 1;    // compute the ratio variable
      printf("\nMusic is starting\n");
      spdif_apb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
                   + (TINTEN << 2) + (1 << 1) + 1; // TxConfig

      perc = 0;
      old_perc = 0;
      done = 128;                   // 128 byte are already in the buffer
      bufferId = 0;
      while((done * 4) < (size))    // done is in word and size in byte
      {
        if((spdif_apb[4] & mask[bufferId]) == mask[bufferId])
        {       // check if lower/higher buffer is empty
            spdif_apb[4] = mask[bufferId];      // clear the flag
            for(i = 0; i < 32; i++, done++)     // make the sample buffer full
            {
              spdif_apb[i * 2 + 128 + bufferId * 64] =
                                  (sdram[done + 11] & 0xFFFF0000) >> 16;
              spdif_apb[i * 2 + 129 + bufferId * 64] =
                                  (sdram[done + 11] & 0x0000FFFF);
            }
            bufferId = bufferId ^ 1;    // change the selected buffer

            perc = (int) (100 * (done * 4)/size); // compute the actual percentage
        }
        if(perc != old_perc)               // check if percentage has changed
        {
            printf(" %u%%", perc);
            uart[0] = 0x0D;
            old_perc = perc;
        }
      }
      printf("\nMusic Ended\n");
      spdif_apb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
                   + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
    }
  }
  return 0;
}
```

ANNEXE M

Code C du programme qui reçoit un fichier par le port Ethernet et le lit

```
/*  this programm can load a music file in a SDRAM tought a Ethernet*/
/*  connection and can play it.                                     */
/*  Valentini Samuel 23.11.07                                       */

#include <stdio.h>

#define UARTAREA (0x80000100)
#define SPDIFAHBAREA (0xA0000000)
#define SPDIFAPBAREA (0x80020000)
#define MEMCTRLAREA (0x80010000)
#define SDRAMAREA (0x60000000)
#define WB_SLOW_AREA (0xA0020000)
#define ETHERNET_AREA (0x80040000)

#define MODE (0x0)
//#define RATIO (0x3)
#define UDATEN (0x0)
#define CHSTEN (0x0)
#define TINTEN (0x0)
//#define FREQ (0x01)
#define GSTAT (0x0)
#define PREEM (0x0)
#define COPY (0x1)
#define AUDIO (0x0)

int main(int argc, char *argv[])
{

  /*********************************************************************
  *                          INITIALISATION                            *
  *********************************************************************/

  int i = 0;                // iteration variable
  int j = 0;                // iteration variable
  unsigned int perc;        // percentage of music read
  unsigned int old_perc;    // old percentage of music read
  unsigned int size = 0;    // size of the file
  unsigned int done = 0;    // size done (download or read)
  int choice;               // choice of the user
  char bufferId;            //
  int mask [2] = {2, 4};    //
  int pack_size;            // size of the packet
  int offset;               // offset of the data in a packet

  int freq;                 // freq of music in Hz
  char spdif_freq;          // number to put in the SPDIF register for the freq
  int ratio = 1;            // number to put in the SPDIF register for the freq
                            // and the clock

  volatile unsigned int *spdif_ahb = (volatile unsigned int *)SPDIFAHBAREA;
  volatile unsigned int *spdif_apb = (volatile unsigned int *)SPDIFAPBAREA;
  volatile unsigned int *memctrl = (volatile unsigned int *)MEMCTRLAREA;
  volatile unsigned int *sdram = (volatile unsigned int *)SDRAMAREA;
  volatile unsigned int *uart = (volatile unsigned int *)UARTAREA;
  volatile unsigned int *slow_ahb = (volatile unsigned int *)WB_SLOW_AREA;
  volatile unsigned int *ether = (volatile unsigned int *)ETHERNET_AREA;

  volatile unsigned short *pack = (volatile unsigned short *)SDRAMAREA + 512;
  volatile unsigned int *data = (volatile unsigned int *)SDRAMAREA + 4352;

  uart[3] = 27;                 // set uart baud rate : 115200
  uart[2] = 3;                  // set uart receiver and transmitter enable

  printf("First write\n");
  spdif_ahb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
               + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
  spdif_ahb[3] = 0x0;   // TxIntMask

  spdif_apb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
               + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
  spdif_apb[3] = 0x0;   // TxIntMask
  printf("%08X, %08X\n", spdif_ahb[1], spdif_ahb[2]);

  memctrl[1] = 0x96385060;      // MCFG2, set sdram options     0x95B85060
```

```
  printf("mcfg2 : %08X\n", memctrl[1]);

  memctrl[2] =  0xC2000;//0x185000

  ether[0] = 0x0;

  while(1)
  {
    i = 0;
    printf("Welcome\n1: Load the file\n2: Play music (ahb)\n3: Play music (apb)\n");
    while(i == 0)
    {
      if((uart[1] & 0x01) == 1)
      {
        choice = uart[0];
        i = 1;
      }
    }

    if(choice == 49)
    {

      /*************************************************************************
      *                              HEADER AND SIZE                           *
      *************************************************************************/

      ether[0] = 0x0;
      ether[1] = 0xFF;          // clear flags
      ether[2] = 0x5E;          // MAC address MSB
      ether[3] = 0x0;           // MAC address LSB
      ether[6] = SDRAMAREA;     // descriptor pointer

      offset = 21;

      ether[7] = 0xC0A80035;

      for(i = 0; i < 8; i++)
      {
        sdram[1 + i * 2] = (unsigned int)pack + i * 0x800;  // data address
        memctrl[1] = 0x96285060;    // MCFG2, do a LOAD-COMMAND-REGISTER
        sdram[i * 2] = 0x800;
        memctrl[1] = 0x96285060;    // MCFG2, do a LOAD-COMMAND-REGISTER
      }
      ether[0] = 0x2;           // start reception
      printf("\nPlease send the file\n");
      done = 0;
      bufferId = 0;

      while(done < 44)  // header (44 bytes)
      {
        if(((ether[1] & 0x04) == 4)/* | (sdram[bufferId * 2] != 0x800)*/)
        {
          ether[1] = 4;             // clear the flag
          if(ether[6] == (SDRAMAREA + 0x40))
            ether[6] = SDRAMAREA;
          ether[0] = 0x2;           // start reception

          if((pack[bufferId * 0x400] == 0xFFFF) & (pack[1 + bufferId * 0x400] == 0xFFFF)
                & (pack[2 + bufferId * 0x400] == 0xFFFF))  //check if it is a broadcast
          {
            uart[0] = 0x2D;         // print a -
          }
          else
          {
            pack_size = sdram[bufferId * 2] & 0x7FF;    // get the length of the packet
            printf("\n%d %d, %08X\n", pack[0x15 + bufferId * 0x400],
                    pack[0x16 + bufferId * 0x400], pack_size);  //print the size
//          if(pack[10 + bufferId * 0x400] == 0x2000)
//            offset = 21;
//          else
//            offset = 17;
            i = offset;
            while(i < (pack_size / 2))
            {
```

```
              data[(done + (i - offset) * 2)/4] = (pack[i + bufferId * 0x400] << 16)
                        + pack[i + 1 + bufferId * 0x400];   // copy data
              memctrl[1] = 0x96285060;  // MCFG2, do a LOAD-COMMAND-REGISTER
              i += 2;
            }
            done += pack_size - (offset * 2); //update done with the length of the packet
            uart[0] = 0x2E;         // print a point
          }
          sdram[bufferId * 2] = 0x800;  //make the descriptor ready
          memctrl[1] = 0x96285060;  // MCFG2, do a LOAD-COMMAND-REGISTER
          bufferId++;
          if(bufferId == 8)
            bufferId = 0;
        }
        ether[0] = 0x2;         // start reception
        if(ether[6] == (SDRAMAREA + 0x40))
            ether[6] = SDRAMAREA;
      }
      printf("\nHeader done\n");
      size = data[10];          // get the size
      s[0] = size & 0x000000FF;         // change from big endian to little endian
      s[1] = (size & 0x0000FF00) >> 8;
      s[2] = (size & 0x00FF0000) >> 16;
      s[3] = (size & 0xFF000000) >> 24;

      size = s[3] + (s[2] << 8) + (s[1] << 16) + (s[0] << 24);

      printf("\nsize : %08X\n", size);

      data[10] = size;          // save the size
      memctrl[1] = 0x96285060;          // MCFG2, do a LOAD-COMMAND-REGISTER

      /*********************************************************************
      *                             FILL THE SDRAM                         *
      *********************************************************************/

      while(done < (size + 44)) // make the sample buffer full
      {
        if(((ether[1] & 0x04) == 4)/* | (sdram[bufferId * 2] != 0x800)*/)
        {
          ether[1] = 4;             // clear the flag
          if(ether[6] == (SDRAMAREA + 0x40))
            ether[6] = SDRAMAREA;
          ether[0] = 0x2;           // start reception

          if((pack[bufferId * 0x400] == 0xFFFF) & (pack[1 + bufferId * 0x400] == 0xFFFF)
                & (pack[2 + bufferId * 0x400] == 0xFFFF))  //check if it is a broadcast
          {
            uart[0] = 0x2D;         // print a -
          }
          else
          {
            pack_size = sdram[bufferId * 2] & 0x7FF;    // get the length of the packet
//          if(pack[10 + bufferId * 0x400] == 0x2000)
//            offset = 21;
//          else
//            offset = 17;
            i = offset;
            while(i < (pack_size / 2))
            { // change from big endian to little endian
              data[(done + (i - offset) * 2)/4] =
              ((pack[i + bufferId * 0x400] & 0xFF00) << 8)
              + ((pack[i + bufferId * 0x400] & 0x00FF) << 24)
              + ((pack[i + 1 + bufferId * 0x400] & 0xFF00) >> 8)
              + ((pack[i + 1 + bufferId * 0x400] & 0x00FF) << 8);
              memctrl[1] = 0x96285060;  // MCFG2, do a LOAD-COMMAND-REGISTER
              i += 2;
            }
            done += pack_size - (offset * 2);//update done with the length of the packet
            uart[0] = 0x2E;         // print a point
          }
          sdram[bufferId * 2] = 0x800;  //make the descriptor ready
          memctrl[1] = 0x96285060;  // MCFG2, do a LOAD-COMMAND-REGISTER
          bufferId++;
```

```
        if(bufferId == 8)
          bufferId = 0;
      }
      ether[0] = 0x2;         // start reception
      if(ether[6] == (SDRAMAREA + 0x40))
          ether[6] = SDRAMAREA;
    }
    ether[0] = 0x0;           // end reception
    printf("Programming done\n");
  }

  else if(choice == 50)              // play with the ahb
  {

    slow_ahb[0] = 0x0;            // select the ahb transmitter in output
    /*************************************************************************
    *                  MAKE THE BUFFER FULL BEFORE START                     *
    *************************************************************************/

    for(i = 0; i < 64; i++)   // make the sample buffer full
    {
      spdif_ahb[i * 2 + 128] = (data[i + 11] & 0xFFFF0000) >> 16;
      spdif_ahb[i * 2 + 129] = (data[i + 11] & 0x0000FFFF);
    }


    /*************************************************************************
    *                          START PLAY MUSIC                              *
    *************************************************************************/

    size = data[10];          // load the size
    freq = data[6];           // load the frequence
    s[0] = freq & 0x000000FF;         // change from big endian to little endian
    s[1] = (freq & 0x0000FF00) >> 8;
    s[2] = (freq & 0x00FF0000) >> 16;
    s[3] = (freq & 0xFF000000) >> 24;

    freq = s[3] + (s[2] << 8) + (s[1] << 16) + (s[0] << 24);

    if(freq == 44100)         // check the wich frequence to send to the transmitter
      spdif_freq = 0;
    else if(freq == 48000)
      spdif_freq = 1;
    else if(freq == 32000)
      spdif_freq = 2;
    else
      spdif_freq = 3;

    spdif_ahb[2] = (spdif_freq << 6) + (GSTAT << 3) + (PREEM << 2)
                 + (COPY << 1) + AUDIO;  // TxChStat
    ratio = ((25000000)/(128 * freq)) - 1;        // compute the ratio variable
    printf("\nfreq : %08X, ratio : %08X, data : %08X\n", freq, ratio, data);
    printf("\nMusic is starting\n");
    spdif_ahb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
                 + (TINTEN << 2) + (1 << 1) + 1; // TxConfig

    perc = 0;
    old_perc = 0;
    done = 64;                    // 128 bytes are already in the buffer
    bufferId = 0;
    while((done * 4) < size)      // done is in word and size in byte
    { // check if lower/higher buffer is empty
      if((spdif_ahb[4] & mask[bufferId]) == mask[bufferId])
      {
        spdif_ahb[4] = mask[bufferId];        // clear the flag
        for(i = 0; i < 32; i++, done++)       // make the sample buffer full
        {
          spdif_ahb[i * 2 + 128 + bufferId * 64] = (data[done + 11] & 0xFFFF0000) >> 16;
          spdif_ahb[i * 2 + 129 + bufferId * 64] = (data[done + 11] & 0x0000FFFF);
        }
        bufferId = bufferId ^ 1;          // change the selected buffer

        perc = (int) (100 * (done * 4)/size); // compute the actual percentage
      }
```

```
      if(perc != old_perc)                    // check if percentage has changed
      {
        printf(" %i%%", perc);
        uart[0] = 0x0D;
        old_perc = perc;
      }
    }
    printf("\nMusic Ended\n");
    spdif_ahb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
             + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
  }

  else if(choice == 51)
  {

    slow_ahb[0] = 0x1;            // select the apb transmitter in output
    /*************************************************************************
    *                 MAKE THE BUFFER FULL BEFORE START                      *
    *************************************************************************/

    for(i = 0; i < 64; i++)   // make the sample buffer full
    {
      spdif_apb[i * 2 + 128] = (data[i + 11] & 0xFFFF0000) >> 16;
      spdif_apb[i * 2 + 129] = (data[i + 11] & 0x0000FFFF);
    }


    /*************************************************************************
    *                         START PLAY MUSIC                               *
    *************************************************************************/

    size = data[10];          // load the size
    freq = data[6];           // load the frequence
    s[0] = freq & 0x000000FF;         // change from big endian to little endian
    s[1] = (freq & 0x0000FF00) >> 8;
    s[2] = (freq & 0x00FF0000) >> 16;
    s[3] = (freq & 0xFF000000) >> 24;

    freq = s[3] + (s[2] << 8) + (s[1] << 16) + (s[0] << 24);

    if(freq == 44100)         // check the wich frequence to send to the transmitter
      spdif_freq = 0;
    else if(freq == 48000)
      spdif_freq = 1;
    else if(freq == 32000)
      spdif_freq = 2;
    else
      spdif_freq = 3;

    spdif_apb[2] = (spdif_freq << 6) + (GSTAT << 3) + (PREEM << 2)
                     + (COPY << 1) + AUDIO;  // TxChStat
    ratio = ((25000000)/(128 * freq)) - 1;        // compute the ratio variable
    printf("\nMusic is starting\n");
    spdif_apb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
                 + (TINTEN << 2) + (1 << 1) + 1; // TxConfig

    perc = 0;
    old_perc = 0;
    done = 64;                    // 128 byte are already in the buffer
    bufferId = 0;
    while((done * 4) < size)      // done is in word and size in byte
    {     // check if lower/higher buffer is empty
      if((spdif_apb[4] & mask[bufferId]) == mask[bufferId])
      {
        spdif_apb[4] = mask[bufferId];        // clear the flag
        for(i = 0; i < 32; i++, done++)       // make the sample buffer full
        {
          spdif_apb[i * 2 + 128 + bufferId * 64] = (data[done + 11] & 0xFFFF0000) >> 16;
          spdif_apb[i * 2 + 129 + bufferId * 64] = (data[done + 11] & 0x0000FFFF);
        }
        bufferId = bufferId ^ 1;          // change the selected buffer

        perc = (int) (100 * (done * 4)/size); // compute the actual percentage
```

```
        }

        if(perc != old_perc)                    // check if percentage has changed
        {
          printf(" %i%%", perc);
          uart[0] = 0x0D;
          old_perc = perc;
        }
      }
      printf("\nMusic Ended\n");
      spdif_apb[1] = (MODE << 20) + (ratio << 8) + (UDATEN << 6) + (CHSTEN << 4)
                    + (TINTEN << 2) + (0 << 1) + 0; // TxConfig
    }

  }
  return 0;

}
```

ANNEXE N

Code C de simulation

```
/*  this programm is usefull for the simulation of the AMBA system  */
/*  with AHB and APB / Wishbon bridges. Change it to test what you  */
/*  need                                                            */
/*  Valentini Samuel 23.11.07                                       */

#include <stdio.h>

#define UARTAREA (0x80000100)
#define SPDIFAHBAREA (0xA0000000)
#define SPDIFAPBAREA (0x80020000)
#define MEMCTRLAREA (0x80010000)
#define SDRAMAREA (0x60000000)
#define SPDIFAPBAREA (0x80020000)
#define WB_FAST_AREA (0xA0010000)
#define RAM_AREA (0x40000000)
#define WB_FAST_APB_AREA (0x80021000)
#define WB_SLOW_AREA (0xA0020000)

int main(int argc, char *argv[])
{

  /*********************************************************************
  *                           INITIALISATION                           *
  *********************************************************************/

  int i = 0;        // iteration variable
  int j;            // iteration variable

  volatile unsigned int *spdif_ahb = (volatile unsigned int *)SPDIFAHBAREA;
  volatile unsigned int *spdif_apb = (volatile unsigned int *)SPDIFAPBAREA;
  volatile unsigned int *memctrl = (volatile unsigned int *)MEMCTRLAREA;
  volatile unsigned int *sdram = (volatile unsigned int *)SDRAMAREA;
  volatile unsigned int *uart = (volatile unsigned int *)UARTAREA;
  volatile unsigned int *fast = (volatile unsigned int *)WB_FAST_AREA;
  volatile unsigned int *slow = (volatile unsigned int *)WB_SLOW_AREA;
  volatile unsigned int *ram = (volatile unsigned int *)RAM_AREA;
  volatile unsigned int *fast_apb = (volatile unsigned int
*)WB_FAST_APB_AREA;

  // normal transfers
  spdif_ahb[128] = 0x1234;
  spdif_ahb[1] = 0x7;

  spdif_ahb[129] = spdif_ahb[1];

  fast[0] = 0x123456;
  fast[1] = fast[0] - 0x23456;

  slow[0] = 0x987654;
  slow[1] = slow[0] - 0x87654;

  // loops
  for(j = 0; j < 10; j++)
  {
    for(i = 0; i < 100; i++)
    {
      fast[0] = i;
    }
  }

  for(j = 0; j < 10; j++)
```

```
  {
    for(i = 0; i < 100; i++)
    {
      ram[i] = i;
    }
  }
  // burst transfers
  *((volatile long long *) 0xa0010000) = 0x0123456789abcdefLL;
  *((volatile long long *) 0xa0000200) = 0x0123456789abcdefLL;
  *((volatile long long *) 0xa0020000) = 0x0123456789abcdefLL;
  return 0;

}
```

ANNEXE O

Tutorial : Compiler et implémenter un design préfabriqué des librairies Gaisler

```
PROGRAM THE FPGA
In Windows:
  - make a copy of the <grlib-gpl-x.x.xx-xxxxx> to <D:\Grlib-test>
  - make a copy of <D:\Grlib-test\designs\XXXXXXX> to <D:\Grlib-test\designs\HEVs>

In Cygwin
  - update <~/.bashrc> => start session with <source .bashrc>
  - run the command:
      cp /cygdrive/c/program\ files/gnuarm/bin/wish84.exe /cygdrive/c/program\
files/gnuarm/bin/wish
  - in </cygdrive/d/Grlib-test/designs/HEVs> run <make xconfig>
      this modifies <config.vhd>
  - run the command:
      make vsim
  - in the simulator, run the commands:
      run
      run -all
      quit
  - run the commands:
      make ise-synp                 Synthesis with Synplify
      make ise-prog-fpga            program the FPGA

RUN A PROGRAM
In Windows:
    - Download the evaluation version of grmon
    - Unzip the files in <D:\xxxxx>

In Cygwin:
    - run the commands:
        sparc-elf-gcc -msoft-float -g -O2 xxxxx.c -o xxxxx.exe
        grmon-eval -dsu
    - in the debugger, run the commands:
        load xxxxx.exe
        verify xxxxx.exe
        run



Here is the .bashrc file :

alias ed='nano -w'
alias ll='ls -la'

PATH="/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:"
PATH="${PATH}/cygdrive/c/WINDOWS:/cygdrive/c/WINDOWS/system32:"
PATH="${PATH}/cygdrive/c/WINDOWS/System32/Wbem:/cygdrive/c/WINDOWS/system32/nls:"
PATH="${PATH}/cygdrive/c/WINDOWS/system32/nls/ENGLISH:/cygdrive/c/Program
Files/Novell/ZENworks/:"
PATH="${PATH}/cygdrive/c/Program Files/Microsoft Visual Studio/Common/Tools/WinNT:"
PATH="${PATH}/cygdrive/c/Program Files/Java/jdk1.5.0_04/bin:"
PATH="${PATH}/cygdrive/c/program files/gnuarm/bin:"
PATH="${PATH}/cygdrive/c/eda/MENTOR~1/LICENS~1:"
PATH="${PATH}/cygdrive/c/eda/xilinx_webpack8/bin/nt:"
PATH="${PATH}/cygdrive/c/Modeltech_6.2f/win32"
PATH="${PATH}/opt/sparc-elf-3.4.4/bin"
export PATH=/cygdrive/c/Program\ Files/Synplicity/fpga_81/bin:$PATH

cd /cygdrive/x/xxxxx
```

ANNEXE P

Tutorial : Utilisation de la SDRAM avec le contrôleur de mémoire des librairies Gaisler

```
 - mettre à jour les génerics du memctrl (sden = 1, sdbits, sdrasel, fast,
invclk)
 - ajouter un bloc clkgen afin d'avoir un sdclk.
   mettre sur l'entrée clkin le signal clk recu de l'oscillateur et sur la
sortie clk sera le clock du circuit.
   Sur l'entrée cgi :
    cgi.pllctrl <= "00";
    cgi.pllrst <= reset;
    cgi.pllref <= clkin;
    pciclkin <= '0';
 - Router les signaux sur la SDRAM
 - Au démarrage du programme, actualiser la valeur du registre mcfg2 du
memory controller.
 - La SDRAM est accesible via la moitié supérieur de la RAM AREA

 - Update the generics of the memctrl (sden = 1, sdbits, sdrasel, fast,
invclk)
 - add a clkgen bloc to have a sdclk
   put on the input clkin the clk from the oscillator and the output clk
will be the clock of the circuit
   On the cgi input :
    cgi.pllctrl <= "00";
    cgi.pllrst <= reset;
    cgi.pllref <= clkin;
    pciclkin <= '0';
 - Route the signals on the SDRAM (sdcke, sdcsn, sdwen, sdrasn, sdcasn,
sddqm, address, data)
 - At the start of the programm, update the value of the mcfg2 register of
the memory controller.
 - The SDRAM is mapped into upper half of the RAM area.
 - For more informations, see the mctrl in the grip document.
```