

Rapport de projet

Interfaçage d'un processeur RISC-V picorv32 avec le système
embarqué de la carte FPGA Zybo Z7-20

Projet RV32-ZYNQ

19 février 2024

MORAL Alexandre - ASSIER Axel

Table des matières

1	Introduction	3
2	Matériel et ressources utilisées	4
2.1	Carte Zybo Z7-20	4
2.2	PicoRV32	5
2.3	Vivado	5
3	Réalisation	7
3.1	Création du projet et de l'IP picorv32_axi	7
3.2	Création du block design	8
3.3	Adressage	10
3.4	Fonctionnement de l'IP AXI GPIO	11
3.5	Validation du design Vivado	12
4	Chaîne de cross-compilation	17
4.1	Installation	17
4.2	Linker Script et crt0	17
5	Tests et Validations	21
5.1	Librairie GPIO	21
5.2	Tests	21
6	Gestion de projet	27
6.1	Comparaison du temps estimé au temps réel	27
6.2	Journal de bord	27
6.3	Problèmes rencontrés	29
7	Conclusion	31
8	Annexe	32
8.1	Bibliographie	32
8.2	Résultats d'implémentation	32

1 Introduction

L'architecture RISC-V, avec son jeu d'instruction open-source, s'est imposée comme une solution de choix pour le développement de systèmes embarqués. Sa flexibilité et sa faible consommation d'énergie sont un atout majeur quant à son utilisation. Cependant, il n'existait pas, jusqu'à très récemment, de FPGA (*Field Programmable Gate Array*) embarquant un ou plusieurs coeurs RISC-V sous la forme *d'IP Hard*.

L'objectif de ce projet vise à explorer le potentiel d'implémentation d'un processeur RISC-V sur une carte FPGA Zybo Z7-20. Cette carte, fabriquée par Digilent Inc., embarque un processeur ARM *dual-core* Cortex-A9 et dispose d'une mémoire DDR3 de 1Go.

L'implémentation du processeur RISC-V devra relever plusieurs défis, comme l'accès à la mémoire DDR3 embarquée, le contrôle des entrées et sorties GPIO et devra pouvoir coexister avec le processeur ARM embarqué de la carte.

Ce projet s'inscrit dans une démarche de recherche au cours du Master parcours Systèmes embarqués et connectés : infrastructures et logiciels (SECIL) de l'université Paul Sabatier III, à Toulouse.

Le succès de ce projet permettra de démontrer la faisabilité d'implémentation et d'utilisation de coeur RISC-V sur une carte FPGA possédant déjà un processeur embarqué afin d'ouvrir la voie à une possible implémentation plus complète d'un autre coeur pour plus de fonctionnalités.

2 Matériel et ressources utilisées

2.1 Carte Zybo Z7-20

La carte Zybo Z7-20 est une carte de développement FPGA conçue par Digilent Inc. .

Elle combine un processeur ARM Cortex-A9 double cœur avec un FPGA Xilinx 7 Series.

Les caractéristiques de la carte sont les suivantes :

- Une prise en charge complète de Vivado
- Un Processeur double cœur Cortex-A9 cadencé à 667 MHz
- 1 Go de mémoire vive DDR3L rapide avec bus 32 bits à 533 MHz
- Des ports USB, HDMI, Ethernet
- Des connecteurs PMOD et PCAM
- Une capacité de programmation par JTAG, Quad-SPI flash et carte microSD

Au niveau FPGA, elle possède les capacités suivantes :

- 13 300 *logic slices*(tranches logiques)
- 53 200 LUTs 6 entrées (*LookUp Table*)
- 106 400 bascules D
- 630 Ko de BRAM

Comme décrit dans la figure 1, il est possible de communiquer depuis la partie FPGA de la carte avec le *Processing System* (PS) via des bus *AMBA Interconnect*.

La norme *AMBA (Advanced Microcontroller Bus Architecture)* définit notamment les bus AXI (*Advanced eXtensible Interface*).

Les bus AXI sont des bus de communication maîtres-esclaves basés sur l'adressage mémoire.

Pour faire communiquer un processeur implémenté sur la carte FPGA et la mémoire du PS nous utiliserons donc le bus AXI Haute Performance (voir figure 1).

2.2 PicoRV32

Sur la partie FPGA de la carte, nous avons implémenté un processeur RISC-V, le PicoRV32.

Le PicoRV32 est un coeur RISC-V *open-source* sous licence ISC.

Ce processeur est un RV32IMC, un RISC-V 32 bits ayant l'ISA (*Instruction Set Architecture*) RISC-V de base avec le support des instructions de multiplication et division (M) et des instructions compressées (C).

Néanmoins, nous avons choisi de n'implémenter que la version RV32I (avec l'ISA RISC-V basique), afin de ne pas utiliser trop de ressources de la carte.

Ce processeur a été choisi car il est de petite taille et compact, facile à utiliser avec des performances raisonnables.

Sa taille est un atout pour notre projet, car notre carte FPGA a des ressources limitées.

De plus, le PicoRV32 fournit une version du CPU avec une interface AXI4-Lite, ce qui est nécessaire pour pouvoir communiquer avec la mémoire du *processing system*.

Ainsi, la version régulière du PicoRV32 n'utilise que les ressources suivantes :

- 917 *logic slices*
- 48 LUTs utilisées en tant que mémoire
- 583 *slice registers*

2.3 Vivado

Afin de pouvoir configurer la carte FPGA, nous avons utilisé l'outil Xilinx Vivado.

Xilinx Vivado est un environnement de développement logiciel créé par Xilinx pour simplifier la conception et la mise en œuvre de systèmes sur puce (SoC) basés sur FPGA.

Dans ce projet, nous avons utilisé la version Vivado 2023.2 avec le support des **7 Series**.

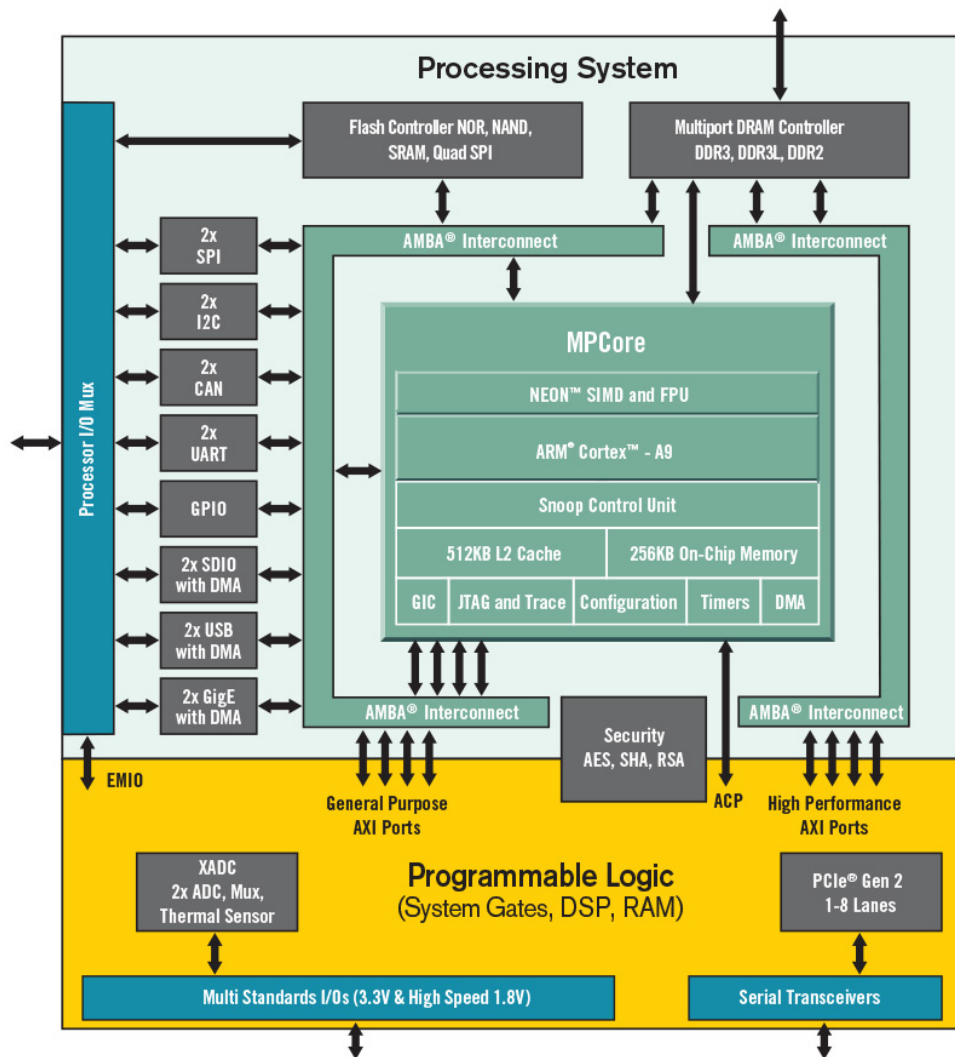


FIGURE 1 – Vue de la carte Zybo Z7

3 Réalisation

3.1 Création du projet et de l'IP `picorv32_axi`

Nous avons créé un nouveau projet Vivado ciblant le langage Verilog et inclus à ce projet le fichier `picorv32.v`, disponible sur le github du projet PicoRV32.

Nous avons encapsulé sous paquet IP le `picorv32_axi` (voir figure 2), la version du PicoRV32 avec l'interface AXI.

Nous avons fait cela afin de pouvoir utiliser cette IP dans un *block design*, pour pouvoir interfacer le système embarqué de la carte et notre CPU.

Ainsi, l'IP créée dispose de plusieurs ports et registres de configuration.

Elle dispose notamment :

- D'un port `clk` pour l'horloge
- D'un port `resetrn` pour la ré-initialisation
- De ports PCPI (*Pico Co-Processor Interface*)
- D'un port `IRQ` (*Interruption Request*) et `EOI` (*End of Interruption*) pour gérer les interruptions
- De ports pour gérer les traces

Les ports PCPI permettent d'implémenter des instructions dans des coeurs externes, mais nous avons pu désactiver cette fonctionnalité dans la configuration de l'IP (`ENABLE_PCPI = 0`).

Nous avons également désactivé l'IRQ (`ENABLE_IRQ = 0`) dans ce projet, et changé la configuration de `PROGADDR_RESET` et `STACKADDR` comme nous allons l'expliquer par la suite.

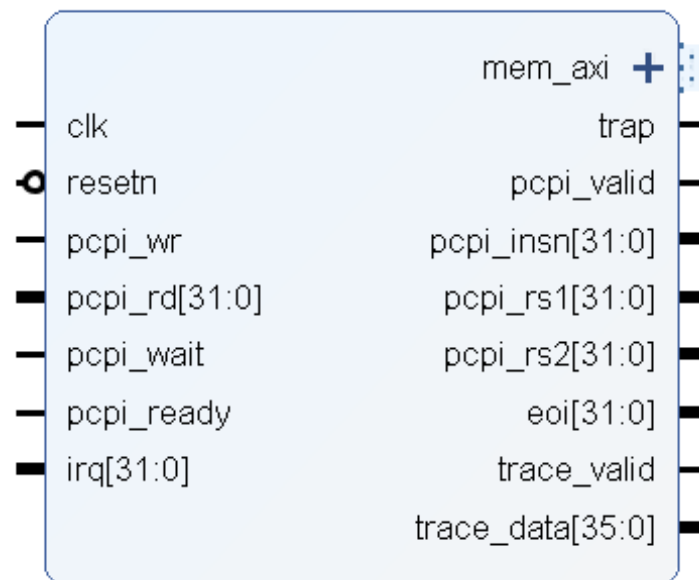


FIGURE 2 – Vue de l'IP du `picorv32_axi`

3.2 Création du block design

Afin de connecter notre IP *picorv32_axi* créée dans la section 3.1, nous avons créé un bloc design (voir figure 5).

Dans ce bloc design, on va connecter plusieurs maîtres (M) et esclaves (S) par un bus AXI.

Ce bloc design est constitué de plusieurs éléments décrits dans la table 1.

Nom	Nom de l'IP	Rôle	Fonction
picorv32_axi_0	picorv32_axi_v1_0	M	Notre processeur RV32I
axi_mem_intercon	AXI Interconnect	S	Permet de connecter plusieurs maîtres et esclaves AXI entre eux
processing_system7_0	ZYNQ7 Processing System	S	Les ressources embarquées de la carte
axi_bram_ctrl_0	AXI BRAM Controller	S	Interfacer les blocs mémoire (BRAM) du FPGA avec l'AXI
blk_mem_gen_0	Block Memory Generator	/	Permet de créer et de personnaliser des blocs de mémoire embarquée (BRAM), pour y mettre notamment du code
axi_gpio_sws_leds	AXI GPIO	S	Contrôles des LEDS et des interrupteurs
axi_gpio_btns	AXI GPIO	S	Contrôles des boutons
ila_pico	Integrated Logic Analyzer	/	Outil de debug qui permet de visualiser les signaux internes une fois le bitstream téléversé sur la carte

TABLE 1 – Éléments et rôles des composants du *block design*

Le picorv32 est donc le seul maître dans notre bus, et permet de commander tous les esclaves. L'*Integrated Logic Analyzer* (ILA) est connecté en sortie du picorv32 pour que l'on puisse observer toutes les requêtes du bus.

Le PS est également relié à deux sorties de la carte, DDR (la mémoire) et FIXED_IO (des entrées et sorties de la carte).

Les AXI GPIO sont quant à eux reliés à des sorties définies dans le *HDL Wrapper* du bloc design. Un *HDL Wrapper* est un fichier verilog généré par Vivado ou l'utilisateur qui enveloppe un bloc design. Il interface le bloc design avec le reste du système FPGA, en définissant ses ports et signaux d'entrée/sorties.

Nous avons donc généré automatiquement avec Vivado *HDL Wrapper* du design, et déclaré en Verilog les entrées et sorties GPIO de la carte que nous souhaitons interfacer (voir figure 3).

```

1 | input [3:0] btns_4bits_tri_i;
2 | output [3:0] leds_4bits_tri_o;
3 | input [3:0] sws_4bits_tri_i;
```

FIGURE 3 – Déclarations des entrées sortie GPIO dans le *HDL Wrapper* du bloc design.

Pour attribuer ces nouvelles entrées et sorties déclarées dans l'enveloppe HDL, nous avons dû utiliser un fichier de contrainte (.xdc) et définir les propriétés des éléments que nous souhaitons

utiliser (voir figure 4).

```

1  ##Switches
2  set_property -dict { PACKAGE_PIN G15      IOSTANDARD LVCMOS33 } [get_ports
   { sws_4bits_tri_i[0] }];
3  set_property -dict { PACKAGE_PIN P15      IOSTANDARD LVCMOS33 } [get_ports
   { sws_4bits_tri_i[1] }];
4  set_property -dict { PACKAGE_PIN W13      IOSTANDARD LVCMOS33 } [get_ports
   { sws_4bits_tri_i[2] }];
5  set_property -dict { PACKAGE_PIN T16      IOSTANDARD LVCMOS33 } [get_ports
   { sws_4bits_tri_i[3] }];
6
7
8  ##Buttons
9  set_property -dict { PACKAGE_PIN K18      IOSTANDARD LVCMOS33 } [get_ports
   { btns_4bits_tri_i[0] }];
10 set_property -dict { PACKAGE_PIN P16      IOSTANDARD LVCMOS33 } [get_ports
   { btns_4bits_tri_i[1] }];
11 set_property -dict { PACKAGE_PIN K19      IOSTANDARD LVCMOS33 } [get_ports
   { btns_4bits_tri_i[2] }];
12 set_property -dict { PACKAGE_PIN Y16      IOSTANDARD LVCMOS33 } [get_ports
   { btns_4bits_tri_i[3] }];
13
14
15 ##LEDs
16 set_property -dict { PACKAGE_PIN M14      IOSTANDARD LVCMOS33 } [get_ports
   { leds_4bits_tri_o[0] }];
17 set_property -dict { PACKAGE_PIN M15      IOSTANDARD LVCMOS33 } [get_ports
   { leds_4bits_tri_o[1] }];
18 set_property -dict { PACKAGE_PIN G14      IOSTANDARD LVCMOS33 } [get_ports
   { leds_4bits_tri_o[2] }];
19 set_property -dict { PACKAGE_PIN D18      IOSTANDARD LVCMOS33 } [get_ports
   { leds_4bits_tri_o[3] }];

```

FIGURE 4 – Déclarations des entrées sortie GPIO dans le fichier de contraintes (.xdc).

3.3 Adressage

Le bus **AXI** étant un bus basé sur l'adressage mémoire, nous avons dû définir des adresses pour les esclaves du bus.

Pour faire cela, Vivado possède un *Address Editor* qui nous permet de définir les adresses.

Par cas d'usage, on met en général la **ROM** à l'adresse **0x0**. Or, dans ce projet ce n'est pas possible, car la **RAM** du PS à une adresse fixe dans la carte ; on doit lui assigner une adresse dont l'*offset* des trente-deux bits de poids faibles est égal à **0x0000_0000**. Or, comme nous avons un processeur trente-deux bits, on est forcé de faire commencer la **RAM** à l'adresse **0x0**.

Nous avons donc décidé d'utiliser les adresses décrites en table 2.

Nom	Adresse de Base	Portée	Adresse haute
/axi_bram_ctrl_0/S_AXI	0xC000_0000	64K	0xC000_FFFF
/axi_gpio_btns/S_AXI	0x4000_0000	64K	0x4000_FFFF
/axi_gpio_sws_leds/S_AXI	0x4001_0000	64K	0x4001_FFFF
/processing_system7_0/S_AXI_HP0	0x0	1G	0x3FFF_FFFF

TABLE 2 – Adresses des esclaves du *block design*.

Il est également intéressant de noter que l'*Address Editor* doit être utilisé si l'on veut augmenter la taille de la **ROM**.

Les blocs IP *Block Memory Generator* et *AXI BRAM Controller* s'adaptent et se modifient en fonction de la portée définie dans cet outil.

Comme nous avons modifié l'adresse de la **ROM**, nous devons faire en sorte que le processeur implémenté commence son programme à cette nouvelle adresse, **0xC000_0000**.

Comme évoqué plus tôt dans la section 3.1, nous allons modifier le paramètre *PROGADDR_RESET* de l'IP du *picorv32_axi*, qui correspond à l'adresse de commencement du programme.

On va donc mettre la valeur **0xC000_0000** pour *PROGADDR_RESET* car le programme sera chargé dans la **ROM**.

Le *picorv32* lira en premier lieu l'instruction située à l'adresse **0xC000_0000**. Il enverra une requête sur le bus **AXI** pour lire la mémoire. Cette requête passera donc par le composant *AXI Interconnect*. Ce composant route les requêtes **AXI** vers l'esclave correspondant.

Si jamais l'adresse demandée n'est pas reconnue par l'*AXI Interconnect*, ce dernier renverra une valeur d'erreur, **0xDECODE1C**, pratique pour trouver des problèmes d'adressage lors de la phase de vérification.

3.4 Fonctionnement de l'IP AXI GPIO

Pour interfacer le processeur RISC-V implémenté sur le FPGA et les différentes entrées sorties GPIO de la carte, on utilise une IP AXI GPIO .

Ce composant possède six registres pour configurer les entrées/sorties, lire les données et gérer les interruptions (voir table 3).

Address Space Offset	Nom	Type d'accès	Valeur par Défaut	Description
0x0000	GPIO_DATA	R/W	0x0	Registre de donnée du Channel 1 AXI GPIO.
0x0004	GPIO_TRI	R/W	0x0	Registre de contrôle 3-état du Channel 1 AXI GPIO.
0x0008	GPIO2_DATA	R/W	0x0	Registre de donnée du Channel 2 AXI GPIO.
0x000C	GPIO2_TRI	R/W	0x0	Registre de contrôle 3-état du Channel 2 AXI GPIO.
0x011C	GIER	R/W	0x0	<i>Global Interrupt Enable Register</i>
0x0128	IP_IER	R/W	0x0	<i>IP Interrupt Enable Register</i>
0x0120	IP_ISR	R/W	0x0	<i>IP Interrupt Status Register</i>

TABLE 3 – Registres du bloc IP AXI GPIO.

Dans notre projet, nous avons utilisé et testé que les registres *GPIOx_DATA* et *GPIOx_TRI*.

Le registre de données AXI GPIO DATA est utilisé pour lire les ports d'entrée et écrire sur les ports de sortie. Lorsqu'un port est configuré comme entrée, l'écriture dans le registre de données AXI GPIO n'a aucun effet.

Pour chaque bit d'E/S programmé comme entrée :

- R : Lit la valeur sur la broche d'entrée.
- W : Aucun effet.

Pour chaque bit d'E/S programmé comme sortie :

- R : La lecture de ces bits renvoie toujours zéro
- W : Écrit la valeur dans le bit correspondant du registre de données AXI GPIO et la broche de sortie.

Le registre de contrôle 3 états AXI GPIO TRI est utilisé pour configurer les ports comme entrée ou sortie.

Chaque broche d'E/S du GPIO AXI est programmable individuellement comme entrée ou sortie. Pour chaque bit :

- 0 = broche d'E/S configurée comme sortie.
- 1 = broche d'E/S configurée comme entrée.

3.5 Validation du design Vivado

Pour valider le design Vivado, nous avons chargé un programme dans la ROM. On utilise le *Block Memory Generator* afin de pouvoir charger un fichier d'extension *.coe*.

Nous avons écrit un programme en assembleur RISC-V qui permet d'interagir avec les interrupteurs, les LEDs et la RAM (voir figure 7).

Ce programme assembleur est ensuite converti en fichier d'extension *.coe*. Cette extension est reconnue par Vivado et nous permet de configurer la BRAM générée par le *Block Memory Generator*.

La première ligne d'un fichier *.coe* renseigne le *radix*, la base du vecteur d'initialisation mémoire (2 = binaire, 16 = hexadecimal).

La seconde ligne du fichier *.coe* correspond au vecteur d'initialisation mémoire, au code du programme.

Le code de la figure 7 traduit en *.coe* est donc disponible en figure 8. Dans ce fichier *.coe*, on a décidé d'utiliser la base binaire (*radix* = 2), on a donc traduit chaque instruction assembleur vers du binaire pour créer notre vecteur d'initialisation mémoire.

Une fois le fichier *.coe* utilisé pour configurer la BRAM on peut valider l'exécution du programme grâce à l'ILA. On obtient donc des chronogrammes qui décrivent l'état du bus AXI (voir figure 6).

Pour simplifier le protocole AXI, on s'intéresse principalement à quatre champs :

- ARADDR : Adresse de lecture
- AWADDR : Adresse d'écriture
- RDATA : Données lues
- WDATA : Données à écrire

Dans notre chronogramme (voir figure 6) on peut observer sur le champ ARADDR des requêtes du processeur pour lire les instructions (0xc000_002c), les données du registre GPIO (0x4001_0000) et la valeur stockée en mémoire (0x0).

Sur le champ AWADDR on constate que l'on écrit WDATA qui vaut 0x4 à l'adresse 0x0 aux alentours de l'échantillon 330.

Aux alentours de l'échantillon 440, on va aller lire ce qu'on avait écrit à cette adresse.

Grâce à l'AXI ILA, nous avons pu vérifier l'exécution de nos programmes assembleurs directement sur la carte.

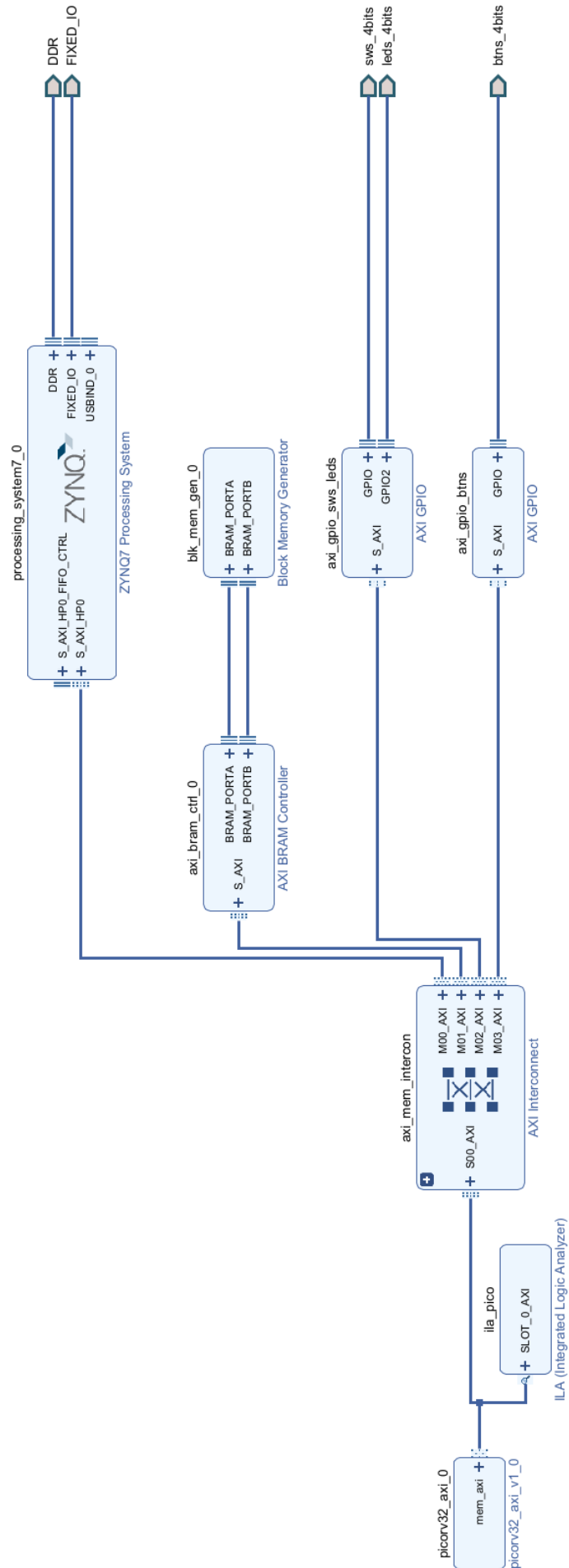


FIGURE 5 – Block design du projet

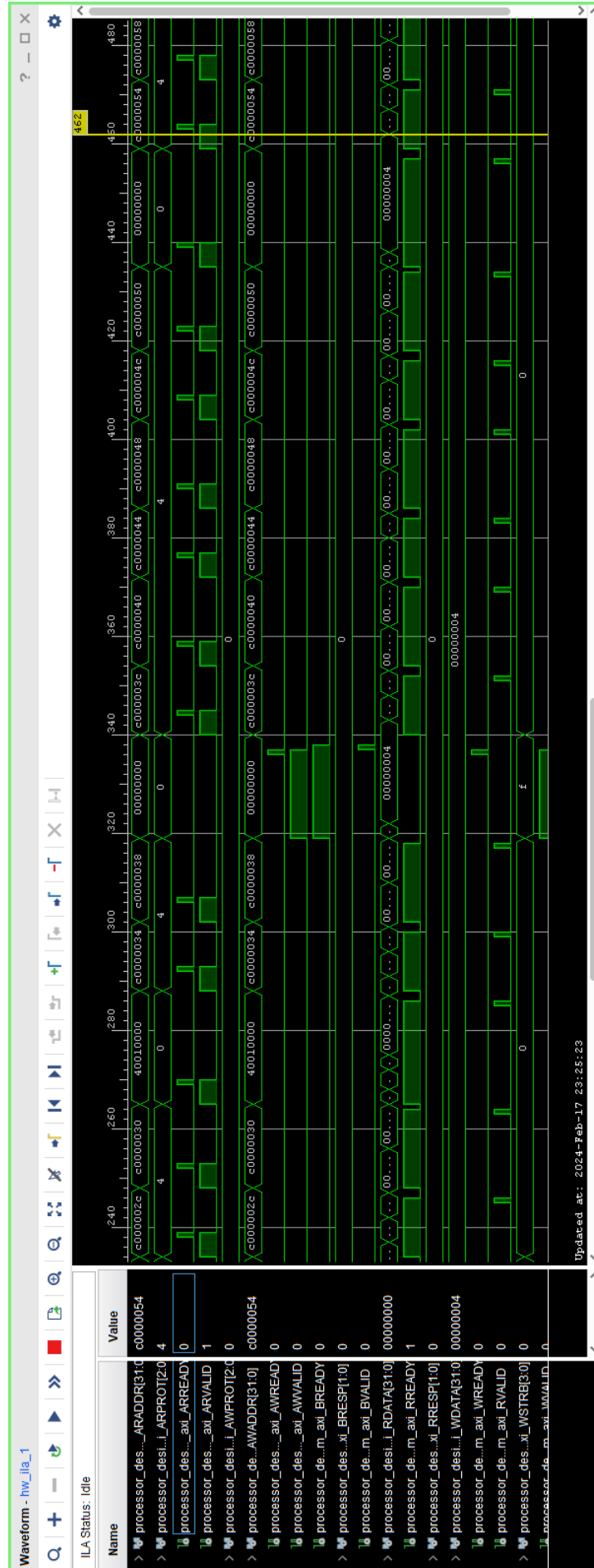


FIGURE 6 – Chronogramme décrivant l'état du bus AXI.

```

lui t0, 0x40010 ;Address of the GPIO Block
lui t1, 0xC0000 ;Address of bootrom

addi t2, x0, 0xff0
lui t3, 0xffff
or t3, t2, t3 ;Mask 0xffff_fff0

;Switches as input, leds as output
lw t2, 0x4(t0);switches GPIOTRI
ori t2,t2,15
sw t2, 0x4(t0)

lw t2, 0xC(t0);leds GPIO2TRI
and t2, t2, t3
sw t2, 0xC(t0)

;Init finished, now read switches data
lw t2, 0x0(t0)
andi t2,t2,15 ;Mask to keep the 4 lb
sw t2, 0x0(x0);Save in memory

;Flush the pipeline
add x0,x0,x0
add x0,x0,x0
add x0,x0,x0
add x0,x0,x0
add x0,x0,x0

;load the switches value
lw t2, 0x0(0x0)
andi t2,t2,15 ;Mask to keep the 4 lb

;Set the leds to switches values
sw t2,0x8(t0)

;Loop the program after the init
jalr x0, 0x2c(t1)

```

FIGURE 7 – Programme en assembleur testant la mémoire et le GPIO.

```
memory_initialization_radix=2;
memory_initialization_vector= 01000000000000010000001010110111
11000000000000000000000001100110111 111111110000000000000001110010011
1111111111111111111111111000110111
00000001110000111110111000110011 000000000100001010100011100000011
00000000111100111110001110010011
00000000011100101010001000100011 000000001100001010100011100000011
00000001110000111111001110110011
00000000011100101010011000100011 0000000000000001010100011100000011
00000000111100111111001110010011
00000000011100000010000000100011 0000000000000000000000000000000110011
0000000000000000000000000000000110011
0000000000000000000000000000000110011 0000000000000000000000000000000110011
0000000000000000000000000000000110011
000000000000000000000000000000010001110000011 00000000111100111111001110010011
00000000011100101010010000100011
000000101100001100000000001100111;
```

FIGURE 8 – Traduction de l’assembleur figure 7 en *.coe*.

4 Chaîne de cross-compilation

4.1 Installation

Pour pouvoir compiler du code C vers une cible RV32I, nous allons devoir installer la [chaîne de cross compilation RISC-V](#).

Les commandes pour l'installation sont décrites en figure 9.

```
sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex
texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
ninja-build git cmake libglib2.0-dev
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
sudo ./configure --with-arch=rv32i --prefix=/opt/riscv32i
sudo make -j$(nproc)
```

FIGURE 9 – Commandes d'installation de la *toolchain* RV32I.

4.2 Linker Script et crt0

Le code compilé en RV32I doit être placé dans la ROM. Dans notre design, nous avons mis l'adresse de base de la ROM en `0xc000_0000` et l'adresse de base de la RAM en `0x0`.

Nous avons donc besoin de faire débiter le code compilé en `0xc000_0000`.

Il faut donc utiliser un *linker script* qui nous permettra de placer les différentes sections de notre programme aux adresses voulues.

Le *linker script* en figure 10 informe le *linker* de l'adresse de début du code exécutable dans l'image mémoire finale via `ENTRY(_start)`.

La fonction `_start` est définie dans le fichier `crt0`. Ce fichier effectue les tâches et routines nécessaires avant de pouvoir appeler la fonction principale du programme, `main()`.

Nous avons donc défini nous même un `crt0` en figure 11 le plus simple possible pour faire fonctionner notre programme. Ce `crt0` va initialiser le registre `sp` (*stack pointer*) du processeur pour définir l'adresse haute de la *stack* puis appeler `main()`. Nous aurions pu également modifier le paramètre `STACKADDR` de notre IP `picorv32_axi`.

Dans le *linker script* sont ensuite définies les mémoires de notre processeur, leurs tailles et leurs permissions relatives.

Dans le cas de la ROM, on peut observer `(rxai!w)`. Cela indique que la ROM peut contenir de tout excepté des sections écrivables.

La RAM (`wxa!ri`) peut contenir de tout, mis à part des sections pré-initialisées et en lecture seule.

Ensuite, le *linker script* définit les sections dans la mémoire.

La section `.text` est la section qui contient le code exécutable. Dans cette section, on indique que l'on met en premier lieu `crt0.o` pour qu'il soit situé au début de la section. Par la suite,

`. = ALIGN(8)` permet d'aligner la mémoire afin d'optimiser les performances de notre code (en alignant la mémoire, on s'assure que les instructions *lw* et *rw* récupèrent une instruction alignée). On inclut dans la section `.text` toutes les sections `.text` et `.text*` de tous les fichiers objets liés.

Enfin, on indique que la section `.text` est assigné à la mémoire ROM grâce à `> ROM`.

On place les données initialisées `.data` dans la RAM juste après la ROM (`> RAM AT> ROM`).

Les données constantes et les chaînes en lecture seule `.rodata` sont placées dans la ROM.

Les données non initialisées `.bss` sont placées dans la RAM.

On définit ensuite une section `.stack` dans la RAM.

Au début de cette section, on place le symbole de fin de pile (`_estack = .;`).

On avance de 1 MB (`. = . + STACK_SIZE;`) et l'on place le symbole de début de pile à la fin (`_sstack = .;`).

On peut voir en figure [12](#) un fichier Makefile permettant de compiler un programme simple en utilisant notre chaîne de compilation croisée.

On utilise `bin2coe` pour pouvoir générer des fichiers `.coe` automatiquement.

```

1  /* First instruction to execute in the program */
2  ENTRY(_start)
3
4  /* Memories */
5  MEMORY {
6      /* ROM where I put the code */
7      ROM (rxai!w) : ORIGIN = 0xC0000000, LENGTH = 0x10000
8      /* RAM where I want to work, to save unit variables, to have my stack... */
9      RAM (wxa!ri) : ORIGIN = 0x00000000, LENGTH = 0x40000000
10 }
11
12 /* Code and data sections */
13 SECTIONS
14 {
15     /* Code */
16     .text : {
17         crt0.o
18         . = ALIGN(8);
19         *(.text)
20         *(.text*)
21         . = ALIGN(8);
22     } > ROM
23
24     /* Initialized data */
25     .data : {
26         . = ALIGN(8);
27         PROVIDE( __global_pointer$ = . );
28         *(.data)
29     } > RAM AT> ROM
30
31     .rodata :
32     {
33         . = ALIGN(8);
34         *(.rodata)           /* .rodata sections (constants, strings, etc.) */
35         *(.rodata*)          /* .rodata* sections (constants, strings, etc.) */
36         . = ALIGN(8);
37     } > ROM
38
39     /* Uninitialized data */
40     .bss : {
41         PROVIDE( __bss_start = . );
42         *(.bss)
43         *(.bss*)
44
45         PROVIDE( __bss_end = . );
46         PROVIDE( _end = . );
47     } > RAM
48
49     /* Stack */
50     .stack (NOLOAD) :
51     {
52         . = ALIGN(8);
53         _estack = .;
54         . = . + STACK_SIZE;
55         . = ALIGN(8);
56         _sstack = .;
57     } > RAM
58 }
59
60 STACK_SIZE = 0x100000; /* 1 MB */

```

FIGURE 10 – *Linker script* utilisé dans la chaîne de compilation croisée.

```
1 //No need since crt0.o is first thing of my .text section
2 //void _start(void) __attribute__((section(".text")));
3
4
5 void _start() {
6     asm ("lui sp,0x100");
7     main();
8     while(1);
9 }
```

FIGURE 11 – Fichier crt0 utilisé dans la chaîne de compilation croisée.

```
1 CC = riscv32-unknown-elf-gcc
2 OBJCOPY = riscv32-unknown-elf-objcopy
3 LFLAGS = -Wl,-Tlinker.ld -Xlinker -Map=main.map
4 CFLAGS = -march=rv32i -mabi=ilp32 -nostartfiles
5 SRC = gpio.c main.c
6 BIN2COE = bin2coe
7
8 MEMORY_WIDTH = 32
9
10
11 default : main.coe
12
13 crt0.o : crt0.c
14     $(CC) -c $^ $@
15
16 main.elf : crt0.o $(SRC)
17     $(CC) $(CFLAGS) $^ -o $@ $(LFLAGS)
18
19 main.bin : main.elf
20     $(OBJCOPY) -O binary $^ $@
21
22 main.coe : main.bin
23     $(BIN2COE) -i $^ -w $(MEMORY_WIDTH) -o $@
24
25 clean:
26     rm -f *.o
27     rm -f *.elf
28     rm -f *.bin
29     rm -f *.coe
```

FIGURE 12 – Fichier Makefile utilisant la chaîne de compilation croisée.

5 Tests et Validations

5.1 Librairie GPIO

Afin de pouvoir interagir avec les GPIO de la carte depuis le code C, nous avons défini des fonctions.

Dans le fichier `gpio.h` (voir figure 14) on définit les adresses des GPIO AXI et leurs *offsets*. On définira également des macros et les en-tête des fonctions qui permettent de lire un registre, d'écrire vers un registre et de configurer nos GPIO.

Dans le fichier `gpio.c` (voir figure 15), on écrit nos fonctions. Dans les fonctions `read_register` et `write_register` on écrit du code assembleur en C.

La première ligne définit l'instruction, et les `%x` les opérandes.

La seconde ligne commençant par `:` définit les opérateurs de sortie, c'est là où l'on enregistre le résultat de l'instruction, de l'assembleur vers le C.

La troisième ligne commençant par `:` définit les opérateurs d'entrée.

5.2 Tests

Afin de tester les fonctions décrites en section 5.1, nous avons réalisé des jeux de tests.

Dans la figure 13 on peut voir un code C permettant de lire l'état d'un interrupteur et d'allumer ou éteindre une LED correspondante.

En compilant ce fichier, on obtient un `.elf` qui nous permet d'observer comment est placé notre programme, à l'aide de la commande `riscv32-unknown-elf-readelf file.elf -a`.

Dans la figure 16, on peut voir que le code a bien été compilé pour du RISC-V, et l'*Entry point address* correspond bien à l'adresse de base de notre ROM, `0xc000_0000`.

Les *Section Headers* en figure 17 nous renseignent sur les adresses des sections définies dans le *linker script*. La table des symboles (voir figure 18), nous donne quant à elle des informations sur l'emplacement des fonctions du programme.

Une fois le fichier converti en binaire, puis en `.coe`, on peut directement le valider sur la carte, en utilisant le même processus décrit en section 3.5.

On peut également appeler gdb (voir figure 19) sur le fichier `.elf riscv32-unknown-elf-gdb file.elf` afin d'obtenir le programme traduit en RISC-V pour vérifier la bonne exécution du programme sur la carte.

Notre design a été validé via les jeux de tests suivants :

- Test mémoire en Assembleur, Boucle addition et enregistrement mémoire
- Test GPIO et mémoire en Assembleur, Boucle Lecture des switch, enregistrement mémoire, lecture mémoire, modification des LEDS
- Test mémoire C, Boucle addition et enregistrement mémoire
- Test mémoire et GPIO C, Boucle lecture des switch, modification des LEDS
- Test mémoire et GPIO C, Boucle lecture des boutons, modification des LEDS

```

1  #include "gpio.h"
2
3  int main()
4  {
5      config_switch(0);
6      config_led(0);
7
8      while(1) {
9          // Read the state of the switch
10         int switch_state = read_switch(0);
11
12         set_led(0, switch_state);
13
14     }
15
16     return 0;
17 }

```

FIGURE 13 – Fichier test_switch_led.c.

```

1  #include "stdint.h"
2
3  #define GPIO_DATA_OFFSET      0x0000
4  #define GPIO_TRI_OFFSET      0x0004
5  #define GPIO_2DATA_OFFSET    0x0008
6  #define GPIO_2TRI_OFFSET     0x000c
7  #define GIER_OFFSET          0x011c
8  #define IP_IER_OFFSET        0x0128
9  #define IP_ISR_OFFSET        0x0120
10
11 #define GPIO_BUTTONS          0x40000000
12 #define GPIO_SWS_LEDS         0x40010000
13
14 //Macro
15 #define MASK(l)                ((1 << (l))-1)
16 #define GET_BITS(x, i, l)      (((x)>>(i))&MASK(l))
17 #define REP_BITS(x, i, l, y)   (((x)&~(MASK(l)<<i))|((y)<<(i)))
18
19 //Output / input
20 #define OUTPUT 0
21 #define INPUT 1
22
23
24 uint32_t read_register(uint32_t address);
25 void write_register(uint32_t address, uint32_t data);
26 void config_led(int i);
27 void config_switch(int i);
28 void config_button(int i);
29 uint32_t read_switch(int i);
30 uint32_t read_button(int i);
31 void set_led(int i, int state);
32

```

FIGURE 14 – Fichier gpio.h.

```

1  #include "gpio.h"
2
3  uint32_t read_register(uint32_t address)
4  {
5      uint32_t reg;
6      asm(    "lw %0, 0(%1);"
7              : "=r;"(reg)
8              : "r"(address)
9          );
10     return reg;
11 }
12
13 void write_register(uint32_t address, uint32_t data)
14 {
15     uint32_t reg;
16     asm(    "sw %0, 0(%1);"
17             :
18             : "r"(data), "r"(address)
19         );
20 }
21
22
23 void config_led(int i)
24 {
25     uint32_t tri_state_led;
26     if (i>31) return;
27     //Read register
28     tri_state_led = read_register(GPIO_SWS_LEDS+GPIO_2TRI_OFFSET);
29     tri_state_led = REP_BITS(tri_state_led,i,1,OUTPUT);
30     //Write to register
31     write_register(GPIO_SWS_LEDS+GPIO_2TRI_OFFSET,tri_state_led);
32 }
33
34 void config_switch(int i)
35 {
36     uint32_t tri_state_sws;
37     if (i>31) return;
38     //Read register
39     tri_state_sws = read_register(GPIO_SWS_LEDS+GPIO_TRI_OFFSET);
40     tri_state_sws = REP_BITS(tri_state_sws,i,1,INPUT);
41     //Write to register
42     write_register(GPIO_SWS_LEDS+GPIO_TRI_OFFSET,tri_state_sws);
43 }
44
45 uint32_t read_switch(int i)
46 {
47     uint32_t data_sws;
48     if (i>31) return -1;
49     //Read register
50     data_sws = read_register(GPIO_SWS_LEDS+GPIO_DATA_OFFSET);
51     data_sws = GET_BITS(data_sws,i,1);
52     //Write to register
53     return data_sws;
54 }
55
56 void set_led(int i,int state)
57 {
58     if (i>31) return;
59     uint32_t data = ( (state>0) << i);
60     write_register(GPIO_SWS_LEDS+GPIO_2DATA_OFFSET,data);
61 }
62
63 ...

```

FIGURE 15 – Fichier gpio.c..

```

ELF Header:
  Magic:    7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                    EXEC (Executable file)
  Machine:                                RISC-V
  Version:                                0x1
  Entry point address:                    0xc0000000
  Start of program headers:               52 (bytes into file)
  Start of section headers:               8928 (bytes into file)
  Flags:                                   0x0
  Size of this header:                     52 (bytes)
  Size of program headers:                 32 (bytes)
  Number of program headers:                4
  Size of section headers:                 40 (bytes)
  Number of section headers:               10
  Section header string table index:       9

```

FIGURE 16 – ELF Header de test_switch_led.elf.

```

Section Headers:
  [Nr] Name                Type          Addr      Off      Size    ES Flg Lk Inf Al
  [ 0]                     NULL           00000000  000000  000000  00      0  0  0
  [ 1] .text                 PROGBITS      c0000000  001000  0003a8  00    AX  0  0  4
  [ 2] .data                 PROGBITS      00000000  002000  000000  00    WA  0  0  1
  [ 3] .rodata               PROGBITS      c00003a8  002000  000000  00    WA  0  0  1
  [ 4] .stack                 NOBITS        00000000  001000  100000  00    WA  0  0  1
  [ 5] .comment               PROGBITS      00000000  002000  00001b  01    MS  0  0  1
  [ 6] .riscv.attributes      RISCV_ATTRIBUTE 00000000  00201b  00001c  00      0  0  1
  [ 7] .symtab                SYMTAB        00000000  002038  0001a0  10      8 13  4
  [ 8] .strtab                STRTAB        00000000  0021d8  0000b7  00      0  0  1
  [ 9] .shstrtab              STRTAB        00000000  00228f  000051  00      0  0  1

```

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
 L (link order), O (extra OS processing required), G (group), T (TLS),
 C (compressed), x (unknown), o (OS specific), E (exclude),
 D (mbind), p (processor specific)

FIGURE 17 – Section Headers de test_switch_led.elf.

Symbol table '.symtab' contains 26 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	c0000000	0	SECTION	LOCAL	DEFAULT	1	.text
2:	00000000	0	SECTION	LOCAL	DEFAULT	2	.data
3:	c00003a8	0	SECTION	LOCAL	DEFAULT	3	.rodata
4:	00000000	0	SECTION	LOCAL	DEFAULT	4	.stack
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	.comment
6:	00000000	0	SECTION	LOCAL	DEFAULT	6	.riscv.attributes
7:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crt0.c
8:	c0000000	0	NOTYPE	LOCAL	DEFAULT	1	\$xrv32i2p1
9:	00000000	0	FILE	LOCAL	DEFAULT	ABS	test_switch_led.c
10:	c0000058	0	NOTYPE	LOCAL	DEFAULT	1	\$xrv32i2p1
11:	00000000	0	FILE	LOCAL	DEFAULT	ABS	gpio.c
12:	c0000098	0	NOTYPE	LOCAL	DEFAULT	1	\$xrv32i2p1
13:	00100000	0	NOTYPE	GLOBAL	DEFAULT	ABS	STACK_SIZE
14:	c00000f8	120	FUNC	GLOBAL	DEFAULT	1	config_led
15:	c0000280	96	FUNC	GLOBAL	DEFAULT	1	read_switch
16:	c0000098	48	FUNC	GLOBAL	DEFAULT	1	read_register
17:	c0000340	104	FUNC	GLOBAL	DEFAULT	1	set_led
18:	c0000170	136	FUNC	GLOBAL	DEFAULT	1	config_switch
19:	c0000000	28	FUNC	GLOBAL	DEFAULT	1	_start
20:	c00001f8	136	FUNC	GLOBAL	DEFAULT	1	config_button
21:	c00002e0	96	FUNC	GLOBAL	DEFAULT	1	read_button
22:	c0000058	64	FUNC	GLOBAL	DEFAULT	1	main
23:	00100000	0	NOTYPE	GLOBAL	DEFAULT	4	_sstack
24:	00000000	0	NOTYPE	GLOBAL	DEFAULT	4	_estack
25:	c00000c8	48	FUNC	GLOBAL	DEFAULT	1	write_register

FIGURE 18 – Table des symboles de test_switch_led.elf.

Dump of assembler code for function main:

```
0xc0000058 <+0>:      addi    sp,sp,-32
0xc000005c <+4>:      sw      ra,28(sp)
0xc0000060 <+8>:      sw      s0,24(sp)
0xc0000064 <+12>:     addi    s0,sp,32
0xc0000068 <+16>:     li      a0,1
0xc000006c <+20>:     jal     0xc0000170 <config_switch>
0xc0000070 <+24>:     li      a0,0
0xc0000074 <+28>:     jal     0xc00000f8 <config_led>
0xc0000078 <+32>:     li      a0,0
0xc000007c <+36>:     jal     0xc0000280 <read_switch>
0xc0000080 <+40>:     mv      a5,a0
0xc0000084 <+44>:     sw      a5,-20(s0)
0xc0000088 <+48>:     lw      a1,-20(s0)
0xc000008c <+52>:     li      a0,0
0xc0000090 <+56>:     jal     0xc0000340 <set_led>
0xc0000094 <+60>:     j       0xc0000078 <main+32>
```

Dump of assembler code for function read_register:

```
0xc0000098 <+0>:      addi    sp,sp,-48
0xc000009c <+4>:      sw      s0,44(sp)
0xc00000a0 <+8>:      addi    s0,sp,48
0xc00000a4 <+12>:     sw      a0,-36(s0)
0xc00000a8 <+16>:     lw      a5,-36(s0)
0xc00000ac <+20>:     lw      a5,0(a5)
0xc00000b0 <+24>:     sw      a5,-20(s0)
0xc00000b4 <+28>:     lw      a5,-20(s0)
0xc00000b8 <+32>:     mv      a0,a5
0xc00000bc <+36>:     lw      s0,44(sp)
0xc00000c0 <+40>:     addi    sp,sp,48
0xc00000c4 <+44>:     ret
```

Dump of assembler code for function write_register:

```
0xc00000c8 <+0>:      addi    sp,sp,-32
0xc00000cc <+4>:      sw      s0,28(sp)
0xc00000d0 <+8>:      addi    s0,sp,32
0xc00000d4 <+12>:     sw      a0,-20(s0)
0xc00000d8 <+16>:     sw      a1,-24(s0)
0xc00000dc <+20>:     lw      a5,-24(s0)
0xc00000e0 <+24>:     lw      a4,-20(s0)
0xc00000e4 <+28>:     sw      a5,0(a4)
0xc00000e8 <+32>:     nop
0xc00000ec <+36>:     lw      s0,28(sp)
0xc00000f0 <+40>:     addi    sp,sp,32
0xc00000f4 <+44>:     ret
```

FIGURE 19 – *Dump* de fonctions de test_switch_led.elf en utilisant gdb.

6 Gestion de projet

6.1 Comparaison du temps estimé au temps réel

Tout au long du projet, nous avons tenu un journal de bord hebdomadaire pour garder une trace de l'avancement du projet.

Ce journal de bord (voir section 6.2) présente un résumé des tâches accomplies chaque semaine et donne une estimation sur le temps passé au projet.

Si l'on compare ce journal de bord au GANTT prévisionnel (voir figure 20), on peut voir que l'implémentation du picorv32 sur la carte a environ pris trois semaines au lieu de deux.

La connexion des entrées et sorties (GPIO) a duré deux semaines au lieu de quatre.

Dans le diagramme GANTT n'apparaît pas une tâche qui a été sous-estimée grandement ; la chaîne de compilation croisée qui a duré deux semaines.

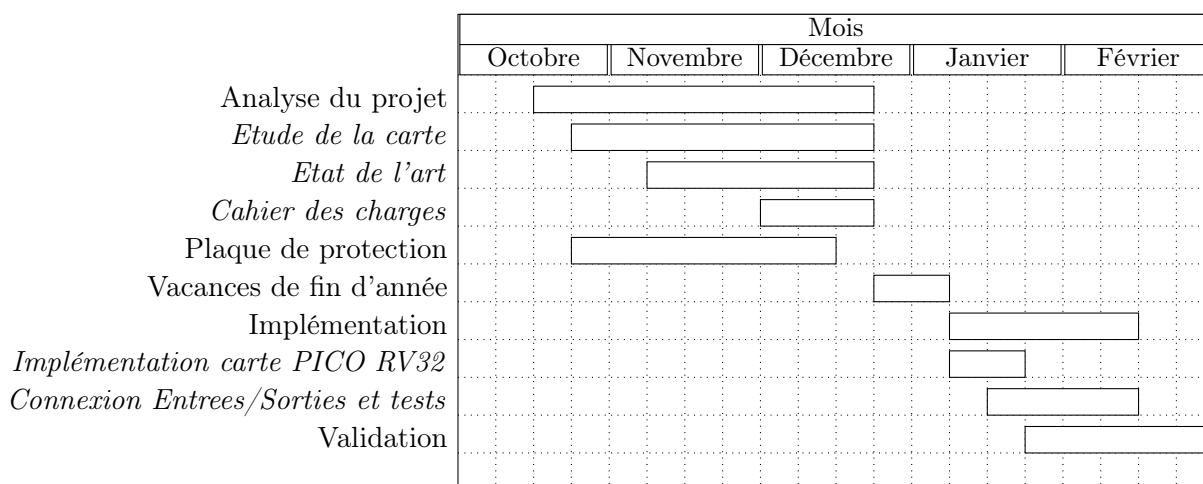


FIGURE 20 – GANTT prévisionnel du Cahier des Charges.

6.2 Journal de bord

Semaine 1 - Janvier 2024

- Installation de Vivado 2023.2 avec le support des boards ZYBO. : MORAL
- Ajout des boards files contenant la Zybo Z7-20 : MORAL
- Creation d'un projet vivado, packaging de l'IP picorv32_axi_0 et création d'un premier bloc design : MORAL
- Recherches sur l'IP du ZYNQ, comment connecter le picorv32 à la mémoire du processeur : MORAL

Semaine 2 - Janvier 2024

- Reunion de projet et découpe en tâches :
 - Recherche AXI : GUICHETEAU

- Recherche PCPI, comment désactiver : ASSIER
- Recherche Processing System : MORAL
- Modification des DQS_TO_CLK_DELAY : MORAL
- Modification des connexions entre l'axi interconnect et rst_ps7_0_50M : MORAL
- Implementation : MORAL

Semaine 3 - Janvier 2024

- Recherches sur le PCPI et documentation : ASSIER
- Recherche sur comment tester du code dans le bloc design : ASSIER - MORAL
- Recherches sur le debug ; ajout d'un ILA : MORAL
- Reunion de projet avec l'encadrant et découpe en tâche
 - Recherche toolchain : ASSIER
 - Recherche sur comment interfacer le processeur et ajouter du code compilé : GUICHETEAU
 - Recherche sur comment on crée un espace d'adressage pour le processeur : GUICHETEAU
- Documentation toolchain et préparation d'un code qui utilise la mémoire : ASSIER
- Mise à jour du design Vivado pour utiliser le port AXI_S_HP0 pour interfacer la mémoire : MORAL
- Création d'un design utilisant une ROM implémentée via de la BRAM : MORAL
- Test mémoire pour la ROM : ASSIER
- Fichier de documentation pour setup la Toolchain RV32I : ASSIER

Semaine 4 - Janvier 2024

- Test implémentation du fichier .coe en bootrom : ASSIER - MORAL
- Modifications du bloc design pour observer les transactions sur le bus AXI (ajouts d'ILA) : MORAL
- Recherches pour faire fonctionner le fichier "test" : ASSIER - MORAL
- Validation et correction du test qui permet d'interfacer la ROM et la mémoire RAM du PS : MORAL
- Ajout de constantes dans le bloc design pour supprimer les warnings du PCPI et de l'IRQ : MORAL

Semaine 5 - Fevrier 2024

- Réunion
- Recherche interfaçage picorv32 - GPIO : GUICHETEAU
- Implémentation interface picorv32 - GPIO : GUICHETEAU MORAL
- Ecriture de documentation sur le fonctionnement du bloc GPIO AXI : MORAL
- Fichier test des GPIO LED en assembleur : MORAL - ASSIER
- Fichier test des GPIO BUTTONS & SWITCH : ASSIER - MORAL

Semaine 6 - Fevrier 2024

- Reunion avec l'encadrant
- Modification des adresses dans le bloc design : MORAL
- Augmentation de la DDR accessible : MORAL
- Optimisation des AXI GPIO : MORAL
- Ajout de commentaires et création, test et validation d'un fichier ASM : MORAL
- Augmentation taille de la BRAM : MORAL
- Travail sur la chaîne de cross compilation, test linker et premier push : MORAL

Semaine 7 - Fevrier 2024

- Modification de la chaîne de cross compilation : (MORAL)
 - Ajout d'un crt0
 - Modification du linker script
 - Modification du linker script pour mettre le crt0 à l'adresse 0xc0000000 après avoir essayé via attribute
 - Modification du stack pointer via la fonction `_start`
 - Validation d'une boucle 'while 1, a = a+1' écrite en C sur la carte
- Creation de fonctions pour interragir avec le GPIO : MORAL - ASSIER
- Makefile pour cross compiler les tests, générer les .coe, .elf et .bin : MORAL
- Ecriture du rapport : MORAL
- Poster : MORAL - ASSIER

Semaine 8 - Fevrier 2024

- Correction et mise à jour du test des switch et leds en C : MORAL
- Validation de tout les tests : MORAL
- Rapport : MORAL
- Nettoyage du git : MORAL
- Création d'un guide d'utilisation du projet : MORAL

6.3 Problèmes rencontrés

Au cours de ce projet, nous avons rencontré quelques problèmes :

- Apprentissage du bus AXI, fonctionnement
- Code assembleur, mauvaises adresses générées et donc problème lors du test sur carte (valeur de lui).
- Recherches sur comment exécuter du code sur la carte
- Compréhension des fichiers au format .coe
- Recherches sur l'activation des PCPI et IRQ du processeur
- Compréhension des erreurs de l'AXI Interconnect
- Apprentissage du linker script et du crt0
- Désactivation du Dual Core ARM de la carte impossible
- Manque de registres CSR pour faire tourner un RTOS
- Répartition du travail et engagement des collaborateurs sur le projet

Malgré ces problèmes, la plupart des objectifs ont été atteints.

Par rapport au travail attendu dans le cahier des charges, il manque :

- L'exécution de code C sur d'autres architectures et sur notre processeur pour vérifier le même résultat et les performances
- La simulation d'entrées non valides provenant d'autre composants
- Les tests de performances

7 Conclusion

Ce projet nous a permis de nous familiariser sur le fonctionnement de Vivado, les *block design*, le protocole de communication AXI, l'assembleur RISC-V, la compilation croisée ainsi que bien d'autres éléments.

Néanmoins, ce projet pourrait être amélioré sur plusieurs points.

Dans le stade actuel du projet, pour faire tourner un programme on doit re-compiler un *bitstream* et re-flasher la carte. Ce processus prend du temps et nécessite l'utilisation de Vivado ; il serait intéressant d'interfacer la flash ou le support de carte SD du PS avec le processeur implémenté pour gagner en temps et en simplicité.

De plus, notre implémentation prend des ressources de la carte FPGA (voir figure 21). On utilise 14% de BRAM de la carte dans le design pour obtenir 64KB de ROM. Cette taille nous limite et si l'on veut faire tourner des programmes plus grands on risque de devoir demander plus de ressources de la carte FPGA.

On est également limité par l'espace d'adressage (32b).

La chaîne de compilation croisée pourrait quant à elle être améliorée en réalisant des optimisations au niveau du crt0.

Il serait intéressant d'essayer de communiquer avec le ZYNQ7 Processing System via un port *32b GP AXI Slave* pour avoir accès au *Central Interconnect* et aux autres ressources de la carte.

Le processeur implémenté pourrait également être changé, car notre implémentation ne dispose pas de CSR (*Control and Status Register*). Le manque de registre **MTIME** par exemple rend difficile l'implémentation de RTOS (*Real Time Operating System*).

On ne dispose pas non plus de MMU ce qui rend difficile l'implémentation d'un kernel linux.

8 Annexe

8.1 Bibliographie

- Zybo Z7 - Digilent Reference. (n.d.). Retrieved from <https://digilent.com/reference/programmable-logic/zybo-z7/start>
- Digilent. (n.d.). Zybo Z7 Reference Manual. Retrieved from <https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual>
- Wolf, C. X. (2015-2021). picorv32. Retrieved from <https://github.com/YosysHQ/picorv32>
- Digilent. (n.d.). Installing Vivado and Vitis. Retrieved from <https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis>
- LupLab @ University of California, Davis. (2021-2023). RISC-V Instruction Encoder/Decoder. Retrieved from <https://luplab.gitlab.io/rvcodecs/>
- Risc-V GNU Toolchain. (n.d.). Copyright (c) 2016, The Regents of the University of California (Regents). All Rights Reserved. GNU GENERAL PUBLIC LICENSE, Version 2, June 1991. GitHub. Retrieved from <https://github.com/riscv-collab/riscv-gnu-toolchain>
- Athalye, A. (n.d.). Copyright (c) bin2coe Retrieved from <https://github.com/anishathalye/bin2coe>
- Moral, A. (2024). Projet RV32-ZYNQ [GitHub repository]. <https://github.com/AlexandreMoG/RV32-ZYNQ/>

8.2 Résultats d'implémentation

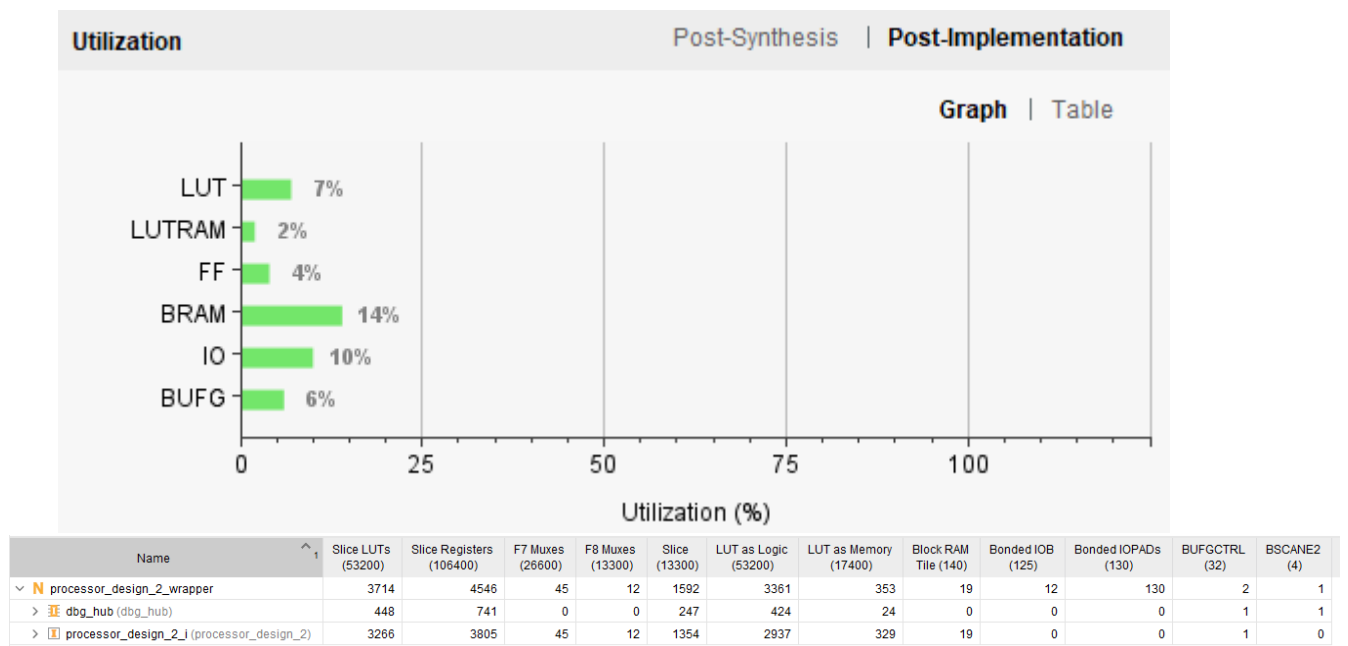


FIGURE 21 – Ressources utilisées par une implémentation du bloc design.