1 Implémentation matérielle

Cette partie a été réalisé par Quentin Forcioli, responsable de la partie Hardware

1.1 Introduction

Tout l'objet de cette partie est de concevoir une design à l'aide de Vivado pour exécuter les calculs de la partie Software. Elle doit fournir des exemple de matériel pour qu'il test leur logicielle et est aidée par la partie HLS pour accélérer les calculs.
La finalité des design est de permettre d'embarquer les calculs ainsi accélérés pour une utilisation sur par exemple du matériel roulant.

1.2 Demo multi-CPU

Une première démo a été conçu pendant le développement des algorithmes jusqu'à l'arrivée des IP HLS.
On profite ainsi de l'absence de dépendance avec les autres parties pour explorer des architecture de système.

1.2.1 Concept et problem

Ce design Multi-CPU se propose de juxtaposer au 2 processeur ARM embarqué dans le *Zynq*, un softcore *microblaze* implémenté dans la logic programmable (PL/FPGA). L'idée étant de faire tourner des codes sur les 2 processeur ou utilisé le *microblaze* pour du contrôle en utilisant les interruption.
Un premier design a été réalisé :

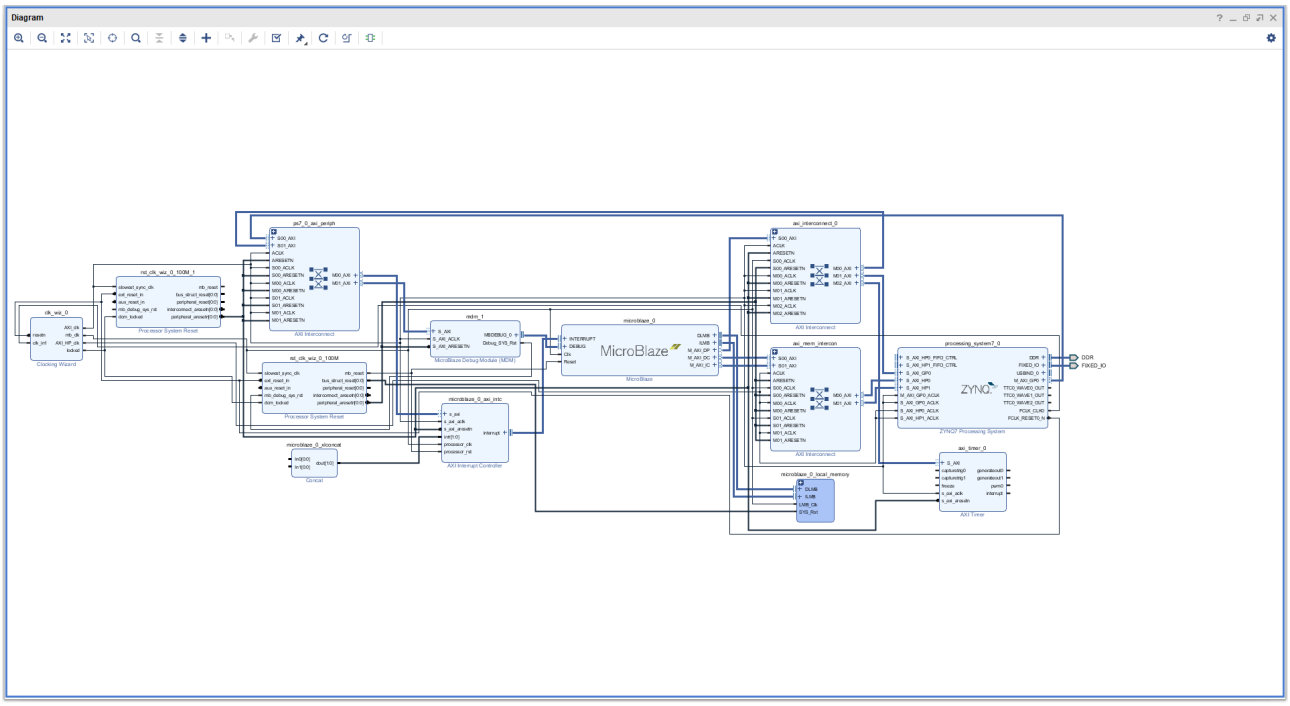

Figure 1 – Exemple de design pour le Multi CPU

Ce design permet de faire tourner un programme sur le FPGA et sur le microblaze.

Maintenant que le microblaze est rajouté, il peut être intéréssant de pouvoir lui passer des donnée. Le microblaze est déja connecté à la RAM du ZYNQ donc en théorie il peut déja recevoir et envoyé des donnés.
On peut déja les faire s'exécuter tout les 2 depuis la RAM du ZYNQ (cela permet d'agrandir le heap et le stack pour éviter les dépassement).

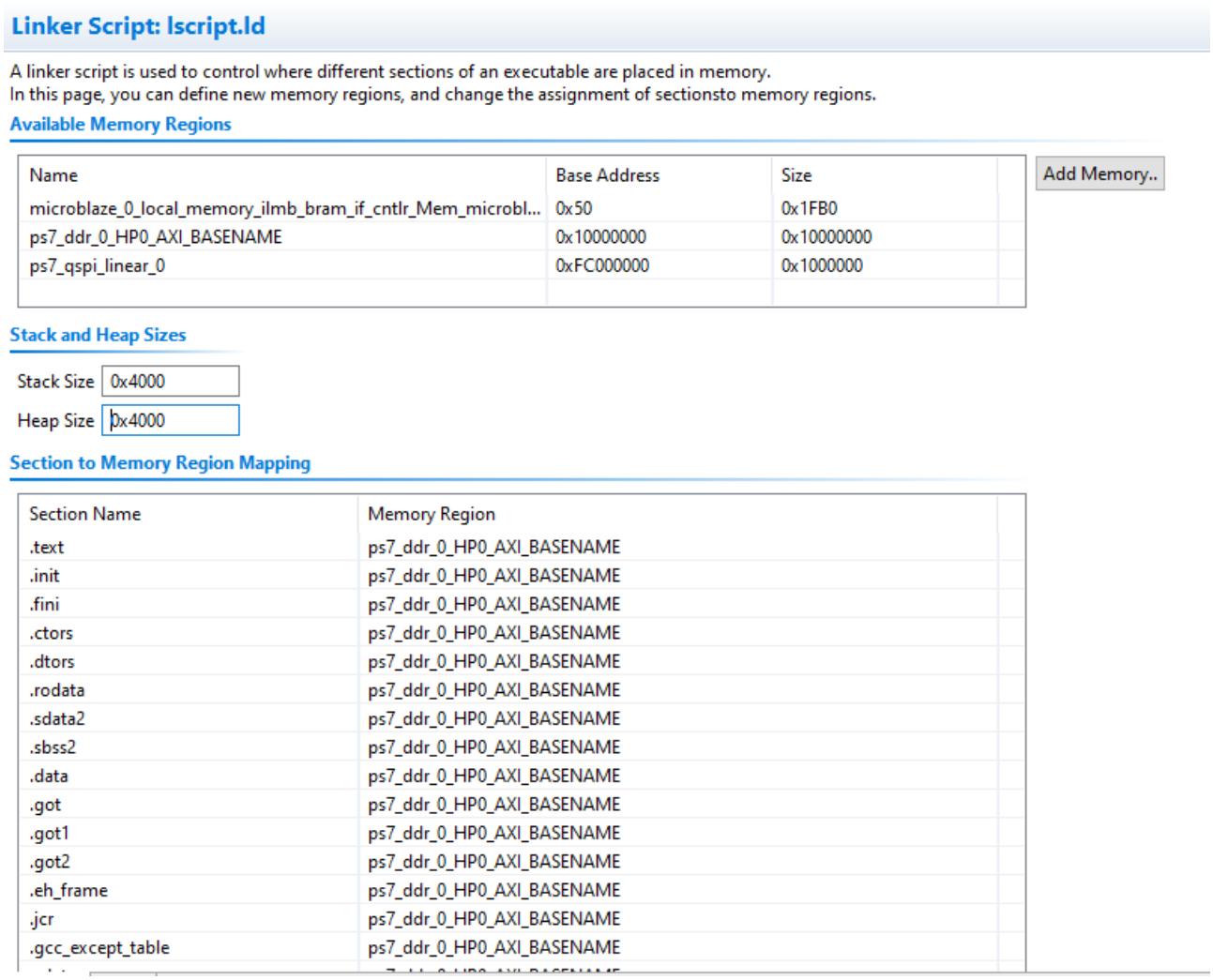

FIGURE 2 – Linker script du microblaze : exécution depuis la DDR et stack et heap étendus

Une des chose que l'on voudrait pouvoir faire c'est passé une grande quantité de donnée et des instruction au microblaze.
Du fait qu'il a accès à la RAM du *Zynq* il faudrait juste pouvoir lui envoyer des instructions. Si possible dans un emplacement fixe de la mémoire.

1.2.2 Utilisation de la BRAM

Pour passer des instruction du *Zynq* au *microblaze*, On décide d'utiliser les BRAM.

On crée ainsi un block design qui a en plus du lien avec la DDR du *Zynq*, a aussi des BRAMs. A l'intérieur de celles-ci, on pourra placer des adresses pour les données et des instructions sur ce qu'il faudra en faire.

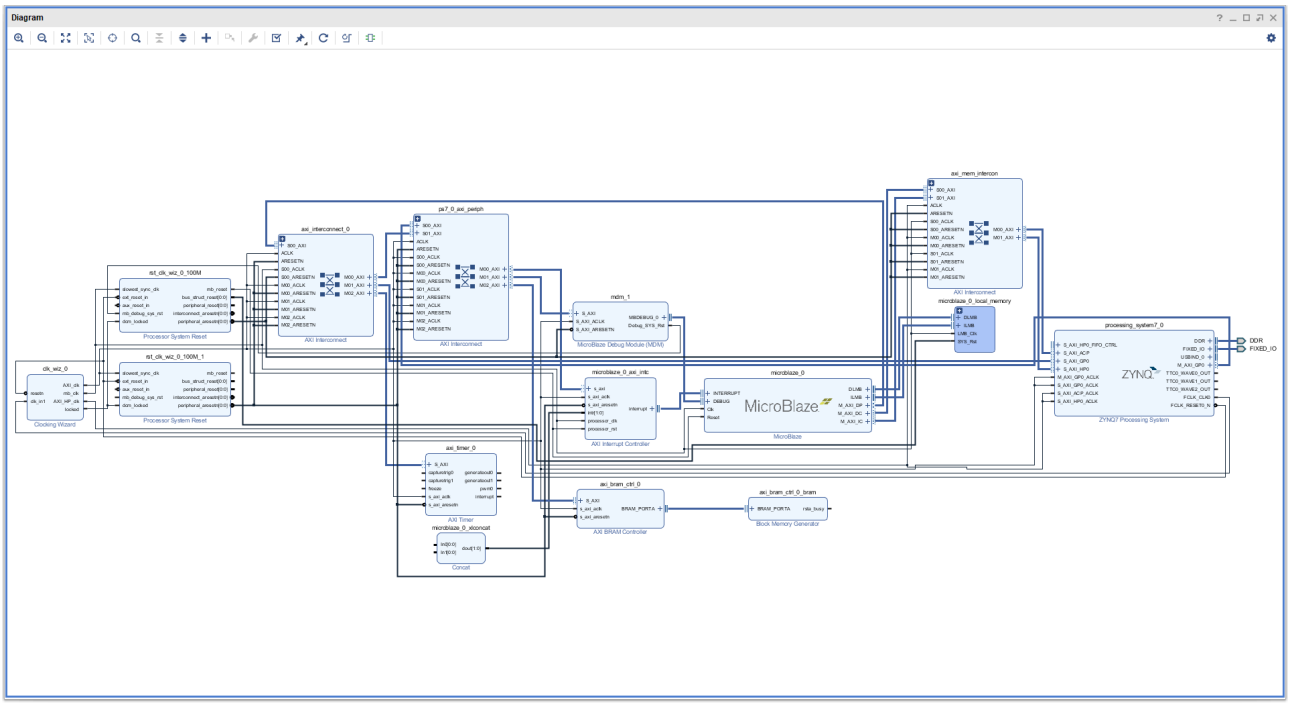

FIGURE 3 – Block design pour tester la BRAM :

1.2.3 Passage de données vers un microblaze

Une fois cela fait la BRAM on a développé 2 applications : une pour le *Zynq* et une pour le *microblaze*.

```c

int main()
{
  XTime Start_Time, End_Time;

    init_platform();

    print("Zynq:Hello World\n\r");
    XTime_GetTime((XTime *) &Start_Time
    );
    char * test =(char*)
    XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR;

    memcpy(test,"un message\n",20);

    while(1){
        XTime_GetTime((XTime *) &
    End_Time);
        sprintf(test,"message Ã %lli",(
    End_Time - Start_Time));
       sleep(1);
    }

```

```c
int main()
{
  char* test=(char*)
   XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR;
    init_platform();

    print("micro:Hello World 2\n\r");
  sleep(10);




    while(1){
      printf("micro:%s\n",test);
      sleep(1);
    }
}
```

FIGURE 4 – Code du microblaze(droite) et du zynq(gauche) pour l'envoie et la réception de message

On peut ainsi facilement passer des adresses du Zynq au microblaze grace à la BRAM.

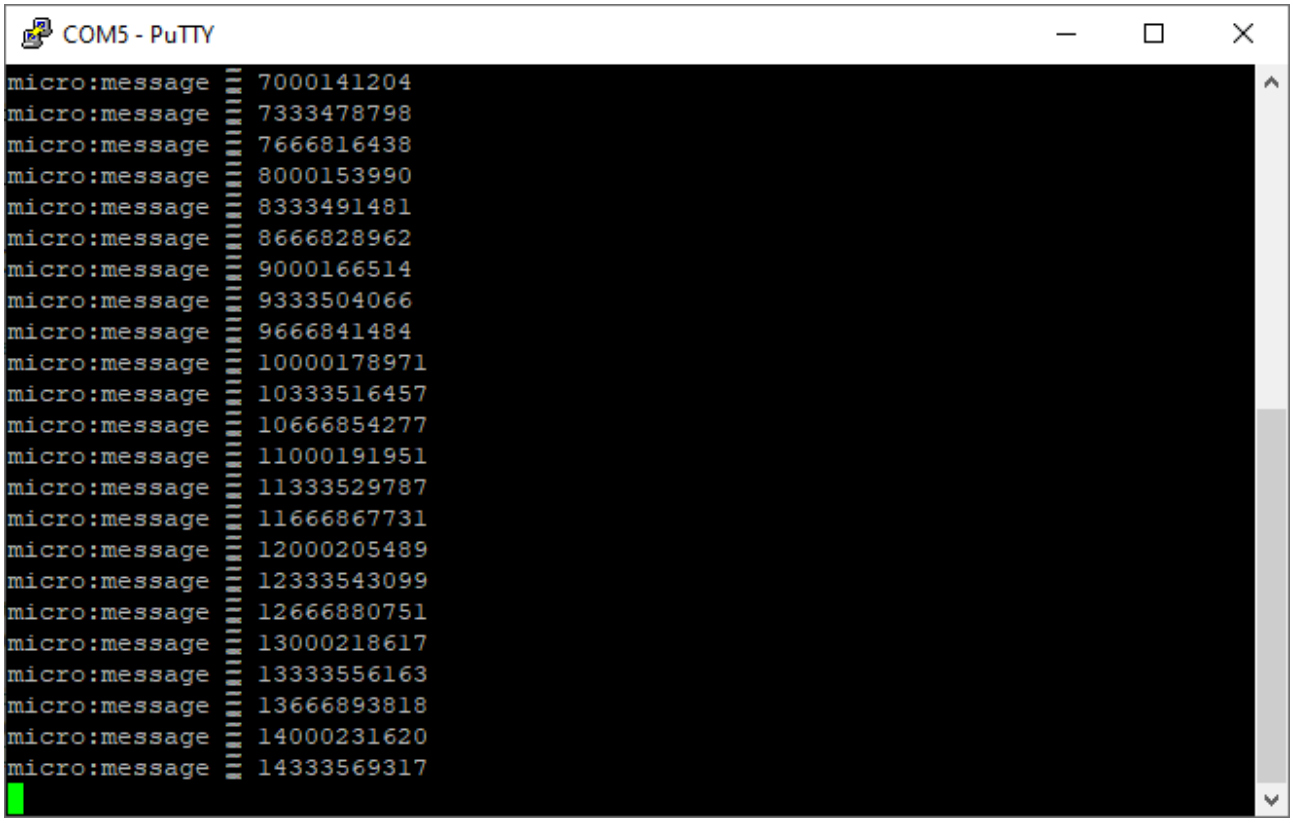

FIGURE 5 – Démonstration de la BRAM : le microblaze affiche un message envoyé depuis le Zynq

La finalité de ce design a été de permettre à l'équipe logicielle de testé leur programme sur la carte.

1.3 IP HLS

Dès que les première IP HLS ont été finalisés, il a été question de les intégrées dans un design. Un design a été conçu pour tester.

1.3.1 Concept

Pour accélérer, les programmes que l'on fait tourner sur les *Zynq*, la partie HLS à réaliser des accélérateur à l'aide des outils *Xilinx*. Ces accélérateurs ce présente comme des IP que l'on peut rajouter dans un block design.

Il faudra envoyer des donnés à ces IP pour quelle fasse les calculs à la place du *Zynq* : Beaucoup plus rapidement.

Ces IP sont commandés par le *Zynq* qui leur donne également des emplacement mémoire comme paramètre. Elles dispose d'accès directe à la mémoire DDR du *Zynq* pour pouvoir charger localement les donné et ranger les résultats.

1.3.2 Implémentation et clocking

Une fois les IP mis dans un repos communs et que ce repertoire est signaler à *Vivado* comme contenant des IPs, On peut les utiiliser dans un block design classique.

1.3.2.a Relier les IP aux Zynq

Les IPs HLS se presente comme suivant :

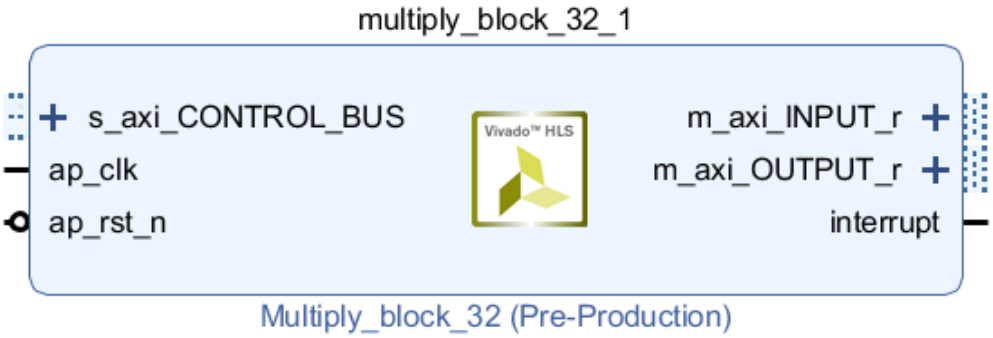

FIGURE 6 – Exemple d'IP on voit les différent port

Elles ont 6 ports :

- Un Port Slave AXILite appelé CONTROL_BUS qui sert a commandé l'IP.
- 2 Port Master AXI appelé INPUT et OUTPUT qui servent à l'IP pour accédé à des données en mémoire.
- Des port clk et rst qui permettent à l'IP d'avoir une clock indépendante (on utilisera cela pour avoir de meilleur performance)
- Un port interrupt qui n'as pas été utilisé mais pourrait être utilisé avec un microblaze pour déclenché une interruption dès que l'IP fini.

On relis cette IP au Zynq de la manière suivante :

- Les ports INPUT et OUTPUT sont relié au Slave ACP du ZYNQ pour pouvoir accédé à ces mémoire et avoir la cohérence des caches.(figure 7).
- On relis les port CONTROL_BUS au Master GP du Zynq pour pouvoir envoyer des commandes (les commandes étant moins critique ne pas passer par les interface rapide ne pose pas trop de problème)

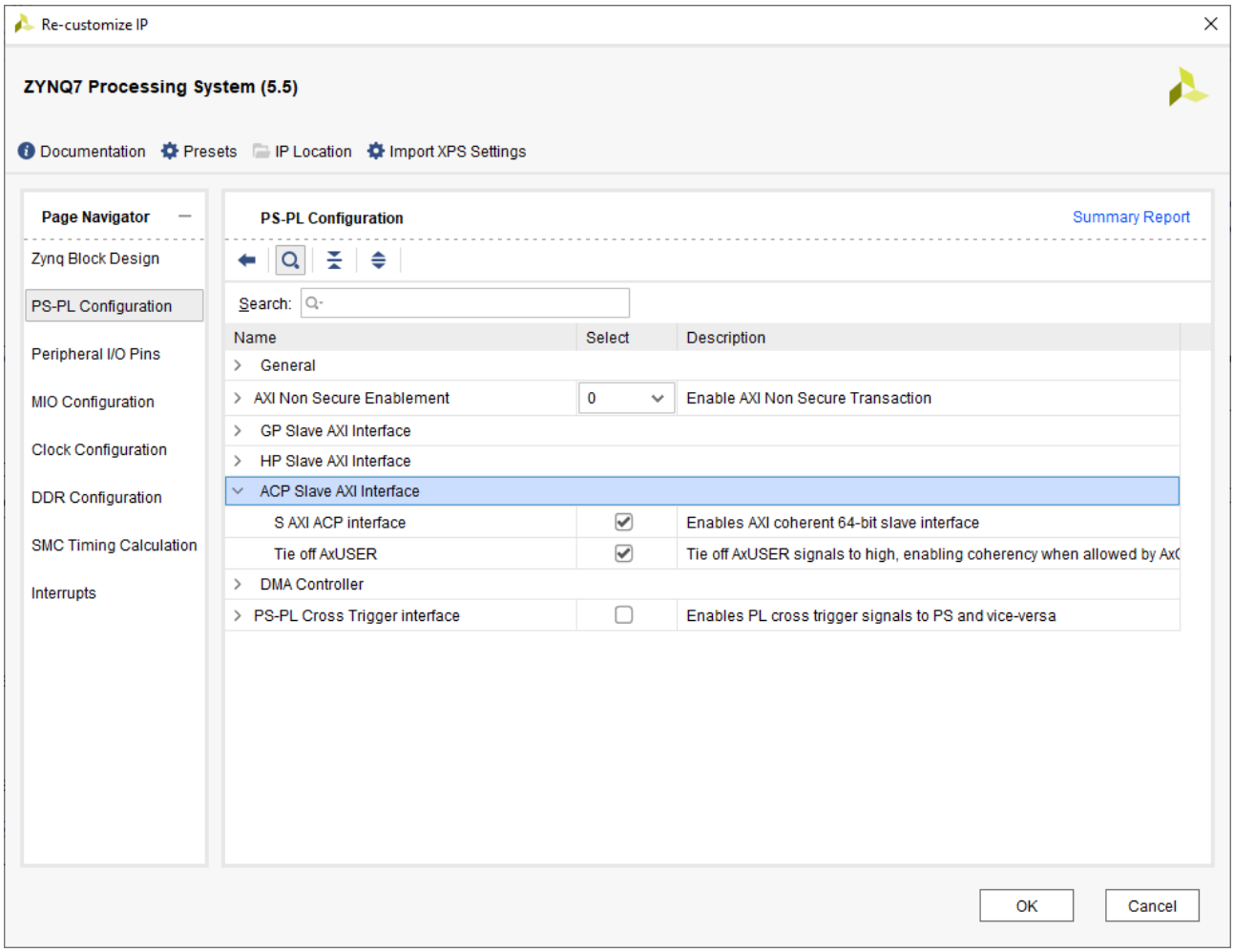

FIGURE 7 – Paramètre du Zynq : ce qu'il faut cocher pour avoir l'interface ACP et la cohérence des caches

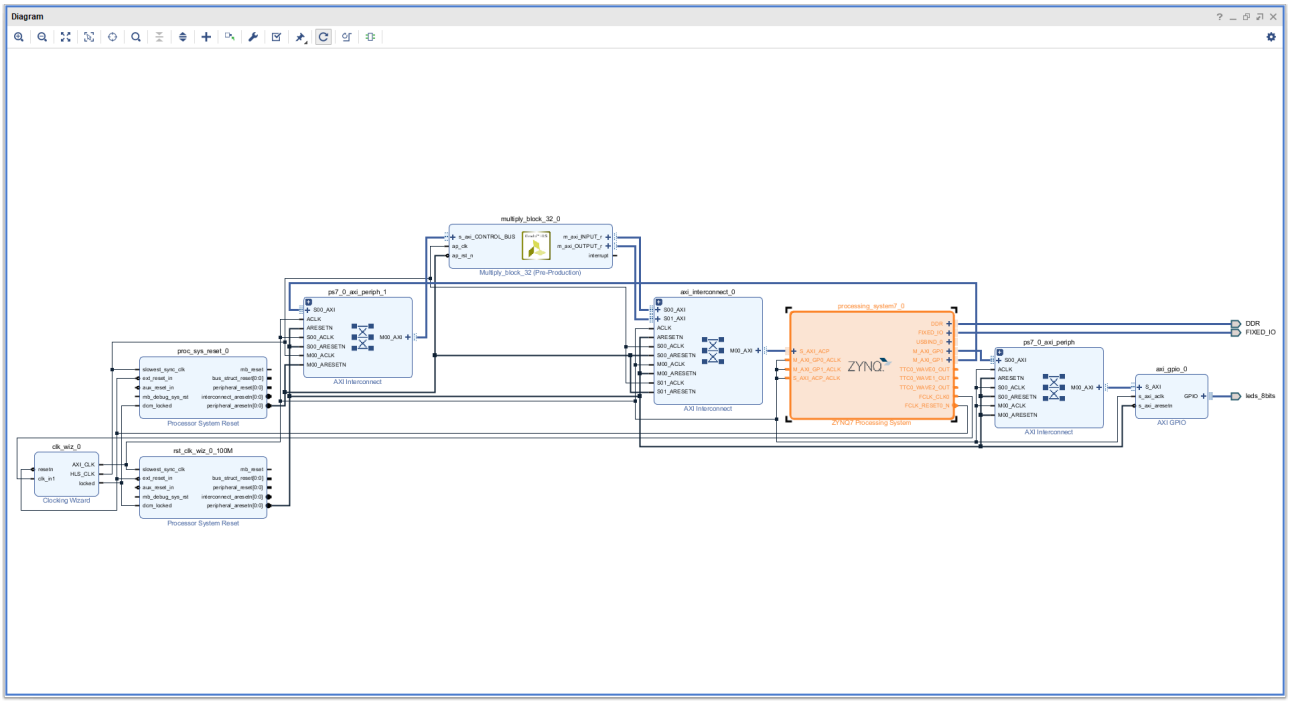

FIGURE 8 – Exemple de design pour l'IP multiply_block_32

On a fait en sorte dans les bock design que les clock des IP et la clock du HLS soit indépendantes. Cela permettra de tiré le plus de performance des IP plus tard.

1.3.2.b Fonction pour le SDK

A partir de là, après synthèse placement et routage du design, on dispose d'un entête spécial par exemple :

```
#include "xmultiply_block_64.h"
```

qui va permettre d'interagir simplement avec une IPS HLS.

Ainsi le code suivant contient 2 fonctions qui permettent d'initialiser une IP HLS(ici *mul64*) et de l'utiliser pour faire un calcul.

```
//fonction d'initilisation de L'IP
void init_multiply_block_ip(XMultiply_block_64* mb,XMultiply_block_64_Config*
    mb_c){
  int status=XMultiply_block_64_CfgInitialize(mb,mb_c);
  XMultiply_block_64_DisableAutoRestart(mb);
  XMultiply_block_64_InterruptGlobalDisable(mb);
  XMultiply_block_64_InterruptDisable(mb, 1);
  if(status!=XST_SUCCESS){
    printf("Multiply Block: init_failed \r\n");
  }
  printf("idle=%lx,ready=%lx,done=%lx\n",XMultiply_block_64_IsIdle(mb),
    XMultiply_block_64_IsReady(mb),XMultiply_block_64_IsDone(mb));
  printf("succes\n");
}

//fonction de lancement du calcul sur l'IP
void multiply_block_hw_call(XMultiply_block_64* mb_p,float* mA, float* mB, float
    * result){

  //on charge les adresse des donnÃ©es et les sorties
  XMultiply_block_64_Set_in_mA(mb_p, (u32)mA);
  XMultiply_block_64_Set_in_mB(mb_p, (u32)mB);
  XMultiply_block_64_Set_out_mC(mb_p, (u32)result);

  //on attend d'Ãªtre prÃªts.
  while(!XMultiply_block_64_IsReady(mb_p));

  //on lance
  XMultiply_block_64_Start(mb_p);

  //on attend d'avoir fini
  while(!XMultiply_block_64_IsDone(mb_p)){

  }
  //les rÃ©sultat sont dÃ©ja rangÃ© donc on a fini.
  return;

}
```

Grâce à l'utilisation des interfaces ACP et du fait qu'on ait activé le "*tie off AxUser*", nous n'avons pas besoin de vider le cache puisque la cohérence est maintenu à travers l'AXI et malgré les modification des données faite par l'IP.

1.3.2.c Optimisation des clock avec les Timings

On peut maintenant chercher à avoir le plus de performances possible avec nos IPs. Pour cela on va utiliser le fait qu'elle aient des clock indépendantes de celle de l'AXI. On peut ainsi changer leur fréquence séparément du reste du système.

Grâce au *timing report* on peut connaitre après implémentation quelle est la marge que l'on a par rapport au temps critique fixé par la clock. On peut ainis corriger cette clock itérativement pour aller jusqu'au moment ou l'on n'a plus aucune marge.

On a put par exemple monter la clock de l'IP *mul64* de 100 à 130 MHZ.

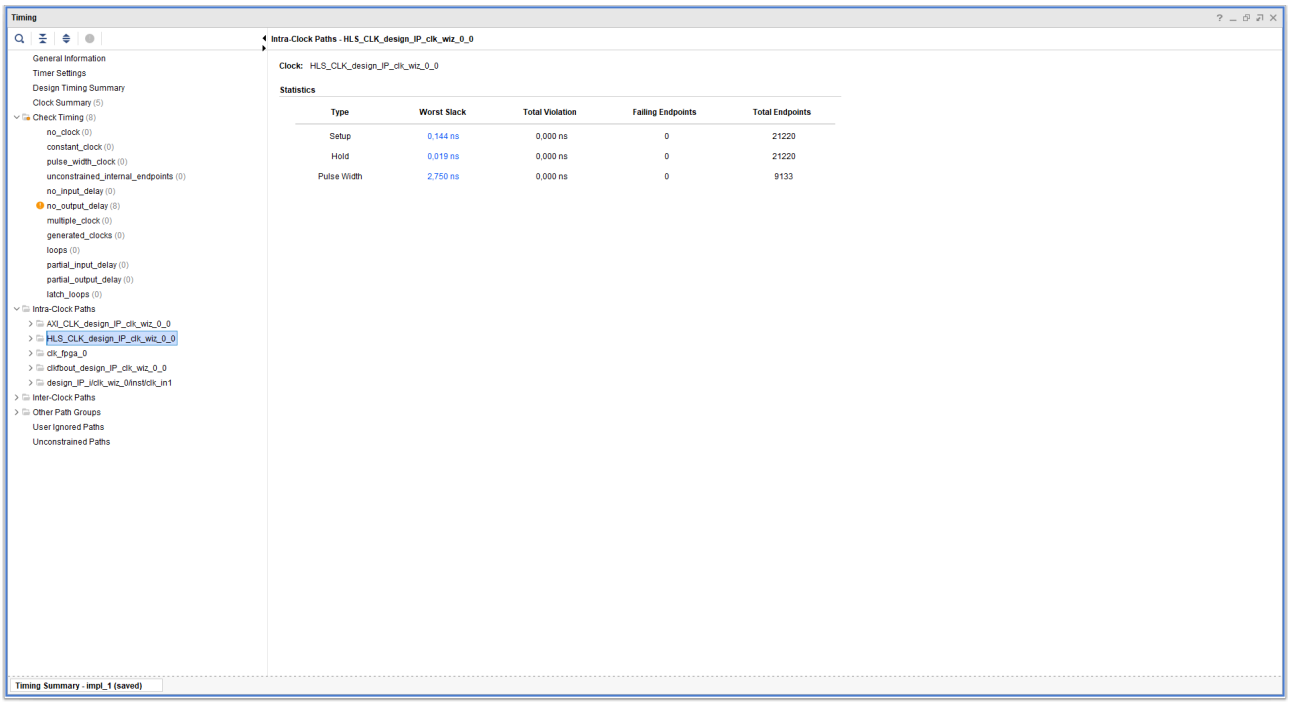

FIGURE 9 – Ecran du timing report permettant de connaitre le clock skew

1.3.3 Performances

En mesurant le temps grâce au timer du *Zynq* on peut évaluer les performance des IPs. Les IPs évalués sont :

- Block matrix multiplication 32 par 32(*mul32*) et 64 par 64(*mul64*)
- Le coefficient de Pearson (*pearson*)
- L'algorithme kmeans (*kmeans*

On compare ainsi les temps Software, Hardware et Hardware avec clock amélioré :

exemple d'IP	temps HW (μs) @100MHZ	temps SW (μs)	fréquence amélioré	temps HW amélioré(μs)
mul64	10280.17417	26132.32432	130 MHZ	7042.11712 useconds
mul32	2411.54655	3281.83483	125MHZ	1809.89489
pearson		5.25826	115 MHZ	21.19820
kmeans	1297.41742	1120841.13814	120MHZ	1082.07808

On a aussi essayé de serrer encore plus les timing en jouant sur la Clock des AXI et sur l'algorithme

de placement (Haute performance).
Avec les réglages suivant :

- AXI_CLK=180MHZ
- HLS_CLK=131MHZ

On a un temps de 6215.74174 μs soit plus de 4 fois moins de temps que la version SW classique.

1.4 Amélioration et future design

Il s'agirait ensuite de construire un design utilisant et les IP et le *microblaze* pour pouvoir, par exemple utiliser l'IP HLS de block matrix multiplication. Pour faire des multiplication de matrice plus grande en utilisant plusieurs fois l'IP sans avoir besoin de monopoliser le *Zynq*, utilisant ainsi les interruption sur le *microblaze*.