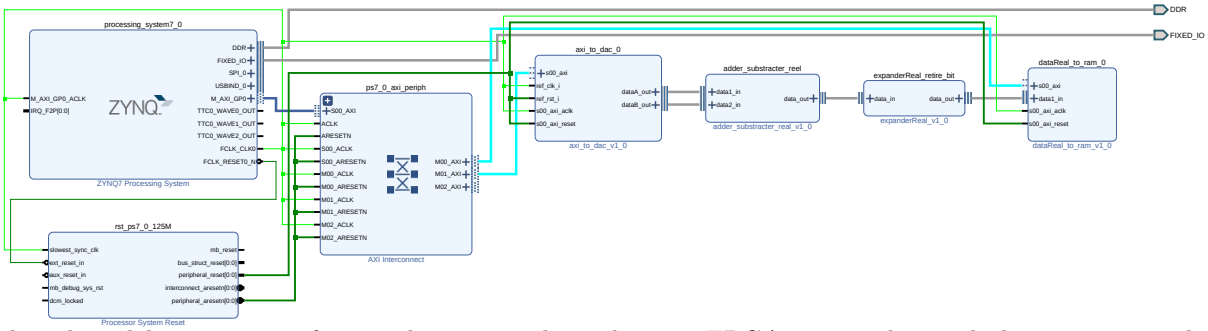
Coprocesseur pour le traitement parallèle de données

22 janvier 2020

Documents fournis : projet Vivado synthétisé et exemple de module noyau

Nous nous intéressons à l'"accélération" de calculs par l'ajout de co-processeurs reconfigurables dans le FPGA à côté du processeur ARM équipant le Zynq. Une des approches qui a permis l'accélération des traitements est l'architecture SIMD (*Single Instruction Multiple Data*) utilisée par de nombreux fondeurs, incluant les jeux d'instructions MMX puis SSE (*Streaming SIMD Extensions*) chez Intel, NEON chez ARM ou cartes graphiques, en particulier nVidia. Dans ces architectures, une même instruction est appliquée à plusieurs données simultanément renseignées dans les registres adéquats, évitant les multiples séquences d'appels des instructions (*opcodes*) selon le cycle *fetch-decode-execute* des architectures SISD (*Single Instruction on Single Data*).

Nous nous proposons d'aborder le problème de l'addition selon le schéma de traitement ci-dessous :



dans lequel le processeur fournit deux mots de 32 bits au FPGA qui se charge de les sommer et de renvoyer le résultat.

1. Sachant que les deux mots de 32 bits sont transmis au bloc `axi_to_dac` par des registres situés à l'adresse de base +4 et +8, et que le résultat du traitement est transmis au bloc `data_to_ram` en écrivant à son adresse de base+0 pour lire le contenu à l'adresse de base+8, démontrer votre capacité, en faisant appel aux commandes du shell, à additionner 6 et 9.
2. Afin de profiter du parallélisme de l'approche SIMD, construire les arguments de la somme afin d'obtenir la somme des nombres de 3 à 6 avec les nombres de 42 à 45, toujours depuis le shell, afin d'effectuer en une seule operation toutes les sommes. $\begin{pmatrix} 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} + \begin{pmatrix} 42 \\ 43 \\ 44 \\ 45 \end{pmatrix} = ?$
3. Afin d'abstraire les interfaces et d'éviter d'accéder aux ressources par leurs emplacements en mémoire, un pilote doit permettre les mêmes fonctionnalités. En partant de l'exemple de module fourni, démontrer l'échange entre le module noyau Linux et le FPGA : on constatera le bon fonctionnement en affichant le résultat par `dmesg`.
4. Plutôt que nécessiter de recompiler le module pour chaque nouvelle opération, nous désirons exposer une interface à l'utilisateur de façon à ce que deux écritures successives dans cette interface se traduisent par le stockage des arguments dans le FPGA, tandis que la lecture de cette interface permette l'obtention du résultat. Démontrer le bon fonctionnement d'un tel pilote.
5. Afin de profiter du parallélisme de l'approche SIMD, modifier le pilote pour accepter 4 paires d'octets successifs qui seront les arguments de la somme, et fournir à l'utilisateur les 4 résultats individuels de la somme. On pourra pour ce faire par exemple lire 4 fois sur l'interface de communication entre le noyau et l'espace utilisateur pour obtenir les 4 résultats. Démontrer la somme des nombres de 42 à 45 avec les nombres de 3 à 6.
6. L'additionneur peut se configurer comme soustracteur. Modifier la chaîne de traitement exécutée dans le FPGA pour pouvoir choisir, depuis le processeur, de soit fournir la somme, soit la différence des arguments. On pourra profiter du commutateur `switchReal` fourni par OscimpDigital pour sélectionner le signal routé, avec la sélection de la voie (0 ou 1) en écrivant dans le registre d'adresse de base+4. Démontrer le bon fonctionnement en appelant le bloc de traitement depuis le shell pour cette fois effectuer la soustraction entre les arguments de la question précédente.
7. Modifier le pilote pour configurer le FPGA pour effectuer l'une ou l'autre fonction. Quel appel système accessible depuis le pseudo-fichier dans `/dev` permet une telle configuration ? On pourra consulter dans `oscimpDigital/linux_driver` le pilote `switch_core`

Réponses

1. la séquence de commandes

   ```
   devmem 0x43c10004 32 6
   devmem 0x43c10008 32 9
   devmem 0x43c00000 32 0
   devmem 0x43c00008 16
   ```

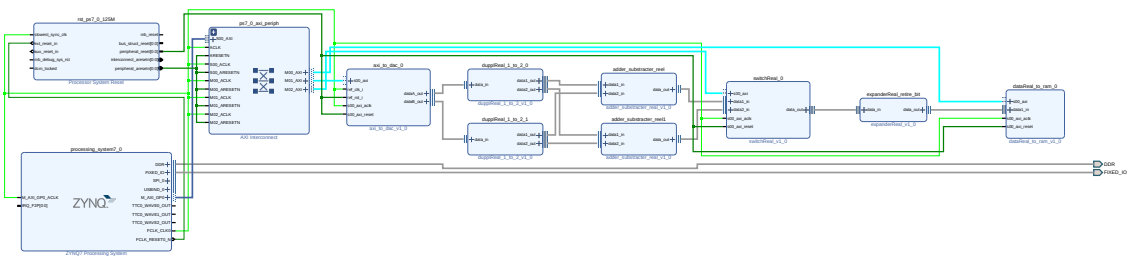
   fournit la réponse `0x000F`

2. la séquence de commandes

   ```
   devmem 0x43c10004 32 0x2a2b2c2d
   devmem 0x43c10008 32 0x03040506
   devmem 0x43c00000 32 0
   devmem 0x43c00008 32
   ```

   fournit la réponse `0x2D2F3133`
   qui est bien 45, 47, 49 et 51.

3.
```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/ioport.h>       // request_mem
#include <linux/io.h>           // ioremap
#include <linux/fs.h>           // fops
#include <linux/miscdevice.h>

#define IO_BASE1 0x43c00000
#define IO_BASE2 0x43c10000
#define mio 0

int hello_start(void);
void hello_end(void);

int hello_start()
{static void __iomem *ram,*dac;
 if (request_mem_region(IO_BASE1,8,"GPIO␣test")==NULL)
     printk(KERN_ALERT "mem1␣request␣failed");
 ram = (void __iomem*)ioremap(IO_BASE1, 8);
 if (request_mem_region(IO_BASE2,8,"GPIO␣test")==NULL)
     printk(KERN_ALERT "mem2␣request␣failed");
 dac = (void __iomem*)ioremap(IO_BASE2, 8);
 writel(4,dac+4);
 writel(2,dac+8);
 writel(1,ram+0);
 printk(KERN_INFO "res:%d\n",readl(ram+8));
 return 0;
}

void hello_end()
{printk(KERN_INFO "Goodbye\n");
 release_mem_region(IO_BASE1, 8);
 release_mem_region(IO_BASE2, 8);
}

module_init(hello_start);
module_exit(hello_end);
```

4. `copy_from_user` et `copy_to_user` dans les fonctions de read et write
5. décalages en entrée et masques en sortie
6. 

Le switch est configuré vers la voie 1 du soustracteur

```
devmem 0x43c20004 32 1  # soustraction
devmem 0x43c10004 32 9
devmem 0x43c10008 32 6
devmem 0x43c00000 32 0  # transfer
devmem 0x43c00008 16    # read result
0x0003
```

tandis que configurer le switch vers la voie 0 connectée à l'additionneur

```
devmem 0x43c20004 32 0  # addition
devmem 0x43c10004 32 9
devmem 0x43c10008 32 6
devmem 0x43c00000 32 0  # transfer
devmem 0x43c00008 16    # read result
0x000F
```

7. `ioctl()`