Communication CPU-FPGA

FPGA comme co-processeur de Xenomai/Linux temps-réel - Présentation Redpitaya/Zynq, bus de communications et communication CPU-FPGA

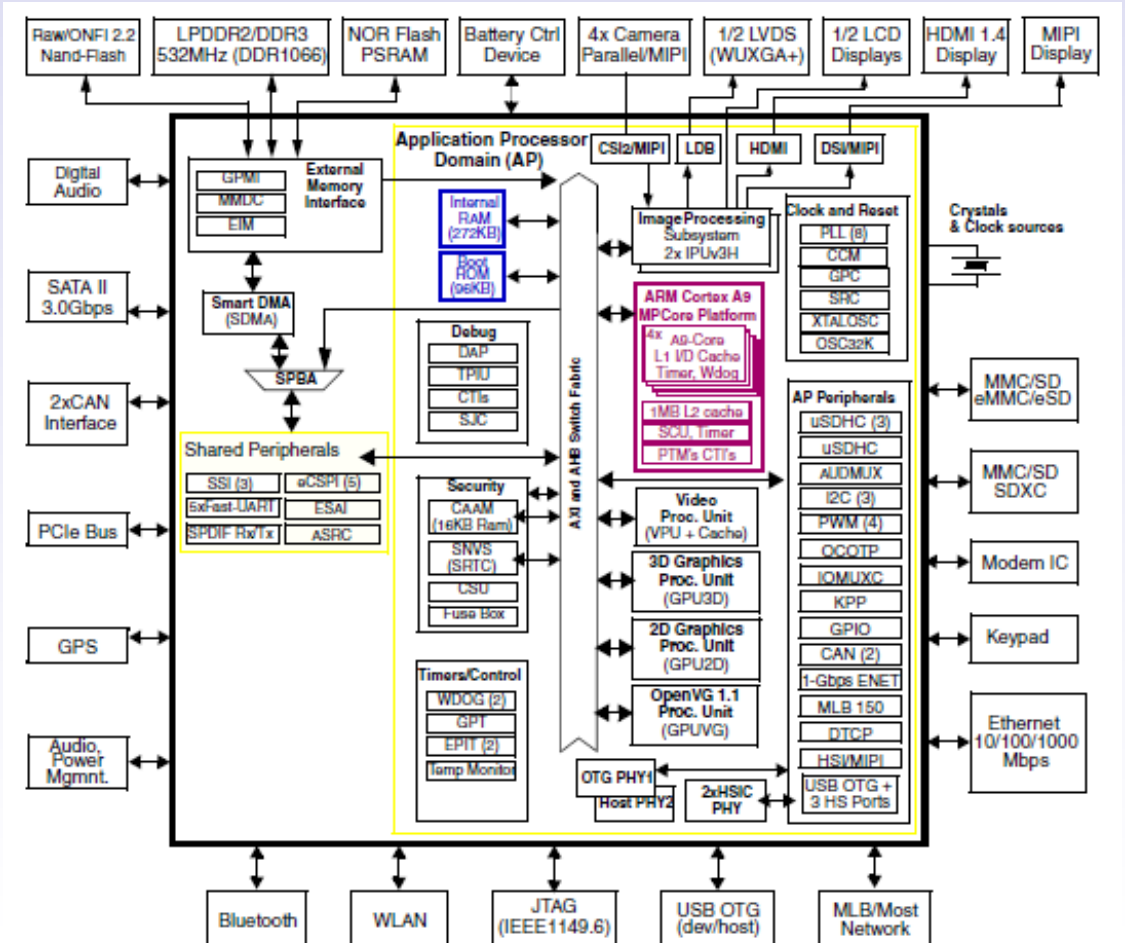
Gwenhaël GOAVEC-MEROU

17 janvier 2019

Slides disponibles sur
`http://www.trabucayre.com/enseignement/presentation_zynq.pdf`

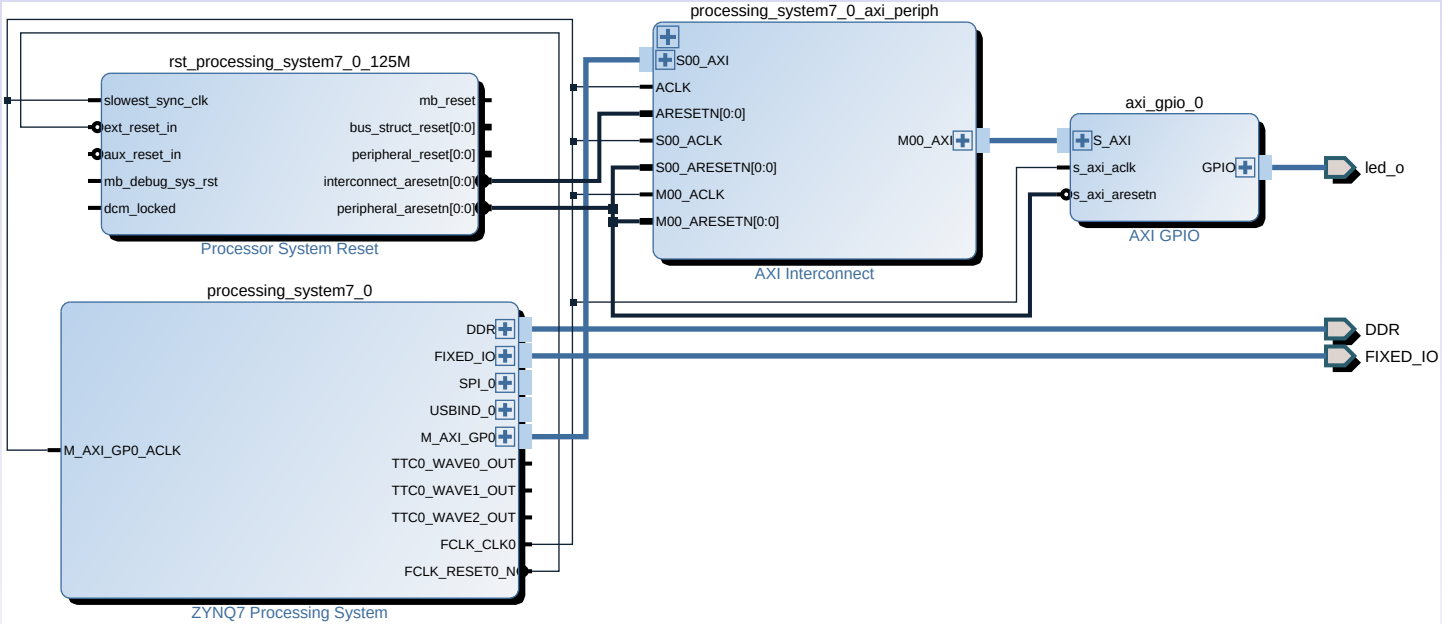
Motivation

- s'approprier le principe de communication entre un processeur et des périphériques matériels
- ne se limite pas simplement à la communication CPU/FPGA → vrai également entre un cœur et les périphériques matériels d'un processeur.



Rappel/acquis

Intégration du bloc `axi_gpio` dans le `FPGA` et exploitation depuis le processeur



Rappel :

- `Zynq Processing System` : partie `PS` du `Zynq` ;
- `Intercon` : décodeur d'adresse ;
- `axi_gpio` : un esclave `AXI` fournie par `Xilinx`

Bus de communication

Déjà vu : protocoles série (SPI, I2C).
Le sujet : protocoles parallèles.
Utilisés pour :

- la communication avec les composants de RAM, de stockage (NAND, NOR, (e)MMC, etc.) ;
- la communication entre un cœur de processeur et les contrôleurs matériels ;
- communication avec les FPGAs

Composés généralement :

- d'un bus d'adresse ;
- d'un ou deux bus de données (bi-directionnels ou maître-esclave et esclave-maître) ;
- signaux de contrôles (requête de lecture/écriture, validité de la donnée, acquittement, interruption)

Principe de fonctionnement

- un maître, n esclaves ;
- mémoire utilisée comme une zone d'accès au périphérique ;
- mémoire découpée : chaque esclave dispose de sa propre zone et d'une sous adresse ;

Attention : les accès à la mémoire sont alignés selon la taille (8bits, 16, 32 ou 64)

```
// ok: ecriture d'un short (16bits) a une adresse multiple de 2 (octets)
pos = 0x02; /* 1 << 1 */
*(unsigned short*)(ptr_fpga+pos) = 0x01;

// fail ecriture d'un short (16bits) a une adresse multiple de 1 (octet)
pos = 0x01; /* 1 << 0 */
*(unsigned short*)(ptr_fpga+pos) = 0x01;
```

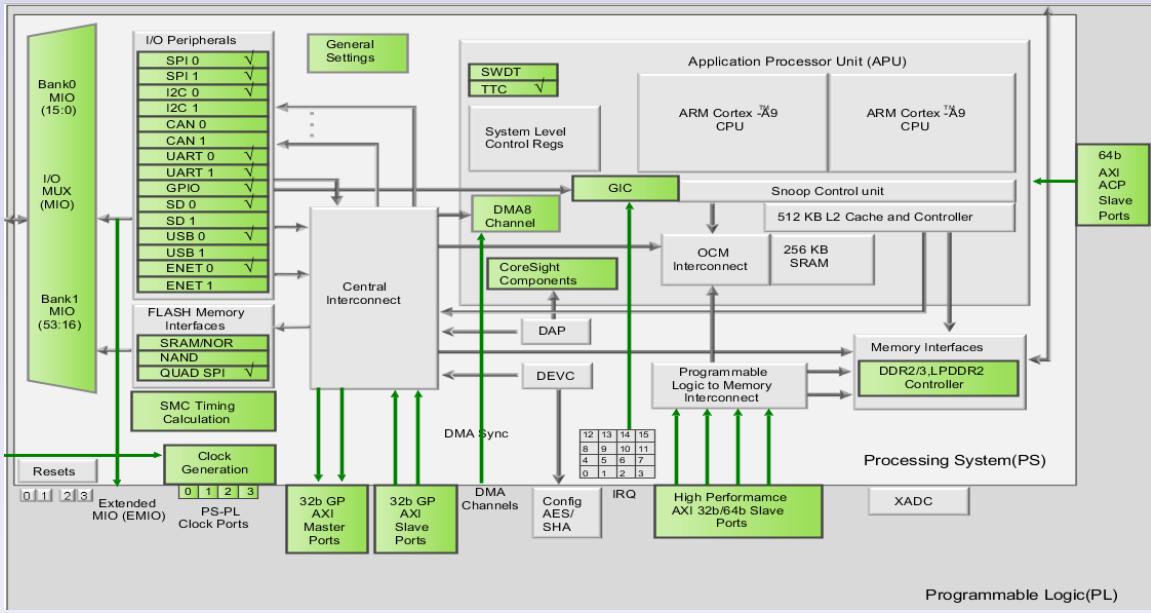
Un bus n octets n'implique pas obligatoirement des accès sur n octets :
ex : contrôleur SPI sur STM32F3/4 : comportement différent selon le type d'accès et la taille des données à transmettre.

Environnement matériel

Le matériel utilisé, une carte Redpitaya (base de Zynq).

Le même boitier contient un `SOC` et un `FPGA`
Caractéristiques :

- processeur double cœur Cortex A9 ;
- FPGA équivalent `artix7`.

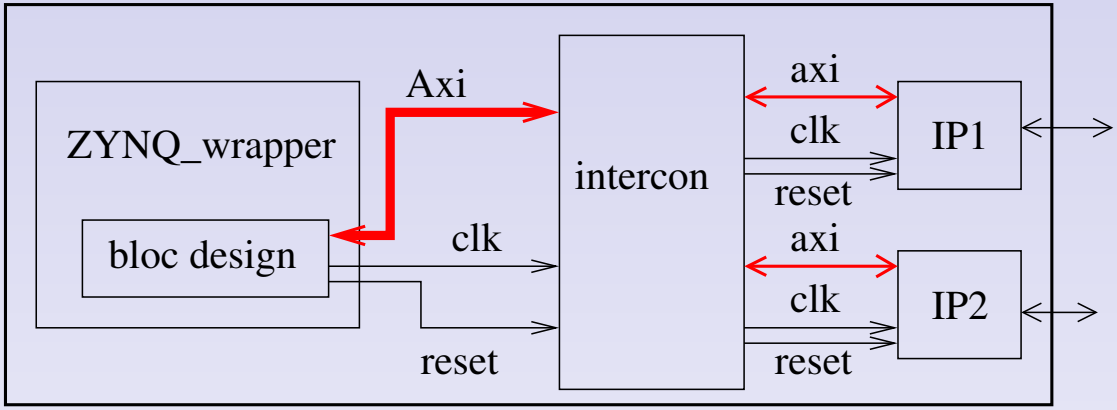


Communications : *AXI slave* et *AXI High-Performance*

- 32 bits de données ;
- 32 bits d'adresse ;
- DMA (Direct Memory Access) avec l'`AXI HP`.
- interruptions ;

Méthodologie également valable pour le bus **wishbone** et **AVALON**

Architecture des designs



Structure interne :

- un `wrapper` pour le bloc design ;
- exportation du bus `AXI`, de l'horloge et du signal de `reset` ;
- décodeur d'adresse pour communiquer avec plusieurs blocs (IPs) : intercon

$\Rightarrow$ simplifie/accélére la phase de développement d'`IP`.

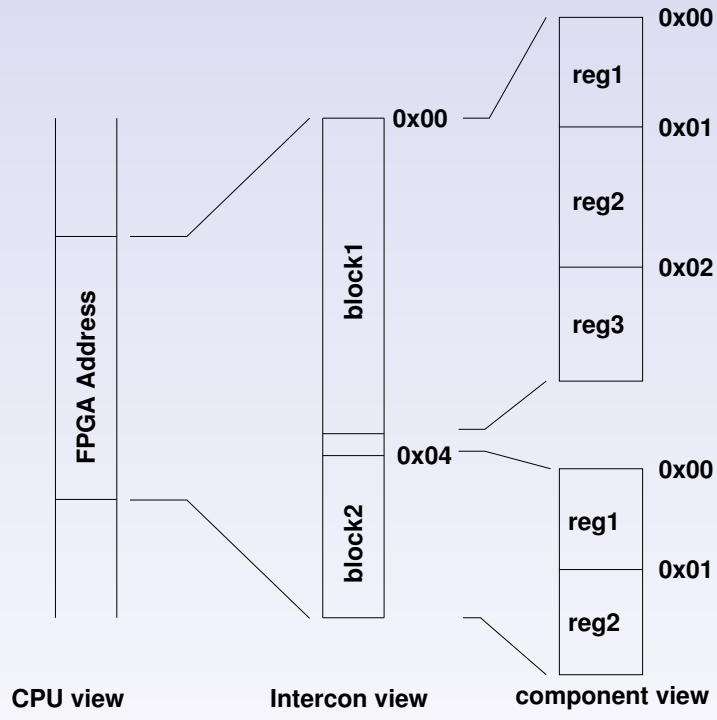
Structure globale automatiquement générée par le logiciel *Peripheral On Demand*[1] (Non traité dans ce TP).

1. `http://github.com/martoni/periphondemand`

Découpage d'adresse et abstraction

Principe :

- des plages d'adresses (1 Gb) dédiées au niveau CPU (adresse absolue) ;
- celle-ci est découpée en sous-adresses pour adresser chaque bloc indépendamment (`intercon` : adresse relative au début de la plage d'adresse) ;
- ce sous espace est également découpé pour accéder aux registres d'un bloc (gérée par le bloc, relatif à son adresse de base).



Présentation bus AXI4 lite

clk et reset : s00_axi_aclk et s00_axi_reset

Écriture :

Adresse/contrôle :

- s00_axi_awaddr
- s00_axi_awvalid
- s00_axi_awready
- s00_axi_bresp
- s00_axi_bvalid
- s00_axi_bready

Données :

- s00_axi_wdata
- s00_axi_wvalid
- s00_axi_wready
- s00_axi_wstrb

Lecture :

Adresse/contrôle :

- s00_axi_araddr
- s00_axi_arvalid
- s00_axi_arready

Données :

- s00_axi_rdata
- s00_axi_rvalid
- s00_axi_rready
- s00_axi_rresp

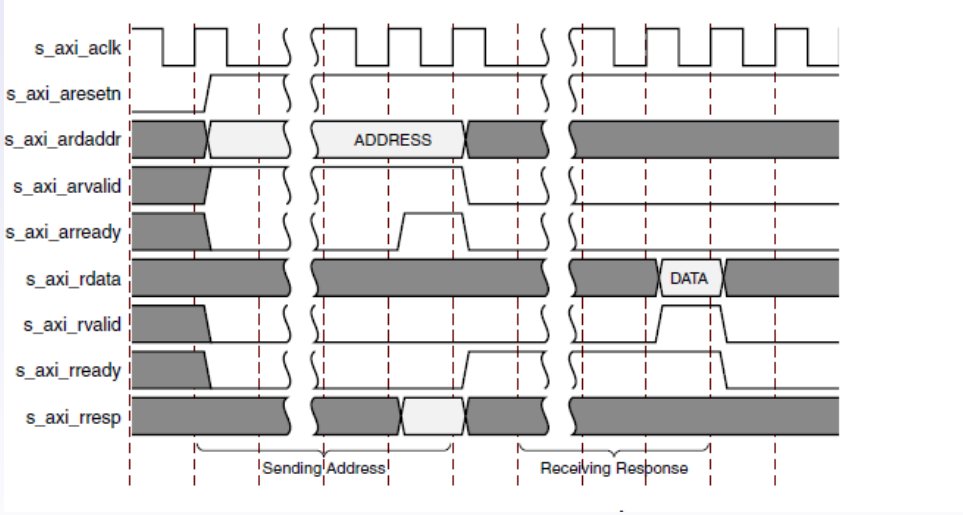
Présentation bus AXI4 lite : read

Adresse :

- s00_axi_araddr
- s00_axi_arvalid
- s00_axi_arready

Données :

- s00_axi_rdata
- s00_axi_rvalid
- s00_axi_rready
- s00_axi_rresp



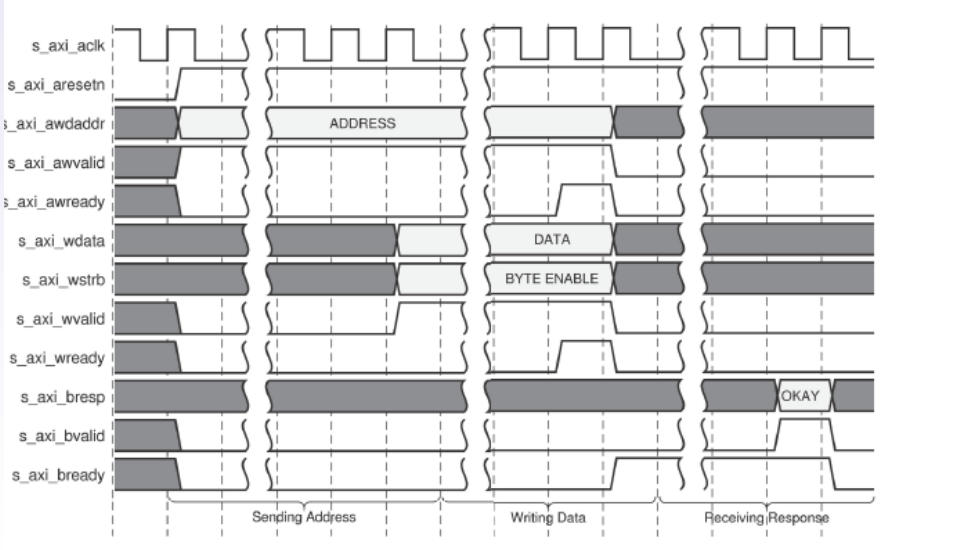
Présentation bus AXI4 lite : write

Adresse/contrôles :

- s00_axi_awaddr
- s00_axi_awvalid
- s00_axi_awready
- s00_axi_bresp
- s00_axi_bvalid
- s00_axi_bready

Données :

- s00_axi_wdata
- s00_axi_wvalid
- s00_axi_wready
- s00_axi_wstrb



Gestion de la communication

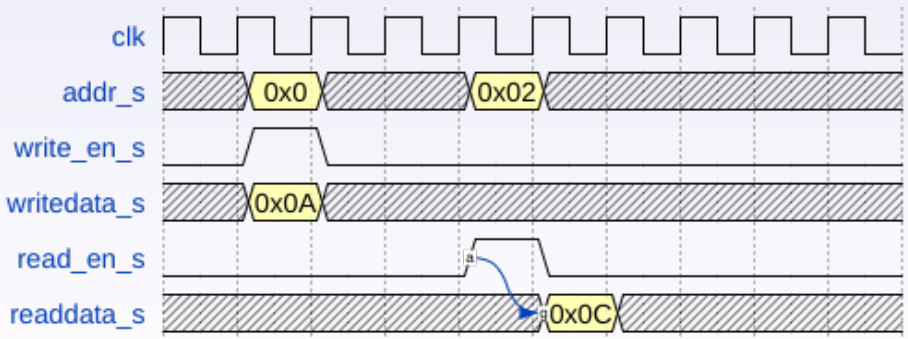
Utilisation d'un `wrapper` pour confiner la partie complexe.

Gestion lecture :

```
read_bloc : process(clk, reset)
begin
  if reset = '1' then
    readdata_s <= (others => '0');
  elsif rising_edge(clk) then
    readdata_s <= readdata_s;
    if read_en_s = '1' then
      case addr_s is
      when REG_ID =>
        readdata_s <= std_logic_vector(→
            ↪to_unsigned(id,dat_size));
      when REG_RESULT =>
        readdata_s <= result_s;
      when others =>
        readdata_s <= (others => '0');
      end case;
    end if;
  end if;
end process read_bloc;
```

Gestion écriture :

```
write_bloc : process(clk, reset)
begin
   if reset = '1' then
     op1_s <= (others => '0');
     op2_s <= (others => '0');
   elsif rising_edge(clk) then
     op1_s <= op1_s;
     op2_s <= op2_s;
    if write_en_s = '1' then
      case addr_s is
      when REG_OP1 =>
        op1_s <= writedata;
      when REG_OP2 =>
        op2_s <= writedata;
      when others =>
      end case;
      end if;
   end if;
end process write_bloc;
```



Outils pour le FPGA

Génération des binaires : **Vivado**

```
source /opt/Xilinx/Vivado/VERSION/settings64.sh
```

Puis, pour lancer l'outil :

```
vivado&
```

Attention : les autres applications peuvent avoir des problèmes à fonctionner dans le terminal utilisé pour `Vivado` (problème de version des bibliothéques).

Génération du `bit.bin` :

un fichier `bif` :

```
all:
{
        proj_dir/objs/proj_name.runs/impl_1/proj_name.bit
}
```

Génération : bootgen −w −image fichier.bif −arch zynq −process_bitstream bin

Ensuite copie dans le répertoire partagé.

Sur la carte :

- copie du fichier bit.bin dans `/lib/firmware`
- flashage : echo "proj_name.bit.bin" > /sys/class/fpga_manager/fpga0/firmware

Communication depuis le terminal : **devmem**

```
devmem 0x43C00000 32 --> read a 32bits @0x43C00000 (reg 0)
devmem 0x43C00004 32 10 --> writes 10 @ 0x43C00004 (reg 1)
```

Communication depuis l'espace-utilisateur

```
#define PAGE_SIZE 4096

#define REG_ID       (0<<2)
#define REG_START    (2<<2)

/* adresse physique du FPGA */
#define FPGA_BASE_ADDR 0x43C00000

[...]
void *ptr_fpga;
/* necessaire pour obtenir un acces a la memoire */
int fd = open("/dev/mem", O_RDWR|O_SYNC);
if (fd < 0) {
  return EXIT_FAILURE;
}

/* obtention d'un pointeur sur la page en memoire virtuelle
 * ou est projete la zone physique
 */
ptr_fpga = mmap(0, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED,
  fd, FPGA_BASE_ADDR);
if (ptr_fpga == MAP_FAILED) {
  return -2;
}

/* lecture */
value = *(unsigned int*)(ptr_fpga + REG_ID);

/* ecriture */
*(unsigned int*)(ptr_fpga + REG_START) = 0x01;
```

Mise en application

1. découverte de la communication : accès en lecture et écriture aux registres
2. évolution pour ajouter un registre ;
3. accès à une RAM pour en lire le contenu :
4. reprise des applications Xenomai ;
5. création du compteur de période.

Utilisation du FPGA (temps-réel matériel) pour qualifier les latences de Xenomai (temps-réel logiciel).

Sujet disponible sur
`http://www.trabucayre.com/enseignement/tp_fpga.pdf`
Codes disponibles sur
`http://www.trabucayre.com/enseignement/tp_fpga_sources.tgz`