Chapitre 5

Modularité en bloc des codes de configuration numérique

1. METHODE DE CONFIGURATION STRUCTUREE DU SYSTEME NUMERIQUE

1.1. Gestion et Traitement des données

Le choix de la plateforme de développement a été orienté à ceux utilisant des microcontrôleurs à cœur standard. Dans ce domaine, les microcontrôleurs tournant autour d'un cœur 8051 d'INTEL, restent les plus répandus et accessibles. Ils ont surtout l'avantage d'être le pionner dans l'architecture *CISC* et dispose actuellement toutes les ressources nécessaires pour tout développement de système embarqué.

Le microcontrôleur AN2131QC de la carte UNIO52

Sur la carte *UNIO52*, le système est piloté par un microcontrôleur *AN2131QC*, qui possède un cœur 8052, avec une capacité de mémoire interne ou *RAM* de 512 octets et de mémoire *ROM* de 8 Ko (au lieu des 128 octets de *RAM* et des 4Ko de *ROM* sur le standard 8051).

Ainsi pour ménager les 512 octets d'espace mémoire interne disponible, les variables mettant en œuvre une quantité et combinaison de registres mémoires assez importante sont placées dans un espace de mémoires alloué extérieurement. Pour le compilateur *RIDE*, cet espace externe est défini par l'attribut *XDATA*.

C'est le cas de la variable *Params*, qui est une concaténation de plusieurs variables et dont le résultat représente un "*drapeau*" définissant l'action demandée.

```
at 0xA0B xdata struct
{
WORD LL;
```

```
WORD UL;
WORD LLL;
WORD Len;
BYTE x1;
BYTE Action;
WORD Offset;
WORD Factor;
WORD Pretime;
WORD Precount;
BYTE x3;
BYTE Flag;
}Params;
```

Ceci en est de même pour le stockage en bloc et traitement des données venant des convertisseurs *ADC* multiplexés, dans la variable *Data* :

```
at 0x1B40 xdata struct
{
short    Adc[8];
} Data;
```

L'utilisation de l'espace de travail externe est surtout incontournable pour le traitement de données spectrométriques, nécessitant un espace de mémoires conséquente pour manipuler les 1024 valeurs de données discrètes d'une chaîne dont la résolution est de 1024 canaux. Pour une résolution de 32bits (*double-mots* ou *DWORD*) par canal, la déclaration de l'espace mémoire externe est la suivante :

```
at 0xA20 xdata DWORD Spec[1024];
```

Les variables *Cnt0*, *Cnt1*, *Stick*, *Start_stop*, *preset_time*, *clear_time* sont aussi placées dans l'espace de mémoire *XDATA RAM*, pour permettre de passer des données afin de contrôler le fonctionnement du code à partir d'un programme informatique sur un microordinateur hôte.

```
at 0x1B00 xdata WORD Cnt0;
at 0x1B02 xdata WORD Cnt1;
at 0x1B04 xdata WORD Stick;
at 0x1AEB xdata WORD start_stop;
at 0x1AED xdata WORD preset_time;
at 0x1AEF xdata WORD clear_time;
```

Une des puissances de ces microcontrôleurs en architecture *CISC* est leurs capacités à exploiter toute combinaison des ressources disponibles. Ainsi on peut étendre les possibilités pour pouvoir utiliser plus d'espace mémoires physiques exploitant toute la capacité d'adressage possible.

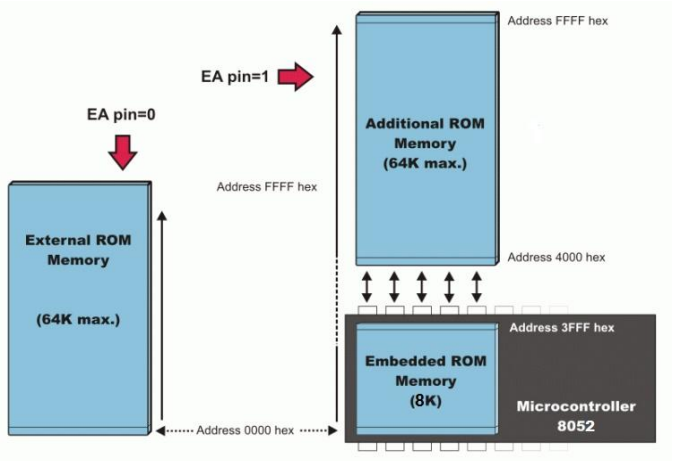

Figure-5. 1 : Espace d'adressage mémoire du microcontroleur AN2131Q

:

Le microcontrôleur *AN2131Q* appartient à la famille *ezUSB,* qui possède dans sa structure interne, les ressources nécessaires à la communication et interface *USB*. Il est totalement compatible avec les outils de développement standards des microcontrôleurs à cœurs *8051* et *8052* : ce qui facilite énormément l'implémentation des codes de programmation.

Dans notre cas, l'outil de développement *RIDE* a été utilisé pour programmer le code système de la carte *UNIO52*, qui est le cœur de la chaîne *MIMCA* [11].

Le composant d'architecture configurable *FPGA* est vu, par le microcontrôleur comme étant un bloc de composant d'Entrée/Sortie : l'accès à chaque élément (registre de mémoire) est effectué par les commandes d'écriture ("*Write*") et de lecture ("*Read*").

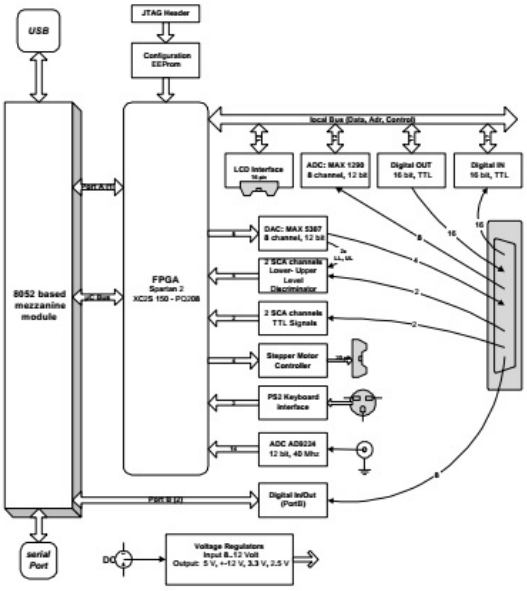

Figure-5. 2 : Schéma bloc des interfaces E/S de la carte UNIO52

Le *FPGA* contrôle les périphériques matériels à travers un bus de 8 et 16 bits. Les données sont transmises vers ces périphériques par un cycle de commande écriture "*WRITE*" du microcontrôleur, et sont lues par un cycle de commande "*READ*".

Le contrôle du convertisseur analogique-numérique *AD9224* de la carte, utilisé pour effectuer la fonction *MCA*, est réalisé en activant (opérations d'écriture et de lecture) les registres correspondants à l'espace mémoire sélectionné par le flag *CS3* : adresses comprises entre *0x40C0* et *0x40CE*.

Tableau-5. 1 : Registres de contrôle des fonctions MCA et leurs adresses

CS3 MCA functions		**Read**	**Write**
0x40C0 .. 0x40C1		16 bit: fast ADC Data signed integer (i16)	16 bit: Lower Level 0..4095 → -2 .. 2 Volt
0x40C2 .. 0x40C3		16 bit: fast ADC Data unsigned integer (u16)	16 bit: Upper Level 0..4095 → -2 .. 2 Volt
0x40C4 .. 0x40C5			16 bit: Lowest Level 0..4095 → -2 .. 2 Volt
0x40C6 .. 0x40C7			16 bit: max Event Len 0..65535→ 0..2700 ms (1/24 MHz)
0x40C8 .. 0x40C9		MCA Peak Maximum 0..4095 → -2 .. 2 Volt	
0x40CA .. 0x40CB		MCA Event Rate	
0x40CC .. 0x40CD		MCA Spec. Memory Address pointer	MCA Spec. Memory Address pointer
0x40CE		MCA Spec. Memory Data	MCA Spec. Memory Data

Dans la programmation du code système, la description matérielle de la carte UNIO52 est contenue dans le fichier source *EzUsb52.h* (constantes et paramètres de configuration des registres de fonctions spéciales du microcontrôleur à cœur 8052), et celle des fonctions nécessaires pour l'appel dynamique des ressources physiques disponibles sur la carte est définie dans la librairie *UnIO52_her.h*.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "..\Mylib\UnIO52_her.h"
...
void main (void)
{
       static WORD mytick, tick;
       unsigned t1s_count;
       unsigned short i,j,k,t;

UnIO52_Init();
timer0_init;
…
```

Le microcontrôleur CY7C68013A de la carte EFM01

Avec la carte *EFM01*, le microcontrôleur *CY7C68013A* qui y est implanté appartient à la famille Cypress *EZ-USB® FX2LP™*, à cœur 8051 et disposant d'un block de mémoire statique plus conséquent de 16Ko, en comparant à celui que possède le microcontrôleur *AN2131Q*.

Le microcontrôleur *EZ-USB* implanté sur la carte *EFM01* est un modèle *CY7C68013A-56LFXC*, ayant un package *QFN* de 56 pins à ports multiplexés.

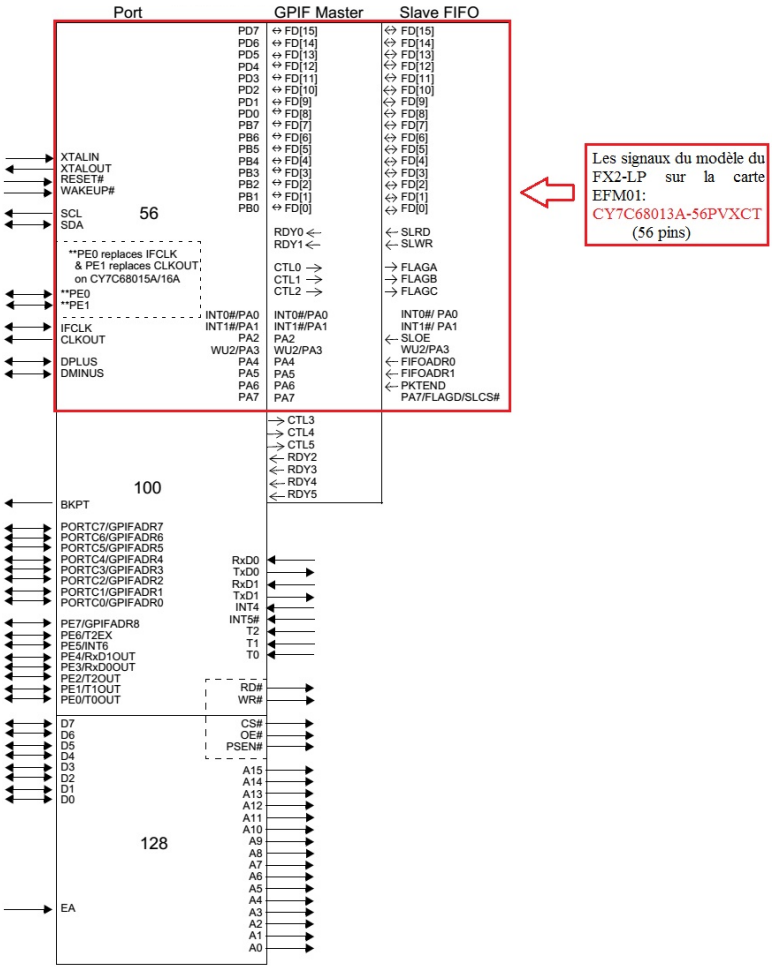

Figure-5. 3 : Caractéristiques physiques du microcontroleur CY7C68013A

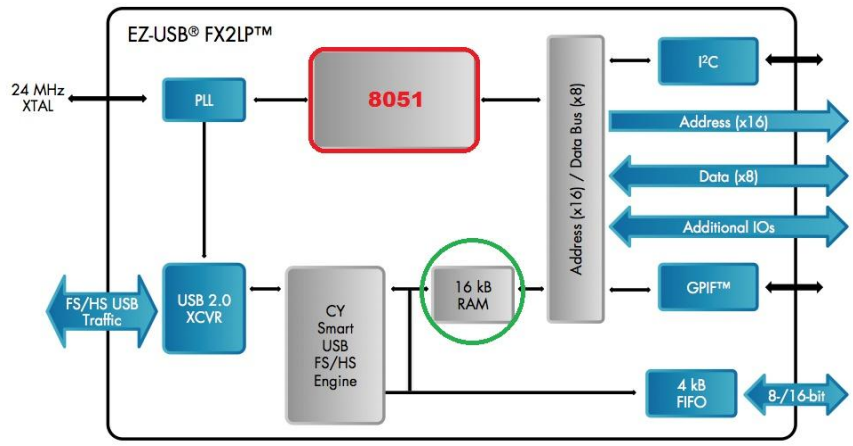

Figure-5. 4 : Schéma de fonctionnement interne du microcontroleur CY7C68013A

Le *CY7C68013A* a surtout la capacité de traiter et transférer les données en masse (paquets allant jusqu'à 512 octets à la fois) et à Haute Vitesse ("High-speed bulk transfer") [63]. Pour que les périphériques d'entrées et de sorties puissent s'y accommoder, la transaction des données est alors effectuée à travers un block de mémoire tampon (*FIFO*) agencé suivant des points terminaux (*EP*).

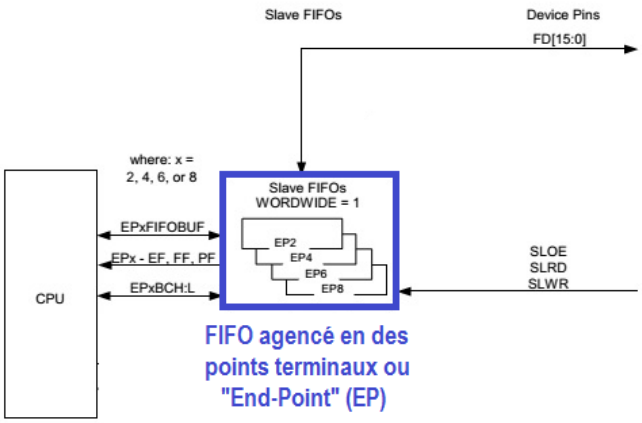

Figure-5. 5 : Agencement des mémoires tampons (FIFO) du microcontroleur CY7C68013A, en points terminaux (End Point)

Du côté de l'ordinateur hôte, la communication s'effectue à travers le processeur d'interface série (*SIE*) qui y est implanté. Ce processeur *SIE* facilite la prise en charge du protocole de communication *USB* et le transfert des paquets de données utilisateur vers les points terminaux (*EP*) appropriés. Chaque point terminal est réalisé avec des registres tampons de 2ko, et est accessible suivant le mécanisme de transfert *FIFO*, contrôlé par un étage maître extérieur générant les commandes de lecture et d'écriture appropriées : "*slave FIFO mode*".

Le mode d'interface est défini suivant le code de configuration écrit sur les bits [1 :0] du registre de fonction spéciale *IFCONFIG*.

Tableau-5. 2 : Bits de configuration du Régistre de Fonction Spéciale IFCONFIG

IFCONFIG			Interface Configuration(Ports, GPIF, slave FIFOs)				E601
b7	b6	b5	b4	b3	b2	b1	b0
IFCLKSRC	3048MHZ	IFCLKOE	IFCLKPOL	ASYNC	GSTATE	IFCFG1	IFCFG0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
1	0	0	0	0	0	0	0

La configuration en mode "*Slave FIFO*", consisterait alors à écrire la valeur hexadécimale 0x03 (ou 00000011 en binaire) sur le registre *IFCONFIG* [63]:

```
void ceFX2FW::InitSlaveFIFO()
{
SetRegister(REG_REVCTL          , 0x00); // 15.5.9
SetRegister(REG_EP2CFG          , 0xa0); // 15.6.2 valid, bulk, 512, 4x
SetRegister(REG_EP6CFG          , 0xe0); // 15.6.2 valid, in, bulk, 512, 4x

SetRegister(REG_IFCONFIG        , 0x03); // 15.5.2 slave FIFO, ext clock

SetRegister(REG_EP2GPIFFLGSEL   , 0x00); // 15.13.8 prog
SetRegister(REG_EP6GPIFFLGSEL   , 0x00); // prog
SetRegister(REG_PINFLAGSAB      , 0x84); // 15.5.3 A=EP2 PF, B=EP2 EF
SetRegister(REG_PINFLAGSCD      , 0x0e); // 15.5.3 C=EP6 FF
SetRegister(REG_EP2FIFOPFH      , 0x80); // 15.6.5
SetRegister(REG_EP2FIFOPFL      , 0x02); // Fixed Level : EP2: (>= 2 Byte,
DECIS=1, PKTSTAT=1)
SetRegister(REG_FIFOPINPOLAR    , 0x00); // default
SetRegister(REG_FIFORESET       , 0x80); // 15.5.4 nak, reset ep2, reset ep6, finish
```

Dans lequel syntaxe, REG_IFCONFIG est une variable contenant l'adresse du registre IFCONFIG, et est définie dans le fichier source ***usbfwif.h*** de l'utilitaire UDKAPI2

```
#define REG_IFCONFIG                                    0x18
```

La fonction SetRegister() fait partie de la classe ceFX2W ci-dessous, définie dans le fichier source ***usbfx2fw.h*** de l'utilitaire UDKAPI2.

```
class ceFX2FW;

/**
 * All devices (EFM01, USBV4F ...) have different coupling
 * between FX2 and peripherals, this abstract base class handles this.
 */
class ceSpecificDevice
{
protected:
/**
 * Represents one bit inside an FX2 register.
 */
class ceFlag
```

```
{
   ushort _usRegister;
   uchar _ucMask;
   bool _bLowActive;
public:
   ceFlag(ushort usRegister, uchar ucMask, bool bLowActive) :
      _usRegister(usRegister),
      _ucMask(ucMask),
      _bLowActive(bLowActive)
   {}
   void Set(ceSpecificDevice *pDevice, bool bEnable);
   bool Get(ceSpecificDevice *pDevice);
};
public:
ceFX2FW *_pDevice;
ceFlag _PROGB;
ceFlag _PWR;
ceFlag _RESET;
ceFlag _INITB;
ceFlag _DONE;
protected:

void SetRegister(ushort usRegister, uchar ucValue);

uchar GetRegister(ushort usRegister);
```

En mode "*Slave FIFO*", les pins d'interface *FIFO* du microcontrôleur sont configurés pour êtres disposés sous contrôle d'un module "Maître" extérieur. Dans notre application, le circuit configurable *XC3S500E* est configuré pour avoir le contrôle de ces lignes d'interface.

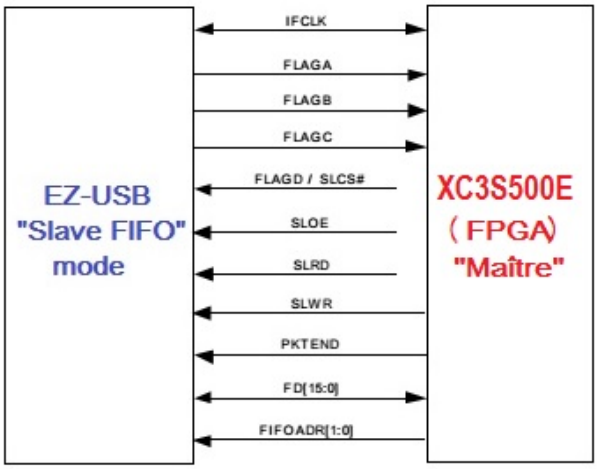

Figure-5. 6 : Configuration en mode "Slave FIFO" du microcontroleur CY7C68013A

L'accès au module *FIFO* est effectué à travers le bus de données *FD* (8 ou 16-bit large), qui est un bus bidirectionnel contrôlé par le signal *SLOE*.

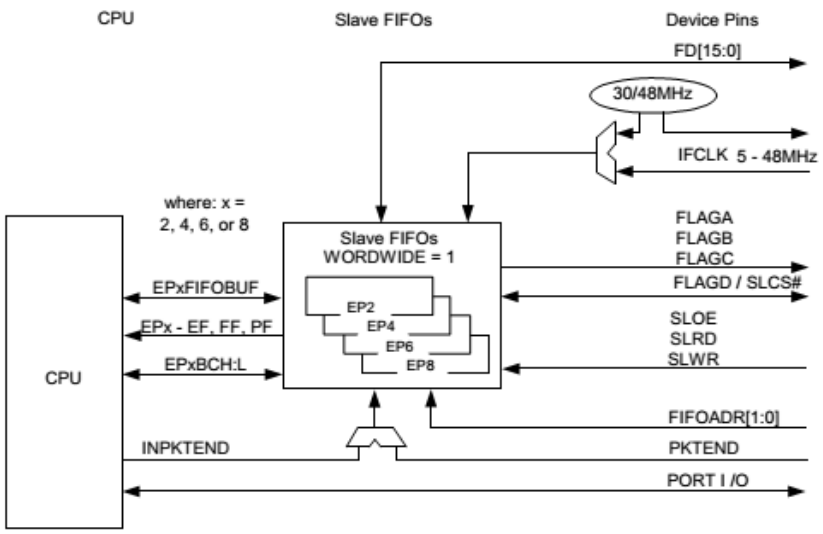

Figure-5. 7 : Les registres de contrôle du bus de données FIFO

Le bus de contrôle *FIFOADR [1:0]* permet de sélectionner le point terminal à brancher sur le bus de données *FD*. Dans notre application on utilise EP2 comme registre de sortie (*OUT, ADR [1:0] = b"00"*) et EP6 pour les données d'entrée (*IN, ADR [1:0] = b"10"*).

Le *FPGA* est alors configuré pour prendre en main le contrôle du *FIFO* et le système de transfert selon le mode "*Slave-FIFO*".

```
entity fx2_slfifo_ctrl is
  generic
  (
    use_xil_dram : boolean := false;
    nr_of_dbgports : positive := 1
  );
  port
  (
    rst_i : in std_logic;

    -- ports connected to cypress fx2 device

    fx2_ifclk_i : in std_logic;
    fx2_flag_n_i : in std_logic_vector(2 downto 0);
    fx2_sloe_n_o : out std_logic;
    fx2_slrd_n_o : out std_logic;
    fx2_slwr_n_o : out std_logic;
    fx2_pktend_n_o : out std_logic;
    fx2_fifoadr_o : out std_logic_vector(1 downto 0);
    fx2_fd_io : inout std_logic_vector(15 downto 0);

…
  );
end fx2_slfifo_ctrl;


architecture RTL of fx2_slfifo_ctrl is
```

```
  signal rst : std_logic := '1';

  signal fx2_ifclk : std_logic := '0';
  signal fx2_flag_n : std_logic_vector(2 downto 0) := (others => '1');
  signal fx2_sloe_n : std_logic := '1';
  signal fx2_slrd_n : std_logic := '1';
  signal fx2_slwr_n : std_logic := '1';
  signal fx2_pktend_n : std_logic := '1';
  signal fx2_fifoadr : std_logic_vector(1 downto 0) := (others => '0');
```

Avant de pouvoir interagir avec le microcontrôleur, le système d'interface *API* devrait être d'abord initialisé avec la fonction *Init()*.

Les fonctions d'appels API sont définies dans le fichier source ***Efm01Board.cpp.***

```
…
CEfm01Board::CEfm01Board(void)
: m_uRealTimerLo(0)
{
m_pDev = 0;
memset(m_nSpectrum,0,1024*sizeof(int));
}

CEfm01Board::~CEfm01Board(void)
{
}

int CEfm01Board::InitializeApi(void)
{
// initialize api
ceDevice::Init();
return 1;
}
```

Le système matériel sera accessible pour toutes opérations de transfert de données en appelant la fonction *Open()*.

```
int CEfm01Board::Open(void)
{
// Enumerate all supported (PCI/USB) devices.
ceDevice::Enumerate(ceDevice::ceDT_ALL);

// If only one device is expected, the following code is enough to access it:
if(0!=ceDevice::GetDeviceCount())
{
   // use first device
   m_pDev = ceDevice::GetDevice(0);

   m_pDev->Open();


   //program (mca.exe) and FPGA bin file (efm01_mca.bin)
   //must be in the same directory

   char szFileName[MAX_PATH];
   ::GetModuleFileName(NULL,szFileName,MAX_PATH);
   char *c = szFileName + strlen(szFileName) - strlen("mca.exe");
   strcpy(c,"efm01_top.bin");
```

```
    FILE *fp =fopen(szFileName,"rb");
    if(fp == NULL)
    {
       char szError[MAX_PATH+32];
       sprintf(szError,"can't find FPGA bin file:\n%s",szFileName);
       AfxMessageBox(szError);
       return 0;
    }
    fclose(fp);

    m_pDev->ProgramFPGAFromBIN(szFileName);

    //m_pDev->ProgramFPGAFromBIN("C:\\Users\\Lecturer\\Documents\\
        MCA\\efm01_mca_part4\\ise\\efm01_top.bin");
    return 1;
}
return 0;
}
```

Tout appel de fonctions devrait être fermé par la fonction ***Close()*** après usage.

```
int CEfm01Board::Close(void)
{
if(m_pDev)
{
   m_pDev->Close();
}
return 0;
}
```

Les classes de fonctions pour accéder au système matériel sont contenues dans le fichier de liaison dynamique *udkapi_vc9_x86.dll*.

1.2. Modulabilité (*IP-Core*; *SOC*)

Pour assurer une meilleure portabilité et faciliter le cycle de développement du projet, les structures standards de base ont été modélisées et intégrées en modules de propriété intellectuelle (*IP-core*).

Les composants configurables de Xilinx (*XC2S150E* ou *XC3S500*) étudiés pour la réalisation du système d'acquisition possèdent des *Cores IP* naturels sous forme matérielle (*Hard IP cores*), tel que les mémoires distribuées en tables ou *LUT*. Cependant les *IP* sous forme de code de programme écrit en langage *HDL* (*Soft IP*) sont aussi utilisés car ils ont l'avantage d'êtres plus souple pour s'adapter aux besoins.

La disponibilité de l'outil de modélisation *MATLAB/SIMULINK*, associé au logiciel propriétaire de *Xilinx*, le "***Système Generator***" nous a permis de bien exploiter la performance et les avantages de l'utilisation des modules mis en core : synthèse des modules de circuits arithmétiques, qui sont propriétés de *Xilinx*, comme des additionneurs et des multiplicateurs de propriété *Xilinx*.

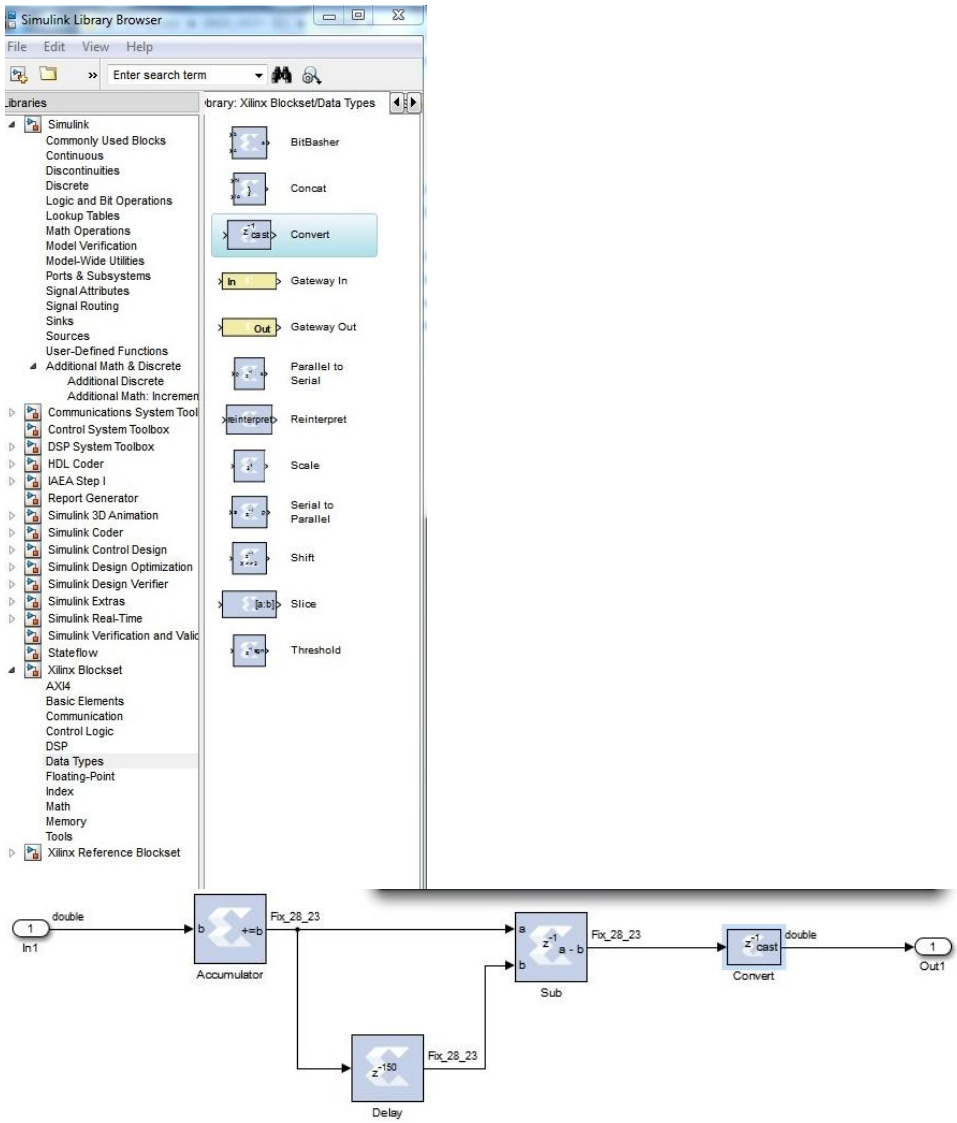

Figure-5. 8 : Modules propriétaires de XILINX pour MATLAB

Une autre technique de mise en œuvre des *Cores IP*, mais à un degré plus poussé qui est l'instanciation des "*Système-On-Chip*" (*SoC*), a été étudiée et implantée pour la migration du système d'acquisition en utilisant le processeur configurable ZYNQ.

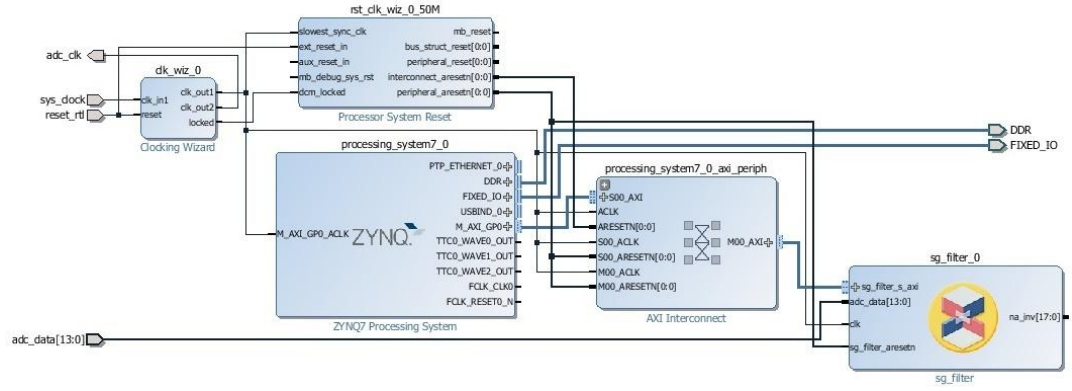

Figure-5. 9 : Modules IP-cores générés par VIVADO

N.B :

- L'acronyme *SOC* désigne une puce intégrant un ensemble de fonctionnalités permettant une utilisation complète sans devoir ajouter d'autres circuits intégrés. Ce qui signifie que tout le système tient sur une seule puce.
- L'intérêt sur l'utilisation des *SOC* se trouve dans le gain de place et de consommation, ainsi que d'une plus grande rapidité pour le traitement des données (pas besoin de passer par des fils conducteurs).

L'adaptabilité du système est aussi renforcé en utilisant un système standard de d'interconnexion (ou bus) pour la liaison entre les *modules on-chip*. Cela faciliterait la réutilisation et l'implantation de l'interface pour d'autres modules.
L'abstraction (numérique) du bus système permet d'améliorer la portabilité du projet car il est plus facile et rapide de modifier la configuration (ajout de périphérique Entrée/Sortie par exemple), par simple programmation du code.

1.3. Structure de Bus ("Whisbone", "AXI")

Les systèmes de bus abstraits (*On-chip*), le "*Wishbone*" et "l'*AXI*" ont été étudié et implanté respectivement sur la plateforme *EFM01* de Cesys et *ZEDBoard*. L'abstraction (binarisation) des bus d'interconnexion est un moyen judicieux pour augmenter le degré de portabilité du système.

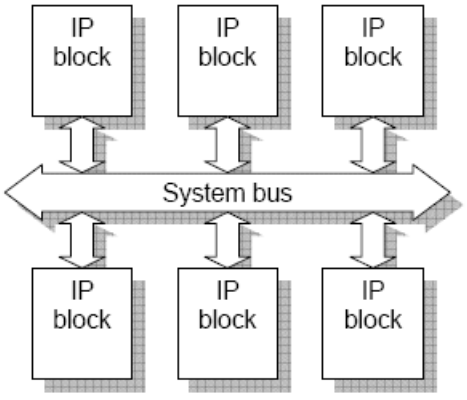

Figure-5. 10 : Schéma de principe d'un bus "on-chip"

Cette technique a été utilisée pour l'implantation en système de *BUS*, des ressources nécessaires pour la transaction des données entre les différents étages de fonctions.