Interface USB caméra sur FPGA4U

Michel Ganguin
Christophe Richon
Julien Ruffin

21 février 2007

1 But du laboratoire

Le but de ce laboratoire est de programmer une interface en VHDL pour une caméra CMOS lm9630 connectée sur la carte FPGA4U et d'utiliser le contrôleur USB High Speed FX2 pour transmettre les images reçues à un ordinateur de bureau.

L'interface caméra doit être capable d'écrire directement dans la mémoire SDRAM de la carte les données qu'elle reçoit. Elle doit donc intégrer une unité DMA. Comme le tout est également une interface programmable, elle devra intégrer une partie esclave avalon pour être contrôlable par le système.

Le système complet devra donc contenir :

- un contrôleur SDRAM
- une interface pour l'USB (FX2)
- une interface programmable pour la caméra avec DMA
- une PLL pour la génération de l'horloge de la caméra
- une interface I^2C pour configurer la caméra
- un bus avalon et un processeur NIOS II pour relier et contrôler le tout.

Les deux premiers composants sont partie intégrante de la FPGA4U.
Le reste est implémenté sur son FPGA par SOPC Builder et Quartus, qui utilisent le contrôleur caméra comme un élément du système qu'ils assemblent. La PLL est un des composants internes du FPGA.

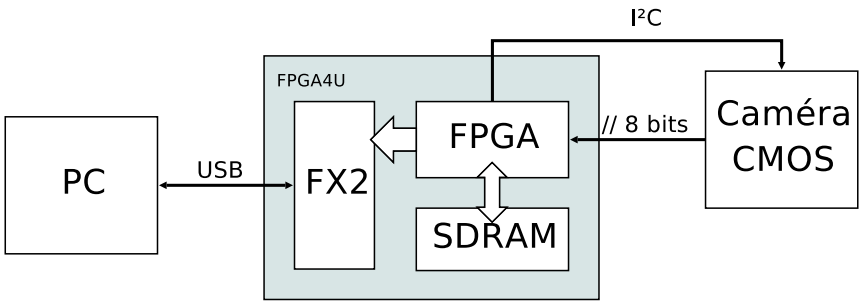

FIG. 1 – Schéma du système

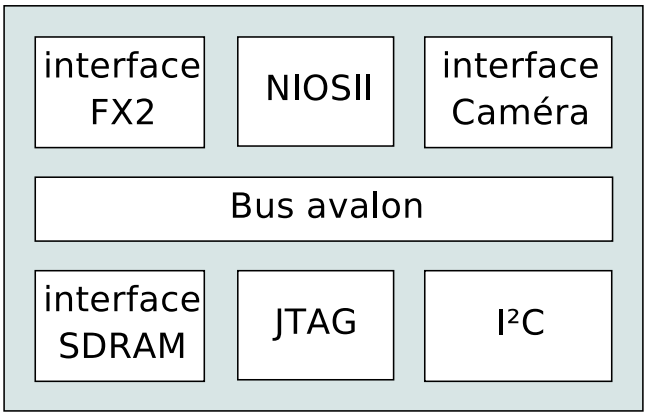

FIG. 2 – Schéma de la partie FPGA

2 Réalisation de l'interface caméra

2.1 Structure du système

Comme décrit au début, le système complet est composé de l'interface caméra, de la SDRAM, de l'interface USB, de l'interface I^2C et du processeur NIOS II pour coordonner acquisitions et transferts, le tout relié par un bus Avalon. Le contrôleur USB et la SDRAM sont des composants physiquement distincts ; à part ces deux-ci et les connecteurs, le système est implémenté sur la FPGA même.

2.2 Structure de l'interface caméra

L'interface caméra est faite de trois sous-composants, chacun avec sa fonction précise. Le maître avalon (AM) assure le rôle de DMA en écrivant les données qui lui sont fournies. L'interface caméra (CI) assure l'acquisition des données de la caméra. Elle possède un tampon interne pour préserver les données lorsque le master avalon est en

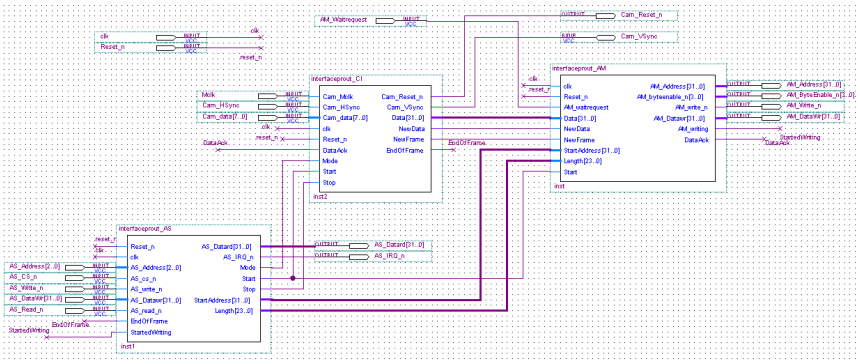

FIG. 3 – Schéma de l'interface caméra

attente d'écriture sur le bus. Enfin, l'esclave avalon (AS) permet la programmation des paramètres des deux autres sous-composants.

Notre interface caméra suit de près la spécification du laboratoire. Ce dernier sert donc de référence pour plus de détails sur le fonctionnement de l'interface.

2.3 Démarche

Deux ou trois choses particulières sont à mentionner à propos de notre démarche de travail. Nous nous sommes divisés le travail à une personne par sous-composant, puis avons convenu à la volée des différents signaux de communication entre eux. Même si cette méthode est peu stricte, ce ne sont pas les quelques erreurs que nous avons fait dans l'établissement de ces conventions qui nous ont ralenti.

L'interface marchait en simulation fonctionnelle sous Modelsim et son intégration dans le système s'est passée sans incident (sauf la prise en main de SOPC Builder). Elle ne marchait pas du tout lors de ses premières tentatives de chargement sur les FPGA4U. Après un grand nombre de corrections, de nouvelles simulations et plusieurs nouveaux essais de chargement, nous sommes aperçu que la simulation était incorrecte à cause d'un signal manquant dans la liste de sensibilité d'un processus. Cette erreur en cachait une autre : une fonction de l'esclave avalon était combinatoire au lieu d'être séquentielle, et l'omission dans la liste de sensibilité la faisait se comporter de manière pseudo-correcte (combinatoire) *en simulation* et pas en réalité.

D'autres problèmes se sont posés, notamment la gestion exacte du *waitrequest* sur le maître avalon, un brin épineuse. Après avoir cherché sans succès à donner la priorité la plus élevée au master de l'interface caméra sur le bus de données, nous avons doté la sous-partie CI de notre interface d'un tampon évitant de perdre des données pendant un blocage du master pour cause de *waitrequest*.

Restait un ultime problème : certaines écritures ne s'effectuaient pas dans la mémoire. Une écriture maître avalon s'effectue en un seul cycle d'horloge, mais ce cycle unique

n'était pas suffisant. Soupçonnant des erreurs de *timing* à cause des avertissements de Quartus disant que certaines contraintes de temps de propagation d'horloge n'étaient pas satisfaites, nous avons d'abord cherché à pousser le *fitter* à ses limites pour optimiser mieux le placement-routage du circuit.

Les résultats étaient satisfaisants, mais ne résolvaient pas le problème. Nous avons alors à titre expérimental changé le nombre de cycles d'horloge écriture de un à deux cycles, et la caméra a fonctionné parfaitement.

Il est à noter, finalement, que le but était d'acquérir des images en continu et que pour ce faire, nous avons utilisé le mode *snapshot* de la caméra et non son mode *vidéo* dédié (la raison se trouve plus bas, dans la partie NIOS). Seul ce premier mode a été testé avec succès. Le mode *vidéo* nécessiterait encore plusieurs corrections.

2.4 Implémentation VHDL

2.4.1 Maître avalon (AM)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-- amen

ENTITY interfaceprout_AM is
    port(

    clk : in std_logic;
    Reset_n : in std_logic;

    AM_Address : out std_logic_vector(31 downto 0);
    AM_byteenable_n : out std_logic_vector(3 downto 0);
    AM_write_n : out std_logic;
    AM_Datawr : out std_logic_vector(31 downto 0);
    AM_waitrequest : in std_logic;

    AM_writing : out std_logic;

    Data : in std_logic_vector(31 downto 0);
    NewData : in std_logic;
    DataAck : out std_logic;
    NewFrame : in std_logic;
    StartAddress : in std_logic_vector(31 downto 0);
    Length : in std_logic_vector(23 downto 0);
    Start: in std_logic
    );
end interfaceprout_AM;

architecture bwaaaah of interfaceprout_AM is

SIGNAL s_base, t_base : std_logic_vector(31 DOWNTO 0);
SIGNAL s_length, t_length : std_logic_vector(23 DOWNTO 0);
TYPE st is (idle, writewait, writing, acked);
SIGNAL state : st;
SIGNAL nftrigger : std_logic;

BEGIN
--Super Richon
--   _()
--  / /--/
--   /\
--  / |
```

```
AM_Address <= t_base;

brouzouf : PROCESS(clk, Reset_n)
BEGIN

    IF(reset_n = '0')
    THEN
        s_length <= X"000000";
        s_base <= X"00000000";
        t_base <= X"00000000";
        t_length <= X"000000";
        AM_writing <= '0';
        AM_write_n <= '1';
        nftrigger <= '0';
        state <= idle;
        -- les transferts se font toujours sur 32 bits
        AM_byteenable_n <= "0000";
    ELSIF(rising_edge(clk))
    THEN

        -- charger les paramètres lors d'un start
        IF(start = '1')
        THEN
            s_base <= StartAddress;
            s_length <= length;
        END IF;

        -- déclencher le trigger ad hoc lors du début d'une image
        IF(newframe = '1')
        THEN
                nftrigger <= '1';
        END IF;

        -- transition de la machine d'états
        CASE state IS
            WHEN idle =>
                -- état inactif, en attente du démarrage
                AM_write_n <= '1';
                DataAck <= '0';
                AM_writing <= '0';
                -- si nouvelle image et nouvelles données de l'
                    image => démarrer
                IF(nftrigger = '1' AND newdata = '1' AND s_length
                    /= X"00000000")
                THEN
                    state <= writing;
                    t_base <= s_base;
                    -- un cycle de moins que la longueur, cause
                    -- décrémentation pas faite dans la dernière
```

```
                écriture
            t_length <= s_length - 4;
            nftrigger <= '0';
        END IF;
    WHEN writing =>
        -- écriture de données sur le bus
        AM_writing <= '1';
        AM_write_n <= '0';
        DataAck <= '0';

        AM_Datawr <= Data; -- on balance la sauce
        state <= writewait;
    WHEN writewait =>
        -- idem, un second cycle, parce que le bus
        -- ne semble pas apprécier l'écriture sur 1 cycle
        -- (douloureusement constaté expérimentalement)
        IF(AM_waitrequest = '0')
        THEN
            DataAck <= '1';
            state <= acked;
        END IF;
    WHEN acked =>
        -- attente de nouvelles données ou de la fin de l'
            image
        AM_write_N <= '1';
        DataAck <= '0';

        IF(t_length = X"00000000")
        THEN
            -- fin de l'image
            AM_Writing <= '0';
            state <= idle;
        ELSIF(nftrigger = '1')
        THEN
            -- nouvelle image, recharger les paramètres
            t_base <= s_base;
            t_length <= s_length;
            state <= writing;
            nftrigger <= '0';
        ELSIF(newdata = '1')
        THEN
            -- nouvelles données, continuer l'écriture
            state <= writing;
            -- incrémentation
            t_base <= t_base + 4;
            t_length <= t_length - 4;
        END IF;

    WHEN OTHERS => -- rien.
END CASE;
```

```
    END IF;
END PROCESS brouzouf;

END bwaaaah; -- bwaaaaaaaaaah
```

2.4.2 Interface caméra (CI)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-- amen

ENTITY interfaceprout_CI is
    port(

    Cam_Reset_n : out std_logic;
    Cam_Mclk : in std_logic;
    Cam_HSync : in std_logic;
    Cam_VSync : inout std_logic;
    Cam_data : in std_logic_vector(7 downto 0);

    clk : in std_logic;
    Reset_n : in std_logic;


    Data : out std_logic_vector(31 downto 0);
    NewData : out std_logic;
    DataAck : in std_logic;
    NewFrame : out std_logic;
    Mode, Start, Stop : in std_logic;
    EndOfFrame: out std_logic

    );

end interfaceprout_CI;

architecture bwaaaah of interfaceprout_CI is
    TYPE tablard IS ARRAY (0 TO 7) OF std_logic_vector(31 downto
        0);
    SIGNAL fifo : tablard;
    SIGNAL ndtrigger : std_logic;
    SIGNAL inptr, outptr : std_logic_vector(2 DOWNTO 0);

    SIGNAL DataBuffer: std_logic_vector(23 downto 0);
    SIGNAL byteEnable: std_logic_vector(1 downto 0);
    SIGNAL sMclkdiv, srMclkdiv,
           MclkEnable, prev_MclkEnable : std_logic;
```

```
    SIGNAL prevHSync, prev2HSync,
           HSyncEnable, HSyncDisable : std_logic;
    SIGNAL prevVSync, prev2VSync,
           VSyncEnable, VSyncDisable : std_logic;
    SIGNAL MyMode: std_logic;
    SIGNAL linecounter: std_logic_vector(6 downto 0);
    TYPE st is (idle, startsnapshot1, startsnapshot2,
                snapshot, acquire, waithsync, waitvsync);
    SIGNAL state : st;

BEGIN
-- Michel

camclock: PROCESS (clk, reset_n)
BEGIN
    IF (Reset_n = '0')
    THEN
        Cam_Reset_n <= '0';
        sMclkdiv <= '0';
        srMclkdiv <= '0';
        prevHsync <= '0';
        prev2Hsync <= '0';
        prevVSync <= '0';
        prev2VSync <= '0';
    ELSIF (rising_edge(clk))
    THEN
        Cam_Reset_n <= '1';

        -- mise à jour des signaux attardés
        sMclkdiv <= Cam_Mclk;
        srMclkdiv <= sMclkdiv;
        prevHSync <= Cam_HSync;
        prev2HSync <= prevHSync;
        prevVSync <= Cam_VSync;
        prev2VSync <= prevVSync;
    END IF;
END PROCESS camclock;

-- détection des flancs montants/descendants
MclkEnable <= sMclkdiv AND NOT(srMclkdiv);
HSyncEnable <= prevHSync AND NOT(prev2HSync);
HSyncDisable <= NOT(prevHSync) AND prev2HSync;
VSyncEnable <=  prevVSync AND NOT(prev2Vsync);
VSyncDisable <= NOT(prevVsync) AND prev2Vsync;

-- sortir du FIFO ce que le master veut
Data <= fifo(conv_integer(outptr));

state_p : PROCESS (clk, reset_n)
BEGIN
```

```
IF (Reset_n = '0')
THEN
    NewData <= '0';
    NewFrame <= '0';
    EndOfFrame <= '0';
    Cam_VSync <= 'Z';
    EndOfFrame <= '0';
    MyMode <= '0';
    DataBuffer <= x"000000";
    byteEnable <= "00";
    linecounter <= "0000000";
    state <= idle;

    inptr <= "000";
    outptr <= "000";
    ndtrigger <= '0';
ELSIF (rising_edge(clk))
THEN
    -- RàZ des impulsions
    NewFrame <= '0';
    NewData <= '0';

    CASE (state) IS
        WHEN idle =>
            -- état inactif, en attente de démarrage
            IF (Mode = '1')
            THEN
                Cam_VSync <= '0';
             ELSIF(Mode = '0')
             THEN
                Cam_VSync <= 'Z';
             END IF;

             IF (Start = '1')
             THEN
                IF(Mode = '1')
                THEN
                    Cam_VSync <= '1';
                    state <= startsnapshot1;
                ELSE
                    state <= waitvsync;
                END IF;
                MyMode <= Mode;
                NewFrame <= '1';
                EndOfFrame <= '0';
                linecounter <= "0000000";
                -- paranoia
                inptr <= "000";
                outptr <= "000";
             END IF;
```

```
WHEN waitvsync =>
    -- attente d'un flanc montant de VSync en mode
       vidéo
    IF(Mymode = '0')
    THEN
        IF(VSyncEnable = '1')
        THEN
            state <= waithsync;
            EndOfFrame <= '0';
            NewFrame <= '1';
        END IF;
    END IF;
WHEN waithsync =>
    -- attente d'un flanc montant de HSync
    IF(MyMode = '1')
    THEN
       Cam_VSync <= '0';
    END IF;

    IF(HSyncEnable = '1')
    THEN
      state <= acquire;
    END IF;

    IF (MyMode = '1' AND linecounter = X"65")
    THEN
        EndOfFrame <= '1'; -- je sai pourkvoi
        state <= idle;
    ELSIF (MyMode = '0' AND VSyncDisable = '1')
    THEN
        EndOfFrame <= '1';
        state <= waitvsync;
    END IF;
WHEN startsnapshot1 =>
    -- délai pour faire durer le VSync
    IF (MclkEnable = '1')
    THEN
                state <= startsnapshot2;
    END IF;
WHEN startsnapshot2 =>
    -- idem que ci-dessus
    IF (MclkEnable = '1')
    THEN
                state <= waithsync;
    END IF;
WHEN acquire =>
    -- acquisition proprement dite, prendre les
       données
    -- caméra pour les envoyer au Master
    EndOfFrame <= '0';
```

```
            IF (MclkEnable = '1')
            THEN
                byteEnable <= byteEnable + "1";
                CASE byteenable IS
                    WHEN "00" =>
                        DataBuffer(7 downto 0) <= Cam_data;
                    WHEN "01" =>
                        DataBuffer(15 downto 8) <= Cam_data;
                    WHEN "10" =>
                        DataBuffer(23 downto 16) <= Cam_data;
                    WHEN "11" =>
                        fifo(conv_integer(inptr)) <= Cam_Data
                           & DataBuffer;
                        inptr <= inptr + "1";
                        byteEnable <= "00";
                    WHEN OTHERS =>
                END CASE;
            END IF;

            IF (HSyncDisable = '1')
            THEN
                IF(MyMode = '1')
                THEN
                  linecounter <= linecounter + "1";
                END IF;
                state <= waithsync;
            END IF;


            IF (MyMode = '0' AND VSyncDisable = '1')
            THEN
                EndOfFrame <= '1';
                state <= waitvsync;
            END IF;
        WHEN OTHERS => NULL;
    END CASE;

    -- incrémenter le outptr quand un DataAck arrive
    -- (s'il était demandé, d'où le trigger)
    IF(DataAck = '1' and ndtrigger = '1')
    THEN
        outptr <= outptr + "1";
        ndtrigger <= '0';
    END IF;

    -- a-t-on de nouvelles données ?
    IF(inptr /= outptr)
    THEN
        IF(ndtrigger = '0')
        THEN
```

```
                -- n'oublions pas que NewData est une impulsion (1
                    clk)
                NewData <= '1';
                ndtrigger <= '1';
            END IF;
        END IF;

    END IF;
    -- fin IF(rising_edge(clk))
END PROCESS state_p;
END bwaaaah; -- bwaaaaaaaaaah
```

2.4.3 Esclave avalon (AS)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-- amen

ENTITY interfaceprout_AS is
    port(
    Reset_n : in std_logic;
    clk : in std_logic;

    AS_Address : in std_logic_vector(2 downto 0);
    AS_cs_n : in std_logic;
    AS_write_n : in std_logic;
    AS_Datawr : in std_logic_vector(31 downto 0);
    AS_read_n : in std_logic;
    AS_Datard : out std_logic_vector(31 downto 0);
    AS_IRQ_n : out std_logic;

    Mode, Start, Stop : out std_logic;
    EndOfFrame, StartedWriting : in std_logic;
    StartAddress : out std_logic_vector(31 downto 0);
    Length : out std_logic_vector(23 downto 0)
    );

end interfaceprout_AS;

architecture bweuah of interfaceprout_AS is
    -- registres internes
    SIGNAL CamAddr : std_logic_vector(31 DOWNTO 0);
    SIGNAL CamLength : std_logic_vector(23 DOWNTO 0);
    SIGNAL CamComm : std_logic_vector(7 DOWNTO 0);
    SIGNAL CamStatus : std_logic_vector(7 DOWNTO 0);
    SIGNAL CamStart : std_logic_vector(7 DOWNTO 0);
```

```
    SIGNAL CamStop : std_logic_vector(7 DOWNTO 0);
    SIGNAL CamSnapshot : std_logic_vector(7 DOWNTO 0);
    -- buffers
    SIGNAL s_mode : std_logic;

    -- signaux intermédiaires "inter-processus"
    SIGNAL reset_irq : std_logic;
    SIGNAL prevSW : std_logic;
    SIGNAL SWDisable : std_logic;
    SIGNAL swtrigger : std_logic;
BEGIN
-- J. Ruffin was here (and failed, multiple times)

startaddress <= CamAddr;
length <= CamLength;

Start <= (CamStart(0) and CamSnapshot(0) and s_mode)
         or (CamStart(0) and not(s_mode));
Stop <= CamStop(0);

s_mode <= CamComm(0);
Mode <= s_mode;

-- envoi conditionnel de la requête d'interruption
AS_IRQ_n <= NOT(CamComm(1) and CamStatus(4));


-- flanc descendant de StartedWriting
SWDisable <= NOT(StartedWriting) and prevSW;

datawrite : PROCESS(clk, reset_n)
BEGIN
    IF (reset_N = '0')
    THEN
        CamAddr <= X"00000000";
        CamLength <= X"003280"; -- 128 * 101;
        CamComm <= X"00";
        CamStart <= X"00";
        CamStop <= X"00";
        CamSnapshot <= X"00";
        reset_irq <= '0';
    ELSIF (rising_edge(clk))
    THEN
        -- RàZ automatiques des registres de commande
        -- CamStart est un peu particulier, il se remet à zéro
        -- quand la conversion démarre effectivement
        -- (cf. mode snapshot qui requiert
        --  CamStart = 1 ET CamSnapshot = 1)
        IF((CamStart(0) = '1' and CamSnapshot(0) = '1'
            and s_mode = '1')
```

```
    or (CamStart(0) = '1' and s_mode = '0'))
THEN
     CamStart <= X"00";
END IF;

CamStop <= X"00";
CamSnapshot <= X"00";

-- RàZ automatique du registre de reset de IRQ
-- (ce qui nous intéresse, c'est une impulsion)
reset_irq <= '0';

-- écritures diverses
IF(AS_write_n = '0' and AS_cs_n = '0')
THEN
     CASE AS_Address IS
          WHEN "000" =>
               CamAddr <= AS_DataWr;
          WHEN "001" =>
               CamLength <= AS_DataWr(23 DOWNTO 0);
          WHEN "010" =>
               CamComm <= AS_DataWr(7 DOWNTO 0);
          WHEN "011" =>
               -- possible, ça ?!?
               -- ouais d'après la spec
               -- on peut remettre le bit 4 à zéro en
                  écrivant 1 dessus...
               -- (sinon pas d'effet)
               -- AS_DataWr(4) | CamStatus(4)  | n. valeur
               -- ------------------------------------------
               --       0      |       0      |    0
               --       0      |       1      |    1
               --       1      |       0      |    0
               --       1      |       1      |    0
               --
               -- => not(AS_DataWr(4) and CamStatus(4))
               -- CamStatus(4) <= NOT(AS_DataWr(4) and
                  CamStatus(4));
               -- impossible sans faire de collisions, il
                  faut passer le reset de ce bit
               -- via un signal supplémentaire...
               IF(CamStatus(4) = '1')
               THEN
                    reset_irq <= '1';
               END IF;
          WHEN "100" =>
               CamStart <= AS_DataWr(7 DOWNTO 0);
          WHEN "101" =>
               CamStop <= AS_DataWr(7 DOWNTO 0);
          WHEN "110" =>
```

```
                    CamSnapshot <= AS_DataWr(7 DOWNTO 0);
                WHEN OTHERS => NULL;
            END CASE;
        END IF;
    END IF;
END PROCESS datawrite;

dataread : PROCESS(clk)
BEGIN
    IF(rising_edge(clk))
    THEN
        IF(AS_read_n = '0' and AS_cs_n = '0')
        THEN
            CASE AS_Address IS
                WHEN "000" =>
                    AS_DataRd <= CamAddr;
                WHEN "001" =>
                    AS_DataRd <= X"00" & CamLength;
                WHEN "010" =>
                    AS_DataRd <= X"000000" & CamComm;
                WHEN "011" =>
                    AS_DataRd <= X"000000" & CamStatus;
                WHEN "100" =>
                    --AS_DataRd <= X"000000" & CamStart;
                    AS_DataRd <= X"00000000";
                WHEN "101" =>
                    --AS_DataRd <= X"000000" & CamStop;
                    AS_DataRd <= X"00000000";
                WHEN "110" =>
                    --AS_DataRd <= X"000000" & CamSnapshot;
                    AS_DataRd <= X"00000000";
                WHEN OTHERS =>
                    AS_DataRd <= X"00000000";
            END CASE;
        END IF;
    END IF;
END PROCESS dataread;

shoop_da_woop : PROCESS(CamStart, CamStop, CamSnapshot, EndOfFrame
    , SWDisable, reset_n, reset_irq, clk)
BEGIN
    IF(reset_n = '0')
    THEN
        CamStatus <= X"00";
        swtrigger <= '0';
        prevSW <= '0';
    ELSE
        -- divers updates du registre de statut
        CamStatus(2) <= StartedWriting;
        CamStatus(1) <= EndOfFrame;
```

```
        IF(rising_edge(clk))
        THEN
                -- mise à jour synchrone des états déterminant
                    SWDisable
                prevSW <= StartedWriting;

                -- mise à zéro de 'run' et du trigger de fin d'
                    écriture de l'image par le Master
                IF((EndOfFrame = '1' and swtrigger = '1') or
                    CamStop(0) = '1')
                THEN
                    CamStatus(0) <= '0';
                    swtrigger <= '0';
                END IF;
        END IF;

        -- activation du trigger lors de la fin de l'écriture d'
            une image par le Master
        IF(SWDisable = '1') THEN
            swtrigger <= '1';
        END IF;

        -- démarrage d'une acquisition
        IF((s_mode = '0' and CamStart(0) = '1')
            or (s_mode = '1' and CamStart(0) = '1' and CamSnapshot
                (0) = '1'))
        THEN
            -- passage de "run" à 1, de l'IRQ à 0
            CamStatus(0) <= '1';
            CamStatus(4) <= '0';
        END IF;

        -- fin d'une acquisition, mettre CamIRQ à 1
        IF(EndOfFrame = '1' and swtrigger = '1')
        THEN
            CamStatus(4) <= '1';
        END IF;

        -- mise à zéro CamIRQ dans CamStatus(4)
        IF(reset_irq = '1')
        THEN
            CamStatus(4) <= '0';
        END IF;
    END IF;
END PROCESS shoop_da_woop;
END bweuah;
```

2.4.4 Intégration dans le système

Notre composant caméra terminé, il fallait encore l'intégrer dans le système fini. Nous l'avons ajouté au projet Quartus final par importation sous SOPC Builder en tant que composant. Tous les signaux d'« interfaçage au monde extérieur » (données, signaux caméra *HSync, VSync*) ont été exportés hors-système en tant qu'entrées-sorties pour être assignés aux *pins* correspondant au connecteur de la caméra sur la carte.

Une petite particularité réside dans le signal *MClk* du composant qui doit prendre l'horloge à 10Mhz venant de la PLL. Une solution possible était d'exporter ce signal vers l'extérieur du système et de l'assigner au signal sortant de la PLL pour la caméra dans le schéma-bloc quartus, mais nous avons préféré définir ce signal comme l'horloge de la partie maître avalon (signal inutile dans notre cas) de notre composant et assigner explicitement l'horloge du maître avalon de l'interface caméra à l'horloge de la PLL dans le système sous SOPC builder.

Ainsi, la transmission de l'horloge à 10Mhz sur notre contrôleur caméra passe par le bus et non pas par l'« extérieur ».

3 Programmation NIOS II

3.1 Démarche

La programmation du NIOS II n'était pas entièrement à faire, puisqu'un programme de base était fourni. Il nous a néanmoins fallu le modifier pour qu'il utilise notre interface correctement.

Ce programme utilise du *double buffering* pour éviter que l'acquisition récrive sur l'image qu'il est en train de transférer par l'USB. Cela implique de pouvoir changer l'adresse d'écriture dans l'interface programmable, ce qui ne peut se faire qu'au début de l'acquisition (avant l'envoi d'un *start*) par convention dans notre composant.

C'est pourquoi nous utilisons le mode *snapshot* et redémarrons une acquisition à chaque fois que nous voulons une nouvelle image : il est alors possible de donner à l'interface l'adresse du tampon que le transfert USB n'est *pas* en train d'utiliser avant de démarrer l'acquisition. Ceci est impossible en mode *vidéo*, puisque l'acquisition n'est démarrée qu'une fois avec et que l'écriture se fait en continu sur la même zone mémoire, comme les paramètres de l'acquisition (emplacement en mémoire et taille) sont chargés et fixés au démarrage.

3.2 Programme C

```
/*-------------------------------------------------------------------------
 * lm9630_if.c - LM9630 CMOS 2D Camera interface
 *-------------------------------------------------------------------------*/
```

```c
#include <stdio.h>
#include "system.h"
#include "i2c.h"
#include "fx2.h"
#include "lm9630_if.h"
#include <sys/alt_irq.h>

/* wait some microseconds */
#define WAIT { int i; for (i=0; i<100; ) i++; }

/* wait for end of transfer */
#define WAIT_FOR_EOT(io,t) { while ((t=io->np_i2c_status) & I2C_TIP) WAIT;
    }

/* espace memoire pour la capture d'images */
char image1[ 128 * 101 ];
//char *image1;
char image2[ 128 * 101 ];
//char *image2;

/*
 * Set register index (no data write)
 */
int i2c_write_index( np_i2c *io ,
                     unsigned char index )
{
  unsigned char t;

  /* Attend la fin du transfert precedent */
  WAIT_FOR_EOT(io, t);

  /* ecriture de l'adresse du peripherique + R/W bit = 0 ( ecriture ) */
  io->np_i2c_data = ( LM9630_I2C_ADDR & 0xFE );
  io->np_i2c_ctrl = ( I2C_START | I2C_WRITE );

  /* Attend la fin du transfert */
  WAIT_FOR_EOT(io, t);

  /* le périphérique repond ? */
  if ( t & I2C_LRA )
    return -CAMERA2D_ENODEV;

  /* Ecriture de l'adresse du registre */
  io->np_i2c_data = ( index );
  io->np_i2c_ctrl = ( I2C_WRITE );

  /* Attend la fin du transfert */
  WAIT_FOR_EOT(io, t);

  /* Mauvaise quittance */
  if ( t & I2C_LRA )
    return -CAMERA2D_EBADACK;
```

```
  /* Quittance du fanion d'interruption
   * du controleur I2C */
  io->np_i2c_data = 0;

  return 0;
}

/*
 * Ecriture d'une valeur dans un registre
 */
int i2c_single_write( np_i2c *io ,
                      unsigned char index ,
                        unsigned char value )
{
  int rc;
  unsigned char t;

  if ((rc=i2c_write_index( io , index & 0x7F )) < 0)
    return rc;

  /* Ecriture de la donnee */
  io->np_i2c_data = ( value );
  io->np_i2c_ctrl = ( I2C_WRITE | I2C_STOP );

  /* Attend la fin du transfert */
  WAIT_FOR_EOT(io, t);

  /* Mauvaise quittance ? */
  if (t & I2C_LRA)
    return -CAMERA2D_EBADACK;

  /* Quittancement du fanion d'interruption */
  io->np_i2c_data = 0;

  return 0;
}

/*
 * Lecture d'une valeur d'un registre
 */
int i2c_single_read( np_i2c *io ,
                       unsigned char index ,
                       unsigned char *value )
{
  int rc;
  unsigned char t;

  /* set index */
  if ((rc=i2c_write_index( io , index & 0x7F )) < 0)
    return rc;

  /* Repete la sequence I2C Start (RepStart) pour
   * effectuer la lecture
   * Ecriture de l'adresse + bit R/W = 1 (lecture) */
```

```
  io->np_i2c_data = ( LM9630_I2C_ADDR | 0x01 );
  io->np_i2c_ctrl = ( I2C_WRITE | I2C_START );

  /* Attend la fin du transfert */
  WAIT_FOR_EOT(io, t);

  /* le peripherique repond ? */
  if ( t & I2C_LRA )
    return -CAMERA2D_ENODEV;

  t = io->np_i2c_data;
  io->np_i2c_ctrl = ( I2C_READ | I2C_STOP | I2C_NO_ACK);

  /* Attend la fin du transfert */
  WAIT_FOR_EOT(io, t);

  /* Lecture de la donnee recue */
  (*value) = io->np_i2c_data;

  return 0;
}

/* Initialise le module caméra LM9630
 */
void lm9630_init( np_i2c* i2c)
{
  /* Configuration de la caméra via I2C */
  i2c_normal_mode(i2c);
  i2c_single_write( i2c , LM9630_REV_ADDR ,     LM9630_REV_DATA );
  i2c_single_write( i2c , LM9630_MCFG_ADDR ,    LM9630_MCFG_DATA );
  i2c_single_write( i2c , LM9630_VGAIN_ADDR ,   LM9630_VGAIN_DATA );
  i2c_single_write( i2c , LM9630_ITIMEL_ADDR ,  LM9630_ITIMEL_DATA );
  i2c_single_write( i2c , LM9630_IDLE_ADDR ,    LM9630_IDLE_DATA );
  i2c_single_write( i2c , LM9630_ITIMEH_ADDR ,  LM9630_ITIMEH_DATA );
  i2c_single_write( i2c , LM9630_POWSET_ADDR ,  LM9630_POWSET_DATA );
  i2c_single_write( i2c , LM9630_OFFSET_ADDR ,  LM9630_OFFSET_DATA );

}

/* Exemple de transfert */

int main(){
  np_i2c* i2c = ( np_i2c * ) I2C_BASE ;
  fx2_struct* fx2 = (fx2_struct*) FX2_FIFO_BASE;

  unsigned int i, j;
  int regModified = 1;

  i2c_normal_mode( i2c );

  printf("LM9630-sw\n");
  printf("Initialisation...");
  //lm9630_init( i2c, lm9630 );
  lm9630_init( i2c );
```

```
printf("ok\n");

// image initialization
// ... TODO ...
// ah bon ?
//init_image();
// BEGIN vaudou
// enregistrer l'image de base
volatile unsigned int *camera = INTERFACEPROUT_0_BASE;
camera[0] = image1;
// commencer l'acquisition en mode snapshot
camera[2] = 1; // mode snapshot sans interruptions
//printf("%x %d\n", camera[3], camera[7]);
camera[4] = 1; // start
camera[6] = 1; // snapshot
//printf("%x %d\n", camera[3], camera[7]);
// END vaudou

char* imgSrc = (char*)image1;
char* imgDest = (char*)image2;

int itime = 1023;
//int avg = 0;

//capture
for (;;) {
  // alternating for double buffer
  char* tmp = imgSrc;
  imgSrc = imgDest;
  imgDest = tmp;

  // update the registers before aquisition
  while(fx2->status & FX2_BUSY);
  // set to endpoint 6
  fx2->ep_address = FX2_EP6;
  while(fx2->ep_address & FX2_EMPTY_N){
    fx2->control = FX2_READ;
    while(fx2->status & FX2_BUSY);
    int address = fx2->data;
    //check that we have one more data
    if(fx2->ep_address & FX2_EMPTY_N){
      fx2->control = FX2_READ;
      //check if address is valid
      if((address&0xFFFF)<0x9){
        while(fx2->status & FX2_BUSY);
        //write in the register
        int datareg = fx2->data;
        i2c_single_write(i2c, address, 0xFF&datareg);
        regModified = 1;
        if (address == LM9630_ITIMEH_ADDR)
          itime = (itime&0xFF) + ((datareg&0x07) << 8 );
        if (address == LM9630_ITIMEL_ADDR)
          itime = (itime&0x700) + (datareg&0xFF);
      }
```

```
    }
  }
  i2c_single_write( i2c , LM9630_ITIMEH_ADDR , (itime&0x700)>>8 );
  i2c_single_write( i2c , LM9630_ITIMEL_ADDR , itime&0xFF );
  //printf("Itime: %x\n", itime&0x7FF);

  //printf("%x\n", camera[3]);
  // Wait for the previous aquisition
  // ... TODO ...
  // BEGIN vaudou
  // polling

  // while RegStatus(4) = 0, attendre...
  //printf("attente de nouvelle image\n");
  //while((*(ptr+0xC) && 0x10) == 0);

  //printf("registre 0 de l'interface : %x\n", camera[0]);
  //printf("pointeur où écrire : %x\n", imgDest);
  do {
   //WAIT;
   //printf("%x %d\n", camera[3], camera[7]);
  } while( (camera[3] & 0x10) == 0 );


  //printf("nouvelle image\n");

  //printf("camera[3] = %x\n", camera[3]);
  //printf("quittance\n");
  // quittance
  camera[3] |= 0x10;

  // aquisition
  // ... TODO ...
  // BEGIN vaudou

  // commencer l'acquisition en mode snapshot

  // remplir l'image de noir
  /*int size = 101*128;
  for(i=0;i<size;i++){
    imgSrc[i] = 0;
  }*/

  // buffer dans lequel écrire
  camera[0] = imgSrc;
  // démarrage
  camera[4] = 1; // start
  camera[6] = 1; // snapshot

  // END vaudou

  // During the aquisition we send the previous image
  while(fx2->status & FX2_BUSY);
  // set to endpoint 8
```

```
    fx2->ep_address = FX2_EP8;
    // send the data via USB
    int offset = 0;
    short* image_short = (short*) imgDest;
    for(i=0; i<101; i++){
      // First 2 byte will contain line number
      while(fx2->status & FX2_BUSY);
      while(! fx2->status & FX2_FULL_N);
      // Now we send the data
      fx2->data = i;
      fx2->control = FX2_WRITE;
      for(j=offset;j<offset+63;j++){
        while(fx2->status & FX2_BUSY);
        fx2->data = image_short[j];
        fx2->control = FX2_WRITE;
      }
      // We send a pktend with last data
      while(fx2->status & FX2_BUSY);
      fx2->data = image_short[offset+63];
      fx2->control = FX2_PKTEND | FX2_WRITE;
      offset+=64;
    }

    // if the registers where modified we send the new value to the GUI
    if(regModified){
      regModified = 0;
      char data;
      int address=0;
      for (address=0; address<9; address++) {
        while (fx2->status & FX2_BUSY);
        fx2->data = 0x8000 | address;
        fx2->control = FX2_WRITE;
        i2c_single_read( i2c , address , &data );
        while (fx2->status & FX2_BUSY);
        fx2->data = 0xFF & data;
        if(address==8)
          fx2->control = FX2_PKTEND | FX2_WRITE;
        else
          fx2->control = FX2_WRITE;
      }
    }
  }
  return 0;
}
```

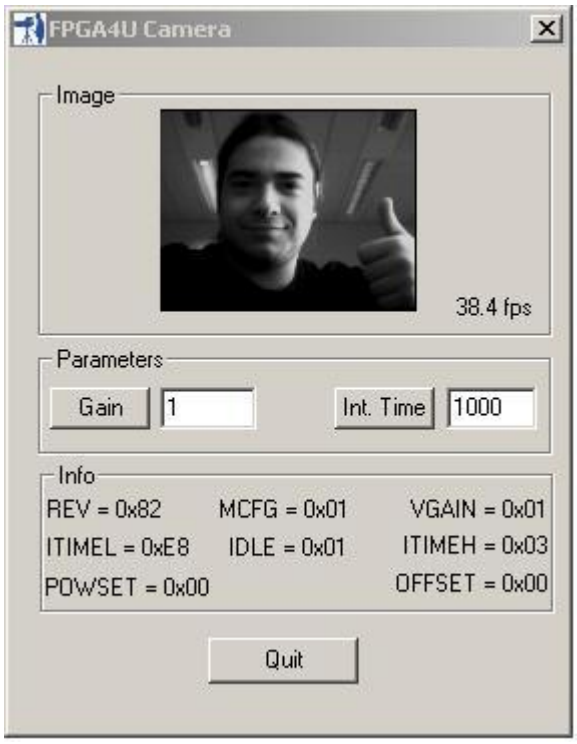

FIG. 4 – La victoire est nôtre.

4 Résultats

Nous arrivons à avoir l'image acquise par la caméra en utilisant le programme conçu à cet effet, exécuté sur le PC sur lequel est branché la FPGA4U par USB. Notre interface caméra marche et est correctement intégrée au système complet. Objectif accompli !