Compte rendu
Contrôleur VGA

16/12/2025

Traitement/Interfaces systèmes embarqués

Adrien BONNAND

POLYTECH
DIJON

Table des matières

Table des figures

Fonctionnement chaîne vidéo

Architecture

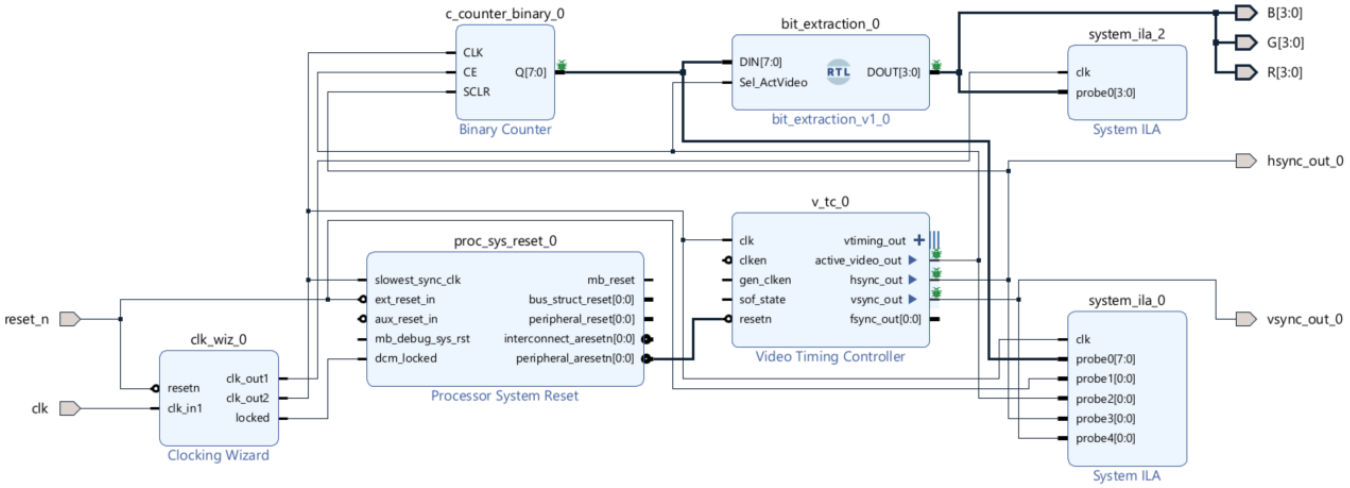

Figure 1 : Architecture chaîne vidéo

Premièrement, une PLL fournit une horloge de 25,2 Mhz. Cette horloge est transmise à la fois à une IP Xilinx permettant de gérer les timings de synchronisation pour l'affichage de données sur un écran. Mais aussi, elle est connectée à un compteur binaire qui permet de changer la valeur des pixels à afficher. Celui-ci possède une entrée *CE* qui autorise l'incrémentation du compteur, uniquement lors de l'écriture d'une ligne de l'image (hors périodes de blank), ainsi qu'un signal *SCLR*, actif quand *Hsync* = 1, afin de réinitialiser le compteur en début de ligne.

La sortie de ce compteur est connectée à un module développé personnellement en VHDL, afin de ne garder que les MSB du compteur. Cela permet le changement de couleur des pixels toutes les 16 incrémentations, et donc l'écriture de bandes verticales de pixels. Le signal *sel_ActVideo* met la valeur de sortie à 0 lorsqu'il y a une écriture dans la partie blank.

Enfin, un debugger hardware (ILA) est utilisé pour enregistrer l'état des signaux importants lors du test sur cible, en s'appuyant sur des BRAMs pour le stockage.

Le fait de générer un motif vidéo (test pattern) est pertinent car cela permet de valider rapidement et simplement le bon fonctionnement d'un système d'affichage. Un motif correct valide la conformité temporelle des signaux de synchronisation (hsync/vsync), du signal active_video ainsi que de la fréquence de changement des pixels.

Sur certains écrans, des bandes noires peuvent apparaître à cause d'une désynchronisation entre les pixels et les signaux de synchronisation (*hsync/vsync*). Ce décalage est souvent dû au traitement interne vieillissant de l'écran, qui n'aligne plus parfaitement les signaux entrants.
Une solution consiste à ajouter un léger délai soit sur les pixels, soit sur les signaux *hsync* et *vsync*, afin de réaligner correctement les informations vidéo. Cela peut être réalisé à l'aide de bascules, d'IP Xilinx comme *IOBDELAY* ou *DCM/PLL* ou tout simplement via les réglages de l'IP *Video Timing Controller* (en décalant le *Sync Start* par exemple).

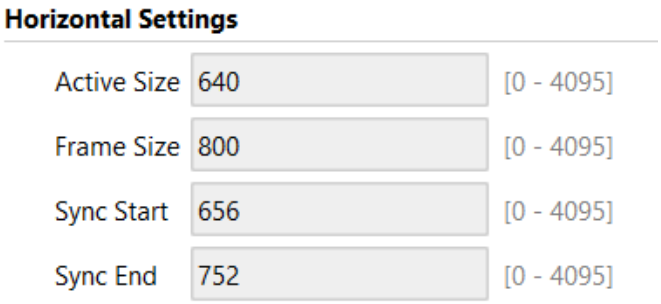

Figure 2 : Réglages IP Video Timing Controller

Le fait de mettre les pixels à zéro quand ActVideo est égale à zéro permet d'éviter l'écriture de pixels parasites dans les zones de blanking. Cela garantit que seules les parties visibles de l'image reçoivent des valeurs de pixels correctes, stabilise l'affichage et évite l'apparition de pixels indésirables.

En fonctionnement

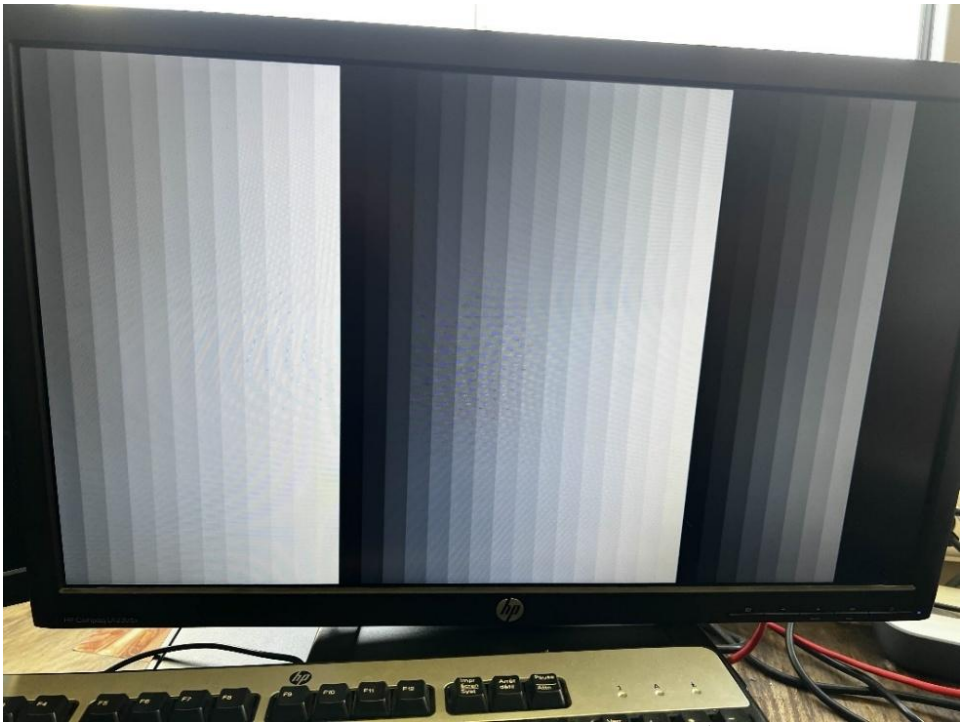

Figure 3 : Fonctionnement de la chaîne vidéo

Simulation (testbench)

Ecriture de l'entièreté d'une image

Figure 4 : Simulation – écriture d'une image complète

Cette simulation, observée d'un point de vue global, montre simplement l'écriture complète d'une image. En effet, les valeurs des pixels évoluent durant toute la phase d'écriture de l'image. Le signal *Hsync* passe régulièrement à 1 pour indiquer l'écriture de chaque ligne, tandis qu'un pulse de *Vsync* en fin de trame signale la fin de l'écriture de l'image.

À la fin du testbench, on observe un « trou » dans le signal *active_video_out*, accompagné de pixels à 0 : cela correspond à la période de « blanking », durant laquelle aucune donnée vidéo n'est transmise.

Début de l'écriture de l'image

Tout d'abord, il est intéressant de voir le début de l'écriture de l'image.

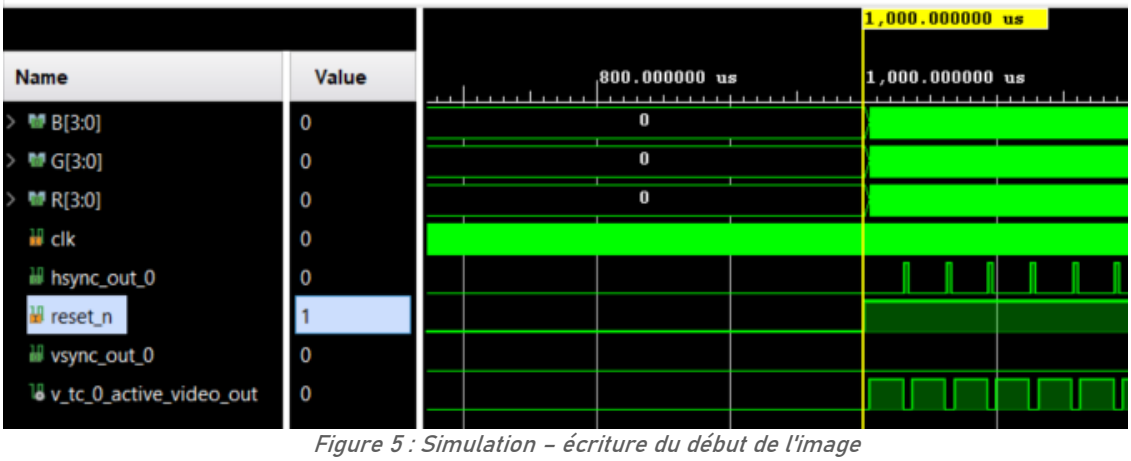

Figure 5 : Simulation – écriture du début de l'image

Dès que le signal *reset* passe à 1 (état inactif), l'écriture de l'image commence. Les pulses de *Hsync* indiquent l'écriture de chaque ligne. De plus, les passages du signal *video_active_out* à 1 montrent que seuls les pixels utiles de l'image sont écrits, excluant ainsi la zone de blanking.

Agrandissement de la fenêtre :

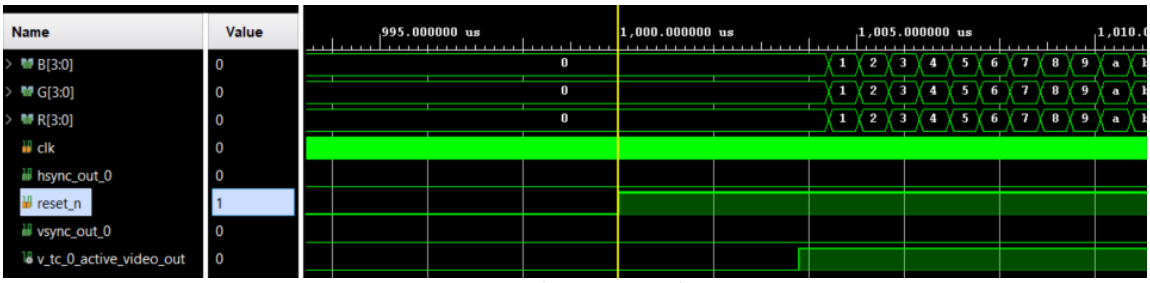

Figure 6 : Simulation – écriture du début de l'image (Zoom)

Le compteur s'incrémente correctement, de 1 en 1, durant l'écriture de la première ligne.

Capture pendant l'écriture de l'image

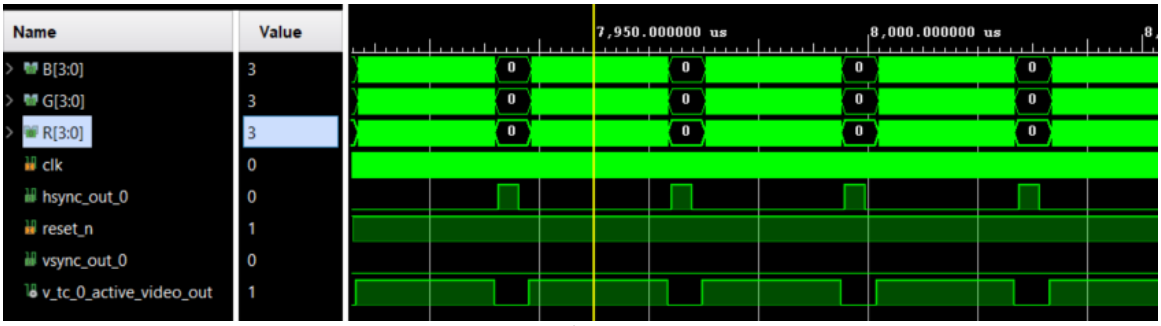

Figure 7 : Simulation – écriture d'un endroit de l'image

Sur cette capture, il est possible de voir plusieurs écritures de lignes, grâce à chaque *Hsync* à 1.

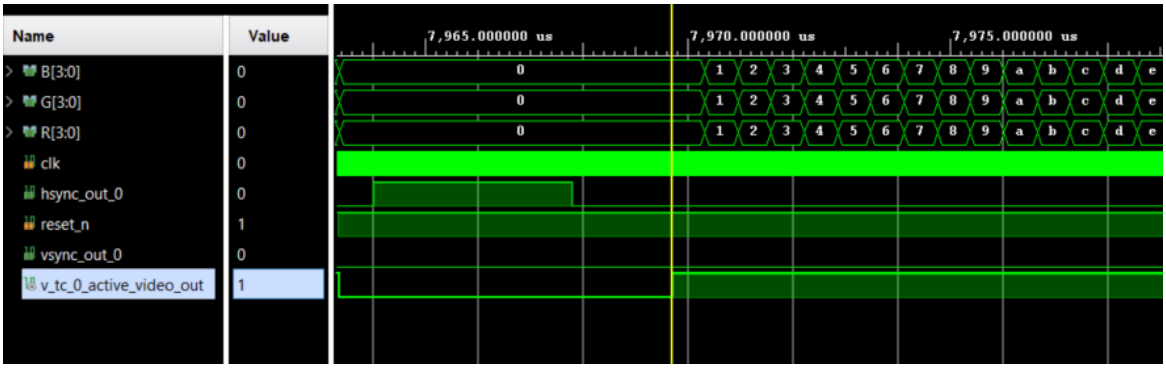

Figure 8 : Simulation – écriture d'un endroit de l'image (Zoom)

L'agrandissement de la fenêtre permet de voir l'incrémentation des pixels. Le retour du compteur à 0 indique que l'on se trouve au début d'une nouvelle ligne d'écriture et montre qu'il est correctement réinitialisé à chaque ligne.

Capture de la fin de l'écriture de l'image

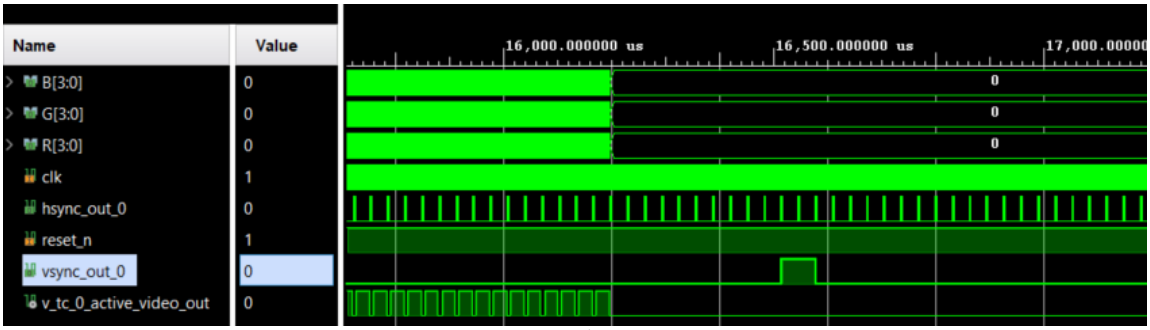

Figure 9 : Simulation – écriture de la fin de l'image

Le compteur reste à 0 pendant plus de 500 µs, ce qui montre que l'écriture de l'image est terminée. Quelques pulses de *Hsync* continuent à apparaître pour générer les lignes vides de la période de « blanking », jusqu'au passage de *Vsync* à 1, signalant la fin de l'image. De plus, *video_active_out* reste à 0 pendant cette même durée, confirmant la présence de la phase de « blanking ».

Débuggeur Hardware (ILA)

Début de l'écriture de l'image

Tout d'abord, il est intéressant de voir le début de l'écriture de l'image.

Trigger utilisé :

```
state wait_vsync:
if (VGA_source_bd_i/system_ila_0/U0/probe3_1 == 1'b1) then
#test on v_tc_0_vsync_out
goto wait_active_video;
else
goto wait_vsync;
endif

state wait_active_video:
if (VGA_source_bd_i/system_ila_0/U0/probe2_1 == 1'bR) then
#test on v_tc_0_hsync_out
reset_counter $counter0;
goto count_LVAL;
else
goto wait_active_video;
endif

state count_LVAL:
if ($counter0 != 16'h0100) then
increment_counter $counter0;
goto count_LVAL;
else
goto start_trigger;
endif

state start_trigger:
trigger;
```

Cette machine à état attend un *Vsync* (fin de l'image) puis un *active_video* à 1 (début de la nouvelle image) et enfin quelques coup d'horloge avant de lancer le trigger.

Capture des signaux internes :

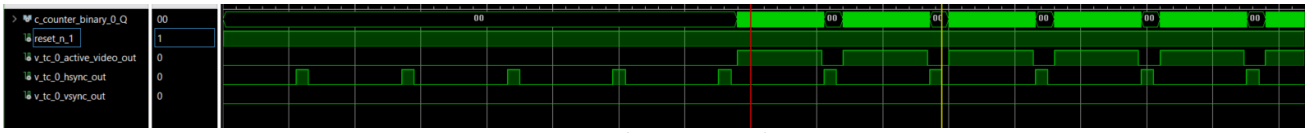

Figure 10 : ILA - écriture du début de l'image

Sur cette capture, les pixels sont correctement à 0 pendant la période de « blanking » et reprennent leur incrémentation dès le début de l'écriture de la nouvelle image.

Capture pendant l'écriture de l'image

Trigger utilisé :

Front montant du signal *active_video_out*

Capture des signaux internes :

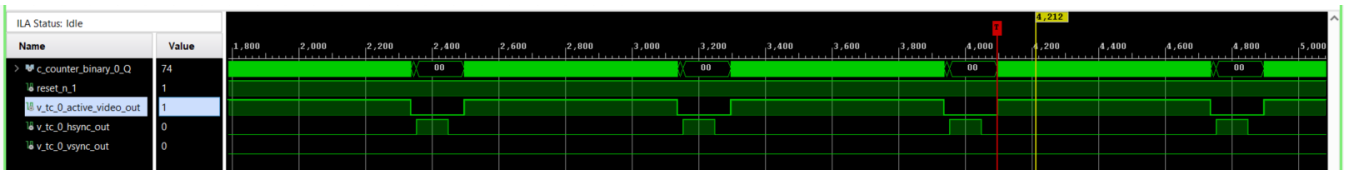

Figure 11 : ILA - écriture d'un endroit de l'image

Sur cette capture, il est possible de voir plusieurs écritures de lignes, grâce à chaque *Hsync* à 1 et la mise à l'état bas du signal *active_video_out*.

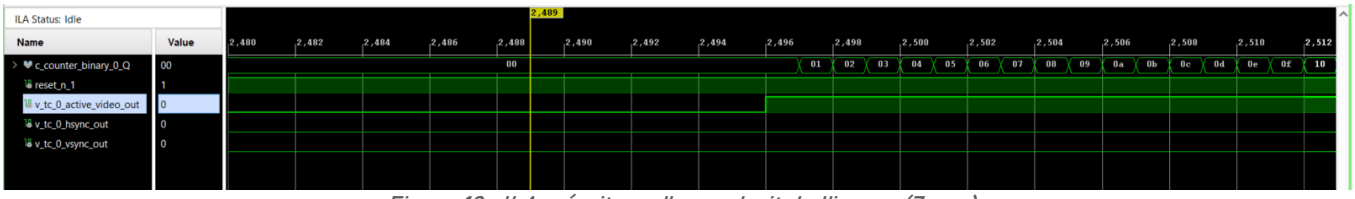

Figure 12 : ILA - écriture d'un endroit de l'image (Zoom)

L'agrandissement de la fenêtre permet d'observer l'incrémentation des pixels, qui recommence à 0 à chaque nouvelle ligne, confirmant que le compteur est correctement réinitialisé.

Capture de la fin de l'écriture de l'image

Trigger utilisé :

```
state wait_vsync:
if (VGA_source_bd_i/system_ila_0/U0/probe3_1 == 1'b1) then
#test on v_tc_0_vsync_out
goto wait_hsync;
else
goto wait_vsync;
endif

state wait_hsync:
if (VGA_source_bd_i/system_ila_0/U0/probe4_1 == 1'bR) then
#test on v_tc_0_hsync_out
goto count_LVAL;
else
goto wait_hsync;
endif

state count_LVAL:
if ($counter0 != 16'h0100) then
increment_counter $counter0;
goto count_LVAL;
else
goto start_trigger;
endif

state start_trigger:
trigger;
```

Cette machine à état attend un *Vsync* (fin de l'image) puis un *Hsync* et enfin quelques coup d'horloge avant de lancer le trigger

Capture des signaux internes :

Figure 13 : ILA – fin de l'écriture d'une image

Le compteur reste à 0, indiquant que l'écriture de l'image est terminée. Quelques pulses de Hsync apparaissent encore pour générer les lignes vides de la zone de « blanking », jusqu'au passage de Vsync à 1, signalant la fin de l'image.

Capture de l'écriture d'une certaine ligne de l'image

Trigger utilisé :

```
state wait_vsync:
if (VGA_source_bd_i/system_ila_0/U0/probe3_1 == 1'b1) then
#test on v_tc_0_vsync_out
reset_counter $counter0;
goto count_active_video;
else
goto wait_vsync;
endif


#state wait_active_video:
#if (VGA_source_bd_i/system_ila_0/U0/probe2_1 == 1'bR) then
##test on v_tc_0_hsync_out
#reset_counter $counter0;
#goto count_active_video;
#else
#goto wait_active_video;
#endif


state count_active_video:
if (VGA_source_bd_i/system_ila_0/U0/probe2_1 == 1'bR) then
increment_counter $counter0;
goto count_LVAL;
else
goto count_active_video;
endif

state count_LVAL:
if ($counter0 != 16'u2) then
goto count_active_video;
else
goto start_trigger;
endif

state start_trigger:
trigger;
```

Cette machine à état attend un *Vsync* (fin de l'image) puis compte un nombre de front montant de *video_active_out* (4) qui représente, à chaque fois, l'écriture d'une nouvelle ligne.

Capture des signaux internes :

Capture des signaux quand le compteur est égal à 2 :

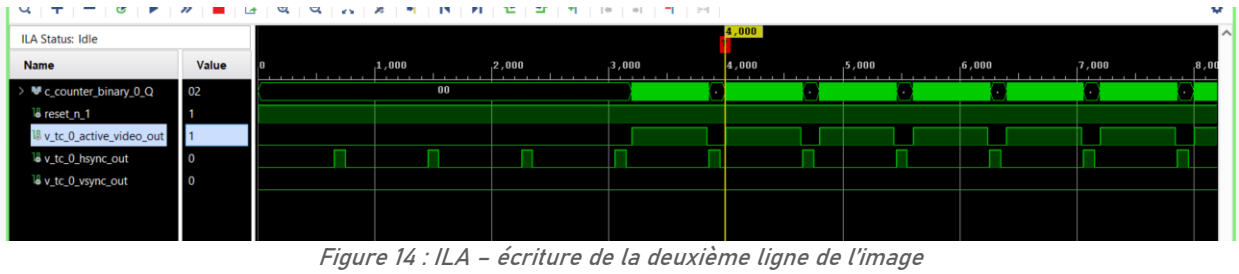

Figure 14 : ILA – écriture de la deuxième ligne de l'image

Capture des signaux quand le compteur est égal à 15 :

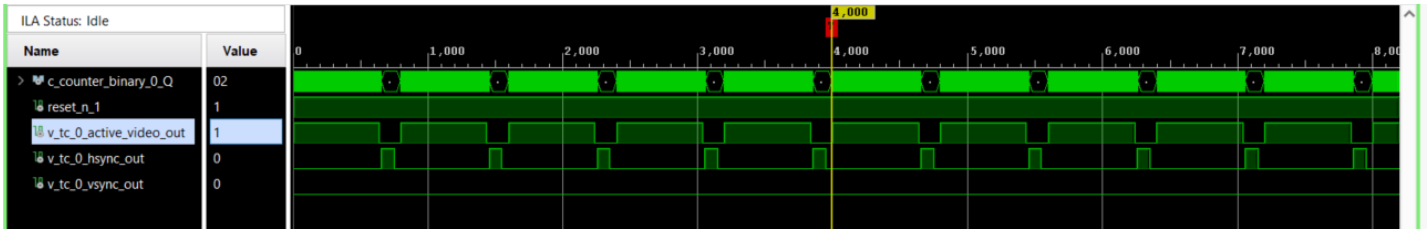

Figure 15 : ILA – écriture de la quinzième ligne de l'image

Conclusion simulation et debugger Hardware

D'après l'ensemble des captures précédentes, il n'y a pas de régression entre la simulation, qui est parfaite, et le test sur cible, où des phénomènes extérieurs peuvent intervenir. Cependant, la simulation reste plus facile à analyser, car elle ne nécessite pas la mise en place de triggers et permet d'observer l'état des signaux sur toute la durée souhaitée. En comparaison, l'ILA est limitée par la taille des BRAM, ce qui restreint la capture dans le temps.

Interfaçage AXI Stream

Architecture avec l'interfaçage AXI STREAM

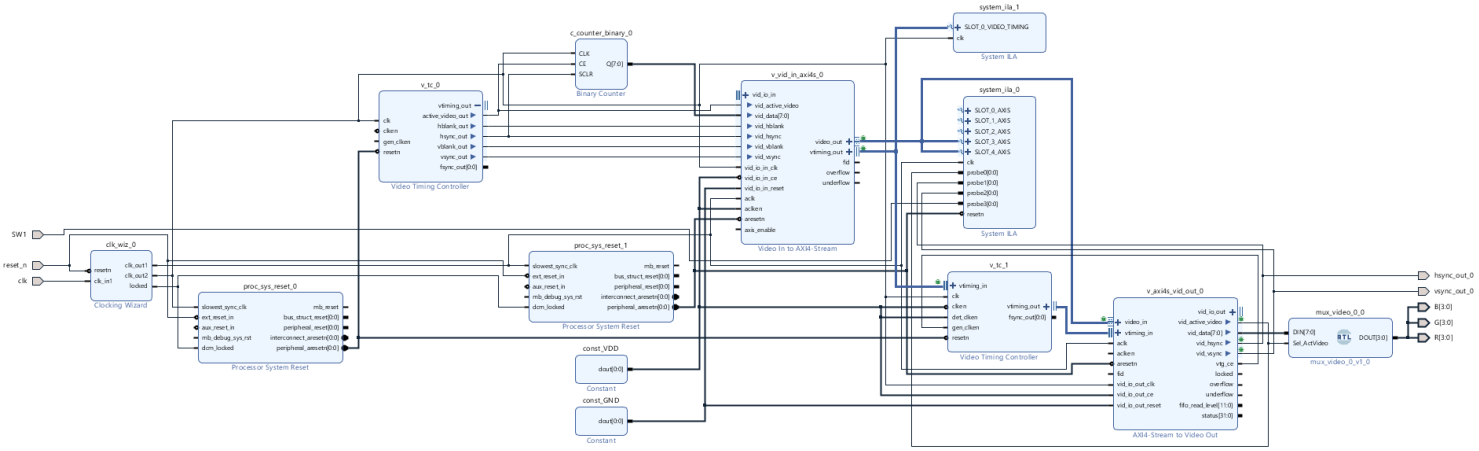

Figure 16 : Architecture interfaçage AXIS

L'IP « AXI4-Stream to Video Out » doit être utilisée en mode slave, car elle reçoit les signaux de synchronisation provenant du module « Video Timing Controller ». C'est ensuite elle qui génère le flux vidéo en respectant les timings fournis par ce contrôleur.

Simulation

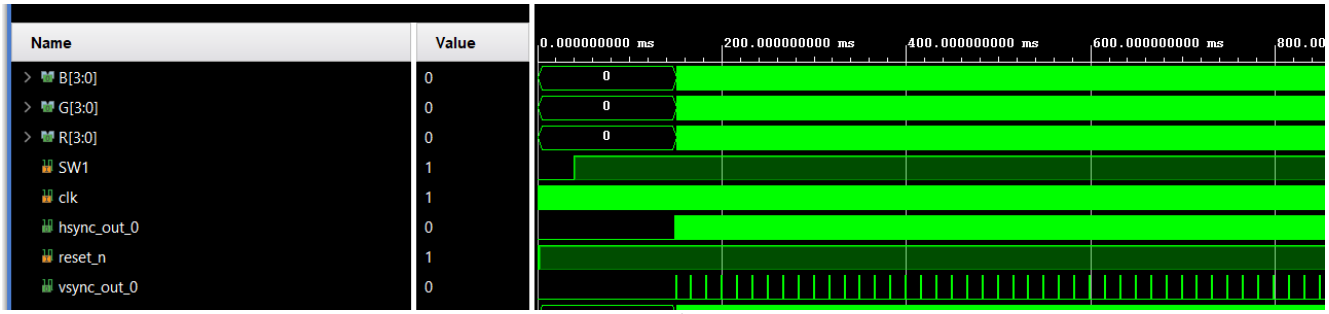

Figure 17 : Simulation - écriture image en AXIS

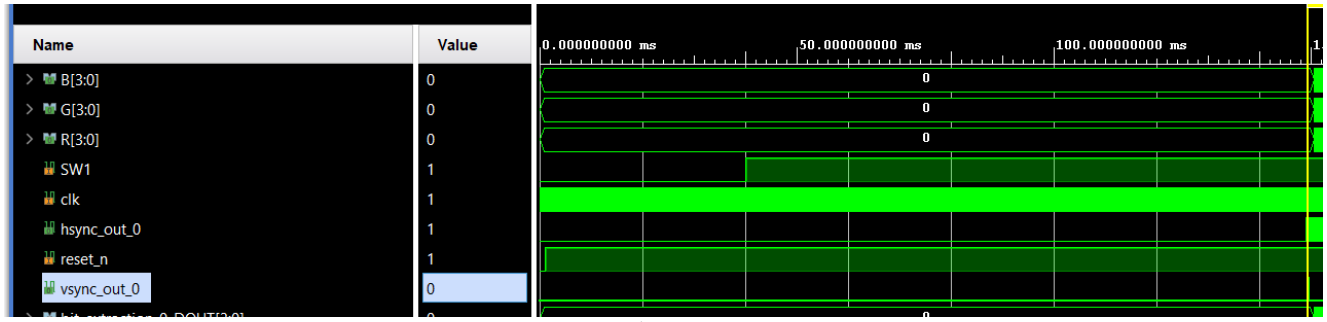

Figure 18 : Simulation - écriture image en AXIS (Zoom)

L'IP « AXI4-Stream to Video Out » met un certain temps à générer le flux vidéo car elle doit « accrocher » le signal AXI stream entrant. Cela met environ 150ms .

Debuggeur Hardware

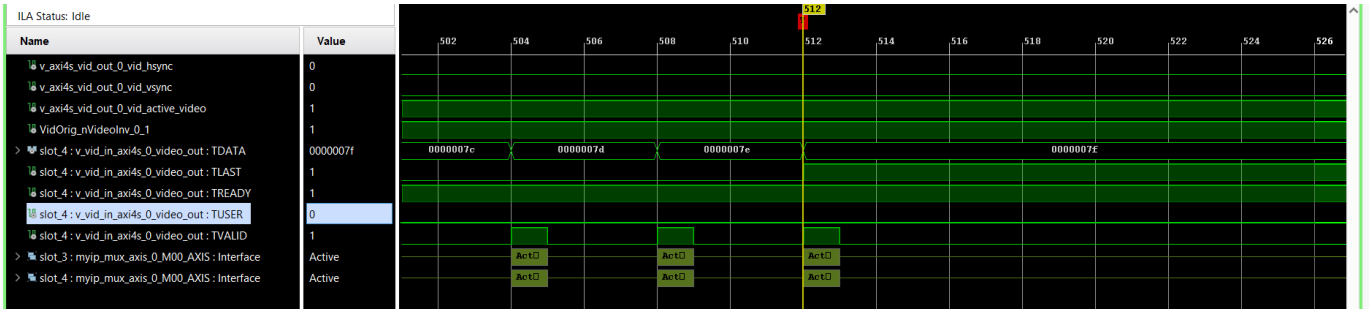

Figure 19 : ILA - écriture image en AXIS

Sur ce chronogramme, il est possible de voir la conversion du flux video en flux AXIS. Le signal *tready* est correctement à l'état haut tout le temps, le *tvalid* indique les nouvelles transactions et le *Tdata* montre la valeur du pixel à un moment donné.

Traitement flux video AXIS

IP inversion VHDL

Architecture

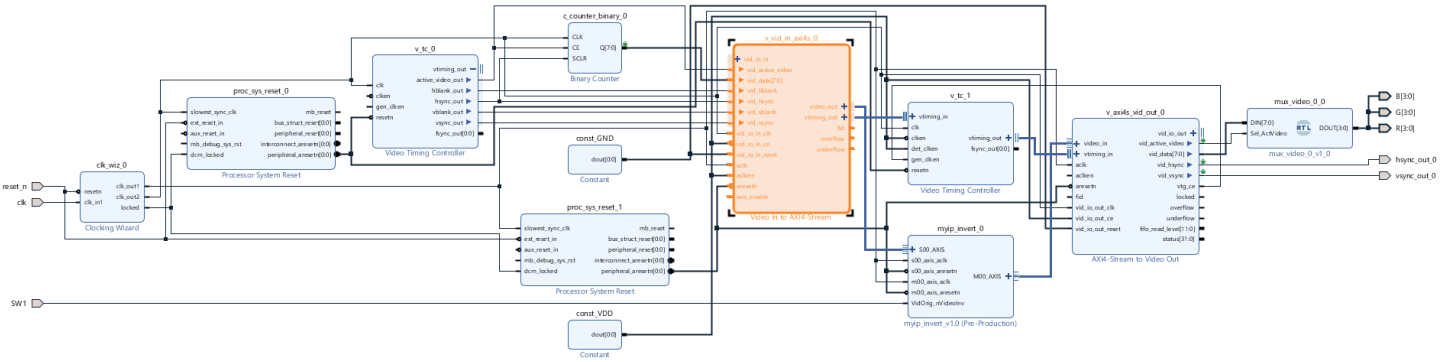

Figure 20 : Architecture IP inversion VHDL

Architecture avec l'IP inversion en 32 bits :

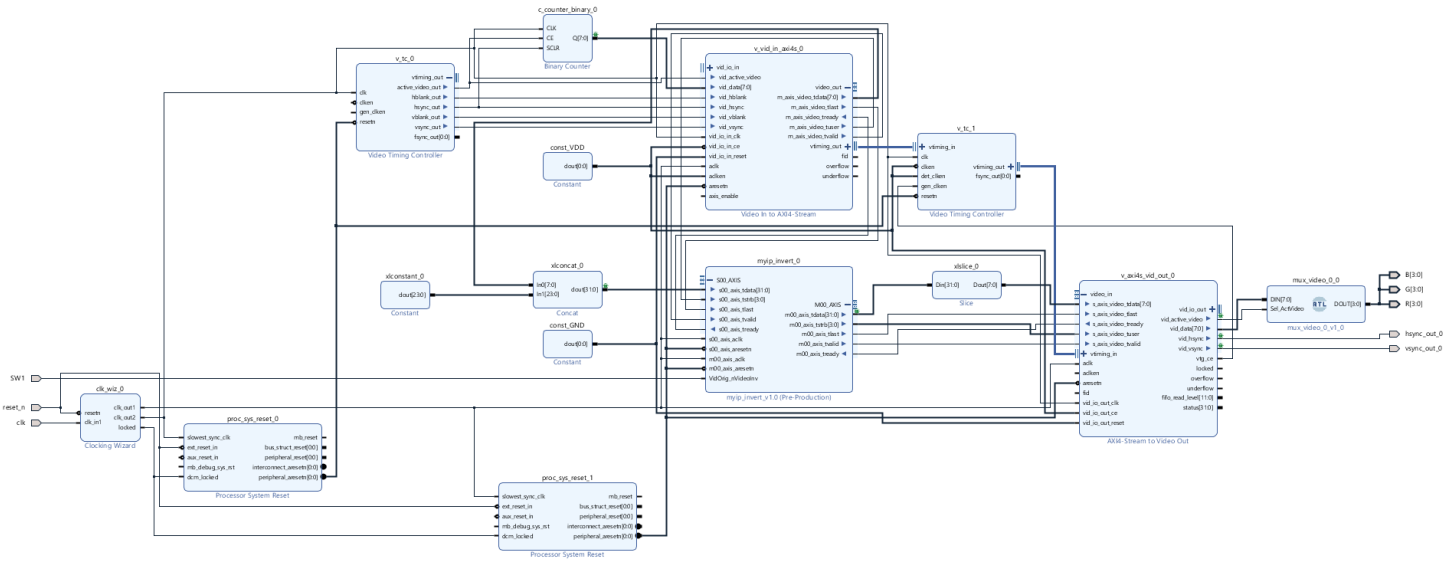

Figure 21 : Architecture IP inversion VHDL (data sur 32bits)

Architecture avec l'IP en 8 bits mais signaux AXIS connectés un par un :

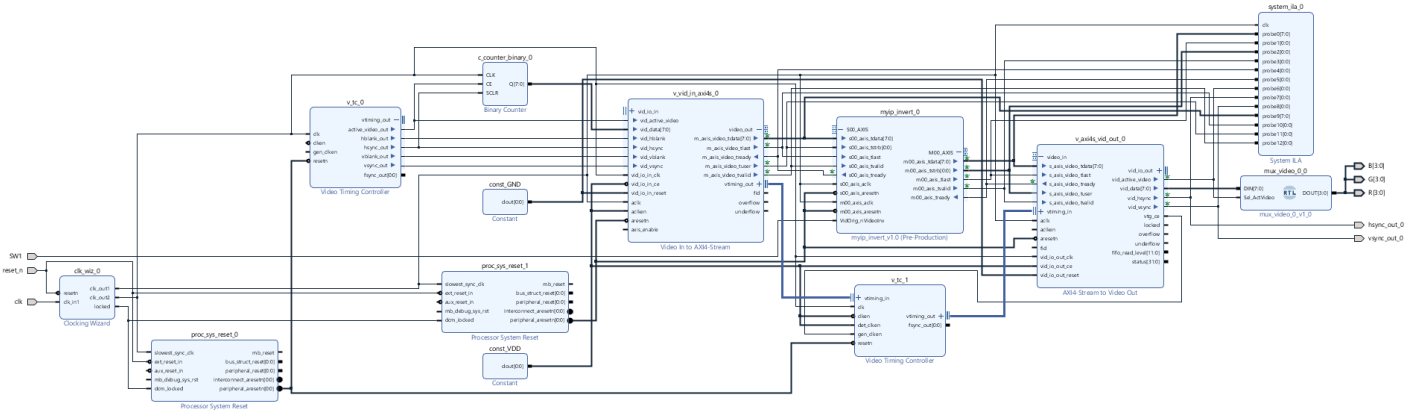

Figure 22 : Architecture IP inversion VHDL (signaux axis connectés un à un)

Lors de l'implantation de l'IP d'inversion (première architecture), aucun pixel ne s'affichait à l'écran. J'ai alors essayé de passer l'IP en 32 bits et de supprimer l'ILA, ce qui a résolu le problème. Cependant, cela me semblait étrange que la taille des données ou l'ILA soient à l'origine du dysfonctionnement. Je suis donc revenu à l'IP en 8 bits, mais cette fois en connectant les signaux AXIS un par un. Après ce correctif, tout a fonctionné correctement et j'ai pu afficher l'inversion sur l'écran.

Code de l'IP inversion VHDL

```
begin

process(s00_axis_aclk)
begin
	if rising_edge(s00_axis_aclk) then
		if s00_axis_aresetn = '0' then
			s00_axis_tready<= '0';
		else
			s00_axis_tready <= m00_axis_tready;
		end if;
	end if;
end process;

data_inverted <= not(s00_axis_tdata) when (VidOrig_nVideoInv = '1') else s00_axis_tdata;

process(m00_axis_aclk)
begin
	if rising_edge(m00_axis_aclk) then
		if m00_axis_aresetn = '0' then
			m00_axis_tvalid<= '0';
			m00_axis_tlast  <= '0';
			m00_axis_tdata  <= (others => '0');
		else
			m00_axis_tlast <= s00_axis_tlast;
			m00_axis_tdata <= data_inverted;


			if(m00_axis_tready = '1' AND s00_axis_tvalid = '1')then
				m00_axis_tvalid <= '1';
			else
				m00_axis_tvalid <= '0';
			end if;
		end if;
	end if;
end process;

m00_axis_tstrb <= s00_axis_tstrb;
```

Le premier process, synchronisé sur l'horloge *s00_axis_aclk*, initialise le signal *tready* de l'interface esclave. Si le reset (*s00_axis_aresetn*) est actif, *tready* est mis à 0. Sinon, il prend la valeur de *m00_axis_tready*, ce qui permet de signaler que le module maître est prêt à recevoir des données.

Ensuite, la ligne suivante réalise l'inversion des données entrantes. Si le signal *VidOrig_nVideoInv* vaut 1, les données tdata sont inversées bit à bit, sinon elles sont transmises telles quelles.

Le second process, synchronisé sur l'horloge *m00_axis_aclk*, gère l'interface maître. Lorsque le reset (*m00_axis_aresetn*) est actif, les signaux *tvalid, tlast* et *tdata* sont initialisés à 0. Sinon, *tlast* est copié depuis l'esclave et *tdata* reçoit les données qui peuvent être inversées. Le signal *tvalid* est activé uniquement si le maître est prêt (*m00_axis_tready* = 1) et que les données de l'esclave sont valides (*s00_axis_tvalid* = 1), sinon il reste à 0.

En fonctionnement

Le deuxième interrupteur, SW1, est à l'état bas (Pas d'inversion) :

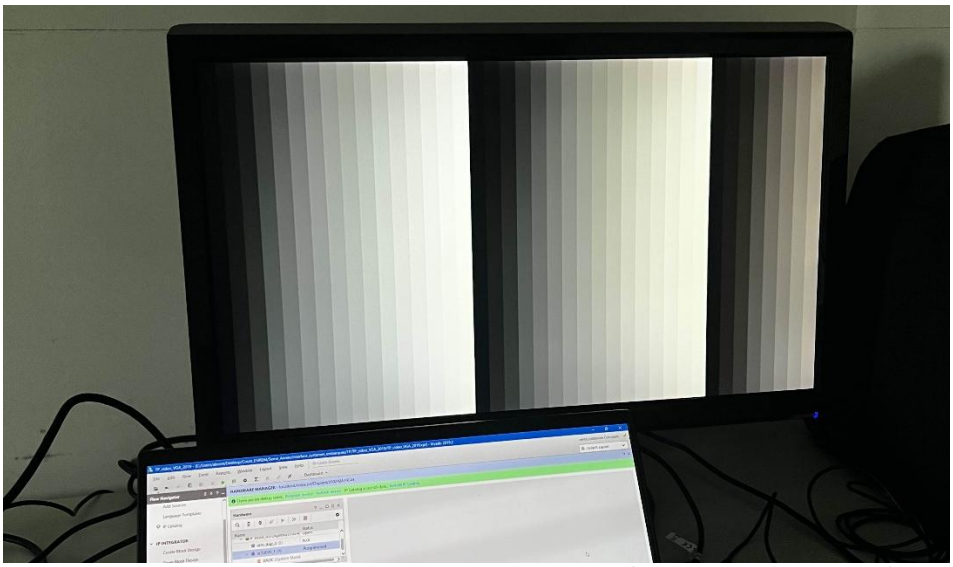

Figure 23 : IP inversion VHDL – affichage sur écran, pixels non inversés

Le deuxième interrupteur, SW1, est à l'état haut (inversion) :

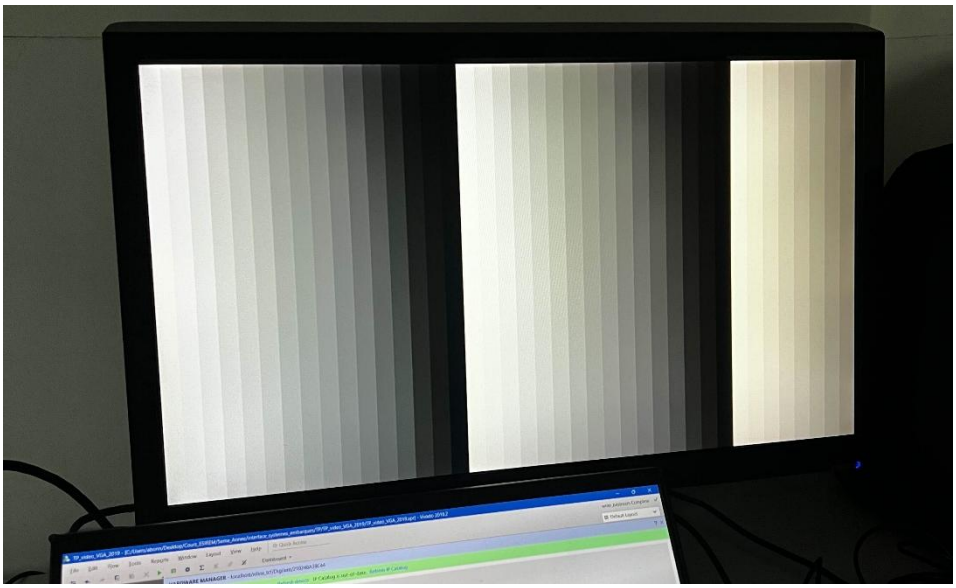

Figure 24 : IP inversion VHDL - affichage sur écran, pixels inversés

Debuggeur Hardware

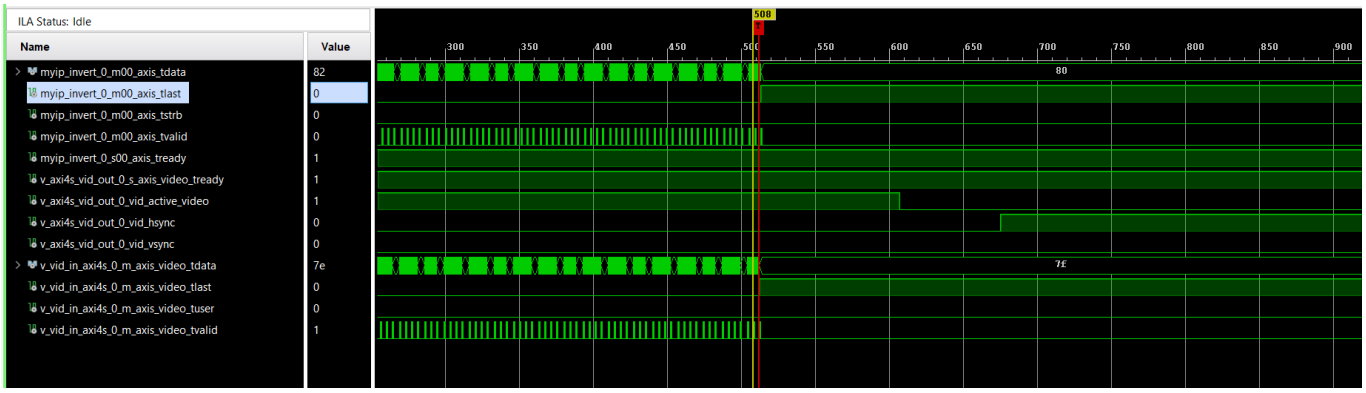

Figure 25 : IP inversion VHDL - ILA

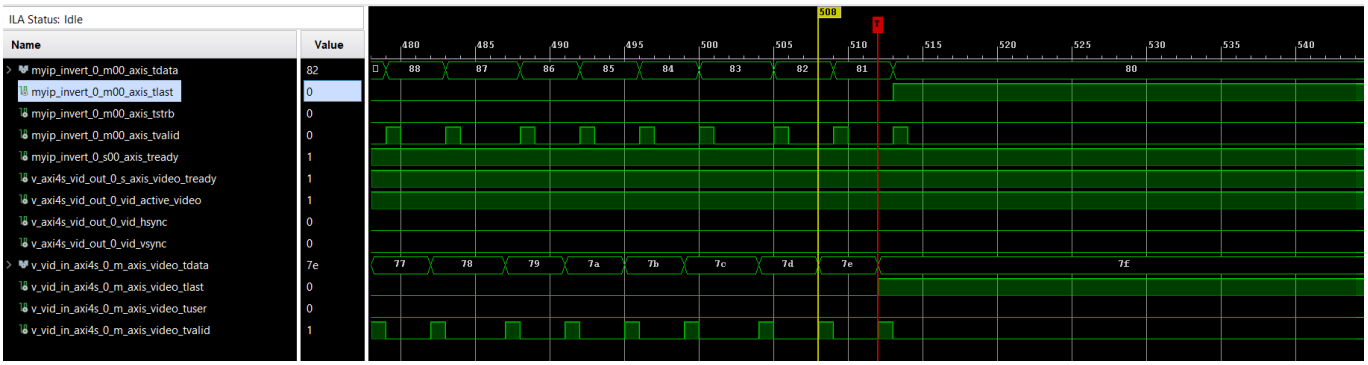

Figure 26 : IP inversion VHDL - ILA (Zoom)

D'après le chronogramme ci-dessus, on observe qu'une transaction AXIS entrante sur S00 est immédiatement suivie, au cycle d'horloge suivant, par une transaction sortante sur M00 depuis l'IP d'inversion, avec les données inversées. Les signaux *tready* du maître et de l'esclave restent constamment à 1, ce qui confirme la disponibilité continue du bloc. De plus, les signaux *tvalid* et *tlast* du maître sont correctement décalés d'un cycle d'horloge par rapport à ceux de l'esclave, assurant une transmission synchronisée des données.

IP inversion HLS

Architecture

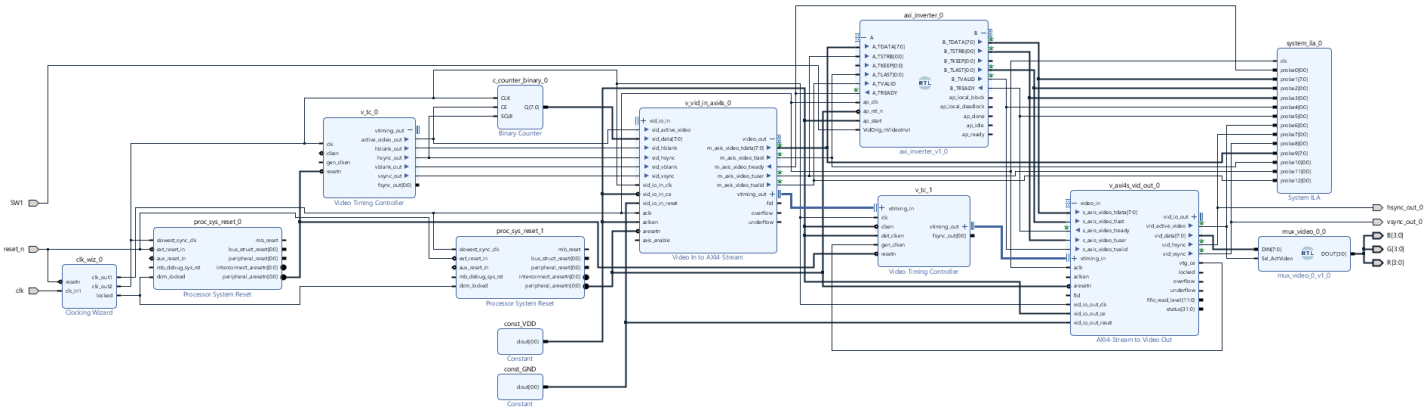

Figure 27 : Architecture IP inversion (HLS)

Code de l'IP inversion

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

#define DWIDTH 8
typedef ap_int<DWIDTH> type;
typedef hls::axis<type, 0, 0, 0> pkt;

void axi_inverter(hls::stream<pkt> &A, bool VidOrig_nVideoInvl,
        hls::stream<pkt> &B)
{
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B

   pkt tmp, t1;
   A.read(tmp); // Read AXI Stream in

   t1 = tmp; // Copy all AXIS signals from tmp

   // Modify only the AXIS data
   if (VidOrig_nVideoInvl) {
      t1.data = ~tmp.data; // Inversion
   } else {
      t1.data = tmp.data;
   }

   B.write(t1); //Write AXI Stream out
}
```

Debuggeur Hardware

Le deuxième interrupteur, SW1, est à l'état haut (inversion) :

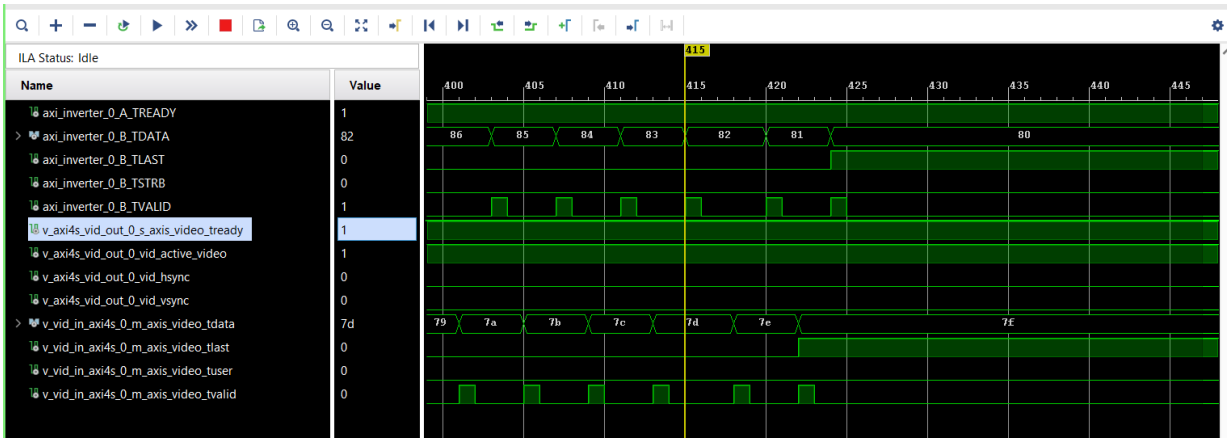

Figure 28 : IP inversion HLS – ILA

Contrairement à l'IP inversion VHDL, celle-ci met deux cycles d'horloge pour faire l'inversion.

IP HLS numéro binôme

Architecture

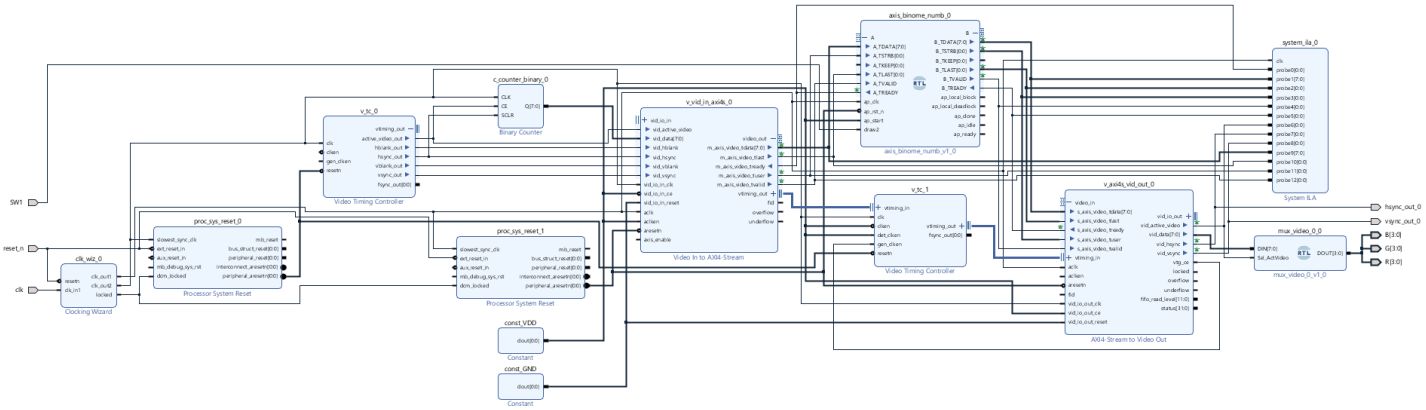

Figure 29 : IP HLS numéro binôme

Code de l'IP inversion

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

#define DWIDTH 8
typedef ap_int<DWIDTH> type;
typedef hls::axis<type, 0, 0, 0> pkt;

const int FONT_HEIGHT = 5;
const int FONT_WIDTH = 3;

// Describe Two symbol on 3x5 pixels
const type two[FONT_HEIGHT] = {
  0b111,
  0b001,
  0b111,
  0b100,
  0b111
};

void axis_binome_numb(hls::stream<pkt> &A, bool draw2, hls::stream<pkt> &B) {
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B

  pkt tmp, t1;
  type mask;

  while (true) {
      for(int y = 0; y < 480; y++){
          for(int x = 0; x < 640; x++){
              A.read(tmp); // Read AXIS input
              t1 = tmp;

              // Select only the pixels located in the top left corner of the image
              if(draw2 && y < FONT_HEIGHT && x < FONT_WIDTH) {
                  // Apply mask to select desired pixel
                  mask = 1 << ((FONT_WIDTH - 1) - x);

                  if((two[y] & mask)){
                      // Display white pixel according to the Two symbol
                      t1.data = 255;
                  }
                  else{
                      t1.data = 0;
                  }
              } else {
                  // No modif of pixels input
                  t1.data = tmp.data;
              }
              B.write(t1); // Write the result to the AXIS output
          }
      }
  }
}
```

Debuggeur Hardware

Le deuxième interrupteur, SW1, est à l'état bas (Pas d'affichage du numéro du binôme) :

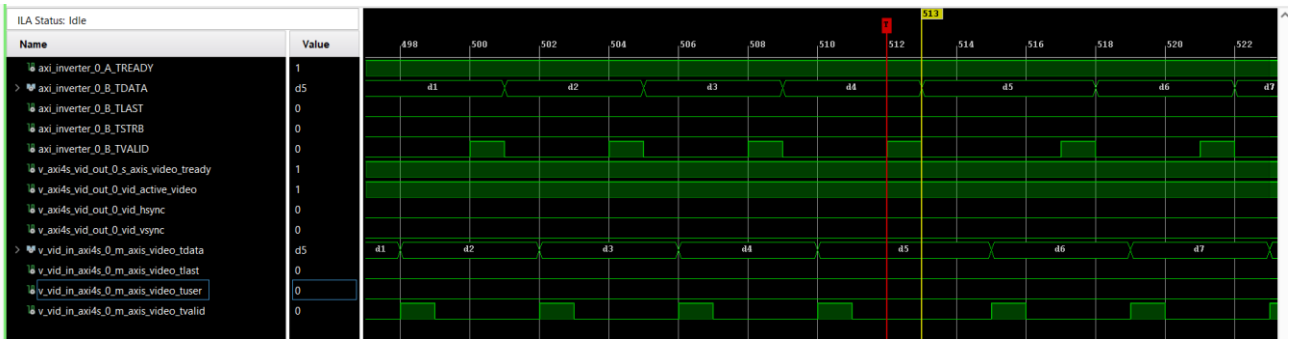

Figure 30 : IP HLS binôme number - ILA

Comme l'IP inversion HLS, celle-ci met deux cycle d'horloge pour faire le traitement. L'ILA a conservé l'ancien nom « axi_inverter », mais c'est bien mon IP Binôme Number qui est connectée.

En fonctionnement

Le deuxième interrupteur, SW1, est à l'état haut (Affichage du numéro du binôme) :

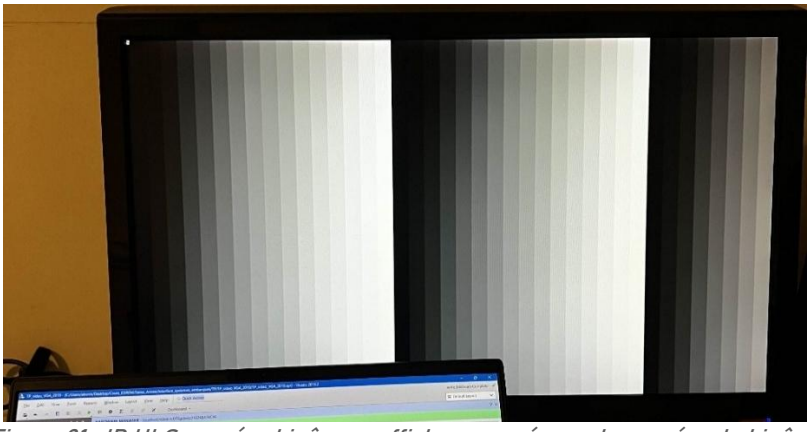

Figure 31 : IP HLS numéro binôme - affichage sur écran du numéro du binôme

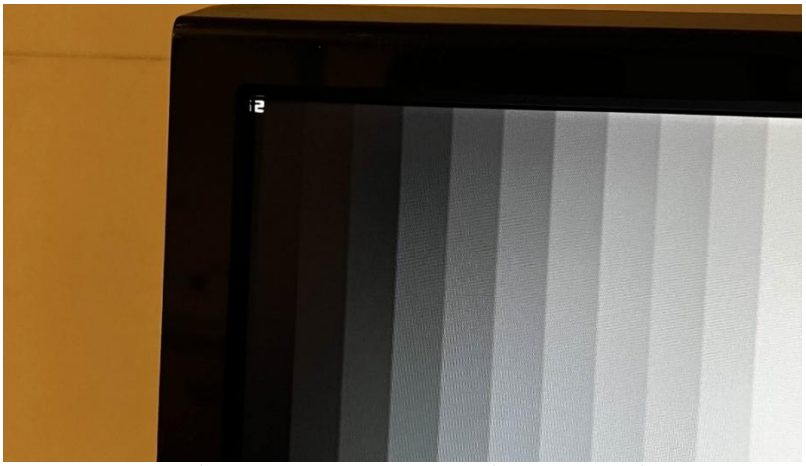

Figure 32 : IP HLS numéro binôme - affichage sur écran du numéro du binôme (Zoom)

Regroupement des trois IPs

Architecture (1ère version)

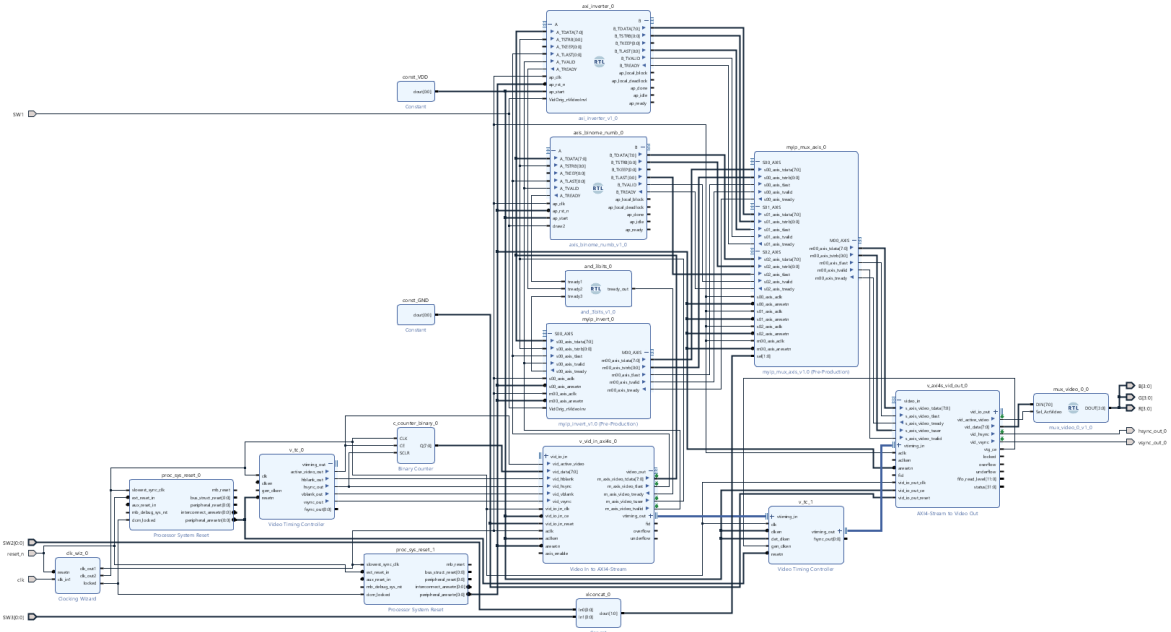

Figure 33 : Architecture avec les trois IPs

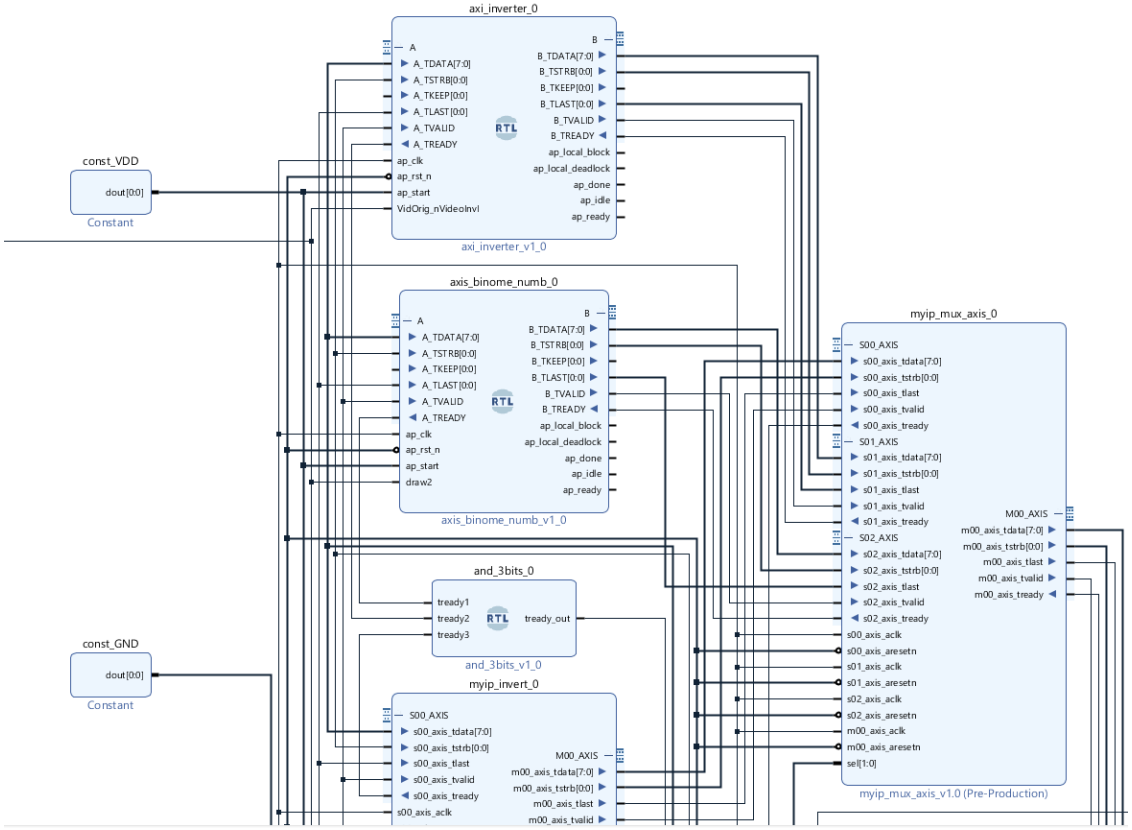

Figure 34 : Architecture avec les trois IPs (Zoom)

Pour relier les modules entre eux, le flux vidéo AXIS est connecté aux trois modules de traitement. Leurs signaux tready sont combinés à l'aide d'un opérateur AND, ce qui permet d'autoriser l'écriture AXIS uniquement lorsque les trois modules sont prêts.

La sortie AXIS vidéo est choisie à l'aide d'un signal sel, qui est contrôlé par deux switchs permettant de sélectionner le flux vidéo souhaité.

Code du multiplexeur

```
architecture arch_imp of myip_axis_mux_v1_0 is

   signal mux_tdata  : std_logic_vector(C_M00_AXIS_TDATA_WIDTH-1 downto 0);
   signal mux_tstrb  : std_logic_vector((C_M00_AXIS_TDATA_WIDTH/8)-1 downto 0);
   signal mux_tvalid : std_logic;
   signal mux_tlast  : std_logic;

begin

   ------------------------------------------------------------------------
   -- AXIS Multiplexeur
   ------------------------------------------------------------------------
   process(s00_axis_tdata, s00_axis_tvalid, s00_axis_tlast,
         s01_axis_tdata, s01_axis_tvalid, s01_axis_tlast,
         s02_axis_tdata, s02_axis_tvalid, s02_axis_tlast,
         s00_axis_tstrb, s01_axis_tstrb, s02_axis_tstrb,
         sel)
   begin
      case sel is
         when "00" =>
            mux_tdata  <= s00_axis_tdata;
            mux_tstrb  <= s00_axis_tstrb;
            mux_tvalid <= s00_axis_tvalid;
            mux_tlast  <= s00_axis_tlast;

         when "01" =>
            mux_tdata  <= s01_axis_tdata;
            mux_tstrb  <= s01_axis_tstrb;
            mux_tvalid <= s01_axis_tvalid;
            mux_tlast  <= s01_axis_tlast;

         when others =>
            mux_tdata  <= s02_axis_tdata;
            mux_tstrb  <= s02_axis_tstrb;
            mux_tvalid <= s02_axis_tvalid;
            mux_tlast  <= s02_axis_tlast;
      end case;
   end process;

   ------------------------------------------------------------------------
   -- AXIS Outputs
   ------------------------------------------------------------------------
   m00_axis_tdata  <= mux_tdata;
   m00_axis_tstrb  <= mux_tstrb;
   m00_axis_tvalid <= mux_tvalid;
   m00_axis_tlast  <= mux_tlast;

   ------------------------------------------------------------------------
   -- TREADY Management
   ------------------------------------------------------------------------
   s00_axis_tready <= m00_axis_tready;
   s01_axis_tready <= m00_axis_tready;
   s02_axis_tready <= m00_axis_tready;

end arch_imp;
```

Les modules de traitement deviennent prêts lorsque l'esclave en sortie du multiplexeur est prêt.

En fonctionnement

L'ensemble d'interrupteurs est configuré avec SW1 à l'état haut et SW2–SW3 positionnés sur 10 (soit la valeur 2) :

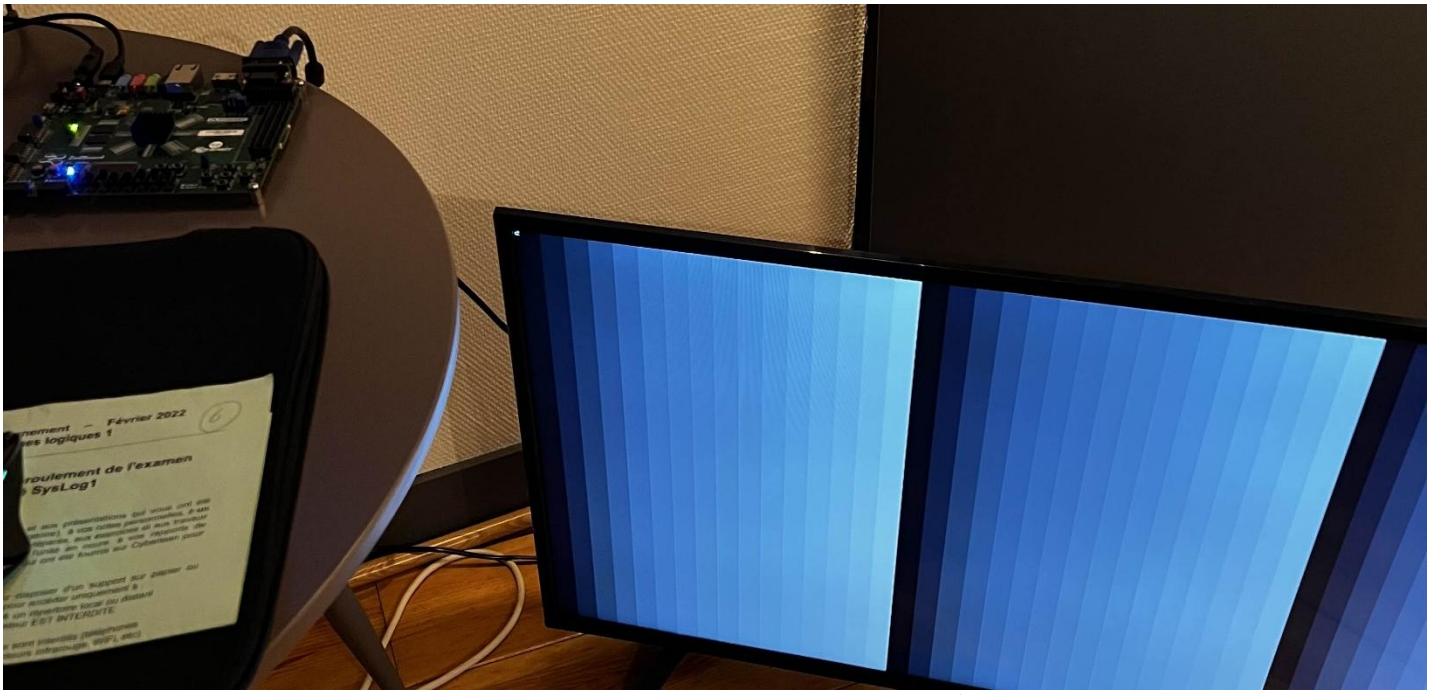

Figure 35 : Regroupement des trois IPs - affichage sur écran

L'ensemble des fonctionnalités fonctionne correctement. Il est possible de sélectionner chacun des modules de traitement grâce aux switchs SW2 et SW3, tandis que le switch SW1 permet d'activer ou non le traitement choisi.

Architecture (2ème version)

Ci-dessous, la version plus propre :

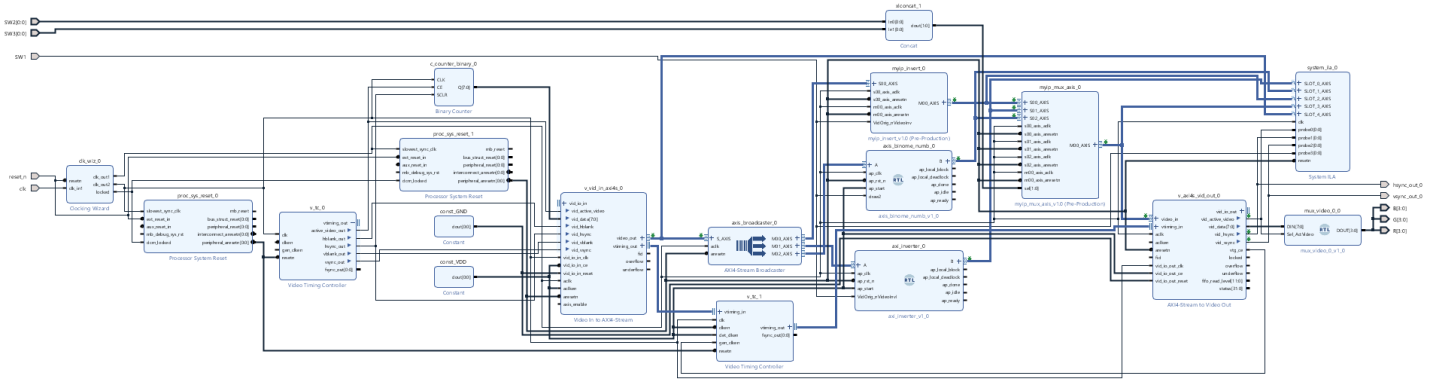

Figure 36 : Architecture avec les trois IPs (Plus lisible)

Dans cette architecture, j'ai ajouté le module « AXIS Broadcaster » afin de connecter facilement le flux vidéo AXIS aux trois modules de traitement. Le multiplexage des sorties est toujours géré par l'IP décrite précédemment.

Debuggeur Hardware

Les switchs SW2 et SW3 sont réglés sur 0 :

Figure 37 : Regroupement des trois IPs – ILA (IP inversion VHDL selectionnée)

Sur ce chronogramme, le flux AXIS sort d'abord de l'IP « Video_In_to_AXIS » (slot 4), puis entre dans l'IP d'inversion en VHDL (slot 2), où il subit un décalage d'un cycle d'horloge dû au temps de traitement. Le multiplexeur AXIS montre que c'est le flux provenant du slot 2 qui est sélectionné en sortie de l'interface maître, et qui est ensuite transmis à l'IP « AXIS_to_Video_Out ».

Les switchs SW2 et SW3 sont réglés sur 10 :

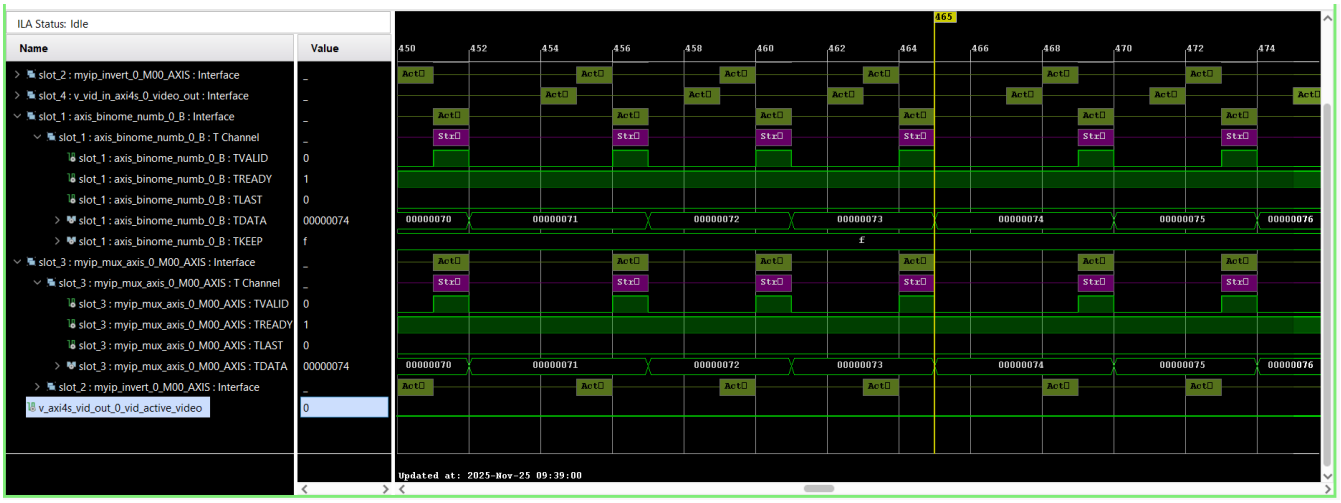

Figure 38 : Regroupement des trois IPs – ILA (IP HLS numéro binôme selectionnée)

Lorsque la combinaison des switches SW2 et SW3 vaut 10 (soit la valeur 2), l'IP Binôme Number est sélectionnée par le multiplexeur. Dans ce cas, le signal AXIS est décalé de deux cycles d'horloge avant d'entrer dans l'IP « AXIS_to_Video_Out ».

En fonctionnement

Cependant, cette version n'est pas fonctionnelle pour une raison encore inconnue. D'après l'ILA, les flux AXIS semblent corrects : les transactions sont régulières, tous les signaux *tready* sont à 1, et les *tlast* s'activent correctement à la fin de chaque image.

Le problème pourrait provenir de la gestion des ports AXIS dans Vivado. En effet, lorsque je connecte les signaux AXIS un par un, tout fonctionne correctement. En revanche, lorsque je relie directement l'ensemble d'un port AXIS à un autre, cela ne fonctionne plus.

Conclusion générale

Ce TP nous a permis de renforcer nos compétences en VHDL sur FPGA, en travaillant sur la génération et la gestion de flux vidéo. Nous avons utilisé une PLL pour fournir l'horloge, un compteur pour incrémenter les pixels et un module VHDL pour créer des bandes verticales, tout en gérant correctement les périodes de blank.

L'utilisation de l'ILA a permis de vérifier le comportement des signaux sur cible, et la génération d'un motif vidéo a confirmé le bon fonctionnement du système et la conformité des signaux *Hsync*, *Vsync* et *active_video*. Nous avons également appris à développer plusieurs modules avec une liaison AXIS que ce soit en VHDL ou à l'aide de Vitis HLS.