CSEE 4840 - Embedded Systems Design

ObTrak Project Final Report

Date: 05/11/2004

Project Team Members:
Marcio Buss (mob2101@columbia.edu)
Anuj Maheshwari (atm2104@columbia.edu)

1. Introduction

Image and pattern recognition has been an area of much research and development in recent years. An immaculate number of complex software has been written in order to get the best level of pattern recognition. The applications of this type of software range from office surveillance systems to tracking of objects for virtual reality gaming. The basis for this project was primitive forms of pattern recognition in black-and-white. We also developed the required infrastructure for color manipulation, namely, YUV to RGB conversion in software *and* in hardware.

The initial part of this document describes the project as a block diagram, highlighting relevant aspects of video image capturing and manipulation. The final sections show in more detail the modules implemented, some trade-off decisions we had to make and how the systems works as a whole. This project uses a XESS XSB-300E Board (*www.xess.com*) containing, among other components, (1) a Xilinx SpartanIIE FPGA with 300K system gates (2) a Philips SAA7114H video decoder (3) a 256K x 16 SRAM (4) a Texas Instruments THS8133B video DAC. The FPGA is loaded with a 32-bit microprocessor (*microblaze*) and we have access to a C compiler for such CPU (*microblaze-gcc*).

2. Project Description

The overall view of the object tracker "OBTRAK" is sketched in the following block diagram.

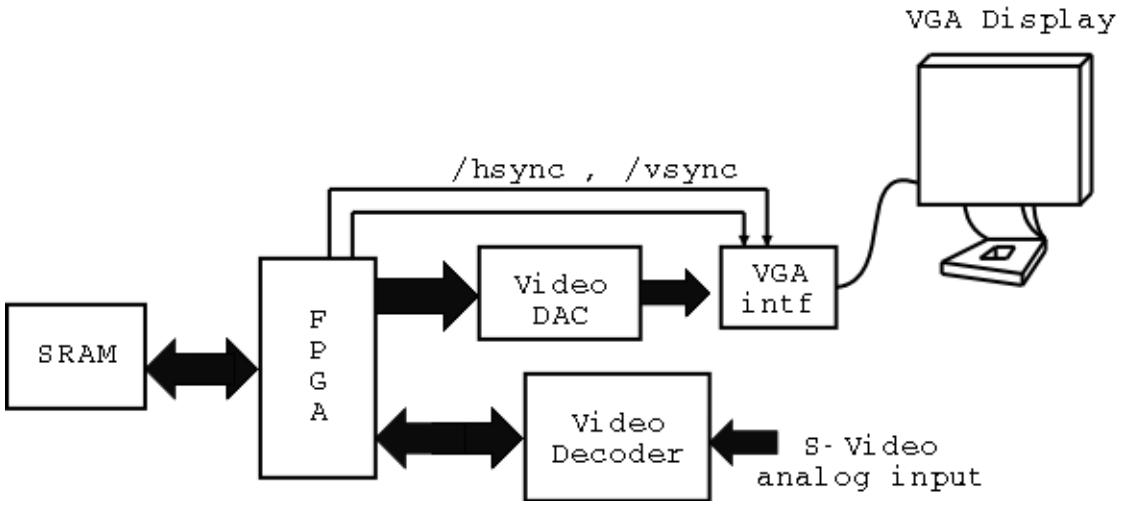


Fig. 1 - Obtrack block diagam.

Basically, the system expects an NTSC analog video signal at the RCA-Jack connector J7 on the XSB board. The analog signal is digitized by the video decoder and arrives at the FPGA through the IPD and HPD buses. IPD and HPD connect to SAA7114H's I-port and H-port, respectively, and are defined in the Xess XSB-300E manual. The IPD bus carries the 8-bit luminance values (Y) and the HPD carries the 8-bit chrominance information (UV). The data format at the video decoder output is YUV 4:2:2, 16-bit output via I-Port and H-Port (See Figure 5), configured through subaddress 93H (value C0H). A description of some relevant configuration registers is given later in the report.

The following figure shows at a higher level of detail the interconnection among the main components shown in Figure 1. It also depicts some relevant modules that were built inside the FPGA such as *video decoder interface* and *block RAMs*. "Block RAMs" are memory elements that can be found inside the SpartanIIE FPGA. They can be used as dual port RAMs with independent clock frequencies, which was very suitable for our project. Basically, *video decoder interface* receives the 8-bit luminance and 8-bit chrominance values from the video decoder and generates the address (*waddr*) for the block RAMs to store the luminance bytes. It also propagates *IDQ* (data valid indicator) and *ICLK* (video decoder output frequency) from the video decoder to the block RAMs. Although the data output from the video decoder interface is 16-bits wide (luma and chroma) only the luminance bytes are stored. More specifically, the block RAMs module contains four dual-port internal RAMs, and each one stores (through port B) a subset of the pixels from a single line of digital video at a time. We have decided to skip every other pixel in order to meet time and space constraints. The three least significant bits of *waddr* are used to select to which block RAM to write a given pixel's luminance. In our design, block 0 stores Y0, Y8, Y16, Y24,… on its memory cells 0,1,2,3,… block 1 stores Y2, Y10, Y18, Y26,… on its addresses 0,1,2,3,…, block 2 stores Y4, Y12, Y20, Y28,… on addresses 0,1,2,3,…, and block 3 stores Y6, Y14, Y22, Y30,… on addresses 0,1,2,3,… Here, Y*n* represents the luminance value of pixel '*n*'.

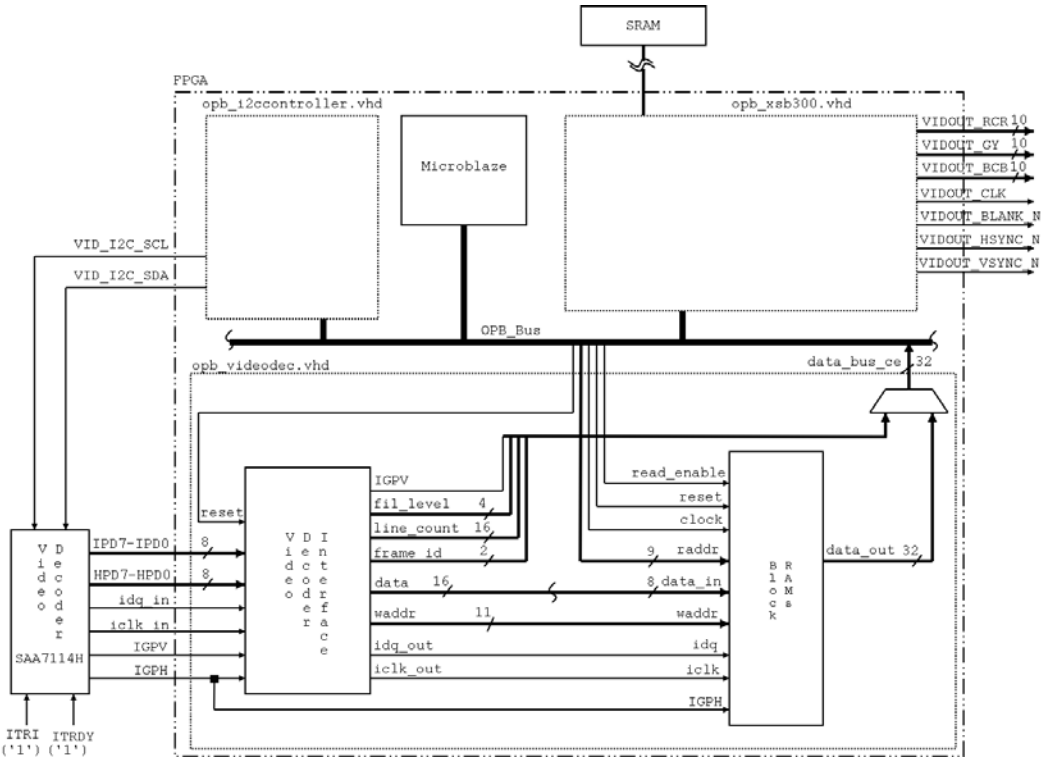


Fig. 2 – FPGA internal modules I2C controller, xsb300 bridge and videodec; SRAM and video decoder also shown.

The *opb_i2ccontroller* module is used to drive the I2C bus where the video decoder is attached. This module is conceptually quite simple – it only allows the microprocessor to write directly to pins VID_I2C_SCL and VID_I2C_SDA. Specifically, a sequence of instructions executed by microblaze is responsible for toggling SCL and SDA outputs in order to send the serial bits that configure the video decoder. The module *opb_xsb300* was developed by Cristian Soviani, and contains a memory controller, a "vga timing" module and a vga "wrapper" module. In short, the memory controller arbitrates the SRAM between the microprocessor and the vga module; the vga module constantly reads the SRAM starting at address 00800000H through 0084B000H, treating each byte as the

luminance information for a single pixel, and sends RGB signals to the video DAC. The *same* luminance byte is sent to the 8 upper bits of VIDOUT_RCR, VIDOUT_GY and VIDOUT_RCB, essentially creating a black-and-white image on the screen. This feature was a modification we had to insert on this module. The 2 lower bits of VIDOUT_RCR, VIDOUT_GY and VIDOUT_RCB are tied to '0'. Figure 3 below shows all the inputs and outputs of *opb_xsb300*.

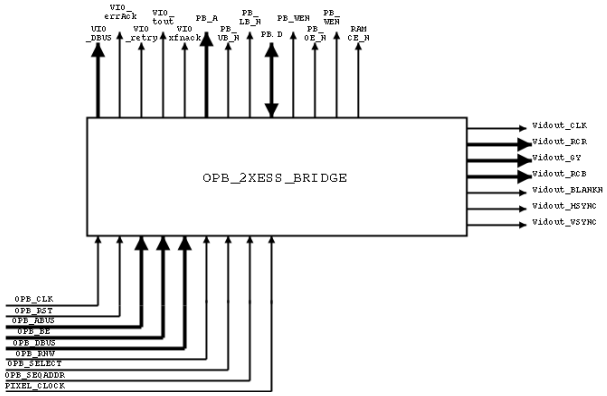


Fig. 3 – opb_xsb300 module block diagram.

BLOCK RAMs

Figure 4 shows in yet greater detail the block RAMs module. As depicted, this module is formed by four block RAMs instantiated as RAMB4_S8_S8, summing up to 512 bytes x 4, or 2048 bytes (See *block_ram.vhd*). As mentioned, these internal dual port memories can operate with independent clock frequencies at each port. In this project, the "B" ports are written to by the video decoder interface at *iclk,* and the "A" ports are read from the microprocessor at *OPB_Clk*.

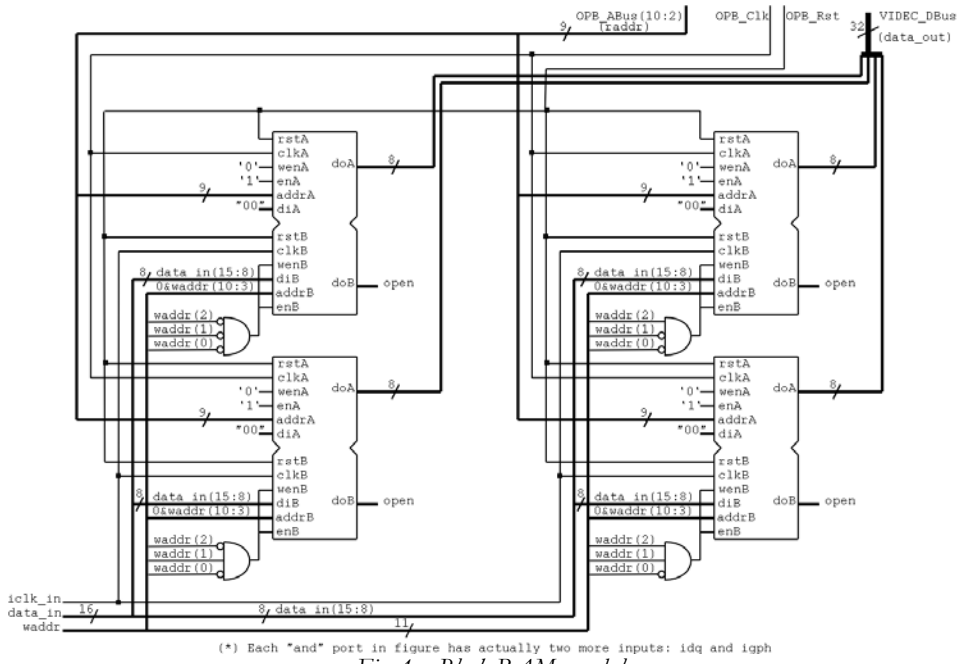


Fig. 4 – Block RAMs module.

The timing diagram on Figure 5 shows the waveforms at the input and output of the video decoder

interface that are relevant for the block RAMs module. It also shows internal signals such as *pix_count* and *active* – the former counts the valid bytes being sent by the video decoder, the latter goes to '1' right after the timing reference code "FF 00 00 SAV" has been transmitted. Notice that *waddr* increments only when *idq_in ='1'* **and** *active*='1'. The rationale here is that we don't want to store those bytes that correspond to timing reference code.

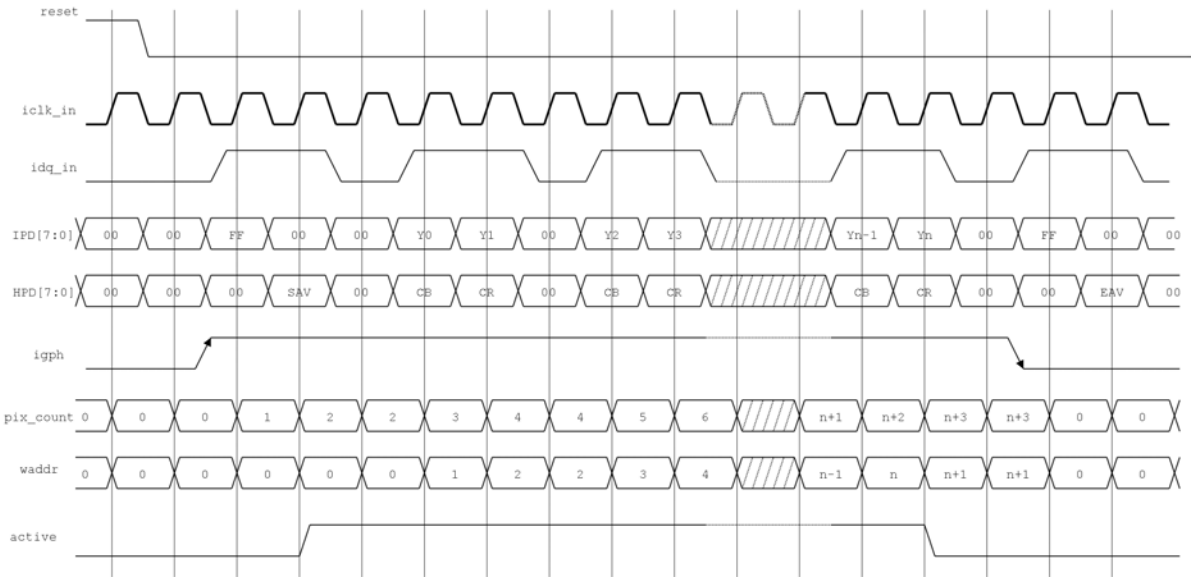


Fig.5 – Timing diagram for the video decoder interface – writing to the block RAMs.

The 16-bit output signal called *data* in Figure 2 is simply the concatenation of IPD and HPD, and the output signals *iclk_out* and *idq_out* are directly connected to *idq_in* and *iclk_in*.

As previously said, the block RAMs are written to by the video decoder interface and read from by microblaze, both operating at different clock frequencies (*iclk* and *OPB*_Clk, respectively). Clearly there is a need for some sort of synchronization between writes to and reads from the block RAMs – even more because this module only stores one line of digital video at a time (note that *waddr* is reset to '0' at the end of every line). In other words, each line of video has to be transferred to the SRAM before the next line is stored. The way we do this synchronization is through the *fil_level* output shown in Figure 2. Basically, this is a four-bit signal on which each bit corresponds to a "flag" indicating that a given section of the current line has been already written to the block RAMs. The existing levels are ¼, ½, ¾ and entire line, from least to most significant bits respectively. The timing diagram below sketches how *fil_level* evolves:

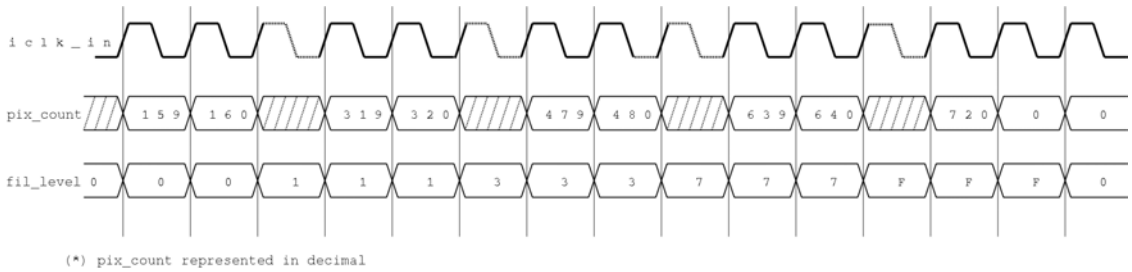


Fig.6 – fil_level timing diagram.

Note from Figure 2 that *fil_level* connects to the OPB data bus through *data_bus_ce* multiplexor. This

means it can be read by microblaze through an "`XIo_In`" instruction. We have specified address 0x01803FFC (allowed addresses must have "8" as the 3rd nibble) to do that, thus we can execute `XIo_In32(0x01803FFC)` and check for a specific level by masking out the other 3 bits. More precisely, the microprocessor can enter into a busy-wait state until, say, ¼ of the current line has been written to the block RAMs. When that "milestone" is detected through polling, microblaze executes a sequence of reads and writes in order to transfer the first quarter of the line from the block RAMs to the SRAM. Then, it undergoes again to a busy-wait state until ½ of the line is reached, transferring the second quarter, and so on until the entire line has been transferred to the SRAM. At this point (entire line), we essentially have synchronized at the "line" level (HSYNC). The following piece of code was extracted from *main.c* and shows the sequence of busy-wait/transfers mentioned here:

```
current_level = 0x01;
for (line_section = 0; line_section < 4; line_section++)
{
   while (! (XIo_In32(0x01803FFC) & current_level) ) ;   /* Busy-wait */

   if (current_level == 0x01) {
     start = 0;
     end   = 160;
   }
   else if (current_level == 0x02) {
     start = 160;
     end   = 320;
   }
   else if (current_level == 0x04) {
     start = 320;
     end   = 480;
   }
   else if (current_level == 0x08) {
     start = 480;
     end   = 640;
   }

   write_video(start, end, line);       /* Transfer the current fourth */
                                        /* of the  line  to  the  SRAM */
   current_level = current_level << 1;
}
```

In the above code, `write_video` is a function that transfers one-fourth of the current line each time it is called, starting at pixel "start" and ending at pixel "end". One point here is that "start" and "end" values identify the actual pixels, apparently not taking into account that we are skipping every other pixel. Nevertheless, `write_video` does take this into account. One final point relates to how we synchronize at the "frame" level (VSYNC). This is done by reading *IGPV* through *data_bus_ce* as well (See Figure 2). We have specified XIo_In32(0x01802FFC) to do that.

READING FROM BLOCK RAMs

A value can be read from the block RAMs (or written to, for this matter) in one clock cycle. However, this does not mean that each read operation *will* take only one cycle. In fact, the following timing diagram shows a complete read operation, which actually takes 3 cycles. We use as an example the instruction XIo_In32(0x018001FC) – the block RAMs are mapped to addresses 0x0180000 to 0x01800200, so this instruction reads four bytes from "somewhere" in the block RAMs. Specifically, byte 0 comes from block 0, byte 1 comes from block 1, byte 2 comes from block 2 and byte 3 comes from block 3. For alignment reasons, we use *OPB_ABus(10 downto 2)* as the actual read address. Therefore, *raddr* points to memory cell **7F** on all blocks. Note that each individual address reads 32 bits, or 4 bytes. Thus, the block RAMs module can be viewed as a black-box memory with an address space of 512 words of 32 bits each (9 bits for *raddr* therefore).

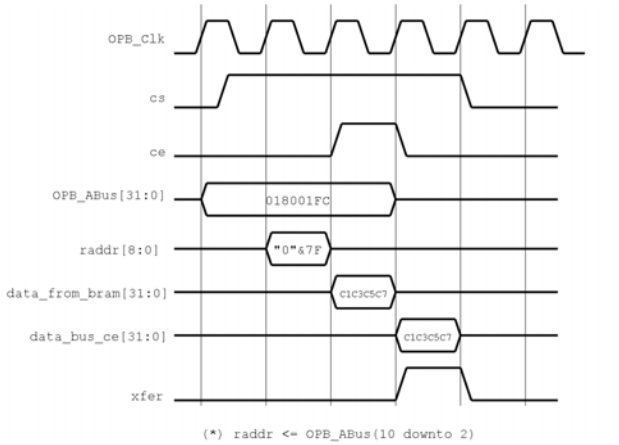


Fig.7 – Read operation timing diagram.

In the above figure, *data_bus_ce* is a 32-bit signal that is connected to the OPB bus data read port (Sln_Dbus in the general case, VIDEC_DBus in our case) through a multiplexor that has "*fil_level* & *frame_id* & *line_count* & *igpv*" as the second input ("&" here means concatenation). *xfer* is the transfer acknowledge signal that our module has to send back to the microprocessor indicating the completion of transfer. Only when this signal toggles to '1' does microblaze "grabs" whatever value is in the data bus. The *xfer* signal, as well as the *ce* signal in the above figure are generated by the following state machine:

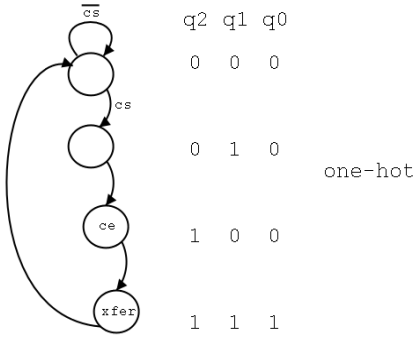


Fig.8 – State machine for read operation.

Basically, *chip select* (*cs*) goes to '1' whenever microblaze is accessing an address within the range 0x01800000 to 0x01803FFF. This corresponds to the block RAMs data space (up to 0x01800200) plus some extra room for future enhancements. The next rise of the clock will sense *cs* at '1', and the first transition of the state machine latches the correct read address at *raddr*. One cycle later the block RAMs make the data available at the A ports, and the *chip enable* (*ce*) signal goes to '1'. The third transition latches the data at *data_bus_ce* and sends the transfer acknowledge signal *xfer* to microblaze. The VHDL code for the chip select and the state machine, coded as "one-hot", follows:

```
cs <= OPB_select when OPB_ABus(31 downto 14) = "00000011000000000" else '0';
process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk='1' then
    q2 <= (not q2 and q1) or (q2 and not q1);
    q1 <= (cs and not q2 and not q1) or (q2 and not q1);
    q0 <= q2 and not q1;
  end if;
end process;
ce <= q2 and not q1 and rnw;
xfer <= q0;
```

YUV to RGB COLOR CONVERSION IN SOFTWARE

An alternative module was created in order to display color video instead of black-and-white, YUV to RGB conversion being done by software. In order to do that, both the hardware and software parts had to be modified. In the hardware side, not only luminance but also chrominance bytes have to be stored. To do so, *diB* inputs of blocks 0 and 2 are connected to the 8 upper bits of *data* signal (See Figure 2), whereas blocks 1 and 3 are connected to the 8 lower bits of *data*. The data organization changes slightly: block 0 now stores Y0, Y4, Y8, Y12,…, block 1 stores CB0, CB4, CB8, CB12,…, block 2 stores Y1, Y5, Y9, Y13,…, and block 3 stores CR1, CR5, CR9, CR13,… Since CB and CR bytes are sent on consecutive clock cycles, the strategy for skipping pixels also changes – we have to store two pixels and skip two. Therefore, the "and" gates at the *enB* inputs have a different input set in order to enable two blocks together each time we want to do a write. Blocks 0 and 1 have /waddr(1) *and* /waddr(0). Blocks 2 and 3 have /waddr(1) *and* waddr(0). Finally, all *addrB* inputs on the "B" ports are connected to *waddr(10 downto 2)*. Almost nothing else needed to be modified in the hardware part. On the software side, we have to be aware that each time microblazes executes an `XIo_In32` instruction it now reads 2 pixels worth of information, organized as Y_i-CB-Y_{i+1}-CR. Therefore, we have to execute twice the number of reads we did before for black-and-white, which stores luminance bytes only. Fortunately, there is enough room in the block RAMs to store one line of color digital video. The following code was extracted from *conversion.c*, and contains the necessary functions for the color space conversion.

```
#include "xbasic_types.h"
#include "xio.h"

#define W 320
#define H 240
#define VGA_START 0x00800000

#define YUV2RGB(y, u, v, r, g, b)\
  r = y + ((v * 1434) / 2048);\
  g = y - ((u * 406) / 2048) - ((v * 595) / 2048);\
  b = y + ((u * 2078) / 2048);\
  r = r < 0 ? 0 : r;\
  g = g < 0 ? 0 : g;\
  b = b < 0 ? 0 : b;\
  r = r > 255 ? 255 : r;\
  g = g > 255 ? 255 : g;\
  b = b > 255 ? 255 : b


void convert_to_color()
{
  int i, r, g, b;
  int y0_u_y1_v;
  int rgb_2_pixs;
  int y0, y1, u, v;
  int rgb_pixel_y0, rgb_pixel_y1;

  for (i = 0; i < W*2*H; i+=4)
  {
    // Read Y0-Cb-Y1-Cr from the SRAM
    y0_u_y1_v = XIo_In32(VGA_START + i);

    // Separate Y0-Cb in two variables
    y0 = (y0_u_y1_v >> 24) & 0xFF;
    u  = (y0_u_y1_v >> 16) & 0xFF;
    u  = u - 128;

    // Separate Y1-Cr in two variables
    y1 = (y0_u_y1_v >> 8) & 0xFF;
    v  = y0_u_y1_v & 0xFF;
```

```
        v  = v - 128;


        // Convert Y0-Cb-Cr to RGB
        YUV2RGB (y0, u, v, r, g, b);

        // Get the 5 most significant bits of 8-bit red
        r = r & 0xF8;

        // Get the 6 most significant bits of 8-bit green
        g = g & 0xFC;

        // Get the 5 most significant bits of 8-bit blue
        b = b & 0xF8;

        // Shift green and blue to form an 16-bit rgb
        r = r << 8;
        g = g << 3;
        b = b >> 3;

        // Pack the just generated 5-6-5 into 16-bits
        rgb_pixel_y0 = r | g | b;

        // Convert Y1-Cb-Cr to RGB
        YUV2RGB (y1, u, v, r, g, b);

        // Get the 5 most significant bits of 8-bit red
        r = r & 0xF8;

        // Get the 6 most significant bits of 8-bit green
        g = g & 0xFC;

        // Get the 5 most significant bits of 8-bit blue
        b = b & 0xF8;

        // Shift green and blue to form an 16-bit rgb
        r = r << 8;
        g = g << 3;
        b = b >> 3;

        // Pack the just generated 5:6:5 into 16 bits
        rgb_pixel_y1 = r | g | b;

        rgb_pixel_y0 = rgb_pixel_y0 << 16;
        rgb_pixel_y0 = rgb_pixel_y0 & 0xFFFF0000;

        rgb_2_pixs = rgb_pixel_y0 | rgb_pixel_y1;

        XIo_Out32(VGA_START + i, rgb_2_pixs);
      }
    }
```

Basically, after an entire frame of video has been transferred to the SRAM, the video capture stops and the stored frame is converted from YUV to RGB. Since we used RGB 565, exactly the same memory space in the SRAM could be used for storing the data before and after conversion. More precisely, 32 bits of memory can store 2 pixels in YUV format (Y_i-CB-Y_{i+1}-CR) or 2 pixels in RGB 565 format ($R_i G_i B_i$-$R_{i+1}G_{i+1}B_{i+1}$). Therefore, we can read 32 bits in YUV, perform the conversion, and store the data back as RGB in the exact same address. As expected, though, the conversion of an entire frame is too slow to allow real time video streaming.

YUV to RGB COLOR CONVERSION IN HARDWARE

Since the color space conversion is highly timing consuming, we decided to transfer the YUV to RGB conversion from software to hardware. In other words, the *video decoder interface* module receives luminance and chrominance information, as before, but it is now enhanced with an internal pipeline

that does essentially what the `YUV2RGB` macro performs in software. In this way, we are able to write RGB565 directly into the block RAMs, using the same hardware for the block RAMs module that was devised in the previous section. As expected, we were able to display real time color video on the screen, since the YUV to RGB conversion in hardware allow us to run the system at the same speed as if we were in black-and-white mode. The following figure shows the color space conversion pipeline that has been implemented in VHDL (See *video_decoder_intf.vhd*).

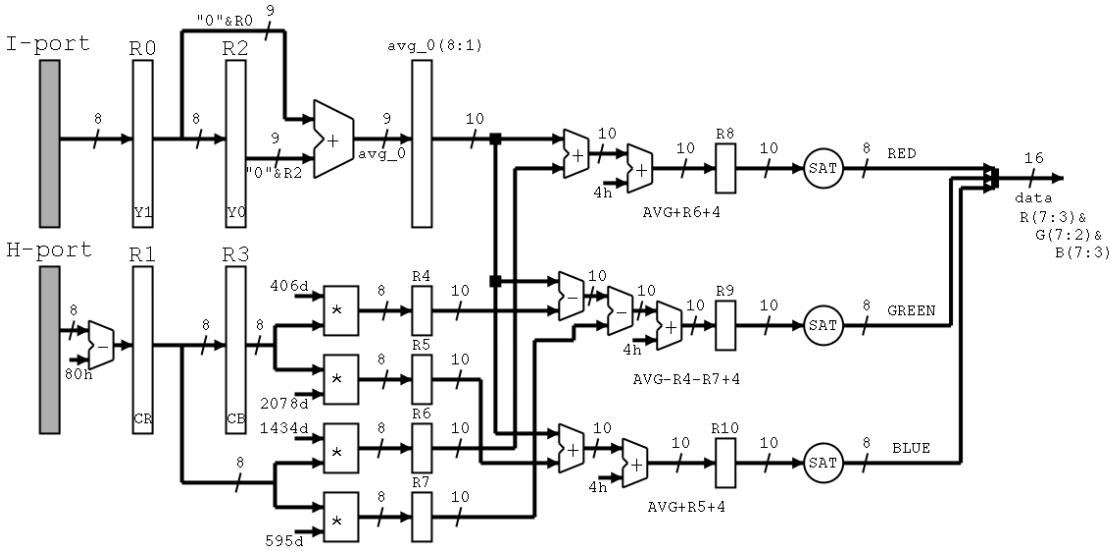


Fig.9 – YUV to RGB color space conversion implemented in the video decoder interface module.

In this mode, we do not "skip" incoming pixels per se. Instead, we average the luminance information for every two pixels being sent as Y_i-CB-Y_{i+1}-CR and use [$(Y_i+Y_{i+1})/2$), CB and CR] to convert from YUV to RGB. In this sense, we still store only half actual line of digital video in the block RAMs but, at the same time, have a smoother representation of the image. The multipliers shown in the above figure were created using the core generator software included in the XST tools. Since all multiplications involve constants, the multipliers were able to perform the operations in one cycle of *iclk*. Figure 10 shows the control state machine for the above pipeline. An important point here is that each transition in this state machine only happens when *iclk* ticks from '0' to '1' **and** *idq_in = '1'*.

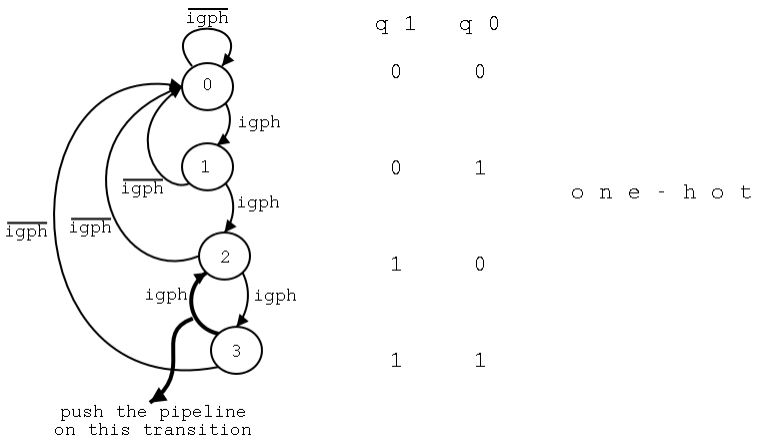


Fig.10 – Control state machine for the YUV to RGB pipeline shown in Figure 9.

In other words, the state machine only operates on valid data, making eventual transitions only when data validator (*idq*) is high. In essence, control stays put at state '0' while there is no incoming video (*igph* = '0'). The next two transitions, 0 to 1 and 1 to 2, are solely responsible for ignoring the first 2 valid bytes of any given line (the rationale is that we want to skip the timing reference code "FF 00 00 SAV"). All the interesting things happen at transitions 2 to 3 and 3 to 2. The former is when the video decoder sends Y_i-CB and the latter is when it sends Y_{i+1}-CR. Right at the second transition the pipeline registers R0, R1, R2 and R3 contain Y_{i+1}, CR, Y_i, CB, as illustrated on Figure 9 for $i = 0$, i.e., considering the first valid pixel of a given line. At exactly this time, we want to "push" the pipeline since we have right set of data on the "fetch" registers. Therefore, when state variables *q1* and *q0* shown in Figure 10 are both *'1'*, (and *idq* is '1') we assert a control register named *'A'* to '1'. The next four cycles will propagate this *'1'* until control register *'D'* is reached. The following code was extracted from *video_decoder_intf.vhd* :

```
-- Propagate "push" thru the pipeline
process (iclk_in)
begin
  if iclk_in'event and iclk_in='1' then
    A <= q1 and q0 and idq_in;
    B <= A;
    C <= B;
    D <= C;
  end if;
end process
```

Register *'D'* is used as the validator for incrementing *waddr* and *pix_count*, setting *active* signal and it is directly connected to *idq_out* signal. Notice that we cannot connect *idq_in* directly to *idq_out* as we did before on Figure 5 for two reasons: (1) there is a delay of 5 cycles due to the 5 stages of the pipeline, and (2) *idq_in* is high for two cycles when transmitting two adjacent pixels Y_i-CB-Y_{i+1}-CR, however we are averaging Y_i and Y_{i+1} and therefore we want *idq_out* to be high during one cycle only for every two valid pixels!

VIDEO DECODER CONFIGURATION REGISTERS

One of the challenges we faced on this project was to configure the video decoder properly and get it running. There are quite a few registers that need to be configured, and some subsets of these registers are dependent on one another. Fortunately, after some experiments with "default" values plus some trial-and-error strategy, we were able to get the right set of values. We list below a subset of subaddresses-values pairs that we found crucial for correct operation in terms of synchronism control and response speed. A great deal of registers is devoted to tasks such as luminance control, chrominance control, hue saturation, etc. that, although relevant, do not interfere directly with the desired "30 frames per second image" being displayed.

Subaddress	Value used	Function
06H	EBH	Horizontal sync start
07H	E0H	Horizontal sync stop
08H	59H	Synchronism control
90H	00H	Task handling
91H	08H	Scaler input source and format definition
92H	10H	Reference signal definition at scaler input
93H	C0H	I-port output formats and configuration
94H	10H	Horizontal input offset (XO)

95H	00H	Horizontal input offset (XO)
96H	D0H	Horizontal input (source) window length (XS)
97H	02H	Horizontal input (source) window length (XS)
98H	0AH	Vertical input offset (YO)
99H	00H	Vertical input offset (YO)
9AH	F2H	Vertical input (source) window length (YS)
9BH	00H	Vertical input (source) window length (YS)
9CH	D0H	Horizontal output (destination) window length (XD)
9DH	02H	Horizontal output (destination) window length (XD)
9EH	F0H	Vertical output (destination) window length (YD)
9FH	00H	Vertical output (destination) window length (YD)

Registers 9CH and 9DH define 720d as the horizontal output window, and registers 9EH and 9FH define 240d as the vertical output window (720 x 240). The complete set of register values and addresses used in the project can be found in *main.c*.

OBJECT TRACKING SOFTWARE

The object tracking module was implemented in software, and can be found in *track_object.c*. It currently does the simple task of finding a white square on a black background. If more than one object is placed in the scene, the tracking algorithm will return the largest white square placed in there. (specifically, the object to be tracked does not necessarily have to be a square. The tracking software actually looks for a white object that eventually circumscribes a white square with minimum side size of *n* pixels, where *n* is a parameter that can be defined prior to compilation). The driving ideology for this algorithm is that of a sliding window. The algorithm looks for maximum consistency with the constraints provided in the form of minimum dimensions (X and Y) to be considered a shape (in this case a rectangle). The pseudo-code follows, and a flowchart is found in the last page of the report.

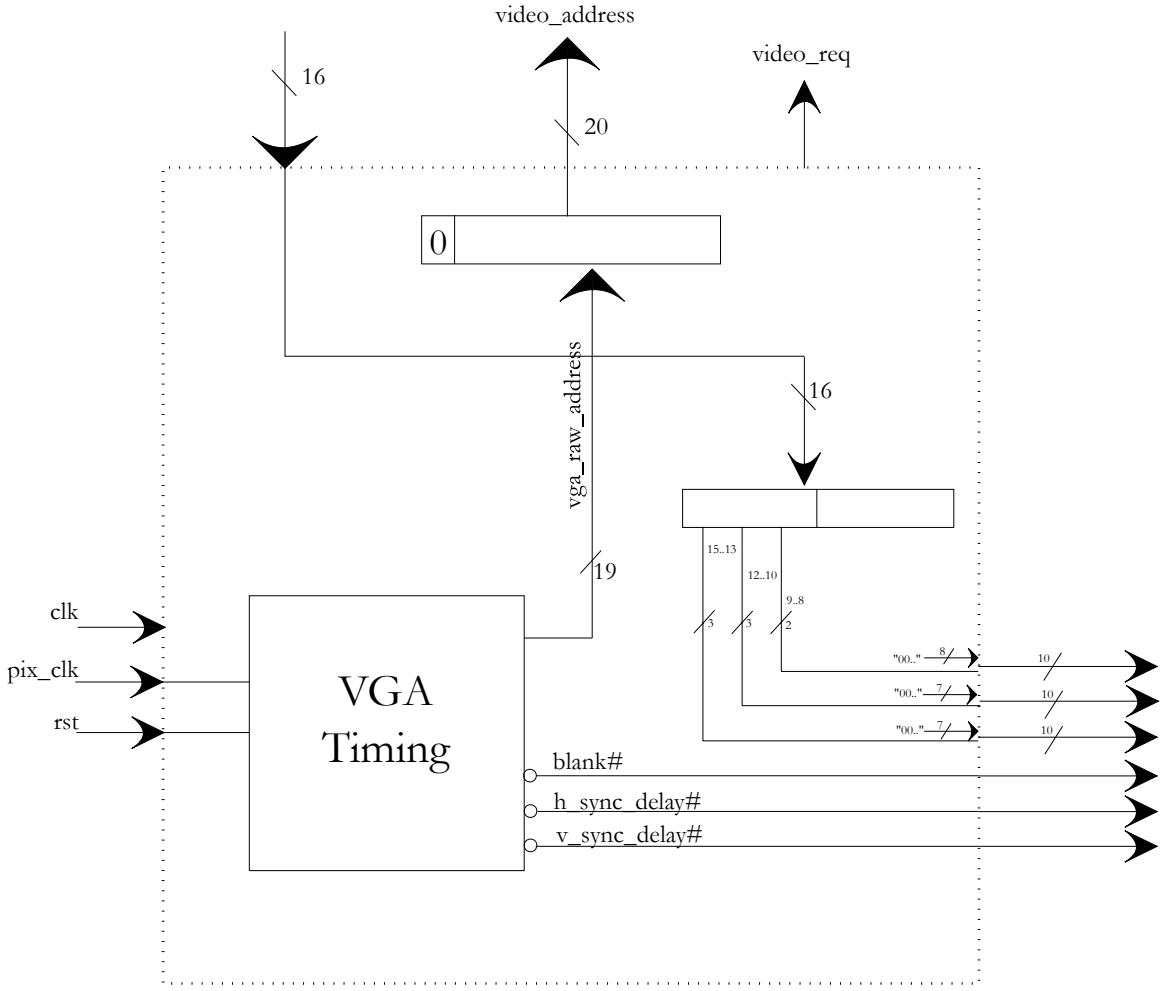
Step 01	`Grab image from video processor / camcorder, perform resizing / color`
Step 02	`Store the image to memory (into a A x B matrix)`
Step 03	`Set counter TOTAL = 0`
Step 04	`Set counters X = 1, Y = 1, MATCH_X=0, MATCH_Y=0`
Step 05	`G = X, H = Y`
Step 06	`TOTAL = TOTAL + CAM_IMAGE[G,H]`
Step 07	`If G < (X+M),  G = G+1; JUMP to Step 06`
Step 08	`If H < (Y+N), H = H+1; Jump to Step 06`
Step 09	`If TOTAL = 0 (all locations 0) MATCH_X = X, MATCH_Y = Y; JUMP to Step 12`
Step 10	`If X < (A-M) X = X+1; JUMP to Step 05`
Step 11	`If Y < (B-N) Y = Y+1; JUMP to Step 05`
Step 12	`If MATCH_X !=0 PRINT "MATCH FOUND AT" MATCH_X , MATCH_Y`

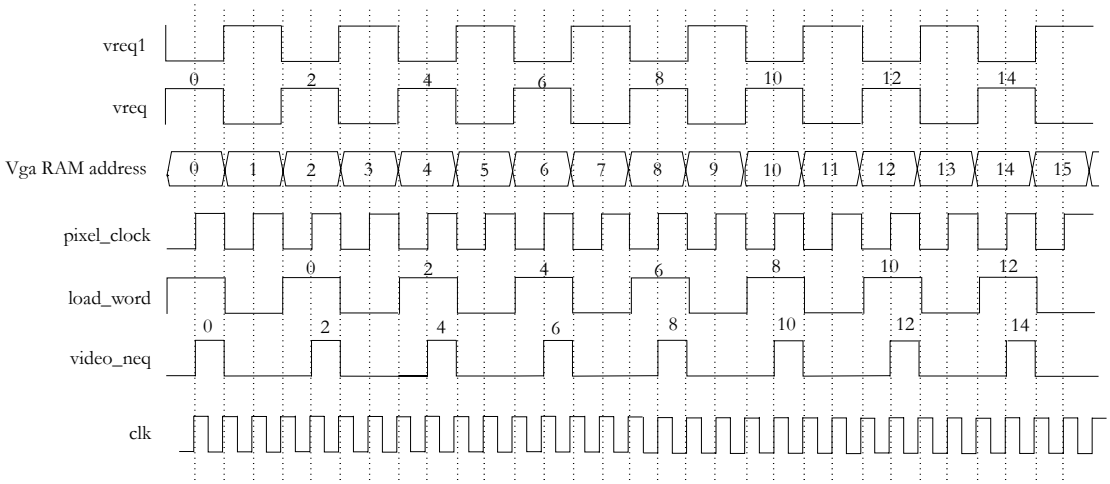
CONCLUSIONS

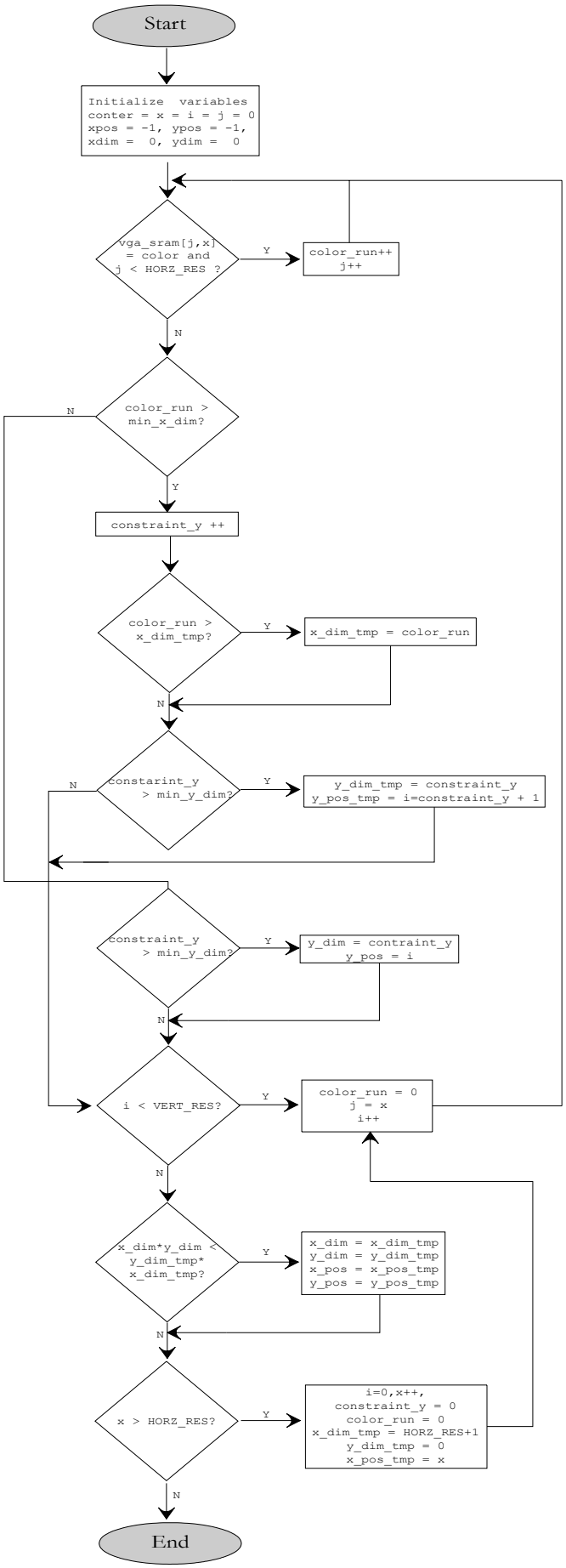
We believe that we have developed a successful project. The goal of tracking an object was achieved and a couple of enhancements were produced in terms of color video. The YUV to RGB conversion in software and then in hardware were very exciting experiences. Overall, having worked with an entire system composed of a microprocessor (microblaze), a C compiler for it (microblaze-gcc), a hardware/software interface through the OPB bus and the hardware modules developed for capturing digital video was a huge learning experience in terms of computer architecture and embedded systems. By designing the same functionality both in software and hardware we were able to truly face the trade-offs associated with this type of engineering decisions. Some of the major challenges faced during the development of the project were related to putting the video decoder to work. Also, the synchronization tasks (horizontal and vertical) required some clever solutions such as the interleaving between writing to and reading from the block RAMs (through the *fil_level* signal). Moreover, although we used polling for these types of synchronizations, a seemly better strategy would be to use interrupts. However, the overhead associated with context switching and machine status saving at each interrupt request would have to be careful analyzed if we wanted to change from polling to interrupts. Another extension would be to use RGB 888 instead of RGB 565. This, however, would require a DRAM controller being implemented inside the FPGA. The reason is because the SRAM does not have enough room for a reasonably large image where each pixel consumes 24 bits. The YUV to RGB hardware module, however, actually converts YUV to RGB 888.

APPENDIX

VGA module and its timing diagram:






Start
Initialize variables
conter = x = i = j = 0
xpos = -1, ypos = -1,
xdim = 0, ydim = 0
vga_sram[j,x] = color and j < HORZ_RES ?
Y
color_run++ j++
N
color_run > min_x_dim?
N
Y
constraint_y ++
color_run > x_dim_tmp?
Y
x_dim_tmp = color_run
N
constarint_y > min_y_dim?
N
Y
y_dim_tmp = constraint_y y_pos_tmp = i=constraint_y + 1
constraint_y > min_y_dim?
Y
y_dim = contraint_y y_pos = i
N
i < VERT_RES?
Y
color_run = 0 j = x i++
N
x_dim*y_dim < y_dim_tmp* x_dim_tmp?
Y
x_dim = x_dim_tmp y_dim = y_dim_tmp x_pos = x_pos_tmp y_pos = y_pos_tmp
N
x > HORZ_RES?
Y
i=0,x++, constraint_y = 0 color_run = 0 x_dim_tmp = HORZ_RES+1 y_dim_tmp = 0 x_pos_tmp = x
N
End

BLACK AND WHITE MODE

VHDL AND C FILES

OPB_VIDEODEC.VHD

```
library IEEE;
use IEEE.std_logic_1164.all;


entity opb_videodec is
  generic (
    C_OPB_AWIDTH : integer := 32;
    C_OPB_DWIDTH : integer := 32;
    C_BASEADDR   : std_logic_vector := X"0180_0000";   -- 512 positions of 32
    C_HIGHADDR   : std_logic_vector := X"0180_3FFF");  -- bits plus extra room.
                                                       -- Each 32 bits in the
                                                       -- block RAMs stores 4
                                                       -- pixels' luminance

  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;

    -- OPB signals
    OPB_ABus    : in std_logic_vector (31 downto 0);
    OPB_BE      : in std_logic_vector (3 downto 0);
    OPB_DBus    : in std_logic_vector (31 downto 0);
    OPB_RNW     : in std_logic;
    OPB_select  : in std_logic;
    OPB_seqAddr : in std_logic;

    -- Slave signals
    VIDEC_DBus    : out std_logic_vector (31 downto 0);
    VIDEC_errAck  : out std_logic;
    VIDEC_retry   : out std_logic;
    VIDEC_toutSup : out std_logic;
    VIDEC_xferAck : out std_logic;

    -- Coming from SAA7114H
    IPort       : in std_logic_vector (7 downto 0);
    HPort       : in std_logic_vector (7 downto 0);
    IDQ         : in std_logic;
    ICLK        : in std_logic;
    IPGV        : in std_logic;
    IPGH        : in std_logic;
    ITRI        : out std_logic;
    ITRDY       : out std_logic
  );
end opb_videodec;

architecture structural of opb_videodec is

-- Buffered version of the signals
-- with the same name in the entity
signal buf_iclk : std_logic;
signal buf_ipgh : std_logic;
signal buf_ipgv : std_logic;
signal buf_idq : std_logic;
signal buf_iport : std_logic_vector (7 downto 0);
signal buf_hport : std_logic_vector (7 downto 0);
```

```
signal buf_itri : std_logic;
signal buf_itrdy : std_logic;

-- Latched versions of the above buffered signals
signal latched_ipgh  : std_logic;
signal latched_ipgv  : std_logic;
signal latched_idq   : std_logic;
signal latched_iport : std_logic_vector (7 downto 0);
signal latched_hport : std_logic_vector (7 downto 0);

-- Signals used when reading from block
-- ram and filling status register
signal cs : std_logic;
signal ce : std_logic;
signal rnw : std_logic;
signal xfer : std_logic;

-- raddr(8 downto 0) is used to address the
-- block RAM. OPB_ABus(13) and OPB_ABus(12), which
-- correspond to raddr(11) and raddr(10), are
-- used to address the filling status register
signal raddr : std_logic_vector (11 downto 0);

-- Signals used by the filling level status
-- The video decoder interface sends a set
-- of signals indicating how much of the
-- current line it has already written into
-- the block RAMs (1/4, 1/2, 3/4 and 1)
-- Microblaze keeps polling this signal
signal filling_level : std_logic_vector(3 downto 0);

-- Count the number of lines being written by the video decoder
signal line_counter : std_logic_vector(15 downto 0);

-- Count the frame (Actually, it's the frame ID
signal frame_counter : std_logic_vector(1 downto 0);

-- Data coming from video decoder interface
signal data_from_decoder : std_logic_vector(15 downto 0);

-- Data bus and latched data bus
signal data_from_bram : std_logic_vector (31 downto 0);
signal data_bus_ce : std_logic_vector (31 downto 0);

-- Signals for the block ram state machine
signal q2, q1, q0 : std_logic;

-- Coming from video_decoder_intf, going to block_ram
signal intf_idq_out  : std_logic;
signal intf_iclk_out : std_logic;
signal waddr : std_logic_vector (10 downto 0);
signal luma_data : std_logic_vector (7 downto 0);

-- Dummy signals. Reserved for future enhancements
-- We currently not write from microblze (XIo_Out)
signal wdata : std_logic_vector (31 downto 0);
signal be    : std_logic_vector (3 downto 0);
```

```
component block_ram is
  port (
    waddr       : in std_logic_vector (10 downto 0);
    data_in     : in std_logic_vector (7 downto 0);
    raddr       : in std_logic_vector (8 downto 0);
    data_out    : out std_logic_vector (31 downto 0);
    idq         : in std_logic;
    iclk        : in std_logic;
    ipgh        : in std_logic;
    clock       : in std_logic;
    read_enable : in std_logic;
    reset       : in std_logic
  );
end component;

component video_decoder_intf is
  port (
    iport     : in std_logic_vector (7 downto 0);
    hport     : in std_logic_vector (7 downto 0);
    idq_in    : in std_logic;
    iclk_in   : in std_logic;
    ipgh      : in std_logic;
    ipgv      : in std_logic;
    data      : out std_logic_vector (15 downto 0);
    waddr     : out std_logic_vector (10 downto 0);
    idq_out   : out std_logic;
    iclk_out  : out std_logic;
    fil_level : out std_logic_vector(3 downto 0);
    line_count: out std_logic_vector(15 downto 0);
    frame_id  : out std_logic_vector(1 downto 0);
    reset     : in std_logic
  );
end component;

component IBUFG is
  port (
    I : in  std_logic;
    O : out std_logic);
end component;

component IBUF
  port (
    I : in  STD_ULOGIC;
    O : out STD_ULOGIC);
end component;

component OBUF
  port(
    O:      out std_ulogic;
    I:      in  std_ulogic
    );
end component;

component FD
  port (
    C : in std_logic;
```

```
    D : in std_logic;
    Q : out std_logic);
end component;

  -- Setting the iob attribute to "true" ensures that instances of these
  -- components are placed inside the I/O pads and are therefore very fast

attribute iob : string;
attribute iob of FD : component is "true";

begin

itrdy_buf : OBUF port map  (
  O => ITRDY,
  I => buf_itrdy
);

itri_buf : OBUF port map  (
  O => ITRI,
  I => buf_itri
);

vbuf : IBUFG port map (
  I => ICLK,
  O => buf_iclk
);

ipgh_pinbuf : IBUF port map (
  I => IPGH,
  O => buf_ipgh
);

ipgh_pinlatch : FD port map (
  C => buf_iclk,
  D => buf_ipgh,
  Q => latched_ipgh
);

ipgv_pinbuf : IBUF port map (
  I => IPGV,
  O => buf_ipgv
);

ipgv_pinlatch : FD port map (
  C => buf_iclk,
  D => buf_ipgv,
  Q => latched_ipgv
);

idq_pinbuf : IBUF port map (
  I => IDQ,
  O => buf_idq
);

idq_pinlatch : FD port map (
  C => buf_iclk,
  D => buf_idq,
```

```
  Q => latched_idq
);

databus : for i in 0 to 7 generate
  I_data_pad : IBUF port map (
      I => IPORT(i),
      O => buf_iport(i));

  I_data_ff : FD port map (
      C => buf_iclk,
      D => buf_iport(i),
      Q => latched_iport(i));

  H_data_pad : IBUF port map (
      I => HPORT(i),
      O => buf_hport (i));

  H_data_ff : FD port map (
      C => buf_iclk,
      D => buf_hport(i),
      Q => latched_hport(i));
end generate;

u1 : block_ram
port map
(
    waddr => waddr,
    data_in => luma_data,
    raddr => raddr(8 downto 0),
    data_out => data_from_bram,
    idq => intf_idq_out,
    iclk => intf_iclk_out,
    ipgh => latched_ipgh,
    clock => OPB_Clk,
    read_enable => '1',
    reset => OPB_Rst
);

u2 : video_decoder_intf
port map (
    iport => latched_iport,
    hport => latched_hport,
    idq_in => latched_idq,
    iclk_in => buf_iclk, -- For tests, use OPB_Clk
    ipgh => latched_ipgh,
    ipgv => latched_ipgv,
    data => data_from_decoder,
    waddr => waddr,
    idq_out  => intf_idq_out,
    iclk_out => intf_iclk_out,
    fil_level => filling_level,
    line_count => line_counter,
    frame_id => frame_counter,
    reset => OPB_Rst
);

-- Chip select for block RAM - port A of block RAMs is memory mapped
```

```
-- The binary number is X"0180" concatenated with binary "00"
cs <= OPB_select when OPB_ABus(31 downto 14) = "00000011000000000" else '0';


-- Latching read address. Used to address port A of block RAMs
process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk = '1' then
    if OPB_RST = '1' then
      raddr <= "000000000000";
    else
      raddr <= OPB_ABus(13 downto 2);
    end if;
  end if;
end process;


-- Latching RNW signal
process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk = '1' then
    if OPB_Rst = '1' then
      rnw <= '0';
    else
      rnw <= OPB_RNW;
    end if;
  end if;
end process;


-- Latching BE signal (byte enable). Dummy signal
process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk = '1' then
    if OPB_Rst = '1' then
      be <= "0000";
    else
      be <= OPB_BE;
    end if;
  end if;
end process;


-- The following process is dummy. It is used to
-- create a mux between this entity and OPB_DBus
process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk = '1' then
    if OPB_Rst = '1' then
      wdata <= X"0000_0000";
    else
      wdata <= OPB_DBus;
    end if;
  end if;
end process;
```

```
-- State machine for reading the block RAM
process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk='1' then
    q2 <= (not q2 and q1) or (q2 and not q1);
    q1 <= (cs and not q2 and not q1) or (q2 and not q1);
    q0 <= q2 and not q1;
  end if;
end process;

-- CE is data latch enable
ce <= q2 and not q1 and rnw;

-- Latch the data coming from the block RAM
-- or from the filling status register
-- at address 01803FFC
process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst='1' then
    data_bus_ce <= X"00000000";
  elsif OPB_Clk'event and OPB_Clk='1' then
    if ce='1' then
      if raddr(11)='1' and raddr(10)='1' then
        data_bus_ce <= "00000000000000000000000000000000" & filling_level;
      elsif raddr(11)='1' and raddr(10)='0' then
        data_bus_ce <= "00000000000000000000000000000000" & latched_ipgv;
      elsif raddr(11)='0' and raddr(10)='1' then
        data_bus_ce <= X"0000" & line_counter;
      elsif raddr(11)='0' and raddr(10)='0' and raddr(9)='1' then
        data_bus_ce <= "00000000000000000000000000000000" & frame_counter(0);
      else
        data_bus_ce <= data_from_bram;
      end if;
    else
      data_bus_ce <= X"00000000";
    end if;
  end if;
end process;

-- Connect luma bits from video decoder interface to block RAMs input data bus
luma_data <= data_from_decoder(15 downto 8);

-- XFER is transfer acknowledge
xfer <= q0;

-- Slave data bus
VIDEC_DBus(31 downto 0) <= data_bus_ce;

-- Tie unused signals to zero
VIDEC_errAck <= '0';
VIDEC_retry <= '0';
VIDEC_toutSup <= '0';
VIDEC_xferAck <= xfer;
buf_itri <= '1';
buf_itrdy <= '1';

end structural;
```

VIDEO_DECODER_INTF.VHD

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity video_decoder_intf is
  port (
    iport     : in std_logic_vector (7 downto 0);
    hport     : in std_logic_vector (7 downto 0);
    idq_in    : in std_logic;
    iclk_in   : in std_logic;
    ipgh      : in std_logic;
    ipgv      : in std_logic;
    data      : out std_logic_vector (15 downto 0);
    waddr     : out std_logic_vector (10 downto 0);
    idq_out   : out std_logic;
    iclk_out  : out std_logic;
    fil_level : out std_logic_vector(3 downto 0);
    line_count: out std_logic_vector(15 downto 0);
    frame_id  : out std_logic_vector(1 downto 0);
    reset     : in std_logic
  );
end video_decoder_intf;

architecture structural of video_decoder_intf is

signal active     : std_logic;
signal pix_count  : std_logic_vector (10 downto 0);
signal pixel_addr : std_logic_vector(10 downto 0);

signal line_counter : std_logic_vector(15 downto 0);
signal frame_counter : std_logic_vector(1 downto 0);

-- The following signals indicate how much of the
-- line was already written into  the  block  RAM
signal one_fourth     : std_logic;
signal half_line      : std_logic;
signal three_quarters : std_logic;
signal entire_line    : std_logic;
signal filling_level  : std_logic_vector(3 downto 0);

begin

-- pixel address - where to store valid pixels in the block RAMs
process (iclk_in, reset)
begin
 if reset='1' then
    pixel_addr <= "00000000000";
 elsif iclk_in'event and iclk_in='1' then
   if ipgh='0' then
     pixel_addr <= "00000000000";
   elsif idq_in = '1' and active = '1' then
     pixel_addr <= pixel_addr + 1;
   end if;
 end if;
end process;
```

```
-- count the actual data coming from iport and hport.
-- Some data is control (FF, 00 , 00 , SAV business)
-- Reset the counter whenever ipgh is zero
process (iclk_in, reset)
begin
  if reset='1' then
    pix_count <= "00000000000";
  elsif iclk_in'event and iclk_in='1' then
    if idq_in='1' then
      if ipgh='0' then
        pix_count <= "00000000000";
      else
        pix_count <= pix_count + 1;
      end if;
    end if;
  end if;
end process;

-- count the number of lines
process (iclk_in, reset)
begin
  if reset='1' then
    line_counter <= X"0000";
  elsif iclk_in'event and iclk_in='1' then
    if ipgv='0' then
      line_counter <= X"0000";
    elsif ipgh='1' and pix_count=719 then
      line_counter <= line_counter + 1;
    end if;
  end if;
end process;

-- give the frame ID
process (iclk_in, reset)
begin
  if reset='1' then
    frame_counter <= "00";
  elsif iclk_in'event and iclk_in='1' then
    if line_counter = 239 and pix_count=719 then
      frame_counter <= frame_counter+1;
    end if;
  end if;
end process;

-- Active means we are within
-- the horizontal line active video
process (iclk_in)
begin
 if iclk_in'event and iclk_in='1' then
   if ipgh='0' then
     active <= '0';
   elsif pix_count = 1 then
     active <= '1';
   elsif pix_count=720 then
     active <= '0';
   end if;
```

```
 end if;
end process;

-- Set output signals according to where
-- in the current line the video decoder
-- is writing the block RAM
process (iclk_in, reset)
begin
 if reset='1' then
   one_fourth <= '0';
 elsif iclk_in'event and iclk_in='1' then
   if pix_count=0 then
     one_fourth <= '0';
   elsif pix_count=161 then
     one_fourth <= '1';
   end if;
 end if;
end process;

process (iclk_in, reset)
begin
 if reset='1' then
   half_line <= '0';
 elsif iclk_in'event and iclk_in='1' then
   if pix_count=0 then
     half_line <= '0';
   elsif pix_count=321 then
     half_line <= '1';
   end if;
 end if;
end process;

process (iclk_in, reset)
begin
 if reset='1' then
   three_quarters <= '0';
 elsif iclk_in'event and iclk_in='1' then
   if pix_count=0 then
     three_quarters <= '0';
   elsif pix_count=481 then
     three_quarters <= '1';
   end if;
 end if;
end process;

process (iclk_in, reset)
begin
 if reset='1' then
   entire_line <= '0';
 elsif iclk_in'event and iclk_in='1' then
   if pix_count=0 then
     entire_line <= '0';
   elsif pix_count=641 then
     entire_line <= '1';
   end if;
 end if;
end process;
```

```
filling_level(0) <= one_fourth;
filling_level(1) <= half_line;
filling_level(2) <= three_quarters;
filling_level(3) <= entire_line;


-- Output signals of this entity

fil_level <= filling_level;
line_count <= line_counter;
frame_id <= frame_counter;

data(15 downto 8) <= iport;
data(7 downto 0)  <= hport;

waddr <= pixel_addr;
iclk_out <= iclk_in;
idq_out <= idq_in;

end structural;



-- Test generator
-- signal pixel_data : std_logic_vector(15 downto 0);

-- begin

-- -- pixel data
-- process (iclk_in, reset)
-- begin
--   if reset='1' then
--     pixel_data <= X"00FF";
--   elsif iclk_in'event and iclk_in = '1' then
--       pixel_data <= pixel_data + X"100";
--   end if;
-- end process;

-- -- pixel address - where to store in the block RAMs
-- process (iclk_in, reset)
-- begin
--   if reset='1' then
--      pixel_addr <= "00000000000";
--   elsif iclk_in'event and iclk_in='1' then
--      pixel_addr <= pixel_addr + 1;
--   end if;
-- end process;


-- data <= pixel_data;
-- waddr <= pixel_addr;
-- iclk_out <= iclk_in;
-- idq_out <= '1';

-- end structural;
```

BLOCK_RAM.VHD

```
library IEEE;
use IEEE.std_logic_1164.all;

-- Four RAMB4_S8_S8 components instantiated.
-- Each one stores 8 bits of information (luma)
-- on each memory cell. Block 0 stores pixels
-- 0,4,8, etc. Block 1 stores pixels 1, 5, 9, etc,
-- Block 2 stores pixels 2, 6, 10, etc. and
-- Block 3 stores pixels 3, 7, 11, etc.
-- and so on.
entity block_ram is
  port (
    -- Address generated by video decoder intf module
    -- (video_decoder_intf.vhd). All block-RAMs see
    -- the same 9 *upper* bits. The remaining 2 *lower*
    -- bits are used to choose which block to store.
    waddr : in std_logic_vector (10 downto 0);

    -- Luminance data coming from the video decoder
    -- The video decoder is actually being configured
    -- to transmit 16-bit data (upper bits are luma,
    -- lower bits are chroma). However, the chroma
    -- bits are just being disconsidered as of now.
    data_in : in std_logic_vector (7 downto 0);

    -- Read address. Generated by microblaze every
    -- time one executes XIO_In32. Microblaze reads
    -- four pixels at a time: pixel "i" from block
    -- 0, pixel "i+1" from block 1, pixel "i+2"
    -- from block 2 and pixel "i+3" from block 3.
    -- That's why the *lower* bits of addr are used.
    raddr : in std_logic_vector (8 downto 0);

    -- Data going to microblaze. The 32 bits read
    -- correspond to 4 pixels, each one coming
    -- from a specific block RAM.
    data_out : out std_logic_vector (31 downto 0);

    -- IDQ is '1' when valid data is
    -- coming from video decoder
    idq : in std_logic;

    -- clock for port B is ICLK
    -- from video decoder
    iclk : in std_logic;

    -- From the video decoder
    ipgh : in std_logic;

    -- clock for port A is
    -- clk from CPU
    clock : in std_logic;

    -- Read enable
    read_enable : in std_logic;
```

```
    -- Reset
    reset : in std_logic
  );
end block_ram;

architecture structural of block_ram is

-- Dual-port block RAM used for storing data coming from video decoder
-- Port B is written by the video decoder intf, Port A is read by CPU.
-- See  "http://www.xilinx.com/bvdocs/appnotes/xapp173.pdf"
component RAMB4_S8_S8
  generic (
    INIT_00, INIT_01, INIT_02, INIT_03, INIT_04, INIT_05,
    INIT_06, INIT_07, INIT_08, INIT_09, INIT_0a, INIT_0b,
    INIT_0c, INIT_0d, INIT_0e, INIT_0f: bit_vector(255 downto 0)
    :=X"0000000000000000000000000000000000000000000000000000000000000000"
  );
  port  (
    DIA,DIB : in STD_LOGIC_VECTOR (7 downto 0);
    ENA,ENB : in STD_logic;
    WEA,WEB : in STD_logic;
    RSTA,RSTB     : in STD_logic;
    CLKA,CLKB     : in STD_logic;
    ADDRA,ADDRB : in STD_LOGIC_VECTOR (8 downto 0);
    DOA,DOB : out STD_LOGIC_VECTOR (7 downto 0)
  );
end component;

-- i_clock is ICLK from video decoder
-- opb_clock is opb_clk from OPB bus
signal i_clock : std_logic;
signal opb_clock : std_logic;

-- Read enable
signal r_en : std_logic;

-- Reset
signal rst : std_logic;

-- Shared address bus for all 4 block RAMs
signal addr_a : std_logic_vector (8 downto 0);
signal addr_b : std_logic_vector (8 downto 0);

-- Enable signals for distinct blocks
signal enb0, enb1, enb2, enb3 : std_logic;

-- Data coming from video decoder interface to B ports
signal data_in_signal : std_logic_vector (7 downto 0);

-- Data going to OPB Bus from A ports
signal data_out_a0 : std_logic_vector (7 downto 0);
signal data_out_a1 : std_logic_vector (7 downto 0);
signal data_out_a2 : std_logic_vector (7 downto 0);
signal data_out_a3 : std_logic_vector (7 downto 0);


begin
```

```
block_0: RAMB4_S8_S8    -- 512 words of 8 bits
port map
(
  DIA => X"00", DIB => data_in_signal,
  ENA => r_en, ENB => '1',
  WEA => '0', WEB => enb0,
  RSTA => rst, RSTB => rst,
  CLKA => opb_clock, CLKB => i_clock,
  ADDRA => addr_a, ADDRB => addr_b,
  DOA => data_out_a0, DOB => open
);


block_1: RAMB4_S8_S8 -- 512 words of 8 bits
port map
(
  DIA => X"00", DIB => data_in_signal,
  ENA => r_en, ENB => '1',
  WEA => '0', WEB => enb1,
  RSTA => rst, RSTB => rst,
  CLKA => opb_clock, CLKB => i_clock,
  ADDRA => addr_a, ADDRB => addr_b,
  DOA => data_out_a1, DOB => open
);


block_2: RAMB4_S8_S8 -- 512 words of 8 bits
port map
(
  DIA => X"00", DIB => data_in_signal,
  ENA => r_en, ENB => '1',
  WEA => '0', WEB => enb2,
  RSTA => rst, RSTB => rst,
  CLKA => opb_clock, CLKB => i_clock,
  ADDRA => addr_a, ADDRB => addr_b,
  DOA => data_out_a2, DOB => open
);


block_3: RAMB4_S8_S8 -- 512 words of 8 bits
port map
(
  DIA => X"00", DIB => data_in_signal,
  ENA => r_en, ENB => '1',
  WEA => '0', WEB => enb3,
  RSTA => rst, RSTB => rst,
  CLKA => opb_clock, CLKB => i_clock,
  ADDRA => addr_a, ADDRB => addr_b,
  DOA => data_out_a3, DOB => open
);

-- Uncomment the following lines if you don't want to skip pixels
-- Then comment the four lines indicating Y0, Y2, Y4, Y6 below.
--enb0 <= idq and ipgh and not waddr(1) and not waddr(0);
--enb1 <= idq and ipgh and     waddr(1) and not waddr(0);
--enb2 <= idq and ipgh and not waddr(1) and not waddr(0);
```

```
--enb3 <= idq and ipgh and     waddr(1) and not waddr(0);

enb0 <= idq and ipgh and not waddr(2) and not waddr(1) and not waddr(0); -- Y0
enb1 <= idq and ipgh and not waddr(2) and     waddr(1) and not waddr(0);  -- Y2
enb2 <= idq and ipgh and     waddr(2) and not waddr(1) and not waddr(0);  -- Y4
enb3 <= idq and ipgh and     waddr(2) and     waddr(1) and not waddr(0);  -- Y6

-- Data out merger
data_out(31 downto 24) <= data_out_a0;
data_out(23 downto 16) <= data_out_a1;
data_out(15 downto 8)  <= data_out_a2;
data_out(7 downto 0)   <= data_out_a3;

-- Data in
data_in_signal <= data_in;


-- Actual bits addressing block RAMs, port A
addr_a <= raddr;


-- Uncomment the following line if you don't want to skip pixels
-- addr_b <= waddr(10 downto 2);

-- Actual bits addressing block RAMs, port B
addr_b <= "0" & waddr(10 downto 3);

-- Connect clocks and reset
i_clock <= iclk;
opb_clock <= clock;
rst <= reset;

-- Read enable
r_en <= read_enable;

end structural;
```

OPB_I2CCONTROLLER.VHD

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


entity opb_i2ccontroller is  -- USER --
  generic
  (
    C_OPB_AWIDTH       : integer          := 32;
    C_OPB_DWIDTH       : integer          := 32;
    C_BASEADDR         : std_logic_vector := X"FEFF0200";
    C_HIGHADDR         : std_logic_vector := X"FEFF02FF");

  port
  (
    --Required OPB bus ports, do not add to or delete
    OPB_ABus     : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE       : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_Clk      : in  std_logic;
    OPB_DBus     : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW      : in  std_logic;
    OPB_Rst      : in  std_logic;
    OPB_select   : in  std_logic;
    OPB_seqAddr  : in  std_logic;
    VID_I2C_DBus     : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    VID_I2C_errAck   : out std_logic;
    VID_I2C_retry    : out std_logic;
    VID_I2C_toutSup  : out std_logic;
    VID_I2C_xferAck  : out std_logic;

    -- USER --
    VID_I2C_SCL      : inout std_logic;
    VID_I2C_SDA      : inout std_logic
  );

end entity opb_i2ccontroller; --USER--

------------------------------------------------------------------------------
-
-- architecture
------------------------------------------------------------------------------
-

architecture imp of opb_i2ccontroller is --USER--

component IOBUF_F_12
 port (
    O : out STD_ULOGIC;
    IO : inout STD_ULOGIC;
    I : in STD_ULOGIC;
    T : in STD_ULOGIC);
end component;

signal wdata : std_logic_vector(0 to 7);
signal rdata : std_logic_vector(0 to 7);
signal rnw  : std_logic;
signal cs, xfer : std_logic;
```

```
signal q0,q1 : std_logic;
signal i2c_din : std_logic;

begin


sda_pad : IOBUF_F_12 port map (
  I => wdata(0),
  IO => VID_I2C_SDA,
  O =>  i2c_din,
  T => wdata(1)
);

scl_pad : IOBUF_F_12 port map (
  I => wdata(2),
  IO => VID_I2C_SCL,
  O => open,
  T => wdata(3)
);

-- Chip select, memory mapped. XIoOut8 for selecting the I2C controller
process (OPB_select, OPB_ABus)
begin

  if(OPB_select='1' and OPB_ABus(0 to 23)=C_BASEADDR(0 to 23)) then
    cs <= '1'; else
    cs <= '0';
  end if;
end process;


-- I2C Bus SDA interconnection
process (OPB_Clk,OPB_Rst)
begin
  if (OPB_Rst='1') then
    wdata <= "11111111";
    rdata <= "00000000";

  elsif OPB_Clk'event and OPB_Clk = '1' then
    rnw <= OPB_RNW;

    if (q1 = '0' and q0 = '1' and rnw='0') then
       wdata <= OPB_DBus(0 to 7);
    end if;

    if (q1='1') then
       rdata <= "00000000";
    elsif (q1='0' and q0='1' and rnw = '1') then
       rdata <= i2c_din & "0000000";
    end if;

  end if;
end process;


process (OPB_Clk,OPB_Rst)
begin
```

```
  if (OPB_Rst = '1') then
    q0 <= '0';
    q1 <= '0';
  elsif OPB_Clk'event and OPB_Clk='1' then
    q1 <= not q1 and q0;
    q0 <= not q1 and not q0 and cs;
  end if;
end process;

xfer <= q1;

VID_I2C_xferAck <= xfer;

VID_I2C_DBus <= rdata & X"000000";

VID_I2C_errAck <= '0';

VID_I2C_retry <= '0';

VID_I2C_toutSup <= '0';

end architecture imp;
```

WRITE_VIDEO.C

```
#include "xbasic_types.h"
#include "xio.h"

#define W 640
#define VGA_START 0x00800000
#define BRAM_START 0x01800000

// Transfer a section of "line" starting at pixel //
// "start" and ending at  pixel  "end"  from  the //
// block RAMs to the SRAM.                        //
void write_video(int start, int end, int line)
{
  int nPixs;
  Xuint32 luma_4pixels;
  Xuint32 bram_addr;
  Xuint32 vga_addr;

  nPixs = (end - start);
  vga_addr = VGA_START + (start>>1) + W*line;
  bram_addr = BRAM_START + (start>>1);

  while (nPixs > 0)
  {
    luma_4pixels = XIo_In32(bram_addr+0);
    XIo_Out32(vga_addr+0, luma_4pixels);

    luma_4pixels = XIo_In32(bram_addr+4);
    XIo_Out32(vga_addr+4, luma_4pixels);

    luma_4pixels = XIo_In32(bram_addr+8);
    XIo_Out32(vga_addr+8, luma_4pixels);

    luma_4pixels = XIo_In32(bram_addr+12);
    XIo_Out32(vga_addr+12, luma_4pixels);

    bram_addr += 16;
    vga_addr += 16;

    // Skip pixels //
    nPixs -=32;

    // If we lose vertical synchronism in the meantime //
    // then break from the "while" and return          //
    if (!XIo_In32(0x01802FFC))
      break;
  }
}
```

CHAR_PRINTING.C

```
#include "xbasic_types.h"
#include "xio.h"
#include "font_8x8.h"

#define W 640
#define H 480
#define VGA_START 0x00800000

#define RED 0xE0
#define GREEN 0x1C
#define BLUE 0x03

void
draw_char (int x, int y, unsigned char ch)
{
  int row, col;
  short int row_template;

  // "Times 8" used  to  index  the //
  // array declared in "font_8x8.h" //
  int init_pos = ch * 8;

  // Print the 8 rows of the character //
  // in the outermost loop             //
  for (row = y; row < y + 8; row++)
  {
    // Read the character into a  short //
    // int variable to be able to shift //
    row_template = fontdata_8x8[row - y + init_pos];

    // Print each pixel that is 1 or 0 //
    // in  the  character's  template  //
    for (col = x; col < x + 8; col++)
    {
      // The varying  amount  of  shifting //
      // at each iteration takes  care  of //
      // analyzing the right bit at a time //
      if ((row_template << col-x) & 0x80)
      {
      XIo_Out8(VGA_START + col + 640*row, RED|GREEN|BLUE);
      }
      // To take  care  of  cleaning  something //
      // already written (write the background) //
      else
      {
        XIo_Out8(VGA_START + col + 640*row, 0);
      }
    }
  }
}

void draw_string(int x, int y, char *s)
{
  while(*s) draw_char(x+=8, y, *s++);
}
```

```
// This function can be used for debuging purposes //
void draw_hex(int x, int y, int n)
{
  int i, d;
  char c;
  for(i=0; i<8;i++){
    d=(n>>28)&0x0F;
    c = d>9 ? d-10+'A' : d+'0';
    draw_char(x+=8, y, c);
    n<<=4;
  }
}
```

TRACK_OBJECT.C

```
#include "xbasic_types.h"
#include "xio.h"

#define HORZ_RES 320
#define VERT_RES 240
#define W 320
#define H 240
#define MIN_X_DIM 2
#define MIN_Y_DIM 2
#define TOLERANCE 10
#define color_match 0xC0
#define VGA_START 0x00800000

/*
   The way this search algorithm works is quite simple. It first traverses
   through the video_sram, looking for a particular "color_match" in a
   sequence. From the current starting position in each line, it calculates
   the number of consecutive instances of "color_match" it can find and stores
   it to a 1-D array (each element corresponding to each horizontal line.

   Then the second segment of the code actually traverses ONLY through this
   1-D array created and looks for a consistent run of values that are more
   than the "MIN_X_DIM". Then once this traversal is complete, it now knows
   how many acceptable values of "color_run" occur in a sequence. Then it
   stores that value temporarily along with the corresponding Y value the
   sequence started at.

   Finally, the values of x_dim_tmp and y_dim_tmp are compared with the current
   values of x_dim and y_dim. Currently the decision is based on area, and if
   need be to speed up the program, it can always be changed to a simple comparison
   between x_dim, y_dim and X_dim_tmp and y_dim_tmp.
*/

void track_object()
{
  int line, pix;
  int color_run;
  int count;
  int i, j, x;
  int upperbound, lowerbound;
  int constraint_y;
  int x_dim, y_dim;
  int x_pos, y_pos;
  int y_pos_tmp, x_pos_tmp;
  int x_dim_tmp, y_dim_tmp;
  Xuint8 video_byte;
  int flag_x_fail = 0;

  count = 0;
  i = 0;
  j = 0;
  x_dim=0;
  y_dim=0;
  constraint_y = 0;
  upperbound = color_match + TOLERANCE;
```

```
lowerbound = color_match - TOLERANCE;

for(x = 0; x < HORZ_RES; x++)
{
  // Traversing through video_sram to get the values of
  // matches for a particular color within a "tolerance"

  // The program can be altered very easily to change the
  // search color on the fly looking for any number of colors
  // by changing the color_match variable.


  // setting the current x_dim_tmp //
  // to a relative infinity        //
  x_dim_tmp = HORZ_RES + 1;

  constraint_y = 0;
  y_dim_tmp = 0;
  x_pos_tmp = x;

  for (i = 0; i < VERT_RES ; i++)
  {
    color_run = 0;
    flag_x_fail = 0;
    /*
     Inner loop that checks values from the current x
     position till the end of the video_sram array

     This is the ONLY place where video_sram is being accessed.
    */
    for(j = x; ((j < HORZ_RES)&&(flag_x_fail == 0)); j++)
    {

    // Checks to see if the current videosram value is out of range
    // if the mismatch was found then the fail_x_flag is set as we
    // are not interested in any of the values that would occur after
    // the first anomalitiy.

    video_byte = XIo_In8(VGA_START + j + 640*i);
    if((video_byte > upperbound ) || (video_byte < lowerbound ))
    {
      flag_x_fail =1;
      break;
    }
    else // else adds one to it's counter
    {
      color_run++;
    }
    }

    // Here is where the 2-d recognition comes into play. This is
    // the second part of the algorithm where it looks for the
    // consistency in consistency of color values per se.

    // If it finds a decent consistency (i.e. one that satisfies
    // both the x_dim and y_dim constraints) then it will store
    // the x coordinate in x_pos_tmp and y coordinate in y_pos_tmp
```

```
    // and the corresponding x and y dimensions in x_dim_temp and
    // y_dim_temp respectivley.

    // We have just encountered a place where we have seen that
    // the color_run value is more than the minimum x dimension
    // so we have to first increment the constraint_y value and
    // possibly update the y_pos_tmp value to the index of the
    // start of the sequence.
    //
    // Also in the current color_run value was found to be lower
    // than the current x_dim_tmp, then you need to reassign it
    // as the x dimension of the rectangle will be decided by the
    // lowest value of color_run in acceptable range of x_dim value

    if (color_run >= MIN_X_DIM)
    {
    constraint_y++;
    if(color_run < x_dim_tmp)
      x_dim_tmp = color_run;
    if (constraint_y > y_dim)
    {
      y_dim_tmp = constraint_y;
      y_pos_tmp = i - constraint_y + 1;
    }
    }

    if (color_run < MIN_X_DIM)
    {
    if (constraint_y > y_dim)
    {
      y_dim = constraint_y;
      y_pos = i;
    }
    constraint_y = 0;
    }
  }



  // Now that the program has the values of the last best rectangle,
  // it compares it with the x and y dimensions of the one that was
  // just found (if any).

  // Currently the reassignment is made if the new rectangle has a
  // greater area than the older one. But for optimization, this can
  // be tossed for simpler decision criteia.

  if ((x_dim_tmp * y_dim_tmp) > (x_dim * y_dim)) {
    x_dim = x_dim_tmp;
    y_dim = y_dim_tmp;
    x_pos = x_pos_tmp;
    y_pos = y_pos_tmp;
  }


} // main for loop
```

```
  // These are the values that are read //
  // out of the image and may be written //
  // to any part of the memory.          //

  for (j = x_pos-2; j < x_pos+ 2 + x_dim; j+=2)
  {
    XIo_Out8(VGA_START + j + 640*(y_pos) ,0x95);
    XIo_Out8(VGA_START + j + 640*(y_pos+y_dim) ,0x95);
  }

  for (i = y_pos-2; i < y_pos+ 2 + y_dim; i+=2)
  {
    XIo_Out8(VGA_START + x_pos + 640*(i) ,0x95);
    XIo_Out8(VGA_START + x_pos+x_dim + 640*(i) ,0x95);
  }

}
```

MAIN.C

```
#include "xbasic_types.h"
#include "xio.h"

#define W 640
#define H 480
#define VGA_START 0x00800000
#define RED 0xE0
#define GREEN 0x1C
#define BLUE 0x03

extern void write_video(int start, int end, int line);
extern void track_object();

// Register addresses for SAA7114H configuration
unsigned char registers [] = {

  // Video decoder "generic" registers //
  0x01, 0x08, // Recommended setting
  0x02, 0xE9, // Analog input control 1 and input selection
  0x03, 0x10, // Analog input control 2
  0x04, 0x90, // Analog input control 3
  0x05, 0x90, // Analog input control 4
  0x06, 0xEB, // Horizontal Sync Start (delay)
  0x07, 0xE0, // Horizontal Sync Stop (delay)
  0x08, 0x59, // Sync control
  0x09, 0x40, // Luminance control
  0x0A, 0x80,
  0x0B, 0x44,
  0x0C, 0x40,
  0x0D, 0x00,
  0x0E, 0x89,
  0x0F, 0x2A, // Chrominance gain
  0x10, 0x0E, // Chrominance control
  0x11, 0x00,
  0x12, 0x46, // RT signal control
  0x13, 0x00,
  0x14, 0x00,
  0x15, 0x11,
  0x16, 0xFE,
  0x17, 0x40,
  0x18, 0x40,
  0x19, 0x80,
  0x1A, 0x00,
  0x1B, 0x00,
  0x1C, 0x00,
  0x1D, 0x00,
  0x1E, 0x00,
  0x30, 0x08, // Audio clock stuff
  0x31, 0x08, // Audio clock stuff
  0x32, 0x02,
  0x33, 0x00,
  0x34, 0xCD,
  0x35, 0xCC,
  0x36, 0x3A,
```

```
0x37, 0x00,
0x38, 0x03,
0x39, 0x10,
0x3A, 0x00,
0x3B, 0x00,
0x3C, 0x00,
0x3D, 0x00,
0x3E, 0x00,
0x3F, 0x00,
0x40, 0x40,
0x41, 0xFF,
0x42, 0xFF,
0x43, 0xFF,
0x44, 0xFF,
0x45, 0xFF,
0x46, 0xFF,
0x47, 0xFF,
0x48, 0xFF,
0x49, 0xFF,
0x4A, 0xFF,
0x4B, 0xFF,
0x4C, 0xFF,
0x4D, 0xFF,
0x4E, 0xFF,
0x4F, 0xFF,
0x50, 0xFF,
0x51, 0xFF,
0x52, 0xFF,
0x53, 0xFF,
0x54, 0xFF,
0x55, 0xFF,
0x56, 0xFF,
0x57, 0xFF,
0x58, 0x40,
0x59, 0x47,
0x5A, 0x06,
0x5B, 0x03,
0x5C, 0x00,
0x5D, 0x3E,
0x5E, 0x00,
0x5F, 0x00,
0x80, 0x10, // Only Task A: 0x10 ;  Both tasks: 0x30.
0x83, 0x01,
0x84, 0xA0,
0x85, 0x10,
0x86, 0x45,
0x87, 0x01,
0x88, 0xF0,

// Task A Registers //
0x90, 0x00,
0x91, 0x08,
0x92, 0x10,
0x93, 0xC0,
0x94, 0x10,
0x95, 0x00,
0x96, 0xD0,
```

```
0x97, 0x02,
0x98, 0x0A,
0x99, 0x00,
0x9A, 0xF2,
0x9B, 0x00,
0x9C, 0xD0, // Horizontal output window size upper bits \ 0xD002 = 720
0x9D, 0x02, // Horizontal output window size lower bits /          by
0x9E, 0xF0, // Vertical output window size upper bits  \  0xF000 = 240
0x9F, 0x00, // Vertical output window size lower bits  /
0xA0, 0x01,
0xA1, 0x00,
0xA2, 0x00,
0xA4, 0x80,
0xA5, 0x40,
0xA6, 0x40,
0xA8, 0x00,
0xA9, 0x04,
0xAA, 0x00,
0xAC, 0x00,
0xAD, 0x02,
0xAE, 0x00,
0xB0, 0x00,
0xB1, 0x04,
0xB2, 0x00,
0xB3, 0x04,
0xB4, 0x00,
0xB8, 0x00,
0xB9, 0x00,
0xBA, 0x00,
0xBB, 0x00,
0xBC, 0x00,
0xBD, 0x00,
0xBE, 0x00,
0xBF, 0x00,

/*
// Task B Registers - Not being used as of now //
0xC0, 0x08,
0xC1, 0x08,
0xC2, 0x10,
0xC3, 0xC0,
0xC4, 0x10,
0xC5, 0x00,
0xC6, 0xD0,
0xC7, 0x02,
0xC8, 0x0A,
0xC9, 0x00,
0xCA, 0xF2,
0xCB, 0x00,
0xCC, 0xD0,
0xCD, 0x02,
0xCE, 0xF0,
0xCF, 0x00,
0xD0, 0x01,
0xD1, 0x00,
0xD2, 0x00,
0xD4, 0x80,
```

```
  0xD5, 0x40,
  0xD6, 0x40,
  0xD8, 0x00,
  0xD9, 0x04,
  0xDA, 0x00,
  0xDC, 0x00,
  0xDD, 0x02,
  0xDE, 0x00,
  0xE0, 0x00,
  0xE1, 0x04,
  0xE2, 0x00,
  0xE3, 0x04,
  0xE4, 0x00,
  0xE8, 0x00,
  0xE9, 0x00,
  0xEA, 0x00,
  0xEB, 0x00,
  0xEC, 0x00,
  0xED, 0x00,
  0xEE, 0x00,
  0xEF, 0x00,*/

  // Reset sequence. Extremelly needed!!
  // Do not comment the following out! //
  0x88, 0xD8,
  0x88, 0xF8,
  0xFF, 0xFF,};

// Witness variable //
int w = 0xFF;

// Provide a delay between signal toggling //
void i2c_delay()
{
    int i;
    for (i = 0; i < 1000; i++);
}

// Write "level" to SCL //
void SCLw(int level)
{
    if (level == 0)
      w &= 0xDF;
    else
      w |= 0x2F;

    // Assert the clock on SCL //
    // according to level      //
    XIo_Out8(0xFEFF0200, w);
    i2c_delay();
}

// Write "level" to SDA //
void SDAw(int level)
{
    if (level == 0)
      w &= 0x7F;
```

```
    else
      w |= 0x8F;

    // Assert the clock on SDA //
    // according to level      //
    XIo_Out8(0xFEFF0200, w);
    i2c_delay();
}

// Read from SDA //
int SDAr()
{
    int MSB = XIo_In8(0xFEFF0200);

    MSB = MSB >> 7;
    MSB &= 1;

    i2c_delay();
    return MSB;
}

// Tristate for SDA //
void SDAt(int rnw)
{
    if (rnw == 0)
      w &= 0xBF;
    else
      w |= 0x4F;

    // Assert the clock on SDA //
    // according to level      //
    XIo_Out8(0xFEFF0200, w);
    i2c_delay();
}

// Tristate for SCL //
void SCLt(int rnw)
{
    if (rnw == 0)
      w &= 0xEF;
    else
      w |= 0x1F;

    // Assert the clock on SDA //
    // according to level      //
    XIo_Out8(0xFEFF0200, w);
    i2c_delay();
}

// Send the start sequence
void send_start( void )
{
    SCLt(0);
    SDAt(0);
    SCLw(1);
    SDAw(0);
    SCLw(0);
```

```
}

// Send the restart sequence
// Needed for read register
void re_start( void )                /* This function must be entered with SDA High
*/
{
    SCLw(1);
    SDAw(0);
    SCLw(0);
}


// Send stop sequence
void send_stop( void )
{
    SCLw(0);
    SDAw(0);
    SCLw(1);
    SDAw(1); /* Should leave with both lines high to indicate finish */
}

// Check acknowledge
int check_ack(void)
{
    int theresult;
    SDAt(1);
    SCLw(1);
    theresult=SDAr();
    SCLw(0);
    SDAw(1); /* Set the output before it becomes active to eliminate spike */
    SDAt(0);
    return theresult;
}

// Send one bit
void send_bit(int x)
{
    x = x & 1;
    SDAw(x);
    SCLw(1);
    SCLw(0);
}

// Send an entire bit
void send_byte(int byte)
{
    int i;
    for (i = 7; i >= 0; i--)
    {
      send_bit(byte >> i);
    }
}

// Read a register from the video decoder
int read_register( int sub_address )
{
```

```
    int id, input = 0;

    send_start();

    // Write slave address for SAA7114H is 43H //
    send_byte(0x42);

    check_ack();

    // Send the subaddress //
    send_byte(sub_address);

    check_ack();

    re_start();

    // Read address //
    send_byte(0x43);

    check_ack();

    SDAt(1);
    for( id=8 ; id>0 ; id=id-1 )
    {
        input=input<<1;
        SCLw(1);
        input=input|SDAr();
        SCLw(0);
    }
    SDAw(1);   /* Set the output prior to enable to eliminate spike and make
                  compatible with Restart */
    SDAt(0);
    SCLw(1);
    SCLw(0);
    send_stop();
    return input;
}

// Write a register into the video decoder
void write_register(int sub_address, int data)
{
    int i;
    // Start conditions //
    send_start();

    for (i = 0; i < 5; i++)
     i2c_delay();

    // Write slave address for SAA7114H is 42H //
    send_byte(0x42);
    check_ack();
    send_byte(sub_address);
    check_ack();
    send_byte(data);
    check_ack();
    send_stop();
}
```

```
void read_one_field()
{
  int line;
  int start, end;
  int line_section;
  Xuint32 current_level;

  line = -1;

  while (1)
  {
    line = line + 1;

    // This variable indicates how much of //
    // the current line has  been  already //
    // written into the block RAMs         //
    current_level = 0x0001;

    for (line_section = 0; line_section < 4; line_section++)
    {
      // Wait for the current line to  be  1/4,  1/2,  3/4 //
      // and full filled. The while below executes 4 times //
      while (!(XIo_In32(0x01803FFC) & current_level))
      {
      // If in the meantime we lose vertical //
      // synchronism, then break             //
      if (!XIo_In32(0x01802FFC))
        break;
      }

      if (current_level == 0x01) {
      start = 0;
      end = 160;
      }
      else if (current_level == 0x02) {
      start = 160;
      end = 320;
      }
      else if (current_level == 0x04) {
      start = 320;
      end = 480;
      }
      else if (current_level == 0x08) {
      start = 480;
      end = 640;
      }

      if (!XIo_In32(0x01802FFC))
      break;

      write_video(start, end, line);
      current_level = current_level << 1;

      if (!XIo_In32(0x01802FFC))
      break;
```

```
    }
    if (!XIo_In32(0x01802FFC))
      break;
  }
}


int main()
{
  int i;

  print("Hello World!\r\n");
  microblaze_enable_icache();

  // Start the bus  protocol  by  sending //
  // a stop handshaking (SDA=1 and SCL=1) //
  send_stop();
  print("Configuring video decoder...");

  i = 0;

  // Configure the video decoder SAA7114H //
  while (registers[i] != 0xFF) {
    write_register (registers[i], registers[i+1]);
    i+=2;
  }
  print("Video decoder configured!\r\n");

  // Clear screen //
  for (i = 0; i < H*W; i++)
    XIo_Out8(VGA_START + i, 0);

  // Wait for a little bit
  for (i=0; i<10000;i++);

  while (1)
  {
    // Wait for the vertical synchronism
    while ((XIo_In32(0x01802FFC)));
    while (!(XIo_In32(0x01802FFC)));

    if (XIo_In32(0x018008FC))
    {
      while ((XIo_In32(0x01802FFC)));
      while (!(XIo_In32(0x01802FFC)));
    }

    read_one_field();
    track_object();
  }

  print("Goodbye\r\n");
  return 0;
}
```

VGA.VHD – parts of this file were modified

```
----------------------------------------------------------------------------
-
--
-- VGA video generator
--
-- Uses the vga_timing module to generate hsync etc.
-- Massages the RAM address and requests cycles from the memory controller
-- to generate video using one byte per pixel
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
----------------------------------------------------------------------------
-

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity vga is
  port (
    clk            : in std_logic;
    pix_clk        : in std_logic;
    rst            : in std_logic;
    video_data     : in std_logic_vector(15 downto 0);
    video_addr     : out std_logic_vector(19 downto 0);
    video_req      : out std_logic;
    VIDOUT_CLK     : out std_logic;
    VIDOUT_RCR     : out std_logic_vector(9 downto 0);
    VIDOUT_GY      : out std_logic_vector(9 downto 0);
    VIDOUT_BCB     : out std_logic_vector(9 downto 0);
    VIDOUT_BLANK_N : out std_logic;
    VIDOUT_HSYNC_N : out std_logic;
    VIDOUT_VSYNC_N : out std_logic);
end vga;

architecture Behavioral of vga is

  -- Fast low-voltage TTL-level I/O pad with 12 mA drive

  component OBUF_F_12
    port (
      O : out STD_ULOGIC;
      I : in STD_ULOGIC);
  end component;

  -- Basic edge-sensitive flip-flop

  component FD
    port (
      C : in std_logic;
      D : in std_logic;
      Q : out std_logic);
  end component;

  -- Force instances of FD into pads for speed

  attribute iob : string;
```

```
  attribute iob of FD : component is "true";

  component vga_timing
    port (
      h_sync_delay         : out std_logic;
      v_sync_delay         : out std_logic;
      blank                : out std_logic;
      vga_ram_read_address : out std_logic_vector (19 downto 0);
      pixel_clock          : in std_logic;
      reset                : in std_logic);
  end component;

  signal r                      : std_logic_vector (9 downto 0);
  signal g                      : std_logic_vector (9 downto 0);
  signal b                      : std_logic_vector (9 downto 0);
  signal blank                  : std_logic;
  signal hsync                  : std_logic;
  signal vsync                  : std_logic;
  signal vga_ram_read_address   : std_logic_vector(19 downto 0);
  signal vreq                   : std_logic;
  signal vreq_1                 : std_logic;
  signal load_video_word        : std_logic;
  signal vga_shreg              : std_logic_vector(15 downto 0);

begin

  st : vga_timing port map (
    pixel_clock => pix_clk,
    reset => rst,
    h_sync_delay => hsync,
    v_sync_delay => vsync,
    blank => blank,
    vga_ram_read_address => vga_ram_read_address);

  -- Video request is true when the RAM address is even

  -- FIXME: This should be disabled during blanking to reduce memory traffic

  vreq <= not vga_ram_read_address(0);

  -- Generate load_video_word by delaying vreq two cycles

  process (pix_clk)
  begin
    if pix_clk'event and pix_clk='1' then
      vreq_1 <= vreq;
      load_video_word <= vreq_1;
    end if;
  end process;

  -- Generate video_req (to the RAM controller) by delaying vreq by
  -- a cycle synchronized with the pixel clock

  process (clk)
  begin
    if clk'event and clk='1' then
      video_req <= pix_clk and vreq;
```

```
  end if;
end process;

-- The video address is the upper 19 bits from the VGA timing generator
-- because we are using two pixels per word and the RAM address counts words

video_addr <= '0' & vga_ram_read_address(19 downto 1);

-- The video shift register: either load it from RAM or shift it up a byte

process (pix_clk)
begin
  if pix_clk'event and pix_clk='1' then
    if load_video_word = '1' then
      vga_shreg <= video_data;
    else
      -- Shift the low byte of read video data into the high byte
      vga_shreg <= vga_shreg(7 downto 0) & "00000000";
    end if;
  end if;
end process;

-- Copy the upper byte of the video word to the color signals
-- Note that we use three bits for red and green and two for blue.

r(9 downto 2) <= vga_shreg (15 downto 8);
r(1 downto 0) <= "00";
g(9 downto 2) <= vga_shreg (15 downto 8);
g(1 downto 0) <= "00";
b(9 downto 2) <= vga_shreg (15 downto 8);
b(1 downto 0) <= "00";


-- Video clock I/O pad to the DAC

vidclk : OBUF_F_12 port map (
  O => VIDOUT_clk,
  I => pix_clk);

-- Control signals: hsync, vsync, and blank

hsync_ff : FD port map (
  C => pix_clk,
  D => not hsync,
  Q => VIDOUT_HSYNC_N );

vsync_ff : FD port map (
  C => pix_clk,
  D => not vsync,
  Q => VIDOUT_VSYNC_N );

blank_ff :  FD port map (
  C => pix_clk,
  D => not blank,
  Q => VIDOUT_BLANK_N );

-- Three digital color signals
```

```
  rgb_ff : for i in 0 to 9 generate

    r_ff : FD port map (
      C => pix_clk,
      D => r(i),
      Q => VIDOUT_RCR(i) );

    g_ff : FD port map (
      C => pix_clk,
      D => g(i),
      Q => VIDOUT_GY(i) );

    b_ff : FD port map (
      C => pix_clk,
      D => b(i),
      Q => VIDOUT_BCB(i) );

  end generate;

end Behavioral;
```