# Digital Picture Frame
Final Report

Embedded Systems
Spring 2005
Samuel Toepke and Nenad Uzunovic

## Introduction:

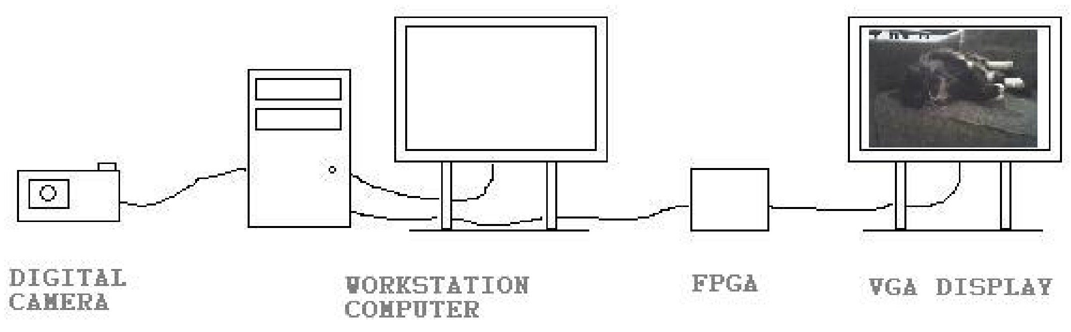
Digital images have replaced film in the common household, and for good reason. Film is expensive, must be bought, must be inserted into the camera, taken out, and developed. Film is expensive, and the development process is time consuming. ‘Hard’ pictures degrade over time, cannot be manipulated, are costly, take up physical space (photo albums), and can be sent only as fast as the mail can go.

Digital images and cameras have vastly improved upon the old process of image capture/management. For a relatively low cost, a digital camera can be bought with a memory stick, and then all the user has to do is point and click. The user will empty the images onto a computer when he/she feels like; then have an empty memory card ready to go. The images can be archived on disk, sent around the world instantly and can be manipulated using any one of the many programs for graphics.

To get digital pictures in a frame, the user will most likely print them out; then manually insert them into the frame. We are suggesting a digital picture frame that allows the user to download the picture onto the picture frame module; then have the module display the picture onto a VGA display.

## Hardware:

1) Personal computer that is capable of manipulating JPEGS, and equipped with a serial port.
2) XESS XSB-300E SPARTAN 2E FPGA. This is a FPGA which is sold by XESS. More information can be found here.
3) VGA monitor.
4) All necessary cables/connectors that are needed to put the pieces of hardware together. We assume that the end user has already used a proprietary digital camera to place the images onto the computer.



## The Flow of the project:

The original idea was to implement a jpeg decoder purely in hardware. Since it was a proud work of a couple of doctorate students, we decided to move on software.

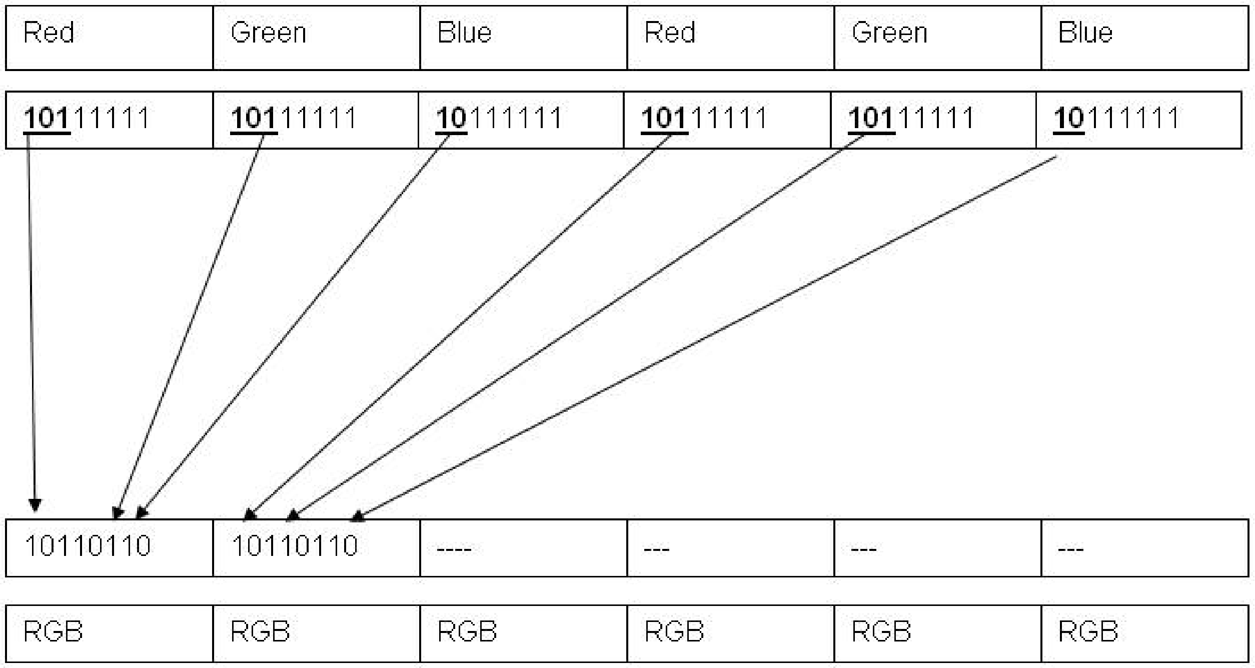
After a month of trying to simplify the robust code of Independent JPEG group and run their library on the board itself, we got a permission of Prof. Edwards to use the DJPEG software and continue

## Software:

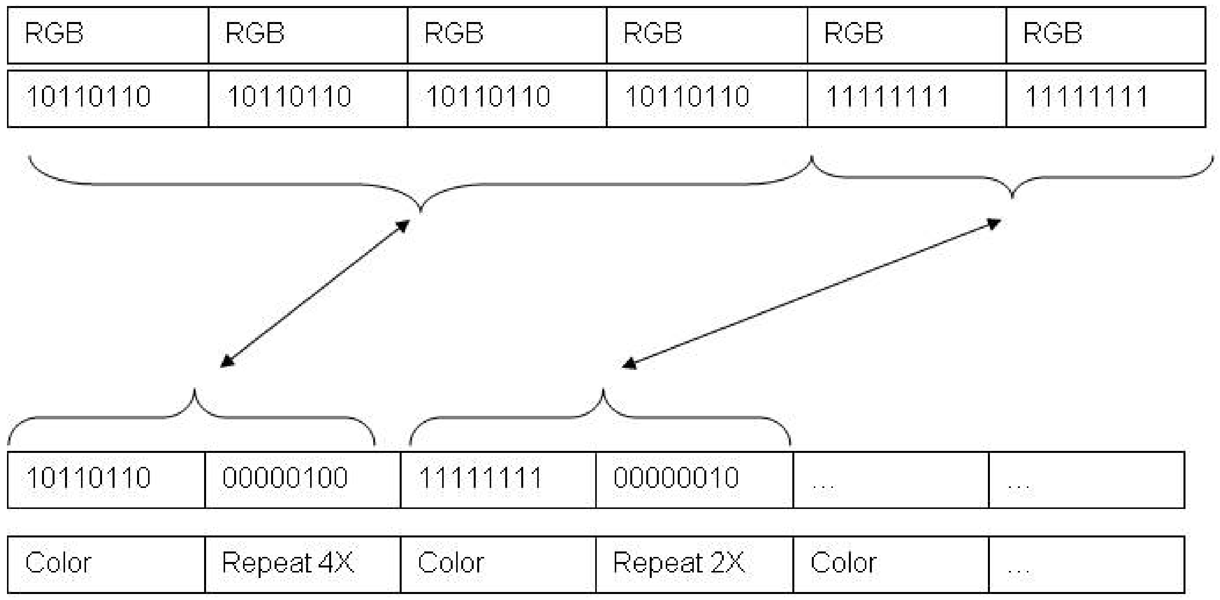
We are using several pieces of software to get the image to its final destination on the monitor. Our test picture is a picture of Olympus, Sam's cat.

1) DJPEG: We are using a class library called djpeg that is distributed by the Independent JPEG group. Information about them, and code, and be found here. The program takes as input a JPEG, we are using an image that is 640X480, and outputs the image in a PPM format. A PPM is a 'portable pixel map', and has information stored in red/green/blue values, more information here.
2) PPM2BIN: We then take the PPM file, strip the padding, and process each pixel. There are 3 bytes of color information, one byte a piece for red/green/blue. We take the first three bits of red, first three bits of green and first two bits of blue, and put them into a new color. This color will now be representative of the pixel.
3) TEST232: A program that Marcio Buss showed us how to use. It simply opens up the serial port and sends a file across the port byte by byte.
4) COMPRESS: It was taking a long time to draw the picture as we were sending a byte for each pixel across the serial cable. 307200 bytes were being sent across at 9600 baud. We decided to use a compression technique that is based on run length encoding. Since many of the adjacent pixels in a JPEG are of the same color, we decided we could create a compression program that would look ahead, and see how many pixels there are in a line that are of the same color. We will then send two bytes, the first byte with the color and the second byte with the number of pixels that are of that color. This look ahead only works for up to 256 pixels, as that is all the information that can be held in eight bits. But that is plenty anyway because (excluding test pictures with only a couple of colors) on the average repetition of the  pixels with same color is less than 50.
5) MAIN.C/ISR.C: On the receiving end. These programs are set up to grab each byte as they come across the serial port. The programs take the first byte, set the color, take the second byte, and loop the number of pixels of that color. This makes picture display much faster. The technique used here is a large FIFO buffer with two pointers chasing each other. First one is the pointer to the fresh data and the second one is the pointer to the data that is being read and processed on the screen. The size of the buffer determines the flexibility of the process, increases the speed and minimizes the probability of error
6) HARDWARE SIMULATION: memoryctrl.vhd, opb_xsb200.vhd, pad_io.vhd, vga_timing.vhd, vga.vhd. These files are taken from last year's Lab 5 and are used to control the video display.

# ppm2bin


Red
Green
Blue
Red
Green
Blue
**101**11111
**101**11111
**10**111111
**101**11111
**101**11111
**10**111111
10110110
10110110
----
---
---
---
RGB
RGB
RGB
RGB
RGB
RGB

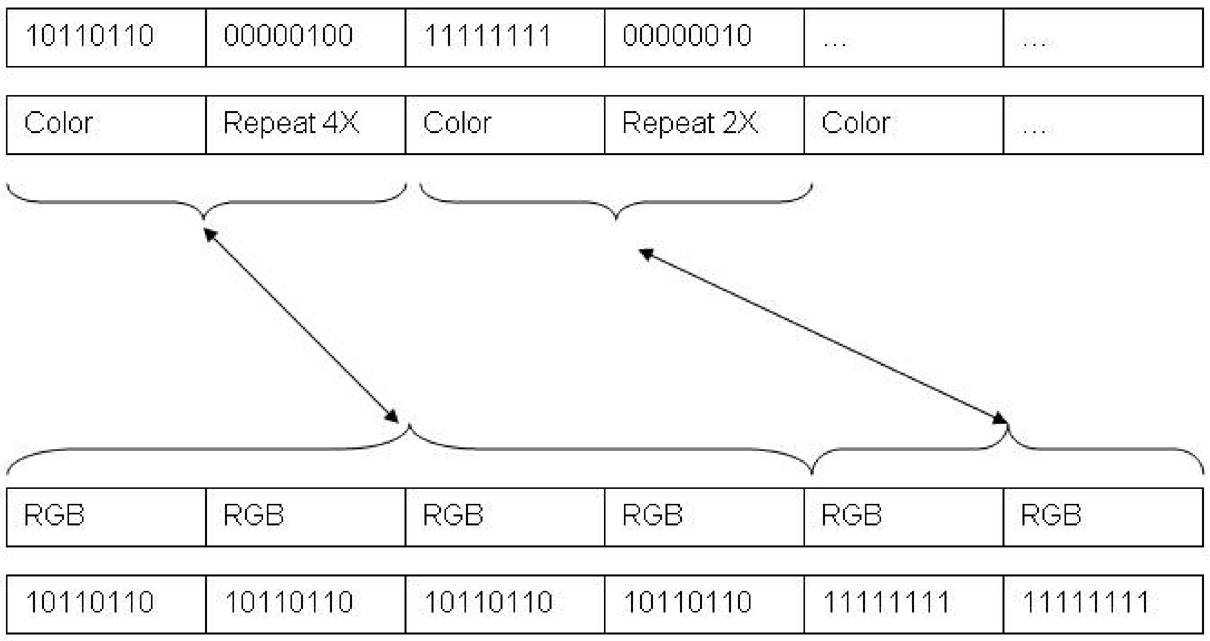
## Compress



| | | | | | |
|---|---|---|---|---|---|
| RGB | RGB | RGB | RGB | RGB | RGB |
| 10110110 | 10110110 | 10110110 | 10110110 | 11111111 | 11111111 |

| | | | | | |
|---|---|---|---|---|---|
| 10110110 | 00000100 | 11111111 | 00000010 | … | … |
| Color | Repeat 4X | Color | Repeat 2X | Color | … |

## Decompress



| | | | | | |
|---|---|---|---|---|---|
| 10110110 | 00000100 | 11111111 | 00000010 | … | … |
| Color | Repeat 4X | Color | Repeat 2X | Color | … |

| | | | | | |
|---|---|---|---|---|---|
| RGB | RGB | RGB | RGB | RGB | RGB |
| 10110110 | 10110110 | 10110110 | 10110110 | 11111111 | 11111111 |

Command Log:

```
#make download
#djpeg < olympus.jpg > olympus.ppm
#./ppm2bin olympus.pnm olympus.bin
#./compress olympus.bin olympus.nas
#./test232 < olympus.nas
```

A user can also browse to the c_source_files and use the command:

```
#sh ./LoadPicture.sh
```

to run a shell script that will automatically put everything on the board and take care of all the picture manipulation. All the user has to do is rename the image ‘picturetoload.jpg’ and place it in the jpg folder.

The above commands will put an image onto the VGA monitor.

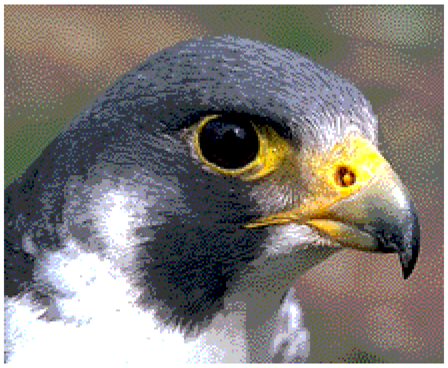
## Continuing Implementation:

Quality: The picture is noticeably fuzzy and blocky. This is because of the sampling that is being done in ppm2bin. There are ways to smooth the picture out, namely using a dithering algorithm. Dithering uses the concept of blurring, or juxtaposing, adjacent pixels to make it appear as though there is a third color.

The benefits of dithering can be seen in these two pictures:

Original full-color photograph

Dithered to 256 colors



The dithering algorithm can be implemented in code, but for our purposes, and under our time constraints, we decided to focus more on the speed of getting the image across. With the speed in place, it will be easier for someone to work on the quality of the picture.

## Conclusions:

Who did what:
Nenad and Sam worked on all parts of the implementation together. Each of us pushed ahead with all testing, development and troubleshooting.

Lessons learned:

- The FPGA has limited memory resources. We spent much time trying to strip down the DJPEG code to no functioning avail.
- The serial port is terribly slow; we even kicked around transferring through USB or parallel, but did not have the time to begin thinking of any of those implementations.
- I personally dislike the lab when it is busy, the best times to work are during the week from ~7 to ~10.

Advice on future projects:

- Of course start early.
- Ask questions, Professor Edwards and the T.A. (s) are there to help, and are more than willing to do so.
- Do not be disheartened with the complexity and apparent difficulty of the work. At the beginning of this project we had no idea how we were going to actually implement anything. But after much time in the lab, and asking questions, we slowly pushed our way forward towards completion.

## Code:

**Code Listing:**
ppm2bin.c
compress.c
test232.c
isr.c
main.c
system.mhs
Makefile
PictureLoad.sh
memoryctyl.vhd
opb_xsb300.vhd
pad_io.vhd
vga.vhd
vga_timing.vhd

**ppm2bin.c:**

```
#include <stdio.h>


FILE *ip, *op;
int red, green, blue, color, red1, green1, blue1;


void fskip(FILE *ip, int num_bytes)
{
  int i;
  for (i=0; i<num_bytes; i++)
    fgetc(ip);
}


int main (int argc, char *argv[])
{

  if ((ip = fopen (argv[1], "rb")) == NULL)
    {
      printf("Cannot open input \n", argv[1]);
      return(1);
    }
  if ((op = fopen(argv[2], "wb")) == NULL)
    {
      printf("Cannont open output \n", argv[2]);
      return(2);
    }

  fskip(ip, 15);

  do
    {
      red1 = fgetc(ip);
      green1 = fgetc(ip);
      blue1 = fgetc(ip);

      red = red1 & 0xE0;
      green = green1 & 0xE0;
      green = green >> 3;
      blue = blue1 & 0xC0;
      blue = blue >> 6;

      color = red | green | blue;
      fputc(color, op);

    }while ((red1!=EOF) | (green1!=EOF) | (blue1!=EOF));
  printf("DONE \n");
  fclose(ip);
  fclose(op);
  return(0);

}
```

**compress.c:**

```
#include <stdio.h>

FILE *ip, *op;

int color, another;
int counter;

int main (int argc, char *argv[])
{

  if ((ip = fopen (argv[1], "rb")) == NULL)
    {
      printf("Cannot open input \n", argv[1]);
      return(1);
    }
  if ((op = fopen(argv[2], "wb")) == NULL)
    {
      printf("Cannont open output \n", argv[2]);
      return(2);
    }

  color = fgetc (ip);
  another = fgetc (ip);
  counter = 1;

 do
   {
      if (color == another)
      {
        counter++;

        if (counter == 120)
          {
            fputc(color, op);
            //printf("color %d   ",color);
            fputc(counter, op);
            //printf("counter %d \n", counter);
            color = another;
            another = fgetc(ip);
            counter = 1;
          }

        another = fgetc(ip);

      } else
        {
          fputc(color, op);
          //printf("color %d   ",color);
          fputc(counter, op);
          //printf("counter %d \n", counter);
          color = another;
          another = fgetc(ip);
          counter = 1;
        }

   } while ((color!=EOF) | (another!=EOF));

 printf("DONE \n");
 fclose(ip);
 fclose(op);
 return(0);
}
```

**test232.c:**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include <errno.h>
```

```
#define DEBUG 0


int main()
{

  char rxbuf[1024], txbuf[1024];

  int fd, nbrx, nbtx, ptx, prx, nb;
  fd_set rfdsin, rfdsout;

  struct termios my_termios;

  fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);


  // configure the serial port
  tcgetattr(fd, &my_termios);
  tcflush(fd, TCIFLUSH);
  my_termios.c_cflag = B9600 | CS8 |CREAD | CLOCAL | HUPCL;
  cfmakeraw(&my_termios);
  cfsetospeed(&my_termios, B9600);
  tcsetattr(fd, TCSANOW, &my_termios);

  nbtx=nbrx=0; ptx=prx=0;

  // the following code tries to be as "nice" as possible

  FD_SET(0, &rfdsin); // stdin
  FD_SET(fd, &rfdsin); // rs232 rx

  while(1){

    FD_ZERO(&rfdsin);
    FD_ZERO(&rfdsout);

    if(nbtx==0) FD_SET(0, &rfdsin); else FD_SET(fd, &rfdsout);
    if(nbrx==0) FD_SET(fd, &rfdsin); else FD_SET(1, &rfdsout);

    select(5, &rfdsin, &rfdsout, NULL, NULL);

    if(nbtx == 0 && FD_ISSET(0, &rfdsin)){ // tx buf empty and data on stdin
      ptx=0;
      nbtx=read(0,txbuf,1024);
      DEBUG && fprintf(stderr,"stdin: read %d\n",nbtx);
    }

    if(nbtx > 0 && FD_ISSET(fd, &rfdsout)){ // have tx data and rs232 free
      nb= write(fd,txbuf+ptx, nbtx);
      DEBUG && fprintf(stderr,"rs232: wrote %d\n",nb);
      ptx += nb;
      nbtx -= nb;
    }

    if(nbrx == 0 && FD_ISSET(fd, &rfdsin)){ // rx buf empty and data on rs232
      prx=0;
      nbrx=read(fd,rxbuf,1024);
      DEBUG && fprintf(stderr,"rs232: read %d\n",nbrx);
    }

    if(nbrx > 0 && FD_ISSET(1, &rfdsout)){ // have rx data and stdout free
      nb=write(1, rxbuf+prx, nbrx);
      DEBUG && fprintf(stderr,"stdout: wrote %d\n",nb);
      prx+= nb;
      nbrx -= nb;
    }
  }

  close(fd);
  }
```

**isr.c:**

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

#define BUFFSIZE 15000

int uart_interrupt_count = 0;
char uart_character;
char buffer[BUFFSIZE];
char *last_in = buffer;
char *last_out = buffer;
int last_in_loop_count = 0;

/*
 * Interrupt service routine for the UART
 */
void uart_handler(void *callback)
{
  Xuint32 IsrStatus;

  Xuint8 incoming_character;

  /* Check the ISR status register so we can identify the interrupt source */
  IsrStatus = XIo_In32(XPAR_MYUART_BASEADDR + XUL_STATUS_REG_OFFSET);

  if ((IsrStatus & (XUL_SR_RX_FIFO_FULL | XUL_SR_RX_FIFO_VALID_DATA)) != 0) {
    /* The input FIFO contains data: read it */
    incoming_character =
      (Xuint8) XIo_In32( XPAR_MYUART_BASEADDR + XUL_RX_FIFO_OFFSET );

    uart_character = incoming_character;
    ++uart_interrupt_count;
  }

  if ((IsrStatus & XUL_SR_TX_FIFO_EMPTY) != 0) {
    /* The output FIFO is empty: we can send another character */
  }

  /*
   * putting data into buffer
   * will read it in main.c
   */

  if (last_in == buffer + BUFFSIZE) // loop around
```

```
    {
      last_in = buffer;
      last_in_loop_count++;
    }

  *last_in = uart_character; //input a new character
  last_in++; //pointer

}
```

**main.c:**

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

#include "xintc_l.h"
#include "xuartlite_l.h"

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

#define BUFFSIZE 15000

// defined in isr.c

extern void uart_handler(void *callback);
extern volatile int uart_interrupt_count;
extern volatile char uart_character;
extern volatile char buffer[BUFFSIZE];
extern volatile char *last_in;
extern volatile char *last_out;
extern volatile int last_in_loop_count;

int z=0;
int init_count;
char color;
char counter;
int last_out_loop_count = 0;


/*
 * setup_interrupts: Initialize the interrupt sources and handlers
 *
 * Should be called once when the system starts
 *
 * The main _interrupt_handler() function from Xilinx
 *
 * Saves and restores CPU context, etc.
 *
 * Sees which interrupts are pending, and for each it
 *     acknowledges the interrupt and
 *     calls a user-defined interrupt handler in Xintc_InterruptVectorTable
 *
 * Place interrupt service routines in isr.c to ensure they are placed in
 * the proper memory segment.
 */
```

```
void setup_interrupts()
{
  /*
   * Reset the interrupt controller peripheral
   */

  /* Disable the interrupt signal */
  XIntc_mMasterDisable(XPAR_INTC_SINGLE_BASEADDR);

  /* Disable all interrupt sources */
  XIntc_mEnableIntr(XPAR_INTC_SINGLE_BASEADDR,0);

  /* Acknowledge all possible interrupt sources
     to make sure none are pending */
  XIntc_mAckIntr(XPAR_INTC_SINGLE_BASEADDR, 0xffffffff);

  /*
   * Install the UART interrupt handler
   */

  XIntc_InterruptVectorTable[XPAR_INTC_MYUART_INTERRUPT_INTR].Handler =
    uart_handler;

  /*
   * Enable interrupt sources
   */

  /* Enable CPU interrupts */
  microblaze_enable_interrupts();

  /* Enable interrupts from the interrupt controller */
  XIntc_mMasterEnable(XPAR_INTC_SINGLE_BASEADDR);

  /* Tell the interrupt controller to accept interrupts from the UART */
  XIntc_mEnableIntr(XPAR_INTC_SINGLE_BASEADDR, XPAR_MYUART_INTERRUPT_MASK);

  /* Enable UART interrupt generation */
  XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);
}

void clear_screen()
{
  int q;
  for (q=0; q<H*W; q++)
    {
      XIo_Out8(VGA_START + q, 0);
    }
}

void write_pixel()
{
  char j;

  color = *last_out; // take color byte

  last_out++; // increment pointer value

  counter = *last_out; // take a byte that says
                       //how many times to repeat the color byte
  if (last_out == buffer + BUFFSIZE) // loop around
    {
      last_out = buffer;
      last_out_loop_count++;
    }
  else
    last_out++;

  for (j=0; j<counter; j++)  // display color byte counter times
    {
      XIo_Out8(VGA_START + z, color);
      z++;
    }
}


int main()
{
  microblaze_enable_icache();

  setup_interrupts();

  clear_screen();

  /* in the next few lines
   * skipping first two bytes
   * initializing stuff (can solve it with header, minor thing)*/

  init_count = uart_interrupt_count;
  while (init_count == uart_interrupt_count){}

  color = uart_character;

  init_count = uart_interrupt_count;
  while (init_count == uart_interrupt_count){}

  counter = uart_character;



  for (;;)
    {

      if (last_out == buffer + BUFFSIZE)   // loop around
      {
        last_out = buffer;
        last_out_loop_count++;
      }

      if (last_in_loop_count > last_out_loop_count)
      {
        if (last_in < last_out)
          {
            write_pixel();
          }
      }

      if (last_out_loop_count == last_in_loop_count)
      {
        if (last_in > last_out + 1)        // write as long as we have fresh data
          write_pixel();
      }
      else {}

    }

  return 0;
}
```

**system.mhs:**

```
# Parameters
PARAMETER VERSION = 2.0.0

# Global Ports

PORT PB_A = PB_A, DIR = OUT, VEC = [19:0]
PORT PB_D = PB_D, DIR = INOUT, VEC = [15:0]
PORT PB_LB_N = PB_LB_N, DIR = OUT
PORT PB_UB_N = PB_UB_N, DIR = OUT
PORT PB_WE_N = PB_WE_N, DIR = OUT
PORT PB_OE_N = PB_OE_N, DIR = OUT
PORT RAM_CE_N = RAM_CE_N, DIR = OUT
PORT VIDOUT_CLK = VIDOUT_CLK, DIR = OUT
PORT VIDOUT_HSYNC_N = VIDOUT_HSYNC_N, DIR = OUT
PORT VIDOUT_VSYNC_N = VIDOUT_VSYNC_N, DIR = OUT
PORT VIDOUT_BLANK_N = VIDOUT_BLANK_N, DIR = OUT
PORT VIDOUT_RCR = VIDOUT_RCR, DIR = OUT, VEC = [9:0]
PORT VIDOUT_GY = VIDOUT_GY, DIR = OUT, VEC = [9:0]
PORT VIDOUT_BCB = VIDOUT_BCB, DIR = OUT, VEC = [9:0]
PORT FPGA_CLK1 = FPGA_CLK1, DIR = IN
PORT RS232_TD = RS232_TD, DIR=OUT
PORT RS232_RD = RS232_RD, DIR=IN
PORT AU_CSN_N =  AU_CSN_N, DIR=OUT
PORT AU_BCLK =  AU_BCLK, DIR=OUT
PORT AU_MCLK = AU_MCLK, DIR=OUT
PORT AU_LRCK = AU_LRCK, DIR=OUT
PORT AU_SDTI = AU_SDTI, DIR=OUT
PORT AU_SDTO0 = AU_SDTO0, DIR=IN

# Sub Components


BEGIN microblaze
 PARAMETER INSTANCE = mymicroblaze
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_ICACHE = 1
 PARAMETER C_ADDR_TAG_BITS = 6
 PARAMETER C_CACHE_BYTE_SIZE = 2048
 PARAMETER C_ICACHE_BASEADDR = 0x00860000
 PARAMETER C_ICACHE_HIGHADDR = 0x0087FFFF
 PORT Clk = sys_clk
 PORT Reset = fpga_reset
 PORT Interrupt = intr
 BUS_INTERFACE DLMB = d_lmb
 BUS_INTERFACE ILMB = i_lmb
 BUS_INTERFACE DOPB = myopb_bus
 BUS_INTERFACE IOPB = myopb_bus
END

BEGIN opb_intc
 PARAMETER INSTANCE = intc
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0xFFFF0000
 PARAMETER C_HIGHADDR = 0xFFFF00FF
 PORT OPB_Clk = sys_clk
 PORT Intr =  uart_intr
 PORT Irq =   intr
 BUS_INTERFACE SOPB = myopb_bus
END

BEGIN bram_block
 PARAMETER INSTANCE = bram
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = conn_0
 BUS_INTERFACE PORTB = conn_1
END

BEGIN opb_xsb300
 PARAMETER INSTANCE = xsb300
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x00800000
 PARAMETER C_HIGHADDR = 0x00FFFFFF
 PORT PB_A = PB_A
 PORT PB_D = PB_D
 PORT PB_LB_N = PB_LB_N
 PORT PB_UB_N = PB_UB_N
 PORT PB_WE_N = PB_WE_N
 PORT PB_OE_N = PB_OE_N
 PORT RAM_CE_N = RAM_CE_N
 PORT OPB_Clk = sys_clk
 PORT pixel_clock = pixel_clock
 PORT VIDOUT_CLK = VIDOUT_CLK
 PORT VIDOUT_HSYNC_N = VIDOUT_HSYNC_N
 PORT VIDOUT_VSYNC_N = VIDOUT_VSYNC_N
 PORT VIDOUT_BLANK_N = VIDOUT_BLANK_N
 PORT VIDOUT_RCR = VIDOUT_RCR
 PORT VIDOUT_GY = VIDOUT_GY
 PORT VIDOUT_BCB = VIDOUT_BCB
 BUS_INTERFACE SOPB = myopb_bus
END

BEGIN clkgen
 PARAMETER INSTANCE = clkgen_0
 PARAMETER HW_VER = 1.00.a
 PORT FPGA_CLK1 = FPGA_CLK1
 PORT sys_clk = sys_clk
 PORT pixel_clock = pixel_clock
 PORT fpga_reset = fpga_reset
END

BEGIN lmb_lmb_bram_if_cntlr
 PARAMETER INSTANCE = lmb_lmb_bram_if_cntlr_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00000FFF
 BUS_INTERFACE DLMB = d_lmb
 BUS_INTERFACE ILMB = i_lmb
 BUS_INTERFACE PORTA = conn_0
 BUS_INTERFACE PORTB = conn_1
END

BEGIN opb_uartlite
 PARAMETER INSTANCE = myuart
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_CLK_FREQ = 50_000_000
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_BASEADDR = 0xFEFF0100
 PARAMETER C_HIGHADDR = 0xFEFF01FF
 PORT OPB_Clk = sys_clk
 PORT Interrupt = uart_intr
 BUS_INTERFACE SOPB = myopb_bus
 PORT RX=RS232_RD
 PORT TX=RS232_TD
END

BEGIN opb_v20
 PARAMETER INSTANCE = myopb_bus
 PARAMETER HW_VER = 1.10.a
 PARAMETER C_DYNAM_PRIORITY = 0
 PARAMETER C_REG_GRANTS = 0
 PARAMETER C_PARK = 0
 PARAMETER C_PROC_INTRFCE = 0
 PARAMETER C_DEV_BLK_ID = 0
 PARAMETER C_DEV_MIR_ENABLE = 0
 PARAMETER C_BASEADDR = 0x0fff1000
 PARAMETER C_HIGHADDR = 0x0fff10ff
 PORT SYS_Rst = fpga_reset
 PORT OPB_Clk = sys_clk
END

BEGIN lmb_v10
 PARAMETER INSTANCE = d_lmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk
 PORT SYS_Rst = fpga_reset
END

BEGIN lmb_v10
 PARAMETER INSTANCE = i_lmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk
 PORT SYS_Rst = fpga_reset
END
```

**Makefile:**

```
# Makefile for CSEE 4840, Lab 3

SYSTEM = system

MICROBLAZE_OBJS = \
      c_source_files/main.o \
      c_source_files/isr.o

LIBRARIES = mymicroblaze/lib/libxil.a

ELF_FILE = $(SYSTEM).elf

NETLIST = implementation/$(SYSTEM).ngc

# Bitstreams for the FPGA

FPGA_BITFILE = implementation/$(SYSTEM).bit
MERGED_BITFILE = implementation/download.bit

# Files to be downloaded to the SRAM

SRAM_BINFILE = implementation/sram.bin
SRAM_HEXFILE = implementation/sram.hex

MHSFILE = $(SYSTEM).mhs
MSSFILE = $(SYSTEM).mss

FPGA_ARCH = spartan2e
DEVICE = xc2s300epq208-6

LANGUAGE = vhdl
PLATGEN_OPTIONS = -p $(FPGA_ARCH) -lang $(LANGUAGE)
LIBGEN_OPTIONS = -p $(FPGA_ARCH) $(MICROBLAZE_LIBG_OPT)

# Paths for programs
```

```
XILINX = /usr/cad/xilinx/ise6.1i
ISEBINDIR = $(XILINX)/bin/lin
ISEENVCMDS = LD_LIBRARY_PATH=$(ISEBINDIR) XILINX=$(XILINX) PATH=$(ISEBINDIR)

XILINX_EDK = /usr/cad/xilinx/edk3.2

MICROBLAZE = /usr/cad/xilinx/gnu
MBBINDIR = $(MICROBLAZE)/bin
XESSBINDIR = /usr/cad/xess/bin

# Executables

XST = $(ISEENVCMDS) $(ISEBINDIR)/xst
XFLOW = $(ISEENVCMDS) $(ISEBINDIR)/xflow
BITGEN = $(ISEENVCMDS) $(ISEBINDIR)/bitgen
DATA2MEM = $(ISEENVCMDS) $(ISEBINDIR)/data2mem
XSLOAD = $(XESSBINDIR)/xsload
XESS_BOARD = XSB-300E

MICROBLAZE_CC = $(MBBINDIR)/microblaze-gcc
MICROBLAZE_CC_SIZE = $(MBBINDIR)/microblaze-size
MICROBLAZE_OBJCOPY = $(MBBINDIR)/microblaze-objcopy

# External Targets

all :
	@echo "Makefile to build a Microprocessor system :"
	@echo "Run make with any of the following targets"
	@echo "  make libs     : Configures the sw libraries for this system"
	@echo "  make program  : Compiles the program sources for all the processor
instances"
	@echo "  make netlist  : Generates the netlist for this system ($(SYSTEM))"
	@echo "  make bits     : Runs Implementation tools to generate the
bitstream"
	@echo "  make init_bram: Initializes bitstream with BRAM data"
	@echo "  make download : Downloads the bitstream onto the board"
	@echo "  make netlistclean: Deletes netlist"
	@echo "  make hwclean  : Deletes implementation dir"
	@echo "  make libsclean: Deletes sw libraries"
	@echo "  make programclean: Deletes compiled ELF files"
	@echo "  make clean    : Deletes all generated files/directories"
	@echo " "
	@echo "  make <target> : (Default)"
	@echo "      Creates a Microprocessor system using default initializations"
	@echo "      specified for each processor in MSS file"


bits : $(FPGA_BITFILE)

netlist : $(NETLIST)

libs : $(LIBRARIES)

program : $(ELF_FILE)

init_bram : $(MERGED_BITFILE)

clean : hwclean libsclean programclean
	rm -f bram_init.sh
```

```
      rm -f _impact.cmd

hwclean : netlistclean
      rm -rf implementation synthesis xst hdl
      rm -rf xst.srp $(SYSTEM).srp

netlistclean :
      rm -f $(FPGA_BITFILE) $(MERGED_BITFILE) \
        $(NETLIST) implementation/$(SYSTEM)_bd.bmm

libsclean :
      rm -rf mymicroblaze/lib

programclean :
      rm -f $(ELF_FILE) $(SRAM_BITFILE) $(SRAM_HEXFILE)

#
# Software rules
#

MICROBLAZE_MODE = executable

# Assemble software libraries from the .mss and .mhs files

$(LIBRARIES) : $(MHSFILE) $(MSSFILE)
      PATH=$$PATH:$(MBBINDIR) XILINX=$(XILINX) XILINX_EDK=$(XILINX_EDK) \
       perl -I $(XILINX_EDK)/bin/nt/perl5lib $(XILINX_EDK)/bin/nt/libgen.pl \
        $(LIBGEN_OPTIONS) $(MSSFILE)

# Compilation

MICROBLAZE_CC_CFLAGS =
MICROBLAZE_CC_OPT = -O3 #-mxl-gp-opt
MICROBLAZE_CC_DEBUG_FLAG =#  -gstabs
MICROBLAZE_INCLUDES = -I./mymicroblaze/include/ # -I
MICROBLAZE_CFLAGS = \
      $(MICROBLAZE_CC_CFLAGS)\
      -mxl-barrel-shift \
      $(MICROBLAZE_CC_OPT) \
      $(MICROBLAZE_CC_DEBUG_FLAG) \
      $(MICROBLAZE_INCLUDES)

$(MICROBLAZE_OBJS) : %.o : %.c
      PATH=$(MBBINDIR) $(MICROBLAZE_CC) $(MICROBLAZE_CFLAGS) -c $< -o $@

# Linking

# Uncomment the following to make linker print locations for everything
# MICROBLAZE_LD_FLAGS = -Wl,-M
MICROBLAZE_LINKER_SCRIPT = -Wl,-T -Wl,mylinkscript
MICROBLAZE_LIBPATH = -L./mymicroblaze/lib/
MICROBLAZE_CC_START_ADDR_FLAG= -Wl,-defsym -Wl,_TEXT_START_ADDR=0x00000000
MICROBLAZE_CC_STACK_SIZE_FLAG=  -Wl,-defsym -Wl,_STACK_SIZE=0x200
MICROBLAZE_LFLAGS = \
       -xl-mode-$(MICROBLAZE_MODE) \
      $(MICROBLAZE_LD_FLAGS) \
      $(MICROBLAZE_LINKER_SCRIPT) \
      $(MICROBLAZE_LIBPATH) \
      $(MICROBLAZE_CC_START_ADDR_FLAG) \
      $(MICROBLAZE_CC_STACK_SIZE_FLAG)

$(ELF_FILE) :  $(LIBRARIES) $(MICROBLAZE_OBJS)
      PATH=$(MBBINDIR) $(MICROBLAZE_CC) $(MICROBLAZE_LFLAGS) \
            $(MICROBLAZE_OBJS) -o $(ELF_FILE)
      $(MICROBLAZE_CC_SIZE) $(ELF_FILE)

#
# Hardware rules
#

# Hardware compilation : optimize the netlist, place and route

$(FPGA_BITFILE) : $(NETLIST) \
            etc/fast_runtime.opt etc/bitgen.ut data/$(SYSTEM).ucf
      cp -f etc/bitgen.ut implementation/
      cp -f etc/fast_runtime.opt implementation/
      cp -f data/$(SYSTEM).ucf implementation/$(SYSTEM).ucf
      $(XFLOW) -wd implementation -p $(DEVICE) -implement fast_runtime.opt \
            $(SYSTEM).ngc
      cd implementation; $(BITGEN) -f bitgen.ut $(SYSTEM)

# Hardware assembly: Create the netlist from the .mhs file

$(NETLIST) : $(MHSFILE)
      XILINX=$(XILINX) XILINX_EDK=$(XILINX_EDK) \
      perl -I $(XILINX_EDK)/bin/nt/perl5lib $(XILINX_EDK)/bin/nt/platgen.pl \
        $(PLATGEN_OPTIONS) -st xst $(MHSFILE)
      perl synth_modules.pl < synthesis/xst.scr > xst.scr
      $(XST) -ifn xst.scr
      rm -r xst xst.scr
      $(XST) -ifn synthesis/$(SYSTEM).scr

#
# Downloading
#

# Add software code to the FPGA bitfile

$(MERGED_BITFILE) : $(FPGA_BITFILE) $(ELF_FILE)
      $(DATA2MEM) -bm implementation/$(SYSTEM)_bd \
        -bt implementation/$(SYSTEM) \
        -bd $(ELF_FILE) tag bram -o b $(MERGED_BITFILE)

# Create a .hex file with data for the SRAM

$(SRAM_HEXFILE) : $(ELF_FILE)
      $(MICROBLAZE_OBJCOPY) \
            -j .sram_text -j .sdata2 -j .sdata -j .rodata -j .data \
            -O binary $(ELF_FILE) $(SRAM_BINFILE)
      ./bin2hex  -a 60000 < $(SRAM_BINFILE) > $(SRAM_HEXFILE)

# Download the files to the target board

download : $(MERGED_BITFILE) $(SRAM_HEXFILE)
      $(XSLOAD) -ram -b $(XESS_BOARD) $(SRAM_HEXFILE)
      $(XSLOAD) -fpga -b $(XESS_BOARD) $(MERGED_BITFILE)
```

**PictureLoad.sh:**

```
#!/bin/sh

cd ..
make download
cd jpg/
djpeg <picturetoload.jpg>picturetoload.ppm
./ppm2bin picturetoload.ppm picturetoload.bin
./compress picturetoload.bin picturetoload.nas
./test232 < picturetoload.nas
```

**memoryctrl.vhd:**

```
-------------------------------------------------------------------------------
--
-- Memory controller state machine
--
-- Arbitrates between requests from the processor and video system
-- The video system gets priority
-- Also handles 32- to 16-bit datapath width conversion (sequencing)
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity memoryctrl is
  port (
    rst : in std_logic;
    clk : in std_logic;
    cs : in std_logic;                  -- chip select (address valid)
    select0 : in std_logic;             -- select (true through whole cycle)
    rnw : in std_logic;                 -- read/write'
    vreq : in std_logic;                -- video request
    onecycle : in std_logic;            -- one- or two-byte access (not four)

    videocycle : out std_logic;         -- acknowledges vreq
    hihalf : out std_logic;             -- doing bytes 2,3 of 32-bit access
    pb_wr : out std_logic;              -- Write request to off-chip memory
    pb_rd : out std_logic;              -- Read request to off-chip memory
    xfer : out std_logic;               -- Transfer acknowledge for OPB

    ce0 : out std_logic;                -- lower 16 bit OPB data latch enable
    ce1 : out std_logic;                -- upper 16 bit OPB data latch enable
    rres : out std_logic;               -- clear OPB data output latches

    video_ce : out std_logic);          -- video buffer latch enable
end memoryctrl;

architecture Behavioral of memoryctrl is
  -- State machine bits
  -- largely one-hot, but one ra and one rb bit can be true simultaneously
  signal r_idle : std_logic;
  signal r_common : std_logic;
  signal r_ra1 : std_logic;
  signal r_ra2 : std_logic;
  signal r_rb1 : std_logic;
  signal r_rb2 : std_logic;
  signal r_rb3 : std_logic;
  signal r_w : std_logic;
  signal r_xfer : std_logic;

  signal vcycle_1 : std_logic;
  signal vcycle_2 : std_logic;

  signal r_idle_next_state : std_logic;
  signal r_common_next_state : std_logic;
  signal r_ra1_next_state : std_logic;
  signal r_ra2_next_state : std_logic;
  signal r_rb1_next_state : std_logic;
  signal r_rb2_next_state : std_logic;
  signal r_rb3_next_state : std_logic;
  signal r_w_next_state : std_logic;
  signal r_xfer_next_state : std_logic;

begin

  -- Sequential process for the state machine

  process (clk, rst)
  begin
    if rst = '1' then
      r_idle <= '1';
      r_common <= '0';
      r_ra1 <= '0';
      r_ra2 <= '0';
      r_rb1 <= '0';
      r_rb2 <= '0';
      r_rb3 <= '0';
      r_w <= '0';
      r_xfer <= '0';
    elsif clk'event and clk='1' then
      r_idle <= r_idle_next_state;
      r_common <= r_common_next_state;
      r_ra1 <= r_ra1_next_state;
      r_ra2 <= r_ra2_next_state;
      r_rb1 <= r_rb1_next_state;
      r_rb2 <= r_rb2_next_state;
      r_rb3 <= r_rb3_next_state;
      r_w <= r_w_next_state;
      r_xfer <= r_xfer_next_state;
    end if;
  end process;

  -- Combinational next-state logic

  r_idle_next_state <= (r_idle and (not cs)) or r_xfer or (not select0);
  r_common_next_state <= select0 and
                         ((r_idle and cs) or (r_common and vreq));
  r_ra1_next_state <= select0 and
                      r_common and (not vreq) and rnw;
  r_ra2_next_state <= select0 and
                      r_ra1;
  r_rb1_next_state <= select0 and
                      ((r_common and not onecycle and not vreq and rnw) or
                       (r_rb1 and vreq));
  r_rb2_next_state <= select0 and
                      (r_rb1 and (not vreq));
  r_rb3_next_state <= select0 and
                      r_rb2;
  r_w_next_state <= select0 and
                    ((r_common and (not rnw) and (not vreq) and
                      (not onecycle)) or
                     (r_w and vreq));
  r_xfer_next_state <= select0 and
                       ((r_common and onecycle and
                         (not rnw) and (not vreq)) or
                        (r_w and (not vreq)) or (r_ra2 and onecycle) or r_rb3);

  -- Combinational output logic

  pb_wr <= (r_common and (not rnw) and (not vreq)) or (r_w and (not vreq));
  pb_rd <= vreq or (r_common and (not vreq) and rnw) or (r_rb1 and (not vreq));

  hihalf <= (r_w and (not vreq)) or (r_rb1 and (not vreq));

  ce0 <= r_ra2;
  ce1 <= r_rb3;
  rres <= r_xfer;
  xfer <= r_xfer;

  -- Two-cycle delay of video request
  -- (implicitly assumes video cycles always succeed)
  process (clk, rst)
  begin
    if(rst = '1') then
      vcycle_1 <= '0';
      vcycle_2 <= '0';
    elsif clk'event and clk='1' then
      vcycle_1 <= vreq;
      vcycle_2 <= vcycle_1;
    end if;
  end process;

  videocycle <= vreq;
  video_ce <= vcycle_2;

end Behavioral;
```

**opb_xsb300.vhd:**

```
-------------------------------------------------------------------------------
--
-- OPB bus bridge for the XESS XSB-300E board
--
-- Includes a memory controller, a VGA framebuffer, and glue for the SRAM
--
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity opb_xsb300 is
```

```
  generic (
    C_OPB_AWIDTH    : integer := 32;
    C_OPB_DWIDTH    : integer := 32;
    C_BASEADDR      : std_logic_vector := X"2000_0000";
    C_HIGHADDR      : std_logic_vector := X"2000_00FF");

  port (
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;
    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;
    pixel_clock : in std_logic;
    UIO_DBus : out std_logic_vector (31 downto 0);
    UIO_errAck : out std_logic;
    UIO_retry : out std_logic;
    UIO_toutSup : out std_logic;
    UIO_xferAck : out std_logic;
    PB_A : out std_logic_vector (19 downto 0);
    PB_UB_N : out std_logic;
    PB_LB_N : out std_logic;
    PB_WE_N : out std_logic;
    PB_OE_N : out std_logic;
    RAM_CE_N : 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;
    PB_D : inout std_logic_vector (15 downto 0));
end opb_xsb300;

architecture Behavioral of opb_xsb300 is

  constant C_MASK : integer := 0; -- huge address window as we are a bridge

  signal addr_mux : std_logic_vector(19 downto 0);
  signal video_addr : std_logic_vector (19 downto 0);
  signal video_data : std_logic_vector (15 downto 0);
  signal video_req : std_logic;
  signal video_ce : std_logic;
  signal i : integer;
  signal cs : std_logic;

  signal onecycle : std_logic ;
  signal videocycle, amuxsel, hihalf : std_logic;
  signal rce0, rce1, rreset : std_logic;
  signal xfer : std_logic;
  signal pb_wr, pb_rd : std_logic;

  signal sram_ce : std_logic;

  signal rnw : std_logic;
  signal addr : std_logic_vector (23 downto 0);

  signal be : std_logic_vector (3 downto 0);
  signal pb_bytesel : std_logic_vector (1 downto 0);

  signal wdata : std_logic_vector (31 downto 0);
  signal wdata_mux : std_logic_vector (15 downto 0);

  signal rdata : std_logic_vector (15 downto 0); -- register data read - FDRE

  component vga
    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 component;

  component memoryctrl
    port (
      rst : in std_logic;
      clk : in std_logic;
      cs : in std_logic;
      select0 : in std_logic;
      rnw : in std_logic;
      vreq : in std_logic;
      onecycle : in std_logic;
      videocycle : out std_logic;
      hihalf : out std_logic;
      pb_wr : out std_logic;
      pb_rd : out std_logic;
      xfer : out std_logic;
      ce0 : out std_logic;
      ce1 : out std_logic;
      rres : out std_logic;
      video_ce : out std_logic);
  end component;

  component pad_io
    port (
      clk : in std_logic;
      rst : in std_logic;
      PB_A : out std_logic_vector(19 downto 0);
      PB_UB_N : out std_logic;
      PB_LB_N : out std_logic;
      PB_WE_N : out std_logic;
      PB_OE_N : out std_logic;
      RAM_CE_N : out std_logic;
      PB_D : inout std_logic_vector(15 downto 0);
      pb_addr : in std_logic_vector(19 downto 0);
```

```
      pb_ub : in std_logic;
      pb_lb : in std_logic;
      pb_wr : in std_logic;
      pb_rd : in std_logic;
      ram_ce : in std_logic;
      pb_dread : out std_logic_vector(15 downto 0);
      pb_dwrite : in std_logic_vector(15 downto 0));
  end component;

begin

  -- Framebuffer

  vga1 : vga
    port map (
      clk => OPB_Clk,
      pix_clk => pixel_clock,
      rst => OPB_Rst,
      video_addr => video_addr,
      video_data => video_data,
      video_req => video_req,
      VIDOUT_CLK => VIDOUT_CLK,
      VIDOUT_RCR => VIDOUT_RCR,
      VIDOUT_GY => VIDOUT_GY,
      VIDOUT_BCB => VIDOUT_BCB,
      VIDOUT_BLANK_N => VIDOUT_BLANK_N,
      VIDOUT_HSYNC_N => VIDOUT_HSYNC_N,
      VIDOUT_VSYNC_N => VIDOUT_VSYNC_N);

-- Memory control/arbitration state machine

  memoryctrl1 : memoryctrl port map (
    rst => OPB_Rst,
    clk => OPB_Clk,
    cs => cs,
    select0 => OPB_select,
    rnw => rnw,
    vreq => video_req,
    onecycle => onecycle,
    videocycle => videocycle,
    hihalf => hihalf,
    pb_wr => pb_wr,
    pb_rd => pb_rd,
    xfer => xfer,
    ce0 => rce0,
    ce1 => rce1,
    rres => rreset,
    video_ce => video_ce);

-- I/O pads

  pad_io1 : pad_io port map (
    clk => OPB_Clk,
    rst => OPB_Rst,
    PB_A => PB_A,
    PB_UB_N => PB_UB_N,
    PB_LB_N => PB_LB_N,
    PB_WE_N => PB_WE_N,
    PB_OE_N => PB_OE_N,
```

```vhdl
    RAM_CE_N => RAM_CE_N,
    PB_D => PB_D,
    pb_addr => addr_mux,
    pb_rd => pb_rd,
    pb_wr => pb_wr,
    pb_ub => pb_bytesel(1),
    pb_lb => pb_bytesel(0),
    ram_ce => sram_ce,
    pb_dread  => rdata,
    pb_dwrite => wdata_mux);

  sram_ce <= pb_rd or pb_wr;

  amuxsel <= videocycle;

  addr_mux <= video_addr when (amuxsel = '1')
              else (addr(20 downto 2) & (addr(1) or hihalf));

  onecycle <= (not be(3)) or (not be(2)) or (not be(1)) or (not be(0));

  wdata_mux <= wdata(15 downto 0) when ((addr(1) or hihalf) = '1')
               else wdata(31 downto 16);

  process(videocycle, be, addr(1), hihalf, pb_rd, pb_wr)
  begin
    if videocycle = '1' then
      pb_bytesel <= "11";
    elsif pb_rd='1' or pb_wr='1' then
      if addr(1)='1' or hihalf='1' then
        pb_bytesel <= be(1 downto 0);
      else
        pb_bytesel <= be(3 downto 2);
      end if;
    else
      pb_bytesel <= "00";
    end if;
  end process;

  cs <= OPB_select when OPB_ABus(31 downto 20) = X"008" else '0';

  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;

  process (OPB_Clk)
  begin
    if OPB_Clk'event and OPB_Clk = '1' then
      if OPB_RST = '1' then
        addr <= X"000000";
      else
        addr <= OPB_ABus(23 downto 0);
      end if;
    end if;
  end process;

  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;

  process (OPB_Clk)
  begin
    if OPB_Clk'event and OPB_Clk = '1' then
      if OPB_Rst = '1' then
        wdata <= X"00000000";
      else
        wdata <= OPB_DBus;
      end if;
    end if;
  end process;

  process (OPB_Clk)
  begin
    if OPB_Clk'event and OPB_Clk = '1' then
      if video_ce = '1' then
        video_data <= rdata;
      end if;
    end if;
  end process;

  -- Write the low two bytes if rce0 or rce1 is enabled

  process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      UIO_DBus(15 downto 0) <= X"0000";
    elsif OPB_Clk'event and OPB_Clk = '1' then
      if rreset = '1' then
        UIO_DBus(15 downto 0) <= X"0000";
      elsif (rce1 or rce0) = '1' then
        UIO_DBus(15 downto 0) <= rdata(15 downto 0);
      end if;
    end if;
  end process;

  -- Write the high two bytes if rce0 is enabled

  process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      UIO_DBus(31 downto 16) <= X"0000";
    elsif OPB_Clk'event and OPB_Clk = '1' then
      if rreset = '1' then
        UIO_DBus(31 downto 16) <= X"0000";
      elsif rce0 = '1' then
```

```
        UIO_DBus(31 downto 16) <= rdata(15 downto 0);
      end if;
    end if;
  end process;

  -- unused outputs

  UIO_errAck <= '0';
  UIO_retry <= '0';
  UIO_toutSup <= '0';

  UIO_xferAck <= xfer;

end Behavioral;
```

**pad_io.vhd:**

```
-------------------------------------------------------------------------------
--
-- I/O Pads and associated circuity for the XSB-300E board
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
-- Pad drivers, a little glue logic, and flip-flops for most bus signals
--
-- All signals, both control and the data and address busses, are registered.
-- FDC and FDP flip-flops are forced into the pads to do this.
--
-- Only the data bus is mildly complex: it latches data in both directions
-- as well as delaying the tristate control signals a cycle.
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity pad_io is
  port (
    clk : in std_logic;
    rst : in std_logic;
    PB_A : out std_logic_vector(19 downto 0);
    PB_UB_N : out std_logic;
    PB_LB_N : out std_logic;
    PB_WE_N : out std_logic;
    PB_OE_N : out std_logic;
    RAM_CE_N : out std_logic;
    PB_D : inout std_logic_vector(15 downto 0);
    pb_addr : in std_logic_vector(19 downto 0);
    pb_ub : in std_logic;
    pb_lb : in std_logic;
    pb_wr : in std_logic;
    pb_rd : in std_logic;
    ram_ce : in std_logic;
    pb_dread : out std_logic_vector(15 downto 0);
    pb_dwrite : in std_logic_vector(15 downto 0));
end pad_io;

architecture Behavioral of pad_io is

  -- Flip-flop with asynchronous clear
  component FDC
    port (
      C : in std_logic;
      CLR : in std_logic;
      D : in std_logic;
      Q : out std_logic);
  end component;

  -- Flip-flop with asynchronous preset

  component FDP
    port (
      C : in std_logic;
      PRE : 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 FDC : component is "true";
  attribute iob of FDP : component is "true";

  -- Fast off-chip output buffer, low-voltage TTL levels, 24 mA drive
  -- I is the on-chip signal, O is the pad

  component OBUF_F_24
    port (
      O : out STD_ULOGIC;
      I : in STD_ULOGIC);
  end component;

  -- Fast off-chip input/output buffer, low-voltage TTL levels, 24 mA drive
  -- T is the tristate control input, IO is the pad,

  component IOBUF_F_24
    port (
      O : out STD_ULOGIC;
      IO : inout STD_ULOGIC;
      I : in STD_ULOGIC;
      T : in STD_ULOGIC);
  end component;

  signal pb_addr_1: std_logic_vector(19 downto 0);
  signal pb_dwrite_1: std_logic_vector(15 downto 0);
  signal pb_tristate: std_logic_vector(15 downto 0);
  signal pb_dread_a: std_logic_vector(15 downto 0);
  signal we_n, pb_we_n1: std_logic;
  signal oe_n, pb_oe_n1: std_logic;
  signal lb_n, pb_lb_n1: std_logic;
  signal ub_n, pb_ub_n1: std_logic;
  signal ramce_n, ram_ce_n1: std_logic;
  signal dataz : std_logic;

begin
```

```
-- Write enable

we_n <= not pb_wr;

we_ff : FDP port map (
  C => clk, PRE => rst,
  D => we_n,
  Q => pb_we_n1);

we_pad : OBUF_F_24 port map (
  O => PB_WE_N,
  I => pb_we_n1);

-- Output Enable

oe_n <= not pb_rd;

oe_ff : FDP port map (
  C => clk,
  PRE => rst,
  D => oe_n,
  Q => pb_oe_n1);

oe_pad : OBUF_F_24 port map (
  O => PB_OE_N,
  I => pb_oe_n1);

-- RAM Chip Enable

ramce_n <= not ram_ce;

ramce_ff : FDP port map (
  C => clk, PRE => rst,
  D => ramce_n,
  Q => ram_ce_n1);

ramce_pad : OBUF_F_24 port map (
  O => RAM_CE_N,
  I => ram_ce_n1);

-- Upper byte enable

ub_n <= not pb_ub;

ub_ff : FDP port map (
  C => clk, PRE => rst,
  D => ub_n,
  Q => pb_ub_n1);

ub_pad : OBUF_F_24 port map (
  O => PB_UB_N,
  I => pb_ub_n1);

-- Lower byte enable

lb_n <= not pb_lb;

lb_ff : FDP port map (
  C => clk,
    PRE => rst,
    D => lb_n,
    Q => pb_lb_n1);

  lb_pad : OBUF_F_24 port map (
    O => PB_LB_N,
    I => pb_lb_n1);

  -- 20-bit address bus

  addressbus : for i in 0 to 19 generate
    address_ff : FDC port map (
      C => clk, CLR => rst,
      D => pb_addr(i),
      Q => pb_addr_1(i));

    address_pad : OBUF_F_24 port map (
      O => PB_A(i),
      I => pb_addr_1(i));
  end generate;

  -- 16-bit data bus

  dataz <= (not pb_wr) or pb_rd;

  databus : for i in 0 to 15 generate
    dtff : FDP port map (               -- Trisate enable
      C => clk,
      PRE => rst,
      D => dataz,
      Q => pb_tristate(i));

    drff : FDP port map (
      C => clk,
      PRE => rst,
      D => pb_dread_a(i),
      Q => pb_dread(i));

    dwff : FDP port map (
      C => clk,
      PRE => rst,
      D => pb_dwrite(i),
      Q => pb_dwrite_1(i));

    data_pad : IOBUF_F_24 port map (
      O => pb_dread_a(i),
      IO => PB_D(i),
      I => pb_dwrite_1(i),
      T => pb_tristate(i));
  end generate;

end Behavioral;
```

**vga.vhd:**

```
-------------------------------------------------------------------------------
--
-- VGA video generator
--
-- Uses the vga_timing module to generate hsync etc.
```

```vhdl
-- 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 7) <= vga_shreg (15 downto 13);
  r(6 downto 0) <= "0000000";
  g(9 downto 7) <= vga_shreg (12 downto 10);
  g(6 downto 0) <= "0000000";
  b(9 downto 8) <= vga_shreg (9 downto 8);
  b(7 downto 0) <= "00000000";

  -- 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;
```

## vga_timing.vhd

```
-------------------------------------------------------------------------------
--
-- VGA timing and address generator
--
-- Fixed-resolution address generator.  Generates h-sync, v-sync, and blanking
-- signals along with a 20-bit RAM address.  H-sync and v-sync signals are
-- delayed two cycles to compensate for the DAC pipeline.
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vga_timing is
  port (
    pixel_clock  : in std_logic;
    reset        : in std_logic;
    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));
end vga_timing;

architecture Behavioral of vga_timing is

  constant SRAM_DELAY : integer := 3;

-- 640 X 480 @ 60Hz with a 25.175 MHz pixel clock
  constant H_ACTIVE      : integer := 640;
  constant H_FRONT_PORCH : integer := 16;
  constant H_BACK_PORCH  : integer := 48;
  constant H_TOTAL       : integer := 800;

  constant V_ACTIVE      : integer := 480;
  constant V_FRONT_PORCH : integer := 11;
  constant V_BACK_PORCH  : integer := 31;
  constant V_TOTAL       : integer := 524;

  signal line_count  : std_logic_vector (9 downto 0);   -- Y coordinate
  signal pixel_count : std_logic_vector (10 downto 0);  -- X coordinate

  signal h_sync : std_logic;  -- horizontal sync
  signal v_sync : std_logic;  -- vertical sync

  signal h_sync_delay0 : std_logic; -- h_sync delayed 1 clock
  signal v_sync_delay0 : std_logic; -- v_sync delayed 1 clock
  signal h_blank : std_logic;       -- horizontal blanking
  signal v_blank : std_logic;       -- vertical blanking

  -- flag to reset the ram address during vertical blanking
  signal reset_vga_ram_read_address : std_logic;

  -- flag to hold the address during horizontal blanking
  signal hold_vga_ram_read_address : std_logic;

  signal ram_address_counter : std_logic_vector (19 downto 0);

begin

  -- Pixel counter

  process ( pixel_clock, reset )
  begin
    if reset = '1' then
      pixel_count <= "00000000000";
    elsif pixel_clock'event and pixel_clock = '1' then
      if pixel_count = (H_TOTAL - 1) then
        pixel_count <= "00000000000";
      else
        pixel_count <= pixel_count + 1;
      end if;
    end if;
  end process;

  -- Horizontal sync

  process ( pixel_clock, reset )
  begin
    if reset = '1' then
      h_sync <= '0';
    elsif pixel_clock'event and pixel_clock = '1' then
      if pixel_count = (H_ACTIVE + H_FRONT_PORCH - 1) then
        h_sync <= '1';
      elsif pixel_count = (H_TOTAL - H_BACK_PORCH - 1) then
        h_sync <= '0';
      end if;
    end if;
  end process;

  -- Line counter

  process ( pixel_clock, reset )
  begin
    if reset = '1' then
      line_count <= "0000000000";
    elsif pixel_clock'event and pixel_clock = '1' then
      if ((line_count = V_TOTAL - 1) and (pixel_count = H_TOTAL - 1)) then
        line_count <= "0000000000";
      elsif pixel_count = (H_TOTAL - 1) then
        line_count <= line_count + 1;
      end if;
    end if;
  end process;

  -- Vertical sync
```

```
process ( pixel_clock, reset )
begin
  if reset = '1' then
    v_sync <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    if line_count = (V_ACTIVE + V_FRONT_PORCH -1) and
       pixel_count = (H_TOTAL - 1) then
      v_sync <= '1';
    elsif line_count = (V_TOTAL - V_BACK_PORCH - 1) and
          pixel_count = (H_TOTAL - 1) then
      v_sync <= '0';
    end if;
  end if;
end process;

-- Add two-cycle delays to h/v_sync to compensate for the DAC pipeline

process ( pixel_clock, reset )
begin
  if reset = '1' then
    h_sync_delay0 <= '0';
    v_sync_delay0 <= '0';
    h_sync_delay  <= '0';
    v_sync_delay  <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    h_sync_delay0 <= h_sync;
    v_sync_delay0 <= v_sync;
    h_sync_delay  <= h_sync_delay0;
    v_sync_delay  <= v_sync_delay0;
  end if;
end process;

-- Horizontal blanking

-- The constants are offset by two to compensate for the delay
-- in the composite blanking signal

process ( pixel_clock, reset )
begin
  if reset = '1' then
    h_blank <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    if pixel_count = (H_ACTIVE - 2) then
      h_blank <= '1';
    elsif pixel_count = (H_TOTAL - 2) then
      h_blank <= '0';
    end if;
  end if;
end process;

-- Vertical Blanking

-- The constants are offset by two to compensate for the delay
-- in the composite blanking signal

process ( pixel_clock, reset )
begin
  if reset = '1' then
```

```
    v_blank <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    if line_count = (V_ACTIVE - 1) and pixel_count = (H_TOTAL - 2) then
      v_blank <= '1';
    elsif line_count = (V_TOTAL - 1) and pixel_count = (H_TOTAL - 2) then
      v_blank <= '0';
    end if;
  end if;
end process;

-- Composite blanking

process ( pixel_clock, reset )
begin
  if reset = '1' then
    blank <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    if (h_blank or v_blank) = '1' then
      blank <= '1';
    else
      blank <= '0';
    end if;
  end if;
end process;

-- RAM address counter

-- Two control signals:

-- reset_ram_read_address is active from the end of each field until the
-- beginning of the next

-- hold_vga_ram_read_address is active from the end of each line to the
-- start of the next

process ( pixel_clock, reset )
begin
  if reset = '1' then
    reset_vga_ram_read_address <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    if line_count = V_ACTIVE - 1 and
       pixel_count = ( (H_TOTAL - 1 ) - SRAM_DELAY ) then
      -- reset the address counter at the end of active video
      reset_vga_ram_read_address <= '1';
    elsif line_count = V_TOTAL - 1 and
          pixel_count =((H_TOTAL -1) - SRAM_DELAY) then
      -- re-enable the address counter at the start of active video
      reset_vga_ram_read_address <= '0';
    end if;
  end if;
end process;

process ( pixel_clock, reset )
begin
  if reset = '1' then
    hold_vga_ram_read_address <= '0';
  elsif pixel_clock'event and pixel_clock = '1' then
    if pixel_count = ((H_ACTIVE - 1) - SRAM_DELAY) then
      -- hold the address counter at the end of active video
```

```
        hold_vga_ram_read_address <= '1';
      elsif pixel_count = ((H_TOTAL - 1) - SRAM_DELAY) then
        -- re-enable the address counter at the start of active video
        hold_vga_ram_read_address <= '0';
      end if;
    end if;
  end process;

  process ( pixel_clock, reset )
  begin
    if reset = '1' then
      ram_address_counter <= "00000000000000000000";
    elsif pixel_clock'event and pixel_clock = '1' then
      if reset_vga_ram_read_address = '1' then
        ram_address_counter <= "00000000000000000000";
      elsif hold_vga_ram_read_address = '0' then
        ram_address_counter <= ram_address_counter + 1;
      end if;
    end if;
  end process;

  vga_ram_read_address <= ram_address_counter;

end Behavioral;
```