MAYD

MAYD Ain't Yet Doom

Final Report

Vladislav Adzic (va2030@columbia.edu)
Surag Mungekar (ssm2003@columbia.edu)
Nabeel Daulah (nud2001@columbia.edu)
George Yeboah (gdy2101@columbia.edu)

Table of Contents

1. Overview

MAYD is a three-dimensional (3D) environment based on the genre of first person shooter video games such as *Wolfenstein 3D* or *Doom*. MAYD allows the user to walk around the MAYD 3D environment, while accurately rendering the three-dimensional perspective of the user. However, MAYD lacks more complex elements of a complete game such as textures, the ability to shoot enemies, and levels.

For this project, we use the FPGA, UART, and video controller on the Xess XSB-300E board. We use a serial connection via the UART port to receive input controls from the user. We process this input and calculate each frame in a C program running on the MicroBlaze soft-core processor. Modified VGA controller VHDL code from Lab 2 is used to display each frame on the screen. We did not use a full frame buffer in memory, but rather we add some logic to the VGA controller so that much less data needs to be stored in memory.

2. Design

2.1 Raycasting

The method used to simulate the three-dimensional environment is called Raycasting. Unlike other 3D techniques, which create a 3D perspective from a 3D map, raycasting creates a 3D perspective from a 2D map. This greatly simplifies all frame calculations, but limits us to using only the same height for the entire environment. For example, it is impossible to add stairs to lead to another floor using raycasting.

2.2 Components and Connections

We implement the raycasting 3D engine in C and run the code on the MicroBlaze. Our C program processes user input from the UART port. When the engine renders a frame, frame information is written into BRAM's. The VGA controller reads the data from the BRAM's and outputs each frame to the VGA device. *Figure 1* shows how each component is connected. Components are connected to each other through the common OPB bus.

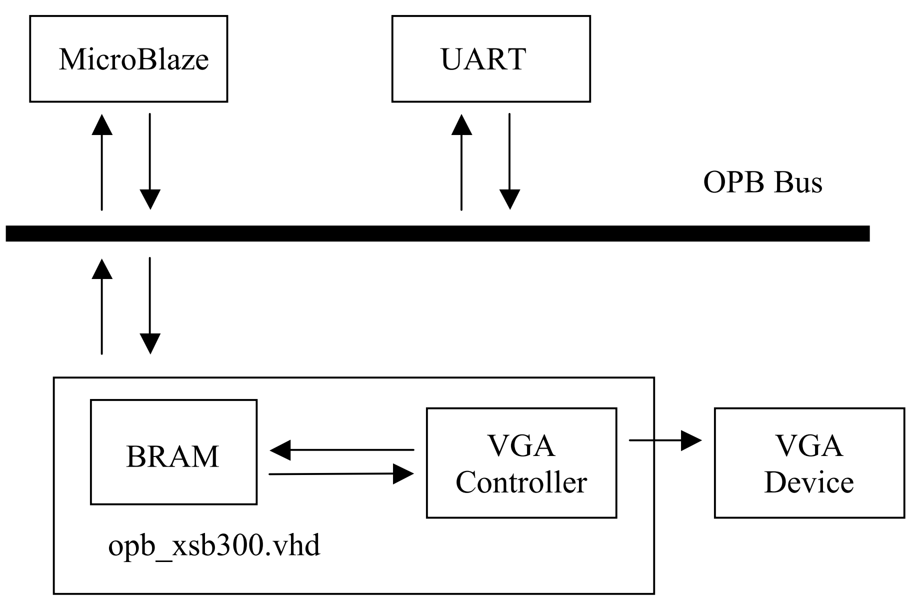

Figure 1 – Component Block Diagram

Another useful illustration is *Figure 2*, which shows the process that creates each frame. The sequence of events is triggered when the user enters a valid input key (see *Figure 3*). When such a key is received (for example, ‘w’), the corresponding action (move forward) is repeated until another key is received.

Our C program runs on the MicroBlaze processor and contains an interrupt handler that writes all user input into an input buffer. The rest of the C program consists of a large loop that first processes any input in the input buffer and then renders a new frame. This loop of processing input and rendering continues until the board is reset.

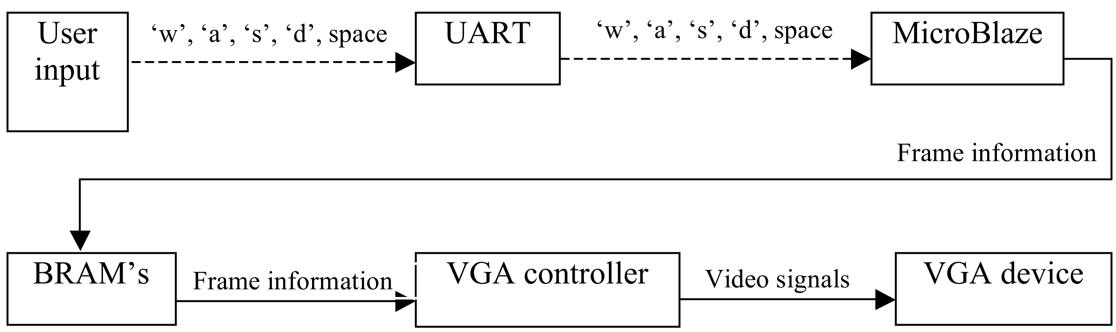

Figure 2 – Data Flow Diagram

Key	Action
‘w’	Start moving the player forward.
‘s’	Start moving the player backwards.
‘a’	Start rotating the player to the left.
‘d’	Start rotating the player to the right.
space	Stop the current action, if any.

Figure 3 – Valid Input Controls

2.3 *Limitations*

The target platform of our project is a MicroBlaze processor running on the XESS XSB-300E FPGA board. This is an embedded platform that presents us with a set of very restricting constraints.

The MicroBlaze processor does not have a native floating unit. This means that all calculations have to be performed using fixed-point arithmetic. Fixed-point arithmetic involves splitting a regular integer into a whole part and a fraction part. For example, the upper 16 bits can represent the whole number and the lower 16 bits can represent the fraction part. There are a few limitations associated with fixed-point arithmetic:

- The range of values represented of a fixed-point number must be determined beforehand.
- With a larger range of values of a fixed-point number comes decreased precision of the fractional portion. This is due to the limited number of bits that are available in an integer.

The MicroBlaze lacks a hardware multiplier and a hardware divider. This means that all multiplication and division operators do not compile into native multiply and divide assembly instructions, but are instead converted into calls to multiply and divide functions. These functions are much slower and require many clock cycles to complete. Without hardware implementations of these operations, we must reduce the number of multiplications and divisions used in our code as much as possible.

Of interesting note is that, apparently, some Spartan FPGA chips do contain a hardware multiplier. We tried enabling it by running *gcc* with the *"-mno-xl-soft-mul*" command line option. Though the generated assembly did contain the “mul” instruction, no multiplication actually occurred when tested. We confirmed that the specific FPGA we use does not contain a hardware multiplier. [i]

The MicroBlaze processor is a capable RISC processor and runs at a speedy 50MHz. However, because of the very slow multiplication and division, our code runs slower than might be expected. We hypothesize that a 386 processor running at a lower clock rate would achieve better performance, because it contains a native multiplier and divider. [ii]

3. Implementation

3.1 Raycasting

The first step to using raycasting is to define a world map. We used a 16-by-16 array of 8-bit color values. A value of zero represents open space in the map, and all non-zero values are different colored blocks. A color value of *1* designates the block as “magic block”, a special multi-colored block. Later, we will describe how the magic blocks are colored. All other (greater than *1)* values in the world map are the actual 8-bit color values that are sent to the video controller.

Before raycasting can occur, we need a few more pieces of information. The first are *posX* and *posY*, which represent the current position of the player. The values *dirX* and *dirY* are the direction in which the player is facing. Finally, *planeX* and p*laneY* is a vector perpendicular to the direction vector, and represents half of the player’s point of view. *Figure 4* shows how these numbers are related to each other.

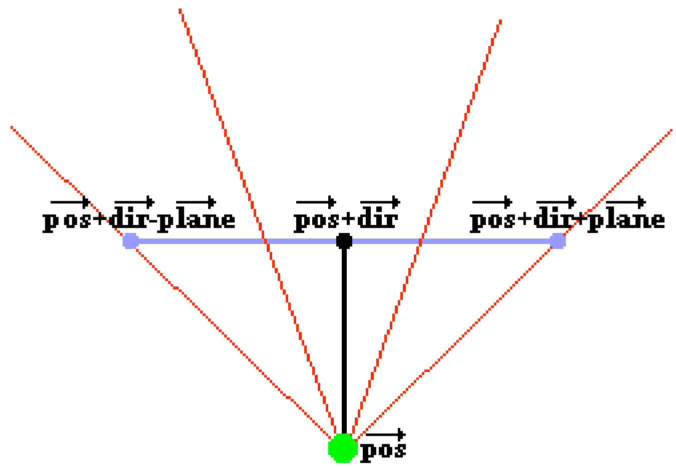

Figure 4 – Basic Raycasting Vectors [iii]

Raycasting creates 640 (one for each vertical bar on the screen) evenly spaced vectors within the player's field of vision. The left most vector is equal to *pos+dir-plane*, and the right most is equal to *pos+dir+plane*. Each of the 640 vectors (or rays), are extended through open space until a wall is reached.[iii] Each ray is extended by an amount proportional to the ray's original length, and a counter keeps track of the number of times the ray was extended until a wall was reached. *Figure 5* shows an illustration of ray extension.

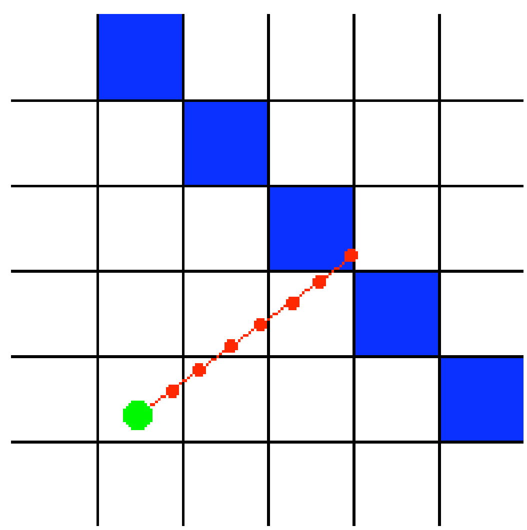

Figure 5 – Illustration of Ray Extension [iii]

Ray extension of each of the 640 rays will generate 640 counters, each proportional to the distance from the player to a wall, in the given ray direction. Finally, we divide a constant by each counter to calculate and draw the height of each vertical bar on the screen. The color of each vertical bar is the same as the color of the wall the ray hit when extended. However, to create a better looking rendering of the world, we give sides of blocks different tints. All walls parallel to the X-axis get one tint, and all walls perpendicular to the X-axis get another tint. This entire process effectively creates a three-dimensional perspective from the 2D map.

The only exception to the coloring rules is the "magic block". The color of each individual vertical bar is determined by the components of the vector that reached the magic block.

Player movement is calculated with relative ease. The actions 'move forward' and 'move backward' are accomplished easily by just moving the user along the direction that the player is facing. The actions 'rotate left' and 'rotate right' are accomplished using hard-coded values of *sine* and *cosine* of a single angle. The amount of rotation is fixed, so we do not require a complete lookup table of *sine* and *cosine* values. Most of our numbers use 4 bits -16 bits fixed-point arithmetic; the remaining bits are required to prevent overflow operations when multiplying with other numbers.

3.2 Implementation Details

After each frame is calculated using the raycasting technique it must be transported to the screen somehow. We originally used a full frame buffer in SRAM to accomplish this. However, this meant that for every frame we executed over 300,000

(640*480) writes to memory, and the video controller had to request that many read operations. This was very wasteful. We switched to using BRAM's and a much more efficient transfer system. We divide the BRAM's into three segments of memory:

- The first stores the start position of each vertical bar.
- The seconds stores the stop position of each vertical bar.
- The third stores the color of each vertical bar.

We also change the VHDL code of the video controller to access the BRAM's and act accordingly. We accomplished this using the controller's internal *Hcount* and *Vcount* signals. One problem we encountered was the fact that the *Vcount* counter counts blanking signals. This caused some complications that meant we could not extend walls into the bottom most rows of the screen.

3.3 Code Optimization

We spent great deal of time optimizing our C code. When we initially ported our C code to the MicroBlaze, it took 15 seconds to draw a single frame. Now we get frame rates of about 15 frames per second. Optimization can make a huge difference in the performance. In this section we will outline a few of the types of optimizations we made.

- Elimination of unnecessary variables. For example, we had an integer array containing indexes to another integer array. It seemed very smart at the time, but these two arrays were easily combined into one array.
- We realized every single line of code makes a difference. Even though sometimes we saved only or two lines of assembly, each of these lines are in a

loop that runs hundreds or thousands of times, so the savings accumulate very quickly.

- Why divide by 1000 when you can right shift by 10? The performance gain is huge, but sometimes the loss in precision is unacceptable.
- “Strength Reduction” – see Professor Edward’s slides.
- Sometimes the C compiler may not be as smart as you expect (but it is very smart). Change your code and see the differences that it makes using *objdump*.

4. Advice for Future Students

- **MAYD's Law of Big O:** Combing *m* groups of *n* lines of code is $O(n \log m)$. Writing *n* lines of code is $O(n^2)$ time. Think about it.
- **MAYD's Binary Law:** When feeling depressed, try debugging VHDL code. Be happy you never have to debug VHDL. Unless you do.
- **MAYD's Law of Correlated Embedded-ness:** The more embedded the platform, the more embedded the bugs.
- **Corollary to MAYD's Law of Correlated Embedded-ness:** If possible, write and debug C code on your PC, then port to the MicroBlaze.
- **MAYD's Law of Diminishing Returns:** The more optimized your code is, the more effort it will take to further optimize.
- **Corollary to MAYD's Law of Diminishing Returns:** Code can always be optimized further, but sometimes it takes infinite effort.
- **MAYD's First Law of Office Hours**: For every minute the TA's explain something to you, you will save 26 minutes of trying to figure it by yourself.
- **MAYD's Second Law of Office Hours**: For every minute the TA's explain something to you, you will first wait 26 minutes for them to get to you.
- **MAYD's Law of Crossing Language Barriers:** *objdump* is your friend; even if you don't speak the same language, you can understand a word here and there.

5. Responsibilities

- Vladislav Adzic: Lead C and VHDL Developer
- Surag Mungekar: Raycasting / Game Software / Testing
- Nabeel Daulah: Raycasting / Game Software / Testing
- George Yeboah: Game Software / Testing

We would like to thank Professor Edwards, Marcio Buss (a very dedicated TA), and Cristian Soviani for all their help.

Appendix A: Source Code

The rest of this document contains the source code of all the files that we wrote or modified.

---

[i] "MicroBlaze™ Software Reference Guide." Xilinx: The Programmable Logic Company™. 2002. Xilinx Inc. 11 May 2005 <http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/swref.pdf>

[ii] "IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture." Intel Corporation – Welcome to Intel.com. 2005. Intel Corporation. 11 May 2005 <http://download.intel.com/design/Pentium4/manuals/25366515.pdf>

[iii] "Raycasting." Lode's Computer Graphics Tutorial. 2004. Lode Vandevenne. 11 May 2005 <http://www.student.kuleuven.ac.be/~m0216922/CG/raycasting.html>

```
/*
Possible improvements:
-write frame to initial buffer as it is being generated
-after each frame is caclulated, and on sync, copy over to second buffer
-hack up a vhdl hw multiplier
-texturing via vga controller... but would have to sacle textures


-authors: vlad, surag, nabeel, george
*/


#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"
#include "xparameters.h"
#include "xuartlite_l.h"
#define R   0xE0 //red
#define G   0x1C //green
#define B   0x02 //blue
#define Y   0xFC //yellow
#define V   0xE2 //violet
#define W   0xFE //white

//sine/cosines scaled to 11 bits
#define ISINVALUE   0x00000132
#define ICOSVALUE   0x000007E9
#define ISINNVALUE  0xFFFFFECE
#define ICOSNVALUE  0x000007E9

#define SCREENWIDTH 640
#define VGA_START 0x00800000
#define HEIGHT 480
#define HEIGHTMINUS1 479
#define SCALEDHEIGHT 15360
#define HALFHEIGHT 240

extern void XIntc_DefaultHandler (void *);
extern void Xhw_InterruptAHandler (void *);
extern void microblaze_enable_icache();
extern void microblaze_enable_interrupts();
extern void microblaze_disable_interrupts();

//a circular buffer is sized to hold input data
#define BUFSIZE 2 //input character buffer size
char buffer[BUFSIZE+1];
char *lastin  = buffer;
char *lastout = buffer;

/*
 * 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 );
```

```c
    //add incoming character to buffer
    if (lastin == buffer + BUFSIZE)
      //rollover to beginning of buffer if necessary
      lastin=buffer;
    *lastin = incoming_character;
    lastin++;

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



int main()
{
    Xuint8 worldMap[16][16]= {{W,W,W,W,W,W,W,W,W,W,W,Y,Y,Y,Y,Y},
                              {W,0,0,B,B,0,0,0,0,0,0,0,0,0,0,Y},
                              {W,0,0,0,G,G,G,G,G,0,0,0,0,0,0,Y},
                              {W,0,0,0,0,0,0,0,0,0,0,0,0,0,0,Y},
                              {W,R,R,R,R,R,R,R,R,R,0,0,0,0,0,Y},
                              {W,0,0,0,0,0,G,0,0,R,0,0,0,0,0,Y},
                              {W,0,G,G,G,G,G,0,0,R,0,0,0,0,0,Y},
                              {W,0,B,0,0,Y,0,0,V,R,G,G,G,0,0,Y},
                              {W,0,B,0,0,Y,0,0,V,1,1,0,B,0,0,W},
                              {W,0,B,0,0,Y,0,0,V,0,0,0,B,0,0,W},
                              {W,0,B,0,0,Y,0,0,V,0,0,0,B,0,0,W},
                              {W,0,B,0,0,Y,0,0,V,0,0,0,B,0,0,W},
                              {W,0,B,0,0,Y,0,0,V,0,0,0,B,0,0,W},
                              {W,0,0,0,0,0,0,0,V,0,0,0,0,0,0,W},
                              {W,0,0,0,0,0,0,0,V,0,0,0,0,0,0,W},
                              {W,W,W,W,W,W,W,W,W,W,W,W,W,W,W,W}};

    char *end_of_buffer = buffer+2;
    Xuint8 draw=1;
    char c;
    char process;

    int posX   =  2<<16, posY   = 2<<16;
    int dirX   = -1<<10, dirY   = 0;
    int planeX = 0,      planeY = (43254)>>6;
    int factor;
    int oldDirX, oldPlaneX;
    int lineHeight, drawStart, drawEnd;
    Xuint8 myColor;
    Xuint32 addr = XPAR_VGA_BASEADDR;
    register int count, oldMapX=0, oldMapY, mapX, mapY, rayDirX, rayDirY;
    register int x,y, temp;

    buffer[2]='\0';
    XIntc_RegisterHandler( XPAR_INTC_BASEADDR, XPAR_MYUART_DEVICE_ID,
                           (XInterruptHandler)uart_handler, (void *)0);
    XIntc_mEnableIntr( XPAR_INTC_BASEADDR, XPAR_MYUART_INTERRUPT_MASK);
    XIntc_mMasterEnable( XPAR_INTC_BASEADDR );
    XIntc_Out32(XPAR_INTC_BASEADDR + XIN_MER_OFFSET, XIN_INT_MASTER_ENABLE_MASK);
    XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);
    print("Welcome to MAYD\r\n\r\n");
    microblaze_enable_icache();
    microblaze_enable_interrupts();


    addr = XPAR_VGA_BASEADDR +800-640 -16;

    for (;;)
```

```
    {
        //get next character
        microblaze_disable_interrupts();
        if (lastin != lastout)
        {
            c=*lastout;
            microblaze_enable_interrupts();
            if (lastout == end_of_buffer)
                lastout = buffer;
            else
                lastout++;

            if ( (c == 's') || (c == 'w') || (c == 'a') || (c == 'd'))
              process=c;
            else if (c == 32) //space
              process = 0;
        }
        else
            microblaze_enable_interrupts();

            if (process == 'w') //up
            {
                temp = dirX <<2;
                if(worldMap[(posX + temp)>>16][posY>>16] == 0)
                    posX += temp;
                temp = dirY <<2;
                if(worldMap[posX>>16][(posY + temp)>>16] == 0)
                    posY += temp;
                draw=1;
            }
            else if (process == 's') //down
            {
                temp = dirX <<2;
                if(worldMap[(posX - temp)>>16][posY>>16] == 0)
                    posX -= temp;
                temp = dirY <<2;
                if(worldMap[posX>>16][(posY - temp)>>16] == 0)
                   posY -= temp;
                draw=1;
            }
            else if (process == 'a') //left
            {
                oldDirX   = dirX;
                oldPlaneX = planeX;
                dirX   = (int)(dirX      * ICOSVALUE - dirY   * ISINVALUE)>>11;
                dirY   = (int)(oldDirX   * ISINVALUE + dirY   * ICOSVALUE)>>11;
                planeX = (int)(planeX    * ICOSVALUE - planeY * ISINVALUE)>>11;
                planeY = (int)(oldPlaneX * ISINVALUE + planeY * ICOSVALUE)>>11;
                draw=1;
            }
            else if (process == 'd') //right
            {
                oldDirX   = dirX;
                oldPlaneX = planeX;
                dirX   = (int)(dirX      * ICOSNVALUE - dirY   * ISINNVALUE)>>11;
                dirY   = (int)(oldDirX   * ISINNVALUE + dirY   * ICOSNVALUE)>>11;
                planeX = (int)(planeX    * ICOSNVALUE - planeY * ISINNVALUE)>>11;
                planeY = (int)(oldPlaneX * ISINNVALUE + planeY * ICOSNVALUE)>>11;
                draw=1;
            }

        if (draw!=1)
            continue;

        //frame calulating part
```

```
        factor=0xFFFEFFCD;

        addr = XPAR_VGA_BASEADDR +800-640 -16;
        for(x = 0; x < SCREENWIDTH; x++)
        {
            factor +=0xCD;
            //initalize
            count = 0;
            rayDirX = dirX + ((planeX * factor) >>16);
            rayDirY = dirY + ((planeY * factor) >>16);
            mapX = posX;
            mapY = posY;

            //raycasting is loads of fun!
            while ((temp = worldMap[mapX>>16][mapY>>16]) == 0)
            {
                count++;
                oldMapY = mapY;
                oldMapX = mapX;
                mapX += rayDirX;
                mapY += rayDirY;
            }

            //set color
            myColor = temp;
            if(worldMap[oldMapX>>16][mapY>>16]) myColor+=1;
            if(myColor==1) myColor = 0xFF & rayDirX & rayDirY;

            //cant get this to inline
            lineHeight = SCALEDHEIGHT / count;

            temp = lineHeight >>1;
            drawStart = HALFHEIGHT - temp;
            drawEnd   = HALFHEIGHT + temp;
            if(drawStart < 0)          drawStart = 0;
            if(drawEnd   >= HEIGHT)    drawEnd   = HEIGHTMINUS1;

            //do actual drawing
            XIo_Out8(addr+x,drawStart);
            XIo_Out8(addr+x+1024,drawEnd-240);
            XIo_Out8(addr+x+2048,myColor);
        }
        draw=0;
    }
    return 0;
}
```

```
-------------------------------------------------------------------------------
--
-- Text-mode VGA controller for the XESS-300E
--
-- Uses an OPB interface, e.g., for use with the Microblaze soft core
--
-- Stephen A. Edwards
-- sedwards@cs.columbia.edu
--
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity opb_xsb300e_vga is

  generic (
    C_OPB_AWIDTH : integer                       := 32;
    C_OPB_DWIDTH : integer                       := 32;
    C_BASEADDR   : std_logic_vector(31 downto 0) := X"FEFF1000";
    C_HIGHADDR   : std_logic_vector(31 downto 0) := X"FEFF1FFF");

  port (
    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;

    VGA_DBus       : out std_logic_vector (31 downto 0);
    VGA_errAck     : out std_logic;
    VGA_retry      : out std_logic;
    VGA_toutSup    : out std_logic;
    VGA_xferAck    : out std_logic;

    Pixel_Clock    : in std_logic;

    -- Video signals
    VIDOUT_CLK     : out std_logic;
    VIDOUT_RED     : out std_logic_vector(9 downto 0);
    VIDOUT_GREEN   : out std_logic_vector(9 downto 0);
    VIDOUT_BLUE    : 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 opb_xsb300e_vga;

architecture Behavioral of opb_xsb300e_vga is

  constant BASEADDR : std_logic_vector(31 downto 0) := X"FEFF1000";

  -- Video parameters

  constant HTOTAL : integer := 800;
  constant HSYNC : integer := 96;
  constant HBACK_PORCH : integer := 48;
  constant HACTIVE : integer := 640;
  constant HFRONT_PORCH : integer := 16;
```

```
  constant VTOTAL : integer := 525;
  constant VSYNC : integer := 2;
  constant VBACK_PORCH : integer := 33;
  constant VACTIVE : integer := 480;
  constant VFRONT_PORCH : integer := 10;

  -- 512 X 8 dual-ported Xilinx block RAM
  component RAMB4_S8_S8
    port (
      DOA   : out std_logic_vector (7 downto 0);
      ADDRA : in std_logic_vector (8 downto 0);
      CLKA  : in std_logic;
      DIA   : in std_logic_vector (7 downto 0);
      ENA   : in std_logic;
      RSTA  : in std_logic;
      WEA   : in std_logic;
      DOB   : out std_logic_vector (7 downto 0);
      ADDRB : in std_logic_vector (8 downto 0);
      CLKB  : in std_logic;
      DIB   : in std_logic_vector (7 downto 0);
      ENB   : in std_logic;
      RSTB  : in std_logic;
      WEB   : in std_logic);
  end component;

  -- Signals latched from the OPB
  signal ABus : std_logic_vector (31 downto 0);
  signal DBus : std_logic_vector (31 downto 0);
  signal RNW  : std_logic;
  signal select_delayed : std_logic;

  -- Latched output signals for the OPB
  signal DBus_out : std_logic_vector (31 downto 0);

  -- Signals for the OPB-mapped RAM
  signal ChipSelect : std_logic;            -- Address decode
  signal MemCycle1, MemCycle2 : std_logic;  -- State bits
  signal RamPageAddress : std_logic_vector(2 downto 0);
  signal RamSelect : std_logic_vector (7 downto 0);
  signal RST, WE : std_logic_vector (7 downto 0);
  signal DOUT0, DOUT1, DOUT2, DOUT3 : std_logic_vector(7 downto 0);
  signal DOUT4, DOUT5, DOUT6, DOUT7 : std_logic_vector(7 downto 0);
  signal ReadData : std_logic_vector(7 downto 0);

  -- Signals for the video controller
  signal LoadNShift : std_logic;        -- Shift register control
  signal FontData  : std_logic_vector(7 downto 0);  -- Input to shift register
  signal ShiftData : std_logic_vector(7 downto 0);  -- Shift register data
  signal VideoData : std_logic;         -- Serial out ANDed with blanking

  signal Hcount : std_logic_vector(9 downto 0);  -- Horizontal position (0-800)
  signal Vcount : std_logic_vector(9 downto 0);  -- Vertical position (0-524)
  signal HBLANK_N, VBLANK_N : std_logic;   -- Blanking signals

  signal START_POSITION : std_logic_vector(9 downto 0);
  signal STOP_POSITION : std_logic_vector(9 downto 0);

  signal START_POSITIONa : std_logic_vector(7 downto 0);
  signal STOP_POSITIONa : std_logic_vector(7 downto 0);


  signal COLOR : std_logic_vector(7 downto 0);
```

```
--  signal FontLoad, LoadChar : std_logic;  -- Font/Character RAM read triggers
--signal FontAddr : std_logic_vector(10 downto 0);
--  signal CharRamPage : std_logic_vector(2 downto 0);
--  signal CharRamSelect_N : std_logic_vector(4 downto 0);
--  signal FontRamPage : std_logic_vector(1 downto 0);
--  signal FontRamSelect_N : std_logic_vector(2 downto 0);
--  signal CharAddr : std_logic_vector(11 downto 0);
--  signal CharColumn : std_logic_vector(9 downto 0);
--  signal CharRow : std_logic_vector(9 downto 0);
--  signal Column : std_logic_vector(6 downto 0); -- 0-79
--  signal Row : std_logic_vector(4 downto 0);   -- 0-29
  signal EndOfLine, EndOfField : std_logic;

  signal DOUTB0, DOUTB1, DOUTB2, DOUTB3 : std_logic_vector(7 downto 0);
  signal DOUTB4, DOUTB5, DOUTB6, DOUTB7 : std_logic_vector(7 downto 0);

  signal msel : std_logic;

begin  -- Behavioral

  -- Port A is used for communication with the OPB
  -- Port B is for video

process(pixel_clock)
  begin
    if pixel_clock'event and pixel_clock = '1' then
      msel <= Hcount(9);
    end if;
  end process;

  START_POSITIONa <= DOUTB1 when (msel = '1') else DOUTB0;
  STOP_POSITIONa  <= DOUTB3 when (msel = '1') else DOUTB2;


  START_POSITION <= "00" & START_POSITIONa;
  STOP_POSITION <= ("00" & STOP_POSITIONa) + 240;

  COLOR          <= DOUTB5 when (msel='1') else DOUTB4;



  RAMB4_S8_S8_0 : RAMB4_S8_S8
    port map (
      DOA   => DOUT0,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(0),
      WEA   => WE(0),
      DOB   => DOUTB0,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

  RAMB4_S8_S8_1 : RAMB4_S8_S8
    port map (
      DOA   => DOUT1,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
```

```
      RSTA  => RST(1),
      WEA   => WE(1),
      DOB   => DOUTB1,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

    RAMB4_S8_S8_2 : RAMB4_S8_S8
    port map (
      DOA   => DOUT2,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(2),
      WEA   => WE(2),
      DOB   => DOUTB2,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

    RAMB4_S8_S8_3 : RAMB4_S8_S8
    port map (
      DOA   => DOUT3,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(3),
      WEA   => WE(3),
      DOB   => DOUTB3,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

    RAMB4_S8_S8_4 : RAMB4_S8_S8
    port map (
      DOA   => DOUT4,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(4),
      WEA   => WE(4),
      DOB   => DOUTB4,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

    RAMB4_S8_S8_5 : RAMB4_S8_S8
    port map (
      DOA   => DOUT5,
      ADDRA => ABus(8 downto 0),
```

```
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(5),
      WEA   => WE(5),
      DOB   => DOUTB5,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

  RAMB4_S8_S8_6 : RAMB4_S8_S8
    port map (
      DOA   => DOUT6,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(6),
      WEA   => WE(6),
      DOB   => DOUTB6,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

    RAMB4_S8_S8_7 : RAMB4_S8_S8
    port map (
      DOA   => DOUT7,
      ADDRA => ABus(8 downto 0),
      CLKA  => OPB_Clk,
      DIA   => DBus(7 downto 0),
      ENA   => '1',
      RSTA  => RST(7),
      WEA   => WE(7),
      DOB   => DOUTB7,
      ADDRB => Hcount(8 downto 0),
      CLKB  => Pixel_Clock,
      DIB   => X"00",
      ENB   => '1',
      RSTB  => '0',
      WEB   => '0');

  -----------------------------------------------------------------------------
  --
  -- OPB-RAM controller
  --
  -----------------------------------------------------------------------------

  -- Unused OPB control signals
  VGA_errAck <= '0';
  VGA_retry <= '0';
  VGA_toutSup <= '0';

  -- Latch the relevant OPB signals from the OPB, since they arrive late
  LatchOPB: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Abus <= ( others => '0' );
      DBus <= ( others => '0' );
      RNW <= '1';
```

```
      select_delayed <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then
      ABus <= OPB_ABus;
      DBus <= OPB_DBus;
      RNW <= OPB_RNW;
      select_delayed <= OPB_Select;
    end if;
  end process LatchOPB;

  -- Address bits 31 downto 12 is our chip select
  -- 11 downto 9 is the RAM page select
  -- 8 downto 0 is the RAM byte select

  ChipSelect <=
    '1' when select_delayed = '1' and
       (ABus(31 downto 12) = BASEADDR(31 downto 12)) and
       MemCycle1 = '0' and MemCycle2 = '0' else
    '0';

  RamPageAddress <= ABus(11 downto 9);

  RamSelect <=
    "00000001" when RamPageAddress = "000" else
    "00000010" when RamPageAddress = "001" else
    "00000100" when RamPageAddress = "010" else
    "00001000" when RamPageAddress = "011" else
    "00010000" when RamPageAddress = "100" else
    "00100000" when RamPageAddress = "101" else
    "01000000" when RamPageAddress = "110" else
    "10000000" when RamPageAddress = "111" else
    "00000000";

  MemCycleFSM : process(OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      MemCycle1 <= '0';
      MemCycle2 <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then
      MemCycle2 <= MemCycle1;
      MemCycle1 <= ChipSelect;
    end if;
  end process MemCycleFSM;

  VGA_xferAck <= MemCycle2;

  WE <=
  RamSelect when ChipSelect = '1' and RNW = '0' and OPB_Rst = '0' else
    "00000000";

  RST <=
   not RamSelect when ChipSelect = '1' and RNW = '1' and OPB_Rst = '0' else
    "11111111";

  ReadData <= DOUT0 or DOUT1 or DOUT2 or DOUT3 or
              DOUT4 or DOUT5 or DOUT6 or DOUT7 when MemCycle1 = '1'
              else "00000000";

  -- DBus(31 downto 24) is the byte for addresses ending in 0

  GenDOut: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      DBus_out <= ( others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
      DBus_out <= ReadData & ReadData & ReadData & ReadData;
```

```
    end if;
  end process GenDOut;

  VGA_DBus <= DBus_out;

  -----------------------------------------------------------------------------
  --
  -- Video controller
  --
  ------------------------------------------------------------------------------

  -- Horizontal and vertical counters

  HCounter : process (Pixel_Clock, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Hcount <= (others => '0');
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine = '1' then
        Hcount <= (others => '0');
      else
        Hcount <= Hcount + 1;
      end if;
    end if;
  end process HCounter;

  EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

  VCounter: process (Pixel_Clock, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Vcount <= (others => '0');
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine = '1' then
        if EndOfField = '1' then
          Vcount <= (others => '0');
        else
          Vcount <= Vcount + 1;
        end if;
      end if;
    end if;
  end process VCounter;

  EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';

  -- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK

  HSyncGen : process (Pixel_Clock, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      VIDOUT_HSYNC_N <= '0';
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine = '1' then
        VIDOUT_HSYNC_N <= '0';
      elsif Hcount = HSYNC - 1 then
        VIDOUT_HSYNC_N <= '1';
      end if;
    end if;
  end process HSyncGen;

  -- The -1 correction doesn't appear here to correct for the
  -- registered video signal outputs.

  HBlankGen : process (Pixel_Clock, OPB_Rst)
  begin
```

```
    if OPB_Rst = '1' then
      HBLANK_N <= '0';
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if Hcount = HSYNC + HBACK_PORCH then
        HBLANK_N <= '1';
      elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
        HBLANK_N <= '0';
      end if;
    end if;
  end process HBlankGen;

  VSyncGen : process (Pixel_Clock, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      VIDOUT_VSYNC_N <= '0';
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine ='1' then
        if EndOfField = '1' then
          VIDOUT_VSYNC_N <= '0';
        elsif VCount = VSYNC - 1 then
          VIDOUT_VSYNC_N <= '1';
        end if;
      end if;
    end if;
  end process VSyncGen;

  VBlankGen : process (Pixel_Clock, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      VBLANK_N <= '0';
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine ='1' then
        if Vcount = VSYNC + VBACK_PORCH - 1 then
          VBLANK_N <= '1';
        elsif VCount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
          VBLANK_N <= '0';
        end if;
      end if;
    end if;
  end process VBlankGen;

  -- Registered video signals going to the video DAC
  VideoOut: process (Pixel_Clock, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      VIDOUT_BLANK_N <= '0';
      VIDOUT_RED   <= "0000000000";
      VIDOUT_BLUE  <= "0000000000";
      VIDOUT_GREEN <= "0000000000";
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      VIDOUT_BLANK_N <= VBLANK_N and HBLANK_N;

      --color pick logic here
      if (Vcount < START_POSITION) then
        VIDOUT_RED   <= "1000000000";
        VIDOUT_GREEN <= "1000000000";
        VIDOUT_BLUE  <= "1111111111";
      elsif (Vcount > STOP_POSITION) then  --240
        VIDOUT_RED   <= "1100000000";
        VIDOUT_GREEN <= "1100000000";
        VIDOUT_BLUE  <= "1100000000";
      else
        VIDOUT_RED   <= COLOR(7 downto 5) & "0000000";
        VIDOUT_GREEN <= COLOR(4 downto 2) & "0000000";
        VIDOUT_BLUE  <= COLOR(1 downto 0) & "00000000";
```

```
      end if;
    end if;
  end process VideoOut;

  VIDOUT_CLK <= Pixel_Clock;
end Behavioral;
```