CSEE W4840 – Embedded Systems & Design
Final Project Report
[TAMF]

Date: 5/11/2004

Project Team Members:
Essa S. Farhat (esf2012@columbia.edu)
Eveliza Herrera (eh486@columbia.edu)
Rhonda L. Jordan (rlj33@columbia.edu)
Amon R. Wilkes (arw2017@columbia.edu)

Table of Contents

1. Introduction

The Spartan™-IIE Field-Programmable Gate Array (FPGA) is the main repository of programmable logic on the XSB board. The board is provided to students in the Embedded Systems CSEE W4840 course, which includes many features such as a 4 Mbit Flash RAM, a 256K x 16 SRAM, 16M x 16 SDRAM, Video DAC, Compact flash interface, a video decoder and many other features. The video decoder is a Philips SAA7114H chip that can accept up to six signals through dual RCA jacks and dual S-Video connectors. Applications of this include capturing and scaling video images to be provided as digital video stream through the image port of a VGA controller, for display via the frame buffer, or for capture to system memory[1]. The Flash RAM, SRAM, and SDRAM are used to store general–purpose data. The Video DAC generates the analog red, green, and blue signals for the VGA display while the FPGA generates the horizontal and vertical sync pulses directly. In our project we will use some of the features on the XSB board to create a video effects generator.

We will use the SRAM chip to store an image from the computer and we will use the Texas Instrument video DAC (THS8133B) to generate the video signals for a VGA display. We will have user input to control different shapes of the image. Our goal is to display an image onto the screen, horizontally compress the image about the center of the screen to different compression rates, and be able to create desired shapes: a triangle where our heads will be compressed and our feet of normal shape, an hourglass shape where we would compress inwards as we get to the middle of the screen and then decompress outwards as we get to the bottom of the screen, and be able to invert our image. Our project is entitled *TAMF*, an acronym for "Thing-a-Ma-Flipper."

2. Project Design

After several approaches to our project we finally arrived at an effective schematic. We have included below a sketch of our project layout and a timeline documenting the progress made during the course of project design.

2.1 Project Layout

Below is a schematic of our project design.

1 Philips Semiconductor SAA7114H. Data Sheet. March 14, 2000 (pg. 4)

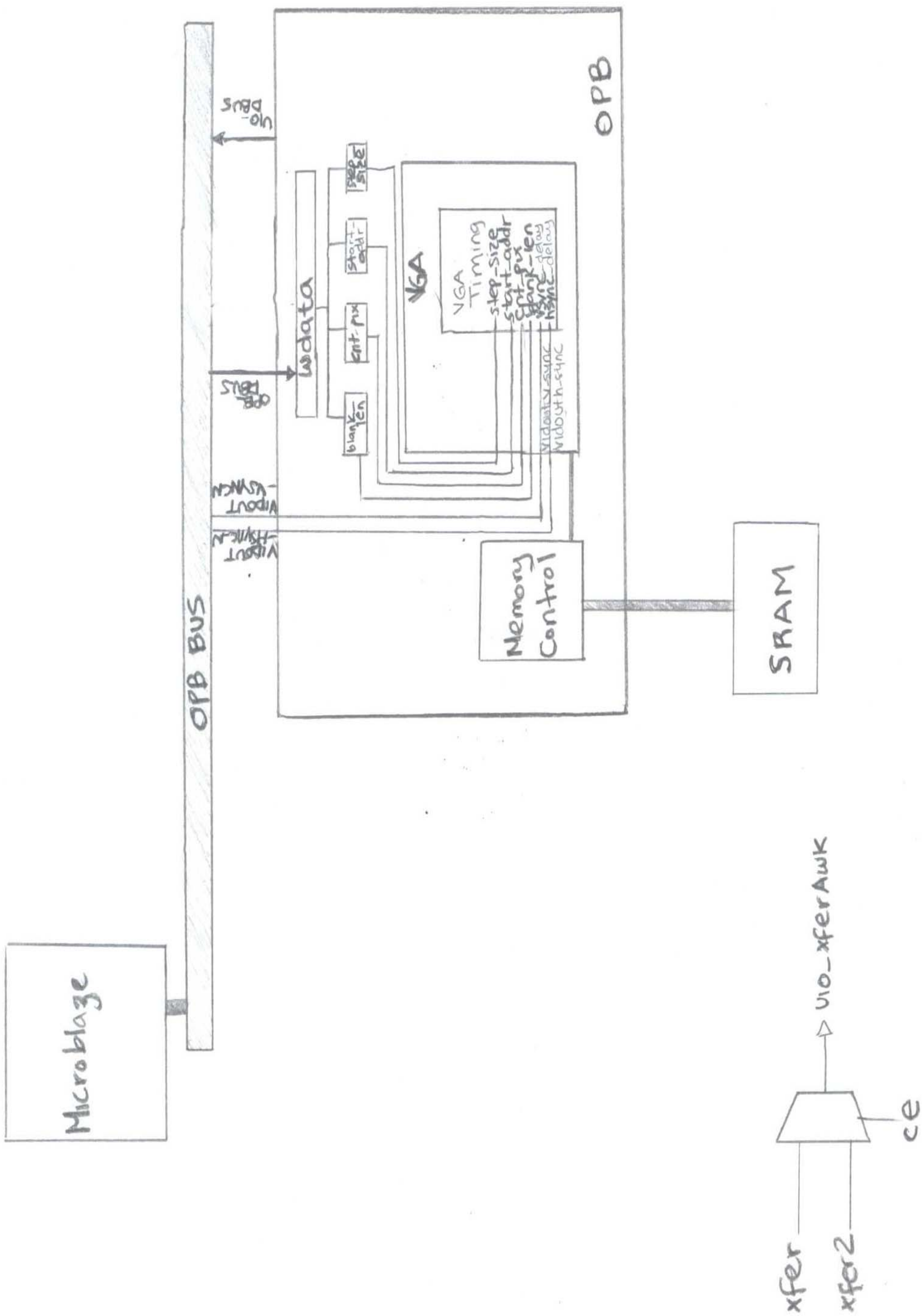
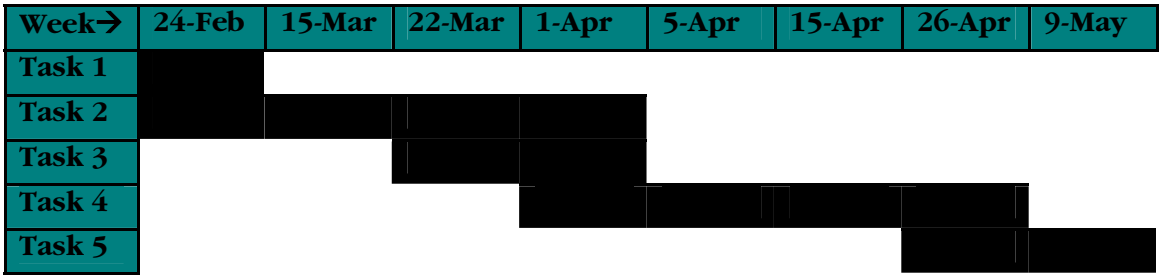

OPB BUS
Microblaze
OPB
VGA
wdata
Memory Control
SRAM
uio_xferAwk
xfer
xfer2
ce

Figure 1

2.2 Project Timeline

The following is our list of completion dates for project milestones.

Task 1: Project Proposal (due 2/24)
Task 2: Begin system implementation & Program coding design
Task 3: Detailed Project Design (due 4/1)
Task 4: Continue system implementation, system testing and debugging, and Project Demo (due 4/15)
Task 5: Project report and demo preparations



3. Image Capture and Display

The first major task of this project was to capture and display an image. The following describes in detail the steps that were taken to perform these tasks.

3.1 C-Program design

Our original hypothesis of image capture included reading the stored image information from RAM and using a C program to convert each pixel into ASCII values which will then be used to write to the screen. Our original implementation would read 1-byte images and output them as ASCII text one byte at a time. The data was then stored in 2500 byte blocks, NOT 512 byte blocks. Since the binary imagery is character data, it cannot be browsed. However, the character data can easily be converted into integer values.

Here is our original synopsis of the standard format for an image W pixels wide by H pixels high.

Example: 6 bytes of data. Three two-byte integers (high order byte first - "big endian"):

Bytes 1 -> 2: The number of Columns in the image, W. Valid values are 1-65535.
Bytes 3 -> 4: The number of Rows in the image (H). Valid values are 1-65535.
Bytes 5 -> 6: The number of colors. Valid values are 1 or 3.

The number of colors is 1 if the image is monochrome (gray scale) or 3 if the image is color. In that case there are three color planes stored in this order: Red (first), Green, Blue (last).

Bytes 7 -> (WxH + 6): Monochrome color data if the number of colors is 1.
Bytes 7 -> (WxH + 6): Red color data if the number of colors is 3.
Bytes (WxH + 6) -> (2xWxH + 6): Green color data
Bytes (2xWxH + 6) ->(3xWxH + 6): Blue color data

With any program or project design, there are always changes made during the implementation of the project which aren't realized at the time of design. Our initial program had taken an image and converted each pixel character data into sequential integer ASCII values. From there, we had taken the integer value, completely ignoring RAM, and stored the ASCII values in a ROM, similar to the task executed in Lab 3 with the font_8x8 ROM file. We then took the integer value, corresponding to the pixel value, and enabled that pixel on the screen. This process remains the same in our final program design and was implemented in VHDL.

Before the actual use of the C program we had to take a *.jpg file and convert it to a *.pnm file. A "PNM" file stands for "Portable Any Map" and is intended to cover all six variations of bit/gray/pixel maps, (PBM, PGM, and PPM) whether ASCII or binary. PNM files aren't compressed and are two-dimensional. We used a GNU Image Manipulation Program (GIMP) to read a JPG file and write out a copy of the PNM file format.

Our final C program implemented more efficient algorithm and was much more compact. The PNM file converted every pixel into a long list of ASCII integer values. Each 3 consecutive ASCII integer values represent one color bit, starting with RED. A total of 9 consecutive numbers represents all three colors: R, G, and B. The C program does byte shifting on the ASCII integer values and the output is a binary address of each set of pixel colors.

3.2 C-Code

```
#include <stdio.h>

int main()
{
  int r, g, b;
  int i;
  int color;

  /*required to start reading after the 4th line*/
  for ( i = 0 ; i < 4 ; i++ )
    while (getchar() != '\n');

  for (;;)
    {
      color = 0;

      /* fills in the lower most bits */
      if (scanf("%d\n", &r) != 1) return 1;
      if (scanf("%d\n", &g) != 1) return 1;
      if (scanf("%d\n", &b) != 1) return 1;

      r = r & 0xF8;
      g = g & 0xFC;
      b = b & 0XF8;

      r = r << 8;
      g = g << 3;
      b = b >> 3;

      color = r | g | b;

      putchar(color >> 8);
```

```
            putchar(color & 0xff);

          } //infinite for()

        return 0;
      } //end main()
```

3.3 Compilation

In the following section, the steps required for compiling the C program, converting an image from *.pnm to *.bin and finally converting the binary file to hex in order for mapping into the SRAM of the Xilinx board are listed.

Step 01	$ gcc -o convert convert.c Makes an object file, named convert
Step 02	$ ./convert Runs the file
Step 03	$ ./convert < picture_name.pnm > picture_name.bin Line to make the conversion. creates binary file named picture_name.bin
Step 04	$ ./bin2hex -a 0 <picture_name.bin> picture_name.hex Uses a supplied program to convert from binary to hex
Step 05	$ /usr/cad/xess/bin/xsload -b XSB-300E -ram picture.hex Loads the hex picture into SRAM.

Several changes were made to the Makefile to automate these steps and for required compilation within the Make:

```
SRAM_BINFILE = images/<picture_name>.bin
SRAM_HEXFILE = images/<picture_name>.hex

# To create a .hex file with data for the SRAM

$(SRAM_HEXFILE) : $(SRAM_BINFILE)
      ./bin2hex -a 0 < $(SRAM_BINFILE) > $(SRAM_HEXFILE)
```

3.4 Image Resolution

The original vga.vhd code was initialized for RGB values of 3-2-3 bits respectively, which meant that each pixel had an 8 bit value and there were two pixels per memory address. However the 3-2-3 RGB values left the picture with a very low resolution and we were unable to clearly see all of the faces in our image. For instance, the image is a picture of our system design group and one of the member's eyes were excluded from screen. We then decided to change the RGB values to a 5-6-5 scale, which would make each pixel 16 bits, thus taking up one memory address per pixel. After this modification we ran in to another problem. To represent each pixel as 16 bits we would need to use 300 Kbyte words of memory for a screen of 640 by 480 pixels. However the FPGA has only 512 Kbyte words of memory available on its SRAM. This along with the other programs that we needed to store onto the

SRAM proved to be too big for the FPGA to handle so we changed the image size to 320 by 240. These dimensions worked for our purposes; however Professor Edwards thought that the display would look better with a larger image. After a few calculations we were able to come up with a picture size that would be larger than 320 X 240 but would leave enough space in SRAM to implement high resolution (i.e. 5-6-5). This led us to the size that we are now using, 480 X 360. After deciding on a picture size, we next needed to center the image to our screen, which is larger than the picture, so that it won't be displayed in one corner. In order to center the image we simply changed the initialization of the screen display values. This was accomplished by subtracting 240 pixels from H_ACTIVE, since our active area was now only 480 pixels long, and distributing it evenly to H_FRONT_PORCH and H_BACK_PORCH. We did the same for the 180 pixels that were left over from V_ACTIVE. The final product was an image that had high resolution and was centered in the middle of the screen with a length of 480 pixels and a height of 360 pixels.

4. Image Manipulation

After the image had been converted to hexadecimal values, saved in SRAM, and printed to the screen, we next attempted to manipulate our image. It was our goal to initially compress the image and then produce interesting shapes with the image, such as triangular shapes, hour-glass shapes, etc. However, we ran into a few difficulties and made the following attempts to make our project a success.

4.1 Compression: Attempt #1

We began by tweaking the RAM address generator in the vga_timing.vhd file to increment by two pixels instead of one, with the hope that the image would compress. However, this did not work the way in which desired: the image compressed; however, two of the same compressed images were displayed side by side.

4.2 Compression: Attempt #2

After our first attempt and failure to compress our image, we consulted Prof. Stephen Edwards, who suggested creating a C program that generates four important values: blank_len, step_size, start_addr, and cnt_pix.

- blank_len : represents blank length, i.e. the number of pixels on each line of the image that will be skipped before pixels are displayed. This value one word long.

- step_size : represents step size. This value is essential to compression as it describes the increment of addresses between displayed pixels on each line. For example, if step_size is 1, we display image information from addresses 1, 2, 3…. If step_size is 2, we display image information from addresses 1, 3, 5,… This value two words long.

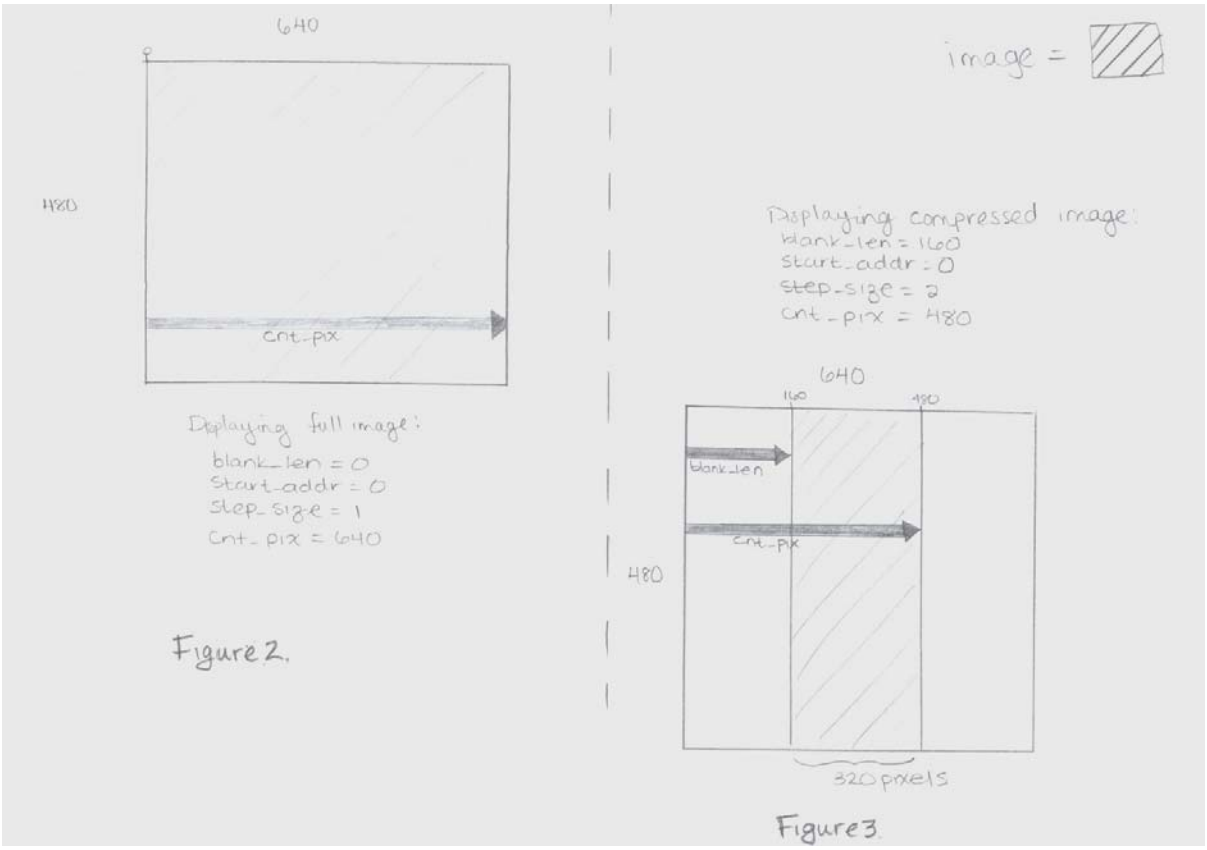
- start_addr : represents start address. For the purposes of this project, this value will always be zero and for each line will increment by 480. This value is two words long.

- cnt_pix : represents the last address on a single line of pixels at which image information will be displayed. This value is one word long.

Example 1: screen size is 640 x 480; image size 640 x 480
blank_len is 0, step_size is 1, start_addr is 0, and cnt_pix is 640

Example 2: screen size is 640 x 480; image size 320 x 480
blank_len is 160, step_size is 2, start_addr is 0, and cnt_pix is 480

Figure 2 and Figure 3 below are visual illustrations of our four golden values and the above examples.



These four "golden values" will be generated by the C program and sent to specific addresses in SRAM (not occupied with image information). In vga.vhd, we called/accessed these addresses to save the golden values in a shift register, and then retrieved these values from the shift register (in order to process a single line of pixels) all in one cycle during horizontal blanking so that during the rise of the next clock cycle we could retrieve the four values for the next line of pixels; this was to be done until every line of the image was processed.

The following code was used to implement the aforementioned concepts:

Process 1: if rst = '1', we want sel ='0'; otherwise, as soon as blanking begins, sel = '1' for six consecutive cycles in order to read only six words

```
process (pix_clk,rst)
begin
  if rst = '1' then
    sel <= '0';
  elsif pixel_count = (H_ACTIVE + H_FRONT_PORCH - 1) then
    sel <='1';
  elsif pixel_count = (H_ACTIVE + H_FRONT_PORCH – 1 + 6) then
    sel <='0';
  endif;
end process;
```

Process 2: if rst = '1' we reset info_addr to zero. If rst = '0' & sel = '0' we want info_addr = INIT_ADDR. Otherwise, we want to increment info_addr by 1 and step up to the next address

```
process (pix_clk,rst)
begin
  if rst = '1' then
```

```
    info_addr <= X"00000";
  elsif sel = '0' then
    info_addr <= INIT_ADDR;
  elsif sel = '1' then
    info_addr <= info_addr + 1;
  end if;
end process;
```

Process 3: When sel = '1', we'd like to read from the addresses in which the six words (four golden values) are stored. Otherwise, we want to continue reading form the image

```
video_addr <= vga_ram_read_address (19 downto 0) when sel = '0'
      else info_addr;
```

Process 4: The following process will load the data read of the six words (four golden values) into a register named info_reg whenever sel = '1'. Otherwise it will load the information to the default register, vga_shreg.

```
process (pix_clk)
begin
  if pix_clk' event and pix_clk = '1' then
    if sel = '0' then
      if load_video_word = '1' then
      end if;
    end if;
  else
    info_reg <= video_data;
  end if;
end process;
```

Process 5: The following process reads from info_reg at every clock fall and assigns the information to six words (the four golden values).

```
process (pix_clk)
begin
  if pix_clk' event and pix_clk = '0' then
    if pixel_count = H_ACTIVE + H_FRONT_PORCH – 1 + 1 then
      start_addr (31 downto 16) <= info_reg;
    if pixel_count = H_ACTIVE + H_FRONT_PORCH – 1 + 2 then
      start_addr (15 downto 0) <= info_reg;
    if pixel_count = H_ACTIVE + H_FRONT_PORCH – 1 + 3 then
      step_size (31 downto 16) <= info_reg;
    if pixel_count = H_ACTIVE + H_FRONT_PORCH – 1 + 4 then
      step_size (15 downto 0) <= info_reg;
    if pixel_count = H_ACTIVE + H_FRONT_PORCH – 1 + 5 then
      blank_len (15 downto 0) <= info_reg;
    if pixel_count = H_ACTIVE + H_FRONT_PORCH – 1 +1 then
      cnt_pix (15 downto 0) <= info_reg;
    end if;
  end if;
end process;
```

Below is Figure 4 and 5, which show info_addr connectors and timing diagrams respectively.

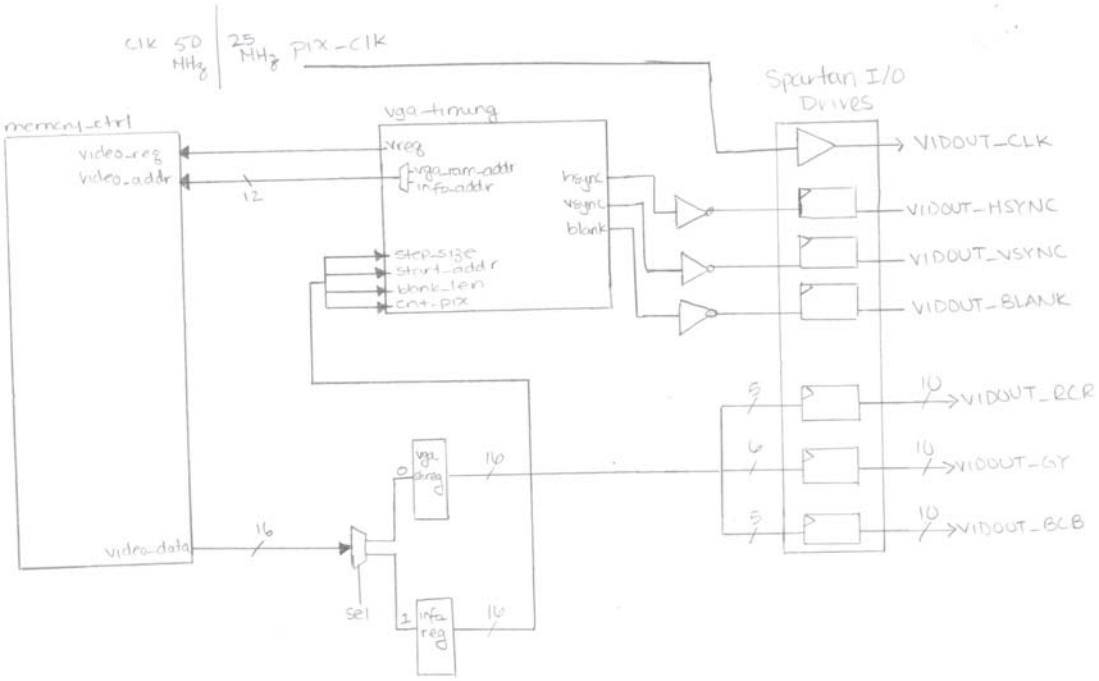


Figure 4.

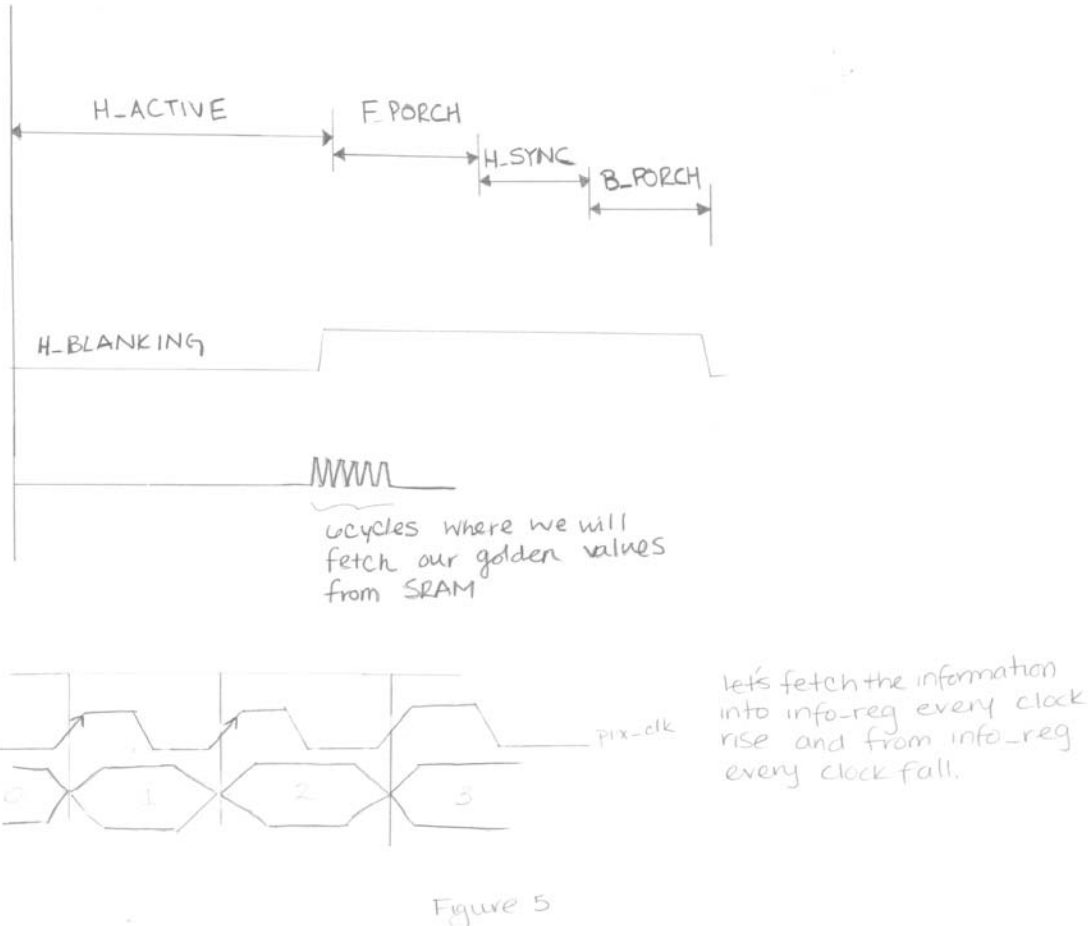


Figure 5

One of the main problems that we were having when testing these processes was that we did not know which addresses to write to and read from in SRAM. We thought that we would be

able to calculate how much space the picture is taking in SRAM and then determine available addresses to write to. After a few attempts at working this problem out and not getting any results, Prof. Edwards suggested we use the first few addresses to store these values even though they would be overwriting the picture. We changed our C-code so that we were sending in blue, red, green and white pixels to the SRAM, we then tried to read from these first few addresses and display these pixels on the screen. This worked perfect, therefore we knew for a fact that we can read from and write to the SRAM. Since this was working properly we decided to move on and test our processes. In order to test our processes we did it one step at a time. First, in vga.vhd, we directly inserted constants into our four golden variables. This worked, however because we were only able to input constants, we weren't able to increment start_addr, therefore we were only displaying one line of pixels over and over. In order to prove that it was really working the correct way we chose start_addr to begin reading at the center of our image so that we will be able to open the image on the side and verify that this line of pixel is correct. We were able to verify the one line of pixel that it was using and so we went on to feeding different constants and making the line of pixels compress to different sizes. This all worked properly therefore we confirmed that this process was correct. We used the method that Prof Edwards discussed in class for debugging a problem. We started at one point and if that worked then we moved back a few steps and tried again until we found where our problem was. Our next step was testing to see if info_reg was able to hold a value, and if our variables were able to read that value. We sent in a constant to info_reg, instead of sending in video_data, and we then read in from info_reg to one of our variables, and this did work, therefore we found where one of our problems was occurring. We knew that it had to take in the constant and so the only way that the variables would not be able to read from info_reg once the processes ran was if 'sel' was never setting to '1'. We then moved up one step where 'sel' was being generated to see why 'sel' was never setting to 1. 'Sel' relied on pixel_count so it took us a while to figure out why this would cause a problem. We coded it so that whenever pixel_count reached the point on the screen where horizontal blanking occurred, select would switch to 1. After a while we realized that we had made a small error, we realized that we never made a connection of pixel_count between vga.vhd and vga_timing.vhd!

4.3 Compression: Attempt #3

After attempting to make the connection of pixel_count between vga_timing.vhd and vga.vhd and making no progress, we decided to ask Marcio Buss to help us. Marcio pointed out that our problem lied in making pixel_count an 'out' port in vga_timing.vhd and a 'signal' in vga.vhd. The problem with this is that we were generating pixel_count in vga_timing.vhd and we used operations such as "pixel_count <= pixel_count +1", which is not allowed if pixel_count is of mode 'out'. We decided to leave pixel_count as a signal as it was before and create a new port called pix_cnt, which at the end was used to take the value of pixel_count (pix_cnt <=pixel_count;). We then used pix_cnt to communicate this value to vga.vhd. Once we fixed this we were still having problems with info_reg. There were a lot of things that could have gone wrong; therefore Marcio suggested that, instead of working this problem out in hardware, we should manipulate it in software.

We decided that it would be easier; we would not have to deal with choosing a correct address to start saving our four golden values. We decided to have the microblaze send these values directly to the OPB bus instead of saving them onto SRAM. We started implementing this idea by going into the opb_xsb300.vhd file and creating a state machine that would control a select and enable signal that would tell it when we are writing to it from the C program. We also made a multiplexer which depended on the chip enable that we created and was able to send back a transfer acknowledgment allowing it to keep reading and writing. We then created four registers called blank_len, start_addr, cnt_pix, and step_size. These four registers are connected to vga.vhd and allow the values to run through vga.vhd into vga.vhd_timing, where they are then being used to generate the ram address counter. In the OPB there is a signal called wdata which holds any information transmitted from the OPB_Bus. Wdata is of size 32 and so we used wdata to transmit information to the four registers at different times. We wanted to make sure that our connections were working

properly; we therefore fed in direct values to our golden variables and it worked perfectly. Our next step was to test whether we were able to read and write into OPB. We did this by instructing the C-Program (main.c) to send specific values to blank_len, start_addr, step_size, and cnt_pix. After playing around with these two files we determined that all connections were working. We were able to continuously display the same image, and we used minicom to print out the values of the variables being read from OPB. Once we were certain that everything was working properly, we moved onto the next step, which involved synchronizing the C-Program with the vhd files.

Once again Marcio was there to help and make this synchronization work. Hsync is generated in vga.vhd_timing so we had to bring it in from vga.vhd_timing through vga.vhd and through the OPB into the C-Program. This allowed us to be able to access the full image by increasing start _addr at the end of each line. It wasn't too difficult to get the H_Sync working since it was simply making proper connections and using it accordingly in the C-program. Once we got the horizontal and vertical sync working we were able to display the full image.

At this point, the only task left to execute was fixing and improving our C-program to generate values that would result in displaying our image in interesting shapes. The C-program had to be tweaked so that it can generate different values for our four golden variables. We were able to change the values directly in the C-program to compress the image to any size desired. However we were unable to create different functions to create multiple shapes.

4.4 Final Attempt

After several hours of attempting to create arrays and different functions to calculate our golden values we were becoming a little worried that we would not accomplish our goal as the deadline was a day away. At approximately 12:30am on Sunday night, we were able to get a hold of Christian to take a look at our C-code. He realized immediately what was wrong with our code. At one point in our project making we had edited the Makefile, and had commented some lines out. We never uncommented them and so the c-code was never getting uploaded to the memory. It was a careless mistake but once we uncommented those lines things starting flowing smoothly and very fast. We created structures that held 3 significant values, blank_len, cnt_pix and step_size. We defined two arrays that will do two different calculations. One will do the calculations for our horizontal compression and the triangle shape, and the other array will calculate the values necessary to create an hourglass shape. The triangle shape was not difficult to implement, all we had to do was make sure that we changed cnt_pix, blank_len and step_size everytime that start_addr was changed. By 3am that night (less than 3 hrs) we had the program finished. We created a user interface where the user would type in 'H' for horizontal compression, 'T' for the triangle shape, and 'G' for the hourglass shape. We made sure that it was not case sensitive, and created a menu where the user would be informed of the keys that they can use. As default, if the user were to input a letter not specified in the menu then we would just set the full image on the screen. Everything worked perfect and what is most impressive is that our horizontal compression not only compresses it, but it inverts the picture as well.

5. Conclusion

The project was a success. We were able to not only compress the image and make it invert, but to make other shapes with our image. The user interface was useful to making our project more organized and clear. We would like to thank Christian for helping us to clear our program last minute and therefore allowing us to complete our project. All our constraints were met and we are extremely pleased with our results. Our long hours in the lab and the great effort that we each put into this project paid of very well.

6. Lessons Learned

This project and the time elapsed during this project has been very interesting. This was the first semester that any of our group members had seen VHDL. Our knowledge of the hardware language was very limited. However, we made a great deal of improvement during the first few weeks of working with the lab. We learned a great deal on our own, or harassed Prof Edwards, Christian Soviani, or Josh Mackler with questions. We all made a great effort in trying to understand what was needed to make our project work, and also as a while attempted to understand more of VHDL itself. CSEE 4840, Embedded System Design, taught by Prof. Stephen Edwards, was very interesting; our group simply hoped, however, that we had a stronger background in computer organization and VHDL so that we may have been a little more creative with the final capstone project. It was a very important step when we all agreed on working on a video display project; as Prof. Edwards warned, it would be much easier to debug as most problems and issues with the code would present themselves visually.

At the final stages of project design and implementation, our group has learned a great deal. One of the most important aspects of group projects is who you are working with. We were very fortunate in that we had such wonderful team members. Everyone worked just as hard and everyone made put forth equal effort into our project. We were all understanding of each other's workloads and worked around each other's schedules. We all were essentially on the same level in terms of our knowledgeability of C and VHDL, and this worked out well; we did not have one c -programmer or one person doing most of the VHDL, we all worked just as hard in all areas and this helped for all of us to gain a better understanding of the programming languages and a better understanding of the overall project in itself. We were open to each other's ideas and never rejected any suggestions or thoughts; and we always made time to discuss new thoughts and, if necessary, attempted to implement the ideas.

We learned that a team requires organization and structure at an early beginning. It is very important to schedule mandatory weekly meeting times in order to make progress and to lay down the fundamentals for the language. It won't be correct to say that by having a structurally solid administrative and cordially functional team that there won't be problems they will face later during the project. Even a functionally sound team will run into unplanned issues, which arise due to unforeseen circumstances (i.e. issues with compilation, grammar definitions, scheduling, etc.). This semester has been by far the most difficult for all of us and has become a real test of time management.

This capstone project has been a tremendous learning experience like no other. Besides the typical time management, communication, and project planning details the group had to work out, there were other major issues that arose during the course of working on this project. One important idea and practice that should always be executed when working on a project, especially for embedded system design, is to regularly draw block diagrams. It is imperative that one understands each and every one of the numerous interconnections of various .vhd, .mhs, .c, and .opb files. Before this semester, project requirements only involved one or two parts. However, in this project, there are many different parts and blocks that are interconnected. Initially, we felt we could attempt this project the way we attempted the labs of the course: we came up with a method or idea and simply began to type. However, in this project, the block diagrams were not only conducive to a visual understanding of what was happening; the diagrams helped us conceptually and also reminded us of connections that should have been made but were forgotten. By the end of this project, if one of the team members had an idea and couldn't create a block diagram, it felt as if we were trying to blindly implement a process. Block diagrams are good!!

It should be noted that, just as it is important for one to use block diagrams to understand the interconnections of various files, it is also imperative to take a few steps *before* you actually begin this final project. Throughout the first half of CSEE 4840, students were given six labs to complete. The majority of the assignments involved tweaking and editing only one of the

program files to fulfill the requirements given by Prof. Edwards. However, in order to run the entire program, there existed numerous files that students didn't even have to touch let alone open. Nevertheless, it is extremely important to review and become familiar with all of the files, such as the Makefile, .opb, and .mhs files, not only the .c and .vhd files. Inevitably, there came the time when these files had to be understood to even begin to efficiently design the capstone project.

Lastly, this project has allowed for a great deal of learning and experience. We definitely learned that things don't turn out the way planned, no matter how much preparation is given. This idea is clearly demonstrated by the fact that our project proposal and detailed project design describing in detail the architecture of our project, both hardware and software, is *nothing* like this final report. We made four completely different attempts to implement our TAMF. It involved persistence and a great deal of patience. We learned to always print out copies of code. It is important to have copies of your code at different times in the process and comment on what your output is like. There were a few instances in which we deleted code or Christian advised us to delete some processes, and for reference we would want to go back to it and not remember what we had. This is also helpful in documenting progress.

Finally, we learned to not be afraid of utilizing other brains. At first we were nervous about jumping into this project so quickly and so clueless; however we did a great job dealing with it. After a great deal of harassing, a lot of emailing, and hours and hours in the lab performing trial and error, we were able to accomplish most of our goals. We found that the key is to ask questions. Often times our classmates were able to put us five steps ahead by simply answering a single question of ours. For instance, Marcio Buss was able to help us a great deal. He was always willing to answer questions, even when he was working with his own project, and he ended up helping multiple groups with no hesitation.

As demonstrated above, our group has learned a great deal and working on this project has taught lessons that can be applied in various instances throughout the rest of our lives. After the many, many hours and nights of working on our beloved TAMF, our group would like to make one suggestion: it would be wise in the future to have comfortable pillows in the lab for the long nights in which students will want to take short 15 minute naps :-).

7. Code Modules

The following sub-sections contain the main source codes for our final project.

7.1 Makefile

```
# Makefile for CSEE 4840, Final Project -TAMF

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_CODE_BINFILE = implementation/sram.bin
SRAM_CODE_HEXFILE = implementation/sram.hex

SRAM_BINFILE = images/group2_320X240.bin
SRAM_HEXFILE = images/group2_320X240.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
         rm -r xst

sysclean : rm implementation/system.ngc
         rm implementation/xsb300_wrapper.ngc
         rm -rf xst
```

```
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_LINKER_SCRIPT =
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) : $(SRAM_BINFILE)
	./bin2hex  -a 0 < $(SRAM_BINFILE) > $(SRAM_HEXFILE)

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

# Download the files to the target board

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

7.2 convert.c

```
#include <stdio.h>

int main()
{
  int r, g, b;
  int i;
  int color;

  for ( i = 0 ; i < 4 ; i++ )
    while (getchar() != '\n')
      ;

  for (;;)
    {
      color = 0;

      /* fills in the lower most bits */
      if (scanf("%d\n", &r) != 1) return 1;
      if (scanf("%d\n", &g) != 1) return 1;
      if (scanf("%d\n", &b) != 1) return 1;

      r = r & 0xF8;
      g = g & 0xFC;
      b = b & 0XF8;

      r = r << 8;
      g = g << 3;
      b = b >> 3;
```

```
      /*Make color 16 bits by combining RGB with an OR statement*/
      color = r | g | b;

      putchar(color >> 8);
      putchar(color & 0xff);

    }

  return 0;
}
```

7.3 main.c

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

/* // defined in isr.c */
extern void uart_handler(void *callback);
extern int uart_interrupt_count;
extern char uart_character [256];

#define W 480
#define H 360
#define VGA_START 0x00800000
#define RED 0xF800
#define GREEN 0x07E0
#define BLUE 0x001F
#define NOT_VERT_SYNC (XIo_In32(0x01800014))
#define VERT_SYNC     (!XIo_In32(0x01800014))
#define NOT_HORZ_SYNC (XIo_In32(0x01800010))
#define HORZ_SYNC     (!XIo_In32(0x01800010))


/*
 * setup_interrupts: Initialize the interrupt sources and handlers
 *
 * Should be called once when the system starts
 *
 * The main _interrupt_handler() function from Xilinx
 *7
 * 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);
}

//Creating our Structure

typedef struct {
  int blank_len;
  int step_size;
  int cnt_pix;
} line_info;

line_info lines[H];
line_info lines2[H];


//This computes all the major operations required to calculate our necessary
//values.

void compute_linfo()
{
  int i;
  int npix;

  for(i=0;i<H;i++){

    npix = i;

    lines[i].blank_len = (W-npix)/2;
    lines[i].step_size = (W<<12) / (npix +1);
    lines[i].cnt_pix= npix + (W-npix)/2;
  }

  npix = H;
  for(i=0;i<H;i++){

    if(i <= 180)
      npix--;
    else
      npix=i;

    lines2[i].blank_len = (W-npix)/2;
    lines2[i].step_size = (W<<12) / (npix +1);
    lines2[i].cnt_pix= npix + (W-npix)/2;

  }
}

// This function will create a triangle shape where our heads are compressed
// and our bodies are semi-normal

void Tri_comp()
{
  char c;
  int i, x;

  int start_addr;

  int j =0;
  int nl;



  while(1)
```

```
    {
      while(VERT_SYNC);
      if(j == 500)
          return;
      nl=-88;
      start_addr = -480*90;

      while(1)
          {
            while(NOT_HORZ_SYNC);

            // Blank length
            XIo_Out32(0x01800000, lines[nl].blank_len);

            // Start address
            XIo_Out32(0x01800004, start_addr);

            // Step size
            XIo_Out32(0x01800008, lines[nl].step_size);

            // Cnt pix
            XIo_Out32(0x0180000C, lines[nl].cnt_pix);

            start_addr += 480;
            nl ++;

            if (VERT_SYNC)
              break;

            while(HORZ_SYNC);
          }//second while
      j++;
    }

}

//This function will take our image and form it into an hourglass shape

void Hour_Glass()
{
  char c;
  int i, x;

  int start_addr;

  int j =0;
  int nl;

   while(1)
    {
      while(VERT_SYNC);
      if(j == 500)
          return;
      nl=-89;
      start_addr = -480*90;

      while(1)
          {
            while(NOT_HORZ_SYNC);

            // Blank length
            XIo_Out32(0x01800000, lines2[nl].blank_len);

            // Start address
            XIo_Out32(0x01800004, start_addr);

            // Step size
            XIo_Out32(0x01800008, lines2[nl].step_size);

            // Cnt pix
            XIo_Out32(0x0180000C, lines2[nl].cnt_pix);

            start_addr += 480;
            nl ++;

            if (VERT_SYNC)
              break;
```

```
            while(HORZ_SYNC);
          }//second while
      j++;
    }

}

//This will show the full image with normal dimensions

void Full()
{
  int j=0;
  int start_addr;
  int i = 1<<12;

  while(1)
    {
      while(VERT_SYNC);
      if(j==10)
          return;
      start_addr = -480*90;

      while(1)
          {
            while(NOT_HORZ_SYNC);

            // Blank length
            XIo_Out32(0x01800000, 3);

            // Start address
            XIo_Out32(0x01800004, start_addr);

            // Step size
            XIo_Out32(0x01800008,i);

            // Cnt pix
            XIo_Out32(0x0180000C, 480 );

            start_addr += 480;


            if (VERT_SYNC)
              break;

            while(HORZ_SYNC);
          }//second while
      j++;
    }
}

/* This will horizontally compress our image and once it reaches
   the center of the screen it will begin to invert and once again go back
   to its original size
*/

void Horz_comp()
{
  char c;
  int i, x;

  int start_addr;

  int j =0;
  int nl;

  x=0;
  nl=H ;
  while(1)
    {
      while(VERT_SYNC);



      /*compresses*/
      if(j  < H-1){
          nl--;
```

```
          start_addr = -W*90;
          i=lines[nl].step_size;
      }
      /*start expanding*/
      else if(j < 2*H  ){
          start_addr = -W*90 +W;

        i= - lines[nl].step_size;
          nl++;
      }
      /*delay and return*/
      else if (j<3*H ){
          nl--;
          start_addr = -W*90 + W  ;
          i=-lines[nl].step_size;

          //return;
      }  else{
          nl++;
          start_addr = -W*90;
          i=lines[nl].step_size;
          if(nl==H+1)
            return;
      }


      while(1)
          {
            while(NOT_HORZ_SYNC);

            // Blank length
            XIo_Out32(0x01800000, lines[nl].blank_len);

            // Start address
            XIo_Out32(0x01800004, start_addr);

            // Step size
            XIo_Out32(0x01800008, i);

            // Cnt pix
            XIo_Out32(0x0180000C, lines[nl].cnt_pix);


              start_addr += W;


            if (VERT_SYNC)
              break;

            while(HORZ_SYNC);
          }//second while

      j++;
    }

}


int main()
{
  char *prt_char;
  char *read_char;
  int j;


  // Enable the instruction cache: makes the code run 6 times faster
   microblaze_enable_icache();

  print("Hello and welcome to our final project!\r\n");
  print("Here is our MENU: \r\n");
  print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\r\n");
  print("~ To compress and invert us please enter 'H' ~\r\n");
  print("~ To see us in an hourglass please enter 'G' ~\r\n");
  print("~ To see us in a pyramid please enter 'T'    ~\r\n");
  print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\r\n");
  print("Please note our program is NOT case sensitive.\r\n");
  print("What would you like to display?");
```

```
  j =0;


  compute_linfo();

   setup_interrupts();

  for (;;) {

     prt_char = &uart_character[j];


    //We will disable the interrupt for a short while
    microblaze_disable_interrupts();
    read_char = &uart_character[uart_interrupt_count%256];
    microblaze_enable_interrupts();

    if(read_char == prt_char){ // buffer is not empt
          Full();
    }

    /*handles input from minicom*/
    if(read_char != prt_char){ // buffer is not empty

      /*if h is pressed will begin Horz_comp() */
      if ( *prt_char=='t' || *prt_char=='T'){
          Tri_comp();

      }//end if

      /*or if t  is entered, then makes a triangle form of the image */
      else if (*prt_char=='h'|| *prt_char=='H'){
          Horz_comp();

      }
      /*or if t  is entered, then makes a triangle form of the image */
      else if (*prt_char=='g'|| *prt_char=='G'){
          Hour_Glass();
      }


      /* otherwise, print to the screen */
      else{
          Full();

      }

      j=(j+1)%256;

    }//if

  }//for
  return 0;
}
```

7.4 vga.vhd

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

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

    blank_len  : in std_logic_vector(15 downto 0);
    start_addr : in std_logic_vector(31 downto 0);
    step_size  : in std_logic_vector(31 downto 0);
    cnt_pix    : in std_logic_vector(15 downto 0);

    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

  constant H_ACTIVE             : integer := 480;
  constant H_FRONT_PORCH        : integer := 96;
  constant H_BACK_PORCH         : integer := 128;
  constant H_TOTAL              : integer := 800;


  -- 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;
      blank_len                 : in std_logic_vector(15 downto 0);
      start_addr                : in std_logic_vector(31 downto 0);
      step_size                 : in std_logic_vector(31 downto 0);
      cnt_pix                   : in std_logic_vector(15 downto 0);
      pix_cnt                   : out std_logic_vector(10 downto 0));
  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);

  signal RAG_counter            : std_logic_vector(19 downto 0);
  signal pixel_count            : std_logic_vector(10 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,
    blank_len => blank_len,
    start_addr => start_addr,
    step_size => step_size,
    cnt_pix => cnt_pix,
    pix_cnt => pixel_count
    );

  vreq <= '1';

  -- 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;


  video_addr <= vga_ram_read_address(19 downto 0);


  process (pix_clk)
  begin
    if pix_clk'event and pix_clk='1' then
      vga_shreg <= video_data;
    end if;
  end process;


  -- RGB 5-6-5

  r(9 downto 5) <= vga_shreg (15 downto 11);
  r(4 downto 0) <= "00000";
  g(9 downto 4) <= vga_shreg (10 downto 5);
  g(3 downto 0) <= "0000";
  b(9 downto 5) <= vga_shreg (4 downto 0);
  b(4 downto 0) <= "00000";

  -- 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;
```

7.5 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
-- Modified by : E. Farhat, E. Herrera, R.Jordan, A. Wilkes
--------------------------------------------------------------------------------

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);
    blank_len                  : in std_logic_vector(15 downto 0);
    start_addr                 : in std_logic_vector(31 downto 0);
    step_size                  : in std_logic_vector(31 downto 0);
    cnt_pix                    : in std_logic_vector(15 downto 0);
    pix_cnt                    : out std_logic_vector(10 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 := 480;
  constant H_FRONT_PORCH : integer := 96;
```

```
  constant H_BACK_PORCH  : integer := 128;
  constant H_TOTAL       : integer := 800;

  constant V_ACTIVE      : integer := 360;
  constant V_FRONT_PORCH : integer := 71;
  constant V_BACK_PORCH  : integer := 91;
  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
  signal myblank : std_logic;

  -- 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 (31 downto 0);

  signal active_begin   : std_logic;


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
      blank <= h_blank or v_blank or myblank;
    end if;
  end process;

--ram address generator

  active_begin <='1' when pixel_count = (blank_len - SRAM_DELAY) else '0';

  process (pixel_clock, reset )
  begin
    if reset ='1'  then
      ram_address_counter<=X"00000000";
    elsif pixel_clock'event and pixel_clock ='1' then
     if active_begin ='1' then
       ram_address_counter(31 downto 12) <= start_addr(19 downto 0);
       ram_address_counter(11 downto 0)  <=  X"000";
     else
       ram_address_counter <= ram_address_counter + step_size;
     end if;
    end if;

  end process;

  process (pixel_clock, reset )
  begin
    if reset ='1'  then
      myblank <= '0';
    elsif pixel_clock'event and pixel_clock ='1' then
      if pixel_count = blank_len  then
        myblank <= '0';
      elsif pixel_count = cnt_pix then
        myblank <= '1';
      end if;
    end if;
  end process;


  vga_ram_read_address <= ram_address_counter(31 downto 12);
  pix_cnt <= pixel_count;

end Behavioral;
```

7.6 system.mhs

```
# Essa Farhat
# Eveliza Herrera
# Rhonda Jordan
# Amon Wilkes

# System.mhs file for Final Project - TAMF

# Parameters
PARAMETER VERSION = 2.0.0

# Global Ports

# Signals of opb_xsb300 module
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

#Signals for video decoder I2C Bus
#PORT VID_I2C_SCL = VID_I2C_SCL, DIR = INOUT
#PORT VID_I2C_SDA = VID_I2C_SDA, DIR = INOUT

# 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

PORT OPB_Clk = sys_clk
```

```
#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 = 0x000007FF
 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
```

7.7 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
-- Modified by E. Farhat, E. Herrera, R. Jordan, A. Wilkes
--------------------------------------------------------------------------------

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;

  -- Added
  signal cs2, q2, q1, q0, ce, xfer2 : std_logic;
  signal blank_len, cnt_pix : std_logic_vector(15 downto 0);
  signal start_addr, step_size : std_logic_vector(31 downto 0);
  signal horz, vert, hsync_n, vsync_n : std_logic;
  signal data_bus, data_bus_ce, data_bus_rce : std_logic_vector(31 downto 0);

  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;

      -- Added
      blank_len  : in std_logic_vector(15 downto 0);
      start_addr : in std_logic_vector(31 downto 0);
      step_size  : in std_logic_vector(31 downto 0);
      cnt_pix    : in std_logic_vector(15 downto 0);

      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,

      -- Added the follwing 4
      blank_len  => blank_len,
      start_addr => start_addr,
      step_size  => step_size,
      cnt_pix    => cnt_pix,

      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 => hsync_n,
      VIDOUT_VSYNC_N => 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,
    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';

cs2 <= OPB_select when OPB_ABus(31 downto 16) = X"0180" 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;

process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst='1' then
    horz <= '0';
  elsif OPB_Clk'event and OPB_Clk='1' then
    horz <= hsync_n;
```

```
  end if;
end process;

process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst='1' then
    vert <= '0';
  elsif OPB_Clk'event and OPB_Clk='1' then
    vert <= vsync_n;
  end if;
end process;


process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    data_bus_ce <= X"00000000";
  elsif OPB_Clk'event and OPB_Clk = '1' then
    if ce='1' and rnw='1' then
      if addr(4 downto 0)="00000" then
        data_bus_ce <= X"0000" & blank_len;
      elsif addr(4 downto 0)="00100" then
        data_bus_ce <= start_addr;
      elsif addr(4 downto 0)="01000" then
        data_bus_ce <= step_size;
      elsif addr(4 downto 0)="01100" then
        data_bus_ce <= X"0000" & cnt_pix;
      elsif addr(4 downto 0)="10000" then
        data_bus_ce <= X"0000000" & "000" & horz;
      elsif addr(4 downto 0)="10100" then
        data_bus_ce <= X"0000000" & "000" & vert;
      end if;
    else
      data_bus_ce <= X"00000000";
    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
    data_bus_rce(15 downto 0) <= X"0000";
  elsif OPB_Clk'event and OPB_Clk = '1' then
    if rreset = '1' then
      data_bus_rce(15 downto 0) <= X"0000";
    elsif (rce1 or rce0) = '1' then
      data_bus_rce(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
    data_bus_rce(31 downto 16) <= X"0000";
  elsif OPB_Clk'event and OPB_Clk = '1' then
    if rreset = '1' then
      data_bus_rce(31 downto 16) <= X"0000";
    elsif rce0 = '1' then
      data_bus_rce(31 downto 16) <= rdata(15 downto 0);
    end if;
  end if;
end process;

process (OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk='1' then
    q2 <= (not q2 and q1) or (q2 and not q1);
    q1 <= (cs2 and not q2 and not q1) or (q2 and not q1);
    q0 <= q2 and not q1;
  end if;
```

```
  end process;

  ce <= (q2 and not q1) or (q0) ;

  xfer2 <= q0;

  --writing into our 4 registers

  process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst='1' then
      blank_len <= X"0000";
      start_addr <= X"00000000";
      step_size <= X"00000000";
      cnt_pix <= X"0000";
    elsif OPB_Clk'event and OPB_Clk='1' then
      if ce='1' and rnw='0' then
        if addr(3 downto 0)= "0000" then
          blank_len <= wdata(15 downto 0);
        elsif addr(3 downto 0)= "0100" then
          start_addr <= wdata;
        elsif addr(3 downto 0)="1000" then
          step_size <= wdata;
        elsif addr(3 downto 0)="1100" then
          cnt_pix <= wdata(15 downto 0);
        end if;
      end if;
    end if;
  end process;

  VIDOUT_HSYNC_N <= hsync_n;
  VIDOUT_VSYNC_N <= vsync_n;

  -- unused outputs

  data_bus <= data_bus_ce when ce='1' else data_bus_rce;

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

  UIO_xferAck <= xfer when ce='0' else xfer2;

end Behavioral;
```