Tower Defense

Design of Embedded Systems-Advanced course

2012-10-26

Robin Palmblad
dt08rp0@student.lth.se

Emma Hilmersson
pi07eh2@student.lth.se

Niclas Thuning
et06nt3@student.lth.se

Abstract

This report is about implementing the game Tower Defence on a FPGA Nexys 3 board. Tower Defence is a classic game where the players' goal is to prevent enemies from destroying the players base by building towers that kills the enemies.

There is "no standard hardware on an FPGA". You mean the Wizard generated system.

Modules have to be added to the standard hardware on the FPGA. The game logic needs a timer and a keyboard controller. The timer updates the game in a specific time interval and the keyboard controller interrupts the game when a key is pressed.

The game is displayed on a screen over VGA. To do this a VGA controller is used which is connected through a bus to the Microblaze. The VGA controller is also connected to a background generator, a foreground generator and a mouse displayer which decides what to show on the screen.

Table of Contents

1 Introduction

This report describes how to implement the game Tower Defense with a Nexus 3 FPGA. Tower Defense is a strategy game, where the player builds towers to kill the enemy, who are trying to destroy your base.

The final architecture is almost as the initial proposal. The main different is that a keyboard is used instead of a mouse. The Microblaze was also connected to the BRAM, to be able to update the matrix of the gameboard which are saved here.
Due to lack of time the audio controller was never implemented.

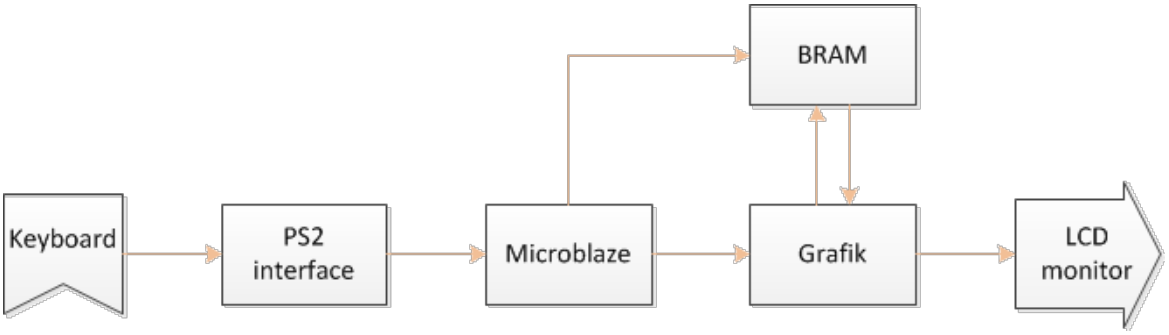


Figure 1: Block diagram of the implementation

The game logic is implemented in software. A player controls the game with a keyboard and the game logic reacts to the keystrokes through interrupts. The game logic is updated with timer polling and at each polling the software sends information about what to display on the monitor. This data is sent to hardware through the PLB and are used in BRAM and the graphic accelerator. The graphic accelerator are responsible of displaying the background, the foreground and the cursor. What to be displayed in the foreground are decided by the gamelogic and this is stored in BRAM.

game logic

2 Hardware solution

The main structure of the hardware consists of a Microblaze, graphic accelerator and a memory which contains the board games output. The difference components are connected via a bus and internal signals. All the communication through the Microblaze is done by the bus which is connected to different registers on the graphic accelerator

Memory mapped access to the accelerator?

and the memory.

2.1 Graphic accelerator

To make the graphics on the monitor to work, a vga controller is needed. Besides the vga controller the color to the monitor needs to be controlled. In this solution there is a background generator, foreground generator and a mouse displayer

.

By putting background generator, foreground generator and mouse displayer after each other the RGB output can be controlled. The components are placed after its priority in which layer the output should be. The bottom layer consists of the background and on the top of background the foreground should be placed. The last layer is the mouse displayer which should always be visibly on the screen.

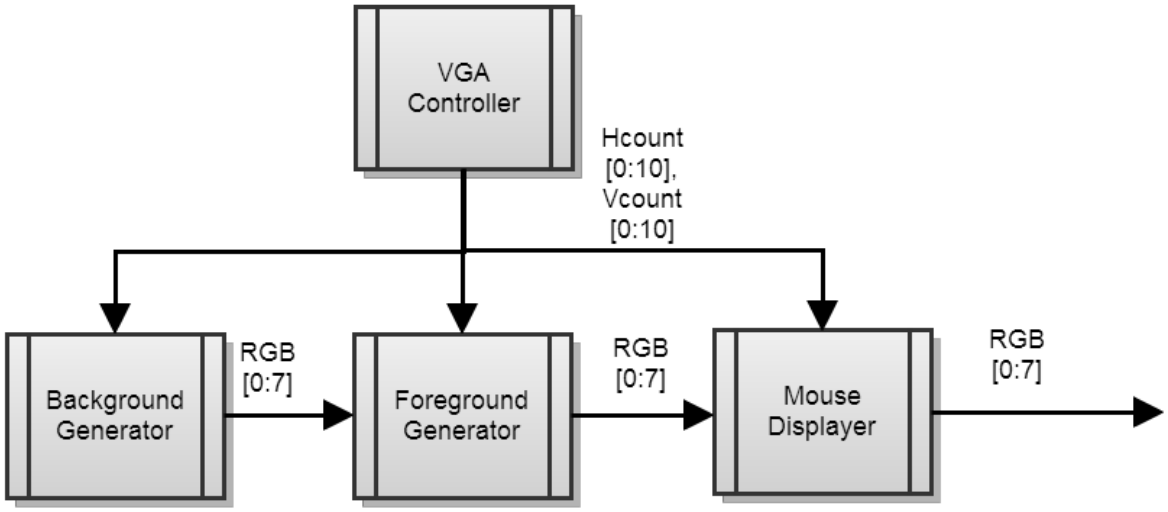


Figure 2: Block diagram of the graphics accelerator

2.1.1 VGA controller

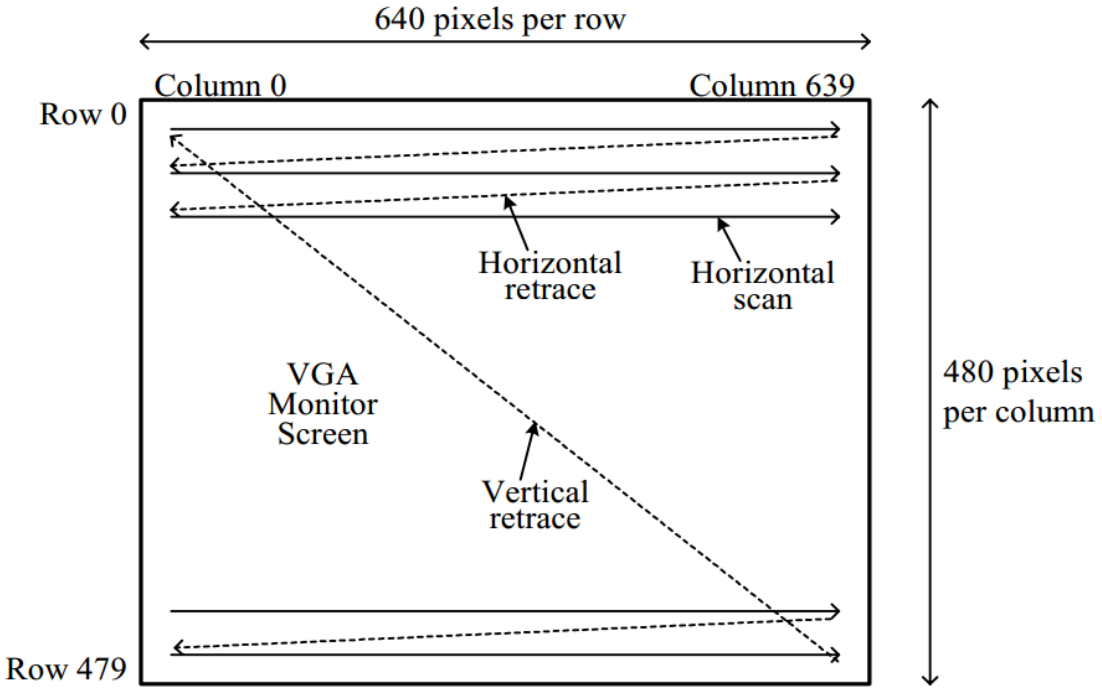
Before sending the RGB color to the monitor it's important to know when it's possible to send the RGB color. The VGA controller in this project was already finished[1] and implemented into the hardware. The behavior of the controller is described below.

It's only during 640 cc (clock cycles) of 800 cc on horizontal line it's possible to write to the VGA output. The reason for that is because the horizontal scan needs to retrace and restart to prepare for the next scan of horizontal line. For the vertical line it's the same principle but instead of cc it has hc (horizontal cycles). Because the vertical line has to retrace and prepare for the next frame. The vertical line has 480 hc in the range

of 528 hc for one frame.

The solution for knowing when to write is done by a signal called blank, which goes low when it's possible to write. To know which pixel to write to two signals called horizontal sync (HS) and vertical sync (VS) are generated. These signals keeps track of current horizontal and vertical position of the pixel the corresponding scan currently is representing.

The monitor needs to update with a frequency of 60 frames per second to avoid flicker. A clock with 25 MHz is enough for this demand. This clock is synchronized with the background generator, the foreground generator and the mouse displayer for synchronization of the RGB outputs.



You should always give the source of your pictures, if you borrowed them from somewhere!

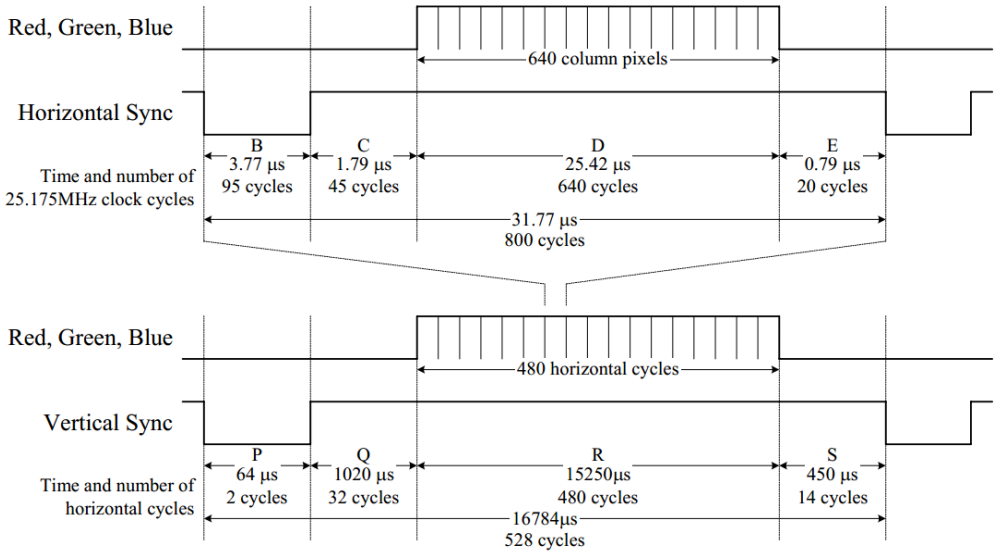


Figure 3: Demonstration on how the monitor scan works

2.1.2 Background generator

To generate the background a background generator is used. The background is never changed and therefore the contains of how the background should be displayed is stored in a ROM memory.

The background is built up by tiles of 16x16 pixels. Tiles of 30x30 builds up the left side of the background which contains the grass and path. The remaining right side is gray. The gray area is displaying the menu, score and lives. Here it’s also a tower which the user has to choose for the tower to be placed.

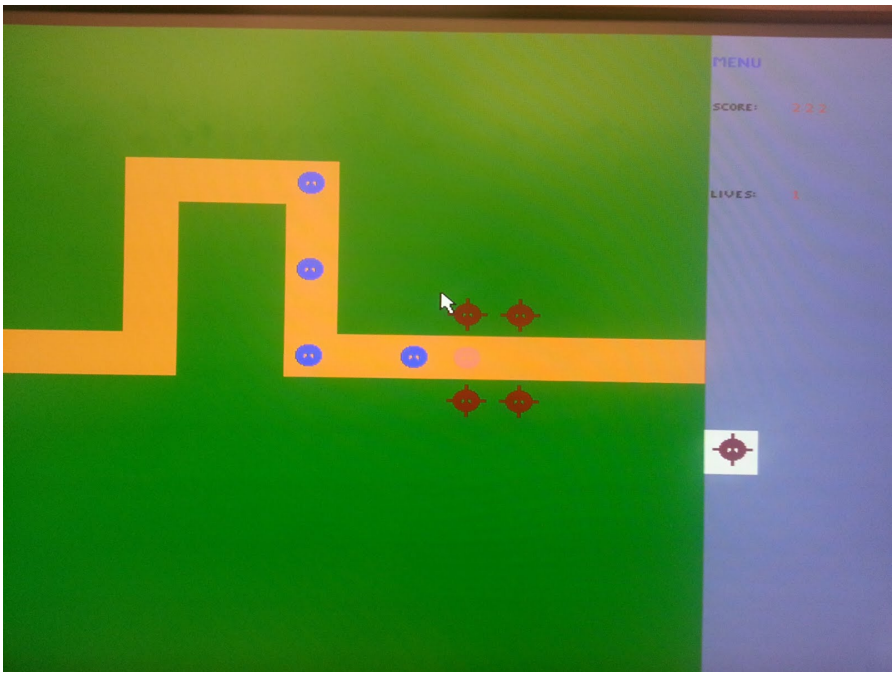


Figure 4: A screenshot of the game

When storing the information of the background in the ROM it will be stored like a long vector. By choosing each horizontal line of 32 tiles the technic of how to get the correct tile would be easier. The last 2 tiles are just overwrite by the gray area. By using the number 32 a shift operation could be done easily for the calculation of the tile. For each vertical line to by display an integer division of 16 of the vertical line is done, the same is done for the horizontal line. Then by increasing the number by a multiplication of 32 for the vertical line and then add the number of which horizontal line to be displayed the exact tile in the long vector of the ROM is used.

```
xdiff := "0000" & hcount(10 downto 4);
ydiff := "0000" & vcount(10 downto 4);
```

By shifting with 4 bits and only use the MSB:s we get a division of 16 and an integer as result. This is done for placing the correct pixel in right tile.

Romaddress = "(ydiff (4 downto 0)" & "xdiff(4 downto 0)"

The equation above show how to calculate which ROM address to be read. By shifting ‘ydiff’ by 5 bits a multiplication of 32 is done.

Ok! Detailed enough description of how the background works.

Now the background knows what to be displayed and it will then send the RGB color for the object further into the foreground generator.

2.1.3 Foreground generator

The responsibility of the foreground generator is to display enemies, towers, score and lives. The foreground generator gets its information of enemies and towers from a BRAM memory. The method for displaying and get the information is the same as in the background generator but with some modification. Instead of a ROM it uses a BRAM (Game board memory) and the enemies and towers contains of 32x32 pixel so the modulo is changed to 32.

Are you still using tiles to hold information? Or is it a list of objects or a fixed array of objects?

The graphics of how an enemy and a tower looks like is stored in a ROM memory. The shape of the objects is done by having '1' for color and '0' for transparent.

The score and lives are stored into a register. The register is updated from the Microblazer via the bus. By using a binary to BCD(binary-coded decimal) component the correct number can be displayed. The shape of each number from zero to nine is stored into a ROM and has, as the tower and enemy, information about if the pixel should be transparent or not.

If the foreground discovers during the scan of the monitor that it should display something it will replace the colors from the background and turn it into its own, in other case the colors from the background just passes by the foreground generator. The foreground generator sends the color to the mouse displayer. This is all done pixel by pixel.

2.1.4 Mouse displayer

The last step is to display the mouse over the foreground and its output signals, red, green and blue (RGB), are connected directly to the pin at the FPGA.
The mouse displayer was taken from Digilents website[2] and just modified to be able to change the cursor when a tower was selected.

References are usually annotated by square brackets: [2]

BTW: You might want to learn LATEX, which is a great help when writing reports like this.

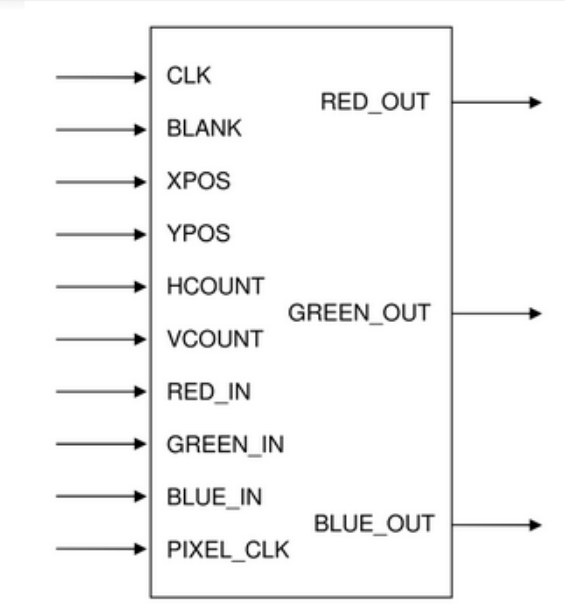


Figure 5: The component of the mouse displayer

XPOS and YPOS, which are the positions of the mouse, are taken from a register, sent from software. HCOUNT, VCOUNT and BLANK comes from the VGA controller. The RGB in signals are from the foreground generator.
A mouse will be display if the players don't decide to place a tower, then the cursor will change to look like a small tower until the tower is placed on the screen.

The pictures of the mouse and the tower are stored in a ROM memory and are 16x16 pixels. The difference of HCOUNT and VCOUNT and the position of the mouse, will decide the address of the ROM. For storing the picture of the mouse 2 bits are used, where "00" means black, "01" is white and the others are transparent, which means that the RGB signals in will be the output.

2.2 Game board memory

This memory is generated from Xilinx, when creating a new peripheral and has been modified to have two ports, which are connected to the foreground generator.
A vector of 256 representing a 16x16 matrix of 2 bits is stored in the memory and the addresses are updated from the Microblaze, via PLB. The matrix corresponds to the game board and each address contains information of what to shown on the screen. The input signal from the foreground generator decides the read address and the output from this module gives the foreground information about what to display.

2.3 PS2 controller

The PS2 controller only had to be added since it could be found in Xilinx Platform Studio when creating the hardware. An interrupt controller was added and connected to the PS2 controller to be able to handle when keys are pressed on the keyboard. The external ports L12 and J13 are connected to the PS2 controller representing the keyboard.

2.4 Device Utilization Summary

The table below show how the hardware resources are used. Because we didn't have time to implement real picture in BRAM the memory usage is small.

	Used	Available	% utilization
Slices LUTS	6395	9112	70%
Slice Registers	5685	18224	31%
Memory	202	2176	9%

Table 1: Device Utilization Summary

When you talk about hardware is customary to also give the clock frequency, and the type of FPGA used (in this case).

3 Software solutions

To put the game together the software consists of a timer for updating the game, a keyboard for reading inputs from the player, the game logic and sends information to registers and memory for deciding what to display on the screen. The software is written in C-programming language and writes values to the hardware. The hardware then reads the values as bits, therefore a common representation of the values in software and hardware had to be decided.

3.1 Timer

For updating the game in a specific time interval a timer is used. At each timer interval code is executed in the game logic and a reset of the timer is performed. A timer has to be added to the hardware to be able to use a timer in the software.

3.2 Keyboard

A keyboard is used as game controller and an interrupt controller handles keys pressed on the keyboard. The interrupt controller moves the cursor when pressing ‘w’, ‘a’, ‘s’ and ‘d’. ‘k’ is a multifunctional button and ‘l’ is used for resetting the game. The cursor has two representations, one for the game logic and one for the mouse displayer for removing unnecessary conversions. To change the appearance or position of the cursor the program writes to a register.

3.3 Game logic

I do not quite understand here: Do you poll the timer? How does that work in order to keep a natural flow of the object movement? Since you use interrupts on the keyboard why not interrupts on the timer as well?

The game logic binds everything together. It updates the game each time polling is done on the timer. At each update the program can finish the game by stopping the timer polling, change the state of enemies and update the menu. Every time something is to be changed on the display the program writes to the BRAM memory what to remove or what to display. The board is defined as a 16x16 matrix where every element in the matrix corresponds to a square of 32x32 pixels and the last row and column is just for making the calculations easier in the hardware. To update the matrix with new values two matrixes are used and one of them is always the new instance and the other one is the current instance.

3.4 Registers & memory

The program sends information to registers and memory to be able to decide what to show on the display. To be able to send information the program includes drivers for registers (VGA) and memory. The program can then use functions to write to registers and memory. The information is sent in form of integers represented as hex or decimal and is read by the hardware as bits.

3.4.1 Registers

For deciding cursor position, cursor appearance, score and lives, registers are used. The cursor position is decided by register 0(reg0) where the first half of the bits is the x-position and the other half is the y-position. When sending ones to reg1 the cursor changes into a tower and when sending zeros the cursor changes into a mouse cursor. Score and lives are placed in reg2 where the first half of the bits represents lives and the other half represents score.

3.4.2 Memory

The memory is used in another way than the registers. Instead of sending information to the same place as with the registers, when sending information to the memory it is decided where on the memory the information will be sent. The memory is represented as a long vector, to decide where on the memory to send the information the matrix position is converted to a vector position and the data is just sent as a short number representing a type.

4 Installation manual

Nexys

To run the game a keyboard and a VGA monitor has to be connected to a Nexus 3 FPGA board and the file download.bit has to be downloaded to the board. To move the cursor the keys ‘a’,’w’,’d’ and ‘s’ are used and ‘k’ is used for clicking, either to buy a tower or to place it. ‘l’ is used for restarting the game.

5 Problems

5.1 Hardware

The major problem during this project was to understand how the Xilinx program worked and how to add new components. This problem occurred because a new version was used in this course compared with the previous course and we had no other experience with Xilinx. After some time this was figured out, mostly by trying and asking other groups and the teacher.

The initial plan was to control the game with a mouse, but this led to a lot of problems. First,
because of a new board with USB input instead of PS2, led to that the mouse controller found at Diligent website couldn’t be used. After failure trying to modify this controller the inbuilt PS2 controller were used and sent the signals directly to software. But the software couldn’t get any interrupts from the mouse so it was determined that a Keyboard would be used instead.

5.2 Software

The keyboard and the timer changes and updates the game. The first approach to solve this was to use interrupt for both the timer and the keyboard. This never got to work with a common interrupt controller for both even though a solution from another project was tested. To work around this problem polling was used on the timer instead of interrupt.

Interrupts caused another problem when reading from the keyboard in small intervals. The bytes read places in a queue so that the cursor will move in the previous direction instead of the actual one and not move enough steps when pressing keys fast. The interrupts were handled slow because there was too much code in the interrupt handler. To solve this as much code as possible was moved outside of the handler.

Ok! Good

6 Lessons learned

The lessons learned during working with this project were how to divide the work and that it's important that the whole group doesn't focus on the same problem. In the beginning of this project all tried to get the mouse to work, which led to that nothing else was done. It would have been better if one focused on the mouse problem and the others focused on moving on.

Much time was spent on learning the new Xilinx which led to a better understanding of Xilinx and how to use it.

7 Contributions

For this project we divided the work into two main parts, hardware and software. Due to that one of the project member already had good knowledge of c-programming and the other two in VHDL-programing, we could easily divide the work. We decided to do more effort on the hardware because we were two person coding in VHDL.

All the c-programming was done by Robin. For the hardware, Niclas and Emma did everything by splitting the work up and also collaborate in most parts. All were responsible for testing and debugging the program.
The software part in this report is written by Robin, Emma wrote about the mouse displayer, the game board memory and the problems in the hardware. The vga

controller, background generator and the foreground generator is written by Niclas. The rest of the report was done together.

8 Reference

[1] VGA controller reference component: http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2

[2] PS2 Mouse Displayer reference component: http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2

Good work, in the end! You had some difficulties on the way but after a good deal of work, you came through. Nicely done, even if more improvements would have make the game better.