FPGA Tank Game

(EDA385 project report)

Shuai Cheng, soc11sch@student.lu.se
Longyang Lin, soc11lli@student.lu.se
Yuhang Sun, soc11ysu@student.lu.se

October 26, 2012

Contents

1 Introduction

This report presents a system that allows you to play a tank game on VGA monitor with keyboard control. The system is developed in Digilent Nexys3 FPGA board with *Xilinx platform studio* 14.2 and *Xilinx software development kits*.

The idea of the game is to let the player control a tank to protect the base and destroy the enemy tanks which is controlled by computer and they will try to attack the player's tank and base. The player can move the player tank left, right, up and down in the screen by pressing the related key on the keyboard. The player can also press one key to make the player tank fire to destroy the enemy tanks. If the player tank or base is hit by the enemy bullet, the player lose and the game will restart again from level 1. There are different topographic of background and they will have different effects when hit by the tanks and bullets.

The design is built with a single processor PLB base system using *Xilinx MicroBlaze* and automatically generated by *Xilinx platform studio*. In order to reduce memory usage and achieve the real time display, a graphical accelerator is designed and implemented. All the graphics used in this game are built by blocks of 32x32 pixels. The background and objects are built up by grouping these blocks together in different ways. The *MicroBlaze* is responsible of calculating the screen position of these blocks and the graphics accelerator does the actual drawing to the screen.

This report is organized as follows: Section 2 gives specification of this project. In section 3 the FPGA hardware is detailed. Section 4 presents how the *MicroBlaze* software is implemented. The problems and their solutions during the implementation are shown in section 5. Conclusions and discussions are given in section 6. Section 7 and section 8 list the lessons learned and contributions of the team members.

2 Specifications

The project should be able to run in the following conditions:

- Hardware platform: Digilent Nexys3 Spartan-6 FPGA Board
- Keyboard control and VGA monitor display
- Screen resolution is 640x480 pixels at 60 Hz
- Update the player tank position and fire bullets by pressing the keys
- Update the enemy tank position and fire bullets by software AI
- Detect and handle collision between player and topographic, enemy and topographic, player and enemy, bullet and topographic, bullet and player, bullet and enemy

The system overview should maybe show that you are using the PLB bus for the peripherals.

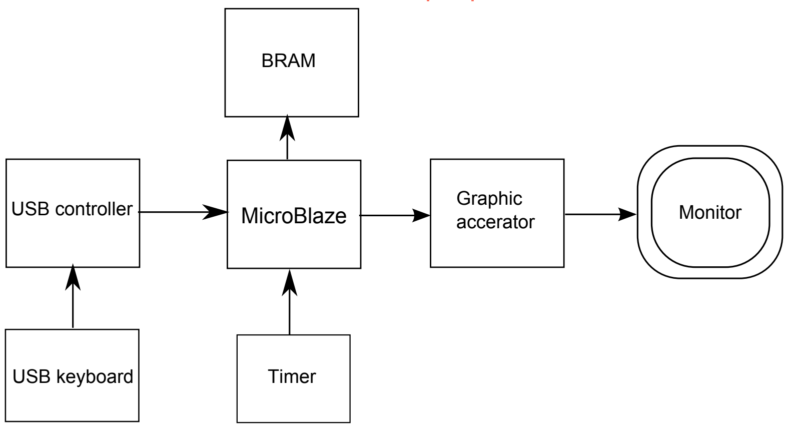


Figure 1: The overall hardware architecture of the design

3 Hardware

The overall architecture of this design is shown in figure 1. The design is based on the single processor PLB system generated by *Xilinx platform studio* 14.2 where the timer and graphic accelerator are designed and implemented by ourselves. The software codes are stored in BRAM and run by the *MicroBlaze*. All the peripherals are connected to *MicroBlaze* via *PLB* bus. Section 3.1 describes the detail implementation of the graphic accelerator. The USB keyboard and timer peripheral are presented in section 8. In addition, the *PLB* interface between graphic accelerator and *MicroBlaze* is given. The overall hardware testing is carried out by *Xilinx ISE project navigator*.

3.1 Graphic accelerator

The main task of the graphic accelerator is to decide which color to be displayed on the screen based on the Background RAM and Object RAM. Hence the *MicroBlaze* can update the Background RAM and Object RAM via *PLB* bus to control what to be displayed on the monitor. The graphic accelerator mainly consists of three parts, which are Background Renderer, Foreground Renderer and VGA controller as shown in figure 2.

Base on the requirement[1], the VGA controller generates horizontal sync signal hs, vertical sync signal vs and blanking signal *blank* for the VGA monitor using 25 MHz pixel clock. In addition, the VGA controller also gives two signals $Hcount$ and $Vcount$ to indicate the current pixel. The Background renderer outputs the RGB value of background at the current pixel where the Foreground renderer outputs the RGB value of foreground objects at the current pixel. Moreover, there is a multiplexer to choose which RGB value to output. If the RGB value of foreground object is 255, which means that the color is white(we use white as background for the foreground object picture), the multiplexer will select the RGB value of background to achieve transparent display between foreground object and background. The detail implementation of background

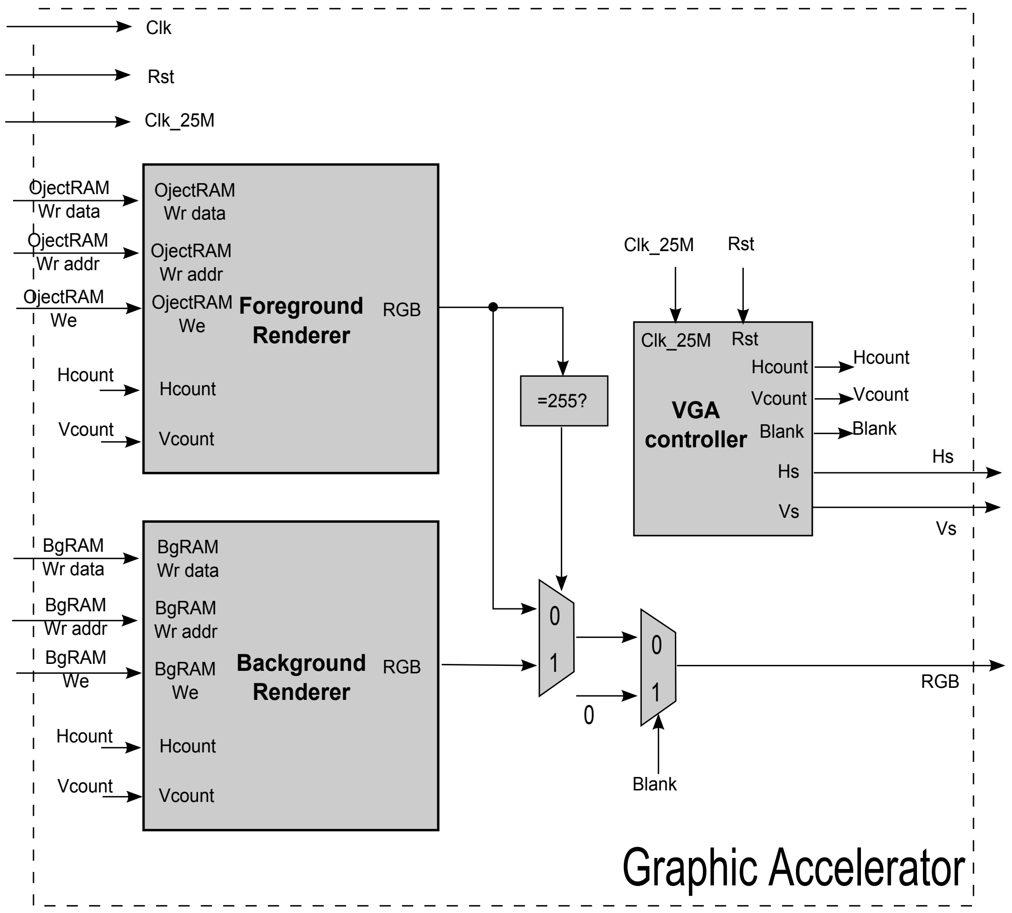


Figure 2: The block diagram of graphic accelerator

renderer and foreground renderer are described in the following section 3.1.1 and 3.1.2, respectively.

3.1.1 Background renderer

The overall block diagram of background renderer is shown in figure 3. 640x480 pixels can be divided by 20x15 tiles and each tile contains 32x32 pixels. The background RAM only stores the bitmap indexes of each tile, where the bitmap ROM only stores the color indexes of each pixel. By applying this addressing feature, the total hardware cost can be reduced dramatically. Figure 4 shows the words stored in different memories. There are 3 types of background tile in this game resulting in only 2 bits in each word of background RAM. Since the background RAM contains 300(20x15) words, the read address for the current pixel can be calculated by

$$bgram\_rd\_addr = Hcount[10::5] + Vcount[10::5] * 20 \tag{1}$$

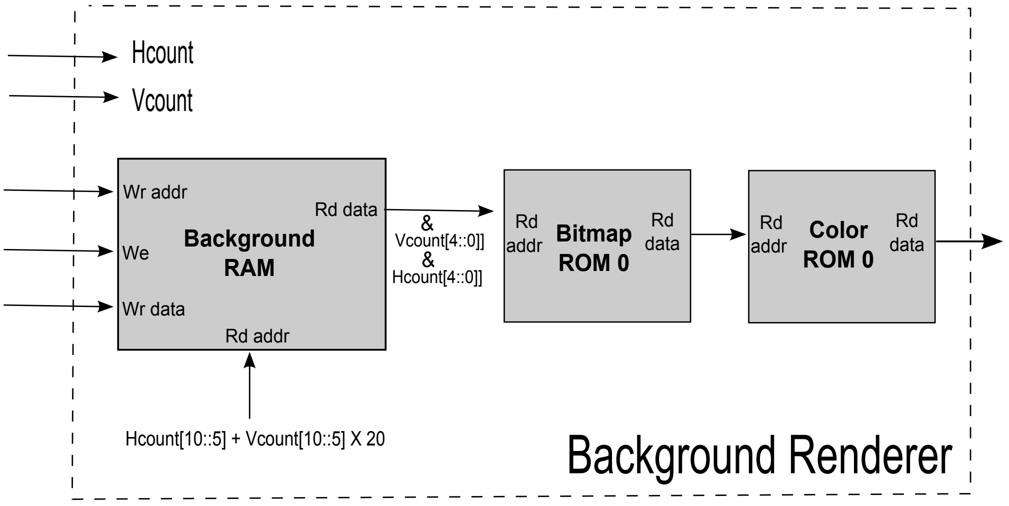


Figure 3: The block diagram of background renderer

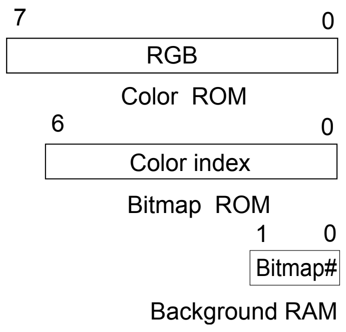


Figure 4: Words in the different memory of background renderer

where the *Hcount* and *Vcount* are given by the VGA controller as describes in the previous section. The read address of bitmap ROM for each pixel is generated by

$$bitmap\_rd\_addr = bgram\_rd\_data * 1024 + (Vcount\%32) * 32 + Hcount\%32 \tag{2}$$

which can be done in a very smart way without any calculation as show in figure 3.

3.1.2 Foreground renderer

Figure 5 shows the block diagram of foreground renderer. The idea of the foreground is to store the color index for a row(640 pixels) using a row buffer. In order to speed up and catch the timing, there are two identical row buffers, one used for odd row and the other used for even row making sure that one is under read if the other is under write. The row buffer only stores the color indexes of each pixel in each word while the object RAM stores the bitmap indexes, and its x-coordinate and y-coordinate of the left upper

Good informative schematics, and text description. The VGA controller overall seems to be complex and well thought out.

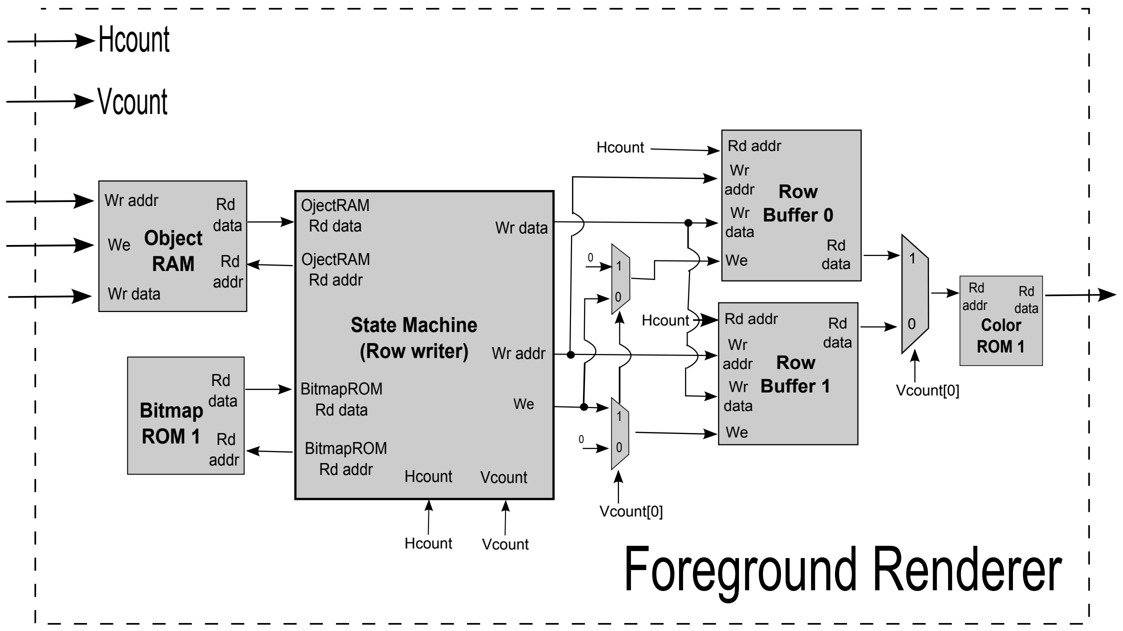


Figure 5: The block diagram of foreground renderer

conner in each word as shown in figure 6. The object RAM contains 128 words which means that it supports up to 128 objects(32x32 pixels each) showing in the screen.

There is a state machine to decide what to store in the row buffer according to the information of object RAM. Figure 7 shows the ASMD chart of this state machine. The state machine is waiting on the state *idle* and will jump to state *row_reset* when the signal *vcount* is in the visible area. The state *row_reset* is used for clearing all the values in the row buffer to make sure that the pixel displayed in this row is not mixed up with the previous values. After *row_reset*, the state machine will move to state $s1$ to decide whether the current object in the object RAM needs to be displayed on this row. If the current object should not be shown in the current row, then the next object coordinates are read and repeats the steps in state $s1$ until all the 128 objects have been read. If the current object should be presented in this row, the state machine will jump to state $s2$ to write 32 values in the row buffer. state $s1$ and $s3$ are just used for compensating the delay when reading value from memories. The address calculation of the bitmap ROM is similar to the one of background renderer as shown in the *default value* part of the ASMD chart. Does that mean that you have a 128 (or more) latency until you can decide which object to display? How do you solve the issue of displaying the object at the right location?

3.2 USB keyboard and timer peripheral

The keyboard will generate an interrupt signal when it has been pressed. The port of keyboard is USB but the protocol is PS2 actually. The timer is implemented by two registers: the first one is called delay register which store the delay period and the other called control register which for starting, stopping and checking if the timer has

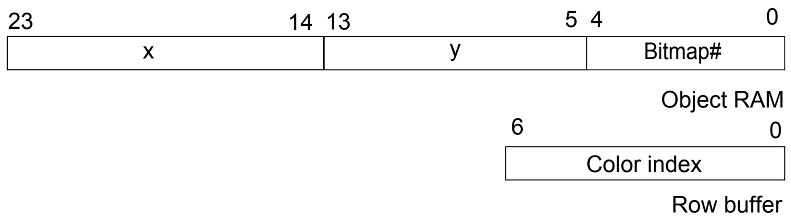


Figure 6: Words in the different memory of foreground renderer(words in bitmap ROM and color ROM are the same as the ones in background renderer)

expired. The delay register can be set by the software application to determine how long of a delay it requires. This value will remain in the delay register unchanged until the software application need to change it for some reason it needs to. The control register will only use the first two bits, being bit 0 and bit 1. Bit 0 is read-only and used by the timer peripheral to inform the software application that the timer has expired. Bit 1 is readable and writeable, and it will be used by the software application to make the timer *run*. When a "1" is written to this bit, the timer will start counting down from the delay value. When a "0" is written to this bit, the timer will reset. The timer peripheral architecture is shown in Figure 8.

3.3 PLB interface

The communication between *MicroBlaze* and graphic accelerator is based on PLB [2]. The graphic accelerator connects to a PLB slave and the PLB slave can receive the data from PLB and providing the address decoding. It should not connect to a PLB master because the graphic accelerator only accepts the write data for the background RAM and the object RAM and it does not send any data to the *MicroBlaze*. Figure 9 shows the connection and address mapping between PLB and graphic accelerator. Notice that the bit 13 to 31 are not used in the register 0 of the PLB slave and it can provide a further extension if the graphic accelerator needs more bits to communicate.

4 Software

The software is written in C, which is mainly responsible for the game logic such as the player drawing, the enemy drawing, the collision handling, updating the corresponding information in the object RAM and background RAM. Figure 10 shows the flow chart of the game logic. The interrupt routine of the keyboard and timer are described in section 4.1. Section 4.3 and section 4.4 presents how the player tank, enemy tanks, bullets are controlled and updated. Section 4.2 gives how the topographic of background is drew. Finally, collision detection and handle is given in section 4.5.

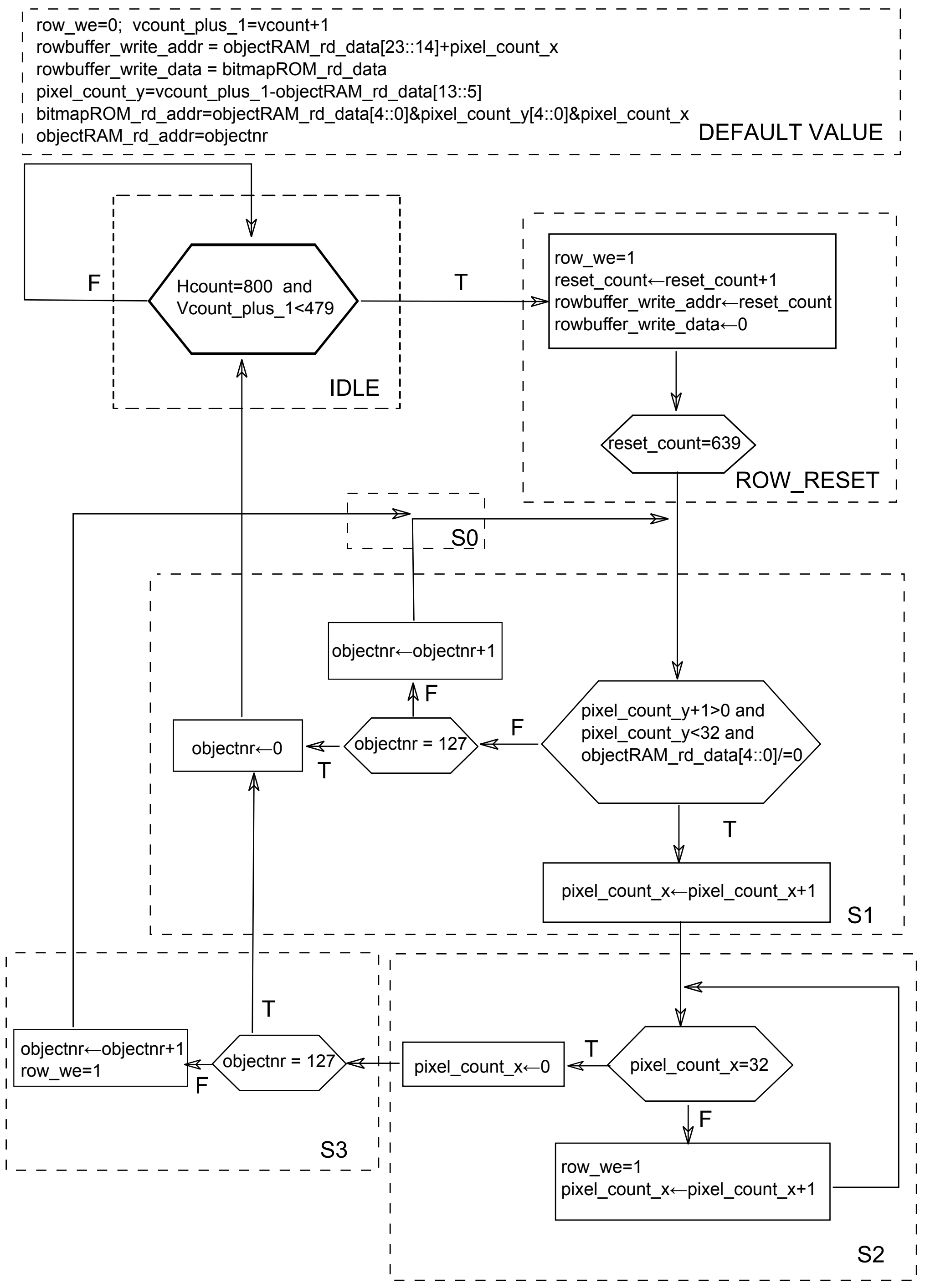

row_we=0; vcount_plus_1=vcount+1
rowbuffer_write_addr = objectRAM_rd_data[23::14]+pixel_count_x
rowbuffer_write_data = bitmapROM_rd_data
pixel_count_y=vcount_plus_1-objectRAM_rd_data[13::5]
bitmapROM_rd_addr=objectRAM_rd_data[4::0]&pixel_count_y[4::0]&pixel_count_x
objectRAM_rd_addr=objectnr
DEFAULT VALUE
F
Hcount=800 and Vcount_plus_1<479
T
IDLE
row_we=1
reset_count←reset_count+1
rowbuffer_write_addr←reset_count
rowbuffer_write_data←0
reset_count=639
ROW_RESET
S0
objectnr←objectnr+1
F
objectnr←0
T
objectnr = 127
F
pixel_count_y+1>0 and pixel_count_y<32 and objectRAM_rd_data[4::0]/=0
T
pixel_count_x←pixel_count_x+1
S1
T
objectnr←objectnr+1
row_we=1
F
objectnr = 127
S3
T
pixel_count_x←0
pixel_count_x=32
F
row_we=1
pixel_count_x←pixel_count_x+1
S2

Figure 7: The ASMD chart of the state machine in foreground renderer

So you implemented your own timer? Why? (Note: OPB is not supported anymore. Maybe you mean PLB)

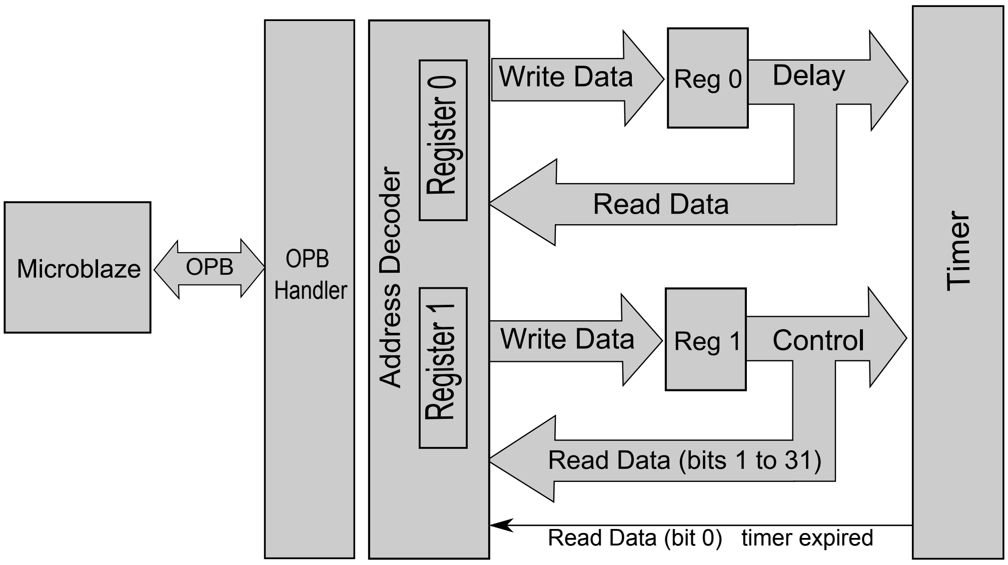


Figure 8: the timer peripheral

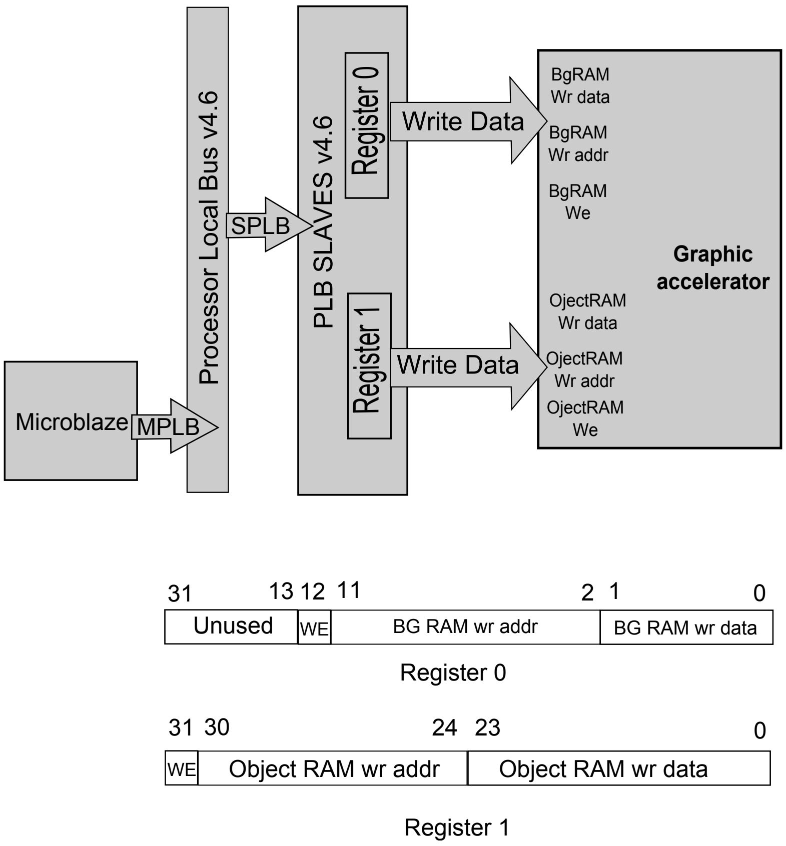


Figure 9: PLB connection and the address mapping with graphic accelerator

Again, informative diagrams - well done.

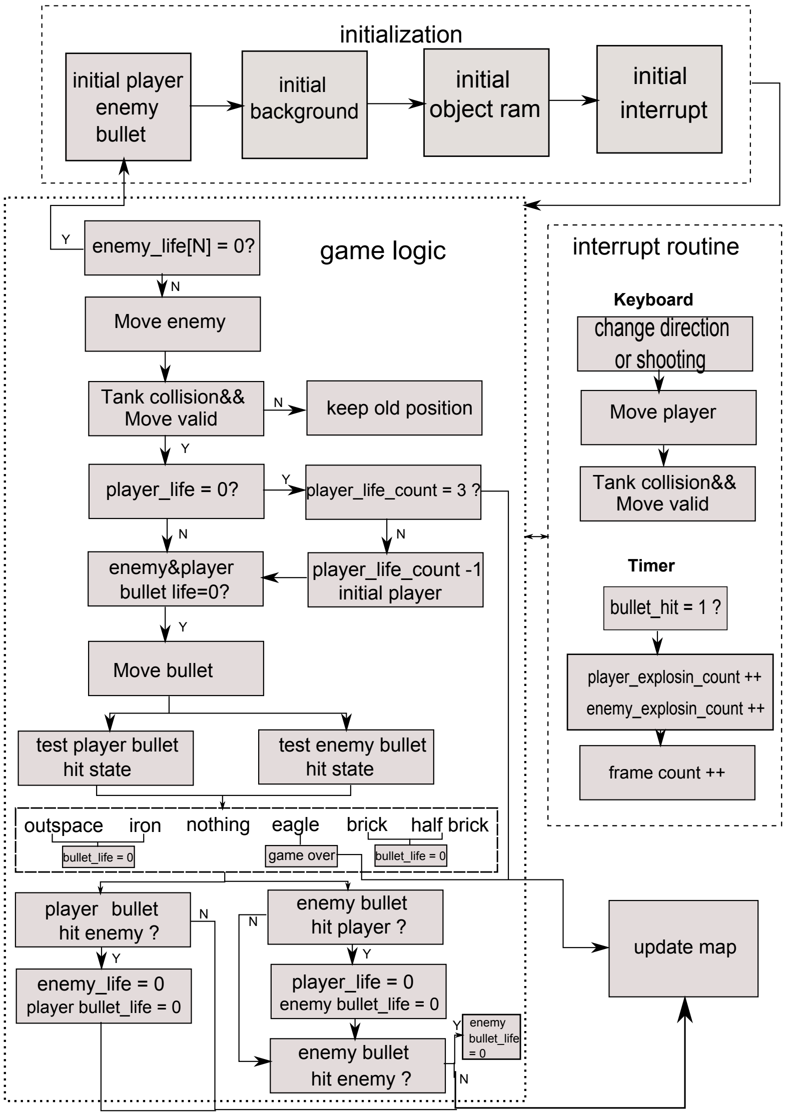


Figure 10: The flow chart of the game logic

4.1 Interrupt routine

This part will describe how to initialize interrupt in software before the main loop start running. The interrupt handler functions are also described.

The software will process the interrupt through an interrupt handler function which gets called whenever timer expires or keyboard has been pressed.

First of all, the ps2 driver should be initialize as shown in listing 1.

```
ConfigPtr = XPs2_LookupConfig(Ps2DeviceId);
XPs2_CfgInitialize(Ps2InstPtr, ConfigPtr, ConfigPtr>BaseAddress);
```

Listing 1: Ps2 driver initialization

Then initialize the interrupt controller and register the interrupt handler function of keyboard and timer as shown in listing 2.

```
XIntc_Initialize(IntcInstPtr, INTC_DEVICE_ID);
XIntc_Connect(IntcInstPtr,IntrId,(XInterruptHandler)XPs2_IntrHandler,Ps2Ptr);
```

Listing 2: Interrupt register

Next start the interrupt controller. Specify real mode so that the PS/2 device can cause interrupts through the interrupt controller. The data received interrupts and the global interrupt in the PS/2 device are enabled as shown in listing 3. The timer interrupt

```
XIntc_Start(IntcInstPtr, XIN_REAL_MODE);
XIntc_Enable(IntcInstPtr, IntrId);
XPs2_IntrEnable(&Ps2Inst, XPS2_IPIXR_RX_FULL);
XPs2_IntrGlobalEnable(&Ps2Inst);
```

Listing 3: Interrupt enable and start

setup steps are differ from keyboard since the timer is custom-defined. First, register the interrupt handler as shown in listing 4.

Following that, interrupt handler of interrupt controller itself must be enabled. Timer and macroblaze interrupt will be start after that as shown in listing 5.

4.2 Background settings

The background consists of two layers. Both layers are updated row by row. The furthest layer shows the ground. The top layer shows the objects such as the bricks, irons, enemy tanks, etc.

The VGA is 640*480 pixels and the tile map is 32*32 pixels. Hence a static u8 map [15][20] is defined to express each tile map in background. The static u8 map [15][20] consist of three types: 0 for the ground, 1 for the bricks, 2 for the irons and 3 for the eagle. The game has two levels. After initialize the background during the

```
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
XPAR_XPS_INTC_0_MY_TIMER_0_IP2INTC_IRPT_INTR,
MY_TIMER_Intr_Handler,(void *)XPAR_MY_TIMER_0_BASEADDR);
```

Listing 4: Timer interrupt register

```
XIntc_MasterEnable(XPAR_XPS_INTC_0_BASEADDR);

XIntc_EnableIntr(XPAR_XPS_INTC_0_BASEADDR, (XPAR_MY_TIMER_0_IP2INTC_IRPT_MASK |
XPAR_PS2_MOUSE_KEYBOARD_IP2INTC_IRPT_1_MASK));

MY_TIMER_mWriteReg(baseaddr, MY_TIMER_INTR_IPIER_OFFSET,
 0x00000001);

MY_TIMER_mWriteReg(baseaddr, MY_TIMER_INTR_DIER_OFFSET,
 INTR_TERR_MASK | INTR_DPTO_MASK | INTR_IPIR_MASK);

MY_TIMER_mWriteReg(baseaddr, MY_TIMER_INTR_DGIER_OFFSET,
INTR_GIE_MASK);

microblaze_enable_interrupts();
```

Listing 5: Interrupt enable

game logic, the value in map[15][20] will be sent to the background RAM for the graphic accelerator to display on the VGA monitor. So if any value in map[15][20] is changed, it will immediately update in the background RAM.

4.3 Player control and update

In every frame, the interrupt will detect the keyboard input to control the player tanks movement, shooting or stand by, then update the its parameters. The player tank is defined by a structure including the coordinate, direction, life, etc. For each direction of the player, there is a corresponding picture, same as the enemy tanks. Once the movement key of the player tank is pressed, the updated player tank's coordinate needs to pass the "valid movement test" function before the updated value to be written into the object RAM. Figure 11 shows how this function works. When the tank moves towards different direction, the corresponding corner's coordinates will be tested with the value of *map*[15][20] to check whether there is an obstacle. If yes, the player tank's coordinate will not update during this movement.

After updating the player tank's parameters during the game logic, the corresponding changes will send to the object RAM hence the graphic accelerator can update the player tank on the VGA monitor. The maximum bullets number of the player tank can exist at one time are three. The player tank has 3 lives. Once the player tank is hit by the enemies' bullets, it will be initialize again, which means that it will lost 1 life and be back to the left bottom position in the screen. If the player tank lost 3 lives, the game

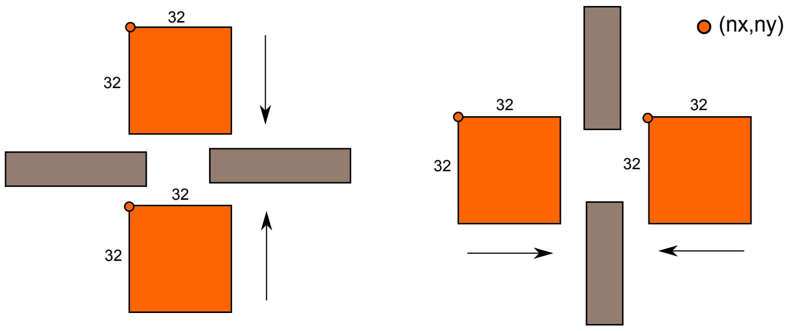


Figure 11: Valid movement testing scheme

logic will be initialize and restart.

4.4 Enemy control and update

There are four enemy tanks, which are born at the top of the map. Here a counter is used for counting by a timer in hardware, to update the objects information in every 33 ms so that the enemy tanks will move one step in each frame. The step size of the enemy tank is the same of player tank, which is 16 pixels. Once the collision with topographic is detected during enemy movement by using the same function as player tank, the "ENEMY AI" function is called, which will keep its old coordinate value and randomly generate an integer between 1 to 4 for a new direction choosing. All the enemies have been set to shoot in every ten seconds and they only have 1 life which means that they will disappeared when hit by the player tanks bullet. But if they shoot each other, the bullets will just disappear and do not affect any thing.

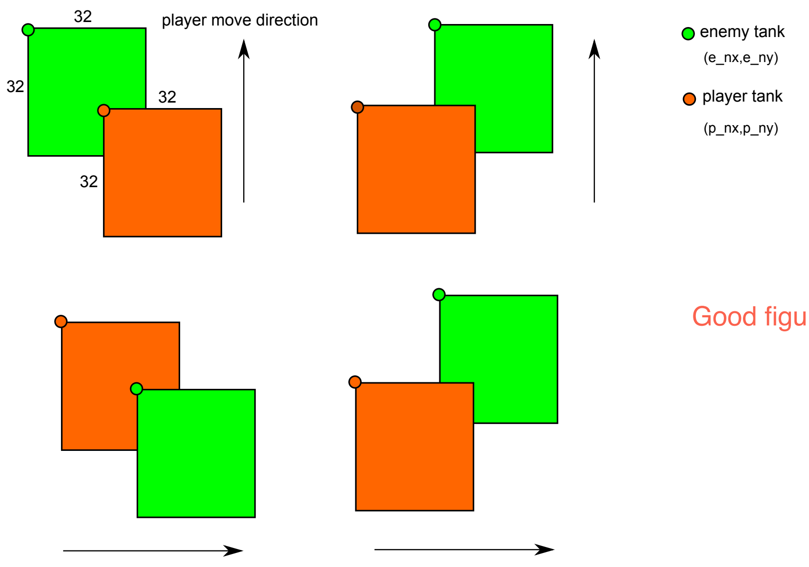
4.5 Collision

In this game, the collision has five types:

- The collision between player tank and enemy tanks
- The collision between enemy tanks and enemy tanks
- The collision between bullets and player tank
- The collision between bullets and enemy tanks
- The collision between bullets and topographic

Figure 12 shows how the collision is detected for moving towards different direction of player tank. For both player tank, enemy tanks and bullets, only the left top conner of the coordinates(p_nx, p_ny for player tank, e_nx, e_ny for enemy tank as shown in figure) are stored in their structures. Hence, if the condition

$$p\_nx <= (e\_nx{+}32)\ and\ (p\_nx{+}32) >= e\_nx\ and\ (p\_ny{+}32) >= e\_ny\ and\ p\_ny <= (e\_ny{+}32) \tag{3}$$



Good figure!

Figure 12: Collision detection scheme

is fulfilled, the player tank(p_nx, p_ny) has collision with enemy tank(e_nx, e_ny). The same detection is used for the other collisions. If the bullets detect the collision, there is an explosion animation displaying which is handle by displaying 3 pictures continuously with the help of timer. If a bullet hits a brick, the corresponding brick will disappear half and the brick will totally disappear if it gets one more hit.

5 Implementation Problems

Not all the formats are supported by xil_printf.

5.1 Software debug

The command "xil_print" was used as the main way for debugging, but it is not always working in this project. This command sometimes will make the program crash and we didn't realize that at the beginning. So we must be careful when using this command and have a sense that this command sometimes will create problem.

The speed of the bullet and the tank moves extremely fast at the beginning because it moves one step in every frame. To solve this we added a counter. When the counter counts to 4000 frames, the tanks and the bullets will move one step.

5.2 Interrupt system

As described before, the interrupt system include two types of interrupts: timer and keyboard. The keyboard use Usb port but follow PS2 protocol actually which is explained very clear in the document [1]. The document about how to setup XPS timer as interrupt is not very clear and the example is very less. So the timer interrupt in this project is done by two simple register. This idea is strongly inspired by [3]. The interrupt initial function is also changed a bit in new version of EDK software 14.1. The function names for the drivers have changed to remove all instances of $\_m$ from the driver name, but the parameters for the drivers have all remained the same. For example, the function $XIntc_mMasterEnable$ has been renamed to $XIntc_MasterEnable$ and all of the parameters are identical between the two functions.

5.3 Picture drawing

The pictures are drew using 8 bits each for red, green and blue color(in total 24 bits). Since the FPGA VGA port is only support 8 bits color(3 bits each for red and green, 2 bits for blue), the picture needs to be transform in 8 bits color. In order to get the best transformation for keeping the original style of the picture, MatLab is used for solving this problem. First the pictures are converted into ”true color image” format, which the color value is during 0 to 1 each for red, green and blue. Then we let this red and green color value multiply by 3 and blue color value multiply by 2. After that all these three values are round to an integer. These steps can be easily done in MatLab and finally we get the color transformation from 24 bits to 8 bits done. Then the pictures are converted into ”index image” format for separate the color index and color value and written in bitmap ROM and color ROM respectively.

Ingenious solution, and nice idea.

5.4 Hardware limitation

Hardware limitation is a big issue for most of the projects in this course since most of the projects are games and games are memory intensive project. Of course it is also a serious problem for our project. In the beginning, the 16 KB BRAM is used for $MicroBlaze$ and all the memories used in graphic accelerator are block memories. As the software size increasing when adding more game logic, the software codes cannot fit in the 16 KB BRAM and we have to use 32 KB BRAM. When using the 32 KB BRAM for $MicroBlaze$, the design was too large to map in FPGA board. Then we tried several ways to decrease the hardware size and each trying of course needs to re-synthesis the hardware which waste a lot of time. Finally, we decided to use distribute memories instead of block memories for the graphic accelerator for saving block rams since the distribute memories are using logic slices to build the memory logic. We suggest that the $Xilinx$ tool should estimate the hardware resource faster instead of finding the space problem during exactly mapping after half an hour.

6 Conclusions and Discussions

The design works as expected and has been demonstrated in the lecture. Due to the limitation of time, this is only the basic version of the game and we think that the game can be more and more fun if adding more features. The detail suggestions for the further possible improvements are described in section 6.3, where the device occupancy and memory usage of the software codes are presented in section 6.1 and 6.2 respectively.

Resources	Used	Utilization(%)
Occupied Slices	2,155 out of 2,278	94
Slice LUTs	6,035 out of 9,112	66
Bonded IOBs	136 out of 232	58
RAM16BWERs	32 out of 32	100
BUFG/BUFGMUXs	4 out of 16	25
ILOGIC2/ISERDES2s	30 out of 248	12
BSCANs	1 out of 4	25

Table 1: Device utilization

.Text(Bytes)	.Data(Bytes)	.BSS(Bytes)	DEC(Bytes)	HEX(Bytes)
25594	1396	3094	30084	7584

Table 2: Memory usage of software codes

6.1 Device occupancy

The synthesis report shows that the maximum clock frequency is 143.529 MHz. The device utilization of the FPGA board is shown in table 1.

6.2 Memory usage

The memory usage of the software codes is shown in table 2. The ".text" represents code command where the ".data" represents the initialized static and global variables and the ".bss" represents uninitialized or zero initialized static and global variables. The software optimization level is set to "$-Os(Optimize\ for\ size)$". Consequently, the BRAM using in the *MicroBlaze* should be 32 KB to be able to run this game.

6.3 Possible improvements

Software: The software part is mainly about the game logic. The first suggestion of course is to have more topographic of background such as forest, ice, buildings and so on as well as adding the corresponding effect for these new topographic. In addition, we can add more types of enemy tanks and give different ability for them such as different size, different lives, different bullets, different AI for increasing the difficulty of the game. It is also awesome to create some bosses for different level and increase the level number. It is better to develop a way to automatically generate background map and enemy for saving memory resource. Moreover, a game menu and animation of winning and losing are needed for a better user experience.

Hardware: The graphic accelerator can be improved to have more layers for supporting more transparent display. When using the distribute memory instead of block memory, it is possible to make a better state machine since the distribute memory does not have output delay for reading while block memory has at least one clock cycle delay for reading.

7 Lessons Learned

Shuai Cheng: I have experienced and learnt a lot through this project. First of all, I realized that it is extremely important to choose a project topic that I am really interested in. This interest motivated me to work diligently throughout the project despite all the challenges encountered. Moreover, this interest encouraged me to think out of box. We felt heartily the sense of accomplishment when we finished the project.

Following that, communication among group members is essential. Regular meeting is a good way to keep track on our progress and make sure the we follow the project time line properly. Our regular discussions have fostered the team spirit and inspired more ideas.

Last but not least, I have learnt that we need to find a good way to debug the designed system. In our case, we used Chipscope to debug the PLB communication between microblaze and hardware. It is useful but time consuming since we have to assign the output port for the signal to be observed. Every time we did this, we have to do the synthesis hardware all over again.

Longyang Lin: From this project, I have learned many things. First of all, planning. Planning is one of the most important things for a project as well as following the schedule. In our case is that we do the things slowly at the beginning and realize that the project is not small after several weeks, which makes that we work for the more than 18 hours per day in the last week. That is a true story if you ask the clean staffs of our university and they met us at the early morning since we stayed at the lab for whole night. If we do a better planning in the beginning, I think that we do not have to work such hard at the end and it is a very good lesson for me to understand that.

It is very fun to develop a game. You will find a lot of bugs when you are testing and most of these bugs will let you laugh. In our project, most of the time have been spent on hardware and software debugging. There are always some small details that make you crazy and have to spend several hours to solve. Hence I also improved my skills of debugging in VHDL and C. All in all, this is a wonderful project that creating a fun game and improving my practical ability.

Yuhang Sun: From this project, it makes following the schedule is the most important. We spent much time at the last week. We just finish the project before the deadline. If we have more time, we can add more advanced application into the project, which will makes better. It is an efficient way to separate the works at the beginning, which improves my C programming skills. Through this project, I can use the C language and the Xilinx expertly. Debugging is important, and it will takes much time.

8 Contributions

Shuai Cheng:

- Project report: writing or contributing to sections 3.2, 4, 5.2, 7, 8
- Keyboard and timer peripheral: research, development, debugging and testing
- Software and hardware interface : research, development, debugging and testing
- Game logic: research and testing

Longyang Lin:

- Project report: writing or contributing to sections 1, 2, 3.1, 3.3, 4, 5.1, 5.3, 5.4, 6, 7, 8 and appendix A
- Graphic accelerator: research development, debugging and testing
- Game graphics : research, drawing and transforming all the pictures
- Game logic: debugging and testing

Yuhang Sun:

- Project report: writing or contributing to sections 4, 5.1, 7, 8
- Game logic: research, development, debugging and testing

References

[1] Nexys3 Board Reference Manual,
http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf

[2] LogiCORE IP Processor Local Bus(PLB) v4.6 (v1.05a),
http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf

[3] timer and interrupt,
http://www.fpgadeveloper.com/2008/10/timer-with-interrupts.html

Appendix: User Manual

Entire system setup:
1. Connect a USB keyboard and a VGA monitor to Digilent Nexys3 FPGA board.
2. Connect the FPGA board to a computer using USB cable to download bit file.

Use ”w,a,s,d” of the keyboard to move the player tank up, left, down or right. Press ”enter” to fire a bullet and the player only can fire maximum 3 bullets at the same time. The player have 3 lives to complete the game. If the player run out of life it will automatically jump to the level 1 for restarting the game.

A very good report, interesting to read, with many insights. Overall a very good project. Well done!