02320 HW/SW programming

3 weeks course

Space Invaders

25th of January 2019

Christina Juulmann
s170735

1 Introduction

The objective of this project is to implement a Space Invaders-like game on the Zybo board. The game is displayed on a monitor through the VGA port attacted to the FPGA. With the use of Vivado and Xilinx SDK, the project will design a digital system that handles VGA output as well as user input from button presses on the onboard buttons, thus functioning as game control.

1.1 Requirement specification

Minimum requirements are established for the project. These are categorized into *game mechanics* and *game control* and describes what the project should implement as a minimum:

Game mechanics

- Option of playing as single player
- Have a high score and point system
- Game should be able to end with a "game over" screen
- Player should have x amount of life which the game over state depend on
- Player and enemies should be able to attack eachother
- User should be able to play in various levels of difficulties

Game control

- Display game on monitor using VGA output
- Control player movement using onboard buttons (moving left/right)
- Player should be able to fire a shot using one button from the board

Moreover, are the following extentions established if time allows it:

- Option of 2 players
- Advanced point system enabling bonus points
- Video Agumented Reality controller instead of buttons using image processing techniquies

2 Analysis

In order to acheive the requirments a breif analysis is conducted to investigate what internal requirements can further be estabilshed.

2.1 Internal requirements

The game must be implemented on a Zybo board, thus making use of the Vivado tool and Xillinx SDK is found fit; The hardware modelling language will be of VHDL, though this project will make use of IP cores from the Vivado tool, thus working with a GUI approach that requires no VHDL development. Moreover, software will be written in the C language using the Xilinx SDK.

Player control will make use of buttons attached to the FPGA and the frame display will make use of the VGA port. Since both components are real time, an interrupt setup is found fit for these components of the project.

Frames that are to be displayed also needs buffering, thus a framebuffer mechanism will be needed.

In order to offer different difficulties for the gameplay there will be a need for tweeking different parameters of the game, for instance some random number generation and probability calculations based on simple statistical knowledge, in order to generate random enemy attacks with probabilty theory.

2.2 Project risk assesment

At the beginning of the project it was discussed whether or not focus should be on implementing the video input and thus the AR controller, in oppose to implementing buttons for player control.

It was found too risky, as the consequences were assesed to be that of the project might ending with no showcase at the demondstration, thus the project aims to implement buttons for player control as a first iteration, and if time alows it, a second iteration can implement the video input.

3 Design

The digital system comprising the project will be build from a Zynq-based design. An all programmable system on chip (APSoC) design implemented by the Zynq-device is a smart way to design a digital system, since this method allows design reuse through interllectual property functional blocks (IP cores) available from the Xilinx catalouge. [1] (p. 2). This project builds from the *hdmi-in* tutorial (using VGA output) provided from Digilent [2] which includes multiple of such IP cores. In the next sections these IPs are further ellaborated on with respect for their use, as well as an overview of what should be added to the project besides the material found in the tutorial.

3.1 System design

A breif overview of the main building blocks for the project is listed below.

- Zynq processing system
- Processor system resets
- VGA output
- Button interrupt
- AXI interconnect
- Double buffering

3.1.1 Zynq processing system

The Zynq PS holds 256K RAM and 32KB Level 1 cache [1] (p.18) from which the framebuffer can be stored and accesed. However, it is costly to access memory, and reading while writing to the framebuffer at the same time will result in a flickering display/animation. To accomodate this challenge the project will implement a double buffering mechanism.[3] The idea is to have two framebuffers; a front-buffer and a back-buffer. While the front-buffer can be read, the back-buffer is where we write to, and when the display needs to update the whole frame allocated in the back-buffer will be swapped to the front. In this way we only read from memory once i.e. when the buffer is swapped, and flickering is avoided.

3.1.2 VGA output

The frames that are to be displayed willl be written to the Zynq-device memory, as mentioned above, thus the VGA output will need a method to fetch these data from memory. Moreover, does data need to be converted to VGA video source, which can be achieved by the AXI4-Stream to Video Out IP core. Reading data from memory can be achieved by the use of AXI Video Direct Memory Access (VDMA) IP.[4]

The VDMA block provides an asynchrouns clock mode which will generate five clock domains; an AXI4-Lite domain, memory map to stream (mm2s) and stream to memory map (s2mm) on the streaming side, as well as mm2s and s2mm on the memory map side. The two clocking domains on the memory map side is required for the Zynq processor memory access. The other two clock domains on the streaming side are required in order to use the AX4-Stream to Video Out block, since this block sends data in streaming mode. Furthermore, the VDMA configured in async clock mode requires the AXI4-Lite clock to have lower frequency than both mm2s and s2mm clock domains on the memory map side, [4] (p. 47) thus the system will also implement two Processor System Reset modules which can be configured with the desired lower clock frequency.

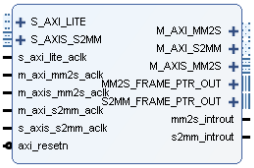


Figure 1: AXI VDMA block

Returning to the VGA output, lets first breifly define a video signal; a signal that is divided into two phases, one phase where pixels are drawn, called the front porch and back porch, and one where they are not, called the sync interval. The sync interval happens inbetween the two drawing phases (front and back porch) and are defined as a horizontal and vertical sync. The former defines the horizontal line at a give refresh and the latter a vertical line (the refresh frequency for drawing a new screen).[5](p.19) [6] At last the pixelclock defines the available time to display a pixel.

The AXI4-Stream to Video Out module converts from the AXI4 stream to video output supplying the desired syncing signals. [7] Moreover, the pixel clock can be provided by a dynamic clock generator which will reference the system clock.

3.1.3 Button interrupt

Buttons on the FPGA will be used to generate interrupts, which will be handled in software. Moreover, buttons are configured in programmable logic (PL) by the use of GPIO IP core, thus providing an AXI interface.

The interrupt setup requires a connection for the CPU to receive and handle interrupts from various interrupt sources. The ARM CPU comes with one Interrupt Request line (IRQ) which will be used for this. However, since the video system will use this connection as well, a Generic Interrupt Controller (GIC) is to be setup to handle multiple interrupt request for the CPU. Furthermore, an interrupt handler for the buttons will be needed in order to configure and initialize the GIC. Below is an illustration of the setup.

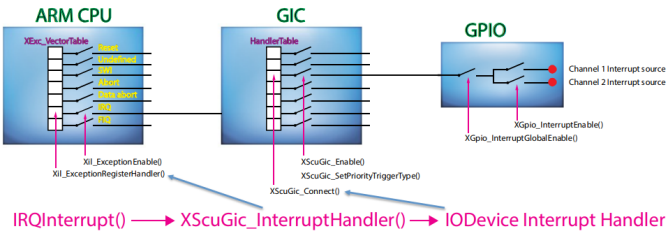


Figure 2: Interrupt setup of GPIO in PL to PS
[8]

3.2 Game logic

The next sections will describe the structure of the game and how it should be implemented, as well as, defining the gameplay.

3.2.1 Gameplay

This version of Space Invaders will be implemented as a single player shooter game, where the player controls a laser cannon that moves from left to right at the bottom of the screen. The objective of the game is to shoot down aliens, spawning in clusters of 3 rows with 10 aliens in each row. Aliens will decend from the top in a special patteren moving from the screen edges and down, i.e. moving right, down then left, down, etc. Moreover, aliens will attack back by firing shots at the player as well. The game is over if the player dies from the alien attacks or if the aliens reaches the player at the bottom of the screen. The Player wins if he shoots down all the aliens from which he will set a highscore.

3.2.2 Game states

The game should be structed into three game states; a start game-, in-game- and game-over state. The first and latter states depends on user input. User should press a button in order to jump to the *in-game* state as well as when in *game over* state, it is up to the user to decide if he wants to replay the game, thus jumping to *start game* state. However, the transition from *start game* to *game over* should happen if the player dies or wins the game. Below is a simple state diagram illustrating this idea.

3.2.3 Game objects and mechanics

In order to make up a game-play the game needs as a minimum a hero and some enemies. Both objects/ characters should hold multiple attributes such as x and y coordinates defining their positions on the display, health describing if they are alive, and some ammunition empowering them to initiate attacks.

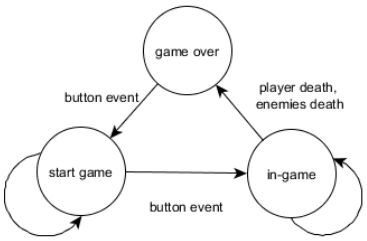


Figure 3: Game states

All game obejcts will be defined independently and as structures having at least x and y coordineates as memebers.

Moreover, each object will have a function associated for every ability it holds. These abilities might include the following,

Hero	**Enemies**
attack	attack
move left/right	move left/right/down
die	die

Table 1: Game object abilities

The enemiy attacks should be based on different parameters. In this way the parameters can be set relative to the current level that the user is playing. Some of these parameters that should be implemented is; frequency of enemy attacks, probability of an enemy attack and how fast enemies will decend. The probabilty parameter can be designed from statistical probability theory taken from the Bernouli distribution.

Let E be a random variable following the Bernouli distribution, then its discrete probability distribution takes the value 1 with probability p and 0 with the probabilty $q = 1 - p$

Thus the probability distribution of a single trail has the sample space of success/failure. In other words, every enemy when they have the chance to attack will with a given probability attack.

4 Implementation

In the following sections a breif overview is made of how the game was actually implemented. In the appendix 8.1 a block diagram of the final digital system is illustrated.

4.1 Frames

A double buffer is implemented having two buffers declared as two arrays; a front-buffer called $frameBufA$ and a back-buffer called $frameBufB$. Frames are written to the back-buffer and right after copied to the front-buffer using the C library function $memcpy()$. Moreover, frames are written to the buffers using pointers, thus keeping down the memory consumption.

Drawing a frame can be illustrated using a coordinate system as attempted illustrated in the figure below, where origin (i.e. $(x_0, y_0) = (0, 0)$) is in the top left corner. Drawing happens in a double for-loop where we for every x-position go through all y-positions. However instead of starting at position 0.0 every time, we start at the coordinates for each objects to be drawn. In this way a vertical line is drawn for every x-position within the game objects boundaries. Moreover, there is a factor 3 on every object to be drawn on a frame because of the RGB color system implemented in the VGA cable and monitors. Thus one pixel takes up three spaces in the framebuffer arrays.

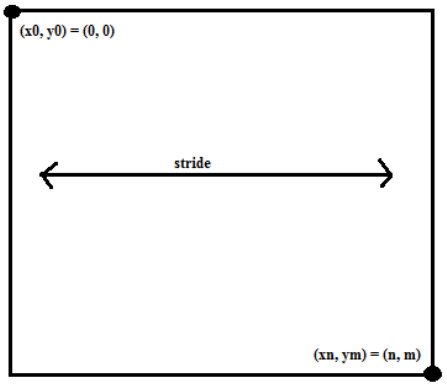


Figure 4: Drawing a frame

4.2 Button interrupt

The system is configured to utilize interrupts in Vivado. The GPIO IP core is configured, by checkmarking the "enable interrupts", and the Zynq PS block by checkmarking the "Fabric Interrupt". This will add an interrupt port on both blocks which are to be connected. The latter will add the IRQ line on the Zynq PS thus enabling interrupts requests from PL to PS. The remaing setup is done in software. As mentioned in the design section, the IRQ line needs to be initialized and configured, which is done by the $Xil\_ExceptionEnable()$ function, which makes the connection to the GIC and is called inside the initial setup function $InterruptSystemSetup()$. This function also makes the connection from the interrupt source to the interrupt handler by a call to $XGpio\_InterruptEnable()$ function. Most important, however, is the instance of the build in interrupt controller $XScuGIC\ INTCInst$, which makes the connection from the GIC handlertable to the interrupt source and thereby handles all the interrupt requests and their associated functions.

4.3 Game logic

The game logic is structued in in two main parts; a 'define object' logic and a 'drawing logic'. All objects thus has a define and drawing function associated with it, e.g. $DefineHero()$ and $DrawHero()$. Furthermore, game objects are declared as structs and the definition functions will thus set the memebers of the struct which they associate. Below are illustrations of the implemented game objects.

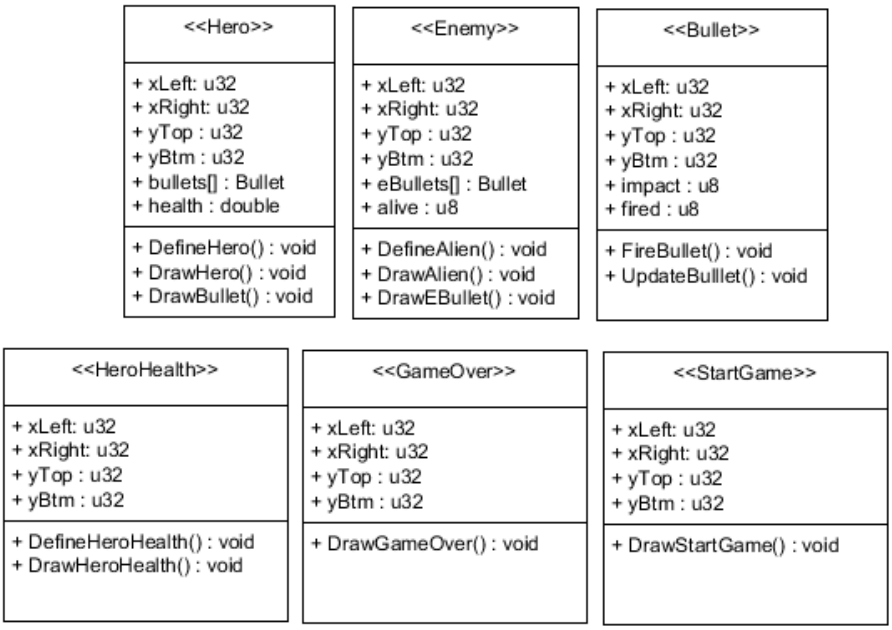


Figure 5: C structs for game objects

4.3.1 Game states

The game states are implemented as suggested in the design section. Using an *enum* type to define the three states *startgame*, *ingame*, and *gameover*. The whole program is encapsualted in an infinite loop, besides global and initial variables that are set before entering the loop. It is in here that the three game state cycle through each states. The game-state is initially in *startgame* and thus the start-game scope. Here we ask if a button is pressed, and if so the game-state is set to *ingame* and thereby breaking the first scope and enters the next. We do not break out of the *ingame* scope until player dies or wins. If this is the case the game-state is set to *gameover*, and we wait for a button to be pressed in order to start over and re-enter *startgame* state. See below for a flow diagram illustrating this (flow diagram has excluded other game functionality not concerning game states).

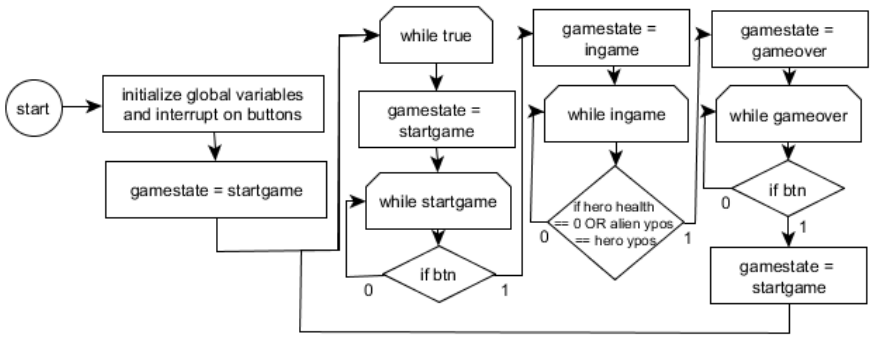


Figure 6: Flow diagram of game states

4.3.2 Game levels

Three game levels were implemented having the following parameters relative to the current level:

- Player life
- Alien movement speed
- Freqency of alien attack
- Probabilty of alien attack

In level 1 the player gets 10 lives. Alien movement and attack freqency is controlled by a counter constituting an 'event control' for aliens, which will coun to 32. Moreover, there is 20% probability of an alien will fire a shot.

In level 2 the player gets 5 lives. The event control counter increments to 16, thus more frequent alien events will occur and there is a 40% probability of an alien will attack.

I level 3 the player gets only 2 lives. The event control counter increments to 12 and there is a 60% probability of an alien will attack.

5 Test

During implementation various parts of the game were tested. A breif test summary is shown in the table below.

Test summary

Test case	Expected results	Actual results	Remarks
Double buffering	Smooth display update	Somewhat smooth update	Expand double buffer to multiple frames
Game levels	Increased difficulty in gameplay	Somewhat increase in difficulty	Constrains on player ammunition
Game over	Player dies when out of life	As expected	
Button control	Instant move on button press	As expected	Display update challenge

As briefly summerized in the table above where the four main parts of the project tested during development.

Double buffering: As double buffering were implemented it showed a big improvement from the first iteration where the game did only support one framebuffer. However when movement were increased, thus calling for more frequent updates of the display, the same patteren of lack starts to show. It is not much though, but can be detected.

Game levels: The random number generation and probability implemenation of wheter an alien will attack strongly increased the gameplay difficulty, however as the player still holds 100 lasers in the laser cannon, it is still somewhat easy to shoot down all enemies. The biggest impact are that the player only has two lives and aliens attack with a probability of 60%, which makes it hard to avoid alien bullets as the decend.

Game over: The life mechanism works as expected where life shown in the bottom left corner is updated accordingly to player hits. Moreover, does the game go to game over state when player is out of life.

Button control: When buttons are pressed the player should move instantely. It turned out as expected. However, in relation to the framebuffers, it cam be seen that the player will start to flicker a small bit when button is pressed down, and push the limits of the screen update.

6 Discussion

Most functionality was successfully implemented, however, certain areas of the project could be improved. From the test the following areas of interest was realized:

- Expantion of framebuffers to hold more than one frame at a time could pose a good solution to accomodate flickering displays.
- Gameplay level could be refined in order to be more challenging. Introducing constrains on player ammunition capacity as level increases suggest a good solution. It was found that, despite the aliens growing more aggressive, the player was still able to fire away in a laserbeam-like defense thus not challenged as much as expected.

Furthermore, the highscore and point system was not implemented and tested, despite there exists a good base to get that last requirement implemented. Because of the well defined game objects and drawing functionality, one solution to the implementation could be to follow the same patteren as the rest of the game. For example adding another memeber to the Hero struct called "kills" which would increment for every alien he killed. This member could then be referenced when highscore needs to be written.

An idea for implementing the highscore display, would require a drawing method like the others, however the frames that needs to be written would be ten different images of the numbers zero to nine. These could be kept in an array and referenced relative to the kill member of the Hero structure. When highscore reaches ten, eleven etc. one could mask the "kill" member in order to get the tens and ones of the number, by dividing by ten and applying the modolus operator respectively.

A final remark, in the original Space Invaders game the player is given "bunkers" from which he can hide behind and avoid alien attacks. Moreover, the aliens moves faster depended on how many aliens the player have actually shot down. These features were not implemented in this project, nor were they in the requirements list, but would be a good starting point for optimizing on the gameplay thus making it more challenging.

7 Conclusion

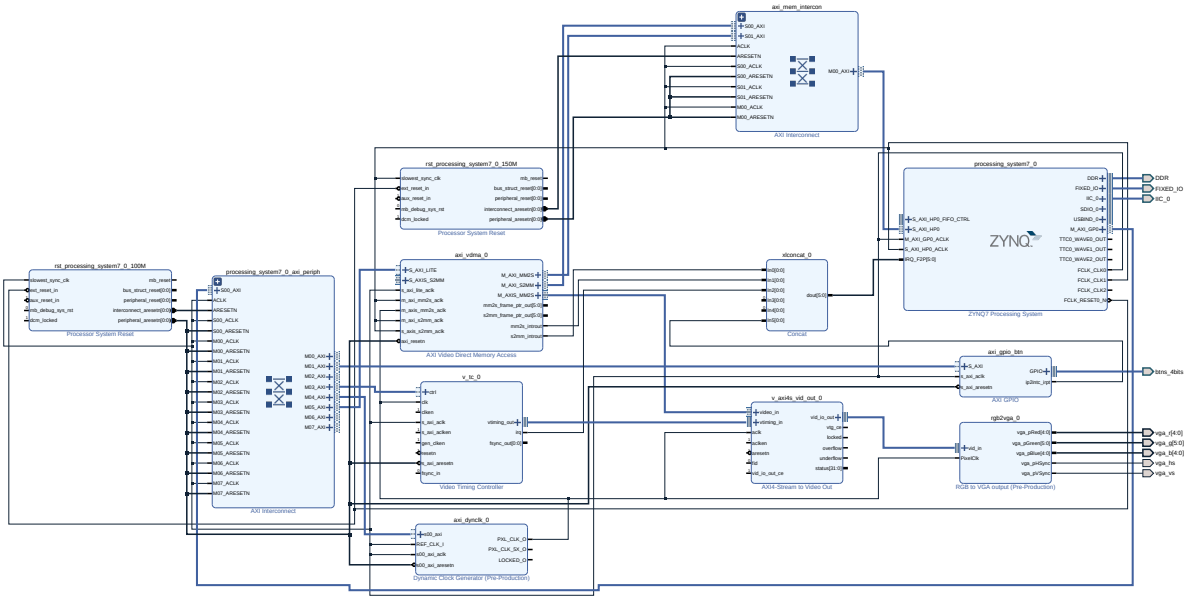
All of the requirements were implemented successfully except for one. The highscore and pointssystem did not manage to get implemented at all, however a solution for its implementation is proposed in the discussion.

The project managed to implement a game where the user can play as single player and control the player with buttons from the FPGA. The user can also play in three diffeent levels which is defined by the amount of life the player gets, frequency of enemy attacks defined by a counter control event, and probability of enemy attacks inspired from probability theory. Moreover, enemies are able to shoot at the player, the player can shoot back at enemies by pushing a button and win the game as well as loose the game and end up as game over.

Game objects are succesfully written to the screen using VGA and double buffering for optimized display updates when game objects move.

8 Appendix

8.1 Block diagram af digital system



8.2 Contributions

I am using the interrupts.h file from the previous group work which is made by Hamza Nasser Awad (s160490) as well as the if-else block where the variable skew is incremented/decremented. However, all other materiale is my own work. All $Draw$, $define$ logic etc. is work that I have done, with insiration from the tutorial (hdmi-in) which we were given by the supervisor. Images of aliens and player is done by using various online tools.

References

[1] Louise H. Crockett et al. *The Zynq book*. Strathclyde Academic Media, Glasgow, Scotland, UK, 2014.

[2] Digilent. Zybo hdmi input demo, . URL `www.reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-hdmi-input-demo/start`.

[3] OSDev. Double buffering. URL `www.wiki.osdev.org/Double_Buffering`.

[4] Xilinx. Axi video direct memory access v6.2. URL `https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf`.

[5] Digilent. Zybo reference manual, . URL `www.reference.digilentinc.com/_media/reference/programmable-logic/zybo/zybo_rm.pdf`.

[6] FPGA Cookbook. Video timings: Vga, svga, 720p, 1080p. URL `www.timetoexplore.net/blog/video-timings-vga-720p-1080p`.

[7] Xillinx. Axi4-stream to video out v4.0. URL `www.xilinx.com/support/documentation/ip_documentation/v_axi4s_vid_out/v4_0/pg044_v_axis_vid_out.pdf`.

[8] Edward A. Todirica. Lecture 9. URL `www.cn.inside.dtu.dk/cnnet/filesharing/download/07668f2e-14c1-4a0a-b704-1db681081ddb`.