ECE532 - Team 19: Final Project Report
Jet Fighters Video Game

Fahad Khan
Mufan Wang
Salman Fazal
Usman Yaqoob

April 13, 2017

Table of Contents

1 Overview

1.1 Project Motivation

The project is motivated by the idea of Xbox or Wii whose games allow interactions between users and the hardware. The design provides a representation of the game console that utilizes Nexys4 FPGA, which creates a unique experience for gaming where multiple players can play against each other using cellphones as the controller. The basic design idea is illustrated as Figure 1 below.

Fig 1. Design Idea

1.2 Project Goals

The goal of design is to realize the functionalities for all IP blocks as listed in Table 1.4.1 in section 1.4.1, to integrate these blocks and to create a smooth interaction between software and the hardware components.

1.3.1 Functional

- Interactive multi-player gaming interface .
- Be able to use cell phones as game controllers.
- Real time interaction between the software and the hardware
- Integration with Sound
- Multiple PMOD BT2 sensors integration with interrupts

1.3 Block Diagram

A block diagram of system is shown in Figure 2. The functionality of each individual block is described in section 1.41..

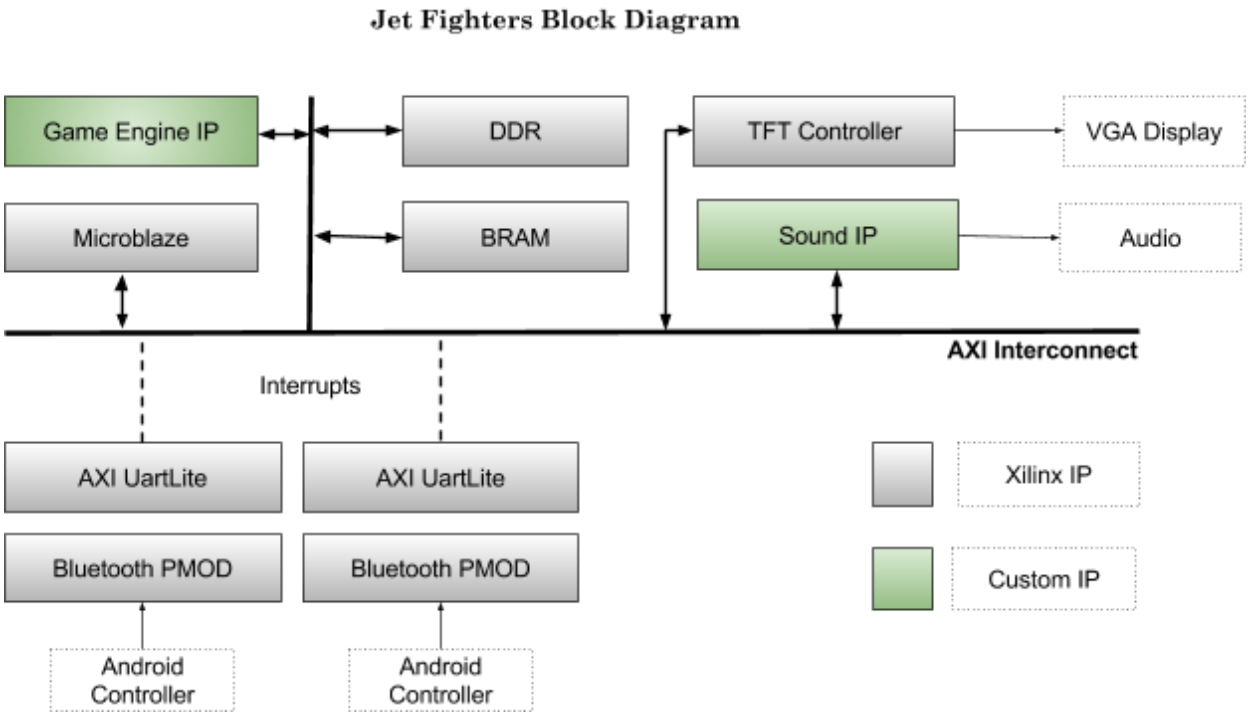


Fig 2. Block Diagram

1.4 IP Descriptions

This section describes each IP components contained in the design . The Figure below illustrates the overall design flow diagram in the project.

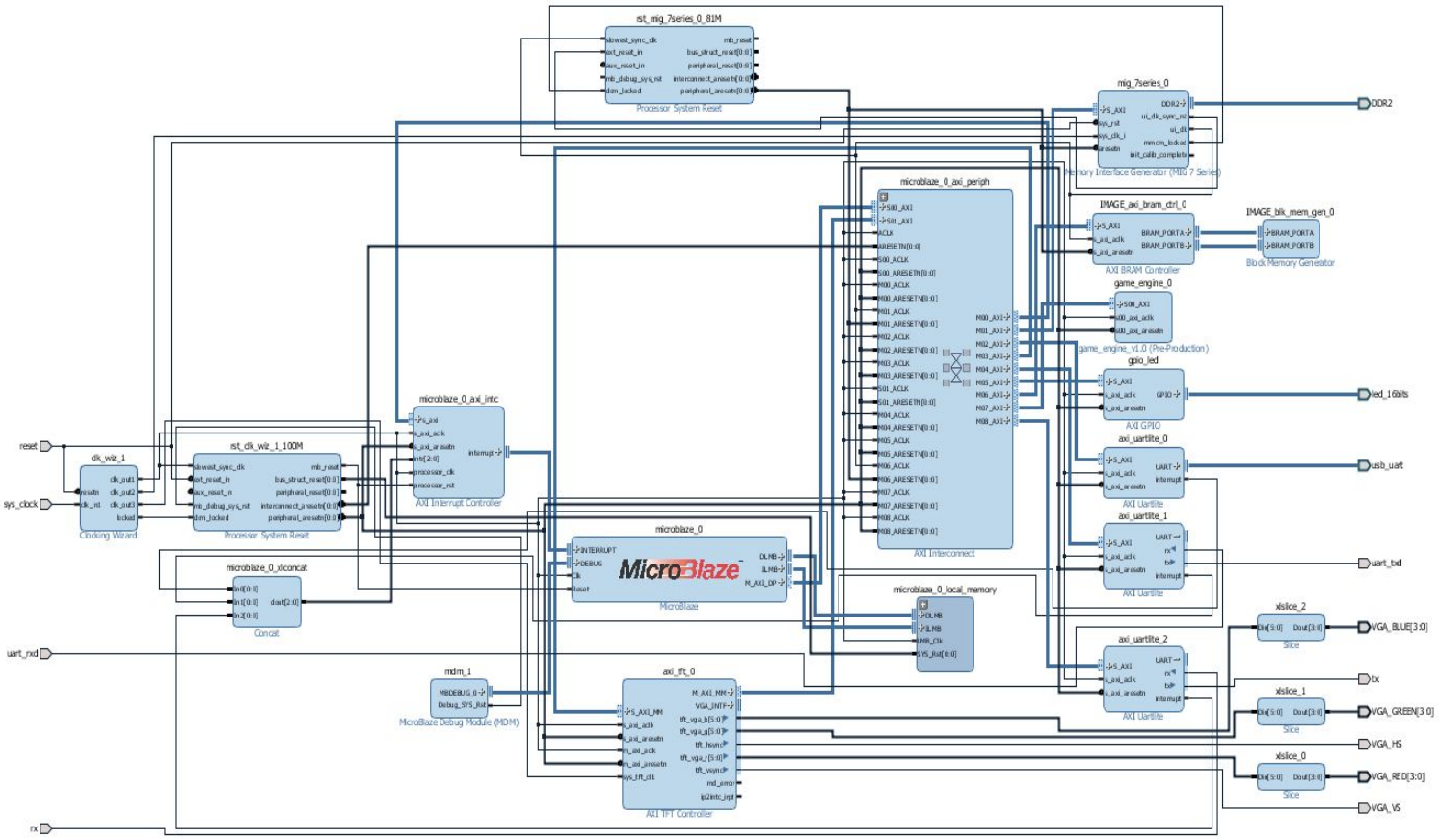


Fig 3. Flow Diagram

1.4.1 Description of major IP blocks

The Table 1 below contains brief Description of our IP blocks

Block	Version No.	Description
TFT[1]	2.0	This block is the interface for the VGA. It is taken from the vivado IP catalog.
Microblaze	9.6	Soft-core processor taken from vivado IP catalog.
Audio IP	1.0	Custom IP block that handles audio output
Game Engine IP	1.0	Custom IP block that handles all the game logic. This block was created by ourselves.
UART	2.0	This block allowed the communication of Microblaze and the PMOD bluetooth sensor and also allowed debugging. This was taken from the vivado IP catalog.
DDR	4.0	This block is the memory we used to load our image and sound files. It also provided us to hold the Image buffers for the VGA output. This was taken from the vivado IP catalog.
BRAM	4.0	This block provided us with soft memory. It held a few images before we moved our data to DDR. This was taken from the vivado IP catalog.

Table 1. Block Descriptions

[1] *Xilinx XPS TFT manual[online]. Accessed February 3rd, 2017 available, https://www.xilinx.com/products/intellectual-property/xps_tft.html*

2 Outcome

All the features were implemented as they were listed in the proposal. The implemented features include VGA, audio, game engine, Bluetooth PMOD, Microblaze and memory (DDR + BRAM). The design worked as it was intended. It provided a smooth and engaging gaming experience for the user.

While the final design worked as intended, a few things could have been improved had the time constraints permitted:

1) The frame drawing module could have been implemented in hardware to improve rendering speed instead of being implemented in software. This might have improved the Frame rate of the game because currently pixel drawing is implemented over the TFT interface in microblaze.
2) The audio IP, while it works properly should not have used its internally generated clock. However, various problems were encountered when this was attempted and an internally generated clock was used.
3) Another key area that can be improved is the Game Engine IP. The synthesis time for the final build exceeded 40 minutes. This made many iterations difficult to carry out, especially accounting for the subjective nature of gameplay and inaccuracies of simulation vs real-life. It could have been updated to make the gameplay more complex and utilize the benefits of hardware over software. Increasingly complex systems, while a lot easier to implement in software, get a significant performance boost when implemented in hardware.

The following could have been done differently:

1) Xilinx had a published pulse width modulation IP on one of its github repositories. A significant amount of time was spent incorporating PWM into the audio IP which could have been spent towards working on other aspects of the project had the team known it existed.
2) Xilinx has included a Bluetooth IP in its github IP repository as well. Using this IP is relatively more complex and was utilized in our original approach at the expense of time. In retrospect, the AXI UartLite IP is very easy to use and implement.

3 Project Schedule

This section compares our original Milestone plans with the actual weekly implementation.

3.1 Original Milestones

Table 2. Original Milestones

Milestone 1	Start implementing the bluetooth interface and coordinate interrupts from multiple devices. This one of the most time consuming portions of the project. Start working with the VGA module as well.
Milestone 2	Get the Bluetooth interface working by this point and interfacing with an external device. This will involve some UART data retrieval on the FPGA since the PMOD will interface with the sensor via UART
Testbench Demo	Testbench for the bluetooth module to be used as a metric for further testing.
Milestone 4	Finish with the VGA module and have some game engine functionality. The should be able to do something on the VGA output
Mid-Project Demo	Mid-Project Demo Have most of the components: bluetooth, VGA and game engine finished. Start integrating them. Start implementing audio(optional)
Milestone 6	Have some portions integrated, preferably all of them.
Final Demo	Final Demo Have everything integrated and working. FINAL DEMO

3.2 Accomplished Milestones

Table 3. Accomplished Milestones

Week1	Basic Bluetooth interface was working using the UART protocol. Used an open-source android application and configured it to work with this project. FPGA was able to send and receive data.
Week2	VGA interface was working and was able to draw basic objects and shapes on the screen and move them objects around. At the same time we were able to control the movement of those objects on the screen from the cell phone. The signals from the phone were configured to work with basic movement. Work started on the Audio IP.
Week3	Started working on game engine module. Implemented method to convert raw images into hex and store them in BRAM. Created multiple testbench frameworks that tested game engine functionality and BRAM data consistency. Attempts made to load data onto the Audio IP
Week4	Work in progress on the verilog game engine module. Successfully integrated the 2nd bluetooth sensor for multiplayer support . Wrote and verified game logic in C code to test the functionality before integration into verilog game engine. Able to load data onto the Audio IP and started testing.
Week5	Worked further on the game engine and added shooting in both missiles and also added collision detection. Able to move both planes with precision. Fixed bugs and got optimum frequencies for the Audio IP.
Week6	Successfully integrated full game engine verilog module into the project. Successfully implemented method to load image/sound files into DDR memory as BRAM was not enough. Generated new Audio data files
Week7	Successfully completed Audio IP and integrated it with the project along with the ability to play custom tracks. Added Music to intro of game, added collision sound effect, added special effects

3.3 Differences between proposed and accomplished milestones

1) Our initial assumptions were that Bluetooth protocol is very complex and that is why in our proposed milestones we allocated a lot more time for the bluetooth interface. Upon figuring out that the phone simply connects to the bluetooth sensor and that all we needed to do was to implement the UART protocol for the data transfer we got the basic bluetooth interface working, and got the phone to send and receive data to and from the FPGA.

2) Our testbench demo in the proposed milestone was about the Bluetooth sensor but since this was already done and verified by us in the 1st week we instead created our testbench framework for the game-engine module and for the BRAM (to verify image pixel color values).

3) In our proposed milestone the sound part of the project was kept optional in case we had time for it. Since the Bluetooth module did not take that much time we were able to also create and implement the Sound IP and successfully integrate it within our project.

4 Description of the System Components

This section describes the major functions of the design and the interactions between the hardware IP and the MicroBlaze processor.

4.1 Game Engine

Table 4. below shows the Game Engine IP registers and their addresses. Each of the register will be described in detail.

Table 4: Registers

Address Space offset	Register Name	Access Type	Default Value	Description
0x0000	slv_reg0	Read/Write	0x0	Control Register
0x0004	slv_reg1	Read	0x0	Status Registers
0x0008	slv_reg2	Read	0x0	Player1_Coordinate
0x000C	slv_reg3	Read	0x0	Player2_Coordinate
0x0010	slv_reg4	Read	0x0	Missle_1
0x0014	slv_reg5	Read	0x0	Missle_2

Control Register(slv_reg0)

C_S_AXI_DATA_WIDTH -1	18	17	16:13	12:9	8:5	4:1	0

Bit 0 is a data valid bit. This bit will be flagged to logic one when the bluetooth data processed in the MicroBlaze is ready to be processed by the IP. The correct setting of this bit will reject next readings if the current updating is not finished. Bytes 16:1 are used to communicate the speeds and direction received from the interrupting bluetooth controller. Bytes 17 and 18 are used as indicators of a "shooting event" in the game. The IP uses these inputs to process output that can be used reliably.

Status Register(slv_reg1)

C_S_AXI_DATA_WIDTH -1	0

The first bit of this register are used to ensure the correctness of data transfer between the Game Engine IP and the MicroBlaze. This status bit is flagged to logic one correspondingly when the valid reading bits (will be described later) is registered. This bit will be flagged to low when the current reading is finished. This protocol will ensure that there are no duplication of data transfer or missing of data collections.

Player1_Coordinate(slv_reg2)

C_S_AXI_DATA_WIDTH -1	21:11	10:0

These registers contain gameobject 1 Y and X axes coordinates, which will be used in the MicroBlaze to draw the updating graphs. Bits [10:0] will contain the X coordinates and bits [21:11] will contain the Y coordinate

Player2_Coordinate(slv_reg3)

C_S_AXI_DATA_WIDTH -1	21:11	10:0

These registers contain gameobject 1 Y and X axes coordinates, which will be used in the MicroBlaze to draw the updating graphs. Bits [10:0] will contain the X coordinates and bits [21:11] will contain the Y coordinate

Missile_1(slv_reg4)

C_S_AXI_DATA_WIDTH -1	23	22:12	11:1	0

Bit 23 is used to indicate the hit effect between the bullets and the gaming objects. Bit 0 is used to control the subsequent drawing the bullet, in which case no two bullets will be drawing on the same screen for one gaming object. Bits [11:1] and bits [22:12] are used to indicate the X and Y axis coordinates for missile.

Missile_2(slv_reg5)

C_S_AXI_DATA_WIDTH -1	23	22:12	11:1	0

Bit 23 is used to indicate the hit effect between the bullets and the gaming objects. Bit 0 is used to control the subsequent drawing the bullet, in which case no two

bullets will be drawing on the same screen for one gaming object. Bits [11:1] and bits [22:12] are used to indicate the X and Y axis coordinates for missile.

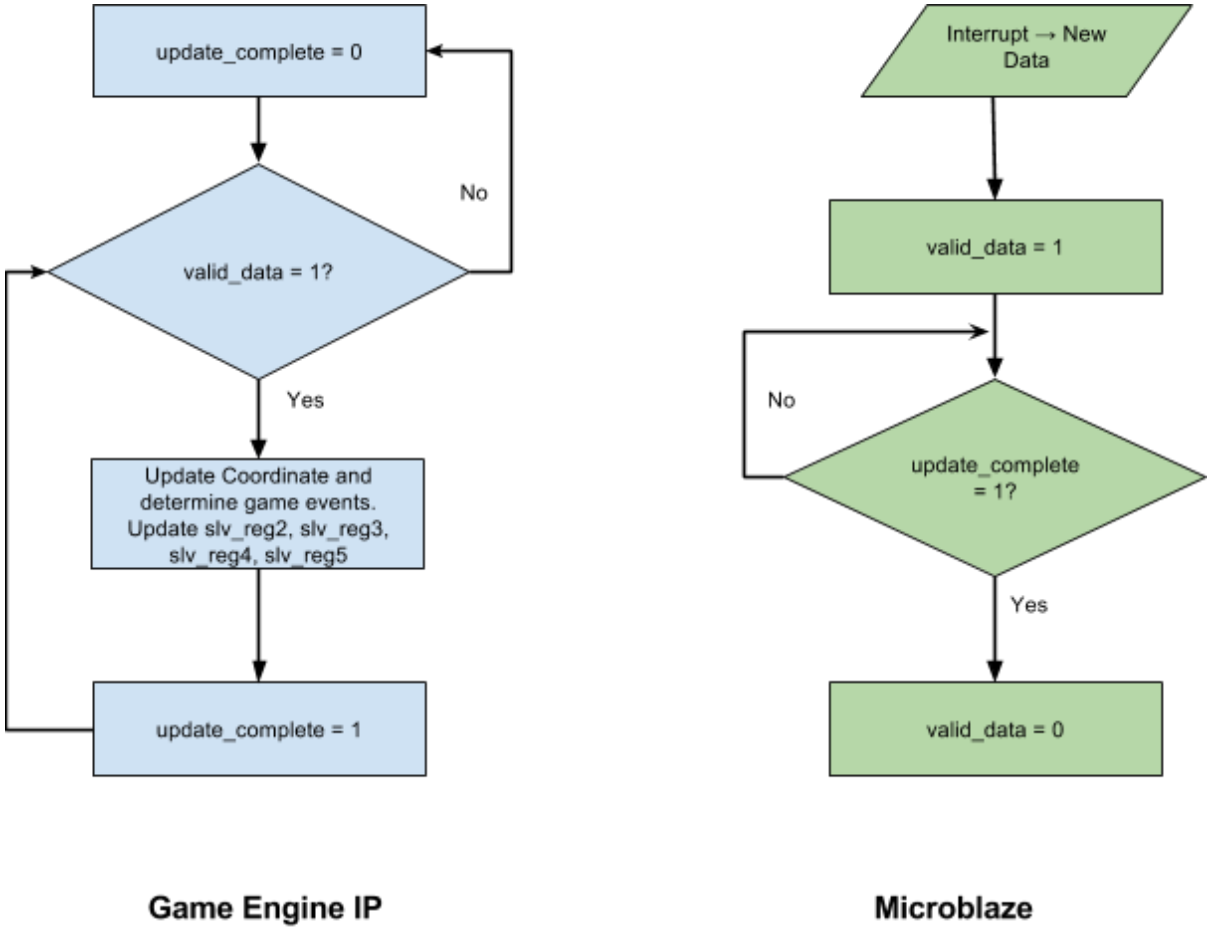


Fig 4. Communication between Microblaze and Game Engine IP

4.2 Bluetooth & Interrupts

To allow for multiplayer gameplay, two Bluetooth PMODs must be attached to the Nexys4 DDR board. The default mode for operation for these Bluetooth PMODs is using the UART Interface with 8 data bits, a single stop bit and no parity bit. To enable the Baud Rate Setting (9600), a jumper must be connected to PIO7.

Upon receival of data, a UART interrupt is triggered, which is then handled by Microblaze. As shown in Fig 4. Interrupts are initialized on both of the UART blocks receiving data from the Bluetooth PMODs. Both of them are tested to ensure that they do not fail. The function **SetupInterruptSystem(...)** is used to first

initialize the interrupt controller and then connect both of the interrupt devices to the interrupt controller. Two separate interrupt receive handlers **RecvHandler(...) & RecvHandler2(...)** are programmed to allow interrupts to received from two different bluetooth controller and be handled separately and uniquely. They are set as the recv handler of each UART device respectively. Finally interrupts are enabled on both of the devices and the code proceeds to the next part.

```
* Initialize the UartLite drivers so that it's ready to use.
Status = XUartLite_Initialize(&UartLite_1, DeviceId_1);
if (Status != XST_SUCCESS) {    return XST_FAILURE;}

Status = XUartLite_Initialize(&UartLite_2, DeviceId_2);
if (Status != XST_SUCCESS) {    return XST_FAILURE;}

* Perform self-tests to ensure that the hardware was built correctly.
Status = XUartLite_SelfTest(&UartLite_1);
if (Status != XST_SUCCESS) {    return XST_FAILURE;}
Status = XUartLite_SelfTest(&UartLite_2);

if (Status != XST_SUCCESS) {    return XST_FAILURE;}

* Connect the UartLite to the interrupt subsystem such that interrupts can
Status = SetupInterruptSystem(&UartLite_1, &UartLite_2);
if (Status != XST_SUCCESS) {    return XST_FAILURE;}

* Setup the handlers for the UartLite that will be called from the
XUartLite_SetSendHandler(&UartLite_1, SendHandler, &UartLite_1);
XUartLite_SetRecvHandler(&UartLite_1, RecvHandler, &UartLite_1);
XUartLite_SetRecvHandler(&UartLite_2, RecvHandler2, &UartLite_2);

* Enable the interrupt of the UartLite so that interrupts will occur.
XUartLite_EnableInterrupt(&UartLite_1);
XUartLite_EnableInterrupt(&UartLite_2);
```

Fig 5. Setting up and Enabling Interrupts

```
void RecvHandler(void *CallBackRef, unsigned int EventData)
{
    TotalReceivedCount = EventData;
    /* Address of UART Device */
    int gotint = *((int *)(0x40620000));
    /* Received Byte*/
    unsigned char got = gotint;
```

Fig 6. Interrupt Receive Handler

We separately configured an Android Bluetooth Controller and set up a basic communication protocol specific to this game.

Fig 7. Android Bluetooth Controller

Movements generate the following byte stream on UART: 'X' '*i*' 'Y' '*j*'
(*i,j* being the coordinates of acceleration ranging from 0-9)
Button Press: 'R' , 'G' , 'Y' , 'B'
Button Release: 'r' , 'g' , 'y, 'b'

Once received by the appropriate receive handler, they are communicated to the game engine by writing to the appropriate registers for further processing, and are used as needed.

4.3 Audio IP

Nexys DDR uses Mono Audio output jack to transfer audio signals to external audio sources. It is driven by a Sallen Butterworth low pass 4th-order filter to transfer those signals. It requires a pulse width modulated (PWM) or pulse density modulated (PDM) signal to operate. This project employs Pulse Width modulated signal to do this.

The IP itself performs pulse width modulation on the music input data. It first enables audio on the jack by writing a one bit 1 on the output wire, referred to as AUD_SD. It then reads the data in the source register and starts counting up to it. While its count is less than the value in the source register, it outputs a single bit on the audio output wire, referred to as AUD_PWM. It starts outputting a 0 on AUD_SD after the count exceeds the value in the source register.

The audio jacked and speakers work with audio downsampled to 8 KHz and the rest of the system operates at 100 MHz. The IP needs to have enough time to count up to the source register value before it is updated. To counteract this issue, the IP

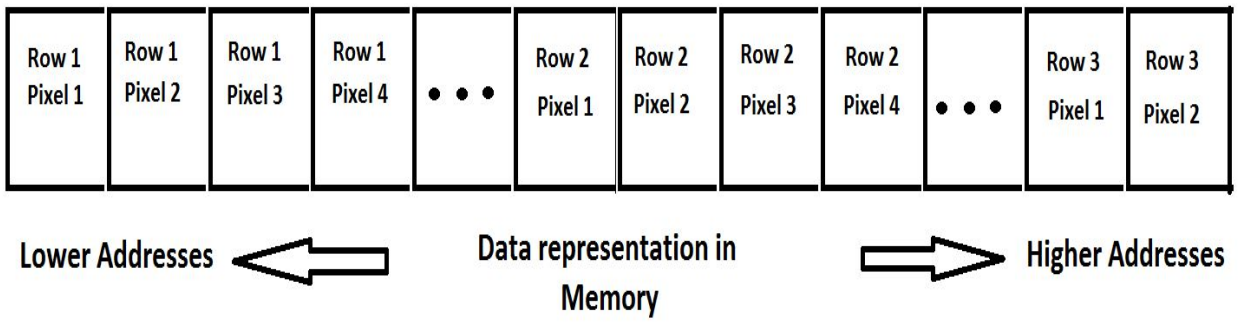
is passed the start and end address of the music data and it updates another register which contains the next address it needs to read the data from based on its own frequency. The microblaze system then reads that register fetches the data from the requested address and updates the source register for the IP. The IP updates the requested address at its own clock frequency and this solves the issue of updating data at the appropriate frequency.

The following points summarize how the IP works:

1. The audio is enabled by writing 1 on AUD_SD
2. The IP's clock starts counting at 8 KHz
3. Microblaze writes the data starting address and the starting data into the IP.
4. The IP starts counting up to the written value at the system frequency (100 MHZ)and outputs 1 on AUD_PWM until it counts up to it.
5. On every positive clock edge the the IP adds 4 to the current address and updates the request address.
6. Microblaze then fetches data from the requested address into the IP
7. Step 4, 5 and 6 keep repeating
8. If the IP reaches the end music data, it loops back to the start address of the data

4.4 DDR/BRAM Access

The memory is primarily used for the VGA double buffers and for storing of images. We converted images into hex files which are placed in row order i.e. we load the pixels row-wise into memory array and then when drawing on the screen we fetch it the same way from the memory.



4.5 Multiple Frame Buffering with DDR and TFT Controller

The graphics portion of the project, was implemented primarily by updating DDR memory segments which were pointed to by the TFT pointer. The TFT Controller then rendered an output of 18 bits of RGB (6 bits each) which was sliced to 12 bits RGB (4 bits each) and streamed to the VGA pins.

The 128Mb DDR memory was creatively segmented to allow multiple buffering for smooth animations without flickering, tearing and other artifacts. This created a seamless experience. Different memory segments were used for the loading screen, menu screen and the gameplay to allow for seamless transition.

5 Description of Design Tree

Our project is available at the link
Accessible:www.github.com/Uzmany/G19_JetFightersVideoGame

The main files of the project are shown in the folder hierarchy:

- ➔ **G19_JetFightersVideoGame**
  - → README.md
  - → **bin_files**
  - → **docs**
    - → video_link
    - → android_application_link
    - → presentation_slides
    - → group_report
  - → **srcs**
    - → .../vivado_project_files

6 Tips and Tricks

Moving flame effect:

To display a moving video effect such as moving flame the following needs to be done:
Store multiple images at different addresses in the memory and load them in alternate frames while updating the screen to create the moving effect. You can simply use an image application like MS-Paint to extend the exhaust for the 2nd image like we have done below.

Image1: narrow exhaust

Image2: Wide exhaust

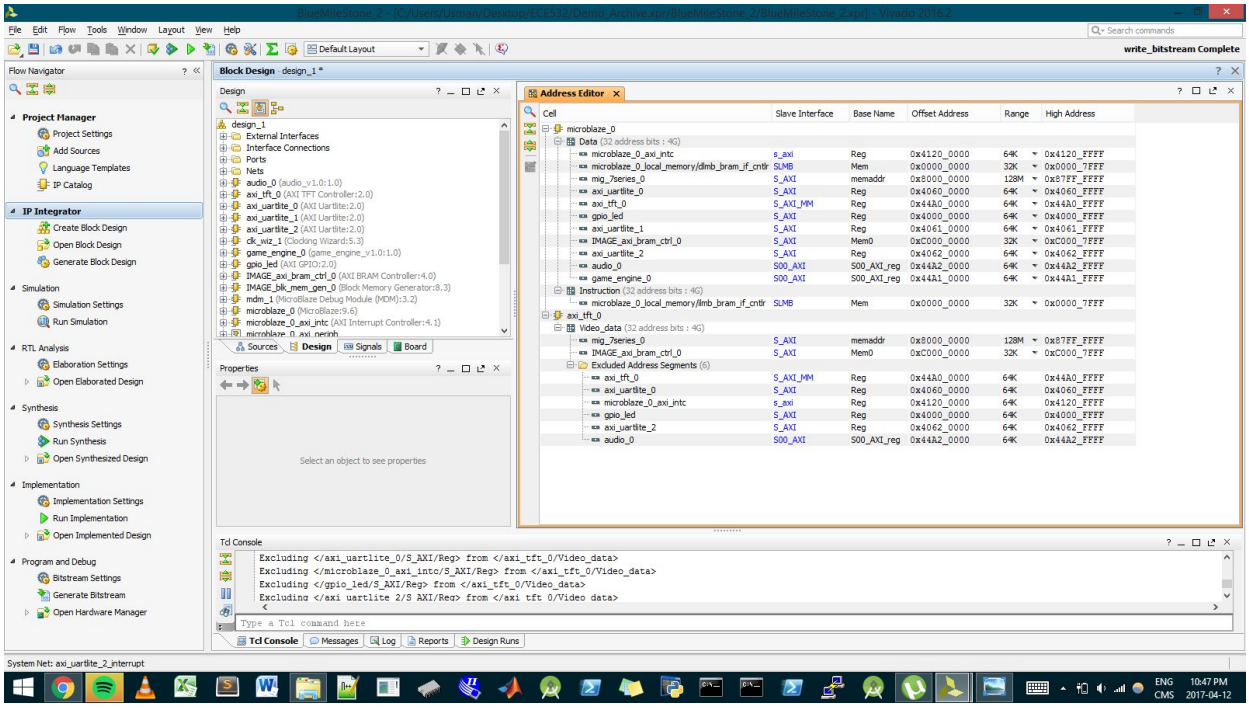
Loading the 2 images above in alternative frames gave us the moving jet flame effect.

7 Link: Video

The link below contains the video of our project demo.
Available: **https://www.youtube.com/watch?v=If0QKyL7WnY**

8. Appendices

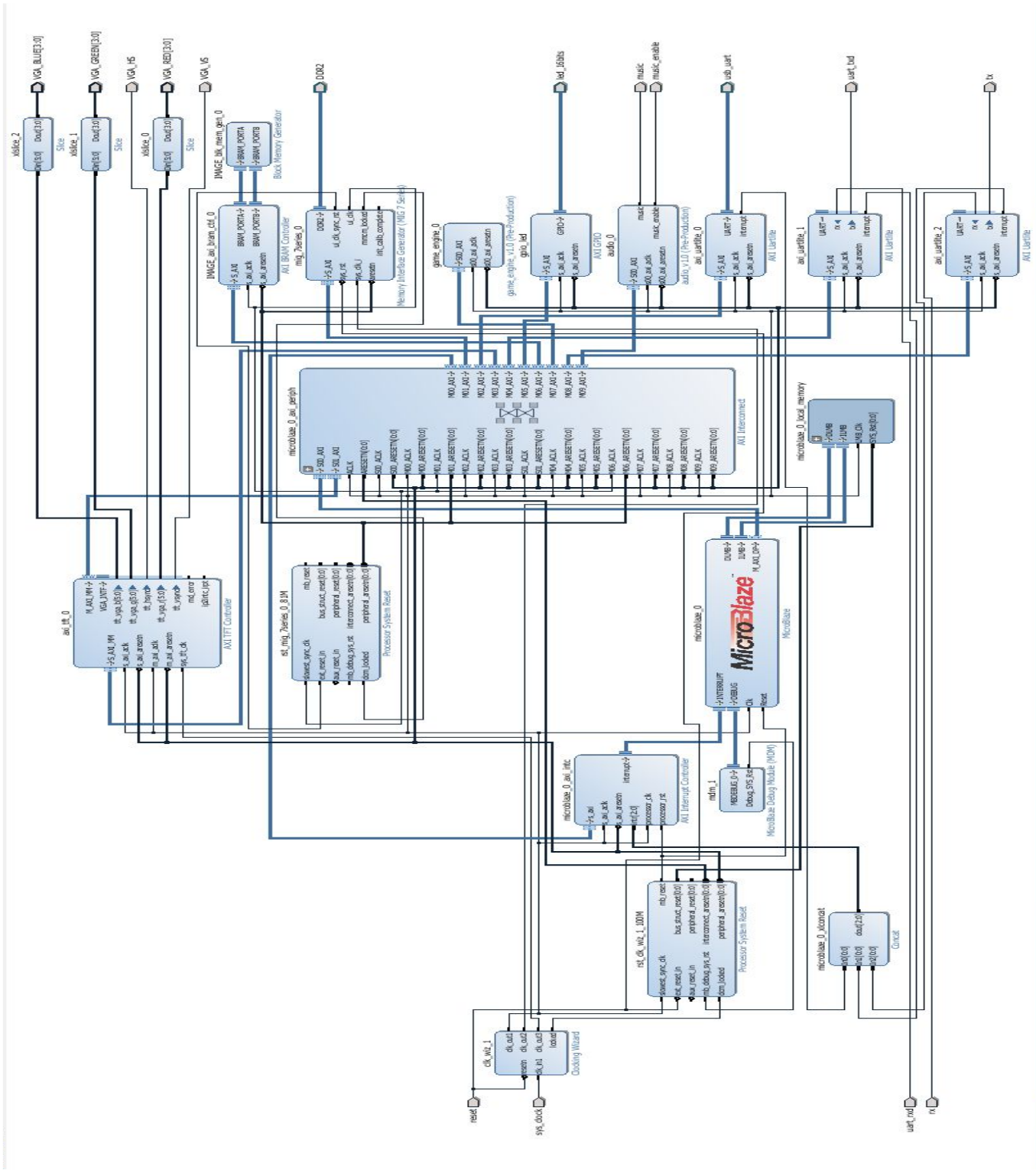
Appendix A: Address Allocation and Mapping



Address Map for processor microblaze_0

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
IMAGE_axi_bram_ctrl_0	0xc0000000	0xc0007fff	S_AXI	MEMORY
audio_0	0x44a20000	0x44a2ffff	S00_AXI	REGISTER
axi_tft_0	0x44a00000	0x44a0ffff	S_AXI_MM	REGISTER
axi_uartlite_0	0x40600000	0x4060ffff	S_AXI	REGISTER
axi_uartlite_1	0x40610000	0x4061ffff	S_AXI	REGISTER
axi_uartlite_2	0x40620000	0x4062ffff	S_AXI	REGISTER
game_engine_0	0x44a10000	0x44a1ffff	S00_AXI	REGISTER
gpio_led	0x40000000	0x4000ffff	S_AXI	REGISTER
microblaze_0_axi_intc	0x41200000	0x4120ffff	s_axi	REGISTER
microblaze_0_local_memory_dlmb_bram_if_cntlr	0x00000000	0x00007fff	SLMB	MEMORY
mig_7series_0	0x80000000	0x87ffffff	S_AXI	MEMORY

Appendix B: Design Block Diagram



Appendix C: Game Console and Game Play Images