Galaxian

CSEE 4840 Embedded System Design

*Department of Computer Science
†Department of Electrical Engineering
†Department of Computer Engineering
School of Engineering and Applied Science,
Columbia University in the City of New York

March 2011

Xiaotian Huo
Computer Engineering Department
xh2144@columbia.edu

Yaolong Gao
Electrical Engineering Department
yg2258@columbia.edu

Qi Ding
Computer Science Department
qd2110@columbia.edu

Feng Ding
Electrical Engineering Department
fd2266@columbia.edu

Abstract:

The main goal of this project is to implement a classic video game Galaxian based on the Altera Cyclone II FPGA board. The project involves both hardware set up and software programming. Moreover, this project is focused on the video signal processing and image display with the FPGA board. This design report will provide the general idea of the project and the details of the implementation.

Introduction:

Galaxian is a 1979 fixed shooter arcade game by Namco and released by Midway Mfg. in the US. The game is a great success on the first generation of Family Computer platform (also known as FCs). This game introduced several firsts to the game industry at that time. Although true color (as opposed to a color overlay for a game that was otherwise black and white) began appearing as early as 1975, Galaxian took graphics a step further with multi-colored animated sprites and explosions, a crude theme song, different colored fonts for the score and high score, more prominent background "music" and the scrolling starfield, and graphic icons that showed the number of ships left and how many rounds the player had completed. Due to these features, rebuilding this game becomes a perfect project to exploit the FPGA board with its powerful image processing functionality.

To implement the Galaxian game, the project will involve both hardware set up and software programming. Especially, due to the complex display of the game, the hardware set up will take the most of the work. Including: the display of images like “spaceship” and “enemy”, the arrangement of enemies in the screen and the set up of PS2 keyboard.

For the software, the difficulty lies in how the enemies fly down and then return to its previous position. The algorithm will have to handle the path of the fly, the position that avoids collision among enemies themselves and how to aim at the player.

The game play is very simple. The player gets 3 lives and wins by eliminating all the enemies and loses if lost all lives. The player can control the spaceship to move right and left with keyboard.

Project Details:

1. General Idea

   The general idea of the project is that the Altera Cyclone II FPGA board is used to display the game graphics and the C code will control the objects' positions and introduce the play rules and simple enemy AI. This kind of infrastructure will help to reduce the hardware complexity and save resources on the board. The following sections will introduce how to organize the hardware and set up the peripherals on the FPGA board in order to realize the game display and the infrastructure the software.

2. Hardware Organization and Peripheral Set Up

   In this project, there are three major hardware devices, i.e. the FPGA board, the display, keyboard and sound box. To make these devices work properly, they should be connected together and set up the FPGA board. The figure below is the block diagram of the basic hardware architecture and thus makes the FPGA board programmable.

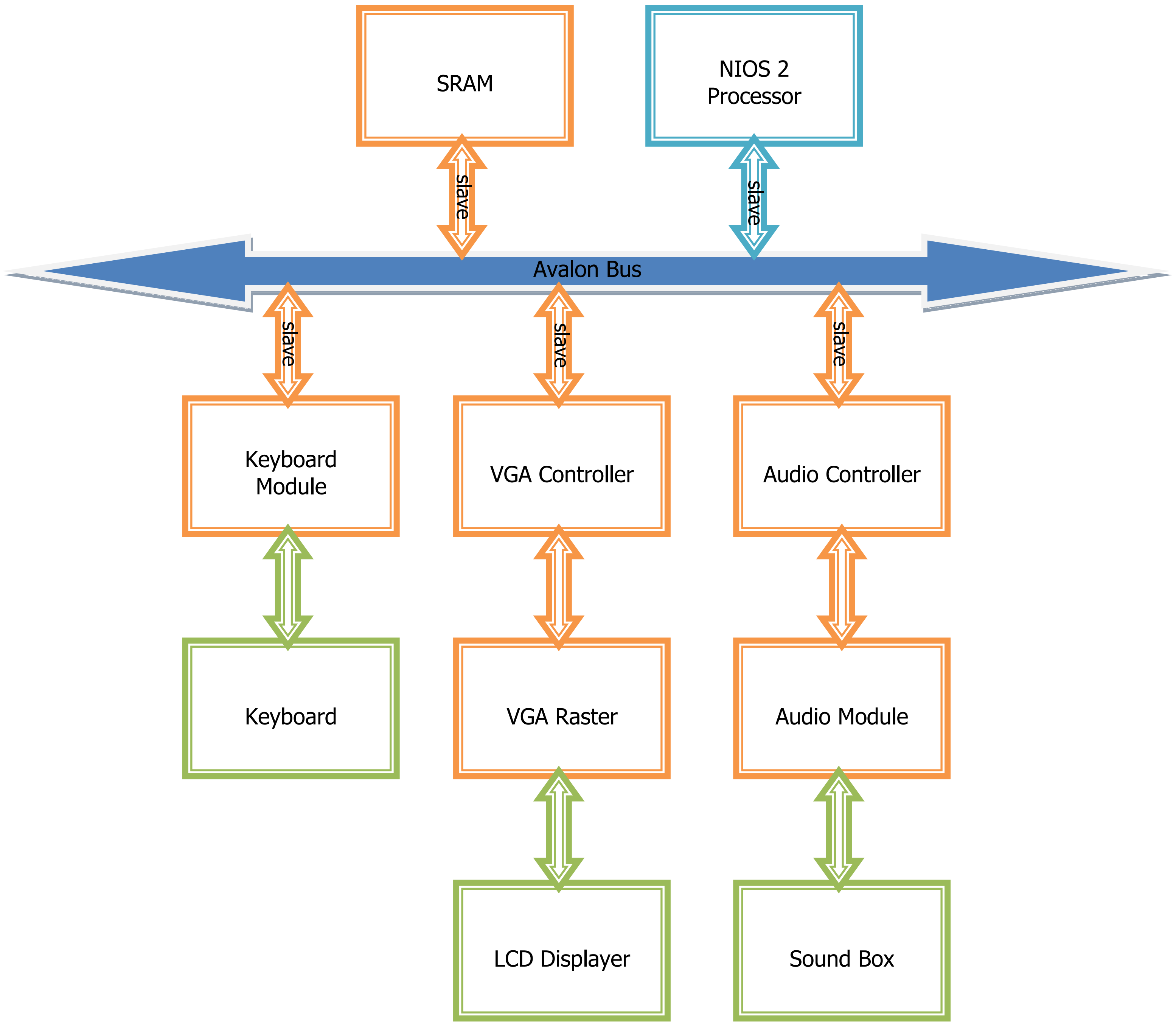


Fig 2-1 Basic Architecture

The green block in fig 2-1 indicates the real hardware devices and all the other blocks together with the Avalon Bus is the architecture of the FPGA board. Here, orange blocks stands for Avalon slaves and the blue block is the CPU. The keyboard module, VGA controller and the Audio controller are interfaces which make the block can communicate through the Avalon bus. Here, the keyboard module uses the given code which provided in Lab 3 hence its interface and control logic written in VHDL code is in a same file. The VGA controller and Audio controller are written in our project. All these three blocks will be bind to the Avalon bus with SOPC builder in the Quartus. The VGA raster is the block which actually communicates with the LCD displayer through VGA port and displays the game graphics. In the VGA block, the basic graphic patterns will be pre-set and the software will send the coordinates of positions to the VGA raster hence realize the control of the game graphics. The audio controller and audio module works in a similar way to the VGA controller and VGA raster. Software will tell the audio block when to play which kind of sound. The details of the implementation will be provided in the upcoming sections.

2.1 Keyboard

The keyboard in this project is used to let the player control the space ship in the game. Though as an input device, it does not have much communication with other block in the FPGA architecture. With the SPOC builder, the data from keyboard can be visited from Avalon bus in the C code. Then, the software will send the corresponding new coordinates to the VGA raster and update the position of the space ship. Thus, the control of the game is realized.

2.2 VGA Blocks

We decided to build a two dimensional matrix that can display up to 8 colors at once. Each element in the matrix is composed of a 3-bit vector. The value of the vector indicates different colors. In our design, the colors are designated like this:

Black: “000”

Blue: “001”

White: “010”

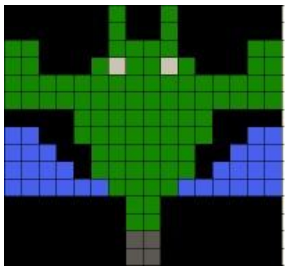
Red : “011”

Yellow: “100”

Brown: 101

Green: “110”

We generate our icons mostly in 16×16 pixels, that means a 16×16 matrix is needed to store the color information. For example, the icon of the “bee enemy” is shown below:

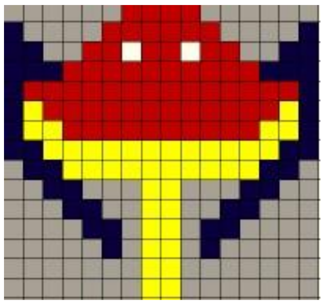


The matrix that describes this bee is like this:

```
Signal sprite_bee :matrix := (
("000","000","000","000","000","000","110","000","000","110","000","000","000","000","000","000"),
("000","000","000","000","000","000","110","000","000","110","000","000","000","000","000","000"),
("110","110","000","000","000","000","110","110","110","110","000","000","000","000","110","110"),
("110","110","000","000","000","110","010","000","110","010","110","000","000","000","110","110"),
("110","110","110","110","110","110","110","110","110","110","110","110","110","110","110","110"),
("110","110","110","110","110","110","110","110","110","110","110","110","110","110","110","110"),
("000","000","000","000","110","110","110","110","110","110","110","110","000","000","000","000"),
("001","001","000","000","110","110","110","110","110","110","110","110","000","000","001","001"),
("001","001","001","000","000","110","110","110","110","110","110","000","000","001","001","001"),
("001","001","001","001","000","110","110","110","110","110","110","000","001","001","001","001"),
("001","001","001","001","001","001","110","110","110","110","001","001","001","001","001","001"),
("000","000","000","000","000","000","000","110","110","000","000","000","000","000","000","000"),
("000","000","000","000","000","000","000","110","110","000","000","000","000","000","000","000"),
("000","000","000","000","000","000","000","110","110","000","000","000","000","000","000","000"),
("000","000","000","000","000","000","000","110","110","000","000","000","000","000","000","000"),
("000","000","000","000","000","000","000","110","110","000","000","000","000","000","000","000")
)
```

Similarly, we can generate the graphics of other icons in the same way:

Big_bee:



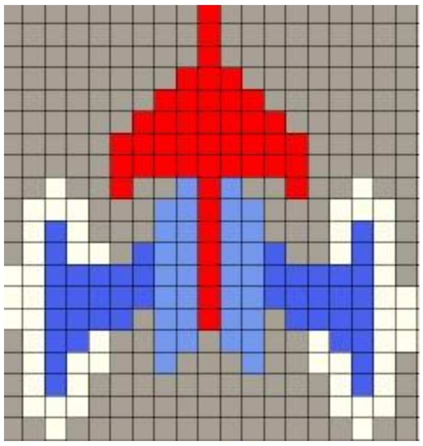
It's matrix description:

Signal sprite_big_bee :matrix := (

```
("000","000","000","000","000","000","011","011","011","011","000","000","000","000","000","000")
("001","000","000","000","000","011","011","011","011","011","011","000","000","000","000","001")
("001","001","000","000","011","011","010","011","011","001","011","011","000","000","001","001")
("001","001","001","011","011","011","011","011","011","011","011","011","011","001","001","001")
("001","011","011","011","011","011","011","011","011","011","011","011","011","011","011","001")
("001","100","011","011","011","011","011","011","011","011","011","011","011","011","100","001")
("001","100","100","011","011","011","011","011","011","011","011","011","011","100","100","001")
("001","001","100","100","100","100","100","100","100","100","100","100","100","100","001","001")
("000","001","001","100","100","100","100","100","100","100","100","100","100","001","001","000")
("000","000","001","001","000","000","000","100","100","000","000","000","001","001","000","000")
("000","000","000","001","001","000","000","100","100","000","000","001","001","000","000","000")
("000","000","000","000","001","001","000","100","100","000","001","001","000","000","000","000")
("000","000","000","000","000","001","000","100","100","000","001","000","000","000","000","000")
("000","000","000","000","000","000","000","100","100","000","000","000","000","000","000","000")
("000","000","000","000","000","000","000","100","100","000","000","000","000","000","000","000")
("000","000","000","000","000","000","000","100","100","000","000","000","000","000","000","000")
)
```

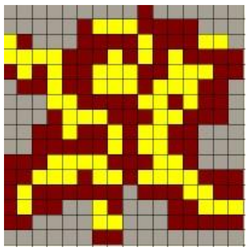
Plane:

We use 20×20 pixel matrix to describe the plane:



```
Signal sprite_plane :matrix := (
("000","000","000","000","000","000","000","000","000","011","000","000","000","000","000","000","000","000","000","000")
("000","000","000","000","000","000","000","000","000","011","000","000","000","000","000","000","000","000","000","000")
("000","000","000","000","000","000","000","000","000","011","000","000","000","000","000","000","000","000","000","000")
("000","000","000","000","000","000","000","000","011","011","011","000","000","000","000","000","000","000","000","000")
("000","000","000","000","000","000","000","011","011","011","011","011","000","000","000","000","000","000","000","000")
("000","000","000","000","000","000","011","011","011","011","011","011","011","000","000","000","000","000","000","000")
("000","000","000","000","000","011","011","011","011","011","011","011","011","011","000","000","000","000","000","000")
("000","000","000","000","000","011","011","011","011","011","011","011","011","011","000","000","000","000","000","000")
("000","000","010","000","000","011","000","000","001","011","001","000","000","011","000","000","010","000","000","000")
("000","010","010","010","000","000","000","001","001","011","001","001","000","000","000","010","010","010","000","000")
("000","010","001","010","000","000","000","001","001","011","001","001","000","000","000","010","001","010","000","000")
("000","010","001","010","000","000","001","001","001","011","001","001","001","000","000","010","001","010","000","000")
("000","010","001","001","001","001","001","001","001","011","001","001","001","001","001","001","001","010","000","000")
("010","010","001","001","001","001","001","001","001","011","001","001","001","001","001","001","001","010","010","000")
("010","010","001","001","001","001","000","001","001","011","001","001","000","001","001","001","001","010","010","000")
("010","010","001","001","010","000","000","001","001","000","001","001","000","000","010","001","001","010","010","000")
("000","010","001","010","010","000","000","001","000","000","000","001","000","000","010","010","001","010","000","000")
("000","010","001","010","000","000","000","000","000","000","000","000","000","000","000","010","001","010","000","000")
("000","010","010","010","000","000","000","000","000","000","000","000","000","000","000","010","010","010","000","000")
("000","000","010","000","000","000","000","000","000","000","000","000","000","000","000","000","010","000","000","000")
)
```

Destroyed enemy:



```
Signal sprite_plane :matrix := (
("000","000","000","000","000","000","000","000","000","101","000","000","000","000","000","000")
("000","000","000","000","000","101","100","100","100","101","101","101","101","000","000","000")
("100","101","000","000","000","101","101","101","101","100","101","101","101","000","000","000")
("100","100","100","100","101","101","101","100","101","100","101","100","101","101","101","101")
("101","100","101","100","101","101","100","100","100","101","101","100","100","100","101","101")
("000","101","101","100","101","101","100","100","100","101","100","100","100","101","101","000")
("000","000","000","100","100","101","101","101","101","100","100","101","100","101","101","000")
("000","000","000","000","100","100","101","101","101","101","100","100","101","101","101","000")
("000","000","101","101","100","101","101","100","101","100","100","101","101","000","000","000")
("000","000","101","101","101","101","101","100","101","100","100","101","101","000","000","000")
("101","101","101","100","100","100","100","101","101","101","100","100","101","000","000","000")
("101","101","100","100","100","101","100","100","101","100","100","100","100","101","101","101")
("101","100","101","101","101","101","101","101","000","101","101","101","100","100","101","101")
("101","100","101","101","000","000","100","101","000","000","000","101","101","100","100","100")
("101","101","000","000","000","000","100","100","000","000","000","101","101","101","101","101")
("000","000","000","000","000","000","101","101","000","000","000","000","000","000","000","000")
)
```

2.3 Audio Blocks

The sound output in the game is implemented with WM8731 Audio CODEC (combining ADCs and DACs) provided on the DE2 board introduced in the Lab 3. However, in our project, we will not synthesis the sound but load the saved music file in directly in the audio module. Another choice is that we can implement the interface of the SD card on the FPGA board. This will add complexity to the project but will provide extra freedom.

3. Software Infrastructure and Implementation

With the properly set up hardware, the software can access to each module through the Avalon bus. The communication with the keyboard is simple and clear. The C code will try to read the input that the player made using the keyboard. Then the program will react to the input with switch/case statement. The program will update the coordinates of the space ship and send it to the VGA raster block. Meanwhile, in the program the corresponding values of the space ship position and check to see it the space ship will be hit by the enemy or not.

Another big part of the software is to control the movement of the enemy. First, the whole enemy will move from right to left in a regular pattern. The program will send the position of the left up corner of the enemy matrix block to the VGA raster and the VGA will update the position whole enemy block. Secondly, the program will control parts of the enemy to fly down from the enemy block and towards the space ship in some kinds of path. This is a little bit difficult since when the enemy flies down, it involves the strange path that the enemy flies and how it returns to its previous position.

Apart from these, the bullet that the space ship shot is another complex design. In the original game, there is at most one bullet in the screen. This feature can be implemented by checking the status of the bullet to see whether it reaches the top of the screen or hit an enemy.

References:

1. *en.wikipedia.org/wiki/Galaxian*