Tagliatela College of Engineering

EE659: System-On-Chip

Fall 2011

(Video Game Project)

Ali Al-Bayaty

Dec. 6, 2011

Table of Contents

Table of Figures

List of Tables

1. Introduction

The aim of this SoC project is to design a video game by using Xilinx FPGA and NES controller. The overall project will be divided into two parts mainly: The first part is to design software for the game using C language, while the second part is to design the hardware for the controller using VHDL. Besides, each part will be divided into different sections, and then those sections will be combined and connected together at the end to give the overall project meaning, which is the video game. The hardware part of the video game project is done by using Xilinx ISE with VHDL while the software part is done by using Xilinx SDK with IP (Intellectual Property) core and C programming language. And, both the hardware and the software parts are implemented on Spartan© 3E 1600 Microblaze Dev Board.

First of all, a hardware interface between the FPGA and NES controller is designed and implemented via VHDL to read the status of the controller's buttons and synchronize the results on Xilinx IP core. Secondly, the first part of the sequential and the related parts should be created to build the graphic interface with the Microblaze processor on Xilinx FPGA board. So, three main signals are generated from Xilinx board to the screen to display a graph on it. Because of the big overall project, the memory space is the most valuable part to use and take benefits. So, the tile map technique is used in order to reduce the size of Xilinx memory. By the same function of output to screen with the capabilities to differentiate and display the sprites and backgrounds together.

Thirdly, the interfaces of NES controller and the tiles/sprites displaying with the Microblaze processor are connected together in this section. These interfaces are achieved by using the IP cores which generated from Xilinx EDK Platform Studio. While, the goals of this lab are to control the hardware that already programmed by VHDL with C code, to display the status of the controller key to the 6-bit LED, while in VHDL they were 8-bit LED, to add the hardware/IP cores to the PLB bus, to use the BRAM instead of the LUT memory for storing the tiles and sprites, and to increase the resolution from 160x120 to 320x240 on the LCD screen. Then, creating moving backgrounds, these backgrounds or screens are loaded from the background tile map which invoked by using the direction buttons from NES controller. The lab combined the IP

cores for the VGA and the controller, which are already created and tested. In order to move the backgrounds for each button pressed, there is a need for large amount of memory to store the feature of each screen. A data structure programming of linked lists for each element on the screen is used to reduce the memory usage.

Fourthly, a moving player is added and created to the previously added parts, and this player is moving without affecting the backgrounds. Because there are two tile maps, one for the background and the other for the foreground, the player is drawn continuously on the second tile map while keeping the first without change after loading the desirable screen. The player should have the following movement, going right and left, climbing the ladder upward and downward, jumping in the same position, left jump, and right jump. Besides, the player has the appearance of dying while stepping over the fire. A customized timer is used with this project in order to slow down the overall motions of the player.

Finally, the final part of the video game is dedicated to creating and moving three types of. These enemies have the same characteristics like the player character has, which are the data structure to build them, the linked list to link them together with the other sprites, to be drawn in the foreground tile map continuously, and to interact with the player and the surrounding environment like the walls and blocks. Each one of these three enemies has a specific movement, horizontal, vertical or diagonal. These enemies have the ability to kill the player by touching or be killed by the player's weapon.

2. Background

2.1 NES Controller

The NES controller has eight buttons (A, B, Select, Start, Up, Down, Left and Right) that can be read using the signal "nes_data" as an output from the controller to the FPGA. When these eight buttons are pressed, the controller generates a negative low output data, so, the FPGA should invert these indications of the pressed buttons. The buttons are stored in a shift register inside the controller before sending to the FPGA by "nes_data"; In short, there are another two input signals to the controller in order to control the synchronization of the pressed buttons and the output data from the controller to the FPGA, which are: "nes_latch" and "nes_clk". The latch used first to fetch the button (A) for 12 µsec, while the other seven buttons are fetched sequentially using "nes_clk" for 7 times 12 µsec period at 50% duty cycle, as shown Figure 1.

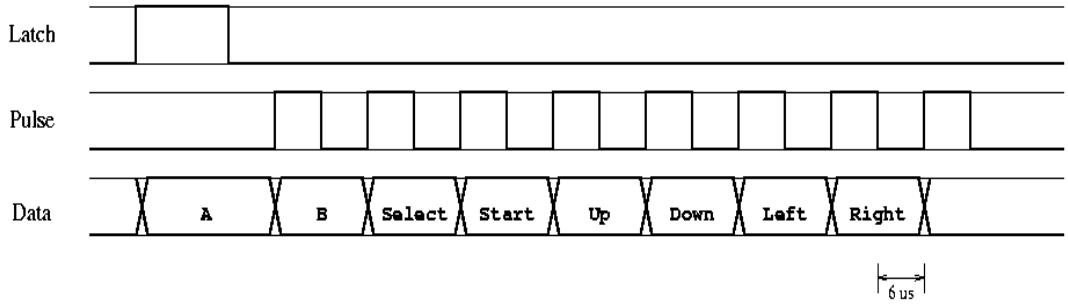

Figure 1: NES Controller Clocking Synchronization

While the hardware interfacing between connecting pins of the FPGA (J1) as described in the UCF file and the controller's pins, as shown in Figure 2:

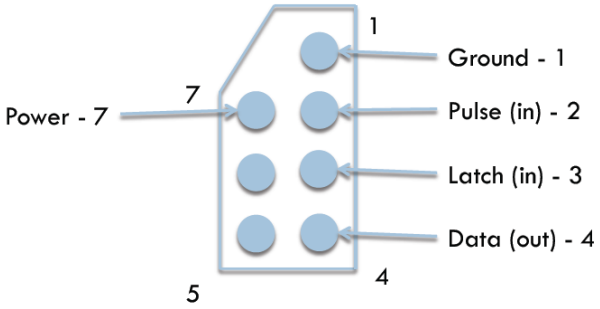

Figure 2: NES Controller Pins-out

The UCF file:

```
NET "reset" LOC = N17;
NET "clk_50" LOC = C9;
NET "led[0]" LOC = D4;
NET "led[1]" LOC = C3;
NET "led[2]" LOC = E6;
NET "led[3]" LOC = D6;
NET "led[4]" LOC = D13;
NET "led[5]" LOC = A7;
NET "led[6]" LOC = G9;
NET "led[7]" LOC = A8;
NET "nes_latch" LOC = N15;
NET "nes_clk" LOC = N14;
NET "nes_data" LOC = E15;
```

The Xilinx FPGA works at 50 MHz, while NES controller deals with 12 μsec "for nes_latch" and 12 μsec period at 50% duty cycle for "nes_clk". Besides, the overall project works with FSM model. So, the program divided into two separated processes, the first process aims to calculate the frequency divider for both clocks, while the second process is for the state machines for the checking of the controller's eight buttons, as illustrated by Figure 3.

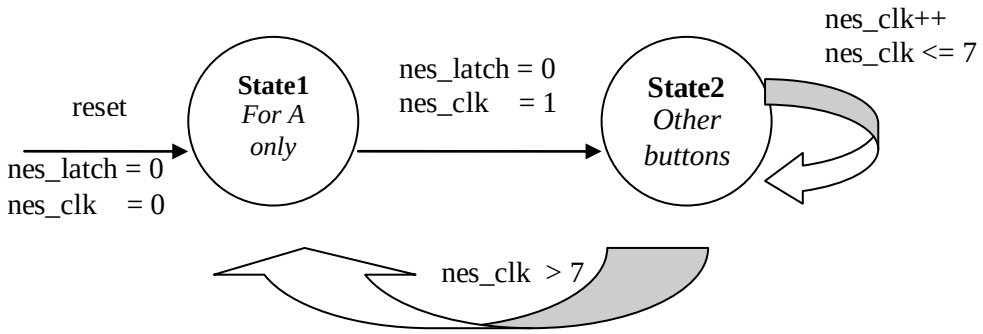

Figure 3: NES Controller FSM

2.2 VGA

The Spartan-3E board includes a VGA display port and DB15 connector, indicated as below. Connect this port directly to most PC monitors or flat-panel LCD displays using a standard monitor cable, as shown in Figure 4:

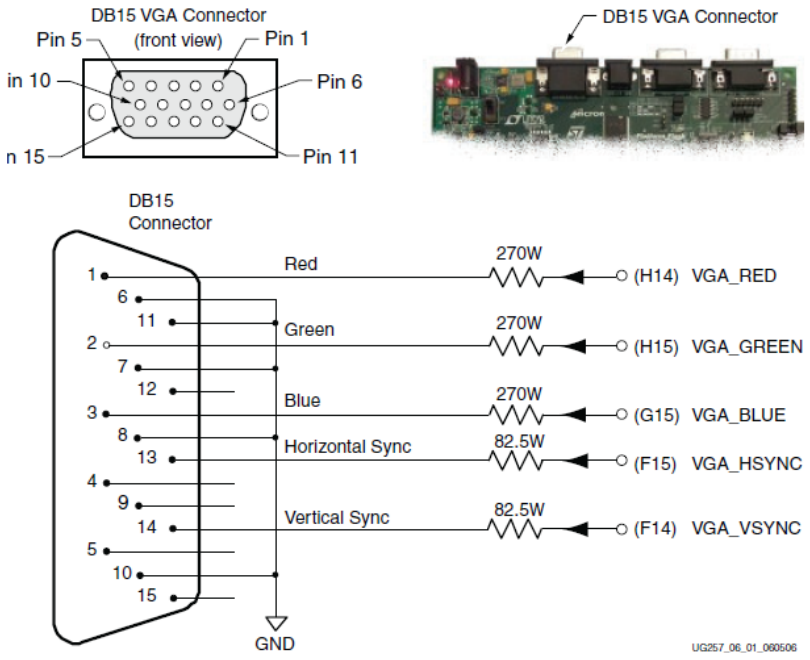

Figure 4: Xilinx VGA Connections

The Spartan-3E FPGA controls/pins five VGA signals: Red (R), Green (G), Blue (B), Horizontal Sync (HS), and Vertical Sync (VS), all available on the VGA connector. Each color line has a series resistor to provide 3-bit color, with one bit each for Red, Green, and Blue. The series resistor uses the 75 resistor VGA cable termination to ensure that the color signals remain in the VGA-specified 0V to 0.7V range. The HS and VS signals are TTL level. Drive the R, G, and B signals High or Low to generate the eight possible colors as shown in Table 1:

Table 1: RGB Color Combinations

Colors	Red (R)	Green (G)	Blue (B)
Black	0	0	0
Blue	0	0	1
Green	0	1	0
Cyan	0	1	1
Red	1	0	0
Magenta	1	0	1
Yellow	1	1	0
White	1	1	1

VGA signal timing is specified, published, copyrighted, and sold by the Video Electronics Standards Association (VESA). The following VGA system and timing information is provided as an example of how the FPGA might drive a VGA monitor in 640 by 480 mode. CRT-based VGA displays use amplitude-modulated, moving electron beams (or cathode rays) to display information on a phosphor-coated screen. LCD displays use an array of switches that can impose a voltage across a small amount of liquid crystal, thereby changing light permittivity through the crystal on a pixel-by-pixel basis. Although the following description is limited to CRT displays, LCD displays have evolved to use the same signal timings as CRT displays. Consequently, the following discussion pertains to both CRTs and LCD displays.

Within a CRT display, current waveforms pass through the coils to produce magnetic fields that deflect electron beams to transverse the display surface in a "raster" pattern, horizontally from left to right and vertically from top to bottom. As shown in Figure 5, information is only displayed when the beam is moving in the "forward" direction—left to right and top to bottom—and not during the time the beam returns back to the left or top edge of the display. Much of the potential display time is therefore lost in "blanking" periods when the beam is reset and stabilized to begin a new horizontal or vertical display pass. The signal timings are derived for a 640-pixel by 480-row display using a 25 MHz pixel clock and 60 Hz ±1 refresh. As shown in figure 2, the relation between each of the timing symbols. The timing for the sync pulse width (TPW) and front and back porch intervals (TFP and TBP) are based on observations from various VGA displays. The front and back porch intervals are the pre- and post-sync pulse times. Information cannot be displayed during these times.

Two VHDL files will be used, one for generating the required signals, while the other for displaying the desirable data on the screen. The first file gives HSYNC and VSYNC signals to the screen with the locations of the current pixel as "pixel_x" and "pixel_y" signals to the second file. Plus, this file is responsible to generate the 25MHz clock from the system clock, so, there is no need for frequency divider in the second file. The second file takes the current pixel locations and converts them to one dimension that will be used as an index to access the tile map as illustrated in Equation (1), this tile map used to give the desirable colors to the screen through RGB signal, as shown in Figure 6.

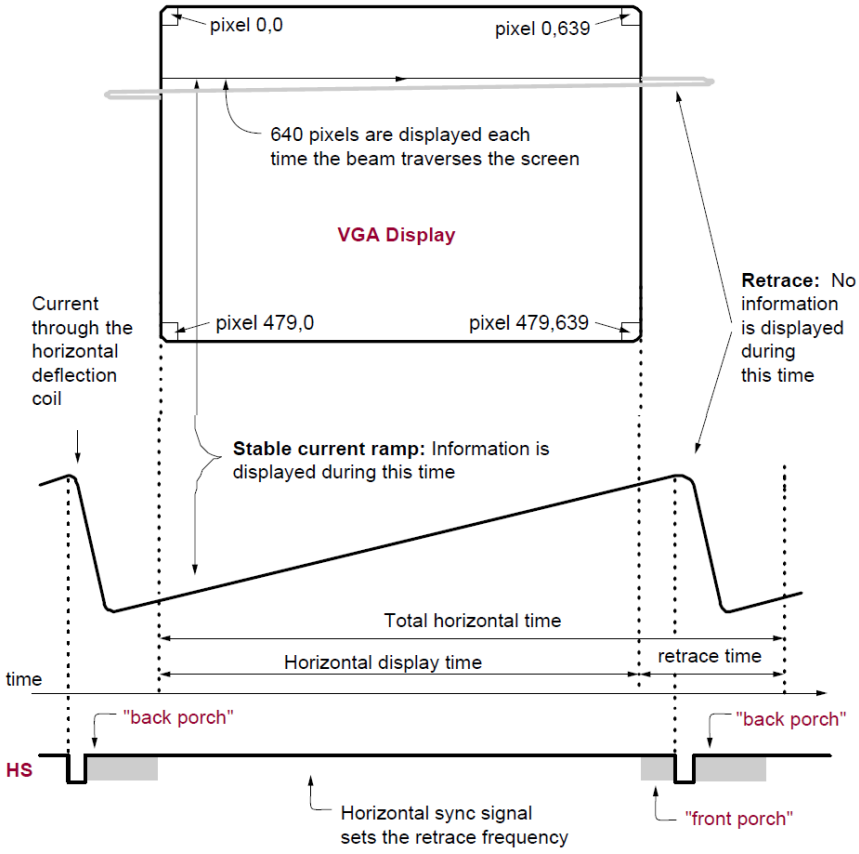

Figure 5: VGA Display Timing

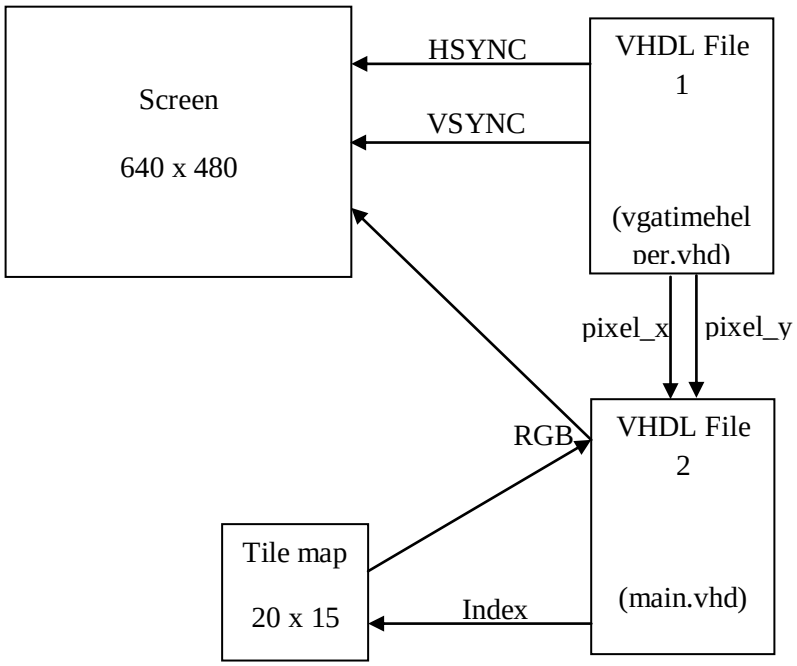

Figure 6: VHDL Files Relationship

```
x = pixel_x / 32        ( Mapping of x-axis, 640/20 = 32 )
y = pixel_y / 32        ( Mapping of y-axis, 640/20 = 32 )
index1 = y * (Total No. of Col) + x
       = y * 20 + x        …… Equation (1)
```

2.3 Tile Maps

In order to display the final pictures on the screen that consist of a collection of characters and backgrounds, there is a need for palettes to differentiate these objects, and so, they will not interfere and over-write each other. These palettes can be done using three different types of maps, each one has the ability to draw the dedicated objects on the screen, these maps are: foreground, background and sprites. Foreground map consists of the characters only as objects above the backgrounds, while the background map consists of different sets of objects to be drawn behind the characters from the foreground map, both maps are pointed to the third one which has the list of all character and background objects as sets of pixels. The third map called the sprites map, and consists of 31 rows by 64 columns, each row represents either a character or a background, while the columns represent the linear mapping of a 2D array of 8x8. As a result, each character or background can be represented by 8x8 array for simplicity.

In order to draw in a specific location, that provided by the VGA time helper file, with the information provided by objects either by foreground or background maps, there is a need for indexing formulas to access each location separately and draw the accurate sets of pixels on screens. Equation (1) gives the exact location and mapping from 640x480 screen resolution to 20x15 tile-maps resolution. While Equation (2) and (3) indicates the locations of rows, a character and background, based on the information provided by the contents of index1 from foreground and background tile-maps:

```
index2 = pixel_y(4 downto 0)/4 * (8) + pixel_x(4 downto 0)/4 +
         foreground(index1)*64;      …… Equation (2)

index3 = pixel_y(4 downto 0)/4 * (8) + pixel_x(4 downto 0)/4 +
         background(index1)*64;      …… Equation (3)
```

Now, after getting the indices of character and background in the sprites map, their contents of pixels are available and the decision of their drawing is based on the transparency bit (the fourth bit) in the character pixel sets, if it's clear (0), means to draw the character pixel regardless the background pixel, but if it's set (1), means to ignore the character pixel and draw the background one instead. So, the relationship of VHDL files including the tile-maps and sprites are illustrated in Figure 7.

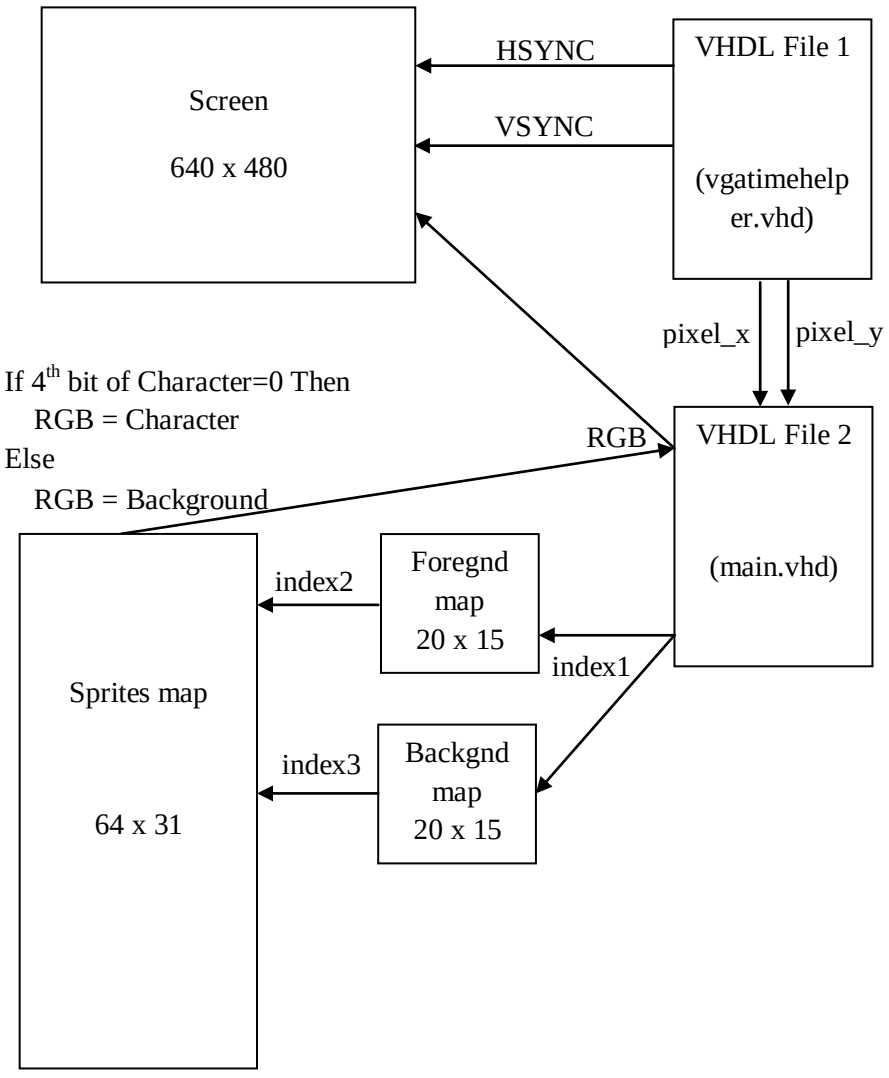

Figure 7: VHDL Files and Maps Relationship

2.4 IP (Intellectual Property) Core

In order to control the hardware with C code, there is a need for the IP core that will be attached to the slave PLB. This IP core is is based on the VHDL files to control the overall core. These files have the appropriate hierarchy to operate in order, which are:

1. LAB.vhdl: the original file of the previous lab.
2. user_logic.vhd: the file that creates and controls the signals and registers from the previous file and passed to the next file.
3. LAB_ip_core.vhd: the last file that will be used to create the desirable IP core.

All the signals (ports) from the first file are used either as registers, which are the communication channels between the hardware and C code through the PLB bus, or as real hardware pins, which later be written in UCF file that generated and modified from the Xilinx EDK Platform Studio. Then, creating two IP cores, one for the VGA and the other for the NES controller. The VGA part has some modifications in the original VHDL file, which are:

1. Using empty tile and sprite maps, which to be filled from the C code.
2. Using the BRAM instead of the LUT memory by creating three separated processes without sensitivity lists.
3. Changing the resolution from 160x120 (20x15 tiles) to 320x240 (40x30 tiles) by transforming from Equation (1) to Equation (4).

```
x = pixel_x / 16        ( Mapping of x-axis, 480/30 = 16 )
y = pixel_y / 16        ( Mapping of y-axis, 640/40 = 16 )
index1 = y * (Total No. of Col) + x
       = y * 40 + x                    …… Equation (4)
```

4. Changing the indices for the tile and sprite maps by transforming from Equation (2) & (3) to Equation (5) & (6), as shown below:

```
index2 = pixel_y(3 downto 0)/2 * (8) + pixel_x(3 downto 0)/2 +
         foreground(index1)*64;      …… Equation (5)

index3 = pixel_y(3 downto 0)/2 * (8) + pixel_x(3 downto 0)/2 +
         background(index1)*64;      …… Equation (6)
```

While the second IP core, NES controller, does not have any changes in the original VHDL file, except that 8-bit LED will be 6-bit LED display.

Table 2 shows the differences by using the signals (ports) in the original files and how to convert them in IP cores, either to registers or UCF; Registers if there is a need for a storage space between the VHDL and C code, and UCF for a hardware pins.

Table 2: Registers and UCF in IP Core

IP Core	Port (VHDL)	Type	Register	UCF	Internal Signal
VGA	clk	in: 1-bit			Bus2IP_Clk
	hsync	out: 1-bit		Yes	
	vsync	out: 1-bit		Yes	
	rgb	out: 3-bit		Yes	
	we	in: 1-bit	Yes		
	add_bus1	in: 11-bit	Yes		
	add_bus2	in: 11-bit	Yes		
	add_bus3	in: 11-bit	Yes		
	data_bus1	in: 5-bit	Yes		
	data_bus2	in: 5-bit	Yes		
	data_bus3	in: 4-bit	Yes		
Controller	clk_50	in: 1-bit			Bus2IP_Clk
	reset	in: 1-bit		Yes	
	led	out: 8-bit	Yes		
	nes_latch	out: 1-bit		Yes	
	nes_clk	out: 1-bit		Yes	
	nes_data	in: 1-bit		Yes	

2.5 Linked Lists Data Structure

In order to move between different screens, there is a need for different sets of sprites for each screen, which means different background tile maps. So, if this way of technique is implemented, there is more memory required for each screen that has a unique and a separated background tile map. On the other hand, this project can be implemented for the moving screens that have a unified background tile map that changed for each update of the different screens, this technique that can be done by using the linked lists, each linked list can be represented by an individual element in the screen. Hence, if the first screen has three different drawing positions of the same sprite, this means only one element is used for that repeated sprite. The linked lists are implemented by using C code from Xilinx SDK, with the following structure:

```
struct element{
      char x, y;      // For X and Y dimensions
      char direction; // Drawing Direction,0:left-to-right,1:up-to-down
      char repeat;    // How many times to draw the same sprite
      char sprite;    // The location of sprite in the Sprites Map
      struct element* next;  // The linked list to the next element
};
```

Besides, there is another linked list for the consecutive screens, which is:

```
struct screen{
      struct element *head; // The head that points to the first
                            // element of the same screen
      struct screen *next;  // The linked list to the next screen
};
```

The screen resolution resizing is already achieved from the previous lab, so now, the tile maps for the background and the foreground are all become 40x30, which means 320x240 screen resolution.

And, to draw different six screens with different elements organization for each, the sequence of drawing these screens with their elements is illustrated with the following steps:

1. Declaring and initializing the screens counter to one.
2. Declaring Background and Foreground tile maps as 40x30, without initialization.
3. Declaring and initializing the Sprites map for 32 sprites.
4. Declaring and memory allocating the pointers for the element and screen structures.
5. Assigning the desirable data to the elements to be displayed, and assigning these elements to the first screen.
6. Loading the screen.
7. Checking the pressed button from the controller. If it is pressed, go to step (7) and increment the counter for the next screen, else keep checking.
8. Clearing the previous screen and its elements.
9. Assigning the next desirable data to the elements to be displayed, and assigning these elements to the next screen.
10. Go to step (5).
11. If the counter reaches the sixth screen, keeps the current screen and do nothing.

In order to move the player in the foreground tile map regardless of the background tile map, there is a need for another drawing function to be used with the foreground only. This function works like the previous project's function of drawing the screens from the background tile map, but this time, it will be used with the player, i.e. the foreground. The same structure is used for the units to be set with the screens is used again in the player characteristics setting, as shown below:

```
struct element{
      char x, y;       // For X and Y dimensions
      char direction; // Drawing Direction,0:left-to-right,1:up-to-down
      char repeat;     // How many times to draw the same sprite
      char sprite;     // The location of sprite in the Sprites Map
      struct element* next;  // The linked list to the next element
};
```

And, because the player consists of ten sprites, the sequence of filling the player structure is repeated for ten times. On the other hand, a function called `constructor()` is implemented to achieve and simplify the operations of filling the structure with the required parameters and sprites.

As mentioned before, there is a need for drawing the player on the foreground tile map regardless of the background. Besides, the player has the customized movements in all directions plus jumping and dying. This function plays the double role of drawing the player and moving it, all these operations are done by making the function more generic because of its parameters, as shown below:

```
void paintfgd( struct element *units, int shiftx, int shifty, int clear )
```

where:
units: for player's linked list of ten sprites.

shiftx: zero means there is no movement, positive number means to move the player to the right, while the negative means to move to the left.

shifty: zero means there is no movement, positive number means to move the player downward, while the negative means to move upward.

clear: one means to clear the foreground tile map and used with the movements, while zero means to append to the foreground without clearing the player and used with dying and moving enemies (next project).

In order to show the effects of movement, some sprites are changed while moving the player to right or left, and even in jumping. These effects are applied to the legs only due to the limited resources of the sprites. And, several and continuous tests are used for the player in the combination with the background tile map for the sounding objects. For example, the player is always in the falling-free mode, and he stops when there is no clear sprite below him in the background tile map. In addition, three different jumps are implemented, up-jump, right-jump and left-jump. All of these different jumps are testing the three sprites above the player, and

increment the distance of jumping. For example, if there are only two clear sprites above the player, the distance of jumping will be only two. There is another testing condition which used to test the incoming data from the controller. If there is no data, that means there is no movement, then update the player with the normal legs.

In general, there are three conditions for the enemy in each screen or level:

1. To create the enemy, then disable this condition and enable the other two below.
2. To move the enemy continuously with its specific movement that depends on the current screen or level.
3. To kill the enemy if it has been touched by the character's weapon, then disable item two and three for this screen.

And, in order to make the game more exciting and challenging, two flames are added to the game and represented as enemies also, which have the same characteristics:

1. The flames are appeared in desirable screens with defined positions.
2. The flames are continuously moving when the screen is loaded.
3. Each flame considered as a non-destroyable enemy
4. The character is killed when touching the flame.
5. The character's weapon is destroyed when touching the flame.

3. Design Methodology

3.1 Software Part

1. Generating the “nes_latch” clock of 12 µsec period at 100% duty cycle.
2. Generating the “nes_clk” clock of 12 µsec period at 50% duty cycle.
3. The VHDL file is based on the FSM model in order to detect NES controller’s buttons.
4. The 25 MHz VGA clock is generated from the VHDL helper file, so, the main VHDL file is purposed to deal with a tile map only (faster processing and handling).
5. Converting the 2D to 1D in order to access the tile map (faster access with few variables).
6. Minimizing the number of lines in the main VHDL file due to the simplified equation.
7. There is no need for the PROCESS statements in the main VHDL file, because everything work in combinational logic and not sequential.
8. Filling the tiles and sprites memory spaces at the beginning of the software.
9. Using IP core’s registers to communicate with the hardware, instead of C variables.
10. Declaring the tile maps for the background and foreground only, without initialization.
11. Declaring and initializing the sprites map only.
12. Using FOR loop as desirable delay, no need for extra timer setting (more space).

3.2 Hardware Part

1. Both “nes_latch” and “nes_clk” are generated from the system clock of 50MHz.
2. Interfacing NES controller with FPGA using (J1) pins.
3. Board’s LEDs work as indications to the pressed buttons of NES controller.
4. Using small tile map in order to reduce the amount of Xilinx memory and keep it for future project development.
5. Interfacing the LCD screen directly to Xilinx VGA port with the designated pins.
6. Able to see that each cell in the tile map is equal to 32x32 pixels on the LCD screen.
7. Creating separated BRAMs by using separated processes.
8. Using BRAM to store the tile and sprite maps, and reduce the size of the LUT memory.
9. Changing the display resolution from the 160x120 to 320x240.
10. Using NES controller to change the player directions and behavior.

4. Life Long Learning

1. Taking the advantage of frequency divider technique to generate desirable clocks.
2. In order to make the program simpler with no extra signals (more hardware), “ness_latch” and “nes_clk” changed to IN and OUT mode.
3. In order to make the program simpler and without extra signals (more hardware), only the equation and output signals are used in the VHDL main file (without PROCESSES).
4. Using CASE statements instead of IF statements to decrease the complexity of the hardware.
5. Because bitwise shifting considered as operations. So, it also required more hardware to be achieved. Thus, direct selections of bits are designed for “pixel_x” and “pixel_y” in order to reduce the hardware.
6. Accessing the tile-maps and sprites with mathematical expressions, which yield to an ease of manipulations but with complex hardware.
7. Optimize the software by using one FOR loop to fill the maps.
8. Instead of using divisions in Equation (5) and (6), direct bits mapping is used to reduce the generated hardware.
9. Most of the variables in C code are global, to manage the memory size of the SRAM.
10. Using the same structure for screen’s elements, the character, the character’s weapon, the flames and the enemies, less memory usage.
11. Separating the drawing and clearing of the screens and player as two functions, one for foreground tile map and other for background.
12. Using the controller’s arrows to change the directions of the player, B button to jump, SELECT button to reload the same level and START button to reload the first level.
13. Using a desirable FOR loop as a global timer for the overall game, less hardware usage.
14. Using a desirable FOR loop as a delay for a strawberry explosion.

5. Source Code

(This section is omitted, in order to reduce the overall number of pages!)

5.1 Hardware Part (VHDL)

5.1.1 NES Controller

5.1.2 Interfacing the GPIO

A. No Device Driver

B. Low Level Driver

C. High Level Driver

5.1.3 Interfacing with Timer IP core

5.1.4 Interfacing with Timer IP core using Interrupts

5.1.5 VGATimeHelper (Screen Coordination and Clock Generator)

5.1.6 VGAMain (Locations for characters and backgrounds in sprites map)

5.1.7 VGA IP Core

5.1.8 NES Controller IP Core

5.1.9 UCF

5.2 Software Part (C Code)

```
// Global Variables
// MACROs
// Foreground tile-map (40x30): objects
// Background tile-map (40x30): screens
// Sprites map (64x32): characters and weapons
// Structure of Elements
// Structure of Screens
// BRAMs
// Functions: initializations, clearing and filling
// Functions: character’s different movements
// main()
```

6. Results

The results are tested as followed:

1. Initiate the reset button, to reset the two processes (freq divider, FSM).
2. All the LEDs are ON when the NES controller not connected to FPGA, means that all states are working properly after inverting the zero level inputs (open) to ON.
3. When the controller connected to FPGA, the LEDs work properly according to sequences of the "nes_latch" and "nes_clk": (A➔B➔Select➔Start➔Down➔Up➔Left➔Right)
4. The Timing Summary:

```
---------------
Speed Grade: -4
   Minimum period: 8.270ns (Maximum Frequency: 120.917MHz)
   Minimum input arrival time before clock: 3.385ns
   Maximum output required time after clock: 4.368ns
   Maximum combinational path delay: No path found
```

5. The project worked properly for the three different equivalent methods. And, the results are tested as shown in Table 3.

Table 3: Results of the Equivalent Methods

Methods	No. of 4-input LUT	No. of occupied Slices
1 "*as Equation (1)"*	117	73
2 "*Replacing Division by Shifting"*	117	73
3 *"Replace Shifting by direct bits wiring"*	113	71

```
Methods Timing Summary:
---------------
Speed Grade: -4
(Method 1 & 2)
     Minimum period: 5.449ns (Maximum Frequency: 183.520MHz)
     Maximum output required time after clock: 18.420ns
(Method 3)
     Minimum period: 5.689ns (Maximum Frequency: 175.783MHz)
     Maximum output required time after clock: 13.837ns
```

6. The project worked properly by drawing the characters without overlapping with the backgrounds. And, the results are the same for both methods, as shown in Table 4.

Table 4: Results of Characters Methods

Methods	No. of 4-input LUT	No. of occupied Slices
1 *"Normal Division"*	511	323
2 *"Replacing Division by direct bits wiring"*	511	323

```
Methods Timing Summary:
---------------
Speed Grade: -4
(Method 1 & 2)
      Minimum period: 5.897ns (Maximum Frequency: 169.584MHz)
      Minimum input arrival time before clock: No path found
      Maximum output required time after clock: 20.755ns
      Maximum combinational path delay: No path found
```

7. The project worked properly with the desirable results for VGA and NES controller IP cores. The results can be classified as in Figures 8, 9, 10, 11, 12, and 13 with their Timing Summary.

7.1 VGA hardware results for 160x120 resolution:

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	33	29,504	1%	
Number of 4 input LUTs	2,388	29,504	8%	
Number of occupied Slices	1,218	14,752	8%	
Number of Slices containing only related logic	1,218	1,218	100%	
Number of Slices containing unrelated logic	0	1,218	0%	
Total Number of 4 input LUTs	2,406	29,504	8%	
Number used as logic	984			
Number used as a route-thru	18			
Number used for Dual Port RAMs	1,404			
Number of bonded IOBs	50	250	20%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	6.46			

Figure 8: VGA (160x120) Device Utilization

```
Synthesizing Unit <vga>.
    Related source file is "C:/proj5/vga/main.vhd".
WARNING:Xst:653 - Signal <reset> is used but never assigned. This sourceless signal
WARNING:Xst:646 - Signal <pixel_y<1:0>> is assigned but never used. This unconnected
WARNING:Xst:646 - Signal <pixel_x<1:0>> is assigned but never used. This unconnected
WARNING:Xst:646 - Signal <index2<31:11>> is assigned but never used. This unconnecte
WARNING:Xst:646 - Signal <index1<31:11>> is assigned but never used. This unconnecte
WARNING:Xst:646 - Signal <index<31:9>> is assigned but never used. This unconnected
    Found 300x5-bit dual-port RAM <Mram_back_map> for signal <back_map>.
    Found 2048x4-bit dual-port RAM <Mram_sprites> for signal <sprites>.
    Found 2048x4-bit dual-port RAM <Mram_sprites_ren> for signal <sprites>.
    Found 300x5-bit dual-port RAM <Mram_fore_map> for signal <fore_map>.
    Found 10-bit adder for signal <index$add0000> created at line 112.
    Found 5x5-bit multiplier for signal <index$mult0000> created at line 112.
    Found 11-bit adder for signal <index1$add0000> created at line 114.
    Found 11-bit adder for signal <index2$add0000> created at line 117.
    Found 6-bit adder for signal <index2$add0001> created at line 117.
    Summary:
        inferred   4 RAM(s).
        inferred   4 Adder/Subtractor(s).
        inferred   1 Multiplier(s).
Unit <vga> synthesized.

=========================================================================
```

Figure 9: VGA (160x120) BRAM Components

```
Timing Summary:
---------------
Speed Grade: -4
     Minimum period: 5.939ns (Maximum Frequency: 168.392MHz)
     Minimum input arrival time before clock: 5.971ns
     Maximum output required time after clock: 18.930ns
     Maximum combinational path delay: No path found
```

7.2 VGA hardware results for 320x240 resolution:

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	27	29,504	1%	
Number of 4 input LUTs	4,019	29,504	13%	
Number of occupied Slices	2,046	14,752	13%	
Number of Slices containing only related logic	2,046	2,046	100%	
Number of Slices containing unrelated logic	0	2,046	0%	
Total Number of 4 input LUTs	4,042	29,504	13%	
Number used as logic	1,495			
Number used as a route-thru	23			
Number used for Dual Port RAMs	2,524			
Number of bonded IOBs	54	250	21%	
Number of BUFGMUXs	1	24	4%	
Number of MULT18X18SIOs	1	36	2%	
Average Fanout of Non-Clock Nets	6.76			

Figure 10: VGA (320x240) Device Utilization

```
Synthesizing Unit <vga>.
    Related source file is "C:/proj5/vga/main.vhd".
WARNING:Xst:653 - Signal <reset> is used but never assigned. This sourceless signal
WARNING:Xst:646 - Signal <pixel_y<0>> is assigned but never used. This unconnected s
WARNING:Xst:646 - Signal <pixel_x<0>> is assigned but never used. This unconnected s
WARNING:Xst:646 - Signal <index2<31:11>> is assigned but never used. This unconnecte
WARNING:Xst:646 - Signal <index1<31:11>> is assigned but never used. This unconnecte
WARNING:Xst:646 - Signal <index<31:11>> is assigned but never used. This unconnected
    Found 1200x5-bit dual-port RAM <Mram_back_map> for signal <back_map>.
    Found 2048x4-bit dual-port RAM <Mram_sprites> for signal <sprites>.
    Found 2048x4-bit dual-port RAM <Mram_sprites_ren> for signal <sprites>.
    Found 1200x5-bit dual-port RAM <Mram_fore_map> for signal <fore_map>.
    Found 12-bit adder for signal <index$add0000> created at line 109.
    Found 6x6-bit multiplier for signal <index$mult0000> created at line 109.
    Found 11-bit adder for signal <index1$add0000> created at line 111.
    Found 11-bit adder for signal <index2$add0000> created at line 114.
    Found 6-bit adder for signal <index2$add0001> created at line 114.
    Summary:
        inferred   4 RAM(s).
        inferred   4 Adder/Subtractor(s).
        inferred   1 Multiplier(s).
Unit <vga> synthesized.
```

Design Summary

Figure 11: VGA (320x240) BRAM Components

```
Timing Summary:
---------------
Speed Grade: -4
     Minimum period: 10.869ns (Maximum Frequency: 92.002MHz)
     Minimum input arrival time before clock: 6.356ns
     Maximum output required time after clock: 25.879ns
     Maximum combinational path delay: No path found
```

7.3 NES Controller hardware results:

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	70	29,504	1%	
Number used as Flip Flops	69			
Number used as Latches	1			
Number of 4 input LUTs	131	29,504	1%	
Number of occupied Slices	91	14,752	1%	
Number of Slices containing only related logic	91	91	100%	
Number of Slices containing unrelated logic	0	91	0%	
Total Number of 4 input LUTs	167	29,504	1%	
Number used as logic	131			
Number used as a route-thru	36			
Number of bonded IOBs	13	250	5%	
IOB Latches	8			
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.48			

Figure 12: NES Controller Device Utilization

```
Timing Summary:
---------------
Speed Grade: -4
   Minimum period: 8.270ns (Maximum Frequency: 120.917MHz)
   Minimum input arrival time before clock: 3.385ns
   Maximum output required time after clock: 4.368ns
   Maximum combinational path delay: No path found
```

7.4 EDK Platform Studio (IP Cores) result:

Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	3,044	29,504	10%	
Number used as Flip Flops	3,035			
Number used as Latches	9			
Number of 4 input LUTs	5,729	29,504	19%	
Number of occupied Slices	4,667	14,752	31%	
Number of Slices containing only related logic	4,667	4,667	100%	
Number of Slices containing unrelated logic	0	4,667	0%	
Total Number of 4 input LUTs	5,908	29,504	20%	
Number used as logic	3,859			
Number used as a route-thru	179			
Number used for Dual Port RAMs	1,724			
Number used as Shift registers	146			
Number of bonded IOBs	66	250	26%	
IOB Flip Flops	31			
IOB Master Pads	1			
IOB Slave Pads	1			
Number of ODDR2s used	22			
Number of RAMB16s	21	36	58%	
Number of BUFGMUXs	5	24	20%	
Number of DCMs	2	8	25%	
Number of BSCANs	1	1	100%	
Number of MULT18X18SIOs	3	36	8%	
Average Fanout of Non-Clock Nets	4.02			

Figure 13: IP Cores Device Utilization

8. The project worked properly as planned for the player's movements and action, like throwing the berries. Different movements are tested with different combination of the pressed buttons of the NES controller. Plus, different sprites of player's legs are associated with the specified motions. The player is able to switch between the screens (levels) of the game, when he reaches the specific location. The indication of death is worked properly as a ring and wings, too.

Three types of enemies are displayed and tested with each screen/level. Each type of enemy has its own sprite and directional movement (horizontal, vertical or diagonal). Also, flames are considered to be as enemies and have only one type of movement, which is the horizontal one.

Six screens (levels) are tested and worked perfectly, plus the seventh one worked as an indication of completing and winning the game, as shown in Figures 14, 15, 16, 17, and 18.

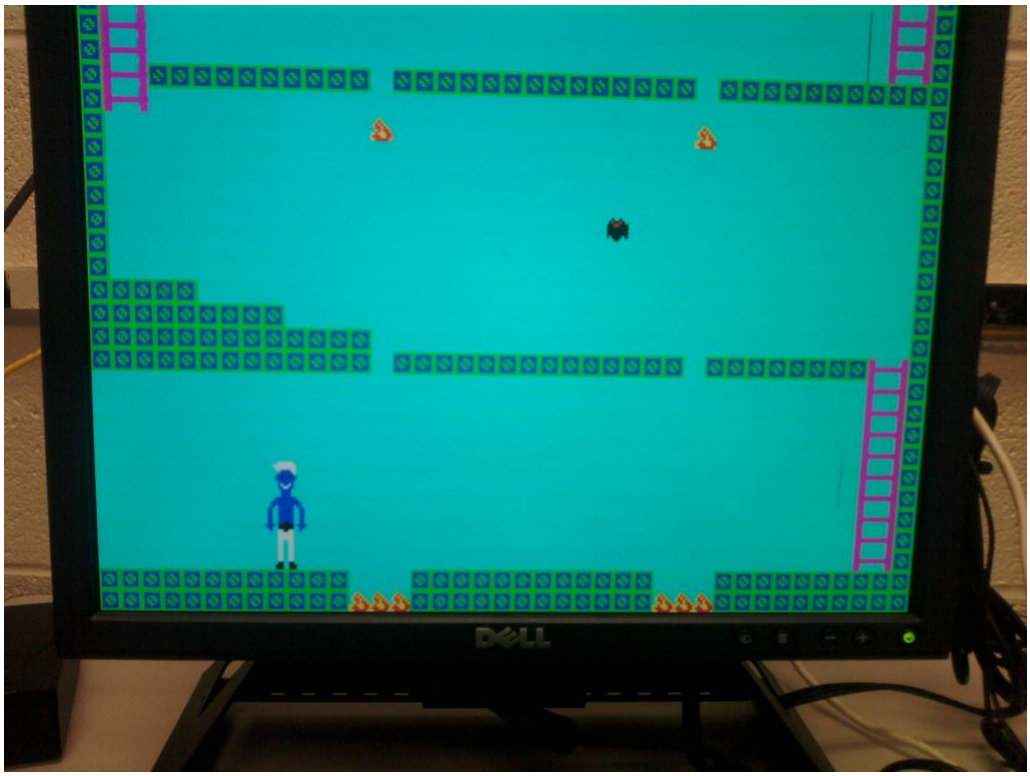

Figure 14: Screen 1 with Fires and Bat Enemy

Figure 15: Indication of Death when Touching Fire or Enemy

Figure 16: Screen 2 with Fires and Skull Enemy

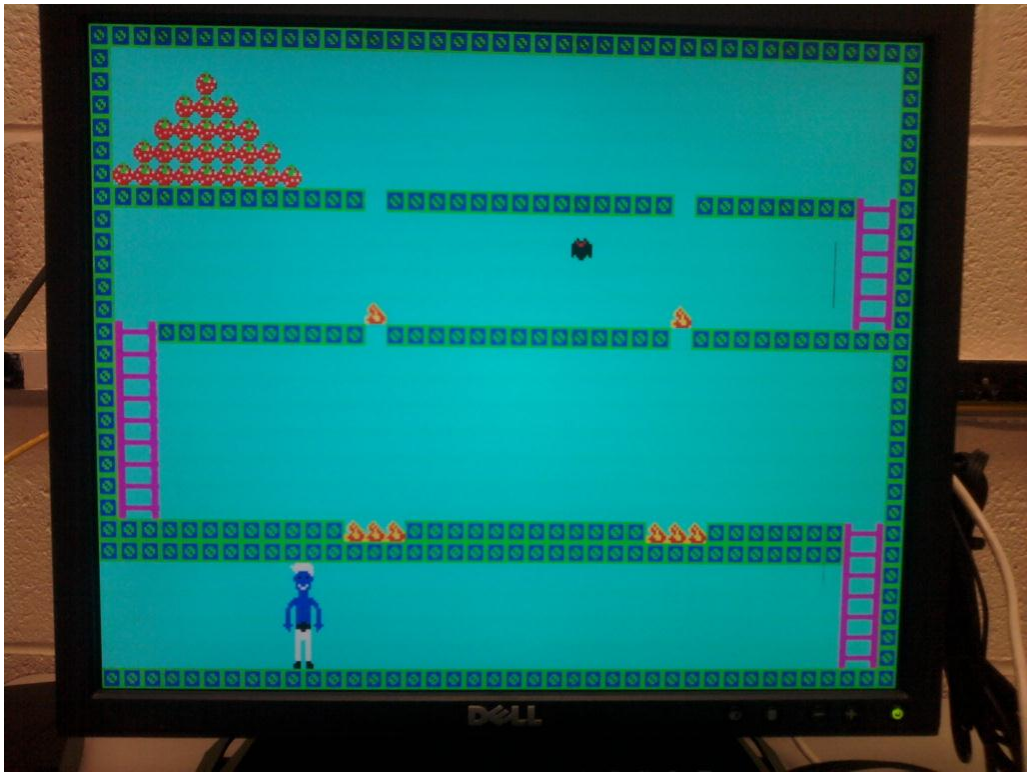

Figure 17: Screen 6 with Fires, Bat Enemy and Prize

Figure 18: Screen 7 with Winner Indication

7. Conclusion

1. Getting the knowledge about the SoC and embedded system with Xilinx EDK.
2. Getting the knowledge about NES controller and how to connect it with Xilinx FPGA.
3. Being able to manipulate LEDs using VHDL and UCF file.
4. Being able to generate two output clocks from the FPGA to NES controller using frequency divider (25MHz).
5. Being able to manipulate GPIO by using the three levels of interfacing, No Device Driver, Low Level Driver, and High Level Driver.
6. Associating State Machines with the generated clocks in real time to synchronizing the incoming data from NES and converting them as outputs using the LEDs of FPGA.
7. Being able to use FOR loops as a timing delay counter.
8. Being able to use the Timer IP core as a timing delay counter in the program with any of the three levels of GPIO interfacing.
9. Being able to interface the Timer IP core with the interrupt and re-initiate all the required procedures inside the ISR.
10. Getting the knowledge about mapping from the tile-map coordinates to LCD screen real coordinates.
11. Getting the knowledge about how to apply a mathematical equation and its effects on Xilinx hardware using normal divisions, bitwise shifting and direct bits mapping.
12. Develop three derived methods from the same equation in order to reduce the hardware, LUT and slices in FPGA.
13. The overall design is based on the combinational logic.
14. Getting the knowledge about mapping from the tile-map indices to the desirable contents of the sprites map for characters and backgrounds.
15. All the maps are becoming LUT and not memory spaces (RAM)
16. Being able to increase the VGA resolution from 160x120 to 320x240.
17. Developing new equations for the tiles and sprites indices.
18. Being able to choose the sufficient memory size of SRAM by using the global variables in C code.

19. The hardware of the VGA output is based on the combinational logic and not on processes.
20. Using the linked list of elements for one screen, and differentiating them from other screens.
21. Initializing the tile maps of background and foreground on request.
22. Using a defined FOR loop as a timer.
23. Separating the drawing and clearing of the screens and player as two functions, one for the foreground tile map and another for the background.
24. Implementing the hardware part of the project with VHDL and the software with C code.
25. Using the same structure for the screen's elements, the character, the character's weapon, the flames and the enemies.
26. Using the controller's arrows to change the directions of the player, A to fire, B button to jump, SELECT button to reload the same level and START button to reload the first level.
27. Using MACRO as variable replacement and name representation for colors, objects, enemy types.
28. Using MACRO as a function replacement for the repeated expressions along the game program.
29. Terminating the killed enemy from the linked list, in order to decrease the delay by drawing the invisible killed enemy on the foreground tile map each time.
30. In my consideration, this project added to me more experiences in VHDL and C programming. Plus, the ability to design and implement different tasks by using different techniques and abstraction approach.