Electrical and Computer Engineering

Parks College of Saint Louis University

ECE 4225: Hardware/Software Co-Design

Block Chaser

(Embedded System Video Game Design)

December 16th, 2021

By Janssen Gamilla : Computer Engineering Student

Abstract

Embedded systems are typically thought of as computer hardware systems embedded with software to perform a specific task. In this paper, a simple video game was designed to showcase an embedded hardware/software system. The Xilinix Basys 3 FPGA Development board was employed in this project to design the hardware component of this project, while the MicroBlaze softcore processor along with a C program served as the software component. The design of the game will be introduced, as well as the design and its implementation. An overview of historical solutions to this kind of project will be discussed. Then the design of the completed project will be analyzed.

Introduction

The goal of this project is to design an embedded system that utilizes the communication between hardware and software. A simple video game design was to be implemented with a simple objective for the player. The functions of this game were to be identified and then analyzed to determine whether they are to be implemented in hardware or software. For this game, the user has to be able to control the game, reach a goal, and have the game restart upon completion of the goal. The game's system also should be able to restart itself upon completion of the game. The user should be able to view the game via a VGA monitor. The original goal of this project was to recreate the famous "Snake" video game. After some deliberation, a simplified version of it was designed, which is called "Block Chaser". The goal is similar to snake, however the user's character does not increment in length. Rather it changes the color of the block when reaching the "target" block on the screen.

In this project, there must be communication between hardware and software components in the system. This project was constrained by the memory capacity of the Xilinx Basys 3 FPGA development board and Hardware Description Language (VHDL used). This board has many libraries and packages as well as many ports that could be interfaced to, leading to a few amount of hardware constraints. A 640x480 @60Hz VGA monitor was what the design for the game's graphics were designed towards. The project's development was also constrained by the timeframe of the Fall 2021 academic semester.

In order to tackle this project, the game was first broken down into a functional decomposition, typically verb-noun pairs. These functions serve as small blocks that build up the overall design of the project. Once these are all defined, they are analyzed to determine whether they belong in the hardware or software portion of the project. To allow the game to be controlled by the user, push buttons serve as the user input, handled through hardware. Those inputs then get processed by software to update another hardware module that updates the graphics on the VGA monitor.

To accomplish this project, the four push buttons on the Basys 3 board were used as the user inputs. In hardware, the four buttons are used to drive a value that is wired to the MicroBlaze softcore processor. It has a C program that reads that value and determines which directional button was pressed. Three VGA frame buffers were implemented for each RGB color, whose registers get written into by software based on what needs to be drawn on the screen. They are wired to a VGA controller in hardware that implements the vertical and horizontal synchronizations of a VGA monitor and processes the RGB values to output onto a 640x480 @60Hz VGA monitor. For the actual game design, the user's block remains still at the start of the game and upon a directional button press, the game starts and the user's block will start to move in that direction until a different button is pressed. The goal is to reach the white target blocks on the screen to increase the score and reach the score required to win the game. Once that score is reached, the game will reset and the user's block will return to the center of the screen. An example of what the game looks like can be seen in the appendix of this report.

This report will dive into some of the historical approaches to embedded system design with hardware and software, as well as how it's developed. An in depth description of the design, especially the method in which graphics are displayed onto the screen, are examined. The work performed throughout the development of this project is discussed to drive the design process. This report concludes with discussion regarding lessons learned and issues encountered from this project, as well as improvements that could be made in the future for the project.

Background

As the "Snake" game was the basis for the design of this project, its past implementation will be discussed as well as approaches to video game design in embedded systems. Video games have been around for many decades now. Due to the technology available at the start, developers had to use clever designs to develop those games. The definition of a video game in

this report is something that a player can interact with, which manipulates computer-generated images on a screen.

The variety of “Snake” games that all share the same type of gameplay can originate from the arcade game “Blockade”, published by Gremlin in 1976. This game was housed in a video game cabinet, with an Intel 8080 processor, a 256x224 pixel @60Hz screen that was capable of displaying two colors, and 4 buttons. The intel 8080 is compatible with transistor-transistor logic (TTL) chips, which allows arcade games to combine more and more of these chips depending on how complex the game is [1]. With how customizable the circuit boards of these games are, there was not a consistent high level programming language that could be used. Instead, game designers had to utilize assembly language and build the functions of their games around sets of instructions [2]. For these arcade game screens, their graphics were typically designed around Raster Graphics and displayed on a CRT display. Raster graphics are simply bitmaps for a certain image or sprite stored in a file that would be scanned by something like the CRT screen and displayed. The data of these graphical patterns are stored in an area of memory called a frame buffer, and then those values are retrieved from a refresh buffer and displayed onto the screen one row at a time [3]. Put simply, assembly instructions (software) would be used to select certain graphics that are loaded to be displayed through hardware. As complexity for video games greatly increased and the transistor count per chip advanced computing technology, new techniques were created to design games. Similar to most embedded systems with I/O, video game platforms like the Atari utilized Interrupt Service Routines (ISR) to update graphics or handle user input. When these interrupt routines are called, the respective functions for the peripheral are executed. With this method, something like joystick movement is allowed to occur at any time [4].

The next step in video games following the classic cabinet style arcade machines would be video game consoles. The earliest consoles still utilized the same methods as the arcade machines, employing assembly and similar graphics systems. But it was with the introduction of ROMs that could be inserted into these consoles and run a game, then removed and replaced with another game that could also be run on the console, that was truly the advancement here. Nowadays, video game consoles have reached such great computing power that they contain hardware comparable to high end PCs. Discs are starting to become obsolete as video games require all of their files to be stored on the console’s memory. There are many standard interfaces now like USB for controllers, and HDMI for video output that are common across most modern consoles that don’t require any new special design on the hardware or software side.

Design/Work Performed

Design

The major components to this project were handling user input, reading that input, updating graphics based on that input, then displaying those graphics on the monitor. The first step was designing a hardware module that processed the user input. A simple VHDL file (top.vhd) was created with 5 inputs (clock and 4 buttons) and a single output. This module was designed to output a 4-bit vector that will be “0000” if no buttons are pressed. Depending on the button pressed, one of the bit positions will be written as 1. For example, if the up button is pressed, the output vector will have a value of “0001”. This module is then mapped to an AXI GPIO block that is wired to the processor. Afterwards in software, it was ensured that the vector from the user input module was read and reflected which button was pressed.

Next, the VGA Controller and frame buffer were designed through hardware. The VGA controller was designed with counters to achieve the correct Hsync and Vsync timings, as well as write the RGB values out of the Basys 3’s VGA port (see vga_signals.vhd)[5] [6]. This was achieved through signal assignments. The frame buffer design connects the software to the VGA controller. The design for this frame buffer was achieved by accounting for the screen’s resolution, and dividing it up in a way that results in a manageable amount of indices to write to. Additionally, the amount of memory on the board is not capable of writing each individual pixel on a 640x480 screen (307,200 total pixels). The screen was divided into 16 x 16 pixel blocks, giving us a 40x30 grid. However, this results in 1,200 total indices to write to, which is still a large amount to keep track of. So, the screen was divided again into 4-block wide chunks, seen in Figure 1. This way, there are only 300 registers in each frame buffer that need to be written to. Values are written into these frame buffers through software.

Figure 1.) Values driven into each frame buffer to color a single block in a chunk.

When implementing the project's design in software, the first step was figuring out how to write to the registers of each frame buffer and actually getting color to appear on the screen. Pointers to each of the frame buffers were used. Then determining the correct way to utilize x and y positions on the grid to write to a certain register (location on the screen) was written. Due to the way the screen was divided, Equation 1 was used to map x and y values in the software to map to the registers.

Register location = y*10 + x

Equation 1.) Formula used to write to a certain register based on x and y values.

Next, the movement of blocks was implemented in software. Each block (user and target) have their own x and y coordinates, but the user block also has an x direction and y direction that is kept track of to continue the path of the block after the user makes it go a certain direction. A function is continuously called to update the user block's position, determine whether it changed direction, and then draw its graphics on the screen based on its new position. Since the target block just needs to be static throughout the game, it only needed its own x and y coordinates. Due to the way the screen was divided into those 4-block wide chunks, there was a technique employed to make it look like the block was moving. This was necessary since even if the x and y coordinates changed for the user block, it could technically still be in the same 4-block chunk. In order to deal with this, the x value was masked with 0x03 so that we accessed the correct block in the chunk with the help of switch-case statements. In each of those cases, the frame buffers are written an rgb value. These rgb values are masked with a 32 bit number to ensure that the correct position in the chunk is written to. It also takes in parameters for each color so that the user block can change color. This can be seen in the draw_pixel() method, found in the appendix.

The target block had a similar way of getting drawn to the screen. To determine if the target block and user block collided on the screen, their respective x and y values were compared. If they were equal, the color of the user block would change and the target block would be assigned a new random x and y value, then get drawn onto the screen. This would also update the score of the game.

The color changing functionality of the game was also implemented in hardware. Since there was no 16 bit data type available, the uint32_t (32 bit values) data type was used to write to the frame buffer registers. Since we only needed the least significant 16 bits to write to each register, the most significant 16 bits were used to keep track of the user block's current color.

Those bits would undergo bit masking and an equality operator to determine the current color of the block and then assign the next color to the block.

Testing

After each design of the project, they were tested prior to moving on to the next design portion of the project. After the push-button handler was designed, it was ran through a behavioral simulation, where its clock value was driven and each button was driven high at a time. The output vector was used to test whether the correct bits were going high. Once the correct code in software was implemented to read in the output vector, simple if statements were used to output the cases if each of the buttons were pressed. Print statements were used to indicate each button press on the console. Another behavioral simulation was done on the vga signal module to ensure that the Hsync and Vsync signals were running at the correct frequencies. This was simply done by simulating the clock in the module.

When testing the software implementations for the project, the debugger in Xilinx's SDK proved to be very useful. Testing each new function added to the program was typically done using print statements in the method that output x and y positions, conditionals, or even specific lines in code to ensure that a specific function was actually getting run. Outside of the console, the monitor was observed to test whether a function written was correct. This was mostly done when testing that the user block was moving in the right direction on button presses, if the user block was changing the correct color, or if the game was resetting correctly upon a game win.

Conclusions and Next Steps

In this project, the goal was to design an embedded system that combined hardware and software design. This was to be done by implementing a simple video game that handled user input and outputs graphics onto a VGA screen. Upon completion of the project, it can be concluded that these goals were accomplished, with some slight compromises in the design to do so.

There were a multitude of things that I learned throughout the development of this project. The major lesson that I learned was the amount of attention to detail required when dealing with a hardware-software design. Making sure to test after completing a design was also a lesson that I had to learn. Figuring out the correct timing for the Hsync and Vsync for the VGA monitor was crucial in getting the board to output graphics. After implementing the modules for the graphics, there wasn't any output on the monitor. It wasn't until the VGA signal module was

inspected that it was discovered that the counters were not driving the sync values to the proper frequencies, causing the problem. If the module had been tested before being added to the design, then this problem could have been avoided. Another lesson I learned was how much creativity it takes for a project like this to meet the constraints of a system. Dividing the 640x480 to eventually get 300 register to write to for each frame buffer was something that also required a lot of attention to detail. Being able to come up with a design like this to meet the constraints of memory on the board is something important for designers of systems like this.

Some issues that were encountered other than the lack of monitor output resided with the VGA controller. It was observed that portions of the screen were being drawn twice on the screen, due to a bug that was not able to be found in time. For example, a block drawn in the top left corner of the screen was also drawn at a spot in the bottom right corner of the screen. It was confirmed that it was being drawn twice when testing it with a moving graphic, where they matched each other's movement. This issue was temporarily resolved by restricting the bounds of the game, to the areas of the screen that didn't result in this duplication.

Another problem that was faced, but could be improved upon is the way the graphics are updated on the screen. In a moving block , the way the previous frame is removed from the screen is done by "wiping" the entire screen. This wiping was basically drawing the whole screen black right before drawing the new image. This updating method causes some observable blinking in the game while in play. This could be improved upon by handling this in hardware by reading register values in the frame buffers, so the whole screen wouldn't have to be wiped.

Due to the complexity in the way graphics are drawn, the game had to be very simple in design. Originally, this project was supposed to implement the snake game. However, there were some components of the game that could have been assigned to hardware (like reading register values to update and keep track of graphics) that led to me opting to design a basic game that only dealt with single blocks on the screen. If I were to implement the snake game, I would have utilized a linked list data structure to keep track of the head of the snake, as well as the rest of its body. Then implement more of the graphics portion of the design in hardware. For the Block Chaser game itself, some things I would implement in the future is a way to actually lose the game, by going out of bounds or colliding with obstacles generated. I would also implement a way to actually reset the game using the center button on the Basys 3 board in the case that the game ever gets stuck.

Overall, the goals of this project were accomplished and I was pleased with the results. I was able to create a game that had a goal and way to "win". Portions of the design were

assigned to hardware and software. I've grown an appreciation for the effort and meticulousness required for embedded systems like this. Looking back on the development process of this project, there are definitely aspects that I would have done differently, especially with the graphics.

Appendix

top.vhd

```
entity top is
    Port ( clk : in STD_LOGIC;
           up : in STD_LOGIC;
           down : in STD_LOGIC;
           left : in STD_LOGIC;
           right : in STD_LOGIC;
           btn_output: out STD_LOGIC_VECTOR(3 downto 0));
end top;

architecture Behavioral of top is
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if up = '1' then
                btn_output(0) <= '1';
            elsif down ='1' then
                btn_output(1) <= '1';
            elsif left ='1' then
                btn_output(2) <= '1';
            elsif right = '1' then
                btn_output(3) <= '1';
            else
                btn_output <= "0000";
            end if;
        end if;
    end process;
end Behavioral;
```

vga_signals.vhd

```
entity vga_signals is
    Port ( clk : in STD_LOGIC;
           r : out STD_LOGIC_VECTOR (3 downto 0);
           g : out STD_LOGIC_VECTOR (3 downto 0);
           b : out STD_LOGIC_VECTOR (3 downto 0);
           hSync : out STD_LOGIC;
           vSync : out STD_LOGIC;
           red : in STD_LOGIC_VECTOR(3 downto 0);
           green : in STD_LOGIC_VECTOR(3 downto 0);
           blue : in STD_LOGIC_VECTOR(3 downto 0);
           row : out STD_LOGIC_VECTOR(5 downto 0);
           col : out STD_LOGIC_VECTOR(5 downto 0));
```

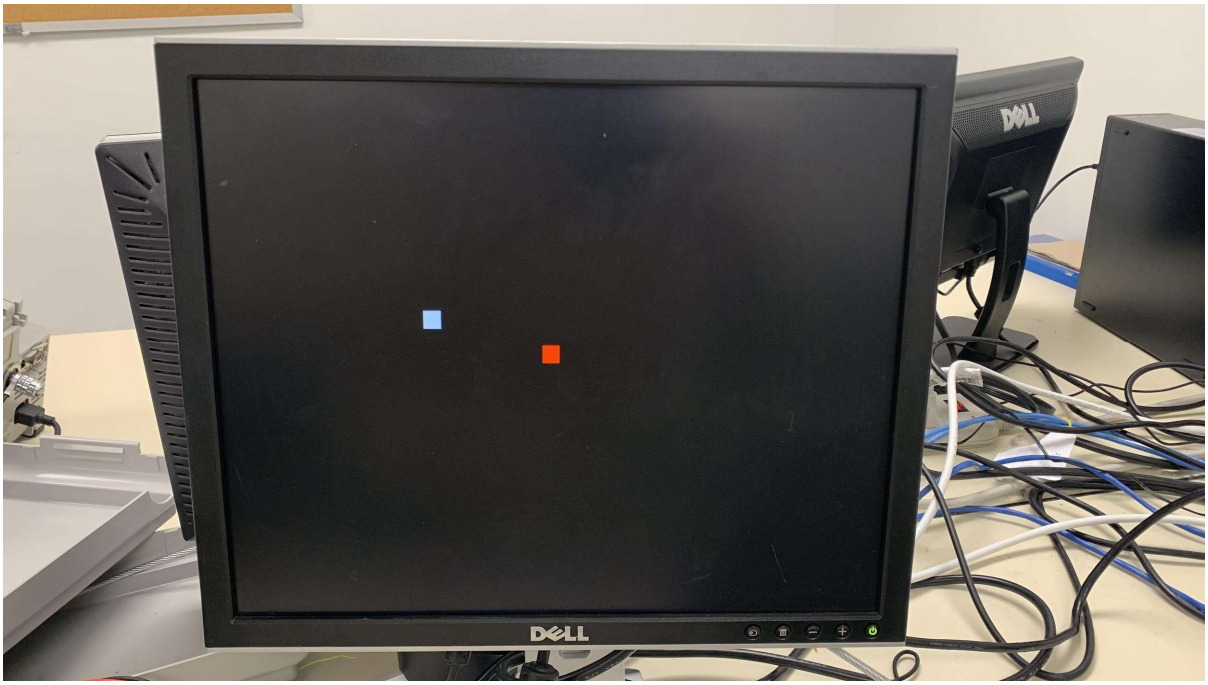
```
end vga_signals;

architecture Behavioral of vga_signals is
signal hValid, vValid : std_logic;
signal clk_25MHz : std_logic := '0';
begin

process(clk)
variable cnt : unsigned(1 downto 0) := "00";
begin
    if rising_edge(clk) then
        cnt := cnt + 1;
    end if;
    clk_25MHz <= not cnt(1);
end process;

process(clk_25MHz)
variable hCnt, vCnt : integer := 0;
variable colT, rowT : std_logic_vector(9 downto 0);
begin
    if rising_edge(clk_25Mhz) then
        hCnt := hCnt + 1;
        if hCnt < 640 then
            hValid <= '1';
        else
            hValid <= '0';
        end if;
        if hCnt > 655 and hCnt < 751 then
            hSync <= '0';
        else
            hSync <= '1';
        end if;
        colT := std_logic_vector(to_unsigned(hCnt, 10));
        if hCnt = 800 then
            hCnt := 0;
        end if;
        if hCnt = 780 then
            vCnt := vCnt + 1;
        end if;
        if vCnt < 480 then
end Behavioral;
```

draw_pixel() method

```
void draw_pixel(int x, int y,int r, int g, int b) {
      int x1, x2;
            clearScreen();
            x1 = x & 0x03; //mask with 0000 0011 to get 2 LSBs
            x2 = x >> 2;
            switch(x1) {
            case 0:
                  fb_Red[y*10+x2] = r & 0x0000000F;
                  fb_Green[y*10+x2] = g & 0x0000000F;
                  fb_Blue[y*10+x2] = b & 0x0000000F;
                  break;
            case 1:
```

```
                fb_Red[y*10+x2] = r &  0x000000F0;
                fb_Green[y*10+x2] = g & 0x000000F0;
                fb_Blue[y*10+x2] = b & 0x000000F0;
                break;
        case 2:
                fb_Red[y*10+x2] = r &  0x00000F00;
                fb_Green[y*10+x2] = g & 0x00000F00;
                fb_Blue[y*10+x2] = b & 0x00000F00;
                break;
        case 3:
                fb_Red[y*10+x2] = r &  0x0000F000;
                fb_Green[y*10+x2] = g & 0x0000F000;
                fb_Blue[y*10+x2] = b & 0x0000F000;
                break;
        default:
                break;
        }
}
```

Block Chaser Screen



Bibliography

[1] “Arcade video game,” Wikipedia, Dec. 11, 2021. https://en.wikipedia.org/wiki/Arcade_video_game#Technology (accessed Dec. 16, 2021).

[2] “Atari Graphics and Arcade Game Design-Chapter 4,” www.atariarchives.org. https://www.atariarchives.org/agagd/chapter4.php (accessed Dec. 17, 2021).

[3] Wikipedia Contributors, “Raster graphics,” Wikipedia, Feb. 06, 2020. https://en.wikipedia.org/wiki/Raster_graphics (accessed Dec. 17, 2021).

[4] “Systems Guide: Interrupts,” www.atarimagazines.com. https://www.atarimagazines.com/vbook/interrupts.html (accessed Dec. 17, 2021).

[5] Digilent, “Basys 3™ FPGA Board Reference Manual”. Distributed by Digilent. (accessed Dec. 13, 2021) https://digilent.com/reference/_media/basys3:basys3_rm.pdf

[6] tinyvga.com, “VGA Signal 640 x 480 @ 60 Hz Industry standard timing”. Distributed by SECONS Ltd. (accessed Dec.13, 2021)