**Creating the Plava Gaming Platform Using the Spartan-3E**

|  |  |  |  |
| --- | --- | --- | --- |
| Ashton Snelgrove  University of Utah  1518 S. 1000 E.  Salt Lake City, UT 84105  801-819-6344  snelgrov@eng.utah.edu | Jacob Sanders  University of Utah  6564 S. 2520 E.  Cottonwood Heights, UT 84121  602-486-5777  jacob.sanders@utah.edu | Matthew Steadman  University of Utah  4318 Arden Ct  Taylorsville, UT  84123  801-783-0989  steadman@eng.utah.edu | William Graham  University of Utah  737 E 700 S  Salt Lake City, UT 84111  801-620-0266  U0527893@utah.edu |
|  |  |  |  |

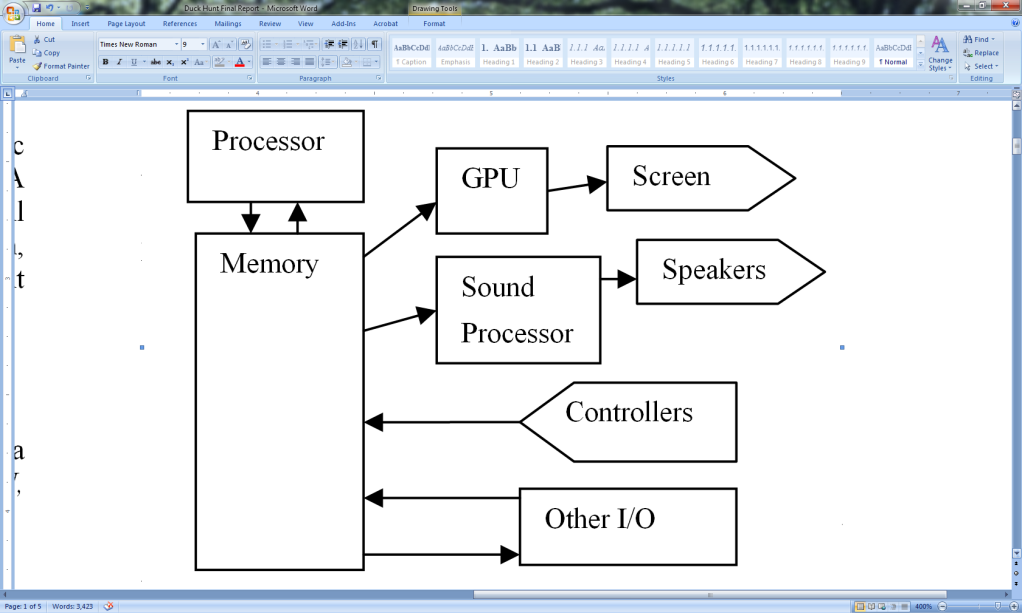
**ABSTRACT**

Figure 1. High Level Block Diagram

In this paper, we will describe the development of a generic console game platform, using the Xilinx Spartan-3E FPGA board, I/O devices, and other additional circuitry. We will then demonstrate a sample application of the platform, implementing a clone of the Nintendo Entertainment System game Duck Hunt.

**Categories and Subject Descriptors**

B.6.3 [**Verilog]**: Design and implementation through a Xilinx FPGA of a gaming platform – *Processor, GPU, digital sound reproduction*

**General Terms**

Design, Languages

**Keywords**

FPGA, Processor Design, 24-bit color, GPU, Assembly language, Light gun

# INTRODUCTION

The intent of this project was to create Plava, a 16-bit generic gaming platform that can be used to run and play video games. Plava is the Croatian word for the color blue, and refers to the development team, which through the development cycle was referred to as Team Blue.

Plava is capable of generating 24-bit color images, can play up to ten sounds at the same time, and is compatible with the Super Nintendo Entertainment System game pad controller and the Nintendo Entertainment System Zapper light gun. Use of the light gun requires an external VGA to TV converter box that is capable of displaying the images generated by the GPU on a standard tube television. A block diagram describing the Plava system can be found in figure 1.

Permission is granted to make digital or hard copies of this work for any educational or personal purposes provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice, all author names, and full citation on the first page. Written permission to otherwise copy or distribute this work must be obtained from any one of the authors, according to the contact information given.

# THE PLAVA PROCESSOR

Plava uses a 16-bit processor built around a modified version of the CR-16 instruction set. The processor contains sixteen 16-bit registers and an additional 5-bit program state register (PSR).

## Instruction Set

The Plava instruction set is a binary (two argument) instruction set. In most instructions the first input is the destination of the data following the execution of the instruction. The second instruction is an argument to the operation. There are two types of most instructions, a register based operation where two registers are given and the result stored in the first, and an immediate type instruction where a scalar value can be processed in place of the secondary register.

There are a total of twelve basic arithmetic instructions, including addition, subtraction, and multiplication. Six of these are addition instructions. These include both register and immediate type instructions for standard addition, addition with carry, and addition that does not affect the PSR. There are 4 subtraction operations, again both register and immediate versions of a standard subtraction, and subtraction with carry. The other two basic arithmetic operations are a register and immediate type of multiplication.

There are also register and immediate based bitwise and, or, and exclusive or operations comprising a total of six bitwise operations. Shifting operations are also implemented for register and immediate arguments, for both arithmetic and logical shifting. The value in the first register is shifted by the number stored in the second register or an immediate value. Left shifts are indicated by a positive shift amount, and right shifts by a negative shift amount.

For getting and setting data values there are several instructions. The move command allows the programmer to move a value stored in a register or from an immediate value into another register. Since there is a limitation in the number of bits available for an immediate value there is an additional load upper immediate instruction which loads the immediate value into the upper 8 bits of the register rather than the lower 8 bits. The move immediate command sign extends the immediate value, and as such overwrites the upper 8 bits. To move a full 16-bit immediate value into a register the move command and then the load command are required, using the move immediate first, which only overwrites the upper 8-bits, leaving the lower 8-bits alone. There is also a load and store command which enables you to load or store values from the system’s memory.

Two additional instructions were added to the instruction set, test and not. Test performs a bitwise and for the purpose of setting the PSR flags, but does not write the result back to register. Not implements a unary bitwise not on the contents of a register.

The PSR was used to store information on the result of certain instructions. The five flag bits store whether the previous instruction resulted in a carry or borrow, if the value of the second operand was less than the first, if arithmetic overflow occurred, if the result of the operation was zero, and if the result was negative. Any branch or jump instructions following the instruction that set the PSR will use the PSR flags to decide whether or not the jump or branch occurs.

The branch and jump commands enable choices to be made within the execution of programs and are the only conditional instructions in the instruction set. There are a total of sixteen mnemonics used with branches and jumps that signify a unique condition determined by the status of the PSR, set by previous instruction. These mnemonics as well as their meanings are shown in Table 1.

## Architecture

The Plava processor architecture consists of a controller that interprets and fetches instructions, and an arithmetic logic unit (ALU) which processes the instructions. This two part design enables all instructions except the memory load instruction to be executed in two clock cycles, using a two stage fetch and calculate process. Storage of the result of the calculate step are pipelined with the fetch stage.

The first stage is responsible for loading an instruction and decoding the operation. During the first clock cycle the instruction is fetched from current program counter location in memory and sets all appropriate multiplexer values in the rest of the processor.

Table 1. Conditional Jump and Branch information

The second stage is responsible for calculation required for the operation. In the second clock cycle values are loaded from registers and sent into the ALU to be processed. The results from the ALU are then written back during the fetch phase of the next instruction. This partial pipelining of the write back and instruction fetch stages pipelined enabling us to effectively execute 25 million instructions per second.

The only exception to the two stage cycle is the load instruction, which requires three clock cycles. This is due to the address being loaded from register, then the data at that address fetched from memory, then the data finally being written back during the following fetch phase.

### Arithmetic Logic Unit

The ALU-chip is the most complex piece of the processor and as such will be addressed individually. The ALU-chip is the place where all calculation is actually performed. It expects as inputs the values to be used in the calculation, an operation code, a function code, and a condition code; the latter three values are determined by the controller, which gets them from the instruction. For jumps and branches, the current values of the PSR is read and used in conjunction with the condition code to calculate the control flow. The output of the ALU is the result of the calculation. Certain operations will also modify the flags in the PSR register.

# MEMORY MAP

We laid out the memory map for our computer using the provided block rams on the Spartan 3e board. The deciding factor for how of the block ram was allocated to each purpose was partly based on usage and importance, at least in the beginning. Since the biggest memory hogs in our hardware were going to be the processor and the fancy graphic processor these components were assigned address space first. The processor got the first 8 kilo-words (Kw) of space on the block rams, followed by the next 10 Kw being given to the GPU to manage sprite and palette data. The GPU also used 1 word for status of the screen. Once the major processing units were mapped the rest of the space was allocated for the numerous components we wanted to incorporate, such as the switches (mainly for testing), game controllers, the sound controller, and other I/O devices. An example of memory mapped locations is shown in table 2. This enabled the programmer direct access to read and write the values for any I/O device, and made interaction with every component as simple as checking and updating values in specific locations in memory.

# GRAPHICS PROCESSING UNIT

## Overview

The graphics processing unit of the Plava platform is a bitmapped sprite based system. The system is capable of VGA output at True Color (24-bit) levels, and provides to the programmer an easy to use memory mapped interface.

The system is based loosely on the graphics implementation of the Super Nintendo Entertainment System (SNES) [2]. The basic concepts of sprite object meta-data, bitmapped sprite tables, and a palette lookup table were used. These features were expanded to use the capabilities of the Spartan-3e FPGA.

## Sprite Object Table

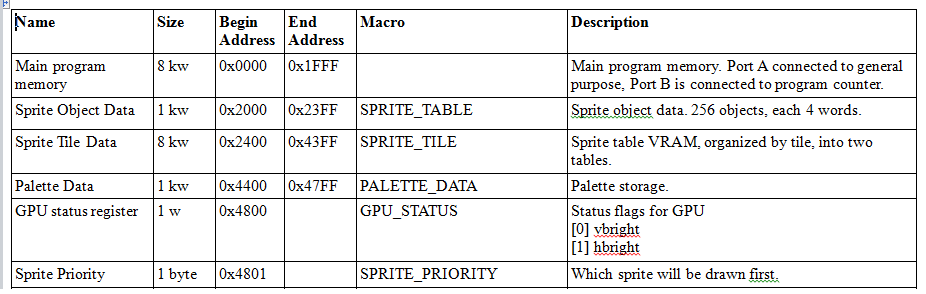
The sprite object table consists of 256 objects, each of which contains information on the screen coordinate of each object, depth priority, horizontal and vertical flipping, sprite table selection, size, and palette selection. The programmer is able to manipulate these objects through an array of structures located in memory. This allows the programmer to easily perform basic sprite manipulations, such as translation, by reading and writing directly to memory.

## Bitmap Tile Table

The system contains two 128x128 pixel sprite tile tables. 16-color indexed bitmaps are split into 8x8 pixel tiles and placed on the table. Each table therefore 16x16 tiles. Each sprite object contains a tile coordinate, which is the X and Y coordinate of the upper left tile of the sprite. By using the horizontal and vertical size from the sprite object, a section of the sprite table is isolated, which forms the complete sprite.

## Rendering

The VGA standard generates 640x480 pixels at a refresh rate of 60 Hz. This provides significant time per screen refresh to place the sprites onto the screen. However, because of memory limitations, only the next scan line and the current scan line can be stored in a frame buffer. A block RAM is dedicated to the front and back scan line buffer.

At the beginning of each scan line, the front and back buffer are swapped. The back buffer then begins calculating the contents of the next scan line. Initially, the system reads from a register, called sprite priority, which determines the rendering order of sprites. When drawing a scan line, the first sprite drawn will be the sprite indicated by this register. After this sprite is drawn, the GPU then draws sprites in numerical order. For example, if the priority is set to 10, then the 10th sprite will be drawn, then the 11th, and so on. This allows the programmer to prioritize the drawing of specific sprites.

As each sprite object is read from the sprite object table, the system culls sprites without the active flag and sprites which do not intersect the current scan line. If the sprite is active and does intersect, the sprite data is then passed on to the line buffer to be drawn. The line buffer calculates the address of a 1x8 pixel horizontal slice from the tile table. This is then copied into the back buffer at the appropriate horizontal offset, along with depth priority information and the palette from the sprite object.

Table 2. Sample of Memory Mapped Locations

Rendering follows consistent rules dealing with overlapping sprites. A pixel is considered transparent if the index of the pixel is 0. If a pixel has not been updated this line, then the pixel from the sprite being rendered is automatically written to the line. If the pixel has already been updated, then the new pixel is checked for transparency, and ignored if transparent. If non-transparent, then the depth priorities of the two pixels are compared. If the new pixel has a depth priority greater than the current pixel, then the new pixel is written. Otherwise if the new depth priority is less than or equal to the current pixel, the pixel remains the same. This means that the sprites on the same depth will be prioritized on a first rendered basis.

## Utilities and Testing

To facilitate the creation of tile data, several scripts were written to convert PNG image files into the ASCII format used by the synthesis engine for describing memory initialization. Scripts were also written to convert memory format back into images. This allowed the writing of a test bench which will run the current system for a single VGA refresh cycle and write the color value of each pixel to a file. Once converted, this allows the programmer to see the output of the graphics system without loading the system on the FPGA.

## Output Generation

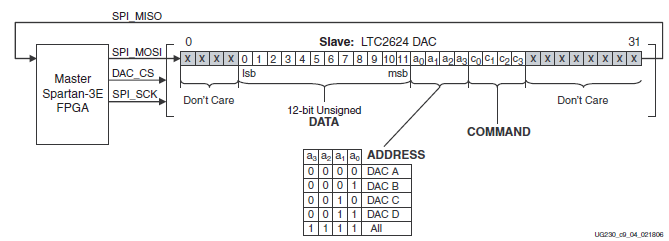
As the scan line moves across the screen horizontally, the front buffer is read pixel by pixel. If the pixel is non-transparent, then the pixel is used, otherwise a separately generated background index is used. The final index is then passed into the palette lookup table, which in turn outputs the final color to the display.

As each bitmap in the sprite table is indexed, a palette lookup table is used to convert a palette index number into a 24-bit RGB color. The system uses 16 colors per palette, with 32 distinct palettes available. The palette is selected as part of the sprite object, and the palette index is selected by each pixel in the sprite bitmap.

As a final step, the brightness register is used to do fade effects. Setting this register to 255 will allow full color output, while setting this register to 0 will completely blank the screen. Setting the register in between will provide a scaling factor on each of the channels.

The on-board VGA connector of the Spartan-3E starter development kit only supports 1 bit per color channel, for 3-bit color. The Plava System uses a off-board video digital to analog converter (DAC) to allow 8-bits per color channel, for 24-bit color. The DAC selected is the Texas Instruments THS8134B [5]. This chip is designed specifically for the use of video systems, including VGA; it features three matched DACs and a parallel digital interface. The circuit was built using the datasheet and a reference design by Altium [1].

# SOUND SYSTEM

We debated a few different ways to implement sound, but with the wide assortment of different hardware available on the Spartan 3e board we decided to take advantage of the components that were already on hand. To accomplish this we broke down the sound into four separate modules one to generate a sample frequency to output our sounds, another module to act as the interface between the flash and the programmer, a mixer module to combine all sounds currently playing into one sample, and one to serially output sound data to the digital analog converter (DAC). The onboard hardware allowed us to play up to 100 sounds concurrently although we elected for a more ear friendly 10 sounds. There is a 12-bit DAC onboard which was the obvious choice to output the sound data to a speaker. To store the sounds we decided to use the built-in Intel Strataflash since it provided 16 MB of storage space with a relatively fast read speed. We did scale the precision of our sounds down to 8-bit due to the fact that we could only load either 8 or 16 bits from flash memory at a time and using 8 bit was easier than having to deal with truncating the top 4 bits of a 16 bit value in order to interact with the 12-bit DAC.

In order to play a sound the programmer uses the memory map to send in an address to any number of the various sounds stored in flash and sets the amplitude for the sound. From there the sound controller deals with the details of when and where to read from memory, fetching sound data from the Strataflash for each selected sound and outputting the data sample by sample into the mixer module. Once in the mixer, the sound data from all the current sounds is combined and adjusted for amplitude to make a single 8 bit sample which then gets sent to the DAC controller. Finally the sound data is arranged in the specific format outlined figure 2 and is shifted out to the onboard DAC, which then outputs to a speaker.

Figure 2. Data Format for Sound

The particular DAC on this board used a Serial Peripheral Interface (SPI). To communicate with the DAC it required a specific bit sequence as indicated from the above figure from 0 to 31 there are 4 don’t care bits, then 12 bits of data, 4 bit address to specify 1or all of the 4 built in DAC modules, followed by a 4 bit command code (typically “0011” to write to DAC), and ending with 8 don’t cares. To implement this interface we designed a DAC controller module that read in the 12 bits of data describing the sample of all currently playing sounds and shifted the data out bit by bit to the DAC while simultaneously shifting the new 32 bit sequence in.

We were originally going to go with the SPI Serial Flash that is built into the Spartan 3e board, but after implementing the SPI module we decided the SPI Flash was going to be too slow and complicated to deal with to make it worth the effort. It took 200 clock cycles to read out each sound and the SPI wire on the board was shared between multiple hardware devices which would have just further complicated our design.

# USER INTERACTION

User input devices implemented in our project include: Super Nintendo Entertainment System (SNES) controller, and a Nintendo Entertainment System (NES) light gun. There are also a number of on board LED’s and switches that the programmer can utilize.

## Light Gun

The NES light gun is connected to the system using four wires: 5 volts, light sensor output, trigger output, and ground. When the trigger is pulled, a 50 ms active low signal is sent out of the gun. The signal from the light sensor is a 5 ms active high pulse generated when light is detected from the CRT. The light gun interface sets a shot register high when the trigger input is high, and sets a hit register high when the trigger and sensor inputs are high. The shot and hit registers are concatenated and mapped to memory. The idea is for the software programmer to blank the screen on a shot signal, which will in turn set the hit signal high if the gun is pointing at the white target. Many problems were apparent when testing the sensor output signal. For the initial test, we connected the output sensor of the gun to a multi-meter and pointed it at an LCD monitor. We were unable to receive any signal from the sensor. Of no avail, we tried the same thing with a CRT monitor; the gun would not detect light. Upon realization that the multi-meter may not be sensitive enough to detect the 5ms pulses, we hooked the output sensor to an oscilloscope. This test failed as well. As stated in an MIT lab write-up, the sensor and trigger signals are not self generating, thus requiring a 10-kΩ pull-up resistor connected to the sensor and 5 volts [4]. This, however, did not work for our CRT monitor. After acquiring a VGA to composite video converter, we displayed different colors on the screen and the gun was finally detecting light.

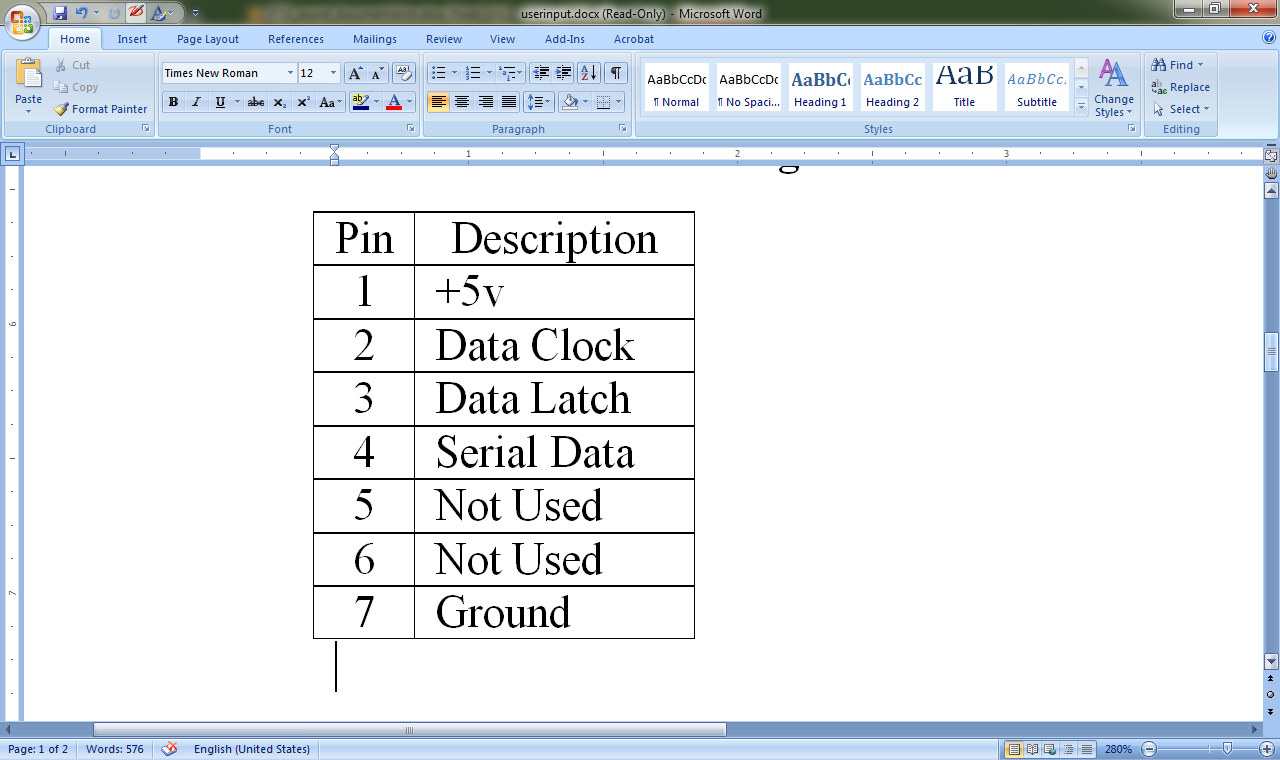


Table 3. Super Nintendo Controller Pin Out

## Super Nintendo Controller

The SNES controller is connected to the system using 5 wires. The pin-out of the controller connecter is shown in table 1. Every 60 Hz an enable signal is set and the processor sends a 12 µs pulse on pin three, instructing the internal IC’s of the controller to latch the current state of the buttons. Following the data latch, fifteen 6 µs pulses are sent to the controller on pin two. On the falling edge of each clock pulse, the button states are serially shifted out on pin four. The timing diagram shown in figure 1 shows the 12 µs long data latch pulse followed by 6 µs long pulses and the order in which the button states are shifted out. After the states of the 12 buttons have been shifted out, four additional signals are shifted out (always high). The button states are then concatenated and sent out of the controller interface into the memory map, allowing the software programmer to access the most recent state of the controller at any given time.

## Additional Features

### Direct Memory Access

The Plava system provides a feature for moving large amounts of data from the on-board StrataFlash ROM into main memory. Upon writing to the control registers, the processor execution is halted until the requested amount of memory has been copied from the destination address in ROM into the source address in main memory. This can be used to quickly swap tile or palette tables in the GPU or to utilize a bootloader for loading programs stored in the ROM. This gives our system a potential to be a truly

generic platform, as once the hardware has been synthesized, the developer simply provides a ROM image containing the complete application code and data.

### Pseudo-Random Number Generator

A pseudo-random number generator (PRNG) is implemented in hardware through the use of parallel Linear Finite Shift Registers (LFSR). LFSR are a common way of generating strings of bits with high periodicity. Each bit of the PRNG is fed by a single LFSR. A seed value can be written to the PRNG, with the least significant bit of each LFSR set to the corresponding seed bit.

### Rotary Encoder

The on-board rotary encoder controls a memory mapped counter register. Rotation to the right increments the counter, while rotation to the left decrements the counter.

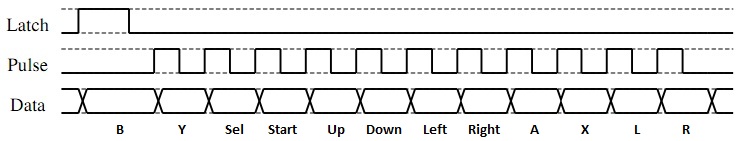
### On-board Buttons and Switches

All of the additional toggle and push button switches on the Spartan 3E starter kit are mapped into a memory mapped register.

# PROGRAMMING

## Assembler

The assembler for the Plava platform was designed to provide programmer friendly features to increase programmer productivity and ease of use. The assembler application was implemented in Python, and runs from the command line.

The assembler was written in Python 3.1, using the PLY Lex-Yacc module [3]. PLY provides a Python version of the Unix lexing and parsing tools *lex* and *yacc.* A set of lexer rules were written using regular expressions to recognize tokens such as operations, labels, registers, and immediate values. A set of parse rules were implemented as a Context Free Grammar. A macro system is implemented to take C-style *.define* statements for macro replacement.

The system works in a three pass fashion. First, the assembly code is scanned line by line, parsing macro definitions and replacing macro instances with the replacement value. Next, lexer tokenization and grammar parsing is applied. As the assembly code is tokenized, operations are counted to calculate the addresses of labels. As the grammar is applied to the tokens, a list of operation objects is created. Finally, once parsing is complete, the list of operation objects is traversed, with each operation producing a set of machine code instructions. These instructions are then written to the output file.

Figure 3. Super Nintendo Controller Timing Diagram

The Plava assemble features *.text* and *.data* segments, allowing the programmer to define variables, arrays, and strings during coding. The labels of the data segment can be used in the same manner as other labels, allowing the programmer to refer easily to the addresses of data values.

Several additional features are provided by the assembler for ease of development. Registers can be referred to by number ($0 - $15), or by the MIPS convention (e.g. $t0, $s1, $a2, $v1). The conventions of MIPS were used during development. Saved registers ($s0-$s3) , argument registers ($a0-$a2) , the frame pointer ($fp) , and the stack pointer ($sp) are callee saved. Temporary registers ($t0-$t3), return value registers ($v0-$v1), and the return address register ($ra) are caller saved. Various pseudo-operations were provided to allow easy manipulation of the stack, stack frames, and procedure calls. Additional pseudo operations are provided for loading 16-bit immediate values, no-op, and arithmetic negation.

As a consequence of having sophisticated lexer rules written, these rules were easily converted to the lexer rules used for syntax highlighting in the Gnome *gedit* text editor. This proved to be extremely valuable during development.

To aid in debugging, the assembler also provides a command line object dump mode, modeled after the Unix *objdump* utility. Following compilation, if the appropriate command line flag is set, the assembler displays to the programmer a listing of all macros found, and a human readable table of machine code matched with the corresponding labels and assembly code.

## Demo Application – Duck Hunt

The Plava system was to be demonstrated with a clone of the popular Nintendo Entertainment System game Duck Hunt. This game utilizes all I/O functionalities of the system and takes advantage of the palette abilities of the GPU in a manner which is essential for proper game play. Unfortunately a working version of the game is still in development, and has not yet been demonstrated, though the system has been shown to work with smaller portions of this game.

The Duck Hunt game uses both sprite tables, the first table being used for living duck animations as well as alpha-numeric characters. There are a total of 6 living duck sprites (8x5 tiles) used to animate the flapping of the ducks wings, and to change the direction the duck faces. This leaves one row of sprites for characters; through tricky assignment of palettes to the alpha-numeric characters we were able to overlay four characters into each tile. The second sprite table contained animation frames for the death of a duck should the player be successful.

From a high level the game repeatedly steps through three states, splash screen, game play, and credits. During game play the first thing that happens is that the player is given three bullets. The sound module plays a sound effect of a shot gun being cocked three times, and then a duck is released. The Duck can be controlled by another player using a Super Nintendo controller or with random movements generated by the computer. When the player pulls the trigger the palette for the background is changed to an all black palette, and the duck’s palette is changed to an all white palette. The light detection signal from the gun is then read and if light is detected the game registers a hit, otherwise a miss is registered. On a miss the bullet count drops by one and the duck continues to fly. If the player should run out of bullets, or fail to shoot the duck within 10 seconds, the duck will fly off the top of the screen. On a hit the duck death animation occurs and the players score is increased. Once this is completed another duck is released. The game is over once three ducks have flown away, the credits roll and then the splash screen is displayed. When the player pulls the trigger a new game will begin.

# CONCLUSIONS

Overall, the experience afforded us by creating Plava was indeed a beneficial one. We learned many things about hardware design, and real-world applications of hardware. We studied how the Super Nintendo processes its graphics and were able to learn a lot from it. We learned how the controllers worked and how the light gun worked which was especially interesting. In the sound module we had to learn how to mix sound waves and how to digitally reproduce them. Our project had a large amount of hardware systems that, despite their individual complexity, were able to be integrated easily due to the central role and importance of the memory map, see Figure 1. The value of the memory map was incredible.

In the end our hardware system was very powerful and versatile. The only portion that was limited was the background module, and we have ideas now about how to make it more effective for other applications. One thing that we could have done better was to focus on running a game to demonstrate the power of the hardware rather than focusing so much time and effort on perfecting each piece. If we had spent a little bit more time working with the components that were complete, rather than adding additional pieces such as the sound and the light gun, or trying to perfect our GPU we would have had a much more impressive demo of our hardware.

# ACKNOWLEDGEMENTS

Texas Instruments provided development samples of the THS8134B video digital to analog converter.

# REFERENCES

[1] Altium Limited. November 2008, PB-01 Audio/Video Peripheral Board Schematic, Accessed November, 2010, from: http://wiki.altium.com/download/ attachments/3080506/PB01+Audio-Video+ Peripheral+ Board+Schematics.pdf?version=1&modificationDate=1228268275552

[2] Anomie. July 2007, Anomie’s Register Doc. Accessed November, 2010,from:

http://www.romhacking.net/ docs/ [196]regs.txt

[3] Beazley, David. PLY (Python Lex-Yacc) Accessed November 2010, from: http://www.dabeaz.com/ply/

[4] Kasemset, F. Clock, A. and Kloster, D. Video Target Practice. Accessed November, 2010, from Massachusetts Institute of Technology: http://web.mit.edu/6.111/www/s2004/PROJECTS/12/ report12.pdf

[5] Texas Instruments. March 2000, THS 8134B Triple 8-bit, 80MSPS Video D/A Converter With Tri-level Sync Generation, Accessed November, 2010, from: http://www.ti.com/ lit/gpn/ths8134b