ECE 385

Fall 2014

Chip 8 Interpreter

Avdhesh Garodia
AB7/Friday at 11 AM

Introduction

The final project we chose to do was a implementation of CHIP-8. CHIP-8 is an interpreted programming language that was made in the 70's for simple games, such as pong or space invaders. The implementation of Chip-8 requires a multitude of things to fully function. From a FSM that allows all 35 opcodes to function to a display of 64x32 using sprites. The goal of this project is to be able to run an entire chip-8 interpreter on the fpga, as well as be able to load in games and play them through the fpga. The reason that when given a wide variety of choices we decided to select Chip-8 was due to the complexity of project. Not only was the concept of making a console that can run games interesting, but also the depth of the work needed to be done. Another facet that attracted us to the chip-8 project was the applications down the road in classes like ECE 411. Chip-8 allows not only to maximize the knowledge we learned this semester in ECE 385 but also builds a road towards the future. Instead of a simpler projects that are simply games itself this project focuses on a interpreter. Rewriting a game like snake or pong has almost no creativity as well as being similar to a streamlined project, while creating chip-8 is the next level of a game. Working on Chip-8 we created a platform for games. Allowing us to be able to create games on the platform. We felt that it had more value as a project instead of simply creating a game.

CHIP 8

Here I will state some information about the CHIP-8 system so the rest of the report is easier to understand.

Registers :

V[0-15] - 16 registers each of which is 8 bits wide, these are usually referenced V0 … VF and I will hold that convention. Register VF is used as a FLAG for collision detection, addition overflow, and other operations.

I - A single register 16 bits wide that is used for storing sprite addresses, there are opcodes to read and write this value

SP - The Stack Pointer, A single register 8 bits wide that is used to store the current top value of the Stack.

STACK - 16 registers 16 bits wide that hold the values to the previous PC values when a subroutine is called

Delay_Timer - an 8 bit register that can be set and loaded by certain opcodes, this value will automatically count down at every cluck cycle until it reaches 0

Sound_Timer - similar to the delay_timer in that its set and counts down, however when it reaches 0 it plays a sound. This feature is not implemented.

IR - Instruction Register, a 16 bit register that stores the current Instruction that is being executed

PC - The program counter, A single register 16 bits wide that stores the address of the next instruction

Chip 8 also has some system memory, this stores the sprites for the characters 0 - F in Hex so that programs can use those without having to write that themselves.

Operation of Interpreter

Right now the operation of the circuit is fairly difficult, this is because the the memory sits in the CPU as one giant register. This allows me to write CHIP-8 programs into memory as needed. However if another program that is written by someone else needs to be loade, I have a written code separately in software to generate the verilog memory assignments. To run a program the memory must be loaded with the program you want to run and then the code can be compiled and executed on the FPGA. However, the code takes around twenty minutes to compile since a lot of device resources are used. This makes the debugging process very tedious. A simple state machine could be implemented to load in programs from the wait state, but this would not be very useful in the actual development of the interpreter so it ignored.

Inputs:

Start - Begins the execution of the state machine.

Reset - Resets the System

Keys - The interpreter takes in user input in the form on 16 keys, only one of these keys can be high at a time.

Outputs:

VGA - The FPGA has A VGA output, This is used to display the screen on which all the CHIP-8 programs run. The CHIP8 display has a resolution of 64 by 32 which is then scaled by 10 make the display better suited to see.

Hex Driver - The current instruction that is being executed is sent to the Hex Drivers. This is useful in debugging programs that are not working correctly.

Modules

Not many modules are actually needed in the implementation of the CHIP-8 interpreter. Most of the logic that needed to drive the interpreter is part of the CPU module.

cpu: This is the main module of the System. It contains the FSM and the logic to implement all out the opcodes
clk_divider: Used to reduce the frequency of the incoming clock, More on this later
Hexdriver: Used to output to the Hex Displays
processor: This is the top level module that contains all of the the other modules.
random_number_gen: This module was used to generate random numbers, more on this later
colormapper: This module has the same ideas as the module in lab 7 in that it maps the pixels to their correct colors. However this module had to be written so that it would map the current display to the VGA monitor while also scaling the display up by a factor of 10.
vga_controller: This is the same module provided in lab 7, it generates the VGA signals and the DrawX and DrawY signals that the color mapper can use.
debouncer: This module is used to debounce the buttons, start and reset
testbench: This module is used to run simulations on the system. Very essential when writing something this complex.

Design Process

CHIP-8 was designed to be a simple language to write games. There were never any hardware implementations of CHIP-8 until some calculators picked it up in the 90's. This project is not trying to emulate the hardware of existing CHIP-8 systems but instead, I am trying to design my own CHIP-8 interpreter. This is what makes this project so difficult since, most of the project revolves around the design process instead of the actual implementation of the design. I started this design process by writing a software interpreter. This gave me a better understanding of the CHIP-8 language, and helped my in my hardware design. However there were many things I was able to do in software that I knew would be much harder to implement in hardware.

I started the hardware implementation by working on the main execution cycle of the CPU. This was just a state machine that looped from the FETCH state to the DECODE state along with a WAIT state to start and reset the program. I wanted the design process to start simple and build on already working models and optimize. There are many naive methods to implement the designs in hardware, and since our FPGA is so powerful we can take advantage of that fact and build a design that works but does not optimize for power consumption and device resources. These optimizations can be

done after the the basic design has been laid out. For several aspects of this design it is very hard to determine optimal solutions until they are actually implemented and tested.

Opcodes

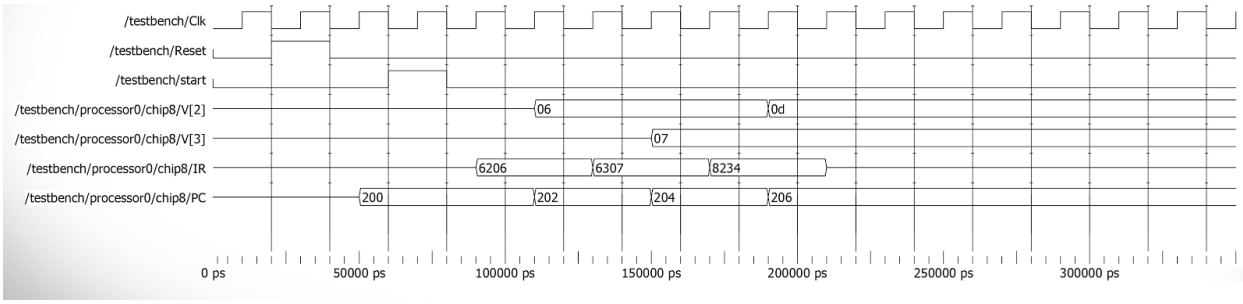
After the basic CPU execution cycle was implemented, I began work on the CHIP-8 opcodes. CHIP 8 has around 40 opcodes, some of which can be very difficult to implement, while others can be done in just a few lines. Each opcode thats written needs to be tested individually since If an opcode is not implemented correctly, programs will not execute. It is easier to debug the opcodes one by one as they are written, instead of when a complex program is executing and we have to isolate which opcode is giving us an issue. This process was done using modelsim since, modelsim although very tedious makes debugging hardware much easier and is essential when working with complex designs. Modelsim lets me look at the state of the CPU, the contents of the memory and the registers in real time which lets me confirm that the opcodes are doing what they are supposed to.

Simulations & Programs

To make the testing of the CPU more comprehensive, I decided to write some programs using CHIP 8 assembly to test the instructions this makes the testing process easier. For instance to test the ADD opcode I would need to force two registers to be certain values add then add them. Instead I used a different opcode to load the registers with two values and then added them. This is the program I wrote:

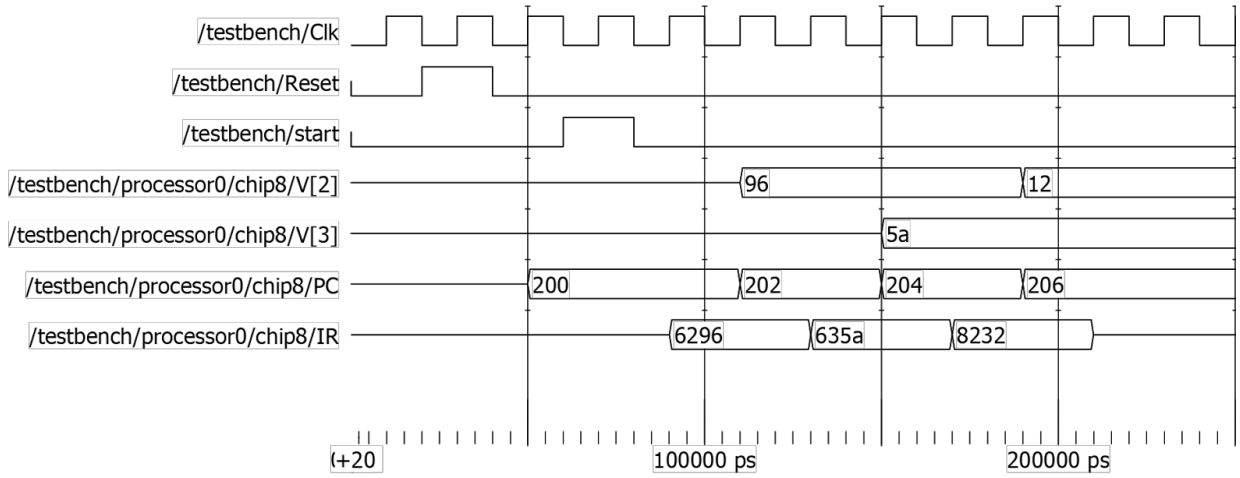
6206 LD R2 <- 06
6307 LD R2 <- 07
8234 R2 <- R2 + R3

The program was just stored in the WAIT state of the cpu so that the program is loaded into memory when the CPU starts executing. This Program was then simulated in Modelsim.



As We can see in the Simulation Register 2 is loaded with 6 and Register 3 is loaded with 7, After The add instruction runs 6+7 = D is loaded into register 2.

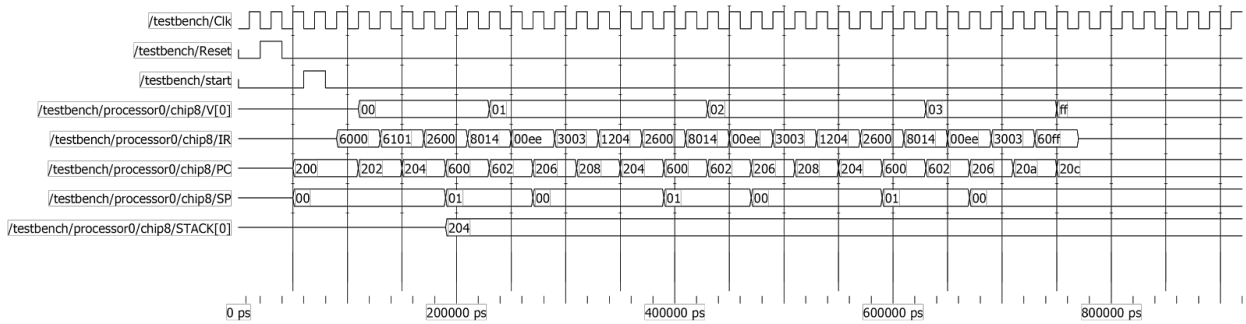
ANother Simulation the program 6396,635a,8232 can be seen in the IR as it runs
It loads R2 with 96 and R3 with 5a. Then R2 = 96 & 5a which is 12.



Finally I wanted to test Subroutines in System Verilog So I wrote This program. I have Included the memory addresses since this program uses conditionals to jump and even calls a subroutine. Since an opcode is 16 bits long 2 places in memory hold the opcode.

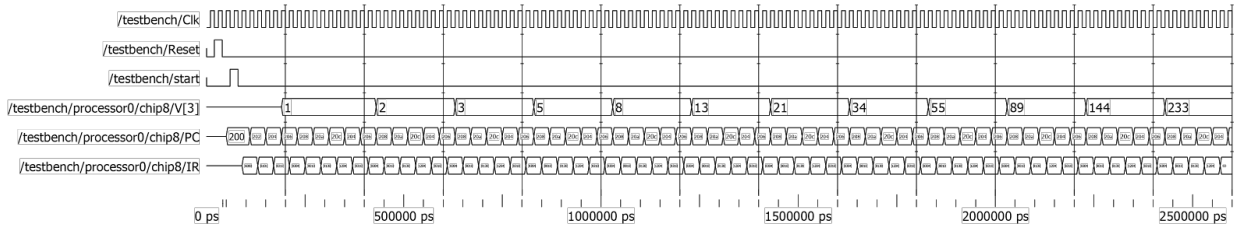
```
memory[12'h200] = 8'h60; //Load Register 0 with 0
memory[12'h201] = 8'h00;
memory[12'h202] = 8'h61; // Load Register 1 with 1
memory[12'h203] = 8'h01;

memory[12'h204] = 8'h26;  //Call Subroutine at x600
memory[12'h205] = 8'h00;

memory[12'h206] = 8'h30; // If x==3 skip next instruction
memory[12'h207] = 8'h03;
memory[12'h208] = 8'h12;  //Jump to x204
memory[12'h209] = 8'h04;
memory[12'h20A] = 8'h60; //Load R0 with FF
memory[12'h20B] = 8'hFF;

memory[12'h600] = 8'h80; // Add R0 <- R1 + R0
memory[12'h601] = 8'h14;
memory[12'h602] = 8'h00; // Return from subroutine
```

memory[12'h603] = 8'hEE;



THis program Basically Increments Register 0 by one until it equals 3 in a loop, then it sets Register 0 to ff to signal that the loop is done.

1: 6000 // Load register 0 with 0
2: 6101 // Load Register 1 with 1
3: 8310 // Set Register 3 to Register 1
4: 8304 // Add R3 = R3 + R0
5: 8010 // Set Register 0 to Register 1
6: 8130 // Set Register 1 to Register 3
7: 1204 // Jump to line 2

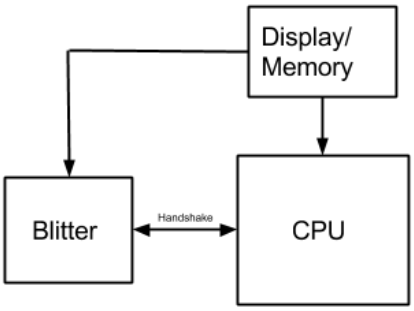


As you can see the fibonacci numbers are generated 1 2 3 5 …

The Display

The display is by far one of the hardest aspects of this project. It was not too difficult to draw the display to the VGA monitor which is what any game related project would require. However for this project I had to consider how to draw to the screen using an opcode that draws sprites from memory and how to represent the 64 by 32 display block in a way that makes drawing simple and efficient. My first approach was very naive, like my software implementation I stored the display logic as one logic block 2048 bits wide. Then to do the drawing I set up a state machine that would loop as many times as the height of the sprite and then XOR 8 bits of the display with the correct location in memory. It sounds simple but it is not, since I needed to calculate the position of the array I needed to draw to based on the X and Y values that we are

drawing to. I also needed to be able to get the 8 bits of the display from the 2048 bit register. This required an initial multiplication to calculate the position and then several additions in each loop to calculate the offsets. The approach worked however, using this method the code would take over 40 minutes to compile since I was using 47% of the device resources. I quickly realized that the approach I used in my software implementation was not good enough and moved on to a different approach. In this approach I store the display logic as a 2D array of 1 bit logic elements. This removes the multiplications that are necessary, however the logic is still very expensive and takes 20 minutes to compile. After doing some research online I learned that systems usually have dedicated logic to do the copy of the sprites from memory to the display. Systems use something called a blitter (Block Image Transfer), this is not specific to CHIP-8, it is used by many older systems. Nowadays however They have been replaced by modern GPU's . A blitter is a block of logic that runs in parallel along with the CPU, The CPU tells the blitter what sprite to draw and then the blitter XOR's the corresponding memory and display bits. This logic block has not yet been implemented since it is an optimization which I don't want to explore until other aspects of the code have been tested and implemented. However I can lay out the design and the improvements that could be achieved by using an XOR blitter.



THe Blitter can run parallel alongside the CPU, this lets me run the blitter at a much higher clock speed. Right now the drawing is down on the CPU which runs at a much slower clock speed so that the user can actually interact with the programs. However this causes the sprites to flash, while being drawn. A bittler would draw then much faster reducing this. To use the blitter a handshake protocol would be needed between the CPU and the blitter. The cpu would tell the bitter when to start and what to draw and then wait until the blitter returns a signal signifying that it has completed.

Display Programs

To test the that the display works correctly some more CHIP-8 programs were written, however instead of testing them using modelsim, I tested these using a VGA monitor. These programs were used to test if the color mapper correctly draws the display registers onto the VGA monitor, and also all of the opcodes used in the drawing process are tested.

Program 1:

```
6206 //Load register 2 with 7
6307 //Load register
A206 //Set Register I to the location of our Sprite
D233 // Draw the Sprite in memory[I] at (V2, V3) that is 3 bytes tall
```

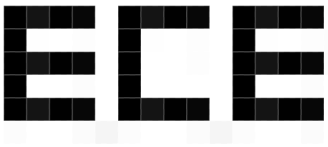
Before this program can be run memory at location I must be forced to the sprite that I want to draw,

```
memory[12'h206] = 8'h8F;
memory[12'h207] = 8'h8F;
memory[12'h208] = 8'h8F;
```

This program will just draw whatever is in memory location x206. This Was used to test that I was able to draw sprites and that they would show up on the VGA monitor

```
6201 6307 610E F129 D235 //Draws  xE
6207 610C F129 D235      //Draws  xC
620D 610E F129 D235      //Draws  xE
```

This program was run on the system to the Drawing Function and the system memory of the CHIP - 8. Since the CHIP - 8 stores the sprites for all the numbers 0 - F in Hex in system memory I was able to write a program that sets the I register to the correct number and then runs the draw opcode. The results on the screen are shown on the left. The sprite data stored in system memory for these characters is very simple



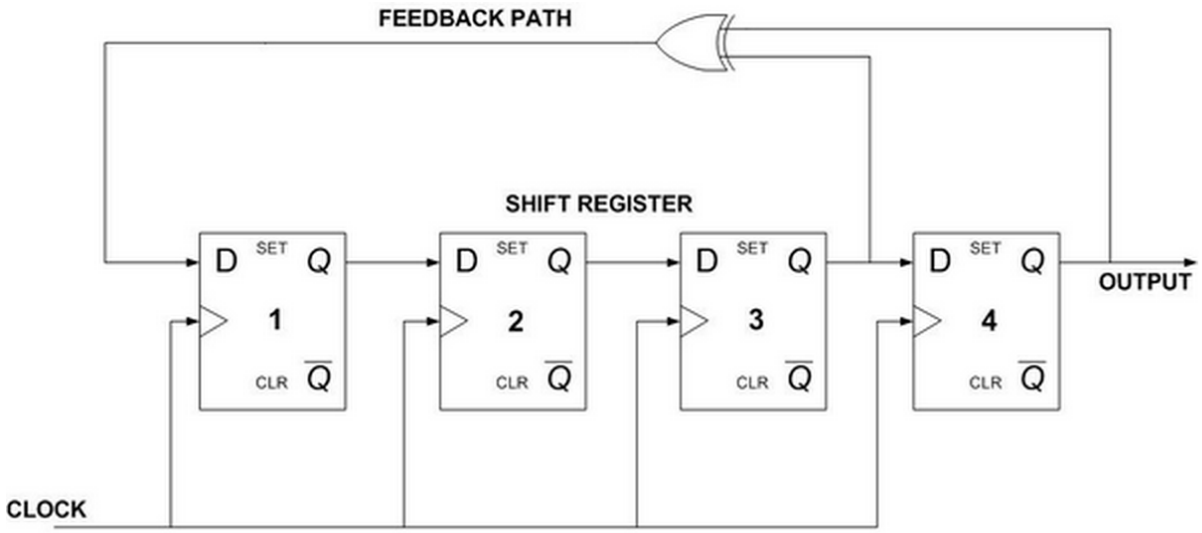
```
0xF0, 0x80, 0x80, 0x80, 0xF0, // C The characters are only 4 bits wide
0xF0, 0x80, 0xF0, 0x80, 0xF0, // E The last 4 bits are 0
```

This program can easily be extended to write any characters and even symbols we want.

Random Number Generator

One of the opcodes for CHIP-8, The C opcode specifically requires a random number to be generated. This is done using a Linear Feedback Shift Register. This is a shift Register whose inputs are controlled by doing an XOR on other bits of the shift

register. This is basically a feedback loop. The starting values of the register is called the seed which is what determines the random number. These numbers are not truly random though only pseudo random, It is impossible to generate truly random numbers and even after you have made a random number generator you cannot know if you succeeded or failed since there is no way to verify randomness. With this method of generating random numbers eventually you will hit a loop of numbers that keep repeating.

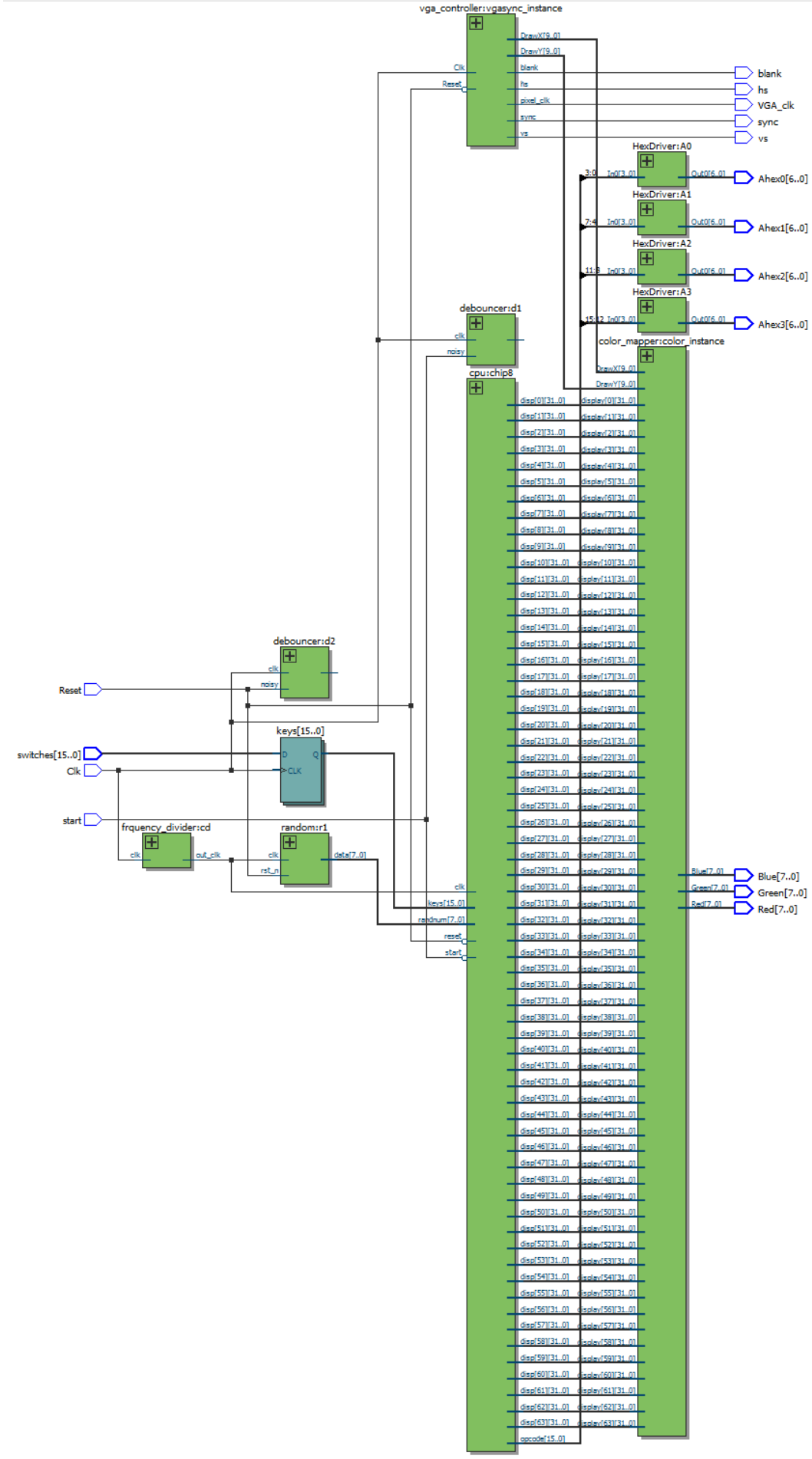


Clock Divider

Since the FPGA has a very fast clk speed the programs when I first ran them ran at too fast a speed for a user to interpret. Things would draw too fast. I needed to write a simple clock divider that is able to take our clock and generate a slower clock. This is done by using a counter that counts on every clock edge. Then when the counter is at a desired value we can clock our new clock. I needed to divide the frequency of our clock by about 35,000 to get a reasonable speed.

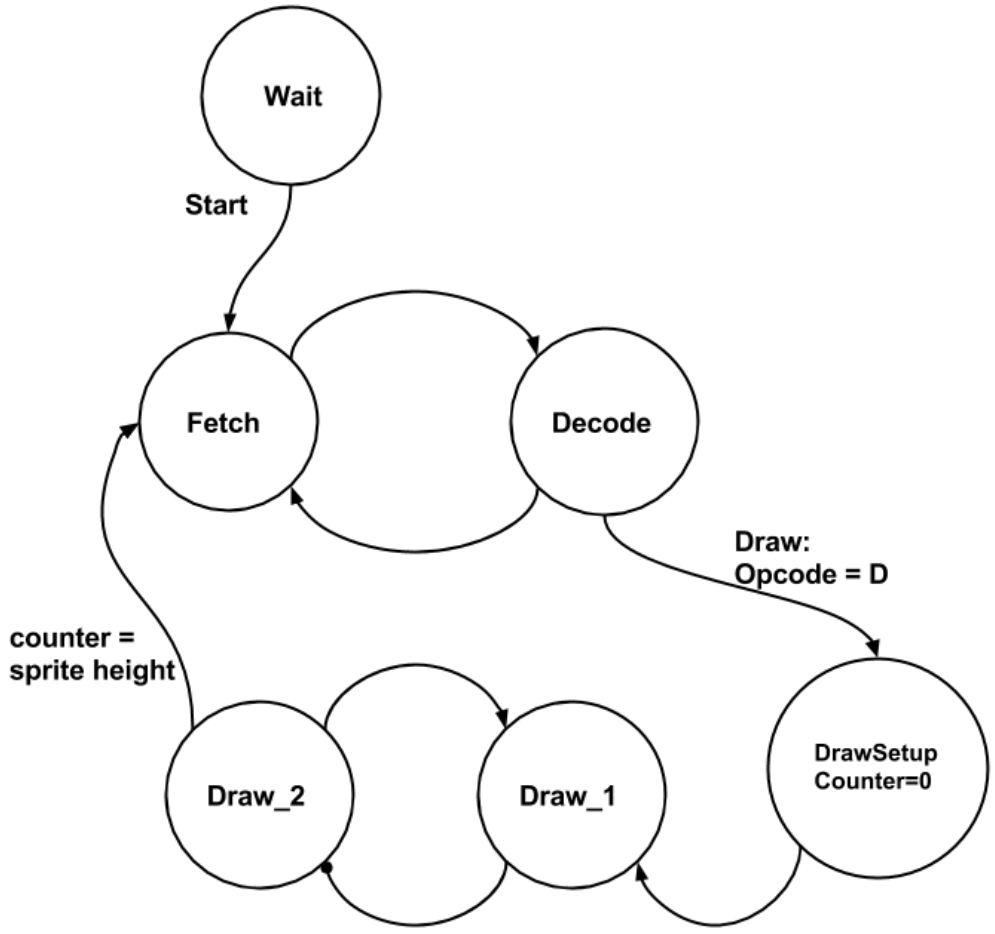
Block Diagrams

The following is a top level block diagram, It needs to pass the display from the cpu to the color mapper so its fairly messy.


vga_controller:vgasync_instance
DrawX[9..0]
DrawY[9..0]
Clk
Reset
blank
hs
pixel_clk
sync
vs
blank
hs
VGA_clk
sync
vs
HexDriver:A0
HexDriver:A1
HexDriver:A2
HexDriver:A3
In0[3..0]
Out0[6..0]
Ahex0[6..0]
Ahex1[6..0]
Ahex2[6..0]
Ahex3[6..0]
debouncer:d1
clk
noisy
color_mapper:color_instance
cpu:chip8
debouncer:d2
Reset
switches[15..0]
Clk
start
keys[15..0]
frquency_divider:cd
out_clk
random:r1
rst_n
data[7..0]
keys[15..0]
randnum[7..0]
reset
start
opcode[15..0]
Blue[7..0]
Green[7..0]
Red[7..0]

State Diagram

The Following is the state diagram that is used to drive the FSM is the CPU, it is not very complex, most of the work is done in the decode step;
State Descriptions:
Wait: loads the memory and the PC and waits till start is pressed
Fetch: Load The IR from memory at the address stored in PC
Decode: Run the opcode that was loaded, if the opcode is D go to the Draw Setup state else we can go back to Fetch.
DrawSetup: Setup the variables needed to draw, in particular set the counter to 0
Draw_1: Draw to the screen, increment counter
Draw_2: If the counter equals the height of the sprite then we are done drawing go to the Fetch state, else we go back to the Draw_1 state.



Design Stats

LUT	29564
DSP	0
Memory(BRAM)	0
Flip-Flop	2399
Frequency	221.51 Mhz
Static Power	101.39 mW
Dynamic Power	0.00 mW
Total Power	156.94 mW

This is very computationally intensive since it uses 28% of the device resources.

Optimizations

Right now my design is very large, mostly due to the display logic. A lot can be done to reduce the complexity of the design. Right now much of the design is done using some of System Verilog's higher level functions, this leaves a lot of the complexity of the design up to the synthesizer. This is bad since I have no idea how it compiles these high level logic functions. To truly optimization the system I need to re design various aspects of the System to account for even lower level elements. Since CHIP-8 is an older system I also could be taking better advantages of modern technology. Nowadays we are able to run systems are much higher clock speeds. I can split the cpu up into several takes and reuse several of the components I have already built instead. Speed is not an issue since as mentioned the clock already needs to be divided significantly for the CPU to even run at a usable speed. However a lot of these optimization are beyond my current scope of knowledge and would require a lot of research

Conclusion

This project was very ambitious and was mostly used as a learning experience for me since several areas of the project were very difficult to implement. Going into this project I assumed that if I can write it in software then a hardware implementation should not be too hard. I was very wrong, this project was very difficult to implement. Also debugging was very difficult as well since modelsim needed to be used a lot and

since the code takes 20 minutes to compile debugging on the FPGA is also difficult. Even though this project was very difficult I enjoyed it, since a lot of its implementation was left to me and I had to come up with designs that I thought might be optimal. Overall, I think I did a fairly good job with this project, and will continue to work on It.

Future Work

Some things that I am considering for continuing this project. As mentioned above a Blitter module could be implemented to make the display writing better. also the memory and display could be stored in RAM to take up less resources on the FPGA. One thing that I really want to implement is Super Chip - 8 which is an expansion on Chip - 8 that adds 10 opcodes, and a larger display resolution. Super Chip 8 has more programs written for it which are better than CHIP 8. Finally, it would be interesting to add opcodes that have functions to write to memory using different addressing modes. This would allow the system, to do anything that any other processor could do and it would be interesting to see if a simple operating system could be written

References

The following link contains technical specifications about CHIP-8
http://devernay.free.fr/hacks/chip8/C8TECH10.HTM