CHIP-8 Emulation on CY8CKIT-059 PSoC Kit

6.115 Final Project Report

Keshav Gupta (keshav21@mit.edu)

April 8, 2019

1 Introduction

CHIP-8 is a simple interpreted programming language. It was first developed for the COSMAC VIP computer to make programming games for it easier. It supports a 64×32 monochrome display, 4×4 keypad input and a monotone buzzer for sound output. Despite it's limited hardware resources, it proved to be very easy to write games in, resulting in interesting implementations of Connect-4, Space Invaders and many more such classics.

The inspiration for this project comes from the final project idea about Atari 2600 emulation on the class web page, which prompted me to read about console emulators. With some research I determined that CHIP-8 was a good starting point for emulation since it hits the right simplicity-to-usefulness point.

2 System Overview

The system uses VGA output and an OLED screen for display, a generic buzzer for sound and the onboard EEPROM which stores program data. New games or programs can be loaded onto the board using the USB UART connection and a simple Python script that I implemented.

2.1 Internal Hardware Overview

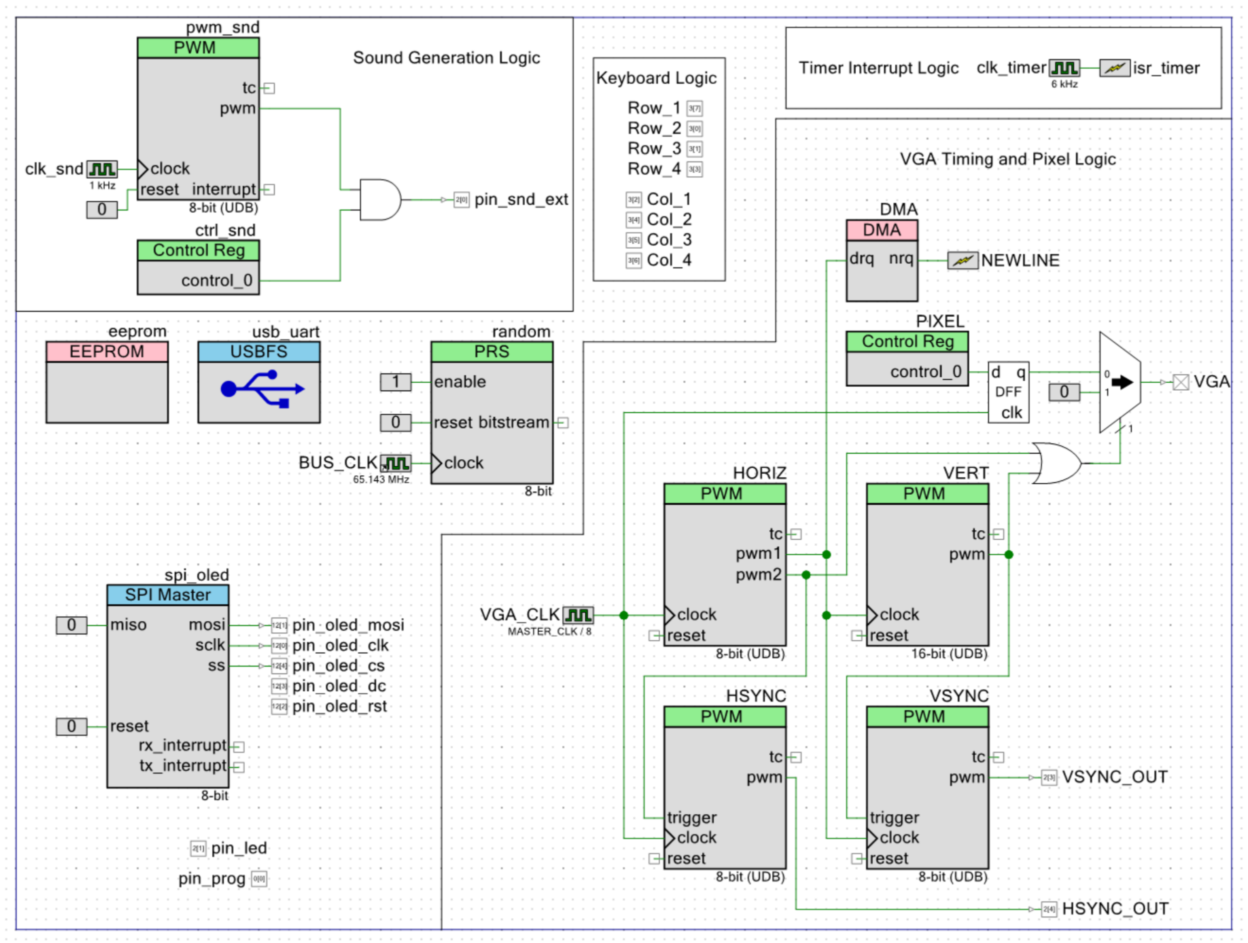


Figure 1: System Internal Hardware Diagram

Figure 1 details the circuit schematic inside the PSoC Creator software, and which exists inside the PSoC too.

2.2 External Hardware Overview

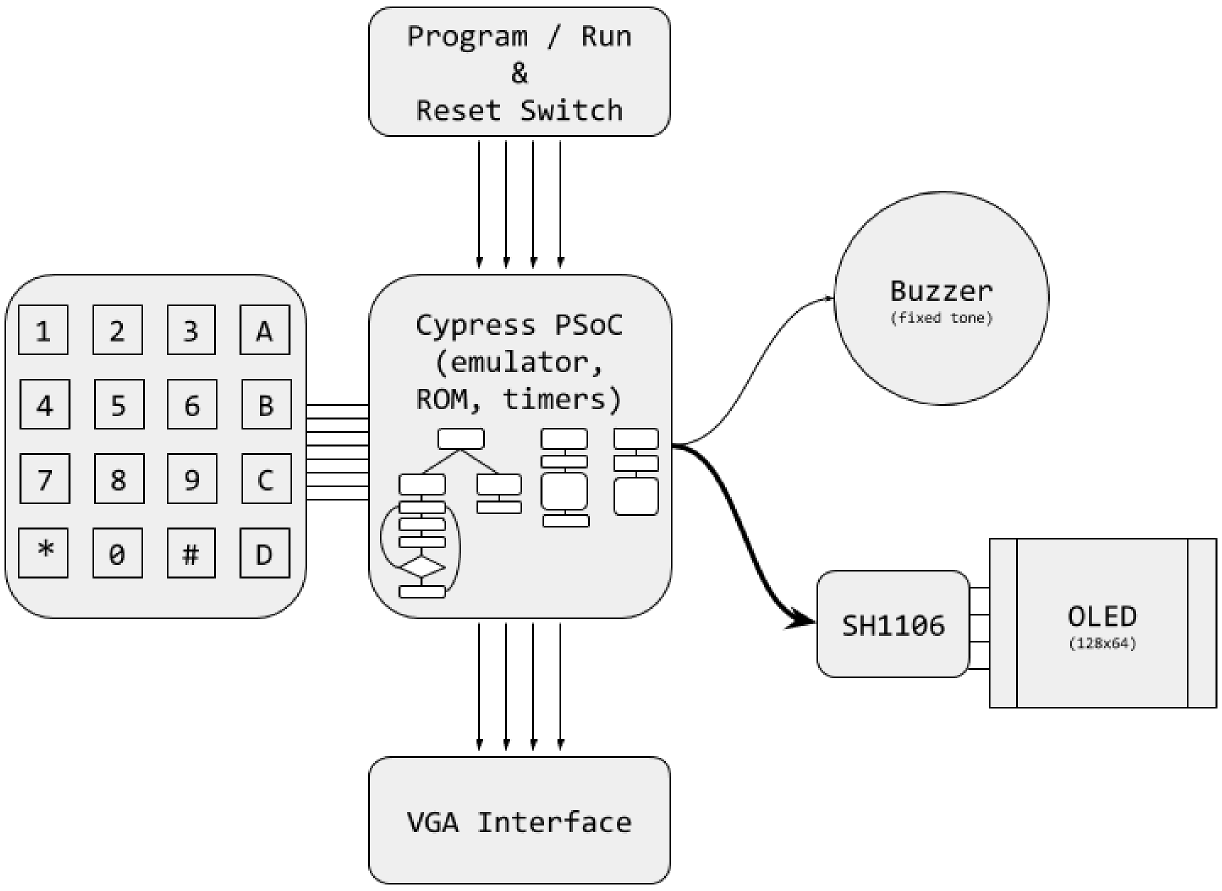
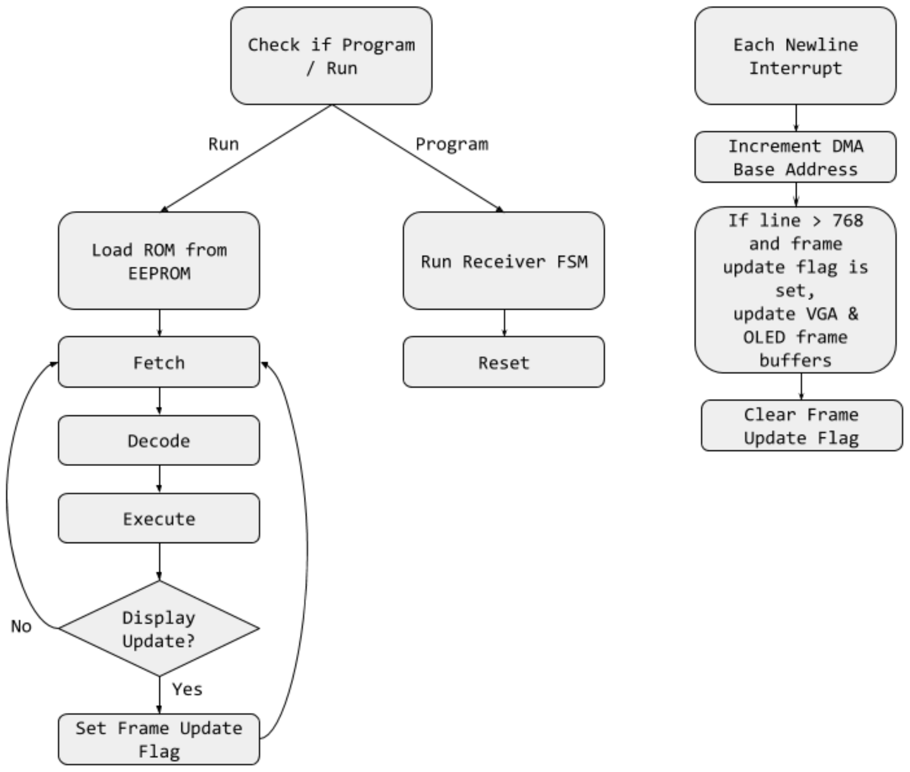


Figure 2: System External Hardware Diagram

Figure 2 describes the circuit I assembled outside of the PSoC, on a Veroboard.

2.3 Software Overview


Check if Program / Run
Run
Program
Load ROM from EEPROM
Run Receiver FSM
Fetch
Reset
Decode
Execute
Display Update?
No
Yes
Set Frame Update Flag
Each Newline Interrupt
Increment DMA Base Address
If line > 768 and frame update flag is set, update VGA & OLED frame buffers
Clear Frame Update Flag
Each Timer Interrupt
Decrement both Sound and Delay if > 0
If sound > 0 still, assert the sound control bit, otherwise clear it.

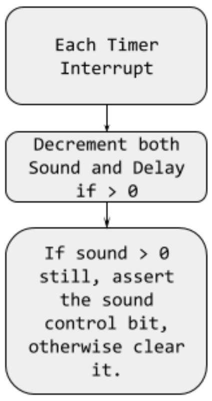

Each Timer Interrupt
Decrement both Sound and Delay if > 0
If sound > 0 still, assert the sound control bit, otherwise clear it.

Figure 3: Software Flowchart

3 System Description

This section describes each subsystem of the project in detail. Code and the Cypress PSOC Creator project can be found on the GitHub repository for this project [2].

3.1 Emulator Core

The emulator core is implemented in `procr.c`. It has access to a working copy of the program data, as well as abstract interface endpoints from other subsystems such as the Keyboard and Display.

The working memory is stored as a 4096 8-bit byte array, the video data is stored as a `uint64_t` array `vram` of size 32, one bit for each pixel on the screen. I decided upon this representation since this would make it very convenient to move pixels across the entire screen, simply by shifting the number left or right.

The stack is implemented as a 16 long 16 bit array and a stack pointer as an index into the array, with the primary purpose of keeping track of return values for function calls.

3.1.1 Random Number Generation

I use a Pseudo-Random Number Generator component to generate random bytes upon request from the program under emulation.

3.2 Keyboard

I use the Official 6.115 Keyboard for input but instead of using the 74C922 driver IC as originally planned, I implemented my own driver written in software (`keybd.c`). I made this choice because both behaviors - waiting for a keypress, and determining if a key is currently pressed - were required.

The four row lines are connected to pins pulled down by a 10k resistor internally. To check if a key is currently pressed, the corresponding column line is asserted and the corresponding row line is checked.

The other behavior, waiting for a key, is implemented entirely in software. All 16 keys are checked one by one and the key first found to pressed is returned. I also implemented a key debouncing FSM, which works by sampling the key once every 5 milliseconds, for 50 milliseconds, and then waits for the key to be released.

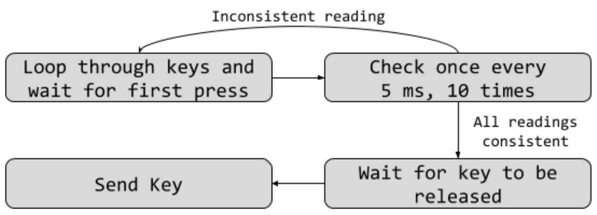

Inconsistent reading
Loop through keys and wait for first press
Check once every 5 ms, 10 times
All readings consistent
Wait for key to be released
Send Key

Figure 4: Key Debounce FSM

It shall be noted that the physical row and column numbers are different from the values written on the keycaps, therefore I use a matrix to convert between the two.

3.3 VGA

My implementation of VGA derives heavily from the staff PSoC VGA implementation. I retained the PWM blocks that generate the timing for 1024 × 768 (60 Hz) VGA. I then increased the clock frequency from the default of 24 MHz to 65 MHz to match the desired pixel frequency.

I changed the color information register from 6 bits to 1 bit, and rewrote the newline interrupt service handler to convert from the processor `vram` (Section 3.1) to the `vbuf` appropriately. Since processing each of the individual pixels takes considerable CPU time, the OLED buffer (Section 3.4) is also populated along with `vbuf`.

3.4 OLED and SPI

The OLED driver (SH1106 [1]) connects to the PSoC over a 4 pin SPI connection, of which three pins: `CS` (Chip Select), `SI` (Slave In) and `SCL` (Slave Clock) are provided by the hardware SPI module available as a digital logic block, whereas the 4th pin `DC` (Data/Command) is provided by an additional Digital Pin interface. This connection is provided by the `u8g2` library, which is only available for the Arduino platform, but I ported it to work on the PSoC.

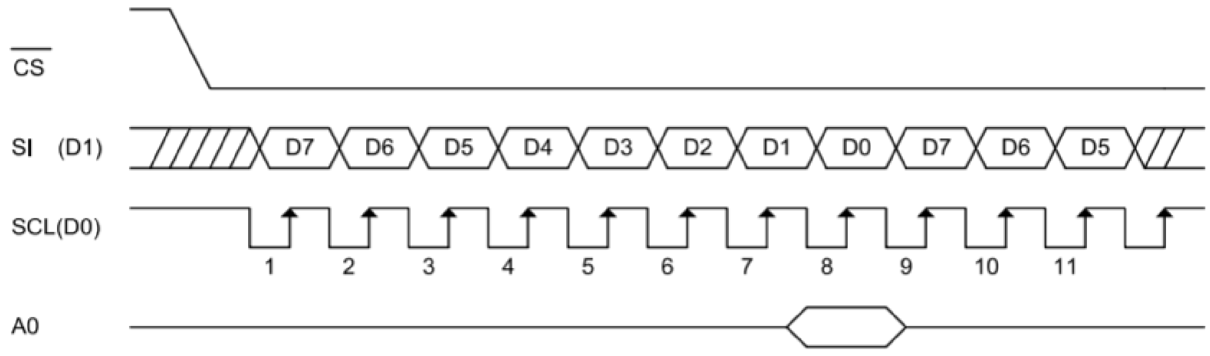


Figure 5: Desired SPI waveforms, according to the SH1106 Datasheet [1]

- `CS`: This pin is active low, therefore when asserted low, it enables the OLED and it receives data from the PSoC.
- `SI`: This pin shifts out data to be sent to the OLED.
- `SCL`: `SI` is sampled at every rising edge of this pin if `CS` is low.
- `DC`: This pin dictates whether the input data is interpreted as Data signals (low) or Command signals (high).

The data for the OLED is stored in an internal buffer in the `u8g2` library implementation, but I access it directly. I did not find this to have any undesirable side effects.

3.4.1 OLED Transmission Bandwidth

The datasheet [1] states that the maximum transfer speed in 500kbps, however I experimented with much faster speeds of upto 18Mbps (limited only by the hardware SPI digital block), and found those to work reliably in the long run too. I finally settled onto 16.28 Mbps, since it divides well into the 65.14 MHz clock speed. At this rate, it takes a total of 680 μs to update the internal OLED buffer for one frame and write it to the display. Combined with 70 μs it takes to update the VGA buffer, it fits comfortably within the blanking period of 770 μs, therefore no SRAM bus contention issues arise.

3.5 Sound and Timer

The primary means of controlling timing on the CHIP8 platform is through two timers: Delay and Sound. Both count down at 60 Hz when nonzero. Both these timers can be written to. The Delay timer can be read from, thereby enabling the program to time events with reasonable precision and the Sound timer makes a beep when nonzero, thereby enabling the program to emit a beep of a certain duration.

For the sound, I write values to a control register, and when ANDed with a PWM signal of 1 KHz it produces the requried waveform. I used a piezoelectric buzzer that consumes around 5 mA and can be powered by the PSoC without any buffering required.

3.6 USB UART and EEPROM

I used a USB UART component to enable communication with a computer to download a program to the onboard EEPROM without requiring reprogramming of the entire PSoC flash.

I designed a very simple protocol to achieve transmission, and wrote a simple Python script that transmits ROM data. The receiver is a simple FSM, with a state diagram as in Figure 6.

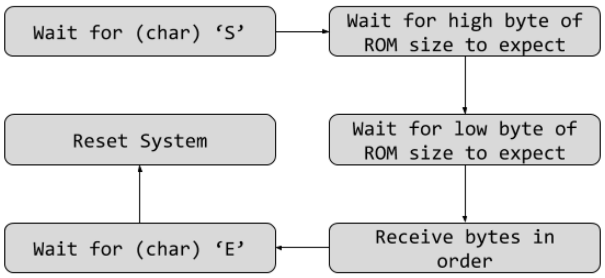

Wait for (char) 'S'
Wait for high byte of ROM size to expect
Wait for low byte of ROM size to expect
Receive bytes in order
Wait for (char) 'E'
Reset System

Figure 6: UART Receiver FSM

4 Challenges

This section describes some of the challenges I faced while working on the project.

- Initially, the OLED and the VGA buffers were refreshed separately, on different interrupts. The DMA faced multiple issues with bus contention due to the OLED interrupts being out of sync with the VGA interrupts and overlapping, which was evident from equidistant lines on the VGA output. I solved this by updating both buffers at the same time and updating them in sync on the same interrupt service routine.
- My initial implementation for the Keyboard relied on asserting the Column line for a single cycle, checking the row line on the next cycle, and asserting it low on the

third cycle, thereby giving the column line only 46 ns to go high and back. Such a sharp rise and fall were impossible given the large impedance of the keyboard, so I introduced artificial delays of around 1 ms, which I found to be enough to guarantee reliable readings from the keyboard.

- I could not get UART transmission to work with the Python script, since I was not aware of the Python abstractions for dealing with data on a byte level. I had to resort to using the `struct` module, which helped in the conversion from characters and integers to raw bytes.
- The `u8g2` library includes a huge library of fonts, which totals over 145 MB after compilation. Since Link Time Optimization (LTO) was not enabled for the debug build, the compiler tried to fit all that unused data onto the PSoC flash. This problem vanished when I enabled LTO.

5 Acknowledgements

First off, I really appreciate all the effort Professor Leeb has put into the class, and thank him for the same. I also thank all the TAs who've saved me from countless hours of debugging. This class made me appreciate all the advances in ECE which I otherwise took for granted.

References

[1] SH1106 (OLED Driver) Datasheet: `https://www.velleman.eu/downloads/29/infosheets/sh1106_datasheet.pdf`

[2] `6.115-final` on GitHub: `https://www.github.com/keshavgupta21/6.115-final`