Frito: an FPGA Chip-8 Multiplexing Emulator

Linh Nguyen
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
Email: linnie@mit.edu

Vetri Vel
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA, USA
Email: vetri@mit.edu

Justin Yu
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
Email: jyu161@mit.edu

***Abstract*—We present an FPGA-based emulator for the Chip-8 interpreter, implemented as an emulated processor on an FPGA. It supports several customization options which can be set by the user in a configuration menu and is designed in parallel manner, which can allow multiple Chip-8 instances to run on the same processor. Our source code is at https://github.com/fractal161/frito/.**

1. Background

Chip-8 is a tiny interpreted programming language originally built on the COSMAC VIP, consisting of 35 instructions reminiscent of assembly. It has 4KB of RAM, uses sprite-based rendering on a 64x32 pixel buffer consisting of two colors, and can output a singular audio tone at various intervals.

We emulate each of the outward-facing components of the system with varying levels of faithfulness. Graphics are communicated through HDMI using 1280×720 resolution, audio through our board's standard output port, while input comes through a 4×4 keypad, similar to the original hardware.

In Frito, the fundamental module is the processor, which connects all other modules together in manipulating state. To do this, it communicates with modules which handle graphics, audio, and player input. The state itself is contained inside the memory module, which passes queries to the correct Chip-8 instance and also connects to a multiplexer module, which looks at the video buffer state to determine what to draw onscreen. In addition, we have a toggle between this and a custom configuration interface, but this is less relevant to the project itself and more a way to show its capabilities with less friction. A high-level block diagram can be found in the appendix at the end of this report.

2. Multiplexing (Justin)

Frito is designed to be multiplexed, which we define as having multiple instances running in parallel. Our system is capable of supporting up to 36 simultaneous instances. The need to support this feature influenced many of the core design choices, so we elaborate on those here.

Frito's memory core resides in a single BRAM of width 8. This contains the entire processor state, including the program's main memory, the video buffer, the processor's registers, and the stack. This requires a total of 4407 bytes, which is just enough to fit on one 36 kilobit BRAM. This design causes a considerable slowdown in just about every operation, as even reading a register incurs several cycles of latency. We can somewhat mitigate this latency through pipelining read and write requests, but this cannot help if, say, the next byte to be fetched depends on the most recent one requested. Despite this, our design lets one processor module easily switch between different contexts by changing what BRAM it is pointing to, which is effectively instant.

For simplicity, we exclusively use BRAM storage for each memory core. This is enough space to display a 6×6 grid, and since it appears we have the equivalent of seventy-five 18 kilobits of BRAM space, this is the limit of our approach.

Running the processors is done in a threaded fashion: every simulated tick, the processor module updates each core sequentially, and the system waits for the next tick once each cycle is finished. With a simulated clock speed of 500hz, this allows us $\frac{10^8}{500 \cdot 36} \approx 5555$ FPGA cycles per instruction, which is well above what's necessary.

Naturally, the ability to run multiple instances is meaningless if we have no way to view them. This aspect is delegated to the *video multiplexer*. This is driven by the HDMI's 74.25 MHz clock, and displays each active processor in a configurable grid. To ensure the spacing of each display is consistent, we use a division module to compute the optimal padding and infer which processor to render using the supplied `hcount` and `vcount` of the HDMI signal generator.

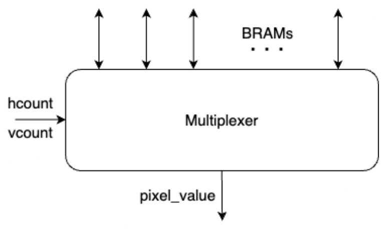


Figure 1. Diagram of the function of the multiplexer

3. Emulation

3.1. Processor (Justin)

Implementing a processor for an interpreted language is somewhat improper, and as a result we have to make several compromises. For example, the original interpreter takes a variable amount of time to execute each instruction depending on several factors, like the instruction's arguments and even the time within a frame.[1] Since a thorough simulation of these timings is prohibitively time-consuming, we compromise by selecting a constant clock rate of 500Hz, which is recommended by [2]. To further address potential timing issues, we will also allow the user to fine-tune this in the configuration screen.

We implement a sequential processor using a conventional state machine approach. Pipelining would be largely ineffective in this context, because the execution `CLS` and `DRW` (clear screen and draw sprite, respectively) dominate the runtimes of all other operations, and this is inherently because they write to several memory locations. The processor will spend the majority of the time in an idle state, waiting for the next Chip-8 clock cycle (the module itself is driven by the FPGA's 100MHz clock). From here, the processor fetches the program counter (which is stored in the BRAM for modularity) and then fetches the opcode it points to. Afterwards, each instruction's execution is handled in a separate state entirely within the processor. The two exceptions are `CLS` and `DRW`, which both manipulate the video buffer. These are handled by a separate video module, whose implementation is detailed in a later section. The precise definition of each instruction is documented in [3], which also contains information on compatibility quirks.

3.2. Memory (Justin)

The design of this module was heavily influenced by the discussion on multiplexing from earlier, so this is essentially a wrapper around a dual-port BRAM. One port is reserved for the *video multiplexer*, which benefits from consistent access to the Chip-8 instance's VRAM to determine what pixels to draw on screen. Thus, all other modules which request access to the BRAM must do so through the other port, which include the processor module, the video module, and a debug module for displaying memory addresses on the seven segment display. The system that coordinates this access uses a simple protocol that relies on a strict hierarchy between the types of modules that want access, which prefers the processor, then the video, then debug.

Game ROMs are instantiated at build-time in each memory core. While it would be nice to allow ROM selection from an in-game menu rather than needing to rebuild the module each time, this tradeoff was ultimately made so all BRAMs could be used for processors and not just as cold storage for a game that might be accessed. This setup does allow us to control which game appears on which spot in the Chip-8 grid pretty easily.

3.3. Video (Vetri)

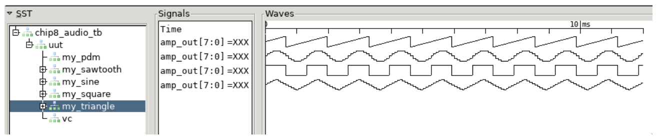
The video module responds to two types of instructions: draw sprite and clear buffer. As mentioned previously, the 64 by 32 pixel video buffer is stored in BRAM, access to which is mediated by the memory module. Since each pixel is binary, the video buffer is stored as 256 consecutive bytes in BRAM. To implement the clear buffer instruction, the video module sets each byte in the video buffer to zero through the memory module. To implement the draw sprite instruction, it requests bytes from the memory module, first for sprite data, then for old video buffer values. It XORs these bytes to get the new video buffer value, which it then writes back to the buffer through the memory module. Sprites are one byte wide and the x position at which to start drawing sprite does not have to be a multiple of 8, so drawing each line of the sprite usually requires updating two adjacent bytes in the video buffer. Sprites are drawn such that there is clipping in both the x and y directions. Only when a sprite is completely out of the screen does it wrap around to the other side. Correct implementation of clipping was confirmed using the "Quirks" ROM, which tests the functionality of essential Chip-8 features [4].

3.4. Audio (Linh)

The audio module takes in inputs for timbre (type of wave), tone (frequency), volume, and produces an audio sound that corresponds to the inputs. The module generates an audio signal by having a clock management system that will generates a clock of approximately 98.3 MHz. With a decimation factor of 1024, it is able to output an audio signal at 3kHz. This sample rate was chosen because we are representing only up to 1.5kHz of audio content and want to reduce processing power.

Each type of wave has its own module and generates its respective wave with the appropriate sampling rate through a trigger that corresponds to the phase step of 3kHz. When the trigger is high, the phase increments by a phase increment that corresponds to the input tone, which can be found within a tone look up table module that maps a tone to its corresponding phase increment that outputs audio at 3kHz.

1. For example, the draw sprite instruction `Dxyn` will wait until blanking to begin execution.



```
Time
amp_out[7:0] =XXX
amp_out[7:0] =XXX
amp_out[7:0] =XXX
amp_out[7:0] =XXX
```

Figure 2. Testbench results for sawtooth, sine, square, and triangle waves, respectively

The top 6 bits of the phase value gets generated and gets looked up within its corresponding waveform look up table that outputs an amplitude.

Each 'Wave' module (Sine, Square, Triangle, Sawtooth) is instantiated within the 'Audio' module. The selected timbre from the 'Wave' module serves as the audio output, which is then routed into a 'Volume Control' module. The 'Volume Control' module adjusts the volume of the audio output based on the input volume. If the 'active in' signal is high from our processor, the input to the 'Volume Module' will be the output from the 'Wave' module, and if the 'active in' signal is low, the the input to the 'Volume Module' will be 0. This conditional ensures that when the output of 'Volume Control' gets fed into a 'PDM' module, the output of the 'PDM' module can correctly output sound when the 'Active in' signal is low. The purpose of the 'PDM' module was to upsample the audio output. The output of the 'PDM' module gets feed into the speakers of the FPGA to produce the audio to the user when they plug in headphones or a speaker to the FPGA.

3.4.1. Verification of Audio. In order to verify the wave and tones of our output, we used a variety of test benches. First, to verify the sine, square, triangle, and sawtooth shape of each of our wave output, we made a test bench to output the waves in GTK wave. Through this simulation, we were able to verify that our waveforms are correct and produced an accurate sound.

For testing the accuracy of the tones in our generated audio output, we connected the FPGA to an oscilloscope to measure the output frequency of our sounds, and we were able to verify that the output frequency of our audio were 750Hz and 325Hz.

In order to test the volume of our audio, we used the FPGA switches to correspond to different volume outputs to verify that our sound's volume can be altered based on user input.

Lastly, we verified that our 'Audio' module worked succinctly with our Chip-8 system by using an existing audio test rom that played morse code of "SOS". We were able to use this test rom to produce audio at varying tones, soundwave, and volume based on the user's input.

3.5. Input (Vetri)

The input module interfaces with the 4x4 matrix keypad and outputs a 16-bit value. Each bit corresponds to the state of a key - 0 for unpressed and 1 for pressed. The 4x4 keypad has 4 pins corresponding to columns and 4 pins corresponding to rows. When a key is pressed, it connects its corresponding row and column together. Pressed keys are determined by setting one column to 0 and the rest to 1, then checking which rows also get pulled down to a 0, which indicates the keys along that column that are pressed. This is done for each column in order. It takes 8 cycles to poll the whole keypad. The four row pins are wired to `pmoda[3:0]` and treated as inputs, while the four column pins are wired to `pmodb[3:0]` and treated as outputs.

To prevent rows from floating low without a key press, all four rows are connected via pull-up resistors to 3.3V, as illustrated in figure 3. This ensures that unless a pressed key directly connects a row to a column, rows will remain high. We use 10kΩ resistors and the wiring is done on a breadboard. One limitation of our system is that if two or more keys on the same row are pressed simultaneously, there is indeterminate behavior where the row is connected to both ground and 3.3V at the same time.

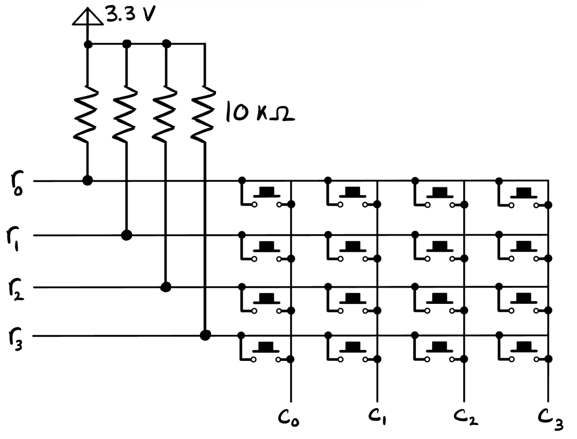


Figure 3. Schematic of the 4x4 matrix keypad and circuit wiring

4. Configuration Menu (Justin)

Frito includes a configuration interface that can be used to tune many of the underspecified parameters of the Chip-8 specification. While it would have been convenient to reuse much of the work done for the emulator itself, the architecture is just vastly more limited than what's possible, so this module instead defines an ad-hoc format for displaying monospaced text, along with a cursor that indicates which row can be modified. A majority of parameters can be represented with a single hexadecimal digit.

We use two BRAMS for this: the first stores the data for each symbol (a letter or number) and the names for custom menu items (for example, the `TIMBRE` parameter can be one of sine, triangle, square, or sawtooth, so we indicate this using actual words instead of numbers), and one for storing the current visual state. The first BRAM ends up being 2160 bytes (for some reason I overallocated space for 256 symbols, we actually use around 36), while the second is 920 bytes, so each fits in an 18kbit space. We aren't able to combine these due to how the ports are assigned: the former needs one port for fetching symbol data and the other for fetching menu row data, while the latter needs one port for receiving updates (when a parameter is changed), and another for reading data to send through HDMI.

5. Evaluation

Our Chip-8 emulator is feature complete, meaning it supports all instructions and passes all standard tests. Specifically, we verified the module works with the Corax, Flag, Input, Audio, and Quirks test cases from the Chip-8 Test Suite [4].

We are also able to run just about every game that we tested, which were sourced John Earnest's Chip8 Archive [5]. While each game is able to work decently well, we observed slight differences in their speeds compared to the example runs provided in the Archive; this is ultimately due to our simplistic assumption of a 500hz clock cycle. In addition, some fast-moving graphics exhibit visible screen tearing, but this is generally not noticeable, even with the smallest screens.

Multiplexing was verified in two ways: first by testing multiple instances of one ROM that uses randomness, and by testing several ROMs at the same time. For example, we used Rock Paper Scissors with the former and were able to observe many different scores with the same number of attempts. Verifying the latter was even more straightforward, as we already know how each game behaves when presented on its own.

Each instruction seems to complete in an acceptable time, which we determine by simulating the processor's performance on test ROMs in iVerilog. As discussed before, the longest taking instructions by far are `CLS` and `DRW`, which clear the screen and draw a sprite. `CLS` takes around 270 cycles to execute, and this should be stable since it accepts no arguments. On the other hand, `DRW` takes 95 cycles to draw four rows of data; since the maximum number of rows is 15, this still falls comfortably within our 5555 cycle threshold.

As discussed above, it is unlikely that we can improve on our usage of resources using this approach; when combining the thirty six 36kbit instances used for the processors with the two 18kbit instances used for the config menu, we almost exactly use up all of our BRAM space. Any enhancements would require a complete refactor using a different storage system, like through DDR3 or streamed through something like Manta. However, this introduces a considerable latency overhead, making it unlikely that we could substantially improve on the number of concurrently running processors.

6. Takeaways

Throughout the process of designing this project, we came to the following realizations:

- Unnecessary complexity is unnecessary. In particular, the design of the memory module (which the writer of this bullet originally thought was clever) ended up solving a problem that never really existed and caused several times more bugs which often needed even uglier patches. Having a simpler interface would easily have been worth the extra boilerplate that would've been incurred.
- Having many checkpoints is good. The original scope of the project included compatibility with alternate Chip-8 specifications, as well as a much larger grid of processors. However, once these goals became infeasible given the allotted time, our intermediate objectives became more than acceptable fallbacks.

7. Contributions

Each of our previous sections were all essentially done in full by the credited person. In addition, Justin handled the high level design (especially pertaining to the multiplexing), Vetri was responsible for the diagrams in this report, while Linh directed and edited our final video.

References

[1] F. Moseley, Manta, https://fischermoseley.github.io/manta/ (accessed Nov. 22, 2023).

[2] J. Sommerich, "Chip 8 Instruction Scheduling and Frequency," Jackson S, https://jackson-s.me/2019/07/13/Chip-8-Instruction-Scheduling-and-Frequency.html (accessed Nov. 22, 2023).

[3] T. V. Langhoff, "Guide to making a CHIP-8 emulator," Tobias V. Langhoff, https://tobiasvl.github.io/blog/write-a-chip-8-emulator/ (accessed Nov. 22, 2023).

[4] Timendus, "Chip8-Test-Suite," GitHub, https://github.com/Timendus/chip8-test-suite (accessed Nov. 22, 2023).

[5] J. Earnest, "CHIP-8 Archive," https://johnearnest.github.io/chip8Archive/ (accessed Nov. 23, 2023).

Appendix

We include the block diagram for the processor on the next page.

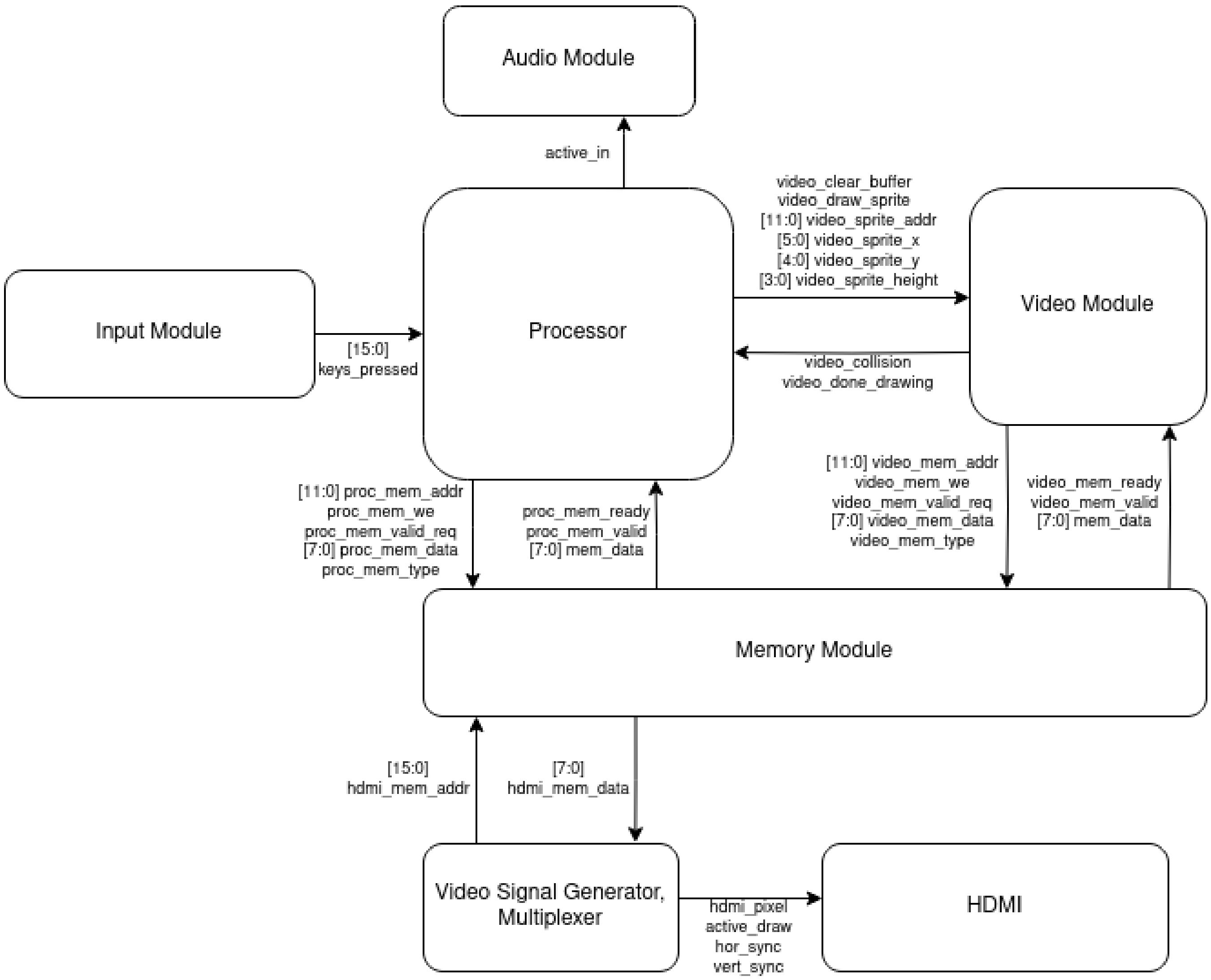


Figure 4. Block diagram for the processor.