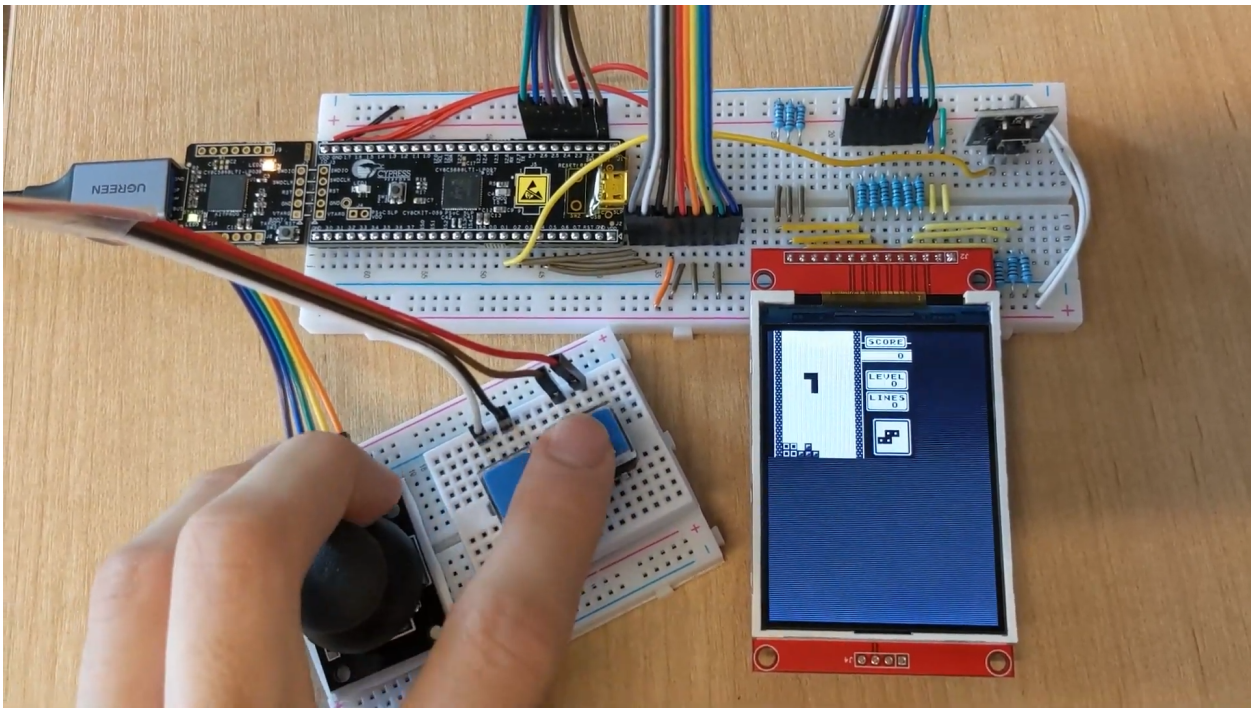
The PSOC 5 Game Boy Emulator

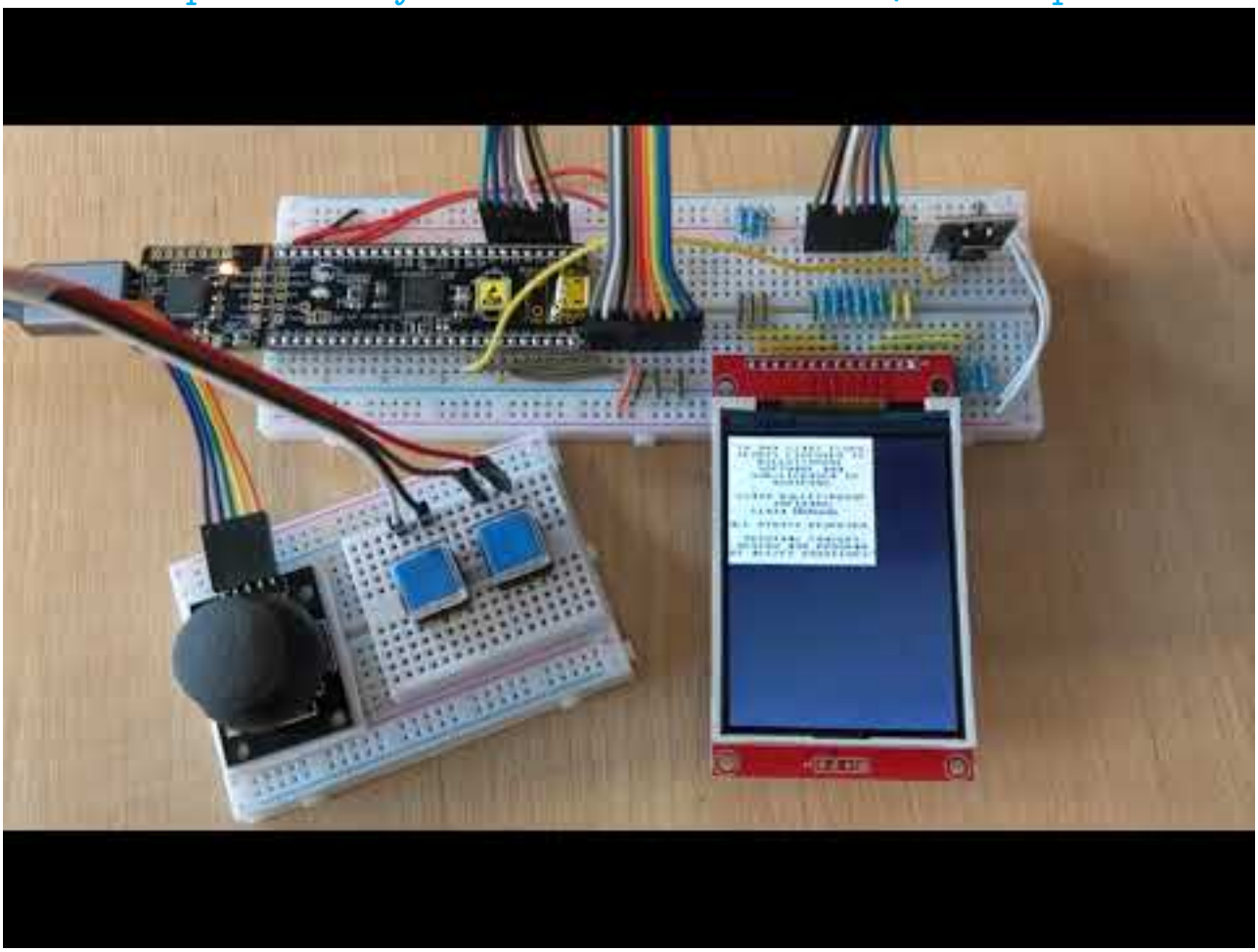
A 6.115 Final Project

Raymond Tran
raytran@mit.edu



May 14, 2021

A brief video report is available here:
https://www.youtube.com/watch?v=4Qr0IfhJqd8



Contents

1 Introduction and Motivation **1**

2 Project Overview and Layout **1**
- 2.1 Hardware . . . 1
- 2.2 Software . . . 2

3 Report Format **3**

4 Memory Module **3**

5 Registers **4**

6 The CPU **5**
- 6.1 Writing the opcodes . . . 5
- 6.2 Basic Fetch/Decode/Execute Loop . . . 6

7 Adding a unit testing framework **7**

8 Connecting the TFT display **8**

9 Measuring CPU Performance & The Display Bottleneck **9**
- 9.1 Optimizing for speed . . . 10
  - 9.1.1 First try, Debug build, 24 Mhz Clock . . . 10
  - 9.1.2 Debug build, 79 Mhz Clock . . . 10
  - 9.1.3 Release build, -O3 . . . 12
  - 9.1.4 Release build, -O3, Additional Code Inlined . . . 12
- 9.2 Display Bottleneck . . . 13

10 Implementing a Debugger **13**

11 Video Display System (GPU) **15**
- 11.1 How the Game Boy Stores Video Data . . . 15
  - 11.1.1 Tile Encoding . . . 15
  - 11.1.2 Tile Mapping . . . 16
  - 11.1.3 Scrolling . . . 16
- 11.2 Display Timing . . . 17
- 11.3 DMA Transfer . . . 19

12 Using Test ROMS **21**

13 Implementing Interrupts **22**

14 Back to the GPU: Window and Sprites **23**

15 Emulating DMA **25**

16 Adding the hardware input **26**

17 Emulating Timer Interrupts **30**

18 Tetris! And other ROMS **32**

19 Custom Boot ROM **34**

20 Conclusion **34**
20.1 Where to go from here 34

21 Appendix **34**
21.1 Code 34
21.2 Useful Game Boy Emulator Development Links 34

1 Introduction and Motivation

The goal of this project is to create an emulator for the original 1989 Game Boy system on the PSOC 5 that is able to play the original Tetris game. The 1989 Game Boy was an 8-bit handheld game console developed by Nintendo. The console sold over a million units within weeks of its initial release, and was the first in Nintendo's Game Boy series. Notable titles on the system included Tetris, Pokemon, and The Legend of Zelda.

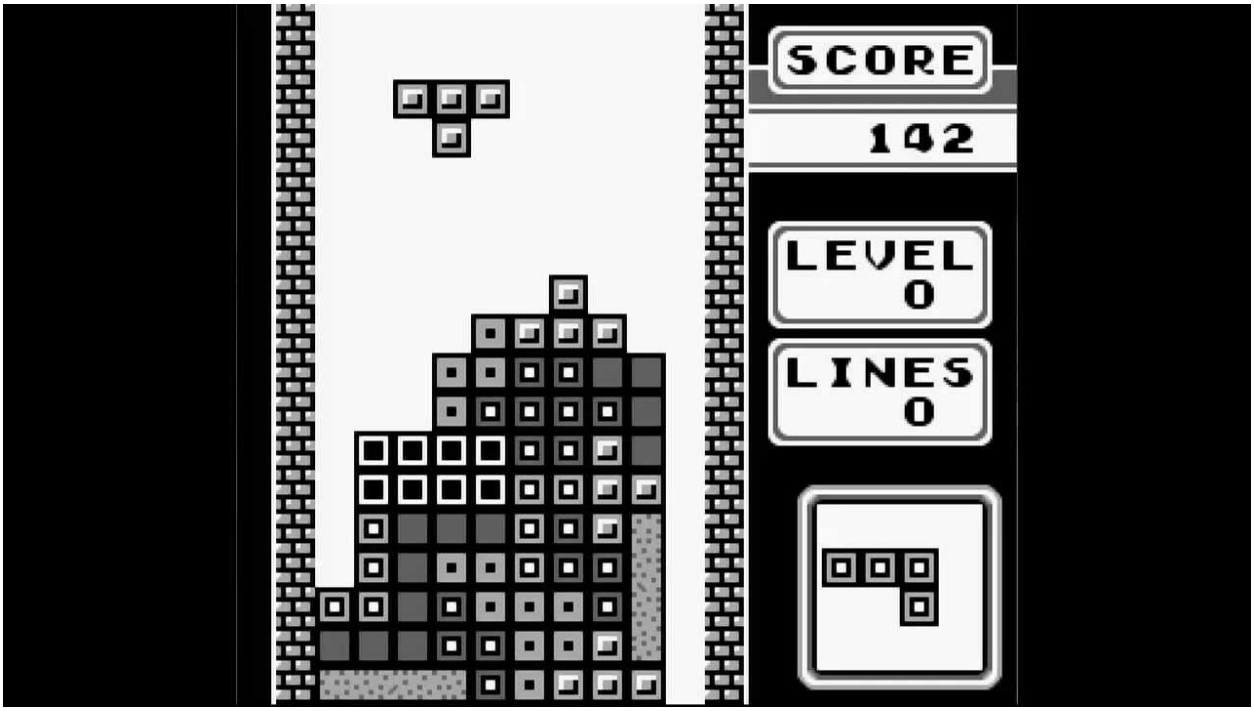


Figure 1: 1989 Tetris. The goal!

The original Game Boy ran on a Sharp LR35902 CPU, which was a hybrid chip between the Intel 8080 and the Zilog Z80. The CPU ran at 4.19 Mhz, with 4 cycles per machine cycle. It featured 8K bytes of RAM, along with 8K bytes of video RAM, with a 160x144 px display resolution.

The PSOC 5, with its ARM Cortex-M3 processor running at 79Mhz turns out to be just enough to emulate the Game Boy Classic.

I have found several emulators of the Game Boy Classic running on micro-controllers similar to the PSOC 5, but I have not found any on the PSOC 5 itself. I chose this project because it is a good opportunity to test my knowledge of assembly/computer architecture.

2 Project Overview and Layout

2.1 Hardware

The hardware for the system is straightforward:

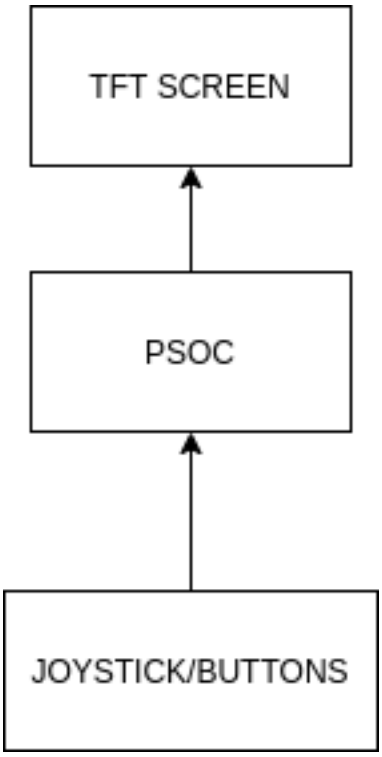


Figure 2: Hardware overview

As shown, the PSOC takes input from the joysticks and buttons, and displays the emulated Game Boy on the TFT screen.

2.2 Software

The software is where things get more interesting:

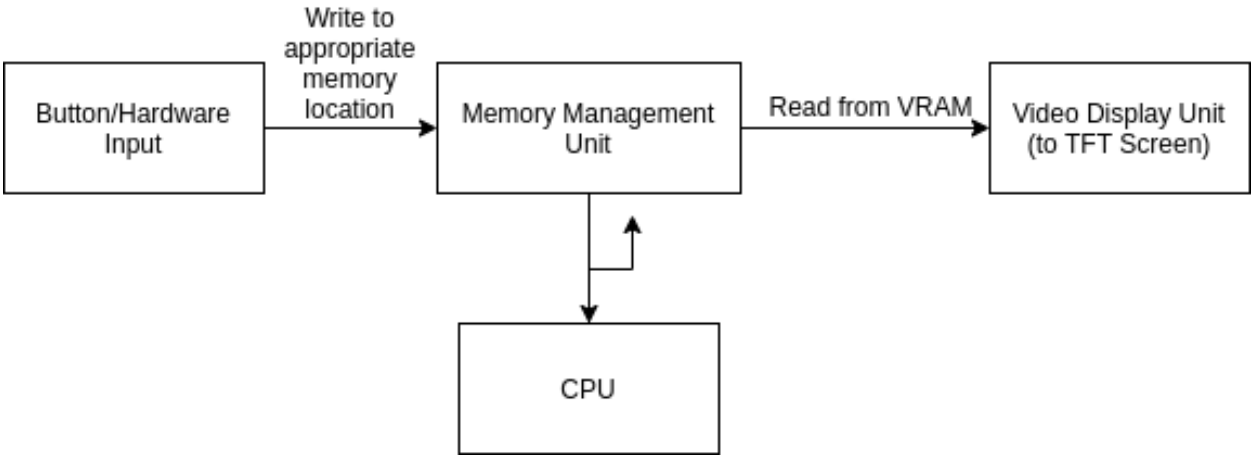


Figure 3: Software overview

As shown in the simplified diagram, the emulator simulates the memory layout of the Gameboy Classic and manipulates it by emulating the CPU. The video display unit reads from the VRAM and displays the results on the TFT screen.

3 Report Format

The order of the following sections reflect the approximate order that each module was implemented, as well as the relevant implementation details.

4 Memory Module

I started with writing the memory management unit, since a CPU is not really a useful device without memory to read from/write to.

This part of the code manages the memory access for the emulator. The original Game Boy had 16-bit addressable memory, but only 8K of actual RAM and 8K of video RAM.

Here is a picture of the memory map:

GameBoy Memory Areas

$FFFF	Interrupt Enable Flag
$FF80-$FFFE	Zero Page - 127 bytes
$FF00-$FF7F	Hardware I/O Registers
$FEA0-$FEFF	Unusable Memory
$FE00-$FE9F	OAM - Object Attribute Memory
$E000-$FDFF	Echo RAM - Reserved, Do Not Use
$D000-$DFFF	Internal RAM - Bank 1-7 (switchable - CGB only)
$C000-$CFFF	Internal RAM - Bank 0 (fixed)
$A000-$BFFF	Cartridge RAM (If Available)
$9C00-$9FFF	BG Map Data 2
$9800-$9BFF	BG Map Data 1
$8000-$97FF	Character RAM
$4000-$7FFF	Cartridge ROM - Switchable Banks 1-xx
$0150-$3FFF	Cartridge ROM - Bank 0 (fixed)
$0100-$014F	Cartridge Header Area
$0000-$00FF	Restart and Interrupt Vectors

Figure 4: Memory mapping

And here is a relevant snippet of the memory unit:

```
typedef struct Memory {
    uint8_t wram[WRAM_SIZE];          // work ram
    uint8_t eram[EXTERNAL_RAM_SIZE]; // external ram
    uint8_t vram[VRAM_SIZE];          // video ram

    uint8_t oam[OAM_SIZE];      // Sprite attribute table (OAM)
    uint8_t zero_page[ZERO_PAGE_SIZE];        // High address RAM (stack here)
    uint8_t interrupt_enable;  // interrupt enable (located on 0xFFFF)
    uint8_t interrupt_flag;    // interrupt flag (located on 0xFF0F)

    // GPU registers
    uint8_t lcdc;              // LCD Control (R/W located on 0xFF40)
    uint8_t lcdstatus;         // LCD Status  (R/Wlocated on 0xFF41)
    uint8_t scroll_y;          // SCY (R/W located on 0xFF42)
    uint8_t scroll_x;          // SCX (R/W located on 0xFF43)
```

```
    uint8_t current_scan_line; // LY  (R located on 0xFF44)
    uint8_t lyc;               // LYC (used for ly compare interrupts)
    uint8_t background_palette; // (W located on 0xFF47)
    uint8_t obp0;              // OBP0 object palette 0 (located on 0xFF48)
    uint8_t obp1;              // OBP1 object palette 1 (located on 0xFF49)
    uint8_t wx;                // window x position + 7
    uint8_t wy;                // window y position

    // Serial communication
    uint8_t sb;  //SB serial transfer data    (located on 0xFF01)
    uint8_t sc;  //SC serial transfer control (located on 0xFF02)
    // Joypad
    uint8_t joyp;              //Joypad register located on 0xFF00;

    // Timer
    uint8_t timer_divider;     // Timer divider DIV
    uint8_t timer_counter;     // Timer counter TIMA
    uint8_t timer_modulo;      // Timer Modulo TMA
    uint8_t timer_control;     // Timer Control TAC
} Memory;
```

As shown, the WRAM and VRAM are implemented as large arrays in memory. The Memory struct is also responsible for holding many registers related to graphics, serial communication, and other MMIO.

Notice, however, that this struct does not include the ROM for the Game Boy. This is due to the space constraints of the PSOC 5, which does not have enough RAM to hold an entire Game Boy ROM. It does, however, have a whopping 64Kb of flash (program) memory. This is enough to hold the ROMs. In a separate file, I declare these large constant arrays:

```
#include "rom.h"
#include "stdint.h"
const uint8_t bios[256] = {
      0x31,0xFE,0xFF,0xAF,0x21,0xFF,0x9F,0x32,0xCB,0x7C,0x20,0xFB,0x21,0x26,0xFF,0x0E,
       /// ommitted for clarity...
      0xF5,0x06,0x19,0x78,0x86,0x23,0x05,0x20,0xFB,0x86,0x20,0xFE,0x3E,0x01,0xE0,0x50};


// Original 1989 Tetris
const uint8_t rom[0x8000] = {
      0xC3,0x0C,0x02,0x00,0x00,0x00,0x00,0x00,0xC3,0x0C,0x02,0xFF,0xFF,0xFF,0xFF,0xFF,
      // 100s of lines ommitted for clarity...
      0xFF,0xFF,0xFF,0xFF,0xFF,
   }
```

These arrays are populated directly with the instructions from the ROMs. I obtained the original Tetris ROM and original boot BIOS (the ROM that displays the Nintendo Logo) from the popular emulation site https://vimm.net/. I converted them directly into C-style arrays using the xxd linux tool.

```
xxd -i -c 16 <ROM_FILE>
```

With these arrays declared const., the compiler was smart enough to write these to flash memory. The actual fetching and writing to and from these memory locations happens through the fetch() function of the Memory struct, which handles the relevant mapping from instructions to array locations in a simple if-else chain.

5 Registers

The next most important piece of code is the registers on the CPU. The Sharp LR35902 CPU had 7 main registers labeled A, B, C, D, E, F, H. The interesting part of Game Boy assembly is that these registers could also be used as 16-bit registers. For example, there is the HL register, which uses H as the higher byte and L as the lower byte.

This does pose a slightly awkward problem when it comes to programming Game Boy emulators. To implement 16-bit registers, many emulators use helper functions like the following:

```
void set_hl(int new_hl);
```

In this emulator, I took advantage of C's union type, which forces variables to share the same memory locations. My implementation of the CPU registers looks like:

```
typedef struct Registers {
    struct {
        union {
            struct {
                uint8_t f;
                uint8_t a;
            };
            uint16_t af;
        };
    };
    /*
    ..... more registers bc, de, hl
    ommitted from this paper for clarity
    */
    uint16_t pc;    // program counter
    uint16_t sp;    // stack pointer
    bool ime;       // interrupt enable
    bool ime_enable_req;  //used to delay ei by 1 instr
} Registers;
```

By using a union between the anonymous struct and the 16 bit integer af, I can write registers.a = 0x3 and be able to read registers.af as 0x30.

Also notice the inclusion of the standard program counter (pc) and stack pointer (sp) registers, as well as the interrupt enable flag. The interrupt enable flag is in this struct rather than Memory since it is not mapped to a memory location in the original Game Boy.

6 The CPU

Finally, with the registers and memory unit available, I moved on to writing the basic CPU.

The external interface for using the CPU was kept very simple:

```
typedef struct Cpu {
    Registers reg;
    Memory* mem;
    bool inBios;
} Cpu;

// Handles one round of fetch/decode/execute
// Returns the number of machine cycles taken for the instruction
// 4 clock cycles == 1 machine cycle
int tick(Cpu* cpu);
// Resets the cpu to the starting state, clearing all registers etc
void reset_cpu(Cpu *cpu);
```

As shown, the CPU has memory to manage, registers to use, and functions to execute the next instruction and to reset the CPU.

6.1 Writing the opcodes

In my instruction_set.h header file, I declared each individual operation of the Sharp LR35902. I made sure to declare each one as inline as a hint to the compiler to inline the function for performance. Here is an example, looking at the xor instruction:

```c
static inline void xor_a_b(Cpu* cpu, uint8_t b){
    cpu->reg.a ^= b;
    set_zero_flag(&cpu->reg, cpu->reg.a == 0);
    set_subtraction_flag(&cpu->reg, false);
    set_carry_flag(&cpu->reg, false);
    set_half_carry_flag(&cpu->reg, false);
}
static inline uint8_t xor_a_r8(Cpu* cpu, uint8_t* reg){
    xor_a_b(cpu, *reg);
    return 1;
}
static inline uint8_t xor_a_mhl(Cpu* cpu){
    xor_a_b(cpu, fetch(&cpu->mem, cpu->reg.hl, cpu->inBios));
    return 2;
}
static inline uint8_t xor_a_n8(Cpu* cpu){
    xor_a_b(cpu, fetch_and_increment_pc(cpu));
    return 2;
}
```

Notice how I used one function to implement each possible invocation of the xor, and that each instruction sets various flags on the flag register.

In general, most of the instructions take on this form:

```c
// Returns the result of a op b
// Used only in instruction_set.h
static inline uint8_t instruction_a_b(Cpu* cpu, uint8_t a, uint8_t b);

// Returns the number of machine cycles taken
// Given pointers to the registers
static inline uint8_t instruction_r8_r8(Cpu* cpu, uint8_t* reg1, uint8_t* reg2);

// Returns the number of machine cycles taken
// For running [register] op [value of memory @reg.hl]
static inline uint8_t instruction_r8_mhl(Cpu* cpu, uint8_t* reg)

// Returns the number of machine cycles taken
// For running [register] op [immediate value]
static inline uint8_t instruction_r8_n8(Cpu* cpu, uint8_t* reg)
```

6.2 Basic Fetch/Decode/Execute Loop

The tick() method looks like:

```c
int tick(Cpu* cpu){
    // Fetch
    uint8_t instruction = fetch_and_increment_pc(cpu);

    // Check for CB-prefixed instructions
    if (instruction == 0xCB) {
        // This is a CB-prefixed instruction!
        // Have to read the next one
        uint8_t cb_instr = fetch_and_increment_pc(cpu);
        return execute_cb_prefix(cpu, cb_instr);
    } else {
        // Regular instruction
        return execute_normal(cpu, instruction);
    }
}
```

Note the check for 0xCB-prefixed instructions. The Sharp LR35902 was able to extend its instruction set beyond just the 2^8 available numbers by prefixing some instructions with 0xCB. This means these instructions take at least 1 additional cycle to read, so they are typically uncommon instructions such as bit-shifting.

The implementation of the tick() method on the Cpu required a lot thinking between writing clear code and writing fast code. In particular the process of decoding an instruction's opcode to its relevant function can be a challenge. It is possible to do this by creating a large array of function pointers, and indexing into this array with the opcode itself, but given the space and speed constraints on the PSOC, I chose to use a large switch-case instead. execute_normal() and execute_cb_prefix() both use large switch cases like so:

```
1
2  // Assumes that the pc is already incremented to point to the next instr
3  static inline int execute_normal(Cpu* cpu, uint8_t instruction){
4      switch (instruction){
5          case 0x0: return nop(cpu); //NOP
6          case 0x1: return ld_r16_n16(cpu, &cpu->reg.bc); //LD BC,u16
7          case 0x2: return ld_mr16_a(cpu, &cpu->reg.bc); //LD (BC),A
8          case 0x3: return inc_r16(cpu, &cpu->reg.bc); //INC BC
9          case 0x4: return inc_r8(cpu, &cpu->reg.b); //INC B
10         case 0x5: return dec_r8(cpu, &cpu->reg.b); //DEC B
11         case 0x6: return ld_r8_n8(cpu, &cpu->reg.b); //LD B,u8
12         case 0x7: return rlca(cpu); //RLCA
13         case 0x8: return ld_mn16_sp(cpu); //LD (u16),SP
14         case 0x9: return add_hl_r16(cpu, &cpu->reg.bc); //ADD HL,BC
15         case 0xa: return ld_a_mr16(cpu, &cpu->reg.bc); //LD A,(BC)
16         case 0xb: return dec_r16(cpu, &cpu->reg.bc); //DEC BC
17         case 0xc: return inc_r8(cpu, &cpu->reg.c); //INC C
18         case 0xd: return dec_r8(cpu, &cpu->reg.c); //DEC C
19         case 0xe: return ld_r8_n8(cpu, &cpu->reg.c); //LD C,u8
20         case 0xf: return rrca(cpu); //RRCA
21         case 0x10: return stop(cpu);  //STOP
22         case 0x11: return ld_r16_n16(cpu, &cpu->reg.de); //LD DE,u16
23         case 0x12: return ld_mr16_a(cpu, &cpu->reg.de); //LD (DE),A
24         case 0x13: return inc_r16(cpu, &cpu->reg.de); //INC DE
25         /*
26         ----------------------------------------
27          and again and again until we reach 0xFF
28         ----------------------------------------
29         */
```

The advantage of using large-switch cases over large if-else chains is that the compiler is able to optimize these switch-cases into look-up tables. After fiddling with the build settings in PSOC Creator to make sure optimizations were on, I was able to look at the generated assembly to confirm that this was the case:

```
1  276:cpu.c          ****      switch (instruction){
2  539                      .loc 3 276 0
3  540 0004 0446            mov   r4, r0
4  541                      .loc 3 277 0
5  542 0006 FE29            cmp   r1, #254
6  543 0008 01F25186        bhi   .L29
7  544 000c DFE811F0        tbh   [pc, r1, lsl #1]
```

The last instruction is a tbh, which according to the ARM assembly manual, stands for Table Branch Halfword. Our PSOC has 32-bit words, so a half word is 16 bits, which is more than enough for the number of opcodes that the Game Boy has.

7 Adding a unit testing framework

What's a good software project without unit tests? After writing so much code already, I was almost certain I had some latent bugs that I had to fix. PSOC Creator does not appear to have an easy way to add unit tests by default, but I was able to add a testing framework called Ceedling. I followed the instructions outlined in this post.

With Ceedling, I was able to write simple unit tests that helped me catch bugs in my code. Here is an example of a unit test, testing the hl 16-bit register.

```
void test_hl(void){
   Registers regs;
   regs.h = 0x03;
   regs.l = 0x05;
   TEST_ASSERT_EQUAL_HEX16(0x0305,regs.hl);
   regs.hl = 0x0123;
   TEST_ASSERT_EQUAL_HEX16(0x01, regs.h);
   TEST_ASSERT_EQUAL_HEX16(0x23, regs.l);
}
```

Given the short amount of time and the massive amount of code in the project, I wasn't able to get to full unit test coverage, but they were still a useful tool for the code that I did cover.

8 Connecting the TFT display

It was time to get the display connected. Following the instructions at http://web.mit.edu/6.115/www/document/TFT_User_Manual.pdf, I was able to connect the TFT screen and use the emWin library to draw text.

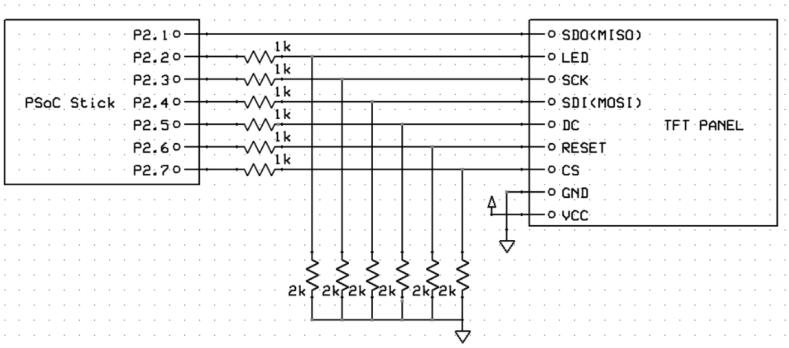


Figure 5: TFT Screen Schematic

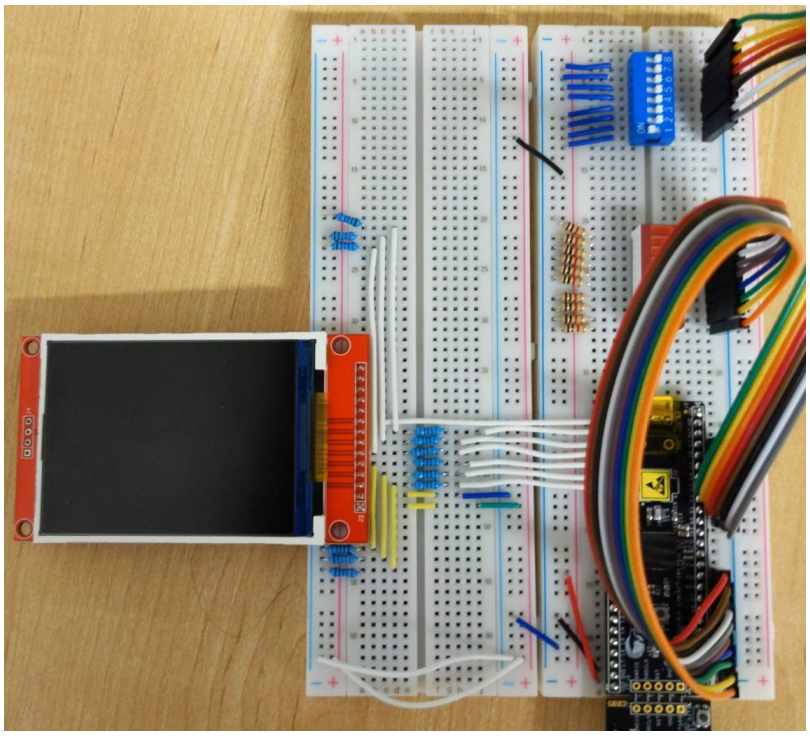


Figure 6: Initial hardware screen connection. Ignore the buttons/resistors on the left side; those are remains from a previous assignment.

9 Measuring CPU Performance & The Display Bottleneck

With the TFT screen connected, I decided now was a good time to evaluate the performance of the CPU to ensure that it was feasible to emulate the Game Boy at all. I modified main.c to start the fetch/decode/execute loop on the BIOS and time the result.

I added this timer:

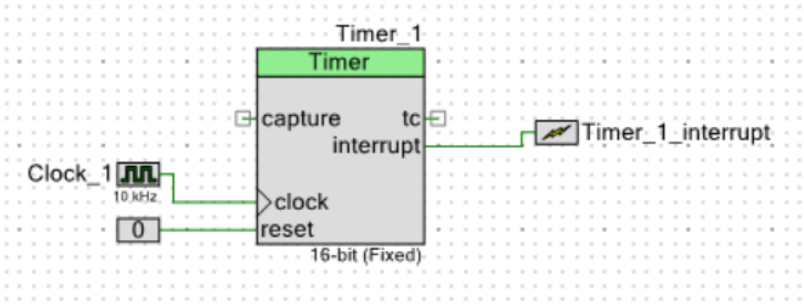


Figure 7: Timer used for measuring CPU performance

Which would generate an interrupt at a certain frequency. In the interrupt handler, I displayed the relevant speed information on the TFT screen.

```
CY_ISR(Timer_1_Handler){
    sprintf(buffer, "On-time (sec): %d  \n"
        "Instrs/second: %lu \n"
        "Cycles/second: %lu \n"
```

```
        "Machine Cycles/second:\n %lu ", (int) seconds, total_instrs*60/*/4*/, ↩
            total_cycles, total_cycles*60/*/4*/);
    total_cycles = 0;
    total_instrs = 0;

    GUI_DispStringAt(buffer, 0, 0);
    seconds += 0.01666;
    Timer_1_ReadStatusRegister(); //Clear timer register to leave interrupt
}
```

This turned out to be very valuable when it came to optimizing my code for speed. Here is the progression, with stats measured at 4-second intervals. Pay close attention to the "Machine cycles/second" metric:

9.1 Optimizing for speed

9.1.1 First try, Debug build, 24 Mhz Clock

Figure 8: First try. Not so fast.

As shown, this only ran at 185K machine cycles/second. This was rather disappointing, considering the original game boy ran at 1M machine cycles/second.

9.1.2 Debug build, 79 Mhz Clock

I modified the PSOC's default 24 Mhz clock to instead run at the maximium 79 Mhz through PSOC Creator. This was as simple as changing the desired clock speed in PSOC creator to 79 Mhz:

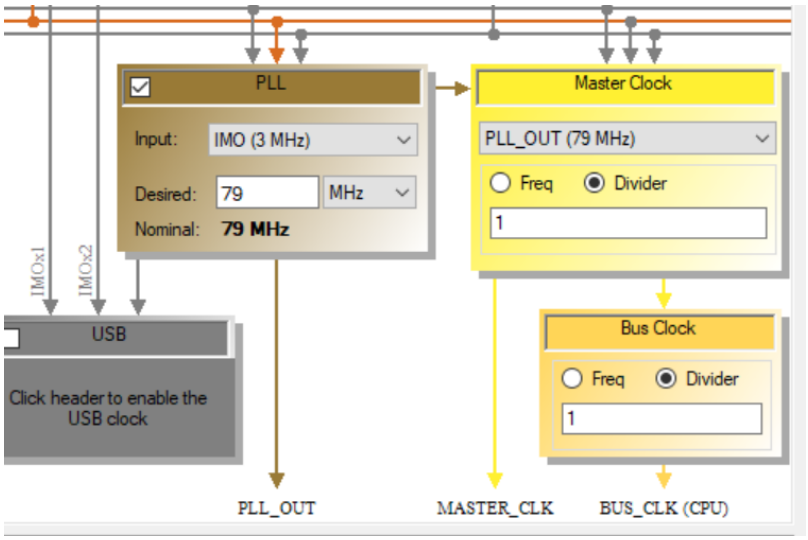


Figure 9: PSOC 5 system clock adjustment GUI

This results in a 3x speedup, as you would expect. Props to Cypress for making this so easy!

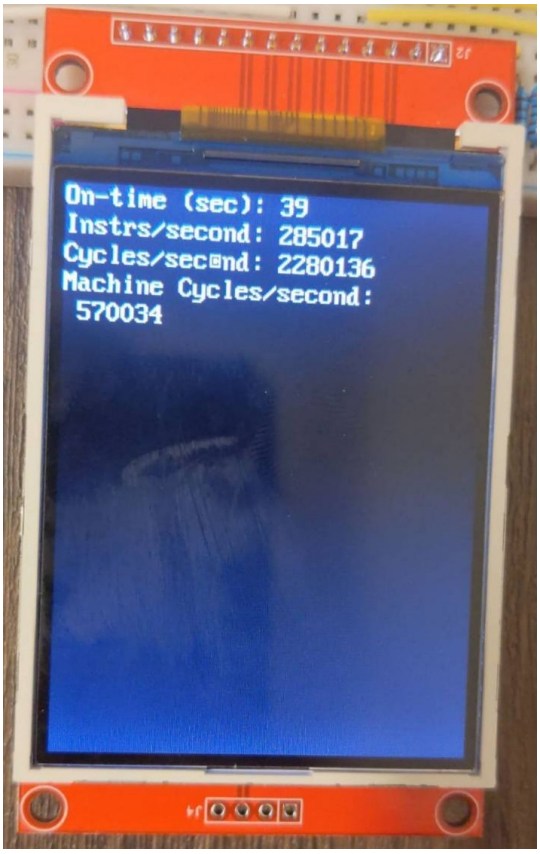


Figure 10: Quite a bit faster!

9.1.3 Release build, -O3

From there, I updated my build settings to run in release mode, optimizing for speed (-O3) rather than PSOC Creator's default (-Os).

Figure 11: Now that's some speed!

Now that's quite the improvement! Still felt like I could do even better, though.

9.1.4 Release build, -O3, Additional Code Inlined

Lastly, I was able to squeeze out 3 K more machine cycles/second by going through my existing code and inlining additional functions.

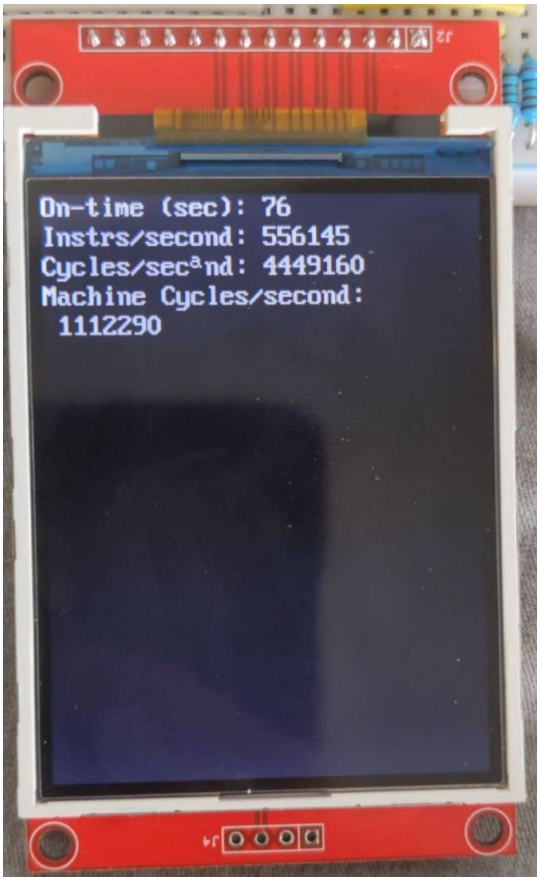


Figure 12: Now we're gaming!

I was satisfied here, considering the original Game Boy ran at 1M machine cycles/second.

9.2 Display Bottleneck

With my code optimized for speed, I started changing the update frequency of the display.

From these stats, I noticed that writing to the TFT screen using the emWin library appeared to be a big bottleneck in performance. In particular, writing to the display once every 4 seconds runs at around 1.2M machine cycles/second, but writing to the display 4x a second results in only around 700 machine cycles/second. I chose to address this later.

10 Implementing a Debugger

Next I added a basic debugging interface for the CPU.

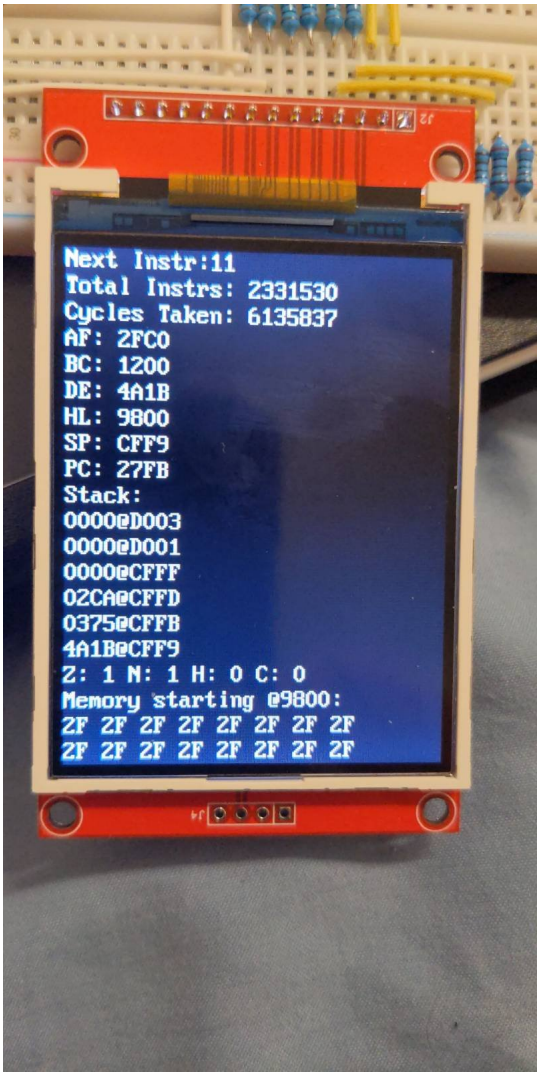


Figure 13: Debug interface. Shows registers, stack, flags, and even memory starting at any arbitrary location. Essential!

I added a button interrupt to step the CPU and refresh the display. Additionally, I added a UART component to communicate with my computer over serial. With this debugger, I decided it was time to debug the boot ROM. The Game Boy boot ROM is available here: https://gbdev.gg8.se/wiki/articles/Gameboy_Bootstrap_ROM

I ran through the Game Boy's boot ROM and was able to catch a few bugs in my implementation by comparing the registers/stack of my emulator to the well known Game Boy emulator bgb http://bgb.bircd.org/

Eventually, I made it through the execution and encountered what appeared to be an infinite loop. Inspecting the BIOS code, we notice the following:

```
Addr_0064:
   LD A,($FF00+$44)   ; $0064  wait for screen frame
   CP $90             ; $0066
   JR NZ, Addr_0064   ; $0068
```

This code waits for a new frame to start, indicated when the value at location 0xFF44 == 0x90. Without a video display system, however, this would never happen. That meant it was time to implement the video

display!

11 Video Display System (GPU)

The original Game Boy's video display system worked similarly to a VGA display. Another chip on the system was responsible for reading from the VRAM and displaying scan lines onto the LCD.

11.1 How the Game Boy Stores Video Data

The Game Boy utilized a tile mapping system to reduce the amount of memory required for displaying games.

11.1.1 Tile Encoding

Each tile is an 8x8 px square. Tile data is stored in VRAM at locations 0x8000-0x97FF. Instead of storing colors directly, pixels are encoded as color IDs ranging from 0-3. This is sometimes called 2bpp (2-bit per pixel) encoding. Each tile requires 16 bytes of data, 2 bytes per row, for a total of 384 tiles.

For example, here is a single tile in Pokémon Red/Blue Version (a window from a house):

```
0x8000:
[FF 00 7E FF 85 81 89 83 93 85 A5 8B C9 97 7E FF]
```

Consider just the first row, which requires 2 bytes: 0xFF and 0x00. The first byte is the low byte, and the second byte is the high byte. The first pixel of the first row has its color ID given by

```
two bits {MSB high byte, MSB low byte}
low  = 0xFF = 11111111
high = 0x00 = 00000000
MSB high    = 1
MSG low     = 0
           => 01
```

So the first pixel has a color ID of 1. If you repeat this process for each pixel across all rows, you end up with the following tile:

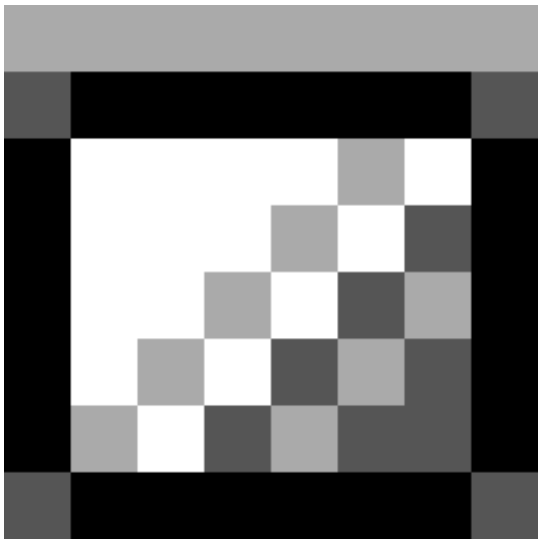


Figure 14: Example tile. Notice the first pixel is a light grey–corresponds to color ID of 1.

Here is a graphic that displays this more concretely:

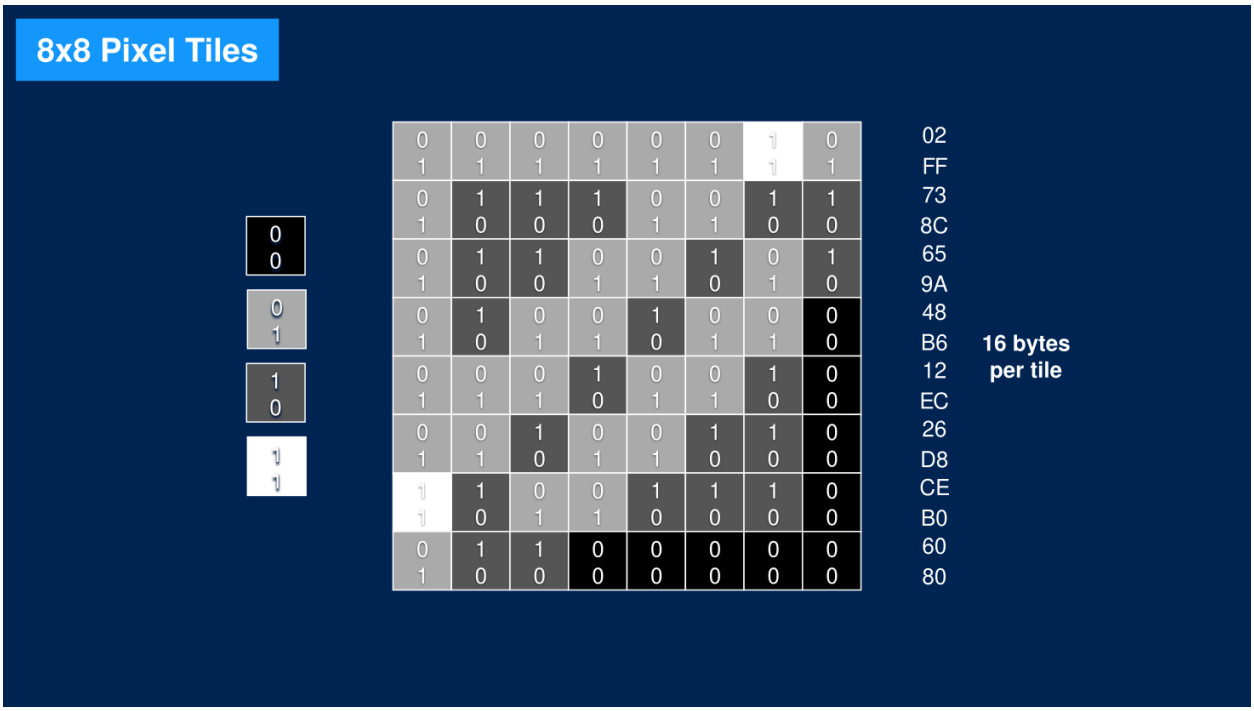


Figure 15: Concrete tile encoding summary

11.1.2 Tile Mapping

There are two 32x32 tile maps in VRAM at addresses 0x9800-0x9BFF and 0xC00-0x9FFF. Each entry at these map locations acts as a tileID (simple offset) into the tile data described above.

11.1.3 Scrolling

Given that each tile map is 32x32 tiles, and each tile is 8x8 px, the Game Boy has enough data for a 256x256 sized image. Since the display is only 160x144 px, the extra space is used to implement scrolling windows. There are two registers SCY and SCX (scrollX and scrollY) that control the start of the display section.

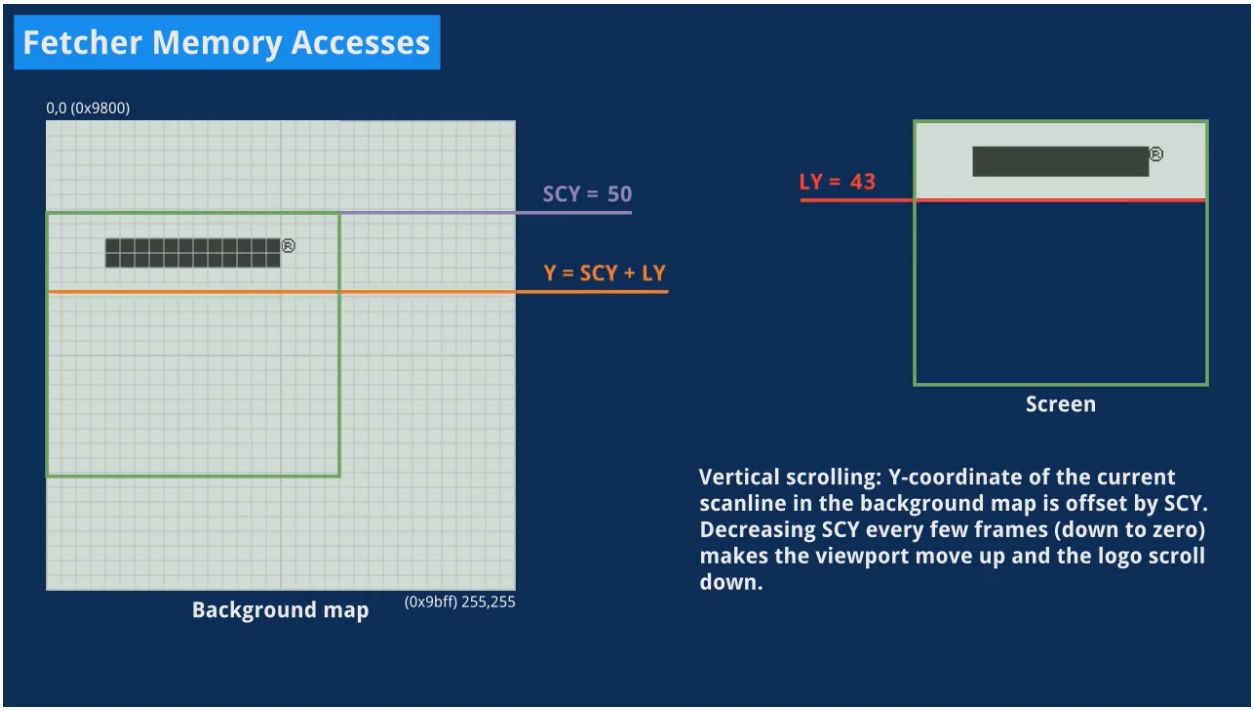


Figure 16: Scrolling on the Game Boy. The logo in the boot ROM isn't moving down; the entire window is moving up!

Pixels that exceed the bounds of the display wrap around.

11.2 Display Timing

Again, since the original Game Boy's pixel processing unit (PPU) ran on its own clock, the timing between the PPU and CPU is precisely defined as follows:

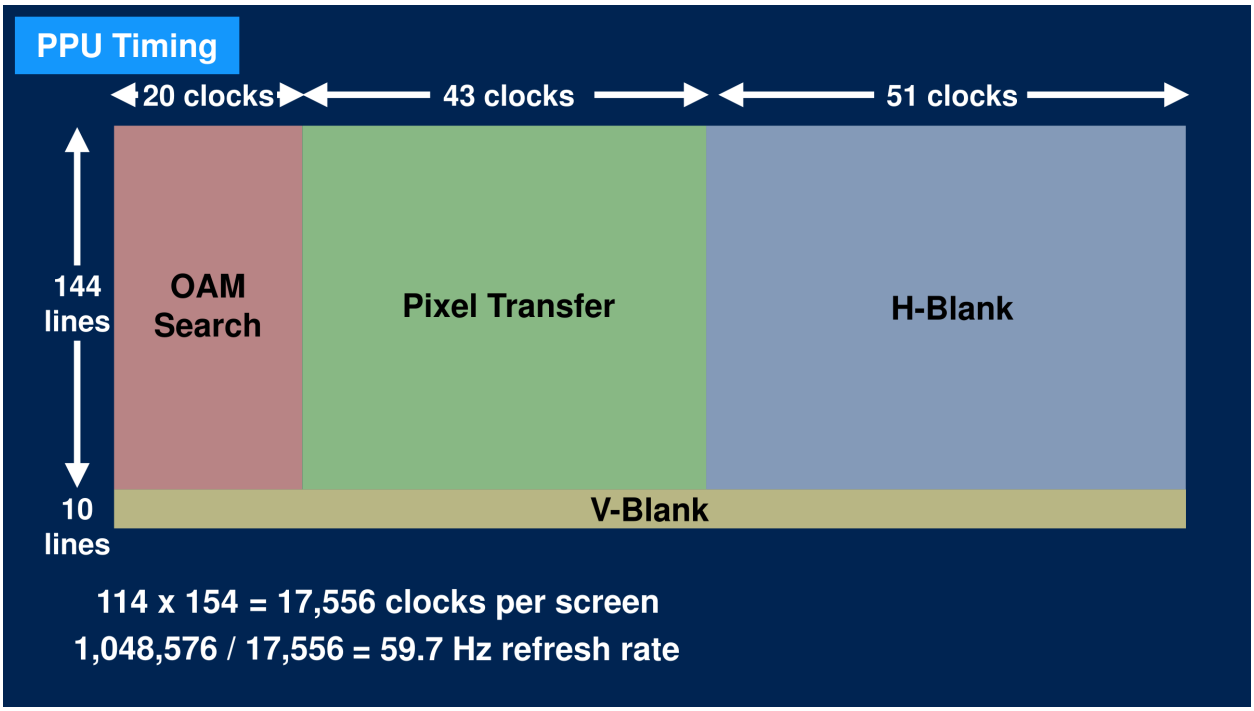


Figure 17: Timing diagram for the PPU

As shown in the diagram, each scanline runs through the OAM-SEARCH, Pixel Transfer, and H-Blank stages. OAM-Search stands for Object Attribute Memory search; this is where the PPU is collecting data about sprites. Pixel transfer is the actual communication time. During this time access to VRAM by the CPU is blocked to prevent memory corruption. At the end of rendering 144 lines, the PPU emits a V-Blank. The behavior is summarized here. Note that the LY register holds the current scan line.

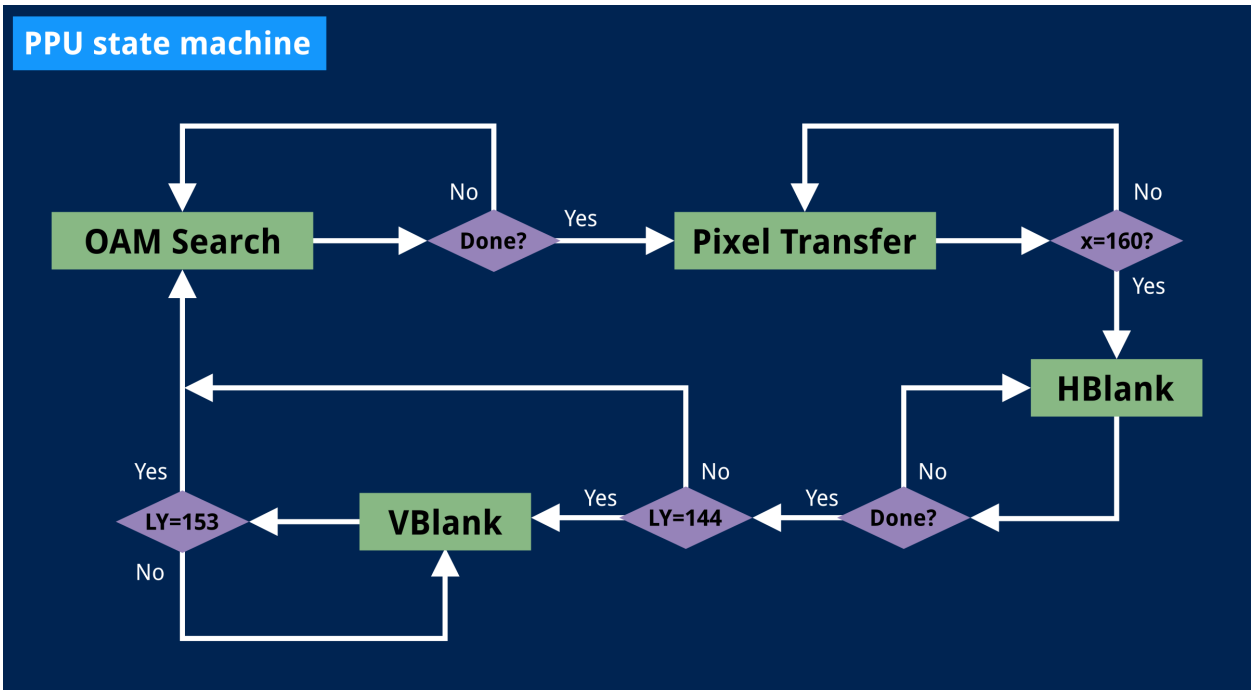


Figure 18: PPU state machine

In my emulator, this is implemented using a simple struct Gpu, which maintains a mode clock that is incremented each time the CPU runs an instruction.

```
uint8_t original_mode = gpu->mode;
gpu->mode_clock += delta_machine_cycles;

switch (original_mode) {
    // OAM Read, scanline
    case OAM_MODE:
    if (gpu->mode_clock >= OAM_READ_TIME_MACHINE_CYCLES) {
        // mode switch to VRAM read/pixel transfer (mode 3)
        gpu->mode_clock = 0;
        gpu->mode = PIXEL_TRANSFER_MODE;
    }
    // VRAM Read, scanline active
    case PIXEL_TRANSFER_MODE:
    if (gpu->mode_clock >= VRAM_READ_TIME_MACHINE_CYCLES){
        // mode switch to HBlank
        gpu->mode_clock = 0;
        gpu->mode = HBLANK_MODE;

        //Draw a full line
        if (!DEBUG_MODE)
            renderLine(mem);
    }
```

Notice that instead of rendering pixel-by-pixel like a true PPU would, I chose to render line-by-line for simplicity. Rendering is done in the renderLine() function. The renderLine() function itself is fairly simple (simply selects the relevant memory info to display), but implementing it was difficult due to the various indexing tricks required. See the implementation in the code for more details.

11.3 DMA Transfer

To address the display bottleneck described in section 9.2, I utilized the DMA (Direct Memory Access) function of the PSOC 5.

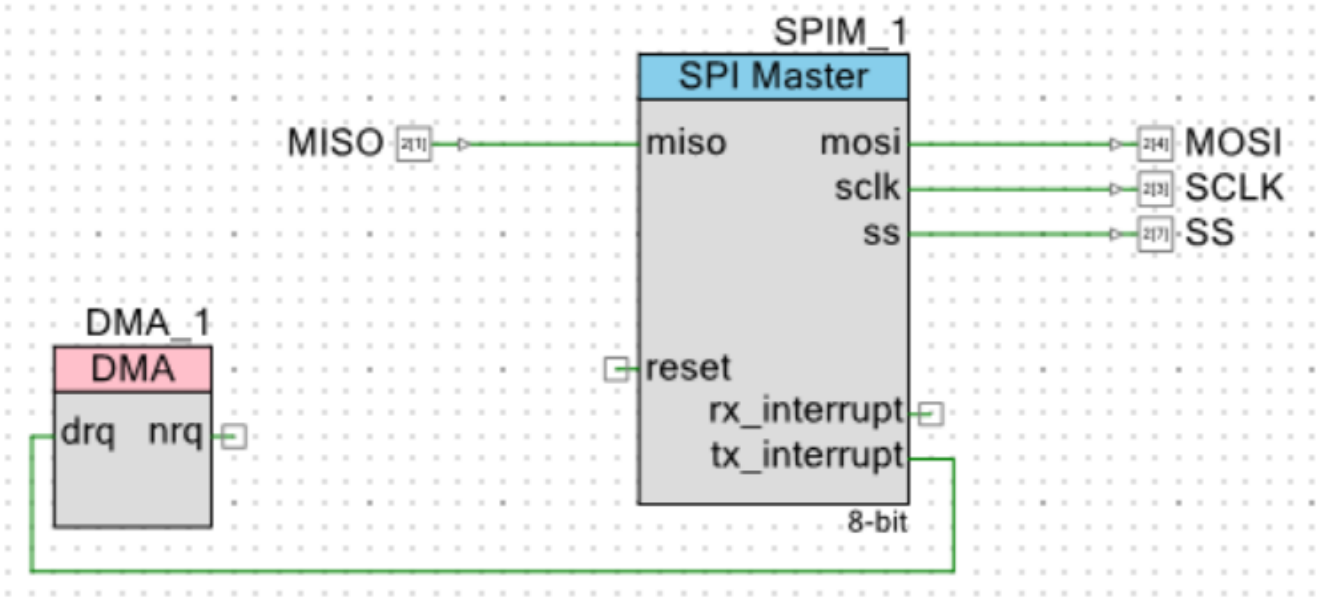


Figure 19: PSOC 5 DMA component + SPI. Notice the interrupt line leaving the SPI Master.

The SPI Master component is initialized with a 4 byte tx buffer. It is set to emit a tx_interrupt every

time the SPI buffer is empty. This initiates a 1 byte DMA burst transfer, which transfers data from a buffer in the PSOC RAM to the SPI buffer. Interrupts are fired continuously from the SPI Master until the DMA transaction is complete (when the full buffer length has been sent). After the tx transaction is complete, another transaction is chained immediately to transfer a "disable tx_interrupt" byte to the SPI Master interrupt status mask.

New data transmissions are initialized by enabling the SPI Master's tx_interrupt.

```
    void setupDma(uint8_t* dma_buff, uint32_t burstLength){
     /* Disable the TX interrupt of SPIM */
    SPIM_1_TX_STATUS_MASK_REG&=(~SPIM_1_INT_ON_TX_EMPTY);

    /* Take a copy of SPIM_TX_STATUS_MASK_REG which will be used to disable the TX interrupt using DMA */
    InterruptControl=SPIM_1_TX_STATUS_MASK_REG;

   //Init DMA, 1 byte bursts, each burst requires a request
    txChannel = DMA_1_DmaInitialize(DMA_TX_BYTES_PER_BURST, DMA_TX_REQUEST_PER_BURST, HI16(((uint32)&dma_buff[0])), HI16(((uint32)SPIM_1_TXDATA_PTR)));

    //Allocate TD to transfer x bytes
    txTD = CyDmaTdAllocate();

    //Allocate TD to disable the SPI Master TX interrupt
    InterruptControlTD = CyDmaTdAllocate();

    // txTD = From the memory to the SPIM
    CyDmaTdSetAddress(txTD, LO16(((uint32)&dma_buff[0])), LO16(((uint32) SPIM_1_TXDATA_PTR)));

    // Set the source address as variable 'InterruptControl' which stores the value 0 to disable the SPI_INT_ON_TX_EMPTY
   // and the destination is Control_Reg_SPIM_ctrl_reg__CONTROL_REG
    CyDmaTdSetAddress(InterruptControlTD, LO16((uint32)&InterruptControl), LO16((uint32)&SPIM_1_TX_STATUS_MASK_REG));

    // Set TD_tx transfer count as "burstLength" to transfer the data packet
    // Next Td as InterruptControlTD, and auto increment source address after each transaction
    CyDmaTdSetConfiguration(txTD,burstLength,InterruptControlTD, TD_INC_SRC_ADR );

    // Set InterruptControlTD with transfer count 1, next TD as txTD
    // Also enable the Terminal Output . This can be used to monitor whether transfer is complete
   CyDmaTdSetConfiguration(InterruptControlTD,1,txTD, 0 );

    // Terminate the chain of TDs; this clears any pending request to the DMA
    CyDmaChSetRequest(txChannel, CPU_TERM_CHAIN);
    CyDmaChEnable(txChannel,1);

    // Set TD_tx as the initial TD associated with channel_tx
    CyDmaChSetInitialTd(txChannel, txTD);

    // Enable the DMA channel - channel_tx
    CyDmaChEnable(txChannel,1);
}
```

Note that it is important to not write into the dma buffer at the same time that the DMA transaction is occurring. The state of the DMA transfer can be checked by looking at the SPI Master's interrupt status register. This behavior is abstracted away in the following interface:

```
// This sets up a DMA chain that transfers data in dma_buff over SPI,
// then clears the SPI Interrupt on Empty Flag
// To start a "new" DMA transfer use startDMATransfer()
// burstLength specifies how many bytes to send in one transaction
```

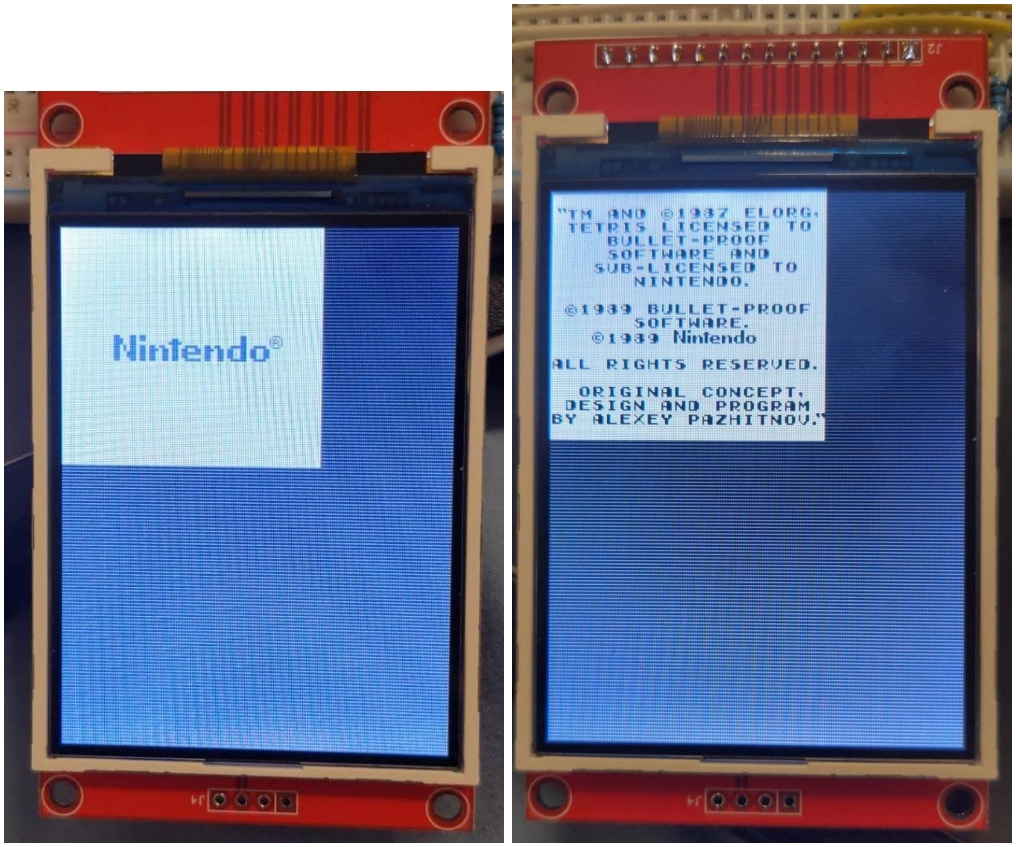
```
void setup_dma(uint8_t* dma_buff, uint32_t burstLength);

// Returns true if DMA is ready for another round
bool is_dma_ready(void);
// Starts a new DMA transfer
void start_dma_transfer(void);
```

The renderLine() function writes into the dma buffer and initiates a new transfer request at the end start of every HBlank mode.

This results in a significant speedup in the emulator's speed; visually there appeared to be at least a 2x speedup.

With the above components implemented, I was able to display the Nintendo Logo as well as the Tetris copyright screen:



(a) Nintendo Logo! (b) The Tetris Copyright screen!

Figure 20: Starting to get there!

12 Using Test ROMS

With my basic CPU and GPU system more or less complete, I then tested my CPU instructions with test ROMS from https://github.com/c-sp/gameboy-test-roms.

To do this, I first had to make a better way of switching ROMs. Due to the limited amount of memory on the system, each ROM is declared as a constant byte array. To facilitate easier ROM switching, I added the following precompiler if-else chain to my rom.c file:

```
#if ROM == TETRIS
const uint8_t rom[] = {...}
```

```
#elif ROM == TEST_ROM_CPU_INSTRS_1_SPECIAL
const uint8_t rom[] = {...}
#elif ROM == TEST_ROM_CPU_INSTRS_3_OP_SP_HL
const uint8_t rom[] = {...}
//.... more ROMS
```

Then, I created a new header file emumode.h, where I declared the following:

```
/*
This file contains precompiler definitions that change the function of the emulator
*/
#ifndef EMU_MODE_H
#define EMU_MODE_H
#define DEBUG_MODE true
#define DEBUG_TRACE_THROUGH_SERIAL false
#define DEBUG_TRACE_THROUGH_SERIAL_BREAKPOINT 0x0100    // where to start serial trace
#define DEBUG_BREAKPOINT_ON true
#define DEBUG_BREAKPOINT 0x27FB
#define DEBUG_MEMORY_DISPLAY_LOC 0x9800
#define DEBUG_SHOW_VRAM_ON_BUTTON false

// ROM list
#define TETRIS 0
#define TEST_ROM_CPU_INSTRS_1_SPECIAL 1
#define TEST_ROM_CPU_INSTRS_3_OP_SP_HL 3
#define TEST_ROM_CPU_INSTRS_4_OP_R_IMM 4
#define TEST_ROM_CPU_INSTRS_5_OP_RP 5
#define TEST_ROM_CPU_INSTRS_6_LD_R_R 6
#define TEST_ROM_CPU_INSTRS_7_JR_CALL_RET_RST 7
#define TEST_ROM_CPU_INSTRS_8_MISC_INSTRS 8
#define TEST_ROM_CPU_INSTRS_9_OP_R_R 9
#define TEST_ROM_CPU_INSTRS_10_BIT_OPS 10
#define TEST_ROM_CPU_INSTRS_11_OP_A_MHL 11
// ROM selection
#define ROM TETRIS

#define START_IN_BIOS true
#endif
```

This allowed me to very quickly change ROMs between compilation by setting the ROM definition to the target ROM. I also placed other settings here, such as whether or not to enable the debugger.

After running through the test ROMS and performing extensive debugging, my emulator slowly passed all tests.

13 Implementing Interrupts

Now it was time to implement interrupts. There are five different types of interrupts on the Game Boy:

Interrupt	ISR Location
Vertical blank	0x0040
LCD status triggers	0x0048
Timer overflow	0x0050
Serial link	0x0058
Joypad press	0x0060

The interrupt enable register at location 0xFFFF controls which interrupts are on.

Bit	When 0	When 1
0	Vblank off	Vblank on
1	LCD stat off	LCD stat on
2	Timer off	Timer on
3	Serial off	Serial on
4	Joypad off	Joypad on

At location 0xFF0F is the interrupt flag, where corresponding bits are set (using the same scheme above) when an interrupt is requested. Interrupts are fired when they are both requested and when they are enabled for that specific interrupt.

There is also a separate interrupt master enable (IME), which enables/disables all interrupts. The IME is not mapped to a memory location; it can only be toggled with the ei (enable interrupts) and di (disable interrupts) instructions. In my emulator, this is implemented in the Registers struct described earlier.

Implementing the interrupt system was fairly straightforward. I added this section to my CPU tick() method:

```
    int interrupt_enable = cpu->mem->interrupt_enable;
    int interrupt_flag = cpu->mem->interrupt_flag;
    int active_interrupts = interrupt_enable & interrupt_flag;
    if (cpu->reg.ime && active_interrupts){
        // disable interrupts (until a reti can re-enable)
        cpu->reg.ime = false;
        cycles_taken += 5;   // takes an additional 5 cycles to service interrupt
        // service the interrupt
        if (active_interrupts & VBLANK_INTERRUPT_REG_MASK) {
            rst_vec(cpu, VBLANK_ISR_LOC);
            cpu->mem->interrupt_flag &= ~VBLANK_INTERRUPT_REG_MASK;
        } else if (active_interrupts & LCD_STAT_INTERRUPT_REG_MASK) {
            rst_vec(cpu, LCD_STAT_ISR_LOC);
            cpu->mem->interrupt_flag &= ~LCD_STAT_INTERRUPT_REG_MASK;
        } else if (active_interrupts & TIMER_INTERRUPT_REG_MASK) {
            rst_vec(cpu, TIMER_ISR_LOC);
            cpu->mem->interrupt_flag &= ~TIMER_INTERRUPT_REG_MASK;
        } else if (active_interrupts & SERIAL_INTERRUPT_REG_MASK) {
            rst_vec(cpu, SERIAL_ISR_LOC);
            cpu->mem->interrupt_flag &= ~SERIAL_INTERRUPT_REG_MASK;
        } else if (active_interrupts & JOYPAD_INTERRUPT_REG_MASK) {
            rst_vec(cpu, JOYPAD_ISR_LOC);
            cpu->mem->interrupt_flag &= ~JOYPAD_INTERRUPT_REG_MASK;
        }
    }
```

And at the start of every VBLANK period in my GPU.c, I added the following:

```
// request interrupt
mem->interrupt_flag |= VBLANK_INTERRUPT_REG_MASK;
```

And with that, my emulator was able to get to the home screen of Tetris!

14 Back to the GPU: Window and Sprites

In addition to the background, the Game Boy also had the ability to render a "window", which is another tileset on top of the background. It was usually used to display static images on top of moving backgrounds, such as a HUD display side scrolling games.

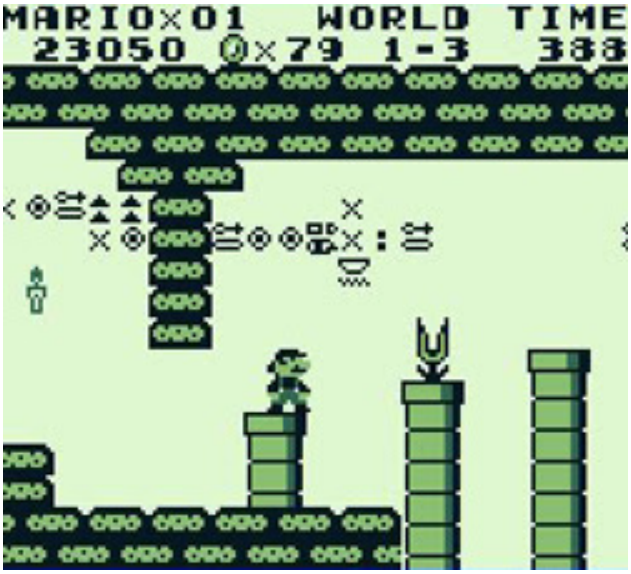


Figure 21: Super Mario Land (1989). The top HUD bar was rendered with the Game Boy's window, allowing the background to move while keeping the HUD static on the screen. Mario himself was a sprite.

The Game Boy also had sprites. Sprites allowed developers to render graphics that were not tile-aligned. The Game Boy stored information about up to 40 sprites at once in the OAM (Object Attribute Memory) located at 0xFE00 - 0xFEA0.

Each sprite entry in OAM consists of 4 bytes. The first specifies the y-position of the sprite. The second specifies the x-position. The third specifies the tile index for the sprite in the background map. And the fourth byte contains boolean flags for certain controllable sprite properties.

I added sprite functionality in my emulator in a separate function render_sprite_on_scanline(). Its implementation is very similar to the rendering of background tiles and window tiles, though it has slightly more complexity to account for edge cases such as overlapping sprites and the option of rendering 8x16 sized sprites.

Here was another case where it was very useful to find pre-made test ROMS. I used the very popular dmg-acid2 test found here https://github.com/mattcurrie/dmg-acid2, which tests all the basic functions of the GPU. With a successful renderer, the ROM displays a smiling image on the screen:

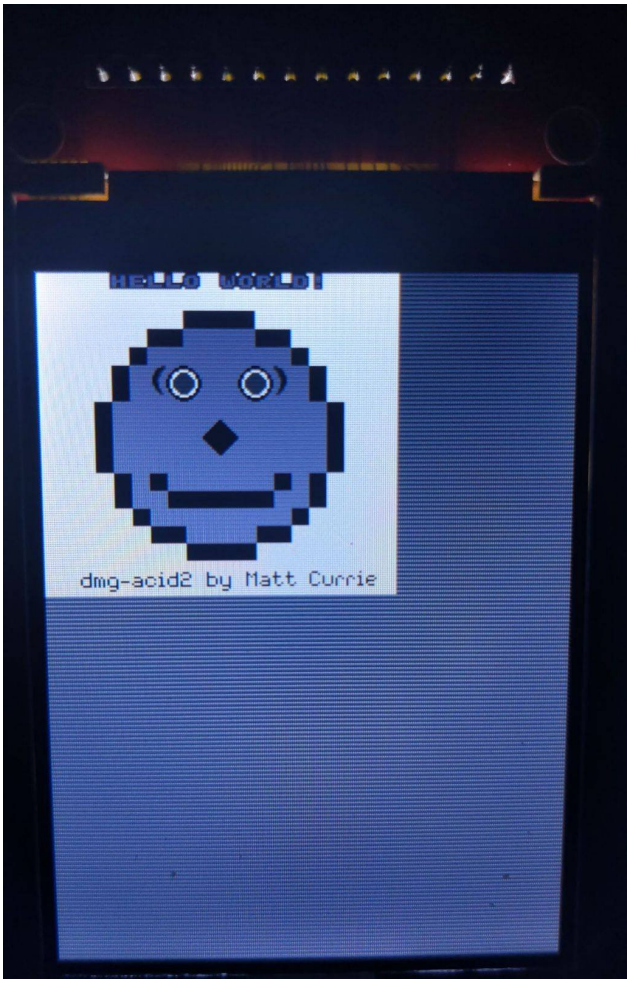


Figure 22: A happy friend

15 Emulating DMA

Much like the PSOC 5, the Game Boy had a DMA controller that allowed it to copy data to the OAM very quickly. Writing the value XX to 0xFF46 starts a DMA transfer of the following form:

```
Source:      0xXX00-0xXX9F   ;XX = 0x00 to 0xDF
Destination: 0xFE00-0xFE9F
```

For now, I implemented this directly in my memory struct, copying bytes straight away. This technically is incorrect, since the DMA transfer takes time to complete, but it is close enough to run Tetris.

```
// Emulates a dma transfer
static void start_dma(Memory* mem, uint8_t xx){
    // Source:      $XX00-$XX9F   ;XX = $00 to $DF
    // Destination: $FE00-$FE9F
    uint16_t source = xx << 8;
    int i;  // copy 160 bytes
    for (i=0;i<160;i++){
        mem->oam[i] = fetch(mem, source + i, false);
    }
}
...
```

```
void write_mem(Memory* memory, uint16_t address, uint8_t data) {
    switch (address) {
        ....
       case OAM_DMA_LOC:
                start_dma(memory, data);
                break;
        ....
```

16 Adding the hardware input

Now it was time to actually add the joystick and buttons on my board.

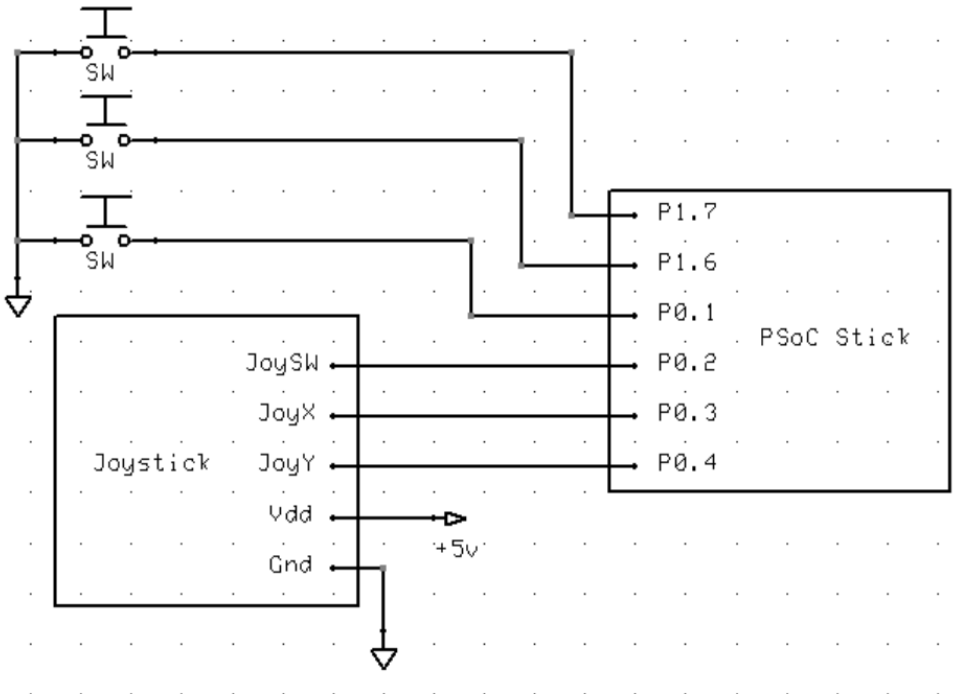


Figure 23: Schematic for buttons/switches.

I redid some of the wiring to be more compact. Notice that this new configuration uses one breadboard instead of two:

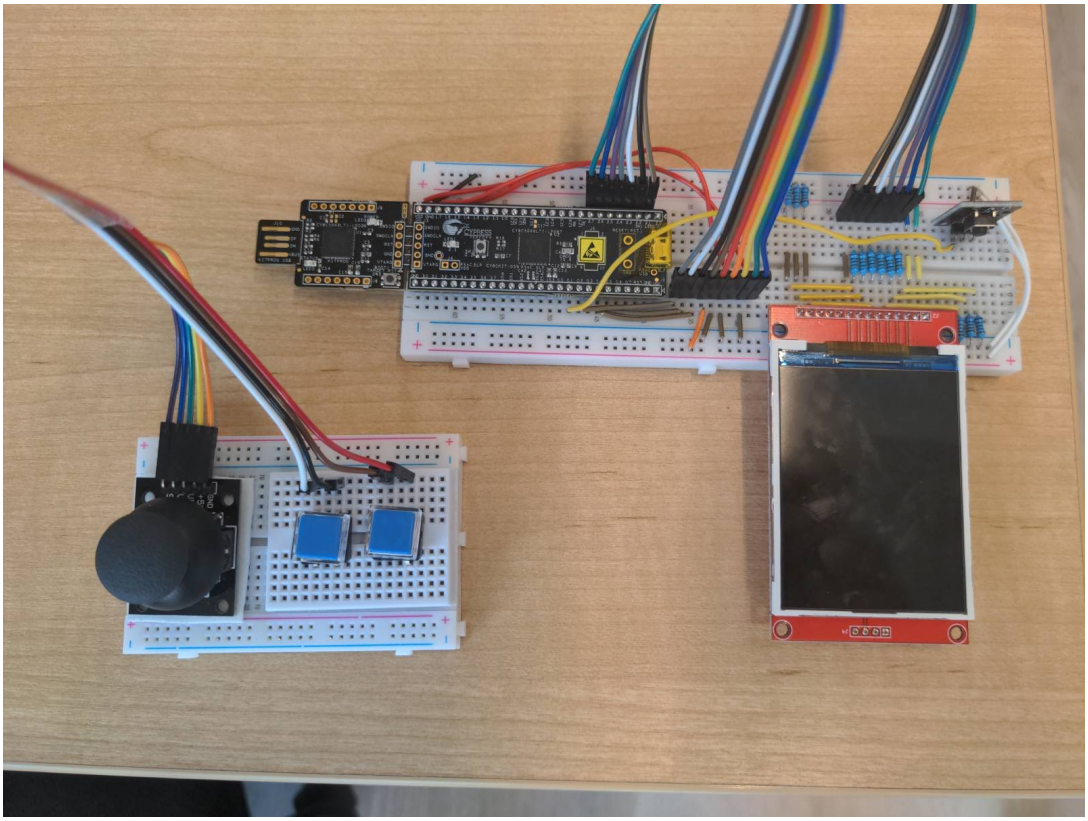


Figure 24: Final hardware layout

On the left you can see the small "controller" that I made out of leftover breadboard and sticky tape. It connects to the main board over the jumper wires.

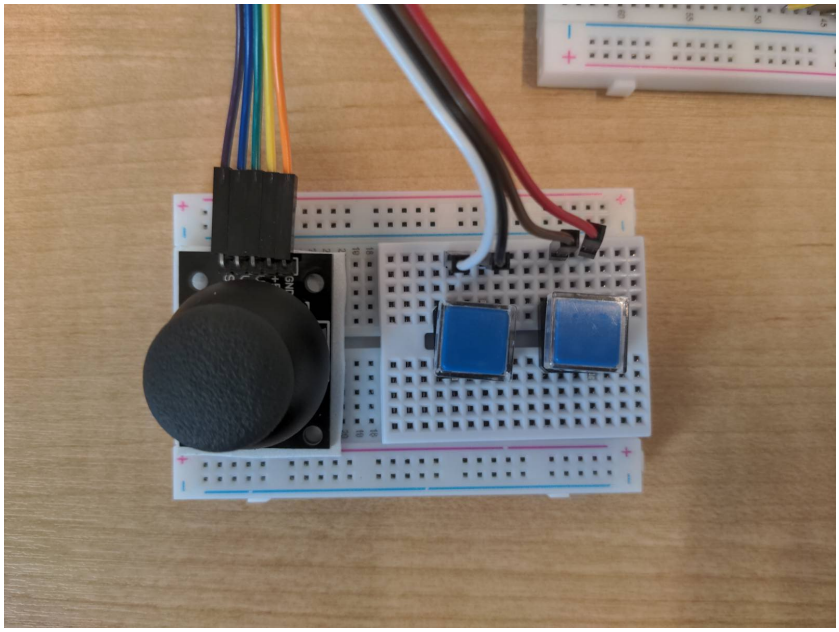


Figure 25: Close-up view of the controller

On the PSOC Creator side, I added a debouncer component for each button, combined with a 100 Hz clock:

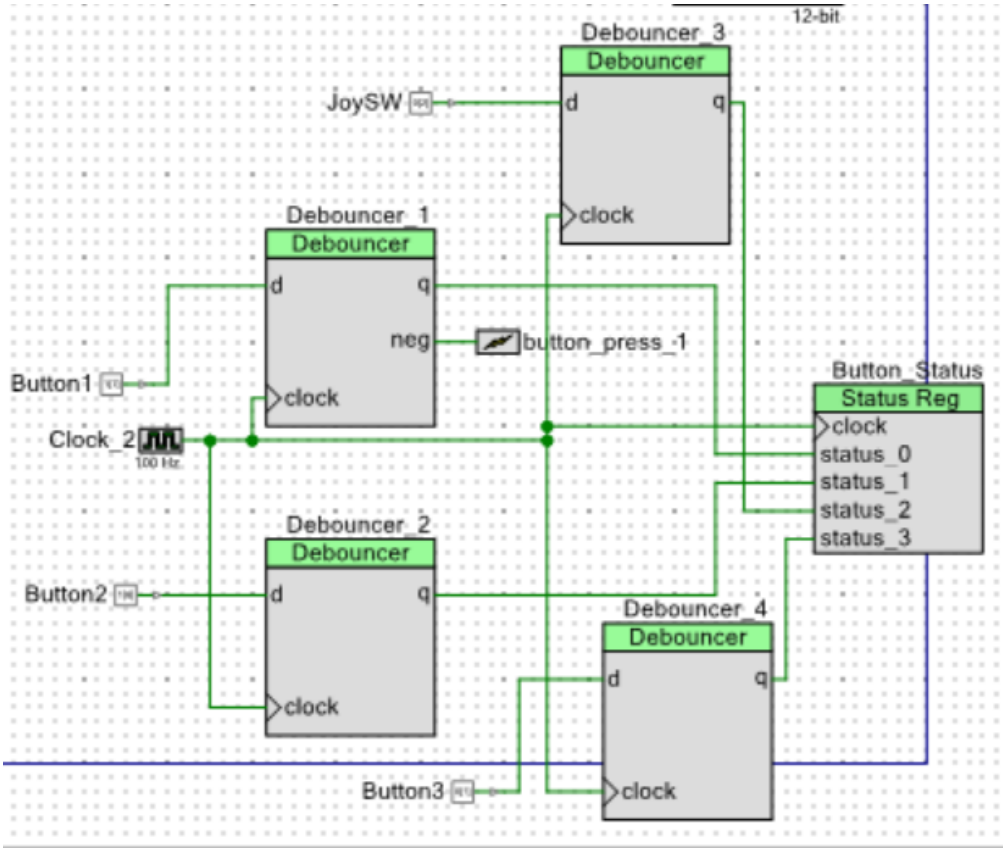


Figure 26: Debounced buttons; 100 Hz clock on each debouncer

And for the joystick, I used the two SAR ADCs available on the PSOC 5 to read the analog inputs:

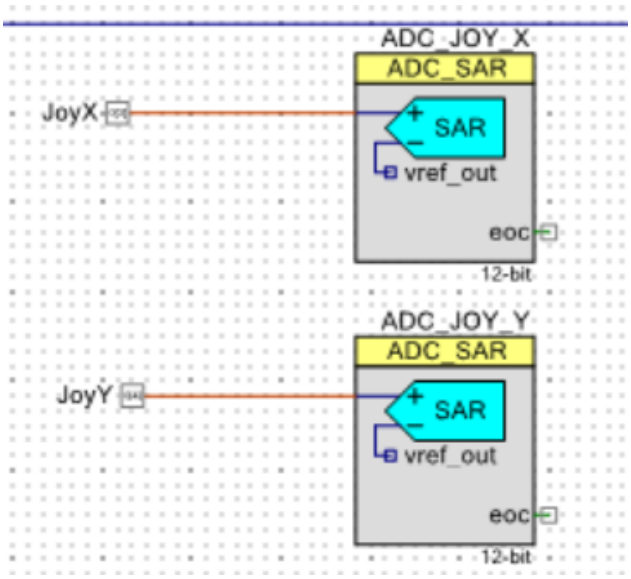


Figure 27: Reading analog inputs from the joystick using the ADCs.

The Game Boy itself has all of its buttons as a memory mapped peripheral at location 0xFF00.

Adding this to my emulator was very simple. I started with a new struct called Mmio to handle all of the mapping between hardware and memory locations.

```
/*
This struct handles mapping physical hardware on the PSOC to memory locations
*/
#ifndef MMIO_H
#define MMIO_H
#include "memory.h"
typedef struct Mmio {
    Memory* mem;
} Mmio;
void setup_mmio(Mmio* mmio, Memory* mem);
void tick_mmio(Mmio* mmio);
#endif
```

And in the tick_mmio() funciton, I simply write to the appropriate memory location for the hardware:

```
void tick_mmio(Mmio* mmio) {
    int joyx = ADC_JOY_X_GetResult16();
    int joyy = ADC_JOY_Y_GetResult16();
    uint8_t buttons = Button_Status_Read();
    bool button1 = !(buttons & 0b0001);
    bool button2 = !(buttons & 0b0010);
    bool joy_sw = !(buttons &  0b0100);
    bool button3 = !(buttons & 0b1000);

    // Map inputs to gameboy inputs
    bool left_pushed = joyx > 3700;
    bool right_pushed = joyx < 500;
    bool up_pushed = joyy < 500;
    bool down_pushed = joyy > 3700;
    bool a_pushed = button1;
    bool b_pushed = button2;
    bool start_pushed = joy_sw;
    bool select_pushed = button3;

    // Write to memory location 0xFF00
    //Bit 7 - Not used
    //Bit 6 - Not used
    //Bit 5 - P15 Select Action buttons    (0=Select)
    //Bit 4 - P14 Select Direction buttons (0=Select)
    //Bit 3 - P13 Input: Down  or Start    (0=Pressed) (Read Only)
    //Bit 2 - P12 Input: Up    or Select   (0=Pressed) (Read Only)
    //Bit 1 - P11 Input: Left  or B        (0=Pressed) (Read Only)
    //Bit 0 - P10 Input: Right or A        (0=Pressed) (Read Only)
    //uint8_t selected = fetch(mmio->mem, 0xFF00, false);
    uint8_t selected = mmio->mem->joyp;
    uint8_t to_write = selected & 0x30;
    if (((selected >> 5) & 0b1) == 0) {
        // action button selected
        // direction button selected
        to_write |= (!a_pushed) & 0b1;
        to_write |= ((!b_pushed) << 1) & 0b10;
        to_write |= ((!select_pushed) << 2) & 0b100;
        to_write |= ((!start_pushed) << 3) & 0b1000;
        //write_mem(mmio->mem, 0xFF00, to_write);
        mmio->mem->joyp = to_write;
    } else if (((selected >> 4) & 0b1) == 0) {
        // direction button selected
        to_write |= (!right_pushed) & 0b1;
        to_write |= ((!left_pushed) << 1) & 0b10;
        to_write |= ((!up_pushed) << 2) & 0b100;
        to_write |= ((!down_pushed) << 3) & 0b1000;
```

```
47          //write_mem(mmio->mem, 0xFF00, to_write);
48          mmio->mem->joyp = to_write;
49      } else {
50          //write_mem(mmio->mem, 0xFF00, 0xFF);
51          mmio->mem->joyp = to_write;
52      }
53  }
```

Notice that the write_mem() calls are commented out to instead write directly to $mem- > joyp$. This was another (tiny) optimization that helped improve the performance of the emulator very slightly.

And with that done, I tried to boot Tetris:

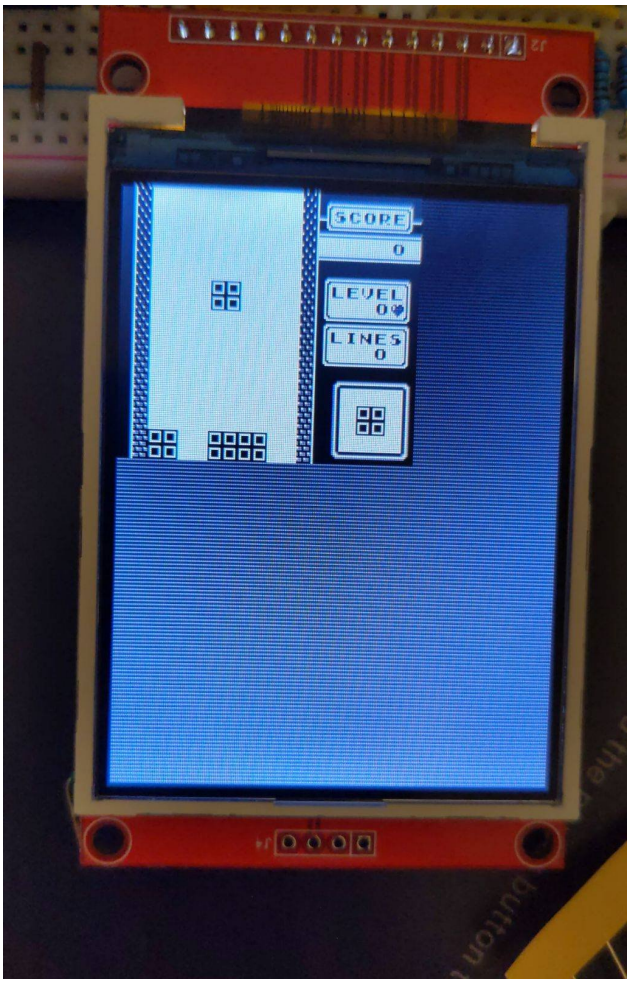


Figure 28: World's easiest Tetris game.

Not quite right. It turns out that Tetris uses the Game Boy's timer to generate random numbers to choose blocks. Without this implemented, all of my blocks were squares. While this did make Tetris extremely easy to play, it isn't quite what we're after.

17 Emulating Timer Interrupts

The Sharp LR35902 has an interesting hardware timer system. There are four memory registers:

Memory Location	Abbrev.	Name	Function
0xFF04	DIV	Divider Register	Incremented at 16384Hz. Any writes reset it to 0
0xFF05	TIMA	Timer Counter	Incremented at the speed specified by TAC (0xFF07). Overflows trigger a timer interrupt
0xFF06	TMA	Time Modulo	The value inside this register is used as the reset value for TIMA when TIMA overflows.

0xFF07	TAC	Timer Control	Timer control register for TIMA. Uses the following scheme:
			`Bit 2 - Timer Enable`
			`Bits 1-0 - Input Clock Select`
			`00: CPU Clock / 1024`
			`01: CPU Clock / 16`
			`10: CPU Clock / 64`
			`11: CPU Clock / 256`

To emulate these hardware timers, I started another struct Timer to manage the timer interrupts and timing in my CPU. This was the final piece of the Tetris puzzle:

```c
#ifndef TIMER_H
#define TIMER_H
#include "memory.h"
typedef struct Timer {
    Memory* mem;
    int internal_clock;    //internal clock counting elapsed m-cycles
    int baseclock; // fastest timer speed; increments every 4 m-cycles
    int divclock;  // DIV register increments at 1/16th the rate of a regular increment (4*16 m-cycles)
} Timer;

// Initializes a new timer
void setup_timer(Timer* timer, Memory* mem);

// processes the next tick of the timer
// Takes in the # of machine cycles that elapsed
void tick_timer(Timer* timer, uint8_t delta_machine_cycles);

#endif
```

The Timer struct simply emulates another state machine that updates itself based on the amount of machine cycles that have passed, much like the one in the GPU. The tick_timer() function looks like:

```c
void tick_timer(Timer* timer, uint8_t delta_machine_cycles){
    timer->internal_clock += delta_machine_cycles;
    // Keep ticking the timer until the internal clock < 4
    // It is possible for ticks to take >4 machine cycles when RSTs occur
    while (timer->internal_clock >= 4) {
        // the DIV register is ALWAYS counting!
        timer->internal_clock -= 4;
        timer->divclock++;
        if (timer->divclock == 16){
            // the DIV register is incremented once every 16*4 m-cycles
            timer->mem->timer_divider++;
            timer->divclock = 0;
        }

        // Check if timers are enabled
        uint8_t timer_control = timer->mem->timer_control;
        //Bit  2   - Timer Enable
```

```
        //Bits 1-0 - Input Clock Select
        //   00: CPU Clock / 1024 =  4096 Hz   = once every 4*64 m-cycles = 64 timer.↩
            base_clock s
        //   01: CPU Clock / 16   =  262144 Hz = once every 4 m-cycles =   1 timer.↩
            base_clock s
        //   10: CPU Clock / 64   =  65536 Hz  = once every 4*4 m-cycles = 4 timer.↩
            base_clock s
        //   11: CPU Clock / 256  =  16384 Hz  = once every 4*16 m-cycles = 16 timer.↩
            base_clock s

        if (timer_control & 100){
            timer->baseclock++; // the fastest base clock speed is once every 4 m-cycles
            int base_clock_threshold;
            switch (timer_control & 3){
                case 0:
                base_clock_threshold = 64;
                break;
                case 1:
                base_clock_threshold = 1;
                break;
                case 2:
                base_clock_threshold = 4;
                break;
                case 3:
                base_clock_threshold = 16;
                break;
            }

            // Time to increment TIMA?
            if (timer->baseclock >= base_clock_threshold) {
                timer->baseclock = 0;

                if (timer->mem->timer_counter == 0xFF) {
                    // This increment will cause an overflow; request interrupt
                    timer->mem->interrupt_flag |= INTERRUPT_ENABLE_TIMER_MASK;
                    // And refill wil the timer modulo value
                    timer->mem->timer_counter = timer->mem->timer_modulo;
                } else {
                    timer->mem->timer_counter++;
                }
            }
        }
    }
}
```

This is quite a simple implementation of the timer, but it is more than enough for our purposes!

18 Tetris! And other ROMS

Now, I tried booting Tetris, and was able to play a full Tetris game! I also tried Dr. Mario, which played flawlessly.

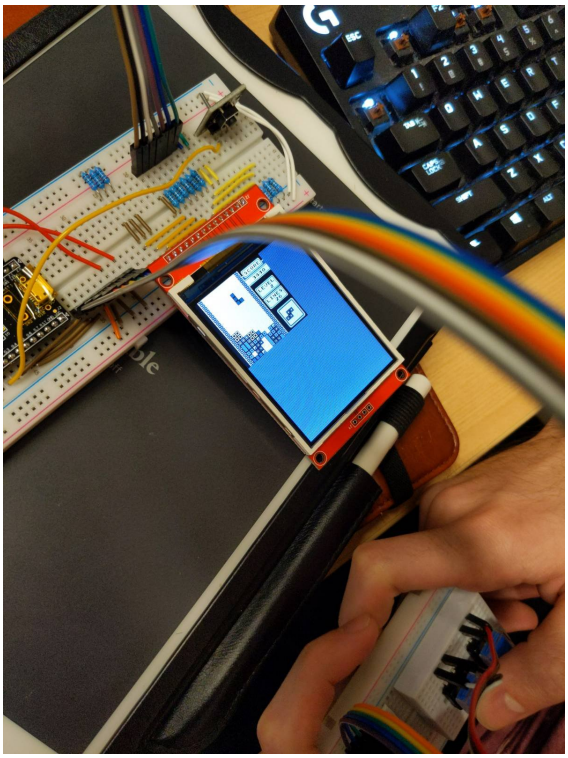


Figure 29: Playing some Tetris

Figure 30: Dr. Mario in action (I have no clue how this game works)

19 Custom Boot ROM

With my emulator working, I decided to add some flare by giving it a custom boot ROM:

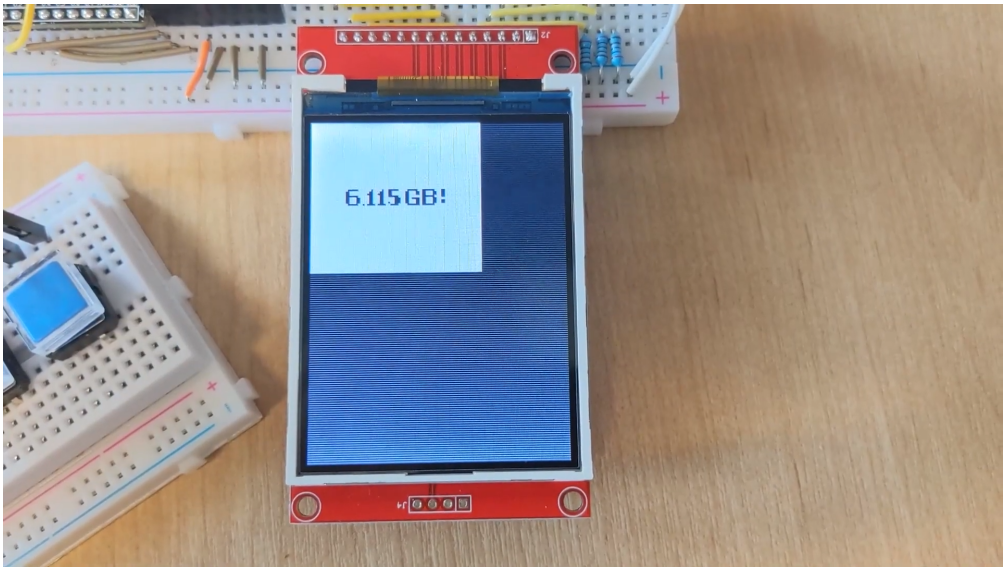


Figure 31: Custom boot logo

This was accomplished with the help of this excellent open-source project:
https://github.com/dobyrch/bootrom-gen

20 Conclusion

And with that, my emulator project is complete! I actually finished about two weeks earlier than the due date, but this was simply because I started a few weeks earlier.

20.1 Where to go from here

I'm happy with the current state of the emulator, but there are certainly areas where it can be improved given more time/effort. A huge section of a good emulator is currently missing: sound. The sound system on the Game Boy is actually quite complicated, and it would likely require significant investment in both the hardware of the system and the emulator. Additionally, the emulator is not able to run games larger than 32kB. This is a constraint of the hardware; the system could certainly be extended with additional memory hardware, but I did not have the foresight to request additional memory at the start of the project. Lastly, the speed of the emulator could probably be improved as well with more effort.

21 Appendix

21.1 Code

The full source code can be found here on my GitHub page:
https://github.com/raytran/psoc5-gb-emulator/

21.2 Useful Game Boy Emulator Development Links

- https://gbdev.io/pandocs/ This is a tremendously useful resource for Game Boy internals.

- https://izik1.github.io/gbops/ The most accurate instruction set reference for the Game Boy
- https://rgbds.gbdev.io/docs/v0.4.2/gbz80.7 Provides more in-depth explanations on opcodes.
- https://www.youtube.com/watch?v=HyzD8pNlpwI Extremely detailed talk on the Game Boy system. Was very useful for implementing the GPU.
- https://vimm.net/ ROM archive

6.115 Rocks!

Thank you to Professor Leeb & the rest of the 6.115 course staff for the wonderful semester!