Mini Project Report

on

"An Emulator for Atari 2600 Video Console System"

submitted in partial fulfillment of requirement
for the award of degree of

Bachelor of Engineering

in

Computer Science and Engineering

Submitted by

Shreeyash Pandey   **Ayush Raj Singh**

Anshul Nagrare

Guided by

Prof. Vikrant Chole

Department of Computer Science and Engineering
G H Raisoni Institute of Engineering and Technology, Nagpur
(Formerly Known as G H Raisoni Academy of Engineering & Technology)
(An Autonomous Institute Affiliated to Rashtrasant Tukadoji Maharaj Nagpur University, Nagpur)
Accredited by NBA and NAAC with A+ Grade
2021-22

DECLARATION

we hereby declare that the Mini Project Report titled "**An Emulator for Atari 2600 Video Console System"** submitted herein has been carried out by us in the Department of Computer Science and Engineering of G. H. Raisoni Institute of Engineering & Technology, Nagpur. The work is original and has not been submitted earlier as a whole or in part for the award of any degree / diploma at this or any other Institution / University.

Project Members

Shreeyash Pandey

Ayush Raj Singh

Anshul Nagrare

Date: 31st May, 2022

CERTIFICATE

The Mini Project Report titled **"An Emulator for Atari 2600 Video Console System"** submitted by, **Shreeyash Pandey, Ayush Raj Singh and Anshul Nagrare** , 6th Semester C.S.E. As a part of the Bachelor of Engineering in Computer Science and Engineering by Rashtrasant Tukdoji Maharaj Nagpur University, Nagpur is an authentic work carried out by them under my guidance in the Department of Computer Science and Engineering of G. H. Raisoni Institute of Engineering and Technology, Nagpur during the academic session 2021-2022.

To the best of my knowledge, the matter embodied in the Mini Project report has not been submitted to any other University/ Institute for the award of any degree or diploma.
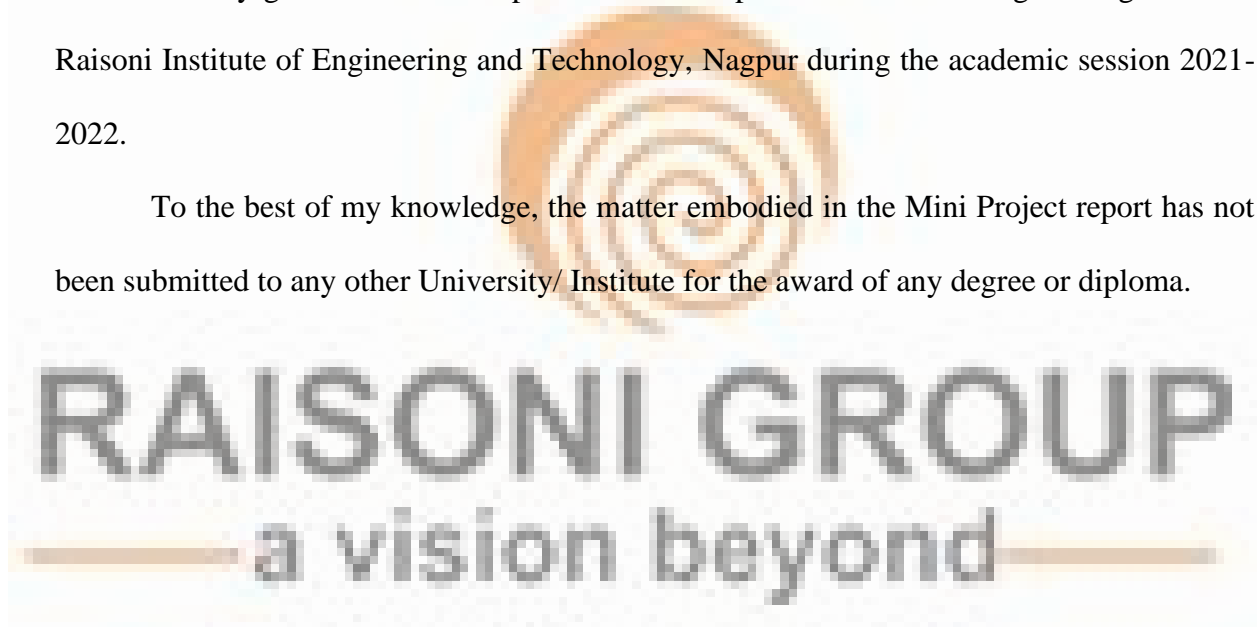
Prof. Vikrant Chole
Assistant Professor
Computer Science and Engineering
G.H.R.I.E.T., Nagpur

Dr. Sonali Ridhorkar
Head of Department and Associate
Professor
Computer Science and Engineering
G.H.R.I.E.T., Nagpur

ACKNOWLEDGEMENT

<div align="right">

Shreeyash Pandey

Ayush Raj Singh

Anshul Nagrare

</div>

INDEX

LIST OF FIGURES

1. INTRODUCTION

Atari, Inc.

Atari, Inc. was an American video game developer and home computer company founded in 1972 by Nolan Bushnell and Ted Dabney. Atari was responsible for the early video game arcades and is well known for its games and home video game consoles.

The company was primarily founded for the objective of developing arcade video games for the market. With the ongoing advancement of technology in the early-mid 90s, computer hardware technology emerged and flourished well because of the fierce competition between the tech giants of the time, like: MOS Technology, Inc. ("MOS" stands for Metal Oxide Semiconductor); Motorola; Intel. The rivalry increased with the competitors making the commercially manufactured computer hardware available to the general public in forms of cheaper computer and other systems.

Out of this cheaper hardware, the most prominent was the MOS Technology's 6502 CPU, it outperformed the more complex 6800 and Intel 8080, but it cost less and was easier to use. A few versions were developed soon after, these were the 6503 and the 6507. The 6507 was later bought by Atari to implement it in its pioneering achievement of home video game consoles (Atari 2600).

Atari 2600, VCS

The Atari 2600, also known as the Atari Video Computer System (Atari VCS), was released as the first home video game console by Atari, Inc. It was released in 1977, and was the pioneer of home video game consoles and popularized the cheaper microprocessor-based systems in the video game industry with features like swappable ROM cartridges. Before this the only places you can enjoy playing video games were the Arcade centers. These having the well-known arcade machines or the arcade consoles were the only way of indulging in pumped-up sessions, trying to break the highest record. The arcade consoles were big and definitely not portable and required a ton of work just to set up a system running. Also, generally a single arcade console was capable of running only a single non-swappable game.

So, the Atari introducing a way of playing games at home was a huge deal to many people. It introduced a new form of entertainment that can be enjoyed by the whole family at an affordable price. After its release it went through a lot of improvement, from running its intended game pong to running a game with advanced visuals, like Activison's "Pitfall!". Around the mid-1980s, Atari 2600 was the dominant home video game console.

Emulators

In computing, an emulator is hardware or software that enables one computer system (called the host) to behave like another computer system (called the guest). An emulator generally emulates a computer hardware, allowing any compatible software or hardware to run on it. Emulation is defined as the process of emulating (or imitation) of a program or hardware by another computer program. For example, many knock-off handheld video game consoles emulates the original consoles like the Nintendo 64, Gameboy Advance, Nintendo DS, and more only because of the popularity of these consoles and the vast majority of games being developed for them. These knock-off consoles emulate the original system so well that despite not being original, it can perfectly run the software and games originally made for the other authentic consoles. Another type of such emulators are the software programs that emulates the respective consoles. Many video game enthusiasts that like to play old retro games make use of such emulators as the respective consoles are no longer in the line of production or are otherwise very hard to find a working one.

2. LITERATURE REVIEW

There are unfortunately not many texts on writing Atari 2600 emulators however there are many books and articles written on writing emulators for other consoles. For example, this[6] video series by David Barr goes into creating an emulator for Nintendo NES System. These videos will prove essential when writing VCS.

The Bible of 2600 information is the manual[2] written by Steve Wright. It describes functionality of the TIA, timing, colors, audio generation, graphics objects and their manipulation, PIA, timers, IO ports and controllers.

Another Important manual regarding TIA is the Hardware Manual[7] produced by Atari Inc. It details TIA and PIA registers, their configurations, and their consequences.

Some tutorials on 2600 programming through assembly are available too. These include "2600 101" by Kirk Israel[5] and "Making Games for Atari" by Steven Hugg[1].

[8] is a handy reference for 2600 Hardware specification. It specifies the division of memory in the 2600 and more.

The manual[3] produced for 6502 on the commodore should suffice for implementing the 6502 CPU.

Furthermore, stella[9], the already-existing emulator for Atari 2600 can be used as a reference implementation.

3. OBJECTIVES

- To design and write a NTSC TV-Spec compliant emulator that can run at-least one Atari 2600 game.
- To experience the process of designing large software programs and the challenges involved.
- To understand program-compile-debug cycle and development environments of different systems.
- To learn and use build systems such as Make and CMake, compiler tool-chains such as Gcc and debugging tools such GDB and Valgrind.
- To understand principles of software engineering and testing
- To understand fundamentals of computer architecture, instructions and how programs are run by a CPU.

4. METHODOLOGY

Three major components of the VCS were identified:

1. CPU (Central Processing Unit)

2. TIA (Graphics Processing Unit)

3. PIA (Peripheral Input Processing Unit)

Logically, these three components are the only modules in the system. Each module processes its input and passes its output via an interface to the next module. However, physically the program can be made more modular and be turned into their own structures. Structures that share features with their parent module but are unique enough to deserve their own module. These structures were identified as the program was being built. A module contains both: its interface and implementation. The final 6 modules (in order of their creation) are:

1. Mspace

2. Log

3. Main

4. CPU

5. TIA

6. Pia

Fig 4.1 VCS Flow Diagram.

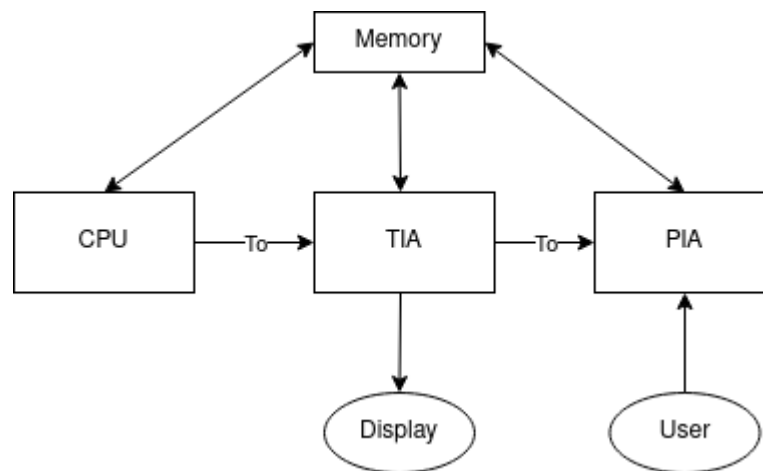The above flow diagram illustrates how communication takes place in the VCS. The memory contains instructions which are read and executed by the CPU. CPU changes the global state of the VCS and passes control to the TIA which executes its procedures and updates the DISPLAY. The TIA then passes control to the PIA which reads inputs from users, updates the MEMORY and passes control back to the CPU.

5. SOFTWARE AND HARDWARE REQUIREMENTS

The following sections outline the basic system requirements for running an Emulator under various operating systems.

General

- SDL version 2.0.5 or greater, latest version highly recommended
- 15/16-bit color minimum; 24/32-bit color graphics card highly recommended
- Enough RAM for the OS + 256MB RAM for the emulation; 512MB+ highly recommended
- Mouse with real paddles required for paddle emulation
- ROM image for the game: pong.

Windows

The Windows version of Emulator is designed to work on Windows 7/8/10 with the following:

- Direct3D or OpenGL capable video card
- Visual C++ 2017/2019 Community is required to compile the Emulator source code

Linux

The Linux version of Emulator is designed to work on a Linux Workstation with the following:

- i386 or x86_64 class machine, with 32 or 64-bit distribution
- OpenGL capable video card
- Other architectures (MIPS, PPC, PPC64, etc.)
- GNU g++ v/7 or Clang v/5 (with C++17 support) and the make utility are required for compiling the Stella source code

6.DESIGN AND IMPLEMENTATION

The final 6 modules (in order of their creation) are:

1. Mspace (Memory/Address Space):

      Interface: Fetch and set addresses.

      Implements: Address Space of the VCS.

2. Log:

      Interface: Log functions

      Implements: Log functions.

3. Main:

      Interface: None

      Implements: Entrypoint. All initializations are done here.

4. CPU:

      Interface: Execute instructions.

      Implements: All instructions.

5. TIA:

      Interface: Display and refresh the screen.

      Implements: All graphical functions.

6. PIA:

      Interface: Read inputs and set timers.

      Implements: Inputs and timers.

A detailed description of these 6 modulesis in order.

Module: Main (main.c)

A walkthrough from main() to the end helps in understanding the design of the program.

```
int main(int argc, char *argv[]) {
    // ...
    emu_init(argc, argv);
    // ...
```

```
        while (pc < CARMEM_END - 1) {
              machine_cycles = run_cpu();
              color_clocks = run_tia(machine_cycles);
              run_pia(machine_cycles);
              pc = fetch_PC();
        }
}
```

The main function is short and sweet. All it has to do is initialize variables and structures by calling emu_init() and enter a loop. emu_init() initializes all the global structures that will be explained later in this section. The loop calls three functions over and over again and exits if the program counter (pc) is beyond the last byte in the memory.

```
cycles_t run_cpu() {
        addr_t pc = fetch_PC();
        byte_t opcode = fetch_byte(pc);
        cycles += inst_exec(opcode);
        pc = fetch_PC();
        pc += inst_bytes(opcode);
        set_PC(pc);
        return cycles;
}
```

run_cpu() fetches the next opcode from the memory, executes that instruction and updates the PC.

```
cycles_t run_tia(cycles_t machine_cycles) {

        cycles_t clocks = machine_cycles * 3;
        for (unsigned int i = 0; i < clocks; ++i) {
                tia_exec();
        }
        return clocks;
}
```

The TIA in VCS is three times faster than the CPU. This means that for every CPU cycle, there are three TIA cycles executing concurrently. To avoid needless complexity, this emulator follows a serial mode of execution. The CPU executes its instructions and passes the number of cycles that it uses to the TIA. The TIA then runs for cpu_cycles * 3 cycles. This is visible in run_tia() with the for loop running machine_cycles * 3 times. tia_exec() is the interface provided by the TIA module.

```
cycles_t run_pia(cycles_t machine_cycles) {
```

```
        handle_input();
        cnt_pia_cycles(machine_cycles);
        return machine_cycles;
}
```

run_pia() calls the interfacing functions of the PIA module and returns.

Module: mspace (mspace.c)

```
/* Total Address space */
static byte_t mspace[0xffff]; /* 16KB */


/* CPU Registers */


static byte_t A;           /* Accumulator */
static byte_t X;           /* General Purpose Register X */
static byte_t Y;           /* General Purpose Register X */
static addr_t S = RAM_END; /* Stack Pointer */
static byte_t P = 32;      /* Program Status Word. 32 bcoz the 5th
bit

                                  is supposed to be logical 1
at all times */
static addr_t PC;              /* Program Counter */


/* Interfacing functions */
byte_t fetch_byte(addr_t addr);
void set_byte(addr_t addr, byte_t b);


void load_cartridge(char *filename);
```

mspace[] is the total address space. It contains the TIA registers, RAM, Cartridge and PIA registers. CPU registers are self-explanatory and so are memory fetch and set functions. The maximum cartridge size supported in the VCS is 4KB. The total address space is 16KB. A 4KB portion of the total address space is used to store the running program (the cartridge). load_cartride() loads a file (also called a ROM) into the running memory at CARMEM_START (defined in mspace.h). mspace also has many auxiliary functions.

Module: CPU (cpu.c)

The CPU is the biggest module in the program. A whopping 2000 sloc. It implements all 151 legal 6502 instructions. It contains the following structures and functions:

```c
typedef int (*inst_fptr) (byte_t opcode);

typedef struct inst_t {
    int bytes;
    int cycles;
    inst_fptr exec;
    char *name;
} inst_t;

static inst_t inst_tbl[INSTN];

void inst_tbl_init() {
    // ...
    inst_assign(0x69, 2, 2, adci,"adci");
    inst_assign(0x65, 2, 3, adcz,"adcz");
    inst_assign(0x75, 2, 4, adczx,"adczx");
    // ...
}


int adci(byte_t opcode) {
    byte_t operand = fetch_operand(opcode);
    byte_t A = fetch_A();
    // TODO: Check for Overflow (STATUS_V)
    if (A + operand > 255) {
        set_STATUS(STATUS_C);
    }
    A += operand;
if ((A >> 7) == 1) {  // 7th bit of A is set
        set_STATUS(STATUS_N);
    }
    if (A == 0) {                  // Result of last operation was zero
        set_STATUS(STATUS_Z);
    }
    set_A(A);
    return 0;
}
```

The CPU is designed around a hash-table-function-pointer dispatch strategy. inst_tbl[] is a hash table in which the indexes are the opcodes of instructions and the values are inst_t objects

that contain information of an instruction: size, cycles and most importantly a function pointer to the function that implements the instruction. A snippet of inst_tbl_init() shows how inst_tbl[] is initialized.

```
/* Interface functions */


char *inst_name(byte_t opcode);
byte_t inst_bytes(byte_t opcode);
byte_t inst_cycles(byte_t opcode);
byte_t inst_exec(byte_t opcode) {
      return ((inst_tbl[opcode])).exec(opcode);
}
```

These functions use an opcode as an index to look up into inst_tbl and return corresponding values. inst_exec() calls the function associated with an opcode i.e. this function executes an instruction.
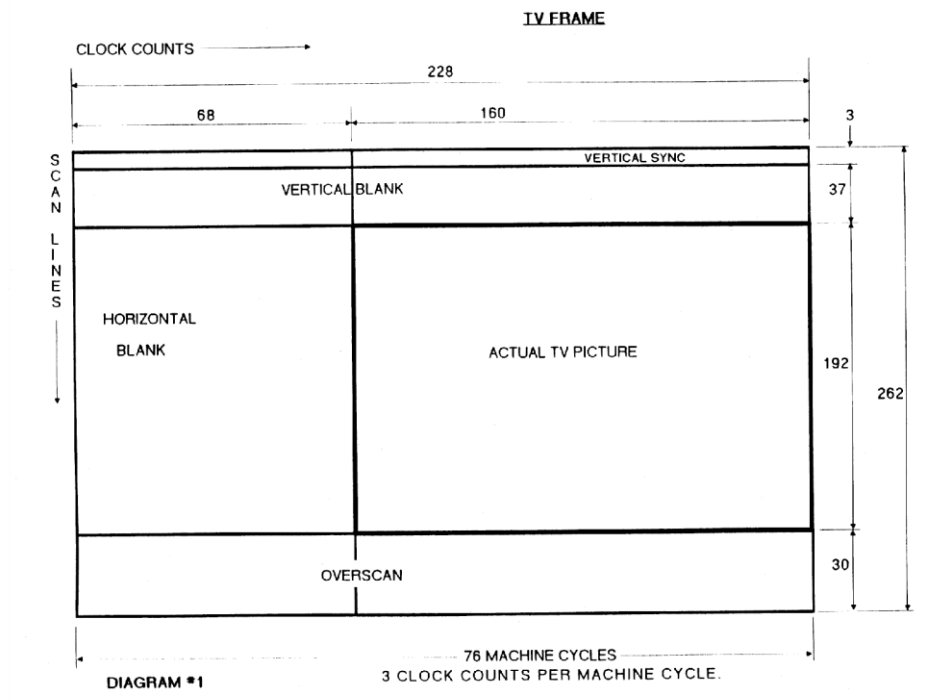
Module: TIA (tia.c)

How TIA works

Fig 6.1 NTSC Frame [2]

A frame starts at the top left of the screen. An electron-gun sweeps through the screen illuminating each pixel according to the voltage provided to it. At the top of the frame the gun is in VBLANK region i.e., it isn't shooting electrons right now and won't be for next 40 scanlines. A programmer may use this time for heavy computations. On the start of the 41st scanline, the gun doesn't shoot for first 68 color clocks. This is the HBLANK region. On the 41st scanline,

at the 69th color clock, the first pixel is displayed. The electron sweeps across the screen, displaying pixels till it reaches the end, after which it has to be reset to the next line on the left. The time it takes to reset the gun is the Horizontal Blank (HBLANK) time. Likewise, as the gun reaches the end of a frame, it has to be reset back to the top left. The time it takes to do this is the Vertical Blank (VBLANK). Consider overscan to be extra time for more computations. The resolutions given in Fig 4.2 demonstrate the over scan color clocks consumed by each region for a full treatment on TIA, refer TODO (TIA manual, Steve wright)

Implementation:

```
static unsigned int hi = 0;      // horizontal index
static unsigned int vi = 0;      // vertical index
static unsigned int chi = 0;      // horizontal index in the visible
region
static unsigned int cvi = 0;       // vertical index in the visible
region

static unsigned int ti = 0;      // total index
static unsigned int cti = 0;     // total index in the visible region

void tia_exec() {
    if (isonscreen()) {
        place_pixel();
        if (cti >= VISIBLE_HEIGHT * VISIBLE_WIDTH - 1) {
        /* at the end of a frame */
            display();
        }
    }
}
```

libSDL allows a user of the API to provide an array of pixel of a foreknown size to be displayed. We use this to our advantage. static unsigned int hi = 0; // horizontal index static unsigned int vi = 0; // vertical index static unsigned int chi = 0; // horizontal index in the visible region static unsigned int cvi = 0; // vertical index in the visible region static unsigned int ti =

0; // total index static unsigned int cti = 0; // total index in the visible region void tia_exec() { if (isonscreen()) { place_pixel(); if (cti >= VISIBLE_HEIGHT * VISIBLE_WIDTH - 1) { /* at the end of a

frame */ display(); } } } The TIA updates a frame buffer of pixels at every cycle and displays it at the end of a frame. tia_exec() does this. tia_exec() is also responsible for maintaining and updating total index pointers.Total index pointers (hi, vi, ti) contain positions in the total frame.

```
void place_pixel() {
      pixel_t p = select_pixel();
      frame_buffer[cti] = p;

      chi++;
      if (chi >= VISIBLE_WIDTH) {
            chi = 0;
            cvi++;
      }
      if (cvi >= VISIBLE_HEIGHT) {
                  cvi = 0;
      }
      cti = cal_total_cindex(chi, cvi);
}
```

place_pixel() is responsible for filling the frame buffer. place_pixel() is also responsible for maintaining visible frame pointers. Visible frame pointers contain positions in the visible frame.static pixel_t color_map[256];

```
static pixel_t color_map[256];

pixel_t select_pixel() {
      pixel_t rv;
      rv = current_pf_pixel();
      rv = current_mx_pixel();
      rv = current_bl_pixel();
      rv = current_pl_pixel();
      return color_map[rv];
}
```

VCS only works with 8-bit colors. libSDL accepts 32-bit RGBA. color_map[] is the solution to this problem. It maps all possible 256 8-bit colors to an equivalent 32-bit RGBA. select_pixel() decides which pixel is to be stored in the frame buffer by following a rule. current_xx_pixel() functions execute according to this rule. The background color should be that of the playfield or the background, current_pf_pixel() decides which one.

current_mx_pixel() which executes after will either return a new missile pixel if applicable or else a playfield pixel, the default. Likewise, all following functions return the pixel return by previous function if a change in pixels in not applicable. Include optional: strobe registers
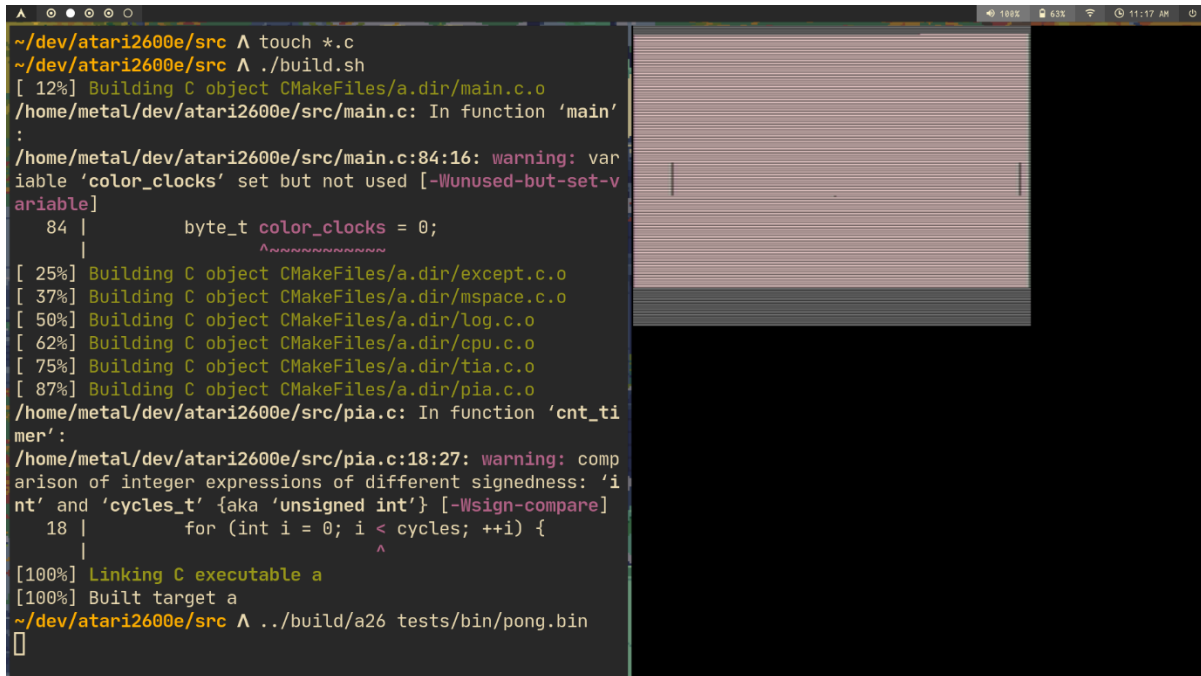
Module: PIA (pia.c)

The PIA is simpler and shorter than the other two modules. There are only three functions: void pia_process_input(int code); void cnt_pia_cycles(uint32_t cycles); void set_timer(byte_t intervals, uint32_t number)set_timer() is called when one of the PIA timer registers are written. cnt_pia_cycles() is called every CPU cycle. It updates the timers. pia_process_input() accepts inputs from users and sets appropriate registers in the Memory.

Module: Log (log.c)

Log.c is a MIT-Licensed open-source single-header library (Credit: https://github.com/rxi) used to for convenient and easy logging functions. This module shall not be present in release versions of code as it is only required for debugging.

7. RESULTS AND DISCUSSION

Fig 7.1 The game "Pong" runs on our Emulator.

The game "Pong" was used to test the emulator. It ran successfully. The game is controlled through 8 keys (4 for each player). Player 1 is controlled with W, A, S, D keys and player 2 is controlled through arrow keys Up, Down, Left, Right. The object of the game is to prevent an opponent from scoring a goal. Each player gets a Paddle which can be moved up and down. By moving these paddles and timing them correctly, a player can strike the ball which then ricochets its way to the other player.

The speed of emulation of the paddles turned out to be greater than the ball. This is speculated to be happening due to the rate at which graphical objects are aligned vertically versus horizontal alignment. Horizontal movement turned out to be slower than vertical because a horizontal move is shown only after the end of a frame whereas vertical movements take effect after every scan line.

A slight jitter is seen which could be caused by the timing (or lack thereof) in the program. Cycle-accurate emulators (Emulators that replicate behavior of a host to the cycle-level) do not face this problem as they meet the expectations of programs which were programmed to work on a true, cycle-accurate system.

Color, it is observed, does not match that of our reference implementation as different color-pallettes are used by both programs.

The objectives have been adequately realized. Tools like compilers and debuggers have been used and understood. Many topics relating to Computer Architecture, Instruction Sets, Intelligent solutions to programming in resource constrained environments, Program Organization and Program Structure has been thoroughly understood.

Limitations

- This program has only been tested with the game "Pong."
- Implementation of the CPU module, although tested, could use automated testing utilities.
- Program lacks a vibrant user-interface which can ease the use of loading roms into programs.
- ROMS can only be loaded at the start of the program.
- Game ROMS are difficult to obtain as they are protected by Copyright.

8. FUTURE SCOPE

Many VCS games of greater complexity use "illegal instructions" i.e. instructions that are not officially documented but supported by the CPU. As there were no official documentations available, illegal instructions had to be left out of this implementation. As a Result, many games are unplayable.

Moreover, games that don't fit in 4KB of cartridge space cannot be supported directly. VCS supported it through a technique called "bank switching". Due to time constraints, this too has been left out from the program.

Future scope of this program would entail implementing aforementioned techniques. Moreover, it can be made more user-friendly by providing user interfaces.

ROM hot-plugging can be implemented easily by writing to the cartridge memory and calling a reset-interrupt procedure.

Emulators are of paramount importance when real hardware is not available or too expensive. It can greatly speed up a development process by allowing a developer to integrate direct testing on the hardware into his/her development cycle.

9. REFERENCES

[1] Steven E. Hugg, "Making Games for The Atari 2600", ISBN-13: 978-1541021303ISBN-10: 1541021304, 2016.

[2] Steve Wright, "2600 (STELLA) Programmer's Guide", 1979 (12/03/1979)

[3] Commodore Business Machine, Inc., "Commodore 64 Programmer's Reference Guide", ISBN 0-672-22056-3, 1983

[4] Norbert Landsteiner, "6502 "Illegal" Opcodes Demystified", Masswerk.at, 2021

[5] Kirk Israel, "2600 101 - Tutorial", Alienbill.com, 2004

[6] David Barr, "NES Emulator from scratch", Self-published, 2020

[7] Atari Inc., "TIA 1A Hardware Manual", Archive.org, 1977

[8] No Cash, "Atari 2600 Specifications", Problemkaputt.de

[9] Stella, "Stella - A multiplatform emulator for Atari 2600", Self-published