Cosmic
A Software Simulated 8-Bit Computer Architecture

Clay Buxton
Computer Engineering, Computer Science
Elizabethtown College
Elizabethtown, PA
buxtonc@etown.edu

Kevin Carman
Computer Engineering, Computer Science
Elizabethtown College
Elizabethtown, PA
carmank@etown.edu

***Abstract*—The Cosmic processor, graphical user interface (GUI), and assembler aim to educate students with little or no knowledge of computer architecture or assembly language. Derived from 8-bit processors of the 1980's, Cosmic was built with simplicity and modularity in mind. The simplicity is in the fundamentally basic and well-documented instruction set, while the modularity stems from the design of the system. The system was designed in a way that anyone could write their own simulation environment, graphics stack, peripherals, drivers, and more to use the Cosmic processor at its core. Written in C++ and Python with minimal dependencies, Cosmic is lightweight and portable, which makes it perfect for running on almost any system, including running headless on systems with low graphical power. While no experimentation on its educational properties has currently been conducted, Cosmic is ready with informative labs for students to engage with. Modern, complex assembly languages will no longer be a starting point for novice programmers.**

***Index Terms*—retro-computing, simulation, emulation, microprocessor, 8-bit architecture.**

I. Introduction

intro and problem definition

II. Background

background

III. Design Constraints

design constraints

IV. Timeline

timeline fall/spring semester

V. Budget

At first, we thought since our entire project is in software, that nothing would need to be bought. As development continued, we decided to purchase two 4GB Raspberry Pis to experiment interfacing Cosmic with real-world hardware. For just under $90 each, these Raspberry Pis greatly outclassed the under-powered ones that were available for use in the Elizabethtown Computer Engineering lab. Two of them were purchased so that each one of us could experiment with them in our own way on our own time without having to worry about conflicting with each other and the classes that use the already available.

VI. Social, Ethical, and Environmental Impacts

A. Social Impacts

Our project doesn't have any direct social impacts. It's a project that many people will find interesting and that could be useful in an educational environment.n.

B. Ethical Impacts

The biggest ethical factor of our project is about the code itself. We decided to keep the project 100% open source. We think this is important for a project like this to allow others to learn from our design. All of the libraries included are also all open source.

C. Environmental Impacts

Our project will have no impact on the environment, as it will be written entirely in software. Hardware that is interfaced with Cosmic such as Raspberry Pis will be completely reusable afterwards.

VII. Design

design tables and stuff copy pasta from design choices + system level design

VIII. Implementation

design philosophy + design implementation

IX. Testing Methodologies

Cosmic was designed to be modular for developers to be able to create their own parts of the system. A byproduct of this also made Cosmic incredibly easy to test. We made a significant effort to automate as much of the testing as possible to verify that the builds we produce are stable on any platform.

While testing Cosmic, we use three primary ways of testing the system, assembler, and any other correlated code.

A. *Unit Testing*

Unit testing is a great way to make sure that parts are working exactly as we expect them to as things get updated and features get added. Currently, the only portion of the project that is unit tested is the Cosmic Processor itself. As of the writing of this document, we have 565 assertions over 54 different test cases that are all passing. These tests check each instruction in a variety of different ways to make sure that they are working as intended. Not only does this provide a great way to make sure the processor is working correctly, but while writing these tests, we found numerous bugs throughout the processor. Listing 1 shows an example of one of the many test cases for the SHLX Instruction.

```
/* 0x4C-0x4F */
TEST_CASE("shlx", "[opcodes]"){
    cosproc proc = cosproc(MemoryRead, MemoryWrite);
    //Imm
    /*
    0000: 4C 00 01 ...
    */
    reset(&proc);
    memory[0x00] = 0x4C;
    memory[0x02] = 0x01;
    proc.r[0] = 0x44;
    proc.r[1] = 0x22;
    proc.cycle();
    REQUIRE(proc.r[0] == 0x88);
    REQUIRE(proc.r[1] == 0x44);
}
```

Listing 1. A unit test for the SHLX instruction.

We set up the processor and memory to properly execute the test case we are trying to replicate. The processor then executes the instructions and the memory, registers, or flags are checked depending on the purpose of the test. For the processor, we used Catch2 to write our unit testing, a multi-paradigm test framework for C++. In the future, the unit tests will be written for the assembler using the pytest framework.

B. *Manual Testing*

Since the majority of the system is not thoroughly tested or is difficult to test (ex. GUI), manual testing is done to ensure that everything is working well and doesn't break. This constitutes us going in and trying to do everything from using the project normally, to breaking the GUI, to running edge cases. While this helps us discover bugs, it also encourages us to think of ways to improve the GUI and the overall usability of our project. While our unit tests focus on individual instructions or functions, our manual tests ensure that the entire system as a whole is working accordingly.

C. *Automated Testing*

Using Travis CI, every time we push an update or change to our GitHub repository, it kicks off several separate builds of the Cosmic system. Cosmic gets compiled and tested on five different environments; x86, ARM, Linux, Windows, and macOS. These builds compile in a clean environment to ensure that the system can compile and run correctly across each platform, not just the systems we are using to design it. The builds start by cloning the repository, then downloading the necessary packages, then finally compiling and running our unit tests. After the builds are complete, we get notified if something causes them to fail so that we know what to fix.

D. *Testing Results*

Using the results of our manual and automated testing, we track bugs and other issues using GitHub. Whenever a bug, undesired behavior, or improvement needs to be made, an issue is opened on GitHub and tracked in our Project tracker. The bug is then cataloged, assigned, and worked on. Once it's verified that the bug is squashed, the issue is then closed.

X. Results

Three introductory labs were created to teach how to write assembly and utilize Cosmic in an educational environment as well as show off some of the features of Cosmic. The labs are generated from a compact and clean LaTeX template, and are easily editable and reusable. They include information such as the lab objective, a prelab section that asks the students to familiarize themselves with specific instructions or other key aspects of Cosmic beforehand, a during lab section that lays out the problem the students must solve, a grading section that breaks down exactly what is required of the student for the lab, and a helpful links section that directs to Cosmic documentation and other helpful places.

A. *Basic Operations*

The first lab asks the student to calculate and the nth Fibonacci number. Since our processor, is only 8 bits, the following constraint is applied: $1 \leq n \leq 13$. This lab teaches the basics of Cosmic, such as using instructions, variables, and loops in the assembler to carry out simple tasks. A sample solution is shown in Listing 2.

```
byte fib = 6
;The Fibonacci number we wish to calculate
byte counter = 2
byte first = 0
byte second = 1
;Initialize the first two numbers in the sequence to
    be 1
byte final = 0
JMP #08
end:
    MOV R0 final
    HCF
    ;End of program
MOV #1 R0
CMP fib
JZS #end
;Check to see if fib is 1
calculate:
    MOV first R0
    ADD second
    MOV second first
    MOV R0 second
    MOV fib R0
    CMP counter
    MOV second R0
    ;Needed since we store R0 in final
    JZS #end
    INC counter
    JMP #11
```

Listing 2. Cosmic assembly to find the nth Fibonacci number.

Since we are calculating the nth Fibonacci number, we stored our n, in this case 6, as fib. A Fibonacci number is calculated by adding together the previous two numbers in the sequence, so we stored the first and second numbers of the sequence accordingly. We defined a counter variable to keep track of which Fibonacci number we are on, and we initialize it to two because if we want the first number in the sequence, we can just return 1 without any calculations. Lastly, we defined a final variable to store our answer in. We then defined an end loop, which is only called after we have generated our answer, to store our final answer in. The MOV, CMP, and JZS instructions following the end of the end loop act as an if statement to check is fib is equal to 1. If it is not, then we head into the calculate loop. This loop moves our first variable into the accumulator and adds the second to it. Then it moves the second variable to the first, and the accumulator to the second. It then checks to see if we have calculated the nth Fibonacci number or not. If we need to continue calculating, the counter is incremented and JMP is called to bring us back to the start of the calculate loop. If the answer has been found, the end loop is called and the program ends after it executes. After the program ends, 8, the 6th Fibonacci number, should be stored in the final variable.

After a program like the one found in Listing 2 is written, it is as simple as clicking Assemble in the Cosmic editor to assemble the code into machine code and load it into the memory editor. All the user has to do afterwards is simply click Run to start executing the program.

B. Basic Graphics

The second lab asks the student to write a program to output a gradient to the Cosmic video screen. Since Cosmic's video output works by reading the video memory for brightness values for each pixel, ranging from 0x00 to 0xFF, the student must loop through the video memory and store either decreasing or increasing values of brightness for the pixels. This lab again reinforces the basics of Cosmic and introduces the usage of the video output. A sample solution is shown in Listing 3.

```
byte brightness = FE
;Start at full brightness -1 so that the program
    terminates correctly.
word start = 8000
;Starting pixel
MOVX #BFFF R0
;Store the end location of the video memory.
loop:
  MOV brightness @start
  ;Set the brightness of this pixel
  DEC brightness
    DEC brightness
    ;Decrement the brightness by two because the
    video out is 64x64
  INCX start
  CMPX start
  JNZ #loop
  ;Repeat until the brightness reaches 0
HCF
;End of program
```

Listing 3. Cosmic assembly to output a gradient to the video screen.

Since Cosmic's video screen is 64x64, we define our initial brightness to be 0xFE. We also define a word, 16 bits instead of 8 bits, starting location for the beginning of the video memory. We then MOV the end location of the video memory to the 16-bit accumulator as a reference. Next, we defined a loop to MOV the current brightness value to the current pixel in video memory. We then decrement the brightness variable twice, since each line of the video screen is only 64 pixels long, and increment our position in video memory. Lastly, we compare our position in video memory to the accumulator, since we stored the end location there, to determine if we should continue drawing or end the program. The outcome is a smooth gradient as can be seen in Figure 1.

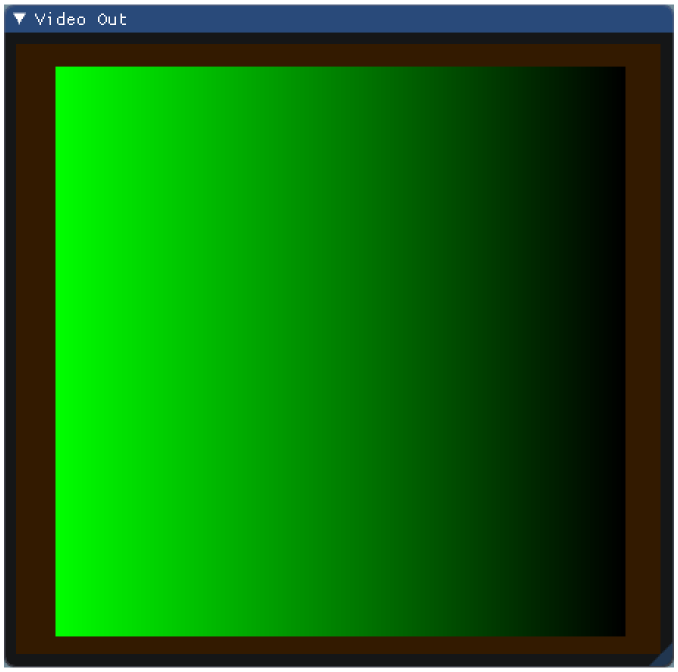

Fig. 1. The gradient produced in Lab 2.

C. Real-world Interfacing

The third and final lab asks the student to write a simple program to read input from a button hooked up to a Raspberry Pi and turn on an LED when input is detected. This lab introduces the student to interfacing Cosmic with something physical like a Raspberry Pi. Even though the task in the lab is simple, it opens up a whole new world of opportunities to explore. In Cosmic, Raspberry Pi GPIO pins are mapped to the values in memory from 0xC402 to 0xC41E. The most significant bit in the GPIO bytes signify whether the pin is an input or an output. The remaining 7 bits set it ON or OFF. If the 7 bits equal zero, then the pin if OFF. If they equal something greater than 0, then it is ON. A sample solution is shown in Listing 4.

```
word inputLoc = C402
word outputLoc = C403
start:
  MOV inputLoc R0
```

```
    AND FF
  JNZ #turnON
    JMP #turnOFF
turnON:
  MOV FF outputLoc
    JMP #start
turnOFF:
  MOV 80 outputLoc
  JMP #start
```

Listing 4. Cosmic assembly to interface with a Raspberry Pi.

Since the pins are mapped directly to memory in Cosmic, we defined two words to store the input and output locations of the pins we want to use. We then defined a start loop that determines if the LED should be turned on or off based on ANDing the input value with 0xFF and jumping accordingly. If the LED needs to be turned on, the loop jumps to the turnON label that MOVs 0xFF to the output location and jumps back to the start loop. If the LED needs to be turned off, the loops jumps to the turnOFF label that MOVs 0x80 to the output location and jumps back to the start loop. This allows the student to infinitely poll the input from the Raspberry Pi.

XI. Conclusion

In the end, we could not be happier with the product we delivered. We committed a significant amount of time to Cosmic throughout the last year to get it to where it is now and learned a lot along the way. We hope that Cosmic, or even just the idea of Cosmic, enables and encourages more people to get involved with the retro-computing community and to learn about low level computer design and functionality.

XII. Acknowledgments

people who helped on the project + dr Li?

References

[1] Schweighauser, M. (2019). Schweigi/assembler-simulator. [online] GitHub. Available at: https://github.com/Schweigi/assembler-simulator [Accessed 7 Sep. 2019].

[2] GitHub. (2019). Commander X16. [online] Available at: https://github.com/commanderx16 [Accessed 13 Sep. 2019].