Cosmic
Testing Report

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

I. Introduction

As described in our Design Report, 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.

II. Testing Methodologies

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 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++. We are currently working on writing unit tests for the now finished assembler and plan to use 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 and 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 download the necessary packages, then finally they compile and run our unit tests. After the builds are complete, we get notified if something causes them to fail so that we know what to fix.

III. 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 one. Once it's verified that the bug is squashed, the issue is then closed.