Cosmic
Preliminary Simulations and Code

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. Overview

During the development of Cosmic, special care has been made for the code to follow the goals of the project. Along with that, all of the code has been made open source for further education and improvement.

II. Progress and Plan

Since the end of the fall semester, we've made many additions to the project. Most of the progress we've made has been in the assembler and the build system. We now have an automated build system in Travis CI that builds, tests, and runs our project on Windows, macOS, and Linux as well as on x86_64 and ARM architectures. We squashed a few bugs in our processor and wrote more tests to verify the validity of the opcodes. The assembler also got a complete rewrite after a few failed attempts. The new assembler design is much more modular and easy to write for.

We also gained a surprising amount of external interest in our project. Over the winter break, we reached out to various communities for advice and feedback on our project. They were very interested in helping us succeed, and one individual even went as far as helping up rewrite our video out since he had much more experience with OpenGL than we do.

Moving forward, there are a few heavy tasks and a few light tasks that we plan to accomplish before SCAD 2020. The heavy tasks include finishing the assembler, which has proven to be more complicated than we anticipated, fully implement video out, and to explore Raspberry Pi integration. Some of the more manageable tasks include finishing the last few instruction unit tests, writing small test programs to show off our architecture, and better interrupt handling. All of these tasks and more can be found in our GitHub project page/issues list.

III. Design Philosophy

A. Modularity

From the beginning, we have been trying to make the project as modular as possible. This is partially to be similar to physical components, where each chip has one purpose. However, the more significant reason is to make the project easily extensible to anyone.

Two examples of modular design are the bus system and our execution cycle. Using callbacks, we have an easily callable memory bus that can be used by anything to quickly and concurrently write and read to memory. This allows a developer creating a new subsystem to communicate with the rest of the system by just calling the memory read and write functions.

```
Write(0xFF,0x1000); //Writing to Memory
Read(0x5000) //Reading from Memory
```

Listing 1. Example of the Memory Bus being called

The second example, the execution cycle, allows for a developer to easily add new opcodes to the processor in just a few lines of code. All that needs to be added is an entry into the instruction set and a function to execute.

```
InstructionSet[0x02] = (Instruction){&cosproc::IMP,&
    cosproc::PUSH,"PUSH",1};

...

void cosproc::PUSH(uint16_t src){
  Write(sp,r[0]);
  sp--;
}
```

Listing 2. Example of instruction set entry and function

In addition to the processor, the assembler is also built similarly. This allows the new opcodes to be easily added to the assembler as well. The modular design of the assembler also ended up being much easier to write and much more flexible.

B. Readability

More emphasis was put on writing code that was readable and easy to understand above performance. The project isn't made to be a high-performance application, and there is very little that can be done in the software that will stress a modern computer that would require optimization. Instead, readable code accomplishes our desire to make the code easy to understand and extend.

C. Automation and Testing

We have included extensive testing and automatic CI/CD for our project. After every push upstream, Travis CI kicks off a series of builds that compile, test, and run the program on x86_64 Windows 10, macOS 10.14, and Ubuntu 16.04, along with Ubuntu 16.04 on ARM.

IV. Design Implementation

All of our code can be found on Github at: https://github.com/clbx/Cosmic

A. Cosmic Processor Cycle

The cosmic processor works primarily off function pointers to call the proper opcode and carry out the instruction. This loop is shown in listing 3.

```
cosproc::Debug cosproc::cycle(){
  uint8_t opcode = Read(pc); //Fetch
  Instruction currentInstruction = InstructionSet[
    opcode]; //Decode
  execute(currentInstruction); //Execute
  Debug debugPackage;
  debugPackage.pc = pc;
  debugPackage.instruction = currentInstruction;
  pc += currentInstruction.bytes; //Writeback
  return debugPackage;
}

void cosproc::execute(Instruction i){
  uint16_t src = (this->*i.addressing)();
  (this->*i.opcode)(src);
}
```

Listing 3. Instruction Execution Loop

In this loop, the desired opcode is read from memory and decoded in the instruction set. The instruction set gives an Instruction type that holds vital information about the instruction like size, mnemonic, and, most importantly, the function pointers for the addressing mode and instruction. The function is then executed by first running the addressing mode function, which returns the place in memory where the instruction will do its operation. After execution, a new debug package is made to return information to the debug console, and is built using the data from the Instruction Set. The program counter is then increased by the amount given from the Instruction Set, and the debug package is returned to the environment.

B. Cosmic Processor Addressing and Execution

Once the execution portion of the cycle is called, the system first calls the specified addressing mode function. This function returns a place in memory for the opcode to look at. This was made to keep the processor modular if more addressing modes were ever to be added.

```
uint16_t cosproc::ABS(){
  uint16_t val =  (Read(pc+1) << 8 | Read(pc+2)); //
    Return 16bit address of where to look at data
  return val;
}
```

Listing 4. The Absolute Addressing Mode

Once the position in memory is found, the function opcode is called.

```
/* 0x10-0x12 ADD */
void cosproc::ADD(uint16_t src){
  uint8_t data = Read(src);

  unsigned int temp = r[0] + data;

  //Set Negative
  st[1] = temp >= 0x80;
  //Set Carry
  st[2] = temp > 0xFF;
  //Set Overflow
  st[3] = ((r[0]^temp)&(data^temp)&0x80) != 0;

  //Set Value
  r[0] = temp & 0xFF;

  //Set Zero
  st[0] = r[0] == 0;
}
```

Listing 5. The ADD function

The function shown in listing 5 is for the ADD opcode. This takes a number and adds it to the accumulator and sets flags accordingly.

C. Environment

The environment is primarily driven by ImGui, an immediate open-source graphics library. This has been fundamental to developing our GUI as it makes an easy way to see everything that is going on in the machine. ImGui uses OpenGL for hardware rendering so that it runs flawlessly on any modern computer even under intense load. The GUI is easily extensible as well.

D. Assembler

The assembler has a similar overall design to the processor cycle. An opcode is given, which resolves to a function to execute. The assembler has a few steps along the way since opcodes are not the only thing it encounters. Every line is processed individually in the assembler, the first thing that happens is it gets tokenized into individual parts then fed into the main logic loop. This checks to see what kind of input the line is. It will resolve it to a variable assignment/change, a label, a comment, or an opcode, as seen in listing 6.

```
def assemble(tokens):
    #print(tokens)
    #If its variable creation:
    if(tokens[0] in types and tokens[2] == "="):
        createVar(tokens)

    #If its variable assigning:
    elif(tokens[0] in variableTable and tokens[1] ==
     "="):
        updateVar(tokens)

    #If its a label
    elif(tokens[0][-1] == ":"):
        handleLabel(tokens)

    #If its a comment
    elif(tokens[0][0] == ";"):
        pass

    #Else its an opcode
    else:
        tokens = resolveVariables(tokens)
        try:
            eval(tokens[0])(tokens)
        except NameError:
            error("Unknown Input {}".format(tokens))
```

Listing 6. Assembler Logic

If the type of input is determined to be a variable creation, an entry is made into the variable table to keep track of its size and value. This is eventually added to the variable memory space on the final assembly.

If the type of input is determined to be a variable assignment. It will be replaced with a MOV instruction that will update the variable space.

If the type of input is determined to be a label, the current position in memory and name are recorded in the label table. Any further references to the label will be replaced with the memory position.

If the type of input is determined to be a comment, it is ignored.

If the type of input is determined to be an opcode, the function correlated with this opcode is executed, and the machine code is added depending on the process of that opcode.

E. Example Cosmic Code

Currently, Cosmic assembly is very subject to change as we add new features. Listing 7 has an example of Cosmic assembly code.

```
start:
    ;set up our variables
    byte n = 50
    word prevprevnumber = 0
    word prevnumber = 0
    word currnumber = 1
loop:
    ;get the fibbonacci number
    MOV prevnumber prevprevnumber
    MOV currnumber prevnumber
    MOV prevprevnumber R0
    ADD prevnumber
    MOV R0 currnumber
    ;increment the counter
    MOV n R0
    DEC
    CMP #0
    ;jump back if n is not yet 0
    JZS loop

    ;Fibonacci number will be in currnumber
```

Listing 7. Cosmic Assembly to find the Nth Fibonacci Number

This snippet of code will find the Nth Fibonacci number and put it in the memory location at currnumber.