

Cosmic

Final Simulation and Real-time 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. ASPIRATIONS

We introduced Cosmic as a fully simulated 8-bit micro-computer architecture that was designed to act like a physical chip, but be completely software based. We aspired to write everything from an instruction set, to an operating system for it, alongside a rich GUI environment and an assembler. Some of the early design constraints included a RISC-like instruction set containing less than 50 total instructions, the design to be derived from the Zilog Z80 and MOS 6502 microprocessors, a budget of \$0, and that timing would not be a constraint.

As development went on, our focus shifted to designing Cosmic to be a tool to learn both assembly and low level computing. The instruction set and architecture were mostly flushed out by this point, so we turned to improving the GUI environment, writing the assembler, and even writing some introductory labs that could be used to teach how Cosmic works.

In the end, all of our original design constraints were met except for the \$0 budget. We purchased a Raspberry Pi that was powerful enough to run Cosmic so that we could create and test a way for Cosmic to interface with the real-world through the GPIO pins on the Pi. Everything, including our architecture, GUI environment, and assembler, is currently in a working state.

II. REAL-TIME CODE

Three introductory labs were created to teach how to write assembly and utilize Cosmic in an educational environment. 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 aspect 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.

All of the labs, their accompanying solutions, and the rest of the Cosmic source code can be found in our [GitHub repository](#).

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 1.

```
1 byte fib = 6
2 ;The Fibonacci number we wish to calculate
3 byte counter = 2
4 byte first = 0
5 byte second = 1
6 ;Initialize the first two numbers in the sequence to
   be 1
7 byte final = 0
8 JMP #08
9 end:
10     MOV R0 final
11     HCF
12     ;End of program
13 MOV #1 R0
14 CMP fib
15 JZS #end
16 ;Check to see if fib is 1
17 calculate:
18     MOV first R0
19     ADD second
20     MOV second first
21     MOV R0 second
22     MOV fib R0
23     CMP counter
24     MOV second R0
25     ;Needed since we store R0 in final
26     JZS #end
27     INC counter
28     JMP #11
```

Listing 1. 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 if 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 1 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 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 2.

```

1 byte brightness = FE
2 ;Start at full brightness -1 so that the program
  terminates accordingly.
3 word start = 8000
4 ;Starting pixel
5 MOVX #BFFF R0
6 ;Store the end location of the video memory.
7 loop:
8   MOV brightness @start
9   ;Set the brightness of this pixel
10  DEC brightness
11   DEC brightness
12   ;Decrement the brightness by two because the
    video out is 64x64
13  INCX start
14  CMPX start
15  JNZ #loop
16  ;Repeat until the brightness reaches 0
17 HCF
18 ;End of program

```

Listing 2. 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 is OFF. If they equal something greater than 0, then it is ON. A sample solution is shown in Listing 3.

```

1 word inputLoc = C402
2 word outputLoc = C403
3 start:
4   MOV inputLoc R0
5   AND FF
6   JNZ #turnON
7   JMP #turnOFF
8 turnON:
9   MOV FF outputLoc
10  JMP #start
11 turnOFF:
12  MOV 80 outputLoc
13  JMP #start

```

Listing 3. 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 loop 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.

III. OUTCOME

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.