N2EMU

Requirements Analysis Document

Team 09
Bryan Borer
Alex Chmelka
Dennis Feng
Dominic Hezel
Tyler Zinsmaster

Introduction

Purpose

The client, Exciting Emulators, requested an emulator for the NIOS II processor, which can interpret the majority of the assembly functionality and behavior of the NIOS II processor accurately, with exceptions as specified.

Scope

This emulator will be able to run all assembly of the assembly commands specified in the intel NIOS II manual, with the exception of those instructions involving cache management and floating point commands.

The emulator does not aim to fully reproduce any hardware quirks or I/O devices on real NIOS II systems, only the base functionalities of the processor, at a high level. So, only in regards to behavior rather than the physical hardware or logic characteristics.

The program will feature a simple yet functional GUI in order to fulfill the success criteria. This GUI includes a virtual I/O system including a HEX display and switches in order to emulate a what a real, physical display would function as.

Objectives and Success Criteria

Objectives: Properly emulate all specified behaviors of a NIOS II processor, allow interaction through a functional GUI.

Success Criteria: This project is considered successful once it fulfills all functional and nonfunctional requirements involved in the emulation of a NIOS II processor's behavior through a useful interface.

Definitions, Acronyms, and Abbreviations

NIOS: Netware Input/Output System. A brand of processors made by Intel, whose assembly language and behavior this project will interpret and emulate the second generation of.

Emulator/Emulation: For the context of this project an emulator can be defined as a program or system which wishes to recreate (emulate) the behavior of another system. In this case, the project will be emulating the behavior of the NIOS II processor.

GUI: Graphical User Interface. What the user sees when interacting with the program.

References

CSCE 361 Increment Instruction sheets

Intel NIOS II manual:
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

JNIOSEMU (Existing Java Based NIOS II Emulator): https://stpe.github.io/jniosemu/

Overview

Our program roughly matches the MVC architecture.

Our program revolves around three main systems, the Processor, GUI, and Behavioral systems. The processor does the meat of the computation, executing assembly instructions, storing registry/memory values, and having processor state control functionality. The Behavioral system largely exists to feed the processor instructions loaded from a file as well as relevant information from command line input. At its base, it gives the processor a line by line reinterpretation of assembly code to execute. However, it also takes command line inputs from the user, and accepts a program file as loaded in by the user. The GUI has two parts, Input and Output. The Input is made up of buttons and a command prompt. The buttons allow for the access of processor state control functionality by the user in GUI, for pausing, running, stepping, resetting, and for loading text files into the behavioral core. When the processor needs to be paused, resumed, or stepped, it takes input from the a button on the GUI. The command prompt allows for the input of commands which are passed to the Behavioral system. The Input also includes switches which change the values of a specific memory address specified in the GUI. The Output allows the viewing values through a register box, memory viewing box, and miscellaneous box, which are updated by the processor as instructions are performed. The Output also includes a HEX display which can be changed through an assembly program by manipulating the specified memory values in the GUI.

See general class descriptions below:

USER - The user will upload the assembly file to the main system, start and pause the emulator, and use the command line to set breakpoints and set points in memory to view.

BEHAVIORAL CORE- The behavioral core is the "brain" of the program, which determines how the different subsystems interact to scenarios. The behavioral core will take the assembly code and break it down to individual instructions. Will not work with incorrect syntax, but will accept any .txt file. The main component of the “model” portion of the MVC architecture.

INSTRUCTION INTERPRETER- The instruction Interpreter takes in instructions from the behavioral core, identifies the instruction, as well as input and output values.

PROCESSOR - The processor will run the identified instructions, updating the registry, memory values, and other relevant processor values, as long as it is in the “running” state. The main component of the “controller” portion of the MVC architecture.

THREADS - Threads are a class within the program which provide multithreading functionality. When the program is run, two main threads start, for the processor and for the GUI.

LOADFILE- This class exists purely to break an input text file into individual lines for use by the other classes.

GUI: This class is the window of communication between the user and the internal architecture. It includes buttons, switches, a HEX display, and various text boxes which are all described below, within the red-dashed section.

COMMAND PROMPT: The command prompt accepts user input from keyboard, and allows for setting up to 16 memory locations to view in the memory box viewing window, In addition, breakpoints can be set per line so that the system pauses when desired. Breakpoints can also be removed per line.

BUTTONS:

Load File: Brings up a file select prompt, accepts .txt files, and sends them to the behavioral core.

Run: The run button will run the program only once the processor is no longer in the NOT READY state (the file has been input). The run button also starts the program again after a break or pause-induced pause.

Pause: The pause button stops the running of the program.

Reset: The reset button returns the program counter to zero, resets registers and memory.

Step Up: The step-up button is used to increment the program counter to the next instruction, for debugging purposes.

Browse: This button opens a file navigation window, where you can manually select the file to load into the program.

Execute: This button executes command line commands.

Write .txt : Creates subfolder logs/ if it doesn’t exist, writes relevant logging information to a .txt file, of which an unlimited number can be made. You can clear all of the log files with rmLogs command line command.

SWITCHES:

Can be toggled on and off while the program is running, so as to set the values of corresponding locations in memory.

BOX VIEWING WINDOWS:

Memory: This window will keep track of up to sixteen memory addresses and show the hexadecimal value contained at each memory address.

Registry: The registry box window lists each register and its corresponding hexadecimal value.

Text File: This window shows the code which has been loaded as a text file within the emulator.

Misc.: Includes viewing for the Program Counter, Return Address, Processor State, and Breakpoint Lines.

File Path:This box shows the path to the file we will load. Can be edited to specify path via keyboard input.

Message Box: Contains errors and Messages that will pop up during program execution, defined within the programs. Can be cleared via the command line cmes command.

7 SEGMENT HEX DISPLAYS:

These exist to display the values of the last 4 hex digits at the specified memory location 1000, in 4 individual 7 segment implementations.

Current System

Functional Requirements

F1. Mimic subset of externally observable behavior of the processor as specified.

F2. Read a text file that contains labels and assembly instructions, one per line.

F3. Memory is to be represented as a 64 kilobyte byte-addressable block of memory. (An array of 65,536 bytes). The four contiguous bytes stored at up to 16 memory addresses at any given time should be viewable as addressed by the user.

F4. The processor is a 32-bit little-endian processor. Memory addresses only require 16 bits. The program counter and the return address indicate the line of the file that contained the assembly instructions. (For example, branch and jump commands)

F5. The contents of the program counter and the 32 general-purpose registers are to be displayed to the user. (In Hexadecimal)

F6. The program should indicate that the processor is “NOT READY”, “PAUSED”, or “RUNNING”. NOT READY specifies that no file has been loaded. When a file is loaded, the processor is PAUSED and a “RUN” button can be pressed to begin program execution. At any point, a “PAUSE” button can be pressed to put the program back into the PAUSE state.

F7. Breakpoints can be specified per line by the user, and when program execution reaches a breakpoint, the processor is placed automatically into the PAUSE state.

F8. A reset button can set the program counter back to 1.

F9. Error handling for incorrect assembly code shall be implemented.

Nonfunctional Requirements

N1. I/O devices need not be emulated.

N2. We do not need to implement backstepping outside of that naturally contained in reset functionality.

N3. Only the behavior of the processor needs to be emulated.

System Models

Scenarios

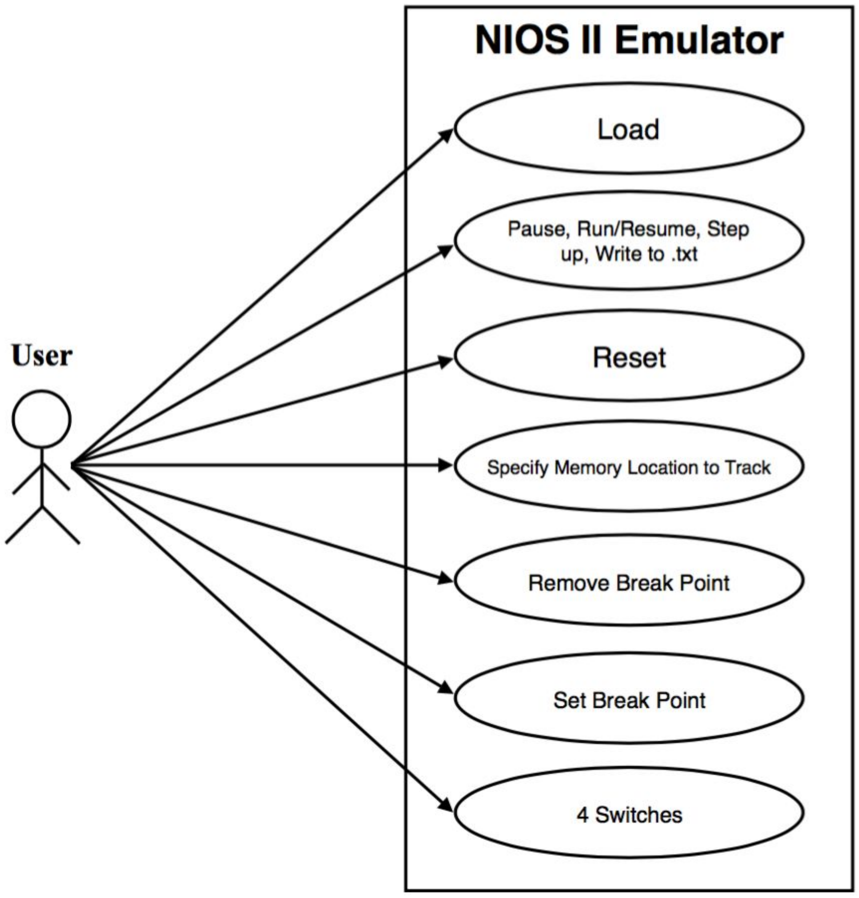
Scenario 1 - File Execution

1. Dennis is a sophomore Computer Engineering student at the University of Nebraska-Lincoln, is late to his 6:30 PM lab section for his Computer Organization class. However, he has forgotten his NIOS II programming circuit board on which he is supposed to run his code. Instead of going home to get his board, he decides to use a friend’s emulator to run his assembly code.
2. First, Dennis writes his assembly code into a text file and saves it on his computer. He then presses the Browse button on the emulator’s GUI and selects his file. He then presses the Load button.
3. As soon as the file is uploaded to the emulator, the BEHAVIORAL CORE parses each line and sends them into the INSTRUCTION INTERPRETER to get ready for execution.
4. Dennis text file is this assembly code:

```
add r0, r0, r0

orhi r1, r0, 8

ori r2, r1, 0x10

ori r3, r1, 0x40

ori r4, r1, 0x50

addi r5, r0, 1

loop: ldw r6, (r3)

stw r6, (r1)
stw r5, (r2)

ldw r6, (r4)

beq r6, r7, loop

add r7, r6, r0

addi r5, r5, 1

br loop
```

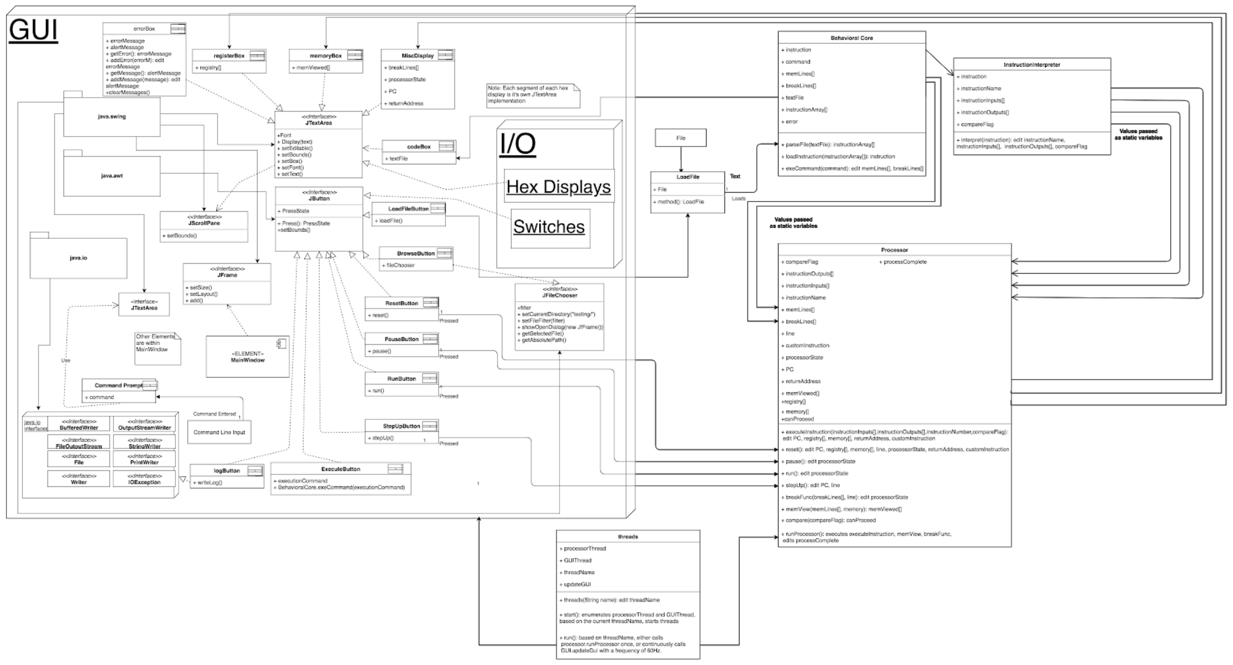
5. Dennis presses the RUN button to execute every line of assembly code.
6. The PROCESSOR updates the relevant GUI elements.
7. Dennis checks the GUI elements to make sure his values are correct.
8. Dennis checks the MEMORY BOX to make sure that he stored his own data values in memory correctly.

Use Case Model

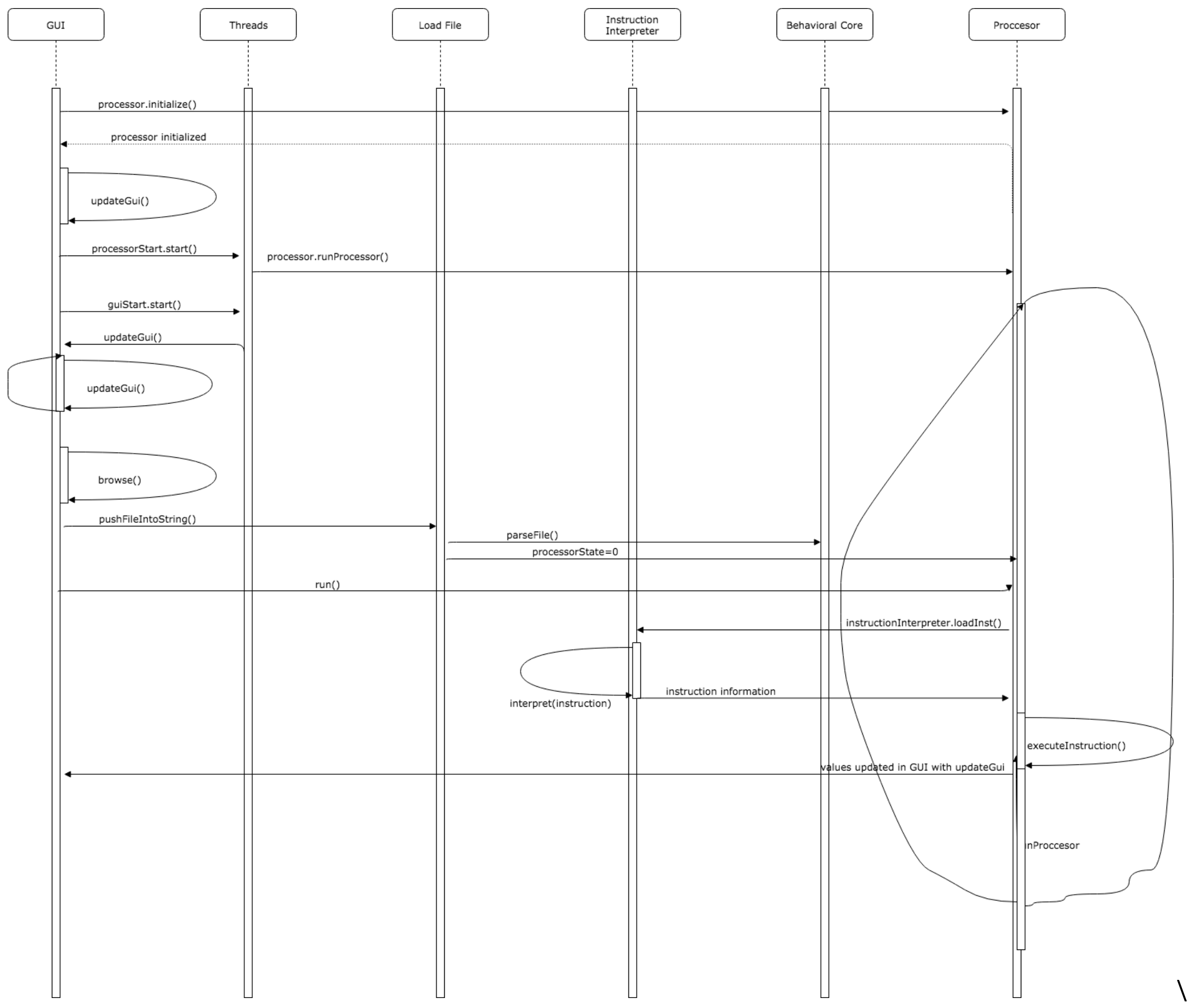

NIOS II Emulator
User
Load
Pause, Run/Resume, Step up, Write to .txt
Reset
Specify Memory Location to Track
Remove Break Point
Set Break Point
4 Switches

Object Model

N2EMU Class Diagram


GUI
errorBox
+ errorMessage
+ alertMessage
+ getError(): errorMessage
+ addError(errorM): edit errorMessage
+ getMessage(): alertMessage
+ addMessage(message): edit alertMessage
+clearMessages()
registerBox
+ registry[]
memoryBox
+ memViewed[]
MiscDisplay
+ breakLines[]
+ processorState
+ PC
+ returnAddress
Note: Each segment of each hex display is it's own JTextArea implementation
java.swing
java.awt
java.io
<<interface>> JTextArea
+Font
+ Display(text)
+ setEditable()
+ setBounds()
+ setBox()
+ setFont()
+ setText()
codeBox
+ textFile
<<interface>> JButton
+ PressState
+ Press(): PressState
+setBounds()
LoadFileButton
+ loadFile()
BrowseButton
+ fileChooser
<<interface>> JScrollPane
+ setBounds()
<<interface>> JFrame
+ setSize()
+ setLayout()
+ add()
«interface» JTextArea
Other Elements are within MainWindow
«ELEMENT» MainWindow
Use
Command Prompt
+ command
Command Entered
1
Command Line Input
java.io interfaces
<<interface>> BufferedWriter
<<interface>> OutputStreamWriter
<<interface>> FileOutputStream
<<interface>> StringWriter
<<interface>> File
<<interface>> PrintWriter
<<interface>> Writer
<<interface>> IOException
logButton
+ writeLog()
ExecuteButton
+ executionCommand
+ BehavioralCore.exeCommand(executionCommand)
ResetButton
+ reset()
PauseButton
+ pause()
RunButton
+ run()
StepUpButton
+ stepUp()
Pressed
1
I/O
Hex Displays
Switches
<<interface>> JFileChooser
+filter
+ setCurrentDirectory("testing/")
+ setFileFilter(filter)
+ showOpenDialog(new JFrame())
+ getSelectedFile()
+ getAbsolutePath()
File
LoadFile
+ File
+ method(): LoadFile
Text
Loads
Behavioral Core
+ instruction
+ command
+ memLines[]
+ breakLines[]
+ textFile
+ instructionArray[]
+ error
+ parseFile(textFile): instructionArray[]
+ loadInstruction(instructionArray[]): instruction
+ exeCommand(command): edit memLines[], breakLines[]
InstructionInterpreter
+ instruction
+ instructionName
+ instructionInputs[]
+ instructionOutputs[]
+ compareFlag
+ interpret(instruction): edit instructionName, instructionInputs[], instructionOutputs[], compareFlag
Values passed as static variables
Processor
+ compareFlag
+ processComplete
+ instructionOutputs[]
+ instructionInputs[]
+ instructionName
+ memLines[]
+ breakLines[]
+ line
+ customInstruction
+ processorState
+ PC
+ returnAddress
+ memViewed[]
+registry[]
+ memory[]
+canProceed
+ executeInstruction(instructionInputs[],instructionOutputs[],instructionNumber,compareFlag): edit PC, registry[], memory[], returnAddress, customInstruction
+ reset(): edit PC, registry[], memory[], line, processorState, returnAddress, customInstruction
+ pause(): edit processorState
+ run(): edit processorState
+ stepUp(): edit PC, line
+ breakFunc(breakLines[], line): edit processorState
+ memView(memLines[], memory): memViewed[]
+ compare(compareFlag): canProceed
+ runProcessor(): executes executeInstruction, memView, breakFunc, edits processComplete
threads
+ processorThread
+ GUIThread
+ threadName
+ updateGUI
+ threads(String name): edit threadName
+ start(): enumerates processorThread and GUIThread, based on the current threadName, starts threads
+ run(): based on threadName, either calls processor.runProcessor once, or continuously calls GUI.updateGui with a frequency of 60Hz.

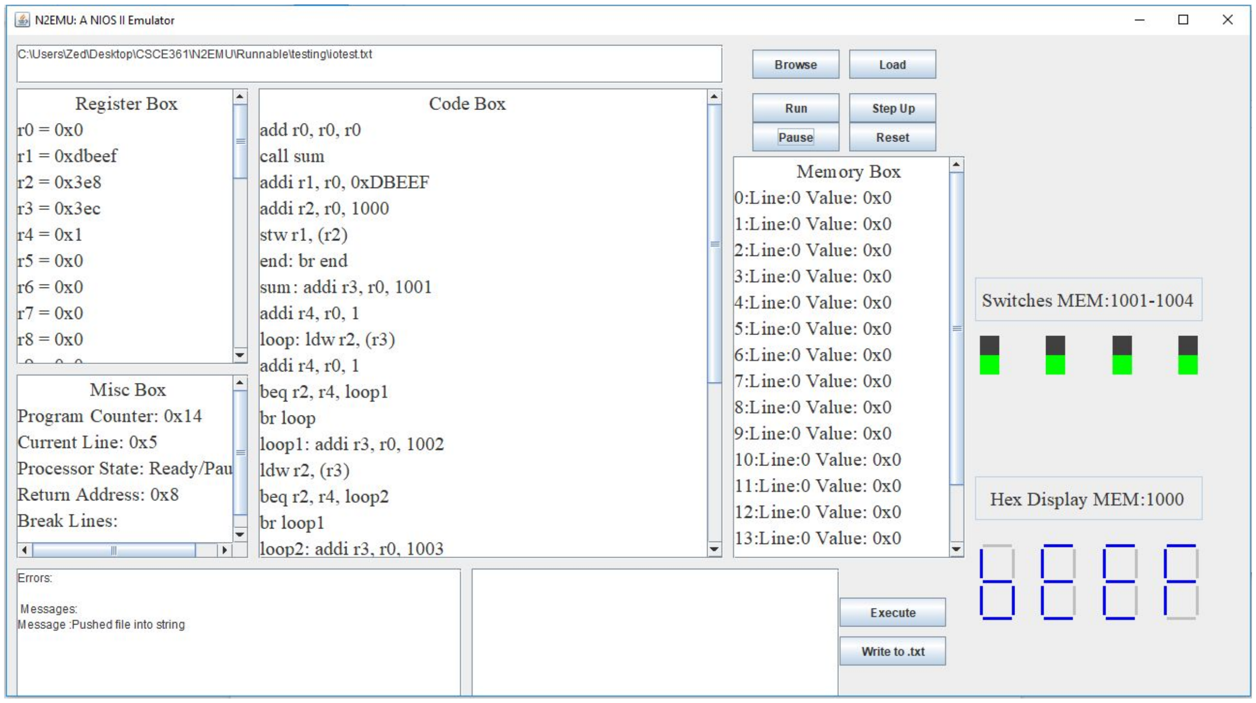
Dynamic Model



Above figure is the “File Execution” sequence Diagram

User Interface: Navigational Paths and Screen Mockups

Program Screen



C:\Users\Zed\Desktop\CSCE361\N2EMU\Runnable\testing\iotest.txt

Register Box

```
r0 = 0x0
r1 = 0xdbeef
r2 = 0x3e8
r3 = 0x3ec
r4 = 0x1
r5 = 0x0
r6 = 0x0
r7 = 0x0
r8 = 0x0
```

Code Box

```
add r0, r0, r0
call sum
addi r1, r0, 0xDBEEF
addi r2, r0, 1000
stw r1, (r2)
end: br end
sum: addi r3, r0, 1001
addi r4, r0, 1
loop: ldw r2, (r3)
addi r4, r0, 1
beq r2, r4, loop1
br loop
loop1: addi r3, r0, 1002
ldw r2, (r3)
beq r2, r4, loop2
br loop1
loop2: addi r3, r0, 1003
```

Memory Box

```
0:Line:0 Value: 0x0
1:Line:0 Value: 0x0
2:Line:0 Value: 0x0
3:Line:0 Value: 0x0
4:Line:0 Value: 0x0
5:Line:0 Value: 0x0
6:Line:0 Value: 0x0
7:Line:0 Value: 0x0
8:Line:0 Value: 0x0
9:Line:0 Value: 0x0
10:Line:0 Value: 0x0
11:Line:0 Value: 0x0
12:Line:0 Value: 0x0
13:Line:0 Value: 0x0
```

Misc Box

```
Program Counter: 0x14
Current Line: 0x5
Processor State: Ready/Pau
Return Address: 0x8
Break Lines:
```

```
Errors:

Messages:
Message :Pushed file into string
```

Proposed System

No Change

Glossary

Break Point - A point in an assembly program that the use chooses for the program to pause when it reaches that point.

Behavioral Core - The brain of the program. This interacts will all of the other classes in the program.

MVC (Model View Controller) Architecture - A software architecture that breaks up

Thread - A thread is the program’s path of execution.