Computer Emulator:

Project Planning Document

Written By: Stephen Wilson, Sean Perts, and Jonathan Erickson

Table of Contents:

Section 1: Introduction

1.1 Document Purpose

This document will be used by the client, Gusty Cooper, to know and understand the projected schedule of the project. This project is intended to benefit future CPSC 305 students; they will use it to gain a better understanding of how computers operate at a low level. This document will estimate the amount of time to design and develop the application's functions.

1.2 System Scope

The emulator application is a low-level simulation of a computer. It will show assembly code translated into machine code, and show how that code is moved throughout the memory and registers of a simulated computer. It will also demonstrate how ARM instructions are fetched, decoded, and executed. This will all be done through a Graphical User Interface. The application will be used in the classroom as a teaching aid. It will be utilized to teach Computer Systems & Architecture (CPSC 305) students at the University of Mary Washington.

1.3 References

1.3.1 Gusty's Requirements Document

Professor Gusty provided the implementation team with a document of his requirements for the project.

1.3.2 Gusty's Preliminary Code

Professor Gusty provided the implementation team with preliminary code. He structured the code to provide a basic framework for the project.

1.3.3 Gusty's ARM Assembly Presentation

Professor Gusty provided the implementation team a detailed presentation of the basic function of ARM Assembly.

1.4 Overview

The remainder of the document is divided into various sections in order to delve into the details of what the application will accomplish, the required and non-required functions stated by the client, the system abilities necessary to run the application, and ending with additional resource information. Section two describes the application and how it can be applied in greater detail. It introduces the client and their intentions for the product, as well as the constraints on the project. Section three contains the project schedule. This includes the approach, milestones, a work breakdown structure, a gantt chart, and a task dependency chart. Section four contains the appendix, which includes a glossary, authoring information, and links to additional documents.

Section 2: Project Description

2.1 System Overview

The application will perform basic data manipulations in order to provide the user with insight as to how information is moved and stored to the memory or to registers by the Central Processing Unit of the system. The program will allow users to create and run programs written in the ARM Assembly language. Users will be able to view different translations of the ARM Assembly, such as hexadecimal and binary. Users will be able to run their program by stepping through each line of their code or through multiple lines at once. In doing so, users will be able to see a display of how their code is allocated to cache, memory, and how it is manipulated on the processing registers of a computer. In addition to this basic functionality, users will be able to interact directly with the application.

The user will also be provided with the ability for more advanced interaction with the application, to accommodate the user's progression in their understanding of computer functionality. The user will be able to load memory from a file or set memory individually for any and all sixteen 32-bit registers. The application will also allow for programs written in ARM Assembly, hexadecimal or binary to be loaded into specific addresses in memory as well. In addition to loading and storing data, the user will be able to experiment with breakpoints, branching, the passing of parameters, stacks, heaps, and ARM Assembly operation and condition codes. These more complex functions will help give the user a deeper grasp of how their computer functions on a low-level.

2.2 Client Characteristics

Gusty Cooper is a professor of six years at the University of Mary Washington. As a mentor in the Computer Science department, Professor Gusty Cooper teaches classes focused around object oriented analysis in the Java Programming language.

In the coming Fall semester at the University, Gusty will be instructing a course focused around Hardware Assembly. One of the many concepts taught will be Assembly Language, being largely focused around the ARM processor. To use as a teaching aid, Professor Gusty Cooper would like a software that visibly emulates the behavior of the ARM processor. The program will simulate, step by step, the loading and storing of memory as a program written in the ARM assembly language is loaded into the simulator. Professor Gusty Cooper's hopes are that his students will be able to form a better understanding of low-level computing concepts introduced throughout the course with the help of this visual aid.

2.3 User Characteristics

The intended users for this application are the students who enroll in the University's *Computer Systems & Architecture* course. This 300-level Computer Science course is primarily attended by students who are pursuing a major or minor degree in the Computer Science department. These students will be learning the basic and low-level operation of computers. Students will learn the material from a bottom up approach, starting at the circuit level and proceeding to assembly language. The program will be a beneficial asset as a visual aid to instruct students and allow them an interactive approach to learning the concepts explored in the course.

2.4 Functional Requirements

1. The application's GUI must be constructed with JavaFX.
2. The application must be developed following an object-oriented design schema.
3. The user must be able to load ARM assembly and machine code in hex or binary into the application.
4. The user must be able to load a program from a file into main memory.
5. The application must allow for portions (or the entirety) of main memory to be filled with values from an external file.
6. The user must be able to step through the ARM assembly code one step at a time.
7. The application must respond to each step requested by the user by placing the values in the correct register cell, memory cell, or cache cell.
8. The user must be able to set breakpoints in the ARM assembly code console so when the application runs the code, filling the registers, memory, and cache, it will stop at the breakpoints.
9. The application must be able to emulate the load/store byte/word, data processing, ARM instructions.
10. The user must be able to specify how many lines of code to play through at one time. For example, when the user selects the multiple step button, the application must be able to go through the user specified number of steps.
11. The user must be able to click a reset button which will then require the application to reset the program counter back to the first line of the program, and clear the memory for the program to restart again.
12. The application must return focus to the input box after a program has finished running.
13. The user must be able to change the values of the cells in the registers and memory.
14. The user must be able to see the contents of the main memory, caches, and registers changing in real-time.
15. When the user clicks the stop button, the application must be able to stop the execution of the program.
16. The user must be able to save the ARM assembly code to an external file.
17. The user must be able to save the memory to an external file.
18. The application must be able to save what is inputted by the user for the next run through of the program.
19. The user should be able to access an error log if the application fails or errors out at any time.

20. The application should be able to highlight what is changing in the registers, cache, and memory as the code is running in real time.

2.5 General Constraints

1. The application will have no security features as they are not necessary.
2. The application will not need access to networking of any kind.
3. The application is only responsible for handling the following ARM instructions: Data processing, Load/Store Byte/Word, and Branch.
4. The application will not need to handle the following ARM instructions: multiply, long multiply, swap, load/store multiple, halfword transfer, branch exchange, coprocessor data transfer, coprocessor data operation, coprocessor register transfer, and a software interrupt.

Section 3: Project Schedule

3.1 Approach

3.1.1 Backend for ARM assembly: A lot of time will be spent on fully learning ARM'S behavior, mainly through experimentation as well as looking at documentation. As a team, we have come up with a plan for reasonable organization within the emulated ARM infrastructure.
Additionally, ARM Assembly has many functions and operands to manipulate the registers. Each of these op codes must be simulated properly, including branching, loading registers, bitwise shifts, and arithmetic sequences. These will be able to accomplish many tasks to manipulate data, including printing strings, creating variables, and doing what a high level programming language otherwise would be capable of doing.

3.1.2 File Input/Output, Save/Load: 3: Still to be decided. We will structure the file input and output to support 32 bit ARM syntax. If the files are to be read line by line, this task should not be the most difficult(Side note: If no file, might be a lot of errors until we check. Create a new file so that is blank.).
The largest issue will be parsing. While parsing itself is fairly simple, parsing in such a way that is efficient becomes more difficult. The next important part of this would be to separate the code from the variables, as well as understand what the different variables in assembly language mean. The three main sections are the

.text, .bss, and .data sections which differentiate different variable types. These will need to be analyzed separately from the actual code, which itself is declared by the .text section. From that point onwards, the main issue becomes understanding how many arguments there are per line based on the initial instruction. Besides that, saving should be little issue besides busy-work, taking some amount of time to complete.

3.1.3 GUI- Use JavaFX: After implementing the backend of the program, we will start by creating a basic window. Following this, we will implement several panes to represent registers and to display the manipulation of data. Another task involved will be a "Step" button used to step through each phase of data manipulation in ARM (Load, Add, etc).
The most difficult part of the GUI will be getting used to the updated JavaFX library. Because it is understanding a library rather than understanding assembly language, there should be few issues with this. There will be a number of visual bugs more than likely, but there is little trouble in quickly looking up solutions online for how the library works.

3.2 Milestones and Deliverables

ARM BACKEND FUNCTIONALITIES:

- **Create 16 registers**
  An emulated set of 16 registers will be implemented and duplicate the functionality in a java environment.

Get Load to function

  - Arm assembly program. Registering Syntax.
- **Conditionals**
  - Compare
    - Two registers: Return a conditional operator =, !=, >, <
  - Take operators
- **Branching**
  - Unconditionally: Goto statement(branch to this label)
  - Conditionally: If compare returns true, goto label.
  - Store value of where to return to after the branch in reg 14.
- **Get Add, Subtract, multiply, divide, and other OPCode instructions to work**
  - Parts of this have been supplied in sample code from the client.
- **Creating working command line version**

-this will be the primary functioning bit of the program. This needs to be fully functional(mostly functional) before graphics can be fully implemented

FILE IN/OUT

- **Parse normal ARM file**
- **ARM file is assumed to be 32 bit syntax.**
- **Data must be saved to Java Backend.**
- **Output a copy of the program,**
- **Give Text File.**
- **Store in doubly linked arraylist.**
- **Save instruction components into Java Nodes.**
- **Once loaded into the arraylist, go through and begin executing.**
- **Save state of program at specific point (middle, end, etc)**

FRONTEND/GRAPHICS:

- **Functional GUI**

-Graphical user interface. This modest visual aspect will follow after the core functionality of the program is running in a comprehensible fashion.

- **Create Window**
    - Use a universally accepted ratio.
- **Instruction display**
- **Another Section displays registers.**
- **Third section: Blank. Use case: Click a register and display in this box**
- **Control Section: Step/Save/**

3.3 Work Breakdown Structure

3.3.1 Create 16 Registers: March 12 - 13 Registers must be made before Add and Subtract. About 5 hours of work(We need this before anything else. Whole team will work simultaneously on this before individual assignments). Registers can be easily emulated using java structures.

3.3.2 Memory storage: March 13 - 14 This section will mostly involve creating the stack, heap, and creating a method of simulating stored memory in such a way that it is accessible and easily understood. In java memory can be emulated using Arrays.

3.3.3 Get Load to Work: March 18 - 19 This would be handled after the registers are made, and would be the ideal next goal, approximately 3-4 hours estimated.

3.3.4 Conditionals: March 15-18 returning of operators for next function. In theory, should not take terribly long to implement, approximately 2-3 hours.

3.3.5 Branching: March 16-18 Should be implemented after conditionals are operational. Logic is straight forward if conditional is true, then do branch, 4-5 hours.

3.3.6 Get the OP Codes to work: 15-18 Basic instruction commands, client has provided code for these basic functions. This should take 6 - 8 hours approximately.

3.3.7 Parse Files: March 15-18 Simple parsing logic for ARM syntax. It should take some time to learn, and then to implement. 16 - 24 hours approximately, due to many unknowns.

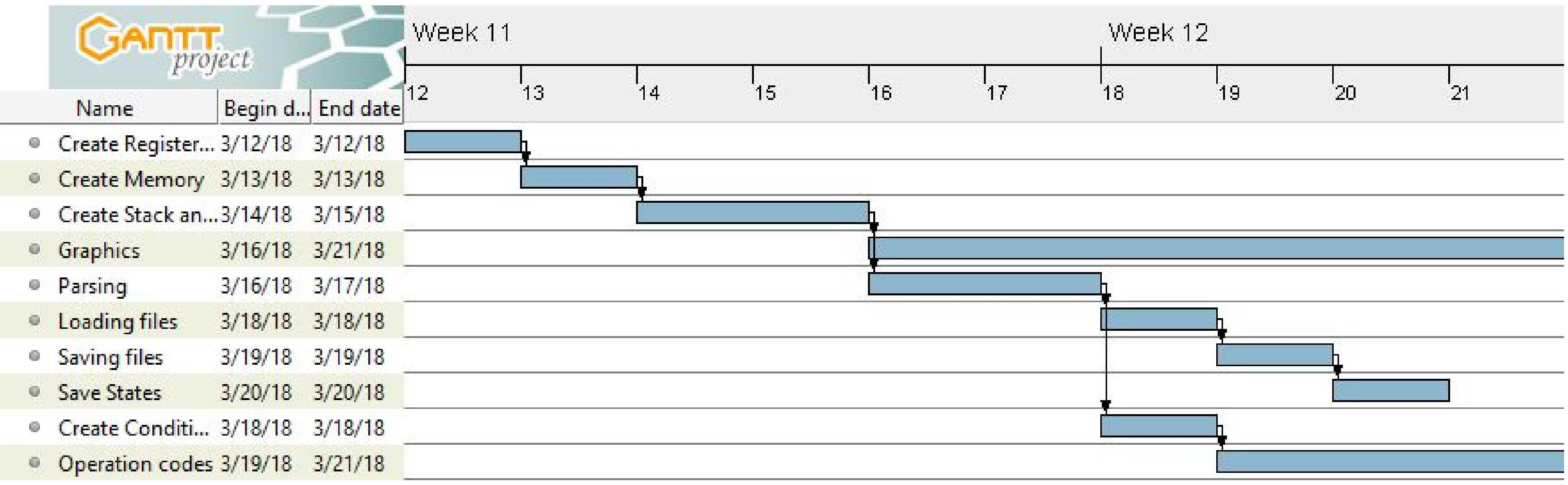
3.3.8 Doubly Linked List storage: March 16-17 Trial and error in trying out different data structures for storing data. 10 - 16 hours, approximately to account for experimentation.

3.3.9 Save/Load/Execution: March 17-19 The trickiest bit will be the save state. Extra hours to compensate for possible bugs, approximately 20 - 30 hours to find and implement an appropriate algorithm.

3.3.10 Create Window: March 15-16: Basic graphics work to create the main window of the program, approximately 4 -6 hours.
3.3.11 Instruction Display/Data Displays: March 16-21: Configure the graphics window to display detailed instruction data and execution states, approximately 12 - 16 hours.

3.4 Gantt chart

The Gantt Chart describes the scheduling of each part of the project. This will allow a more strict schedule to complete different content in, structuring the process of the project and providing a path to the completion of an ARM Assembly Emulator.



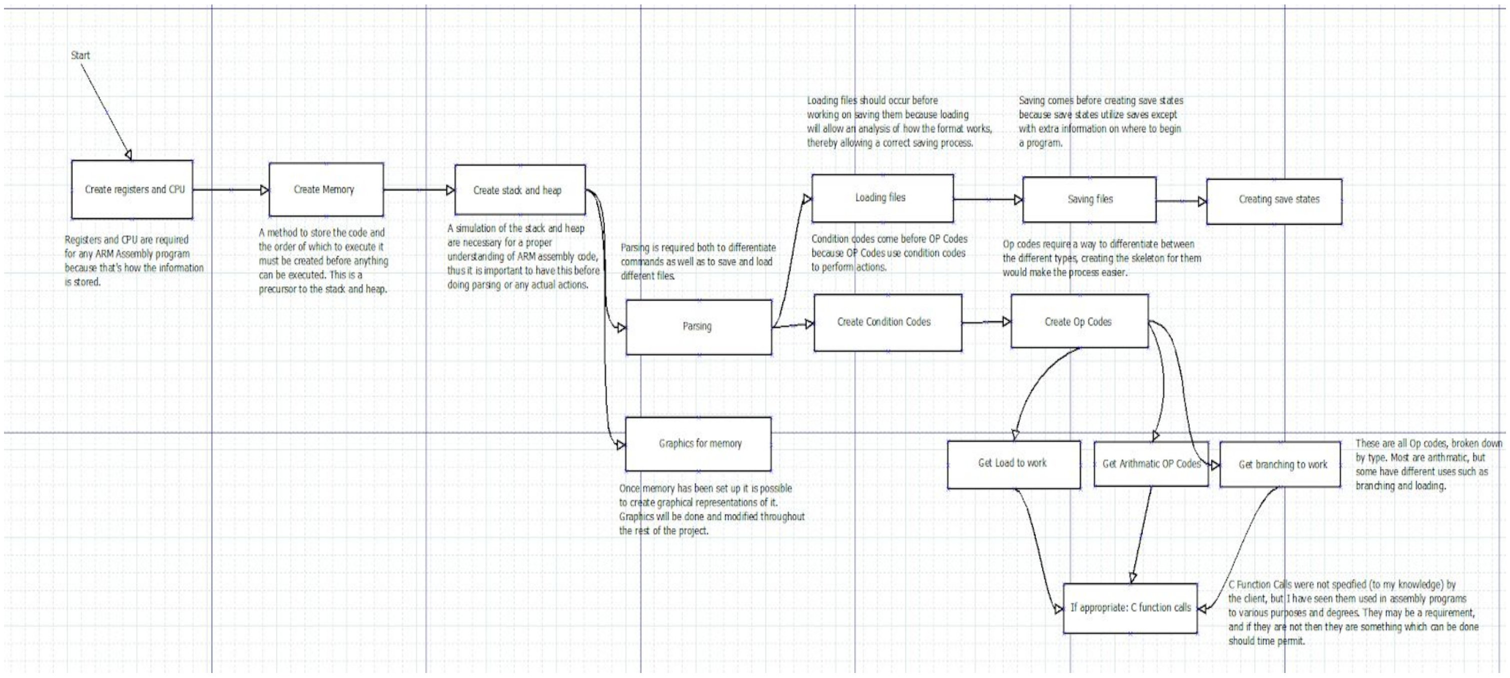
Name	Begin d...	End date
Create Register...	3/12/18	3/12/18
Create Memory	3/13/18	3/13/18
Create Stack an...	3/14/18	3/15/18
Graphics	3/16/18	3/21/18
Parsing	3/16/18	3/17/18
Loading files	3/18/18	3/18/18
Saving files	3/19/18	3/19/18
Save States	3/20/18	3/20/18
Create Conditi...	3/18/18	3/18/18
Operation codes	3/19/18	3/21/18

This chart takes place in the month of March, 2018. The first important thing to complete would be the variables and the CPU to process the necessary information. The registers will hold the values which will be manipulated. Memory will be created next, as a prerequisite for storing and manipulating all of the information presented to the user or by the user. Graphics will take the entirety of the time because the graphics will continually be updated as the program evolves to incorporate more functionality. While doing graphics, parsing will begin, which is necessary to create the different arguments and differentiate them. This will lead into both saving/loading files as well as working with the different operands involved in ARM Assembly Code.

3.5 Task Dependency Diagram

Link to higher quality image of Task Dependency Diagram: https://drive.google.com/file/d/1tUToRkwwM9d7I1E-hnErnn686AF-B2be/view?usp=sharing

The task dependency diagram shows the process of which to accomplish all of the stated milestones. These are ordered by what can be accomplished in a linear fashion.



Registers and CPU are required for any ARM Assembly program because that's how the information is stored.

A method to store the code and the order of which to execute it must be created before anything can be executed. This is a precursor to the stack and heap.

A simulation of the stack and heap are necessary for a proper understanding of ARM assembly code, thus it is important to have this before doing parsing or any actual actions.

Parsing is required both to differentiate commands as well as to save and load different files.

Once memory has been set up it is possible to create graphical representations of it. Graphics will be done and modified throughout the rest of the project.

Loading files should occur before working on saving them because loading will allow an analysis of how the format works, thereby allowing a correct saving process.

Saving comes before creating save states because save states utilize saves except with extra information on where to begin a program.

Condition codes come before OP Codes because OP Codes use condition codes to perform actions.

Op codes require a way to differentiate between the different types, creating the skeleton for them would make the process easier.

These are all Op codes, broken down by type. Most are arithmatic, but some have different uses such as branching and loading.

C Function Calls were not specified (to my knowledge) by the client, but I have seen them used in assembly programs to various purposes and degrees. They may be a requirement, and if they are not then they are something which can be done should time permit.

Section 4: Appendix

4.1 Glossary

4.1.1 Cache: Eight 32-bit lines of direct mapped, one-way set associative cache memory. The cache is filled from main memory and, as such, is not fillable by the user.

4.1.2 ARM Condition Codes:

Bin.	Cond.	Desc.
0000	EQ	equal
0001	NE	not equal
0010	CS or HS	carry set / unsigned higher or same
0011	CC or LO	carry clear / unsigned lower
0100	MI	minus / negative
0101	PL	plus / positive or zero
0110	VS	overflow set
0111	VC	overflow clear
1000	HI	unsigned higher or same
1001	LS	unsigned lower or same
1010	GE	signed greater than or equal
1011	LT	signed less than
1100	GT	signed greater than
1101	LE	signed less than or equal
1110	AL or omitted	always

4.1.3 Main Memory: Main memory is an 8192-byte continuous memory bank independent of the registers and cache. The main memory is typically displayed in 4-byte (32-bit) quantities.

4.1.4 ARM Operation Codes:

Bin.	Cond.	Desc.
0000	AND	regd ← rega & argb
0001	EOR	regd ← rega ^ argb
0010	SUB	regd ← rega - argb
0011	RSB	regd ← argb - rega
0100	ADD	regd ← rega + argb
0101	ADC	regd ← rega + argb + carry
0110	SBC	regd ← rega - argb - !carry
0111	RSC	regd ← argb - rega - !carry
1000	TST	set flags for rega & argb
1001	TEQ	set flags for rega ^ argb
1010	CMP	set flags for rega - argb
1011	CMN	set flags for rega + argb
1100	ORR	regd ← rega | argb
1101	MOV	regd ← arg
1110	BIC	regd ← rega & ~argb

1111 MVN regd ← ~argb

4.1.5 Registers: The ARM architecture User registers consist of sixteen 32-bit registers. For this project they will be simulated using java structures.

4.1.6 Single Step / Multiple Step: The process of moving through a program one (or many) steps at a time. The program would execute one (or many) action(s), then wait to continue.

4.2 Author Contributions

As a Team: Section 1, 2 (Borrowed from Requirements), 3.2

Stephen Wilson: Initially carried over and reworked some of sections 1 and 2 from requirements doc for peer review. Wrote Sections 3.2, 3.1. Various editing/proofreading. .

Sean Perts: Title page, Table of Contents, Contributed language to Section 1.1, 1.2, 1.3, and 1.4. Edited Section 2.3 and 2.4. Completed Sections 4.1 and 4.3. Added to section 3.3. Styled whole document.

Jonathan Erickson: Largely helped with planning as well as editing portions of section 3. Wrote a majority of 3.1, worked some with 3.2, and and did the graphics for the Gantt Chart and Task Dependency Diagram. Also did the chart explanations. Did some proofreading.

4.3 Additional Documents

4.3.1: Gusty Cooper's ARM Assembly Presentation
https://docs.google.com/presentation/d/1V8ibvrittWz99EvUg1V6v8hPAJOgJG0diBPv0bd2KWA/

4.3.2: Gusty Cooper's Functional Requirement Notes
https://docs.google.com/document/d/16Kpqb6m23e5sYCJ6m6vzHFUmD6bgfacMdroHkMFS9Dg/edit?usp=sharing