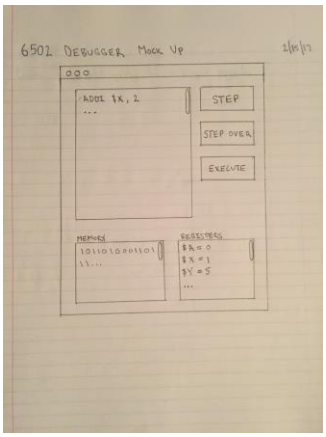
Contents



NOTES:

We could organize this document into two things, “requirements for processor” and “requirements for debugger”. They are two different things really, and serve different functions. The debugger is going to be free of design in this document just like the address book, but the processor is already “designed.” Certain functions of the processor are already inherently designed, so they could be in this document.

Formatted: Font: 12 pt, Not Bold

1 Introduction

Team Robo Sharks aims to provide you (the retro software developer) with a tool that will allow you to analyze your 6502 assembly code. We understand that the audience for retro programming may not be a profitable target group. However, we at Team Robo Sharks are passionate about technology that brought us into this day and age. Our mission is to provide a debugging tool so you can efficiently explore assembly code written for any device or emulator with a 6502 instruction set. This open source debugger is being developed with a community centric focus as support for commercial utilities has been deprecated. Devices that contained this 8-bit microprocessor include such items such as Atari, Apple II, Nintendo Entertainment System, Commodore PET as well as other gadgets from Motorola and Intel. These companies have long moved on from this technology however we feel that the educational value and fun factor of learning and coding for these platforms is applicable to begin learning about the architecture and data flow of modern processors.~~worth the effort.~~

1.1 Proposed System

The Robo Shark debugger will contain the necessary tools to analyze 6502 binary machine code. This analysis occurs as you, the user, input your assembly code instructions to an input window and use the provided debugging features.~~using the interactive features provided.; step through the lines of code~~. Each line will be ~~translated~~ assembled by the debugger to 6502 instructions and will ~~return~~ the values in memory and registers will be updated in real time.~~as well as the values in the available registers~~. Included is a mock up of the graphical user interface the user might expect to see:

1.2 Scope

As mentioned in the introduction we feel that our product will be valuable to both the hobby programmer as well as an educational tool. We anticipate computer architecture and computer engineering schools will be interested in incorporating our product with their curriculum to enhance learning objectives such as teaching assembly code.

We also need a “how to use this document section” similar to the one in the “A simple Address Book” in class.

1.3 Using these Pages

(example organization, functional and quality requirements can go to sections 3 and 4 like normal though)

1. Begin with a functional requirements statement (split into requirements of the processor & requirements of the debugger)

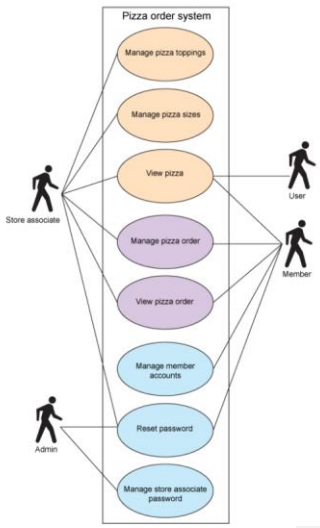
2.Begin with a quality requirements statement

2. Then view the use cases (could we add something like the attached document? I think this would help our score on this, especially with our few use cases possible.

3.(Let’s talk about where the gui mockups come in, as like in class, it’s kinda weird to put in here. Some of the example ones do though, let’s just find out where to put it.)

4.

Figure 1. Pizza order system



2 Concept of Operations

2.1 Operational Constraints

Constraint	Impact
Limited production time.	Less features than desired.
Limited team size.	Open source support format.
Limited experience.	Possible technical difficulties (delays).

2.2 Process Descriptions

Details	Process Description
Process Name	~~Loader~~Importer & Assembler
Purpose	~~User assembly input.~~Manage user input into something the processor understands.

Description	Allow for user to type assembly (requirement of debugger) and to assemble this into binary (requirement for processor).~~Purpose of the debugger is to analyze user input. Loader takes in that user input.~~

Details	Process Description
Process Name	Memory
Purpose	Retain information loaded.
Description	Required for maintaining state.

Commented [HW1]: We don't have state right now, but we could change this into required for loading instructions into RAM and allowing the user to reference other memory in their instructions.

Details	Process Description
Process Name	Registers
Purpose	Store values to be computed.
Description	Used for moving values and executing computation.

Details	Process Description
Process Name	Arithmetic Logic Unit
Purpose	Compute instructions provided.
Description	~~Math.~~Decoded instructions will only use the computational logic found here to execute instructions.

Details	Process Description
Process Name	User Interface
Purpose	Allow user to interact with debugger.
Description	Contains the functions to Step, Execute, Step-over, Display of register values, Display of current memory, and to write assembly into the processor.~~.~~

2.3 Use Cases

1. *Loader*
    1. **Brief Description**

1. This use case describes how you load 6502 assembly code.
2. **Actors**
    1. User
    2. Debugger
3. **Preconditions**
    1. The debugger context must be active.
4. **Basic Flow**
    1. The debugger presents the user with the option to input their 6502 assembly code.
    2. The user inputs their 6502 assembly code into the debugger.
5. **Alternative Flow**
    1. The debugger presents the user with the option to import 6502 assembly code from file.
        1. The user selects the option to import a file.
        2. The debugger presents a view of files from the user's directory.
        3. The user imports their 6502 assembly file.
        4. GOTO Post Condition
        ~~1. imports their 6502 into the debugger from file.~~

    ~~2. The user exits the debugger.~~
6. **Post Condition**
    1. Assembly code loaded is assembled into the processor, and a~~nto~~ active debugging session can begin.~~.~~

Do we want to use these definitions?

• (*Step Into or Step*) A method is about to be invoked, and you want to debug into the code of that method, so the next step is to go into that method and continue debugging step-by-step.

• (*Step Over*) A method is about to be invoked, but you're not interested in debugging this particular invocation, so you want the debugger to execute that method completely as one entire step.

Formatted: Indent: Left: 0.5", No bullets or

Formatted: Justified

Formatted: Justified, Indent: Left: 0.5", No bullets or numbering

Formatted: Font: Not Italic

2. *Step Function*
    1. **Brief Description**
        1. This use case describes how you step through the loaded 6502 assembly code.
    2. **Actors**
        1. User
        2. Debugger
    3. **Preconditions**
        1. The debugger context must be active and 6502 assembly code must be loaded into the debugger.
    4. **Basic Flow**
        1. The debugger presents the user with an option to step into the loaded 6502 assembly code.

2. The user activates the step option.
5. **Alternative Flow**
    1. The user exits the debugger. (I think this has to be explained more, as well as the ones below with this alt. flow, as this option is never presented, and the post condition is still unkown.)
    ~~1.~~2. Undo for an alternative flow & functionality?
6. **Post Condition**
    1. The debugger will step into the next line of code, displaying (if any) memory loaded and register values (if any) as a result of code execution. This action increments the program counter (marker for last executed line of 6502 assembly code) allowing the program (loaded 6502 assembly code) to continue onto the next step iteration.

3. *Step-Over Function*
    1. **Brief Description**
        1. This use case describes how you step over the loaded 6502 assembly code for debugging.
    2. **Actors**
        1. User
        2. Debugger
    3. **Preconditions**
        1. The debugger context must be active and 6502 assembly code must be loaded into the debugger.
    4. **Basic Flow**
        1. The debugger presents the user with an option to step over a line of 6502 assembly code.
        2. The user activates the step over option.
    5. **Alternative Flow**
        1. The user exits the debugger.
    6. **Post Condition**
        1. The debugger will step over the next line of code; not performing any action. The program counter remains at the last non executed line of code until a step action or execute call is made.

4. *Execute Function*
    1. **Brief Description**
        1. This use case describes how you execute all of your loaded 6502 assembly code for debugging.
    2. **Actors**
        1. User
        2. Debugger
    3. **Preconditions**
        1. The debugger context must be active and 6502 assembly code must be loaded into the debugger.
    4. **Basic Flow**
        1. The debugger presents the user with the option to execute the loaded 6502 assembly code.

2. The user activates the execute option.

5. **Alternative Flow**
    1. The user exits the debugger.
6. **Post Condition**
    1. The debugger will execute the 6502 assembly code that has been loaded.

3 Behavioral Requirements

4 Quality Requirements