Asm (Assembler) Program

Project RISCII (PR0003)

“Breakdown of RISCII assembler into verification and validation steps”

Rev A
28-JUN-2024

Purpose

This document is meant to describe the larger task of creating the RISCII assembler program. It primarily defines the requirements used to verify the created program, but also records the user story and use cases for validation purposes, as well.

The contents of this document should effectively drive project efforts in creating the assembler portion of the compiler. It is expected that this document is referenced during the planning, development, and testing of the assembler program.

Abbreviations

Abbreviation	**Description**
ISA	Instruction Set Architecture
word	2 byte value, unless context dictates it is referring to a sequence of letters (i.e. natural language)
ASCII	American Standard Code for Information Interchange- common standard used to encode text symbols

References

Name	Document Number
Instruction Set Architecture	PR0001
Mcu Design	PR0002
Asm Design	PR0004

User Story

This section captures the main “ask” of the assembler program. In high level, general terms, the assembler and how it works is described. See the subsection below.

User Story

For Project RISCII, a program is needed to convert assembly level instructions and data across one or more files into, if successful, one binary image that can be programmed and run on the RISCII microprocessor. This program must be an independent executable that can be evoked via the command line, though will be called as part of the larger RISCII compiler. While primarily an assembler, it effectively also acts as the linker, too.

Assembly files primarily consist of instructions (as defined in document PR0001 Instruction Set Architecture). Their arguments are heavily based on the ISA, though are typed using human readable characters. In general. Instructions are placed in the text section of the binary image (as defined in document PR0002 Mcu Design) in the same order they appear in their files (the order instructions some files appear, beyond being deterministic and mechanics to set a starting instruction, is arbitrary).

Along with instructions, global data may be declared. Data can be of any length and initialized with literal or abstracted- but literal at compile time -values, though is rounded to fit within a multiple of a word. This should allow anything from a single global integer to an initialized multi-byte data type from being declared as global data. Global data is placed in the data section of the binary image (the order of individual declarations, beyond being deterministic, is arbitrary).

Furthermore, a special type of global data can be declared with no initial value. The data can be of any length, though is rounded to fit within a multiple of a word. This data

is not directly placed in the binary image, but influences the assembly process to save an extra region of free memory, contiguous to the initialized global data region, for these values. This extra region is referred to as the “bss” section (the order of individual declarations, beyond being deterministic, is arbitrary).

In general, all values and arguments found at the assembly level should be human readable (i.e. values don’t have to be given in binary or hexadecimal). The assembler should be able to interpret decimal and hexadecimal integers, as well as ASCII strings as appropriate to the context. Additionally, for human readability, comments should be supported.

The assembler should abstract direct addresses from the user with alpha-numeric labels. These labels can be assigned to specific portions of the text and data sections for use later on (i.e. access address, set register to address value). On top of this, the assembler should have unique labels which allow the user to select an instruction as the start of the text section, as well as determine the starting and ending addresses of the binary image (in the storage address space), data section, and bss section. Labels should be resolved at compile time to allow the literal address values to be used in the binary image.

With regards to command line usage, the user should be able to provide one or more files for assembling, as well as be able to choose the name of the created binary image, control how much, if any, standard text output is given, ask for warnings to be treated as errors, and provide a blurb helping the user understand how to call the assembler executable. All standard output messages should be concise, but informative. When an error is reached, the assembler should terminate unless it is analyzing a file, in which case, it should finish analyzing the file before terminating in an effort to provide more warnings/errors.

For future proofing the design, the assembler should be designed for additional command line and assembly language features. These features will likely involve additional arguments that alter how the binary image is created (i.e. optimizations), add new language features (i.e. one line functions that act as multiple instructions), or involve non-assembly input/outputs (i.e. read in a linker script, output a map file).

Use Cases

This section takes the statements made in the user story and attempts to break it down into individual, independent pieces. These “use cases” focus on how the assembler program will be used and, thus, how it may be verified and validated.

Executable Use Cases

Use Case #0: Assembler Program
As a user, I want a program that can convert assembly files into one, binary image (if the assembly process was successful).

Use Case #1: RISCII Binary Images
As a user, I want binary images created by the assembler to be able to be programmed and run on the RISCII microprocessor (as defined in document PR0002 Mcu Design).

Use Case #2: Independent Executable
As a user, I want the assembler program to be a stand-alone executable (running on the host operating system).

Use Case #6: Assembler Reliability
As a user, I want the assembler to create the binary image in a deterministic manner (i.e. instructions and data should not be placed differently for the same input files).

Use Case #26: Command Line Name Option
As a user, I want to specify the name I want the binary image to be given when it is created by the assembler.

Use Case #27: Command Line Standard Output
As a user, I can select how much standard output, if any, I want displayed while assembling the binary image.

Use Case #28: Standard Output Quality
As a user, I want all standard output to be concise, but informative.

Use Case #29: Standard Output Help Blurb
As a user, I want the assembler to output a helpful message to help users understand how to call the assembler executable correctly.

Use Case #32: Warnings as Errors
As a user, I want to be able to have the assembler treat warnings as errors (i.e. a warning will prevent the binary image from being created).

Use Case #33: Error Termination
As a user, I want errors to result in termination of the program, but not before finishing analyzing the current file (if applicable).

Language Use Cases

Use Case #3: Instruction Set
As a user, I want to infer instructions and their arguments (as defined in document PR0001 Instruction Set Architecture) in an assembly file.

Use Case #4: Instruction Placement
As a user, I want my inferred instructions to be placed in the text section of the binary image in the same sequence I wrote them in the assembly file.

Use Case #5: Text Section Start
As a user, I want to be able to tell the assembler which inferred instruction is the start of my assembly level program.

Use Case #7: Initialized Global Data
As a user, I want to declare global data with an initialized value.

Use Case #8: Initialized Global Data Placement
As a user, I want my declared, initialized global data to be placed in the data section of the binary image

Use Case #9: Initialized Global Data Size
As a user, I want to declare global, initialized data of any size that is a multiple of a word.

Use Case #10: Initialized Global Data Types
As a user, I want to initialize global data with any literal value or label value.

Use Case #11: Uninitialized Global Data
As a user, I want to declare global data that is explicitly uninitialized.

Use Case #12: Uninitialized Global Data Reservation
As a user, I want declared global, uninitialized data to NOT be placed in the binary image, but instead inferred in binary code as reserved memory.

Use Case #13: Bss Region Location
As a user, I want the bss section, if inferred by the assembler, to be placed next to the data section in free memory.

Use Case #14: Uninitialized Global Data Size
As a user, I want to declare global, uninitialized data of any size that is a multiple of a word.

Use Case #15: Human Readability
As a user, I want all arguments in an assembly file to be displayed in human readable characters.

Use Case #16: Integer Interpretation
As a user, I want the assembler to interpret decimal and hexadecimal integers in the assembly file.

Use Case #17: ASCII String Interpretation
As a user, I want the assembler to interpret ASCII string literals in the assembly file.

Use Case #18: Supported Comments
As a user, I want the assembler to support non-code comments in the assembly file.

Use Case #19: Supported Labels
As a user, I want the assembler to support labels (alpha-numeric tokens, similar to variables in many programming languages) in the assembly file.

Use Case #20: Label Definition
As a user, I want to be able to define a label.

Use Case #21: Label Binding
As a user, I want defined labels to be bound to a given instruction or global data.

Use Case #22: Label Usage
As a user, I want to be able to use the label as an address that can be accessed or saved as an integer value.

Use Case #23: Special Text Section Label
As a user, I want the assembler to declare a special label that the user can use to define the first instruction in the text section.

Use Case #24: Special Boundary Labels
As a user, I want the assembler to define labels that refer to the end of the binary image, data section, and bss section (the start of each item is either constant or the end of another section).

Use Case #25: Label Resolution
As a user, I want labels to be resolved into addresses during compile time.

Additional Use Cases

Use Case #30: Scalable Command Line Options
As a user, I want the assembler to be written such that the command line options available can be easily extended.

Use Case #31: Scalable Assembly Language
As a user, I want the assembler to be written such that the assembler language can be easily extended to have additional features.

Detailed Behavior

This section dives deeper into the use caes- breaking down how each use case is realized in the final product. This effectively ensures the final product matches the original desire of the user story through its use cases (i.e. product validation).

On a realistic project, this would be a back and forth discussion (and eventual agreement) between the customer and the developers. For this personal project (and for brevity), this is effectively covered in document PR0004 Asm Design, which specifies how the assembler should behave in detail. Refer to that document for this section.

Revision History

Rev	Date	Description of Changes	Initials
A	28-JUN-2024	Initial draft- focus on story + use cases	J.E.