Asm (Assembler) Design

Project RISCII (PR0004)

“Description of internal and external design of RISCII assembler”

Rev A
28-JUN-2024

Purpose

This document is meant to describe how the RISCII assembler program is designed. It primarily focuses on “external” interfaces and usage of the program, though also goes into some detail about the detailed design of the program.

The contents of this document should effectively describe how to write and assemble assembly level programs using the assembler. It is expected that this document is referenced for developing assembly level libraries and compiler back-end systems.

Abbreviations

Abbreviation	**Description**
MCU	Microcontroller unit, though can also be used to refer to a microprocessor
decimal	In the context of low-level software, decimal is often used to describe integer counting using Arabic symbols (e.g. 2, -5, 0). It does NOT infer use of a decimal point
ASCII	American Standard Code for Information Interchange- common standard used to encode text symbols
ISA	Instruction Set Architecture

References

Name	Document Number
Instruction Set Architecture	PR0001
Mcu Design	PR0002
Asm Program	PR0003

Command Line Usage

This section describes how the assembler is used from the command line. By invoking the assembler and its arguments, the creation of binary images can be controlled and customized.

Basic Usage

The minimum viable use of the assembler is assembling a single assembly file into a binary image. The call begins with the name of the assembler executable (e.g. "asmld.exe" followed by the name (and path) of a file- preferably one containing assembly language, defined further in this document.

Example:

```
$ asmld.exe foobar.asm
$ ls
asmld.exe   foobar.asm   foobar.bin
```

In the example above, the assembler is called to read the contents of a file called "foobar.asm" and create a binary image based on its contents. This binary image, called "foobar.bin" in this case, is created in the same directory the assembler was called in and names it based on the first (or only) file read from. If the file already exists, it will overwrite the old file with the new one.

To assemble a multi-file assembly program, multiple files can be named after the assembler executable. The assembler will read each file (in the order they are given) and create one binary image based on their contents. See below for examples.

Examples:

```
$ asmld.exe foo.asm bar.asm
$ asmld.exe red.asm blue.asm
$ asmld.exe blue.asm bar.asm
$ls
asmld.exe   bar.asm   blue.asm   blue.bin   foo.asm   foo.bin   red.asm   red.bin
```

With regards to return codes, the assembler returns 0 if the binary image was created successfully. If not, a return code matching the offending warning or error is returned (see further in this document for details).

Command Line Options

While the assembler's command line usage is primarily for passing in the assembly files needed to create the binary image, additional options can be given to customize how the assembler processes the files and reports its status. The following subsections cover these options in detail.

Help Option (-h)

The help option causes the assembler to print out a small help menu (seen example below). This menu provides an overview of the assembler's current version, how it's used, and what options are available (i.e. a truncated version of this section).

Example:

```
$ asmld.exe -h
RISCII Assembler (and Linker)- version X.X.X
Usage: asmld.exe [options] files…
Options:
   -h          print this help blurb and exit
   -o <arg> name binary image <arg>
   -ll <arg> set verbosity of std output. Options are:
               quiet  : no std output
               err    : errors only
               warn   : errors and warnings
               info   : errors, warnings, and info
               debug : all possible output
   -we       treat warnings as errors
```

The help option always causes the assembler to terminate without creating the binary image, and even works when it is the only input given to the assembler. Furthermore, it is implicitly called when no arguments are no provided to the assembler on the command line.

Output File Option (-o)

The output file option allows the user to choose the filename the assembler uses to create the binary image. It is used by giving the desired filename just after the "-o" flag. See the example below.

Example:

```
$ asmld.exe -o my_bin foo.asm
$ ls
asmld.exe   foo.asm   my_bin
```

The filename given must be a single token (i.e. cannot contain whitespaces) and must come just after the "-o" flag. The "-o" flag causes the next token to be interpreted as the output filename (instead of an assembly file to assemble). If the output filename requires a type, it is the caller's responsibility to add while using the output file option.

Log Level Option (-ll)

The log level option allows the user to choose the granularity of the printed stdout output generated by the assembler. It is used by adding one of several predetermined input tokens just after the "-ll" flag. See example below.

The following tokens (and their effects) are recognized as log levels:

- quiet	do not generate stdout output
- err	generate error stdout output only
- warn	generate error and warning stdout output
- info	generate error, warning, and informational stdout output
- debug	generate all possible stdout output

Example:

```
$ asmld.exe foo.asm
asmld.exe [ERR] "foo.asm" not found
$ asmld.exe -ll info foo.asm
asmld.exe [INFO] opening "foo.asm"
asmld.exe [ERR] "foo.asm" not found
```

If the option is not given, log level “err” is chosen by default. If given, the argument just after “-ll” must be one of the tokens specified above, otherwise the assembler will error out. The “-ll” flag causes the next token to be interpreted as the log level specifier (instead of an assembly file to assemble).

Warnings As Errors Option (-we)

The warnings as errors options allows the user to have the assembler allow warnings to act similar to errors- allowing them to prevent the creation of the binary image. This allows the user to effectively subject the assembly program to additional scrutiny (with respect to binary image creation- the amount of error and warning checks do not change). See the example below.

Example:

```
$ asmld.exe -we foo.asm
asmld.exe [WARN] unused label “foobar”
asmld.exe treating warning as an error- exiting…
$ ls
asmld.exe   foo.asm
$ asmld.exe foo.asm
$ ls
asmld.exe   foo.asm   foo.bin
```

Adding the “-we” flag primarily allows warnings to prevent binary image creation. However, the flag also allows warnings to be printed with the same priority as errors (i.e. even if “-ll err” is specified, warnings will be printed if “-we” is also specified).

Assembly Language

This section describes the assembly language supported by the assembler. Files written in this language can be interpreted by the assembler and converted into one, unified binary image.

Variable Arguments

Before diving into the language’s mechanics, it is important to understand how the arguments of the language work. “Variable argument” refers to any non-keyword token in the language. A summary of each one’s syntax and use is given below.

Flag Value

Flag values are used to refer to “flag bits” within instructions in a more human readable manner. Flag values begin with a ‘%’ followed by one or more of lowercase letters. See below for examples.

Flags: %a   %s   %nzpc   %np

Valid flag characters are dependent on the instruction. Further sections will describe the valid flags. Document PR0001 Instruction Set Architecture describes the underlying instructions and their flags in more detail.

Register Value

Register values are used to refer to registers within the MCU’s register file. Register values begin with a ‘$’ followed by one or more digits. See below for examples.

Registers: $0 $1 $6 $7

According to document PR0001 Instruction Set Architecture, the instruction set is designed around a register file with 8 registers. Because of this, only 8 register values (i.e. $0 through $7) are considered valid.

Immediate Value

Immediate values refer to literal integer values. They can be given as decimal or hexadecimal values. Decimal values may begin with a leading ‘+’ or ‘-’ while hexadecimal values always begin with ‘0x’. See below for examples.

Decimals:	2	-5	0	+6
Hexadecimals:	0x4	0xA	0x2aB1	0x0

How the integer is interpreted, in terms of size and signedness, is driven by the context. Further sections will describe this context as needed.

String Literal Value

String literal values refer to an array of ASCII characters contained within quotation marks (i.e. “”). The assembler supports all ASCII characters, as well as the following escaped characters:

- \n newline character- 0x0A
- \r carriage return- 0x0D
- \t tab- 0x09
- \” quotation mark- 0x22
- \0 null- 0x00

See below for examples of string literals.

String Literals: “Hello” “World 123!” “a \”quote\”\r\n\0” “”

Each character in the string takes one byte, with every two characters packed into one word. In accordance with document PR0003 Asm Program, string literals are rounded to a multiple of a word, adding a packed byte equivalent to 0x00 as needed (or no bytes if the string literal is empty- e.g. “”).

Note that string literals are NOT implicitly terminated with a NULL (i.e. 0x00) character.

Label Value

Label values are alphanumeric tokens used to refer indirectly to addresses in the binary image. They begin with a letter (either case) or a ‘_’ followed by any number of letters (either case), digits, or underscores. See below for examples of labels.

Labels: foo Bar2 BIG_NAME _

In accordance with document PR0001 Instruction Set Architecture, labels, as addresses (byte addressable), are always sized to one word. Their exact values are determined during compile time and are used in place of the given names.

Array Value

Array values are composite values made up of integer, string literal, and label values. The array is contained in curly brackets (i.e. ‘{‘ and ‘}’) and delimited using whitespaces (i.e. space, tab and newline). See below for examples.

Arrays: {0 1 2 3} {“Hello ” _myLabel} {0x0 -5 “\0” bar} {}

For sizing, each element is first fit to size independently (integers sized to one word each), then sized as one value. In accordance with document PR0003 Asm Program, arrays are rounded to a multiple of a word, adding a packed byte equivalent to 0x00 as needed (or no bytes if the array is empty- e.g. {}).

Instructions

The primary unit of an assembly level program is the instruction. Instructions represent the binary instructions stored in the text section of the created binary image (as defined in document PR0002 Mcu Design).

Instructions are expressed using select keywords followed by select sequences of variable arguments. The syntax of these instructions and their effect are described in the following subsections.

Direct Instructions

Direct instructions correspond directly to instructions within the ISA (see document PR0001 Instruction Set Architecture). Each instruction has an associated keyword and valid sequences of variable arguments.

The table below outlines all valid direct instruction sequences. Valid tokens are specified in the table (and marked with '*' if optional) in the order they should appear (i.e. left to right). Cells for variable arguments describe how the argument is used in the instruction.

Keyword	**Flags**	**Register 1**	**Register 2**	**Register 3**	**Immediate**	**Description**
AND		Destination	Operand 1	Operand 2		Bitwise AND
AND		Destination	Operand 1		Operand 2	Bitwise AND
ORR		Destination	Operand 1	Operand 2		Bitwise OR
ORR		Destination	Operand 1		Operand 2	Bitwise OR
XOR		Destination	Operand 1	Operand 2		Bitwise XOR
XOR		Destination	Operand 1		Operand 2	Bitwise XOR
SHL		Destination	Operand 1	Operand 2		Shift Logical Left
SHL		Destination	Operand 1		Operand 2	Shift Logical Left
SHR	a*	Destination	Operand 1	Operand 2		Shift Right
SHR	a*	Destination	Operand 1		Operand 2	Shift Right

ADD		Destination	Operand 1	Operand 2		Arithmetic Addition
ADD		Destination	Operand 1		Operand 2	Arithmetic Addition
SUB		Destination	Operand 1	Operand 2		Arithmetic Subtraction
SUB		Destination	Operand 1		Operand 2	Arithmetic Subtraction
LBI	s*	Destination			Source	Load Byte Immediate
LDR		Destination	Pointer		Offset	Load Memory Word
STR		Source	Pointer		Offset	Store Memory Word
BRC	n* z* p* c* (must be at least one flag)				Offset	Branch Conditional
JPR			Pointer		Offset	Jump Base Register
JPR	r					Return From Interrupt
JLR		Destination	Pointer		Offset	Jump and Link Register
SWP		Destination	Pointer		Offset	Swap Register and Memory
NOP						No Operation
HLT						Halt

- Table 1: Direct Instruction Sequences

For immediate values, the size of signedness of each instance is dependent on the underlying instruction’s syntax. As long as the value can be expressed for the instruction's number of immediate bits and sign, the assembly instruction will be translated properly. See further in the document for warnings and errors regarding immediate truncation.

For flag values, the flags stated above are based on the underlying instruction's flag bits. The meaning of each flag is as follows:

1) a arithmetic flag- tells SHR to perform arithmetic, not logical, shifting
2) s shift flag- tells LBI to left shift the byte into, not overwrite the destination
3) n negative flag- tells BRC to branch for a negative result
4) z zero flag- tells BRC to branch for a zero result
5) p positive flag- tells BRC to branch for a positive result
6) c carryout flag- tells BRC to require result to have carry out to branch
7) r return flag- tells JPR to return from interrupt instead of performing a jump

In the case of BRC, its four flags may be included in any configuration, so long as at least one flag is present. See the ISA for more details regarding each flag's effect.

Functions

Functions represent multiple direct instructions in one line. While not desirable from a "1-1" assembly-to-binary view, functions help simplify common operations and ease translation from assembly constructs to binary instructions.

At present, only one function is supported by the assembler- the load address function. This function is used to load a label's address value (determined at compile time) into a specified register. Similar to a direct instruction, it begins with its keyword "_la" followed by variable arguments- a register and a single label. See below for examples.

Examples:

```
_la     $0 myAddr
_la     $7 dataSectionPtr
_la     $4 functionPtr
```

This function effectively wraps two LBI instructions into one line, allowing the assembly language to refer to addresses abstractedly without worrying about the exact values needed for the LBI instructions.

Global Data

The secondary unit of an assembly level program is global data. Global data represents the binary explicitly stored and initialized in the data section of the binary image (as defined in document PR0002 Mcu Design) or the implicitly stored in the “bss” section of free memory- a feature of the assembler compilation process.

(make sure to note endian/direction behavior of variables)

Initialized Data

Initialized global data represents all data with a preset value (and, indirectly, all data in the data section of the binary image). Any data initialization begins with the keyword “.data” followed by an immediate, string literal, label, or array argument setting the initial value. See below for examples.

Examples:

```
.data 5
.data “Hello World!\0”
.data _myLabel
.data {0x1234 0x5678 “foobar” 0}
```

In terms of sizing, all arguments are sized to fit within a multiple of a word (signed or unsigned). Immediate values and labels are always sized to one word, while string literals and arrays are sized to a multiple of a word (including 0 if the given string or array is empty). A packed byte equivalent to 0x00 is added as needed.

In terms of placement, the assembler favors “left-to-right” thinking. Characters in a string literal are placed at successively larger byte addresses (with the lower byte of a word interpreted as the lower byte address). Elements in an array are placed at successively large word addresses. Immediate values, being sized to one word, are placed as is.

Uninitialized Data

A feature of the assembler (compared to the raw binary image) is declaring uninitialized global data by implicitly reserving a portion of the free memory. Uninitialized data declaration begins with the keyword “.bss” followed by an immediate describing the number of bytes to reserve. See below for examples.

Examples:

```
.bss 20
.bss 0x1
.bss 0
```

The immediate value provided is scoped to be within an unsigned word (i.e. negative values are not allowed). In accordance with document PR0003 Asm Program, reserved regions are rounded to a multiple of a word, adding a packed, uninitialized byte as needed (or no bytes if 0 bytes are reserved- e.g. .bss 0).

Labels

One of the main abstractions the assembly language provides is labels. Labels act as addresses, allowing the user to track and reference various locations in the program without having to manually set, infer, and/or record the addresses.

Labels can refer to most addresses in both the program and data address spaces (as defined by PR0001 Instruction Set Architecture), but each label (i.e. label name) can only represent one address in one address space per program.

Defining a Label

Labels are defined by providing a new label value followed by a colon (i.e. ‘:’). This not only declares the label name, but defines it by linking the label to the next global data or instruction sequence found in the file. See below for examples.

Examples:

```
myLabel:
ADD $0 $1 $2

dataLabel: .data 4
textLabel:

HLT
```

In the examples above, “myLabel” links to the ADD instruction, “dataLabel” to the global data sequence, and “textLabel” to the HLT instruction. Linking to an instruction or global data sequence determines which program space the address is intended for (i.e. program and data, respectively).

As expected, labels cannot be equivalent to keywords. See Appendix A for a list of keywords. Likewise, defined labels cannot reuse already defined label names.

Using a Label

To use a label, use the label name in a suitable context (e.g. the _la function). At compile time, the label name will be resolved to the address it was linked to when being defined. See below for examples.

Examples:

```
myLabel:
ADD $0 $1 $2
_la   $3 dataLabel
_la   $4 myLabel

dataLabel:
.data 9
```

In the examples above, both “myLabel” and “dataLabel” in the _la functions will be resolved to the addresses of the ADD instruction and global data instance, respectively. Notice that “dataLabel” was used before it was defined. As long as a label name is defined somewhere in the assembly level program, the label name can be used anywhere in the program- in any file and any line.

Note that resolved label addresses are always aligned to word addresses. Instructions and global data are always aligned to word addresses. As such, so are resolved labels.

Special Labels

While labels are intended to be defined and used by the user, the assembler declares and/or defines a few special labels, as well. These labels are meant to give the user better control and information about the created binary image.

The assembler implicitly declares the __START label. This label is used to tell the assembler which instruction is the start of the entire program. It is declared instead of defined as it is expected the user will define it, linking the label name to an instruction. The assembler cannot create a binary image if this label is not defined.

The assembler also implicitly defines labels __SIZE, __BSS, and __FREE. __BSS and __FREE are defined as the first word addresses of the bss section and free memory section (i.e. data address space past data and bss sections), respectively. __SIZE defines the size of the binary image (in bytes) within the storage address space. These defined labels can be used to give the program better awareness of its address space usage.

```
|============|
|   Unused   |
|   Storage  |
|============|  __SIZE
|============|
|   Binary   |
|   Image    |
|============|  0x0000
```

- Image 1: __SIZE Label, Storage Address Space

```
|============|
|    Free    |
|   Memory   |
|============|  __FREE
|============|
|     Bss    |
|   Section  |
|============|  __BSS
|============|
|    Data    |
|   Section  |
|============|  0x0000
```

- Image 2: __FREE and __BSS Labels, Data Address Space

Note that __SIZE, __BSS, and __FREE are ultimately determined by the size of the assembled program. It is possible, though unlikely, that these address values could overflow or access mapped memory for a precisely sized program, thus causing unexpected issues when used.

Other Features

This section covers additional features that are generally not covered in the “grand strokes” of the language.

Comments

Comments can be added to any line of the assembly file by adding the ‘;’ character. All text to the right of this character until the next newline will NOT be parsed as assembly code. See below for examples.

Examples:

```
ADD $0 $1 $2 ; this is a comment
.bss 25

; this is a comment, too
```

Warnings and Errors

This section documents the assembler’s implemented warnings and errors. Each warning and error describes a general, but scoped issue the assembler encountered while assembling. For ease, each one is given an identifying number, output through both the stdout output and assembler’s return code.

Warnings

Warnings are issues that do not prevent creating the binary image, but are likely not intended by the user (and possibly require the assembler to “alter” the interpretation of the source code).

Warning ID	**Description**
100	A flag character was valid, but repeated within a single flag value- only one of each character is needed per flag value.
101	Number in register value does not refer to an existing register (according to ISA). Number is truncated to map to a register.
102	Number in immediate value is too large to fit within the instruction/data word. Number is truncated to fit within the context.
103	String literal has no length (i.e. has no characters). It will not result in any bytes added to the binary image.
104	Array has no length (i.e. has no elements). It will not result in any bytes added to the binary image.
105	Label is defined by the user, but never used. This has no adverse effects, but is likely unintended behavior.

- Table 2: Assembler Warnings

Errors

Errors are issues that prevent the assembler from creating the binary image. These issues require the user to make changes to the source code to resolve.

Error ID	**Description**
200	No files were given to the assembler- at least one is required to create a binary image. It is possible an option/flag that takes an argument was specified and the file was interpreted as the flag's argument.
201	Option/flag requiring an argument is specified, but the argument itself is missing. The argument must be provided to continue.
202	Option/flag requiring an argument from a predetermined set is specified, but the argument given was not from that set. The argument must come from the option's specified set of arguments.
203	File given could not be found, opened, and/or read. This often is the result of a bad path name to the file, or perhaps the file is "locked" from reading by another program or OS permissions.
204	Sequence of characters in the file does not match any token in the assembly language. This can be caused by forgetting whitespaces between tokens or mistyping a token.
205	Assembler expected the token to match one item from a list of tokens. This is likely due to bad sequencing of the tokens.
206	Assembler expected a specified token to follow another. This is likely due to bad sequencing of the tokens.
207	Assembler finished parsing the file's tokens, but expected more. This can be caused by the last instruction/data/etc being cut off before finishing.
208	Label name is declared in multiple places. Each label declaration must use a unique name. It is also possible the label name used matches one the assembler's built-in label names (see Appendix A for details).
209	Label name is referenced in the assembly program, but never defined. This prevents the assembler from knowing what value to populate the reference with.
210	Label name was declared, but did not link to either an instruction or data

	instance. This is caused by defining a label at the end of the file (i.e. labels link to the next instruction of data instance in the file- thus it cannot be the last item in the file).
211	Flag value has a character that does not correspond to any known flag in the assembly language. The offending flag should be deleted.
212	Flag value has a character that, while recognized by the assembler, is not a valid flag for the instruction it is modifying. The offending flag should be removed.
213	String literal has an escape character not recognized by the assembler. The sequence may be valid by general ASCII standards, but it is not recognized for this assembler/program.
214	The assembly program can (for the most part) be assembled, but the final binary image will be too large for the target hardware. Thus, the assembler does not bother creating the binary image.
215	The assembly program can (for the most part) be assembled into a binary image, but the data and bss sections will overrun the target hardware's free memory space. Thus, the assembler does not bother creating the binary image.
216	The assembly program can be successfully assembled, but the assembler could not create/open/write the binary image file. The file may already exist and be "locked" by another program or OS permissions.
217	The label the assembler uses to find the starting instruction was not defined in the assembly program. The assembly program must define the specific label in order to continue.
218	The label the assembler uses to find the starting instruction was linked to a data element, not an instruction. The label must be defined and linked to an instruction in order to continue.

- Table 3: Assembler Errors

Appendix A: Assembly Language Reserved Keywords

AND	ORR	XOR	SHL	SHR
ADD	SUB	LBI	LDR	STR
BRC	JPR	JLR	SWP	NOP
HLT	_la	.data	.bss	

- Table A1: Assembly Language Keywords

__START	__SIZE	__BSS	__FREE	

- Table A2: Assembly Language Reserved Label Names

Revision History

Rev	**Date**	**Description of Changes**	**Initials**
A	28-JUN-2024	Initial draft- focus on language and option flags	J.E.