Assembler Documentation

Ruben Saunders

August 2024

Contents

1 Overview

The assembler takes **one** assembly source file and produces a binary output file, which may be executed by the processor.
Disclaimer: The assembly language designed for this project is relatively simplistic. Its goal is to allow for full operation of the processor with minimal-to-no additional features. This is so focus can be given to the development of a high-level language.

1.1 Command-Line Interface

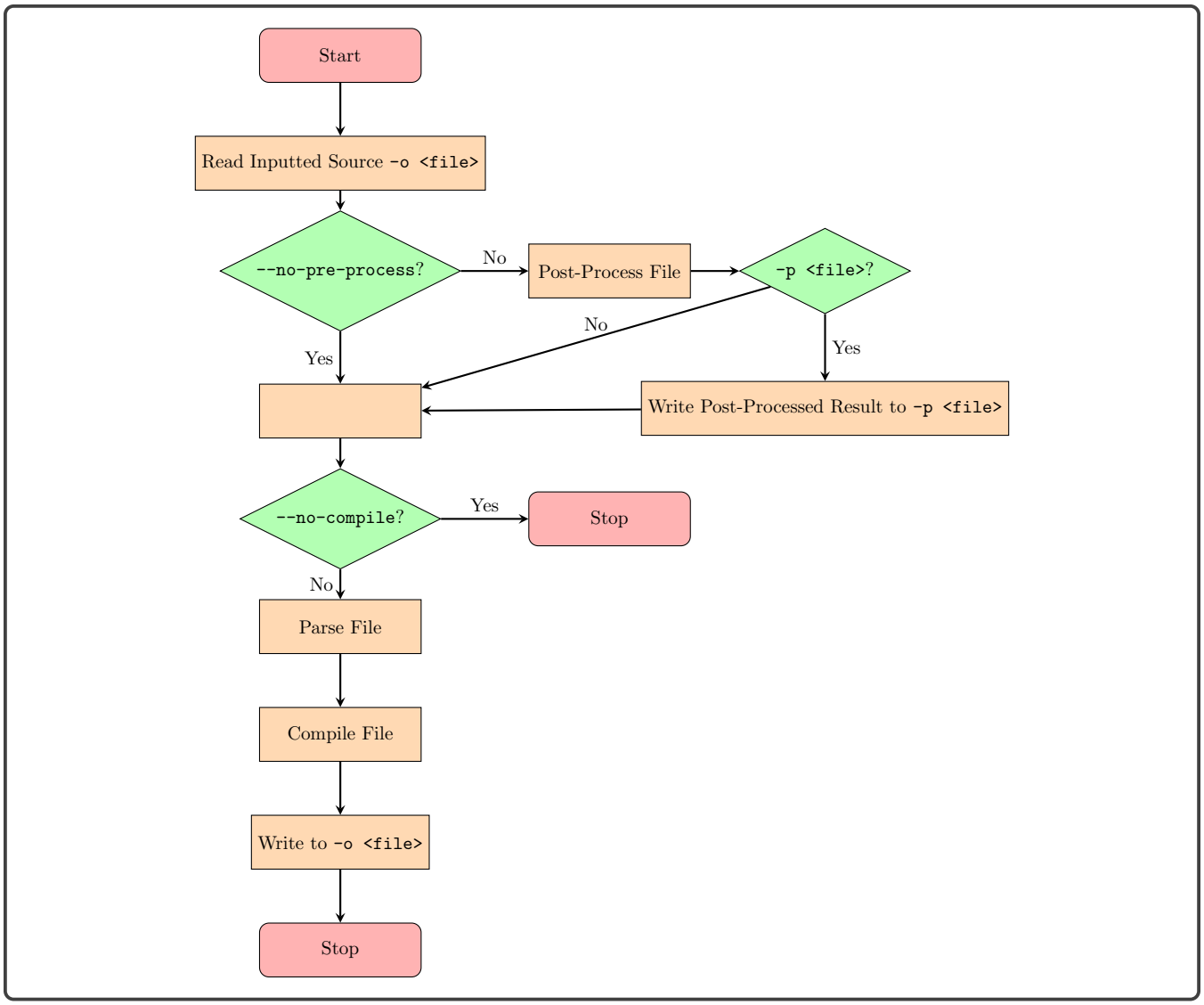
The assembler executable is called as follows:

```
$ ./assembler <input_file> -o <output_file> [flags]
```

The output file is provided after the `-o` flag. The following optional flags are available:

- `-d`: enables debug mode. In this mode, detailed results from each step are output to `stdout`.
- `-l <path>`: path of the library directory (see `%include`). The library path is calculated by `<executable path>/<lib path>`, with a default `<lib path>=url`.
- `--no-pre-process`: skips the pre-processing step.
- `--no-compile`: skips the compilation step. **Note** in this case, the `-o` flag is not compulsory.
- `-p <filename>`: writes the post-processed source to `filename`.
- `-r <filename>`: reconstruct the assembly from the compiled output and write it to `filename`.

1.2 Process Flow



2 Pre-Processor

The pre-processor runs prior to compilation. It modifies the source file's contents before being passed to the compiler. Core to the pre-processor are *directives*, which are commands to the pre-processor. Directives are prefixed with a percent (%) symbol.

Directive	Signature	Description
Constant Definition	`%define <name> <value>`	(Re-)Defines a constant `name` with the given value. **Note** `<value>` extends to the end of the line. **Note** the start is trimmed of whitespace, but trailing whitespace is preserved.
End	`%end`	Marks the end of a macro definition.
Include File	`%include [lib:]<path>`	Reads the file contents of `<path>.asm` and inserts into source. **Note** the `lib:` prefix navigates to the library directory (specified by the `-l` flag).
Macro Definition	`%macro <name> [<args ...>]`	(Re-)Defines a macro with the given name and arguments. See the below section for more information.
Ignore Line	`%rm <data ...>`	This removes the line from the source.
Halt & Cut	`%stop`	Stops the pre-processor at this line. This and all lines succeeding it are removed from the source.

2.1 Macros

Macros are like pre-processor functions, and may be used to for meta-programming purposes and simplifying code. Macros have a name and a number of arguments. The macro's body extends from after the newline to the next `%end` directive. Note that arguments do not have types, as types do not exist at this level.
When referenced, the data after the macro name are split by whitespace and passed position-wise to the arguments. The argument names are substituted with their values in the macro's body before the reference is itself substituted by this body.
For example,

```
%macro inc reg
    add reg, reg, 1
%end

inc $r1
```

is replaced with

```
add $r1, $r1, 1
```

3 Assembly Syntax

After pre-processing is the parsing & compilation stage. This section will cover the syntax of the assembly language. Note that some syntax, such as addressing modes, is covered in the processor documentation document.
The parser reads the source line-by-line, top-to-bottom. The most common syntax is:

```
[label:] <mnemonic>[conditional][.datatype] [args ...] [; comment]
```

Alternatively, the line may start with a directive. Note, this is different from a pre-processor directive; see below for more information.

3.1 Labels

Labels are defined at start of a line, starting with an alphabetic character or an underscore and any number of alphanumeric characters or underscores, and finally a colon.
A label is a symbol name tied to a memory address; when defined, a label adopts the value of the byte offset in memory. A label may be redefined, in which case its address value will be updated.
A label is referenced in an argument by stating its name. How a reference is handles depends on if the label is defined:

- If it has been defined, the label is directly substituted by its address as an immediate, `<imm>`, or as an address reference, `<addr>`, as expected by the instruction signature.

- Otherwise, this is known as a forward-reference. This reference is noted and, when the label is defined in the future, this reference will be replaced by the address. All forward-references must be resolved by the end of compilation, or an error will be raised.

Special Labels There exist a few special labels:

- `main` – sets the entry point of the program. By default, the entry point is `+0x00`.
- `interrupt_handler` – sets the location of the interrupt handler. By default, the location of the interrupt handler is `+0x400`.

Special labels may only be declared once.

3.2 Mnemonics

Mnemonics are used to introduce instructions, and define the desired function to be taken. Any arguments supplied provide necessary data for the function to carry out its operation. The processor documentation document details all legal instructions.
The mnemonic itself consists of three parts.

- The base: this is the operation listed in the ISA table, and is the function name.
- Conditional test: this is a string such as "z", "eq", etc. that may be optionally suffixed to most instructions.
- Datatype: declared as a suffixed dot followed by a type string, this determines the type of data to be operated on in the case of some instructions.

Again, the above table only lists brief description; see the ISA table for more information on each part.

3.2.1 Pseudo-Instructions

Pseudo-instructions are marked as such in the processor specification. As a RISC processor, the instruction set is limited and compact. Pseudo-instructions are compromises for common operations, written as an intrinsic instruction but expanded into its equivalent intrinsic form by the assembler.

3.3 Arguments

The argument types are detailed in the process documentation document, but syntax will be detailed here.

- Immediate: a numeric value, e.g., `42`.
- Memory Address: a parenthesised numeric value, e.g., `(42)`.
- Register: a register name suffixed by a dollar '$', e.g., `$r1`.
- Register Indirect: a parenthesised register, optionally prefixed by a numeric value, e.g., `2($sp)`.

The following list the possible forms of a numeric literal.

- Integer: a sequence of digits of a given base. A base can be specified with a $0b$ prefix, where b is a base specifier, one of:
  - `b`: binary, base-2.
  - `d`: decimal, base-10, the default.
  - `o`: octal, base-8.
  - `t`: trinary, base-3.
  - `x`: hexadecimal, base-16.

  E.g., `42`, `0xf5`.
- Decimal integer: same as above, but contains a decimal point. Bytes are inserted as either a float (32-bit) or double (64-bit). E.g., `3.14`, `0xf2.e`.
- Character literal: a character enclosed in apostrophes yields the character's ASCII value. Escape sequences are also supported by using a backslash prefix. E.g., `'a'`, `'\0'`, `'\n'`.

3.4 Directives

Similar to pre-processor directives, these give commands to the compiler and do not relate to actual instructions. A directive is prefixed with a dot, and must be at the start of a line.

Directive	Syntax	Description
Loading Data These directives are for inserting data into memory in-place. Arguments have the same syntax as an `<imm>`, but may also be strings. Strings automatically insert a null character after it is closed.		
Load Byte(s)	`.byte ...`	Load a sequence of 8-bit bytes into memory.
Load Data	`.data ...`	Load a sequence of half-words (32-bit) into memory.
Load Word(s)	`.word ...`	Load a sequence of words (64-bit) into memory.
Reverse Space	`.space` n	Reserve n bytes of memory.
Manipulating Location		
Set Offset	`.offset` n	Set positional offset in bytes to n. **Note** a warning will be generated if decreasing behind current offset.

3.4.1 Data Directives

`.byte`, `.data`, and `.word` are used to load data in-place into a binary. Their argument are space– or comma-separated units of data.

- A numeric literal as described in section 3.3, e.g., `42`.
- A character literal, enclosed in single quotes, e.g., `'b'`.
- A string literal, enclosed in double quotes, e.g., `"Hello"`. Note, the string will be null-terminated.
- A declared label (meaning labels cannot be used prior to declaration here).

Each unit will be inserted as dictated by the expected size of the directive, i.e., `.data` will load each character in a string as an integer. Labels will always be unsigned words.
If no data is provided, a single immediate of zero will be assumed. I.e., `.data` is the same as `.data 0`.

4 Assembly Reconstruction

If the `-r` flag is provided, the assembly source will be reconstructed. This reconstruction is a one-to-one matching to the compiled machine code, as all pseudo-instructions have been expanded by this point.
If the debug flag, `-d`, is present, additional debug info is printed to the file as comments. Each line is suffixed by a comment in the form

```
... ; <source>+<offset>
```

where

- `source` is the `<file>:<line>` of its original location (prior to pre-processing and expansion of pseudo-instructions).
- `offset` is the byte offset of this line in RAM. (This is equal to $pc for an instruction.)