General Assembler System Specification

Preface

This is the System Specification for GenA, a General assembly Program.

Throughout this specification, whenever one section references an idea from another section, there will be a reference and a hyperlink to the referenced section. The purpose of this, above all, is to stress the modularization of the system's specification. A consequence of this, hopefully, is a swift conclusion to the inquiries of GenA's users, no matter the nature of their inquiry, so long as it's about what GenA does.

In any file mentioned in this specification, GenA will concatenate the next line after a line that ends with the character ' '. It will then remove any white space between the beginning of a line and the first character and also remove any white space between the last character and the end of the line.

All things mentioned in this specification are case insensitive unless otherwise stated.

Henceforth, mention of AVR ISA will be in reference to this resource AVR® Instruction Set Manual (microchip.com).

How general is general? GenA can assemble code to **any** digital processor. GenA comes with an AVR ISA (Instruction Set Architecture) file (see ISA File) and if the user wishes to assemble code for a different processor, they must use the format stated in the ISA File section to create the ISA for their processor.

Groups of identical delimiters/separators, will be referred to in this document as a single unit, and will be recognized in inputs as a single unit, that unit being, naturally, the only unit present in the group.

Description

GenA is a two pass general assembly program capable of assembling assembly programs (seeAssembly Files) written for any processor. Assembled programs are converted into Intel HEX machine code and a listing file is provided (see Outputs). GenA supports labels, forward references, and a variety of pseudo operations (seePseudo Operations). GenA is distributed with an ISA file for the AVR processor but also allows for user specified ISA files (see ISA File). GenA is command line based (see User Interface) and written in C++.

Inputs

Assembly Files

GenA takes in, through the command line, the path to an assembly file. (see User Interface). All included assembly files (see Pseudo Operations) must share the same extension as this first file. Assembly begins with that first file.

Assembly files must adhere to the ISA specified in the user provided ISA file (see the ISA File). GenA is distributed with an AVR ISA file (avr_isa.txt) and it can be used to assemble AVR Assembly programs. The program counter can be referenced in assembly lines using the `$` special character.

Include Files

Include files must have extension `.inc` and are used to store symbolic constants. These constants can only be used in assembly files that include these files (see Pseudo Operations for file inclusion). Comments in include files can be declared by using the ";" character and end with the end of the line.

Pseudo Operations

GenA supports pseudo operation usage for the provided assembly files. Only one pseudo operation can be used per line and they must follow the syntactical format specified. The supported pseudo operations are:

- File inclusion
- Section headers
- Code organization
- Constant literals
- Symbolic constants
- In line expressions

File Inclusion A file includes another file by using following format:

```
.include "[path to file]"
```

The file must an assembly or include file and the path is relative to the file that includes it.

During assembly, the contents of the included assembly files are put in place of this pseudo operation. Files that have already been included by other files will not be included again by GenA. In general, it is best practice to not have circular includes.

Section Headers Section headers are used to divide the assembly files up into code and data sections. `.code` and `.data` must exist as the only string in their line.

`.code` All assembly code must be written in a `.code` section. A `.code` section is a section of an assembly file between a line with the `.code` pseudo operation and either another section, or the end of the file.

`.data` (optional) This section can be used to declare the data segment for variables. The section is defined as a section of the assembly file between a `.data` pseudo operation and either another section or the end of the file. Two types of variables are supported in the `.data` section, local and global.

Local variables must be uniquely named across both local variables within the same file and all global variables in all files. For more on global variables see below. Local variables do not have to be uniquely named across all local variables in other files. Local variables must be declared like so:

```
VariableName: BYTECOUNT
```

Global variables are variables that must be uniquely named across all other global variables and all local variables and all labels in all files. For example, if there is a global "count" variable in one file, no other file can have a global or local "count" variable or label. Global variables must be declared like so:

```
.global VariableName: BYTECOUNT
```

The `VariableName` is the name of the variable (cannot include whitespaces) and `BYTECOUNT` is the decimal representation of the amount of bytes allocated to the variable.

Only `.global` variables can be referenced from any scope outside the assembly file it is contained in. Labels behave like variables when it comes to their naming rules and relationship with the `.global` pseudo operation.

It is best to have unique local, global, and label names across all files, as this is how these elements are normally incorporated into assembly files.

All variables and labels will be replaced in assembly code by their address in data and program memory respectively at assembly time. More on these memory spaces can be seen in the ISA file section.

Code organization The `.org` pseudo operation is used in the following manner:

```
.org <ADDRESS>
```

to set the line in code it precedes to that address in program memory. More on this piece of memory in the ISA file section. The start of assembly will default be 0 unless otherwise set by `.org`.

Constant Literals Constant literals are non-aliased values put into assembly code. By default, these values are interpreted as decimal numbers, however, it may be prefixed in the following ways:

- `0DVALUE` - decimal representation
- `0HVALUE` - hexadecimal representation
- `0QVALUE` - octal representation

- `OBVALUE` - binary representation

An example usage could be:

```
ADD R1, 0B00000001 ; add 1 to Register 1
```

Symbolic constants Symbolic constants can only be declared inside an include (.inc) file. Symbolic constants must be declared like so:

```
.define NAME VALUE
```

where `NAME` is the name of the symbol (no white spaces) and `VALUE` is the associated value. `VALUE` is interpreted like how a constant literal is by GenA.

An example definition could be:

```
.define BINARY_ONE 0B00000001                ;binary form of 1
```

For assembly, whenever a string matching any constant is found in the Operand of an assembly line, it will be replaced with its corresponding symbolic value. Symbolic value names must be distinct across all include files.

In-Line Expressions The operators, `+`, `-`, `*`, `/`, `%`, `<<`, and `>>` are supported between any combination of symbolic constants, literals, labels, and variable names.

ISA File

A single ISA (Instruction Set Architecture) file must be included in the list of files of in the folder specified by the user through the command line (see User Interface). GenA comes with an AVR ISA file (avr_isa.txt) and if the user wants to supply their own for another processor, it must be in the following format.

The first line must be the relative path to a C++ file. This file functions only within the scope of this ISA file and the standard for such a file is rigorously defined and so its format, as it pertains to the ISA file, will briefly explained without an entirely new section. This C++ file must contain one or more functions with the **case sensitive** names `F1, F2, F3, ...` and so on. These functions will be discussed in detail later. An example is as follows:

```
"C:\Users\user\GenA_ISA\functions.cpp"
```

The second line must be the decimal representation of the number of bits in a single word of the processor's program data then the data data separated by a whitespace. If your processor shares program and data space then only put a single number.

The third line must be the decimal representation of the number of words in the processor's program data then the data data separated by a whitespace. If your processor shares program and data space then only put a single number.

In line four, the user must define the order and delineation of the elements in a line of assembly. A line of assembly can contain the following elements:

- Label
- OpName
- Operand
- Comment

Delimiters must be distinct, and the last element must be the Comment, which is delimited by the end of the line. The other elements can be arranged in any order. In this first line there cannot be white space between any element and its following delimiter unless the delimiter is a white space. An example of this line is as follows:

`Label: OpName Operand; Comment`

Lines of assembly must follow this order and delineation, although they may not need to include all elements.

Every element can stand alone except the Operand, which must always have an OpName in the same line. Every element can exist in the same line with any number of the other elements. Every element can only exist in a line once.

Labels follow the same uniqueness rules as variable names and hold the same relationship to the `.global` pseudo operation (see Pseudo Operations).

All following lines must take the form of a Code Macro.

A Code Macro is used to specify how the OpName and Operand can be converted to machine code. A Code Macro consists of the following strings separated by white spaces.

- 1st string: the string representation of the `OpName`.
- 2nd string: the hex representation of the `OpCode` number.
- Next set of strings: combination of Symbols and Values.
  - Symbols will be represented as `Sym` followed by the string corresponding to it.
  - Values will be represented as `Val`. (e.g. `SymR Val` for R16)
- Last strings must be the **case sensitive** function name of one of the functions `F1, F2, F3, ...` within the specified C++ file followed by the decimal representation of the number of bits in the instruction. These are to be separated by a white space.

The `OpName` is used to identify Code Macros while parsing assembly and the functions specified in those Code Macros are used to turn the `OpCode` and the `Operand`, in the form of symbols and values, into machine code.

Each function must take in the `OpCode` as an integer, and each `Val` as a string. Each value will be the entire string after the previous Symbol and before the next Symbol or the end of the line **excluding white space**. Each function must output a signed integer that fits into the specified bit count for the Code Macro it is used in. An `n` bit number should be between `-2^(n-1) - +2^n - 1`. Each function has access to a global variable called `pc` for the program counter.

For example the Code Macro for the instruction `ADD Rd, Rs` might look as follows:

```
ADD AA SymR Val Sym,R Val F1 16
```

`F1` must, in this case, return a 16 bit integer which might derive from the `OpCode` followed by the `d` and `s` as integer encodings. When parsing a line of assembly that looks like this:

```
ADD R1, R2
```

the inputs to `F1` would be `AA` as an integer and `1` and `2` as strings. These could be converted to decimals and concatenated to the `OpCode` to produce the instruction. In general, Code Macros and their accompanying functions can produce the ISA for **any** digital processor.

Outputs

Upon successful completion, GenA will output a complete message to the command line. GenA can output two files, whose names are specified on the command line, to a single folder, whose name is also specified on the command line (see User Interface). The first is the machine code object file. One object file will be produced per successful assembly, and it will follow Intel HEX format and thus have `.hex` extension.

The second is a listing file (`.lst`). The first thing on every line of the listing file is the line number relative to the listing file. Everything past the ";" character in a line is considered a comment. Comments contain the header, the symbol table, and user code. For lines where user code has instructions, the program data and address precede the comment in hexadecimal format. An example portion of a listing file output could be:

```
1; FolderName assembled by GenA with config file "config.txt"
2; Start of code:
3 0000       ; UpdateTimer:                ; Update the timer by 1s
4 0000 1A34  ; LDD R16, counter            ; retrieve the counter
5 0001 1C7F  ; ADIW R16, 1_SECOND          ; add 1 second


6 0010 261A  ; STS counter, R16            ; store the new value
```

All previous sections are required to understand the next two sections.

User Interface

GenA is command line based. To use GenA the user must first clone the GitHub repository `https://github.com/josharchibald/General-Assembler`. The user must then navigate to the `General-Assembler/GenA` directory and run `make`. Now the command `gena` can be run from anywhere in the terminal. `gena` is used with the following arguments.

- `-h`
  - (Optional) The help flag will display a usage message. Stops the program.
- `-f`
  - The main file flag must be followed by the relative path to the main assembly file for the program the user wishes to assemble.
- `-i`
  - The ISA file flag must be followed by the relative path of the ISA file for the processor the user wishes GenA to assemble for.
- `-o`
  - (Optional) The output folder flag must be followed by the relative path of the folder the user wishes the output to go to. Outputs will share the folders name. If the folder does not exist it will be created. If the folder does exists, files within maybe overwritten. If this flag is not specified, this directory takes the default value of a directory with the same name as the main file, without the extension, in the wordking directory.
- `-l`
  - The list flag must be present for a listing (`.lst`) file to be produced.
- `-v`
  - the version flag will display a version number, last update date, and a link to the most recent github repository. Stops the program.

Flags and their arguments must be separated by spaces. Each path flag, `-f`, `-i`, and `-o`, can only be used once.

Error Handling

If an error of any kind occurs, all file outputs will not be produced.

All errors are as follows:

- The `-i` flag is not specified when the `-f` flag is.
  - Usage error. Stops program.
- The `-f`, `-i`, or `-o` flag has been duplicated.
  - Usage error. Stops program.
- The file after the `-f` cannot be found. Stops program.
  - `Error: [file path] file not found`
- The C++ file specified in the first line of the ISA file cannot be found. Stops program.
  - `Error: ISA file [ISA file path] C++ file [C++ file path] not found`
- Lines two or three in the ISA file are not configured as specified. Stops the program.
  - `Error: ISA file [ISA file path] missing memory information on line [line number]`
- Line four in the ISA file contains repeated delimiters. Stops program.
  - `Error: ISA file [ISA file path] line 4 repeated delimiter '[repeated delimiter]' for [element] and [element]`
- Line four in the ISA file is missing elements. Stops program.
  - `Error: ISA file [ISA file path] line 4 missing element(s): [missing element(s)]`
- The OpCode for a Code Macro is not a valid hexadecimal number.
  - `Error: ISA file [ISA file path] line [line number] invalid OpCode "[Opcode]"`
- A Code Macro is formatted incorrectly.
  - `Error: ISA file [ISA file path] line [line number] invalid string "[invalid string]"`
- An included file is not of the right extension.
  - `Error: Assembly file [file path] line [line number] file [file path] has invalid extension`
- An included file cannot be found.
  - `Error: Assembly file [file path] line [line number] file [file path] cannot be found`
- A line of assemble is formatted incorrectly.

  - `Error: Assembly file [file path] line [line number] invalid string "[invalid string]"`

- A pseudo operation is not recognized.
  - `Error: Assembly file [file path] line [line number] has unrecognized pseudoOp [pseudoOp]`
- A line of assemble has an Operand with no OpName.
  - `Error: Assembly file [file path] line [line number] has Operand with no OpName [line elements]`
- There are duplicate labels.
  - `Error: Assembly file [file path] line [line number] and Assembly file [file path] line [line number] have duplicate label [label name]`
- An OpName is not recognized.
  - `Error: Assembly file [file path] line [line number] has unrecognized OpName [line elements]`
- An Operand is invalid.
  - `Error: Assembly file [file path] line [line number] invalid Operand [line elements]`
  - The program will display this message if a function used from the C++ file supplied in the ISA file returns `NULL`. It is up to the function to display a more detailed error message.
- Memory limit exceeded.
  - `Error: Assembly file [file path] line [line number] exceeds [memory space] memory limit of [number of words] words`
- An instruction is too large.
  - `Error: Assembly file [file path] line [line number] instruction larger than [number of bits] bits [line elements]`