The SSM Assembler

An assembly program starts with a *code section*. This is optionally followed by a *data section*. See Figure 1. Blank lines and single-line comments are permitted and are ignored by the assembler (as in Java, a single-line comment starts with `//` and extends to the end of the line).

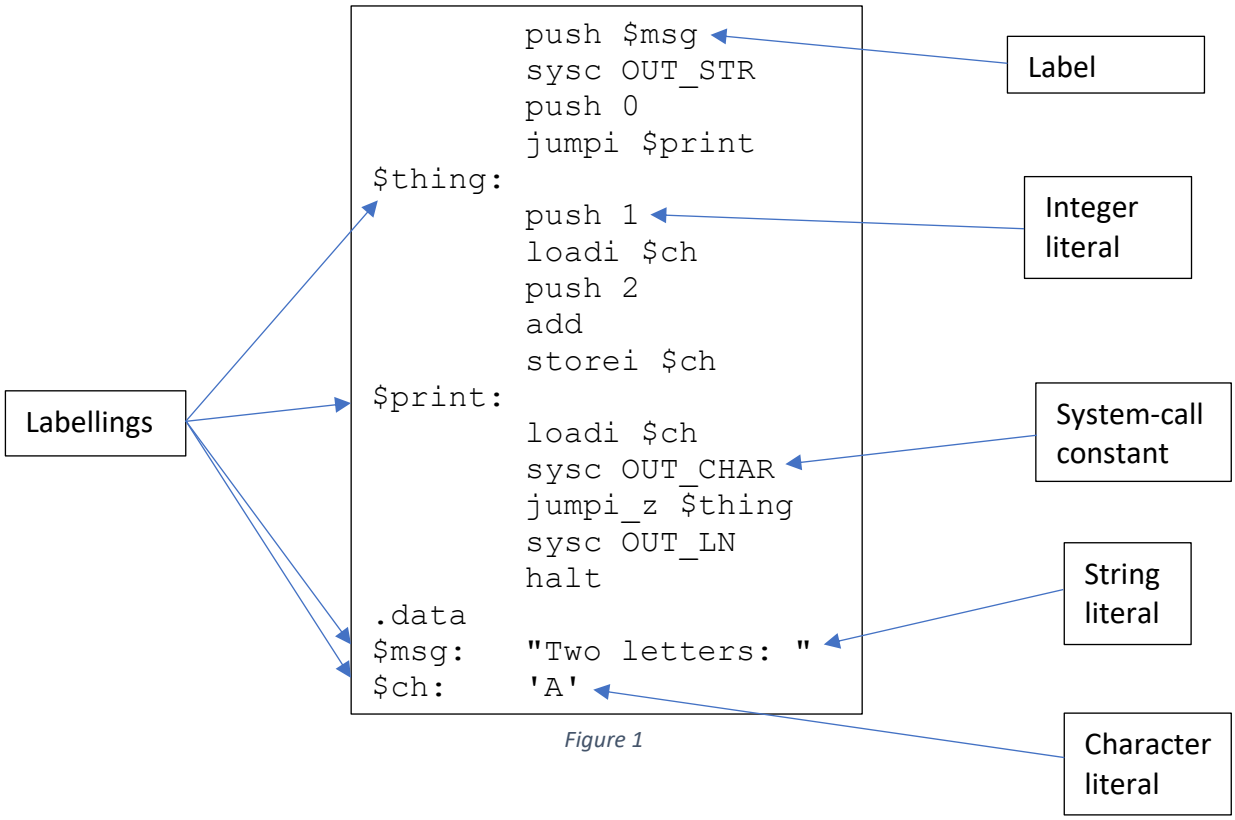

Figure 1

Code

The code section consists of a sequence of instructions, with each instruction on a separate line. The opcode for an instruction is written as a so-called "mnemonic", which is simply a name (see *The SSM Instruction Set* for the full list of instructions and their mnemonics). For instructions which occupy more than one byte, the mnemonic must be followed by a literal which specifies the remaining bytes (in some cases, a *label* can be used instead of a literal; see below). There are four kinds of literal which can be used for this purpose:

> **Integer literals (4-bytes)**: signed-decimal integer literals. Examples: `0`, `-9`, `2764`.
> **Character literals (4-bytes)**: a single ASCII character in single quotes. Examples: `'w'`, `'9'`, `'\n'`. Escape sequences can be used in character literals, as specified in the Java documentation for String.translateEscapes(). The 1-byte ASCII code is left-padded with zeroes to make a 4-byte word.
> **Hex literals (1-byte)**. A hexadecimal literal, starting with `0x` and ending with exactly two hexadecimal digits, representing a 1-byte unsigned integer. Example: `0xf3` (equivalent to decimal `243`).
> **System-call constants (1-byte)**. The SSM system calls are numbered (0, 1, etc.) and each of these numbers has a corresponding symbolic constant name (OUT_BYTE, OUT_CHAR, etc) which is recognised by the assembler as a 1-byte unsigned integer.

The assembler allows any kind of literal to be used as part of an instruction, even if the byte-length of the literal value (shown above in brackets) does not match the number of bytes required for the instruction. In case of such a mismatch, the data defined by the literal will be either left-padded with zeroes or some high-order bytes will be discarded, as required. Note that the byte-length of a character literal is defined to be four bytes, even though only 1-byte ASCII characters are supported by the SSM; all character literals therefore define a four-byte word where the three high-order bytes are all zeroes.

Data

The data section starts with `.data` followed by a sequence of literals, each on a separate line. The effect is to insert the bytes defined by the literals into the generated binary immediately after the instruction sequence. No padding is applied: each literal generates a fixed number of bytes as specified by its byte-length.

There is one additional kind of literal which can be used in the data section but not in the code section:

> **String literal**: a sequence of 8-bit ASCII characters enclosed in double quotes. Examples: `"Hello"`, `"Two\nLines"`. Escape sequences can be used in string literals, as specified in the Java documentation for String.translateEscapes(). The byte-length of a string literal is 2+*n* where *n* is the length of the string (after escape sequences have been translated). These bytes are made up of an unsigned 2-byte representation of the string length, followed by one byte per character in the string. Note that strings in this format can be printed using the OUT_STR system-call.

Labels

Each line (in both the code and data sections) can optionally be labelled with one or more labels (if there is more than one, each labelling must be on a separate line). A *label* can be any mixture of `$`-symbols, @-signs, letters, digits and underscore characters, but it *must* start with a `$`-symbol. A *labelling* is a label followed by a colon. Based on the position of the labelling, the assembler will "resolve" each label to its corresponding two-byte memory address within the generated SSM binary. Labels can be used instead of literals within the code section but only for push (in this case, the two-byte address is left-padded with zeroes to create a four-byte word) and instructions which include a memory address (loadi, storei, jumpi, etc). **Note**: labellings (like comments and blank lines) don't occupy any space at all (ie don't exist) in the generated machine code, so when the assembler is "counting bytes" in order to resolve labels to memory addresses, it skips over any lines in the code section which don't contain an instruction, and it skips over any lines in the data section which don't contain a literal.

Figure 2 (next page) shows a listing of the code from Figure 1 with "byte numbering" added. These numbers (which look like line-numbers in a conventional code listing) actually show the memory address which will contain (the first byte of) the numbered instruction or data-item. From Figure 2 we can see that the label `$thing` will be resolved to 15, `$print` will be resolved to 32, `$msg` will be resolved to 43 and `$ch` will be resolved to 58.

```
     0:    push $msg
     5:    sysc OUT_STR
     7:    push 0
    12:    jumpi $print
$thing:
    15:    push 1
    20:    loadi $ch
    23:    push 2
    28:    add
    29:    storei $ch
$print:
    32:    loadi $ch
    35:    sysc OUT_CHAR
    37:    jumpi_z $thing
    40:    sysc OUT_LN
    42:    halt
.data
$msg:
    43:    "Two letters: "
$ch:
    58:    'A'
```

Figure 2