COMP5570 Assignment 3
Writing an Assembler

Version 2022.12.05

You may notify me of errors or seek clarifications either via the anonymous question asking page (which will be set up in the Assessments section on the Moodle page), or via email to d.j.barnes@kent.ac.uk

4th December 2022: Important simplification for instructions involving ROM labels.

In order to simply the coding of jump instructions that reference ROM labels, I will definitely only assess your code with SAL test programs where **the binary value for a ROM label will fit into 6 bits**. This means that all jump instructions that reference a ROM label will be encodable in a single 16-bit instruction. However, be aware that this does not mean all jump instructions can be encoded in a single 16-bit value because it is also possible to use explicit numeric addresses in jump instructions, and such numeric addresses in the test data might require more than 6 bits. So:

```
    JLT r3, destination
```

will definitely be encodable in only 16 bits and have a Size bit of 0 but:

```
    JLT r3, 12345
```

will require 32 bits (22 bits of operand) and have a Size bit of 1.

Introduction

This assignment is designed to provide you with a practical understanding of the task of writing an assembler.

Date set: 24th November 2022.
Date due: 12th December 2022 Deadline 23:55.

You will write an assembler for a low-level language called SAL (Simplified Assembly Language). SAL is a language designed by me for this particular assessment so you should not expect to find any additional information about it elsewhere on the Web/Internet. This document is the definitive definition/description of SAL.

SAL offers a way to write simply assembly-level programs that are converted to strings of binary digits. SAL is not related to Hack but is closer in style to traditional assembly languages, and so offers a realistic learning experience.

You may ask questions about this assessment via email to d.j.barnes@kent.ac.uk or via the COMP 5570 anonymous question asking page, a link to which may be found in the Assessments section on the module's Moodle page.

The sections below describe the SAL language and the requirements of the assembler you must write.

It is possible that corrections might need to be made from time to time to this assessment, so please keep an eye on the Version date at the top of this page to ensure that you have the latest version.

Outline requirements

- You must submit the source files of a program that translates from SAL assembler to its binary translation. However, the 'binary' translation must actually be a text file containing 1 or more lines of text, each exactly 16 characters long, with each character being either 1 or 0. For instance:

  ```
  0000000110101010
  1100010001000100
  etc.
  ```
- The submission will be via an Upload area in the Assessments section of the module's Moodle page.
- It must be possible to run your program from the command line on raptor.
- Your program must take a single command-line argument which is the name of the SAL source file to be translated.
- The assembler must only accept source files with a '.sal' file suffix.
- The assembler must write a text file as output. The output file name must have the same prefix as the SAL source file but a '.bin' suffix. For instance, if the SAL file is called prog.sal then the output file must be called prog.bin. The file written to must be in the same directory/folder as the SAL source file.
- Translation of each SAL instruction will require either 1 or 2 16-character binary strings to be written to the output file. Each 16-character string must be written on a line of its own with no additional spacing around it, or blank lines in between instructions.
- Your program will only be tested with valid SAL source files, so you do not need to report on errors. However, if you think your program has detected an error in its input it must write a message on the standard error output (System.err in Java) in the following format, including the source file line number on which the error was detected. Your program should then exit:

  ```
  Error line NNN
  ```

  Where NNN is the line number on which the error was detected.

- You may implement the program in a language of your choice under the constraints that the language must be available to the marker on raptor and it must be possible for the marker both to compile (where required by the language) and execute your code without having to install a particular IDE or any other software, such as libraries or build tools. These constraints are important. Any libraries required must either be bundled with your submission or already pre-installed and accessible to the marker.

- It is **essential** that your submission includes sufficiently detailed instructions to allow the marker to run your programs, particularly if it is written in a language other than Java. Further details of what this will mean in practice will be provided before the deadline.

Plagiarism and Duplication of Material

The work you submit must be your own. We will run checks on all submitted work in an effort to identify possible plagiarism or other academic misconduct, and take disciplinary action against anyone found to have broken the University's rules on academic integrity. You are also reminded that seeking assistance from external 'homework' sites is not permitted and might constitute 'contract cheating' which is forbidden by the University.

Some guidelines on avoiding plagiarism:

- One of the most common reasons for programming plagiarism is leaving work until the last minute. Avoid this by making sure that you know what you have to do (that is not necessarily the same as how to do it) as soon as an assessment is set. Then decide what you will need to do in order to complete the assignment. This will typically involve doing some background reading and programming practice. If in doubt about what is required, ask a member of the course team.
- Another common reason is working too closely with one or more other students on the course. *Do not* program together with someone else, by which I mean do not work together at a single PC, or side by side, typing in more or less the same code. By all means *discuss* parts of an assignment, but do not thereby end up submitting the same code.
- It is not acceptable to submit code that differs only in the comments and variable names, for instance. It is very easy for us to detect when this has been done and we will check for it.
- **Never** let someone else have a copy of your code, no matter how desperate they are. Always advise someone in this position to seek help from their class supervisor or lecturer. Otherwise, they will never properly learn for themselves.
- It is not acceptable to post assignments on sites such as Chegg, Freelancer, etc. and we treat such actions as evidence of attempted plagiarism, regardless of whether or not work is paid for.

Further advice on plagiarism and collaboration is also available.

We reserve the right to apply checks to programs submitted for assignment in order to guard against plagiarism and to use programs submitted to test and refine our plagiarism detection methods both during the course and in the future.

The SAL language: lexical and syntactic conventions

A SAL program is stored in a file with a '.sal' suffix. The assembler must translate a single .sal file on each run. The input file must be named as a command-line argument. Any additional arguments must be ignored. For instance, if your program is written in Java with its main method in a class called Main, you would run it as follows:

```
    java -cp . Main prog.sal
```

Comments: As in both Java and Hack, text beginning with two forward-slash character (//) up to the end of the line on which it occurs is a human-readable comment and requires no translation. Comments are ignored by the assembler.

Whitespace: Blank lines are ignored by the assembler. Each SAL instruction or label must be written on a single line. Unlike in Hack, whitespace is used to separate an instruction mnemonic from its operands. Additional whitespace may be used anywhere within a line, for instance to indent instructions or enhance readability.

Instruction mnemonics: All instruction mnemonics are either 3 or 4 case-sensitive alphabetic characters long, entirely in upper-case.

Numeric constants: Numeric constants must be positive integer values, written in decimal notation, in the range 0-32767.

Labels: Labels are used as symbolic names for memory addresses. They consist of 1 or more alphabetic, numeric and underscore characters starting with an alphabetic character. SAL distinguishes between labels for ROM instruction addresses and those for RAM data addresses. All labels for RAM data addresses must be 'declared' in advance in a 'data section' that appears before the instructions. The instructions appear in a 'code section' (further details below). ROM labels are 'declared' by using them to label the following instruction.

Each RAM label corresponds to a 16-bit address, starting at 0. The first RAM label in the data section would be associated with RAM address 0, the second with address 1, and so on.

A label may not be used for a ROM address if it has been declared in the data section as a RAM label. A label must not match either a SAL instruction name (opcode) or register name.

Instruction addresses start at ROM address 0 and each instruction occupies either 1 or 2 16-bit addresses.

The SAL language: data and instructions

Note that 'translation' of the data section does not directly result in any code being generated. Rather, it should result in a symbol table being created within the assembler, associating each variable name with a corresponding address, starting at 0 and increasing sequential by 1 for each variable.

The following short example shows both data and code sections and how they are introduced in a SAL source file:

```
.data
    sum
    x
    y
.code
    LOAD r0, x
    ADD r0, r1
    INC r1
```

The data section is introduced by the symbol '.data' which must be on a line by itself. If there is no data then the data section may be completed omitted. If present, there must be zero or more names for RAM locations, each on its own line.

The instructions are introduced by the symbol '.code' which must be on a line by itself. If there are no instructions then the code section may be completed omitted. If present, there must be zero or more instructions and/or instruction labels, each on a separate line.

The SAL language: instruction labels

Instruction labels may only appear in the code section and correspond to ROM addresses. A label must be followed on the line by a colon character, optionally separated by spaces. The label name does not include the colon symbol. A label must appear on a line by itself but multiple labels, on successive lines, may be used to label the same instruction. For instance:

```
start:
loop:
    LOAD r0, x
    ADD r0, #345
    INC r1
    STORE r0, sum
    JMP loop
```

Instruction encoding: conventions

This section describes the instructions available in the SAL language and their binary encodings. SAL instructions are encoded as either 1 or 2 16-character binary strings.

Please note the following definitions of operands which are used in the descriptions:

- 'reg' is one of 8 registers: r0, r1, …, r7. Register names are not case-sensitive.
- 'addr' may be either a number (such as `2476`) or a variable label (such as `sum`).
- A 'value' operand is a positive integer (0-32767) written with a preceding '#' character (such as `#2476`).
- An operand may be either a register (reg), an address (addr) or a value. If present, an operand is always separated from the preceding register name by a comma.
- Labels used with any of the jump instructions must be instruction labels and not variable labels.

It is important to distinguish between the addr and value operands `2476` and `#2476`. The former is an 'addr' and the latter is a 'value'. They are distinguished by the presence or absence of a `#` character. See below for further details.

Instruction encoding: format

SAL instructions are encoded as either 1 or 2 16-character binary strings. The first 16-bits are encoded as follows:

Opcode	**Reg**	**Size**	**RVA**	**Operand**
xxxx	xxx	x	xx	xxxxxx

- Opcode – 4 bits. Binary encoding of the operation.

- Reg – 3 bits. Register r0 == 000, r1 == 001, …, r7 = 111. The JMP instruction does not use a register, so the Reg bits must be 000 in that case.

- Size – 1 bit. 0 if the instruction is 16 bits in length; 1 if it is 32 bits in length. The size bit provides a way to keep instructions down to 16 bits rather than 32 if the Operand value is small enough to fit within 6 bits. For instance, #17 is small enough to fit into 6 bits, so the Size bit would be 0 but #2476 is too large, so the Size bit would be 1 and the extra 16 bits would be used.

- RVA – 2 bits. 00 if the Operand is a register; 01 if the Operand is a numeric value; 10 if the Operand is an address; 11 if there is no Operand (INC, DEC, NOT – see below).

- Operand – 6 or 22 bits.
    - If the Size bit is 0 then the instruction is only 16 bits and the Operand field is 6 bits. The Operand field contains either the identity of a register (RVA bits are 00), a 6-bit literal value (RVA == 01), or a 6-bit address (RVA == 10).
    - If the Size bit is 1 then the full operand is the concatenation of the 6-bit operand field and the following 16 bits. The 6-bit operand field contains the most-significant bits of the operand. Note that if the Size bit is 1 then RVA field can only be set to either 01 or 10, because the additional 16-bits are not required to encode a 3-bit register value in the Operand field.
    - If RVA is 11 (INC, DEC and NOT take no second operand) then the Size bit must be 0 and the 6-bit operand field must be 000000.

Instruction encoding: opcodes

These are the 4-bit opcode encodings of the SAL instructions. All non-jump operations leave the result in the register named immediately after the operation.

Instruction	Encoding	Format	Operation
ADD	0000	ADD reg, operand	Add operand to the value in register reg.
SUB	0001	SUB reg, operand	Subtract operand from the value in register reg.
AND	0010	AND reg, operand	Bitwise And operand with the value in register reg.
OR	0011	OR reg, operand	Bitwise Or operand with the value in register reg.
JMP	0100	JMP addr	Jump to the given address.
JGT	0101	JGT reg, addr	Jump to the given address if the value in reg is greater than zero.
JLT	0110	JLT reg, addr	Jump to the given address if the value in reg is less than zero.
JEQ	0111	JEQ reg, addr	Jump to the given address if the value in reg is equal to zero.
unused	1000		
INC	1001	INC reg	Add 1 to the value in register reg.
DEC	1010	DEC reg	Subtract 1 from the value in register reg.
NOT	1011	NOT reg	Bitwise Not the value in register reg.
LOAD	1100	LOAD reg, operand	Copy the operand into register reg. If the operand is an address then the value at the corresponding RAM location is copied into reg.
STORE	1101	STORE reg, operand	Store the value in register reg to the operand location. Operand can only be either a reg or addr.
unused	1110		
unused	1111		

It is not necessary to understand the exact details of each operation, simply how it must be encoded.

Example encodings

In the following examples, colon characters have been used between the various fields in the Encoding column to make it easier to match the encodings to the encoding format. These must not appear in the program's output.

Instruction	Encoding	Notes
ADD r3, #5	0000:011:0:01:000101	ADD opcode is 0000. R3 encoding is 011. Size is 0 because the operand fits in

		6 bits. RVA is 01 because the operand is a value. The operand is 000101 as it encodes the value 5.
ADD r3, 5	0000:011:0:10:000101	ADD opcode is 0000. R3 encoding is 011. Size is 0 because the operand fits in 6 bits. RVA is 10 because the operand is an addr. The operand is 000101 as it encodes the value 5.
ADD r3, r0	0000:011:0:00:000000	ADD opcode is 0000. R3 encoding is 011. Size is 0 because the operand fits in 6 bits. RVA is 00 because the operand is a reg. The operand is 000000 as it encodes r0.
ADD r3, #256	0000:011:1:01:000000 0000000100000000	ADD opcode is 0000. R3 encoding is 011. Size is 1 because the operand does not fit in 6 bits. RVA is 01 because the operand is a value. The 22-bit operand encodes the value 256.
NOT r1	1011:001:0:11:000000	NOT opcode is 1011. R1 is 001. Size is 0 because there is no Operand. RVA is 11 because there is no Operand. The operand is 000000 because there is no Operand.
JMP loop	0100:000:0:10:110001	(Assuming loop is address 49). JMP opcode is 0100. Reg is 000 because it is a JMP instruction. Size is 0 because the Operand fits in 6 bits. RVA is 10 because the operand is an addr. Operand is 110001, the encoding of 49.

Assessment

Marks will be based entirely on functionality. You will **not** be evaluated on your coding. Therefore, it is essential that your submission can be executed. If it cannot be executed, it will be impossible to assess its functionality and you will automatically receive a mark of zero.

Your implementation will be marked with the help of an automatic test harness. The marking test harness will invoke the assembler with a variety of source files containing examples of the different instructions and other features of the SAL language. Some sample will be provided before the deadline for you to self-test. Marks will be awarded according to how well your code performs in these tests. An implementation that functions correctly for all tests will score 100%. We will not be releasing the test data used for marking so it is important that you supplement the tests we do provide with additional ones of your own.

Date of this version: 2022.12.05
Updates to the original version: `JMP sum` changed to `JMP loop` to avoid confusion.
Corrected the 2nd binary string output for ADD r3, #256 as there were too many bits in the previous version.
ROM label simplification added, for assessment purposes.
Explicit statement that an operand is always separated from the preceding register name with a comma.