## COMP5570 Assignment 3 Writing an Assembler Copyright 2023 David J. Barnes Version 2023.12.05

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](mailto:d.j.barnes@kent.ac.uk)

## Use of AI

**You are forbidden from using AI assistance to complete this assignment.**

We are fully aware that the use of AI (such as ChatGPT) to solve assignments is increasing.

However, for this module, I believe that doing so would prevent you from achieving some of the important learning outcomes of this module. Use of AI, therefore, will be treated as cheating: you would be submitting work that you have not done, in order to gain marks towards a degree.

If you are caught cheating by using AI assistance then you will face academic misconduct procedures which could be placed permanently on your academic record.

## Introduction

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

Date set: 21st November 2023.

Date due: 11th December 2023 Deadline 23:55.

You will write an assembler for a low-level language called MAL (My Assembly Language). MAL 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 MAL.

MAL offers a way to write simply assembly-level programs that are converted to strings of binary digits. MAL is slightly 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](mailto: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 MAL 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.

## Starter source code

Starter source code has been provided as two Java classes: Main and Assembler. The Main class already codes several of the requirements given below. The Assembler class is a simple outline that reads the contents of the input file reader for translation but does not translation. You are free to use as much or as little of the Assembler class as you wish. The code is available from the Assessments section of the Moodle page.

## Outline requirements

* You must submit the source files of a program that translates from MAL assembler to its binary translation. However, the ‘binary’ translation must actually be a text file containing 1 or more lines of text, each exactly 8 characters long, with each character being either 1 or 0. For instance:  
    
   10101010  
   01000100  
   etc.
* The submission will be via an Upload area in the Assessments section of the module’s Moodle page.
* Your program must be written in Java and it must contain a class with a main method. The class with the main method must be called Main. Sample starter code has been provided to help you with this.
* Your program must take a single command-line argument which is the name of the MAL source file to be translated. The sample code does this.
* The assembler must only accept source files with a ‘.mal’ file suffix. The sample code does this.
* The assembler must write a text file as output. The output file name must have the same prefix as the MAL source file but a ‘.bin’ suffix. For instance, if the MAL file is called prog.mal then the output file must be called prog.bin. The file written to must be in the same directory/folder as the MAL source file. The sample code does this.
* Translation of each MAL instruction will require either 1 or 2 8-character binary strings to be written to the output file. Each 8-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 MAL 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.

## The MAL language: lexical and syntactic conventions

A MAL program is stored in a file with a ‘.mal’ suffix. The assembler must translate a single .mal file on each run. The input file must be named as a command-line argument. Any additional arguments must be ignored. With its main method in a class called Main, you would run it as follows:

java -cp . Main prog.mal

**Comments**: There are no comments in the MAL language.

**Whitespace**: Blank lines are ignored by the assembler. Each MAL 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. Apart from the whitespace separating an instruction name from its operands, whitespace is otherwise completely optional and must not be relied on elsewhere as a separator.

**Opcodes**: All opcodes (instruction names) are either 3 (e.g., ADD), 4 (e.g., COPY) or 5 (e.g., LOADN) 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 0-255.

**Commas:** A comma is used to separate two operands in those instructions that have two operands. There may be whitespace before and/or after a comma or no whitespace at all.

**Labels**: There are no labels in a MAL program.

Instruction addresses start at ROM address 0 and each instruction occupies either 1 or 2 8-bit addresses. Data addresses start at RAM address 0 and each data value occupies 16 bits.

## The MAL language: data and instructions

## Instruction encoding: conventions

This section describes the instructions available in the MAL language and their binary encodings. MAL instructions are encoded as either 1 or 2 8-character binary strings.

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

* ‘reg’, 'reg1' and 'reg2' refer to either register A or register D. Both registers are 16-bit registers.
* ‘num’ may be any positive integer in the range 0 to 255 (inclusive).

## Instruction encoding: format

MAL instructions are encoded as either 1 or 2 8-character binary strings.

There are 10 opcodes in the language and are encoded by 4 bits. The 4 bits are encoded as the top (most significant) 4 bits of each instruction. The 4-bit patterns are as follows:

|  |  |
| --- | --- |
| Instruction | Encoding |
| LOADN | 0000 |
| LOADA | 0001 |
| ADD | 0010 |
| SUB | 0011 |
| JMP | 0100 |
| JGT | 0101 |
| JLT | 0110 |
| JEQ | 0111 |
| COPY | 1000 |
| STORE | 1001 |

When included in an instruction, register A is encoded as the two bits 01 and register D is encoded as the two bits 10. For example, the instruction ADD A,D would be encoded as 00100110:

0010 (the top 4 bits) encode ADD

01 (the next 2 bits) encode register A

10 (the bottom 2 bits) encode register D

When an instruction includes a 'num' operand, num is always encoded as the 8-bit binary representation of the value of num. The 8 bits follow the 8-bit encoding of the rest of the instruction. So, instructions containing a 'num' operand are always encoded as two 8-bit values. As only positive values of num are allowed, the full 8 bits are used for its encoding. For example, 255 would be encoded as 11111111 and 3 would be encoded as 00000011.

The instruction LOADA 254 would be encoded as:

00010000 (0001 for LOADA and 0000 for no registers)

11111110 (the encoding of 254)

When an instruction includes both a reg and a num, the reg value is encoded as two bits immediately after the 4 bits of the opcode and the bottom 2 bits are zero, while the num is encoded as the next 8 bits.

The instruction JLT A, 3 would be encoded as:

01100100 (0110 for JLT, 01 for A, 00 as padding)

00000011 (the encoding of 3)

## Instruction formats and operations

This is the complete set of instructions and their meanings.

|  |  |  |
| --- | --- | --- |
| Instruction | Format | Operation |
| LOADN | **LOADN** num | Load the value num into register A. Note that register A is implicit and is not encoded in the translation. |
| LOADA | LOADA num | Load the value stored at memory location num into register A. Note that register A is implicit and is not encoded in the translation. |
| ADD | ADD reg1 , reg2 | Add the value in reg2 to the value in reg1, leaving the result in reg1. |
| SUB | SUB reg1 , reg2 | Subtract the value in reg2 from the value in reg1, leaving the result in reg1. |
| JMP | JMP num | Jump to the instruction whose address is num. |
| JGT | JGT reg , num | Jump to the instruction whose address is num if the value in reg is greater than zero. |
| JLT | JLT reg , num | Jump to the instruction whose address is num if the value in reg is less than zero. |
| JEQ | JEQ reg , num | Jump to the instruction whose address is num if the value in reg is equal to zero. |
| COPY | COPY reg1, reg2 | Copy the value in reg2 into reg1. |
| STORE | STORE num | Store the value in register A to memory location num. |

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 A,D | 0010:01:10 | The first 4 bits are the encoding of ADD, the next 2 bits encode register A and the final 2 bits encode register D. |
| LOADN 3 | 0000:0000  00000011 | The first 4 bits are the encoding of LOADN and the second 4 bits are padding. The second 8 bits are the encoding of 3. |
| LOADA 254 | **0001:0000**  11111110 | The final 0000 of the first 8 bits is padding to 8 bits as register A is not encoded explicitly. |
| JLT A,3 | 0110:01:00  00000011 | The final 00 of the first 8 bits is padding to 8 bits. |
| COPY D,A | 1000:10:01 | The first 4 bits are the encoding of COPY, the next 2 bits encode register D and the final 2 bits encode register A. |
| JMP 9 | 0100:0000  00001001 | The final 0000 of the first 8 bits is padding to 8 bits. |
| STORE 16 | 1001:0000  00010000 | The final 0000 of the first 8 bits is padding to 8 bits. |

## 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 MAL 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.

## 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.

Further advice on [plagiarism and collaboration](http://www.cs.kent.ac.uk/teaching/student/assessment/plagiarism.local) is also available.

Date of this version: 2023.12.05

2023.12.05: There was an 'N' missing from the formats table in the second column for the LOADN instruction.

2023.11.22: Correct to the encoding of the example LOADA 254 which was previously incorrectly encoded in the table of examples.