COMP5570 Assignment 3, 15% of Overall Mark
Writing an Assembler
Version 2024-11-18

Please notify me of errors, ask questions, or request clarifications either via email (s.marr@kent.ac.uk), or during the Tuesday drop-in sessions at 3pm in my office in Kennedy 210.

Use of AI

The use of generative AI is strongly discouraged.

While the use of AI systems such as ChatGPT and GitHub Copilot is common, research suggests that your own learning, the deeper understanding of concepts, and the long-term retention are reduced. Thus, students who rely on AI may not gain sufficient practical preparation to pass their exam.

If you choose to use generative AI, the general rules for academic integrity still apply and simply reproducing the verbatim output of any such tool may constitute plagiarism.

Introduction

This assignment is designed to give you a practical understanding of how to write an assembler and similar structured user input. It is 15% of the overall module mark.

Date set: 18th November 2024
Date due: 9th December 2024 at 23:59

You will write an assembler for a low-level language called New Hack Assembler (NHA). NHA is designed for this assessment, and there are no additional information on the Internet. Thus, this document is the only and definitive definition/description of NHA. For questions and clarifications, please contact me directly.

Like the Hack assembler seen in the lectures, NHA allows us to write assembly-level programs that are converted to strings of binary digits. NHA targets the same machine as Hack but is closer in style to traditional assembly languages, and thus offers a more realistic learning experience.

The sections below describe NHA 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 document to ensure that you have the latest version.

Starter source code

Starter source code has been provided as a single Java file that contains all relevant elements. The `Main` class already implements 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 no translation. The `TestInput` and `TestOutput` classes provide some basic test data for your convenience.

You are required to use the provide Java file. And while you can add new methods, classes, and other elements you may need, you will need to keep the basic functionality provided to you so that the code can be marked automatically. The code is available from the Assessments section of the Moodle page.

Outline requirements

- You must submit your implementation of a *New Hack Assembler* so that it translates NHA input to its textual binary translation. This means, the 'binary' translation must a text file containing one or more lines of text, each exactly 16 characters long, with each character being either 1 or 0. For instance:

  ```
  0000000000000010
  1110110000010000
  ```
  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 runnable on Java 17. It must keep the classes and methods originally provided, but you can add classes and methods as needed.
- Your program must be able to run the tests, when it is given the "`test`" argument, and translate an NHA source file, when it is given a file name. The starter code implements this already.
- The assembler must only accept source files with a '`.nha`' 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 NHA source file but a '`.bin`' suffix.
  For instance, if the NHA file is called prog.nha then the output file must be called prog.bin. The file written to must be in the same directory/folder as the NHA source file. The sample code does this.
- Translation of each NHA instruction will require one 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 NHA source files, so you do not need to report on errors. However, if you think your program has detected an error in its input and you want to report this to the user, the message must be written 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:

```
sourceFile:lineNumber: message
```

Or, if you want to include the column as well, the format should be:

```
sourceFile:lineNumber:column: message
```

In both cases, the place holders should be replaced with the appropriate values. For instance, the line number should indicate where the error was found. The benefit of this error format is, that most IDEs can recognize it, and allow you to click on the line to navigate to the corresponding location in the source file.

- You are expected to implement the `Main.iUsedAi()` and `Main.aiExplanation()` methods as part of your declaration of use of AI.

The NHA language: lexical and syntactic conventions

A NHA program is stored in a file with a '.nha' suffix. The assembler must translate a single .nha file on each run, or run the tests depending on the given command line parameter. 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 Main.java prog.nha
```

Or, to run tests:

```
java Main.java test
```

Comments: There are no comments in the NHA language.

Whitespace: Blank lines are ignored by the assembler. Each NHA 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.

Mnemonics: All mnemonics (instruction names) are case-insensitive.

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

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

Labels: There are no labels in a NHA program.

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

The NHA language: Overall Design and Examples

The New Hack Assembler resembles the AT&T assembler syntax. Each line starts with the mnemonic for an instruction, which is then followed by 0-3 arguments. To work with the Hack instruction architecture, we support instructions to load values into registers, store values into memory, add, subtract, and jump.
To load values into the A register, we can write the following instructions:

```
ldr A, $42
```

Here, `ldr A, $42` loads the constant 42 into the A register. It thus corresponds to the A-instructions in the Hack assembler. The dollar sign `$` indicates a numeric constant. The mnemonic `ldr` is an abbreviation of load register.

To load a value into the D register, we can write:

```
ldr D, (A)
```

This means, D is the target of the operation, and we use the value in the A register to read from memory, which is indicated by the parentheses. To store are value in a similar way, we use:

```
str (A), D
```

Again, the parentheses indicate that we do not access A, but the memory at the address that is indicated by A. Thus, the value of the D register is stored in memory at address A.

Instruction encoding: format

NHA instructions are encoded as 16-character binary strings targeting the Hack architecture.
Though, we only support a restricted version of Hack to simplify the task at hand.
The following table defines the different variants of instructions and how to encode them.

Description	**Format**	**Example**	**Encoding**
Load constant into the A register	`ldr A, $num`	`ldr A, $42`	As Hack A-instruction: `0000000000101010` One 0, and then 15-bit for the value.
Load value from source register or memory into target register	`ldr target, src`	`ldr D, (A)`	As Hack C-instruction `111accccccddd000` Bits for a, c, and d as per *src* and *target* encoding tables
Store value in memory	`str (A), src`	`str (A), D`	As Hack C-instruction `111accccccc001000` Bits for a and c as per *src* encoding table

Adding values from registers or memory, storing result into target register	`add target, D, A` `add target, D, (A)`	`add D, D, A` `add D, D, (A)`	As Hack C-instruction `1110000010ddd000` with A `1111000010ddd000` with (A) Bits for d as per *target* encoding table
Subtracting values from registers or memory, storing result into target register	`sub target, D, A` `sub target, D, (A)`	`sub D, D, A` `sub D, D, (a)`	As Hack C-instruction `1110010011ddd000` with A `1111010011ddd000` with (A) Bits for d as per *target* encoding table
Conditional jumps based on value from register	`jgt src` same for `jeq, jge, jlt, jne, jle`	`jeq D`	As Hack C-instruction `111accccccc000jjj` Bits for a and c as per *src* encoding, and j as per *jump* encoding table
Unconditional jump	`jmp`	`jmp`	As Hack C-instruction `1110101010000111`

src		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	(A)	1	1	0	0	0	0
a=0	a=1						

target	d1	d2	d3	the value is stored in
D	0	1	0	D register
A	1	0	0	A register

jump	j1	j2	j3	effect:
JGT	0	0	1	if src > 0 jump
JEQ	0	1	0	if src = 0 jump
JGE	0	1	1	if src ≥ 0 jump
JLT	1	0	0	if src < 0 jump
JNE	1	0	1	if src ≠ 0 jump
JLE	1	1	0	if src ≤ 0 jump

Marking of Main Task (96% of marks)

Marks will be based entirely on functionality. You will **not** be evaluated on your programming style. 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 various features of the NHA language. Some examples are part of the test command and allow 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 96%. 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.

Marking of Declaration of AI Use (4% of marks)

As part of the `Main` class and the `test` command, the methods `iUsedAI()` and `aiExplanation()` need to be implemented. Marks will be awarded for the correctness of the Java, i.e., that `iUsedAI()` returns a boolean, and `aiExplanation()` returns a string, not the string's content.

This task is intended to make you reflect on your use or not use of AI, which tools you used, and why, or which alternative resources you relied on instead.

Plagiarism and Duplication of Material

The work you submit must be your own. We will run checks on all submitted work 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 is also available.

Date of this version: 2024-11-18