# Assignment 2 Description

## Assignment 2 - Writing an Assembler

## Weighting and Due Dates

- Marks for this assignment contribute 7.5% of the overall course mark.
- Marks for functionality will be awarded automatically by the web submission system.
- **Due dates: Milestone** - 11:55pm Tuesday of week 9, **Final** - 11:55pm Friday of week 9.
- **Late penalties:**The maximum mark awarded will be reduced by 25% per day / part day late. Marks above the cap will be discarded.
- **Core Body of Knowledge (CBOK) Areas:** abstraction, design, hardware and software, data and information, and programming (see here (Links to an external site.)Links to an external site. for a detailed description of the CBOK).

## Project Description

In this assignment you will complete a variation of project 6 in the nand2tetris course. A detailed description of **Nand2Tetris Project 6** tailored to this course is shown below. The executable programs, **assembler-m** and **assembler-f**, will read a Hack Assembly Language program from standard input and produce the Hack machine code for the Hack Assembly Language program, formatted as sixteen zeros or ones per line, on standard output.

**Note**: for this assignment we have modified the syntax of the C-instruction as noted below so that instructions such as AM=D+1 are now written as D+1=>AM.

#### SVN Repository

You must create a directory in your svn repository named: **<year>/<semester>/cs/assignment2**. This directory must only contain the following files and directories - the web submission system will check this:

- **Makefile** - this file is used by **make** to compile your programs - **do not modify this file**.
- **assembler-m.cpp** - C++ source file
- **assembler-f.cpp** - C++ source file
- **.cpp** C++ source files - names must start with **assembler-m-**, **assembler-f-**, or **shared-**
- **.h** C++ include files - names must start with **assembler-m-**, **assembler-f-**, or **shared-**
- **lib** - this directory contains precompiled programs and components - **do not modify this directory**.
- **includes** - this directory contains **.h** files for precompiled classes - **do not modify this directory**.
- **tests** - this directory contains a test script and test data

**Note**: if the **lib.a** and **.o** files in the **lib** directory do not get added to your svn repository you will need explicitly added them using:

% svn add lib/lib.a lib/*.o

# Submission and Marking Scheme

This assignment has two *assignments* in the web submission system named: ***Assignment 2 - Milestone Submissions*** and ***Assignment 2 - Final Submissions***.

**Assignment 2 - Milestone Submissions: due 11:55pm Tuesday of week 9**

The marks awarded by the web submission system for the milestone submission contribute up to 20% of your marks for assignment 2. Your milestone submission mark, after the application of late penalties, will be posted to the myuni gradebook.

**Your programs must be written in C++** and will be tested using Hack Assembly Language programs that that may or may not be syntactically correct. Although a wide range of tests may be run, including a number of ***secret*** tests, marks will only be recorded for those tests that use the program **assembler-m**. Your assembler will be compiled using the **Makefile** included in the zip file attached below. The **assembler-m** program will be compiled using the file **assembler-m.cpp** and any **.cpp** files with names starting with either **assembler-m-** or **shared-** and linked to a precompiled main program and library functions. The **assembler-f** program will be compiled using the file **assembler-f.cpp** and any **.cpp** files with names starting with either **assembler-f-** or **shared-** and linked to a precompiled main program and library functions.

**Assignment 2 - Final Submissions: due 11:55pm Friday of week 9**

The marks awarded for the final submission contribute up to 80% of your marks for assignment 2.

Your mark for the final submission will be either the marks awarded by the web submission system or, if they exceed either the marks awarded for your logbook or the marks awarded for your coding, your final submission mark will be the geometric mean of these three marks. Your final submission mark, after the application of late penalties, will be posted to the myuni gradebook.

**Important**: the logbook must have entries for all work in this assignment, including your milestone submissions. See "Assessment of Practical Assignments" for the logbook rubric.

**Automatic Marking**

The automatic marking will compile and test both of your assemblers in exactly the same way as for the milestone submission. The difference is that marks will be recorded for **all** of the tests including the ***secret*** tests. **Note:** if your programs fail any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!

**Code Review Marking**

The marks for your coding are based on a manual review of your code that will look at your coding style, your commenting and program structure. We expect that all code that you write will be laid out consistently, it will be easy to read, it will have a clear structure, it will not be a single main function, each function will be prefixed by an appropriate comment and any key logic or data structures will also be appropriately commented.

**Note:** the way in which code is written is extremely important in the real world. One of the major costs of software development is maintenance. That is, taking existing code that needs bugs fixed or modified in order to add new functionality. If code is poorly structured, does not make good use of abstractions, is badly formatted, lacks comments or the comments are confusing, it makes the task of maintaining the code very unpleasant and more error prone than it should.

To reflect the importance of how code is written, the code review assessment may be used to limit your final submission mark, as described above. See "Assessment of Practical Assignments" for the code review rubric.

# Nand2Tetris Project 6: The Assembler

## Background

Low-level machine programs are rarely written by humans. Typically, they are generated by compilers. Yet humans can inspect the translated code and learn important lessons about how to write their high-level programs better, in a way that avoids low-level pitfalls and exploits the underlying hardware better. One of the key players in this translation process is the assembler -- a program designed to translate code written in a symbolic machine language into code written in binary machine language.

This project marks an exciting landmark in our Nand to Tetris odyssey: it deals with building the first rung up the software hierarchy, which will eventually end up in the construction of a compiler for a Java/C++ like high-level language. The relevant reading for this project is Chapter 6. Two useful tools are the supplied Assembler and the supplied CPU Emulator, both available in your tools directory. These tools allow experimenting with a working assembler before setting out to build one yourself. For more information about these tools, refer to the Assembler Tutorial (Links to an external site.)Links to an external site..

## Objective

Write an Assembler program that translates programs written in the **modified** symbolic Hack assembly language into binary code that can execute on the Hack hardware platform built in the previous projects. This will involve using a precompiled tokeniser for the Hack assembly language to implement a parser that recognises labels, A-instructions and C-instructions using tokens returned by the tokeniser. You **must use** the **tokens** class provided in the zip file attached below for all input.

You will build two versions of the assembler, **assembler-m** and **assembler-f**. **assembler-m** is a basic assembler designed to translate assembly programs that contain no symbols. Its main function is precompiled and will call the **assembler_m()**

function that is in the file **assembler-m.cpp**. **assembler-f** is an extension of your basic assembler with symbol handling capabilities and a second pass, yielding the final assembler. Its main function is precompiled and will call the **assembler_f()** function that is in the file **assembler-f.cpp**. The test programs that we supply below are designed to support this staged implementation strategy. The initial version of the **assembler_m()** function simply writes out details of the tokens that are read by the precompiled tokeniser.

If you wish to create additional source files for your assembler programs they must be named as follows. Additional **.cpp** files specific to the program **assembler-m** must have names that start with **assembler-m-**, additional **.cpp** files specific to the program **assembler-f** must have names that start with **assembler-f-** and additional files used by both programs have names that start with **shared-**.

## Contract

There are several ways to describe the desired behavior of your assemblers:

1. When reading a **modified** Hack assembly language program from standard input, your assembler should translate valid assembly language into the correct Hack binary code and write this to standard output.
2. For each **L.asm** file in the **tests** directory, the output of **assembler-m** must match the corresponding **.hack** output file.
3. For each **.asm** file in the **tests** directory, the output of **assembler-f** must match the corresponding **.hack** output file.
4. Your assembler must produce no output if an error is found. Examples of errors include, multiple definitions of a label, an A-instruction with a number that is too large, or a C-instruction with multiple destination components, multiple jump components or no ALU component.

## Compiling and Running Your Programs

You programs can be compiled with one of the following three commands. To compile both:

% make

To compile just **assembler-m**:

% make assembler-m

To compile just **assembler-f**:

% make assembler-f

You can run either or both of your programs against all tests provided in the tests sub-directory using one of the following commands. **Note**: the first command also runs the tests using working versions of the programs so you know that the test scripts are working.

% make test

Testing working assembler-m against hack files
Checking "./lib/w-assembler-m < tests/AddL.asm | diff - tests/AddL.hack" - test passed
...

To test the **assembler-m** program use the following command:

% make test-assembler-m

Testing student assembler-m against hack files

Checking "./assembler-m < tests/AddL.asm | diff - tests/AddL.hack" - test failed

...

To test the **assembler-f** program use the following command:

% make test-assembler-f

Testing student assembler-f against hack files

Checking "./assembler-f < tests/AddL.asm | diff - tests/AddL.hack" - test failed

...

The test scripts do not show the program outputs, just passed or failed, but they do show you the commands being used to run each test. You can cut-paste these commands if you want to run a particular test yourself and see all of the output.

**Note:** Do not modify the **Makefile** or the sub-directories **includes** and **lib**. They will be replaced during testing by the web submission system.

## The Tokeniser

You **must** use the precompiled tokeniser functions provided by the library in the zip file attached below. The tokeniser functions are described in the file **tokeniser.h**. This tokeniser returns the followings tokens:

| **Token Class** | | **Definition** | **Example Token** | **Token Value** |
|---|---|---|---|---|
| "address" | ::= | '@' **(** name **|** number **)** | **@hello** | "hello" |
| "label" | ::= | '(' name ')' | **(END)** | "END" |
| "dest" | ::= | 'MD' **|** 'AM' **|** 'AD' **|** 'AMD' | **Am** | "AM" |
| "dest-comp?" | ::= | 'A' **|** 'M' **|** 'D' | **a** | "A" |
| "comp" | ::= | '0' **|** '1' **|** '-1' **|** '!D' **|** '!A' **|** '-D' **|** '-A' **|** 'D+1' **|** 'A+1' **|**<br>'D-1' **|** 'A-1' **|** 'D+A' **|** 'D-A' **|** 'A-D' **|**<br>'D&A' **|** 'D|A' **|** '!M' **|**<br>'-M' **|** 'M+1' **|** 'M-1' **|** 'D+M' **|** 'D-M' **|**<br>'M-D' **|** 'D&M' **|** 'D|M' | **D-A** | "D-A" |
| "jump" | ::= | 'JMP' **|** 'JLT' **|** 'JLE' **|** 'JGT' **|** 'JGE' **|**<br>'JEQ' **|** 'JNE' | **JGT** | "JGT" |
| "arrow" | ::= | '=>' | **=>** | "=>" |
| "semi" | ::= | ';' | **;** | ";" |
| "null" | ::= | 'NULL' | **nuLl** | "NULL" |

| Additional Rules | | Definition | Example Text |
|---|---|---|---|
| name | ::= | letter **(** letter **\|** digit **)*** | "_he:82m.Uch$" |
| number | ::= | digit digit* | "99" |
| letter | ::= | 'a'-'z' **\|** 'A'-'Z' **\|** '$' **\|** '_' **\|** ':' **\|** '.' | "$" |
| digit | ::= | '0'-'9' | "1" |

**Notes:**

- if an error occurs or the end of input is reached, the token class "?" is returned
- it is an error to find a character that cannot be part of token or is not a space " ", tab "\t", carriage return "\r" or newline "\n"
- single line comments that start with "//" and finish at the next newline character "\n"
- when searching for the start of the next token all spaces, tabs, carriage returns, newlines and comments are ignored
- newlines are not significant to the tokeniser so more than one instruction can be on the same line or an instruction could be split over multiple lines
- name, number, letter and digit are never returned as token classes
- letters in a name are case sensitive, all other occurrences of letters are converted to uppercase
- in a definition the or operator **|** separates alternative components
- in a definition the round brackets **( )** which are not inside single quotes are for grouping components of token
- in a definition the star character ***** indicates that the preceding component of a token may appear 0 or more times

## The Assembler Parser

The following table describes the structure of an assembly language file that must be translated into machine code. **Note:** there are no references to lines in these rules. All whitespace, including newline characters, and comments will have been removed by the precompiled tokeniser. Therefore these rules are defined purely in terms of the token classes in the previous table.

**VERY IMPORTANT**: These rules contain a change to the syntax of a C-instruction used in the text book. In Hack assembly language the destination component of a C-instruction comes before the alu operator but, in this alternate syntax, the destination component comes second. The connecting '=' has also been replaced by an arrow '=>'. For example:

D=A+1

is written as

A+1 => D

| Term | | Definition |
|---|---|---|

| | | |
|---|---|---|
| program | ::= | **(**label **|** address **|** C-instr**)* eof** |
| C-instr | ::= | aluop **[**destination**] [**ajump**]** |
| destination | ::= | arrow **(**null **|** dest **|** dest-comp?**)** |
| aluop | ::= | dest-comp? **|** comp |
| ajump | ::= | semi **(**null **|** jump**)** |

**Notes:**

- in a definition **eof** indicates the end of the input
- in a definition the or operator **|** separates alternative components
- in a definition the round brackets **( )** are for grouping components
- in a definition the square brackets **[ ]** which indicates that the enclosed components may appear 0 or once
- in a definition the star character ***** indicates that the preceding component may appear 0 or more times

## Lookup Tables

Your assemblers will need a way of generating the correct set of bits for each part of a C-instruction. For example, "JMP" needs to be mapped to "111". To do this you will need some sort of lookup table to record these mappings. The **assembler-f** program also needs a lookup table so that it can implement a symbol table to record label addresses and the memory locations for variables. You should use the precompiled symbol table classes described in the **symbols.h** file to create any lookup tables that you require. The **symbols.h** file includes examples of how to use these lookup tables.

You will need several lookup tables, one for the symbol table and one for each component of a C-instruction. Although most parts of a C-instruction are unique, the M, A and D registers need to be mapped to different sets of bits depending on whether they are used as an alu operation or a destination. For example, "A" as a destination would be "100" but as an alu operation it would be "0110000".

## Test Programs

In addition to the tests in **tests** directory, we will use a number of ***secret*** tests that may contain illegal tokens and / or multiple labels and instructions on the same line and / or syntactic errors. **Note:** these tests are ***secret***, if your programs fail any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!

**The Pong program** supplied in the **tests** directory was written in the Java-like high-level Jack language and translated into the Hack assembly language by the Jack compiler and a Hack Virtual Machine translator. (The Virtual Machine, Jack and the Jack compiler are described in Chapters 7-8, 9 and 10-11, respectively). Although the

original Jack program is only about 300 lines of Jack code, the executable Pong code is naturally much longer. Running this interactive program in the supplied CPU Emulator is a slow affair, so don't expect a high-powered Pong game. This slowness is actually a virtue, since it enables your eye to track the graphical behavior of the program. And don't worry! as we continue to build the software platform in the next few projects, Pong and and other games will run much faster.

- assignment2.zip
-