# Lab 3 – Linker/Loader

# Modified Assembler User's Guide

CSE 3903

Spring 2023


Group: Worst Name Ever

Sade Ahmed

Jeremy Bogen

Mani Kamali

Giridhar Srikanth


Date of Submission: 04/19/2023

# Table of Contents

# Installation Guide

## System Requirements

The user's machine must be run on Java 8 or above to support the assembler. As per the Oracle documentation, the following operating systems will support those versions of Java:

RAM: 128MB
Disk space: 124 MB for JRE; 2 MB for Java Update
Processor: Minimum Pentium 2 266 MHz processor
Browsers: Internet Explorer 9 and above, Microsoft Edge, Firefox, Chrome

### Windows

Windows 11 (64 bit only) 8u311 and above
Windows 10 (8u51 and above)
Windows 8 (Modern UI is not supported)
Windows 8 SP1* (No longer supported by MS)
Windows Vista SP2* (No longer supported by MS)
Windows Server 2022
Windows Server 2019
Windows Server 2016
Windows Server 2012 R2
Windows Server 2012
Windows Server 2008 R2 SP

### MacOS

macOS 12 (8u311 and above)
maxOS 11 (8u281 and above)
OS X 10.9 and above
OS X 10.8.3 and above
Administrator privileges for installation
64-bit browser (A 64-bit browser (Safari, for example) is required to run Oracle Java on macOS)

### Linux

Oracle Linux 8 (8u221 and above)
Oracle Linux 7 (64-bit) (8u20 and above)
Oracle Linux 6.(32-bit and 64-bit)
Oracle Linux 5.5+
Red Hat Enterprise Linux 8 (8u221 and above)
Red Hat Enterprise Linux 7 (64-bit) (8u20 and above)

Red Hat Enterprise Linux 6 (32-bit and 64-bit)
Red Hat Enterprise Linux 5.5+
Suse Linux Enterprise Server 15 (8u201 and above)
Suse Linux Enterprise Server 12 (64-bit)[(2)] (8u31 and above)
Suse Linux Enterprise Server 11 (32-bit and 64-bit)
Suse Linux Enterprise Server 10 SP2+ (32-bit and 64-bit)
Ubuntu Linux 21.04 (8u291 and above)
Ubuntu Linux 20.10 (8u271 and above)
Ubuntu Linux 20.04 LTS (8u261 and above)
Ubuntu Linux 19.10 (8u241 and above)
Ubuntu Linux 19.04 (8u231 and above)
Ubuntu Linux 18.10 (8u191 and above)
Ubuntu Linux 18.04 LTS (8u171 and above)
Ubuntu Linux 17.10 (8u151 and above)
Ubuntu Linux 17.04 (8u131 and above)
Ubuntu Linux 16.10 (8u131 and above)
Ubuntu Linux 16.04 LTS (8u102 and above)
Ubuntu Linux 15.10 (8u65 and above)
Ubuntu Linux 15.04 (8u45 and above)
Ubuntu Linux 14.10 (8u25 and above)
Ubuntu Linux 14.04 LTS (8u25 and above)
Ubuntu Linux 13
Ubuntu Linux 12.04 LTS

# Building and Running

The WNE Assembler uses Gradle to build and run. There are instructions here to both build with the command line and Eclipse. But the command line is the main supported environment for users.

## Getting the Code

An archive file of the most recent release of the assembler can be downloaded on the Github repository's releases page. Extracting this archive results in the folder that is used for the following sections on building and running the program.

## Running and Building Using the Command Line

Change directories so that the program's root folder is your current working directory. You may want to double-check your java version by running "`java -version`", our program only runs with java 17 or greater.

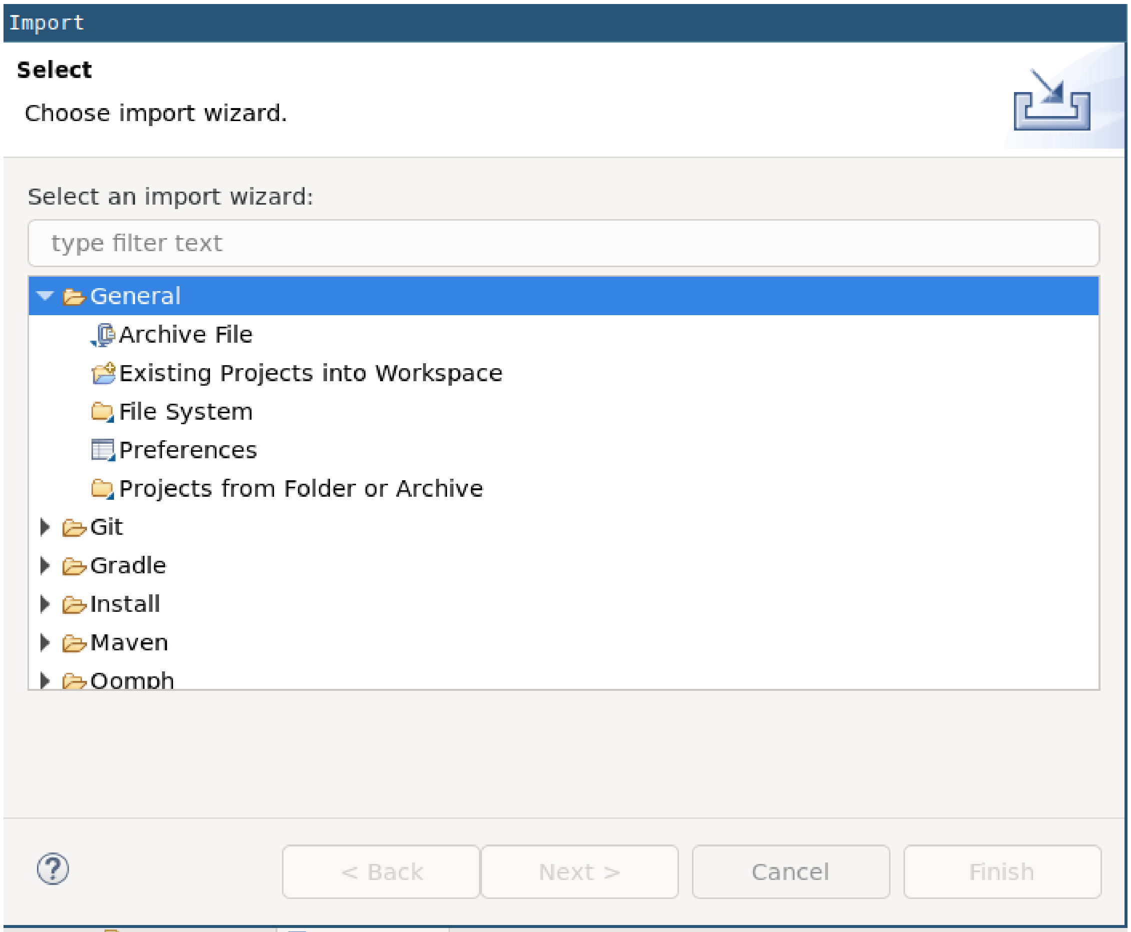
If your working directory is the program's root directory and you have an up-to-date version of Java, running the program is very simple. To run the program simply run "`./gradlew run --args="lab2 ../input.txt ../out.obj ../list.listing`". Note: the program runs from the `app/` directory, which is why the input and output files use `../`.

The project can also be run with a simple "`./gradlew build`".

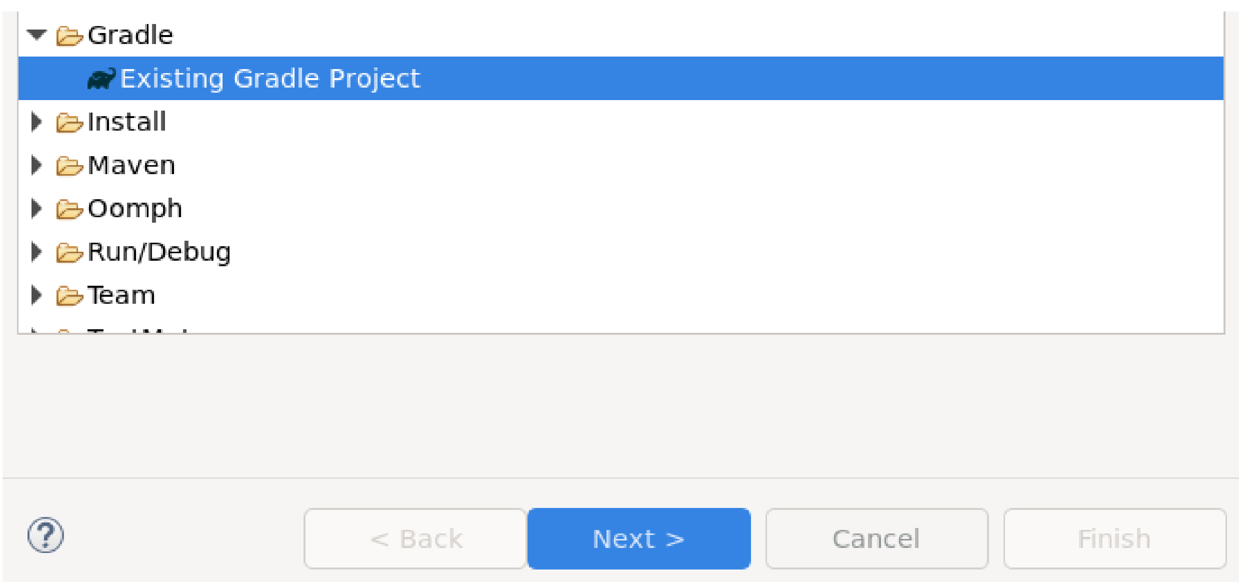
## Building with Eclipse

First, in Eclipse, go to File -> Import…

That should bring you to this dialogue:



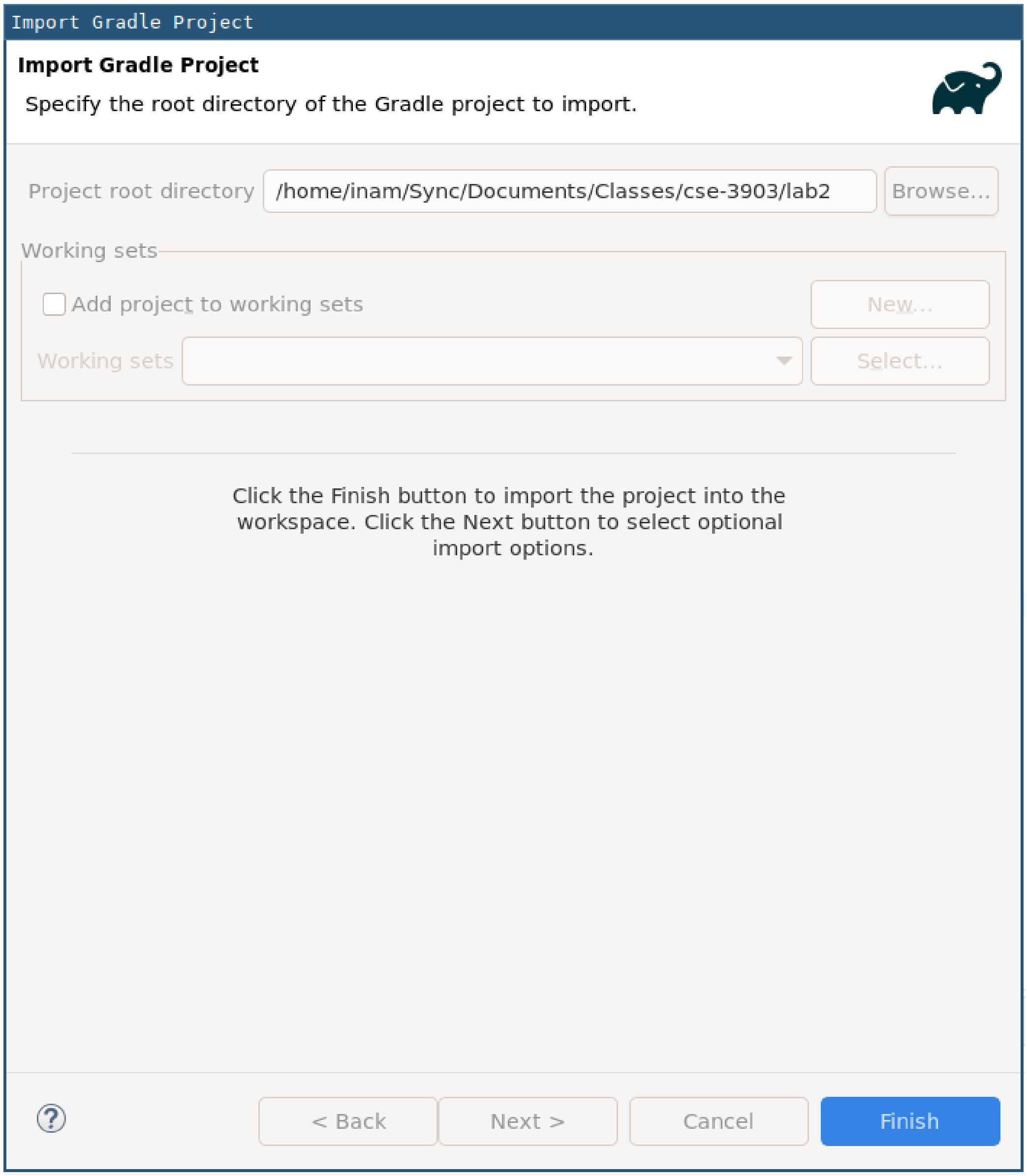

Then, open the Gradle folder and select Existing Gradle Project, should look like so:



Then, hit next.

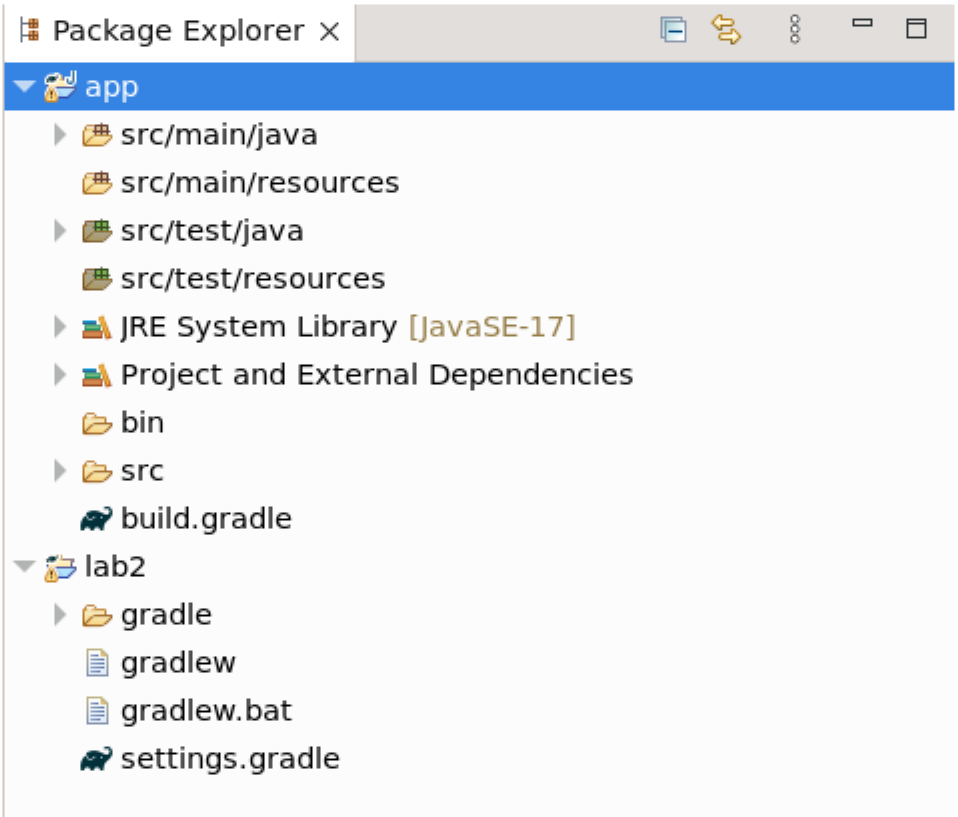
This will bring you to the Gradle Import welcome page. Feel free to read through the text here, or don't, and press next again.

Here, browse to the root directory of the repository. Here's what that looks like for me:

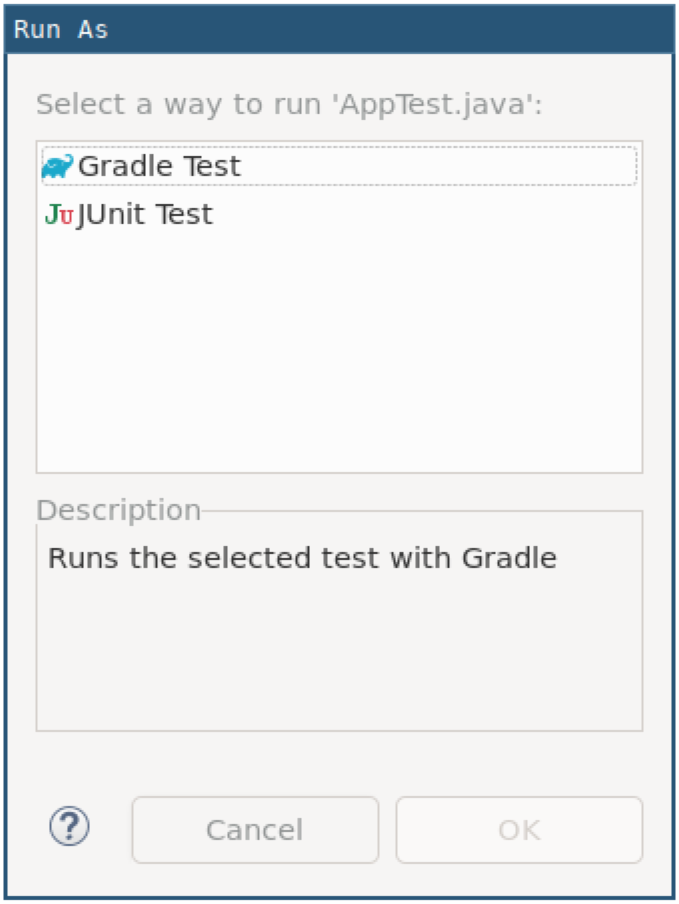



And then, just hit Finish.

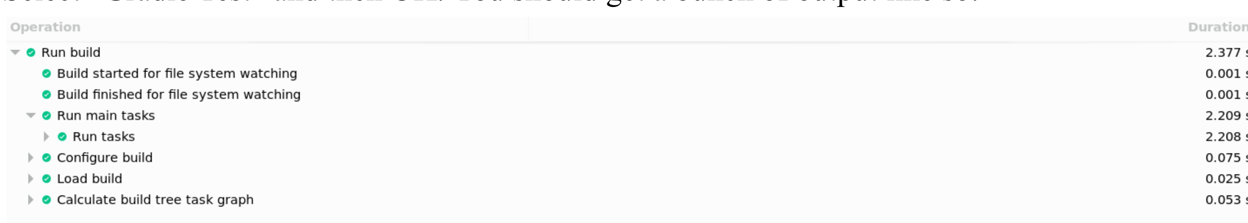
Somewhat confusingly, this will produce TWO directories in the project explorer:




The app directory is the one we actually will be working in (except for documentation and stuff, possibly). To run the program, just select the app directory and hit the green Run App button. To run a test, select the test file, in this case, AppTest.java, and hit the same button. That will bring up this menu:

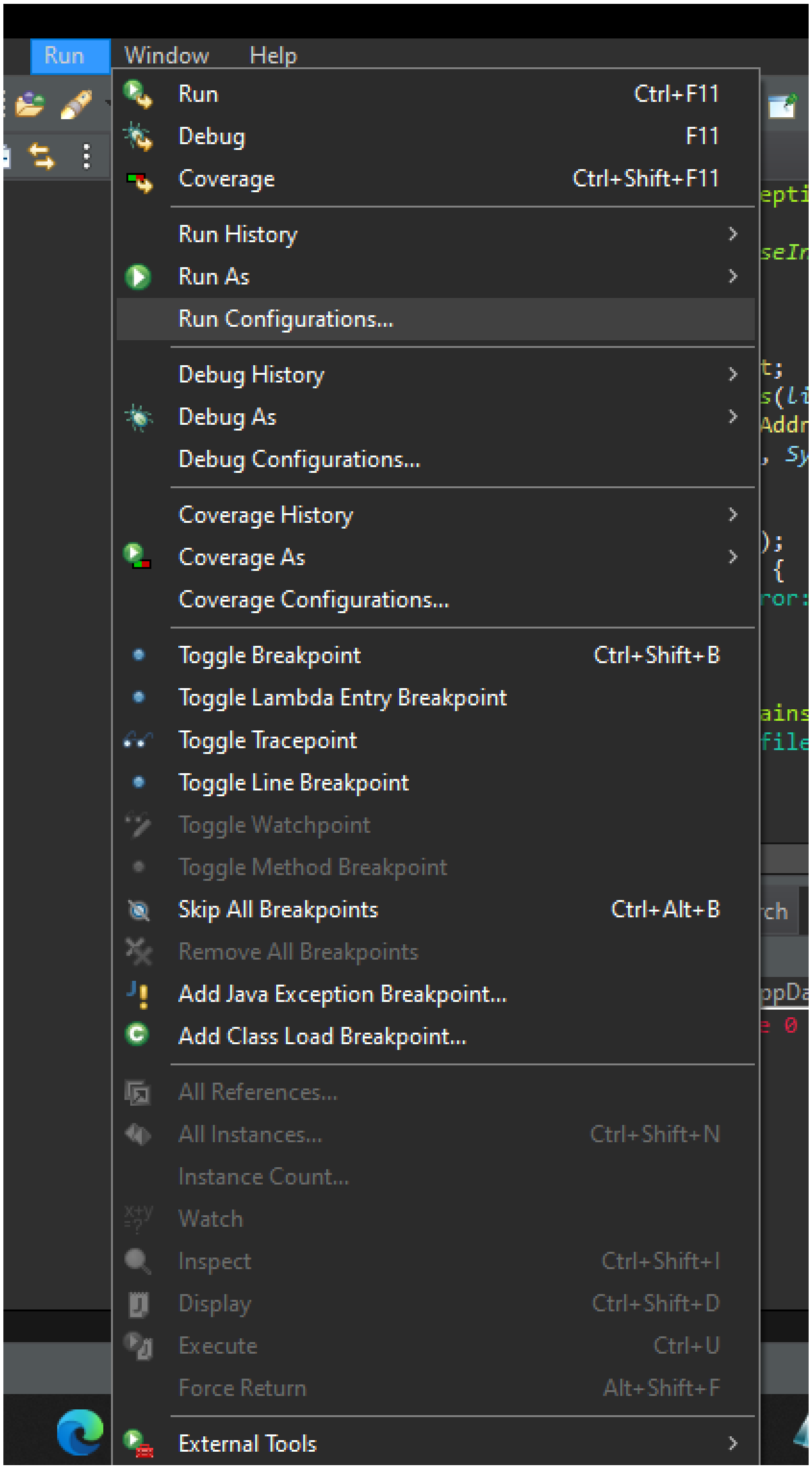



Select “Gradle Test” and then OK. You should get a bunch of output like so:

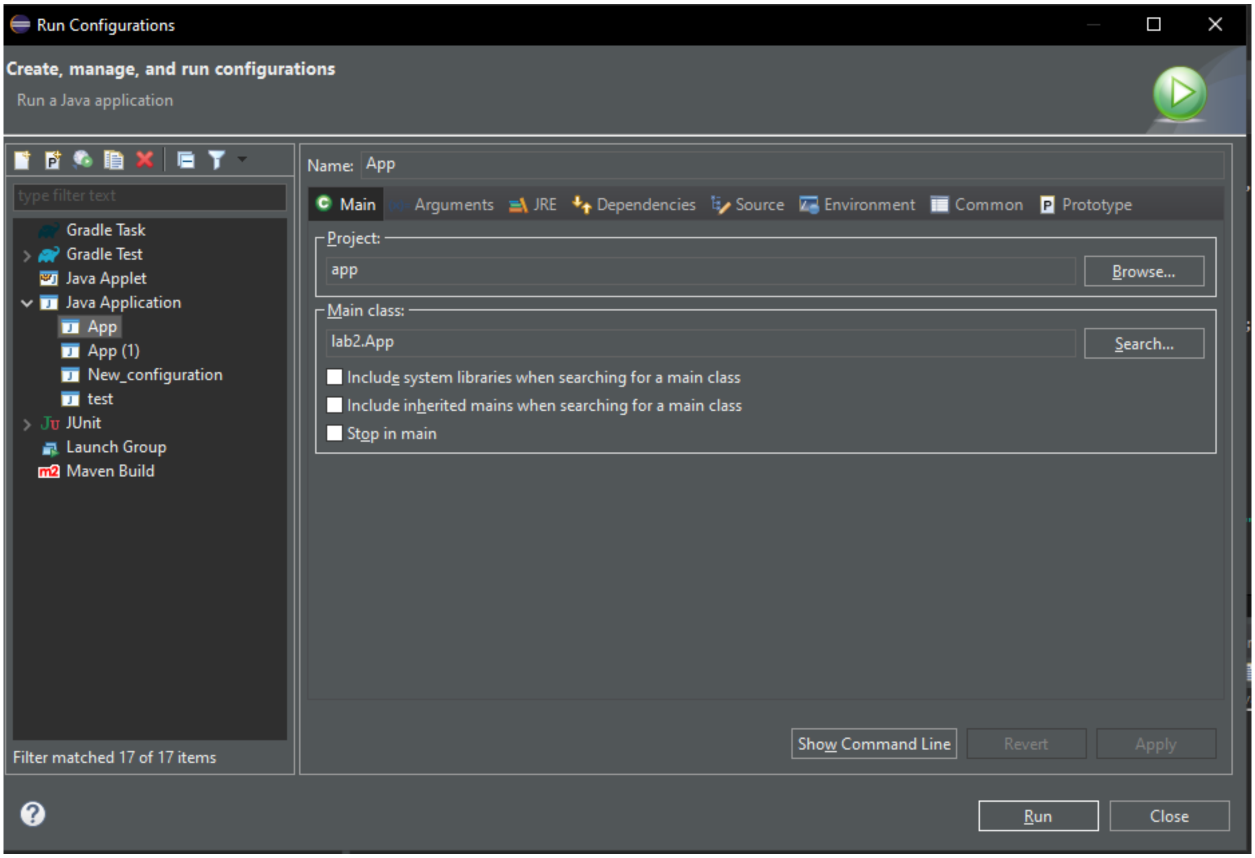



Alternatively you can run the program through eclipse

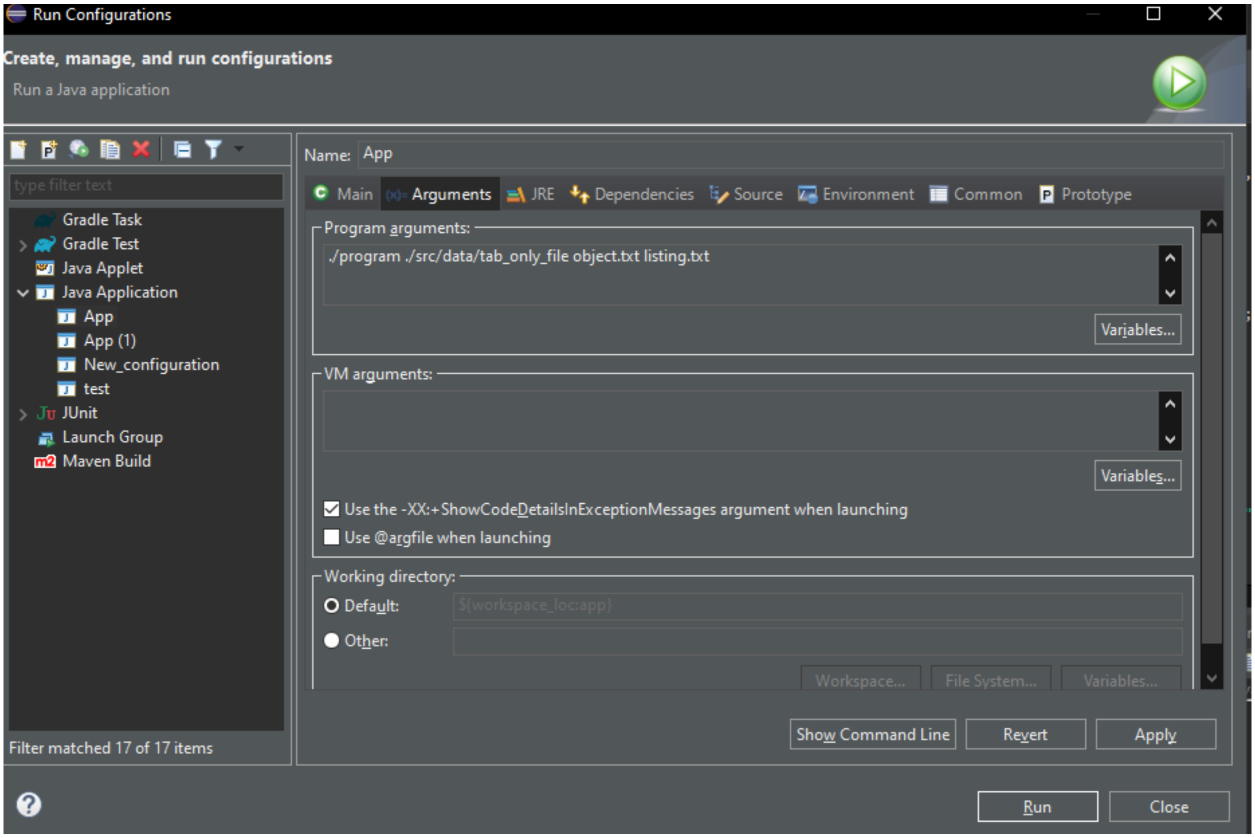
Select Run, Run Configurations




Click, Java Application then App


Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
Gradle Task
Gradle Test
Java Applet
Java Application
App
App (1)
New_configuration
test
JUnit
Launch Group
Maven Build
Filter matched 17 of 17 items
Name: App
Main
Arguments
JRE
Dependencies
Source
Environment
Common
Prototype
Project:
app
Browse...
Main class:
lab2.App
Search...
Include system libraries when searching for a main class
Include inherited mains when searching for a main class
Stop in main
Show Command Line
Revert
Apply
Run
Close

Select the Arguments tab and in the Program arguments pass the your arguments in the following order ./program <path> <object output file> <listing output file>




# Program Usage

## Overview of the Assembly language

### Contents of the assembly file

The contents of the assembly file include labels, operational fields, operands, and comments. Labels are alphanumeric characters that are generally used to jump, store, or load into different parts of the program. The operational field (opcodes for short) is the instructions that change the state of the machine. The operands specify which parts of the machine should change. The comments allow explaining what the code does, without affecting the program. The character ‘;’ signifies that everything written after it is a comment till the next line.

## Assembler input file

The assembly input file must follow the following format: positions one to six should contain any labels, positions seven to nine should be spaces[1], positions ten to fourteen should be the operational field (opcode), positions fifteen to seventeen should be spaces[1], positions eighteen to the end of the file should contain the operands with any comments. The input file format is summarized in Table 1 given below:

**Table 1:** Summary for the required format per line of assembly code.

| Position | Meaning |
|---|---|
| 1-6 | Label |
| 7-9 | Spaces (i.e, ‘ ’) |
| 10-14 | Operational field (opcode) |
| 15-17 | Spaces (i.e, ‘ ’) |
| 18-End | Operand and comments |

[1] Note: If the user fails to meet the above format, an error is thrown to the user as the program is trying to execute. Any missing fields, whitespaces that are not spaces (e.g, a tab character), or unintended characters in the required fields will throw an error and terminate execution.

## Labels

As mentioned above, labels are alphanumeric characters that can be six characters long to stay true to the required format of the assembly language. Labels cannot start with the characters ‘R’, ‘x’, or ‘#’ as these serve other purposes in the assembly language. These restrictions are case-sensitive, meaning that a label is allowed to start with ‘r’ and ‘X’. Labels are also case-sensitive (i.e, the label “Labs” is different from the label “labs” as one has an upper-case letter, while has only lower-case letters).

## Instructions

There are different instructions based on the machine state to be changed to. These instructions can be categorized into data processing instructions, data movement instructions, and control flow instructions[1].

*Data flow instructions*: the three instructions used for processing data are ADD, AND, and NOT. The ADD and AND instructions accept either two source registers or a source register with a five-bit immediate. The NOT instruction accepts a source register. The machine performs the necessary operation and loads the result into the destination register. The ADD instruction adds the value stored in the source register with either the other source or the immediate value, which is sign extended to get a sixteen-bit quantity, and the result is loaded into the destination register. The AND instruction performs bitwise-AND on the same parameters as the ADD instructions. The NOT instruction performs bitwise-complement on the source register and loads it into the destination register.

*Data movement instructions*: the seven instructions used for the movement of data are LD, LDI, LDR, LEA, ST, STI, and STR. The LD, LDI, LEA, ST, and STI instructions accept a destination register if it is a load instruction or source instruction if it is a store instruction, and an address (which is converted to a nine-bit page offset). For each of these instructions, an address is formed by taking the upper seven bits of the address of the next instruction to execute and concatenate the bits from the page offset to perform the necessary operations. For the LEA instruction, the formed address is directly stored in the destination register. The instructions LD and ST go to memory formed from the address and either take the value from memory and store it in the register or store the value from the register and place it in memory. The LDI and STI instructions follow a similar procedure to LD and ST, but they go to the memory address specified in the current memory location traversed to by the formed address and perform the necessary loading or storing. The LDR and SDR instructions also accept a source or destination register, but additionally accept another register (known as the base register) and a six-bit index. These instructions perform the same operation as LD and ST, but the address is formed by adding the value stored in the base register and the six-bit index, interpreted as a positive quantity.

*Flow of control instructions*: the five instructions used for changing the flow of instructions (address) are BRx, JSR/JMP, JSRR/JMPR, RET, and TRAP. The BRx and JSR/JMP instruction accepts an address (which is converted to a nine-bit offset), the JSRR/JMPR instruction accepts a base register and a six-bit index value, RET does not accept any parameters, and TRAP accepts an eight-bit trap vector. The BRx instruction is a conditional branch that changed the next instruction to execute to the address specified by the operand. The address is formed the same way as the page offset instructions. There are several variations[2] to the BRx instructions, that determine when to take the branch. The eight variations are BR (no-op), BRN (negative branch), BRZ (zero branch), BRP (positive branch), BRNZ, BRNP, BRZP, and BRNZP. The JSR/JMP instruction is for jumping to the address specified by the operand. Similar to address formation as the BRx instruction, a major difference is the type of jump being run. JSR says that the address for the next instruction is to be saved (before the jump) to register seven, whereas JMP is an unconditional jump. Similarly, with JSRR/JMPR, the address is formed to the LDR instruction, but JSRR would save the address of the next instruction (before the jump) to register seven, whereas JMPR is an unconditional jump. The RET instruction sets the address of the next instruction to point to as the value stored in register seven. The TRAP instructions execute a certain set of instructions based on the eight-bit trap vectors from the operand. The list of instructions TRAP could execute is given below in Table 2.

**Table 2:** The TRAP instruction table for executing "system" calls.

| Hex value | Name | Description |
|---|---|---|
| 0x21 | OUT | Write the character (lower eight bits) of register zero to the console |
| 0x22 | PUTS | Write the null-terminated string pointed to by register zero to the console |
| 0x23 | IN | Print a prompt on the screen and read a single character from the keyboard. The character is copied to the screen and its ASCII code is copied to register zero. The upper eight bits of register zero are cleared |
| 0x25 | HALT | Halt execution and print a message to the console |
| 0x31 | INN | Print a prompt on the screen and read a decimal number from the keyboard. The number is echoed to the screen and stored in register zero. The given number is a value between in the range [-0x7FFF, 7FFF]. |
| 0x43 | RND | Store a random number in register zero. |

The accepted instruction format is summarized (with examples) in Figure 3 below. A summary of the instruction interpretation in binary can be seen in Figure 4 below.

| Instruction | Example |
|---|---|
| ADD DR,SR1,SR2 | `ADD R0,R3,R0` |
| ADD DR,SR1,imm5 | `ADD R3,R3,#-1` |
| AND DR,SR1,SR2 | `AND R5,R5,R3` |
| AND DR,SR1,imm5 | `AND R3,R3,xF` |
| BRx addr | `BRZP x3020` |
| DBUG | `DBUG` |
| JSR addr | `JSR Mult` |
| JMP addr | `JMP ShutDn` |
| JSRR BR,index6 | `JSRR R2,x0` |
| JMPR BR,index6 | `JMPR R4,x10` |
| LD DR,addr | `LD Acc,Value` |
| LDI DR,addr | `LDI R0,x3100` |
| LDR DR,BR,index6 | `LDR R0,R4,xA` |
| LEA DR,addr | `LEA R0,Msg1` |
| NOT DR,SR | `NOT R2,R2` |
| RET | `RET` |
| ST SR,addr | `ST R5,ANSWR` |
| STI SR,addr | `STI R3,x3000` |
| STR SR,BR,index6 | `STR R2,R0,Offset` |
| TRAP trapvect8 | `TRAP x25` |

**Figure 3:** Summary of the assembly instruction format with examples.

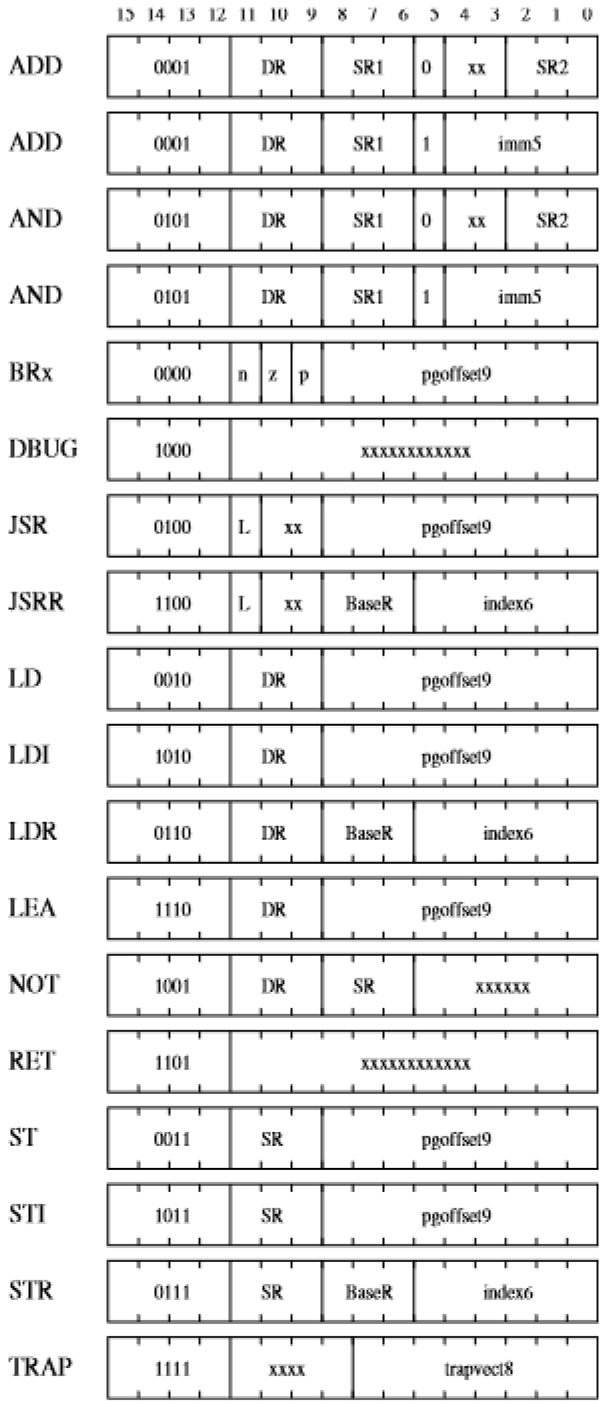



**Figure 4:** Summary of the instructions in binary format.

[1] Note: All the instructions are also case-sensitive and must start with upper case letters. Otherwise, an error is thrown to the user and the program terminates.
[2] Note: The program requires this format for the BRx instruction. Any other combinations will throw an error and terminate the program.

## Operands

The operands portion of the assembly are the commands to determine the parts of the machine that would change states. Anything that involves direct access to the eight registers can be invoked with the character 'R' followed by the register number. For example, to access register 1, it is written as "R1" in assembly. For different parts of the parameters of a given operational field, there cannot be any whitespaces in-between, and must be separated by commas. For example, the instruction ADD instruction with a destination register 1 and two source registers, register three and four, would look like "ADD  R1,R3,R4".

Constants can be directly written for the assembler code as well, where a number followed by an 'x' denotes a hex number, and a '#' denotes a decimal number. For example, the hex constant "0x20" would be written as "x20" in assembly and the decimal constant 34 would be written as "#34". All the hex constants must be written as a positive value between the range of [0x0, 0xFFFF] and decimal constants can be written either in the range of [-32768, 32767][1] or [0, 65535][1].

There are different situations where a constant must follow a certain range or can be replaced with a symbol. Symbols are assembler labels that get a value (such as an address or pre-defined value during assembly) when interpreting the assembly language. Symbols can be interpreted as either absolute symbols or relative symbols. Absolute symbols are assembler labels that are given a value (explicit definition). Relative symbols are assembler labels that are assigned a value implicitly (given an address value based on the instruction pointer). The different situations can be classified as register, immediate, index, trap vector, literal, and address.

*Register*: As mentioned above, there are eight registers in the abstract machine, and can be accessed with the character 'R' before the register number to access. Absolute symbols can also act as a substitute for direct register access. If the symbol is between the range [0, 7], then that value is used in place of the register[2].

*Immediate*: Immediate values are the imm5 values for instructions ADD and AND. When using a constant value, decimal values have to be within the range of [-16, 15] and hex values have to be in the range [0x0, 0xFF]. Absolute symbols can substitute for the constant values[2] if within the proper range specified for the immediate[3].

*Index*: Index values are the ind6 values for instructions that perform index addressing. When using a constant value, decimal values have to be within the range of [0, 63] and hex values have to be in the range [0x0, 0x3F]. Absolute symbols can substitute for the constant values[2] if within the proper range specified for the index.

*Trap vector*: Trap vector values are the trapvect8 values of the TRAP instruction. When using a constant value, decimal values have to be within the range of [0, 255] and hex values have to be in the range [0x0, 0xFF]. Absolute symbols can substitute for the constant values[2] if within the proper range specified for the trap vector.

*Literal*: Literals are a special constant-symbol value that can only be used for the LD instruction[4]. A literal starts with the equals characters (i.e, '=') followed by a decimal or hex constant. The range for which the constant value can be from [-32768, 32767] for decimal or [0, 65535] or hex. When the user uses a literal, the value for the literal is placed at the last address defined by the user. When parsing the assembly language, the address at which the literal is stored is used when creating the page offset[5].

*Address*: Address values are used as values that require a page offset. When using a constant value, decimal values have to be within the range of [0, 65535] and hex values have to be in the range [0x0, 0xFFFF]. Absolute symbols can substitute for the constant values[2] if within the proper range specified for an address. Relative symbols are allowed only for instructions that take an address as a parameter. Even though a complete address is given, only the lower nine bits are used to fit the binary format of the given instruction (the page offset[5]). Any instruction with relative symbols in the address parameters and the program is relocatable, that line of assembly becomes relocatable.

[1] Note: The decimal constant range applies for different situations, such as no signed-bit for address operations or signed-bit for literals.
[2] Note: The absolute symbols need to be within the specified ranges. Otherwise, they will be invalid values and the program will throw an error and terminate.
[3] Note: When an absolute symbol is used as the last operand for an ADD or AND instruction, it is treated as an immediate (imm5) value rather than as a register.
[4] Note: Any other instruction that uses a literal will be flagged as an error and the program will terminate the program.
[5] Note: It is important to note that the page offsets only work because the address formed is by taking the upper seven bits of the address of the next instruction pointer by the instruction pointer and concatenating to the page offset. If the given address is not in the same page range (the machine is represented as 128 pages, which has 512 words per page), then the assembler flags this as an error and terminates further execution.

## Comments

Comments are an optional field that is either the start of an assembly line or in the later part of the operand. When the assembly line begins with a comment, it is assumed the entire line is a comment and is ignored. Otherwise, a comment must be appended to the end of the operand after all the necessary operations are given.

## Pseudo-Ops

In the assembly language of the abstract machine, other opcodes can be utilized with the given instruction set. These are called Pseudo Operations (Pseudo-Ops for short). The six Pseudo-Ops of the abstract machine are .ORIG, .END, .EQU, .FILL, .STRZ, and .BLKW[1].

*.ORIG*: This is the first non-comment Pseudo-Op at the start of the assembly program. The user could input an address for the operand field to denote where to start the program. The provided address must be a hex value in the range [0x0, 0xFFFF]. If the user defines an address in the operand field, the address is considered an absolute address to load the program, making the program an absolute program. If the user fails does not fill the operand field, the program defaults to start addressing at memory 0x0, making the program a relocatable program. The .ORIG pseudo-op needs to be accompanied by a label in the label field, which would become the segment name of the program[2].

*.END*: This Pseudo-Op indicates when the end of a program has been reached. The user could provide an optional address, between the ranges [0, 65535] for decimal and [0x0, 0xFFFF] for hex,  in the operand field to signify where to start the execution of the program or with a symbol. If the user does not provide a value in the operand field, then execution of the program begins at the first address in the segment. There should be no labels present for this opcode.

*.EQU*: This Pseudo-Op allows for creating a constant within a program for a label. Either decimal constants from [-32768, 32767] or hex constants from [0x0, 0xFFF] are allowed values to define a label. Pre-defined symbols are allowed in the operand field of this Pseudo-Op.

*.FILL*: This Pseudo-Op fills a value specified in the assembly to the current location of memory instructions being loaded. The range for which the constant value can be from [-32768, 32767] for decimal or [0x0, 0xFFF] or hex. Absolute or relative symbols within the given range are allowed at the operand position. If a relative symbol is loaded to the specified location and the program is relocatable, the entire line of assembly is relocatable.

*.STRZ*: This Pseudo-Op takes a string as its operand and places the null-terminated string in memory. Each location of memory contains a character (lower eight bits of the string) from the given string. The last element of the string would contain the value of null (0x0000). The ASCII value for each character is stored in memory, with the upper eight bits (from a total of sixteen bits) cleared.

*.BLKW*: Reserves a block of memory based on the value specified n the operands. The range for which the constant value can be from [1, 65535] for decimal or [0x1,0xFFFF] or hex. The operand field can be an absolute symbol if within the specified range.

[1] Note: All the instructions are also case-sensitive and must start with upper case letters. Otherwise, an error is thrown to the user and the program terminates.
[2] Note: Only the .ORIG label is considered as the segment name, the other labels would be used as symbols in the assembler.

## Size of opcodes

Each opcode has a definite/variable length that determines the amount of memory required to reserve. Any machine instructions and the .FILL instructions occupy only one block of memory. The .ORIG, .END, and .EQU instructions take no blocks of memory. The .STRZ and .BLKW instructions reserve memory based on the value defined for them (i.e, the string length plus null termination or the hex/decimal constant or absolute symbol). Table 5 below summarizes the block of memory required for each operation.

**Table 5:** Size of each opcode for reserving memory.

| Instruction | Memory allotment |
|---|---|
| .ORIG, .END, .EQU | 0 blocks of memory |
| All machine instructions, .FILL | 1 block of memory |
| .STRZ | Size of the string + null termination block(s) of memory |
| .BLKW | Decimal/Hex constant/Absolute symbol value block(s) of memory |

## Interpreting a sample input

A simple assembly program with the proper format and required instructions can be seen in Figure 6 below.

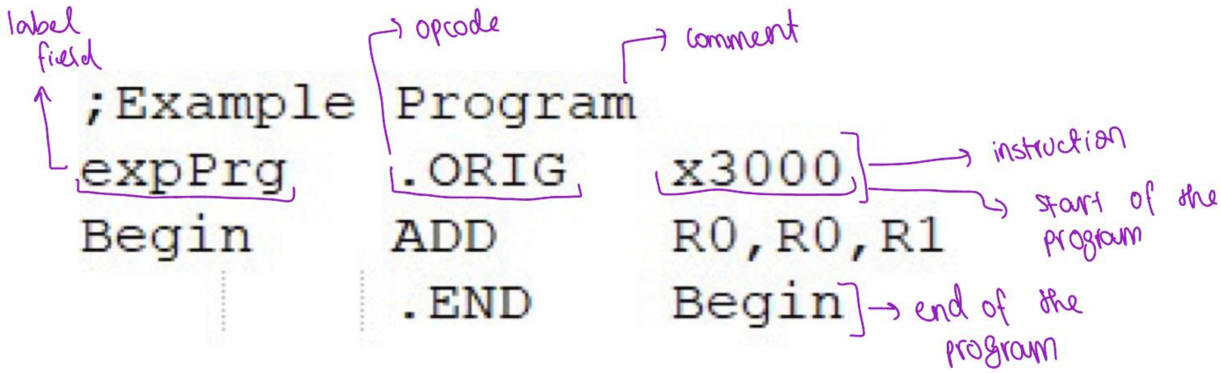



**Figure 6:** Interpreting ADD instruction in object file format.

As seen in Figure 6 above, the assembly line contains the label field, the opcode field, and the operand field (optionally with comments). The whitespaces present in between each field (the opcode and operand field) are required to be considered a valid line of assembly code. As mentioned previously, a label (up to six characters) is required for the **.**ORIG opcode and omitting a label for the **.**END opcode. For the operand fields, all the instructions are written contiguously, separated by a comma without other characters.

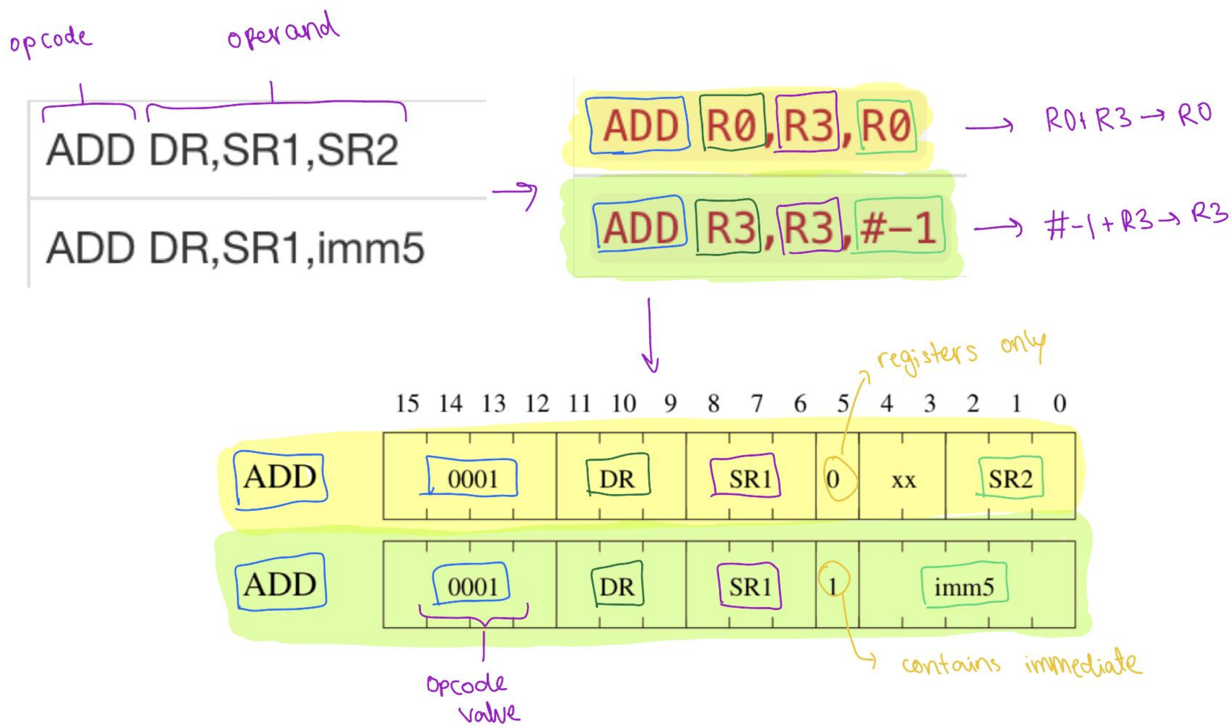



**Figure 7:** Interpreting ADD instruction in object file format.

Figure 7 gives an overview of how an input file is interpreted. For the instruction given above in Figure 7, the first ADD instruction's output would be "0x10C0" based on the operands and binary representation. Additional information on the output file is given in the "Understanding the Output" section.

## Understanding the Output

The two files produced by the assembler program are the main output file and the listing file.

### Main output file

The main output file is the file that will become the input for the lab 1 simulator. This includes having a singular header record, any number of text records, and an end record.

The Header Record is split into 4 sections, those being:
Record position 1: H (to indicate Header)
Record positions 2-7: a 6-character segment name (Must be exactly 6 characters)
Record positions 8-11: a set of hex characters denoting the IPLA (initial program load address)
Record positions 12-15: a set of hex characters denoting the full length of the segment

Note that the segment of memory reserved must be within the bounds of the total address space

HLab3EG30B00018

**Figure 8:** Example of a header record

The Text Records are split into 4 sections, those being:
Record position 1: T (to indicate Text)
Record positions 2-5: a set of hex characters denoting the address at which the information is to be stored.
Record positions 6-9: a set of hex characters that represent the contents of the address specified by positions 2-5
Records positions 10-12: modification records for relocatable programs

T30B0127F

**Figure 9:** Example of a text record

The End Records are split into 2 sections, those being:
Record position 1: E (to indicate End)
Record positions 2-5: a set of hex characters denoting the address at which execution is to begin

E30B0

**Figure 10:** Example of an end record

## Listing file

The listing file becomes a bit more complicated, as it is a more complex file, that is not used for input to any other program and is meant to be a much more detailed breakdown of how the code is translated from input to output.

This file is split into seven columns, as follows:

- (Addr hex): This column contains the address where the contents are being stored, contained between parentheses. This column should have an equivalent in the text record positions 2-5. This column will always be 6 characters long including the parentheses, followed by a single space.

- Contents (hex): This column contains the contents to be stored in hex format. This column should have an equivalent in the text record positions 6-9. This column will always be 4 characters long, followed by 2 spaces.

- Contents (binary): This column is the binary translation of the hex string listed directly before. This column will always be 16 characters long, followed by a single space.

- (line #): This column contains the line number that correlates to the assembly code from the input file. In the case of an instruction like .STRZ, there can be multiple lines containing the same line number. This column will always be 6 characters long including the parentheses, followed by a single space.

- Label: This column contains the label or symbol from the line of assembly code. If there is no label in the line of assembly code then it will be just a blank space in this column. This column will always be 6 characters long (with spaces filling at the end of the label, not 6 characters long) followed by 10 spaces.

- Instruction: This column contains the instruction from the line of assembly code. In the case of a .STRZ instruction, only the first line will contain the instruction, whereas the remaining lines that are still going over the same instruction, will have blank space instead. This column will always be 5 characters long (with spaces filling at the end of the instruction if the instruction is not 5 characters long) followed by a single space.

- Operands: This column contains the operands from the line of assembly code. In the case of a .STRZ instruction, only the first line will contain the operands, whereas the remaining lines that are still going over the same instruction, will have blank space instead. Because operands do not have a length requirement or limit, neither does this column, therefore this column will be as long as the operand is.

Note: If the line is for a literal at the end of the listing file, no label, instruction, or operands will be recorded, and the line number will be replaced by the word “lit”.

Note: The listing file does not include any comments from the input file.

Note: The listing file will output the operands exactly as they are in the input file, meaning commas are included and no spaces. In the case of a .STRZ instruction, this will include the quotation marks surrounding the string.

Note: Any do not care value (given by the xx’s in the assembly format as seen in Figure 4) are replaced with zeros.

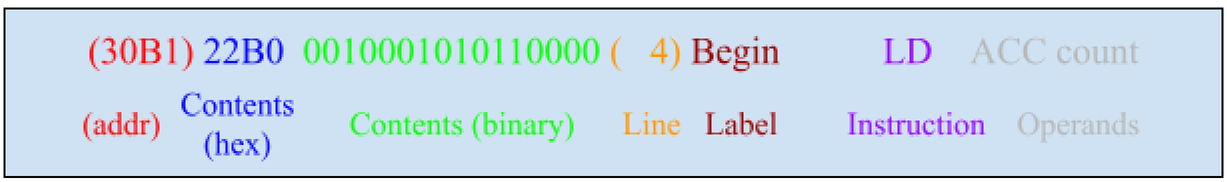



**Figure 11:** Example of a listing file line

# Errors

## Validator and App Errors

**Table 12:** All the errors handled in the Validator.

| Message | Error |
|---|---|
| Error: <file name> (No such file or directory) | File Not Found |
| An Empty Line is NOT valid | Empty Input File |
| Error: EQU and ORIG must have a label on line <line number><br>Line: <line> | ORIG or END pseudo-op with Labels |
| Error: Line too short on line <line number><br>Line: <line> | Line is less than 18 characters long |
| Error: Invalid operation on line <line number><br>Line: <line> | Invalid Operand in ORIG or END pseudo-ops |
| WAS EXPECTING " + "." + <type> + " GOT " + <operation> | Invalid Operation |
| Error: The following line is invalid: <line><br>Line: <line> | Invalid Instruction |
| Error: LINE MUST HAVE PSEUDO OP OR MACHINE OP on line <line number><br>Line: <line> | No comment or operation found on a line |
| Error: Symbol already exists<br>Line: <line> | Repeat Symbols |
| Error: this line is invalid on line <line number><br>Line: <line> | Not enough operations/operands in a line |
| Not enough arguments were provided to the program." +<br>"This program requires three arguments in the following order: ./program <input path> <object file output path> <listing file output path> | Application argument requirements are not met |

| | |
|---|---|
| A file must Have exactly one .ORIG and one .END pseudo op | Missing either an orig or end |
| Error: Invalid operation on line <line number><br>Line: <line> | No operands and pseudo op is not ORIG or END |
| Error: Invalid Operand <operand> on line <line number><br>Line: <line> | Invalid operands to instruction |
| Error: Invalid Operation on line <line number><br>Line: <line> | Invalid operation |
| Error: Label must start with an alphabetic character that is NOT a R OR an x on line <line number><br>Line: <line> | Label starts with an x or R |
| Error: White space must NOT contain any characters on line <line number><br>Line: <line> | Characters that aren't spaces in white space sections of input lines |
| No Label Allowed for ENT or EXT operations on line <line number><br>Line: <line> | ENT and/or EXT operations must not have a label |
| ENT and EXT must be declared right after ORIG on line <line number><br>Line: <line> | ENT and/or EXT exist in the file that are not positioned after ORIG pseudo op |
| ORIG must be the first pseudo op in the file | ORIG is not the first non comment line in the file |

## Pass2 Errors

The errors checked under Pass2 are the ranges given for the constants/absolute symbols, proper usage of symbols in allowed locations, and page ranges for address operations. All errors are handled by throwing an Exception to the user with the line of assembly code that is incorrect. Table X1 highlights the type of error and how it is handled (error message).

**Table 13:** All the errors handled in Pass 2.

| Message | Error |
|---|---|
| General checks | |
| “Line [num]: Invalid [assembly line] value” where the line number is specified under the line and [assembly line] is the component that tried invoking it. For example, if a register value is greater than 0xFFFF, then the assembly line would read Line [num]: Invalid register value”. | Invalid values used throughout the assembly lines that are outside the range of the machine’s possible value |
| “Line [num]: Symbol “[symbol]” is not found in the symbol table” | Undefined symbols used in the operand field of the assembly code |
| “Line [num]: Invalid [assembly line] value” | Absolute symbols used in the operand field of the assembly code that is outside the range of the machine’s possible value |
| “Line [num]: Segment size is greater than 0xFFFF” | When the segment size for the program is greater than 0xFFFF (maximum size) |
| | |
| Registers | |
| “Line [num]: Register value not within range (decimal and hex: [0 - 7])” | Invalid register value that is outside the range of register accepted value |
| “Line [num]: Symbol “[symbol]” is not an absolute symbol” | Relative symbol used as a register value |
| “Line [num]: Register value not within range (decimal and hex: [0 - 7])” | Absolute symbol outside the range of the accepted register value |
| | |
| Immediates | |
| "Line [num]: Imm5 value is not within the specified range [#-16 - #15] or [0x0 - 0x1F]” | Invalid immediate (imm5 values of the instructions) value that is outside the range of immediate accepted value |
| “Line [num]: Symbol “[symbol]” is not an absolute symbol” | Relative symbol used as an immediate value |

| | |
|---|---|
| "Line [num]: Imm5 value is not within the specified range [#-16 - #15] or [0x0 - 0x1F]” | Absolute symbol outside the range of the accepted immediate value |
| | |
| Index | |
| "Line [num]: Index6 value is not within the specified range [#0 - #63] or [0x0 - 0x3F]" | Invalid index (index6 values of the instructions) value that is outside the range of immediate accepted value |
| “Line [num]: Symbol “[symbol]” is not an absolute symbol” | Relative symbol used as an index value |
| "Line [num]: Index6 value is not within the specified range [#0 - #63] or [0x0 - 0x3F]" | Absolute symbol outside the range of the accepted index value |
| | |
| Trap vector | |
| "Line [num]: Trapvect8 value is not within the specified range [#0 - #255] or [0x0 - 0xFF]" | Invalid trap vector (trapvect8 values of the instructions) value that is outside the range of immediate accepted value |
| “Line [num]: Symbol “[symbol]” is not an absolute symbol” | Relative symbol used as a trap vector value |
| "Line [num]: Trapvect8 value is not within the specified range [#0 - #255] or [0x0 - 0xFF]" | Absolute symbol outside the range of the accepted trap vector value |
| | |
| Address | |
| "Line [num]: Address value is not within the specified range [#0 - #65535] or [0x0 - 0xFFFF]" | Invalid address value that is outside the range of address accepted value |
| "Line [num]: Address value is not within the specified range [#0 - #65535] or [0x0 - 0xFFFF]" | Relative symbol outside the range of the accepted address value |
| "Line [num]: Address value is not within the specified range [#0 - #65535] or [0x0 - | Absolute symbol outside the range of the accepted address value |

| | |
|---|---|
| 0xFFFF]" | |
| "Line [num]: Address value is not within the same page number as PC (PC at page: #[pg_num1], Defined Address at page: #[pg_num2])" where the pg_num1 is the page number at which the current program counter is pointing at whereas pg_num2 is the page number the specified address is currently at | Any address operation no in the same page range as the program counter (pointer to the next instruction to execute) |

# Modified Appendix

## Additional Contents to the Assembly Language

There are new elements added to the assembler that allows for using symbols from different assembly files, known as external symbols.

### Addition of Pseudo-Ops

To enable using external symbols, new Pseudo-Operations are added to the assembler to allow for the processing of external symbols internally. These ops include the **.**ENT and **.**EXT pseudo-ops.

**.***ENT*: This pseudo-op is a reference to the **ent**ry name of a symbol. The list of symbols defined in the operand portion of the op says those symbols' values will be defined in the current segment. It also permits those symbols to be referenced in other segments.

**.***EXT*: This pseudo-op is a reference to the **ext**ernal name of a symbol. The list of symbols defined in the operand portion of the op says those symbols' are legitimate symbols from different segments and can be processed in the current segment. The symbol must be defined in a different segment.

The syntax for these ops follows an opcode and operand-only format. The operand portion of the code can only contain symbols and comments. There are no restrictions on the number of symbols defined under these operations, but each symbol must be as long as a regular assembler label (at most six characters long) and must be separated by a comma. Anything else will terminate the program with an error. The syntax followed by the pseudo-ops is given in Table 1M below.

**Table 1M:** Pseudo-ops syntax.

| Pseudo-op | Syntax |
|---|---|
| **.**ENT | **.**ENT Sym1,Sym2 |
| **.**EXT | **.**EXT Sym1,Sym2 |

There are a few requirements when defining symbols under these pseudo-ops.

- Each of the external symbols defined in the operand field must be a relative symbol.
- These pseudo-ops can only be used after the **.**ORIG pseudo-op and before any other instruction.

- If the pseudo-op is used within the specified lines of assembly (requirement 2), then multiple occurrences of the op are permitted.

Since the symbols defined by these ops are relative, they can only be used in fields that accept relative symbols (address field of the operand). Using these symbols in any other fields will crash the program with an error message printed to the console.

### Additional Changes to the Main Output File

As mentioned previously, the assembler creates an object file and listing file, but new additions are made only to the object file. Two of the new changes for the object file include N-records for **.**ENT symbols and X-records (modification records) for **.**EXT symbols.

Addition of X records

For each symbol defined by the **.**EXT has unknown values and cannot be determined until all the assembly files are converted to object files. Before passing the relocatable object file to the linker/loader, the program must know which records use external symbols. So, each of those records is attached with an X-record to the end. The X-records include a nine-bit and sixteen-bit variant. The nine-bit X-records are attached for instructions that use page offset as part of their fields (address field). The sixteen-bit X-records are used for instructions where the entire operand field requires only an address. These records replace the previously used M-records to use the segment name appended to the end of X-records instead.

The format of the X-records is as follows:

1. X
2. Bit-offset (9 or 16)
3. Symbol name/Segment name

Figure 1M below shows examples of text records with X-records with both modifiers.

**Figure 1M:** X-record examples.

```
T00000085X9Sym → T 0000 0085 X 9 Sym1
T00000000X16Sym2 → T 0000 0000 X 16 Sym2
```

Addition of N records

For each symbol given by the **.**ENT op in the operand field is defined in the provided segment. The N-records allow for defining each of those symbols for the linker/loader. The N records are placed after the header record and before any text record. The N-record contains the name of the symbol and the value assigned to it, separated by an equal sign. As the N-records use an equal sign to indicate the value of a symbol, literals are not permitted to be used as entry symbols.

The format of the N-records is as follows:

1. N
2. =
3. Value of the symbol in hex (without the "x")

Figure 2M below shows examples of text records with N-records with symbols of different character lengths.

**Figure 2M:** N-record examples.

```
NSym1=0 → N Sym1 = 0
NLabel1=4 → N Label1 = 4
```

## Additional Errors

With the modified assembler, there are a few more errors checked for the new pseudo-ops. These errors are noted in Table 2M below.

**Table 2M:** All the new errors handled by the modified assembler.

| Error Message | Error | Potential resolution |
|---|---|---|
| Line [num]: External symbol "[symbol]" cannot be used as an absolute symbol | If trying to use an external symbol as an absolute symbol | Only use external symbols where relative symbols are allowed |
| Line [num]: Symbol "[symbol]" is already defined in the symbol table (cannot be used as an external symbol) | If trying to use an entry symbol as an external symbol | Do not define the same entry and external symbols |
| Line [num]: More than 5 entry symbols defined in the operand field | If the user defines more than 5 symbols in the .ENT operand field | Define a total of 1-5 symbols in the operand field |
| Line [num]: More than 5 external symbols defined in the operand field | If the user defines more than 5 symbols in the .EXT operand field | Define a total of 1-5 symbols in the operand field |
| No Labels Allowed for ENT or EXT operations on line <line number> | ENT and/or EXT operations must not have a label | Remove the label |

| | | |
|---|---|---|
| Line: <line> | | |
| ENT and EXT must be declared right after ORIG on line <line number><br>Line: <line> | ENT and/or EXT exist in the file that are not positioned after ORIG pseudo op | Place this line right after the ORIG pseudo op |