CSCI 2122 Assignment 4

Due date: 11:59pm, Friday, March 22, 2024, submitted via git

Objectives

The purpose of this assignment is to practice your coding in C, and to reinforce the concepts discussed in class on program representation.

In this assignment[1] you will implement a binary translator[2] like Rosetta[3]. Your program will translate from a simple instruction set (much simpler than x86) to x86 and generate x86 assembly code. The code will then be tested by assembling and running it. This assignment is divided into two parts to make it simpler. In the first part, you will implement the loader and a simple translator, which translates the simpler instructions. In the second part, you will extend the translator to translate more complex instructions.

Preparation:

1. Complete Assignment 0 or ensure that the tools you would need to complete it are installed.
2. Clone your assignment repository:

   `https://git.cs.dal.ca/courses/2024-winter/csci-2122/assignment-4/????.git`

   where ???? is your CSID. Please see instructions in Assignment 0 and the tutorials on Brightspace if you are not sure how.

Inside the repository there is one directory: **`xtra`**, where code is to be written. Inside the directory is a **`tests`** directory that contains tests that will be executed each time you submit your code. Please do not modify the **`tests`** directory or the **`.gitlab-ci.yml`** file that is found in the root directory. Modifying these files may break the tests. These files will be replaced with originals when the assignments are graded. You are provided with sample **`Makefile`** files that can be used to build your program. If you are using CLion, a **`Makefile`** will be generated from the **`CMakeLists.txt`** file generated by CLion.

Background:

For this assignment you will translate a binary in a simplified RISC-based 16-bit instruction set to x86-64 assembly. Specifically, the X instruction set comprises a small number (approximately 30) instructions, most of which are two bytes (one word) in size.

The X Architecture has a 16-bit word-size and 16 general purpose 16-bit registers (r0 . . . r15). Nearly all instructions operate on 16-bit chunks of data. Thus, all values and addresses are 16 bits in size. All 16-bit values are also encoded in big-endian format, meaning that the most-significant byte comes first.

Apart from the 16 general purpose registers, the architecture has two special 16-bit registers: a program counter (PC), which stores the address of the next instruction that will be executed, and the status (F), which stores bit-flags representing the CPU state. The least significant bit of the status register (F) is the condition flag, which represents the truth value of the last logical test operation. The bit is set to true if the condition was true, and to false otherwise.

[1] The idea for this assignment came indirectly from Kyle Smith.
[2] https://en.wikipedia.org/wiki/Binary_translation
[3] https://en.wikipedia.org/wiki/Rosetta_(software)

Additionally, the CPU uses the last general-purpose register, **r15**, to store the pointer to the program stack. This register is incremented by two when an item is popped off the stack and decremented by two when an item is pushed on the stack. The program stack is used to store temporary values, arguments to a function, and the return address of a function call.

The X Instruction Set

The instruction set comprises approximately 30 instructions that perform arithmetic and logic, data movement, stack manipulation, and flow control. Most instructions take registers as their operands and store the result of the operation in a register. However, some instructions also take immediate values as operands. Thus, there are four classes of instructions: 0-operand instructions, 1-operand instructions, 2-operand instructions, and extended instructions, which take two words (4 bytes) instead of one word.

All but the extended instructions are encoded as a single word (16 bits). The extended instructions are also one word but are followed by an additional one-word operand. Thus, if the instruction is an extended instruction, the PC needs an additional increment of 2 during the instruction's execution. As mentioned previously, most instructions are encoded as a single word. The most significant two bits of the word indicates whether the instruction is a 0-operand instruction (00), a 1-operand instruction (01), a 2-operand instruction (10), or an extended instruction (11). For a 0-operand instruction encoding, the two most significant bits are 00 and the next six bits represent the instruction identifier. The second byte of the instruction is 0.

2	6	8
00	######	00000000

2	1	5	4	4
01	0	#####	RRRR	0000

2	1	5	8
01	1	#####	IIIIIIII

For a 1-operand instruction encoding, the two most significant bits are 01, the next bit indicates whether the operand is an immediate or a register, and the next five bits represent the instruction identifier. If the third most significant bit is 0, then the four most significant bits of the second byte encode the register that is to be operated on (0… 15). Otherwise, if the third most significant bit is 1, then the second byte encodes the immediate value.

2	6	4	4
10	######	SSSS	DDDD

For a 2-operand instruction encoding, the two most significant bits are 10, and the next six bits represent the instruction identifier. The second byte encodes the two register operands in two four-bit chunks. Each of the 4-bit chunks identifies a register (r0 … r15).

2	1	5	8
11	0	#####	00000000

2	1	5	4	4
11	1	#####	RRRR	0000

For an extended instruction encoding, the two most significant bits are 11, the next bit indicates whether a second register operand is used, and the next five bits represent the instruction identifier. If the third most significant bit is 0, then the instruction only uses the one-word immediate operand that follows the instruction. Otherwise, if the third most significant bit is 1, then the four most significant bits of the second byte encode a register (1 … 15) that is the second operand.

The instruction set is described in Tables 1, 2, 3, and 4. Each description includes the mnemonic (and syntax), the encoding of the instruction, the instruction's description, and function. For example, the **add**, **loadi**, and **push** instructions have the following descriptions:

Mnemonic	Encoding	Description	Function
add rS, rD	10000001 **S D**	Add register rS to register rD.	rD ← rD + rS

loadi V, rD	11100001 **D 0**	Load immediate value or address V into register rD.	rD ← memory[PC]
			PC ← PC + 2

push rS	01000011 **S 0**	Push register rS onto program stack.	r15 ← r15 - 2 memory[r15] ← rS

First, observe that the **add** instruction takes two register operands and adds the first register to the second. All 2-operand instructions operate only on registers and the second register is both a source and destination, while the first is the source. It is a 2-operand instruction; hence the first two bits are 10, its instruction identifier is 000001 hence the first byte of the instruction is 0x81.

Second, the **loadi** instruction is an extended instruction that takes a 16-bit immediate and stores it in a register. Hence, the first two bits are 11, the register bit is set to 1, and the instruction identifier is 00001. Hence, the first byte is encoded as 0xE1.

Third, the **push** instruction is a 1-operand instruction, taking a single register operand. Hence, the first two bits are 01, the immediate bit is 0, and the instruction identifier is 00011. Hence, the first byte is encoded as 0x43.

Note that **S** and **D** are 4-bit vectors representing S and D.

Table 1: 0-Operand Instructions

Mnemonic	Encoding	Description	Function
ret	00000001 **0**	Return from a procedure call.	P C ← memory[r15] r15 ← r15 + 2
cld	00000010 **0**	Stop debug mode	See Debug Mode below.
std	00000011 **S 0**	Start debug mode	See Debug Mode below.

Table 1: 1-Operand Instructions

Mnemonic	Encoding	Description	Function
neg rD	01000001 **D 0**	Negate register rD .	rD ← −rD
not rD	01000010 **D 0**	Logically negate register rD .	rD ←!rD
inc rD	01001000 **D 0**	Increment rD .	rD ← rD + 1
dec rD	01001001 **D 0**	Decrement rD .	rD ← rD – 1

push rS	01000011 **S 0**	Push register rS onto the pro- gram stack.	r15 ← r15 – 2
			memory[r15] ← rS

pop rD	01000100 **D 0**	Pop value from stack into register rD.	rD ← memory[r15]
			r15 ← r15 + 2

out rS	01000111 **S 0**	Output character in rS to std- out.	output ← rS (see below)

br L	01100001 **L**	Branch relative to label L if condition bit is true.	if F & 0x0001 == 0x001:
			PC ← PC + L – 2

jr L	01100010 **L**	Jump relative to label L.	PC ← PC + L – 2

Table 3: 2-Operand Instructions

Mnemonic	Encoding	Description	Function
add rS , rD	10000001 **S D**	Add register rS to register rD .	rD ← rD + rS
sub rS , rD	10000010 **S D**	Subtract register rS from register rD.	rD ← rD - rS
mul rS , rD	10000011 **S D**	Multiply register rD by register rS.	rD ← rD * rS
and rS , rD	10000101 **S D**	And register rS with register rD .	rD ← rD & rS
or rS , rD	10000110 **S D**	Or register rS with register rD .	rD ← rD | rS
xor rS , rD	10000111 **S D**	Xor register rS with register rD .	rD ← rD ^ rS

test rS_1, rS_2	10001010 **S D**	Set condition flag to true if and only if $rS_1 \wedge rS_2$ is not 0.	if rS_1 & rS_2 != 0:
			  F ← F | 0x0001
			else:
			  F ← F & 0xFFFE

cmp rS_1, rS_2	10001011 **S D**	Set condition flag to true if and only If $rS_1 < rS_2$.	if $rS_1 < rS_2$:
			  F ← F | 0x0001
			else:
			  F ← F & 0xFFFE

equ rS_1, rS_2	10001100 **S D**	Set condition flag to true if and only if $rS_1 == rS_2$.	if $rS_1 == rS_2$:
			  F ← F | 0x0001
			else:
			  F ← F & 0xFFFE

mov rS , rD	10001101 **S D**	Copy register rS to register rD .	rD ← rS
load rS , rD	10001110 **S D**	Load word into register rD from memory pointed to by register rS.	rD ← memory[rS]
stor rS , rD	10001111 **S D**	Store word from register rS to memory at address in register rD.	memory[rD] ← rS
loadb rS , rD	10010000 **S D**	Load byte into register rD from memory pointed to by register rS.	rD ← (byte)memory[rS]
storb rS , rD	10010001 **S D**	Store byte from register rS to memory at address in register rD.	(byte)memory[rD] ← rS

Table 3: Extended Instructions

Mnemonic	Encoding	Description	Function
jmp L	11000001 **0**	Absolute jump to label L.	PC ← memory[PC]

call L	11000010 **0**	Absolute call to label L..	r15 ← r15 – 2
			memory[r15] ← PC + 2
			PC ← memory[PC]

loadi V, rD	11100001 **D 0**	Load immediate value or address V into register rD.	rD ← memory[PC]
			PC ← PC + 2

Note that in the case of extended instructions, the label **L** or value **V** are encoded as a single word (16-bit value) following the word containing the instruction. The **0** in the encodings above represents a 4-bit 0 vector.

An assembler is provided for you to use (if needed). **Please see the manual at the end of the assignment.**

The Xtra Translation Specification (IMPORTANT)

The binary translation is conducted in the following manner. The translator

1. Opens the specified file containing the X binary code.
2. Outputs a prologue (see below), which will be the same for all translations.
3. It then enters a loop that
   a. Reads the next instruction from the binary
   b. Decodes the instruction, and
   c. Outputs the corresponding x86 assembly instruction(s). If the instruction is an extended, an additional two bytes will need to be read.
   d. The loop exits when the instruction composed of two 0 bytes is read.
4. Outputs an epilogue.

Prologue

The translator first outputs a simple prologue that is the same for all translations. The prologue is shown on the right.

```
.globl test
test:
    push %rbp
    mov %rsp, %rbp
```

Epilogue

After the translator finishes translating, it outputs a simple epilogue that is the same for all translations. The epilogue is shown on the right.

```
    pop %rbp
    ret
```

Translation

Each X instruction will need to be translated into the corresponding instruction or instructions in x86-64 assembly. See table on right for examples. Most instructions will have a direct corresponding instruction in x86 assembly so the translation will be easy. Some instructions, like the **`equ`**, **`test`**, and **`cmp`**, instructions may require multiple x86 instructions for a single X instruction. **Note:** The translator will need to perform a register mapping.

X Instruction	Output x86 Assembly
`mov r0, r1`	`mov %rax, %rdi`
`loadi 42, r0`	`mov $42, %rax`
`push r0`	`push %rax`
`add r0, r1`	`add %rax, %rdi`

Register Mapping

The X architecture has 16 general and the ***F*** status register. In x86-64 there are also 16 general purpose registers. The register mapping on the right must be used when translating from X to x86-64. Note that for this exercise register ***r13*** will not be used by the X executables. Instead of ***r13*** (X) being mapped to ***r15*** (x86), the ***F*** register (X) is mapped to register ***r15*** (x86). **Note**: for this assignment, It is ok to map 16-bit registers to 64-bit registers.

X Registers	x86 Registers
`r0`	`%rax`
`r1`	`%rbx`
`r2`	`%rcx`
`r3`	`%rdx`
`r4`	`%rsi`
`r5`	`%rdi`
`r6`	`%r8`
`r7`	`%r9`
`r8`	`%r10`
`r9`	`%r11`
`r10`	`%r12`
`r11`	`%r13`
`r12`	`%r14`
`F`	`%r15`
`r14`	`%rbp`
`r15`	`%rsp`

Debug mode STD and CLD

The **`std`** and **`cld`** X instructions enable and disable debug mode on the X architecture. However, debug mode does not exist in x86-64. Instead, when a **`std`** instruction is encountered, the translator should set an internal *debug* flag in the translator and, clear the debug flag when it encounters the **`cld`** instruction.

When the *debug* flag is true, the translator should output the assembly code on the right **before** translating each X instruction.

```
call debug
```

Output and the OUT Instruction (For Task 2)

On the X architecture, the **`out rN`** instruction outputs to the screen the character stored in register ***rN***. However, no such instruction exists in x86-64. Instead, the **`out`** instruction is translated to a call to the function **`outchar(char c)`**, which performs this function. Recall that the first argument is passed in register **`%rdi`**. Consequently, the corresponding translation code will need to save the current value of **`%rdi`** on the stack, move the byte to be printed into **`%rdi`**, call **`outchar`**, and restore **`%rdi`**.

Your task will be to implement the Xtra binary translator which is used to translate into x86 assembly programs assembled with the X assembler.

Task 1: Implement the Simple Xtra

Your first task is to implement a simple version of the translator that works for the simple instructions. The source file `main.c` should contain the **`main()`** function. The translator should:

1. Take one (1) argument on the command line: The argument is the object/executable file of the program to translate. For example, the invocation
   ```
   ./xtra hello.xo
   ```
   instructs the translator to translate the program `hello.xo` into x86-64 assembly.
2. Open for reading the file specified on the command-line.
3. Output (to stdout) the prologue.
4. In a loop,
   a. Read in instruction.
   b. If the instruction is 0x00 0x00, break out of the loop.
   c. Translate the instruction and output (to stdout) the x86-64 assembly.
5. Output (to stdout) the epilogue.

Input

The input to the program is via the command line. The program takes one argument, the name of the file containing the assembled X code.

Processing

All input shall be correct. All program files shall be at most 65536 bytes (64KB). The translator must be able to translate **all** instructions **except**:

Instruction	Description
ret	Return from a procedure call.
br L	Branch relative to label L if condition bit is true.
jr L	Jump relative to label L.
jmp L	Absolute jump to label L.
call L	Absolute call to label L.
load rS , rD	Load word into register rD from memory pointed to by register rS.
stor rS , rD	Store word from register rS to memory at address in register rD.
loadb rS , rD	Load byte into register rD from memory pointed to by register rS.
storb rS , rD	Store byte from register rS to memory at address in register rD.
out rS	Output character in rS to stdout.

Recommendation: While no error checking is required, it may be helpful to still do error checking, e.g., make sure files are properly opened because it will help with debugging as well.

Output

Output should be to stdout. The output is the translated assembly code. The format should ATT style assembly. The exact formatting of the assembly is up to you, but the assembly code will be passed through the standard assembler (**as**) on `timberlea`. See next section for how to test your code. (See example)

Testing

To test your translator, the test scripts will assembler, link, and run the translated code! ☺ A `runit.sh` script is provided. The script takes an X assembly file as an argument:

```
./runit.sh foo.xas
```

The script:

1. Assembles the `.xas` file with the provided (**`xas`**) to create a `.xo` file.
2. Runs **`xtra`** on the `.xo` file, to create a corresponding x86 `.s` assembly file.
3. Assembles, compiles, and links the generated assembly file with some runner code, creating an executable. The runner is composed of `runner.c`, `regsdump.s`, and the translated `.s` file. Please DO NOT delete the first two files.
4. Runs the executable.

This script is used by the test scripts and is also useful for you to test your code.

Most of the tests use the **`std`** instruction to turn on debugging and output the state of the registers after each instruction is executed. For most of the tests the output being compared are the register values.

Example

Original X assembly code	**Translated x86 code**

```
   loadi 2, r0
   loadi 3, r1
   loadi 4, r2
   loadi 5, r3
   loadi 7, r5
   std          # turn debugging on
   add r2, r3
   mul r2, r1
   cld          # turn debugging off
   neg r0
   inc r5

.literal 0
```
	```
	.globl test
	test:
	    push %rbp
	    mov %rsp, %rbp
	    mov $2, %rax
	    mov $3, %rbx
	    mov $4, %rcx
	    mov $5, %rdx
	    mov $7, %rdi
	    call debug
	    add %rcx, %rdx
	    call debug
	    imul %rcx, %rbx
	    call debug
	    neg %rax
	    inc %rdi
	    pop %rbp
	    ret
	```

Task 2: The Full Translator

Your second task is to extend **`xtra`** to translate the instructions exempted in Task 1. Implement translation for the following instructions.

Instruction	Description
ret	Return from a procedure call.
br L	Branch relative to label L if condition bit is true.
jr L	Jump relative to label L.
jmp L	Absolute jump to label L.
call L	Absolute call to label L.
load rS , rD	Load word into register rD from memory pointed to by register rS.
stor rS , rD	Store word from register rS to memory at address in register rD.
loadb rS , rD	Load byte into register rD from memory pointed to by register rS.
storb rS , rD	Store byte from register rS to memory at address in register rD.
out rS	Output character in rS to stdout.

Input
The input is the same as Task 1.

Processing
The processing is the same as for Task 1. The challenge is that translation is a bit more challenging.

First, for many of the additional instructions you will need to emit more than one assembly instruction. This is particularly true for the conditional branching and output instructions.

Second, for the branching instructions you will need to compute the labels where to branch to. The easy solution is to create a label for each instruction being translated. The label should encode the address in the name. For example, **`L1234`** would be the label for the X instruction at address ***1234***. By doing this, you will not need to keep a list or database of labels.

Third, the addresses used by the load AND

Output
The output is the same as Task 1.

Example

Original X assembly code	Translated x86 code
`  loadi 1, r0`	`.globl test`
`  jmp j1`	`test:`
`j2:`	`    push %rbp`
`  loadi 3, r0`	`    mov %rsp, %rbp`
`  jmp j3`	`.L0000:`
`j1:`	`    mov $1, %rax`
`  loadi 2, r0`	`.L0004:`
`  jmp j2`	`    jmp .L0010`
`j3:`	`.L0008:`
`  std          # turn debugging on`	`    mov $3, %rax`
`  loadi 4, r0`	`.L000c:`
	`    jmp .L0018`
`.literal 0`	`.L0010:`
	`    mov $2, %rax`
	`.L0014:`
	`    jmp .L0008`
	`.L0018:`
	`.L001a:`
	`    call debug`
	`    mov $4, %rax`
	`.L001e:`
	`    call debug`
	`    pop %rbp`
	`    ret`

Hints and Suggestions
- Use the unsigned short type for all registers and indices.
- Use two files: one the main program and one for the translator loop.
- Start early, this is the hardest assignment of the term and there is a lot to digest in the assignment specifications.

Assignment Submission

Submission and testing are done using Git, Gitlab, and Gitlab CI/CD. You can submit as many times as you wish, up to the deadline. Every time a submission occurs, functional tests are executed, and you can view the results of the tests. To submit use the same procedure as Assignment 0.

Grading

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The assignment will be graded based on three criteria:

Functionality: “Does it work according to specifications?”. This is determined in an automated fashion by running your program on several inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Quality of Solution: “Is it a good solution?” This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: “Is it well written?” This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Style Guide in the Assignment section of the course in Brightspace.

The following grading scheme will be used:

Task	100%	80%	60%	40%	20%	0%
Functionality (20 marks)	Equal to the number of tests passed.					
Solution Quality Task 1 (15 marks)	Implemented correctly. Code is robust.	Implemented correctly. Code is not robust.	Some minor bugs.	Major flaws in implementation	An attempt has been made.	No code submitted or code does not compile
Solution Quality Task 2 (5 marks)	Implemented correctly. Code is robust.	Implemented correctly. Code is not robust.	Some minor bugs.	Major flaws in implementation	An attempt has been made	
Code Clarity (10 marks) Indentation, formatting, naming, comments	Code looks professional and follows all style guidelines	Code looks good and mostly follows style guidelines.	Code is mostly readable and mostly follows some of the style guidelines	Code is hard to read and follows few of the style guidelines	Code is not legible	

Assignment Testing without Submission

Testing via submission can take some time, especially if the server is loaded. You can run the tests without submitting your code by using the provided `runtests.sh` script. Running the script with no arguments will run all the tests. Running the script with the test number, i.e., 00, 01, 02, 03, … 09, will run that specific test. Please see below for how run the script.

Get your program ready to run

If you are developing directly on the unix server,

1. SSH into the remote server and be sure you are in the **xtra** directory.
2. Be sure the program is compiled by running **make**.

If you are using CLion

1. Run your program on the remote server as described in the CLion tutorials.
2. Open a remote host terminal via **Tools → Open Remote Host Terminal**

If you are using VSCode

1. Run your program on the remote server as described in VSCode tutorials.
2. Click on the Terminal pane in the bottom half of the window or via **Terminal → New Terminal**

Run the script

3. Run the script in the terminal by using the command:

```
./runtest.sh
```

to run all the tests, or specify the test number to run a specific test, e.g. :

```
./runtest.sh 07
```

You will see the test run in the terminal window.

The X Assembler (xas)

An assembler (xas) is provided to allow you to write and compile programs for the X Architecture. To make the assembler, simply run "make xas" in the **`xtra`** directory. To run the assembler, specify the assembly and executable file on the command-line. For example,

```
./xas example.xas example.xo
```

Assembles the X assembly file **`example.xas`** into an X executable **`example.xo`**.

The format of the assembly files is simple.

1. Anything to the right of a # mark, including the #, is considered a comment and ignored.
2. Blank lines are ignored.
3. Each line in the assembly file that is not blank must contains a directive, a label and/or an instruction. If the line contains both a label and an instruction, the label must come first.
4. A label is of the form

   `identifier:`

   where identifier consists of any sequence of letters (A-Za-z), digits (0-9), or underscores ( ) as long the first character is not a digit. A colon (:) must terminate the label. A label represents the corresponding location in the program and may be used to jump to that location in the code.
5. An instruction consists of a mnemonic, such as add, loadi, push, etc., followed by zero or more operands. The operand must be separated from the mnemonic by one or more white spaces. Multiple operands are separated by a comma.

6. If an operand is a register, then it must be in the form r# where # ranges between 0 and 15 inclusively. E.g., **`r13`**.
7. If the instruction is an immediate, then the argument may either be a label, or an integer. Note: labels are case-sensitive. If a label is specified, no colon should follow the label.
8. Directives instruct the assembler to perform a specific function or behave in a specific manner. All directives begin with a period and are followed by a keyword. There are three directives: .literal, .words and .glob, each of which takes an operand.
   (a) The **`.literal`** directive encodes a null terminated string or an integer at the present location in the program. E.g.,
   ```
   mystring:
   .literal "Hello World!"
   myvalue:
   .literal 42
   ```
   encodes a nil-terminated string followed by a 16-bit (1 word) value representing the decimal value 42. In this example, there are labels preceding each of the encodings so that it is easy to access these literals. That is, the label **`mystring`** represents the address (relative to the start of the program) where the string is encoded, and the label **`myvalue`** represents the address (relative to the start of the program) of the value. This is used in the **`hello.xas`** example.
   (b) The **`.words`** directive sets aside a specified number of words of memory at the present location in the program. E.g.,
   ```
   myspace:
   .words 6
   ```
   allocates 6 words of memory at the present point in the program. This space is not initialized to any specific value. Just as before, the label preceding the directive represents the address of the first word, relative to the start of the program. This is used in xrt0.xas to set aside space for the program stack.
   (c) The **`.glob`** directive exports the specified symbol (label) if it is defined in the file and imports the specified symbol (label) if it is used but not defined in the file. E.g.,
   ```
   .glob foo
   .glob bar
   ...
   loadi bar, r0
   ...
   foo:
     .literal "Hello World!"
   ```
   declares two symbols (labels) foo and bar. Symbol foo is declared in this file, so it will be exported (can be accessed) in other files. The latter symbol, bar, is used but not defined. When this file is linked, the linker looks for the symbol (label) definition in other files makes all references to the symbol refer to where it is defined.

Note: it is recommended that you place literals and all space allocations at the end of your program, so that they will not interfere with program itself. If you do place literals in the middle of your program, you will need to ensure that your code jumps around these allocations.

There are several example assembly files provided (ending in `.xas`). You can assemble them by invoking the assembler, for example:
```
./xas example.xas example.xo
```
This invocation will cause the assembler to read in the file `example.xas` and generate an output file `example.xo`. Feel free to look at the code for the assembler.