Lab 3 – Linker/Loader

Modified Assembler Test Plan

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

Overview

To test both the overall assembler, as well as the individual components and methods, we have two primary forms of testing. The first is the JUnit testing to test the individual components and methods. Every single method in each class has several tests to ensure that all cases, including normal and edge cases (and error cases if necessary) are dealt with to ensure smooth operation of individual methods. None of the unit tests require any user input. The second portion of our testing was the System level tests, which revolve around fully functional assembly code files that are run through our assembler, and checked against our handmade outputs to ensure that the assembler translated the files properly and with no errors occurring. These system tests attempted to test multiple features of the assembler to ensure that, beyond just the unit tests, the different methods wouldn't interfere with each other and everything could work together in a meaningful way that went beyond just one or two things at a time.

System-Level Testing

Test Case:	Routine Sample Input

Input File	;234567890123456789012345689O
	; Example Program
	Lab2EG .ORIG x30B0
	count .FILL #4
	Begin LD ACC,count ;R1 <- 4
	LEA R0,msg
	loop TRAP x22
	;print "hi! "
	ADD ACC,ACC,#-1 ;R1--
	BRP loop
	JMP Next
	msg .STRZ "hi! "
	Next AND R0,R0,x0 ;R0 <- 0
	NOT R0,R0 ;R0 <- xFFFF
	ST R0,Array ;M[Array] <- xFFFF
	LEA R5,Array
	LD R6,=#100 ;R6 <= #100
	STR R0,R5,#1 ;M[Array+1] <= xFFFF
	TRAP x25
	ACC .EQU #1
	; ----- Scratch Space -----
	Array .BLKW #3
	.FILL x10
	.END Begin

Expected Output	Object File

Output:

Object FIle

```
HLab2EG30B00018

T30B00004

T30B122B0
```

```
T30B2E0B7

T30B3F022

T30B4127F

T30B502B3

T30B640BC

T30B70068

T30B80069

T30B90021

T30BA0020

T30BB0000

T30BC5020

T30BD9000

T30BE30C3

T30BFEAC3

T30C02CC7

T30C17141

T30C2F025

T30C60010

T30C70064

E30B1
```

Listing File

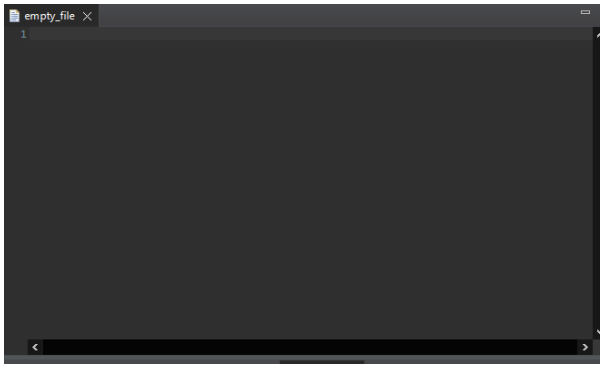
```
      ( 2) Lab2EG .ORIG x30B0

(30B0) 0004 0000000000000100 ( 3) count .FILL #4

(30B1) 22B0 0010001010110000 ( 4) Begin LD ACC,count

(30B2) E0B7 1110000010110111 ( 5) LEA R0,msg

(30B3) F022 1111000000100010 ( 6) loop TRAP x22

(30B4) 127F 0001001001111111 ( 7) ADD ACC,ACC,#-1

(30B5) 02B3 0000001010110011 ( 8) BRP loop

(30B6) 40BC 0100000010111100 ( 9) JMP Next

(30B7) 0068 0000000001101000 ( 10) msg .STRZ "hi! "

(30B8) 0069 0000000001101001 ( 10)

(30B9) 0021 0000000000100001 ( 10)

(30BA) 0020 0000000000100000 ( 10)

(30BB) 0000 0000000000000000 ( 10)

(30BC) 5020 0101000000100000 ( 11) Next AND R0,R0,x0

(30BD) 9000 1001000000000000 ( 12) NOT R0,R0

(30BE) 30C3 0011000011000011 ( 13) ST R0,Array

(30BF) EAC3 1110101011000011 ( 14) LEA R5,Array

(30C0) 2CC7 0010110011000111 ( 15) LD R6,=#100

(30C1) 7141 0111000101000001 ( 16) STR R0,R5,#1

(30C2) F025 1111000000100101 ( 17) TRAP x25

                                                    ( 18) ACC .EQU #1

(30C3)                                    ( 20) Array .BLKW #3

(30C6) 0010 0000000000010000 ( 21) .FILL x10

                                                    ( 22) .END Begin

(30C7) 0064 0000000001100100 ( lit)
```

Test Case:	Comment Only File
Input File	;F3jae53acbCgKI5 ;b8TSy8fKNRphztk ;M4WuSwiEfisQBHe ;lyQd4jNkys99YhA ;DKZE8DTmurNsCzV ;bsehWacRsY5sojU ;6EJjANS2450hglr ;8E7vKHev30GJIzg ;0x7sdSTG6RHsDnK ;wduDqOaS9plXRZP ;
Expected Output	Exception stating the file must have exactly one of .ORIG and .END pseudo op

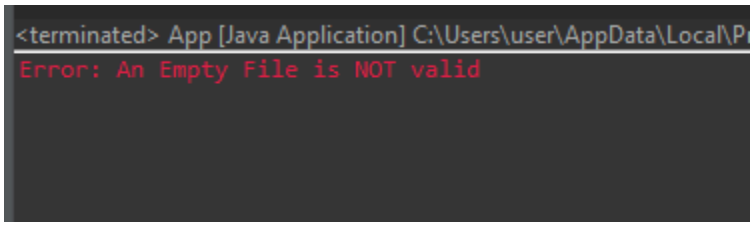
Output:

```
<terminated> App [Java Application] C:\Users\user\AppData\Local\Programs\Eclipse
A file must Have excatly one .ORIG and one .END psuedo op
```

Test Case:	Empty File

Input File	
Expected Output	Error noting we cannot have an empty file as an input

Output:



Test Case:	Input lines character counts are less than 18
Input File	Lab2EG .ORIG x30B0 .END x30B0
Expected Output	Error noting the line is not of the expected length

Output:

```
<terminated> App [Java Application] C:\Users\user\AppDa
Error: Line too short. On line 1
Line: .END    x30B0
```

Test Case:	Invalid Test where a label begins with ‘R’

Input File	;234567890123456789012345689
	0
	; Example Program
	Lab2EG .ORIG x30B0
	count .FILL #4
	Begin LD ACC,count ;R1 <- 4
	LEA R0,msg
	Roop TRAP x22 ;print "hi! "
	ADD ACC,ACC,#-1 ;R1--
	BRP loop
	JMP Next
	msg .STRZ "hi! "
	Next AND R0,R0,x0 ;R0 <- 0
	NOT R0,R0 ;R0 <- xFFFF
	ST R0,Array ;M[Array] <- xFFFF
	LEA R5,Array
	LD R6,=#100 ;R6 <= #100
	STR R0,R5,#1 ;M[Array+1] <= xFFFF
	TRAP x25
	ACC .EQU #1
	; ----- Scratch Space -----
	Array .BLKW #3

	.FILL x10 .END Begin
Expected Output	Error noting a label's syntax is invalid

Output:

```
<terminated> App [Java Application] C:\Users\user\AppData\Local\Programs\Eclipse Adoptium\jdk-17.0.4.101-hotspot\bin\javaw.e
Error: Label must start with an alphabetic character that is NOT a R OR an x On line 6
Line: Roop     TRAP    x22             ;print "hi! "
```

Test Case:	Using a literal for instruction LEA
Input File	;234567890123456789012345689 ; Example Program Lab2EG .ORIG x30B0 count .FILL #4 Begin LD ACC,count ;R1 <- 4 LEA R0,msg loop TRAP x22 ;print "hi! " ADD ACC,ACC,#-1 ;R1-- BRP loop JMP Next msg .STRZ "hi! " Next AND R0,R0,x0 ;R0 <- 0 NOT R0,R0 ;R0 <- xFFFF ST R0,Array ;M[Array] <- xFFFF LEA R5,Array

	``` LEA R6,=#100 ;R6 <= #100 STR R0,R5,#1 ;M[Array+1] <= xFFFF TRAP x25 ACC .EQU #1 ; ----- Scratch Space ----- Array .BLKW #3 .FILL x10 .END Begin ```
Expected Output	Error noting the operand is invalid and printing the line

Output:

```
<terminated> App [Java Application] C:\Users\user\AppData\Local\Programs\
Error: Invalid Operand R6,=#100 On line 15
Line:          LEA      R6,=#100        ;R6 <= #100
```

Test Case:	ORIG Pseudo Op without label

Input File	```
	;234567890123456789012345689 0
	; Example Program
	.ORIG x30B0
	count .FILL #4
	Begin LD ACC,count ;R1 <- 4
	LEA R0,msg
	loop TRAP x22 ;print "hi! "
	ADD ACC,ACC,#-1 ;R1--
	BRP loop
	``

	JMP Next msg .STRZ "hi! " Next AND R0,R0,x0 ;R0 <- 0 NOT R0,R0 ;R0 <- xFFFF ST R0,Array ;M[Array] <- xFFFF LEA R5,Array LD R6,=#100 ;R6 <= #100 STR R0,R5,#1 ;M[Array+1] <= xFFFF TRAP x25 ACC .EQU #1 ; ----- Scratch Space ----- Array .BLKW #3 .FILL x10 .END Begin
Expected Output	Error noting ORIG operand must have a label

Output:

```
<terminated> App [Java Application] C:\Users\user\AppData\Local\Programs\Eclip
Error: Error: EQU and ORIG must have a label  On line 3
Line:          .ORIG   x30B0
```

Test Case:	Smallest Valid Input Fine
Input File	Lab2EG .ORIG .END
Expected Output	HLab2EG00000000 E0000

Output:

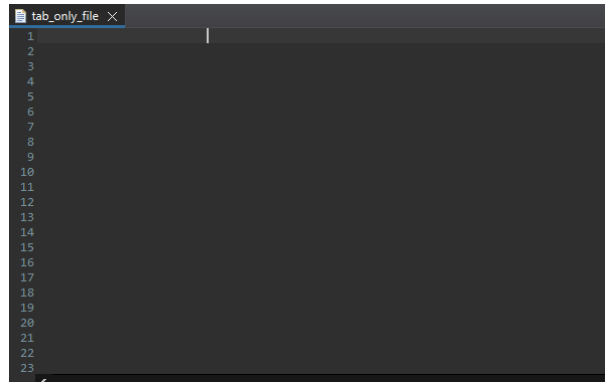
Object FIle

```
HLab2EG00000000
E0000
```

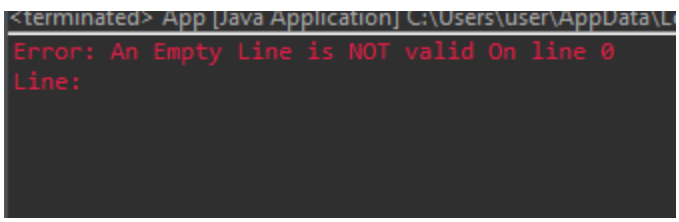
Listing File

```
                    (   0) Lab2EG        .ORIG
                    (   1)               .END

```

Test Case:	File with only tabs

Input File	
Expected Output	Error noting input lines cannot be Empty

Output:



Test Case:	Literal

Input File	;Literal world baby
	Litera .ORIG x2
	reg0 .EQU #0
	reg01 .EQU reg0
	reg1 .EQU #1
	reg11 .EQU reg1
	Begin LD reg0,=#1
	LD reg01,=#2
	LD reg1,=#3
	LD reg11,=#4
	.END Begin

Expected Output	4 blocks of memory allocated for LD, another 4 for literals

Output:

Object file:

```
HLitera00020008
T00022006
T00032007
T00042208
T00052209
T00060001
T00070002
T00080003
T00090004
E0002
```

Listing file:

```
                                (    1) Litera          .ORIG x2
                                (    2) reg0            .EQU  #0
                                (    3) reg01           .EQU  reg0
                                (    4) reg1            .EQU  #1
                                (    5) reg11           .EQU  reg1
(0002) 2006 0010000000000110 (    6) Begin           LD    reg0,=#1
(0003) 2007 0010000000000111 (    7)                 LD    reg01,=#2
(0004) 2208 0010001000001000 (    8)                 LD    reg1,=#3
(0005) 2209 0010001000001001 (    9)                 LD    reg11,=#4
                                (   10)                 .END  Begin
(0006) 0001 0000000000000001 ( lit)
(0007) 0002 0000000000000010 ( lit)
(0008) 0003 0000000000000011 ( lit)
(0009) 0004 0000000000000100 ( lit)
```

Test Case:	Relocatable

Input File	`; Relative Program`
	`relPrg  .ORIG`
	`count  .EQU  x4`
	`Begin  ADD  count,count,count`
	`  AND  R1,R2,x1F`
	`Beans1  LD  R0,=x37 ;literal relocatable??`

	`Maiden .FILL Beans1 ;16 bit M record`
	`BRN Maiden ;just a regular old jump`
	`.STRZ "donezo!"`
	`.END Begin`

<u>Expected Output</u>	All the address operations have 9-bit relocatable modification records, and .FILL has 16-bit modification record

Output:

Object file:

```
HrelPrg0000000E
T00001924
T000152BF
T0002200DM9
T00030002M16
T00040803M9
T00050064
T0006006F
T0007006E
T00080065
T0009007A
T000A006F
T000B0021
T000C0000
T000D0037
E0000
```

Listing file:

```
                          (   1) relPrg          .ORIG
                          (   2) count           .EQU  x4
(0000) 1924 0001100100100100 (   3) Begin         ADD
count,count,count
(0001) 52BF 0101001010111111 (   4)               AND   R1,R2,x1F
(0002) 200D 0010000000001101 (   5) Beans1        LD    R0,=x37
```

```
(0003) 0002 0000000000000010 (   6) Maiden          .FILL Beans1
(0004) 0803 0000100000000011 (   7)                 BRN   Maiden
(0005) 0064 0000000001100100 (   8)                 .STRZ "donezo!"
(0006) 006F 0000000001101111 (   8)
(0007) 006E 0000000001101110 (   8)
(0008) 0065 0000000001100101 (   8)
(0009) 007A 0000000001111010 (   8)
(000A) 006F 0000000001101111 (   8)
(000B) 0021 0000000000100001 (   8)
(000C) 0000 0000000000000000 (   8)
                             (   9)                 .END  Begin
(000D) 0037 0000000000110111 ( lit)
```

Unit Level Testing

Validator

Method under test: **validateAndSplitLine()**

Test	Purpose	Input	Expectated Result
validateAndSplitLineSimple()	Normal use-case for the method.	ACC .EQU #1	{ACC, .EQU, #1}

Method under test: **validate()**

Test	Purpose	Input	Expectated Result
validateTestComment()	Validating a line that is just a	;2345678901234567890123456890	No exception

	comment, normal use-case.		
validateTestOrig()	Testing normal .ORIG usage	Lab2EG .ORIG	No exception
validateTestOrigAbsolute()	Testing normal .ORIG usage with provided address.	Lab2EG .ORIG x30B0	No exception
validateTestEnd()	Testing normal .END usage	.END Begin	No exception
validateTestFill()	Testing normal FILL usage	.FILL x10	No exception
validateImproperLine()	Try to validate an improper line.	2345678901234567890123456890	Exception with message: “LINE MUST HAVE PSEUDO OP OR MACHINE OP”
validateImproperSize()	Try to validate a line that is too short.	.FILL x10	TooShortException
validateTestCharInColumn	Try to validate a	x .FILL x10	IllegalArgumentException

7()	line with an alphanumeric character where there should be spaces.		
validateTestCharInColumn8()	Try to validate a line with an alphanumeric character where there should be spaces.	x.FILL x10	IllegalArgumentException

Method under test: **getLineType()**

Test	Purpose	Input	Expectated Result
lineTypeTestComment()	Validates that getLineType() correctly identifies a normal comment line.	;2345678901234567890123456890	Validate.OpType.COMMENT
lineTypeTestEqu()	Validates that getLineType() correctly identifies a normal EQU	ACC .EQU #1	Validate.OpType.EQU

	line.		
invalidEqu()	Validates that getLineType() correctly identifies an EQU line without an arg.	ACC .EQU	Validate.OpType.EQU

Pass One

incLoc()

Test Case 1:	Test with .STRZ instruction
Test Scenario:	LocCount = 0x0 STRZ operand = “hi! “
Expected result:	Since .STRZ increments the location counter by the length of the operand, the location counter is incremented by 5

Test Case 2:	Test with .BLKW instruction
Test Scenario:	LocCount = 0x0 BLKW operand = #3
Expected result:	Since .BLKW increments the location counter by the value of the operand, the location counter is incremented by 3

Test Case 3:	Test with .EQU instruction
Test Scenario:	LocCount = 0x0
Expected result:	Since .EQU does not increment the location counter, the location counter remains at 0

Test Case 4:	Test with other types of instruction
Test Scenario:	LocCount = 0x0
Expected result:	Since any instruction other than .STRZ, .BLKW, .EQU, .ORIG, and .END increments the instruction by 1, the location counter here is incremented by 1

Test Case 5:	Test with location counter wrapping
Test Scenario:	LocCount = 0xFFFE STRZ operand = “hi! “
Expected result:	Since our assembler implements the instruction counter wrapping when hitting the max value of 0xFFFF, with .STRZ with this operand incrementing the location counter by 5, the location counter wraps around and ends at 0x4

Test Case 6:	Test with multiple, different types of instructions
Test Scenario:	LocCount = 0x0 STRZ operand = “hi! “ BLKW operand = #3
Expected result:	With the STRZ instruction incrementing the counter by 5, the BLKW instruction incrementing the counter by 3, and the EQU instruction incrementing the counter by 0, the new location counter ends up being 0x8

getEQUVal()

Test Case 1:	The operand of the EQU instruction is an absolute value

Test Scenario:	EQU operand = x25
Expected result:	The symbol array that gets filled comes out to [“x25”, “A”]

Test Case 2:	The operand of the EQU instruction is an absolute symbol
Test Scenario:	EQU operand = sym2 Sym2 value = x25
Expected result:	The symbol array that gets filled comes out to [“x25”, “A”]

Test Case 3:	The operand of the EQU instruction is a relative symbol
Test Scenario:	EQU operand = sym sym operand = “x2”
Expected result:	The symbol array that gets filled comes out to [“x2”, “R”]

fillSymArr()

Test Case 1:	Fill the symbol array with a relative symbol
Test Scenario:	Symbol = “sym” Instruction = TRAP Operand = x25 LocCount = 0x0
Expected result:	Symbol array should be filled with [“x0”, “R”] and the location counter should not be incremented yet

Test Case 2:	Fill the symbol array with an absolute value as the operand

Test Scenario:	Symbol = “sym” Instruction = .EQU Operand = x25 LocCount = 0x0
Expected result:	Symbol array should be filled with [“x25”, “A”] and the location counter should not be incremented yet

Test Case 3:	Fill the symbol array with an absolute symbol as the operand
Test Scenario:	Symbol = “sym3” Instruction = .EQU Operand = sym2 LocCount = 0x0
Expected result:	Symbol array should be filled with [“x25”, “A”], since sym2 has a value of x25 and is absolute, and the location counter should not be incremented yet

Test Case 4:	Fill the symbol array with a relative symbol as the operand
Test Scenario:	Symbol = “sym3” Instruction = .EQU Operand = sym2 LocCount = 0x3
Expected result:	Symbol array should be filled with [“x2”, “R”], since sym2 is a relative symbol with a value of x2, and the location counter should not be incremented yet

fillLitTable()

Test Case 1:	Fill the literal table with a single literal
Test Scenario:	LitArray = {“=x100”} LocCount = 0x0

Expected result:	LitTable is filled with {“=x100”, 0x0} Location counter = 0x1

Test Case 2:	Fill the literal table with multiple literals
Test Scenario:	LitArray = {“=x100”, “=x25”, “=x1”} LocCount = 0x0
Expected result:	LitTable is filled with 3 sets of key/values: {“=x100”, 0x0}, {“=x25”, 0x1}, {“=x1”, 0x2} Location counter = 0x3

Test Case 3:	Fills the literal table with 0 literals
Test Scenario:	LitArray = {} LocCount = 0x0
Expected result:	LitTable is still empty after the run Location counter = 0x0

fillSymTable()

Test Case 1:	Fills the symbol table with a single value
Test Scenario:	List = {{“Lab2aa”, “.ORIG”, “x3000”}, {“Begin”, “LD”, “ACC, count”}, {“”, “.END”, “Begin”}} LocCount = 0x0
Expected result:	There should only be one symbol in the table after the method runs, with the key “Begin”, and a symArray of [“x0”, “R”] Location counter = 0x1

Test Case 2:	Fills the symbol table with multiple values
Test Scenario:	List = {{“Lab2aa”, “.ORIG”, “x3000”}, {“Begin”, “LD”, “ACC, count”}, {“”, “LEA”, “R0, msg”}, {“loop”, “TRAP”, “x22”}, {“”,

	“ADD”, “ACC,ACC,#-1”}, {“”, “.END”, “Begin”}} LocCount = 0x0
Expected result:	The symbol table will have 2 sets of values. The first has a key of “Begin” and a symArray of [“x0”, “R”], and the second having a key of “loop” and a symArray of [“x2”, “R”] Location counter = 0x4

Test Case 3:	Fills the symbol table with no values
Test Scenario:	List = {} LocCount = 0x0
Expected result:	The symbol table will be empty, and the location counter will still be 0x0

Pass Two

Method under testing: registers

Test	Purpose	Input	Expected
invalidRegister()	When a register cannot be parsed	List = {{“”, “ADD”, “R11379115623,R1,#1”, “0”}}	"Line 0: Invalid register value"
invalidRegisterRange()	When the register range is not the desired one	List = {{“”, “ADD”, “R11,R1,#1”, “0”}}	"Line 0: Register value not within range (decimal and hex: [0 - 7])"
invalidRegSymNotVal()	When the register is a symbol not in the symbol table	List = {{“”, “ADD”, “not_sym,R1,#1”, “0”}}	"Line 0: Symbol \"not_sym\" is not found in the symbol table"
invalidRegSymNotAbs()	When the register symbol is not absolute	List = {{“”, “ADD”, “Sym2,R1,#1”, “0”}}	"Line 0: Symbol \"Sym2\" is not an absolute symbol”

invalidRegSym()	When the register symbol value cannot be parsed	List = {{“”, “ADD”, “Sym4,R1,#1”, “0”}}	"Line 0: Invalid register value"
invalidRegSymNotRange()	When the register symbol is not within the desired range	List = {{“”, “ADD”, “Sym3,R1,#1”, “0”}}	"Line 0: Register value not within range (decimal and hex: [0 - 7]

Method under testing: immediate

Test	Purpose	Input	Expected
invalidImmediate()	When a immediate cannot be parsed	List = {{“”, “ADD”, “R1,R1,xFFFFFFF”, “0”}}	"Line 0: Invalid immediate value"
invalidImmediateRange()	When the immediate range is not the desired one	List = {{“”, “ADD”, “R1,R1,xFF”, “0”}}	"Line 0: Imm5 value is not within the specified range [#-16 - #15] or [0x0 - 0x1F]"
invalidImmSymNotVal()	When the immediate is a symbol not in the symbol table	List = {{“”, “ADD”, “R1,R1,not_sym”, “0”}}	"Line 0: Symbol \"not_sym\" is not found in the symbol table"
invalidImmSymNotAbs()	When the immediate symbol is not absolute	List = {{“”, “ADD”, “R1,R1,Sym2”, “0”}}	"Line 0: Symbol \"Sym2\" is not an absolute symbol"
invalidImmSym()	When the immediate symbol value cannot be parsed	List = {{“”, “ADD”, “R1,R1,Sym4”, “0”}}	"Line 0: Invalid immediate value"
invalidImmSymNotRange()	When the immediate symbol is not within the desired range	List = {{“”, “ADD”, “R1,R1,Sym3”, “0”}}	"Line 0: Imm5 value is not within the specified range [#-16 - #15] or [0x0 - 0x1F]"

Method under testing: index

Test	Purpose	Input	Expected
invalidIndex()	When a index cannot be parsed	List = {{“”, “LDR”, “R1,R0,xFFFFFFF”, “0”}}	"Line 0: Invalid index6 value"
invalidIndexRange()	When the index range is not the desired one	List = {{“”, “LDR”, “R1,R0,xFF”, “0”}}	"Line 0: Index6 value is not within the specified range [#0 - #63] or [0x0 - 0x3F]"
invalidIndSymNotVal()	When the index is a symbol not in the symbol table	List = {{“”, “LDR”, “R1,R0,not_sym”, “0”}}	"Line 0: Symbol \"not_sym\" is not found in the symbol table"
invalidIndSymNotAbs()	When the index symbol is not absolute	List = {{“”, “LDR”, “R1,R0,Sym2”, “0”}}	"Line 0: Symbol \"Sym2\" is not an absolute symbol"
invalidIndSym()	When the index symbol value cannot be parsed	List = {{“”, “LDR”, “R1,R0,Sym4”, “0”}}	"Line 0: Invalid index6 value"
invalidIndSymNotRange()	When the index symbol is not within the desired range	List = {{“”, “LDR”, “R1,R0,Sym3”, “0”}}	"Line 0: Index6 value is not within the specified range [#0 - #63] or [0x0 - 0x3F]"

Method under testing: trap vector

Test	Purpose	Input	Expected
invalidTrap()	When a trap vector cannot be parsed	List = {{“”, “TRAP”, “xFFFFFFF”, “0”}}	"Line 0: Invalid trapvect8 value"
invalidTrapRange()	When the trap vector range is not the desired one	List = {{“”, “TRAP”, “xFFF”, “0”}}	"Line 0: Trapvect8 value is not within the specified range [#0 - #255] or [0x0 - 0xFF]"
invalidTrapSymNotVal()	When the trap vector is a symbol not in the symbol table	List = {{“”, “TRAP”, “not_sym”, “0”}}	"Line 0: Symbol \"not_sym\" is not found in the symbol table"

invalidTrapSymNotAbs()	When the trap vector symbol is not absolute	List = {{“”, “TRAP”, “Sym2”, “0”}}	"Line 0: Symbol \"Sym2\" is not an absolute symbol"
invalidTrapSym()	When the trap vector symbol value cannot be parsed	List = {{“”, “TRAP”, “Sym4”, “0”}}	"Line 0: Invalid trapvect8 value"
invalidTrapSymNotRange()	When the trap vector symbol is not within the desired range	List = {{“”, “TRAP”, “Sym3”, “0”}}	"Line 0: Trapvect8 value is not within the specified range [#0 - #255] or [0x0 - 0xFF]"

Method under testing: addresses

Test	Purpose	Input	Expected
invalidAddress()	When a address cannot be parsed	List = {{“”, “BRP”, “xFFFFFF”, “0”}}	“Line 0: Invalid address value”
invalidAddressRange()	When the address range is not the desired one	List = {{“”, “LD”, “R1,#-12”, “0”}}	"Line 0: Address value is not within the specified range [#0 - #65535] or [0x0 - 0xFFFF]"
invalidAddressPageRange()	When the address value is not within the same page range as the location counter	List = {{“”, “ST”, “R1,x3000”, “0”}}	"Line 0: Address value is not within the same page number as PC (PC at page: #0, Defined Address at page: #24)"
invalidAddressNoSym()	When the address is a symbol not in the symbol table	List = {{“”, “BRNZ”, “not_sym”, “0”}}	"Line 0: Symbol \"not_sym\" is not found in the symbol table"
invalidAddressSymValue()	When the address symbol value cannot be parsed	List = {{“”, “JMP”, “Sym4”, “0”}}	"Line 0: Invalid address value"
invalidAddressSymNotRange()	When the address symbol value is not	List = {{“”, “JMP”, “Sym5”, “0”}}	"Line 0: Address value is not within

	within the desired range		the specified range [#0 - #65535] or [0x0 - 0xFFFF]"
invalidAddressSymAbsNotPageRange()	When the address of an absolute symbol value is not within the same page range as the location counter	List = {{“”, “LEA”, “R0,Sym3”, “0”}}	"Line 0: Address value is not within the same page number as PC (PC at page: #0, Defined Address at page: #7)”
invalidAddressSymRelNotPageRange()	When the address of an relative symbol value is not within the same page range as the location counter	List = {{“”, “LEA”, “R0,Sym2”, “0”}}	"Line 0: Address value is not within the same page number as PC (PC at page: #0, Defined Address at page: #24)”
invalidAddressLiteralNotPageRange()	When the address of an literal address is not within the same page range as the location counter	List = {{“”, “.ORIG”, “x7F99”, “0”}, {“”,”LD”,”R0,=#98”, ”1”,{“”,”.END”,”x7F99”,””}}}	"Line 1: Address value is not within the same page number as PC (PC at page: #63, Defined Address at page: #64)"

String Parser

Method under testing: canParseInt(), isLiteral()

Test	Purpose	Input	Expected
decMin()	Testing parse int when given the lowest valid decimal value	“#-32768”	true
decReg()	Testing parse int when given a normal valid decimal value	“#0”	true
decMax()	Testing parse int when given the	“#65535”	true

	highest valid decimal value		
hexMin()	Testing parse int when given the lowest valid hex value	“x0”	true
hexReg()	Testing parse int when given a normal valid hex value	“x5”	true
hexMax()	Testing parse int when given the highest valid hex value	“xFFFF”	true
invalidEmpty()	Testing parse int when given an invalid entry (empty)	“”	false
invalidStart()	Testing parse int when given an invalid entry (no starting characters)	“pnemono”	false
invalidDec()	Testing parse int when given an invalid entry (unreadable value)	“#4ff2”	false
invalidHex()	Testing parse int when given an invalid entry (unreadable value)	“xrref”	false
invalidDecOBO()	Testing parse int when given an invalid entry (decimal out of range)	“#600000”	false
invalidHexOBO()	Testing parse int when given an invalid entry (hex out of range)	“xFFFFFF”	false

Method under testing: parseImmediate(), parseIndex(), parseTrapVect(), parseAddress(),

Test	Purpose	Input	Expected
decMin()	Testing parse int when given the lowest valid decimal value	minimum_range_value	minimum_range_value
decReg()	Testing parse int when given a normal valid decimal value	“#0”	0
decMax()	Testing parse int when given the highest valid decimal value	maximum_range_value	maximum_range_value
hexMin()	Testing parse int when given the lowest valid hex value	minimum_range_value	minimum_range_value
hexReg()	Testing parse int when given a normal valid hex value	“xF”	0xF
hexMax()	Testing parse int when given the highest valid hex value	maximum_range_value	maximum_range_value
invalidDecNeg()	Testing with an invalid input (lower than lowest decimal value)	“#-20”	-0xFFFF
invalidDecPos()	Testing with an invalid input (greater than largest decimal value)	“#600000”	-0xFFFF
invalidHexNeg()	Testing with an invalid input (lower than lowest hex value)	“x-1”	-0xFFFF
invalidHexPos()	Testing with an	“xFFFFF’	-0xFFFF

	invalid input (greater than largest hex value)		

Machine Ops Table

Method(s) tested per case: containsOp(), getSize(), getOpcode(), getFormat(), getInstructionName()

Test	Purpose	Input	Expected
testADD()	Testing if ADD gets loaded properly	“ADD”	True, 1, 0b0001, NONE, ADD
testAND()	Testing if AND gets loaded properly	“AND”	True, 1, 0b0101, NONE, AND
testBR()	Testing if BR gets loaded properly	“BR”	True, 1, 0b0000, PGOFFSET, BR
testBRN()	Testing if xBRN gets loaded properly	“BRN”	True, 1, 0b0000, PGOFFSET, BRN
testBRZ()	Testing if BRZ gets loaded properly	“BRZ”	True, 1, 0b0000, PGOFFSET, BRZ
testBRP()	Testing if BRP gets loaded properly	“BRP”	True, 1, 0b0000, PGOFFSET, BRP
testBRNZ()	Testing if BRNZ gets loaded properly	“BRNZ”	True, 1, 0b0000, PGOFFSET, BRNZ
testBRNP()	Testing if BRNP gets loaded properly	“BRNP”	True, 1, 0b0000, PGOFFSET, BRNP
testBRZP()	Testing if BRZP gets loaded properly	“BRZP”	True, 1, 0b0000, PGOFFSET, BRZP
testBRNZP()	Testing if BRNZP gets loaded properly	“BRNZP”	True, 1, 0b0000, PGOFFSET, BRNZP
testDBUG()	Testing if DBUG gets loaded properly	“DBUG”	True, 1, 0b1000, NONE, DBUG
testJSR()	Testing if JSR gets	“JSR”	True, 1, 0b0100,

	loaded properly		PGOFFSET, JSR
testJMP()	Testing if JMP gets loaded properly	“JMP”	True, 1, 0b0100, PGOFFSET, JMP
testJSRR()	Testing if JSRR gets loaded properly	“JSRR”	True, 1, 0b1100, INDEX, JSRR
testJMPR()	Testing if JMPR gets loaded properly	“JMPR”	True, 1, 0b1100, INDEX, JMPR
testLD()	Testing if LD gets loaded properly	“LD”	True, 1, 0b0010, PGOFFSET, LD
testLDI()	Testing if LDI gets loaded properly	“LDI”	True, 1, 0b1010, PGOFFSET, LDI
testLDR()	Testing if LDR gets loaded properly	“LDR”	True, 1, 0b0110, INDEX, LDR
testLEA()	Testing if LEA gets loaded properly	“LEA”	True, 1, 0b1110, PGOFFSET, LEA
testNOT()	Testing if NOT gets loaded properly	“NOT”	True, 1, 0b1001, NONE, NOT
testRET()	Testing if RET gets loaded properly	“RET”	True, 1, 0b1101, NONE, RET
testST()	Testing if ST gets loaded properly	“ST”	True, 1, 0b0011, PGOFFSET, ST
testSTI()	Testing if STI gets loaded properly	“STI”	True, 1, 0b1101, PGOFFSET, STI
testSTR()	Testing if STR gets loaded properly	“STR”	True, 1, 0b0111, INDEX, STR
testTRAP()	Testing if TRAP gets loaded properly	“TRAP”	True, 1, 0b1111, NONE, TRAP
testLower()	Testing if invalid operations are returned	“trap”	False, -1, -1, null, null
testMixed()	Testing if invalid operations are returned	“lDi”	False, -1, -1, null, null

testUnknown()	Testing if invalid operations are returned	“not_valid”	False, -1, -1, null, null

Pseudo Ops Table

Method(s) tested per case: containsOp(), getLength(), getFormat(), getInstructionName()

Test	Purpose	Input	Expected
testORIG()	Testing if .ORIG gets loaded correctly	".ORIG"	True, 0, DEFINITE, ORIG
testEND()	Testing if .END gets loaded correctly	“.END”	True, 0, DEFINITE, END
testEQU()	Testing if .EQU gets loaded correctly	“.EQU”	True, 0, DEFINITE, EQU
testFILL()	Testing if .FILL gets loaded correctly	“.FILL”	True, 0, DEFINITE, FILL
testSTRZ()	Testing if .STRZ gets loaded correctly	“.STRZ”	True, 0, VARIABLE, STRZ
testBLKW()	Testing if .BLKW gets loaded correctly	“.BLKW”	True, 0, VARIABLE, BLKW
testLower()	Invalid input for when lowercase letters are given	“.orIG”	False, -1, null, null
testMixed()	Invalid input for when mixed-case letters are given	“.orig”	False, -1, null, null
testUnknown()	Invalid input for when unknown instructions are given	“.JANK”	False, -1, null, null
testblockLength_decMin()	Testing block length when given the lowest valid decimal value	“#1”	1
testblockLength_decReg()	Testing block length	“#50”	50

	when given a normal valid decimal value		
testblockLength_decMax()	Testing block length when given the highest valid decimal value	“#65535”	65535
testblockLength_hexMin()	Testing block length when given the lowest valid hex value	“x1”	0x1
testblockLength_hexReg()	Testing block length when given a normal valid hex value	“x50”	0x50
testblockLength_hexMax()	Testing block length when given the highest valid hex value	“xFFFF”	0xFFFF
testblockLength_invalidEmpty()	Testing block length when given an invalid entry (empty)	“”	-1
testblockLength_invalidNoParse()	Testing block length when given an invalid entry (no starting characters)	“25”	-1
testblockLength_invalidDecVal()	Testing block length when given an invalid entry (unreadable value)	“#25F”	-1
testblockLength_invalidHexVal()	Testing block length when given an invalid entry (unreadable value)	“x25R”	-1
testblockLength_invalidDecNegOBO()	Testing block length when given an invalid entry (decimal out of range)	“#-25”	-1
testblockLength_invalidDecPosOBO()	Testing block length	“#600000”	-1

	when given an invalid entry (decimal out of range)		
testblockLength_invalidHexNegOBO()	Testing block length when given an invalid entry (hex out of range)	“x-25”	-1
testblockLength_invalidHexPosOBO()	Testing block length when given an invalid entry (hex out of range)	“x600000”	-1

Modified Appendix

System Testing

Test 1

Prompt	Result
Purpose of the test:	Testing if all the BRx instructions append modification records to the text records if the program is relocatable.
Input file:	```;Example prog Prog .ORIG .ENT start .EXT test ; start BR test BRN test BRZ test BRP test BRNZ test BRNP test BRZP test BRNZP test .END start``` Expected output (explained):	All the text records contain an “X9” modification record with the external symbol “test”. An N-record should be created for the “Start” entry symbol.
Actual output (object file):	```HProg 00000008 Nstart=0 T00000000X9test T00010800X9test T00020400X9test T00030200X9test T00040C00X9test T00050A00X9test T00060600X9test T00070E00X9test E0000``` Expected output (explained):

Actual output (listing file):	

```
                                (   1) Prog            .ORIG
                                (   2)                 .ENT  start
                                (   3)                 .EXT  test
(0000) 0000 0000000000000000 (   5) start           BR    test
(0001) 0800 0000100000000000 (   6)                 BRN   test
(0002) 0400 0000010000000000 (   7)                 BRZ   test
(0003) 0200 0000001000000000 (   8)                 BRP   test
(0004) 0C00 0000110000000000 (   9)                 BRNZ  test
(0005) 0A00 0000101000000000 (  10)                 BRNP  test
(0006) 0600 0000011000000000 (  11)                 BRZP  test
(0007) 0E00 0000111000000000 (  12)                 BRNZP test
                                (  13)                 .END  start
```

Test 2

Prompt	Result
Purpose of the test:	Testing if the JMP and JSR instructions append modification records to the text records if the program is relocatable.
Input file:	

```
;Example prog
Prog      .ORIG
          .ENT    start
          .EXT    test
;
start     JMP     test
          JSR     test
          .END    start
```

Expected output (explained):	All the text records contain an “X9” modification record with the external symbol “test”. An N-record should be created for the “Start” entry symbol.
Actual output (object file):	

```
HProg  00000002
Nstart=0
T00004000X9test
T00014800X9test
E0000
```

Actual output (listing file):	

```
                                (   1) Prog            .ORIG
                                (   2)                 .ENT  start
                                (   3)                 .EXT  test
(0000) 4000 0100000000000000 (   5) start           JMP   test
(0001) 4800 0100100000000000 (   6)                 JSR   test
                                (   7)                 .END  start
```

Test 3

Prompt	Result
Purpose of the test:	Testing if LOAD instructions append modification records to the text records if the program is relocatable.

Input file:	```;Example prog
	Prog .ORIG
	.ENT start
	.EXT test
	;
	start LD R0,test
	LDI R1,test
	LEA R2,test
	.END start```

Expected output (explained):	All the text records contain an “X9” modification record with the external symbol “test”. An N-record should be created for the “Start” entry symbol.
Actual output (object file):	```HProg 00000003 Nstart=0 T00002000X9test T0001A200X9test T0002E400X9test E0000```
Actual output (listing file):	```(1) Prog .ORIG (2) .ENT start (3) .EXT test (0000) 2000 0010000000000000 (5) start LD R0,test (0001) A200 1010001000000000 (6) LDI R1,test (0002) E400 1110010000000000 (7) LEA R2,test (8) .END start```

Test 4

Prompt	Result

Purpose of the test:	Testing if STORE instructions append modification records to the text records if the program is relocatable.
Input file:	```;Example prog Prog .ORIG .ENT start .EXT test ; start ST R0,test STI R1,test .END start``` (Prog .ORIG; .ENT start; .EXT test; start ST R0,test; STI R1,test; .END start)
Expected output (explained):	All the text records contain an “X9” modification record with the external symbol “test”. An N-record should be created for the “Start” entry symbol.
Actual output (object file):	```HProg 00000002 Nstart=0 T00003000X9test T0001B200X9test E0000``` (H record: Prog, 00000002; N record: start=0; T records: 00003000 X9test, 0001B200 X9test; E record: 0000)
Actual output (listing file):	```(1) Prog .ORIG (2) .ENT start (3) .EXT test (0000) 3000 0011000000000000 (5) start ST R0,test (0001) B200 1011001000000000 (6) STI R1,test (7) .END start``` (Listing: .ORIG at 3000; ST R0,test at 3000 = 0011000000000000; STI R1,test at B200 = 1011001000000000; .END start)

Test 5

Prompt	Result
Purpose of the test:	Testing if .FILL instructions append modification records to the text records if the program is relocatable.
Input file:	```;Example prog Prog .ORIG .ENT start .EXT test1,test2,test3 ;``` (Prog .ORIG; .ENT start; .EXT test1,test2,test3; ;)

	`start .FILL test1`
	`.FILL test2`
	`.FILL test3`
	`.END start`

Expected output (explained):	All the text records contain an “X16” modification record with the external symbol “test”. An N-record should be created for the “Start” entry symbol.

Actual output (object file):	`HProg 00000003`
	`Nstart=0`
	`T00000000X16test1`
	`T00010000X16test2`
	`T00020000X16test3`
	`E0000`

Actual output (listing file):	`( 1) Prog .ORIG`
	`( 2) .ENT start`
	`( 3) .EXT test1,test2,test3`
	`(0000) 0000 0000000000000000 ( 5) start .FILL test1`
	`(0001) 0000 0000000000000000 ( 6) .FILL test2`
	`(0002) 0000 0000000000000000 ( 7) .FILL test3`
	`( 8) .END start`

Test 6

Prompt	Result
Purpose of the test:	Routine test for testing a combination of instructions with entry and external records.

Input file:	`;234567890123456789012345678901234567890`
	`;label___opppp___operandsandcomments...`
	`;`
	`Main .ORIG`
	`.EXT Displ,V`
	`.ENT Start`
	`.EXT X`
	`;`
	`Start JSR Displ ;Display 6..0`
	`LD R1,V ;r1 <- M[V]`

	``` ST R1,X ;M[X] <- r1 JSR Displ ;Display 2..0 TRAP x25 ;halt .END Start ```
Expected output (explained):	Modifications records on the text records of all the instructions (with the external symbol) but TRAP.
Actual output (object file):	``` HMain 00000005 NStart=0 T00004800X9Displ T00012200X9V T00023200X9X T00034800X9Displ T0004F025 E0000 ```
Actual output (listing file):	``` (3) Main .ORIG (4) .EXT Displ,V (5) .ENT Start (6) .EXT X (0000) 4800 0100100000000000 (8) Start JSR Displ (0001) 2200 0010001000000000 (9) LD R1,V (0002) 3200 0011001000000000 (10) ST R1,X (0003) 4800 0100100000000000 (11) JSR Displ (0004) F025 1111000000100101 (12) TRAP x25 (13) .END Start ```

Test 7

Prompt	Result
Purpose of the test:	When an external symbol is used, but is not imported through the .EXT op

Input file:	```
	; Poggers err?
	Main .ORIG
	.ENT Start
	;
	Start JSR Displ ;error
	LD R1,V ;error
	ST R1,X ;error
	``

	`JSR Displ ;error`
	`TRAP x25`
	`.END Start`

Expected output (explained):	An error is thrown saying the symbol does not exist.

Actual output (object file):	`Error: Line 4: Symbol "Displ" is not found in the symbol table`
	`lab2.Exceptions$Pass2Exception: Line 4: Symbol "Displ" is not found in the symbol table`
	`at Passes.Pass2.isRelativeSymbol(Pass2.java:1677)`
	`at Passes.Pass2.jsrOp(Pass2.java:844)`
	`at Passes.Pass2.assembleMachineOpLine(Pass2.java:333)`
	`at Passes.Pass2.parseInput(Pass2.java:159)`
	`at lab2.App.main(App.java:105)`

Actual output (listing file):	None

Test 8

Prompt	Result
Purpose of the test:	Declaring a symbol in the .ENT op, but not defining it.

Input file:	`; Poggers err?`
	`Main .ORIG`
	`.EXT Displ`
	`.ENT Start ; error`
	`.EXT X`
	`;`
	`Start1 JSR Displ`
	`LD R1,V`
	`ST R1,X`
	`JSR Displ`
	`TRAP x25`
	`.END Start1`

Expected output (explained):	An error is thrown saying the symbol is not defined.

Actual output (object file):	`Error: Line 3: Symbol "Start" is not found in the symbol table`
	`lab2.Exceptions$Pass2Exception: Line 3: Symbol "Start" is not found in the symbol table`
	`  at Passes.Pass2.isRelativeSymbol(Pass2.java:1677)`
	`  at Passes.Pass2.entOp(Pass2.java:1404)`
	`  at Passes.Pass2.assemblePseudoOpLine(Pass2.java:414)`
	`  at Passes.Pass2.parseInput(Pass2.java:154)`
	`  at lab2.App.main(App.java:105)`

Actual output (listing file):	None

Test 9

Prompt	Result
Purpose of the test:	Trying to define a symbol in the .EXT op when it is defined in the file locally (is in the symbol table).

Input file:	`; Poggers err?`
	`Main  .ORIG`
	`  .EXT  Displ`
	`  .ENT  Start`
	`  .EXT  X`
	`;`
	`Start  JSR  Displ`
	`Displ  LD  R1,V  ;error`
	`  ST  R1,X`
	`  JSR  Displ`
	`  TRAP  x25`
	`  .END  Start`

Expected output (explained):	An error is thrown saying the symbol already exists.

Actual output (object file):	`Error: Line 2: Symbol "Displ" is already defined in the symbol table (cannot be used as an external symbol)`
	`lab2.Exceptions$Pass2Exception: Line 2: Symbol "Displ" is already defined in the symbol table (cannot be used as an external symbol)`
	`  at Passes.Pass2.extOp(Pass2.java:1438)`
	`  at Passes.Pass2.assemblePseudoOpLine(Pass2.java:417)`
	`  at Passes.Pass2.parseInput(Pass2.java:154)`
	`  at lab2.App.main(App.java:105)`

Actual output (listing file):	None

Test 10

Prompt	Result
Purpose of the test:	Trying to use .EXT symbols in the place where only absolute symbols are allowed.

Input file:	`; Poggers err?`
	`Main  .ORIG`
	`  .EXT  Displ`
	`  .ENT  Start`
	`  .EXT  X`
	`;`
	`Start  JSR  Displ`
	`  LD  X,V ;error`
	`  ST  R1,X`
	`  JSR  Displ`
	`  TRAP  x25`
	`  .END  Start`

Expected output (explained):	An error is thrown saying relative symbols cannot be used in place of absolute symbols.
Actual output (object file):	`Error: Line 7: External symbol "X" cannot be`

	used as an absolute symbol
	lab2.Exceptions$Pass2Exception: Line 7: External symbol "X" cannot be used as an absolute symbol
	  at Passes.Pass2.absoluteSymbolTable(Pass2.java:1650)
	  at Passes.Pass2.ldOp(Pass2.java:978)
	  at Passes.Pass2.assembleMachineOpLine(Pass2.java:345)
	  at Passes.Pass2.parseInput(Pass2.java:159)
	  at lab2.App.main(App.java:105)

Actual output (listing file):	None

Test 11

Prompt	Result
Purpose of the test:	Routine test to test all the modifications records: X16 for external and internal, X9 for external and internal. Checking for forward referencing as well.

Input file:	;The regular prog
	;
	Prog .ORIG
	  .EXT ext1,ext2
	  .ENT ent3
	  .EXT ext3
	;
	ent3 BRNZP ext3
	;
	Where JSR ext1
	It .STRZ "this is life kid!"
	All .FILL ext2
	Began .FILL Where
	Was LD R0,Began

	<pre>When STI R1,ext2
	I .EQU ext2
	;
	TWas LEA R2,I
	Born TRAP x25
	.END Where</pre>

Expected output (explained):	For each instruction that allows for relative symbols, they produce X9[external_symbol_name] for external symbols or X9[segment_name] if using a local relative symbol and the same for **.**FILL ops but it is an X16 modifier. Forward reference symbols with external symbols should be correctly used.

Actual output (object file):	<pre>HProg 0000001A
	Nent3=0
	T00000E00X9ext3
	T00014800X9ext1
	T00020074
	T00030068
	T00040069
	T00050073
	T00060020
	T00070069
	T00080073
	T00090020
	T000A006C
	T000B0069
	T000C0066
	T000D0065
	T000E0020
	T000F006B
	T00100069
	T00110064
	T00120021
	T00130000
	T00140000X16ext2
	T00150001X16Prog
	T00162015X9Prog
	T0017B200X9ext2</pre>

	T0018E400X9ext2 T0019F025 E0001
Actual output (listing file):	(0000) 0E00 0000111000000000 (7) ent3 BRNZP ext3 (0001) 4800 0100100000000000 (9) Where JSR ext1 (0002) 0074 0000000001110100 (10) It .STRZ "this is life kid!" (0003) 0068 0000000001101000 (10) (0004) 0069 0000000001101001 (10) (0005) 0073 0000000001110011 (10) (0006) 0020 0000000000100000 (10) (0007) 0069 0000000001101001 (10) (0008) 0073 0000000001110011 (10) (0009) 0020 0000000000100000 (10) (000A) 006C 0000000001101100 (10) (000B) 0069 0000000001101001 (10) (000C) 0066 0000000001100110 (10) (000D) 0065 0000000001100101 (10) (000E) 0020 0000000000100000 (10) (000F) 006B 0000000001101011 (10) (0010) 0069 0000000001101001 (10) (0011) 0064 0000000001100100 (10) (0012) 0021 0000000000100001 (10) (0013) 0000 0000000000000000 (10) (0014) 0000 0000000000000000 (11) All .FILL ext2 (0015) 0001 0000000000000001 (12) Began .FILL Where (0016) 2015 0010000000010101 (13) Was LD R0,Began (0017) B200 1011001000000000 (14) When STI R1,ext2 (15) I .EQU ext2 (0018) E400 1110010000000000 (17) TWas LEA R2,I (0019) F025 1111000000100101 (18) Born TRAP x25 (19) .END Where