

LAB #2: Assembler Lab

DUE: Friday Nov 15th.

In this lab you are to write an assembler for the abstract machine.

1 Input

The input to the assembler is a file containing an assembly language program. Each line of the file will have the following format:

Position	Meaning
1-6	Label, if any, left justified
7-9	Unused (i.e., white space)
10-14	Operation field
15-17	Unused (i.e., white space)
18-end of record	Operands and comments (comments begin with a semicolon (;))
Exception:	A semicolon (;) in the first record position indicates that the entire record is a comment.

Notes:

1. This (somewhat rigid) record structure is intended as a convenience for your parsing of program lines. It allows you to treat each line in the file as an array, with particular ranges of the array containing certain fields.
2. Despite the previous comment, you may *choose to* allow slightly different syntax for input programs, if you find that easier. For example, you may allow the fields to be separated by tabs rather than spaces. Please keep in mind, however, that your graders will prepare test cases following the guidelines above, so your assembler should accept this as valid input. Substantially changing the input format requirements will cost you time during your interactive grading.
3. You may also impose an upper limit on the length of input lines.
4. For the purposes of this assignment, you may assume a maximum of 100 symbols, 50 literals, and 200 source records in any given program. However, these constraints should be easy to change (be sure your programmer's guide gives instructions for changing them).

Labels

Labels may be up to 6 alphanumeric characters (e.g., they may not include blanks). The first character of a label must be alphabetic, but must not be an “R” or an “x”.

Alphabetic characters can be upper or lower case, and an upper case character should be treated differently from its corresponding lower case character.

Instructions

You may require instructions to be all uppercase. The following instructions, with their corresponding format, must be supported by your assembler:

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
DEBUG		DEBUG	
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

The corresponding machine code for these instructions is given in the handout describing the machine’s characteristics.

Operands and Comments

In the “operands and comments” field, you may prohibit the “operand” part from including blanks. Registers are indicated by their explicit name (such as R3). Constants are written either in hexadecimal notation (preceded by a lowercase “x”) or as decimal integers, which may be positive or negative (preceded by “#”).

An imm5 operand must be in the range #-16..#15, or x0..x1F. An addr operand must be in the range #0..#65535, or x0..xFFFF. Note that only the least significant 9 bits of this value are used

in the machine code encoding. An `index6` operand must be in the range `#0..#63`, or `x0..x3F`. A `trapvect8` operand must be in the range `#0..#255`, or `x0..xFF`.

Symbols can be used in place of any operand. The *only* place where a relocatable symbol can be used is in the final operand of the following instructions: `BR`, `JSR`, `JMP`, `LD`, `LDI`, `LEA`, `ST`, and `STI` (i.e., replacing an `addr` operand). Any other use of a symbol requires the symbol to be absolute.

Symbols can be used in place of an explicit register name (such as `R3`). In this case, the (absolute) symbol must have a value in the range 0 to 7. Symbols can also be used in place of immediate arguments (`imm5`, `index6`, and `trapvect8`). Again, in this case, the (absolute) symbol must have a value in the appropriate range. In the case of `imm5`, where there are two different ranges, depending on whether a decimal or hex number is used, the symbol value must be in the union of the two ranges (i.e., `#-16..x1F`).

When a symbol is used as the last argument for `ADD` or `AND` instructions, it is always interpreted as an `imm5` operand, rather than a source register. That is, the first form of these instructions is used, where bit 5 is set.

For the `LD` instruction, the `addr` operand can contain a literal. A literal is denoted by an “=”, followed by a constant (e.g., `=x2A`, `=#-1`). A literal causes the assembler to: generate a location for the literal (after the last programmer-defined location), place the value indicated by the operand in that location, and use the address of that location in the instruction. Literals are allowed only with the `LD` instruction. The value of a literal must be in the range `#-32768..#32767` for decimal constants, and the range `x0..xFFFF` for hex constants.

For every `addr` operand, its page number is checked against the page number of the PC when that instruction is executed (i.e., the location counter plus 1 of the instruction in question). If they are not the same, the assembler should flag this as a fatal error.

Pseudo Ops

Your assembler should handle the following pseudo ops:

- .ORIG** This must be the first non-comment record in the source program. The operand, if present, must be a hex number in the range `x0..xFFFF`. The operand indicates the absolute address at which the program is to be loaded. If the operand is absent, the program is relocatable. If the program is relocatable, it must fit within a single page of memory. The `.ORIG` statement also must have a label, which is the name of the segment.
- .END** Indicates the end of the input program. An optional operand (a hex integer in the range `x0..xFFFF` or a symbol) indicates the address at which execution is to begin. If no operand is present, execution begins at the first address in the segment.
- .EQU** Equates the symbol in the label field with the “value” of the operand field, essentially creating a constant within the assembly program. The operand field can be a previously defined symbol or a constant. The constant can be written either as a decimal integer (preceded with `#`) or a hex number (preceded with `x`).

.FILL Defines a one-word quantity whose contents is the value of the operand. The operand is either a symbol a (hex or decimal) constant. Decimal constants must be in the range #-32768..#32767, while hex constants must be in the range x0..xFFFF. If a symbol is used, its value must be in the union of these two ranges (ie #-32768..xFFFF). This value is placed by the assembler in the word of memory that the .FILL pseudo-op occupies. The assembler location counter is moved forward one word.

.STRZ Defines a block of words to hold the characters of the null-terminated string in the pseudo-ops operand field. The operand string is enclosed in quotation marks. The ASCII code for each character is stored in bits [7:0], while bits [15:8] are cleared. Since a null (x0000) is added at the end, a STRZ pseudo-op whose operand is “test” will occupy 5 words in memory.

.BLKW Sets up a block of storage. The number of words in the block must be at least one and at most xFFFF, as indicated by the constant or previously defined absolute symbol in the operand field. This command moves the assembler location counter forward the corresponding number of words. The block of storage is not initialized by this pseudo op.

The .ORIG and .EQU pseudo op instructions **require** labels. If the operand of .EQU or .BLKW is a symbol, that symbol must be defined earlier in the program. Labels are optional on the .FILL, .STRZ, and .BLKW pseudo ops. No label is allowed on .END pseudo ops. .FILL, .STRZ, and .BLKW require memory allocation while .ORIG, .EQU, and .END do not.

2 Output

Your assembler should have two primary outputs:

1. an object file (which will subsequently be the input file for the linker/loader you will write in Lab 3), and
2. a listing for the user

Your object file should provide all of the information needed by a linker loader to generate the input to the 560 machine simulator defined in Lab 1. It should include a header record, text records, an end record, as well as other record types as appropriate. That is, your object file should provide the memory contents associated with the instructions and data of the source program, along with information for the loader concerning the relocatability of the object file information and the size of the program’s address space. Don’t forget to deal with the location at which execution is to begin. For acceptance of this lab, the object file should be written to a file, information about which should be specified in your user’s guide. Remember that this file will be “consumed” by the program you write in the next lab.

The listing you output for the user should contain the source program and its assembly, in some suitable format. It does not need to include comments. Remember that this output is for human consumption and should be designed to be as useful as possible to programmers. A recommended format follows.

(Addr hex)	Contents hex	Contents binary	(line #)	Label	Instruction	Operands
------------	-----------------	--------------------	----------	-------	-------------	----------

Your assembler should print meaningful diagnostics if errors in assembly are encountered. It should be capable of detecting errors involving each of the following conditions: invalid operation, invalid label, invalid operand (symbol, literal, integer, register), undefined reference, and multiple definition of a symbol.

You must turn in a programmer's guide, user's guide, test plan with test results, meeting minutes, and a peer evaluation of the other members of your group. While a design review is not required for this lab, the grader and instructor are available during office hours and by appointment for consultation and informal design reviews. You should not begin coding until your design and pseudocode are done. It is much easier to change pseudocode than C++ or Java.

3 Breakdown of Lab 2 grade

User's guide: 20%
 Programmer's guide: 20%
 Coding: 10%
 Your testing: 15%
 Our testing: 20%
 Meeting Minutes: 5%
 Peer review: 10%

4 Sample Input

```

; 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

```

5 Sample Output

The program given above would result in an object file (header record, text records, etc.) being written.

```
HLab2EG30B00018  
T30B00004  
T30B122B0  
T30B2E0B7  
T30B3F022  
T30B4127F  
T30B502B3  
T30B640BC  
T30B70068  
T30B80069  
T30B90021  
T30BA0020  
T30BB0000  
T30BC5020  
T30BD903F  
T30BE30C3  
T30BFEAC3  
T30C02CC7  
T30C17141  
T30C2F025  
T30C60010  
T30C70064  
E30B1
```

In addition to this object file, a listing file should also be produced. The recommended format of this listing file would produce the following display.

```

( 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) 903F 1001000000111111 ( 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)

```

Please note that there are many appropriate variations on this format. For example, the above listing does not show comments, but you may elect to show these as well. This example also does not include relocation information, which you can include in various ways. The goal of the listing file format is to provide enough information for the user (or a tester, or the grader) to quickly inspect the assembler output and confirm whether it is correct.