ECE 206, Fall 2001: Lab 3

Data Movement Instructions

Learning Objectives

This lab will give you practice with a number of LC-2 programming constructs. In particular you will cover the following topics:

- Load/store usage
- Addressing modes
- Looping constructs

In order to practice using data movement instructions, you will write two simple encryption programs.

Background

Data Movement Instructions

Data movement instructions move information between memory and registers. The LC-2 supports three modes by which data can be moved between memory and registers. These are *direct*, *indirect*, and *base+offset* (also known as *relative*). Each mode is available for both loads and stores. Recall from lecture that a *load* operation moves data from a memory location to a register and a *store* moves data from a register to a memory location.

The LC-2 provides seven instructions that move data. There are three load operations (LD, LDR, and LDI), and three store operations (ST, STR, STI). The prefixes LD and ST indicate load and store, respectively, while no suffix indicates the direct addressing mode, I indicates indirect, and R indicates base+offset.

The LEA instruction is also considered a data movement instruction, even though it does not involve moving data between memory and a register. Instead, it calculates an address relative to the current page (as in direct mode) and stores the address in a register. (This is "immediate" mode, since the instruction's operand – the address – is specified by the instruction itself.)

Direct

Direct mode (LD and ST) is useful for referencing memory locations on the same page as the load instruction. (An LC-2 page is a 512-word region of memory.) Since the memory address is specified in the instruction, the location of the data being referenced will be the same each time the instruction is executed. In other words, the address is not data-dependent.

Direct mode is useful when a constant value is needed to initialize a register, but the value is too large to be specified with immediate mode. For example (in assembly language and machine language):

```
x3100  0010 011 110000000    ; load value from x3180
x3101  0001 011 011 1 11111  ; decrement counter
x3101  0000 010 101111111    ; if zero, done
...
x317E  0000 111 100000001    ; back to top of loop
x317F  1111 0000 00100101    ; halt (TRAP x25)
x3180  0010011100010000      ; init counter (10,000)
```

```
          LD    R3, LoopCt   ; lots of loop iterations
Loop      ADD   R3, R3, #-1
          BRz   DONE
          ...
          BRnzp Loop
DONE      TRAP  x25
LoopCt    .FILL 10000        ; too big to use imm mode
```

Another use is for temporary storage. For example, we may need to save the value of a register while we use it for something else. We can allocate some memory nearby to save and restore the register's value.

```
x3100  0011 000 100110110    ; save R0 to x3136
x3101  1111 0000 00100011    ; get char in R0 (TRAP x23)
...
x3134  0010 000 100110110    ; done with char, restore R0
x3135  0000 111 100110111    ; branch over temp storage
x3136  0000000000000000      ; space for temp storage
x3137  ...
```

```
          ST    R0, SaveR0   ; save R0
          IN                 ; get char (in R0)
          ...
          LD    R0, SaveR0   ; done with char, restore R0
          BRnzp  NEXT
SaveR0 .BLKW  1
NEXT      ...
```

The disadvantage of direct mode is that we may only reference locations that are in the same page. To access the rest of memory, we must use either base+offset or indirect mode.

Base+Offset

Base+offset addressing (LDR and STR) calculates the address by adding a small unsigned index (specified in the address) to the contents of a register. Since a register can contain any value, this mode can be used to address any location in memory. In the following example, the data to be loaded (at address x4000) is on a different page than the instruction (at x3001).

```
x3000  0010 101 000001010     ; load addr from x300A
x3001  0110 010 101 000000    ; load data from (R5+0)
...
x300A  0100000000000000       ; address of data
```

```
          .ORIG  x3000
          LD     R5, FarData  ; use direct mode to load address
          LDR    R2, R5, 0    ; use base+offset to load data
          ...
FarData   .FILL  x4000
```

(We could also use indirect mode for this example – see below. Base+offset mode is a better choice if we will be accessing the data many times, since it avoids the extra memory read.)

The use of a register also makes it easy to process a sequence of memory locations. In the following example, we are calculating a table of powers of two. We use the LEA instruction to set R2 to the first location in the table. On each loop iteration, we use STR to store into the table, then increment R2 (the base address) to point to the next location.

```
x3000  1110 010 000001011     ; load table base addr into R2
x3001  0101 000 000 1 00000   ; clear R0, then add 1
x3002  0001 000 000 1 00001
x3003  0101 001 001 1 00000   ; clear R1, then add 8
x3004  0001 001 001 1 01000   ; (R1 is loop ctr)
x3005  0111 000 010 000000    ; store value (R0) to table
x3006  0001 000 000 000 000   ; double R0 (next value)
x3007  0001 010 010 1 00001   ; incr R2 (next table addr)
x3008  0001 001 001 1 11111   ; decr R1 (loop ctr)
x3009  0000 001 000000101     ; br to store if not done
x300A  0000 111 000010011     ; skip table storage
x300B  0000000000000000       ; 8 words of storage for table
x300C  0000000000000000
...
```

```
          LEA   R2, TwosTable
          AND   R0, R0, 0     ; initialize R0 to 1
          ADD   R0, R0, 1
          AND   R1, R1, 0     ; initialize R1 (loop ctr) to 8
          ADD   R1, R1, 8
Loop      STR   R0, R2, 0     ; write next table entry
          ADD   R0, R0, R0    ; compute next power of two
          ADD   R2, R2, #1    ; point to next table entry
          ADD   R1, R1, #-1
          BRp   Loop
          BRnzp Next
TwosTable .BLKW  8            ; storage for 8-element table
Next      ...
```

Indirect

With indirect mode (LDI and STI), the address specified by the instruction is not the location to be accessed. Instead, it contains the *address* of the data to be accessed. Therefore, two memory accesses are required: first, a memory read to get the address; second, a memory read or write to transfer data from/to memory.

As with base+offset, indirect mode can also be used to access any location memory, not just within the same page, since the contents of the first memory location can be any 16-bit value. The first address, however, must be within the same page, since it is specified using a page offset (just like direct mode).

Here is the same example used above, accessing data on a different page, this time using indirect mode:

```
x3000  1010 010 000001010     ; load data from addr at x300A
...
x300A  0100000000000000       ; address of data
```

```
          .ORIG x3000
          LDI   R5, FarData   ; use indirect mode to load data
          ...
FarData   .FILL x4000
```

Printing to the Console

In order to make your programs a little more interesting, we'll want to print the original string and the encrypted string to the console. This is done using the TRAP instruction, which has not been covered in the lecture yet.

You don't have to understand how the TRAP instruction works (yet). Just understand the following: If R0 contains the starting address of a string, then the LC-2 instuction 1111000000100010 (TRAP x22) prints that string to the console.

A string is just a sequence of ASCII characters in memory. The last character of the string is always null (ASCII x00). Now, ASCII characters only require 7 bits, and most machines use an 8-bit representation, with a zero in the most significant bit (bit 7). The LC-2, however, doesn't have byte-oriented instructions or addresses, so we store each ASCII character in a 16-bit word, rather than a byte.

Suppose we have the string "Cat" in the LC-2 memory, starting at address x3100. In memory, this looks like this:

Address	*Data*	*ASCII character*
x3100	x0043	'C'
x3101	x0061	'a'
x3102	x0074	't'
x3103	X0000	'\0' (null)

If we put the value x3100 into R0 and execute 1111000000100010, the string "Cat" will appear on the console output. The null character has to be there, or the printing routine won't know when to stop printing.

To print a single character, rather than a string, we put the *character* (not the address) into R0 and then execute 1111000000100001 (TRAP x21). For example, we can print a carriage return (ASCII x0D) with the following sequence of instructions:

```
0101000000000000      ; clear R0
0001000000101101      ; set R0 = 13 (x0D)
1111000000100001      ; print char in R0 (TRAP x21)
```

```
AND     R0, R0, #0    ; clear R0
ADD     R0, R0, x0D   ; set R0 to CR (x0D)
TRAP    x21           ; print R0
```

Note the difference between using a value as an argument (printing a character), compared to using an address (a pointer) as an argument (printing a string).

 Name:_____

Prelab Exercises (30 pts)

If you are submitting the prelab electronically, put the answers to all four questions into a single text file, named "prelab3.txt".

1. Data movement instructions (14 pts)

Fill in the missing parts of the instructions and/or interpretations below. For target address, specify (a) the address of the data for direct mode (as in "x5807"), (b) the base register and offset for base+offset mode (as in "R7 + 9"), or (c) the address of the load/store address for indirect mode (as in "x5807").

Address	*Instruction*	*Operation*	*Src/Dst*	*Target Address*
x2300	0110101010101001	_____	_____	_____
x7F36	____010011111111	LDR	R2	_____
x51F9	1010_____	_____	R6	x5074
x220C	____111_____	STR	_____	R6+34
x352E	0111110100101101	_____	_____	_____
x_____	0011001_____	_____	R1	x63A9

2. Choice of addressing mode (6 pts)

For each of the following scenarios, specify the most appropriate data movement instruction: LD, LDR, LDI, LEA, ST, STR, or STI.

(a) A register must be saved to a temporary memory location (and will be restored shortly thereafter).

(b) Write a value into a memory location 1000 words away from the current instruction location. This location will be accessed only a few times over the lifetime of the program.

(c) A load instruction is part of a loop, accessing a different location (depending on program data) for each loop iteration.

 Name:_____

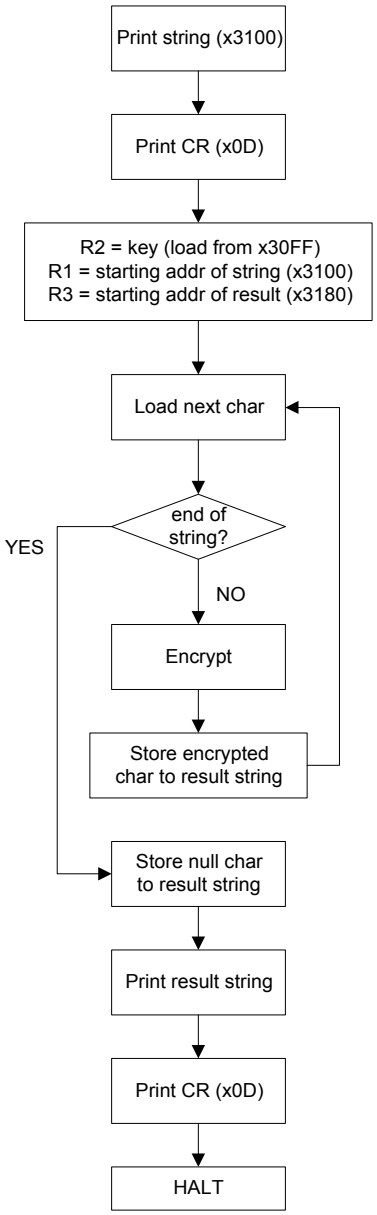
3. (5 pts) Write a sequence of LC-2 instructions (in machine language or assembly language) to add a fixed offset (in R2) to the ASCII character (in R1), wrapping around the first printable ASCII character (x20) if necessary. If the character is a non-printable character, x7f or less than x20, it should not be modified. (See Lab Exercise 1 for more details.) The result must also be stored in R1. Provide descriptive comments with your code.

4. (5 pts) Write a sequence of LC-2 instructions (in machine language or assembly language) to XOR the value in R2 with the value in R1, storing the result in R1. (See Lab Exercise 2 for more details.) Provide descriptive comments with your code.

Laboratory Exercises (70 pts)

In these two lab exercises, you will be implementing two very simple data encryption algorithms. (They are both easy to break, so don't use this to actually protect anything!)

A flowchart for both programs is shown in the figure below. First, we print the input string, which starts at address x3100, to the console, followed by a carriage return (CR). We load the encryption key into R2. Then we enter a loop in which we load the next character of the string, encrypt it using the key, and then store it to a result string, which starts at address x3180. Finally, after all the characters in the string have been encrypted, we print the encrypted string and a CR to the console.



1. Null Cipher (30 pts)

For your first program, we'll skip the encryption step entirely. This is known as a "null" cipher. The task here is to write the code for the other parts of the algorithm. Print the string at x3100, copy it to x3180, and then print the new string. The two strings should look exactly alike.

You may write the program in either assembly language or machine language. In either case, include a program header and plenty of helpful comments. Here are some detailed requirements:

- Your program must start at location x3000.
- The string to be encrypted is stored in memory, beginning at location x3100. A string is a sequence of ASCII characters that ends with a zero (the null character).
- The encrypted string will be stored in memory, beginning at location x3180.
- R1 contains the address of the character being encrypted. Initially, this will be x3100. You must load this value in your program, not by setting in the simulator.
- R2 contains the key. This is initially stored in location x30FF. You must load this value into R2 from your program, not by setting it in the simulator. (You must do this load for the null cipher, even though it's not used. This will make the next two exercises simpler.)
- R3 contains the address for storing the encrypted character. Initially, this will be x3180. You must load this value in your program, not by setting in the simulator.
- As shown in the flowchart, print the string before and after encryption, ending with a carriage return (ASCII x0D).
- Use the other registers any way you like during the program. Remember to initialize registers as needed – do not assume that they will be set to any particular value when the program begins running. (A good way to test your program is to "randomize" the machine before loading and running your program.)
- Provide a header that contains the following information: your name, your lab section, the date, and a short description of the program. Also provide comments to describe the operation of your program. *You must provide both a header and comments to receive full credit.*

To test your program, load the file "lab3test.obj" into the simulator *in addition* to your object file. This will load a test string ("This is a #@$*!% test string for ECE 206 Lab 3, Fall 2001.") into memory at location x3100. See the Appendix for instructions to create and load your own test string, if you like. It can't be more than 128 characters, including the null terminator.

The test string file will need to be loaded each time you reinitialize or randomize the state of the simulated machine. It doesn't matter whether you load the test string file before or after your program's object file.

This is what the console output should look like for the test string:

```
This is a #@$*!% test string for ECE 206 Lab 3, Fall 2001.
This is a #@$*!% test string for ECE 206 Lab 3, Fall 2001.
```

Submit your program in a filed named "lab3-1.bin" if machine language, or "lab3-1.asm" if assembly language.

2. Caesar Cipher (20 pts)

The first encryption algorithm is based on a *Caesar cipher*, used by the couriers for Julius Caesar, and probably earlier. The idea is to shift the alphabet by a fixed number of positions. Shifting by 1, for instance, causes 'A' to become 'B', 'B' to become 'C', and so forth. At the end of the alphabet, we wrap around to the beginning, so 'Z' becomes 'A'.

The number of positions to shift is the key. Presumably, only the person who knows the key can correctly decipher the encoded message.

Our variant on the Caesar cipher is to add a fixed number (the key) to the ASCII representation of each character. For this exercise, we only want to encrypt printable ASCII characters, and we also want the result to have only printable characters. The characters below x20 are non-printable, as is the DEL character (x7F). (See the ASCII table in Appendix E of the text.) You may assume that all the characters in the input string are printable characters.

Based on the program from Exercise 1 and the code that you wrote for Prelab Exercise 3, write a program to encrypt a string using our modified Caesar cipher. *You must include a program header and comments to get full credit.*

Hint: Use memory locations to store the constant values used to test whether the result is a printable character. Put these values in memory locations after your program and load them, rather than trying to "compute" them using add-immediate.

Here is the output for the test string, when the key is 3.

```
This is a #@$*!% test string for ECE 206 Lab 3, Fall 2001.
Wklv#lv#d#&C'-$(#whvw#vwulqj#iru#HFH#539#Ode#6/#Idoo#53341
```

Test your program with a larger key (by setting location x30FF before running the program) to make sure the wraparound code works.

Note: The key can be negative! Make sure your program works for both positive and negative keys.

Submit your program in a filed named "lab3-2.bin" if machine language, or "lab3-2.asm" if assembly language.

3. XOR Cipher (20 pts)

In the digital age, a popular way to mask data bits is to XOR[1] them with a random bit string. XOR is fast and simple, and the encryption and decryption algorithms are the same. To recover the original data (the *plaintext*) simply XOR the encrypted data (the *ciphertext*) with the same random bit string.

Here's an example of a 16-bit data word being encrypted with a 16-bit random key:

```
     0110101000010111   (plaintext)
⊕    1011010010100101   (key)
     1101111010110010   (ciphertext)
```

Note that wherever the key bit is one, the ciphertext bit is the inverse of the plaintext bit. Whenever the key bit is zero, the ciphertext bit is the same as the plaintext bit. To decrypt, we XOR with the same key:

```
     1101111010110010   (ciphertext)
⊕    1011010010100101   (key)
     0110101000010111   (plaintext)
```

The most secure use of XOR for encryption is called a *one-time pad*. The random bit string (the key) is the same number of bits as the plaintext, and the same bit string is never used more than once. We will use a much weaker form of encryption: a single 16-bit key will be used over and over.

Based on the program from Exercise 1 and the code that you wrote for Prelab Exercise 4, write a program to encrypt a string the XOR cipher. *You must include a program header and comments to get full credit.*

Here is the output for the test string when the key is 3:

```
This is a #@$*!% test string for ECE 206 Lab 3, Fall 2001.
Wkjp#jp#b# C')"&#wfpw#pwqjmd#elq#F@F#135#Oba#0/#Eboo#1332-
```

Try a key with more ones in it to make the output more interesting. We recommend that you leave the upper 8 bits of your key as zero. (The output routine apparently prints both bytes of a word, so you'll get two output characters per input character if the upper bits of the key are non-zero.)

Submit your program in a filed named "lab3-3.bin" if machine language, or "lab3-3.asm" if assembly language.

[1] XOR (⊕) is exclusive OR. XOR is true if exactly one of its inputs is true. $A \oplus B = \overline{A}B + A\overline{B}$.

Appendix: Writing Your Own Test String File

The easiest way to provide your own test string and key is to build an object file that loads the data into the correct memory locations. The source for the test file used above is:

```
        .ORIG x30FF
;
; key is at location x30ff
;
        .FILL 3      ;test key is 3
;
; input string follows (starting at x3100)
;
        .STRINGZ "This is a #@$*!% test string for ECE 206 Lab 3, Fall 2001."

        .END
```

If you haven't covered assembly language yet, you may not understand what's going on here. But you can create a file that looks just like this one, replacing the key value and the string with whatever you wish. Assemble the file to create an object file, and load it into the simulator to test your program.

The string must not be more than 127 characters, because you also need space for the terminating null character, and the encrypted string will be stored at location x3180.