

LC-2 Programmer's Reference and User Guide

University of Michigan EECS 100

The screenshot shows a window titled "LC-2 Simulator" with a menu bar containing "File", "Run", "Set Values", "Display", "Options", and "Help". The window is divided into several sections:

- CPU**:
 - General Purpose Registers**: R0 = 7fff (32767), R1 = ffff (65535), R2 = 0000 (0), R3 = 0000 (0), R4 = 0000 (0), R5 = 0000 (0), R6 = 0000 (0), R7 = fd79 (64889).
 - Special Registers**: PC = 3000, IR = b19a, CC (NZP) = 001.
- Memory**: A list of memory addresses from [3000] to [3014] with their contents and instructions. All contents are 0000 and instructions are ; brnop. The address [3014] has a value of \$3000.
- Information**:
 - Memory cleared
 - Registers cleared
 - LC-2 Simulator Version 5.0 (January 1, 1999)
 - Copyright (C) 1995-1999 by Matt Postiff (postiffm@umich.edu)
 - Portions of X-interface copyright (C) 1990-1994 by James R. Larus (larus@cs.wisc.edu).
 - ALL RIGHTS RESERVED.
 - See the file NOTICE in the source distribution for a full copyright notice.

Matt Postiff

Copyright (C) Matt Postiff 1995-1999. All rights reserved.
Written permission of the author is required for duplication of this document.

Table of Contents

Table of Contents	i
List of Tables	iii
List of Figures	v
Chapter 1 Introduction to the LC-2	1
1.1 Notational Conventions	3
Chapter 2 LC-2 Programmer's Reference	7
2.1 LC-2 Programming Model	9
2.2 Instruction Set	10
2.3 Summary of Instruction Formats and Semantics	28
2.4 Addressing Modes	31
2.5 Branching Modes	32
2.6 The LC-2 System Calls	33
2.7 The LC-2 Hardware Registers	34
2.8 The LC-2 Memory Map	35
Chapter 3 LC-2 Machine- and Assembly-Programming	37
3.1 Introduction to Program Execution in the LC-2	39
3.2 The LC-2 Instruction Encoding	40
3.3 Condition Codes in the LC-2 Processor	41
3.3.1 Condition Field in the Instruction Encoding	41
3.4 Keyboard Input and Console Output	42
3.5 Programming the LC-2 in Machine Language	43
3.5.1 Machine-Language File Formats	43
3.6 Programming the LC-2 in Assembly Language	46
3.6.1 Introduction	46
3.6.2 LC-2 Assembly Language	46
3.6.3 The Assembly Process	49
3.6.4 Some Examples	49
Chapter 4 LC-2 Programming Tools Guide	51
4.1 The convert Program	53
4.2 The assemble Program	54
4.2.1 The Listing File	54
4.3 The Simulator	55
4.3.1 The Simulator Menu Bar	56
Chapter 5 Appendices	61
Appendix A ASCII Reference	63

Appendix B Simulator Script Reference	65
Appendix C Installing the LC-2 Software	67

List of Tables

Table 1. Notational Conventions	3
Table 2. The Logical Operators	5
Table 3. Specifying when a branch occurs by the relationship between the instruction's nzp bits and the condition codes.	14
Table 4. LC-2 Instruction Format Summary.	28
Table 5. LC-2 addressing modes	31
Table 6. LC-2 branching modes	32
Table 7. LC-2 System Calls	33
Table 8. LC-2 Hardware Registers	34
Table 9. Condition Codes in the LC-2 Processor	41
Table 10. The Possibilities for the Condition Field in the Instruction (nzp)	41
Table 11. The elements of a line of an LC-2 assembly language program	46
Table 12. LC-2 Assembler reserved characters	47
Table 13. LC-2 assembler pseudo-ops	48
Table 14. LC-2 assembler output files	54
Table 15. ASCII Code Reference	63

List of Figures

Figure 1. LC-2 Memory Map	35
Figure 2. The binary representation of an LC-2 program (left) and its equivalent hexadecimal representation (right).	43
Figure 3. A more complicated LC-2 program	44
Figure 4. The program of Figure 2 shown in assembly language.	49
Figure 5. The program of Figure 3 shown in assembly language.	50
Figure 6. Assembler-produced listing file for dumbadd.asm	54
Figure 7. The LC-2 simulator	55
Figure 8. The LC-2 console window	56
Figure 9. The LC-2 Simulator menu bar.	57
Figure 10. The LC-2 Simulator File menu (A) and associated dialog boxes (B-D).	57
Figure 11. The LC-2 Simulator Run menu (A) and associated dialog boxes (B-D).	58
Figure 12. The LC-2 Simulator Set Values menu (A) and an associated dialog box (B).	59
Figure 13. The LC-2 Simulator Display menu (A) and an associated dialog box (B).	59
Figure 14. The LC-2 Simulator Options (A) and Help menus (B).	60
Figure 15. The LC-2 Simulator Popup menu.	60

The Little Computer 2 (LC-2) is a simple computer used to introduce general-purpose computing devices to first-year engineering students with no previous background in computer architecture or logic design. This document introduces the use and programming of the LC-2. The remainder of this chapter defines notation used throughout this manual. The **LC-2 Programmer's Reference** in Chapter 2 is a detailed reference manual for students who will write several programming assignments on the machine. Chapter 3 details LC-2 machine-language and assembly-language programming. The **LC-2 Programming Tools Guide** in Chapter 4 provides information concerning use of the assembler and simulator. The appendices provide an ASCII character reference and some detail about the simulation tools.

The LC-2 and accompanying materials have been used for the introductory computing course (EECS 100) at the University of Michigan since the Fall of 1995. Winter 1999 is the eighth semester in which the LC-2 has been used.

The course lectures, the textbook, and this document complement each other, and neither in isolation will cover the whole LC-2.

1.1. Notational Conventions

This section summarizes the notational conventions used throughout this manual.

A word is a 16-bit quantity with bit order increasing from right to left, so that the left bit is numbered 15 and the right bit is numbered 0. Bit 15 is the *most significant bit*, while bit 0 is the *least significant bit*. The following figure illustrates:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 1 lists the symbols and terms used throughout the manual.

Table 1: Notational Conventions

Notation	Meaning
\$Number, XNumber, OXNumber, 0xNumber	The number is specified in base-16 (hexadecimal).
#Number	The number is specified in base-10 (decimal).
A<l:r>	Denotes part of a value, specifically, the bits in positions l through r of value A (usually a register or memory location). For example, if the value in the PC is (hex) \$301A = (binary) 0011 0000 0001 1010, then PC<15:9> refers to the 7 bits 0011 000. If only one bit needs to be denoted, the notation is usually abbreviated as A<l>, the :r part being left off. For example, PC<2:2> is the same as PC<2>, which is the single bit found at bit position 2 of the PC (0 in this case).
A @ B	Concatenation of A and B. For example, if A is 0011 01 and B is 00 1100 1000, A @ B = 0011 0100 1100 1000.
BaseR	Base Register. one of R0..R7 whose contents specify the base address of a memory structure.
COND	Condition upon which a branch occurs. These are based on the processor status bits N, Z, and P. See the BR instruction.
page	The set of 2^9 consecutive memory locations which share the same high 7 address bits (bits <15:9> of the address).
current page	The page where the currently executing instruction resides.
DR	Destination Register; one of R0..R7 which specifies where the result of an instruction should be written
general purpose register	Any register R0..R7.

Table 1: Notational Conventions

Notation	Meaning
imm5	5-bit immediate value; 5 bits in an instruction used as a literal (immediate) value; sign extended to 16 bits. Range {-16..15}.
index6	6-bit immediate value; These 6 bits are zero extended and added to a base register to form a memory address. Range {0..63}. Note that index6 is always zero or positive.
LABEL	An assembler construct that identifies a location on the current page. It is translated into a pgoffset9 (see below).
L	Link bit; specifies whether a JSR instruction will save the value of the next PC in R7. If L == 1, the next PC is saved in R7. If L == 0, the next PC is <i>not</i> saved in R7.
MB1	Must Be One. In an instruction encoding, bits shown as MB1 must be set to 1 by the programmer or assembler. Zeroes in these bit positions may result in unexpected behavior.
MBZ	Must Be Zero. In an instruction encoding, bits shown as MBZ must be set to 0 by the programmer or assembler. Non-zero values in these bit positions may result in unexpected behavior.
mem[address]	Denotes the contents of memory at the given address.
OPC	Opcode field of instruction; 4 bits which specify the instruction to be executed by the processor.
PC	Program Counter; 16-bit, processor-internal register which contains the memory address of the <i>next</i> instruction to be fetched. For example, during execution of the instruction at address A, the PC contains address A+1. The PC is sometimes called the Instruction Pointer (IP).
pgoffset9	9 bits of offset on a page (a page address). PC<15:9> is concatenated with pgoffset9 to form a 16-bit memory address. Range {0..511}.
setcc(X)	Indicates that condition codes N, Z and P are set based on the value of X. If X is negative, N=1, Z=0, P=0. If X is zero, N=0, Z=1, P=0. If X is positive, N=0, Z=0, P=1.
SEXT(A)	Sign extension of A. The most significant bit position of A is duplicated as many times as necessary to extend A to 16 bits. For example, if A = 1100 00, then SEXT(A) = 1111 1111 1111 0000.
SR, SR1, SR2	Source Register (1 or 2); one of R0..R7 which specifies from where an instruction operand is taken.

Table 1: Notational Conventions

Notation	Meaning
trapvect8	8-bit trap number used in the TRAP instruction. Range {0..255}.
unix%	The Unix command prompt.
ZEXT(A)	Zero extension of A. Enough zero bits are added to the most significant end of A to extend it to 16 bits. For example, if A = 1100 00, then ZEXT(A) = 0000 0000 0011 0000. Note that ZEXT(A) = SEXT(A) if the most significant bit of A is 0.

Table 2 shows the truth tables of the logical operators used in this manual.

Table 2: The Logical Operators

	X	Y	X AND Y
	0	0	0
AND	0	1	0
	1	0	0
	1	1	1
	X	Y	X OR Y
	0	0	0
OR	0	1	1
	1	0	1
	1	1	1
	X		NOT X
NOT	0		1
	1		0

LC-2 Programmer's Reference

This chapter, the **LC-2 Programmer's Reference**, is a detailed reference manual for the LC-2 architecture. This chapter is divided into 8 sections. Section 1 gives an overview of LC-2 programming; 2 details the instruction set of the LC-2; 3 is an instruction summary; and 4 and 5 contain descriptions of addressing and branching modes. Section 6 lists the trap routines provided with the simulator; 7 lists the hardware registers for accessing video, keyboard, and machine control functions of the LC-2; and 8 shows a memory map of the machine.

2.1. LC-2 Programming Model

The LC-2 has 8 general purpose registers, each of which is 16 bits wide. The arithmetic and logic units operate on 16 bit words. Addresses are 16 bits wide, so the machine has 2^{16} (65, 536 or 64K) words, or 128 KB of memory.

The instruction set contains 16 basic instructions. There are operates (ADD, AND, and NOT), data movement instructions (LD, LDI, LDR, ST, STI, STR, LEA), flow control instructions (BR, JSR, JMP, JSRR, JMPR, RET), and control instructions (RTI and TRAP).

The simplest addressing scheme, direct, forms addresses by concatenating the top 7 bits of the program counter with 9 bits of page address specified in the instruction. This means that the memory space can be viewed as divided into $2^7 = 128$ pages, each of $2^9 = 512$ words. The other addressing modes of the LC-2 are indirect, base+index, and immediate. See “Addressing Modes” on page 31 for more information.

Input and output devices are mapped as part of the memory address space. See “The LC-2 Hardware Registers” on page 34 for more details.

2.2. Instruction Set

Each of the following pages describes an LC-2 instruction. The descriptions are laid out as follows:

INSTRUCTION

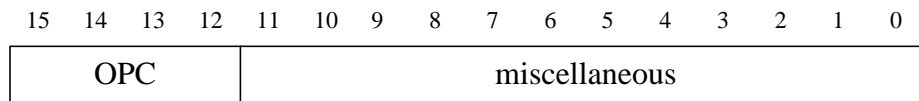
Short description

Assembler Format(s):

mnemonic [arguments]

This is the assembly language format of the instruction.

Encoding(s):



Machine code equivalent to the assembly language format.

Operation:

A formal description of the instruction.

Description:

English description of the action of the instruction, including any side effects. Includes special notes and references to related instructions. Includes a note about which, if any, of the condition codes are modified by the instruction. If an instruction modifies a condition code, the condition code is set or cleared depending on the result of the operation performed by the instruction. So,

if the result of the operation is negative, then $N = 1, Z = 0, P = 0$.

if the result of the operation is zero, then $N = 0, Z = 1, P = 0$.

if the result of the operation is positive, then $N = 0, Z = 0, P = 1$.

Example(s):

Some example uses of the instruction.

ADD

Add

Assembler Formats:

```
ADD DR, SR1, SR2
ADD DR, SR1, imm5
```

Encodings:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				DR			SR1			0	MBZ		SR2		
0001				DR			SR1			1	imm5				

Operation:

```
if (bit<5> == 0) {
    DR = SR1 + SR2
}
else if (bit<5> == 1) {
    DR = SR1 + SEXT(imm5)
}
setcc(DR)
```

Description:

If bit<5> == 0, then add the contents of SR1 and SR2. If bit<5> == 1, then add the contents of SR1 and the sign extended immediate value. The result in both cases is placed in DR and sets the condition codes.

Any carry generated at the most significant bit of the addition is discarded, and overflow is not signalled.

Examples:

```
ADD    R2, R3, R4    ; R2 = R3 + R4
ADD    R2, R3, #7    ; R2 = R3 + 7
```

AND

Bitwise Logical AND

Assembler Formats:

```
AND DR, SR1, SR2
AND DR, SR1, imm5
```

Encodings:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0101				DR			SR1			0	MBZ		SR2		
0101				DR			SR1			1	imm5				

Operation:

```
if (bit<5> == 0) {
    DR = SR1 AND SR2
}
else if (bit<5> == 1) {
    DR = SR1 AND SEXT(imm5)
}
setcc(DR)
```

Description:

If bit<5> == 0, then perform the bitwise AND of the contents of SR1 and SR2. If bit<5> == 1, then perform the bitwise AND of the contents of SR1 and the sign extended immediate value. The result in both cases is placed in DR and sets the condition codes.

Examples:

```
AND    R2, R3, R4    ; R2 = R3 AND R4
AND    R2, R3, #7    ; R2 = R3 AND 7
```

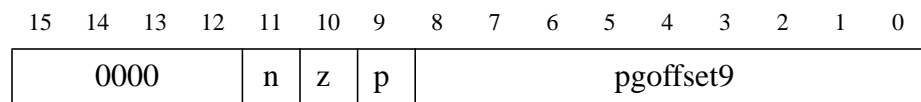
BR

Branch to Location on Current Page

Assembler Formats:

	Equivalent Mnemonics
BRn LABEL	BRlt LABEL
BRz LABEL	BReq LABEL
BRp LABEL	BRgt LABEL
BRnz LABEL	BRle LABEL
BRnp LABEL	BRne LABEL
BRzp LABEL	BRge LABEL
BRnzp LABEL	BR LABEL
BRNOP	NOP

Encoding:



Operation:

```
if ((n AND N) OR (z AND Z) OR (p AND P)) {  
    PC = PC<15:9> @ pgoffset9  
}
```

Description:

If the condition is true, branch to the specified location on the current page. The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form the new 16-bit PC value, which is written to the PC only if the condition is true. The condition codes are not modified.

The nzp bits specify how to compute the condition under which the processor should take the branch. Table 3 explains the nzp encodings.

Table 3: Specifying when a branch occurs by the relationship between the instruction's nzp bits and the condition codes.

nzp	Condition Codes Set	Condition	nzp	Condition Codes Set	Condition
000	NA	Do not branch under any condition	100	N	Negative Less Than
001	P	Positive Greater Than	101	N or P	Negative or Positive Not Equal
010	Z	Zero Equal	110	N or Z	Negative or Zero Less than or Equal
011	Z or P	Zero or Positive Greater than or Equal	111	N, Z, or P	Negative, Zero, or Positive Unconditional

The processor examines the n, z, and p bits in the instruction to determine which LC-2 condition codes to test. If n is set, test N; if n is not set, ignore N. If z is set, test Z, etc. If any of the condition codes tested in this way is set, the branch is taken. If none of n, z, or p are set in the instruction, the processor treats the branch instruction as it would treat a NOP. If n, z, and p are all set, the processor branches unconditionally, regardless of the condition code setting.

JSR/JMP and JSRR/JMPR also control program flow.

Zero is neither positive nor negative.

Example:

```
BRzp LOOP ; Branch to LOOP if last result was zero or positive
```

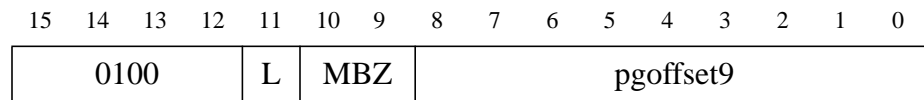
JSR JMP

Jump to Subroutine Jump

Assembler Format:

```
JSR LABEL          (L = 1)
JMP LABEL          (L = 0)
```

Encoding:



Operation:

```
if (L == 1) {
    R7 = PC
}
PC = PC<15:9> @ pgoffset9
```

Description:

Unconditionally jump to the specified location on the current page. If the link bit L is set, the PC is saved in R7, allowing a later return. The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form the new 16-bit PC value. The condition codes are not modified.

Examples:

```
JSR    FOO          ; Jump to FOO, put return PC into R7
JMP    FOO          ; Jump to FOO
```

JSRR JMPR

Jump to Subroutine through Register Jump through Register

Assembler Format:

JSRR BaseR, index6 (L = 1)
JMPR BaseR, index6 (L = 0)

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1100				L	MBZ			BaseR			index6				

Operation:

```
if (L == 1) {  
    R7 = PC  
}  
PC = BaseR + ZEXT(index6)
```

Description:

If the link bit L is set, the PC is saved in R7, allowing a later return. The index is zero extended to 16 bits and added to the contents of BaseR to form the new PC value. The condition codes are not modified.

Examples:

```
JSRR  R2, #10 ; Jump to R2 + 10, put return PC into R7  
JMPR  R2, #10 ; Jump to R2 + 10
```

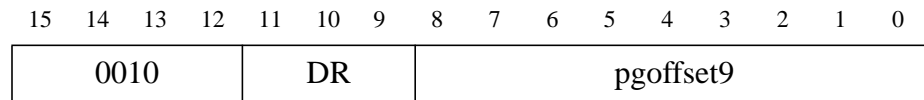

LD

Load Direct from Memory to Register

Assembler Format:

LD DR, LABEL

Encoding:



Operation:

DR = mem[PC<15:9> @ pgoffset9]
setcc(DR)

Description:

Load a register from the specified location on the current page. The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form a 16-bit memory address. The contents of memory at this address is loaded into DR and is used to set the condition codes.

Example:

LD R4, COUNT ; R4 = mem[COUNT]

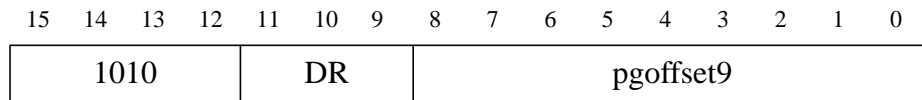
LDI

Load Indirect from Memory to Register

Assembler Format:

```
LDI DR, LABEL
```

Encoding:



Operation:

```
DR = mem[ mem[ PC<15:9> @ pgoffset9 ] ]  
setcc(DR)
```

Description:

Load a register indirectly from the specified location. The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form a 16-bit memory address. The contents of memory at this location is used as the address of the data that is loaded into DR. The data loaded into DR sets the condition codes.

Example:

```
LDI    R4, POINTER    ; R4 = mem[mem[ POINTER ] ]
```

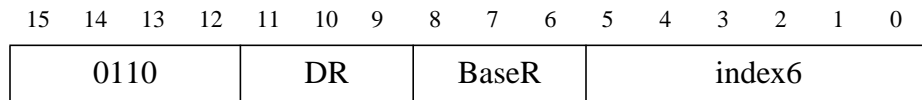
LDR

Load from mem[Base + Index] to Register

Assembler Format:

LDR DR, BaseR, index6

Encoding:



Operation:

DR = mem[BaseR + ZEXT(index6)]
setcc(DR)

Description:

Load a register using a base register and index. The index is zero extended to 16 bits and added to the contents of BaseR to form a memory address. The contents of memory at this address is loaded into DR and sets the condition codes. Note that index6 is a positive offset.

Example:

LDR R4, R2, #10 ; R4 = contents of mem[R2 + 10]

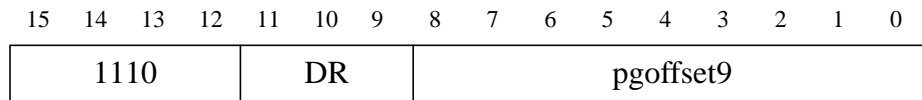
LEA

Load Effective Address

Assembler Format:

LEA DR, LABEL

Encoding:



Operation:

DR = PC<15:9> @ pgoffset9
setcc(DR)

Description:

The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form a 16-bit memory address. This address is placed into DR and sets the condition codes.

Example:

LEA R4, F00 ; R4 = address of F00

NOT

Bitwise NOT (invert or complement)

Assembler Format:

NOT DR, SR

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1001				DR				SR				MB1			

Operation:

DR = NOT(SR)
setcc(DR)

Description:

Perform the bitwise complement operation on the contents of SR and place the result in DR. The result sets the condition codes.

Example:

NOT R4, R2 ; R4 = NOT(R2)

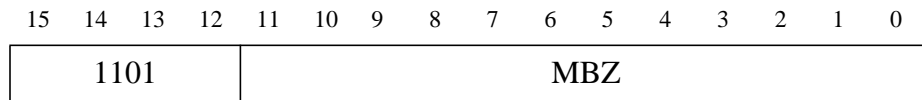
RET

Return from Subroutine

Assembler Format:

RET

Encoding:



Operation:

PC = R7

Description:

Load the PC with the value in R7, thus allowing a return from a previous JSR or JSRR instruction. The condition codes are not modified.

Example:

RET ; PC = R7

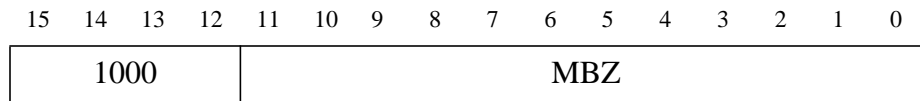
RTI

Return from Interrupt

Assembler Format:

RTI

Encoding:



Operation:

```
setcc(mem[R6])  
R6 = R6 - 1  
PC = mem[R6]  
R6 = R6 - 1
```

Description:

Pop the top of the stack, and load the value into PC. The condition codes are not modified.

Example:

```
RTI                ; first pop condition codes  
                   ; then pop PC
```

Notes:

On an external interrupt, the initiating sequence pushes the PC onto the stack, then pushes the current condition codes onto the stack, then loads the PC with the starting address of the service routine. The last instruction in the service routine is RTI, which returns control to the interrupted program by popping the stack and loading the value popped into the condition codes, then popping the stack again and loading the value popped into the PC.

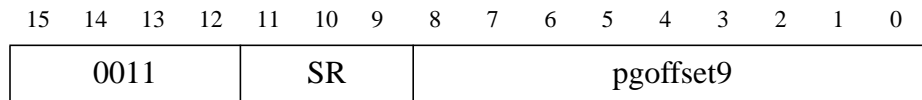
ST

Store Direct from Register to Memory

Assembler Format:

ST SR, LABEL

Encoding:



Operation:

$\text{mem}[\text{PC}\langle 15:9 \rangle @ \text{pgoffset9}] = \text{SR}$

Description:

Store from the specified register to the specified location on the current page. The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form a 16-bit memory address. The contents of SR are copied into memory at this address. The condition codes are not modified.

Example:

ST R4, COUNT ; mem[COUNT] = R4

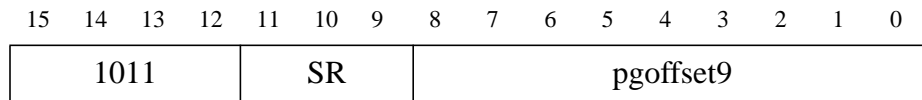
STI

Store Indirect from Register to Memory

Assembler Format:

STI SR, LABEL

Encoding:



Operation:

$\text{mem}[\text{mem}[\text{PC}\langle 15:9 \rangle @ \text{pgoffset9}]] = \text{SR}$

Description:

Store from the specified register to the indirectly specified location. The top 7 bits of the current program counter are concatenated with the pgoffset9 field to form a 16-bit memory address. The contents of memory at this address are used as the address where the contents of SR are to be copied. The condition codes are not modified.

Example:

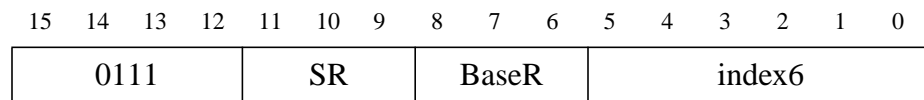
STI R4, POINTER ; mem[mem[POINTER]] = R4

STR Store from Register to mem[Base + Index]

Assembler Format:

STR SR, BaseR, index6

Encoding:



Operation:

$\text{mem}[\text{BaseR} + \text{ZEXT}(\text{index6})] = \text{SR}$

Description:

Store from the specified register to the specified location. The offset is zero extended to 16 bits and added to the contents of BaseR to form a memory address. The contents of SR are copied into memory at this address. The condition codes are not modified.

Example:

STR R4, R2, #10 ; mem[R2 + 10] = R4

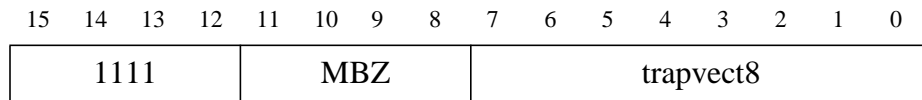
TRAP

Call System Routine

Assembler Format:

```
TRAP trapvect8
```

Encoding:



Operation:

```
R7 = PC  
PC = mem[ ZEXT(trapvect8) ]
```

Description:

Execute the system call specified by the trap number. The PC is saved in R7, allowing a later return. The trapvect8 field is zero extended to 16 bits. The contents of this memory location are placed into the PC. The condition codes are not modified.

Each of the first $2^8 = 256$ memory locations contains the starting address for the system call specified by its corresponding trap number. This area of memory is called the trap vector table. See “The LC-2 System Calls” on page 33 for more information.

Example:

```
TRAP    x23                ; Execute the IN system call
```

2.3. Summary of Instruction Formats and Semantics

Table 4: LC-2 Instruction Format Summary

Mnemonic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
*ADD	0001				DR			SR1			0	MBZ		SR2		
*ADD	0001				DR			SR1			1	imm5				
*AND	0101				DR			SR1			0	MBZ		SR2		
*AND	0101				DR			SR1			1	imm5				
BR	0000				n	z	p	pgoffset9								
JSR	0100				L	MBZ		pgoffset9								
JSRR	1100				L	MBZ		BaseR			index6					
*LD	0010				DR			pgoffset9								
*LDI	1010				DR			pgoffset9								
*LDR	0110				DR			BaseR			index6					
*LEA	1110				DR			pgoffset9								
*NOT	1001				DR			SR			MB1					
RET	1101				MBZ											
RTI	1000				MBZ											
ST	0011				SR			pgoffset9								
STI	1011				SR			pgoffset9								
STR	0111				SR			BaseR			index6					
TRAP	1111				MBZ				trapvect8							

*Indicates instructions that modify the condition codes.

The following Verilog summarizes the behavior of each instruction. The Verilog variable containing the instruction is called `inst`. Symbols such as `DR`, `ZEXT`, `trapvect8`, etc. are used as defined earlier in this document. The register file is a memory array called `R[]`.

```
// Memory State
    reg [15:0] mem [65535:0];      // primary memory

// Processor State
    reg [15:0] PC;                // program counter
    reg [15:0] R [7:0];          // register file
    reg N, Z, P;                // condition codes

// Instruction Format
    reg [15:0] i;                // instruction
    reg [3:0] opc = i[15:12];     // opcode
    reg [4:0] imm5 = i[4:0];      // immediate data
    reg [8:0] pgoffset9 = i[8:0]; // page address
    reg [5:0] index6 = i[5:0];   // immediate index
    reg [7:0] trapvect8 = i[7:0]; // trap vector
    reg [2:0] DR = i[11:9];       // destination register
    reg [2:0] SR = i[11:9];       // source registers
    reg [2:0] SR1 = i[8:6];
    reg [2:0] SR2 = i[2:0];
    reg [2:0] BaseR = i[8:6];
    reg L = i[11];                // link bit

// Interpret the instruction stream
task interpret begin
    repeat begin
        i = MP[PC];
        PC = PC + 1;
        execute;
    end
end // task interpret

// Execute an instruction
task execute begin
    case (i[15:12])
        `ADD: begin
            if (i[5] == 0) R[DR] = R[SR1] + R[SR2];
            if (i[5] == 1) R[DR] = R[SR1] + SEXT(imm5);
            set_condition_codes(R[DR]);
        end
        `LD: begin
```

```

    R[DR] = mem[ {PC[15:9], pgoffset9} ];
    set_condition_codes(R[DR]);
    end
`ST: mem[ {PC[15:9], pgoffset9} ] = R[SR];
`JSR: begin // also called `JMP
    if (L == 1) R[7] = PC;
    PC = {PC[15:9], pgoffset9};
    end
`AND: begin
    if (i[5] == 0) R[DR] = R[SR1] & R[SR2];
    if (i[5] == 1) R[DR] = R[SR1] & SEXT(imm5);
    set_condition_codes(R[DR]);
    end
`LDR: begin
    R[DR] = mem[ R[BaseR] + ZEXT(index6) ];
    set_condition_codes(R[DR]);
    end
`STR: mem[ R[BaseR] + ZEXT(index6) ] = R[SR];
`BR: begin
    if ((i[11] & N) | (i[10] & Z) | (i[9] & P))
        PC = {PC[15:9], pgoffset9};
    end
`NOT: begin
    R[DR] = ~R[SR1];
    set_condition_codes(R[DR]);
    end
`LDI: begin
    R[DR] = mem[ mem[ {PC[15:9], pgoffset9} ] ];
    set_condition_codes(R[DR]);
    end
`STI: mem[ mem[ {PC[15:9], pgoffset9} ] ] = R[SR];
`JSRR: begin // also called `JMRR
    if (L == 1) R[7] = PC;
    PC = R[SR1] + ZEXT(index6);
    end
`RET: PC = R[7];
`RTI: begin
    PC = mem[ R[6] ];
    R[6] = R[6] - 1;
    end
`LEA: begin
    R[DR] = {PC[15:9], pgoffset9};
    set_condition_codes(R[DR]);
    end
`TRAP: begin
    R[7] = PC;

```

```

        PC = mem[ ZEXT(trapvect8) ];
        end
    endcase
end // task execute

// Computing condition codes
task set_condition_codes;
    input [15:0] a;
begin
    N = Z = P = 0;
    if (a[15] == 1) N = 1;
    else if (a == 16'b0) Z = 1;
    else P = 1;
end // task set_condition_codes
endtask

```

2.4. Addressing Modes

An addressing mode defines the way in which data is accessed. The LC-2 has five addressing modes which are summarized in the table below.

Table 5: LC-2 addressing modes

Name of Addressing Mode	Meaning	Example
register	The data is in the specified register.	add r1, r2, r3
immediate (literal)	The data is in the instruction. In the example, the #5 is the immediate data.	add r1, r2, #5
	The data to be loaded is the literally -specified address on the current page.	lea r1, LABEL
	(This addressing mode is often referred to as <i>literal mode</i> since the bits in the instruction “literally” are used as data.)	
direct	The data is at the specified address on the current page.	ld r1, LABEL
indirect	The address of the data is contained in the location specified on the current page.	ldi r1, LABEL
base+index	Data is at the address formed by adding the zero-extended index (#3 in the example) to the base register (r2 in the example).	ldr r1, r2, #3

Direct addressing is provided in the `pgoffset9` field (`INST<8:0>`) of the LD and ST instructions. The 9 bits directly identify the memory address on the current page. `PC<15:9>` specifies the current page. Since there are 7 bits to identify the current page, there can be $2^7 = 128$ pages. A complete 16-bit address is formed by concatenating the page number `PC<15:9>` and the `pgoffset9` (i.e., `PC<15:9> @ INST<8:0>`). This address is used as the address of the 16-bit data element.

The indirect addressing mode allows the contents of a memory location to contain the address of the data element. [You will see when we get to programming in C that this is a very useful concept, and has a special name. That is, a location that contains the address of the data is referred to as a “pointer” because it points to the data.] It is implemented with the LDI and STI instructions. `INST<8:0>` are used exactly as they are in the direct addressing mode to form an address. However, the resulting address contains the address of the data element to be fetched (in the case of LDI) or stored (in the case of STI).

The base+index addressing mode allows the programmer to specify a memory location as an offset from some particular starting address. The starting address is contained in a base register `BaseR`. The offset is specified as the index in the instruction. A memory address is formed by adding the contents of the base register with the zero-extended offset. This address is then used for the LDR or STR operation. It is convenient to use the base+index addressing mode to process sequential data structures such as strings, records (e.g. activation records), arrays of structures, etc. Supposing that the base register points to the beginning of an array of items, it must be incremented by the size of each element to traverse through the array. The index can specify a field in each item.

2.5. Branching Modes

A branching mode defines the way in which a branch or jump takes place. The LC-2 has two basic modes which are summarized below.

Table 6: LC-2 branching modes

Examples	Meaning
BR LABEL JSR LABEL	The target of the branch or jump is the specified address on the current page.
JSRR BaseR, index6	The target of the jump is the sum of the contents of the register <code>BaseR</code> and the zero-extended 6-bit index encoded in the instruction.

Direct branches are provided by the BR and JSR instructions. `INST<8:0>` directly identify the target address on the current page. `PC<15:9>` specifies the current page. A complete 16-bit address is formed by concatenating the page number `PC<15:9>` and the `pgoffset9` (that is, `PC<15:9> @ INST<8:0>`). This is the target address placed into the PC.

Jumps to locations not on the current page can be implemented with the JSRR instruction. The new PC is formed by adding the contents of the `BaseR` with the 6-bit index.

The L (link) field in the JSR instructions distinguish them from the BR instruction. If L is set (`L == 1`), the current PC is copied to R7 before the jump takes place. This action forms a *link* to the calling subroutine for a later return. If L is clear (`L == 0`), the current PC is discarded.

2.6. The LC-2 System Calls

The LC-2 provides several trap routines for the programmer's convenience. They are as follows.

Table 7: LC-2 System Calls

Trap Number	Assembler Name	Description
\$20	GETC	Read a single character from the keyboard. The character is not echoed onto the console. Its ASCII code is copied into R0. The high 8 bits of R0 are cleared.
\$21	OUT	Write a character in R0<7:0> to the console.
\$22	PUTS	Write a string pointed to by R0 to the console.
\$23	IN	Print a prompt on the screen and read a single character from the keyboard. The character is echoed onto the console, and its ASCII code copied into R0. The high 8 bits of R0 are cleared.
\$24	PUTSP	Write a packed string pointed to by R0 to the console.
\$25	HALT	Prints a message on the console and halts execution.
Others		All other trap vectors point to a routine which prints a message on the console indicating that a trap was executed for which no other routine was defined.

2.7. The LC-2 Hardware Registers

Certain memory addresses specify registers which have special meanings to the LC-2.

Table 8: LC-2 Hardware Registers

Location	I/O Register Name	I/O Register Function
\$F3FC	Video Status Register	Bit<15>, the ready bit, indicates whether the video device is ready to receive another character to print on the screen. Also known as CRTSR.
\$F3FD	Horizontal Screen Position	Not implemented.
\$F3FE	Vertical Screen Position	Not implemented.
\$F3FF	Video Data Register	A character written in the low byte of this register will be displayed on screen. If the high byte is not 0, it too will be written (packed strings). Also known as CRTDR.
\$F400	Keyboard Status Register	Bit<15>, the ready bit, indicates whether the keyboard has received a new character. Also known as KBSR.
\$F401	Keyboard Data Register	Bits<7:0> contain the last character typed on the keyboard. Also known as KBDR.
\$FFFF	Machine Control Register	Bit<15> is the clock enable bit. When cleared, instruction processing stops because the clock signal stops pulsing. Also known as MCR.

2.8. The LC-2 Memory Map

The LC-2's complete 16-bit, 64 KW memory space is shown below. There are $2^7 = 128$ pages with 512 words each. Memory between \$3000 and \$CFFF (40 KW) is available for user programs. Each block represents two pages.

Figure 1: LC-2 Memory Map

	X000-X3FF	X400-X7FF	X800-XBFF	XC00-XFFF
0000	Vector Table	Operating System Code	Operating System Code	Operating System Code
1000	Reserved	Reserved	Reserved	Reserved
2000	Reserved	Reserved	Reserved	Reserved
3000				
4000				
5000				
6000				
7000				
8000				
9000				
A000				
B000				
C000				
D000	Reserved	Reserved	Reserved	Reserved
E000	Reserved	Reserved	Reserved	Reserved
F000	Video RAM* Video registers	Keyboard System RAM**	System RAM**	Boot ROM

* packed memory: 2 characters are stored in each word

** memory mapped I/O devices

*LC-2 Machine- and
Assembly-Programming*

This chapter discusses programming the LC-2 in machine-
and assembly-language.

3.1. Introduction to Program Execution in the LC-2

The LC-2's memory is logically laid out in a linear array. Each entry in the array is identified by a unique 16-bit number (its *address*). Each entry contains 16 bits of information.

LC-2 instructions are stored in its memory. Each instruction is 16 bits long. To begin execution of an instruction, the processor first retrieves (*fetches*) the instruction from the memory location which contains the instruction. The instruction is then *decoded* and *executed*.

As part of the execution, many instructions also record whether the result of execution is negative, zero, or positive. This is done by setting or clearing the three corresponding condition codes (N, Z, and P).

After finishing execution, whatever results the instruction produces are recorded, and the next instruction is processed. The processor keeps track of the next instruction with a special register called the *program counter* (PC). The program counter specifies the address from which the next instruction is to be retrieved.

From the programmer's perspective, a *machine instruction* is the most basic operation that a computer can carry out. In the case of the LC-2, there are 16 basic instructions, each 16 bits long, which the processor knows how to execute directly. These include operate instructions (which process information), data movement instructions, and control instructions (which modify the normal sequential flow of the program).

3.2. The LC-2 Instruction Encoding

Consider an arbitrary LC-2 instruction which we will call INST. When the LC-2 executes INST, it examines bits <15:12>, the opcode, to figure out what the instruction is telling it to do. For example, the opcode may represent “add.” The rest of the bits in the instruction tell the LC-2 what quantities to add and where to put the result.

INST<11:9> usually specifies a destination register or condition codes. The destination register (DR) is specified as 000, 001, 010, 011, 100, 101, 110, or 111, representing registers zero through seven (R0..R7). For instance, if bits <15:9> = 1001 010, the opcode is NOT and the result will be placed into R2.

One exception to this general rule is the `st` instruction, where INST<11:9> actually specify a *source* register.

In the case of `and`, `add`, and `not`, at least one source register (SR or SR1) is required. It is encoded in bits <8:6>. It is encoded in the same fashion as the DR field.

The `add` and `and` instructions are different from `not` in that they need two source operands. It is also convenient to be able to literally specify in the instruction a number to add or and. To accomplish both in the same instruction format, the LC-2 supports the execution of the following instructions:

- a) `add R0, R1, R2` ; R0 = R1 + R2
- b) `add R0, R0, #1` ; R0 = R0 + 1 (increment R0)
- c) `and R0, R1, R2` ; R0 = R1 & R2
- d) `and R0, R0, #0` ; R0 = R0 & 0 (clear R0)

To specify a) and c), INST<5> is 0. This indicates that the last 3 bits of the instruction, INST<2:0> are encoded as the second source register (R2 in both cases above). The intervening bits INST<4:3> are 0. To specify b) and d), INST<5> is 1. The remaining 5 bits <4:0> designate a 5 bit number. This number is sign extended to 16 bits, which allows us to use positive as well as negative numbers in the 5-bit immediate field. The allowed values range from -16 to 15.

Some instructions require a nine-bit field which is a page offset. As will be explained shortly, this offset tells the processor where to fetch data from on the current page (defined by the program counter). This page offset field is contained in INST<8:0>.

3.3. Condition Codes in the LC-2 Processor

The hardware for an LC-2 computer (or in our case, the simulator) contains three flip flops which retain properties about the results computed by the processor. The flip flops, or more appropriately the signals that they store, are called the *condition codes* of the LC-2. Sometimes condition codes are referred to as *flags*. The instructions that may modify the contents of those flip flops are detailed in the **LC-2 Programmer's Reference Manual**. The three condition codes are described in Table 9.

Table 9: Condition Codes in the LC-2 Processor

Name	Meaning
N	Set if bit 15 of the instruction result is 1. That is, if the result is <i>negative</i> when interpreted as a two's complement number. Cleared otherwise.
Z	Set if the instruction result is <i>zero</i> (\$0000). Cleared otherwise.
P	Set if the instruction result is <i>positive</i> , that is, neither negative nor zero. Note that \$0000 is neither positive nor negative. Cleared otherwise.

3.3.1. Condition Field in the Instruction Encoding

To control the flow of a program, branch instructions are used to skip or repeat sequences of instructions. The LC-2 provides several ways to perform such *branches*. The programmer controls the conditions under which a branch may occur by using three bits in the branch instruction encoding which we call the 'nzp' bits. The 3 bits together form the *condition field* of the branch instruction. For the LC-2, these bits may be used in any combination to specify when a branch occurs. Table 10 shows all the combinations.

Table 10: The Possibilities for the Condition Field in the Instruction (nzp)

nzp	Take the branch...
000	...never
100	...if N is set (only negative)
010	...if Z is set (only zero)
001	...if P is set (only positive)
110	...if N or Z is set (negative or zero, but not positive)
011	...if Z or P is set (zero or positive, but not negative)
101	...if N or P is set (negative or positive, but not zero)
111	...if any of N, Z, or P are set (after the processor sets the condition codes, at least one bit is always set, so the branch will always be taken ^a)

a. Note that upon machine startup, the contents of the condition codes are undefined. Using the condition codes in a branch instruction before setting them results in undefined operation.

3.4. Keyboard Input and Console Output

The trap instruction has an 8-bit field $\text{INST}\langle 7:0 \rangle$ which is a *trap number*. The trap number indexes a table at the start of the LC-2's memory which provides access to special operating system routines.

Two instances of the `trap` instruction are the `in` and `out` operations. To avoid explaining all of the details behind this instruction, we will simply say that it should be used “as is” until a more complete explanation can be given.¹ When an `in` instruction is reached in a program, program execution stops until a key on the keyboard is pressed. The ASCII representation of the pressed key is placed into the low half of the R0 register and control is returned to the program. The upper half of the R0 register $\text{R}\langle 15:8 \rangle = 00000000$ after use of the `in` instruction. The `out` instruction works similarly. When encountered in a program, the character contained in the low half of R0 is written to the LC-2 console. The upper half of the register $\text{R}\langle 15:8 \rangle$ must be 00000000 for the operation to work correctly.

1. Lectures in class will fully explain this. Bear with us as the textbook for the course is not yet written.

3.5. Programming the LC-2 in Machine Language

3.5.1. Machine-Language File Formats

An LC-2 program written in machine language is formatted as a sequence of lines, one 16-bit instruction per line. The first line must indicate where the program should be loaded into the LC-2's memory. The load position is usually \$3000 (hex) or 0011000000000000 (binary) for a normal program. The remaining lines are machine instructions or data which will be loaded into contiguous locations of LC-2's memory.

The program can be written in your favorite text editor in binary notation as in Figure 2. The spaces in the binary representation are for readability only.

Figure 2: The binary representation of an LC-2 program (left) and its equivalent hexadecimal representation (right).

0011	0000	0000	0000	3000
1111	0000	0010	0011	F023
0001	0010	0010	0000	1220
1111	0000	0010	0011	F023
0001	0000	0000	0001	1001
1111	0000	0010	0001	F021
1111	0000	0010	0101	F025

The program may also be written as hexadecimal notation (consisting of the characters 0 through 9 and A through F), as on the right of Figure 2.

These listings represent a very simple LC-2 program. To understand what this program does, simply decode each of the instructions using the information from the reference manual. The first line instructs the simulator where to load the program into the LC-2's memory. The second line is an input instruction which reads a character from the keyboard into R0. The third line adds R0 and 0 and places the result into R1 (which effectively copies² R0 into R1). The next instruction reads a second character into R0. The fifth line adds R0 and R1 and places the result into R0. The sixth line writes the character in R0 to the console. Finally, the last line is a HALT instruction which stops instruction processing.

LC-2 programs can also be stored in true binary format (*not* like that in Figure 2). This format is nearly impossible to read directly, unlike the textual forms shown above. It is similar to the format in which commercial software is distributed.

Incidentally, the program of Figure 1 is provided in the class examples directory in the file called `dumbadd.obj`. Try it using the simulator and see what happens.

2. This type of copy operation is often called a "move."

A more complicated example is shown in Figure 3.

Figure 3: A more complicated LC-2 program

Page Address (hex)	Program Hex	Function
	3000	Program load address
000	E612	Load the full address of a buffer at page address \$012 into register R3
001	54A0	AND R2 with \$0 (produces \$0) and place result into R2.
002	F023	Read a character from the keyboard
003	1236	Add \$36 to the character read in and place result into R1 This is the same as subtracting \$0A. How? Consider that the value \$36 is sign extended before the addition and remember your two's complement arithmetic. Why? So that we can perform a comparison against the Enter key [newline].
004	8409	If the last result was zero, branch to location \$009
005	70C0	Store R0 into memory pointed to by R3+0
006	16E1	Increment R3 (move pointer one word ahead)
007	14A1	Increment R2 (add 1 to counter)
008	4802	Jump to location \$002 on this page
009	16FF	Decrement the pointer to point to the last character read into the buffer
00A	5482	AND R2 with itself and place result into R2. This action is to set the flag bits according to R2 (I am using it specifically for the side effect, just to make sure that the flags are set correctly).
00B	8411	If R2 was zero, branch to location \$010 on this page
00C	60C0	Load a character into R0 from memory pointed to by R3+0
00D	F021	Write the character to the console
00E	14BF	Decrement R2 (subtract 1 from counter)
00F	16FF	Decrement R3 (move pointer one word backward)
010	480A	Jump to offset \$0A on this page
011	F025	HALT service routine

Figure 3: A more complicated LC-2 program

Page Address (hex)	Program Hex	Function
012	0000	The next 10 locations are reserved to hold characters of an input string.
013	0000	
014	0000	
015	0000	
016	0000	
017	0000	
018	0000	
019	0000	
01A	0000	
01B	0000	

As before, we use the reference manual to figure out what each line of the program does. The third column of the table summarizes the function of each line, though a lot of high-level information has been left out. Try to figure out what the program does and what registers R0, R2, and R3 are used for.

This program is called `reverse.obj` and is in your `eeecs100` directory. Run it through the simulator and see what happens. (Hint: type a few characters and then hit the ENTER key.) Correlate the results with the code that you see above.

3.6. Programming the LC-2 in Assembly Language

3.6.1. Introduction

As you have probably realized, it is tedious to write even small programs in the LC-2's machine language. To facilitate the development of larger programs, a language is provided which allows the use of symbolic names for opcodes as well as memory locations. It is called the LC-2 *assembly language*.

Because the LC-2 can only execute machine language instructions, a translation step is necessary to convert the symbolic language into a binary one. A program called an *assembler* performs this function; the processing is called *assembling*.

An assembly language program (the *source program*) consists of a sequence of instructions, one per line. Each line can contain either (1) a comment, which is ignored by the assembler; (2) a real instruction, which corresponds to a machine language instruction; or (3) a pseudo-op. Pseudo-ops, which will be discussed presently, are so named because they do not correspond to machine language instructions, but rather are messages provided by the assembly language programmer to the assembler program to help in the translation of the assembly language program to binary machine language.

3.6.2. LC-2 Assembly Language

The format of each line of an assembly language program is as follows:

`LABEL (whitespace) MNEMONIC (whitespace) ARG1, ARG2, ... (whitespace) ; comment`

Any amount of whitespace (including the space and tab characters) can be used between elements. The four basic elements of a *line* of assembly code are described in Table 11:

Table 11: The elements of a line of an LC-2 assembly language program

Element	Meaning
LABEL	A descriptive label for the line; not necessary unless there is a need to identify the information in the corresponding memory location: for example, if the line is the destination of a jump instruction or the source of data to be loaded.
MNEMONIC	The mnemonic for the assembly language instruction; ADD, LD, JSR, or pseudo-ops like .ORIG, and special names for which the assembler provides a shortcut, such as IN, OUT, and HALT.
ARG1, ARG2, ...	Opcode-dependent arguments; ADD requires 3, LD requires 2, HALT requires none. Each argument is separated by at least some whitespace, and optionally a comma.

Table 11: The elements of a line of an LC-2 assembly language program

Element	Meaning
; comment	Text describing the function of the line in the context of the given program (not the obvious ‘add R0 to R1’ but ‘increment the character count’); all text beyond the semicolon is ignored by the assembler and is only there for the reader’s benefit.

A label is used to identify a memory location. There are two main uses for a label. If the memory location contains data, we want to be able to refer to it using a symbolic name. A label can also be used to symbolically identify an instruction. Another instruction (a jump or branch) can then explicitly tell the processor what instruction to execute next by using the identifying label attached to the desired instruction.

Labels can be constructed using a combination of almost any characters that the assembler recognizes. Exceptions include characters that the assembler reserves for some special use. The characters with reserved uses are shown in Table 12.

Table 12: LC-2 Assembler reserved characters

Character	Reserved Use
; or //	identifies the start of a comment
\$, x, X, 0x, 0X	start of a hexadecimal constant
%, b, B	start of a binary constant
#	start of a decimal constant
R, r	denote a register identifier; the LC-2 registers are R0, R1, ..., R7.

Depending on the instruction, a constant value may be specified. The length of that constant also depends on the instruction, and the assembler will check to see that the constants are in range. For example, the constant value ten can be represented as any of the following: #10, \$A, %1010, \$0A, #010, etc. If an instruction such as an ADD is used with an immediate value that is too large to fit into the immediate field of the instruction (5 bits in this case), the assembler will produce an error because it cannot correctly complete the translation from the assembly language program.

To properly translate an assembly language program into machine language, the assembler needs some help from the assembly language programmer. This help comes in the form of pseudo-ops, which are messages to the assembler. Each pseudo-op occupies one line of the assembly language program. Table 13 details these pseudo-ops and their purposes.

The rest of the instruction syntax is best shown by example; please see the next sections for details.

Table 13: LC-2 assembler pseudo-ops

Pseudo-op	Function
.ORIG	Tells the assembler where to load the code in the simulator's memory. The value specified is put as the first word in the object file. It should be the first non-comment line of the source program. The value is usually \$3000 for user programs. <i>Example:</i> <pre> .ORIG \$3000</pre>
.FILL	Directs the assembler to reserve a location and initialize the value of the location to the argument given. Most often used to provide constant values, but also to reserve working memory for your program. <i>Examples:</i> <pre>Label1 .FILL \$0000 ; initialize to 0 Label2 .FILL \$000A ; initialize to #10</pre>
.STRINGZ	Directs the assembler to reserve a number of locations for the specified string. Strings are arranged one character per word, with the high byte of the word initialized to zero; a zero word follows the string to denote its termination (as in the C programming language). Thus, the number of memory locations reserved for the string is equal to one more than the length of the string. <i>Example:</i> <pre>Label3 .STRINGZ "This is a string"</pre>
.BLKW	Directs the assembler to reserve a number of locations and initialize them to the given value. Useful to reserve working space for your program. <i>Example:</i> <pre>Label4 .BLKW 10 \$0000</pre>
.END	Signifies the end of the source code. There should be no more text after this pseudo-op. <i>Example:</i> <pre> .END</pre>

3.6.3. The Assembly Process

Recall that to provide the convenience of assembly language and satisfy the need of the computer to have its instructions in machine code, some intermediate translational steps are needed to translate the assembly language to machine code.

For the LC-2, the assembler does all the necessary steps. It converts the text of the assembly language source code into several formats. The first two are the textual versions of the hex and binary machine code which you have already seen. These are provided for the programmer's benefit, because they are easy to read.

No machine executes programs in this textual format. You might notice that if you print out an executable file on a Windows, Macintosh, or Unix workstation, you generally get a lot of non-printable characters and beeps. These 'garbage' characters are just the result of converting the textual version into true binary. So, the binary 0010 1010 0101 1110 becomes `*^`. The assembler provides output in true binary (the `.obj` file) which the simulator can read directly.

3.6.4. Some Examples

The program in Figure 2 is shown below in the easier-to-read assembly format:

Figure 4: The program of Figure 2 shown in assembly language

```
; dumbadd.asm
; This LC2 program performs the following operations:
;     1) input two characters
;     2) adds the characters
;     3) output the result

        .ORIG $3000          ; directive: program load location
; Get the character inputs
; the IN service routine provides a prompt for each character
        IN                  ; read character into R0
        ADD R1, R0, #0      ; move R0 into R1
        IN                  ; read character into R0

; do the addition
        ADD R0, R0, R1      ; add numbers

; write the results
        OUT                  ; write the single-character result
        HALT
        .END                ; directive: no more code
```

As a point of style, notice how each line is commented and arranged using tabs to make it easy to read.

After writing the assembly language program, the next step is to assemble it. This is demonstrated in "The assemble Program" on page 54.

The example in Figure 3 is now reproduced in assembly language.

Figure 5: The program of Figure 3 shown in assembly language

```
; reverse.asm
; This LC2 program performs the following operations:
;     1) input up to 10 characters
;     2) print out the characters in reverse order
; If more than 10 characters are typed, the program may break.
; R0 is the current character being read or printed
; R1 is a temporary
; R2 is the count of characters that have been read in so far
; R3 is the address into the buffer (same as bufptr)

        .ORIG $3000                ; directive: program load location

; Set up
        lea R3, Buffer              ; R3 points to start of buffer
        and R2, R2, #0             ; zero out character count

; Read characters and store them
READ    in                          ; read a character from keyboard
        add R1, R0, $-A            ; subtract ASCII value of ENTER key
        brz PRINTIT               ; if ENTER pressed, we're done with
                                   ; input
        str R0, R3, #0             ; store R0 (character) into buffer
        add R3, R3, #1             ; increment the buffer pointer
        add R2, R2, #1             ; increment character count
        jmp READ                   ; read the next character

PRINTIT
        add R3, R3, #-1            ; move pointer back to last
                                   ; character read

; Print the characters in reverse order
PRINT   and R2, R2, R2             ; and R2 to itself to set the flags
        brz DONE                  ; if R2 is 0, no more to print
        ldr R0, R3, #0             ; load character pointed to by R3
        out                         ; print the character
        add R2, R2, #-1            ; decrement character count
        add R3, R3, #-1            ; decrement the buffer pointer
        jmp PRINT                  ; print the next character

DONE    HALT

; Memory fills
Buffer  .BLKW 10 $0000            ; 10 character buffer
        .END                       ; directive: no more code
```

*LC-2 Programming Tools
Guide*

This chapter discusses the use of the assembler and simulator software and how to run LC-2 programs using the simulator.

4.1. The `convert` Program

The `convert` program is used to translate programs written in machine language into a binary format acceptable to the LC-2 simulator. It transforms your input text consisting of 0's and 1's (or hexadecimal text) into true binary (machine code) format. Typing:

```
unix% convert -b2 myfile outfile
```

tells `convert` to assume that the program was written in binary (`-b` stands for base and 2 means base 2, or binary). `myfile` is your program written in 0's and 1's, and `outfile` is where the binary output will be written. If the `-b2` is left off, it is assumed. You can use `-b16` if your program is written in hexadecimal notation. If you type the command incorrectly, `convert` will display a message explaining its usage rules.

Note that it is necessary to convert your text input into binary format in order for the simulator to run your program.

4.2. The assemble Program

To assemble a program into a real binary machine code file, type:

```
unix% assemble myfile.asm outfile
```

This command will cause the assembler to read the assembly code from `myfile.asm` and produce several output files, detailed in Table 14:

Table 14: LC-2 assembler output files

Output Filename	Purpose
<code>outfile.obj</code>	The binary executable (for loading into the simulator)
<code>outfile.lst</code>	Lists the assembly language program with line numbers and the text of the code produced
<code>outfile.bin</code>	Textual representation (binary notation) of executable.
<code>outfile.hex</code>	Textual representation (hexadecimal notation) of executable.
<code>outfile.sym</code>	Lists the symbol table

4.2.1. The Listing File

Listing 6 shows an example of a listing file produced by the assembler. The first column shows the hexadecimal page address of the instruction or data on that line. Notice that 0000 is duplicated: this is the assembler's way of showing that the first line (3000 in this case) is not really part of the program, but indicates to the simulator where to load the program in memory. The second column shows the hexadecimal of the instruction or data for that line. The third column shows the equivalent binary. The fourth column lists the line number of the original source file, and the fifth column shows the code that was entered at that line number in the source file.

Figure 6: Assembler-produced listing file for `dumbadd.asm`

```
(0000) 3000 0011000000000000 ( 11) .orig 0x3000
(0000) F023 1111000000100011 ( 13) in
(0001) 1220 0001001000100000 ( 14) add r1 r0 0x0000
(0002) F023 1111000000100011 ( 15) in
(0003) 1001 0001000000000001 ( 18) add r0 r0 r1
(0004) F021 1111000000100001 ( 21) out
(0005) F025 1111000000100101 ( 22) halt
```

4.3. The Simulator

The LC-2 simulator is a program that allows you to run LC-2 programs. Typing the command:

```
unix% simulate
```

causes a representation of the simulator to appear on the monitor as shown in Figure 7.

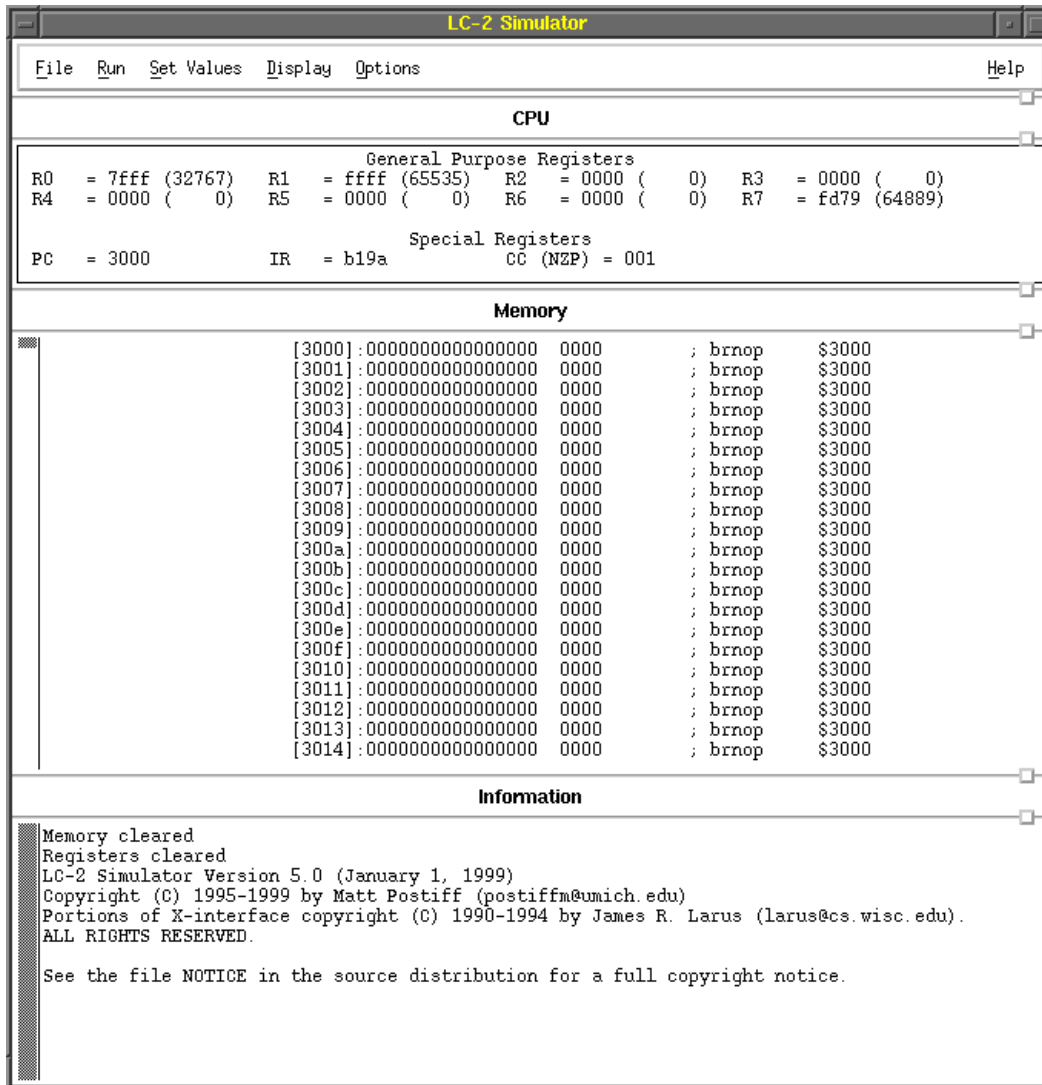
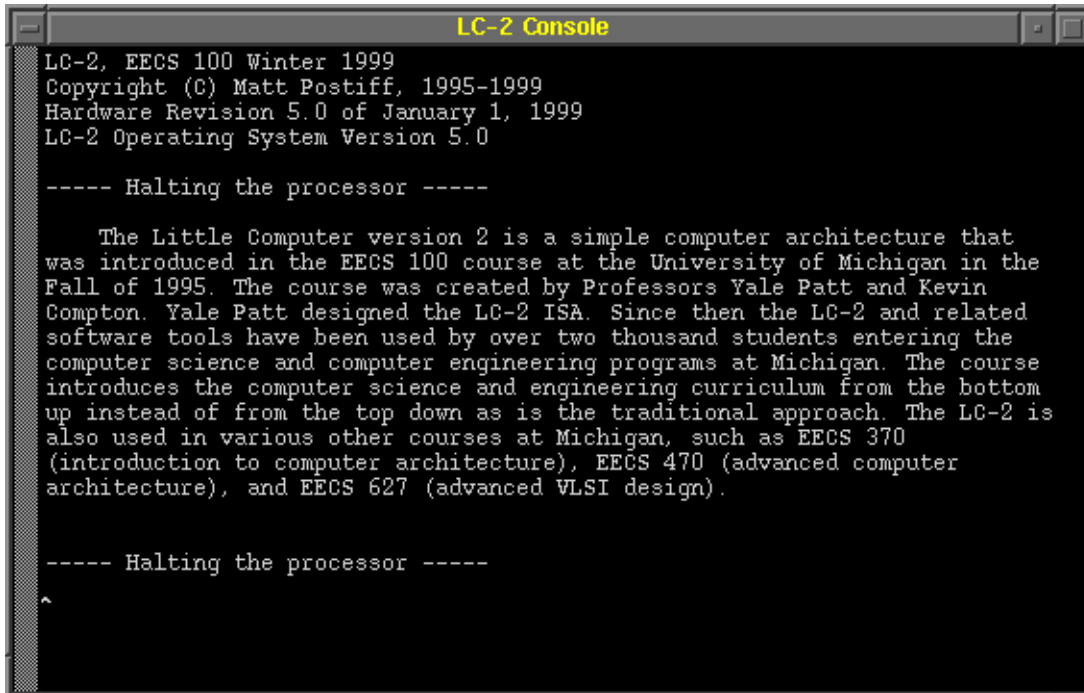


Figure 7: The LC-2 simulator

The usual X-window controls such as `-fg`, `-bg`, and `-geometry` can be passed to **simulate** to modify its default appearance.

Also appearing on your monitor is the LC-2's console, where input and output (`in` and `out` instructions) occurs for the simulated machine. Figure 8 shows the console upon simulator startup.



```
LC-2 Console
LC-2, EECS 100 Winter 1999
Copyright (C) Matt Postiff, 1995-1999
Hardware Revision 5.0 of January 1, 1999
LC-2 Operating System Version 5.0

----- Halting the processor -----

The Little Computer version 2 is a simple computer architecture that
was introduced in the EECS 100 course at the University of Michigan in the
Fall of 1995. The course was created by Professors Yale Patt and Kevin
Compton. Yale Patt designed the LC-2 ISA. Since then the LC-2 and related
software tools have been used by over two thousand students entering the
computer science and computer engineering programs at Michigan. The course
introduces the computer science and engineering curriculum from the bottom
up instead of from the top down as is the traditional approach. The LC-2 is
also used in various other courses at Michigan, such as EECS 370
(introduction to computer architecture), EECS 470 (advanced computer
architecture), and EECS 627 (advanced VLSI design).

----- Halting the processor -----
^
```

Figure 8: The LC-2 console window

When the simulator is started, the file `lc2.log` is created. After the simulation, this file will contain a copy of all the output that occurred during the session.

The simulator window is organized into four major sections. The top section is a menu which allows control over the simulated LC-2. The menu bar will be discussed in detail in the next section.

The second section, labeled ‘CPU’ shows the current state of the CPU registers. The contents of the general purpose registers are shown in both their hexadecimal and unsigned decimal forms. The condition code (CC) register is displayed as its component N (negative), Z (zero), and P (positive) bits.

The third section, labeled ‘Memory’ shows the contents of memory. Each line shows the address and binary and hexadecimal contents of the memory being displayed. To the right (after the semicolon), an instruction is displayed. If this memory location were executed as an instruction, this is what would be executed. Of course, some locations are data, so the instruction interpretation would not make much sense. A status column is present at the far left of the memory window. A ‘B’ in this column indicates that a breakpoint is set at that line in memory. Between the single status column and the memory address, there may be a label. This label is taken from the symbol file from the assembler. (See “The assemble Program” on page 54.) Finally, note that the memory window is currently limited to displaying addresses in the range `0x3000..0x3200`. To display more memory, use the `Display|Memory` menu option.

The fourth section, labeled ‘Information’ provides miscellaneous information and helpful messages.

4.3.1. The Simulator Menu Bar

This section describes the simulator menu bar.

The menu bar is shown in Figure 9 below. The menu bar has options to control the simulator's behavior, such as how and when to load a program, run it, and display memory and register values. The following paragraphs and figures describe these options in more detail.



Figure 9: The LC-2 Simulator menu bar.

Figure 10 shows the File menu and dialog boxes that appear when the various options are selected. The File menu contains commands to specify to the simulator to load a program or script, to run a single user-specified script command, or to exit the simulator.

The first two options, load a program or script, display the same standard open file dialog box that appears in many X-window applications. The dialog box allows the user to navigate the directory structure and locate a file. Selecting the file and clicking OK will load the program or script. Scripts are described in Appendix B and are not considered further in this chapter. The simulator expects programs to be in the `.obj` file format, as described in Section 4.1 and Section 4.2. (The file name need not end in `.obj`, but this is our convention.)

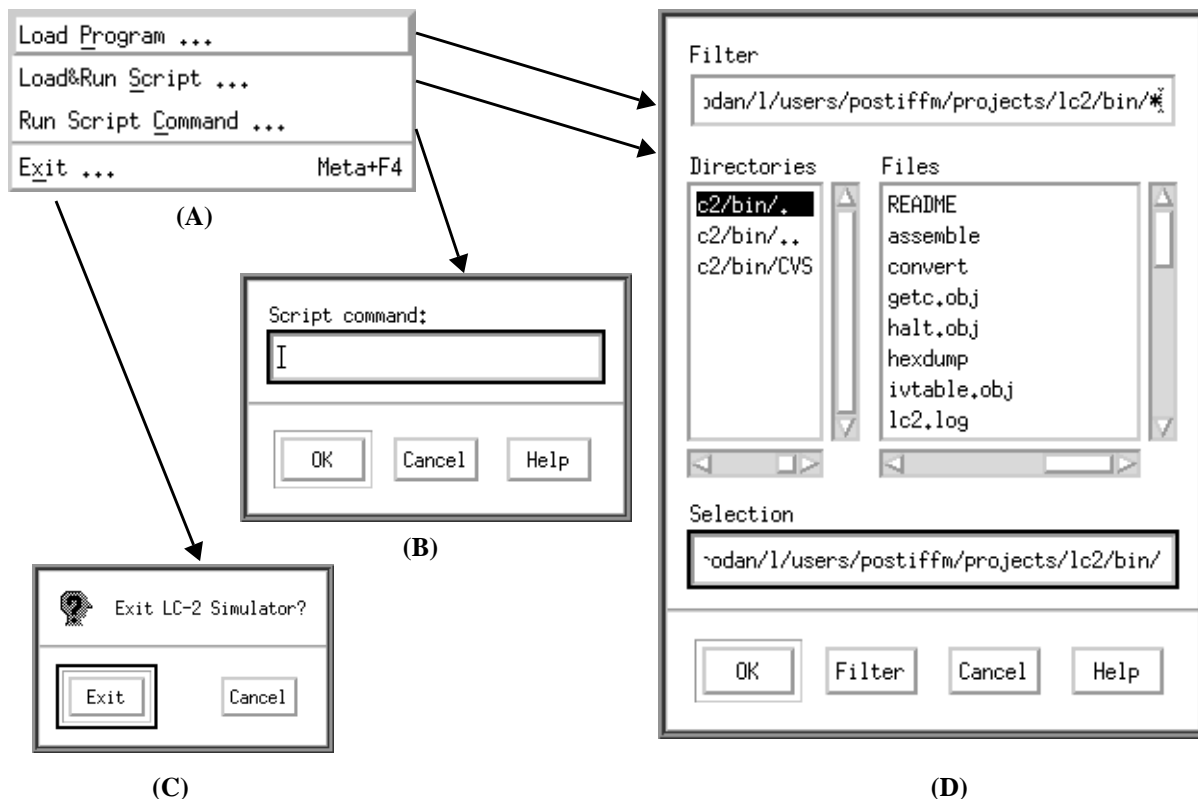


Figure 10: The LC-2 Simulator File menu (A) and associated dialog boxes (B-D).

Figure 11 shows the Run menu and the dialog boxes for each menu item. The Run menu contains commands to control the execution of a program in the simulator. The “Run program” option displays a dialog box which allows the user to specify the program starting address (by default, 0x3000). The program begins execution when the Run button is clicked.

The “Step Program” option is similar, except it assumes the PC is pointing to the starting instruction and asks the user for the number of instructions to step (default is 1 instruction). Clicking the Step button will execute one instruction. This mode is useful for debugging.

Finally, the “Breakpoints” option allows the user to add or remove breakpoints. For example, if a breakpoint is added at address 0x3002, each time the simulator encounters the instruction at 0x3002, it stops execution so the user can examine the LC-2 register and memory state. This is useful for debugging when a bug only appears after many instructions have executed and it is too onerous to single step the whole program..

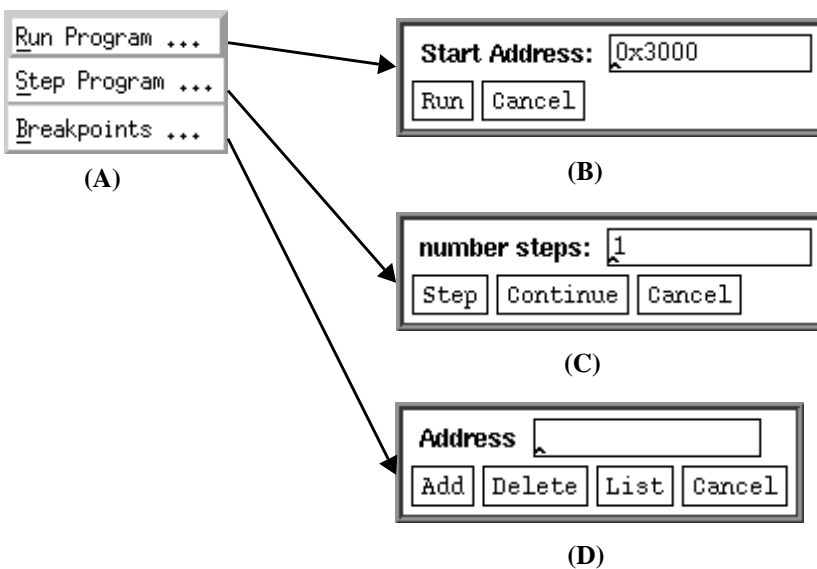


Figure 11: The LC-2 Simulator Run menu (A) and associated dialog boxes (B-D).

Figure 12 shows the Set Values Menu and a dialog box for one of the menu items. The Set Values menu contains commands to change the values of memory locations or registers while a simulation is running. This is useful when single-stepping a program, for example, because during execution, if a register or memory value is wrong, the programmer can change it to the correct value and determine if the remainder of the program works properly.

The first option, “Reinitialize machine,” completely reinitializes the simulator, clears registers and memory, and reloads the operating system routines and interrupt vector table. The “Clear Registers” option clears only the registers.

The third option, “Set Register or memory,” allows the programmer to set any register or memory location to a specified value. Values must be specified in the same notation used by the assembler (0x3000, #10, and so on).

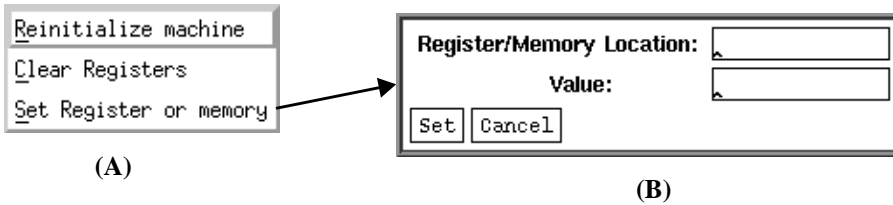


Figure 12: The LC-2 Simulator Set Values menu (A) and an associated dialog box (B).

Figure 13 shows the Display Menu and one associated dialog box. The Display menu contains commands to change the range of memory displayed in the Memory window of the simulator, as well as dump the register contents to the Information section and refresh the display if it somehow was corrupted (sometimes this happens on a Unix console window).

The Display|Memory dialog box, shown in Figure 13, allows the user to change what range of memory is displayed in the Memory window. The From: and To: boxes take LC-2 memory addresses. When the Print button is selected, the Memory section displays that region of memory. Whenever a program is loaded, the Memory section displays memory starting from the program’s starting address.

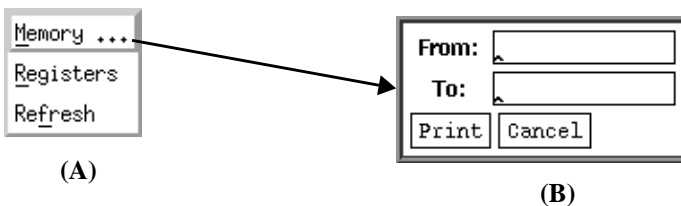


Figure 13: The LC-2 Simulator Display menu (A) and an associated dialog box (B).

Figure 14 shows the Options and Help menus. The Options menu provides two simulator options. Both are toggle switches, whose ON state is indicated by a depressed button appearing next to the menu item. Thus, in Figure 14(A), the simulator is set to step into traps but not to dump an instruction trace.

The first simulator option, “Step into traps,” tells the simulator whether the user would like to view system call code one instruction at a time while stepping through it. This is primarily useful for debugging system call routines, which you hopefully will not have to do!

The second option, “Dump instruction trace,” instructs the simulator to print a message for each instruction it executes. This can be useful for debugging.

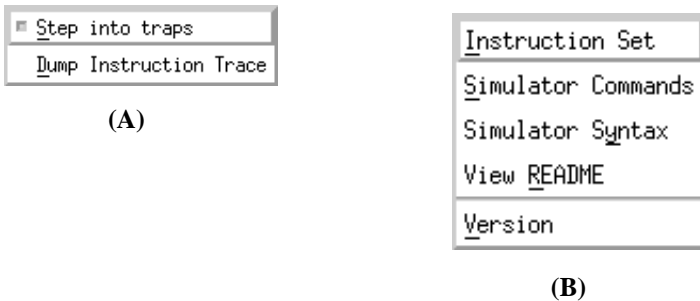


Figure 14: The LC-2 Simulator Options (A) and Help menus (B).

The Help menu, Figure 14(B), contains various commands to display help messages in the simulator Information section. The Instruction Set option prints information about the mnemonics and opcodes supported by the simulator. The Simulator Commands option gives a summary of the scripting language. The scripting language, which is described in Appendix B, is useful for debugging and also automatic grading of LC-2 programs. The Simulator Syntax option describes the invocation options that can be used when the simulator is started at the command line. The View README option prints the README file to the information section. The README file may contain information that is newer than present in this manual. The last option, Version, prints copyright and version information.

Figure 15 shows the right-mouse button popup menu. The popup menu contains commands for the commonly-used functions in the simulator. These options have been described above. The menu is accessed by single-clicking the right mouse button inside the simulator main window or the console window.



Figure 15: The LC-2 Simulator Popup menu.

This section contains an ASCII chart, information on automating the LC-2 simulator using scripts, and installation instructions for the LC-2 software.

Appendix A. ASCII Reference

Following is the ASCII character set shown in octal, decimal, and hexadecimal. C language ‘\x’ escapes are noted.

Table 15: ASCII Code Reference

Octal	Decimal	Hex	Character	Octal	Decimal	Hex	Character
000	0	00	NUL ‘\0’	040	32	20	SPACE
001	1	01	SOH	041	33	21	!
002	2	02	STX	042	34	22	“
003	3	03	ETX	043	35	23	#
004	4	04	EOT	044	36	24	\$
005	5	05	ENQ	045	37	25	%
006	6	06	ACK	046	38	26	&
007	7	07	BEL ‘\a’	047	39	27	‘
010	8	08	BS ‘\b’	050	40	28	(
011	9	09	HT ‘\t’	051	41	29)
012	10	0A	LF ‘\n’	052	42	2A	*
013	11	0B	VT ‘\v’	053	43	2B	+
014	12	0C	FF ‘\f’	054	44	2C	,
015	13	0D	CR ‘\r’	055	45	2D	-
016	14	0E	SO	056	46	2E	.
017	15	0F	SI	057	47	2F	/
020	16	10	DLE	060	48	30	0
021	17	11	DC1	061	49	31	1
022	18	12	DC2	062	50	32	2
023	19	13	DC3	063	51	33	3
024	20	14	DC4	064	52	34	4
025	21	15	NAK	065	53	35	5
026	22	16	SYN	066	54	36	6
027	23	17	ETB	067	55	37	7
030	24	18	CAN	070	56	38	8
031	25	19	EM	071	57	39	9
032	26	1A	SUB	072	58	3A	:
033	27	1B	ESC	073	59	3B	;
034	28	1C	FS	074	60	3C	<
035	29	1D	GS	075	61	3D	=
036	30	1E	RS	076	62	3E	>
037	31	1F	US	077	63	3F	?

Table 15: ASCII Code Reference

Octal	Decimal	Hex	Character	Octal	Decimal	Hex	Character
100	64	40	@	140	96	60	`
101	65	41	A	141	97	61	a
102	66	42	B	142	98	62	b
103	67	43	C	143	99	63	c
104	68	44	D	144	100	64	d
105	69	45	E	145	101	65	e
106	70	46	F	146	102	66	f
107	71	47	G	147	103	67	g
110	72	48	H	150	104	68	h
111	73	49	I	151	105	69	i
112	74	4A	J	152	106	6A	j
113	75	4B	K	153	107	6B	k
114	76	4C	L	154	108	6C	l
115	77	4D	M	155	109	6D	m
116	78	4E	N	156	110	6E	n
117	79	4F	O	157	111	6F	o
120	80	50	P	160	112	70	p
121	81	51	Q	161	113	71	q
122	82	52	R	162	114	72	r
123	83	53	S	163	115	73	s
124	84	54	T	164	116	74	t
125	85	55	U	165	117	75	u
126	86	56	V	166	118	76	v
127	87	57	W	167	119	77	w
130	88	58	X	170	120	78	x
131	89	59	Y	171	121	79	y
132	90	5A	Z	172	122	7A	z
133	91	5B	[173	123	7B	{
134	92	5C	\ `\'	174	124	7C	
135	93	5D]	175	125	7D	}
136	94	5E	^	176	126	7E	~
137	95	5F	_	177	127	7F	DEL

Appendix B. Simulator Script Reference

The simulator supports a simple scripting language to facilitate in the testing and grading of programs. To run a script from the command line, use the following command:

```
unix% simulate "script-command" ["script command" ... ]
```

This will cause the simulator to perform its normal startup activities and run the script commands that you specify. Note that the double-quotes surrounding the script command are necessary. The two most useful commands to run from the command line are:

```
unix% simulate "x script_file"  
unix% simulate "l prog.obj"
```

where **x** in the first example tells the simulator to execute the given script file. A script file contains a sequence of script commands, one per line. Each command must be less than 80 characters long. Currently, no other characters (including comments) are allowed in the script file. The syntax for addresses and values are the same as those input through the graphical interface.

The second command shows how to automatically load (**l**) a program into the simulator upon startup.

The simulator writes output to the Information window and copies any data from the Information window to a file called `lc2.log` in the simulator's working directory. Thus the following script allows us to automatically load a program, run it to completion, and dump the register values and a memory location to `lc2.log`:

```
l prog1.obj  
g 3000  
r  
m 3014
```

Please see the help provided within the simulator (from the Help|Commands menu item) to get a current listing of script commands.

Appendix C. Installing the LC-2 Software¹

An installation script has been provided to set up an `eeecs100` directory and install the LC-2 assembler and simulator in your CAEN home directory. Add the EECS 100 software directory to your path so that you can run the various software tools:

```
unix% set path = ($path /afs/engin.umich.edu/class/perm/eeecs100/bin)
```

You will probably want to add this command to your `.cshrc` file so you don't have to do it manually every time you log in. Then, type:

```
unix% lc2install
```

This will create a directory for you to do classwork in. When you want to assemble or simulate a file, type the object or assembly source file using your favorite text editor, assemble it, and simulate it all within your `eeecs100` directory.

The remainder of this manual uses the notation `unix%` to represent the Unix command prompt. We assume that `~/eeecs100` is the current working directory. If you are not familiar with the Unix computing environment, ask your teaching assistant or consult CAEN for Unix documentation.

1. These instructions are specific to the University of Michigan Computer Aided Engineering computing environment.