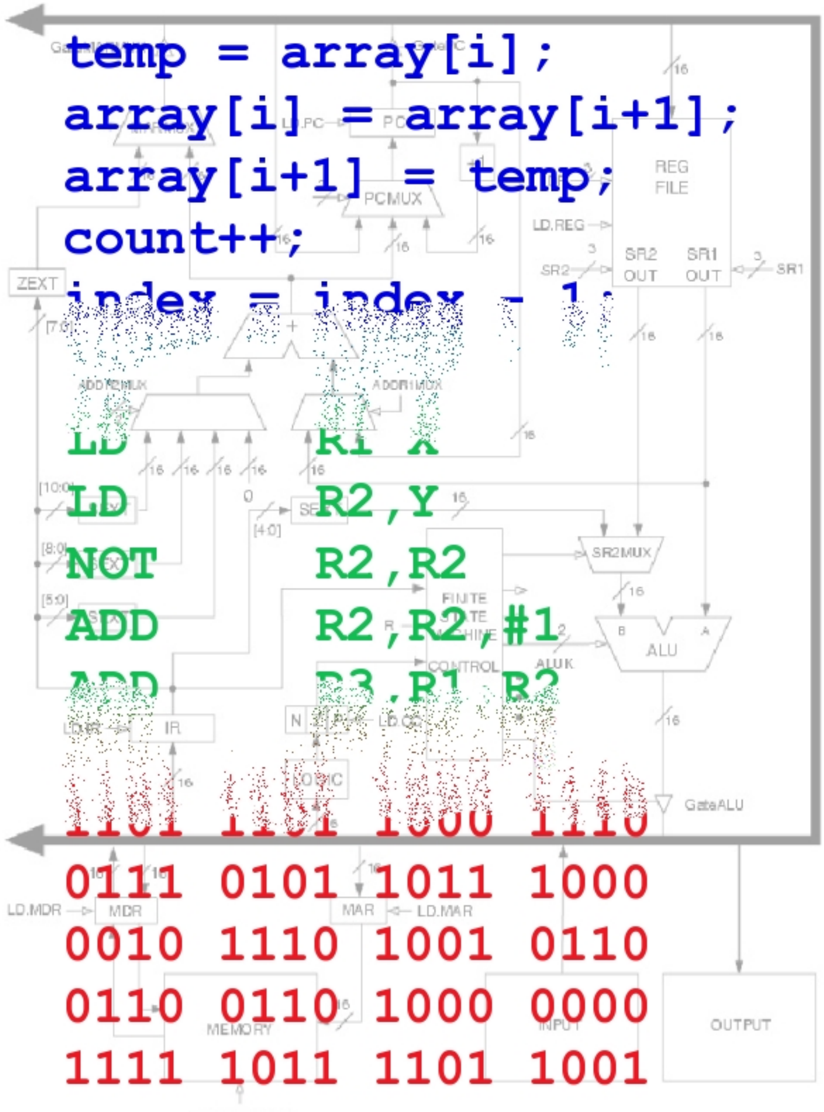
LC-3 Assembly Language
A Manual



George M. Georgiou and **Brian Strader**

California State University, San Bernardino

August 2005

CONTENTS

LIST OF CODE LISTINGS

LIST OF FIGURES

Programming in LC-3

Parts of an LC-3 Program

```
1 ; LC-3 Program that displays
2 ; "Hello World!" to the console
3         .ORIG     x3000
4         LEA       R0, HW  ; load address of string
5         PUTS              ; output string to console
6         HALT              ; end program
7 HW      .STRINGZ "Hello World!"
8         .END
```

Listing 1: “Hello World!” in LC-3.

The above listing is a typical hello world program written in LC-3 assembly language. The program outputs “Hello World!” to the console and quits. We will now look at the composition of this program.

Lines 1 and 2 of the program are comments. LC-3 uses the semi-colon to denote the beginning of a comment, the same way C++ uses “//” to start a comment on a line. As you probably already know, comments are very helpful in programming in high-level languages such as C++ or Java. You will find that they are even more necessary when writing assembly programs. For example in C++, the subtraction of two numbers would only take one statement, while in LC-3 subtraction usually takes three instructions, creating a need for further clarity through commenting.

Line 3 contains the .ORIG pseudo-op. A pseudo-op is an instruction that you can use when writing LC-3 assembly programs, but there is no corresponding instruction in LC-3’s instruction set. All pseudo-ops start with a period. The best way to think of pseudo-ops are the same way you would think of preprocessing directives in C++. In C++, the #include statement is really not a C++ statement, but it is a directive that helps a C++ complier do its job. The .ORIG pseudo-op, with its numeric parameter, tells the assembler where to place the code in memory.

Memory in LC-3 can be thought of as one large 16-bit array. This array can hold LC-3 instructions or it can hold data values that those instructions will manipulate. The standard place for code to begin at is memory location x3000. Note that the “x” in front of the number indicates it is in hexadecimal. This means that the “.ORIG x3000” statement will put “LEA R0, HW” in memory location x3000, “PUTS” will go into memory location x3001, “HALT” into memory location x3002, and so on until the entire program has been placed into memory. All LC-3 programs begin with the .ORIG pseudo-op.

Lines 4 and 5 are LC-3 instructions. The first instruction, loads the address of the “Hello World!”

string and the next instruction prints the string to the console. It is not important to know how these instructions actually work right now, as they will be covered in the labs.

Line 6 is the HALT instruction. This instruction tells the LC-3 simulator to stop running the program. You should put this in the spot where you want to end your program.

Line 7 is another pseudo-op .STRINGZ. After the main program code section, that was ended by HALT, you can use the pseudo-ops, .STRINGZ, .FILL, and .BLKW to save space for data that you would like to manipulate in the program. This is a similar idea to declaring variables in C++. The .STRINGZ pseudo-op in this program saves space in memory for the "Hello World!" string.

Line 8 contains the .END pseudo-op. This tells the assembler that there is no more code to assemble. This should be the very last instruction in your assembly code file. .END can be sometimes confused with the HALT instruction. HALT tells the simulator to stop a program that is running. .END indicates where the assembler should stop assembling your code into a program.

Syntax of an LC-3 Instruction

Each LC-3 instruction appears on line of its own and can have up to four parts. These parts in order are the label, the opcode, the operands, and the comment.

Each instruction can start with a label, which can be used for a variety of reasons. One reason is that it makes it easier to reference a data variable. In the hello world example, line 7 contains the label "HW." The program uses this label to reference the "Hello World!" string. Labels are also used for branching, which are similar to labels and goto's in C++. Labels are optional and if an instruction does not have a label, usually empty space is left where one would be.

The second part of an instruction is the opcode. This indicates to the assembler what kind of instruction it will be. For example in line 4, LEA indicates that the instruction is a load effective address instruction. Another example would be ADD, to indicate that the instruction is an addition instruction. The opcode is mandatory for any instruction.

Operands are required by most instructions. These operands indicate what data the instruction will be manipulating. The operands are usually registers, labels, or immediate values. Some instructions like HALT do not require operands. If an instruction uses more than one operand like LEA in the example program, then they are separated by commas.

Lastly an instruction can also have a comment attached to it, which is optional. The operand section of an instruction is separated from the comment section by a semicolon.

LC-3 Memory

LC-3 memory consists of 2^{16} locations, each being 16 bits wide. Each location is identified with an address, a positive integer in the range 0 through $2^{16} - 1$. More often we use 4-digit hexadecimal numbers for the addresses. Hence, addresses range from x0000 to xFFFF.

The LC-3 memory with its various regions is shown in figure 1 on page ix.

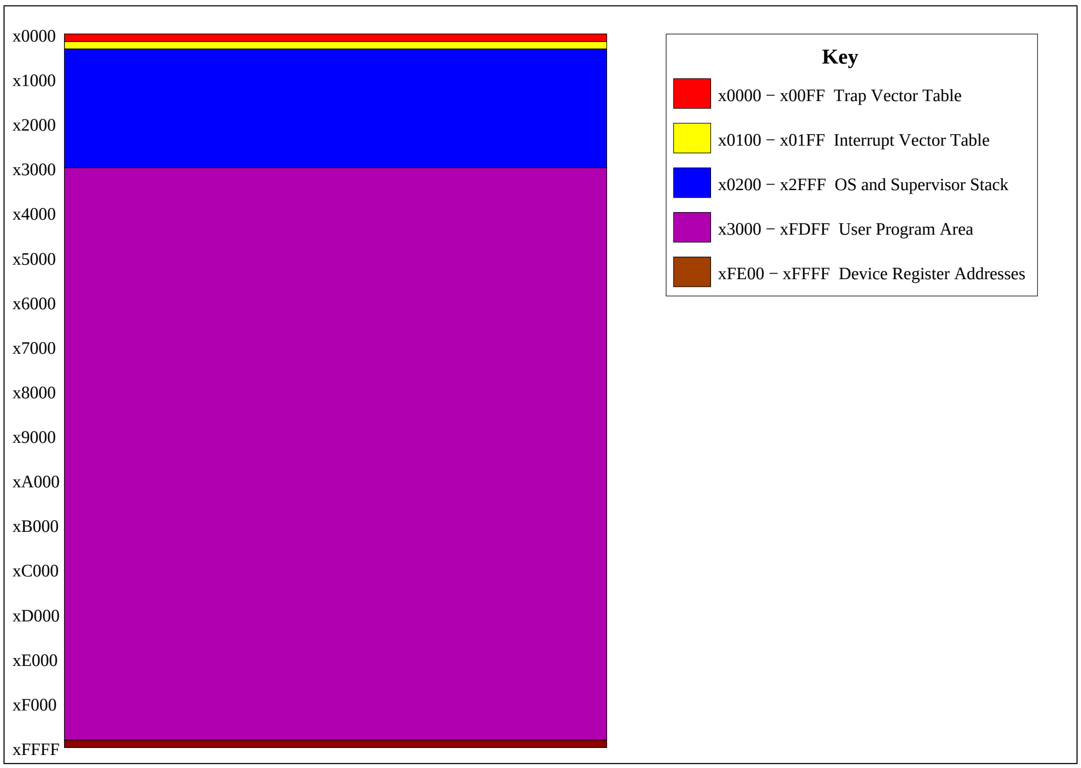


Figure 1: LC-3 memory map: the various regions.

LC3 Quick Reference Guide

Instruction Set

Op	Format	Description	Example
ADD	ADD DR, SR1, SR2 ADD DR, SR1, imm5	Adds the values in SR1 and SR2/imm5 and sets DR to that value.	ADD R1, R2, #5 The value 5 is added to the value in R2 and stored in R1.
AND	AND DR, SR1, SR2 AND DR, SR1, imm5	Performs a bitwise and on the values in SR1 and SR2/imm5 and sets DR to the result.	AND R0, R1, R2 A bitwise and is preformed on the values in R1 and R2 and the result stored in R0.
BR	BR(n/z/p) LABEL Note: (n/z/p) means any combination of those letters can appear there, but must be in that order.	Branch to the code section indicated by LABEL, if the bit indicated by (n/z/p) has been set by a previous instruction. n: negative bit, z: zero bit, p: positive bit. Note that some instructions do not set condition codes bits.	BRz LPBODY Branch to LPBODY if the last instruction that modified the condition codes resulted in zero. BRnp ALT1 Branch to ALT1 if last instruction that modified the condition codes resulted in a positive or negative (non-zero) number.
JMP	JMP SR1	Unconditionally jump to the instruction based upon the address in SR1.	JMP R1 Jump to the code indicated by the address in R1.
JSR	JSR LABEL	Put the address of the next instruction after the JSR instruction into R7 and jump to the subroutine indicated by LABEL.	JSR POP Store the address of the next instruction into R7 and jump to the subroutine POP.
JSRR	JSSR SR1	Similar to JSR except the address stored in SR1 is used instead of using a LABEL.	JSSR R3 Store the address of the next instruction into R7 and jump to the subroutine indicated by R3's value.
LD	LD DR, LABEL	Load the value indicated by LABEL into the DR register.	LD R2, VAR1 Load the value at VAR1 into R2.
LDI	LDI DR, LABEL	Load the value indicated by the address at LABEL's memory location into the DR register.	LDI R3, ADDR1 Suppose ADDR1 points to a memory location with the value x3100. Suppose also that memory location x3100 has the value 8. 8 then would be loaded into R3.
LDR	LDR DR, SR1, offset6	Load the value from the memory location found by adding the value of SR1 to offset6 into DR.	LDR R3, R4, #-2 Load the value found at the address (R4 –2) into R3.
LEA	LEA DR, LABEL	Load the address of LABEL into DR.	LEA R1, DATA1 Load the address of DATA1 into R1.
NOT	NOT DR, SR1	Performs a bitwise not on SR1 and stores the result in DR.	NOT R0, R1 A bitwise not is preformed on R1 and the result is stored in R0.
RET	RET	Return from a subroutine using the value in R7 as the base address.	RET Equivalent to JMP R7.

RTI	RTI	Return from an interrupt to the code that was interrupted. The address to return to is obtained by popping it off the supervisor stack, which is automatically done by RTI.	RTI Note: RTI can only be used if the processor is in supervisor mode.
ST	ST SR1, LABEL	Store the value in SR1 into the memory location indicated by LABEL.	ST R1, VAR3 Store R1's value into the memory location of VAR3.
STI	STI SR1, LABEL	Store the value in SR1 into the memory location indicated by the value that LABEL's memory location contains.	STI R2, ADDR2 Suppose ADDR2's memory location contains the value x3101. R2's value would then be stored into memory location x3101.
STR	STR SR1, SR2, offset6	The value in SR1 is stored in the memory location found by adding SR2 and offest6 together.	STR R2, R1, #4 The value of R2 is stored in memory location (R1 + 4).
TRAP	TRAP trapvector8	Performs the trap service specified by trapvector8. Each trapvector8 service has its own assembly instruction that can replace the trap instruction.	TRAP x25 Calls a trap service to end the program. The assembly instruction HALT can also be used to replace TRAP x25.

Symbol Legend

Symbol	Description	Symbol	Description
SR1, SR2	Source registers used by instruction.	LABEL	Label used by instruction.
DR	Destination register that will hold the instruction's result.	trapvector8	8 bit value that specifies trap service routine.
imm5	Immediate value with the size of 5 bits.	offset6	Offset value with the size of 6 bits.

TRAP Routines

Trap Vector	Equivalent Assembly Instruction	Description
x20	GETC	Read one input character from the keyboard and store it into R0 without echoing the character to the console.
x21	OUT	Output character in R0 to the console.
x22	PUTS	Output null terminating string to the console starting at address contained in R0.
x23	IN	Read one input character from the keyboard and store it into R0 and echo the character to the console.
x24	PUTSP	Same as PUTS except that it outputs null terminated strings with two ASCII characters packed into a single memory location, with the low 8 bits outputted first then the high 8 bits.
x25	HALT	Ends a user's program.

Pseudo-ops

Pseudo-op	Format	Description
.ORIG	.ORIG #	Tells the LC-3 simulator where it should place the segment of code starting at address #.
.FILL	.FILL #	Place value # at that code line.
.BLKW	.BLKW #	Reserve # memory locations for data at that line of code.
.STRINGZ	.STRINGZ "<String>"	Place a null terminating string <String> starting at that location.
.END	.END	Tells the LC-3 assembler to stop assembling your code.

LAB 1

ALU Operations

1.1 Problem Statement

The numbers X and Y are found at locations **x3100** and **x3101**, respectively. Write an LC-3 assembly language program that does the following.

- Compute the sum $X+Y$ and place it at location **x3102**.
- Compute X **AND** Y and place it at location **x3103**.
- Compute X **OR** Y and place it at location **x3104**.
- Compute **NOT**(X) and place it at location **x3105**.
- Compute **NOT**(Y) and place it at location **x3106**.
- Compute $X+3$ and place it at location **x3107**.
- Compute $Y-3$ and place it at location **x3108**.
- If the X is even, place 0 at location **x3109**. If the number is odd, place 1 at the same location.

The operations **AND**, **OR**, and **NOT** are bitwise. The operation signified by **+** is the usual arithmetic addition.

1.1.1 Inputs

The numbers X and Y are in locations **x3100** and **x3101**, respectively:

x3100	X
x3101	Y

1.1.2 Outputs

The outputs at their corresponding locations are as follows:

x3102	$X+Y$
x3103	X AND Y
x3104	X OR Y
x3105	NOT(X)
x3106	NOT(Y)
x3107	$X+3$
x3108	$Y-3$
x3109	Z

where Z is defined as

$$Z = \begin{cases} 0 & \text{if } X \text{ is even} \\ 1 & \text{if } X \text{ is odd.} \end{cases} \tag{1.1}$$

1.2 Instructions in LC-3

LC-3 has available these ALU instructions: **ADD** (arithmetic addition), **AND** (bitwise and), **NOT** (bitwise not).

1.2.1 Addition

Adding two integers is done using the **ADD** instruction. In listing 1.1, the contents of registers **R1** and **R2** and added and the result is placed in **R3**. Note the values of integers can be negative as well, since they are in two's complement format. **ADD** also comes in *immediate* version, where the second operand can be a constant integer. For example, we can use it to add 4 to register **R1** and place the result in register **R3**. See listing 1.1. The constant is limited to 5 bits two's complement format. Note, as with all other ALU instructions, the same register can serve both as a source operand and the destination register.

```
; Adding two registers
      ADD R3, R1, R2 ; R3 ← R1 + R2
; Adding a register and a constant
      ADD R3, R1, #4 ; R3 ← R1 + 4
; Adding a register and a negative constant
      ADD R3, R1, #−4 ; R3 ← R1 − 4
; Adding a register to itself
      ADD R1, R1, R1 ; R1 ← R1 + R1
```

Listing 1.1: The **ADD** instruction.

1.2.2 Bitwise AND

Two registers can be bitwise **AND**ed using the **AND** instruction, as in listing 1.2 on page 1–3. **AND** also comes in the *immediate* version. Note that an immediate operand can be given in hexadecimal form using x followed by the number.

1.2.3 Bitwise NOT

The bits of a register can be inverted (flipped) using the bitwise **NOT** instruction, as in listing 1.3 on page 1–3.

```
; Anding two registers
        AND R3, R1, R2 ; R3 ← R1 AND R2
; Anding a register and a constant
        ADD R3, R1, xA ; R3 ← R1 AND 0000000000001010
```

Listing 1.2: The **AND** instruction.

```
; Inverting the bits of register R1
        NOT R2, R1 ; R2 ← NOT(R1)
```

Listing 1.3: The **NOT** instruction.

1.2.4 Bitwise OR

LC-3 does not provide the bitwise **OR** instruction. We can use, however, **AND** and **NOT** to built it. For this purpose, we make use of De Morgan's rule: X OR Y = NOT(NOT(X) AND NOT(Y)). See listing 1.4.

```
; ORing two registers
        NOT R1, R1      ; R1 ← NOT(R1)
        NOT R2, R2      ; R2 ← NOT(R2)
        AND R3, R1, R2  ; R3 ← NOT(R1) AND NOT(R2)
        NOT R3, R3      ; R3 ← R1 OR R2
```

Listing 1.4: Implementing the **OR** operation.

1.2.5 Loading and storing with LDR and STR

The instruction **LDR** can be used to load the contents of a memory location into a register. Knowing that X and Y are at locations **x3100** and **x3101**, respectively, we can use the code in listing 1.5 on page 1–4 to load them in registers **R1** and **R3**, respectively. In the same figure one can see how the instruction **STR** is used store the contents of a register to a memory location. The instruction **LEA R2, Offset** loads register **R2** with the address (**PC + 1 + Offset**), where **PC** is the address of the instruction **LEA** and **Offset** is a numerical value, i.e. the immediate operand. Figure 1.2 on page 1–5 shows the steps it takes to execute the **LEA R2, xFF** instruction.

If instead of a numerical value, a label is given, such as in instruction **LEA R2, LABEL**, then the value of the immediate operand, i.e. the offset, is automatically computed so that **R2** is loaded with the address of the instruction with label **LABEL**.

1.3 How to determine whether an integer is even or odd

In binary, when a number is even it ends with a 0, and when it is odd, it ends with a 1. We can obtain 0 or 1, correspondingly, by using the **AND** instruction as in listing 1.6 on page 1–4. This method is valid for numbers in two's complement format, which includes negative numbers.

1.4 Testing

Test your program for several input pairs of X and Y. In figure 1.1 on page 1–4 an example is shown of how memory should look after the program is run. The contents of memory are shown in decimal,

```
; Values X and Y are loaded into registers R1 and R3.
        .ORIG x3000 ; Address where program code begins
; R2 is loaded with the beginning address of the data
        LEA R2, xFF ; R2 ← x3000 + x1 + xFF (= x3100)
; X, which is located at x3100, is loaded into R1
        LDR R1, R2, x0 ; R1 ← MEM[x3100]
; Y, which is located at x3101, is loaded into R3
        LDR R3, R2, x1 ; R3 ← MEM[x3100 + x1]
        ...
; Storing 5 in memory location x3101
        AND R4, R4, x0 ; Clear R4
        ADD R4, R4, x5 ; R4 ← 5
        STR R4, R2, x1 ; MEM[x3100 + x1] ← R4
```

Listing 1.5: Loading and storing examples.

```
        AND R2, R1, x0001 ; R2 has the value of the least
                          ; significant bit of R1.
```

Listing 1.6: Determining whether a number is even or odd.

hexadecimal, and binary format.

Address	Decimal	Hex	Binary	Contents
x3100	9	0009	0000 0000 0000 1001	X
x3101	-13	FFF3	1111 1111 1111 0011	Y
x3102	-4	FFFC	1111 1111 1111 1100	$X+Y$
x3103	1	0001	0000 0000 0000 0001	X AND Y
x3104	-5	FFFB	1111 1111 1111 1011	X OR Y
x3105	65526	FFF6	1111 1111 1111 0110	NOT(X)
x3106	12	000C	0000 0000 0000 1100	NOT(Y)
x3107	12	000C	0000 0000 0000 1100	$X+3$
x3108	-16	FFF0	1111 1111 1111 0000	$Y-3$
x3108	1	0001	0000 0000 0000 0001	z

Figure 1.1: Example run.

1.5 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of the (X,Y) pairs $(10,20), (-11,15), (11,-15), (9,12)$, screenshots that show the contents of location **x3100** through **x3108**.

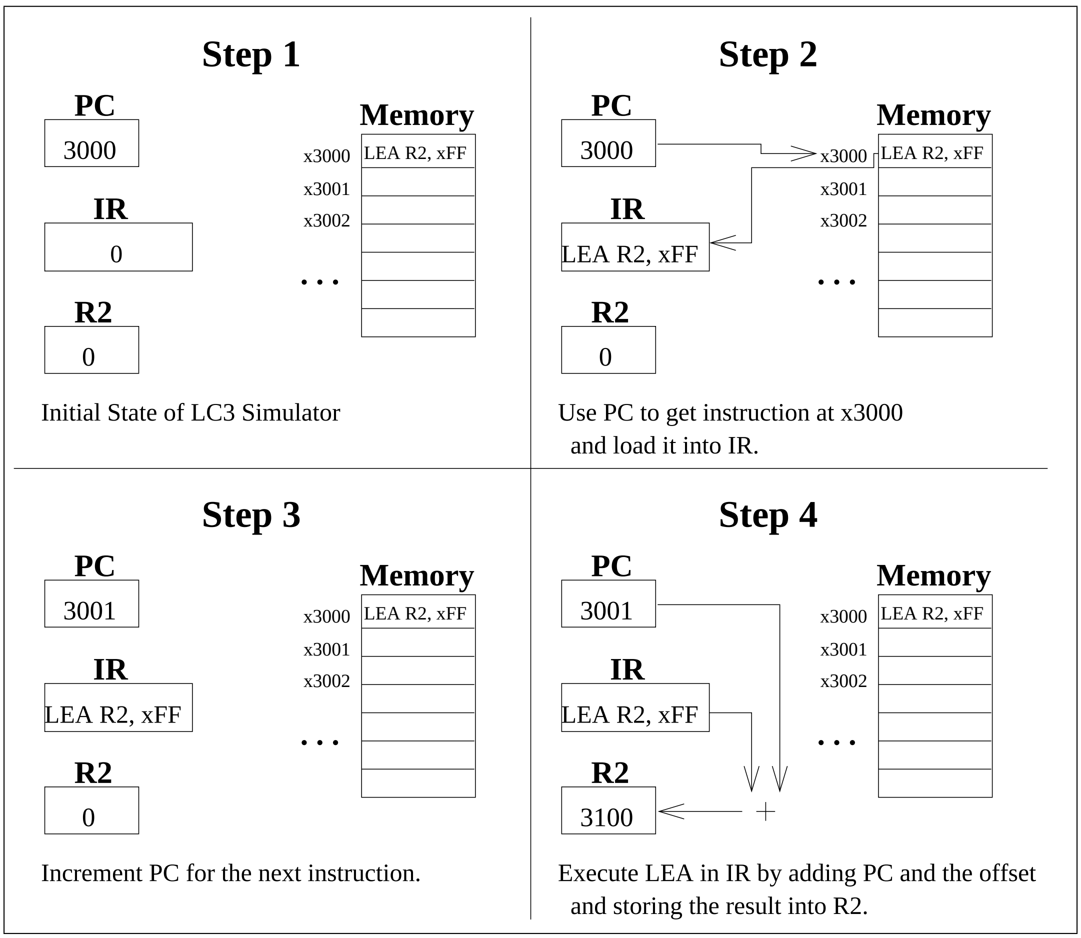


Figure 1.2: The steps taken during the execution of the instruction **LEA R2, xFF**.

LAB 2

Arithmetic functions

2.1 Problem Statement

The numbers X and Y are found at locations **x3120** and **x3121**, respectively. Write a program in LC-3 assembly language that does the following:

- Compute the difference $X - Y$ and place it at location **x3122**.
- Place the absolute values $|X|$ and $|Y|$ at locations **x3123** and **x3124**, respectively.
- Determine which of $|X|$ and $|Y|$ is larger. Place 1 at location **x3125** if $|X|$ is, a 2 if $|Y|$ is, or a 0 if they are equal.

2.1.1 Inputs

The integers X and Y are in locations **x3120** and **x3121**, respectively:

x3120	X
x3121	Y

2.1.2 Outputs

The outputs at their corresponding locations are as follows:

x3122	$X - Y$
x3123	$\|X\|$
x3124	$\|Y\|$
x3125	Z

where Z is defined as

$$Z = \begin{cases} 1 & \text{if } |X| - |Y| > 0 \\ 2 & \text{if } |X| - |Y| < 0 \\ 0 & \text{if } |X| - |Y| = 0 \end{cases} \tag{2.1}$$

2.2 Operations in LC-3

2.2.1 Loading and storing with LDI and STI

In the previous lab, loading and storing was done using the **LDR** and **STR** instructions. In this lab, the similar but distinct instructions **LDI** and **STI** will be used. Number *X* already stored at location **x3120** can be loaded into a register, say, **R1** as in listing 2.1. The *Load Indirect* instruction, **LDI**, is used. The steps taken to execute **LDI R1, X** are shown in figure 2.2 on page 2–5.

```
        LDI R1, X
        ...
        ...
        HALT
        ...
X       .FILL x3120
```

Listing 2.1: Loading into a register.

In listing 2.2, the contents of register **R2** are stored at location **x3121**. The instruction *Store Indirect*, **STI**, is used. The steps taken to execute **STI R2, Y** instruction are shown in figure 2.3 on page 2–5.

```
        STI R2, Y
        ...
        ...
        HALT
        ...
Y       .FILL x3121
```

Listing 2.2: Storing a register.

2.2.2 Subtraction

LC-3 does not provide a subtraction instruction. However, we can build one using existing instructions. The idea here is to negate the subtrahend[1], which is done by taking its two complement, and then adding it to the minuend.

As an example, in listing 2.3 the result of the subtraction $5-3=5+(-3)=2$ is placed in register **R3**. It is assumed that 5 and 3 are already in registers **R1** and **R2**, respectively.

```
; Register R1 has 5 and register R2 has 3
; R4 is used as a temporary register. R2 could have been used
; in the place of R4, but the original contents of R2 would
; have been lost.  The result of 5-3=2 goes into R3.
        NOT R4, R2
        ADD R4, R4, #1  ; R4 ← -R2
        ADD R3, R1, R4  ; R3 ← R1 - R2
```

Listing 2.3: Subtraction: $5-3=2$.

[1] Subtrahend is a quantity which is subtracted from another, the minuend.

2.2.3 Branches

The usual linear flow of executing instructions can be altered by using branches. This enables us to choose code fragments to execute and code fragments to ignore. Many branch instructions are conditional which means that the branch is taken only if a certain condition is satisfied. For example the instruction **BRz TARGET** means the following: if the result of a previous instruction was zero, the next instruction to be executed is the one with label **TARGET**. If the result was not zero, the instruction that follows **BRz TARGET** is executed and execution continues as normal.

The exact condition for a branch instructions depends on three *Condition Bits:* **N (negative), Z (zero), and P (positive)**. The value (0 or 1) of each condition bit is determined by the nature of the result that was placed in a destination register of an earlier instruction. For example, in listing 2.4 we note that at the execution of the instruction **BRz LABEL** N is 0, and therefore the branch is not taken.

```
        ...
        AND R1, R1, x0   ; Since R1 ← 0, N = 0, Z = 1, P = 0
        ADD R2, R1, x1   ; Since R2 ← 1, N = 0, Z = 0, P = 1
        BRz LABEL
        ...
LABEL   ...
```

Listing 2.4: Condition bits are set.

Table figure 2.1 shows a list of the available versions of the branch instruction. As an example

BR	branch unconditionally	BRnz	branch if result was negative or zero
BRz	branch if result was zero	BRnp	branch if result was negative or positive
BRn	branch if result was negative	BRzp	branch if result was zero or positive
BRp	branch is result was positive	BRnzp	branch unconditionally

Figure 2.1: The versions of the BR instruction.

consider the code fragment in listing 2.5. The next instruction after the branch instruction to be executed will be the **ADD** instruction, since the result placed in **R2** was 0, and thus bit **Z** was set. The **NOT** instruction, and the ones that follow it up to the instruction before the **ADD** will never be executed.

```
       AND R2, R5, x0 ; result placed in R2 is zero
       BRz TARGET     ; Branch if result was zero (it was)
       NOT R1, R3
       ...
       ...
TARGET ADD R5, R1, R2
       ...
```

Listing 2.5: Branch if result was zero.

2.2.4 Absolute value

The absolute value of an integer X is defined as follows:

$$|X| = \begin{cases} X & \text{if } X \geq 0 \\ -X & \text{if } X < 0. \end{cases} \tag{2.2}$$

One way to implement absolute value is seen in listing 2.6.

```
; Input:  R1 has value X.
; Output: R2 has value |X|.
        ADD R2, R1, #0   ; R2 ← R1, can now use condition codes
        BRzp ZP          ; If zero or positive, do not negate
        NOT R2, R2
        ADD R2, R2, #1   ; R2 = −R1
ZP      ...              ; At this point R2 = |R1|
        ...
```

Listing 2.6: Absolute value.

2.3 Example

At the end of a run, the memory locations of interest might look like this:

x3120	9
x3121	-13
x3122	22
x3123	9
x3124	13
x3125	2

2.4 Testing

Test your program for these X and Y pairs:

X	Y
10	12
13	10
-10	12
10	-12
-12	-12

Figure 2.4 on page 2–6 is table that shows the binary representations the integers -32 to 32, that can helpful in testing.

2.5 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of the (X,Y) pairs $(10,20),(-11,15),(11,-15),(12,12)$, screenshots that show the contents of location **x3120** through **x3125**.

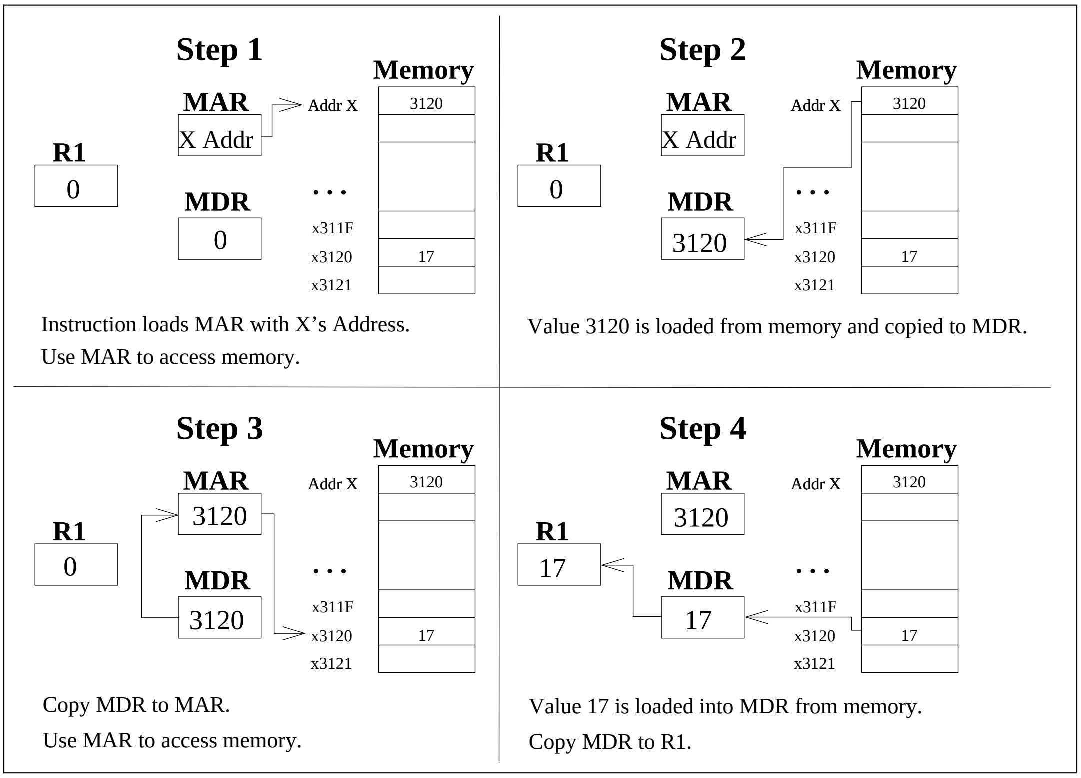


Figure 2.2: The steps taken during the execution of the instruction **LDI R1, X**.

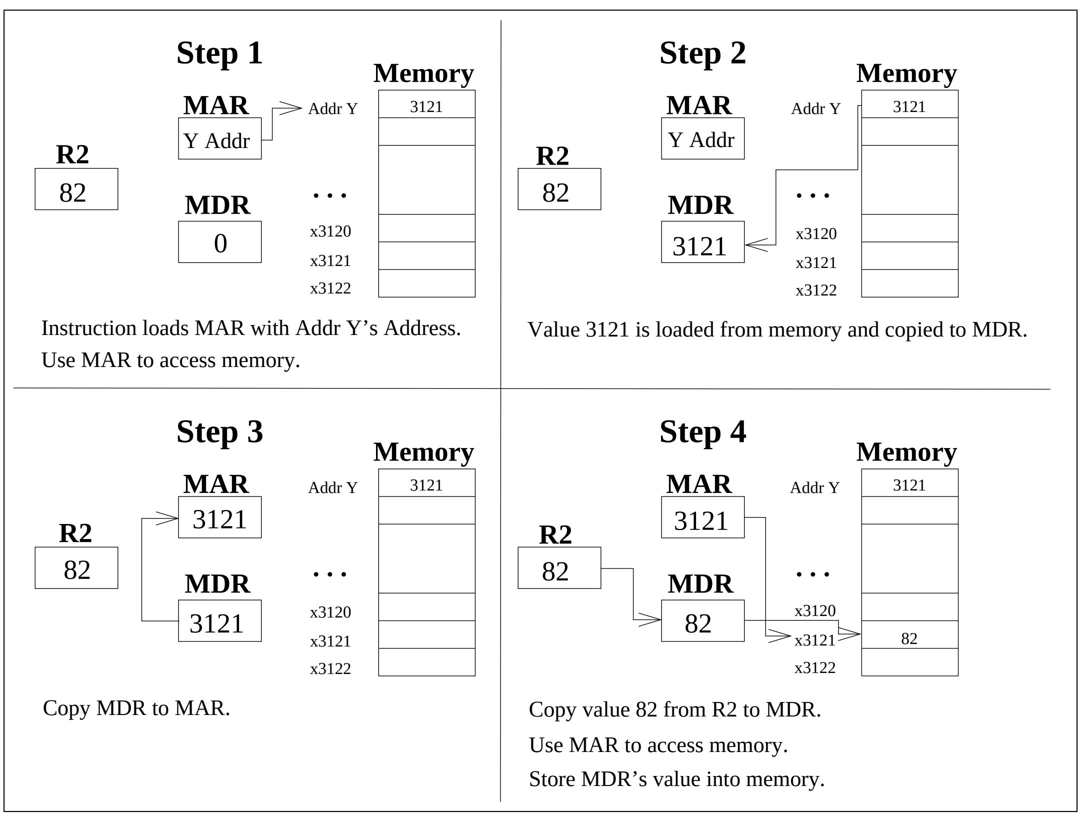


Figure 2.3: The steps taken during the execution of the instruction **STI R2, Y**.

Decimal	2's Complement	Decimal	2's Complement
0	0000000000000000	-0	0000000000000000
1	0000000000000001	-1	1111111111111111
2	0000000000000010	-2	1111111111111110
3	0000000000000011	-3	1111111111111101
4	0000000000000100	-4	1111111111111100
5	0000000000000101	-5	1111111111111011
6	0000000000000110	-6	1111111111111010
7	0000000000000111	-7	1111111111111001
8	0000000000001000	-8	1111111111111000
9	0000000000001001	-9	1111111111110111
10	0000000000001010	-10	1111111111110110
11	0000000000001011	-11	1111111111110101
12	0000000000001100	-12	1111111111110100
13	0000000000001101	-13	1111111111110011
14	0000000000001110	-14	1111111111110010
15	0000000000001111	-15	1111111111110001
16	0000000000010000	-16	1111111111110000
17	0000000000010001	-17	1111111111101111
18	0000000000010010	-18	1111111111101110
19	0000000000010011	-19	1111111111101101
20	0000000000010100	-20	1111111111101100
21	0000000000010101	-21	1111111111101011
22	0000000000010110	-22	1111111111101010
23	0000000000010111	-23	1111111111101001
24	0000000000011000	-24	1111111111101000
25	0000000000011001	-25	1111111111100111
26	0000000000011010	-26	1111111111100110
27	0000000000011011	-27	1111111111100101
28	0000000000011100	-28	1111111111100100
29	0000000000011101	-29	1111111111100011
30	0000000000011110	-30	1111111111100010
31	0000000000011111	-31	1111111111100001
32	0000000000100000	-32	1111111111100000

Figure 2.4: Decimal numbers with their corresponding 2's complement representation

LAB 3

Days of the week

3.1 Problem Statement

- Write a program in LC-3 assembly language that keeps prompting for an integer in the range 0-6, and each time it outputs the corresponding name of the day. If a key other than '0' through '6' is pressed, the program exits.

3.1.1 Inputs

At the prompt "**Please enter number:** ," a key is pressed.

3.1.2 Outputs

If the key pressed is '0' through '6', the corresponding name of the day of the week appears on the screen. Precisely, the correspondence is according to this table:

Code	**Day**
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

When the day is displayed, the prompt "**Please enter number:** " appears again and the program expects another input. If any key other that '0' through '6' is pressed, the program exits.

3.2 The lab

3.2.1 Strings in LC-3

It will be necessary to define the prompt "**Please enter number:** " and the days of the week as strings in memory. All strings should terminate with the NUL character (ASCII 0). In LC-3 one character per memory location is stored. Each location is 16 bits wide. The 8 most significant bits are 0, while the 8 least significant bits hold the ASCII value of the character. Strings terminated with the NUL character can be conveniently defined using the directive **.STRINGZ "ABC"** , where

"ABC" is any alphanumeric string. It automatically appends the NUL character to the string. As an example, a string defined in assembly language and the corresponding contents of memory are shown in figure 3.1.

```
        .ORIG  x3100
        .STRINGZ  "Sunday"
```

```
x3100  0053  ;   S
x3101  0075  ;   u
x3102  006e  ;   n
x3103  0064  ;   d
x3104  0061  ;   a
x3105  0079  ;   y
x3106  0000  ;  NUL
```

Figure 3.1: The string "Sunday" in assembly and its corresponding binary representation

3.2.2 How to output a string on the display

To output is a string on the screen, one needs to place the beginning address of the string in register **R0**, and then call the **PUTS** assembly command, which is another name for the instruction **TRAP x22**. For example, to output "ABC", one can do the following:

```
          LEA R0, ABCLBL ; Loads address of ABC string into R0
          PUTS
          ...
          HALT
          ...
ABCLBL    .STRINGZ  "ABC"
          ...
```

The **PUTS** command calls a system trap routine which outputs the NUL terminated string the address of its first character is found in register **R0**.

3.2.3 How to read an input value

The assembly command **GETC**, which is another name for **TRAP x20**, reads a single character from the keyboard and places its ASCII value in register **R0**. The 8 most significant bits of **R0** are cleared. There is no echo of the read character. For example, one may use the following code to read a single numerical character, 0 through 9, and place its value in register **R3**:

```
          GETC          ; Place ASCII value of input character into R0
          ADD R3, R0, x0   ; Copy R0 into R3
          ADD R3, R3, #-16; Subtract 48, the ASCII value of 0
          ADD R3, R3, #-16
          ADD R3, R3, #-16; R3 now contains the actual value
```

Notice that it was necessary to use three instructions to subtract 48, since the maximum possible value of the immediate operand of ADD is 5 bits, in two's complement format. Thus, -16 is the most we can subtract with the immediate version of the **ADD** instruction. As an example, if the pressed key was "5", its ASCII value 53 will be placed in **R0**. Subtracting 48 from 53, the value 5 results, as expected, and is placed in register **R3**.

3.2.4 Defining the days of the week

For ease of programming one may define the days of the week so the they have the same length. We note that "Wednesday" has the largest string length: 9. As a NUL terminated string, it occupies 10 locations in memory. In listing 3.1 define all days so that they have the same length.

```
            ...
            HALT
            ...
DAYS       .STRINGZ "Sunday   "
           .STRINGZ "Monday   "
           .STRINGZ "Tuesday  "
           .STRINGZ "Wednesday"
           .STRINGZ "Thursday "
           .STRINGZ "Friday   "
           .STRINGZ "Saturday "
```

Listing 3.1: Days of the week data.

If the numerical code for a day is i (a value in the range 0 through 6, see section 7.1.2 on page 7–1), the address of the corresponding day is found by this formula:

$$\text{Address\_of(DAYS)} + i * 10 \tag{3.1}$$

Address_of(DAYS) is the address of label **DAYS**, which is the beginning address of the string "Sunday." Since LC-3 does not provide multiplication, one has to implement it. One can display the day that corresponds to i by means of the code in listing 3.2, which includes the code of listing 3.1. Register **R3** is assumed to contain i.

```
            ...
; R3 already contains the numerical code of the day i
            LEA  R0, DAYS        ; Address of "Sunday" in R0
            ADD  R3, R3, x0      ; To be able to use condition codes
; The loop (4 instructions) implements R0 ← R0 + 10 * i
LOOP        BRz  DISPLAY
            ADD  R0, R0, #10     ; Go to next day
            ADD  R3, R3, #-1     ; Decrement loop variable
            BR   LOOP
DISPLAY     PUTS
            ...
            HALT
            ...
DAYS       .STRINGZ "Sunday   "
           .STRINGZ "Monday   "
           .STRINGZ "Tuesday  "
           .STRINGZ "Wednesday"
           .STRINGZ "Thursday "
           .STRINGZ "Friday   "
           .STRINGZ "Saturday "
```

Listing 3.2: Display the day.

3.3 Testing

Test the program with all input keys '0' through '6' to make sure the correct day is displayed, and with several keys outside that range, to ascertain that the program terminates.

3.4 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of the input $i = 0, 1, 4, 6$, screenshots that show the output.

LAB 4

Fibonacci Numbers

4.1 Problem Statement

1. Write a program in LC-3 assembly language that computes F_n, the $n-$th Fibonacci number.
2. Find the largest F_n such that no overflow occurs, i.e. find $n = N$ such that F_N is the largest Fibonacci number to be correctly represented with 16 bits in two's complement format.

4.1.1 Inputs

The integer n is in memory location **x3100**:

x3100	n

4.1.2 Outputs

x3101	F_n
x3102	N
x3103	F_N

4.2 Example

x3100	6
x3101	8
x3102	N
x3103	F_N

Starting with 6 in location **x3100** means that we intend to compute F_6 and place that result in location **x3101**. Indeed, $F_6 = 8$. (See below.) The actual values of N and F_N should be found by your program, and be placed in their corresponding locations.

4.3 Fibonacci Numbers

The Fibonacci F_i numbers are the members of the Fibonacci sequence: $1, 1, 2, 3, 5, 8, \ldots$. The first two are explicitly defined: $F_1 = F_2 = 1$. The rest are defined according to this recursive formula: $F_n = F_{n-1} + F_{n-2}$. In words, each Fibonacci number is the sum of the two previous ones in the Fibonacci sequence. From the sequence above we see that $F_6 = 8$.

4.4 Pseudo-code

Quite often algorithms are described using *pseudo-code*. Pseudo-code is not real computer language code in the sense that it is not intended to be compiled or run. Instead, it is intended to describe the steps of algorithms at a high level so that they are easily understood. Following the steps in the pseudo-code, an algorithm can be implemented to programs in a straight forward way. We will use pseudo-code[1] in some of the labs that is reminiscent of high level languages such as C/C++, Java, and Pascal. As opposed to C/C++, where group of statements are enclosed the curly brackets "{" and "}" to make up a compound statement, in the pseudo-code the same is indicated via the use of indentation. Consecutive statements that begin at the same level of indentation are understood to make up a compound statement.

4.5 Notes

- Figure 4.1 is a schematic of the contents of memory.

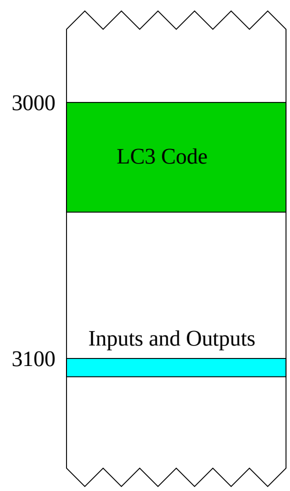


Figure 4.1: Contents of memory

- The problem should be solved by iteration using loops as opposed to using recursion.
- The pseudo-code for the algorithm to compute F_n is in listing 4.1. It is assumed that $n > 0$.

```
if n ≤ 2 then
   F ← 1
else
   a ← 1  // F_{n-2}
   b ← 1  // F_{n-1}
   for i ← 3 to n do
      F ← b + a  // F_n = F_{n-1} + F_{n-2}
      a ← b
      b ← F
```

Listing 4.1: Pseudo-code for computing the Fibonacci number F_n iteratively

[1]The pseudo-code is close to the one used in *Fundamentals of Algorithmics* by G. Brassard and P. Bratley, Prentice Hall, 1996.

- The way to detect overflow is to use a similar for-loop to the one in listing 4.1 on page 4–2 which checks when F first becomes negative, i.e. bit 16 becomes 1. See listing 4.2.
  Caution: upon exit from the loop, F does not have the value of F_N. To obtain F_N you have to slightly modify the algorithm in listing 4.2.

```
a ← 1  //  F_{n-2}
b ← 1  //  F_{n-1}
i ← 2  //loop index
repeat
    F ← b + a  //  F_n = F_{n-1} + F_{n-2}
    if F < 0 then
        N = i
        exit
    a ← b
    b ← F
    i ← i + 1
```

Listing 4.2: Pseudo-code for computing the largest $n = N$ such that F_N can be held in 16 bits

4.6 Testing

The table in figure 4.2 on page 4–4 will help you in testing your program.

4.7 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of $n = 15$ and $n = 19$, screen shots that show the contents of locations **x3100, x3101, x3102** and **x3103**, which show the values for F_{15} and F_{19}, respectively, and the values of N and F_N.

n	F_n	F_n in binary
1	1	0000000000000001
2	1	0000000000000001
3	2	0000000000000010
4	3	0000000000000011
5	5	0000000000000101
6	8	0000000000001000
7	13	0000000000001101
8	21	0000000000010101
9	34	0000000000100010
10	55	0000000000110111
11	89	0000000001011001
12	144	0000000010010000
13	233	0000000011101001
14	377	0000000101111001
15	610	0000001001100010
16	987	0000001111011011
17	1597	0000011000111101
18	2584	0000101000011000
19	4181	0001000001010101
20	6765	0001101001101101
21	10946	0010101011000010
22	17711	0100010100101111
23	28657	0110111111110001
24	46368	1011010100100000
25	75025	0010010100010001

Figure 4.2: Fibonacci numbers table

LAB 5

Subroutines: multiplication, division, modulus

5.1 Problem Statement

- Given two integers X and Y compute the product XY (multiplication), the quotient X/Y (integer division), and the modulus $X \pmod{Y}$ (remainder).

5.1.1 Inputs

The integers X and Y are stored at locations **3100** and **3101**, respectively.

5.1.2 Outputs

The product XY, the quotient X/Y, and modulus $X \pmod{Y}$ are stored at locations **3102**, **3103**, and **3104**, respectively. If X,Y inputs are invalid for X/Y and $X \pmod{Y}$ (see section 5.2.5 on page 5–3) place 0 in both locations **3103** and **3104**.

5.2 The program

5.2.1 Subroutines

Subroutines in assembly language correspond to functions in C/C++ and other computer languages: they form a group of code that is intended to be used multiple times. They perform a logical task by operating on parameters passed to them, and at the end they return one or more results. As an example consider the simple subroutine in listing 5.1 on page 5–2 which implements the function $fn = 2n+3$. The integer n is located at **3120**, and the result Fn is stored at location **3121**. Register **R0** is used to pass parameter n to the subroutine, and **R1** is used to pass the return value fn from the subroutine to the calling program.

Execution is transfered to the subroutine using the **JSR** ("jump to subroutine") instruction. This instruction also saves the return address, that is the address of the instruction that follows **JSR**, in register **R7**. See figure 5.1 on page 5–2 for the steps taken during execution of **JSR.** The subroutine terminates execution via the **RET** "return from subroutine" instruction. It simply assigns the return value in **R7** to the **PC**.

The program will have two subroutines: **MULT** for the multiplication and **DIV** for division and modulus.

```
        LDI  R0, N       ; Argument N is now in R0
        JSR  F           ; Jump to subroutine F.
        STI  R1, FN
        HALT
N       .FILL 3120       ; Address where n is located
FN      .FILL 3121       ; Address where fn will be stored.
                         ; Subroutine F begins
F       AND  R1, R1, x0  ; Clear R1
        ADD  R1, R0, x0  ; R1 ← R0
        ADD  R1, R1, R1  ; R1 ← R1 + R1
        ADD  R1, R1, x3  ; R1 ← R1 + 3.  Result is in R1
        RET              ; Return from subroutine
        END
```

Listing 5.1: A subroutine for the function $f(n) = 2n + 3$.

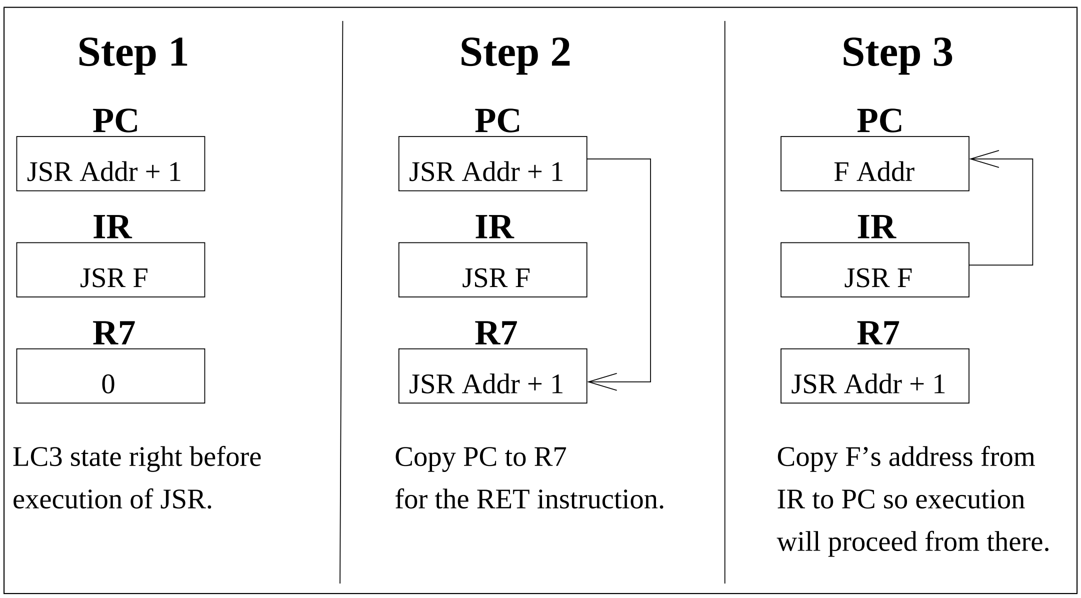


Figure 5.1: The steps taken during execution of **JSR.**

5.2.2 Saving and restoring registers

Make sure that at the beginning of your subroutines you save all registers that will be destroyed in the course of the subroutine. Before returning to the calling program, restore saved registers. As an example, listing 5.2 on page 5–3 shows how to save and restore registers **R5** and **R6** in a subroutine.

5.2.3 Structure of the assembly program

The general structure of the assembly program for this problem can be seen in listing 5.3 on page 5–3.

```
SUB         ...                 ; Subroutine is entered
            ST R5, SaveReg5     ; Save R5
            ST R6, SaveReg6     ; Save R6
            ...                 ; use R5 and R6
            ...

            LD R5, SaveReg5     ; Restore R5
            LD R6, SaveReg6     ; Restore R6
            RET                 ; Back to the calling program
SaveReg5    .FILL x0
SaveReg6    .FILL x0
```

Listing 5.2: Saving and restoring registers **R5** and **R6**.

```
            ...
            JSR MULT;   Jump to the multiplication subroutine
            ...     ;   Here product XY is in R2
            JSR DIV ;   Jump to the division and mod subroutine

            HALT
            ...
            ...     ;   Multiplication subroutine begins
MULT        ...     ;   Save registers that will be overwritten
            ...     ;   Multiplication Algorithm
            ...     ;   Restore saved registers
            ...     ;   R2 has the product.
            RET     ;   Return from subroutine
                    ;   Division and mod subroutine begins
DIV         ...
            ...
            RET
            END
```

Listing 5.3: General structure of assembly program.

5.2.4 Multiplication

Multiplication is achieved via addition:

$$XY = \underbrace{X + X + \ldots + X}_{Y \text{times}} \tag{5.1}$$

Listing 5.4 on page 5–4 shows the pseudo-code for the multiplication algorithm. Parameters X and Y are passed to the multiplication subroutine **MULT** via registers **R0** and **R1**. The result is in **R2**.

5.2.5 Division and modulus

Integer division X/Y and modulus $X \pmod{Y}$ satisfy this formula:

$$X = X/Y * Y + X \pmod{Y} \tag{5.2}$$

Where X/Y is the quotient and $X \pmod{Y}$ is the remainder. For example, if $X = 41$ and $Y = 7$, the equation becomes

$$41 = 5 * 7 + 6 \tag{5.3}$$

```
// Multiplying XY. Product is in variable prod.
sign ← 1  // The sign of the product
if X < 0 then
   X = −X              // Convert X to positive
   sign = −sign
if Y < 0 then
   Y = −Y              // Convert Y to positive
   sign = −sign
prod ← 0               // Initialize product
while Y ≠ 0 do
   prod ← prod + X
   Y ← Y − 1
if sign < 0 then
prod ← −prod           // Adjust sign of product
```

Listing 5.4: Pseudo-code for multiplication.

Subroutine **DIV** will compute both the quotient and remainder. Parameter X is passed to **DIV** through **R0** and Y through **R1**. For simplicity division and modulus are defined only for $X \geq 0$ and $Y > 0$. Subroutine **DIV** should check if these conditions are satisfied. If, not it should return with **R2 = 0**, indicating that the results are not valid. If they are satisfied, **R2 = 1**, to indicate that the results are valid. Overflow conditions need not be checked at this time. Figure 5.2 summarizes the input arguments and results that should be returned.

Register	Input parameter	Result
R0	X	X/Y or 0 if invalid
R1	Y	$X \pmod Y$ or 0 if invalid
R2		1 if results valid, 0 otherwise

Figure 5.2: Input parameters and returned results for **DIV**.

Listing 5.5 shows the pseudo-code for the algorithm that performs integer division and modulus functions. The quotient is computed by successively subtracting Y from X. The leftover quantity is the remainder.

```
// Finding the quotient X/Y and remainder X mod Y.
quotient ← 0    // Initialize quotient
remainder ← 0   // Initialize remainder (in case input invalid)
valid ← 0       // Initialize valid
if X < 0 or Y ≤ 0 then
   exit
valid = 1
temp ← X        // Holds quantity left
while temp ≥ Y do
    temp = temp − Y
    quotient ← quotient + 1
remainder ← temp
```

Listing 5.5: Pseudo-code for integer division and modulus.

5.3 Testing

You should first write the **MULT** subroutine, thoroughly test it, and then proceed to implement the **DIV** subroutine. Thoroughly test **DIV**. Finally, test the program as a whole for various inputs.

5.4 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of the (X,Y) pairs $(100,17),(211,4),(11,-15),(12,0)$, screenshots that show the contents of locations **3100** through **3104**.

LAB 6

Faster Multiplication

6.1 Problem Statement

Write a faster multiplication subroutine using the *shift-and-add* method.

6.1.1 Inputs

The integers X and Y are stored at locations **3100** and **3101**, respectively.

6.1.2 Outputs

The product XY is stored at location **x3102**.

6.2 The program

The program should perform multiplication by subroutine **MULT1**, which is an implementation of the so-called shift-and-add algorithm. Overflow is not checked.

6.2.1 The shift-and-add algorithm

Before giving the algorithm, we consider an example multiplication. We would like to multiply $X = 1101$ and $Y = 101011$. This can be done with the shift-and-add method which resembles multiplication by hand. Figure 6.1 shows the steps. The bold bits are the bits of the multiplier scanned right-to-left. The result is initialized to zero, and then we consider the bits of the multiplier from right to left: if the bit is 1 the multiplicand is added to the product and then shifted to the left by one position. If the bit is 0, the multiplicand is shifted to the left, but no addition is performed.

101011	← Multiplicand
1101	← Multiplier
101011	**1**: Add and shift
1010110	**0**: Shift (not added)
10101100	**1**: Add and shift
101011000	**1**: Add and shift
1000101111	← Result

Figure 6.1: Shift-and-add multiplication

Let $X = x_{15}x_{14}x_{13}\ldots x_1x_0$ and $Y = y_{15}y_{14}y_{13}\ldots y_1y_0$ be the bit representations of multiplier X and multiplicand Y. We would like to compute the product $P = XY$. For the time, we assume that both X and Y are positive, i.e. $x_{15} = y_{15} = 0$. The multiplication algorithm is described in listing 6.1. Recall that in binary, multiplication by 2 is equivalent to a left shift.

```
// Compute product P ← XY
// Y is the multiplicand
// X = x_15 x_14 x_13 ... x_1 x_0 is the multiplier
P ← 0                    // Initialize product
for i=0 to 14 do          // Exclude the sign bit
    if x_i = 1 then
        P ← P + Y        // Add
    Y ← Y + Y            // Shift left
```

Listing 6.1: The shift-and-add multiplication.

6.2.2 Examining a single bit in LC-3

Suppose we would like to check whether the least significant bit (LSB) of **R1** is 0 or 1. We can do that with these instructions:

```
        AND R2, R2, x0    ;
        ADD R2, R2, x1    ; Initialize R2 to 1
        AND R0, R1, R2    ;
        BRz ISZERO        ; Branch if LSB of R1 is 0
        ...
ISZERO  ...
        ...
```

To test the next bit of **R1**, we shift to the left the 1 in **R2** with **ADD R2, R2, R2**, and then again we do:

```
        AND R0, R1, R2    ;
        BRz ISZERO        ; Branch if next bit of R1 is 0
```

We notice that by adding **R2** to itself, the only bit in **R2** that is 1 shifts to the left by one position.

6.2.3 The MULT1 subroutine

Subroutine **MULT1** to be written should be used to perform the multiplication. Parameters X and Y are passed to **MULT1** via registers **R0** and **R1**. The result is in **R2**. The multiplication should work even if the parameters are negative numbers. To achieve this, use the same technique of the algorithm in listing 5.4 on page 5–4 to handle the signs.

Registers that are used in the subroutine should be saved and then restored.

6.3 Testing

Test the **MULT1** subroutine for various inputs, positive and negative.

6.4 What to turn in

- A hardcopy of the assembly source code.

- Electronic version of the assembly code.
- For each of the (X,Y) pairs $(100,17),(-211,-4),(11,-15),(12,0)$, screenshots that show the contents of locations **3100** through **3102**.

LAB 7

Compute Day of the Week

7.1 Problem Statement

Write an LC-3 program that given the day, month and year will return the day of the week.

7.1.1 Inputs

Before execution begins, it is assumed that locations **x31F0, 31F1,** and **x31F2** contain the following inputs:

x31F0	The usual number of the month
x31F1	The day of the month
x31F2	The year

For the example we have been using, **June 1, 2005**, we could use this code fragment in a different module:

```
.ORIG x31F0
.FILL #6
.FILL #1
.FILL #2005
```

7.1.2 Outputs

The outputs are:

- A number between 0 and 6 that corresponds to the days of the week, starting with Sunday, should be stored in location **x31F3**.
- The corresponding name of the day is displayed on the screen.

7.1.3 Example

The program to be written answers this question: what was the day of the week on January 1, 1900?
Answer: Monday

7.2 Zeller's formula

The day of the week can be found by using Zeller's formula[1]:

$$f = k + (13m - 1)/5 + D + D/4 + C/4 - 2C, \tag{7.1}$$

where the symbol "/" represents integer division. For example $9/2 = 4$. Using as example the date **June 1, 2005**, the symbols in the formula have the following meaning:

- k is the day of the month. In the example, $k = 1$.
- m is the month number designated in a special way: March is 1, April is 2, ..., December is 10; January is 11, and February is 12. If x is the usual month number, i.e. for January x is 1, for February x is 2, and so on; then m can be computed with this formula: $m = (x+21)\%12+1$, where % is the usual modulus (i.e. remainder) function. Alternatively, m can be computed in this way:

$$m = \begin{cases} x+10, & \text{if } x \leq 2 \\ x-2, & \text{otherwise.} \end{cases} \tag{7.2}$$

  In our example, $m = 4$.
- D is the last two digits of the year, but if it is January or February those of the previous year are used. In our example, $D = 05$.
- C is for century, and it is the first two digits of year. In our example, $C = 20$.
- From the result f we can obtain the day of the week based on this code:

$f\%7$	Day
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

  For example, if $f = 123$, then $f\%7 = 4$, and thus the day was Thursday. Again, % is the modulus function.

7.3 Subroutines

To compute the modulus (%), integer division (/), and multiplication, subroutines **MULT** and **DIV**, which were written for a previous lab, should be used.

Make sure that **MULT** and **DIV** subroutines save and restore all registers they use, except those that are used to return results. Use **R0** and **R1** to pass parameters, and **R0, R1** and **R2** to return the results.

7.3.1 Structure of program

The general structure of the program appears in listing 7.1 on page 7–3. The problem of displaying the name of the day on the screen was solved in Lab 3.

[1] "Kalender-Formeln" von Rektor Chr. Zeller in Markgröningen, *Mathematisch-naturwissenschaftliche Mitteilungen des mathematisch-naturwissenschaftlichen Vereins in Wrttemberg*, ser. 1, 1 (1885), pp.54-58 – in German.

```
        .ORIG  x3000
        ...
        ...  ; MULT and DIV are called a number of times
        ...
        ...
        PUTS ; Display day of the week on screen
        HALT
DAYS    .STRINGZ "Sunday    "
        .STRINGZ "Monday    "
        .STRINGZ "Tuesday   "
        .STRINGZ "Wednesday"
        .STRINGZ "Thursday  "
        .STRINGZ "Friday    "
        .STRINGZ "Saturday  "
        ...

MULT    ...   ; Beginning of MULT subroutine

        ...
        RET
DIV     ...   ; Beginning of DIV subroutine

        ...
        RET
       .END
```

Listing 7.1: Structure of the program.

7.4 Testing: some example dates

Test your program using these dates:

September 11, 2001	Tuesday
June 6, 1944	Tuesday
September 1, 1939	Friday
November 22, 1963	Friday
August 8, 1974	Thursday

7.5 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of the random dates in the table below, screenshots that show the contents of memory locations **x31F0** through **x31F3**.

Date	Day of the week
January 3, 1905	
June 6, 1938	
June 23, 1941	
May 7, 1961	
Date this lab is due	

LAB 8

Random Number Generator

8.1 Problem Statement

- Generate random numbers using a Linear Congruential Random Number Generator (LCRNG).

8.1.1 Inputs and Outputs

The seed, which is an integer in the range 1 to 32766, is found at location **x3100**. When the program is executed, 20 random numbers in the interval 1 to $2^{15}-2$ are generated and displayed.

8.2 Linear Congruential Random Number Generators

A LCRNG is defined by the this recurrence equation:

$$x_n \leftarrow a\,x_{n-1} + c \mod m \tag{8.1}$$

The multiplicative constant a, the constant c, and modulus m are integers that are chosen and fixed. Given the seed x_0, a random number sequence is generated: $x_1, x_2, x_3, \ldots$, with the x_i's being in the range 0 to $m-1$. Eventually the sequence will repeat itself. In most cases, it is desirable that the period of repetition is as long as possible.

Using the subroutines **MULT** and **DIV**, used in earlier labs, one can write a program in LC-3 to generate random numbers based on equation (8.1). There is, however, the possibility that intermediate operations, such as $a\,x_{n-1}$, cause an overflow. In the case where $c=0$, to avoid overflow we use Schrage's method[1]. In this method, the recurrence is

$$x_n \leftarrow a\,x_{n-1} \mod m, \tag{8.2}$$

and multiplication $a\,x$ is performed in the following fashion:

$$a\,x \mod m = \begin{cases} a\,(x \bmod q) - r\,(x/q) & \text{if } \geq 0 \\ a\,(x \bmod q) - r\,(x/q) + m & \text{otherwise,} \end{cases} \tag{8.3}$$

where

$$q = m/a, \quad r = m \bmod a. \tag{8.4}$$

As always, "/" denotes integer division. To ensure no overflow while performing the computations in equation (8.3), multiplier a and modulus m must be chosen so that $0 \leq r < q$. Listing 8.1 on page 8–2 has the algorithm to generate 20 random numbers.

[1] Schrage, L. 1979, ACM Transactions on Mathematical Software, vol. 5, pp. 132–138.

```
// Algorithm for the iteration x ← a x mod m
// using Schrage's method
a ← 7      // a, the multiplicative constant is given
m ← 32767  // m = 2^15 −1, the modulus is given
x ← 10     // x, the seed is given
q = m/a
r = m mod a
for 1 to 20 do
    x ← a * (x mod q) − r * (x/q)
    if x < 0 then
       x ← x + m
    output x
```

Listing 8.1: Generating 20 random numbers using Schrage's method.

For two's complement 16-bit arithmetic, which is the LC-3 case, the largest possible m is $2^{15}-1$. Using this value for m, to produce a maximal non-repeating sequence[2] of random numbers one can choose $a = 7$. The seed x_0 should never be 0; it should be any number from 1 to $2^{15}-2 = 32766$.

Your program should implement equation (8.2) on page 8–1 with the algorithm found in listing 8.1.

8.3 How to output numbers in decimal

The assembly command **OUT,** which is shorthand for **TRAP x21,** outputs the single ASCII character found in the 8 least significant bits of **R0**. (See listing 8.2 for an example.) We can use **OUT,**

```
; We would like to display in decimal the digit in register R3
; which happens to be negative
          ...
          NOT R3, R3        ; Negate R3 to obtain positive version
          ADD R3, R3, #1
          LD R0, MINUS      ; Output '−'
          OUT
          LD R0, OFFSET     ; Output digit
          ADD R0, R0, R3
          OUT
          ...
          HALT
MINUS   .FILL x2D          ; Minus sign in ASCII
OFFSET  .FILL x30          ; 0 in ASCII
```

Listing 8.2: Displaying a digit.

therefore, to output the decimal digits of a number one by one. We can obtain the digits by successively applying the $\mod 10$ on the number and truncating, until we obtain 0. This produces the digits from right to left. For example if the number we would like to output is $x219 = 537$, by applying the above procedure we obtain the digits in this order: $7,3,5$. Thus, we have to output them in reverse order of their generation. For this purpose we can use a stack, with operations **PUSH** and **POP**.

[2] I.e., all integers in the range 1 to $2^{15}-2$, will be generated before the sequence will repeat itself.

```
// We would like to output n as a decimal
left ← n // remaining value
sign ← 1 // sign of n
if n < 0 then
    sign = −sign   // n is negative
    left ← −n
if left = 0 then
    digit ← 0                  // in case n = 0
    push digit
while left ≠ 0 do
    digit ← left mod 10   // generate a digit
    push digit              // push digit on stack
    left ← left/10
if sign < 0 then
    output '−'              //number is negative
while not(stack_empty) do
    pop digit
    output digit
```

Listing 8.3: Output a decimal number.

8.3.1 A rudimentary stack

The stack that is described here is a rudimentary one[3]. It is intended for this problem only. There are three operations, i.e. subroutines, that involve the stack: **PUSH, POP**, and **ISEMPTY**. **PUSH** pushes the contents of register **R0** on the stack, **POP** pops the top of the stack in register **R0**, and **ISEMPTY** returns 1 in **R0** if the stack is empty and 0 if the stack is non-empty. Register **R6** points to the top of the stack. The following have to be borne in mind when writing your program:

- **R6** should be initialized to $x4000$, the base of the stack, and not be overwritten while manipulating the stack.
- **R7** will be used (implicitly) to store the return address when calling a subroutine.
- Always **ISEMPTY** should be called before proceeding to call **POP**, to check whether the stack is empty. If empty, **POP** should not be called.

Listing 8.4 on page 8–4 shows the implementation of the stack subroutines.

8.4 Testing

Using $a = 7$, $m = 32767$ in equation (8.2) on page 8–1, and starting with various seeds x_0, the first 10 random numbers generated in each case are listed in figure 8.1 on page 8–4.

8.5 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For $a = 7$, $m = 32767$ and seed $x_0 = 100_{10}$, a screenshot showing the first 20 random numbers generated.

[3]For a more sophisticated implementation of a stack see Chapter 10 of the textbook *Introduction to Computing Systems* by Patt and Patel

```
        .ORIG x3000
; Your program goes here
        ...
        ...
        LD R6, BASE         ; Top of stack points to base
        ...
        JSR PUSH            ; Jump to PUSH subroutine
        ...
        HALT                ; Your program ends here
BASE    .FILL x4000
        ...                 ; More program data here
        ...
        ...                 ; Subroutines for stack begin
PUSH    ADD R6, R6, #-1     ; Move top of the stack up
        STR R0, R6, #0      ; Store R0 there
        RET
POP     LDR R0, R6, #0      ; Load R0 with top of stack
        ADD R6, R6, #1      ; Move top of stack down
        RET
ISEMPTY LD R0, EMPTY
        ADD R0, R6, R0
        BRz IS              ; Branch if at base of stack
        ADD R0, R0, #0      ; R0 ← 0, stack is not empty
        RET
IS      AND R0, R0, #0
        ADD R0, R0, #1      ; R0 ← 1, stack is empty
        RET
EMPTY   .FILL xC000         ; -x4000
        END
```

Listing 8.4: The code for the stack.

	$\mathbf{x_0}$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
Decimal	1	7	49	343	2401	16807	19348	4368	30576	17430	23709
Hex	0001	0007	0031	0157	0961	41A7	4B94	1110	7770	4416	5C9D
Decimal	6	42	294	2058	14406	2541	17787	26208	19621	6279	11186
Hex	0006	002A	0126	080A	3846	09ED	457B	6660	4CA5	1887	2BB2
Decimal	9	63	441	3087	21609	20195	10297	6545	13048	25802	16779
Hex	0009	003F	01B9	0C0F	5469	4EE3	2839	1991	32F8	64CA	418B
Decimal	10	70	490	3430	24010	4235	29645	10913	10857	10465	7721
Hex	000A	0046	01EA	0D66	5DCA	108B	73CD	2AA1	2A69	28E1	1E29
Decimal	178	1246	8722	28287	1407	9849	3409	23863	3206	22442	26026
Hex	00B2	04DE	2212	6E7F	057F	2679	0D51	5D37	0C86	57AA	65AA
Decimal	1000	7000	16233	15330	9009	30296	15470	9989	4389	30723	18459
Hex	03E8	1B58	3F69	3BE2	2331	7658	3C6E	2705	1125	7803	481B

Figure 8.1: Sequences of random numbers generated for various seeds x_0.

LAB 9

Recursive subroutines

9.1 Problem Statement

Implement the recursive square function in LC-3 as it is described in section 9.2.4 on page 9–2.

9.1.1 Inputs

The value n is found at location **x3100.**

9.1.2 Output

The value $f(n) = n^2$ is saved at location **x3101.**

9.2 Recursive Subroutines

A subroutine, or function, is recursive when it calls itself. Mathematically, a recursive function is one that is being used in its own definition. In what follows we will give the mathematical definitions of some well-known recursive functions.

9.2.1 The Fibonacci numbers

The Fibonacci numbers F_n, which were encountered in an earlier lab, are defined as follows:

$$F(n) = \begin{cases} n, & \text{if } n \leq 2 \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases} \tag{9.1}$$

Using pseudo-code, the algorithm for F_n is shown in listing 9.1 on page 9–2.

9.2.2 Factorial

The factorial function $f(n) = n!, n \geq 0$, is defined as follows:

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * f(n-1) & \text{if } n > 0. \end{cases} \tag{9.2}$$

```
// Compute the Fibonacci number F(n), n ≥ 1
function F(n)
if n ≤ 2
    return 1
else
    return F(n−1) + F(n−2)
```

Listing 9.1: The pseudo-code for the recursive version of the Fibonacci numbers function.

Non-recursively, the factorial function is defined as follows:

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1) * \ldots * 1, & \text{if } n > 0. \end{cases} \tag{9.3}$$

The first few values of $f(n) = n!$ are shown in figure 9.1.

n	0	1	2	3	4	5	6	7	8	9	10
$n!$	1	1	2	6	24	120	720	5040	40320	362880	3628800

Figure 9.1: The first few values of $f(n) = n!$.

9.2.3 Catalan numbers

Catalan numbers $C_n, n \geq 0$, are defined as follows:

$$C_n \equiv \frac{1}{n+1}\binom{n}{2n} = \frac{(2n)!}{(n+1)!n!}. \tag{9.4}$$

Recursively, the Catalan numbers can be defined as

$$C_{n+1} = \frac{2(2n+1)}{n+2}\, C_n, \tag{9.5}$$

with $C_0 = 1$. An alternative recursive definition is

$$C_n = \begin{cases} 1, & \text{if } n = 0 \\ \sum_{i=0}^{n-1} C_i C_{n-1-i}, & \text{if } n > 0. \end{cases} \tag{9.6}$$

The first few values of C_n are shown in figure 9.2.

n	0	1	2	3	4	5	6	7	8	9	10
C_n	1	1	2	5	14	42	132	429	1430	4862	16796

Figure 9.2: The first few Catalan numbers C_n.

9.2.4 The recursive square function.

The familiar square function $\text{square}(n) = n^2$ can be defined recursively as well:

$$\text{square}(n) = \begin{cases} 0, & \text{if } n = 0 \\ \text{square}(n-1) + 2n - 1, & \text{if } n > 0. \end{cases} \tag{9.7}$$

n	0	1	2	3	4	5	6	7	8	9	10
square(n)	0	1	4	9	16	25	36	49	64	81	100

Figure 9.3: Some values of square(n).

The first few values of square(n) are shown in figure 9.3.

In this lab, you asked to implement the recursive square function as a subroutine, and call it from the main program. Your program should work for negative numbers as well, however the square(n) subroutine should never be called with a negative argument: there will be a *stack overflow,* which is explained in the section that follows. In that and the other sections that follow you will find details that will help you in the implementation of the square(n) subroutine.

9.3 Stack Frames

When a program (or subroutine) A calls a subroutine B with one of either instruction **JSR** and **JSRR**, automatically the return address to A is saved in register **R7**. While executing, if subroutine B calls another subroutine C, then the return address to B will again be saved in **R7**, which would overwrite the previous value. When it is time to return to A, there will be no record of the proper return address. This situation shows the need to have a bookkeeping method that will save return addresses. This need is further demonstrated when having a subroutine that calls itself, i.e. a recursive subroutine. In this case, beyond the return address other information, such as parameters and return value, needs to be allocated for each invocation of the subroutine. The efficient solution to this problem is to have that information saved on a stack.

The space on the stack associated with the invocation of a subroutine is called *frame.* The stack consists of many frames, stacked in the order by which they are called from their corresponding subroutines. If subroutine A calls subroutine, B calls subroutine C, and C calls itself two times, the stack will have the structure of figure 9.4. When a subroutine returns, its corresponding frame is removed from the stack.

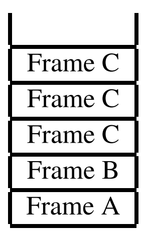


Figure 9.4: The structure of the stack.

A typical frame has the structure in figure 9.5 on page 9–4. The *frame pointer*, also known known as *dynamic link,* points to the first parameter and is used to refer to items within the frame via offsets. Register **R5** is used hold the value of the current frame pointer. The frame pointer of the calling subroutine is saved on the frame of the called subroutine. When the called subroutine returns, the frame pointer is restored in **R5**, and is ready to be used in referring to items within the current frame.

During the execution of a program, while subroutines are called and return, the stack grows and shrinks accordingly. Every time a subroutine is entered, a frame is created; by the time a subroutine returns, all the elements of the frame will have been popped from the stack, and the frame will not exist anymore. If the size of the stack grows too large, i.e. there are too many outstanding subroutines, there is the danger of not having sufficient space to accommodate it, and it will cause an error, which is commonly referred to as *stack overflow.*

The pseudo-code algorithm to implement recursive subroutines is shown in listing 9.2 on page 9–4. It demonstrates how subroutine frames are created on the run-time stack, and destroyed. It is a

Local Variable 2	
Local Variable 1	
Frame Pointer	
Return Address	Frame
Return Value	
Parameter 2	
Parameter 1	

Figure 9.5: A typical frame

summary of the description in the textbook[1].

```
// The calling program
...
PUSH Parameter1          // repeat as needed for additional
                          //parameters
CALL F()                 // jump to F's code
ReturnValue  ← POP       // pop the return value off the stack
POP                      // pop the parameters off the stack,
                          //repeat as needed
...
// The function (subroutine) F

F()                      // beginning of function F
PUSH ReturnValue         // create a place on the stack for the
                          //return value
PUSH ReturnAddress       // push the return address onto the stack
PUSH FramePointer        // push the FramePointer for previous
                          //function
FramePointer ← StackPointer−1    // set the new frame pointer to
                          //the
                                 //location of the first local
                                                        //variable
                                                        //
PUSH LocalVar1           // push local function variables, repeat
                          //as needed
...                                  // function body
LocalVar1 ← POP // pop local variables off the stack, repeat as
                          //needed
FramePointer ← POP     // restore the old frame pointer
ReturnAddress ← POP      // restore ReturnAddress so the caller can
                          // be
                                 //   returned to
return                   // return to the caller, end of F()
```

Listing 9.2: The pseudo-code for the algorithm that implements recursive subroutines.

Register **R6** is used as the stack pointer, which points to the top of the stack. When referring to a variable on the stack, one should access it through reference to the Frame Pointer, which is Register **R5**. For example, suppose the function is nearly complete and the return value is in **R0** and it is

[1] *Introduction to Computing System,* by Yale N. Patt and Sanjay J. Patel, pages 385–393

desired to store it at the Return Value location on the stack. Assuming only one parameter and only one register saved on the stack, the offset will be 3, as seen by the figure below:

Offset	Ptr Location	Stack
0	Current FramePointer $\longrightarrow$	Register1
1		FramePointer (for last function)
2		ReturnAddress
3		ReturnValue
4		Parameter1

To store **R0** at the ReturnValue location, following instruction is used:

```
STR     R0, R5, #3      ; store the return value on the stack
```

9.4 The McCarthy 91 function: an example in LC-3

9.4.1 Definition

The *McCarthy 91* function $M(n)$ has been invented by John McCarthy, the inventor of the Lisp programming language (late 1950's). It is defined for $n = 1, 2, 3, \ldots$, as follows:

$$M(n) = \begin{cases} M(M(n+11)), & \text{if } 1 \leq n \leq 100 \\ n - 10, & \text{if } n > 100. \end{cases} \tag{9.8}$$

Remarkably, $M(n)$ takes the value 91 for $1 \leq n \leq 101$. For values $n \geq 102$ it takes the value $n - 10$. In listing 9.3 the algorithm of $M(n)$ is specified in pseudo-code.

```
// Compute the McCarthy 91 function M(n), n is a positive integer
function M(n)
// n is ≥ 1
if n ≤ 100
   return M(M(n+11))
else
   return n - 10
```

Listing 9.3: The pseudo-code for the recursive McCarthy 91 function.

9.4.2 Some facts about the McCarthy 91 function

The McCarthy 91 $M(n)$ function for some numbers, $1 \leq n \leq 100$, while executing calls itself a number of times, while for $n > 100$ $M(n)$ is called once. Figure 9.6 on page 9–6 shows the growth and shrinkage of the stack during execution for $n = 1, 20, 50, 80$, and 99. A unit of time corresponds to either creation or destruction of a frame on the stack.

For $n = 1$, since the curve becomes 0 at time $= 402$, $M(n)$ is executed 201 times. Figure 9.7 on page 9–6 shows the number of times $M(n)$ is executed for various n.

The size of the stack measured as the number of frames on it for each n in the range 1..123 is shown in figure 9.8 on page 9–8.

9.4.3 Implementation of McCarthy 91 in LC-3

As an example, in this section we give the implementation of the McCarthy 91 function in LC-3. The general algorithm of listing 9.2 on page 9–4 is (slightly) modified in two ways:

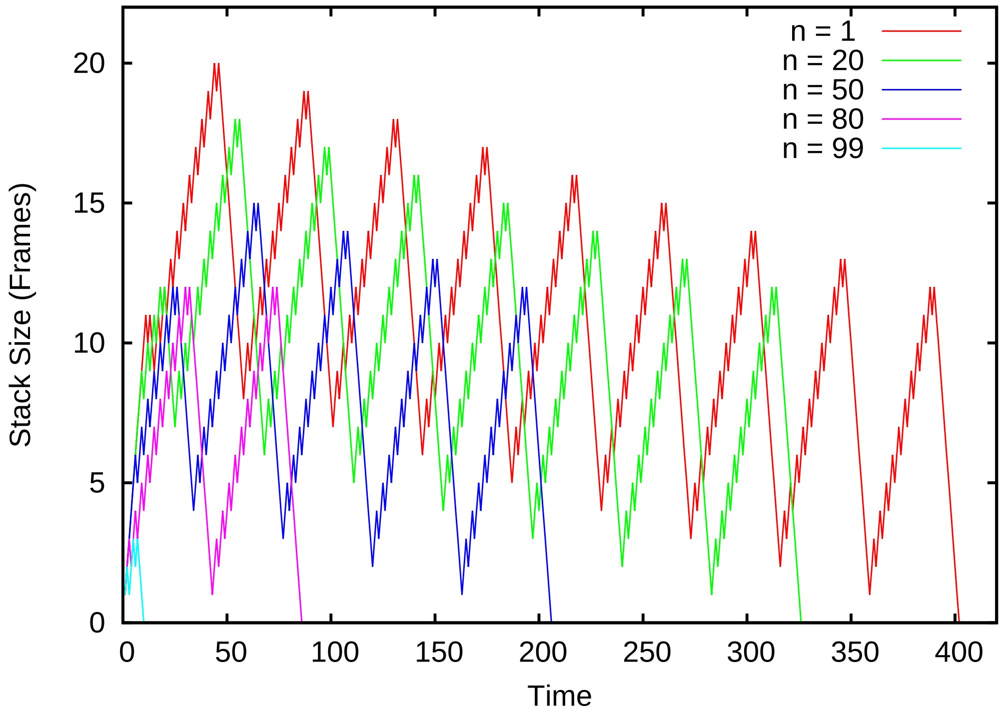


Figure 9.6: Stack size in frames during execution.

n	Number of times $M(n)$ called
1	201
20	163
50	103
80	43
99	5
100	3
101	1
102	1

Figure 9.7: Table that shows how many times the function $M(n)$ is executed before it returns the value for various n.

- The Return Address register **R7** is saved to a temporary location (**R0**) immediately after the function F() is called because PUSH and POP will overwrite **R7**.

- The second change is that registers will be used for temporary storage, as opposed to using local variables, and thus registers used will be saved and then restored.

The modified algorithm with these changes is shown in listing 9.4 on page 9–7.

The source code for the program that calls the McCarthy 91 subroutine appears in listing 9.5 on page 9–8, the push and pop subroutines in listing 9.6 on page 9–9, and the McCarthy 91 subroutine itself on listing 9.7 on page 9–9. The complete program, which is a concatenation of the code in the three aforementioned figures, can be saved on your disk, if your pdf browser supports it, by right-clicking here →.

```
// The calling program
...
PUSH Parameter1          // repeat as needed for additional
                          //parameters
CALL F()                 // jump to F's code
ReturnValue  ← POP       // pop the return value off the stack
POP                      // pop the parameters off the stack,
                          //repeat as needed

...

// The function (subroutine) F

F()                      // beginning of function F
TempVar ←ReturnAddress   // save ReturnAddress (R7) to a temp
                          //variable (R0)
PUSH ReturnValue         // create a place on the stack for the
                          //return value
PUSH TempVar             // push the ReturnAddress onto the stack
PUSH FramePointer        // push the FramePointer for previous
                          //function
FramePointer ← StackPointer−1    // set the new frame pointer to
                          //the location of the
                                    //   first register value
PUSH Register1           // push registers for saving, repeat as
                          //needed
...                              // function body
Register1 ← POP // pop register values off the stack, repeat as
                          //needed
FramePointer ← POP       // restore the old frame pointer
ReturnAddress ← POP      // restore ReturnAddress so the caller can
                          // be
                                 //   returned to
return                   // return to the caller, end of F()
```

Listing 9.4: The pseudo-code for the McCarthy 91 recursive subroutine.

9.5 Testing

Test the square(n) subroutine for various inputs, positive and negative. **Reminder:** You should never pass a negative parameter to square(n). First convert it to positive.

9.6 What to turn in

- A hardcopy of the assembly source code.
- Electronic version of the assembly code.
- For each of the inputs $0, 1, 7, -35$, screenshots that show the contents of locations **x3100** through **x3101**.
- Answer of this question: for each input above what is the maximum size of the stack in terms of frames?

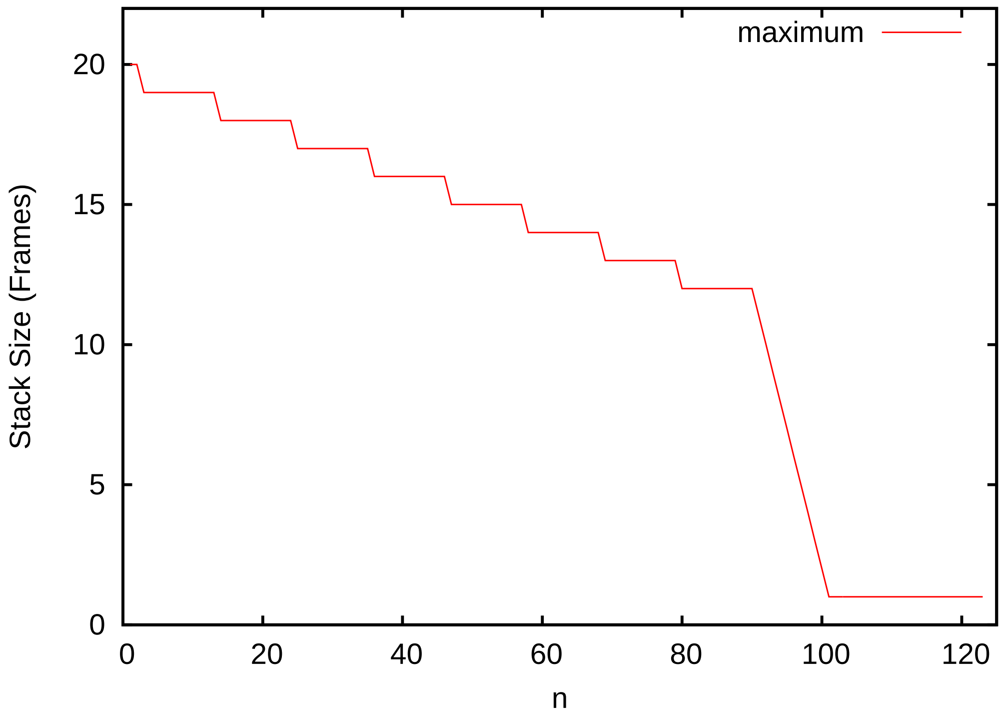


Figure 9.8: Maximum size of stack in terms of frames for n.

```
; Program that uses McCarthy 91 subroutine MC91
; It takes the input from x3100
; It stores the output at x3101
; and outputs the ASCII character of the value to the console
          .ORIG    x3000
          LD       R6, STKBASE      ; set the initial stack pointer

          ; Push(Parameter1)
          LDI      R0, INPUT        ; load function input into R0
          JSR      PUSH             ; push INPUT on stack as parameter1
          ; call McCarthy91
          JSR      MC91
          ; ReturnValue <- Pop()
          JSR      POP              ;
          OUT                       ; print ASCII value of return value
                                    ;   note: ASCII(91) = [
          STI      R0, OUTPUT       ; store the value at x3101

          ; Pop()
          JSR      POP              ; pop off parameter
          HALT
STKBASE   .FILL    x4000            ; stack base address
INPUT     .FILL    x3100            ; McCarthy91 input
OUTPUT    .FILL    x3101            ; McCarthy91 output
```

Listing 9.5: The program that calls the McCarthy 91 subroutine.

```
;push and pop subs
PUSH    ADD     R6, R6, #-1     ; Move top of the stack up
        STR     R0, R6, #0      ; Store R0 there
        RET
POP     LDR     R0, R6, #0      ; Load R0 with top of stack
        ADD     R6, R6, #1      ; Move top of stack down
        RET
```

Listing 9.6: The stack subroutines PUSH and POP.

```
;McCarthy91 function
        ;TempVar <- ReturnAddress
MC91    ADD R0, R7, #0          ; save R7 to R0 so it can be pushed
                        ; on the stack later
        ; Push(ReturnValue)
        JSR     PUSH            ; any value will do for ReturnValue
                                ; space
        ; Push(TempVar)
        JSR     PUSH            ; ReturnAddress is already in R0
        ; Push(FramePointer)
        ADD     R0, R5, #0      ; transfer frame pointer to R0
        JSR     PUSH            ; save frame pointer
        ; FramePointer <- StackPointer-1
        ADD     R5, R6, #-1     ; create a new frame pointer based
                                ;on R6
        ; Push(Register1)
        ADD     R0, R1, #0      ; save R1 by pushing it on the
                                ;stack
        JSR     PUSH            ; save frame pointer

        ; load Parameter1 into R0
        LD      R1, PARAM1      ; load offset
        ADD     R1, R5, R1      ; get address of Parameter1
        LDR     R0, R1, #0      ; load Parameter1 into R0

        ; test to see if Parameter1 ≤ 100
        LD      R1, NEG100      ; load -100
        ADD     R1, R0, R1      ; R1 <- Parameter1 - 100
        BRnz    LESS100         ; if it is ≤ 100 jump to that code

        ; since Parameter1 > 100, add -10 to R0 and cleanup
OVER100 ADD     R0, R0, #-10    ; R0 will be stored in the
                        ;ReturnValue space
                                ; at cleanup
        BRnzp   CLEANUP

        ; since Parameter1 ≤ 100, call recursively MC91(MC91(
                                ;Parameter1+11))
LESS100 ADD     R0, R0, #11     ; add 11 to parameter1 and pass it
                        ;to MC91

        ; call MC91(Parameter1+11)
```

```
        ; Push(Parameter1)
        JSR     PUSH            ; push R0 on stack as Parameter1
        ; call McCarthy91
        JSR     MC91
        ; ReturnValue <- Pop()
        JSR     POP             ; the return value is now in R0
        ADD     R1, R0, #0      ; save the return value into R1
        ; Pop()
        JSR     POP             ; pop off Parameter1

        ; now call MC(MC91(Parameter1+11)) = MC(R1)
        ; Push(Parameter1)
        ADD     R0, R1, #0      ; move the return value of MC91(
                                ;Parameter1+11) back to R0
        JSR     PUSH            ; push R0 on stack as Parameter1
        ; call McCarthy91
        JSR     MC91
        ; ReturnValue <- Pop()
        JSR     POP             ; the return value is now in R0
        ADD     R1, R0, #0      ; save the return value into R1
        ; Pop()
        JSR     POP             ; pop off Parameter1
        ADD     R0, R1, #0      ; move the return value of MC91(
                                ;Parameter1+11) back to R0
                                ;  for cleanup

;store what is in R0 into the ReturnAddress space on the stack
CLEANUP LD      R1, RETVAL      ; load offset
        ADD     R1, R5, R1      ; get address of ReturnAddress
        STR     R0, R1, #0      ; store R0 at ReturnAddress

        ; Register1 <-Pop()
        JSR     POP
        ADD     R1, R0, #0      ; restore R1 from stack
        ; FramePointer <- Pop()
        JSR     POP
        ADD     R5, R0, #0      ; restore R5 from stack
        ; ReturnAddress <- Pop()
        JSR     POP
        ADD     R7, R0, #0      ; restore ReturnAddress from stack
        RET
;refer to variables by offsets from the frame pointer
RETVAL  .FILL   #3
PARAM1  .FILL   #4
NEG100  .FILL   #-100

        .END
```

Listing 9.7: The McCarthy 91 subroutine