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.

```
1        LDI R1, X
2        ...
3        ...
4        HALT
5        ...
6 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.

```
1        STI R2, Y
2        ...
3        ...
4        HALT
5        ...
6 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.

```
1 ; Register R1 has 5 and register R2 has 3
2 ; R4 is used as a temporary register. R2 could have been used
3 ; in the place of R4, but the original contents of R2 would
4 ; have been lost.  The result of 5-3=2 goes into R3.
5        NOT R4, R2
6        ADD R4, R4, #1 ; R4 ← -R2
7        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.

```
1          ...
2          AND R1, R1, x0   ; Since R1 ← 0, N = 0, Z = 1, P = 0
3          ADD R2, R1, x1   ; Since R2 ← 1, N = 0, Z = 0, P = 1
4          BRz LABEL
5          ...
6 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.

```
1          AND R2, R5, x0 ; result placed in R2 is zero
2          BRz TARGET     ; Branch if result was zero (it was)
3          NOT R1, R3
4          ...
5          ...
6 TARGET   ADD R5, R1, R2
7          ...
```

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.

```
1 ; Input:  R1 has value X.
2 ; Output: R2 has value |X|.
3         ADD R2, R1, #0  ; R2 ← R1, can now use condition codes
4         BRzp ZP         ; If zero or positive, do not negate
5         NOT R2, R2
6         ADD R2, R2, #1  ; R2 = −R1
7 ZP      ...             ; At this point R2 = |R1|
8         ...
```

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**.

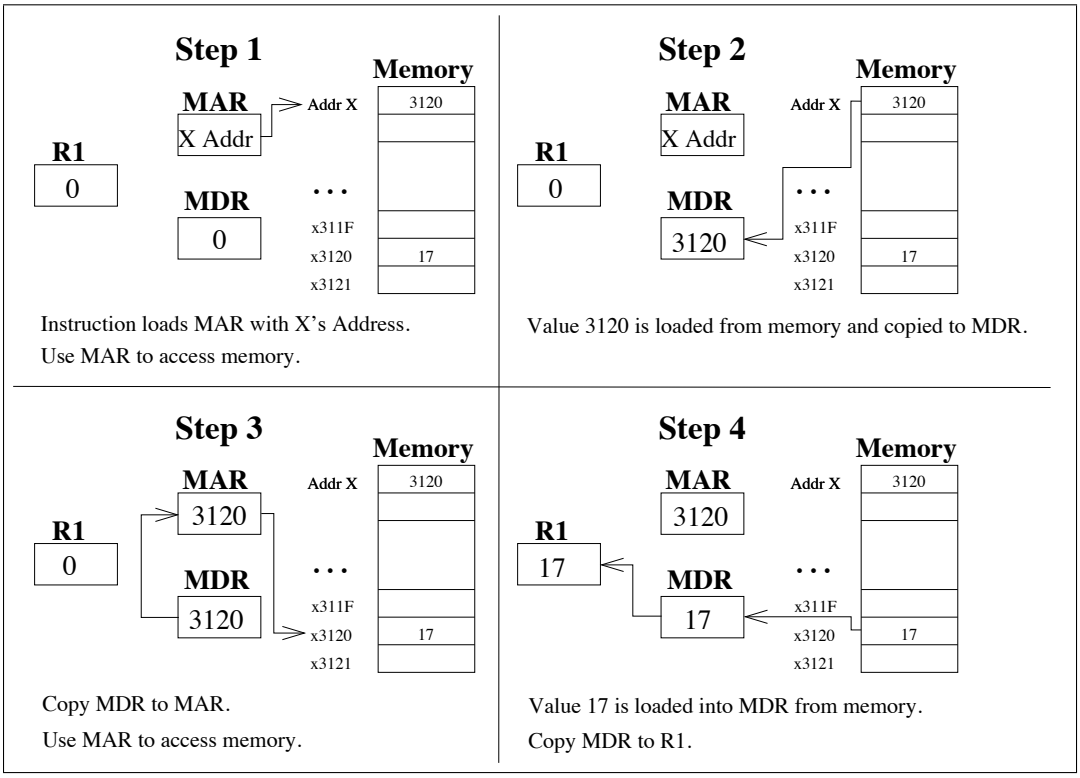

Figure 2.2: The steps taken during the execution of the instruction **LDI R1, X**.

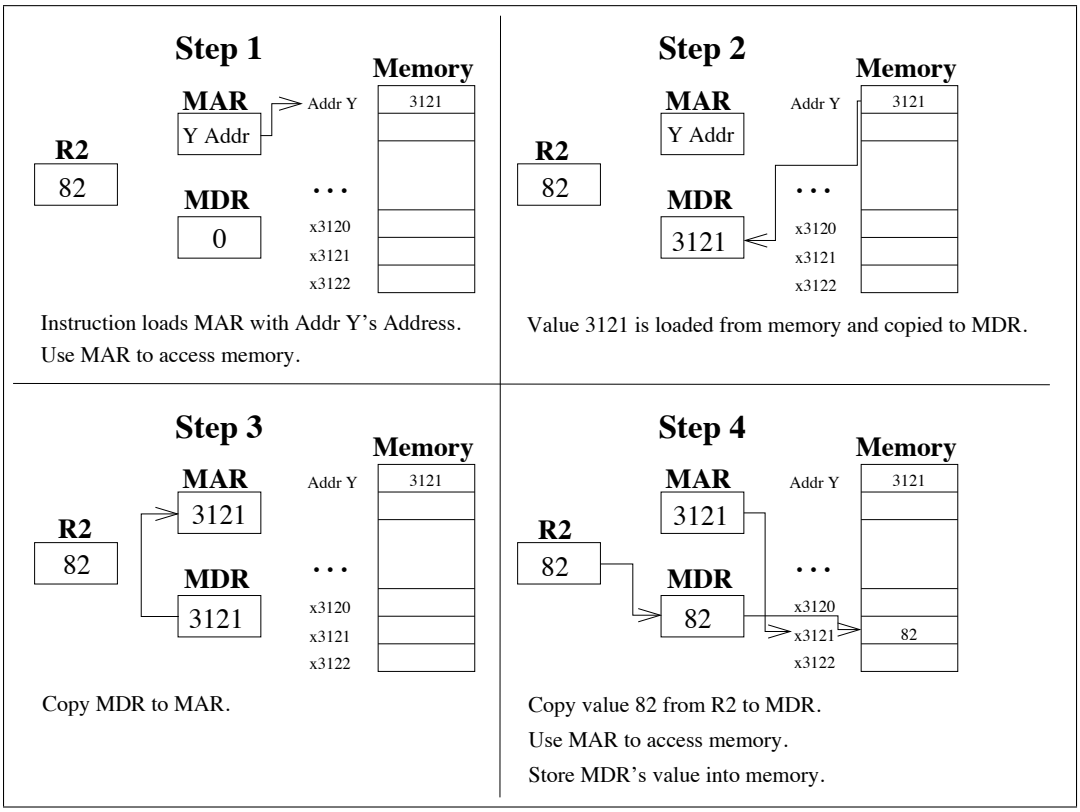

Figure 2.3: The steps taken during the execution of the instruction **STI R2, Y**.

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