CSCI 2510 Computer Organization 2020-21

Assignment 2

Deadline: October 20, 2020 (TUE) 14:30pm

Submission Notes:

(1) For each of the following written exercises (*Questions 1~3*), please show your steps and explain in detail when needed to receive full credit.

(2) Submit three files named **assignment2.pdf** (for *Question 1~3*), **stack.asm** (for *Programming Exercise*), and **report.pdf** (for *Programming Exercise Report*) to Blackboard before the deadline (14:30pm on Oct 20).

(3) Late submission is not acceptable.

Question 1 (10 pts)

Suppose that registers R1, R2 and R3 contain the decimal numbers 256, 384 and 512, respectively, and LOC corresponds to the memory address 1024 in decimal. Specify the addressing mode and the effective address (EA) for each of the following operands:

(a) R3

(b) (R3)

(c) (R2, R3)

(d) LOC

(e) -128(R2)

Question 2 (20 pts)

Given two 4-bit registers R1 and R2 storing signed integers in 2's-complement format. Please specify the condition flags that will be affected by **SUB R1, R2**.

Note 1: The ***SUB*** *instruction subtracts the value of R2 from the value of R1 (i.e., $R1 = R1 - R2$).*

Note 2: You only need to specify the ***N*** *(negative),* ***Z*** *(zero), and* ***V*** *(overflow) condition flags, since the* ***C*** *(carry) condition flag has a different definition for the subtract instruction.*

(a) **R1** = $(3)_{10}$ and **R2** = $(4)_{10}$

(b) **R1** = $(1)_{10}$ and **R2** = $(1)_{10}$

(c) **R1** = $(3)_{10}$ and **R2** = $(-6)_{10}$

(d) **R1** = $(-1)_{10}$ and **R2** = $(1)_{10}$

(e) **R1** = $(-7)_{10}$ and **R2** = $(3)_{10}$

(f) **R1** = $(7)_{10}$ and **R2** = $(6)_{10}$

Question 3 (20 pts)

The below program adds up a list of *n* numbers, where the size *n* is stored in memory address **N**, and **NUM1** denotes the memory address of the first number. Rewrite the program so that the numbers in the list are accessed in the reverse order: that is. the first number accessed is the last one in the list, and the last number accessed is at memory location NUM1.

LABEL	OPCODE	OPERAND	COMMENT
	Load	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, addr NUM1	Get address of the first number.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer to the list.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	Branch back if not finished.
	Store	R3, SUM	Store the final sum.

Programming Exercise (50 pts)

In addition to the processor stack, it may be convenient to maintain our own stack in programs. In this programming exercise, we are going to implement a stack using MASM IA-32 assembly language. In our implementation, the stack is allocated a fixed amount of memory space to store **at most ten** positive numbers of 32-bits (**`dword`**), and the stack grows toward **lower-numbered** address locations. In addition, the stack can be manipulated via the following functions:

- `pushnum`: Input a **`positive number`** to push it onto the top of stack;
- `popnum`: Input **`0`** to pop and print out the number from the top of the stack;
- `gettop`: Input **`-1`** to print out the number on the top of the stack without popping it;
- `getsize`: Input **`-2`** to print out the size of numbers that have been pushed into the stack;
- `showstack`: Input **`-3`** to print out the contents of the stack.

Note: It is not allowed to define additional variables in ".data".

Exercise 1 (30 pts)

Complete the provided MASM IA-32 assembly program named **stack.asm** to implement a stack. There are **six** "missing lines" in total.

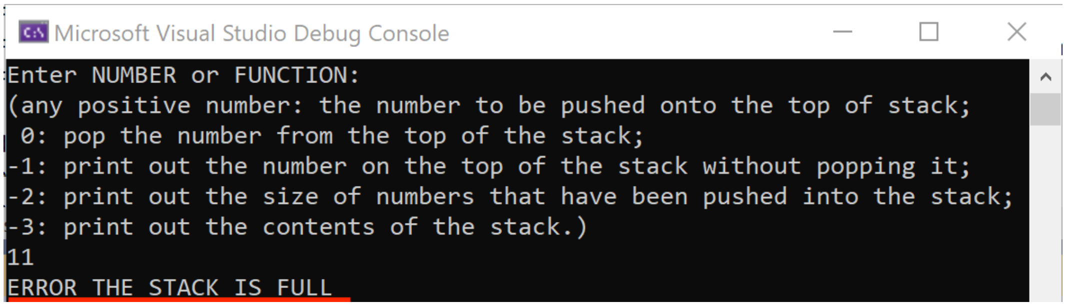
(a) Test your program using the input sequence `1 2 3 4 5 0 0 0 0 0` and paste the screenshots of your results in the report.

(b) Test your program using the input sequence `1 0 1 2 0 0 3 4 0 0` and paste the screenshots of your results in the report.

Exercise 2 (10 pts)

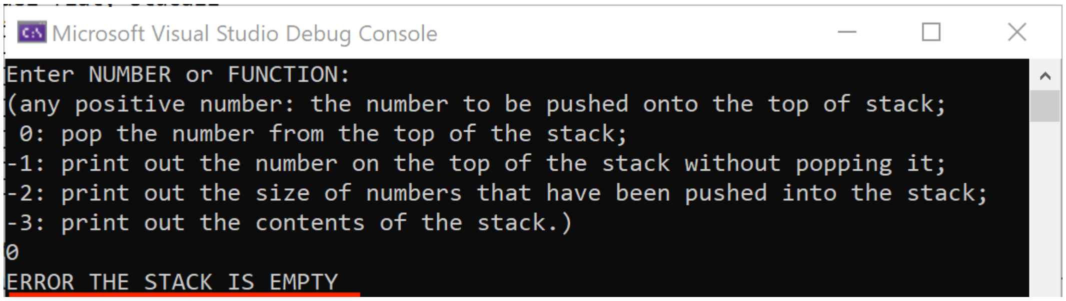
Our stack is only allocated a fixed amount of space in the memory. Therefore, it is important to avoid pushing an item onto the stack when the stack has reached its maximum size. Also, it is important to avoid attempting to pop an item off an empty stack. Revise the program **stack.asm** to handle the following two possible errors by showing alert messages as follows:

- **Possible error 1:** Push a number into the stack when the stack is full.



```
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
11
ERROR THE STACK IS FULL
```

- **Possible error 2:** Pop a number from the stack when the stack is empty.



```
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
0
ERROR THE STACK IS EMPTY
```

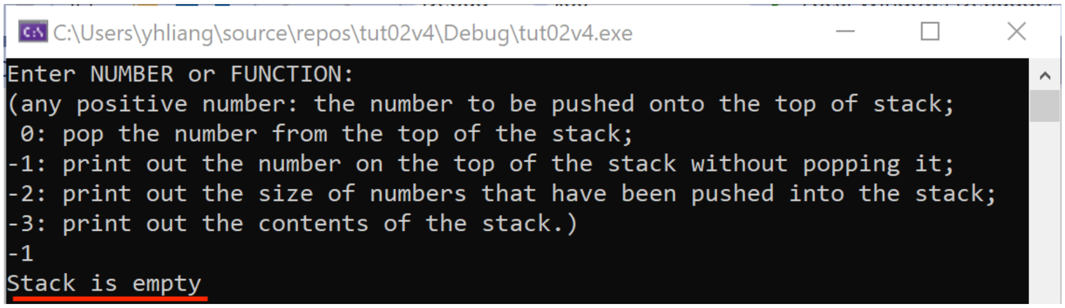
(a) Test your program using the input sequence `1 2 3 0 0 0 0` and paste the screenshots of your results in the report.

(b) Test your program using the input sequence `1 2 3 4 5 6 7 8 9 10 11` and paste the screenshots of your results in the report.

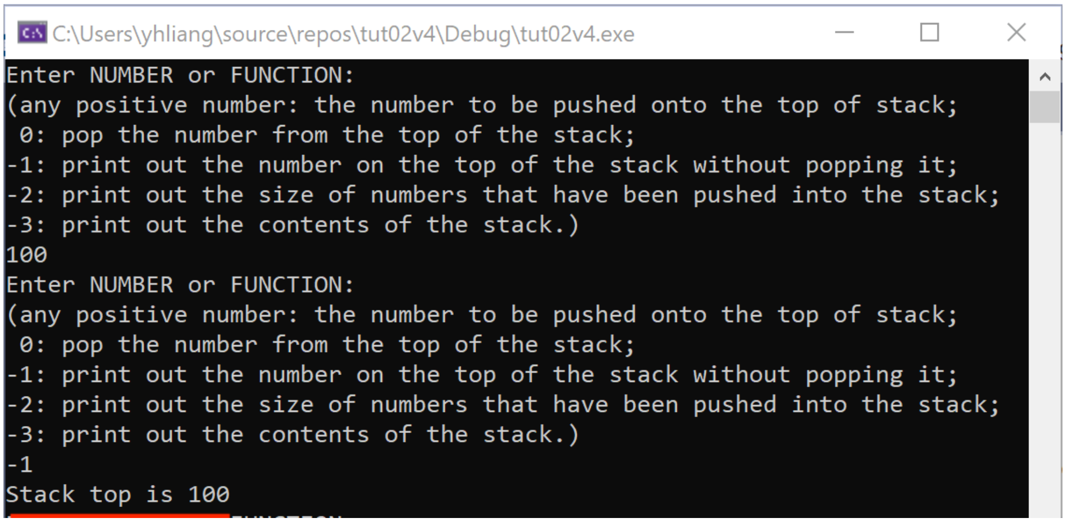
Exercise 3 (10 pts)

Implement the following two new functions `gettop` and `getsize` in the program **stack.asm**:

- `gettop`: Print out the number on the top of the stack <u>without</u> popping it.

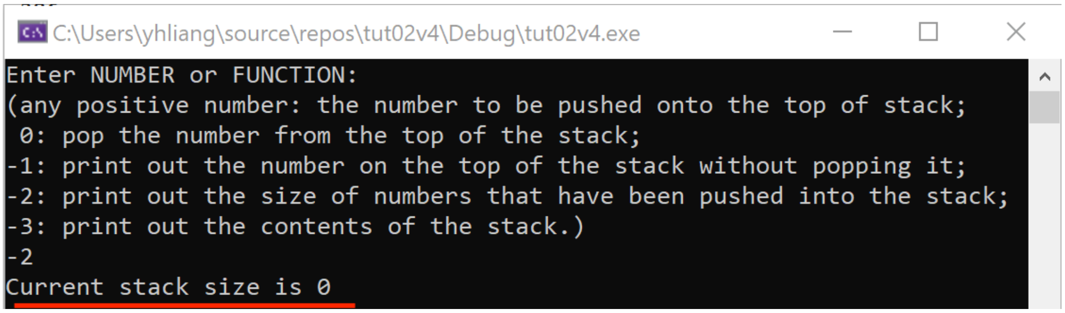


```
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
-1
Stack is empty
```

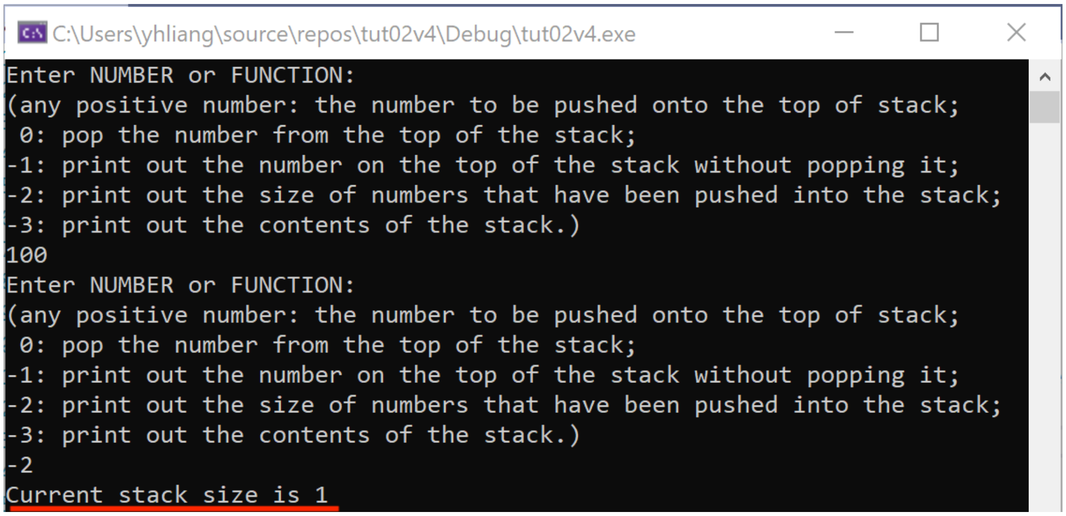


```
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
100
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
-1
Stack top is 100
```

- `getsize`: Print out the size of numbers that **have been pushed** into the stack.



```
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
-2
Current stack size is 0
```



```
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
100
Enter NUMBER or FUNCTION:
(any positive number: the number to be pushed onto the top of stack;
 0: pop the number from the top of the stack;
-1: print out the number on the top of the stack without popping it;
-2: print out the size of numbers that have been pushed into the stack;
-3: print out the contents of the stack.)
-2
Current stack size is 1
```

(a) Test your program using the input sequence `1 2 3 -1 4 5 -1 0 0 0 0 0 -1 0` and paste the screenshots of your results in the report.

(b) Test your program using the input sequence `1 0 1 2 -2 0 0 3 4 0 0 -2` and paste the screenshots of your results in the report.