Project 2: Stack Implementation

- **Introduction:**

For this lab, you will write a program that implements a stack. This will entail creating a stack data structure, operations on the stack, and a main loop. The stack data structure is a set of memory locations that holds the stack content and all necessary information about the stack (i.e. the address of the top / bottom of the stack, the current size of the stack, etc). The stack operations you must support and the input/output format are given below.

- **Specifications**:

0. Your stack should have the size of 10, and initialized with 9 items on the stack. Each item is a number, and the 9 values are from your UIN – the last digit at the bottom of the stack.
1. Get a character from the user, which will indicate the operation.
2. If the operation is `+` (push operation):
   If the stack is full, print "Stack overflow!".
   Otherwise, get a number from the user and push it onto the stack.
   Loop back to 1.
3. If the operation is `-` (pop operation):
   If the stack is empty, display "Stack underflow!".
   Otherwise, pop a number from the top of the stack and print it to the output.
   Loop back to 1.
4. If the operation is `d` (display stack content):
   If the stack is empty, print "Stack empty."
   Otherwise, print the contents of the stack from top to bottom
   (in the following format: *position: value*).
   Loop back to 1.
5. If the operation is `e` (empty stack) :
   Empty the stack (no output is necessary) and loop back to 1.
6. If the operation is `n` (number of items):
   Print the number of items currently on the stack - only display the number, no other text.
   Loop back to 1.
7. If the operation is `q` (quit):
   Print "Goodbye." and quit the program.
8. If the operation is none of the above
   Print "Invalid operation!" and loop back to 1.

- **Example Input / Output**

```
Operation: d
1: 6
2: 5
3: 4
4: 3
5: 2
6: 1
7: 0
8: 9
9: 8
Operation: -
Popped 6
Operation: -
Popped 5
Operation: n
7 items
Operation: +
Push: 180
Operation: d
1: 180
2: 4
3: 3
4: 2
5: 1
6: 0
7: 9
8: 8
Operation: e
Operation: n
0 items
```

```
Operation: +
Push: 345
Operation: +
Push: 111
Operation: +
Push: 23
Operation: -
Popped 23
Operation: +
Push: 12345
Operation: +
Push: 9888
Operation: +
Push: 17
Operation: d
1: 17
2: 9888
3: 12345
4: 111
5: 345
Operation: +
Push: 1
Operation: n
6 items
Operation: +
Push: 2
```

```
Operation: +
Push: 3
Operation: +
Push: 4
Operation: d
1: 4
2: 3
3: 2
4: 1
5: 17
6: 9888
7: 12345
8: 111
9: 345
Operation: +
Push: 100
Operation: +
Stack overflow!
Operation: e
Operation: d
Stack empty.
Operation: u
Invalid operation!
Operation: -
Stack underflow!
Operation: q
Goodbye.
```

Here we use red color to indicate user input, and blue color to indicate program output.

Suppose the UIN is 654321098 to initialize the stack.

- **What to turn in (in a zipped file):**
  1. Your assembly file, well organized and commented.
  2. Lab report, containing the following parts:
     a) Program features:
        i. Is your program running correctly on all features? If not, describe the malfunction in details.
        ii. Have you run into any significant problems / bugs in this project? Describe how they are solved.
     b) Implementation details:
        i. How did you implement the stack? How many bytes for each item on the stack? What other data structures did you use to support the implementation of the stack? Where are your top and bottom of the stack? How did you define the stack pointer?
        ii. How did you implement the operations of push and pop?
        iii. How did you support the other operations – checking for overflow / underflow, displaying stack content / depth of the stack, and emptying the stack?
     c) Provide 5 screenshots on the running I/O of your program to show the behavior of your program. Be thorough cover all the possible cases.
- **You will be graded on the following criteria:**
  1. program functionality
  2. thoroughness and insights in answering the questions in the lab report
  3. code quality
- **Special notice:** You can consult with your friends for ideas to implement the project, but make sure to write your own code. You are responsible to make sure your code is unique enough. If your code looks "similar enough" to somebody else's code, both will be deducted 30 points for "lack of originality".
- **Submission notes:**

  1. The code file should be separate (do not copy & paste your code into the lab report pdf) from the report, and should be named with .asm extension.
  2. If you are making more than 1 submission, you will be graded based on the latest submission of the project. Please make sure the latest submission have the final version of code as well as the report. (When we download the files, we will download only the last attempt. If someone has report in the first attempt and code in the second attempt, then only code will be downloaded by us)

Kai Zhao 670720413
ECE 267 Spring 2012
2012 April 4

Project 2: Stack Implementation

2.

a) **Program features:**

i) **What have you implemented?**

- Error tolerance for non-integer pushes. I.e. the program will not get an exception error from inserting non-integers. Error tolerance for non-integer pushes. I.e. the program will not get an exception error from inserting non-integers.
- Any string of length 30 bytes is allowed to be pushed into the stack.
- Error tolerance for lengthy item pushes. If the item is greater than 30 chars, then the user will have to reenter the item to push.
- The double space error is fixed. E.g. when a string is read, a null byte is automatically inserted into the item, which print out as \n as string, which results in spacing errors. Therefore, there is error tolerance for the error tolerance.
- Error tolerance for every operation when there is no items in the stack.
- NEW Operation! The stack can be inverted by inserting 'i' into the operation. Stack inversion will flip the stack so that items at the bottom of the stack will be on top of the stack and vice versa. This operation is useful when the user gets confused with the direction of the stack.
- NEW Operation! The items in the stack can be moved in pairs. When the user wants to move items, the program will prompt the user for the item number of the 2 items that the user wants to switch. Note: Unlike the display stack content operation, items are numbered from bottom to top. The item number of the displayed bottom item is 1. There is error tolerance if the user inputs an integer that is greater than the length of the stack.

- NEW Operation! The stack can be rotated by inserting 'r' into the operation. Stack rotation will push each item farther into the stack and the first item at the bottom of the stack will be reset on top of the stack. This operation is useful to pop the stack from the bottom without having to delete the rest of the stack.
- NEW OPERATION!!! As the final boss of this project, the stack can be sorted. This operation will compare each item and place the item with the lowest ASCII value on the bottom of the stack and vice versa. If the first byte of two items is the same, then the program will check the next byte. This feature is useful to sort the stack. When this operation is combined with the inversion operation, then the user may sort from lowest to highest or vice versa by ASCII values.

ii) Is your program running correctly on all features? If not, describe the malfunction in details.

Answer: Since the last revision, yes, my program is running correctly on all features.

iii) Have you run into any significant problems/bugs in this project? Describe how they were solved.

Answer: Yes, I ran into a lot of significant problems.

- I still struggled with string inputs and how to display strings because registers are too small to hold an entire string. It turns out to be much easier to find the correct memory location to read the string on before reading the string as opposed to reading the string and then moving the string to the correct memory location. Furthermore, string inputs appear to have a null byte at the end, which differs from regular number inputs of my UIN, which causes the spacing to mess up. Therefore, I added a null terminator at the end

of each digit of my UIN. This problem was solved by monitoring the data layout to learn how to correctly use strings.

- I struggled with the top/bottom/first/last item of the stack. I am still not sure that if you build your stack in memory from bottom up, is the "bottom" of the stack still considered the first item into the stack. Furthermore, Example Input / Output seems to show that the UIN has filled the 1st through 9th item of the stack, so I THOUGHT that consequent pushes will be the 10th item. However, I realized higher number means that they are farther into the stack, and so I build my stack in the wrong direction. More confusion resulted from when I decided to build a top bottom stack because of historical precedence as mentioned on page 114 of the book, which meant increasing the memory address (going down on the memory layout) is adding to the top of the stack. I fixed this by rewriting the program.
- I had problems with the error tolerance. To do the error tolerance of invalid input, the program had to read in a string, which lead to aesthetic problems and requiring more memory, which lead to input length limitations and did not allowed the memory to be stored into a register, which meant that displaying the stack is not just as simple as displaying contents of a register, which increased the difficulty of figuring how to display the stack, which was solved by re-rewriting the program.

b) Implementation details:

i) How did you implement the stack? How many bytes for each item on the stack? What other data structures did you use to support the implementation of the stack? Where are your top and bottom of the stack? How did you define the stack pointer?

Answer: The stack was implemented in memory with 16 bytes allotted for each item from top to bottom (stack grows from higher to lower memory addresses). The other data structure I used to support the stack is a stack pointer (points to which item I am working with), a stack counter (tells me how many items are in the stack), an address holder (displays the address of the current item that I working with), and a display counter (used to be compared with the number of items in the stack to determine how far I should adjust the stack pointer). The top of the stack is the first address in memory, 0x10010000 while the bottom of the stack is determined by the stack pointer after adjustment from the comparison of the number of items in the stack and a counter. The bottom of the stack is located on 0x1001009c when the stack is full. I define the stack pointer as an integer in hexadecimal.

ii) How did you implement the operations of push and pop?

Answer: The push operation was implemented by, first, checking for overflow, second, increasing the number of items in the stack, third, finding the memory location of where the item should be pushed, and finally, prompting the user for a string. The pop operation in implemented by, first, checking for underflow, second, find the item on top of the stack, third, display the item, and finally, decreasing the number of items in the stack.

iii) How did you support the other operations – checking for overflow / underflow, displaying stack content / depth of the stack, and emptying the stack?

Answer: Overflow, underflow, and depth of the stack was done by checking the number of items in the stack, which was modified accordingly everything the stack size changes. Displaying the stack content was done by comparing the number of items in the stack with a counter in a loop to aim the stack pointer to the first item of the stack, then, displaying all items after the first item. Emptying the stack was simply done by setting the number of items in the stack to 0. The extra

operation of inversion was done by, first, set up the initial conditions to invert only half of the words in the stack with their counter parts, then, iterate through the inversion of each pair. The extra operation of moving items in the stack was done by first writing the method to move items, second, ask the user for the item numbers the user want to move, third, call the move item method, and finally, do error tolerance. The extra operation of rotation was done by, first, extracting the 1st item, second, using a loop inside of another loop to shift every item farther into the stack, third, find the location of the top item, and finally, placing the 1st item into the memory location of the original top item to have every item rotated by one. The extra operation of sorting the stack was done by a bubble sort, first, set up the initial and ending conditions, second, find the address of two items that may need sorting, third, compare byte by byte to find if the order is correct or needs to be switched, four, switch the items if necessary, fifth, repeat step two through step four for every item in the stack, and finally, repeat step two through step five for every item in the stack.

c) **Provide 5 screenshots on the running I/O of your program to show the behavior of your program. Be thorough cover all the possible cases.**

Answer:

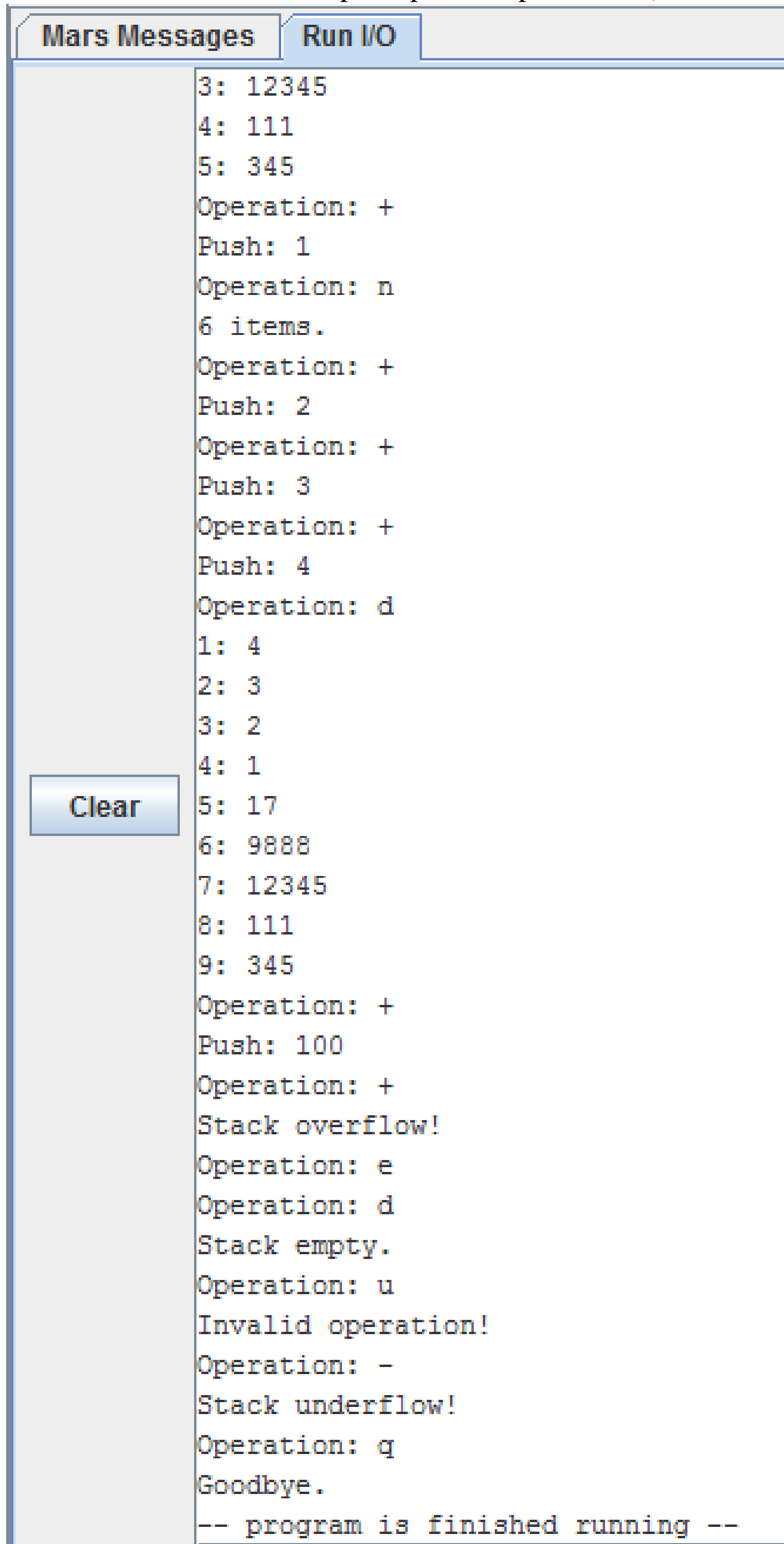
- Screenshot 1: Example Input / Output Part 1 (Commands taken from Project Assignment)

Mars Messages | Run I/O

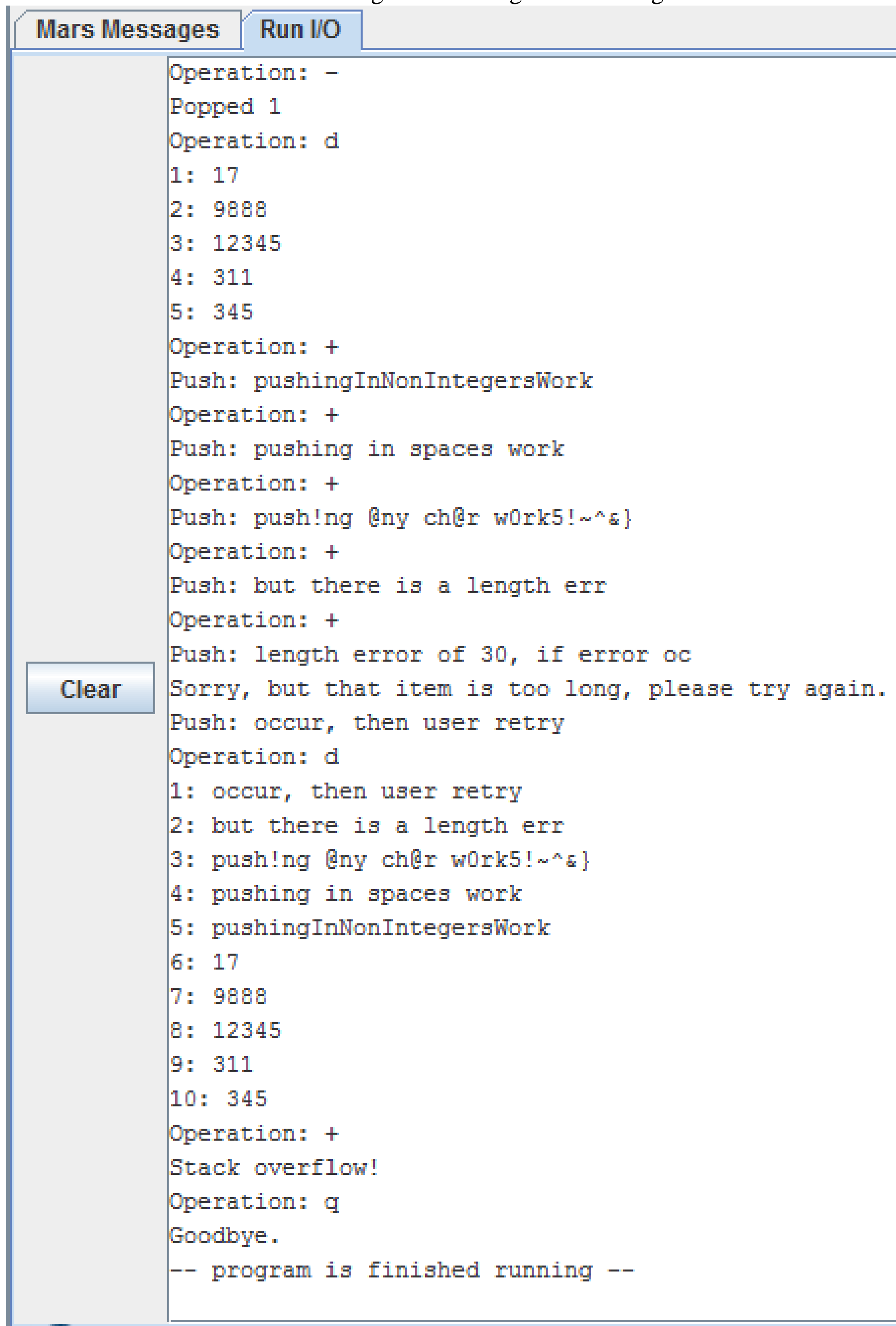
Clear

```
Operation: n
7 items.
Operation: +
Push: 180
Operation: d
1: 180
2: 4
3: 3
4: 2
5: 1
6: 0
7: 9
8: 8
Operation: e
Operation: n
0 items.
Operation: +
Push: 345
Operation: +
Push: 311
Operation: +
Push: 23
Operation: -
Popped 23
Operation: +
Push: 12345
Operation: +
Push: 9888
Operation: +
Push: 17
Operation: d
1: 17
2: 9888
3: 12345
4: 311
5: 345
Operation: +
```

- Screenshot 2: Example Input / Output Part 2 (Commands taken from Project Assignment)



- Screenshot 3: Error Checking of Non-Integers and Strings



- Screenshot 4: Move Items in Stack and Rotate Stack

sages Run I/O

```
Operation: d
1: 4
2: 3
3: 2
4: 1
5: 0
6: 9
7: 8
Operation: m
Note: Unlike the display stack content operation, items are numbered from bottom to top.
The item number of the displayed bottom item is 1. Item number of the first item to move: 1
Item number of the second item to move: 8
That item number is greater than the size of the stack! Please try again.
Item number of the second item to move: 3
Operation: d
1: 4
2: 3
3: 2
4: 1
5: 8
6: 9
7: 0
Operation: r
Operation: d
1: 0
2: 4
3: 3
4: 2
5: 1
6: 8
7: 9
Operation: e
Operation: m
There are no items to move!
Operation: r
There are no items to rotate!
Operation:
```

- Screenshot 5: Sort Stack and Invert Stack

```
Mars Messages   Run I/O

Operation: d
1: abb
2: cab
3: bac
4: aba
5: 123
6: abc
Operation: i
Operation: d
1: abc
2: 123
3: aba
4: bac
5: cab
6: abb
Operation: s
Operation: d
1: cab
2: bac
3: abc
4: abb
5: aba
6: 123
Operation: i
Operation: d
1: 123
2: aba
3: abb
4: abc
5: bac
6: cab
Operation: e
Operation: i
There are no items to invert!
Operation: s
There are no items to sort!
Operation:

Clear
```