NEW YORK CITY COLLEGE OF TECHNOLOGY

THE CITY UNIVERSITY OF NEW YORK

300 JAY STREET, BROOKLYN, NY 11201-1909

Department of Computer Engineering Technology

CET 3510 - OL25 - Microcomputer Systems Technology lab

LABORATORY REPORT No_010

Lab_No_010 The Stack and LIFO Data Structure

Professor: Dr. Rubén Velásquez, Ph.D.

Spring 2021

Student's Name: Ashahi Shafin

Student's ID: 23607352

Preparation date: *April 28, 2021*

Due date: *May 05, 2021*

1. Table of content

2. OBJECTIVE

Laboratory No 10. The stack segment in the memory is where the x86 processor maintains the stack. The stack store important information about program including local variables, subroutine information, and temporary dat. The 80x86 controls its stack via the ESP (stack pointer) registers. The x86 decrements the stack pointer by the size of the data when applying the PUSH instruction and then it copies the data to memory where the ESP is then pointing. The POP instruction copies the data from memory location ESP before adjusting the value ESP. Recall the declared variable and its size. Generate random numbers (4 bytes) between 0and 100. Generate random even and odd number between 0 to 100. Practice push and pop instructions and observe the changes of the ESP value.

3. LABORATORY TOOLS TO PERFORM THE TASK

Computer NZXT

Microsoft visual studio 2015 Software
Microsoft Word 2019 Software
Microsoft Notepad Software

4. SOURCE CODE

Laboratory No_010 Example 10.1

Source Code

```
#include <stdio.h>
#include <iostream>
#include <time.h>
#include "windows.h"
using namespace std;
#define size 6



int main()
{
	printf("Lab_No_010 The Stack and LIFO Data Structure\n");
	printf("Module: C++ progrmming \n");
	printf("Ashahi Shafin, ID#23607352\n");
	printf("CET3510-OL25\n");
	printf("Presentation date: April 28, 2021\n");
	printf("Due date: May 5, 2021\n");
	printf("Example 10.1 StackApp.cpp\n");
	printf("file name: 10.1 StackApp.cpp\n");
```

```
	printf("--------------------------------------------------------------------------------
\n");

int i;
int arr_data[size];
int arr_pop[size];
int stack_addr_push[size];
int stack_addr_pop[size];
int espb, espa;
int temp = 0;

//randomize seed
srand(time(0));

//random number 0 to 100
for (i = 0; i < size; i++)
{
	arr_data[i] = rand() % 100;

	}
	cout << "Generated random numbers store at the array" << endl;
	for (i = 0; i < size; i++)
{
	cout << arr_data[i] << " ";
}
	cout << endl;

	cout << "================================PUSH================================" << endl;
	_asm
	{
		mov espb, ESP;
	}
	cout << "The hexadecimal value of the ESP before the PUSH instruction is executed "
		<< hex << espb << endl;
	for (i = 0; i < size; i++)
	{
		temp = arr_data[i];
		_asm
		{
			mov eax, temp;
			push eax;
			mov espa, esp;
		}
		stack_addr_push[i] = espa;
		cout << "Push the number " << dec << arr_data[i] << " onto the stack" << endl;
		cout << "The hexadecimal value of the ESP after the PUSH instruction is executed "
			<< hex << espa << endl;
	}
	cout << endl;
	cout << "================================POP================================" << endl;
	_asm
	{
		mov espb, ESP;

	}
	cout << "The hexadecimal value of the ESP before the POP instruction is executed "
		<< hex << espb << endl;
	for (i = 0; i < size; i++)
	{
```

```
		_asm
		{
			pop eax;
			mov espa, esp;
			mov temp, eax;
		}
		arr_pop[i] = temp;
		stack_addr_pop[i] = espa;
		cout << "The number " << dec << arr_pop[i] << " popped off from the stack" <<
endl;
		cout << "The hexadecimal value of the ESP after the POP instruction is executed "
			<< hex << espa << endl;
	}
	cout << endl;
	cout << "================================================================" << endl;
	cout << "The data stored onto the stack in order are " << endl;
	for (i = 0; i < size; i++)
	{
		cout << dec << arr_data[i] << " ";
	}
	cout << endl;
	cout << "----------------------------------------------------------------" << endl;
	cout << "The ESP addresses onto the stack in order are " << endl;
	for (i = 0; i < size; i++)
	{
		cout << hex << stack_addr_push[i] << " ";
	}
	cout << endl << endl;;
	cout << "================================================================" << endl;
	cout << "The values popped off from the stack in order are " << endl;
	for (i = 0; i < size; i++)
	{
		cout << dec << arr_pop[i] << " ";
	}
	cout << endl;
	cout << "----------------------------------------------------------------" << endl;
	cout << "The ESP addresses popped off the stack in order are" << endl;
	for (i = 0; i < size; i++)
	{
		cout << hex << stack_addr_pop[i] << " ";
	}
	cout << endl;

	system("pause");
	exit(0);
	return 0;
}
```

5. *SOURCE CODE LINE DESCRIPTION*

Laboratory No_010 Example 10.1

Line	Source Code Description
01	*#include "stdafx.h"*

Precompiled Header **stdafx**. **h** is basically used in Microsoft Visual Studio to let the compiler know the files that are once compiled and no need to compile it from scratch. ... **h**" before this **includes** then the compiler will find the compiled header files from **stdafx**. **h** and does not compiled it from scratch.

02	*#include "stdio.h"*

The stdio.h (standard library header) is a file with ".h" extension that contains the prototypes of standard input-output functions used in c.

03	#include<iostream>

h, **iostream** provides basic input and output services for C++ programs. **iostream uses** the objects cin , cout , cerr , and clog for sending data to and from the standard streams input, output, error (unbuffered), and log (buffered) respectively.

04	int main(void)
	{
	MAIN PROGRAM
	}

In C and C++ **int main**(**void**) **means** that the function takes NO arguments. ... **Int main**(**void**) is used in C to restrict the function to take any arguments, if you do not put **void** in those brackets, the function will take ANY number of arguments you supply at call.

05	`int i;`
	`int arr_data[size];`
	`int arr_pop[size];`
	`int stack_addr_push[size];`
	`int stack_addr_pop[size];`
	`int espb, espa;`
	`int temp = 0;`

A **variable** is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a **variable** can be changed during program execution. A **variable** is only a name given to a memory location, all the operations done on the **variable** effects that memory location.

06	`srand(time(0));`

srand() function is an inbuilt function in **C++** STL, which is **defined** in <cstdlib> header file. **srand**() is used to initialise random number generators. This function gives a starting point for producing the pseudo-random integer series. The argument is passed as a seed for generating a pseudo-random number.

07	`for (i = 0; i < size; i++)`

loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.

08	`cout << "Generated random numbers store at the array" << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

09	`for (i = 0; i < size; i++)`

loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.

10	`cout << arr_data[i] << " ";` `cout << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

11	_asm
	{
	For mnemonic assembly instructions declaration
	explained in lines 12.
	}

_asm-declaration gives the ability to embed assembly language source code within a **C++** program. This declaration is conditionally-supported and implementation defined, meaning that it may not be present and, even when provided by the implementation, it does not have a fixed meaning.

12	`mov espb, ESP;`

Mnemonic MOV instruction loads an operand source ESP register to an operand destination espb register.

13	`cout << "The hexadecimal value of the ESP before the PUSH instruction is executed "` `<< hex << espb << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

14	`for (i = 0; i < size; i++)`

loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.

15	_asm
	{
	For mnemonic assembly instructions declaration
	explained in lines 16.
	}

_asm-declaration gives the ability to embed assembly language source code within a **C++** program. This declaration is conditionally-supported and implementation defined, meaning that it may not be present and, even when provided by the implementation, it does not have a fixed meaning.

16	`mov eax, temp;`
	`push eax;`
	`mov espa, esp;`

Mnemonic MOV instruction loads an operand source temp and esp register to an operand

destination eax and espa register while being push.

17	`cout << "Push the number " << dec << arr_data[i] << " onto the stack" << endl;`
	`cout << "The hexadecimal value of the ESP after the PUSH instruction is executed "`
	`<< hex << espa << endl;`
	`cout << endl;`
	`cout << "================================POP================================" << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

19	`_asm`
	`{`
	For mnemonic assembly instructions declaration
	explained in lines 20.
	`}`

_asm-declaration gives the ability to embed assembly language source code within a **C++** program. This declaration is conditionally-supported and implementation defined, meaning that it may not be present and, even when provided by the implementation, it does not have a fixed meaning.

20	`pop eax;`
	`mov espa, esp;`
	`mov temp, eax;`

Mnemonic MOV instruction loads an operand source esp and eax register to an operand destination eax to temp register while being pop.

21	`cout << "The number " << dec << arr_pop[i] << " popped off from the stack" << endl;`
	`cout << "The hexadecimal value of the ESP after the POP instruction is executed "`
	`<< hex << espa << endl;`
	`cout << endl;`
	`cout << "============================================================" << endl;`
	`cout << "The data stored onto the stack in order are " << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

22	`for (i = 0; i < size; i++)`

loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.

23	`cout << hex << stack_addr_push[i] << " ";`
	`cout << "------------------------------------------------------------" << endl;`
	`cout << "The ESP addresses onto the stack in order are " << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

24	`for (i = 0; i < size; i++)`

loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.

25	`cout << hex << stack_addr_push[i] << " ";`
	`cout << endl << endl;;`
	`cout << "============================================================" << endl;`
	`cout << "The values popped off from the stack in order are " << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.	
27	`for (i = 0; i < size; i++)`
loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.	

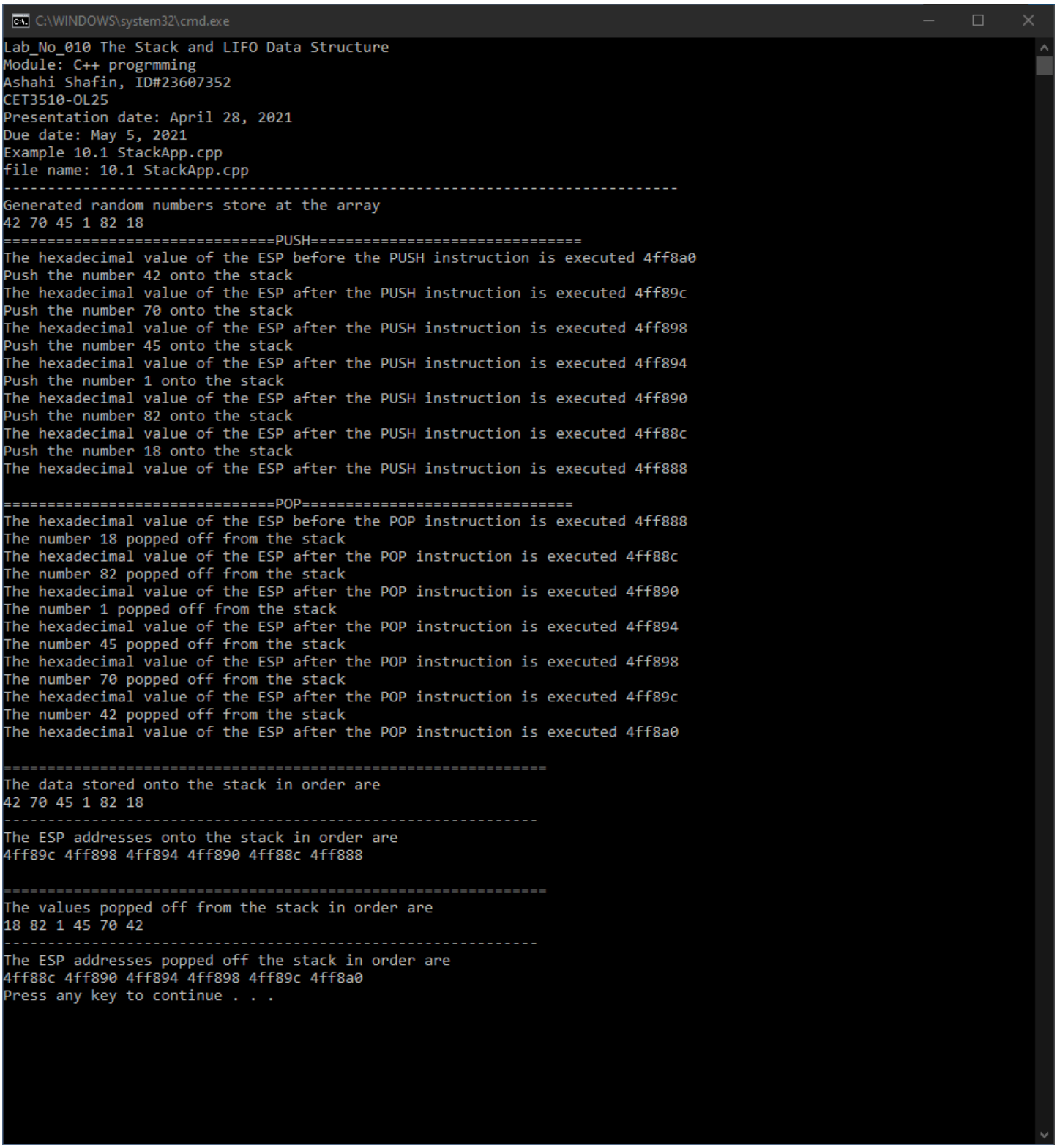
28	`cout << dec << arr_pop[i] << " ";`
	`cout << endl;`
	`cout << "------------------------------------------------------------" << endl;`
	`cout << "The ESP addresses popped off the stack in order are" << endl;`

The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.	
29	`for (i = 0; i < size; i++)`
loop basically just **means**: don't **do** any loop setup; loop forever (breaks, returns and so forth notwithstanding); and. don't **do** any post-iteration processing.	
30	`cout << hex << stack_addr_pop[i] << " ";` `cout << endl;`
The **cout** object in **C++** is an object of class ostream. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.	
31	`system("pause");`
The **system**() function is a part of the C/**C++** standard library. It is used to pass the commands that can be executed in the command processor or the terminal of the operating **system**, and finally returns the command after it has been completed.	
32	`exit(0);`
The **exit** function, declared in <stdlib. h>, terminates a **C++** program. ... By convention, a return code of zero **means** that the program completed successfully. You can use the constants EXIT_FAILURE and EXIT_SUCCESS , also **defined** in <stdlib. h>, to indicate success or failure of your program.	
33	`return 0;`
The **return** statement **returns** the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements.	

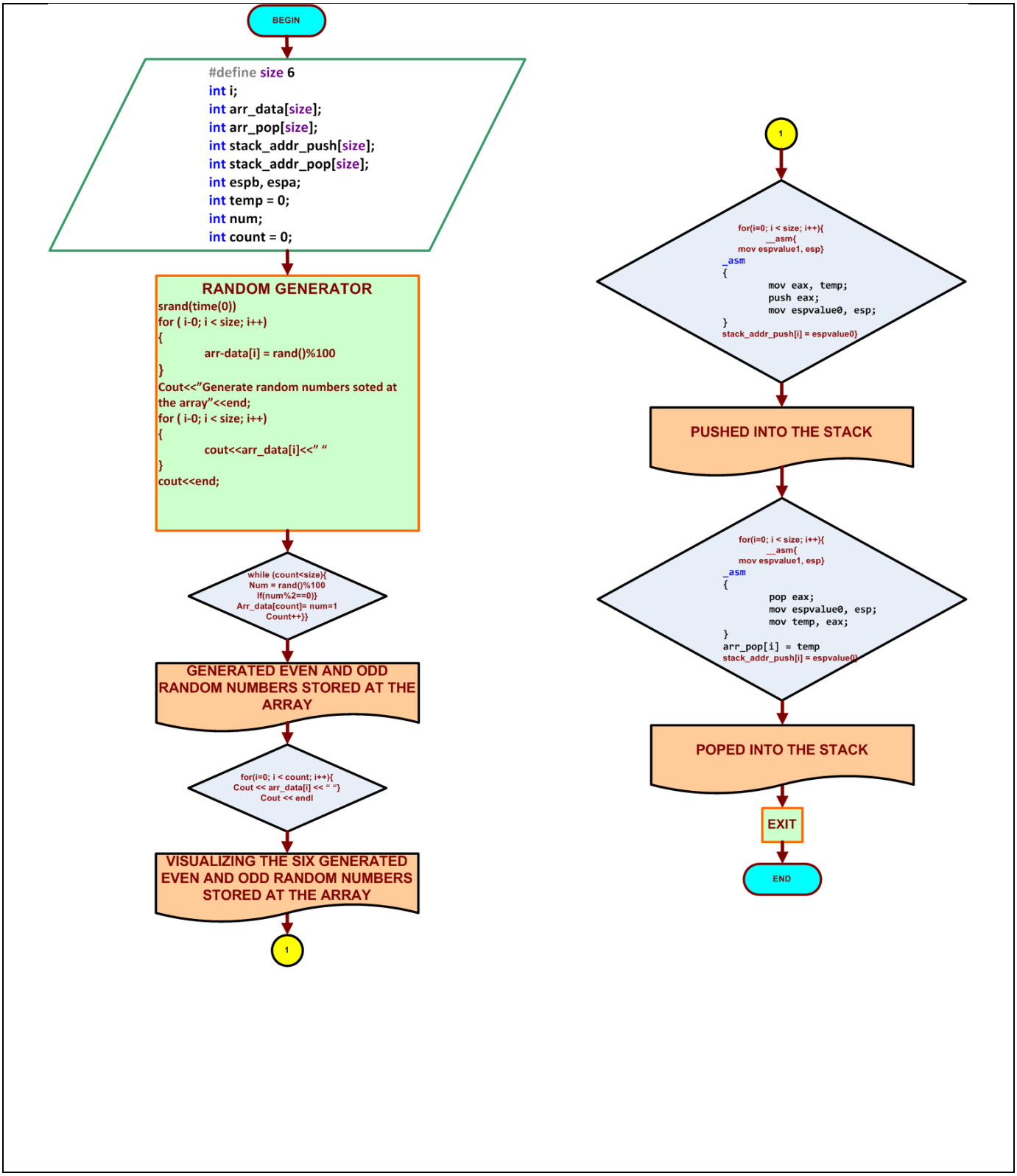
6. EXPLANING OUTPUT RESULTS

Laboratory No_010 Example 10.1

i. This is the output console of example 10.1 for laboratory 10. It shows the six random generated numbers and then they get popped and pushed. It also shows the hexadecimal value of each numbers and if they were onto the stack or popped from the stacked. The it shows order for onto and addressees for push and then the popped off and addresses in order too.


C:\WINDOWS\system32\cmd.exe
Lab_No_010 The Stack and LIFO Data Structure
Module: C++ progrmming
Ashahi Shafin, ID#23607352
CET3510-OL25
Presentation date: April 28, 2021
Due date: May 5, 2021
Example 10.1 StackApp.cpp
file name: 10.1 StackApp.cpp
Generated random numbers store at the array
42 70 45 1 82 18
==========PUSH==========
The hexadecimal value of the ESP before the PUSH instruction is executed 4ff8a0
Push the number 42 onto the stack
The hexadecimal value of the ESP after the PUSH instruction is executed 4ff89c
Push the number 70 onto the stack
The hexadecimal value of the ESP after the PUSH instruction is executed 4ff898
Push the number 45 onto the stack
The hexadecimal value of the ESP after the PUSH instruction is executed 4ff894
Push the number 1 onto the stack
The hexadecimal value of the ESP after the PUSH instruction is executed 4ff890
Push the number 82 onto the stack
The hexadecimal value of the ESP after the PUSH instruction is executed 4ff88c
Push the number 18 onto the stack
The hexadecimal value of the ESP after the PUSH instruction is executed 4ff888
==========POP==========
The hexadecimal value of the ESP before the POP instruction is executed 4ff888
The number 18 popped off from the stack
The hexadecimal value of the ESP after the POP instruction is executed 4ff88c
The number 82 popped off from the stack
The hexadecimal value of the ESP after the POP instruction is executed 4ff890
The number 1 popped off from the stack
The hexadecimal value of the ESP after the POP instruction is executed 4ff894
The number 45 popped off from the stack
The hexadecimal value of the ESP after the POP instruction is executed 4ff898
The number 70 popped off from the stack
The hexadecimal value of the ESP after the POP instruction is executed 4ff89c
The number 42 popped off from the stack
The hexadecimal value of the ESP after the POP instruction is executed 4ff8a0
The data stored onto the stack in order are
42 70 45 1 82 18
The ESP addresses onto the stack in order are
4ff89c 4ff898 4ff894 4ff890 4ff88c 4ff888
The values popped off from the stack in order are
18 82 1 45 70 42
The ESP addresses popped off the stack in order are
4ff88c 4ff890 4ff894 4ff898 4ff89c 4ff8a0
Press any key to continue . . .

7. *PROGRAM FLOW CHART:*



8. Table No 10.1 for even and odd number

Instruction	*The ESP address after execution of PUSH or POP*	*PUSH or POP integer data from the top of stack*
PUSH	*0x4ff89c*	*42*
	0x4ff898	*70*
	0x4ff894	*75*
	0x4ff890	*1*
	0x4ff89c	*82*
	0x4ff888	*18*
POP	*0x4ff88c*	*18*
	0x4ff890	*82*
	0x4ff894	*1*
	0x4ff898	*45*
	0x4ff89c	*70*
	0x4ff8a0	*42*

9. COMMENTS

Comments on Outputs and Tables is that since it random evetime you have different number stored in the array and then they will be pushed and then popped with the different addresses. Then it puts them in order for use so we can see it better.

10.CONCLUSION

In this experiment, the module shows us when you have number generated in an array it can be pushed and popped with different addresses. This is because they are put in different order when they are being popped and pushed. One thing hard about this lab was that when you build the code you have make sure cout was correctly spaced so the words don't connect to the adderesses.

A real-life application for laboratory No 10 would be in a computer because computer they are always moving files and sharing them and when this is happen they have transfer 16 at the same time and this very useful with servers because they need to transfer files and connect with each other's.