**TASK 1**

Reflection gives us objects (of type Type) that describe assemblies, modules, and types. We use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.

**We load instructions from file and save in a static variable instrucionslist in SVMVirtualMachine.cs. We loop on the list and match with opcode, when we found that type then create object of Iinstruction using reflection.**

**(IInstructionWithOperand)Activator.CreateInstance(type);**

Then we return this object as required and in second function where we have operands as well, we attach these operatnds and use class **IInstructionWithOperand** for type cast.

**TASK 2**

In task number 2, we make a loop on program list. Program list, itself of IInstruction. We attach the object of Virtual machine to each IInstruction and call run method for each one to execute each instruction using for loop on list. Further more we have line number for sequencing and also the have information that debug point is added or not. We perform our function depending upon requirement.

**TASK 3**

**Outlines your mechanism for searching in additional DLL’s for SML instruction implementations and only loading assemblies containing instructions that are used into memory**

We load all files from the directory, filtering by its type “.dll”, which are used in our memory. Assembly loads all the instructions. But we take by filtering with our required thing and cast in generic type to use.

**TASK 4**

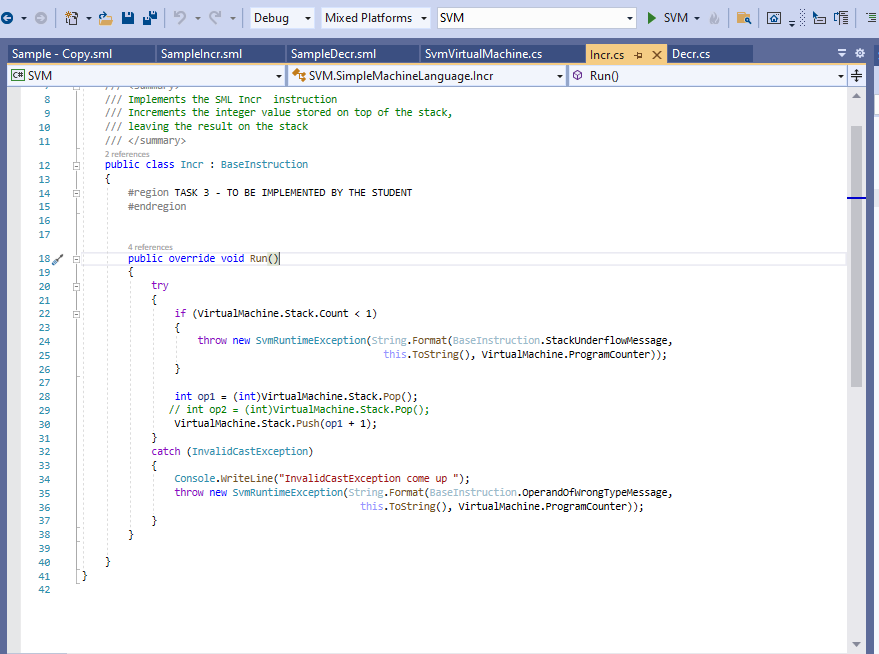
**Discusses why it make sense to refactor the SvmVirtualMachine class and introduce an IVirtualMachine interface**

SvmVirtualMachine class have properties and function which can be accessed. We make an interface named IVirtualMachine and can define fields and fnunction inside. We than later interact with these values. It will increase secure way of using and optimization and minmizing amount of memory used.

**TASK 5 –DEBUGGING**

describes how you provided an IVirtualMachine instance to the incr and decr instructions during unit testing

We inherit BaseInstruction and base instruction has Virtural Machine. Because the class SvmVirtualMachine also use inheritacnce. So basically we are not creating ay instance , we just use that interface methods.Following screen shot is also attached.

****

**includes a screenshot of the Visual Studio Test Explorer or Test Results window showing all of the unit tests in your solution and the red/green pass/fail status for each test**

**Following are short descriptions for test cases.**

**Increment**

We just make object of the Incr class and call its method run. After increment it also leaves incremented value on the top. I also attached the screen shot above and this method passed..

**Decrement**

We just make object of the Decr class and call its method run. After increment it also leaves incremented value on the top. I also attached the screen shot above and this method passed..

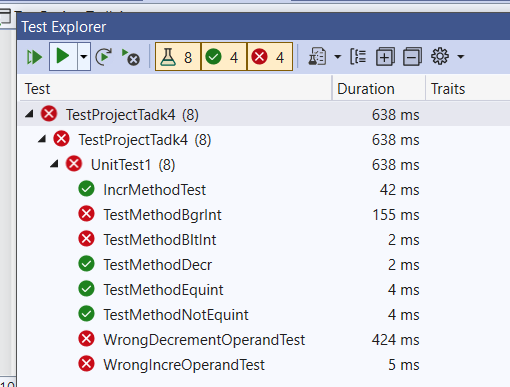
**equint value branch\_location** - if the value operand is equal to the integer value on top of the stack and we SvmRuntimeException generated if the value on top of the stack is not an integer.

**notequ branch\_location -** In this function we just check that top two values of stack are equal or not.

**bgrint value branch\_location –** By using this instruction we are just checking that the value we pushed to stack is bigger than the range of integer values. If it is bigger value then we throw a message via exception as mentioned in task.

**bltint value branch\_location–** By using this instruction we are just checking that the value we pushed to stack is less than the range of integer values. If it is less value then we throw a message via exception as mentioned in task.

I add files for sml instruction and run one by one. Further more also added test cases to check either they are working fine or not. I used same class as for Task no 4 for test cases.   
  
**bgrint value branch\_location**  and **bltint value branch\_location** .  
  
test cases of above mentioned function are false because they throw exception as required in this task for bigger value than int or less value than int. I also attached screen shot.  
  
Test cases of other two functions passed . **equint value branch\_location** and. **bgrint value branch\_location.**

****

I’m not able to load form to show debugger and stack values due to some incompatibility of .Net versions. Although I implemented. Following is the way in which I implemented. I add a field boolean isDebugLine and int Line number also with instruction.  
  
if the line is debugged then I show windows, displaying stack information and return the loop. When i get call back by click of Continue button then i resume the loop from the point where we go to debugger screen and it continue.

**Outlines your mechanism for maintaining labels for instructions, finding an instruction with a specific label and making this the next instruction to be executed**

**TASK 8 – LOOPING**

lists your SML source code for the solution to Task 8, along with a screen capture showing the output produced by this program when it is executed

For this task, I added sml file named SampleTask8.sml in solution.

Following is sml code

loadstring "Count="

loadint 10

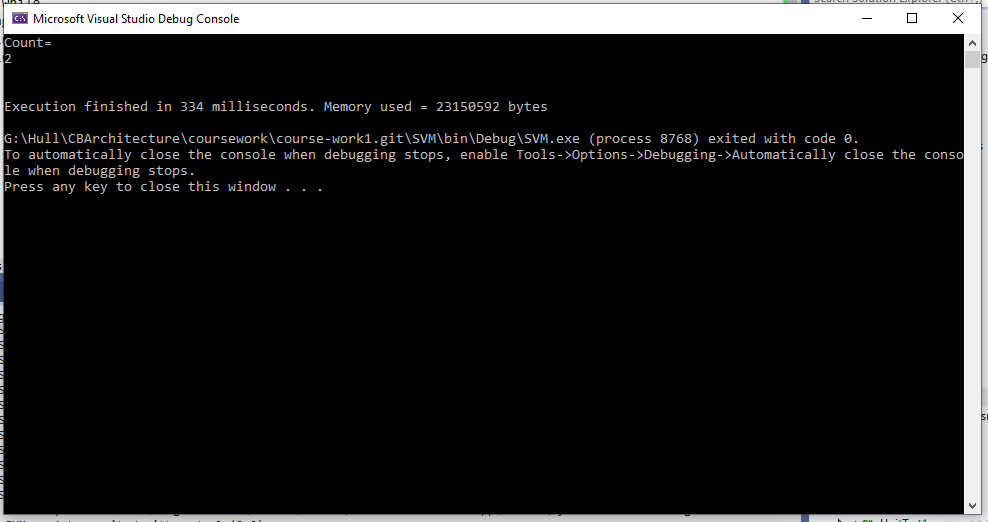
%while% decr

bltint 1 while

writestring

writestring

This function sml will run and gives us the following print.



• outline how you ensure that only instructions from strongly named assemblies are loaded

**identifies ways in which your virtual machine implantation could be optimized, in terms of improving compilation and execution speed and minimizing the amount of memory used**

To make sure that only instructions from strongly named assemblies are loaded. We have to load the assembly by their exact name and rest will be ignored. So, the assemblies that have classes in it and have some type of implementation should be available to use. Rest is garbage. And we will not use. We will make sure by name and its package. We will have to use only that fields which are required and even if we can use one field then we will not use two fields to save memory.

**The End.**