The Simple Virtual Machine

Component based architecture

Ieaun Roberts

201603546

Table of Contents

Task 1 3
Task 2 4
Task 4 4
Task 5 5
Task 7 7
Task 8 8
Optimizations 9

Task 1

Algorithm for matching opcodes to SML instructions / DLL instructions

The JITcompiler handles matching opcodes to their respective instruction.

First it attempts to find any additional assemblies by searching in its current directory for .dll files, if any are found their files are extracted and added to a list of type assembly. This operation only occurs if the list is empty to prevent constantly searching for dll's. If no assemblies are found a null value is added to the list.

After this list of additional assemblies is assigned the program attempts to match the given string opcode to either an assembly in the SVM project using reflection to find the types of methods, or to one of the additional assemblies in the .dll list.

The opcode and assembly name are both altered to lower case and compared to each other, if they match then the assembly is checked to see that it is of type "`IInstruction`".

After successfully passing those checks an instance of the instruction is created using "`Activator.CreateInstance(T) as IInstruction`"

Or

"`Activator.CreateInstance(T) as IInstructionWithOperand`"

If no matching type is found, a SvmCompilationException is thrown.

```
//Console.WriteLine("Opcode:" + opcode);
// load all assemblies
Type[] thissss = Assembly.GetExecutingAssembly().GetTypes();
foreach (Type T in thissss)
{
    //Console.WriteLine(T.Name);
    //find assembly that matches the opcode
    if (T.Name.ToLower() == opcode.ToLower())
    {
        // if it implemented type IInstruction
        if (T.BaseType.GetInterfaceMap(typeof(IInstruction)).InterfaceType.ToString()
                                            == "SVM.VirtualMachine.IInstruction")
        {
            instruction = Activator.CreateInstance(T) as IInstruction;
            //Console.WriteLine("Created _ instruction :" + T.Name);
        }
    }
}
```

Task 2

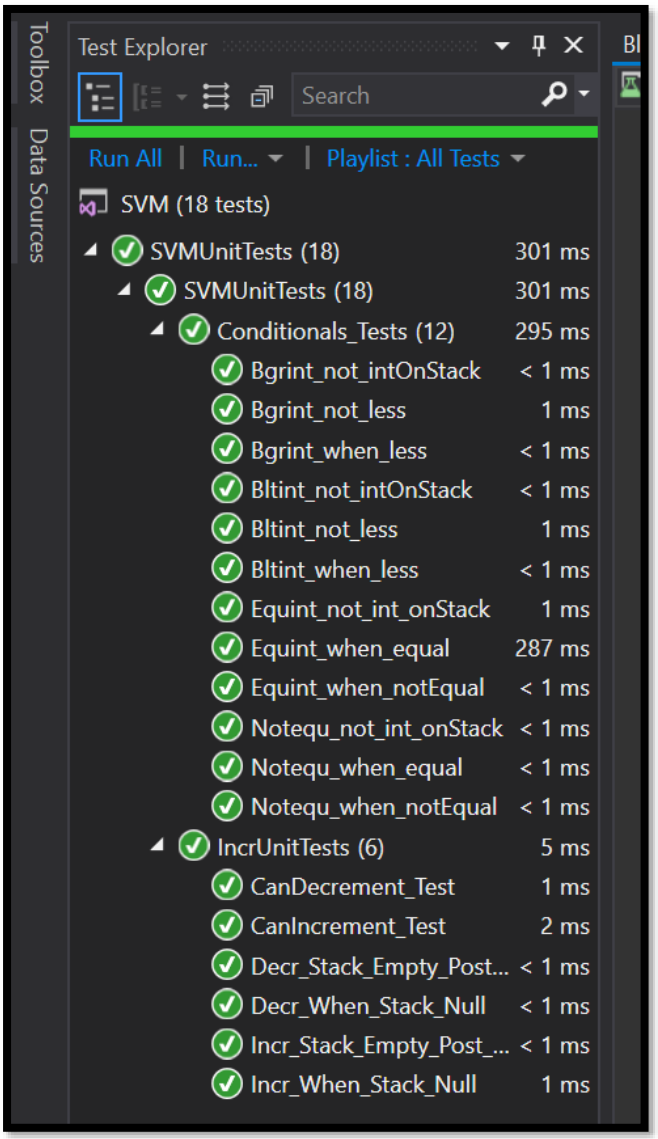
SvmVirtualMachine.Run()

Using the program counter, the method loops through the list of instructions. Executing the instruction in the list where line number = program counter.

The current instance of the virtual machine (this) is given to the instruction and instruction.run() is called which executes the instruction. After execution the program counter is incremented.

Checks are also in place to verify positions of breakpoints, if program counter = line with break point then the appropriate data is added to class “Debug_data” including the code frame and current instruction, this class implements the “IDebugFrame” interface.

“debugger” is the class that implements the “IDebugger” interface and takes “Debug_data” as a parameter as it needs a “IDebugframe” instance. This object is used to populate the debug UI form.



[Figure above] Test explorer window with Incr/Decr test and Conditional tests

Task 4

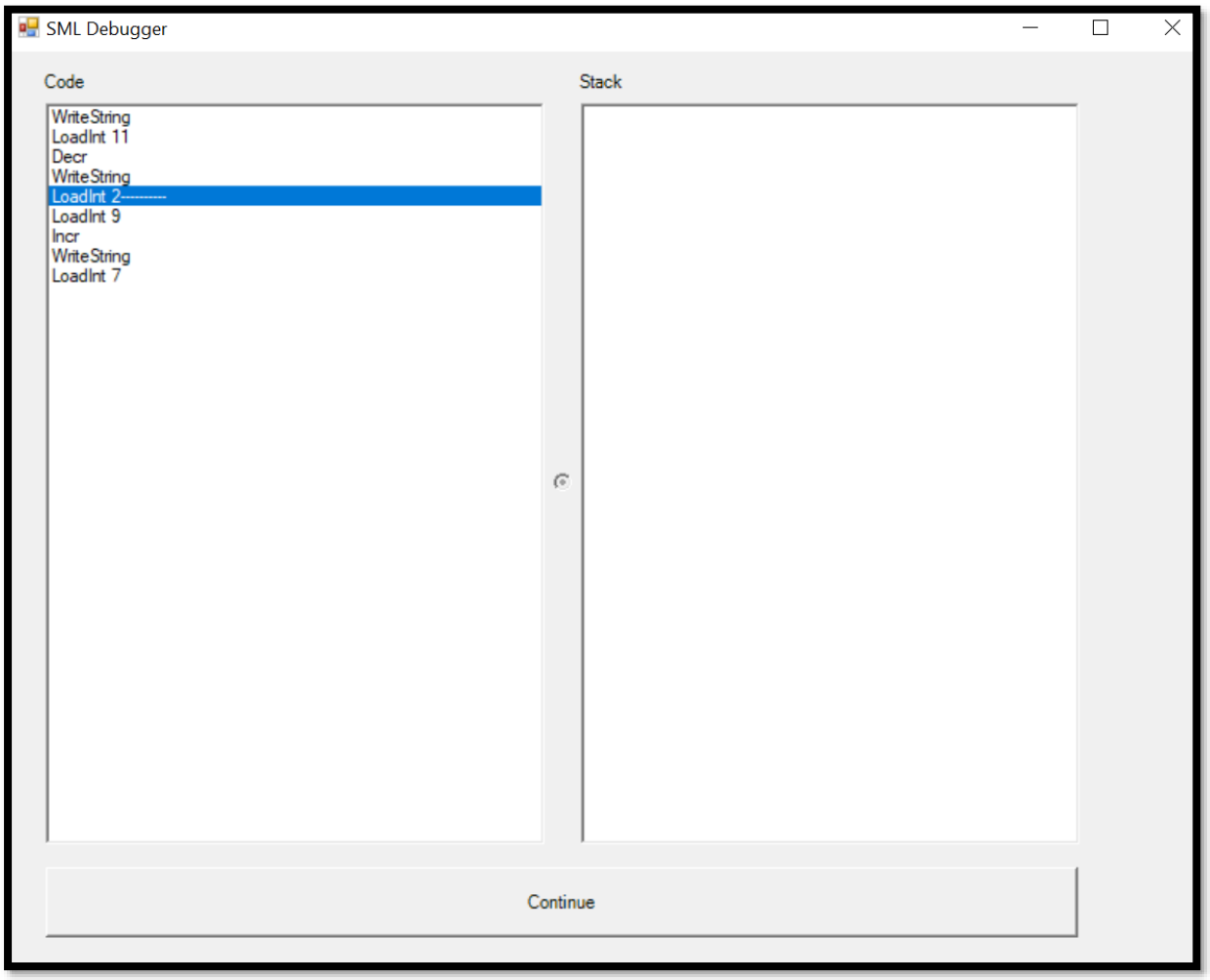
IVirtualMachine Interface

To standardize the core behavior needed by any instance of the virtual machine, the IVirtualMachine interface was created to define fields needed for normal operation.

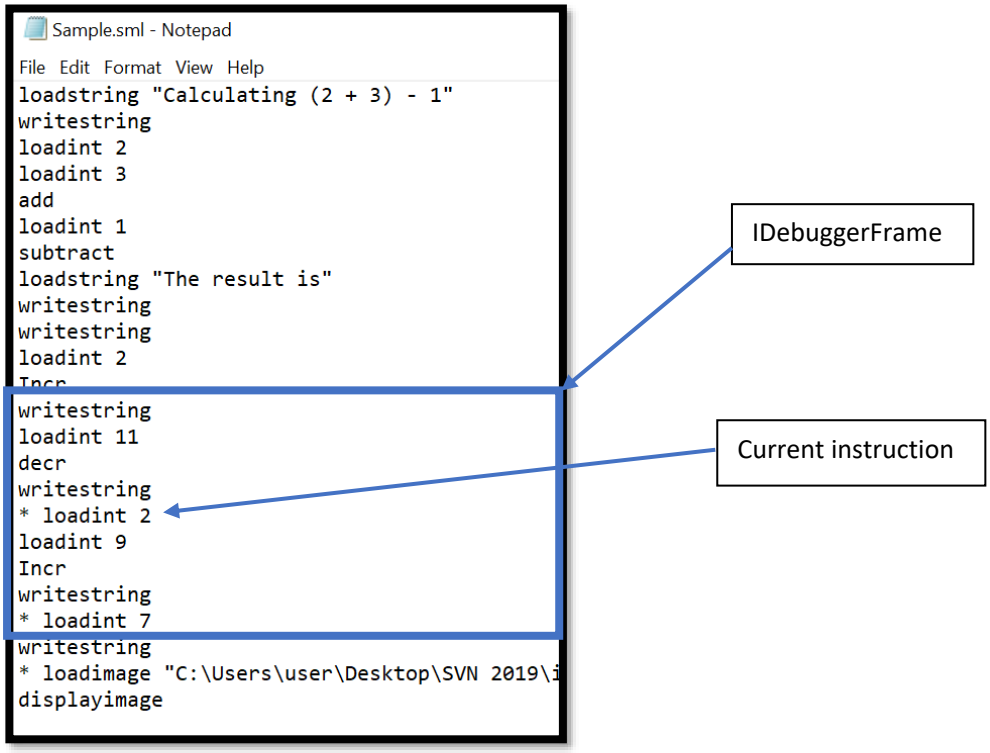
Incr/Decr Unit Tests

Using Moq, a Mock object was created to test individual units without the need to instantiate other objects, specifically without needing to instantiate SvmVirtualMachine instances. This mock object was then populated with default values and used in the unit tests as Incr and Decr require an IVirtualMachine object.

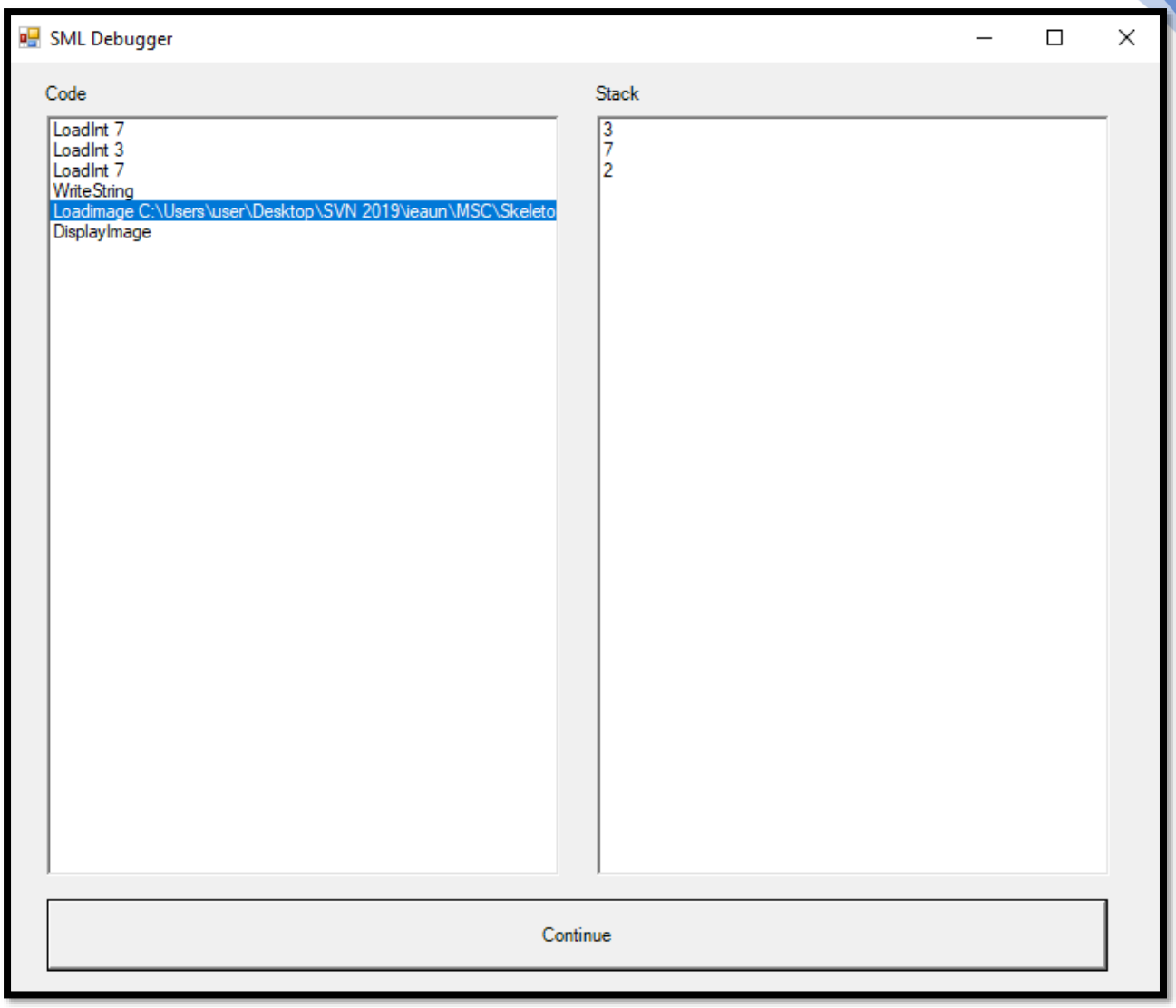
Task 5



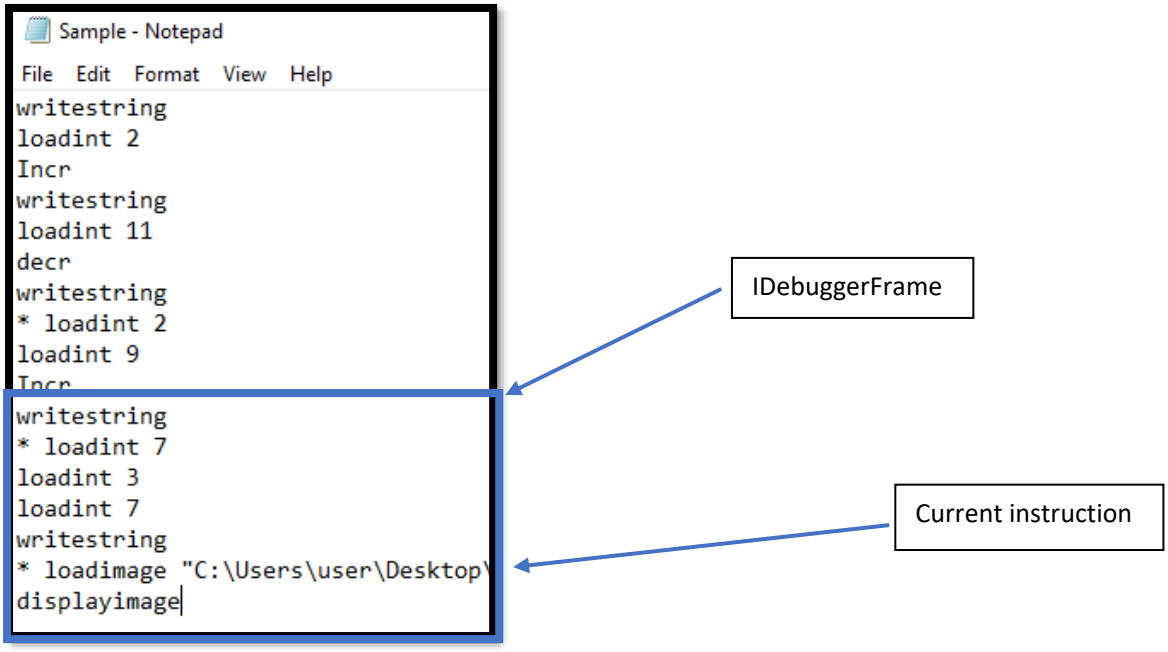
[Figure above] Debug window



[Figure above] SML code



[Figure above] Debug window with LIFO stack



[Figure above] SML code

Task 7

Labels and conditionals

Labels are procured in the method “ParseInstruction()” by searching to see if the string instruction contains a “%” as its first char to indicate a label, the resultant label and line number are stored in a hash table as a key-value pair respectively.

```
private void ParseInstruction(string instruction, int lineNumber)
{
    #region TASK 5 & 7 - MAY REQUIRE MODIFICATION BY THE STUDENT
    //Console.WriteLine("Parse Instruction: ["+ instruction+ "] Line number: ["+ lineNumber+"]");
    if (instruction.Contains("* "))
    {
        LinesWithBreakPoints.Add(lineNumber);
        Console.WriteLine("Breakpoint at line: "+ lineNumber);
        instruction = instruction.Replace("*","").Trim();
    }

    if (instruction[0] == '%')
    {
        string [] How_many = instruction.Split('%'); //(%addone%) = lenght of 3 on split
        if (How_many.Length == 3)
        {
            string [] Values = instruction.Split(' ');
            string label = Values[0].Trim();
            instruction = Values[1].Trim();
            //Console.WriteLine("Label :" + label+ " Instruction" + instruction);
            Labels.Add(label, lineNumber); //add to label hashtable
        }
    }
```

Whenever an instruction is executed, the program inspects the top item of the stack, if it finds a label it searches the hash table to find its line number and changes program counter to that line number so that on the next execution, it executes an instruction at index = program counter in the list containing all instructions.

```
//Console.WriteLine("Stack :"+ virtualMachine.stack.Count);
if (virtualMachine.stack.Count > 0)
{
    string Peek_stack_for_label = virtualMachine.stack.Peek().ToString().Trim();
    if (Peek_stack_for_label.Contains("%"))
    {
        virtualMachine.stack.Pop(); // dont need the lable there anymore
        //Console.WriteLine("------------Cond change prog counter------------");
        foreach (var key in Labels.Keys)
        {
            //Console.WriteLine(key);
        }

        //Console.WriteLine("Peek Value :" + Peek_stack_for_label);
        //Console.WriteLine("Program counter :" + programCounter);
        try
        {
            programCounter = (int)Labels[Peek_stack_for_label];
        }
        catch { Console.WriteLine("Couldnt get new programcounter value from label HT, continuing instead"); }
        //Console.WriteLine("Program counter conditional change to :" + programCounter);
        //Console.WriteLine("Going to instruction: ["+ program[programCounter] + "] Stack contents: ["+ virtual
       // Console.WriteLine("----------------------");
    }
```

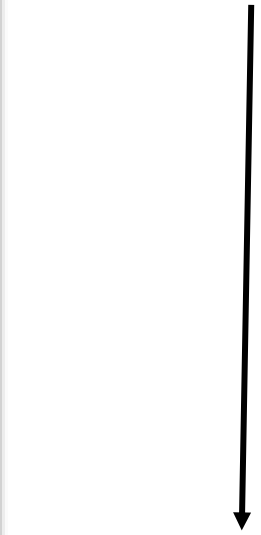
Task 8

```
In Run
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [10] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [9] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [8] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [7] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [6] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [5] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [4] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [3] On Line: 2
-----------------------
%Minus_One% Pushed to stack
-------------Cond change prog counter------------
%Minus_One%
Going to instruction: [Decr] Stack contents: [2] On Line: 2
-----------------------
Count =
1
Out Run
```

[Figure above left] In depth output

```
int count = 10;
while (count > 1)
{
    Count--;
}
Console.WriteLine("Count =");
Console.WriteLine(count);
```

[Figure above] C# code, while greater than loop



```
loadint 10
Incr
%Minus_One% Decr
bltint 1 %Minus_One%
loadstring "Count ="
writestring
writestring
```

[Figure above] SML equivalent code

```
C:\Users\user\Desktop\SVN 2019\ieaun\MSC\Skeleton Solution 1920\SVM\t
Count =
1


Execution finished in 14 milliseconds. Memory used = 13914112 bytes
```

[Figure above] Simple output

Task 9

Strongly named assemblies

To determine whether an external dll is strongly named, a check in the JITCompiler was implemented to verify that the current assembly has a “public key” / “public key token” value that is not set to null. As mentioned in task 1, if the assembly meets the criteria the program adds the it to a list (“dll_Assemblies”) to be used later in the matching process.

```
try
{
    foreach (string dll in Directory.GetFiles(Environment.CurrentDirectory, "*.dll"))
    {
        Assembly A = Assembly.LoadFile(dll);
        string Fullname = A.FullName.ToLower();
        //Console.WriteLine(Fullname);
        string[] Temp = Fullname.Split(',');
        foreach (string section in Temp)
        {
            if (section.Contains("publickey"))
            {
              // Console.WriteLine(section);
               if (!section.Contains("null"))
               {
                   Console.WriteLine(section);
                   dll_Assemblies.Add(A);
                   Console.WriteLine("Strong name----------->  " + section + " " + Fullname);
               }
            }
        }
    }
}
catch
{
    //not an assembly
}
```

Optimizations

1. Debug window – completely move away from the use of threads to signal that the continue button has been clicked. The use of thread count and creating and killing threads to increase and decrease thread count to noticeable levels to communicate is inefficient and although did work 99% of the time the 5 second delay between the form being updated (after the continue button has been clicked) and all the threads being aborted is asking too much in terms of patients from a user and threads from a processor.
2. Labels – currently the way the SVM jumps from 1 label to another is by pushing the label to the stack when encountered in a conditional instruction and the if statement validates to true (!= , ==). After the conditional instruction is done the stack contents are peeked at and if the first item on the stack is a label (%AddOne%) then it matches the label to a line number and sets the program counter to that line number.
3. Safety –task 10 has not been implemented so the most significant optimization would be the inclusion of this security feature.