Note: ProjectItems that have Task List items associated with them are output in Bold and Red

ProjectItem 'BVOperation.cs' has no task items

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace BajanVincyAssembly.Models.ComputerArchitecture
8   {
9       /// <summary>
10      /// All BV Assembly Opertions
11      /// </summary>
12      public enum BVOperation
13      {
14          ADDNS = 1,
15          ADDCONST = 2,
16          ADDPOS = 3,
17          SUBNS = 4,
18          SUBCONST = 5,
19          SUBPOS = 6,
20          LOGICAND = 7,
21          LOGICANDCOSNT = 8,
22          LOGICOR = 9,
23          LOGICORCONST = 10,
24          SHIFTLEFT = 11,
25          SHIFTLEFTPOS = 12,
26          SHIFTLEFTCONST = 13,
27          SHIFTRIGHT = 14,
28          SHIFTRIGHTPOS = 15,
29          SHIFTRIGHTCONST = 16,
30          FROMEMEM = 17,
31          FROMMEMCONST = 18,
32          FROMCONST = 19,
33          TOMEM = 20,
34          TOMEMCONST = 21,
35          TOCONST = 22,
36          TOCONSTCONST = 23,
37          COPY = 24,
38          COPYERASE = 25,
39          LESSTHEN = 26,
40          LESSTHENPOS = 27,
41          LESSTHENCONST = 28,
42          LESSTHENEQ = 29,
43          LESSTHENEQPOS = 30,
44          LESSTHENEQCONST = 31,
45          MORETHEN = 32,
46          MORETHENPOS = 33,
47          MORETHENCONST = 34,
48          MORETHENEQ = 35,
49          MORETHENEQPOS = 36,
50          MORETHENEQCONST = 37,
51          XOR = 38,
52          XORCONST = 39,
53          SAVEADDRESS = 40,
54          GOTO = 41,
55          GOTOCONST = 42,
56          EQ = 43,
57          EQCONST = 44,
58          GOTOEQ = 45,
59          GOTOEQCONST = 46,
60          GOTONOEQ = 47,
61          GOTONOEQCONST = 48,
62          GOTOMORETHEN = 49,
    1       2
```

```
63          GOTOMORETHENCONST = 50,
64          GOTOLESSTHEN = 51,
65          GOTOLESSTHENCONST = 52,
66          JUMPLABEL = 53
67      }
68
69      /// <summary>
70      /// Contains Information about BV Operations
71      /// </summary>
72      public class BVOperationInfo
73      {
74          /// <summary>
75          /// Instantiates a new instance of the <see cref="BVOperationInfo" class/>
76          /// </summary>
77          public BVOperationInfo()
78          {
79
80          }
81
82          /// <summary>
83          /// Gets BV Operation Loopkup Dictionary
84          /// </summary>
85          public static readonly Dictionary<string, BVOperation> BVOperationLookup =
            new Dictionary<string, BVOperation>()
86          {
87              { "addns", BVOperation.ADDNS },
88              { "addconst", BVOperation.ADDCONST },
89              { "addpos", BVOperation.ADDPOS },
90              { "subns", BVOperation.SUBNS },
91              { "subconst", BVOperation.SUBCONST },
92              { "subpos", BVOperation.SUBPOS },
93              { "logicand", BVOperation.LOGICAND },
94              { "logicandcosnt", BVOperation.LOGICANDCOSNT },
95              { "logicor", BVOperation.LOGICOR },
96              { "logicorconst", BVOperation.LOGICORCONST },
97              { "shiftleft", BVOperation.SHIFTLEFT },
98              { "shiftleftpos", BVOperation.SHIFTLEFTPOS },
99              { "shiftleftconst", BVOperation.SHIFTLEFTCONST },
100             { "shiftright", BVOperation.SHIFTRIGHT },
101             { "shiftrightpos", BVOperation.SHIFTRIGHTPOS },
102             { "shiftrightconst", BVOperation.SHIFTRIGHTCONST },
103             { "frommem", BVOperation.FROMEMEM },
104             { "frommemconst", BVOperation.FROMMEMCONST },
105             { "fromconst", BVOperation.FROMCONST },
106             { "tomem", BVOperation.TOMEM },
107             { "tomemconst", BVOperation.TOMEMCONST },
108             { "toconst", BVOperation.TOCONST },
109             { "toconstconst", BVOperation.TOCONSTCONST },
110             { "copy", BVOperation.COPY },
111             { "copyerase", BVOperation.COPYERASE },
112             { "lessthen", BVOperation.LESSTHEN },
113             { "lessthenpos", BVOperation.LESSTHENPOS },
114             { "lessthenconst", BVOperation.LESSTHENCONST },
115             { "lesstheneq", BVOperation.LESSTHENEQ },
116             { "lesstheneqpos", BVOperation.LESSTHENEQPOS },
117             { "lesstheneqconst", BVOperation.LESSTHENEQCONST },
118             { "morethen", BVOperation.MORETHEN },
119             { "morethenpos", BVOperation.MORETHENPOS },
120             { "morethenconst", BVOperation.MORETHENCONST },
121             { "moretheneq", BVOperation.MORETHENEQ },
122             { "moretheneqpos", BVOperation.MORETHENEQPOS },
123             { "moretheneqconst", BVOperation.MORETHENEQCONST },
124             { "xor", BVOperation.XOR },
125             { "xorconst", BVOperation.XORCONST },
```

```
126             { "saveaddress", BVOperation.SAVEADDRESS },
127             { "goto", BVOperation.GOTO },
128             { "gotoconst", BVOperation.GOTOCONST },
129             { "eq", BVOperation.EQ },
130             { "eqconst", BVOperation.EQCONST },
131             { "gotoeq", BVOperation.GOTOEQ },
132             { "gotoeqconst", BVOperation.GOTOEQCONST },
133             { "gotonoeq", BVOperation.GOTONOEQ },
134             { "gotonoeqconst", BVOperation.GOTONOEQCONST },
135             { "gotomorethan", BVOperation.GOTOMORETHEN },
136             { "gotomorethanconst", BVOperation.GOTOMORETHENCONST },
137             { "gotolessthen", BVOperation.GOTOLESSTHEN },
138             { "gotolessthenconst", BVOperation.GOTOLESSTHENCONST }
139         };
140
141         /// <summary>
142         /// Gets BV Operation Code Loopkup Dictionary
143         /// </summary>
144         public static readonly Dictionary<BVOperation, string>
            BVOperationCodeLookup = new Dictionary<BVOperation, string>()
145         {
146             { BVOperation.ADDNS, "000001 " },
147             { BVOperation.ADDCONST, "000010 " },
148             { BVOperation.ADDPOS, "000011 " },
149             { BVOperation.SUBNS, "000100 " },
150             { BVOperation.SUBCONST, "000101 " },
151             { BVOperation.SUBPOS, "000110 " },
152             { BVOperation.LOGICAND, "000111 " },
153             { BVOperation.LOGICANDCOSNT, "001000 " },
154             { BVOperation.LOGICOR, "001001 " },
155             { BVOperation.LOGICORCONST, "001010 " },
156             { BVOperation.SHIFTLEFT, "001011 " },
157             { BVOperation.SHIFTLEFTPOS, "001100 " },
158             { BVOperation.SHIFTLEFTCONST, "001101 " },
159             { BVOperation.SHIFTRIGHT, "001110 " },
160             { BVOperation.SHIFTRIGHTPOS, "001111 " },
161             { BVOperation.SHIFTRIGHTCONST, "010000 " },
162             { BVOperation.FROMEMEM, "010001 " },
163             { BVOperation.FROMMEMCONST, "010010 " },
164             { BVOperation.FROMCONST, "010011 " },
165             { BVOperation.TOMEM, "010100 " },
166             { BVOperation.TOMEMCONST, "010101 " },
167             { BVOperation.TOCONST, "010110 " },
168             { BVOperation.TOCONSTCONST, "010111 " },
169             { BVOperation.COPY, "011000 " },
170             { BVOperation.COPYERASE, "011001 " },
171             { BVOperation.LESSTHEN, "011010 " },
172             { BVOperation.LESSTHENPOS, "011011 " },
173             { BVOperation.LESSTHENCONST, "011100 " },
174             { BVOperation.LESSTHENEQ, "011101 " },
175             { BVOperation.LESSTHENEQPOS, "011110 " },
176             { BVOperation.LESSTHENEQCONST, "011111 " },
177             { BVOperation.MORETHEN, "100000 " },
178             { BVOperation.MORETHENPOS, "100001 " },
179             { BVOperation.MORETHENCONST, "100010 " },
180             { BVOperation.MORETHENEQ, "100011 " },
181             { BVOperation.MORETHENEQPOS, "100100 " },
182             { BVOperation.MORETHENEQCONST, "100101 " },
183             { BVOperation.XOR, "100110 " },
184             { BVOperation.XORCONST, "100111 " },
185             { BVOperation.SAVEADDRESS, "101000 " },
186             { BVOperation.GOTO, "101001 " },
187             { BVOperation.GOTOCONST, "101010 " },
188             { BVOperation.EQ, "101011 " },
```

```
189                 { BVOperation.EQCONST, "101100 " },
190                 { BVOperation.GOTOEQ, "101101 " },
191                 { BVOperation.GOTOEQCONST, "101110 " },
192                 { BVOperation.GOTONOEQ, "101111 " },
193                 { BVOperation.GOTONOEQCONST, "110000 " },
194                 { BVOperation.GOTOMORETHEN, "110001 " },
195                 { BVOperation.GOTOMORETHENCONST, "110010 " },
196                 { BVOperation.GOTOLESSTHEN, "110011 " },
197                 { BVOperation.GOTOLESSTHENCONST, "110100 " },
198                 { BVOperation.JUMPLABEL, "110101 " }
199             };
200         }
201     }
```

ProjectItem 'Instruction.cs' has no task items

```
1   using BajanVincyAssembly.Services.Registers;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace BajanVincyAssembly.Models.ComputerArchitecture
9   {
10      /// <summary>
11      /// Contains instruction information
12      /// </summary>
13      public class Instruction
14      {
15          /// <summary>
16          /// Instantiates a new instance of the <see cref="Instruction" class/>
17          /// </summary>
18          public Instruction()
19          {
20
21          }
22
23          public static int InstructionAddressPointer = 200;
24
25          /// <summary>
26          /// Gets or sets Instruction Address
27          /// </summary>
28          public int InstructionAddress { get; set; } = InstructionAddressPointer;
29
30          /// <summary>
31          /// Gets or sets BV Operation
32          /// </summary>
33          public BVOperation Operation { get; set; }
34
35          /// <summary>
36          /// Gets or sets destination register
37          /// </summary>
38          public string DestinationRegister { get; set; }
39
40          /// <summary>
41          /// Gets or sets destination value
42          /// </summary>
43          public int DestinationValue { get; set; }
44
45          /// <summary>
46          /// Operand A Register
47          /// </summary>
48          public string OperandARegister { get; set; }
49
50          /// <summary>
51          /// Gets or sets Operand A
52          /// </summary>
53          public int OperandA { get; set; }
54
55          /// <summary>
56          /// Operand B Register
57          /// </summary>
58          public string OperandBRegister { get; set; }
59
60          /// <summary>
61          /// Gets or sets Operand B
62          /// </summary>
```

1 2

```
63          public int OperandB { get; set; }
64
65          /// <summary>
66          /// Gets or sets Operand Immediate
67          /// </summary>
68          public int OperandImmediate { get; set; }
69
70          /// <summary>
71          /// Gets or sets Offset
72          /// </summary>
73          public int Offset { get; set; }
74
75          /// <summary>
76          /// Gets or sets jump label
77          /// </summary>
78          public string JumpLabel { get; set; }
79
80          /// <summary>
81          /// Prints Binary Format of Instruction
82          /// </summary>
83          /// <returns></returns>
84          public override string ToString()
85          {
86              return $"{BVOperationInfo.BVOperationCodeLookup[this.Operation]} | {
                (!string.IsNullOrEmpty(DestinationRegister) ?
                Registry.registerAddressLookup[DestinationRegister] :
                Convert.ToString(0, 2).PadLeft(5, '0'))} | { (!string.IsNullOrEmpty
                (OperandARegister) ? Registry.registerAddressLookup[OperandARegister]
                : Convert.ToString(0, 2).PadLeft(5, '0'))} | { (!string.IsNullOrEmpty
                (OperandBRegister) ? Registry.registerAddressLookup[OperandBRegister]
                : Convert.ToString(0, 2).PadLeft(5, '0'))} | { Convert.ToString
                (OperandImmediate, 2).PadLeft(16, '0')}";
87          }
88      }
89  }
```

ProjectItem 'Register.cs' has no task items

```
1   namespace BajanVincyAssembly.Models.ComputerArchitecture
2   {
3       /// <summary>
4       /// Contains information about a register
5       /// </summary>
6       public class Register
7       {
8           /// <summary>
9           /// Instantiates a new instance of the <see cref="Register" class/>
10          /// </summary>
11          /// <param name="name"></param>
12          public Register(string name, int base10Value = 0, string word = "")
13          {
14              this.Name = name;
15              this.Base10Value = base10Value;
16              this.Word = word;
17          }
18
19          public Register()
20          {
21
22          }
23
24          /// <summary>
25          /// Gets or sets Name
26          /// </summary>
27          public string Name { get; set; }
28
29          /// <summary>
30          /// Gets or sets Base 10 Value
31          /// </summary>
32          public int Base10Value { get; set; }
33
34          /// <summary>
35          /// Gets or sets Hex Value
36          /// </summary>
37          public string HexValue { get { return $"0X{string.Format("{0:x8}",
            this.Base10Value)}"; } }
38
39          /// <summary>
40          /// Gets or sets Word
41          /// </summary>
42          public string Word { get; set; }
43
44          /// <summary>
45          /// Clears Register
46          /// </summary>
47          public void Clear()
48          {
49              this.Base10Value = 0;
50          }
51      }
52  }
```

ProjectItem 'ObjectExtensions.cs' has no task items

```
1   using Newtonsoft.Json;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace BajanVincyAssembly.Models
9   {
10      /// <summary>
11      /// Collection of extension methods for all objects
12      /// </summary>
13      public static class ObjectExtensions
14      {
15          /// <summary>
16          /// Performs and returns deep clone copy of an object
17          /// </summary>
18          /// <typeparam name="T">Class Type</typeparam>
19          /// <param name="currentObject"> Object to clone</param>
20          /// <returns></returns>
21          public static T DeepClone<T> (this T currentObject)
22          {
23              if (currentObject == null)
24              {
25                  return default(T);
26              }
27
28              var objectSerialized = JsonConvert.SerializeObject(currentObject);
29              var objectDeserialized = JsonConvert.DeserializeObject<T>
                (objectSerialized);
30
31              return objectDeserialized;
32          }
33      }
34  }
```

ProjectItem 'ValidationInfo.cs' has no task items

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace BajanVincyAssembly.Models.Validation
8   {
9       /// <summary>
10      /// Contains Validation Information
11      /// </summary>
12      public class ValidationInfo
13      {
14          /// <summary>
15          /// Instantiates a new instance of the <see cref="ValidationInfo" class/>
16          /// </summary>
17          public ValidationInfo()
18          {
19          }
20
21          /// <summary>
22          /// Gets or sets IsValid
23          /// </summary>
24          public bool IsValid { get; set; } = true;
25
26          /// <summary>
27          /// Gets or sets validation messages
28          /// </summary>
29          public List<string> ValidationMessages { get; set; } = new List<string>();
30      }
31  }
```

ProjectItem 'BVCompiler.cs' has no task items

```
1   using BajanVincyAssembly.Models;
2   using BajanVincyAssembly.Models.ComputerArchitecture;
3   using BajanVincyAssembly.Models.Validation;
4   using System;
5   using System.Collections.Generic;
6   using System.Linq;
7   using System.Text;
8   using System.Threading.Tasks;
9
10  namespace BajanVincyAssembly.Services.Compilers
11  {
12      /// <summary>
13      /// Compiler for BV Assembly Code
14      /// </summary>
15      public class BVCompiler : ICompile<Instruction>
16      {
17          /// <summary>
18          /// Instantiates a new instance of the <see cref="BVCompiler" class/>
19          /// </summary>
20          public BVCompiler()
21          {
22
23          }
24
25          /// <summary>
26          /// Line Delimitter for assembly code
27          /// </summary>
28          public static readonly string[] LineDelimitter = { "\r", "\n" };
29
30          /// <summary>
31          /// Comment Signature
32          /// </summary>
33          public static readonly string CommentSignarture = "//";
34
35          /// <inheritdoc cref="ICompile{T}"/>
36          public IEnumerable<Instruction> Compile(string code)
37          {
38              IEnumerable<Instruction> instructions = new List<Instruction>();
39
40              if (this.ValidateCode(code).IsValid)
41              {
42                  IEnumerable<string> linesOfCode =
                    this.GetLinesOfCodeWithNoComments(code);
43
44                  // Build Instructions from code and return them
45                  BVInstructionBuilder bvInstructionBuilder = new
                    BVInstructionBuilder();
46                  instructions = linesOfCode.Select(lineOfCode =>
                    bvInstructionBuilder.BuildInstruction(lineOfCode));
47              }
48
49              return instructions;
50          }
51
52          /// <inheritdoc cref="ICompile{T}"/>
53          public ValidationInfo ValidateCode(string code)
54          {
55              ValidationInfo validationInfo = new ValidationInfo();
56
57              IEnumerable<string> linesOfCode = this.GetLinesOfCodeWithNoComments
                (code);
58
    1   2   3
```

```
59              if (linesOfCode.Any())
60              {
61                  BVOperationValidationChecks bvOperationValidationChecks = new
                    BVOperationValidationChecks(linesOfCode);
62                  validationInfo =
                    bvOperationValidationChecks.ValidationInfo.DeepClone();
63              }
64
65              return validationInfo;
66          }
67
68          /// <summary>
69          /// Gets Lines of Code with no comments
70          /// </summary>
71          /// <param name="code">Code</param>
72          /// <returns>Collection of strings</returns>
73          private IEnumerable<string> GetLinesOfCodeWithNoComments(string code)
74          {
75              IEnumerable<string> linesOfCode = new List<string>();
76
77              if (!string.IsNullOrEmpty(code))
78              {
79                  linesOfCode = code.Split(BVCompiler.LineDelimitter,
                    StringSplitOptions.RemoveEmptyEntries);
80
81                  if (linesOfCode.Any())
82                  {
83                      // Removes comment lines
84                      linesOfCode = linesOfCode.Where(line =>
85                      {
86                          var lineTrimmed = line.Trim().ToLower();
87
88                          var first2Characters = lineTrimmed.Substring(0, 2);
89
90                          var lineIsAComment = string.Equals(first2Characters,
                        BVCompiler.CommentSignarture,
                        StringComparison.InvariantCultureIgnoreCase);
91
92                          if (lineIsAComment)
93                          {
94                              return false;
95                          }
96                          else
97                          {
98                              return true;
99                          }
100                     }).ToList();
101
102                     // Trims Spaces around code lines
103                     if (linesOfCode.Any())
104                     {
105                         linesOfCode = linesOfCode.Select(line => line.Trim());
106                     }
107                 }
108             }
109
110             return linesOfCode;
111         }
112     }
113 }
```

ProjectItem 'BVInstructionBuilder.cs' has no task items

```
1   using BajanVincyAssembly.Models.ComputerArchitecture;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace BajanVincyAssembly.Services.Compilers
9   {
10      /// <summary>
11      /// Transforms BV Assembly into Insructions
12      /// </summary>
13      public class BVInstructionBuilder
14      {
15          public BVInstructionBuilder()
16          {
17
18          }
19
20          /// <summary>
21          /// Builds a BV Instruction from a line of code
22          /// </summary>
23          /// <param name="lineOfCode"></param>
24          /// <returns></returns>
25          public Instruction BuildInstruction(string lineOfCode)
26          {
27              Instruction.InstructionAddressPointer += 32;
28              Instruction instruction = new Instruction();
29
30              string[] operationPartsSplitter = { " " };
31              var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
32
33              if (!operationParts.Any())
34              {
35                  throw new Exception($"Invalid Instruction Found: -> $
                    {lineOfCode}");
36              }
37
38              string operationFound = operationParts[0].Trim().ToLower();
39              bool jumpLabelFound =
                BVOperationValidationChecks.JumpLabelregex.Match
                (operationFound).Success;
40
41              if (!jumpLabelFound && !BVOperationInfo.BVOperationLookup.ContainsKey
                (operationFound))
42              {
43                  throw new Exception( $"Invalid Operation Found: -> $
                    {lineOfCode}");
44              }
45
46              BVOperation operation = jumpLabelFound ? BVOperation.JUMPLABEL :
                BVOperationInfo.BVOperationLookup[operationFound];
47
48              instruction.Operation = operation;
49
50              switch (operation)
51              {
52              case BVOperation.ADDNS:
53                      instruction.DestinationRegister = operationParts[1].Replace
                    (",", "").Trim();
```

```
54                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
55                  instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
56                  break;
57              case BVOperation.ADDCONST:
58                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
59                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
60                  instruction.OperandImmediate = int.Parse(operationParts[3]);
61                  break;
62              case BVOperation.ADDPOS:
63                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
64                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
65                  instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
66                  break;
67              case BVOperation.SUBNS:
68                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
69                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
70                  instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
71                  break;
72              case BVOperation.SUBCONST:
73                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
74                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
75                  instruction.OperandImmediate = int.Parse(operationParts[3]);
76                  break;
77              case BVOperation.SUBPOS:
78                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
79                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
80                  instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
81                  break;
82              case BVOperation.LOGICAND:
83                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
84                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
85                  instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
86                  break;
87              case BVOperation.LOGICANDCOSNT:
88                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
89                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
90                  instruction.OperandImmediate = int.Parse(operationParts[3]);
91                  break;
92              case BVOperation.LOGICOR:
93                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
94                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
```

```
95                  instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
96                  break;
97              case BVOperation.LOGICORCONST:
98                  instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
99                  instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
100                 instruction.OperandImmediate = int.Parse(operationParts[3]);
101                 break;
102             case BVOperation.SHIFTLEFT:
103                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
104                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
105                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
106                 break;
107             case BVOperation.SHIFTLEFTPOS:
108                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
109                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
110                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
111                 break;
112             case BVOperation.SHIFTLEFTCONST:
113                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
114                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
115                 instruction.OperandImmediate = int.Parse(operationParts[3]);
116                 break;
117             case BVOperation.SHIFTRIGHT:
118                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
119                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
120                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
121                 break;
122             case BVOperation.SHIFTRIGHTPOS:
123                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
124                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
125                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
126                 break;
127             case BVOperation.SHIFTRIGHTCONST:
128                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
129                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
130                 instruction.OperandImmediate = int.Parse(operationParts[3]);
131                 break;
132             case BVOperation.FROMEMEM:
133                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
134                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
135                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
136                 break;
```

```
137             case BVOperation.FROMMEMCONST:
138                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
139                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
140                 instruction.OperandImmediate = int.Parse(operationParts[3]);
141                 break;
142             case BVOperation.FROMCONST:
143                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
144                 instruction.OperandImmediate = int.Parse(operationParts[2]);
145                 break;
146             case BVOperation.TOMEM:
147                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
148                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
149                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
150                 break;
151             case BVOperation.TOMEMCONST:
152                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
153                 instruction.OperandARegister = operationParts[3].Replace(",",
                "").Trim();
154                 instruction.OperandImmediate = int.Parse(operationParts[2]);
155                 break;
156             case BVOperation.TOCONST:
157                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
158                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
159                 instruction.OperandImmediate = int.Parse(operationParts[3]);
160                 break;
161             case BVOperation.TOCONSTCONST:
162
163                 break;
164             case BVOperation.COPY:
165                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
166                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
167                 break;
168             case BVOperation.COPYERASE:
169                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
170                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
171                 break;
172             case BVOperation.LESSTHEN:
173                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
174                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
175                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
176                 break;
177             case BVOperation.LESSTHENPOS:
178                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
179                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
180                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
```

```
181                 break;
182             case BVOperation.LESSTHENCONST:
183                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
184                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
185                 instruction.OperandImmediate = int.Parse(operationParts[3]);
186                 break;
187             case BVOperation.LESSTHENEQ:
188                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
189                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
190                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
191                 break;
192             case BVOperation.LESSTHENEQPOS:
193                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
194                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
195                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
196                 break;
197             case BVOperation.LESSTHENEQCONST:
198                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
199                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
200                 instruction.OperandImmediate = int.Parse(operationParts[3]);
201                 break;
202             case BVOperation.MORETHEN:
203                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
204                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
205                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
206                 break;
207             case BVOperation.MORETHENPOS:
208                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
209                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
210                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
211                 break;
212             case BVOperation.MORETHENCONST:
213                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
214                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
215                 instruction.OperandImmediate = int.Parse(operationParts[3]);
216                 break;
217             case BVOperation.MORETHENEQ:
218                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
219                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
220                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
221                 break;
222             case BVOperation.MORETHENEQPOS:
```

```
223                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
224                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
225                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
226                 break;
227             case BVOperation.MORETHENEQCONST:
228                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
229                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
230                 instruction.OperandImmediate = int.Parse(operationParts[3]);
231                 break;
232             case BVOperation.XOR:
233                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
234                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
235                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
236                 break;
237             case BVOperation.XORCONST:
238                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
239                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
240                 instruction.OperandImmediate = int.Parse(operationParts[3]);
241                 break;
242             case BVOperation.SAVEADDRESS:
243                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
244                 instruction.JumpLabel = operationParts[2].Replace(",",
                "").Trim();
245                 instruction.OperandImmediate =
                instruction.InstructionAddress;
246                 break;
247             case BVOperation.GOTO:
248                 instruction.OperandARegister = operationParts[1].Replace(",",
                "").Trim();
249                 break;
250             case BVOperation.EQ:
251                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
252                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
253                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
254                 break;
255             case BVOperation.EQCONST:
256                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
257                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
258                 instruction.OperandImmediate = int.Parse(operationParts[3]);
259                 break;
260             case BVOperation.GOTOEQ:
261                 instruction.DestinationRegister = operationParts[1].Replace
                (",", "").Trim();
262                 instruction.OperandARegister = operationParts[2].Replace(",",
                "").Trim();
263                 instruction.OperandBRegister = operationParts[3].Replace(",",
                "").Trim();
264                 break;
```

```
265                 case BVOperation.GOTOEQCONST:
266                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
267                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
268                     instruction.OperandImmediate = int.Parse(operationParts[3]);
269                     break;
270                 case BVOperation.GOTONOEQ:
271                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
272                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
273                     instruction.OperandBRegister = operationParts[3].Replace(",",
                        "").Trim();
274                     break;
275                 case BVOperation.GOTONOEQCONST:
276                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
277                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
278                     instruction.OperandImmediate = int.Parse(operationParts[3]);
279                     break;
280                 case BVOperation.GOTOMORETHEN:
281                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
282                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
283                     instruction.OperandBRegister = operationParts[3].Replace(",",
                        "").Trim();
284                     break;
285                 case BVOperation.GOTOMORETHENCONST:
286                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
287                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
288                     instruction.OperandImmediate = int.Parse(operationParts[3]);
289                     break;
290                 case BVOperation.GOTOLESSTHEN:
291                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
292                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
293                     instruction.OperandBRegister = operationParts[3].Replace(",",
                        "").Trim();
294                     break;
295                 case BVOperation.GOTOLESSTHENCONST:
296                     instruction.DestinationRegister = operationParts[1].Replace
                        (",", "").Trim();
297                     instruction.OperandARegister = operationParts[2].Replace(",",
                        "").Trim();
298                     instruction.OperandImmediate = int.Parse(operationParts[3]);
299                     break;
300                 case BVOperation.JUMPLABEL:
301                     instruction.JumpLabel = operationParts[0].Substring(0,
                        operationParts[0].Length -1);
302                     break;
303                 }
304 
305                 return instruction;
306             }
307         }
308     }
```

ProjectItem 'BVOperationValidationChecks.cs' has no task items

```
1   using BajanVincyAssembly.Models;
2   using BajanVincyAssembly.Models.ComputerArchitecture;
3   using BajanVincyAssembly.Models.Validation;
4   using BajanVincyAssembly.Services.Registers;
5   using System;
6   using System.Collections.Generic;
7   using System.Linq;
8   using System.Text;
9   using System.Text.RegularExpressions;
10  using System.Threading.Tasks;
11
12  namespace BajanVincyAssembly.Services.Compilers
13  {
14      /// <summary>
15      /// Runs BV Operation Validation Checks
16      /// </summary>
17      public class BVOperationValidationChecks
18      {
19          /// <summary>
20          /// Instantiates a new instance of the <see
            cref="BVOperationValidationChecks" class/>
21          /// </summary>
22          /// <param name="linesOfCode">Lines of Code</param>
23          public BVOperationValidationChecks(IEnumerable<string> linesOfCode)
24          {
25              this._LinesOfCode = linesOfCode.DeepClone();
26
27              this.ValidateLinesOfCode(this._LinesOfCode);
28          }
29
30          /// <summary>
31          /// Lines of Code to validate
32          /// </summary>
33          private readonly IEnumerable<string> _LinesOfCode;
34
35          /// <summary>
36          /// Register Cache
37          /// </summary>
38          private readonly IRegistry<Register> _Registry = new Registry();
39
40          /// <summary>
41          /// Jump Label Regex
42          /// </summary>
43          public static readonly Regex JumpLabelregex = new Regex(@"[a-zA-Z0-9]+:"
44
45          /// <summary>
46          /// Letters Only Regex
47          /// </summary>
48          public static readonly Regex LettersOnlyregex = new Regex(@"[a-zA-Z0-9]+"
49
50          /// <summary>
51          /// Validation Info for all lines of code
52          /// </summary>
53          public ValidationInfo ValidationInfo { get; private set; } = new
            ValidationInfo();
54
55          /// <summary>
56          /// Validates Lines of Code and Updates Validation Info
57          /// </summary>
58          /// <param name="linesOfCode">Lines of Code to Validate</param>
59          private void ValidateLinesOfCode(IEnumerable<string> linesOfCode)
60          {
1   2   3
```

```
61              if (!linesOfCode.Any())
62              {
63                  return;
64              }
65
66              foreach (string lineOfCode in linesOfCode)
67              {
68                  string[] operationPartsSplitter = { " " };
69                  var operationParts = lineOfCode.Split(operationPartsSplitter,
                    StringSplitOptions.RemoveEmptyEntries);
70
71                  if (!operationParts.Any())
72                  {
73                      this.UpdateValidationInfo(false, new List<string>() {
                        $"Invalid Instruction Found (No Operation Parts Found): -> $
                        {lineOfCode}" });
74                      continue;
75                  }
76
77                  string operationFound = operationParts[0].Trim().ToLower();
78                  bool jumpLabelFound = JumpLabelregex.Match
                    (operationFound).Success;
79
80                  if (!BVOperationInfo.BVOperationLookup.ContainsKey
                    (operationFound) && !jumpLabelFound)
81                  {
82                      this.UpdateValidationInfo(false, new List<string>() {
                        $"Invalid Operation Found: -> {lineOfCode}" });
83                      continue;
84                  }
85
86                  BVOperation operation = jumpLabelFound ? BVOperation.JUMPLABEL :
                    BVOperationInfo.BVOperationLookup[operationFound];
87
88                  ValidationInfo lineOfCode_ValidationInfo = new ValidationInfo();
89
90                  switch (operation)
91                  {
92                  case BVOperation.ADDNS:
93                          lineOfCode_ValidationInfo =
                        this.RunADDNSInstructionValidationCheck(lineOfCode);
94                          break;
95                  case BVOperation.ADDCONST:
96                          lineOfCode_ValidationInfo =
                        this.RunADDConstInstructionValidationCheck(lineOfCode);
97                          break;
98                  case BVOperation.ADDPOS:
99                          lineOfCode_ValidationInfo =
                        this.RunADDPOSInstructionValidationCheck(lineOfCode);
100                         break;
101                 case BVOperation.SUBNS:
102                         lineOfCode_ValidationInfo =
                        this.RunSUBNSInstructionValidationCheck(lineOfCode);
103                         break;
104                 case BVOperation.SUBCONST:
105                         lineOfCode_ValidationInfo =
                        this.RunSubConstInstructionValidationCheck(lineOfCode);
106                         break;
107                 case BVOperation.SUBPOS:
108                         lineOfCode_ValidationInfo =
                        this.RunSUBPOSInstructionValidationCheck(lineOfCode);
109                         break;
110                 case BVOperation.LOGICAND:
```

```
111                 lineOfCode_ValidationInfo =
                this.RunLogicAndInstructionValidationCheck(lineOfCode);
112                 break;
113             case BVOperation.LOGICANDCOSNT:
114                 lineOfCode_ValidationInfo =
                this.RunLogicAndConstInstructionValidationCheck(lineOfCode);
115                 break;
116             case BVOperation.LOGICOR:
117                 lineOfCode_ValidationInfo =
                this.RunLogicOrInstructionValidationCheck(lineOfCode);
118                 break;
119             case BVOperation.LOGICORCONST:
120                 lineOfCode_ValidationInfo =
                this.RunLogicOrConstInstructionValidationCheck(lineOfCode);
121                 break;
122             case BVOperation.SHIFTLEFT:
123                 lineOfCode_ValidationInfo =
                this.RunShiftLeftInstructionValidationCheck(lineOfCode);
124                 break;
125             case BVOperation.SHIFTLEFTPOS:
126                 lineOfCode_ValidationInfo =
                this.RunShiftLeftPosInstructionValidationCheck(lineOfCode);
127                 break;
128             case BVOperation.SHIFTLEFTCONST:
129                 lineOfCode_ValidationInfo =
                this.RunShiftLeftConstInstructionValidationCheck
                (lineOfCode);
130                 break;
131             case BVOperation.SHIFTRIGHT:
132                 lineOfCode_ValidationInfo =
                this.RunShiftRightInstructionValidationCheck(lineOfCode);
133                 break;
134             case BVOperation.SHIFTRIGHTPOS:
135                 lineOfCode_ValidationInfo =
                this.RunShiftRightPosInstructionValidationCheck(lineOfCode);
136                 break;
137             case BVOperation.SHIFTRIGHTCONST:
138                 lineOfCode_ValidationInfo =
                this.RunShiftRightConstInstructionValidationCheck
                (lineOfCode);
139                 break;
140             case BVOperation.FROMEMEM:
141                 lineOfCode_ValidationInfo =
                this.RunFromMemInstructionValidationCheck(lineOfCode);
142                 break;
143             case BVOperation.FROMMEMCONST:
144                 lineOfCode_ValidationInfo =
                this.RunFromMemConstInstructionValidationCheck(lineOfCode);
145                 break;
146             case BVOperation.FROMCONST:
147                 lineOfCode_ValidationInfo =
                this.RunFromConstInstructionValidationCheck(lineOfCode);
148                 break;
149             case BVOperation.TOMEM:
150                 lineOfCode_ValidationInfo =
                this.RunToMemInstructionValidationCheck(lineOfCode);
151                 break;
152             case BVOperation.TOMEMCONST:
153                 lineOfCode_ValidationInfo =
                this.RunToMemConstInstructionValidationCheck(lineOfCode);
154                 break;
155             case BVOperation.TOCONST:
156                 lineOfCode_ValidationInfo =
                this.RunToConstInstructionValidationCheck(lineOfCode);
```

```
157                 break;
158             case BVOperation.TOCONSTCONST:
159                 lineOfCode_ValidationInfo =
                this.RunToConstConstInstructionValidationCheck(lineOfCode);
160                 break;
161             case BVOperation.COPY:
162                 lineOfCode_ValidationInfo =
                this.RunCopyInstructionValidationCheck(lineOfCode);
163                 break;
164             case BVOperation.COPYERASE:
165                 lineOfCode_ValidationInfo =
                this.RunCopyEraseInstructionValidationCheck(lineOfCode);
166                 break;
167             case BVOperation.LESSTHEN:
168                 lineOfCode_ValidationInfo =
                this.RunLessThenInstructionValidationCheck(lineOfCode);
169                 break;
170             case BVOperation.LESSTHENPOS:
171                 lineOfCode_ValidationInfo =
                this.RunLessThenPosInstructionValidationCheck(lineOfCode);
172                 break;
173             case BVOperation.LESSTHENCONST:
174                 lineOfCode_ValidationInfo =
                this.RunLessThenConstInstructionValidationCheck(lineOfCode);
175                 break;
176             case BVOperation.LESSTHENEQ:
177                 lineOfCode_ValidationInfo =
                this.RunLessThenEqInstructionValidationCheck(lineOfCode);
178                 break;
179             case BVOperation.LESSTHENEQPOS:
180                 lineOfCode_ValidationInfo =
                this.RunLessThenEqPosInstructionValidationCheck(lineOfCode);
181                 break;
182             case BVOperation.LESSTHENEQCONST:
183                 lineOfCode_ValidationInfo =
                this.RunLessThenEqConstInstructionValidationCheck
                (lineOfCode);
184                 break;
185             case BVOperation.MORETHEN:
186                 lineOfCode_ValidationInfo =
                this.RunMoreThenInstructionValidationCheck(lineOfCode);
187                 break;
188             case BVOperation.MORETHENPOS:
189                 lineOfCode_ValidationInfo =
                this.RunMoreThenPosInstructionValidationCheck(lineOfCode);
190                 break;
191             case BVOperation.MORETHENCONST:
192                 lineOfCode_ValidationInfo =
                this.RunMoreThenConstInstructionValidationCheck(lineOfCode);
193                 break;
194             case BVOperation.MORETHENEQ:
195                 lineOfCode_ValidationInfo =
                this.RunMoreThenEqInstructionValidationCheck(lineOfCode);
196                 break;
197             case BVOperation.MORETHENEQPOS:
198                 lineOfCode_ValidationInfo =
                this.RunMoreThenEqPosInstructionValidationCheck(lineOfCode);
199                 break;
200             case BVOperation.MORETHENEQCONST:
201                 lineOfCode_ValidationInfo =
                this.RunMoreThenEqConstInstructionValidationCheck
                (lineOfCode);
202                 break;
203             case BVOperation.XOR:
```

```
204                 lineOfCode_ValidationInfo =
                this.RunXORInstructionValidationCheck(lineOfCode);
205                 break;
206             case BVOperation.XORCONST:
207                 lineOfCode_ValidationInfo =
                this.RunXORConstInstructionValidationCheck(lineOfCode);
208                 break;
209             case BVOperation.SAVEADDRESS:
210                 lineOfCode_ValidationInfo =
                this.RunSaveAddressInstructionValidationCheck(lineOfCode);
211                 break;
212             case BVOperation.GOTO:
213                 lineOfCode_ValidationInfo =
                this.RunGoToInstructionValidationCheck(lineOfCode);
214                 break;
215             case BVOperation.EQ:
216                 lineOfCode_ValidationInfo =
                this.RunEqInstructionValidationCheck(lineOfCode);
217                 break;
218             case BVOperation.EQCONST:
219                 lineOfCode_ValidationInfo =
                this.RunEqConstInstructionValidationCheck(lineOfCode);
220                 break;
221             case BVOperation.GOTOEQ:
222                 lineOfCode_ValidationInfo =
                this.RunGoToEqInstructionValidationCheck(lineOfCode);
223                 break;
224             case BVOperation.GOTOEQCONST:
225                 lineOfCode_ValidationInfo =
                this.RunGoToEqConstInstructionValidationCheck(lineOfCode);
226                 break;
227             case BVOperation.GOTONOEQ:
228                 lineOfCode_ValidationInfo =
                this.RunGoToNoEqInstructionValidationCheck(lineOfCode);
229                 break;
230             case BVOperation.GOTONOEQCONST:
231                 lineOfCode_ValidationInfo =
                this.RunGoToNoEqConstInstructionValidationCheck(lineOfCode);
232                 break;
233             case BVOperation.GOTOMORETHEN:
234                 lineOfCode_ValidationInfo =
                this.RunGoToMoreThenInstructionValidationCheck(lineOfCode);
235                 break;
236             case BVOperation.GOTOMORETHENCONST:
237                 lineOfCode_ValidationInfo =
                this.RunGoToMoreThenConstInstructionValidationCheck
                (lineOfCode);
238                 break;
239             case BVOperation.GOTOLESSTHEN:
240                 lineOfCode_ValidationInfo =
                this.RunGoToLessThenInstructionValidationCheck(lineOfCode);
241                 break;
242             case BVOperation.GOTOLESSTHENCONST:
243                 lineOfCode_ValidationInfo =
                this.RunGoToLessThenConstInstructionValidationCheck
                (lineOfCode);
244                 break;
245             case BVOperation.JUMPLABEL:
246                 lineOfCode_ValidationInfo =
                this.RunGoToJumpLabelInstructionValidationCheck(lineOfCode);
247                 break;
248             }
249
```

```
250                 this.UpdateValidationInfo(lineOfCode_ValidationInfo.IsValid,
                    lineOfCode_ValidationInfo.ValidationMessages);
251             }
252         }
253
254         /// <summary>
255         /// Updates Validation Info
256         /// </summary>
257         /// <param name="isValid">validity to logicall append</param>
258         /// <param name="validationMessage">Validation Message to Add</param>
259         private void UpdateValidationInfo(bool isValid, List<string>
                validationMessage)
260         {
261             this.ValidationInfo.IsValid = this.ValidationInfo.IsValid && isValid;
262             this.ValidationInfo.ValidationMessages.AddRange(validationMessage);
263         }
264
265         /// <summary>
266         /// Runs Instruction Validation Check for AddNS
267         /// </summary>
268         /// <param name="lineOfCode">line of code to validate</param>
269         /// <returns>Validation Info</returns>
270         private ValidationInfo RunADDNSInstructionValidationCheck(string
                lineOfCode)
271         {
272             ValidationInfo validationInfo = new ValidationInfo();
273             const string operationName = "addns";
274
275             string[] operationPartsSplitter = { " " };
276             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
277
278             if (!operationParts.Any())
279             {
280                 validationInfo.IsValid = validationInfo.IsValid && false;
281                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
282             }
283
284             if (operationParts.Length != 4)
285             {
286                 validationInfo.IsValid = validationInfo.IsValid && false;
287                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
288                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
289             }
290
291             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
292             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
293
294             foreach (string operationOperandPart in operationOperandParts)
295             {
296                 if (!this._Registry.Exists(operationOperandPart))
297                 {
298                     validationInfo.IsValid = validationInfo.IsValid && false;
299                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
300                 }
301             }
302
303             return validationInfo;
```

```
304         }
305
306         /// <summary>
307         /// Runs Instruction Validation Check for AddConst
308         /// </summary>
309         /// <param name="lineOfCode">line of code to validate</param>
310         /// <returns>Validation Info</returns>
311         private ValidationInfo RunADDConstInstructionValidationCheck(string
            lineOfCode)
312         {
313             ValidationInfo validationInfo = new ValidationInfo();
314             const string operationName = "addconst";
315
316             string[] operationPartsSplitter = { " " };
317             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
318
319             if (!operationParts.Any())
320             {
321                 validationInfo.IsValid = validationInfo.IsValid && false;
322                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
323             }
324
325             if (operationParts.Length != 4)
326             {
327                 validationInfo.IsValid = validationInfo.IsValid && false;
328                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
329                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
330             }
331
332             int num = 0;
333             if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
334             {
335                 validationInfo.IsValid = validationInfo.IsValid && false;
336                 validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
337                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
338             }
339
340             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
341             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
342
343             foreach (string operationOperandPart in operationOperandParts)
344             {
345                 if (!this._Registry.Exists(operationOperandPart))
346                 {
347                     validationInfo.IsValid = validationInfo.IsValid && false;
348                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
349                 }
350             }
351
352             return validationInfo;
353         }
354
355         /// <summary>
356         /// Runs Instruction Validation Check for AddPos
```

```
357         /// </summary>
358         /// <param name="lineOfCode">line of code to validate</param>
359         /// <returns>Validation Info</returns>
360         private ValidationInfo RunADDPOSInstructionValidationCheck(string
            lineOfCode)
361         {
362             ValidationInfo validationInfo = new ValidationInfo();
363             const string operationName = "addpos";
364
365             string[] operationPartsSplitter = { " " };
366             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
367
368             if (!operationParts.Any())
369             {
370                 validationInfo.IsValid = validationInfo.IsValid && false;
371                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
372             }
373
374             if (operationParts.Length != 4)
375             {
376                 validationInfo.IsValid = validationInfo.IsValid && false;
377                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
378                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
379             }
380
381             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
382             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
383
384             foreach (string operationOperandPart in operationOperandParts)
385             {
386                 if (!this._Registry.Exists(operationOperandPart))
387                 {
388                     validationInfo.IsValid = validationInfo.IsValid && false;
389                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
390                 }
391             }
392
393             return validationInfo;
394         }
395
396         /// <summary>
397         /// Runs Instruction Validation Check for SubNS
398         /// </summary>
399         /// <param name="lineOfCode">line of code to validate</param>
400         /// <returns>Validation Info</returns>
401         private ValidationInfo RunSUBNSInstructionValidationCheck(string
            lineOfCode)
402         {
403             ValidationInfo validationInfo = new ValidationInfo();
404             const string operationName = "subns";
405
406             string[] operationPartsSplitter = { " " };
407             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
408
409             if (!operationParts.Any())
410             {
```

```
411             validationInfo.IsValid = validationInfo.IsValid && false;
412             validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
413         }
414
415         if (operationParts.Length != 4)
416         {
417             validationInfo.IsValid = validationInfo.IsValid && false;
418             validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
419             validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, #3");
420         }
421
422         var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
423         var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
424
425         foreach (string operationOperandPart in operationOperandParts)
426         {
427             if (!this._Registry.Exists(operationOperandPart))
428             {
429                 validationInfo.IsValid = validationInfo.IsValid && false;
430                 validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
431             }
432         }
433
434         return validationInfo;
435     }
436
437     /// <summary>
438     /// Runs Instruction Validation Check for SubConst
439     /// </summary>
440     /// <param name="lineOfCode">line of code to validate</param>
441     /// <returns>Validation Info</returns>
442     private ValidationInfo RunSubConstInstructionValidationCheck(string
            lineOfCode)
443     {
444         ValidationInfo validationInfo = new ValidationInfo();
445         const string operationName = "subconst";
446
447         string[] operationPartsSplitter = { " " };
448         var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
449
450         if (!operationParts.Any())
451         {
452             validationInfo.IsValid = validationInfo.IsValid && false;
453             validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
454         }
455
456         if (operationParts.Length != 4)
457         {
458             validationInfo.IsValid = validationInfo.IsValid && false;
459             validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
460             validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, 5");
461         }
462
463         int num = 0;
```

```
464             if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
465             {
466                 validationInfo.IsValid = validationInfo.IsValid && false;
467                 validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
468                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
469             }
470
471             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
472             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
473
474             foreach (string operationOperandPart in operationOperandParts)
475             {
476                 if (!this._Registry.Exists(operationOperandPart))
477                 {
478                     validationInfo.IsValid = validationInfo.IsValid && false;
479                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
480                 }
481             }
482
483             return validationInfo;
484         }
485
486         /// <summary>
487         /// Runs Instruction Validation Check for SubPos
488         /// </summary>
489         /// <param name="lineOfCode">line of code to validate</param>
490         /// <returns>Validation Info</returns>
491         private ValidationInfo RunSUBPOSInstructionValidationCheck(string
                lineOfCode)
492         {
493             ValidationInfo validationInfo = new ValidationInfo();
494             const string operationName = "subpos";
495
496             string[] operationPartsSplitter = { " " };
497             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
498
499             if (!operationParts.Any())
500             {
501                 validationInfo.IsValid = validationInfo.IsValid && false;
502                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
503             }
504
505             if (operationParts.Length != 4)
506             {
507                 validationInfo.IsValid = validationInfo.IsValid && false;
508                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
509                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
510             }
511
512             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
513             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
514
```

```
515             foreach (string operationOperandPart in operationOperandParts)
516             {
517                 if (!this._Registry.Exists(operationOperandPart))
518                 {
519                     validationInfo.IsValid = validationInfo.IsValid && false;
520                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
521                 }
522             }
523
524             return validationInfo;
525         }
526
527         /// <summary>
528         /// Runs Instruction Validation Check for LogicAnd
529         /// </summary>
530         /// <param name="lineOfCode">line of code to validate</param>
531         /// <returns>Validation Info</returns>
532         private ValidationInfo RunLogicAndInstructionValidationCheck(string
            lineOfCode)
533         {
534             ValidationInfo validationInfo = new ValidationInfo();
535             const string operationName = "logicand";
536
537             string[] operationPartsSplitter = { " " };
538             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
539
540             if (!operationParts.Any())
541             {
542                 validationInfo.IsValid = validationInfo.IsValid && false;
543                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
544             }
545
546             if (operationParts.Length != 4)
547             {
548                 validationInfo.IsValid = validationInfo.IsValid && false;
549                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
550                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
551             }
552
553             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
554             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
555
556             foreach (string operationOperandPart in operationOperandParts)
557             {
558                 if (!this._Registry.Exists(operationOperandPart))
559                 {
560                     validationInfo.IsValid = validationInfo.IsValid && false;
561                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
562                 }
563             }
564
565             return validationInfo;
566         }
567
568         /// <summary>
569         /// Runs Instruction Validation Check for LogicAndConst
```

```
570         /// </summary>
571         /// <param name="lineOfCode">line of code to validate</param>
572         /// <returns>Validation Info</returns>
573         private ValidationInfo RunLogicAndConstInstructionValidationCheck(string
            lineOfCode)
574         {
575             ValidationInfo validationInfo = new ValidationInfo();
576             const string operationName = "logicandconst";
577
578             string[] operationPartsSplitter = { " " };
579             var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
580
581             if (!operationParts.Any())
582             {
583                 validationInfo.IsValid = validationInfo.IsValid && false;
584                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
585             }
586
587             if (operationParts.Length != 4)
588             {
589                 validationInfo.IsValid = validationInfo.IsValid && false;
590                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
591                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, 5");
592             }
593
594             int num = 0;
595             if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
            out num))
596             {
597                 validationInfo.IsValid = validationInfo.IsValid && false;
598                 validationInfo.ValidationMessages.Add($"Non-Number Found ($
                {operationParts[3]}): -> ${lineOfCode}");
599                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, 5");
600             }
601
602             var operationOperandPartsRaw = new List<string> { operationParts[1],
            operationParts[2] };
603             var operationOperandParts = operationOperandPartsRaw.Select(part =>
            part.Replace(",", "").Trim());
604
605             foreach (string operationOperandPart in operationOperandParts)
606             {
607                 if (!this._Registry.Exists(operationOperandPart))
608                 {
609                     validationInfo.IsValid = validationInfo.IsValid && false;
610                     validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
611                 }
612             }
613
614             return validationInfo;
615         }
616
617         /// <summary>
618         /// Runs Instruction Validation Check for LogicOr
619         /// </summary>
620         /// <param name="lineOfCode">line of code to validate</param>
621         /// <returns>Validation Info</returns>
```

```
622         private ValidationInfo RunLogicOrInstructionValidationCheck(string lineOfCode)
623         {
624             ValidationInfo validationInfo = new ValidationInfo();
625             const string operationName = "logicor";
626 
627             string[] operationPartsSplitter = { " " };
628             var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
629 
630             if (!operationParts.Any())
631             {
632                 validationInfo.IsValid = validationInfo.IsValid && false;
633                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
634             }
635 
636             if (operationParts.Length != 4)
637             {
638                 validationInfo.IsValid = validationInfo.IsValid && false;
639                 validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
640                 validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, #3");
641             }
642 
643             var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2], operationParts[3] };
644             var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
645 
646             foreach (string operationOperandPart in operationOperandParts)
647             {
648                 if (!this._Registry.Exists(operationOperandPart))
649                 {
650                     validationInfo.IsValid = validationInfo.IsValid && false;
651                     validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
652                 }
653             }
654 
655             return validationInfo;
656         }
657 
658         /// <summary>
659         /// Runs Instruction Validation Check for LogicOrConst
660         /// </summary>
661         /// <param name="lineOfCode">line of code to validate</param>
662         /// <returns>Validation Info</returns>
663         private ValidationInfo RunLogicOrConstInstructionValidationCheck(string lineOfCode)
664         {
665             ValidationInfo validationInfo = new ValidationInfo();
666             const string operationName = "logicorconst";
667 
668             string[] operationPartsSplitter = { " " };
669             var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
670 
671             if (!operationParts.Any())
672             {
673                 validationInfo.IsValid = validationInfo.IsValid && false;
674                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
```

```
675             }
676
677             if (operationParts.Length != 4)
678             {
679                 validationInfo.IsValid = validationInfo.IsValid && false;
680                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
681                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
682             }
683
684             int num = 0;
685             if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
686             {
687                 validationInfo.IsValid = validationInfo.IsValid && false;
688                 validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
689                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
690             }
691
692             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
693             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
694
695             foreach (string operationOperandPart in operationOperandParts)
696             {
697                 if (!this._Registry.Exists(operationOperandPart))
698                 {
699                     validationInfo.IsValid = validationInfo.IsValid && false;
700                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
701                 }
702             }
703
704             return validationInfo;
705         }
706
707         /// <summary>
708         /// Runs Instruction Validation Check for ShiftLeft
709         /// </summary>
710         /// <param name="lineOfCode">line of code to validate</param>
711         /// <returns>Validation Info</returns>
712         private ValidationInfo RunShiftLeftInstructionValidationCheck(string
                lineOfCode)
713         {
714             ValidationInfo validationInfo = new ValidationInfo();
715             const string operationName = "shiftleft";
716
717             string[] operationPartsSplitter = { " " };
718             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
719
720             if (!operationParts.Any())
721             {
722                 validationInfo.IsValid = validationInfo.IsValid && false;
723                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
724             }
725
726             if (operationParts.Length != 4)
727             {
```

```
728             validationInfo.IsValid = validationInfo.IsValid && false;
729             validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
730             validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, #3");
731         }
732
733         var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
734         var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
735
736         foreach (string operationOperandPart in operationOperandParts)
737         {
738             if (!this._Registry.Exists(operationOperandPart))
739             {
740                 validationInfo.IsValid = validationInfo.IsValid && false;
741                 validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
742             }
743         }
744
745         return validationInfo;
746     }
747
748     /// <summary>
749     /// Runs Instruction Validation Check for ShiftLeftPos
750     /// </summary>
751     /// <param name="lineOfCode">line of code to validate</param>
752     /// <returns>Validation Info</returns>
753     private ValidationInfo RunShiftLeftPosInstructionValidationCheck(string
            lineOfCode)
754     {
755         ValidationInfo validationInfo = new ValidationInfo();
756         const string operationName = "shiftleftpos";
757
758         string[] operationPartsSplitter = { " " };
759         var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
760
761         if (!operationParts.Any())
762         {
763             validationInfo.IsValid = validationInfo.IsValid && false;
764             validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
765         }
766
767         if (operationParts.Length != 4)
768         {
769             validationInfo.IsValid = validationInfo.IsValid && false;
770             validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
771             validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, #3");
772         }
773
774         var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
775         var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
776
777         foreach (string operationOperandPart in operationOperandParts)
778         {
779             if (!this._Registry.Exists(operationOperandPart))
```

```
780                 {
781                     validationInfo.IsValid = validationInfo.IsValid && false;
782                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
783                 }
784             }
785
786             return validationInfo;
787         }
788
789         /// <summary>
790         /// Runs Instruction Validation Check for ShiftLeftConst
791         /// </summary>
792         /// <param name="lineOfCode">line of code to validate</param>
793         /// <returns>Validation Info</returns>
794         private ValidationInfo RunShiftLeftConstInstructionValidationCheck(string
                lineOfCode)
795         {
796             ValidationInfo validationInfo = new ValidationInfo();
797             const string operationName = "shiftleftconst";
798
799             string[] operationPartsSplitter = { " " };
800             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
801
802             if (!operationParts.Any())
803             {
804                 validationInfo.IsValid = validationInfo.IsValid && false;
805                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
806             }
807
808             if (operationParts.Length != 4)
809             {
810                 validationInfo.IsValid = validationInfo.IsValid && false;
811                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
812                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
813             }
814
815             int num = 0;
816             if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
817             {
818                 validationInfo.IsValid = validationInfo.IsValid && false;
819                 validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
820                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
821             }
822
823             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
824             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
825
826             foreach (string operationOperandPart in operationOperandParts)
827             {
828                 if (!this._Registry.Exists(operationOperandPart))
829                 {
830                     validationInfo.IsValid = validationInfo.IsValid && false;
831                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
```

```
832                 }
833             }
834
835             return validationInfo;
836         }
837
838         /// <summary>
839         /// Runs Instruction Validation Check for ShiftRight
840         /// </summary>
841         /// <param name="lineOfCode">line of code to validate</param>
842         /// <returns>Validation Info</returns>
843         private ValidationInfo RunShiftRightInstructionValidationCheck(string
            lineOfCode)
844         {
845             ValidationInfo validationInfo = new ValidationInfo();
846             const string operationName = "shiftright";
847
848             string[] operationPartsSplitter = { " " };
849             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
850
851             if (!operationParts.Any())
852             {
853                 validationInfo.IsValid = validationInfo.IsValid && false;
854                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
855             }
856
857             if (operationParts.Length != 4)
858             {
859                 validationInfo.IsValid = validationInfo.IsValid && false;
860                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
861                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
862             }
863
864             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
865             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
866
867             foreach (string operationOperandPart in operationOperandParts)
868             {
869                 if (!this._Registry.Exists(operationOperandPart))
870                 {
871                     validationInfo.IsValid = validationInfo.IsValid && false;
872                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
873                 }
874             }
875
876             return validationInfo;
877         }
878
879         /// <summary>
880         /// Runs Instruction Validation Check for ShiftRightPos
881         /// </summary>
882         /// <param name="lineOfCode">line of code to validate</param>
883         /// <returns>Validation Info</returns>
884         private ValidationInfo RunShiftRightPosInstructionValidationCheck(string
            lineOfCode)
885         {
886             ValidationInfo validationInfo = new ValidationInfo();
```

```
887             const string operationName = "shiftrightpos";
888
889             string[] operationPartsSplitter = { " " };
890             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
891
892             if (!operationParts.Any())
893             {
894                 validationInfo.IsValid = validationInfo.IsValid && false;
895                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
896             }
897
898             if (operationParts.Length != 4)
899             {
900                 validationInfo.IsValid = validationInfo.IsValid && false;
901                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
902                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
903             }
904
905             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
906             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
907
908             foreach (string operationOperandPart in operationOperandParts)
909             {
910                 if (!this._Registry.Exists(operationOperandPart))
911                 {
912                     validationInfo.IsValid = validationInfo.IsValid && false;
913                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
914                 }
915             }
916
917             return validationInfo;
918         }
919
920         /// <summary>
921         /// Runs Instruction Validation Check for ShiftRightConst
922         /// </summary>
923         /// <param name="lineOfCode">line of code to validate</param>
924         /// <returns>Validation Info</returns>
925         private ValidationInfo RunShiftRightConstInstructionValidationCheck(string
                lineOfCode)
926         {
927             ValidationInfo validationInfo = new ValidationInfo();
928             const string operationName = "shiftrightconst";
929
930             string[] operationPartsSplitter = { " " };
931             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
932
933             if (!operationParts.Any())
934             {
935                 validationInfo.IsValid = validationInfo.IsValid && false;
936                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
937             }
938
939             if (operationParts.Length != 4)
940             {
```

```
941                 validationInfo.IsValid = validationInfo.IsValid && false;
942                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
943                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
944             }
945
946             int num = 0;
947             if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
948             {
949                 validationInfo.IsValid = validationInfo.IsValid && false;
950                 validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
951                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
952             }
953
954             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
955             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
956
957             foreach (string operationOperandPart in operationOperandParts)
958             {
959                 if (!this._Registry.Exists(operationOperandPart))
960                 {
961                     validationInfo.IsValid = validationInfo.IsValid && false;
962                     validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
963                 }
964             }
965
966             return validationInfo;
967         }
968
969         /// <summary>
970         /// Runs Instruction Validation Check for FromMem
971         /// </summary>
972         /// <param name="lineOfCode">line of code to validate</param>
973         /// <returns>Validation Info</returns>
974         private ValidationInfo RunFromMemInstructionValidationCheck(string
                lineOfCode)
975         {
976             ValidationInfo validationInfo = new ValidationInfo();
977             const string operationName = "frommem";
978
979             string[] operationPartsSplitter = { " " };
980             var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
981
982             if (!operationParts.Any())
983             {
984                 validationInfo.IsValid = validationInfo.IsValid && false;
985                 validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
986             }
987
988             if (operationParts.Length != 4)
989             {
990                 validationInfo.IsValid = validationInfo.IsValid && false;
991                 validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
```

```
992                 validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
993             }
994
995             var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
996             var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
997
998             foreach (string operationOperandPart in operationOperandParts)
999             {
1000                if (!this._Registry.Exists(operationOperandPart))
1001                {
1002                    validationInfo.IsValid = validationInfo.IsValid && false;
1003                    validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
1004                }
1005            }
1006
1007            return validationInfo;
1008        }
1009
1010        /// <summary>
1011        /// Runs Instruction Validation Check for FromMemConst
1012        /// </summary>
1013        /// <param name="lineOfCode">line of code to validate</param>
1014        /// <returns>Validation Info</returns>
1015        private ValidationInfo RunFromMemConstInstructionValidationCheck(string
                lineOfCode)
1016        {
1017            ValidationInfo validationInfo = new ValidationInfo();
1018            const string operationName = "frommemconst";
1019
1020            string[] operationPartsSplitter = { " " };
1021            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1022
1023            if (!operationParts.Any())
1024            {
1025                validationInfo.IsValid = validationInfo.IsValid && false;
1026                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1027            }
1028
1029            if (operationParts.Length != 4)
1030            {
1031                validationInfo.IsValid = validationInfo.IsValid && false;
1032                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1033                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1034            }
1035
1036            int num = 0;
1037            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
1038            {
1039                validationInfo.IsValid = validationInfo.IsValid && false;
1040                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
1041                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1042            }
1043
```

```
1044            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2] };
1045            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
1046
1047            foreach (string operationOperandPart in operationOperandParts)
1048            {
1049                if (!this._Registry.Exists(operationOperandPart))
1050                {
1051                    validationInfo.IsValid = validationInfo.IsValid && false;
1052                    validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
1053                }
1054            }
1055
1056            return validationInfo;
1057        }
1058
1059        /// <summary>
1060        /// Runs Instruction Validation Check for FromConst
1061        /// </summary>
1062        /// <param name="lineOfCode">line of code to validate</param>
1063        /// <returns>Validation Info</returns>
1064        private ValidationInfo RunFromConstInstructionValidationCheck(string lineOfCode)
1065        {
1066            ValidationInfo validationInfo = new ValidationInfo();
1067            const string operationName = "fromconst";
1068
1069            string[] operationPartsSplitter = { " " };
1070            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
1071
1072            if (!operationParts.Any())
1073            {
1074                validationInfo.IsValid = validationInfo.IsValid && false;
1075                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
1076            }
1077
1078            if (operationParts.Length != 3)
1079            {
1080                validationInfo.IsValid = validationInfo.IsValid && false;
1081                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
1082                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, 5");
1083            }
1084
1085            int num = 0;
1086            if (operationParts.Length >= 3 && !int.TryParse(operationParts[2], out num))
1087            {
1088                validationInfo.IsValid = validationInfo.IsValid && false;
1089                validationInfo.ValidationMessages.Add($"Non-Number Found (${operationParts[2]}): -> ${lineOfCode}");
1090                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, 5");
1091            }
1092
1093            var operationOperandPartsRaw = new List<string> { operationParts[1] };
1094            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
```

```
1095
1096            foreach (string operationOperandPart in operationOperandParts)
1097            {
1098                if (!this._Registry.Exists(operationOperandPart))
1099                {
1100                    validationInfo.IsValid = validationInfo.IsValid && false;
1101                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1102                }
1103            }
1104
1105            return validationInfo;
1106        }
1107
1108        /// <summary>
1109        /// Runs Instruction Validation Check for ToMem
1110        /// </summary>
1111        /// <param name="lineOfCode">line of code to validate</param>
1112        /// <returns>Validation Info</returns>
1113        private ValidationInfo RunToMemInstructionValidationCheck(string
                lineOfCode)
1114        {
1115            ValidationInfo validationInfo = new ValidationInfo();
1116            const string operationName = "tomem";
1117
1118            string[] operationPartsSplitter = { " " };
1119            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1120
1121            if (!operationParts.Any())
1122            {
1123                validationInfo.IsValid = validationInfo.IsValid && false;
1124                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1125            }
1126
1127            if (operationParts.Length != 4)
1128            {
1129                validationInfo.IsValid = validationInfo.IsValid && false;
1130                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1131                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1132            }
1133
1134            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
1135            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1136
1137            foreach (string operationOperandPart in operationOperandParts)
1138            {
1139                if (!this._Registry.Exists(operationOperandPart))
1140                {
1141                    validationInfo.IsValid = validationInfo.IsValid && false;
1142                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1143                }
1144            }
1145
1146            return validationInfo;
1147        }
1148
1149        /// <summary>
```

```
1150        /// Runs Instruction Validation Check for ToMemConst
1151        /// </summary>
1152        /// <param name="lineOfCode">line of code to validate</param>
1153        /// <returns>Validation Info</returns>
1154        private ValidationInfo RunToMemConstInstructionValidationCheck(string
            lineOfCode)
1155        {
1156            ValidationInfo validationInfo = new ValidationInfo();
1157            const string operationName = "tomemconst";
1158
1159            string[] operationPartsSplitter = { " " };
1160            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1161
1162            if (!operationParts.Any())
1163            {
1164                validationInfo.IsValid = validationInfo.IsValid && false;
1165                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
1166            }
1167
1168            if (operationParts.Length != 4)
1169            {
1170                validationInfo.IsValid = validationInfo.IsValid && false;
1171                validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
1172                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, 5, #2");
1173            }
1174
1175            int num = 0;
1176            if (operationParts.Length >= 3 && !int.TryParse(operationParts[2],
            out num))
1177            {
1178                validationInfo.IsValid = validationInfo.IsValid && false;
1179                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                {operationParts[2]}): -> ${lineOfCode}");
1180                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, 5, #2");
1181            }
1182
1183            var operationOperandPartsRaw = new List<string> { operationParts[1],
            operationParts[3] };
1184            var operationOperandParts = operationOperandPartsRaw.Select(part =>
            part.Replace(",", "").Trim());
1185
1186            foreach (string operationOperandPart in operationOperandParts)
1187            {
1188                if (!this._Registry.Exists(operationOperandPart))
1189                {
1190                    validationInfo.IsValid = validationInfo.IsValid && false;
1191                    validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
1192                }
1193            }
1194
1195            return validationInfo;
1196        }
1197
1198        /// <summary>
1199        /// Runs Instruction Validation Check for ToConst
1200        /// </summary>
1201        /// <param name="lineOfCode">line of code to validate</param>
1202        /// <returns>Validation Info</returns>
```

```
1203        private ValidationInfo RunToConstInstructionValidationCheck(string lineOfCode)
1204        {
1205            ValidationInfo validationInfo = new ValidationInfo();
1206            const string operationName = "toconst";
1207
1208            string[] operationPartsSplitter = { " " };
1209            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
1210
1211            if (!operationParts.Any())
1212            {
1213                validationInfo.IsValid = validationInfo.IsValid && false;
1214                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
1215            }
1216
1217            if (operationParts.Length != 4)
1218            {
1219                validationInfo.IsValid = validationInfo.IsValid && false;
1220                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
1221                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, 5");
1222            }
1223
1224            int num = 0;
1225            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3], out num))
1226            {
1227                validationInfo.IsValid = validationInfo.IsValid && false;
1228                validationInfo.ValidationMessages.Add($"Non-Number Found ({operationParts[3]}): -> ${lineOfCode}");
1229                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, 5");
1230            }
1231
1232            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2] };
1233            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
1234
1235            foreach (string operationOperandPart in operationOperandParts)
1236            {
1237                if (!this._Registry.Exists(operationOperandPart))
1238                {
1239                    validationInfo.IsValid = validationInfo.IsValid && false;
1240                    validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
1241                }
1242            }
1243
1244            return validationInfo;
1245        }
1246
1247        /// <summary>
1248        /// Runs Instruction Validation Check for ToConstConst
1249        /// </summary>
1250        /// <param name="lineOfCode">line of code to validate</param>
1251        /// <returns>Validation Info</returns>
1252        private ValidationInfo RunToConstConstInstructionValidationCheck(string lineOfCode)
1253        {
1254            ValidationInfo validationInfo = new ValidationInfo();
```

```
1255            const string operationName = "toconstconst";
1256
1257            string[] operationPartsSplitter = { " " };
1258            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1259
1260            if (!operationParts.Any())
1261            {
1262                validationInfo.IsValid = validationInfo.IsValid && false;
1263                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1264            }
1265
1266            if (operationParts.Length != 4)
1267            {
1268                validationInfo.IsValid = validationInfo.IsValid && false;
1269                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1270                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, 2, 5");
1271            }
1272
1273            int num = 0;
1274            if (operationParts.Length >= 3 && !int.TryParse(operationParts[2],
                out num))
1275            {
1276                validationInfo.IsValid = validationInfo.IsValid && false;
1277                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[2]}): -> ${lineOfCode}");
1278                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, 2, 5");
1279            }
1280            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
1281            {
1282                validationInfo.IsValid = validationInfo.IsValid && false;
1283                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
1284                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, 2, 5");
1285            }
1286
1287            var operationOperandPartsRaw = new List<string> { operationParts[1]
                };
1288            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1289
1290            foreach (string operationOperandPart in operationOperandParts)
1291            {
1292                if (!this._Registry.Exists(operationOperandPart))
1293                {
1294                    validationInfo.IsValid = validationInfo.IsValid && false;
1295                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1296                }
1297            }
1298
1299            return validationInfo;
1300        }
1301
1302        /// <summary>
1303        /// Runs Instruction Validation Check for copy
1304        /// </summary>
1305        /// <param name="lineOfCode">line of code to validate</param>
```

```
1306        /// <returns>Validation Info</returns>
1307        private ValidationInfo RunCopyInstructionValidationCheck(string
            lineOfCode)
1308        {
1309            ValidationInfo validationInfo = new ValidationInfo();
1310            const string operationName = "copy";
1311
1312            string[] operationPartsSplitter = { " " };
1313            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1314
1315            if (!operationParts.Any())
1316            {
1317                validationInfo.IsValid = validationInfo.IsValid && false;
1318                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
1319            }
1320
1321            if (operationParts.Length != 3)
1322            {
1323                validationInfo.IsValid = validationInfo.IsValid && false;
1324                validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
1325                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2");
1326            }
1327
1328            var operationOperandPartsRaw = new List<string> { operationParts[1],
            operationParts[2] };
1329            var operationOperandParts = operationOperandPartsRaw.Select(part =>
            part.Replace(",", "").Trim());
1330
1331            foreach (string operationOperandPart in operationOperandParts)
1332            {
1333                if (!this._Registry.Exists(operationOperandPart))
1334                {
1335                    validationInfo.IsValid = validationInfo.IsValid && false;
1336                    validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
1337                }
1338            }
1339
1340            return validationInfo;
1341        }
1342
1343        /// <summary>
1344        /// Runs Instruction Validation Check for CopyErase
1345        /// </summary>
1346        /// <param name="lineOfCode">line of code to validate</param>
1347        /// <returns>Validation Info</returns>
1348        private ValidationInfo RunCopyEraseInstructionValidationCheck(string
            lineOfCode)
1349        {
1350            ValidationInfo validationInfo = new ValidationInfo();
1351            const string operationName = "copyerase";
1352
1353            string[] operationPartsSplitter = { " " };
1354            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1355
1356            if (!operationParts.Any())
1357            {
1358                validationInfo.IsValid = validationInfo.IsValid && false;
```

```
1359                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1360            }
1361
1362            if (operationParts.Length != 3)
1363            {
1364                validationInfo.IsValid = validationInfo.IsValid && false;
1365                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1366                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2");
1367            }
1368
1369            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
1370            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1371
1372            foreach (string operationOperandPart in operationOperandParts)
1373            {
1374                if (!this._Registry.Exists(operationOperandPart))
1375                {
1376                    validationInfo.IsValid = validationInfo.IsValid && false;
1377                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1378                }
1379            }
1380
1381            return validationInfo;
1382        }
1383
1384        /// <summary>
1385        /// Runs Instruction Validation Check for Lessthen
1386        /// </summary>
1387        /// <param name="lineOfCode">line of code to validate</param>
1388        /// <returns>Validation Info</returns>
1389        private ValidationInfo RunLessThenInstructionValidationCheck(string
                lineOfCode)
1390        {
1391            ValidationInfo validationInfo = new ValidationInfo();
1392            const string operationName = "lessthen";
1393
1394            string[] operationPartsSplitter = { " " };
1395            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1396
1397            if (!operationParts.Any())
1398            {
1399                validationInfo.IsValid = validationInfo.IsValid && false;
1400                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1401            }
1402
1403            if (operationParts.Length != 4)
1404            {
1405                validationInfo.IsValid = validationInfo.IsValid && false;
1406                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1407                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1408            }
1409
1410            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
```

```
1411            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1412
1413            foreach (string operationOperandPart in operationOperandParts)
1414            {
1415                if (!this._Registry.Exists(operationOperandPart))
1416                {
1417                    validationInfo.IsValid = validationInfo.IsValid && false;
1418                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1419                }
1420            }
1421
1422            return validationInfo;
1423        }
1424
1425        /// <summary>
1426        /// Runs Instruction Validation Check for LessthenPos
1427        /// </summary>
1428        /// <param name="lineOfCode">line of code to validate</param>
1429        /// <returns>Validation Info</returns>
1430        private ValidationInfo RunLessThenPosInstructionValidationCheck(string
                lineOfCode)
1431        {
1432            ValidationInfo validationInfo = new ValidationInfo();
1433            const string operationName = "lessthenpos";
1434
1435            string[] operationPartsSplitter = { " " };
1436            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1437
1438            if (!operationParts.Any())
1439            {
1440                validationInfo.IsValid = validationInfo.IsValid && false;
1441                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1442            }
1443
1444            if (operationParts.Length != 4)
1445            {
1446                validationInfo.IsValid = validationInfo.IsValid && false;
1447                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1448                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1449            }
1450
1451            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
1452            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1453
1454            foreach (string operationOperandPart in operationOperandParts)
1455            {
1456                if (!this._Registry.Exists(operationOperandPart))
1457                {
1458                    validationInfo.IsValid = validationInfo.IsValid && false;
1459                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1460                }
1461            }
1462
1463            return validationInfo;
1464        }
```

```
1465
1466        /// <summary>
1467        /// Runs Instruction Validation Check for LessthenConst
1468        /// </summary>
1469        /// <param name="lineOfCode">line of code to validate</param>
1470        /// <returns>Validation Info</returns>
1471        private ValidationInfo RunLessThenConstInstructionValidationCheck(string
            lineOfCode)
1472        {
1473            ValidationInfo validationInfo = new ValidationInfo();
1474            const string operationName = "lessthenconst";
1475
1476            string[] operationPartsSplitter = { " " };
1477            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1478
1479            if (!operationParts.Any())
1480            {
1481                validationInfo.IsValid = validationInfo.IsValid && false;
1482                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1483            }
1484
1485            if (operationParts.Length != 4)
1486            {
1487                validationInfo.IsValid = validationInfo.IsValid && false;
1488                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1489                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1490            }
1491
1492            int num = 0;
1493            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
1494            {
1495                validationInfo.IsValid = validationInfo.IsValid && false;
1496                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
1497                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1498            }
1499
1500            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
1501            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1502
1503            foreach (string operationOperandPart in operationOperandParts)
1504            {
1505                if (!this._Registry.Exists(operationOperandPart))
1506                {
1507                    validationInfo.IsValid = validationInfo.IsValid && false;
1508                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1509                }
1510            }
1511
1512            return validationInfo;
1513        }
1514
1515        /// <summary>
1516        /// Runs Instruction Validation Check for LessthenEq
1517        /// </summary>
```

```
1518        /// <param name="lineOfCode">line of code to validate</param>
1519        /// <returns>Validation Info</returns>
1520        private ValidationInfo RunLessThenEqInstructionValidationCheck(string
            lineOfCode)
1521        {
1522            ValidationInfo validationInfo = new ValidationInfo();
1523            const string operationName = "lesstheneq";
1524
1525            string[] operationPartsSplitter = { " " };
1526            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1527
1528            if (!operationParts.Any())
1529            {
1530                validationInfo.IsValid = validationInfo.IsValid && false;
1531                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
1532            }
1533
1534            if (operationParts.Length != 4)
1535            {
1536                validationInfo.IsValid = validationInfo.IsValid && false;
1537                validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
1538                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, #3");
1539            }
1540
1541            var operationOperandPartsRaw = new List<string> { operationParts[1],
            operationParts[2], operationParts[3] };
1542            var operationOperandParts = operationOperandPartsRaw.Select(part =>
            part.Replace(",", "").Trim());
1543
1544            foreach (string operationOperandPart in operationOperandParts)
1545            {
1546                if (!this._Registry.Exists(operationOperandPart))
1547                {
1548                    validationInfo.IsValid = validationInfo.IsValid && false;
1549                    validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
1550                }
1551            }
1552
1553            return validationInfo;
1554        }
1555
1556        /// <summary>
1557        /// Runs Instruction Validation Check for LessthenEqPos
1558        /// </summary>
1559        /// <param name="lineOfCode">line of code to validate</param>
1560        /// <returns>Validation Info</returns>
1561        private ValidationInfo RunLessThenEqPosInstructionValidationCheck(string
            lineOfCode)
1562        {
1563            ValidationInfo validationInfo = new ValidationInfo();
1564            const string operationName = "lesstheneqpos";
1565
1566            string[] operationPartsSplitter = { " " };
1567            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1568
1569            if (!operationParts.Any())
1570            {
1571                validationInfo.IsValid = validationInfo.IsValid && false;
```

```
1572                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1573            }
1574
1575            if (operationParts.Length != 4)
1576            {
1577                validationInfo.IsValid = validationInfo.IsValid && false;
1578                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1579                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1580            }
1581
1582            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
1583            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1584
1585            foreach (string operationOperandPart in operationOperandParts)
1586            {
1587                if (!this._Registry.Exists(operationOperandPart))
1588                {
1589                    validationInfo.IsValid = validationInfo.IsValid && false;
1590                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1591                }
1592            }
1593
1594            return validationInfo;
1595        }
1596
1597        /// <summary>
1598        /// Runs Instruction Validation Check for LessthenEqConst
1599        /// </summary>
1600        /// <param name="lineOfCode">line of code to validate</param>
1601        /// <returns>Validation Info</returns>
1602        private ValidationInfo RunLessThenEqConstInstructionValidationCheck(string
                lineOfCode)
1603        {
1604            ValidationInfo validationInfo = new ValidationInfo();
1605            const string operationName = "lesstheneqconst";
1606
1607            string[] operationPartsSplitter = { " " };
1608            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1609
1610            if (!operationParts.Any())
1611            {
1612                validationInfo.IsValid = validationInfo.IsValid && false;
1613                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1614            }
1615
1616            if (operationParts.Length != 4)
1617            {
1618                validationInfo.IsValid = validationInfo.IsValid && false;
1619                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1620                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1621            }
1622
1623            int num = 0;
```

```
1624            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3], out num))
1625            {
1626                validationInfo.IsValid = validationInfo.IsValid && false;
1627                validationInfo.ValidationMessages.Add($"Non-Number Found (${operationParts[3]}): -> ${lineOfCode}");
1628                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, 5");
1629            }
1630
1631            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2] };
1632            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
1633
1634            foreach (string operationOperandPart in operationOperandParts)
1635            {
1636                if (!this._Registry.Exists(operationOperandPart))
1637                {
1638                    validationInfo.IsValid = validationInfo.IsValid && false;
1639                    validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
1640                }
1641            }
1642
1643            return validationInfo;
1644        }
1645
1646        /// <summary>
1647        /// Runs Instruction Validation Check for Morethen
1648        /// </summary>
1649        /// <param name="lineOfCode">line of code to validate</param>
1650        /// <returns>Validation Info</returns>
1651        private ValidationInfo RunMoreThenInstructionValidationCheck(string lineOfCode)
1652        {
1653            ValidationInfo validationInfo = new ValidationInfo();
1654            const string operationName = "morethen";
1655
1656            string[] operationPartsSplitter = { " " };
1657            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
1658
1659            if (!operationParts.Any())
1660            {
1661                validationInfo.IsValid = validationInfo.IsValid && false;
1662                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
1663            }
1664
1665            if (operationParts.Length != 4)
1666            {
1667                validationInfo.IsValid = validationInfo.IsValid && false;
1668                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
1669                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, #3");
1670            }
1671
1672            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2], operationParts[3] };
1673            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
1674
```

```
1675            foreach (string operationOperandPart in operationOperandParts)
1676            {
1677                if (!this._Registry.Exists(operationOperandPart))
1678                {
1679                    validationInfo.IsValid = validationInfo.IsValid && false;
1680                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1681                }
1682            }
1683
1684            return validationInfo;
1685        }
1686
1687        /// <summary>
1688        /// Runs Instruction Validation Check for MoreThenPos
1689        /// </summary>
1690        /// <param name="lineOfCode">line of code to validate</param>
1691        /// <returns>Validation Info</returns>
1692        private ValidationInfo RunMoreThenPosInstructionValidationCheck(string
        lineOfCode)
1693        {
1694            ValidationInfo validationInfo = new ValidationInfo();
1695            const string operationName = "morethenpos";
1696
1697            string[] operationPartsSplitter = { " " };
1698            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1699
1700            if (!operationParts.Any())
1701            {
1702                validationInfo.IsValid = validationInfo.IsValid && false;
1703                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1704            }
1705
1706            if (operationParts.Length != 4)
1707            {
1708                validationInfo.IsValid = validationInfo.IsValid && false;
1709                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1710                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1711            }
1712
1713            var operationOperandPartsRaw = new List<string> { operationParts[1],
            operationParts[2], operationParts[3] };
1714            var operationOperandParts = operationOperandPartsRaw.Select(part =>
            part.Replace(",", "").Trim());
1715
1716            foreach (string operationOperandPart in operationOperandParts)
1717            {
1718                if (!this._Registry.Exists(operationOperandPart))
1719                {
1720                    validationInfo.IsValid = validationInfo.IsValid && false;
1721                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1722                }
1723            }
1724
1725            return validationInfo;
1726        }
1727
1728        /// <summary>
1729        /// Runs Instruction Validation Check for MoreThenConst
```

```
1730        /// </summary>
1731        /// <param name="lineOfCode">line of code to validate</param>
1732        /// <returns>Validation Info</returns>
1733        private ValidationInfo RunMoreThenConstInstructionValidationCheck(string
            lineOfCode)
1734        {
1735            ValidationInfo validationInfo = new ValidationInfo();
1736            const string operationName = "morethenconst";
1737
1738            string[] operationPartsSplitter = { " " };
1739            var operationParts = lineOfCode.Split(operationPartsSplitter,
            StringSplitOptions.RemoveEmptyEntries);
1740
1741            if (!operationParts.Any())
1742            {
1743                validationInfo.IsValid = validationInfo.IsValid && false;
1744                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
1745            }
1746
1747            if (operationParts.Length != 4)
1748            {
1749                validationInfo.IsValid = validationInfo.IsValid && false;
1750                validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
1751                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, 5");
1752            }
1753
1754            int num = 0;
1755            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
            out num))
1756            {
1757                validationInfo.IsValid = validationInfo.IsValid && false;
1758                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                {operationParts[3]}): -> ${lineOfCode}");
1759                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, 5");
1760            }
1761
1762            var operationOperandPartsRaw = new List<string> { operationParts[1],
            operationParts[2] };
1763            var operationOperandParts = operationOperandPartsRaw.Select(part =>
            part.Replace(",", "").Trim());
1764
1765            foreach (string operationOperandPart in operationOperandParts)
1766            {
1767                if (!this._Registry.Exists(operationOperandPart))
1768                {
1769                    validationInfo.IsValid = validationInfo.IsValid && false;
1770                    validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
1771                }
1772            }
1773
1774            return validationInfo;
1775        }
1776
1777        /// <summary>
1778        /// Runs Instruction Validation Check for MoreThenEq
1779        /// </summary>
1780        /// <param name="lineOfCode">line of code to validate</param>
1781        /// <returns>Validation Info</returns>
```

```
1782        private ValidationInfo RunMoreThenEqInstructionValidationCheck(string lineOfCode)
1783        {
1784            ValidationInfo validationInfo = new ValidationInfo();
1785            const string operationName = "moretheneq";
1786
1787            string[] operationPartsSplitter = { " " };
1788            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
1789
1790            if (!operationParts.Any())
1791            {
1792                validationInfo.IsValid = validationInfo.IsValid && false;
1793                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
1794            }
1795
1796            if (operationParts.Length != 4)
1797            {
1798                validationInfo.IsValid = validationInfo.IsValid && false;
1799                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
1800                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, #3");
1801            }
1802
1803            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2], operationParts[3] };
1804            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
1805
1806            foreach (string operationOperandPart in operationOperandParts)
1807            {
1808                if (!this._Registry.Exists(operationOperandPart))
1809                {
1810                    validationInfo.IsValid = validationInfo.IsValid && false;
1811                    validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
1812                }
1813            }
1814
1815            return validationInfo;
1816        }
1817
1818        /// <summary>
1819        /// Runs Instruction Validation Check for MoreThenEqPos
1820        /// </summary>
1821        /// <param name="lineOfCode">line of code to validate</param>
1822        /// <returns>Validation Info</returns>
1823        private ValidationInfo RunMoreThenEqPosInstructionValidationCheck(string lineOfCode)
1824        {
1825            ValidationInfo validationInfo = new ValidationInfo();
1826            const string operationName = "moretheneqpos";
1827
1828            string[] operationPartsSplitter = { " " };
1829            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
1830
1831            if (!operationParts.Any())
1832            {
1833                validationInfo.IsValid = validationInfo.IsValid && false;
1834                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
```

```
1835            }
1836
1837            if (operationParts.Length != 4)
1838            {
1839                validationInfo.IsValid = validationInfo.IsValid && false;
1840                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1841                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1842            }
1843
1844            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
1845            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1846
1847            foreach (string operationOperandPart in operationOperandParts)
1848            {
1849                if (!this._Registry.Exists(operationOperandPart))
1850                {
1851                    validationInfo.IsValid = validationInfo.IsValid && false;
1852                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1853                }
1854            }
1855
1856            return validationInfo;
1857        }
1858
1859        /// <summary>
1860        /// Runs Instruction Validation Check for MoreThenEqConst
1861        /// </summary>
1862        /// <param name="lineOfCode">line of code to validate</param>
1863        /// <returns>Validation Info</returns>
1864        private ValidationInfo RunMoreThenEqConstInstructionValidationCheck(string
                lineOfCode)
1865        {
1866            ValidationInfo validationInfo = new ValidationInfo();
1867            const string operationName = "moretheneqconst";
1868
1869            string[] operationPartsSplitter = { " " };
1870            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1871
1872            if (!operationParts.Any())
1873            {
1874                validationInfo.IsValid = validationInfo.IsValid && false;
1875                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1876            }
1877
1878            if (operationParts.Length != 4)
1879            {
1880                validationInfo.IsValid = validationInfo.IsValid && false;
1881                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1882                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1883            }
1884
1885            int num = 0;
1886            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
1887            {
```

```
1888                validationInfo.IsValid = validationInfo.IsValid && false;
1889                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
1890                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1891            }
1892
1893            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
1894            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1895
1896            foreach (string operationOperandPart in operationOperandParts)
1897            {
1898                if (!this._Registry.Exists(operationOperandPart))
1899                {
1900                    validationInfo.IsValid = validationInfo.IsValid && false;
1901                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1902                }
1903            }
1904
1905            return validationInfo;
1906        }
1907
1908        /// <summary>
1909        /// Runs Instruction Validation Check for XOR
1910        /// </summary>
1911        /// <param name="lineOfCode">line of code to validate</param>
1912        /// <returns>Validation Info</returns>
1913        private ValidationInfo RunXORInstructionValidationCheck(string lineOfCode)
1914        {
1915            ValidationInfo validationInfo = new ValidationInfo();
1916            const string operationName = "xor";
1917
1918            string[] operationPartsSplitter = { " " };
1919            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1920
1921            if (!operationParts.Any())
1922            {
1923                validationInfo.IsValid = validationInfo.IsValid && false;
1924                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1925            }
1926
1927            if (operationParts.Length != 4)
1928            {
1929                validationInfo.IsValid = validationInfo.IsValid && false;
1930                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1931                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
1932            }
1933
1934            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
1935            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1936
1937            foreach (string operationOperandPart in operationOperandParts)
1938            {
1939                if (!this._Registry.Exists(operationOperandPart))
1940                {
```

```
1941                    validationInfo.IsValid = validationInfo.IsValid && false;
1942                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1943                }
1944            }
1945
1946            return validationInfo;
1947        }
1948
1949        /// <summary>
1950        /// Runs Instruction Validation Check for XORConst
1951        /// </summary>
1952        /// <param name="lineOfCode">line of code to validate</param>
1953        /// <returns>Validation Info</returns>
1954        private ValidationInfo RunXORConstInstructionValidationCheck(string
                lineOfCode)
1955        {
1956            ValidationInfo validationInfo = new ValidationInfo();
1957            const string operationName = "xorconst";
1958
1959            string[] operationPartsSplitter = { " " };
1960            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
1961
1962            if (!operationParts.Any())
1963            {
1964                validationInfo.IsValid = validationInfo.IsValid && false;
1965                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
1966            }
1967
1968            if (operationParts.Length != 4)
1969            {
1970                validationInfo.IsValid = validationInfo.IsValid && false;
1971                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
1972                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1973            }
1974
1975            int num = 0;
1976            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
1977            {
1978                validationInfo.IsValid = validationInfo.IsValid && false;
1979                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
1980                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
1981            }
1982
1983            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
1984            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
1985
1986            foreach (string operationOperandPart in operationOperandParts)
1987            {
1988                if (!this._Registry.Exists(operationOperandPart))
1989                {
1990                    validationInfo.IsValid = validationInfo.IsValid && false;
1991                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
1992                }
```

```
1993            }
1994
1995            return validationInfo;
1996        }
1997
1998        /// <summary>
1999        /// Runs Instruction Validation Check for SaveAddress
2000        /// </summary>
2001        /// <param name="lineOfCode">line of code to validate</param>
2002        /// <returns>Validation Info</returns>
2003        private ValidationInfo RunSaveAddressInstructionValidationCheck(string lineOfCode)
2004        {
2005            ValidationInfo validationInfo = new ValidationInfo();
2006            const string operationName = "saveaddress";
2007
2008            string[] operationPartsSplitter = { " " };
2009            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
2010
2011            if (!operationParts.Any())
2012            {
2013                validationInfo.IsValid = validationInfo.IsValid && false;
2014                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
2015            }
2016
2017            if (operationParts.Length != 3)
2018            {
2019                validationInfo.IsValid = validationInfo.IsValid && false;
2020                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
2021                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, labelName");
2022            }
2023
2024            int num = 0;
2025            if (operationParts.Length >= 3 && int.TryParse(operationParts[2], out num))
2026            {
2027                validationInfo.IsValid = validationInfo.IsValid && false;
2028                validationInfo.ValidationMessages.Add($"Number Found ({operationParts[2]}): -> ${lineOfCode}");
2029                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, labelName");
2030            }
2031
2032            var jumpLabelRegexValidation = LettersOnlyregex.Match(operationParts[0]);
2033            if (!jumpLabelRegexValidation.Success)
2034            {
2035                validationInfo.IsValid = validationInfo.IsValid && false;
2036                validationInfo.ValidationMessages.Add($"Invalid Jump Label Format Found: -> ${lineOfCode}");
2037                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, labelName");
2038            }
2039
2040            var operationOperandPartsRaw = new List<string> { operationParts[1] };
2041            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
2042
2043            foreach (string operationOperandPart in operationOperandParts)
```

```
2044            {
2045                if (!this._Registry.Exists(operationOperandPart))
2046                {
2047                    validationInfo.IsValid = validationInfo.IsValid && false;
2048                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2049                }
2050            }
2051
2052            return validationInfo;
2053        }
2054
2055        /// <summary>
2056        /// Runs Instruction Validation Check for GoTo
2057        /// </summary>
2058        /// <param name="lineOfCode">line of code to validate</param>
2059        /// <returns>Validation Info</returns>
2060        private ValidationInfo RunGoToInstructionValidationCheck(string
                lineOfCode)
2061        {
2062            ValidationInfo validationInfo = new ValidationInfo();
2063            const string operationName = "goto";
2064
2065            string[] operationPartsSplitter = { " " };
2066            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2067
2068            if (!operationParts.Any())
2069            {
2070                validationInfo.IsValid = validationInfo.IsValid && false;
2071                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2072            }
2073
2074            if (operationParts.Length != 2)
2075            {
2076                validationInfo.IsValid = validationInfo.IsValid && false;
2077                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2078                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1");
2079            }
2080
2081            int num = 0;
2082            if (operationParts.Length >= 2 && !int.TryParse(operationParts[1],
                out num))
2083            {
2084                validationInfo.IsValid = validationInfo.IsValid && false;
2085                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[1]}): -> ${lineOfCode}");
2086                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1");
2087            }
2088
2089            var operationOperandPartsRaw = new List<string> { operationParts[1]
                };
2090            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2091
2092            foreach (string operationOperandPart in operationOperandParts)
2093            {
2094                if (!this._Registry.Exists(operationOperandPart))
2095                {
2096                    validationInfo.IsValid = validationInfo.IsValid && false;
```

```
2097                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2098                }
2099            }
2100
2101            return validationInfo;
2102        }
2103
2104        /// <summary>
2105        /// Runs Instruction Validation Check for Eq
2106        /// </summary>
2107        /// <param name="lineOfCode">line of code to validate</param>
2108        /// <returns>Validation Info</returns>
2109        private ValidationInfo RunEqInstructionValidationCheck(string lineOfCode)
2110        {
2111            ValidationInfo validationInfo = new ValidationInfo();
2112            const string operationName = "eq";
2113
2114            string[] operationPartsSplitter = { " " };
2115            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2116
2117            if (!operationParts.Any())
2118            {
2119                validationInfo.IsValid = validationInfo.IsValid && false;
2120                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2121            }
2122
2123            if (operationParts.Length != 4)
2124            {
2125                validationInfo.IsValid = validationInfo.IsValid && false;
2126                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2127                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
2128            }
2129
2130            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
2131            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2132
2133            foreach (string operationOperandPart in operationOperandParts)
2134            {
2135                if (!this._Registry.Exists(operationOperandPart))
2136                {
2137                    validationInfo.IsValid = validationInfo.IsValid && false;
2138                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2139                }
2140            }
2141
2142            return validationInfo;
2143        }
2144
2145        /// <summary>
2146        /// Runs Instruction Validation Check for EqConst
2147        /// </summary>
2148        /// <param name="lineOfCode">line of code to validate</param>
2149        /// <returns>Validation Info</returns>
2150        private ValidationInfo RunEqConstInstructionValidationCheck(string
            lineOfCode)
2151        {
```

```
2152            ValidationInfo validationInfo = new ValidationInfo();
2153            const string operationName = "eqconst";
2154
2155            string[] operationPartsSplitter = { " " };
2156            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2157
2158            if (!operationParts.Any())
2159            {
2160                validationInfo.IsValid = validationInfo.IsValid && false;
2161                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2162            }
2163
2164            if (operationParts.Length != 4)
2165            {
2166                validationInfo.IsValid = validationInfo.IsValid && false;
2167                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2168                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2169            }
2170
2171            int num = 0;
2172            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
2173            {
2174                validationInfo.IsValid = validationInfo.IsValid && false;
2175                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
2176                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2177            }
2178
2179            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
2180            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2181
2182            foreach (string operationOperandPart in operationOperandParts)
2183            {
2184                if (!this._Registry.Exists(operationOperandPart))
2185                {
2186                    validationInfo.IsValid = validationInfo.IsValid && false;
2187                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2188                }
2189            }
2190
2191            return validationInfo;
2192        }
2193
2194        /// <summary>
2195        /// Runs Instruction Validation Check for GoToEq
2196        /// </summary>
2197        /// <param name="lineOfCode">line of code to validate</param>
2198        /// <returns>Validation Info</returns>
2199        private ValidationInfo RunGoToEqInstructionValidationCheck(string
                lineOfCode)
2200        {
2201            ValidationInfo validationInfo = new ValidationInfo();
2202            const string operationName = "gotoeq";
2203
2204            string[] operationPartsSplitter = { " " };
```

```
2205            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2206
2207            if (!operationParts.Any())
2208            {
2209                validationInfo.IsValid = validationInfo.IsValid && false;
2210                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
2211            }
2212
2213            if (operationParts.Length != 4)
2214            {
2215                validationInfo.IsValid = validationInfo.IsValid && false;
2216                validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
2217                validationInfo.ValidationMessages.Add($"Correct Format: ->
                {operationName} #1, #2, #3");
2218            }
2219
2220            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
2221            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2222
2223            foreach (string operationOperandPart in operationOperandParts)
2224            {
2225                if (!this._Registry.Exists(operationOperandPart))
2226                {
2227                    validationInfo.IsValid = validationInfo.IsValid && false;
2228                    validationInfo.ValidationMessages.Add($"Unknown Register
                    Found (${operationOperandPart}): -> ${lineOfCode}");
2229                }
2230            }
2231
2232            return validationInfo;
2233        }
2234
2235        /// <summary>
2236        /// Runs Instruction Validation Check for GoToEqConst
2237        /// </summary>
2238        /// <param name="lineOfCode">line of code to validate</param>
2239        /// <returns>Validation Info</returns>
2240        private ValidationInfo RunGoToEqConstInstructionValidationCheck(string
                lineOfCode)
2241        {
2242            ValidationInfo validationInfo = new ValidationInfo();
2243            const string operationName = "gotoeqconst";
2244
2245            string[] operationPartsSplitter = { " " };
2246            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2247
2248            if (!operationParts.Any())
2249            {
2250                validationInfo.IsValid = validationInfo.IsValid && false;
2251                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                (No Operation Parts Found): -> ${lineOfCode}");
2252            }
2253
2254            if (operationParts.Length != 4)
2255            {
2256                validationInfo.IsValid = validationInfo.IsValid && false;
2257                validationInfo.ValidationMessages.Add($"Invalid Instruction
                Format Found: -> ${lineOfCode}");
```

```
2258                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2259            }
2260
2261            int num = 0;
2262            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
2263            {
2264                validationInfo.IsValid = validationInfo.IsValid && false;
2265                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
2266                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2267            }
2268
2269            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
2270            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2271
2272            foreach (string operationOperandPart in operationOperandParts)
2273            {
2274                if (!this._Registry.Exists(operationOperandPart))
2275                {
2276                    validationInfo.IsValid = validationInfo.IsValid && false;
2277                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2278                }
2279            }
2280
2281            return validationInfo;
2282        }
2283
2284        /// <summary>
2285        /// Runs Instruction Validation Check for GoToNoEq
2286        /// </summary>
2287        /// <param name="lineOfCode">line of code to validate</param>
2288        /// <returns>Validation Info</returns>
2289        private ValidationInfo RunGoToNoEqInstructionValidationCheck(string
                lineOfCode)
2290        {
2291            ValidationInfo validationInfo = new ValidationInfo();
2292            const string operationName = "gotonoeq";
2293
2294            string[] operationPartsSplitter = { " " };
2295            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2296
2297            if (!operationParts.Any())
2298            {
2299                validationInfo.IsValid = validationInfo.IsValid && false;
2300                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2301            }
2302
2303            if (operationParts.Length != 4)
2304            {
2305                validationInfo.IsValid = validationInfo.IsValid && false;
2306                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2307                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
2308            }
2309
```

```
2310            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2], operationParts[3] };
2311            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
2312
2313            foreach (string operationOperandPart in operationOperandParts)
2314            {
2315                if (!this._Registry.Exists(operationOperandPart))
2316                {
2317                    validationInfo.IsValid = validationInfo.IsValid && false;
2318                    validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
2319                }
2320            }
2321
2322            return validationInfo;
2323        }
2324
2325        /// <summary>
2326        /// Runs Instruction Validation Check for GoToNoEqConst
2327        /// </summary>
2328        /// <param name="lineOfCode">line of code to validate</param>
2329        /// <returns>Validation Info</returns>
2330        private ValidationInfo RunGoToNoEqConstInstructionValidationCheck(string lineOfCode)
2331        {
2332            ValidationInfo validationInfo = new ValidationInfo();
2333            const string operationName = "gotonoeqconst";
2334
2335            string[] operationPartsSplitter = { " " };
2336            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
2337
2338            if (!operationParts.Any())
2339            {
2340                validationInfo.IsValid = validationInfo.IsValid && false;
2341                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
2342            }
2343
2344            if (operationParts.Length != 4)
2345            {
2346                validationInfo.IsValid = validationInfo.IsValid && false;
2347                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
2348                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, 5");
2349            }
2350
2351            int num = 0;
2352            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3], out num))
2353            {
2354                validationInfo.IsValid = validationInfo.IsValid && false;
2355                validationInfo.ValidationMessages.Add($"Non-Number Found (${operationParts[3]}): -> ${lineOfCode}");
2356                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, 5");
2357            }
2358
2359            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2] };
2360            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
```

```
2361
2362            foreach (string operationOperandPart in operationOperandParts)
2363            {
2364                if (!this._Registry.Exists(operationOperandPart))
2365                {
2366                    validationInfo.IsValid = validationInfo.IsValid && false;
2367                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2368                }
2369            }
2370
2371            return validationInfo;
2372        }
2373
2374        /// <summary>
2375        /// Runs Instruction Validation Check for GoToMorethen
2376        /// </summary>
2377        /// <param name="lineOfCode">line of code to validate</param>
2378        /// <returns>Validation Info</returns>
2379        private ValidationInfo RunGoToMoreThenInstructionValidationCheck(string
                lineOfCode)
2380        {
2381            ValidationInfo validationInfo = new ValidationInfo();
2382            const string operationName = "gotomorethen";
2383
2384            string[] operationPartsSplitter = { " " };
2385            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2386
2387            if (!operationParts.Any())
2388            {
2389                validationInfo.IsValid = validationInfo.IsValid && false;
2390                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2391            }
2392
2393            if (operationParts.Length != 4)
2394            {
2395                validationInfo.IsValid = validationInfo.IsValid && false;
2396                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2397                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, #3");
2398            }
2399
2400            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2], operationParts[3] };
2401            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2402
2403            foreach (string operationOperandPart in operationOperandParts)
2404            {
2405                if (!this._Registry.Exists(operationOperandPart))
2406                {
2407                    validationInfo.IsValid = validationInfo.IsValid && false;
2408                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2409                }
2410            }
2411
2412            return validationInfo;
2413        }
2414
2415        /// <summary>
```

```
2416        /// Runs Instruction Validation Check for GoToMoreThenConst
2417        /// </summary>
2418        /// <param name="lineOfCode">line of code to validate</param>
2419        /// <returns>Validation Info</returns>
2420        private ValidationInfo RunGoToMoreThenConstInstructionValidationCheck
            (string lineOfCode)
2421        {
2422            ValidationInfo validationInfo = new ValidationInfo();
2423            const string operationName = "gotomorethenconst";
2424
2425            string[] operationPartsSplitter = { " " };
2426            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2427
2428            if (!operationParts.Any())
2429            {
2430                validationInfo.IsValid = validationInfo.IsValid && false;
2431                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2432            }
2433
2434            if (operationParts.Length != 4)
2435            {
2436                validationInfo.IsValid = validationInfo.IsValid && false;
2437                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2438                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2439            }
2440
2441            int num = 0;
2442            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
2443            {
2444                validationInfo.IsValid = validationInfo.IsValid && false;
2445                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
2446                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2447            }
2448
2449            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
2450            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2451
2452            foreach (string operationOperandPart in operationOperandParts)
2453            {
2454                if (!this._Registry.Exists(operationOperandPart))
2455                {
2456                    validationInfo.IsValid = validationInfo.IsValid && false;
2457                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2458                }
2459            }
2460
2461            return validationInfo;
2462        }
2463
2464        /// <summary>
2465        /// Runs Instruction Validation Check for GoToLessthen
2466        /// </summary>
2467        /// <param name="lineOfCode">line of code to validate</param>
2468        /// <returns>Validation Info</returns>
```

```
2469        private ValidationInfo RunGoToLessThenInstructionValidationCheck(string lineOfCode)
2470        {
2471            ValidationInfo validationInfo = new ValidationInfo();
2472            const string operationName = "gotolessthen";
2473
2474            string[] operationPartsSplitter = { " " };
2475            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
2476
2477            if (!operationParts.Any())
2478            {
2479                validationInfo.IsValid = validationInfo.IsValid && false;
2480                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
2481            }
2482
2483            if (operationParts.Length != 4)
2484            {
2485                validationInfo.IsValid = validationInfo.IsValid && false;
2486                validationInfo.ValidationMessages.Add($"Invalid Instruction Format Found: -> ${lineOfCode}");
2487                validationInfo.ValidationMessages.Add($"Correct Format: -> {operationName} #1, #2, #3");
2488            }
2489
2490            var operationOperandPartsRaw = new List<string> { operationParts[1], operationParts[2], operationParts[3] };
2491            var operationOperandParts = operationOperandPartsRaw.Select(part => part.Replace(",", "").Trim());
2492
2493            foreach (string operationOperandPart in operationOperandParts)
2494            {
2495                if (!this._Registry.Exists(operationOperandPart))
2496                {
2497                    validationInfo.IsValid = validationInfo.IsValid && false;
2498                    validationInfo.ValidationMessages.Add($"Unknown Register Found (${operationOperandPart}): -> ${lineOfCode}");
2499                }
2500            }
2501
2502            return validationInfo;
2503        }
2504
2505        /// <summary>
2506        /// Runs Instruction Validation Check for GoToLessThenConst
2507        /// </summary>
2508        /// <param name="lineOfCode">line of code to validate</param>
2509        /// <returns>Validation Info</returns>
2510        private ValidationInfo RunGoToLessThenConstInstructionValidationCheck(string lineOfCode)
2511        {
2512            ValidationInfo validationInfo = new ValidationInfo();
2513            const string operationName = "gotolessthenconst";
2514
2515            string[] operationPartsSplitter = { " " };
2516            var operationParts = lineOfCode.Split(operationPartsSplitter, StringSplitOptions.RemoveEmptyEntries);
2517
2518            if (!operationParts.Any())
2519            {
2520                validationInfo.IsValid = validationInfo.IsValid && false;
2521                validationInfo.ValidationMessages.Add($"Invalid Instruction Found (No Operation Parts Found): -> ${lineOfCode}");
```

```
2522            }
2523
2524            if (operationParts.Length != 4)
2525            {
2526                validationInfo.IsValid = validationInfo.IsValid && false;
2527                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2528                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2529            }
2530
2531            int num = 0;
2532            if (operationParts.Length >= 4 && !int.TryParse(operationParts[3],
                out num))
2533            {
2534                validationInfo.IsValid = validationInfo.IsValid && false;
2535                validationInfo.ValidationMessages.Add($"Non-Number Found ($
                    {operationParts[3]}): -> ${lineOfCode}");
2536                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName} #1, #2, 5");
2537            }
2538
2539            var operationOperandPartsRaw = new List<string> { operationParts[1],
                operationParts[2] };
2540            var operationOperandParts = operationOperandPartsRaw.Select(part =>
                part.Replace(",", "").Trim());
2541
2542            foreach (string operationOperandPart in operationOperandParts)
2543            {
2544                if (!this._Registry.Exists(operationOperandPart))
2545                {
2546                    validationInfo.IsValid = validationInfo.IsValid && false;
2547                    validationInfo.ValidationMessages.Add($"Unknown Register
                        Found (${operationOperandPart}): -> ${lineOfCode}");
2548                }
2549            }
2550
2551            return validationInfo;
2552        }
2553
2554        /// <summary>
2555        /// Runs Instruction Validation Check for JumpLabel
2556        /// </summary>
2557        /// <param name="lineOfCode">line of code to validate</param>
2558        /// <returns>Validation Info</returns>
2559        private ValidationInfo RunGoToJumpLabelInstructionValidationCheck(string
                lineOfCode)
2560        {
2561            ValidationInfo validationInfo = new ValidationInfo();
2562            const string operationName = "labelName:";
2563
2564            string[] operationPartsSplitter = { " " };
2565            var operationParts = lineOfCode.Split(operationPartsSplitter,
                StringSplitOptions.RemoveEmptyEntries);
2566
2567            if (!operationParts.Any())
2568            {
2569                validationInfo.IsValid = validationInfo.IsValid && false;
2570                validationInfo.ValidationMessages.Add($"Invalid Instruction Found
                    (No Operation Parts Found): -> ${lineOfCode}");
2571            }
2572
2573            if (operationParts.Length != 1)
2574            {
```

```
2575                validationInfo.IsValid = validationInfo.IsValid && false;
2576                validationInfo.ValidationMessages.Add($"Invalid Instruction
                    Format Found: -> ${lineOfCode}");
2577                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName}:");
2578            }
2579
2580            var jumpLabelRegexValidation = LettersOnlyregex.Match(operationParts
                [0]);
2581            if (!jumpLabelRegexValidation.Success)
2582            {
2583                validationInfo.IsValid = validationInfo.IsValid && false;
2584                validationInfo.ValidationMessages.Add($"Invalid Jump Label Format
                    Found: -> ${lineOfCode}");
2585                validationInfo.ValidationMessages.Add($"Correct Format: ->
                    {operationName}");
2586            }
2587
2588            return validationInfo;
2589        }
2590    }
2591}
```

ProjectItem 'ICompile.cs' has no task items

```
1   using BajanVincyAssembly.Models.ComputerArchitecture;
2   using BajanVincyAssembly.Models.Validation;
3   using System;
4   using System.Collections.Generic;
5   using System.Linq;
6   using System.Text;
7   using System.Threading.Tasks;
8
9   namespace BajanVincyAssembly.Services.Compilers
10  {
11      /// <summary>
12      /// Interface definition for Compilers
13      /// </summary>
14      /// <typeparam name="T"></typeparam>
15      interface ICompile<T>
16      {
17          /// <summary>
18          /// Get Validation Information about supplied code
19          /// </summary>
20          /// <param name="code">Code to validate</param>
21          /// <returns>Validation Info about code</returns>
22          ValidationInfo ValidateCode(string code);
23
24          /// <summary>
25          /// Compiles Code and Returns collection of instructions
26          /// </summary>
27          /// <param name="code">Code to compile</param>
28          /// <returns>Collection of Instructions</returns>
29          IEnumerable<T> Compile(string code);
30      }
31  }
```

ProjectItem 'IProcessor.cs' has no task items

```
1   using BajanVincyAssembly.Models.ComputerArchitecture;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace BajanVincyAssembly.Services.Processor
9   {
10      /// <summary>
11      /// Interface definition for processor operations
12      /// </summary>
13      public interface IProcessor
14      {
15          /// <summary>
16          /// Gets a snapshot of the current collection of registers
17          /// </summary>
18          /// <returns>Collection of registers</returns>
19          IEnumerable<Register> GetRegisters();
20
21          /// <summary>
22          /// Gets all the instructions
23          /// </summary>
24          /// <returns>Collection of Instructions</returns>
25          IEnumerable<Instruction> GetInstructions();
26
27          /// <summary>
28          /// Indicates if there is another instruction to process
29          /// </summary>
30          /// <returns>True if there is another instruction to process</returns>
31          bool HasAnotherInstructionToProcess();
32
33          /// <summary>
34          /// Processes the next instruction
35          /// </summary>
36          void ProcessNextInstruction();
37
38          /// <summary>
39          /// Gets Next Instruction
40          /// </summary>
41          /// <returns></returns>
42          Instruction GetNextInstruction();
43      }
44  }
```

ProjectItem 'Processor.cs' has no task items

```
1   using BajanVincyAssembly.Models;
2   using BajanVincyAssembly.Models.ComputerArchitecture;
3   using BajanVincyAssembly.Services.Registers;
4   using System;
5   using System.Collections.Generic;
6   using System.Linq;
7   using System.Text;
8   using System.Threading.Tasks;
9
10  namespace BajanVincyAssembly.Services.Processor
11  {
12      /// <summary>
13      /// Processes all BV Instructions
14      /// </summary>
15      public class Processor : IProcessor
16      {
17          /// <summary>
18          /// Instantiates a new instance of the <see cref="Processor" class/>
19          /// </summary>
20          public Processor(IEnumerable<Instruction> instructions)
21          {
22              this._Registry = new Registry();
23              this._Instructions = instructions;
24          }
25
26          /// <summary>
27          /// Registry Management System
28          /// </summary>
29          private IRegistry<Register> _Registry;
30
31          /// <summary>
32          /// Collection of Instructions Processed
33          /// </summary>
34          private IEnumerable<Instruction> _Instructions = new List<Instruction>();
35
36          /// <summary>
37          /// Program Instruction Pointer
38          /// </summary>
39          private int _ProgramInstructionPointer = 0;
40
41          /// <summary>
42          /// Updates Program Instruction Pointer
43          /// </summary>
44          /// <param name="instructionNumberToProcessNext">Instruction Number to
            processes next</param>
45          private void SetProgramInstructionPointer(int
            instructionNumberToProcessNext)
46          {
47              this._ProgramInstructionPointer = instructionNumberToProcessNext;
48          }
49
50          /// <summary>
51          /// Finds and returns Index of Jump Label Instruction
52          /// </summary>
53          /// <param name="jumpLabel"></param>
54          /// <returns>integer</returns>
55          private int FindIndexOfJumpLabelInstruction(string jumpLabel)
56          {
57              int instructionCounter = this._Instructions.ToList().FindIndex
                ((instruct) => instruct.Operation == BVOperation.JUMPLABEL
    1   2   3
```

```
58                  && string.Equals(instruct.JumpLabel, jumpLabel,
                    StringComparison.InvariantCultureIgnoreCase));
59
60                  if (instructionCounter == -1)
61                  {
62                      throw new Exception($"Unknown Jump Label Found! -> $
                        {jumpLabel}");
63                  }
64
65                  return instructionCounter;
66              }
67
68              /// <inheritdoc cref="IProcessor"/>
69              public IEnumerable<Register> GetRegisters()
70              {
71                  return this._Registry.GetRegisters();
72              }
73
74              /// <inheritdoc cref="IProcessor"/>
75              public IEnumerable<Instruction> GetInstructions()
76              {
77                  var instructionsCopy = this._Instructions.DeepClone();
78
79                  return instructionsCopy;
80              }
81
82              /// <inheritdoc cref="IProcessor"/>
83              public bool HasAnotherInstructionToProcess()
84              {
85                  return this._ProgramInstructionPointer < this._Instructions.Count();
86              }
87
88              /// <inheritdoc cref="IProcessor"/>
89              public void ProcessNextInstruction()
90              {
91                  if (this._Instructions.Any())
92                  {
93                      Instruction instructionToProcess = this.GetNextInstruction();
94                      this.RunInstructionThroughPipeline(instructionToProcess);
95                  }
96              }
97
98              /// <inheritdoc cref="IProcessor"/>
99              public Instruction GetNextInstruction()
100             {
101                 Instruction instruction;
102
103                 instruction = this._Instructions.Skip
                    (this._ProgramInstructionPointer).Take(1).FirstOrDefault();
104
105                 return instruction;
106             }
107
108             /// <summary>
109             /// Runs the supplied instruction through this pipelined processor
110             /// </summary>
111             /// <param name="instruction"></param>
112             private void RunInstructionThroughPipeline(Instruction instruction)
113             {
114                 Register destinationRegister;
115                 Register operandARegister;
116                 Register operandBRegister;
117                 int operandIntermediate;
```

```
118             int instructionCounter;
119
120             switch (instruction.Operation)
121             {
122             case BVOperation.ADDNS:
123                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
124                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
125                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
126                 destinationRegister.Base10Value =
                operandARegister.Base10Value + operandBRegister.Base10Value;
127                 this._Registry.SaveRegister(destinationRegister);
128                 break;
129             case BVOperation.ADDCONST:
130                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
131                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
132                 operandIntermediate = instruction.OperandImmediate;
133                 destinationRegister.Base10Value =
                operandARegister.Base10Value + operandIntermediate;
134                 this._Registry.SaveRegister(destinationRegister);
135                 break;
136             case BVOperation.ADDPOS:
137                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
138                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
139                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
140                 destinationRegister.Base10Value =
                operandARegister.Base10Value + operandBRegister.Base10Value;
141                 this._Registry.SaveRegister(destinationRegister);
142                 break;
143             case BVOperation.SUBNS:
144                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
145                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
146                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
147                 destinationRegister.Base10Value =
                operandARegister.Base10Value - operandBRegister.Base10Value;
148                 this._Registry.SaveRegister(destinationRegister);
149                 break;
150             case BVOperation.SUBCONST:
151                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
152                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
153                 operandIntermediate = instruction.OperandImmediate;
154                 destinationRegister.Base10Value =
                operandARegister.Base10Value - operandIntermediate;
155                 this._Registry.SaveRegister(destinationRegister);
156                 break;
157             case BVOperation.SUBPOS:
158                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
159                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
160                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
```

```
161                 destinationRegister.Base10Value =
                operandARegister.Base10Value - operandBRegister.Base10Value;
162                 this._Registry.SaveRegister(destinationRegister);
163                 break;
164             case BVOperation.LOGICAND:
165                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
166                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
167                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
168                 destinationRegister.Base10Value =
                operandARegister.Base10Value & operandBRegister.Base10Value;
169                 this._Registry.SaveRegister(destinationRegister);
170                 break;
171             case BVOperation.LOGICANDCOSNT:
172                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
173                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
174                 operandIntermediate = instruction.OperandImmediate;
175                 destinationRegister.Base10Value =
                operandARegister.Base10Value & operandIntermediate;
176                 this._Registry.SaveRegister(destinationRegister);
177                 break;
178             case BVOperation.LOGICOR:
179                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
180                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
181                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
182                 destinationRegister.Base10Value =
                operandARegister.Base10Value & operandBRegister.Base10Value;
183                 this._Registry.SaveRegister(destinationRegister);
184                 break;
185             case BVOperation.LOGICORCONST:
186                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
187                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
188                 operandIntermediate = instruction.OperandImmediate;
189                 destinationRegister.Base10Value =
                operandARegister.Base10Value | operandIntermediate;
190                 this._Registry.SaveRegister(destinationRegister);
191                 break;
192             case BVOperation.SHIFTLEFT:
193                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
194                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
195                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
196                 destinationRegister.Base10Value =
                operandARegister.Base10Value << operandBRegister.Base10Value;
197                 this._Registry.SaveRegister(destinationRegister);
198                 break;
199             case BVOperation.SHIFTLEFTPOS:
200                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
201                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
202                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
```

```
203                 destinationRegister.Base10Value =
                operandARegister.Base10Value << operandBRegister.Base10Value;
204                 this._Registry.SaveRegister(destinationRegister);
205                 break;
206             case BVOperation.SHIFTLEFTCONST:
207                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
208                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
209                 operandIntermediate = instruction.OperandImmediate;
210                 destinationRegister.Base10Value =
                operandARegister.Base10Value << operandIntermediate;
211                 this._Registry.SaveRegister(destinationRegister);
212                 break;
213             case BVOperation.SHIFTRIGHT:
214                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
215                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
216                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
217                 destinationRegister.Base10Value =
                operandARegister.Base10Value >> operandBRegister.Base10Value;
218                 this._Registry.SaveRegister(destinationRegister);
219                 break;
220             case BVOperation.SHIFTRIGHTPOS:
221                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
222                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
223                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
224                 destinationRegister.Base10Value =
                operandARegister.Base10Value >> operandBRegister.Base10Value;
225                 this._Registry.SaveRegister(destinationRegister);
226                 break;
227             case BVOperation.SHIFTRIGHTCONST:
228                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
229                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
230                 operandIntermediate = instruction.OperandImmediate;
231                 destinationRegister.Base10Value =
                operandARegister.Base10Value >> operandIntermediate;
232                 this._Registry.SaveRegister(destinationRegister);
233                 break;
234             case BVOperation.FROMEMEM:
235
236                 break;
237             case BVOperation.FROMMEMCONST:
238
239                 break;
240             case BVOperation.FROMCONST:
241                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
242                 operandIntermediate = instruction.OperandImmediate;
243                 destinationRegister.Base10Value = operandIntermediate;
244                 this._Registry.SaveRegister(destinationRegister);
245                 break;
246             case BVOperation.TOMEM:
247
248                 break;
249             case BVOperation.TOMEMCONST:
250
```

```
                break;
            case BVOperation.TOCONST:

                break;
            case BVOperation.TOCONSTCONST:

                break;
            case BVOperation.COPY:
                destinationRegister = this._Registry.GetRegister
            (instruction.DestinationRegister);
                operandARegister = this._Registry.GetRegister
            (instruction.OperandARegister);
                destinationRegister.Base10Value =
            operandARegister.Base10Value;
                this._Registry.SaveRegister(destinationRegister);
                break;
            case BVOperation.COPYERASE:
                destinationRegister = this._Registry.GetRegister
            (instruction.DestinationRegister);
                operandARegister = this._Registry.GetRegister
            (instruction.OperandARegister);
                destinationRegister.Base10Value =
            operandARegister.Base10Value;
                this._Registry.SaveRegister(destinationRegister);
                operandARegister.Base10Value = 0;
                this._Registry.SaveRegister(operandARegister);
                break;
            case BVOperation.LESSTHEN:
                destinationRegister = this._Registry.GetRegister
            (instruction.DestinationRegister);
                operandARegister = this._Registry.GetRegister
            (instruction.OperandARegister);
                operandBRegister = this._Registry.GetRegister
            (instruction.OperandBRegister);
                destinationRegister.Base10Value =
            operandARegister.Base10Value < operandBRegister.Base10Value ? 1:
            0;
                this._Registry.SaveRegister(destinationRegister);
                break;
            case BVOperation.LESSTHENPOS:
                destinationRegister = this._Registry.GetRegister
            (instruction.DestinationRegister);
                operandARegister = this._Registry.GetRegister
            (instruction.OperandARegister);
                operandBRegister = this._Registry.GetRegister
            (instruction.OperandBRegister);
                destinationRegister.Base10Value =
            operandARegister.Base10Value < operandBRegister.Base10Value ? 1 :
            0;
                this._Registry.SaveRegister(destinationRegister);
                break;
            case BVOperation.LESSTHENCONST:
                destinationRegister = this._Registry.GetRegister
            (instruction.DestinationRegister);
                operandARegister = this._Registry.GetRegister
            (instruction.OperandARegister);
                operandIntermediate = instruction.OperandImmediate;
                destinationRegister.Base10Value =
            operandARegister.Base10Value < operandIntermediate ? 1 : 0;
                this._Registry.SaveRegister(destinationRegister);
                break;
            case BVOperation.LESSTHENEQ:
                destinationRegister = this._Registry.GetRegister
            (instruction.DestinationRegister);
```

```
295                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
296                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
297                 destinationRegister.Base10Value =
                operandARegister.Base10Value <= operandBRegister.Base10Value ? 1
                : 0;
298                 this._Registry.SaveRegister(destinationRegister);
299                 break;
300             case BVOperation.LESSTHENEQPOS:
301                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
302                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
303                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
304                 destinationRegister.Base10Value =
                operandARegister.Base10Value <= operandBRegister.Base10Value ? 1
                : 0;
305                 this._Registry.SaveRegister(destinationRegister);
306                 break;
307             case BVOperation.LESSTHENEQCONST:
308                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
309                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
310                 operandIntermediate = instruction.OperandImmediate;
311                 destinationRegister.Base10Value =
                operandARegister.Base10Value <= operandIntermediate ? 1 : 0;
312                 this._Registry.SaveRegister(destinationRegister);
313                 break;
314             case BVOperation.MORETHEN:
315                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
316                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
317                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
318                 destinationRegister.Base10Value =
                operandARegister.Base10Value > operandBRegister.Base10Value ? 1 :
                0;
319                 this._Registry.SaveRegister(destinationRegister);
320                 break;
321             case BVOperation.MORETHENPOS:
322                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
323                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
324                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
325                 destinationRegister.Base10Value =
                operandARegister.Base10Value > operandBRegister.Base10Value ? 1 :
                0;
326                 this._Registry.SaveRegister(destinationRegister);
327                 break;
328             case BVOperation.MORETHENCONST:
329                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
330                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
331                 operandIntermediate = instruction.OperandImmediate;
332                 destinationRegister.Base10Value =
                operandARegister.Base10Value > operandIntermediate ? 1 : 0;
333                 this._Registry.SaveRegister(destinationRegister);
```

```
334                 break;
335             case BVOperation.MORETHENEQ:
336                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
337                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
338                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
339                 destinationRegister.Base10Value =
                operandARegister.Base10Value >= operandBRegister.Base10Value ? 1
                : 0;
340                 this._Registry.SaveRegister(destinationRegister);
341                 break;
342             case BVOperation.MORETHENEQPOS:
343                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
344                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
345                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
346                 destinationRegister.Base10Value =
                operandARegister.Base10Value >= operandBRegister.Base10Value ? 1
                : 0;
347                 this._Registry.SaveRegister(destinationRegister);
348                 break;
349             case BVOperation.MORETHENEQCONST:
350                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
351                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
352                 operandIntermediate = instruction.OperandImmediate;
353                 destinationRegister.Base10Value =
                operandARegister.Base10Value >= operandIntermediate ? 1 : 0;
354                 this._Registry.SaveRegister(destinationRegister);
355                 break;
356             case BVOperation.XOR:
357                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
358                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
359                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
360                 destinationRegister.Base10Value =
                operandARegister.Base10Value ^ operandBRegister.Base10Value;
361                 this._Registry.SaveRegister(destinationRegister);
362                 break;
363             case BVOperation.XORCONST:
364                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
365                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
366                 operandIntermediate = instruction.OperandImmediate;
367                 destinationRegister.Base10Value =
                operandARegister.Base10Value ^ operandIntermediate;
368                 this._Registry.SaveRegister(destinationRegister);
369                 break;
370             case BVOperation.SAVEADDRESS:
371                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
372                 destinationRegister.Base10Value =
                instruction.InstructionAddress;
373                 destinationRegister.Word = instruction.JumpLabel;
374                 this._Registry.SaveRegister(destinationRegister);
375                 break;
```

```
376             case BVOperation.GOTO:
377                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
378                 instructionCounter = this.FindIndexOfJumpLabelInstruction
                (operandARegister.Word);
379                 this._ProgramInstructionPointer = instructionCounter + 1;
380                 break;
381             case BVOperation.EQ:
382                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
383                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
384                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
385                 destinationRegister.Base10Value =
                operandARegister.Base10Value == operandBRegister.Base10Value ? 1
                : 0;
386                 this._Registry.SaveRegister(destinationRegister);
387                 break;
388             case BVOperation.EQCONST:
389                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
390                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
391                 operandIntermediate = instruction.OperandImmediate;
392                 destinationRegister.Base10Value =
                operandARegister.Base10Value == operandIntermediate ? 1 : 0;
393                 this._Registry.SaveRegister(destinationRegister);
394                 break;
395             case BVOperation.GOTOEQ:
396                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
397                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
398                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
399                 if (operandARegister.Base10Value ==
                operandBRegister.Base10Value)
400                 {
401                     instructionCounter =
                    this.FindIndexOfJumpLabelInstruction
                    (destinationRegister.Word);
402                     this._ProgramInstructionPointer = instructionCounter +
                    1;
403                 }
404                 break;
405             case BVOperation.GOTOEQCONST:
406                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
407                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
408                 operandIntermediate = instruction.OperandImmediate;
409                 if (operandARegister.Base10Value == operandIntermediate)
410                 {
411                     instructionCounter =
                    this.FindIndexOfJumpLabelInstruction
                    (destinationRegister.Word);
412                     this._ProgramInstructionPointer = instructionCounter +
                    1;
413                 }
414                 break;
415             case BVOperation.GOTONOEQ:
416                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
```

```
417                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
418                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
419                 if (operandARegister.Base10Value !=
                operandBRegister.Base10Value)
420                 {
421                     instructionCounter =
                    this.FindIndexOfJumpLabelInstruction
                    (destinationRegister.Word);
422                     this._ProgramInstructionPointer = instructionCounter +
                    1;
423                 }
424                 break;
425             case BVOperation.GOTONOEQCONST:
426                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
427                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
428                 operandIntermediate = instruction.OperandImmediate;
429                 if (operandARegister.Base10Value != operandIntermediate)
430                 {
431                     instructionCounter =
                    this.FindIndexOfJumpLabelInstruction
                    (destinationRegister.Word);
432                     this._ProgramInstructionPointer = instructionCounter +
                    1;
433                 }
434                 break;
435             case BVOperation.GOTOMORETHEN:
436                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
437                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
438                 operandBRegister = this._Registry.GetRegister
                (instruction.OperandBRegister);
439                 if (operandARegister.Base10Value >=
                operandBRegister.Base10Value)
440                 {
441                     instructionCounter =
                    this.FindIndexOfJumpLabelInstruction
                    (destinationRegister.Word);
442                     this._ProgramInstructionPointer = instructionCounter +
                    1;
443                 }
444                 break;
445             case BVOperation.GOTOMORETHENCONST:
446                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
447                 operandARegister = this._Registry.GetRegister
                (instruction.OperandARegister);
448                 operandIntermediate = instruction.OperandImmediate;
449                 if (operandARegister.Base10Value >= operandIntermediate)
450                 {
451                     instructionCounter =
                    this.FindIndexOfJumpLabelInstruction
                    (destinationRegister.Word);
452                     this._ProgramInstructionPointer = instructionCounter +
                    1;
453                 }
454                 break;
455             case BVOperation.GOTOLESSTHEN:
456                 destinationRegister = this._Registry.GetRegister
                (instruction.DestinationRegister);
```

```
457                     operandARegister = this._Registry.GetRegister
                    (instruction.OperandARegister);
458                     operandBRegister = this._Registry.GetRegister
                    (instruction.OperandBRegister);
459                     if (operandARegister.Base10Value <
                    operandBRegister.Base10Value)
460                     {
461                         instructionCounter =
                     this.FindIndexOfJumpLabelInstruction
                     (destinationRegister.Word);
462                         this._ProgramInstructionPointer = instructionCounter +
                     1;
463                     }
464                     break;
465                 case BVOperation.GOTOLESSTHENCONST:
466                     destinationRegister = this._Registry.GetRegister
                    (instruction.DestinationRegister);
467                     operandARegister = this._Registry.GetRegister
                    (instruction.OperandARegister);
468                     operandIntermediate = instruction.OperandImmediate;
469                     if (operandARegister.Base10Value < operandIntermediate)
470                     {
471                         instructionCounter =
                     this.FindIndexOfJumpLabelInstruction
                     (destinationRegister.Word);
472                         this._ProgramInstructionPointer = instructionCounter +
                     1;
473                     }
474                     break;
475                 }
476
477                 // Decide if we need to increment Instruction Pointer
478                 switch (instruction.Operation)
479                 {
480                 case BVOperation.GOTO:
481                 case BVOperation.GOTOEQ:
482                 case BVOperation.GOTOEQCONST:
483                 case BVOperation.GOTONOEQ:
484                 case BVOperation.GOTONOEQCONST:
485                 case BVOperation.GOTOMORETHEN:
486                 case BVOperation.GOTOMORETHENCONST:
487                 case BVOperation.GOTOLESSTHEN:
488                 case BVOperation.GOTOLESSTHENCONST:
489                     break;
490                 default:
491                     this._ProgramInstructionPointer++;
492                     break;
493                 }
494             }
495
496         }
497     }
```

ProjectItem 'IRegistry.cs' has no task items

```
1   using System.Collections.Generic;
2
3   namespace BajanVincyAssembly.Services.Registers
4   {
5       /// <summary>
6       /// Interface defintion for register operations
7       /// </summary>
8       public interface IRegistry<T>
9       {
10          /// <summary>
11          /// Indicates if there exists a register with the supplied Name
12          /// </summary>
13          /// <param name="registerName">Register Name</param>
14          /// <returns></returns>
15          bool Exists(string registerName);
16
17          /// <summary>
18          /// Get Register with supplied Name
19          /// </summary>
20          /// <param name="registerName">Register Name</param>
21          /// <returns>Register</returns>
22          T GetRegister(string registerName);
23
24          /// <summary>
25          /// Save Register into Registry Cache
26          /// </summary>
27          /// <param name="register"> Register to save</param>
28          /// <returns>Register</returns>
29          T SaveRegister(T register);
30
31          /// <summary>
32          /// Resets a register
33          /// </summary>
34          /// <param name="registerName">Register Name</param>
35          /// <returns>Register</returns>
36          T ClearRegister(string registerName);
37
38          /// <summary>
39          /// Gets Registers
40          /// </summary>
41          /// <returns>Collection of Registers</returns>
42          IEnumerable<T> GetRegisters();
43
44          /// <summary>
45          /// Resets all registers in Registry Cache
46          /// </summary>
47          /// <returns>Collection of Registers</returns>
48          IEnumerable<T> ResetRegisters();
49      }
50  }
```

ProjectItem 'Registry.cs' has no task items

```
using BajanVincyAssembly.Models;
using BajanVincyAssembly.Models.ComputerArchitecture;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BajanVincyAssembly.Services.Registers
{
    /// <summary>
    /// Registry System
    /// </summary>
    public class Registry : IRegistry<Register>
    {
        /// <summary>
        /// Instantiates a new instance of the <see cref="Registry" class/>
        /// </summary>
        public Registry()
        {
            this.BuildRegistry();
        }

        /// <summary>
        /// Gets or Sets Registers
        /// </summary>
        private Dictionary<string, Register> Registers;

        /// <summary>
        /// Register Address Lookup
        /// </summary>
        public static readonly Dictionary<string, string> registerAddressLookup =
        new Dictionary<string, string>()
        {
            { "#temp0", "000001 " },
            { "#temp1", "000010 " },
            { "#temp2", "000011 " },
            { "#temp3", "000100 " },
            { "#temp4", "000101 " },
            { "#temp5", "000110 " },
            { "#temp6", "000111 " },
            { "#temp7", "001000 " },
            { "#temp8", "001001 " },
            { "#temp9", "001010 " },
            { "#bv0", "001011 " },
            { "#bv1", "001100 " },
            { "#bv2", "001101 " },
            { "#bv3", "001110 " },
            { "#bv4", "001111 " },
            { "#bv5", "010000 " },
            { "#bv6", "010001 " },
            { "#bv7", "010010 " },
            { "#bv8", "010011 " },
            { "#bv9", "010100 " }

        };

        /// <summary>
        /// Builds new Registry Cache
        /// </summary>
        private void BuildRegistry()
        {
1   2   3
```

```
62              this.Registers = new Dictionary<string, Register>();
63
64              this.Registers.Add("#temp0", new Register("#temp0"));
65              this.Registers.Add("#temp1", new Register("#temp1"));
66              this.Registers.Add("#temp2", new Register("#temp2"));
67              this.Registers.Add("#temp3", new Register("#temp3"));
68              this.Registers.Add("#temp4", new Register("#temp4"));
69              this.Registers.Add("#temp5", new Register("#temp5"));
70              this.Registers.Add("#temp6", new Register("#temp6"));
71              this.Registers.Add("#temp7", new Register("#temp7"));
72              this.Registers.Add("#temp8", new Register("#temp8"));
73              this.Registers.Add("#temp9", new Register("#temp9"));
74              this.Registers.Add("#bv0", new Register("#bv0"));
75              this.Registers.Add("#bv1", new Register("#bv1"));
76              this.Registers.Add("#bv2", new Register("#bv2"));
77              this.Registers.Add("#bv3", new Register("#bv3"));
78              this.Registers.Add("#bv4", new Register("#bv4"));
79              this.Registers.Add("#bv5", new Register("#bv5"));
80              this.Registers.Add("#bv6", new Register("#bv6"));
81              this.Registers.Add("#bv7", new Register("#bv7"));
82              this.Registers.Add("#bv8", new Register("#bv8"));
83              this.Registers.Add("#bv9", new Register("#bv9"));
84          }
85
86          /// <inheritdoc cref="IRegistry{T}"/>
87          public Register ClearRegister(string registerName)
88          {
89              Register register = null;
90
91              if (this.Exists(registerName))
92              {
93                  register = this.Registers[registerName];
94                  register.Clear();
95                  register = register.DeepClone();
96              }
97
98              return register;
99          }
100
101         /// <inheritdoc cref="IRegistry{T}"/>
102         public bool Exists(string registerName)
103         {
104             bool registerExists = !string.IsNullOrEmpty(registerName) &&
                this.Registers.ContainsKey(registerName);
105
106             return registerExists;
107         }
108
109         /// <inheritdoc cref="IRegistry{T}"/>
110         public Register GetRegister(string registerName)
111         {
112             Register register = null;
113
114             if (this.Exists(registerName))
115             {
116                 register = this.Registers[registerName];
117                 register = register.DeepClone();
118             }
119
120             return register;
121         }
122
123         /// <inheritdoc cref="IRegistry{T}"/>
124         public IEnumerable<Register> GetRegisters()
```

```
125             {
126                 var listOfRegisters = this.Registers.Values.ToList();
127                 var deepCloneOfRegisters = listOfRegisters.DeepClone();
128
129                 return deepCloneOfRegisters;
130             }
131
132             /// <inheritdoc cref="IRegistry{T}"/>
133             public IEnumerable<Register> ResetRegisters()
134             {
135                 foreach (var register in this.Registers)
136                 {
137                     register.Value.Clear();
138                 }
139
140                 var listOfRegisters = this.Registers.Values.ToList();
141                 var deepCloneOfRegisters = listOfRegisters.DeepClone();
142
143                 return deepCloneOfRegisters;
144             }
145
146             /// <inheritdoc cref="IRegistry{T}"/>
147             public Register SaveRegister(Register register)
148             {
149                 Register savedRegister = null;
150
151                 if (register != null
152                     && this.Exists(register.Name))
153                 {
154                     this.Registers[register.Name] = register;
155                     savedRegister = this.Registers[register.Name];
156                     savedRegister = savedRegister.DeepClone();
157                 }
158
159                 return savedRegister;
160             }
161         }
162     }
```

ProjectItem 'NotificationWindow.xaml' has no task items

```
<UserControl x:Class="BajanVincyAssembly.UserControls.NotificationWindow"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:BajanVincyAssembly.UserControls"
             mc:Ignorable="d"
             d:DesignHeight="500" d:DesignWidth="600">
    <Grid HorizontalAlignment="Stretch">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition>
            </ColumnDefinition>
        </Grid.ColumnDefinitions>
        <GroupBox Header="Notifications:" Margin="10,10,10,10" FontWeight="Bold" Foreground="#e60000" FontSize="12">
            <ListBox Margin="10,10,10,10" ItemsSource="{Binding Notifications}" Background="Black">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock FontWeight="Regular" Foreground="Red" Margin="10,10,10,10" IsEnabled="False" Text="{Binding}"/>
                        </StackPanel>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </GroupBox>
    </Grid>
</UserControl>

```

ProjectItem 'NotificationWindow.xaml.cs' has no task items

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using System.Windows;
7   using System.Windows.Controls;
8   using System.Windows.Data;
9   using System.Windows.Documents;
10  using System.Windows.Input;
11  using System.Windows.Media;
12  using System.Windows.Media.Imaging;
13  using System.Windows.Navigation;
14  using System.Windows.Shapes;
15
16  namespace BajanVincyAssembly.UserControls
17  {
18      /// <summary>
19      /// Interaction logic for NotificationWindow.xaml
20      /// </summary>
21      public partial class NotificationWindow : UserControl
22      {
23          public NotificationWindow()
24          {
25              InitializeComponent();
26              DataContext = this;
27          }
28
29          /// <summary>
30          /// Gets or sets notifications
31          /// </summary>
32          public List<string> Notifications { get; set; } = new List<string>();
33      }
34  }
```

ProjectItem 'App.config' has no task items

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
    </startup>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="System.Numerics.Vectors"
publicKeyToken="b03f5f7f11d50a3a" culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-4.1.4.0" newVersion="4.1.4.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Runtime.CompilerServices.Unsafe"
publicKeyToken="b03f5f7f11d50a3a" culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-4.0.6.0" newVersion="4.0.6.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Buffers" publicKeyToken="cc7b13ffcd2ddd51"
culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-4.0.3.0" newVersion="4.0.3.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.ValueTuple" publicKeyToken="cc7b13ffcd2ddd51"
culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-4.0.3.0" newVersion="4.0.3.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

ProjectItem 'App.xaml' has no task items

```
<Application x:Class="BajanVincyAssembly.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:BajanVincyAssembly"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>

```

ProjectItem 'App.xaml.cs' has no task items

```
1   using System;
2   using System.Collections.Generic;
3   using System.Configuration;
4   using System.Data;
5   using System.Linq;
6   using System.Threading.Tasks;
7   using System.Windows;
8
9   namespace BajanVincyAssembly
10  {
11      /// <summary>
12      /// Interaction logic for App.xaml
13      /// </summary>
14      public partial class App : Application
15      {
16      }
17  }
```

ProjectItem 'MainWindow.xaml' has no task items

```
1   <Window x:Class="BajanVincyAssembly.MainWindow"
2           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6           xmlns:local="clr-namespace:BajanVincyAssembly"
7           mc:Ignorable="d"
8           Title="Bajan Vincy Assembly IDE" MinHeight="700" Width="800">
9
10      <Grid HorizontalAlignment="Stretch">
11          <Grid.RowDefinitions>
12              <RowDefinition Height="*"></RowDefinition>
13              <RowDefinition Height="0.7*"></RowDefinition>
14          </Grid.RowDefinitions>
15          <Grid.ColumnDefinitions>
16              <ColumnDefinition Width="*"></ColumnDefinition>
17              <ColumnDefinition Width="0.7*"></ColumnDefinition>
18          </Grid.ColumnDefinitions>
19          <Grid Grid.Row="0" Grid.Column="0" HorizontalAlignment="Stretch">
20              <Grid.RowDefinitions>
21                  <RowDefinition Height="30"></RowDefinition>
22                  <RowDefinition Height="*"></RowDefinition>
23              </Grid.RowDefinitions>
24              <Grid.ColumnDefinitions>
25                  <ColumnDefinition></ColumnDefinition>
26              </Grid.ColumnDefinitions>
27              <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="0">
28                  <StackPanel.Resources>
29                      <Style TargetType="Button">
30                          <Setter Property="Padding" Value="5, 5, 5, 5"></Setter>
31                          <Setter Property="FontWeight" Value="Bold"></Setter>
32                      </Style>
33                  </StackPanel.Resources>
34                  <Button Name="Button_Compile" Background="DarkBlue" Foreground="White"
    Click="Button_Click_CompileBVAssemblyCode">Compile</Button>
35                  <Button Name="Button_RunAll"  Background="Green" Foreground="White"
    Click="Button_Click_RunAll">Run All</Button>
36                  <Button Name="Button_Debug" Background="YellowGreen"
    Foreground="White" Click="Button_Click_Debug">Debug</Button>
37                  <Button Name="Button_DebugNext" Background="YellowGreen"
    Foreground="White" Click="Button_Click_Debug">Debug: Next</Button>
38                  <Button Name="Button_Stop" Background="DarkRed" Foreground="White"
    Click="Button_Click_Stop">Stop</Button>
39              </StackPanel>
40              <GroupBox Grid.Row="1" Grid.Column="0" Header="Assembly Code"
    Margin="10,10,10,10" BorderBrush="Black">
41                  <TextBox x:Name="TextBox_Code" Background="White" IsEnabled="True"
    VerticalAlignment="Stretch" HorizontalAlignment="Stretch" TextWrapping="Wrap"
    Margin="5,5,5,5" Padding="5,5,5,5" AcceptsReturn="True"></TextBox>
42              </GroupBox>
43          </Grid>
44          <GroupBox Grid.Row="0" Grid.Column="1" Header="Registers and Memory"
    Margin="10,10,10,10" BorderBrush="Black">
45              <ListView x:Name="ListViewOfRegisters" Margin="5">
46                  <ListView.View>
47                      <GridView>
48                          <GridViewColumn Header="Name" Width="Auto"
    DisplayMemberBinding="{Binding Name}"/>
49                          <GridViewColumn Header="Base 10 Value" Width="Auto"
    DisplayMemberBinding="{Binding Base10Value}"/>
50                          <GridViewColumn Header="Value" Width="Auto"
    DisplayMemberBinding="{Binding HexValue}"/>
```

```
51                      </GridView>
52                  </ListView.View>
53              </ListView>
54          </GroupBox>
55          <TabControl Grid.Row="1" Grid.ColumnSpan="2" HorizontalAlignment="Stretch">
56              <TabItem Header="Compile Errors">
57                  <GroupBox Grid.Row="1" Grid.ColumnSpan="2" Header=""
    Margin="10,10,10,10" BorderBrush="Black">
58                      <ScrollViewer>
59                          <TextBlock x:Name="TextBlock_CompileErrors" Background="White"
    Foreground="Red" VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
    TextWrapping="Wrap" Margin="5,5,5,5" Padding="5,5,5,5"></TextBlock>
60                      </ScrollViewer>
61                  </GroupBox>
62              </TabItem>
63              <TabItem Header="Run Time Errors">
64                  <GroupBox Grid.Row="1" Grid.ColumnSpan="2" Header=""
    Margin="10,10,10,10" BorderBrush="Black">
65                      <ScrollViewer>
66                          <TextBlock x:Name="TextBlock_RunTimeErrors" Background="White"
    VerticalAlignment="Stretch" HorizontalAlignment="Stretch" TextWrapping="Wrap"
    Margin="5,5,5,5" Padding="5,5,5,5"></TextBlock>
67                      </ScrollViewer>
68                  </GroupBox>
69              </TabItem>
70              <TabItem Header="Output">
71                  <GroupBox Grid.Row="1" Grid.ColumnSpan="2" Header=""
    Margin="10,10,10,10" BorderBrush="Black">
72                      <ScrollViewer>
73                          <TextBlock x:Name="TextBlock_Ouput" Background="White"
    Foreground="DarkGreen" VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
    TextWrapping="Wrap" Margin="5,5,5,5" Padding="5,5,5,5"></TextBlock>
74                      </ScrollViewer>
75                  </GroupBox>
76              </TabItem>
77          </TabControl>
78      </Grid>
79  </Window>
80
```

ProjectItem 'MainWindow.xaml.cs' has no task items

```
1   using BajanVincyAssembly.Models.ComputerArchitecture;
2   using BajanVincyAssembly.Models.Validation;
3   using BajanVincyAssembly.Services.Compilers;
4   using BajanVincyAssembly.Services.Processor;
5   using BajanVincyAssembly.Services.Registers;
6   using BajanVincyAssembly.UserControls;
7   using System;
8   using System.Collections.Generic;
9   using System.Collections.ObjectModel;
10  using System.Linq;
11  using System.Text;
12  using System.Threading.Tasks;
13  using System.Windows;
14  using System.Windows.Controls;
15  using System.Windows.Data;
16  using System.Windows.Documents;
17  using System.Windows.Input;
18  using System.Windows.Media;
19  using System.Windows.Media.Imaging;
20  using System.Windows.Navigation;
21  using System.Windows.Shapes;
22
23  namespace BajanVincyAssembly
24  {
25      /// <summary>
26      /// Interaction logic for MainWindow.xaml
27      /// </summary>
28      public partial class MainWindow : Window
29      {
30          public MainWindow()
31          {
32              InitializeComponent();
33
34              DataContext = this;
35
36              this.OuputMessages.Add($"Initializing IDE...");
37              this.UpdateViewOfOuputMessages();
38
39              this.Reset();
40
41              this.ListViewOfRegisters.ItemsSource = this.Registers;
42          }
43
44          /// <summary>
45          /// Compiler for BV Assembly Code
46          /// </summary>
47          private ICompile<Instruction> _BVCompiler = new BVCompiler();
48
49          /// <summary>
50          /// Validation Info for Code
51          /// </summary>
52          private ValidationInfo _Code_ValidationInfo = new ValidationInfo();
53
54          /// <summary>
55          /// Processor to process BV Assembly Instructions
56          /// </summary>
57          private IProcessor _Processor;
58
59          /// <summary>
60          /// Contain Information about Registers
61          /// </summary>
```

1 2

```
62          public ObservableCollection<Register> Registers = new
            ObservableCollection<Register>();
63
64          /// <summary>
65          /// Instructions loaded in the processor
66          /// </summary>
67          public ObservableCollection<Instruction> ProcessorInstructions = new
            ObservableCollection<Instruction>();
68
69          /// <summary>
70          /// Ouput Messages
71          /// </summary>
72          public ObservableCollection<string> OuputMessages = new
            ObservableCollection<string>();
73
74          /// <summary>
75          /// Compiles BV Assembly Code
76          /// </summary>
77          /// <param name="linesOfCode"></param>
78          public void Button_Click_CompileBVAssemblyCode(object sender,
            RoutedEventArgs e)
79          {
80              this.Reset();
81
82              var rawCode = this.TextBox_Code.Text;
83
84              this._Code_ValidationInfo = this._BVCompiler.ValidateCode(rawCode);
85
86              this.UpdateViewOfCompileErrors();
87
88              if (!this._Code_ValidationInfo.IsValid)
89              {
90                  this.ShowNotificationsWindow(new List<string>() { $"There are
                    compile Issues. Check 'Compile Errors' Tab!" });
91              }
92              else
93              {
94                  // Generate/Build Instructions
95                  IEnumerable<Instruction> compiledInstructions =
                    this._BVCompiler.Compile(rawCode);
96                  this._Processor = new Processor(compiledInstructions);
97
98                  this.UpdateLatestSnapshotOfProcessorInstructions();
99
100                 if (this.ProcessorInstructions.Any())
101                 {
102                     // Jump onto Main UI Thread
103                     Application.Current.Dispatcher.Invoke(new Action(() =>
104                     {
105                         this.Button_Compile.IsEnabled = false;
106                         this.Button_RunAll.IsEnabled = true;
107                         this.Button_Debug.IsEnabled = true;
108                         this.Button_DebugNext.IsEnabled = true;
109                         this.Button_Stop.IsEnabled = true;
110                     }));
111                 }
112             }
113         }
114
115         /// <summary>
116         /// Begins Running all the user's code
117         /// </summary>
118         /// <param name="sender"></param>
119         /// <param name="e"></param>
```

```
120         public void Button_Click_RunAll(object sender, RoutedEventArgs e)
121         {
122             this.OuputMessages.Add($"Running all code...");
123             this.UpdateViewOfOuputMessages();
124
125             // Jump onto Main UI Thread
126             Application.Current.Dispatcher.Invoke(new Action(() =>
127             {
128                 this.Button_Compile.IsEnabled = false;
129                 this.Button_RunAll.IsEnabled = false;
130                 this.Button_Debug.IsEnabled = false;
131                 this.Button_DebugNext.IsEnabled = false;
132                 this.Button_Stop.IsEnabled = true;
133             }));
134
135             while (this._Processor.HasAnotherInstructionToProcess())
136             {
137                 try
138                 {
139                     Instruction nextInstruction =
                        this._Processor.GetNextInstruction();
140                     this.OuputMessages.Add($"Running next Insruction...$
                        {Enum.GetName(typeof(BVOperation),
                        nextInstruction.Operation)} -> {nextInstruction.ToString
                        ()}");
141                     this.UpdateViewOfOuputMessages();
142                     this._Processor.ProcessNextInstruction();
143                     this.UpdateLatestSnapshotOfRegistryState();
144                 }
145                 catch (Exception exception)
146                 {
147                     this.UpdateViewOfRunTimeErrors(new List<string>() {
                        exception.Message });
148
149                     // Jump onto Main UI Thread
150                     Application.Current.Dispatcher.Invoke(new Action(() =>
151                     {
152                         this.Button_Compile.IsEnabled = true;
153                         this.Button_RunAll.IsEnabled = false;
154                         this.Button_Debug.IsEnabled = false;
155                         this.Button_DebugNext.IsEnabled = false;
156                         this.Button_Stop.IsEnabled = false;
157                     }));
158
159                     this.ShowNotificationsWindow(new List<string>() { $"There
                        are Run Time Issues. Check 'Run Time Errors' Tab!" });
160                 }
161             }
162
163             this.OuputMessages.Add($"Finished Executing all code...");
164             this.UpdateViewOfOuputMessages();
165         }
166
167         /// <summary>
168         /// Begins Debugging Program
169         /// </summary>
170         /// <param name="sender"></param>
171         /// <param name="e"></param>
172         public void Button_Click_Debug(object sender, RoutedEventArgs e)
173         {
174             // Jump onto Main UI Thread
175             Application.Current.Dispatcher.Invoke(new Action(() =>
176             {
177                 this.Button_Compile.IsEnabled = false;
```

```
178                 this.Button_RunAll.IsEnabled = false;
179                 this.Button_Debug.IsEnabled = false;
180                 this.Button_DebugNext.IsEnabled = true;
181                 this.Button_Stop.IsEnabled = true;
182             }));
183 
184             if (this._Processor.HasAnotherInstructionToProcess())
185             {
186                 Instruction nextInstruction = this._Processor.GetNextInstruction
                    ();
187                 this.OuputMessages.Add($"Running next Insruction...${Enum.GetName
                    (typeof(BVOperation), nextInstruction.Operation)} ->
                    {nextInstruction.ToString()}");
188                 this.UpdateViewOfOuputMessages();
189 
190                 try
191                 {
192                     this._Processor.ProcessNextInstruction();
193                     this.UpdateLatestSnapshotOfRegistryState();
194                 }
195                 catch (Exception exception)
196                 {
197                     this.UpdateViewOfRunTimeErrors(new List<string>() {
                        exception.Message });
198 
199                     // Jump onto Main UI Thread
200                     Application.Current.Dispatcher.Invoke(new Action(() =>
201                     {
202                         this.Button_Compile.IsEnabled = true;
203                         this.Button_RunAll.IsEnabled = false;
204                         this.Button_Debug.IsEnabled = false;
205                         this.Button_DebugNext.IsEnabled = false;
206                         this.Button_Stop.IsEnabled = false;
207                     }));
208 
209                     this.ShowNotificationsWindow(new List<string>() { $"There
                        are Run Time Issues. Check 'Run Time Errors' Tab!" });
210                 }
211             }
212 
213             // Jump onto Main UI Thread
214             Application.Current.Dispatcher.Invoke(new Action(() =>
215             {
216                 this.Button_DebugNext.IsEnabled =
                    this._Processor.HasAnotherInstructionToProcess() ? true : false;
217             }));
218         }
219 
220         /// <summary>
221         /// Stops Running Code
222         /// </summary>
223         /// <param name="sender"></param>
224         /// <param name="e"></param>
225         public void Button_Click_Stop(object sender, RoutedEventArgs e)
226         {
227             this.Reset();
228         }
229 
230         /// <summary>
231         /// Resets IDE State
232         /// </summary>
233         private void Reset()
234         {
235             this._Processor = new Processor(this.ProcessorInstructions);
```

```
236             this._Code_ValidationInfo = new ValidationInfo();
237             this.OuputMessages = new ObservableCollection<string>();
238             this.OuputMessages.Add($"Resetting IDE...");
239             this.UpdateLatestSnapshotOfRegistryState();
240             this.UpdateViewOfCompileErrors();
241             this.UpdateViewOfRunTimeErrors(new List<string>() { string.Empty });
242             this.UpdateViewOfOuputMessages();
243
244             // Jump onto Main UI Thread
245             Application.Current.Dispatcher.Invoke(new Action(() =>
246             {
247                 this.Button_Compile.IsEnabled = true;
248                 this.Button_RunAll.IsEnabled = false;
249                 this.Button_Debug.IsEnabled = false;
250                 this.Button_DebugNext.IsEnabled = false;
251                 this.Button_Stop.IsEnabled = false;
252             }));
253         }
254
255         /// <summary>
256         /// Gets and saves latest snapshot of registry cache. Updates Registry
            View
257         /// </summary>
258         private void UpdateLatestSnapshotOfRegistryState()
259         {
260             // Get Latest State Of Registers
261             // Jump onto Main UI Thread
262             Application.Current.Dispatcher.Invoke(new Action(() =>
263             {
264                 this.Registers.Clear();
265
266                 var registerState = this._Processor.GetRegisters().ToList();
267
268                 if (registerState.Any())
269                 {
270                     foreach (var register in registerState)
271                     {
272                         this.Registers.Add(register);
273                     }
274                 }
275             }));
276
277         }
278
279         /// <summary>
280         /// Update View of Compile Errors
281         /// </summary>
282         private void UpdateViewOfCompileErrors()
283         {
284             var compileErrors = string.Join($"{string.Join(string.Empty,
                BVCompiler.LineDelimitter)}",
                this._Code_ValidationInfo.ValidationMessages);
285
286             this.TextBlock_CompileErrors.Text = compileErrors;
287         }
288
289         /// <summary>
290         /// Update View of Run Time Errors
291         /// </summary>
292         /// <param name="runTimeErrors"></param>
293         private void UpdateViewOfRunTimeErrors(List<string> runTimeErrors)
294         {
295             var runTimeErrorsStr = string.Join($"{string.Join(string.Empty,
                BVCompiler.LineDelimitter)}", runTimeErrors);
```

```
296
297             this.TextBlock_RunTimeErrors.Text = runTimeErrorsStr;
298         }
299
300         /// <summary>
301         /// Update View of Output Messages
302         /// </summary>
303         private void UpdateViewOfOuputMessages()
304         {
305             var outputMessages = string.Join($"{string.Join(string.Empty,
                BVCompiler.LineDelimitter)}", this.OuputMessages);
306
307             this.TextBlock_Ouput.Text = outputMessages;
308         }
309
310         /// <summary>
311         /// Gets and saves latest snapshot of Processor Instructions. Updates
                Porcessor Instructions
312         /// </summary>
313         private void UpdateLatestSnapshotOfProcessorInstructions()
314         {
315             // Get Latest State Of Processor Instructions
316             // Jump onto Main UI Thread
317             Application.Current.Dispatcher.Invoke(new Action(() =>
318             {
319                 this.ProcessorInstructions.Clear();
320
321                 var processorStateOfInstructions =
                    this._Processor.GetInstructions().ToList();
322
323                 if (processorStateOfInstructions.Any())
324                 {
325                     foreach (var instruction in processorStateOfInstructions)
326                     {
327                         this.ProcessorInstructions.Add(instruction);
328                     }
329                 }
330             }));
331         }
332
333         /// <summary>
334         /// Shows notifications
335         /// </summary>
336         /// <param name="notifications">Notifications to show</param>
337         private void ShowNotificationsWindow(List<string> notifications)
338         {
339             if (notifications != null)
340             {
341                 NotificationWindow notificationsWindow = new NotificationWindow()
342                 {
343                     Notifications = notifications
344                 };
345                 Window window = new Window
346                 {
347                     Title = "Notifications Window",
348                     Content = notificationsWindow,
349                     Width = 500,
350                     Height = 300,
351                     MinHeight = 300,
352                     MinWidth = 500,
353                     MaxHeight = 300,
354                     MaxWidth = 500,
355                     Owner = this,
356                     WindowStartupLocation = WindowStartupLocation.CenterOwner
```

```
357                     };
358
359                     window.ShowDialog();
360                 }
361             }
362         }
363     }
```

ProjectItem 'packages.config' has no task items

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Microsoft.Bcl.AsyncInterfaces" version="1.1.0" targetFramework="net472" />
  <package id="Newtonsoft.Json" version="12.0.3" targetFramework="net472" />
  <package id="System.Buffers" version="4.5.0" targetFramework="net472" />
  <package id="System.Memory" version="4.5.3" targetFramework="net472" />
  <package id="System.Numerics.Vectors" version="4.5.0" targetFramework="net472" />
  <package id="System.Runtime.CompilerServices.Unsafe" version="4.7.0" targetFramework="net472" />
  <package id="System.Text.Encodings.Web" version="4.7.0" targetFramework="net472" />
  <package id="System.Threading.Tasks.Extensions" version="4.5.2" targetFramework="net472" />
  <package id="System.ValueTuple" version="4.5.0" targetFramework="net472" />
</packages>
```

U

V

W

X