Proposal

Hanzhou Tang
hanzhout@smu.edu

Jiutian Yu
jiutiany@smu.com

2019 April

Abstract

We want to build an assembly simulator which should support basic x86 assembly instructions.

1 Motivation

To implement a basic architecture is a direct way to reflect what we are going to learn in this class. We could also say we are going to implement a simple virtual machine. Beyond simply implementing the original architecture, we want to add some new functions based on this class to complete the architecture.

2 Why Java

It's tempted to implement an assembly simulator in a low level language like C or C++. However, after some considerations, we decided to choose Java. There are several reasons which will be show below.

2.1 Java is easy for memory management

With C++11, it has smarter pointers to make life easier [**?**]. However, it's implemented in library level instead of language supporting. Sometimes, when we mistakenly mixed raw pointers and smart pointers , smart pointers may become useless.

2.2 Java is easy to test

There are lots of powerful java testing framework to do unit test. For example, JUnit or Groovy Spock. Meanwhile, because Java support proxy object, it's much easy to mock objects and record function invoking.

2.3 Java is easy for package manage

With the help of gradle [**?**], it's very easy for us to import different libraries and do deployment. We believe it could save us lots of time and efforts.

2.4 Java is easy to integrate with REST API

If we could finish our project well, we may want to implement an online version for all users. We could easily provide REST API with the help of Spring [**?**].

3 Our registers

We do want to implement some basic functionality of x86 platform. So we decide to imitate ten 32 bits registers. A table, which is Table1, of our registers is shown below. The table originally comes from [**?**].

As you can see, we ignore all segment registers. The reason is that according to our design, only one process can run at one time and no context switch. We think, in this case, segment information is unnecessary. Maybe we're wrong, we may change our decision later.

4 Our instruction set

We want to support a subset of assembly instructions on x86 platform. By doing some researches [**?**], we provide a table which contains all instructions we want support.

5 Our progress

We finished most of our assembler and instruction set.

In the assembler, the data segment can support expression, such as ($ - 10)

We didn't re-invent the instruction set, instead, we follow the Intel standard to represent our instructions. For more detail, you can find at http://www.c-jump.com/CIS77/CPU/x86/lecture.html

The only thing left is to implement the virtual machine which supports our instruction.

By some discussion, we decide to implement a 5-stage virtual machine to execute our instructions.

We also want to provide the ability to expose our internal status. We may choose to export internal status by html. If we can finish our virtual machine early.

We provide part of our implementation of assembler and instruction.

Register	Descriptions
EAX	Accumulator. 0 to 7 can be refered as AL. 8 to 15 can be refered as AH. 0 to 15 can be refered as AX.
EBX	Memory pointer, base Register. 0 to 7 can be refered as BL. 8 to 15 can be refered as BH. 0 to 15 can be refered as BX.
ECX	Loop control. 0 to 7 can be refered as CL. 8 to 15 can be refered as CH. 0 to 15 can be refered as CX.
EDX	Integer multiplication, integer division. 0 to 7 can be refered as DL. 8 to 15 can be refered as DH. 0 to 15 can be refered as DX.
ESI	String instruction source pointer.
EDI	String instruction destination pointer.
ESP	Stack Pointer.
EBP	Stack frame base Pointer.
EIP	Instruction pointer register.
EFLAGS	Flag register.

Table 1: The registers we want to imitate

mov	Copy data from one place to another place.
push	Push register, memory location or immediate value onto stack.
pop	Pop the first item from stack.
add	Add two number
sub	Subtraction
cbw	Sign-extends register AL.
cwd	Sign-extends register AX.
bswap	Reverse the bytes of a 32-bit register.
and	Logic and.
or	Logic or.
xor	Logic xor.
not	Logic not.
sal/shl	Left shift.
sar	Arithmetric right shift.
shr	Logic right shift.
cmpsb/cmpsw/cmpsd	Compare the values at location.
lodsb/lodsw/lodsd	Loads the values at location.
stosb/stosw/stosd	Save the values at register to memory.
rep/repne	Repeat a specified instruction by condition
jmp/jcc/jecxz	Unconditonal/conditional jump
call	Push content to stack then do unconditonal jump.
ret	Pop stack then do unconditonal jump.
enter	Create a stack frame for function.
leave	Remove a stack frame of function.
loop/loope/loopz/loopne/loopnz	Loop.
nop	Advance the instruction pointer.

Table 2: The instructions we want to imitate

6 Part of our source code

1. **Simpler Lexer**

```
1  package assembler;
2
3  import org.apache.commons.lang3.StringUtils;
```

```
4     import org.apache.commons.lang3.tuple.Pair;
5     import org.apache.log4j.Logger;
6     import org.springframework.stereotype.Component;
7
8
9     import java.math.BigInteger;
10    import java.nio.file.Files;
11    import java.nio.file.Paths;
12    import java.util.*;
13    import
    java.util.concurrent.atomic.AtomicInteger;
14    import java.util.function.Predicate;
15    import java.util.stream.Collectors;
16    import java.util.stream.Stream;
17
18    /**
19     * The real implementation of Lexer interface.
20     * The Simple will first strip all comments and
    then split content into different tokens by
    delimiters.
21     * @author  Hanzhou Tang
22     */
23    @Component
24    public class SimpleLexer implements Lexer {
25        private static final Logger LOGGER =
    Logger.getLogger(SimpleLexer.class);
26        //private Map<Integer, String> lines = null;
27        private List<SourceCodeWrapper> lines =
    null;
28        public List<SourceCodeWrapper> getLines() {
29            return lines;
30        }
31
32        public static Set<Character> delimiters =
    new HashSet<>();
33        public static Map<String, Token> char2Token
    = new HashMap<>();
34
35        static {
36            delimiters.add(' ');
```

```
37              delimiters.add('\t');
38              delimiters.add('\n');
39              delimiters.add(',');
40              delimiters.add(':');
41              delimiters.add('+');
42              delimiters.add('-');
43              delimiters.add('*');
44              delimiters.add('/');
45              delimiters.add('[');
46              delimiters.add(']');
47              delimiters.add('(');
48              delimiters.add(')');
49              delimiters.add('$');
50              delimiters.add('\'');
51              char2Token.put(",", Token.Comma);
52              char2Token.put(":", Token.Colon);
53              char2Token.put("+", Token.Add);
54              char2Token.put("-", Token.Sub);
55              char2Token.put("*", Token.Mul);
56              char2Token.put("/", Token.Div);
57              char2Token.put("[",
    Token.LeftSquareBracket);
58              char2Token.put("]",
    Token.RightSquareBracket);
59              char2Token.put("\n", Token.NewLine);
60              char2Token.put("(", Token.LeftParent);
61              char2Token.put(")", Token.RightParent);
62              char2Token.put("$", Token.DollarSign);
63              char2Token.put("'",Token.Quote);
64          }
65
66          Status status = null;
67
68          @Override
69          public Status getStatus() {
70              return status;
71          }
72
73          public void setStatus(Status status) {
74              this.status = status;
```

```
75          }
76
77          private Pair<Integer, String>
    stripComments(final Map.Entry<Integer, String>
    record) {
78              String retStr =
    StringUtils.chomp(record.getValue());
79              int index = retStr.indexOf(';');
80              if (index != -1) {
81                  retStr = retStr.substring(0, index);
82              }
83              retStr = StringUtils.strip(retStr);
84              if (StringUtils.isNotEmpty(retStr)) {
85                  retStr += "\n";
86              }
87              return Pair.of(record.getKey(), retStr);
88          }
89
90          @Override
91          public void readFile(String filename)
    throws Exception {
92              AtomicInteger counter = new
    AtomicInteger();
93              lines = Files.lines(Paths.get(filename))
94                      .collect(Collectors.toMap(s ->
    counter.incrementAndGet(), s -> s))
95
    .entrySet().stream().map(this::stripComments).filter(x
    -> StringUtils.isNotEmpty(x.getValue()))
96                      .map(e -> new
    SourceCodeWrapper(e.getKey(),e.getValue())).collect(Collectors.toL
97              if(lines.size()>0){
98                  status = new Status();
99
    status.setCodeOfCurrentLine(lines.get(0).getCode());
100             }
101             else{
102                 throw  new Exception("file is
    empty");
103             }
```

```
104         }
105
106
107         private LexemeTokenWrapper
    getNextLexemeAndToken(){
108             Token token = getNextToken();
109             if(token.equals(Token.EndofContent)){
110                 return new
    LexemeTokenWrapper(Token.EndofContent,"");
111             }
112             else{
113                 return new
    LexemeTokenWrapper(getStatus().getCurrentToken(),getStatus().getCu
114             }
115
116         }
117
118         @Override
119         public List<LexemeTokenWrapper>
    lookAheadK(int k){
120             Status oldStatus = new
    Status(getStatus());
121             List<LexemeTokenWrapper> ret =
    Stream.generate(this::getNextLexemeAndToken).limit(k).collect(Coll
122             setStatus(oldStatus);
123             return ret;
124         }
125
126         public static int findStr_if(final String
    str, final int begin, Predicate<Character>
    predicate) {
127             int i = begin;
128             for (; i < str.length(); i++) {
129                 if (predicate.test(str.charAt(i))) {
130                     return i;
131                 }
132             }
133             return i;
134         }
135
```

```
136         @Override
137         public Token getNextToken() {
138             if (status.getIterator()>=lines.size())
    {
139                 return Token.EndofContent;
140             } else {
141                 if (status.getCharIndex() ==
    status.getCodeOfCurrentLine().length()) {
142 
    status.setIterator(status.getIterator()+1);
143 
    if(status.getIterator()==lines.size()){
144                         return Token.EndofContent;
145                     }
146                     SourceCodeWrapper wrapper =
    lines.get(status.getIterator());
147                     status.setCharIndex(0);
148 
    status.setCodeOfCurrentLine(wrapper.getCode());
149 
    status.setLineIndex(wrapper.getLineNumber());
150 
    status.setCurrentToken(Token.Invalid);
151                     status.setCurrentLexeme("");
152                 }
153 
    status.setCurrentToken(Token.Invalid);
154                 final String str =
    status.getCodeOfCurrentLine();
155                 int i = findStr_if(str,
    status.getCharIndex(), ch -> ch == '\n' ||
    !Character.isWhitespace(ch));
156                 int j = findStr_if(str, i,
    delimiters::contains);
157                 if (i == j) {
158                     status.setCurrentLexeme("" +
    str.charAt(i));
159                     status.setCharIndex(i + 1);
160                     if
    (char2Token.containsKey(status.getCurrentLexeme()))
```

```
    {
161                         Token token =
    char2Token.get(status.getCurrentLexeme());
162
    status.setCurrentToken(token);
163                 } else {
164                     LOGGER.error("the Character
    (" + status.getCurrentLexeme() + ") " + "at line
    " + status.getLineIndex() + " is invalid");
165                 }
166             } else {
167                 String lexeme =
    str.substring(i, j);
168                 status.setCurrentLexeme(lexeme);
169                 status.setCharIndex(j);
170                 if
    (StringUtils.isNumeric(lexeme)) {
171
    status.setCurrentToken(Token.Number);
172                 } else if
    (lexeme.startsWith(".")) {
173
    status.setCurrentToken(Token.DotString);
174                 } else {
175
    status.setCurrentToken(Token.String);
176                 }
177             }
178             if (status.getCurrentToken() ==
    Token.Invalid) {
179                 LOGGER.warn("the word (" +
    status.getCurrentLexeme() + ") at line " +
    status.getLineIndex() + " is invalid");
180                 //throw new Exception("the word
    (" + status.getCurrentLexeme() + ") at line " +
    status.getLineIndex() + " is invalid");
181             }
182             return status.getCurrentToken();
183         }
184     }
```

```
185
186     }
187
```

---

2. Simple Parser

```
1     package assembler;
2
3     import OutputFile.DataType;
4     import OutputFile.ObjFile;
5     import org.apache.log4j.Logger;
6     import
    org.springframework.beans.factory.annotation.Autowired;
7     import org.springframework.stereotype.Component;
8
9     import java.math.BigInteger;
10    import java.util.List;
11    import java.util.Optional;
12
13    /**
14     * A parser to convert a assembly file into
    binary file.
15     * For now, the parser will pass code segment
    twice.
16     * It will collect all label information in the
    first time.
17     * In the second time, the real parsing is done.
18     * @author Hanzhou Tang
19     */
20
21    @Component
22    public class SimpleParser {
23        @Autowired
24        private Lexer lexer;
25        private static final Logger LOGGER =
    Logger.getLogger(SimpleParser.class);
26
27        public Lexer getLexer() {
28            return lexer;
```

```
29          }
30
31          public ObjFile parse(String name) throws
    Exception {
32              lexer.readFile(name);
33              ObjFile objFile = new ObjFile();
34              parse(objFile);
35              return objFile;
36          }
37
38          protected void parse(ObjFile obj) throws
    Exception {
39              LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
40              if
    (wrapper.getToken().equals(Token.DotString) &&
    ".data".equals(wrapper.getLexeme())) {
41                  dataSegment(obj);
42              }
43          }
44
45          protected void dataName(ObjFile obj) throws
    Exception {
46              LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
47              if
    (wrapper.getToken().equals(Token.String)) {
48                  Optional<DataType> dataType =
    DataType.of(wrapper.lexeme);
49                  if (!dataType.isPresent()) {
50
    obj.getDataSegment().addNmae(wrapper.getLexeme());
51                      lexer.getNextToken();
52                      dataName(obj);
53                  }
54              }
55          }
56
57          protected void dataDefine(ObjFile obj)
    throws Exception {
```

```
58              LOGGER.info("in data Define");
59              dataName(obj);
60              Token token = lexer.getNextToken();
61              LOGGER.info("in data Define, token " +
    token);
62              if (token.equals(Token.String)) {
63                  String lexem =
    lexer.getStatus().getCurrentLexeme();
64                  Optional<DataType> dataType =
    DataType.of(lexem);
65                  if (dataType.isPresent()) {
66                      dataList(obj, dataType.get());
67                  } else {
68                      throw new Exception("not find
    data type define in data segment at line " +
    lexer.getStatus().getLineIndex());
69                  }
70              }
71              LOGGER.info("exit data define");
72          }
73
74          protected void moreDataDefine(ObjFile obj)
    throws Exception {
75              LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
76              if
    (wrapper.getToken().equals(Token.EndofContent)) {
77                  return;
78              }
79              LOGGER.info("wrapper token " +
    wrapper.getToken() + " lexeme (" +
    wrapper.getLexeme() + ")");
80              if
    (wrapper.getToken().equals(Token.NewLine)) {
81                  match(Token.NewLine);
82              }
83              wrapper = lexer.lookAheadK(1).get(0);
84              if (wrapper.getToken() !=
    Token.DotString) {
85                  dataDefine(obj);
```

```
86                  moreDataDefine(obj);
87              }
88          }
89
90          protected void dataList(ObjFile obj,
    DataType dataType) throws Exception {
91              data(obj, dataType);
92              moreData(obj, dataType);
93
94          }
95
96          protected void match(String str) throws
    Exception {
97              lexer.getNextToken();
98              if
    (!str.equals(lexer.getStatus().getCurrentLexeme()))
    {
99                  throw new Exception("Expected " +
    str + " at line " +
    lexer.getStatus().getLineIndex() +
100                         ". However, we got " +
    lexer.getStatus().getCurrentLexeme());
101             }
102         }
103
104         protected void match(Token t) throws
    Exception {
105             Token token = lexer.getNextToken();
106             if (!t.equals(token)) {
107                 throw new Exception("Expected token
    " + t + " at line " +
    lexer.getStatus().getLineIndex() +
108                         ". However, we got " +
    token);
109             }
110         }
111
112         /**
113          * For now, only supports 3 kinds of data.
114          * They are string, like '1dfdssasad'.
```

```
115         * Number, like 123.
116         * And dup, like dup 10 (123).
117         * We should support $ and basic arithmetic
    operations here later.
118         *
119         * @param obj      the return object file
120         * @param dataType see Enum dataType
121         * @throws Exception exception
122         */
123        protected void data(ObjFile obj, DataType
    dataType) throws Exception {
124            LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
125            if
    (wrapper.getToken().equals(Token.Quote)) {
126                match(Token.Quote);
127                match(Token.String);
128
    obj.getDataSegment().addData(lexer.getStatus().currentLexeme,
    dataType);
129                match(Token.Quote);
130            } else if
    (wrapper.getToken().equals(Token.String) &&
    "dup".equals(wrapper.getLexeme())) {
131                int currentLocation =
    obj.getDataSegment().getCurrentLocation();
132                match(Token.String);
133                match(Token.Number);
134                int times =
    Integer.valueOf(lexer.getStatus().getCurrentLexeme());
135                LOGGER.info("dup times " + times +
    " current location " + currentLocation);
136                match(Token.LeftParent);
137                dataList(obj, dataType);
138                LOGGER.info("after parse data list
    location " +
    obj.getDataSegment().getCurrentLocation());
139                List<Byte> tmpData =
    obj.getDataSegment().getPortionFrom(currentLocation);
140                for (int i = 1; i < times; i++) {
```

```
141
    obj.getDataSegment().addData(tmpData);
142                 }
143                 match(Token.RightParent);
144             } else {
145                 BigInteger number = expr(obj);
146
    obj.getDataSegment().addData(number, dataType);
147             }
148         }
149
150
151         protected BigInteger expr(ObjFile obj)
    throws Exception {
152             BigInteger term = term(obj);
153             return moreTerm(obj, term);
154         }
155
156         protected BigInteger term(ObjFile obj)
    throws Exception {
157             BigInteger factor = factor(obj);
158             return moreFactor(obj, factor);
159         }
160
161         //negative is not very good here
162         protected BigInteger moreTerm(ObjFile obj,
    BigInteger left) throws Exception {
163             LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
164             if
    (wrapper.getToken().equals(Token.Add)) {
165                 match(Token.Add);
166                 BigInteger right = term(obj);
167                 return moreFactor(obj,
    left.add(right));
168             } else if
    (wrapper.getToken().equals(Token.Sub)) {
169                 match(Token.Sub);
170                 BigInteger right = term(obj);
171                 return moreFactor(obj,
```

```
left.subtract(right));
172             }
173             return left;
174         }
175
176
177         protected BigInteger factor(ObjFile obj)
    throws Exception {
178             LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
179             BigInteger number = null;
180             if
    (wrapper.getToken().equals(Token.Sub)) {
181                 match(Token.Sub);
182                 return factor(obj).negate();
183             } else if
    (wrapper.getToken().equals(Token.LeftParent)) {
184                 match(Token.LeftParent);
185                 number = expr(obj);
186                 match(Token.RightParent);
187                 return number;
188             }
189             wrapper = lexer.lookAheadK(1).get(0);
190             if
    (wrapper.getToken().equals(Token.Number)) {
191                 match(Token.Number);
192                 number = new
    BigInteger(lexer.getStatus().currentLexeme);
193             } else if
    (wrapper.getToken().equals(Token.String)) {
194                 match(Token.String);
195                 final String str =
    lexer.getStatus().getCurrentLexeme();
196                 if ("sizeof".equals(str)) {
197                     match(Token.String);
198                     Optional<DataType> type =
    DataType.of(lexer.getStatus().getCurrentLexeme());
199                     if (!type.isPresent()) {
200                         throw new Exception("not
    find data type define after sizeof operator at
```

```
line " + lexer.getStatus().getLineIndex());
201                     } else {
202                         number =
BigInteger.valueOf(type.get().getSize());
203                     }
204                 } else {
205                     int location =
obj.getDataSegment().getLocationByName(str);
206                     if (location == -1) {
207                         throw new Exception("not
find label " + str + " at line " +
lexer.getStatus().getLineIndex());
208                     } else {
209                         number =
BigInteger.valueOf(location);
210                     }
211                 }
212             } else if
(wrapper.getToken().equals(Token.DollarSign)) {
213                 match(Token.DollarSign);
214                 int location =
obj.getDataSegment().getCurrentLocation();
215                 number =
BigInteger.valueOf(location);
216             }
217             if (number == null) {
218                 throw new Exception(" number is not
defined at line " +
lexer.getStatus().getLineIndex());
219             }
220             return number;
221         }
222
223
224         protected BigInteger moreFactor(ObjFile
obj, BigInteger left) throws Exception {
225             LexemeTokenWrapper wrapper =
lexer.lookAheadK(1).get(0);
226             if
(wrapper.getToken().equals(Token.Mul)) {
```

```
227                 match(Token.Mul);
228                 BigInteger right = factor(obj);
229                 return moreFactor(obj,
    left.multiply(right));
230             } else if
    (wrapper.getToken().equals(Token.Div)) {
231                 match(Token.Div);
232                 BigInteger right = factor(obj);
233                 return moreFactor(obj,
    left.divide(right));
234             }
235             return left;
236         }
237 
238         public void moreData(ObjFile obj, DataType
    dataType) throws Exception {
239             LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
240             if
    (wrapper.getToken().equals(Token.Comma)) {
241                 match(Token.Comma);
242                 data(obj, dataType);
243                 moreData(obj, dataType);
244             }
245         }
246 
247 
248         protected void dataSegment(ObjFile obj)
    throws Exception {
249             LexemeTokenWrapper wrapper =
    lexer.lookAheadK(1).get(0);
250             if
    (wrapper.token.equals(Token.DotString) &&
    ".data".equals(wrapper.getLexeme())) {
251                 lexer.getNextToken();
252             }
253             moreDataDefine(obj);
254         }
255      }
256 
```

3. **Eight-Bit Register**

```
1
2     package Instructions;
3
4     import javax.swing.text.html.Option;
5     import java.util.Map;
6     import java.util.Optional;
7     import java.util.function.Function;
8     import java.util.stream.Collectors;
9     import java.util.stream.Stream;
10
11    /**
12     * All registers contain 8 bits data.
13     */
14    public enum EightBitsRegister implements
    Register {
15        AL("000"),
16        CL("001"),
17        DL("010"),
18        BL("011"),
19        AH("100"),
20        CH("101"),
21        DH("110"),
22        BH("111");
23        private final String registerCode;
24
25        EightBitsRegister(String code) {
26            registerCode = code;
27        }
28
29        @Override
30        public RegisterLength getRegisterLength() {
31            return RegisterLength.EIGHT;
32        }
33
34        @Override
35        public String getRegisterName() {
36            return name().toLowerCase();
37        }
```

```
38
39          @Override
40          public String getRegisterCode() {
41              return registerCode;
42          }
43
44          private static Map<String, Register>
    codeToRegister =
45
    Stream.of(values()).collect(Collectors.toMap(Register::getRegister
    Function.identity()));
46
47          public static Optional<Register> of(String
    registerCode) {
48              return
    Optional.ofNullable(codeToRegister.get(registerCode));
49          }
50
51          private static Map<String, Register>
    nameToRegister =
52
    Stream.of(values()).collect(Collectors.toMap(Register::getRegister
    Function.identity()));
53
54          public static Optional<Register>
    fromName(String name) {
55              return
    Optional.ofNullable(nameToRegister.get(name.toLowerCase()));
56          }
57      }
58
```

4. Operation Code

```
1   package Instructions;
2
3   import common.BitSetUtils;
4
5   import java.util.BitSet;
```

```
6       import java.util.Map;
7       import java.util.Optional;
8       import java.util.function.Function;
9       import java.util.stream.Collectors;
10      import java.util.stream.Stream;
11
12      /**
13       * All Enum types inherit from Op interface.
14       */
15      public enum OpCode implements Op {
16          ADD("000000"),
17          ADC("000001"),
18          SUB("000010"),
19          SBB("000011"),
20          MUL("000100"),
21          DIV("000101"),
22          MOV("000110"),
23          PUSH("000111"),
24          POP("001000"),
25          CBW("001001"),
26          CWD("001010"),
27          BSWAP("001011"),
28          AND("001100"),
29          OR("001101"),
30          XOR("001110"),
31          NOT("001111"),
32          SAL("010000"),
33          SHL("010001"),
34          SAR("010010"),
35          SHR("010011"),
36          CMPSB("010100"),
37          CMPSW("010101"),
38          CMPSD("010110"),
39          LODSB("010111"),
40          LODSW("011000"),
41          LODSD("011001"),
42          STOSB("011010"),
43          STOSW("011011"),
44          STOSD("011100"),
45          REP("011101"),
```

```
46          REPNE("011110"),
47          JMP("011111"),
48          JCC("100000"),
49          JECXZ("100001"),
50          CALL("100010"),
51          RET("100011"),
52          ENTER("100100"),
53          LEAVE("100101"),
54          LOOP("100110"),
55          LOOPE("100111"),
56          LOOPZ("101000"),
57          LOOPNE("101001"),
58          LOOPNZ("101010"),
59          NOP("101011");
60          private final String opcode;
61
62          OpCode( String bits) {
63              assert bits.length() == Op.SIZE;
64              opcode = bits;
65          }
66
67          @Override
68          public String getMemonic() {
69              return name();
70          }
71
72          @Override
73          public BitSet getBits() {
74              BitSet bitSet = null;
75              try{
76                  bitSet =
    BitSetUtils.fromString(opcode);
77              }
78              catch (Exception e){
79                  getOpLogger().info("catch exception
    " + e);
80              }
81              return bitSet;
82          }
83
```

```
84          @Override
85          public  String getOpCode(){
86              return opcode;
87          }
88
89          public static Optional<Op> of(final String
    opcode) {
90              return
    Optional.ofNullable(bitsToOpCode.get(opcode));
91          }
92
93          private static final Map<String, Op>
    memToOpCode = Stream.of(OpCode.values())
94
    .collect(Collectors.toMap(Object::toString,
    Function.identity()));
95
96          private static final Map<String, Op>
    bitsToOpCode = Stream.of(OpCode.values())
97
    .collect(Collectors.toMap(Op::getOpCode,
    Function.identity()));
98
99          public static Optional<Op> fromMem(final
    String mem) {
100             return
    Optional.ofNullable(memToOpCode.get(mem.toUpperCase()));
101         }
102     }
103
```