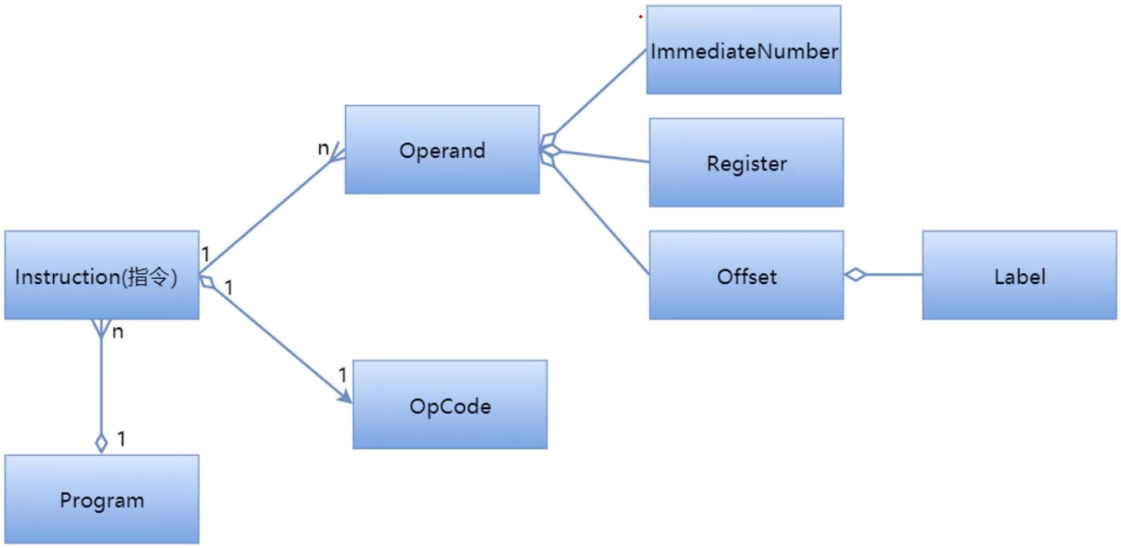
Compiler Experiment : Simple Compiler

Author：软件81金之航

StuId：2183411101

四.指令翻译 Generator以及 虚拟机Virtue Machine执行




```
Program:程序
Instruction:
Operand:操作数
OpCode:操作符
ImmediateNumber:立即数
Register：寄存器
Offset:偏移量
Label:标签 eg:L0
```

操作码：

```
public static final OpCode ADD = new OpCode(AddressingType.REGISTER, "ADD",
(byte) 0x01);
public static final OpCode SUB = new OpCode(AddressingType.REGISTER, "SUB",
(byte) 0x02);
public static final OpCode MULT = new OpCode(AddressingType.REGISTER,
"MULT", (byte) 0x03);
public static final OpCode ADDI = new OpCode(AddressingType.IMMEDIATE,
"ADDI", (byte) 0x05);
public static final OpCode SUBI = new OpCode(AddressingType.IMMEDIATE,
"SUBI", (byte) 0x06);
public static final OpCode MULTI = new OpCode(AddressingType.IMMEDIATE,
"MULTI", (byte) 0x07);
public static final OpCode MFLO = new OpCode(AddressingType.REGISTER,
"MFLO", (byte) 0x08);
```

```
public static final OpCode EQ = new OpCode(AddressingType.REGISTER, "EQ", (byte) 0x09);
public static final OpCode BNE = new OpCode(AddressingType.OFFSET, "BNE", (byte) 0x15);
public static final OpCode SW = new OpCode(AddressingType.OFFSET, "SW", (byte) 0x10);
public static final OpCode LW = new OpCode(AddressingType.OFFSET, "LW", (byte) 0x11);
public static final OpCode JUMP = new OpCode(AddressingType.JUMP, "JUMP", (byte) 0x20);
public static final OpCode JR = new OpCode(AddressingType.JUMP, "JR", (byte) 0x21);
public static final OpCode RETURN = new OpCode(AddressingType.JUMP, "RETURN", (byte) 0x22);

eg1:BNE S0,S1,L0  //compare S0 and S1, if equals jump to L100, else next
    0x15    0x0a    0x0b    100
    Opcode  S0      S1      Label
Len:6       5       5       16

eg2:ADD S0,S1,S0  //and S0 and S1, load into S0
    0x15    0x0a    0x0b    0x0b    100
    Opcode  S0      S1      S1      Label
Len:6       5       5       5       11
```

寄存器：

```
public static final Register ZERO = new Register("ZERO", (byte) 1);
public static final Register PC = new Register("PC", (byte) 2);
public static final Register SP = new Register("SP", (byte) 3);
public static final Register STATIC = new Register("STATIC", (byte) 4);
public static final Register RA = new Register("RA", (byte)5);
public static final Register S0 = new Register("S0", (byte) 10);
public static final Register S1 = new Register("S1", (byte) 11);
public static final Register S2 = new Register("S2", (byte) 12);
public static final Register LO = new Register("LO", (byte) 20);
```

运算：

```
~ 取反
| 位或
^ 异或
& 位与
<< 左移
>> 右移
>>> 无符号左移
<<< 无符号右移
```

1.OpCodeGen

生成相应操作码：

```
public class OpCodeGen {

```

```
    public OpCodeProgram gen(TAProgram taProgram){
        var program = new OpCodeProgram();
        var taInstructions = taProgram.getInstructions();
        var labelHash = new Hashtable<String, Integer>();

        for(var taInstruction : taInstructions) {
            program.addComment(taInstruction.toString());
            switch(taInstruction.getType()) {
                case ASSIGN:
                    genCopy(program, taInstruction);
                    break;
                case GOTO:
                    genGoto(program, taInstruction);
                    break;
                case CALL:
                    genCall(program, taInstruction);
                    break;
                case PARAM:
                    genPass(program, taInstruction);
                    break;
                case SP:
                    genSp(program, taInstruction);
                    break;
                case LABEL:
                    if(taInstruction.getArg2() != null &&
taInstruction.getArg2().equals("main")) {
                        program.setEntry(program.instructions.size());
                    }
                    labelHash.put((String) taInstruction.getArg1(),
program.instructions.size());
                    break;
                case RETURN:
                    genReturn(program, taInstruction);
                    break;
                case FUNC_BEGIN:
                    genFuncBegin(program, taInstruction);
                    break;
                case IF: {
                    genIf(program, taInstruction);
                    break;
                }
                default:
                    throw new NotImplementedException("Unknown type:" +
taInstruction.getType());
            }

        }
        this.relabel(program, labelHash);
        return program;
    }

    private void genIf(OpCodeProgram program, TAInstruction instruction) {
//        var exprAddr = (Symbol)instruction.getArg1();
        var label = instruction.getArg2();
        program.add(Instruction.bne(Register.S2, Register.ZERO, (String)
label));
    }

```

```
    private void genReturn(OpCodeProgram program, TAInstruction taInstruction) {
        var ret = (Symbol)taInstruction.getArg1();
        if(ret != null) {
            program.add(Instruction.loadToRegister(Register.S0, ret));
        }
        program.add(Instruction.offsetInstruction(
                OpCode.SW ,Register.S0, Register.SP, new Offset(1)
        ));

        var i = new Instruction(OpCode.RETURN);
        program.add(i);
    }

    /**
     * 重新计算Label的偏移量
     * @param program
     * @param labelHash
     */
    private void relabel(OpCodeProgram program, Hashtable<String, Integer> labelHash){
        program.instructions.forEach(instruction -> {
            if(instruction.getOpCode() == OpCode.JUMP || instruction.getOpCode() == OpCode.BNE)             {
                var idx = instruction.getOpCode()==OpCode.BNE?2 : 0;
                var labelOperand = (Label)instruction.opList.get(idx);
                var label = labelOperand.getLabel();
                var offset = labelHash.get(label);
                labelOperand.setOffset(offset);
            }
        });

    }

    private void genSp(OpCodeProgram program, TAInstruction taInstruction) {
        var offset = (int)taInstruction.getArg1();
        if(offset > 0) {
            program.add(Instruction.immediate(OpCode.ADDI, Register.SP,
                    new ImmediateNumber(offset)));
        }
        else {
            program.add(Instruction.immediate(OpCode.SUBI, Register.SP,
                    new ImmediateNumber(-offset)));
        }
    }

    private void genPass(OpCodeProgram program, TAInstruction taInstruction) {
        var arg1 = (Symbol)taInstruction.getArg1();
        var no = (int)taInstruction.getArg2();
        program.add(Instruction.loadToRegister(Register.S0, arg1));
        // PASS a
        program.add(Instruction.offsetInstruction(OpCode.SW, Register.S0, Register.SP,
                new Offset(-(no))));
    }
```

```
108
109     void genFuncBegin(OpCodeProgram program, TAInstruction ta) {
110         var i = Instruction.offsetInstruction(OpCode.SW, Register.RA,
    Register.SP, new Offset(0));
111         program.add(i);
112     }
113
114     void genCall(OpCodeProgram program, TAInstruction ta){
115         var label = (Symbol)ta.getArg1();
116         var i = new Instruction(OpCode.JR);  //RA <- PC
117         i.opList.add(new Label(label.getLabel()));
118         program.add(i);
119
120     }
121
122     void genGoto(OpCodeProgram program, TAInstruction ta) {
123         var label = (String)ta.getArg1();
124         var i = new Instruction(OpCode.JUMP);
125         // label对应的位置在relabel阶段计算
126         i.opList.add(new Label(label));
127         program.add(i);
128
129     }
130
131     void genCopy(OpCodeProgram program, TAInstruction ta) {
132         // result = arg1 op arg2
133         // result = arg1
134         var result = ta.getResult();
135         var op = ta.getOp();
136         var arg1 = (Symbol)ta.getArg1();
137         var arg2 = (Symbol)ta.getArg2();
138         if(arg2 == null) {
139             program.add(Instruction.loadToRegister(Register.S0, arg1));
140             program.add(Instruction.saveToMemory(Register.S0, result));
141         } else {
142             program.add(Instruction.loadToRegister(Register.S0, arg1));
143             program.add(Instruction.loadToRegister(Register.S1, arg2));
144
145             switch (op) {
146                 case "+":
147                     program.add(Instruction.register(OpCode.ADD,
    Register.S2, Register.S0, Register.S1));
148                     break;
149                 case "-":
150                     program.add(Instruction.register(OpCode.SUB,
    Register.S2, Register.S0, Register.S1));
151                     break;
152                 case "*":
153                     program.add(Instruction.register(OpCode.MULT,
    Register.S0, Register.S1,null));
154                     program.add(Instruction.register(OpCode.MFLO,
    Register.S2, null, null));
155                     break;
156                 case "==" :
157                     program.add(Instruction.register(OpCode.EQ,
    Register.S2, Register.S1, Register.S0));
158                     break;
159             }
```

```
            program.add(Instruction.saveToMemory(Register.S2, result));
        }
    }
}
```

2.Instruction

指令的编码与解码

```
public class Instruction {

    private static final int MASK_OPCODE = 0xfc000000;  //1111 1100
    private static final int MASK_R0 = 0x03e00000;
    private static final int MASK_R1 = 0x001f0000;
    private static final int MASK_R2 = 0x0000f800;
    private static final int MASK_OFFSET0 = 0x03ffffff;
    private static final int MASK_OFFSET1 = 0x001fffff;
    private static final int MASK_OFFSET2 = 0x000007ff;
    private OpCode code;
    ArrayList<Operand> opList = new ArrayList<>();

    public Instruction(OpCode code) {
        this.code = code;
    }

    public static Instruction jump(OpCode code, int offset) {
        var i = new Instruction(code);
        i.opList.add(new Offset(offset));
        return i;
    }

    public static Instruction offsetInstruction(
            OpCode code,
            Register r1,
            Register r2,
            Offset offset) {
        var i = new Instruction(code);

        i.opList.add(r1);
        i.opList.add(r2);
        i.opList.add(offset);
        return i;

    }

    public static Instruction loadToRegister(Register target, Symbol arg) {
        // 转成整数，目前只支持整数
        if (arg.getType() == SymbolType.ADDRESS_SYMBOL) {
            return offsetInstruction(OpCode.LW, target, Register.SP, new
Offset(-arg.getOffset()));
        } else if (arg.getType() == SymbolType.IMMEDIATE_SYMBOL) {
            return offsetInstruction(OpCode.LW, target, Register.STATIC,
new Offset(arg.getOffset()));
        }
```

```
44          throw new NotImplementedException("Cannot load type " +
arg.getType() + " symbol to register");
45      }
46
47      public static Instruction saveToMemory(Register source, Symbol arg) {
48          return offsetInstruction(OpCode.SW, source, Register.SP, new
Offset(-arg.getOffset()));
49      }
50
51      public static Instruction bne(Register a, Register b, String label) {
52          var i = new Instruction(OpCode.BNE);
53          i.opList.add(a);
54          i.opList.add(b);
55          i.opList.add(new Label(label));
56          return i;
57      }
58
59      public static Instruction register(OpCode code, Register a, Register b,
Register c) {
60          var i = new Instruction(code);
61          i.opList.add(a);
62          if (b != null) {
63              i.opList.add(b);
64          }
65          if (c != null) {
66              i.opList.add(c);
67          }
68          return i;
69      }
70
71      public static Instruction immediate(OpCode code, Register r,
ImmediateNumber number) {
72          var i = new Instruction(code);
73          i.opList.add(r);
74          i.opList.add(number);
75          return i;
76      }
77
78      public OpCode getOpCode() {
79          return this.code;
80      }
81
82      @Override
83      public String toString() {
84          String s = this.code.toString();
85
86          var prts = new ArrayList<String>();
87          for (var op : this.opList) {
88              prts.add(op.toString());
89          }
90          return s + " " + StringUtils.join(prts, " ");
91      }
92
93      public static Instruction fromByCode(int code) throws
GeneratorException {
94          byte byteOpcode = (byte) ((code & MASK_OPCODE) >>> 26);
95          var opcode = OpCode.fromByte(byteOpcode);
96          var i = new Instruction(opcode);
```

```
97
98          switch (opcode.getType()) {
99              case IMMEDIATE: {
100                 var reg = (code & MASK_R0) >> 21;
101                 var number = code & MASK_OFFSET1;
102                 i.opList.add(Register.fromAddr(reg));
103                 i.opList.add(new ImmediateNumber((int) number));
104                 break;
105             }
106             case REGISTER: {
107                 var r1Addr = (code & MASK_R0) >> 21;
108                 var r2Addr = (code & MASK_R1) >> 16;
109                 var r3Addr = (code & MASK_R2) >> 11;
110                 var r1 = Register.fromAddr(r1Addr);
111
112                 Register r2 = null;
113                 if (r2Addr != 0) {
114                     r2 = Register.fromAddr(r2Addr);
115                 }
116                 Register r3 = null;
117                 if (r3Addr != 0) {
118                     r3 = Register.fromAddr(r3Addr);
119                 }
120                 i.opList.add(r1);
121                 if (r2 != null) {
122                     i.opList.add(r2);
123                 }
124                 if (r3 != null) {
125                     i.opList.add(r3);
126                 }
127                 break;
128             }
129             case JUMP: {
130                 var offset = code & MASK_OFFSET0;
131                 i.opList.add(Offset.decodeOffset(offset));
132                 break;
133             }
134             case OFFSET: {
135                 var r1Addr = (code & MASK_R0) >> 21;
136                 var r2Addr = (code & MASK_R1) >> 16;
137                 var offset = code & MASK_OFFSET2;
138                 i.opList.add(Register.fromAddr(r1Addr));
139                 i.opList.add(Register.fromAddr(r2Addr));
140                 i.opList.add(Offset.decodeOffset(offset));
141                 break;
142             }
143         }
144         return i;
145     }
146
147     public Integer toByteCode() {
148         int code = 0;
149         //Opcode -> Int
150         //0x01
151         //|--opcode--|----|----|
152         int x = this.code.getValue();
153         code |= x << 26;
154         switch (this.code.getType()) {
```

```
            case IMMEDIATE: {
                //|--opcode--|--r0--|--immediate number--|
                var r0 = (Register) this.opList.get(0);

                code |= r0.getAddr() << 21;
                code |= ((ImmediateNumber) this.opList.get(1)).getValue();
                return code;
            }
            case REGISTER: {
                //|--opcode 6--|--r0 5--|--r1 5--|--r2 5--|--null--|
                var r1 = (Register) this.opList.get(0);
                code |= r1.getAddr() << 21;
                if (this.opList.size() > 1) {
                    code |= ((Register) this.opList.get(1)).getAddr() <<
16;
                    if (this.opList.size() > 2) {
                        var r2 = ((Register) this.opList.get(2)).getAddr();
                        code |= r2 << 11;
                    }
                }
                break;
            }
            case JUMP:
                if (this.opList.size() > 0) {
                    code |= ((Offset)
this.opList.get(0)).getEncodedOffset();
                }
                break;

            case OFFSET:
                var r1 = (Register) this.opList.get(0);
                var r2 = (Register) this.opList.get(1);
                var offset = (Offset) this.opList.get(2);
                //|--code--|--r1--|--r2--|--offset--|
                code |= r1.getAddr() << 21;
                code |= r2.getAddr() << 16;
                code |= offset.getEncodedOffset();
                break;
        }
        return code;
    }

    public Operand getOperand(int index) {
        return this.opList.get(index);
    }
}
```

3.VirtueMachine

模拟虚拟机执行指令：

```
public class VirtualMachine {

    int registers[] = new int[31];
    int[] memory = new int[4096];
    /*
```

```
     * 静态区
     * 程序区
     * 堆
     * 空闲区
     * 栈
     * */
    int endProgramSection = 0;
    int startProgram = 0;

    /**
     * 初始化
     */
    public VirtualMachine(ArrayList<Integer> staticArea, ArrayList<Integer>
opcodes, Integer entry) {

        int i = 0;
        /**
         * 静态区
         */
        for(; i < staticArea.size(); i++) {
            memory[i] = staticArea.get(i);
        }

        /**
         * 程序区
         */
        int j = i;
        startProgram = i;
        int mainStart = entry + i;
        for(; i  < opcodes.size() + j; i++) {
            memory[i] = opcodes.get(i - j);
        }
        /*
        * f(){}
        * main(){}
        * ...
        * SP - ?
        * CALL MAIN
        * SP + ?
        * */
        registers[Register.PC.getAddr()] = i-3;
        endProgramSection = i;

        /**
         * 栈指针
         */
        registers[Register.SP.getAddr()] = 4095;
    }

    private int fetch() {
        var PC = registers[Register.PC.getAddr()];
        return memory[(int) PC];
    }

    private Instruction decode(int code) throws GeneratorException {
        return Instruction.fromByCode(code);
    }

```

```
63      private void exec(Instruction instruction) {
64
65          byte code = instruction.getOpCode().getValue();
66          System.out.println("exec:" + instruction);
67
68          switch (code) {
69              case 0x01: { // ADD
70                  var r0 = (Register)instruction.getOperand(0);
71                  var r1 = (Register)instruction.getOperand(1);
72                  var r2 = (Register)instruction.getOperand(2);
73                  registers[r0.getAddr()] = registers[r1.getAddr()] +
    registers[r2.getAddr()];
74                  break;
75              }
76              case 0x09:
77              case 0x02: { // SUB
78                  var r0 = (Register) instruction.getOperand(0);
79                  var r1 = (Register) instruction.getOperand(1);
80                  var r2 = (Register) instruction.getOperand(2);
81                  registers[r0.getAddr()] = registers[r1.getAddr()] -
    registers[r2.getAddr()];
82                  break;
83              }
84              case 0x03: { // MULT
85                  var r0 = (Register) instruction.getOperand(0);
86                  var r1 = (Register) instruction.getOperand(1);
87                  registers[Register.LO.getAddr()] = registers[r0.getAddr()]
    * registers[r1.getAddr()];
88                  break;
89              }
90              case 0x05: { // ADDI
91                  var r0 = (Register) instruction.getOperand(0);
92                  var r1 = (ImmediateNumber) instruction.getOperand(1);
93                  registers[r0.getAddr()] += r1.getValue();
94                  break;
95              }
96              case 0x06: { // SUBI
97                  var r0 = (Register) instruction.getOperand(0);
98                  var r1 = (ImmediateNumber) instruction.getOperand(1);
99                  registers[r0.getAddr()] -= r1.getValue();
100                 break;
101             }
102 //            case 0x07: // MULTI
103 //                break;
104             case 0x08: { // MFLO
105                 var r0 = (Register) instruction.getOperand(0);
106                 registers[r0.getAddr()] = registers[Register.LO.getAddr()];
107                 break;
108             }
109             case 0x10: { // SW
110                 var r0 = (Register) instruction.getOperand(0);
111                 var r1 = (Register) instruction.getOperand(1);
112                 var offset = (Offset) instruction.getOperand(2);
113                 var R1VAL = registers[r1.getAddr()];
114                 memory[(int) (R1VAL + offset.getOffset())] =
    registers[r0.getAddr()];
115                 break;
116             }
```

```
117             case 0x11: { //LW
118                 var r0 = (Register) instruction.getOperand(0);
119                 var r1 = (Register) instruction.getOperand(1);
120                 var offset = (Offset) instruction.getOperand(2);
121                 var R1VAL = registers[r1.getAddr()];
122                 registers[r0.getAddr()] = memory[(int) (R1VAL +
offset.getOffset())];
123                 break;
124             }
125             case 0x15 : { // BNE
126                 var r0 = (Register)instruction.getOperand(0);
127                 var r1 = (Register)instruction.getOperand(1);
128                 var offset = (Offset)instruction.getOperand(2);
129                 if(registers[r0.getAddr()] != registers[r1.getAddr()]) {
130                     registers[Register.PC.getAddr()] = offset.getOffset() +
startProgram - 1;
131                 }
132                 break;
133             }
134             case 0x20 : { // JUMP
135                 var r0 = (Offset) instruction.getOperand(0);
136                 registers[Register.PC.getAddr()] = r0.getOffset() +
startProgram - 1;
137                 break;
138             }
139             case 0x21: { // JR
140                 var r0 = (Offset) instruction.getOperand(0);
141                 // 将返回地址存入ra
142                 registers[Register.RA.getAddr()] =
registers[Register.PC.getAddr()];
143                 registers[Register.PC.getAddr()] = r0.getOffset() +
startProgram - 1;
144                 break;
145             }
146             case 0x22 : { // RETURN
147                 if(instruction.getOperand(0) != null) {
148                     // match返回值
149                 }
150                 var spVal = registers[Register.SP.getAddr()];
151                 registers[Register.PC.getAddr()] = memory[spVal];
152                 break;
153             }
154         }
155     }
156
157     public boolean runOneStep() throws GeneratorException {
158         var code = fetch();
159         var instruction = decode(code);
160         exec(instruction);
161         registers[Register.PC.getAddr()] += 1;
162         System.out.println(registers[Register.PC.getAddr()] + "|" +
endProgramSection);
163         return registers[Register.PC.getAddr()] < endProgramSection;
164     }
165
166     public void run() throws GeneratorException {
167         // 模拟CPU循环
168         //   fetch: 获取指令
```

```
169             //   decode: 解码
170             //   exec: 执行
171             //   PC++
172             while(runOneStep());
173         }
174     }
```

4.Example

虚拟机执行测试：

输入语句

```
1  func main() int { var a = 2*3+4 \n return \n }
```

具体实现：

```
1   @Test
2       public void calcExpr() throws LexicalException, ParseException,
    GeneratorException {
3           var source = "func main() int { var a = 2*3+4 \n return \n }";
4           //call main
5           var astNode = Parser.parse(source);
6           var translator = new Translator();
7           var taProgram = translator.translate(astNode);
8           var gen = new OpCodeGen();
9           var program = gen.gen(taProgram);
10          var statics = program.getStaticArea(taProgram);
11          var entry = program.getEntry();
12          var opcodes = program.toByteCodes();
13
14          var vm = new VirtualMachine(statics, opcodes, entry);
15          vm.run();
16          System.out.println("SP:" + vm.getRegisters()
    [Register.SP.getAddr()]);
17      }
```

输出Output：

```
1  //func main() int { var a = 2*3+4 \n return \n }
2
3  exec:SUBI SP 1  //生成call main(),栈指针-1
4  exec:JR 0  //跳转到main()，JR和J的区别就是是否保存当前栈指针
5  exec:SW RA SP 0  //保存当前栈指针
6  exec:LW S0 STATIC 0  //从静态符号表中取出offset=0的值，这里也就是2，放入S0
7  exec:LW S1 STATIC 1  //从静态符号表中取出offset=1的值，这里也就是3，放入S1
8  exec:MULT S0 S1  //计算2*3
9  exec:MFLO S2  //低32位存入S2
10 exec:SW S2 SP -2  //所得结果从S2写入内存
11 exec:LW S0 SP -2  //从内存中取出到S0，这里是一步多余的操作，没有进行代码优化
12 exec:LW S1 STATIC 2  //从静态符号表中取出offset=1的值，这里也就是4，放入S1
13 exec:ADD S2 S0 S1  //计算6+4存入S2
14 exec:SW S2 SP -3  //所得结果从S2写入内存
15 exec:LW S0 SP -3  //从内存中取出到S0，这里是一步多余的操作，没有进行代码优化，a=10
```

```
exec:SW S0 SP -1  //写入内存
exec:SW S0 SP 1  //写入调用call main()
exec:RETURN 0  //返回
exec:ADDI SP 1  //栈指针+1
SP:4095  //结束
```

总结

```
这次编译原理实验有许多收获，主要有：
1.软件体系结构上的收获：坚持写测试用例，保证底层的健壮，这样以后可以避免很多bug，或者bug不知道从何处调起，一个健壮的底层可以让我少走很多弯路。
2.算法和数据结构：算法和数据结构基础知识一定要足够扎实，这样才能在coding的过程中得心应手，用少量的代码去实现高效的工作。
3.计算级底层原理：计算机基础知识一定要足够熟悉，比如这次的虚拟机是从github上找的，有一部内容没看懂，还需要进一步学习。编译器的设计其实用到了许多操作系统和计算机组成原理的知识，自己的基础知识没有打牢，这次实验一些底层原理让我颇为困扰，之后会进一步学习。
4.高效使用工具，这次实验代码量大概在5k行左右，熟练的使用Lombok，tabnine，idea alt insert，git开发等提高了不少效率，代码行数大概在4k行左右。
5.为什么要做编译器？在实际编码过程中，需要非常得有耐心，细心，考虑各种文法，分析方式，优化手段，写好测试用例等等。一个良好的编译器需要精心打磨，不断优化升级，需要寻找相关书籍，系统地学习一遍知识体系。它对基础知识的积累与掌握，对编程语言的认识与理解，对框架的学习与运用，对以后的发展道路，有很大帮助。
```

全程使用Github开发，代码可以在https://github.com/TerenceStark/Compiler找到：

代码提交情况：