```java
/*
    Adder
    -adds the two integer values it holds, makes the result available to be
    obtained by another object
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class Adder {

    //Default Constructor
    Adder(){
    }

    //Add function
    Integer add(Integer val1, Integer val2){
        return val1 + val2;
    }
}

```

```
 1 /*
 2     Address Lines: pass memory addresses to one
another over the address bus
 3  */
 4 package simple_computer_simulation;
 5
 6 /**
 7  *
 8  * @author bhitt
 9  */
10 public class AddressLines {
11     //Properties
12     private Integer val;
13
14     //Default Constructor
15     AddressLines(){
16         val=0;
17     }
18
19     //Mutator
20     void set(Integer value){
21         val = value;
22     }
23
24     //Accessor
25     Integer get(){
26         return val;
27     }
28 }
29
```

```java
/*
    Complementer
    -changes the sign of the value it holds
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class Complementer {

    //Default Constructor
    Complementer(){
    }

    //complement function
    Integer complement(Integer value){
        value *=-1;
        return value;
    }

}

```

```
 1 /*
 2     Control Lines: used to send signals to
coordinate
 3     and manage the activities of the motherboard
components
 4     Read  -> 0
 5     Write -> 1
 6  */
 7 package simple_computer_simulation;
 8
 9 /**
10  *
11  * @author bhitt
12  */
13 public class ControlLines {
14     //Properties
15     private Integer val;
16
17     //Default Constructor
18     ControlLines(){
19         val=0;
20     }
21
22     //Mutator
23     Integer set(Integer value){
24         //set the signal to read or write
25         val = value;
26         return val;
27     }
28
29     //Accessor
30     Integer get(){
31         return val;
32     }
```

```
33
34 }
35
```

```java
/*
    Data lines: data transferred between peripherals
    , memory, and the CPU
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class DataLines {
    //Properties
    private Integer val;

    //Default Constructor
    DataLines(){
        val=0;
    }

    //Mutator
    void set(Integer value){
        val = value;
    }

    //Accessor
    Integer get(){
        return val;
    }
}

```

```java
/*
    Instruction Register: holds the current instruction code
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class InstructionRegister {
    //Properties
    private Integer value;
    //Default Constructor
    InstructionRegister(){
        value=0;
    }
    //Accessor
    Integer getVal(){
        return value;
    }
    //Mutator
    void setVal(Integer val){
        value = val;
    }
}

```

```java
/*
    Industrial Standard Architecture
 */
package simple_computer_simulation;

import java.time.LocalDateTime;
import java.util.List;

/**
 *
 * @author bhitt
 */
public class ISA {
    //Properties
    private Register R0;
    private Register R1;
    private Register R2;
    private Register R3;
    private Adder adder;
    private Complementer complementer;
    private Printer printer;
    private Reader reader;
    private AddressLines addressLines;
    private DataLines dataLines;
    private ControlLines controlLines;
    private MemoryControl memoryControl;
    private MemoryAddressRegister mAR;
    private MemoryDataRegister mDR;
    private Status status;
    private ProgramCounter programCounter;
    private InstructionRegister
instructionRegister;

    //Default Constructor
```

```
34      ISA(){
35          build();
36      }
37
38      //build method : instantiates necessary
components
39      void build(){
40          //instantiate four registers
41          R0 = new Register();
42          R1 = new Register();
43          R2 = new Register();
44          R3 = new Register();
45          //Instantiate components
46          adder = new Adder();
47          complementer = new Complementer();
48          printer = new Printer();
49          reader = new Reader();
50          addressLines = new AddressLines();
51          dataLines = new DataLines();
52          controlLines = new ControlLines();
53          status = new Status();
54          memoryControl = new MemoryControl();
55          mAR = new MemoryAddressRegister();
56          mDR = new MemoryDataRegister();
57          status = new Status();
58          programCounter = new ProgramCounter();
59          instructionRegister = new
InstructionRegister();
60          //set status running to true
61          status.setRunning(true);
62      }
63
64      //load program memory
65      void loadProg(Integer fW, Integer fI,
```

```
List<Integer> instructions){
 66          //set the program counter to the first
instruction address
 67          programCounter.setCounter(fI);
 68          //load program into memory starting at the
first word
 69          for(Integer number: instructions){
 70              memoryControl.setMemory(fW, number);
 71              fW++;
 72          }
 73      }
 74
 75      //run method : simulates the program execution
 76      void run(){
 77          while(status.getRunning()){
 78              fetch();
 79              adjustPC();
 80              execute();
 81          }
 82
 83         //final display information
 84         System.out.println("Branden Hitt " +
LocalDateTime.now());
 85      }
 86
 87      void fetch(){
 88          //get an address from the PC
 89          //obtain the instruction from memory
 90          //deliver the instruction to the
instruction register
 91          instructionRegister.setVal(memoryControl.
getMemory(programCounter.getCounter()));
 92          //trace stuff
 93          System.out.println("Starting Location:
```

```
"+programCounter.getCounter()+"   OPCode:"
+instructionRegister.getVal());
 94     }
 95
 96     void adjustPC(){
 97         //change the program counter to its new
value
 98         programCounter.setCounter(programCounter.
getCounter()+1);
 99     }
100
101     void execute(){
102         //variables
103         Integer op1,op2,op3;
104         //get the instruction from the the
instruction register
105         Integer code = instructionRegister.
getVal();
106 //         System.out.println("---------------");
107 //         System.out.println("code:"+code);
108 //         System.out.println("program counter:"
+programCounter.getCounter());
109         //decode the instruction
110         //call the appropriate method to carry out
the instruction
111         if(code==110){
112             //get three operands
113             op1 = memoryControl.getMemory
(programCounter.getCounter());
114             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
115             op3 = memoryControl.getMemory
(programCounter.getCounter()+2);
116             //increment program counter
```

```
117             programCounter.setCounter
(programCounter.getCounter()+3);
118             //call correct version of add
119             ADD(op1,op2,op3);
120         }else if(code==120){
121             //get three operands
122             op1 = memoryControl.getMemory
(programCounter.getCounter());
123             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
124             op3 = memoryControl.getMemory
(programCounter.getCounter()+2);
125             //increment program counter
126             programCounter.setCounter
(programCounter.getCounter()+3);
127             //call subInstruction
128             SUB(op1,op2,op3);
129         }else if(code==160){
130             //get operand
131             op1 = memoryControl.getMemory
(programCounter.getCounter());
132             //increment program counter
133             programCounter.setCounter
(programCounter.getCounter()+1);
134             //call Dec
135             if(op1==0){
136                 DEC(R0);
137             }else if(op1==1){
138                 DEC(R1);
139             }else if(op1==2){
140                 DEC(R2);
141             }else if(op1==3){
142                 DEC(R3);
143             }else{
```

```
144                     HALT();
145                 }
146             }else if(code==440){
147                 //get operand
148                 op1 = memoryControl.getMemory
(programCounter.getCounter());
149                 //increment program counter
150                 programCounter.setCounter
(programCounter.getCounter()+1);
151                 //call BRNZ
152                 BRNZ(op1);
153             }else if(code==810){
154                 //call READ
155                 readInstruction();
156             }else if(code==820){
157                 //call Print
158                 printInstruction();
159             }else if(code==000){
160                //call NOOP
161               NOOP();
162             }else if(code==999){
163                 //call HALT
164                 HALT();
165             }else if(code==510){
166                 //get two operands
167                 op1 = memoryControl.getMemory
(programCounter.getCounter());
168                 op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
169                 //increment program counter
170                 programCounter.setCounter
(programCounter.getCounter()+2);
171                 //call MOVE
172                 MOVE(op1,op2);
```

```
173         }else if(code==610){
174             //get two operands
175             op1 = memoryControl.getMemory
(programCounter.getCounter());
176             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
177             //Load Absolute - use address parameter
with no modifications
178             //increment program counter
179             programCounter.setCounter
(programCounter.getCounter()+2);
180             //call load for correct register
181             if(op1==0) LOAD(R0,readMemory(op2));
182             else if(op1==1) LOAD(R1,readMemory
(op2));
183             else if(op1==2) LOAD(R2,readMemory
(op2));
184             else if(op1==3) LOAD(R3,readMemory
(op2));
185             else HALT();
186         }else if(code==620){
187             //get two operands
188             op1 = memoryControl.getMemory
(programCounter.getCounter());
189             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
190             //increment program counter
191             programCounter.setCounter
(programCounter.getCounter()+2);
192             //LOAD register indirect - get value
from register
193             Integer temp = 0;
194             if(op2==0) temp = R0.getVal();
195             else if(op2==1) temp = R1.getVal();
```

```
196             else if(op2==2) temp = R2.getVal();
197             else if(op2==3) temp = R3.getVal();
198             else HALT();
199             //choose correct destination
200             //call load for correct register
201             if(op1==0) LOAD(R0,temp);
202             else if(op1==1) LOAD(R1,temp);
203             else if(op1==2) LOAD(R2,temp);
204             else if(op1==3) LOAD(R3,temp);
205             else HALT();
206         }else if(code==630){
207             //get two operands
208             op1 = memoryControl.getMemory
(programCounter.getCounter());
209             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
210             //increment program counter
211             programCounter.setCounter
(programCounter.getCounter()+2);
212             //LOAD auto increment register indirect
- get value from register and increment
213             Integer temp = 0;
214             if(op2==0) temp = R0.getVal()+1;
215             else if(op2==1) temp = R1.getVal()+1;
216             else if(op2==2) temp = R2.getVal()+1;
217             else if(op2==3) temp = R3.getVal()+1;
218             else HALT();
219             //choose correct destination
220             //call load for correct register
221             if(op1==0) LOAD(R0,temp);
222             else if(op1==1) LOAD(R1,temp);
223             else if(op1==2) LOAD(R2,temp);
224             else if(op1==3) LOAD(R3,temp);
225             else HALT();
```

```
226         }else if(code==640){
227             //get two operands
228             op1 = memoryControl.getMemory
(programCounter.getCounter());
229             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
230             //increment program counter
231             programCounter.setCounter
(programCounter.getCounter()+2);
232             //Load immediate
233             //call load for correct register
234             if(op1==0) LOAD(R0,op2);
235             else if(op1==1) LOAD(R1,op2);
236             else if(op1==2) LOAD(R2,op2);
237             else if(op1==3) LOAD(R3,op2);
238             else HALT();
239         }else if(code==710){
240             //get two operands
241             op1 = memoryControl.getMemory
(programCounter.getCounter());
242             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
243             //increment program counter
244             programCounter.setCounter
(programCounter.getCounter()+2);
245             //call Store on correct reg
246             if(op1==0) STORE(R0,op2);
247             else if(op1==1) STORE(R1,op2);
248             else if(op1==2) STORE(R2,op2);
249             else if(op1==3) STORE(R3,op2);
250             else HALT();
251         }else if(code==720){
252             //get two operands
253             op1 = memoryControl.getMemory
```

```
(programCounter.getCounter());
254             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
255             //increment program counter
256             programCounter.setCounter
(programCounter.getCounter()+2);
257             //get the correct address
258             Integer add=0;
259             if(op2==0) add=R0.getVal();
260             else if(op2==1) add=R1.getVal();
261             else if(op2==2) add=R2.getVal();
262             else if(op2==3) add=R3.getVal();
263             else HALT();
264             //Store the source at the address
265             if(op1==0) STORE(R0,add);
266             else if(op1==1) STORE(R1,add);
267             else if(op1==2) STORE(R2,add);
268             else if(op1==3) STORE(R3,add);
269             else HALT();
270         }else if(code==730){
271             //get two operands
272             op1 = memoryControl.getMemory
(programCounter.getCounter());
273             op2 = memoryControl.getMemory
(programCounter.getCounter()+1);
274             //increment program counter
275             programCounter.setCounter
(programCounter.getCounter()+2);
276             //get the correct address
277             Integer add=0;
278             if(op2==0) add=R0.getVal()+1;
279             else if(op2==1) add=R1.getVal()+1;
280             else if(op2==2) add=R2.getVal()+1;
281             else if(op2==3) add=R3.getVal()+1;
```

```
282                 else HALT();
283                 //Store the source at the address
284                 if(op1==0) STORE(R0,add);
285                 else if(op1==1) STORE(R1,add);
286                 else if(op1==2) STORE(R2,add);
287                 else if(op1==3) STORE(R3,add);
288                 else HALT();
289             }else{
290                 //if the instruction is invalid, dump
memory and halt
291                 HALT();
292             }
293 
294         }
295 
296         //--------INSTRUCTION SET--------//
297 
298         //Read instruction
299         //-reads an integer from the keyboard and
stores it in R0
300         void readInstruction(){
301             //put read signal on the control lines
302             controlLines.set(0);
303             //read in through reader
304             reader.setBuffer();
305             //throw read value onto bus
306             dataLines.set(reader.getOutput());
307             //throw bus value onto register zero
308             R0.setVal(dataLines.get());
309         }
310 
311         //Print instruction
312         //-prints(displays) the integer contained in R0
313         void printInstruction(){
```

```
314             //System.err.println
("\t\t\tprintInstruction " + R0);  //for instruction
trace
315         //put write signal on control lines object
316         controlLines.set(1);
317         //Throw data on the bus using its
components
318         dataLines.set(R0.getVal());
319         //grab data from the bus and throw it on
the printer
320         printer.setBuffer(dataLines.get());
321         //print data from the printer
322         System.out.println(">> " + printer.
getBuffer());
323     }
324
325     //Move instruction
326     //regB <- [regA]
327     void moveInstruction(Register regA, Register
regB){
328             //System.err.println("\t\t\tMOVE " +
regA + "," + regB);  //for instruction trace
329         regB.setVal(regA.getVal());
330     }
331     void MOVE(Integer regA, Integer regB){
332         //get values
333         Integer temp1 =0;
334         //regA value
335         if(regA==0) temp1 = R0.getVal();
336         else if(regA==1) temp1 = R1.getVal();
337         else if(regA==2) temp1 = R2.getVal();
338         else if(regA==3) temp1 = R3.getVal();
339         else HALT();
340         //store in regB
```

```
341         if(regB==0) R0.setVal(temp1);
342         else if(regB==1) R1.setVal(temp1);
343         else if(regB==2) R2.setVal(temp1);
344         else if(regB==3) R3.setVal(temp1);
345         else HALT();
346     }
347 
348     //Add instruction
349     //regC <-[regA] + [regB]
350     void addInstruction(Register regA, Register
regB, Register regC){
351             //System.err.println("\t\t\tADD " +
regA + "," + regB + "," + regC); //for instruction
trace
352         //add the values and store in regC
353         regC.setVal(adder.add(regA.getVal(),regB.
getVal()));
354     }
355 
356     void ADD(Integer regA, Integer regB, Integer
regC){
357         //get values
358         Integer temp1 =0, temp2=0;
359         //regA value
360         if(regA==0) temp1 = R0.getVal();
361         else if(regA==1) temp1 = R1.getVal();
362         else if(regA==2) temp1 = R2.getVal();
363         else if(regA==3) temp1 = R3.getVal();
364         else HALT();
365         //regB value
366         if(regB==0) temp2 = R0.getVal();
367         else if(regB==1) temp2 = R1.getVal();
368         else if(regB==2) temp2 = R2.getVal();
369         else if(regB==3) temp2 = R3.getVal();
```

```
370          else HALT();
371          //store in regC
372          if(regC==0) R0.setVal(adder.add(temp1,
temp2));
373          else if(regC==1) R1.setVal(adder.add(temp1,
temp2));
374          else if(regC==2) R2.setVal(adder.add(temp1,
temp2));
375          else if(regC==3) R3.setVal(adder.add(temp1,
temp2));
376          else HALT();
377      }
378
379      //Sub instruction
380      //regC <- [regB] - [regA]
381      void subInstruction(Register regA, Register
regB, Register regC){
382          //System.err.println("\t\t\tSUB " + regA +
"," + regB + "," + regC);
383          // add regB and the complement of regA and
store into regC
384          regC.setVal(adder.add(regB.getVal(),
complementer.complement(regA.getVal())));
385      }
386
387      void SUB(Integer regA, Integer regB, Integer
regC){
388          //get values
389          Integer temp1 =0, temp2=0;
390          //regA value
391          if(regA==0) temp1 = R0.getVal();
392          else if(regA==1) temp1 = R1.getVal();
393          else if(regA==2) temp1 = R2.getVal();
394          else if(regA==3) temp1 = R3.getVal();
```

```
395             else HALT();
396             //regB value
397             if(regB==0) temp2 = R0.getVal();
398             else if(regB==1) temp2 = R1.getVal();
399             else if(regB==2) temp2 = R2.getVal();
400             else if(regB==3) temp2 = R3.getVal();
401             else HALT();
402             //store in regC
403             if(regC==0) R0.setVal(adder.add
(complementer.complement(temp1), temp2));
404             else if(regC==1) R1.setVal(adder.add
(complementer.complement(temp1), temp2));
405             else if(regC==2) R2.setVal(adder.add
(complementer.complement(temp1), temp2));
406             else if(regC==3) R3.setVal(adder.add
(complementer.complement(temp1), temp2));
407             else HALT();
408         }
409         //LOAD instruction
410         //loads a value into the register
411         void LOAD(Register destination, Integer source)
{
412                 //System.err.println("\t\t\tLOAD " +
destination + "," + source);  //for instruction trace
413             //store source into destination
414             destination.setVal(source);
415         }
416
417
418
419         //STORE instruction
420         //stores the value from a register into memory
location
421         void STORE(Register source, Integer
```

```
destination){
422             //System.err.println("\t\t\tSTORE " +
source + "," + destination);  //for instruction trace
423         //Grab source value and place it in memory
at the destination address
424         storeMemory(destination,source.getVal());
425     }
426
427     //DEC instruction
428     //decrements a register value
429     void DEC(Register reg){
430         //decrement a register value by one through
adder
431         reg.setVal(adder.add(reg.getVal(),-1));
432     }
433
434     //Reading a word from memory
435     Integer readMemory(Integer address){
436         //put address on the MAR
437         mAR.set(address);
438         //mAR -> address lines
439         addressLines.set(mAR.get());
440         //set signal on control lines
441         controlLines.set(0);
442         //address lines -> memory control -> memory
-> mDR
443         dataLines.set(memoryControl.getMemory
(addressLines.get()));
444         //datalines -> mDR
445         mDR.set(dataLines.get());
446         //return data from mDR
447         return mDR.get();
448     }
449
```

```
450     //Store a word in memory at address
451     void storeMemory(Integer address, Integer
value){
452         //put value on the MDR
453         mDR.set(value);
454         //send data to the data lines from the MDR
455         dataLines.set(mDR.get());
456         //put address on the MAR
457         mAR.set(address);
458         //send address to the address lines from
the MAR
459         addressLines.set(mAR.get());
460         //set signal on the control lines
461         controlLines.set(1);
462         //use memory control to store the data in
memory and the address
463         memoryControl.setMemory(addressLines.get(),
dataLines.get());
464     }
465
466     //HALT instruction
467     //signals end of execution
468     void HALT(){
469         System.out.println("HALT() -> Program will
terminate.");
470         memoryDump();
471         status.setRunning(false);
472     }
473
474     //NOOP instruction
475     //will be ignored when executed
476     void NOOP(){
477         //nothing happens
478     }
```

```
479
480     //BRNZ instruction
481     //branch to an absolute address
482     void BRNZ(Integer address){
483         programCounter.setCounter(address);
484     }
485
486     //Memory Dump
487     void memoryDump(){
488         memoryControl.memoryDump();
489     }
490 }
491
```

```java
/*
    Memory address register: either stores the memory address
    from which data will be fetched from the CPU, or the address
    to which data will be sent and stored
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class MemoryAddressRegister {
    //Properties
    private Integer val;

    //Default Constructor
    MemoryAddressRegister(){
        val = 0;
    }

    //Mutator
    void set(Integer value){
        val=value;
    }

    //Accessor
    Integer get(){
        return val;
    }

}

```

32

```
 1 /*
 2     Memory Data Register: register of the computer's
 3     control unit that contains the data to be stored
 4     in the computer storage, OR the data after a
fetch
 5     from computer storage.
 6  */
 7 package simple_computer_simulation;
 8
 9
10 /**
11  *
12  * @author bhitt
13  */
14 public class MemoryDataRegister {
15     //Properties
16     private Integer val;
17
18     //Default Constructor
19     MemoryDataRegister(){
20         val=0;
21     }
22
23     //Mutator
24     void set(Integer value){
25         val = value;
26     }
27
28     //Accessor
29     Integer get(){
30         return val;
31     }
32
33
```

```
34 }
35
```

```java
/*
    Memory:
 */
package simple_computer_simulation;


/**
 *
 * @author bhitt
 */
public class Memory {
    //Properties
    private Integer [] values;

    //Default Constructor
    Memory(){
    //1024 word size register
        values = new Integer[1024];
        for(int i=0;i<1024;i++){
            values[i]=0;
        }
    }

    //Mutator
    void set(Integer address, Integer value){
        values[address] = value;
    }

    //Accessor
    Integer get(Integer address){
        return values[address];
    }


```

```
35      //Memory Dump
36      void memoryDump(){
37          for(int i=0;i<1024;i++){
38              if(values[i] != 0){
39                  System.out.println("Memory Dump
Address:"+i+" | Content:"+values[i]);
40              }
41          }
42      }
43  }
44
```

```java
/*
    Memory Control: controls the memory
 */
package simple_computer_simulation;


/**
 *
 * @author bhitt
 */
public class MemoryControl {
    //Properties
    private Memory memory;

    //Default constructor
    MemoryControl(){
        memory = new Memory();
    }

    //Mutators
    void setMemory(Integer address, Integer value){
        memory.set(address, value);
    }

    //Accessors
    Integer getMemory(Integer address){
        return memory.get(address);
    }

    //Memory dump call
    void memoryDump(){
        memory.memoryDump();
    }
}
```

35

```
 1 /*
 2     Printer
 3     -displays the content of its integer buffer.
Precedes the value
 4     with the string ">>" to flag it as output from
the program.
 5     Note: the blank spaces following the * and >>.
 6  */
 7 package simple_computer_simulation;
 8
 9 /**
10  *
11  * @author bhitt
12  */
13 public class Printer {
14     //Properties
15     private Integer buffer;
16
17     //Default Constructor
18     Printer(){
19         buffer=0;
20     }
21
22     //Constructor
23     Printer(Integer buff){
24         buffer = buff;
25     }
26
27     //Accessor
28     Integer getBuffer(){
29         return buffer;
30     }
31
32     //Mutator
```

```
33      void setBuffer(Integer buff){
34          buffer = buff;
35      }
36
37
38  }
39
```

```java
/*
    Program Counter: holds the next address of instruction
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class ProgramCounter {
    //Properties
    private Integer address;
    //Default Constructor
    ProgramCounter(){
        address=0;
    }
    //Mutator
    void setCounter(Integer add){
        address = add;
    }
    //Accessor
    Integer getCounter(){
        return address;
    }
}

```

```java
/*
    Program Loader: loads in the program and stores it in memory
 */
package simple_computer_simulation;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author bhitt
 */
public class ProgramLoader {
    //Properties
    private String displayName;
    private Integer firstWord;
    private Integer firstInstruction;
    private List<Integer> instructions;
    //Default Constructor
    ProgramLoader(){
        //instantiate list
        instructions = new ArrayList<>();
        //Read in from the file "SimpleComputer_Program1.txt
        File programFile = new File ("SimpleComputer_Program1.txt");
        //check to see the file exists
        if(!programFile.exists()){
```

```
32              System.out.println("The program file
'SimpleComputer_Program1.txt' does not exist.");
33          }else{
34              try {
35                  //read in from file
36                  Scanner fileIn = new Scanner
(programFile);
37                  //read in display name
38                  displayName = fileIn.nextLine();
39                  //display the program name
40                  System.out.println(displayName);
41                  //get the memory location where the
first program word is to be loaded
42                  firstWord = fileIn.nextInt();
43                      //System.out.println("First word
at memory location:"+firstWord);
44                  //get the memory location of the
first instuction to be executed
45                  firstInstruction = fileIn.nextInt();
46                      //System.out.println("First
instruction to be executed at memory location:"
+firstInstruction);
47                  //read in the rest of the
instructions
48                  while(fileIn.hasNextLine()){
49                      //get a line
50                      String line = fileIn.nextLine();
51                      //System.out.println(line);
52                      line = line.split("/")[0];
53                      //System.out.println(line);
54                      String[] numbers = line.split("
");
55                      for(int i=0;i<numbers.length;
i++){
```

```
56                              if(!numbers[i].isEmpty()){
57                                  instructions.add
(Integer.parseInt(numbers[i]));
58                              }
59                          }
60                      }
61
62                      //output list TESTING PURPOSES
63 //                     for(int i=0;i<instructions.size();
i++){
64 //                         System.out.println("["+i+"] :"
+instructions.get(i));
65 //                     }
66                  } catch (FileNotFoundException ex) {
67                      Logger.getLogger(ProgramLoader.
class.getName()).log(Level.SEVERE, null, ex);
68                  }
69              }
70      }
71      //Accessors
72      Integer getFW(){
73          return firstWord;
74      }
75      Integer getFI(){
76          return firstInstruction;
77      }
78      List<Integer> getInstruction(){
79          return instructions;
80      }
81 }
82
```

```java
/*
    Reader
    -reads an integer from the keyboard and stores
    it in its buffer. Precedes the actual read by
sending
    a "*" character as a prompt.
 */
package simple_computer_simulation;

import java.util.Scanner;

/**
 *
 * @author bhitt
 */
public class Reader {
    //Properties
    private Integer buffer;

    //Default Constructor
    Reader(){
        buffer = 0;
    }
    //Constructor
    Reader(Integer buff){
        buffer = buff;
    }

    //Accessor
    Integer getOutput(){
        return buffer;
    }

    //Mutator
```

```
34      void setBuffer(){
35          //Prompt for input
36          System.out.print("* ");
37          //grab input
38          Scanner input = new Scanner(System.in);
39          buffer = input.nextInt();
40      }
41  }
42
```

```java
/*
    Register
    -holds an integer value
 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class Register {
    //Properties
    private Integer val;

    //Default Constructor
    Register(){
        val = 0;
    }

    //Constructor
    Register(Integer value){
        val = value;
    }

    //Mutator
    void setVal(Integer value){
        val = value;
    }

    //Accessor
    Integer getVal(){
        return val;
    }

```

```
35 }
36
```

```java
/*

 */
package simple_computer_simulation;

/**
 *
 * @author bhitt
 */
public class Simple_Computer_Simulation {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // instantiate the computer object and build
it
        ISA computer = new ISA();
        // load the program
        ProgramLoader programLoader = new
ProgramLoader();
        computer.loadProg(programLoader.getFW(),
programLoader.getFI(), programLoader.getInstruction());
        //run the ISA
        computer.run();

    }

}

```

```
 1 /*
 2     Status: includes a Zero indicator to be set if
the result
 3     of any arithmetic operation is zero
 4  */
 5 package simple_computer_simulation;
 6
 7 /**
 8  *
 9  * @author bhitt
10  */
11 public class Status {
12     //Properties
13    private boolean flag;
14    private boolean running;
15
16     //Default Constructor
17     Status(){
18         flag = false;
19     }
20
21     //Mutators
22     void setFlag(boolean val){
23         flag = val;
24     }
25
26     void setRunning(boolean val){
27         running = val;
28     }
29
30     //Accessors
31     boolean getFlag(){
32         return flag;
33     }
```

```
34
35      boolean getRunning(){
36          return running;
37      }
38
39  }
40
```