Lab #07 Assembler

Name

Convert the following assembly code into "symbol less" code by replacing each symbol (variable or label) with its corresponding value (number). Also, please label the ROM address (line number) for each real instruction.

1. Sum.asm

Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)
`// Computes sum = R2 + R3`		
`// (R2 refers to RAM[2])`		
	0	`// computes sum = RAM[2]+ RAM[3]`
`@R2`	1	`@2`
`D=M`	2	`D=M`
	3	`@3`
`@R3`	4	`D=D+M`
`D=D+M  // Add R2 + R3`	5	`@0`
	6	`M=D`
`@sum`		
`M=D  // sum = R2 + R3`		

2. Max.asm

Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)
`// Computes R2=max(R0, R1)`		
`// (R0,R1,R2 refer to`		
`// RAM[0],RAM[1],RAM[2])`		
	0	`// computes R2=max(R0,R1)`
`@R0`	1	`@0`
`D=M`	2	`D=M`
`@R1`	3	`@1`
`D=D-M`	4	`D=D-M`
`@OUTPUT_FIRST`	5	`@OUTPUT_FIRST`
`D;JGT`	6	`D;JGT`
`@R1`	7	`@1`
`D=M`	8	`D=M`
`@OUTPUT_D`	9	`@OUTPUT_D`
`0;JMP`	10	`0;JMP`
`(OUTPUT_FIRST)`	11	`(OUTPUT_FIRST)`
`@R0`	12	`@0`
`D=M`	13	`D=M`
`(OUTPUT_D)`	14	`(OUTPUT_D)`
`@R2`	15	`@2`
`M=D`	16	`M=D`
`(INFINITE_LOOP)`	17	`(INFINITE_LOOP)`
`@INFINITE_LOOP`	18	`@INFINITE_LOOP`
`0;JMP`	19	`0;JMP`

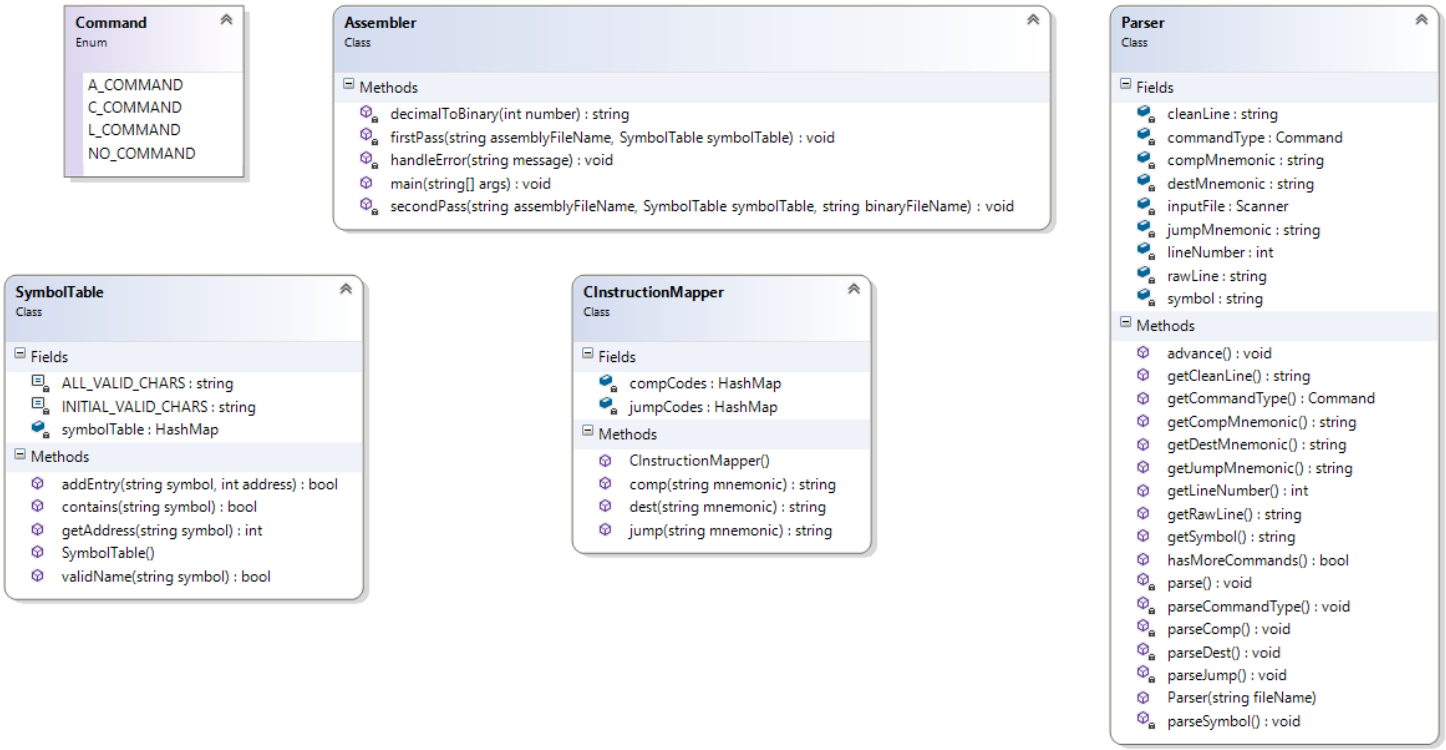
Lab #07 Assembler

3. Rect.asm

Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)
// Draws a rectangle at		
// the top-left corner of		
// the screen.		
// The rectangle is 16	1	@16
// pixels wide and R0	2	D=M
// pixels high.	3	@INFINITE_LOOP
	4	D;JLE
@R0	5	@counter
D=M	6	M=D
@INFINITE_LOOP	7	@16384
D;JLE	8	D=A
@counter	9	@address
M=D	10	M=D
@SCREEN	11	(LOOP)
D=A	12	@address
@address	13	A=M
M=D	14	M=-1
(LOOP)	15	@address
@address	16	D=M
A=M	17	@32
M=-1	18	D=D+A
@address	19	@address
D=M	20	M=D
@32	21	@counter
D=D+A	22	MD=M-1
@address	23	@LOOP
M=D	24	D;JGT
@counter	25	(INFINITE_LOOP)
MD=M-1	26	@INFINITE_LOOP
@LOOP	27	0;JMP
D;JGT		
(INFINITE_LOOP)		
@INFINITE_LOOP		
0;JMP		

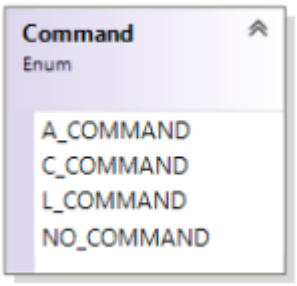
Lab #07 Assembler

UML Diagram of Entire Assembler Program



A brief Java refresher:

Write Java code to implement the following enum:




```
public enum A_COMMAND {
}
public enum C_COMMAND {
}
public enum L_COMMAND {
}
public enum NO_COMMAND {
}
```

What does the following code display?

```
String code = "\t0;JMP  //unconditional jump  ";
System.out.println(code.trim());
```

```
0;JMP
```

How would you extract the `JMP` from the `code` string above? Write the Java code to do so.

```
public class Main {
    public static void main(String[] args) {
        String code = "\t0;JMP //unconditional jump ";
        String jmp = extractJMP(code);
        System.out.println("Extracted JMP: " + jmp);
    }
    public static String extractJMP(String code) {
        String[] parts = code.split(";");
        String lastPart = parts[parts.length - 1];
        String jmp = lastPart.trim();
        return jmp;
    }
}
```

What is assigned to the variable `dest` ?

```
String code = "D=M;JGT";
int index = code.indexOf('=');
String dest = (index != -1) ?
      code.substring(0, index) : null;
```

dest is assigned to the value D

Lab #07 Assembler

Write pseudocode for the following helper methods:

- ❑ <u>String cleanLine(String rawLine)</u>

```
//DESCRIPTION:   cleans raw instruction by removing non-essential parts
//PRECONDITION:  String parameter given (not null)
//POSTCONDITION: returned without comments and whitespace

    String cleanLine(String rawLine)

        cleanedLine = ""

        isInsideComment = false

        for char in rawLine

            if char is '/' AND next char is '/'
                isInsideComment = true
                  break
            if NOT isInsideComment
                cleanedLine += char

        cleanedLine = cleanedLine.trim()

        return cleanedLine
```

- ❑ <u>Command parseCommandType(String cleanLine)</u>

```
//DESCRIPTION:   determines command type from parameter
//PRECONDITION:  String parameter is clean instruction
//POSTCONDITION: returns A_COMMAND (A-instruction),
//                C_COMMAND (C-instruction), L_COMMAND (Label) or
//                NO_COMMAND (no command)



          Command parseCommandType(String cleanLine)

              if cleanLine is null OR cleanLine is empty
                  return NO_COMMAND

                  if cleanLine starts with '@'
                  return A_COMMAND

              if cleanLine starts with '(' AND cleanLine ends with ')'
                  return L_COMMAND

                  if cleanLine contains '=' OR cleanLine contains ';'
                  return C_COMMAND

              return NO_COMMAND
```

Lab #07 Assembler

- boolean isValidName(String symbol)

```
//DESCRIPTION:   checks validity of identifiers for assembly code symbols
//PRECONDITION:  start with letters or "_.$:" only, numbers allowed after
//POSTCONDITION: returns true if valid identifier, false otherwise



  boolean isValidName(String symbol)

      if symbol is null OR symbol is empty
          return false

      firstChar = symbol[0]

      if NOT (firstChar is letter OR firstChar is '_' OR firstChar is '.' OR firstChar is '$' OR firstChar
  is ':')
          return false

      for i = 1 to length of symbol - 1
          currentChar = symbol[i]

          if NOT (currentChar is letter OR currentChar is digit OR currentChar is '_' OR currentChar is '.'
  OR currentChar is '$' OR currentChar is ':')
              return false

      return true
```

- String decimalToBinary(int number)

```
//DESCRIPTION:   converts integer from decimal notation to binary notation
//PRECONDITION:  number is valid size for architecture, non-negative
//POSTCONDITION: returns 16-bit string of binary digits (first char is MSB)



              String decimalToBinary(int number)
                  binaryString = ""
                  if number < 0
                      return "ERROR: Number must be non-negative"
                  if number is 0
                      binaryString = "0000000000000000"
                      return binaryString
                  while number > 0
                      lsb = number % 2
                      binaryString = concatenate lsb with binaryString
                      number = number / 2
                  while length of binaryString < 16
                      binaryString = concatenate "0" with binaryString
                  return binaryString
```

Lab #07 Assembler

```
CInstructionMapper
-  compCodes : HashMap<String, String>
-  destCodes : HashMap<String, String>
-  jumpCodes : HashMap<String, String>
+  CInstructionMapper()
+  comp(mnemonic : String) : String
+  dest(mnemonic : String) : String
+  jump(mnemonic : String) : String
```

Write pseudocode for the following Code methods:

- ❑ Code()

```
//DESCRIPTION:   initializes hashmaps with binary codes for easy lookup
//PRECONDITION:  comp codes = 7 bits (includes a), dest/jump codes = 3 bits
//POSTCONDITION: all hashmaps have lookups for valid codes
```

```
Class Code

    compCodes : HashMap<String, String>
    destCodes : HashMap<String, String>
    jumpCodes : HashMap<String, String>

     **unsure about what the
     pseudocode would entail
     for the hashmap**
```

```
Code()
       compCodes = new HashMap<String, String>()
       compCodes.put("0", "0101010")
       compCodes.put("1", "0111111")
       compCodes.put("-1", "0111010")
       compCodes.put("D", "0001100")
       compCodes.put("A", "0110000")
       compCodes.put("!D", "0001101")
       compCodes.put("!A", "0110001")
       compCodes.put("-D", "0001111")
       compCodes.put("-A", "0110011")
       compCodes.put("D+1", "0011111")
       compCodes.put("A+1", "0110111")
       compCodes.put("D-1", "0001110")
       compCodes.put("A-1", "0110010")
       compCodes.put("D+A", "0000010")
       compCodes.put("D-A", "0010011")
       compCodes.put("A-D", "0000111")
       compCodes.put("D&A", "0000000")
       compCodes.put("D|A", "0010101")
       compCodes.put("M", "1110000")
       compCodes.put("!M", "1110001")
       compCodes.put("-M", "1110011")
       compCodes.put("M+1", "1110111")
       compCodes.put("M-1", "1110010")
       compCodes.put("D+M", "1000010")
       compCodes.put("D-M", "1010011")
       compCodes.put("M-D", "1000111")
       compCodes.put("D&M", "1000000")
       compCodes.put("D|M", "1010101")

       destCodes = new HashMap<String, String>()
       destCodes.put(null, "000")
       destCodes.put("M", "001")
       destCodes.put("D", "010")
       destCodes.put("MD", "011")
       destCodes.put("A", "100")
       destCodes.put("AM", "101")
       destCodes.put("AD", "110")
       destCodes.put("AMD", "111")

       jumpCodes = new HashMap<String, String>()
       jumpCodes.put(null, "000")
       jumpCodes.put("JGT", "001")
       jumpCodes.put("JEQ", "010")
       jumpCodes.put("JGE", "011")
       jumpCodes.put("JLT", "100")
       jumpCodes.put("JNE", "101")
       jumpCodes.put("JLE", "110")
       jumpCodes.put("JMP", "111")
```

- ❑ String comp(String mnemonic)

```
//DESCRIPTION:   converts to string of bits (7) for given mnemonic
//PRECONDITION:  hashmaps are built with valid values
//POSTCONDITION: returns string of bits if valid, else returns null
```

Lab #07 Assembler

- ❑ String dest(String mnemonic)

```
//DESCRIPTION:   converts to string of bits (3) for given mnemonic
//PRECONDITION:  hashmaps are built with valid values
//POSTCONDITION: returns string of bits if valid, else returns null



       function String comp(String mnemonic)
           if (mnemonic is not a valid mnemonic in the hashmap) then
               return null

           String compCode = retrieve_comp_code_from_hashmap(mnemonic)

           return compCode
```

- ❑ String jump(String mnemonic)

```
//DESCRIPTION:   converts to string of bits (3) for given mnemonic
//PRECONDITION:  hashmaps are built with valid values
//POSTCONDITION: returns string of bits if valid, else returns null


           function String jump(String mnemonic)
               if (mnemonic is not a valid mnemonic in the hashmap) then
                   return null

               String jumpCode = retrieve_jump_code_from_hashmap(mnemonic)

               return jumpCode
           end function
```

Lab #07 Assembler

Write pseudocode for the following SymbolTable methods:

```
SymbolTable
-  INITIAL_VALID_CHARS : String
-  ALL_VALID_CHARS : String
-  symbolTable : HashMap<String, Integer>
+  SymbolTable()
+  addEntry(symbol : String, address : int) : boolean
+  contains(symbol : String) : boolean
+  getAddress(symbol : String) : int
-  isValidName(symbol : String) : boolean
```

- ❑ SymbolTable()

```
//DESCRIPTION:   initializes hashmap with predefined symbols
//PRECONDITION:  follows symbols/values from book/appendix
//POSTCONDITION: all hashmap values have valid address integer
class SymbolTable
    symbolTable : HashMap<String, Integer>
    function SymbolTable()
        symbolTable = new HashMap<String, Integer>
```

- ❑ boolean addEntry(String symbol, int address)

```
//DESCRIPTION:   adds new pair of symbol/address to hashmap
//PRECONDITION:  symbol/address pair not in hashmap (check contains() 1st)
//POSTCONDITION: adds pair, returns true if added, false if illegal name
class SymbolTable
    symbolTable : HashMap<String, Integer>
    function boolean addEntry(String symbol, int address)
        if (symbolTable.contains(symbol)) then
            return false
        if (not isValidName(symbol)) then
            return false
        symbolTable.put(symbol, address)
        return true
    end function
end class
```

- ❑ boolean contains(String symbol)

```
//DESCRIPTION:   returns boolean of whether hashmap has symbol or not
//PRECONDITION:  table has been initialized
//POSTCONDITION: returns boolean if arg is in table or not
function boolean contains(String symbol)
    // contains method of the hashmap to check if the symbol is present
    boolean isPresent = symbolTable.contains(symbol)
    return isPresent
end function
end class
```

- ❑ int getAddress(String symbol)

```
//DESCRIPTION:   returns address in hashmap of given symbol
//PRECONDITION:  symbol is in hashmap (check w/ contains() first)
//POSTCONDITION: returns address associated with symbol in hashmap
// Data members
// Constructor and other methods
//Method for address with given symbol from hashmap
// Get address with given symbol using get
```

- ❑ boolean isValidName(String symbol) //same as earlier but rewrite using constants

Lab #07 Assembler

```
                              Parser
+  NO_COMMAND : char // 'N'            //constants
+  A_COMMAND : char //  'A'
+  C_COMMAND : char //  'C'
+  L_COMMAND : char //  'L'

-  inputFile : Scanner                 //file stuff +
   debugging
-  lineNumber : int
-  rawLine : String

-  cleanLine : String             //parsed command parts
-  commandType : char
-  symbol : String
-  destMnemonic : String
-  compMnemonic : String
-  jumpMnemonic : String
+  Parser(inFileName : String)         //drivers
+  hasMoreCommands() : boolean
+  advance() : void

-  cleanLine() : void                  //parsing helpers
-  parseCommandType() : void
-  parse() : void
-  parseSymbol() : void
-  parseDest() : void
-  parseComp() : void
-  parseJump() : void

+  getCommandType() : char             //useful getters
+  getSymbol() : String
+  getDest() : String
+  getComp() : String
+  getJump() : String

+  getRawLine() : String               //debugging getters
+  getCleanLine() : String
+  getLineNumber() : int
```

- ❑ cleanLine() : void //same as part 1 but rewrite using instance variables
- ❑ parseCommandType() : void //same as part 1 but rewrite using instance variables

Lab #07 Assembler

Write pseudocode for the following Parser methods:

- ❑ Parser(String fileName)

```
//DESCRIPTION:   opens input file/stream and prepares to parse
//PRECONDITION:  provided file is ASM file
//POSTCONDITION: if file can't be opened, ends program w/ error message


    class Parser
        inputFile : Scanner
        function Parser(String fileName)
            try
                inputFile = new Scanner(fileName)
            catch (FileNotFound exception)
                print("Error: The provided file cannot be found or opened.")
                exit program with error
            end try
        end function
    end class
```

- ❑ boolean hasMoreCommands()

```
//DESCRIPTION:   returns boolean if more commands left, closes stream if not
//PRECONDITION:  file stream is open
//POSTCONDITION: returns true if more commands, else closes stream

             class Parser
                 inputFile : Scanner
                 function boolean hasMoreCommands()
                     boolean hasMoreCommands = inputFile.hasNextLine()
                     if (not hasMoreCommands)
                         inputFile.close()
                     end if
                     return hasMoreCommands
                 end function
             end class
```

- ❑ void advance()

```
//DESCRIPTION:   reads next line from file and parses it into instance vars
//PRECONDITION:  file stream is open, called only if hasMoreCommands()
//POSTCONDITION: current instruction parts put into instance vars
```

Lab #07 Assembler

- ❑ void parseSymbol()

```
//DESCRIPTION:   parses symbol for A- or L-commands
//PRECONDITION:  advance() called so cleanLine has value,
//   call for A- and L-commands only
//POSTCONDITION: symbol has appropriate value from instruction assigned
```

- ❑ void parseDest()

```
//DESCRIPTION:   helper method parses line to get dest part
//PRECONDITION:  advance() called so cleanLine has value,
//   call for C-instructions only
//POSTCONDITION: destMnemonic set to appropriate value from instruction
```

- ❑ void parseComp()

```
//DESCRIPTION:   helper method parses line to get comp part
//PRECONDITION:  advance() called so cleanLine has value,
//   call for C-instructions only
//POSTCONDITION: compMnemonic set to appropriate value from instruction
```

Lab #07 Assembler

- ❑ void parseJump()

```
//DESCRIPTION:   helper method parses line to get jump part
//PRECONDITION:  advance() called so cleanLine has value,
//   call for C-instructions only
//POSTCONDITION: jumpMnemonic set to appropriate value from instruction
```

- ❑ void parse()

```
//DESCRIPTION:   helper method parses line depending on instruction type
//PRECONDITION:  advance() called so cleanLine has value
//POSTCONDITION: appropriate parts (instance vars) of instruction filled
```

Lab #07 Assembler

- ❑ Command getCommandType()

```
//DESCRIPTION:   getter for command type
//PRECONDITION:  cleanLine has been parsed (advance was called)
//POSTCONDITION: returns Command for type (N/A/C/L)
```

- ❑ String getSymbol()

```
//DESCRIPTION:   getter for symbol name
//PRECONDITION:  cleanLine has been parsed (advance was called),
//   call for labels only (use getCommandType())
//POSTCONDITION: returns string for symbol name
```

- ❑ String getDestMnemonic()

```
//DESCRIPTION:   getter for dest part of C-instruction
//PRECONDITION:  cleanLine has been parsed (advance was called),
//  call for C-instructions only (use getCommandType())
//POSTCONDITION: returns mnemonic (ASM symbol) for dest part
```

- ❑ String getCompMnemonic()

```
//DESCRIPTION:   getter for comp part of C-instruction
//PRECONDITION:  cleanLine has been parsed (advance was called),
//  call for C-instructions only (use getCommandType())
//POSTCONDITION: returns mnemonic (ASM symbol) for comp part
```

Lab #07 Assembler

- ❑ String getJumpMnemonic()

```
//DESCRIPTION:   getter for jump part of C-instruction
//PRECONDITION:  cleanLine has been parsed (advance was called),
//  call for C-instructions only (use getCommandType())
//POSTCONDITION: returns mnemonic (ASM symbol) for jump part
```

- ❑ String getRawLine()

```
//DESCRIPTION:   getter for rawLine from file (debugging)
//PRECONDITION:  advance() was called to put value from file in here
//POSTCONDITION: returns string of current original line from file
```

- ❑ String getCleanLine()

```
//DESCRIPTION:   getter for cleanLine from file (debugging)
//PRECONDITION:  advance() and cleanLine() were called
//POSTCONDITION: returns string of current clean instruction from file
```

- ❑ int getLineNumber()

```
//DESCRIPTION:   getter for lineNumber (debugging)
//PRECONDITION:  n/a
//POSTCONDITION: returns line number currently being processed from file
```