Lab #07 Assembler

Name Raul Aguilar

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])`
 	
`@R2`
`D=M`
 	
`@R3`
`D=D+M  // Add R2 + R3`
 	
`@sum`
`M=D  // sum = R2 + R3`
	0
	1
	2
	3
	4
	5
		@2
		D=M
		@3
		D=D+M
		@16
		M=D

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])`
 	
`  @R0`
`  D=M`
`  @R1`
`  D=D-M`
`  @OUTPUT_FIRST`
`  D;JGT`
`  @R1`
`  D=M`
`  @OUTPUT_D`
`  0;JMP`
`(OUTPUT_FIRST)`
`  @R0`
`  D=M`
`(OUTPUT_D)`
`  @R2`
`  M=D`
`(INFINITE_LOOP)`
`  @INFINITE_LOOP`
`  0;JMP`
	0
	1
	2
	3
	4
	5
	6
	7
	8
	9
	10
	11
	12
	13
	14
	15
		@0
		D=M
		@1
		D=D-M
		@10
		D;JGT
		@1
		D=M
		@12
		0;JMP
		@0
		D=M
		@2
		M=D
		@14
		0;JMP

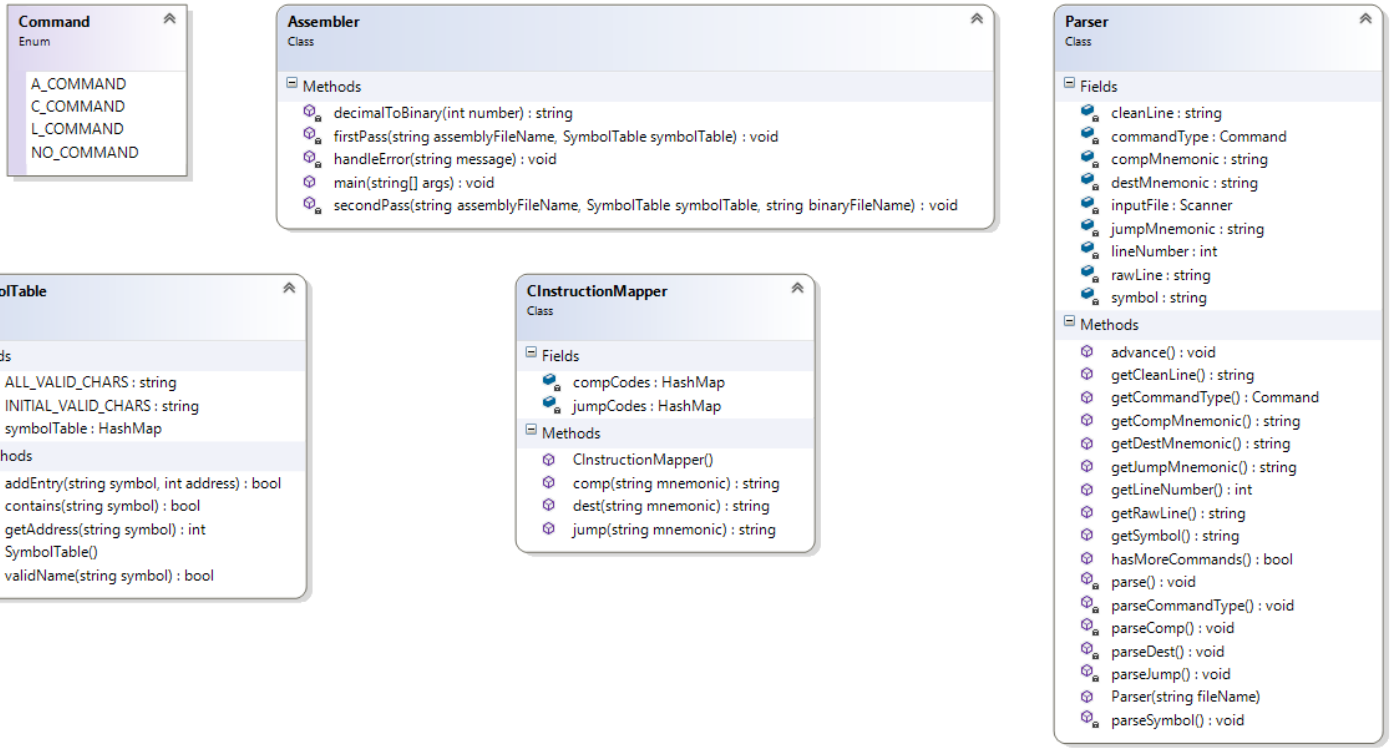
Lab #07 Assembler

3. Rect.asm

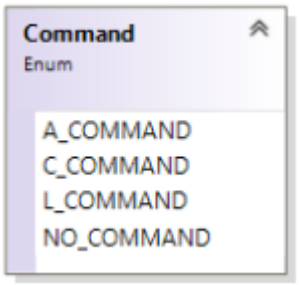
Assembly Code (raw with symbols)	ROM Address (line number)	Assembly Code (cleaned and without symbols)
// Draws a rectangle at	0	@0
// the top-left corner of	1	D=M
// the screen.	2	@23
// The rectangle is 16	3	D;JLE
// pixels wide and R0	4	@16
// pixels high.	5	M=D
	6	@16384
@R0	7	D=A
D=M	8	@17
@INFINITE_LOOP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

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.	

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;`	

Lab #07 Assembler

Write pseudocode for the following helper methods:

- ❑ String cleanLine(String rawLine)

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

```
if rawLine is empty
   cleanLine is empty
else remove whitespace and comments
```

- ❑ Command parseCommandType(String cleanLine)

```
//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)
```

```
if cleanLine is empty or length 0,
    then commandType is NO_COMMAND
otherwise,
    retrieve the first char in cleanLine
    if char is '('
        then commandType is Label
    else if char is '@'
        then commandTye is A-instruction
    else
        commandType is C-instruction
```

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
```

create isValidName flag
loop through each character in symbol string to compare against characters in a VALID_CHARS string
if a character in symbol string is not found in VALID_CHARS,
then symbolName is invalid
set flag to false
display error message with line number
exit program
otherwise,
symbol name is valid
set flag to true

return flag status

- ❑ 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)
```

create empty string called binary

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
```

- ❑ 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
```

return comp code retrieved from mnemonic key

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
```

- ❑ 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
```

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
        initialize predefined symbols into HashMap
```

- ❑ 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
      create entryAdded flag
      if HashMap contains symbol already,
          set flag to false
      otherwise, if symbol name is valid
          then add symbol to table
          set flag to true
      return flag status
```

- ❑ 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
        if SymbolTable contains key,
            return true
        else
            return false
```

- ❑ 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
        get address value from table
```

- ❑ 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
```

- ❑ 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

    if input file has next line
       return true
    else
       close file
       return false
```

- ❑ 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

       Increment line number
       set rawLine to the next line in file
       run cleanLine()
       run parseCommandType()
       run parse()
```

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
```

if L-command,
  get index of opening parenteses
  get index of closing paventeses
  symbol is after first paventeses and before closing paventeses
if A-command,
  symbol name is after first char (@)

- ❑ 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
```

initialize binary codes
get index of equals sign (=) from c-instruction
if no index is found,
  there is no dest - set destMnemonic to null code
if index is found,
  get substring from beginning of instruction to equal sign index
  clean line
  retrieve code for destMnemonic

- ❑ 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
```

if commandType == L-command or A-command
    run parseSymbol()

if commandType == C-command
    run parseComp()
    run parseDest()
    run parseJump()

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
```