Lab 21: HACK Assembler: Overall Construction

1. Objective

In this lab, we will implement the HACK assembler. The implementation will be done using pseudo code.

1.1 Assembler Overview

An assembler is a program that will take assembly code and translate it into the corresponding machine instructions (binary). The task for this lab and the next will be to design an assembler for HACK.

2. Look Up Tables Recap

2.1 Symbol Tables

Given a symbol, the symbol table will provide the equivalent address in ROM or data memory.

The symbol table was defined as

```
symbolTable = HashTable(string, integer)
```

And contains redefined symbols, labels, and variables.

2.2 Instruction Tables

Three tables were defined for instructions, one for the comp section, one for dest, and one for jump. The format of the assembly is `dest = comp ; jump`

```
compTable = HashTable(string, integer)
destTable = HashTable(string, integer)
jumpTable = HashTable(string, integer)
```

2.3 Table creation

For the sake of making the code modular, let us assume the work from the last lab was put into functions.

```
createSymbolTable(asmFileName)
     // return a symbol table which has been populated

createCompTable()
     // return a comp table which has been populated

createDestTable()
     // return a dest table which has been populated

createJumpTable()
     // return a jump table which has been populated
```

3. Assembly construction

3.1 Idea

1. Create and populate the symbol table
2. Create and populate the instruction tables
3. Parse each line of the assembly, tokenize, and translate to binary

The assembler program will take a file name as input.

For each line:

- Skip if it is a comment or empty line
- If it is an A instruction, translate to binary or loop up in symbol table
- If it is a C instruction, break it up into its parts and look up each one in the correct instruction table.

3.2 Main Implementation

```
asmFileName = argument[1]

symbolTable = createSymbolTable(asmFileName)
compTable = createCompTable()
```

```
destTable = createDestTable()

jumpTable = createJumpTable()

file = openFile(asmFileName)

if (failedToOpen(file)):
     printErrorMessage("failed to open file")
     exit

fileNameWithoutExtension = asmFileName.Strip(".asm")

binaryFileOut = openFile(fileNameWithoutExtension + ".hack")

for line in file:

     line = removeWhitespace(line)
     line = stripComments(line)

     if ( isEmptyLine(line) ):
          continue

     if ( isAInstruction(line) ):
          bin = aInstruction(line)
          binaryFileOut.Write(bin)
          continue

     if ( isCInstruction(line) ):
          bin = cInstruction(line)
          binaryFileOut.Write(bin)
          continue

binaryFileOut.close()
file.close()
```

3.3 A Instruction Implementation

```
function isAInstruction(line):
     if (line[0] != '@') return false
     dropAt = line[1 : len(line) -1 ]
```

```
     if ( positiveInteger(dropAt) ):
          return true

     if ( isDigit(dropAt[0]) ):
          return false

     if ( validSymbolChars(dropAt) ):
          return true

     return false

function aInstruction(line):
     dropAt = line[1 : len(line) -1 ]

     if ( positiveInteger(dropAt) ):
          bin = toBinary15Bit(dropAt)
          return ( '0' + bin )

     if ( onlyAlphabetChars(dropAt) ):
          bin = symbolTable[dropAt]
          return ( '0' + bin)

     error()
// because of the checks done before calling aInstruction, this
// line should not be reached
```

3.4 C Instruction Implementation

```
function isCInstruction(line):

     if NOT ( ( countChars(line, '=') == 1
          OR countChars(line, ';') == 1) ) :
               return false

     tokens = splitStringAtChars(line, ['=', ';'])

     if (tokens.size() != 2 AND tokens.size() != 3 ):
          return false

     if (tokens.size() == 2 ):
```

```
          if ( countChars(line, '=') == 1 ):
               // dest and comp
               if ( NOT destTable.contains(tokens[0]) ):
                    return false
               if ( NOT compTable.contains(tokens[1]) ):
                    return false
          else:
               // comp and jump
               if ( NOT compTable.contains(tokens[0]) ):
                    return false
               if ( NOT jumpTable.contains(tokens[1]) ):
                    return false
     else:
          // contains dest, comp, and jump
          if ( NOT destTable.contains(tokens[0]) ):
               return false

          if ( NOT compTable.contains(tokens[1]) ):
               return false

          if ( NOT jumpTable.contains(tokens[2]) ):
               return false

     return true

function cInstruction(line):
     tokens = splitStringAtChars(line, ['=', ';'])
     prefix = "111"

     if (tokens.size() == 2):
          if ( countChars(line, '=') == 1 ):
               // dest and comp
               dest = destTable[ tokens[0] ]
               comp = compTable[ tokens[1] ]
               jump = "000"
               if (dest == null || comp == null):
                    error() // not value C instruction
          else:
               // comp and jump
               dest = "000"
               comp = compTable[ tokens[0] ]
```

```
          jump = jumpTable[ tokens[1] ]
          if (comp == null || jump == null ):
               error() // not valid C instruction

else: // all 3 tokens present
     dest = destTable[ tokens[0] ]
     comp = compTable[ tokens[1] ]
     jump = jumpTable[ tokens[2] ]

     if (dest == null || comp == null || jump == null):
          error() // not valid C instruction

return prefix + comp + dest + jump
```