Report for Lab-6a: LC-3 Assembler

1 Algorithm

The assembly proccess is accomplished by 2-Pass scan (as Chap 7.3 describes).

1st - Scan: Construct Symbol Table

In this part, the program use dead-loop to get input from `stdin` (break when having `.END`):

- To calculate the address of labels, we use `int lineNo` which grows @ each iteration .
- For each line:
  1. Split the whole line by *comma* or *space* , remove those token with length 0.
  2. Neglect *empty lines* : If the current line has 0 tokens, neglect it.
  3. Discover labels:

     Since label only appears at the beginning of a line, we just check the 1st token of current line.
     If it's NOT in the list of instruction keywords, then we push `(label, lineNo)` into symbol table.
  4. Push current line & lineNo into `todolist` .
  5. Special cases (`lineNo` grows more than 1):
     - For `.BLKW #N` , it should additionally grows `N-1` .
     - For `.STRINGZ "str"` , it should additionally grows `len(str)` .

2nd - Scan: Generate Machine Code

This part simply handle each instruction stored in `todolist` , for each instruction:

- We take `lineNo+1` as `PC` (used when calculate the offset to a specified label)
- The action to be taken is determined by token `operation` . Since Python don't have *SWITCH-CASE* grammar, this is implemented by multi branch *IF-ELIF* structure.
  - The basic concept is to generate code for `operation` by looking up in a hash table, then generate code for each operands. And in the end, print them to the screen.
  - Decimal number & Hexadecimal number would be distringuished by the 1st char of the token.
  - When calculate offset, we also check the 1st char of the token to decide whether it's a label or a decimal number (the former require us to look up the symbol table & turn `PC - addr(label)` into binary).

2 Code

Environment

```
keywords = {                   # dict(keywords) is used to:
    # opcodes                  # - judge whether a token is a label:
    "ADD":   "0001",           #   => token is NOT in keywords.keys()
    "AND":   "0101",           # - a hash table for some infos:
    # trap vectors             #   - for calc / data-move instructions: opcode
    "GETC":  "x20",            #   - for TRAP instructions: trap vector(hex)
    "OUT" :  "x21",            #   - for BR instructions: opcode + condition
    # BRs
    "BR" :   "0000111",
    "BRn":   "0000100",
    ...
}

todo = [                       # list(todo) store the token list for each instruction
  [int lineNo, str operation, str operand1, str operand2, ...],
  ...
]

symbols = {                    # dict(symbols) is the symbol table for the program
  "labelName": lineNo,         # - key:   str(labelName)
  ...                          # - value: int(lineNo of label)
}

regs = {                       # simplyfy the proccess of gen machine code of regs
    "R0": "000",               # we can use regs[token of reg] to get machine code
    "R1": "001",
    ...
}
```

Utils

- Token ⟶ Binary Number

```
def str2binN(immStr, len):             # turn string into binary num of given length
    if immStr[0] == 'x':               # judge the base of given string by 1st char
        return binN(immStr, 16, len)
    else:
        return binN(immStr, 10, len)

def binN(immStr, base, len):
    # we use abs(negNum) & 0b111..1(len*1) to calc 2's complement of negative number
    mask = {5 : 0x1F, 6 : 0x3F, 8 : 0xFF, 9 : 0x1FF, 11: 0x7FF, 16: 0xFFFF}
    # the immStr is in the form of "#NNNN" / "#-NNNN"
    if immStr[1] == '-':
        if int(immStr[1:], base) == 0:
            return str("").rjust(len, '0')
        else:
            return str(bin(int(immStr[1:], base) & mask[len]))[2:]
    else: # use rjust() to fill '0' to the left
        return str(bin(int(immStr[1:], base)))[2:].rjust(len, '0')
```

- Label ⟶ Binary Offset

```
def label2binN(pc, label, len):             # we should specify length for off9/off11
    offset = "#" + str(symbols[label] - pc) # lookup symbol table to get addr(label)
    return str2binN(offset, len)            # turn offset = addr(label) - PC to bin form
```

1st - Scan

```
while True:
    curline = re.split(',|\s', input())                  # split by comma & space
    curline = list(filter(lambda x : len(x)>0, curline)) # delete len(token) == 0

    if len(curline) == 0:    # neglect empty lines (with 0 tokens)
        continue
    if curline[0] == ".END": # stop when having '.END'
        break

    lineNo  += 1             # increase lineNo

    token = curline[0]
    if token not in keywords.keys(): # if it's not a operation, then it's a label
        symbols[token] = lineNo      # store into symbol table
        del curline[0]               # remove token from token list
        token = curline[0]

    if token == ".STRINGZ":          # there might be space & comma in the string
        begin = line.find('\"')      # pick up the entire string be tween ""
        end   = line.find('\"', begin+1)
        curline = [".STRINGZ", line[begin+1:end]]

    curline.insert(0, lineNo)
    todo.append(curline)             # store lineNo, tokenList into todoList

    # handle: .BLKW / .STRINGZ
    if   token == ".BLKW":
        lineNo += int(curline[2][1:]) - 1
    elif token == ".STRINGZ":
        lineNo += len(curline[2])-2 # token(string) has " on its both sides
```

2nd - Scan

> Just select some typical types

```python
# just part of this if-elif stucture
for line in todo:
    pc = line[0]+1  # PC = lineNo + 1
    token = line[1] # operation
    # pseudo ops
    if   token == ".ORIG":
        print(str2binN(line[2], 16)) # for .ORIG, just turn token2 into 16-bit binary
    elif token == ".BLKW":
        for i in range(int(line[2][1:])): # for .BLKW, just print N * x7777
            print(keywords[token])        # keywords[".BLKW"] = bin(x7777)
    elif token == ".STRINGZ":
        for ch in line[2]:                # for .STRINGZ, print bin(asc(str[idx]))
            print(str(bin(ord(ch)))[2:].rjust(16, '0'))
        print("0000000000000000")         # add '\0' to the end
    ...
    # calculations
    elif token in ["ADD", "AND"]:
        res = keywords[token] + regs[line[2]] + regs[line[3]] # opcode + dst + reg1
        # is arg3 reg/imm5 ?
        if line[4][0] == 'R':
            res += "000" + regs[line[4]]      # arg3 is a reg
        else:
            res += "1" + str2binN(line[4], 5) # arg3 is imm5
        print(res)
    ...
    # data-movement
    elif token in ["LD", "ST", "LDI", "STI", "LEA"]:
          res = keywords[token] + regs[line[2]] # opcode + reg1
          # is arg2 decimal/label?
          if line[3][0] == '#':
              res += str2binN(line[3], 9)       # arg2 is decimal number
          else:
              res += label2binN(pc, line[3], 9) # arg2 is label
          print(res)
   ...
    # traps
    elif token in ["TRAP", "GETC", "OUT", "PUTC", "PUTS", "IN", "PUTSP", "HALT"]:
        res = "11110000"
        if token == "TRAP":
            res += str2binN(line[2], 8)         # use 8-bit trap vector
        else:
            res += str2binN(keywords[token], 8) # get trap vector from dict(keywords)
        print(res)
    # branched
    elif token in ["BR", "BRn", "BRz", "BRp", "BRnz", "BRzp", "BRnp", "BRnzp"]:
        res = keywords[token]                  # get opcode + cond from dict(keywords)
        # is arg2 decimal/label?
        if line[2][0] == '#':
            res += str2binN(line[2], 9)
        else:
            res += label2binN(pc, line[2], 9)
        print(res)
    # JMPs
    elif token == "JMP":
        print("1100000" + regs[line[2]] +"000000") # jump to Rx
    elif token == "RET":
        print("1100000111000000")                  # jump to R7
    # JSRs
    elif token == "JSR":
        res = "01001"
        # is arg2 decimal/label?
        if line[2][0] == '#':
            res += str2binN(line[2], 11)
        else:
            res += label2binN(pc, line[2], 11)
        print(res)
    elif token == "JSRR":
        print("0100000" + regs[line[2]] + "000000")
    ...
```