Lab6-a LC-3 Assembler Report

- author: MiBoyu
- programming language: Python

Algorithm explanation

The progress of turning LC-3 assembly code to binary machine code consists of two passes. In the first pass, the assembler goes through the assembly code and generates a **symbol table**, which records all labels and their locations in the code. In the second pass, the assembler goes through the assembly code again. For each line of the code, it extracts the instruction, registers, immediate values, and offsets from the line, then transforms the labels into offsets according to the symbol table, and generates the machine code finally.

Obtain the beginning address

Using the `split()` method of string class of Python. The first line of assembly code is guaranteed to be `.ORIG vect16` instruction. For example, if the first instruction is `.ORIG x3000'`, after using `split()`, we get a list like

```
['.ORIG', 'x3000']
```

The last element of this list is a hexadecimal number starting with `x`. To get its value, we can add a `"0"` to it and use `eval()` function to get it value from the hexadecimal number starting with `0x`.

Generate the symbol table

Get all labels

Since no label occupies a separate line, for a line `s` in the assembly code, `s.split()` returns a list which contains its label(if exists) and instructions.

For example, if `s1` is a line of the assembly code with a label, and `s1` is `"LABEL ADD R0, R0, #0"`, then `s1.split()` will be

```
['LABEL', 'ADD', 'R0,', 'R0,', '#0']
```

If `s2` is a line of the assembly code without a label, `s2` is `"ADD R0, R0, #0"`, then `s1.split()` will be

```
['ADD', 'R0,', 'R0,', '#0']
```

Since there is at most one label for one instruction, the first element of the list is either a label or an instruction. Labels can not be instruction names or numbers, so if the first element is not in the instruction set of LC-3, it must be a label.

Get a label's address

Except instrution `.BLKW` and `.STRINGZ`, each other instrution only occupies one location in memory. The number of locations occupied by `.BLKW` and `.STRINGZ` is corresponding to its operand.

The operand of `.BLKW` instruction is a `#` and an unsigned decimal number. We can get the string of decimal number by `split()` function, and get the value by its slice from the second character to the end. For example, if the string `s` of the unsigned decimal number is `'#77'`, the slice `s[1:]` will be `'77'`, and its value is `eval('77') = 77`.

The operand of `.STRINGZ` is a string. To get the string's length, we find the first `"` character and the last `"` character in the line, characters between them is the string's contents. The number of memory locations it occupies is `length of the string + 1` (`0` in the end occupies a location).

In the beginning, `PC` is set to the beginning address, and after the assembler goes through a line, `PC` will increase by the number of memory locations it occupies. When a label is found, the current `PC` is its address. The symbol table is represented as a map. The map's key is labels and its value is addresses.

Assemble

Remove labels

First remove all labels from the original assembly code. For each line with a label, we have got the list after `.split()`, so we can combine them to a new string without the label. The method is add a space after each word in the list, ignoring the label.

For example, the list afer `split()` is:

```
['LABEL', 'ADD', 'R0,', 'R0,', '#0']
```

Ignore the label:

```
['ADD', 'R0,', 'R0,', '#0']
```

Add spaces and turn it to a string:

```
"ADD R0, R0, #0 "
```

Match registers, immediates, offsets, vectors

Use the `re` module to match all registers, immediates, offsets in a line.

- get registers

  The following regular expression can get all strings that starting with "R" and a digit between 0 and 7 following. So it can find all registers in a string `s`.

  ```
  reg = re.findall(r'R[0-7]', s)
  ```

  `reg` will be a list containing all registers in string `s`.

  For example, if `s` is `"ADD R0, R1, R2"`, `reg` will be `['R0', 'R1', 'R2']`.

- get immediates

  The immediates can be `#` decimal number or `x` hexadecimal number and they may be negitive.

  The following regular expression can get decimal numbers that starting with "#" and some characters between '0' and '9' or

  '-' following. The length of the substring following '#' is between 1 and 6: `#0` is the kind of shortest and `#-65536` is the kind of longest.

  ```
  data = re.findall(r'#[0-9-]{1,6}', s)
  ```

  `data` will be a list containing decimal immediates in string `s`.

  For example, if `s` is `"ADD R0, R1, #-65536"`, `data` will be `['#-65536']`.

  Then its value can be obtained by `eval(data[0][1:])`, which is talked before.

  The following regular expression can get hexadecimal numbers that starting with "x" and some characters between '0' and '9' or 'a' and 'f' or 'A' and 'F' or '-' following. The length of the substring following 'x' is between 1 and 5: `x0` is the kind of shortest and `x-FFFF` is the kind of longest.

  ```
  data_hex = re.findall(r'x[-0-9a-fA-F]{1,5}', s)
  ```

  `data_hex` will be a list containing hexadecimal immediates in string `s`.

  For example, if `s` is `"ADD R0, R1, x-FFFF"`, `data` will be `['x-FFFF']`. The first element, `data_hex[0]` of it is the hexadecimal immediate.

  To get its value, first check if its second character is '-'. If so, the immediate is negative, the value will be `-1 * eval("0x" + data_hex[0][2:])`, which ignores the 'x' and '-' and adds "0x" to it to obtain its value by `eval()`, then it times -1 to keep it negetive. If its second character is not '-', the value will be `eval("0" + data_hex[0][1:])`.

- get theoffset

  The label or offset always appears at the end of a line, so we can get the string of label or offset of line `s` by `s.split()[-1]`.

  For example, if `s` is `"STI R0, LABEL"`, then `s.split()` will be `['STI', 'R0,', 'LABEL']`, so `s.split()[-1]` will be `'LABEL'`. Moreover, if `s` is `"BRnzp #7"`, then `s.split()` will be `['BRnzp', '#7']`, so `s.split()[-1]` will be `'#7'`.

  Labels can start with: letters `A-Z` `a-z`, underlines `_`, so we can distinguish labels and offsets according to its first character. If the first character is `#`, then it is a offset, we can get its value by `eval()`, else it is a label. The address of a label can be get through the symbol table. Then the offset will be `address of label - PC`.

- get the vector

  The vector is always at the end of a line, so we can get it like getting the label and offset. The method to get the value of a hexadecimal number has been talked.

Generate the machine code

Reset `PC` to the starting address, then when a line is to be assembled, `PC` increaseds by 1.

After `split()` , the first element of returned list is the instruction, then machine code can be generated according to it.

- `ADD` or `AND`

  If we get 3 registers R_i, R_j, R_k, the machine code is

  ```
  (opcode << 12) | (i << 9) | (j << 6) | k
  ```

  If we get 2 registers R_i, R_j and the immediate `imm`, the machine code is

  ```
  (opcode << 12) | (i << 9) | (j << 6) | (1 << 5) | imm & 0x1f # reserve 5 bits of imm
  ```

  If the instruction is `ADD`, the opcode will be 1, if the instruction is `AND`, the opcode will be 5.

- `NOT`

  We can only get 2 registers: R_i, R_j, the machine code is

  ```
  (9 << 12) | (i << 9) | (j << 6) | 0x3f # the last 7 bits are all 1
  ```

- `LD`, `ST`, `LDI`, `LEA`, `STI`

  In these instructions, we will get a register, R_i and an offset `ofs`. The machine code is

  ```
  (opcode << 12) | (i << 9) | ofs & 0x1ff # 9-bit offset
  ```

- `LDR` or `STR`

  In these instructions, we will get 2 registers R_i, R_j and an offset `ofs`. The machine code is

  ```
  (opcode << 12) | (i << 9) | (j << 6) | imm & 0x3f # 6-bit offset
  ```

  If the instrcution is `LDR`, the opcode is 6. If the instrcution is `STR`, the opcode is 7.

- `TRAP`, `BR`, `JSR`

  In these instructions, we will get an offset `ofs` or an immediate `imm` or a vector `vec`. They are all at the end of line, so their value can be obtained in the same way as offset. To simplify the expression, let `ofs` be their values. Then the machine code is

  ```
  (opcode << 9) | (ofs & 0x1ff)
  ```

  `TRAP`'s opcode is 0, `JSR`'s is 4, `BR`'s opcode lies on it's condition.

- `JMP`, `JSRR`, `RET`

  In these instructions, we will get a register, R_i. The machine code is

  ```
  (opcode << 12) | (i << 6)
  ```

`JMP`'s opcode is 12, `JSRR`'s is 4. `RET` is the special case of `JMP` that the register is R_7

- `.ORIG`, `.FILL`

  Get the value of the immediate or the vector by the same way as offset. The machine code is the value itself.

- `.RTI`

  The machine code is `1000 0000 0000 0000`

- `.BLKW`

  Get the value of the immediate `n` by the same way as offset. Then the machine code is n consecutive `0x7777`s.

  Then the `PC` should increase by `n-1`, so that it actually increases by `n`.

- `.STRINGZ`

  Find the first `"` character and the last `"` character in the line, then get the length of string `L`. The machine codes are the ASCII codes of all characters, and a `0` at the end.

  Then the `PC` should increase by `L`, so that it actually increases by `L+1`.

- `GETC` , `OUT`, `PUTS`, `IN`, `PUTSP`, `HALT`

  The machine code is fixed.

Generate the result and output

After a line is assembled, add the machine code(s) to `res` list. In the end, print only one machine code in 16-bit binary format in a line.

Essential parts of code

map register or label to number:

```
def reg2num(self, s):
    return self.register_to_num[s] # map R0-R7 to 0-7
def label2num(self, s):
    return self.label_list[s] # map label to its address
```

Get immediates, offsets and registers

```
def assemble(self, s):
    #assemble a single line
    word_list = s.split()
    inst = word_list[0] # instruction
    reg = re.findall(r'R[0-7]', s) # get registers
    data = re.findall(r'#[0-9*-]{1,7}', s) # get decimal imm
    data_hex = re.findall(r'x[-0-9a-fA-F]{1,5}', s) # get hexadecimal imm
    label = word_list[-1] # get label
    if label in self.label_list:
        ofs = self.label2num(label) - self.PC # get offset from label
    elif label[0] == '#':
        ofs = eval(label[1:]) # get the value from decimal number
    elif label[0] == 'x': # get the value from hexadecimal number
        if len(label) > 2 and label[1] == '-':
            ofs = -1 * eval("0x" + label[2:]) #negative
```

```
        else:
            ofs = eval("0" + label)  #positive
            if len(data): #there is a decimal immediate
                imm = eval(data[0][1:])
            elif len(data_hex): #there is a hexadecimal immediate
                if len(data_hex[0]) > 2 and data_hex[0][1] == '-':
                    imm = -1 * eval("0x" + data_hex[0][2:])
                else:
                    imm = eval("0" + data_hex[0])
```

process `.BLKW` and `.STRINGZ` instruction

```
if inst == ".BLKW":
    for i in range(ofs):
        self.res.append(0x7777) # fill 0x7777
        self.PC += 1
    self.PC -= 1 # make sure the increment of PC is ofs

elif inst == ".STRINGZ":
    head, tail = 0, len(s) - 1
    while s[head] != '\"':
        head += 1 # the head of string
    while s[tail] != '\"':
        tail -= 1 # the tail of string
        for c in range(head + 1, tail):
            self.res.append(ord(s[c]))
            self.PC += 1
        self.res.append(0) # 0 at the end
```

main function, input and output

```
def main():
    ASSEMBLER = Assembler() # create a Assembler object
    lst = []
    s = input().strip().split()
    word_list = s.split()
    data = eval("0" + word_list[-1]) # get the starting address
    ASSEMBLER.PC = data #set PC
    while 1:
        s = input().strip()     #input the assembly code, ignoring extra space
        if s != "": #ignore empty line
            if s.split()[-1] == ".END": #when the input is ".END", stop inputting
                break
            lst.append(s) # add to lst
    ASSEMBLER.generate_label(lst)
    ASSEMBLER.PC = data # reset PC
    for line in lst: # assemble line by line
        ASSEMBLER.PC += 1
        ASSEMBLER.assemble(line)
    for n in ASSEMBLER.res:
        print('{:0>16b}'.format(n & 0xffff)) # output in 16-bit binary format
```