COLORADO SCHOOL OF MINES
ELECTRICAL ENGINEERING & COMPUTER SCIENCE DEPARTMENT

CSCI-410
Elements of Computing Systems
Spring 2014

PY-04

The primary goal of this assignment is to familiarize you with Python dictionaries. To do this, you will implement a disassembler that takes HACK files and produces an annotated assembly code listing, which you should also find useful as you implement your own assembler.

Output File Format

The output of the disassembler should be a legal assembly language file that has the same name as the HACK file but with a .dis extension (to avoid overwriting the original .asm file should it be in the same directory). This file need not have a header (since it is not one of the extensions that the Header class will recognize). The format of each line should be as follows:

```
assembly_instruction // address: 0xHEXA machine_instruction
```

See the example listing at the end of this document for reference.

The assembly instruction should be a legal instruction using the following format for C-type instructions:

```
[AMD=]OPC[;JMP]
```

Since this file is primarily for debugging purposes, it is helpful for each part of the instruction to be in well-defined fields (columns) in the listing. Thus, the dest will consist of three characters, {AMD}, in that order in columns 1-3 with each character being replaced by a space as appropriate. The equals sign, if any, will be printed in column 5. The OPC is the instruction mnemonic, which is never longer than three characters. It should be left justified starting in column 7. If a jump field is present, the semicolon should be located in column 12 and the three-character jump mnemonic starting in column 14.

For A-type instructions, the format should be

```
@ integer
```

The '@' symbol should be in column 5 (i.e., the same column as the C-type instruction's '=' symbol) with the decimal integer left-justified (no leading zeros) starting in column 7 (i.e., directly above the C-type instruction's OPC mnemonic).

Each line should be followed by a comment that gives the instructions address, the hexadecimal representation of the instruction, and a parsed version of the machine instruction itself. The comment delimiters should start in column 21. The address should be printed, following a single space, as a five-digit integer with leading zeros followed immediately by a colon (which should place the colon in column 29). Following a space, the four-digit uppercase hexadecimal representation of the instruction (with leading zeros, if needed) should appear surrounded by square brackets (this should place the closing square bracket in column 36.

The remainder of the comment, following a single space after the hex representation, is a parsed version of the machine instruction. While the details of the parsing depend on the type of instruction, both types require twenty characters which should place the last character on the line in column 57.

The two formats for the machine instruction are as follows:

A-Type: **`0 xxx xxxx xxxx xxxx`**

C-Type: **`111 a cccccc ddd jjj`**

The Main, FileSet, and Util classes

The Main, FileSet, and Util classes for this project are already written for you. You may modify FileSet and Util to suit your purposes, but you are to use the Main class untouched. Note that the grading script will may replace the one in your submission ZIP file with a clean copy.

If you use these classes untouched, then do not alter their headers, either. But, if you do modify one of them, update the header with your information, put a disclaimer comment after the header indicating that it is a modified file based on one provided to you (similar to the disclaimer in the HDL files), and indicating what the modifications were.

Dictionaries

You might be asking how dictionaries fit into all of this. To disassemble the C-type instructions, you will use three dictionaries, one each for the dest, the comp, and the jump fields. The dest and jump dictionaries will have eight entries while the comp dictionary will have twenty-eight. Simply put, the keys to the dictionaries will be the binary patterns read from the machine code file and the values retrieved will be the mnemonics.

You will implement two new classes in Python for your disassembler: Parser and Code. The Parser object is given an input stream (in other words, open a .hack file and pass the file object to the Parser's constructor). The Parser reads lines from the input file and provides information about the line such as what type of instruction it is, what the address is, and what the various fields are.

It might be useful to note that the Parser object knows about the hardware architecture in so much as how it parses the machine instructions, but it knows nothing about the assembly language format. The Code object, on the other hand, knows how assembly language mnemonics are mapped to bit sequences, but it does not know how those bit sequences are put together to form a complete machine instruction. This partitioning would make it possible to use a different assembly language with the same hardware platform or the same assembly language with different hardware platforms, provided the two were reasonably compatible. For instance, it would not be too hard to imagine a HACK with a 32-bit wide data path that simply used 32-bit wide instructions for convenience and organized the bits within the instructions differently. Alternatively, it would be quite reasonable to implement a more traditional assembly language style that still targets the HACK platform. Both of these are actually quite common in the embedded microcontroller world given the high number of variants on the same basic platform as well as several different assembly language "camps" that exist.

Command Line Format

Subject to the specifics of how you need to invoke Python scripts on your platform, the command line usage should be:

```
prompt> unHack (filename|directory)
```

As before, the user will do one of the following actions:

1) From within the directory containing the file, run the program passing it a filename (including extension).

2) From the directory immediately above the files to be modified, run the program passing it the directory name. The program will then work through all of the .hack

There are no command line options for this project.

Invalid Instructions

There are two types of invalid instructions. The first are instructions that are neither A-type nor C-type. Any instruction that begins with a leading 0 bit is an A-type instruction, but only instructions that begin with three leading 1 bits are C-type instructions. Any instruction that does not fall into one of these two categories is an invalid instruction type.

The second type of invalid instructions are those that are C-type instructions but which do not map to valid computation mnemonic. While these are not valid in a strictly conforming sense as far as the defined assembly language instructions are concerned, they are, in fact, valid instructions as far as the Hack hardware is concerned. Thus, you will generate a new type of computation mnemonic, called an 'X' instruction, which consists of three characters. The first is the letter 'X' and the last two are the zero-padded, uppercase hexadecimal digits corresponding to the seven bits in the instruction.

Static Methods

A method that can be called independent of any instance object of that class is known as a "class method" and the proper way to declare such a method is as a static method as follows:

```
class Fred:

    @staticmethod
    def aStaticMethod(arguments):
        …

    def anInstanceMethod(self, arguments):
        …
```

Notice that static methods not only are preceded by the "**`@staticmethod`**" decorator. In addition, they do not include the initial implied "self" argument. To call a static method you use the name of the class instead of the name of an instance object of that class.

Parser class specification

Routine	Arguments	Returns	Description
constructor	file (stream)	--	Prepares to read the .hack file bound to the stream.
hasMoreInstructions	--	Boolean	True if more instructions are yet to be processed.
advance	--	--	Make the next instruction the current instruction.
address	--	integer	Returns the address of the current instruction.
instruction	--	string	Returns the entire instruction as a sixteen bit string of '0' and '1' characters.
hexInstruction	--	string	Returns the instruction as a four digit hexadecimal value.
parsedInstruction	--	string	Returns a string the represents the instruction parsed appropriately for its type.
instructionType	--	"A_TYPE" "C_TYPE" "INVALID"	Returns the type of the instruction.
value	--	integer	Returns the integer value associated with an A-type instruction.
dest	--	string	Returns the instruction fragment associated with the destination field of a C-type instruction.
comp	--	string	Returns the instruction fragment associated with the computation field of a C-type instruction.
jump	--	string	Returns the instruction fragment associated with the jump field of a C-type instruction.

As you can see, only the constructor takes an argument. All other functions work with the current instruction (or, in some cases, relative to the current instruction). Here are more detailed requirements for each public method in the Parser class.

`constructor`

The argument is a file (or stream) object for the machine code file (the .hack file) containing the machine instructions to be disassembled.

hasMoreInstructions

This method returns True if there are still more instructions to be disassembled.

advance

Makes the next instruction the current instruction.

address

This method returns the address of the current instruction. The address of the first instruction is 0 and instructions are stored at sequential addresses thereafter.

hexInstruction

This method returns the four-digit string corresponding to the zero-padded hexadecimal representation of the current instruction. All alphabetic digits are uppercase.

parsedInstruction

This method returns the 20-character parsed representation of the current instruction.

A-Type: **0 xxx xxxx xxxx xxxx** C-Type: **111 a cccccc ddd jjj**

Invalid: **xxxxxxxxxxxxxxxx**

instructionType

This method returns the type of the current instruction using one of the following strings:

"**A_TYPE**", "**C_TYPE**", or "**INVALID**"

value

This method returns the integer value loaded by an A-type instruction.

dest

This method returns the three-bit string encoding the destinations in the order {A, D, M}.

comp

This method returns the seven-bit string encoding the computation as {a c1…c6}.

jump

This method returns the three-bit string encoding the jump conditions as {LT, EQ, GT}.

Code class specification

Routine	Arguments	Returns	Description
a_type	integer	string	Returns an entire A-type instruction.
c_type	dest (string) comp (string) jump (string)	string	Returns the entire C-type instruction given the three mnemonics that define it.
destMnemonic	dest (string)	string	Returns the destination mnemonic give the three-bit pattern that encodes it.
compMnemonic	comp (string)	string	Returns the computation mnemonic give the seven-bit pattern that encodes it.
jumpMnemonic	jump (string)	string	Returns the jump mnemonic give the three-bit pattern that encodes it.
invalid_type	--	string	Returns "*** INVALID ***"

Note that the Code class has no constructor; in fact, you do not instantiate objects of the Code class at all. Like the Util class, it contains static class methods.

Here are more detailed requirements for each public method in the Code class.

`a_type`

This method returns the A-type instruction that loads the given integer.

`c_type`

This method returns the C-type instruction corresponding to the three components given.

`destMnemonic`

This method returns the destination mnemonic given the three-bit string that encodes it.

`compMnemonic`

This method returns the computation mnemonic given the seven-bit string that encodes it. If the string does not map to a valid mnemonic, then it should return "XHH" where HH is the two-digit, zero padded hexadecimal representation of the seven-bit string.

`jumpMnemonic`

This method returns the jump mnemonic given the three-bit string that encodes it.

`Invalid_type`

This method returns the string "**`*** INVALID ***`**".

Example Output File

The following is the disassembler output for file mult.hack from ECS-04.

```
    @ 1             // 00000: [0001] 0 000 0000 0000 0001
  D = M             // 00001: [FC10] 111 1 110000 010 000
    @ 4             // 00002: [0004] 0 000 0000 0000 0100
M   = D             // 00003: [E308] 111 0 001100 001 000
    @ 2             // 00004: [0002] 0 000 0000 0000 0010
M   = 0             // 00005: [EA88] 111 0 101010 001 000
    @ 3             // 00006: [0003] 0 000 0000 0000 0011
M   = 1             // 00007: [EFC8] 111 0 111111 001 000
    @ 3             // 00008: [0003] 0 000 0000 0000 0011
  D = M             // 00009: [FC10] 111 1 110000 010 000
    @ 28            // 00010: [001C] 0 000 0000 0001 1100
      D    ; JEQ    // 00011: [E302] 111 0 001100 000 010
    @ 0             // 00012: [0000] 0 000 0000 0000 0000
  D = D&M           // 00013: [F010] 111 1 000000 010 000
    @ 20            // 00014: [0014] 0 000 0000 0001 0100
      D    ; JEQ    // 00015: [E302] 111 0 001100 000 010
    @ 4             // 00016: [0004] 0 000 0000 0000 0100
  D = M             // 00017: [FC10] 111 1 110000 010 000
    @ 2             // 00018: [0002] 0 000 0000 0000 0010
M   = D+M           // 00019: [F088] 111 1 000010 001 000
    @ 3             // 00020: [0003] 0 000 0000 0000 0011
  D = M             // 00021: [FC10] 111 1 110000 010 000
M   = D+M           // 00022: [F088] 111 1 000010 001 000
    @ 4             // 00023: [0004] 0 000 0000 0000 0100
  D = M             // 00024: [FC10] 111 1 110000 010 000
M   = D+M           // 00025: [F088] 111 1 000010 001 000
    @ 8             // 00026: [0008] 0 000 0000 0000 1000
      0    ; JGE    // 00027: [EA83] 111 0 101010 000 011
    @ 28            // 00028: [001C] 0 000 0000 0001 1100
      0             // 00029: [EA80] 111 0 101010 000 000
```