CSE4100-1: System Programming (Spring 2019)
Project #1: Shell Environment

Handed out: March 7, Due: March 25, 11:59PM (KST)

1. Goal

This project is a preliminary step to implementing a SIC/XE machine. In this project, we are going to implement a shell environment that executes user commands, a memory space where object code will be loaded, and an opcode table which translates mnemonics to opcodes.

2. Requirements

2.1. Overview

- When you run your program, a shell prompt should appear on the screen, waiting for your command. The shell prompt should look like the following:

```
sicsim>
```

- The shell should accept various kinds of user commands and execute them. The shell should support the following commands. Each command will be explained later.

· shell related commands: **h**elp**, d**ir**, q**uit**, hi**story

· memory related commands: **du**mp**, e**dit**, f**ill**, reset**

· opcode related commands: **opcode, opcodelist**

- Your program should create a memory space of size 1MB, and an opcode table which must be implemented using a hash table.

2.2. User-Command Specification

2.2.1. Shell related commands

(1) h[elp] ☜ user can enter 'h', or 'help'.
This command will display all commands supported by the shell.

Example:

```
sicsim> help
```

```
h[elp]
d[ir]
q[uit]
hi[story]
du[mp] [start, end]
e[dit] address, value
f[ill] start, end, value
reset
opcode mnemonic
opcodelist
```

※ For this command and all other commands, the shell prompt should appear again once the program executes one command. (The program should not end after executing one command.)

(2) d[ir]
This command shows the files in the current directory.

※ You must NOT implement this command using system call functions such as system() or exec(). Tips: include **dirent.h** and **sys/stat.h**.
※ You may or may not include `.` and `..` in your output. (If you run `ls`, . and .. are shown with other files.)
※ You should output "/" next to the name if it is a directory, and "*" if the file is an executable file. (Refer to the following example.)

Example:

```
sicsim> dir

Desktop/    Work/       dead.letter mail/
Mail/       a.out*      Ingabi/
```

(3) q[uit]
This command will exit the program.

(4) hi[story]
This command will show the previous commands. The latest command must be displayed at the end of the list.

Example:

```
sicsim> history
```

```
1     dump
2     dump 14, 37
3     e 14, E3
...   ☜ the program should display the full list.
908   reset
909   d
910   history
```

※ In order to support this command, the program should keep track of all previous user commands. You must implement this using a **linked list**.

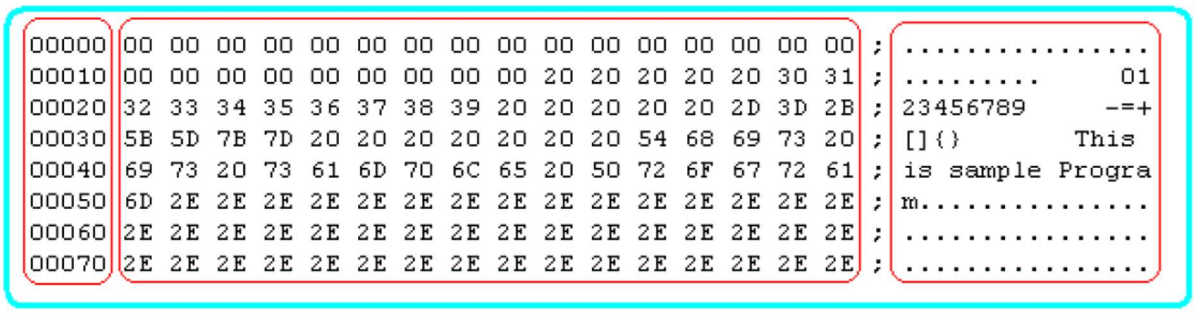
※ When you run **history**, that is also a command, so it should be displayed at the end of the list.

※ Invalid commands are NOT included in the list of previous commands.

2.2.2. Memory related commands

In later projects, we will be creating object files and load them onto a memory space. Here, we create a memory space of 1Mbytes and implement commands related to the memory space.

(1) du[mp] [start, end]
Output the contents of the memory space. The output format should be as below.



```
00000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 36 37 38 39 20 20 20 20 20 2D 3D 2B ; 23456789     -=+
00030 5B 5D 7B 7D 20 20 20 20 20 20 20 54 68 69 73 20 ; []{}       This 
00040 69 73 20 73 61 6D 70 6C 65 20 50 72 6F 67 72 61 ; is sample Progra
00050 6D 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E ; m...............
00060 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E ; ................
00070 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E ; ................
```

The leftmost column shows the memory address. The numbers are **5-digit hexadecimal** numbers.

The middle column shows the content of memory in hexadecimal numbers. The alphabets in hexadecimal numbers should be in **capital letters**.

The rightmost column shows the memory contents in **ASCII code**. The ASCII code should be printed for bytes that are in the range of 20 – 7E (in hexadecimal number). Otherwise, '**.**' should be printed.

This command has optional arguments. In the following, how the program should execute the command in various cases is described.

- case 1: no argument

```
sicsim> dump
```

· Print 160 bytes of memory. One line consists of 16 bytes. (Refer to one of the figures.)

· The program should remember the last address that was printed. If the user executes **dump** again, the output should start from (last address + 1).

· When the user executes **dump** for the first time, the starting address is 0.

· If the last address goes over the boundary (0xFFFFF), stop at the last address.

· If the last address is 0xFFFFF and the user executes dump, the output starts from 0.

- case 2: 1 argument (start)

```
sicsim> dump start
```

· Print 160 bytes of memory from the `start` address.

· The program should remember the last address that was printed. Later if the user executes **dump**, the output should start from (last address + 1).

· If the last address goes over the boundary (0xFFFFF), stop at the last address.

· If the `start` address is not within the boundary (0x00000 – 0xFFFFF), output an error message.

- case 3: 2 arguments (start, end)

```
sicsim> dump start, end
```

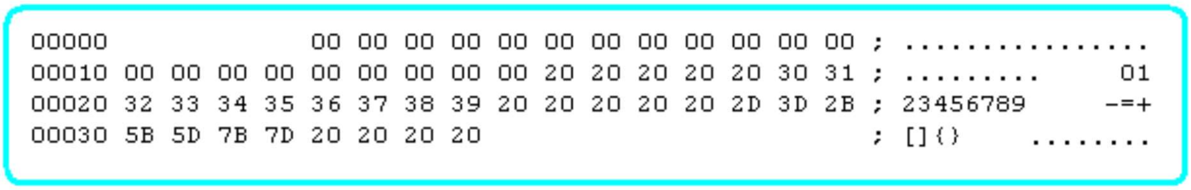
· Print memory from the `start` address to the `end` address.

· If the `start` address or the `end` address is not within the boundary, output an error message.

· If the `start` address is larger than the `end` address, output an error message.

Example:

```
sicsim> dump 4, 37
```



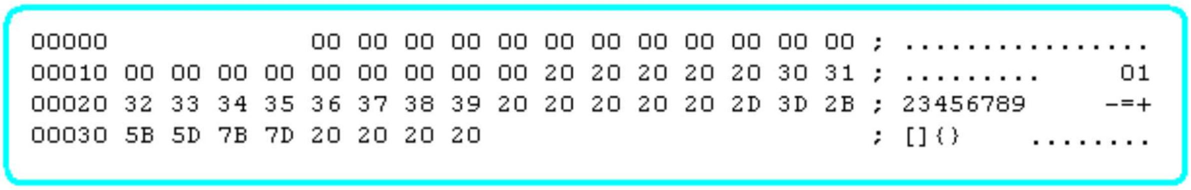
```
00000             00 00 00 00 00 00 00 00 00 00 00 00 ;     ............
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 36 37 38 39 20 20 20 20 20 2D 3D 2B ; 23456789     -=+
00030 5B 5D 7B 7D 20 20 20 20                         ; []{}    ........
```

※ Since we are only showing memory from 4 to 37, the memory contents of 0-3 and 38-3F are left as blank in the middle column. However, in the rightmost column, they must be printed as '.'

(2) e[dit] address, value
This command sets the memory content at `address` to `value`.

Example:

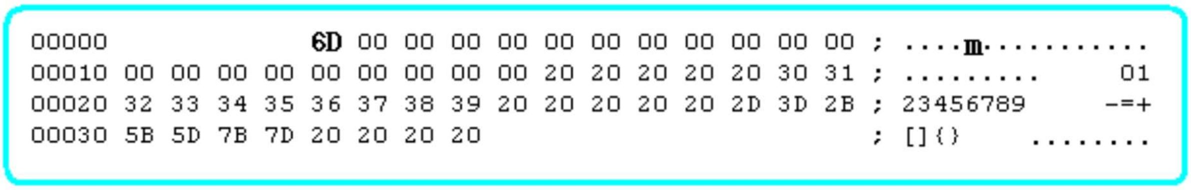
```
sicsim> dump 4, 37
```



```
00000             00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 36 37 38 39 20 20 20 20 20 2D 3D 2B ; 23456789     -=+
00030 5B 5D 7B 7D 20 20 20 20                         ; []{}    ........
```

```
sicsim> e 4, 6D

sicsim> du 4, 37
```

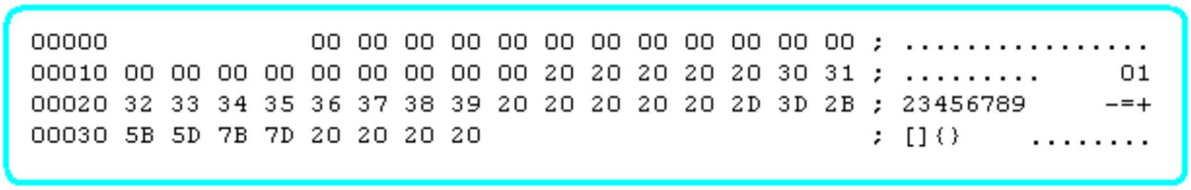


```
00000             6D 00 00 00 00 00 00 00 00 00 00 00 ; ....m...........
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 36 37 38 39 20 20 20 20 20 2D 3D 2B ; 23456789     -=+
00030 5B 5D 7B 7D 20 20 20 20                         ; []{}    ........
```

(3) f[ill] start, end, value
This command sets the memory contents from `start` to `end` to `value`.

Example:

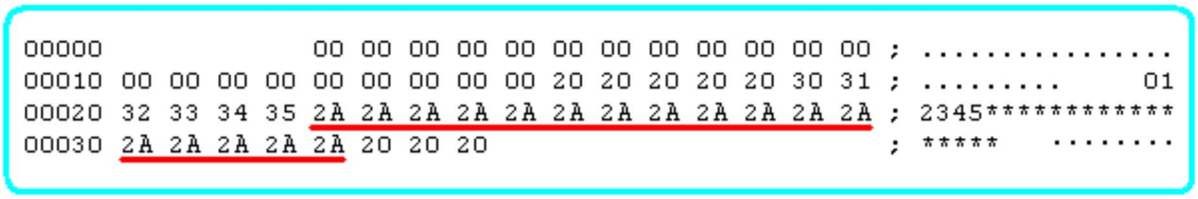
```
sicsim> du 4, 37
```



```
00000             00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 36 37 38 39 20 20 20 20 20 2D 3D 2B ; 23456789     -=+
00030 5B 5D 7B 7D 20 20 20 20                         ; []{}    ........
```

```
sicsim> f 24, 34, 2A

sicsim> du 4, 37
```

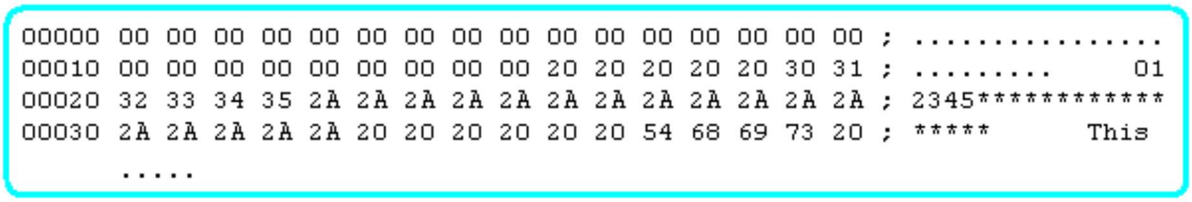


```
00000                00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A ; 2345************
00030 2A 2A 2A 2A 2A 20 20 20                         ; *****   ........
```

(4) reset
This command sets contents of the entire memory to zero.

Example:

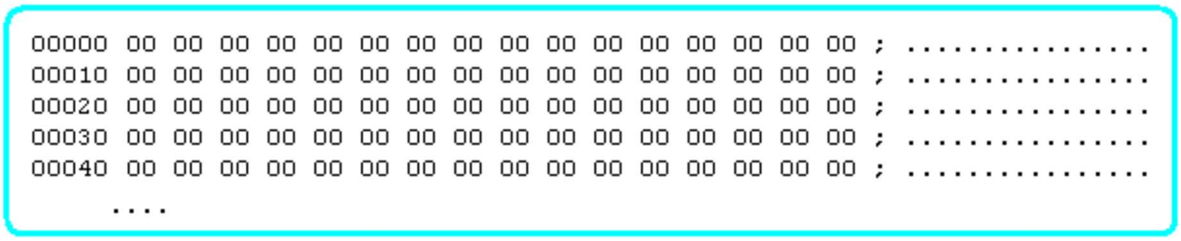
```
sicsim> du
```



```
00000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00010 00 00 00 00 00 00 00 00 00 20 20 20 20 20 30 31 ; .........     01
00020 32 33 34 35 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A ; 2345************
00030 2A 2A 2A 2A 2A 20 20 20 20 20 20 54 68 69 73 20 ; *****      This
      .....
```

```
sicsim> reset

sicsim> du 0
```



```
00000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
      ....
```

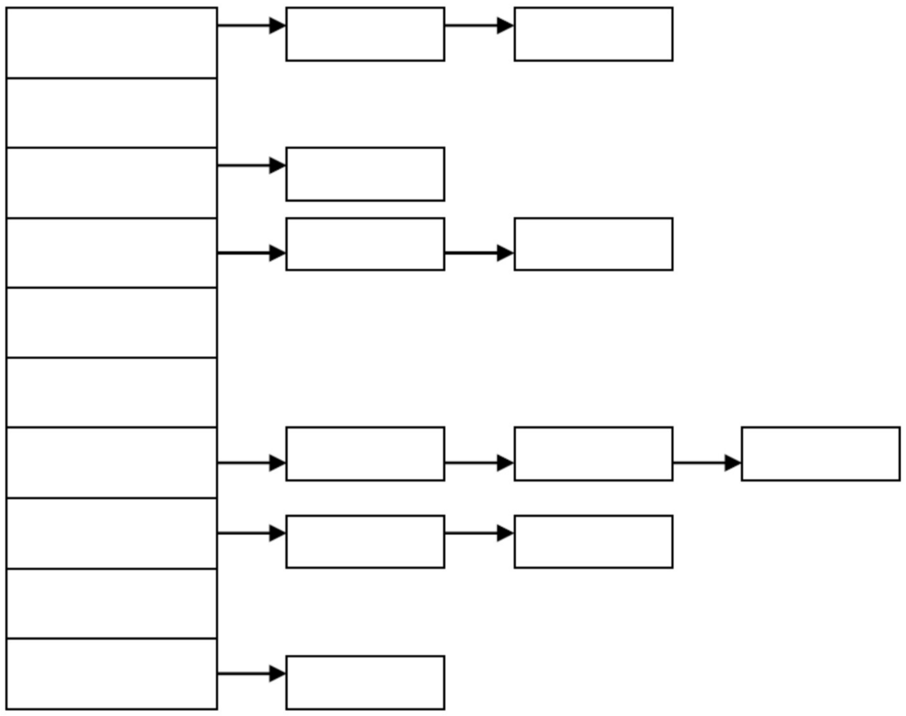
2.2.3. Opcode related commands

- In order to translate a SIC source code to an object code using an assembler, we need to be able to change mnemonics into opcodes.

- The opcode table is in Appendix A of the textbook, and also is given out as a text file (`opcode.txt`). You should use this file. The file is in the following format. Note that all the instruction names are in capital letters.

```
18     ADD    3/4
58     ADDF   3/4
40     AND    2
...    ...    ...
DC     WD     3/4
```

- When the program starts, the program should read from the `opcode.txt` file and create an opcode table in the form of a hash table. **The size of the hash table should be 20**. You may design your own hash function.



- Since the number of mnemonics is much larger than the hash table size, there will be hash collisions. In that case, the mnemonics that are hashed into the same bucket should be chained as a linked list as the above figure.

(1) opcode mnemonic
This command shows the opcode of the corresponding mnemonic.

Example:

```
sicsim> opcode ADD
opcode is 18.

sicsim> opcode LDB
opcode is 68.
```

(2) opcodelist
This command outputs the opcode hash table in the following format. (Note that your output may be different from the example below, because of difference in the hash functions.)

Example:

```
sicsim> opcodelist

0 : [ADD, 18] -> [JEQ, 30]
1 : [STS, 7C] -> [LDS, 6C]
...
19 : [LPS, D0]
```

2.3. Other Requirements

- The user commands are all in lowercase letters.

- Arguments to the commands are all in hexadecimal numbers. The alphabets in hexadecimal numbers could be either in uppercase or lowercase letters.

- For invalid inputs (wrong commands, wrong arguments, wrong format, arguments that are outside the valid range, etc.), the program should deal with them properly, such as displaying proper error messages. The program must not crash.

- The acceptable and unacceptable input formats (regarding spaces) are as follows. The program should accept all acceptable formats.

```
(o) dump AA, AB
(o) dump AA  ,   AB
(o) dump AA,AB
(x) dump AA AB
(x) dumpAA, AB
```

(When there are two arguments, the arguments are separated by a comma.)

- Your program should be successfully compiled using **gcc**.

- When compiled, your program should not generate any warnings. You can use gcc –Wall option to check for all warnings.

3. Submission

3.1. Things to Submit

(1) program source: you must document your source code so that others can understand what each part of code is doing.

(2) Makefile: you should write a Makefile that will compile your source. The binary file should be named using your student ID (explained later.)

(3) A document file: this document file should describe how you implemented your program. When you write your document file, think like this: a new employee needs to take over your work and extend your program. He/she should be able to understand your code and be able to work on it after reading your document. A sample document will be posted on cyber campus.

(4) A README file that includes simple instructions to compile and run the program.

(5) opcode.txt: the file used as an input to the program.

3.2. Instructions on Submission

- Make a directory named "sp20161234_proj1". The numeric part should be **your student ID**.

- Put all the files in the directory, and compress the directory itself using **tar**.

- Do NOT put binary files in the directory.

- For the source files, you may have one or more c files. Use your student ID as the name for the c file that contains the main function.

- You should have at least one header file. Use your student ID as the name for one of the h files.

- Write the Makefile so that the output binary file is named "20161234.out". The numeric part should be **your student ID**.

- When you make a tar file, do NOT use the z option (which makes a gz compressed file.)

Example:

```
sp20161234_proj1/

  README
  document.docx
  20161234.c        ☜ This file contains the main function.
  20161234.h
  Makefile
  opcode.txt
```

The file for submission

```
sp20161234_proj1.tar
```

upload this file on the **cyber campus**.

3.3. List for Self-Check before Submission

(1) Did you implement everything specified in the requirements (including specific instructions for what data structure to use)?
(2) Does your program compile successfully using gcc?
(3) Does your program compile without giving any warnings?
(4) Does your program run all the commands correctly?
(5) Does your program handle exceptional cases such as wrong inputs?
(6) Did you name all your files based on given instructions?
(7) Do you have at least one header file?
(8) Did you write a Makefile?
(9) Did you write a README.
(10) Did you document your source code properly?
(11) Did you write a report document?
(12) Did you include everything necessary in your tar file?

3.4. Late Submissions

Late submissions are accepted for five days after the deadline. 10% of the points are deducted for each day. Submissions are not accepted after five days.