**과목명: 시스템프로그래밍**

**CSE4110-02**

**<<Project #1>>**

**서강대학교 [공학부 컴퓨터공학과]**

**[20171662]**

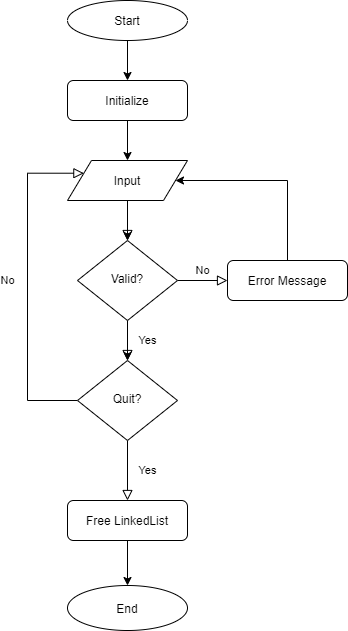
**[이나연]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 20171662.c
      1. main()
      2. isHex()
      3. right\_input\_form()
   2. shellcmd.c
      1. printdir()
      2. insertNode()
      3. ShowAll
      4. FreeAll
   3. memcmd.c
      1. print\_mem()
      2. onlydump()
      3. dumpstart()
      4. dumpstartend()
      5. editvalue()
      6. fillvalue()
      7. reset\_mem()
   4. opcmd.c
      1. init()
      2. destructor()
      3. search()
      4. add()
      5. show()
4. **전역 변수 정의**
   1. int end\_index
   2. char\* input
   3. char\* modified\_input
   4. Node\* head
   5. Opcode\*\* hashTable
   6. char helpprint[]
5. **코드 설명**
   1. #define 매크로
      1. MEM\_SIZE
      2. OPCODE
      3. TABLE\_SIZE
   2. 20171662.c
      1. main()
      2. isHex()
      3. right\_input\_form()
   3. shellcmd.c
      1. printdir()
      2. insertNode()
      3. ShowAll()
      4. FreeAll()
   4. memcmd.c
      1. print\_mem()
      2. onlydump()
      3. dumpstart()
      4. dumpstartend()
      5. editvalue()
      6. fillvalue()
      7. reset\_mem()
   5. opcmd.c
      1. init()
      2. destructor()
      3. search()
      4. add()
      5. show()
6. **기타 코드**
   1. 20171662.h
   2. makefile
7. **프로그램 개요**

해당 프로그램은 SIC/XE Machine을 구현하기에 앞서 Assembler, Link, Loader들이 실행되게 될 쉘(shell)과 컴파일을 통하여 만들어진 object 코드가 적재되고 실행될 메모리공간과 mnemonic (ADD, COMP, FLOAT, etc.)을 opcode값으로 변환하는 OPCODE 테이블과 관련 명령어들을 구현하는 프로그램이다.

1. **프로그램 설명**
   1. **프로그램 흐름도**



전체적인 프로그램의 흐름도는 다음과 같다. 먼저 프로그램이 실행되면, 추후 command들이 정상적으로 실행되기 위하여 필요한 여러 변수들에 메모리를 할당하여 주는 등의 Initialization을 실행한다. 그런 다음,

>>> sicsim> “input”

의 형태로 input을 받아 만약 해당 input이 valid하다면 해당 command의 기능을 수행하도록 하고, 만약 그렇지 않다면 적절한 에러 처리와 함께 다시 input을 새로 받는다. 만약 quit 이나 q에 해당하는 프로그램 종료에 관한 command가 들어오면, 할당된 메모리를 해제한 후 프로그램을 정상적으로 종료한다.

1. **모듈 정의**

해당 프로그램에선 효율적인 프로그램 구현을 위하여 프로그램을 총 4가지의 c파일로 분할하였다. 우선 main()함수가 포함되어 있는 20171662.c 파일과 더불어 Shell 관련 명령어의 수행을 돕는 기능이 담긴 shellcmd.c, 메모리 관련 명령어의 수행을 돕는 기능이 담긴 memcmd.c, 그리고 OPCODE TABLE 관련 명령어의 수행을 돕는 기능이 담긴 opcmd.c 파일이 해당 프로그램을 구성한다.

* 1. **20171662.c**

해당 파일은 위에서 언급했던 대로 main()함수가 내장되어 있는 파일이다. 프로그램 시작 후 main()함수 내에서 ‘sicsim>’과 같이 Unix shell과 유사한 입력 프롬프트 상태를 출력한 후 input을 받아 해당 기능을 수행하도록 각각의 명령어에 따라 적절한 함수를 매칭시켜준다. 만약 해당 input이 정해져 있는 명령어들 이외의 값이 들어왔을 때엔 에러 처리를 해주어 다시 input을 받게끔 한다.

* + 1. **main()**

해당 모듈은 대표적으로 명령 프롬프트에서 입력을 받는 역할을 수행한다. 한 줄 전체를 입력 받은 후, ‘,’, ‘ ‘, ‘\n’을 기준으로 tokenize하여 명령어의 각 인자를 분리한다. 만약 분리하여 얻은 인자가 주어진 명령어 중에 있지 않다면, 적절한 에러 문구를 출력한 후 다시 input을 받는다. 만약 주어진 명령어가 input으로 들어오게 된다면, 적절한 함수를 사용하여 해당 기능을 수행한다. 보통 모든 함수는 다른 파일에 내장되어 있지만, h[elp] 명령어 같은 경우 수행해야할 기능이 단순 출력이기 때문에 해당 파일 내에서 모든 기능 구현이 이루어진다. 이러한 대표적인 작업 이외에도 전역 변수에 대한 초기화 작업 (Linked List 구현을 위한 메모리 할당, opcode.txt를 입력 받아 OPCODE TABLE에 대한 hash Table 구성 등) 역시 수행한다.

* + 1. **isHex()**

해당 모듈은 가상 메모리의 address나 value값으로 들어온 인자가 16진수의 형태인지를 확인해준다. 메모리 관련 명령어인 dump, edit, fill 명령어들은 16진수의 인자를 받아 조건에 맞게끔 메모리를 변형하거나 출력해주어야 하기 때문에 이러한 16진수 확인은 필수적이다.

* + 1. **right\_input\_form()**

해당 모듈은 tokenize 된 input을 다시 원래의 ‘적절한’ input form으로 하나의 string으로 다시 붙여주는 역할을 한다. 예를 들어, ‘dump 4, 27’이라는 input을 ‘dump’, ‘4’, ‘27’의 세 부분으로 나누어준 후 history linked list에 들어갈 input 기록을 위하여 다시 ‘dump 4, 27’의 형태로 새로운 문자열에 저장해준 후 반환해준다.

* 1. **shellcmd.c**

해당 파일은 shell 관련 명령어를 수행하는 모듈들이 내장되어 있는 파일이다. 20171662.c 의 main() 모듈에서 shell 관련 명령어를 입력 받았을 때, 이 파일 안에 들어있는 모듈 중 매칭되는 모듈이 수행되며 해당 명령어의 기능을 수행하게 된다. d[ir], hi[story], q[uit](의 일부) 기능이 수행되도록 하는 모듈들이 내장 되어있다.

* + 1. **printdir()**

해당 모듈은 Shell 관련 명령어 중 d[ir] 기능을 수행하는 모듈이다. 현재 디렉토리에 있는 모든 파일을 출력한다. 이는 dirent.h, sys/stat.h 라이브러리에 내장되어 있는 기능을 사용하여 구현하였다. 출력할 때에 실행 파일은 파일 이름 옆에 ‘\*’ 표시를, 디렉터리는 ‘/’ 표시를 하게끔 하였다.

* + 1. **insertNode()**

해당 모듈은 Shell 관련 명령어 중 hi[story] 기능 중 일부를 수행하는 모듈이다. 모든 유효한 명령어가 main()에서 input으로 들어올 때 마다 해당 명령어를 Linked List 형태로 저장하여야 하는데, 해당 모듈은 Linked List 맨 뒤에 새로운 Node를 추가하는 역할을 한다. 만약 잘못된 명령어를 입력하는 경우엔 해당 Linked List에 남기지 않는다.

* + 1. **ShowAll()**

해당 모듈 역시 hi[story] 기능을 수행하는 모듈이다. main() 모듈에서 hi[story]가 input으로 들어왔을 때 현재까지 사용한 명령어들을 순서대로 번호와 함께 보여주어야 한다. 가장 최근 사용한 명령어가 리스트의 하단에 오도록 하는 기능을 수행한다. insertNode()에서 생성된 Linked List를 처음부터 끝까지 출력해준다.

* + 1. **FreeAll()**

해당 모듈은 hi[story]를 위해 만들어진 Linked List에 할당된 메모리를 해제하는 역할을 한다. q[uit] 명령어가 입력되었을 시에 프로그램을 바로 종료하는 것이 아닌 사용된 메모리를 해제하고 프로그램을 종료시키기 위해 만들어졌다.

* 1. **memcmd.c**

해당 파일은 메모리 관련 명령어를 수행하기 위한 모듈들이 내장되어 있는 파일이다. 추후에 구현하게 될 assembler, linker & loader를 통하여 만들어진 object 파일을 올려서 실행하게 된 shell 내의 메모리 공간에 관련된 명령어들로써 이 프로그램에서는 사이즈가 1Mbyte(16\*65536)인 가상의 메모리 공간을 구현하여야 했다. 해당 메모리 공간은 main()에서 할당 및 초기화 해주었다. 구현해야 할 메모리 관련 명령어로는 du[mp] [start, end] / e[dit] address, value / f[ill] start, end, value / reset이 있다.

* + 1. **print\_mem()**

해당 모듈은 du[mp] 명령어가 들어왔을 때 주어진 start address부터 end address까지를 출력하는 모듈이다. main함수에서 boundary check를 한 뒤 du[mp] [start, end]를 위한 기본 모듈에서 start 및 end address를 print\_mem()에 전달한다. 가장 왼쪽 칼럼은 출력하는 메모리 주소를 나타낸다. 주소는 5 자리로 고정하고 16진수로 출력하게끔 하였다. 또한 16진수의 알파벳은 모두 대문자로 출력하게끔 하였다. 가운데 칼럼은 메모리 내용을 16진수 형태로 나타내었고, 역시 알파벳은 대문자로 출력하게끔 설정하였다. 가장 오른쪽 칼럼은 메모리 내용을 byte별로 대응하는 ASCII code 형태로 보여준다. ASCII code로 출력하는 범위는 16진수로 20~7E이며, 이 이외의 값은 ‘.’로 출력한다.

* + 1. **onlydump()**

해당 모듈은 du[mp] 만 input으로 들어왔을 때 main()에서 사용되는 모듈이다. 전역 변수로 선언된 end\_index를 사용하는데, dump 명령어가 처음 시작될 때는 end\_index가 0xFFFFF으로 다시 0번지부터 10줄을 출력하게끔 print\_mem()에 0번지부터 10번째 줄의 마지막 번지를 넘긴 후 출력한다. 그런 다음 end\_index를 159로 설정하고, 그 다음부터는 end\_index를 해당 출력 범위의 맨 마지막 번지수를 저장한다. 그런 다음 다시 dump 가 들어오면 end\_index + 1 번지수부터 10 줄을 print\_mem()을 이용하여 출력하게끔 한다. 만약 10번째 줄의 마지막 번지수가 0xFFFFF보다 크게 되면, 0xFFFFF까지만 출력하게끔 한다.

* + 1. **dumpstart()**

해당 모듈은 du[mp] start 만 input으로 들어왔을 때 main()에서 사용되는 모듈이다. start 부분에 들어온 주소값부터 10줄을 출력하는데, 만약 start의 주소값이 메모리 범위를 벗어나는 경우 dumpstart()가 바로 호출되는 대신 main()에서 에러 문구 출력 후 다시 input을 받게끔 한다. 따라서 dumpstart()에는 정상적인 start address가 들어오게 되는데, 만약 start 주소값부터 160번째 번지수가 0xFFFFF를 넘어가는 경우, 0xFFFFF까지만 출력하게끔 print\_mem() 모듈에 적절한 번지수를 넘긴다.

* + 1. **dumpstartend()**

해당 모듈은 du[mp] start, end가 input으로 들어왔을 때 main()에서 호출되는 모듈이다. start부터 end까지의 주소값을 출력하는데, main()함수에서 start와 end의 주소값 boundary checking을 하여 만약 유효하지 않은 주소값이 들어왔을 시 해당 모듈은 호출이 되지 않는다. 따라서 해당 모듈 내에서는 들어온 start와 end값이 그대로 print\_mem()모듈에 전달되게 된다.

* + 1. **editvalue()**

해당 모듈은 e[dit] address, value가 input으로 들어왔을 때 main()에서 호출되는 모듈이다. main()에서 해당 모듈이 호출되기 전 input으로 들어온 address가 범위를 벗어나지 않는지, value가 범위를 벗어나지 않는지를 먼저 checking하기 때문에 해당 모듈은 모든 값들이 유효할 때만 호출된다. 따라서 input으로 들어온 번지수의 값을 input으로 들어온 value로 대체한다.

* + 1. **fillvalue()**

해당 모듈은 f[ill] start, end, value가 input으로 들어왔을 때 main()에서 호출되는 모듈이다. 이 input도 main()에서 boundary checking이 주소값과 value값 모두에 수행되므로 만약 적절하지 않은 값이 들어올 시 해당 모듈은 호출되지 않는다. 따라서 해당 모듈에는 유효한 값만 들어온다. 해당 모듈은 start address부터 end address까지의 값들을 모두 value값으로 대체하는 역할을 수행한다.

* + 1. **reset\_mem()**

해당 모듈은 reset이 input으로 들어왔을 때 main()에서 호출되는 모듈이다. 해당 모듈은 가상 메모리에 저장되어있는 모든 값들을 다 0으로 초기화해주는 역할을 한다.

* 1. **opcmd.c**

해당 파일은 OPCODE TABLE 관련 명령어를 수행하는 기능을 갖춘 모듈을 모아놓은 파일이다. sic source 코드를 어셈블러를 통해서 object코드로 변환시키기 위해서는 주어진 명령어(mnemonic)를 해당하는 opcode로 변환하는 작업이 필요하다. 우선 미리 주어진 opcode.txt에 instruction name과 그에 해당하는 opcode 및 argument 개수가 담겨있다. main()에서 관련 명령어가 들어오기 전 프로그램이 시작할 때에 미리 해당 opcode 리스트를 Linked List형태로 구성된 hashTable로 구현하게끔 하였다. 해당 파일에서는 hashTable 구성 및 input으로 주어진 instruction의 opcode를 찾아주는 모듈, 그리고 opcodelist가 input으로 들어왔을 시 hashTable을 모두 보여주는 모듈이 존재한다.

* + 1. **init()**

해당 모듈은 main() 초기에 hashTable[i]에 메모리를 할당해준 후 next를 NULL로 초기화해주는 역할을 한다. hashTable[i]는 총 20개가 존재한다. 이는 TABLE\_SIZE로 헤더 파일에 정의해두었다.

* + 1. **destructor()**

해당 모듈은 q[uit]가 input으로 들어왔을 때 존재하던 hashTable의 메모리를 할당 해제해주는 역할을 한다.

* + 1. **search()**

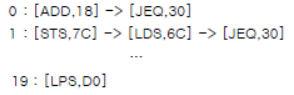
해당 모듈은 opcode ‘instruction name’이 input으로 들어왔을 때 해당 instruction의 opcode를 찾아주는 역할을 한다. 미리 짜여진 hashTable에서 instruction name의 첫 알파벳의 (ascii code) mod 20을 index로 삼아 찾은 다음 해당 instruction의 opcode를 return한다. 만약 주어진 instruction이 list에 없다면, -1을 반환하여 없음을 나타낸다. main()에서 해당 리턴값을 받은 후 -1이면 적절한 에러 문구를 출력한다.

* + 1. **add()**

해당 모듈은 main() 초기에 hashTable을 구성할 때에 호출된다. instruction name의 첫 알파벳의 (ascii code) mod 20을 key값으로 삼아 hashTable[key]에 새로운 node가 추가되게끔 하였다. 존재하는 node들에 맨 앞에 제일 나중 node가 들어가게끔 하는 node 추가 방식을 사용하였다. node는 main()에서 미리 만들어져 key값과 같이 parameter로 들어오게 된다.

* + 1. **show()**

해당 모듈은 opcodelist가 input으로 들어왔을 때 호출되는 모듈이다. hashTable에 존재하는 모든 node들을 각 key값에 따라 출력해준다. 출력 형식은 아래와 같다.



1. **전역 변수 정의**
   1. **int end\_index**

해당 전역 변수는 du[mp] 명령어를 위해 사용된다. du[mp] 첫 사용시, 시작 주솟값은 0이지만, 계속된 해당 명령어 호출 시 끝난 주솟값의 다음 주솟값부터 10줄을 출력해야한다. 이러한 기능을 위해선 마지막 주솟값을 전역 변수에 저장해놓고 du[mp]가 실행될 때마다 그 값을 업데이트해줄 필요가 있다. 이러한 역할을 하는 것이 end\_index이다. main() 초기에 0xFFFFF를 저장해둔 다음, du[mp]가 실행될 때 마다 새로운 end address 값을 넣어준다.

* 1. **char\* input**

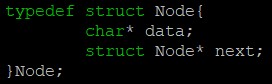
해당 전역 변수는 main()에서 처음 input을 받을 때 사용된다. sicsim> ‘command’를 사용자가 입력하게 되는데, 해당 command 부분을 입력 받아 저장하게 된다.

* 1. **char\* modified\_input[100]**

해당 전역 변수는 char\* input으로 받아진 command 부분을 ‘,’, ‘ ‘, ‘\n’을 기준으로 tokenize하여 각 ‘단어’의 시작 주소를 modified\_input[i]에 각각 저장하는 용도로 쓰인다. command 와 해당 command에 대한 argument를 나누어 생각하기 위함이다. 또한 첫번째 ‘단어’로 해당 명령어가 유효한 명령어인지 확인하기 위해서도 쓰인다. 나머지 argument는 address나 value 등의 값을 저장하게끔 한다.

* 1. **Node\* head**

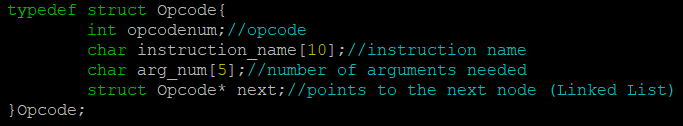
해당 전역 변수는 hi[story]에 사용될 Linked List의 노드를 Node라는 구조체를 사용하여 나타낼 때 해당 리스트의 첫 부분을 가리키는 포인터이다.



해당 구조체는 위와 같다. head 변수에는 값을 넣지 않고, head의 next부터 들어온 유효한 명령어들을 순서대로 저장하게 된다. 명령어는 char\* data에 담기게 된다.

* 1. **Opcode\*\* hashTable**

해당 전역 변수는 OPCODE TABLE에 관련된 명령어를 수행하기 위하여 만들어졌다. OPCODE TABLE의 구성요소들을 Linked List형태로 구현된 hash Table로 저장하여야 하는데, 각 opcode를 저장할 노드의 역할을 Opcode 구조체가 한다. Opcode 구조체는 다음과 같다.



int opcodenum에 mnemonic에 해당하는 opcode를 넣고, instruction\_name에 mnemonic을 저장하며, arg\_num에는 argument 개수를 저장해준다. hash Table의 key값은 각 mnemonic의 (첫 글자의 ASCII code) mod 20으로 설정해주고 저장한다. 이 Linked List역시 첫 hashTable[i]에는 값을 지정해주지 않으며, 그 다음 노드부터 값을 넣어준다.

* 1. **char helpprint[]**

해당 전역 변수는 위 5개 전역 변수와는 다르게 20171662.c파일에 선언되어있다. 해당 문자열에는 h[elp]가 input으로 들어왔을 시 출력해야 하는 문자열을 담고 있다.



1. **코드 설명**
   1. **#define 매크로**
      1. **MEM\_SIZE**

#define MEM\_SIZE 1<<20

메모리 관련 명령어는 모두 1Mbyte의 가상 메모리를 바탕으로 수행된다. MEM\_SIZE는 해당 가상 메모리의 공간을 가리킨다.

* + 1. **OPCODE**

#define OPCODE “opcode.txt”

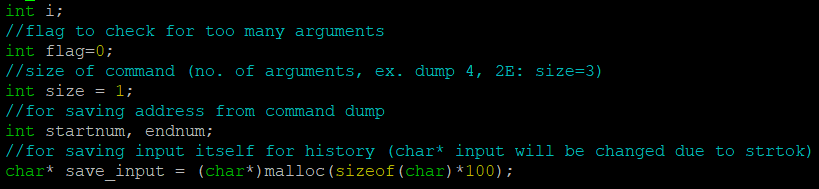
OPCODE TABLE 관련 명령어는 모두 주어진 ‘opcode.txt’파일 안에 들어있는 mnemonic과 그에 해당하는 opcode를 바탕으로 수행된다. OPCODE는 해당 파일명을 가리킨다.

* + 1. **TABLE\_SIZE**

#define TABLE\_SIZE 20

hash Table의 크기는 20으로 고정해야한다. TABLE\_SIZE는 hash Table의 크기를 가리킨다.

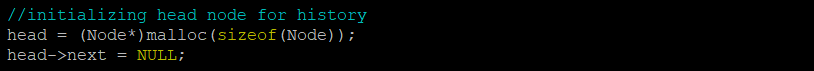
* 1. **20171662.c**
     1. **main()**



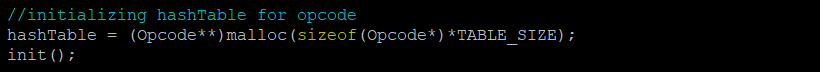
초기 변수 선언은 위와 같다. 우선 int i의 경우 for문을 사용할 때 필요한 변수로 선언하였다. int flag=0의 경우 이후 input을 tokenize하면서 int size를 이용하여 argument의 개수를 저장하는데, 만약 argument의 가능한 최대 크기를 벗어나게 되면 flag=1로 바뀌며 다시 input을 받게끔 하게 한다. int startnum, endnum은 메모리 관련 명령어 중 dump를 수행할 때에 start address 및 end address를 저장하는 역할을 한다. 특히 input을 받을 때엔 문자열로 받게 되어 tokenize한 후에도 address를 가리키는 값들이 문자열형태로 저장되어있기 때문에, strtol를 이용하여 16진수로 변환하여 해당 변수에 저장한다. char\* save\_input은 tokenize된 input을 history에 저장할 수 없으므로 tokenize되기 전 input의 사본을 저장해두는 역할을 한다. 메모리는 100 char만큼 할당해준다.



그런 다음 전역 변수 end\_index를 0xFFFFF로 초기화해준다. 나중에 dump의 함수에서 만약 마지막 번지수가 0xFFFFF를 가리킨다면, 다시 0번째 번지부터 10줄을 출력해야하기 때문에 프로그램의 효율성을 위하여 위와 같이 초기화해주었다.



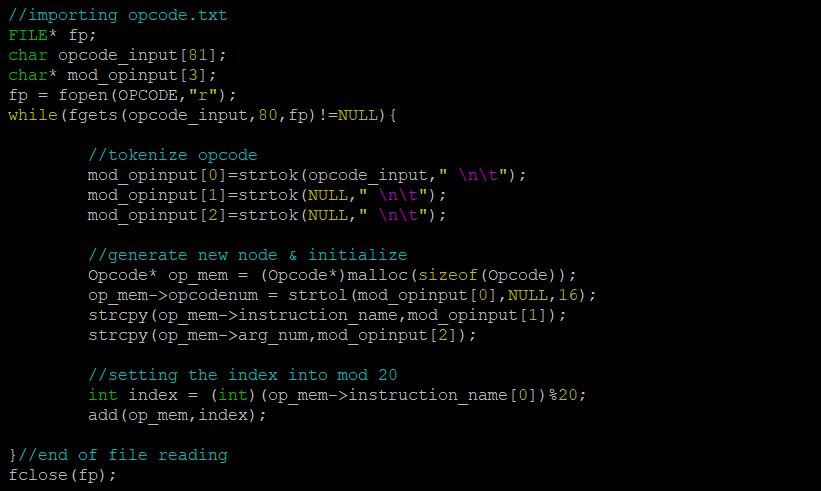
hi[story] 명령어 구현을 위하여 Linked List를 사용한다. 해당 리스트의 head node를 가리키게 하기 위하여 전역 변수 Node\* head를 헤더 파일에 선언해주었고, 그 변수에 메모리를 할당해준다. 그런 다음 next에 NULL을 넣어 주어 리스트를 초기화해준다.



OPCODE TABLE 관련 명령어를 위한 mnemonic과 opcode가 담긴 hash Table을 만들어야 한다. 우선 전역 변수로 선언된 hashTable에 메모리를 할당해 준 후, opcmd.c 파일 안의 init()함수를 호출하여 내부 메모리도 각각 할당해준다. (자세한 사항은 5.5.1 참고)



메모리 관련 명령어를 위해선 가상의 1Mbyte 메모리를 사용해야 한다. 전역 변수로 선언된 my\_mem의 값을 모두 0으로 초기화해준다.

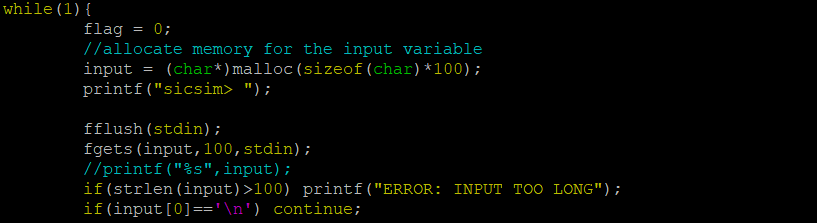


현재 디렉토리에 저장된 opcode.txt를 hashTable로 import한다. 파일 포인터를 사용하여 해당 파일을 읽기 전용으로 연 다음, char opcode\_input[81]에 opcode.txt의 한 줄을 전체로 입력 받는다. 입력은 fgets가 NULL을 반환하는, 즉 파일의 끝까지 계속 입력 받는다.

한 줄을 입력 받은 후엔 strtok을 ‘ ‘, ‘\n’, ‘\t’를 기준으로 tokenize하여 각각 char\* mod\_opinput[]에 각 token의 시작 주소값을 넣어준다. 총 세가지의 token이 한 줄에 존재하므로 mod\_opinput[0]부터 [2]에 차례대로 넣어준다.

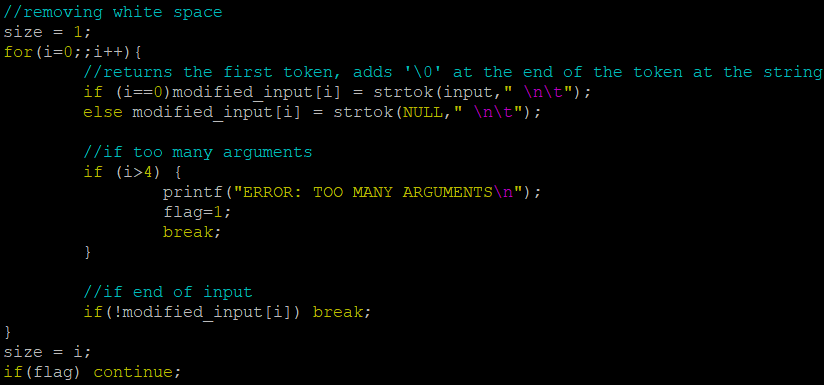
그런 다음, struct Opcode\* op\_mem이라는 새로운 노드를 생성하고 메모리를 할당하여 준 후, 각 token을 알맞은 struct의 member에 저장해준다. 첫 token은 opcode를 가리키는 관계로 16진수를 가리키고 있기 때문에, 문자열로 받은 상태에서 strtol을 사용하여 16진수로 변환하여 op\_mem->opcodenum에 저장해준다. 나머지 mnemonic과 마지막 argument 개수는 문자열 그대로 op\_mem->instruction\_name과 op\_mem->arg\_num에 각각 strcpy를 사용하여 저장해준다.

그런 다음 새로운 int형 변수 index를 선언해주어 해당 변수에 현재 노드가 hashTable의 어느 key에 저장될지를 저장해준다. hashTable의 key는 mnemonic의 첫 글자의 ASCII code mod 20으로 설정해준다. 그 후 add()모듈에 해당 노드와 key를 parameter로 전달해주어 전역변수로 선언된 hashTable[key]에 저장된다. (자세한 사항은 5.5.4 참고)

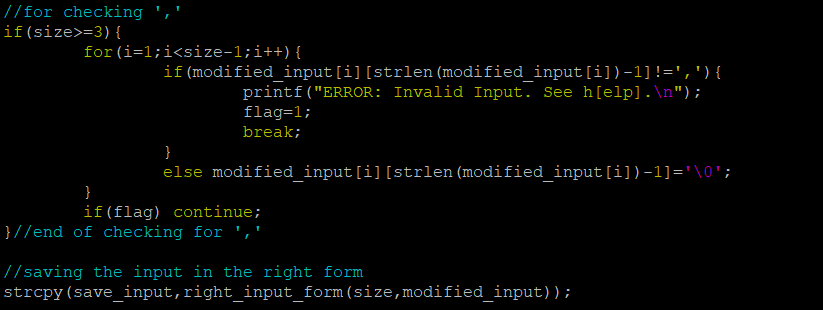


while(1)로 quit명령어가 들어올 때까지 input을 받게끔 한다. 그런 다음 flag=0으로 초기화해주어 이전의 flag결과가 영향을 미치지 않도록 한다. 그런 다음 전역 변수 input에 100만큼 메모리 할당을 해준다. 그 후 “sicsim> “을 출력해주어 unix shell과 유사한 입력 프롬프트상태가 되도록 한다.

input을 fgets로 받기 전 fflush(stdin)을 해주어 garbage 값이 들어오지 않도록 버퍼를 비워준다. 그런 다음 한 줄을 input에 한 번에 입력 받는다. 만약 input이 너무 길면 에러 문구를 출력하게 해준다. 또한, 만약 사용자가 아무것도 입력을 하지 않고 엔터만 쳤을 경우, 다시 “sicsim> “을 출력해주어 input을 받게끔 한다.

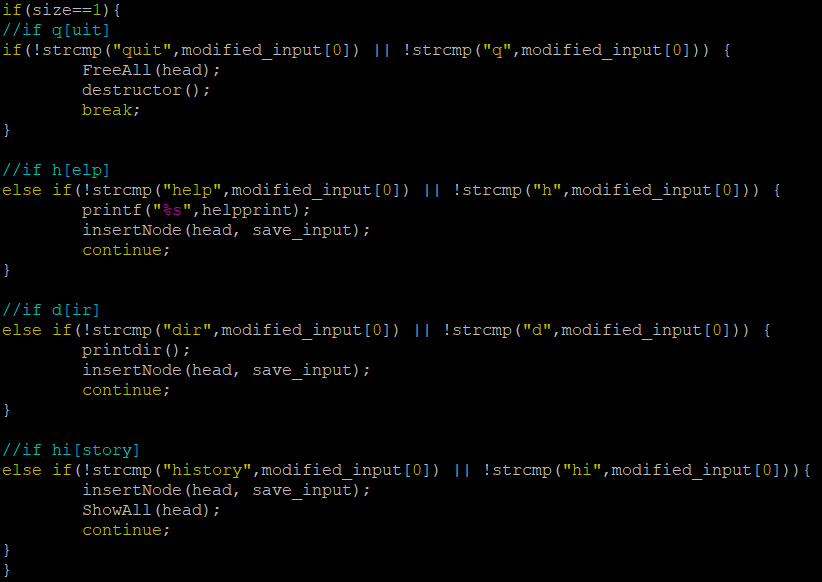


input을 tokenize하여 원하는 값만 저장할 수 있게끔 해주어야 한다. 우선 size를 1로 초기화해준다. 모든 command는 최소 크기가 1이기 때문이다. (만약 입력이 들어온다면.) strtok로 들어온 input을 ‘,’, ‘\n’, ‘\t’를 기준으로 나눈다. strtok는 해당 캐릭터가 나오면 ‘\0’으로 대체시킨다. 각 token의 시작 주소를 modified\_input[i]에 각각 넣어준다. 만약 token의 개수가 4보다 커지게 되면 해당하는 명령어가 없으므로 에러 문구와 함께 flag를 1로 설정해준 후 tokenize를 즉시 중단한다. 그 외에는 계속 token을 저장하다가 마지막 token이 NULL을 가리키면 tokenize를 중지한다. 그런 token의 개수를 size에 저장하고, flag가 1이라면 유효하지 않은 값이 들어왔다는 뜻이므로 다시 input을 받게끔 하기 위해 continue 해준다.



들어온 input에서 argument간의 구분은 대표적으로 ‘,’를 통해서 한다. 따라서, 위에선 공백만 없애주며 tokenize한 것이고, argument에 해당하는 token의 끝에 쉼표가 오는지를 확인하는 작업이 필요하다. argument는 두번째 token부터이므로, 해당 token부터 확인해준다. 또한 맨 마지막 argument역시 마지막에 쉼표가 붙으면 안되므로, 마지막 직전까지의 argument만 확인해준다. 만약 끝에 쉼표가 있어야 하는 argument에 쉼표가 붙어있지 않다면, error 문구를 출력해준 뒤 다시 input을 받게끔 한다. 만약 쉼표가 정상적으로 붙어있다면 해당 쉼표를 마치 strtok함수처럼 ‘\0’로 대체해준다.

그런 다음 right\_input\_form()모듈을 통하여 정상적으로 tokenize된 input들을 다시 적절한 input form으로 변환해준 후 history linked list를 위하여 save\_input 문자열에 저장해준다. (자세한 사항은 5.2.3 참고)



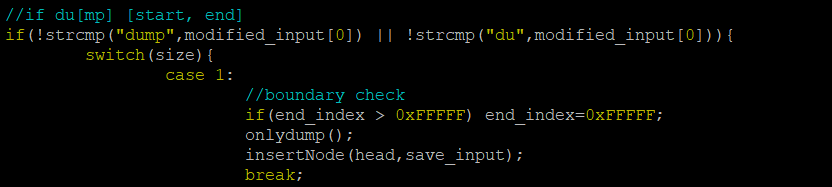
만약 size가 1이라면 command 가 한 단어로만 이루어진 명령어 라는 뜻이므로 기본 쉘 명령어 중 하나일 수 있다. q[uit], h[elp], d[ir], hi[story] 모두 command size가 1이므로 첫 token이 이 중 하나이더라도 만약 size가 1이 아니라면 유효하지 않은 명령어일 것이다.

만약 input이 ‘q’, ‘quit’중 하나라면 modified\_input[0]역시 이 둘 중 하나와 같을 것이다. strcmp를 사용하여 비교해준 다음, 맞다면 quit의 기능을 수행하여준다. quit의 기본 기능은 쉘을 종료하는 것이지만, 프로그램을 종료시키기 전 history와 hash Table에 사용된 Linked List에 할당된 메모리를 해제하는 것이 바람직하므로 FreeAll()모듈과 destructor()모듈을 사용하여 메모리를 할당 해제해준 후 break로 while문에서 나오게끔 한다. (자세한 사항은 FreeAll(): 5.3.4, destructor(): 5.5.2 참고)

만약 input이 ‘h’, ‘help’중 하나라면 modified\_input[0]역시 이 둘 중 하나와 같을 것이다. strcmp를 사용하여 비교해준 다음, 맞다면 help의 기능을 수행하여준다. 앞서 전역 변수로 선언 및 정의해준 helpprint 문자열을 그대로 출력해준다. 해당 명령어는 또한 유효한 명령어이므로 insertNode()모듈을 사용하여 history Linked List에 save\_input을 저장해준다. (자세한 사항은 5.3.2 참고)

만약 input이 ‘d’, ‘dir’중 하나라면 modified\_input[0]역시 이 둘 중 하나와 같을 것이다. strcmp를 사용하여 비교해준 다음, 맞다면 dir의 기능을 수행하여준다. dir의 기능으로는 현재 디렉토리에 있는 모든 파일을 출력하고, 디렉토리라면 ‘/’를, 실행파일이라면 ‘\*’를 뒤에 붙여주어야 한다. printdir()모듈을 사용하여 해당 기능을 수행하게 해준다. (자세한 사항은 5.3.1 참고) 해당 명령어는 또한 유효한 명령어이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)

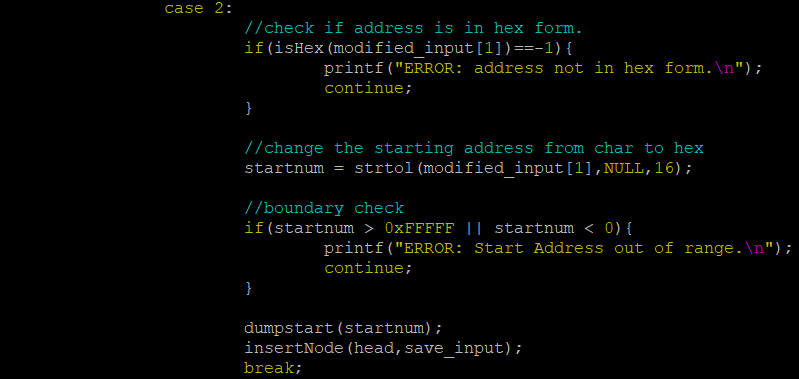
만약 input이 ‘hi’, ‘history’중 하나라면 modified\_input[0]역시 이 둘 중 하나와 같을 것이다. strcmp를 사용하여 비교해준 다음, 맞다면 history의 기능을 수행하여준다. history의 기능으로는 여태 input으로 들어온 유효한 명령어를 모두 출력해주는 것이다. 해당 명령어 또한 유효한 명령어이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고) 그런 다음 ShowAll()모듈을 사용하여 Linked List에 저장된 모든 노드들을 순서대로 번호와 함께 출력해준다. (자세한 사항은 5.3.3 참고)



dump명령어는 command size가 1부터 3까지 있기 때문에 위처럼 size를 먼저 확인하지 않고 첫 token이 ‘dump’ 또는 ‘du’인지부터 확인해주었다. 맞다면, dump 관련 명령어일 것이므로 switch문을 사용하여 해당 command의 size를 확인해주었다.

만약 size가 1이라면, start나 end address가 주어지지 않고 전역 변수로 선언된 end\_index를 사용하여 end\_index의 다음 번지수부터 10줄을 출력해야 하는 ‘du[mp]’라는 것이다. 우선 end\_index의 값을 체크해준다. 만약 그 전의 end\_index가 0xFFFFF보다 더 크게 저장되었다면, 다시 0번지부터 출력해야 하므로 end\_index를 0xFFFFF로 초기화 해주게끔 하였다.

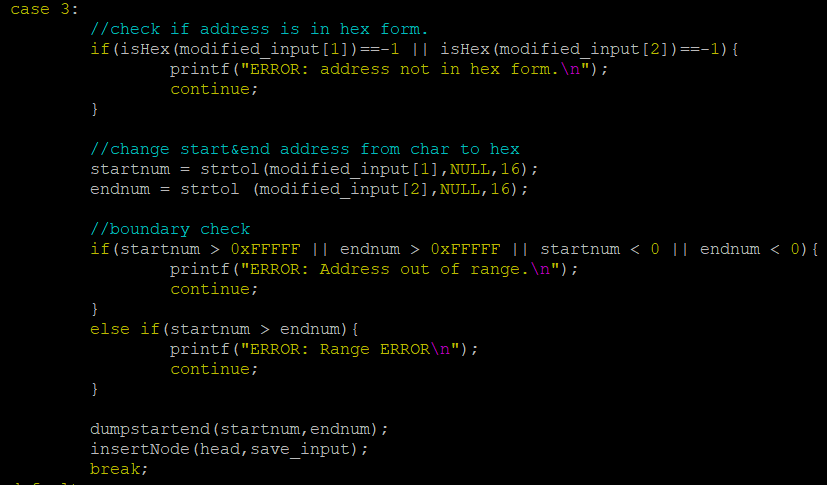
그런 다음 onlydump()모듈을 사용하여 해당 기능을 수행하도록 하였다. (자세한 사항은 5.4.2 참고) 해당 명령어는 유효한 명령어이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)



만약 size가 2라면 start address가 주어진 ‘du[mp] start’라는 것이다. 따라서 modified\_input[1]에는 문자열로 표현된 start address가 저장되어 있을 것이다. 우선 해당 문자열에 들어있는 값이 16진수의 형태로 되어있는지를 isHex()를 통하여 확인한다. (자세한 사항은 5.2.2 참고) 그렇지 않다면 에러 문구를 출력한 후 input을 다시 받게끔 한다. 16진수의 형태라면 해당 문자열을 strtol을 사용하여 16진수로 바꾸어 준 후 startnum 변수에 저장해준다.

그런 다음 boundary check를 수행하여 시작 주소값이 0xFFFFF보다 큰 지, 0보다 작은 지를 확인해준다. 만약 그렇다면, 유효하지 않은 주소값이므로 에러 문구를 출력한 후 다시 input을 받게끔 continue한다.

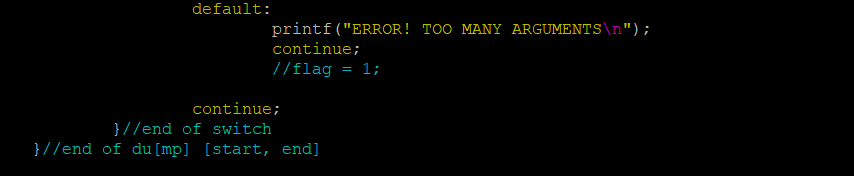
유효한 주소값이라면 dumpstart()모듈을 사용하여 start address부터 10줄, 또는 0xFFFFF를 넘어간다면 0xFFFFF까지 출력하게끔 한다. (자세한 사항은 5.4.3 참고) 이 함수가 수행된다면 유효한 명령어 라는 뜻이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)



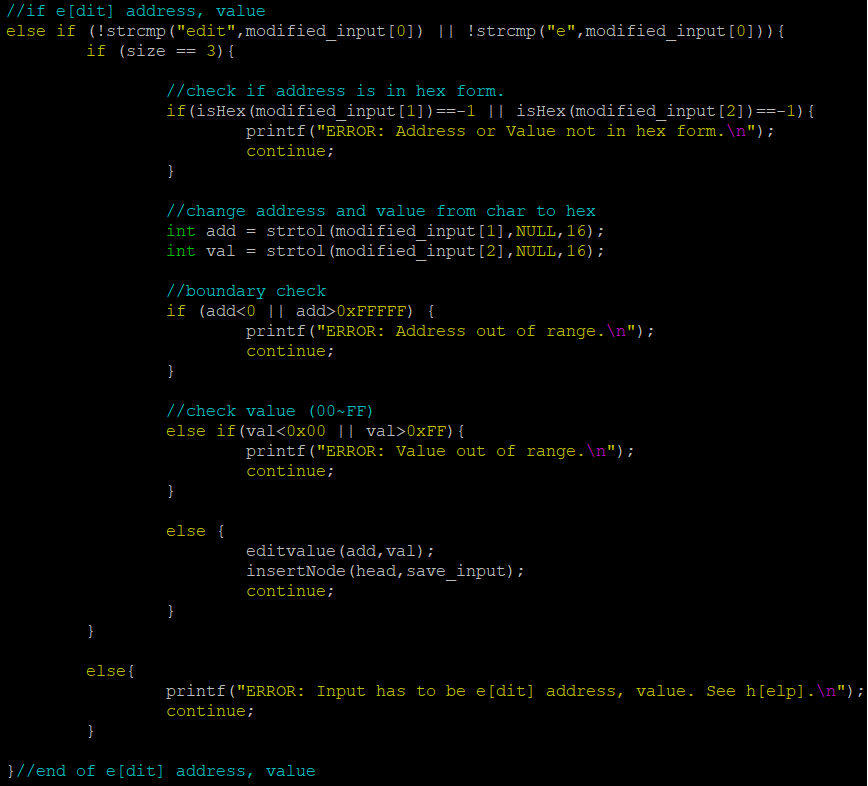
만약 size가 3이라면 start address와 end address가 모두 주어진 ‘du[mp] start, end’라는 것이다. 따라서 modified\_input[1]과 [2]에는 문자열로 저장된 16진수로 표현 되어있는 address가 들어있을 것이다. isHex()모듈을 통하여 해당 address가 적절한 16진수의 형태인지 확인해준다. (자세한 사항은 5.2.2 참고) 그렇지 않다면 에러 문구 출력을 한 후 다시 input을 받게끔 한다. 16진수의 형태가 맞다면, 위와 같이 문자열로 저장된 start와 end address를 16진수로 바꾸어 준 후 각각 startnum과 endnum에 저장해준다.

start와 end address가 0~0xFFFFF안에 있는 범위인지 살핀다. 만약 주소를 벗어나면 에러 문구 출력과 함께 다시 input을 받도록 한다. 또한 start 주소값이 end 주소값보다 큰 경우도 에러 문구를 출력한 후 다시 input을 받도록 한다.

유효한 주소값이 들어왔을 경우 dumpstartend()모듈을 사용하여 해당 범위 내의 메모리값들을 출력하게끔 한다. (자세한 사항은 5.4.4 참고) 이 함수가 수행된다면 유효한 명령어 라는 뜻이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)



만약 size가 1,2,3이 아니라면 size가 더 크다는 것이다. argument의 개수는 최대 2개이므로 argument가 너무 많다는 에러 문구를 출력한 후 다시 input을 받게끔 한다.



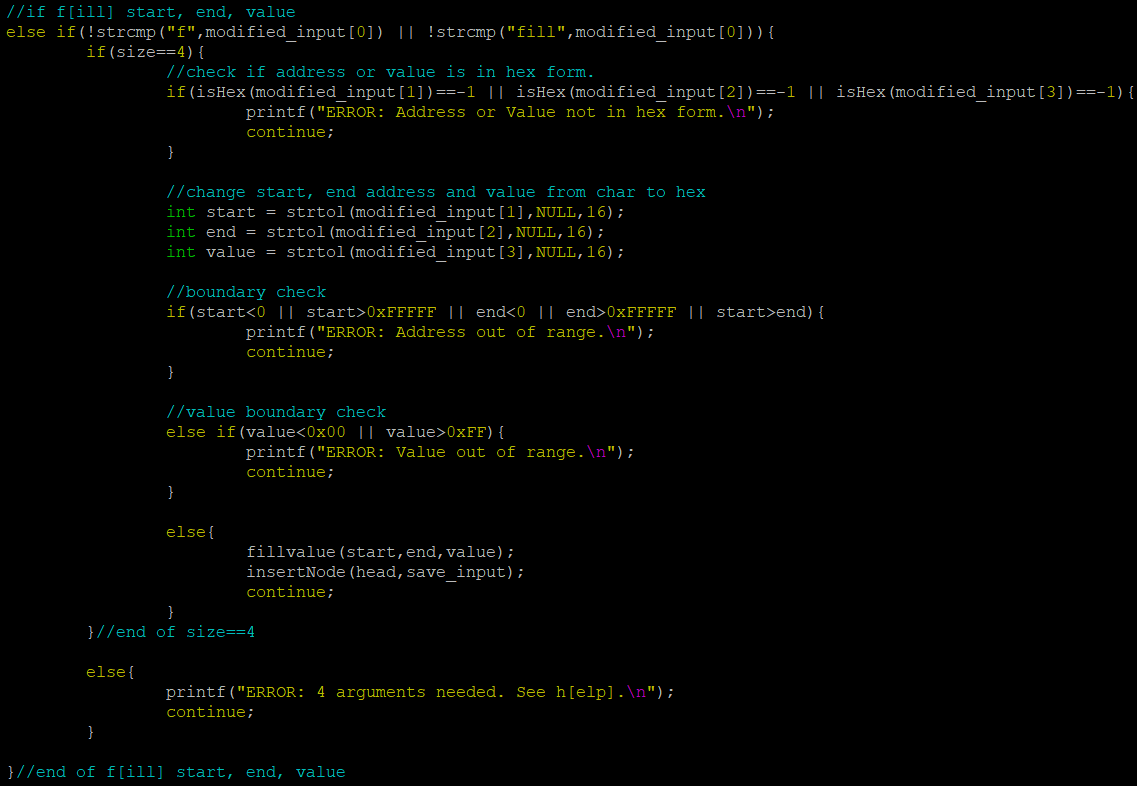
만약 modified\_input[0]이 ‘edit’ 또는 ‘e’라면 ‘e[dit] address, value’라는 것이다. 따라서 command size는 3이어야 하기 때문에 size가 3인지 먼저 확인한다.

명령어 개수가 유효하다면 modified\_input[1]에 문자열로 저장된 값을 바꿀 address가, modified\_input[2]에 문자열로 저장된 value가 저장되어 있을 것이다. 이러한 값들은 모두 16진수여야하기 때문에, isHex()모듈을 통하여 값들이 모두 16진수의 형태인지를 파악한다. (자세한 사항은 5.2.2 참고) 16진수의 형태가 아니라면 에러 무구 출력을 한 후 다시 input을 받게끔 한다. 16진수의 형태가 맞다면 int형 변수 add에 해당 주소값을 16진수로 저장해준다. 또한 int형 변수 val에 value값 역시 문자열에서 16진수값으로 바꾸어 저장해준다.

그 다음으로 address와 value의 범위가 유효한지 살펴본다. address가 0~0xFFFFF안에 있는지, value가 0~FF안에 있는지 확인한 후 범위 밖에 있으면 에러 문구 출력과 함께 다시 input을 받게끔 continue한다.

유효한 주소값과 value까지 확인했다면, editvalue()모듈을 사용하여 메모리상의 주어진 주소값에 주어진 값을 넣는다. (자세한 사항은 5.4.5 참고) 이 함수가 수행된다면 유효한 명령어 라는 뜻이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)

size가 3이 아니라면 argument의 개수가 적절치 못하다는 것으로 에러 문구 출력과 함께 다시 input을 받게끔 한다.



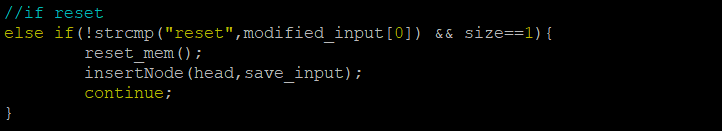
만약 modified\_input[0]이 ‘f’ 또는 ‘fill’이라면 ‘f[ill] start, address, value’라는 것이다. 따라서 command size는 4이어야 하기 때문에 size가 4인지 먼저 확인한다.

명령어 개수가 유효하다면 modified\_input[1]과 [2]에 문자열로 저장된 값을 바꿀 address가, modified\_input[3]에 문자열로 저장된 value가 저장되어 있을 것이다. 이러한 값들은 모두 16진수여야하기 때문에, isHex()모듈을 통하여 값들이 모두 16진수의 형태인지를 파악한다. (자세한 사항은 5.2.2 참고) 16진수의 형태가 아니라면 에러 무구 출력을 한 후 다시 input을 받게끔 한다. 16진수의 형태가 맞다면 int형 변수 start, end에 해당 주소값을 16진수로 저장해준다. 또한 int형 변수 value에 value값 역시 문자열에서 16진수값으로 바꾸어 저장해준다.

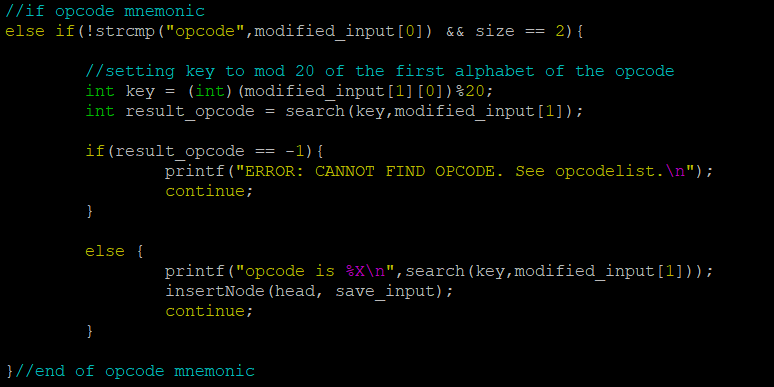
그 다음으로 address와 value의 범위가 유효한지 살펴본다. address가 0~0xFFFFF안에 있는지, value가 0~FF안에 있는지 확인한 후 범위 밖에 있으면 에러 문구 출력과 함께 다시 input을 받게끔 continue한다.

유효한 주소값과 value까지 확인했다면, fillvalue()모듈을 사용하여 메모리상의 주어진 주소값의 범위에 주어진 값을 넣는다. (자세한 사항은 5.4.6 참고) 이 함수가 수행된다면 유효한 명령어 라는 뜻이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)

size가 4가 아니라면 argument의 개수가 적절치 못하다는 것으로 에러 문구 출력과 함께 다시 input을 받게끔 한다.



만약 modified\_input[0]이 ‘reset’이고 size가 1이라면, reset 명령어라는 것이다. 따라서 가상 메모리의 모든 값을 다 0으로 초기화하는 기능을 수행하는 reset\_mem()모듈을 호출한다. (자세한 사항은 5.4.7 참고) 이 함수가 수행된다면 유효한 명령어 라는 뜻이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)

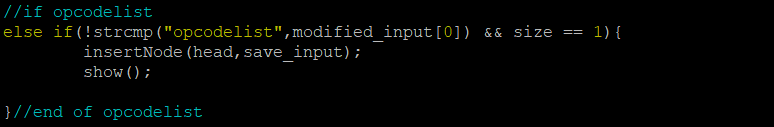


만약 size가 2이고 modified\_input[0]이 ‘opcode’라면, opcode mnemonic 명령어라는 것이다. 따라서 주어진 mnemonic에 대한 key를 먼저 구한 후, main()초기에 만들어두었던 hash Table[key]의 노드들을 탐색하며 mnemonic에 해당하는 노드가 있는지 확인해야 한다.

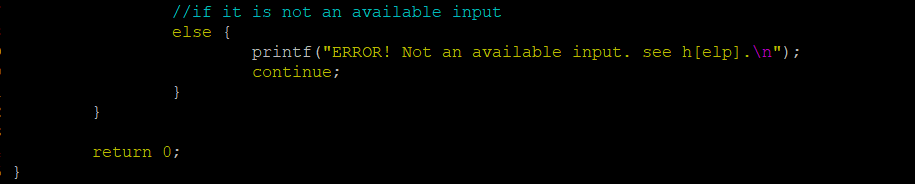
key는 주어진 mnemonic의 첫 알파벳 글자의 ASCII code의 mod 20값으로 삼고, search()모듈을 통하여 hashTable[key]의 노드들 중에 modified\_input[1]에 저장된 주어진 mnemonic이 있는지를 탐색한다. (자세한 사항은 5.5.3 참고) search()의 반환값은 int형 변수 result\_opcode에 저장한다.

만약 result\_opcode가 -1이라면 주어진 mnemonic이 리스트에 없다는 뜻이므로, 에러 문구를 출력한 후 다시 input을 받게끔 한다. 그렇지 않다면 opcode를 찾았다는 뜻이므로 해당 opcode를 출력한다.

opcode가 출력되었다는 것은 주어진 input이 유효하다는 뜻이므로 insertNode()모듈을 사용하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고)

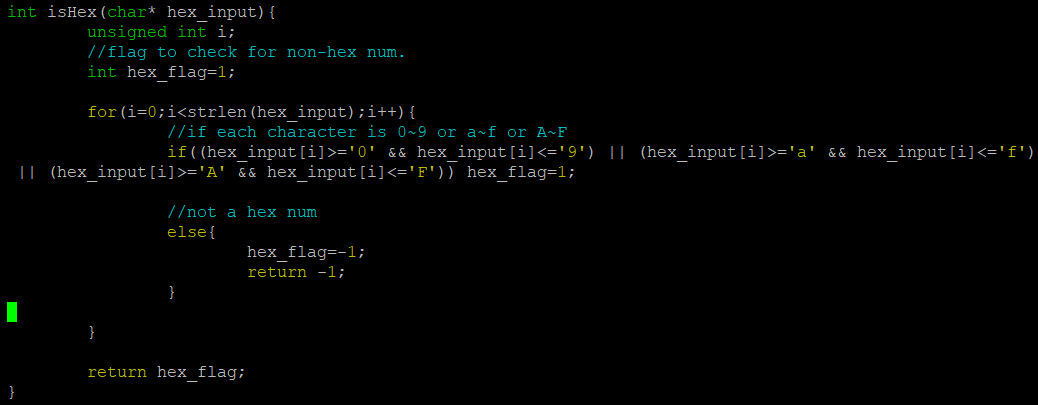


만약 size가 1이고 modified\_input[0]이 ‘opcodelist’라면, opcodelist 명령어라는 것이므로 존재하는 hashTable의 내용을 모두 출력하면 된다. 해당 명령어는 유효한 명령어이므로 insertNode()를 통하여 history Linked List에 저장해준다. (자세한 사항은 5.3.2 참고) 그런 다음 show()모듈을 통하여 hashTable()의 내용을 모두 출력해준다. (자세한 사항은 5.5.5 참고)



위에 언급된 명령어들에 모두 해당이 안된다면, 수행할 수 있는 명령어가 아니므로 에러 문구를 출력한 후 다시 input을 받게끔 한다. quit으로 while문에서 break시 프로그램을 종료한다.

* + 1. **isHex()**

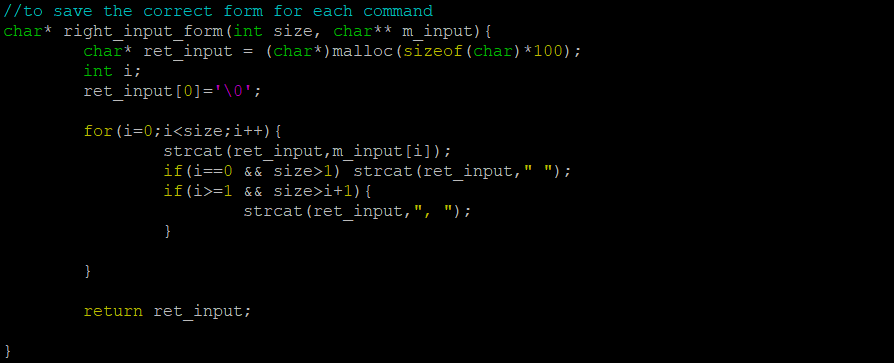


해당 모듈은 input으로 들어온 주솟값이나 value값이 적절한 16진수의 형태로 표현이 되어있는지를 확인해주는 역할을 한다. 예를 들어, 5N이 input으로 들어왔을 때, 이는 16진수가 아니므로 strtol을 사용하여 16진수로 변환 해주기 전 에러 문구를 출력하고 다시 input을 받게끔 해야 한다.

address 및 value를 저장하고 있는 (modified\_input[1]/[2]/[3]) 문자열을 parameter로 char\* hex\_input으로 받는다. 또한 현재 문자가 16진수의 한 자리수가 맞는지를 나타내기 위하여 int hex\_flag=1로 초기화한다.

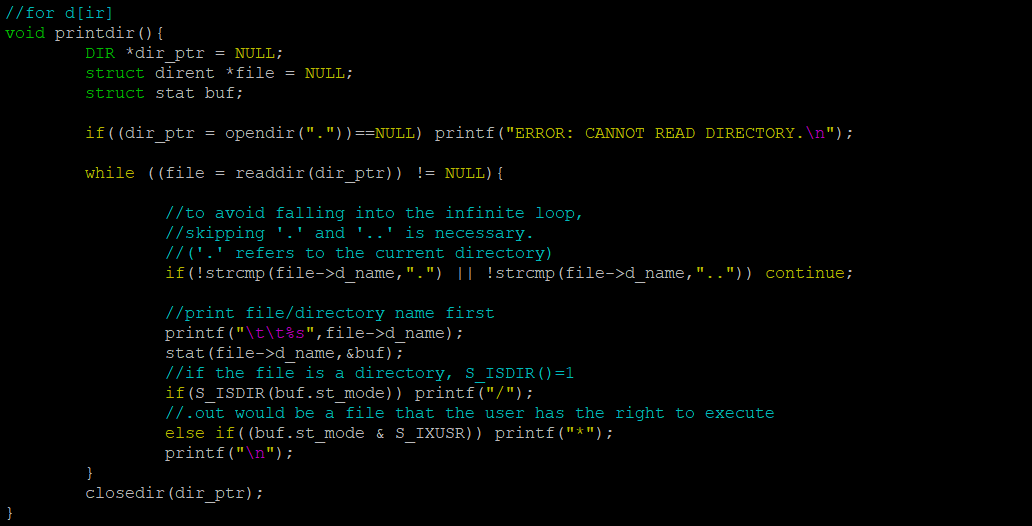
받은 문자열속 문자 하나하나를 순서대로 확인하며 0~9, a~f, A~F 안의 문자인지 확인한다. 만약 해당 범위안에 있는 문자가 아니라면 적절한 16진수의 형태가 아니라는 뜻이므로 flag를 -1로 변환한 후 -1을 바로 반환한다. 만약 모든 문자가 다 적절한 16진수의 형태로 표현되어있다면 for loop을 정상적으로 끝낼 것이므로 hex\_flag 역시 1로 유지될 것이다. 해당 flag값을 반환하여 정상적인 16진수임을 확인한다.

* + 1. **right\_input\_form()**



해당 모듈은 tokenize된 input을 다시 맞는 input form으로 변환시켜주는 역할을 한다. 우선 ret\_input이라는 문자열에 메모리를 적절히 할당해준 후 첫 인덱스의 값을 NULL로 초기화해준다. 그런 다음 modified\_input의 주소값을 m\_input으로 읽어온 후 modified\_input[i]의 문자열을 ret\_input에 strcat을 이용하여 이어 붙인다. 첫 명령어 뒤에는 size가 1일 경우 아무것도 붙지 않지만, size가 2일때부턴 뒤에 ‘ ‘가 붙어야 하기 때문에 strcat을 이용하여 공백을 생성해준다. 그런 다음 argument부터 맨 마지막 argument가 아니라면 ‘, ‘를 붙여주게끔 한다. 완성된 문자열을 반환하여 main에서 save\_input에 저장할 수 있게끔 해준다.

* 1. **shellcmd.c**
     1. **printdir()**



해당 모듈의 목적은 현재 디렉토리에 있는 모든 파일을 출력하는 것이다. 이 때 디렉토리는 뒤에 ‘/’표시를, 실행 파일은 뒤에 ‘\*’ 표시를 해준다.

DIR\* dir\_ptr은 디렉토리를 검색하기 위한 DIR 구조체 포인터 변수이다. 해당 변수는 후에 나오는 opendir(), readdir(), closedir() 모듈에 사용된다. 해당 변수에 현재 디렉토리에 대한 정보가 들어가게된다.

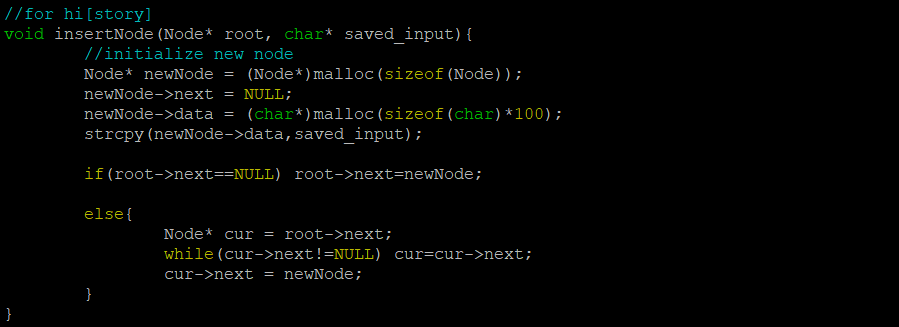
struct dirent \*file은 현재 디렉토리에 들어있는 파일의 정보를 저장하기 위해 만들어진 dirent 구조체 포인터 변수이다. 해당 구조체의 member 중 d\_name을 활용하여 파일 명을 출력하게끔 사용된다.

struct stat buf는 파일의 상태를 저장할 수 있는 stat 구조체 변수이다. 현재 가리키고 있는 파일이 디렉토리이거나 실행 파일인지를 파악하기 위하여 사용된다.

우선 opendir(“.”)을 사용하여 현재 디렉토리에 대한 정보를 dir\_ptr에 저장한다. 만약 NULL이 반환된다면 해당 디렉토리를 읽을 수 없다는 뜻이므로 에러 문구를 출력한다.

readdir(dir\_ptr)의 정보를 file에 저장해가며 디렉토리 내의 파일을 하나하나 순서대로 파악한다. 만약 파일을 다 읽어왔을 땐 해당 모듈이 NULL을 반환할 것이므로 그 땐 while 반복문에서 나오게 된다. 각 파일을 돌 때, 파일명이 ‘.’ 또는 ‘..’이라면 적절한 파일이 아닌 현재/다른 디렉토리를 가리키는 것이므로 넘어간다. 그렇지 않다면 우선 파일 명을 출력한다. 그런 다음 해당 파일의 상태를 stat()모듈을 통하여 저장한 뒤 S\_ISDIR()을 통하여 해당 파일이 디렉토리인지 파악한다. 디렉토리가 맞다면 ‘/’를 뒤에 출력한다. 만약 실행파일이라면 사용자가 실행 권한이 있는 파일이므로 S\_IXUSR()모듈을 통하여 파악한다. 실행 파일이 맞다면 ‘\*’를 뒤에 출력한다. 그런 다음 그 다음 파일로 넘어가기 전 줄바꿈을 해준다. 모든 파일을 다 본 후엔 closedir()해준다.

* + 1. **insertNode()**

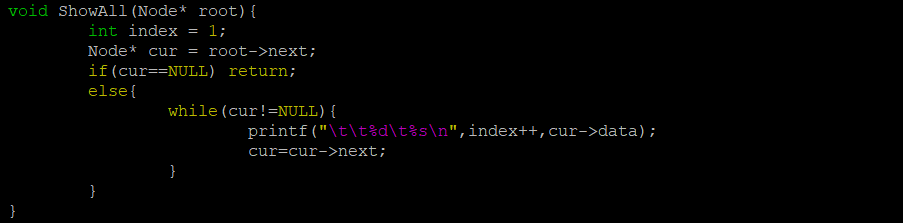


해당 모듈은 history Linked List의 head를 root로, input 한 줄을 담은 문자열을 saved\_input으로 parameter를 전달받는다. 전달받은 문자열을 새로운 노드에 담아 존재하는 Linked List의 맨 마지막 노드로 추가하는 역할을 한다.

우선 Node\* newNode라는 Node 포인터 변수를 선언한 후 메모리를 Node 하나 만큼 할당해준다. 그런 다음 해당 노드의 next는 NULL로, data에는 parameter로 전달받은 input 한 줄을 그대로 strcpy를 사용하여 저장한다.

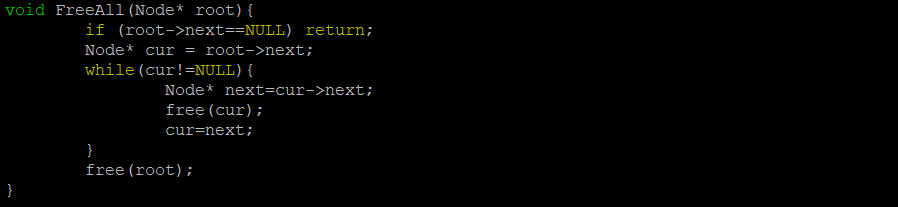
만약 head 뒤에 Node가 하나도 없다면, head노드 바로 다음에 생성한 노드를 가리키게 한다. 만약 이미 Node가 하나 이상 존재한다면, Node\* cur라는 변수를 선언하여 head의 가장 마지막 노드를 가리키게끔 한다. 그런 다음 cur의 next가 생성한 노드를 가리키게끔 한다.

* + 1. **ShowAll()**



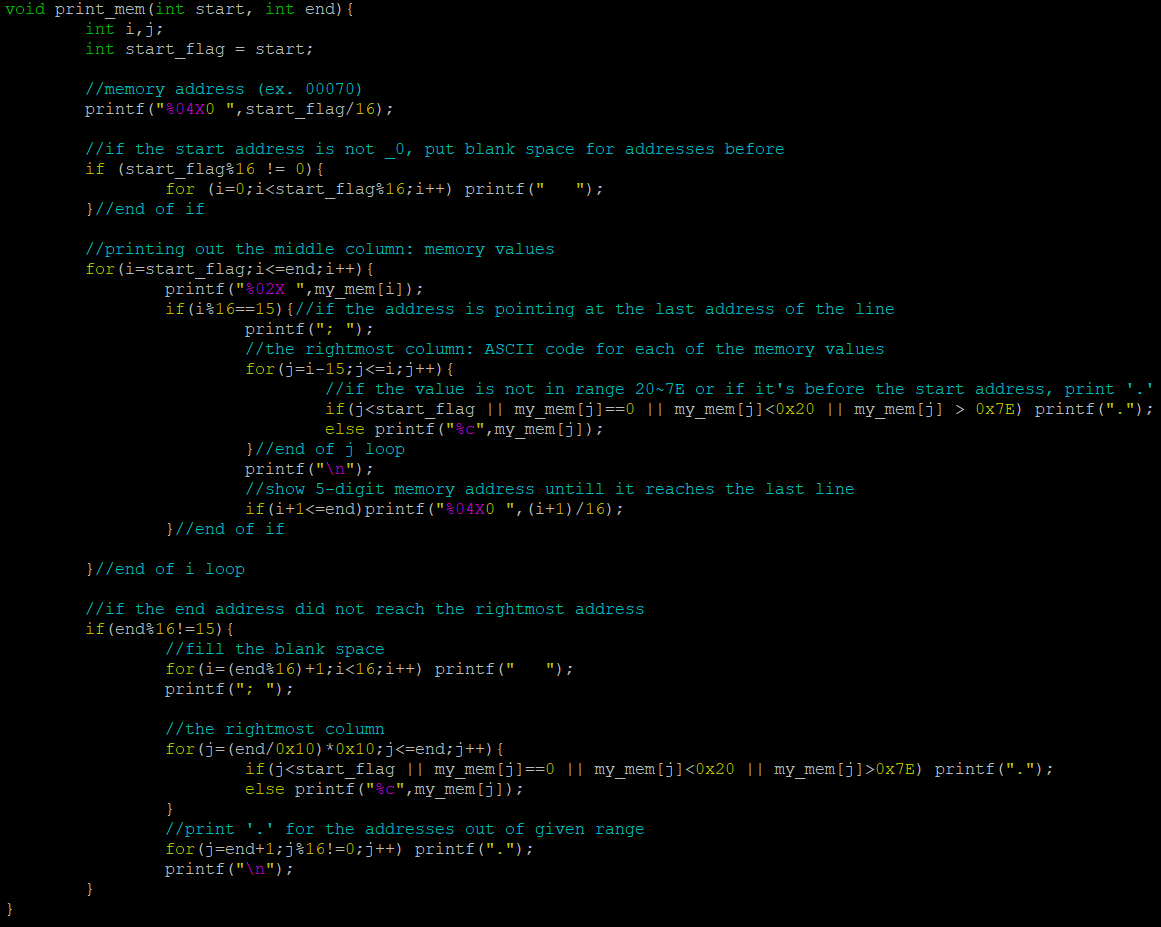
해당 모듈은 history Linked List의 head 부분을 Node\* root로 parameter로 받아 해당 리스트에 있는 모든 노드들을 순서대로 출력한다. index라는 int형 변수로 몇 번째 노드인지를 나타내고, Node\* cur 포인터 변수를 root 바로 다음 노드부터 맨 마지막 노드까지 방문하여 각 노드에 저장되어있는 그 동안 입력된 유효한 명령어들을 들어온 순서대로 출력한다.

* + 1. **FreeAll()**



해당 모듈은 input으로 q[uit]이 들어왔을 시 history Linked List에 사용된 메모리를 할당 해제해주는 역할을 한다. head 노드를 Node\* root 로 parameter를 전달 받아 Node\* cur 포인터 변수와 Node\* next 변수로 리스트 안의 모든 노드에 대한 메모리를 할당 해제해주고, 마지막에 root node까지 할당 해제해준다.

* 1. **memcmd.c**
     1. **print\_mem()**

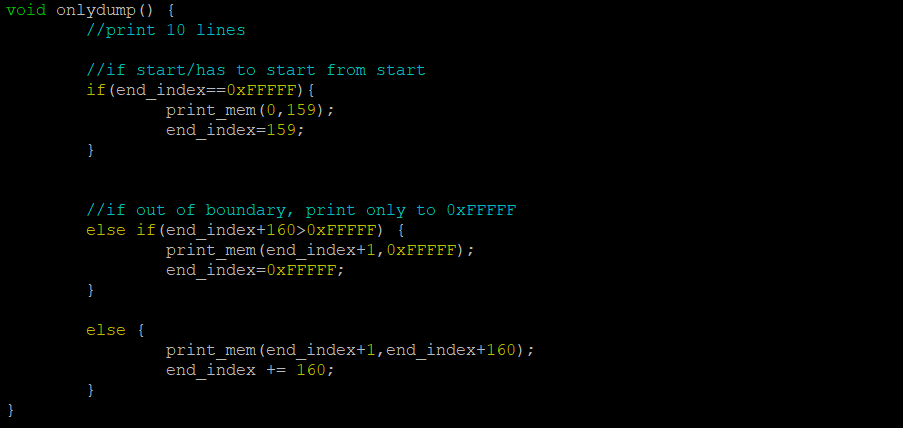


해당 모듈은 dump 명령어 관련 모듈로부터 parameter로 start 및 end address를 전달받은 후 해당 범위 안의 메모리를 모두 출력하는 역할을 한다. 가장 왼쪽 칸엔 5자리 16진수로 출력하는 줄의 메모리 주소를 나타내고, 중간 칸엔 메모리 주소에 저장된 값을 2자리 16진수로 나타내고, 가장 오른쪽 칸엔 각각의 메모리 주소가 가리키는 값의 ASCII Code 값을 출력해준다. 이때 해당 값이 0이거나 20~7E가 아니라면, ‘.’으로 출력해준다.

우선 시작하는 주소의 주소를 사용하여 해당 줄에 해당하는 주소를 출력한다. 그런 다음 해당 줄의 모든 메모리 주소에 들어가 있는 값들을 출력 하여준다. 만약 start address가 해당 줄의 0번째 번지수가 아니라면, start address 전까지는 blank space로 표현한다. 2자리 16진수로 start address 부터 end address까지의 값을 출력하는데, 한 줄의 출력이 끝나면 바로 해당 줄의 값들을 ASCII Code로 변환하여 출력해준다. 이 때는 start address 부터가 아닌 해당 줄 전체 값을 출력해주어야 하는데, start address 이전 값들은 모두 ‘.’로 출력한다. 또한 들어있는 값이 0이거나 20~7E 범위 밖에 있다면 역시 ‘.’로 출력한다. 그 이외의 20~7E 사이에 있고 start address와 end address 범위내의 값들은 해당 값이 가리키는 ASCII Code를 출력한다.

마지막 줄에 다다랐을 때에도 end address까지만 값을 출력하고, end address가 줄 중간에 위치해 있다면 그 이후의 값들도 blank space로 표현한다. 역시 위와 동일한 기준으로 가장 오른쪽 칸에 해당 값에 매칭되는 ASCII Code 또는 ‘.’로 출력한다.

* + 1. **onlydump()**

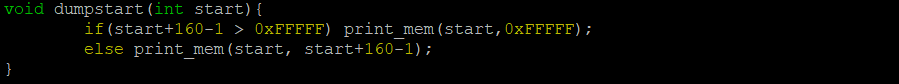


해당 모듈은 du[mp]만 명령어로 들어왔을 때 main()에서 호출되는 모듈이다. du[mp]는 내부에 저장된 end address를 기준으로 end address+1 번지부터 10줄을 출력하는 기능을 가진다. 만약 end address에 가능한 제일 큰 주소값인 0xFFFFF가 저장되어 있을 시, 다시 처음 0번지부터 출력해야함을 의미하므로 print\_mem()모듈에 start address를 0, end address를 159 (10번째줄 가장 마지막 번지수)를 parameter로 넘겨준 후, end address를 저장하는 역할을 하는 전역 변수 end\_index를 159로 바꾸어준다.

만약 end\_index+1인 start address는 가능한 범위 내에 있지만, 10줄 이후의 번지수가 0xFFFFF보다 더 크게 된다면, 0xFFFFF까지만 출력하게끔 하기 위해 print\_mem()에 시작 번지수를 end\_index+1, 끝 번지수를 0xFFFFF로 넘겨준다. 그런 다음 end\_index를 0xFFFFF로 업데이트 시켜준다.

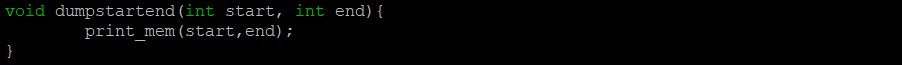
만약 위의 모든 경우가 아니라면, 정상적으로 10줄 모두 출력할 수 있으므로 end\_index+1부터 end\_index+160 번지수 내의 모든 값들을 출력하게끔 print\_mem()에 해당 범위를 넘겨준 후 end\_index를 160 증가시켜준다.

* + 1. **dumpstart()**



해당 모듈은 du[mp] start 가 input으로 들어왔을 때 main()에서 호출되는 함수이다. 주어진 start address로부터 총 10줄을 출력하는 기능을 가진다. 유효한 범위 내에 있는 start만이 parameter로 들어오기 때문에 start address이후 마지막 10번째 줄의 마지막 번지수가 0xFFFFF인지를 확인해주어, 만약 0xFFFFF이하라면 정상적으로 10줄을 출력하게끔 print\_mem()에 start와 start+159를 parameter로 넘겨주고, 그렇지 않다면 print\_mem()에 start와 0xFFFFF를 parameter로 넘겨준다.

* + 1. **dumpstartend()**



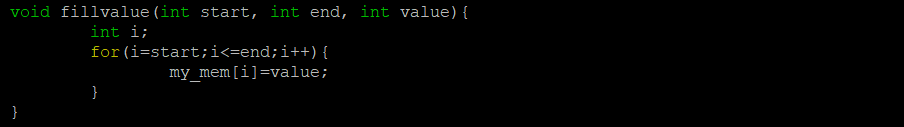
해당 모듈은 du[mp] start, end가 input으로 들어왔을 때 main()에서 호출되는 함수이다. start~end 주솟값에 해당하는 메모리를 출력하는 기능을 가진 함수이다. 주어진 start, end address 모두 main()에서 boundary checking이 완료된 후 유효한 주솟값만이 parameter로 전달된 것이기 때문에, 주어진 start와 end address를 그대로 print\_mem()에 전달해준다.

* + 1. **editvalue()**



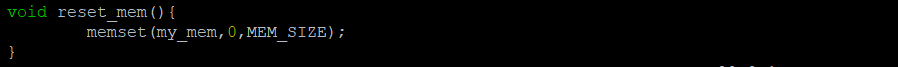
해당 모듈은 e[dit] address, value가 input으로 들어왔을 때 main()에서 호출되며, 주어진 address의 value를 주어진 value로 대체하는 기능을 가진다. 이미 main()에서 유효한 주솟값과 value값을 parameter로 넘겨준 것이므로, my\_mem의 주어진 주솟값에 저장되어 있는 값을 주어진 value로 대체한다.

* + 1. **fillvalue()**



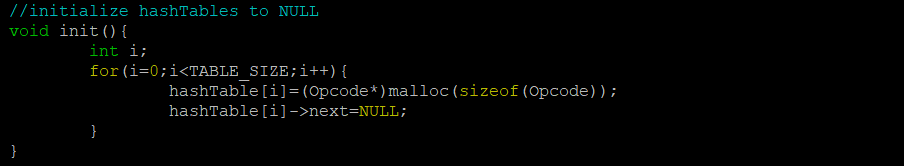
해당 모듈은 f[ill] start, end, value가 input으로 들어왔을 때 main()에서 호출되며, 주어진 주솟값 범위 내의 모든 값을 주어진 value로 바꾸어주는 기능을 가진다. 이미 main()에서 유효한 주솟값과 value값이 parameter로 주어진 것이므로, 해당 start ~ end address내의 value를 주어진 value로 바꾸어준다.

* + 1. **reset\_mem()**



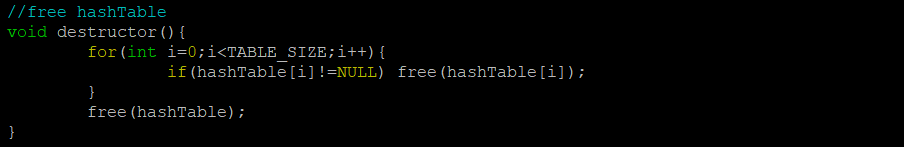
해당 모듈은 ‘reset’이 input으로 들어왔을 때 main()에서 호출되며, 가상 메모리 공간 안의 모든 값을 0으로 초기화시켜주는 기능을 가진다. 단순히 전체 주솟값의 범위를 초기화해주는 기능을 가지므로, memset()모듈을 사용한다.

* 1. **opcmd.c**
     1. **init()**



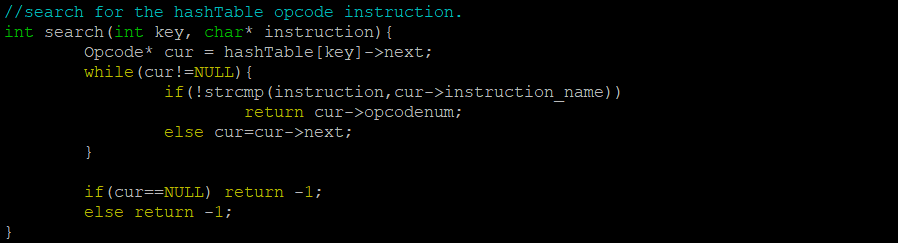
해당 모듈은 main()함수 초기에 Opcode\*\* hashTable 메모리 할당 후 내부적으로 hashTable[i]에 대한 메모리도 모두 할당해주기 위하여 호출된다. hashTable내의 모든 Node들의 next를 NULL로 초기화한다.

* + 1. **destructor()**



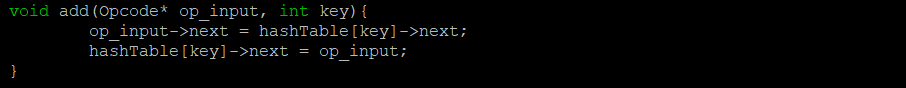
해당 모듈은 q[uit] 명령어가 들어왔을 때 main()에서 호출되며, 생성된 hashTable에 할당된 모든 메모리를 해제하는 역할을 한다. 이러한 기능을 통해 메모리 누수를 막을 수 있다.

* + 1. **search()**



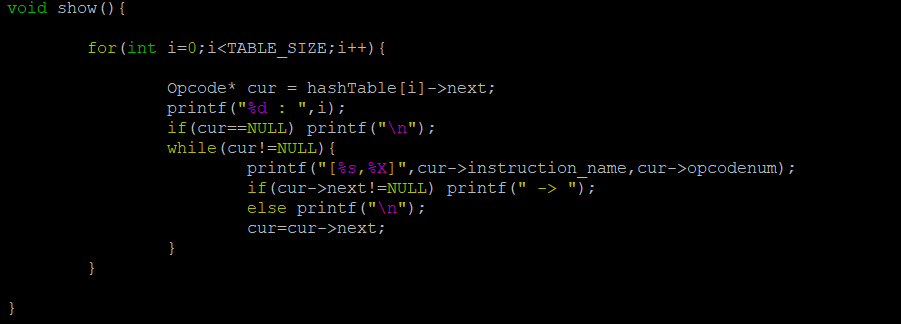
해당 모듈은 주어진 mnemonic이 hashTable내에 존재하는지, 존재한다면 해당 opcode를 반환해주는 기능을 가진다. ‘opcode mnemonic’ input이 들어왔을 때 수행된다. 주어진 mnemonic의 첫 글자의 ASCII Code값 mod 20을 key로, mnemonic 문자열을 instruction 문자열로 parameter로 받은 다음, 전역 변수로 선언된 hashTable의 key번째 노드 리스트를 탐색한다. 만약 찾았을 시 해당 mnemonic의 opcode를 반환하고, 찾지 못했으면 -1을 반환한다.

* + 1. **add()**



해당 모듈은 main()초기에 주어진 opcode.txt를 기준으로 각 mnemonic과 opcode, 그리고 argument 조건을 담은 Opcode\* 형태의 노드와 key값을 인자로 받아 hashTable[key] Linked List에 노드를 추가하는 역할을 한다. 노드는 헤드 노드 바로 다음에 저장하는 방법을 사용했다. (add\_node\_in\_front)

* + 1. **show()**

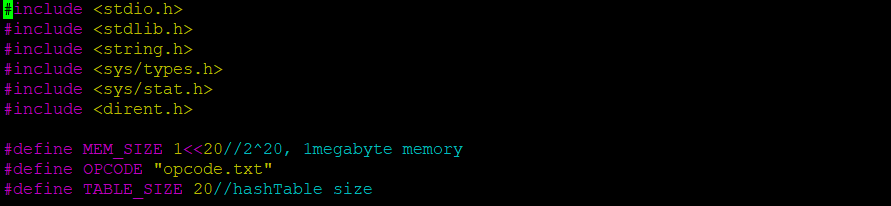


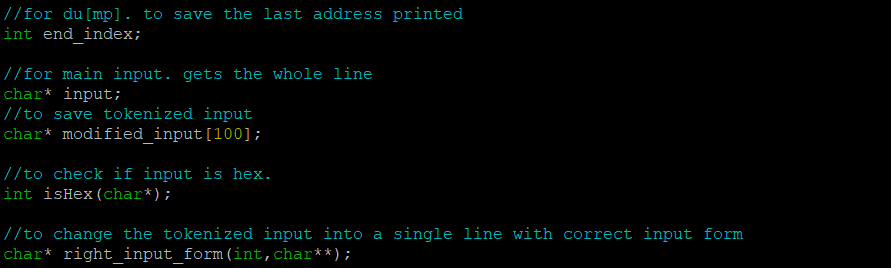
해당 모듈은 opcodelist가 input으로 들어왔을 때 main()에서 호출되며, hashTable의 모든 key값에 해당하는 Linked List를 모두 출력하는 기능을 가진다. 출력 형태는 다음과 같다.

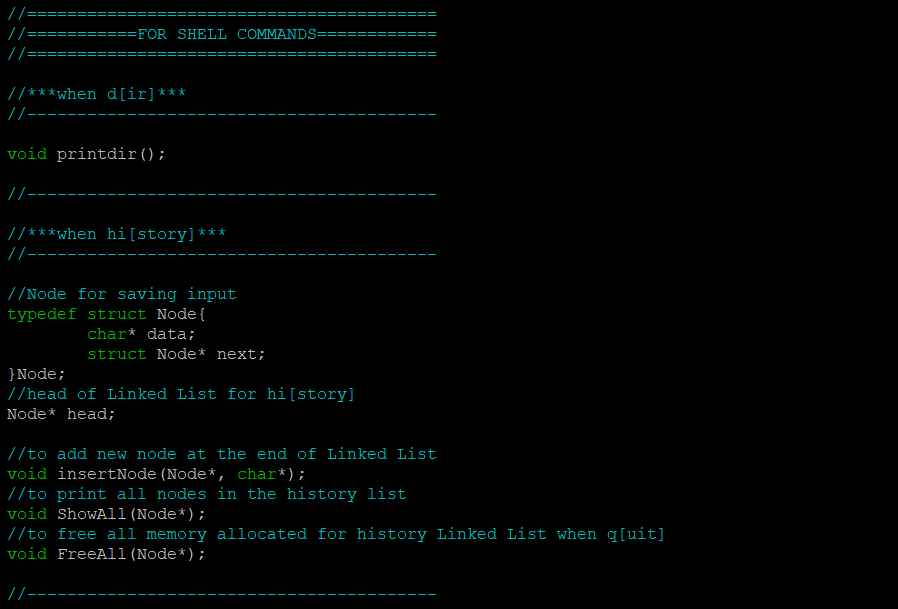


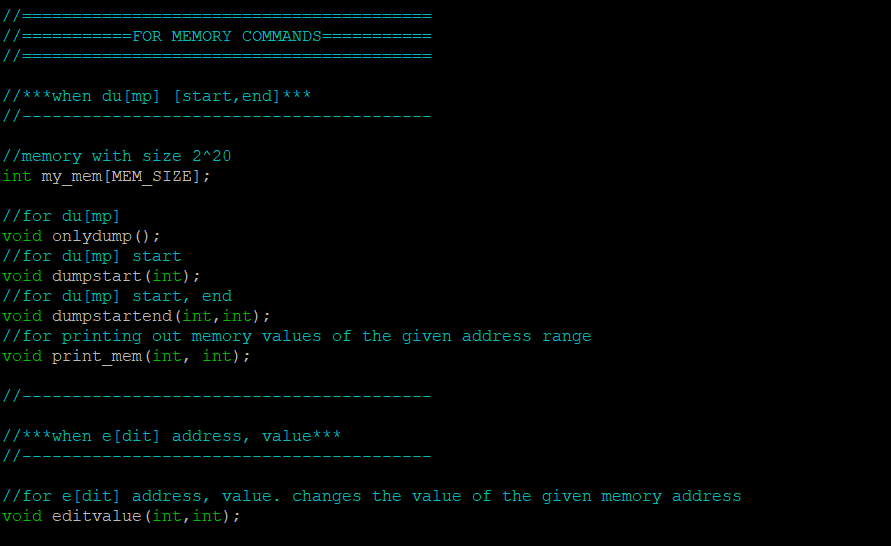
Opcode\* cur 포인터 변수를 활용하여 각각의 key값에 대응하는 Linked List의 노드들을 모두 출력해주었다.

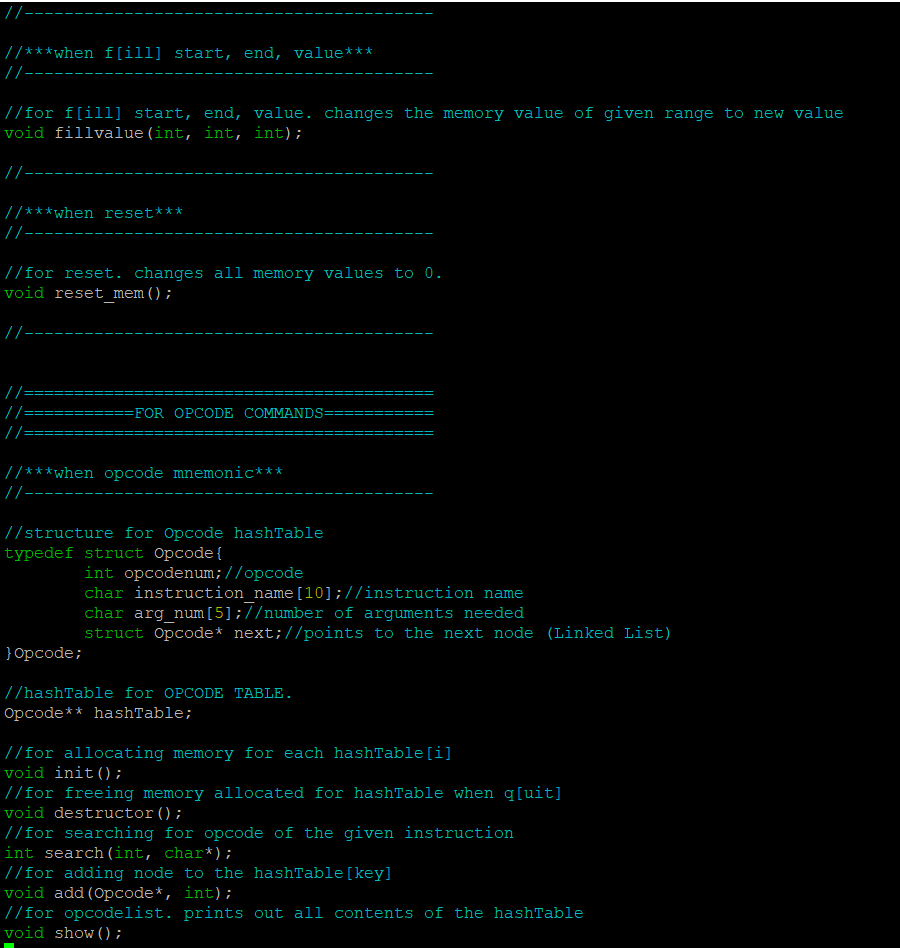
1. **기타 코드**
   1. **20171662.h**











* 1. **makefile**

