**과목명: 시스템프로그래밍**

**2분반**

**<<Project #3>>**

**서강대학교 [컴퓨터공학과]**

**[20171666]**

**[이예은]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 정의한 모듈에 대해 간략적인 설명
4. **전역 변수 정의**
5. **코드 설명**
6. **프로그램 개요**

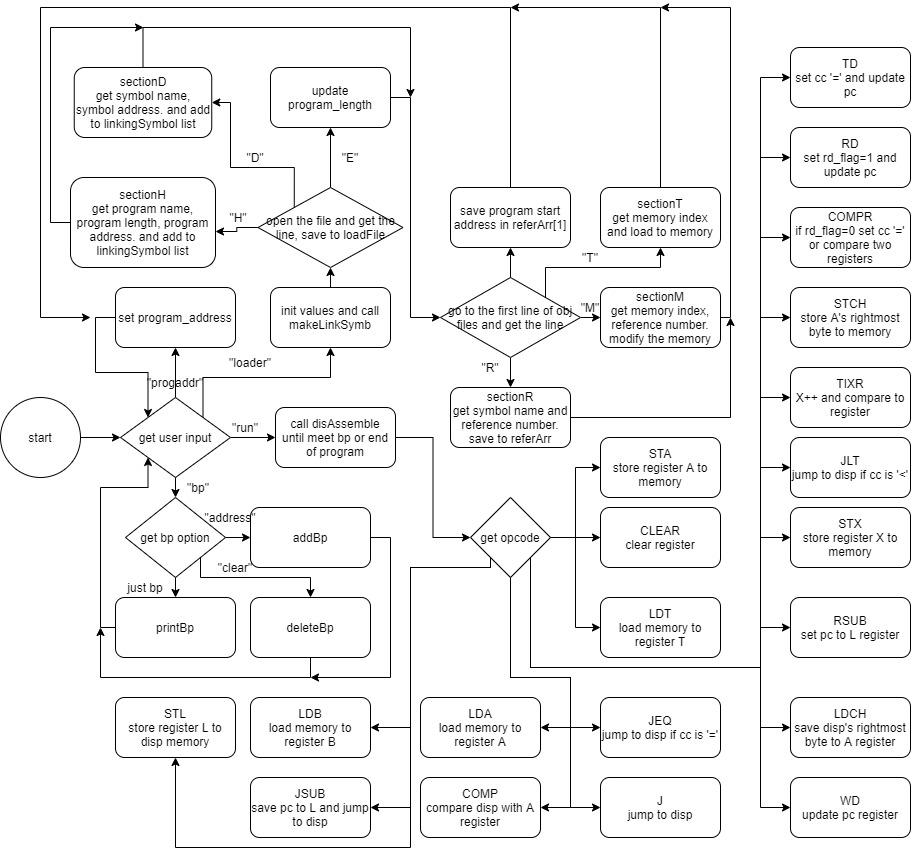
이 프로그램은 지난 프로젝트 2에서 만든 SIC/XE 머신에 추가로 linking loader를 구현한 프로그램입니다. 추가된 명령어는 progaddr, loader, bp, run입니다. 이전 assembler에서 생성된 obj 파일을 읽어 이를 link 시켜 메모리에 load하는 기능을 수행합니다. 또한 load된 메모리를 읽어 프로그램을 실행시킵니다. 이전 프로젝트와 마찬가지로 사용자가 명령을 입력하면 그에 따른 결과를 보여주는 방식으로 구현된 프로젝트입니다.

1. **프로그램 설명**

프로그램의 전반적인 동작은 이전 프로젝트와 같으며, loader 명령어의 경우 크게 2개의 pass로 이루어집니다. 첫번째 pass에서 프로그램들을 link하고 그에 따른 symbol table을 만듭니다. 이후 pass2에서 load하는 과정을 수행합니다. Run 명령어는 사용자가 지정한 breakpoint에 멈춰가며 메모리에 load된 프로그램을 실행합니다. 이전과 마찬가지로 사용자가 quit 명령을 입력하기 전까지 프로그램이 실행됩니다.

* 1. 프로그램 흐름도

이전 프로젝트에서 추가된 부분만 흐름도를 만들었습니다.



1. **모듈 정의**

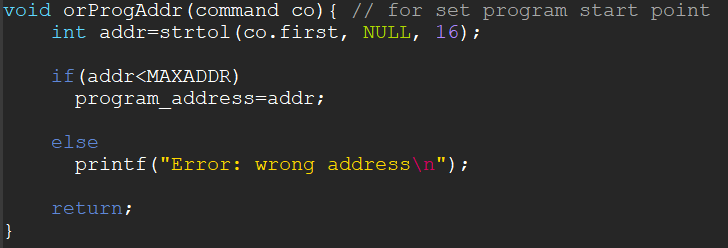
추가 및 수정된 함수만 설명하였습니다.

|  |  |
| --- | --- |
| Void orProgAddr(command) | Progaddr 명령을 수행합니다. Command를 인자로 받아 범위를 넘어가지 않는 주소가 들어오면 program 시작주소를 그 주소로 바꿔줍니다. 넘어간다면 에러 메시지를 출력합니다. |
| Void orLoader(command, char[]) | Loader 명령을 수행합니다. Command와 유저입력을 인자로 받아 받은 파일에 맞춰 makeLinkSymb를 불러 pass1을 수행하고, 메모리에 load하는 pass2 과정을 수행합니다. |
| Void makeLinkSymb(char[]) | Obj 파일이름을 인자로 받아 파일을 열어 linking을 위한 pass1을 수행합니다. 파일을 읽어 linkingSymbol 리스트를 추가하고 load 맵을 출력합니다. 또한 loadFileAdd를 불러 obj 파일을 저장합니다. |
| Void addLinkSymb(int, char[]) | 주소와 symbol의 이름을 인자로 받아 linkingSymbol 리스트에 추가합니다. |
| Void loadFileAdd(char[]) | 파일 한줄을 받아 obj 파일 저장용 리스트에 넣어줍니다. |
| Void loadFileDelete() | Obj 파일 저장용 리스트를 초기화해줍니다. |
| Void sectionH(char[]) | H 라인을 인자로 받아 프로그램 이름, 시작주소, 길이를 구해 linkingSymbol 리스트에 추가해줍니다. 또한 길이를 current\_length에 넣어줍니다. |
| Void sectionD(char[]) | D 라인을 인자로 받아 symbol 이름, 주소를 구해 linkingSymbol 리스트에 추가해줍니다. |
| Void sectionR(char[]) | R 라인을 인자로 받아 reference number와 symbol 이름을 구해 이름으로 symbol의 주소를 구합니다. 이후 referArr 배열에 reference number에 맞춰 주소를 넣어줍니다. |
| Void sectionT(char[]) | T 라인을 인자로 받아 시작주소와 T 라인의 길이를 구합니다. 이후 길이에 맞춰 라인을 읽고 이를 memory에 차례대로 load 해줍니다. |
| Void sectionM(char[]) | M 라인을 인자로 받아 시작주소와 바꾸려는 길이, reference number, 연산자를 구합니다. 이후 시작주소와 길이에 따라 메모리에 접근해 값을 얻고 reference number에 따라 referArr에 접근하여 둘을 연산해줍니다. 이후 다시 메모리에 올려줍니다. |
| Char\* getSubstring(int, int, char[]) | 시작 인덱스와 끝 인덱스, 문자열을 인자로 받습니다. 이후 시작 인덱스부터 끝 인덱스까지의 값을 새로운 문자열에 넣어 이 문자열을 반환해줍니다. |
| Void orBp(command, char[]) | Command와 유저입력을 인자로 받습니다. Bp만 받았다면 printBp를 불러 bpList를 출력해주고 clear를 받았다면 deleteBp를 불러 bpList를 초기화해줍니다. 주소값을 받았다면 load 한 프로그램의 범위안에 있는 경우에 bpList에 추가해줍니다. 아니라면 에러 메시지를 출력합니다. |
| Void printBp() | bpList를 출력해줍니다. |
| Void addBp(int) | Bp의 위치를 인자로 받아 bpList에 추가해줍니다. |
| Void deleteBp() | bpList를 초기화해줍니다. |
| Void orRun() | Run 명령을 수행합니다. 처음 프로그램을 run 하는 것이라면 register를 초기화해주고 시작합니다. 이후 bp를 만나거나 프로그램의 끝에 도달할 때까지 disAssemble을 불러 프로그램을 실행합니다. |
| Void initRegister() | Register 배열을 초기화해줍니다. 이 때 pc와 L 레지스터는 각각 프로그램 시작주소와 프로그램 길이를 넣어줍니다. |
| Void printRegister() | Register를 출력합니다. |
| Void disassemble(int) | Pc register의 값을 인자로 받아 메모리의 처음을 읽어 opcode와 ni 값을 뽑아냅니다. 이후 이를 executeInstruction으로 넘겨주어 opcode에 맞게 프로그램이 실행되게 합니다. |
| Void executeInstruction(int, int, int) | Opcode, ni, pc register의 값을 인자로 받아 opcode에 맞는 동작을 수행합니다. |
| Void opStore(int, int, int) | Opcode 중 ST 의 opcode를 수행합니다. Register number, pc register의 값, xbpe의 값을 인자로 받아 형식에 맞게 disp를 구하고 이를 메모리에 저장해줍니다. |
| Void opLoad(int, int, int, int) | Opcode 중 LD 의 opcode를 수행합니다. Register number, pc의 값, xbpe, ni의 값을 인자로 받아 형식에 맞게 disp 및 값을 구하고 이를 알맞은 register에 넣어줍니다. |
| Void opCondJump(int, int, int, int) | Opcode 중 조건에 따라 jump하는 opcode를 수행합니다. Condition, pc의 값, xbpe, ni를 인자로 받아 형식에 맞게 disp 및 값을 구하고 condition이 현재 cc와 같으면 pc register에 disp를 넣어줍니다. |
| Int getDisp(int, int, int) | 형식, pc의 값, xbpe를 인자로 받아 형식에 맞게 메모리에 저장되어 있는 상대적인 disp 값을 뽑아냅니다. 이후 xbpe의 값에 따라 pc 또는 base, x register와 더해주고 이를 반환해줍니다. |
| Int calculateDisp(int, int, int) | Disp와 ni, xbpe를 인자로 받아 ni의 addressing 방식에 따라 disp가 가리키는 메모리의 값을 getValue를 통해 구해 반환해줍니다. |
| Int getValue(int) | Disp를 인자로 받아 disp가 가리키는 메모리의 값을 구해 반환합니다. |
| Int isStop() | 현재 pc register의 값과 bpList에 있는 값을 비교해 같다면 1을 반환하고 없다면 0을 반환합니다. |

1. **전역 변수 정의**

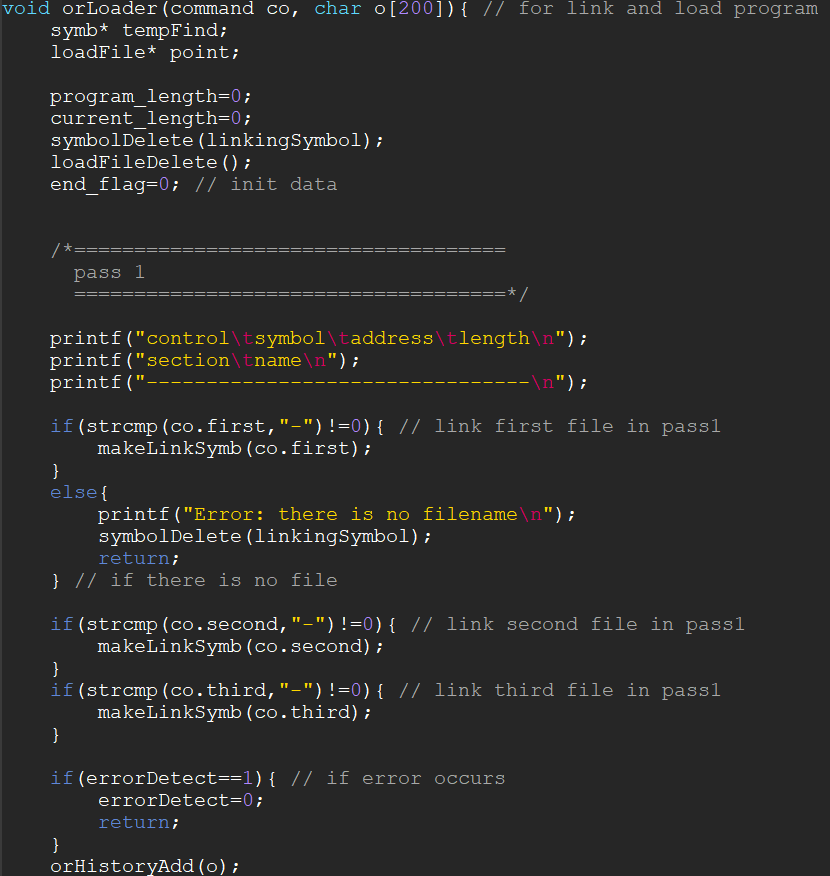
|  |  |
| --- | --- |
| loadFile | Obj 파일을 저장하기 위한 구조체. 한 줄을 저장하기 위한 문자열과 다음 줄을 가리키는 포인터로 이루어져 있다. |
| breakP | Break point를 저장하기 위한 구조체. Bp의 위치를 나타내는 loc와 다음 bp를 가리키는 포인터로 이루어져 있다. |
| Int program\_address | 프로그램의 시작 주소 |
| Int program\_length | 전체 프로그램의 길이 |
| Int current\_length | 현재 linking중인 프로그램의 길이 |
| Int errorDetect | 에러를 확인하기 위한 flag |
| Int referArr[256] | Reference number에 따라 reference symbol의 주소를 저장하는 배열 |
| Int reg[10] | Register 배열. |
| Int cc | >=< 의 condition을 저장하는 변수 |
| Int rd\_flag | RD opcode를 위한 flag |
| Int end\_Flag | 프로그램의 끝을 확인하기 위한 flag |
| Int stop\_flag | 프로그램이 bp에 의해 멈췄는지를 확인하기 위한 flag |
| Symb\* linkingSymbol | Linking의 symbol을 저장하는 table. |
| loadFile\* Lst | Obj 파일 list의 시작 포인터 |
| loadFile\* Led | Obj 파일 list의 끝 포인터 |
| breakP\* bpList | Break point의 list |

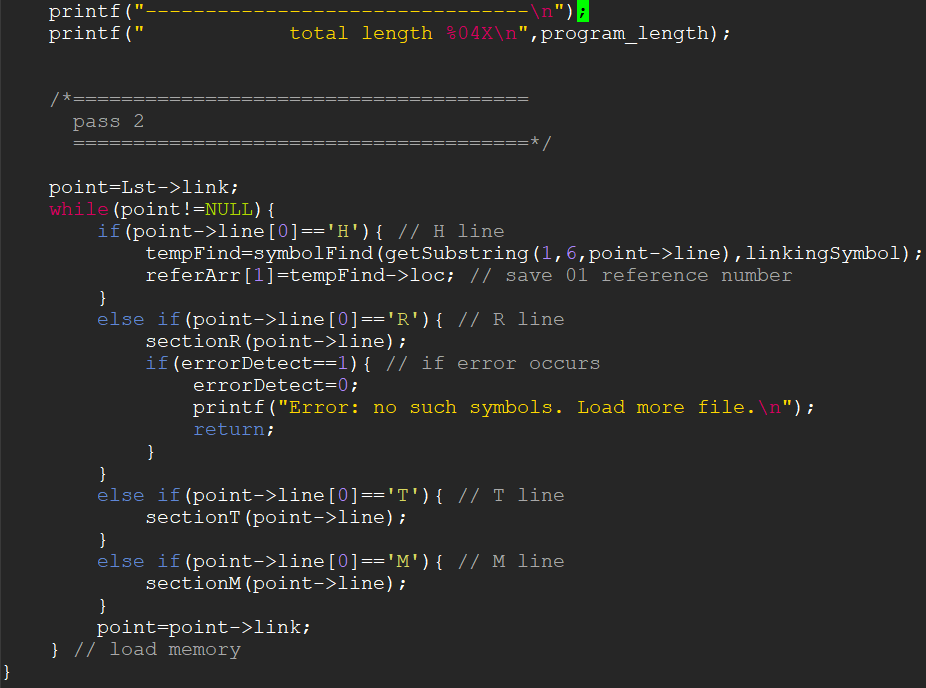
1. **코드 설명**
   1. orProgAddr



Command를 인자로 받아 co.first에 저장되어 있는 주소를 얻는다. 이 주소값이 MAXADDR보다 작은 경우에만 program\_address에 넣어주어 프로그램 시작 주소를 저장한다. 만약 MAXADDR보다 크다면 메모리의 범위를 넘어간 것이므로 에러 메세지를 출력한다.

* 1. orLoader





Command와 유저입력을 인자로 받는다. 먼저 linking loader에 필요한 각종 변수들과 list들을 초기화해준다. 이후 첫번째 파일 이름을 인자로 makeLinkSymb를 부른다. 만약 파일 이름을 받지 않았다면 에러 메시지를 출력하고 linking symbol list를 초기화해준다. 이후 두번째, 세번째 파일 이름을 받았다면 이에 따라 차례로 makeLinkSymb를 불러준다.

pass1을 수행하는 makeLinkSymb에서 에러가 났는지 확인하기 위해 errorDetect를 검사해 1이면 그대로 함수를 종료하고 아니라면 history list에 추가 및 total length를 출력해 load map을 완성한다. 이후 pass2로 진입한다.

Pass2에서는 Linking symbol을 만들면서 obj 파일을 저장한 loadfile list를 차례대로 순회하며 메모리에 load하는 작업을 한다. H 라인을 만났다면 reference number 01이 프로그램의 시작주소 이므로 H 라인의 프로그램 이름을 읽어 이를 이용해 symbolFind로 시작주소의 loc를 얻는다. 이후 이를 referArr[1]에 저장해준다.

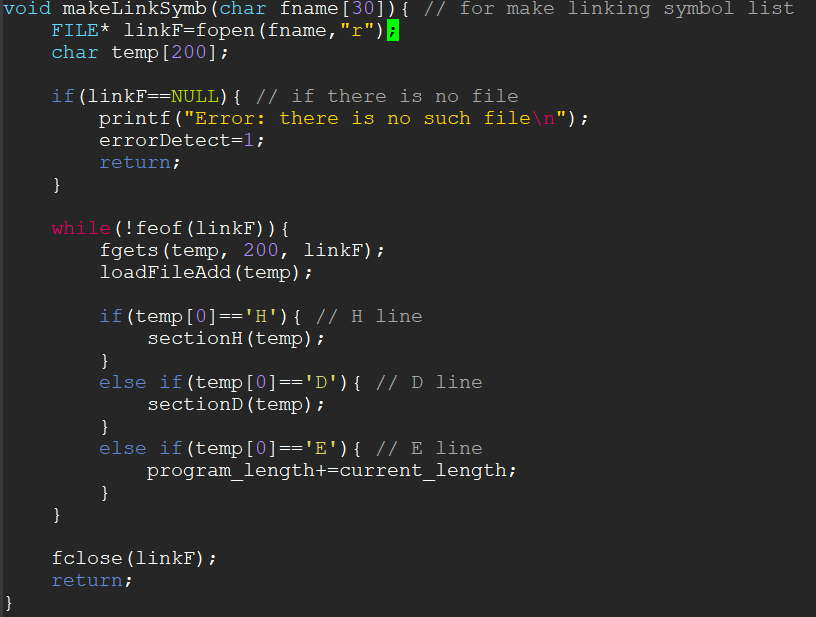
R 라인을 만났다면 sectionR을 불러 R라인을 처리해준다. 처리 중 에러가 발생했는지 확인하기 위해 errorDetect를 검사하고 발생했다면 에러 메시지 출력 후 함수 종료, 아니라면 다음 줄로 넘어간다.

E 라인을 만났다면 sectionT를 불러 T 라인을 처리해준다.

M 라인을 만났다면 sectionM을 불러 M 라인을 처리해준다.

위 과정을 loadfile list의 끝에 다다를 때까지 반복한다.

* 1. makeLinkSymb



Pass1인 linking symbol list를 만드는 함수이다. 먼저 인자로 받은 파일 이름으로 해당 obj 파일을 연다. 만약 파일이 없다면 에러 메시지를 출력하고 errorDetect를 1로 설정한 후 함수를 종료한다.

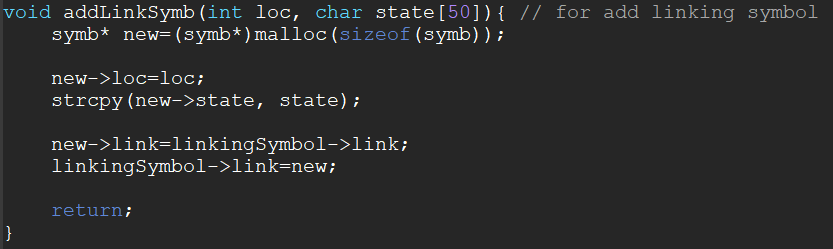
파일을 eof에 도달할 때까지 한 줄씩 읽으면서 loadFileAdd를 불러 loadfile list에 추가해준다. 또한 각각 line에 따라 처리해준다. H 라인을 만났다면 sectionH를 불러 H 줄을 처리해준다.

D 라인을 만났다면 sectionD를 불러 D 라인을 처리해준다.

E 라인을 만났다면 program\_length에 current\_length를 더해 전체 프로그램 길이를 업데이트 해준다.

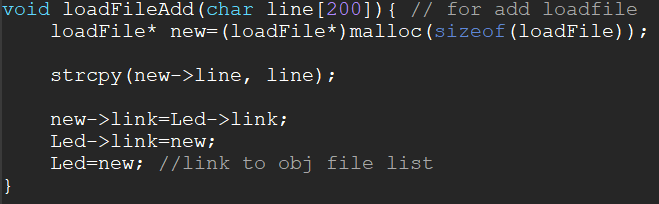
모든 과정이 끝났다면 파일을 닫고 함수를 종료한다.

* 1. addLinkSymb



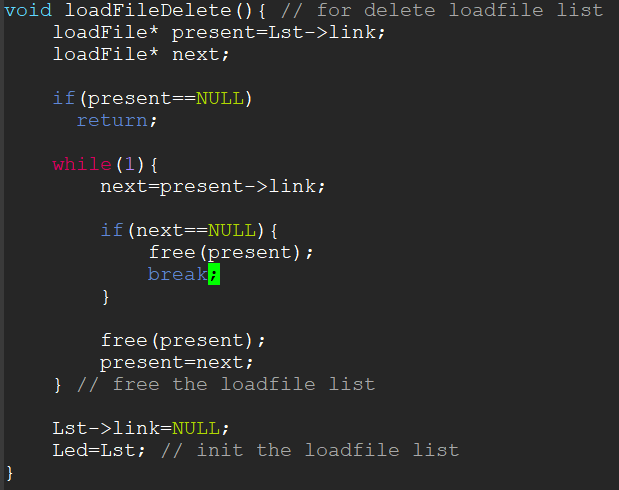
Symbol의 주소와 이름을 인자로 받아 linkingSymbol list에 추가해준다.

* 1. loadFileAdd



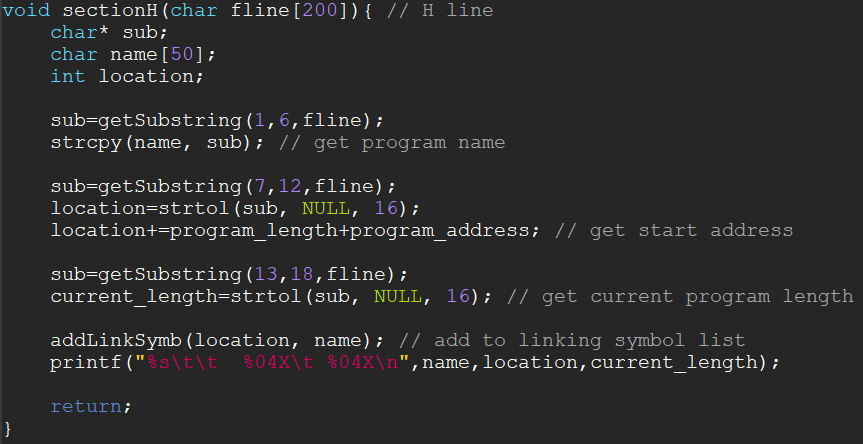
파일의 한 줄을 인자로 받아 obj 파일을 저장하는 list에 추가해준다.

* 1. loadFileDelete



Loadfile list를 초기화해준다. Lst, list의 시작지점부터 끝 지점까지 순회하며 free 해주고 시작점을 NULL로 초기화한다.

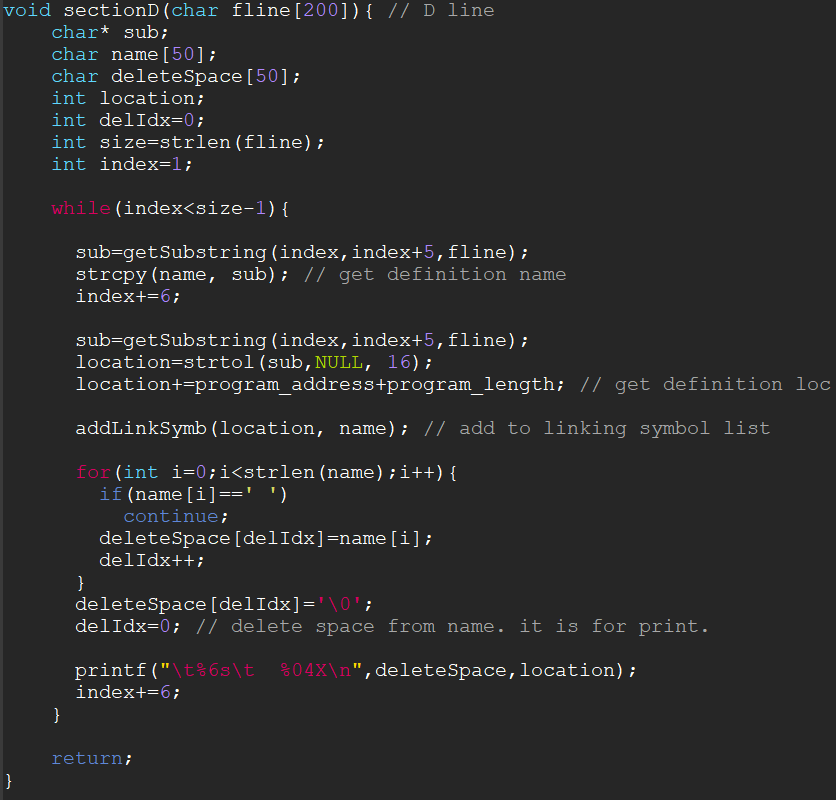
* 1. sectionH



파일의 한 줄을 인자로 받아 getSubstring 함수를 이용해 자른다.

1-6까지는 프로그램의 이름이므로 이를 name에 저장하고, 7-12는 프로그램의 시작주소로 이를 location에 저장한다. 13-18은 현재 프로그램의 전체 길이로 current\_length를 갱신한다. 이후 이 정보들을 addLinkSymb에 인자로 넘겨주어 linkingSymbol list에 추가한다. 이후 H 라인의 load map을 출력한다.

* 1. sectionD

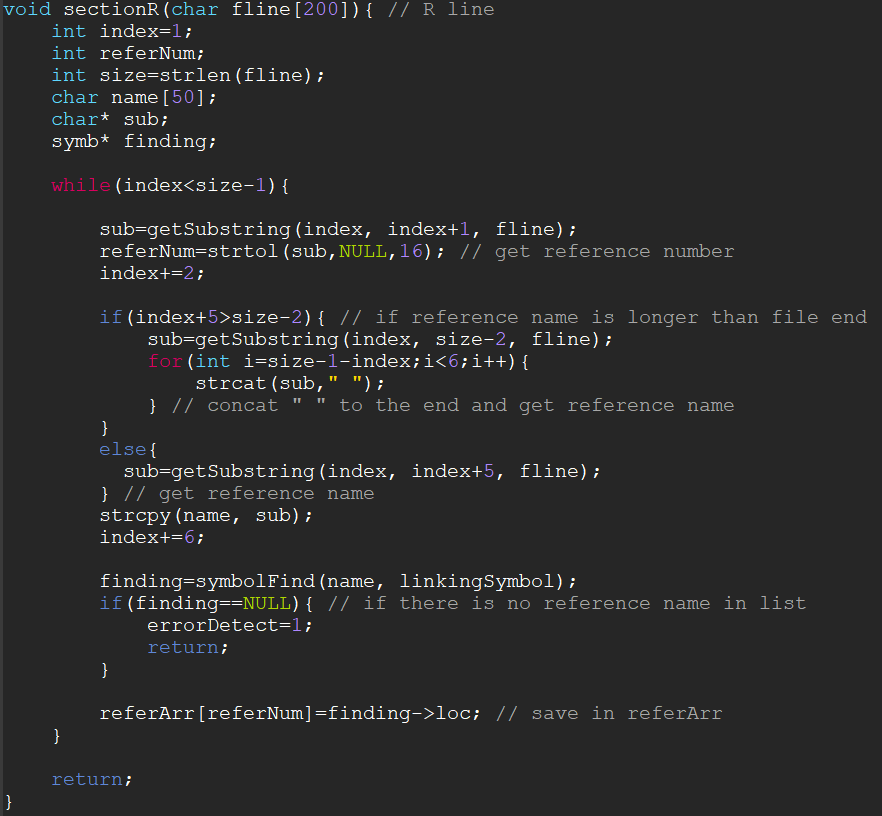


D 라인을 인자로 받아 getSubstring으로 적절히 잘라 원하는 값을 얻는다.

D 라인의 끝까지 돌며 symbol의 이름을 index~index+5의 6자리 문자열로 받아 저장하고 이후 6자리로 symbol의 주소를 받아 program\_address와 program\_length를 더해 실제 주소를 location에 저장한다. 이 값들을 인자로 주어 addLinkSymb를 불러 symbolList에 추가해준다.

이후 출력양식을 위해 name의 뒤의 공백을 제거하고 load map을 출력한다.

* 1. sectionR

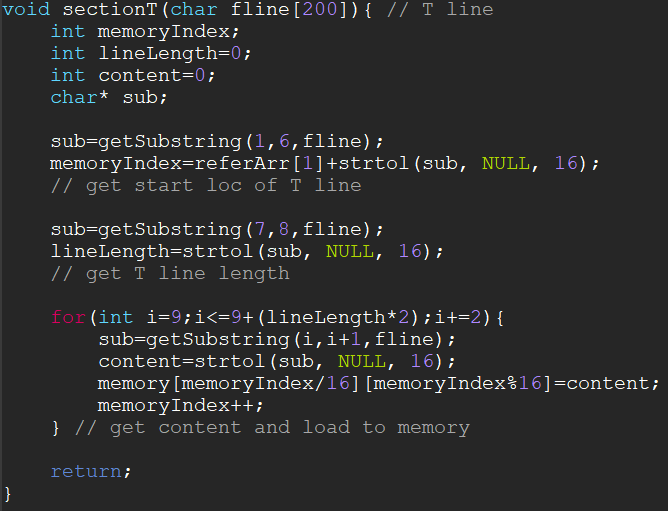


R 라인을 인자로 받아 getSubstring을 이용해 적절히 잘라 원하는 값을 얻는다.

R 라인의 끝까지 돌며 index~index+1로 2자리의 reference number를 얻고 다음 6자리의 symbol 이름을 얻는다. 만약 symbol의 이름이 6자리보다 작다면 탐색을 용이하게 하기 위해 뒤에 공백을 넣어 6자리를 맞춰준다.

이후 얻는 이름을 이용해 symbFind 함수를 불러 해당 symbol의 주소를 얻고 이를 referArr에 reference number에 따라 넣어준다.

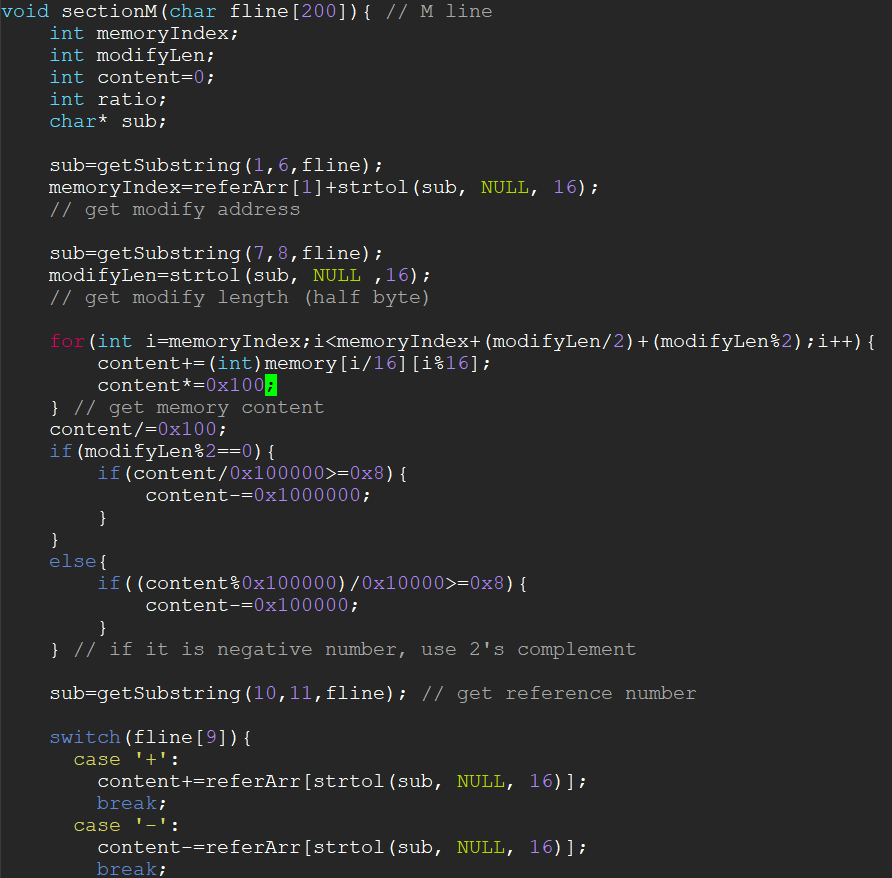
* 1. sectionT



T 라인을 인자로 받아 getSubstring으로 적절히 잘라 원하는 값을 얻는다.

먼저 1~6은 T 라인의 시작 주소로 이와 프로그램 시작주소를 더해 T 라인의 내용을 저장할 memory의 위치를 얻는다. 이후 T 라인을 2개씩 끊어 끝까지 돌며 해당 인덱스부터 차례대로 메모리에 넣어준다.

* 1. sectionM



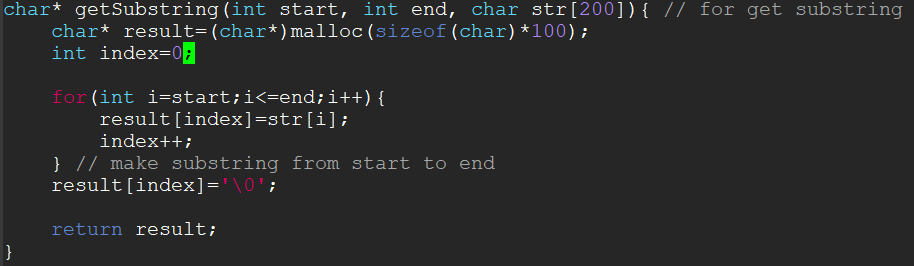
M 라인을 인자로 받아 getSubstring으로 적절히 잘라 원하는 값을 얻는다. 먼저 처음 1~6은 modify 하려는 시작주소로 memoryIndex에 저장하고 7~8은 modify 하려는 half byte의 개수이므로 이를 modifyLen에 저장해준다.

memoryIndex부터 memory에 접근해 modifyLen 만큼 읽어 content에 저장해준다. 이 때 만약 content의 가장 왼쪽 값이 0x8 보다 크다면 이는 음수임을 뜻하므로 2의 보수법을 활용해 음수로 바꿔준다.

이후 10~11은 reference number로 referArr에 알맞게 접근해 modify 연산을 위한 값을 얻고, 9는 연산자로 이에 맞게 content에 연산해준다.

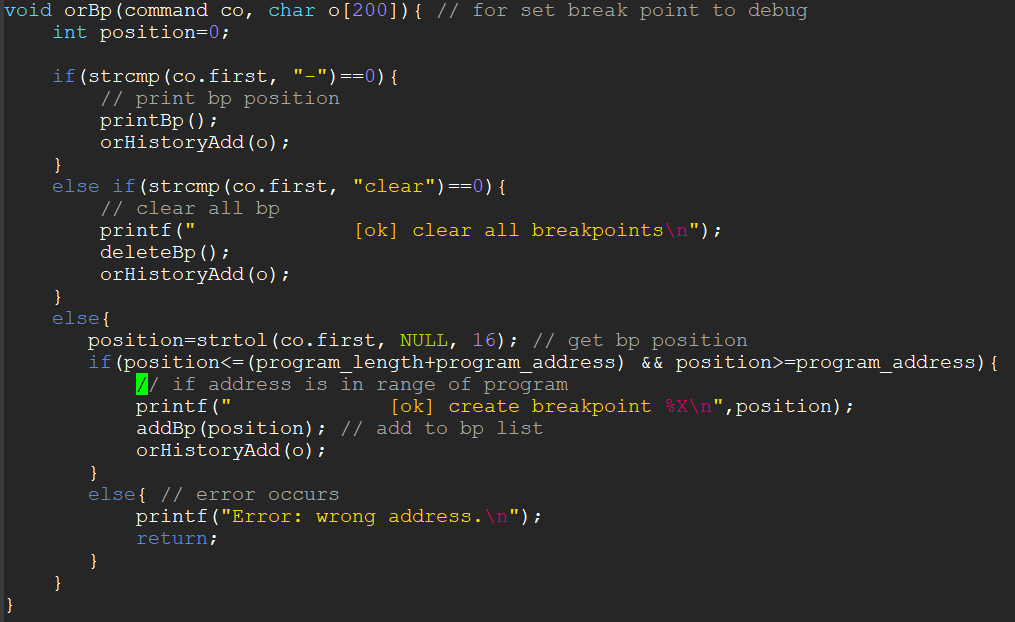
같은 주소에 연산을 여러 번 할 수 있으므로 modify 이후 content는 음수가 나올 수 있다. 이를 보정하기 위해 음수이면 2의 보수법을 활용해 보정해주고 다시 해당 memoryIndex의 memory에 접근해 차례대로 memory에 넣어준다.

* 1. getSubstring



자를 문자열의 시작 index와 끝 index, 문자열을 인자로 받는다. 이후 새로운 문자열에 start 부터 end 까지의 문자를 넣어주고 마지막에 \0를 넣어 문자열을 만들어 반환해준다.

* 1. orBp



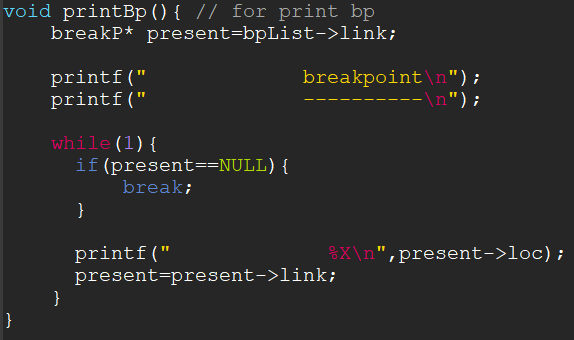
Command와 유저입력을 인자로 받아 bp 명령을 수행한다.

Bp 만 받았다면 printBp를 불러 bp를 출력하고 history list에 추가한다.

Clear를 받았다면 deleteBp를 불러 bplist를 초기화하고 history list에 추가한다.

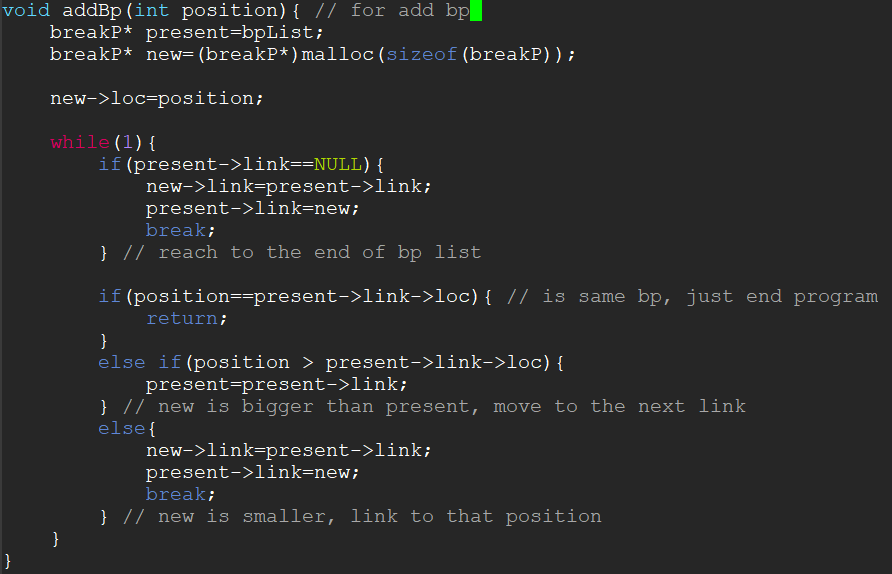
주소 값을 받았다면 이 주소 값이 load 한 프로그램의 범위 안에 있다면 addBp를 불러 bpList에 추가해준다. 이후 history list에도 추가한다. 범위 밖에 있다면 에러 메세지를 출력하고 함수를 종료한다.

* 1. printBp



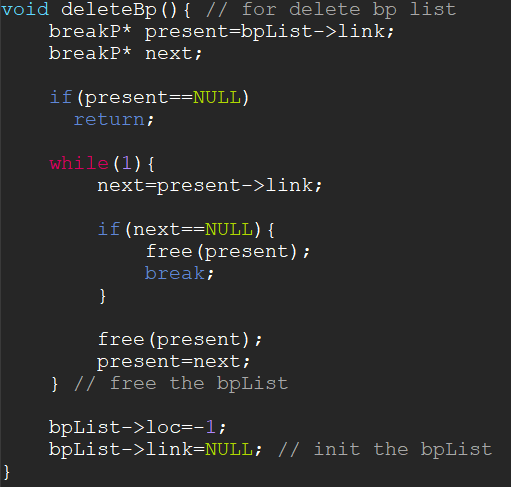
bpList를 list의 끝까지 돌며 break point들을 출력해준다.

* 1. addBp



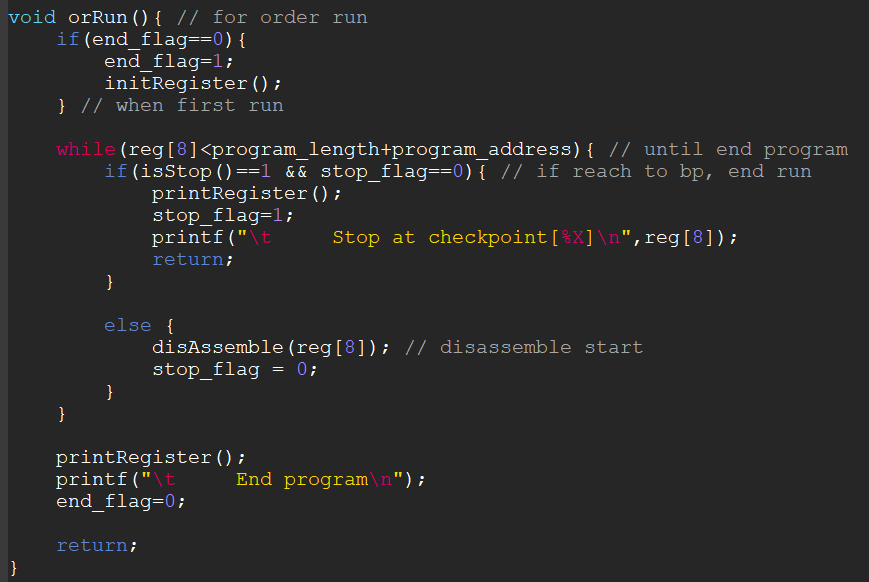
Bp의 주소를 인자로 받아 새로운 breakP 노드를 만들고 이를 bpList에 추가한다. 이 때 bpList를 끝까지 순회하며 list의 끝이나 나보다 주소값이 큰 노드를 만났다면 그 자리에 추가하고 아니라면 다음 노드로 이동하는 식의 삽입 정렬을 이용해 bpList에 추가한다. 만약 같은 주소의 bp가 이미 bpList에 있다면 바로 함수를 종료한다.

* 1. deleteBp



bpList를 끝까지 순회하며 노드들을 free 시켜준다. 이후 bpList의 처음을 초기화한다.

* 1. orRun

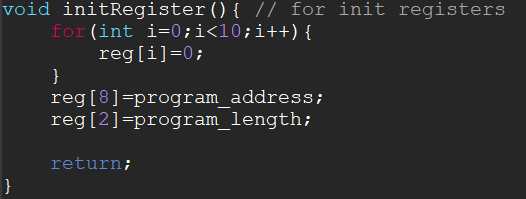


Run 명령을 수행한다. 가장 먼저 end\_flag가 0으로 프로그램이 처음 시작하는 것이라면 end\_flag를 1로 세팅해 진행 중인 프로그램이라는 표시를 하고 initRegister를 불러 register를 초기화한다.

이후 pc register의 값이 프로그램의 끝에 도달할 때까지 disAssemble에 pc 값을 인자로 넘겨주어 opcode를 계속해서 처리해준다.

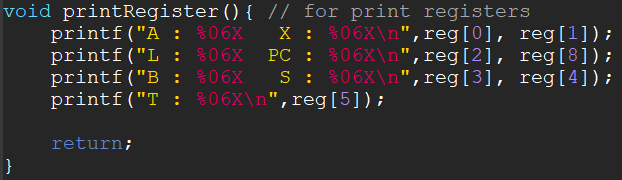
이 때 isStop 함수를 불러 1을 반환되어 bp에 걸렸다면 printRegister로 현재 register 값을 출력하고 멈췄다는 의미의 stop\_flag를 1로 세팅하고 함수를 종료한다.

* 1. initRegister



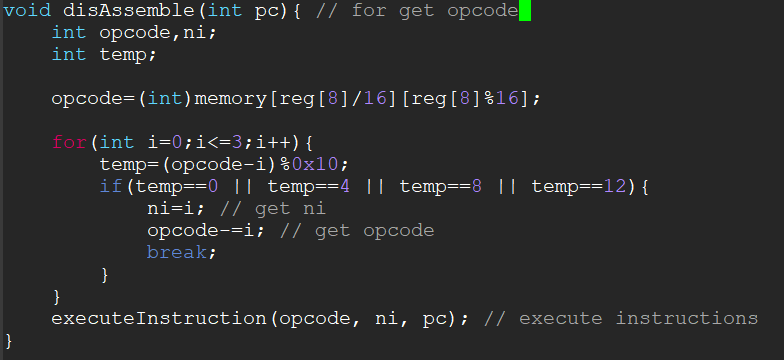
Register를 초기화해준다. 모든 register를 0으로 초기화하지만 pc는 program\_address를 넣어 프로그램 시작주소로, L은 program\_length를 넣어 프로그램의 길이로 세팅해준다.

* 1. printRegister



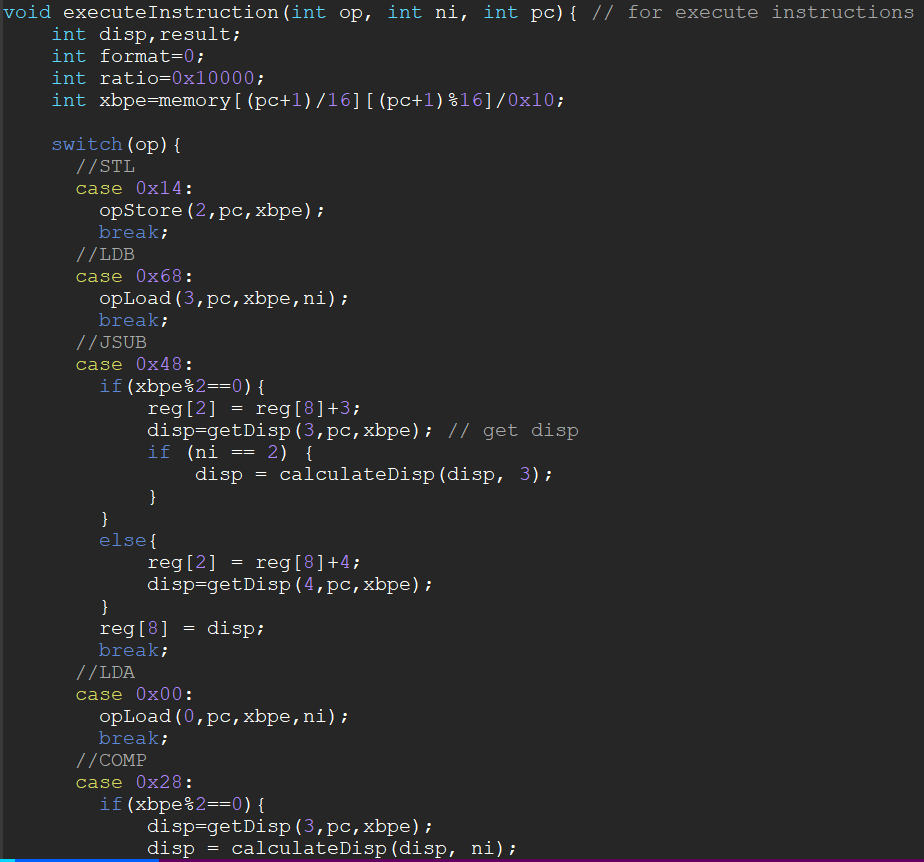
A, X, L, PC, B, S, T register의 값을 차례로 출력해준다.

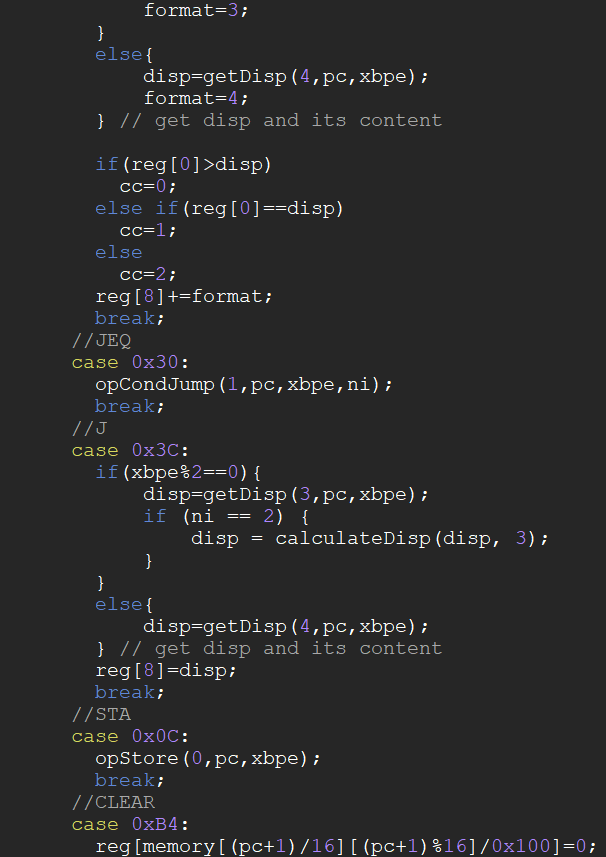
* 1. disAssemble

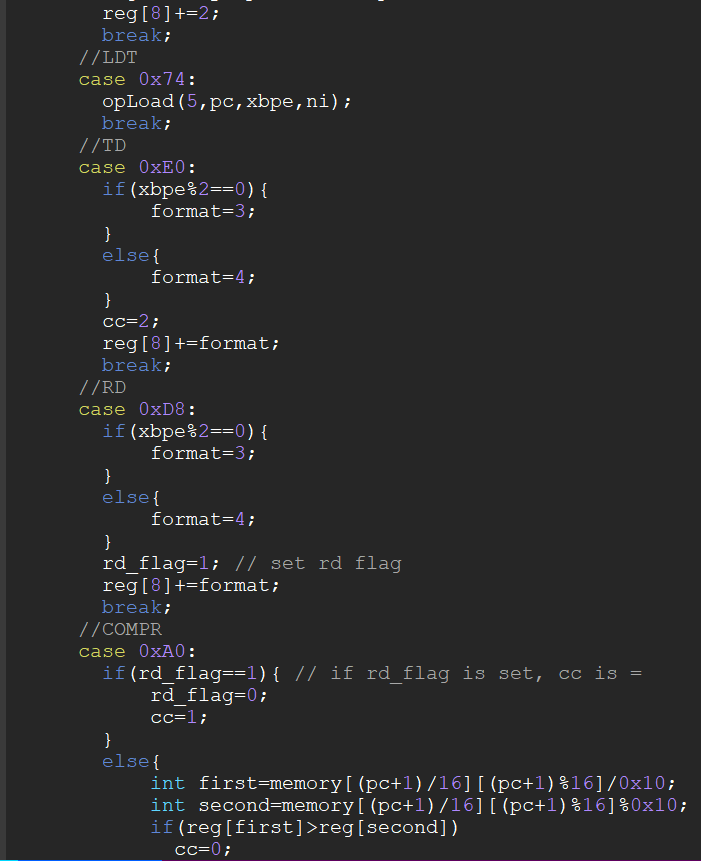


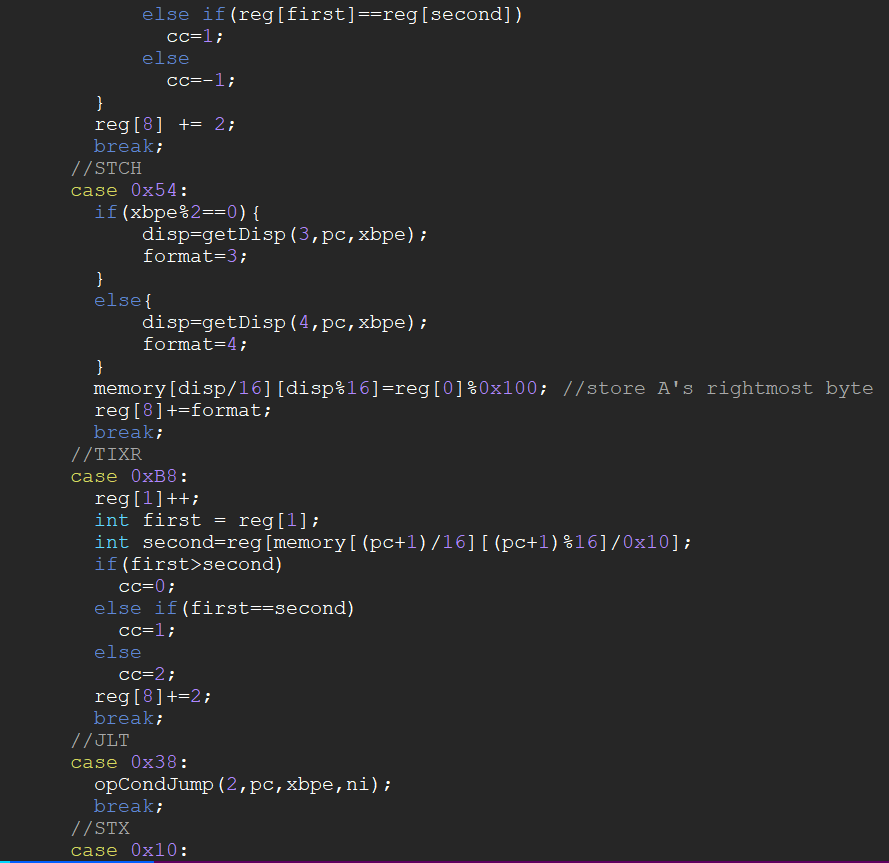
Pc register의 값을 인자로 받아 그 주소의 메모리에 접근해 1바이트를 읽어온다. 모든 opcode는 뒷자리가 0, 4, 8, 12로 끝나고 ni의 값은 0~3까지 가능하므로 0, 4, 8, 12로 끝나는 곳을 찾기 위해 for문을 통해 0~3까지 빼며 검사한다. 0, 4, 8, 12로 끝나는 지점을 찾았다면 그 때 뺀 값을 ni에 넣어주고 ni를 뺀 것이 opcode가 되므로 이를 저장한다. 이후 executeInstruction에 opcode, ni, pc의 값을 인자로 넘겨주어 opcode에 맞게 동작을 처리한다.

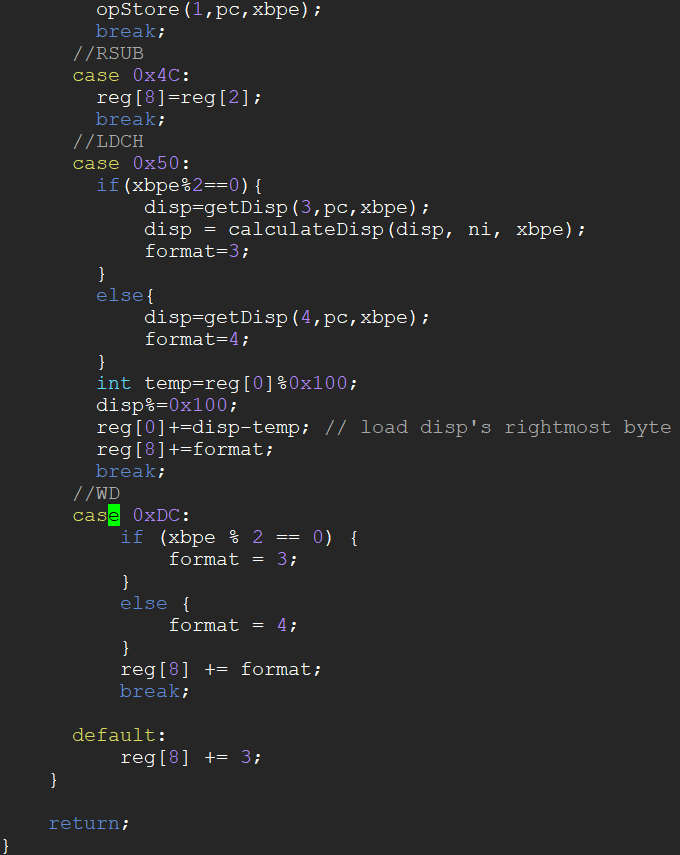
* 1. executeInstruction











Opcode, ni, pc의 값을 인자로 받아 opcode에 따라 명령을 수행한다.

STL: opStore 함수에 register number 2, pc, xbpe를 인자로 넣어 불러준다.

LDB: opLoad 함수에 register number 3, pc, xbpe, ni를 인자로 넣어 불러준다.

JSUB: xbpe로 형식에 따라 L register에 다음 pc 값을 넣어주고 getDisp와 calculateDisp를 통해 얻은 disp 값을 pc에 넣어준다.

LDA: opLoad 함수에 register number 0, pc, xbpe, ni를 인자로 넣어 불러준다.

COMP: xbpe로 형식에 따라 getDisp, calculateDisp를 불러 disp 값을 얻고 이를 A register와 비교해 cc에 값을 넣어준다. 이후 pc를 다음 instruction으로 갱신한다.

JEQ: opCondJump 함수에 condition 1, pc, xbpe, ni를 인자로 넣어 불러준다.

J: xbpe로 형식에 따라 getDisp, calculateDisp를 불러 disp 값을 얻고 이를 pc register에 넣어준다.

STA: opStore 함수에 register number 0, pc, xbpe를 인자로 넣어 불러준다.

CLEAR: pc 주소의 메모리에 접근해 register number를 얻고 해당 register number의 register를 0으로 초기화해준다. 이후 pc를 다음 instruction으로 갱신한다.

LDT: opLoad 함수에 register number 5, pc, xbpe, ni를 인자로 넣어 불러준다.

TD: cc를 2로 세팅하고 xbpe로 형식에 따라 pc register의 값을 갱신한다.

RD: xbpe로 형식에 따라 pc register의 값을 갱신하고 다음 COMPR에서 사용되는 rd\_flag를 1로 세팅한다.

COMPR: rd\_flag가 1이면 cc를 ‘=’인 1로 세팅하고 pc register의 값을 갱신한다. Rd\_flag가 0이면 pc 주소의 메모리에 접근해 2개의 register number를 얻는다. 이후 이 둘을 비교하여 cc를 >(0), =(1), <(2)로 세팅해준다. Pc register의 값을 갱신한다.

STCH: xbpe로 형식에 따라 getDisp를 불러 disp 값을 얻는다. 얻은 disp 주소 메모리에 A register의 가장 오른쪽 1byte의 값을 넣는다. 이후 pc register의 값을 갱신한다.

TIXR: X register의 값을 +1 해주고 pc 주소 메모리에 접근해 register number를 얻는다. 이후 X register의 값과 register number에 맞는 register의 값을 비교해 cc를 >(0), =(1), <(2)로 세팅해준다. Pc register의 값을 갱신한다.

JLT: opCondJump 함수에 condition 2, pc, xbpe, ni를 인자로 넣어 불러준다.

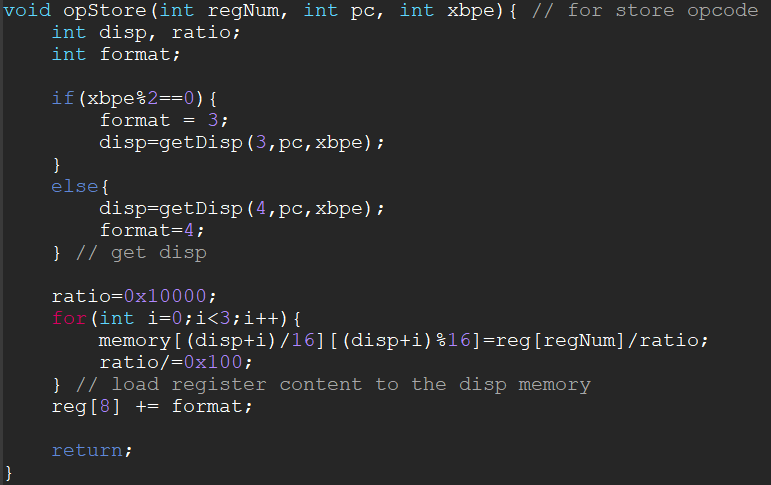
STX: opStore 함수에 register number 1, pc, xbpe를 인자로 넣어 불러준다.

RSUB: pc register에 L register의 값을 넣어준다.

LDCH: xbpe 형식에 따라 getDisp와 calculateDisp를 불러 disp 값을 얻는다. 이 disp의 rightmost byte를 구해 A register에 넣어준다. 이후 pc register를 갱신한다.

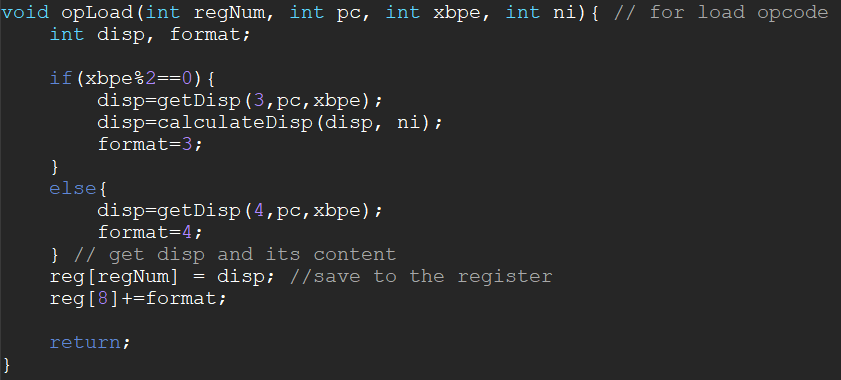
WD: xbpe로 형식에 따라 pc register를 갱신한다.

* 1. opStore



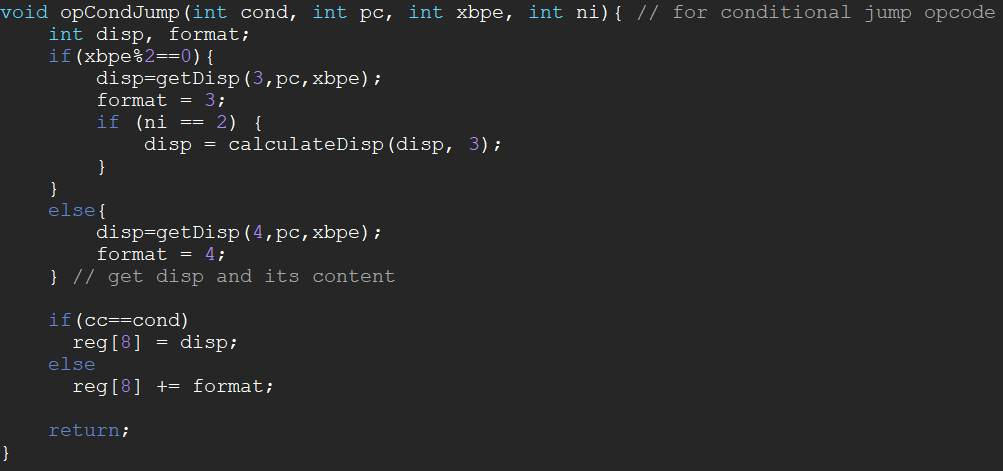
자주 나오는 ST의 opcode를 수행하기 위한 함수. Register number와 pc, xbpe를 인자로 받는다. 이후 xbpe를 통해 3, 4형식을 알아낸 후 이에 맞게 getDisp를 불러 disp 값을 얻는다. 이후 얻는 disp 주소의 메모리에 접근해 register number에 맞는 register 값을 넣어준다. Pc register도 갱신해준다.

* 1. opLoad



자주 나오는 LD의 opcode를 수행하기 위한 함수. Register number와 pc, xbpe, ni 값을 인자로 받는다. 먼저 xbpe를 통해 3, 4형식의 정보를 알아내고 이를 이용해 getDisp를 불러 disp 주소를 얻는다. 얻은 disp와 ni를 인자로 calculateDisp를 불러 addressing에 맞게 메모리 값을 얻는다. 이후 register number에 맞는 register에 그 값을 넣어준다. Pc register도 갱신해준다.

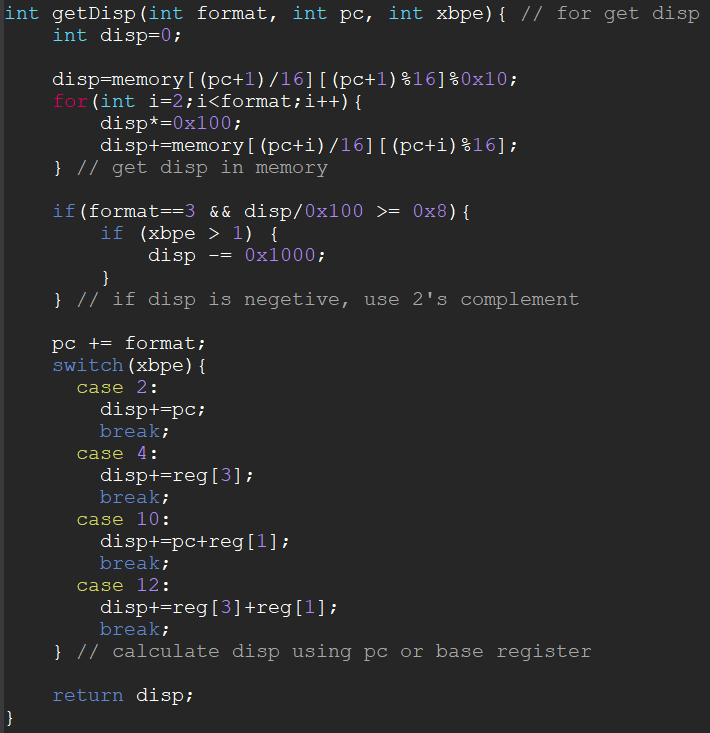
* 1. opCondJump



자주 나오는 조건에 따라 jump하는 opcode를 수행한다. 조건 값 cond와 pc, xbpe, ni를 인자로 받는다. 먼저 xbpe를 이용해 3, 4형식에 맞게 getDisp를 불러 disp를 얻는다. 이후 jump는 원래 disp의 값을 그대로 사용하지만 @로 indirect addressing일 때만 disp를 주소로 해 메모리 값을 한 번 더 얻어야 하므로 ni가 2일 때 calculateDisp를 불러 그에 맞게 값을 얻어준다.

이후 현재 cc가 조건 값 cond와 같은 경우에 pc에 얻은 disp 값을 넣어주고 아니라면 format에 맞게 다음 opcode로 pc를 갱신해준다.

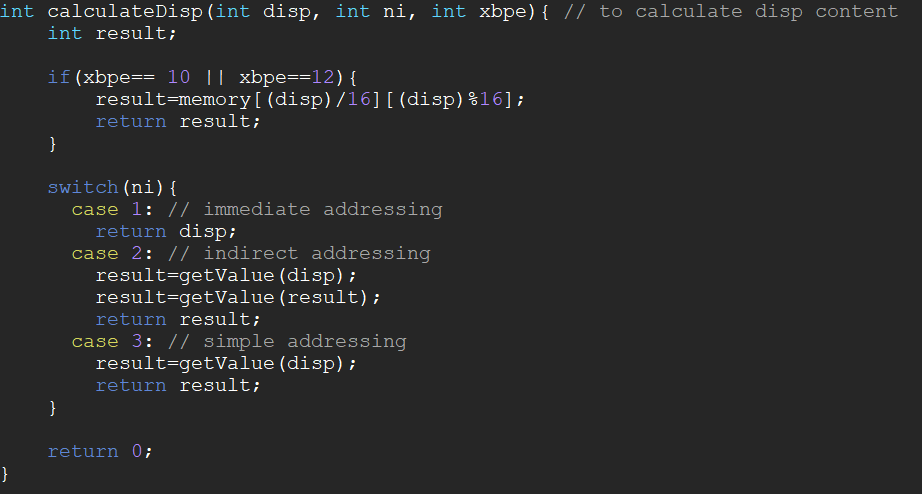
* 1. getDisp



형식과 pc, xbpe를 인자로 받아 disp를 계산한다. 먼저 pc 주소의 메모리에 접근해 형식에 따라 disp를 얻는다. 만약 형식이 3형식이고 얻은 disp의 가장 왼쪽 값이 0x8보다 크다면 음수를 의미하므로 2의 보수를 이용해 음수로 바꿔준다. 또한 disp 연산 시 pc는 다음 instruction을 가리켜야 하므로 pc에 format을 더해 연산하게 한다.

이후 xbpe에 따라 pc, base, x register 값을 적절히 더해 최종 disp 값을 얻어 이를 반환해준다.

* 1. calculateDisp



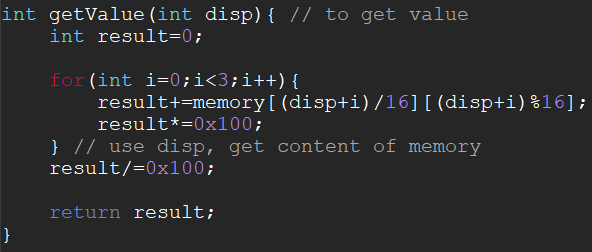
Disp와 ni를 인자로 받는다. Ni에 따라 addressing 기법이 달라지므로 ni가 1이면 immediate addressing으로 그냥 disp를 반환한다.

Ni가 2이면 indirect addressing으로 disp 주소의 메모리 값을 얻는 과정을 2번 반복한다. 이후 얻은 값을 반환한다.

Ni가 3이면 simple addressing으로 disp 주소의 메모리 값을 얻는 과정을 1번 반복한다. 이후 얻은 값을 반환한다.

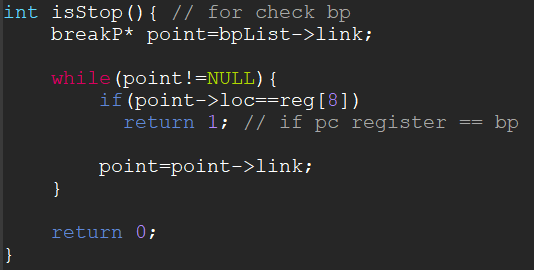
만약 xbpe가 10 또는 12이면 x register를 사용해 인덱스로 접근하는 것이므로 getValue를 사용하지 않고 바로 그 주소의 값 1byte 만 받아 반환해준다.

* 1. getValue



Disp를 인자로 받아 이 주소의 메모리에 접근한다. 이후 해당 주소부터 3바이트를 읽어 저장하고 이를 반환해준다.

* 1. isStop



bpList를 끝까지 순회하며 pc register의 값과 비교한다. 만약 같다면 멈춰야 하므로 1을 반환하고 끝까지 순회하는 동안 같은 값이 없었다면 0을 반환한다.