# 시스템 소프트웨어와 실습

## 프로젝트 보고서

컴퓨터공학과

2010111661

임준수

담당교수 : 문봉교 교수님

제출일 : 15-11-06

Contents.

## 지원하는 명령어 설명

### Format 1 : SIO, TIO, HIO, FIX, FLOAT, NORM

패스 1에서 명령어 마지막 'O' 를 가지는 명령어는 format1 명령어 밖에 없고 실수형 명령어 FIX, FLOAT, NORM은 OR조건으로 구분한다. Format 1 명령어는 패스1에서 location counter를 구할 때 1byte 크기만큼 증가한다. Format 1은 1byte opcode로만 구성되어 있기 때문에 증가 크기는 1byte이다. Location counter를 구한 후 명령어의 format을 IMRArray[ArrayIndex]->formType=formatOne에 저장한다.

패스2에서는 opcode를 opcode table에서 검색한 후 명령어를 구분하기 위해서 패스1에 동일한 조건을 사용한다. opcode에 해당하는 연산코드를 inst_fmt_opcode에 저장한다. inst_fmt_opcode에 저장된 연산코드를 IMRArray[loop]->ObjectCode에 저장한다. Format 1 명령어는 1byte로 표현되기 때문에 플래그와 TA가 존재하지 않는다. 그러므로 object code는 연산코드로만 구성된다.

### Format 2 : ADDR, CLEAR, COMPR, DIVR, MULR, RMO, SHIFTL, SHIFTR, SUBR, SVC, TIXR

Fotmat2 명령어를 구분하는 방법은 명령어 마지막 단어가 R로 끝나거나 RMO, SHIFTL, SVC인 경우 format2 명령어로 구분한다. Format 2 명령어의 형식은 2byte로 표현되므로 패스 1에서 location counter를 구할 때 2byte만큼 증가한다. 2byte가 증가된 후에 intermediate file에서 명령어의 형식을 구분하기 위해서 IMRArray[ArrayIndex]->formType=formatTwo 코드를 실행한다. formatTwo는 2를 가지는 enum상수이다.

패스2에서 format2 명령어를 구분하는 방법은 패스 1에서 구분하는 방법과 동일하다. Format2 명령어 처리 구문이 실행되면 opcode를 opcode table에서 검색한 후 opcode의 연산코드를 inst_fmt_opcode에 저장한다. Format2명령어는 2byte, 16bit로 표현되는데 opcode는 8bit를 차지한다. Opcode를 연산자 위치로 이동하기 위해서 8bit를 왼쪽 시프트 연산자를 사용해서 이동시킨다. 왼쪽 시프트 연산자는 2^n으로 표현되는데 8bit만큼 왼쪽으로 이동했기 때문에 opcode의 연산코드*(2^8)으로 표현된다. 8bit만큼 왼쪽으로 이동한 명령어를 IMRArray[loop]->ObjectCode에 저장한다. Format2 명령어는 레지스터가 2개이거나 1개인 경우 2가지 경우가 존재하기 때문에 2개의 분기를 구성해야 된다. 첫 번째 분기는 레지스터가 2개인 경우로

```
if(isalpha(operand[0]) && operand[1]==',' && isalpha(operand[2]))
```

와 같이 표현된다. 피연산자 첫 번째 원소가 알파벳이고 두 번째 원소가 ',' 세 번째 원소가 알파벳이면 레지스터가 2개인 피연산자를 의미한다. 조건이 참이 되어 분기가 실행되면 피연산자에 저장된 레지스터를 register table에서 검색한다. REGTAB에서 레지스터를 검색하면 레지스터 번호를 읽어온다. 레지스터가 2개의 레지스터 중에서 첫 번째 레지스터이면 레지스터 번호에 16을 곱한다. 그 이유는 명령어가 목적코드로 변환되면 16bit에서 앞쪽 8bit는 명령어가 차지하고 뒤쪽

8bit가 레지스터 번호가 저장되는데 목적코드를 표현할 때 hex로 표현되기 때문에 8bit에서 앞쪽 4bit를 채우기 위해서는 16^1을 해야된다. 두 번째 레지스터는 8bit에서 뒤쪽 4bit를 차지하기 때문에 레지스터번호*(16^0)을 하게 된다. 피연산자에 레지스터가 1개인 경우에는

```
else if(isalpha(operand[0]) && operand[1]=='₩0')
```

로 표현된다. 분기가 참이 되어 실행되면 레지스터를 REGTAB에서 검색한 후 검색 결과로 나오는 레지스터 번호와 16을 곱한다. 곱한 결과를 inst_fmt_adderss에 저장한다. 16을 곱하는 이유는 레지스터가 1개인 경우 8bit에서 앞쪽 4bit에 레지스터 번호가 저장되고 뒤쪽 4bit에는 0이 채워지기 때문이다. 분기가 끝나면

```
inst_fmt = inst_fmt_opcode+inst_fmt_address;
IMRArray[loop]->ObjectCode = inst_fmt;
```

위에 코드가 실행된다. Format2는 목적코드를 만들 때 opcode와 레지스터 번호를 가지는 address만 더하면 된다. 그러므로 format2 명령어는 패스1과 패스2에서 모두 처리된다.

## Format 3 : ADD, AND, COMP, DIV, J, JEQ, JGT, JLT, JSUB, LDA, LDB, LDCH, LDL, LDS, LDT, LDX, LPS, MUL, OR, RD, RSUB, SSK, STA, STB, STCH, STI, STL, STS, STSW, STT, STX, SUB, TD, TIX, WD

Format 3 명령어는 24bit로 표현된다. 패스 1에서 format3 명령어의 location counter를 구할 때는 LOCCTR[LocctrCounter] = loc + (int)(OPTAB[Counter].Format-'0') 코드를 사용한다. Opcode를 버퍼로부터 읽어온 다음에 OPTAB에서 검색을 하고 Counter를 얻는다. 얻은 Counter를 이용해서 OPTAB에 Format 멤버변수에 접근해서 '3'을 가져온다. '3'는 아스키코드이므로 '0'을 빼서 정수형 3으로 변환시키고 LOCCTR에 저장한다.

패스2에서 format3명령어의 분기 조건은 else if(IMRArray[loop]->formType==formatThree)인데 패스1에서 format3 명령어의 LOCCTR을 구하면서 중간파일에 format type을 저장했다. 그러므로 패스2에서 분기 조건이 실행되면 조건이 참이 되어서 분기가 실행된다. 분기 내부에서는 OPTAB에 있는 연산자 연산코드를 가져온 다음 3h를 더한다. SICXE 명령어는 n, I flag가 1이므로 3h를 더해서 opcode를 구한다. Opcode를 구한 다음 24bit에서 opcode가 위치하는 시작 8bit에 opcode를 저장하기 위해서 왼쪽 시프트 연산자를 사용한다. 시프트 연산자를 사용해서 bit를 16bit만큼 이동한 다음에 opcode를 중간파일 objectCode에 저장한다. Opcode 연산이 끝나면 중간파일에서 operand를 읽어온다. Operand를 읽어온 다음 if(strcmp(opcode, "RSUB")) 명령어가 실행되는데 opcode가 RSUB 아닐 때 플래그와 변위가 설정된다. 명령어가 RSUB이면 플래그와 변위를 설정하지 않고 opcode가 목적코드가 된다. RSUB가 아니어서 분기가 실행되면 indexed addressing을 판별하는 조건분기가 나온다.

```
if (operand[strlen(operand)-2] == ',' && (operand[strlen(operand)-1] == 'X' || operand[strlen(operand)-1] == 'x'))
```

피연산자에 ,X 가 존재하면 indexed addressing 이므로 위에 조건이 참이 되고 플래그를 A로 설정한다. 플래그 A는 2(pc-relative)+8(indexed)를 의미한다. 플래그를 설정한 다음 ,X를 지운다. ,X가 존재하지 않으면 플래그는 pc-relative를 의미하는 2가 된다. 플래그를 설정한 다음 symbol table에서 operand를 검색한다. Symbol table에서 operand를 검색해서 일치하는 레이블을 찾으면 레이블의 주소와 PC가 가리키는 주소를 빼서 relative-address를 구한다.

inst_fmt_address = (long)SYMTAB[search_symtab].Address - LOCCTR[loop];//pc-relative

위에 명령어는 relative-address를 구하는 법을 보여준다. Address를 구한 다음

inst_fmt = inst_fmt_opcode + inst_fmt_index + inst_fmt_address;
IMRArray[loop]->ObjectCode = inst_fmt;

코드가 실행되면서 목적코드가 생성된다. Opcode와 플래그를 나타내는 index, address를 더하면 format3 목적코드가 된다. 생성된 목적코드를 중간파일 ObjectCode에 저장한다.

## Format 4 :

Format 4 명령어는 format 3 opcode에 +가 붙어있는 형태이다. Format4 명령어가 되면 기존 24bit에서 32bit으로 확장된다. 패스1에서 format4 명령어를 구분하는 방법은 if(opcode[0]=='+') 인 경우를 format4 명령어로 구분했다. 해당 조건이 참이 되면

```
for(i=0; i<strlen(opcode)-1; i++)
        opcode[i]=opcode[i+1];
opcode[strlen(opcode)-1]='₩0';
```

opcode 앞에 +를 제거하는 반복문이 실행된다. 위에 코드 동작 방법은 opcode[0]의 '+'을 뒤에 원소들을 앞으로 한 칸씩 이동시켜서 덮어씌우는 방식이다. 최종적으로 반복문이 종료되면 '+'는 삭제된다. '+'가 삭제되면 opcode가 OPTAB에 존재하는지 검색한다. 검색 후 location counter를 4byte만큼 증가시킨다. Location counter를 4byte 증가시킨 후에 IMRArray[ArrayIndex]->formType=formatFour; 코드가 실행되면서 format 4 형식인 것을 저장한다.

패스2에서 format4 형식을 구분하는 분기 조건은

if(opcode[0]=='+' && IMRArray[loop]->formType==formatFour) 와 같다. Opcode 앞에 '+' 가 존재하고 form type이 format 4이면 format 4를 구분하는 조건은 참이 된다. 조건이 참이 되어서 분기 내부가 실행되면 패스1에서 '+' 를 제거하기 위해 사용한 for반복문을 사용한다. 반복문을 사용해서 '+' 을 제거한 다음 OPTAB에서 opcode를 검색한다. Opcode가 검색되면 Counter에 OPTAB에서 opcode의 위치가 저장되고 OPTAB에서 연산코드를 가져온다. 가져온 연산코드와 3h를 더해서 SICXE 명령어 opcode로 변환한다. Format4는 32bit로 표현되므로 opcode가 위치하는 앞에 8bit로 이동하기 위해서 <<=24를 한다. 왼쪽 시프트 연산을 24bit만 하면 opcode는 본래에 위치로 이동한다. Opcode를 시프트 연산한 다음 중간파일 ObjectCode에 저장한다. 저장 후에 피연산자를 읽어와서 indexed addressing인지 확인한다. 아래에 조건은 indexed address을 확인하기 위한 조건이다.

```
if (operand[strlen(operand)-2] == ',' && (operand[strlen(operand)-1] == 'X' || operand[strlen(operand)-1] == 'x'))
```

피연산자에 ,X 가 존재하면 플래그를 9로 설정한다. 플래그 9는 1(extended)+8(indexed)를 의미한다. 플래그를 설정한 다음 ,X를 지운다. ,X가 존재하지 않으면 플래그는 extended를 의미하는 1이 된다. 플래그를 설정한 다음 symbol table에서 operand를 검색한다. Symbol table에서 operand를 검색해서 일치하는 레이블을 찾으면 레이블의 주소를 inst_fmt_address에 저장한다. Format4형식은 relative-addressing이 아닌 absolute-addressing이므로 레이블의 주소가 target address가 된다. 그 다음 아래 명령어가 실행되면서 목적코드를 생성한다.

```
inst_fmt =  inst_fmt_opcode + inst_fmt_index + inst_fmt_address;
IMRArray[loop]->ObjectCode = inst_fmt;
```

## Floating point : LDF, COMPF, ADDF, DIVF, MULF, SUBF, STF, FIX, FLOAT, NORM

Floating point instruction의 핵심은 실수를 저장하는 것이다. 패스 2에서 WORD에 저장된 피연산자를 실수로 변환한 다음 중간파일에 저장한다.

```
else if (!strcmp(opcode, "WORD")){
                        strcpy(operand,IMRArray[loop]->OperandField);
                        if(strchr(operand,'.'))
                        {
                                IMRArray[loop]->FloatNum=atof(operand);
                                IMRArray[loop]->FloatFlag=1;
                        }
                        else
                        {
                                IMRArray[loop]->ObjectCode = StrToDec(operand);
                        }
                }
```

Opcode가 WORD이면 피연산자를 읽어와서 operand변수에 임시 저장한다. 저장된 operand를 strchr을 사용해서 실수인지 판별한다. Strchr은 2번째 매개변수로 전달되는 문자가 delimiter가 되므로 피연산자 내부에 ' . ' 이 존재하면 문자형 실수가 된다. 참인 분기가 실행되면 operand를 atof를 사용해서 문자형 실수에서 정수형 실수로 변환한 다음 중간파일 FloatNum에 저장한다. 그런 다음 FloatFlag를 1로 설정한다. 만약 실수가 아니고 정수이면 피연산자를 정수로 변환한 다음 중간파일 ObjectCode에 정수를 저장한다. FloatFlag를 1로 설정하는 이유는 object 파일과 list 파일을 만들 때 기존 %x로 실수가 표현되지 않기 때문에 분기를 줘서 실수 표현 코드를 실행하기 위해서이다. Object, list 파일에서 실수를 파일에 출력하기 위한 코드는 아래와 같다.

```
_float.f = temp_float[xx];
for(i=0; i<4; i++){
    printf("%02x", _float.c[i]);
    fprintf(fptr_obj, "%02x", _float.c[i]);
}
```

해당 코드가 동작하는 방법은 다음과 같다. _float는 union인데 union에 멤버변수 float f에 실수를 저장한다. Union은 멤버변수에서 가장 큰 자료형이 union의 크기가 되므로 현재 공용체의 크기는 4byte가 된다. 그런 다음 실수를 파일에 출력하기 위해서 for문이 실행되는데 _float 내부에 char c[4] 배열을 이용해서 실수를 출력한다. _float.c[i]는 1byte를 나타내므로 4byte 실수에서 1byte를 읽어와서 %02x로 파일에 출력한다. 실수는 4byte이므로 반복문은 4번 반복되고 파일에는 16진수 8자리로 실수가 출력된다. 그러므로 실수가 파일에 16진수 형식으로 저장된다.

실수 연산을 지원하는 명령어는 기존 format3, format1 명령어의 동작방식과 같다. FIX, FLOAT, NORM은 format1 명령어로 처리되고 ADDF, COMPF, DIVF, LDF, MULF, STF, SUBF 명령어는 기본 format3로 처리되고 opcode에 '+'가 붙으면 format4 형식으로 처리된다.

실행 캡쳐는 16p부터 18p까지 존재한다.

전체 소스프로그램은 제출한 압축파일에 존재한다. (소스코드 이름 : Xe_Assembler.c)

## Algorithms of Assembler Pass 1 and Pass2.

```
void main (void)
{
	//변수 선언
	FILE* fptr;

	char filename[15];
	char label[32];
	char opcode[32];
	char operand[32];

	int loc = 0;
	int line = 0;
	int loop;
	int is_empty_line;
	int is_comment;
	int loader_flag = 0;
	int search_symtab;
	int search_regtab;
	int i, x;

	unsigned long inst_fmt;
	unsigned long inst_fmt_opcode;
	unsigned long inst_fmt_index;
	unsigned long inst_fmt_address;

	printf(" ********************************************************************************₩n");
	printf(" * Program: SICXE ASSEMBRER                                                    *₩n");
	printf(" *                                                                             *₩n");
	printf(" * Procedure:                                                                  *₩n");
	printf(" *   - Enter file name of source code.                                         *₩n");
	printf(" *   - Do pass 1 process.                                                      *₩n");
	printf(" *   - Do pass 2 process.                                                      *₩n");
	printf(" *   - Create ₩"program list₩" data on sic.list.(Use Notepad to read this file) *₩n");
	printf(" *   - Create ₩"object code₩" data on sic.obj.(Use Notepad to read this file)   *₩n");
	printf(" *   - Also output object code to standard output device.                      *₩n");
	printf(" ********************************************************************************₩n");


	printf("₩nEnter the file name you want to assembly (sic.asm):");
	scanf("%s",filename);
```

```
        fptr = fopen(filename,"r");//파일 열기

        //예외처리
        if (fptr == NULL)
        {
                printf("ERROE: Unable to open the %s file.₩n",filename);
                exit(1);
        }

/********************************* PASS 1 **********************************/
        printf("Pass 1 Processing...₩n₩n");
        while (fgets(Buffer,256,fptr) != NULL)
        {
                is_empty_line = strlen(Buffer);

                Index = 0;
                j = 0;
                //버퍼로부터 레이블을 읽어와 변수에 저장
                strcpy(label,ReadLabel());

                //명령어인지 주석인지 판단하는 구문
      if (Label[0] == '.')
                        is_comment = 1;
                else
                        is_comment = 0;

                //읽어온 명령어의 길이가 1이상 이면서 주석이 아닐 때 실행되는 구문
                if (is_empty_line>1 && is_comment!=1)
                {
                        Index = 0;
                        j = 0;

                        //intermediate structure 힙에 동적 할당
                        IMRArray[ArrayIndex] = (IntermediateRec*)malloc(sizeof(IntermediateRec));

                        //버퍼로부터 레이블을 읽어와 중간파일에 저장
                        IMRArray[ArrayIndex]->LineIndex = ArrayIndex;
                        strcpy(label,ReadLabel());
                        strcpy(IMRArray[ArrayIndex]->LabelField,label);
                        SkipSpace();

                        //버퍼로부터 연산자를 읽어온 다음 중간파일에 저장
                        //연산자가 START이면 피연산자를 읽어온 다음 중간파일에 저장
                        //피연산자를 16진수로 변환한 다음 location counter에 저장
                        //16진수로 변환된 수는 프로그램의 시작주소가 됨.
                        if (line == 0)
                        {
                                strcpy(opcode,ReadOprator());
                                strcpy(IMRArray[ArrayIndex]->OperatorField,opcode);
                                if (!strcmp(opcode,"START"))
                                {
                                        SkipSpace();//연산자와 피연산자 사이의 공백 제거
                                        strcpy(operand,ReadOperand());
                                        strcpy(IMRArray[ArrayIndex]->OperandField, operand);/* [A] */
                                        LOCCTR[LocctrCounter] = StrToHex(operand);
                                        start_address = LOCCTR[LocctrCounter];
                                }
                                //START연산자가 없으면 프로그램의 시작주소는 0
                                else
                                {
                                        LOCCTR[LocctrCounter] = 0;
```

```
                        start_address = LOCCTR[LocctrCounter];
                }
        }
        //START연산자가 포함된 명령어를 제외한 모든 명령어 처리 구문
        //연산자와 피연산자를 버퍼로부터 읽어온 다음 중간파일에 저장
        else
        {
                strcpy(opcode,ReadOprator());
                strcpy(IMRArray[ArrayIndex]->OperatorField,opcode);
                SkipSpace();
                strcpy(operand,ReadOperand());
                strcpy(IMRArray[ArrayIndex]->OperandField,operand);

                //연산자가 END가 아닐 때 실행
                if (strcmp(opcode,"END"))
                {
                        if (label[0] != '₩0')
                        {
                                //레이블이 symbol table에 존재하는지 확인
                                //레이블이 존재하면 중복 심볼이므로 에러 출력 후
파일 종료

                                if (SearchSymtab(label))
                                {
                                        fclose(fptr);
                                        printf("ERROE: Duplicate Symbol₩n");
                                        FoundOnSymtab_flag = 0;
                                        exit(1);
                                }
                                //symbol table에 레이블 저장
                                RecordSymtab(label);
                        }

                        //format4일 때 실행되는 구문
                        if (SearchOptab (opcode) || opcode[0]=='+')
                        {
                                if(opcode[0]=='+')
                                {
                                        //연산자 앞에 붙은 +를 제거한 형태로
만드는 반복문

                                        //배열의 0번지에 저장된 +를 뒤에 저장된
데이터를 앞으로 한칸씩 이동해서 +삭제

                                        for(i=0; i<strlen(opcode)-1; i++)
                                                opcode[i]=opcode[i+1];
                                        opcode[strlen(opcode)-1]='₩0';

                                        //연산자가 OPTAB에 존재하는지 확인
                                        //format4이므로 길이는 4byte 증가
                                        //중간파일 formType에 format4인 것 저장
                                        SearchOptab(opcode);
                                        LOCCTR[LocctrCounter] = loc + 4;
                                        IMRArray[ArrayIndex]->formType=formatFour;
                                }
                                //연산자 마지막 철자에 R이 붙어있으면서 OR이 아닐 때
실행

                                //OR은 format3이므로 해당 구문에서 실행되면 안됨.
                                //연산자 마지막 철자에 R이 붙어있으면 레지스터 연산.
                                //레지스터 연산은 format2이므로 길이 2만큼 증가
                                //중간파일 formType에 format2인 것 저장
```

```
                                        else if(opcode[strlen(opcode)-1]=='R' &&
strcmp(opcode, "OR"))
                                        {
                                                LOCCTR[LocctrCounter] = loc + 2;
                                                IMRArray[ArrayIndex]->formType=formatTwo;
                                        }
                                        //연산자 마지막 철자에 R이 안 붙은 format2 명령어
처리 구문
                                        //중간파일 formType에 format2인 것 저장
                                        else if(!strcmp(opcode, "SHIFTL") || !strcmp(opcode,
"RMO") || !strcmp(opcode, "SVC"))
                                        {
                                                LOCCTR[LocctrCounter] = loc + 2;
                                                IMRArray[ArrayIndex]->formType=formatTwo;
                                        }
                                        //format1 연산자 처리 구문
                                        //연산자 마지막 철자가 O이거나 FIX, FLOAT, NORM일 때
실행
                                        //중간파일 formType에 format1인 것 저장
                                        //format1이므로 길이 1만큼 증가
                                        else if(opcode[strlen(opcode)-1]=='O'
|| !strcmp(opcode, "FIX") || !strcmp(opcode, "FLOAT") || !strcmp(opcode, "NORM"))
                                        {
                                                LOCCTR[LocctrCounter] = loc + 1;
                                                IMRArray[ArrayIndex]->formType=formatOne;
                                        }
                                        //format3 처리 구문
                                        //OPTAB에 저장되어있는 format을 읽어와 정수로 변환한
다음 location counter에 저장
                                        //중간파일 formType에 format3인 것 저장
                                        else
                                        {
                                                LOCCTR[LocctrCounter] = loc +
(int)(OPTAB[Counter].Format-'0') ;
                                                IMRArray[ArrayIndex]->formType=formatThree;
                                        }
                                }
                                //실행 연산자가 아닌 메모리 연산자인 경우 실행
                                //WORD는 SICXE에서 3byte로 표현되므로 길이 3 증가
                                //RESW는 피연산자에 저장된 수*3만큼 길이 증가
                                //RESB는 피연산자에 저장된 수*1만큼 길이 증가
                                //BYTE 연산자의 피연산자는 C' 또는 X'와 같이 쓰이므로 C'---',
X'--' 사이의 길이 측정 후 location counter에 저장
                                else if (!strcmp(opcode,"WORD"))
                                        LOCCTR[LocctrCounter] = loc + 3;
                                else if (!strcmp(opcode,"RESW"))
                                        LOCCTR[LocctrCounter] = loc + 3 * StrToDec(operand);
                                else if (!strcmp(opcode,"RESB"))
                                        LOCCTR[LocctrCounter] = loc + StrToDec(operand);
                                else if (!strcmp(opcode,"BYTE"))
                                        LOCCTR[LocctrCounter] = loc + ComputeLen(operand);
                                else{//예외처리
                                        fclose(fptr);
                                        printf("ERROE: Invalid Operation Code₩n");
                                        exit(1);
                                }
                        }
                }
                //START를 제외한 명령어 길이를 중간파일에 저장하기 위해서 LocctrCounter-1을 함.
```

```
                        loc = LOCCTR[LocctrCounter];
                        IMRArray[ArrayIndex]->Loc = LOCCTR[LocctrCounter-1];
                        LocctrCounter++;
                        ArrayIndex++;
                }
                FoundOnOptab_flag = 0;
                line += 1;
        }
        //프로그램 전체 길이 계산
        //반복문이 종료될 때 LocctrCounter가 1증가하면서 종료되었으므로 마지막 명령어의 위치는 LocctrCounter-
2를 해야 됨.
        program_length = LOCCTR[LocctrCounter-2]- LOCCTR[0];

        //symbol table 출력
        //레이블과 주소를 출력한다.
        printf("SYMBOL TABLE₩n");
        printf("-----------------₩n");
        printf("LABEL₩tADDRESS₩n");
        printf("-----------------₩n");
        for(loop=0; loop<ArrayIndex; loop++)   //symbol table print!!
        {
                if(isalpha(SYMTAB[loop].Label[0]))
                        printf("%s₩t%x₩n",SYMTAB[loop].Label,SYMTAB[loop].Address);
        }

/********************************* PASS 2 **********************************/
        printf("₩nPass 2 Processing...₩n");

        //intermediate stucture array에 길이만큼 반복
        for (loop = 1; loop<ArrayIndex; loop++){
                inst_fmt_opcode = 0;
                inst_fmt_index = 0;
                inst_fmt_address = 0;
                IMRArray[loop]->FloatFlag = 0;

                strcpy(opcode,IMRArray[loop]->OperatorField);

                //연산자가 OPTAB에 존재하면 실행
                if (SearchOptab(opcode) || opcode[0]=='+'){
                        //format4인 경우 실행
                        if(opcode[0]=='+' && IMRArray[loop]->formType==formatFour)
                        {
                                //연산자 앞에 붙은 +를 제거한 형태로 만드는 반복문
                                //배열의 0번지에 저장된 +를 뒤에 저장된 데이터를 앞으로 한칸씩 이동해서
+삭제
                                for(i=0; i<strlen(opcode)-1; i++)
                                        opcode[i]=opcode[i+1];
                                opcode[strlen(opcode)-1]='₩0';

                                //연산자를 OPTAB에서 검색한 후 연산자의 머신코드와 3h를 더한 다음
임시변수에 저장
                                //3h를 더하는 이유는 SICXE명령어의 opcode 8bit에서 n , i flag가 항상
1이므로 3을 더한다.
                                //임시변수에 저장된 opcode를 좌측으로 24bit만큼 이동한다.
                                //임시변수에 저장된 opcode를 중간파일에 저장한다.
                                //피연산자를 중간파일에서 읽어온 다음 operand변수에 저장
                                //피연산자가 operand,X 형태이면 indexed addressing이므로 플래그의 값은
9가 됨
```

```
                                        //operand,X 형태가 아니면 플래그의 값은 1이 됨
                                        //피연산자를 SYMTAB에서 검색한 다음 SYMTAB에 저장된 주소를
inst_fmt_address에 저장
                                        //inst_fmt_opcode + inst_fmt_index + inst_fmt_address를 더하면
목적코드가 된다.
                                        SearchOptab(opcode);
                                        inst_fmt_opcode = OPTAB[Counter].ManchineCode+0x03;
                                        inst_fmt_opcode <<=24;
                                        IMRArray[loop]->ObjectCode = inst_fmt_opcode;
                                        strcpy(operand,IMRArray[loop]->OperandField);

                                        if (operand[strlen(operand)-2] == ',' && (operand[strlen(operand)-1] ==
'X' || operand[strlen(operand)-1] == 'x')){
                                        inst_fmt_index = 0x00900000;
                                        operand[strlen(operand)-2] = '₩0';
                                        }
                                        else
                                                inst_fmt_index = 0x00100000;

                                        for(search_symtab = 0; search_symtab<SymtabCounter; search_symtab++){
                                                if(!strcmp(operand, SYMTAB[search_symtab].Label))
                                                inst_fmt_address = (long)SYMTAB[search_symtab].Address ;
                                        }
                                        inst_fmt =  inst_fmt_opcode + inst_fmt_index + inst_fmt_address;
                                        IMRArray[loop]->ObjectCode = inst_fmt;
                                }
                                //format2인 경우 실행
                                //format2는 16bit이므로 opcode를 8bit 좌측으로 이동
                                //레지스터 연산은 피연산자가 2개이거나 1개인 경우가 존재한다.
                                //피연산자로 쓰인 레지스터를 REGTAB에서 검색한다.
                                //피연산자 레지스터가 2개인 경우 첫 번째 레지스터는 레지스터의 값에다가 16을
곱한다.
                                //두 번째 레지스터는 레지스터의 값에다가 1을 곱한다. 2개의 값을 더한 후 address에
저장
                                //16과 1을 곱하는 이유는 operand field가 1byte인데 첫 번째 레지스터는 앞쪽 4bit,
두 번째 레지스터는 뒤쪽 4bit에 저장되기 때문이다.
                                //피연산자가 1개인 경우 REGTAB에서 레지스터를 검색한 후 레지스터의 값과 16을 곱한
다음 address에 저장
                                //16을 곱하는 이유는 레지스터가 1개일 때 1byte에서 상위 4bit에 값이 저장되고 하위
4bit는 0으로 채워지기 때문이다.
                                //inst_fmt_opcode+inst_fmt_address의 값이 목적코드가 된다. format2는 플래그가
없으므로 inst_fmt_index는 없다.
                                else if(opcode[strlen(opcode)-1]=='R'&& IMRArray[loop]->formType==formatTwo)
                                {
                                        inst_fmt_opcode = OPTAB[Counter].ManchineCode;
                                        inst_fmt_opcode <<=8;
                                        IMRArray[loop]->ObjectCode = inst_fmt_opcode;
                                    strcpy(operand,IMRArray[loop]->OperandField);

                                        if(isalpha(operand[0]) && operand[1]==',' && isalpha(operand[2]))
                                        {
                                                for(search_regtab=0; search_regtab<7; search_regtab++)
                                                {
                                                        if(operand[0]==REGTAB[search_regtab].regName)
                                                                inst_fmt_address +=
REGTAB[search_regtab].regNum*16;
                                                        if(operand[2]==REGTAB[search_regtab].regName)
                                                                inst_fmt_address +=
```

```
REGTAB[search_regtab].regNum*1;
                                        }
                                }
                                else if(isalpha(operand[0]) && operand[1]=='WO')
                                {
                                        for(search_regtab=0; search_regtab<7; search_regtab++)
                                        {
                                                if(operand[0]==REGTAB[search_regtab].regName)
                                                        inst_fmt_address +=
REGTAB[search_regtab].regNum*16;
                                        }

                                }
                                inst_fmt = inst_fmt_opcode+inst_fmt_address;
                                IMRArray[loop]->ObjectCode = inst_fmt;
                        }
                        //format2인 경우 실행
                        //format2는 16bit이므로 opcode를 8bit 좌측으로 이동
                        //SHIFTL, RMO, SVC 연산은 피연산자가 2개이거나 1개인 경우가 존재한다.
                        //피연산자로 쓰인 레지스터를 REGTAB에서 검색한다.
                        //피연산자 레지스터가 2개인 경우 첫 번째 레지스터는 레지스터의 값에다가 16을
곱한다.
                        //두 번째 레지스터는 레지스터의 값에다가 1을 곱한다. 2개의 값을 더한 후 address에
저장
                        //16과 1을 곱하는 이유는 operand field가 1byte인데 첫 번째 레지스터는 앞쪽 4bit,
두 번째 레지스터는 뒤쪽 4bit에 저장되기 때문이다.
                        //피연산자가 1개인 경우 REGTAB에서 레지스터를 검색한 후 레지스터의 값과 16을 곱한
다음 address에 저장
                        //16을 곱하는 이유는 레지스터가 1개일 때 1byte에서 상위 4bit에 값이 저장되고 하위
4bit는 0으로 채워지기 때문이다.
                        //inst_fmt_opcode+inst_fmt_address의 값이 목적코드가 된다. format2는 플래그가
없으므로 inst_fmt_index는 없다.
                        else if((!strcmp(opcode, "SHIFTL") || !strcmp(opcode, "RMO") || !strcmp(opcode,
"SVC")) && IMRArray[loop]->formType==formatTwo)
                        {
                                inst_fmt_opcode = OPTAB[Counter].ManchineCode;
                                inst_fmt_opcode <<=8;
                                IMRArray[loop]->ObjectCode = inst_fmt_opcode;
                            strcpy(operand,IMRArray[loop]->OperandField);

                                if(isalpha(operand[0]) && operand[1]==',' && isalpha(operand[2]))
                                {
                                        for(search_regtab=0; search_regtab<7; search_regtab++)
                                        {
                                                if(operand[0]==REGTAB[search_regtab].regName)
                                                        inst_fmt_address +=
REGTAB[search_regtab].regNum*16;

                                                if(operand[2]==REGTAB[search_regtab].regName)
                                                        inst_fmt_address +=
REGTAB[search_regtab].regNum*1;
                                        }
                                }
                                else if(isalpha(operand[0]) && operand[1]=='WO')
                                {
                                        for(search_regtab=0; search_regtab<7; search_regtab++)
                                        {
                                                if(operand[0]==REGTAB[search_regtab].regName)
                                                        inst_fmt_address +=
REGTAB[search_regtab].regNum*16;
                                        }
```

```
                                        }
                                        inst_fmt = inst_fmt_opcode+inst_fmt_address;
                                        IMRArray[loop]->ObjectCode = inst_fmt;
                                }
                                //format1인 경우 실행
                                //format1은 1byte이므로 OPTAB에 machine code가 목적코드가 된다.
                               else if((opcode[strlen(opcode)-1]=='0' || !strcmp(opcode, "FIX") || !strcmp(opcode,
"FLOAT") || !strcmp(opcode, "NORM"))
                                        && IMRArray[loop]->formType==formatOne)
                                {
                                        inst_fmt_opcode = OPTAB[Counter].ManchineCode;
                                        IMRArray[loop]->ObjectCode = inst_fmt_opcode;

                                        inst_fmt=inst_fmt_opcode;
                                        IMRArray[loop]->ObjectCode=inst_fmt;
                                }
                                //format3인 경우 실행
                                //opcode는 SICXE이므로 연산코드+3h를 해서 구한다.
                                //foramt3는 24bit를 사용해서 명령어를 표현하므로 opcode는 16bit 좌측으로 이동한다.
                               //opcode가 RSUB인 경우에는 플래그 비트와 TA가 필요없다. 연산코드가 목적코드가 된다.
                                //format3는 relative-addressing으로 구동되므로 해당 프로그램에서는 pc-relative
addressing을 사용했다.
                                //TA와 pc가 가리키는 주소를 빼서 relative-address를 구했다.
                                //operand,X인 경우 indexed addressing이므로 플래그는 8+2=A가 된다.
                                //operand,X가 아닌 경우 플래그는 2가 된다.
                                //inst_fmt_opcode+inst_fmt_index+inst_fmt_address의 값이 목적코드가 된다.
                                else if(IMRArray[loop]->formType==formatThree)
                                {
                                        inst_fmt_opcode = OPTAB[Counter].ManchineCode+0x03;
                                        inst_fmt_opcode <<=16;
                                        IMRArray[loop]->ObjectCode = inst_fmt_opcode;
                                        strcpy(operand,IMRArray[loop]->OperandField);

                                        if(strcmp(opcode, "RSUB"))
                                        {
                                                if (operand[strlen(operand)-2] == ',' &&
(operand[strlen(operand)-1] == 'X' || operand[strlen(operand)-1] == 'x')){
                                                        inst_fmt_index = 0x00A000;
                                                        operand[strlen(operand)-2] = '₩0';
                                                }
                                                else
                                                        inst_fmt_index = 0x002000;


                                                for(search_symtab = 0; search_symtab<SymtabCounter;
search_symtab++){
                                                        if(!strcmp(operand, SYMTAB[search_symtab].Label))
                                                                //PC-relative addressing.
                                                        inst_fmt_address =
(long)SYMTAB[search_symtab].Address - LOCCTR[loop];
                                                }
                                        }
                                        inst_fmt =  inst_fmt_opcode + inst_fmt_index + inst_fmt_address;
                                        IMRArray[loop]->ObjectCode = inst_fmt;
                                }

                        }
                        //연산자가 WORD인 경우 실행
                        //피연산자를 중간파일에서 가져온 다음 피연산자 내부에 '.'이 있는지 확인
```

```
                    //'.'이 존재하면 피연산자의 값은 실수이다.
                    //strchr로 실수를 판별한 후 실수이면 피연산자를 atof로 실수로 변환한 다음 중간파일
FloatNum에 저장
                    //실수이면 중간파일 FloatFlag는 1이 됨
                    //실수가 아니면 피연산자를 정수로 변환한 다음 중간파일 ObjectCode에 저장
                    else if (!strcmp(opcode, "WORD")){
                              strcpy(operand,IMRArray[loop]->OperandField);
                              if(strchr(operand,'.'))
                              {
                                        IMRArray[loop]->FloatNum=atof(operand);
                                        IMRArray[loop]->FloatFlag=1;
                              }
                              else
                              {
                                        IMRArray[loop]->ObjectCode = StrToDec(operand);//float 수정할 곳
                              }
                    }
                    //연산자가 BYTE인 경우 실행
                    //BYTE C'---' 형태인 경우 for문을 사용해서 C'---' 내부의 값을 int형으로 변환한 다음
ObjectCode에 저장
                    //ObjectCode에 저장된 값을 좌측으로 8bit 이동한다. 이동하는 이유는 C'를 없애기 위해서
                    //BYTE X'--' 형태인 경우 X'--' 내부의 값을 hex로 변환한 다음 ObjectCode에 저장한다.
                    //ObjectCode에 저장된 값을 좌측으로 8bit 이동한다. 이동하는 이유는 X'를 없애기 위해서
                    else if (!strcmp(opcode,"BYTE")){
                              strcpy(operand,IMRArray[loop]->OperandField);
          IMRArray[loop]->ObjectCode = 0;

                              if(operand[0]=='C' || operand[0]=='c' && operand[1]=='₩''){
                                        for (x = 2; x<=(int)(strlen(operand)-2); x++){
                                                  IMRArray[loop]->ObjectCode=IMRArray[loop]->ObjectCode +
(int)operand[x];
                                                  IMRArray[loop]->ObjectCode<<=8;
                                        }
                              }

          if(operand[0]=='X' || operand[0]=='x' && operand[1]=='₩''){
                                        char *operand_ptr;
                                        operand_ptr = &operand[2];//X' 다음에 값을 가리킴
                                        *(operand_ptr+2)='₩0';//X'--'에서 마지막 ' 를 지움
                                        for (x=2; x<=(int)(strlen(operand)-2); x++){
                                                  IMRArray[loop]->ObjectCode=IMRArray[loop]->ObjectCode +
StrToHex(operand_ptr);
                                                  IMRArray[loop]->ObjectCode<<=8;
                                        }
                              }
                              //좌측으로 8bit 이동한 것을 원래 상태로 복귀
                    IMRArray[loop]->ObjectCode>>=8;
                    }
          }


          CreateProgramList();//List file creation
          CreateObjectCode();//object file creation

          //intermediate structure free
          for (loop = 0; loop<ArrayIndex; loop++){
        free(IMRArray[loop]);
          }

          printf("Compeleted Assembly₩n");
```

```
        fclose(fptr);//파일 종료
}
```

## Symbol Table :

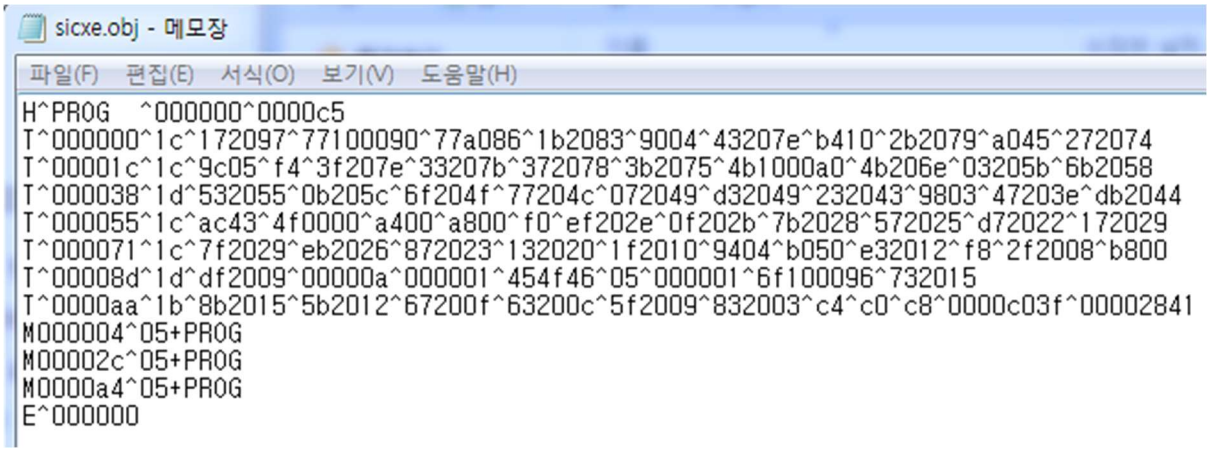
```
Enter the file name you want to assembly (sic.asm):project.asm
Pass 1 Processing...

SYMBOL TABLE
------------------
LABEL   ADDRESS
------------------
LABEL1  14
LABEL2  22
LABEL3  38
LABEL4  55
LABEL5  6e
LABEL6  82
REF1    90
STAT    93
EOF     96
DEVICE  99
RETADR  9a
BOX     9d
TEST    a0
FLOAT1  bf
FLOAT2  c2

Pass 2 Processing...
Creating Object Code...
```

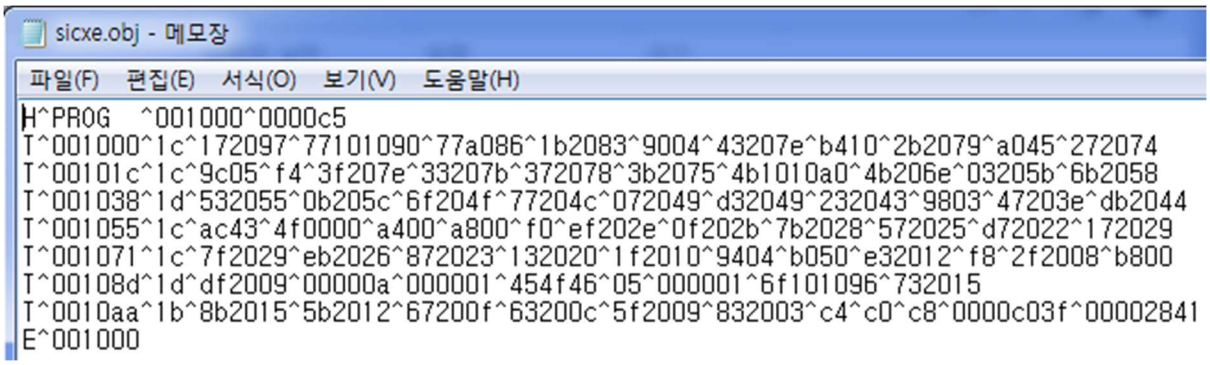
Symbol table은 패스 1에서 생성된다. 파일로부터 명령어를 읽어와서 버퍼에 저장한 다음 명령어를 파싱해서 레이블을 구한다. 레이블을 구한 후에 RecordSymtab 함수를 호출하면서 인자로 레이블을 전달한다. 전달된 레이블은 symbol table에 저장되고 레이블의 주소는 location counter를 저장하는 LOCCTR에서 읽어온 다음 symbol table에 저장한다. 그러므로 패스 1이 끝나면 위와 같은 symbol table를 얻을 수 있다.

## Before Program Relocation



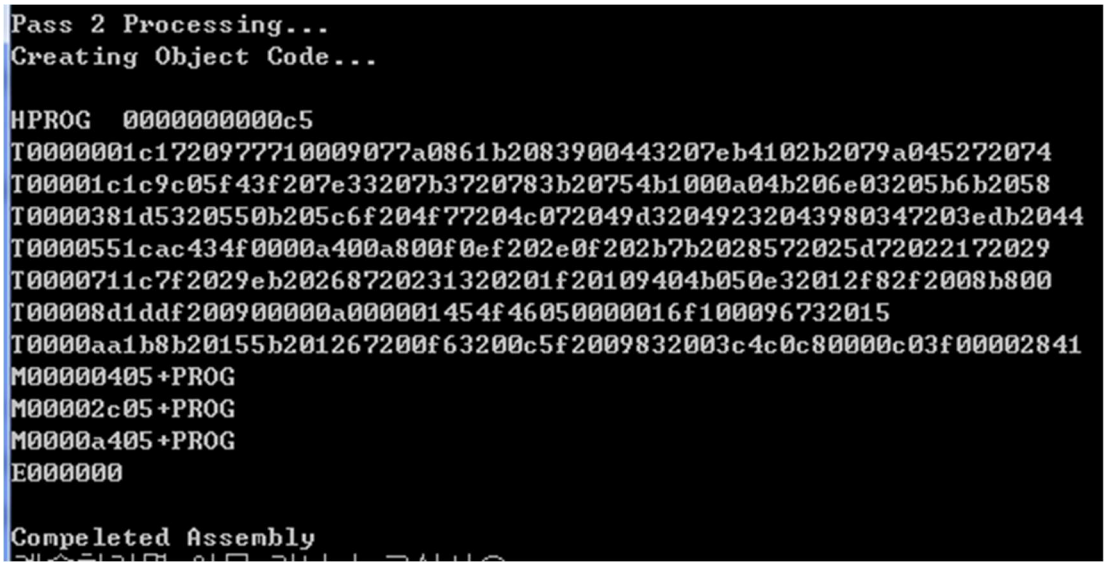
해당 실행 결과는 program relocation이 되지 않은 경우이다. START 명령어의 operand가 0이면 프로그램의 location counter는 상대주소를 가진다. Format3 명령어는 relative-addressing을 하므로 M레코드가 필요 없다. 그러나 format4 명령어는 absolute-addressing을 하므로 목적프로그램이 생성될 때 M레코드가 삽입되어야 한다. M레코드는 프로그램이 loader에 의해 loading될 때 수정해야 될 위치를 나타낸다.

## After Program Relocation




START의 operand가 load point를 가지는 경우이다. 프로그램이 load point를 가지는 상태에서 목적프로그램이 생성되면 프로그램은 loading시에 재배치가 필요 없다. 그러므로 위에 결과는 load point 1000을 가지므로 M레코드가 삽입되지 않는다.

## cmd에서 실행결과 화면 ( before program relocation)




어셈블러의 패스 1과 패스2가 끝난 후 콘솔화면에 목적프로그램이 출력된다. 해당 결과는 load point가 지정되지 않은 프로그램이므로 목적프로그램에는 M레코드가 삽입되어 있다.

## 생성된 LIST 파일

sicxe.list - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
LOC     LABEL       OPERATOR    OPERAND     OBJECT CODE
0000    PROG        START       0
0000                STL         RETADR      172097
0003                +LDT        REF1        77100090
0007                LDT         REF1,X      77a086
000a                ADD         REF1        1b2083
000d                ADDR        A,S         009004
000f                AND         REF1        43207e
0012                CLEAR       X           00b410
0014    LABEL1      COMP        REF1        2b2079
0017                COMPR       S,T         00a045
0019                DIV         REF1        272074
001c                DIVR        A,T         009c05
001e                HIO                     0000f4
001f                J           TEST        3f207e
0022    LABEL2      JEQ         TEST        33207b
0025                JGT         TEST        372078
0028                JLT         TEST        3b2075
002b                +JSUB       TEST        4b1000a0
002f                JSUB        TEST        4b206e
0032                LDA         REF1        03205b
0035                LDB         REF1        6b2058
0038    LABEL3      LDCH        REF1        532055
003b                LDL         RETADR      0b205c
003e                LDS         REF1        6f204f
0041                LDT         REF1        77204c
0044                LDX         REF1        072049
0047                LPS         STAT        d32049
004a                MUL         REF1        232043
004d                MULR        A,B         009803
004f                OR          REF1        47203e
0052                RD          DEVICE      db2044
0055    LABEL4      RMO         S,B         00ac43
0057                RSUB                    4f0000
005a                SHIFTL      REF1,A      00a400
005c                SHIFTR      REF1,A      00a800
005e                SIO                     0000f0
005f                SSK         REF1        ef202e
0062                STA         REF1        0f202b
0065                STB         REF1        7b2028
0068                STCH        REF1        572025
006b                STI         REF1        d72022
006e    LABEL5      STL         RETADR      172029
0071                STS         BOX         7f2029
0074                STSW        BOX         eb2026
0077                STT         BOX         872023
007a                STX         BOX         132020
007d                SUB         REF1        1f2010
0080                SUBR        A,S         009404
0082    LABEL6      SVC         T           00b050
0084                TD          DEVICE      e32012
0087                TIO                     0000f8
0088                TIX         STAT        2f2008
```

```
008b            TIXR    A           00b800
008d            WD      DEVICE      df2009
0090    REF1    WORD    10          00000a
0093    STAT    WORD    1           000001
0096    EOF     BYTE    c'EOF'      454f46
0099    DEVICE  BYTE    X'05'       000005
009a    RETADR  RESW    1
009d    BOX     RESW    1
00a0    TEST    WORD    1           000001
00a3            +LDS    EOF         6f100096
00a7            LDF     FLOAT1      732015
00aa            COMPF   FLOAT2      8b2015
00ad            ADDF    FLOAT2      5b2012
00b0            DIVF    FLOAT2      67200f
00b3            MULF    FLOAT2      63200c
00b6            SUBF    FLOAT2      5f2009
00b9            STF     FLOAT1      832003
00bc            FIX                 0000c4
00bd            FLOAT               0000c0
00be            NORM                0000c8
00bf    FLOAT1  WORD    1.5         0000c03f
00c2    FLOAT2  WORD    10.5        00002841
00c5            END     PROG
```

어셈블러 패스1 , 패스2가 끝나면 intermediate structure array에 저장되어 있는 데이터를 CreateProgramList(), CreateObjectCode() 함수를 호출해서 object file과 List file을 생성한다. 위에 실행화면은 CreateProgramList 함수가 호출된 후에 생성된 sicxe.list 파일의 캡쳐 화면이다. list파일을 보면 00a3까지는 instruction set 1에 결과를 나타내고 00a7부터 00be까지는 instruction set 2에 결과를 나타낸다. 00bf, 00c2는 실수를 저장한 결과이다.

## 소감 :

이번 프로젝트 과제를 수행하면서 어셈블러의 동작 과정에 대해 매우 깊게 이해할 수 있었다. 기존 어셈블러의 실행과정을 어렴풋이 알았다면 프로젝트를 마무리한 현재에는 매우 자세한 동작 과정을 알 수 있게 되었다. 시스템 프로그램이라는 것이 응용프로그램처럼 눈에 띄는 역할을 하지 않지만 우리가 모르는 사이 매우 중요한 역할을 하고 시스템프로그램이 존재하지 않는다면 응용프로그램도 사용할 수 없음을 깨달았다.