시스템프로그래밍(나) Project 1

- SIC/XE Assembler -

컴퓨터학부 정해성

목차

1. 동기 및 목적

본 보고서는 지난 파싱 과제의 연장선인, SIC/XE 문법으로 작성된 소스를 오브젝트 프로그램으로 바꾸는 어셈블러를 구현하는 과제에 대한 보고서입니다.

Pass 1에서 중간 파일을 생성하지 않고 오브젝트 프로그램 생성에 필요한 정보들을 토큰 테이블과 심볼 테이블, 리터럴 테이블에 모두 저장해두는 방식으로 진행하였습니다. 그리고 오브젝트 프로그램을 생성할 때에는 `object_code` 객체 하나로 모든 것을 구성할 수 있게 설계하였습니다.

2. 설계/구현 아이디어

- **instruction 파싱, 소스 파싱**

  파싱하는 과정은 지난 과제와 동일하나, inst_table.txt 형식이 콤마 문자로 구분되어 있던 것을 탭 문자('₩t')로 구분하는 것으로 변경하면서 파싱하는 구현 코드를 간결하게 할 수 있었습니다.

- **Pass 1**

  Pass 1 단계에서는 Pass 2 단계에서 활용하기 위한 토큰 테이블, 심볼 테이블, 리터럴 테이블을 설정합니다. Pass 2에서는 이 세 테이블만 활용하여 오브젝트 프로그램을 구성할 것이므로 각 구조체 정의에 필드를 추가하는 것이 필요했습니다.

  우선 token 구조체에는 `addr` 필드를 추가하였습니다. 이미 Pass 1에서 location counter를 계산해주고 있기 때문에 Pass 2에서 한 번 더 계산하는 것이 번거롭다고 판단했기 때문입니다. 해당 필드는 Pass 2에서 PC 상대주소를 계산할 때 활용됩니다. 그리고 symbol 구조체에는 `rflag`와 `csect_name`이라는 필드를 추가하였습니다. `rflag`는 해당 심볼의 주소가 상대적인지를 확인하는 플래그이며, `csect_name`은 심볼이 정의된 control section의 이름을 담고 있습니다. literal 구조체에는 `bytes`, `type`, `value` 필드를 추가하였습니다. 해당 리터럴이 차지하는 메모리의 바이트 크기를 `bytes` 필드에 넣어두었으며, `type`과 `value`는 이름에서도 알 수 있듯이 리터럴의 실제 타입과 값을 나타냅니다.

  Pass 1은 소스 코드의 각 라인을 파싱한 결과를 직접 활용하며, 크게 심볼 처리, 리터럴 처리, 연산자 처리 총 세 단계로 구분할 수 있습니다.

  심볼 처리 단계에서는 토큰의 label이 존재할 때만 수행됩니다. 일반적으로는

location counter(locctr)와 해당 label로 심볼을 구성하여 심볼 테이블에 삽입하고, 이 과정은 insert_label_into_symtbl() 함수에서 수행합니다. 하지만 EQU 지시어가 사용된 경우에는 값을 계산해주는 등 특수한 처리가 있어야 합니다. 이 과정은 calculate_equ() 함수에서 수행하며, 피연산자가 "*"일 경우 해당 위치의 locctr로 대입해주고 수식이 있다면 이름이 같은 심볼을 심볼 테이블에서 찾아 해당 주소값을 더하거나 빼주는 방식입니다. 덧셈 연산과 뺄셈 연산만 가능하며 괄호가 들어간 연산은 아직 구현하지 못했습니다.

다음은 리터럴 처리 단계입니다. 이 단계는 단순히 피연산자의 첫 번째 문자가 '=' 이면 수행되는 것으로, insert_literal_into_littbl() 함수에서 구현하고 있습니다. 리터럴과 관련한 함수가 하나 더 정의되어 있는데, set_literal_addr()이라는 함수가 그것입니다. 리터럴이 정의되는 메모리 주소는 바로 정해지지 않습니다. END로 프로그램이 종료되거나, CSECT로 새로운 control section이 생기거나, LTORG 지시어로 리터럴의 위치를 지정해줄 때 비로소 리터럴의 주소가 정해집니다. 이 세 지시어 중 하나를 만났을 때 set_literal_addr() 함수는 호출함으로써 주소가 정해지지 않은 채 쌓여 있던 리터럴들을 해소시키고 있습니다.

마지막으로 수정사항이 가장 많았던 연산자 처리 단계입니다. 연산자가 지시어가 아니라면 instruction table에서 대응되는 명령어를 찾고 형식에 맞춰 locctr 값을 갱신해줍니다. 만약 해당 연산자가 지시어라면 해당 지시어의 동작에 맞게 구현하였는데, EXTREF, EXTDEF는 Pass 2에서 수행되므로 제외하였습니다.

- **Pass 2**

오브젝트 프로그램을 구성하기 위해 필요한 레코드의 구조체(header_record, text_record, end_record, modify_record, define_record, reference_record)를 정의하고 control_section 구조체의 필드로 적절히 정의하였습니다. 그리고 오브젝트 프로그램은 여러 control section으로 구성될 수 있기 때문에 object_code 구조체의 필드는 control_section 배열로 정의되어 있습니다.

Pass 2에서는 Pass 1에서 파싱한 토큰이 저장되어 있는 토큰 테이블을 순회하면서 연산자에 맞게 적절히 레코드를 채워 넣습니다. 각 레코드의 내용을 채우는 과정을 간단히 설명하겠습니다.

- header_record: START나 CSECT가 등장하여 새로운 control section가 시작될 때 정의합니다.
- define_record: 먼저 symbol 필드와 심볼의 개수를 나타내는 symbol_cnt를

EXTDEF가 등장했을 때 대입해줍니다. 그리고 이후에 토큰에서 label이 나올 때마다 define_record에 등록된 label인지 확인하고, 존재한다면 대응되는 위치에 상대주소 값을 넣습니다. 해당 label의 상대주소 값은 토큰의 addr 필드를 참조하여 확인합니다.

- reference_record: EXTREF가 등장했을 때 피연산자의 심볼을 바로 추가해줍니다.
- end_record: header_record가 정의될 때 함께 정의되며, header_record에도 지정되어 있는 시작주소로 설정됩니다.
- text_record, modification_record: 토큰의 연산자가 instruction_table에 존재한다면, opcode와 토큰의 nixbpe 필드를 적절히 조합하여 오브젝트 코드를 삽입합니다. 만약 EXTREF로 정의된 심볼을 피연산자로 사용하고 있다면 주소 부분을 0으로 치환하고, modification_record의 필드를 추가합니다.

- **오브젝트 프로그램 생성**

오브젝트 프로그램 생성 단계에서는 Pass 2에서 정의한 object_code의 레코드들을 오브젝트 프로그램의 형식에 맞게 출력합니다. make_objectcode_output()이라는 함수에서 수행합니다.

3. **수행 결과**

output_symtab.txt

```
COPY    0
FIRST   0       1 COPY
CLOOP   3       1 COPY
ENDFIL  17      1 COPY
RETADR  2A      1 COPY
LENGTH  2D      1 COPY
BUFFER  33      1 COPY
BUFEND  1033    1 COPY
MAXLEN  1000    1 COPY
RDREC   0
RLOOP   9       1 RDREC
EXIT    20      1 RDREC
INPUT   27      1 RDREC
MAXLEN  28      1 RDREC
WRREC   0
WLOOP   6       1 WRREC
```

1의 의미는 상대주소임을 의미하는 `rflag` 값입니다.

output_littab.txt

```
=C'EOF' 30
=X'05'  1B
```

output_objectcode.txt

```
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
E000000
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
E000000
HWRREC 00000000001C
RLENGTHBUFFER
E000000
```

text_record, modification_record는 아직 구현하지 못했습니다.