시스템프로그래밍(F049-1)

과제1 레포트

Infix Notation Calculator

아주대학교 소프트웨어학과

202220775

박민정

제출일: 2025.04.13.

가)프로그램 개요

본 과제의 목표는 사용자가 입력한 중위 표기 수식을 후위 표기법(Postfix)으로 변환하고, 이를 계산하여 결과값을 출력하는 프로그램을 구현하는 것이다. 입력은 문자열 형태로 받으며, "2+3*4" 같은 수식을 처리하여 "234*+" 형식의 후위 표기법으로 바꾸고, 계산 결과 14를 출력한다. 이 작업은 C언어로 알고리즘을 구현하고 이를 기반으로 하여, 동일한 기능을 SIC 어셈블리어로 구현하였다.

나)프로그램 구조

전체 프로그램은 크게 입력 처리-> 후위표기-> 변환결과 출력-> 후위표기 계산-> 최종 결과 출력의 흐름으로 동작하며, 이 흐름은 각각의 서브루틴으로 세분화된다.

1.MAIN: 전체 흐름 제어

메인 프로그램에서는 프로그램의 초기화를 수행한 뒤, user 로부터 중위표기식 입력을 받고, 이를 후위표기식으로 변환한 후 출력하며, 계산 결과를 출력하고 종료하는 일련의 흐름을 관리한다. 각 주요 기능은 별도의 서브루틴(JSUB)으로 모듈화 되어있다.

2. RDREC: 사용자 입력 문자열 읽기

입출력 장치로부터 한 글자씩 입력을 받아 READRES 에 저장하고, 엔터(ASCII 10)가 입력될시 입력을 종료한다. 입력된 문자열의 길이는 LENGTH 에 저장되며, 후속 루틴에서 반복문의 종료 조건으로 사용된다.

3. INTOPOST: 중위 표기식을 후위 표기식으로 변환

이 서브루틴은 핵심적인 변환 알고리즘을 담당한다. READRE 에 저장된 입력 수식을 하나씩 읽어가며, 숫자는 POSTFIX 배열에 바로 저장하고, 연산자는 스택에 쌓은 후 우선순위를 비교해 pop 여부를 결정한다. 이 과정에서 연산자는 ASCII 코드값으로 비교되며, 우선순위는 GETPRE 를 통해 구한다. 변환된 후위표기식은 POSTFIX 에 저장된다. 중간 스택 연산은 PUSH, POP 서브루틴을 통해 수행되며, 연산자 스택은 1 바이트 기준으로 처리된다.

4. GETPRE: 연산자 우선순위 반환

연산자의 ASCII 값에 따라 +, -는 우선순위 1, *, /는 우선순위 2 를 부여하여 A 레지스터에 저장한다. 이를 통해 INTOPOST 에서 현재 연산자와 스택 top 연산자의 우선순위를 비교할 수 있다.

5. WRREC1: 후위표기식 출력

POSTFIX 배열에 저장된 문자열을 한 글자씩 꺼내어 출력 장치에 출력한다. 문자열 끝까지 반복 출력하며, 마지막에 개행 문자가 출력된다.

6. CALCU: 후위 표기식 계산

POSTFIX 를 다시 읽어가며, 숫자는 스택에 push 하고, 연산자는 두 개의 숫자를 pop 하여 계산 후 다시 push 한다. 계산 연산은 ADD, SUB, MUL 명령어로 수행되며, 숫자 스택은 3 바이트(정수용)로 구성되기 때문에 STACK2, PUSHW, POPW 를 사용한다. 최종 결과는 스택에 남은 값 하나로 계산이 완료된다.

7. PUSH, POP, PUSHW, POPW: 스택 연산

-PUSH, POP: 1 바이트 연산자 스택용 서브루틴 (POSTFIX 구성 시 사용)

-PUSHW, POPW: 3 바이트 숫자 스택용 서브루틴 (후위식 계산 시 사용)

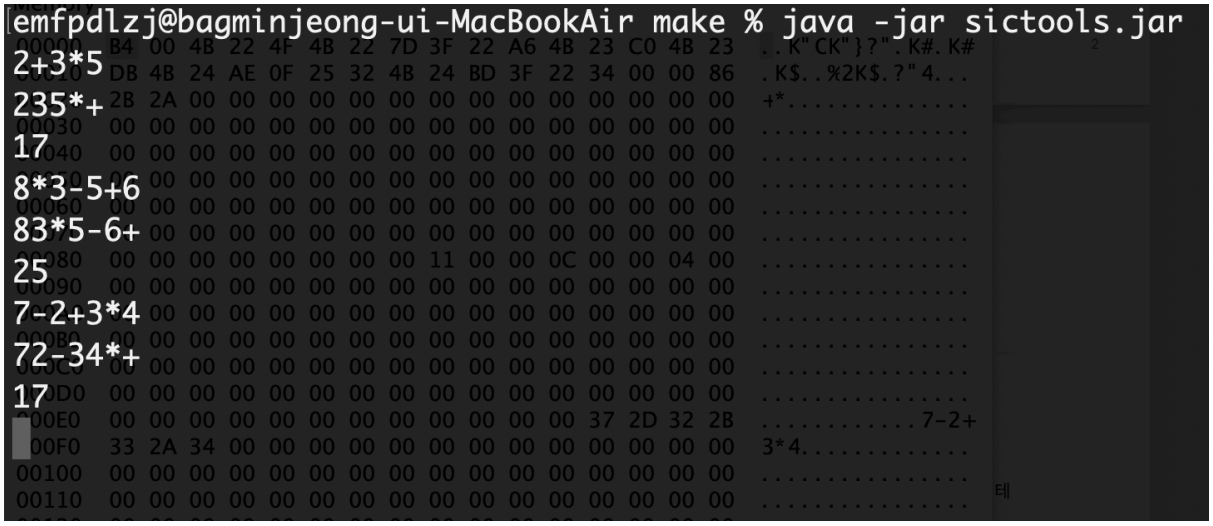
각 스택 포인터는 전용 변수(STACKP)로 관리되며, 값 삽입 시 주소 증가, 추출 시 주소 감소를 수동으로 구현했다.

8. PRINTDEC: 계산 결과를 10 진수로 변환하여 출력

SIC/XE 는 직접적인 정수 출력이 불가능하기 때문에, 계산된 정수값을 백의 자리, 십의 자리, 일의 자리로 나눠 각각 ASCII 코드로 변환한 뒤 출력 장치로 전송했다. 나눗셈, 곱셈, 뺄셈 등을 조합하여 수동으로 10 진수 변환을 수행한다.

이처럼 프로그램은 입력부터 계산까지 각 기능이 JSUB 로 나뉘어 모듈화되어 있고, READRES, POSTFIX, STACK, STACK2 등 다양한 버퍼를 통해 데이터를 교환한다.

다)테스트케이스 실행 결과



라)분석 및 고찰

이번 과제는 중위표기식을 후위표기식으로 변환하고, 변환된 후위표기식을 계산하는 기능을 C 언어와 SIC 어셈블리어 두 가지 방식으로 구현하는 프로젝트였다. 구현을 진행하면서 두 언어의 구조적 차이와 접근 방식의 차이를 직접적으로 체감할 수 있었고, 특히 하드웨어와 가까운 저수준 언어인 어셈블리어에서는 C 언어보다 훨씬 더 세세하게 메모리와 연산을 관리해야 한다는 점을 깨달았다.

먼저 구현 과정에서 가장 큰 어려움은 스택 구현이었다. C 언어에서는 배열과 전역 변수를 사용하여 숫자 스택과 연산자 스택을 명확하게 분리하고, push/pop 함수로 추상화할 수 있어 로직 구성에 집중할 수 있었다. 반면, SIC 어셈블리어에서는 별도로 RESB 혹은 RESW 로 메모리를 할당하고, 스택 포인터를 직접 증가 및 감소시키며 스택 구조를 구현해야 했다. Calcupost 와 Intopostfix 에서 각각 3 바이트와 1 바이트 스택을 나눠 사용해야 했기 때문에, 서브루틴도 따로 만들어야 했고, 이로 인해 전체 코드가 두 배 이상 복잡해졌다. 메모리 접근에 대한 이해가 좀 더 있었다면 코드를 많이 줄일 수 있었을 것 같다.

또한, 문자와 정수 간 변환 역시 주의가 필요했다. C 언어에서는 '3' - '0'처럼 간단한 연산으로 ASCII 코드를 숫자로 바꾸는 것이 가능했지만, 어셈블리어에서는 SUB #48 명령을 반복적으로 사용해야 했고, 다시 문자로 출력할 때는 ADD #48 을 사용해 직접 변환 과정을 넣어야 했다. 특히 연산기호 비교에서도 COMP #42 (*), COMP #43 (+) 처럼 ASCII 코드 값을 기억하고 비교해야 했기 때문에 연산자 처리가 매우 불편하고 직관성이 떨어졌다. 숫자나 기호를 그대로 비교하지 못하고, ASCII 코드값을 외워서 명령어에 넣는 방식은 초반에 많이 헷갈렸던 부분 중 하나였다.

입출력 관련 부분에서도 주의가 많이 필요했다. RD, WD 명령을 통해 1 바이트 단위로 입출력을 처리해야 하다 보니, 입력 문자열을 저장하는 버퍼(READRES), 출력할 후위표기 배열(POSTFIX) 등 각 입력/출력 버퍼의 크기와 인덱스를 수시로 관리해야 했다. C 언어에서는 배열의 경계나 입력 스트링 처리에 크게 신경 쓰지 않아도 되는 반면, 어셈블리에서는 하나라도 인덱스 계산이 틀리면 엉뚱한 메모리 접근이 발생하기 때문에 작은 실수도 큰 오류로 이어질 수 있었다. 오류를 예방하기위해 같은 기능으로 여러군데에서 쓰이는 변수의 경우, 여러번 다르게 변수를 정의하여 사용하다보니 메모리 공간 낭비가 많았던 것 같다. 입출력 버퍼의 크기 제한, 입력 종료 조건(엔터키), 반복문 인덱스 등을 명확하게 다뤄야 했고, 그 과정에서 자연스럽게 메모리를 다룰 때 안정성을 높이기 위한 습관도 생기게 되었다.

SIC 어셈블리어에서 가장 불편하게 느껴졌던 부분 중 하나는 조건문이나 반복문을 직접 구성해야 한다는 점이었다. C 에서는 while, if, else 같은 고수준 제어문을 직관적으로 작성할 수 있지만, 어셈블리에서는 조건 분기 명령어(JEQ, JLT, JGT)와 레이블을 조합해 흐름을 수작업으로 제어해야 한다. 단순한 반복문 하나를 구성하는 데에도 레지스터 초기화, 증가, 비교, 분기처리까지 모두 나열해야 했으며, 이로 인해 로직의 복잡도와 디버깅 시간도 함께 늘어났다.

또한, 연산 과정에서 사용할 수 있는 주요 레지스터가 A 하나뿐이라는 점도 불편하게 작용했다. A 레지스터를 기준으로 연산이 수행되기 때문에, 중간값을 처리하거나 다른 연산을 준비하려면 STA 로 값을 따로 저장하거나 CLEAR A 를 자주 호출해야 했다. 이런 제약 속에서 레지스터 관리가 익숙하지 않은 상태로 작업하다 보니 초반에는 A 레지스터에 값이 덮어쓰여 계산 오류가 나는 일이 자주 발생했다. 더불어, JSUB 명령어를 사용할 때는 L 레지스터에 저장된 복귀 주소가 덮어쓰기 되지 않도록 관리하는 것도 매우 중요했다. 중첩 호출이 생길 수 있는 구조에서는 L 값을 임시로 다른 메모리 공간에 저장해두고 복구하는 방식으로 안정성을 확보해야 했다.

어셈블리어를 사용하면서 생소했던 개념 중 하나는 바로 "레지스터에 접근한다"는 개념 자체였다. C 에서는 변수나 배열을 선언하고 그 이름을 이용해 자유롭게 접근할 수 있지만, 어셈블리에서는 모든 데이터가 메모리 주소에 기반하며, 이를 다루기 위해선 A, X, L 같은 제한된 레지스터에 값을 올리고 내리는 과정을 반드시 거쳐야 한다. 이 과정은 불편하면서도 동시에 컴퓨터의 동작 방식에 대해 더욱 구체적으로 이해할 수 있는 기회를 제공했다.

또 하나의 불편했던 점은 구간 주석이 지원되지 않는다는 점이었다. C 언어처럼 /* ... */ 형식으로 코드 블록을 주석 처리할 수 없고, 한 줄씩 .이나 ;로 시작하는 주석을 반복적으로 달아야 해서 디버깅이나 테스트 과정에서 번거로움이 컸다.

메모리 관련해서도 차이가 많았다. C 에서는 int arr[10]처럼 간단히 배열을 선언하고 인덱스로 접근할 수 있지만, 어셈블리에서는 .RESB 로 공간을 확보하고 X 레지스터와 함께 인덱싱하거나 주소를 수동 계산해서 접근해야 하므로 배열 구조를 직접 구성하는 것이 매우 번거로웠다. 이 점은 초반 설계에서 시간을 많이 소모하게 만든 요소 중 하나였다.

하지만 그에 반해 편했던 점도 있다. 자료형 구분이 없다는 점은 생각보다 유용했다. 정수, 문자 등의 타입을 구분하지 않고 바이트 단위로만 데이터를 처리하면 되기 때문에, 타입 캐스팅이나 타입 선언에 신경 쓸 필요가 없어 구현 자체는 단순했다. 이 점은 오히려 어셈블리어가 가진 구조적 단순함이 장점으로 느껴지는 부분이었다.

결과적으로 이 프로젝트는 알고리즘 구현 능력뿐만 아니라, 저수준 언어에서의 메모리 구조 이해와 명령어 활용 능력을 동시에 요구하는 과제였다. 계산 결과가 정상적으로 출력되었을 때의 성취감은 C 언어보다 어셈블리어 쪽이 훨씬 더 컸고, 컴퓨터 내부 동작 원리에 대한 이해도 함께 깊어진 경험이었다. 이번 경험은 저수준 언어에 대한 두려움을 줄이고, 향후 임베디드 시스템이나 시스템 프로그래밍 분야로의 진입에 자신감을 줄 수 있는 좋은 계기가 되었다.