U-CODE Interpreter

[프로젝트]

분석 및 설계 보고서

과목명	소프트웨어프로젝트
학 과	컴퓨터공학과
이 름	김민성
제작기간	2015.11 ~ 2015.12

1 문제 개요

U-CODE Interpreter는 스택 기반으로 동작하는 간단한 CPU에서 실행되는 중간 언어로써, 입력을 U-Code로 작성한 프로그램으로 하고, 그에따른 프로그램의 실행결과를 화면에 출력하고 실행 과정 및 통계정보를 보여주는 파일(*.lst)를 생성하는 U-CODE Interpreter를 제작한다.

2 제한 요소

- 배열, 제어문, 함수를 모두 사용하는 세 가지 이상의 C 소스 프로그램을 준비하고 이 프로그램에 대응하는 U-Code로 작성한 프로그램을 생성한다.
- U-Code 설명에서 제시한 사항을 준수한다.
- 단계별 처리결과를 확인할 수 있는 기능(화면 및 파일)을 제공한다.
- U-Code의 임시 저장 장치는 단하나의 스택만 존재하므로, 임시 저장할 장소는 스택으로 제한한다.
- 메모리는 1차원 배열로 정의한다.
- U-Code의 모든 데이터는 단일 한크기의 정수형이므로, 다루는 데이터 타입은 int형으로 제한한다.
- U-Code 프로그램은 정해진 열을 지켜 작성한다.1~10번째 열까지는 레이블을 사용하고, 11번째 열은 공백,12번째 열부터 명령어가 나타날 수 있으며, 명령어 및 피연산자 사이에는 최소한 하나 이상의 공백이 있어야한다.
- U-Code(*.uco)파일은 한번에 하나씩만 입력받는다.
- 시스템함수의 입출력의 경우, 한번에 하나의 정수만 입력받고 하나의 정수만을 출력한다.
- %로 시작하는 부분부터 그 행의 끝까지 주석으로 인식한다.

3 요구 분석

1. 기능적 요구사항

가. 출력으로 프로그램의 실행결과와 실행 과정 및 통계정보를 보여주는 파일(*.lst)를 제공하여야한다.

나. 단계별 처리 결과를 확인할 수 있는 기능(화면 및 파일)을 제공한다.

2. 비기능적 요구사항

가. 입력으로는 U-Code로 작성한 프로그램(*.uco)만을 입력받는다.

나. 시스템 함수인 write는 사용자로부터 한 개의 정수만을 입력받고, read는 사용자로부터 한 개의 정수만을 출력한다.

4 U-CODE 명령어 분석 및 설계

1. 프로그램 구성 명령어

해당 명령어들은 인터프리터가 동작 하기전에 일련의 준비하는 단계들로, 실행한 명령 횟수에서는카운트 되지않는다.따라서,사용한 명령횟수에는 카운트되지만 실행한 명령횟수에는 포함시키지않는다.

명령어	의미	동작	설계(sudo-code)
nop	no operation	아무 작업도 수행하지 않으며, 주로 레이블위치에 사용됨	Assemble 단계에서 레이블의 위치 인덱스를 저장하여 놓고, execute 단계에서는 아무 역할도 하지않음
bgm n	begin	프로그램의 시작점을 나타내며, n은 전역변수의 총량을 나타낸다.	Assemble 단계에서,메모리 1차원 배열에 대해서, 전역변수의 구간을 정의한다.
sym b n s	symbol	변수가 속한 블록(b)과 블록 내에서의 오프셋(n) 및 크기(s)를 나타낸다.	변수의크기에 맞게 배열의 구간을 정의한다.
end	end	함수 또는 프로그램의 끝을나타낸다.	함수또는 프로그램의 끝을 나타내며, 프로그램의 끝인 경우엔,PC 값에 특정값을 주어서 프로그램을 종료한다.

2.데이터 이동 연산자

U-CODE에서는 메모리(1차원 배열)에 있는 모든 데이터는 연산을 실행하기 전에 스택으로 이동해야 하고, 저장하기 위헤서는 스택에서 메모리 배열로 이동해야한다.이때 각 변수의 오프셋은 sym명령에서 할당된 번지수를 이용한다. 그리고, 이 명령에서는 메모리에 접근하므로, 메모리 접근 횟수를 카운트 해줘야한다.

명령어	의미	동작	설계(sudo-code)

lod b n	load	b 블록 n 오프셋의 데이터를 스택에 넣는다.즉,(b n)으로 표현되는 주소에 있는 변수의 값이 스택에 저장된다.	stack.push (Memory[b , n])
lda b n	load address	b 블록 n 오프셋의 실제 메모리 번지를 스택에 넣는다.즉,(b n)으로 표현되는 주소 자체가 스택에 저장되며, 배열 참조를 위해 활용된다.	stack.push ((b,n)으로 표현된주소)
ldc c	load constant	상수값 c가 스택에 저장된다.	stack.push(c)
str b n	store	스택 꼭대기의 값을 (b n)으로 표현되는 주소의 메모리에 저장한다.	Memory[b,n] = stack.pop()
ldi	load indirect	간접 주소법을 이용해 메모리의 값을 스택에 가져 온다.스택 꼭대기의 값을 pop하여 주소값으로 사용하고, 데이터를 스택에 저장한다.	stack.push (Memory[stack.pop()])
sti	store indirect	간접 주소법을 이용해 스택 꼭대기의 값을 메모리에 저장한다.저장할 변수의 주소와 저장할값,두개가 스택에 pop 된다.	temp = stack.pop() Memory[stack.pop()] = temp

3.단항 연산자

단항 연산자의 모든 피연산자는 스택의 맨위에 있으며 결과는 다시 스택에 저장된다.따라서, 메모리에대한 접근횟수에서는 카운트되지않고, 실행한 명령어 횟수에서만 카운트 된다.

명령어	의미	동작	설계(sudo-code)
not	not	피연산자의 진리값을 변경	stack[top] = !stack[top]
neg	negation	피연산자의 음양값으로 변경	stack[too] = -stack[top]
inc	increment	피연산자의 값이 하나 증가	++stack[top]
dec	decrement	피연산자의 값이 하나 감소	--stack[top]
dup	duplicate	스택 꼭대기의값을 복사한값을 스택 꼭대기에 적재	stack[top++]=stack[top]

4.이항 연산자

이항연산자의 모든 피연산자는 스택꼭대기와 그 하나 아래의 값이며, 아래의값이 앞에 오고, 꼭대기의 값이 뒤에 오는 피연산자이다.이 명령어들역시 메모리에 직접 접근하지않고, 바로 ,스택에서 연산이 이루어지므로, 메

모리 접근 횟수는 카운트 하지않는다.

명령어	의미	동작	설계(sudo-code)
add	add	덧셈 연산을 수행한다.	stack[top-1] = stack[top-1] + stack[top] top--
sub	subtract	뺄셈 연산을 수행한다.	stack[top-1] = stack[top-1] - stack[top] top--
mult	multiply	곱셈 연산을 수행한다	stack[top-1] = stack[top-1] * stack[top] top--
div	divide	나눗셈 연산을 수행한다.	stack[top-1] = stack[top-1] / stack[top] top--
mod	modulo	나머지 연산을 수행한다.	stack[top-1] = stack[top-1] % stack[top] top--
gt	greater than	> 논리 연산을 수행한다.	stack[top-1] = stack[top-1] > stack[top] top--
lt	less tan	< 논리 연산을 수행한다.	stack[top-1] = stack[top-1] < stack[top] top--
ge	gt or equal	>= 논리 연산을 수행한다.	stack[top-1] = stack[top-1] >= stack[top] top--
le	lt or equal	<= 논리 연산을 수행한다.	stack[top-1] = stack[top-1] <= stack[top] top--
eq	equal	== 논리 연산을 수행한다.	stack[top-1] = stack[top-1] == stack[top] top--
ne	not equal	!= 논리 연산을 수행한다.	stack[top-1] = stack[top-1] != stack[top] top--
and	and	&& 논리 연산을 수행한다.	stack[top-1] = stack[top-1] && stack[top] top--
or	or	|| 논리 연산을 수행한다.	stack[top-1] = stack[top-1] || stack[top] top--

swp	swap	스택의 꼭대기값과 스택의 꼭대기 아래의 값을 서로 교체한다.	temp = stack[top-1] stack[top-1] = stack[top] stack[top] = temp

5.흐름 제어

이 명령어는 피연산자로 레이블을 가지고있으며, 메모리접근은 이루어지지 않으므로, 메로리 접근횟수는 카운트하지않는다.또한, tjp나 fjp는 스택꼭대기의 값을 기준으로 참이면 점프하는 tjp , 거짓이면 점프하는 fjp가 있다.

명령어	의미	동작	설계(sudo-code)
ujp label	unconditional jump	지정한 label로 무조건 이동	label로 점프
tjp label	jump on true	stack[top]의 값이 참이면 label로 이동	if(stack.pop()) label로 점프
fjp label	jump on false	stack[top]의 값이 거짓이면 label로 이동	if(!stack.pop()) label로 점프

6.함수 정의 및 호출

명령어	의미	동작	설계(sudo-code)
call label	call	label로 지정된 함수를 호출	이동하기전에 복귀주소를 저장하고,해당레이블로 PC값 이동
ret	return	반환값없이 리턴	원래주소로 PC값 이동
retv	return with value	반환값이 있는 리턴.retv를 사용하기 전에 반환값을 스택에 저장해야한다.	stack.push(반환값) 원래주소로 PC값 이동
ldp	load parameters	함수의 실인자들을 스택에저장.이 명령 다음부터 호출을 호출하기 전까지 매개변수들을 스택에 저장한다.	이 명령어이후에는 호출전 준비작업들을 수행한다.
proc b n	procedure	함수의 시작을 나타내며 b는 블록 번호, n은 지역 변수 및 매개 변수의 전체 크기	지역변수 및 매개변수의 총 크기에 맞게 배열의 구간을 정의한다.

7.입출력 처리

U-CODE Interpreter는 입출력의 값을 정수 하나의 값으로 제한 한다.시스템함수는 크게 3가지만을 정의하여 사용한다.

명령어	동작	설계(sudo-code)
read(i)	외부 입력값을 읽어 스택 꼭대기에 저장된 주소로 저장한다.	temp = stack.pop() Memory[temp] = 입력된값
write(i)	스택 꼭대기의 값을 출력한다.	Print(stack.pop())
lf()	줄바꿈 문자(개행)를 출력한다.	Print('\n')

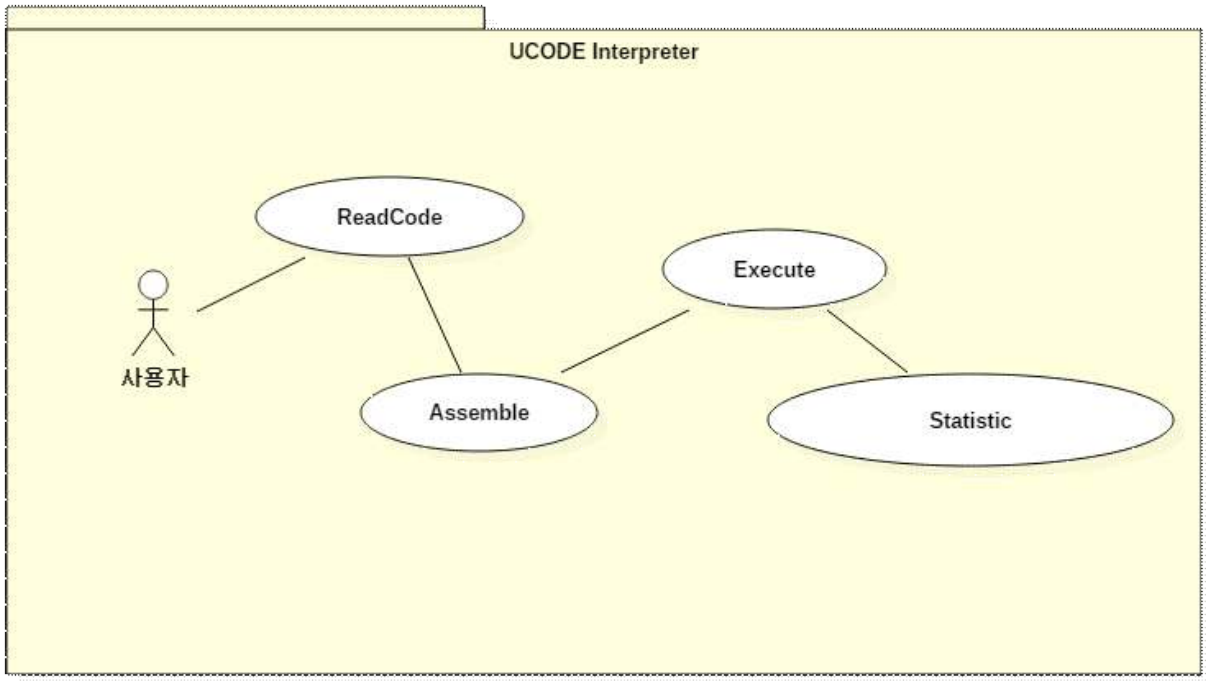
8.주석

명령어	동작	설계(sudo-code)
%	%로 시작하는 부분은 주석이므로, 해석하지않고 바로 건너뛴다.	Assemble단계에서 명령어를 읽을 때,%로 시작하기전 까지의 명령어를 추출하여 다시 저장한다.

5 유스케이스 명세화

■ 서비스 개요

이 프로그램은 사용자가 중간 언어인 U-CODE 파일(*.uco)를 입력하면, 그에 따른 프로그램의 실행결과를 화면에 출력하고,실행 과정 및 통계 정보를 보여주는 파일인(*.lst)를 생성하는 프로그램이다.



<U-Code Interpreter 유스케이스 다이어그램>

■ ReadCode 유스케이스

사용자로부터 UCODE 파일(*.uco)를 입력받아서,줄 단위로 코드를 읽어오는 단계이다.

■ Assemble 유스케이스

일반적으로, 중간언어의 경우엔, 2-pass로 작업이 이루어진다. 따라서, Assemble 단계에서는 코드를 처음부터 끝까지 읽으면서 준비 작업을 한다.이단계 에서는 먼저 ReadCode에서 줄단위의 명령어를 주석을 제외하고, 읽어오는 작업을 수행한다. 그리고, 각 레이블의 위치를 저장하고, 프로그램을 실행하는데 필요한 메모리 블록의 크기를 할당한다.

■ Execute 유스케이스

Assemble 단계에서 반환한 PC값을 받아서,명령어를 하나 하나 실행하고, 입출력

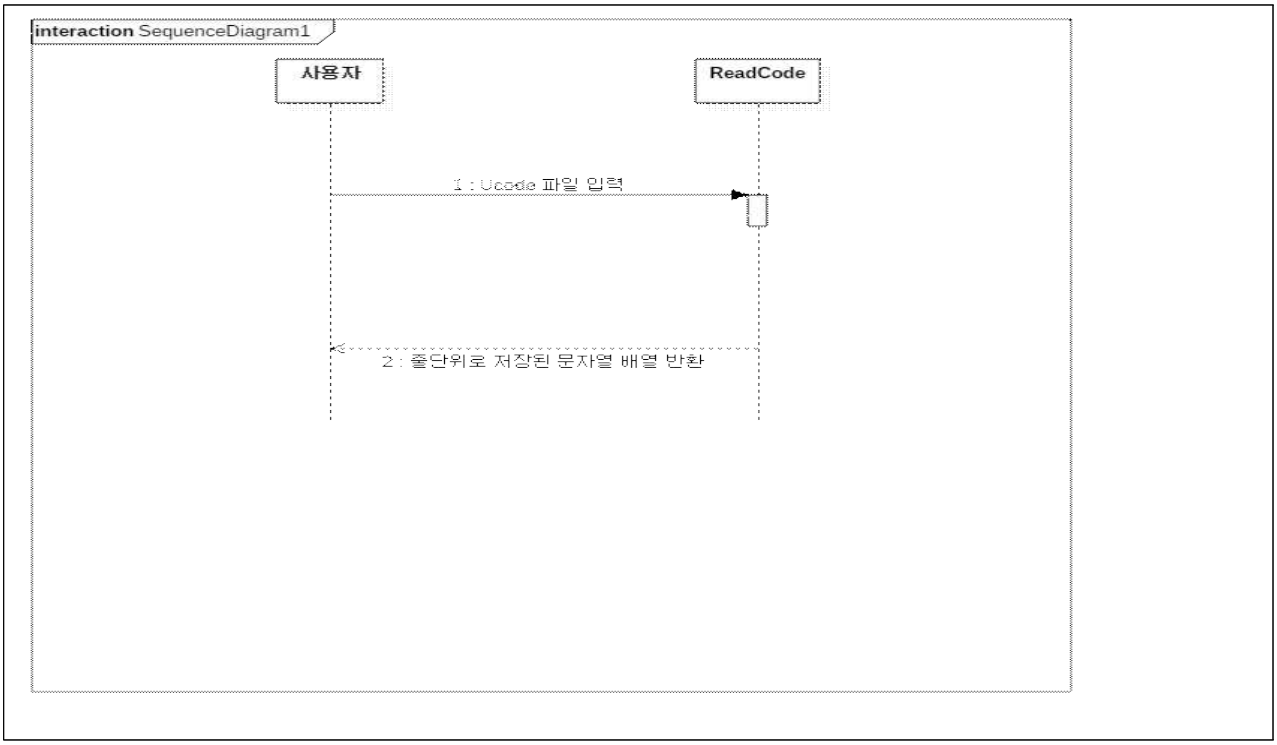
함수(Read,Write,lf)를 통해서,실행결과를 화면에 출력한다.

■ Statistic 유스케이스

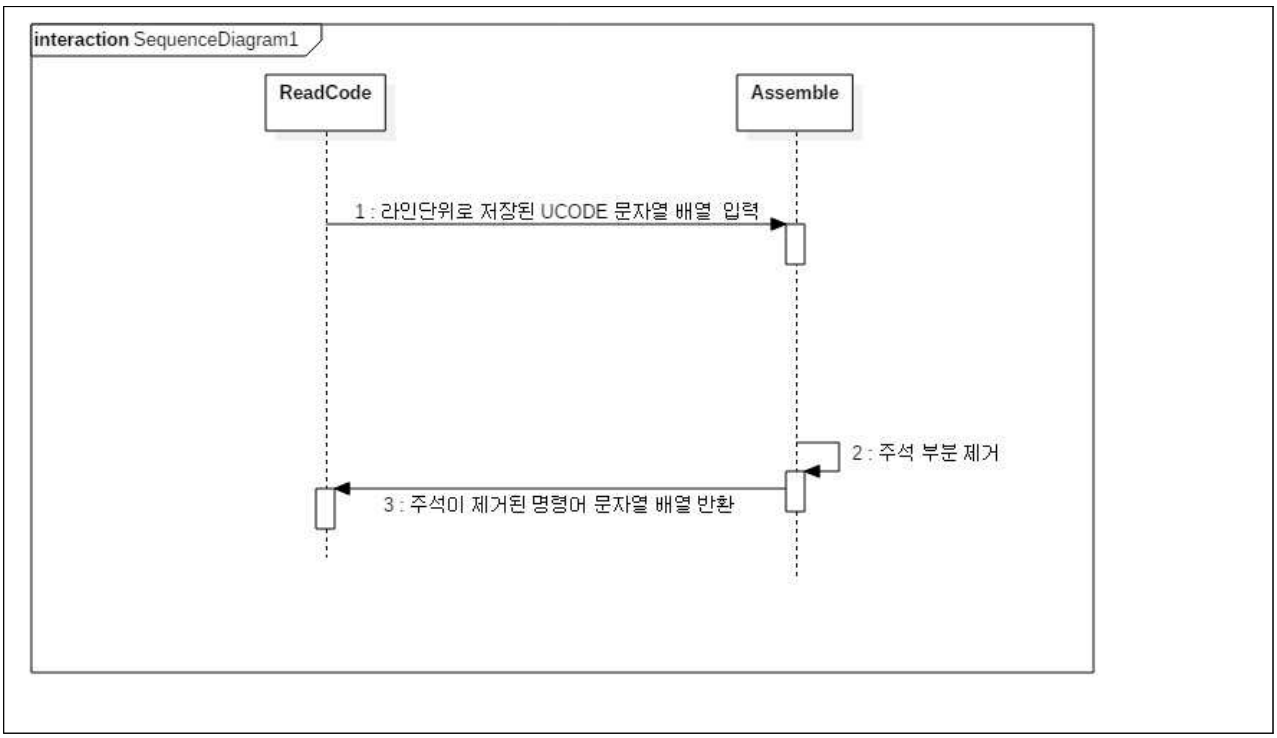
지금까지 실행한 명령어에 대한 통계파일를 생성한다.이 프로그램에서는 입력된 명령어,명령어의 실행 순서,각 명령어의 실행횟수,메모리에 접근한 횟수,명령어의 총 실행횟수, 실행결과를 출력한다.

6 유스케이스 실체화

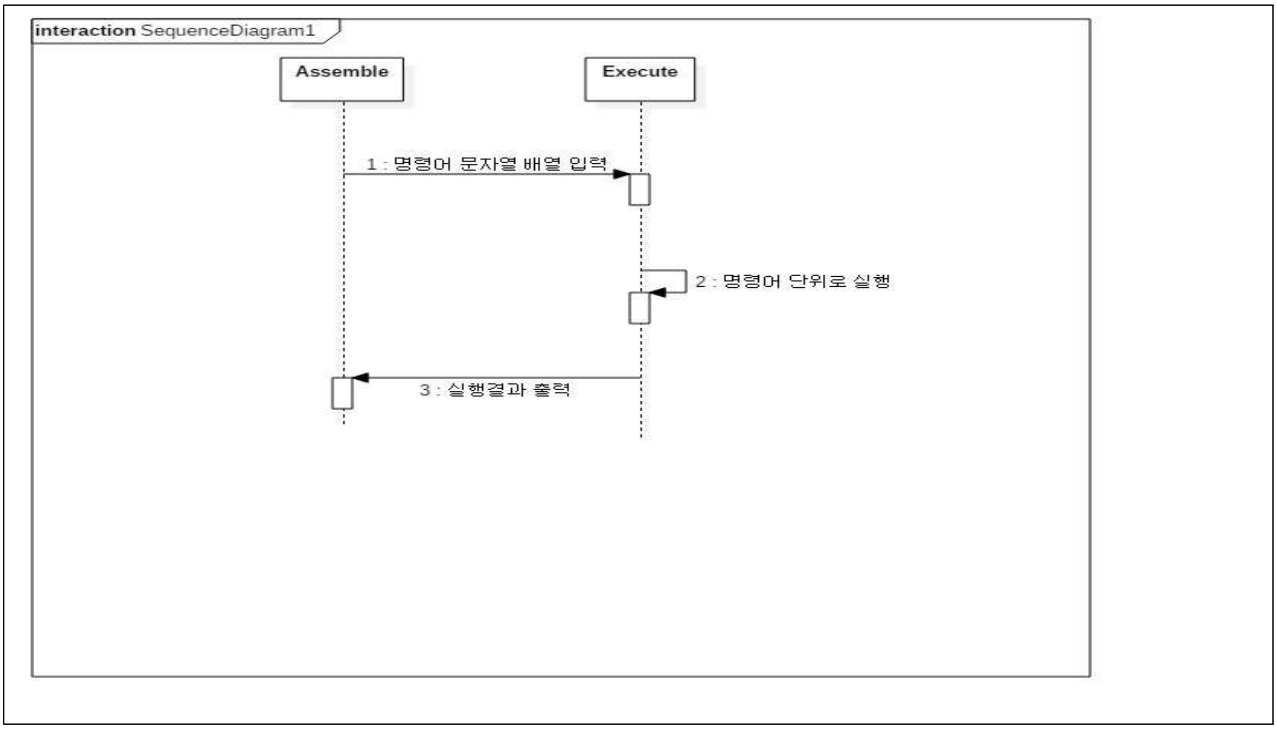
1. ReadCode 시퀀스 다이어그램



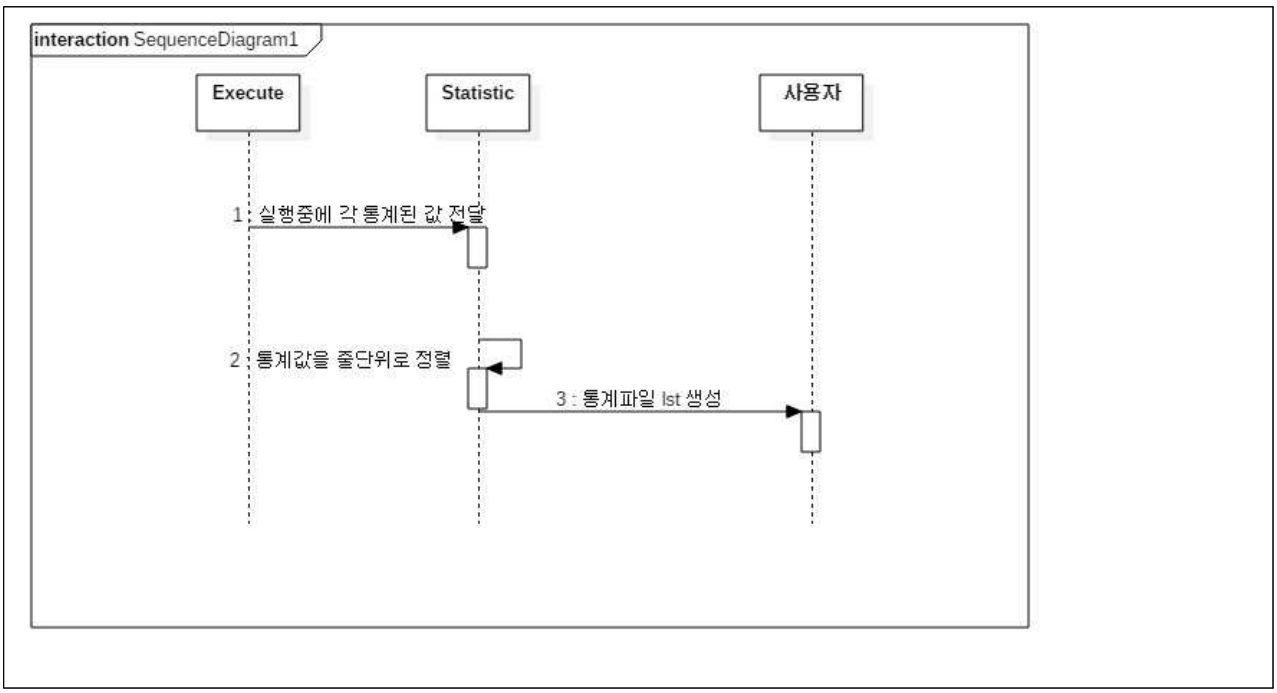
2. Assemble 시퀀스 다이어그램



3. Execute 시퀀스 다이어그램



4. Statistic 시퀀스 다이어그램



5. 대체 흐름

○ 입력 오류

▸ 입력으로 받은 U-CODE파일의 이름이 존재하지않을 경우, 오류 메시지 출력후 다시 입력받도록 한다.

▸ Interprete 진행 도중, 명령어에 산술에러(0으로 나눔)나 논리적 에러가 있으면 에러메세지를 출력하고 프로그램을 종료한다.

▸ Assemble 진행 도중, 존재하지않는 명령어가 있으면, 에러메세지를 출력하고 프로그램을 종료한다.

7 입출력 설계(Text-based-UI)

● 입력값

1.사용자로부터 10개의 입력값중 홀수만을 더한 합의값을 출력하는 프로그램

■ C파일

```
void main (void)
{
        int i;
        int result;
        int a[10];
        i = result = 0;
        while(i < 10)
        {
                odd(i,a);
                result = result + a[i];
                ++i;
        }
        printf("%d",result);
}

void odd (int i,int a[])
{
```

```
        int j;
        scanf("%d",&j);
        if(j % 2 == 0)
                j = 0;
        a[i] = j;
}
```

■ U-CODE(*.uco) [입력값]

```
        bgn 0           %프로그램의 시작 - 전역변수는 0개
        ldp             %함수 호출 전 인자를 스택에 올리기 위한 명령
        call main       %main 함수 호출
        end             %프로그램 종료
main    proc 0 12       %함수 시작 - 블록(0)과 지역변수의 총량 선언
        sym 0 0 1       %i 변수의 블록(0)과 오프셋(0)과 크기(1) 설정
        sym 0 1 1       %result 변수의 블록(0)과 오프셋(1)과 크기(1)설정
        sym 0 2 10      %a배열의 블록(0)과 오프셋(2)와 크기(10) 설정
        ldc 0           %상수 0을 스택에 적재
        dup             %스택의 top의값(0)을 복사하여 스택에 적재
        str 0 1         %스택의값을 블록 0, 오프셋 1위치에 적재
        str 0 0         %스택의값을 블록0, 오프셋 0위치에 적재
$$0     nop             %while 문의 시작점 - 반복시 돌아올 위치 $$0
        lod 0 0         %블록 0,오프셋 0의 값을 스택에 적재(i)
        ldc 10          %상수 10을 스택에 적재
        lt              %스택 꼭대기의 두값을 비교하여 결과를 적재
        fjp $$1         %스택의값이 거짓이면 $$1로 점프
        ldp             %함수 호출 전 인자를 스택에 올리기위한 명령
        lod 0 0         %블록 0,오프셋 0의 값을 스택에 적재(i)
        lda 0 2         %블록 0,오프셋 2의 주소를 스택에 적재(a의주소)
        call odd        %odd 함수 호출
        lod 0 1         %블록 0,오프셋 1의 값을 스택에 적재(result)
        lod 0 0         %블록 0,오프셋 0의 값을 스택에 적재(i)
        lda 0 2         %블록 0,오프셋 2의 주소를 스택에 적재(a의 주소)
        add             %스택 꼭대기의 두값을 더하여 결과를 적재(a[i])
        ldi             %스택 꼭대기 값을 주소로하여 있는값을 스택에 적재
        add             %스택의 두꼭대기에있는 값을 더하여 결과를 적재(reuslt + a[i])
        str 0 1         %스택 꼭대기의 값을 블록0,오프셋 1의 위치에 적재
        lod 0 0         %블록 0,오프셋 0의 값을 스택에 적재(i)
```

```
        inc             %스택의 값을 하나 증가시킴
        str 0 0         %블록 0,오프셋 0의 값을 스택에 적재(i)
        ujp $$0         %$$0의 위치로 무조건 점프
$$1     nop             %$$1의 위치 표시 (아무 동작 없음 - nop)
        ldp             %함수 호출 전 인자 적재
        lod 0 1         %블록 0,오프셋 1의 값을 스택에 적재(result)
        call write      %스택의 값을 출력
        end             %main 함수 종료
odd     proc 1 3        %함수 시작 - 블록(1)과 지역변수의 총량 선언
        sym 1 0 1       %i 인자의 블록(1)과 오프셋(0)과 크기(1) 설정
        sym 1 1 1       %a 인자의 블록(1)과 오프셋(1)과 크기(1) 설정
        str 1 1         %스택의 값을 블록 1,오프셋 1에 저장(a주소 인자)
        str 1 0         %스택의 값을 블록1,오프셋 0에 저장(i인자)
        sym 1 2 1       %j 인자의 블록(1)과 오프셋(2)과 크기(1) 설정
        ldp             %함수 호출 전 인자 적재
        lda 1 2         %블록 1,오프셋 2의 주소를 스택에 적재(j)
        call read       %read 함수 호출. 지정 주소에 입력값 저장
        lod 1 2         %블록 1,오프셋 2의 값을 적재(j)
        ldc 2           %상수 2를 스택에 적재
        mod             %스택의 두꼭대기에있는 값을 나눈나머지의 결과를 적재(j % 2)
        ldc 0           %상수 0을 스택에 적재
        eq              %스택 두꼭대기의 값을 비교하여 결과를 적재
        fjp $$2         %스택의 값이 거짓이면 $$2로 점프
        ldc 0           %상수 0을 스택에 적재
        str 1 2         %스택의 값을 블록1,오프셋 2에 저장(j인자)
$$2     nop             %$$2 위치 표시 (아무 동작 없음 - nop)
        lod 1 0         %블록1, 오프셋 0의값을 적재(i)
        lod 1 1         %블록1, 오프셋 1의값을 적재(a)
        add             %스택 꼭대기의 두값을 더하여 결과를 적재(a[i])
        lod 1 2         %블록1, 오프셋 2의값을 적재(j)
        sti             %스택 꼭대기의 값을 꼭대기 이전의값을 주소로하여 저장(a[i] = j)
        ret             %반환값을 가지지 않고 리턴
        end             %odd 함수 종료
```

● 출력값

■ 실행결과(출력 화면)

```
>>UcodeInterpreter.exe  odd.uco
[Assembling.....]
[Executing.....]
실행 결과
1
3
4
5
7
9
10
15
20
11
51
```

■ 통계 파일(*.lst)

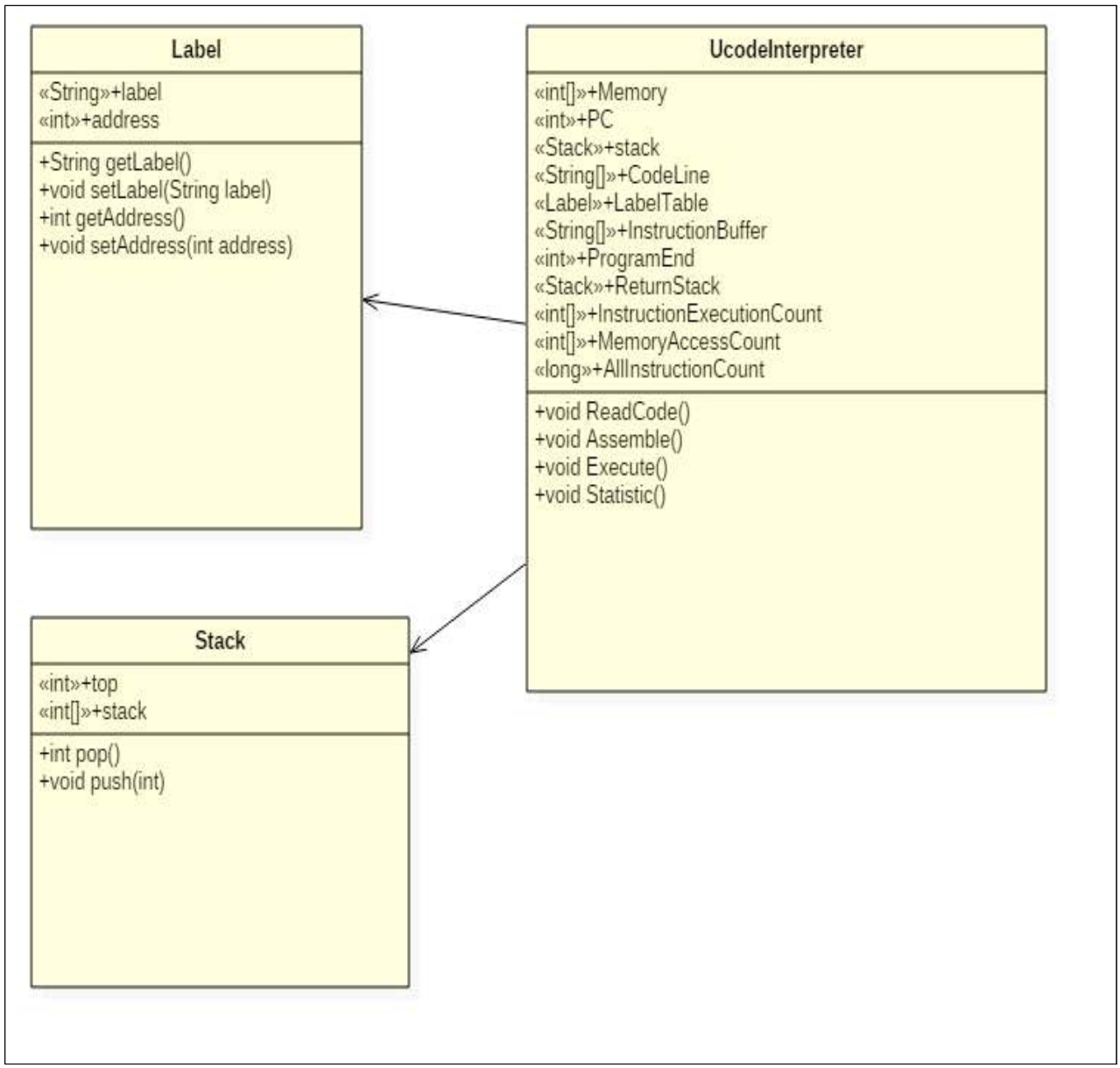
```
라인      레이블       명령어
---------------------------------
 1                    bgn 0
 2                    ldp
 3                    call main
 4                    end
 5       main         proc 0 12


.... 생략 .....


---------------------------------
           *실행 순서*
```

```
------------------------------
라인     레이블      명령어
------------------------------
 1                  bgn 0
 2                  ldp
 3                  call main
 4                  end
 5        main      proc 0 12


.... 생략 .....



------------------------------
          *실행 결과*
51
------------------------------
명령어별 사용횟수

  nop  3          bgn 1          sym 3          lod 10         lda 3
  ldc 5           str 7          ldi 1          sti 1          not 0
  neg 0           inc 1          dec 0          dup 1          add 3
  sub 0           mult 0         div 0          mod 1          gt 0
  lt 1            ge 0           le 0           eq 1           ne 0
  and 0           or 0           swp 0          tjp 0          fjp 2
  call 4          ret 1          retv 0         ldp 4          proc 2
  end 3


--------------------------------
명령어 메모리 접근 횟수 : 27
--------------------------------
사용한 명령어 수 : 56
--------------------------------
실행한 명령어 수 : 1084
--------------------------------
```

8 데이터 구조 설계

데이터 구조	설명
int Memory [1000]	데이터를 저장하는데 사용할 메모리
int PC	다음 명령어를 실행할 주소를 가리키는 프로그램 카운터 (Program Counter)
Stack stack	연산을 위해 사용할 임시기억장치(Stack)
String CodeLine[1000]	처음 *.uco파일로부터 코드를 읽어올 때, 저장할 코드를 줄 단위로 저장할 배열
String InstructionBuffer [1000]	주석을 제외한 명령어를 저장할 배열
int ProgramEnd	프로그램의 끝을 나타내는 주소 (main 프로시저 호출이 끝난후의 end 명령어의 주소에 해당함)
Stack ReturnStack	함수를 호출할 때,복귀할 주소를 저장할 Stack
int InstructionExecutionCount [37]	각 명령어가 실행된 횟수를 저장할 배열
int MemoryAccessCount[5]	메모리에 접근하는 명령어의 경우 메모리 접근횟수를 카운트시켜서, 저장할 배열
long AllInstructionCount	전체 명령어 실행횟수를 저장할 변수

9 클래스 다이어그램



클래스다이어그램 설명

Label클래스의 경우엔, 라벨만 저장할 것이아니라, Assemble단계에서 Label의 위치 즉, Label의 주소값이 필요하므로, 클래스로 선언하였습니다.Stack의 경우엔, 문제에서 정의한 U-CODE Interpreter는 임시기억장치로 스택을 사용하기 때문에, 스택을 정의하였습니다.UcodeInterpreter클래스에서는 해당 유스케이스에서 정의한 4가지 연산을 수행하는 프로그램의 핵심 클래스라고할수 있습니다.

10 주요 클래스 명세서

■Label 클래스

개요

Label의 값과 Label의 위치를 나타내는 클래스로써, Assemble 단계에서, 레이블별로 해당 레이블과 주소를 나타낸다.

책임과 역할

- Assemble 단계에서 레이블의 위치 저장

제약 조건

- 레이블의 길이는 12글자를 넘지 않아야한다.

주요 메소드

메소드명	반환형	범위	매개변수	설명
getLabel	String	public	void	해당 레이블의 값을 반환한다.
setLabel	void	public	String	해당 레이블의 값을 setting 한다.
getAddress	int	public	void	해당레이블의 주소값을 반환한다.
setAddress	void	public	int	해당 레이블의 주소를 setting 한다.

■UcodeInterpreter 클래스

개요

프로그램의 핵심클래스로써, 사용자로부터 입력받은 *.uco 파일을 읽어와서 명령어를 해석하여 프로그램의 실행결과와 통계파일(*.lst)를 출력하는 역할을 수행한다.

책임과 역할

- ReadCode 단계에서 코드를 줄단위로 읽어온다.
- Assemble 단계에서 코드를 주석문을 제외한 명령어만 읽어오고, 레이블의 주소를 지정하고,각 메모리 블록의 사이즈를 할당한다.
- Execute 단계에서 읽어온 명령어를 하나하나 해석하면서, 프로그램의 실행결과를 출력한다.
- Statistic 단계에서는 명령어를 읽으면서 저장했던 통계파일들을 보기좋은 형식으로 정렬하여, *.lst 파일에 저장하고 출력하는 역할을 수행한다.

제약 조건

- 반드시 *.uco파일을 입력해야한다.
- 문제에서 정의된 의외의 명령어는 허용하지않는다.

주요 메소드

메소드명	반환형	범위	매개변수	설명
ReadCode	void	public	void	사용자로부터 입력받은 *.uco 파일을 읽어와서 코드라인 단위로 CodeLine 배열에 저장한다.
Assemble	void	public	void	CodeLine에 저장된 줄단위의 명령어를 InstructionBuffer 배열에 주석문을 제외한 명령어를 저장하고,레이블의 위치주소를 저장한다.또한, 프로그램에서 사용할 메모립블록의주소를 할당한다.
Execute	void	public	void	InstructionBuffer를 공백단위로 나눠서 명령어를 하나씩 읽으면서 프로그램을 실행한다.이 과정에서, 통계파일에 필요한 변수들을 카운트한다.
Statistic	void	public	void	Execute에서 저장했던 통계파일에 필요한 변수들을 보기좋게 정렬하여 *.lst 파일로 출력한다.

11 주요 알고리즘

■ReadCode 알고리즘

```
Function ReadCode

1.파일 입출력을 이용해서, 코드를 줄 단위로 읽어온다.
2.읽어온 코드를 줄단위로  CodeLine 배열에 하나씩 저장한다.

end Function
```

■Assemble 알고리즘

```
Function Assemble
1. 레이블 읽기
라인 = 0
while( CodeLine 배열의 끝에 도달할때까지 )
  (1)문자열 인덱스를 기준으로 0~ 10번째 인덱스까지 문자열을 추출한다.
  (2)추출한 문자열의 공백을 제거한다.
  if( 추출한 문자열이 공백이 아니라면 )
      LabelTable에 해당 Label의 문자열과 라인(주소)값을 저장한다.
  end if
라인++
end while

2.명령어 읽기
while( CodeLine 배열의 끝에 도달할때까지 )
    (1)문자열 인덱스 기준으로 11번째 인덱스부터 주석이 나오기 전까지 문
     자열를 추출한다.
    (2)추출한 문자열를 InstructionBuffer에 저장한다.
end while
```

```
3.메모리 블록 할당
while( InstructionBuffer 배열에 끝에 도달할 때 까지 )
   (1)명령어 부분만 문자열을 추출한다.
   switch(명령어)
        case "메모리 할당 명령어" :
        메모리 블록 시작주소 할당
        break
         .....
   end switch
end while
end Function
```

■Execute 알고리즘

```
Function Execute
   PC = 0 %프로그램카운터(Program Counter 0으로 초기화)

   while( PC가 프로그램의 마지막번지를 가리킬때까지 )
      (1)명령어를 InstructionBuffer[PC] 로부터 읽어온다.
      (2)명령어 부분의 문자열만을 추출한다.

      switch(명령어)
         case "nop" :
         ......
         break
         case "ldi" :
         break
         ......

         default :
              Print(에러 : 처리할 수 없는 명령어입니다!)
              프로그램 강제종료
      end switch
```

```
    end while
end Function
```

■Statistic 알고리즘

```
Function Statistic
(1)파일 출력스트림을 연다.
(2) Output-------------사용한 U-CODE-------------------------)
(3) Output( Label        Line        Instruction      Operand )
(4) OutputInstruction 배열에 들어있는 문자열를 줄단위로 출력)
(5) Ouput(-------------------명령어 실행횟수 통계--------------------)
(6) Output(명령어의 전체 실행 횟수 : AllInstructionCount)
(7) Output-------------------------------------)
(8) Output(메모리 접근 횟수 : MemoryAccessCount배열의 합)
(9) Ouput(MemoryAccessCount배열의 각 요소의 횟수 출력)
(10) Output-------------------------------------)
(11) Output(InstructionExecutionCount배열의 각 요소의 횟수 출력)
End Statistic
```