소프트웨어 프로젝트 계획서

팀 명	UCI		
작 품 명	mU-Code Interpreter		
개발기간	2025년 10월 27일 ~ 2025년 12월 03일		
지도교수	컴퓨터공학전공		000
구분	학년	학번	성명
책임자(팀장)	2	2022----	이강석
팀원	2	2024----	권기영

소프트웨어프로젝트 과목에서 수행하는 프로젝트 계획서를 첨부와 같이 제출합니다.

2025년 11월 09일

국립금오공과대학교 컴퓨터공학부 컴퓨터공학전공

프로젝트 계획서

mU-Code Interpreter

목차

1. 서론

이 프로젝트는 Pascal 언어의 중간 코드로 설계된 P-Code를 일반화하고 간략화한 mU-Code를 실행할 수 있는 인터프리터와 가상 머신 구조를 설계 및 구현하는 것이다. 기본적으로 mU-Code를 해석해 실행하는 인터프리터를 구현하고, 파일 기반의 mU-Code 입력, mU-Code의 실행, 실행 결과 및 통계, 단계별 실행 과정, 실행 단계에서의 오류 감지 기능 그리고 GUI 환경 등을 모두 제공

하는 프로그램을 구현하는 것을 이 프로젝트의 목표로 한다.

중간 언어는 고급 언어에서 기계어로 변환하는 컴파일 과정의 중간 단계로, 다양한 고급 언어와 다양한 기계어 사이를 연결하는 역할을 한다. 서로 다른 언어로 작성된 프로그램이라도 같은 중간 언어로 바꾸는 과정을 이용하면 여러 CPU에서 명령어를 처리하는 방법이 다르더라도 같은 중간 언어를 사용하기 때문에 문제 없이 실행할 수 있다. 즉, 언어-중간언어 / 중간언어-기계어 번역 과정을 분리할 수 있다.

인터프리터는 명령어 인출 -> 의미 분석 -> 실행을 반복하는 프로그램이다. 전체 코드를 모두 기계어로 변환 후 실행하는 컴파일러와 달리 명령어를 읽는 즉시 실행하기 때문에 비교적 빠른 실행 준비 시간과 쉬운 디버깅이라는 장점이 있다. 그러나 한 줄씩 실행하는 것은 비교적 속도가 느리고 최적화가 어렵기 때문에 대표적인 인터프리터 언어인 Python과 Java는 각각의 가상 머신 구조에 맞게 구현된 바이트코드라고 부르는 중간 언어로 컴파일 후 바이트코드를 한줄씩 읽어 인터프리터로 대응되는 기계어 코드를 실행하는 방법을 사용한다. 프로젝트의 mU-Code도 이런 바이트코드의 일종으로 볼 수 있다.[i] [ii] [iii]

가상 머신은 컴퓨터의 CPU, RAM, 하드디스크 등의 주요 부품들을 소프트웨어적으로 구현하여 사용하는 것이다. 인터프리터에서도 프로그램을 실행하기 위해서 가상 머신 구조를 사용하기도 한다. JVM이 대표적인 인터프리터를 위한 가상 머신이다. 가상 머신은 맞는 바이트코드만 있으면 프로그램을 실행할 수 있기 때문에 고급 언어의 종류는 중요하지 않고 해당 언어가 가상 머신에 맞는 바이트코드로 전환될 수 있는지가 중요하다. kotlin 언어가 자바 바이트코드로 번역되어 JVM에서 실행된다. 이런식으로 고급 언어에서 중간 언어로 바꿔주는 컴파일러와 CPU에 맞는 가상 머신만 있으면, 어떤 언어를 쓰는지 어떤 CPU를 쓰는지 상관없이 실행되기 때문에 이식성과 유연함이 좋다는 장점이 있다. 이 프로젝트에서도 mU-Code를 실행하기 위해서 mU-Code에 맞는 가상 머신 구조를 사용한다.[iv] [v]

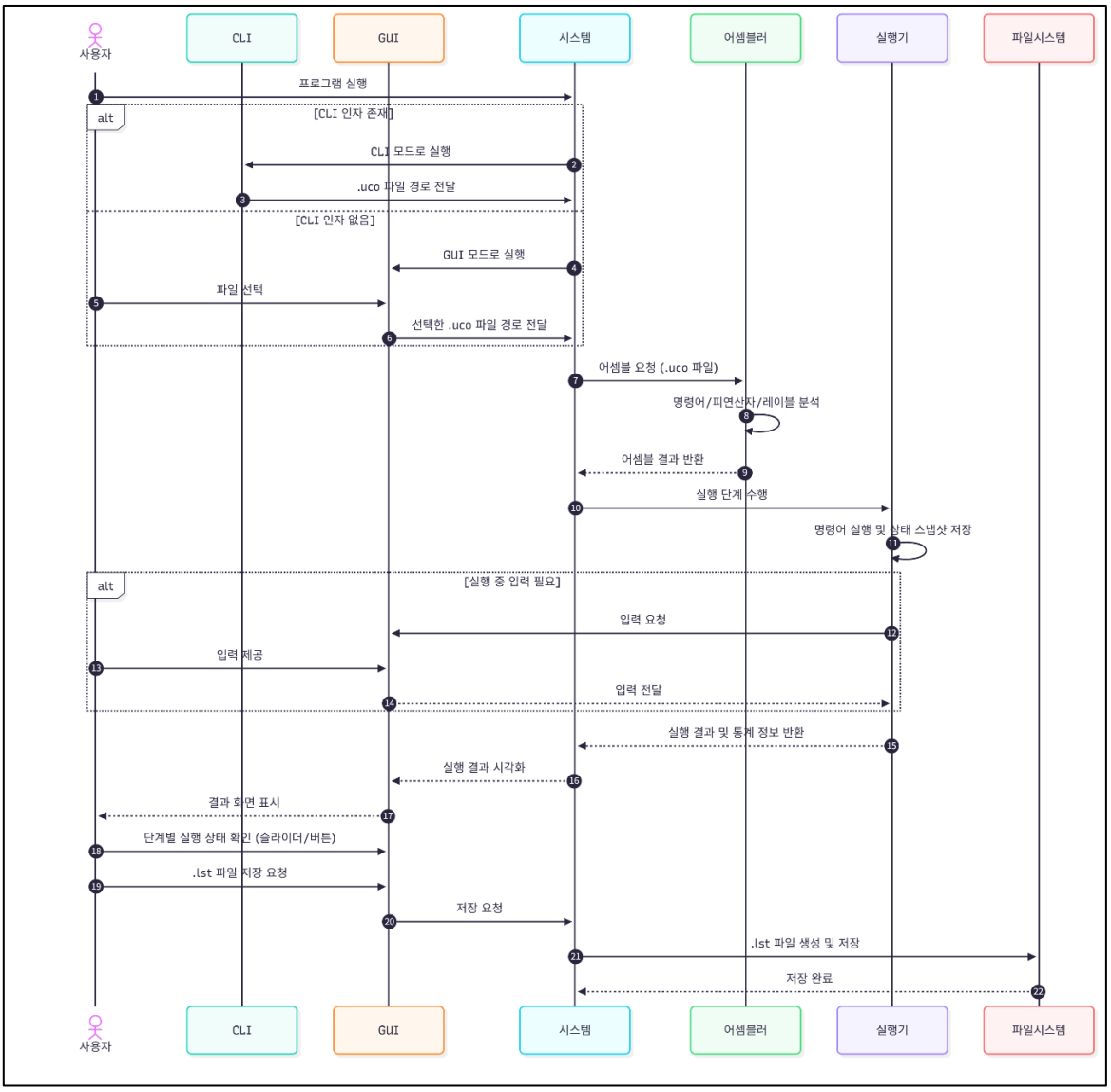
mU-Code 인터프리터를 구현하며 프로그램의 실행 과정과 가상 머신 구조를 이해하는 것을 목표로 한다. 기존 고급 언어에서는 디버깅 등의 기능으로 실행 과정을 확인할 수 있지만 Low-level에서 명령어 단위로 컴퓨터가 어떻게 동작하는지는 확인하기는 어렵다. 이 프로젝트에서는 중간 언어를 직접 실행하고 분석하면서 이런 Low-level에서의 컴퓨터 동작을 보여준다. 추후 프로젝트를 확장하여 C나 C++등의 언어를 mU-Code로 변환해주는 기능을 추가하고 mU-Code에도 명령어를 몇 가지 추가하여 다양한 언어를 실행할 수 있을 것이다. mU-Code 인터프리터를 구현하는 것은 컴퓨터 구조와 프로그램 실행에 대한 이해도를 높이고 추후 확장한다면 새로운 나만의 프로그래밍 언어를 한번 만들어 볼 수도 있을 것으로 기대된다.

2. 개발 내용

2.1 사용 시나리오

본 프로그램은 사용자가 .uco 파일을 입력하면 어셈블 및 실행 과정을 거쳐 .lst 파일을 출력하는 시스템이다. 아래는 대표 시나리오와 대체 시나리오를 통해 전체 입출력 흐름을 설명한다.

2.1.1 대표 성공 시나리오

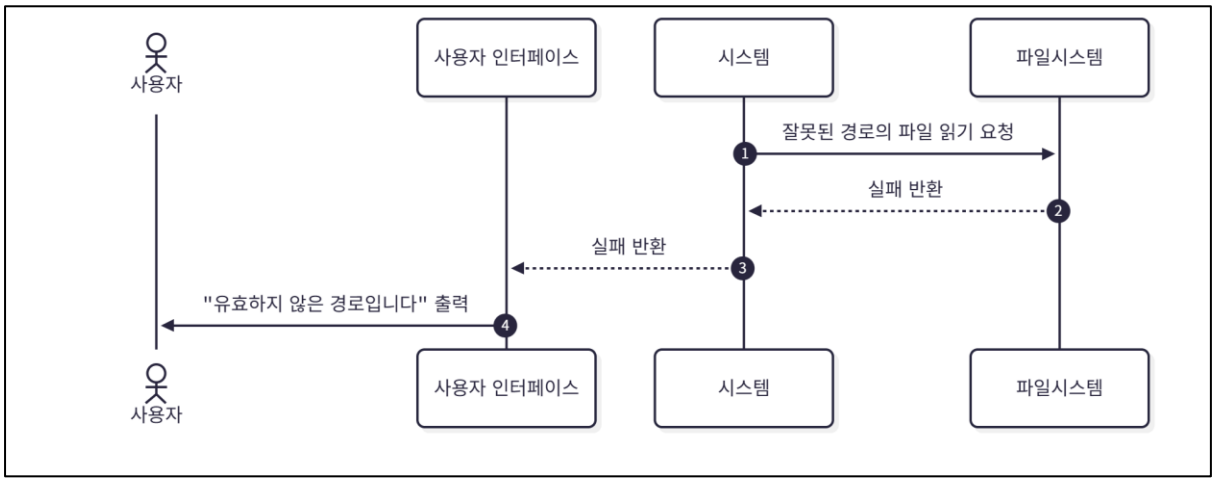


<그림 1 대표 성공 시나리오>

사용자는 프로그램을 실행하면 CLI 인자가 있으면 CLI 모드로 없으면 GUI 모드로 진입한다. GUI 에서는 사용자가 파일을 선택하고 선택된 .uco 파일 경로가 시스템으로 전달된다. 시스템은 어셈블러를 호출하여 명령어와 레이블을 분석하고, 실행기에서 명령어를 수행하며 단계별 상태를 스

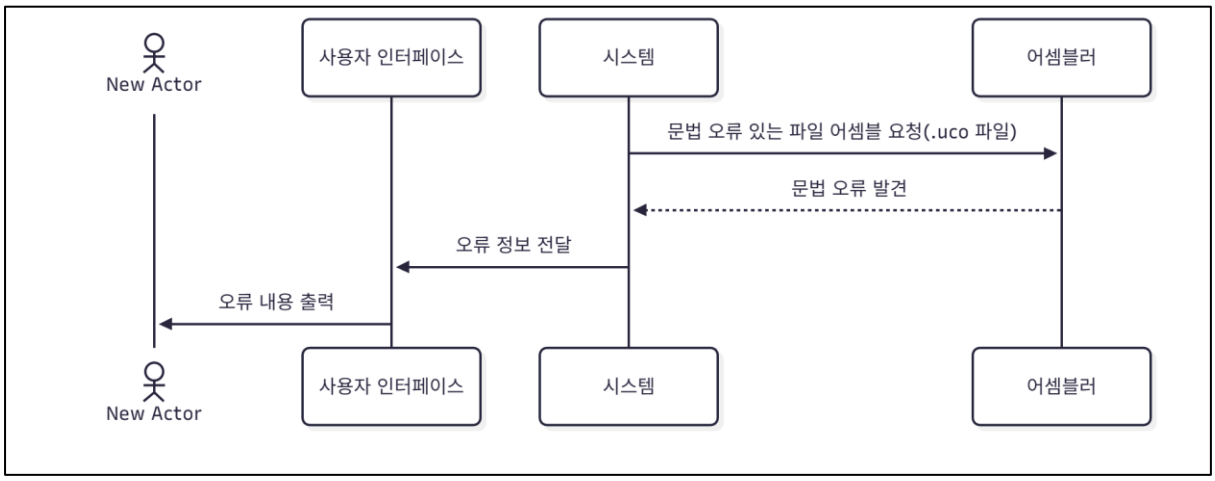
냅샷 형태로 저장한다. 실행 중 사용자 입력이 필요한 경우 GUI를 통해 입력받고, 실행 결과와 통계 정보가 GUI에 시각화되어 표시된다. 사용자는 슬라이더나 버튼을 이용해 특정 단계 상태를 확인할 수 있으며, .lst 파일 저장은 GUI에서 사용자 요청으로 이루어지고, 시스템이 파일시스템에 저장한다.

2.1.2 대체 시나리오



<그림 2 경로 오류 시나리오>

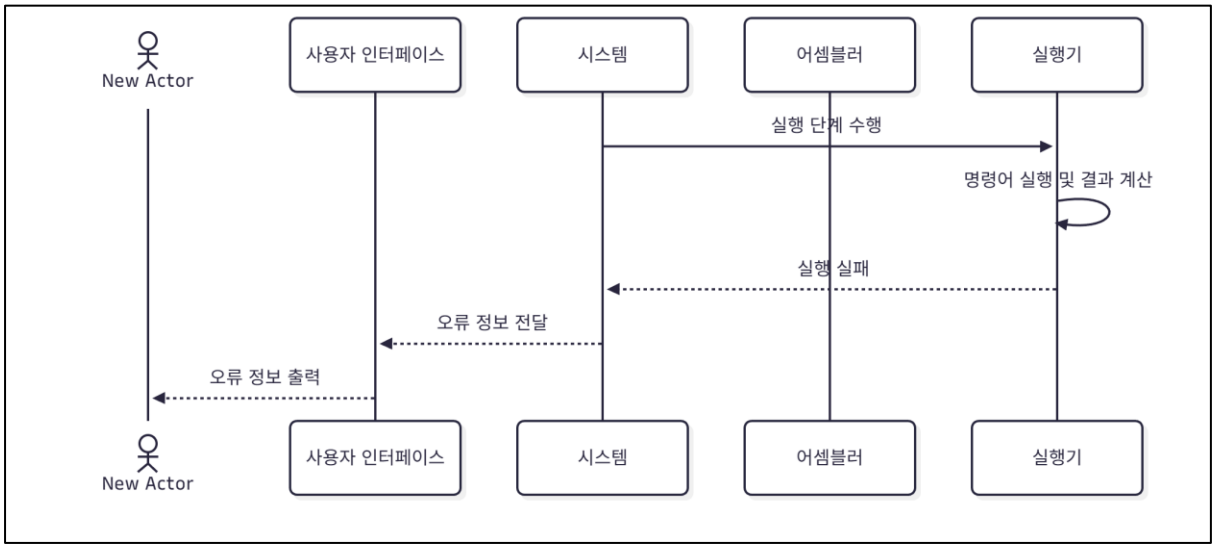
그림은 사용자가 코드 실행을 요청했지만 사용자가 제공한 경로에 파일이 존재하지 않는 대체 시나리오이다. 사용자가 경로를 전달하면 파일시스템에서 경로를 확인하고 만약 파일이 존재하지 않는다면 실패 처리를 하고 시스템에 알려준다. 이후 인터페이스를 통해 사용자에게 파일이 없다는 정보를 표시해준다



<그림 3 문법 오류 시나리오>

그림은 어셈블러로 mU-Code 파일은 정상적으로 넘겨졌으나, mU-Code에 문법 오류가 있는 경

우이다. 어셈블러가 어셈블 과정에서 문법 오류를 발견하고, 다시 시스템으로 문법 오류를 발견했다는 정보를 알린다. 이후 사용자 인터페이스를 통해서 사용자에게 어디서 오류가 발생했는지 내용을 출력해준다.



<그림 4 실행 오류 시나리오>

그림은 어셈블 단계가 성공적으로 끝나고 실행 중 스택 오버플로우 등의 오류가 발생한 경우이다. 실행기가 실행 과정 중 오류를 발견하고 오류에 대한 정보를 기록하고 실행기가 종료된다. 이후 인터페이스로 오류 정보가 전달되면 사용자에게 오류 정보가 출력되게 된다.

2.2 요구 사항 분석

2.2.1 기능 요구 사항

[사용자 기능 요구사항]

1. .uco 파일 입력
   - 사용자는 CLI 인자로 .uco 파일을 입력하거나 GUI에서 "파일 열기" 기능을 통해 선택할 수 있음
   - 입력: .uco 파일 경로
   - 출력: 선택된 파일 내용이 프로그램 내부에 로드됨
   - 대체 흐름: 잘못된 확장자나 존재하지 않는 경로 입력 시 오류 메시지 출력 후 재입력 요청
2. 코드 실행 및 결과 확인
   - 사용자는 "실행" 버튼을 눌러 전체 코드를 즉시 실행시키며 실행 중 입력이 필요할 경우 CLI 또는 GUI 입력 창을 통해 값을 받음
   - 입력: 실행 명령, 필요시 사용자 입력값
   - 출력: 프로그램 실행 결과(표준 출력 및 상태 정보)

- 대체 흐름: 실행 중 오류 발생 시 오류 메시지 창 표시, 실행 중단

3. 단계별 실행 결과 확인
   - 프로그램은 실행 완료 후 단계별 상태 변화를 기록하며 사용자는 해당 기록을 확인할 수 있음(실시간 갱신 아님)
   - 입력: 단계 조회 명령(맨 처음/이전/다음/맨 마지막, 임의 단계)
   - 출력: 선택한 단계의 실행 상태 정보
   - 대체 흐름: 실행 로그가 존재하지 않거나 손상되었으면 경고 메시지 출력
4. 통계 및 결과 저장
   - 실행 결과 및 통계 데이터를 사용자가 지정한 경로로 저장할 수 있음
   - 입력: 저장 경로 및 파일명
   - 출력: .lst 파일 생성
   - 대체 흐름: 저장 실패 시 오류 메시지 출력

[시스템 기능 요구사항]

1. 어셈블 처리
   - .uco 파일을 구문분석하여 opcode 및 피연산자 분석, 명령어 테이블 생성 후 실행 코드 영역 구성
   - 입력: .uco 파일
   - 출력: 내부 code 영역에 저장되는 명령어 및 심볼/label 테이블
   - 대체 흐름: 어셈블 오류 발생 시 해당 라인 번호와 원인 로그 출력
2. 실행 엔진
   - 어셈블 결과를 기반으로 명령어를 순차 실행하고 메모리, 레지스터를 관리함
   - 입력: 내부 code 영역 명령어
   - 출력: 실행 결과 데이터(출력, 로그)
   - 대체 흐름: 잘못된 명령어나 메모리 접근 시 오류 정보 출력 후 종료
3. 오류 처리 및 예외 처리
   - 파일 오류, 구문분석 실패, 실행 중 예외 등 다양한 오류를 감지하고 사용자에게 알림을 전달함
   - 입력: 오류 코드 및 예외 정보
   - 출력: 오류 메시지, 로그 기록
   - 대체 흐름: 오류 발생 시 GUI 경고 창 실행
4. GUI 이벤트 처리
   - GTK 신호 기반으로 버튼 클릭, 파일 선택, 종료 이벤트 등을 처리함
   - 입력: 사용자 상호작용 이벤트
   - 출력: 해당 기능 수행 결과

2.2.2 비기능 요구 사항

항목	요구 사항 내용
성능	5개 예제 프로그램 기준 실행 1초 이내
효율성	명령어 주소 인코딩 및 GUI 실행 흐름 최적화로 메모리 사용 효율화
규격/표준	.uco 파일의 코드 형식 준수 (1~10열: 레이블, 11열: 공백, 12열 이후: 명령어/피연산자)
작동 환경	Ubuntu
개발 환경	C언어, GTK 3.0, GCC 컴파일러
신뢰성	어셈블 및 실행 오류 시 GUI/CLI 모두에서 로그 출력
안전성	잘못된 입력에도 시스템 종료 없이 오류 메시지 출력 후 정상 종료
유지보수성	어셈블러, 실행기, GUI 모듈을 사용하여 분리 구조로 관리
데이터 타입 제한	정수형만 지원
주소 지정 방식	연산 명령: 스택 기반(0주소) 상수/메모리 접근/분기 명령: 피연산자 1개 사용(1주소)
주석 규칙	%로 시작하는 행은 주석으로 인식

3. 설계

3.1 개요

3.1.1 mU-Code 명령어 및 상세 명세[vi]

1. 데이터는 정수형만 있는 것으로 간주한다.
2. CPU내 AC, MBR 등의 레지스터가 없고, 대신 모든 연산이 스택을 기준으로 이루어진다. 따라서 ALU 동작에 대한 명령은 피연산자 없이 스택 꼭대기의 값을 가져와 연산 후 다시 스택 꼭대기에 올려놓는 것을 가정한다(0-addressing). 상수나 메모리에 저장된 값을 가져오거나 반대로 보내는 경우 및 분기 명령어의 경우에는 피연산자가 하나 지정된다(1-addressing).
3. mU-Code 프로그램은 정해진 형식을 지켜서 작성해야한다. 1~10열까지는 레이블, 11번째는 공백, 12열부터 명령어가 나타날 수 있다. 명령어와 피연산자 사이에는 하나 이상의 공백이 있어야 한다.

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
m	a	i	n								p	r	o	c		1	2		
											s	y	m		2		0		1
											l	d	p						

4. %로 시작하는 부분부터 그 행의 끝까지는 주석문으로 인식해 아무 작업도 하지 않는다.

mU-Code 명령어

1. 프로그램 구성 명령

프로그램 구성 명령은 실제로 실행되는 명령이 아니라 실행 전 필요한 요소를 설정하거나, 인간이 프로그램을 이해하는데 필요한 명령들이다. 따라서 opcode가 할당되지 않는다.

명령어	동작
nop	아무 작업도 수행하지 않음 레이블 위치에 주로 사용
sym b n s	변수가 속한 블록(b) 오프셋(n) 크기(s)를 표현한다. - 전역변수의 블록번호는 1 - 지역변수의 블록번호는 2 이상(함수마다 별도) - 오프셋은 0부터 시작 - 일반 정수형은 크기가 1 - 배열형은 배열의 크기만큼
bgn n	프로그램의 시작점 n은 전역 변수의 총량(크기)
end	프로그램의 끝

2. 입출력 처리

mU-Code에서는 입출력에 대한 간단한 시스템 함수를 가정하고 사용할 수 있도록 한다.

명령어	동작
read(i)	외부 입력값을 읽어 CPU 스택 꼭대기에 저장된 주소에 저장한다.
write(i)	CPU 스택 꼭대기의 값을 출력한다. 출력된 값 뒤에 공백을 추가로 출력한다.
lf()	줄바꿈 문자를 출력한다.

3. 함수 정의 및 호출

명령어	동작
proc n	함수의 시작을 나타내며, n은 함수가 사용하는 모든 지역변수의 총량(크기).
ret	함수를 종료하고 복귀. 반환값이 있을땐 ret 전 반환값을 CPU스택에 저장해야 한다.
ldp	함수를 호출하기 전 함수가 사용할 메모리 영역을 설정하고 매개변수 전달을 준비한다.
push	CPU 스택에 올려진 실인자 값을 메모리 스택에 저장한다.
call label	label로 지정된 함수를 호출한다.

4. 흐름 제어

명령어	동작
ujp label	지정한 label로 무조건 이동
tjp label	CPU스택 꼭대기에 저장된 값이 참이면 label로 이동
fjp label	CPU스택 꼭대기에 저장된 값이 거짓이면 label로 이동

5. 데이터 이동 연산자

명령어	동작
lod b n	b 블록 n 오프셋 주소의 데이터를 CPU 스택에 넣는다.
lda b n	b 블록 n 오프셋으로 계산되는 메모리 주소를 CPU 스택에 넣는다.
ldc c	상수값 c를 CPU 스택에 넣는다.

str b n	CPU 스택 꼭대기의 값을 꺼내 b 블록 n 오프셋 주소에 저장한다.
ldi	간접 주소법을 이용해 메모리의 값을 CPU 스택으로 가져온다. CPU 스택 꼭대기의 값을 주소값으로 이용해 해당 주소의 데이터를 가져온다.
sti	간접 주소법을 이용해 CPU스택의 값을 메모리에 저장한다. CPU 스택 꼭대기의 값을 꺼내 주소값으로 이용하고 또 다른 값을 꺼내 저장한 데이터로 사용한다.

6. 이항 연산자

명령어	의미	동작
add sub mult div mod	add subtract multiply divide modulo	stack[top-1] = stack[top-1] + stack[top]; top-- (순서대로 -, *, /, % 등으로 연산자만 바뀌며 동작은 동일)
gt lt ge le eq ne and or	greater than less than gt or equal lt or equal equal not equal and or	stack[top-1] = stack[top-1] > stack[top]; top-- (순서대로 =, <=, ==, !=, &&, || 등으로 연산자만 바뀌며 동작은 동일)

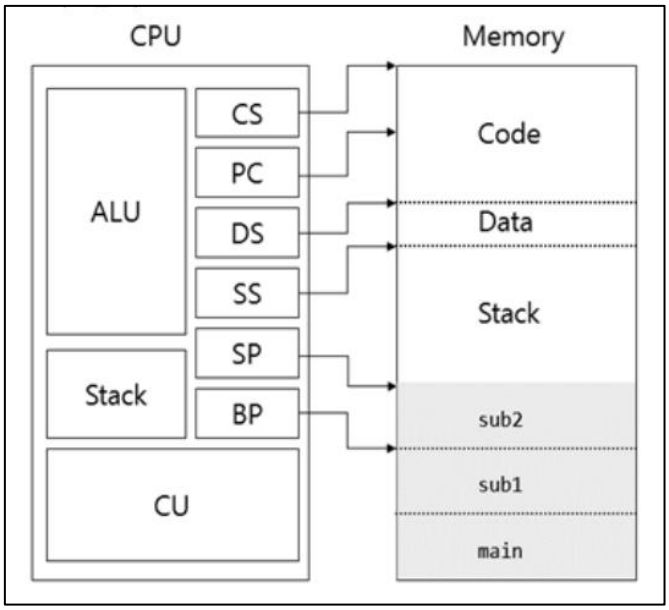
7. 단항 연산자

명령어	동작
not	stack[top] = !stack[top]
neg	stack[top] = -stack[top]

mU-Code Interpreter는 가상 머신 구조를 사용한다.

3.1.2 가상 머신 구조

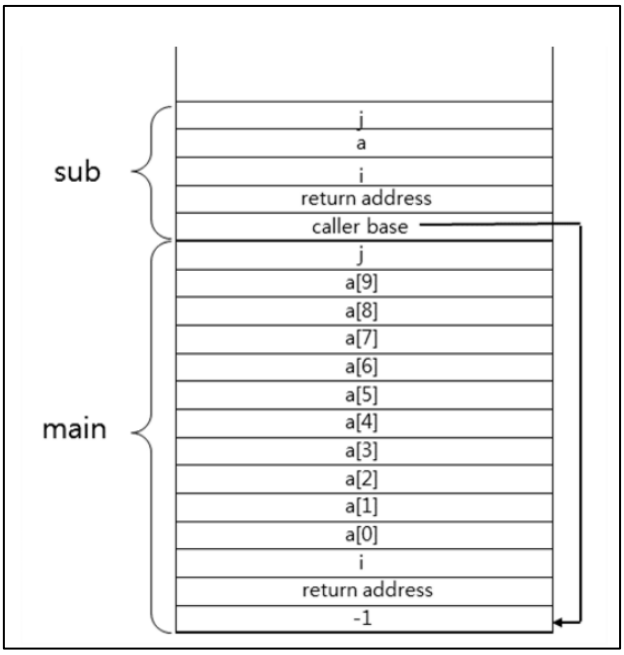
mU-Code 인터프리터를 만들기 위해서 가상 머신을 사용한다. 가상 머신의 구조는 다음과 같다.



<그림 5 가상 머신 구조>

CPU는 ALU와 CU의 기능을 수행하며, 임시 저장 장치로 범용 레지스터 대신 하나의 stack을 이용한다. 이를 CPU 스택이라 한다. 메모리는 1차원 배열로 정의되며 mU-Code의 모든 데이터는 단일 크기의 정수형이므로 메모리 주소는 정수 하나를 저장할 수 있는 단위로 한다.

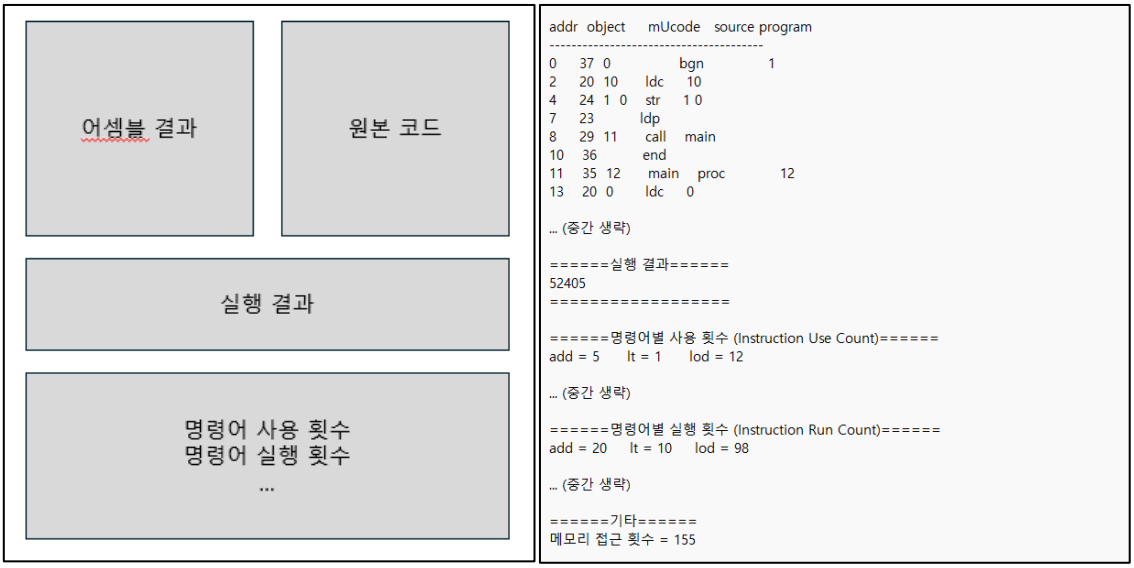
메모리의 Code, Data, Stack 영역은 각각 프로그램, 전역변수, 지역변수를 저장한다. 또한 CPU에는 CS, DS, SS, PC, SP, BP 레지스터들이 있는데 각각 Code, Data, Stack의 시작 주소, 실행중인 명령어의 주소, 현재 스택 꼭대기의 주소, 현재 프레임의 시작 주소를 저장하고 있다.



<그림 6 메모리 스택>

Stack 영역에서는 함수가 호출될 때마다 해당 함수에서 사용할 변수의 크기와 순서대로 스택에 쌓이게 된다. 또, 함수가 사용하는 영역을 프레임이라 하는데 프레임에는 지역변수만 저장되는 것이 아니라. 함수가 종료될 때 복귀 프로그램 명령어의 주소, 자신을 호출한 함수 프레임의 위치 등이 저장된다. return address는 복귀한 명령어 주소, caller base는 자신을 호출한 함수 프레임의 시작 주소이다.

3.1.3 lst 파일

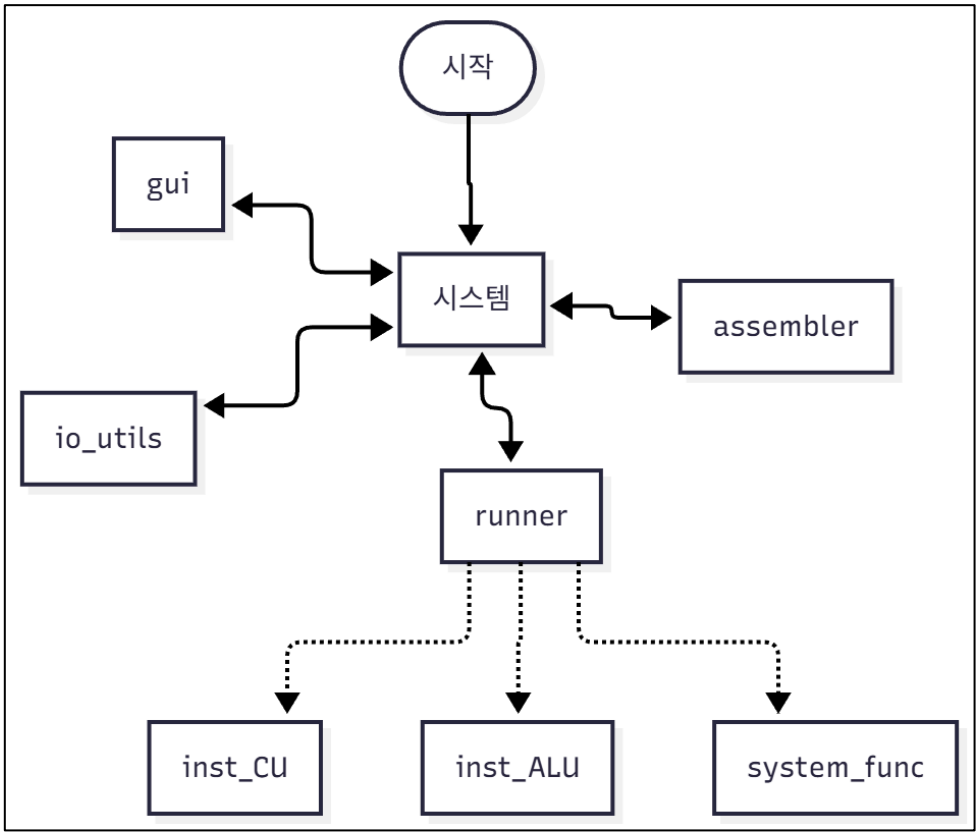


<그림 7 lst 파일 예시>

.lst 파일은 mU-Code Interpreter가 종료되는 시점에 생성되며, 위쪽부터 차례대로 원본 코드와 어셈블 결과, 입력된 프로그램의 실행 결과, 명령어 사용 횟수, 명령어 실행 횟수, 메모리 접근 횟수 등의 정보들이 아래로 들어있다. 이 파일을 통해서 사용자는 프로그램의 실행 통계를 확인할 수 있다.

오른쪽 그림은 lst파일의 예시이다. 위에서부터 어셈블 결과, 원본코드 실행결과, 명령어별 사용 횟수, 명령어별 실행 횟수, 메모리 접근 횟수 등의 정보가 표시된다.

3.2 프로그램 구조 설계



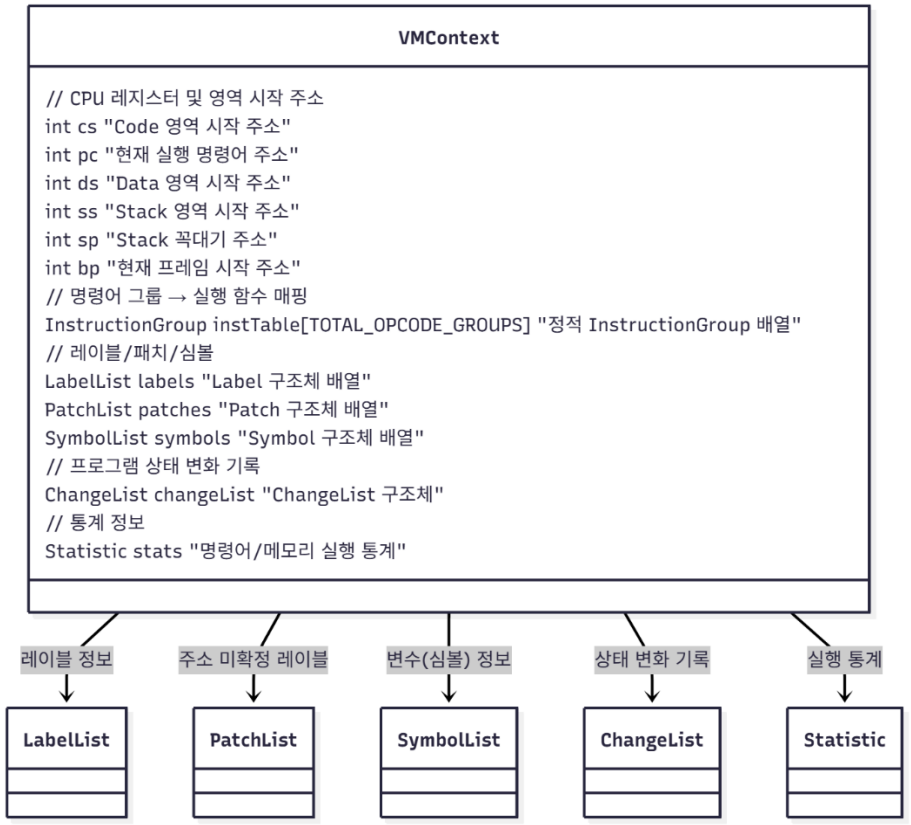
<그림 8 프로그램 실행 흐름도>

위 그림은 전체 모듈의 호출 관계로서 실선은 프로그램의 흐름, 점선은 모듈의 관계를 나타낸다. 프로그램이 시작되면 시스템에서 먼저 io_utils 모듈또는 GUI를 통해 .uco 파일을 입력받아 assembler로 전달해준다. 어셈블 단계가 완료되면 실행을 위해 runner 모듈을 호출한다. 호출된 runner는 명령어 함수들이 들어있는 inst_CU, inst_ALU, system_func등의 모듈을 호출하며 mU-Code를 한줄씩 실행한다.

각 모듈의 역할은 다음과 같다.

- 전체 모듈 및 역할
  - io_utils : *.uco 파일 입력 및 *.lst 파일 출력
  - assembler : 입력받은 mU-Code 어셈블
  - runner : 어셈블 된 명령어 실행
  - inst_ALU : 이항, 단항 연산 명령어 처리
  - inst_CU : 함수 호출 및 데이터 이동, 분기 명령어 처리
  - system_func : 시스템 함수 처리
  - gui : GUI 생성 및 이벤트 처리

3.3 데이터 구조 설계

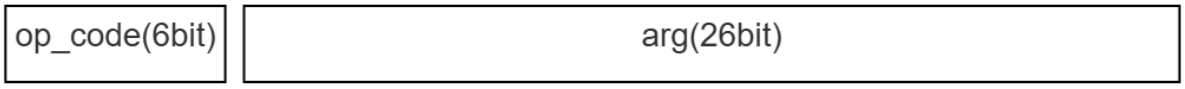


<그림 9 VMContext 구조>

상단의 그림은 가상 머신의 전체 구조를 나타낸다. VMContext는 프로그램 실행 전반을 관리하는 중앙 제어 구조체로 프로그램의 실행 상태를 일관되게 유지하기 위해 사용된다. 가상 머신은 코드 해석, 메모리 접근, 스택 연산, 변수 관리 등 다양한 모듈이 협력하여 동작하므로 이 모든 정보를 하나의 공통 컨텍스트(context)로 묶어 관리할 필요가 있다.

이 구조체는 code, data, stack 영역의 시작 주소와 현재 실행 중인 명령어 주소, 스택 포인터, 베이스 포인터와 같이 CPU 레지스터 실행 상태를 저장한다. 또한, 각 명령어 코드와 실행 함수를 연결하는 InstructionGroup 배열을 보유해 명령어 해석 및 실행 효율을 높인다.

어셈블 중 생성되는 데이터는 별도의 리스트 구조로 관리되는데 LabelList, PatchList, SymbolList는 각각 레이블, 미해결 주소, 심볼 정보를 저장하며 어셈블 및 실행 시 참조된다. 이 외에도 프로그램의 실행 변화 이력을 추적하기 위한 ChangeList, 실행 통계 수집을 위한 Statistic을 포함해 가상 머신의 상태를 완전하게 보관한다.

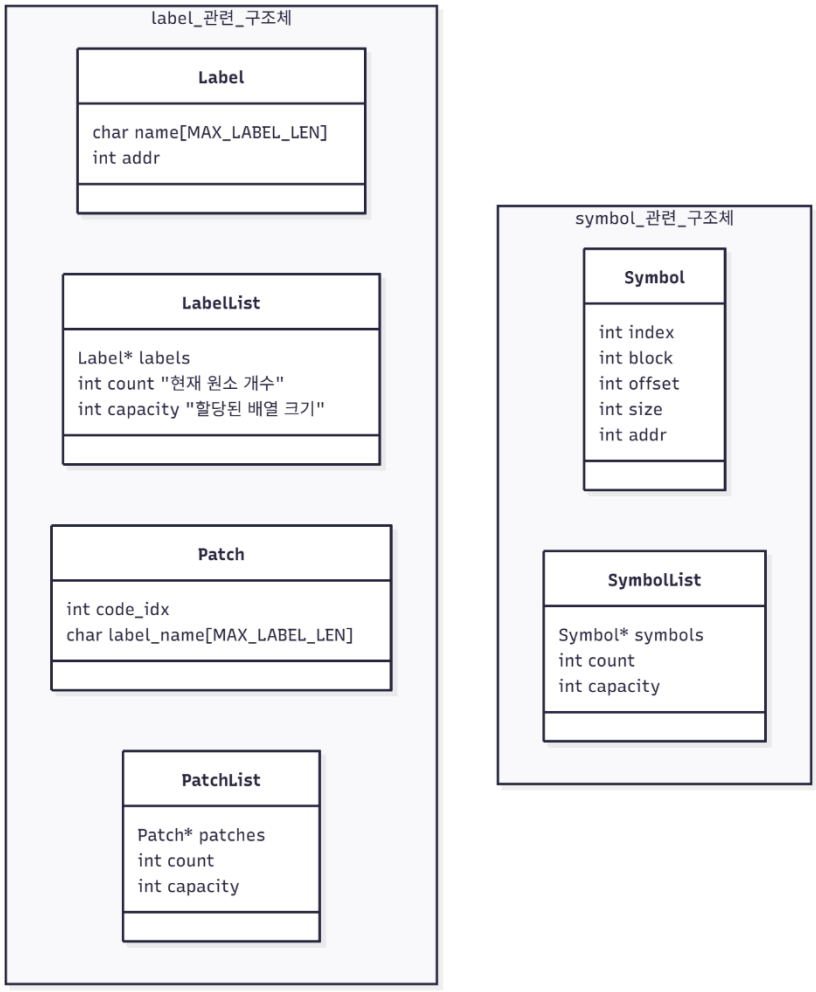


<그림 10 명령어 형식>[vii]

상단의 그림은 명령어 형식을 나타낸다. 명령어는 32비트 정수형(int)으로 저장되며, 상위 6비트는 opcode, 하위 26비트는 피연산자(operand)로 사용된다.

opcode의 상위 3비트는 명령어 그룹, 하위 3비트는 그룹 내 인덱스를 나타내며 이러한 구조는 명령어를 기능 단위로 분류할 수 있게 한다. 예를 들어 함수 정의 및 호출, 흐름 제어 등으로 그룹화하여 InstructionGroup 배열과 1:1로 매핑된다.

이 방식은 명령어 인코딩을 단순화하고 opcode 해석 시 비트 연산만으로 해당 함수 포인터를 빠르게 찾아 실행할 수 있게 한다. 또한 opcode를 enum으로 정의함으로써 코드 가독성과 유지보수성을 향상시킨다.



<그림 11 label 및 symbol 관련 자료구조>

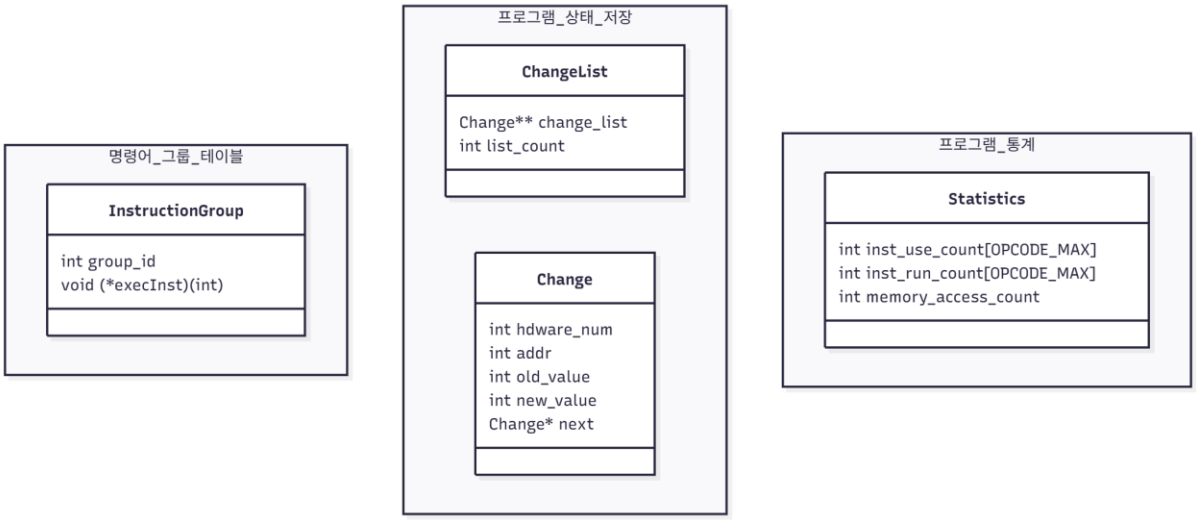
상단의 그림은 어셈블 및 실행 과정에서 사용되는 자료구조를 나타낸다. Label 구조체는 레이블 이름과 코드 내 인덱스(주소)를 필드로 가지며, 프로그램 시작 시 시스템 함수 레이블이 미리 등록되고, 이후 사용자 정의 레이블이 추가된다.

Patch 구조체는 실제 주소가 확정되지 않은 코드 인덱스를 저장하고, 나중에 참조해야 할 레이블

이름을 함께 기록한다. 어셈블이 종료되면 LabelList를 참조해 PatchList에 기록된 주소를 갱신한다.

Symbol 구조체는 변수의 인덱스, 블록 번호(block), 오프셋(offset), 크기(size) 정보를 가지며, 1주소 명령어 체계에 맞춰 변수 접근을 단순화한다. 지역 변수의 경우 베이스 포인터(bp)가 함수 호출마다 바뀌므로, bp가 갱신될 때마다 변수 주소를 재계산해야 한다.

이러한 구조체들을 LabelList, PatchList, SymbolList 형태의 동적 배열로 관리함으로써 실행 중 발생하는 레이블, 미확정 레이블, 심볼 정보를 유연하게 처리할 수 있다.



<그림 12 명령어, 변경 사항 및 통계 관련 자료구조>

InstructionGroup 구조체는 opcode 에 맞게 명령어 처리 함수로 매핑해주는 구조체이다. 함수 포인터를 필드로 가지고 있어 InstructionGroup 을 배열로 생성해 opcode 에 맞는 함수를 호출하는 식으로 사용된다. Change 와 ChangeList 는 프로그램의 상태 변화를 저장하기 위한 구조체이다. 먼저 ChangeList 는 Change** change_list 을 필드로 가지는데 이 필드는 cycle 마다 동적 할당되어 각 사이클의 Change 연결 리스트의 시작을 가리키는 포인터 배열이다. 또한 change_list 의 크기를 기록하기 위한 list_count 가 있다. Change 는 하나의 변경사항을 저장한다. 필드로는 어떤 하드웨어에서, 어떤 주소에, 이전/새로운 값, 다음 Change 주소를 가진다. Statistic 은 .lst 파일을 만들기 위한 통계 자료를 저장하는 함수이다. 명령어 사용/실행 횟수, 메모리 접근 횟수를 필드로 가진다.

3.4 인터페이스 설계

데이터 흐름

기능	사용 데이터	생성 데이터	입출력 방식
어셈블러	.uco 파일 내용	Code 영역(전역 배열) 명령어 사용 횟수 레이블/변수 테이블	io_utils 모듈 사용 전역 배열 사용

		레지스터 세트	
실행기	Code/Data/Stack(전역 배열) VMContext	명령어 실행 횟수 메모리 접근 횟수 프로그램 상태 스냅샷 및 변경 이력	VMContext 전달 전역 배열 참조
오류 검사	현재 접근 주소 레지스터 세트	상태 레지스터(에러)	VMContext 전달 전역 배열 참조
명령어 처리	변수 테이블 CPU 스택 Data/Stack 영역	실행 결과	VMContext 전달 전역 배열 참조
변경 내용 저장	CPU 스택 Data/Stack 영역 레지스터 세트	ChangeList 구조체 Change 구조체	VMContext 전달 전역 배열 참조
GUI 제어	ChangeList 구조체 Change 구조체	GUI	VMContext 전달

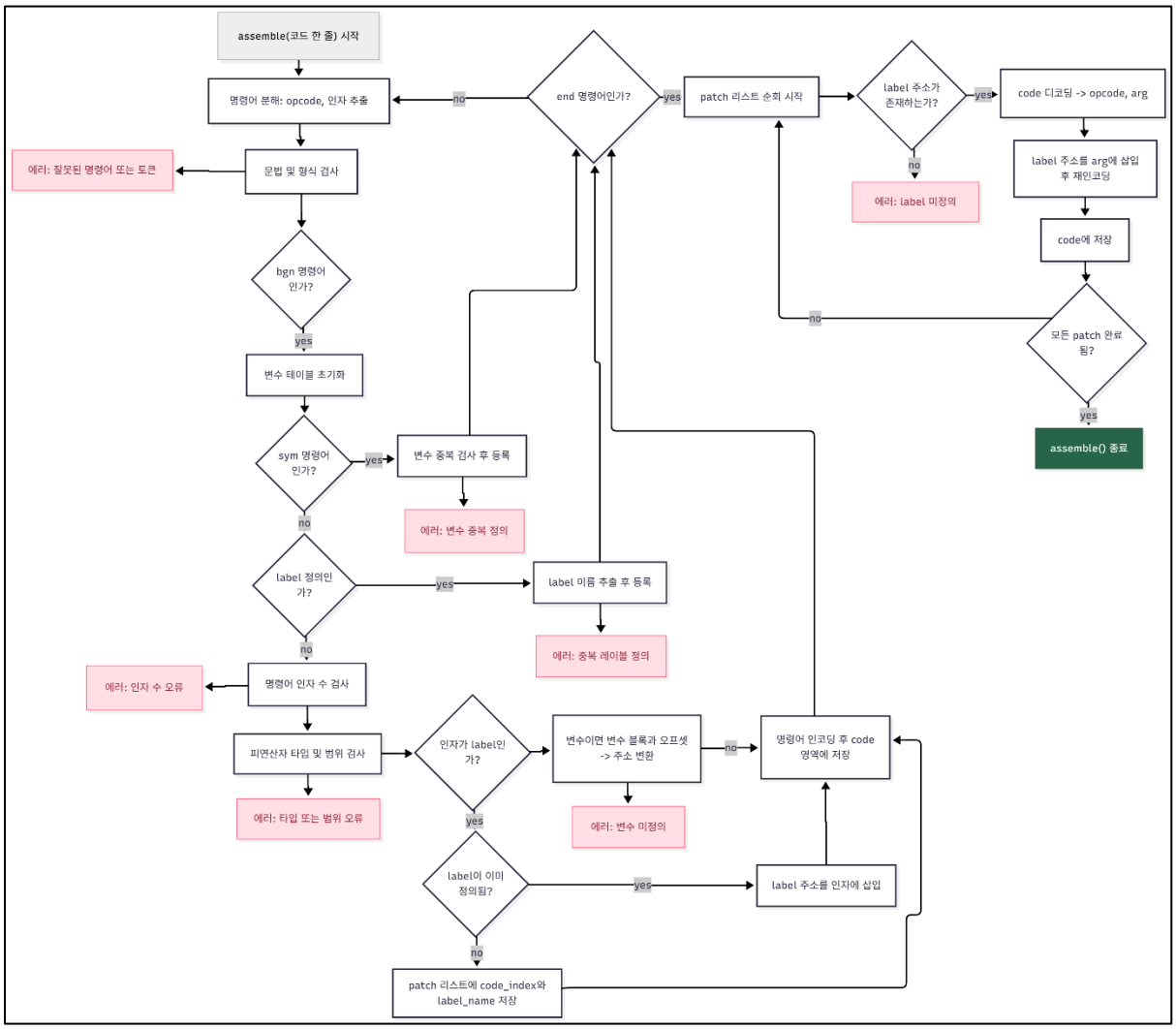
함수 시그니처

모듈	**함수 시그니처**	**인자/반환값**
io_utils	int loadUco(const char* path, char*** lines, int* line_count);	path: 파일 경로 lines: 줄 단위 문자열 포인터 배열 반환 line_count: 줄 수 반환 반환값: 성공 여부
	int saveLst(const char* path);	path: 저장할 파일 경로 반환값: 성공 여부
assembler	int assemble(char** lines, int line_count);	lines: loadUco 결과 line_count: 줄 수 반환값: 성공 여부
runner	int run();	VMContext 참조 반환값: 성공 여부
	int step();	전역 Code/Data/Stack 배열 참조 VMContext 참조 반환값: 성공 여부
inst_CU/ inst_ALU	void (*execInst)(int arg);	전역 Code/Data/Stack 배열 참조 VMContext 참조 반환값: 없음
system_func	void read();	전역 Code/Data/Stack 배열 참조 VMContext 참조

		반환값: 없음
	char* reqRead();	시스템 표준 입력 또는 GUI 위젯 참조 반환값: 입력받은 내용
	void write();	전역 Code/Data/Stack 배열 참조 VMContext 참조 반환값: 없음
	char* reqWrite();	시스템 표준 입력 또는 GUI 위젯 참조 반환값: 입력받은 내용
gui	void startGui();	VMContext 참조 반환값: 없음

3.5 알고리즘 설계

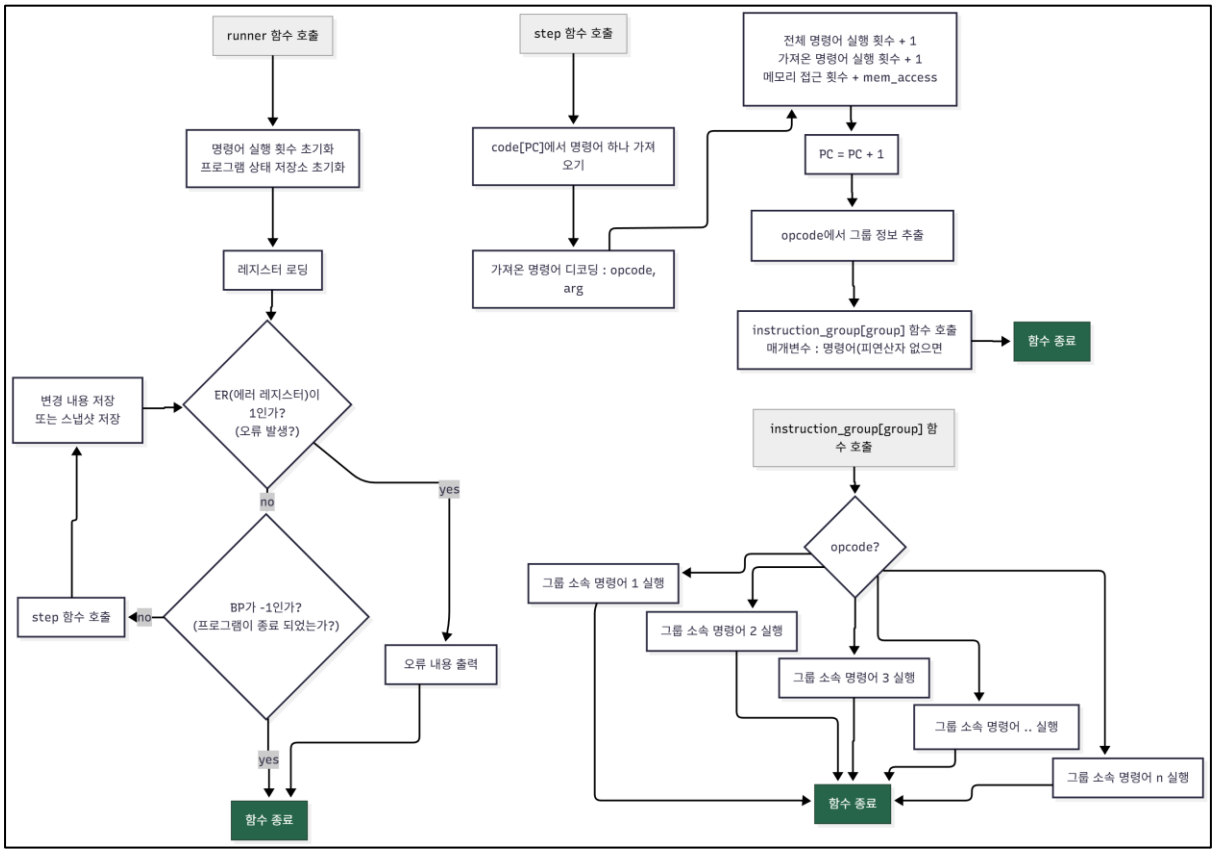
3.5.1 어셈블 단계[viii]



<그림 13 어셈블 단계 흐름도>

어셈블러는 assemble() 함수 호출로 시작되며 한 줄씩 읽은 문자열이 들어온다. 그러면 이 문자열을 구문분석한다. 구문분석된 문자열은 문법과 형식 검사를 거치며, 잘못된 명령어나 토큰이 발견되면 오류로 처리된다. 명령어가 bgn이면 변수 테이블을 초기화하고, 이후 sym 명령어를 통해 변수를 정의한다. 변수 중복이 발생하면 오류가 발생한다. 레이블 정의가 감지되면 레이블 이름을 추출하여 등록하며, 중복된 레이블이 있으면 오류로 처리된다. 일반 명령어의 경우 인자 수와 타입, 범위를 검사하며, 인자가 변수라면 변수 테이블을 통해 주소로 변환하고 정의되지 않은 변수는 오류로 처리된다. 인자가 레이블일 경우에는 정의 여부를 확인하고, 아직 정의되지 않은 레이블이면 patch 리스트에 등록하여 추후 주소를 채우도록 한다. 모든 인자가 처리되면 명령어를 인코딩하여 코드 영역에 저장하며, end 명령어가 나오면 patch 리스트를 순회하면서 레이블 주소를 채워 넣는다. 모든 patch가 완료되면 어셈블러의 실행이 종료된다.

3.5.2 실행 단계

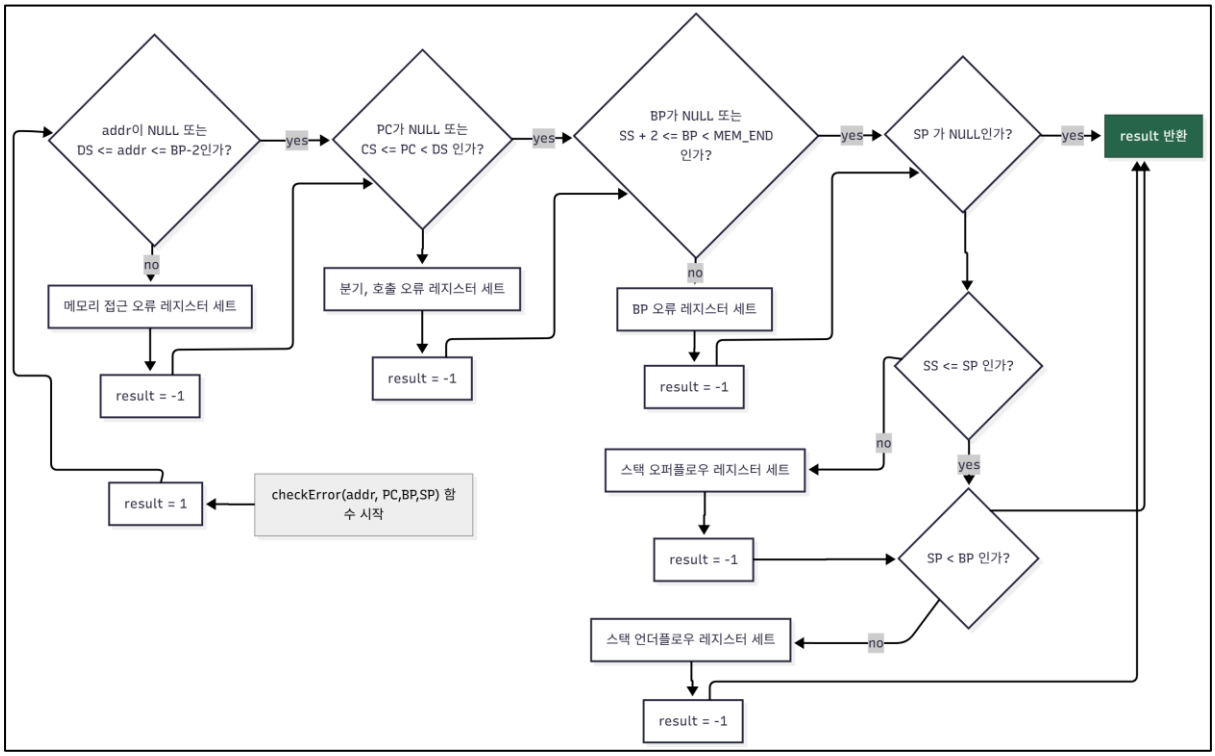


<그림 14 실행 단계 흐름도>

실행기는 runner() 함수 호출로 시작되며, 사용할 각종 통계 정보와 실행에 사용하는 여러 변수들을 초기화 한다. 이후 레지스터 변수들을 불러온다. 실행은 step() 함수 호출과 명령어 실행 후 변경된 사항을 저장하는 것을 반복하는 과정으로 진행된다. step() 함수가 호출되면 메모리의 Code 영역에서 명령어를 하나 가져온다 가져온 명령어에서 opcode와 arg를 추출한 후 opcode가 유효한 opcode라면 통계 정보를 업데이트 하고 PC를 다음 주소로 옮겨준다. opcode의 앞 3비트에서 그룹 번호를 추출하여 명령어 그룹 테이블을 통해서 명령어 그룹 실행 함수를 호출한다. 명령어 그룹 실행 함수는 opcode의 뒤 3비트를 추출해 어떤 명령어인지 해석한다. 이후 그 명령

어와 맞는 명령어 함수를 호출한다. 또한 일정 주기마다 프로그램 상태 스냅샷을 저장해 추후 단계별 실행 할 때 보다 빠르게 상태를 찾아갈 수 있도록 한다.

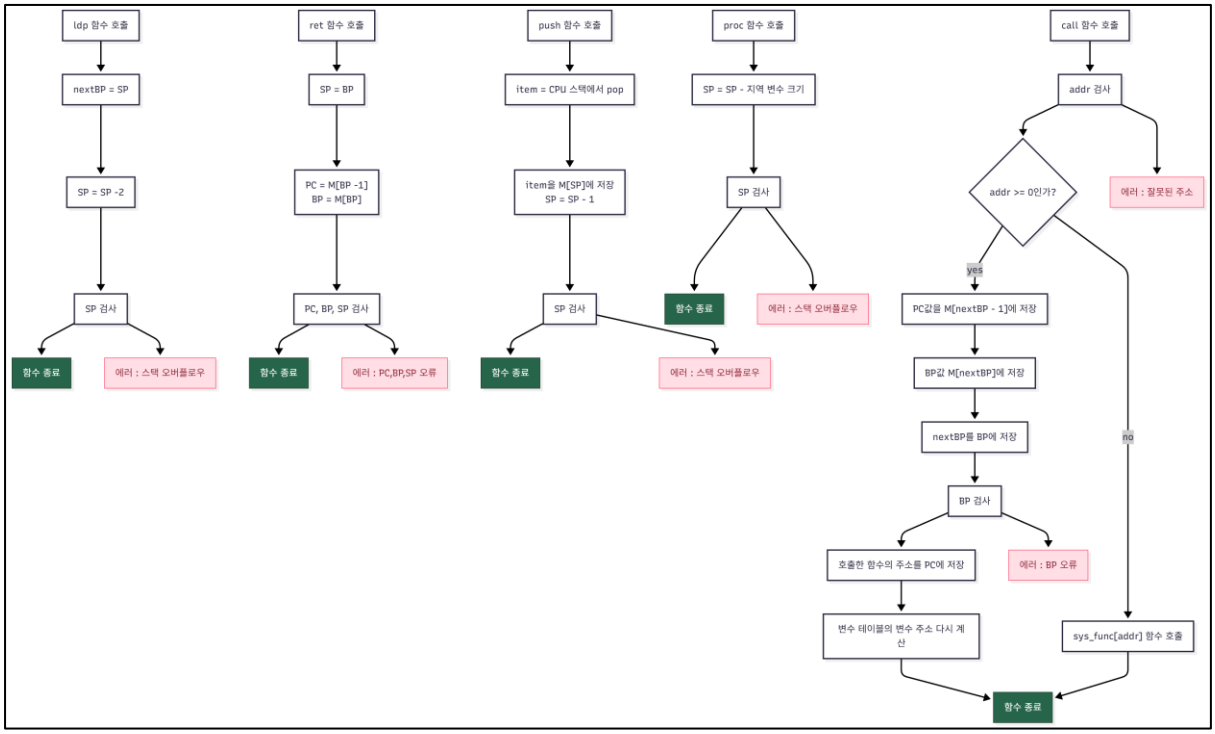
3.5.3 오류 검사



<그림 15 오류 검사 흐름도>

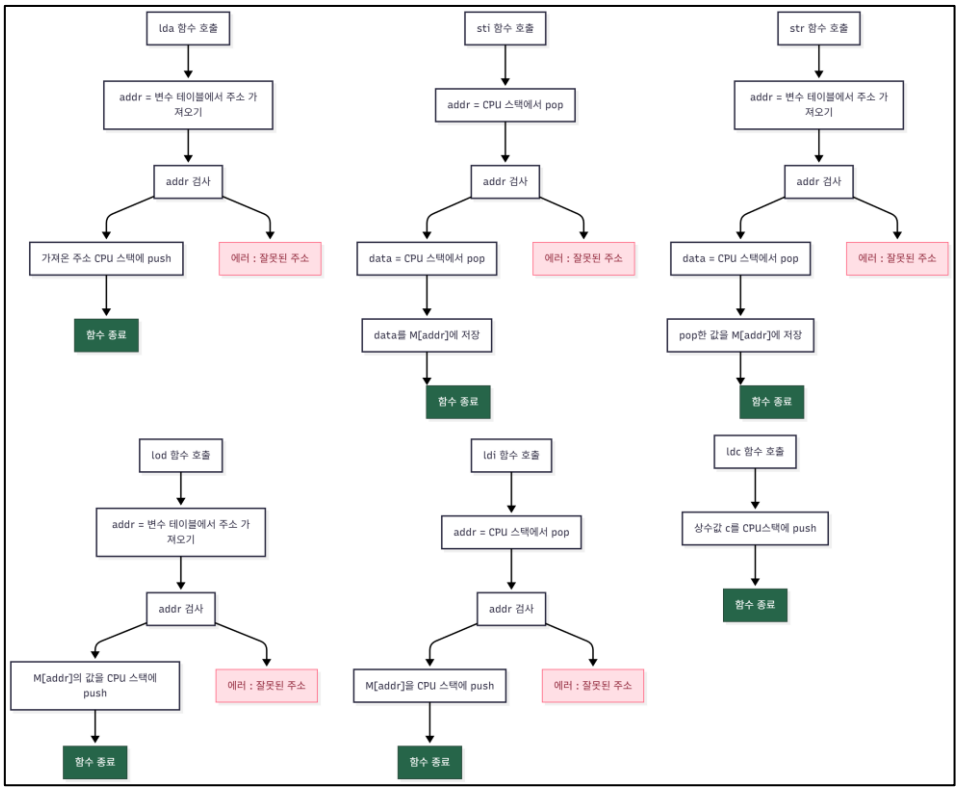
오류 검사는 checkError(addr,PC,BP,SP) 함수를 호출하는 것으로 시작된다. 변수 호출시 인자로 넘겨지는 현재 접근하고자 하는 메모리 주소, PC, BP, SP값을 받아, 각 값들이 정상적인 범위 내에 있는지 검사한다. 만약 검사를 하지 않는려면 NULL을 해당 부분에 NULL을 넘겨줘 검사를 하지 않을 수 있다. 만약 각 값들이 정상 범위에 있지 않다면 결과로 -1을 반환해 실행기 쪽에서 처리할 수 있도록 하고 오류가 있는 값에 대한 상태 레지스터들을 세트하여 실행기측에서 어떤 에러가 있었는지 확인하고. 해당 에러에 대한 내용을 사용자에게 보여줄 수 있다.

3.5.4 명령어 처리



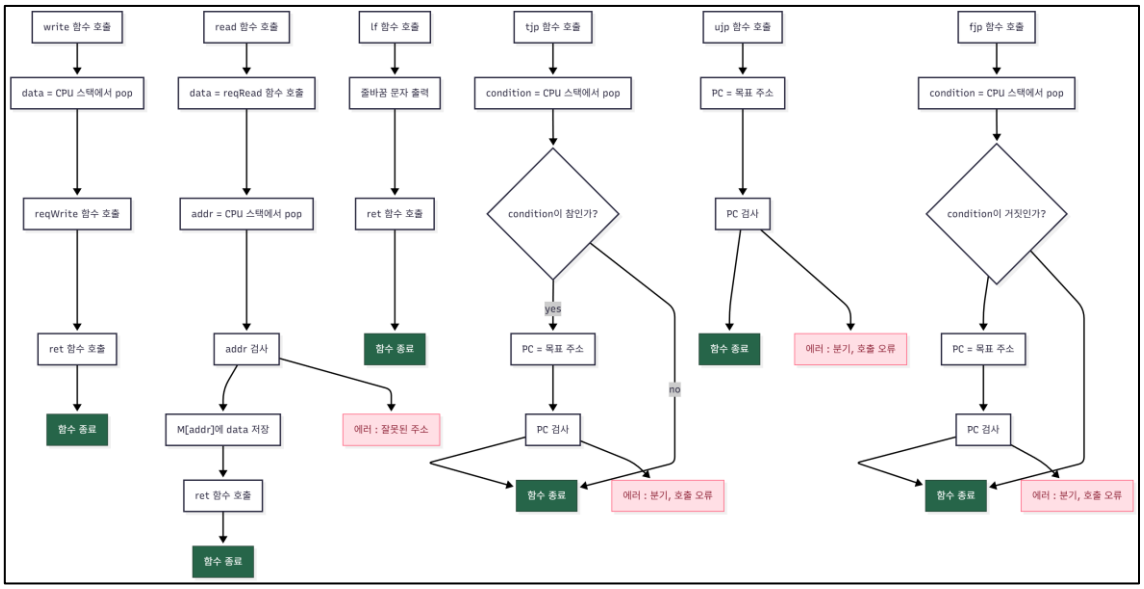
<그림 16 함수 정의, 호출 관련 명령어>

그림은 함수 정의, 호출에 관련된 명령어를 처리하는 함수들이다. ldp는 함수 호출을 준비하는 역할로 다음 BP값이 될 SP를 저장하고, SP의 위치를 조정해 인자 저장 준비를 한다. push는 실인자를 저장하는 역할로 CPU스택에서 값을 꺼내 메모리에서 SP의 위치에 넣어준다. call은 실제로 함수를 호출하는 역할로 call()함수를 호출할 때 인자로 같이 넘겨주는 label의 주소를 이용한다. 만약 호출된 함수가 시스템 함수라면 매핑된 시스템 함수를 호출하고 시스템 함수가 아니라면 복귀할 PC값과 BP값을 메모리에 저장한 뒤 PC값을 label의 주소로 바꿔준다. 이후 변수 테이블의 변수 주소들을 업데이트 한다. proc은 지역 변수 개수만큼의 메모리 스택을 확보하는 역할을 한다. SP를 지역 변수 개수만큼 조정한다. ret는 함수를 끝내고 복귀하는 역할이다. SP를 현재 BP값으로 바꾸고 저장해두었던 이전 PC, BP값을 복구한다.



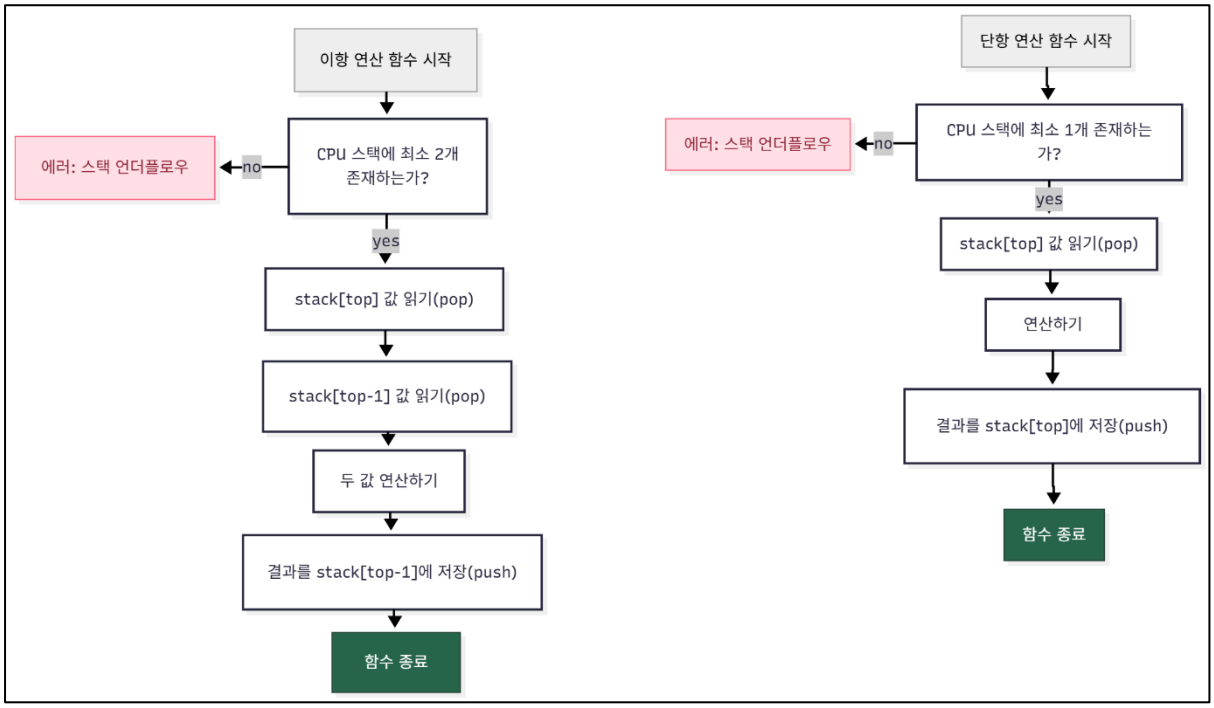
<그림 17 데이터 이동 관련 명령어>

그림은 데이터 이동에 관련된 명령어를 처리하는 함수들이다. lod는 메모리에서 값을 가져오는 역할을 한다. 변수 테이블의 인덱스를 인자로 받아 해당 변수의 주소를 알아낸다. 이후 주소에서 값을 가져와 CPU스택에 저장한다. lda는 변수의 주소를 가져오는 함수이다. 인덱스를 인자로 받아 변수 테이블에서 주소를 CPU스택에 저장한다. ldc는 상수를 CPU스택에 저장한다. ldi는 간접 주소 지정 방식으로 메모리에서 값을 가져오는 역할로 CPU스택에서 주소를 꺼내 메모리에서 꺼낸 주소를 이용해 값을 가져와 CPU스택에 저장한다. str은 메모리에 값을 저장하는 역할로 lod와 동일하게 변수테이블을 활용해 주소를 가져오고 CPU스택의 값을 주소에 저장한다. sti는 간접 주소 지정 방식으로 메모리에 값을 저장하는 역할로 CPU스택에서 주소와 값 둘다 가져와 메모리에 저장한다.



<그림 18 시스템 함수 및 분기 명령어>

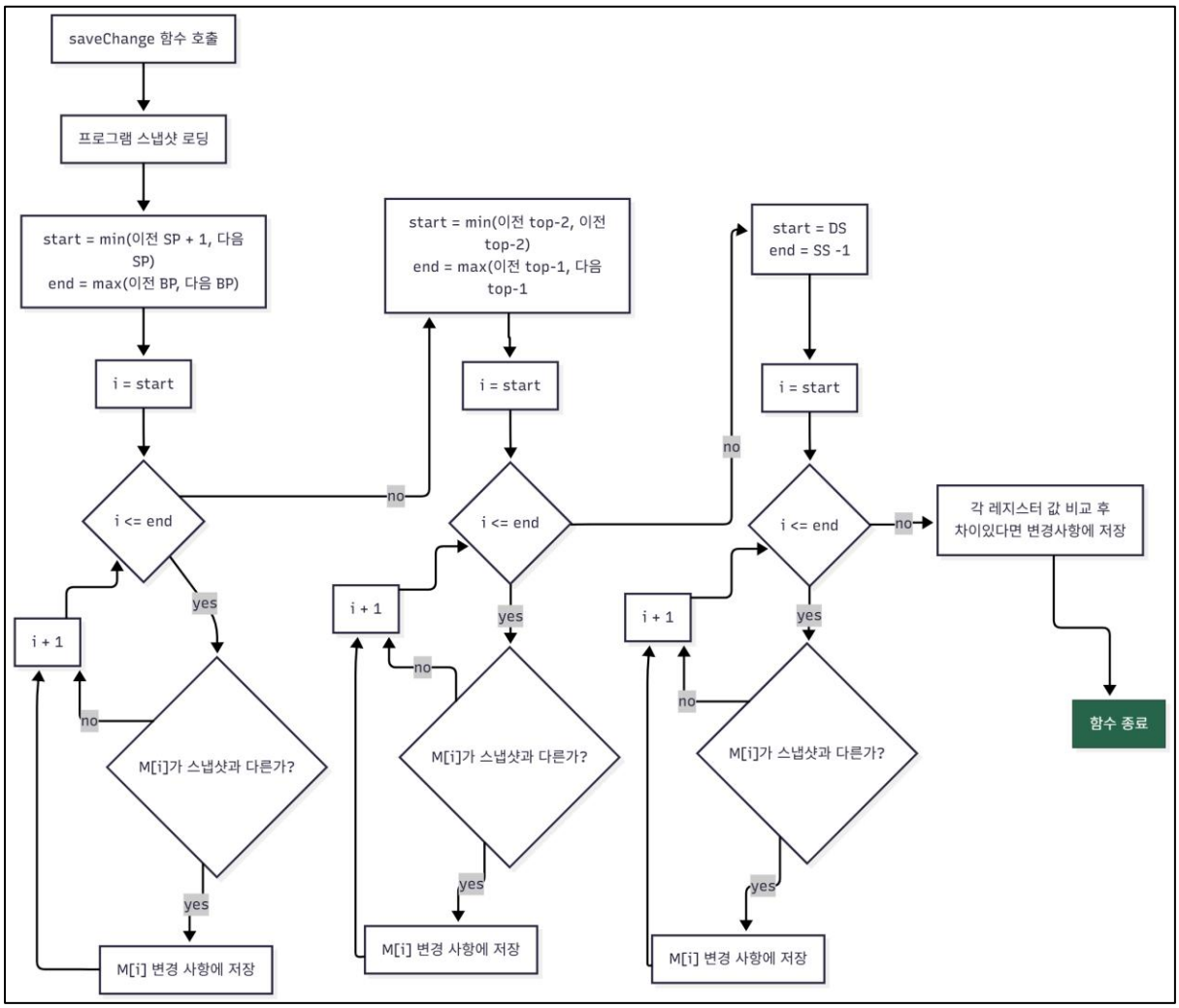
그림은 시스템 함수와 분기 명령어를 처리하는 함수들이다. write 함수는 CPU스택의 값을 출력하는 함수로 reqWrite라는 함수를 다시 호출해 GUI 또는 CLI 환경에 맞는 출력을 한다. read함수는 입력을 받아 CPU스택에서 주소를 꺼내 주소에 저장하는 함수로 reqRead라는 함수를 다시 호출해 GUI 또는 CLI 환경에 맞는 입력을 받는다. lf 함수는 줄바꿈 문자를 출력하는 함수이다. tjp는 조건이 참일 때 분기하는 역할로 CPU스택의 값이 참이면 인자로 받은 주소로 PC를 수정한다. fjp는 조건이 거짓일 때 분기하는 역할로 CPU스택의 값이 거짓이면 인자로 받은 주소로 PC를 수정한다. ujp는 무조건 분기하는 역할로 인자로 받은 주소로 PC를 수정한다.



<그림 19 이항 및 단항 연산 함수>

이항 연산자는 CPU 스택에서 top과 top-1의 두 값을 꺼내 연산한 뒤, 결과를 top-1 위치에 저장하고 top을 감소시키는 방식으로 동작한다. 예를 들어 add 명령어는 stack[top-1] = stack[top-1] + stack[top]; top--;처럼 처리되며, sub, mult, div, mod 등도 연산자만 다를 뿐 동일한 방식으로 수행된다. 비교 연산자(lt, gt, eq)와 논리 연산자(and, or)도 동일하게 top-1에 결과(참/거짓)를 저장한다. 단항 연산자는 스택의 top 값 하나만 사용하며, not은 논리 부정, neg는 부호 반전을 수행한다. 스택에 값이 부족하면 언더플로우 에러가 발생한다.

3.5.5 변경 내용 저장

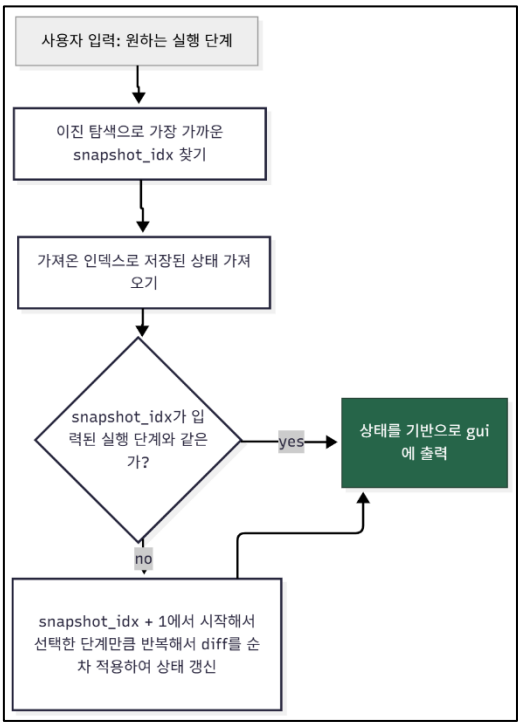


<그림 20 변경 내용 저장 흐름도>

명령어가 실행되고 난 후 프로그램에 어떤 것들이 바뀌었는지 검사한다. 함수가 시작되면 이전 명령어 실행 후의 스냅샷을 가져온다. 이후 현재 프로그램의 상태와 비교를 시작한다. 비교 순서는 메모리의 Stack 영역, CPU 스택, Data 영역, 레지스터 들 중에서 변화가 일어날 수 있는 곳만 비교를 진행한다. 만약에 변경사항이 발견되면 해당 변경사항을 담을 수 있는 Change 구조체 연

결리스트를 만들어 ChangeList에 넣어준다.

3.5.6 GUI 실행 흐름 제어



<그림 21 GUI 제어 흐름도>

이 알고리즘이 실행되기 전에 실행기는 전체 상태를 일정 간격으로 스냅샷 형태로 저장하고, 그 외 단계에서는 변경된 부분만 기록한다. 사용자가 슬라이더나 버튼을 통해 특정 단계를 선택하면, 시스템은 선택한 단계와 가장 가까운 스냅샷의 인덱스를 이진 탐색으로 찾아 해당 시점의 상태를 불러온다. 선택한 단계가 스냅샷 단계와 일치하면 상태를 그대로 GUI에 출력하고, 일치하지 않으면 스냅샷 이후부터 선택한 단계까지의 변경 점을 차례대로 적용하여 상태를 갱신한 뒤 GUI에 반영한다.

이 방식은 전체 상태를 단계마다 저장하지 않고 일정 간격으로만 스냅샷을 생성하기 때문에 메모리 사용량을 크게 줄일 수 있다. 또한 특정 단계로 이동할 때 이진 탐색으로 가장 가까운 스냅샷을 효율적으로 찾고 이후 변경 점만 적용하므로 탐색과 복원 속도가 빠르다. 이러한 구조는 실행 기록이 길어질수록 저장 효율성과 조회 성능 모두에서 높은 확장성을 제공하며, GUI 상에서도 사용자가 원하는 단계로 즉각적이고 부드럽게 이동할 수 있는 경험을 제공한다.

3.6 UI 설계

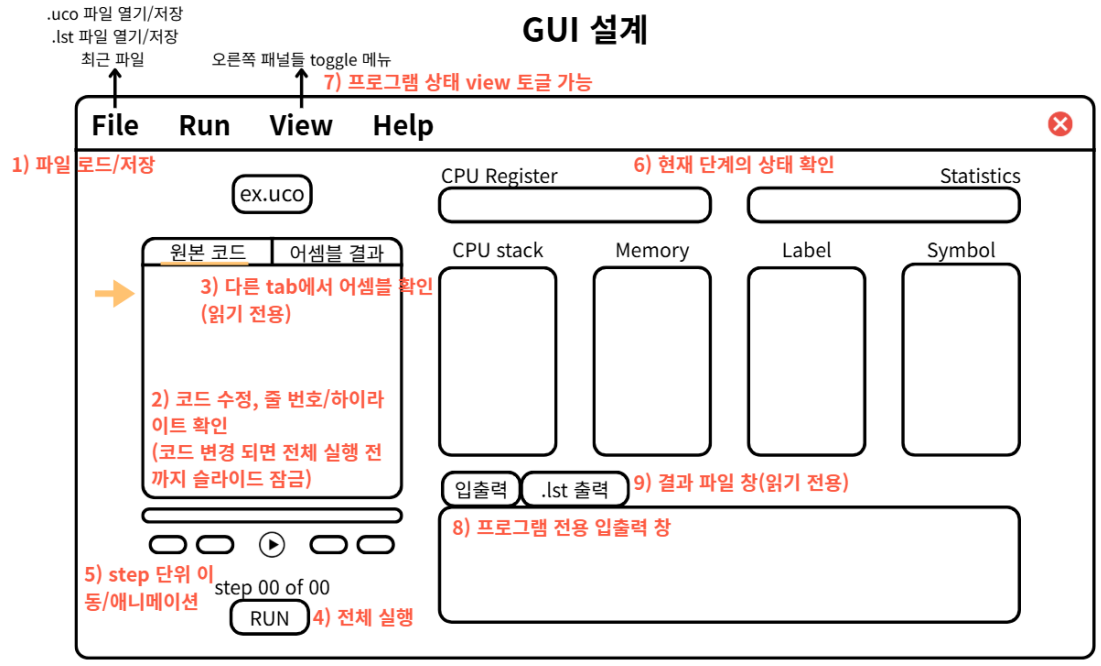
3.6.1 CLI 환경

```
>>> uci ucode.uco
어셈블 시작
어셈블 완료
실행시작
HelloWorld
실행 완료
./Statistics.lst 경로에 통계 파일이 생성되었습니다.
```

<그림 22 CLI 예시>

사용자가 실행하고자 하는 mU-Code 파일을 인자로 인터프리터를 실행하면 CLI 환경으로 프로그램을 실행할 수 있다. 처음 실행하면 입력한 파일에 대해서 어셈블 단계가 시작되면 이 단계는 CLI에 텍스트로 출력되는 어셈블 시작, 어셈블 완료로 추적 할 수 있다. 이후 실행단계가 실행되는데 먼저 실행 시작이 출력되고 사용자가 입력한 프로그램이 실행된다. 사용자가 입력한 프로그램이 종료되면 실행 완료라는 텍스트가 출력되고 lst 파일의 경로를 출력한다.

3.6.2 GUI 환경[ix]



<그림 23 GUI 설계>

GUI는 사용자가 프로그램을 편집, 실행, 상태 확인을 직관적으로 수행할 수 있도록 구성되어 있다. 상단 메뉴바에서 파일 열기와 저장 기능을 제공하며 .uco 파일은 코드 편집창에 로드되고 .lst 파일은 실행 결과 및 통계를 확인할 수 있는 패널에 로드된다. 코드 편집창은 명령어 하이라이트와 줄 번호 표시를 지원하며 어셈블 결과창은 읽기 전용으로 표시된다.

실행 컨트롤 영역에는 전체 실행 버튼과 Step 단위 이동 버튼(First, Prev, Next, Last), 재생/일시정지 토글, 슬라이더를 제공하여 프로그램 실행 흐름을 단계별로 확인할 수 있다.

우측 영역에는 CPU 레지스터, CPU 스택 및 메모리, 레이블과 심볼 리스트, 명령어 통계 패널이 배치되어 있으며, 프로그램 실행 상태를 실시간으로 모니터링할 수 있다. 입출력창에서 입력과 출력 기록을 동시에 확인할 수 있다.

4. 추진 전략 및 일정

4.1 추진 전략

본 프로젝트의 추진 전략은 명확한 역할 분담과 체계적인 개발 절차를 기반으로 하여 효율적이고 안정적인 시스템 구현을 목표로 한다. 먼저 정보 수집 방법으로는 PDF 자료, AI, 구글 검색, GTK3 공식 문서를 활용하여 필요한 기술 및 참고 정보를 확보한다. 개발 절차는 문제 정의, 요구분석, 설계, 구현, 테스트, 배포의 순차적 과정을 따르는 폭포수 모델을 채택하여 각 단계에서 발생할 수 있는 오류와 요구사항 변경에 신속히 대응할 수 있도록 한다.

역할 분담은 각 모듈을 고려하여 이루어진다. 권기영은 어셈블러, ALU, .uco 파일 입력 및 GUI 구

현을 담당하며, 이강석은 ALU를 제외한 실행기(Runner)와 .lst 파일 생성 및 CLI 구현을 담당한다. 이를 통해 모듈 간 중복 작업을 최소화하고 개발 속도를 높인다.

개발 도구로는 C언어와 GTK3 라이브러리를 사용하며, 협업은 GitHub을 기반으로 PR(Pull Request)과 코드 리뷰를 통해 진행한다. 브랜치 정책은 main 브랜치에 안정화된 코드만 반영하고, develop 브랜치는 PR을 통해서만 merge 하며, 기능 개발은 자유롭게 feature 브랜치에서 수행하도록 규정한다. 코드 스타일은 변수 및 파일명은 snake_case, 함수명은 camelCase를 준수하여 유지보수성을 높인다.

이러한 추진 전략은 명확한 정보 수집, 체계적인 개발 절차, 역할 분담, 개발 도구 활용, 브랜치 및 코드 규칙 준수를 통해 프로젝트의 효율성과 안정성을 확보하고, UI를 포함한 어셈블러 및 실행기 시스템의 완성도를 높이는 데 목적을 둔다.

4.2 일정

(수행 기간 : 2025년 10월 27일 ~ 2025년 12월 3일)					
구분 / 수행 내용	1주	2주	3주	4주	5주
기획 및 상세 설계					
입출력 기능 구현					
가상 머신 구조 구현					
어셈블러 구현		권기영			
명령어 함수 구현		이강석			
실행기 구현		이강석			
테스트 코드 구현			이강석		
CLI 구현			이강석		
GUI 구현			권기영		
테스트 및 피드백					
결과 보고서					

5. 참고문헌

[i] [Python] Interpreter and PVM (Python Virtual Machine), https://dsaint31.tistory.com/496

[ii] Compiler Language and Interpreter Language, https://dsaint31.me/mkdocs_site/CE/ch08/ce08_compiler_interpreter/

[iii] 바이트코드, https://namu.wiki/w/바이트코드

[iv] 가상머신, https://namu.wiki/w/가상머신

[v] Java Virtual Machine, https://namu.wiki/w/Java%20Virtual%20Machine?from=자바%20가상%20머신

[vi] SWP 3단계 PDF 참조

[vii] 000 교수님의 컴퓨터 구조 수업 PDF 참조

[viii] Copilot 사용해 어셈블러에서 발생할 수 있는 에러 목록 질문 후 내용 추가

[ix] GTK widget gallery 참고: Gtk – 3.0: Widget Gallery, GUI 실행 제어 기능 추가하는데 참고: Python Tutor code visualizer: Visualize code in Python, JavaScript, C, C++, and Java