# 고급 레벨 과제

## U-Code Interpreter - 계획 및 설계 보고서


| | | | | |
|---|---|---|---|---|
| 제출일: 2022.12.09. | 전공 | 컴퓨터공학과 | | 5조 |
| 어드벤처디자인_02 | 학번 | 20190602 | 20190622 | 20190788 |
| 담당 교수: 김경수 교수님 | 이름 | 설진영 | 손현락 | 이근탁 |

# 목 차

# Ⅰ. 프로젝트의 목표

U-Code는 스택 기반으로 동작하는 간단한 CPU에서 실행되는 것을 가정하는 중간 언어(intermediate language)이다. 중간 언어란 C와 같은 고급 언어로 된 프로그램을 실제 CPU에서 실행될 수 있는 기계어로 변환하는 컴파일 과정의 중간 단계로서, 다양한 고급 언어와 CPU를 연결하는 역할을 한다.

이번 프로젝트에서는 U-Code로 작성된 프로그램을 읽어서 실행하는 인터프리터를 작성한다. 스택 기반의 CPU를 가정하고 있으므로 모든 명령어는 메모리로부터 스택에 데이터를 가져와서 작업하고, 그 결과도 스택에 저장한다.

# Ⅱ. 문제정의

본 과제의 문제정의는 다음과 같다.

1. U-Code로 작성된 프로그램을 해석하여 실행하는 인터프리터를 작성한다.

2. 해당 프로그램의 실행 과정이나 결과를 화면에 표시한다.

3. 소스 uco 코드와 어셈블 결과, 명령어별 통계, 명령어 실행 횟수, 실행 사이클 수 등 실행과 관련된 다양한 정보들을 포함하는 로그 파일(*.lst)을 생성한다.

# Ⅲ. 아이디어 / 접근방법

## 실제 CPU가 명령어를 실행하는 과정을 모방

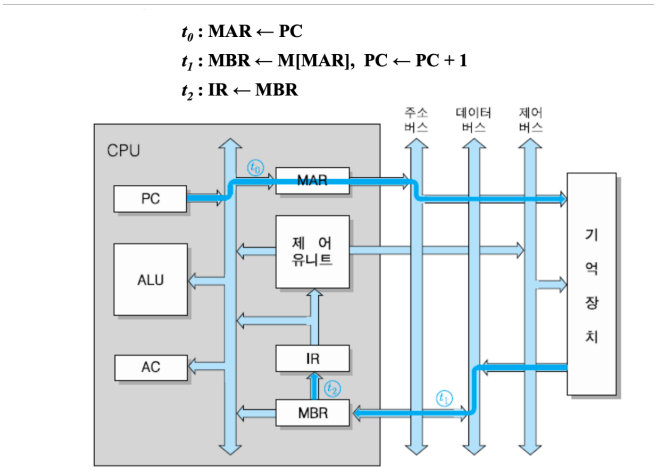


그림 2 명령어 인출 사이클

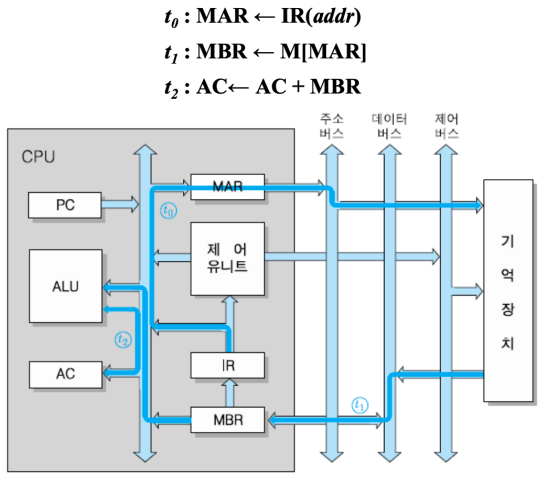


그림 3 ADD 명령어 실행 사이클

실제 CPU에서 명령어를 실행하는 경우 명령어 인출 단계, 명령어 실행 단계 두 과정을 거치게 된다. 명령어 인출 단계에서는 Program Counter(이하 “PC”라 한다)에 저장된 주소 값을 통해 주기억장치에서 명령어를 인출해 IR에 적재하는 과정이다. 이때, 명령어를 인출하는 즉시 PC 값은 다음 명령어를 가리키도록 증가 된다. 명령어 실행 단계에서는 각각의 연산자에 맞게 데이터를 인출하거나 처리, 또는 데이터를 저장하는 과정을 거친다. 서브 루틴을 호출하는 명령인 경우에는 기존의 PC 값을 주기억장치의 스택 영역에 저장하고, 분기할 주소가 PC에 저장된다. 서브 루틴에 복귀하면 스택에 저장된 PC 값을 다시 불러와

남은 명령을 실행한다. 앞서 설명한 실제 명령어 실행 사이클을 기능과 과정을 단순화하여 본 프로젝트에 적용할 수 있다. 우선 사용자로부터 U-Code(*.uco) 파일을 입력받아 열고, Assembler를 통해 토큰화(Tokenize) 및 해석(Assemble) 과정을 거쳐 메모리에 적재할 수 있는 오브젝트 코드(Object Code) 형태로 변환한다. 이후 메모리의 코드 영역에 적재한다. 명령어 적재 과정이 끝나면 제어장치(해당 프로그램에서는 제어장치와 산술논리연산장치가 통합됨)가 레지스터 세트 및 주기억장치와 상호 작용하며 차례대로 명령어들을 실행한다.

### 구현 과정에서의 제한사항

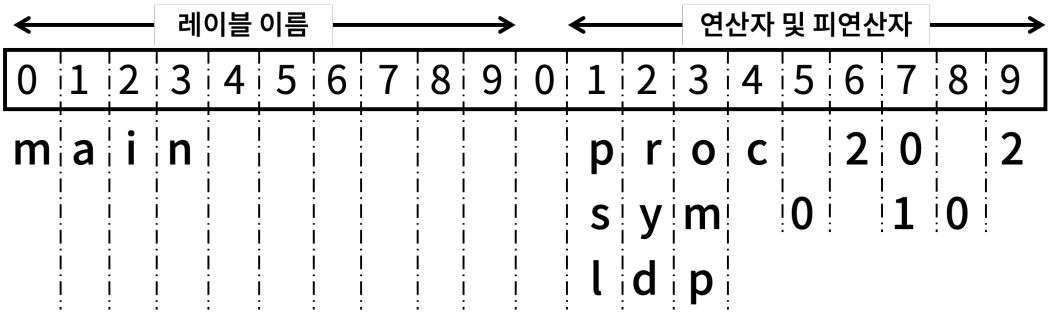


그림 4 U-Code 문법

프로그램 구현 과정에서의 제한사항은 다음과 같다. 우선 U-Code의 모든 데이터는 크기가 1인 정수형만 존재하는 것으로 가정한다. 다양한 자료형을 제공하기엔 프로그램을 구성하고 제작하는 과정이 매우 복잡해지고 시간이 많이 소요될 것을 예상해서이다. 그리고 모든 연산 과정은 스택 기반의 레지스터 하나에서만 이루어지기 때문에 스택에 있는 데이터에 대한 명령의 경우 피연산자(Operand)를 표현하지 않는다. 다만 메모리와 스택 간의 데이터 이동 명령의 경우 피연산자가 1개이며 임의의 정수로 표현한다. 마지막으로 주어진 U-Code의 경우 정해진 열을 지켜 작성되어있는 것으로 가정한다. 레이블 이름을 저장하는 레이블 영역 10칸, 연산자와 피연산자를 저장하는 레이블 영역 10칸으로 크게 두 영역으로 구성되며, 연산자와 피연산자 사이에는 1칸의 공백으로 띄워져 있다.

## Ⅳ. 방법론이나 구현 언어

### 방법론

우선 프로그램의 시작은 사용자로부터 U-Code를 받아들이는 것이다. U-Code의 내용을 프로그램에서 인식할 수 있어야 하기에 주어진 U-Code를 적절히 토큰화하고 해석해 정수로 이루어진 오브젝트 코드로 변환해야 하고 이러한 역할을 하는 객체를 Assembler이라고 가정한다. Assembler 객체는 특정 메소드를 실행하여 U-Code의 시작 주소, 레이블 이름이 선언되어있는 위치, 연산에 필요한 메모리의 공간 등의 정보를 해석해 저장하고 있어야 할 것이다. 그렇게 프로그램이 알아들을 수 있게 '처리'된 코드와 정보들은 또 다른 특정 객체로 적재되도록 한다. 이 객체의 이름은 Memory라고 가정한다. Memory 객체에서는 Assembler 객체의 정보(연산에 필요한 stack, 데이터 영역, 코드 영역의 크기 등)를 읽어와 적합한 크기의 공간을 할당하고, 코드 영역에 Assembler 객체에서 처리된 오브젝트 코드를 저장하도록 한다. 이렇게 입력받은 U-Code로 프로그램이 실행되도록 준비가 된 경우, 특정 객체에서 프로그램 실행을 진행한다. 이 객체의 이름을 ControlUnit, 프로그램이 실행될 때 다양한 레지스터들의 도움을 받으므로 그 객체의 이름을 RegisterSet으로 가정한다. ControlUnit는 RegisterSet의 PC값을 통해 Memory 객체의 코드

영역에서 각 코드를 한 줄씩 읽어와 RegisterSet의 범용 레지스터 스택과 상호작용을 통해 명령어를 처리한다. 함수 호출과 같은 명령은 현재 PC의 값을 Memory 객체의 스택 영역에 저장하고, 함수가 종료된 뒤(함수 종료는 end 명령으로 파악함) 다시 원래 실행하던 명령의 위치로 돌아갈 수 있도록 해야 한다.

U-Code 프로그램의 실질적인 실행 코드는 bgn 명령부터 처음으로 나오는 end 명령까지며, 이 이외의 코드는 실행할 함수나 서브루틴들을 정의한 부분이다. 실질적인 실행 코드가 모두 실행되었다면 U-Code 실행이 완료된 것인데, 이는 end 명령을 더 이상 처리할 수 없는 경우(Memory의 스택 영역이 비어있는 경우)와 같다고 볼 수 있다.

### 구현 언어

앞서 설명한 것과 같이 실제 컴퓨터가 명령을 실행하는 과정을 모방하여 해당 시뮬레이션을 제작하려 한다. 이때, 각 사물들(CPU를 구성하는 장치들과 주기억장치)의 특성이나 역할을 추상화하여 일종의 객체를 구성하고 객체들 간의 상호작용을 통해 프로그램을 개발할 것이므로 객체지향 프로그래밍 언어를 사용하는 것이 적절하다고 판단하였다. 게다가 이번 고급 레벨의 U-Code Interpreter 프로그램을 개발하는 것에 대하여 조원끼리 회의한 결과 생각보다 구조를 구상하고 구현하는 데 주어진 시간 안에 해내는 것은 빠듯할지도 모른다는 결론을 도출했다. 하지만 비슷한 구조로 이루어져 있는 지난 프로젝트는 C++ 언어를 사용해 본 경험이 있기에 이번 프로젝트에서도 구현의 편의를 위해 C++ 언어를 사용하기로 하였다.

## Ⅴ. 추가하고 싶은 기능

### 프로그램의 실행 과정이나 결과를 확인할 수 있는 GUI 구현

사용자로부터 U-Code가 저장되어있는 파일을 전달받아 오브젝트 코드로 해석하고, 이를 바탕으로 프로그램이 동작하는 동안 각 객체의 상태 (스택, 인출한 명령어, 해석한 명령어)와 명령어를 실행한 결과를 동시에 출력하게끔 GUI를 구성해 사용자 경험을 높인다.

### U-Code 디버깅 기능

일반적으로 코드를 한 번에 모두 실행하고 프로그램을 종료한다. 만약, 코드에 문제가 있어 의도하지 않는 값이 나오거나 에러가 발생할 수 있는데, 어떤 경우에서 해당 에러가 발생하는지 파악하기 힘들 수 있다. 따라서 현재 어떤 코드가 실행 중인지 쉽게 파악할 수 있도록 명령을 수행 중인 라인에 화살표 등으로 표시하고, 각 명령 실행 사이에 지연시간을 둔다. 또한, 중단점 기능과 한 줄씩 실행하기 기능 등의 디버깅 기술을 도입을 고려해볼 수 있다.

## Ⅵ. 사용자 요구사항 분석 및 명세

### 현재 개발하는 프로그램에 대한 사용자의 요구사항, 기능별 분류

사용자 요구사항은 U-Code로 작성된 프로그램을 읽어서 실행하는 가상머신 시뮬레이터를 만드는 것이다. 시뮬레이터에서는 프로그램의 실행 결과를 표시해야 하고, 통계정보 파일을 작성해야 한다.

주어진 요구사항에서 필요한 기능은 다음과 같다.

- U-Code 파일 불러오기, U-Code 파일 해독, 해독된 코드 실행, 실행 결과 출력

## 각 기능별로 분류한 사용자 요구사항

1. U-Code 파일 불러오기: U-Code 파일을 불러오는 기능이다.
2. U-Code 파일 해독: 자연어로 작성된 U-Code 파일을 해독하는 기능이다. 공백으로 구분된 각각의 인자들을 토큰화하고, 연산자와 레이블 명 등을 컴퓨터가 해석할 수 있는 정수형으로 변경한다. 해독 과정에서 문제가 발생하면 예외를 반환한다.
3. 해독된 코드 실행: 해독된 코드를 실제로 실행하는 기능이다. 실행 과정에서 문제가 발생하면 예외를 반환한다.
4. 실행 결과 출력: 코드 실행 결과와 통계 파일(*.lst)이 출력된다.

## 각 사용자 요구사항에 대한 입/출력 및 요구사항들 간의 관계

1. U-Code 파일 불러오기: 파일의 경로를 입력받음.
2. U-Code 파일 해독: 불러온 파일의 객체를 입력으로 받는다.
3. 해독된 코드 실행: 해독된 코드를 가지는 객체의 참조와 프로그램의 시작 주소를 입력으로 받는다.
4. 실행 결과 출력: 실행 객체를 입력으로 받는다.

## 사용자 요구사항과 대응되는 핵심 기능 도식화

Usecase 다이어그램을 통해 핵심 기능을 도식화 하였다.

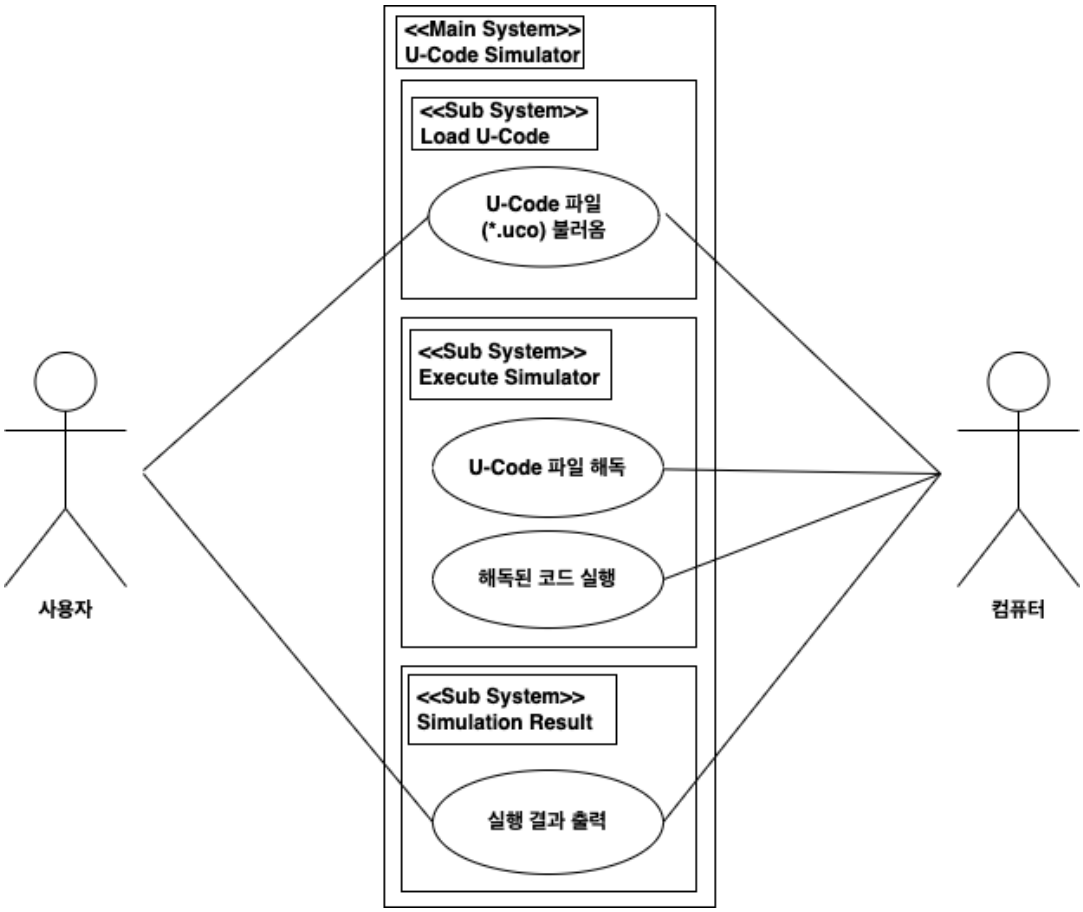


그림 5 Usecase Diagram

# Ⅶ. 프로그램 전체 구조 설계

## 프로그램의 실행 및 제약조건

U-Code의 모든 데이터는 크기가 1인 정수형만 존재하는 것으로 가정한다. 또한 모든 연산 과정은 스택 기반의 레지스터 하나에서만 이루어지기 때문에 스택에 있는 데이터에 대한 명령의 경우 피연산자(Operand)를 표현하지 않는다. 다만 메모리와 스택 간의 데이터 이동 명령의 경우 피연산자가 1개이며 임의의 정수로 표현한다. 마지막으로 주어진 U-Code의 경우 정해진 열을 지켜 작성되어있는 것으로 가정한다. 레이블 이름을 저장하는 레이블 영역 10칸, 연산자와 피연산자를 저장하는 레이블 영역 10칸으로 크게 두 영역으로 구성되며, 연산자와 피연산자 사이에는 1칸의 공백으로 띄워져 있다.

## 프로그램 구성하는 서브루틴 또는 모듈의 리스트

본 프로그램은 총 10개의 모듈로 구성된다. 프로그램 핵심 모듈은 Assembler 클래스, Memory 클래스, ControlUnit 클래스, RegisterSet 클래스로 총 4개이다. 서브 모듈로는 Label 구조체, Instruction 구조체, Opcode 열거체, UcodeException 추상 클래스, UcodeAssemblerException 클래스, UcodeRuntime Exception 클래스가 있으며 총 6개로 구성된다. 모듈과 관련된 보다 자세한 설명은 *Ⅷ. 모듈 설계 및 모듈 간 제어 흐름 설계* 에 명시하였다.

## 프로그램의 통합 구조를 보여주는 다이어그램

모듈의 구조와 사용 관계를 쉽게 파악할 수 있도록 클래스 다이어그램으로 나타내었다.

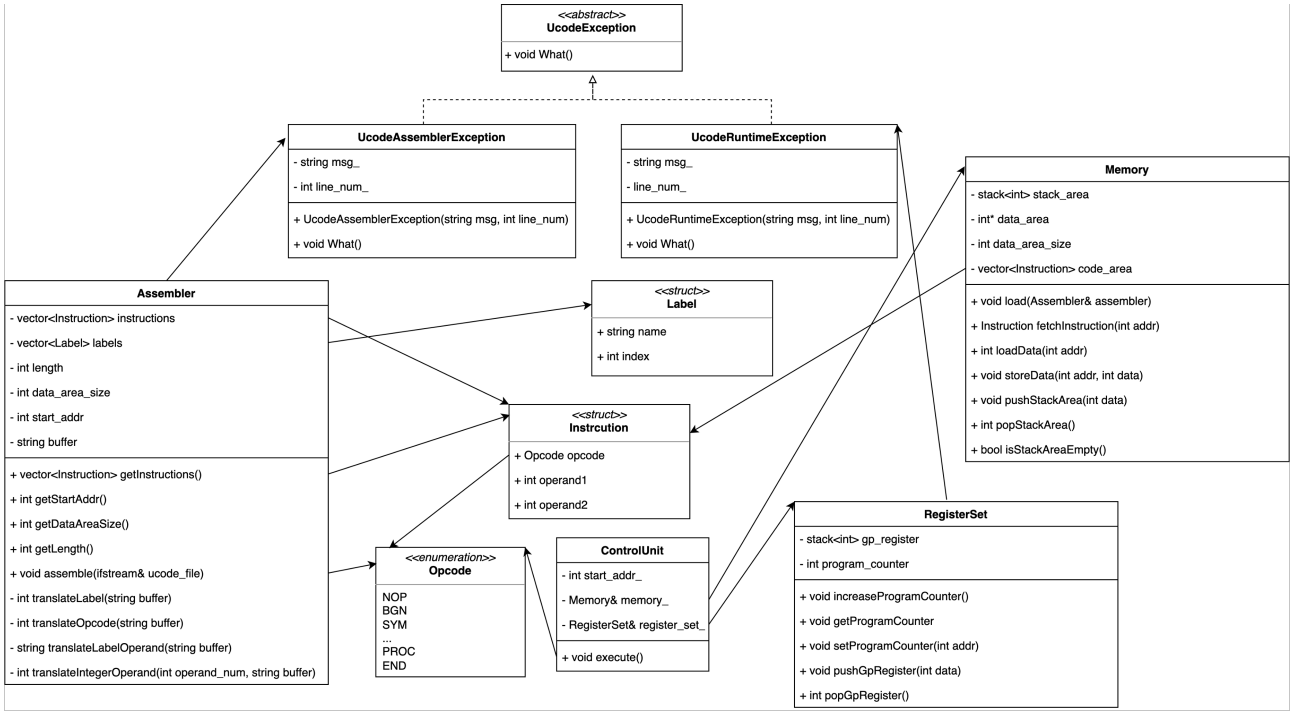


그림 6 Class Diagram

# Ⅷ. 모듈 설계 및 모듈 간 제어흐름 설계

## 모듈의 사용 목적과 기능

1. Assembler 클래스: 입력으로 받은 U-Code 파일을 연산자(Opcode)와 피연산자(Operand)로 적절히 나누고 해석하는 클래스
2. Memory 클래스: Assembler 클래스에서 해석한 오브젝트 코드들이 적재되는 클래스, 실제 컴퓨터의 주기억장치 역할을 수행함.
3. ControlUnit 클래스: Memory 클래스에 적재되어 있는 명령어들을 실제로 실행하기 위한 클래스, 실제 컴퓨터의 제어장치와 산술논리 연산장치의 역할을 수행함.
4. RegisterSet 클래스: ControlUnit 클래스가 명령어를 실행하는 과정에서 필요한 임시 기억 장치(레지스터)들의 집합을 구성하는 클래스, 실제 컴퓨터의 레지스터 세트의 역할을 일부 수행함.
5. UcodeException 클래스: 해당 프로그램 실행 중에 나타날 수 있는 각종 예외를 처리하는 클래스
   - UcodeRuntimeException 클래스: 명령어 실행 과정에서 발생할 수 있는 예외를 처리하는 클래스
   - UcodeAssembleException 클래스: 어셈블러가 명령을 해석하는 과정에서 발생할 수 있는 예외를 처리하는 클래스
6. Label 구조체: 피연산자들 중 레이블 명을 처리하기 위해 사용되는 구조체, 레이블 명의 이름과 위치를 가지고 있는 구조체이며, Assembler 클래스에서 사용됨.
7. Instruction 구조체: 연산자와 피연산자들을 담는 구조체, 하나의 구조체가 하나의 오브젝트 코드를 구성하며 명령어를 해석하는 Assembler 클래스와 적재되는 Memory 클래스에서 사용됨.
8. Opcode 열거체: 연산자들을 10진수 정수로 표현하기 위해 정의된 열거체

## 모듈의 필드 및 메소드 상세

다음은 위에 나열된 각 모듈들의 상세 내용이다.

1. Assembler 클래스

   입력으로 들어오는 U-Code들의 명령어를 기계어로 해독을 담당하는 클래스이다. 멤버 변수로는 해석될 명령어를 벡터로 가지는 instructions 벡터, 분기점들에 대한 정보를 가지는 labels 벡터, 명령어가 총 몇 줄인지 알려주는 length, 변수를 저장할 데이터 영역의 크기를 나타내는 data_area_size, bgn 레이블을 담아 프로그램의 시작주소를 나타내는 start_addr와 데이터 간 이동을 수월하게 해줄 버퍼가 있다. 그리고 메소드로는 입력으로 들어온 U-Code에 있는 여러 가지 형태의 label, opcode, operand를 적절한 형태로 해석하는 translate 계열들이 있고 실제 해석을 수행하는 assemble 메소드가 있다.

2. Memory 클래스

   Memory 클래스는 코드 영역, 데이터 영역, 그리고 스택 영역으로 구성된다. Assmbler 클래스에서 해석된 명령어들은 코드 영역에 적재된다. 스택 영역에는 분기 시 돌아올 위치를 저장하며 명령어에 따라 스택 레지스터의 데이터를 가져와 저장할 수 있는 데이터 영역이 있다. 클래스의 멤버 변수로는 오브젝트 코드들을 가지는 instructions 벡터, 데이터 영역을 구성하는 data_area 포인터와 그 크기를 나타내는 data_size_area 변수가 있고, 스택 영역을 구성하는 stack_area 변수가 있다. 메소드로는 Assembler 클래스에서 해석된 명령어들을 메모리로 가져오는 load, 메모리에서 명령어를 인출 할 수 있도록 하는 fetchInstruction, 데이터를 인출하거나 저장할 수 있도록 하는 loadData, storeData, 스택 영역에 PC값을 저장하거나 불러오기 위한 pushStackArea, popStackArea 메소드가 있다.

3. ControlUnit 클래스

   프로그램 코드들이 적재되어 있는 메모리에서 PC값을 통해 실제로 코드를 인출해 실행하는 클래스이다. 필요한 멤버 변수로는 프로그램의 시작 주소(bgn 레이블의 위치)를 나타내는 start_addr_, 실제로 코드를 인출하거나 실행할 메모리의 참조를 나타내는 memory_, RegisterSet 클래스의 참조를 가지는 register_set_변수가 있다. 메소드는 실제 명령어 인출 및 실행을 수행하는 execute 뿐이다.

4. RegisterSet 클래스

   ControlUnit 클래스가 명령어를 실행하는 과정에서 필요한 임시 기억 장치(레지스터)들의 집합을 구성하는 클래스이다. 해당 클래스는 다음에 실행할 명령어 번지를 나타내는 PC와 연산 수행을 위한 범용 레지스터로 구성되어 있다. 멤버 변수로는 범용 레지스터를 나타내는 gp_register, PC를 나타내는 program_counter 가 있다. 메소드로는 PC값을 1 증가시키기 위한 increaseProgramCounter, 분기를 위해 PC값을 변경하는 경우에 사용될 수 있는 setProgramCounter, 명령어 인출을 위해 PC 값을 사용하는 경우를 위한 getProgramCounter가 있고, 범용 레지스터를 위한 popGpRegister, pushGpRegister가 있다.

5. UcodeException 클래스

   입력으로 들어온 U-Code를 해석(Assemble)하고 실행(execute)하는 동안 발생할 수 있는 예외들을 처리하기 위한 추상 클래스이다. 이를 상속한 UcodeAssemblerException에서는 U-Code를 해석하는 동안 일어나는 오류를 처리하는데 사용되고, UcodeRuntimeException은 실행시간 동안 일어나는 오류를 처리하는데 사용된다. 각 클래스들은 오류가 발생한 위치를 나타내는 line_num_ 그리고 일어난 오류에 대한 정보를 알려주는 What 메소드로 구성된다.

6. 이외의 구조체

   U-Code로부터 읽어온 각 명령어에 대한 정보를 나타내는 Instruction 구조체가 있는데 해당 구조체는 연산자를 나타내는 Opcode와 각 연산에 쓰여야 할 피연산자들을 나타내는 operand1, operand2로 구성된다. 다음으로는 Label에 대한 정보를 담고 있는 Label 구조체가 있고 해당 구조체는 Label의 이름을 나타내는 name과 해당 Label의 위치를 나타내는 index가 있다.

7. Opcode 열거체

   해당 프로그램에서 지원하는 연산자들을 모아놓은 Opcode 열거체가 있다. Assemble 과정과 Execute 과정에서 실행할 명령어의 연산자를 파악하는 데 사용된다. 열거체를 사용하면 명령어들을 10진수로 표현할 수 있게된다.

**모듈의 실행 및 제약조건**

문제를 단순화하기 위해 모든 연산은 스택 기반 레지스터에서만 이루어진다. 또한 실제 폰 노이만 구조의 주기억장치에서는 모든 영역(code, data, stack.. etc)이 하나의 메모리로 구성되고 주소 체계를 공유하지만, 구현의 편의를 위해 각각의 영역을 별도로 나누어 처리하게 된다. 실제 컴퓨터 메모리에서 0번지는 하나이지만, 현 프로젝트에서 메모리의 0번지는 코드 영역의 0번지이거나, 스택 영역의 0번지 또는 데이터 영역의 0번지일 수 있다.

# Ⅸ. 사용자 인터페이스 설계

## 인터페이스의 전체적인 구조 도식화 및 설명

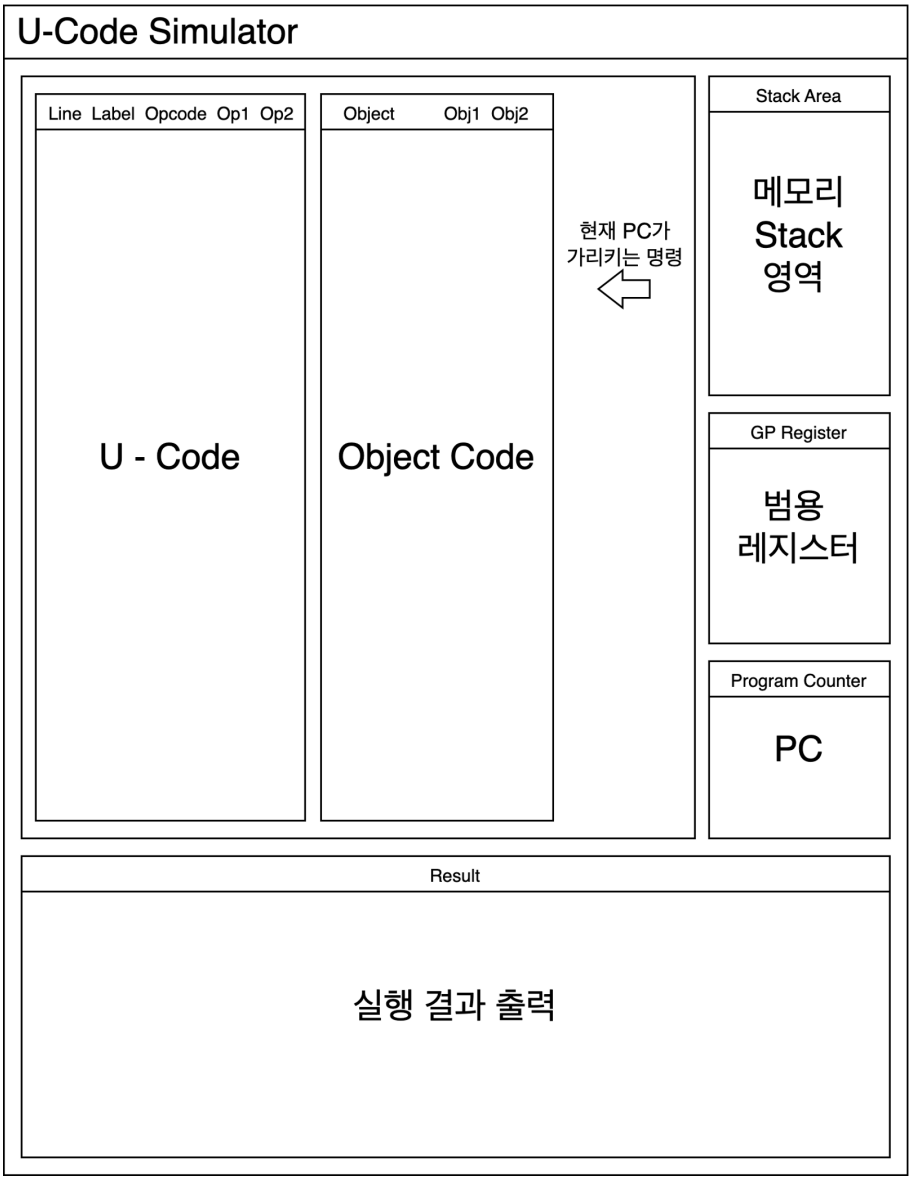


그림 7 인터페이스 구조도

해당 프로그램의 화면 구조는 다음과 같다. 프로그램은 기본적으로 CLI 환경에서 작동하도록 구성한다. 여건이 되는 경우에는 MFC 혹은 기타 다른 GUI 프레임워크를 사용해 GUI 환경을 구성할 예정이다. 화면은 Assemble 과정이 성공적으로 수행되었음을 가정한다. 만약 Assemble에 실패하였다면 별도의 경고창을 띄우며 프로그램을 종료한다. 화면에는 입력으로 넣은 U-Code 파일과 해석된 오브젝트 코드를 표 형태로 출력한다. 우측에는 현재 실행 중인 명령어 위치가 실시간으로 표시되고, 메모리 Stack 영역의 내용과, 범용 레지스터의 내용, PC의 값을 실시간으로 확인할 수 있도록 구성할 예정이다. 실행이 정상적으로 완료된다면 아래의 실행 결과 출력 화면에 출력이 표시되며, 실행 중 RuntimeException이 발생하는 등의 경우에는 해당 경고를 출력하며 종료한다.

# 참고 문헌

UML 강의자료

어드벤쳐디자인 고급레벨 문제

컴퓨터 구조 수업자료