Simple Calculator with ISA

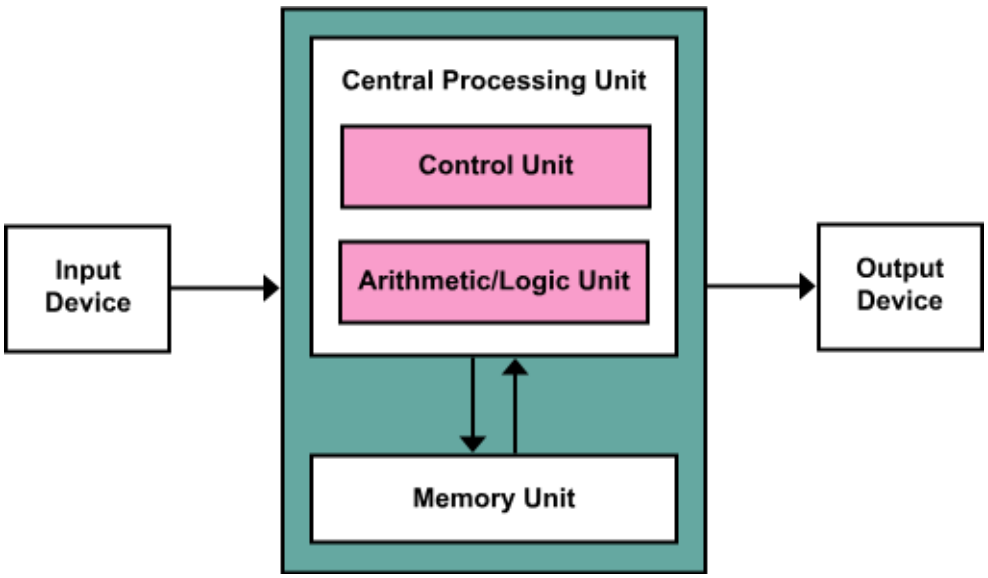
성영준 Yeong-Jun Seong
32202231(남은 휴일 5 일)
단국대학교 모바일시스템공학과

[서론]

본 계산기 프로그램은 폰 노이만 컴퓨터 아키텍처의 핵심 원칙을 반영하여 설계되었으며, 이는 사용자의 입력을 받아들여 중앙 처리 장치(CPU)에서 명령어를 처리하고, 메모리에서 해당 명령어와 필요한 데이터를 가져오는 방식으로 작동한다. 이 과정에서 프로그램은 입력된 명령들을 순차적으로 해석하고, 적절한 산술 또는 논리 연산을 수행하여, 그 결과를 다시 사용자에게 출력함으로써 계산기로서의 기본적인 기능을 수행하게 된다, 이러한 동작은 사용자가 겪는 계산과정의 복잡성을 추상화하고, CPU 내부의 연산 및 메모리의 데이터 관리를 통해 단순하고 직관적인 사용자 인터페이스 뒤에서 수행되는 복잡한 프로세스를 감추는데 중점을 두고 있다. 해당 프로그램을 설계하면서 컴퓨터 구조론의 핵심적인 내용이 되는 폰 노이만 아키텍처에 대한 이해를 심화시키고, 이를 통해 컴퓨팅의 기본 원리와 메커니즘에 대한 명확한 이해를 바탕으로 실용적인 소프트웨어 솔루션을 개발하는 역량을 키울 수 있었다. 이를 통해 이론적 지식을 실제 문제 해결에 적용할 수 있는 능력을 키울 수 있었다.

1. 폰노이만 아키텍처란

본 계산기 프로그램은 폰노이만 아키텍처의 핵심 원칙을 반영하여 설계 되었기에 제작한 계산기 프로그램에 대한 설명에 앞서 폰 노이만 아키텍처에 대한 간단한 설명을 거쳐 제작한 계산기 프로그램에 대한 설명을 이어나갈 예정이다.

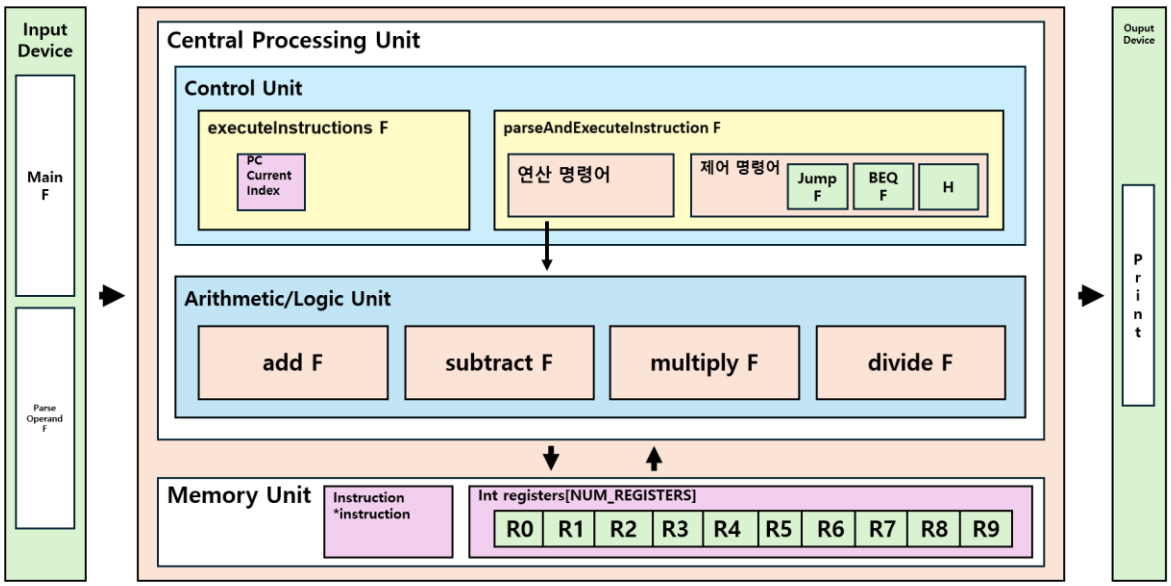


폰 노이만 아키텍처[1]

폰 노이만 아키텍처는 현대 컴퓨터 설계의 근간을 이루는 컴퓨팅 시스템의 구조적 디자인이다. 이 아키텍처는 1945 년 존 폰 노이만에 의해 처음으로 개념화되었으며, 그의 이름을 따서 명명되었다. 폰 노이만 아키텍처의 주요 특징 중 하나는 프로그램 내장 방식이다. 이 방식에서는 실행할 프로그램과 프로그램이 처리할 데이터가 같은 메모리에 저장된다. 이 구조의 핵심 구성 요소로는 중앙 처리 장치(CPU), 메모리, 입출력 장치가 있다. 중앙 처리 장치는 제어 장치(CU)와 산술 논리 연산 장치(ALU)의 두 부분으로 구성된다. 제어 장치는 메모리에 저장된 명령어들을 순차적으로 해석하고, 해당 명령을 실행하기 위해 필요한 제어 신호를 발생시킨다. 산술 논리 연산 장치는 다양한 산술 연산과 논리 연산을 수행하여 처리 속도를 높인다. 메모리 장치는 명령어와 데이터를 저장하며, CPU 가 접근할 수 있는 주소 공간을 제공한다. 입출력 장치는 사용자와 컴퓨터 사이의 상호작용을 가능하게 하며, 외부 장치로부터 데이터를 받아들이고 처리 결과를 전달한다. 폰 노이만 아키텍처의 또 다른 중요한 측면은 프로그램 카운터이다. 프로그램 카운터는 현재 CPU 가 실행하고 있는 명령어의 위치를 가리키며, 명령어가 실행될 때마다 업데이트된다. 이러한 구조는 프로그램의 순차적 실행을 가능하게 하며, 복잡한 프로그램의 효율적인 실행에 기여한다. 전체적으로 폰 노이만 아키텍처는 명령어 사이클을 통해 명령어를 가져오고(페치), 디코드하며, 실행하는 일련의 과정을 체계적으로 정의한다. 이 아키텍처는 단순성과 범용성을 제공함으로써, 현대 컴퓨팅 기술의 발전에 결정적인 역할을 하였다.

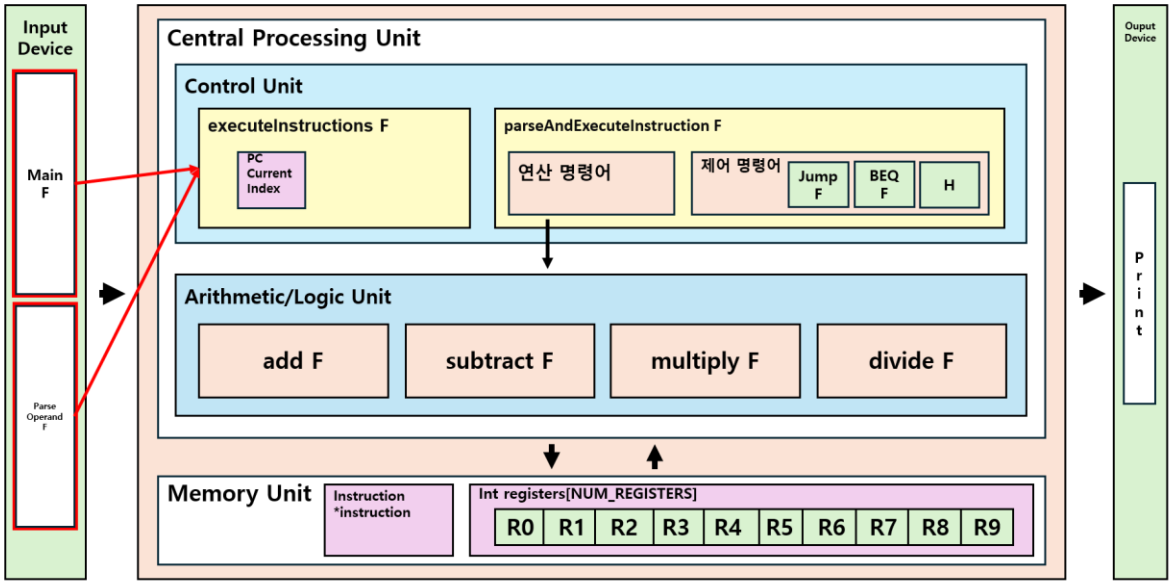
2. 본 Simple Calculator 의 폰 노이만 아키텍처

본 Simple Calculator 또한 이전의 폰 노이만 아키텍처를 따르고 있다.



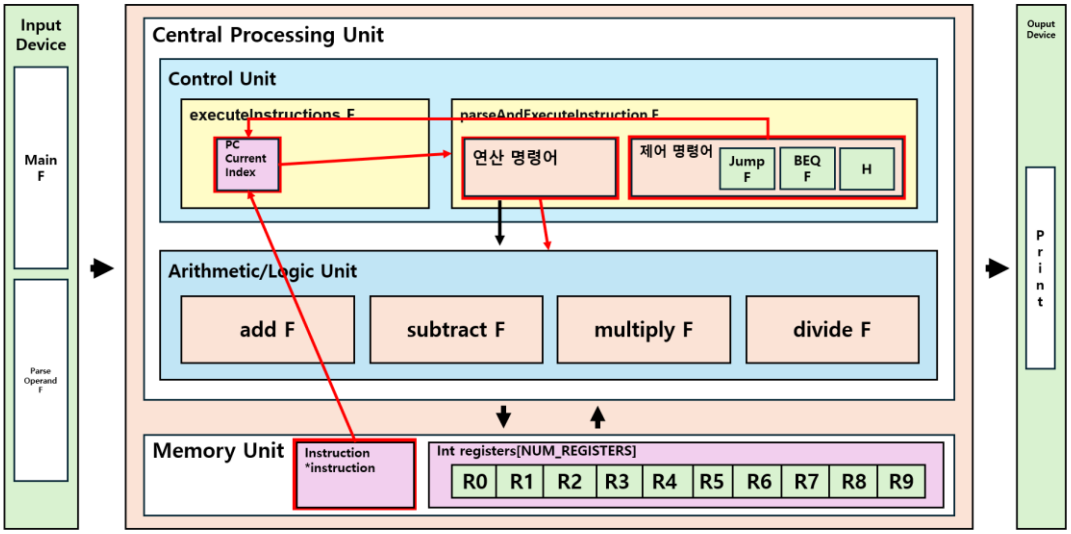
본 Simple Calculator 의 폰 노이만 아키텍처

본 계산기 프로그램은 폰 노이만 아키텍처의 기본 원리를 구현한 소프트웨어로서, 사용자로부터의 입력과 프로그램 내 데이터 처리 과정에 있어 main 함수와 parseOperand 함수를 중심으로 입력 장치(Input Device)의 역할을 수행한다. main 함수는 사용자로부터 명령어가 포함된 파일을 입력 받아, 해당 파일의 경로를 executeInstructions 함수에 전달한다. 이 과정에서 main 함수는 파일 시스템을 통해 프로그램에 명령어를 제공하는 입력 장치의 기능을 모방한다. executeInstructions 함수는 전달받은 파일 경로를 통해 명령어를 읽고, 프로그램의 메모리에 로드하는 과정을 담당한다. 이는 사용자로부터 받은 입력 데이터를 프로그램이 처리할 수 있는 형태로 변환하는 과정이다. parseOperand 함수는 명령어 내의 피연산자를 분석하여, 레지스터 참조인지 또는 직접적인 값을 나타내는지를 판단하고, 이를 적절한 데이터 형태로 변환하는 역할을 한다. 피연산자가 레지스터를 지칭하는 경우, 해당 레지스터의 실제 값을 반환하고, 직접 값을 나타내는 경우에는 이를 16 진수 값으로 변환하여 반환한다. 이러한 과정을 통해, 본 프로그램은 사용자로부터의 입력을 받아들이고, 이를 디코드 및 실행하는 CPU 의 기능을 소프트웨어적으로 구현한다. 따라서, main 함수와 parseOperand 함수는 본 계산기 프로그램에서 입력 장치의 중요한 역할을 담당한다고 할 수 있다.



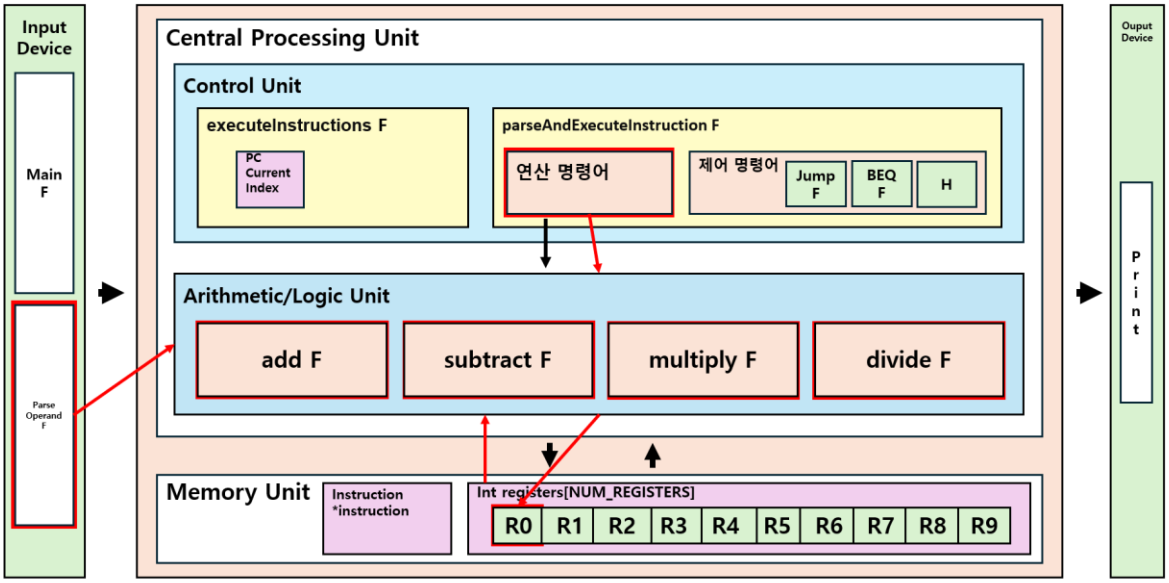
Input Device 의 역할을 하는 Main, ParseOperand 함수

폰 노이만 아키텍처는 컴퓨터의 두뇌 역할을 하며, 모든 데이터 처리 작업을 담당한다. 이는 프로그램의 명령어를 해석하고 실행하는 주요 구성 요소인 Central Processing Unit 과 Memory Unit 으로 나뉘고 CPU 는 또 다시 Control Unit 과 Arithmetic/Logic Unit 으로 나뉜다. 그중에서도 Control Unit(제어 장치)은 명령어를 해석하고, 데이터와 연산의 준비를 위해 ALU 나 메모리에 신호를 보낸다. 본 계산기 프로그램에서는 executeInstructions 함수와 parseAndExecuteInstruction 함수가 Control Unit 의 역할을 수행하고 있다. 먼저 executeInstructions 함수는 가져온 각 명령어를 해석하고 실행하는 역할을 한다. 그중에서도



Control Unit 의 역할을 하는 함수들

CU 가 할일 중 하나인 Program counter 를 관리하여 현재 실행해야 할 명령어의 위치를 알고 있게 하는 것인데 함수내의 currentIndex 변수는 다음에 실행할 명령어의 메모리 주소를 저장하는 Program Counter 의 역할을 하여 명령어 배열에서 순차적으로 명령어를 가져와 다음에 어떤 명령어가 실행될지를 parseAndExecuteInstruction 함수에 전달한다. parseAndExecuteInstruction 함수는 가져온 각 명령어를 해석하고 실행하는 역할을 한다. CU 는 명령어의 종류를 해석하여 해당 명령어가 수행하여야 할 연산의 종류를 결정하고, 필요한 경우 ALU 나 다른 시스템 부분에 연산을 지시한다. 해당 함수는 이러한 과정을 코드 내에서 구현하고 있다. 전달받은 명령어 문자열을 분석하여 연산자와 피연산자를 추출하고 연산자의 종류에 따라 해당하는 연산 함수(add, subtract 등)를 호출하거나 제어 흐름(jump, branchIfEqual 등)을 변경하는 명령을 처리한다. 이 가운데서도 연산 명령어는 CU 가 ALU 에 연산을 지시하는 것과 유사하게 연산을 수행하는 함수를 호출하여 실제 연산을 실행한다. 또한 제어 명령어의 경우 프로그램의 실행 흐름을 변경하는데 예를 들어 jump 명령어는 currentIndex 의 값을 변경하여 다음에 실행될 명령어의 위치를 바꾸고 H 명령어는 프로그램의 실행을 종료한다. 이 과정에서 parseAndExcuteInstruction 함수는 명령어의 해석 및 실행 과정을 총괄하는 CU 의 역할을 수행하며 프로그램의 명령어에 따라 CPU 내부의 연산 실행 또는 프로그램 실행 흐름의 제어와 같은 작업을 지시한다.



Arithmetic/Logic Unit 의 역할을 하는 함수들

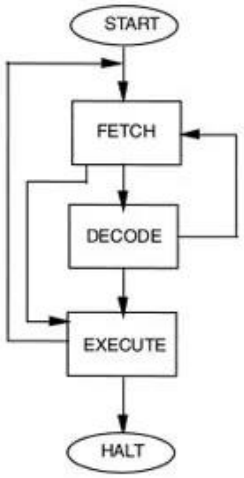
산술 논리 연산 장치(ALU, Arithmetic Logic Unit)는 컴퓨터의 CPU 내에 위치한 핵심적인 구성 요소 중 하나로, 모든 종류의 산술 연산(예: 덧셈, 뺄셈)과 논리 연산(예: AND, OR, NOT)을 수행한다. ALU 는 프로그램 실행 중 발생하는 다양한 계산과 데이터 처리 작업을 담당한다. 해당 프로그램에서는 add 함수가 두 피연산자의 덧셈 연산을 수행하여 registers[0]에 저장하고 subtract 함수가 두 피연산자의 뺄셈 연산을 수행하여 registers[0]에 저장한다. 또한 multiply 함수는 두 피연산자의 곱셈 연산을 수행하여 registers[0]에 저장하고 divide 함수는 두 피연산자의 나눗셈 연산을 수행하여 registers[0]에 저장한다. 이를 통해 ALU 의 기본 기능 중 하나인 산술 연산을 진행할 수 있다. 이 함수들은 모두 프로그램의 명령어에 따라 특정 연산을 수행하고 그 결과를 레지스터에 저장하는 ALU 의 기본 작동 원리를 단순화 하여 구현하고 있다. 코드 내에서 이러한 연산 함수들의 호출은 parseAndExecuteInstruction 함수를 통해 제어되며 이는 ALU 가 제어 장치 CU 의 지시에 따라 연산을 수행하는 과정과 유사하다.

폰 노이만 구조에서 Memory Unit 은 컴퓨터 시스템에서 중요한 역할을 한다. 프로그램의 명령어와 데이터를 저장하며, CPU 는 이 메모리로부터 명령어를 읽어 실행하고, 필요한 데이터를 저장하거나 검색한다. 이러한 메모리는 명령어와 데이터가 동일한 물리적 메모리 공간을 공유하는 특징을 가지고 있습니다. 프로그램 내에서 이 메모리 구조는 Instruction *instructions 배열과 int registers[NUM_REGISTERS] 배열을 통해 구현되어 있다. Instruction *instructions 배열은 파일로부터 읽어들인 프로그램의 모든 명령어를 순차적으로 저장한다. 이 구조는 폰 노이만 구조에서 메모리가 프로그램 명령어를 저장하는 방식을 반영한다. CPU, 이곳에서 parseAndExecuteInstruction 함수는 이 배열에서 명령어를 하나씩 읽어 해석하고 실행한다. 또한 int registers[NUM_REGISTERS] 배열은 데이터를 저장한다. 특히 CPU 의 레지스터를 모방하여 연산에 필요한 데이터 값을 저장하거나 연산 결과를 임시로 저장하는 용도로 사용된다. 각 레지스터는 배열의 한 요소로 표현되며 NUM_REGISTERS 는 사용 가능한 레지스터의 수를 정의한다. 프로그램 내에서

연산 명령어는 이 배열의 요소를 사용하여 연산을 수행하고 연산 결과를 저장한다. 이를 통해 폰 노이만 구조에서 CPU 가 메모리로 부터 데이터를 읽어 레지스터에 저장하고 연산 후 그 결과를 다시 메모리에 저장하는 과정을 보여줄 수 있다. 이를 통해 폰 노이만 구조의 핵심 특징 중 하나인 명령어와 데이터의 메모리 내 통합 저장을 실현할 수 있다.

여기에 더해서 move 함수는 폰 노이만 아키텍처 내에서 주로 CPU 의 작업에 해당한다. 이 함수는 데이터를 한 위치에서 다른 위치로 전송하는 기본적인 연산을 수행하는데 이는 두 가지의 주요 작업에 걸쳐 있다. 먼저 move 함수의 해석 및 실행 지시는 CU 에 의해 이루어진다. CU 는 명령어를 해석하여 move 연산이 필요함을 인지하고 해당 연산을 수행하기 위한 지시를 내린다. 이는 명령어의 decode 과정에 해당하며 어떤 데이터를 어느곳으로 이동시킬지 결정하는 데 필요한 과정을 포함한다. 또한 move 연산이 산술 논리 연산을 직접적으로 수행하지는 않지만 ALU 는 데이터 이동과 관련된 연산을 처리할 수 있는 능력을 가지고 있다. 특히 데이터를 레지스터 간에 이동시키거나 메모리에서 레지스터로 데이터를 로드하는 등의 작업을 포함할 수 있다. 즉 move 함수는 CU 의 명령어 처리 매커니즘과 ALU 의 데이터 관리 기능을 통해 구현되며 이러한 과정 전체가 폰 노이만 아키텍처의 핵심 원리를 반영하고 있다.

3. 명령어 사이클이란

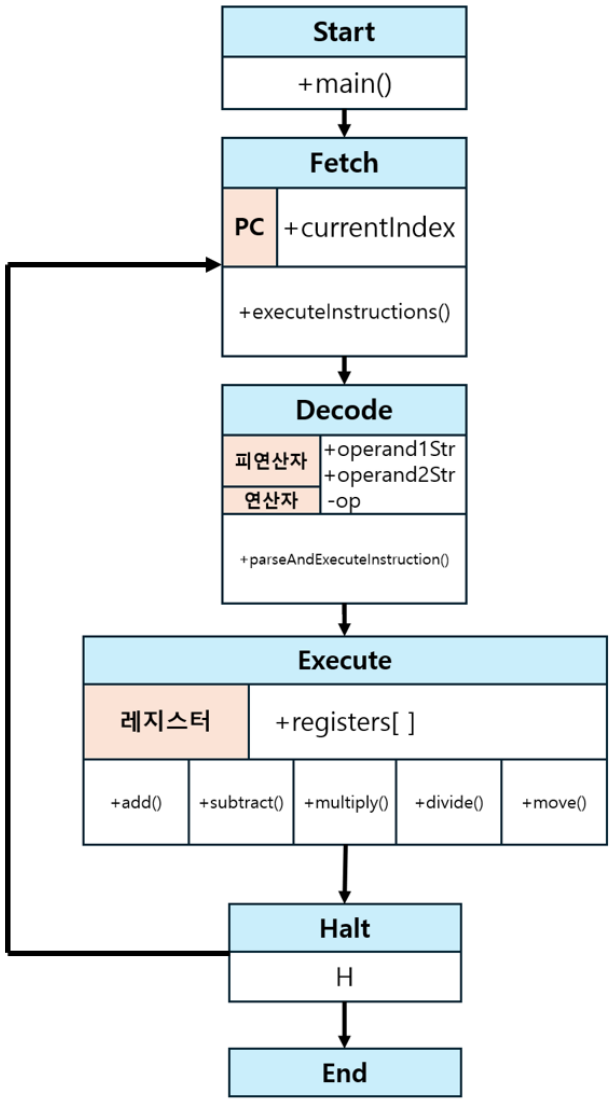


명령어 사이클 Instruction cycle[2]

이전에 폰 노이만 아키텍처에 관한 설명을 할때 '폰 노이만 아키텍처는 명령어 사이클을 통해 명령어를 가져오고(fetch), 디코드(decode), 실행(execute)하는 일련의 과정을 체계적으로 정의한다.' 라고 하였다. 폰 노이만 아키텍처 하에서의 명령어 사이클은 컴퓨터 시스템이 프로그램을 실행하는 기본적인 과정을 정의한다. 이 사이클은 명령어 Fetch, Decode, Execute, 의 세 가지 주요 단계로 구성된다. 이 과정을 통해 컴퓨터는 복잡한 작업을 효율적으로 처리할 수 있다. 첫 번째 단계인 fetch 단계에서는 Program counter 가 가리키는 메모리 주소에서 다음 실행할 명령어를 CPU 로 가져온다. 이 단계가 완료되면 Program counter 는 자동적으로 다음 명령어의 주소로 업데이트된다. 다음으로 decode 단계에서는

CPU 내의 CU 가 fetch 된 명령어를 해석한다. 명령어의 종류를 판별하고 필요한 연산과 피연산자를 식별하여 명령어의 실행을 위한 모든 준비를 마친다. 마지막 execute 단계에서는 CU 에 의해 해석된 명령어에 따라 ALU 가 적절한 연산을 수행한다. 이 단계에서는 산술 연산, 논리 연산, 데이터 이동, 조건 분기 등이 실행될 수 있다. 이러하듯이 폰 노이만 아키텍처의 명령어 사이클은 컴퓨터가 복잡한 프로그램을 순차적이고 체계적으로 실행할 수 있는 기반을 마련한다. 이러한 사이클의 효율적인 관리와 실행은 컴퓨터의 전반적인 성능과 처리 속도를 결정하는 핵심 요소이다.

4. 본 Simple Calculator 의 명령어 사이클



본 Simple Calculator 의 명령어 사이클 Instruction cycle

본 계산기 프로그램 또한 이러한 폰 노이만 아키텍처의 명령어 사이클을 효과적으로 구현하며, 다음은 본 프로그램의 input 파일로부터 입력받은 명령어들을 처리하고 연산

결과를 제공하는 과정을 자세히 설명한다. 프로그램의 실행은 main 함수에서 시작되며, 지정된 파일 경로에서 명령어를 읽어들이는 작업으로 진행된다. 이어, executeInstructions 함수가 파일을 열어 내용을 instructions 배열에 순차적으로 저장함으로써, 명령어 fetch 단계가 수행된다. 해당 배열에 저장된 명령어는 program counter 역할을 하는 currentIndex 에 의해 차례대로 접근되며, 이 변수를 통해 배열에서 명령어를 읽어온 후, 자동으로 다음 명령어의 위치를 가리키도록 증가한다. 명령어가 패치된 후, decode 를 진행하는 parseAndExecuteInstruction 함수는 currentIndex 에 의해 지정된 instructions 배열 내의 명령어를 가져와서 연산자와 피연산자로 분리한다. 그후 해당 명령어(연산자)를 해석한다. 이 단계에서 함수는 명령어의 종류를 결정하고, 필요한 연산과 관련된 피연산자(operand1Str, operand2Str)를 식별한다. 명령어의 종류와 필요한 데이터가 식별된 후, 연산을 수행하는 함수(예: add, subtract 등)가 호출된다. 다음으로 execute 단계에서 디코드된 명령어에 따라, 적절한 연산을 수행하는 함수(예: add, subtract, multiply, divide, move 등)가 호출된다. 이러한 함수들은 ALU 의 역할을 모방하여, 주어진 연산을 실행하고 연산 결과를 레지스터(registers[] 배열)에 저장한다. 이 과정에서 parseOperand 함수가 피연산자를 적절한 형태(레지스터의 값 또는 직접 값)로 변환하여 연산에 사용할 수 있게 한다. 또한 명령어가 메모리 접근을 요구하는 경우, move 명령어를 통해, 이 단계에서 데이터는 메모리와 레지스터 사이에서 전송된다. 이를 메모리 접근(Memory Access)이라고 한다. 이러하듯이 계산기 프로그램은 폰 노이만 아키텍처의 기본 원칙을 효과적으로 모방하여 명령어 사이클을 구현한다. 이러한 구현은 프로그램이 사용자로부터 입력받은 명령어들을 순차적으로 처리하고, 연산 결과를 도출하는 과정에서 중요한 역할을 한다.

5. 본 Simple Calculator 의 주요 기능과 추가 구현 기능

다음으로 프로그램 내에서 구현한 주요 기능에 대한 설명을 이어가도록 하겠다. 먼저 Program counter 의 역할을 하는 currentIndex 에 대해 이야기 하자면

```
while (currentIndex < numInstructions) {
    strncpy(inst_reg, instructions[currentIndex].instruction, MAX_INSTRUCTION_LENGTH);
    parseAndExecuteInstruction(inst_reg, &currentIndex, numInstructions);
    currentIndex++;
}
```

변수에 대한 설명

1. currentIndex: Program counter 의 역할을 하는 변수로 현재 실행 중인 명령어의 인덱스를 나타낸다. 이 값은 프로그램이 실행될 때 0 부터 시작하여, 프로그램이 각 명령어를 실행할 때마다 1 씩 증가한다.
2. numInstructions: 프로그램에 주어진 총 명령어의 수를 나타내는 변수이다.
3. inst_reg: 현재 처리할 명령어를 임시로 저장하는 문자열 변수이다. 이 변수는 parseAndExecuteInstruction 함수로 전달되어 해당 명령어가 실행된다.
4. instructions: 프로그램이 실행할 명령어들을 저장하는 배열이다.

코드에 동작에 대한 설명

1. while 루프는 currentIndex 가 numInstructions 보다 작은 동안 계속 실행된다. 이는 아직 실행되지 않은 명령어가 남아있다는 것이다.
2. 루프 내에서 strncpy 함수를 사용하여 instructions 배열에서 현재 인덱스(currentIndex)에 해당하는 명령어를 inst_reg 변수에 복사한다. 이는 현재 실행할 명령어를 임시 변수에 저장하는 과정이다.
3. parseAndExecuteInstruction 함수는 inst_reg 에 저장된 명령어를 인자로 받아 해당 명령어를 해석하고 실행한다. 이 함수는 명령어의 종류에 따라 적절한 연산을 수행하거나 프로그램의 흐름을 제어한다.
4. 명령어가 성공적으로 처리되면, currentIndex 는 1 증가하여 다음 명령어로 이동한다. 이 과정은 프로그램이 모든 명령어를 순차적으로 처리할 때까지 반복된다.

이러한 방식을 통해 currentIndex 는 프로그램 내에서 Program Counter 의 역할을 수행할 수 있으며 Increment PC 를 구현할 수 있다.

다음은 이 프로그램에서 필요한 기능에 대한 설명이다.

명령어	명령어 기능	명령어에 대한 설명	필요 함수
+	덧셈 명령어	두 값의 합을 더하여 결과를 저장	add( )
-	뺄셈 명령어	첫 번째 피연산자에서 두 번째 피연산자를 뺀 값을 저장	subtract( )
*	곱셈 명령어	두 피연산자의 곱을 계산	multiply( )
/	나눗셈 명령어	첫 번째 피연산자를 두 번째 피연산자로 나눈 결과를 저장	divide( )
M	이동 명령어	한 레지스터의 값을 다른 레지스터로 이동시키거나, 특정 값으로 레지스터를 업데이트	move( )
C	비교 명령어	두 피연산자를 비교하고 그 결과에 따라 조건부 실행	compare( )
J	점프 명령어	프로그램의 다른 부분으로 점프하여 실행	jump( )
B	분기 명령어	특정 조건이 만족될때만 프로그램의 다른 부분으로 점프하여 실행	branchIfEqual( )
H	중단 명령어	프로그램의 실행을 종료	H

이전의 기능을 작동하기 위해 다음의 함수를 사용하여 프로그램 내에서 구현했다.

add(), subtract(), multiply(), divide() 함수 구현

```
void add(int operand1, int isRegister1, int operand2, int isRegister2) {
    int result = operand1 + operand2;
    registers[0] = result;
    printf("R0: %d = %d + %d\n", result, operand1, operand2);
}
```

기본 연산에 관한 함수는 피연산자 1 과 피연산자 2 를 함수의 인자로 받아 각각의 함수에 맞는 연산을 진행한 후 R0 레지스터에(배열)에 저장하였다.

move() 함수 구현

```
if (!isRegister1) {
    printf("Error: First operand must be a register.\n");
    return;
}

int num1 = number[0];
int num2 = number[1];

if (isRegister2) {
    registers[num1] = registers[num2];
} else {

    registers[num1] = operand2;
}
```

move() 함수의 경우 첫 번째 피연산자가 레지스터가 아닌 경우 오류를 출력하도록 하였고 두 번째 피연산자가 레지스터인 경우 두 번째 피연산자의 레지스터의 값을 첫 번째 레지스터로 값을 이동, 두 번째 피연산자가 레지스터가 아닌 경우 두 번째 피연산자의 값을(16 진수)를 첫 번째 레지스터의 값으로 저장하도록 구현하였다.

compare() 함수 구현

```
if (operand1 == operand2) {
    result = 0;
} else if (operand1 > operand2) {
    result = 0;
} else {
    result = 1;
}
registers[0] = result;
```

compare() 함수의 경우 첫 번째 피연산자가 두 번째 피연산자 보다 크거나 같을때 R0 에 0 을 저장, 아닌 경우 R0 에 1 을 저장하도록 구현하였다.

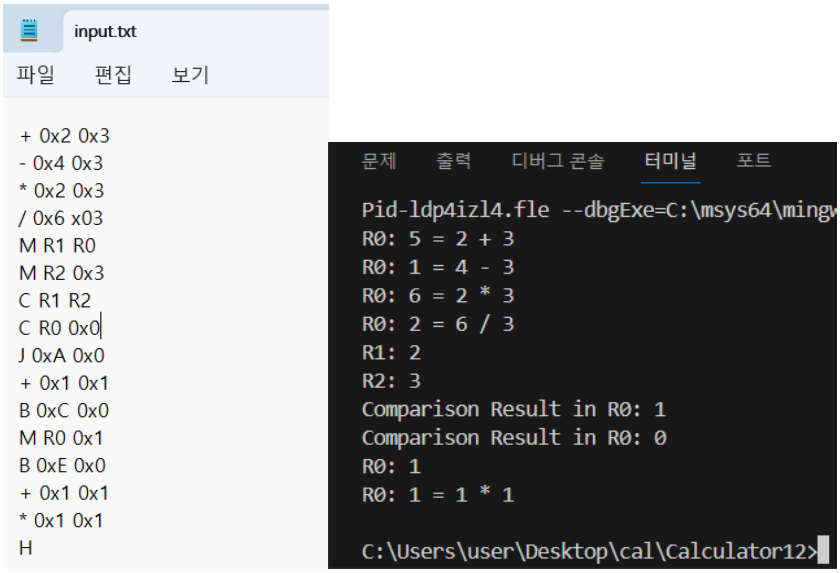
jump(), branchIfEqual() 함수 구현

```
void branchIfEqual(int lineNumber, int *currentIndex, int numInstructions) {
    if (registers[0] == 1 && lineNumber - 1 < numInstructions) {
        *currentIndex = lineNumber - 1;
    }
}
```

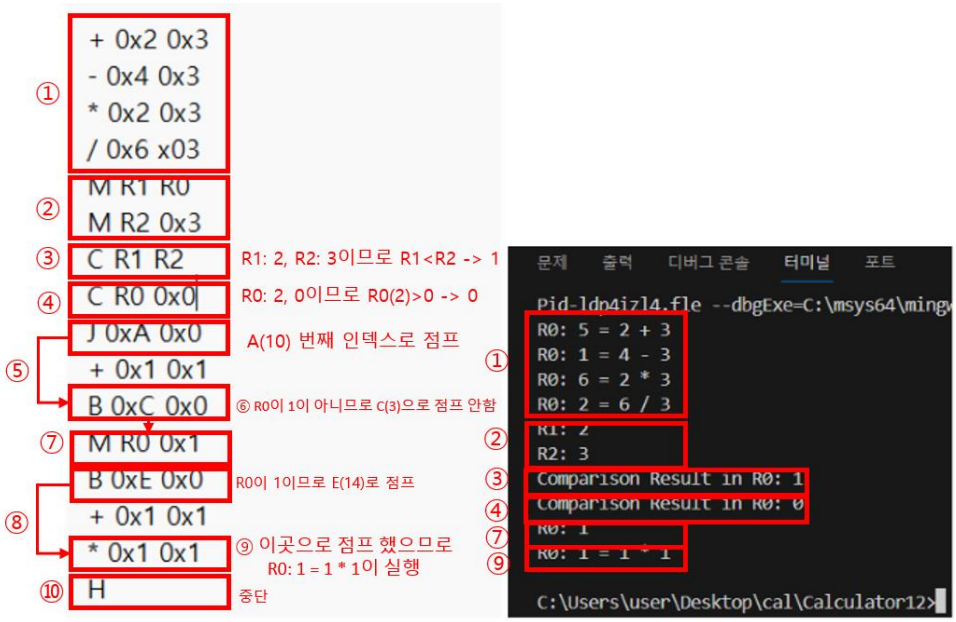
jump(), branchIfEqual() 함수의 경우 앞서 구현했던 Program Counter 의 역할을 하는 currentIndex 변수를 operand1(첫 번째 피연산자) -1 의 위치로 이동시켜주면 된다. 여기서 branchIfEqual()의 경우 R0 레지스터의 값이 1 인 경우에만 해당 위치로 점프를 해야하기 때문에 if(registers[0]==1)의 조건을 추가해준다.

6. 실행 결과

다음과 같이 제대로 실행이 됨을 알 수 있다.

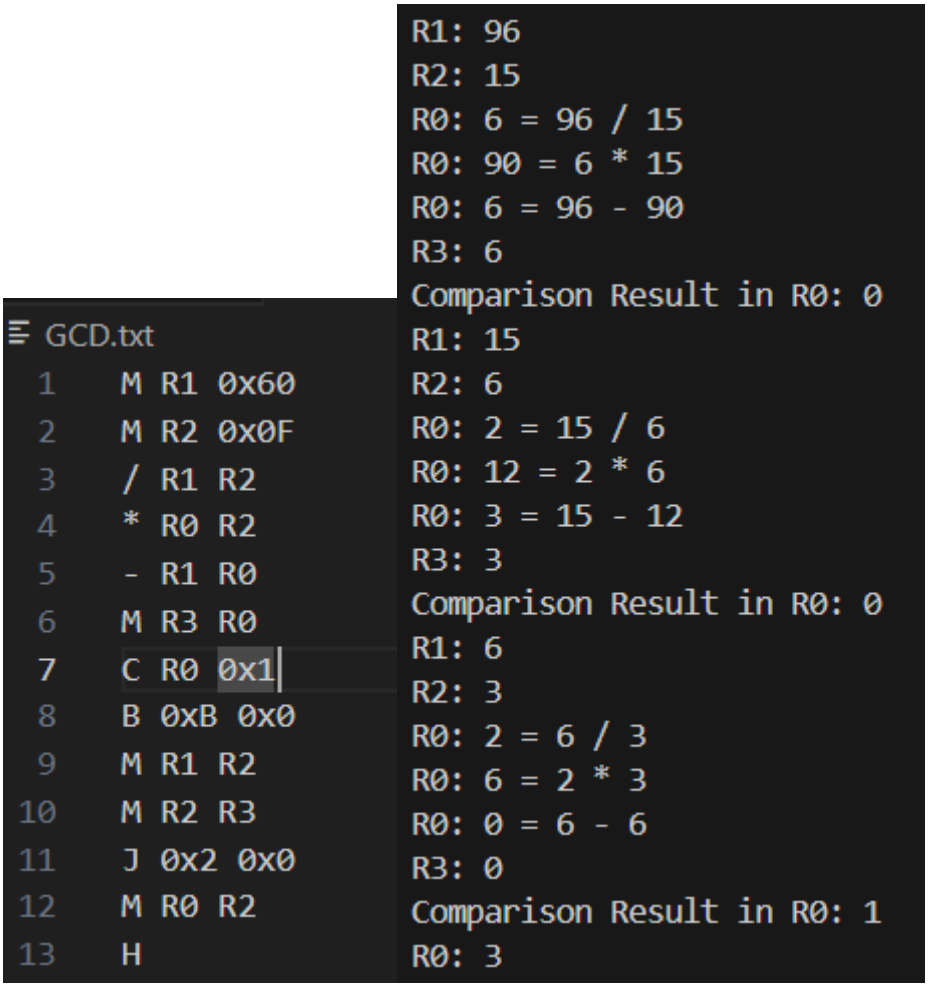


input 파일과 실행 결과



실행 결과에 대한 해석

7. GCD 구현



GCD input 파일과 실행 결과

최대 공약수를 계산하기 위해 가장 유명한 최대 공약수 계산 알고리즘인 유클리드 알고리즘을 사용했다. 이 알고리즘은 다음과 같이 작동한다.

1. 두 수 a 와 b 에서 a 가 b 보다 크다고 가정한다. (만약 a 가 b 보다 작다면 둘의 위치를 바꾼다.)
2. a 를 b 로 나눈 나머지를 계산한다.
3. 만약 나머지가 0 이라면 b 가 최대공약수이다.
4. 그렇지 않다면 a 에 b 를 할당하고 b 에 나머지를 할당한 다음, 2 번의 단계로 돌아간다.

문제를 해결하기 위해 유클리드 호제법 알고리즘을 활용한 나만의 알고리즘을 고안했다.

1. 현재의 경우 96 과 15 라는 수가 이미 주어졌기 때문에 a 가 b 보다 크다는 것이 이미 밝혀졌으므로 1 번의 과정을 넘어가도록 한다.
2. 나머지를 계산하는 명령어가 부재하므로 나머지를 계산하는 알고리즘을 고안한다.

   이를 위해 나는 큰 수 a 를 작은 수 b 로 나눈 후 나온 몫에 b 를 다시 곱하고 이 수를 다시 큰 수 a 에 빼주어 나머지를 구하도록 하였다.

나머지: n, 큰 수: a, 작은 수: b (다음에서 a/b 는 소수점을 제외한 정수의 값이 나온다고 가정)

$$n = (a - (a / b) * b)$$

3.

(1) 만약 A = 0 이면 ~다 를 작동하는 함수가 없으므로 C 명령어의 op1 < op2 면 R0 은 1 이다를 이용한다. 즉 나머지가 0 인 경우(나머지가 0 보다 작은 경우는 없다.) 나머지 < 1 이므로 C R0 0x1 를 이용하여 문제를 해결한다. 이 경우 R0 에 1 이 입력되게 된다.

(2) R0 에 1 이 입력되는 경우 B 명령어를 통해 R0 이 1 이면(나머지가 0 이면) 다음의 4 번 코드를 실행하지 않고 반복문의 밖으로 점프한다.

(3) 이후 R0 에 R2(b)의 값을 M 명령어를 통해 이동시켜주면 R0 에 최대공약수가 입력이 된다.

4. M 명령어를 통해 R1(a)에 R2(b)를 할당하고 사전에 저장해둔 R3(나머지)를 R2(b)에 할당해준다. 이후 다시 2 번의 단계로 J 명령어를 통해 점프(반복)한다.

이러한 알고리즘 및 이를 구현한 input 명령어를 통해 R0 에 최대 공약수를 구할 수 있었다.

[결론]

본 프로젝트를 통해 폰 노이만 아키텍처 기반의 명령어 사이클을 구현하면서, 컴퓨터 과학의 기본적인 원리와 개념을 실제로 적용해보는 경험을 할 수 있었다는 점에서 큰 의미가 있었다. 이러한 과정을 통해, 단순히 이론적인 지식을 넘어서서 실제 소프트웨어 개발 과정에서 중요한 아키텍처의 동작 원리를 이해하고, 이를 바탕으로 한 프로그램의 구현 과정을 체험할 수 있었다. 또한, 다양한 연산을 수행하는 명령어들을 설계하고, 이를 해석하여 실행하는 로직을 개발함으로써, 프로그래밍 능력과 문제 해결 능력을 향상 시킬 수 있었다. 특히, 명령어에 따라 적절한 함수를 호출하여 연산을 수행하고 결과를 저장하는 과정에서는, 프로그램의 흐름을 제어하는 데 필요한 핵심적인 기술들을 배울 수 있었다. 이번 프로젝트를 통해 얻은 경험은 향후 복잡한 소프트웨어 시스템을 설계하고 구현하는 데 있어 소중한 자산이 될 것이라고 생각한다.

[참고 문헌]

[1]폰노이만 구조, 위키백과

https://ko.wikipedia.org/wiki/%ED%8F%B0_%EB%85%B8%EC%9D%B4%EB%A7%8C_%EA%B5%AC%EC%A1%B0#CITEREFGanesan2009

[2]명령어 사이클, CPU/구조와 원리(r46 판)