<프로젝트1 보고서>

컴퓨터구조와 모바일프로세서

간단한 계산기 리포트

학　　과 : 모바일시스템공학과
학　　번 : 32164809
이　　름 : 탁준석
담당교수 : 유시환
제출일자 : 21.03.21
남은 프리데이 : 5일

목 차

1. 개요 3

가. 실험 목표

나. 실험 목적

2. 이론 및 적용 4

가. 폰 노이만 구조

나. 이론의 적용

3. 간단한 계산기 5

가. 프로그램 구조

나. 문제와 해결

다. 개발 환경

라. 결과

4. 고찰 및 느낀 점 8

5. 참고문헌 9

1. 개요

가. 실험 목표

간단한 MIPS ISA로 구성된 텍스트 파일을 읽어 표 1.1과 같은 명령을 처리하는 나만의 CPU 프로그램을 만든다.

기본 명령어	추가 명령어
산술 명령어(Arithmetic Instruction, +-*/)	점프 명령어(Jump Instruction, J)
이동 명령어(Move Instruction, M)	비교 명령어(Compare Instruction, C)
정지 명령어(Halt Instruction, H)	분기 명령어(Branch Instruction, B)

<표 1.1> 기본 명령어와 추가 명령어

나. 실험 목적

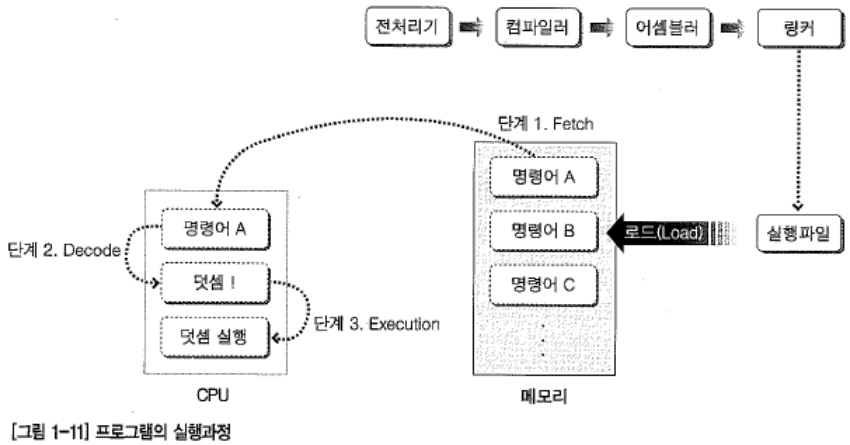
현대 컴퓨터는 폰 노이만 구조를 바탕으로 설계되어 있다. 컴퓨터 구조의 근본적인 논리를 이해하는 것은 하드웨어 개발자뿐만 아니라 소프트웨어 개발자에게도 필수적이다. 폰 노이만 구조의 중앙처리장치, 메모리, 입출력 장치의 관계를 학습하고, 학습한 내용을 바탕으로 간단한 계산기 프로그램을 만들어 중앙처리장치와 메모리가 어떻게 명령을 처리하는지 자세히 이해한다.

2. 이론 및 적용

가. 폰 노이만 구조

폰 노이만 구조의 컴퓨터는 '프로그램의 저장'과 '순차적 실행'이라는 특징을 가진 구조로 크게 중앙처리장치(CPU), 메모리, 입출력 장치로 구성되어있다. 그림 2.1과 같이 중앙처리장치(CPU)는 제어장치와 산술논리 연산장치, 그리고 레지스터로 구성되어있고, 메모리는 명령어와 데이터로 구성되어있다. 프로그램의 실행과정은 그림 2.1과 같다. 컴파일러와 어셈블러를 통해 2진수 코드로 바뀐 실행 파일이 순서대로 메모리에 로드된다. 메모리에 로드된 명령은 CPU의 Fetch, Decode, Execute의 세 단계를 거처 순서대로 처리된다.

- Fetch : 메모리에 로드된 명령을 CPU의 명령 레지스터(IR)에 저장한다.
- Decode : 저장한 명령어가 무엇인지 해석한다.
- Execute : 해석한 명령어의 명령대로 계산을 실행한다.



<그림 2.1> 프로그램의 실행과정

나. 이론의 적용

폰 노이만 구조를 바탕으로 메모리는 텍스트 파일을 통해 구현하고, 중앙처리장치는 간단한 계산기 프로그램을 통해 구현하기로 했다. 먼저, 실행할 프로그램의 역할을 할 텍스트 파일을 작성했다. 컴파일한 프로그램에는 명령어와 데이터로 이루어진 명령이 순서대로 저장되어 있다는 것을 바탕으로 기본 명령어 6개와 추가 명령어 3개, 그리고 레지스터 10개를 사용해 두 수의 최대 공약수를 구하는 프로그램을 작성했다.

다음으로 프로그램이 로드된 메모리의 역할을 할 코드를 작성했다. 중앙처리장치의 명령 레지스터(IR)와 명령 포인터(IP)를 사용하기 위해서 명령어와 데이터가 로드된 메모리가 필요했다. 이를 위해 텍스트 파일의 명령어와 데이터를 복사해 구조체에 저장했고, 생성된 구조체들을 선형 연결 리스트로 순서대로 연결했다. 그 과정에서 표 2.1의 함수들

이 사용되었다.

fopen_s	input.txt를 연다.
fclose	input.txt를 닫는다.
fgets	input.txt에 작성된 명령을 한 줄씩 읽는다.
strtok_s	명령을 공백을 기준으로 문자열 Opcode, Operand1, Operand2로 구분한다.
strcpy_s	문자열 Opcode, Operand1, Operand2를 구조체에 복사한다.

<표 2.1> 메모리 역할에 사용된 함수

마지막으로, 중앙처리장치 역할을 할 간단한 계산기 프로그램을 작성했다. 중앙처리장치는 Fetch, Decode, Execute 과정을 반복하며 명령을 처리하므로 각 과정에 해당하는 함수를 설계했다. 또한, 명령 레지스터(IR)와 명령 포인터(IP)를 사용해 실행할 명령을 저장하고, 다음 명령의 주소를 가리킬 수 있도록 했다. 그 과정에서 표 2.2의 함수들이 사용되었다.

atoi	레지스터 Operand의 번호를 10진수 정수로 바꾼다.
strtol	16진수 Operand를 10진수 정수로 바꾼다.
strcmp	Opcode와 명령어를 직접 비교한다.

<표 2.2> CPU 역할에 사용된 함수

3. 간단한 계산기

가. 프로그램 구조

1) ReadText();

사용자가 작성한 MIPS ISA 텍스트 파일을 열고 fgets 함수로 명령을 순서대로 읽는다. strtok 함수를 사용해 읽은 명령을 공백을 기준으로 세 문자열로 나눈다. 첫 번째 문자열은 Opcode, 두 번째 문자열은 Operand1, 그리고 마지막 문자열은 Operand2에 저장하고 ReadMemory 함수의 인자로 넘긴다. 이와 같은 방법으로 텍스트 파일의 첫 명령부터 마지막 명령까지 LoadMemory 함수로 인자를 전달하고 텍스트 파일을 닫는다.

2) LoadMemory();

새로운 노드를 하나 만들고 strcpy_s 함수를 사용해 전달받은 Opcode, Operand1, Operand2를 새로운 노드에 저장한다. 텍스트 파일의 첫 명령부터 마지막 명령까지 전달받은 Opcode, Operand1, Operand2를 모두 노드에 저장하고, 선형 연결 리스트 구조에 따라 순서대로 연결한다.

3) FetchStage();

함수 FetchStage는 CPU의 Fetch 단계를 실행한다. 명령 포인터(IP)가 가리키는 노드의 Opcode, Operand1, Operand2를 복사해 명령 레지스터(IR) 노드에 저장한다.

4) DecodeAndExecuteStage();

함수 DecodeAndExecuteStage는 CPU의 Decode와 Execute 단계를 실행한다. strcmp 함수를 사용해 명령 레지스터 노드에 저장된 Opcode가 어떤 명령어와 일치하는지 확인한다. 명령어가 표 1.1의 산술 명령어와 비교 명령어일 경우, DecodeOperand 함수에서 Operand1과 Operand2를 해석하고 그 외 명령어일 경우, 표 3.1과 같이 해석한다. 해석된 Operand1과 Operand2로 명령어에 맞는 계산을 실행하고, 다음 노드로 명령 포인터를 옮긴다.

점프 명령어(J)	반복문 for를 사용해 명령 포인터가 [Operand1] 번째 노드를 가리킨다.
분기 명령어(B)	R0가 1이면 반복문 for를 사용해 명령 포인터가 [Operand1] 번째 노드를 가리키고, R0가 0이면 다음 노드를 가리킨다.

<표 3.1> 점프, 분기 명령어의 해석

5) DecodeOperand();

함수 DecodeOperand는 Opcode에 저장된 명령어가 +, -, *, /, M, C일 때 Operand를 해석한다. Operand는 레지스터 또는 16진수 숫자이므로 해당 Operand 첫 글자는 R 또는 0임을 알고 있다. 첫 글자가 R일 경우, 레지스터이므로 atoi 함수를 사용해 문자열로 표현된 레지스터 번호를 정수로 바꾼다. 첫 글자가 R이 아닐 경우, 16진수로 숫자이므로 strtol 함수를 사용해 16진수로 표현된 숫자를 10진수로 바꾼다.

나. 예외 처리

1) input.txt가 존재하지 않을 경우

input.txt가 존재하지 않을 경우, input.txt가 정상적으로 열리지 않았음을 화면에 출력하고 ReadText 함수에서 –1을 반환해 프로그램을 종료한다.

2) 0으로 나누는 경우

Opcode가 /이고, Operand2가 0인 경우, 텍스트 파일 n번째 줄의 Operand가 0임을 화면에 출력하고 DecodeAndExecuteStage 함수에서 –1을 반환해 프로그램을 종료한다.

3) 명령어가 잘못 입력됐을 경우

사용자가 입력한 명령이 기본 명령어와 추가 명령어 그 어떤 것에도 해당하지 않을 경우, 텍스트 파일 n번째 줄의 Opcode가 잘못 입력됐음을 화면에 출력하고 DecodeAndExecuteStage 함수에서 –1을 반환해 프로그램을 종료한다.

다. 문제와 해결

처음 간단한 계산기를 설계했을 때, 폰 노이만 구조를 전혀 고려하지 않고 바로 코드를 작성했다. CPU와 메모리의 역할을 분리하지 않고 오직 메인 함수에서 명령을 한꺼번에 처리하도록 하는 동안 크게 두 가지 문제가 발생했다.

첫 번째 문제는 점프와 분기 명령을 구현하기가 어려웠다. 점프나 분기 명령어를 만나면 Operand1이 가리키는 곳부터 명령을 다시 실행해야 했는데 이미 한번 계산을 끝마친 명령으로 다시 돌아갈 방법이 쉽게 떠오르지 않았다. 물론 점프와 분기 명령어를 만날 때마다 텍스트 파일을 한 번 더 열어서 Operand1 이전 명령은 모두 무시하고 Operand1 이후 명령부터 실행하는 코드를 작성할 수 있었지만, 그러면 해당 명령어가 많아질수록 그에 비례해서 텍스트 파일을 여닫아야 하고, 그만큼 계산 시간이 더 길어지는 상황이 발생했다.

두 번째 문제는 메인 함수에만 모든 코드를 작성해 내가 작성한 코드를 읽기가 힘들었다. 텍스트 파일의 명령을 읽고, 명령어가 무엇인지 확인하고, 명령어에 따라 Operand를 해석하고 계산하는 과정이 체계적으로 구분되지 않아서 컴파일 오류가 발생했을 때 몇 번째 명령에서 무슨 오류가 발생했는지 찾아내는 데 오랜 시간이 걸렸다. 또한, 명령을 해석하는 과정에서 똑같은 코드를 여러 번 재사용하게 되어 변수의 이름이 혼재되었고, 코드의 양이 너무 많아져 한눈에 코드를 파악하기가 어려웠다.

첫 번째 문제를 해결하기 위해 폰 노이만 구조의 메모리의 역할을 할 무언가가 있어야 한다고 생각했고, 그 결과 선형 연결 리스트로 연결된 구조체를 사용하게 되었다. 텍스트 파일의 명령어와 데이터를 로드한 구조체가 있으면 구조체 포인터를 사용해 명령에 접근할 수 있고, 이 포인터를 사용해 점프와 분기 명령이 가리키는 또 다른 명령을 쉽게 실행할 수 있었다.

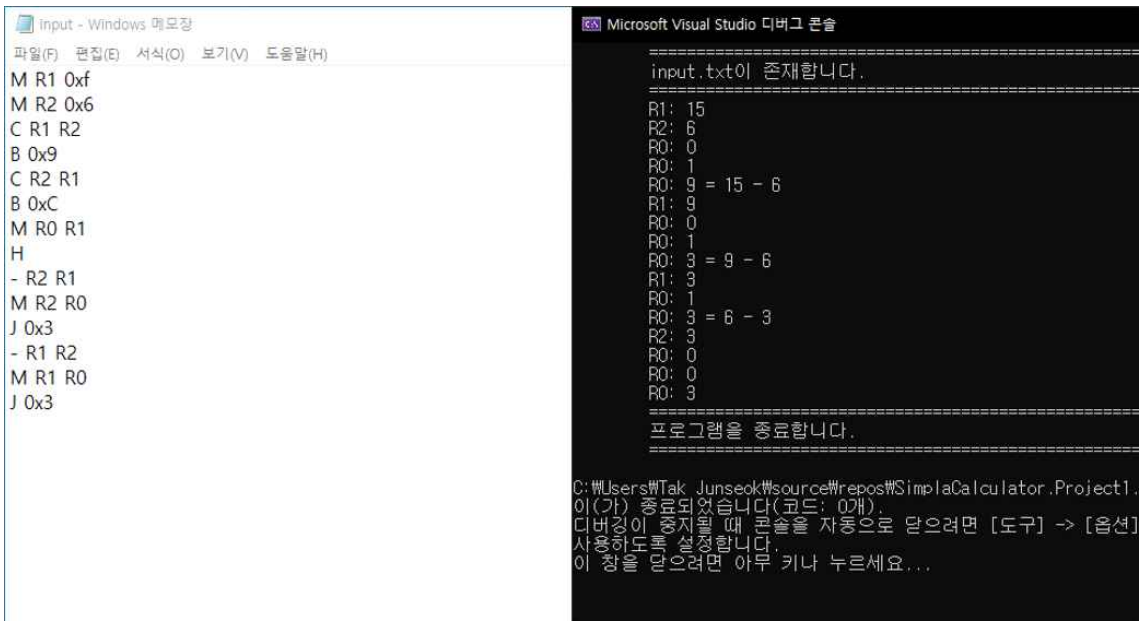
두 번째 문제를 해결하기 위해 프로그램이 메모리에 로드되는 단계와 CPU가 명령을 처리하는 Fetch, Decode, Execute 단계로 코드를 나누었고, 반복되는 코드를 하나로 묶어 각 단계를 처리하는 함수를 설계했다. 텍스트 파일의 명령을 읽는 ReadText 함수, 읽은 명령을 구조체에 담는 LoadMemory 함수, 구조체에 담긴 명령을 하나 불러오는 FetchStage 함수, 명령을 해석하고 실행시켜주는 DecodeandExecute 함수와 DecodeOperand 함수를 설계함으로써 코드가 훨씬 직관적이고 구조적으로 눈에 들어왔고, 지저분했던 메인 함수도 반복문 단 하나로 매우 간결해졌다. 또한, 프로시저 단위로 코드를 실행하면서 오류가 발생하는 부분을 더 빠르게 찾아 수정할 수 있었다.

라. 개발 환경

간단한 계산기 프로그램은 Visual Studio 2019에서 C언어로 작성되었다. input.txt는 SimpleCalculator.c와 같은 폴더에 있어야 한다.

마. 결과

15와 6의 최대 공약수를 구하는 프로그램을 실행해 그림 3.1과 같이 최대 공약수가 3이라는 결과를 얻어냈다.



<그림 3.1> 프로그램 실행 결과

4. 고찰 및 느낀 점

프로젝트1을 마무리하며 이론으로 배운 폰 노이만 구조의 '프로그램 저장'과 '순차적 실행'을 적용해 간단한 계산기 코드를 작성했고, 프로그램을 실행했을 때 프로세서가 명령을 어떻게 처리하는지 알게 되었다.

프로젝트1의 목적을 제대로 이해하지 못했을 때, 그저 안일하게 파일로 입력을 받아 계산하는 코드를 작성하는 것에 몰두해 많은 시간을 투자했다. 하지만 아무런 계획 없이 작성한 코드는 그만큼 많은 문제가 발생했고, 문제를 근본적으로 해결하기 위해서 프로젝트1의 목적을 다시 살펴봤다. 폰 노이만 구조와 자료구조와 알고리즘을 다시 한번 복습한 다음, 코드를 재구성했고, 발생하는 문제를 더 체계적으로 해결할 수 있었다. 프로젝트2부터는 그 목적을 정확하게 이해하고 그에 맞는 접근을 하는 것이 바람직해 보인다.

한편, 내가 만든 간단한 계산기가 더 저렴하고, 더 빠르고, 파워를 덜 소모하며 계산할 방법에 대해 고민해 보았다. 간단한 계산기 프로그램은 오직 한 번에 하나의 명령만 처리할 수 있다. 이를 해결하기 위해 하드웨어의 관점에서 동시에 여러 명령을 처리하는 과정과 CPU 다중코어, 클럭의 개념을 도입해 조금 더 개선해보고 싶다. 소프트웨어 관점에서는 더 가볍고 유연한 프로그램을 만들어 메모리에 로드되는 명령의 양을 줄여, 프로그램 최적화를 해보는 것도 좋을 것 같다.

5. 참고문헌

윤성우(2007). 뇌를 자극하는 윈도우즈 시스템 프로그래밍. 한빛미디어
시바타 보요(2017). Do it! 자료구조와 함께 배우는 알고리즘 입문 C 언어편. 이지스퍼블리싱