I have used **one** of my five free days, leaving four remaining.

Single-cycle MIPS Emulator

32214362 모바일시스템공학과 조윤서

Introduction

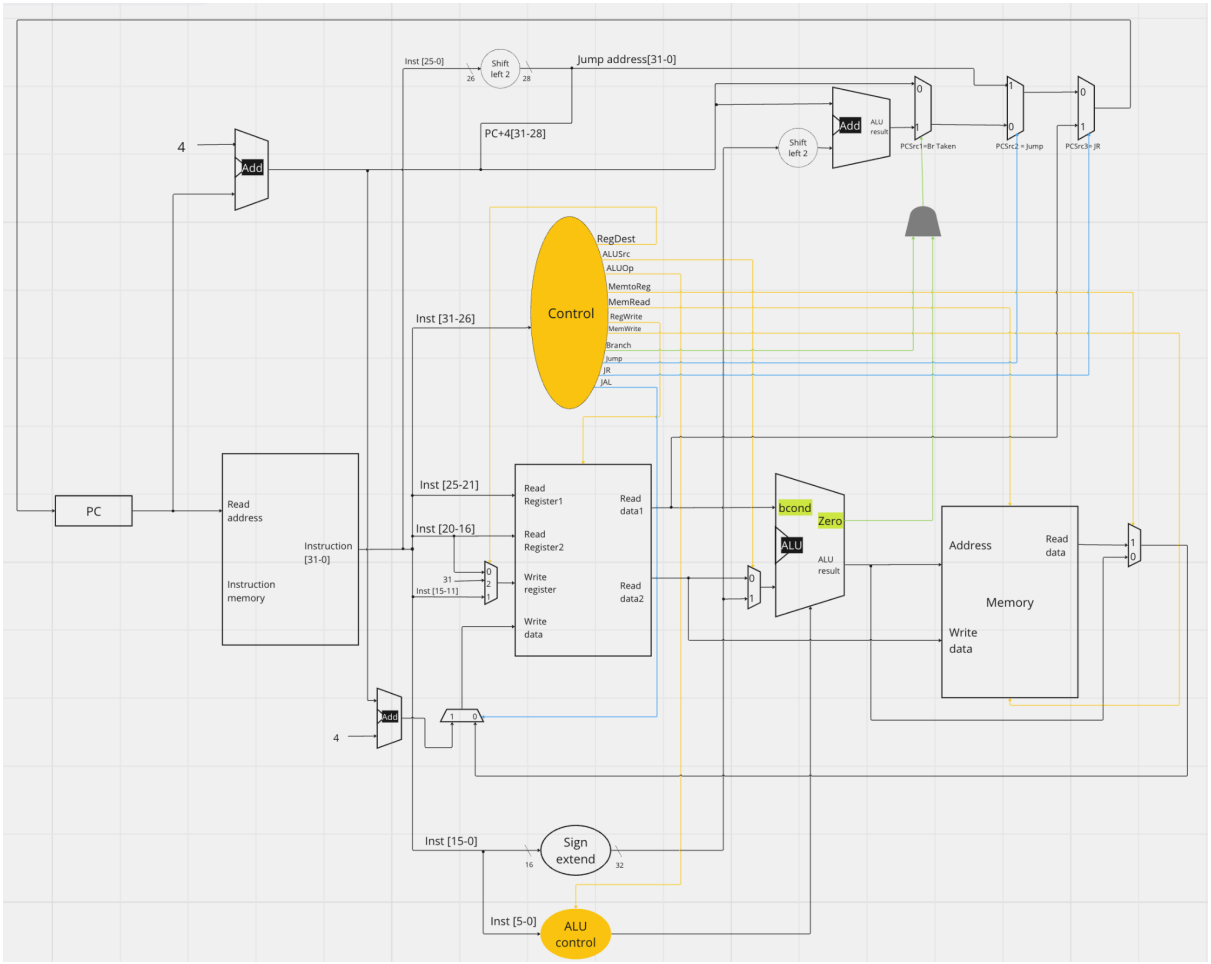
본 프로젝트에서는 MIPS CPU 에뮬레이터를 구현하였다. 이전 과제에서 텍스트 파일을 사용했던 'HW1-simple calculator'와 달리, 이 에뮬레이터는 바이너리 파일을 입력으로 받는다. MIPS 아키텍처를 기반으로 구현된 본 에뮬레이터는, 프로그램의 명령어 실행 과정을 모델링하고 필요한 하드웨어 구성 요소를 소프트웨어적으로 재현한다.

Important Concept

- MIPS 아키텍처
- Single-Cycle
    - Fetch
    - Decode
    - Execute
    - Memory Access
    - Write back

Implementation Process

1. Data Path Design
    a. 본격적인 코드 작업에 앞서 데이터 패스를 먼저 설계하였다. 이는 프로세서의 데이터 처리 과정을 명확히 하기 위한 초기 단계이다.



<Data Path Design>

2. 기능별 함수 구현:
    a. **initiate** 함수
        i. r29(sp)와 r31(ra)를 각각 0x1000000, 0xFFFFFFFF로 초기화하였다.
    b. **load_instructions** 함수
        i. 싱글 사이클을 수행하기 전에, instruction memory에 모든 instructions를 load한다. 이번에는 bin 파일을 입력으로 받기 때문에 바이트 엔디언 변환 작업을 추가하여 파일에서 읽은 데이터의 바이트 순서를 변경하였다.
    c. **Fetch** 함수
        i. Program Counter(PC)가 가리키는 instruction memory의 address에서 instruction을 가져오는 기능을 구현하였다.
    d. **Decode** 함수
        i. instruction을 해석하여 opcode, rs, rt, rd, shamt, funct, imm,

branch address, jump address를 파악하여 instruction의 타입(R, I, J)을 구별하는 기능을 구현하였다.

e. **Execute** 함수
   i. 각 타입별 instruction을 실행한다. ALU(Arithmetic Logic Unit) 함수를 별도로 구현하여 연산을 수행하는 기능을 분리하였다. LW나 SW와 같이 메모리에 접근해야하는 instruction은 ALU 연산이 아닌, 메모리 주소를 계산하였다. execute함수는 처음에는 각 명령어 타입에 따라 별도로 처리하는 방식으로 구현되었으나, 코드의 반복과 복잡성을 줄이기 위해 나중에는 두 개의 입력 변수, input1과 input2를 사용하는 방식으로 개선하였다. 이때, 첫 번째 입력 input1은 레지스터 값R[rs]이다. 두 번째 입력 input2는 명령어가 I타입인지 아닌지에 따라 결정된다. I타입이면 input2는 imm이고, 그렇지 않으면 레지스터 값 R[rt]이다.

f. **Access Memory** 함수
   i. 메모리 접근 단계를 구현하였다. LW의 경우, 메모리로부터 값을 읽어오고, SW의 경우 메모리에 값을 저장한다.

g. **Write Back** 함수
   i. 실행 결과를 레지스터에 저장하는 기능을 구현하였다. 또한, 분기(branch)와 점프(jump) 명령어에 대한 프로그램 카운터(PC) 업데이트도 처리하였다. 특히, jal이나 jalr 명령어의 경우, 점프를 수행하기 전에 레지스터에 값을 저장해야 하기 때문에 이러한 처리가 필요하다. 기본적인 명령어에 대한 PC 업데이트는 execute() 함수 내에서 처리된다.

h. 출력 함수
   i. 각 명령어 실행 후 Architectural state를 출력하는 함수
   ii. 전체 결과를 출력하는 함수

3. Control Signal 추가(option1)
   a. 전체 구현을 마친 후, 제어 신호를 설정하는 함수를 만들었다. 또한, 이 함수에서 설정한 control signal을 사용하여 싱글 사이클 처리 과정의 코드를 재구성하였다. 처음에 design한 data path에서 'RegDest' 제어 신호는 세 개의 입력을 받았다. 하지만 더 효율적인 코드 작성을 위해, 두 개의 입력으로 바꾸고 'RegDest_ra'라는 새로운 제어 신호를

추가하였다. 이 신호는 jal과 jalr 명령어 실행 시 31번 레지스터에 값을 저장하는 데 사용된다.

4. jalr 명령어 추가 구현(option2)
    a. 사실 명령어를 추가 구현하는 것보다도 sample program 작업하는 것이 핵심이었다. fib.mips.asm에서 jal을 jalr로 바꾸는 것으로 끝나는게 아니었기 때문이다. jalr은 jal과 달리 pc값이 Rs address로 점프하기 때문에 그 전에 레지스터에 값을 로드하는 과정을 추가해야했다. 다음은 수정한 mip.asm파일을 bin으로 만드는 터미널 명령어이다.
        i. mips-linux-gnu-as -o fib_edited_final.o fib_edited_final.mips.asm
        ii. mips-linux-gnu-objcopy -O binary -j .text fib_edited_final.o fib_edited_final.bin

5. MIPS binary input program 만들기(option3)
    a. 간단한 선형탐색 프로그램을 만들었다. 다음은 작성한 c파일을 bin으로 만들고 어셈블리어를 확인하는 터미널 명령어이다.
        i. mips-linux-gnu-gcc -c linearSearch.c -o linearSearch.o
        ii. mips-linux-gnu-objcopy -O binary -j .text linearSearch.o linearSearch.bin
        iii. mips-linux-gnu-objdump -d linearSearch.o

Build Configuration / Environment

- Development Environment: **Visual Studio Code**
- Programming Language: **C**

- Build Configuration:
    - ‘main.c’ 컴파일: **gcc main.c**
    - 실행 파일과 ‘simple.bin’ 함께 실행: **./a.out ./test_prog/simple.bin > a.log**
    - 실행 파일과 ‘simple2.bin’ 함께 실행: **./a.out ./test_prog/simple2.bin > a.log**
    - 실행 파일과 ‘simple3.bin’ 함께 실행: **./a.out ./test_prog/simple3.bin > a.log**
    - 실행 파일과 ‘simple4.bin’ 함께 실행: **./a.out ./test_prog/simple4.bin > a.log**
    - 실행 파일과 ‘gcd.bin’ 함께 실행: **./a.out ./test_prog/gcd.bin > a.log**
    - 실행 파일과 ‘fib.bin’ 함께 실행: **./a.out ./test_prog/fib.bin > a.log**

- 실행 파일과 input4.bin’ 함께 실행: **./a.out ./test_prog/input4.bin > a.log**

Screen Capture / Working Proofs

```
============================PROGRAM RESULT=============================
Return value R[2] : 0
Total Cycle : 8
Executed 'R' instruction : 4
Executed 'I' instruction : 4
Executed 'J' instruction : 0
Number of Memory Access Instruction : 2
Number of Branch Taken : 0
=======================================================================
```

<simple.bin의 실행 결과**>**

```
============================PROGRAM RESULT=============================
Return value R[2] : 100
Total Cycle : 10
Executed 'R' instruction : 3
Executed 'I' instruction : 7
Executed 'J' instruction : 0
Number of Memory Access Instruction : 4
Number of Branch Taken : 0
=======================================================================
```

<simple2.bin의 실행결과**>**

```
============================PROGRAM RESULT=============================
Return value R[2] : 5050
Total Cycle : 1330
Executed 'R' instruction : 409
Executed 'I' instruction : 920
Executed 'J' instruction : 1
Number of Memory Access Instruction : 613
Number of Branch Taken : 102
=======================================================================
```

<simple3.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 55
Total Cycle : 243
Executed 'R' instruction : 79
Executed 'I' instruction : 153
Executed 'J' instruction : 11
Number of Memory Access Instruction : 100
Number of Branch Taken : 10
======================================================================
```

<simple4.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 1
Total Cycle : 1061
Executed 'R' instruction : 359
Executed 'I' instruction : 637
Executed 'J' instruction : 65
Number of Memory Access Instruction : 486
Number of Branch Taken : 73
======================================================================
```

<gcd.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 55
Total Cycle : 2679
Executed 'R' instruction : 818
Executed 'I' instruction : 1697
Executed 'J' instruction : 164
Number of Memory Access Instruction : 1095
Number of Branch Taken : 109
======================================================================
```

<fib.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 85
Total Cycle : 23372706
Executed 'R' instruction : 10152862
Executed 'I' instruction : 13219741
Executed 'J' instruction : 103
Number of Memory Access Instruction : 7116606
Number of Branch Taken : 2029699
======================================================================
```

<input4.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 55
Total Cycle : 2896
Executed 'R' instruction : 1091
Executed 'I' instruction : 1750
Executed 'J' instruction : 55
Number of Memory Access Instruction : 1095
Number of Branch Taken : 109
======================================================================
```

<fib_edited_final.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 1
Total Cycle : 65
Executed 'R' instruction : 11
Executed 'I' instruction : 54
Executed 'J' instruction : 0
Number of Memory Access Instruction : 33
Number of Branch Taken : 7
======================================================================
```

<linearSearch.bin의 실행결과**>**

Trial & Errors

Error 1: segmentation fault

- 프로그램 실행 중 메모리 인덱스가 지정된 메모리 크기를 초과하여 발생하였다.

Trial 1: 메모리 사이즈 재설정

- 처음에 메모리 사이즈를 0x400000 (4MB)로 설정한 것을 발견하였다. 메모리 사이즈를 0x4000000 (64MB)로 수정하여 메모리 용량 부족 문제를 해결하였다.

Error 2: Memory access error - Invalid memory index

- 메모리 사이즈를 조정한 이후에도 비슷한 오류가 계속 발생하였다.

Trial 2: 변수의 정확한 사용

- 변수 사용에 실수가 있음을 발견하였다. 구체적으로는, R[rs] 값을 사용해야 할 상황에서 rs값을 사용하여 프로그램이 제대로 실행되지 않았다. 이후에도 비슷한 오류가 뜨면 위에서 선언한대로 내가 제대로 변수를 사용했는지 확인하였다.

Error 3: input4.bin 실행 후 a.log 파일 안열림

Trial 3: 불필요한 printf문을 주석처리함.

- printf문이 많을수록 log파일의 용량이 커짐을 발견하였다.

Error 4: mips.asm 파일 수정 후 bin파일로 컴파일 안됨

Trial 4: 어셈블리어 작성하는 법 익힘

- 디어셈블리 파일에서 그대로 수정해서 컴파일하려고 하였기에 안됨을 발견하였다.

Error 5: mips 컴파일러 존재하지 않음

Trial 5: 아쌈서버 사용

- 내 맥북에는 MIPS 컴파일러가 설치되어 있지 않아, 도커와 homebrew를 이용해 설치하려 했으나 성공하지 못했다. 결국, 아쌈 서버를 사용해보니 훨씬 편리하게 MIPS 컴파일이 가능함을 알게 되었다. 따라서아쌈 서버에서 컴파일하여 bin파일을 만든 후 이를 다운로드 받아와 내 vscode에서 사용하였다.

Lesson

이번 프로젝트를 통해, 하드웨어와 근접한 코딩을 하면서 코딩의 복잡성을 직접 경험할 수 있었다. 특히, 'input4'와 같은 큰 데이터를 처리할 때, 자주 발생한 Visual Studio Code의 강제 종료는 프로그램의 처리 능력과 안정성의 한계를 느끼게 해주었다. 또한, printf 사용이 많을 경우 프로그램의 속도 저하를 경험함으로써, 출력 함수의 사용이 성능에 미치는 영향을 직접 확인하였다. 디버깅 과정이 예상보다 오래 걸리며, 결과물 파일을 확인하는 데에도 상당한 시간이 소요되었다. 특히 어셈블리와 컴파일러에 대한 깊은 이해가 요구되는 옵션 2와 옵션 3의 구현은 매우 어려웠다. 이것이 하루의 freeday를 쓴 이유이기도 하다. 이번 프로젝트에서는 전체적인 구조를 먼저 개략적으로 구상한 후 각 기능을 구현하였지만, 모든 코드를 main.c에 작성한 것은 코드의 가독성을 저하시켰다. 다음 프로젝트에서는 헤더 파일을 적극 활용하여 코드를 더욱 깔끔하고 관리하기 쉬운 형태로 구성할 것이다.