DKU 단국대학교 DANKOOK UNIVERSITY

REPORT

HW2

과목명 : 컴퓨터구조론 (2분반)
교 수 : 남재현 교수님
소속학과: 소프트웨어학과
학 번 : 32190192
이 름 : 구영민
마 감 : 2024.05.28 (화요일까지)

단국대학교 DANKOOK UNIVERSITY

목 차

표 목차

사진 목차

1. Instruction

본 과제의 목표는 단일 사이클 MIPS 아키텍처 기반의 CPU 에뮬레이터를 개발하는 것입니다. 이 에뮬레이터는 부동 소수점 연산을 제외한 MIPS 명령어 세트를 지원하며, MIPS 이진 프로그램을 메모리에 로드하여 실행할 수 있습니다.

에뮬레이터의 기능

- MIPS 이진 프로그램 로드: 실행 전에 메모리에 이진 프로그램을 로드합니다. 이는 프로그램의 실행을 위해 필수적인 단계로, 프로그램 코드가 메모리에 올바르게 배치되어야 합니다.
- 명령어 실행: 로드된 프로그램의 명령어를 실행합니다. 이 과정에서 MIPS 아키텍처의 주요 명령어들을 처리할 수 있어야 합니다.
- 예외 처리: 실행 도중 발생할 수 있는 다양한 예외 상황을 적절히 처리합니다. 이는 에뮬레이터의 안정성과 신뢰성을 보장하는 데 중요합니다.

실행 환경

- 메모리 구성: 에뮬레이터는 충분한 크기의 가상 메모리를 구현해야 하며, 이 메모리는 MIPS 명령어를 저장하고 처리하는 데 사용됩니다.
- 레지스터 초기화: 모든 레지스터는 0으로 초기화되며, $ra (리턴 어드레스 레지스터)와 $sp (스택 포인터)는 각각 0xFFFFFFFF와 0x1000000으로 설정됩니다.
- 프로그램 카운터 (PC): 프로그램의 실행이 종료될 때 PC 값이 0xFFFFFFFF로 설정되면, 에뮬레이터는 프로그램의 실행을 종료합니다.

개발 요구사항

- 단일 사이클 실행: 에뮬레이터는 모든 명령어를 단일 사이클 내에서 처리해야 합니다. 이는 모든 명령어가 하나의 클록 사이클 동안에 시작되고 완료됨을 의미합니다.

- 명령어 세트: 부동 소수점 연산을 제외한 주요 MIPS 명령어들을 지원해야 합니다. 이를 통해 다양한 프로그램을 효과적으로 실행할 수 있습니다.

2. Background

이 프로젝트를 효과적으로 수행하기 위해서는 MIPS 아키텍처의 몇 가지 기본 개념과 제어 신호들을 이해하고 구현할 필요가 있습니다:

1. 제어 신호의 기본 개념

제어 신호의 역할

- 제어 신호는 프로세서가 다음과 같은 작업을 수행하는 데 필요한 지침을 제공합니다
- 어떤 연산을 수행할 것인지 결정
- 데이터 경로의 구성 요소들을 어떻게 활성화하거나 연결할 것인지 조정
- 메모리 접근, ALU 연산, 레지스터 간의 데이터 이동 등을 제어

2. 주요 제어 신호

MIPS 프로세서에서 사용되는 몇 가지 중요한 제어 신호는 다음과 같습니다

RegDst

- 결과값을 저장할 레지스터를 결정합니다.
- 명령어 유형에 따라 신호가 달라집니다: R-type 명령어의 경우 1, I-type 명령어의 경우 0.

ALUSrc

- ALU의 두 번째 입력이 레지스터(rt)에서 올지 메모리나 상수 값(즉, 명령어에서 직접 주어진 값)에서 올지를 결정합니다.
- I-type 명령어에서는 1, R-type 명령어에서는 0으로 설정됩니다.

MemtoReg(메모리-레지스터 결정 신호)

- ALU의 결과나 메모리로부터 읽은 데이터 중 어떤 것을 레지스터에 쓸 것인지 결정합니다.
- load 명령어에서만 1로 설정됩니다.

RegWrite(레지스터 쓰기 활성화 신호)

- 레지스터 파일에 데이터를 쓸지 여부를 결정합니다. 이 신호가 활성화되면 실행 결과가 레지스터에 저장됩니다.
- addu, mult, mflo, lw, addiu, slti와 같이 레지스터에 결과를 다시 쓸 필요가 있는 명령어에 대해 활성화(1)됩니다.

MemRead(메모리 읽기 활성화 신호)

- 메모리 읽기 작업을 활성화합니다. 이 신호가 설정되면, 데이터 메모리에서 데이터가 읽혀지고, CPU 내부로 전송됩니다.
- load 명령어에서만 1로 설정됩니다.

MemWrite (메모리 쓰기 활성화 신호)

- 메모리 쓰기 작업을 활성화합니다. 이 신호가 설정되면, 레지스터에서 데이터가 메모리로 전송되어 저장됩니다.
- store 명령어에서만 1로 설정됩니다.

PCSrc

- 분기(조건부 점프) 명령어의 실행을 제어합니다. 분기 조건이 충족될 경우, 프로그램 카운터(PC)가 업데이트되어 새로운 주소로 점프합니다.

ALUOp (ALU 연산 결정 신호)

- ALU에서 수행할 연산 유형을 결정합니다.

0: AND 연산

2: ADD 연산

3: Multiply 연산

4: Move from LO 연산

6: Subtract 연산

7: Set on less than immediate 연산 (예: slti)

3. testcase의 모든 instruction의 opcode 조사 내용

type	instruction	opcode
R-type	addu	33
	mult	24
	mflo	18
	jr	8
J-type	jal	3
	addiu	9

I-type	sw	43
	lw	35
	slti	10
	bne	5
	bnez	5
	b	4
	beqz	4
	NOP	0

[표 1] opcode 표

4. testcase의 instruction 별 Control 신호 조사 내용

	RegDst	**Regwrite**	**ALUSrc**	**PCSrc**	**MemRead**	**Memwrite**	**MemtoReg**	**ALUOp**
addu	1	1	0	0	0	0	0	2
mult	0	1	0	0	0	0	0	3
mflo	1	1	0	0	0	0	0	4
jr	0	0	0	1	0	0	0	0
jal	0	1	0	1	0	0	0	0
addiu	0	1	1	0	0	0	0	2
sw	0	0	1	0	0	1	0	2
lw	0	1	1	0	1	0	1	2
slti	0	1	1	0	0	0	0	7

bne	0	0	0	1	0	0	0	6
bnez	0	0	0	1	0	0	0	6
b	0	0	0	1	0	0	0	6
beqz	0	0	0	1	0	0	0	6
NOP	0	0	0	0	0	0	0	0

[표 2] Control 표

5. testcase의 instruction 별 ALU 값 조사 내용

ALUop

MIPS 프로세서에서 ALUOp은 ALU에서 수행할 연산의 유형을 결정하는 제어 신호입니다.

이 신호는 명령어의 유형과 필요한 연산에 따라 조정되어, ALU가 적절한 연산을 수행하도록 합니다.

ALUOp의 역할

ALUOp 신호는 ALU에게 수행할 작업을 알려주며, 이는 명령어의 종류에 따라 달라집니다.

예를 들어, 논리 연산, 산술 연산, 비교 연산 등 다양한 종류의 작업이 있을 수 있습니다.

ALUOp 신호는 특정한 몇 비트 값을 사용하여 이러한 연산들을 지정합니다.

ALU	Instruction
0	jr, jal
2	addu, addiu, sw, lw
3	mult

4	mflo
6	bne, bnez, beqz, b
7	Slti

[표 3] ALUop 표

3. Design

이번 과제에서 설계한 MIPS 시뮬레이터는 크게 메모리 관리 부분과 명령어 처리 부분으로 나눌 수 있습니다. 이 두 부분은 MIPS 명령어를 인출, 해독, 실행, 메모리 접근, 결과 쓰기 단계를 거쳐 처리합니다. 각 단계는 별도의 함수로 구성되어 있으며, 함수가 수행하는 역할을 중심으로 코드의 구조와 흐름을 설명하겠습니다.

1. 메모리 관리 부분

메모리 관리 부분은 프로그램 실행을 위해 필요한 명령어와 데이터를 메모리에 적재하고 관리하는 역할을 합니다.

Data 배열 : instruction fetch를 하기 전에 바이너리 파일을 파싱한 32bit instruction값을 순차로 저장합니다.

Memory 배열 : 실제 가상메모리를 하는 역할로 데이터를 메모리에 적재하고 관리하는 역할을 합니다.

1. 메모리 초기화 (init 함수)

init 함수는 시뮬레이터가 실행되기 전에 레지스터와 메모리를 초기화합니다.

초기화 과정에서 스택 포인터(sp)와 반환 주소 레지스터(ra)를 설정합니다.

메모리 배열을 초기화하여 명령어와 데이터를 저장할 준비를 합니다.

2. 데이터 파싱 (parseData 함수)

parseData 함수는 입력된 바이너리 파일에서 명령어 데이터를 읽어와 메모리에 저장합니다. 파일에서 4바이트씩 읽어와 각 명령어를 32비트 이진 문자열로 변환하여 data 배열에 저장합니다.

2. 명령어 처리 부분

명령어 처리 부분은 인출, 해독, 실행, 메모리 접근, 결과 쓰기의 5단계로 구성됩니다. 각 단계는 특정 함수를 통해 수행됩니다.

3. 전체 흐름 (run 함수)

run 함수는 시뮬레이터의 메인 루프를 담당합니다. 각 사이클마다 다음 단계를 수행합니다:

a. 명령어 인출

b. 명령어 해독

c. 명령어 실행

d. 메모리 접근

e. 결과 쓰기

f. 점프 처리 (PC 업데이트)

a. 명령어 인출 (instructionFetch 함수)

instructionFetch 함수는 현재 프로그램 카운터(PC)가 가리키는 주소에서 명령어를

인출합니다.

인출한 명령어를 cur_instruction에 저장하고, PC를 다음 명령어로 이동시킵니다.

사이클 수를 증가시키고, 현재 인출된 명령어와 PC 값을 출력합니다.

b. 명령어 해독 (instructionDecode 함수)

instructionDecode 함수는 인출된 명령어를 해독하여 명령어 타입을 결정합니다.

명령어 타입에 따라 R-type, J-type, I-type 명령어를 해독하는 함수(parseRType, parseJType, parseIType)를 호출합니다.

해독 과정에서 명령어의 각 필드(연산 코드, 레지스터, 즉시 값 등)를 추출하고, 제어 신호를 설정합니다.

c. 실행 (execute 함수)

execute 함수는 명령어의 실행 단계를 담당하며, ALU 연산을 수행합니다. 이 단계에서는 ALUSrc와 ALUOp 제어 신호가 중요한 역할을 합니다.

ALUSrc 제어 신호

- ALUSrc는 ALU의 두 번째 입력이 레지스터 값(readData2)인지 즉시 값(immediate)인지 결정합니다. I-type 명령어의 경우 ALUSrc는 1로 설정되어 즉시 값이 사용됩니다. R-type 명령어의 경우 ALUSrc는 0으로 설정되어 레지스터 값이 사용됩니다.

ALUOp 제어 신호

- ALUOp는 ALU가 수행할 연산을 결정합니다. 각 명령어에 대해 다음과 같은 연산을 수행합니다:

0: jr, jal 명령어에 대해 AND 연산

2: addu, addiu, sw, lw 명령어에 대해 ADD 연산

3: mult 명령어에 대해 곱셈 연산

4: mflo 명령어에 대해 LO 레지스터 값 읽기

6: bnez, bne, beqz, b 명령어에 대해 SUB 연산 (분기 결정)

7: slti 명령어에 대해 SLT 연산 (set less than)

d. 메모리 접근 (memoryAccess 함수)

memoryAccess 함수는 명령어가 메모리 접근을 필요로 하는 경우 이를 처리합니다. 이 단계에서는 MemRead와 MemWrite 제어 신호가 중요한 역할을 합니다.

MemRead 제어 신호

- MemRead는 메모리 읽기 작업이 필요한지를 결정합니다. lw 명령어의 경우 MemRead는 1로 설정되어 메모리에서 데이터를 읽습니다.

MemWrite 제어 신호

- MemWrite는 메모리 쓰기 작업이 필요한지를 결정합니다. sw 명령어의 경우 MemWrite는 1로 설정되어 데이터를 메모리에 씁니다.결과 쓰기 (writeBack)

e. 목적지 레지스터 write (writeBack함수)

writeBack 함수는 연산 결과를 레지스터에 쓰는 단계를 담당합니다. 이 단계에서는 RegWrite와 MemtoReg 제어 신호가 중요한 역할을 합니다.

RegWrite 제어 신호

- RegWrite는 레지스터 쓰기 작업이 필요한지를 결정합니다. addu, mult, mflo, lw, addiu, slti 명령어의 경우 RegWrite는 1로 설정되어 레지스터에 데이터를 씁니다.

MemtoReg 제어 신호

- MemtoReg는 메모리에서 읽은 데이터가 레지스터에 쓰이는지를 결정합니다. lw 명령어의 경우 MemtoReg는 1로 설정되어 메모리에서 읽은 데이터가 레지스터에 저장됩니다.

RegDst 제어 신호

- RegDst는 데이터를 저장할 레지스터가 rt인지 rd인지를 결정합니다. R-type 명령어의 경우 RegDst는 1로 설정되어 rd에 데이터를 저장하고, I-type 명령어의 경우 RegDst는 0으로 설정되어 rt에 데이터를 저장합니다.

f. 점프 처리(possibleJump 함수)

possibleJump 함수는 프로그램 카운터(PC)를 업데이트하여 점프 또는 분기 명령어가 실행될 경우 새로운 PC 값을 설정하는 역할을 합니다.

PCsrc 신호

- PCsrc 신호가 활성화된 경우, 각각의 맞게 pc를 새로 업데이트 해줍니다.

각 단계가 완료된 후, 프로그램 카운터가 특정 값(종료 조건)에 도달하면 시뮬레이션을 종료합니다. 또한, 무한 루프를 감지하기 위해 카운터를 사용하여 일정 사이클 이상 반복될 경우 시뮬레이션을 강제로 종료합니다.

4. Implementation

Testcase의 모든 instruction 구현 완료 하였습니다.

type	instruction	구현 여부
R-type	addu	O
	mult	O
	mflo	O
	jr	O
J-type	jal	O
I-type	addiu	O
	sw	O
	lw	O
	slti	O
	bne	O
	bnez	O
	b	O
	beqz	O
	NOP	O

[표 4] instruction 구현 여부

1. R-type Instructions

addu (Add Unsigned)

```
32190192> Cycle: 143
    [Instruction Fetch] 0x03c0e825  (PC=0x00000058)
    [Instruction Decode] Type: R, Inst: addu sp s8 zero
        opcode: 0, rd: 29 (16777200), rs: 30 (16777200), rt: 0 (0)
        RegDst: 1, RegWrite: 1, ALUSrc: 0, PCSrc: 0, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 2
    [Execute] ALU = 16777200
    [Memory Access] Pass
    [Write Back] Target: sp, Value: 16777200 / newPC: 0x0000005c
```

[사진 1] addu 연산 과정

1. Instruction Fetch: addu 명령어가 PC가 가리키는 위치에서 가져옵니다.
2. Instruction Decode: opcode = 000000. rs, rt, rd를 파싱하여 연산 대상 레지스터를 식별합니다.
3. Execute: 레지스터 rs와 rt의 값을 더합니다.
4. Memory Access: 메모리 접근이 필요 없습니다.
5. Write Back: 결과를 rd에 저장합니다.

mult (Multiply)

```
32190192> Cycle: 74
    [Instruction Fetch] 0x00620018  (PC=0x00000094)
    [Instruction Decode] Type: R, Inst: mult v1 v0
        opcode: 0, rs: 3 (1), rt: 2 (10)
        RegDst: 0, RegWrite: 1, ALUSrc: 0, PCSrc: 0, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 3
    [Execute] mult result: LO=10, HI=0
    [Memory Access] Pass
    [Write Back] newPC: 0x00000098
```

[사진 2] mult 연산 과정

1. Instruction Fetch: mult 명령어를 읽습니다.
2. Instruction Decode: 연산 대상 레지스터 rs, rt를 식별합니다.
3. Execute: rs와 rt를 곱한 결과를 HI, LO 레지스터에 저장합니다.
4. Memory Access: 이 단계는 생략됩니다.

5. Write Back: 결과가 HI, LO에 저장되므로 별도의 레지스터 업데이트가 없습니다.

mflo (Move from LO)

```
32190192> Cycle: 75
    [Instruction Fetch] 0x00001012  (PC=0x00000098)
    [Instruction Decode] Type: R, Inst: mflo v0
        opcode: 0, rd: 2 (16777120)
        RegDst: 1, RegWrite: 1, ALUSrc: 0, PCSrc: 0, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 4
    [Execute] mflo result: 10
    [Memory Access] Pass
    [Write Back] Target: v0, Value: 10 / newPC: 0x0000009c
```

[사진 3] mflo 연산 과정

1. Instruction Fetch: 명령어를 가져옵니다.
2. Instruction Decode: 대상 레지스터 rd를 식별합니다.
3. Execute: LO 레지스터의 내용을 rd로 이동합니다.
4. Memory Access: 이 단계는 생략됩니다.
5. Write Back: LO의 값이 rd에 저장됩니다.

jr (Jump Register)

```
32190192> Cycle: 80
    [Instruction Fetch] 0x03e00008  (PC=0x000000ac)
    [Instruction Decode] Type: R, Inst: jr ra
        opcode: 0, rs: 31 (132)
        RegDst: 0, RegWrite: 0, ALUSrc: 0, PCSrc: 1, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 0
    [Execute] Pass
    [Memory Access] Pass
    [Write Back] newPC: 0x00000084
```

[사진 4] jr 연산 과정

1. Instruction Fetch: jr 명령어를 읽습니다.
2. Instruction Decode: 점프할 주소가 저장된 rs 레지스터를 식별합니다.
3. Execute: rs의 값을 읽어 PC를 업데이트합니다.
4. Memory Access: 이 단계는 생략됩니다.
5. Write Back: PC 업데이트 외에 별도의 작업은 없습니다.

2. J-type Instructions

jal (Jump and Link)

```
32190192> Cycle: 23
    [Instruction Fetch] 0x0c00000d  (PC=0x00000084)
    [Instruction Decode] Type: J, Inst: jal: 13
        opcode: 3, address: 13
        RegDst: 0, RegWrite: 1, ALUSrc: 0, PCSrc: 1, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 0
    [Execute] Pass
    [Memory Access] Pass
    [Write Back]pc를 ra로 update /  newPC: 0x00000034
```

[사진 5] jal 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: 점프 주소를 파싱합니다.
3. Execute: 현재 PC + 4를 ra에 저장하고, 점프 주소로 PC를 업데이트합니다.
4. Memory Access: 이 단계는 생략됩니다.
5. Write Back: ra 레지스터에 반환 주소가 저장됩니다.

3. I-type Instructions

addiu (Add Immediate Unsigned)

```
32190192> Cycle: 145
    [Instruction Fetch] 0x27bd0010  (PC=0x00000060)
    [Instruction Decode] Type: I, Inst: addiu sp sp 16
        opcode: 9, rt: 29 (16777200), rs: 29 (16777200), imm: 16
        RegDst: 0, RegWrite: 1, ALUSrc: 1, PCSrc: 0, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 2
    [Execute] ALU = 16777216
    [Memory Access] Pass
    [Write Back] Target: sp, Value: 16777216 / newPC: 0x00000064
```

[사진 6] addiu 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: rs, rt, 즉시 값 immediate를 파싱합니다.
3. Execute: rs의 값과 immediate를 더합니다.

4. Memory Access: 이 단계는 생략됩니다.

5. Write Back: 계산 결과를 rt에 저장합니다.

sw (Store Word)

```
32190192> Cycle: 2
    [Instruction Fetch] 0xafbf001c  (PC=0x00000004)
    [Instruction Decode] Type: I, Inst: sw ra 28(sp)
        opcode: 43, rt: 31 (-1), rs: 29 (16777184), imm: 28
        RegDst: 0, RegWrite: 0, ALUSrc: 1, PCSrc: 0, MemRead: 0, MemWrite: 1, MemtoReg: 0, ALUOp: 2
    [Execute] ALU = 16777212
    [Memory Access] Store, Address: 0x00fffffc, Value: 4294967295
    [Write Back] newPC: 0x00000008
```

[사진 7] sw 연산 과정

1. Instruction Fetch: 명령어를 가져옵니다.

2. Instruction Decode: rs, rt, immediate를 파싱하여 주소 계산에 사용합니다.

3. Execute: rs의 값에 immediate를 더하여 주소를 계산합니다.

4. Memory Access: 계산된 주소에 rt의 값을 저장합니다.

5. Write Back: 메모리에 데이터를 저장한 후 별도의 레지스터 업데이트는 없습니다.

lw (Load Word)

```
32190192> Cycle: 144
    [Instruction Fetch] 0x8fbe000c  (PC=0x0000005c)
    [Instruction Decode] Type: I, Inst: lw s8 12(sp)
        opcode: 35, rt: 30 (16777200), rs: 29 (16777200), imm: 12
        RegDst: 0, RegWrite: 1, ALUSrc: 1, PCSrc: 0, MemRead: 1, MemWrite: 0, MemtoReg: 1, ALUOp: 2
    [Execute] ALU = 16777212
    [Memory Access] Load, Address: 0x00fffffc, Value: 0
    [Write Back] target: s8, Value: 0 / newPC: 0x00000060
```

[사진 8] lw 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.

2. Instruction Decode: 메모리 주소 계산을 위해 rs, rt, immediate를 파싱합니다.

3. Execute: rs에 immediate를 더하여 메모리 주소를 계산합니다.

4. Memory Access: 계산된 주소에서 데이터를 읽습니다.

5. Write Back: 읽은 데이터를 rt에 저장합니다.

slti (Set Less Than Immediate)

```
32190192> Cycle: 30
    [Instruction Fetch] 0x28420002  (PC=0x00000054)
    [Instruction Decode] Type: I, Inst: slti v0 v0 2
        opcode: 10, rt: 2 (14), rs: 2 (14), imm: 2
        RegDst: 0, RegWrite: 1, ALUSrc: 1, PCSrc: 0, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 7
    [Execute] ALU = 0
    [Memory Access] Pass
    [Write Back] Target: v0, Value: 0 / newPC: 0x00000058
```

[사진 9] slti 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: rs, rt, immediate를 파싱합니다.
3. Execute: rs의 값이 immediate보다 작은지 비교합니다.
4. Memory Access: 이 단계는 생략됩니다.
5. Write Back: 비교 결과를 rt에 저장합니다 (0 또는 1).

bne (Branch on Not Equal)

```
32190192> Cycle: 18
    [Instruction Fetch] 0x14620004  (PC=0x00000060)
    [Instruction Decode] Type: I, Inst: bne v1 v0 4
        opcode: 5, rt: 2 (1), rs: 3 (5), imm: 4
        RegDst: 0, RegWrite: 0, ALUSrc: 0, PCSrc: 1, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 6
    [Execute] ALU = 4
    [Memory Access] Pass
    [Write Back] newPC: 0x00000074
```

[사진 10] bne 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: rs, rt, immediate를 파싱하여 분기 조건을 설정합니다.
3. Execute: rs와 rt가 같지 않다면, PC를 immediate * 4만큼 오프셋합니다.
4. Memory Access: 이 단계는 생략됩니다.
5. Write Back: PC가 업데이트되며, 레지스터는 변하지 않습니다.

bnez(Branch Not Equal to Zero)

```
32190192> Cycle: 10
    [Instruction Fetch] 0x1440fff3  (PC=0x0000004c)
    [Instruction Decode] Type: I, Inst: bnez v0 -13
        opcode: 5, rs: 2 (1), imm: -13
        RegDst: 0, RegWrite: 0, ALUSrc: 0, PCSrc: 1, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 6
    [Execute] ALU = 1
    [Memory Access] Pass
    [Write Back] newPC: 0x0000001c
```

[사진 11] bnez 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: 명령어에서 rs 레지스터와 분기 오프셋을 추출하고 명령어가 분기 명령임을 식별합니다.
3. Execute: rs 레지스터의 값을 검사하여 0이 아니면, PC를 조정해 새로운 주소로 점프하도록 계산합니다.
4. Memory Access: 메모리 접근은 필요 없으므로 이 단계는 생략됩니다.
5. Write Back: BNEZ는 레지스터 값을 업데이트하지 않으므로 이 단계도 생략됩니다.

b (Branch)

```
32190192> Cycle: 16
    [Instruction Fetch] 0x10400004  (PC=0x00000058)
    [Instruction Decode] Type: I, Inst: b 4
        opcode: 4, rt: 0 (0), rs: 2 (0), imm: 4
        RegDst: 0, RegWrite: 0, ALUSrc: 0, PCSrc: 1, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 6
    [Execute] ALU = 0
    [Memory Access] Pass
    [Write Back] newPC: 0x0000006c
```

[사진 12] b 연산 과정

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: 분기 조건을 설정하기 위해 rs, immediate를 파싱합니다.
3. Execute: rs가 0이면, PC를 immediate * 4만큼 오프셋합니다.
4. Memory Access: 이 단계는 생략됩니다.

5. Write Back: PC만 변경됩니다.

beqz (Branch if Equal to Zero)

testcase에 intsruction이 존재하긴 하지만 쓰이진 않는 것 같습니다!

1. Instruction Fetch: 명령어를 읽습니다.
2. Instruction Decode: rs, immediate를 파싱하여 분기 조건을 설정합니다.
3. Execute: rs가 0이면, PC를 immediate * 4만큼 오프셋합니다.
4. Memory Access: 이 단계는 생략됩니다.
5. Write Back: PC만 변경됩니다.

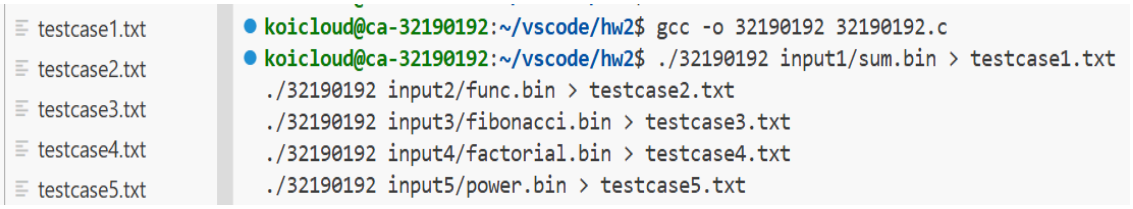
NOP

```
32190192> Cycle: 20
    [Instruction Fetch] 0x00000000  (PC=0x00000078)
    [Instruction Decode] NOP!!!
```

[사진 13] nop 처리 과정

5. Test

1. 실행 결과



[사진 14] 컴파일 과정

txt파일에 따로 저장하여 프로그램 출력 값을 확인했습니다.

[Testcase 1] sum.c

```
32190192> Final Result
    Cycles: 146, R-type instructions: 13, I-type instructions: 101, J-type instructions: 0
    Return value (v0) : 45
```

[사진 15] sum 실행결과

[Testcase 2] func.c

```
32190192> Final Result
    Cycles: 99, R-type instructions: 24, I-type instructions: 60, J-type instructions: 4
    Return value (v0) : 10
```

[사진 16] func 실행결과

[Testcase 3] fibonacci.c

```
32190192> Final Result
    Cycles: 48345, R-type instructions: 9866, I-type instructions: 29601, J-type instructions: 1973
    Return value (v0) : 610
```

[사진 17] Fibonacci 실행결과

[Testcase 4] factorial.c

```
32190192> Final Result
    Cycles: 144, R-type instructions: 34, I-type instructions: 81, J-type instructions: 5
    Return value (v0) : 120
```

[사진 18] factorial 실행결과

[Testcase 5] power.c

```
32190192> Final Result
    Cycles: 108, R-type instructions: 27, I-type instructions: 62, J-type instructions: 4
    Return value (v0) : 1000
```

[사진 19] power 실행결과

2. 예외처리

[exception 1]

```
// 시뮬레이션을 실행하는 주 함수
void run() {
    init(); // 초기화 함수 호출로 레지스터 및 메모리 초기화
    int loopCounter = 0;  // 무한 루프 감지를 위한 카운터

    while (1) { // 무한 루프로 명령어 실행
        loopCounter++;
        if (loopCounter > 1000000) {  // 1000,000 사이클이 넘어가면 루프 감지
            printf("Infinite loop detected, stopping the simulation\n");
            break;
        }
```

[사진 20] 무한 루프 방지 예외처리

[exception 2]

```
int instructionDecode() {

    char type = classifyInstruction(binaryToUint32(cur_instruction));
    if (type == 'N') {
        printf("\t[Instruction Decode] NOP!!!\n");
        return 1;
    }
    else if (type == '?') {  // 잘못된 명령어 타입을 식별
        printf("\t[Instruction Decode] Unknown instruction type\n");
        return 1;
    }
    printf("\t[Instruction Decode] Type: %c, ", type);
```

[사진 21] 잘못된 명령어 예외처리

[exception 3]

```
// 메모리 접근 단계 처리 함수
void memoryAccess() {
    if (ALUResult >= 0 && ALUResult < sizeof(memory) / sizeof(memory[0])) {
        if (MemRead == 1) { // 메모리 읽기 활성화인 경우
            Register[rt] = memory[ALUResult]; // 메모리에서 rt 레지스터로 데이터 로드
            printf("\t[Memory Access] Load, Address: 0x%08x, Value: %d\n", ALUResult, Register[rt]);
        } else if (MemWrite == 1) { // 메모리 쓰기 활성화인 경우
            memory[ALUResult] = Register[rt]; // rt 레지스터에서 메모리로 데이터 저장
            printf("\t[Memory Access] Store, Address: 0x%08x, Value: %u\n", ALUResult, memory[ALUResult]);
        } else {
            printf("\t[Memory Access] Pass\n"); // 메모리 작업 없음
        }
    }
    else{
        printf("\t[Memory Access] Invalid memory access\n");
    }
}
```

[사진 22] 메모리 접근 예외처리

6. Lesson

이번 과제를 통해 얻은 가장 큰 교훈은 컴퓨터의 작동 원리를 단순히 이론적으로 아는 것과 실제로 구현해보는 것 사이에 큰 차이가 있다는 점입니다. 단순한 C 코드도 컴퓨터가 일일이 파싱하여 조각조각 기억해 실행한다는 사실을 직접 경험하면서, 컴퓨터 구조와 동작 원리에 대한 이해가 깊어졌습니다.

1. 명령어 사이클의 중요성

컴퓨터가 명령어를 실행하는 과정에서, 각 명령어가 어떻게 인출되고, 디코딩되며, 실행되고, 메모리 접근을 하고, 결과를 레지스터에 쓰는지를 상세히 이해할 수 있었습니다. 명령어 인출 단계에서는 프로그램 카운터(PC)가 다음 실행할 명령어를 가리키고, 해당 명령어를 메모리에서 읽어옵니다. 그 다음 단계인 명령어 디코드에서는 명령어의 각 필드를 해석하여 필요한 제어 신호를 설정합니다. 이 과정에서 레지스터를 읽고, ALU 연산을 수행하며, 필요에 따라 메모리에서 데이터를 읽거나 쓰는 등의 작업이 수행됩니다. 이 모든 과정이 정확히 이루어져야만 올바른 프로그램 실행이 가능하다는 것을 알게 되었습니다.

2. 제어 신호의 역할

제어 신호는 CPU의 각 동작을 제어하는 중요한 요소입니다. 예를 들어, RegDst 신호는 목적지 레지스터를 결정하고, ALUSrc 신호는 ALU의 두 번째 입력이 레지스터 값인지 즉시 값인지를 결정합니다. MemRead와 MemWrite 신호는 메모리 읽기와 쓰기 동작을 제어하며, PCSrc 신호는 프로그램 카운터의 소스를 결정합니다. 이 과제를 통해 각 제어 신호가 CPU의 동작에 어떻게 영향을 미치는지, 그리고 잘못된 제어 신호가 프로그램 실행에 얼마나 큰 영향을 미칠 수 있는지를 깊이 이해할 수 있었습니다.

3. 이론과 실제의 차이

이론적으로 배운 내용을 실제로 구현해보는 과정에서 많은 차이점을 느꼈습니다. 특히, 이론적으로 이해한 개념이 실제로 어떻게 구현되는지, 그리고 구현 과정에서 어떤 문제들이 발생할 수 있는지를 경험하게 되었습니다. 예를 들어, 이론적으로는 간단해 보였던 명령어 디코딩 과정이 실제로는 많은 세부 사항을 고려해야 한다는 것을 알게 되었습니다. 또한, 하버드 구조의 명령어 메모리와 데이터 메모리의 분리가 실제로 얼마나 효율적인지, 그리고 각각의 메모리가 별도의 버스를 이용해 처리되는 구조가 얼마나 중요한지를 깨달았습니다.

4. 하버드 구조의 장점

하버드 구조의 명령어 메모리와 데이터 메모리의 분리, 그리고 각각의 메모리가 별도의 버스를 이용해 처리되는 구조가 실제로 얼마나 효율적인지를 깨달았습니다. 이론적으로는 이해가 잘 가지 않았던 부분이었지만, 직접 MIPS 회로를 구현하면서 CS계의 엄청난 업적이라는 것을 실감할 수 있었습니다. 명령어와 데이터를 분리하여 처리함으로써, 동시에 여러 작업을 수행할 수 있는 이 구조가 얼마나 효과적인지 깨달았습니다.

5. 전기 신호의 흐름 통제

전기 신호가 항상 흐른다는 점과 이를 통제하는 것이 얼마나 중요한지 감명 깊었습니다. CPU가 0과 1의 신호로 모든 연산을 수행한다는 점은 알고 있었지만, 이를 실제로 구현하고 제어하는 과정에서 전기 신호의 흐름을 정확히 통제하는 것이 얼마나 중요한지 깨달았습니다. 특히, 잘못된 신호 흐름이 전체 시스템에 얼마나 큰 영향을 미칠 수 있는지, 그리고 이를 방지하기 위해 얼마나 많은 노력이 필요한지를 알게 되었습니다.

6. 과제를 하면서 겪은 어려움 및 깨달은 점

이번 프로젝트에서 MIPS 명령어를 이진수 형태로 직접 저장하고 처리하는 방식으로 초기 설계를 진행했습니다. 그러나 이진수로 표현된 데이터를 다루는 과정에서 몇 가지 중요한 어려움을 겪었습니다. 특히, 이진수에서 상위 비트의 '0'이 생략되어 '001111' 같은 코드가 '1111'로만 표현되는 경우, 전체 32비트 길이를 맞추기 위해 추가적인 처리가 필요했습니다. 이는 명령어 파싱 과정을 복잡하게 만들어 효율적인 구현을 어렵게 했습니다. 이 문제를 해결하기 위해, 명령어를 문자열로 저장하고 필요할 때 파싱하는 방식으로 변경했습니다. 이 변경은 데이터의 관리와 처리를 단순화시켜, 보다 명확하고 효율적인 코드 작성을 가능하게 했습니다.

또 다른 도전은 테스트 케이스 3번을 디버깅하는 과정에서 발생했습니다. 특정 입력에 대해 시뮬레이터가 약 50,000회의 사이클을 실행하면서 오류를 찾기 어려운 상황에 직면했습니다. 이 문제를 해결하기 위해 출력 결과를 fibonacci_koo.txt 파일로 리디렉션하여 디버깅했습니다. 이 접근 방식은 오류를 분석하고 수정하는 데 큰 도움이 되었습니다. 더 나아가, 무한 루프에 빠질 가능성을 발견하고, 이를 방지하기 위해 100번 이상 반복되면 자동으로 종료되는 코드를 추가함으로써 더욱 안정적인 시스템을 구현할 수 있었습니다.

7. Reference

MIPS, https://vbrunell.github.io/docs/MIPS%20Programming%20Guide.pdf

MIPS reference data, https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf