<프로젝트2 보고서>

컴퓨터구조와 모바일프로세서

싱글 사이클 MIPS 마이크로아키텍처 리포트

학　　과 : 모바일시스템공학과
학　　번 : 32164809
이　　름 : 탁준석
담당교수 : 유시환
제출일자 : 21.04.19
남은 프리데이 : 4일

목 차

1. 개요

가. 실험 목표

이 실험에서 우리는 MIPS-32를 기준으로 26가지 다양한 명령어들을 처리하는 싱글 사이클 MIPS 마이크로아키텍처 프로그램을 설계한다. 프로그램에서 구현할 MIPS 명령어 집합A, B, C는 다음과 같다. 이 집합들은 곱하기, 나누기 명령어와 부동 소수점 명령어를 포함하지 않는다. 프로그램을 작성했으면 간단한 바이너리 파일을 불러와 명령어가 원활하게 처리되는지 확인한다. 중간 실행 결과와 최종 실행 결과를 출력하는 기본 구현과 각종 명령어의 수, 메모리 접근 횟수, 분기한 횟수를 출력하는 추가 구현도 완료한다.

R-형식 명령어 집합A	{ add, addu, sub, subu, and, or, nor, slt, sltu, sll, srl, jr, jalr }
I-형식 명령어 집합B	{ addi, addiu, andi, ori, beq, bne, slti, sltiu, lw, lui, sw }
J-형식 명령어 집합C	{ j, jal }

<표 1.1> 싱글 사이클 MIPS 마이크로아키텍처 프로그램에서 구현할 명령어 집합

나. 실험 목적

우리는 저번 프로젝트1 간단한 계산기에서 폰 노이만 구조를 바탕으로 코드를 작성, 내장 프로그램과 순차 실행이라는 개념을 이해했다. 그리고 그 과정에서 메모리에 로드할 프로그램으로 MIPS 어셈블리어를 사용했다. 그렇다면 메모리에 로드되는 명령어는 실제로 어떠한 물리적인 과정을 거쳐서 그 종류가 구분되고, 값을 계산하고, 레지스터와 메모리에 저장되는 것일까?

이번 프로젝트2에서는 싱글 사이클 MIPS 마이크로아키텍처 코드를 작성하면서 추상적 개념이었던 명령어 집합 구조(ISA)가 어떻게 물리적인 하드웨어로 구현되는지 학습한다. 그리고 MIPS 명령어 집합과 싱글 사이클, 데이터패스, 그리고 제어신호의 개념에 대해 충분히 숙지한다. 명령어 집합이 하드웨어로 어떻게 표현되며 C언어와 같은 고급 언어와 명령어의 초보적 기능 사이의 연관관계를 자세히 이해해보자.

2. 이론

가. 명령어 집합 구조(Instruction Set Architecture)

컴퓨터 하드웨어에게 일을 시키려면 하드웨어가 알아들을 수 있는 언어로 말을 해야 한다. 하드웨어 언어인 기계어의 단어를 명령어라고 하고 그 어휘를 명령어 집합이라고 한다. 명령어 집합 구조(Instruction Set Architecture, ISA)는 프로세서가 인식해서 처리할 수 있는 명령어 집합, 접근할 수 있는 레지스터, 메모리와의 상호작용을 명세하게 기술한 것으로 컴퓨터 소프트웨어와 하드웨어 사이의 인터페이스를 정의한다. 우리가 이번 프로젝트2에 즐겨 쓸 ISA는 MIPS-32이다. MIPS 테크놀로지에서 개발한 고정길이(RISC) 명령어 집합 구조로 1980년대 초반 이후 설계된 명령어 중 대표적인 것이다. 그럼 지금부터 MIPS 명령어 집합 구조를 한 부분씩 살펴보도록 하자.

1) 명령어 형식(Instruction Format)

가) R-형식 명령어

R-형식 명령어는 <그림 2.1>과 같이 32비트 워드에 6가지 필드를 가지고 있는 명령어로 add, addu, sub, subu, and, or, nor, slt, sltu, sll, srl, jr, jalr 명령어가 이 형식을 만족한다. 각 필드의 역할은 <표 2.1>에서 확인할 수 있다.

opcode	rs	rt	rd	shamt	funct
6비트	5비트	5비트	5비트	5비트	6비트

<그림 2.1> R-형식 명령어의 구조

필드	의미
opcode(operation code)	명령어가 실행할 연산의 종류
rs(register source)	첫 번째 출처 피연산자 레지스터
rt(register target)	두 번째 출처 피연산자 레지스터
rd(register destination)	목적지 레지스터; 연산 결과를 기억한다.
shamt(shift amount)	자리 이동 양
funct(function)	실행할 작업을 표시; opcode 필드에서 연산의 종류를 표시하고 funct 필드에서는 그 중의 한 연산을 구체적으로 지정한다.

<표 2.1> R-형식 명령어의 필드

나) I-형식 명령어

I-형식 명령어는 <그림 2.2>과 같이 32비트 워드에 4가지 필드를 가지고 있는 명령어로 addi, addiu, andi, ori, slti, sltiu, beq, bne, lw, lui, sw 명령어가 이 형식을 만족한다. 각 필드의 역할은 <표 2.2>에서 확인할 수 있다.

opcode	rs	rt	immediate
6비트	5비트	5비트	16비트

<그림 2.2> I-형식 명령어 구조

필드	의미
opcode(operation code)	명령어가 실행할 연산의 종류
rs(register source)	첫 번째 출처 피연산자 레지스터
rt(register target)	목적지 레지스터; 연산 결과를 기억한다.
immediate	직접 피연산자

<표 2.2> I-형식 명령어의 필드

다) J-형식 명령어

J-형식 명령어는 <그림 2.3>과 같이 32비트 워드에 2가지 필드를 가지고 있는 명령어로 j, jal 명령어가 이 형식을 만족한다. 각 필드의 역할은 <표 2.3>에서 확인할 수 있다.

opcode	address
6비트	26비트

<그림 2.3> J-형식 명령어 구조

필드	의미
opcode(operation code)	명령어가 실행할 연산의 종류
address	프로그램 카운터에 저장되는 주소

<표 2.3> J-형식 명령어의 필드

2) 레지스터(Register)

고급언어 프로그램과 달리 MIPS 명령어의 피연산자는 아무 변수나 막 쓸 수 있지 않고, 레지스터라고 하는 제한된 개수의 특수 위치에 있는 것만을 사용할 수 있다. 레지스터는 하드웨어 설계에 사용하는 기본 요소인 동시에 프로그래머에게도 보이는 부분이다. MIPS에서는 32비트 크기의 범용 레지스터를 32개 지원하는데 레지스터 번호에 따른 이름과 사용법은 다음 <표 2.4>와 같다.

레지스터 이름	레지스터 번호	사용법
$0	0	상수 0
$at	1	어셈블러 인시용
$v0-v1	2-3	프로시저 반환 값
$a0-a3	4-7	프로시저 인자
$t0-t7	8-15	임시 변수
$s0-s7	16-23	저장 변수
$t8-t9	24-25	임시 변수
$k0-k1	26-27	운영체제 임시용
$gp	28	전역 포인터
$sp	29	스택 포인터
$fp	30	프레임 포인터
$ra	31	프로시저 반환 주소

<표 2.4> MIPS ISA가 지원하는 레지스터

나. 마이크로아키텍처(Micro Architecture)

마이크로아키텍처란 프로세서의 여러 구성요소와 요소 간 상호 연결을 통해 명령어 집합 구조를 구현한 것을 말한다. 일반적으로 마이크로아키텍처는 메모리, 산술논리 장치, 레지스터, 논리 게이트와 같은 구성요소들이 서로 체계적으로 연결된 블록 다이어그램으로 표현한다. 이 다이어그램은 앞서 말한 요소들을 선으로 연결한 데이터패스(Datapath)와 그들의 작동여부를 제어하는 제어부(Control path)로 나눌 수 있다.

마이크로아키텍처 프로그램을 작성하기 위해서 우리는 MIPS 명령어가 어떠한 순서로 처리되는지 알아야한다. 다행히도, 일부 명령어를 제외하고 대부분의 명령어는 그 순서가 거의 비슷하다. 모든 명령어에 대해 처음 두 단계는 동일하고, 이후 명령을 완결하는데 필요한 조치는 명령어의 종류에 따라 조금 다르다.

1. Instruction fetch
   명령어를 가져오기 위해서 명령어 메모리(Instruction Memory)에 프로그램 카운터(PC)를 보낸다.

2. Instruction decode and register operand fetch
   PC가 가리키는 명령어 워드를 분석해 사용할 레지스터를 선택하고, 선택된 레지스터를 읽는다.

3. Execute and evaluate memory address
   명령어가 지시하는 명령을 ALU를 통해 계산한다. 메모리 참조 명령어는 유효주소를 계산하기 위해, 산술논리 명령어는 Opcode의 실행을 위해, 분기 명령어는 비교를 위해 ALU를 사용한다.

4. Memory operand fetch
   메모리를 참조하는 명령어는 저장을 완결하기 위해 또는 로드될 워드를 읽기 위해 데이터를 가지고 있는 메모리(Data Memory)에 접근한다.

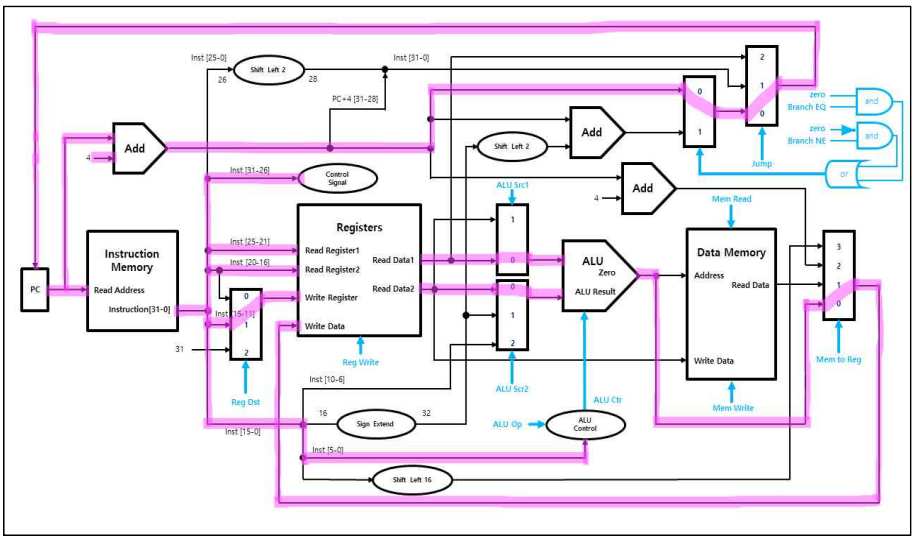
5. Store and write back result
   산술논리 명령어는 계산한 결과를 쓰기 레지스터에 저장한다. 메모리를 참조하는 명령어는 메모리에 계산한 결과를 저장한다.

1) 데이터패스(Datapath)

데이터패스 설계를 시작하는 합리적인 방법은 MIPS 명령어를 실행하는데 필요한 주요 구성요소를 종류별로 알아보는 것이다. 각각의 명령어가 필요로 하는 데이터패스 원소에 대해 알아보고 이 원소들을 가지고 명령어 종류별로 데이터패스의 각 부분들을 어떻게 만들어 가는지에 대해 살펴보도록 하자. 여기서 프로그램 카운터가 다음 명령어의 주소를 가리키게 하는 것과 같이 명령어마다 완전히 동일하게 구현되는 부분들은 두 번 설명하지 않고 생략할 것임을 알린다.

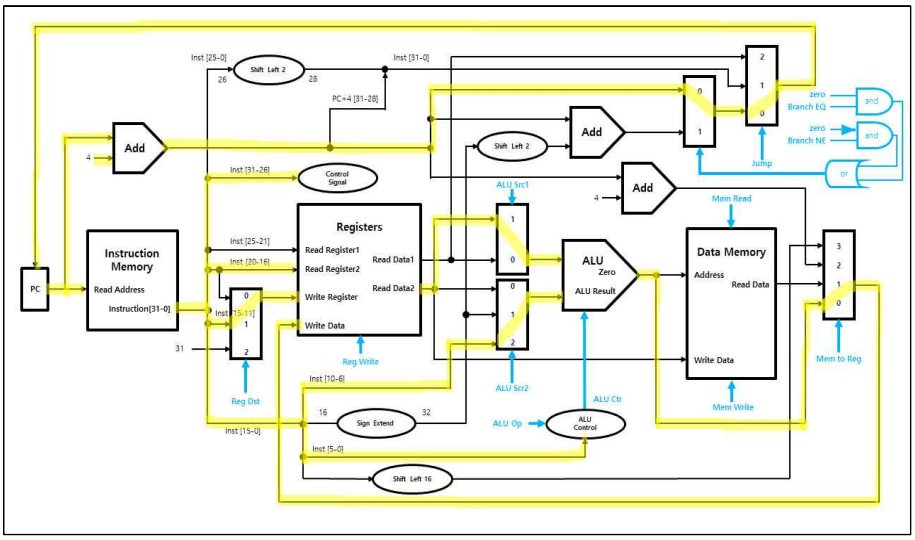
가) R-형식 명령어를 위한 데이터패스

add, addu, sub, subu, and, or, nor, slt, sltu 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, ALU, 그리고 가산기를 연결한 데이터패스로 표현할 수 있다. 먼저, 명령어를 실행하기 위해서는 메모리로부터 명령어를 인출해야 한다. 또한 다음 명령어의 실행을 준비하기 위해 가산기로 프로그램 카운터를 증가시켜 4바이트만큼 떨어진 다음 명령어를 가리키도록 해야 한다. 이 명령어들은 세 개의 레지스터 피연산자(rs, rt, rd)를 가지고 있기 때문에 read Register1, read Register2, write Register를 위한 세 입력과, read Data1, read Data2를 위한 두 출력이 있어야 한다. ALU는 두 데이터를 계산해 ALU Result로 내보낸다. 이 결과는 다시 레지스터 파일의 write Data로 전달되어 레지스터에 저장된다.



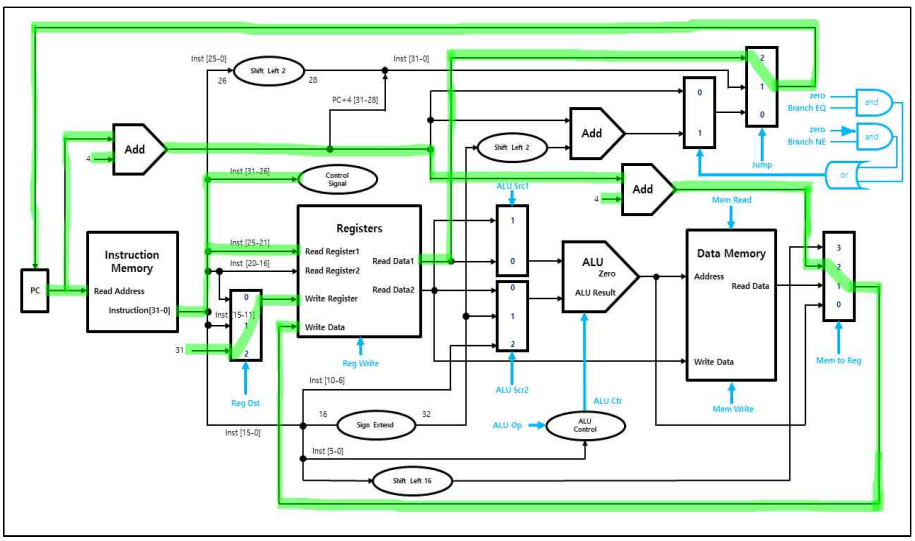
<그림 2.4> add, addu, sub, subu, and, or, nor, slt, sltu 명령어의 데이터패스

sll, srl 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, ALU, 그리고 가산기를 연결한 데이터 패스로 표현할 수 있다. sll, srl 명령어는 R-형식 명령어이지만 레지스터 피연산자로 rs를 사용하지 않는다. 대신 read Register2를 위한 입력 rt에서 나온 출력 데이터가 멀티플렉서를 거쳐 ALU의 첫 번째 피연산자로 사용된다. ALU의 두 번째 피연산자는 명령어의 shift amount가 사용된다. ALU는 read Data2를 shift amount만큼 비트 연산을 하고 결과를 ALU Result로 내보낸다. 이 결과는 다시 레지스터 파일의 write Data로 전달되어 레지스터에 저장된다.



<그림 2.5> sll, srl 명령어의 데이터패스

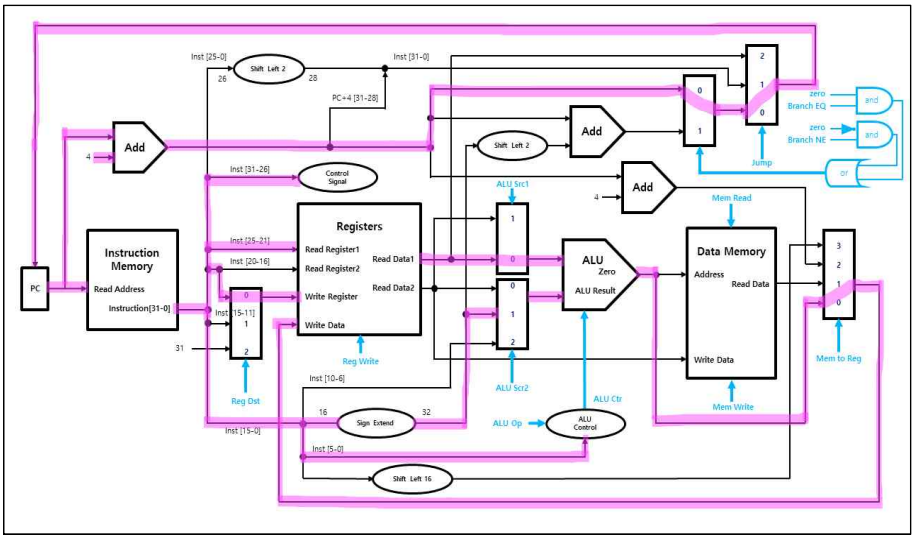
jr, jalr 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, 가산기를 연결한 데이터패스로 표현할 수 있다. jalr 명령어는 write Register에 31, write Data에 PC + 8을 전달하고, PC에는 레지스터 rs로부터 얻은 새 주소 read Data1를 저장한다. j 명령어는 PC에 새 주소 read Data1을 저장만 하면 된다.



<그림 2.6> jr, jalr 명령어의 데이터패스

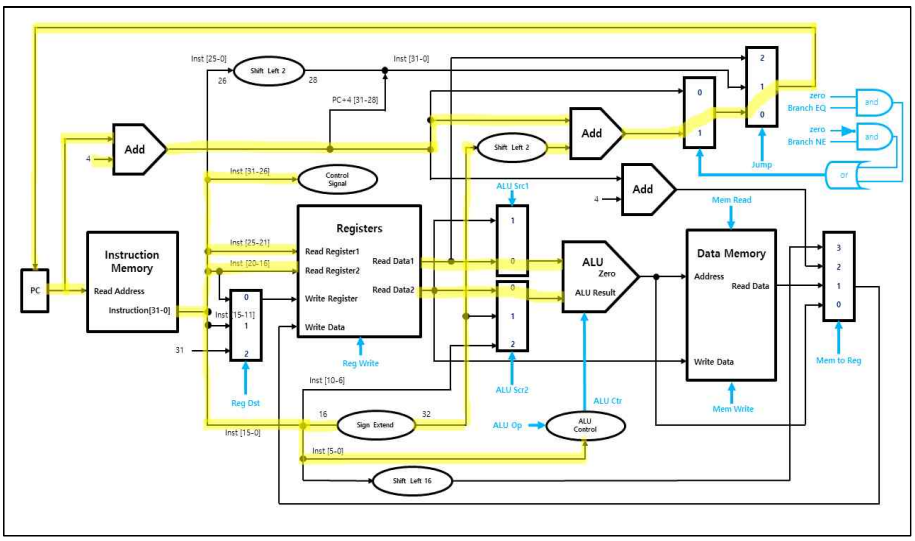
나) I-형식 명령어를 위한 데이터패스

addi, addiu, andi, ori, slti, sltiu 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, ALU, 그리고 가산기를 연결한 데이터패스로 표현할 수 있다. 이 데이터패스는 R-형식 명령어의 산술논리 명령어와 와 유사하나 조금 차이가 있다. Write Register로 레지스터 피연산자인 rd를 사용하지 않고 rt를 사용한다. ALU는 Read Data1과 부호가 확장된 32비트 immediate를 계산해 ALU Result로 내보낸다. 이 결과는 다시 레지스터 파일의 Write Data로 전달되어 레지스터에 저장된다.



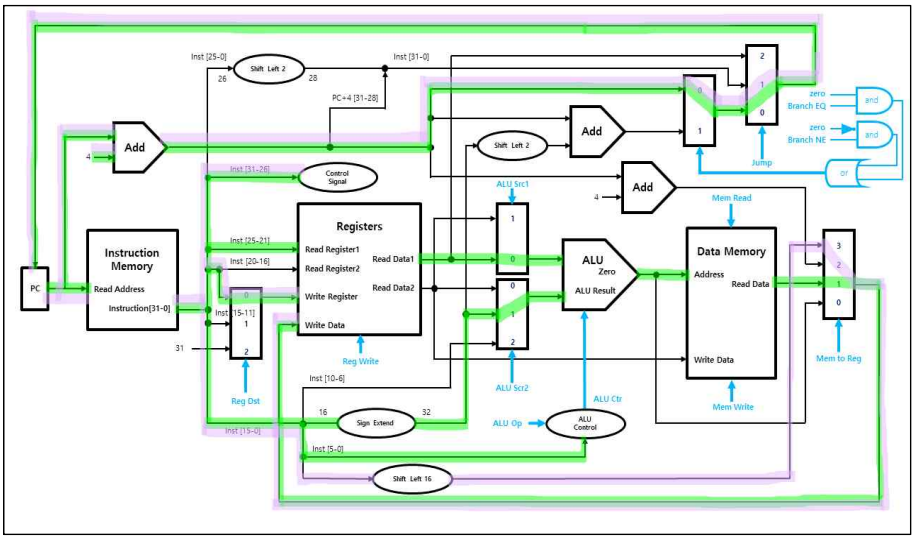
<그림 2.7> addi, addiu, andi, ori, slti, sltiu 명령어의 데이터패스

beq, bne 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, ALU, 그리고 가산기를 연결한 데이터패스로 표현할 수 있다. 분기 명령어는 레지스터 파일에서 write Register를 요구하지 않고, 레지스터 피연산자 rs, rt로부터 도출된 read Data1과 read Data2의 크기를 비교한다. 크기 비교는 sub 연산으로 구현할 수 있다. 그 결과가 분기 조건을 만족하면 프로그램 카운터에 새 주소 PC+4 + { 14{immediate[15]}, immediate, 2'b0 }이 저장되고, 만족하지 않으면 PC+4가 저장된다.



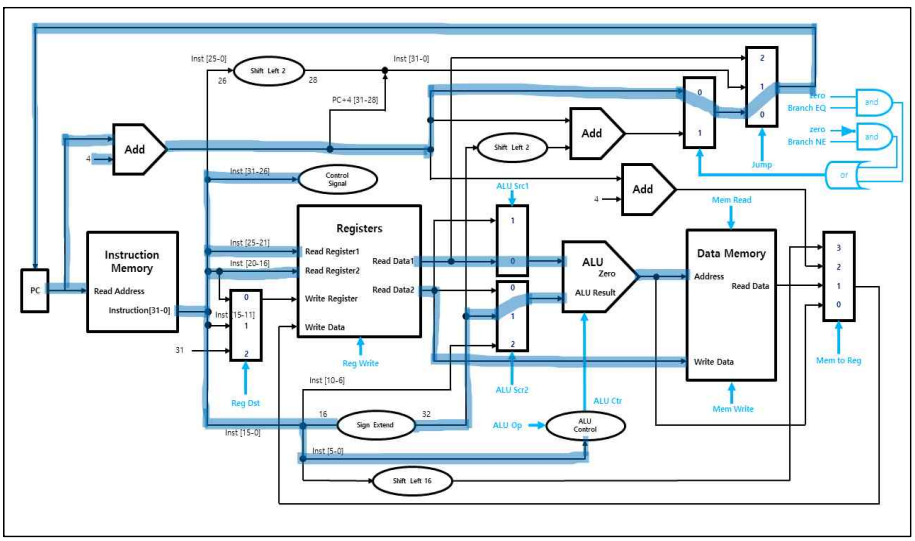
<그림 2.8> beq, bne 명령어의 데이터패스

lw 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, ALU, 데이터 메모리 그리고 가산기를 연결한 데이터패스로 표현할 수 있다. 레지스터 피연산자 rs로 도출된 read Data1과 부호가 확장된 immediate를 더해 ALU의 result로 내보낸다. 데이터 메모리에서 result에 해당하는 address에 저장된 값을 읽어 write Data로 내보낸다. 레지스터 피연산자 rt로 도출된 write Register에 write Data를 저장한다.
lui 명령어는 다른 명령어와 다르게 프로그램 카운터, 명령어 메모리, 가산기를 연결한 데이터패스로 표현할 수 있다. immediate를 오른쪽으로 16비트만큼 이동해 32비트의 워드를 만들고 write Data로 내보낸다. write Register에 해당하는 레지스터 rt에 write Data를 저장한다.



<그림 2.9> lw, lui 명령어의 데이터패스

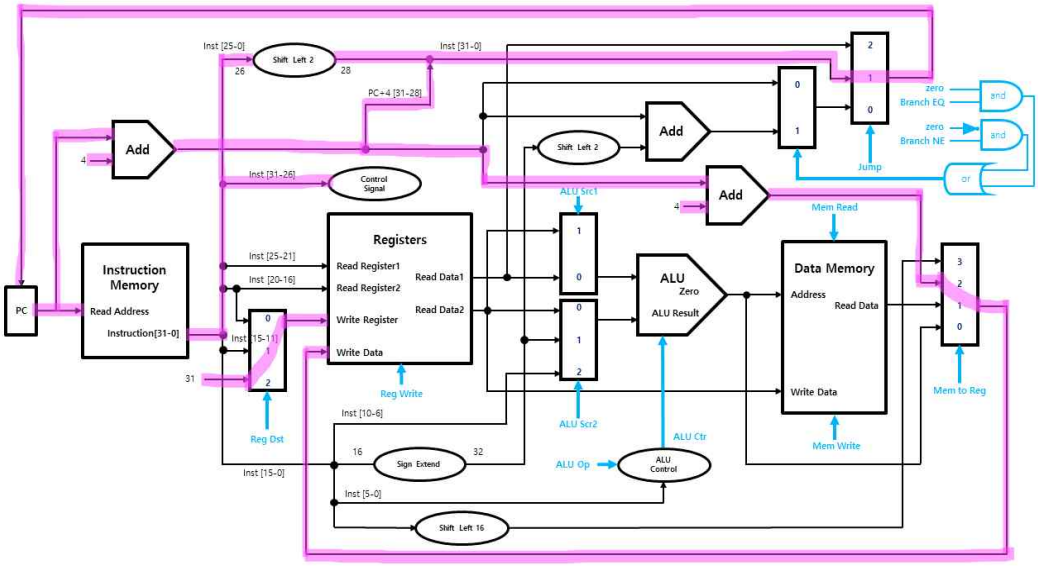
sw 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, ALU, 데이터 메모리 그리고 가산기를 연결한 데이터패스로 표현할 수 있다. 레지스터 피연산자 rs로 도출된 read Data1과 부호가 확장된 immediate를 더해 ALU의 result로 내보낸다. 레지스터 피연산자 rt로 얻은 read Data2를 데이터 메모리의 write Data로 전달한다. 데이터 메모리에서 result에 해당하는 address에 write Data를 저장한다.



<그림 2.10> sw 명령어의 데이터패스

다) J-형식 명령어를 위한 데이터패스

j, jal 명령어는 프로그램 카운터, 명령어 메모리, 레지스터 파일, 가산기를 연결한 데이터패스로 표현할 수 있다. jal 명령어는 write Register에 31, write Data에 PC + 8을 전달하고, PC에는 새 주소 { PC+4[31:28], address, 2'b0 }을 저장한다. j 명령어는 PC에 새 주소 { PC+4[31:28], address, 2'b0 }을 저장만 하면 된다.



<그림 2.11> j, jal 명령어의 데이터패스

2) 멀티플렉서(Multiplexer)

멀티플렉서(Multiplexer, MUX)는 여러 입력 신호들 중에서 하나의 신호를 선택해 출력 신호로 내보내는 논리 회로를 말한다. 우리는 MIPS 명령어의 종류에 따라서 데이터패스의 모양이 조금씩 다르다는 것을 확인했다. 따라서 데이터패스를 하나로 통합할 때, 서로 다른 명령어 형식 사이에서 데이터패스의 구성 요소를 공유하기 위해선 여러 입력 데이터를 연결하는 다중 연결을 구현해야 한다. 이 문제는 멀티플렉서와 제어 논리 신호 적절하게 사용해 입력 데이터가 올바른 목적지에 도착할 수 있도록 함으로써 해결할 수 있다. 이번 프로젝트2에는 모두 6가지의 멀티플렉서(MUX1, MUX2_1, MUX2_2, MUX3, MUX4, MUX5)가 구현되어있다. 각 멀티플렉서의 입력 데이터와 제어 신호는 다음 <표 2.5>와 같다.

멀티플렉서	제어 신호	입력 데이터	번호
MUX1	RegDst	InstInfo.rt	0
		InstInfo.rd	1
		31	2
MUX2_1	ALUSrc1	readData1	0
		readData2	1
MUX2_2	ALUSrc2	readData2	0
		EXT.Zero, EXTSign	1
		InstInfo.shamt	2
MUX3	MemtoReg	ALU.result	0
		DataMem.readData	1
		PC+4	2
		InstInfo.imme << 16	3
MUX4	BranchEqual BranchNotEqual zero	PC	0
		PC+brachAddress	1

MUX5	Jump	PC	0
		jumpAddress	1
		ALU.data1	2

<표 2.5> 프로젝트2에서 사용된 멀티플렉서의 입력 데이터와 제어 신호

3) 제어 논리(Control Logic)

가) 제어 신호(Control Signal)

데이터패스를 통합하면서 새롭게 제시된 멀티플렉서와 데이터 메모리, 그리고 레지스터 파일 쓰기는 명령어에 따라 사용할 수도, 사용하지 않을 수 있다. 따라서 명령어가 어떤 데이터패스에 따라 처리되어야 하는지 제어 논리를 도입해 통제할 필요가 있다.

제어부에 대한 입력은 명령어의 상위 6비트인 opcode로부터 얻을 수 있다. 제어부의 출력은 멀티플렉서를 제어하는데 쓰이는 RegDst, Jump, ALUSrc1, ALUSrc2, MemtoReg 신호, 레지스터 파일과 데이터 메모리에서 읽고 쓰는 것을 제어하기 위한 MemRead, MemWrite, Regwrite 신호로 나눌 수 있다. 다음 <표 2.6>는 이번 프로젝트2에서 사용한 2-입력, 3-입력, 4-입력 멀티플렉서의 제어 신호를 나타낸다.

신호	거짓 효과	참 효과		
	0	1	2	3
RegDst	레지스터 목적지 번호가 rt로부터 주어진다.	레지스터 목적지 번호가 rd로부터 주어진다.	레지스터 목적지 번호가 31이다.	.
Jump	PC+4를 계산하는 가산기의 출력이 PC에 전달된다.	jumpAddress가 PC에 전달된다.	Register[rs]가 PC에 전달된다.	.
Branch	PC+4를 계산하는 가산기의 출력이 PC에 전달된다.	PC+4+branchAddress가 PC에 전달된다.	.	.
ALUSrc1	ALU의 첫 번째 피연산자가 레지스터 파일의 첫 번째 출력으로부터 주어진다.	ALU의 첫 번째 피연산자가 레지스터 파일의 두 번째 출력으로부터 주어진다.	.	.
ALUSrc2	ALU의 두 번째 피연산자가 레지스터 파일의 두 번째 출력으로부터 주어진다.	ALU의 두 번째 피연산자가 명령어의 부호가 확장될 하위 16비트로부터 주어진다.	ALU의 두 번째 피연산자가 명령어의 하위 15-10비트로부터 주어진다.	.
MemRead	없음	읽기 주소에 있는 데이터 메모리 내용이 읽기 데이터 출력으로 나온다.	.	.
MemWrite	없음	쓰기 주소에 있는 데이터 메모리 내용이 쓰기 데이터로 바뀐다.	.	.
MemtoReg	write Data로 보낼 입력을 ALU에서 전달받는다.	write Data로 보낼 입력을 데이터 메모리에서 전달받는다.	write Data로 보낼 입력을 PC+8을 계산하는 가산기에서 전달받는다.	write Data로 보낼 입력을 왼쪽으로 16비트만큼 이동한 immediate에서 전달받는다.
Regwrite	없음	write Register 입력에 해당하는 레지스터에 write Data로 쓰기가 행해진다.	.	.

<표 2.6> 멀티플렉서를 제어하는 제어 신호와 그 효과

나) ALU 제어(ALU Control)

이번 프로젝트2에서는 명령어의 opcode에 따른 ALUOp와 funct에 따른 ALU Control 3비트 제어 신호를 구현해 ALU가 제어 신호에 따른 8가지 간소화된 계산을 제공하도록 하였다. 0b000의 ALUOp 신호를 가지는 R-형식 명령어는 ALUCTR 신호를 통해 ALU가 어떤 계산을 수행해야 하는지 결정한다. 그 외 0b001부터 0b101의 ALUOp 신호를 갖는 명령어들은 ALUCTR 신호를 사용하지 않는다. 한편, 명령어에 부여된 ALUOp 신호와 ALUCTR 신호는 다음 <표 2.7>에서 확인할 수 있다.

ALU Op	명령어	ALU CTR	효과
000	jr, jalr을 제외한 R-형식	000	add
001	andi	001	or
010	ori	010	nor
011	addi, addiu, lw, sw	011	sll
100	beq, bne	100	srl
101	slti, sltiu	101	add
110	.	110	sub
111	.	111	slt

<표 2.7> 3비트 ALU Op와 3비트 ALU CTR

다음 <표 2.8>은 지금까지 설명한 명령어 형식, 멀티플렉서, 제어 신호, ALU 제어가 명령어 별로 총망라되어 있다. 비어있는 칸은 이번 프로젝트에서는 구현하지 않는 명령어이며, x는 don't care로 어떤 신호를 가지든지 명령어 처리에 아무런 영향이 없다는 뜻이다.

타입	명령	Opcode (hex)	Function (hex)	Control											
				RegDst	Jump	Branch	ALUSrc1	ALUSrc2	ALUOp	ALU Ctr	ALU Action	MemRead	MemWrite	MemtoReg	RegWrite
R type	add	0	20	1	0	0	0	0	000	101	add	0	0	0	1
	add unsigned	0	21	1	0	0	0	0	000	101	add	0	0	0	1
	and	0	24	1	0	0	0	0	000	000	and	0	0	0	1
	jump register	0	08	x	2	x	x	x	x	x	.	0	0	x	0
	jump and link register	0	09	2	2	x	x	x	x	x	.	0	0	2	1
	nor	0	27	1	0	0	0	0	000	010	nor	0	0	0	1
	or	0	25	1	0	0	0	0	000	001	or	0	0	0	1
	set less than	0	2a	1	0	0	0	0	000	111	slt	0	0	0	1
	set less than unsigned	0	2b	1	0	0	0	0	000	111	slt	0	0	0	1
	shift left logical	0	00	1	0	0	1	2	000	011	sll	0	0	0	1
	shift right logical	0	02	1	0	0	1	2	000	100	slr	0	0	0	1
	subtract	0	22	1	0	0	0	0	000	110	sub	0	0	0	1
	subtract unsigned	0	23	1	0	0	0	0	000	110	sub	0	0	0	1
I type	add immediate	8	.	0	0	0	0	1	011	101	add	0	0	0	1
	add immediate unsigned	9	.	0	0	0	0	1	011	101	add	0	0	0	1
	and immediate	c	.	0	0	0	0	1	001	000	and	0	0	0	1
	branch on equal	4	.	x	0	1	0	0	100	110	sub	0	0	x	0
	branch on not equal	5	.	x	0	1	0	0	100	110	sub	0	0	x	0
	load byte unsigned	24													
	load halfword unsigned	25													
	load linked	30													
	load upper immediate	f	.	0	0	0	0	0	x	x	.	0	0	3	1
	load word	23	.	0	0	0	0	1	011	101	add	1	0	1	1
	or immediate	d	.	0	0	0	0	1	010	001	or	0	0	0	1
	set less than immediate	a	.	0	0	0	0	1	101	111	slt	0	0	0	1
	set less than immediate unsigned	b	.	0	0	0	0	1	101	111	slt	0	0	0	1
	store byte	28													
	store conditional	38													
	store halfword	29													
	store word	2b	.	x	0	0	0	1	011	101	add	0	1	x	0
J type	jump	2	.	x	1	x	x	x	x	x	.	0	0	x	0
	jump and link	3	.	2	1	x	x	x	x	x	.	0	0	2	1

<표 2.8> MIPS 명령어의 제어 신호와 ALU 제어

3. Single Cycle MIPS Micro Architecture

가. 개발 환경

Single Cycle MIPS Micro Architecture 프로그램은 Windows 10 운영 체제에서 Visual Studio 2019를 통해 C언어로 작성되었다. simple.bin, simple2.bin, simple3.bin, simple4.bin, fib.bin, gcd.bin, input4.bin 파일은 SingleCycleMain.c와 같은 폴더 안에 있어야 한다.

나. 프로그램 분석

이 프로그램은 사용자가 프로그램을 더 쉽게 읽고 이해할 수 있도록 SingleCycleHeader.h, SingleCycleMain.c, 그리고 SingleCycleFunction.c 세 부분으로 나누어 작성되었다. 먼저, Header.h에서는 프로그램에서 사용한 헤더 파일, 전역 변수, 구조체 그리고 함수를 살펴보며 전체적인 틀을 확인한다. 다음으로 Function.c를 분석하면서 위에서 설명한 데이터패스가 어떻게 구현되었는지 자세히 살펴보자. 마지막으로 Main.c를 확인하며 하나의 명령어가 처리되는 싱글 사이클이 어떻게 완성되는지 알아보자. 프로그램 분석 내용은 데이터패스 구현을 중심으로 서술해 일부 코드는 생략되어 있으니 참고하길 바라며, 개인적으로 추가 구현한 내용은 글자 색을 빨갛게 해놓았음을 알린다.

1) SingleCycleHeader.h

SingleCycleHeader.h에는 헤더 파일, 전역 변수, 구조체, 그리고 함수가 선언되어 있다.

```
(생략)

extern int Memory[0x00400000];
extern int Register[32];
extern unsigned int PC;
extern unsigned int binIndex;
unsigned int countInst;
unsigned int countR, countI, countJ;
unsigned int countMemoryAccess;
unsigned int TakenBranch;
int UserSelect;
clock_t start, end;
float TimeExcuted;

(생략)
```

① 16MB에 해당하는 메모리를 위한 배열 Memory[0x00400000]을 선언한다.
② MIPS의 32가지 범용 레지스터를 위한 배열 Register[32]를 선언한다.
③ 다음 명령어의 주소를 저장할 PC 변수와 메모리 배열의 인덱스로 사용할 binIndex 변수를 선언한다.
④ 프로젝트2 추가구현 목록에 해당하는 총 명령어의 수, R-형식 명령어의 수, I-형식 명령어의 수, J-형식 명령어의 수, 메모리 접근 횟수, 분기한 횟수를 위한 변수들을 선언한다.
⑤ 프로그램 총 실행 시간을 측정하기 위한 start, end, TimeExcuted 변수를 선언한다.

```
typedef struct {
    int InstData;
} InstructionFormat;

typedef struct {
    int opcode;
    int rs;
    int rt;
    int rd;
    int imme;
    int funct;
    int shamt;
    int addr;
} InstructionInfomation;

typedef struct {
    int readReg1;
    int readReg2;
    int writeReg;
    int readData1;
    int readData2;
    int writeData;
} Registers;

typedef struct {
    int data1;
    int data2;
    bool zero;
    int result;
} ArithmeticLogicUnit;
```

```
typedef struct {
    int RegDst;
    int Jump;
    int BranchEqual;
    int BranchNotEqual;
    int ALUSrc1;
    int ALUSrc2;
    int ALUCtr;
    int ALUOp;
    int MemtoReg;
    int MemRead;
    int MemWrite;
    int RegWrite;
} ControlSignal;

typedef struct {
    int writeData;
    int readData;
    int address;
} DataMemory;

typedef struct {
    int jumpAddress;
    int branchAddress;
    int Sign;
    int Zero;
} Extend;

(중략)
```

① 구조체 InstForm는 명령어 워드를 저장하는 구조체로, 32비트 명령어 워드를 저장할 InstData 변수가 선언되어 있다.

② 구조체 InstInfo는 명령어의 필드를 저장하는 구조체로, Opcode, rs, rt, rd, shamt, imme, addr, funct 변수가 선언되어 있다.

③ 구조체 REG는 레지스터 파일의 정보를 저장하는 구조체로 읽기 및 쓰기 레지스터 readReg1, readReg2, writeReg 변수와 읽기 및 쓰기 데이터 readData1, readData2, writeData 변수가 선언되어 있다.

④ 구조체 ALU는 ALU의 입력과 출력을 저장하는 구조체로, 첫 번째 피연산자와 두 번째 피연산자를 저장하는 data1, data2 변수와 결과를 저장하는 result 변수, 그리고 분기 명령어의 제어 신호로 사용할 zero 변수가 선언되어 있다.

⑤ 구조체 CTR는 제어 신호의 정보를 저장하는 구조체로, 앞서 설명했던 제어 신호와 ALU 제어에 사용된 변수들이 선언되어 있다.

⑥ 구조체 DataMem는 데이터 메모리의 입력과 출력을 저장하는 구조체로, ALU의 결과를 저장할 address 변수와 읽기 및 쓰기 데이터 readData, writeData 변수가 선언되어 있다.

⑦ 구조체 EXT는 16비트 immediate를 연장하는 구조체로, 점프할 주소를 저장하는 jumpAddress, branchAddress 변수와 32비트로 확장된 immediate를 저장할 Sign, Zero 변수가 선언되어 있다.

2) SingleCycleFunction.c

Function.c는 바이너리 파일을 메모리에 로드하는 함수부터 메모리나 레지스터에 결과를 저장하는 함수까지 데이터패스에 해당하는 모든 부분이 구현되어 있다. Function.c에서는 명령어가 처리되는 순서대로 여러 함수들을 천천히 설명하고자 한다.

가) DoMemory 함수

DoMemory 함수는 우리가 컴퓨터에서 실행할 프로그램, 바이너리 파일들을 메모리에 로드하는 역할을 한다.

```
#include "SingleCycleHeader.h"

void DoMemory(void) {

    (중략)

    while (1) {
        scanf_s("%s", fileName, 32);
        Error = fopen_s(&binFile, fileName, "rb");
        if (binFile == 0 || Error == 1) {
            printf("\n\tbin 파일 불러오기 실패! 파일을 다시 입력하세요. 〉 ");
            continue;
        }
        else
            break;
    }
```

① scanf_s 함수로 사용자가 바이너리 파일의 이름을 입력한다.
② 입력한 문자열을 fileName 배열에 저장되고 binFile 파일 포인터가 fileName을 가리킨다.
③ fopen_s 함수로 바이너리 파일을 바이너리 읽기 모드로 연다.
④ 파일의 이름을 잘못 입력할 경우, continue를 통해 사용자가 파일의 이름을 다시 입력할 수 있도록 한다.
⑤ 파일이 정상적으로 열렸을 경우, break를 통해 while 반복문을 빠져나간다.

```
    while (1) {
        fread(&Memory[Index], sizeof(int), 1, binFile);

        if (feof(binFile))
            break;
        else {
            InstForm.InstData =
                (((Memory[Index] & 0x000000ff) << 24) & 0xff000000) |
                (((Memory[Index] & 0x0000ff00) << 8) & 0x00ff0000) |
                (((Memory[Index] & 0x00ff0000) >> 8) & 0x0000ff00) |
                (((Memory[Index] & 0xff000000) >> 24) & 0x000000ff);
            Memory[Index] = InstForm.InstData;
    (중략)
```

① fread 함수로 binFile 스트림에서 4바이트 크기의 명령어 하나를 읽는다.
② fread 함수로 읽은 명령어는 앞서 설명한 개발 환경으로 인해 바이트 순서가 리틀 엔디언(Little

Endian)으로 저장되어 있으므로, 더 쉬운 명령어 처리를 위해 바이트 순서를 빅 엔디언(Big Endian)으로 바꾼다.

③ feof 함수에 binFile 파일 포인터를 전달해 파일의 끝이 아니면 0, 파일의 끝이면 0이 아닌 값을 반환하도록 한다.

④ 바이너리 파일을 모두 읽었으면 0이 아닌 값을 반환하게 되고 break를 통해 while 반복문을 빠져나간다.

나) DoFetch 함수

DoFetch 함수는 제 1단계, Instruction Fetch의 역할을 한다.

```
void DoFetch(void) {
    binIndex = PC / 4;
    PC = PC + 4;
    return;
}
```

① PC가 4바이트만큼 증가하므로 PC를 4로 나눈 값을 binIndex에 저장한다.

② PC가 다음 명령어를 가리킬 수 있도록 4를 더해 다시 PC에 저장한다.

다) DoDecode 함수

DoDecode 함수는 제 2단계, Instruction Decode의 역할을 한다.

```
void DoDecode(void) {
    InstInfo.opcode = ((Memory[binIndex] & 0xfc000000) >> 26) & 0x0000003f;
    InstInfo.rs = ((Memory[binIndex] & 0x03e00000) >> 21) & 0x0000001f;
    InstInfo.rt = ((Memory[binIndex] & 0x001f0000) >> 16) & 0x0000001f;
    InstInfo.rd = ((Memory[binIndex] & 0x0000f800) >> 11) & 0x0000001f;
    InstInfo.shamt = ((Memory[binIndex] & 0x000007c0) >> 6) & 0x0000001f;
    InstInfo.funct = Memory[binIndex] & 0x0000003f;
    InstInfo.imme = Memory[binIndex] & 0x0000ffff;
    InstInfo.addr = Memory[binIndex] & 0x03ffffff;

    (중략)

    DoControl();
    DoRegister();
    SignExtend();
    ZeroExtend();
    return;
}
```

① 명령어의 Opcode는 31-26비트 부분에 해당한다. 31-26비트를 1, 25-0비트를 0으로 하는 16진수(0xfc00:0000)와 and 연산을 하고 오른쪽으로 26만큼 비트를 이동시킨다. 다음으로 비트를 이동시킬 때 Opcode에 따라서 0이 아닌 1로 채워질 수 있으므로, 이를 방지하기 위해 31-6번째 비트를 0, 5-0번째 비트를 1로하는 16진수(0x0000:003f)와 다시 한 번 and 연산을 한다.

② 명령어의 rs, rt, rd, shamt, funct, imme, addr도 위와 같은 방법으로 구할 수 있다.

③ 명령어의 각 필드를 구분해 구조체 InstInfo의 변수로 모두 저장했으면 DoControl 함수, DoRegister 함수, SigeExtend 함수, 그리고 ZeroExtend 함수를 순서대로 실행한다.

(1) DoControl 함수

DoControl 함수는 DoDecode 함수에서 얻은 Opcode와 funct를 가지고 명령어의 제어 신호를 결정하는 역할을 한다. 앞서 설명한 <표 2.7>을 기준으로 if-else문을 사용해 제어 신호를 초기화한다.

```
void DoControl(void) {
    /***** Default Control *****/
    CTR.RegDst = 0;
    CTR.Jump = 0;
    CTR.BranchEqual = 0;
    CTR.BranchNotEqual = 0;
    CTR.ALUOp = 0b000;
    CTR.ALUSrc1 = 0;
    CTR.ALUSrc2 = 1;
    CTR.MemRead = 0;
    CTR.MemWrite = 0;
    CTR.MemtoReg = 0;
    CTR.RegWrite = 1;

    /***** RegDst *****/
    if    (InstInfo.opcode == 0x0) // R-type
        CTR.RegDst = 1;
    if    (InstInfo.opcode == 0x3 ||
        (InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x9)) // jal, jalr
        CTR.RegDst = 2;

    /***** MemtoReg *****/
    if    (InstInfo.opcode == 0x23) // lw
            CTR.MemtoReg = 1;
    if    (InstInfo.opcode == 0x3 ||
        (InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x9)) // jal, jalr
            CTR.MemtoReg = 2;
    if    (InstInfo.opcode == 0xf) // lui
            CTR.MemtoReg = 3;

    /***** MemRead, MemWrite *****/
    if    (InstInfo.opcode == 0x23) // lw
            CTR.MemRead = 1;
    if    (InstInfo.opcode == 0x2b) // sw
            CTR.MemWrite = 1;

    /***** Jump *****/
    if    (InstInfo.opcode == 0x2 ||
        InstInfo.opcode == 0x3) // j, jal
            CTR.Jump = 1;
    if    ((InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x8) ||
        (InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x9)) // jr, jalr
            CTR.Jump = 2;

    /***** Branch(Not)Equal *****/
    if    (InstInfo.opcode == 0x4) // beq
            CTR.BranchEqual = 1;
    if    (InstInfo.opcode == 0x5) // bne
            CTR.BranchNotEqual = 1;

    /***** RegWrite *****/
    if    (InstInfo.opcode == 0x4 ||
        InstInfo.opcode == 0x5 ||
        InstInfo.opcode == 0x2b ||
        InstInfo.opcode == 0x2 ||
        (InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x8))
            CTR.RegWrite = 0;

    /***** ALUSrc1, 2 *****/
    if    (InstInfo.opcode == 0x0 ||
        InstInfo.opcode == 0x4 ||
        InstInfo.opcode == 0x5)
            CTR.ALUSrc2 = 0;

    if ((InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x0) ||
        (InstInfo.opcode == 0x0 &&
        InstInfo.funct == 0x2)) { // sll, srl
            CTR.ALUSrc1 = 1;
            CTR.ALUSrc2 = 2;
    }

    /***** ALUOp *****/
    if    (InstInfo.opcode == 0x0)
            CTR.ALUOp = 0b000; // R-type
    if    (InstInfo.opcode == 0xc)
            CTR.ALUOp = 0b001; // andi
    if    (InstInfo.opcode == 0xd)
            CTR.ALUOp = 0b010; // ori
    if    (InstInfo.opcode  ==  0x8  ||  //  addi(u)
        InstInfo.opcode == 0x9 ||
        InstInfo.opcode == 0x23 ||
        InstInfo.opcode == 0x2b)
            CTR.ALUOp = 0b011; // lw, sw
    if    (InstInfo.opcode == 0x4 ||
        InstInfo.opcode == 0x5)
            CTR.ALUOp = 0b100; // beq, bne
    if    (InstInfo.opcode == 0xa ||
        InstInfo.opcode == 0xb)
            CTR.ALUOp = 0b101; // slti(u)

    return;
}
```

(2) DoRegister 함수

DoRegister 함수는 명령어의 필드에서 읽은 레지스터 번호를 레지스터 파일의 읽기 및 쓰기 레지스터 read Register, write Register로 전달한다.

```
void DoRegister(void) {
    REG.readData1 = Register[InstInfo.rs];
    REG.readData2 = Register[InstInfo.rt];
    MUX1();
    return;
}
```

① 명령어의 rs, rt 필드에 해당하는 레지스터에 저장되어 있는 값을 불러와 쓰기 데이터 readData1, readData2에 각각 저장한다.

② MUX1 함수로 세 가지 입력 중에서 쓰기 레지스터로 사용될 출력을 내보낸다.

(가) MUX1 함수

MUX1 함수는 제어 신호 RegDst에 따라서 rt, rd, 그리고 31, 이 세 가지의 입력 중에서 출력 write Register를 결정하는 멀티플렉서의 역할을 한다.

```
void MUX1(void) {// RegDst MUX
    switch (CTR.RegDst) {
    case 0:// I 형식 명령어(beq, bne 명령어 제외)
        REG.writeReg = InstInfo.rt;
        break;

    case 1:// R 형식 명령어(jr, jalr 명령어 제외)
        REG.writeReg = InstInfo.rd;
        break;

    case 2:// jal, jalr 명령어
        REG.writeReg = 31;
        break;

    default:
        break;
    }
    return;
}
```

① CTR.RegDst가 0인 경우, 명령어의 rt가 레지스터 파일의 write Register에 저장된다. beq와 bne 명령어를 제외한 I-형식 명령어에 해당한다.

② CTR.RegDst가 1인 경우, 명령어의 rd가 레지스터 파일의 write Register에 저장된다. jr과 jalr을 제외한 R-형식 명령어에 해당한다.

③ CTR.RegDst가 2인 경우, 31이 레지스터 파일의 write Register에 저장된다. jal과 jalr 명령어에 해당한다.

(3) SignExtend 함수와 ZeroExtend 함수

SignExtend 함수와 ZeroExtend 함수는 immediate를 16비트에서 32비트로 확장하는 역할을 한다.

```
void SignExtend(void) {
    switch (InstInfo.imme >> 15) {
    case 0:
        EXT.Sign =  InstInfo.imme |
                    0x00000000;
        break;
    case 1:
        EXT.Sign =  InstInfo.imme |
                    0xffff0000;
        break;
    default:
        break;
    }
    return;
void ZeroExtend(void) {
    EXT.Zero =  InstInfo.imme |
                0x00000000;
    return;
}
```

① SignExtend 함수는 immediate의 부호를 생각해서 16비트에서 32비트로 연장한다. immediate의 부호를 확인하기 위해서 InstInfo.imme를 오른쪽으로 15비트만큼 이동시킨다.

② 그 값이 0인 경우, immediate는 양수이므로 0x0000:0000과 or 연산을 한다.

③ 그 값이 1인 경우, immediate는 음수이므로 0xffff:0000과 or 연산을 한다.

④ ZeroExtend 함수는 immediate의 부호를 생각하지 않고 16비트에서 32비트로 연장한다. 따라서 0x0000:0000과 or 연산을 한다.

라) DoExecute 함수

DoExecute 함수는 제 3단계, Execute의 역할을 한다.

```
void DoExecute(void) {
    DoALUControl();
    MUX2_1();
    MUX2_2();

    switch (CTR.ALUCtr) {
    case 0b000: // and
    ALU.result = ALU.data1 & ALU.data2;
        break;

    case 0b001: // or
    ALU.result = ALU.data1 | ALU.data2;
        break;

    case 0b010: // nor
    ALU.result = ~(ALU.data1 | ALU.data2);
        break;


    case 0b011: // sll
    ALU.result = ALU.data1 << ALU.data2;
        break;

    case 0b100: // srl
    ALU.result = ALU.data1 >> ALU.data2;
        break;

    case 0b101: // add
    ALU.result = ALU.data1 + ALU.data2;
        break;

    case 0b110: // sub
    ALU.result = ALU.data1 - ALU.data2;
        if (ALU.result == 0)
            ALU.zero = true;
        else
            ALU.zero = false;
        break;

    case 0b111: // slt
    ALU.result = (ALU.data1 < ALU.data2) ?
                 1 : 0;
        break;

    (중략)
```

① ALU Control 신호가 0일 경우, ALU의 두 입력 데이터를 and 연산하고 result에 결과를 저장한다.
② ALU Control 신호가 1일 경우, ALU의 두 입력 데이터를 or 연산하고 result에 결과를 저장한다.
③ ALU Control 신호가 2일 경우, ALU의 두 입력 데이터를 nor 연산하고 result에 결과를 저장한다.
④ ALU Control 신호가 3일 경우, ALU의 두 입력 데이터를 sll 연산하고 result에 결과를 저장한다.
⑤ ALU Control 신호가 4일 경우, ALU의 두 입력 데이터를 srl 연산하고 result에 결과를 저장한다.
⑥ ALU Control 신호가 5일 경우, ALU의 두 입력 데이터를 and 연산하고 result에 결과를 저장한다.
⑦ ALU Control 신호가 6일 경우, ALU의 두 입력 데이터를 sub 연산하고 result에 결과를 저장한다.
⑧ ALU Control 신호가 7일 경우, ALU의 두 입력 데이터를 slt 연산하고 result에 결과를 저장한다.

(1) DoALUControl 함수

DoALUControl 함수는 opcode에서 나온 ALUOp과 funct를 가지고 3비트 ALU CTR 신호를 초기화한다. 이 제어 신호는 앞서 설명한 <표 2.7>을 바탕으로 설계되었다.

```
void DoALUControl(void) {
    switch (CTR.ALUOp) {
    case 0b000:
        if (InstInfo.funct == 0x24) // and
            CTR.ALUCtr = 0b000;
        else if (InstInfo.funct == 0x25) // or
            CTR.ALUCtr = 0b001;
        else if (InstInfo.funct == 0x27) // nor
            CTR.ALUCtr = 0b010;
        else if (InstInfo.funct == 0x00) // sll
            CTR.ALUCtr = 0b011;
        else if (InstInfo.funct == 0x02) // srl
            CTR.ALUCtr = 0b100;
        else if (InstInfo.funct == 0x20 ||
            InstInfo.funct == 0x21) // add(u)
            CTR.ALUCtr = 0b101;
        else if (InstInfo.funct == 0x22 ||
            InstInfo.funct == 0x23) // sub(u)
            CTR.ALUCtr = 0b110;
        else if (InstInfo.funct == 0x2a ||
            InstInfo.funct == 0x2b) // slt,(u)
            CTR.ALUCtr = 0b111;
        else if (InstInfo.funct == 0x8) // jr
            CTR.ALUCtr = -1;
        else if (InstInfo.funct == 0x9) // jalr
            CTR.ALUCtr = -1;
        else;
        break;

    case 0b001: // andi (and 이용)
        CTR.ALUCtr = 0b000;
        break;

    case 0b010: // ori (or 이용)
        CTR.ALUCtr = 0b001;
        break;

    case 0b011: // addi(u), lw, sw (add 이용)
        CTR.ALUCtr = 0b101;
        break;

    case 0b100: // beq, bne (sub 이용)
        CTR.ALUCtr = 0b110;
        break;

    case 0b101: // slti(u) (slt 이용)
        CTR.ALUCtr = 0b111;
        break;

    default:
        CTR.ALUCtr = -1; // don't care
        break;
    }
    return;
}
```

① ALUOp 신호가 0b000이고 funct 필드가 0x24인 명령어는 and에 해당, 0b000을 ALU CTR에 저장한다.
② 이와 같이 ALUOp 신호가 0b000이면 funct 필드에 따라서 명령어 해야 할 연산이 결정되고 그에 알맞은 제어 신호를 ALU CTR에 저장한다.
③ ALUOp 신호가 0b001이면 andi 명령어에 해당하므로 ALU CTR에 0b000을 저장한다.
④ ALUOp 신호가 0b010이면 ori 명령어에 해당하므로 ALU CTR에 0b001을 저장한다.
⑤ ALUOp 신호가 0b011이면 addi, addiu, lw, sw 명령어에 해당하고, add 연산을 할 수 있도록 ALU CTR에 0b101을 저장한다.

⑥ ALUOp 신호가 0b100이면 beq, bne 명령어에 해당하고, 분기 조건에 해당하는 비교는 sub 연산으로 구현하므로 ALU CTR에 0b110을 저장한다.

⑦ ALUOp 신호가 0b101이면 slti, sltiu 명령어에 해당하므로 ALU CTR에 0b111을 저장한다.

마) DoMemoryAccess 함수

DoMemoryAccess 함수는 제 4단계, Memory operand fetch의 역할을 한다.

```
void DoMemoryAccess(void) {
    DataMem.address = ALU.result;
    DataMem.writeData = REG.readData2;

    if (CTR.MemRead == 1) { // load word
        DataMem.readData = Memory[DataMem.address / 4];
        countMemoryAccess++;
    }
    else if (CTR.MemWrite == 1) { // store word
        Memory[DataMem.address / 4] = DataMem.writeData;
        countMemoryAccess++;
    }
    else;
    return;
}
```

① ALU에서 연산을 마친 결과를 데이터 메모리의 address에 저장한다.

② 레지스터 파일의 read Data2의 값을 데이터 메모리의 write Data에 저장한다.

③ Memread 제어 신호가 1인 lw 명령어는 address/4의 주소를 가지는 메모리에서 값을 로드한다.

④ 이와 반대로 Memwrite 제어 신호가 1인 sw 명령어는 데이터 메모리의 쓰기 데이터 값을 address/4 주소를 가지는 메모리에 저장한다.

바) DoWriteBack 함수

DoWriteBack 함수는 제 5단계, Store and write back result의 역할을 한다.

```
void DoWriteBack(void) {
    MUX3();
    switch (CTR.RegWrite) {
    case 0:
        break;

    case 1:
        Register[REG.writeReg]
        = REG.writeData;
        break;
    default:
        break;
    }
    MUX4();
    MUX5();
    return;
}
```

① 먼저 MUX3 함수에서 write Data를 전달 받는다.

② 제어 신호 RegWrite가 0일 경우, 레지스터 파일에 쓰기 행위를 하지 않는다.

③ 제어 신호 RegWrite가 1일 경우, 전달받은 write Data를 레지스터에 저장한다.

④ MUX4 함수, MUX5 함수에서 PC를 업데이트한다.

(1) MUX3 함수

MUX3 함수는 제어 신호 MemtoReg에 따라서 ALU.result, DataMem.readData, PC+4, InstInfo.imme << 16, 네 가지의 입력 중에서 출력 write Data를 결정하는 멀티플렉서의 역할을 한다.

```
void MUX3(void) { // MemtoReg MUX
    switch (CTR.MemtoReg) {
    case 0:
        REG.writeData = ALU.result;
        break;

    case 1:
        REG.writeData = DataMem.readData;
        break;

    case 2: // jal, jalr
        REG.writeData = PC + 4;
        break;

    case 3: // lui
        REG.writeData = InstInfo.imme << 16;
        break;

    default:
        break;
    }
    return;
}
```

① 제어 신호 MemtoReg가 0일 경우, ALU의 연산 결과를 write Data에 저장한다.
② 제어 신호 MemtoReg가 1일 경우, 데이터 메모리의 read Data를 write Data에 저장한다.
③ 제어 신호 MemtoReg가 2일 경우, PC+4를 write Data에 저장한다.
④ 제어 신호 MemtoReg가 3일 경우, immediate를 왼쪽으로 16비트만큼 이동한 값을 write Data에 저장한다.

(2) MUX4 함수

MUX4 함수는 BranchEqual, BranchNotEqual, 그리고 zero를 논리 회로로 조합한 제어 신호로 PC, PC+branchAddress, 두 입력 중에서 출력 PC를 결정하는 멀티플렉서의 역할을 한다.

```
void MUX4(void) {// Branch MUX
    BranchAddress();

    if ((CTR.BranchEqual == 1) && (ALU.zero == true)) {// beq
        PC = PC + EXT.branchAddress;
        TakenBranch++;
    }
    else if ((CTR.BranchNotEqual == 1) && (!ALU.zero == true)) {// bne
        PC = PC + EXT.branchAddress;
        TakenBranch++;
    }
    else
        PC = PC;
    return;
}
```

① zero가 참이고 BranchEqual 제어 신호가 1일 경우, beq 명령어의 분기 조건이 만족하므로 PC에 PC+branchAddress를 저장한다.
② zero가 거짓이고 BranchNotEqual 제어 신호가 0일 경우, bne 명령어의 분기 조건이 만족하므로 마찬가지로 PC에 PC+branchAddress를 저장한다.
③ 분기 조건을 만족하지 않는 경우, PC에 아무런 변화를 주지 않는다.

(3) MUX5 함수

MUX5 함수는 jump 제어 신호로 PC, PC+jumpAddress, ALU Data1, 세 입력 중에서 출력 PC를 결정하는 멀티플렉서의 역할을 한다.

```
void MUX5(void) {// Jump MUX
    JumpAddress();

    switch (CTR.Jump) {
    case 0:
        PC = PC;
        break;

    case 1:// j, jal
        PC = EXT.jumpAddress;
        break;

    case 2:// jr, jalr
        PC = ALU.data1;
        break;

    default:
        break;
    }
    return;
}
```

① 제어 신호 jump가 0일 경우, PC에 아무런 변화를 주지 않는다.
② 제어 신호 jump가 1일 경우, PC+jumpAddress를 저장한다.
③ 제어 신호 jump가 2일 경우, PC에 ALU Data1, 즉 rs번째 레지스터를 저장한다.

3) SingleCycleMain.c

SingleCycleMain.c는 Function.c에서 정의된 함수를 반복해서 호출해 하나의 명령어를 수행하는 싱글 사이클의 개념을 완성한다. 또한 기본 구현과 추가 구현에 해당하는 함수를 호출해 결과를 확인할 수 있도록 한다.

```
void main(void) {
    start = clock(); // 시간 측정 시작
    Register[31] = 0xffffffff;
    Register[29] = 0x01000000; // Stack Pointer
    DoMemory();
```

① clock 함수로 프로그램 시간 측정을 시작한다.
② Register[31]($ra)을 0xffff:ffff로 초기화한다.
③ Register[29]($sp)를 0x0100:0000으로 초기화한다.
④ DoMemory 함수로 bin 파일의 명령들을 메모리 배열에 각각 저장한다.

```
    while (PC != 0xffffffff)  {
        DoFetch();              // Instruction Fetch
        DoDecode();             // Instruction Decode
        DoExecute();            // Execute
        DoMemoryAccess();       // Memory Access
        DoWriteBack();          // Write Back

        countInst++;            // Number of Instruction + 1
        BasicState();         } // 기본구현 출력
```

① 싱글 사이클 마이크로아키텍처가 명령을 처리하는 과정을 총 5단계(DoFetch, DeDecode, DeExcute, DoMemoryAccess, DoWriteBack)로 구분하고, 이것을 한 사이클로 정의한다.
② 한 사이클이 끝나면 countInst 변수로 명령 수를 업데이트하고 BasicState 함수로 기본구현 목록을 출력한다.
③ 더 처리할 명령이 없을 경우, 프로그램 카운터(PC)가 –1이 되어 while 반복문을 빠져나온다.

```
    end = clock(); // 시간 측정 종료
    excutedTime = end - start;
    excutedTime /= 1000;
    AdditionalState(); // 추가구현 출력
    return;
}
```

① 프로그램 시간 측정을 종료하고, 프로그램 실행시간을 구한다.
② AdditionalState 함수가 추가구현 목록을 출력한다.

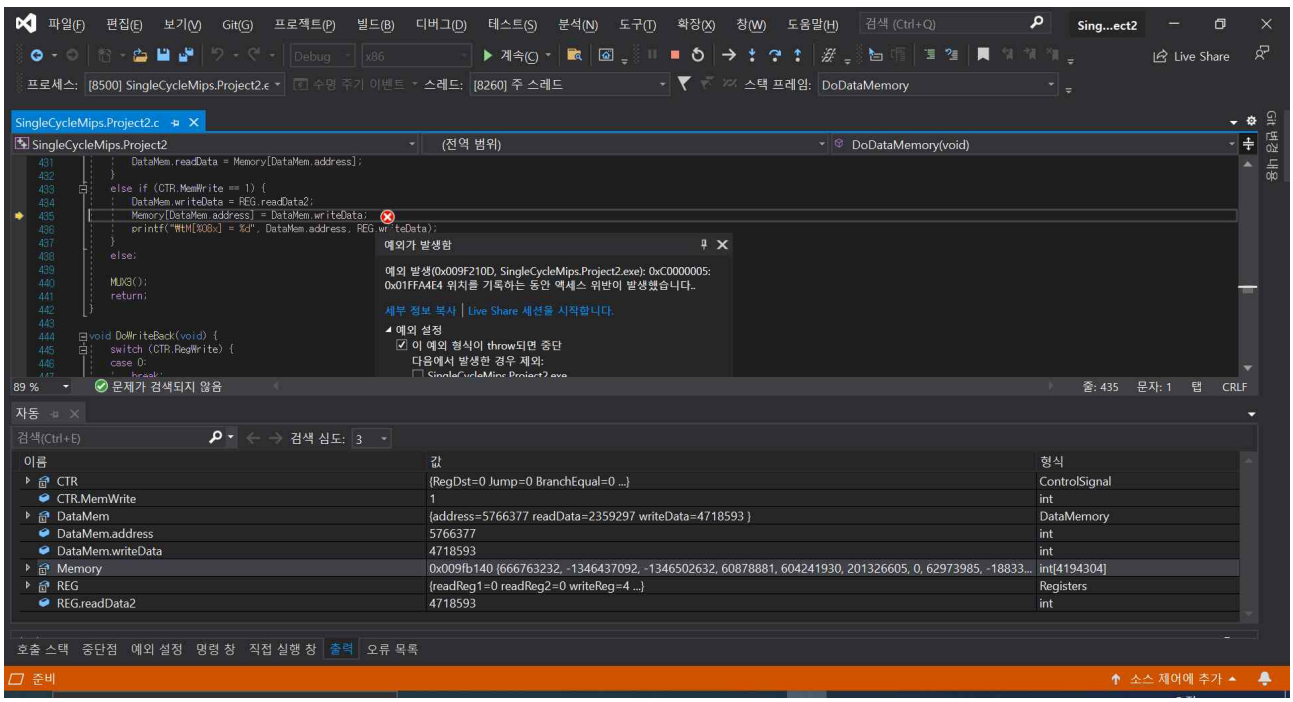
4. 문제 및 해결

제 4장 문제 및 해결에서는 프로그램을 작성하면서 발생했던 몇 가지 문제와 그 해결 과정을 담았다. 600줄이 넘는 코드를 다루면서 시간이 꽤 걸렸던 디버깅 과정, 그리고 프로그램의 실행 시간을 줄이기 위해 노력했던 과정을 서술해보고자 한다.

가. 디버깅

1) Memory Access와 / 4의 누락

첫 번째 오류는 simple4.bin 파일을 실행하는 동안 <그림 4.1>과 같은 메시지와 함께 발생하였다.



<그림 4.1> simple4.bin에서 데이터 메모리의 address를 4로 나누지 않자 발생한 오류

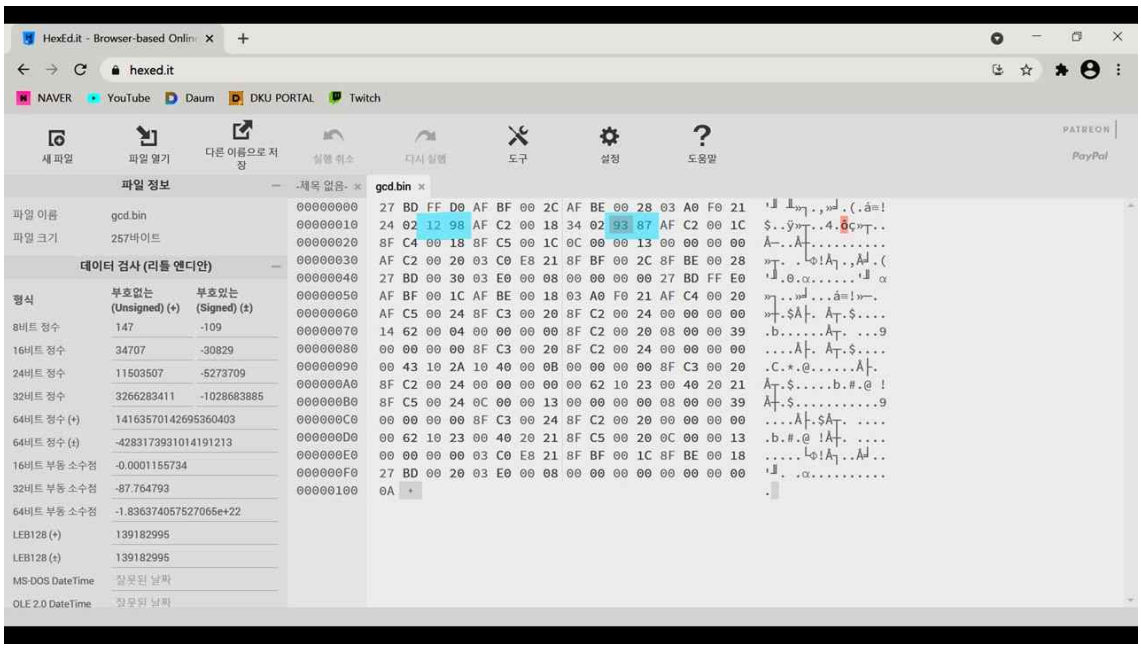
비주얼 스튜디오의 지역 창을 확인했을 때 데이터 메모리의 Address가 5,766,377로 메모리의 크기 0x400000에 해당하는 4,194,304를 초과하는 값이었다. 이 Address 변수가 메모리 배열에 인자로 그대로 전달되어 존재하지 않는 배열의 값을 읽고 쓰려고 했던 과정에서 오류가 발생했고, 곧 데이터 메모리의 Address로 Memory 배열에 접근할 때 4로 나누어주지 않았다는 실수를 범했다는 것을 알게 되었다. 따라서 Memory[DataMem.address];를 Memory[DataMem.address / 4]로 수정했고, simple4.bin이 올바르게 작동하게 되었다.

2) gcd.bin과 Zero Extend 함수의 누락

두 번째 오류는 gcd.bin 파일을 실행하는 동안 0xc0000005: 0x01b0ab20 위치를 기록하는 동안 액세스 위반이 발생했다는 메시지와 함께 발생하였다. 오류의 원인을 밝히기 위해서 몇 가지 가능성 있는 조건을 세웠다.

① 모든 숫자의 최대 공약수가 구해지지 않는가? 그렇다면 함수를 잘못 작성했거나, 데이터패스 연결에 문제가 있을 수 있다.
② 일부 숫자의 최대 공약수만 구해지지 않는가? 그렇다면 오류를 일으키는 숫자와 그렇지 않은 숫자와의 차이는 무엇인가? 차이가 발생하는 코드는 정확하게 작성되었는가?

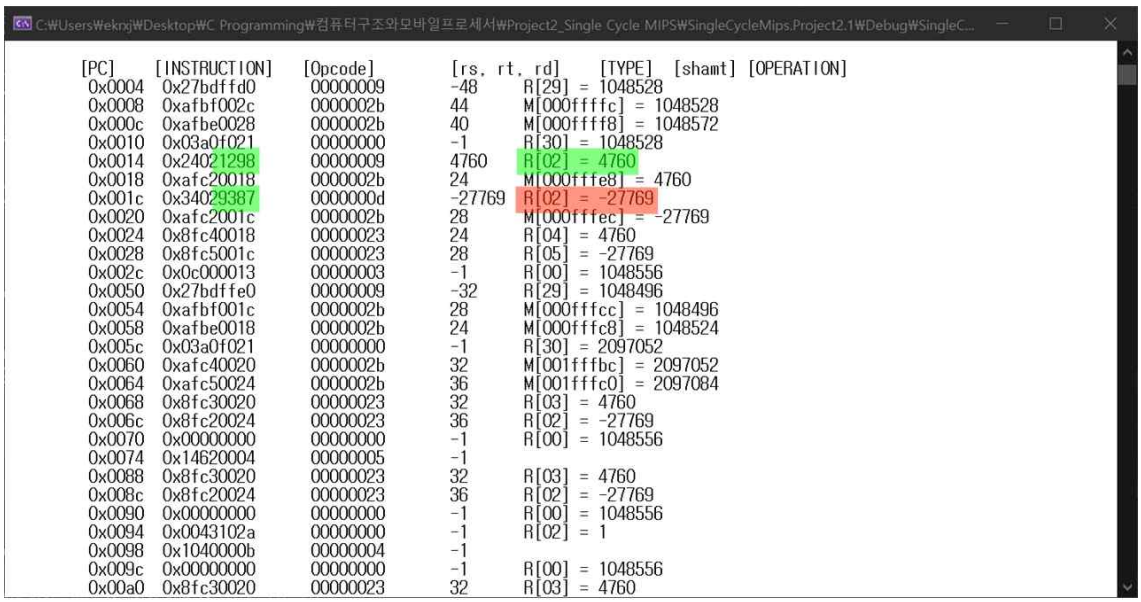
위의 조건을 확인하기 위해 최대 공약수를 구할 두 수를 여러 가지 숫자로 바꾸어보았다. 원본 바이너리 파일에서 4760(0x1298), 37767(0x9387)에 해당하는 부분을 찾았고, 최대 공약수를 구할 여러 가지 숫자, 가령 10과 100으로 바꾸어 gcd2.bin으로 저장했다. 여러 가지 gcd2.bin 파일을 실행한 결과, 모든 숫자가 아닌 일부 숫자의 최대 공약수만 구해지지 않는다는 사실을 밝혀냈다.



<그림 4.2> hexedit 사이트에서 파란 색으로 표시된 바이너리 코드를 수정했다.

최대 공약수가 구해지지 않는 gcd2.bin 파일을 로드하고 프로그램을 한 단계씩 실행하며 중간 계산을 비교해본 결과, 프로그램 카운터가 가리키는 0x1C 명령어에 분명히 37767 저장되어야 할 R[2]($v0)가 <그림 4.2>와 같이 37767의 2의 보수, 즉 –27769가 저장되었다는 것을 발견했다. 해당 숫자를 저장하는 코드를 다시 살펴봤을 때, DoDecode 함수 안에서 호출되어야 할 ZeroExtend 함수가 누락되어 호출되지 않아 SignExtend 함수가 대신 호출되고 있었고 최상위 비트가 1이었던 0x9387이 부호 확장되면서

–27769가 저장되고 있었다. ZeroExtend 함수를 넣어주자 올바른 값이 전달되었고 프로그램이 원활하게 작동하였다.



<그림 4.3> 0x9387이 37767이 아닌 –27769로 $v0에 저장되었다.

나. 프로그램 구조화 및 최적화

1) MUX 함수의 도입

프로그램의 크기가 점점 커지면서 프로그램을 체계적으로 구조화할 필요성이 자연스럽게 나타났다. 그 과정에서 DoRegister, DoExecute, DoWriteBack 함수 안에 if-else문으로 지나치게 길게 작성되었던 멀티플렉서 코드를 새로운 함수로 분리하여 작성키로 했다.

우선, 앞서 설명한 <표 2.5>를 바탕으로 하는 6가지 새로운 멀티플렉서 함수 MUX1, MUX2_1, MUX2_2, MUX3, MUX4, MUX5를 선언하고, 함수 안에 길게 작성되었던 코드를 잘라 멀티플렉서 함수로 정의했다. 다음으로 제어 신호 구분하는 코드를 if-else문이 아닌 switch문으로 교체하여 출력을 결정하도록 했다. 제어 신호를 switch문의 조건으로 전달하면서 멀티플렉서 함수가 더 짧고 간단하게 정의되었고 또, 여러 오류들을 디버깅할 때 오류의 원인을 파악하는데 큰 도움이 되었다.

2) 복잡한 DoExecute 함수와 ALU CTR의 도입

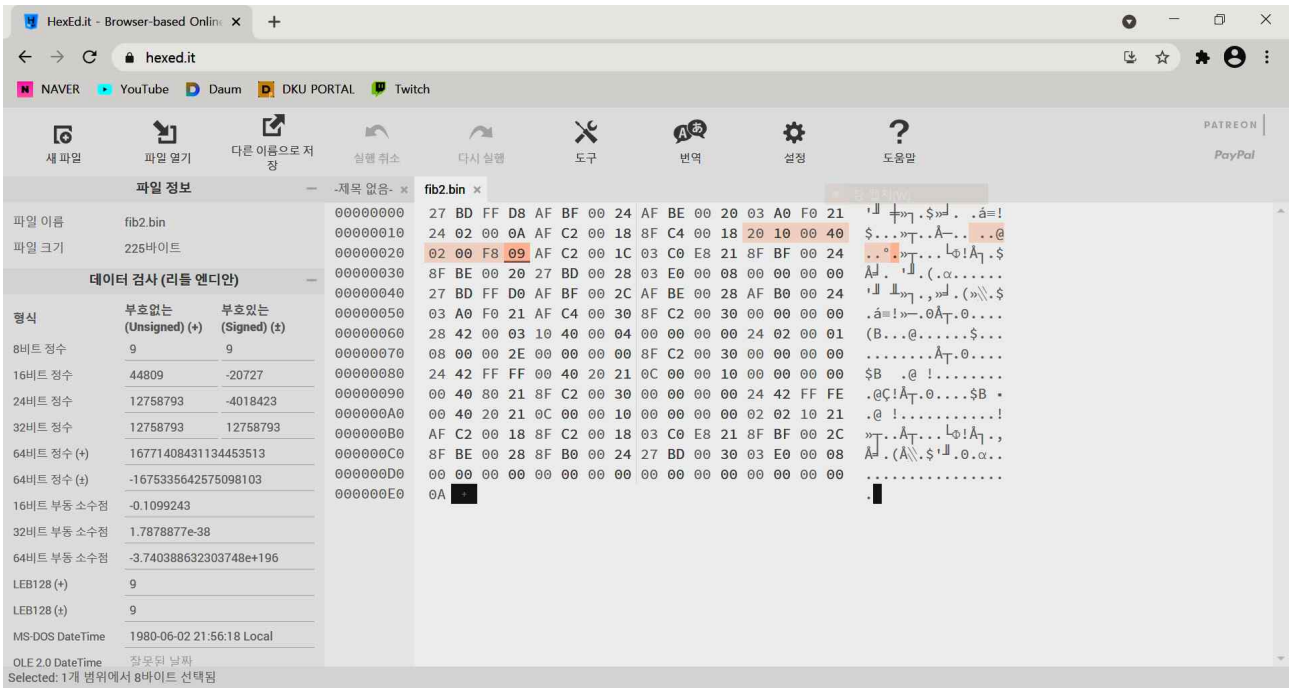
처음 제 3단계, Execute에 해당하는 DoExecute 함수를 정의할 때, opcode와 funct을 if-else문의 조건으로 넣고 26가지 명령어를 일일이 구분하여 해당 명령어가 필요로 하는 연산을 각각 작성했다. 물론 프로그램이 작동하는데 논리적 오류가 없었기에 결과 출력에는 문제가 없었지만, 엄밀하게 따지면 opcode는 ALU를 직접 제어하는 신호가 아니었고 또한 26번에 달하는 지나친 if-else문의 사용으로 코드가 매우 알아보기 힘들었다.

이 문제는 새로운 제어 신호 ALU Ctr을 도입해 해결했다. 같은 연산을 공유하는 명령어는 피연산자와 관계없이 같은 ALU에 제어 신호로 묶었다. 가령 add, add immediate, load word, 그리고 store word는 add 연산을 공유하고 subtract, subtract unsigned, 그리고 branch on equal은 sub 연산을 공유한다. 연산을 공유하는 명령어에게 같은 ALU Ctr 신호를 부여하기 위해 새로운 함수 DoALUControl 함수를 정의했고 앞서 설명한 <표 2.7>과 같이 연산과 ALU Ctr 신호를 대응하는 코드를 작성해주었다. 그 결과,

DeExecute 함수 안에서 switch문으로 총 8가지의 기본 연산을 제공하는 ALU를 구현하게 되었다. 덕분에 명령어마다 중구난방으로 레지스터 파일의 read Data1, 2와 부호가 확장된 immediate를 여러 번 가져오는 것에서 단 두 개의 피연산자 ALU Data1, ALU Data2만을 사용하게 되었고, 코드도 80줄에서 30줄로 대폭 줄이는 효과를 얻을 수 있었다

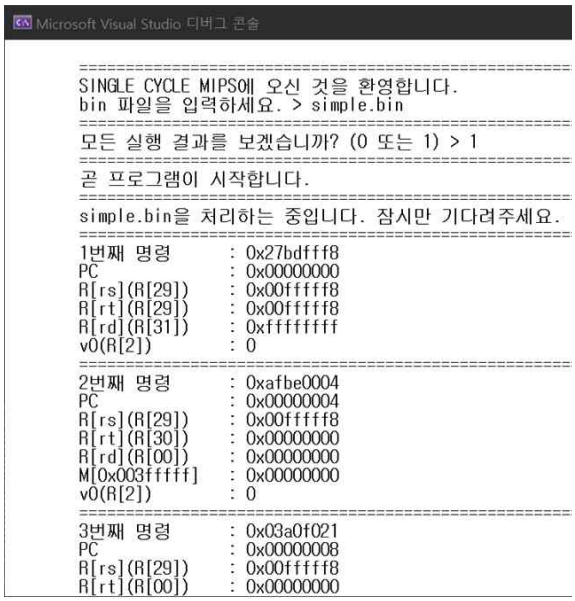
5. 결과

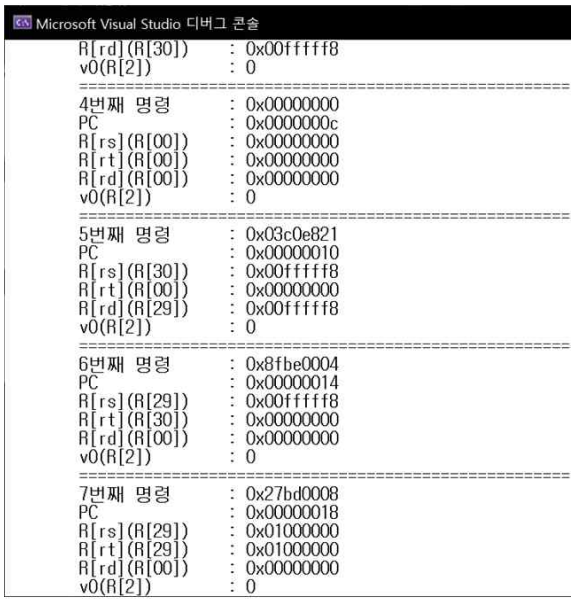
제 5장 결과에서는 강의 자료로 제공한 7가지 테스트 프로그램들을 실행시켜보고 기본 구현과 추가 구현에 해당하는 출력을 확인한다. 기본 구현으로 각 명령어가 실행될 때마다 상태가 변한 레지스터, 메모리 값, 그리고 계산 결과를 출력하고 추가 구현으로는 총 명령어의 수, R-형식 명령어의 수, I-형식 명령어의 수, J-형식 명령어의 수, 메모리 접근 횟수, 분기한 횟수를 출력한다. fib2.bin 파일은 <그림 5.2>와 같이 fib.bin 바이너리 파일의 jal(0x1c: 0c000000)와 nop(0x20: 00000000)을 addi(1c: 20100040) jalr(20: 20200F809)로 수정하여 addi 명령어로 16번 레지스터에 점프할 주소 0x40을 저장하고, jalr 명령어로 점프할 수 있도록 했다. simple3.bin 파일 이후부터는 총 명령어의 수가 기하급수적으로 증가하여 중간 계산 결과를 모두 생략했다.



<그림 5.2> fib.bin의 jal을 jalr로 수정하고 fib2.bin으로 저장했다.

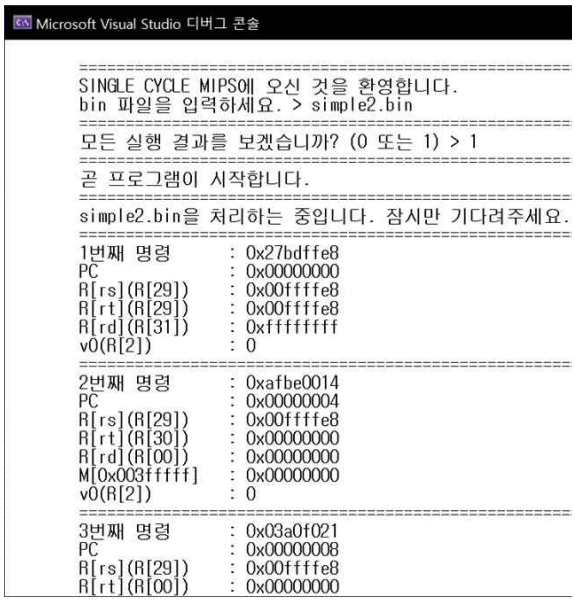
가. Simple.bin

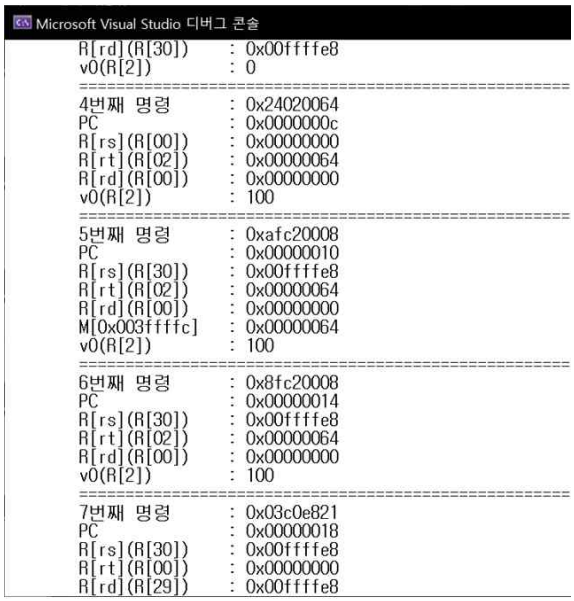

Microsoft Visual Studio 디버그 콘솔
==========
SINGLE CYCLE MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > simple.bin
==========
모든 실행 결과를 보겠습니까? (0 또는 1) > 1
==========
곧 프로그램이 시작합니다.
==========
simple.bin을 처리하는 중입니다. 잠시만 기다려주세요.
==========
1번째 명령 : 0x27bdfff8
PC : 0x00000000
R[rs](R[29]) : 0x00fffff8
R[rt](R[29]) : 0x00fffff8
R[rd](R[31]) : 0xffffffff
v0(R[2]) : 0
==========
2번째 명령 : 0xafbe0004
PC : 0x00000004
R[rs](R[29]) : 0x00fffff8
R[rt](R[30]) : 0x00000000
R[rd](R[00]) : 0x00000000
M[0x003fffff] : 0x00000000
v0(R[2]) : 0
==========
3번째 명령 : 0x03a0f021
PC : 0x00000008
R[rs](R[29]) : 0x00fffff8
R[rt](R[00]) : 0x00000000

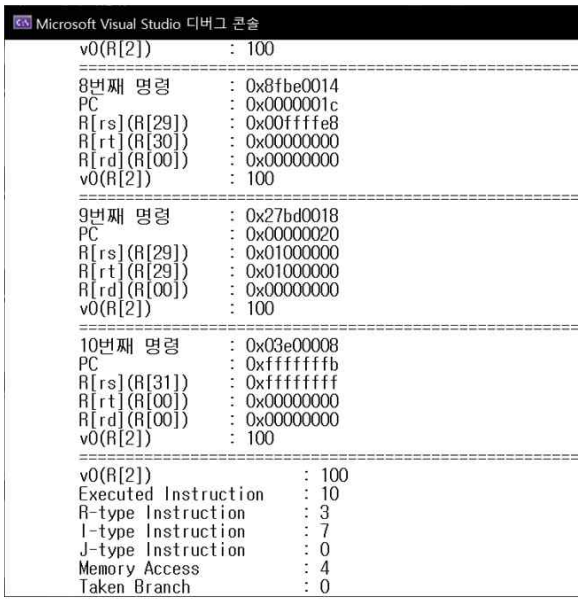

Microsoft Visual Studio 디버그 콘솔
R[rd](R[30]) : 0x00fffff8
v0(R[2]) : 0
==========
4번째 명령 : 0x00000000
PC : 0x0000000c
R[rs](R[00]) : 0x00000000
R[rt](R[00]) : 0x00000000
R[rd](R[00]) : 0x00000000
v0(R[2]) : 0
==========
5번째 명령 : 0x03c0e821
PC : 0x00000010
R[rs](R[30]) : 0x00fffff8
R[rt](R[00]) : 0x00000000
R[rd](R[29]) : 0x00fffff8
v0(R[2]) : 0
==========
6번째 명령 : 0x8fbe0004
PC : 0x00000014
R[rs](R[29]) : 0x00fffff8
R[rt](R[30]) : 0x00000000
R[rd](R[00]) : 0x00000000
v0(R[2]) : 0
==========
7번째 명령 : 0x27bd0008
PC : 0x00000018
R[rs](R[29]) : 0x01000000
R[rt](R[29]) : 0x01000000
R[rd](R[00]) : 0x00000000
v0(R[2]) : 0


Microsoft Visual Studio 디버그 콘솔
v0(R[2]) : 0
==========
v0(R[2]) : 0
Executed Instruction : 8
R-type Instruction : 4
I-type Instruction : 4
J-type Instruction : 0
Memory Access : 2
Taken Branch : 0
프로그램 실행 시간 : 38.417000초
==========
SINGLE CYCLE MIPS를 이용해주셔서 감사합니다.
C:₩Users₩ekrxj₩Desktop₩C Programming₩컴퓨터구조와모바일프로세서₩Project2_Single Cycle MIPS₩SingleCycleMips.Project2.2₩Debug₩SingleCycleMips.Project2.exe(프로세스 14344개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

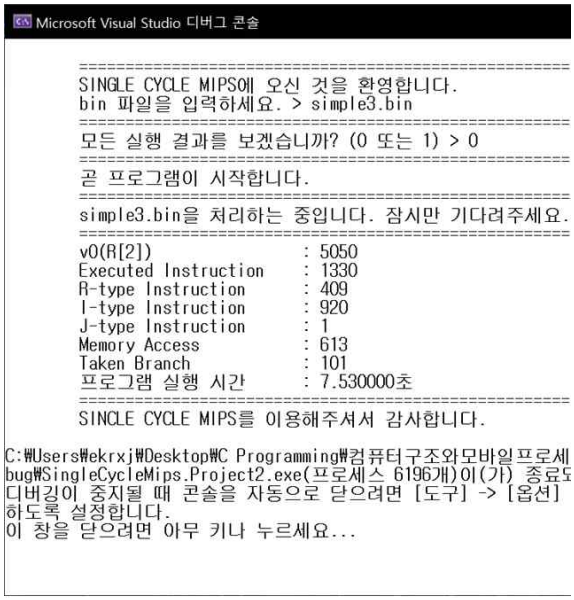
나. Simple2.bin



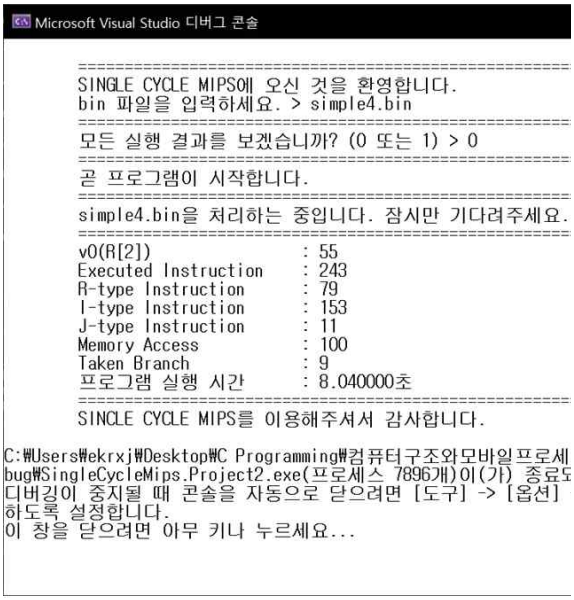




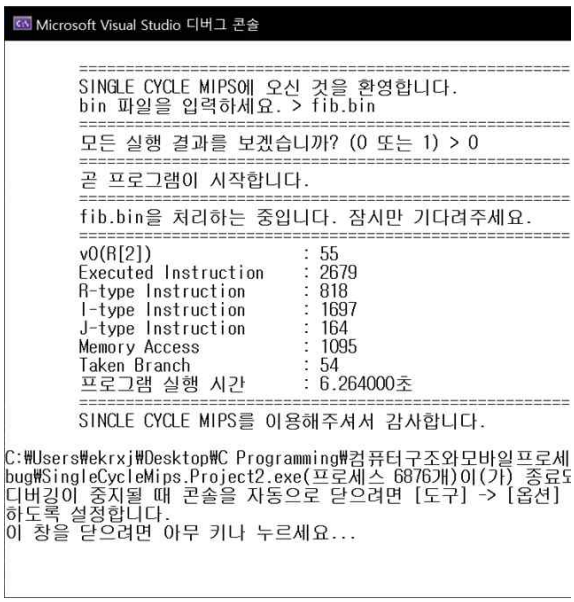
다. Simple3.bin



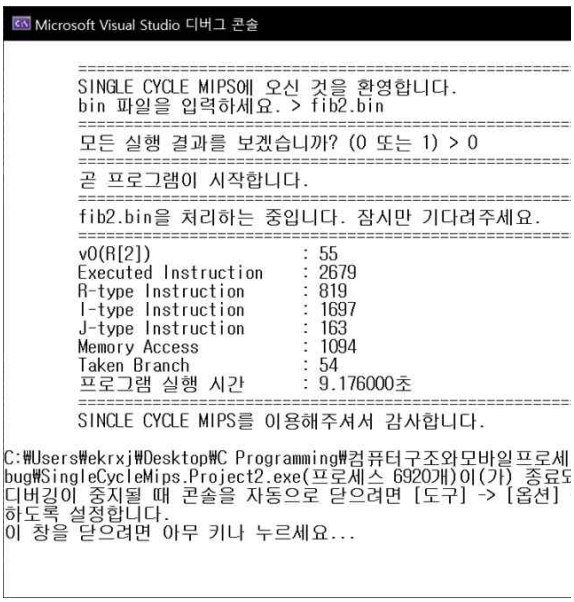
라. Simple4.bin



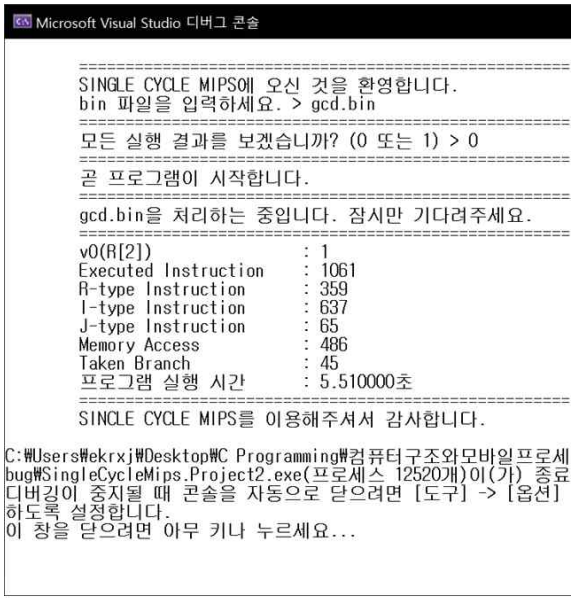
마. fib.bin



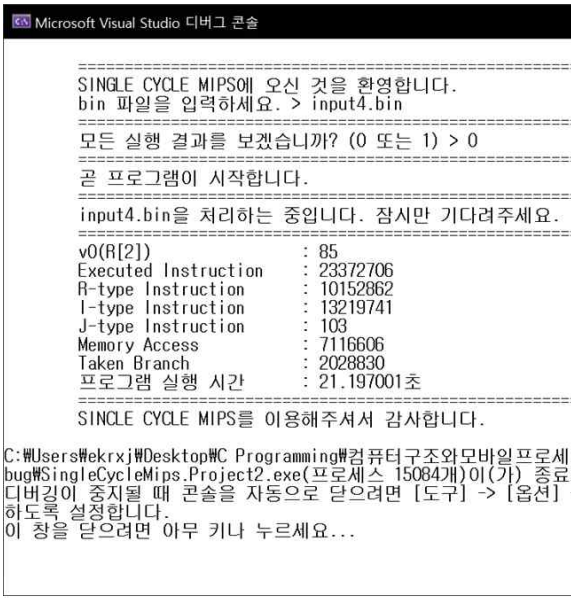
바. fib2.bin



사. gcd.bin



아. input4.bin



6. 고찰 및 느낀 점

우선, 저번 프로젝트1에서 앞으로 프로젝트의 목적을 정확하게 이해하고 이에 맞는 접근을 하는 것이 바람직해 보인다고 언급했었다. 이번 프로젝트2에서는 그때를 교훈삼아 코드부터 바로 작성하지 않고 먼저 MIPS-32 명령어 집합 구조를 이해한 다음, 데이터패스를 그리는 것부터 시작했다. 다음으로 명령어 집합 구조를 기반으로 하는 구조체와 변수를 선언하고, 명령어를 처리하는 5단계를 함수로 표현하면서 프로그램의 틀을 만들어 나갔다. 그려놓은 데이터패스를 비교하면서 함수를 정의하고, 서로 연결하며, 비슷한 과정은 서로 묶어 중복되는 코드를 줄였다. 최종적으로 디버깅 과정까지 모두 거치면서 원활하게 작동하는 마이크로아키텍처 프로그램을 설계할 수 있었다.

프로젝트2를 마무리하며 내가 계획한대로 컴퓨터 구조에 대한 이론을 스스로 학습하고 수업 시간 내용을 숙지하는 기회가 되었다. 개인적으로 학습한 내용을 바탕으로 설계한 프로그램이 좋은 결과를 보여주어서 뿌듯하다. 하지만 여전히 보완해야할 점도 있다. 컴파일에 문제가 없지만 원하는 결과가 나오지 않을 때, 디버깅하는 시간이 너무 길어서 꽤 진땀을 흘렸다. 다음 프로젝트에서는 디버깅 시간을 효과적으로 줄일 수 있는 접근법을 다시 한 번 생각해봐야겠다.

7. 참고 문헌

David A. Patterson, John L. Hennessy.(1994) 컴퓨터구조 및 설계 하드웨어/소프트웨어 인터페이스(고려대학교 박명순, 숭실대학교 김병기·장훈 역). 에드텍