Implementation of Single Cycle MIPS

성영준 Yeong-Jun Seong
32202231(남은 휴일 1일)
단국대학교 모바일시스템공학과

목차

1.1 서론

본 프로그램은 싱글 사이클 프로세서 아키텍처를 기반으로 한 프로그램의 설계 및 구현되었다. 싱글 사이클 프로세서는 모든 명령어를 단일 클록 사이클 동안 실행하는 특성을 가진 컴퓨터 아키텍처로, 각 명령어의 처리 시간이 일관되어 시스템의 예측 가능성과 단순성이 향상된다. 이러한 특성은 프로세서 설계를 단순화시키며, 교육적 환경이나 단순한 제어 시스템에서 특히 유용하게 사용된다.

싱글 사이클 아키텍처의 핵심은 모든 연산이 동일한 클록 사이클에 수행된다는 점이다. 이를 위해, 해당 프로그램에서 프로세서는 명령어를 패치, 디코드, 실행, 메모리 접근, 그리고 결과 쓰기의 일련의 단계로 나누어 처리한다. 각 단계는 별도의 하드웨어 구성 요소에 의해 처리되며, 이들 구성 요소는 동시에 다른 명령어의 다른 단계를 수행할 수 있다.

프로그램의 설계 방식은 이러한 싱글 사이클 아키텍처의 원리를 충실히 따르며, 명령어의 각 단계를 정확하게 구현하기 위해 다양한 하드웨어 시뮬레이션 기법을 사용한다. 예를 들어, ALU(산술 논리 단위)는 산술 및 논리 연산을 담당하며, 분기 및 점프와 같은 제어 명령어는 프로그램 카운터(PC)의 업데이트 메커니즘을 통해 처리된다. 메모리 접근은 데이터와 명령어를 저장하는 메모리 시스템을 통해 이루어지며, 최종 결과는 목적지 레지스터에 쓰여진다.

또한, 제어 유닛은 명령어의 유형에 따라 필요한 제어 신호를 생성하여 각 하드웨어 구성 요소가 적절한 작업을 수행하도록 지시한다. 이 신호들은 멀티플렉서를 통해 구체적인 데이터 경로를 선택하며, 이를 통해 데이터가 올바른 목적지로 전달되거나 적절한 연산이 수행되도록 한다.

이 프로그램의 구현 방식은 특히 시뮬레이션을 통해 실제 하드웨어에 근접한 경험을 제공하기 위해 설계되었다. 이는 학습자가 실제 하드웨어와 유사한 환경에서 프로그램의 흐름을 이해하고, 각 구성 요소의 상호 작용을 실시간으로 관찰할 수 있게 한다.

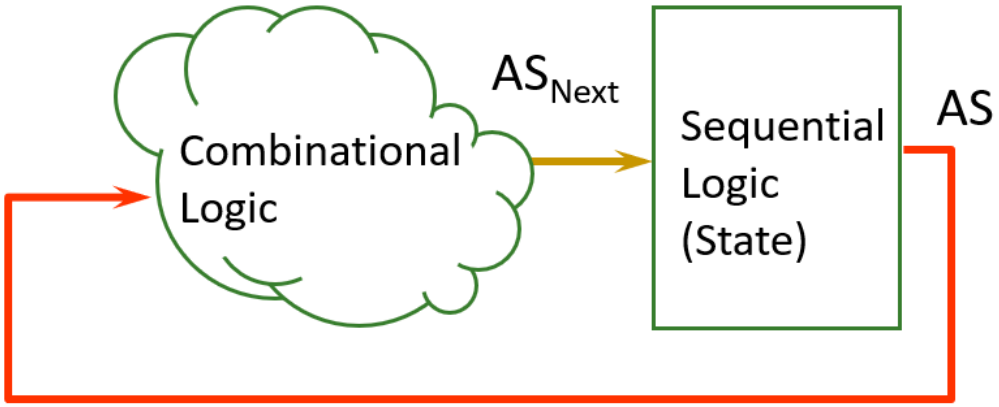
종합적으로, 이 프로그램은 싱글 사이클 프로세서의 설계 및 운용 원리를 따라 개발된 프로그램으로 이 프로그램을 개발하면서 Single cycle에 대한 이해를 심화할 수 있었다.

2.0 Single Cycle에 대해서

본 프로그램은 Single Cycle의 핵심 원칙을 반영하여 설계 되었기에 제작한 프로그램에 대한 설명에 앞서 Single Cycle에 대한 설명을 거쳐 이에 대한 이해를 바탕으로 프로그램에 대한 설명을 이어나갈 예정이다.

2.1 Single cycle

- Single-cycle machine



Single cycle[1]

Single cycle 작동은 한 명령어가 하나의 클록 사이클 내에서 모든 실행 단계를 거치는 과정이다. 이 구조에서는 모든 명령어가 클록 사이클이 끝날 때 완료되며, 모든 상태가 업데이트된다. 이 방식은 각 명령어가 단일 사이클 내에서 실행되도록 보장하며, 제어 장치가 한 번에 전체 명령어에 대한 필요한 신호를 생성해야 한다.

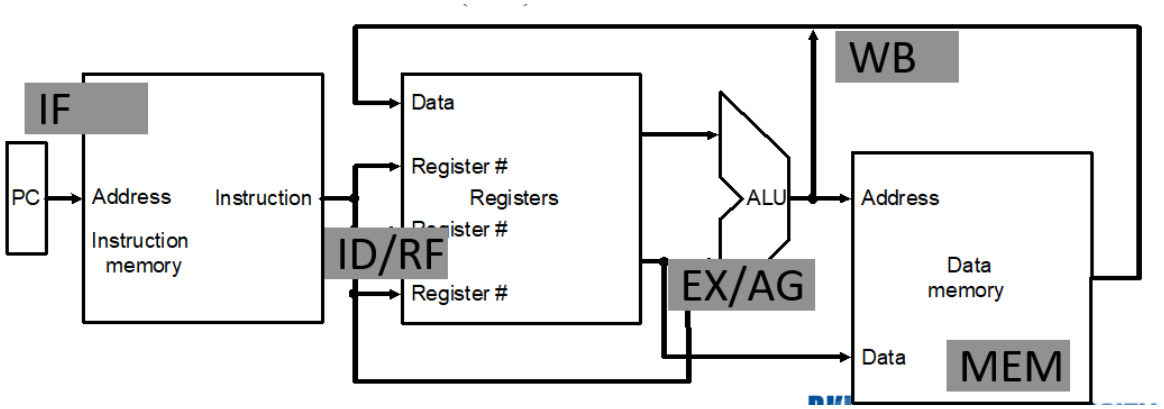
MIPS 아키텍처에서의 single cycle 설정은 모든 명령어가 실행 경로에서 동일한 길이와 간단한 로직을 공유함을 의미하며, 이는 Complex Instruction Set Computing (CISC) 아키텍처에 비해 데이터 경로의 구현을 훨씬 단순화한다. CISC 아키텍처는 다양한 길이와 높은 복잡성을 가진 명령어를 포함하여, 설계를 복잡하게 하고 심지어 간단한 명령어에도 실행 시간이 길어질 수 있다. 왜냐하면 전체 명령어 사이클 시간이 한 클록 사이클 시간을 정의하기 때문이다.

따라서 single cycle 접근 방식은 모든 명령어에 대한 실행 단계를 표준화함으로써 하드웨어 설계를 단순화할 뿐만 아니라 일관된 성능 지표를 보장한다. 명령어 실행에 걸리는 시간이 예측 가능하고 균일하기 때문에, 복잡성과 길이가 다양한 명령어의 실행

시간이 널리 다를 수 있는 CISC 설계보다 효율적인 방법이 된다. 이러한 이유로 single cycle 작동은 프로그램을 예측 가능하고 효율적으로 실행하는 효과적인 방법이라 할 수 있다.

2.2 Data path

Data path에서는 기본적으로 5단계로 구성된다. 이 구조는 각 명령어의 처리를 보다 효율적으로 수행하기 위해 설계되었다. 다음은 데이터 패스에서 각 단계의 기능을 설명한다.



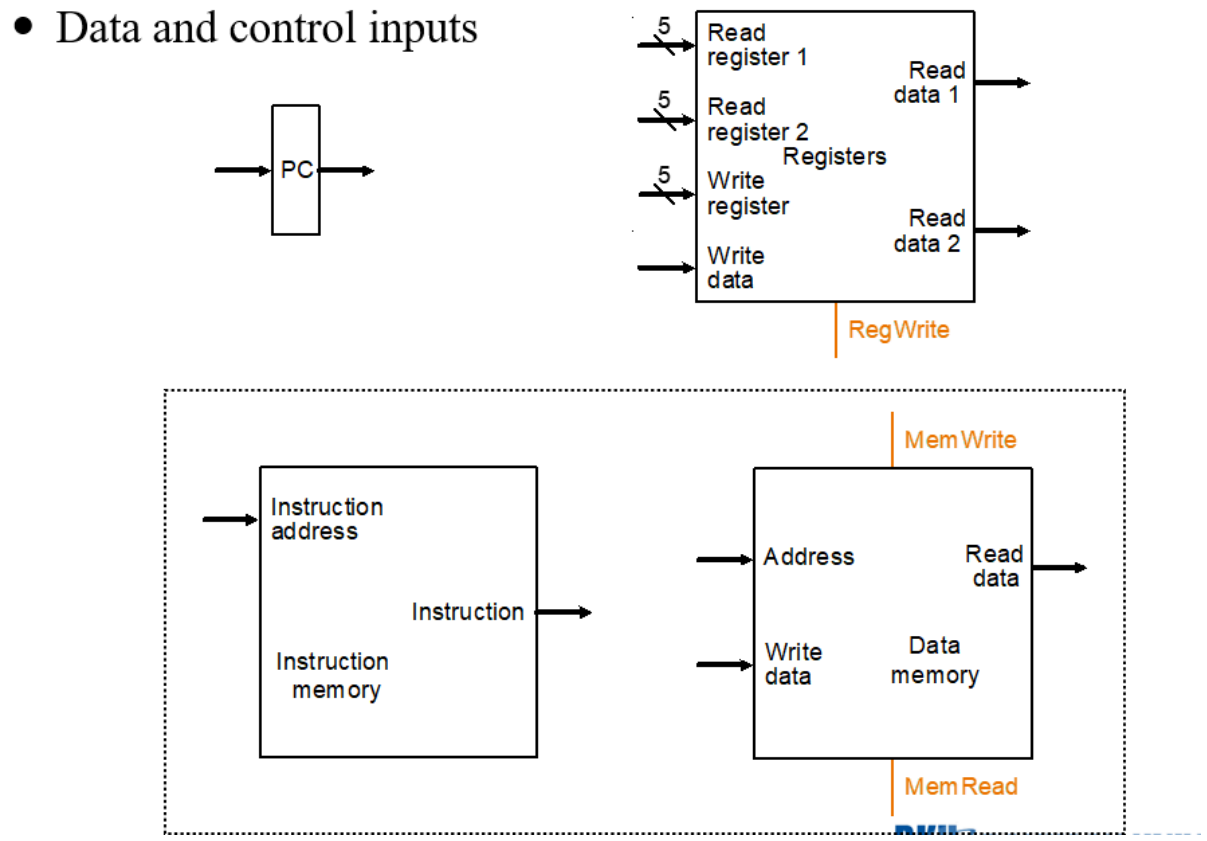
Data path cycle[2]

1. Instruction Fetch (IF): 이 단계에서는 프로그램 카운터(PC)가 가리키는 메모리 주소에서 명령어를 가져온다. 명령어를 가져온 후에는 PC 값을 업데이트하여 다음 명령어의 위치로 이동한다. 이 과정은 전체 사이클의 시작점이다.
2. Instruction Decode and Register Operand Fetch (ID/RF): 가져온 명령어를 디코드하여 어떤 연산을 수행할지 결정한다. 동시에 필요한 레지스터에서 피연산자를 가져오는 작업이 수행된다. 이 단계에서 명령어에 따라 필요한 데이터를 레지스터에서 읽어 다음 단계로 전달한다.
3. Execute/Evaluate Memory Address (EX/AG): 이 단계에서는 앞서 가져온 피연산자를 사용하여 연산을 실행하거나, 메모리 주소를 평가한다. ALU(Arithmetic Logic Unit)가 연산을 처리하고, 메모리 주소 연산의 경우에는 해당 주소를 계산한다.

4. Memory Operand Fetch (MEM): 만약 연산이 메모리 접근을 필요로 한다면, 이 단계에서 메모리에서 데이터를 가져온다. 이 데이터는 다음 단계에서 사용될 수 있으며, 메모리에서 읽기나 메모리로부터의 쓰기가 이뤄질 수 있다.
5. Store/Write Back Result: 연산의 결과를 레지스터나 메모리에 저장한다. 이는 연산 결과를 활용하여 최종 결과를 시스템 내의 적절한 위치에 기록하는 단계이다.

이러한 5단계 구조는 프로세서의 명령어 처리를 최적화하는데 중요한 역할을 한다. 각 단계는 전체 시스템의 성능과 효율성에 기여하며, 복잡한 명령어 집합을 보다 신속하고 정확하게 처리할 수 있도록 설계되었다. 이는 데이터 패스가 프로세서 아키텍처 내에서 핵심적인 부분을 차지한다는 것을 의미한다.

2.3 State element



State element[3]

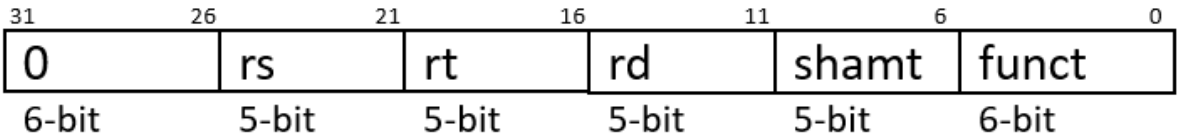
PC(Program Counter), 레지스터, 명령어 메모리, 데이터 메모리, ALU(Arithmetic Logic Unit)는 컴퓨터의 중앙 처리 장치(CPU)에서 핵심적인 역할을 수행하는 구성 요소들이다. 각 구성 요소의 역할과 기능을 보다 자세히 설명하겠다.

1. PC(Program Counter): PC는 현재 처리 중인 명령어의 위치를 가리킨다. PC는 각 명령어가 실행될 때마다 업데이트되며, 대부분의 경우 다음 순차적 명령어의 위치로 자동 증가한다. 그러나 분기(branch) 또는 점프(jump) 명령어와 같은 특정 명령어에 의해 PC 값이 변경될 수 있으며, 이는 프로그램의 실행 흐름을 변경하는 데 중요한 역할을 한다.
2. 레지스터: 레지스터는 CPU 내부의 소규모 데이터 저장 공간으로, 연산에 필요한 데이터를 빠르게 제공하고 결과를 임시로 저장한다. 레지스터의 값은 명령어의 종류에 따라 변할 수 있으며, 이는 제어 신호(control signal)에 의해 결정된다. 레지스터는 명령어 실행의 효율성을 높이기 위해 빠른 접근이 가능하도록 설계되었다.
3. 명령어 메모리(Instruction Memory): 이 메모리는 프로그램에 의해 실행될 명령어들이 저장되어 있는 곳이다. PC값을 사용하여 명령어 메모리에서 현재 실행할 명령어를 가져오며, 이 명령어는 디코드 과정을 거쳐 각 구성 요소에 필요한 작업을 지시한다.
4. 데이터 메모리(Data Memory): 명령어의 실행에 따라 계산된 결과를 저장하거나 필요한 데이터를 로드하는 메모리 부분이다. 데이터 메모리는 ALU 연산의 결과를 저장하거나, 필요한 데이터를 반환할 때 사용된다. 제어 신호에 따라 읽기 또는 쓰기 작업이 수행된다.
5. ALU(Arithmetic Logic Unit): ALU는 다양한 산술 및 논리 연산을 수행하는 CPU의 핵심 부품이다. 연산의 종류는 실행 중인 명령어에 따라 다르며, 이를 통해 데이터를 처리하고 결과를 생성한다. ALU는 연산 결과를 직접 레지스터에 저장하거나 데이터 메모리로 전송할 수 있다.

이러한 구성 요소들은 서로 긴밀하게 연결되어 하나의 명령어가 실행될 때마다 효과적으로 협력하여 처리 속도와 효율을 극대화한다. 컴퓨터의 성능은 이러한 부품들의 설계와 상호 작용에 크게 의존하며, 각 부품의 최적화는 전체 시스템의 효율성을 결정하는 중요한 요소이다.

2.4 R-type

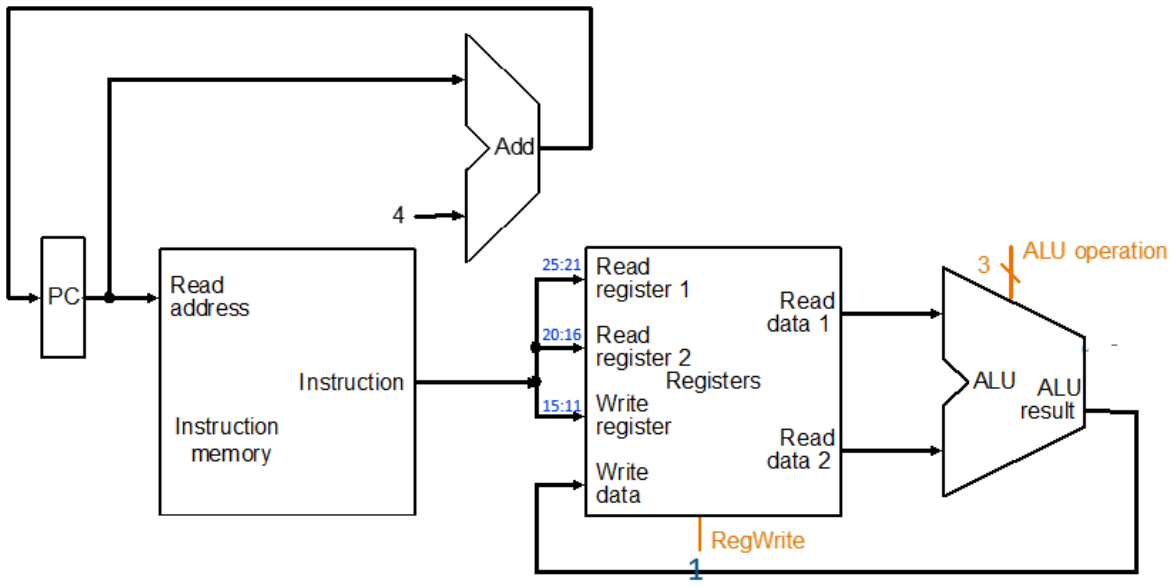
R-type 명령어는 MIPS 아키텍처에서 사용되는 명령어 유형 중 하나로, 주로 레지스터 간의 산술 및 논리 연산을 수행하는 데 사용된다. 이 명령어는 총 32비트로 구성되어 있으며, 각 비트는 특정 기능을 수행하는 데 필요한 정보를 담고 있다. 다음은 R-type 명령어의 구조를 자세히 설명한다.



R-type 명령어의 구조[4]

1. Opcode (6 bits): R-type 명령어의 경우, Opcode는 항상 '000000' (0x0)으로 설정되어 있다. 이는 R-type 명령어가 funct 필드를 사용하여 구체적인 연산 유형을 결정한다는 것을 의미한다.
2. rs (source register, 5 bits): rs 필드는 연산에 사용될 첫 번째 소스 레지스터의 주소를 나타낸다.
3. rt (source register, 5 bits): rt 필드는 연산에 사용될 두 번째 소스 레지스터의 주소를 나타낸다.
4. rd (destination register, 5 bits): 연산 결과가 저장될 대상 레지스터의 주소를 지정한다.
5. shamt (shift amount, 5 bits): 비트 시프트 연산에 사용되는 양을 지정한다. 이 필드는 쉬프트 명령어에서 중요한 역할을 하며, R-type 명령어에서 이 필드를 사용하여 데이터의 비트 위치를 조정할 수 있다.
6. funct (function code, 6 bits): funct 필드는 Opcode가 동일한 R-type 명령어들을 구별하는 데 사용된다. 이 필드는 특정 ALU 연산을 지정하며, 다양한 산술 및 논리 연산을 정의하는 데 사용된다.

R-type 명령어의 데이터 패스에서는 두 가지 중요한 제어 신호가 활용된다.



R-type Data path[5]

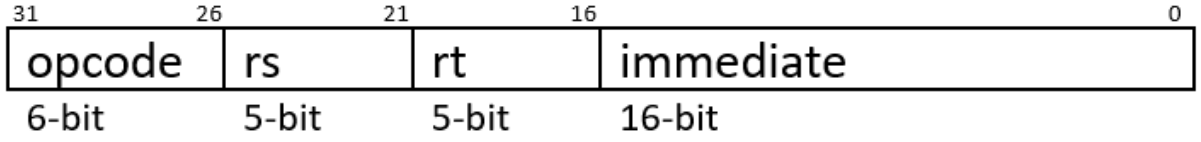
- RegWrite: 이 신호는 명령어 실행 후 연산 결과를 대상 레지스터에 저장할지 여부를 결정한다. ALU에서 계산된 결과는 이 신호가 활성화되었을 때만 레지스터에 쓰여진다.
- ALU operation signal: 이 신호는 funct 필드의 값에 따라 ALU가 수행할 특정 연산을 결정한다. 예를 들어, 덧셈, 뺄셈, 논리곱, 논리합 등 다양한 연산이 이 신호를 통해 지정된다.

R-type 명령어는 MIPS 프로세서에서 중요한 역할을 하며, 효율적인 레지스터 기반 연산을 가능하게 한다. 이 명령어 유형의 설계는 고성능 컴퓨팅 작업에서 중요한 요소로 작용하며, 복잡한 연산을 빠르고 정확하게 수행할 수 있도록 도와준다. 이러한 명령어 구조와 데이터 패스는 프로세서의 성능 최적화에 결정적인 역할을 하며, 이를 통해 고급 프로그래밍 작업과 시스템의 전반적인 효율성이 향상된다.

2.5 I-type

I-type 명령어는 MIPS 아키텍처에서 사용되는 또 다른 중요한 명령어 유형으로, 상수값이나 주소 오프셋이 필요한 연산에 주로 사용된다. R-type 명령어와 구별되는 가장 중요한 특징은 이 명령어 구조 내에 즉시 값(immediate value)이 포함된다는 점이다. I-type 명령어는 메모리 접근, 조건부 분기, 레지스터 사이의 간단한 산술 연산 등에 사용된다. 이러한 명령어는 특히 하드웨어 설계에서 데이터 경로(data path)와 제어 로직을 이해하는 데 중요하다.

I-type 명령어의 구조는 다음과 같이 구성된다:

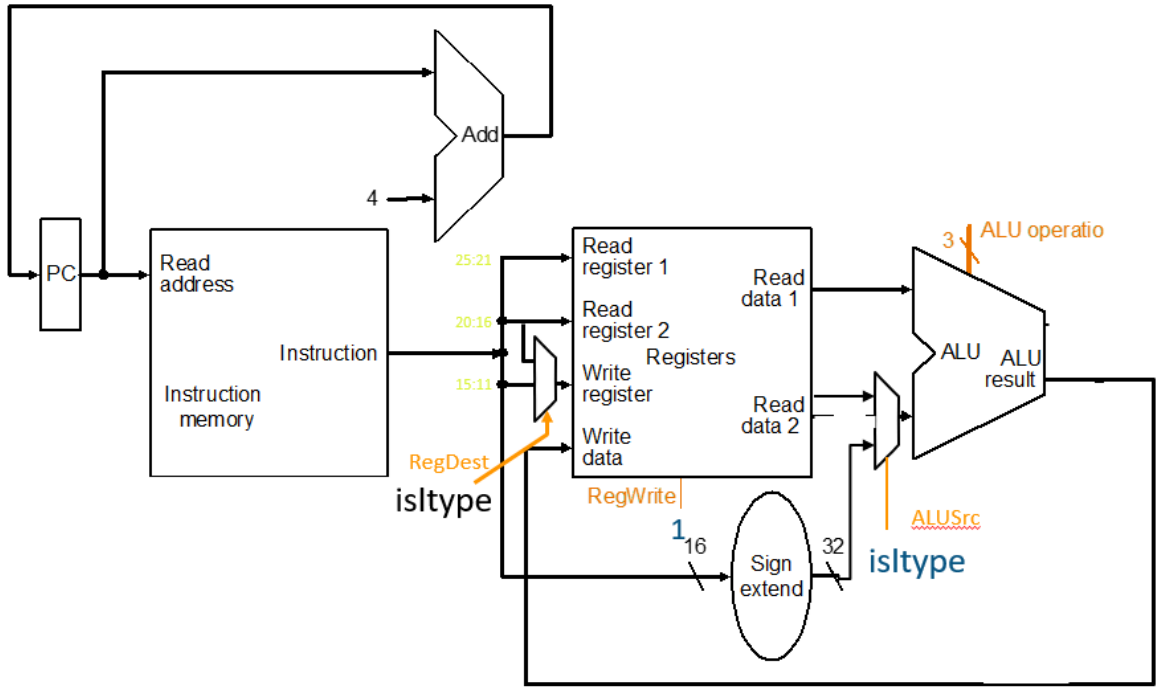


I-type 명령어 구조[6]

1. Opcode (6 bits): 명령어의 유형과 수행할 연산의 범주를 정의한다.
2. rs (source register, 5 bits): 첫 번째 소스 레지스터를 지정하며, 데이터를 제공한다.
3. rt (target register, 5 bits): 연산 결과를 저장할 레지스터를 지정한다.
4. Immediate (16 bits): 연산에 사용될 즉시 값으로, 이 값은 명령어의 실행 동안 32비트로 확장되어 사용된다.

I-type 명령어의 즉시 값은 16비트로 제한되어 있기 때문에, 32비트 시스템에서 이를 사용하기 위해 sign extension이 필수적이다. sign extension 과정에서는 즉시 값의 최상위 비트(sign bit)를 기준으로 나머지 비트를 채워 넣어 전체 32비트로 확장한다. 이 과정은 음수를 적절히 처리하기 위해 중요하며, sign bit가 1인 경우 추가 비트들을 1로, 0인 경우 0으로 채운다.

I-type 명령어의 실행에는 여러 제어 신호가 필요하며, 특히 중요한 신호는 RegDest와 ALUSrc이다:



I-type Data path[7]

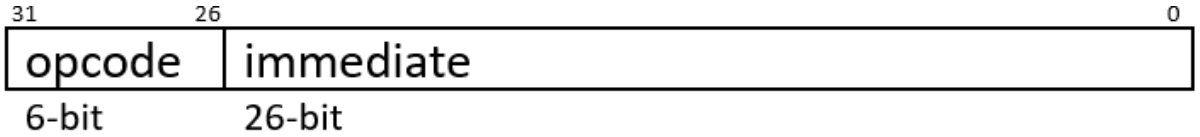
- RegDest: 이 신호는 결과 데이터가 저장될 레지스터를 결정한다. I-type 명령어에서는 일반적으로 rt 레지스터가 대상이 된다.
- ALUSrc: 이 신호는 ALU의 한 입력이 레지스터의 값(rs)이 아닌 확장된 즉시 값(immediate)을 사용하도록 설정한다. 이를 통해 레지스터와 즉시 값 사이의 연산이 가능해진다.

또한, 명령어에 따라 다양한 데이터를 처리하기 위해 멀티플렉서(Mux)가 사용된다. 멀티플렉서는 제어 신호에 따라 입력 중 하나를 선택하여 출력으로 전달하는 기능을 한다. 이는 다양한 데이터 소스 사이에서 유연하게 전환할 수 있도록 하며, 프로세서의 데이터 경로 복잡성을 관리하는 데 필수적이다.

2.6 J-type

J-type 명령어는 MIPS 아키텍처에서 제어 흐름을 직접적으로 관리하는 데 사용되는 명령어 유형으로, 주로 점프(jump) 연산에 활용된다. 이 명령어 유형은 프로그램 카운터(PC)를 크게 변경하여 실행 흐름을 새로운 위치로 이동시키는 데 중점을 둔다. J-type 명령어의 구조는 상대적으로 간단하며, 주로 opcode와 주소 필드로 구성된다.

J-type 명령어는 다음과 같이 구성된다:



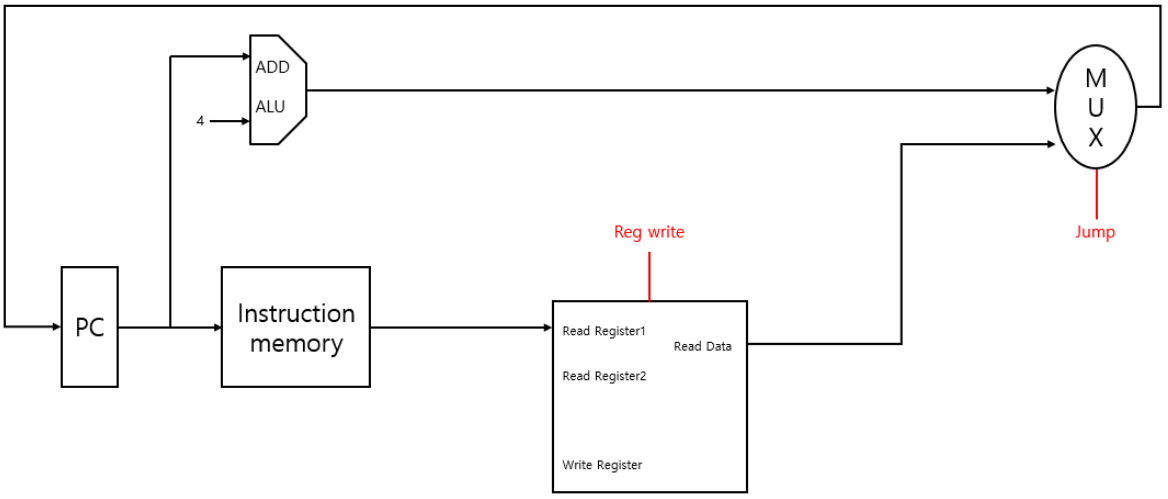
J-type 명령어 구조 [8]

1. Opcode (6 bits): 명령어의 유형을 결정하며, J-type에서는 특정 점프 명령어를 식별하는 데 사용된다.
2. Address (26 bits): 점프할 목적지 주소를 지정한다. 이 주소는 명령어 내에서 직접 제공되며, 실행될 새로운 코드 위치를 나타낸다.

26비트 주소는 32비트 시스템에서 사용하기 위해 확장되어야 한다. 이 과정에서 몇 가지 중요한 단계가 포함된다:

- 비트 연산: 주어진 26비트 주소는 먼저 2만큼 비트 연산(Shift Left by 2)을 수행하여, 4비트 단위로 맞추어진 PC값과 일치시킨다. 이 연산 후 주소는 28비트 길이가 된다.
- 상위 비트 추가: 28비트로 확장된 주소는 여전히 32비트 길이에 미치지 못한다. 따라서 현재 PC의 최상위 4비트를 취하여 이를 주소의 상위 비트로 추가함으로써 최종적으로 32비트 주소를 형성한다.

이러한 주소 확장은 점프 명령어가 목표 위치로 정확하게 점프할 수 있도록 보장하는 데 필수적이다.



- PCSrc 신호: J-type 명령어의 실행 여부를 결정하는 제어 신호로, J-type 명령어일 경우 PCSrc1은 1로 설정되어 PC를 새로운 점프 주소로 업데이트한다. 그렇지 않은 경우에는 0으로 설정되어 기본 실행 흐름을 유지한다.

큰 주소 공간을 관리하기 위해 때때로 레지스터에 주소를 저장하여 사용할 수 있다. 이는 특히 대규모 프로그램 또는 복잡한 제어 구조에서 유용하다. 레지스터를 사용하여 주소를 저장하고 관리함으로써, 프로세서는 더 유연하게 메모리 공간을 활용할 수 있으며, 점프 및 분기 명령어 실행의 정확성과 효율성을 향상시킨다.

J-type 명령어는 프로그램의 실행 흐름을 효과적으로 제어하고, 사용자가 지정한 코드 위치로 빠르게 점프할 수 있도록 설계되었다. 주소의 적절한 확장과 조정, 그리고 정밀한 제어 신호 활용은 이 명령어 유형이 효과적으로 기능하도록 하는 핵심 요소들이다.

2.7 control-signal

	When De-asserted	When asserted	Equation
RegDest	GPR write select according to rt, i.e., inst[20:16]	GPR write select according to rd, i.e., inst[15:11]	opcode==0
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) && (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	steer memory load to GPR wr. port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	opcode==LW
MemWrite	Memory write disabled	Memory write enabled	opcode==SW
$PCSrc_1$	According to $PCSrc_2$	next PC is based on 26-bit immediate jump target	(opcode==J) || (opcode==JAL)
$PCSrc_2$	next PC = PC + 4	next PC is based on 16-bit immediate branch target	(opcode==Bxx) && "bcond is satisfied"

Control signal [9]

MIPS CPU의 데이터 패스에서 제어 신호들은 데이터의 이동 및 명령어의 실행을 안내하는 핵심적인 역할을 한다. 제어 신호들은 제어 유닛에 의해 생성되며, 메모리에서 가져온 명령어를 디코드하여 명령어를 실행하기 위해 필요한 일련의 조치들을 결정한다.

MIPS 아키텍처에서, 각 명령어는 제어 유닛으로 하여금 데이터 패스 내에서 데이터를 어떻게 이동시킬지, 어떤 연산을 수행할지를 결정하는 제어 신호를 설정하게 만든다. 예를 들어, RegDst 신호는 레지스터 목적지를 선택하며, 명령어의 유형에 따라 레지스터를 rt 또는 rd로 결정한다. ALUSrc 신호는 ALU 연산의 두 번째 입력이 레지스터의 데이터인지, 아니면 즉시 값을 사용하는지를 결정한다.

제어 신호들은 각 명령어가 데이터 패스로 전달될 때 초기화되어야 한다. 이 초기화 과정은 opcode나 funct와 같은 명령어 구분을 통해 이루어진다. 이러한 구분에 따라 각 명령어에 맞는 제어 신호가 설정되어, 데이터가 저장될지, 메모리에서 읽어질지 등을 결정한다.

MemRead와 MemWrite 신호들은 메모리 접근이 필요한 명령어에서 중요한 역할을 한다. MemRead는 메모리에서 데이터를 읽을 때 활성화되고, MemWrite는 메모리에 데이터를 쓸 때 활성화된다. PCSrc 신호는 분기 명령어에서 PC 값을 변경할지 여부를 결정한다. 이 신호는 조건부 분기 명령어의 결과에 따라 PC를 다음 명령어 주소로 업데이트하거나 분기 목적지 주소로 변경한다.

제어 신호들은 프로세서가 정확하게 명령어를 수행할 수 있도록 하는 데 중요하다. 이 신호들은 프로그램의 흐름을 제어하고, 데이터의 올바른 이동 경로를 보장하며, 명령어 실행 과정에서 중요한 결정을 내리는 데 사용된다. 이러한 과정은 프로세서의 효율성과 성능에 직결되며, 프로세서 설계의 최적화를 가능하게 한다.

3.0 Program

이전의 설명한 Single cycle의 핵심 원칙들을 반영하여 프로그램이 설계되었다. 이를 Data path의 각 단계에 따라 설명하도록 하겠다.

3.1 Single cycle 실행 전

싱글 사이클 CPU의 프로그램 실행 전 준비 단계를 설명한다. 싱글 사이클 CPU 아키텍처는 모든 명령어가 단 하나의 클록 사이클 내에서 실행되는 특성을 지닌다. 이 특성 때문에, 프로그램을 메모리에 로드하고 모든 하드웨어 구성요소를 적절히 초기화하는 과정이 매우 중요하다.

loda_instructions_into_memory 함수는 프로그램을 메모리에 로드하는 과정을 담당한다. 함수는 먼저 filename 변수를 사용하여 로드할 바이너리 파일의 경로를 지정한다. 이후, 파일을 바이너리 읽기 모드("rb")로 열어 파일 포인터 bin_file을 얻는다. 파일이 정상적으로 열리지 않을 경우, 오류 메시지를 출력하고 프로그램을 종료한다. 파일이 성공적으로 열리면, fseek 함수를 사용하여 파일의 끝으로 이동하고 ftell 함수로 파일의 크기를 확인한다. 확인된 파일 크기만큼 instruction_memory에 메모리를 할당하고, fread 함수를 이용해 파일의 내용을 instruction_memory에 읽어들인다. 마지마으로

파일을 닫고 함수를 종료한다. 이 과정은 싱글 사이클 CPU가 실행할 명령어들을 물리 메모리에 올리는 필수적인 단계이다.

initialize_processor 함수는 프로세서와 관련된 모든 구성 요소를 초기화하는 작업을 수행한다. 먼저, 32개의 레지스터를 위한 메모리 공간을 할당받고, 할당에 실패할 경우 오류 메시지를 출력하고 프로그램을 종료한다. 데이터 메모리와 명령어 유형을 기록할 instruction_types 배열에 대해서도 동일한 메모리 할당 절차를 진행한다. 각 메모리 공간이 성공적으로 할당된 후, 모든 레지스터와 데이터 메모리는 0으로 초기화된다. 특히, ra_reg 레지스터는 함수 호출의 리턴 주소를 저장하는 레지스터로, 프로그램 종료를 위해 0xffffffff로 초기화된다. 스택 포인터(sp_reg) 또한 스택의 시작 주소인 0x10000000으로 초기화된다. 프로그램 카운터(PC)와 명령어 레지스터(instruction_register)는 0으로 설정된다.

이러한 초기화 과정은 프로세서가 오류 없이 정확하게 프로그램을 실행할 수 있는 기반을 마련한다. 레지스터와 메모리의 초기화는 잠재적인 오류를 방지하고, 프로세서의 상태를 명확하게 정의하는 데 필수적이다. 각 구성 요소의 초기화는 싱글 사이클 CPU가 시작될 때 반드시 필요한 절차로, 이를 통해 프로세서는 안정적으로 첫 명령어부터 실행을 시작할 준비가 완료된다.

3.2 fetch

```
void fetch(void) {
    instruction_register = 0;
    for (int i = 0; i < 4; i++) {
        instruction_register = (instruction_register << 8) | instruction_memory[PC + i];
    }
    PC += 4;
    printf("[Fetch Instruction] -> (0x%08x)\n", instruction_register);
}
```

이 코드 부분은 싱글 사이클 CPU 아키텍처에서 명령어를 가져오는(fetch) 동작을 수행하는 것이다. 싱글 사이클 아키텍처는 모든 명령어가 단일 클록 사이클에 수행된다는 특징을 가진다. 따라서, 명령어를 가져오는 동작도 클록 사이클의 한 부분으로 진행된다.

fetch 함수는 instruction_register를 초기화한 후, 4바이트 길이의 명령어를 메모리에서 읽어온다. 이 과정에서 프로그램 카운터(PC)가 가리키는 위치에서 시작해 4바이트를 읽어들이는데, 각 바이트를 8비트씩 왼쪽으로 시프트하면서 instruction_register에

합치는 연산을 반복한다. 각 바이트는 메모리 주소 PC + i에서 읽혀진다. 이는 instruction_memory 배열의 인덱스를 통해 접근된다.

명령어를 완성한 후, 프로그램 카운터는 다음 명령어 위치로 업데이트된다. 이는 PC += 4라는 코드에 의해 수행되며, 이는 MIPS 아키텍처의 명령어가 모두 32비트(4바이트)임을 반영한다. 프로그램 카운터의 업데이트는 다음 명령어를 가져오기 위한 준비 과정이다.

마지막으로, printf 함수를 사용하여 현재 가져온 명령어의 값을 16진수 형태로 출력한다. 이 출력은 프로세서가 현재 어떤 명령어를 처리하고 있는지를 시각적으로 확인할 수 있는 수단을 제공한다. 출력된 명령어 값은 디버깅과 검증 과정에서 유용하게 사용된다.

이러한 fetch 함수의 동작은 명령어 사이클의 첫 단계를 구현하는 핵심적인 부분이며, 프로세서가 정확한 실행을 위해 올바른 명령어를 순서대로 가져오는 것을 보장한다. 이 과정을 통해 프로세서는 주어진 프로그램 코드를 순차적으로 해석하고 실행할 수 있는 기반을 마련한다.

3.3 Decode

```c
void decode(Instrucion* inst, ControlUnit* cu, ALUOutput* output) {
    parse_opcode(inst);
    generate_control_signals(inst, cu);
    if (instruction_register == 0x00000000) {
        printf("[Decode Instruction] -> nop\n");
        return;
    }
    output->read_data1 = registers[inst->rs];
    output->immediate_extension = extend_immediate_logical(inst->immediate_value);
    output->immediate_extension = extend_immediate_arithmetic(output->immediate_extension);
    output->read_data2 = registers[inst->rt];
    output->alu_src_mux_output = Mux_ALUsrc(cu->alu_src, output->immediate_extension, output->read_data2);
```

싱글 사이클 CPU 아키텍처 내에서 명령어를 해석하는 단계인 디코드(Decode) 단계를 자세히 설명한다. 싱글 사이클 아키텍처의 디코드 단계는 명령어를 해석하고, 실행에 필요한 제어 신호를 생성하는 과정이다. 이는 CPU가 다음에 어떤 동작을 수행할지 결정하는 중요한 단계이다.

decode 함수는 우선 parse_opcode 함수를 호출하여 명령어 레지스터에서 오퍼코드, 소스 레지스터, 대상 레지스터, 즉시 값 등을 추출한다. 이렇게 추출된 정보는 Instrucion 구조체 인스턴스에 저장된다. 이후, generate_control_signals 함수를 통해 명령어의 유형(R-type, I-type, J-type)에 따라 CPU의 제어 유닛에 알맞은 제어 신호를 설정한다. 이 신호들은 메모리 접근, ALU 작업 선택, 레지스터 쓰기 활성화 등을 제어한다.

명령어 레지스터가 0x00000000, 즉 NOP(No Operation) 명령일 경우, "[Decode Instruction] -> nop"라는 메시지를 출력하고 함수를 반환한다. NOP 명령은 아무런 연산도 수행하지 않는다.

명령어가 NOP이 아닐 경우, Instrucion 구조체에서 rs와 rt 레지스터의 인덱스를 사용하여 registers 배열에서 해당 레지스터의 값을 읽어 output 구조체의 read_data1과 read_data2에 저장한다. 이렇게 읽은 데이터는 후속 ALU 연산에 사용된다. 또한, 즉시 값은 논리 연산 및 산술 연산 확장 함수를 거쳐 적절히 확장된다.

다음으로, Mux_ALUsrc 함수를 호출하여 ALU의 두 번째 입력을 결정한다. 이 함수는 제어 유닛의 ALU 소스 신호에 따라 즉시 값이 확장된 결과 또는 read_data2 중 하나를 선택한다. 이 값은 ALU 연산에서 두 번째 입력으로 사용된다.

명령어 유형에 따라 다음과 같이 처리된다:

- R-type 명령어의 경우, num_r_type_executions 카운터를 증가시키고, 연관된 레지스터와 함수 코드 정보를 출력한다.
- I-type 명령어의 경우, num_i_type_executions 카운터를 증가시키고, 연관된 레지스터와 즉시 값 정보를 출력한다.
- J-type 명령어의 경우, num_j_type_executions 카운터를 증가시키고, 점프 주소를 출력한다.

이 과정은 각 명령어의 디코딩이 성공적으로 완료되었음을 보여준다. 디코드 단계는 프로세서가 수행할 명령어를 정확히 이해하고, 실행 단계로 넘어가기 위한 준비를 하는 필수적인 과정이다. 명령어의 성공적인 해석과 제어 신호의 생성은 CPU가 다음 단계인 실행(Execute) 단계에서 올바른 연산을 수행할 수 있도록 보장한다.

3.4 Execute

```
void execute(Instrucion* inst, ControlUnit* cu, ALUOutput* output) {
    alu(determine_alu_operation(inst, cu), output, output->read_data1, output->alu_src_mux_output);
}
```

Execute 단계에서는 싱글 사이클 CPU 아키텍처에서의 실행(Execute) 단계를 설명하고, 특히 ALU(산술 논리 장치)의 역할을 중점적으로 다룬다. 실행 단계는 CPU가 디코드

단계에서 해석한 명령어를 실제로 수행하는 과정이다. 이 단계에서 ALU는 핵심적인 역할을 하며, 다양한 산술 및 논리 연산을 수행한다.

execute 함수는 ALU 연산을 실행하는 주요 기능을 담당한다. 함수는 먼저 determine_alu_operation을 호출하여 현재 명령어에 맞는 ALU 연산을 결정한다. 이 함수는 명령어의 유형(R-type, I-type 등)과 특정 연산 코드(예: 덧셈, 뺄셈, AND, OR 등)를 분석하여 적절한 ALU 연산 유형을 반환한다. ALU 연산의 결정은 명령어의 구조와 프로그램 로직에 근거한다.

ALU에 입력되는 데이터는 output->read_data1과 output->alu_src_mux_output 두 가지이다. output->read_data1은 주로 명령어에서 지정한 첫 번째 레지스터의 데이터를 나타내며, output->alu_src_mux_output은 두 번째 데이터 소스로, 명령어에 따라 레지스터 또는 확장된 즉시 값(immediate value)이 될 수 있다. 이 데이터는 decode 단계에서 준비되며, Mux_ALUsrc 함수를 통해 ALU 연산의 두 번째 입력으로 선택된다. 이 함수는 제어 신호에 따라 레지스터 데이터 또는 즉시 값 중 하나를 선택하여 ALU에 제공한다.

alu 함수는 실제 연산을 수행한다. 이 함수는 결정된 연산 유형에 따라 입력된 두 데이터 값을 사용하여 연산을 실행하고, 그 결과를 output->alu_result에 저장한다. ALU는 덧셈, 뺄셈, 비트 AND, OR, 시프트 연산 등을 포함한 다양한 연산을 수행할 수 있다. 예를 들어, 덧셈 연산의 경우, 두 입력 값을 더하고 결과를 저장한다. 연산의 결과는 종종 후속 단계인 메모리 접근이나 결과 쓰기 단계에서 사용된다.

실행 단계의 ALU 연산은 싱글 사이클 CPU에서 중요한 기능을 한다. 이 단계는 모든 연산이 단 하나의 클록 사이클 내에서 완료되도록 보장함으로써, CPU의 효율성과 반응성을 높인다. 또한, 실행 단계는 프로그램의 동작을 결정하는 핵심적인 단계로, ALU의 정확한 연산은 프로그램의 성능과 안정성에 직접적인 영향을 미친다. 이로써, 싱글 사이클 CPU는 각 명령어를 정확하게 실행하고 예측 가능한 성능을 제공한다.

3.5 Memory Access

```c
void access_memory(Instrucion* inst, ControlUnit* cu, ALUOutput* output) {
    uint32_t address = output->alu_result;
    if (cu->mem_write_enable) {
        for (int i = 0; i < 4; i++) {
            data_memory[address + (3 - i)] = (output->read_data2 >> (8 * i)) & 0xff;
        }
        printf("[Store] : Mem[0x%08x] <- r[%d] = 0x%08x\n", address, inst->rt, output->read_data2);
    } else if (cu->mem_read_enable) {
        output->memory_data_read = 0;
        for (int i = 0; i < 4; i++) {
            output->memory_data_read = (output->memory_data_read << 8) + data_memory[address + i];
        }
        printf("[Load] : r[%d] <- Mem[0x%08x] = 0x%08x\n", inst->rt, address, output->memory_data_read);
    }
}
```

싱글 사이클 CPU 아키텍처의 메모리 접근(Memory Access) 과정을 자세히 설명하며, 이는 CPU에서 실행된 결과를 메모리에 저장하거나 메모리에서 데이터를 읽어오는 단계이다. 이 단계는 프로세서의 실행 결과에 따라 메모리의 상태를 업데이트하거나 필요한 데이터를 메모리로부터 가져오는 역할을 한다.

access_memory 함수는 ALU의 결과(output->alu_result)를 사용하여 메모리 주소를 결정하고, 이 주소를 기반으로 데이터의 읽기 또는 쓰기를 수행한다. 이 주소는 이전의 실행 단계에서 ALU에 의해 계산된 결과이며, 특히 로드(load)와 스토어(store) 명령어에서 중요한 역할을 한다.

함수의 실행은 먼저 제어 유닛(ControlUnit)의 mem_write_enable 신호를 검사하여 메모리 쓰기 동작이 활성화되어 있는지 확인한다. 메모리 쓰기가 활성화된 경우, output->read_data2에서 읽은 데이터를 메모리의 해당 주소에 저장한다. 데이터는 4바이트 단위로 처리되며, 각 바이트는 주소에 역순으로 저장된다. 이는 리틀 엔디언 방식을 따르는 시스템에서 일반적으로 사용되는 방법이다. 저장 작업이 완료되면, 어떤 레지스터의 데이터가 어떤 메모리 주소에 저장되었는지를 나타내는 로그 메시지를 출력한다.

만약 메모리 읽기가 활성화되어 있다면, output->memory_data_read 변수를 사용하여 지정된 메모리 주소로부터 4바이트의 데이터를 읽어온다. 데이터는 메모리로부터 순차적으로 읽혀지고, 이 데이터는 대상 레지스터(inst->rt)에 저장될 준비가 된다. 메모리로부터 읽은 데이터에 대한 정보도 출력되어, 어떤 메모리 주소로부터 어떤 데이터가 읽혔는지를 확인할 수 있다.

이 메모리 접근 과정은 싱글 사이클 프로세서의 효율성을 높이는 데 중요한 역할을 한다. 메모리 읽기와 쓰기 작업을 통해 프로세서는 실행 결과를 외부 세계와 동기화할 수 있고, 필요한 데이터를 신속하게 로드하여 다음 연산에 사용할 수 있다. 이 과정은 프로세서의 성능과 반응성에 직접적인 영향을 미치며, 프로그램의 실행 흐름을 정확하게 관리한다.

따라서, 메모리 접근 단계는 데이터의 저장과 검색을 관리함으로써 전체 시스템의 데이터 일관성과 정확성을 보장한다. 이는 싱글 사이클 CPU가 각 클록 사이클에서 하나의 명령어를 처리할 수 있도록 지원하는 핵심적인 기능이다.

3.6 Write Back

```
void write_back(Instrucion* inst, ControlUnit* cu, ALUOutput* output) {
    uint32_t reg_write_data = Mux_MemToReg(cu->mem_to_reg_select, output->memory_data_read, output->alu_result);
    if (cu->reg_write_enable) {
        uint8_t destination_register = Mux_RegDest(cu->dest_reg_sel, ra_reg, inst->rd, inst->rt);
        reg_write_data = Mux_JumpAndLink(cu->jump_link, PC + 4, reg_write_data);
        registers[destination_register] = reg_write_data;
        if (cu->jump_link) {
            printf("[Write Back] : r[%d] <- 0x%08x = 0x%08x + 8\n", ra_reg, PC + 4, PC - 4);
        } else {
            printf("[Write Back] : r[%d] <- 0x%08x\n", destination_register, reg_write_data);
        }
    }
}
```

싱글 사이클 CPU 아키텍처의 마지막 단계인 Write Back 과정을 상세히 설명한다. Write Back 단계는 실행된 명령어의 결과를 레지스터에 저장하는 과정으로, 전체 명령어 사이클을 완료하는 결정적인 단계이다. 이 단계는 명령어에 따라 계산된 결과나 메모리에서 읽은 데이터를 시스템 레지스터에 쓰기 때문에 프로그램의 상태 업데이트에 핵심적인 역할을 한다.

write_back 함수는 여러 가지 중요한 작업을 수행한다. 먼저, Mux_MemToReg 함수를 호출하여 메모리 읽기 작업의 결과(output->memory_data_read)와 ALU 연산의 결과(output->alu_result) 중 하나를 선택한다. 이 선택은 제어 유닛의 mem_to_reg_select 신호에 의해 결정된다. 선택된 데이터는 최종적으로 레지스터에 쓰일 값(reg_write_data)이 된다.

레지스터 쓰기가 활성화되어 있을 경우(cu->reg_write_enable), Mux_RegDest 함수를 통해 데이터가 저장될 레지스터를 결정한다. 이 함수는 명령어 유형에 따라 목적

레지스터(inst->rd) 혹은 대상 레지스터(inst->rt)를 선택하며, jump_link 신호가 있는 경우에는 ra_reg을 사용한다. 결정된 레지스터는 destination_register 변수에 저장된다.

추가로, Mux_JumpAndLink 함수는 점프 및 링크 명령어(jump_link)의 처리를 담당한다. 점프 및 링크 명령어가 활성화되면, 명령어 카운터(PC)에 4를 더한 값을 reg_write_data로 설정하고, 그 값을 ra_reg 레지스터에 저장한다. 이는 함수 호출 시 호출한 위치로 돌아갈 수 있도록 돕는 기능이다.

마지막으로, 결정된 destination_register에 최종 데이터(reg_write_data)를 저장하고, 이 과정을 통해 프로세서의 상태가 업데이트된다. 레지스터에 데이터를 쓰는 동작은 명령어 실행 결과를 시스템에 반영하는 중요한 단계이며, 필요에 따라 이 데이터가 다음 연산의 입력으로 사용될 수 있다.

Write Back 단계의 완료는 프로세서가 단일 클록 사이클 내에서 명령어를 성공적으로 수행했음을 의미한다. 이 단계는 프로그램의 실행 결과를 안정적으로 레지스터에 저장하여, 다음 명령어의 실행을 위한 준비가 완료되었음을 보장한다. 결과적으로, Write Back 단계는 프로세서의 빠르고 효율적인 작동을 지원하며, 프로그램의 정확한 실행을 가능하게 한다.

4.0 Program의 핵심 요소

본 프로그램에는 Single Cycle의 5단계를 거칠 뿐만 아니라 다른 다양한 요소들 또한 구현되어 있다. 이러한 요소들은 더욱 해당 프로그램이 유연하게 작동될 수 있도록 다양한 기능을 제공한다.

4.1 PC

```
void update_processor_state(Instrucion* inst, ControlUnit* cu, ALUOutput* output) {
    uint32_t print_PC = PC;
    uint32_t jump_addr = (inst->jump_address << 2) | (PC & 0xf0000000);
    uint32_t branch_target = (output->immediate_extension << 2) + PC;
    PC = Mux_Jump(cu->jump_control, registers[inst->rs], jump_addr, Mux_Branch(cu->branch_control & output->zero_flag, branch_target, PC));
```

```
if (cu->jump_control) {
    if (inst->function_code == funct_jr) {
        printf("[PCs Update] : (Jump) PC = 0x%08x <- rs : R[%d] = 0x%08x\n", PC, inst->rs, registers[inst->rs]);
    } else {
        printf("[PC Update] : (Jump) PC <- 0x%08x = (0x%08x << 2) | ((0x%08x+4) & 0xf0000000)\n", PC, inst->jump_address, print_PC - 4);
    }
} else if (cu->branch_control && output->zero_flag) {
    branch_taken_count++;
    printf("[PC Update] : (Branch Taken) PC <- 0x%08x = (0x%08x << 2) + 0x%08x (= 0x%08x+4)\n", PC, output->immediate_extension, print_PC, print_PC - 4);
} else {
    printf("[PC Update] : PC <- 0x%08x = 0x%08x+4\n", PC, print_PC);
}
cycle_count++;
```

싱글 사이클 CPU 아키텍처에서 프로그램 카운터(PC) 업데이트 과정을 자세히 설명한다. 프로그램 카운터 업데이트는 프로세서가 다음에 실행할 명령어의 메모리 주소를 결정하는 중요한 단계이다. 이 단계는 명령어의 유형과 조건에 따라 다음 실행할 명령어의 위치를 조정함으로써 프로그램의 흐름을 제어한다.

update_processor_state 함수는 프로그램 카운터를 조정하는 여러 경우를 처리한다. 함수는 먼저 현재의 프로그램 카운터 값을 print_PC에 저장한다. 이 값은 후에 디버깅 및 로깅 목적으로 사용된다.

함수는 두 가지 주요 유형의 주소 계산을 수행한다: 점프 주소(jump_addr)와 분기 목표(branch_target). 점프 주소는 명령어에서 제공된 점프 대상 주소(inst->jump_address)를 기반으로 계산되며, 주소의 하위 비트들을 왼쪽으로 두 비트 시프트하고 현재 PC의 상위 비트와 결합한다. 이는 점프 명령어가 전체 메모리 주소 공간 내에서 정확한 위치로 점프할 수 있도록 한다. 분기 목표는 즉시 값(output->immediate_extension)을 사용하여 계산되며, 이 값은 왼쪽으로 두 비트 시프트된 후 현재 PC에 더해져 분기할 목표 주소를 생성한다.

다음으로, Mux_Jump 및 Mux_Branch 함수를 통해 최종적인 PC 값이 결정된다. Mux_Jump 함수는 점프 명령어의 유형에 따라 적절한 주소를 선택하고, Mux_Branch 함수는 분기 명령어가 활성화되었을 때 조건에 따라 분기할지 아니면 다음 명령어로 진행할지를 결정한다.

만약 점프 제어 신호(cu->jump_control)가 활성화되면, 명령어의 유형에 따라 다음과 같은 로깅이 수행된다:

- 만약 함수로 점프(funct_jr) 명령어가 실행된 경우, 사용된 레지스터와 그 값이 로그에 기록된다.

- 다른 유형의 점프 명령어가 실행된 경우, 계산된 점프 주소와 이 주소가 어떻게 계산되었는지 로그에 기록된다.

분기 명령어(cu->branch_control)가 활성화되고 조건(output->zero_flag)이 참인 경우, 분기가 취해지며, 분기 횟수(branch_taken_count)가 증가하고, 분기가 발생한 주소와 계산 과정이 로그에 기록된다.

이 모든 로그는 프로그램의 흐름과 상태를 추적하는 데 도움을 준다. 함수의 마지막에는 사이클 수(cycle_count)가 증가하여, 명령어 처리의 각 단계가 몇 번 실행되었는지 추적한다.

프로그램 카운터의 업데이트 과정은 싱글 사이클 CPU에서 프로그램의 정확한 실행을 보장하며, 다음 명령어의 선택이 조건과 명령어 유형에 따라 정확하게 이루어 지게 된다.

4.2 ALU

```c
void alu(char alu_ops, ALUOutput* output, uint32_t alu_input1, uint32_t alu_input2) {
    switch (alu_ops) {
        case 0:  // J type instruction (No ALU operation)
            break;
        case 1:  // funct_add, opcode_addi, opcode_addiu, opcode_lw, opcode_sw
            output->alu_result = alu_input1 + alu_input2;
            break;
        case 2:  // and, opcode_andi
            output->alu_result = alu_input1 & alu_input2;
            break;
        case 4:  // funct_nor
            output->alu_result = ~(alu_input1 | alu_input2);
            break;
        case 5:  // or, opcode_ori
            output->alu_result = alu_input1 | alu_input2;
            break;
        case 6:  // funct_slt, opcode_slti, opcode_sltiu
            output->alu_result = alu_input1 < alu_input2;
            break;
        case 7:  // funct_sll
            output->alu_result = alu_input2 << alu_input1;
            break;
```

```c
        case 8:  // funct_srl
            output->alu_result = alu_input2 >> alu_input1;
            break;
        case 9:  // funct_sub, funct_subu
            output->alu_result = alu_input1 - alu_input2;
            break;
        case 10: // opcode_beq
            output->zero_flag = (alu_input1 - alu_input2 == 0);
            break;
        case 11: // opcode_bne
            output->zero_flag = (alu_input1 - alu_input2 != 0);
            break;
        case 12: // opcode_lui
            output->alu_result = alu_input2 << 16;
            break;
        default:
            printf("Error, Matching operation not found\n");
            exit(1);
    }
    return;
}
```

ALU는 CPU의 핵심 구성 요소 중 하나로, 다양한 산술 및 논리 연산을 수행하여 프로세서의 명령어를 실행하는 데 중요한 역할을 한다. 이 alu 함수는 싱글 사이클 CPU에서 이러한 연산들을 구현하며, 입력된 연산 유형(alu_ops)에 따라 적절한 연산을 실행한다.

함수는 두 개의 주요 입력, alu_input1과 alu_input2, 그리고 연산 유형을 나타내는 alu_ops를 받아들인다. 이 입력들은 명령어에 따라 레지스터에서 읽히거나 즉시 값으로부터 제공된다. 연산의 결과는 ALUOutput 구조체의 alu_result 필드에 저장되며, 특정 연산에서는 zero_flag 필드를 설정하여 결과가 0인지 아닌지를 나타낸다.

다음은 각 연산 유형별로 수행되는 작업을 설명한다:

1. 연산 유형 0 (J 타입 명령어): ALU 연산을 수행하지 않는다. 이 경우는 주로 프로그램 흐름 제어에 사용된다.
2. 연산 유형 1 (덧셈 연산): alu_input1과 alu_input2를 더한다. 이 연산은 덧셈, 주소 계산, 메모리 로드 및 스토어 명령어에 사용된다.
3. 연산 유형 2 (AND 연산): 두 입력의 비트 단위 AND 연산을 수행한다. 이는 비트 마스킹과 플래그 설정에 사용된다.
4. 연산 유형 4 (NOR 연산): 두 입력의 NOR 연산을 수행하여 결과의 모든 비트를 반전한다. 이는 논리 연산에서 사용된다.
5. 연산 유형 5 (OR 연산): 두 입력의 비트 단위 OR 연산을 수행한다. 이는 비트 설정 명령어에 활용된다.
6. 연산 유형 6 (Set Less Than 연산): alu_input1이 alu_input2보다 작은 경우 1을, 그렇지 않은 경우 0을 반환한다. 이는 비교 연산에 사용된다.
7. 연산 유형 7 (Shift Left Logical 연산): alu_input2를 alu_input1만큼 왼쪽으로 시프트한다. 이는 값의 크기 조절에 사용된다.
8. 연산 유형 8 (Shift Right Logical 연산): alu_input2를 alu_input1만큼 오른쪽으로 시프트한다. 이는 부호 없는 숫자의 크기를 조절할 때 사용된다.
9. 연산 유형 9 (뺄셈 연산): alu_input1에서 alu_input2를 빼서 결과를 계산한다. 이는 뺄셈 및 차이 계산에 사용된다.
10. 연산 유형 10 (Branch Equal 연산): 두 입력이 동일할 경우 zero_flag를 설정한다. 이는 분기 명령어의 조건 판단에 사용된다.
11. 연산 유형 11 (Branch Not Equal 연산): 두 입력이 다를 경우 zero_flag를 설정한다. 이는 분기 조건 판단에 사용된다.
12. 연산 유형 12 (Load Upper Immediate 연산): alu_input2를 16비트 왼쪽으로 시프트하여 상위 16비트를 로드한다. 이는 즉시 값 로딩에 사용된다.

이러한 다양한 ALU 연산들은 싱글 사이클 CPU에서 명령어를 효과적으로 실행하고, 프로그램의 로직에 따라 적절한 데이터 처리를 가능하게 한다. 각 연산은 명령어의 요구 사항에 따라 정확하게 수행되며, 이는 프로세서의 성능과 반응성을 최적화한다.

4.3 MUX

```c
uint8_t Mux_RegDest(unsigned int dest_reg_sel, uint8_t return_addr, uint8_t rd, uint8_t rt) {
    switch (dest_reg_sel) {
        case 2: return return_addr;
        case 1: return rd;
        default: return rt;
    }
}

uint32_t Mux_ALUsrc(unsigned int alu_src, uint32_t immediate_extension, uint32_t rd2) {
    instruction_types[1] = alu_src;  // Document that we are using an I-type instruction
    return alu_src ? immediate_extension : rd2;
}

uint32_t Mux_Branch(unsigned int pc_src, uint32_t target, uint32_t add4) {
    return pc_src ? target : add4;
}

uint32_t Mux_Jump(unsigned int jump_control, uint32_t reg_rs, uint32_t jump_addr, uint32_t mux_branch) {
    switch (jump_control) {
        case 2: return reg_rs;
        case 1: return jump_addr;
        case 0: return mux_branch;
        default:
            printf("Error, Jump control signal is invalid\n");
            exit(1);
    }
}
```

```c
uint32_t Mux_MemToReg(unsigned int mem_to_reg_select, uint32_t read_data, uint32_t alu_res) {
    return mem_to_reg_select ? read_data : alu_res;
}

uint32_t Mux_Zero_Extend(unsigned int zero_extension, uint32_t zero_extended, uint32_t sign_extended) {
    return zero_extension ? zero_extended : sign_extended;
}

uint32_t Mux_Shift(unsigned int shift_control, uint8_t shift_amount, uint32_t reg_rs) {
    return shift_control ? (shift_amount << reg_rs) : reg_rs;
}

uint32_t Mux_JumpAndLink(unsigned int jump_link, uint32_t ra_PC, uint32_t reg_write_data) {
    return jump_link ? ra_PC : reg_write_data;
}
```

Single cycle CPU에서 MUX는 명령어에 따라 다른 데이터 경로를 선택하는 중요한 기능을 한다. 이는 프로세서가 상황에 따라 다른 입력 소스를 선택할 수 있게 하여 유연성과 효율성을 증가시킨다.

1. Mux_RegDest 함수

Mux_RegDest 함수는 명령어에 따라 결과 데이터가 저장될 레지스터를 선택한다. dest_reg_sel 값에 따라 다음과 같은 선택이 이루어진다:

- 2: 반환 주소 레지스터(return_addr)를 선택한다. 이는 주로 점프와 링크 명령어에서 사용된다.
- 1: 목적 레지스터(rd)를 선택한다. 이는 R-type 명령어에서 일반적이다.
- 0: 대상 레지스터(rt)를 선택한다. I-type 명령어에서 흔히 볼 수 있다.

2. Mux_ALUsrc 함수

Mux_ALUsrc 함수는 ALU의 두 번째 입력을 결정한다. alu_src 신호에 따라 즉시 값(immediate_extension) 또는 두 번째 레지스터 값(rd2) 중 하나를 선택한다. 이는 명령어가 즉시 값을 요구하는지 또는 레지스터 간 연산을 수행하는지에 따라 달라진다.

3. Mux_Branch 함수

Mux_Branch 함수는 분기 명령어의 목적지 주소를 결정한다. pc_src 신호가 활성화되어 있으면 분기 목표 주소(target)를 선택하고, 그렇지 않으면 다음 명령어 주소(add4)를 선택한다.

4. Mux_Jump 함수

Mux_Jump 함수는 점프 명령어의 최종 목적지 주소를 결정한다. jump_control 신호에 따라 레지스터 값(reg_rs), 점프 주소(jump_addr), 또는 분기 결과(mux_branch) 중 하나를 선택한다.

5. Mux_MemToReg 함수

Mux_MemToReg 함수는 메모리 읽기 작업의 결과 또는 ALU 연산 결과 중 어떤 것을 레지스터에 쓸지 결정한다. mem_to_reg_select 신호가 활성화되면 메모리에서 읽은 데이터(read_data)를, 그렇지 않으면 ALU 결과(alu_res)를 선택한다.

6. Mux_Zero_Extend 함수

Mux_Zero_Extend 함수는 즉시 값의 확장 방법을 결정한다. zero_extension 신호가 설정되면 제로 확장된 값(zero_extended)을, 그렇지 않으면 부호 확장된 값(sign_extended)을 선택한다.

7. Mux_Shift 함수

Mux_Shift 함수는 시프트 연산에서 시프트 양을 결정한다. shift_control 신호가 활성화되면 지정된 시프트 양(shift_amount)에 따라 레지스터 값(reg_rs)을 시프트한다.

8. Mux_JumpAndLink 함수

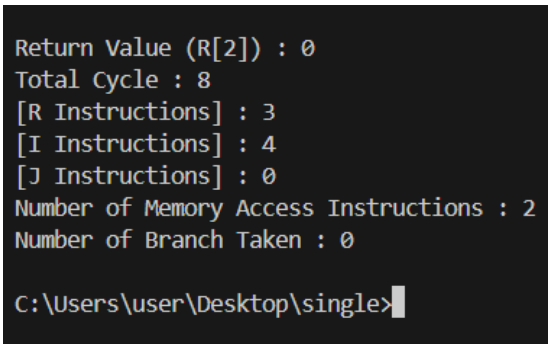
Mux_JumpAndLink 함수는 점프 및 링크 명령어에서 반환 주소를 선택한다. jump_link 신호가 활성화되면 현재 PC 값(ra_PC)을 선택하고, 그렇지 않으면 일반적인 레지스터 쓰기 데이터(reg_write_data)를 사용한다.

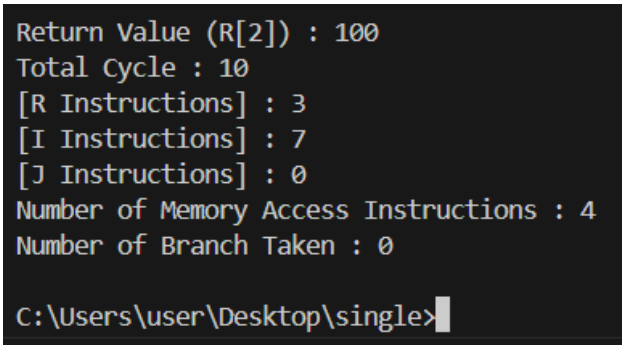
이러한 MUX들은 싱글 사이클 CPU에서 매우 중요한 역할을 한다. 각각의 선택 메커니즘은 명령어의 요구에 따라 최적의 데이터 경로를 제공하며, 이는 프로세서가 효율적으로 명령어를 처리하도록 보장한다. MUX를 통한 유연한 데이터 경로 관리는 CPU의 성능과 반응성을 크게 향상시킨다.

5.1 실행결과

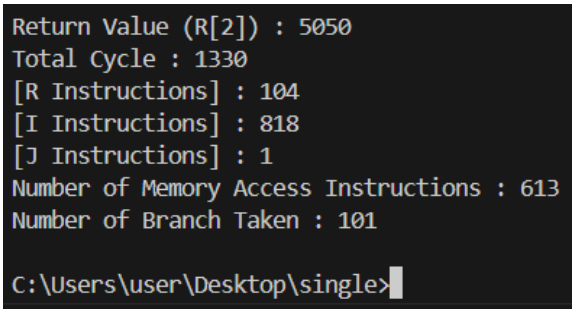
```
const char* filename = "C:\\Users\\user\\Desktop\\Single cycle\\simple4.bin";
```

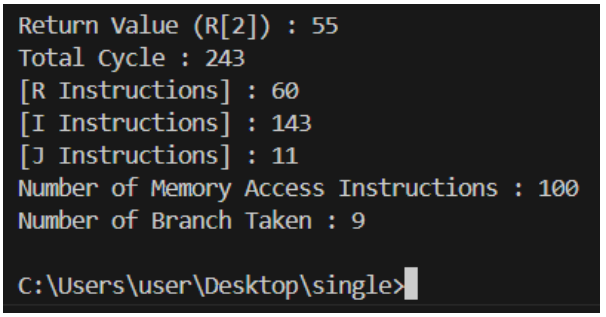
다음에 입력받을 바이너리 파일을 받은 후 실행하면





Simple, Simple2 실행결과

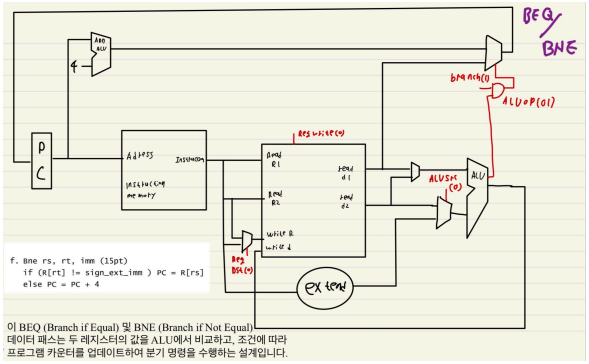


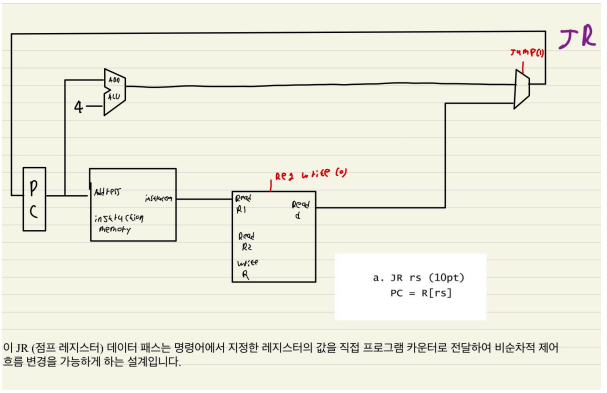


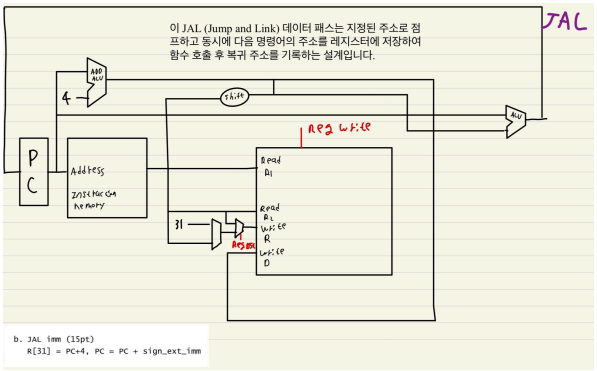
Simple3, Simple4 실행결과

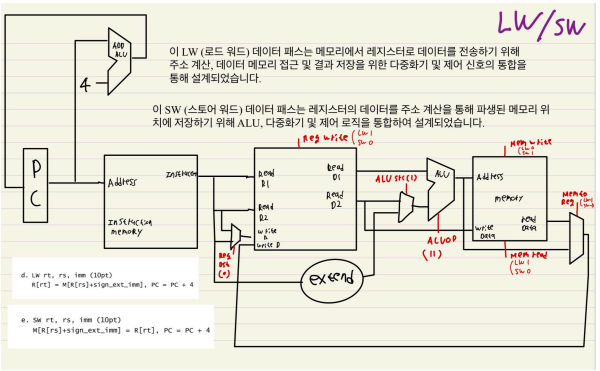
다음과 같이 실행이 되는 것을 확인할 수 있다.

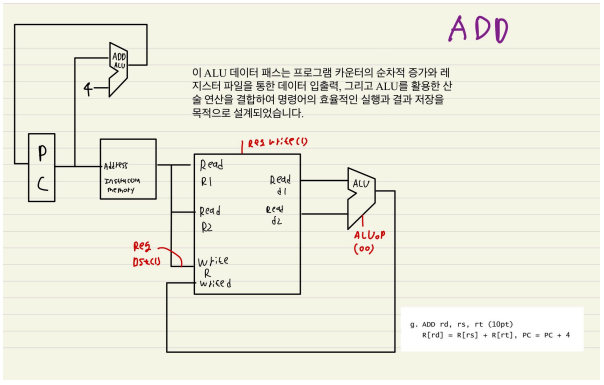
6.1 결론

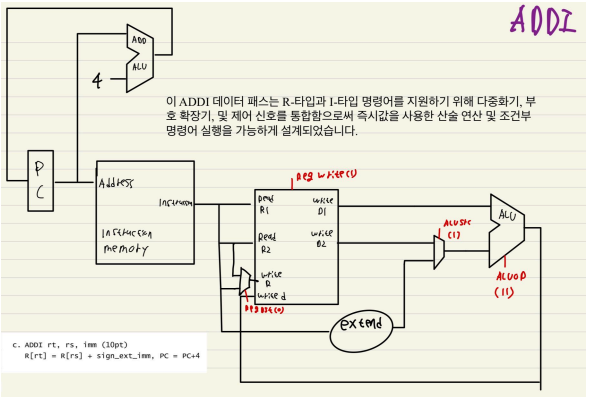












이 프로그램의 개발은 SIngle cycle CPU 아키텍처의 설계와 구현을 통해 컴퓨터 구조에 대한 심도 깊은 이해를 얻는 중요한 경험이었다. 이 프로그램을 구현하기 위해 Data path를 여러번 그려보며 이를 통한 이해를 바탕으로 프로세서의 각 구성 요소와 그들 간의 상호작용을 이해하는 것부터 시작했다. 명령어의 Fetch, Decode, Execute, Memory Access, 그리고 Write Back까지, Single cycle의 각 단계가 어떻게 서로 연결되고 각각의 역할을 수행하는지 학습하는 과정은 매우 흥미로웠다.

프로그램의 핵심인 ALU, 제어 유닛, 레지스터 파일 등을 설계하면서, 각 구성 요소가 명령어를 어떻게 처리하는지에 대한 실질적인 이해를 높일 수 있었다. 특히 ALU 설계는 다양한 산술 및 논리 연산을 어떻게 구현하는지를 배울 수 있는 기회였다. 또한, 제어 신호를 생성하는 로직을 작성하고 테스트하면서, 실제 하드웨어에서 이러한 신호들이 어떻게 다양한 명령어의 실행을 제어하는지 실감할 수 있었다.

메모리 관리 부분도 중요한 학습 과정이었다. 프로그램 카운터의 업데이트와 관련된 로직을 구현하면서, 프로그램의 흐름을 제어하는 데 있어 PC가 얼마나 중요한지 이해했다. 점프와 분기 명령어를 처리하는 방법을 통해, 복잡한 프로그램에서도 CPU가 어떻게 정확한 실행 경로를 유지하는지를 배울 수 있었다.

프로젝트를 진행하며 겪은 도전 중 하나는 디버깅과 오류 수정이었다. 초기 설계에서 예상치 못한 문제들이 발생할 때마다, 문제의 원인을 찾고 해결책을 모색하는 과정이 필수적이었다. 이 과정에서 시스템적 사고와 문제 해결 능력이 크게 향상되었다고 느낀다.

종합적으로, 이 프로젝트는 단순히 기술적 지식을 넓히는 것을 넘어서, 실제 하드웨어 시스템의 작동 원리를 체험하고, 효과적인 팀워크와 문제 해결 전략을 경험하는 기회였다. 이 경험은 나의 전문성을 한층 더 발전시키는 데 큰 도움이 되었으며, 향후 다양한 기술적인 도전을 하는것에 자신감을 심어주었다.

[참고 문헌]

[1]Single cycle, PPT 자료

[2]Data path cycle, PPT 자료

[3]State element, PPT 자료

[4]R-type 명령어의 구조, PPT 자료

[5]R-type Data path, PPT 자료

[6]I-type 명령어 구조, PPT 자료

[7]J-type 명령어 구조, PPT 자료

[8]J-type 명령어 구조, PPT 자료

[9]Control signal, PPT 자료

[10]명령어 사이클, CPU/구조와 원리(r46 판)