제 14강. 4.4 단순한 구현, 4.6 파이프라이닝 개요

4.4 단순한 구현

ALU 제어

ALU 제어선	기능
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

ALU는 명령어 종류에 따라 첫 다섯 가지 기능 중 하나를 수행하게 됨

- 워드 적재, 워드 저장 명령어 : 메모리 주소를 계산하기 위한 덧셈을 하는 데 ALU 사용
- R형식 명령어 : 명령어 하위 6비트의 funct 필드값에 따라 다섯 가지 연산 중 하나를 수행
- 분기 명령어 : 뺄셈 수행

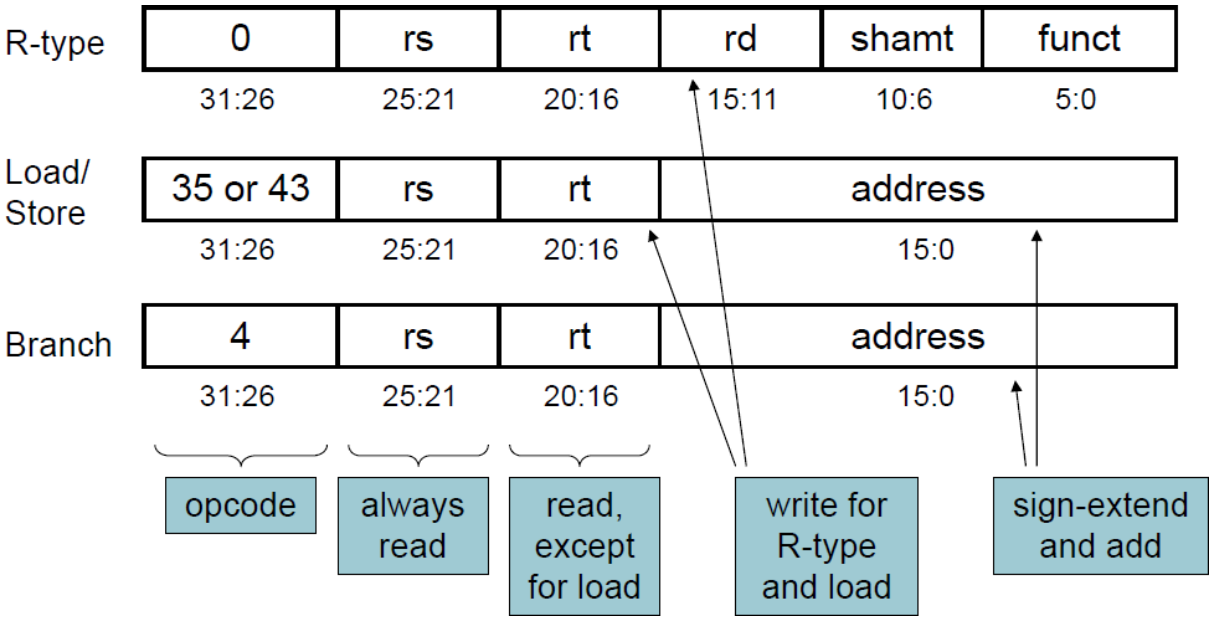
▼그림 4.12 ALU 제어 비트들은 ALUOp 제어 비트와 R형식 명령어의 funct 필드 값에 의해 결정됨.

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

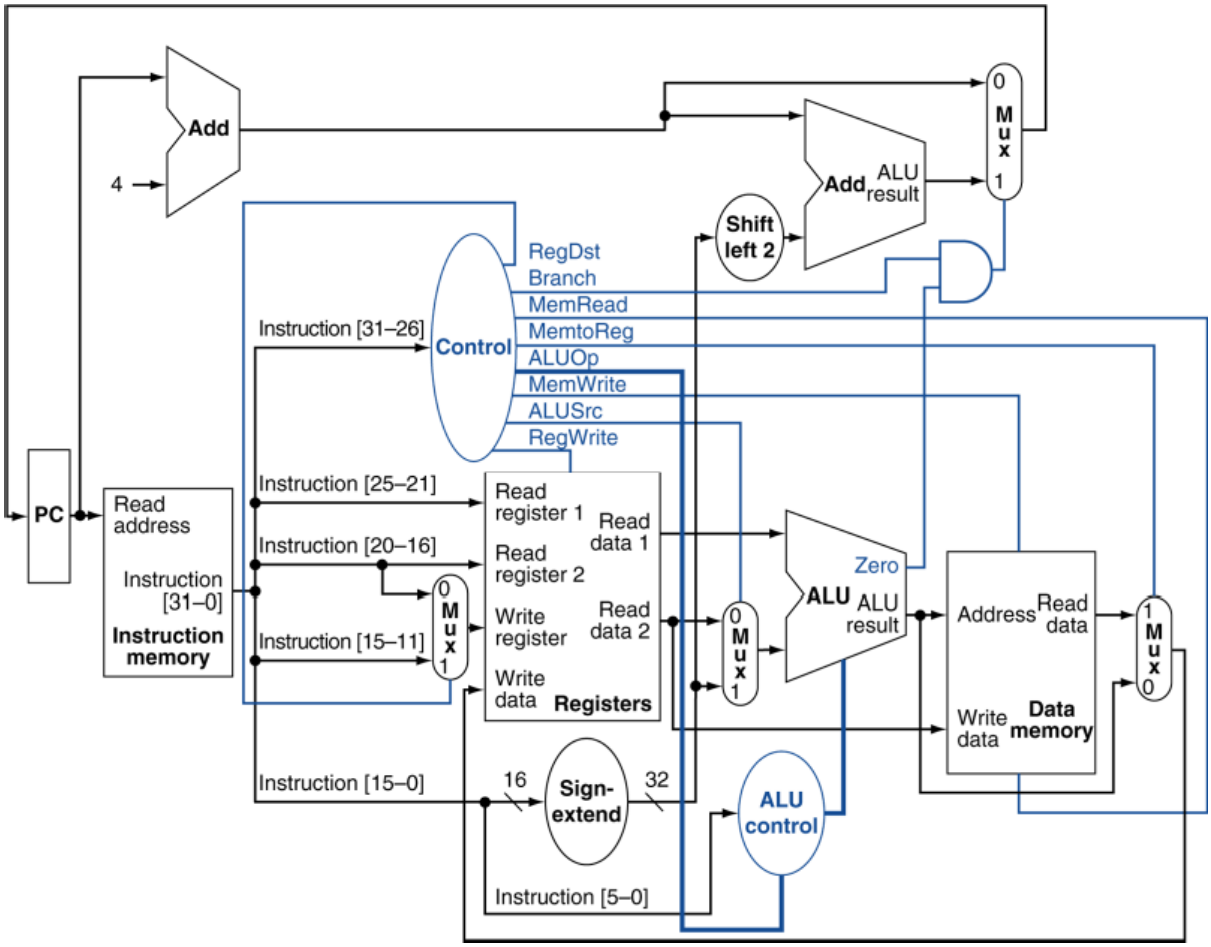
- ALUOp : 수행할 연산이 덧셈(00)인지 뺄셈(01)인지(beq 명령어) funct 필드에 따라 달라지는지(10) 표시
- ALUOp - 2비트, funct - 6bit, ALU control - 4bit

첫번째 열에 나열된 opcode가 ALUOp 비트값을 결정.

주 제어 유닛의 설계



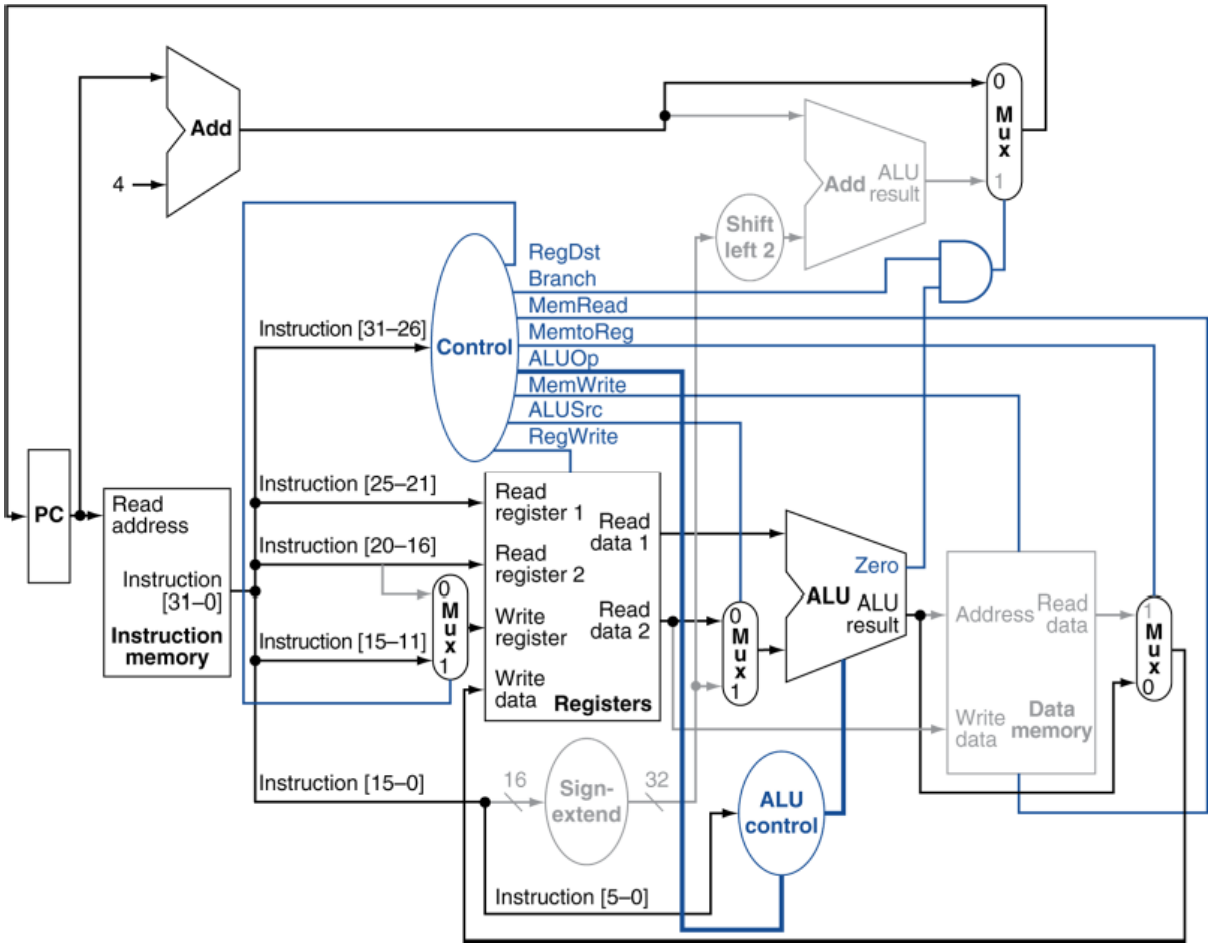
load/store , branch의 경우는 16비트를 sign-extend로 하고 연산



- opcode라 불리는 op필드는 항상 비트 31:26에 들어 있다. 앞으로 이 필드를 Op[5:0]이라 부를 것이다.
- 읽을 레즈스터 2개는 항상 rs, rt 필드에 의해 지정되는데 rs,rt 필드는 비트 25:21, 비트 20:16에 있다.
- 적재 저장 명령어를 위한 베이스 레지스터는 항상 비트 25:21(rs)에 있음
- beq, 적재 저장 명령어를 위한 16비트 변위는 항상 비트 15:0에 있음

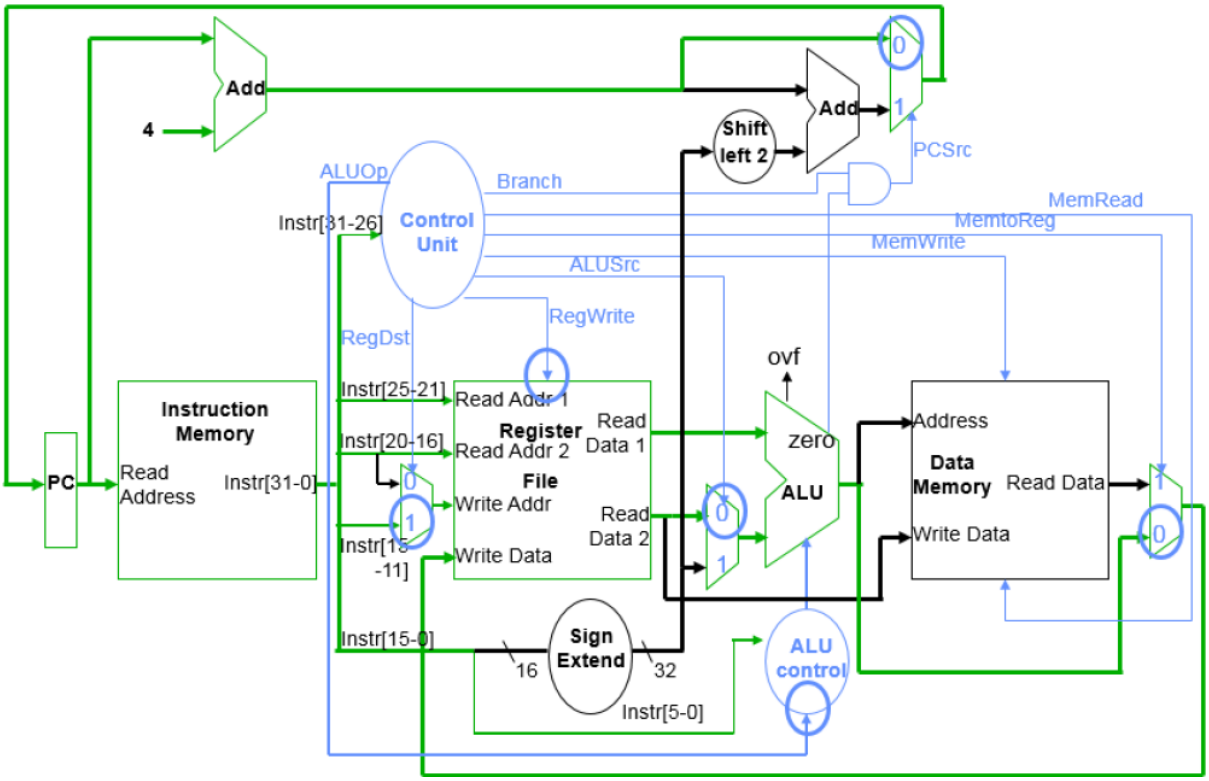
데이터패스의 동작

R형식 명령어

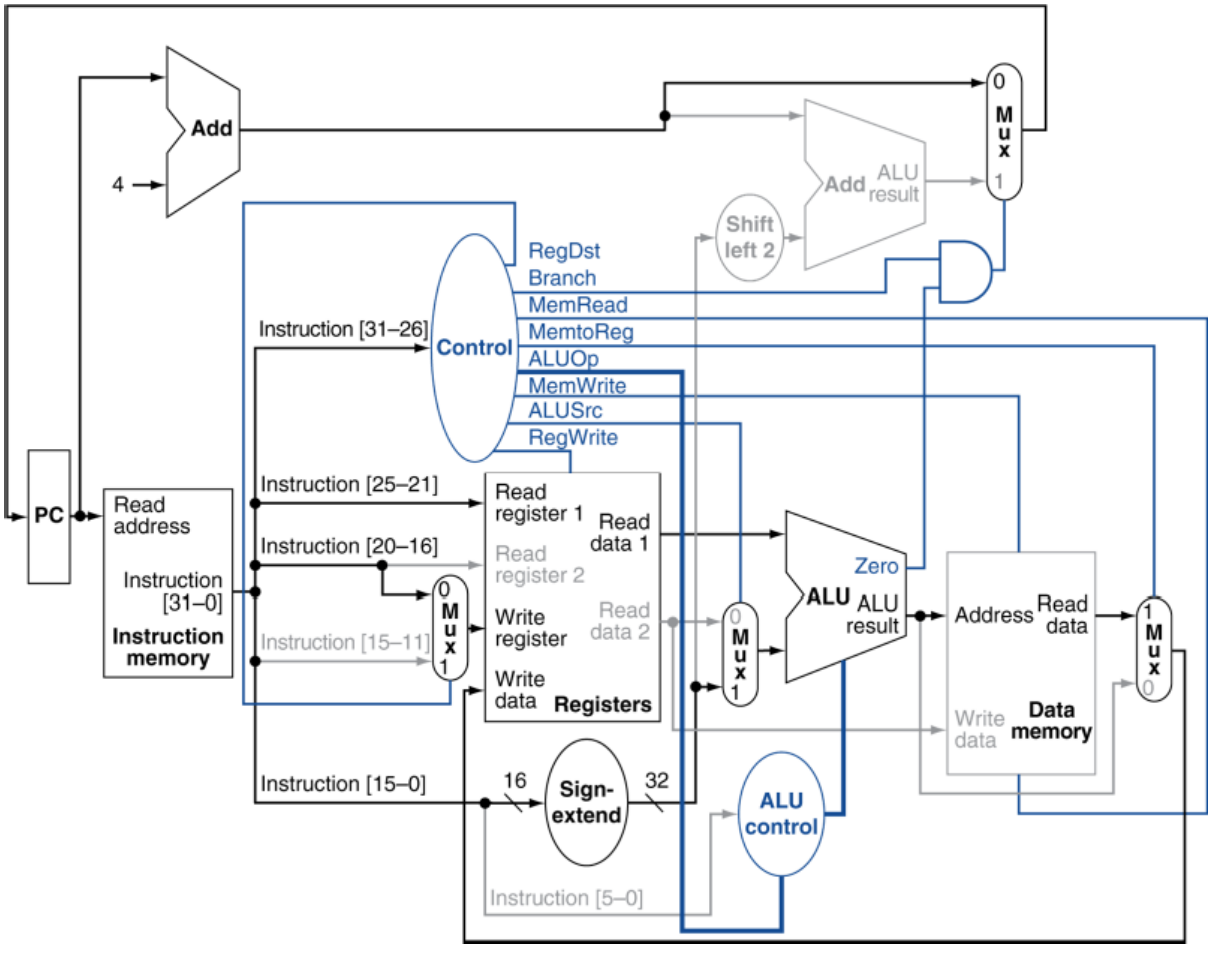


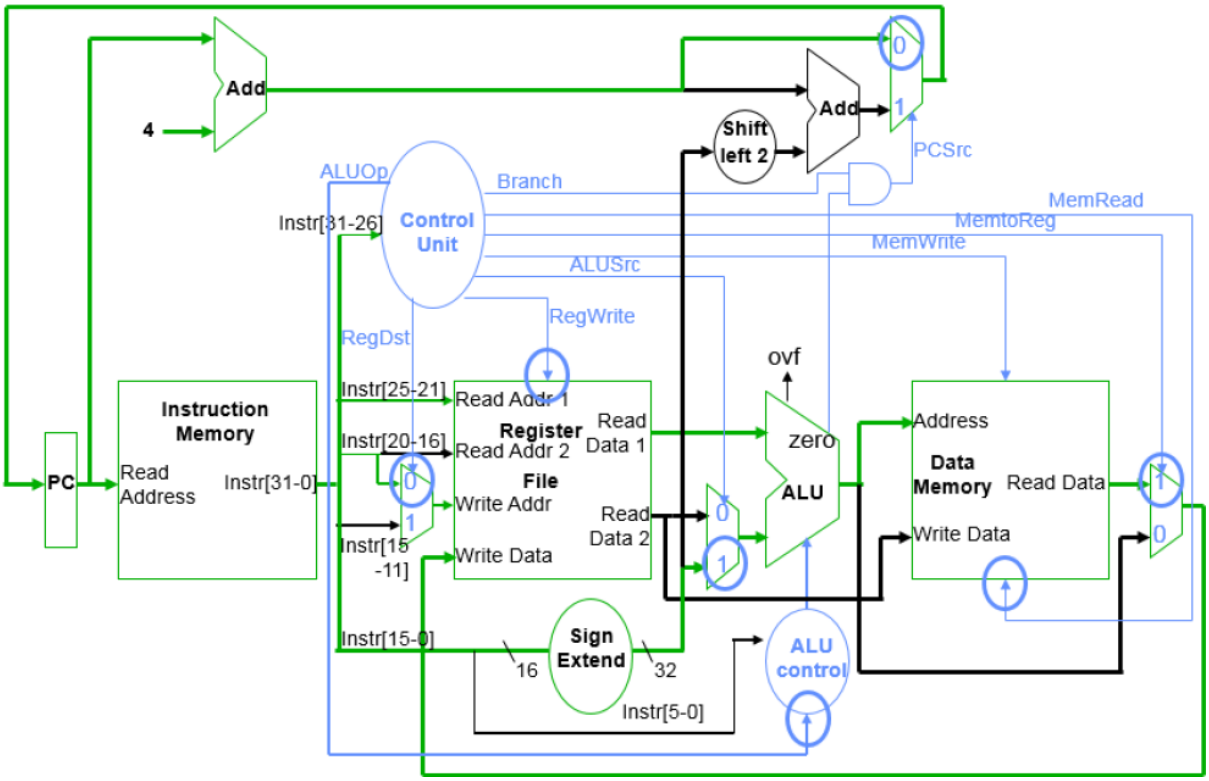
1. 명령어를 인출하고 PC(Program Counter)를 증가시킨다.
2. 레지스터 파일에서 두 레지스터 $t2, $t3를 읽는다. 동시에 주 제어 유닛은 제어선의 값들을 결정
3. ALU는 레지스터 파일에서 읽어 들인 값들에 대해 연산을 하는데 기능 코드를 사용하여 ALU 제어 신호를 만듦
4. ALU의 결과값을 레지스터 파일에 쓰되 명령어의 비트 15:11을 이용하여 목적지 레지스터($t1)를 선택

R 형식 명령어의 Data 및 Control 흐름


Add
4
Add
Shift left 2
PCSrc
ALUOp
Branch
Instr[31-26]
Control Unit
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ovf
Instruction Memory
Read Address
Instr[31-0]
PC
Instr[25-21]
Instr[20-16]
Instr[15-11]
Instr[15-0]
Register File
Read Addr 1
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
zero
ALU
Address
Data Memory
Read Data
Write Data
Sign Extend
16
32
Instr[5-0]
ALU control

적재 명령어

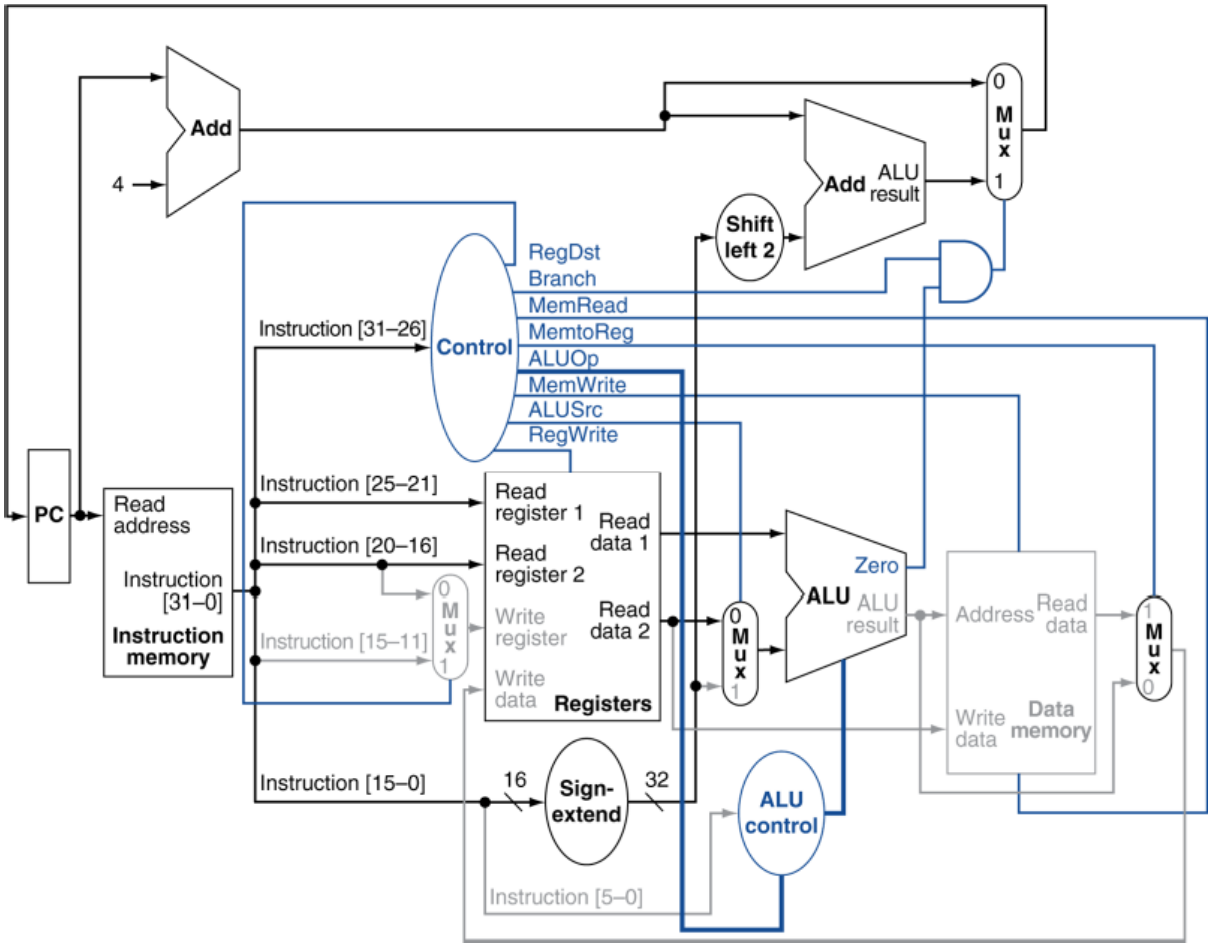

Add
4
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
Sign-extend
32
ALU control
M u x

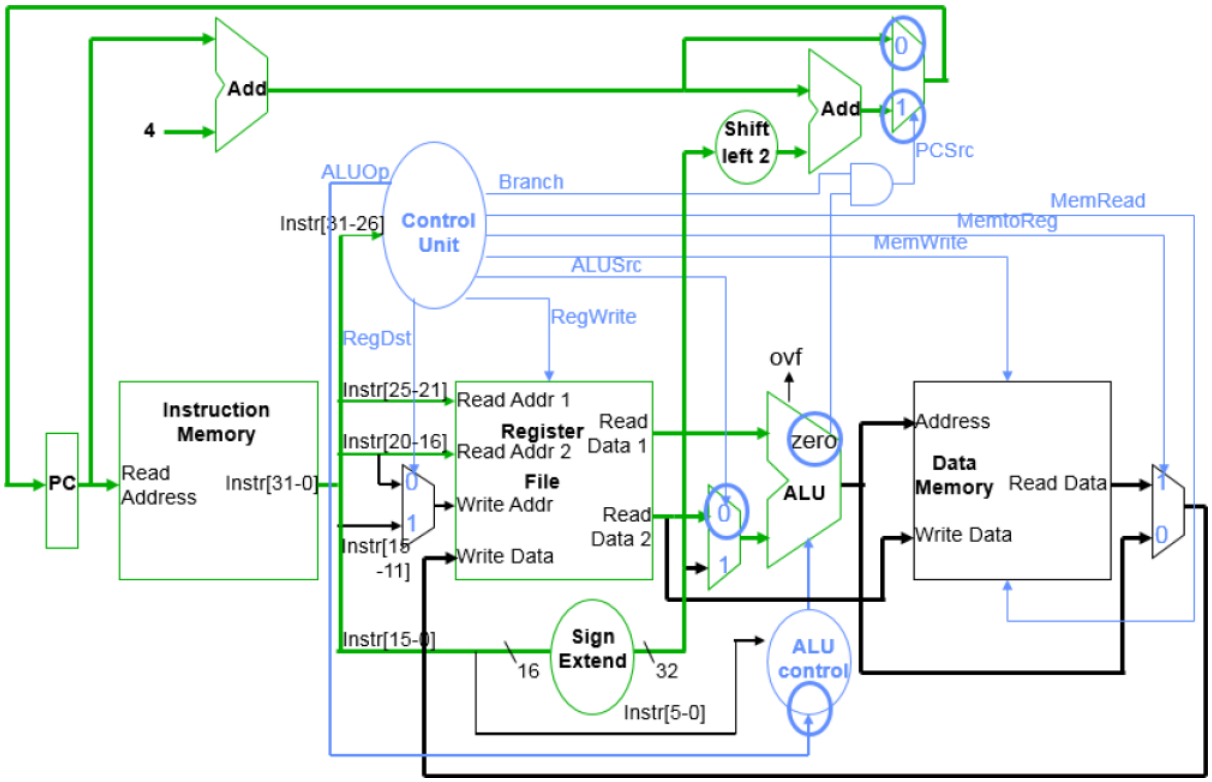


하드웨어는 모든 게 병렬적으로 실행 됨

1. 명령어를 명령어 메모리에서 인출하고 PC 값을 증가시킴
2. 레지스터 파일에서 레지스터 $t2를 읽음
3. ALU는 레지스터 파일에서 읽어들인 값과 명령어의 하위 16비트(offset)를 부호하나ㄱ장한 값의 합을 구한다.
4. 이 합을 데이터 메모리 접근을 위한 주소로 사용함
5. 메모리 유닛에서 가져온 데이터를 레지스터 파일에 씀. 목적지 레지스터는 명령어의 비트 20:16이 지정.

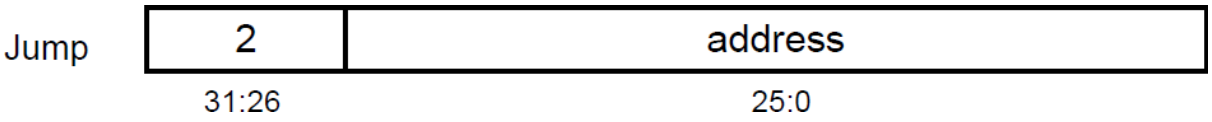
Branch-on-Equal 명령어





zero, branch 가 1이 됨

Implementing Jumps

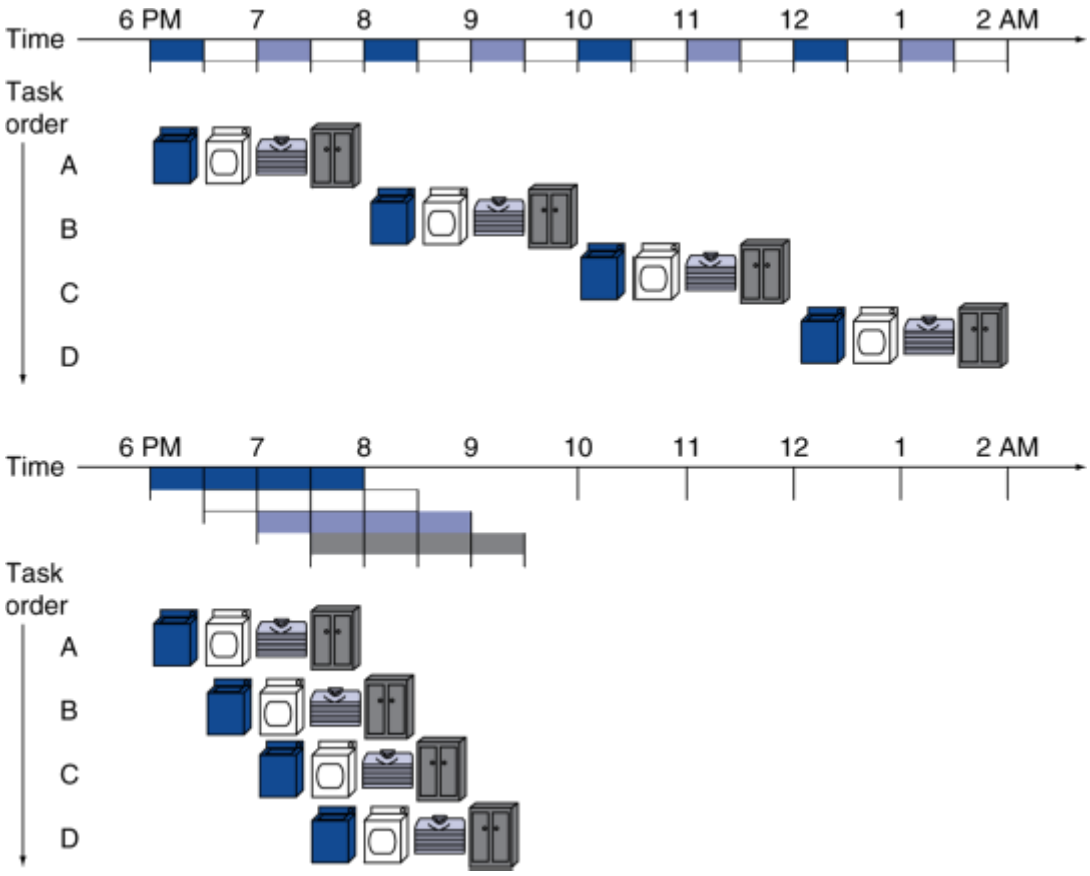


- word address(직접 주소)를 사용한 점프
- concatenation(이어붙이기) 방식으로 PC(Program Counter)를 업데이트
  - 26-bit의 address
  - 위에 left shift를 두 번하여 *4의 결과(28-bit)
  - 현재 PC(Program Counter)의 상위 4비트 뒤에 위를 합쳐서("old PC" & "address*4"), 총 32-bit.
- opcode로부터 해석된(decode) 추가적인 control 신호(signal)이 필요하다.

단일 사이클 구현은 오늘날 왜 사용되지 않는가?

- ‘자주 생기는 일을 빠르게’라는 1장의 핵심 설계 원칙을 위반
- 파이프라이닝은 단일 사이클 데이터패스와 매우 유사한 데이터패스를 사용하지만, 처리율이 훨씬 크기 때문에 매우 효율적 & 여러 개의 명령어를 동시에 실행하여 효율을 높임

4.6 파이프라이닝 개요

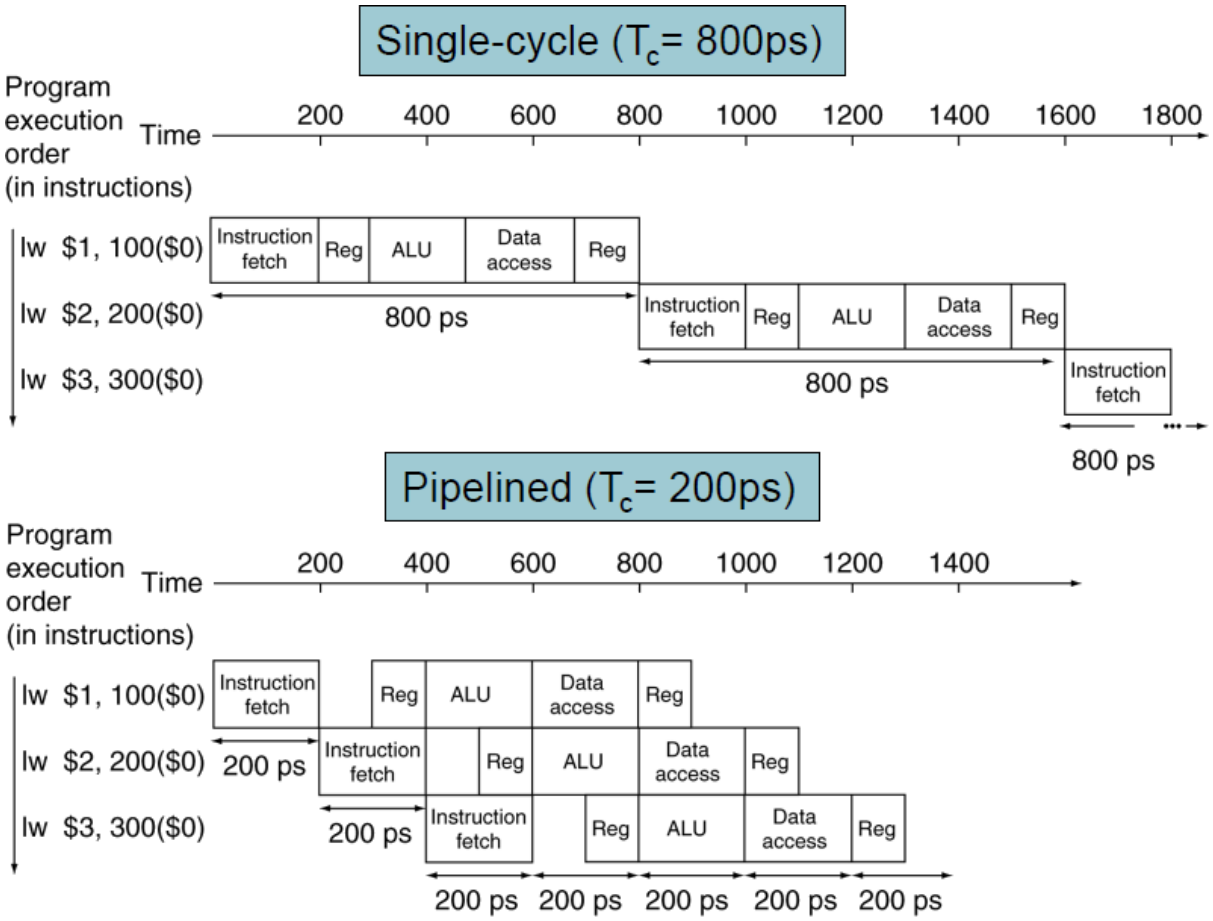


- pipleline된 빨래 : 겹치는 실행
  - 병렬처리(Parallelism)은 성능을 향상시켜 줌.
- 4번의 연속된 작업
  - 속도향상 = 8 / 3.5 = 2.3배의 성능 향상
- 계속
  - 속도향상 = 2n / 0.5n + 1.5 ≒ 4

MIPS Pipeline의 5단계

1. IF (Instruction Fetch) : 메모리로부터 명령어를 가져옴
2. ID (Instruction Decode &register read) 명령어 해독 & 레지스터 읽기
3. EX (EXcute operation) : 연산 수행 혹은 주소 계산
4. MEM (access MEMory operand) : 데이터 메모리의 피연산자에 접근
5. WB (Write result Back to register) : 결과값을 레지스터에 쓴다.

Instruction	Instr fetch	register read	ALU operation	memory access	register write	총 시간
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
R-Format	200ps	100ps	200ps		100ps	600ps
beq	200ps	100ps	200ps			500ps



$$\text{명령어 사이의 시간}_{\text{파이프라인}} = \frac{\text{명령어 사이의 시간}_{\text{파이프라인되지 않음}}}{\text{파이프 단계 수}}$$

딱 떨어지지는 않는 이유가 단계들의 시간이 늘 동일하지 않고

파이프라이닝은 어느 정도의 오버헤드를 발생시킨다.

파이프라이닝은 **개별 명령어의 실행 시간을 줄이지는 못하지만 대신 명령어 처리량을 증대시킴**으로써 성능을 향상시킴

파이프라이닝을 위한 명령어 집합 설계

1. 모든 MIPS 명령어는 같은 길이를 가짐
    a. 첫 번째 파이프라인 단계에서 명령어를 가져오고 그 명령어들을 두 번째 단계에서 해독하는 것을 훨씬 쉽게 해줌
2. MIPS는 몇 가지 안 되는 명령어 형식을 가지고 있음
    a. 모든 명령어에서 근원지 레지스터 필드는 같은 위치에 있음 : 이 같은 대칭성은 하드웨어가 어떤 종류의 명령어가 인출되었는지를 결정하는 동안 레지스터 파일 읽기를 동시에 할 수 있는 것을 의미
3. MIPS는 메모리 피연산자가 적재와 저장 명령어에서만 나타남
4. MIPS 피연산자는 메모리에 정렬되어 있어야 함
    a. 한 데이터 전송 명령어 때문에 메모리 접근을 2번 하는 경우는 없음
    b. 프로세서와 메모리 사이의 데이터 전송은 항상 파이프라인 단계 하나에서 처리됨