CA 4장 프로세서

4.1 서론

컴퓨터 성능은 3가지요인(명령어 개수, 클럭 사이클 시간, CPI)에 결정

기본적인 MIPS 구현

- 메모리 참조 명령어인 lw와 sw
- 산술/논리 명령어인 add, sub, AND, OR, slt
- 같을시 분기 명령어인 beq와 점프 명령어 j

이 부분 집합은 정수형 명령어를 모두 포함하지 않으며(자리이동, 나누기) 부동소수점 명령어는 하나도 포함x, 그렇지만 데이터패스와 제어 유닛을 설계하는 데 사용되는 핵심 원리

명령어 구현에 필요한 일

1. PC를 프로그램이 저장되어 있는 메모리에 보내어 메모리부터 명령어를 가져온다.
2. 읽을 레지스터를 선택하는 명령어 필드를 사용하여 하나 또는 두개의 레지스터를 읽는다. Lw 명령어는 레지스터 하나만 읽으면 되지만 대부분의 다른 명령어는 레지스터 두개를 읽는다.

ALU 사용

1. 메모리 참조 명령어는 주소 계산을 위해 사용
2. 산술/논리 명령어는 연산을 수행하기 위해 사용
3. 분기 명령어는 비교하기 위해 사용

4.2 논리 설계 관례

Information encoded in binary

=>Low voltage = 0, High voltage = 1

데이터 패스 요소에는 두가지 종류의 논리 소자들로 구성된다.

Combinational element(조합 소자)

=> 데이터 값에만 동작하는 소자, 그들의 출력이 현재의 입력에만 의존, 즉 같은 입력이 주어지면 항강 같은 출력을 냄 ex) ALU, AND-gaete, Multiplexer, Adder

State (sequential) elements

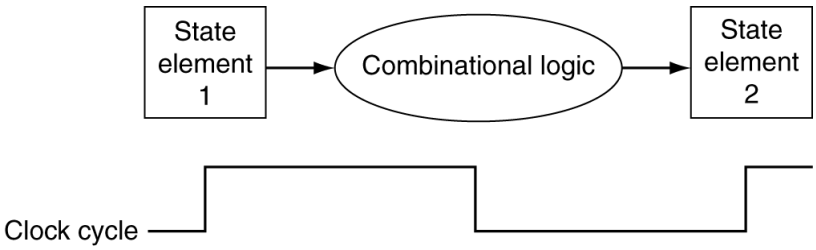
소자에 내부 기억장소가 있어 상태를 갖는 소자. 레지스터나 메모리 같은 메모리 소자.

적어도 2개의 입력과 1개의 출력을 갖는다. 꼭 있어야 되는 입력은 기록할 데이터와 클럭이다. 클럭 입력은 데이터 값이 소자에 기록되는 시점을 결정한다. 상태소자의 출력은 이전 클럭 사이클에 기록된 값이다. Ex) D형 플립플롭

Clocking Methodology(클러킹 방법론)

신호를 언제 읽을 수 있고 언제 쓸 수 있는 지를 정의

Edge-triggered clocking 방법론 => state element에 저장된 값은 클럭 에지에서만 바꿀수 있다는 것을 의미. 클럭에지는 높은 값에서 낮은값, 낮은 값에서 높은 값으로 변하는 시점이다. 모든 조합회로는 상태소자에서 입력을 받고 상태소자로 출력을 내보낸다. 입력은 이전 클럭 사이클에서 쓴 값이고 출력은 다음 클럭 사이클에서 사용할 수 있는 값이다. 이 방법론은 한 클럭 사이클 내에서 feedback이 되지 않는다.

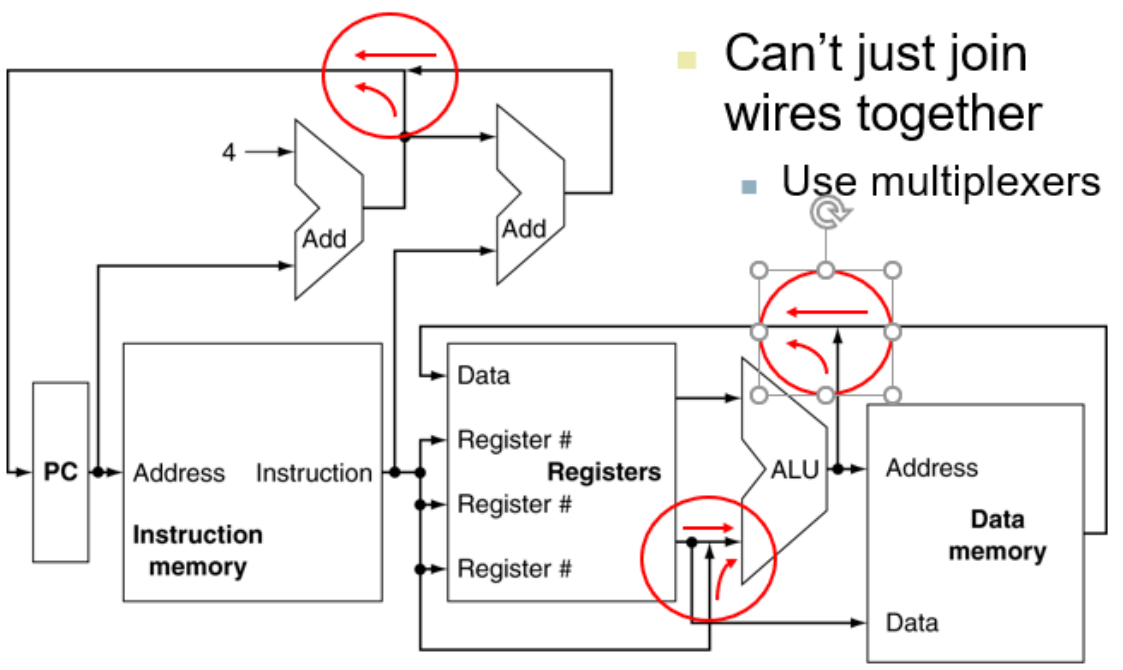


매 클럭 에지마다 상태소자에 쓰기가 행해지는 경우에는 앞으로 쓰기 제어신호 표시x 하지만 상태소자가 매 클럭마다 갱신되는 것이 아니라면 쓰기 제어신호가 분명하게 표시되어야 한다.

클럭 신호와 쓰기 제어신호가 상태소자의 입력이며, 쓰기 제어신호가 인가되고 활성화 클럭 에지일 때만 상태소자가 변하게 된다.

에지 구동 방법론은 race을 발생시킷 않으면서 같은 클럭 사이클에 상태소자를 읽고 쓸 수 있게 해준다.

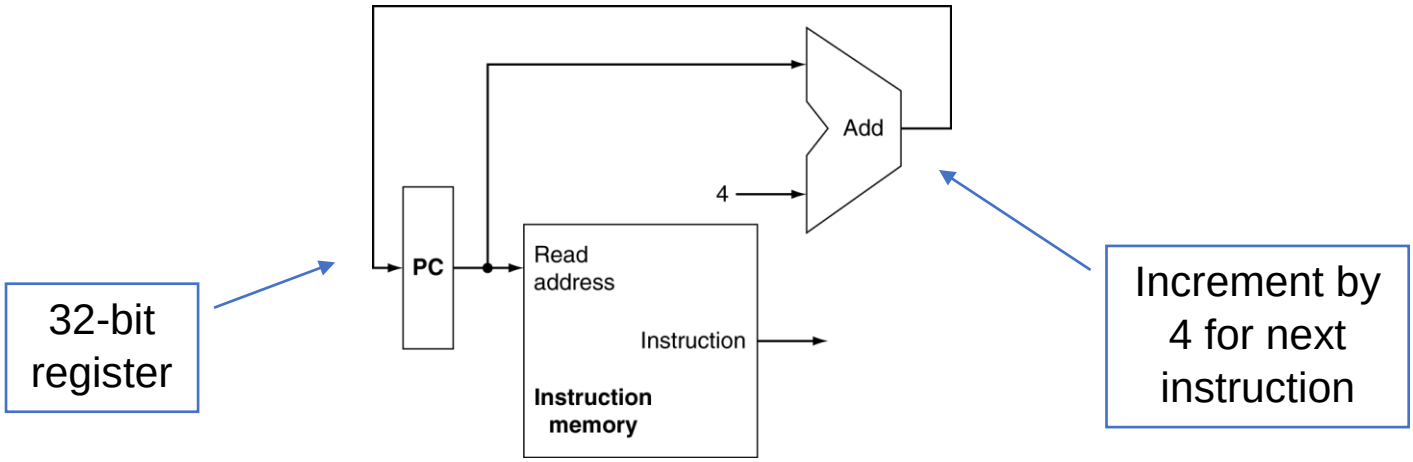
인가된(asserted) => 논리적으로 높은 신호를 표시



4.3 데이터 패스 만들기

데이터패스 구성요소 그뒤에는 추상화

PC : 프로그램에서 실행중인 명령어의 주소를 가지고 있는 레지스터



=> 상태소자는 명령어 메모리와 프로그램 카운터이다. 이 데이터 패스는 명령어를 쓸 필요가 없어서 명령어 메모리는 읽기 접근만 제공하면 된다. 출력은 항상 입력 주소가 지정하는 위치의 내용을 나타내면 읽기 제어신호가 필요하지 않다.

R-Format Instructions

두개의 레지스터 operands을 읽음

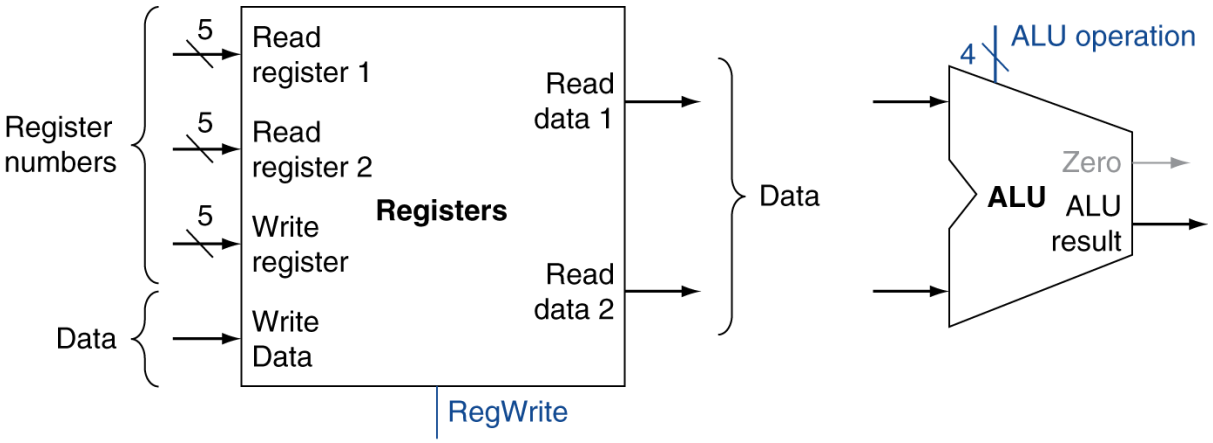
프로세서의 범용 레지스터 32개는 레지스터 파일이라는 구조속에 들어 있다. 레지스터 파일은 파일 내의 레지스터의 번호를 지정하면 어느 레지스터라도 읽고 쓸 수가 있다. 레지스터 파일은 컴퓨터의 레지스터 상태를 갖고 있다.

R 형식 명령어들은 레지스터 피 연산자 세 개를 가지고 있기 때문에, 매 명령어마다 레지스터 파일에서 두 데이터 워드를 읽고 데이터 워드 데이터 워드 하나를 써야 한다.

레지스터에서 데이터 워드를 읽기 위해서는 레지스터의 입력과 출력이 하나씩 필요하다.(입력: 읽을 레지스터 번호 지정, 출력: 레지스터에서 읽은 값을 내보내는 출력)

레지스터에서 데이터 워드를 쓰기 위해서는 입력이 두개 필요하다. (레지스터 번호 지정, 레지스터에 쓸 데이터 값)

레지스터 번호 입력은 32개의 레지스터 중 하나를 지정해야하니 데이터 입력과 출력 버스 모두 5비트 크기이다.



a. Registers b. ALU

=> R형식 ALU 연산을 구현하는 데 필요한 두개의 구성 요소는 레지스터 파일과 ALU이다.

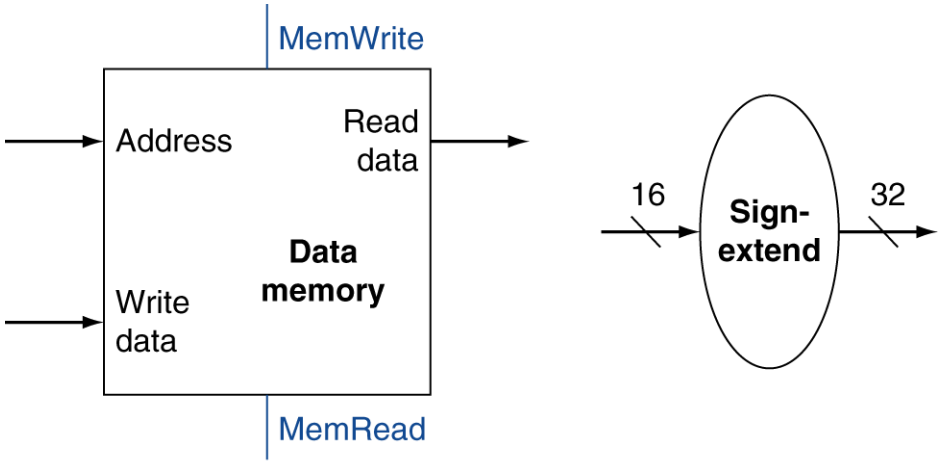
=> 분기 구현시 ALU의 zero 검출 출력을 사용

Load/Store Instructions

이 두 명령어들은 베이스 레지스터와 명령어에 포함되어 있는 16비트 부호있는 변위 필드를 더하여 메모리 주소를 계산한다. => Use ALU, but sign-extend offset

그 외에도 명령어의 16비트 변위 필드 값을 32비트 부호 있는 값으로 sign-extend 하기 위한 유닛이 필요하며 또 읽고 쓸 데이터 메모리가 필요하다. 데이터 메모리는 저장 명령어 일때에만 스기를 해야 한다. 따라서 데이터 메모리는 읽기와 쓰기 제어신호, 주소 입력, 메모리에 쓸 데이터 입력이 필요하다.

레지스터 파일은 항상 Read register 입력이 지정하는 레지스터의 내용을 출력하므로 다른 제어 입력이 필요 없다.



a. Data memory unit b. Sign extension unit

Branch Instructions

beq 명령어는 비교할 레지스터 두개와 16비트 변위의 세 피연산자를 갖는다. 변위는 분기 명령어 주소에 대한 상대적인 branch target address(분기 목적지 주소)를 계산하는 데 사용.

beq 명령어를 구현 하기 위해서는 PC 값에다가 명령어 변위 필드의 부호 확장 값을 더해서 branch target address를 계산해야 한다. 분기 명령어에서 주의 할 점 두가지는 다음과 같다.

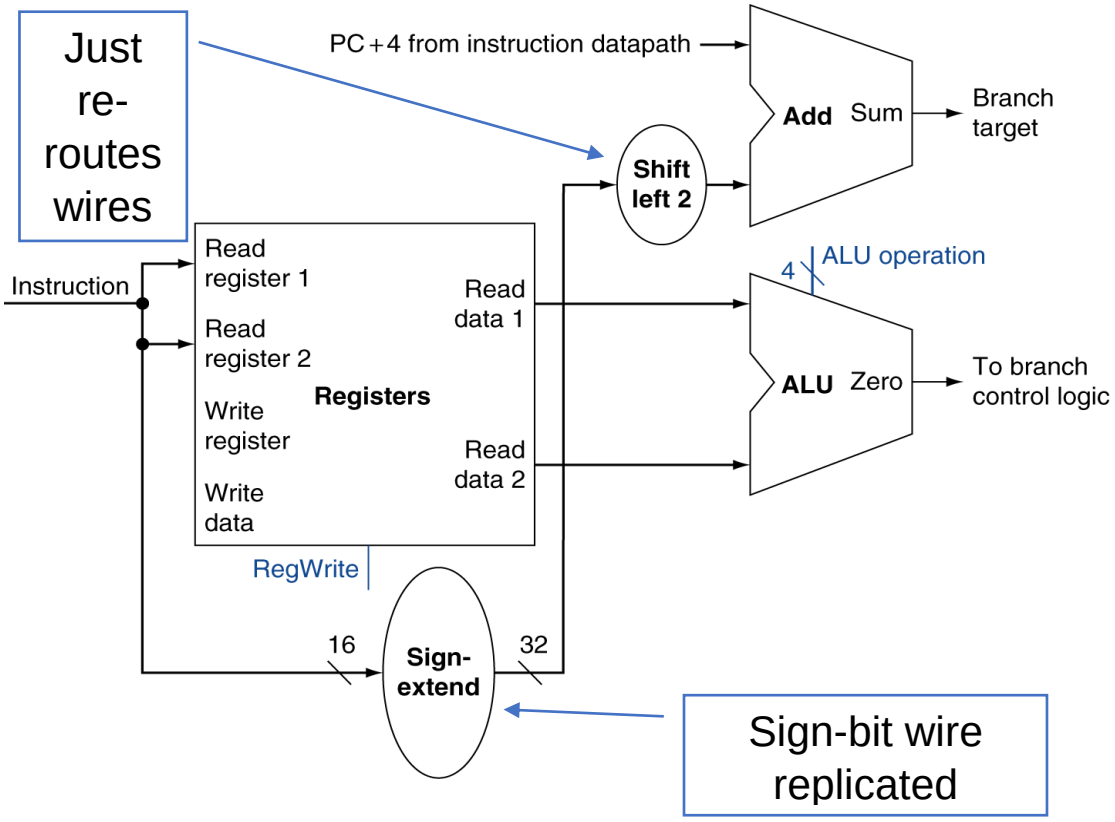
1. PC+4를 branch target address 계산의 베이스로 사용하자
2. 변위 필드를 2비트만큼 shift left한 만큼 워드 변위가 된다고 서술.

분기 데이터패스는 부호확장 유닛과 덧셈기 포함

비교 연산은 ALU을 사용. ALU는 결과가 0인지를 나타내는 출력 신호 제공하기 때문에, 두 레지스터 피연사자를 제어 신호와 함께 ALU에 보내 뺄셈을 하게 된다. ALU의 zero가 assert(인가) 되면 두개의 값이 같다는 것을 알 수 있다.

Jump 명령어는 명령어의 하위 26비트를 2비트만큼 shift left한 값으로 PC의 하위 28비트를 대체한다. 이 자리이동은 점프 변위 뒤에 00을 덧붙이면 된다.

Branch target address -> 증가한 PC값과 명령어의 하위 16비트를 부호 확장하고 왼쪽으로 2비트 자리이동한 값의 합이다.

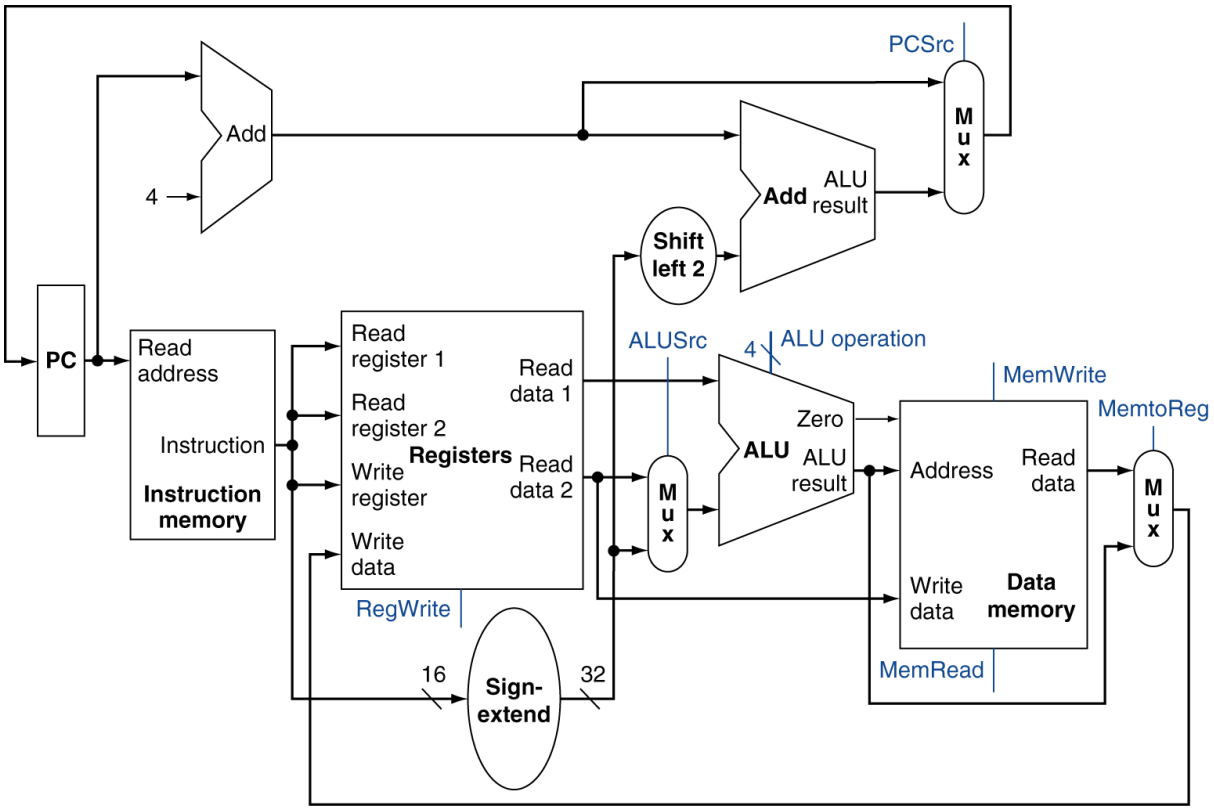


단일 데이터패스 만들기

가장 간단한 데이터패스는 모든 명령어를 한 클럭 사이클에 실행하도록 시도하는 것이다. 이것은 어느 데이터패스 자원도 명령어당 두번 이상 사용x, 두번이상 사용할 필요가 있을시 구성 요소는 필요한 만큼 여러 개 두어야 함. 그러므로 데이터 메모리와는 별도로 명령어 메모리가 필요. 몇몇 기능 유닛은 복제할 필요가 있지만, 많은 구성 요소들은 서로 다른 명령어의 흐름들이 공유하여 사용 가능

서로 다른 명령어 종류들이 데이터패스 구성요소를 공유할려면 그 구성요소의 입력에 여러 개의 연결을 허용해야 하며, 멀티 플렉서와 제어신호를 사용해서 그 입력들 중 하나를 선택해야 한다.

Full Datapath



단일 사이클 데이터패스는 명령어 메모리와 데이터 메모리를 따로따로 가져야 하는 이유

⇨ 프로세서가 명령어를 한 사이클에 실행하는데, 단일 포트의 메모리로는 한 사이클에 두개의 서로 다른 접근을 할 수 없기 때문이다.

4.4 단순한 구현

ALU Control

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Load/Store: F = add(메모리 주소를 계산하기 위해서), **Branch**: F = subtract, **R 형식 명령어**인 경우에는 명령어 하위 6비트 기능 필드 값에 따라서 5가지 연산중 하나를 수행

명령어 기능 필드와 2비트 제어필드(**ALUOp**)를 입력으로 갖는 제어 유닛을 만들어서 4비트 ALU 제어 입력을 발생.

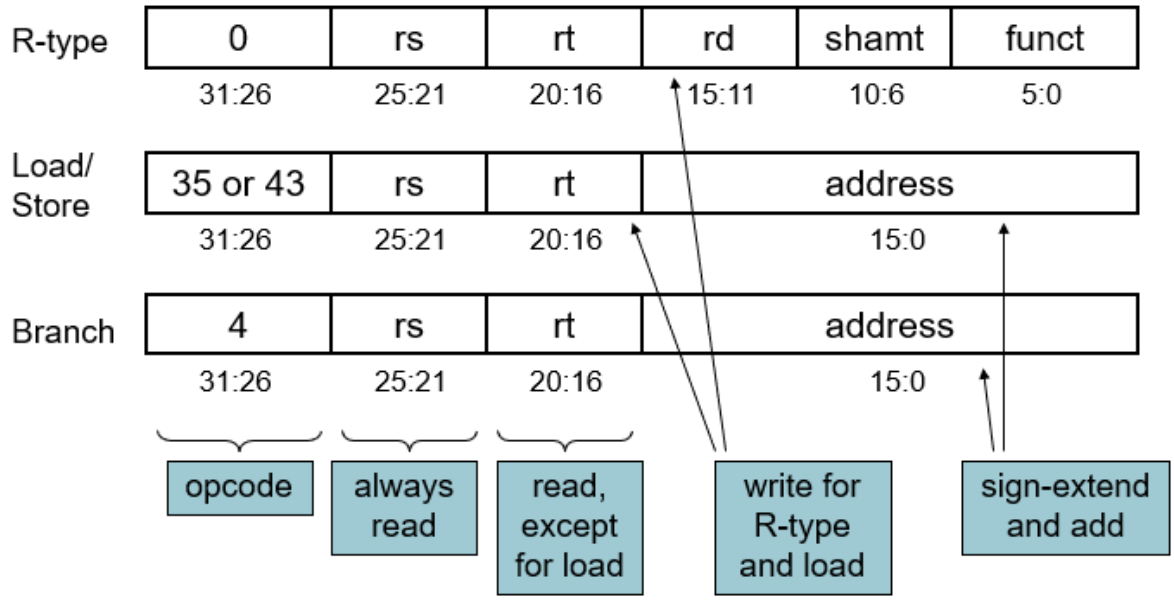
=> ALUOp값 00(덧셈) : lw, sw 01(뺄셈) : beq 10(연산) : 산술/논리 연산

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

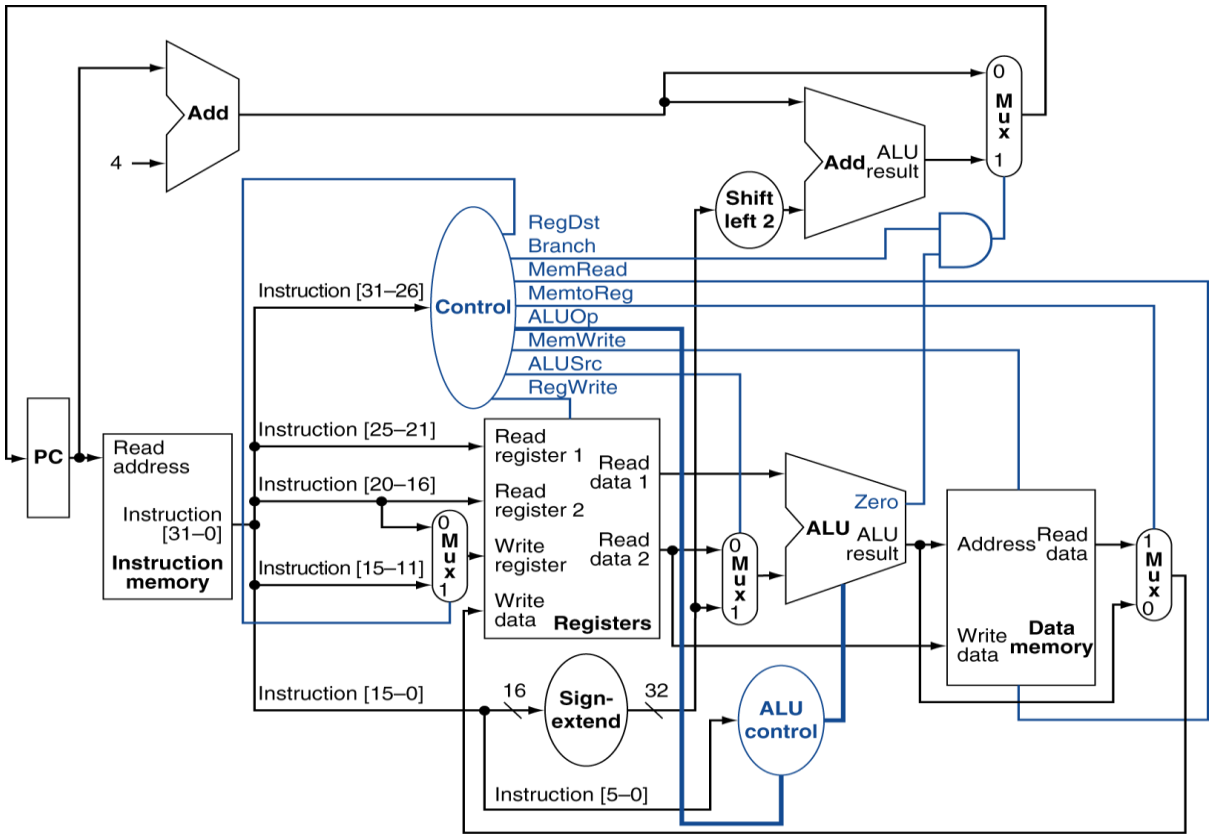
=> 여기서 ALUUP가 00이거나 01이면 해당 ALU 동작은 기능 코드 필드에 영향x 이 경우에 기능 코드 값에 대해 'don't care'라고 말하며 기능 필드는 XXXXXX로 표시된다. ALUOP가 10일 때는 기능 코드 값이 ALU 제어 입력을 결정하는 데 쓰인다.

Instru-ction	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op1	ALU Op0
R type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

The Main Control Unit

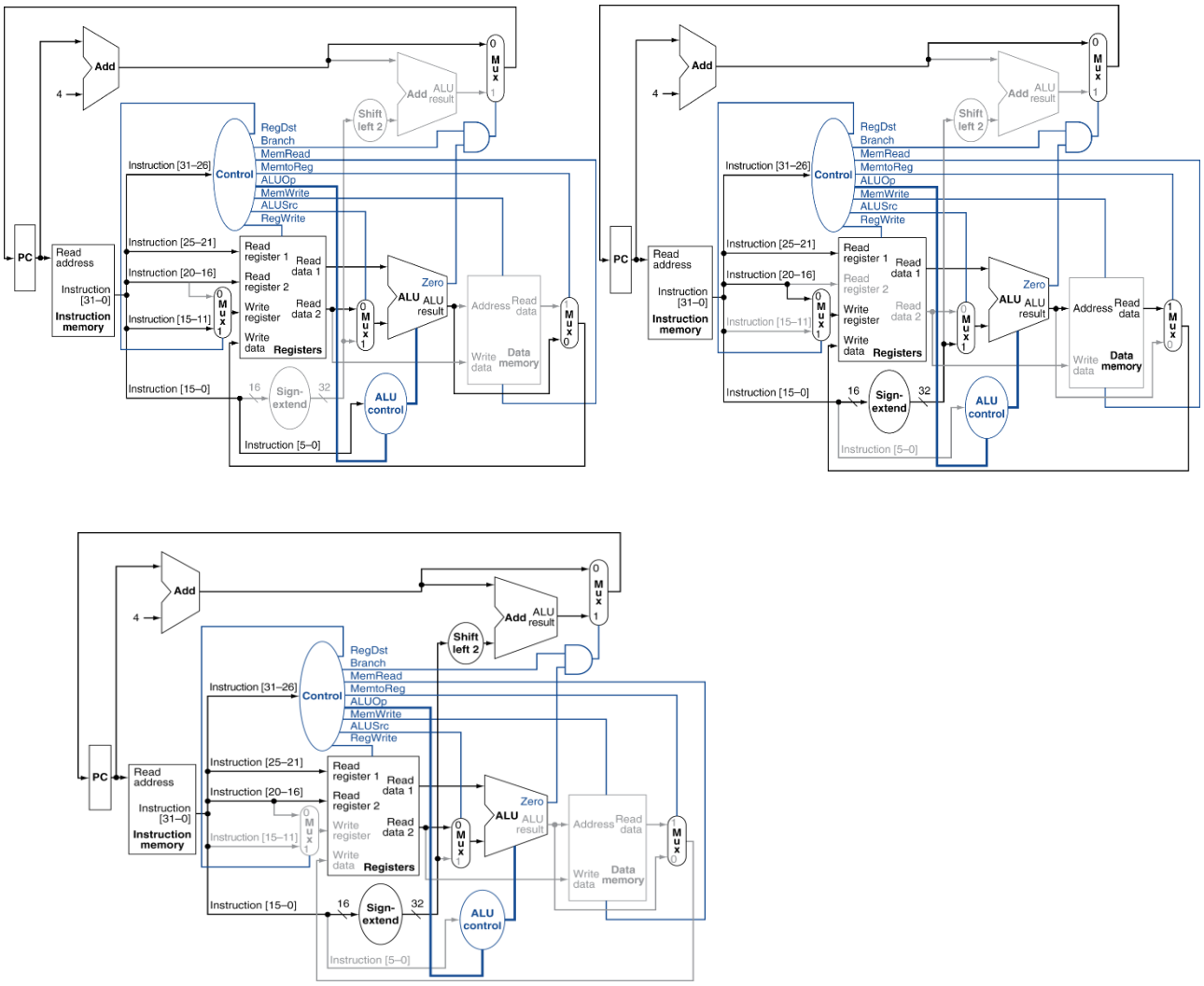


Datapath With Control



=> 제어선은 파란색으로 표시되었고 ALU 제어 블록이 추가되었음. PC는 매 클럭 사이클 끝에서 한번씩 쓰기가 행해지므로 쓰기 제어신호는 필요x, 분기 제어회로는 PC에다 증가된 PC 값을 쓸 것인지 분기 목적지 주소를 쓸 것인지 결정한다.

R-Type Instruction/ Load Instruction/ Branch-on-Equal Instruction



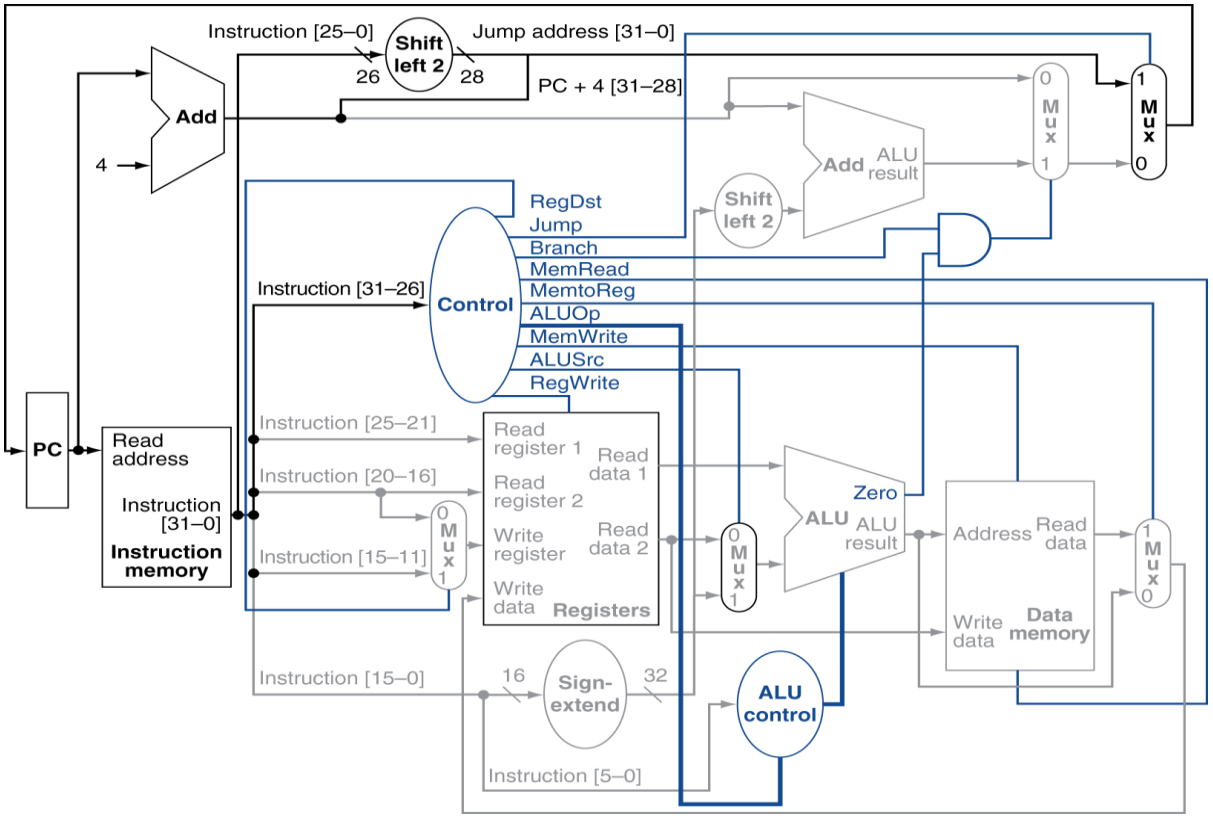
제어 유닛은 제어신호중 PCSrc 제어선을 제외한 나머지 모두를 명령어의 opcode 필드만 보고 결정할 수 있다. PCSrc 선은 실행 중인 명령어가 branch on equal이며 동시에 ALU의 ZERO 출력이 참일 경우에만 인가된다. PCSrc 신호는 제어 유닛에서 나오는 Branch 신호와 ALU의 Zero 신호를 AND한 것이다.

Implementing Jumps

1. 현재 PC+4의 상위 4비트(다음 명령어 주소의 비트 31:28)
2. 점프 명령어의 26비트 수치 필드
3. 비트 00(2) -> 하위 2비트

증가된 PC값, 분기 목적지 PC, 점프 목적지 PC중 하나를 새로운 PC값 근원지로 선택하기 위해 멀티 플렉서가 추가

Jump 명령어를 추가하기 위해 새로운 제어신호가 하나 추가로 필요함. Opcode가 2일때만 인가



신호 이름	인가되지 않은 경우(0)	인가된 경우(0)
RegDst	명령어 rt필드가 write register 번호 입력이 된다.	명령어 rd 필드가 write register 번호 입력이 된다.
RegWrite	아무 일도 생기지 않는다.	Write register 입력이 지정하는 레지스터에 Write data 입력값을 쓴다
ALUSrc	레지스터 파일의 두번째 출력(Read data2)이 ALU의 두번째 피연산자가 된다.	명령어 하위 16비트가 부호확장되어 ALU의 두번째 피연산자가 된다
PCSrc	PC+4가 새로운 PC값이 된다	분기 목적지 주소가 새로운 PC값이 된다.
MemRead	아무 일도 생기지 않는다.	Address 입력이 지정하는 데이터 메모리 내용을 Read Data 출력으로 내보낸다
MemWrite	아무 일도 생기지 않는다.	Address 입력이 지정하는 데이터 메모리 내용을 Write Data 출력으로 내보낸다
MemtoReg	ALU 출력이 레지스터의 Write data 입력이 된다.	데이터 메모리 출력이 레지스터의 Write data 입력이 된다.ㄴ

단일 사이클 구현 사용안하는 이유

1. 단일 사이클 설계는 클럭 사이클이 모든 명령어에 대해 같은 길이를 가져야 하기 때문에 컴퓨터에서 가능한 경로중 가장 긴 경로로 결정된다. 이 최장 경로는 load 명령어가 거의 확실한데, 그 이유는 적재 명령어는 Instruction memory -> register file -> ALU -> data memory -> register file을 거쳐서 그렇다. => 전체적 성능이 좋지 않다.
2. 또한 부동소수점 유닛을 구현하거나 좀더 복잡한 명령어 같은 경우는 단일 사이클로 잘 작동X
3. 클럭 사이클 동안 공유 될 수 없기 때문에 일부 기능 유닛 (예 : 가산기)이 복제되어야하므로 영역을 낭비 할 수 있습니다.

그러나 이것은 이해하기 쉽고 간단하다! => **장점**

Implementing the Control

(Ref: Appendix-D: Mapping Control to Hardware)

Single Cycle Control Unit: ALU control

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

FIGURE C.2.1 The truth table for the three ALU control bits (called Operation) as a function of the ALUOp and function code field. This table is the same as that shown Figure 5.13.

ALUOp는 입력 값 11을 사용하지 않는다. 따라서 진리표에서 10과 01대신 1x와 x1을 사용할 수 있다. 또 R형식 명령어가 사용되는 경우 이 필드의 첫 두 비트는 항상 10이다. 따라서 그들을 don't care항으로 받아들이고 진리표에서 XX로 표시한다.

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

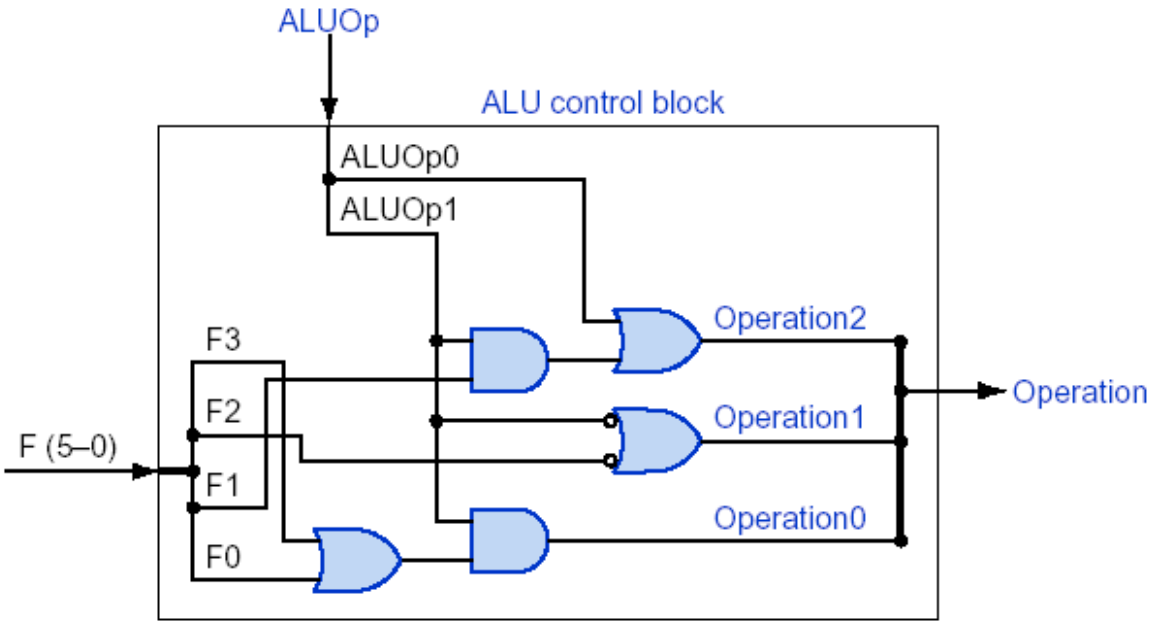
a. The truth table for Operation2 = 1 (this table corresponds to the left bit of the Operation field in Figure C.2.1)

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

b. The truth table for Operation1 = 1

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

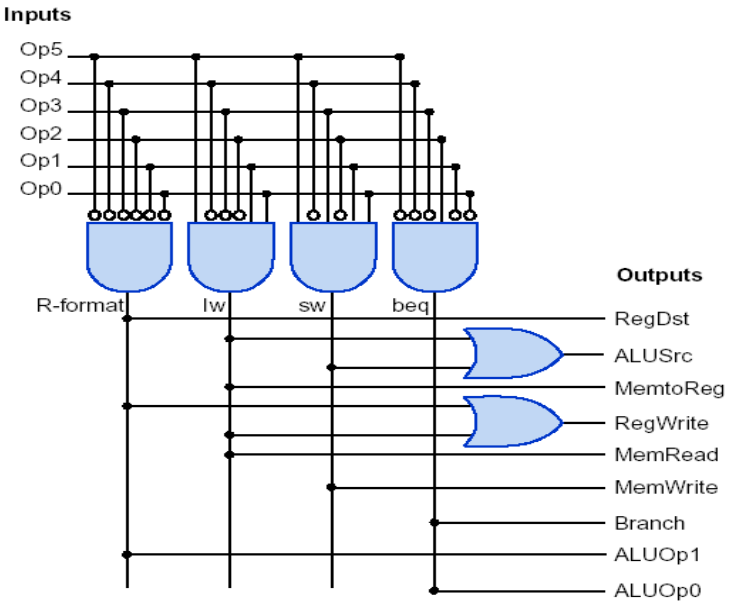
c. The truth table for Operation0 = 1



Operation 2을 고려해보면 ALUOp0이 1이면 항상 참, ALUOp1이 1이고 F1값이 1이면 항상 참이다.

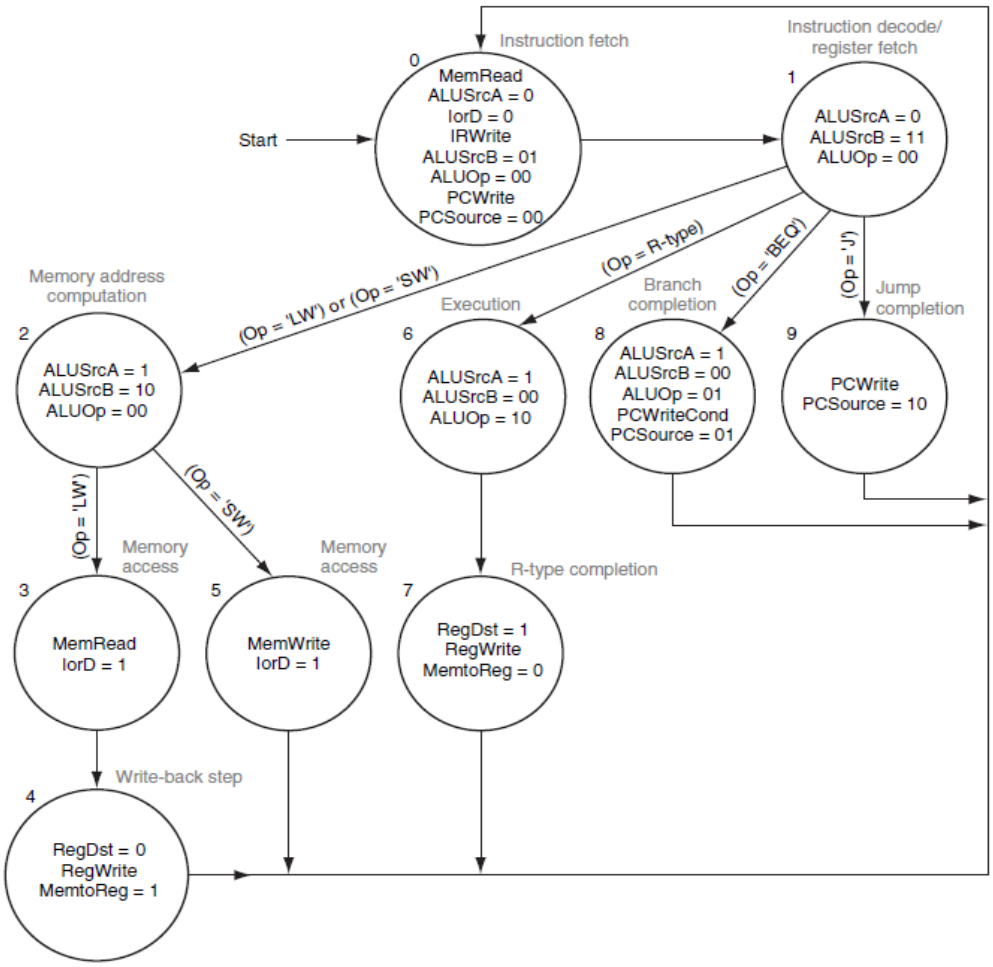
Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE C.2.4 The control function for the simple one-clock implementation is completely specified by this truth table. This table is the same as that shown in Figure 5.22.



⇨ 위에 표를 PLA라고 한다. 그리고 PLA를 논리 회로로 표현하는 것이 밑에 그림이다. 구조화된 두 레벨 논리 배열을 사용하는 것이다. 표가 주어질 때 저 PLA을 그려보자!

Implementing Finite-State Machine Control



Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

FIGURE C.3.3 The logic equations for the control unit shown in a shorthand form.

PCSource0, 1 랑 ALUOp0,1이랑 ALUSrcB0,1 헷갈리지 말자!! A1이면 2진수에서 앞자리 값이 1인 10값이나 11값이다. A0이면 2진수에서 앞자리 값이 0인 01값 또는 11값이다(00은 아니다!), 그러고 값이 0인인 것 포함X(예를 들어 stateo에서 ALUSrcA=0 인데 이것은 포함안한다)

그리고 그냥 state0에서 pcwrite만 쓴것도 포함된다.

다음 상태 비트 NS0은 상태 인코딩에서 다음 상태가 NS0 = 1 일 때마다 활성화되어야한다. 이는 NextState1, NextState3, NextState5, NextState7 및 NextState9에 해당됩니다

$$
\begin{aligned}
\text{NextState1} &= \text{State0} = \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0} \\
\text{NextState3} &= \text{State2} \cdot (\text{Op}[5\text{-}0] = \text{lw}) \\
&= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot Op0 \\
\text{NextState5} &= \text{State 2} \cdot (\text{Op}[5\text{-}0] = \text{sw}) \\
&= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot Op3 \cdot \overline{Op2} \cdot Op1 \cdot Op0 \\
\text{NextState7} &= \text{State6} = \overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0} \\
\text{NextState9} &= \text{State1} \cdot (\text{Op}[5\text{-}0] = \text{jmp}) \\
&= \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{Op5} \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot \overline{Op0}
\end{aligned}
$$

=> NS0 is the logical sum of all these terms

s3	s2	s1	s0
0	1	0	0

g. Truth table for MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Truth table for PCSource1

s3	s2	s1	s0
1	0	0	0

i. Truth table for PCSource0

s3	s2	s1	s0
0	1	1	0

j. Truth table for ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Truth table for ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Truth table for ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Truth table for ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Truth table for ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

p. Truth table for RegDst

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Truth table for PCWrite

s3	s2	s1	s0
1	0	0	0

b. Truth table for PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Truth table for IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Truth table for MemRead

s3	s2	s1	s0
0	1	0	1

e. Truth table for MemWrite

s3	s2	s1	s0
0	0	0	0

f. Truth table for IRWrite

s3	s2	s1	s0
0	1	0	0

g. Truth table for MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Truth table for PCSource1

s3	s2	s1	s0
1	0	0	0

i. Truth table for PCSource0

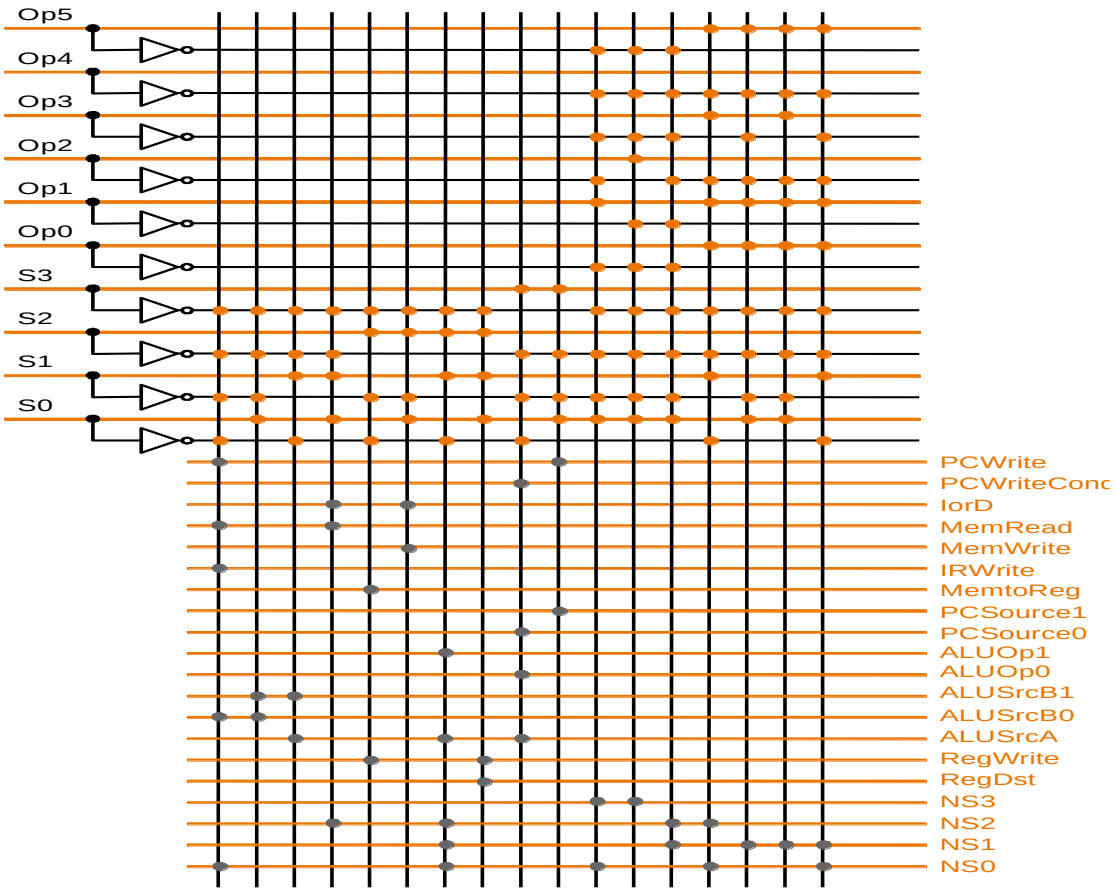
=> 여기서 s3 s2 s1 s0는 스테이트를 나타내는 것이다 1 0 0 1 이면 state9를 나타냄.

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

FIGURE C.3.6 The truth table for the 16 datapath control outputs, which depend only on

=> 예를 0들어 PCWriteCond, PCSource0 및 ALUOp0의 경우이 신호는 state 8에서만 활성화됩니다. 이 세 신호는 하나의 신호로 대체 될 수 있습니다.

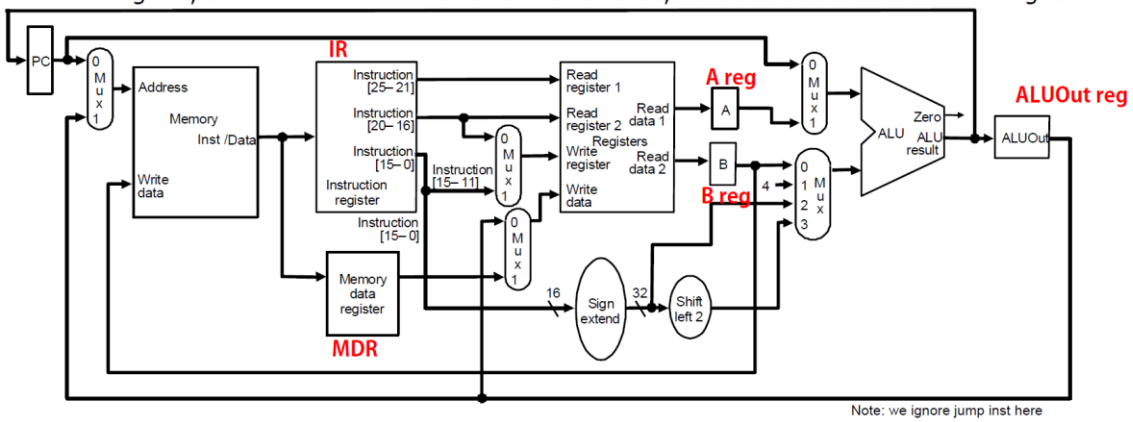
A PLA Implementation



=> PLA의 전체 크기는 (#inputs × #product terms) + (#outputs × #product terms)에 비례합니다. 그림에서 상징적으로 볼 수 있습니다. 이는 그림 D.3.9의 PLA의 전체 크기가 (10 × 17) + (20 × 17) = 510에 비례 함을 의미합니다 PLA를 두 개로 나눌 수 있습니다.
PLA : 16 개의 제어 출력을 생성하는 4 개의 입력과 10 개의 미 터를 가진 하나, 4 개의 다음 상태 출력을 생성하는 10 개의 입력과 7 개의 미 터를 가진 것. 제 1 PLA는 (4 × 10) + (10 × 16) = 200에 비례하는 크기를 가질 것이고, 제 2 PLA는 (10 × 7) + (4 × 7) = 98에 비례하는 크기를 가질 것이다. 총 298 개의 PLA 세포에 비례하는 총 크기, 단일 크기의 약 55 %이다.

Multicycle Datapath Approach

- Additional "internal registers":
  - Instruction register (**IR**) -- to hold current instruction
  - Memory data register (**MDR**) -- to hold data read from memory
  - A register (**A**) & B register (**B**) -- to hold register operand values from register files
  - ALUOut register (**ALUOut**) -- to hold output of ALU, also serves as memory address register (MAR)
- All registers except IR hold data only between a pair of **adjacent** cycles and thus do **not** need **write** control signals; IR holds instructions till **end** of instruction, hence **needs** a **write** control signal



멀티 사이클 장점

1. 몇 개의 명령어들이 사이클의 수가 짧아져서 다음 명령어를 빨리 실행가능,
2. 싱클 사이클보다 빠름
3. 멀티 사이클은 hardware cost도 줄여준다.

   (reduce adders & memory, increase number of registers &muxes)

Instruction register(IR) -> 현재 명령어를 유지, 클럭사이클이 끝나도 Instruction이 끝나기전까지 저장된 값을 유지한다

Multicycle Data Path에서 lw, sw, R-type, branch, jump 는 각각 어떤 Execution step을 거치고 각각 몇 clock cycle이 걸리는가?

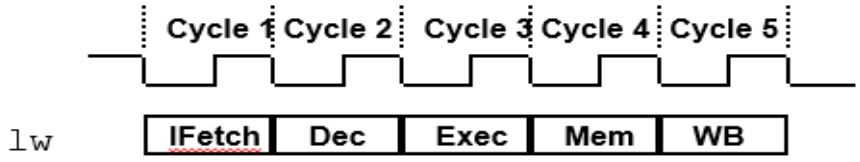
lw 는 5 clock cycle 을 거치고 Instruction Fetch, Instruction Decode, Memory Address Computation, Memory Access, Memory Read Completion 의 step 들을 거친다

sw 는 4 clock cycle 을 거치고 Memory Address Computation 까지 lw 와 동일한 과정을 거치고 Memory Access 에서 끝이난다

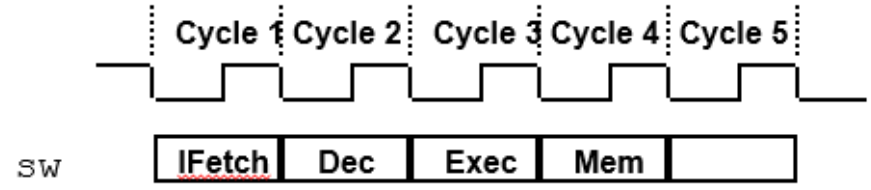
R-Type 은 4 clock cycle 을 거치는데 IR Fetch, ID, ALU execution, R-type Instruction Completion 으로 끝난다

Branch 는 3 clock cycle 을 거친다. IR Fetch, ID, Branch Completion 을 거친다

Jump 또한 Branch 와 동일하다.



- **IFetch: Instruction Fetch and Update PC**
- **Dec: Instruction Decode, Register Read, Sign Extend Offset**
- **Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion;**
- **Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)**
- **WB: Memory Read Completion (RegFile write)**

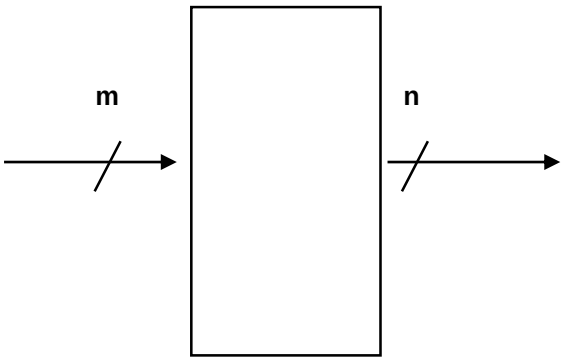


- **IFetch: Instruction Fetch and Update PC**
- **Dec: Instruction Decode, Register Read, Sign Extend Offset**
- **Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion;**
- **Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)**

ROM Implementation

ROM = "Read Only Memory", 메모리 위치 값은 미리 고정되어 있습니다.

A ROM can be used to implement a truth table , m은 height이고 n은 width이다.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., 2^{10} = 1024 different addresses)
- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is 2^{10} x 21 = 21K bits (and a rather unusual size)
- 많은 항목의 경우 출력이 동일하기 때문에 낭비하지 않습니다.

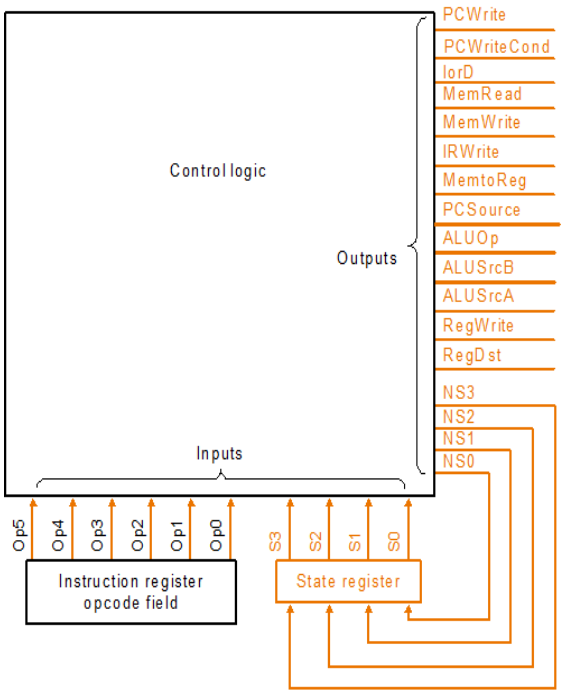
전체의 ROM을 두개의 part로 나누어 구현한다. 16bit data path는 4 bit의 state inputs에만 의존하고 4bit의 NextState bit는 10개의 address bit 모두를 필요로 한다. 이렇게 NS bit를 구하는 ROM과 data path를 구하는 ROM, 두개의 ROM으로 나누어 구현하면 각각 2^4 x 16 와 2^{10} x 4 이므로 이 둘을 더하면 256 + 4096 = 4.3K이다.

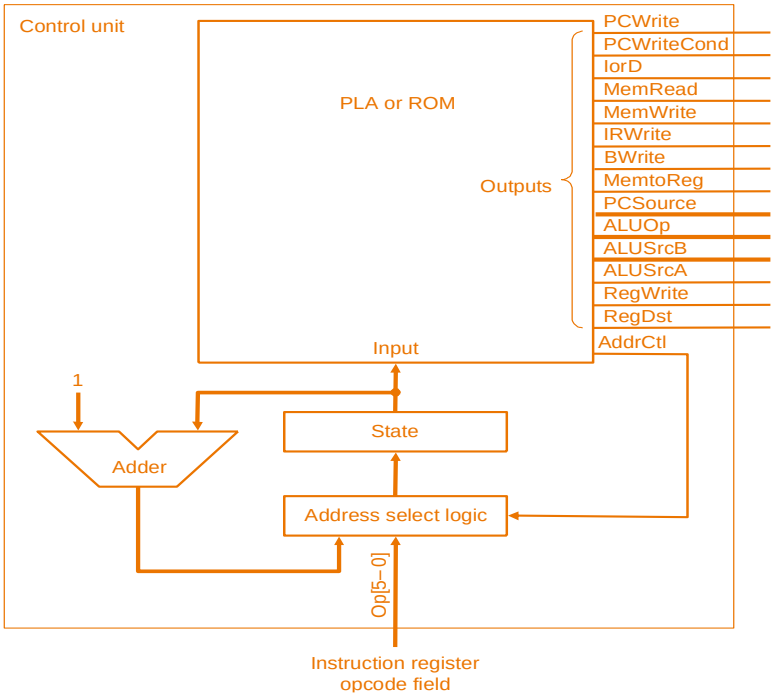
PLA의 경우 $(\#\text{inputs} \times \#\text{product terms}) + (\#\text{outputs} \times \#\text{product terms})$가 size의 값인데 개선 전 PLA는 $(10 \times 17) + (20 \times 17) = 510$이다.

Of course, just as we split the ROM in two, we could split the PLA into two PLAs: one with 4 inputs and 10 minterms that generates the 16 control outputs, and one with 10 inputs and 7 minterms that generates the 4 next-state outputs. The first PLA would have a size proportional to $(4 \times 10) + (10 \times 16) = 200$, and the second PLA would have a size proportional to $(10 \times 7) + (4 \times 7) = 98$. This would yield a total size proportional to 298 PLA cells, about 55% of the size of a single PLA. These two PLAs will be considerably smaller than an implementation using

Another Implementation Style

Complex instructions: the "next state" is often current state + 1

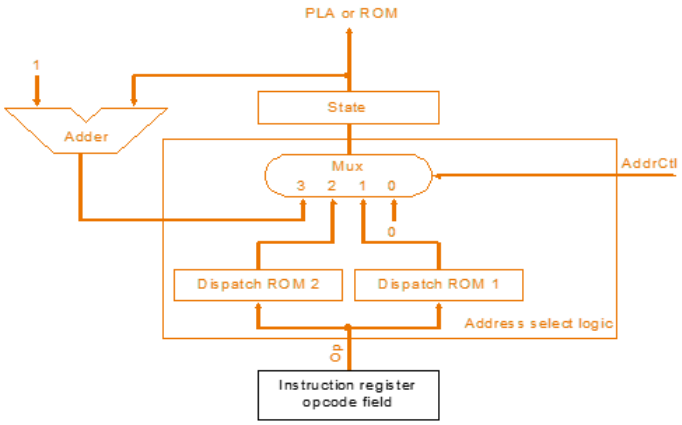




왼쪽이 기존 오른쪽이 새로운 Implementation이다.

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

스스로 점검하기 및 예제

P248, p251, p257~259, p272