Pipelined MIPS CPU

김정현

1. MIPS CPU

2. Pipelining

3. Data Hazard

4. Control Hazard

5. 결과 및 한계

1. MIPS CPU

1) MIPS CPU

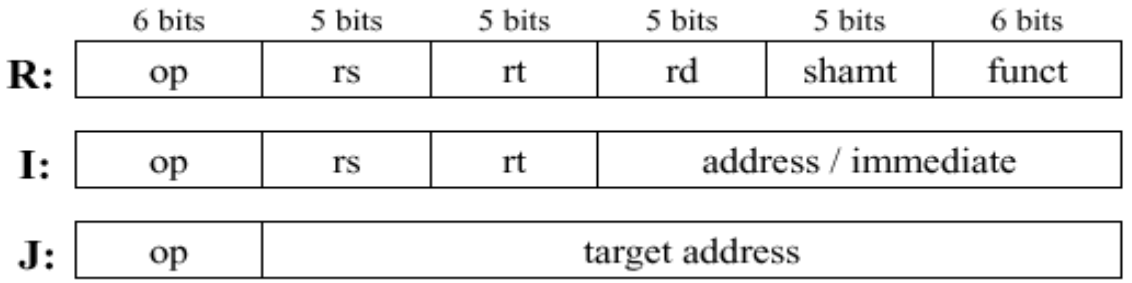
- CPU (Central Processing Unit)
  컴퓨터 시스템을 통제하고 프로그램의 연산을 실행하는 가장 핵심적인 컴퓨터의 제어 장치로써 32-bit나 64-bit로 이루어진 어셈블리어로 된 명령어에 맞게 동작한다.

- MIPS CPU
  MIPS Technology에서 개발한 MIPS Architecture로 만들어진 CPU로써 명령어는 32-bit의 MIPS ISA (Instruction Set Architecture : 명령어 집합 구조)를 따른다.

1. MIPS CPU

2) MIPS ISA (Instruction Set Architecture)

MIPS 명령어 구조는 R, I, J의 3가지 타입으로 구성된다.



- R-type
  이항 연산을 위한 명령어 (ex) 사칙 연산 (+, -, *, /, …), 논리 연산(AND, OR, XOR, …), 대소 비교)

- I-type
  Immediate 연산 (상수 연산), Data memory 접근 (store, load), Branch(분기) 처리 명령어

- J-type
  InstrMmr 상 특정 주소 직접 접근, 함수 실행 관련 명령어

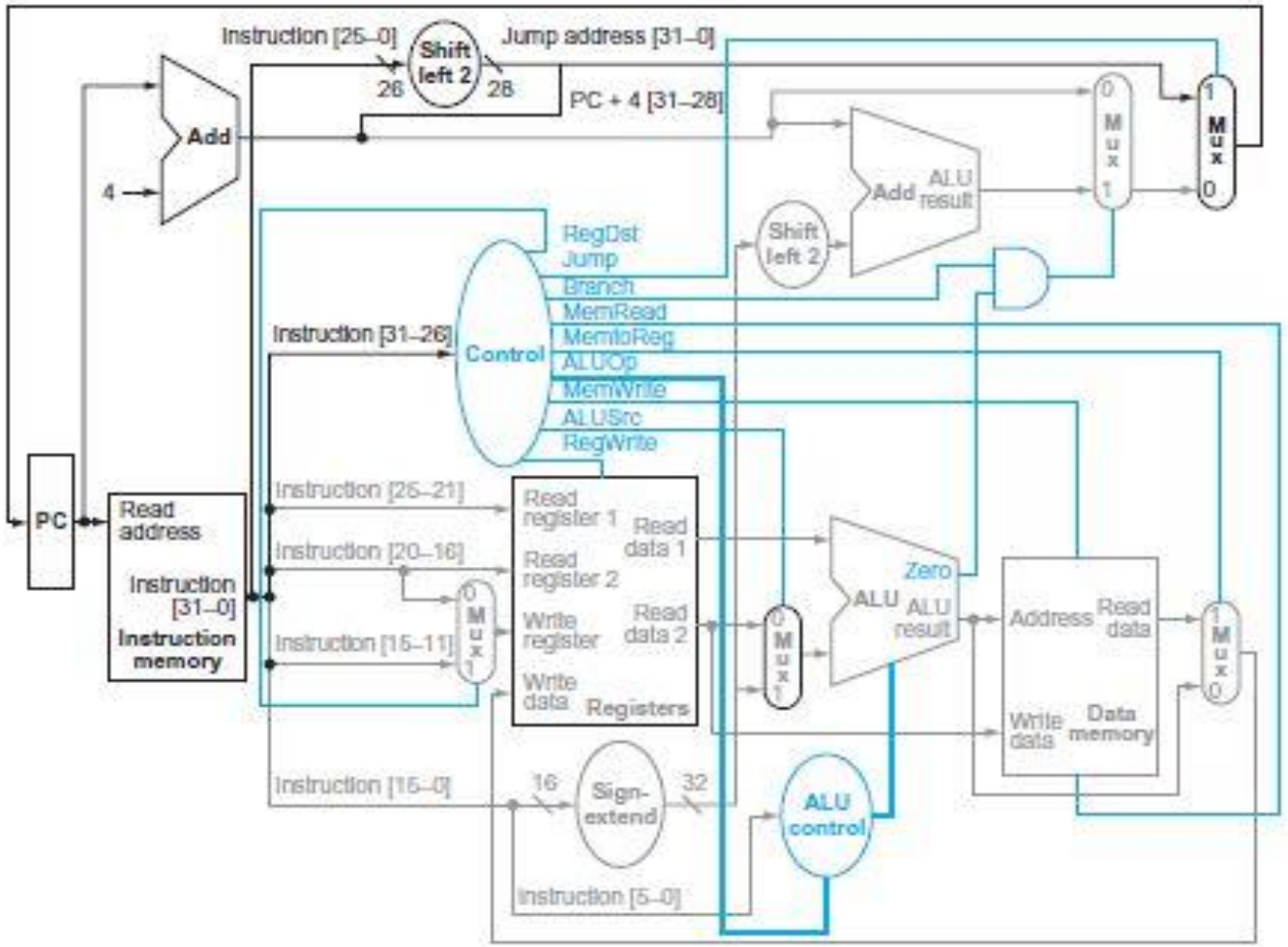
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

1. MIPS CPU

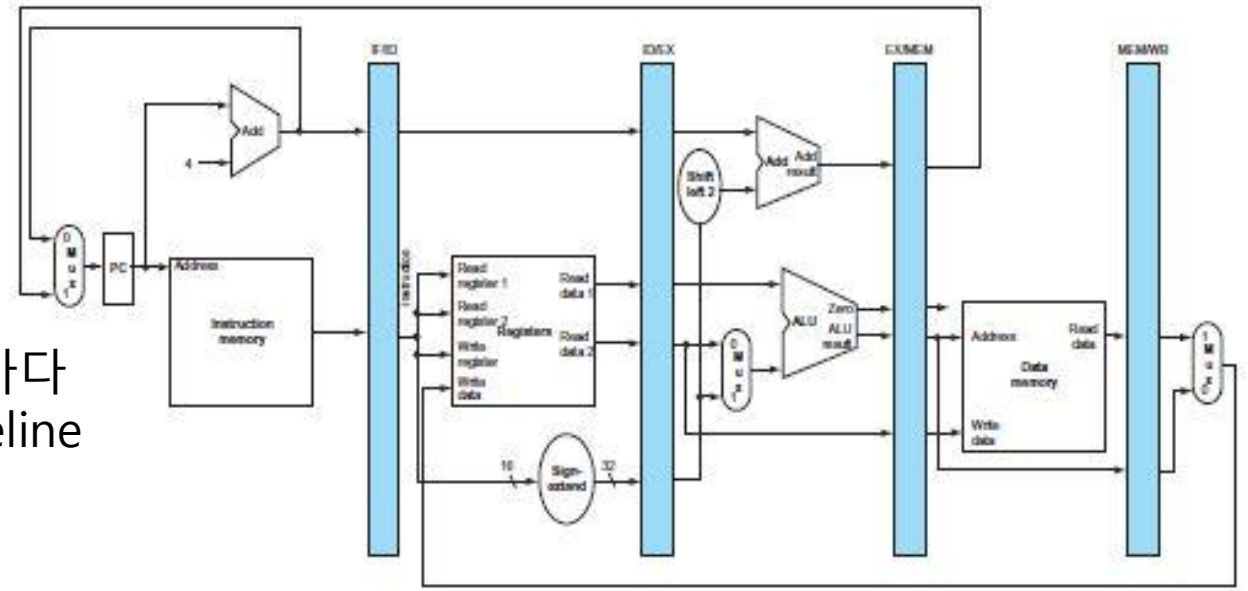
3) Components and Operation

- Instruction Memory : 실행 될 명령어 집합을 가지고 있는 메모리
- PC : InstrMmr의 주소를 출력하여 실행 될 명령어를 선택한다.
- Registers : 32-bit 값을 32개 저장할 수 있는 임시 저장 장치 ($0 ~ $31)
- Control : 명령어(opcode)에 맞게 MUX Selection signal이나 ALU Control signal을 내보낸다.
- ALU : 명령어에 맞는 연산을 수행한다.
- Data Memory : 데이터를 저장하거나 읽어올 수 있는 외부 저장 장치



2. Pipelining

1) Pipelining



CPU를 다섯 부분으로 Buffer를 이용해 나누어 부분마다 각각 다른 명령어를 처리하게 하여 처리 속도를 Pipeline 단계 수에 비례하게 상승시킨다.

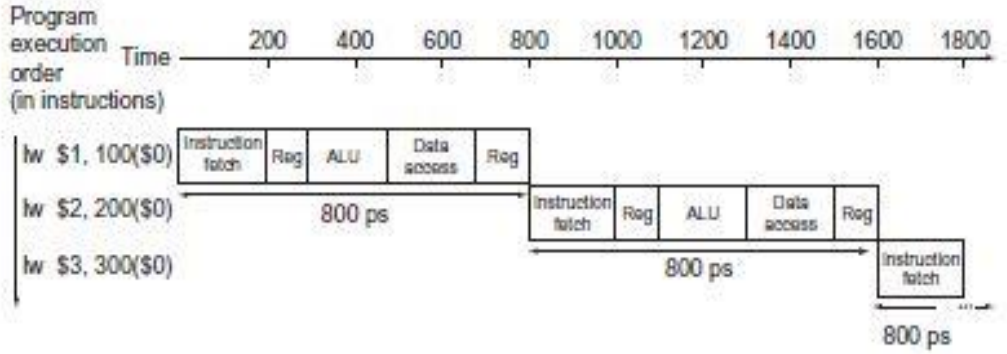
1) IF (Instruction Fetch) : 다음 PC 출력 값에 따라 Instruction memory의 주소를 선택하는 단계

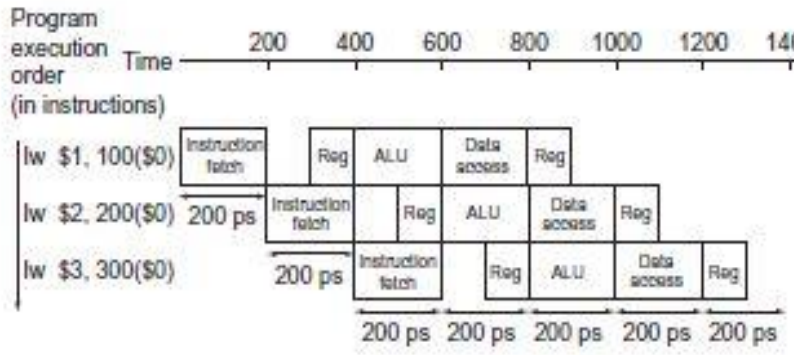
2) ID (Instruction Decode) : Instruction을 해석하여 레지스터 값을 읽는 단계

3) EX (ALU Execution) : ALU 및 주소 연산을 실행하는 단계

4) MEM (Memory Access) : 메모리에 접근하여 처리하는 단계

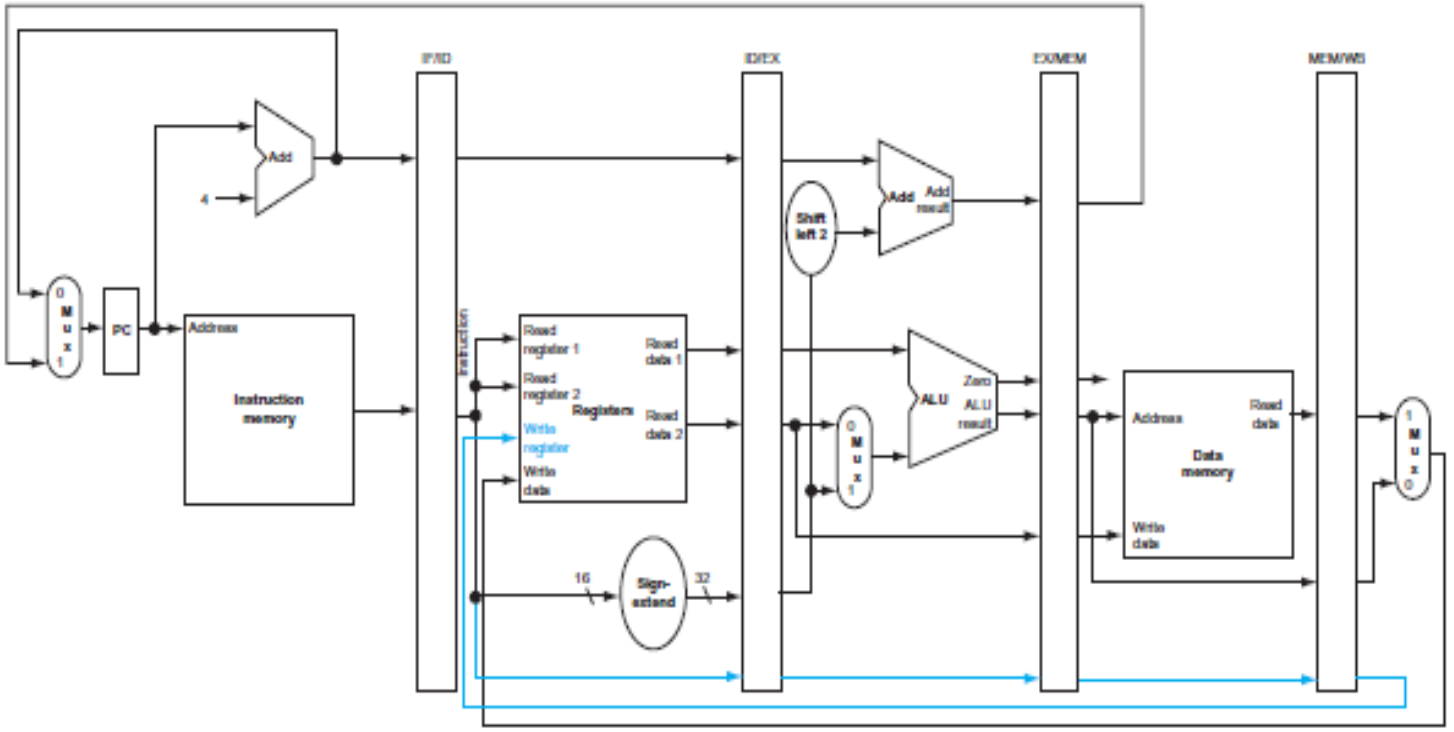
5) WB (Write Back) : 레지스터에 연산 결과 값이나 메모리의 데이터를 쓰는 단계





2. Pipelining

2) Corrected Pipelined Datapath



IF || ID || EX || MEM || WB

- ALU 연산 결과값이나 메모리에서 가져온 데이터를 레지스터에 쓰는 경우에는 쓰여질 레지스터 값이 WB 단계에서 결정되기 때문에 버퍼를 모두 거쳐서 돌아와야 한다.

2. Pipelining

3) Pipelining Control Signals

Control Signal 또한 Pipeline의 명령어 처리 과정과 맞게 Pipelining 되어야 한다.

- EX

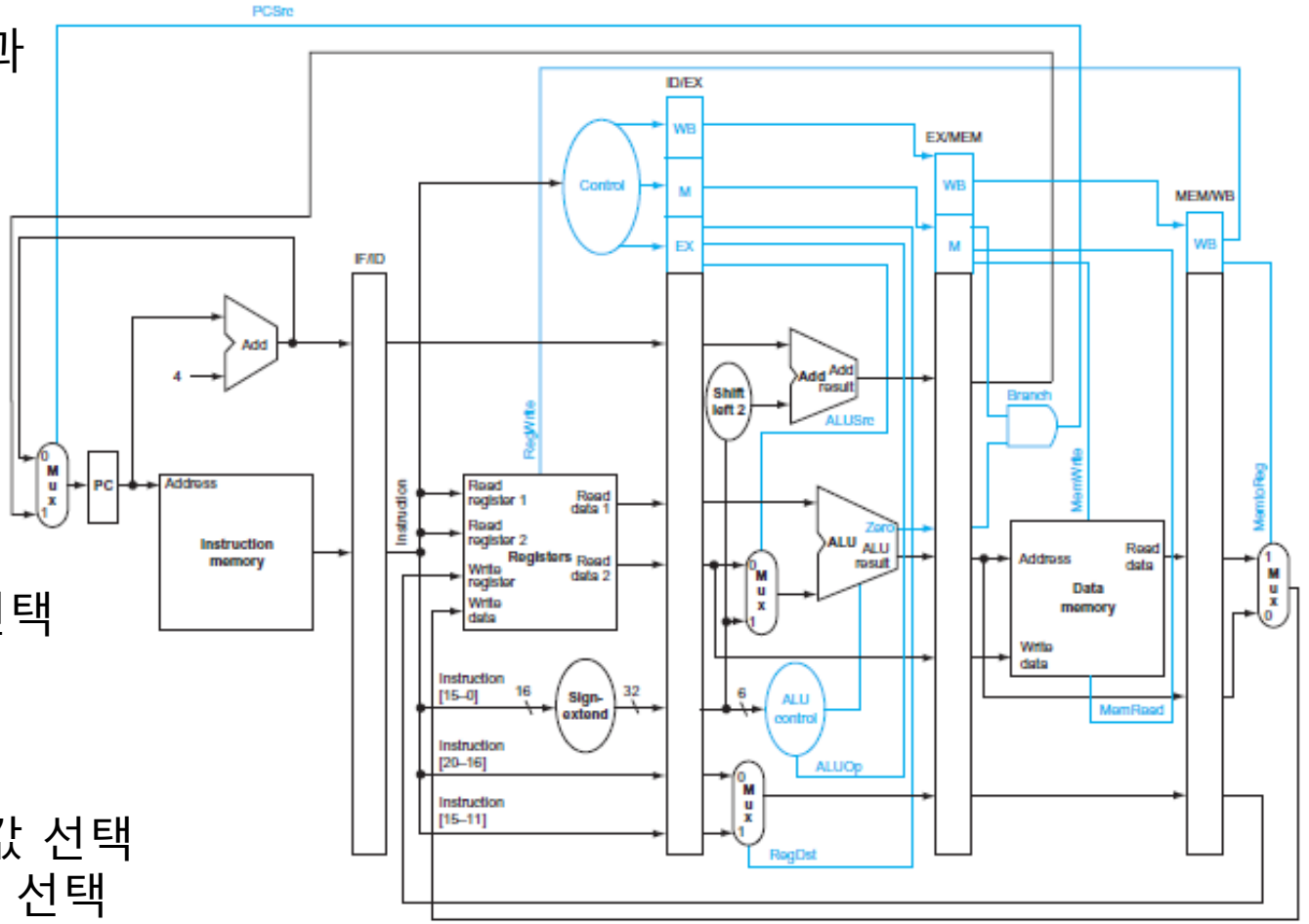
ALUSrc : 레지스터 출력 값 / 명령어 address 선택
ALUOp : ALU의 연산 종류 선택
RegDst : I-type / R-type 연산 선택

- MEM

Branch : 분기 여부 선택
MemWrite / MemRead : Data Memory의 동작 선택

- WB

RegWrite : 레지스터 쓰기 여부 선택
MemToReg : ALU 출력 값 / 데이터 메모리 출력 값 선택
PCSrc : InstrMmr의 바로 다음 주소 / 계산된 주소 선택



2. Pipelining

4) Pipeline Hazards

앞의 처리에도 아직 Pipelining은 수많은 위험요소들을 가지고 있다.

- Data Hazard : 다음의 명령어가 아직 실행되지 않은 그 전의 명령어에 영향을 받을 때 (Data Dependency)

- Control Hazard : 다음에 실행 될 명령어가 그 전의 Branch 명령어 실행이 끝나지 않아 불확실할 때

- Structure Hazard : 한 HW에 여러 곳에서 접근할 때
  → MIPS CPU에선 일어나지 않는다.

3. Data Hazard

1) Data Hazard

Data Hazard 에는 3가지 경우가 존재한다.

1) RAW (Read After Write) : 레지스터에 연산 값을 쓴 후에 읽어야 하는 경우 발생
 ex) Instr_1 : add r1, r2, r3
 Instr_2 : sub r4, r1, r3 → r1에 add 연산 결과가 저장되기도 전에 r1을 읽으면 안 된다.

2) WAR (Write After Read) : 레지스터를 읽은 후에 연산 값을 쓰는 경우 발생
 ex) Instr_1 : sub r4, r1, r3
 Instr_2 : add r1, r2, r3 → 위와 반대의 경우로, 여러 명령어 동시 처리 시 발생 가능하다.

3) WAW (Write After Write) : 레지스터에 연산 값을 쓴 후에 또 써야하는 경우 발생
 ex) Instr_1 : sub r1, r4, r3 → r1에 sub 연산 결과가 add 연산 결과보다 늦게 끝나면 안 되며,
 Instr_2 : add r1, r2, r3 여러 명령어 동시 처리 시 발생 가능하다.

→ 여기서 다룰 Pipelined MIPS CPU는 여러 명령어를 동시에 처리하지 않으니 RAW의 경우만 고려하도록 한다.

3. Data Hazard

1) Data Hazard

- 여러 Data Hazard 해결 방법

1. Stall : Data Hazard 발생 시 한 클럭 동안 아무 동작을 하지 않음으로써 앞의 연산이 끝나길 기다린다.
   → Performance 손실 발생

2. Pipeline Scheduling : 처음부터 Data Hazard가 일어날 명령어 사이에 Dependency가 없는 다른 명령어를 넣어 Hazard를 방지한다.
   → 근본적인 해결 불가

3. Forwarding : Data Hazard 발생 시 앞의 명령어 연산 값을 직접 가져온다.
   → Performance 손실없이 Hazard에 즉각 대응 가능

3. Data Hazard

2) Forwarding Unit

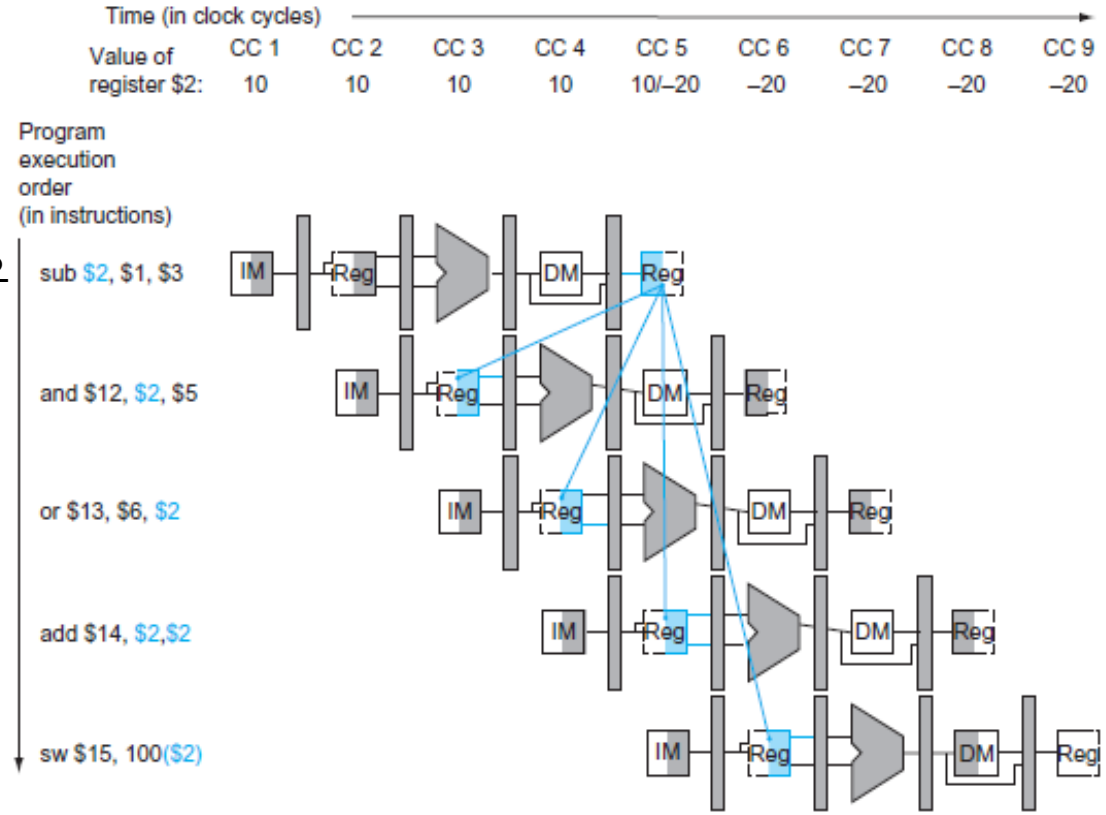
Forwarding : Data Hazard 발생 시 다음 단계의 값을 앞으로 가져온다.

- Data Hazard 발생 시 앞 명령어의 저장 레지스터(rd), 뒤 명령어의 연산 레지스터(rs, rt)가 같다는 사실을 알 수 있다.

- EX 연산 시 Dependency가 MEM, WB 단계에 있는 경우 각각이 있음을 알 수 있다.

그러므로 4가지 경우가 존재한다.

1a. EX/MEM.rd = ID/EX.rs
1b. EX/MEM.rd = ID/EX.rt
2a. MEM/WB.rd = ID/EX.rs
2b. MEM/WB.rd = ID/EX.rt



→ 이 각각의 경우에 대응하여 뒤의 값을 앞으로 가져올 Forwarding Unit 설계 요구!

3. Data Hazard

2) Forwarding Unit

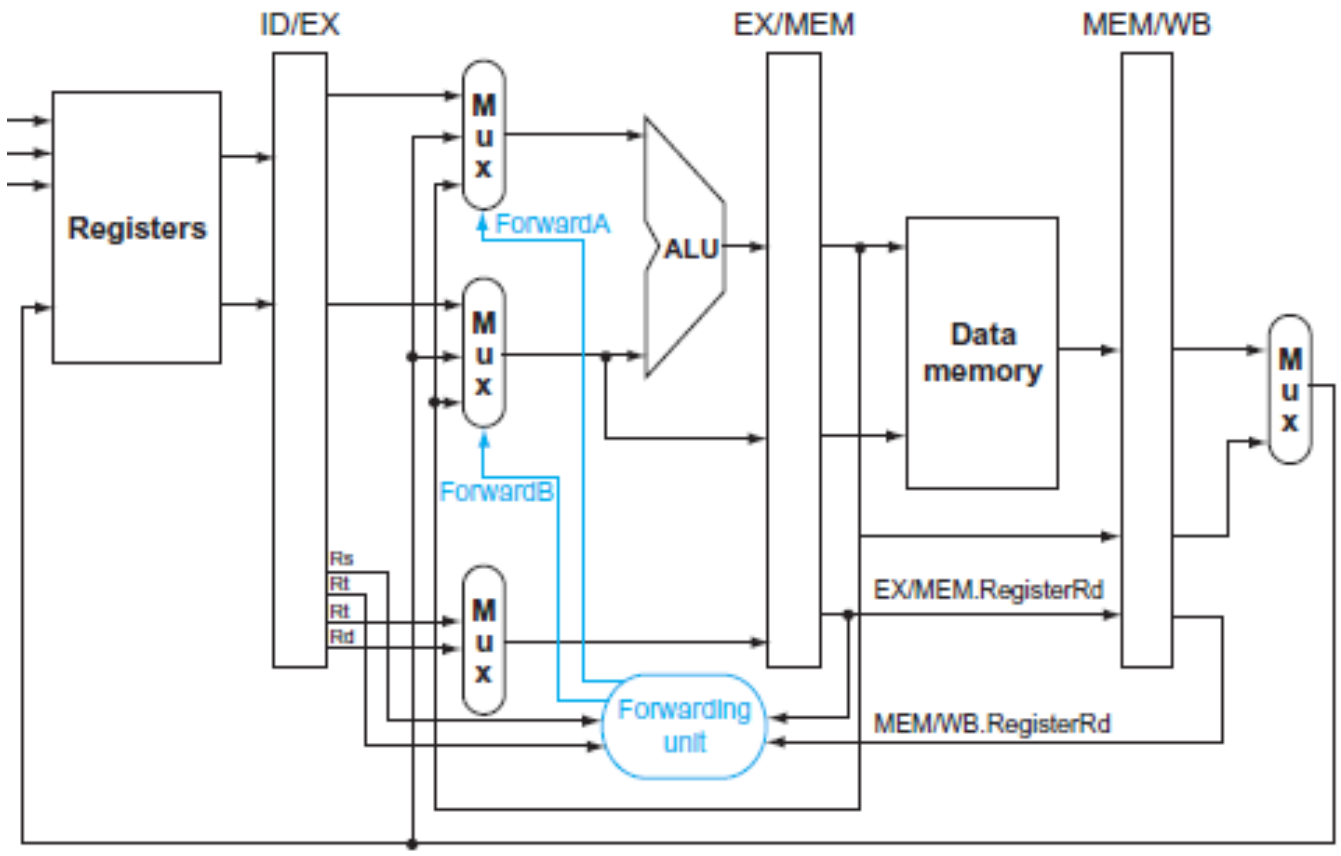
Forwarding Unit을 설계하여 Forwarding을 수행한다.

- 입력으로 비교해야 할 4개의 값을 받는다. (ID/EX.rs, .rd, EX/MEM.rd, MEM/WB.rd)

- 출력으로 MUX selection signal을 보낸다.

- Forwarding이 필요 없을 시 레지스터 값을, MEM 단계 값을 Forwarding 요구 시 그 값을, WB 단계 값을 Forwarding 요구 시 그 값을 선택하여 연산을 진행한다.

- Forwarding이 MEM 단계, WB 단계 둘 다 요구될 경우 MEM 단계의 Forwarding을 선택한다.

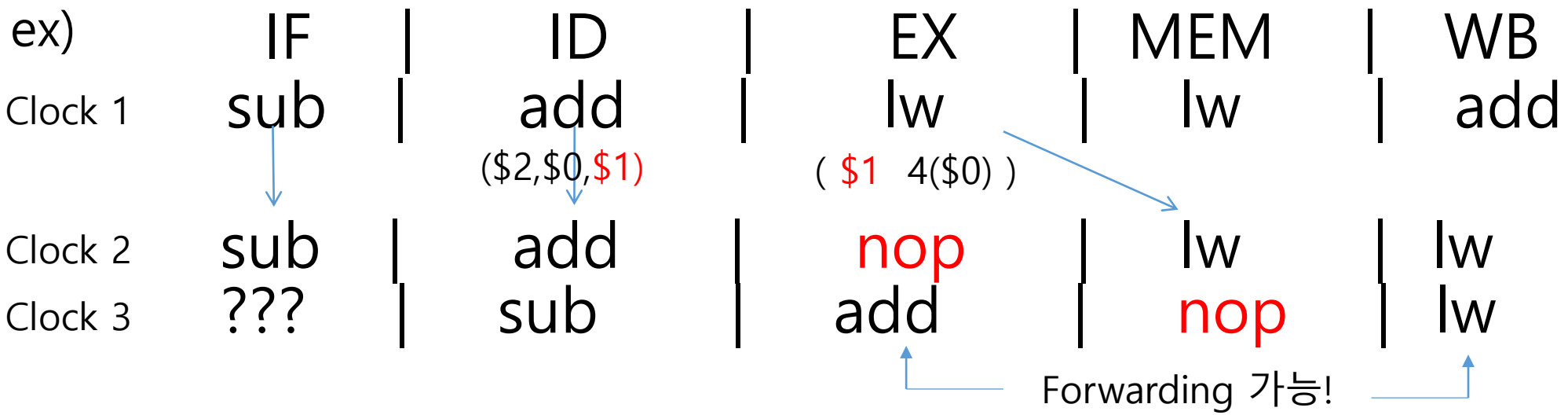


3. Data Hazard

3) Load-use Data Hazard

- Load 명령어의 경우 Hazard 발생 시 Data Memory에서 값을 읽어오므로 한 클럭을 쉬어 주어야 하며 Forwarding으로 해결 불가능하다.

→ Load 명령어 실행 중 Hazard가 발생하면 Load 명령어 바로 앞에 NOP (No Operation : 아무 동작도 안하는 명령어)를 넣어줄 수 있도록 한다.



NOP를 넣어줄 수 있는 Hazard Detection Unit 필요!

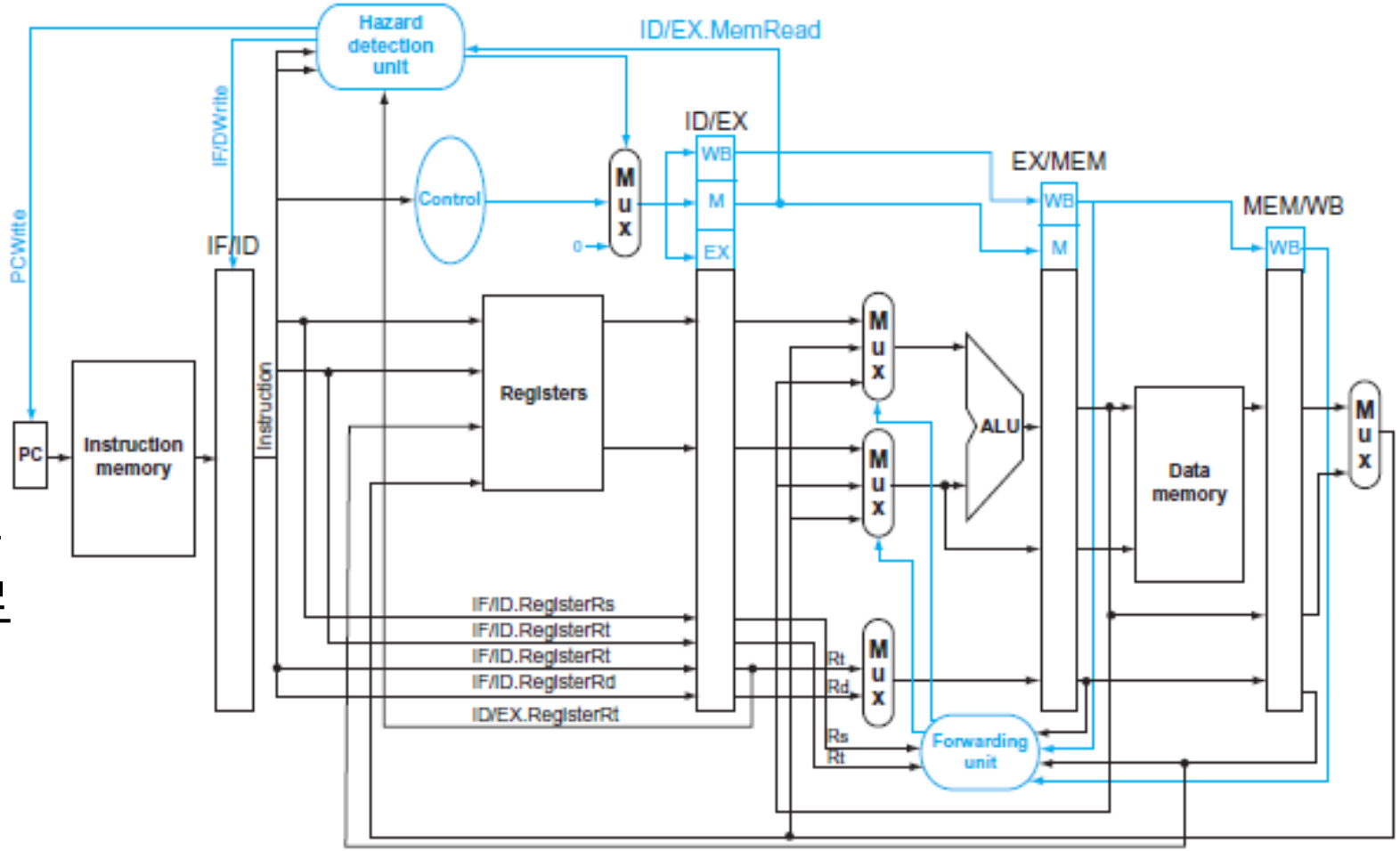
3. Data Hazard

4) Hazard Detection Unit

- ID, EX 단계에서 발생

- ID/EX.rt, IF/ID.rs, .rt 값으로 Dependency 판별

- MemRead가 1이면 Data memory에서 값을 읽는다는 뜻이므로 ID/EX.MemRead로 Load 명령어 판별



3. Data Hazard

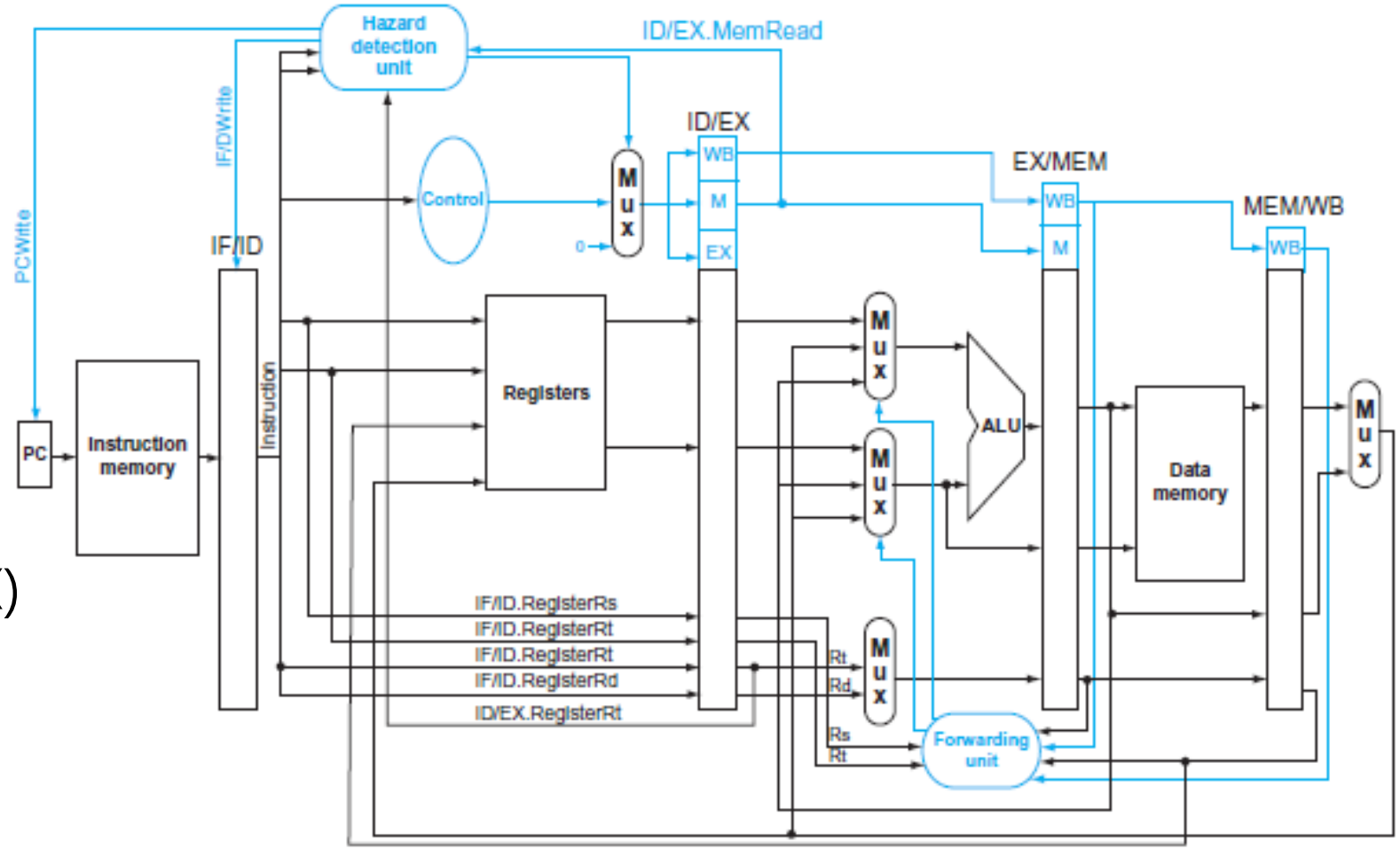
4) Hazard Detection Unit

EX에 NOP를 넣는 방법

- PC 동결 (PCWrite)

- IF/ID 동결 (IF/IDWrite)

- Control signal 을
전부 0으로 보낸다. (MUX)
→ NOP



4. Control Hazard

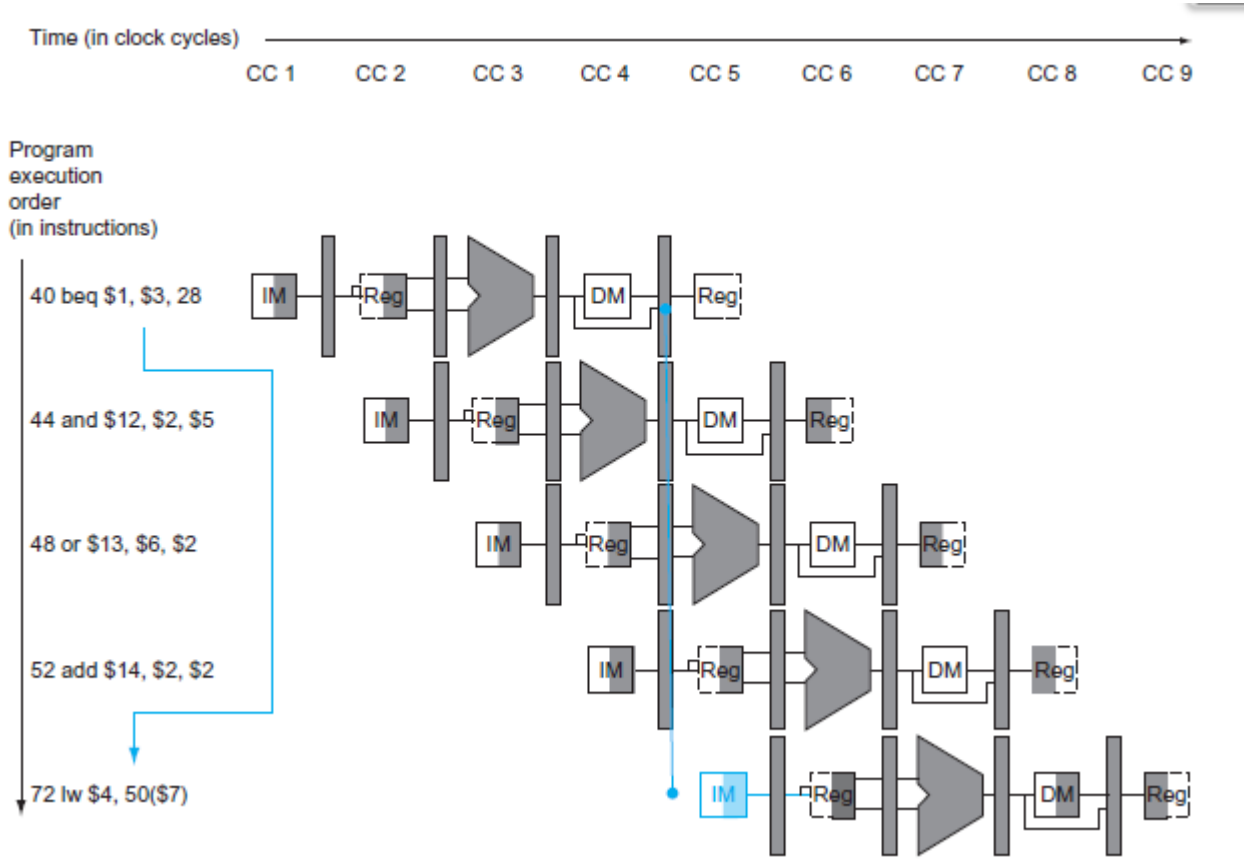
1) Control Hazard

- Control Hazard : Branch 명령어 처리가 끝나지 않아 다음 명령어가 불확실한 상태

- Branch 명령어의 경우 그 결과가 MEM 단계에서 나오기 때문에 Branch가 실행 될 경우 3단계의 명령어를 버려야 (FLUSH) 한다.

- beq (Branch Equal) : rs와 rd의 레지스터가 같은 값을 가지면 Instruction memory 상의 지정 주소로 분기한다.

- Branch Prediction : Branch의 결과를 미리 예상해 실행한 후 Branch 명령어 처리가 끝나 결과가 나왔을 때 처리한다.



4. Control Hazard

2) Branch Prediction

- Branch Prediction : Control Hazard를 해결하기 위해 Branch 명령어 결과를 예측한다.
  → 분기 예상이 실패할 경우 FLUSH를 하여 Pipeline을 비워야 한다.

1. Branch Not Taken : 분기가 일어나지 않을 것이라 예상하고 실행

2. Branch Taken : 분기가 무조건 일어날 것이라 예상하고 실행

3. Execute Both Paths : HW를 2배로 늘려 두 경우 모두 실행 후 선택

4. Taken Backwards Not Taken Forwards : 분기가 앞으로 가는 경우 일어날 것이라 예상,
   뒤로 가는 경우 일어나지 않을 것이라 예상
   → 분기가 앞으로 가는 경우의 대부분 반복문이 원인이므로 적중률이 훨씬 높아진다.

5. Profile-based Prediction : 이전의 Branch 명령어 실행 결과를 기록하여 그에 따라 예상

Branch Not Taken으로 설계 후 FLUSH 단계를 줄이는 방향으로 설계하였다.

4. Control Hazard

3) Reducing FLUSH

- 분기 명령어의 처리를 MEM 단계가 아닌 ID 단계에서 하면 FLUSH 할 단계가 3 → 1단계로 줄어든다!

1. Branch 명령어의 Target Address 처리
   1) PCSrc, JToPC (Control Signal)을 다음 단계로 보내지 않는다.
   2) 모든 주소 값 연산을 ID 단계에서 끝낸다.

2. Branch 명령어의 실행 여부
   1) Equality Unit 추가로 레지스터 값의 비교 결과를 ID 단계에서 바로 체크
   2) ALU의 zero 출력 대신 Unit의 출력 사용 (PCSrc = Branch & zero)

3. Branch 명령어일 경우 바로 앞(IF/ID)에 NOP 삽입
   IF/ID에서 나오는 명령어를 전부 0 처리

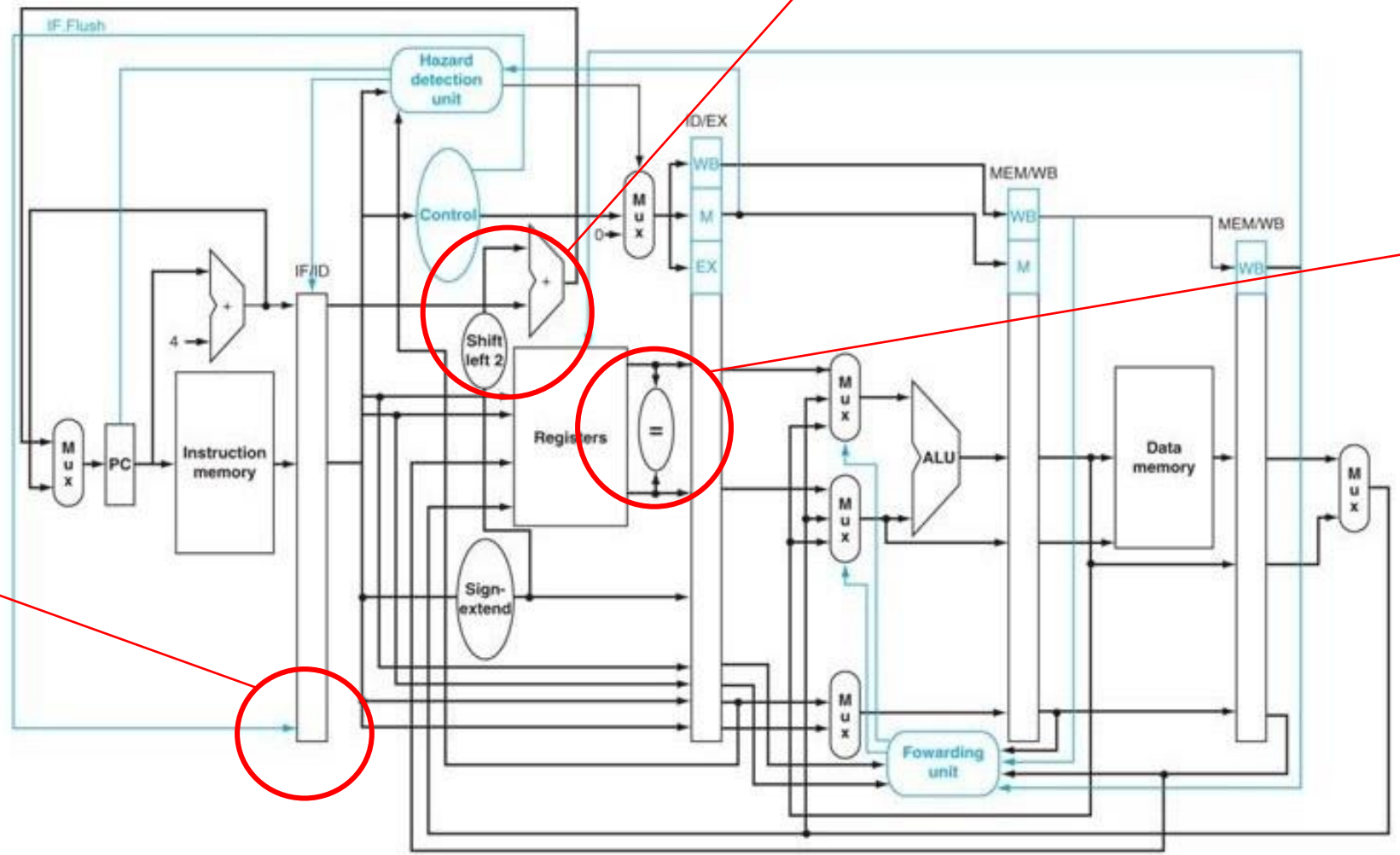
4. Control Hazard

3) Reducing FLUSH

1. Target address 처리

2. Equality Unit

3. NOP 처리



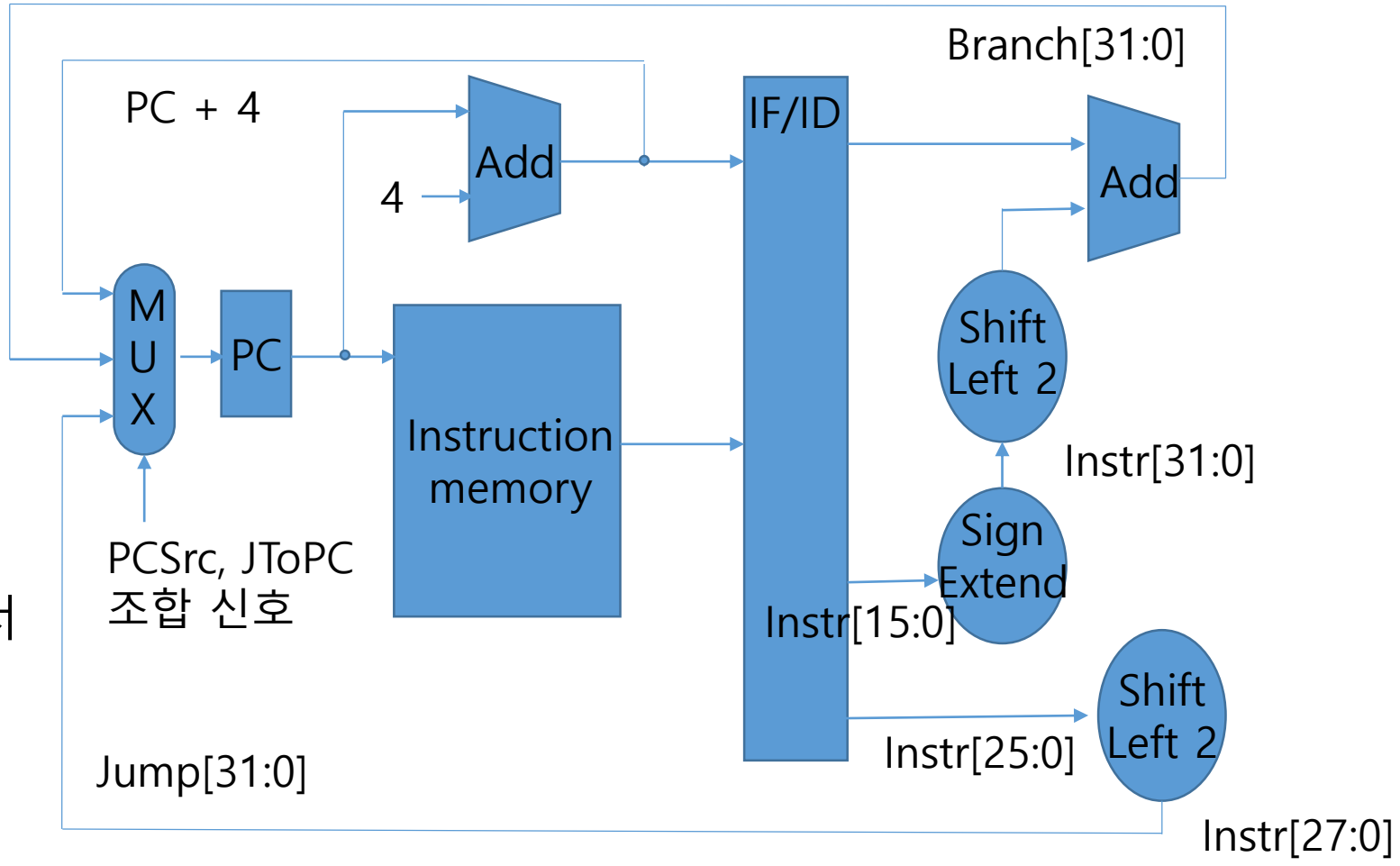
4. Control Hazard

3) Reducing FLUSH

1. Branch 명령어의 Target Address 처리

- Control Signal인 PCSrc (Branch인지 결정), JToPC (Jump인지 결정) 을 조합한 2-bit 신호로 MUX 출력을 결정한다.

- MEM 단계에서 이루어지던 주소 계산을 모두 ID 단계에서 끝낸다.



4. Control Hazard

3) Reducing FLUSH

2. IF Flush 처리

ex)

IF	ID	EX	MEM	WB
j	beq	slt	xor	and

beq가 ID 단계에서 Taken으로 확인 되었을 때

lw	nop	beq	slt	xor

beq 주소의 명령어 Fetch

beq 다음 Flush

- Branch 명령어가 판별될 때에도 Instruction memory에서 다음 명령어가 나오기 때문에 Branch Taken일 경우 그 명령어를 NOP 처리해주어야 한다.

→ Control Unit 에서 Branch 여부 확인 후 IF.Flush 라는 새로운 Control Signal을 IF/ID로 보내어 NOP를 만들 수 있도록 한다.

4. Control Hazard

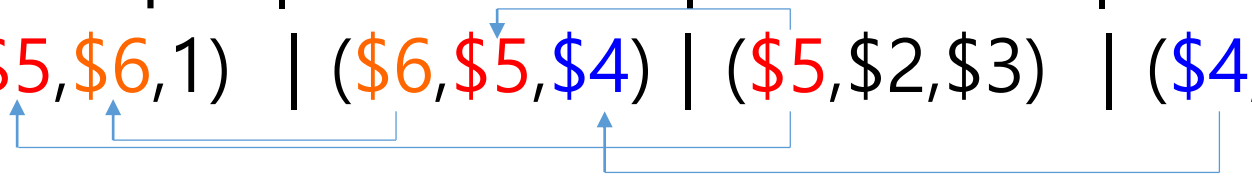
3) Reducing FLUSH

3. Equality Unit : Control Hazard + Data Hazard

- ID 단계에서의 Equality Unit의 동작은 Data Hazard를 일으킬 수 있다. → ID 단계에서의 Forwarding 요구

ex)

IF	ID	EX	MEM	WB
j	beq	slt	xor	and
	($5,$6,1)	($6,$5,$4)	($5,$2,$3)	($4,$2,$3)



- Forwarding은 EX, MEM 단계에서, rt, rs에서 발생하므로 앞의 경우와 마찬가지로 4가지 경우가 존재, 앞과 같은 방식으로 또 다른 Forwarding Unit을 설계한다.

→ Branch Equal Forwarding Unit

4. Control Hazard

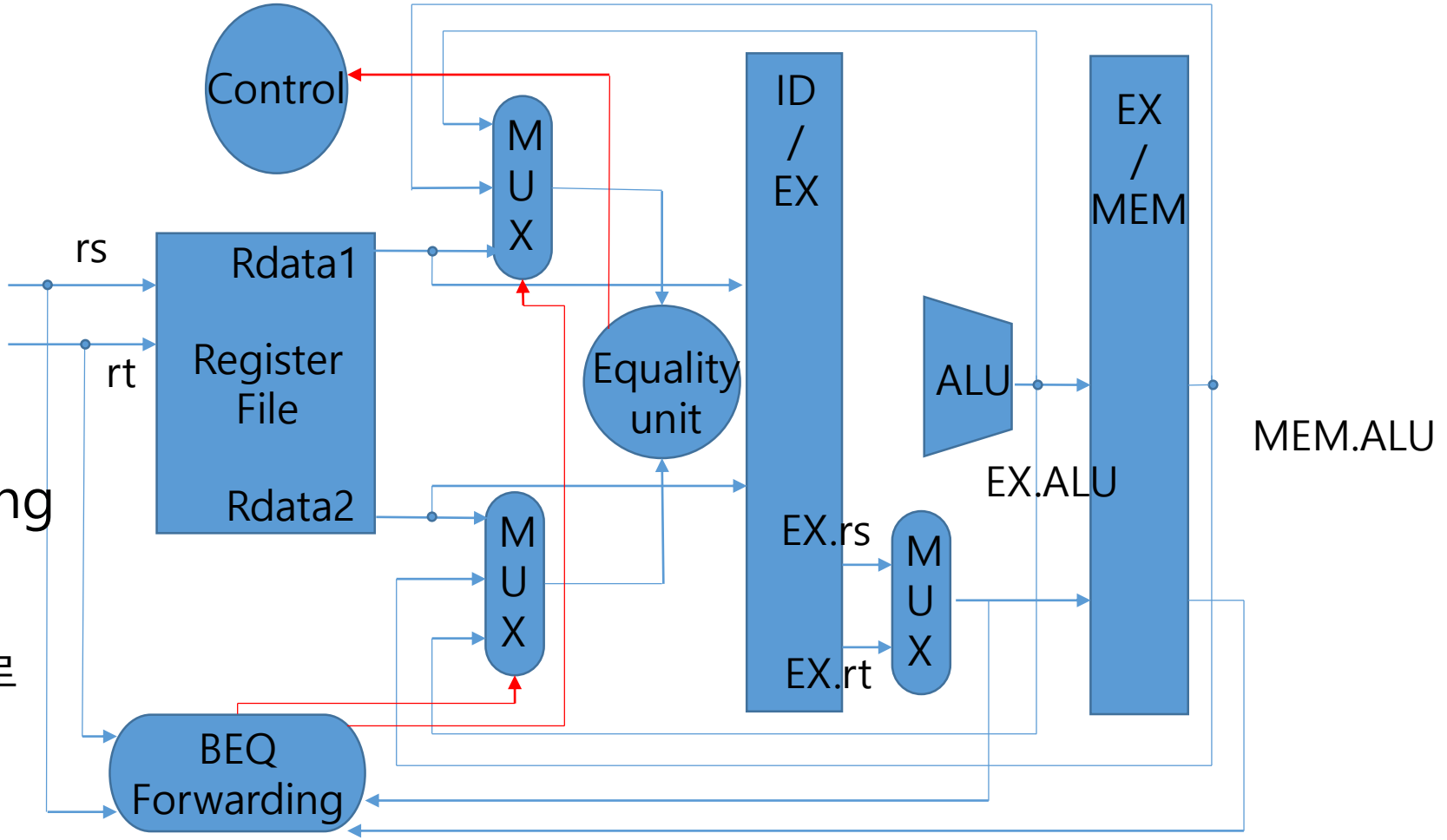
4) Equality Unit / Branch Equal Forwarding

- Equality Unit

MUX로 선택된 값을 비교하여 Control Unit에 결과를 보낸다. 그에 따라 Control Unit이 IF.Flush 신호를 IF/ID로 보내어 NOP를 생성한다.

- Branch Equal Forwarding

앞의 Forwarding과 마찬가지로 EX, MEM 단계의 값을 ID 단계로 가져온다.



5. 결과 및 한계

1) Simulation Result

```
lw s0 0($zero)

lw s1 4($zero)

add s2 s0 s1
sub s3 s0 s1
and s4 s2 s3
xor s5 s2 s3
slt s6 s5 s4
beq s5 s1 1
j 13
lw s7 8($zero)
sw s5 16(s7)
lw s0 16(s7)
j 3
nor s7 s0 $zero
lw t0 40($zero)
lw t1 44($zero)
add t2 t0 t1
sub t3 t0 t1
and t4 t0 t1
xor t5 t0 t1
slt t6 t2 t3
```

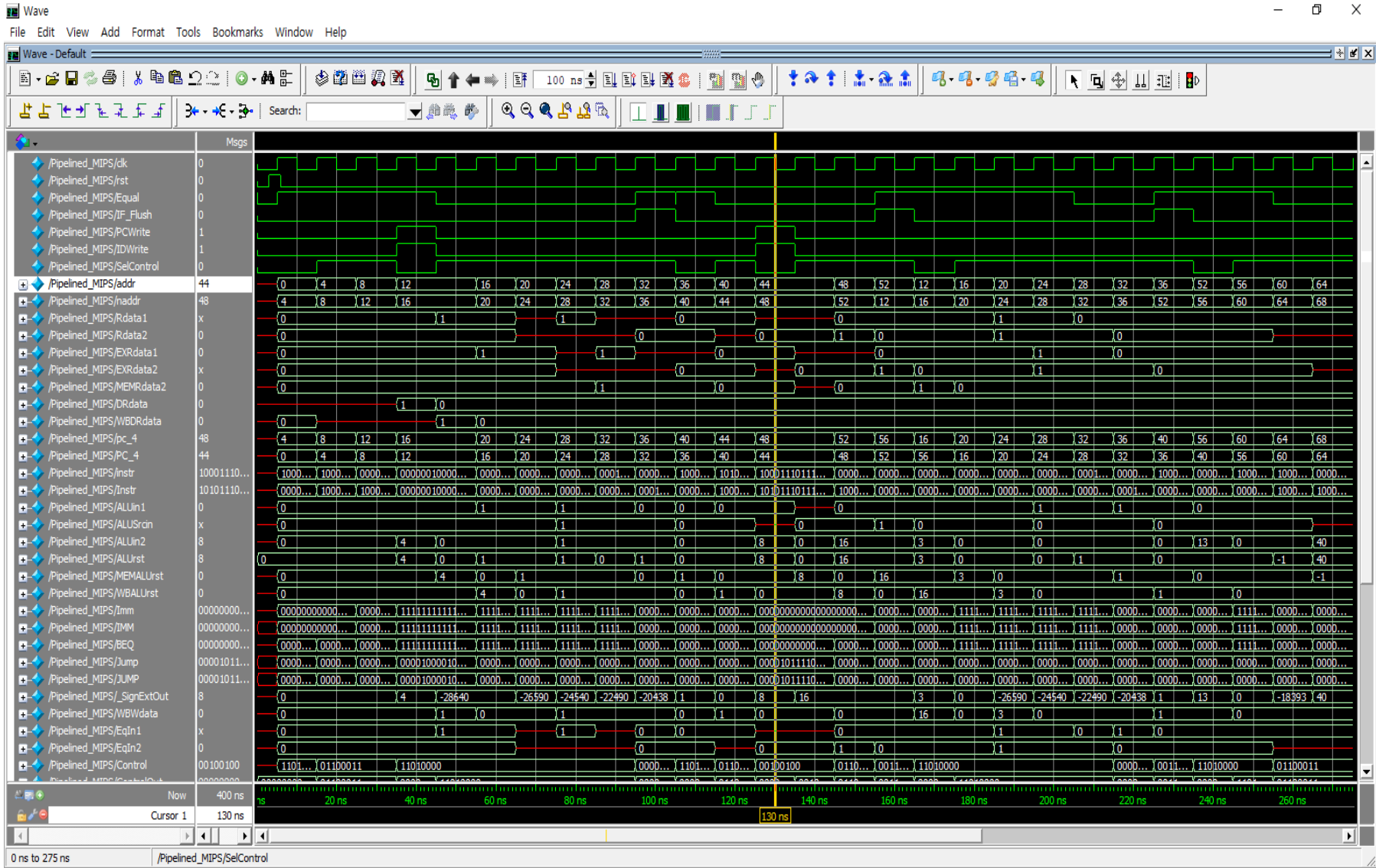
아래의 명령어들을 Instruction memory에 입력한 후 ModelSim 으로 Simulation 하였다.

```
mem[0] = 32'b100011_00000_10000_00000_00000_000000; // lw s0 0($zero) , s0 = 1
mem[1] = 32'b100011_00000_10001_00000_00000_000100; // lw s1 4($zero) , s1 = 0
mem[2] = 32'b000000_10000_10001_10010_00000_100000; // add s2 s0 s1 , s2 = 1
mem[3] = 32'b000000_10000_10001_10011_00000_100010; // sub s3 s0 s1 , s3 = 1
mem[4] = 32'b000000_10010_10011_10100_00000_100100; // and s4 s2 s3 , s4 = 1
mem[5] = 32'b000000_10010_10011_10101_00000_100110; // xor s5 s2 s3 , s5 = 0
mem[6] = 32'b000000_10101_10100_10110_00000_101010; // slt s6 s5 s4 , s6 = 1
mem[7] = 32'b000100_10101_10001_00000_00000_000001; // beq s5 s1 1 , s5 = s1 = 0
mem[8] = 32'b000010_00000_00000_00000_00000_001101; // j 13 , it is not executed at first, but is executed after the jump. (dmem[12])
mem[9] = 32'b100011_00000_10111_00000_00000_001000; // lw s7 8($zero) , s7 = 0
mem[10]= 32'b101011_10111_10101_00000_00000_010000; // sw s5 16(s7) , dmem[4] = 0
mem[11]= 32'b100011_10111_10000_00000_00000_010000; // lw s0 16(s7) , s0 = dmem[4] = 0
mem[12]= 32'b000010_00000_00000_00000_00000_000011; // j 3
mem[13]= 32'b000000_10000_00000_10111_00000_100111; // nor s7 s0 $zero , s7 = 32'b1111_1111_1111_1111_1111_1111_1111_1111;
mem[14]= 32'b100011_00000_01000_00000_00000_101000; // lw t0 40($zero) , t0 = 32'b0101_0101_1010_1010_0101_0101_1010_1010;
mem[15]= 32'b100011_00000_01001_00000_00000_101100; // lw t1 44($zero) , t1 = 32'b0111_0111_1000_1000_0111_0111_1000_1000;
mem[16]= 32'b000000_01000_01001_01010_00000_100000; // add t2 t0 t1 , t2 = 32'b1100_1101_0011_0010_1100_1101_0011_0010; positive value (overflow)
mem[17]= 32'b000000_01000_01001_01011_00000_100010; // sub t3 t0 t1 , t3 = 32'b1101_1110_0010_0001_1101_1110_0010_0010; negative value
mem[18]= 32'b000000_01000_01001_01100_00000_100100; // and t4 t0 t1 , t4 = 32'b0101_0101_1000_1000_0101_0101_1000_1000;
mem[19]= 32'b000000_01000_01001_01101_00000_100110; // xor t5 t0 t1 , t5 = 32'b0010_0010_0010_0010_0010_0010_0010_0010;
mem[20]= 32'b000000_01010_01011_01110_00000_101010; // slt t6 t2 t3 , t6 = 32'd1;
```

5. 결과 및 한계

1) Simulation Result

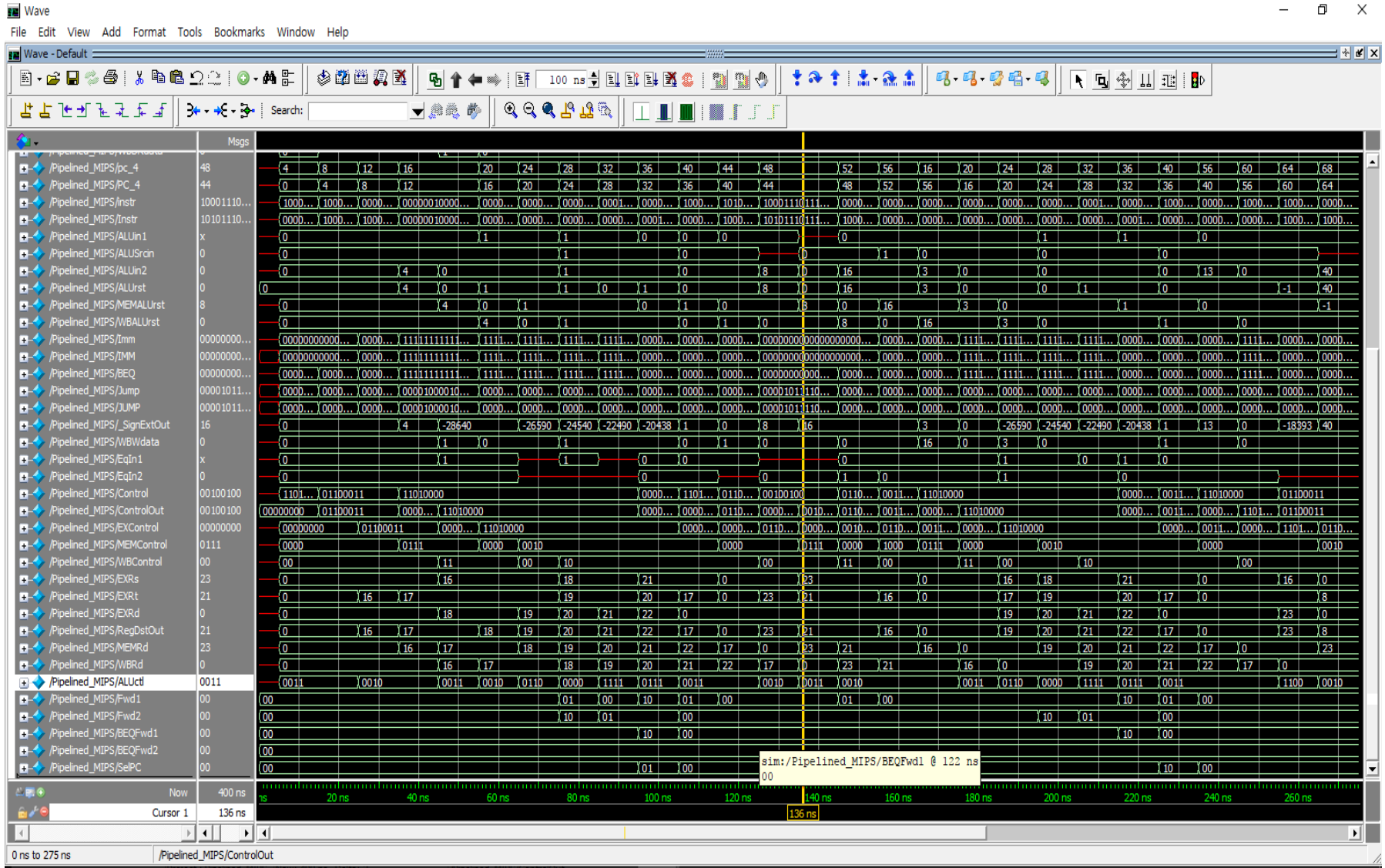
제대로 동작하여 예상대로 j 명령어 연산이 잘 수행되었음을 PC의 출력 값인 Addr 값을 통해 확인할 수 있다.



5. 결과 및 한계

1) Simulation Result

Rs, Rt, Rd 값을 통해 어떤 레지스터의 연산이 진행되는지 확인할 수 있다.



5. 결과 및 한계

2) Limitation

- 더 나은 Branch Prediction 알고리즘 존재

- 실제 입출력을 고려하지 않음

- 인터럽트를 고려하지 않음

- 시뮬레이션의 한계