Design
of Pipelined MIPS Processor

Sept. 26, 1996

Topics

- Instruction processing
- Principals of pipelining
- Inserting pipe registers
- Data Hazards
- Control Hazards

MIPS architecture subset

R-type instructions (add, sub, and, or, slt): rd <-- rs funct rt

0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

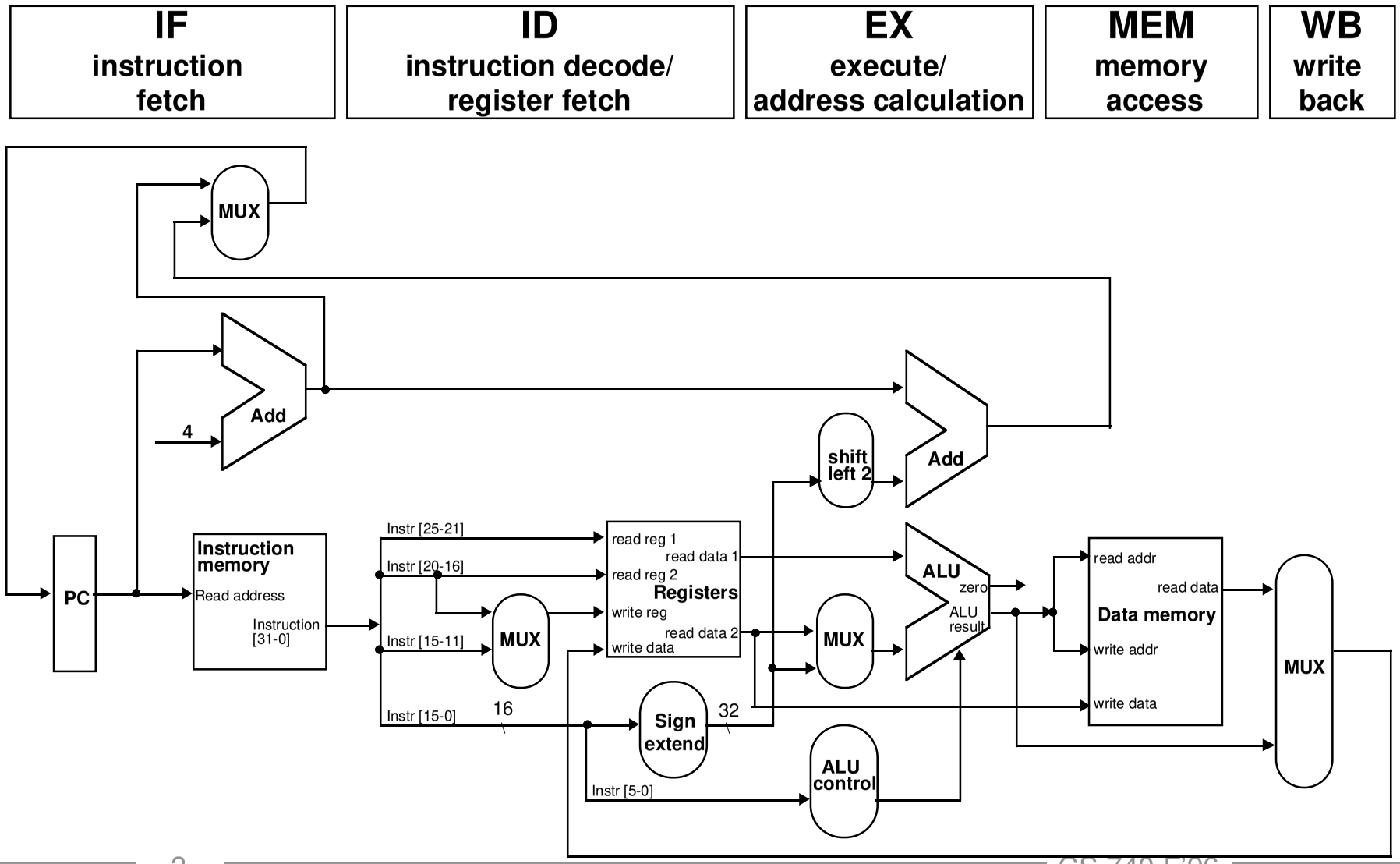
Load: rt <-- Mem[rs + I16] **Store:** Mem[rs + I16] <-- rt

35 or 43	rs	rt	I16
31-26	25-21	20-16	15-0

Branch equal: PC <-- (rs == rt) ? PC + 4 + I16 <<2 : PC + 4

4	rs	rt	I16
31-26	25-21	20-16	15-0

Datapath


IF
instruction fetch
ID
instruction decode/ register fetch
EX
execute/ address calculation
MEM
memory access
WB
write back
MUX
Add
4
PC
Instruction memory
Read address
Instruction [31-0]
Instr [25-21]
Instr [20-16]
Instr [15-11]
Instr [15-0]
Instr [5-0]
read reg 1
read reg 2
Registers
write reg
write data
read data 1
read data 2
16
Sign extend
32
shift left 2
Add
ALU
zero
ALU result
read addr
read data
Data memory
write addr
write data
ALU control
MUX

R-type instructions

IF: Instruction fetch

- IR <-- IMemory[PC]
- PC <-- PC + 4

ID: Instruction decode/register fetch

- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

Ex: Execute

- ALUOutput <-- A op B

MEM: Memory

- nop

WB: Write back

- Register[IR[15:11]] <-- ALUOutput

Load instruction

IF: Instruction fetch

- IR <-- IMemory[PC]
- PC <-- PC + 4

ID: Instruction decode/register fetch

- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

Ex: Execute

- ALUOutput <-- A + SignExtend(IR[15:0])

MEM: Memory

- Mem-Data <-- DMemory[ALUOutput]

WB: Write back

- Register[IR[20:16]] <-- Mem-Data

Store instruction

IF: Instruction fetch

- IR <-- IMemory[PC]
- PC <-- PC + 4

ID: Instruction decode/register fetch

- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

Ex: Execute

- ALUOutput <-- A + SignExtend(IR[15:0])

MEM: Memory

- DMemory[ALUOutput] <-- B

WB: Write back

- nop

Branch on equal

IF: Instruction fetch

- IR <-- IMemory[PC]
- PC <-- PC + 4

ID: Instruction decode/register fetch

- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

Ex: Execute

- Target <-- PC + SignExtend(IR[15:0]) << 2
- Z <-- (A - B == 0)

MEM: Memory

- If (Z) PC <-- target

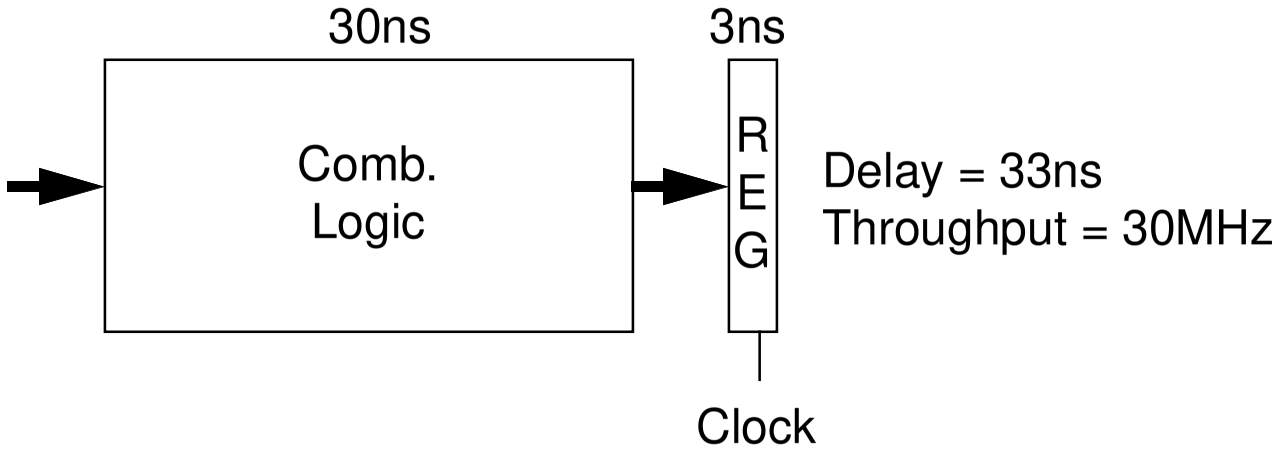
WB: Write back

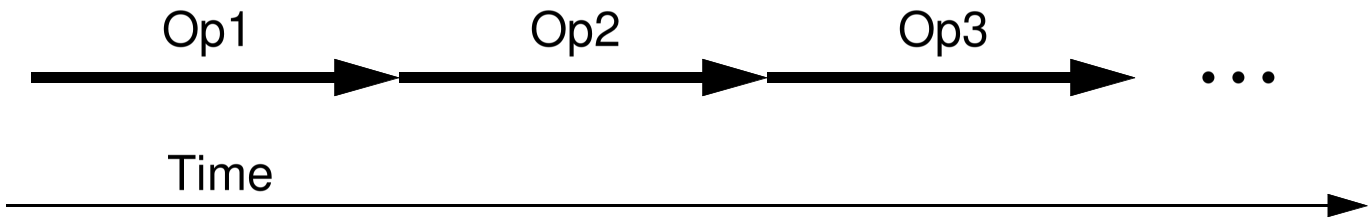
- nop

Is this a delayed branch?

Pipelining Basics

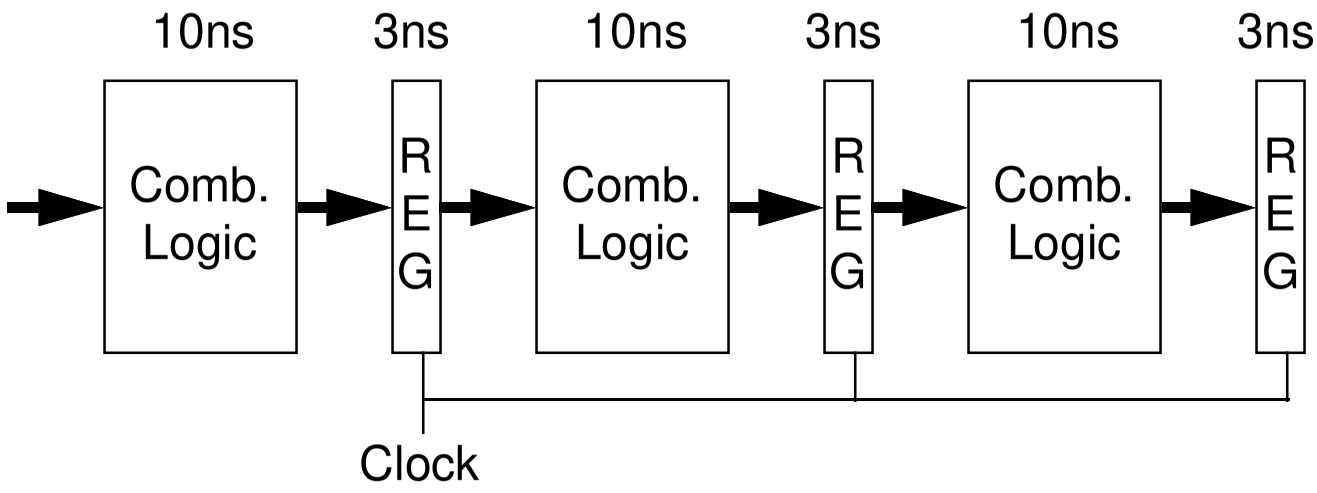
Unpipelined System

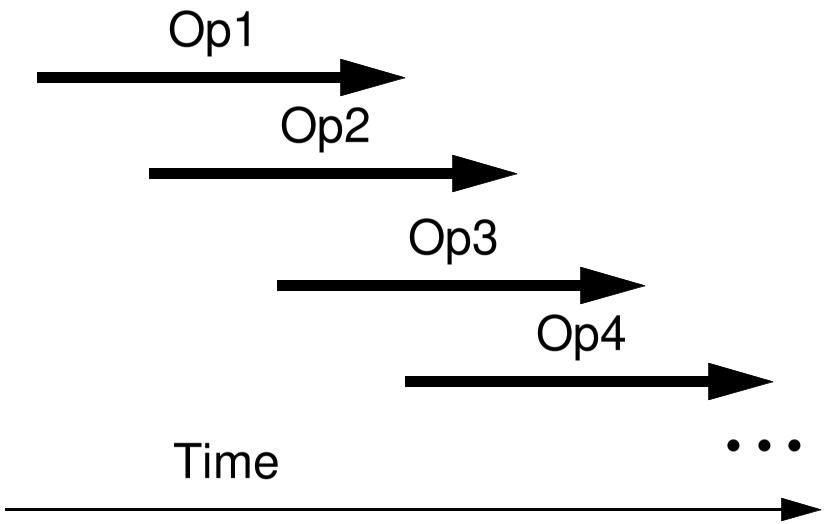




- **One operation must complete before next can begin**
- **Operations spaced 33ns apart**

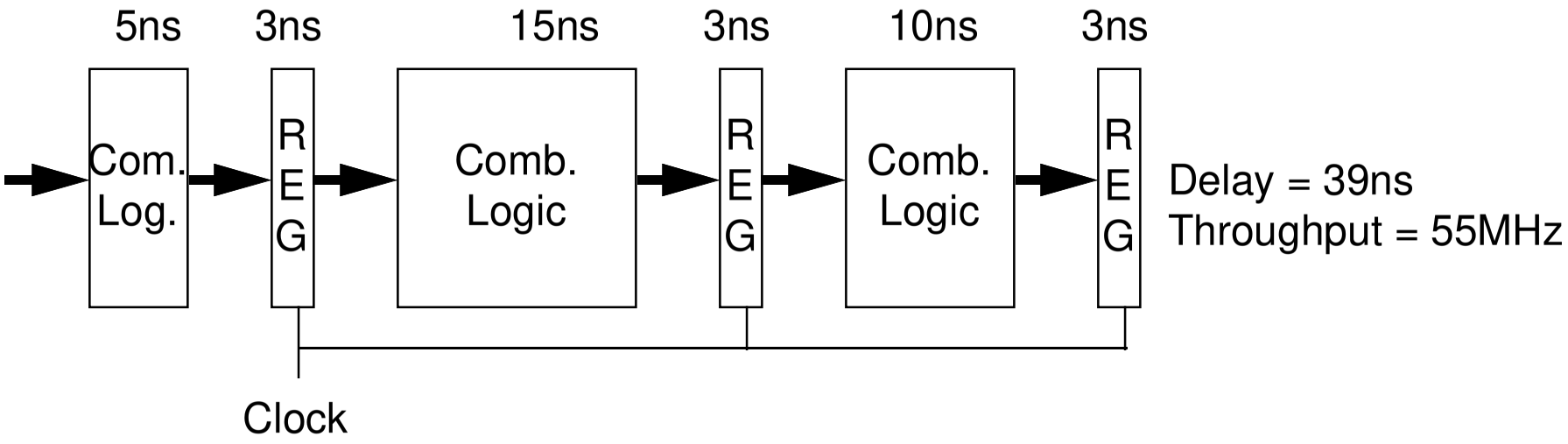
3 Stage Pipelining





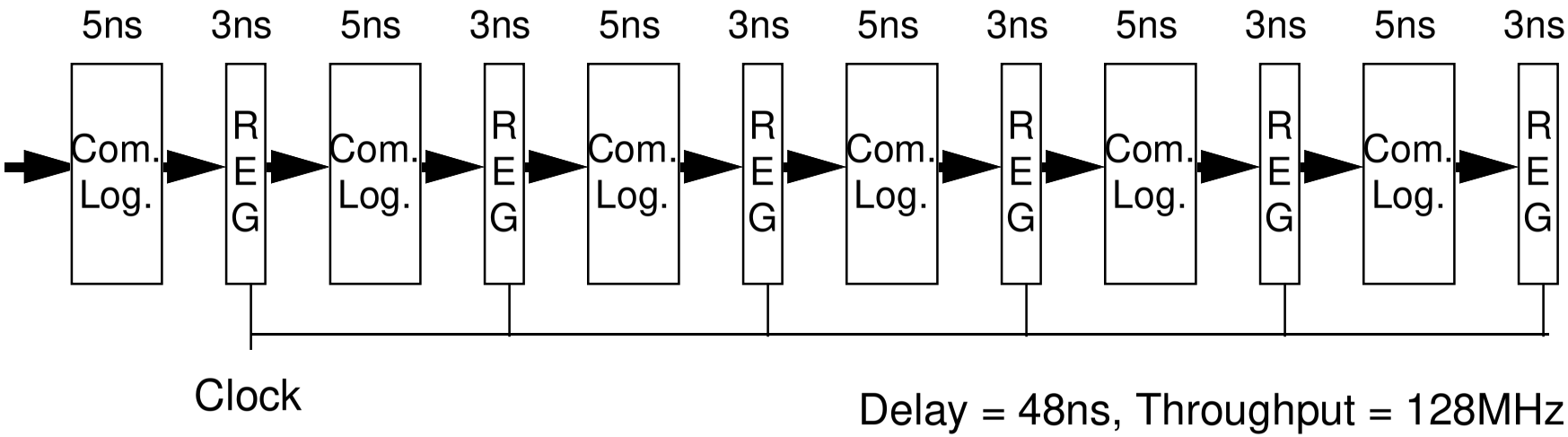
- **Space operations 13ns apart**
- **3 operations occur simultaneously**

Limitation: Nonuniform Pipelining



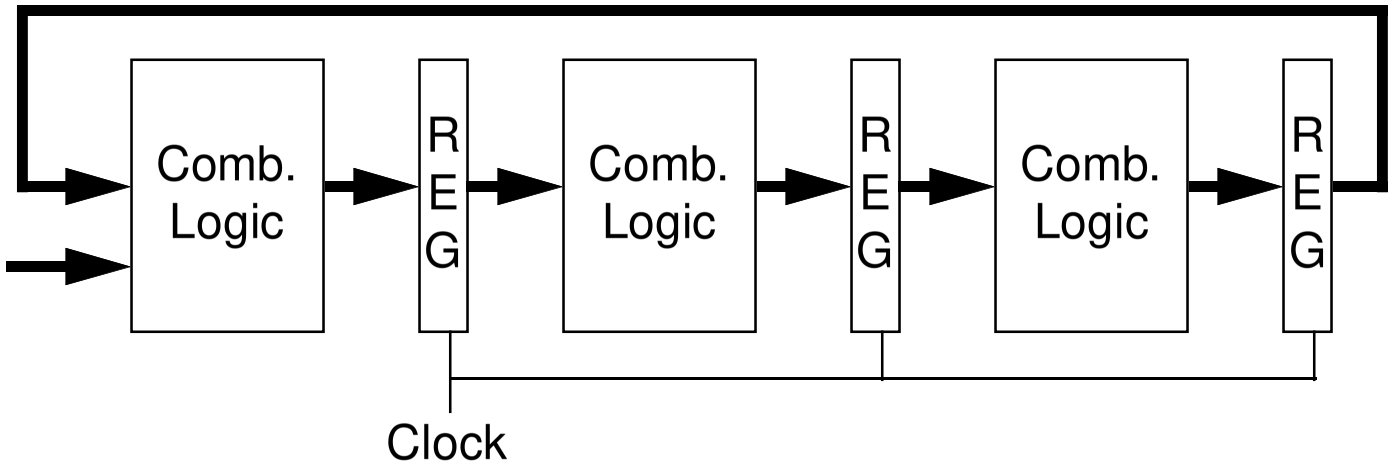
- **Throughput limited by slowest stage**
- **Must attempt to balance stages**

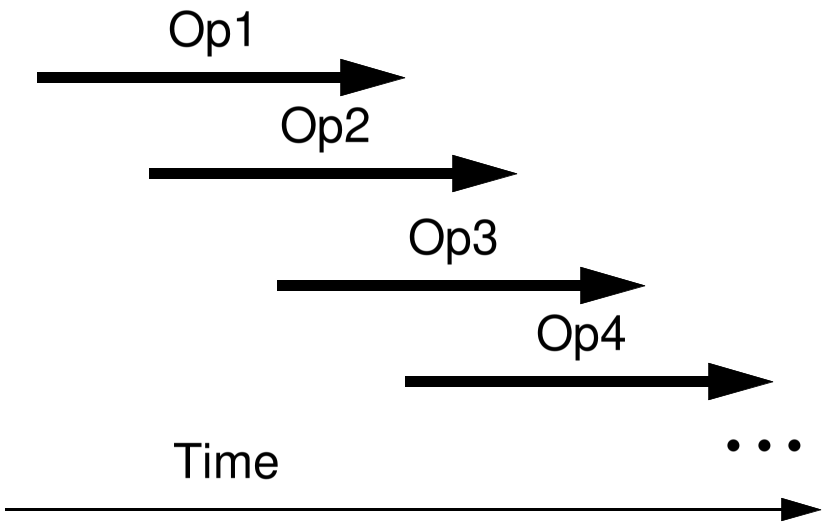
Limitation: Deep Pipelines



- **Diminishing returns as add more pipeline stages**
- **Register delays become limiting factor**
  - Increased latency
  - Small througput gains

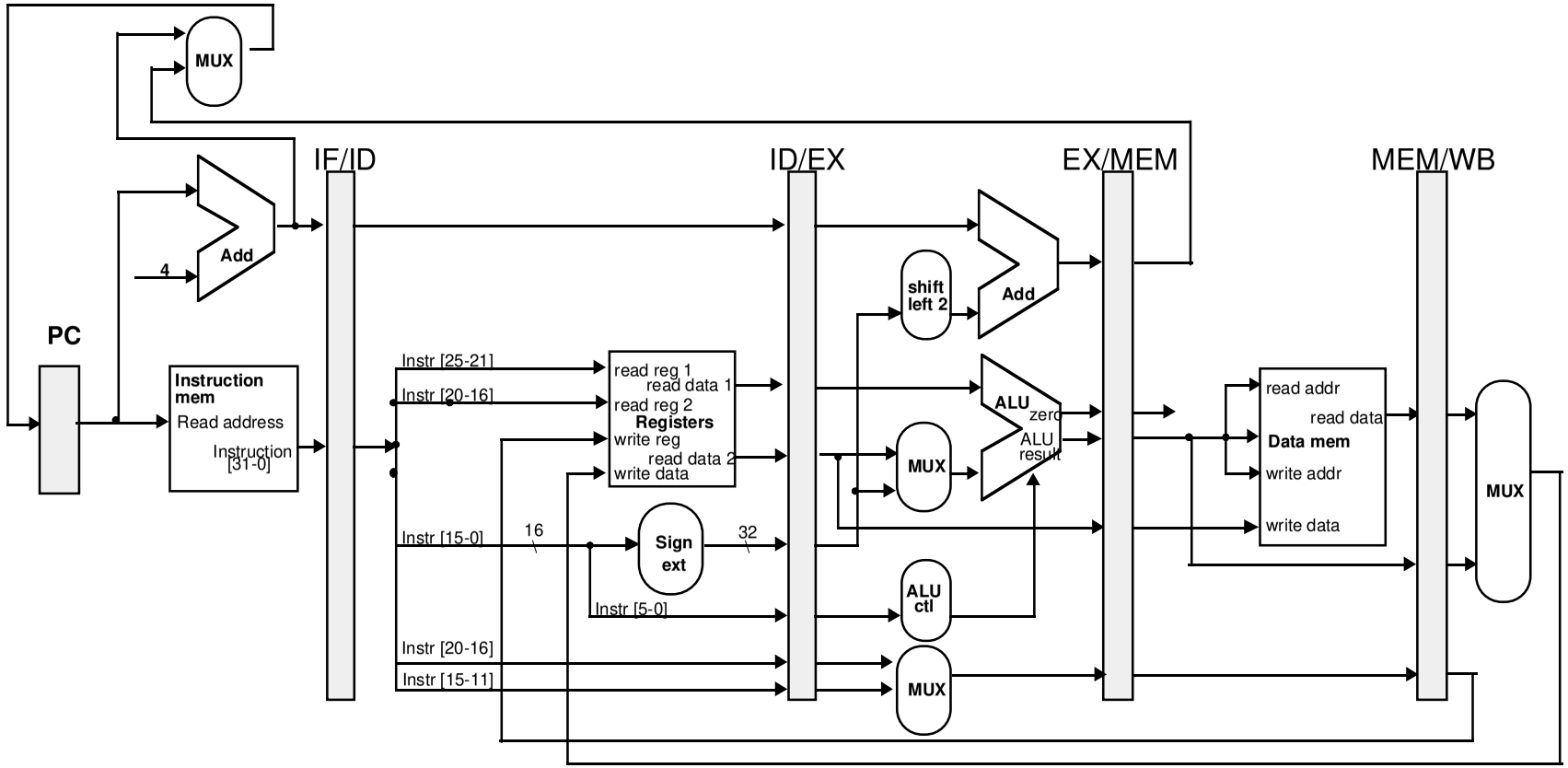
Limitation: Sequential Dependencies



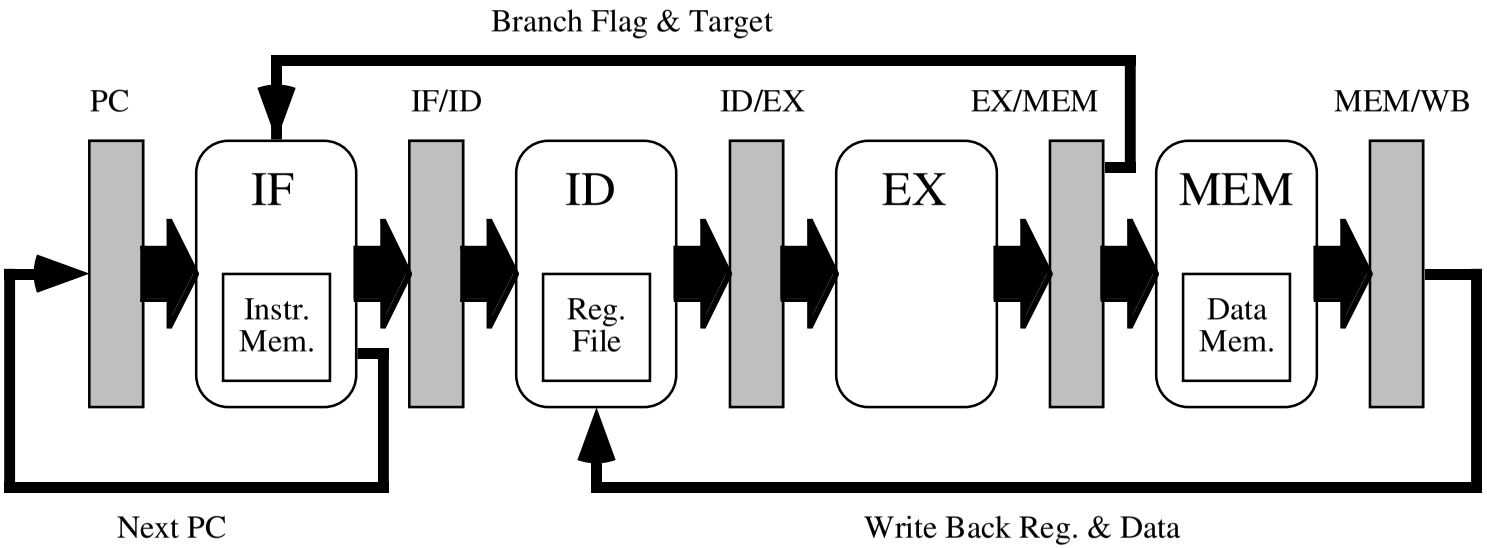


- **Op4 gets result from Op1 !**
- ***Pipeline Hazard***

Pipelined datapath



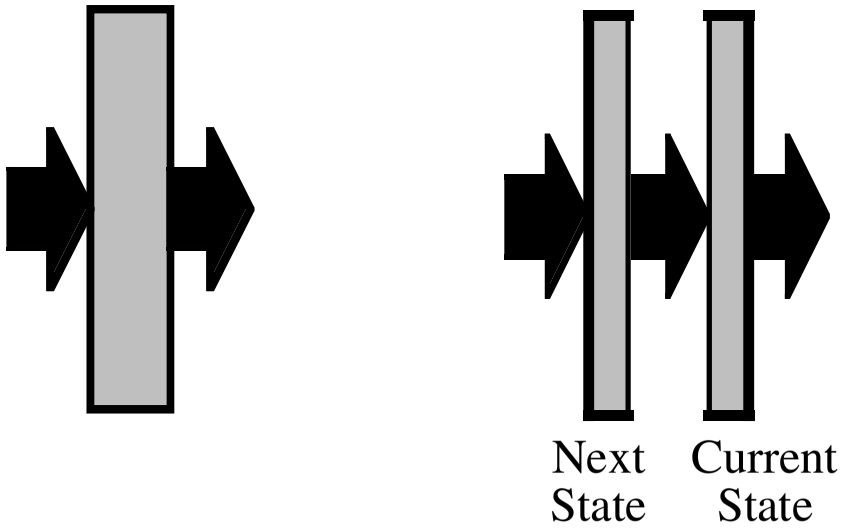
Pipeline Structure



Notes

- **Each stage consists of operate logic connecting pipe registers**
- **WB logic merged into ID**
- **Additional paths required for forwarding**

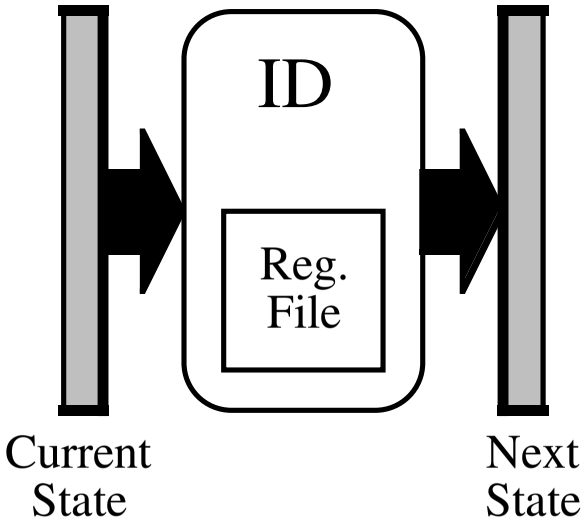
Pipe Register



Operation

- **Current State stays constant while Next State being updated**
- **Update involves transferring Next State to Current**

Pipeline Stage



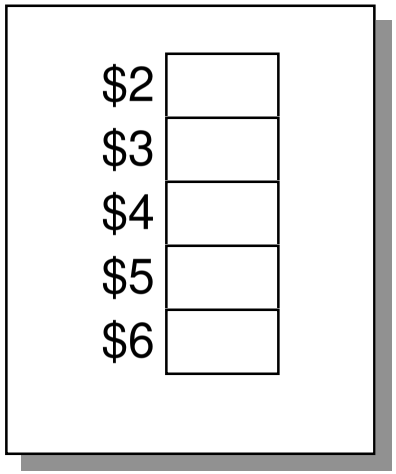
Operation

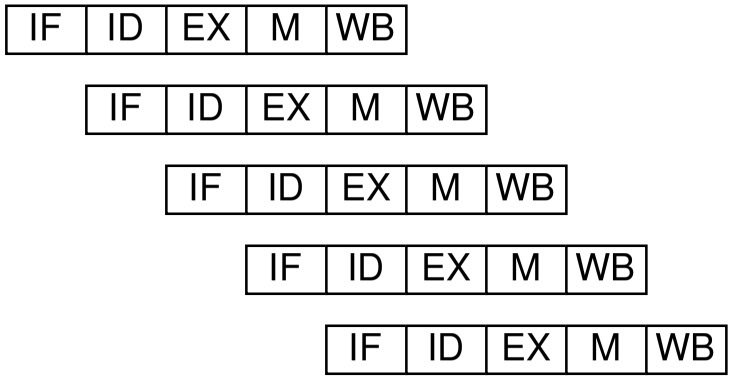
- **Computes next state based on current**
  - From/to one or more pipe registers
- **May have embedded memory elements**
  - Low level timing signals control their operation during clock cycle
  - Writes based on current pipe register state
  - Reads supply values for Next state

Data Hazards in MIPS Pipeline

Problem

- **Registers read in ID, and written in WB**
- **Must resolve conflict between instructions competing for register array**
  - Generally do write back in first half of cycle, read in second
- **But what about intervening instructions?**
- **E.g., suppose initially $2 is zero:**






```
addiu $2, $0, 63
addiu $3, $2, 0
addiu $4, $2, 0
addiu $5, $2, 0
addiu $6, $2, 0
```

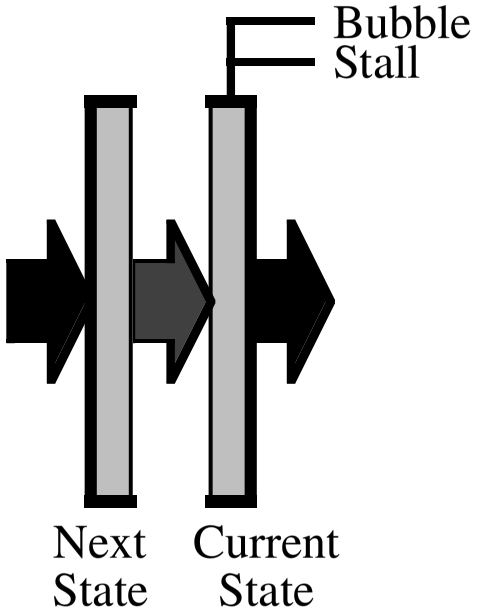
Handling Hazards by Stalling

Idea

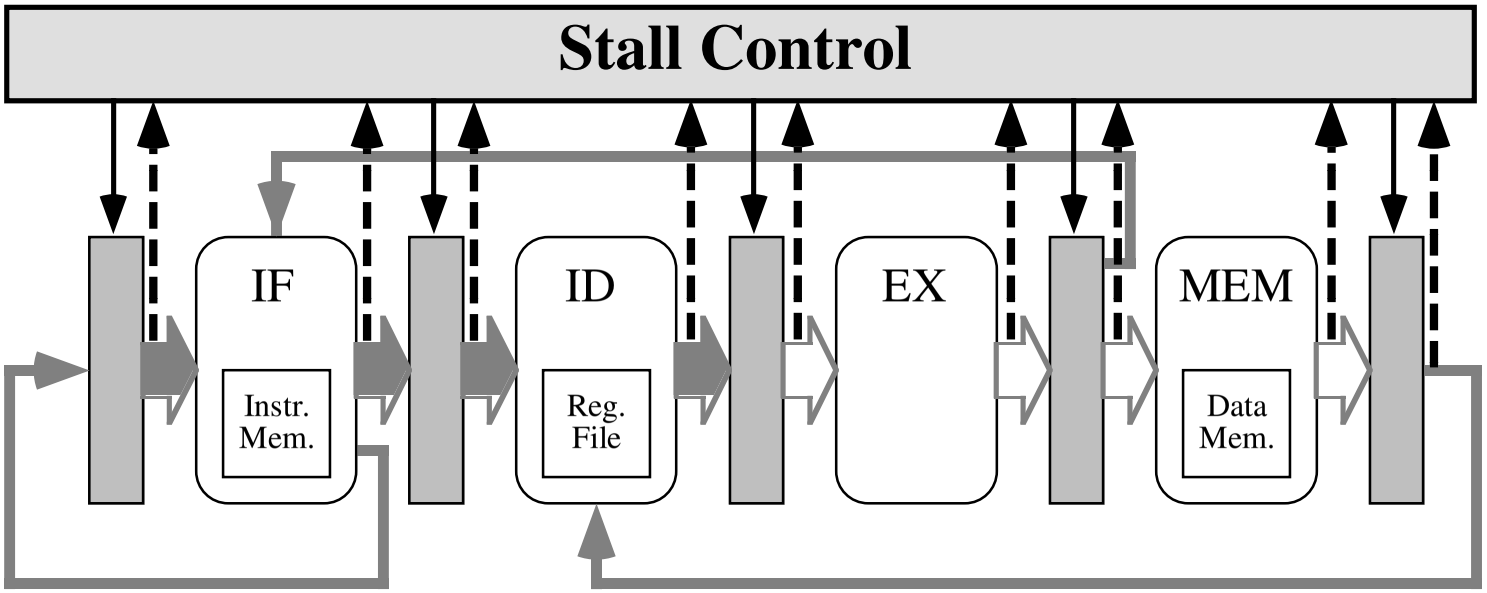
- **Delay instruction until hazard eliminated**
- **Put "bubble" into pipeline**
  - Dynamically generated NOP

Pipe Register Operation

- **Normally transfer next state to current**
- **"Stall" indicates that current state should not be changed**
- **"Bubble" indicates that current state should be set to 0**
  - Stage logic designed so that 0 is like NOP
  - [Other conventions possible]



Stall Control



Stall Logic

- **Determines which stages to stall or bubble on next update**

Observations on Stalling

Good

- **Relatively simple hardware**
- **Only penalizes performance when hazard exists**

Bad

- **As if placed NOP's in code**
  - Except that does not waste instruction memory

Reality

- **Some problems can only be dealt with by stalling**
- **E.g., instruction cache miss**
  - Stall PC, bubble IF/ID
- **Data cache miss**
  - Stall PC, IF/ID, ED/EX, EX/MEM, bubble MEM/WB

Forwarding (Bypassing)

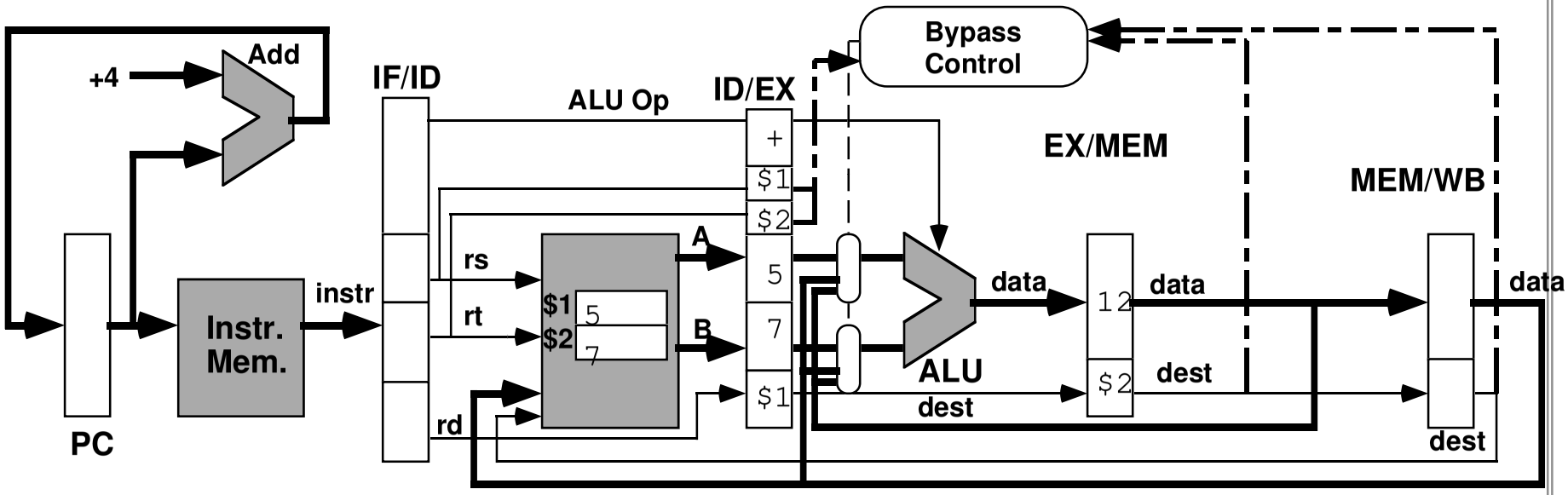
Observation

- **ALU data generated at end of EX**
  - Steps through pipe until WB
- **ALU data consumed at beginning of EX**

Idea

- **Expedite passing of previous instruction result to ALU**
- **By adding extra data pathways and control**

Forwarding for ALU-ALU Hazard



Program:

```
0x18:       add $2, $1, $2
0x1c:       add $1, $1, $2
```

Some Hazards with Loads & Stores

Data Generated by Load

Load-ALU

```
lw $1, 8($2)
add $2, $1, $2
```

Load-Store Data

```
lw $1, 8($2)
sw $1, 12($2)
```

Load-Store (or Load) Addr.

```
lw $1, 8($2)
sw $1, 12($1)
```

Data Generated by Store

Store-Load Data

```
sw $1, 8($2)
lw $3, 8($2)
```

Analysis of Data Transfers

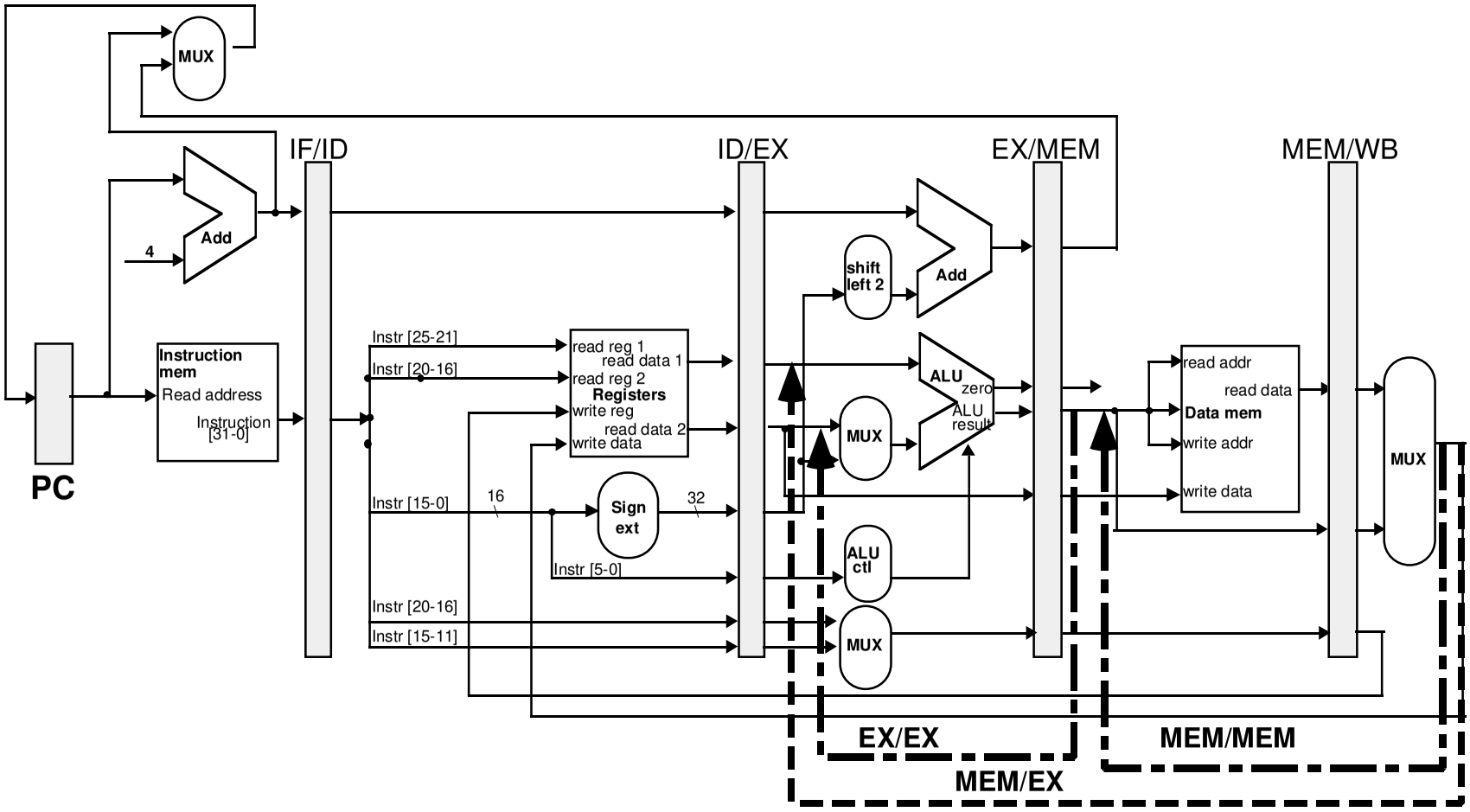
Data Sources

- **Available after EX**
  - ALU Result Reg-Reg Result
- **Available after MEM**
  - Read Data Load result
  - ALU Data Reg-Reg Result passing through MEM stage

Data Destinations

- **ALU A input Need in EX**
  - Reg-Reg Operand
  - Load/Store Base
- **ALU B input Need in EX**
  - Reg-Reg Operand
- **Write Data Need in MEM**
  - Store Data

Complete Bypassing



Impact of Forwarding

Single Remaining Hazard Class

- **Load followed by ALU operation**
  - Including address calculation

MIPS Architecture

- **Don't put instruction after load that uses loaded register**
  - Not even allowed to assume it will be old value

Alternative

- **Stall following instruction one cycle**
- **Then pick up with MEM/EX bypass**

Load-ALU

```
lw $1, 8($2)
add $2, $1, $2
```

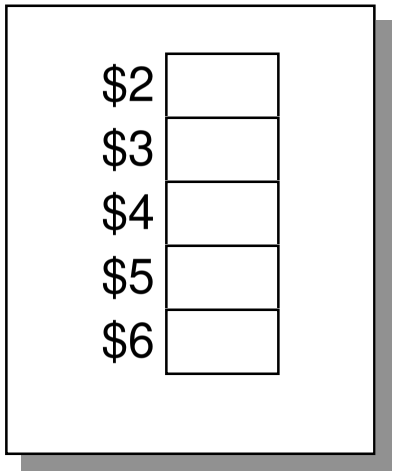
Load-Store (or Load) Addr.

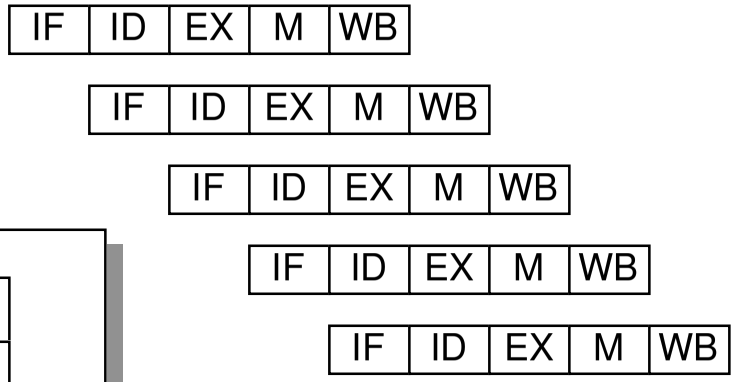
```
lw $1, 8($2)
lw $4, 12($1)
```

Control Hazards in MIPS Pipeline

Problem

- **Instruction fetched in IF, branch condition set in MEM**
- **When does branch take effect?**
- **E.g.: assume initially that all registers = 0**





beq $0, $0, target

addiu $2, $0, 63

addiu $3, $0, 63

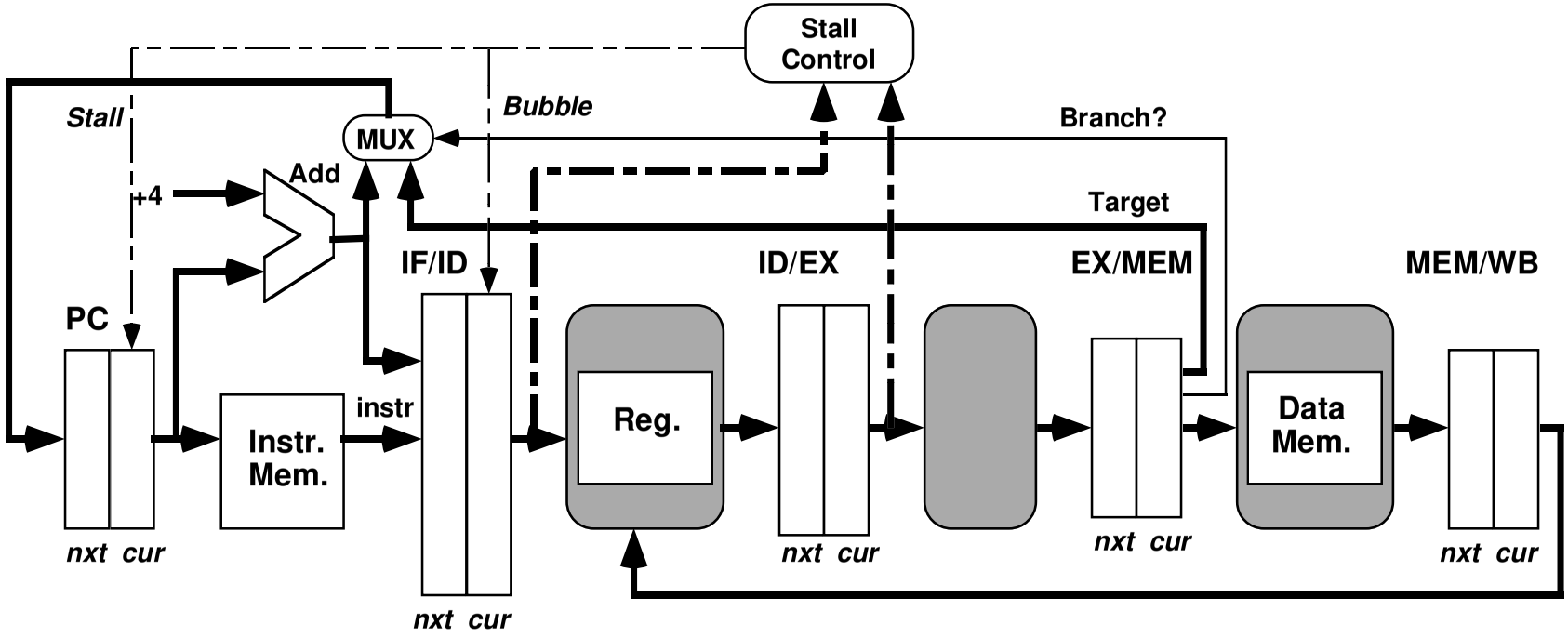
addiu $4, $0, 63

addiu $5, $0, 63

target: addiu $6, $0, 63

Stall Until Resolve Branch

- Detect when branch in stages ID or EX
- Stop fetching until resolve
- Delay slot instruction just before target

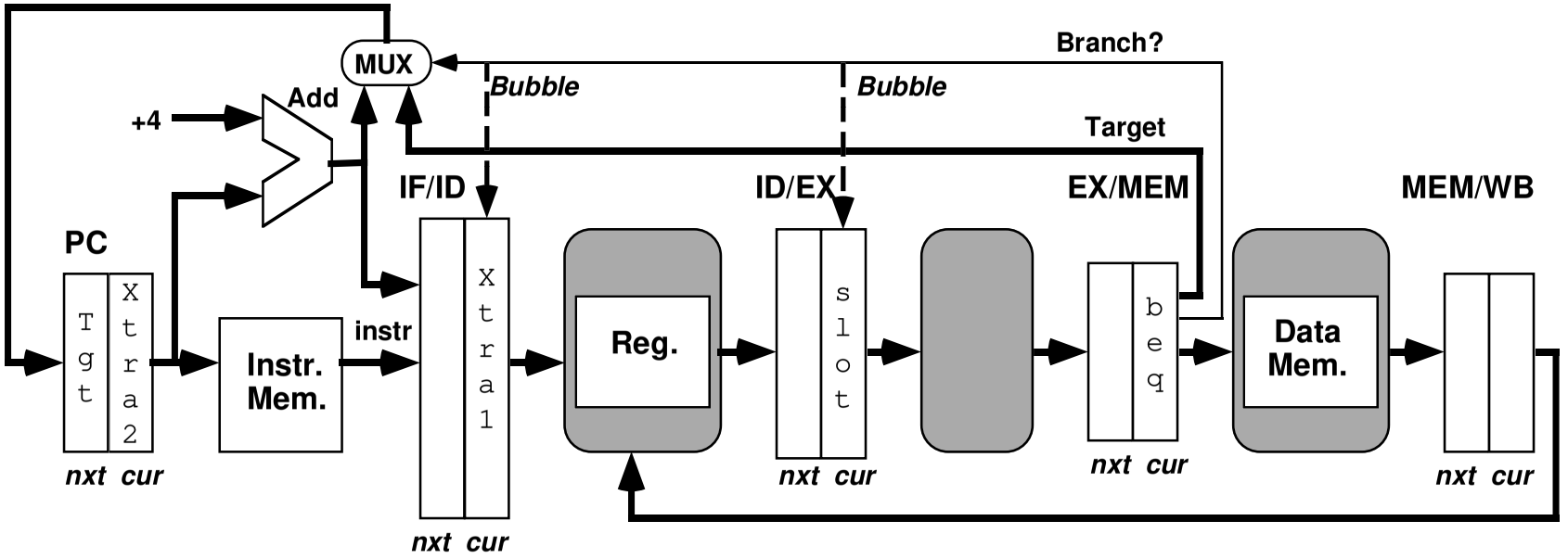


Fetch & Cancel When Taken

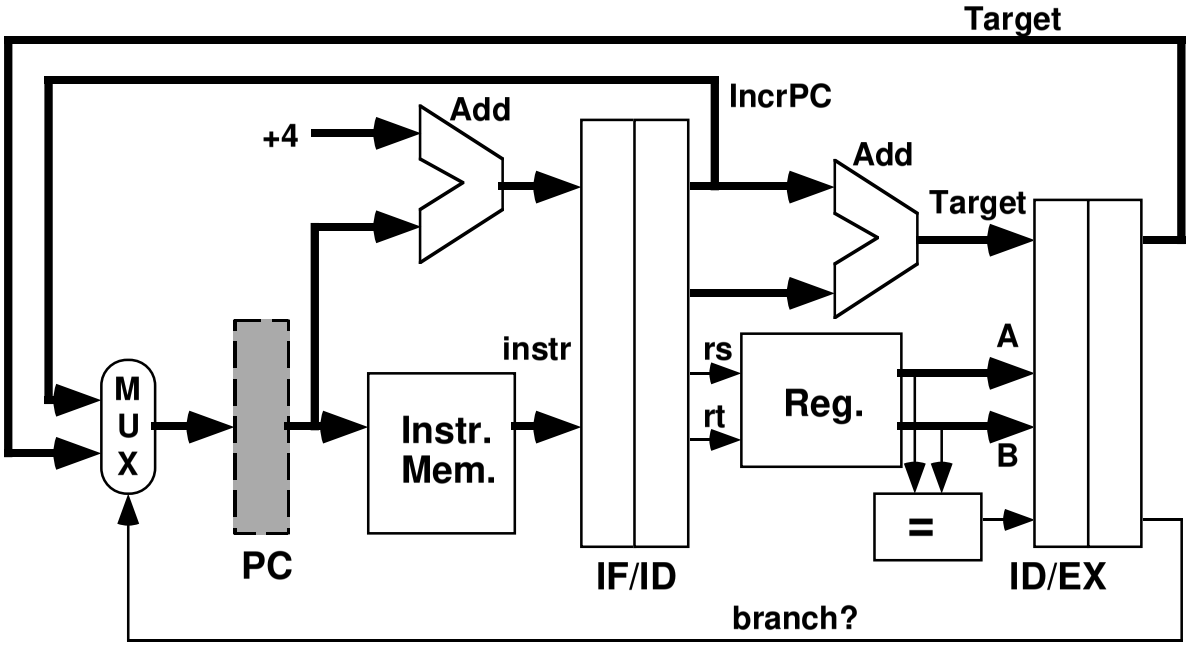
- **Instruction does not cause any updates until MEM or WB stages**
- **Instruction can be "cancelled" from pipe up through EX stage**
  - Replace with bubble

Strategy

- **Continue fetching under assumption that branch not taken**
- **If decide to take branch, cancel undesired ones**



How Does Real MIPS Handle Branches?

Avoids any Branch Penalty!

- **Do not treat PC as pipe register**
  - Loaded at beginning of IF
- **Compute branch target & branch condition in ID**
  - Can fetch at target in following cycle