Handling Data Hazards in Pipelined MIPS Processor

Feb 29, 1996

Topics

- **Pipeline for Reg-Reg Ops.**
- **Hazard handling mechanisms**
  - Stalling
  - Forwarding
- **Adding Loads & Stores**

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

RI-type instructions (addiu): rt <-- rs funct I16

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

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

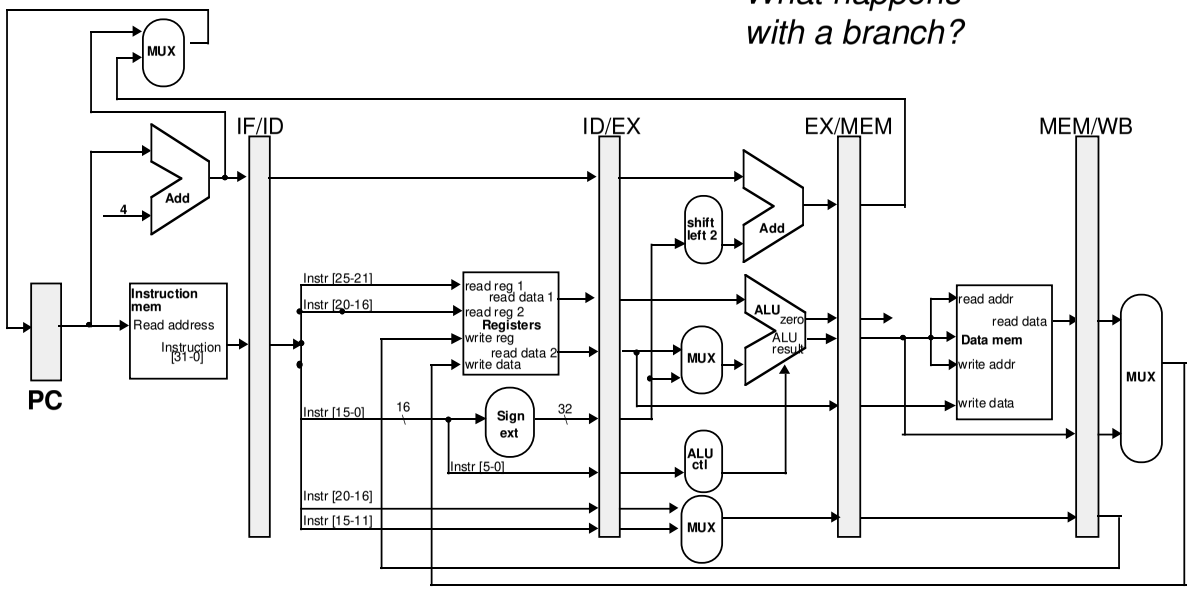
35 or 43	rs	rt	I16
31-26	25-21	20-16	15-0

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

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

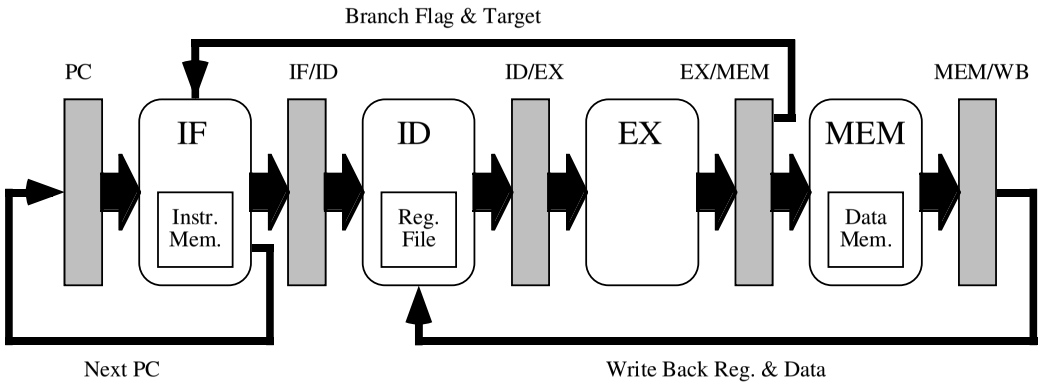
Pipelined datapath

What happens with a branch?



How about that writeback loop?

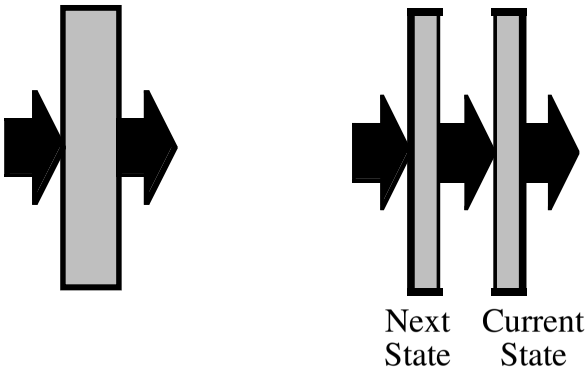
Pipeline Structure



Notes

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

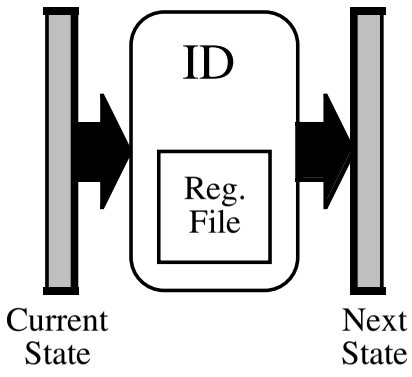
Pipe Register



Operation

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

Pipeline Stage



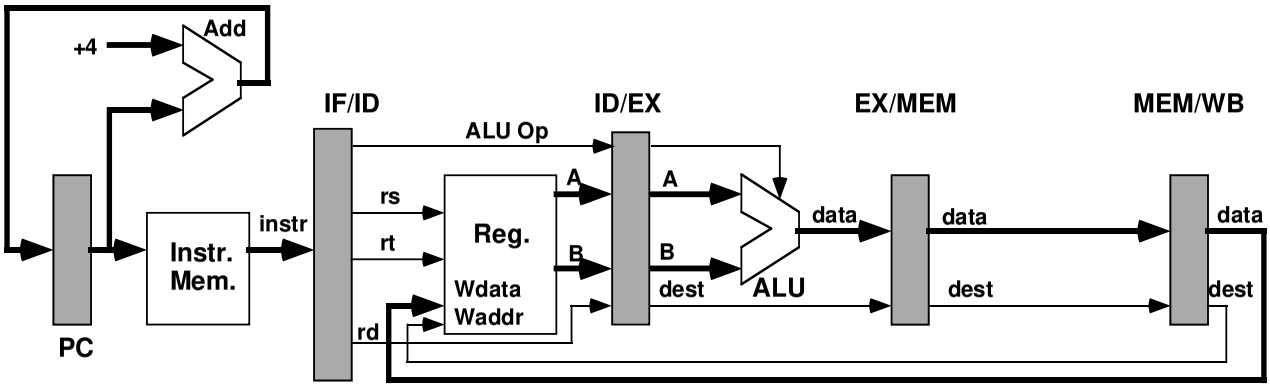
Operation

- **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

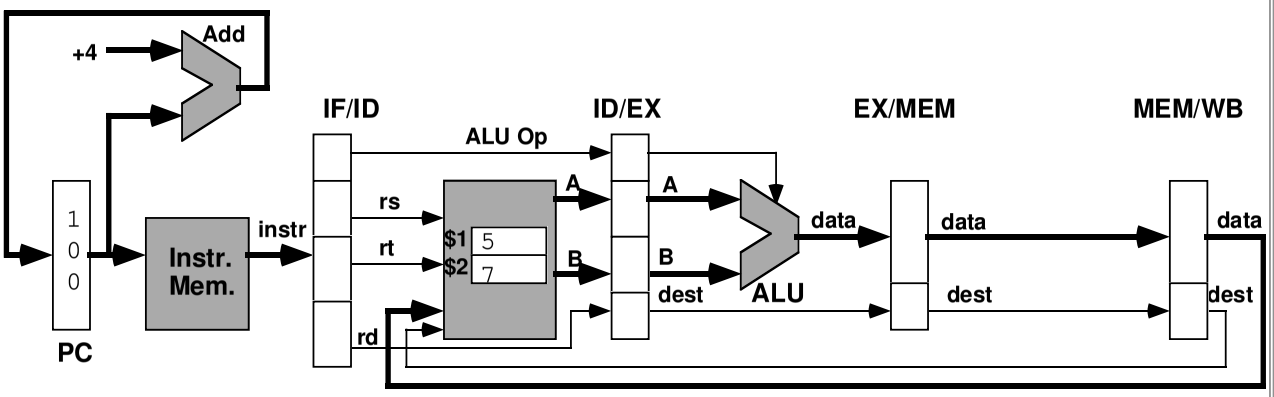
Reg-Reg Instructions

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



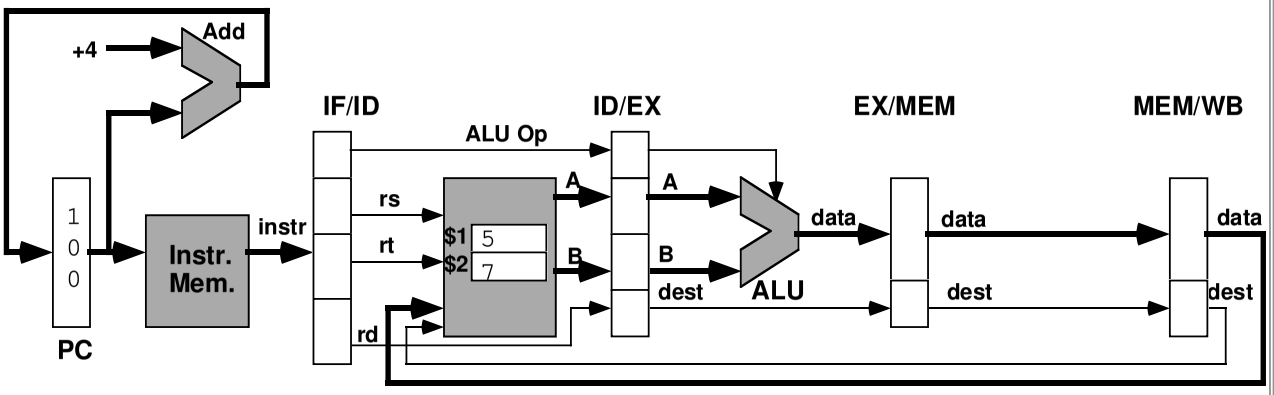
Instruction Execution Example



Program:

```
100: add $2, $1, $2
```

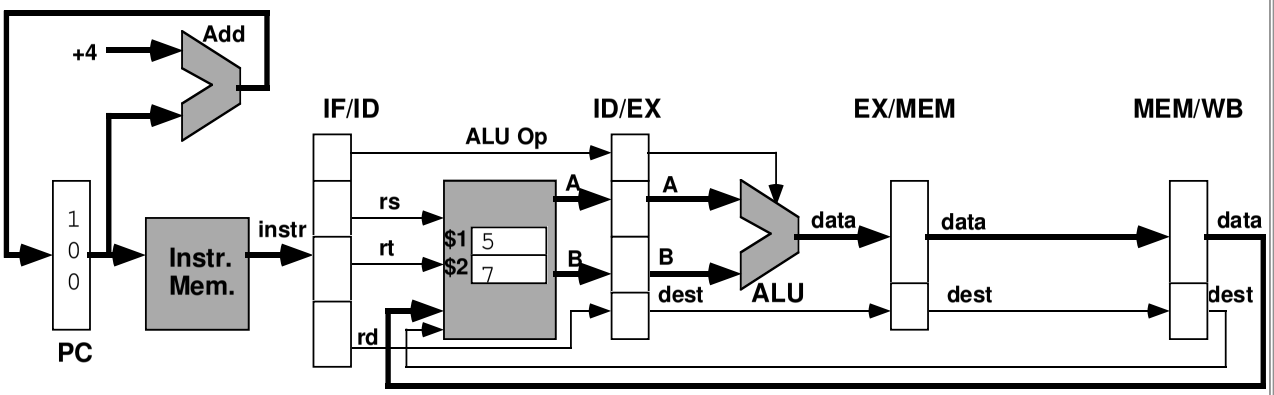
Example #2



Program:

```
100: add $2, $1, $2
104: add $1, $1, $2
```

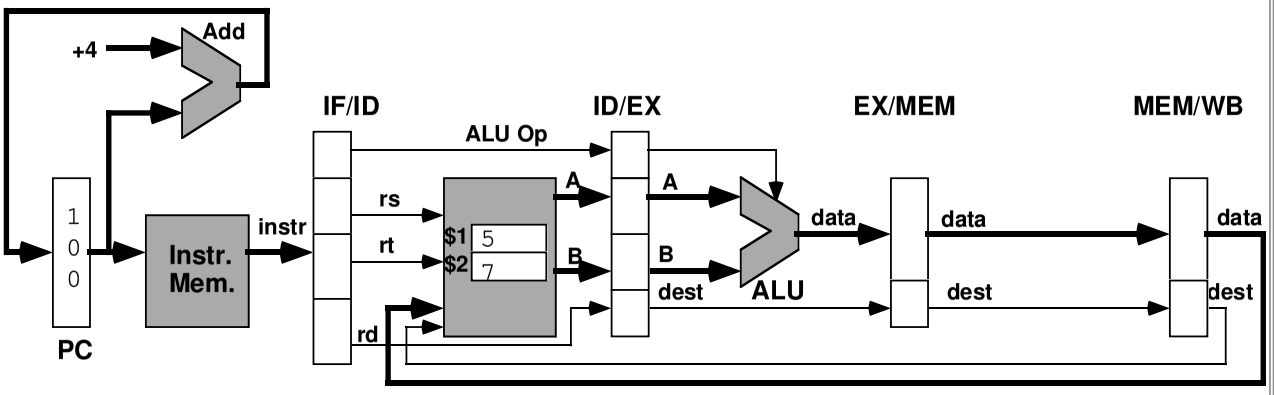

Example #3



Program:

```
100: add $2, $1, $2
104: add $0, $0, $0
108: add $1, $1, $2
```

Example #4



Program:

```
100: add $2, $1, $2
104: add $0, $0, $0
108: add $0, $0, $0
112: add $1, $1, $2
```

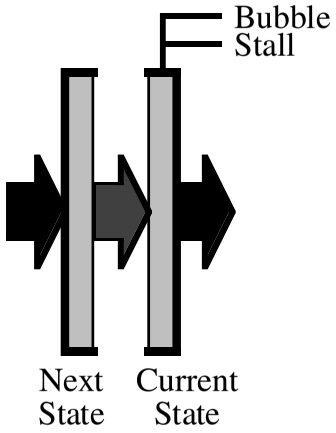

Handling Hazards by Stalling

Idea

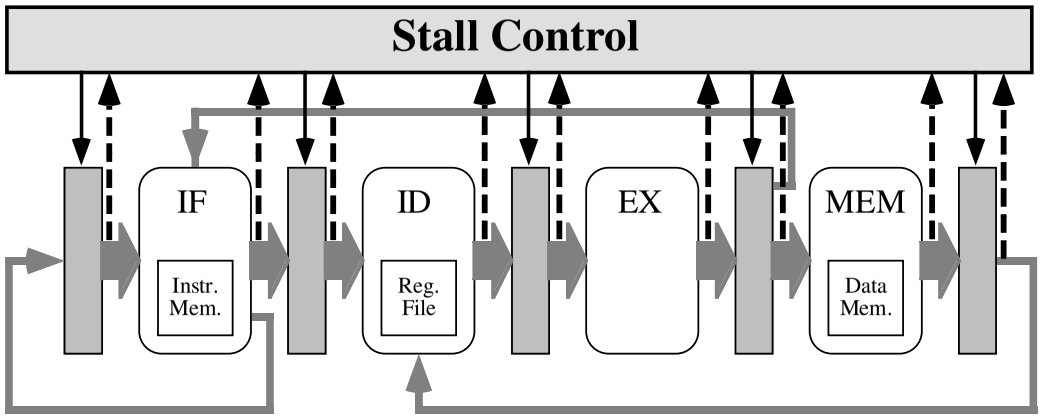
- **Delay instruction until hazard eliminated**
- **Put "bubble" into pipeline**
  - Dynamically generated NOP

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]



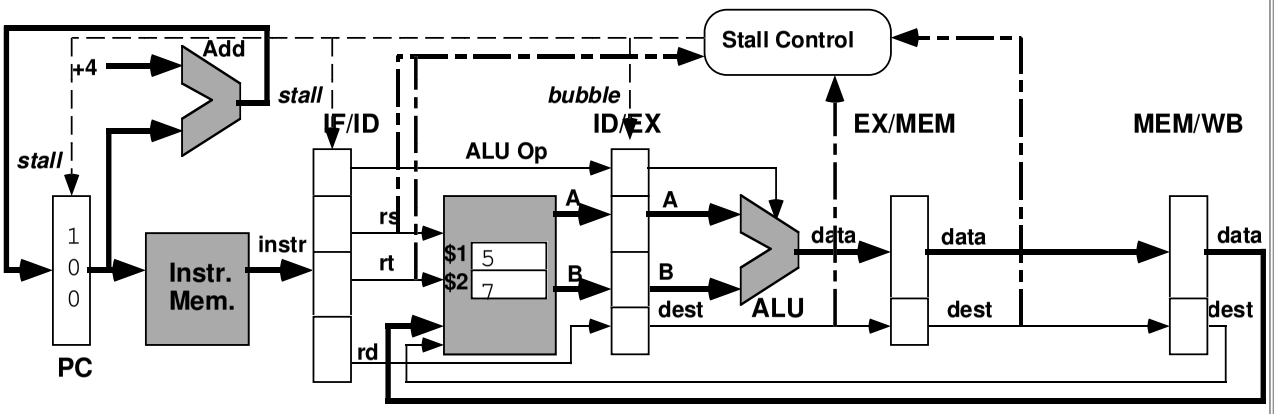
Stall Control



Stall Logic

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

Example #2 Revisited



Program:

```
100: add $2, $1, $2
104: add $1, $1, $2
```

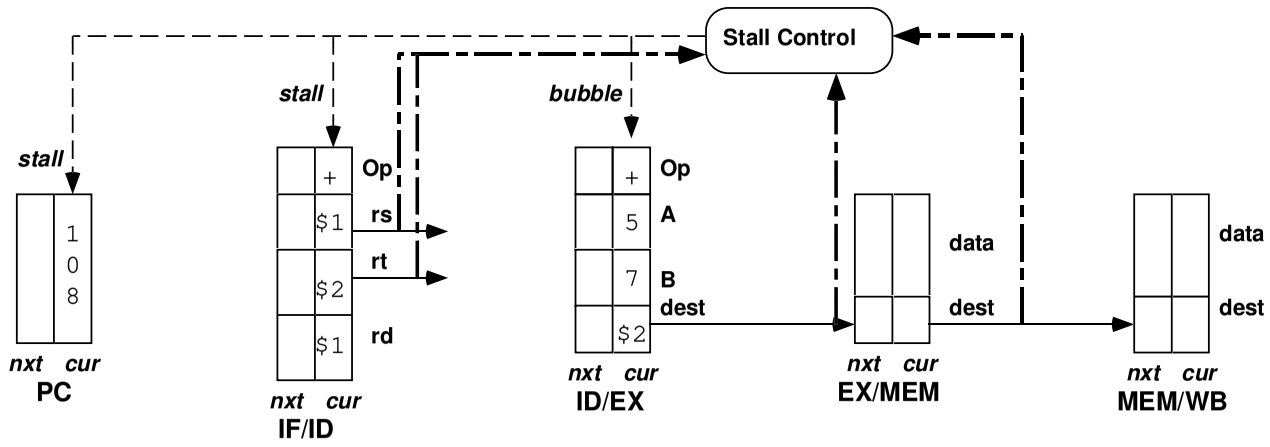
Rule:

- **Stall when rs or rt matches dest in ID/EX or dest in EX/MEM**

Hitting Stall

Program

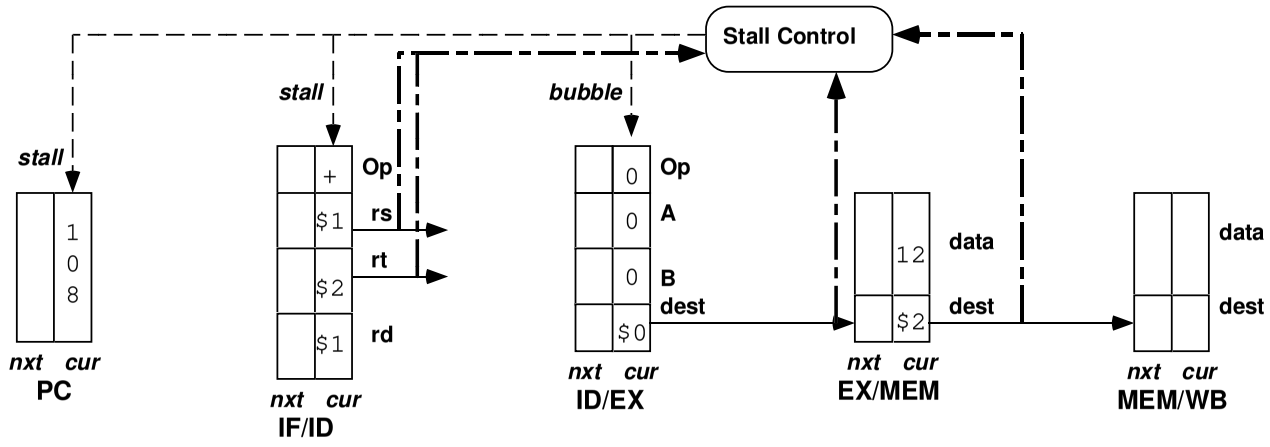
```
100: add $2, $1, $2
104: add $1, $1, $2
```



First Stall Cycle

Program

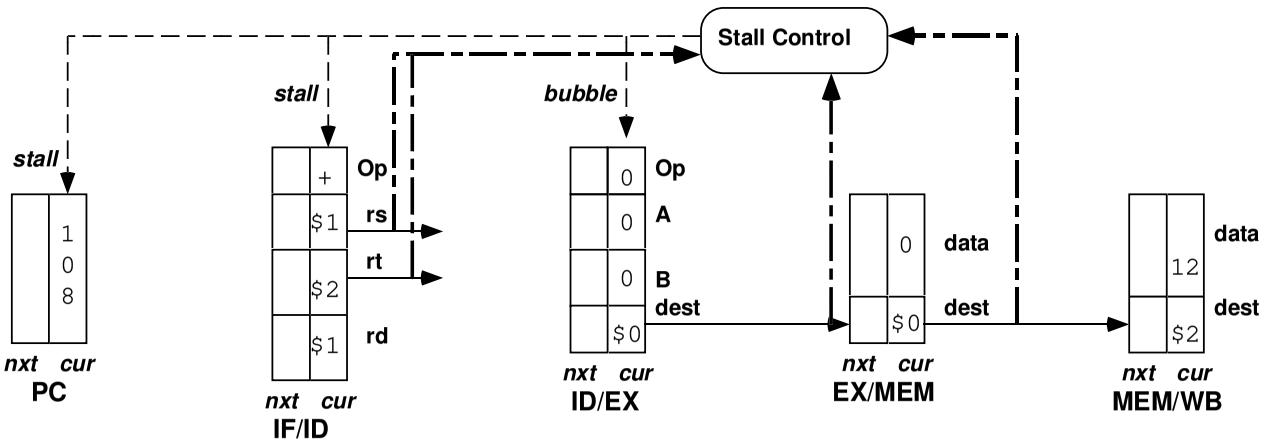
```
100: add $2, $1, $2
104: add $1, $1, $2
```



Second Stall Cycle

Program

```
100: add $2, $1, $2
104: add $1, $1, $2
```



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

Bypassing

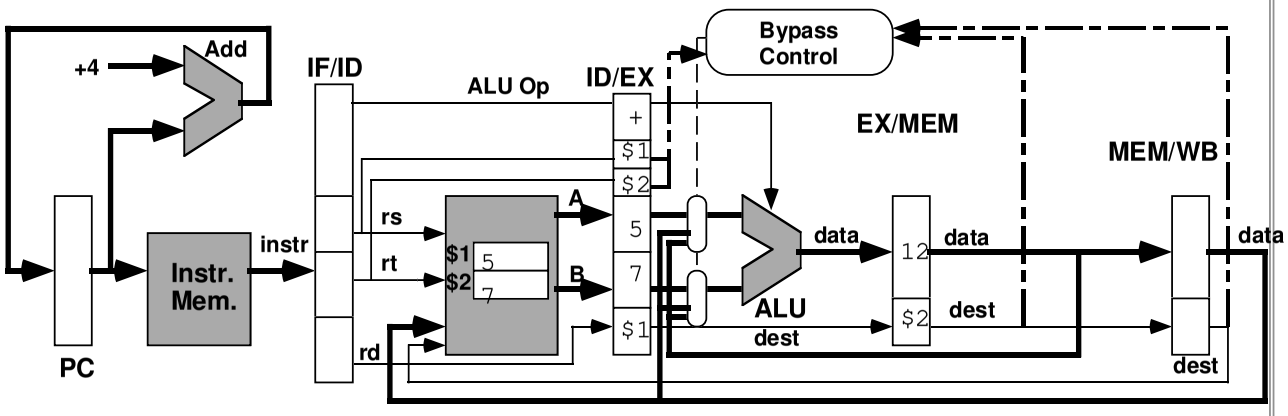
Observation

- **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**

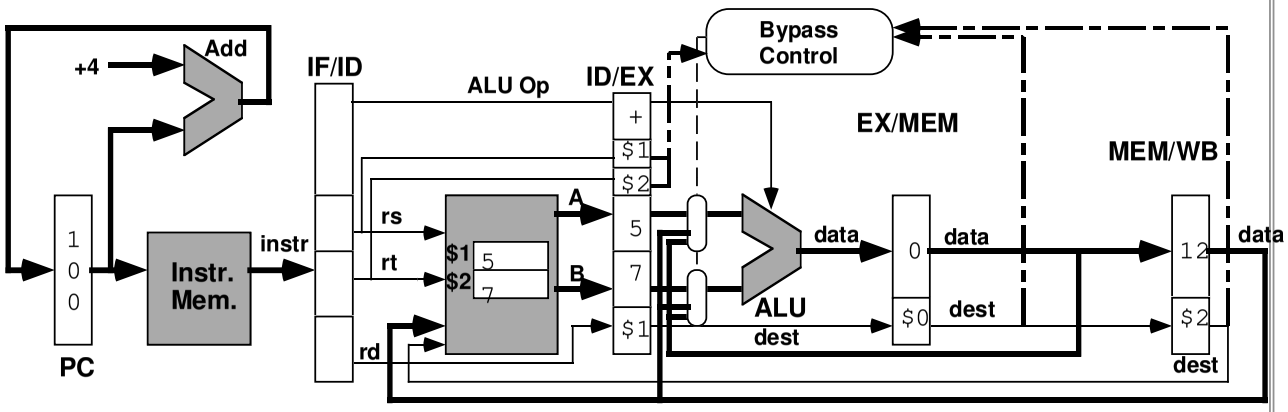
Example #2 Revisited



Program:

```
100: add $2, $1, $2
104: add $1, $1, $2
```

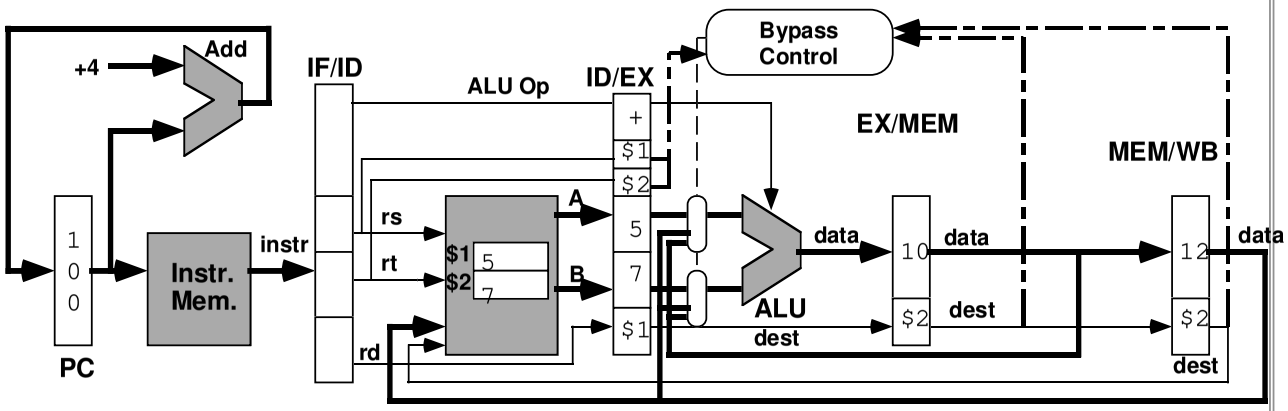
Example #3 Revisited



Program:

```
100: add $2, $1, $2
104: add $0, $0, $0
108: add $1, $1, $2
```


Example #4



Program:

```
100: add $2, $1, $2
104: add $2, $1, $1
108: add $1, $1, $2
```

Hazards with Loads & Stores

Data Generated by ALU Instr.

ALU-ALU

```
add $1, $2, $3
add $4, $1, $2
```

ALU-Store Data

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

ALU-Store (or Load) Address

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

Distant ALU-ALU

```
add $1, $2, $3
add $0, $0, $0
add $4, $1, $2
```

Distant ALU-Store Data

```
add $1, $2, $3
add $0, $0, $0
sw $1, 8($2)
```

Distant ALU-Store (or Load) Addr.

```
add $1, $2, $3
add $0, $0, $0
sw $2, 8($1)
```


More 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

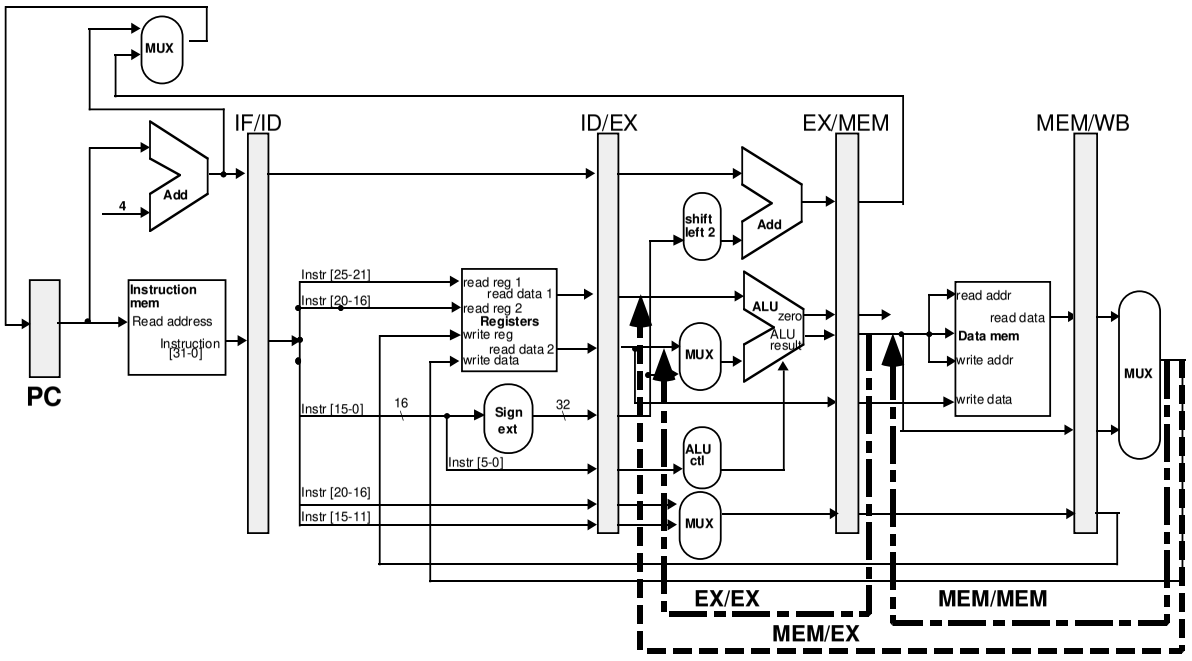
Data Sources

- **Available from EX**
  - ALU Result Reg-Reg Result
- **Available from 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

Adding Bypasses

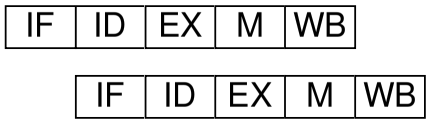


Hazard Analysis #1

Data Generated by ALU Instr.

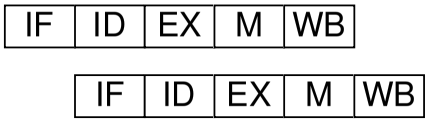
ALU-ALU

```
add $1, $2, $3
add $4, $1, $2
```



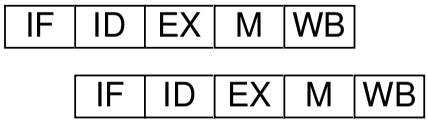
ALU-Store Data

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



ALU-Store (or Load) Address

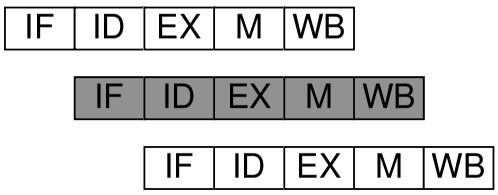
```
add $1, $2, $3
sw $2, 8($1)
```



Hazard Analysis #2

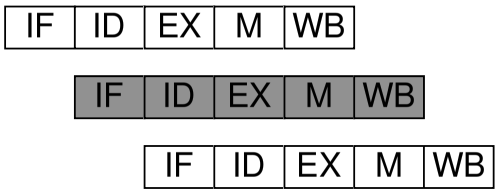
Distant ALU-ALU

```
add $1, $2, $3
add $0, $0, $0
add $4, $1, $2
```



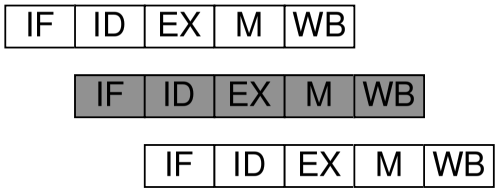
Distant ALU-Store Data

```
add $1, $2, $3
add $0, $0, $0
sw $1, 8($2)
```



Distant ALU-Store (or Load) Addr.

```
add $1, $2, $3
add $0, $0, $0
sw $2, 8($1)
```

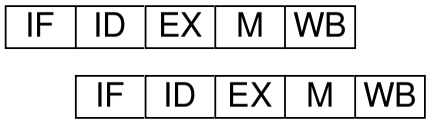


Hazard Analysis #3

Data Generated by Load

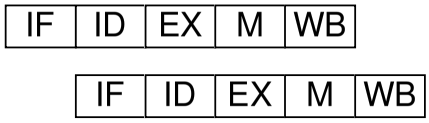
Load-ALU

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



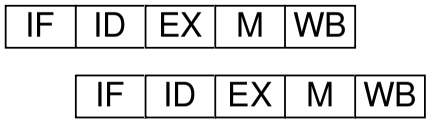
Load-Store Data

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



Load-Store (or Load) Addr.

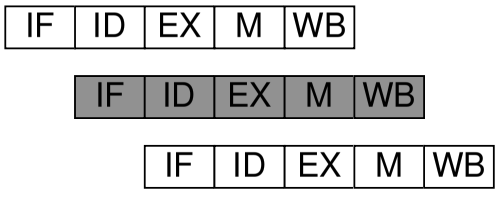
```
lw $1, 8($2)
sw $1, 12($1)
```



Hazard Analysis #4

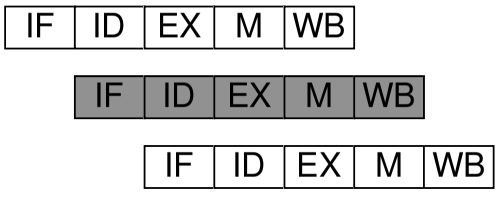
Distant Load-ALU

```
lw $1, 8($2)
add $0, $0, $0
add $2, $1, $2
```



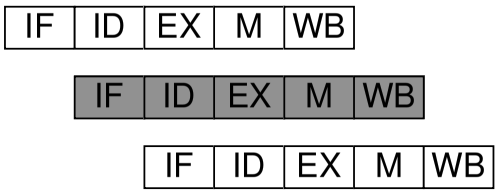
Distant Load-Store Data

```
lw $1, 8($2)
add $0, $0, $0
sw $1, 12($2)
```



Distant Load- L/S Addr.

```
lw $1, 8($2)
add $0, $0, $0
sw $1, 12($1)
```



Impact of Forwarding

Only One Remaining Hazard Class

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

MIPS Architecture

- **Load delay slot**
- **Don't put instruction after load that uses loaded register**
  - Not even allowed to assume it will be old value
  - Assembler will rearrange or insert NOP

My Experiment

- **Force with (no)reorder directives**
- **Gives old value**

Load-ALU

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

Load-Store (or Load) Addr.

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