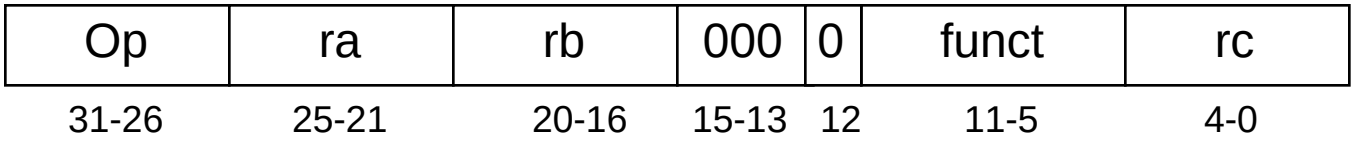
Handling Data Hazards in an Alpha Pipeline March 17, 1998

Topics

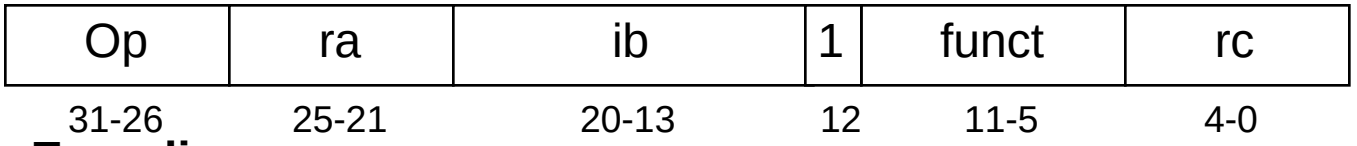
- Hazards in ALU instructions
- Handling by stalling
- Handling by forwarding
- Data hazards with other instruction types
- Systematic testing of hazard-handling logic

Alpha ALU Instructions

RR-type instructions (addq, subq, xor, bis, cmplt): rc <-- ra funct rb



RI-type instructions (addq, subq, xor, bis, cmplt): rc <-- ra funct ib



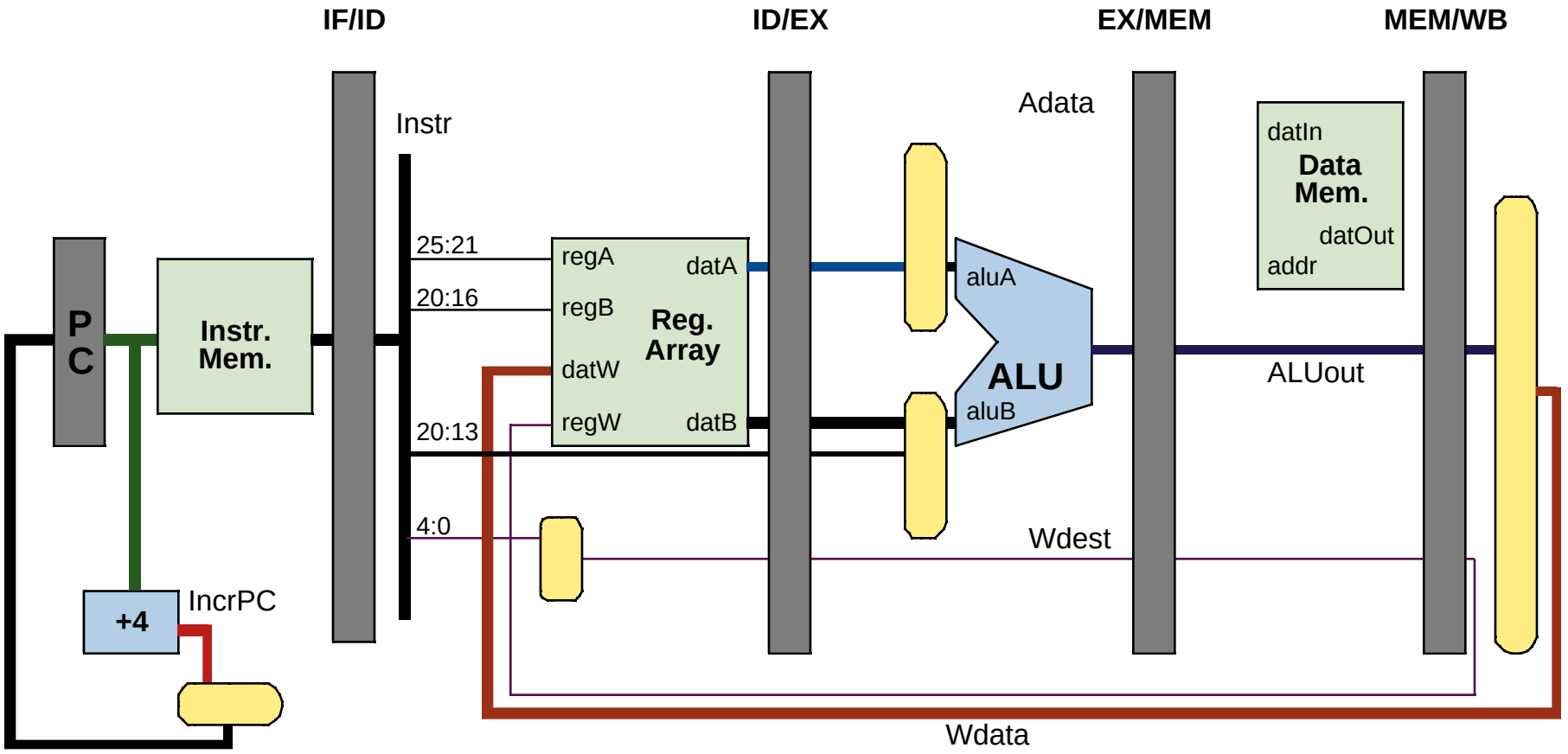
Encoding

- **ib is 8-bit unsigned literal**

Operation	Op field	funct field
`addq`	`0x10`	`0x20`
`subq`	`0x10`	`0x29`
`bis`	`0x11`	`0x20`
`xor`	`0x11`	`0x40`
`cmoveq`	`0x11`	`0x24`
`cmplt`	`0x11`	`0x4D`

Pipelined ALU Instruction Datapath

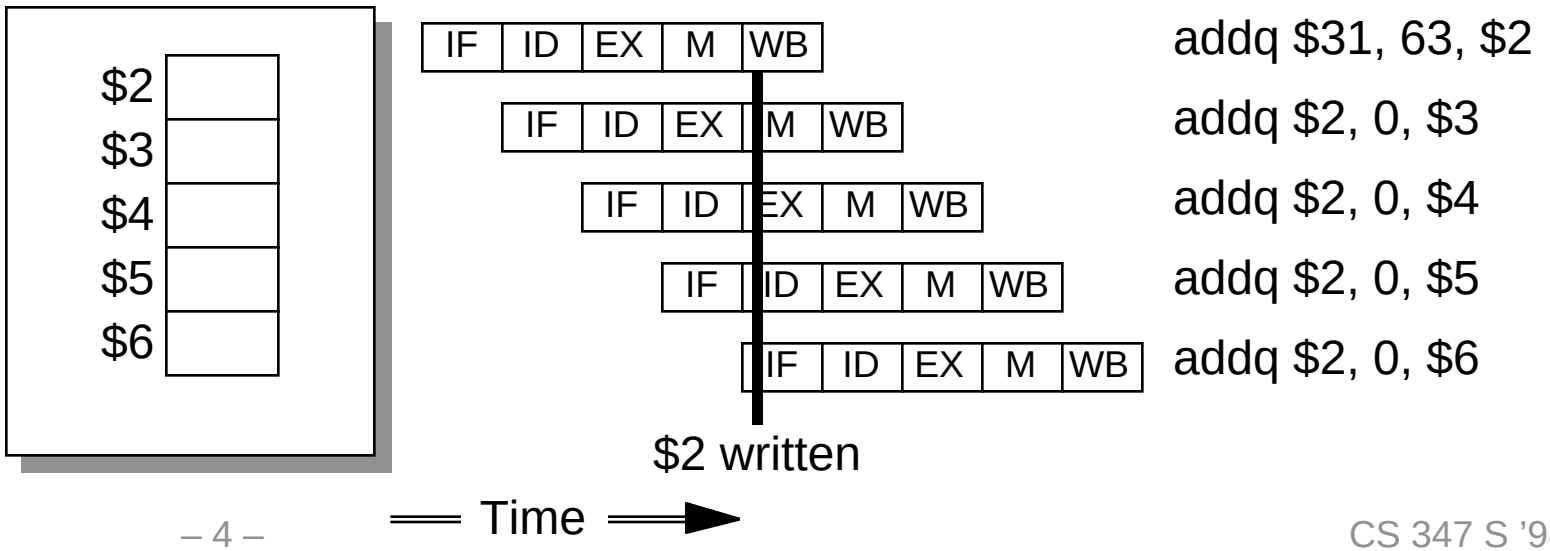
IF	ID	EX	MEM	WB
instruction fetch	instruction decode/ register fetch	execute	memory access	write back



Data Hazards in Alpha 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:**



Simulator Data Hazard Example

Operation

- **Read in ID**
- **Write in WB**
- **Write-before-read register file**

RAW Data Hazard

- **Potential conflict among different instructions**
- **Due to data dependencies**
- **"Read After Write"**
  - Register $2 written and then read

demo04.O

```
0x0: 43e7f402  addq  r31, 0x3f, r2 # $2 = 0x3F
0x4: 40401403  addq  r2, 0, r3     # $3 = 0x3F?
0x8: 40401404  addq  r2, 0, r4     # $4 = 0x3F?
0xc: 40401405  addq  r2, 0, r5     # $5 = 0x3F?
0x10:40401406  addq  r2, 0, r6     # $6 = 0x3F?
0x14:47ff041f  bis   r31, r31, r31
0x18:00000000  call_pal            halt
```

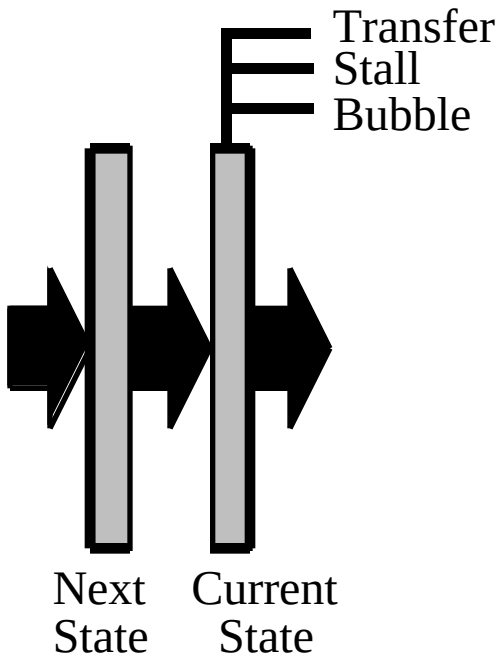
Handling Hazards by Stalling

Idea

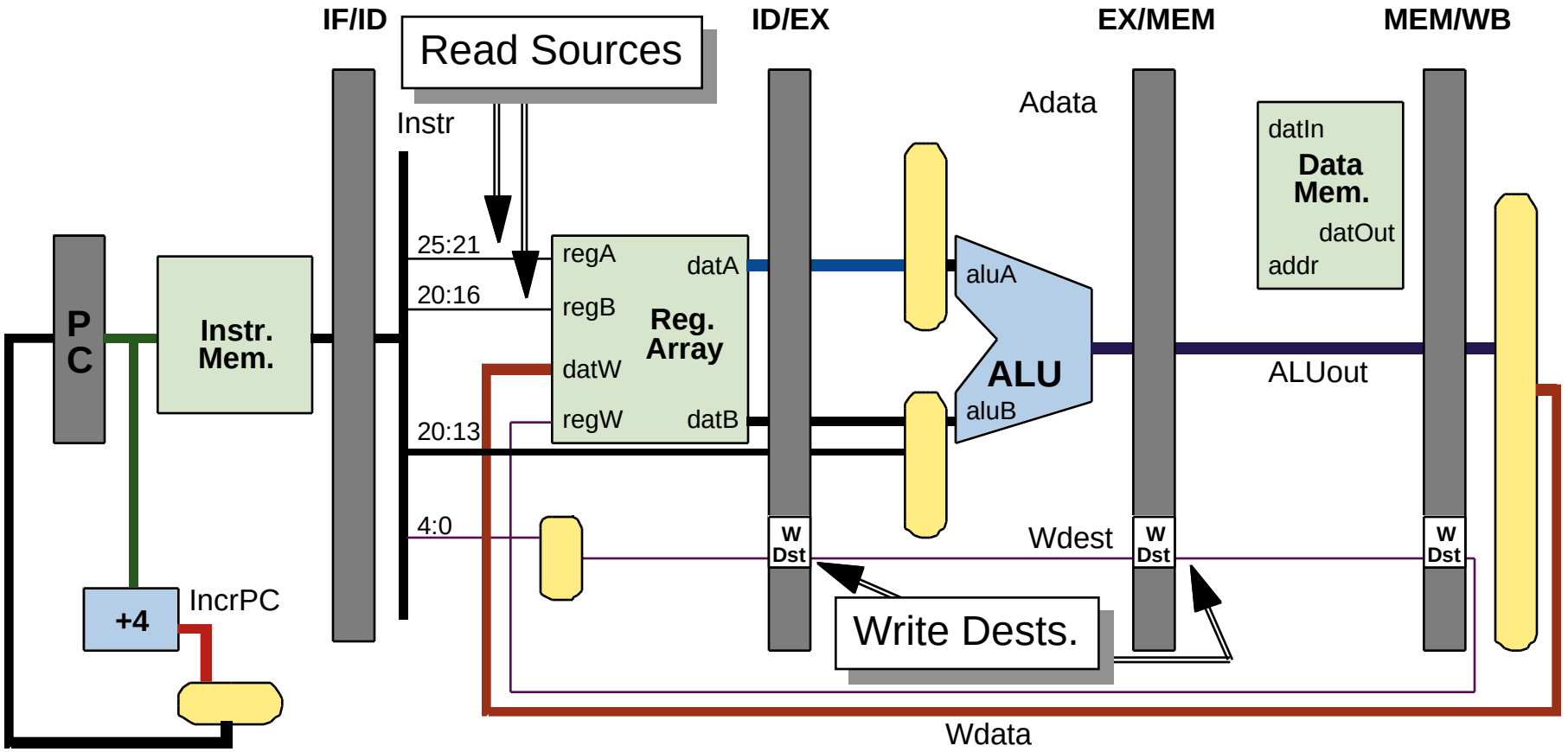
- **Delay instruction until hazard eliminated**
- **Put "bubble" into pipeline**
  - Dynamically generated NOP

Pipe Register Operation

- **"Transfer" (normal operation) indicates should 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]



Detecting Dependencies



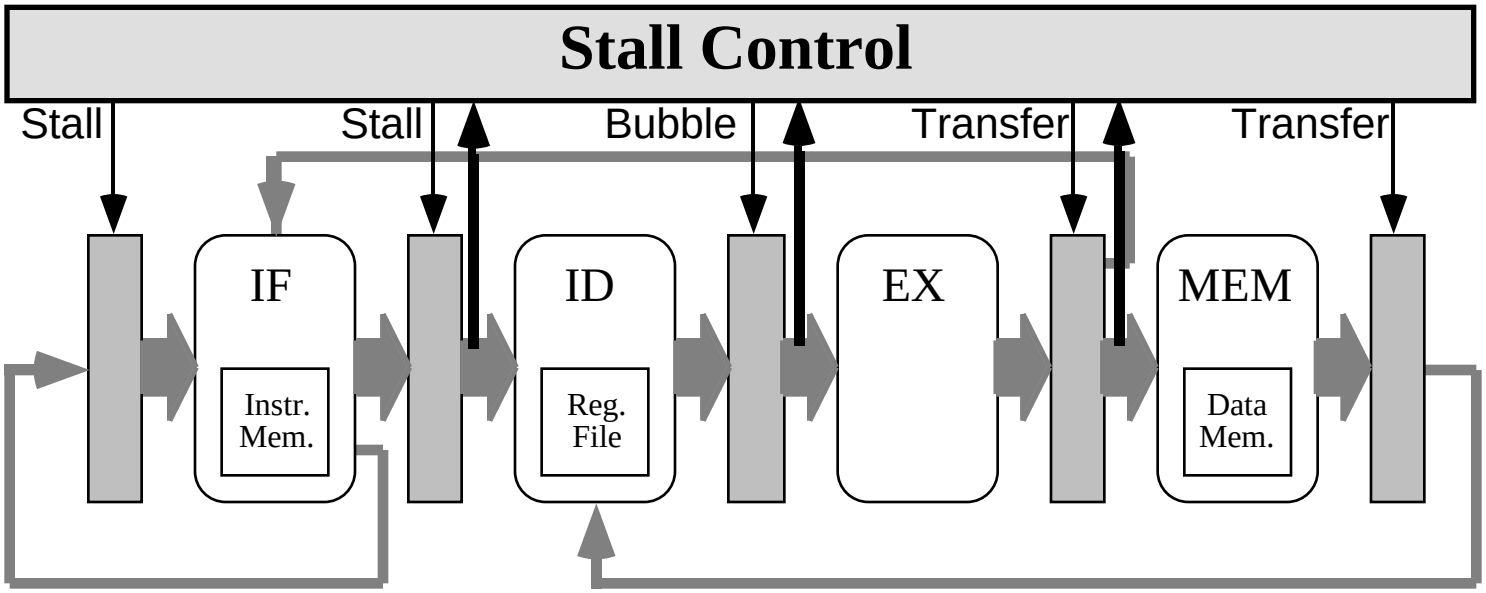
Pending Register Reads

- **By instruction in ID**
- **ID_in.IR[25:21]: Operand A**
- **ID_in.IR[20:16]: Operand B**
  - Only for RR

Pending Register Writes

- **EX_in.WDst: Destination register of instruction in EX**
- **MEM_in.WDst: Destination register of instruction in MEM**

Implementing Stalls



Stall Control Logic

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

Rule:

- **Stall in ID if either pending read matches either pending write**
  - Also stall IF; bubble EX

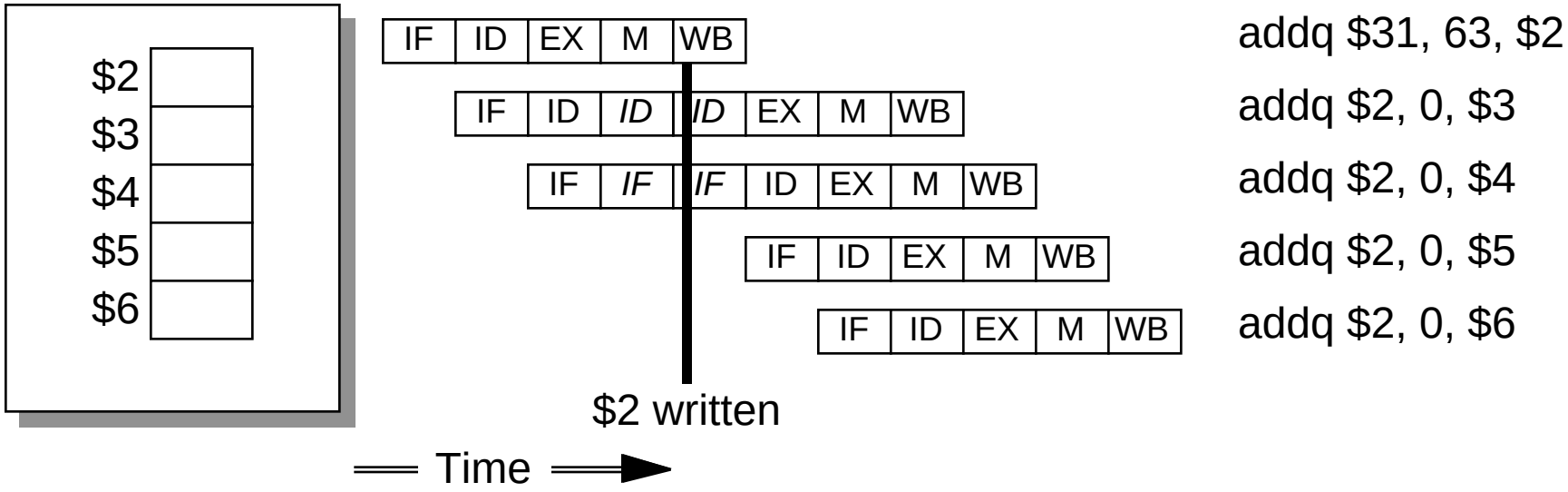
Effect

- **Instructions with pending writes allowed to complete before instruction allowed out of ID**

Stalling for Data Hazards

Operation

- **First instruction progresses unimpeded**
- **Second waits in ID until first hits WB**
- **Third waits in IF until second allowed to progress**



Observations on Stalling

Good

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

Bad

- **As if placed NOPs in code**
  - Except that does not waste instruction memory

Reality

- **Some problems can only be dealt with by stalling**
  - Instruction cache miss
  - Data cache miss
- **Otherwise, want technique with better performance**

Forwarding (Bypassing)

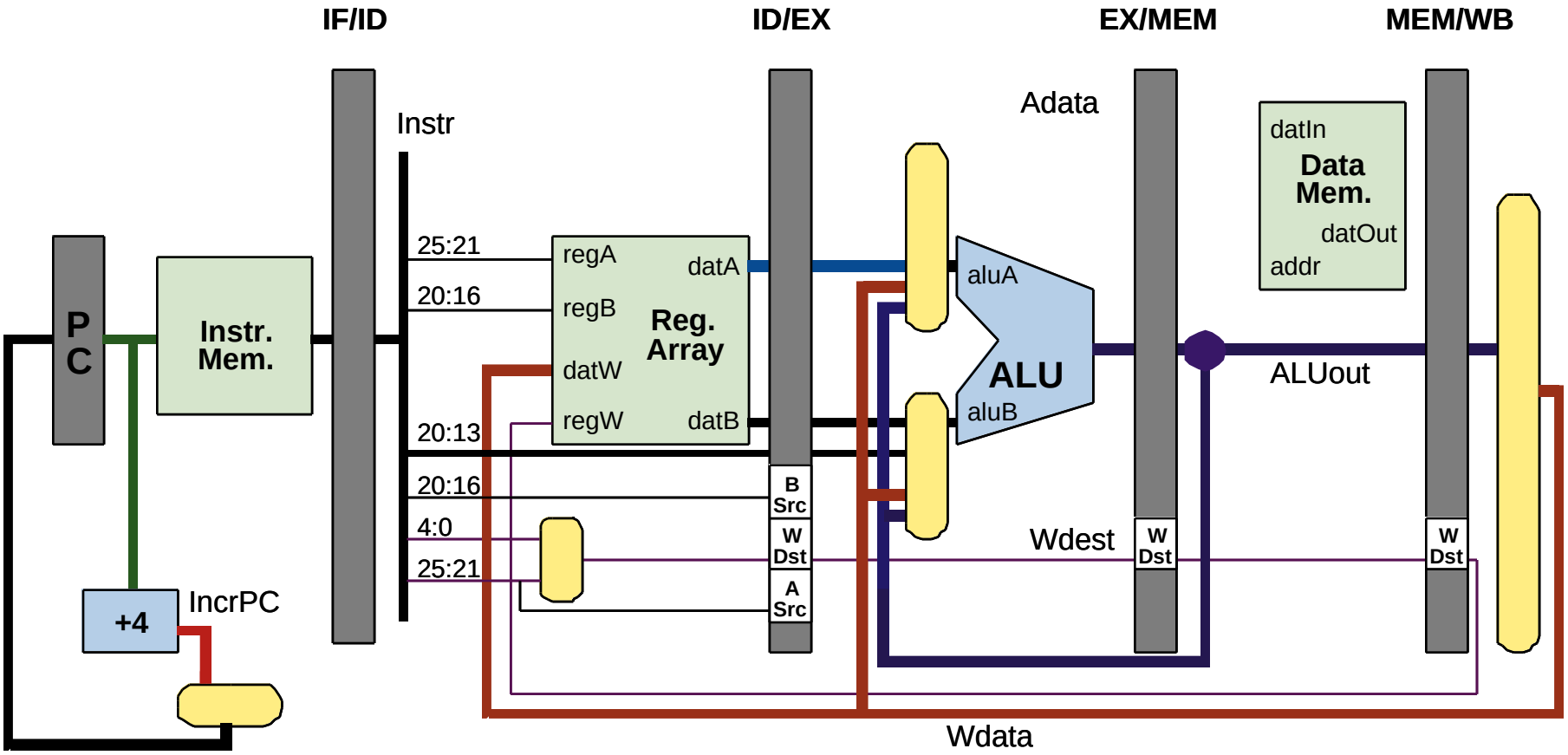
Observation

- **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 Instructions



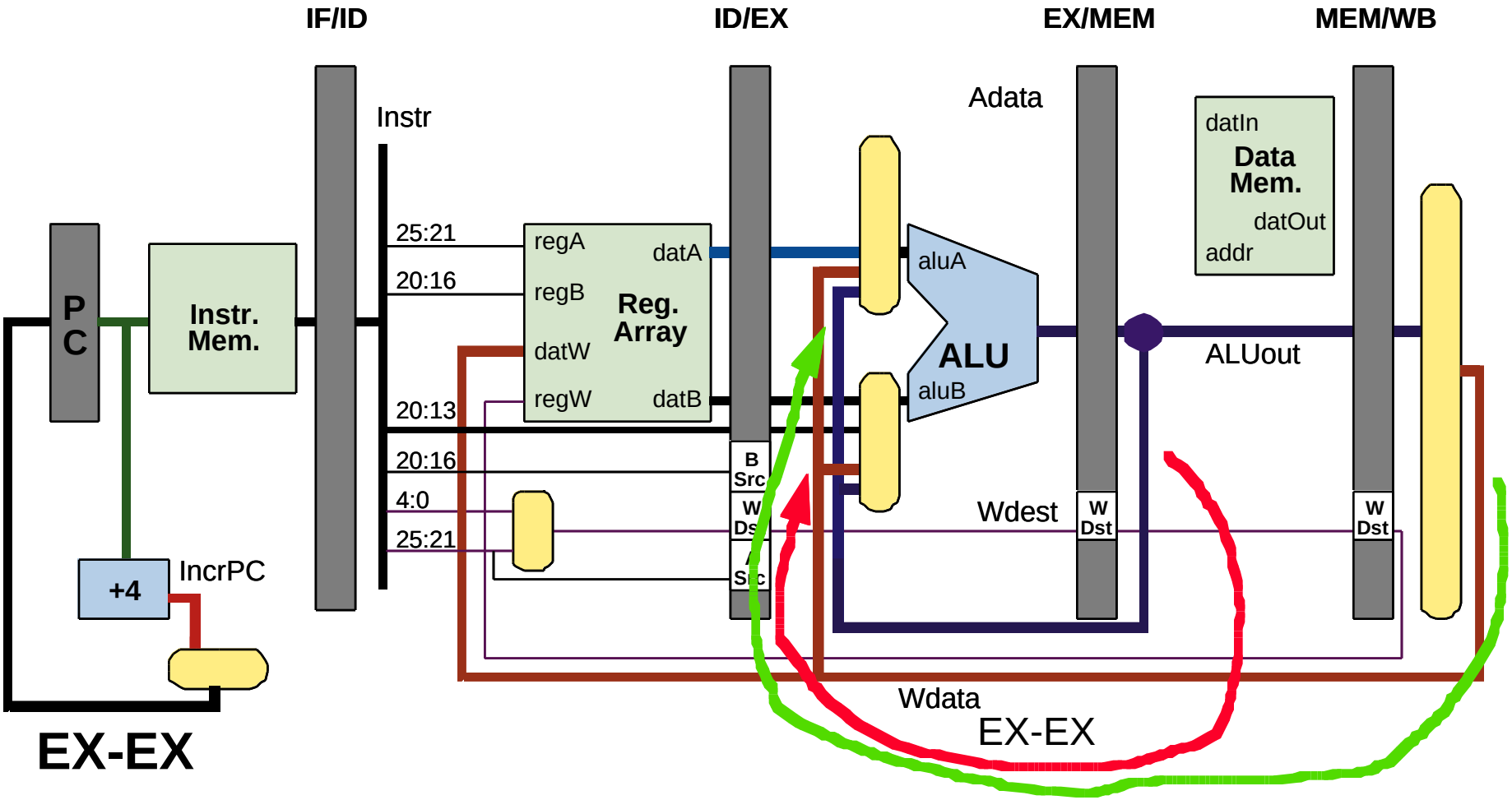
Operand Destinations

- **ALU input A**
  - Register EX_in.ASrc
- **ALU input B**
  - Register EX_in.BSrc

Operand Sources

- **MEM_in.ALUout**
  - Pending write to MEM_in.WDst
- **WB_in.ALUout**
  - Pending write to WB_in.WDst

Bypassing Possibilities



EX-EX

- **From instruction that just finished EX**

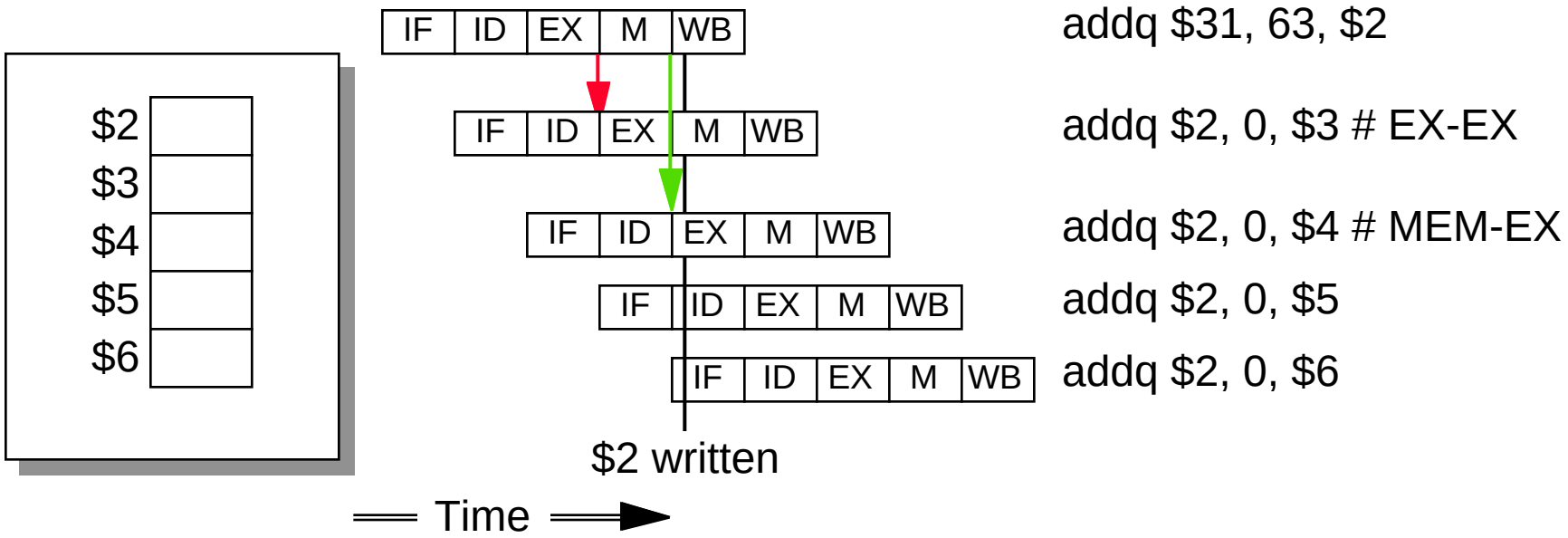
MEM-EX

- **From instruction that finished EX two cycles earlier**

Bypassing Data Hazards

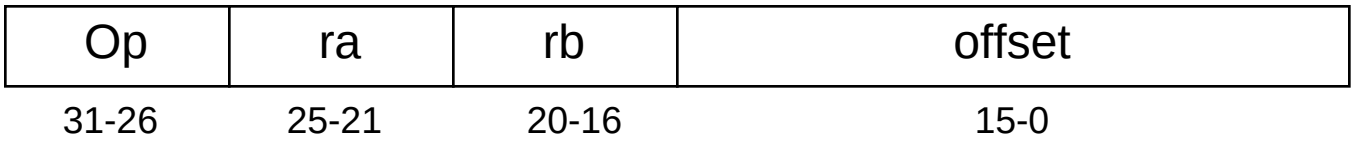
Operation

- **First instruction progresses down pipeline**
- **When in MEM, forward result to second instruction (in EX)**
  - EX-EX forwarding
- **When in WB, forward result to third instruction (in EX)**
  - MEM-EX forwarding

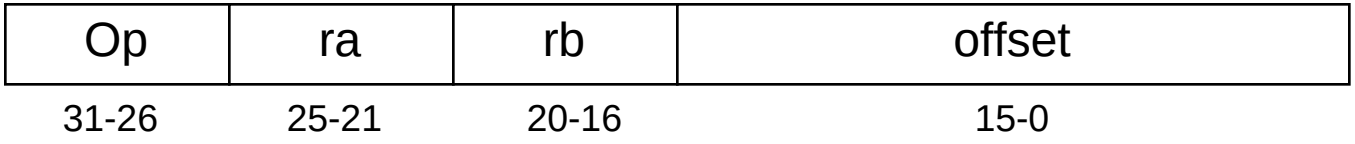


Load & Store Instructions

Load: Ra <-- Mem[Rb +offset]



Store: Mem[Rb +offset] <-- Ra



ID: Instruction decode/register fetch

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

MEM: Memory

- **Load: Mem-Data <-- DMemory[ALUOutput]**
- **Store: DMemory[ALUOutput] <-- A**

WB: Write back

- **Load: Register[IR[25:21]] <-- Mem-Data**

Some Hazards with Loads & Stores

Data Generated by Load

Load-ALU

```
ldq $1, 8($2)
addq $2, $1, $2
```

Load-Store Data

```
ldq $1, 8($2)
stq $1, 16($2)
```

Load-Store (or Load) Addr.

```
ldq $1, 8($2)
stq $2, 16($1)
```

Data Generated by Store

Store-Load Data

```
stq $1, 8($2)
ldq $3, 8($2)
```

Not a concern for us

Data Generated by ALU

ALU-Store (or Load) Addr

```
addq $1, $3, $2
stq $3, 8($2)
```

ALU-Store Data

```
addq $2, $3, $1
stq $1, 16($2)
```

Analysis of Data Transfers

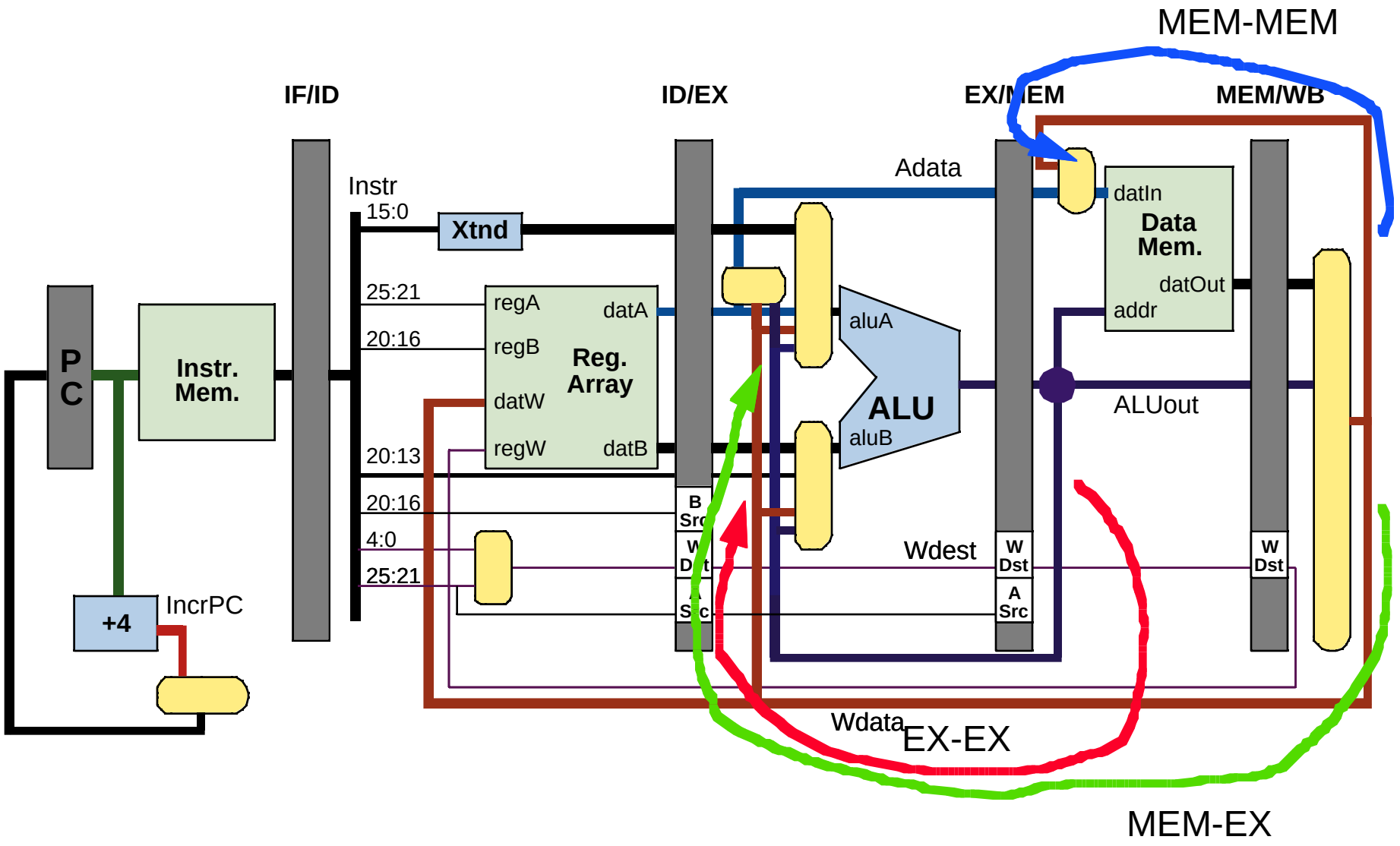
Data Sources

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

Complete Bypassing for ALU & L/S



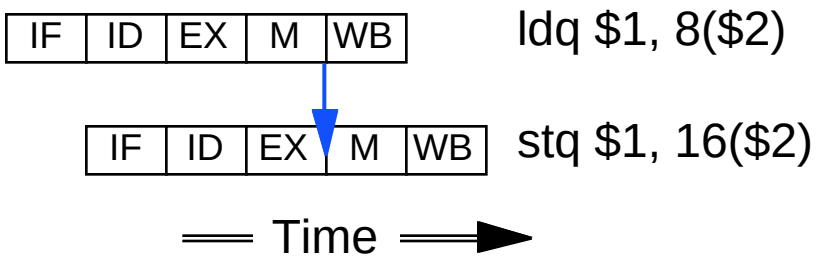
MEM-MEM Forwarding

Condition

- **Data generated by load instruction**
  - Register WB_in.WDst
- **Used by immediately following store**
  - Register MEM_in.ASrc

Load-Store Data

```
ldq $1, 8($2)
stq $1, 16($2)
```



Simulator Data Hazard Examples

- **demo5.O**

```
0x0:   43e7f402 addq    r31, 0x3f, r2      # $2 = 0x3F
0x4:   44420403 bis     r2, r2, r3         # $3 = 0x3F EX-EX
0x8:   47ff041f bis     r31, r31, r31
0xc:   47ff041f bis     r31, r31, r31
0x10:  43e1f402 addq    r31, 0xf, r2       # $2 = 0xF
0x14:  47ff041f bis     r31, r31, r31
0x18:  44420403 bis     r2, r2, r3         # $3 = 0xF MEM-EX
0x1c:  47ff041f bis     r31, r31, r31
0x20:  43e11403 addq    r31, 0x8, r3       # $3 = 8
0x24:  43e21402 addq    r31, 0x10, r2      # $2 = 0x10
0x28:  b4620000 stq     r3, 0(r2)          # Mem[0x10] = 8 MEM-EX, EX-EX
0x2c:  47ff041f bis     r31, r31, r31
0x30:  a4830008 ldq     r4, 8(r3)          # $4 = 8
0x34:  40820405 addq    r4, r2, r5         # $5 = 0x18  Stall 1, MEM-EX
0x38:  47ff041f bis     r31, r31, r31
0x3c:  00000000 call_pal halt
```

Impact of Forwarding

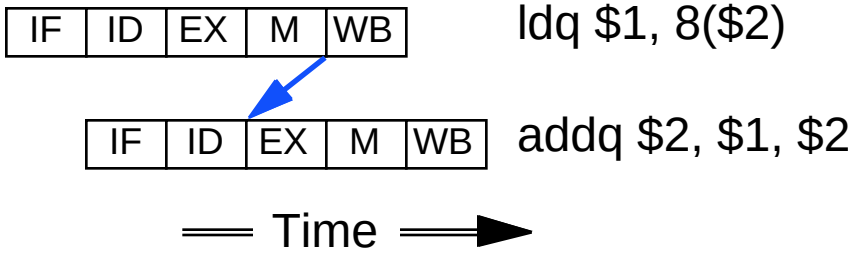
Single Remaining Unsolved Hazard Class

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

Load-ALU

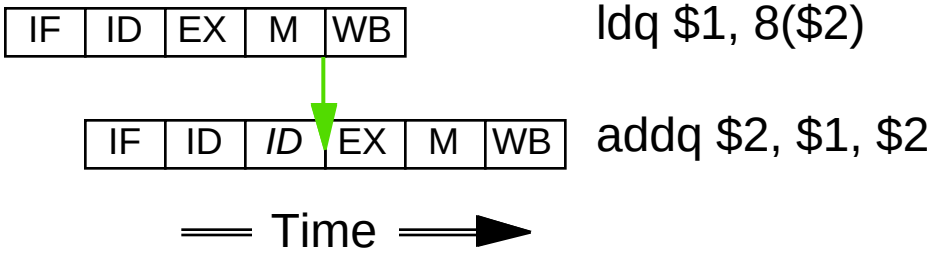
```
ldq $1, 8($2)
addq $2, $1, $2
```

Just Forward?



Value not available soon enough!

With 1 Cycle Stall



Then can use MEM-EX forwarding

Methodology for characterizing and Enumerating Data Hazards

OP	writes	reads
RR	rc	ra, rb
RI	rc	ra
Load	ra	rb
Store		ra, rb

The space of data hazards (from a program-centric point of view) can be characterized by 3 independent axes:

3 possible write regs (axis 1):
RR.rc, RI.rc, Load.ra

6 possible read regs (axis 2):
RR.ra, RR.rb, RI.ra, Load.ra, Store.ra, Store.rb

A dependent read can be a distance of either 1 or 2 from the corresponding write (axis 3):

distance 2 hazard:
RR.rc/RR.ra/2

addq $31, 63, **$2**
addq $31, **$2**, $3
addu **$2**, $31, $4

distance 1 hazard:
RR.rc/RR.rb/1

Enumerating data hazards

distance = 1

writes \ reads	RR.ra	RR.rb	RI.ra	L.rb	S.ra	S.rb
RR.rc						
RI.rc						
L.ra						

distance = 2

writes \ reads	RR.ra	RR.rb	RI.ra	L.rb	S.ra	S.rb
RR.rc						
RI.rc						
L.ra						

Testing Methodology

- **36 cases to cover all interactions between RR, RI, Load, & Store**
- **Would need to consider more read source and write destinations when add other instruction types**

Simulator Microtest Example

```
0x0:   43e21402  addq     r31, 0x10, r2   $2 = 0x10
0x4:   47ff041f  bis      r31, r31, r31
0x8:   47ff041f  bis      r31, r31, r31
0xc:   43e20405  addq     r31, r2, r5     # $5 = 0x10
0x10:  43e50401  addq     r31, r5, r1     # $1 = 0x10
0x14:  47ff041f  bis      r31, r31, r31
0x18:  47ff041f  bis      r31, r31, r31
0x1c:  47ff041f  bis      r31, r31, r31
0x20:  44221803  xor      r1, 0x10, r3    # $1 should == 0
0x24:  47ff041f  bis      r31, r31, r31
0x28:  47ff041f  bis      r31, r31, r31
0x2c:  e4600006  beq      r3, 0x48        # Should take
0x30:  47ff041f  bis      r31, r31, r31
0x34:  47ff041f  bis      r31, r31, r31
0x38:  00000000  call_pal halt            # Failure
0x3c:  47ff041f  bis      r31, r31, r31
0x40:  47ff041f  bis      r31, r31, r31
0x44:  47ff041f  bis      r31, r31, r31
0x48:  00000001  call_pal cflush          # Success
```

demo7.O

- **Tests for single failure mode**
  - ALU Rc --> ALU Ra
  - distance 1
  - RR.rc/RR.ra/1
- **Hits `call_pal 0` when error**
- **Jumps to `call_pal 1` when OK**
- **Error case shields successful case**
- **Grep for ERROR or `call_pal 0`**