Advanced Pipelining
CS740

Sept. 29, 1998

Topics

- **Data Hazards**
  - Stalling and Forwarding
  - Systematic testing of hazard-handling logic
- **Control Hazards**
  - Stalling, Predict not taken
- **Exceptions**
- **Multicycle Instructions**

Alpha ALU Instructions

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

Op	ra	rb	000	0	funct	rc
31-26	25-21	20-16	15-13	12	11-5	4-0

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

Op	ra	ib	1	funct	rc
31-26	25-21	20-13	12	11-5	4-0

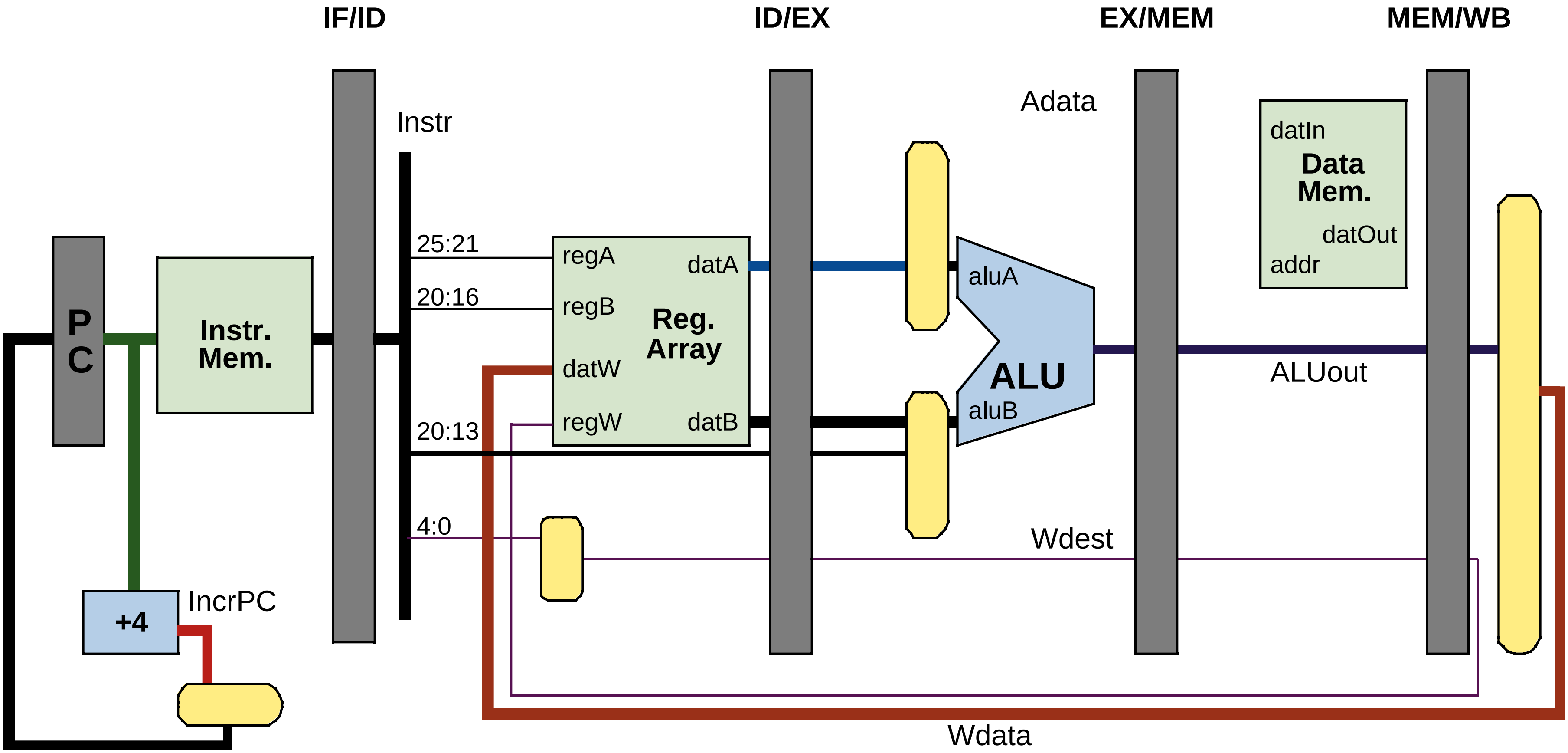
Encoding

- **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

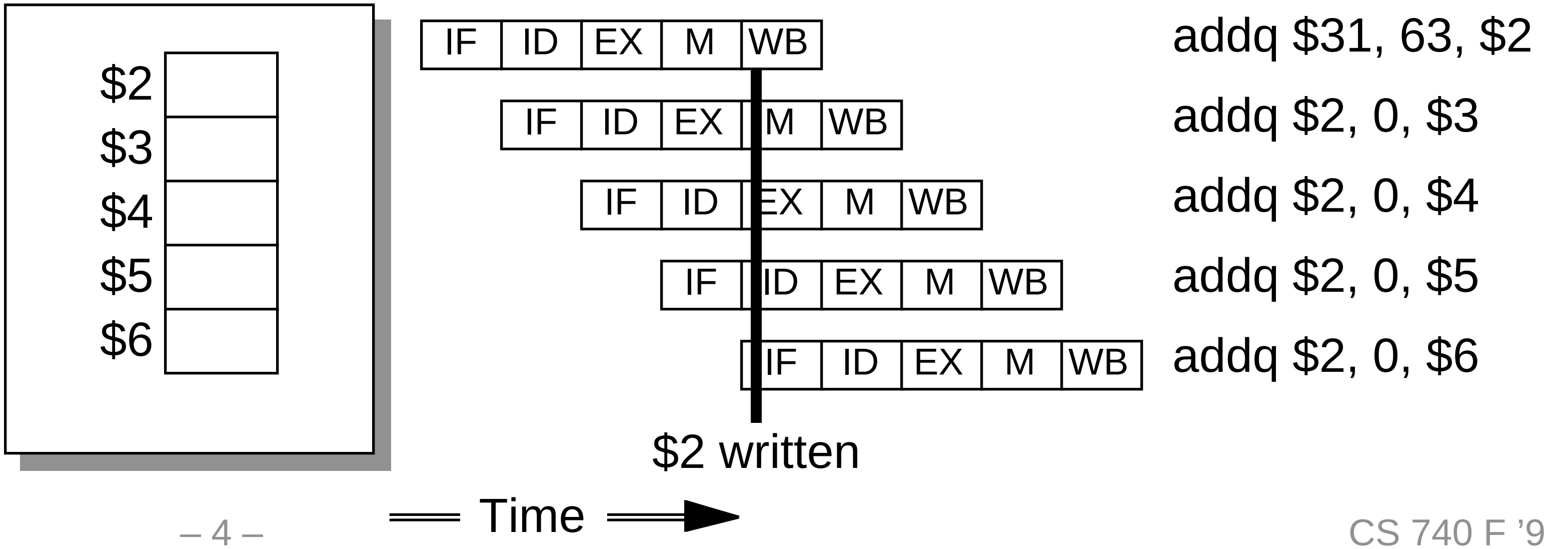
IF	ID	EX	MEM	WB
instruction fetch	instruction decode/ register fetch	execute	memory access	write back



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:**



$2
$3
$4
$5
$6

IF ID EX M WB addq $31, 63, $2
IF ID EX M WB addq $2, 0, $3
IF ID EX M WB addq $2, 0, $4
IF ID EX M WB addq $2, 0, $5
IF ID EX M WB addq $2, 0, $6

$2 written

Time

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
```

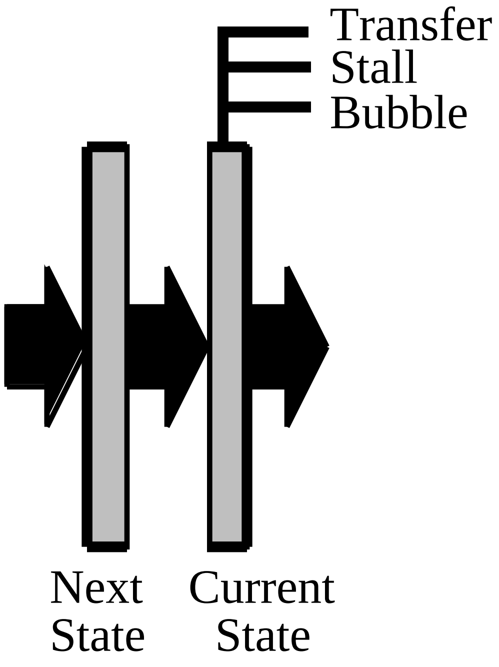
Handling Hazards by Stalling

Idea

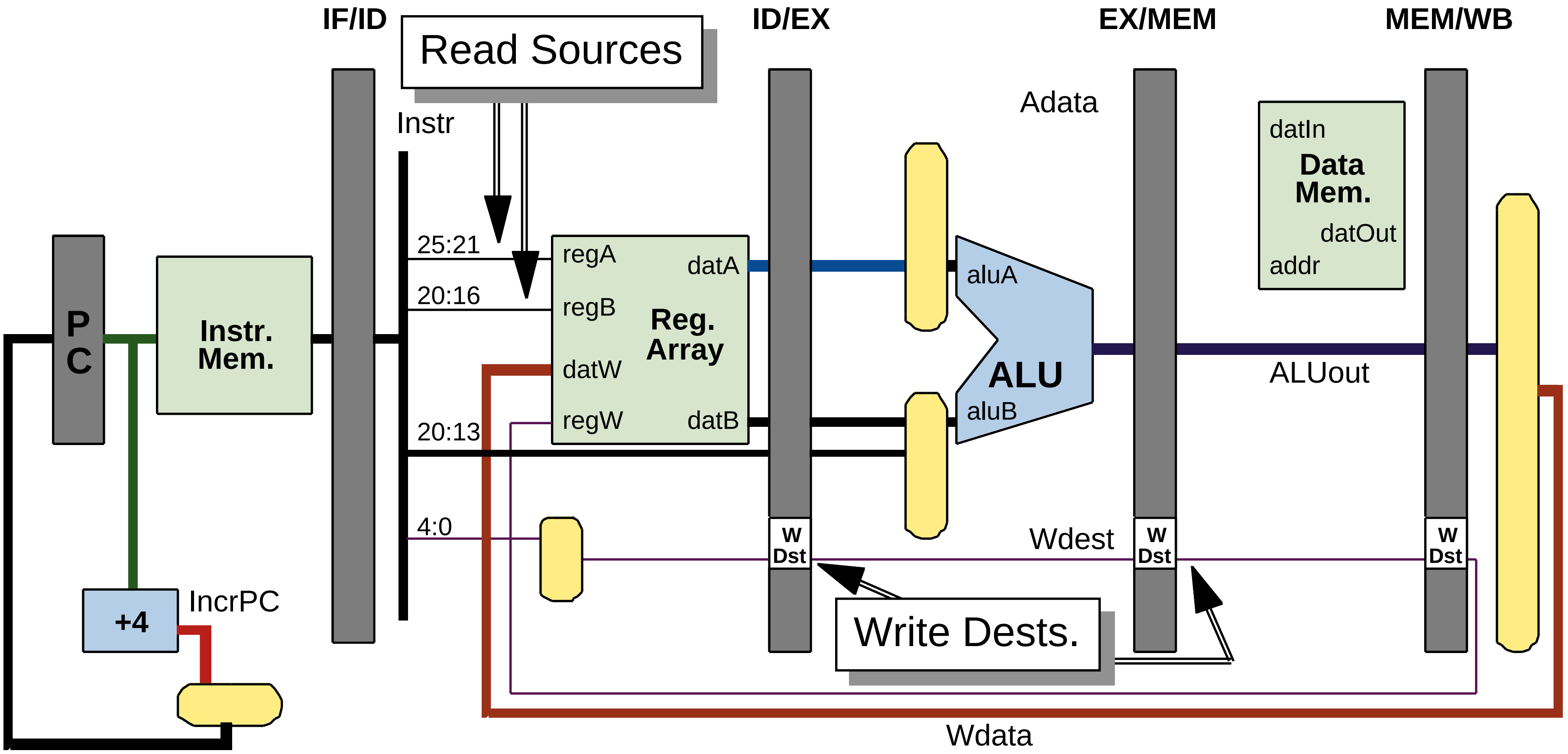
- **Delay instruction until hazard eliminated**
- **Put “bubble” into pipeline**
  - Dynamically generated NOP

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



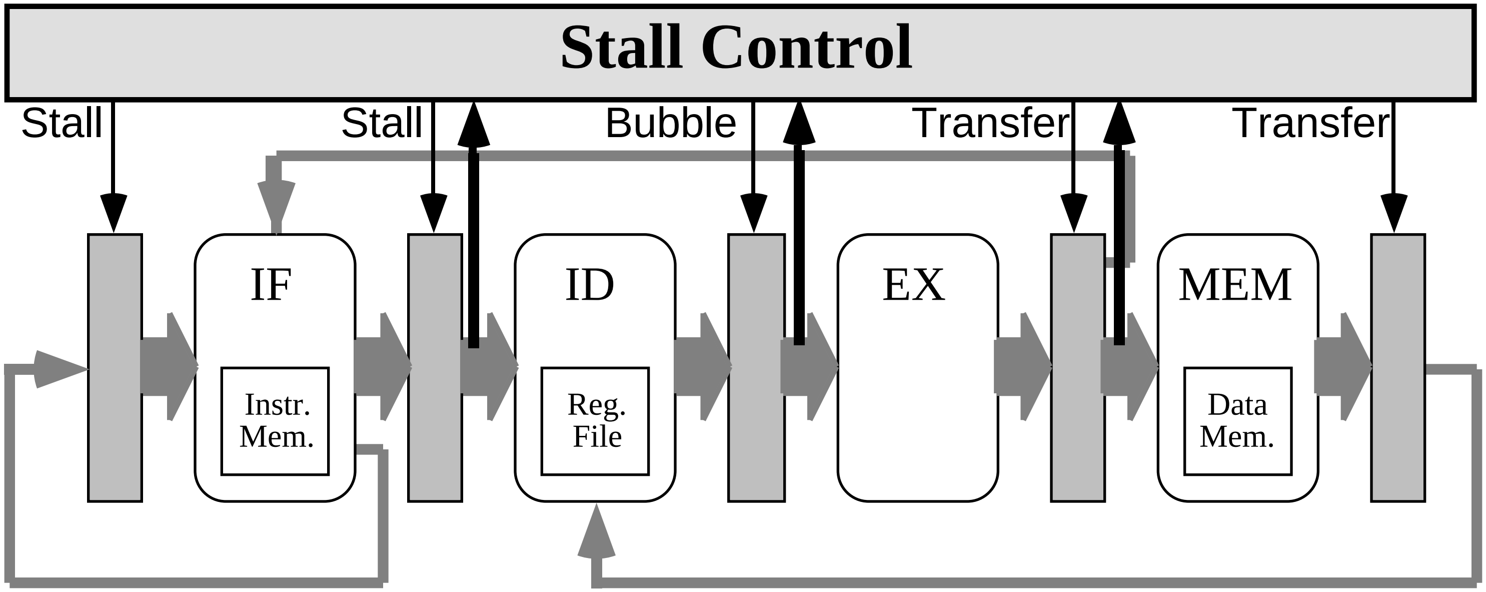
Pending Register Reads

- 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

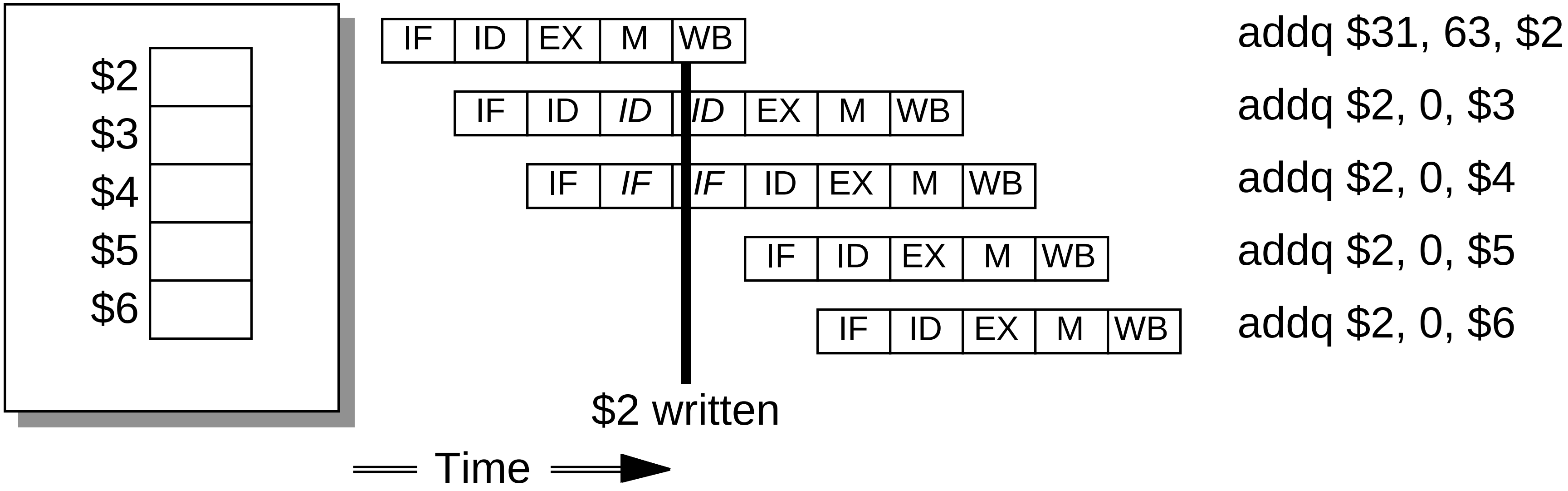
Effect

- **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**



IF	ID	EX	M	WB							addq $31, 63, $2
	IF	ID	*ID*	*ID*	EX	M	WB				addq $2, 0, $3
		IF	*IF*	*IF*	ID	EX	M	WB			addq $2, 0, $4
					IF	ID	EX	M	WB		addq $2, 0, $5
						IF	ID	EX	M	WB	addq $2, 0, $6

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)

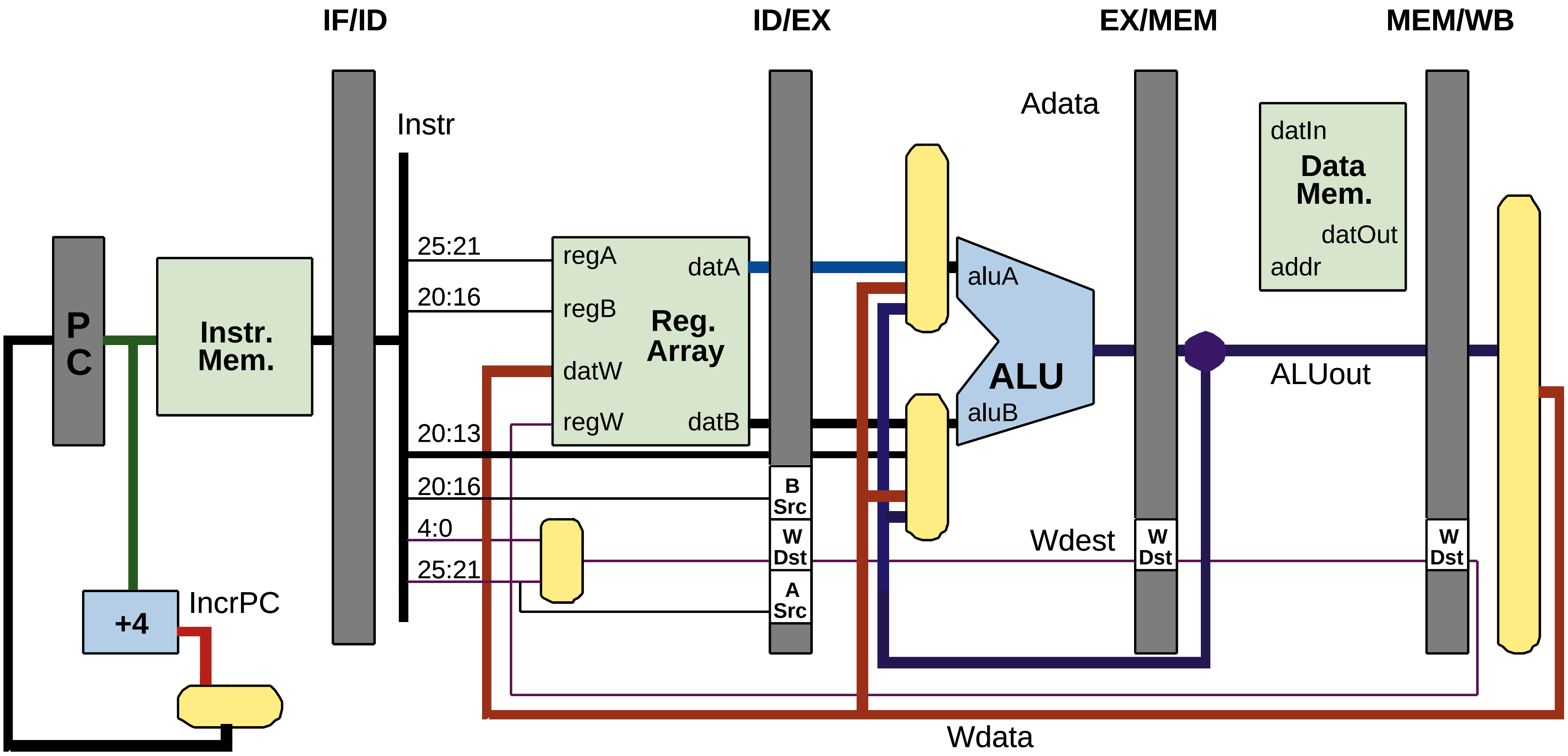
Observation

- **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



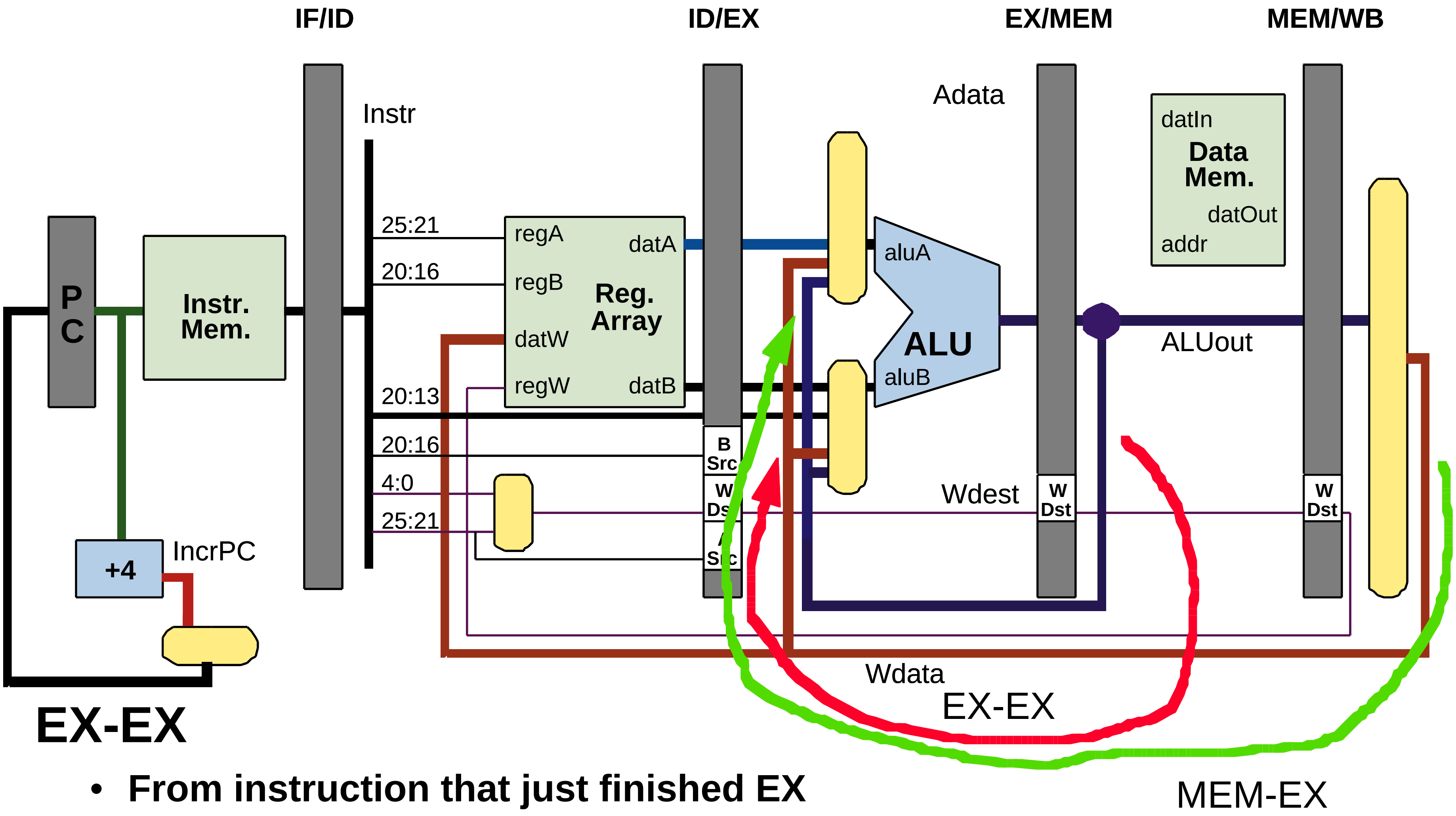
Operand Destinations

- **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

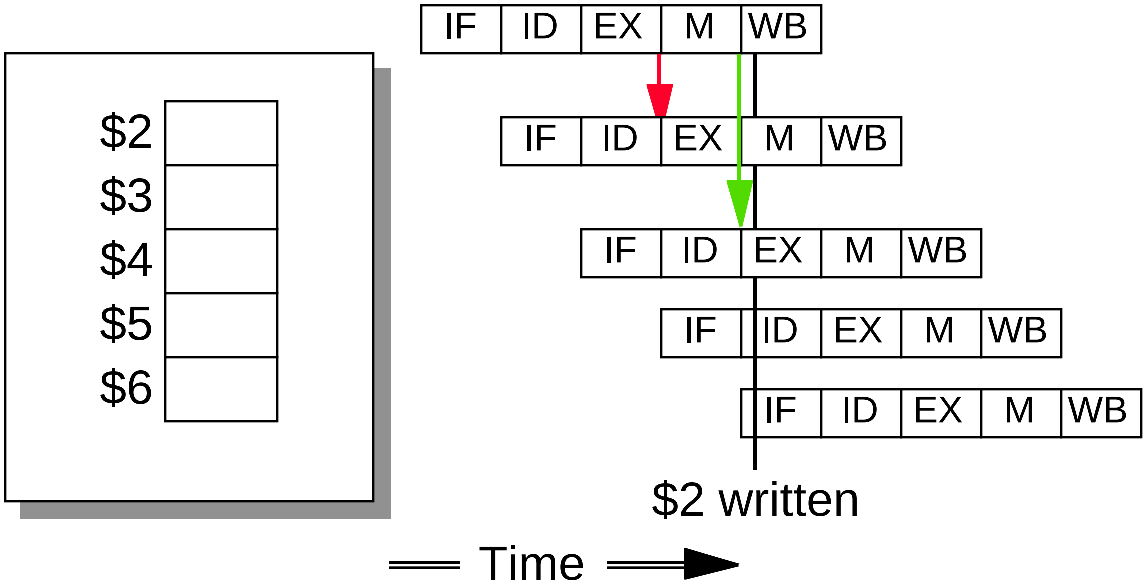
MEM-EX

- 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



$2
$3
$4
$5
$6

IF ID EX M WB

IF ID EX M WB

IF ID EX M WB

IF ID EX M WB

IF ID EX M WB

$2 written

Time

addq $31, 63, $2

addq $2, 0, $3 # EX-EX

addq $2, 0, $4 # MEM-EX

addq $2, 0, $5

addq $2, 0, $6

Load & Store Instructions

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

Op	ra	rb	offset
31-26	25-21	20-16	15-0

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

Op	ra	rb	offset
31-26	25-21	20-16	15-0

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

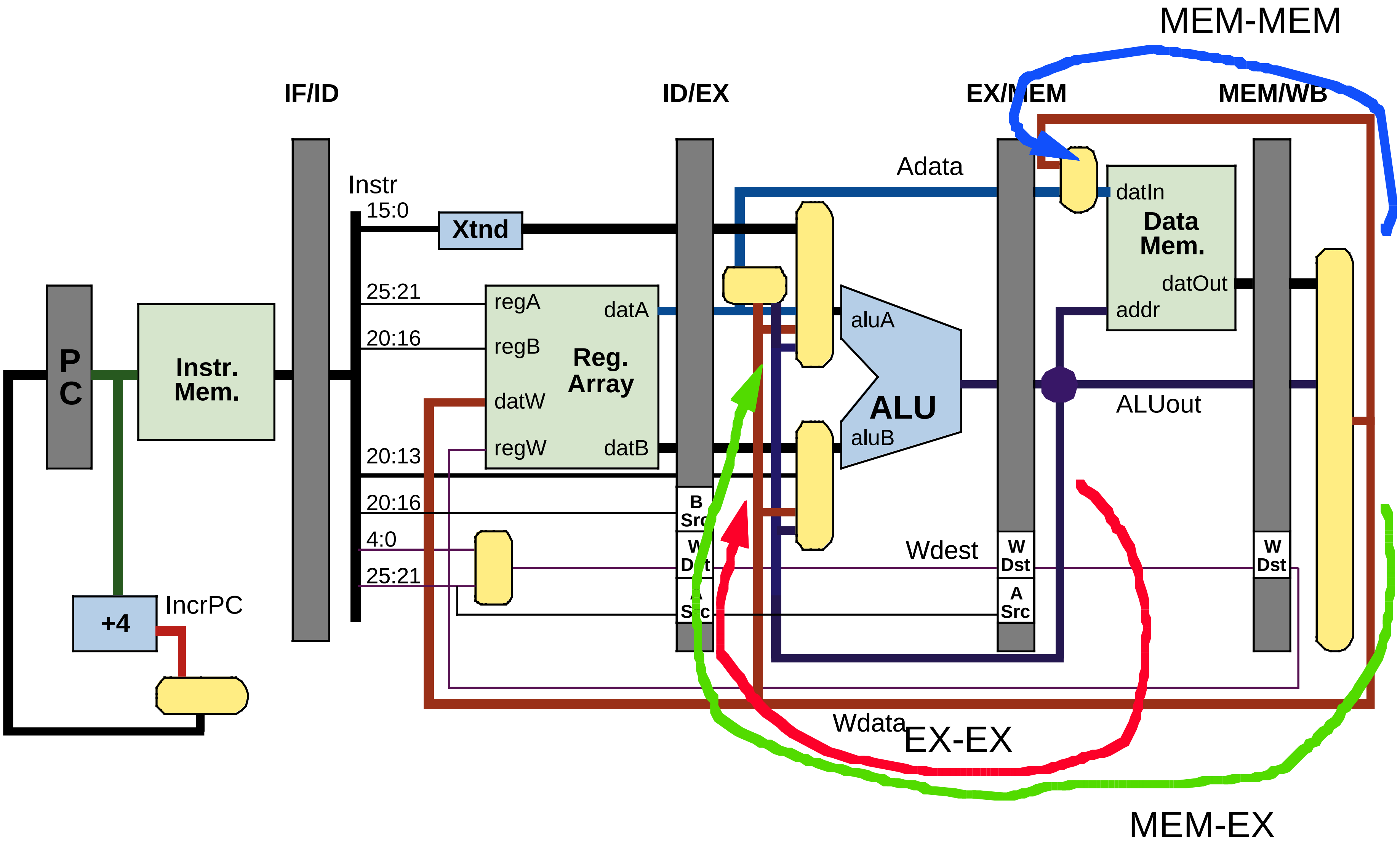
Data Sources

- **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



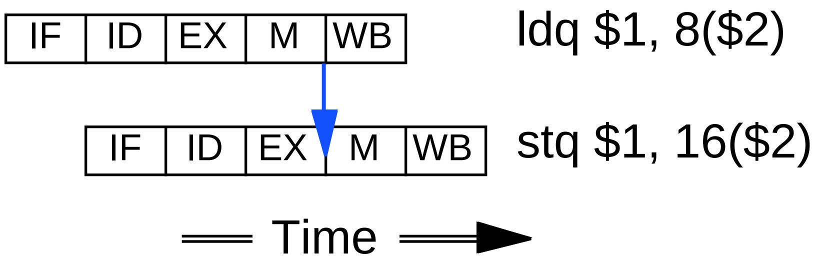
MEM-MEM Forwarding

Condition

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

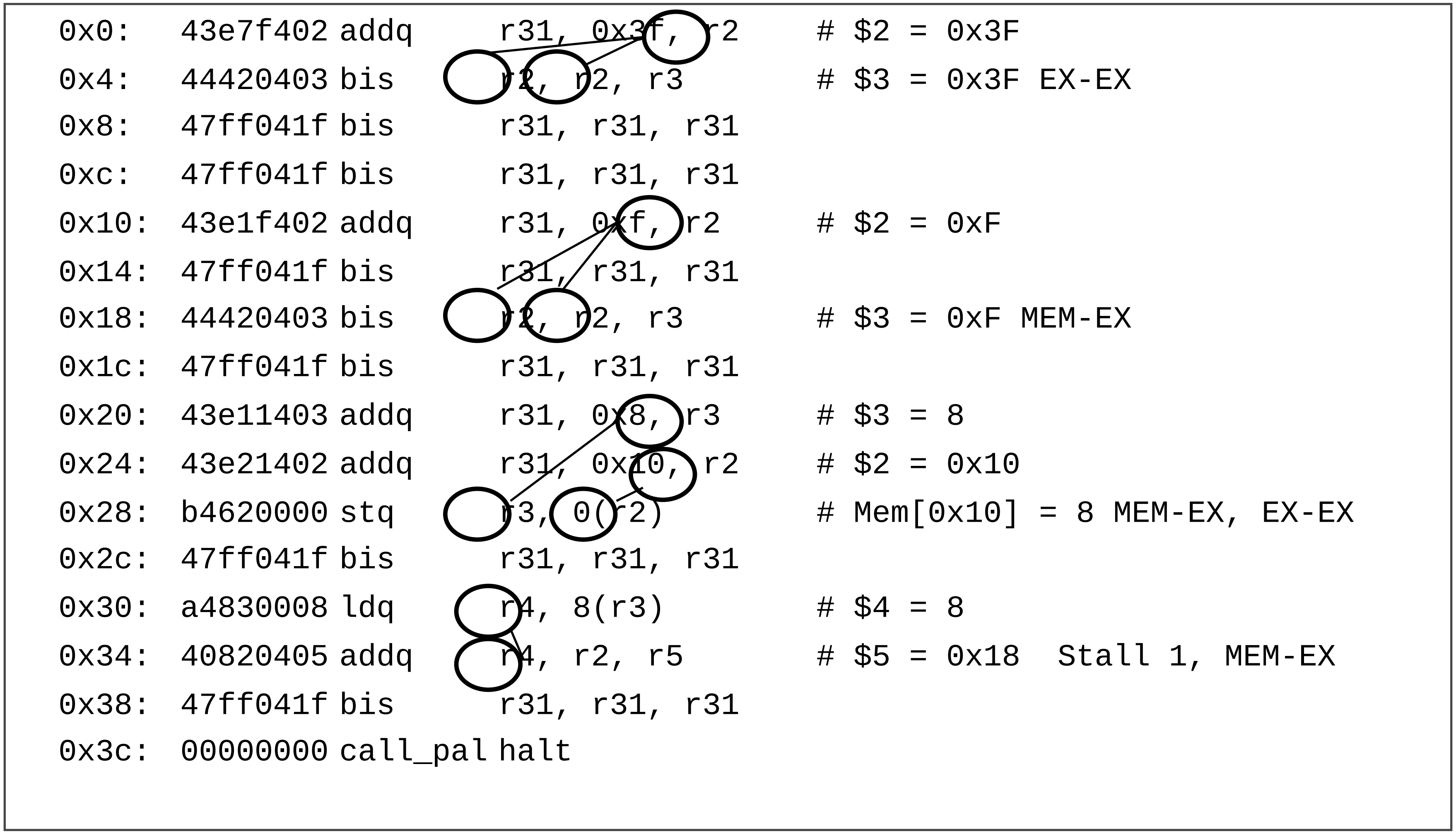
Load-Store Data

```
        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

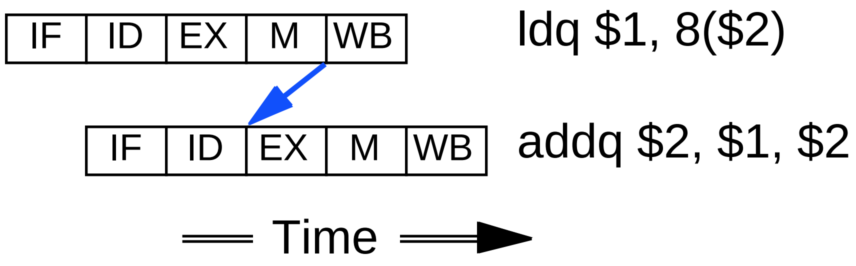
Single Remaining Unsolved Hazard Class

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

Load-ALU

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

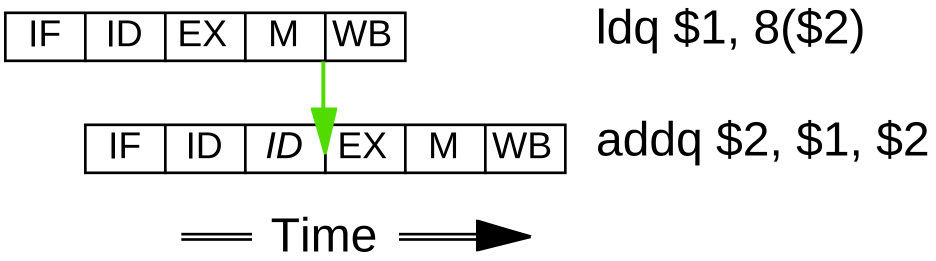
Just Forward?



IF	ID	EX	M	WB		ldq $1, 8($2)
	IF	ID	EX	M	WB	addq $2, $1, $2

Value not available soon enough!

With 1 Cycle Stall



IF	ID	EX	M	WB			ldq $1, 8($2)
	IF	ID	*ID*	EX	M	WB	addq $2, $1, $2

Then can use MEM-EX forwarding

Methodology for characterizing and Enumerating Data Hazards

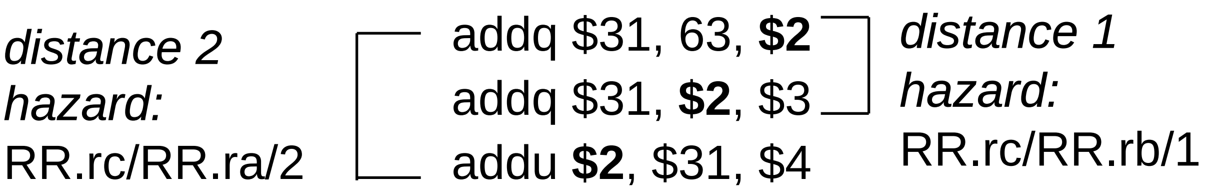
OP	writes	reads
RR	rc	ra, rb
RI	rc	ra
Load	ra	rb
Store		ra, rb

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

reads

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

distance = 2

reads

writes	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`**

Branch Instructions

Cond. Branch: PC <-- Cond(Ra) ? PC + 4 + disp*4 : PC + 4

Op	ra	disp
31-26	25-21	20-0

Sources

- **PC, Ra**

Destinations

- **PC**

Branch [Subroutine] (br, bsr): Ra <-- PC + 4; PC <-- PC + 4 + disp*4

Op	ra	disp
31-26	25-21	20-0

Sources

- **PC**

Destinations

- **PC, Ra**

New Data Hazards

Branch Uses Register Data

- Generated by ALU instruction
- Read from register in ID

Handling

- Same as other instructions with register data source
- Bypass
  - EX-EX
  - MEM-EX

ALU-Branch

```
addq $2, $3, $1
beq $1, targ
```

Distant ALU-Branch

```
addq $2, $3, $1
bis $31, $31, $31
beq $1, targ
```

Load-Branch

```
lw $1, 8($2)
beq $1, targ
```

Jump Instructions

jmp, jsr, ret: Ra <-- PC+4; PC <-- Rb

0x1A	ra	rb	Hint
31-26	25-21	20-16	15-0

Sources

- **PC, Rb**

Destinations

- **PC, Ra**

Still More Data Hazards

Jump Uses Register Data

- Generated by ALU instruction
- Read from register in ID

Handling

- Same as other instructions with register data source
- Bypass
  - EX-EX
  - MEM-EX

ALU-Jump

```
addq $2, $3, $1
jsr $26 ($1), 1
```

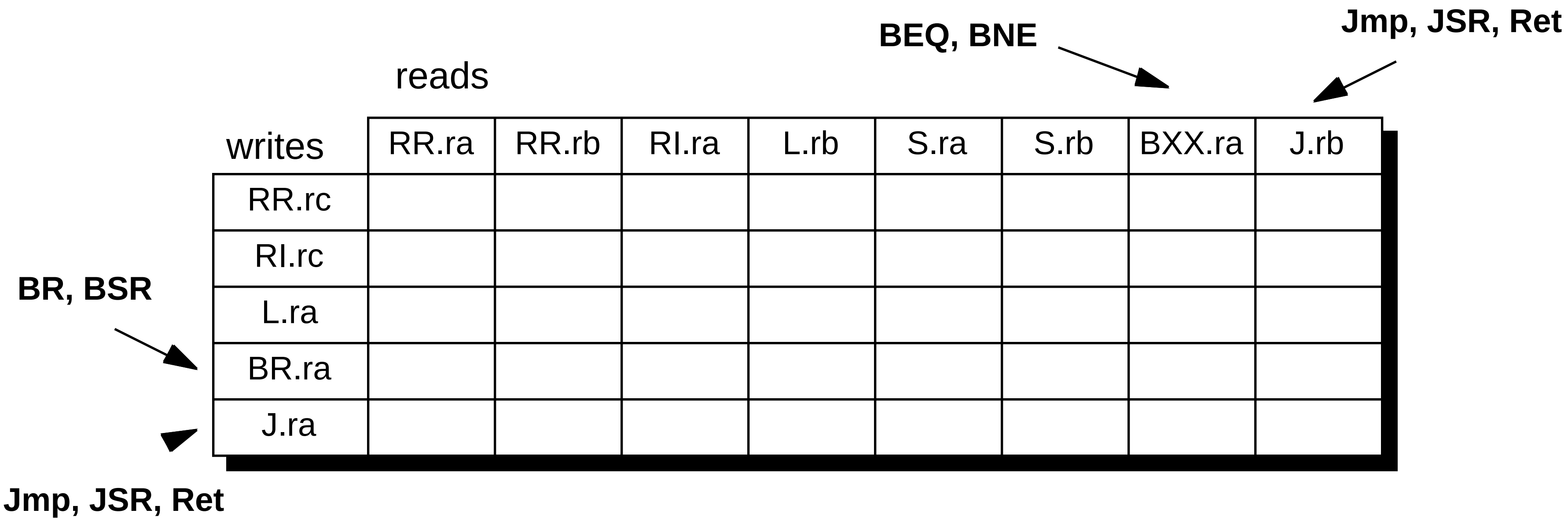
Distant ALU-Jump

```
addq $2, $3, $1
bis $31, $31, $31
jmp $31 ($1), 1
```

Load-Jump

```
lw $26, 8($sp)
ret $31 ($26), 1
```

Enumerating data hazards



reads

writes	RR.ra	RR.rb	RI.ra	L.rb	S.ra	S.rb	BXX.ra	J.rb
RR.rc								
RI.rc								
L.ra								
BR.ra								
J.ra								

Cases

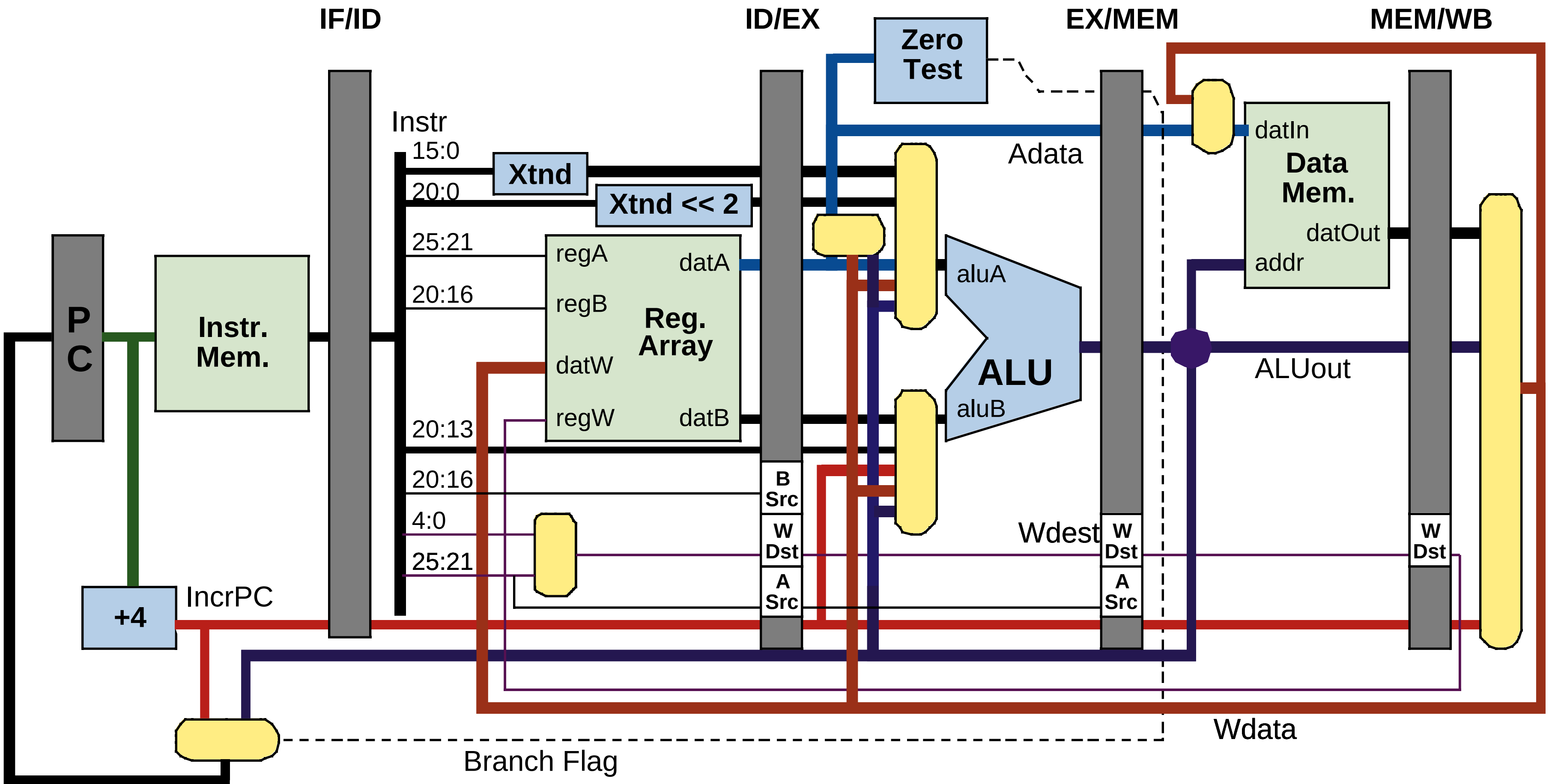
- 2 distances (either 1 or 2)
- 5 classes of writer
- 8 classes of readers

Testing Methodology

- 80 cases to cover all interactions between supported instruction types

Pipelined datapath

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

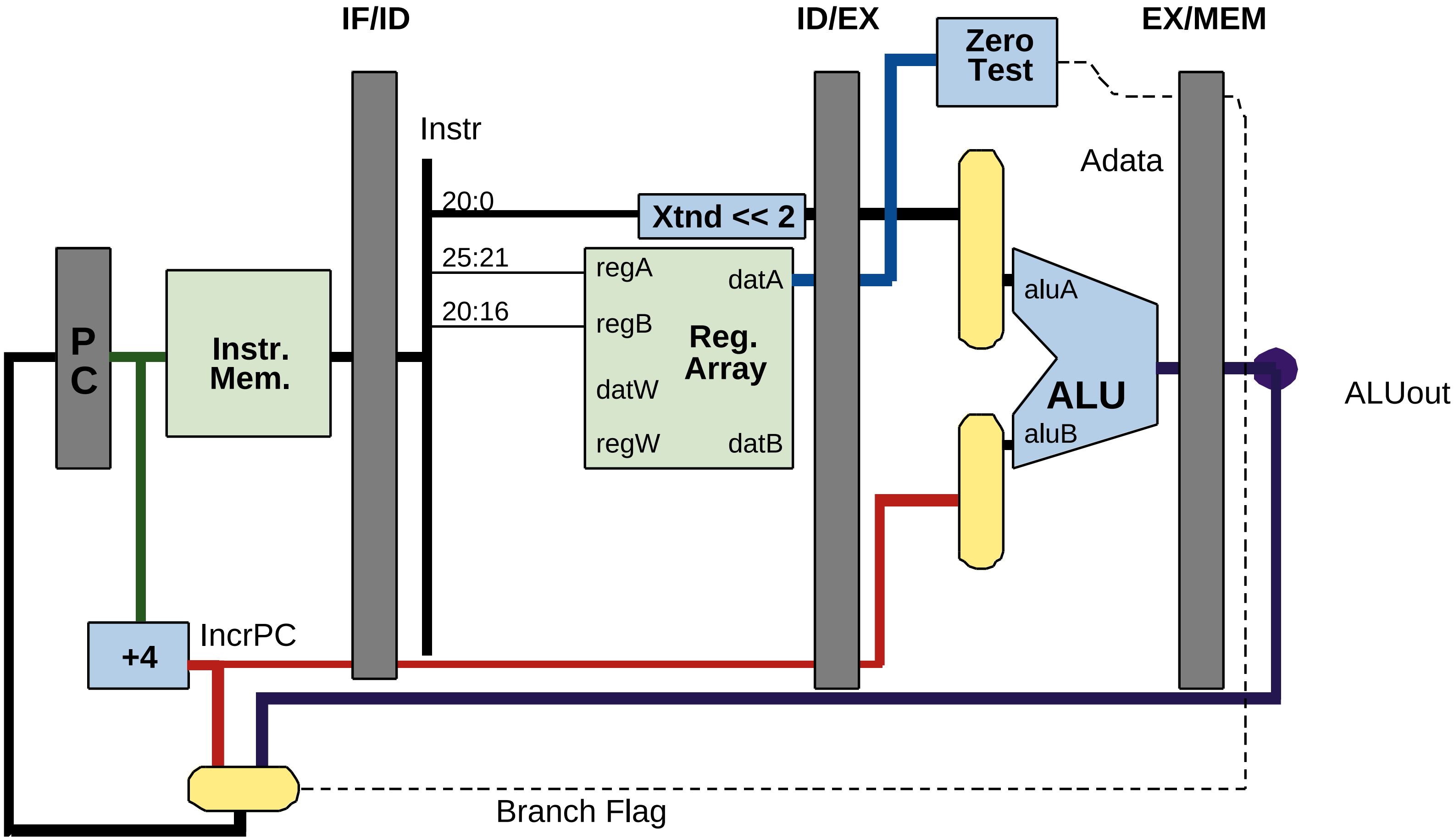


What happens with a branch?

Conditional Branch Instruction Handling

beq: PC <-- Ra == 0 ? PC + 4 + disp*4 : PC + 4

Op	ra	disp
31-26	25-21	20-0



Branch on equal

beq: PC <-- Ra == 0 ? PC + 4 + disp*4 : PC + 4

0x39	ra	disp
31-26	25-21	20-0

IF: Instruction fetch

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

ID: Instruction decode/register fetch

- A <-- Register[IR[25:21]]

Ex: Execute

- Target <-- incrPC + SignExtend(IR[20:0]) << 2
- Z <-- (A == 0)

MEM: Memory

- PC <-- Z ? Target : incrPC

WB: Write back

- nop

Branch Example

Desired Behavior

- **Take branch at 0x00**
- **Execute target 0x18**
  - PC + 4 + disp << 2
  - PC = 0x00
  - disp = 5

Displacement

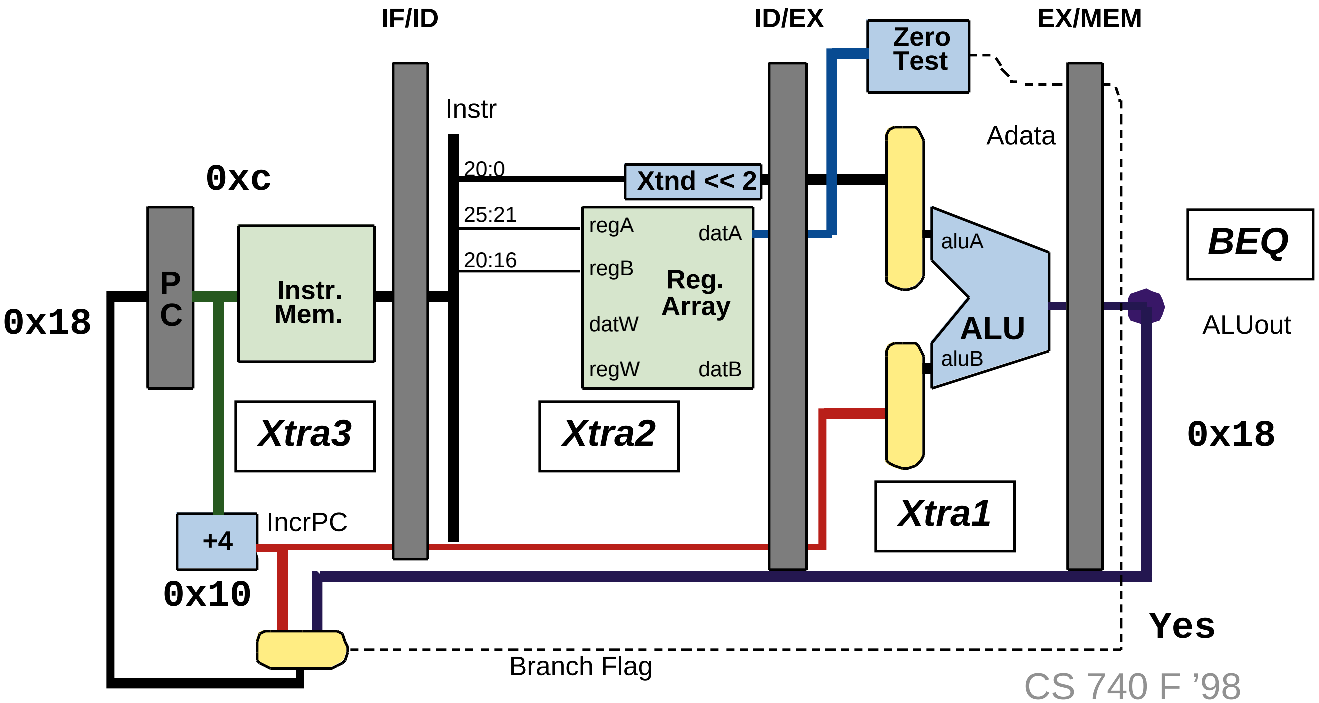
Branch Code (demo08.O)

```
0x0:  e7e00005 beq    r31, 0x18         # Take
0x4:  43e7f401 addq   r31, 0x3f, r1    # (Skip)
0x8:  43e7f402 addq   r31, 0x3f, r2    # (Skip)
0xc:  43e7f403 addq   r31, 0x3f, r3    # (Skip)
0x10: 43e7f404 addq   r31, 0x3f, r4    # (Skip)
0x14: 47ff041f bis    r31, r31, r31
0x18: 43e7f405 addq   r31, 0x3f, r5    # (Target)
0x1c: 47ff041f bis    r31, r31, r31
0x20: 00000000 call_pal                 halt
```

Branch Hazard Example

```
0x0:  beq    r31, 0x18       # Take
0x4:  addq   r31, 0x3f, r1   # Xtra1
0x8:  addq   r31, 0x3f, r2   # Xtra2
0xc:  addq   r31, 0x3f, r3   # Xtra3
0x10: addq   r31, 0x3f, r4   # Xtra4

0x18: addq   r31, 0x3f, r5   # Target
```

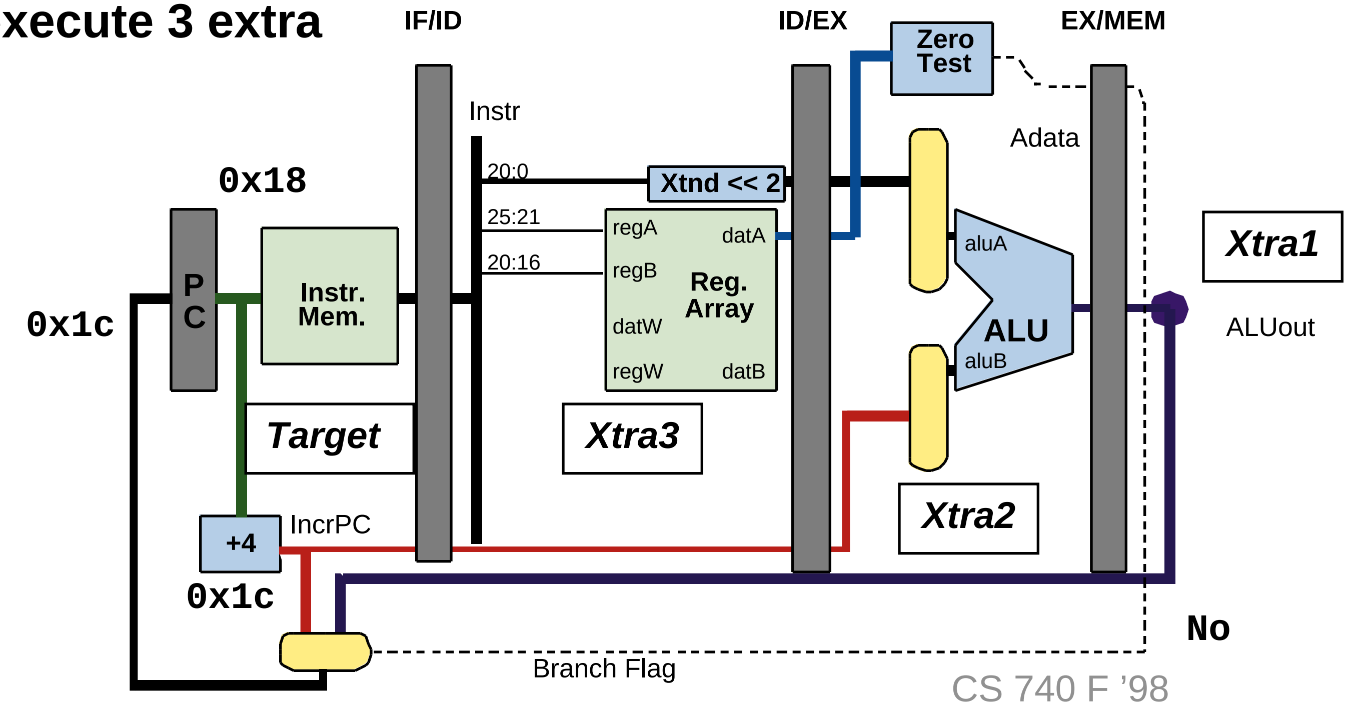
- **With BEQ in Mem stage**



Branch Hazard Example (cont.)

```
0x0:  beq   r31, 0x18      # Take
0x4:  addq  r31, 0x3f, r1  # Xtra1
0x8:  addq  r31, 0x3f, r2  # Xtra2
0xc:  addq  r31, 0x3f, r3  # Xtra3
0x10: addq  r31, 0x3f, r4  # Xtra4

0x18: addq  r31, 0x3f, r5  # Target
```

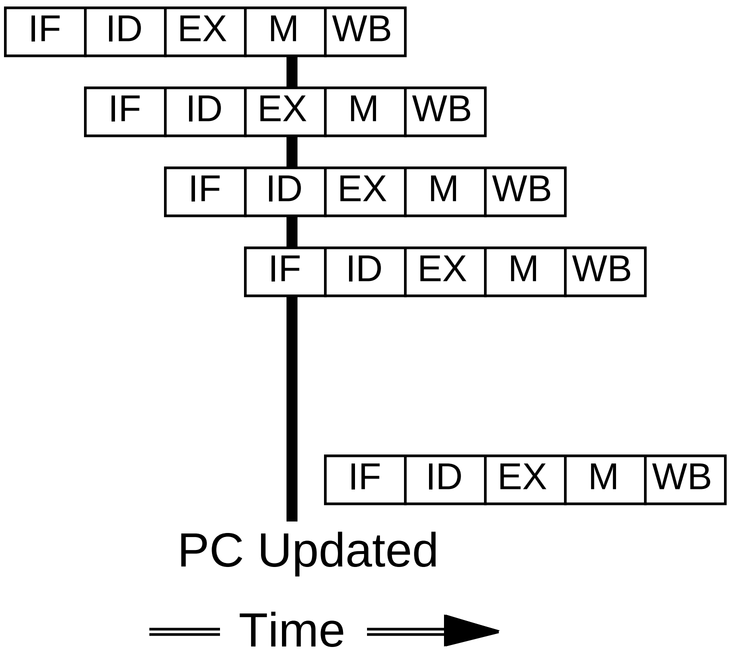
- **One cycle later**
- **Problem: Will execute 3 extra instructions!**



Branch Hazard Pipeline Diagram

Problem

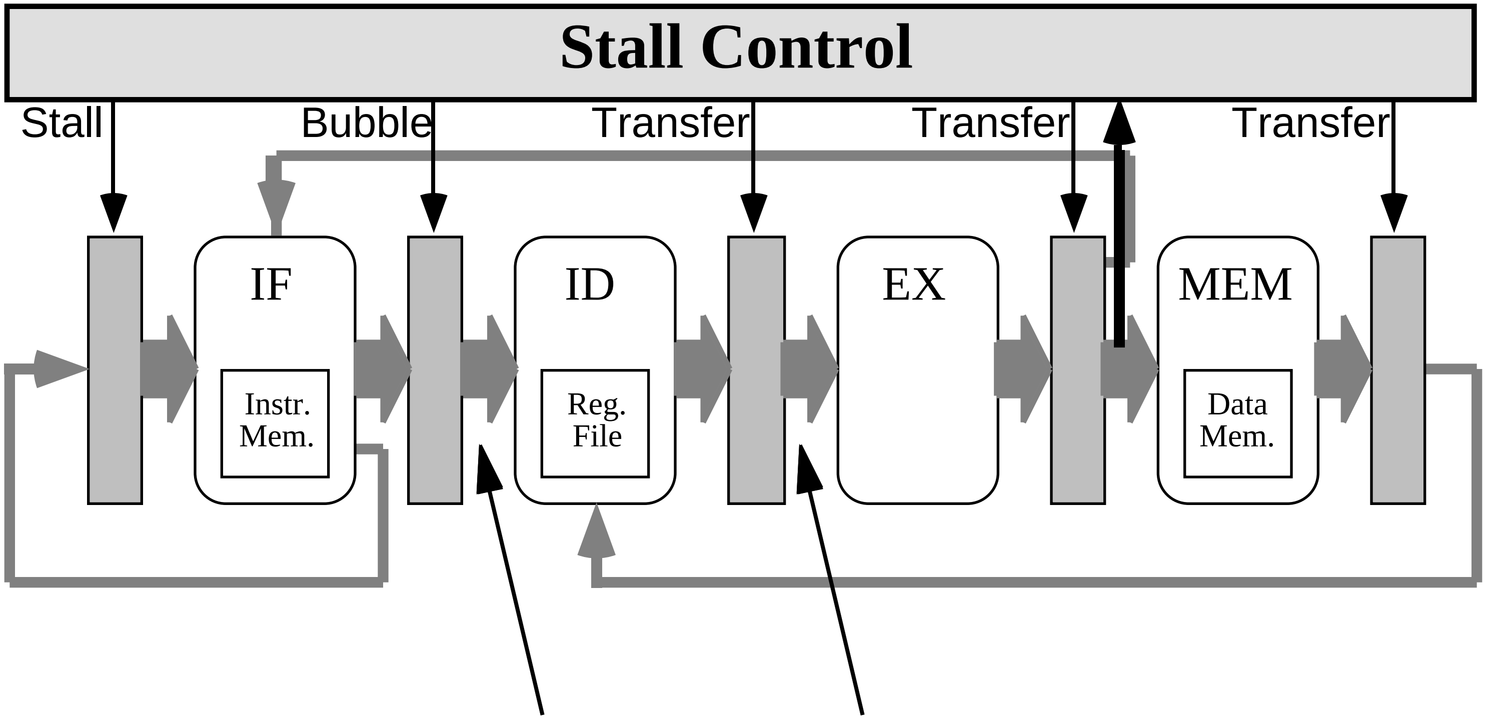
- **Instruction fetched in IF, branch condition set in MEM**



```
beq           $31, target
addq          $31, 63, $1
addq          $31, 63, $2
addq          $31, 63, $3
addq          $31, 63, $4

target: addq  $31, 63, $5
```

Stall Until Resolve Branch

- **Detect when branch in stages ID or EX**
- **Stop fetching until resolve**
  - – Stall IF. Inject bubble into ID

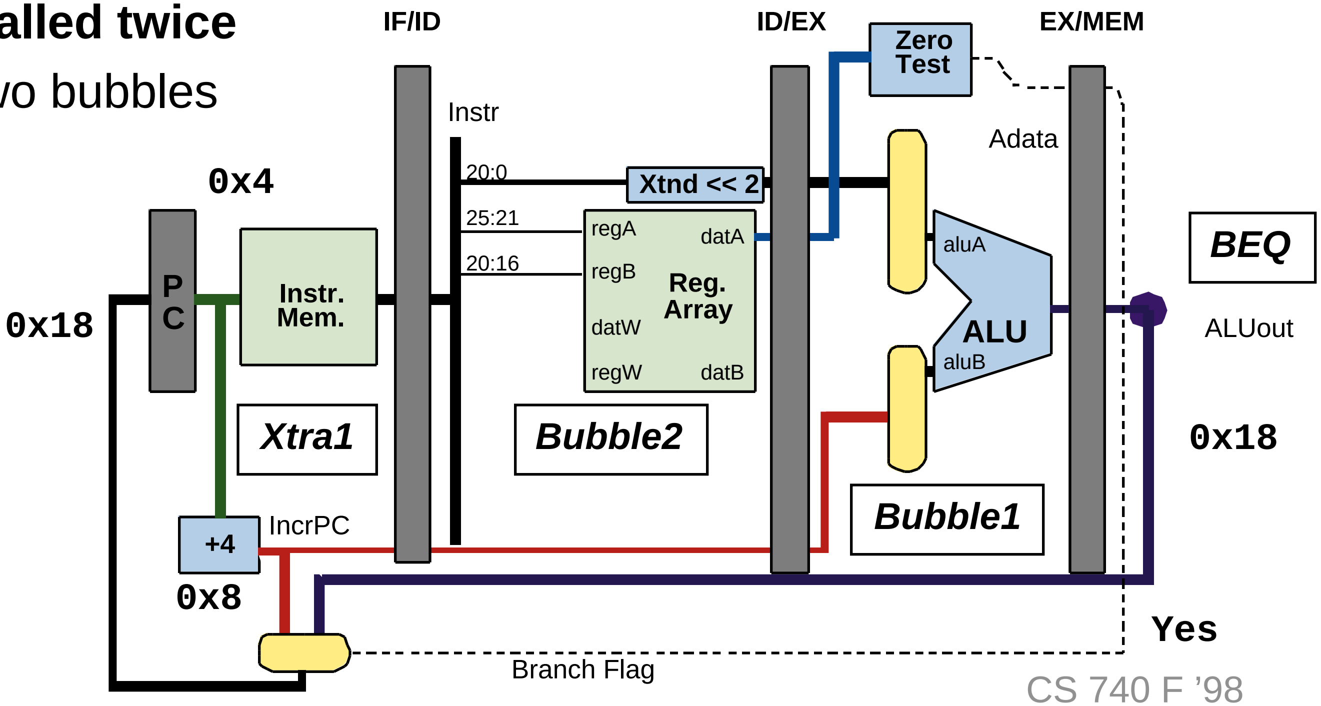


Perform when branch in either stage

Stalling Branch Example

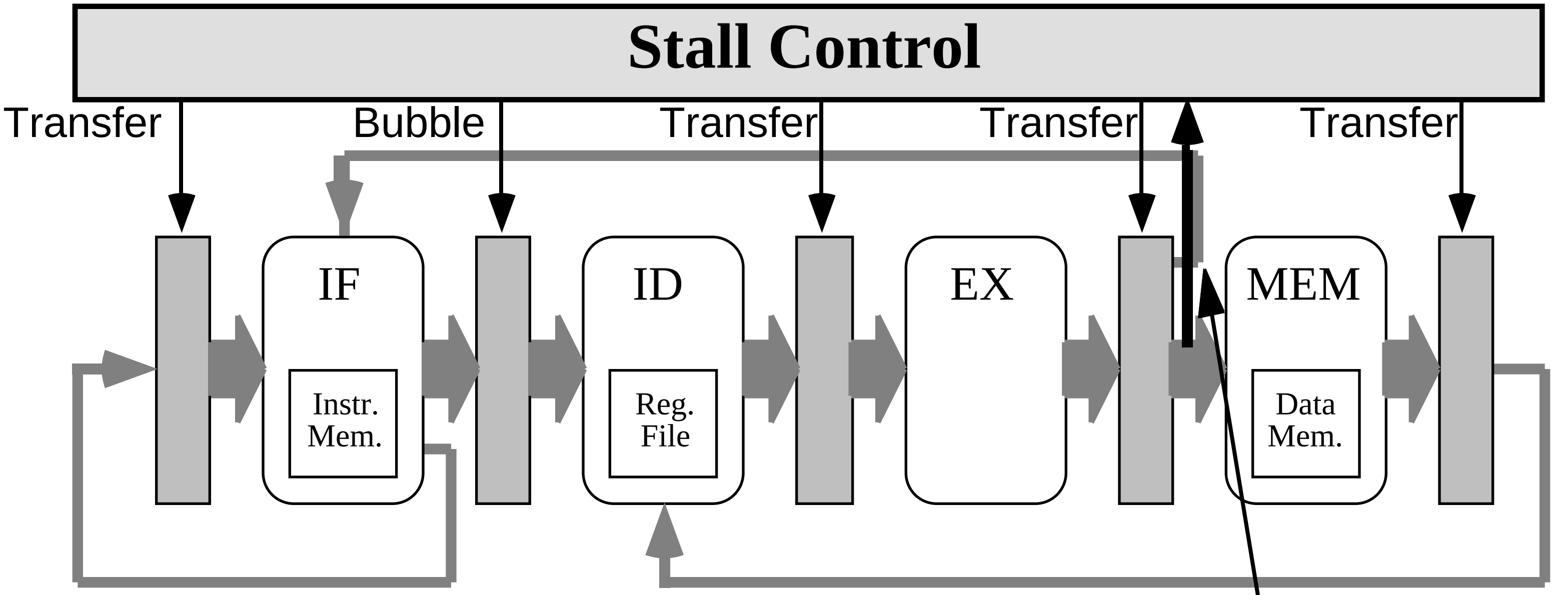
```
0x0:  beq      r31, 0x18        # Take
0x4:  addq     r31, 0x3f, r1    # Xtra1
0x8:  addq     r31, 0x3f, r2    # Xtra2
0xc:  addq     r31, 0x3f, r3    # Xtra3
0x10: addq     r31, 0x3f, r4    # Xtra4

0x18: addq     r31, 0x3f, r5    # Target
```

- **With BEQ in Mem stage**
- **Will have stalled twice**
  - Injects two bubbles



Taken Branch Resolution

- **When branch taken, still have instruction Xtra1 in pipe**
- **Need to flush it when detect taken branch in Mem**
  - Convert it to bubble

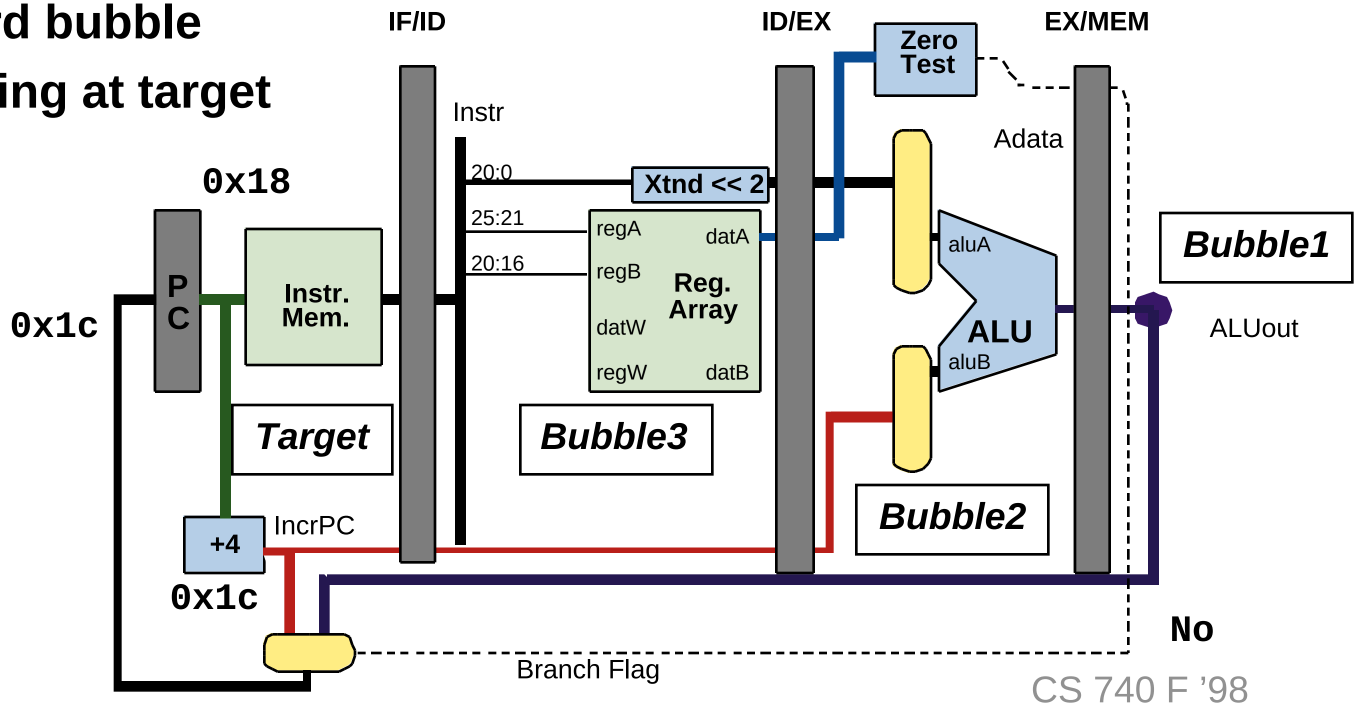


Perform when detect taken branch

Taken Branch Resolution Example

```
0x0:  beq     r31, 0x18        # Take
0x4:  addq    r31, 0x3f, r1    # Xtra1
0x8:  addq    r31, 0x3f, r2    # Xtra2
0xc:  addq    r31, 0x3f, r3    # Xtra3
0x10: addq    r31, 0x3f, r4    # Xtra4

0x18: addq    r31, 0x3f, r5    # Target
```

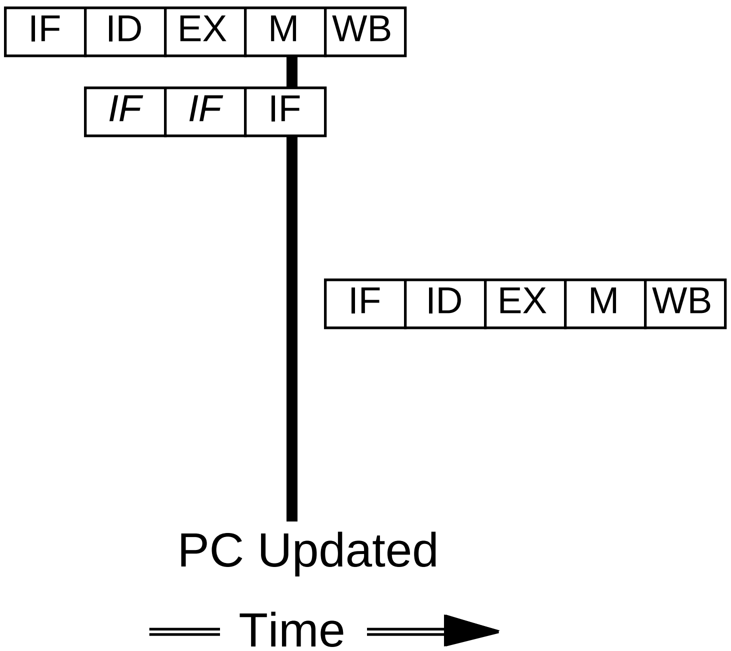
- **When branch taken**
- **Generate 3rd bubble**
- **Begin fetching at target**



Taken Branch Pipeline Diagram

Behavior

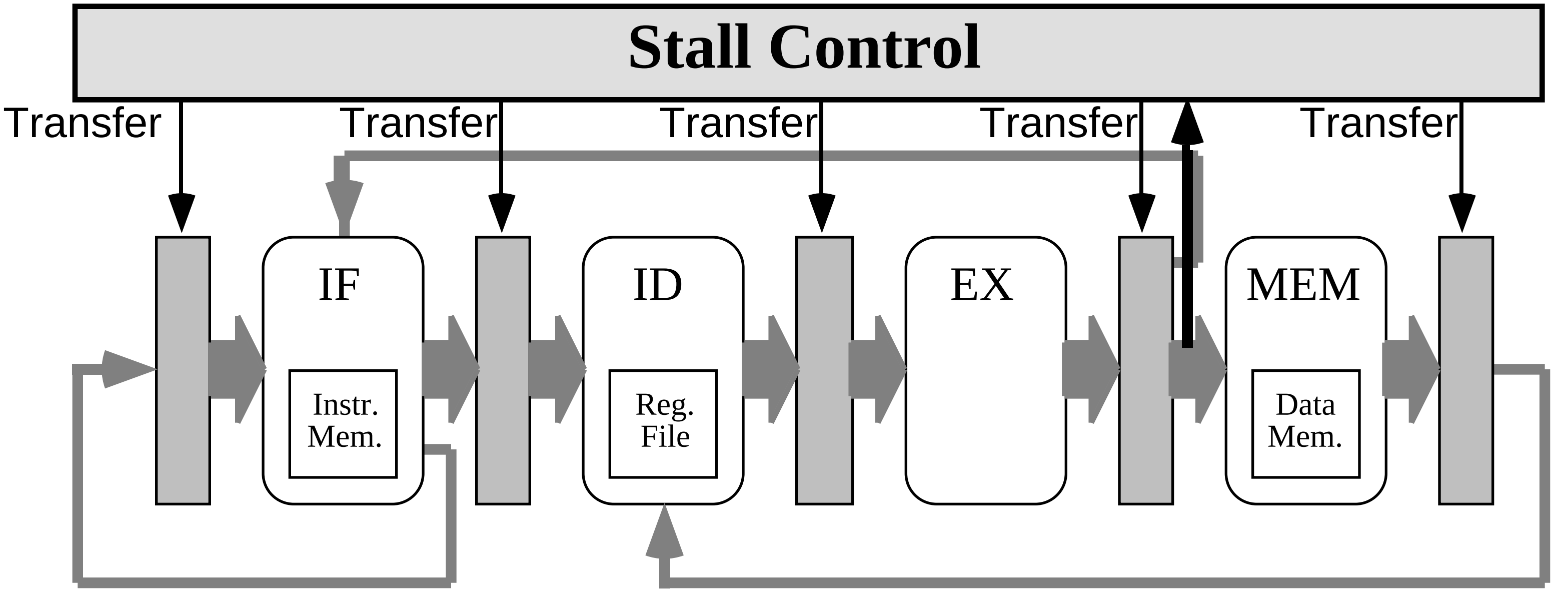
- **Instruction Xtra1 held in IF for two extra cycles**
- **Then turn into bubble as enters ID**



```
beq           $31, target
addq          $31, 63, $1  # Xtra1

target: addq  $31, 63, $5  # Target
```

Not Taken Branch Resolution

- [Stall two cycles with not-taken branches as well]
- When branch not taken, already have instruction Xtra1 in pipe
- Let it proceed as usual

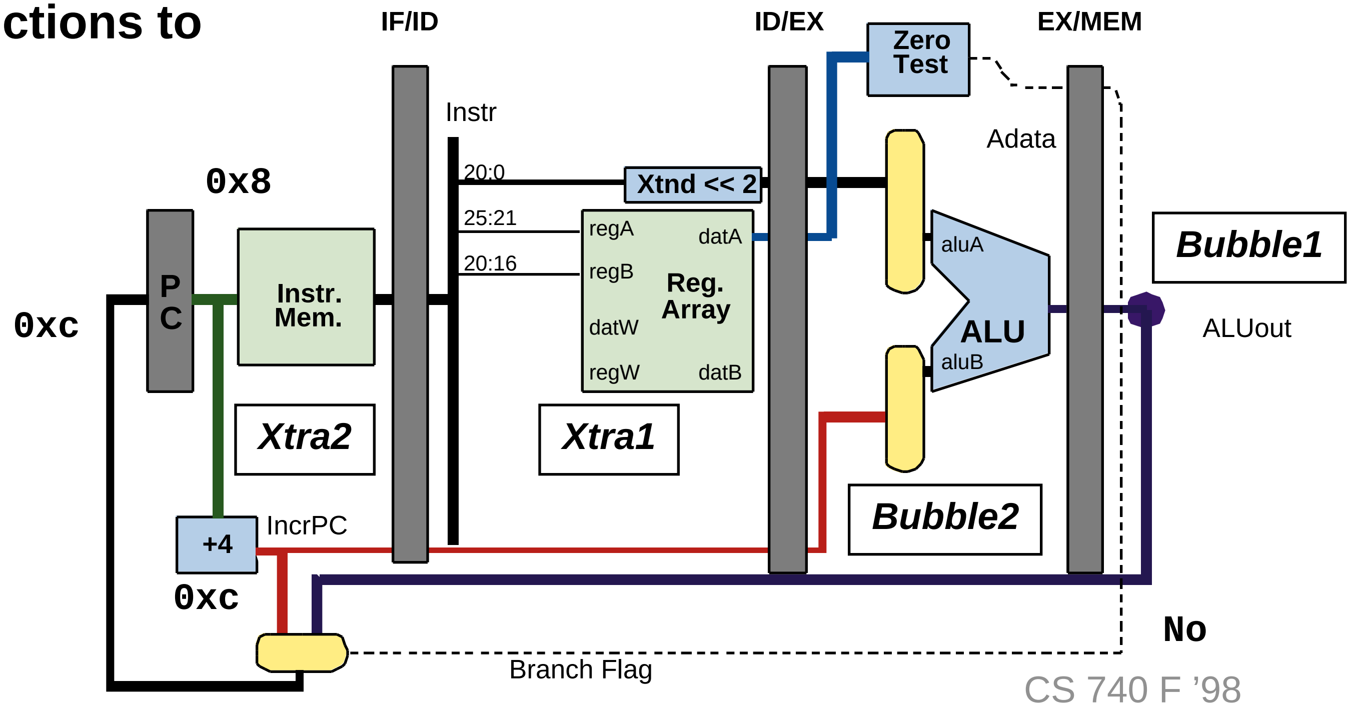


Not Taken Branch Resolution Example

demo09.O

```
0x0:  bne     r31, 0x18          # Don't Take
0x4:  addq    r31, 0x3f, r1      # Xtra1
0x8:  addq    r31, 0x3f, r2      # Xtra2
0xc:  addq    r31, 0x3f, r3      # Xtra3
0x10: addq    r31, 0x3f, r4      # Xtra4
```

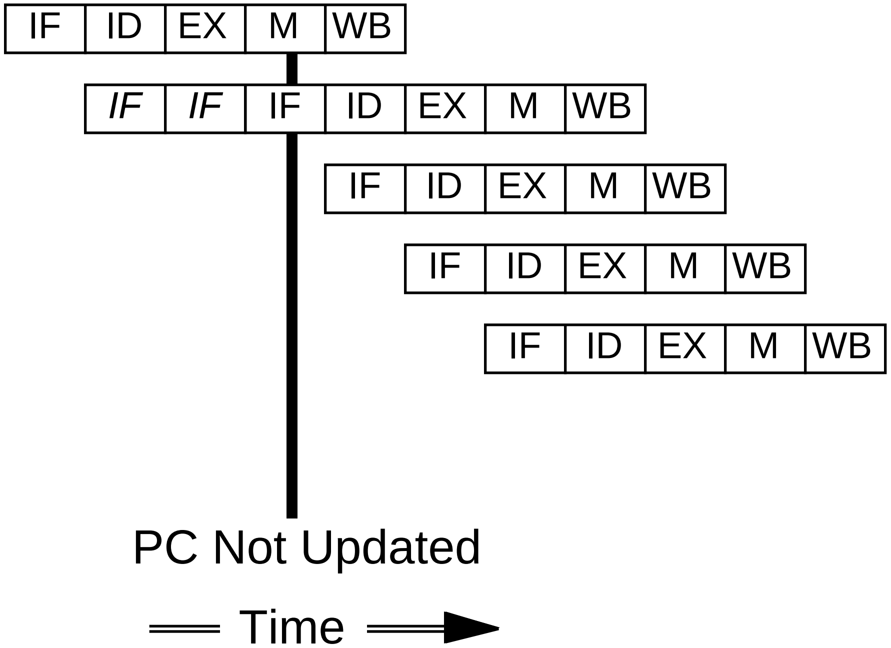
- **Branch not taken**
- **Allow instructions to proceed**



Not Taken Branch Pipeline Diagram

Behavior

- **Instruction Xtra1 held in IF for two extra cycles**
- **Then allowed to proceed**



```
beq     $31, target
addq    $31, 63, $1  # Xtra1
addq    $31, 63, $2  # Xtra2
addq    $31, 63, $3  # Xtra3
addq    $31, 63, $4  # Xtra4
```

Analysis of Stalling

Branch Instruction Timing

- **1 instruction cycle**
- **3 extra cycles when taken**
- **2 extra cycles when not taken**

Performance Impact

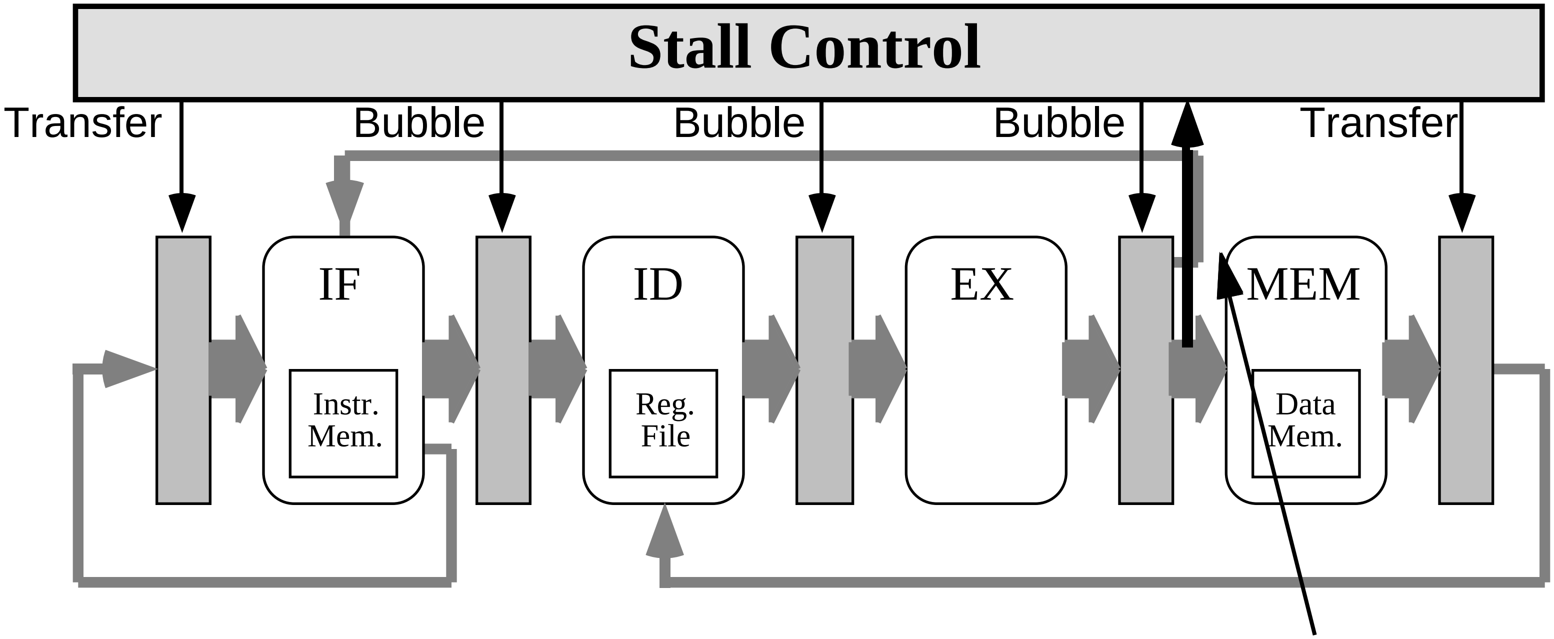
- **Branchs 16% of instructions in SpecInt92 benchmarks**
- **67% branches are taken**
- **Adds 0.16 * (0.67 * 3 + 0.33 * 2) == 0.43 cycles to CPI**
  - Average number of cycles per instruction
  - Serious performance impact

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**

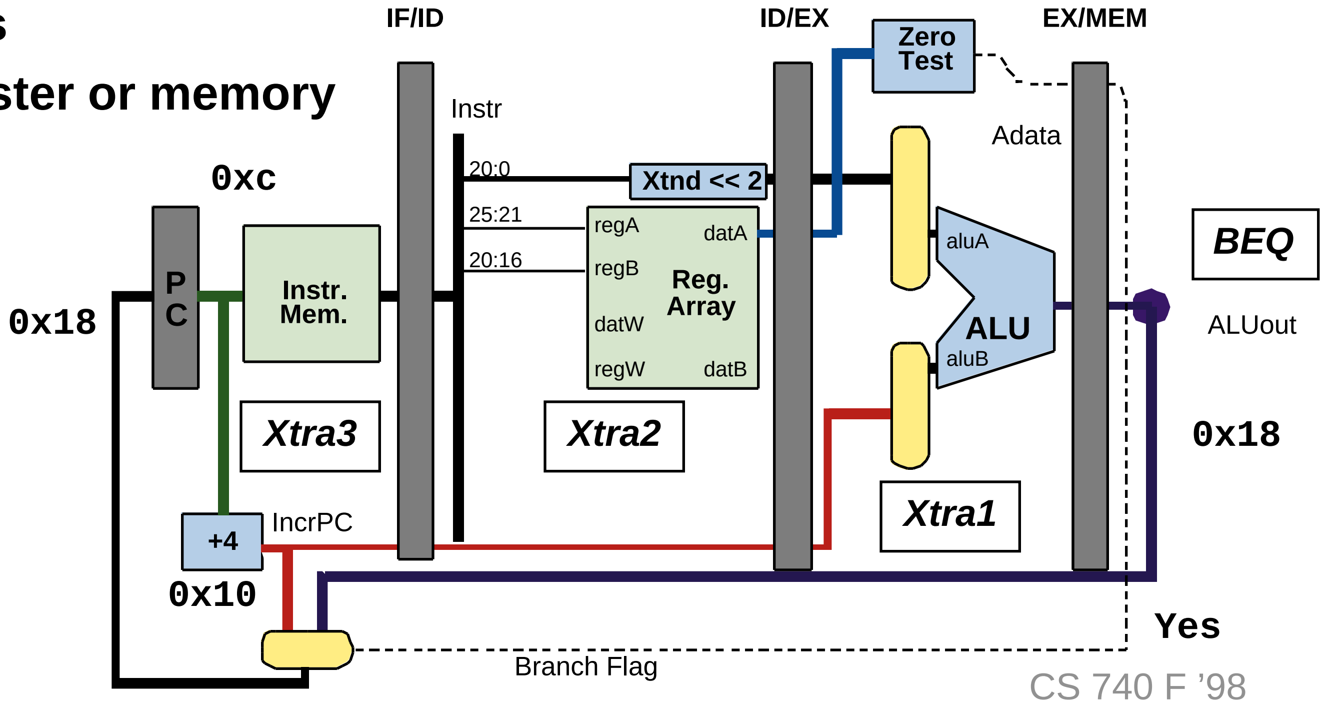


Perform when detect taken branch

Canceling Branch Example

```
0x0:  beq     r31, 0x18        # Take
0x4:  addq    r31, 0x3f, r1    # Xtra1
0x8:  addq    r31, 0x3f, r2    # Xtra2
0xc:  addq    r31, 0x3f, r3    # Xtra3
0x10: addq    r31, 0x3f, r4    # Xtra4

0x18: addq    r31, 0x3f, r5    # Target
```

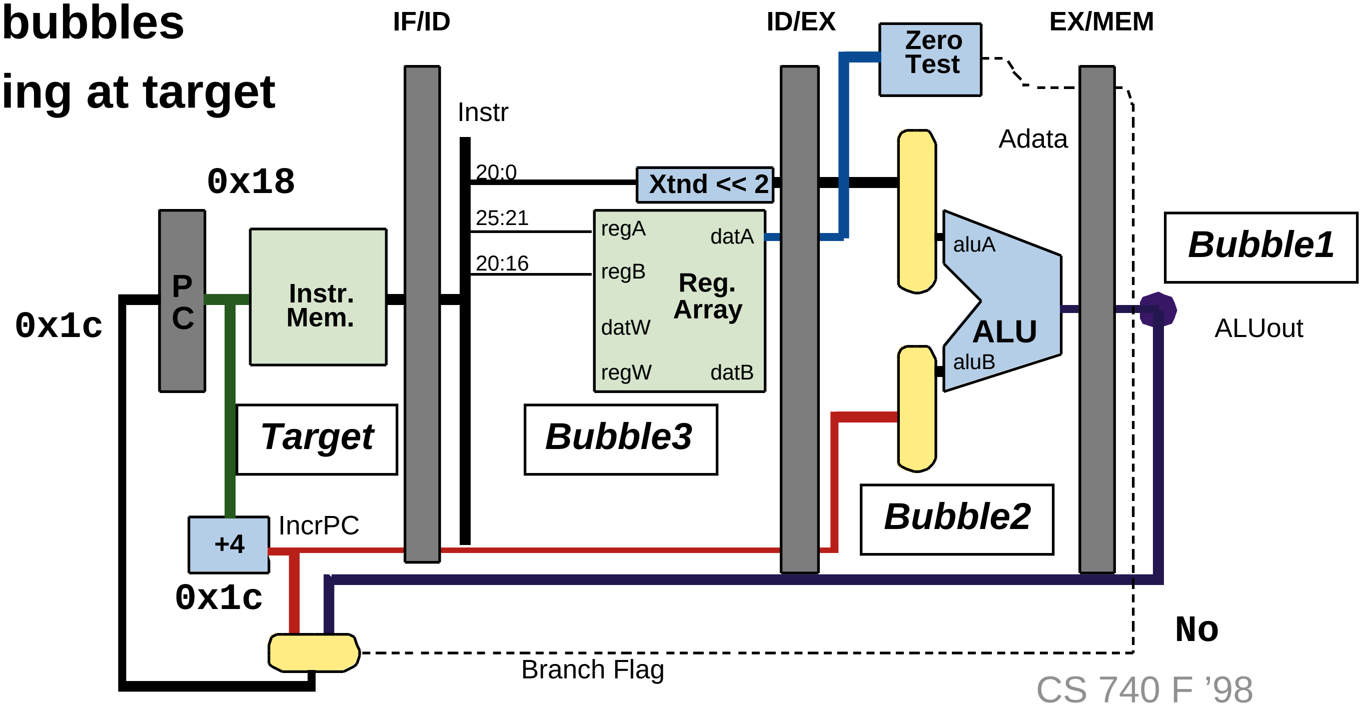
- **With BEQ in Mem stage**
- **Will have fetched 3 extra instructions**
- **But no register or memory updates**



Canceling Branch Resolution Example

```
0x0:  beq     r31, 0x18        # Take
0x4:  addq    r31, 0x3f, r1    # Xtra1
0x8:  addq    r31, 0x3f, r2    # Xtra2
0xc:  addq    r31, 0x3f, r3    # Xtra3
0x10: addq    r31, 0x3f, r4    # Xtra4

0x18: addq    r31, 0x3f, r5    # Target
```

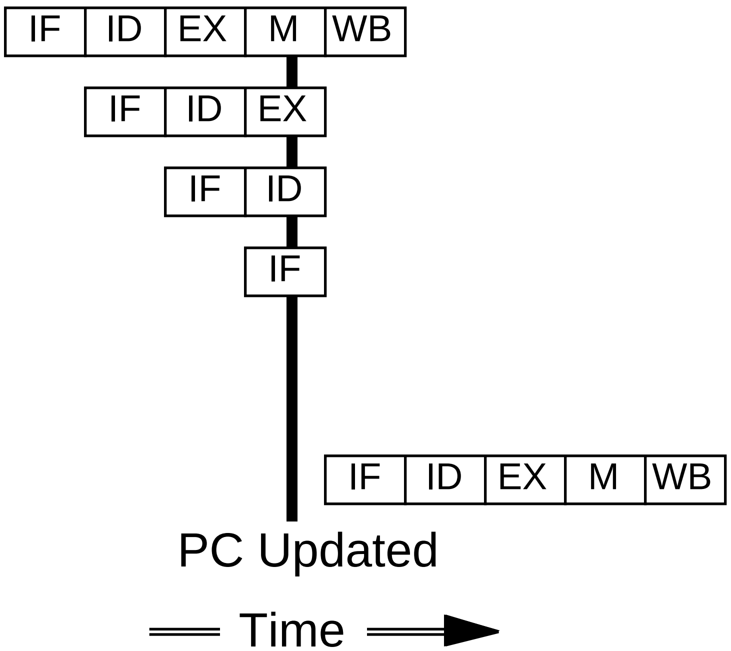
- **When branch taken**
- **Generate 3 bubbles**
- **Begin fetching at target**



Canceling Branch Pipeline Diagram

Operation

- **Process instructions assuming branch will not be taken**
- **When *is* taken, cancel 3 following instructions**

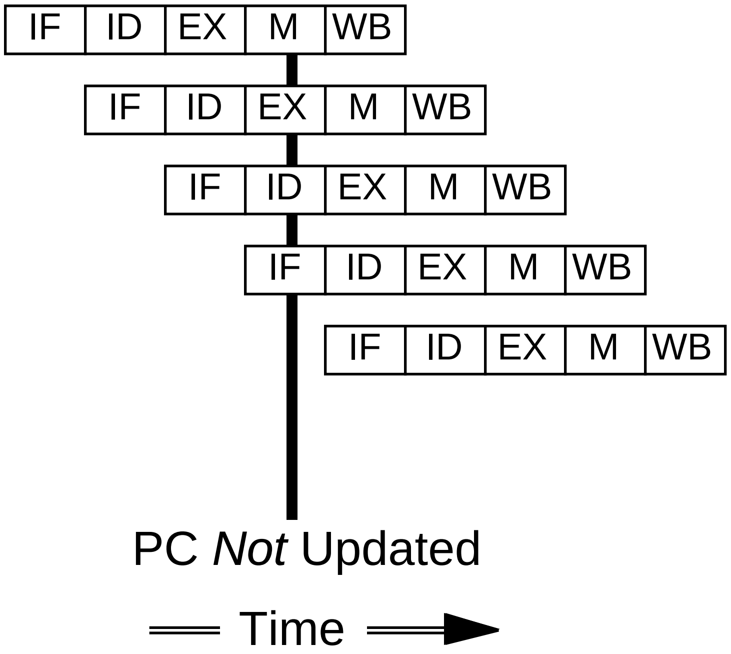


```
beq           $31, target
addq          $31, 63, $1
addq          $31, 63, $2
addq          $31, 63, $3
addq          $31, 63, $4

target: addq  $31, 63, $5
```

Noncanceling Branch Pipeline Diagram

Operation

- **Process instructions assuming branch will not be taken**
- **If really isn’t taken, then instructions flow unimpeded**



```
bne          $31, target
addq         $31, 63, $1
addq         $31, 63, $2
addq         $31, 63, $3
addq         $31, 63, $4

target: addq  $31, 63, $5
```

Branch Prediction Analysis

Our Scheme Implements “Predict Not Taken”

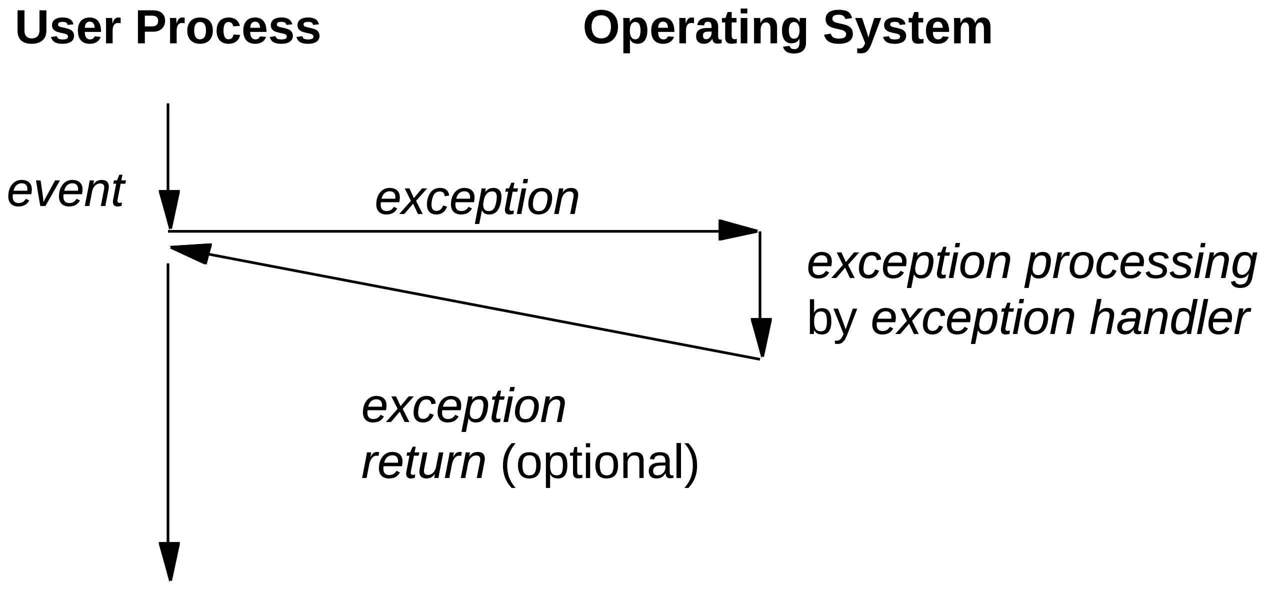
- **But 67% of branches are taken**
- **Impact on CPI: 0.16 * 0.67 * 3.0 = 0.32**
  - Still not very good

Alternative Schemes

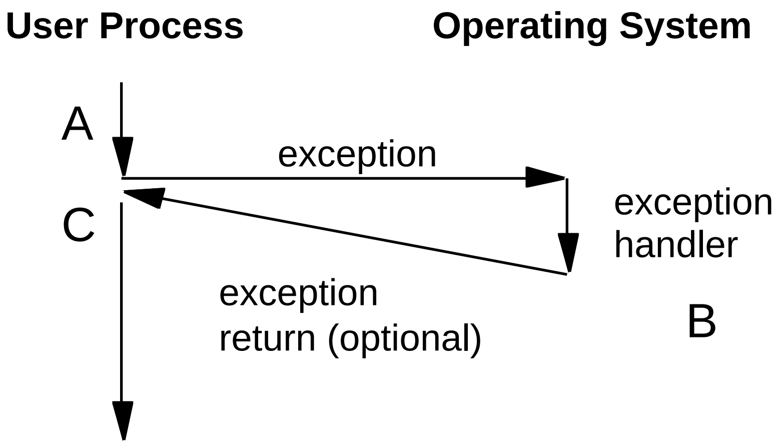
- **Predict taken**
  - Would be hard to squeeze into our pipeline
    - » Can’t compute target until ID
- **Backwards taken, forwards not taken**
  - Predict based on sign of displacement
  - Exploits fact that loops usually closed with backward branches

Exceptions

An *exception* is a transfer of control to the OS in response to some *event* (i.e. change in processor state)



Issues with Exceptions



A1: What kinds of events can cause an exception?

A2: When does the exception occur?

B1: How does the handler determine the location and cause of the exception?

B2: Are exceptions allowed within exception handlers?

C1: Can the user process restart?

C2: If so, where?

Internal (CPU) Exceptions

Internal exceptions occur as a result of events generated by executing instructions.

Execution of a CALL_PAL instruction.

- **allows a program to transfer control to the OS**

Errors during instruction execution

- **arithmetic overflow, address error, parity error, undefined instruction**

Events that require OS intervention

- **virtual memory page fault**

External (I/O) exceptions

External exceptions occur as a result of events generated by devices external to the processor.

I/O interrupts

- **hitting ^C at the keyboard**
- **arrival of a packet**
- **arrival of a disk sector**

Hard reset interrupt

- **hitting the reset button**

Soft reset interrupt

- **hitting ctl-alt-delete on a PC**

Exception handling (hardware tasks)

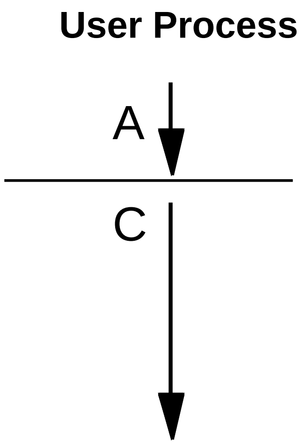
Recognize event(s)

Associate one event with one instruction.

- external event: pick any instruction
- multiple internal events: typically choose the earliest instruction.
- multiple external events: prioritize
- multiple internal and external events: prioritize

Create Clean Break in Instruction Stream

- Complete all instructions before excepting instruction
- Abort excepting and all following instructions
  - this clean break is called a *“precise exception”*



Exception handling (hardware tasks)

Set status registers

- **Exception Address: the EXC_ADDR register**
  - external exception: address of instruction about to be executed
  - internal exception: address of instruction causing the exception
    - » except for arithmetic exceptions, where it is the following instruction
- **Cause of the Exception: the EXC_SUM and FPCR registers**
  - was the exception due to division by zero, integer overflow, etc.
- **Others**
  - which ones get set depends on CPU and exception type

Disable interrupts and switch to kernel mode

Jump to common exception handler location

Exception handling (software tasks)

Deal with event

(Optionally) resume execution

- using special `REI` (return from exception or interrupt) instruction
- similar to a procedure return, but restores processor to user mode as a side effect.

Where to resume execution?

- usually re-execute the instruction causing exception

Precise vs. Imprecise Exceptions

In the Alpha architecture:

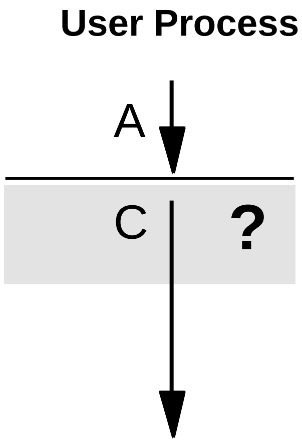
- **arithmetic exceptions may be *imprecise* (similar to the CRAY-1)**
  - motivation: simplifies pipeline design, helping to increase performance
- **all other exceptions are precise**

Imprecise exceptions:

- **all instructions before the excepting instruction complete**
- **the excepting instruction and instructions after it may or may not complete**

What if precise exceptions are needed?

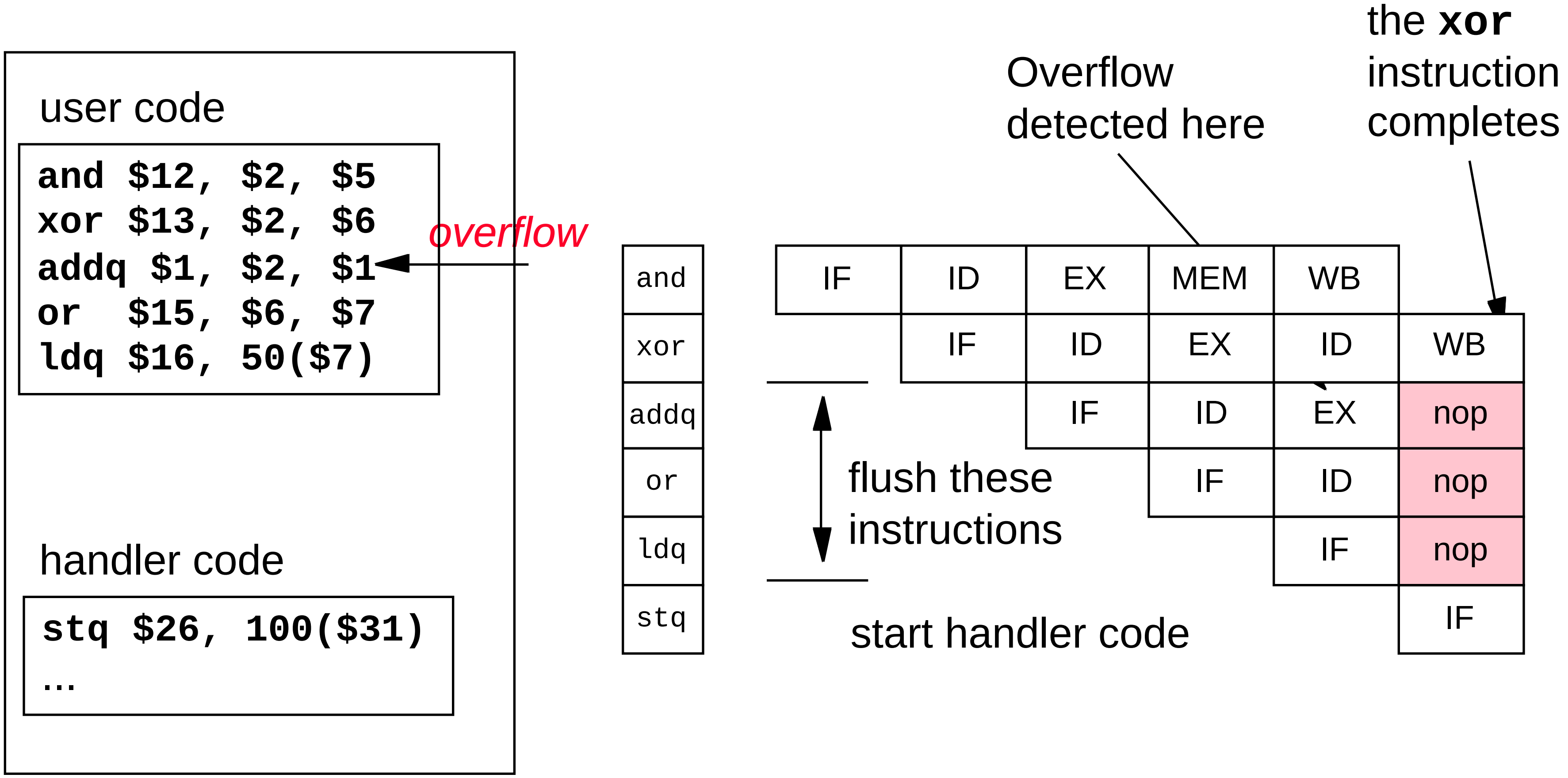
- **insert a TRAPB (trap barrier) instruction immediately after**
  - stalls until certain that no earlier insts take exceptions



In the remainder of our discussion, assume for the sake of simplicity that all Alpha exceptions are precise.

Example: Integer Overflow

(This example illustrates a *precise* version of the exception.)



user code

```
and $12, $2, $5
xor $13, $2, $6
addq $1, $2, $1
or  $15, $6, $7
ldq $16, 50($7)
```

handler code

```
stq $26, 100($31)
...
```

Overflow detected here

the **xor** instruction completes

and	IF	ID	EX	MEM	WB	
xor		IF	ID	EX	ID	WB
addq			IF	ID	EX	nop
or				IF	ID	nop
ldq					IF	nop
stq						IF

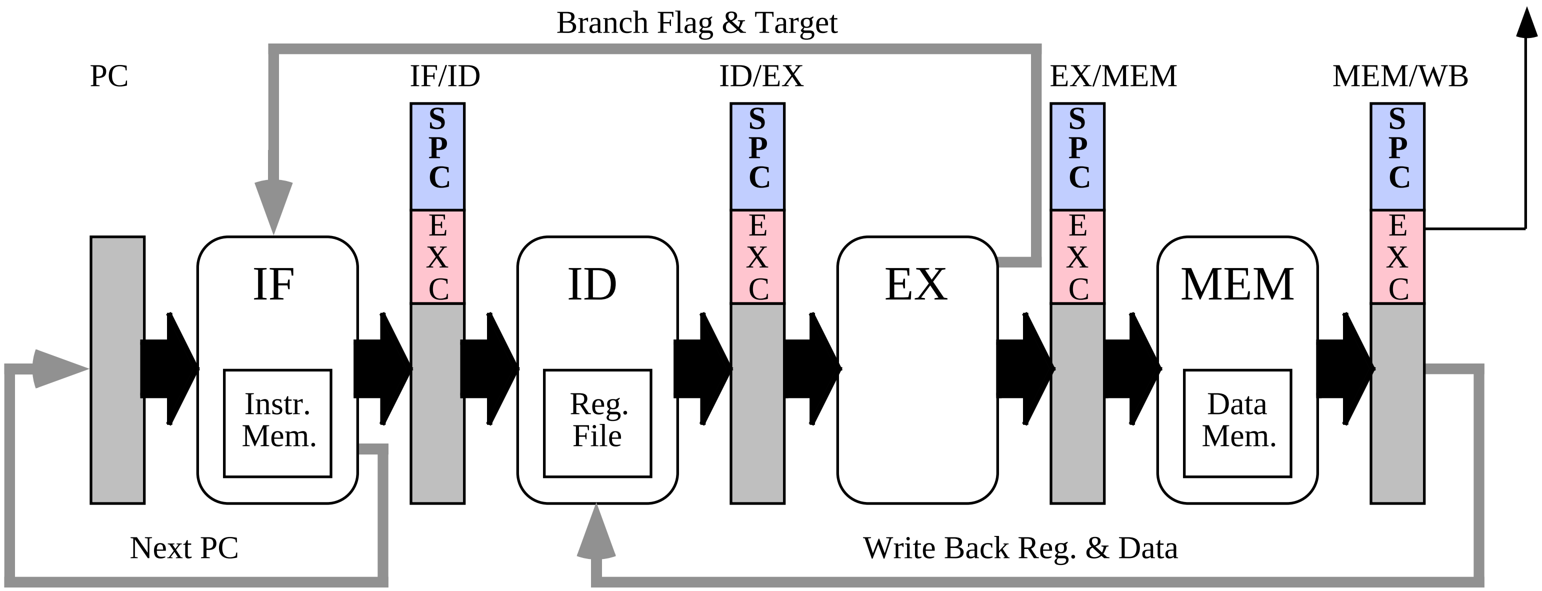
flush these instructions

start handler code

Exception Handling in pAlpha Simulator

Relevant Pipeline State

- **Address of instruction in pipe stage (SPC)**
- **Exception condition (EXC)**
  - Set in stage when problem encountered
    - » IF for fetch problems, EX for instr. problems, MEM for data probs.
  - Triggers special action once hits WB



Alpha Exception Examples

- In directory ***HOME740*/public/sim/demos**

Illegal Instruction (exc01.O)

```
0x0: sll   r3, 0x8, r5   # unimplemented
0x4: addq r31, 0x4, r2   # should cancel
```

Illegal Instruction followed by store (exc02.O)

```
0x0:  addq r31, 0xf, r2
0x4:  sll  r3, 0x8, r5   # unimplemented
0x8:  stq  r2, 8(r31)    # should cancel
```

More Examples: Multiple Exceptions

EX exception follows MEM exception (exc03.O)

```
0x0: addq r31, 0x3, r3
0x4: stq   r3, -4(r31)  # bad address
0x8: sll   r3, 0x8, r5  # unimplemented
0xc: addq r31, 0xf, r2
```

MEM exception follows EX exception (exc04.O)

```
0x0: addq r31, 0x3, r3
0x4: sll   r3, 0x8, r5  # unimplemented
0x8: stq   r3, -4(r31)  # bad address
0xc: addq r31, 0xf, r2
```

These exceptions are detected *simultaneously* in the pipeline!

Which is the excepting instruction?

Final Alpha Exception Example

Avoiding false alarms (exc05.O)

```
0x0:  beq  r31, 0xc      # taken
0x4:  sll   r3, 0x8, r5  # should cancel
0x8:  bis  r31, r31, r31
0xc:  addq r31, 0x1, r2  # target
0x10: call_pal halt
```

Exception detected in the pipeline, but should not really occur.

Implementation Features

Correct

- **Detects excepting instruction**
  - Furthest one down pipeline = Earliest one in program order
  - (e.g., **exc03.O** vs. **exc04.O**)
- **Completes all preceding instructions**
- **Usually aborts excepting instruction & beyond**
- **Prioritizes exception conditions**
  - Earliest stage where instruction ran into problems
- **Avoids false alarms (exc05.O)**
  - Problematic instructions that get canceled anyhow

Shortcomings

- **Store following excepting instruction (exc02.O)**

Requirements for Full Implementation

Exception Detection

- **Detect external interrupts at IF**
  - Complete all fetched instructions

Instruction Shutdown

- **Suspend if unusual condition in MEM or WB**
- **Save proper value of EXC_ADDR**
  - Not always same as SPC
- **Rest of control state**

Handler Startup

- **Begin fetching handler code**

Multicycle instructions

Alpha 21264 Execution Times:

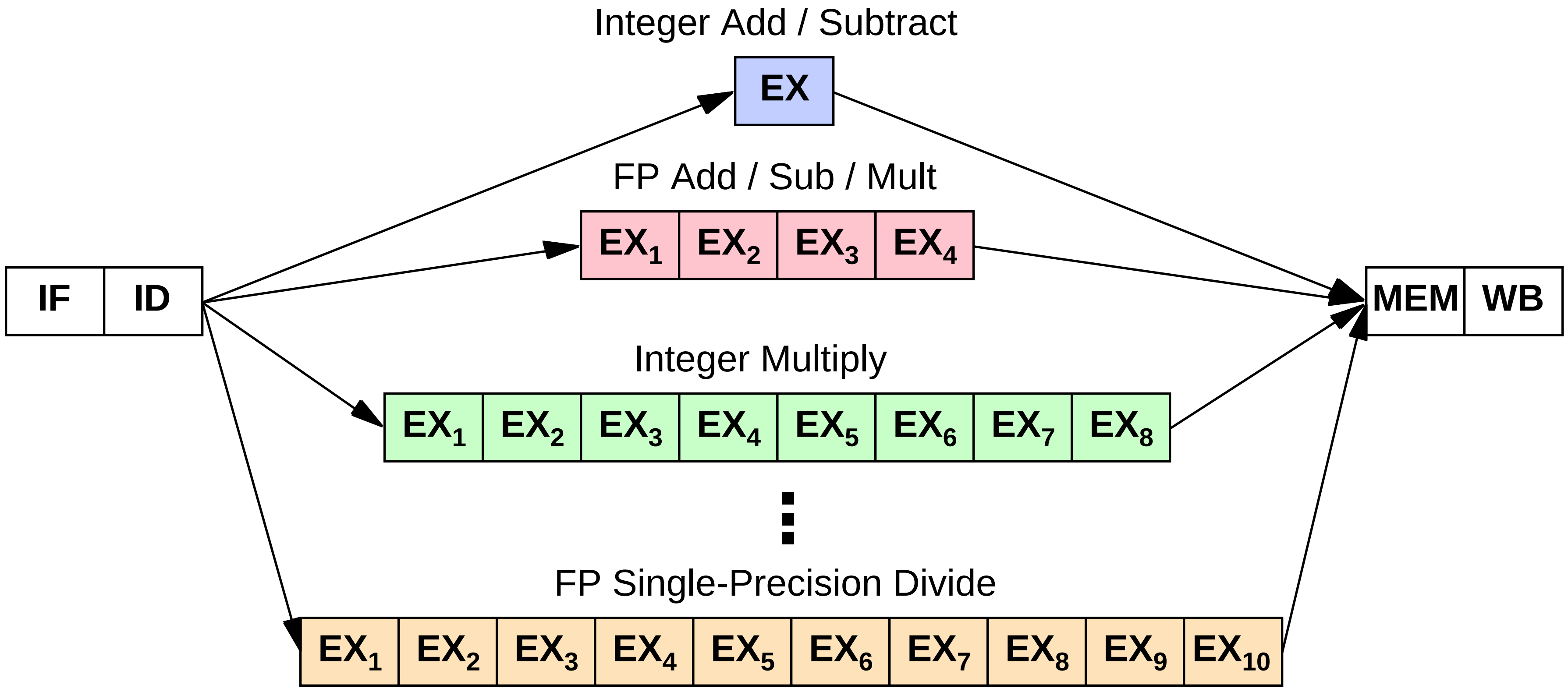
- Measured in clock cycles

Operation	Integer	FP-Single	FP-Double
add / sub	1	4	4
multiply	8-16	4	4
divide	N / A	10	23

H&P Dynamic Instruction Counts:

	Integer	FP Benchmarks	
Operation	Benchmarks	Integer	FP
add / sub	14%	11%	14%
multiply	< 0.1%	< 0.1%	13%
divide	< 0.1%	< 0.1%	1%

Pipeline Revisited



Integer Add / Subtract

EX

FP Add / Sub / Mult

EX_1 EX_2 EX_3 EX_4

IF ID

MEM WB

Integer Multiply

EX_1 EX_2 EX_3 EX_4 EX_5 EX_6 EX_7 EX_8

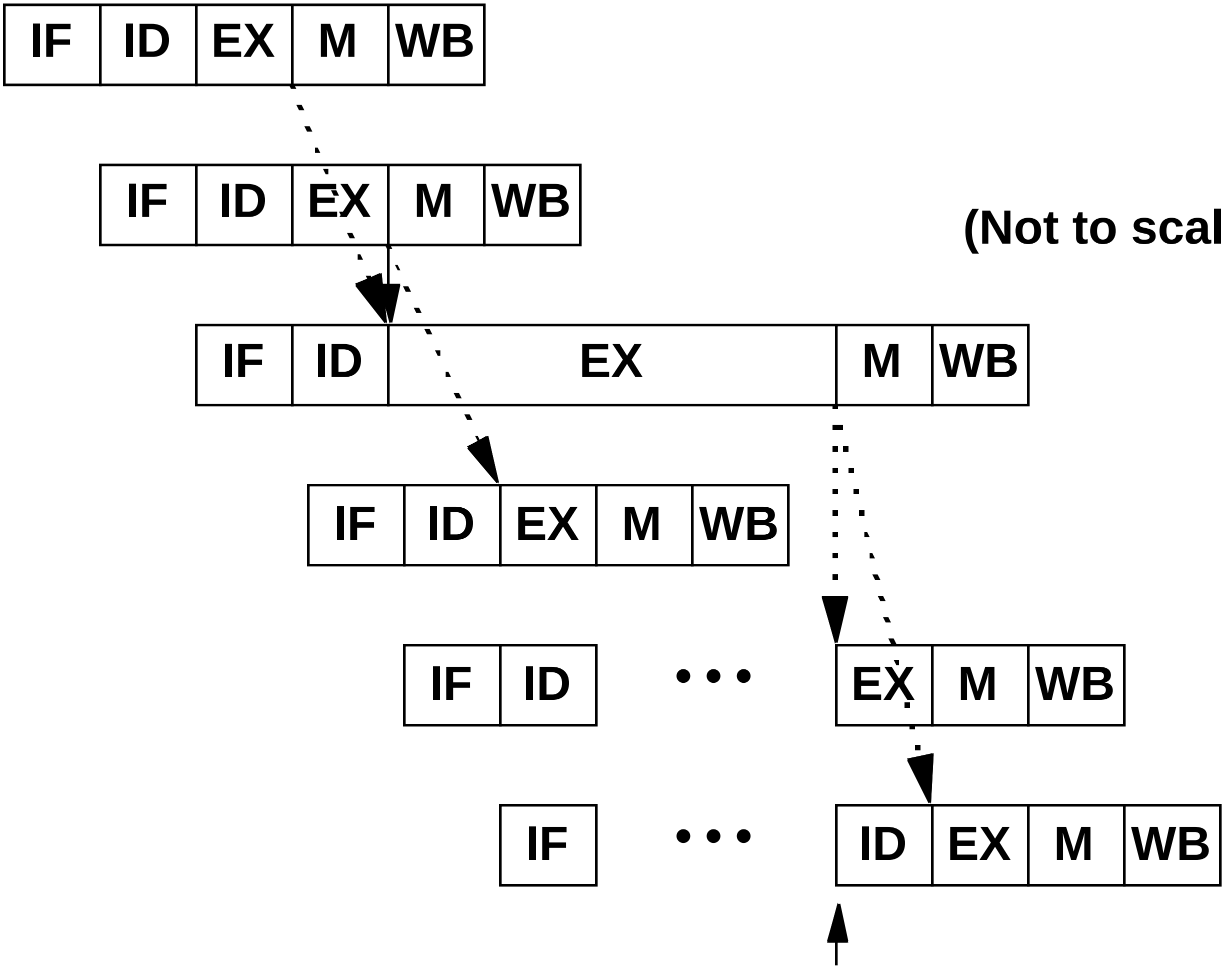
⋮

FP Single-Precision Divide

EX_1 EX_2 EX_3 EX_4 EX_5 EX_6 EX_7 EX_8 EX_9 EX_{10}

Multiply Timing Example

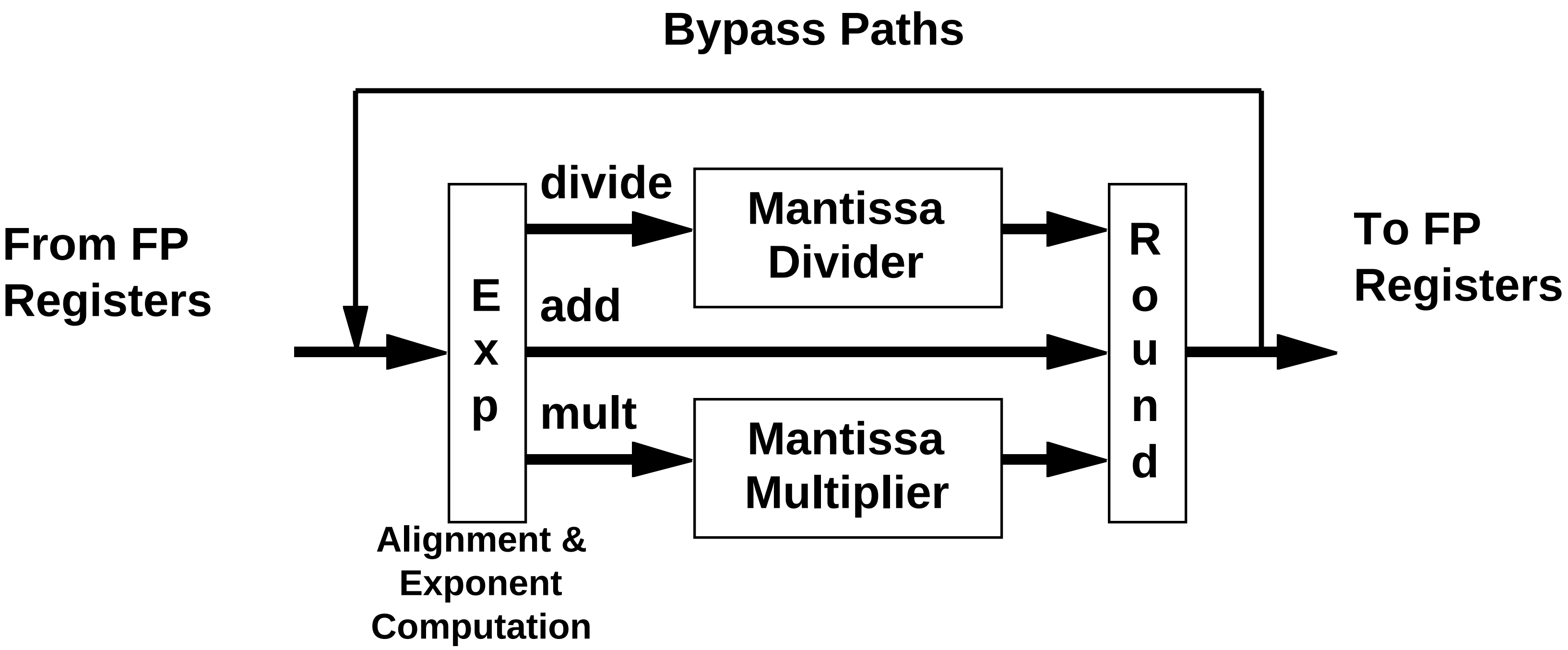
```
bis $31, 3, $2

bis $31, 7, $3

mulq $2, $3, $4

addq $2, $3, $3

bis  $4, $31, $5

addq $2, $4, $2
```



(Not to scale)

Stall while Busy

Floating Point Hardware (from MIPS)



Independent Hardware Units

- Can concurrently execute add, divide, multiply
- Except that all share exponent and rounding units
- Independent of integer operations

Control Logic

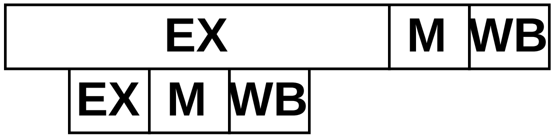
Busy Flags

- **One per hardware unit**
- **One per FP register**
  - Destination of currently executing operation

Stall Instruction in ID if:

- **Needs unit that is not available**
- **Source register busy**
  - Avoids RAW (Read-After-Write) hazard
- **Destination register busy**
  - Avoids WAW hazard

```
divt $f1, $f2, $f4
addt $f1, $f2, $f4
```



EX M WB

EX M WB

Bypass paths

- **Similar to those in integer pipeline**

FP Timing Example

`addt $f2, $f4, $f1`

`divt $f1, $f2, $f6`

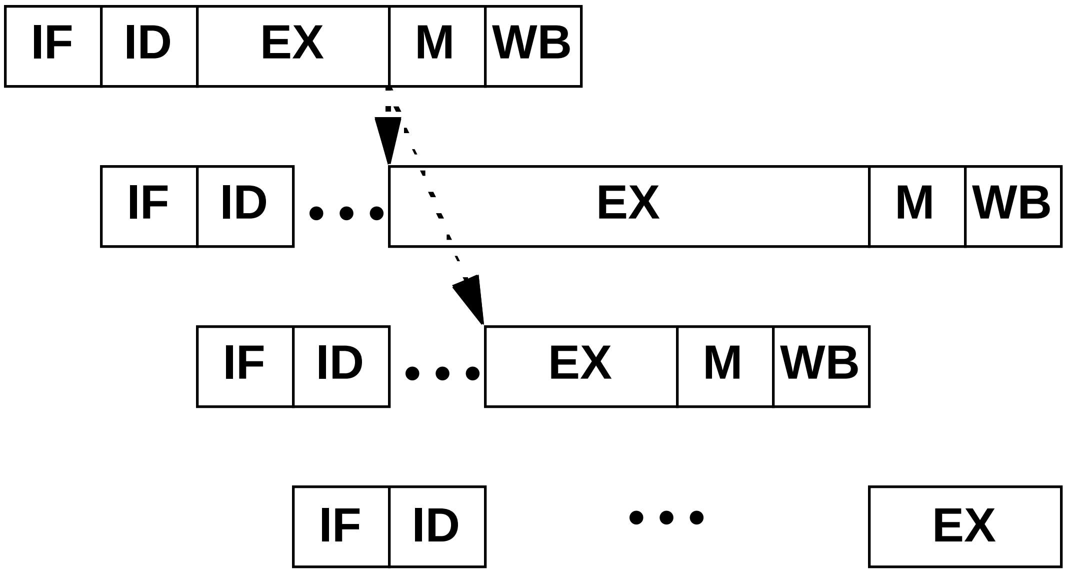
RAW Stall

`addt $f1, $f2, $f8`

Structural Stall

`addt $f1, $f2, $f6`

WAW Stall



Conclusion

Pipeline Characteristics for Multi-cycle Instructions

- **In-order issue**
  - Instructions fetched and decoded in program order
- **Out-of-order completion**
  - Slow instructions may complete after ones that are later in program order

Performance Opportunities

- **Transformations such as loop unrolling & software pipelining to expose potential parallelism**
- **Schedule code to use multiple functional units**
  - Must understand idiosyncracies of pipeline structure