

Basic Pipelining

21 March 2017

Topics

- Objective
- Instruction formats
- Instruction processing
- Principles of pipelining
- Inserting pipe registers

Welcome back!

Mid-semester grades

Median: A-

Grade steps with ± 0.5 in z-score

Exam stats (15% of grade)

Range: 6350 - 19500

Average: 13407

Median: 13375

Std dev: 3230

Will aim for slightly shorter exam next time

Happy to discuss in office hours!

Objective

Design Processor for Alpha Subset

- Interesting but not overwhelming quantity
- High level functional blocks

Initial Design

- One instruction at a time
- Single cycle per instruction

Refined Design

- 5-stage pipeline
 - Similar to early RISC processors
- Goal: approach 1 cycle per instruction but with shorter cycle time

ALPHA INSTRUCTION SET

Alpha Arithmetic Instructions

RR-type instructions (addq, subq, xor, bis, cmplt): $rc \leftarrow ra \text{ funct } rb$

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): $rc \leftarrow ra \text{ funct } ib$

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

Encoding

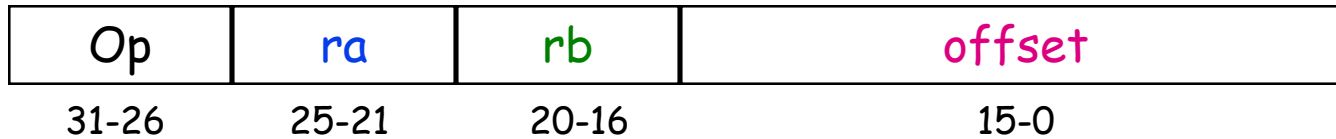
- **ib is 8-bit unsigned literal**

Operation	Op field	funct field
addq	0x10	0x20
subq	0x10	0x29
or/bis	0x11	0x20
xor	0x11	0x40
cmoveq	0x11	0x24 (32b conditional move)
cmplt	0x11	0x4D (compare less-than)

Alpha Load/Store Instructions

Load: $Ra \leftarrow Mem[Rb + offset]$

Store: $Mem[Rb + offset] \leftarrow Ra$



Encoding

- **offset** is 16-bit signed offset

Operation

Op field

ldq

0x29

stq

0x2D

Branch Instructions

Cond. Branch: $PC \leftarrow \text{Cond}(Ra) ? PC + 4 + \text{disp} * 4 : PC + 4$

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

Encoding

- disp** is 21-bit signed displacement

Operation

beq

Op field

0x39

Cond

Ra == 0

bne

0x3D

Ra != 0

Why PC + 4?

Is ALPHA designed for compilers or assembly programmers?
(RISC vs CISC)

Branch [Subroutine] (br, bsr): $Ra \leftarrow PC + 4; PC \leftarrow PC + 4 + \text{disp} * 4$

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

Operation

br

Op field

0x30

bsr

0x34

Transfers of Control

jmp, jsr, ret: $Ra \leftarrow PC+4; PC \leftarrow Rb$

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

Encoding

- High order 2 bits of Hint encode jump type
- Remaining bits give information about predicted destination
- Hint does not affect functionality

Jump Type Hint 15:14

jmp	00
jsr	01
ret	10

By convention:

Instruction	Ra	Rb
jmp	31	-
jsr	-	26
ret	31	26

Instruction Encoding

0x0:	40220403	addq	r1, r2, r3
0x4:	4487f805	xor	r4, 0x3f, r5
0x8:	a4c70abc	ldq	r6, 2748(r7)
0xc:	b5090123	stq	r8, 291(r9)
0x10:	e47ffffb	beq	r3, 0
0x14:	d35ffffa	bsr	r26, 0(r31)
0x18:	6bfa8001	ret	r31, (r26), 1

Object Code

- Instructions encoded in 32-bit words
- Program behavior determined by bit encodings
- Disassembler simply converts these words to readable instructions

Decoding Examples

0x0: 40220403 addq r1, r2, r3

4	0	2	2	0	4	0	3
0100	0000	0010	0010	0000	0100	0000	0011
10		01		02		20	
						03	

0x8: a4c70abc ldq r6, 2748(r7)

a	4	c	7	0	a	b	c
1010	0100	1100	0111	0000	1010	1011	1100
29		06		07		0abc	
						= 2748 ₁₀	

0x10: e47ffffb beq r3, 0

e	4	7	f	f	f	f	b
1110	0100	0111	1111	1111	1111	1111	1011
39		03		1ffffb			
				= -5 ₁₀			

0x18: 6bfa8001 ret r31, (r26), 1

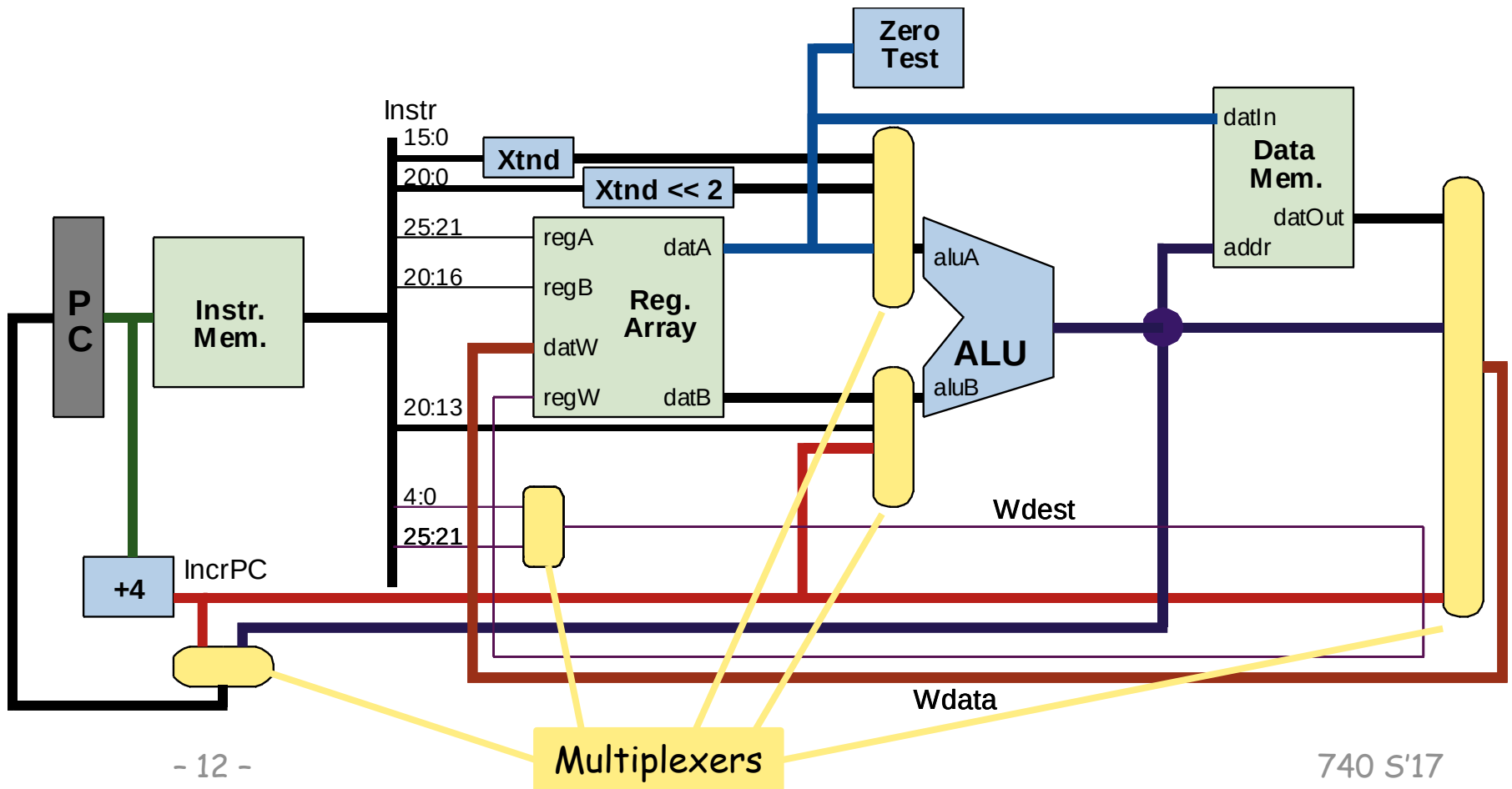
6	b	f	a	8	0	0	1
0110	1011	1111	1010	1000	0000	0000	0001
1a		1f		1a		2	
						= 31 ₁₀ = 26 ₁₀	

Target = 16 # Current PC
 + 4 # Increment
 + 4 * -5 # Disp
 = 0

SINGLE-CYCLE ALPHA IMPLEMENTATION

Datapath

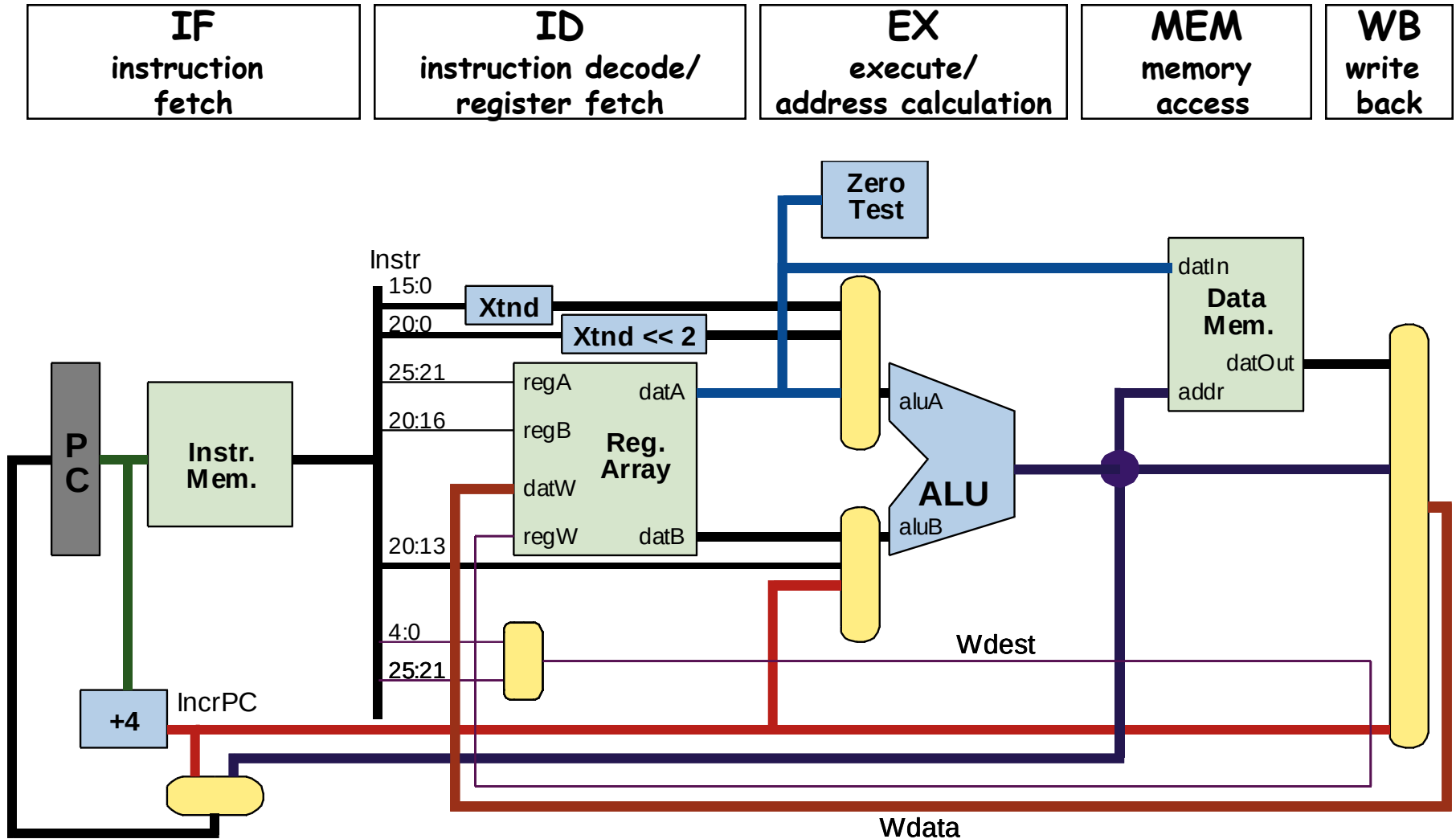
Block diagram for a computer processor, excluding control signals



Datapath



Datapath



Hardware Units

Storage

- **Instruction Memory**
 - Fetch 32-bit instructions
- **Data Memory**
 - Load / store 64-bit data
- **Register Array**
 - Storage for 32 integer registers
 - Two read ports: can read two registers at once
 - Single write port

Functional Units

- | | |
|--------------------|-------------------------------------|
| • +4 | PC incrementer |
| • Xtnd | Sign extender |
| • ALU | Arithmetic and logical instructions |
| • Zero Test | Detect whether operand == 0 |

RR-type instructions

RR-type instructions (addq, subq, xor, bis, cmplt): $rc \leftarrow ra \text{ funct } rb$

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

IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $PC \leftarrow PC + 4$

ID: Instruction decode/register fetch

- $A \leftarrow Register[IR[25:21]]$
- $B \leftarrow Register[IR[20:16]]$

Ex: Execute

- $ALUOutput \leftarrow A \text{ op } B$

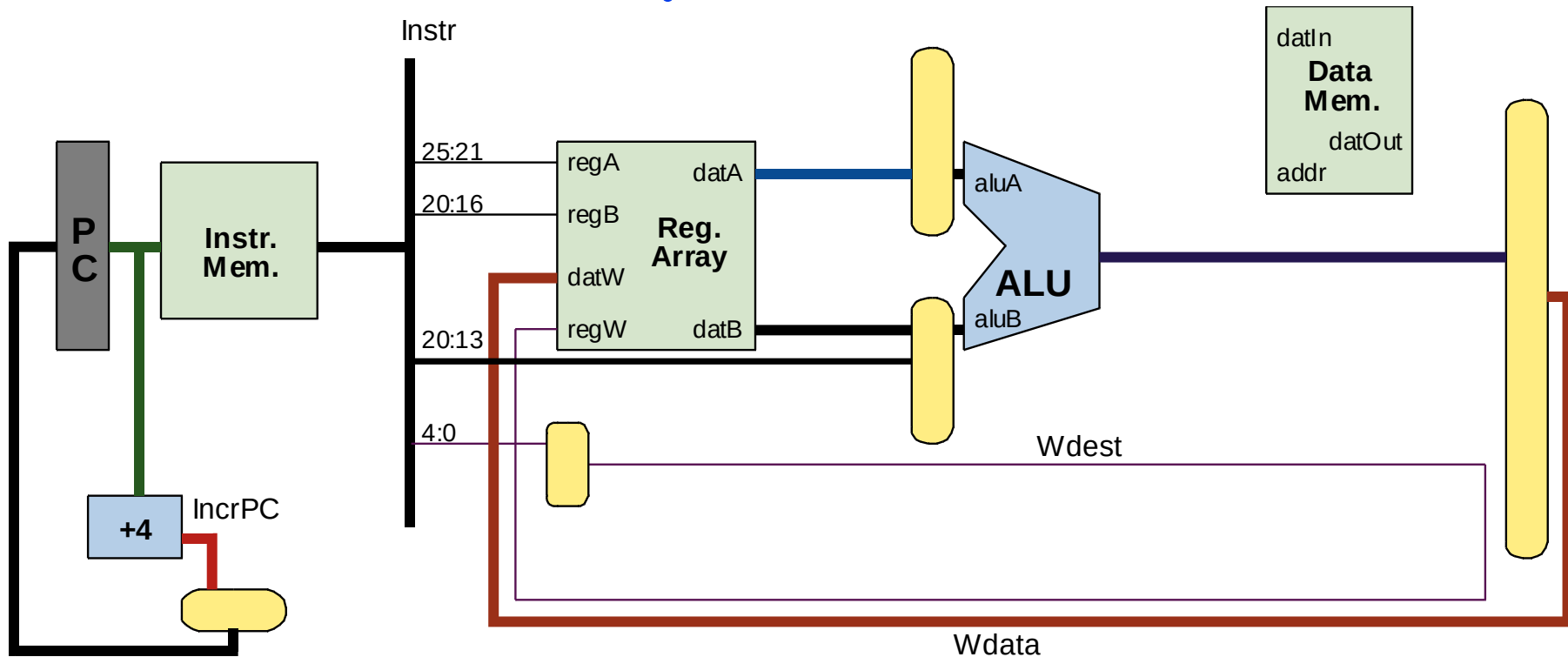
MEM: Memory

- nop

WB: Write back

- $Register[IR[4:0]] \leftarrow ALUOutput$

Active Datapath for RR & RI



ALU Operation

- Input B selected according to instruction type
 - **datB** for RR, **IR[20:13]** for RI
- ALU function set according to operation type

Write Back

- To **Rc**

RI-type instructions

RI-type instructions (addq, subq, xor, bis, cmplt): $rc \leftarrow ra \text{ funct } ib$

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

IF: Instruction fetch

- $IR \leftarrow \text{IMemory}[PC]$
- $PC \leftarrow PC + 4$

ID: Instruction decode/register fetch

- $A \leftarrow \text{Register}[IR[25:21]]$
- $B \leftarrow IR[20:13]$

Ex: Execute

- $ALUOutput \leftarrow A \text{ op } B$

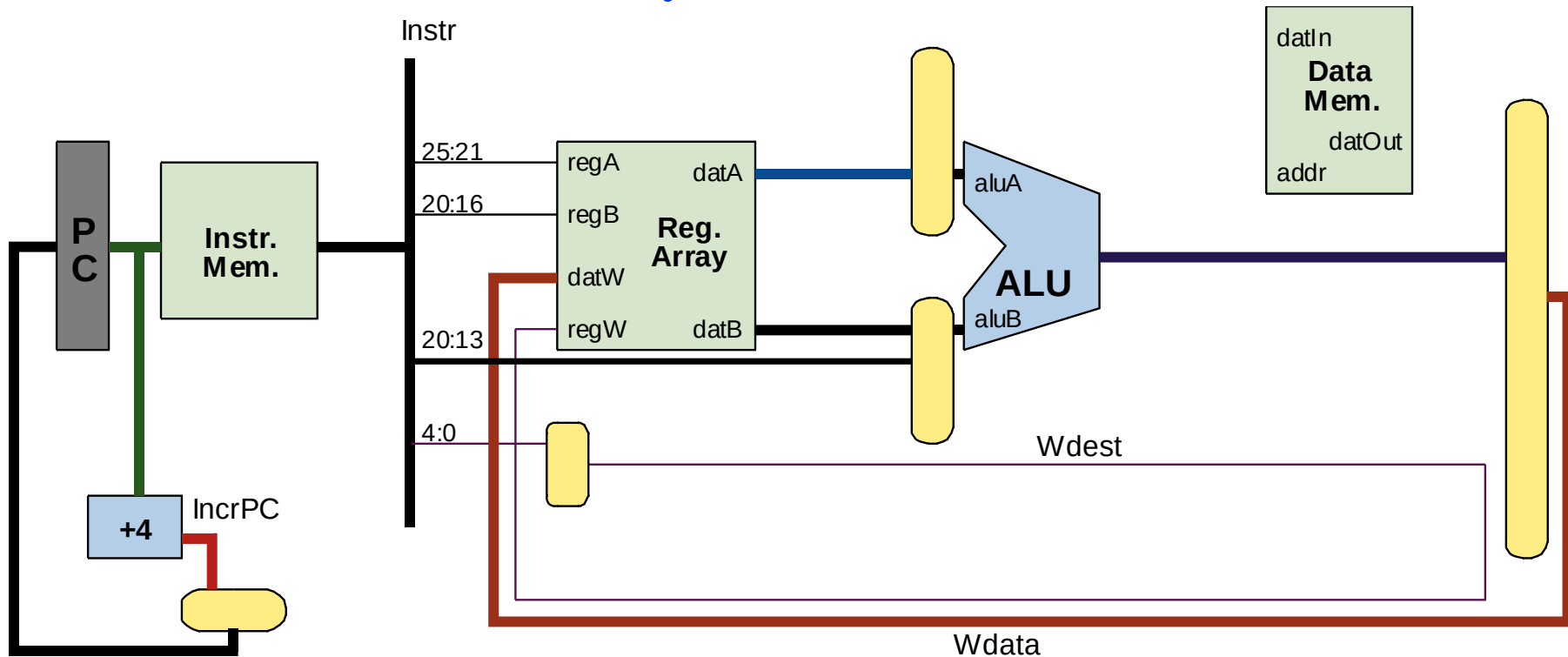
MEM: Memory

- nop

WB: Write back

- $\text{Register}[IR[4:0]] \leftarrow ALUOutput$

Active Datapath for RR & RI



ALU Operation

- Input B selected according to instruction type
 - **datB** for RR, **IR[20:13]** for RI
- ALU function set according to operation type

Write Back

- To **Rc**

Load instruction

Load: $Ra \leftarrow Mem[Rb + offset]$

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

IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $PC \leftarrow PC + 4$

ID: Instruction decode/register fetch

- $B \leftarrow Register[IR[20:16]]$

Ex: Execute

- $ALUOutput \leftarrow B + SignExtend(IR[15:0])$

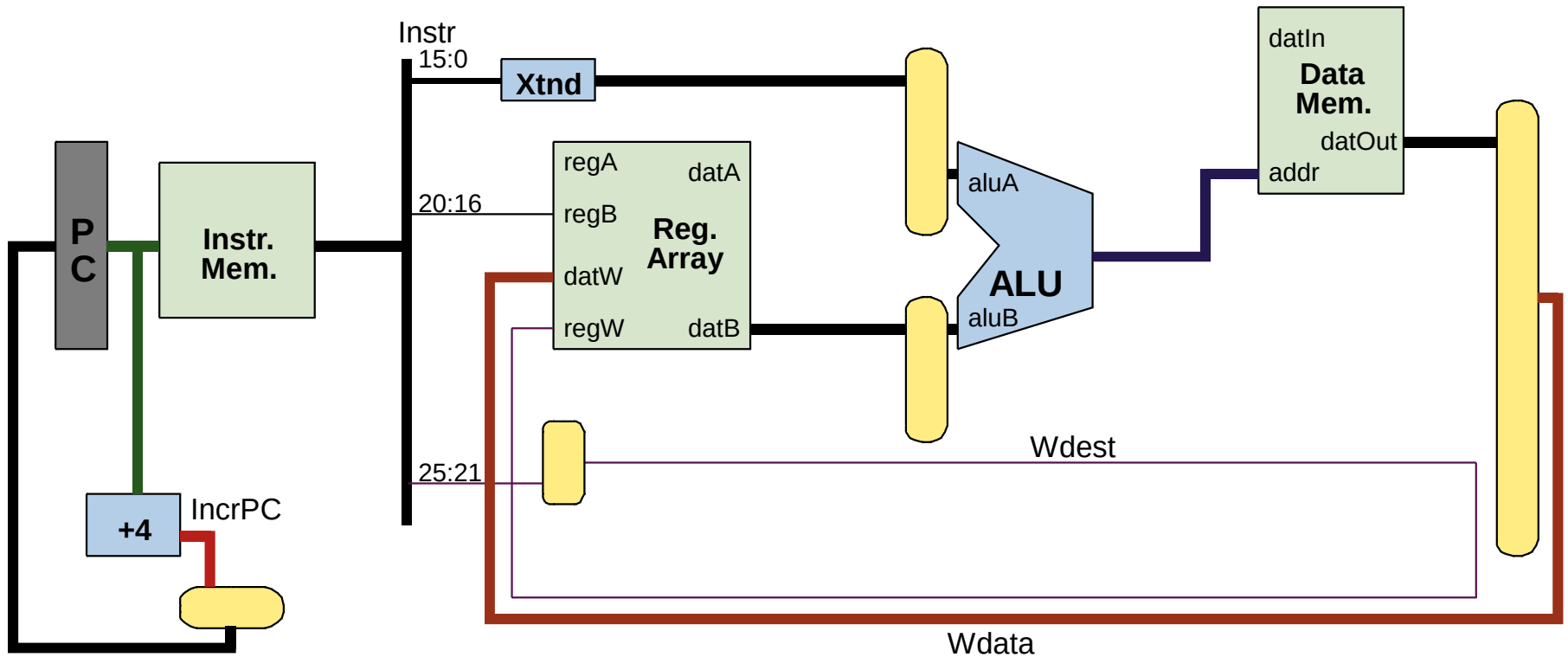
MEM: Memory

- $Mem-Data \leftarrow DMemory[ALUOutput]$

WB: Write back

- $Register[IR[25:21]] \leftarrow Mem-Data$

Active Datapath for Load



ALU Operation

- Used to compute address
 - A input set to extended IR[15:0]
- ALU function set to add

Memory Operation

- Read

Write Back

- To Ra

Store instruction

Store: $\text{Mem}[\text{Rb} + \text{offset}] \leftarrow \text{Ra}$

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

IF: Instruction fetch

- $\text{IR} \leftarrow \text{IMemory}[\text{PC}]$
- $\text{PC} \leftarrow \text{PC} + 4$

ID: Instruction decode/register fetch

- $A \leftarrow \text{Register}[\text{IR}[25:21]]$
- $B \leftarrow \text{Register}[\text{IR}[20:16]]$

Ex: Execute

- $\text{ALUOutput} \leftarrow B + \text{SignExtend}(\text{IR}[15:0])$

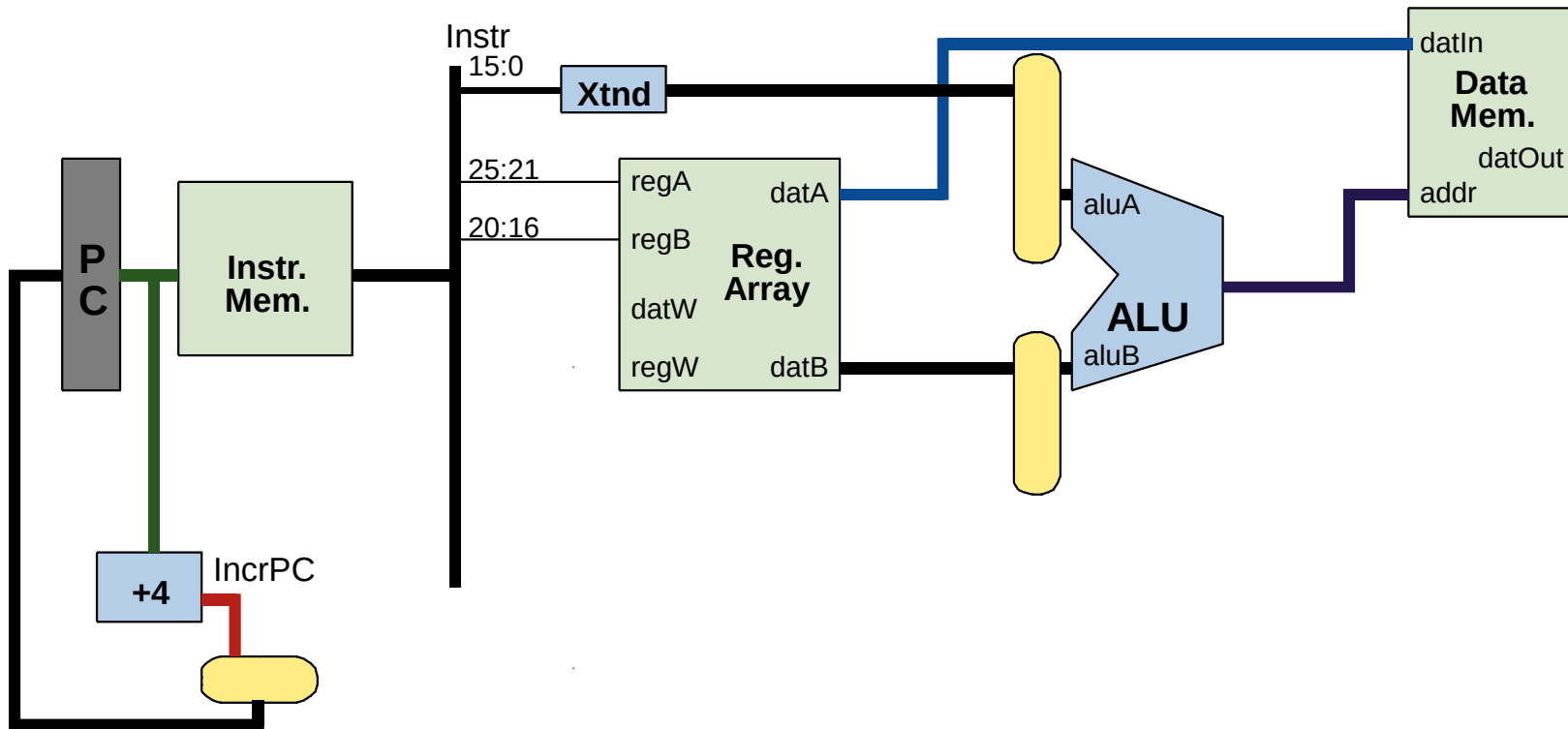
MEM: Memory

- $\text{DMemory}[\text{ALUOutput}] \leftarrow A$

WB: Write back

- nop

Active Datapath for Store



ALU Operation

- Used to compute address
 - A input set to extended IR[15:0]
- ALU function set to add

Memory Operation

- Write

Write Back

- None

Branch on equal

beq: $PC \leftarrow Ra == 0 ? PC + 4 + disp * 4 : PC + 4$

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

IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $incrPC \leftarrow PC + 4$

ID: Instruction decode/register fetch

- $A \leftarrow Register[IR[25:21]]$

Ex: Execute

- $Target \leftarrow incrPC + SignExtend(IR[20:0]) \ll 2$
- $Z \leftarrow (A == 0)$

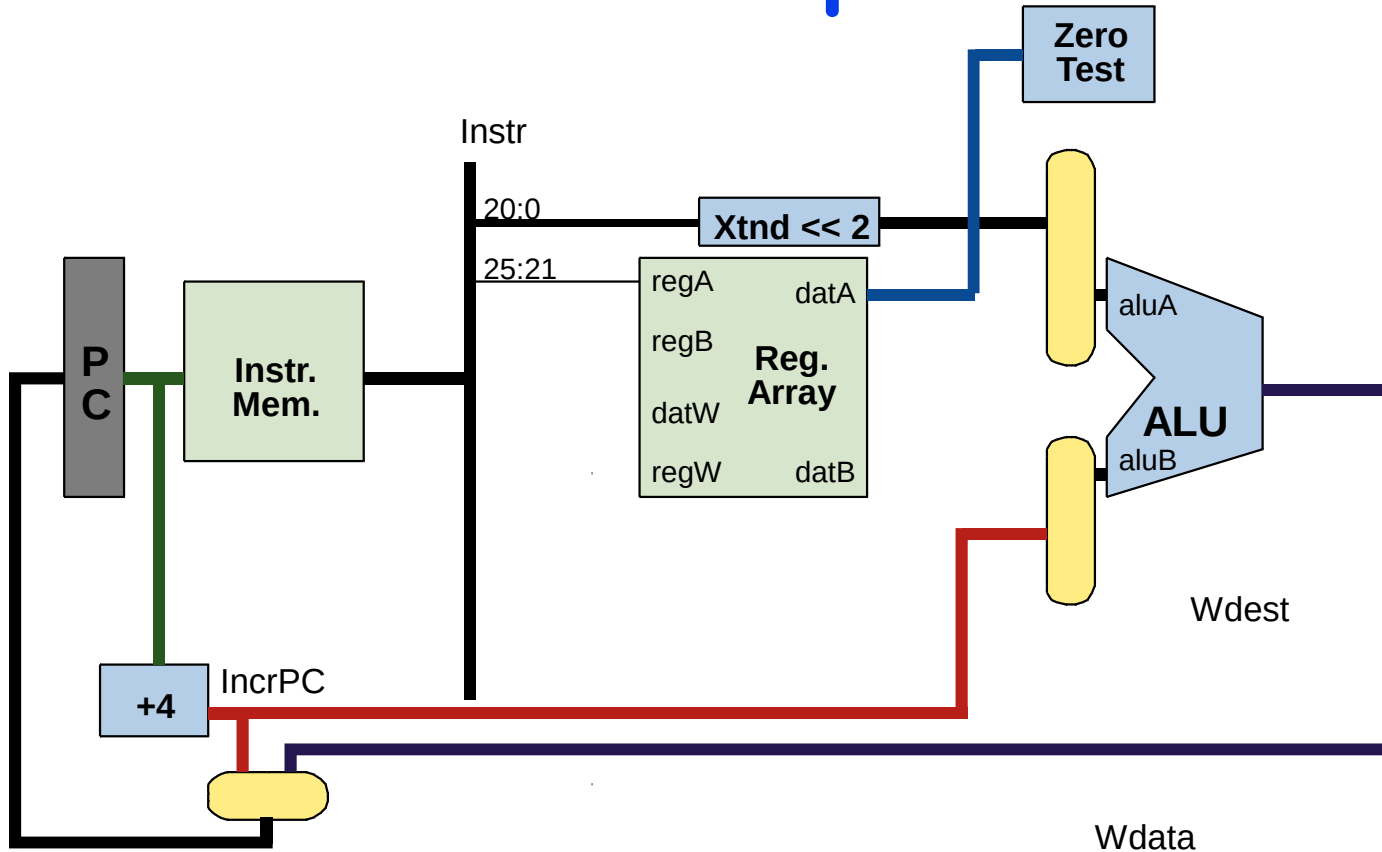
MEM: Memory

- $PC \leftarrow Z ? Target : incrPC$

WB: Write back

- nop

Active Datapath for Branch



ALU Computes target

- A = shifted, extended IR[20:0]
- B = IncrPC
- Function set to add

PC Selection

- Target for taken branch
- IncrPC for not taken

Write Back

- None

Zero Test

- Branch condition depends on if $\text{Reg}[\text{Ra}] == 0$

Branch to Subroutine

Branch Subroutine (**bsr**): $Ra \leftarrow PC + 4$; $PC \leftarrow PC + 4 + disp * 4$

0x34	ra	disp
31-26	25-21	20-0

IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $incrPC \leftarrow PC + 4$

ID: Instruction decode/register fetch

- nop

Ex: Execute

- $Target \leftarrow incrPC + SignExtend(IR[20:0]) \ll 2$

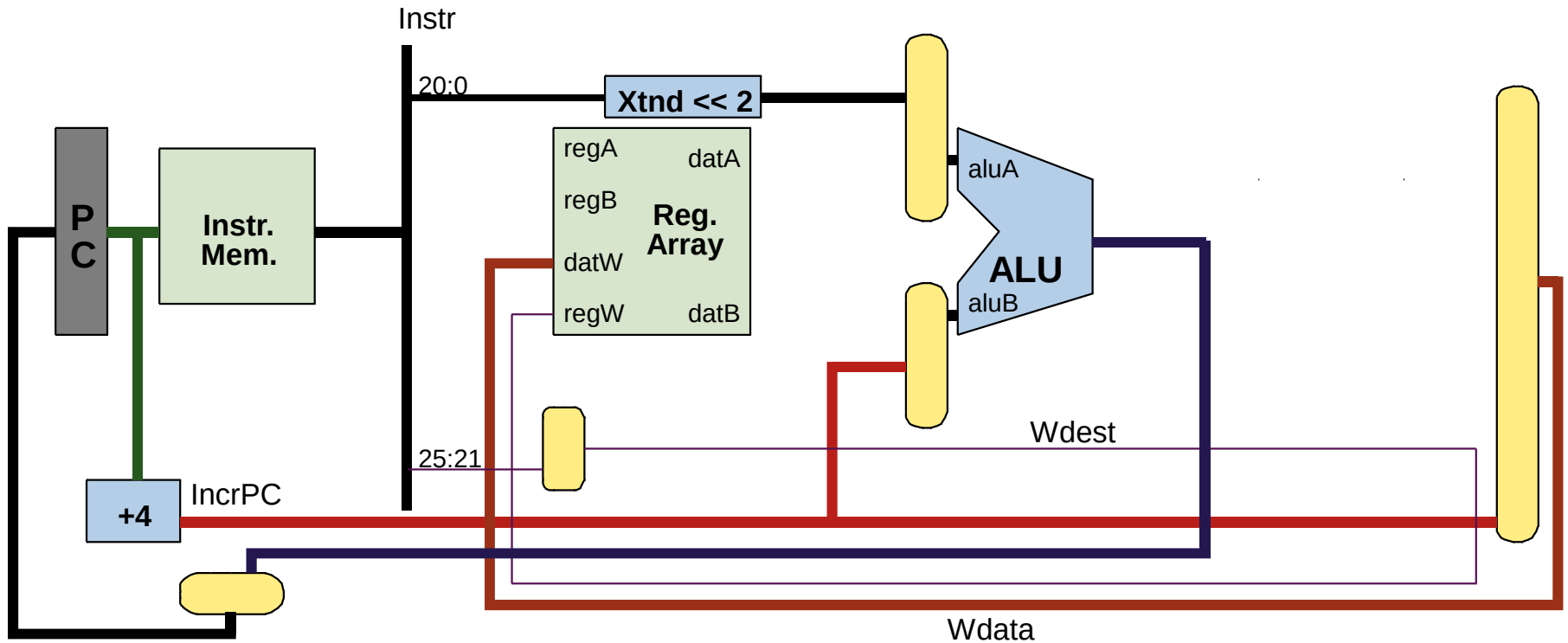
MEM: Memory

- $PC \leftarrow Target$

WB: Write back

- $Register[IR[25:21]] \leftarrow incrPC$

Active Datapath for BSR



ALU Computes target

- A = shifted, extended IR[20:0]
- B = IncrPC
- Function set to add

PC Selection

- Always target

Write Back

- Incremented PC as data

Jump

jmp, jsr, ret: $Ra \leftarrow PC+4; PC \leftarrow Rb$

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

IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $incrPC \leftarrow PC + 4$

ID: Instruction decode/register fetch

- $B \leftarrow Register[IR[20:16]]$

Ex: Execute

- $Target \leftarrow B$

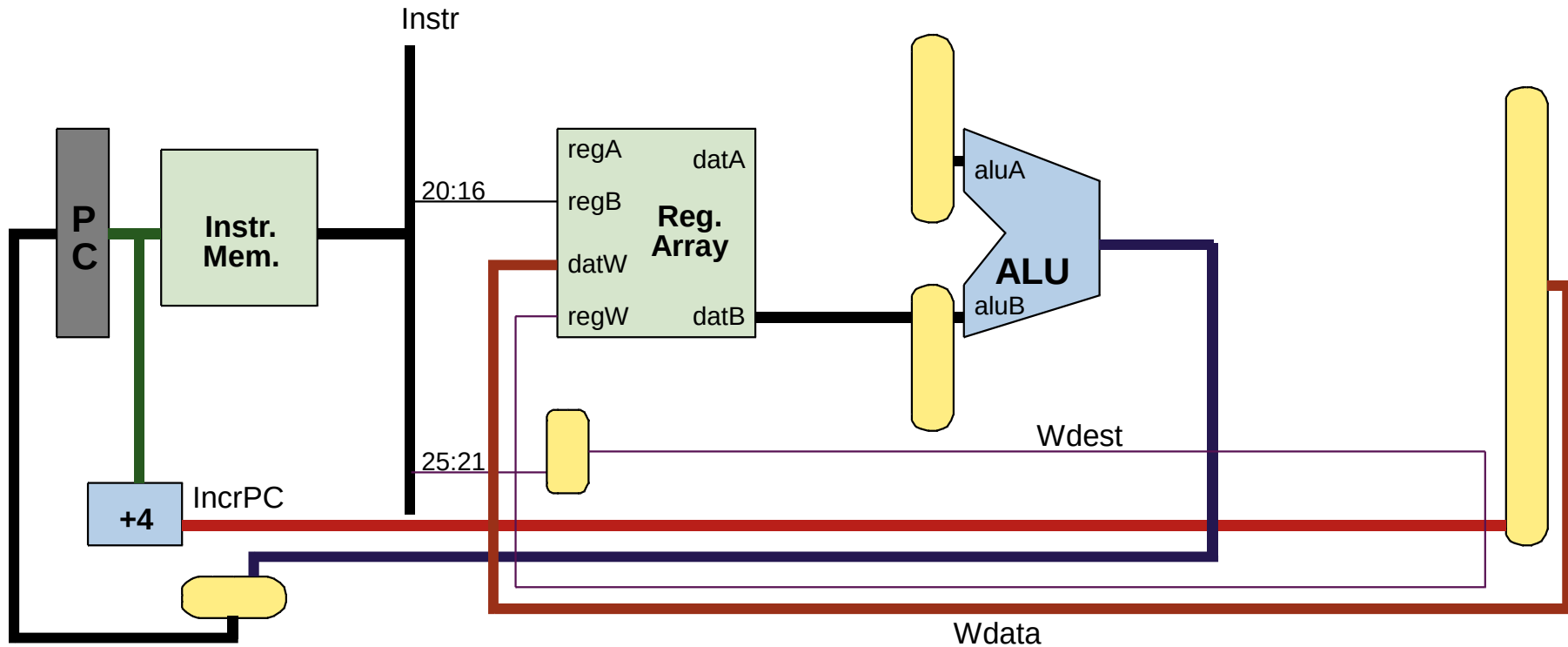
MEM: Memory

- $PC \leftarrow target$

WB: Write back

- $Register[IR[25:21]] \leftarrow incrPC$

Active Datapath for Jumps



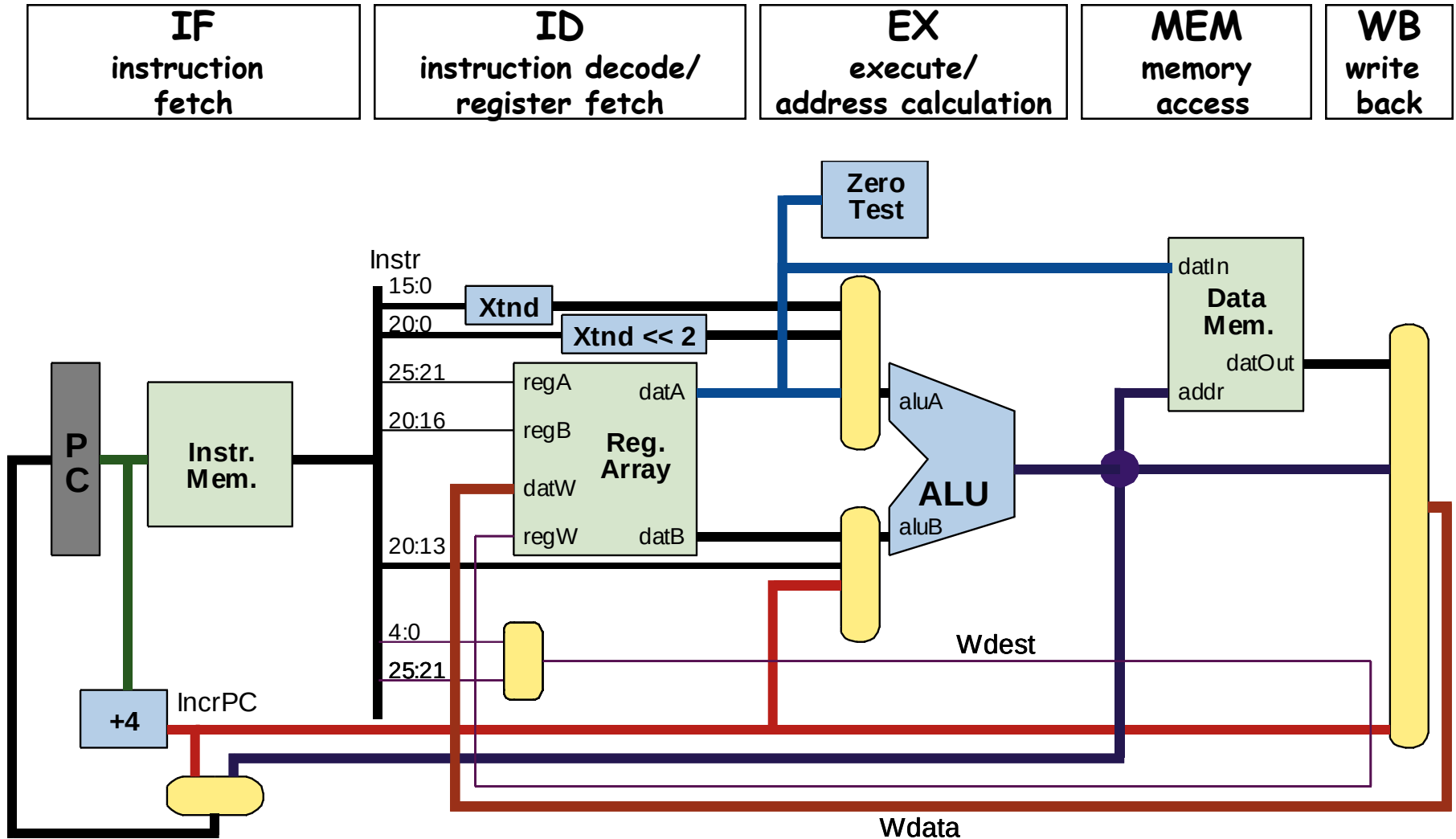
ALU Operation

- Used to compute target
 - B input set to Rb
- ALU function set to select B

Write Back

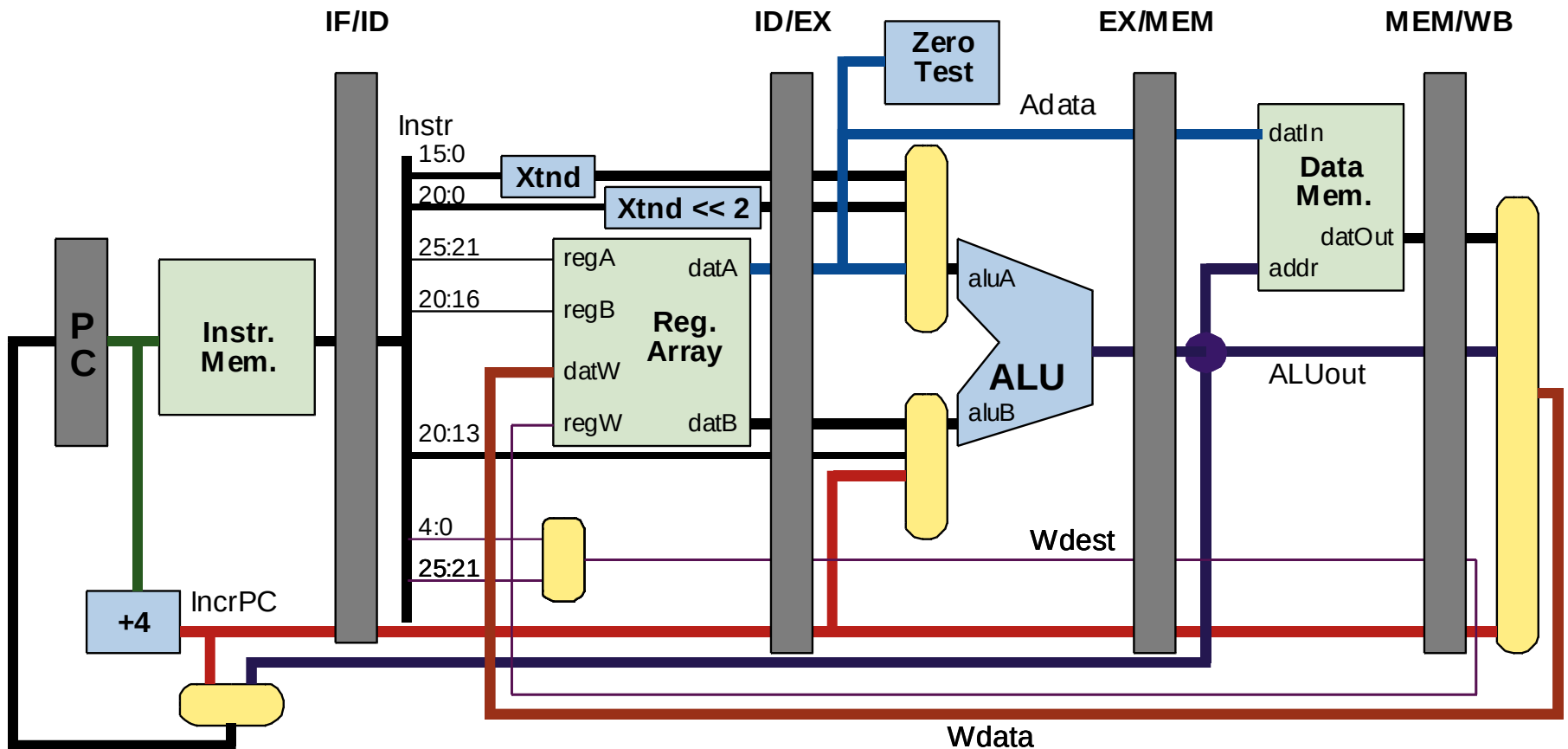
- To Ra
- IncrPC as data

Complete Datapath



PIPELINING

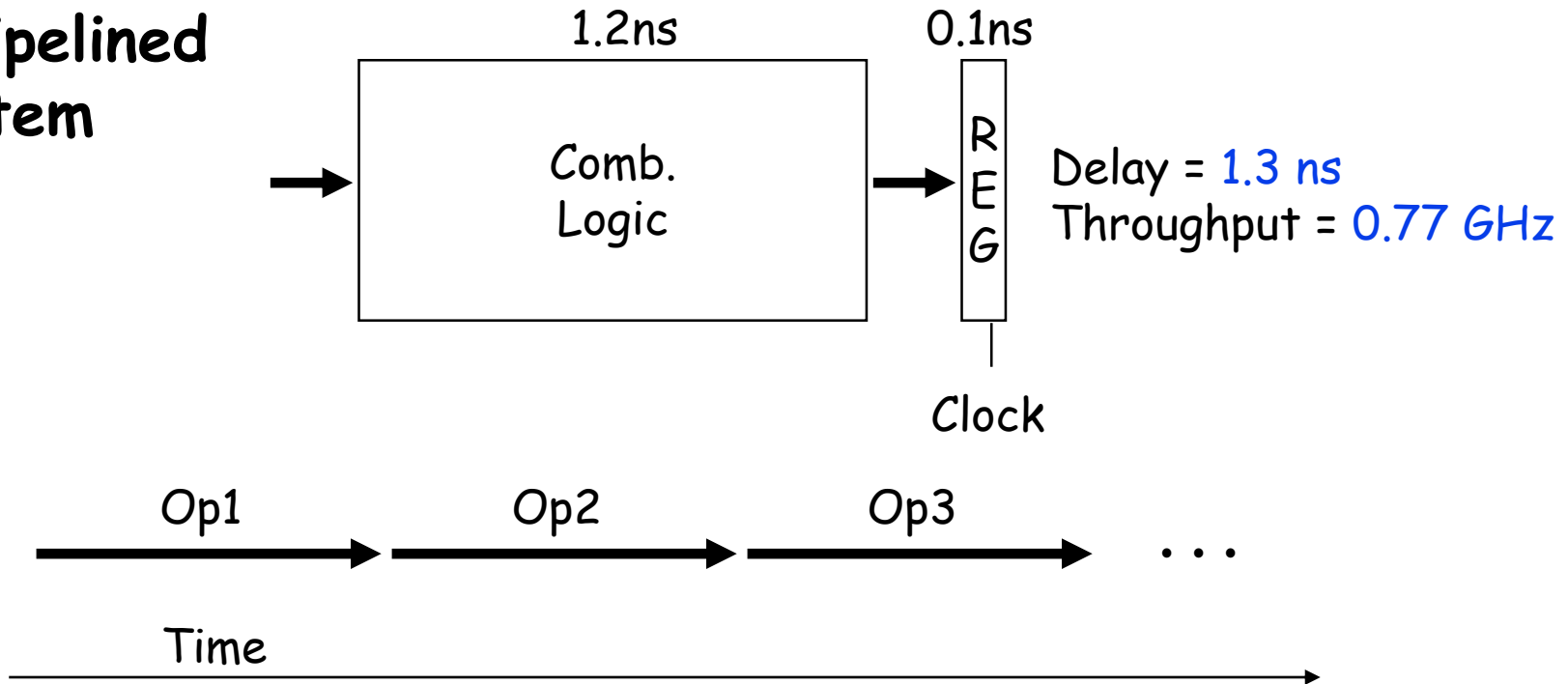
Pipelined Datapath



Same datapath, $\approx 5X$ higher peak throughput!

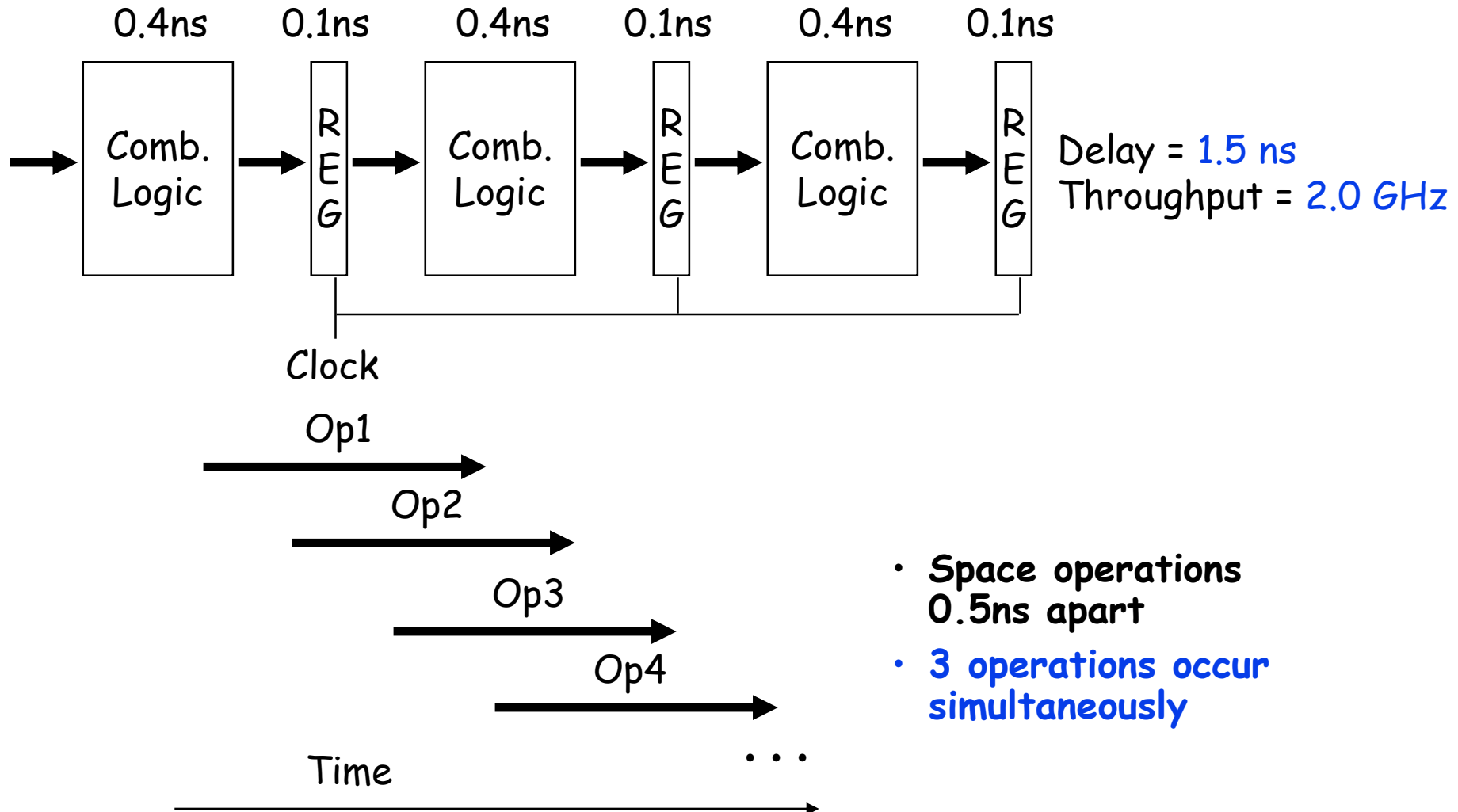
Pipelining Basics

Unpipelined System

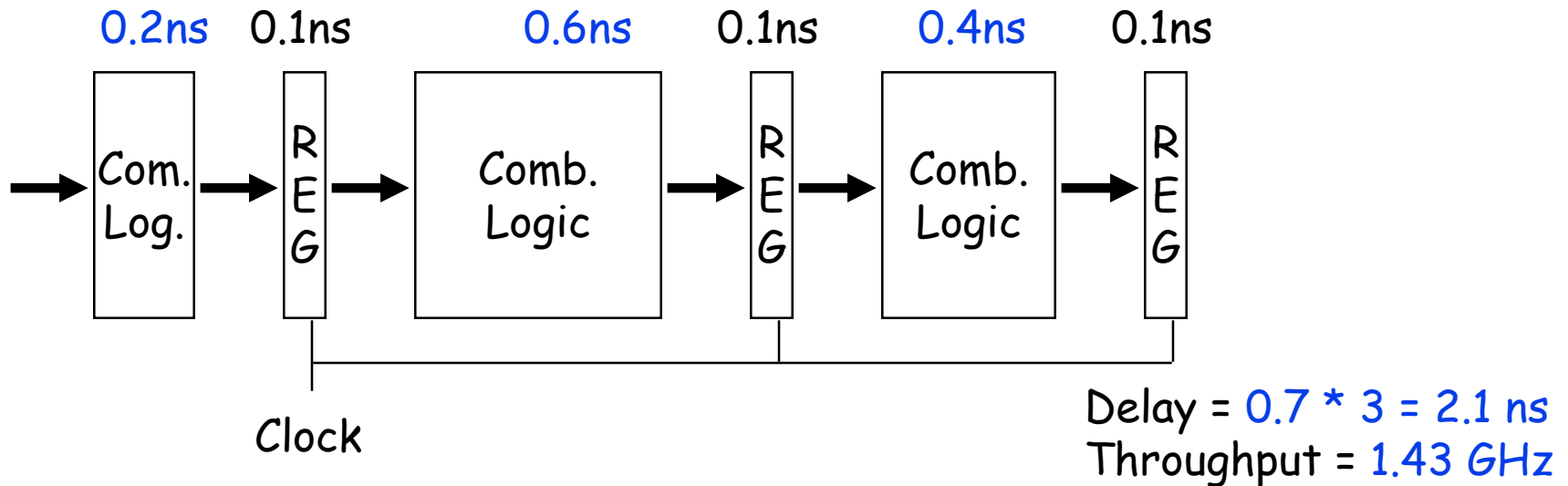


- One operation must complete before next can begin
- Operations spaced 1.3ns apart

3-Stage Pipelining

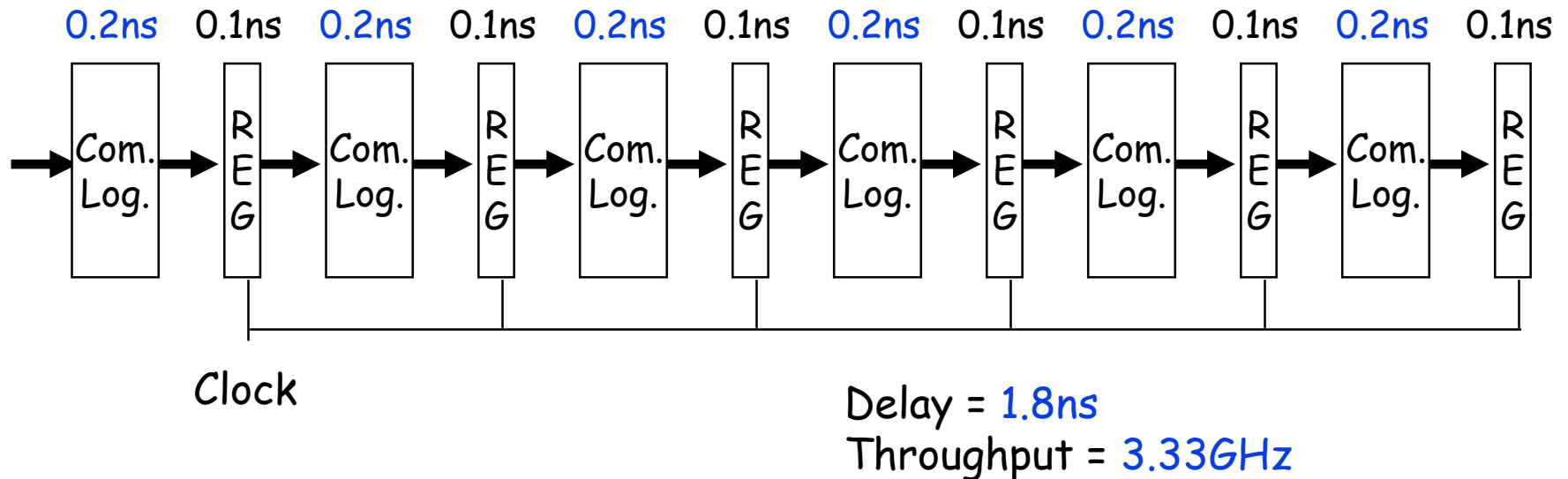


Limitation: Non-uniform Pipelining



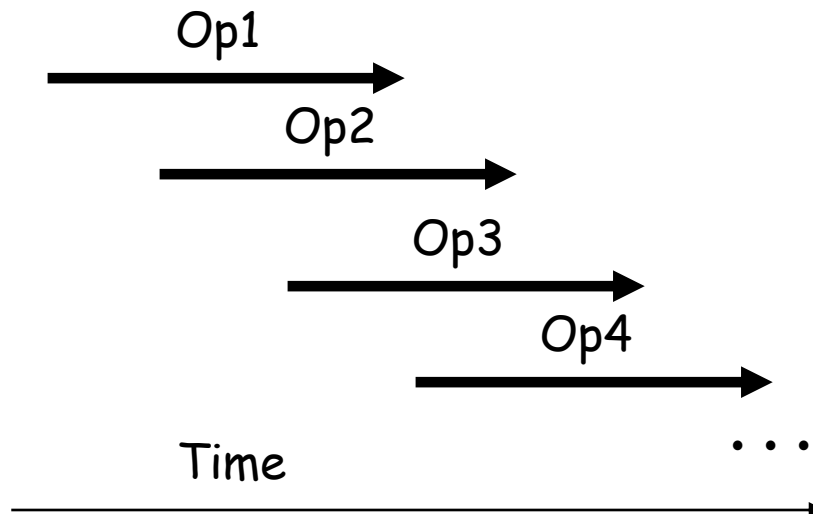
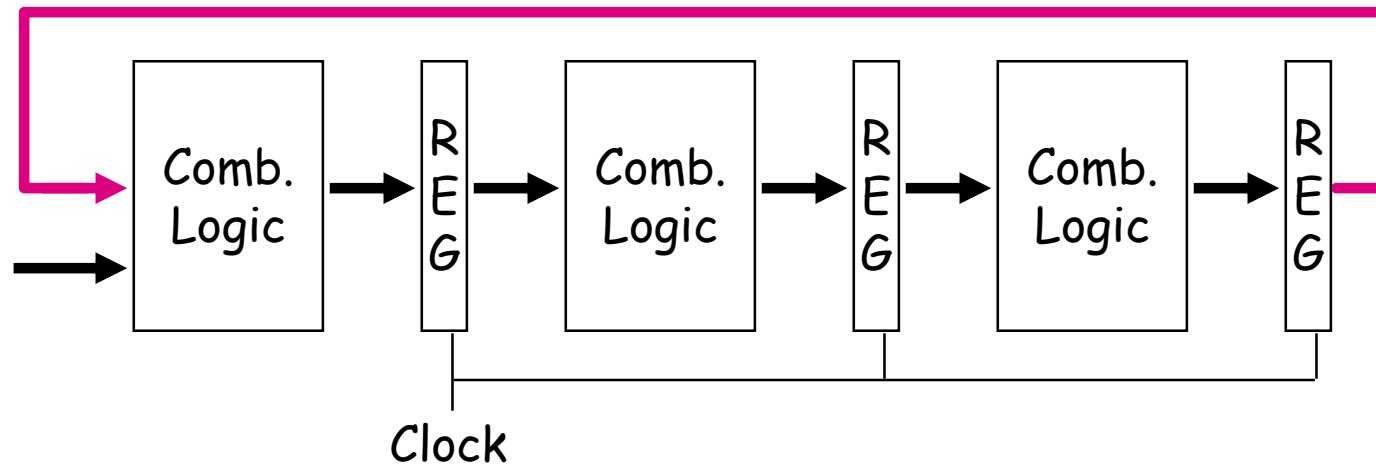
- **Throughput limited by slowest stage**
 - Delay determined by clock period * number of stages
- **Must attempt to balance stages**

Limitation: Deep Pipelines



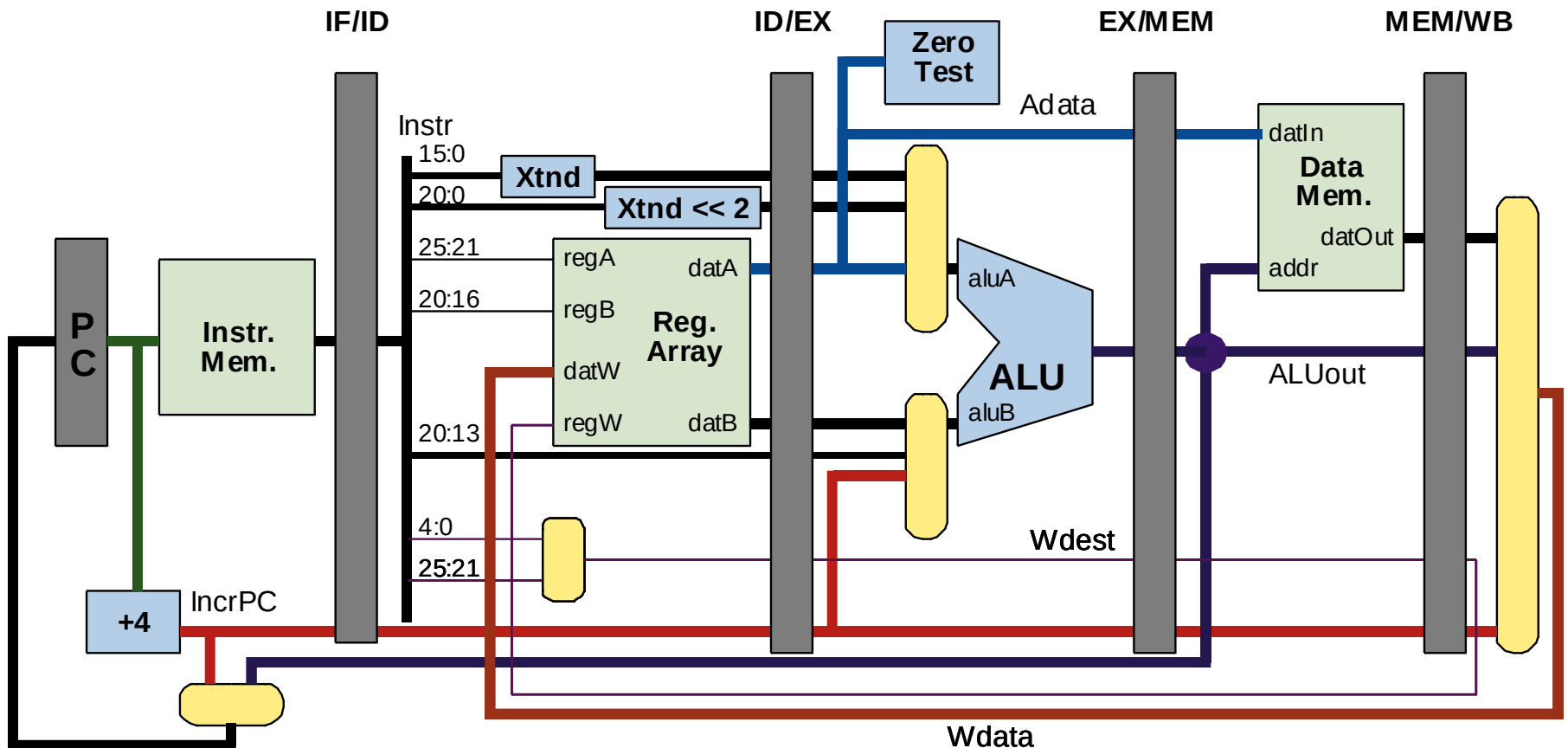
- Diminishing returns as add more pipeline stages
- Register delays become limiting factor
 - Increased latency
 - Small throughput gains
 - (Other architectural problems in practice that flush pipeline)

Limitation: Sequential Dependencies



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

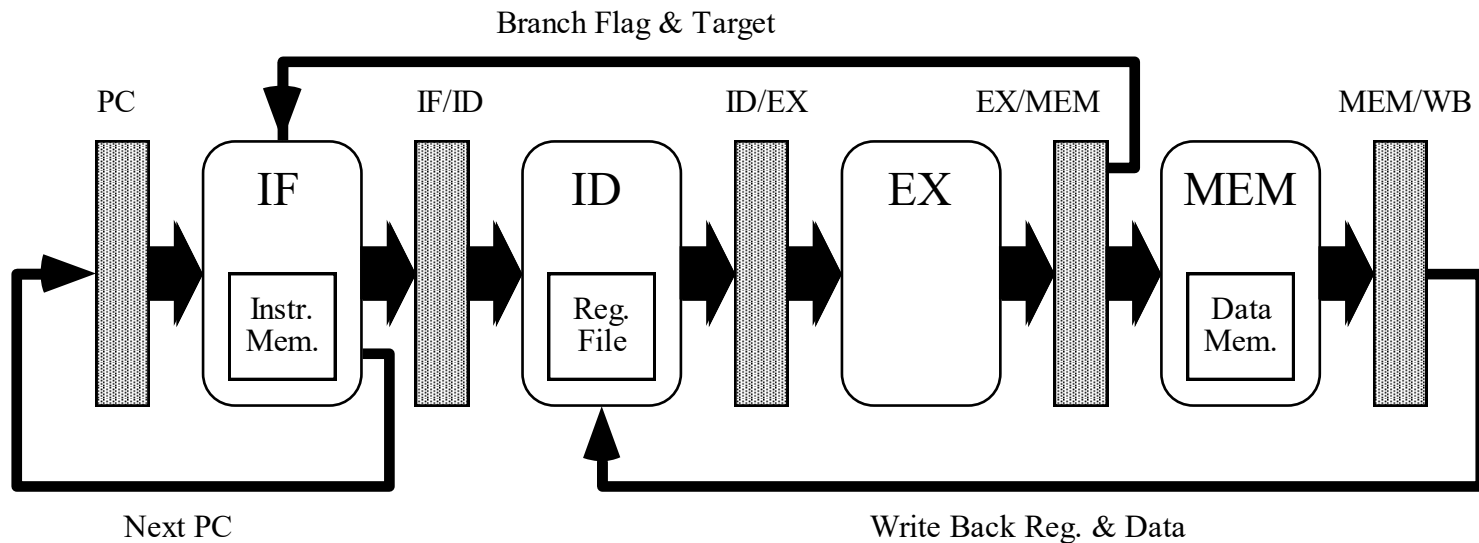
Pipelined Datapath



Pipe Registers

- Inserted between stages
- Labeled by preceding & following stage

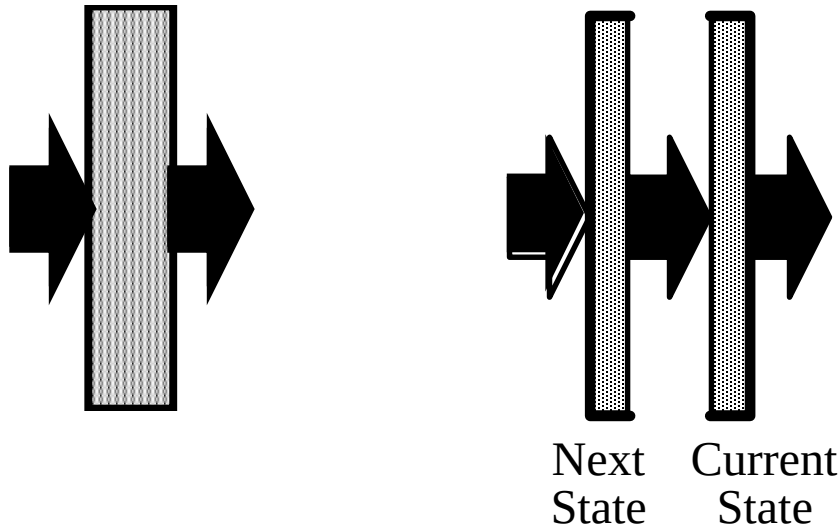
Pipeline Structure



Notes

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

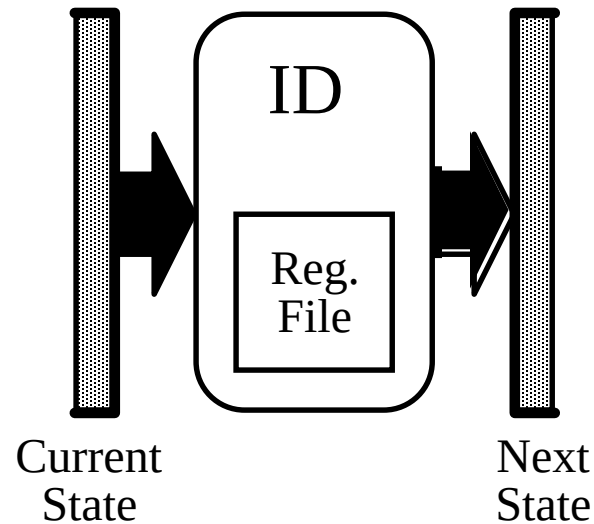
Pipe Register



Operation

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

Pipeline Stage



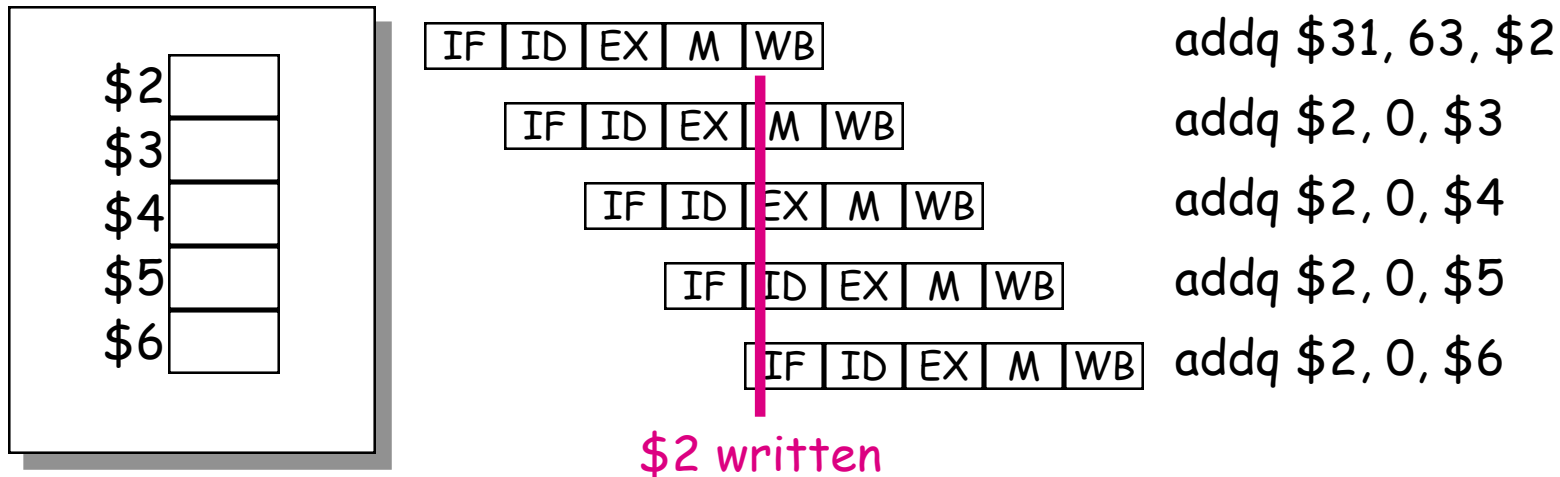
Operation

- **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 Alpha Pipeline

Problem

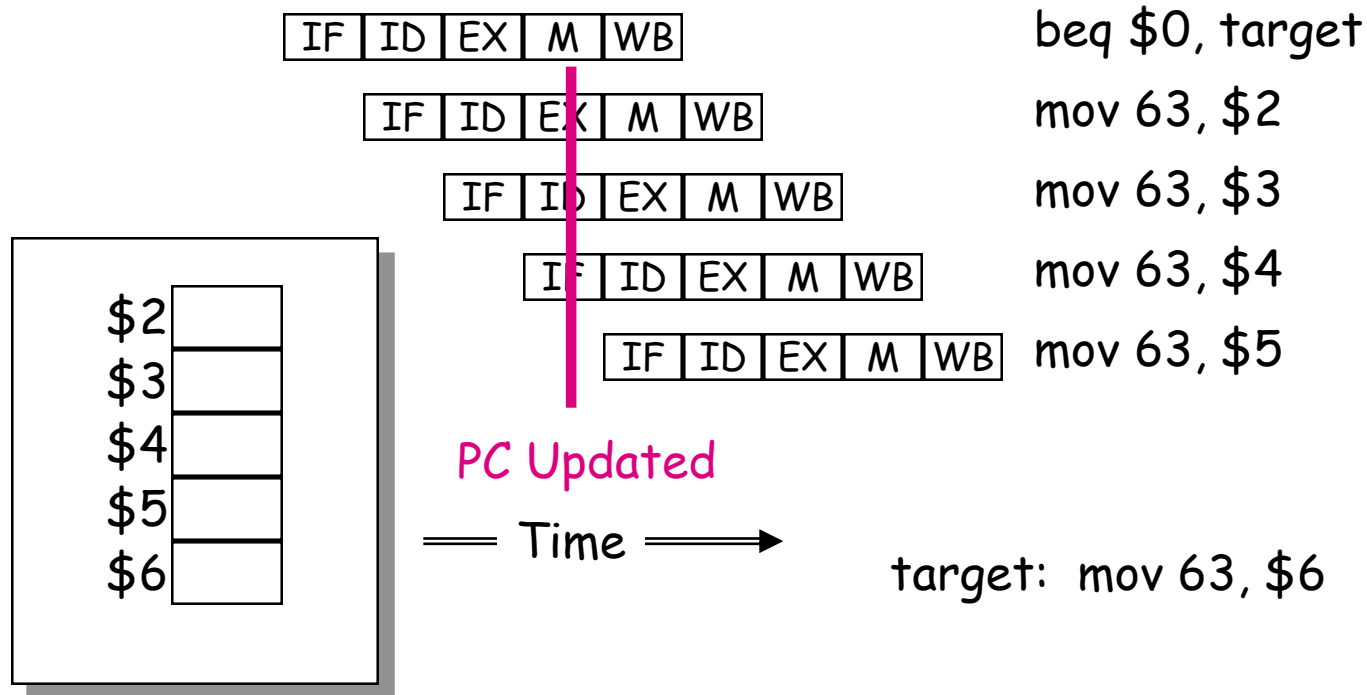
- Registers read in ID, and written in WB
- Must resolve conflict between instructions competing for registers
 - Assume reads get a value written in same stage
- But what about intervening instructions?
- E.g., suppose initially \$2 is zero:



Control Hazards in Alpha Pipeline

Problem

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



Conclusions

RISC Design Simplifies Implementation

- Small number of instruction formats
- Simple instruction processing

RISC Leads Naturally to Pipelined Implementation

- Partition activities into stages
- Each stage simple computation

We're not done yet!

- Need to deal with data & control hazards

ADVANCED PIPELINING

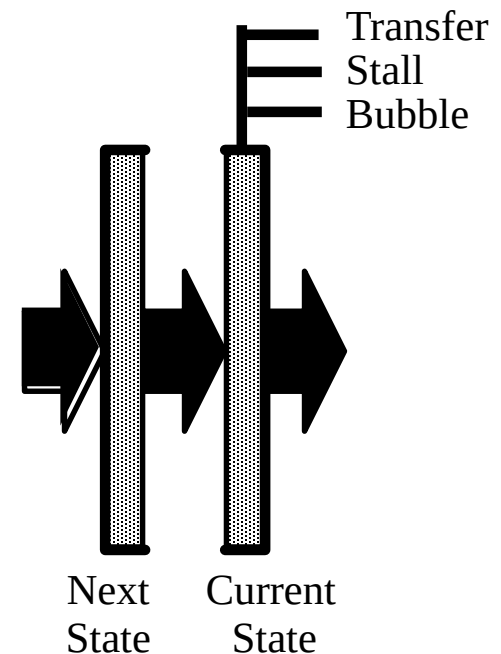
Handling Hazards by Stalling

Idea

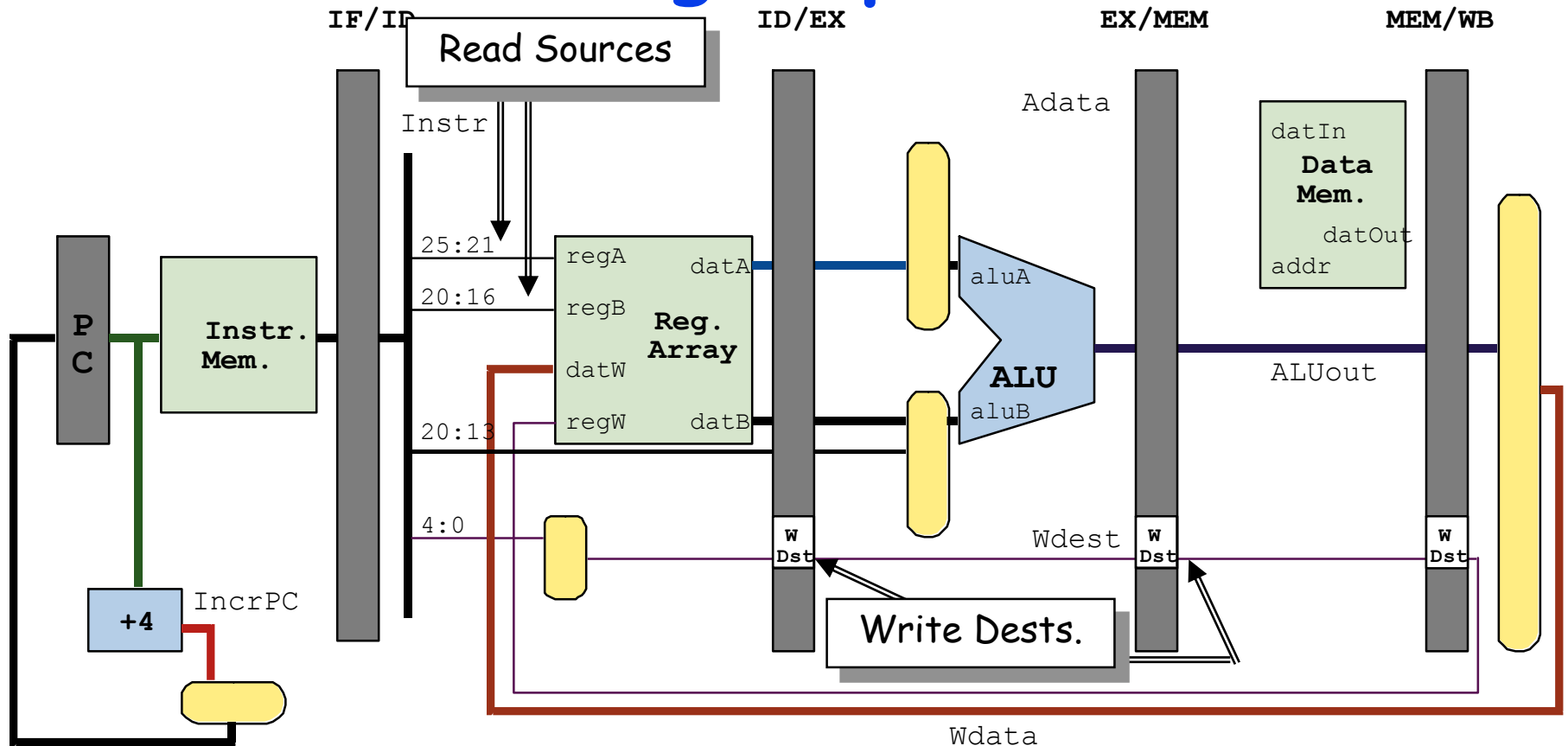
- Delay instruction until hazard eliminated
- Put “bubble” into pipeline
 - Dynamically generated NOP

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 NOP
 - E.g., stage logic designed so that 0 is like NOP



Detecting Dependencies



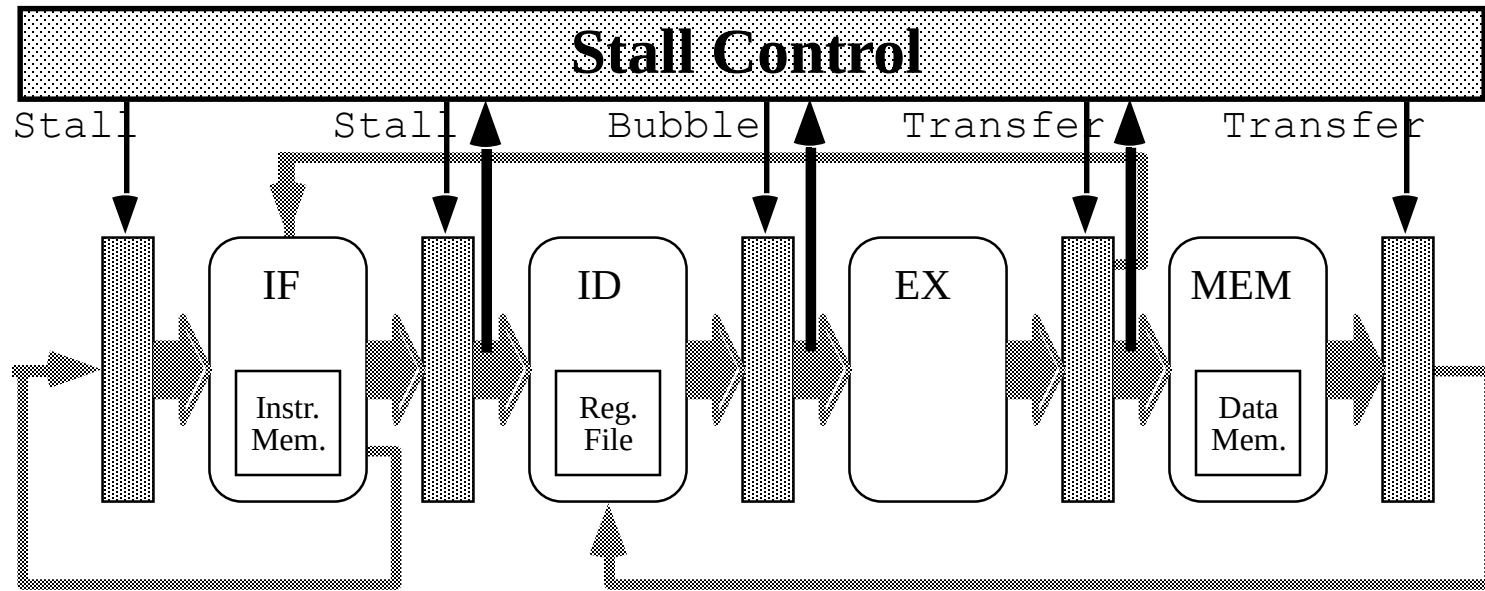
Pending Register Reads

- By instruction in ID
- ID_in.IR[25:21]: Operand A
- ID_in.IR[20:16]: Operand B
 - Only for RR
 - 47 -

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

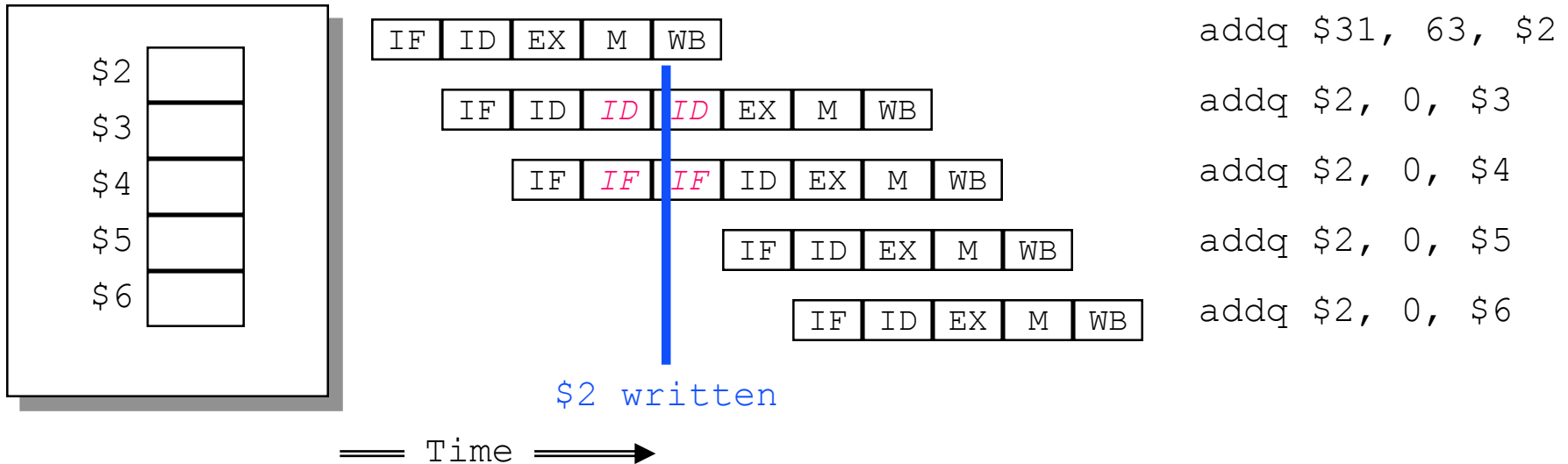
Effect

- 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)

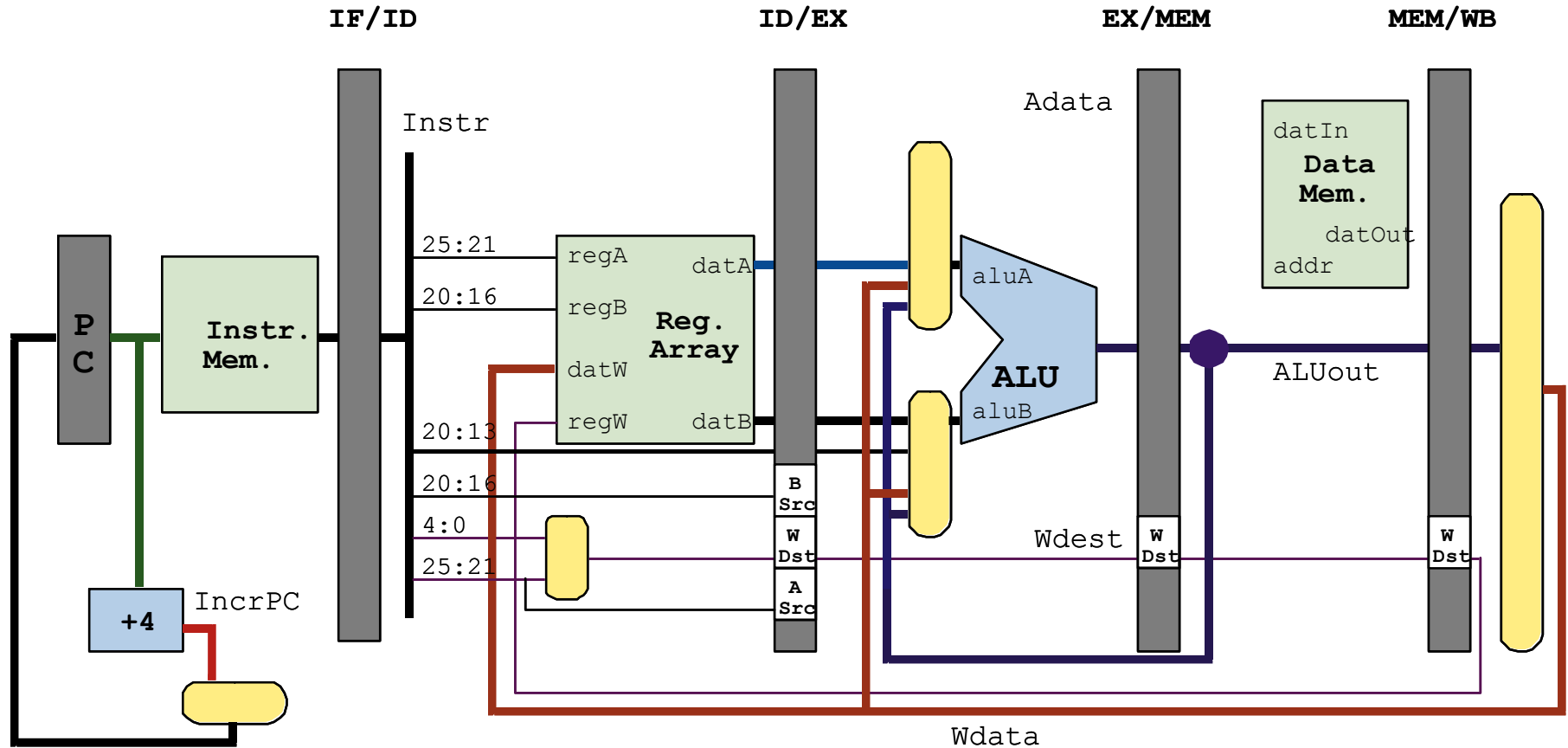
Observation

- 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



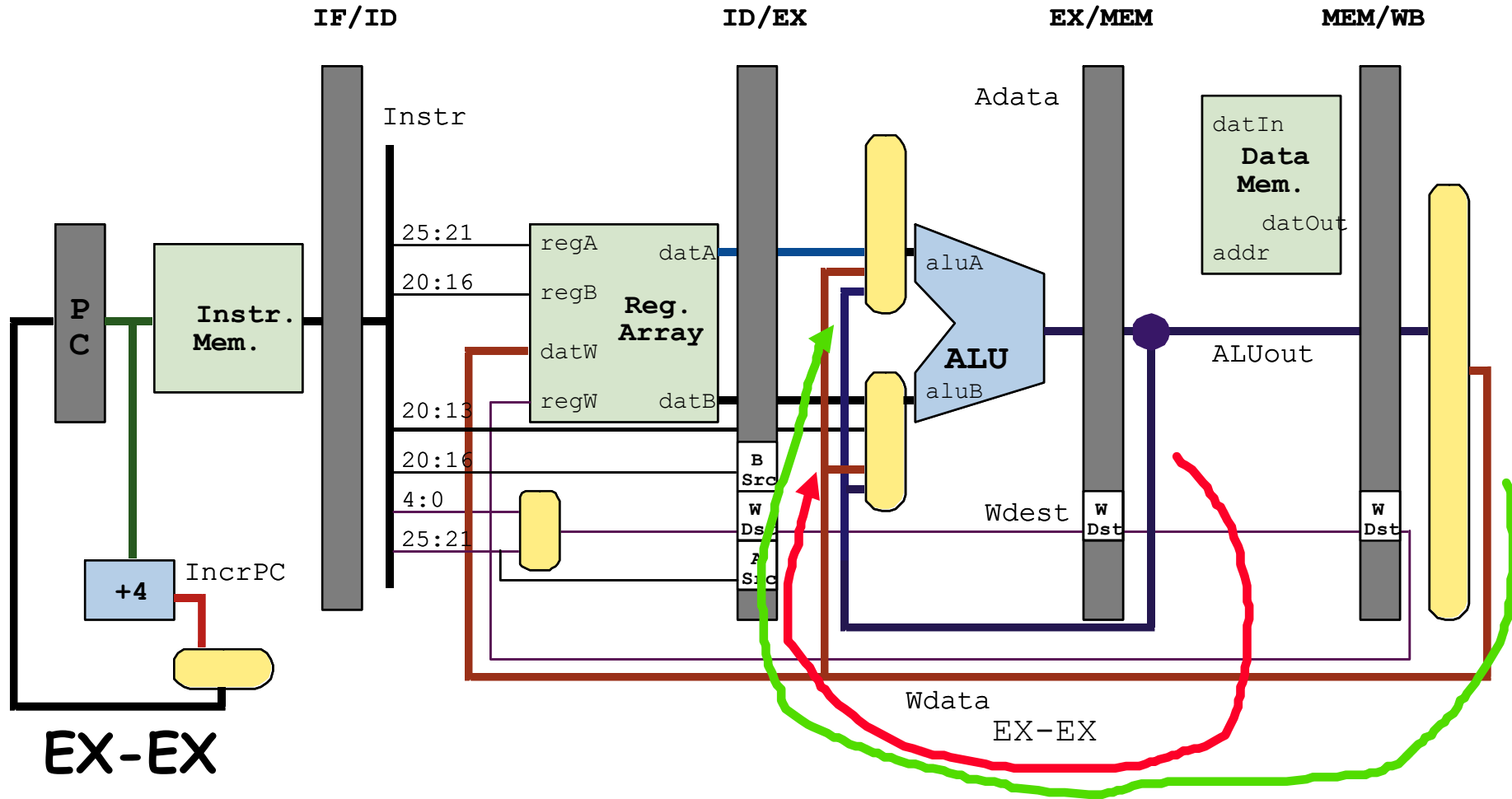
Operand Destinations

- ALU input A
 - Register EX_in.ASrc
 - ALU input B
 - Register EX_in.BSrc
- 52 -

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

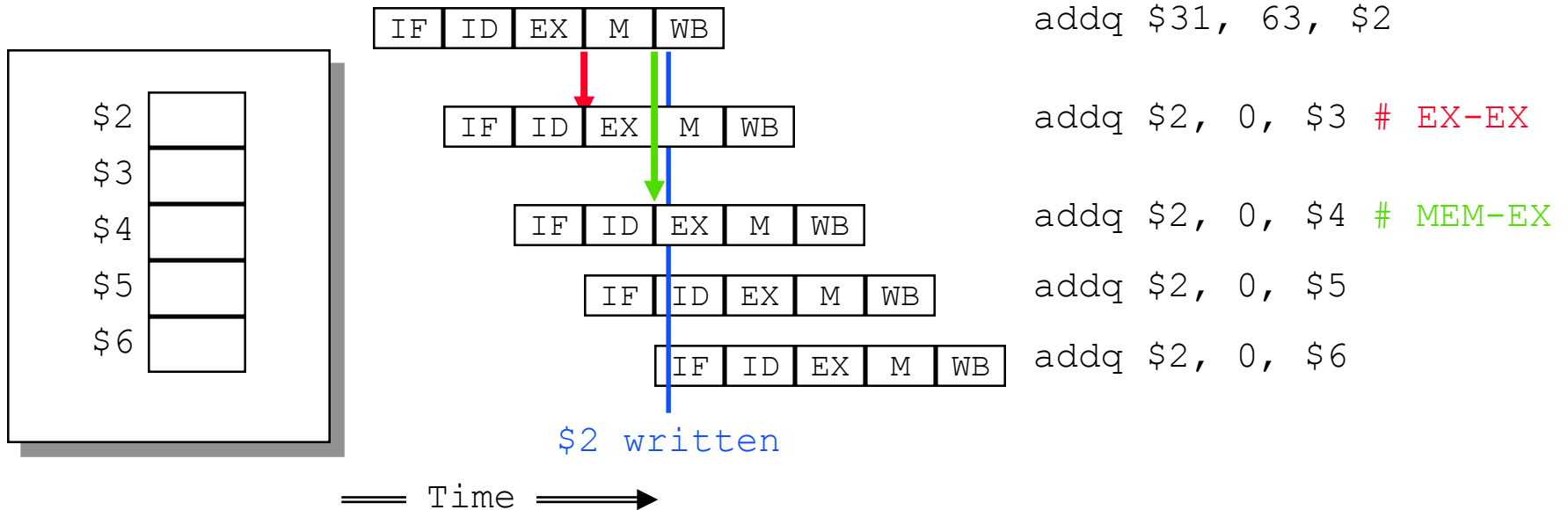
MEM-EX

- 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: $Ra \leftarrow Mem[Rb + offset]$

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

Store: $Mem[Rb + offset] \leftarrow Ra$

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

ID: Instruction decode/register fetch

- **Store:** $A \leftarrow Register[IR[25:21]]$
- $B \leftarrow Register[IR[20:16]]$

MEM: Memory

- **Load:** $Mem-Data \leftarrow DMemory[ALUOutput]$
- **Store:** $DMemory[ALUOutput] \leftarrow A$

WB: Write back

- **Load:** $Register[IR[25:21]] \leftarrow Mem-Data$

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

Some Hazards with Loads & Stores

Data Generated by Load

Load-Store Data

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

Data Generated by Store

Store-Load Data

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

*Not a
concern
for us*

Load-ALU

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

Data Generated by ALU

ALU-Store (or Load) Addr

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

Load-Store (or Load) Addr.

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

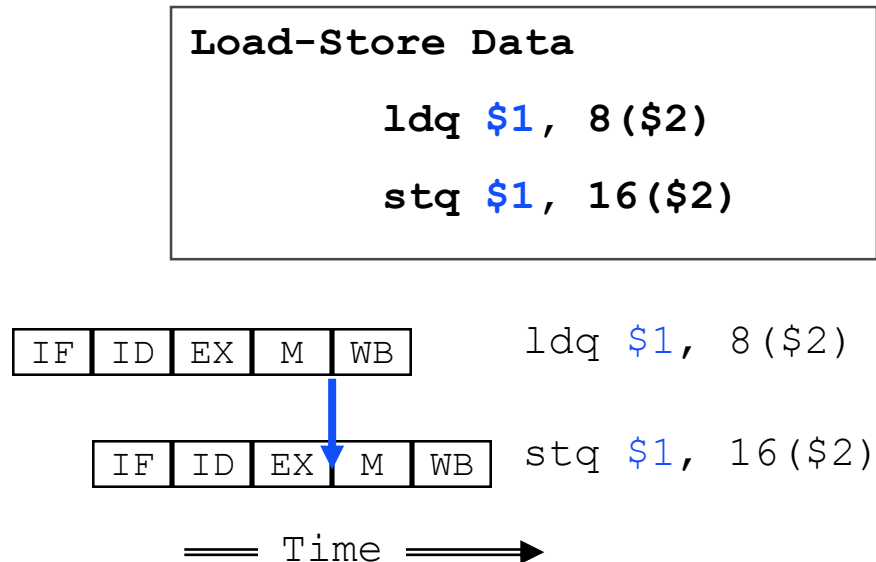
ALU-Store Data

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

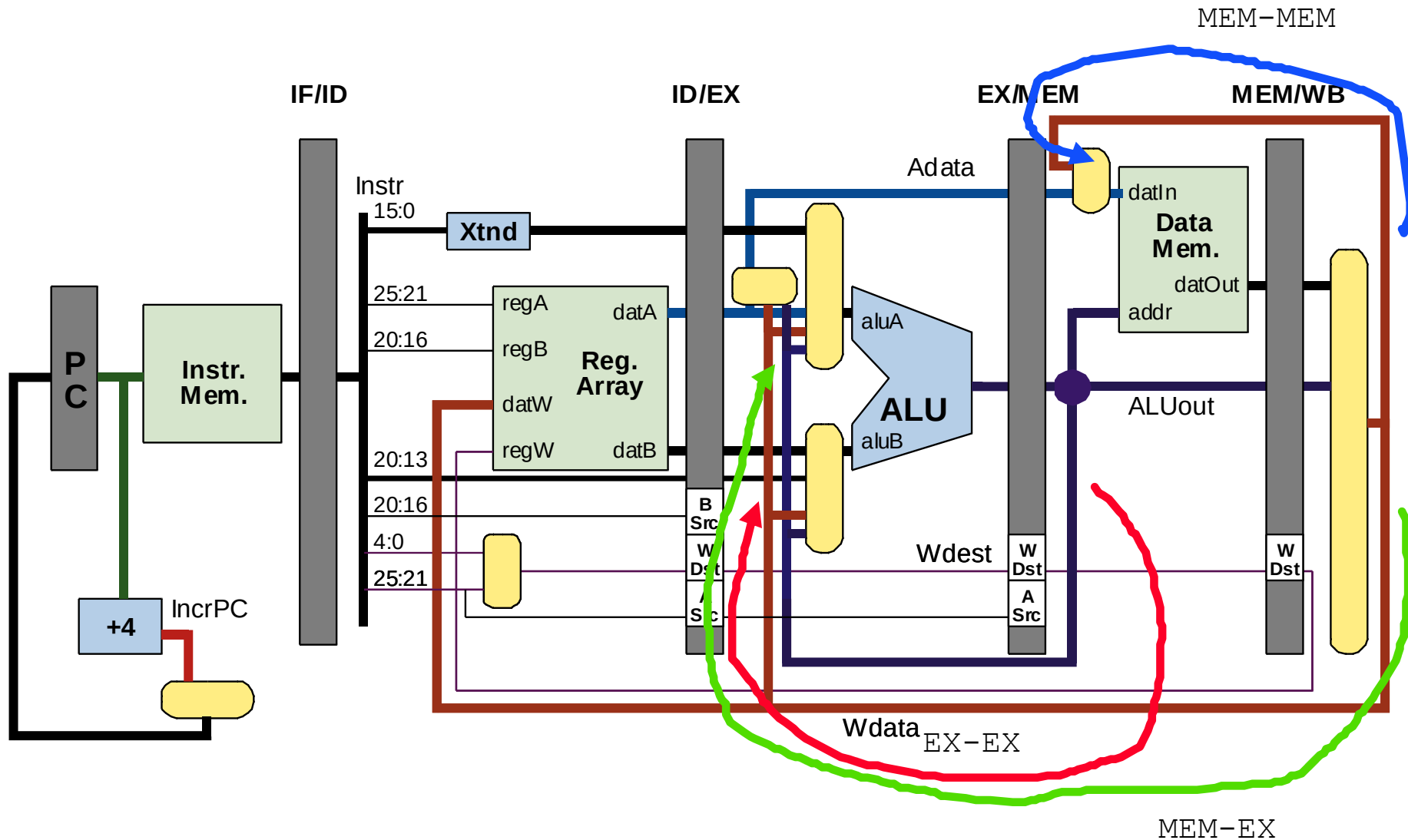
MEM-MEM Forwarding

Condition

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



Complete Bypassing for ALU & L/S



Some Hazards with Loads & Stores

Data Generated by Load

Load-Store Data

MEM-MEM

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

Load-ALU

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

EX-EX

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

ALU-Store Data

EX-EX

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

Impact of Forwarding

Single Remaining Unsolved Hazard Class

- Load followed by ALU operation / address calculation

Load-ALU

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

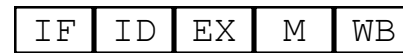
```
addq $2, $1, $2
```

Load-Store (or Load) Addr.

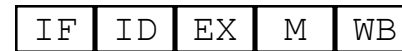
```
ldq $1, 8($2)
```

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

Just Forward?



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

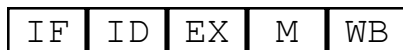


```
addq $2, $1, $2
```

== Time ==>

Value not available soon enough!

With 1 Cycle Stall



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



```
addq $2, $1, $2
```

== Time ==>

Then can use MEM-EX forwarding

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
```

Still More Data Hazards

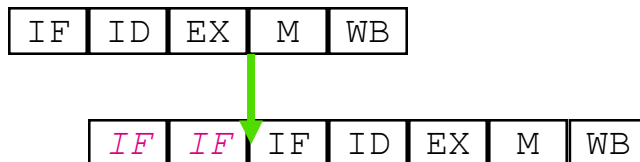
Jump Uses Register Data

- Generated by ALU instruction
- Read from register in ID

Handling

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

But fetch stalls



ALU-Jump

```
addq $2, $3, $1
```

```
jsr $26 ($1), 1
```

Distant ALU-Jump

```
addq $2, $3, $1
```

bis \$31, \$31, \$31

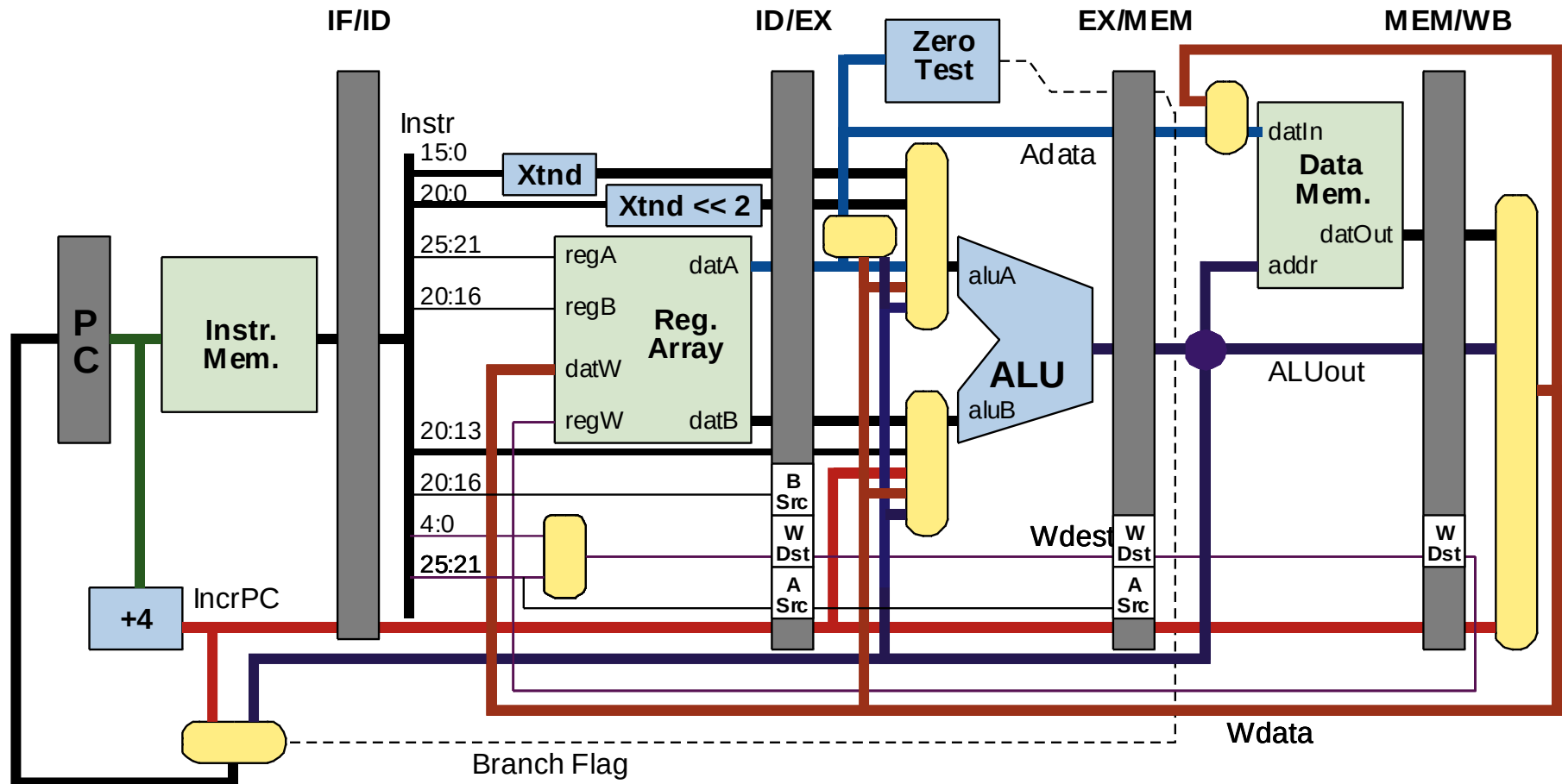
```
jmp $31 ($1), 1
```

Load-Jump

lw \$26, 8(\$sp)

```
ret $31 ($26), 1
```

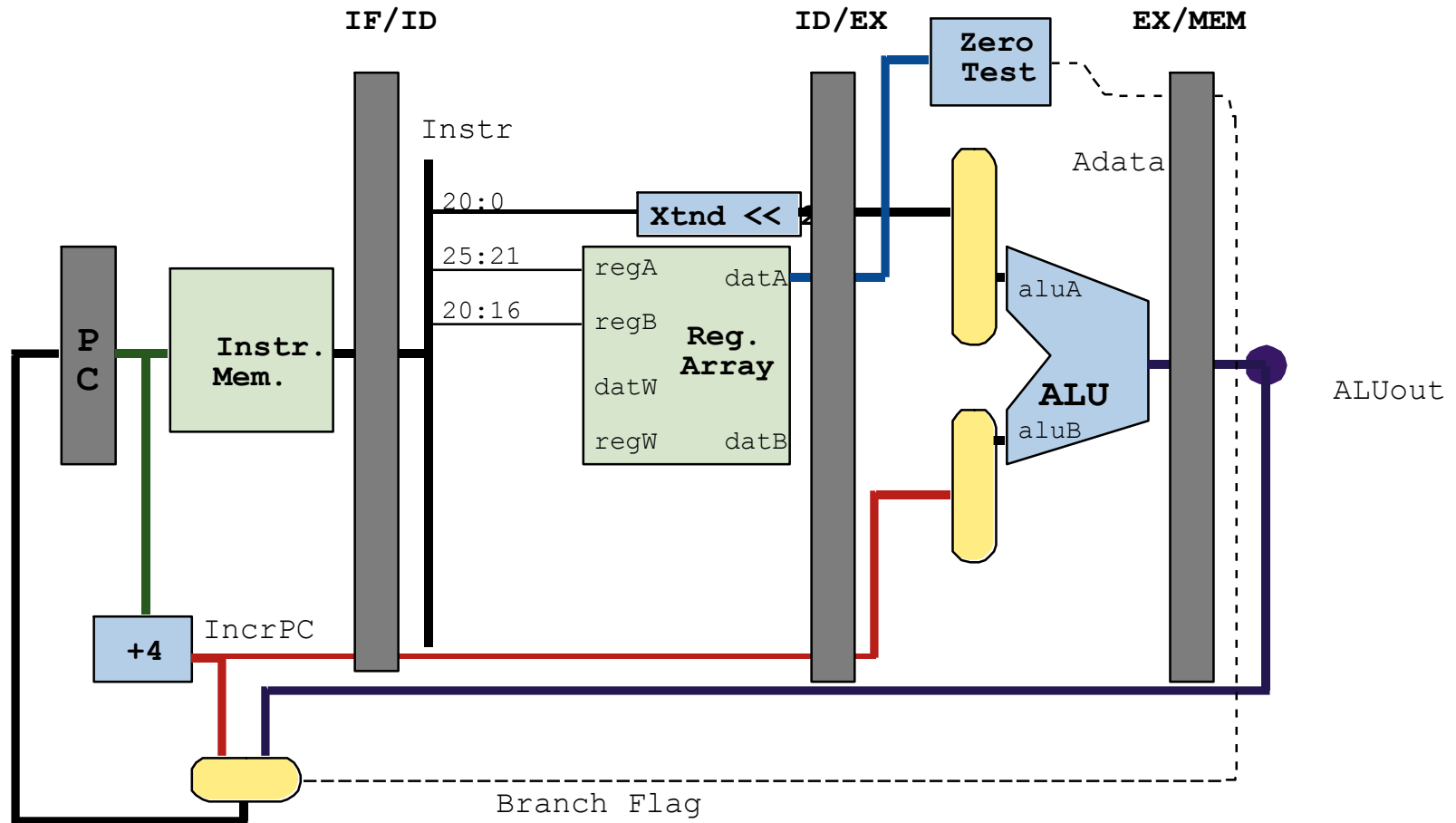
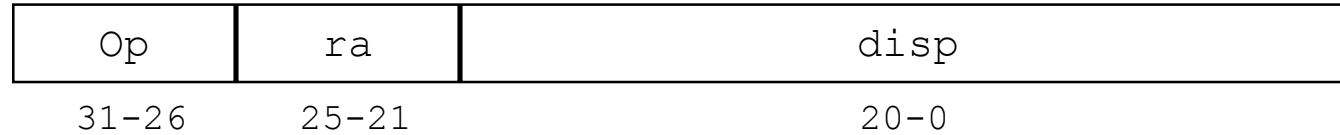
Pipelined datapath



What happens with a branch?

Conditional Branch Instruction Handling

beq: $PC \leftarrow Ra == 0 ? PC + 4 + disp*4 : PC + 4$

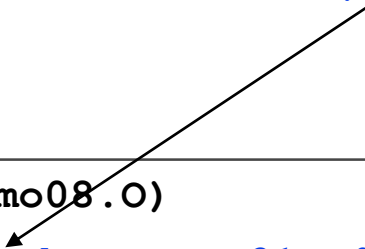


Branch Example

Desired Behavior

- Take branch at 0x00
- Execute target 0x18
 - $PC + 4 + \text{disp} \ll 2$
 - $PC = 0x00$
 - $\text{disp} = 5$

Displacement



Branch Code (demo08.0)

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

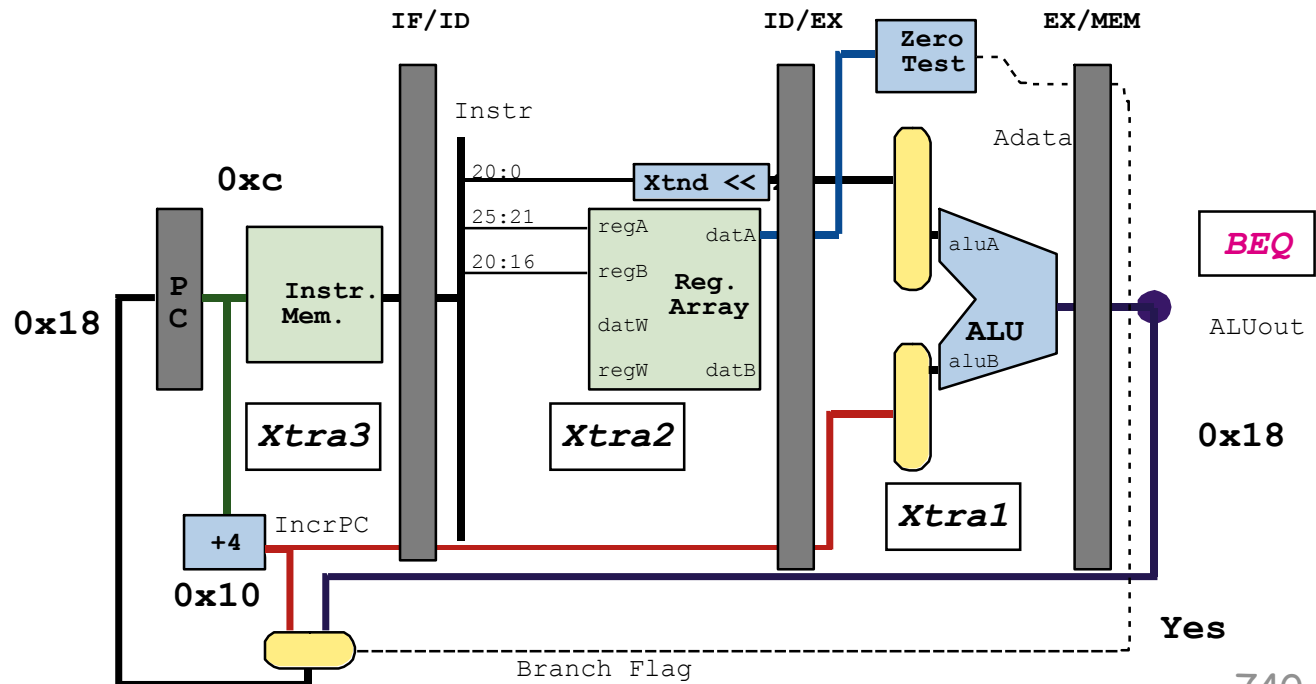
Branch Hazard Example

```

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



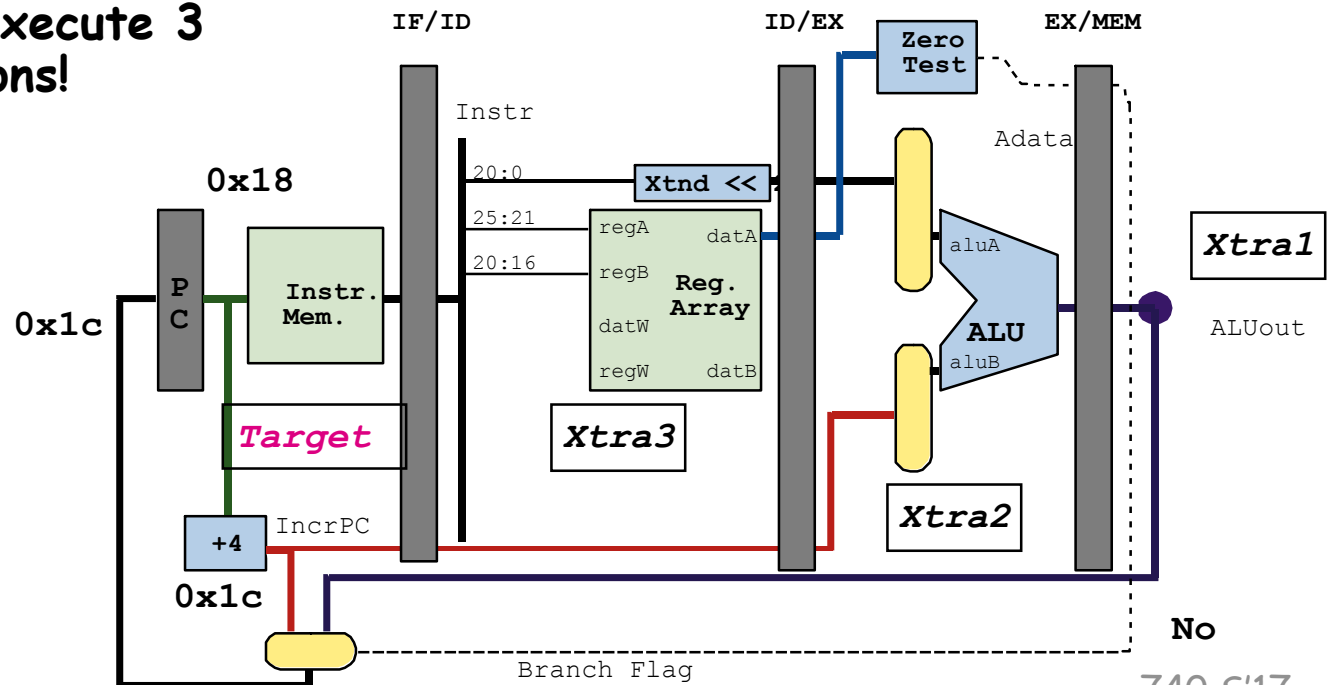
Branch Hazard Example (cont.)

```

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
    
```

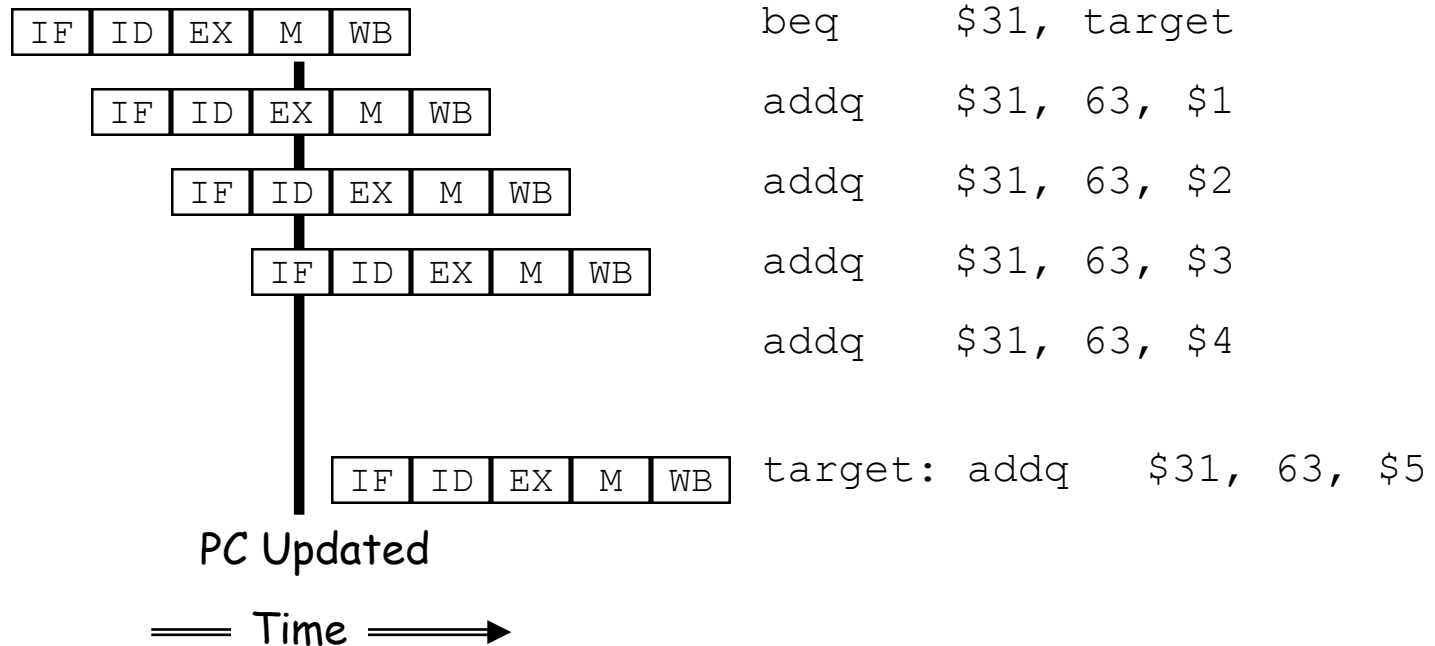
- One cycle later
- Problem: Will execute 3 extra instructions!



Branch Hazard Pipeline Diagram

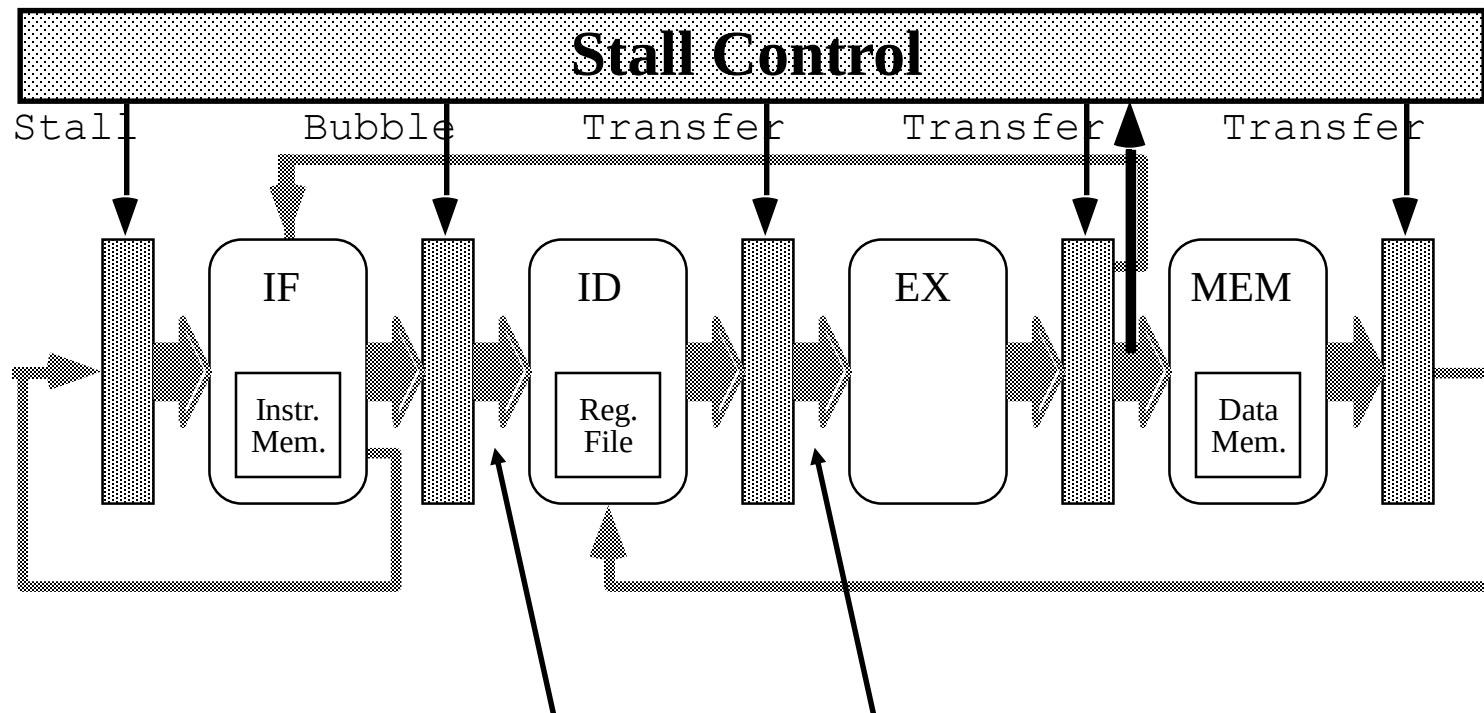
Problem

- Instruction fetched in IF, branch condition set in MEM



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

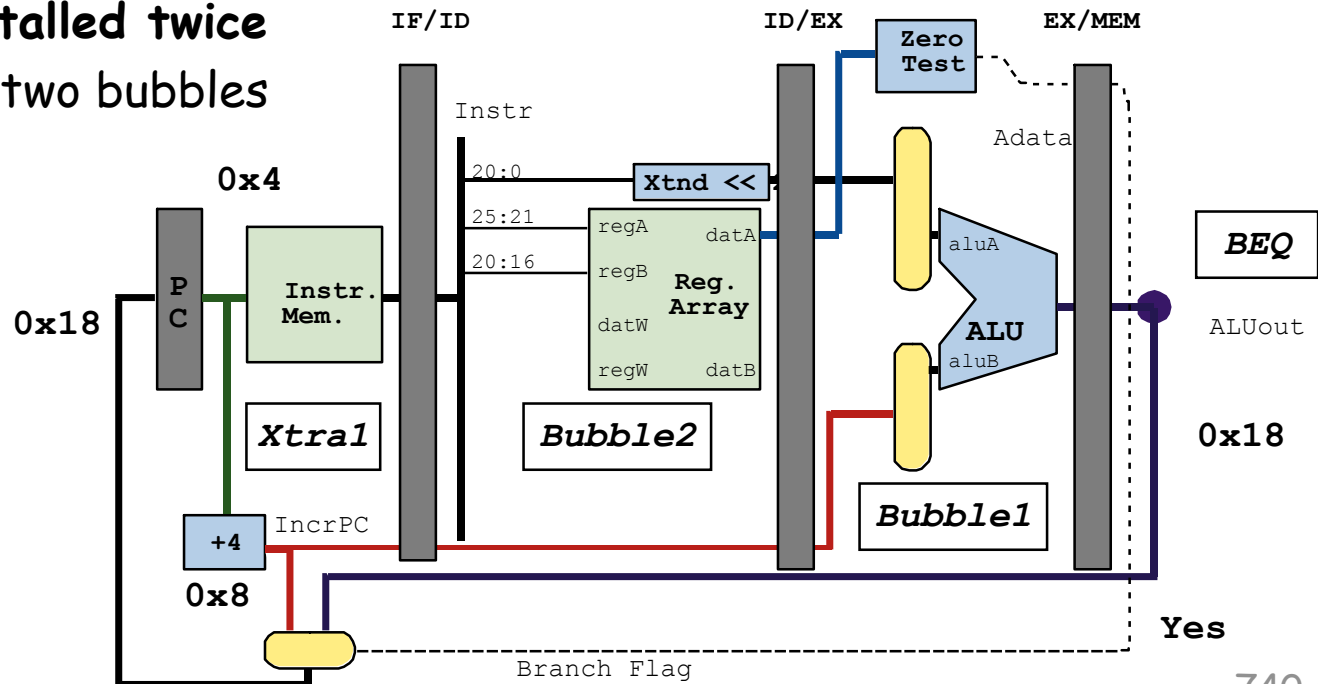
Stalling Branch Example

```

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

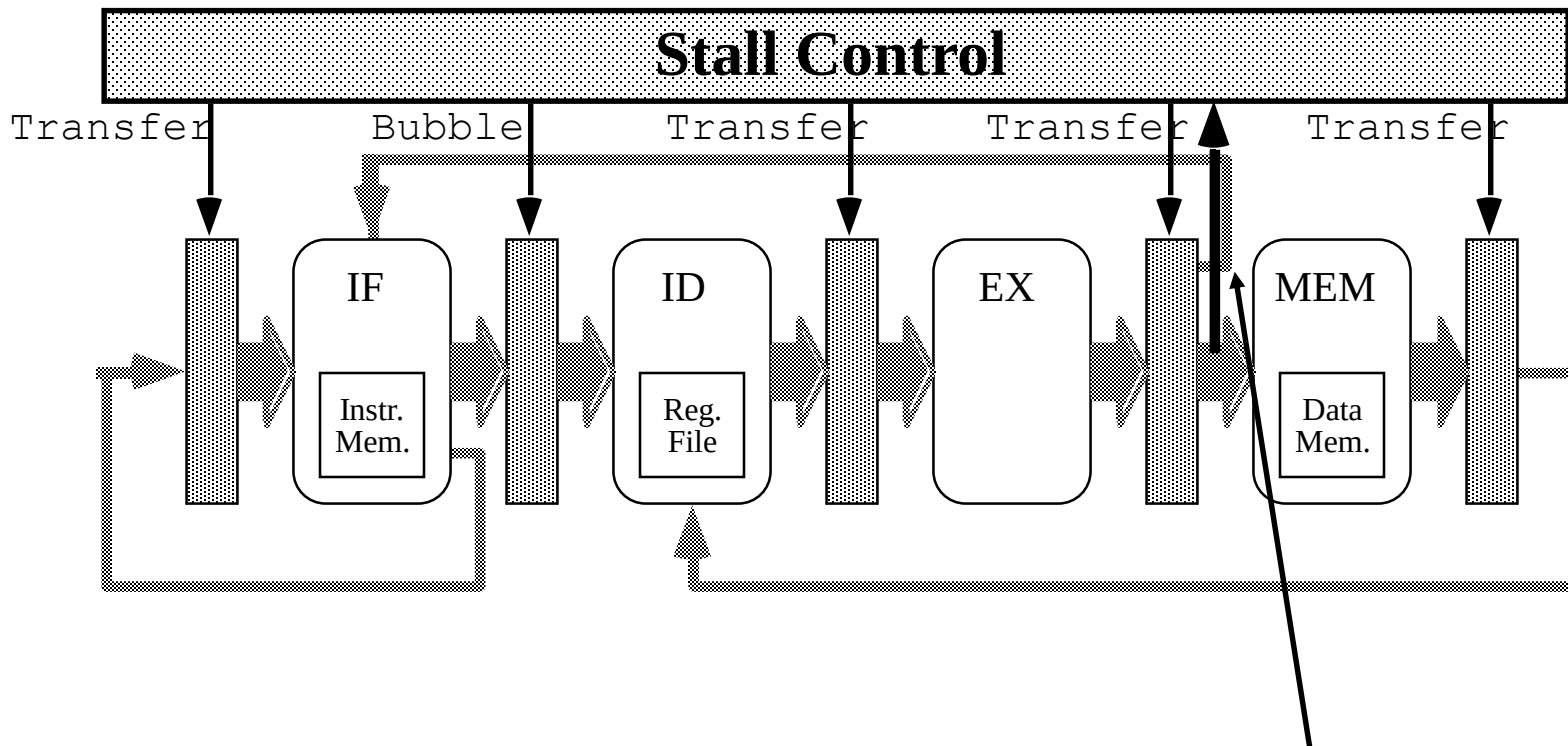
0x18: addq     r31, 0x3f, r5   # Target
    
```

- 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

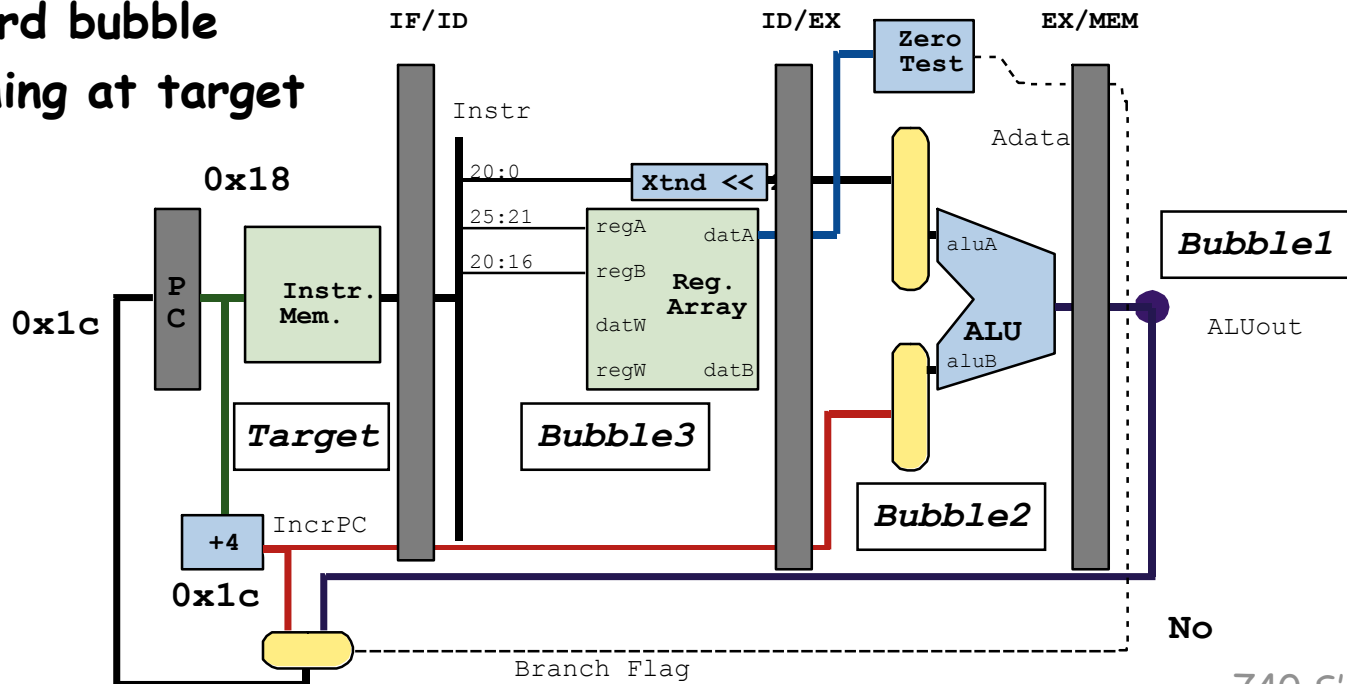
Taken Branch Resolution Example

```

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
    
```

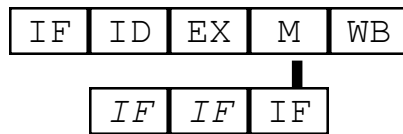
- When branch taken
- Generate 3rd bubble
- Begin fetching at target



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

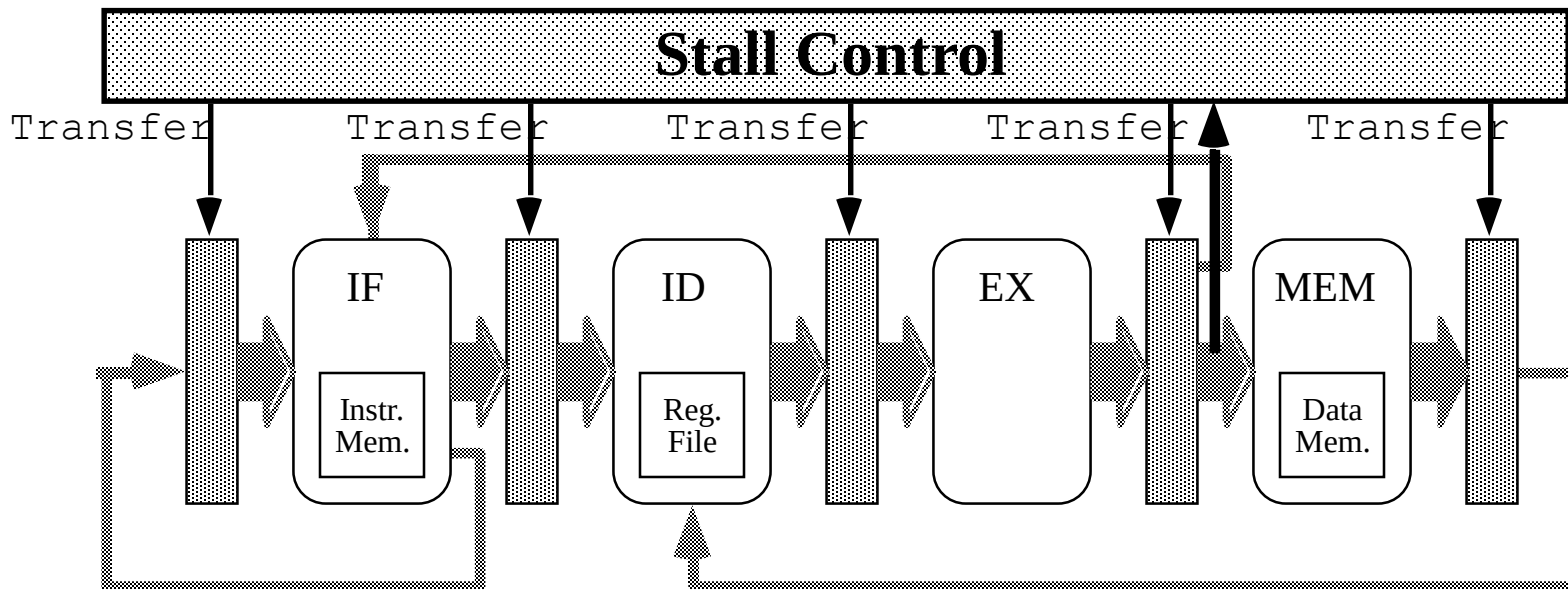
target: addq \$31, 63, \$5 # Target

PC Updated

Time →

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

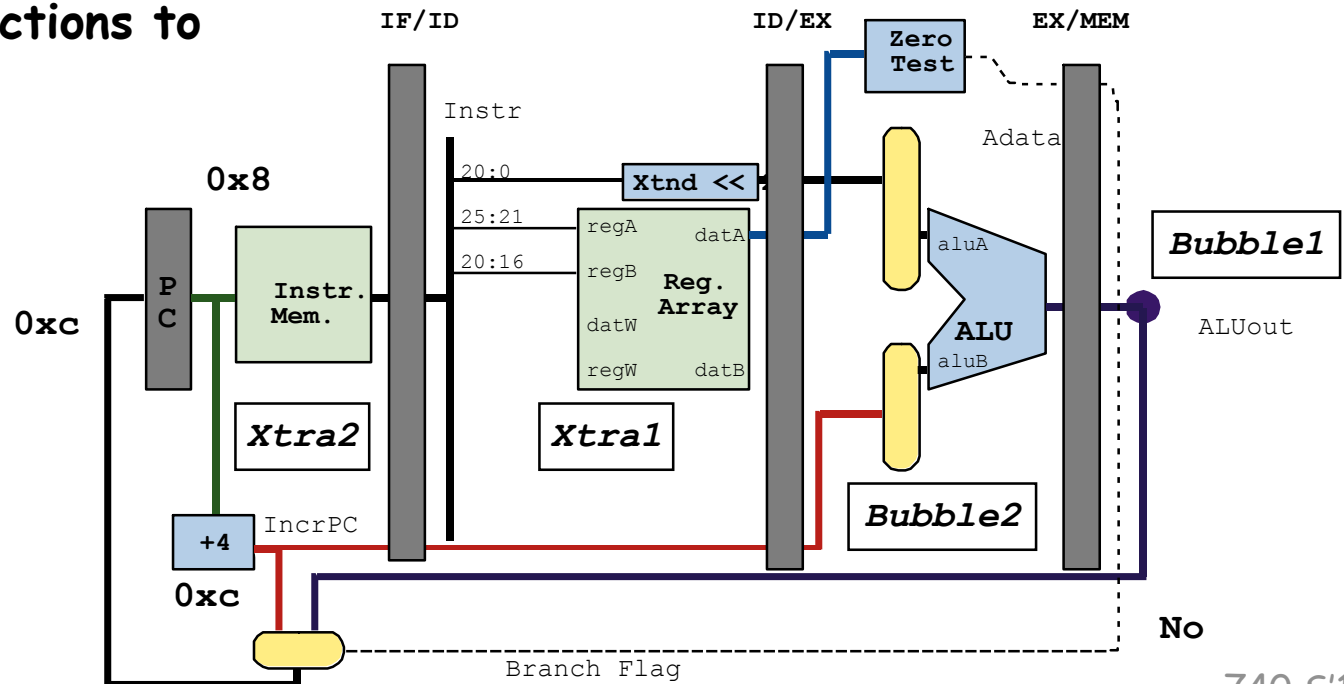


Not Taken Branch Resolution Example

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

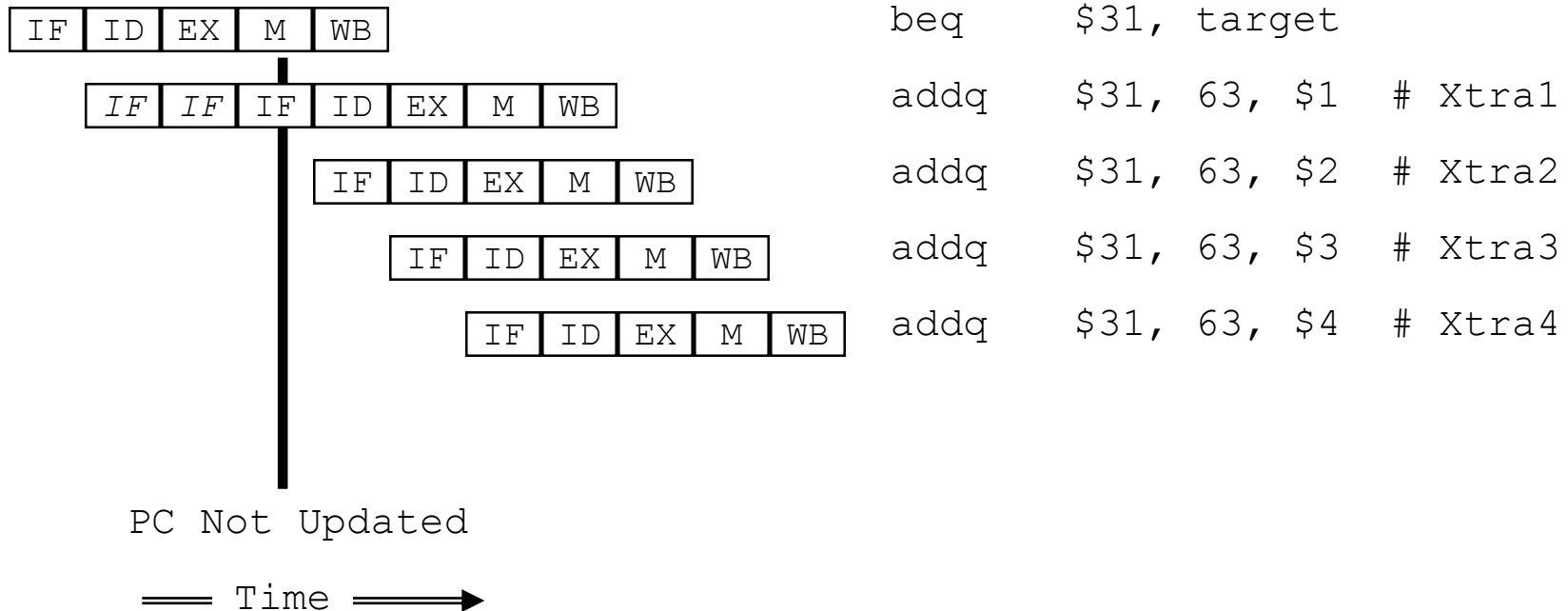
- Branch not taken
- Allow instructions to proceed



Not Taken Branch Pipeline Diagram

Behavior

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



Analysis of Stalling

Branch Instruction Timing

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

Performance Impact

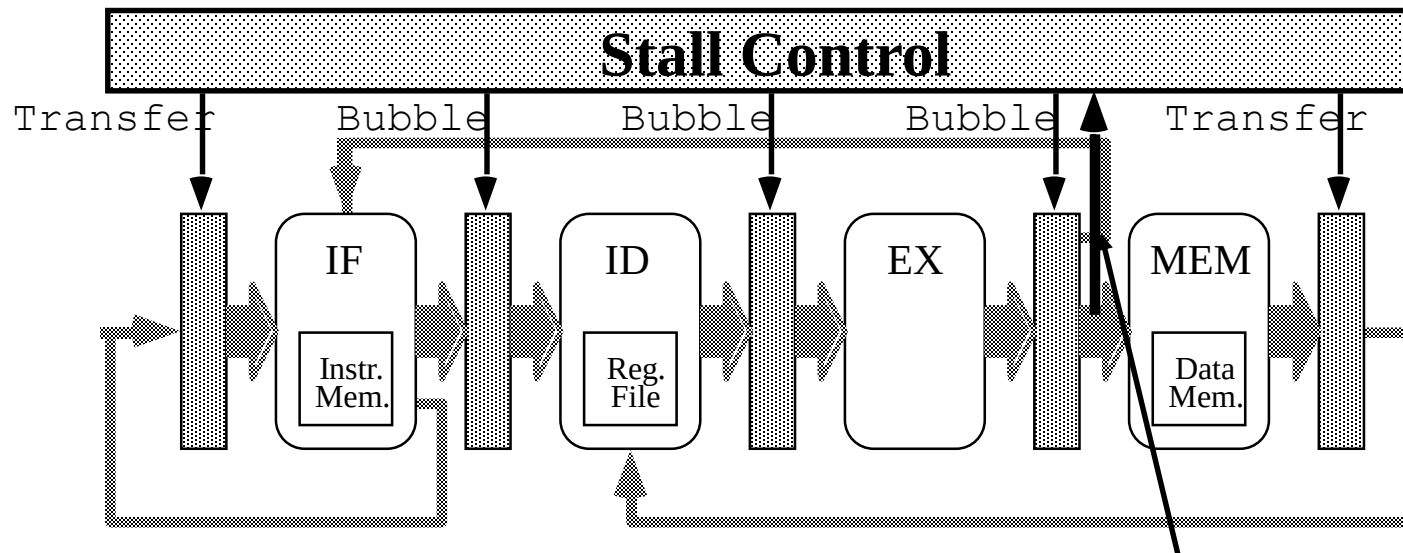
- Branches $\approx 20\%$ of instructions
- $\approx 67\%$ branches are taken
- Adds $0.2 * (0.67 * 3 + 0.33 * 2) = 0.54$ increase to CPI
 - Serious performance impact!

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

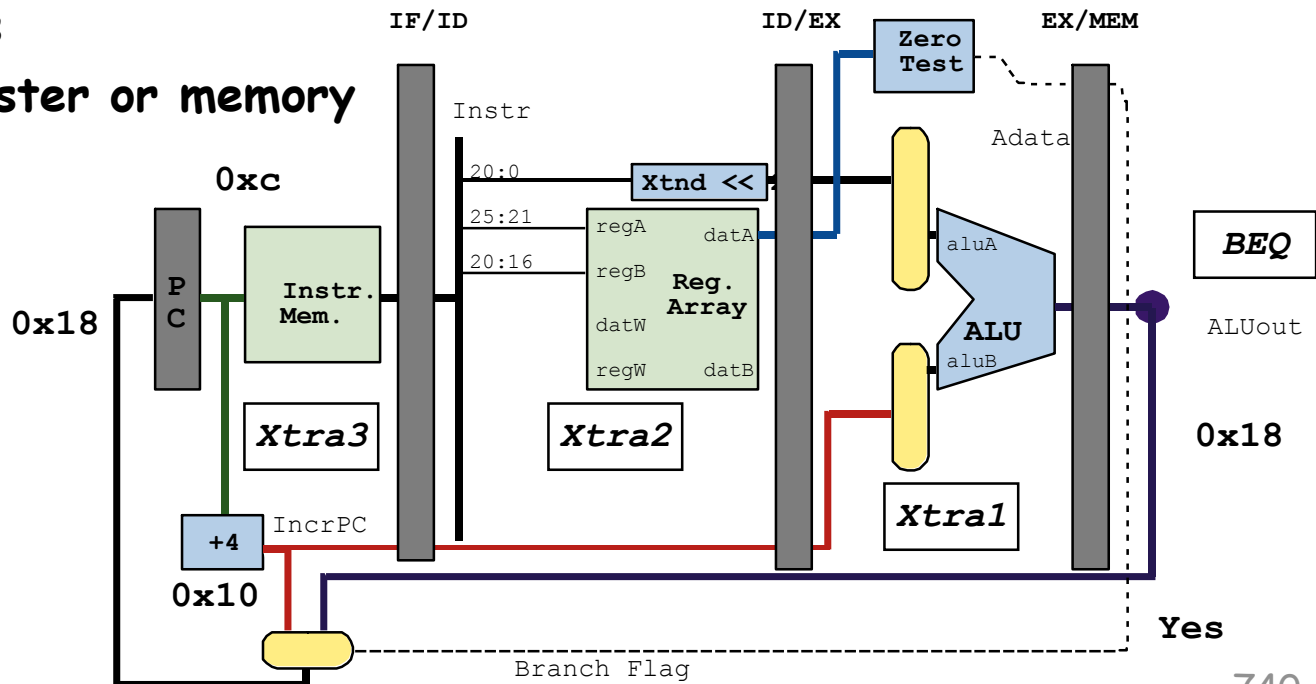
```

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



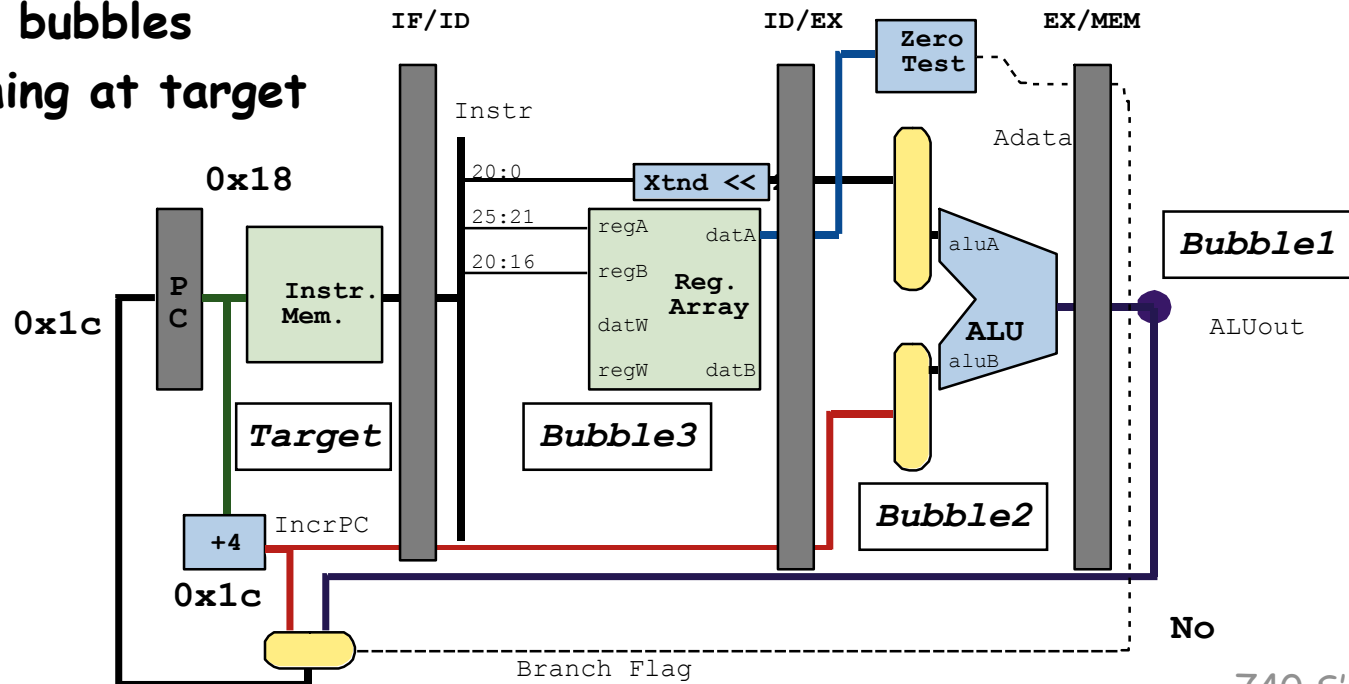
Canceling Branch Resolution Example

```

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
    
```

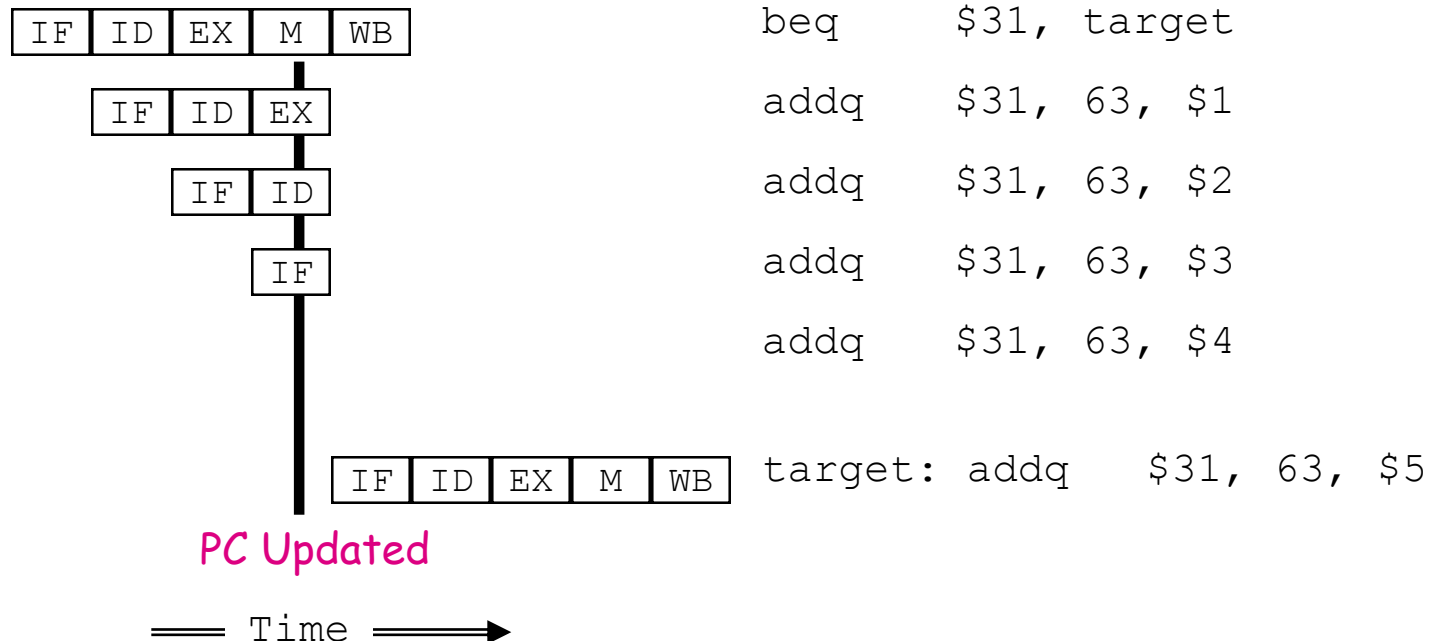
- When branch taken
- Generate 3 bubbles
- Begin fetching at target



Canceling Branch Pipeline Diagram

Operation

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



Noncanceling Branch Pipeline Diagram

Operation

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



Branch Prediction Analysis

Our Scheme Implements “Predict Not Taken”

- But 67% of branches are taken
- Impact on CPI: $0.2 * 0.67 * 3.0 = 0.4$
 - Still not very good

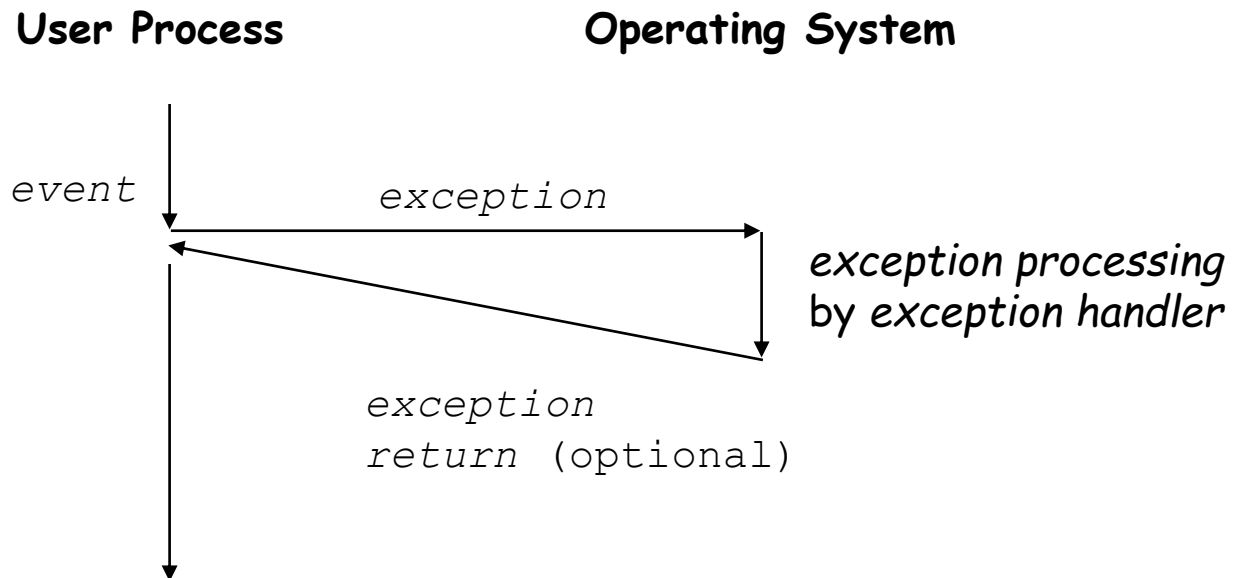
Alternative Schemes

- Predict taken
 - Would be hard to squeeze into our pipeline
 - » Can't compute target until ID → one bubble
 - » MIPS branch delay slot exposes this bubble in ISA
- Backwards taken, forwards not taken
 - Predict based on sign of displacement
 - Exploits fact that loops usually closed with backward branches
- Branch target buffer (BTB) speculates on branch destination in IF
 - What's done in practice

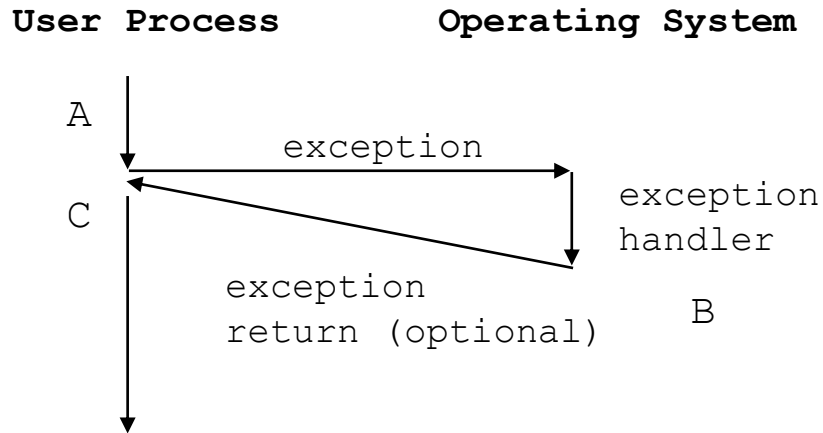
EXCEPTIONS AND MULTI-CYCLE INSTRUCTIONS

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 ctrl-alt-delete on a PC

Exception handling (hardware tasks)

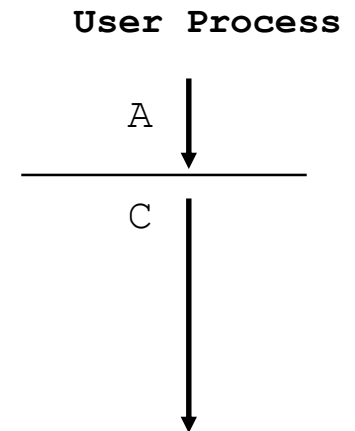
Recognize event(s)

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*

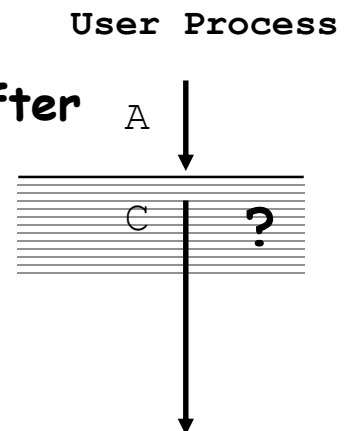
Imprecise exceptions:

- 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?

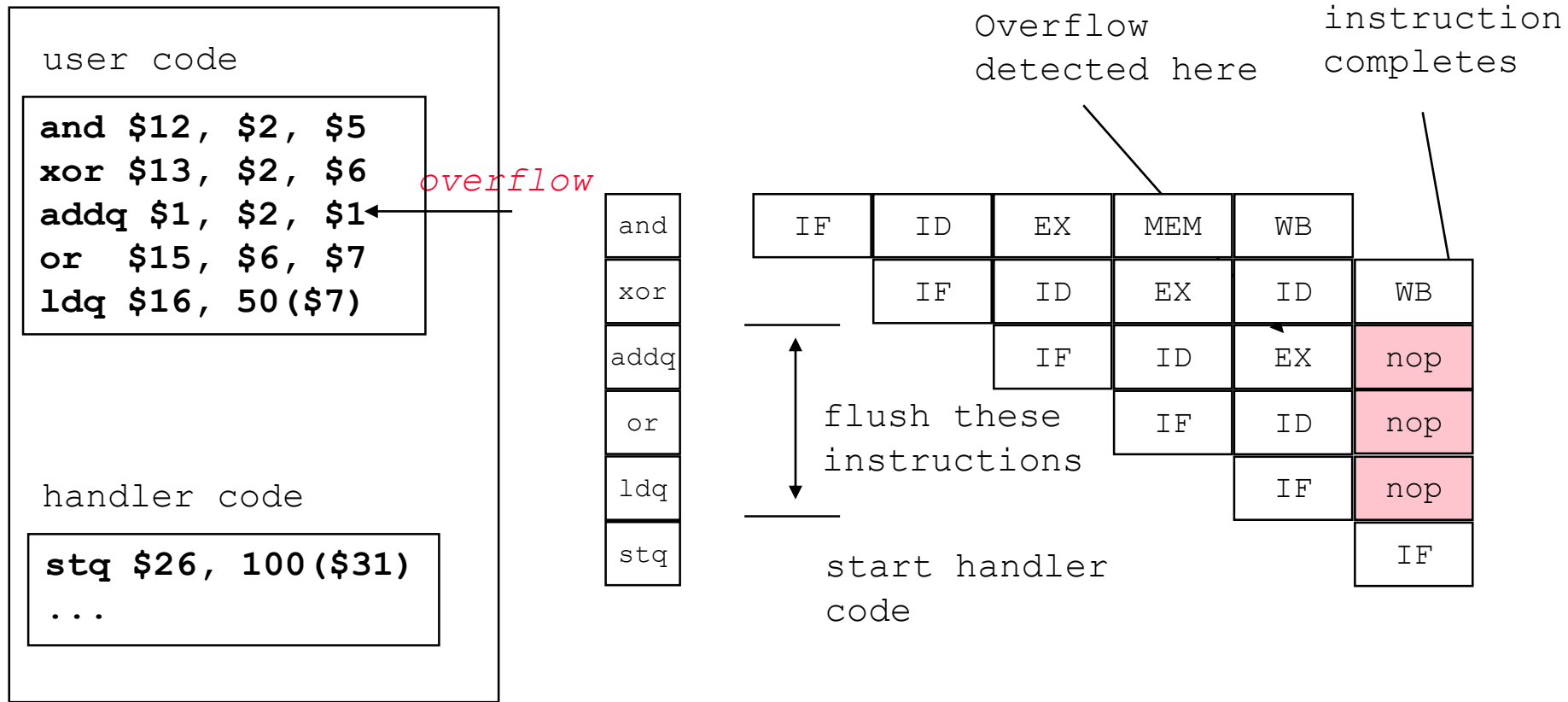
- insert a TRAPB (trap barrier) instruction immediately after
 - stalls until certain that no earlier insts take exceptions

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.)



Multicycle instructions

Alpha 21264 Execution Times:

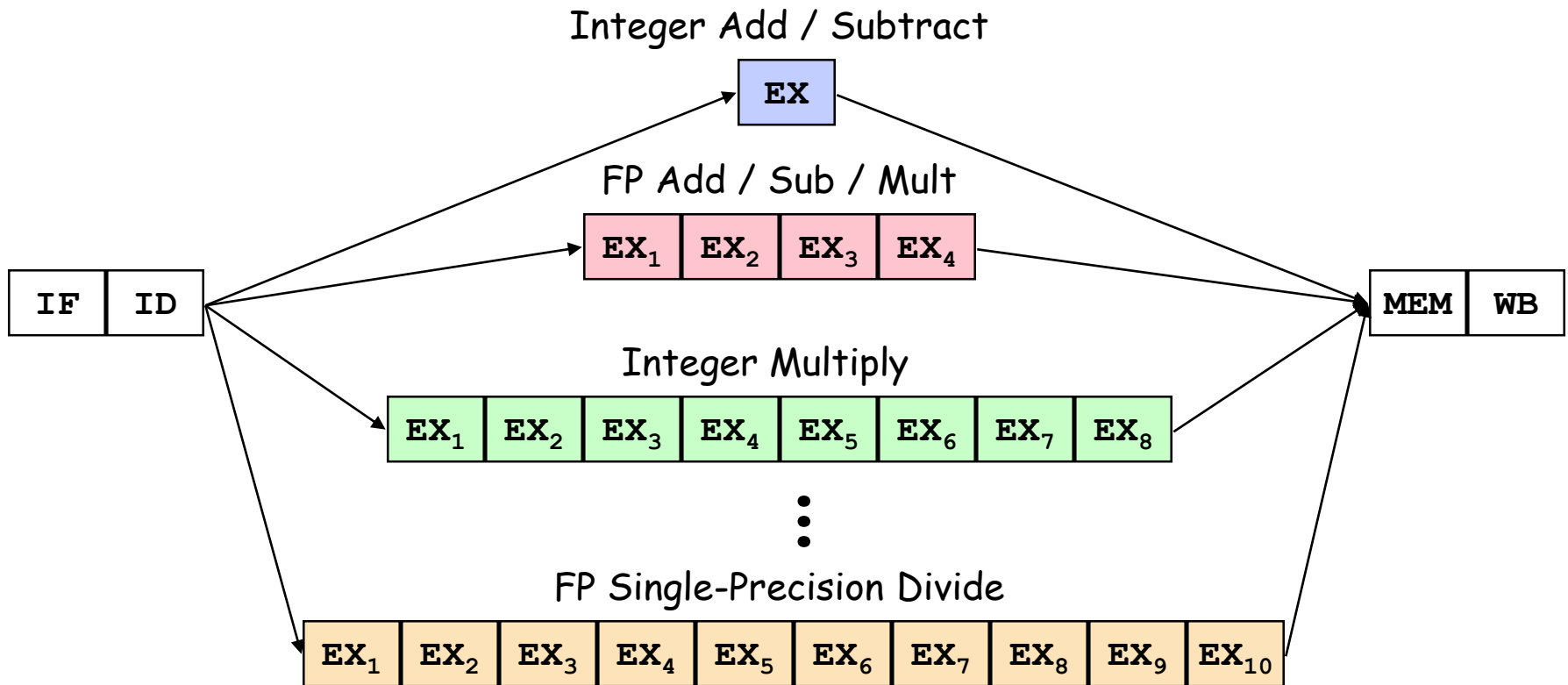
- Measured in clock cycles

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:

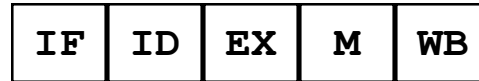
Operation	Integer	FP Benchmarks	
	Benchmarks	Integer	FP
add / sub	14%	11%	14%
multiply	< 0.1%	< 0.1%	13%
divide	< 0.1%	< 0.1%	1%

Pipeline Revisited

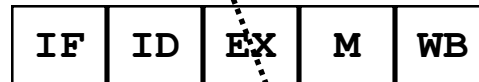


Multiply Timing Example

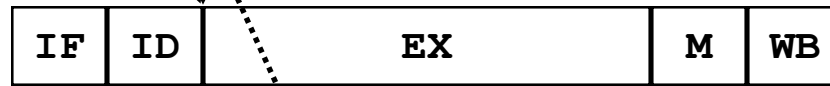
bis \$31, 3, \$2



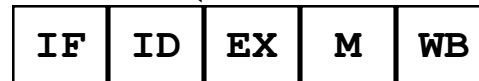
bis \$31, 7, \$3



mulq \$2, \$3, \$4



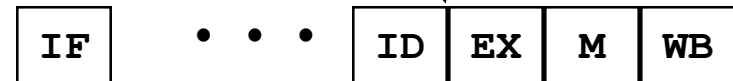
addq \$2, \$3, \$3



bis \$4, \$31, \$5



addq \$2, \$4, \$2



Stall while Busy

(Not to scale)

Conclusion

Pipeline Characteristics for Multi-cycle Instructions

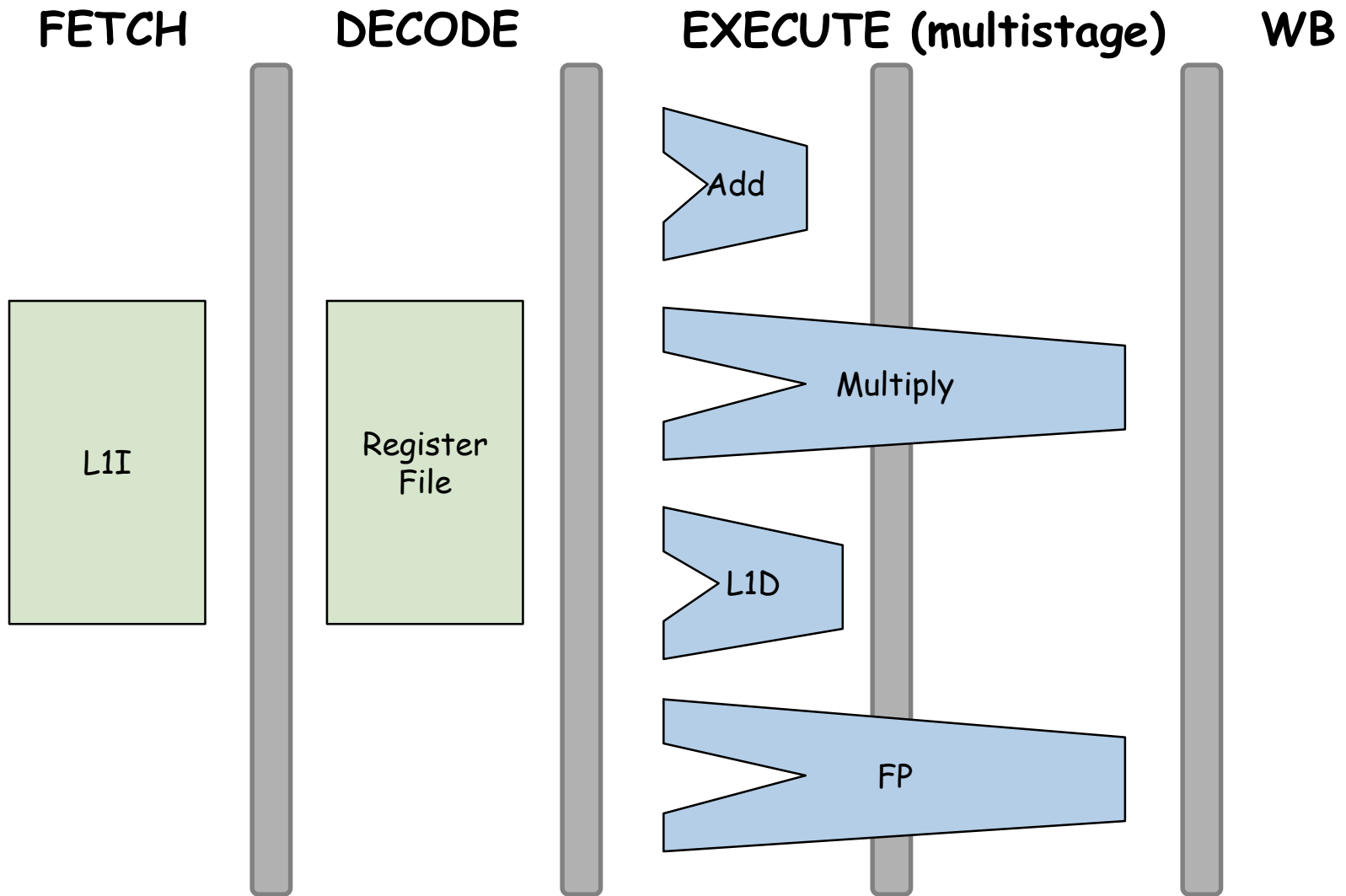
- **In-order issue**
 - Instructions fetched and decoded in program order
- **Out-of-order completion**
 - Slow instructions may complete after ones that are later in program order
 - Reason for imprecise exceptions ... but difficult to reason about

Performance Opportunities

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

SUPERSCALAR

Pipelines Thus Far



Increasing Performance

Execution time = instructions / program x
cycles / instruction x
seconds / cycle

- Increase clock frequency
- Decrease CPI
- How well does pipelining do?

Pipelining performance

Execution time = instructions / program ×
cycles / instruction ×
seconds / cycle

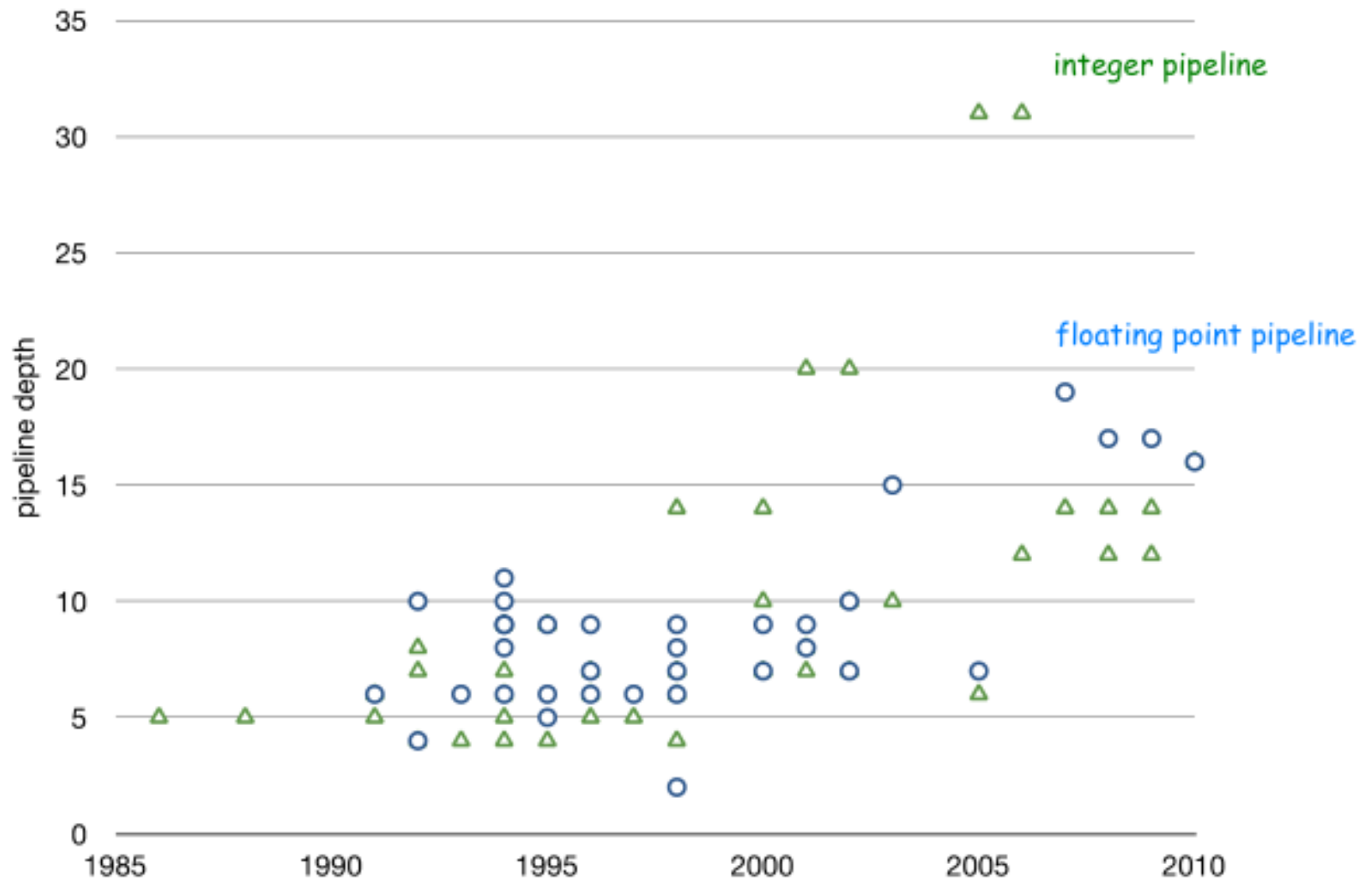
- Increase clock frequency
 - N-stage can give $\approx N\times$ faster clock
- Decrease CPI
 - Pipelining increases CPI due to hazards & stalls (but less than $N\times$)

Limitations of pipelining

Stages can't be increased forever

- Pipeline overheads become significant
- Bypassing more expensive, less effective
- Flushes due to mispredicted branches

Pipeline Depth



Going beyond pipelining

Pipeline processors limited by $CPI \leq 1$
(the “Flynn bottleneck”)

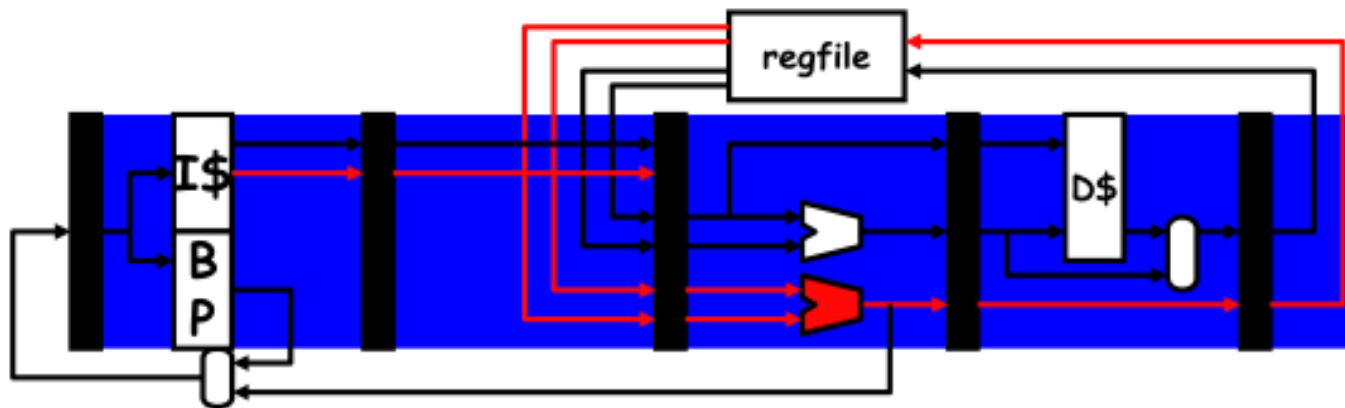
But >1 functional unit in execute stage

Why not issue multiple instructions per cycle?

→ *Superscalar processors*

- Instruction-level parallelism (ILP)
- Today processors typically 4-wide issue

A Typical Dual-Issue Pipeline (1 of 2)



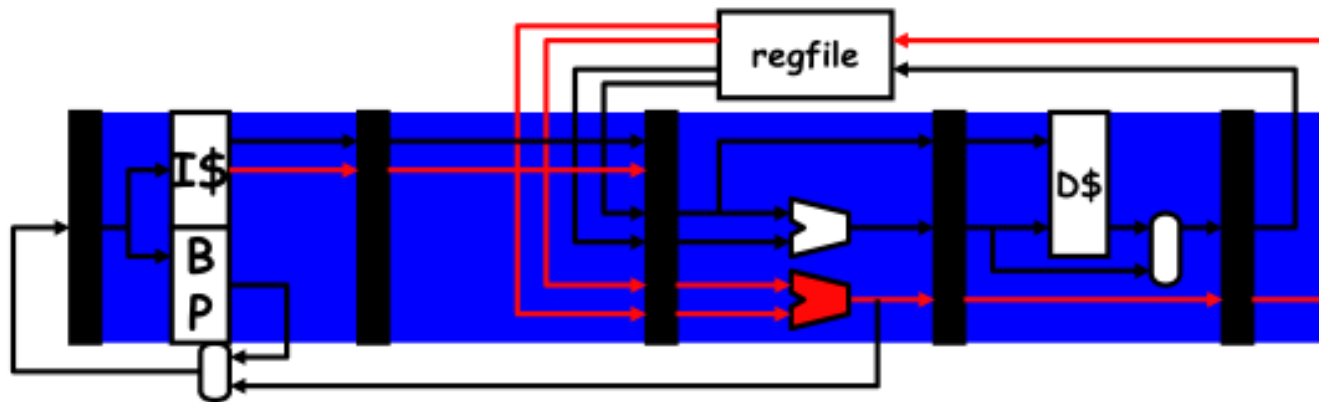
Fetch an entire 16B or 32B cache block

- 4 to 8 instructions (assuming 4-byte average instruction length)
- Predict a single branch per cycle

Parallel decode

- Need to check for conflicting instructions
 - Is output register of I_1 is an input register to I_2 ?
- Other stalls, too (for example, load-use delay)

A Typical Dual-Issue Pipeline (2 of 2)



Multi-ported register file

- Larger area, latency, power, cost, complexity

Multiple execution units

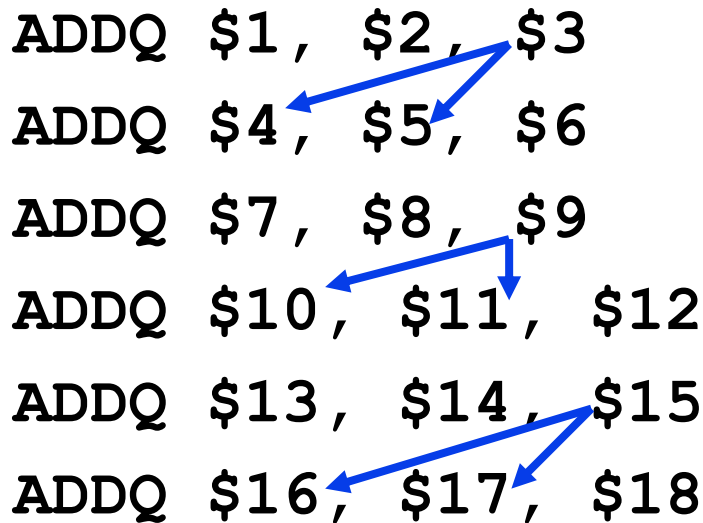
- Simple adders are easy, but bypass paths are expensive

Memory unit

- Single load per cycle (stall at decode) probably okay for dual issue
- Alternative: add a read port to data cache
 - Larger area, latency, power, cost, complexity

Superscalar Example

ADDQ \$1, \$2, \$3
ADDQ \$4, \$5, \$6
ADDQ \$7, \$8, \$9
ADDQ \$10, \$11, \$12
ADDQ \$13, \$14, \$15
ADDQ \$16, \$17, \$18

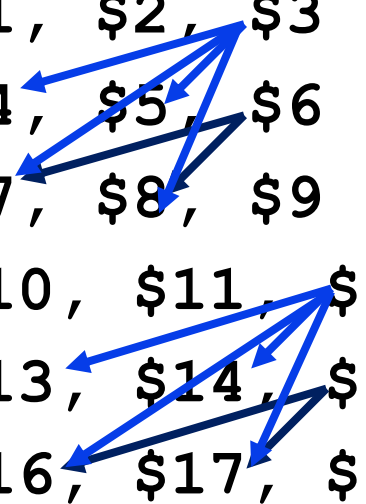


What checks are required for **2**-wide issue?

What does the execution look like? How many cycles?

Superscalar Example

ADDQ \$1, \$2, \$3
ADDQ \$4, \$5, \$6
ADDQ \$7, \$8, \$9
ADDQ \$10, \$11, \$12
ADDQ \$13, \$14, \$15
ADDQ \$16, \$17, \$18



What checks are required for **3**-wide issue?

What does the execution look like? How many cycles?

Superscalar Example

ADDQ \$1, \$2, \$3
ADDQ \$4, \$5, \$6
ADDQ \$7, \$8, \$9
ADDQ \$10, \$11, \$12
ADDQ \$13, \$14, \$15
ADDQ \$16, \$17, \$18



What checks are required for **4**-wide issue?

What does the execution look like? How many cycles?

Superscalar Example

ADDQ \$1, \$2, \$3
ADDQ \$4, \$5, \$6
ADDQ \$7, \$3, \$9
ADDQ \$9, \$11, \$12
ADDQ \$13, \$8, \$15
ADDQ \$16, \$17, \$18



What checks are required for **3**-wide issue?

What does the execution look like? How many cycles?

Superscalar Implementation - F&D

Fetch

- Modest: Just fetch multiple instructions per cycle
- Aggressive: Buffer instructions / predict multiple branches

Decode

- Replicate decoders for each instruction

Superscalar Implementation - Issue

Instruction issue

- Determine which instructions can execute
- $O(N^2)$ checks required for N-wide machine
- Other limitations based on execution units

Register read

- Add read & write ports to register file
- Affects latency & area roughly $O(ports^2)$

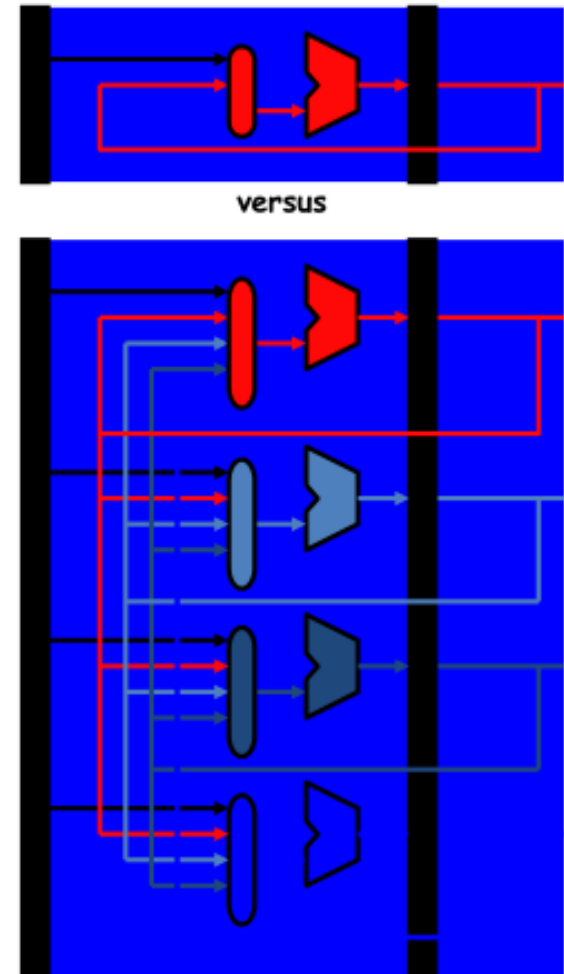
Superscalar Implementation - EX

Replicate functional units?

- Yes for simpler ones like adders
- No for expensive, rarely-used like divide
- Somewhat for expensive, often-used like cache ports

Bypass paths

- $O(PN^2)$ paths required for full bypass (P - pipeline depth)
- N-way muxes at each stage add latency to critical path
- Can add pipeline stages for bypassing but this isn't free



Superscalar Challenges

Quadratic scaling factors

- Dependence checks
- Register file size
- Bypass paths

Speedup limited by ILP

- Rely on compiler
- Still face heavily diminishing returns

➔ Superscalar is a good idea, but limited scaling

Not all N^2 are created equal

Stall logic vs bypass network vs register file - which is the bigger problem?

Bypass network by far

- 64-bit values vs 5-bit register names
- Bypass between stages $O(N^2)$ vs $O(PN^2)$
- Must fit within clock + ALU

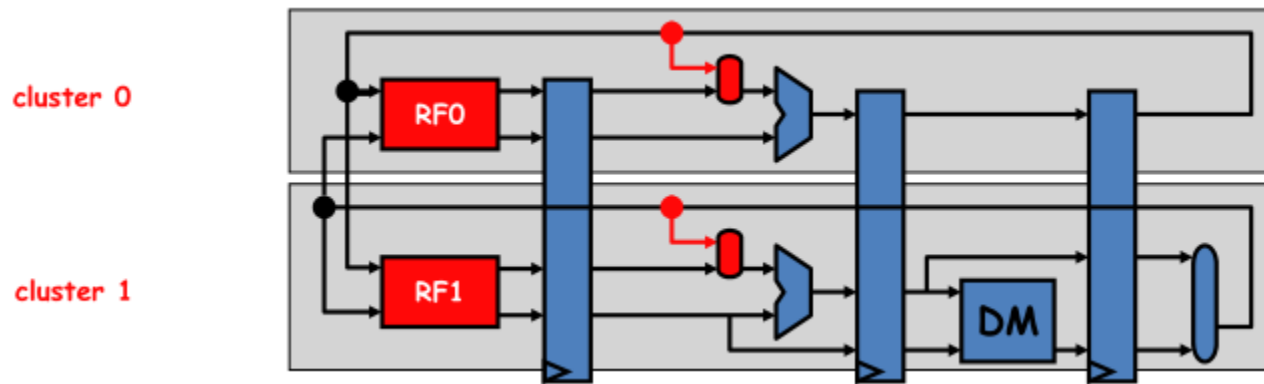
Register file also expensive

Dependency checks are a distant 3rd

Idea: Clustering

Stall logic does full $O(N^2)$ dependence checks

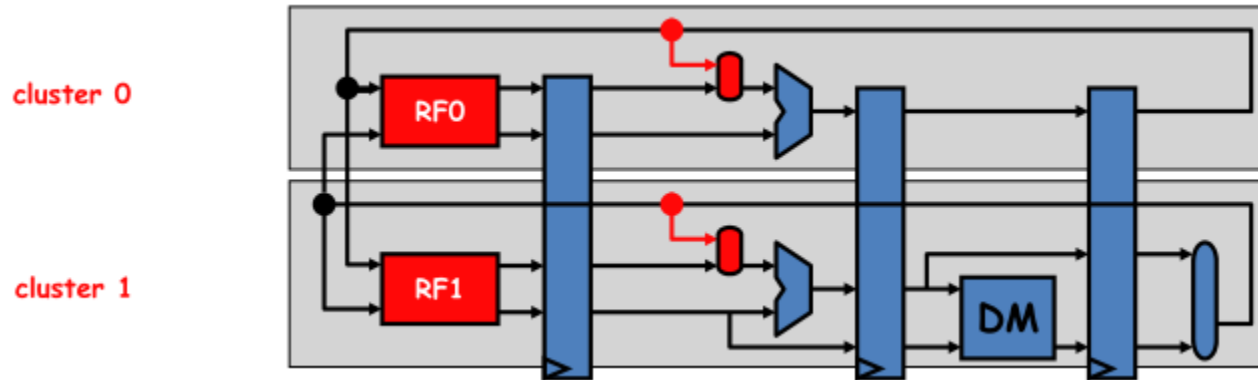
- No real choice, must execute correctly



Cluster execution units and register file

- Full bypass within a cluster (with smaller N)
- Limited bypassing between clusters - takes 1 or 2 cycles

Idea: Clustering



Cluster execution units and register file

Dependent instructions **steered** towards appropriate cluster

Register file **banked** (split) across clusters

- Or replicated – fewer read ports, multiple writes

Other challenges: Superscalar Fetch

What does it mean to fetch multiple insns per cycle?

Same cache block → no problem

If last instruction in block → single issue this cycle?

What about taken branches?

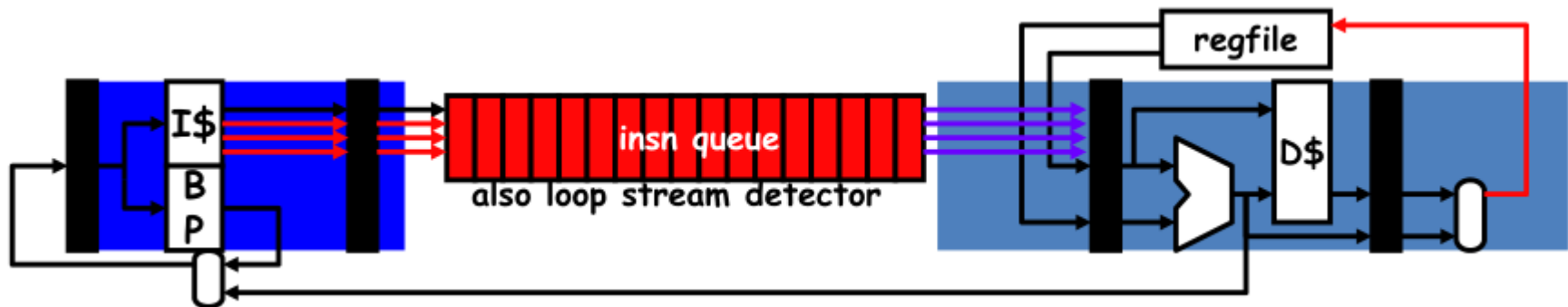
- 20% branches x 50% taken → ~10 instructions between taken branches

Other challenges: Superscalar Fetch

What is the ILP of this program on a 4-wide issue?

```
START:    ADDQ $1, $1, 1
          ADDQ $2, $2, 1
          ADDQ $3, $3, 1
          ADDQ $4, $4, 1
          BEZ $1, START      # assume taken
```

Other challenges: Superscalar Fetch



Over-fetch and buffer

- Add a queue between fetch and decode (18 entries on Core 2)
- Compensates for cycles that fetch less than maximum issue
- Decouples front-end and back-end (see also Decoupled Access Execute [Smith,'82] for a different application of same idea)

Or put entire loops in icache

- Any mispredicted branch falls back to normal fetch
- Macro-ops (eg Core 2) vs micro-ops (Core i7) vs trace cache (P4)

Other challenges: Superscalar Commit

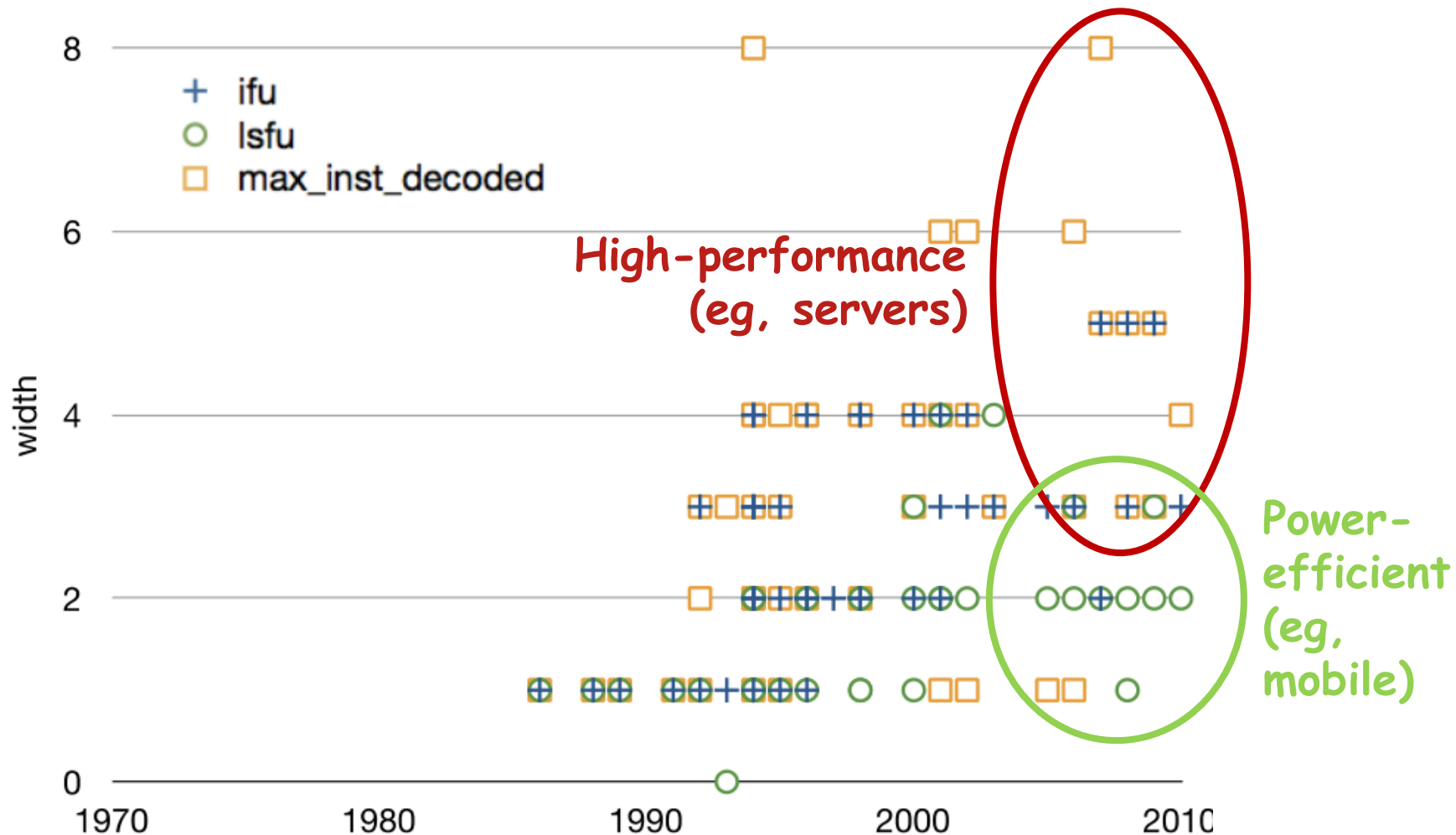
Which instructions write registers in a 4-wide issue?

ADDQ	\$1,	\$2,	\$3	Destinations
ADDQ	\$2,	\$6,	\$1	
ADDQ	\$5,	\$8,	\$3	
ADDQ	\$4,	\$3,	\$1	

Doesn't execute

Need to add dependence checks in writeback

Trends in Superscalar Width



Conclusion: Superscalar

Multiple issue

- Exploits ILP beyond pipelining
- Improves IPC at the cost of clock & energy & area
- 4- to 6-wide issue is about peak justifiable width

Problem spots

- Bypass & register file scale $O(N^2)$
 - Clustering one solution
- Fetch and decode complicated
 - Buffering, loop streaming, trace cache
- Dependence checks also scales $O(N^2)$
 - VLIW tries to fix this (next time)