

Intro to Microarchitecture: Basic Pipelining

15-740 FALL'21

NATHAN BECKMANN

Objective

Design Processor for Alpha Subset

- Interesting but not overwhelming quantity
- Using 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): $rc \leftarrow ra \text{ func } rb$

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

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

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

Encoding

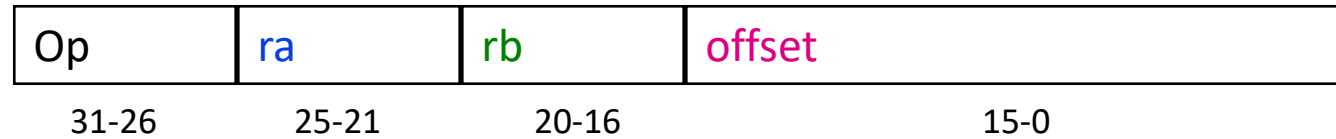
- **ib** is 8-bit unsigned literal

Operation	Op field	func field
addq	0x10	0x20
subq	0x10	0x29
or/bis	0x11	0x20
xor	0x11	0x40
cmoveq	0x11	0x24 (conditional move; $rc = (ra == 0) ? rb : rc$)
cmplt	0x11	0x4D (compare less-than)

ALPHA Load/Store Instructions

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

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



Encoding

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

Operation

ldq

stq

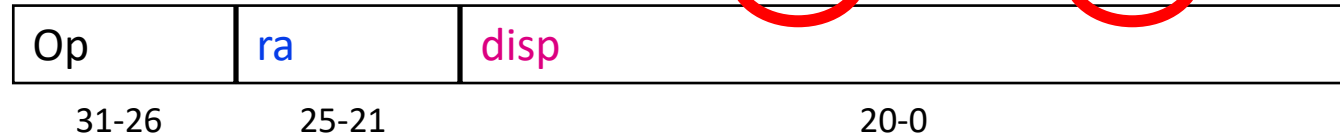
Op field

0x29

0x2D

ALPHA Branch Instructions

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



Encoding

- **disp** is 21-bit signed displacement

Operation

Op field

Cond

beq

0x39

Ra == 0

bne

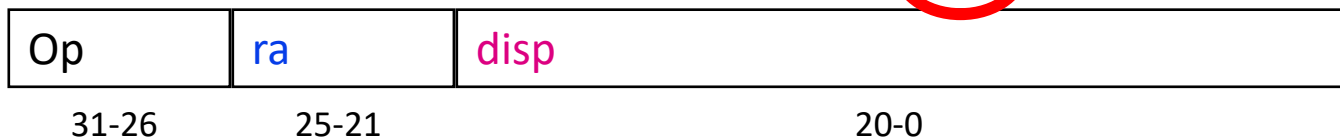
0x3D

Ra != 0

Why PC + 4?

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

Branch [Subroutine]: $Ra \leftarrow PC + 4; PC \leftarrow PC + 4 + \text{disp} * 4$



Operation

Op field

Convention: Ra = 26

br

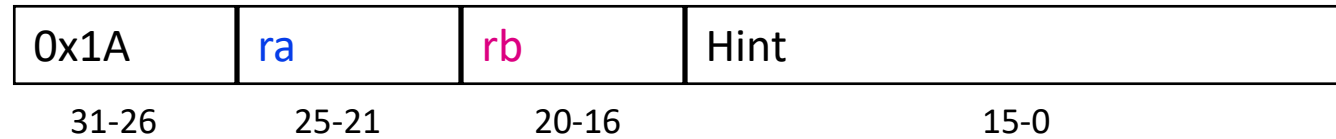
0x30

bsr

0x34

ALPHA Control Transfers

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



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

ALPHA 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			

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

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			

Hardware Design Basics

Control vs. datapath

Datapath: “Block diagram of a computer processor, excluding control signals”

- Large components
 - Logic blocks (“combinational”)
 - Memories (“sequential”)
- Components connected by wires
- Multiplexors (select 1 of n inputs) determine functionality

Control

- Generated from state machine / input (e.g., a program’s instructions)
- Generates signals to control datapath (e.g., mux “sel” input)
- Often not shown in diagrams

Hardware mental model

Combinational logic: Can perform arbitrary computation

- Logic built up from simple “gates” – AND, OR, NOT, etc
 - E.g., can make a 1-bit adder from truth table of addition + logic operations, then combine 1-bit adders to make an n-bit adder
- Essentially a pure functional (side-effect-free) computation model

Sequential logic: Stateless logic connected to memories

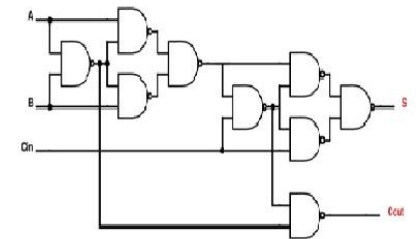
- Memories are also implemented in gates

Everything is parallel, all of the time!

Q: “Where does that algorithm run?”

A: It runs in hardware ... Architects can add the logic wherever they want to run whatever they want (up to area/perf/power limits)

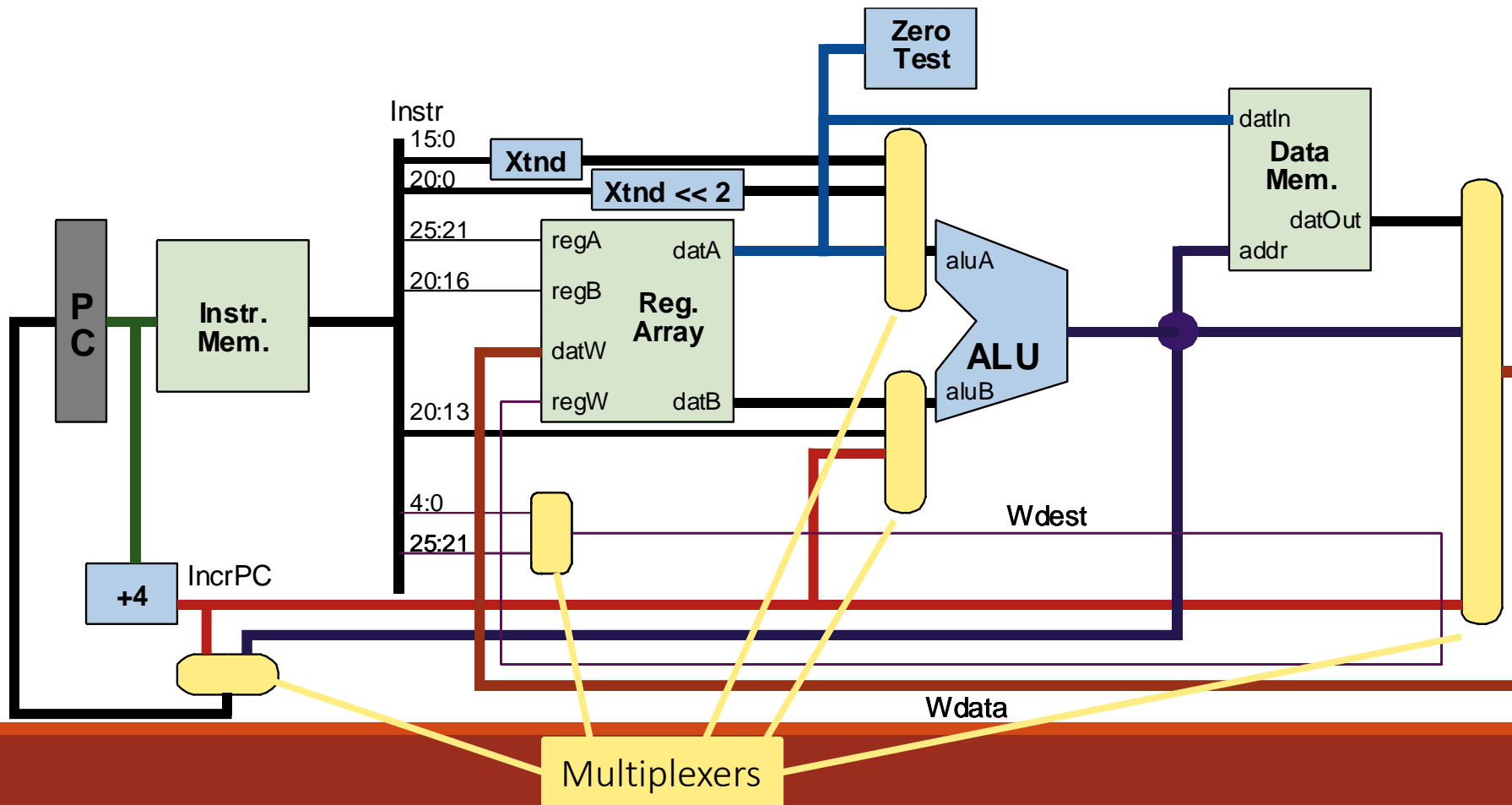
Inputs			Outputs	
A	B	C _{in}	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Single-Cycle ALPHA Implementation

Datapath

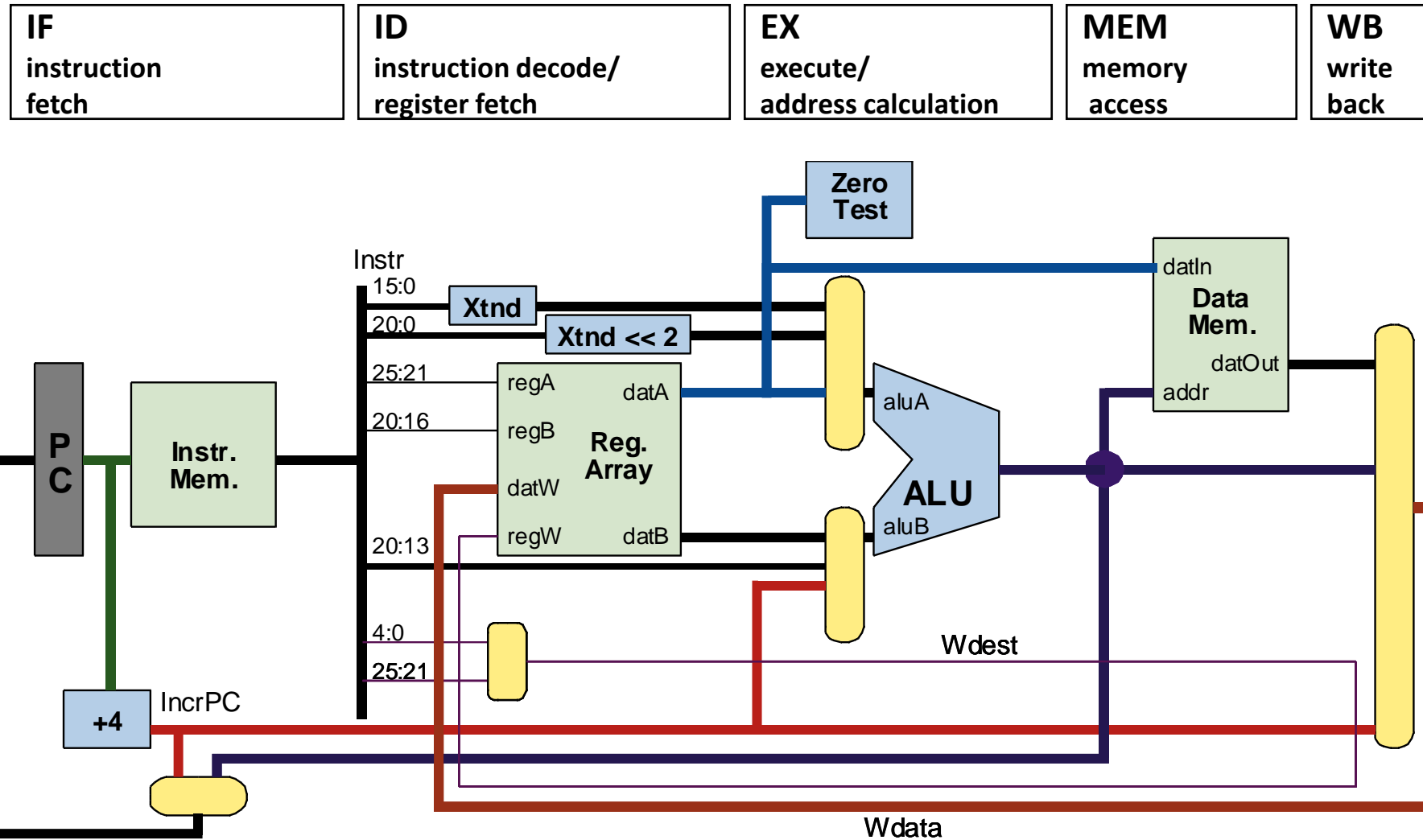
Datapath: Block diagram for a computer processor, excluding control signals



Datapath



Datapath



Hardware Units

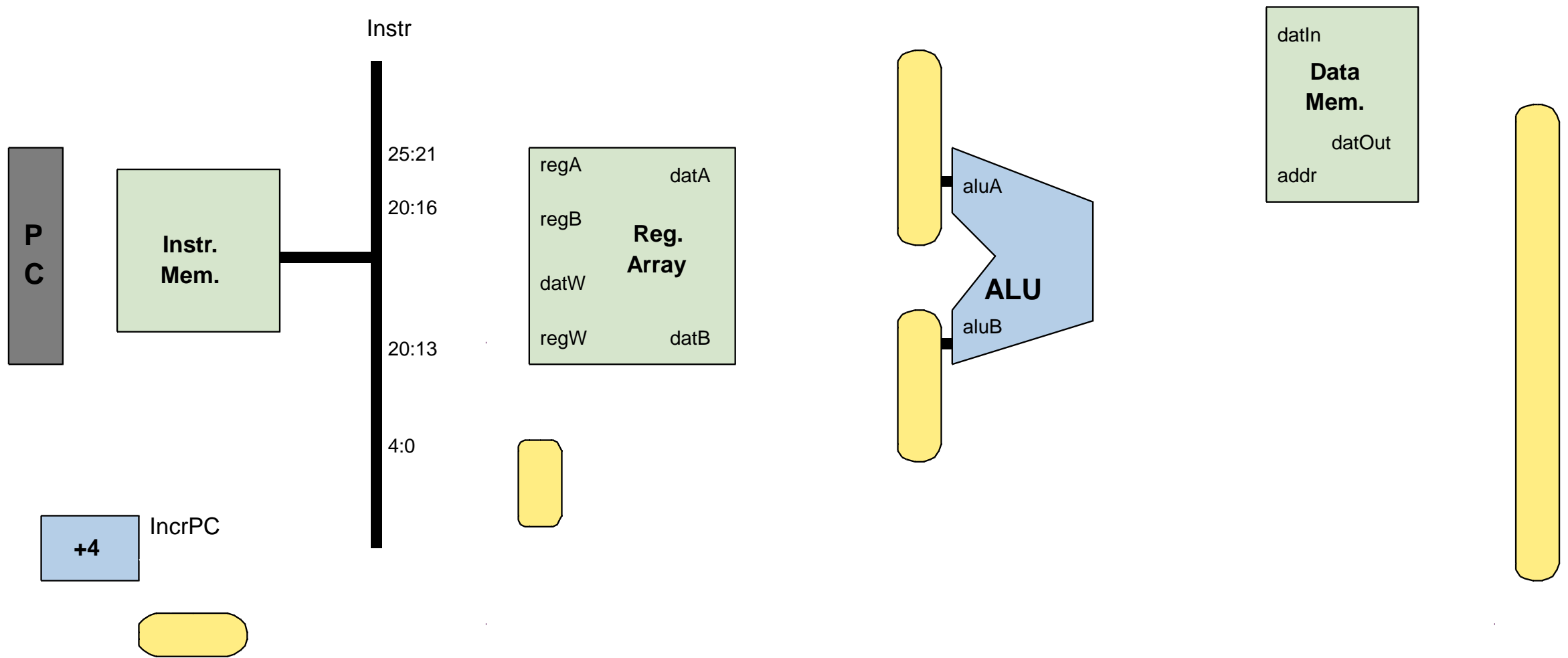
Storage

- **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 (the big one!) |
| ◦ Zero Test | Detect whether operand == 0 |

Datapath



Register-register Instructions

IF: Instruction fetch

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

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

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

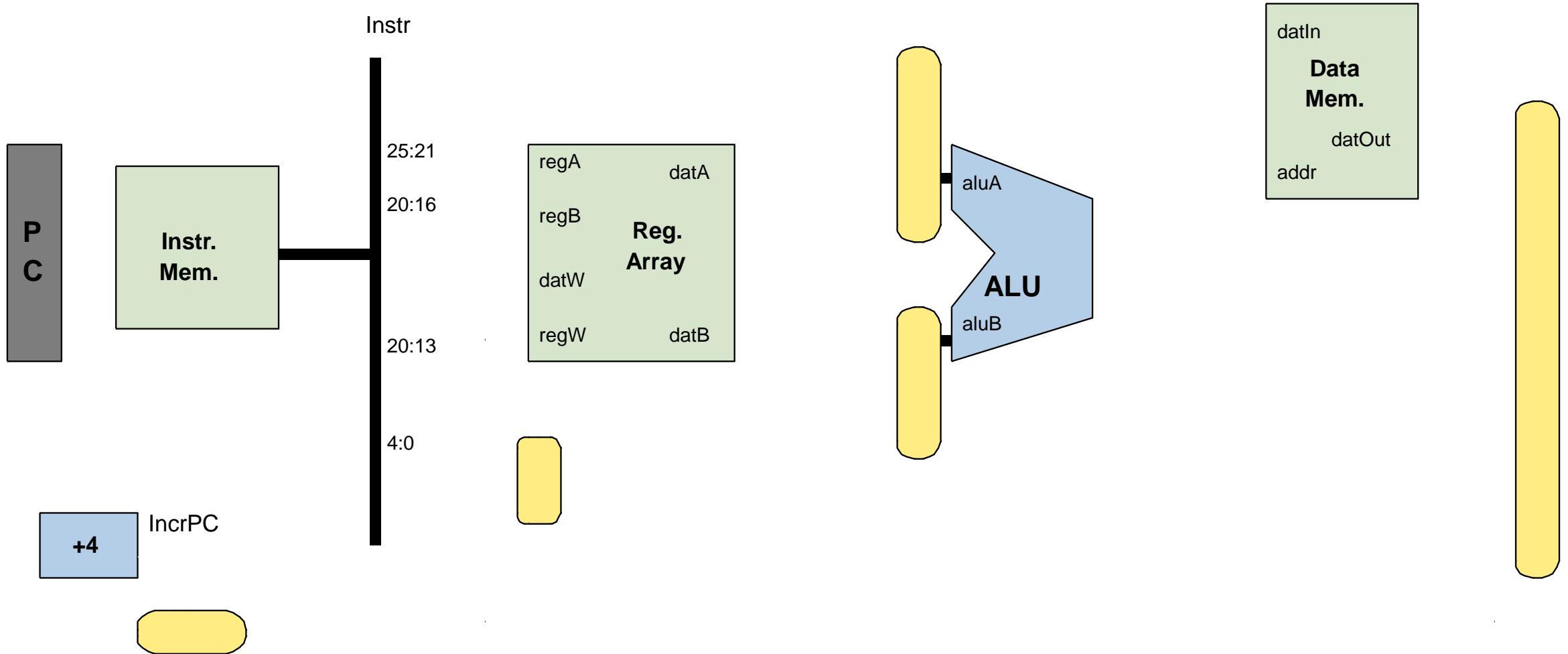
- nop

WB: Write back

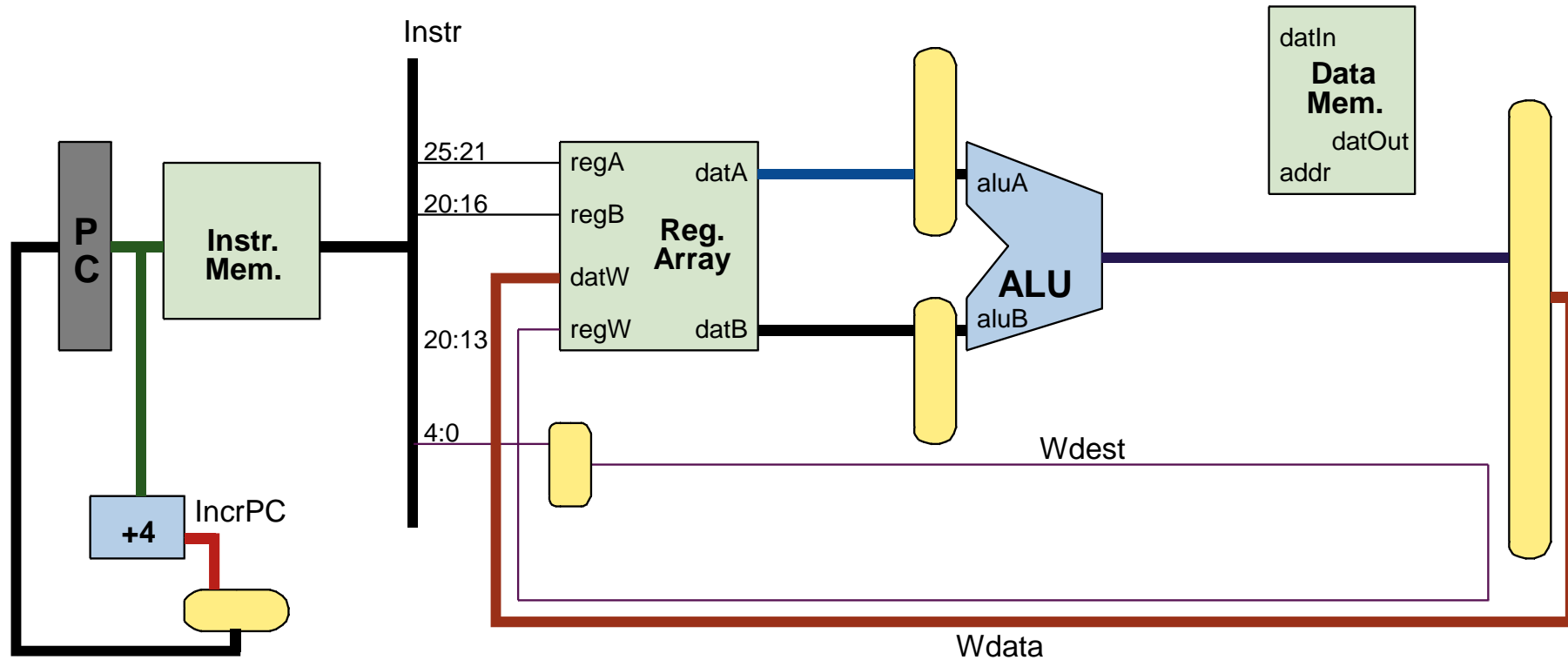
- $Register[IR[4:0]] \leftarrow ALUOutput$

Datapath

$rc \leftarrow ra \text{ funct } rb$



Active Datapath for Reg-Reg Instructions



ALU Operation set per op type

Writeback to Rc

Register-Immediate Instructions

IF: Instruction fetch

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

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

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

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

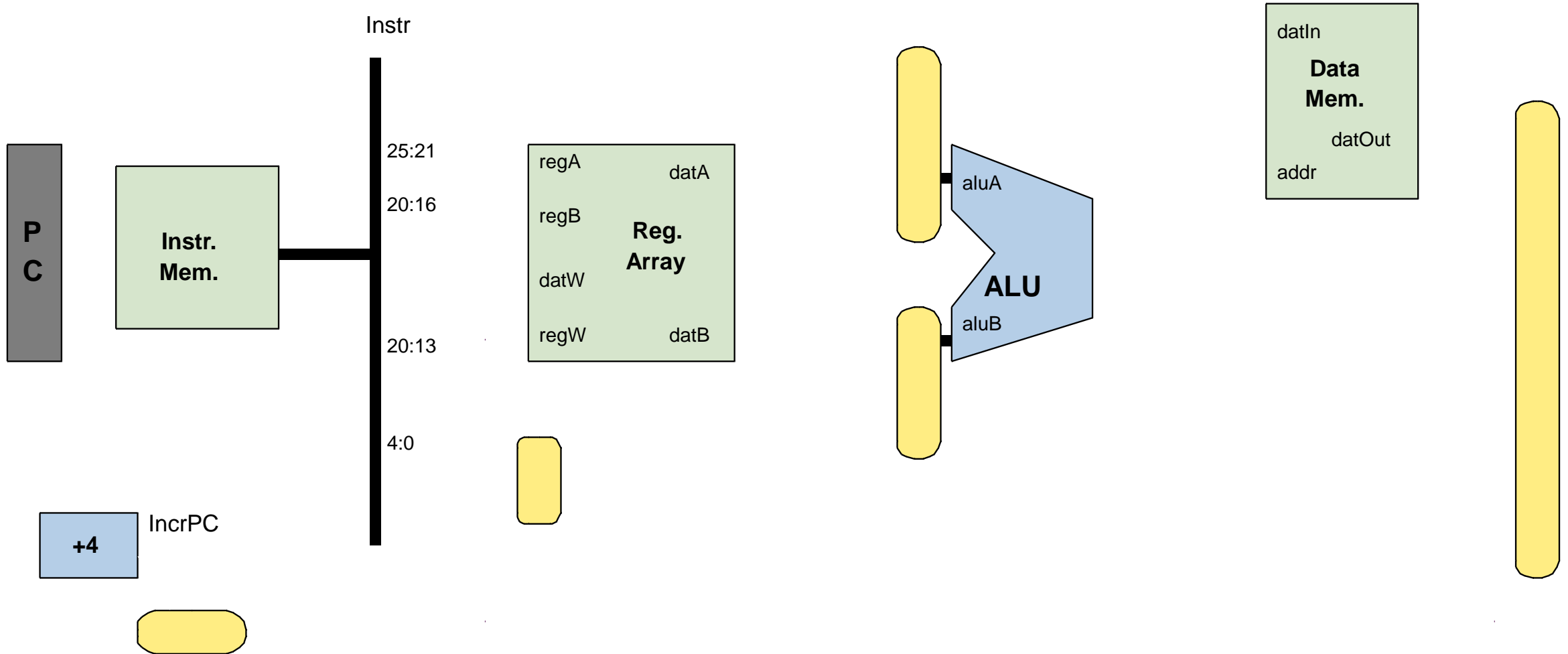
- nop

WB: Write back

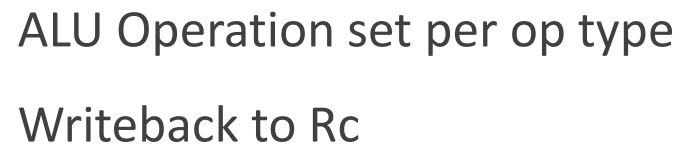
- $\text{Register}[IR[4:0]] \leftarrow ALUOutput$

Datapath

rc <-- ra funct ib



24

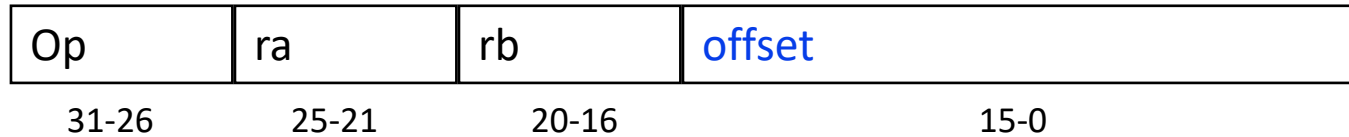


Load Instruction

IF: Instruction fetch

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

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



ID: Instruction decode/register fetch

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

Ex: Execute

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

MEM: Memory

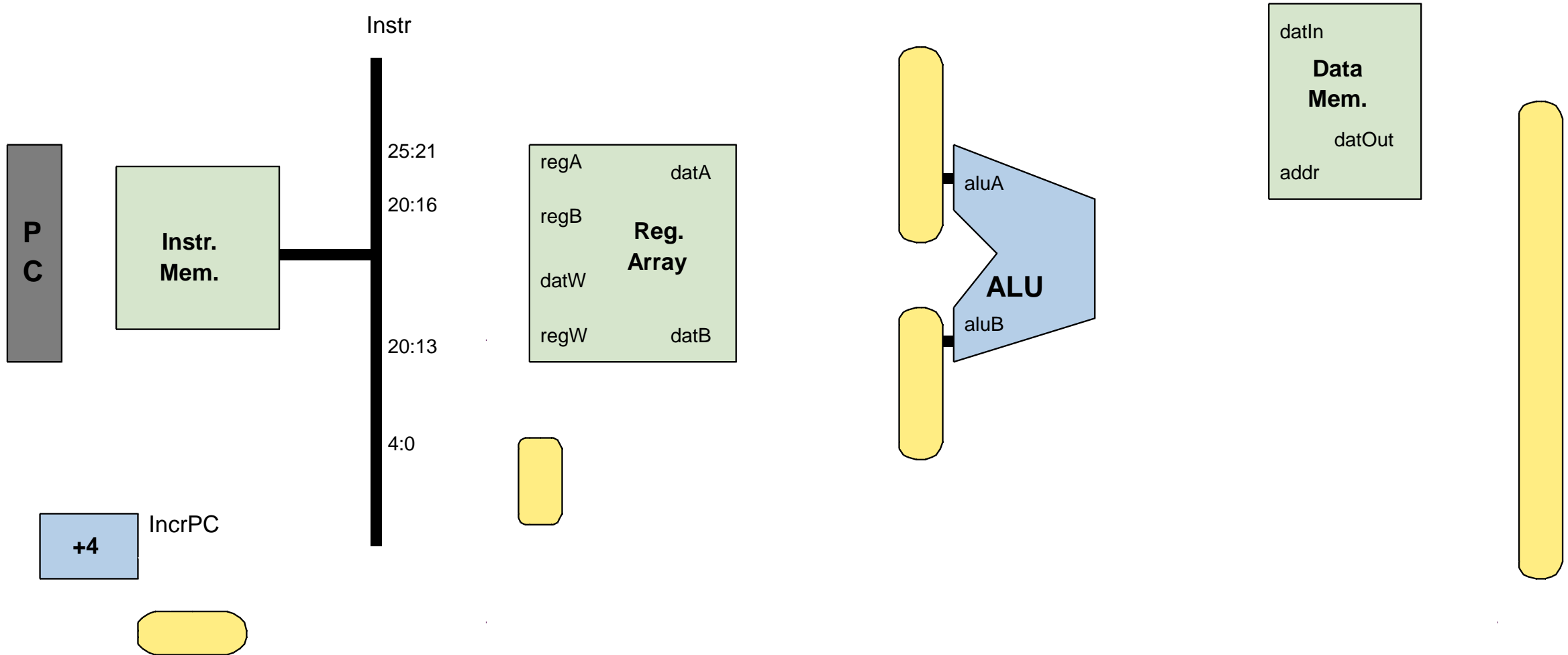
- $Mem\text{-}Data \leftarrow DMemory[ALUOutput]$

WB: Write back

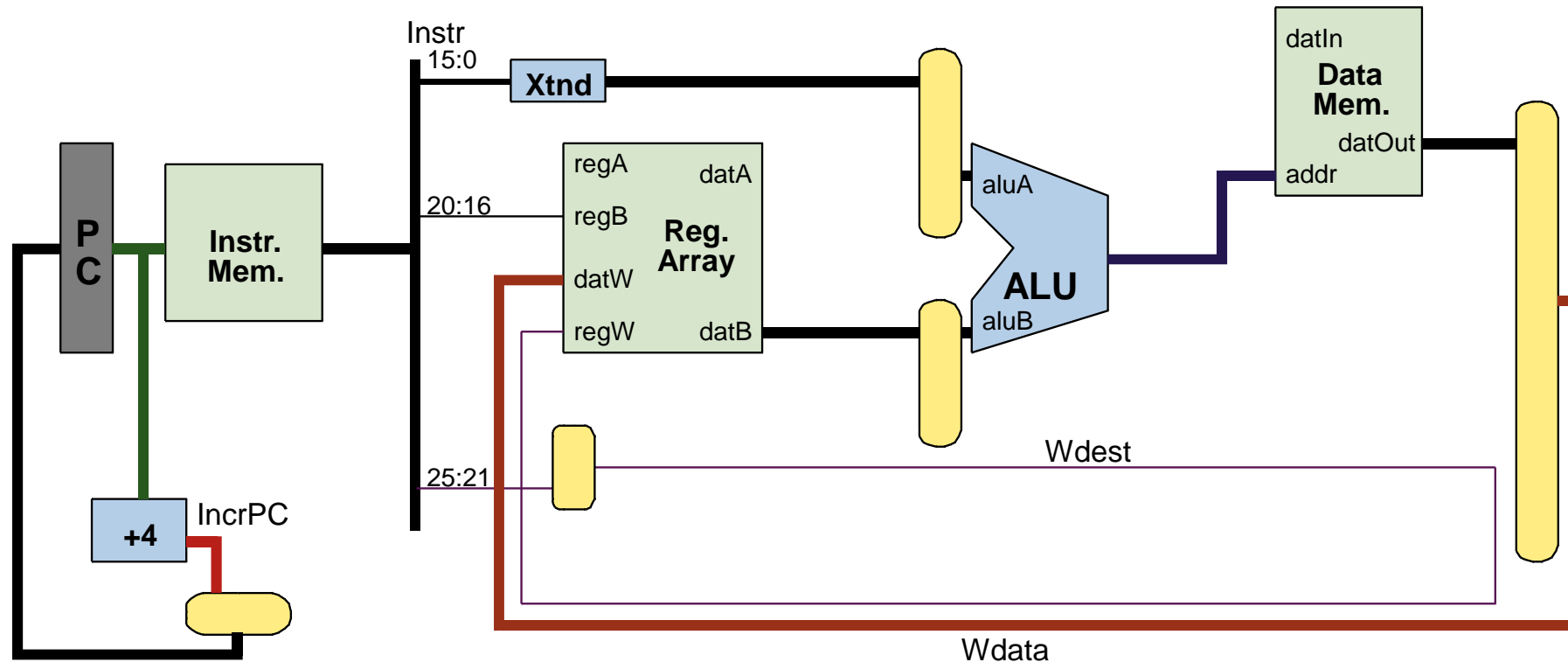
- $Register[IR[25:21]] \leftarrow Mem\text{-}Data$

Datapath

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



Active Datapath for Load



ALU 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

IF: Instruction fetch

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

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

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

ID: Instruction decode/register fetch

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

Ex: Execute

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

MEM: Memory

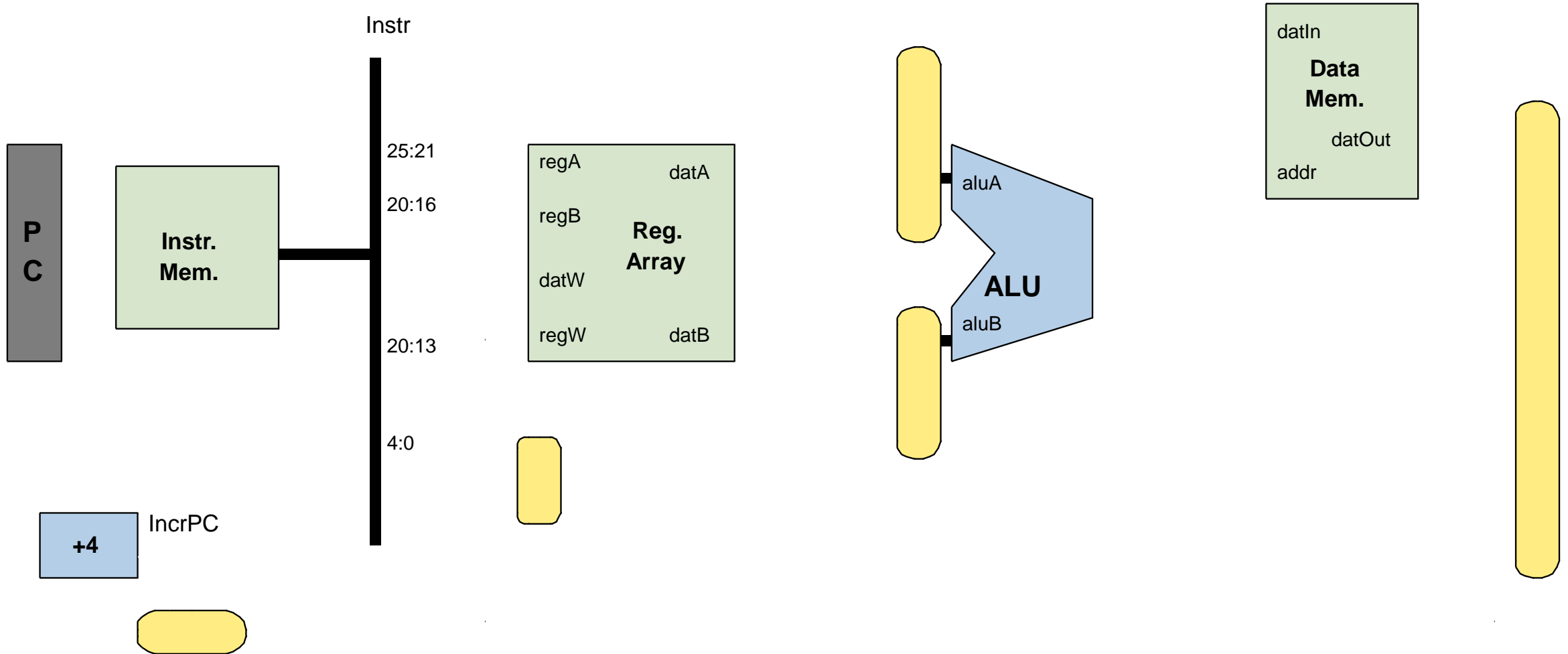
- $DMemory[ALUOutput] \leftarrow A$

WB: Write back

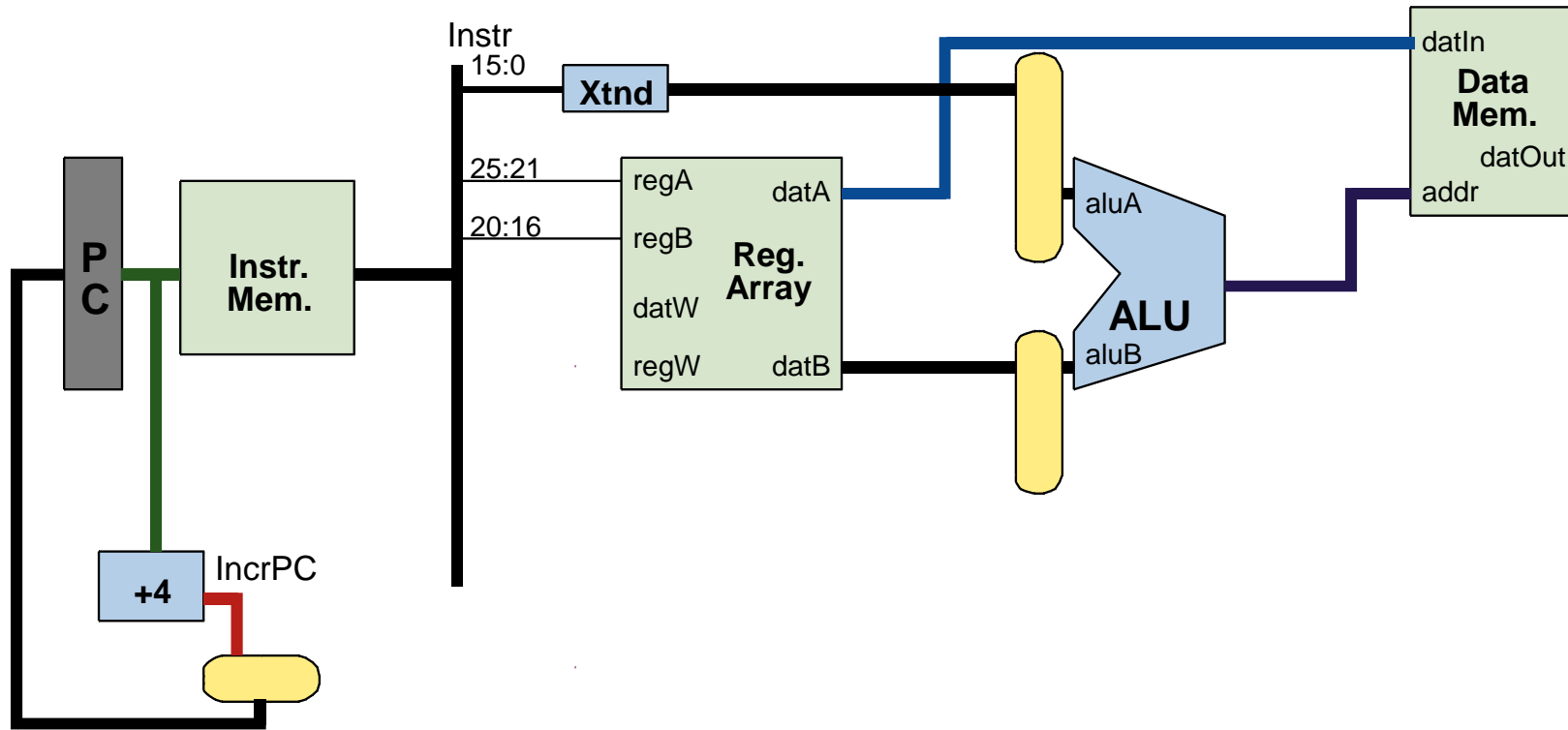
- nop

Datapath

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



Active Datapath for Store



ALU used to compute address

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

Memory Operation

- Write

Write Back

- None

Conditional Branch Instruction

IF: Instruction fetch

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

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

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

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

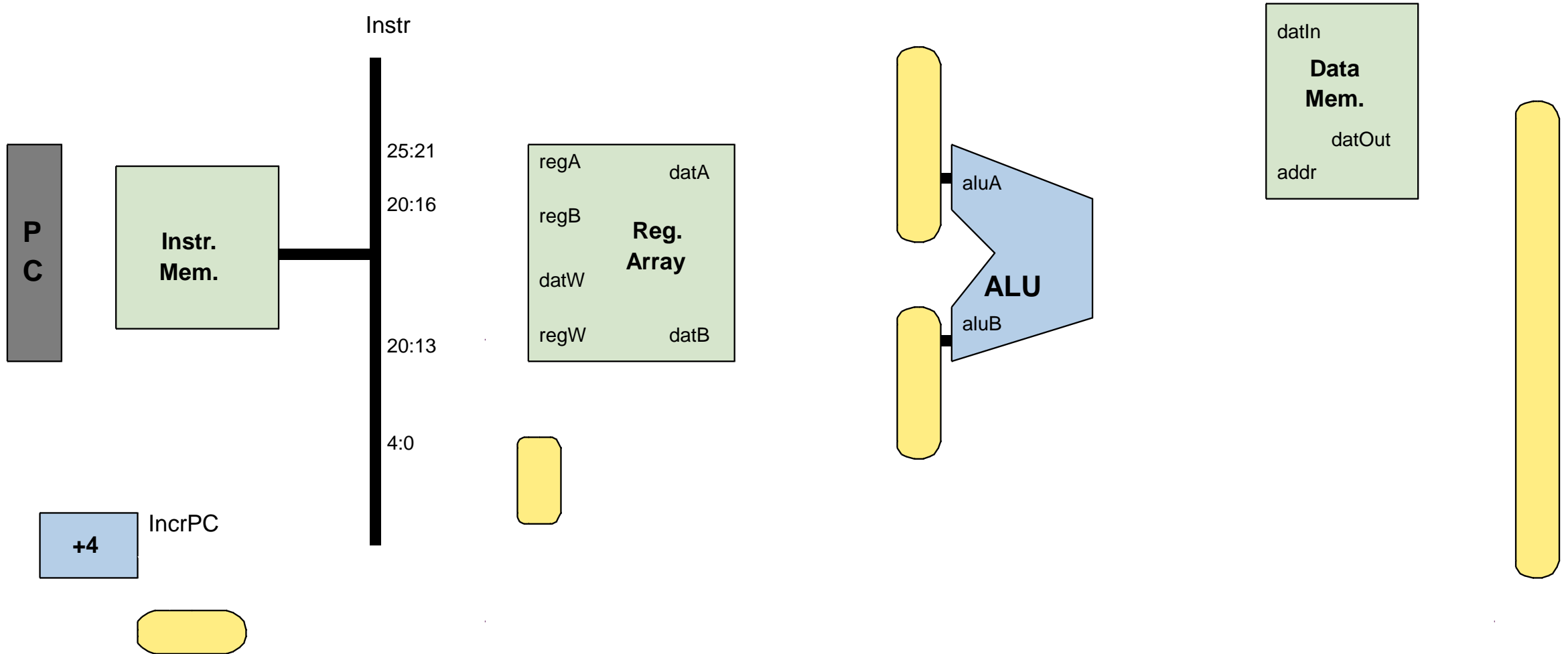
- $PC \leftarrow Z ? Target : incrPC$

WB: Write back

- nop

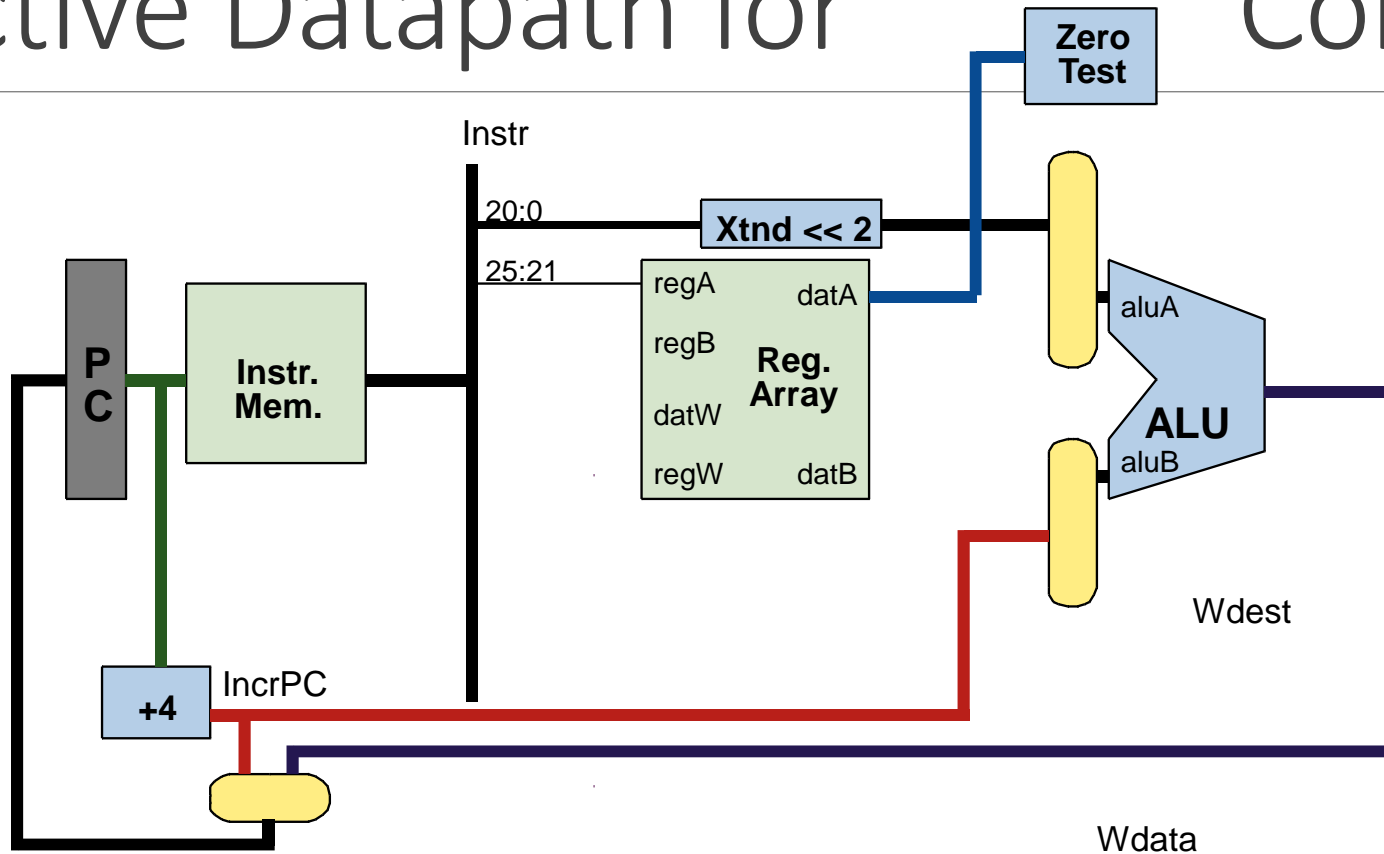
Datapath

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



Active Datapath for

Cond. Branch



ALU computes target

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

Zero Test

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

PC Selection

- Target for taken branch
- IncrPC for not taken

Write Back

- None

Branch to Subroutine

IF: Instruction fetch

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

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

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

ID: Instruction decode/register fetch

- nop

Ex: Execute

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

MEM: Memory

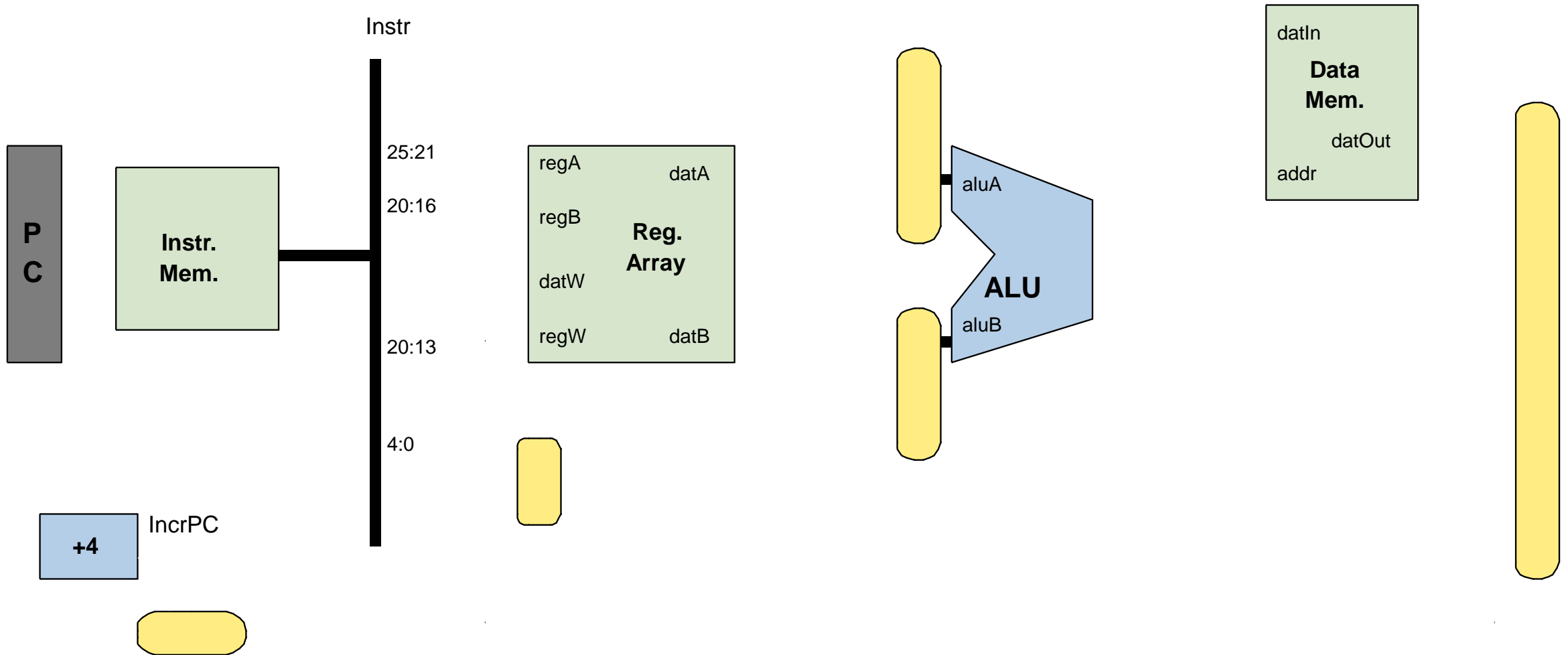
- $PC \leftarrow Target$

WB: Write back

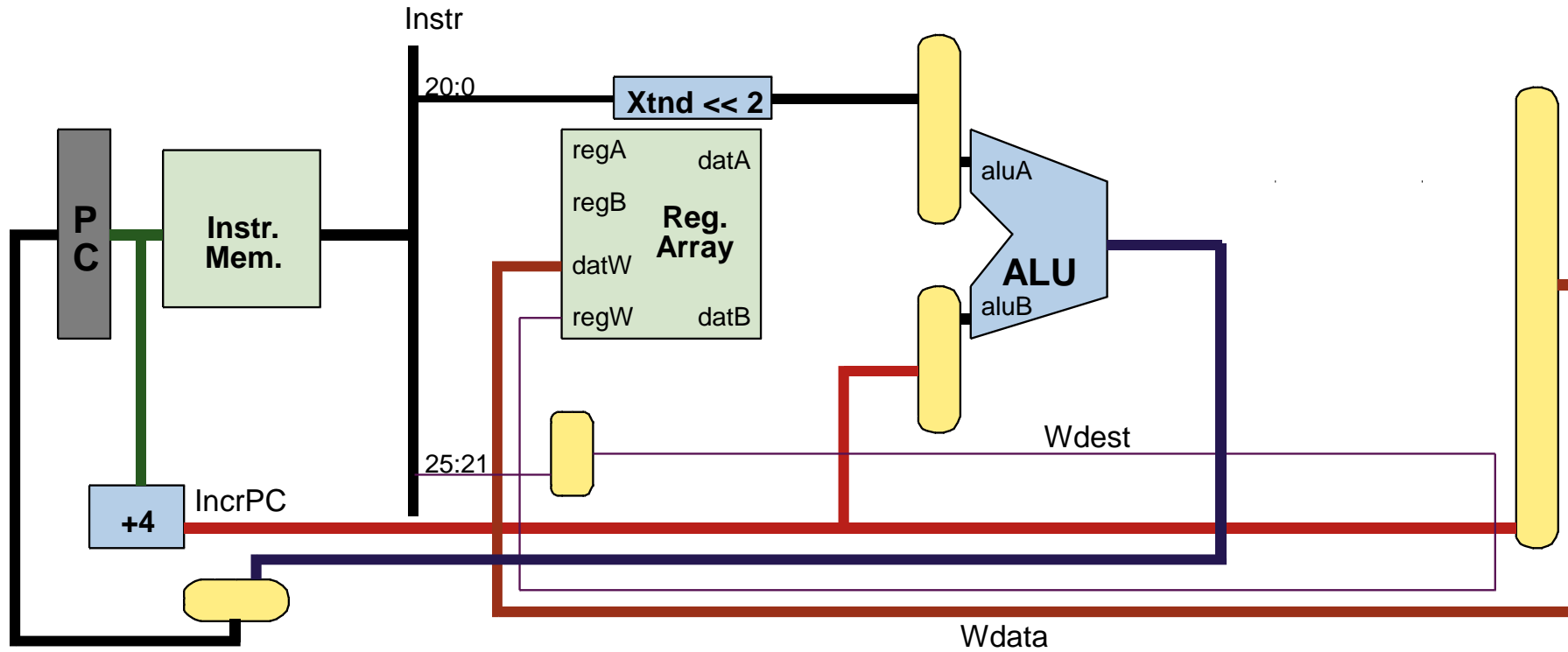
- $Register[IR[25:21]] \leftarrow incrPC$

Datapath

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



Active Datapath for Branch to Subroutine



ALU computes target

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

PC Selection

- Always target

Write Back

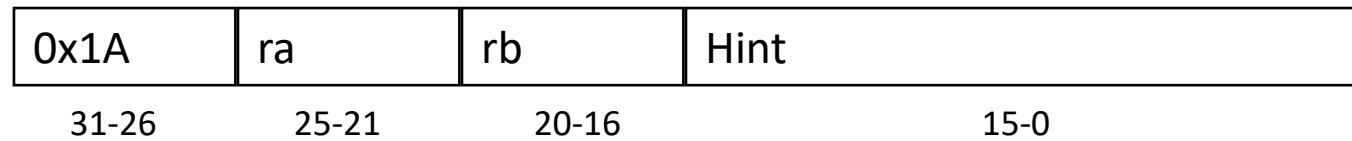
- Incremented PC as data

Jump Instruction

IF: Instruction fetch

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

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



ID: Instruction decode/register fetch

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

Ex: Execute

- $Target \leftarrow B$

MEM: Memory

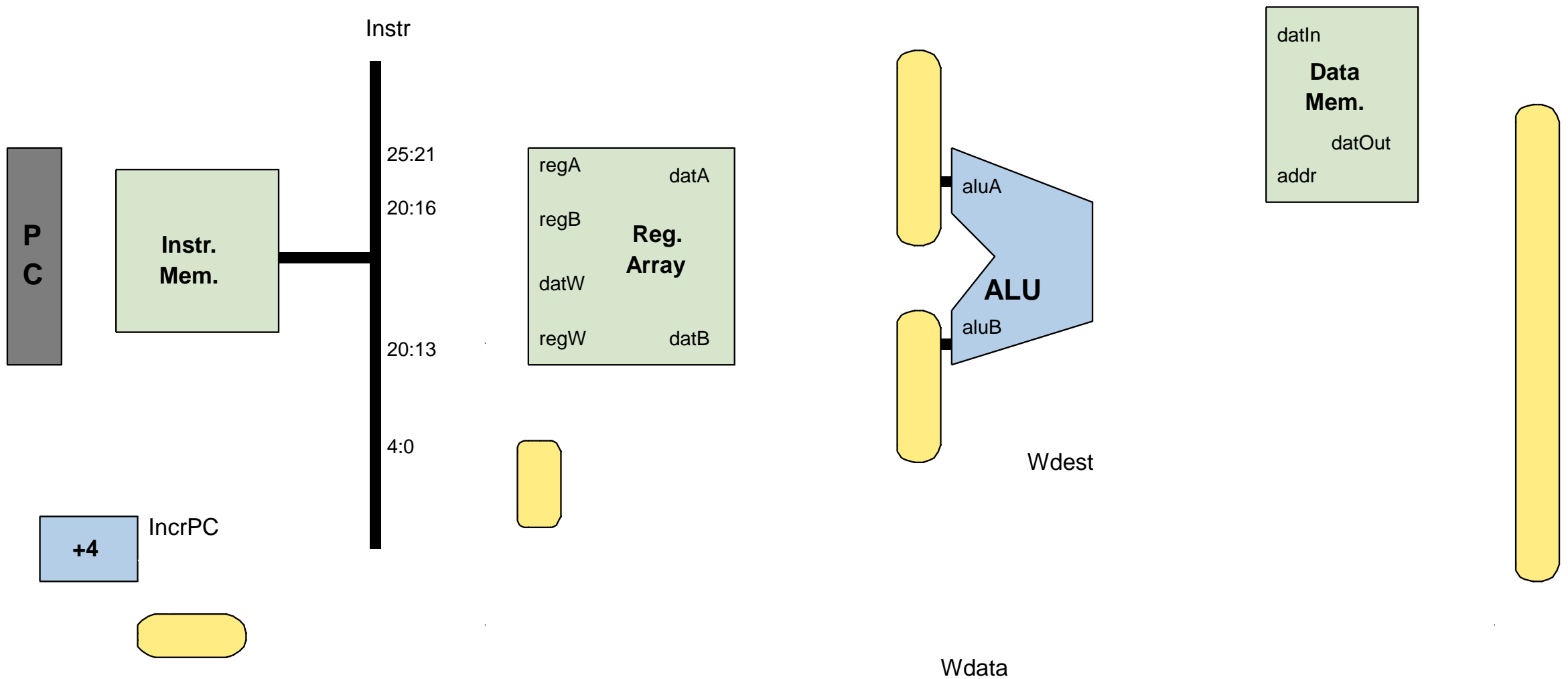
- $PC \leftarrow target$

WB: Write back

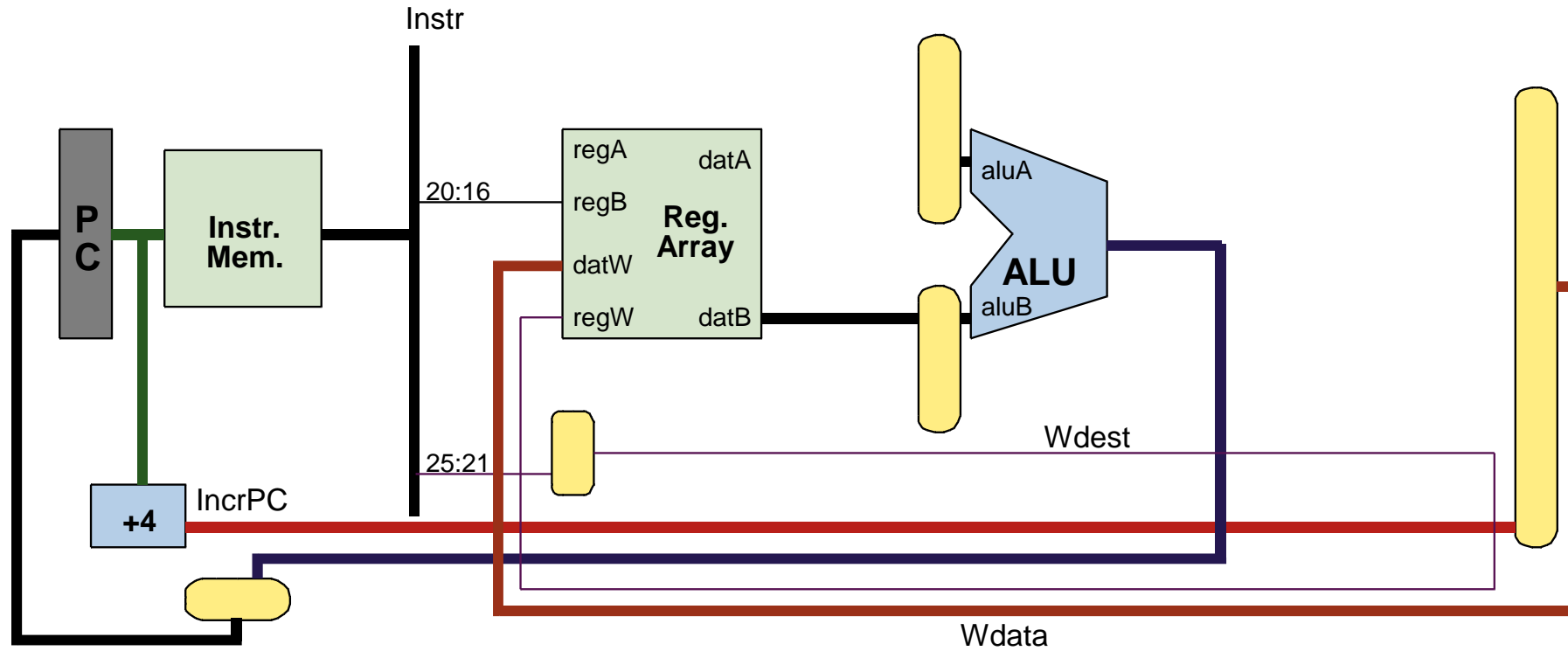
- $Register[IR[25:21]] \leftarrow incrPC$

Datapath

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



Active Datapath for Jumps



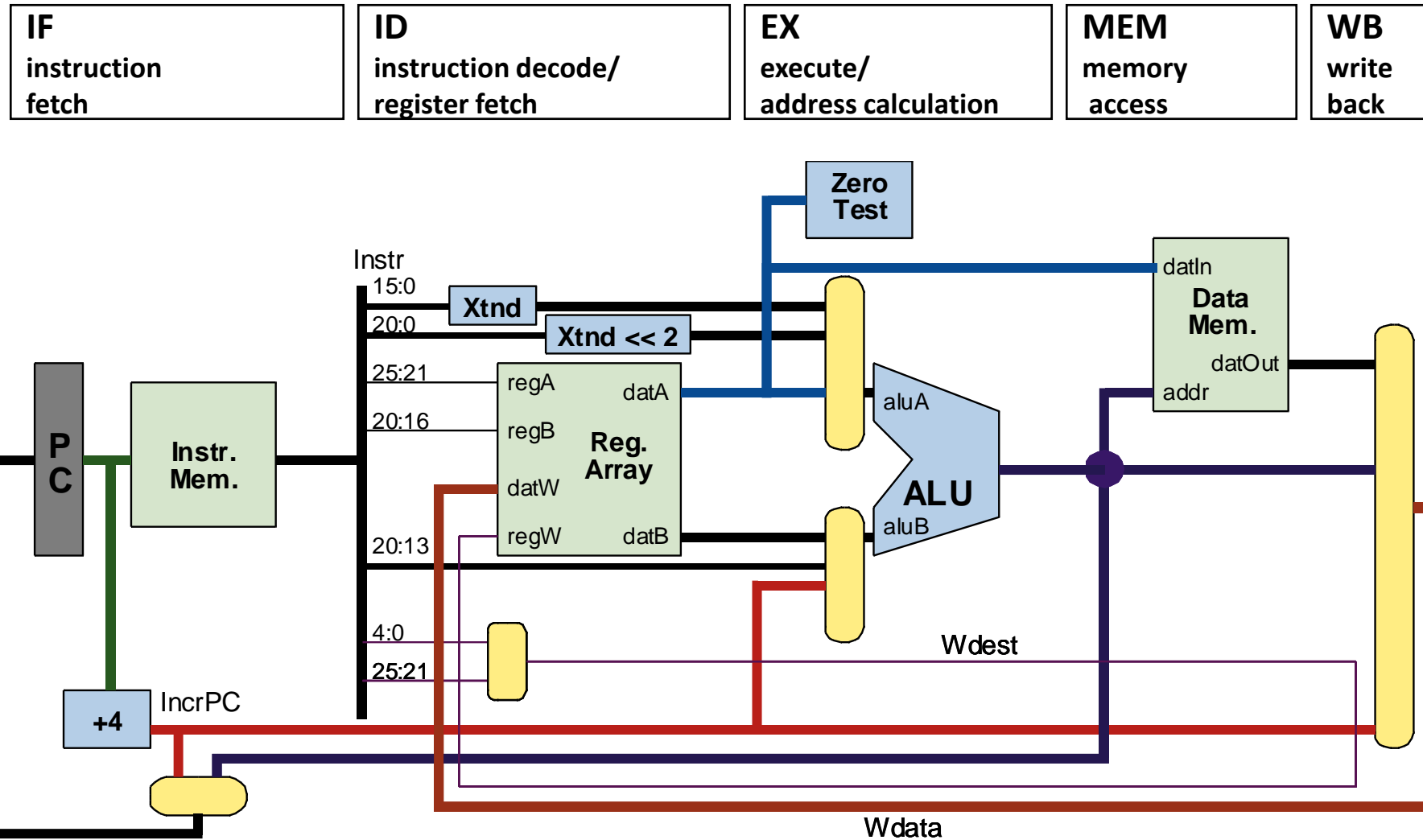
ALU used to compute target

- B input set to Rb
- ALU function set to select B

Write Back

- To Ra
- IncrPC as data

Complete Datapath



What about control?

Implicitly defined already when we looked at each instruction type!

Define control signals as function of instruction op code etc

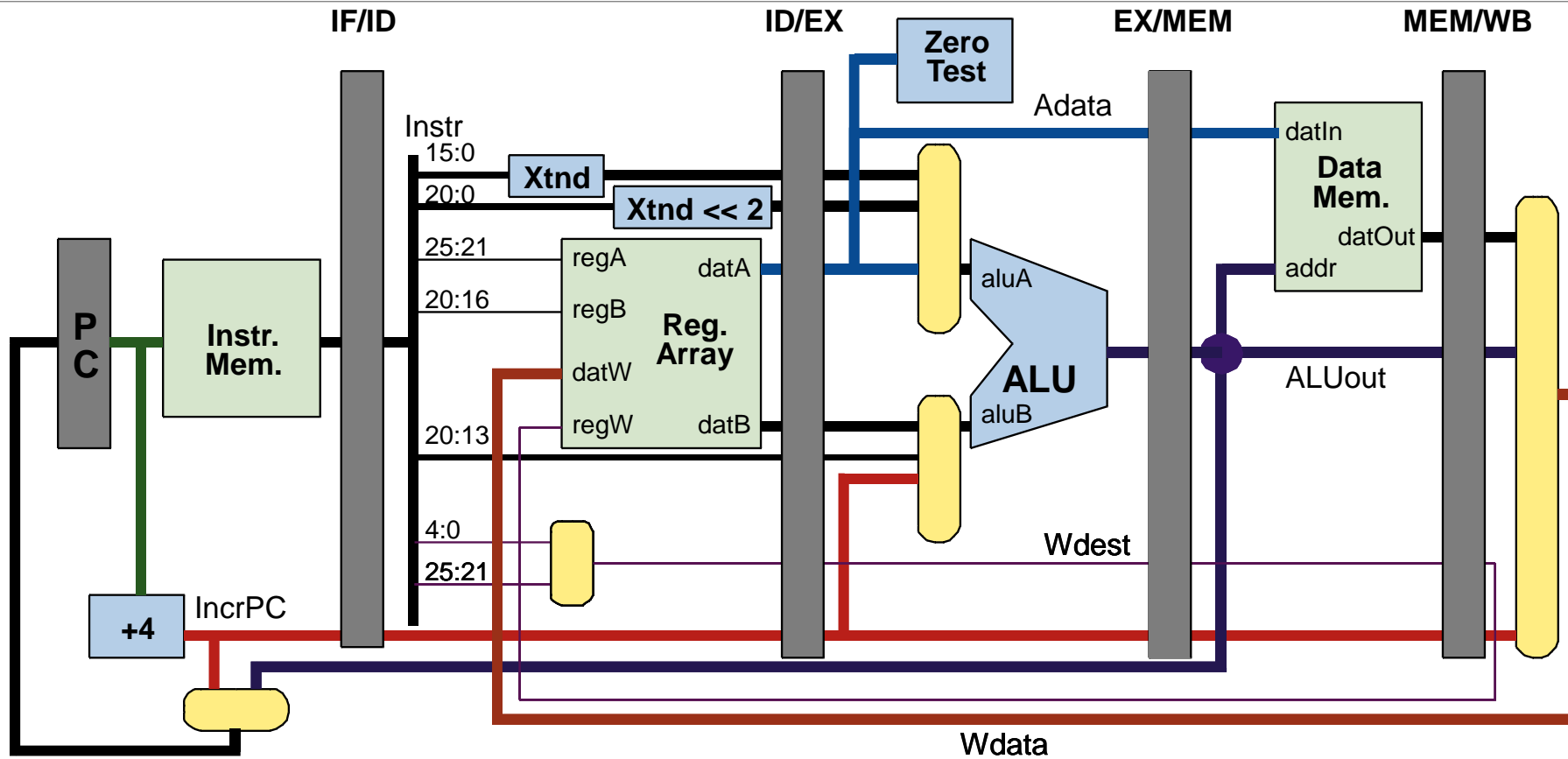
- “1”, “0”, “don’t care”
- ➔ Many ways to generate simple combinational circuit from logic tables
- (Can also be clever about encoding to simplify logic)

Control is more complex with multi-cycle implementations

- E.g., microprogramming

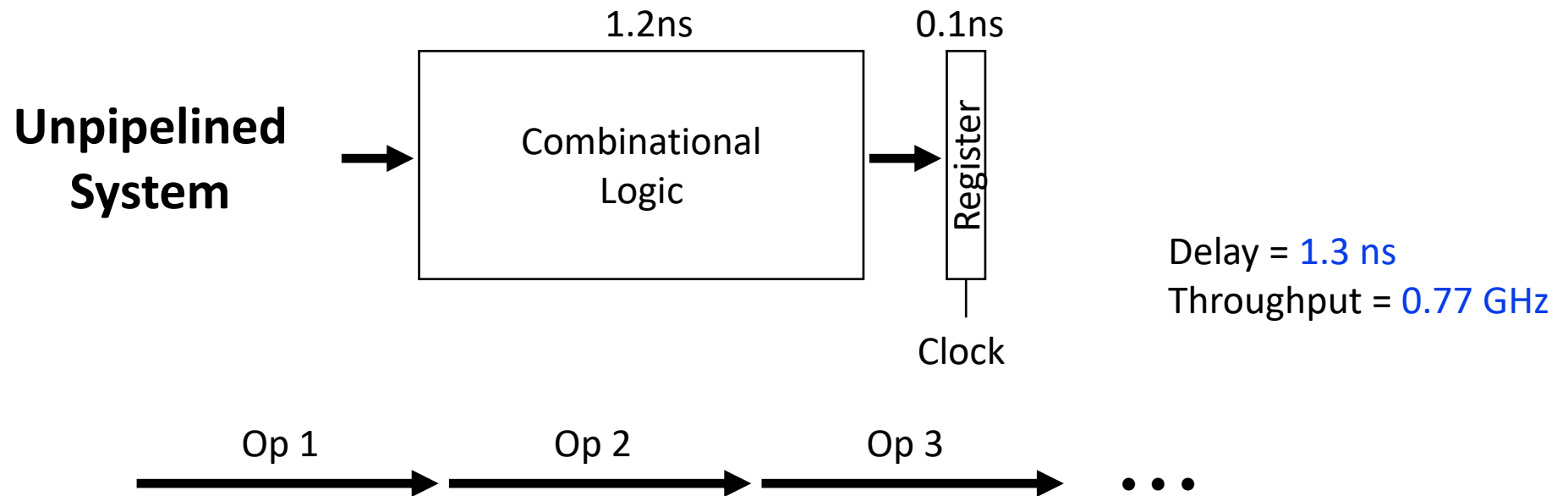
Pipelining: Introduction

Pipelined Datapath

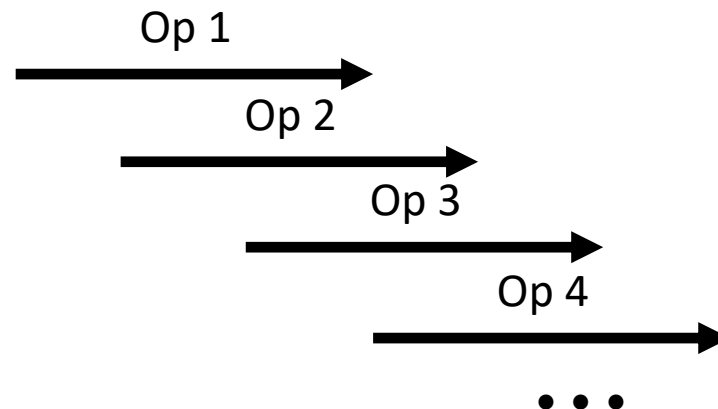
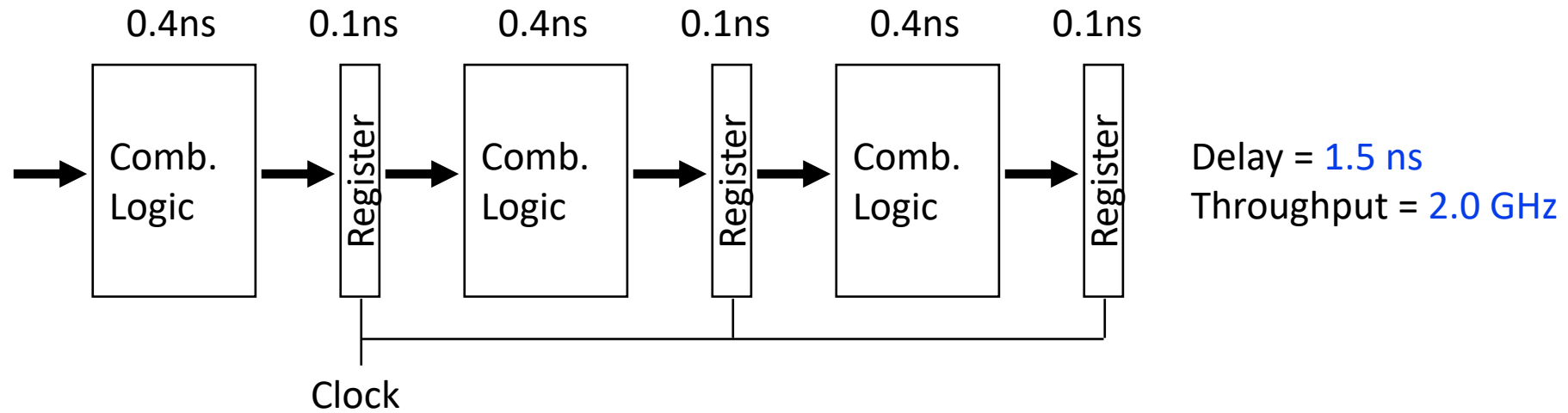


Same datapath, $\approx 5X$ higher peak throughput!

Pipelining Basics

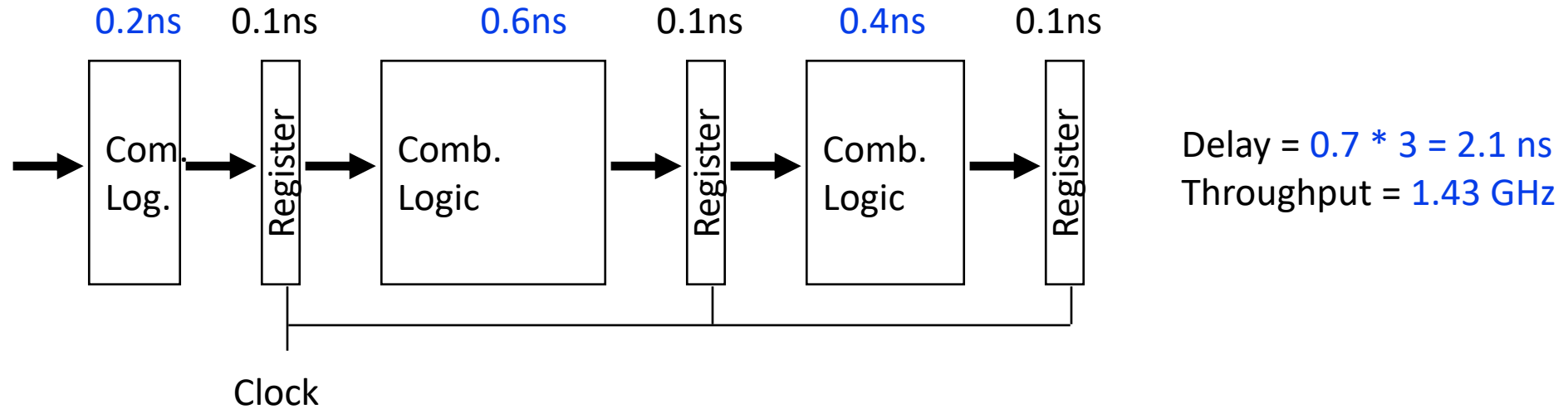


3-Stage Pipeline



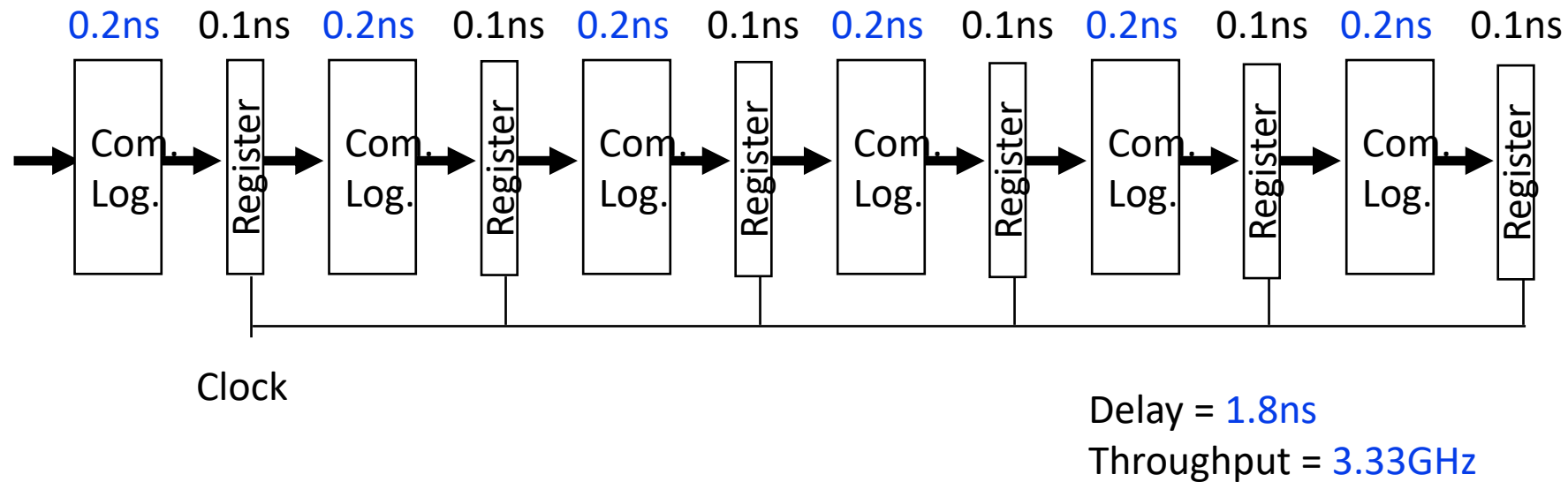
- Space operations 0.5ns apart
- 3 operations occur simultaneously

Limitation: Non-Uniform Pipelining



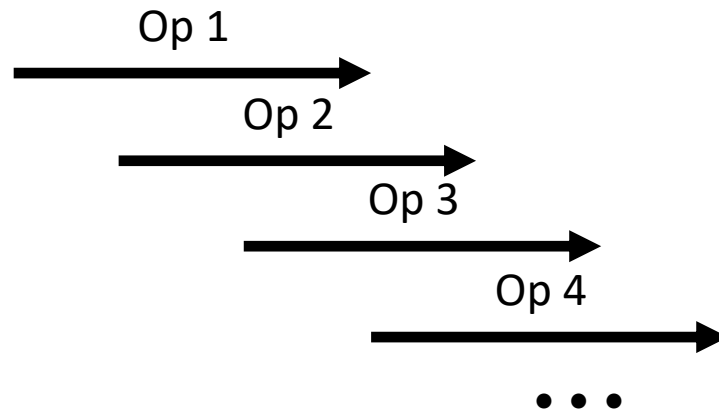
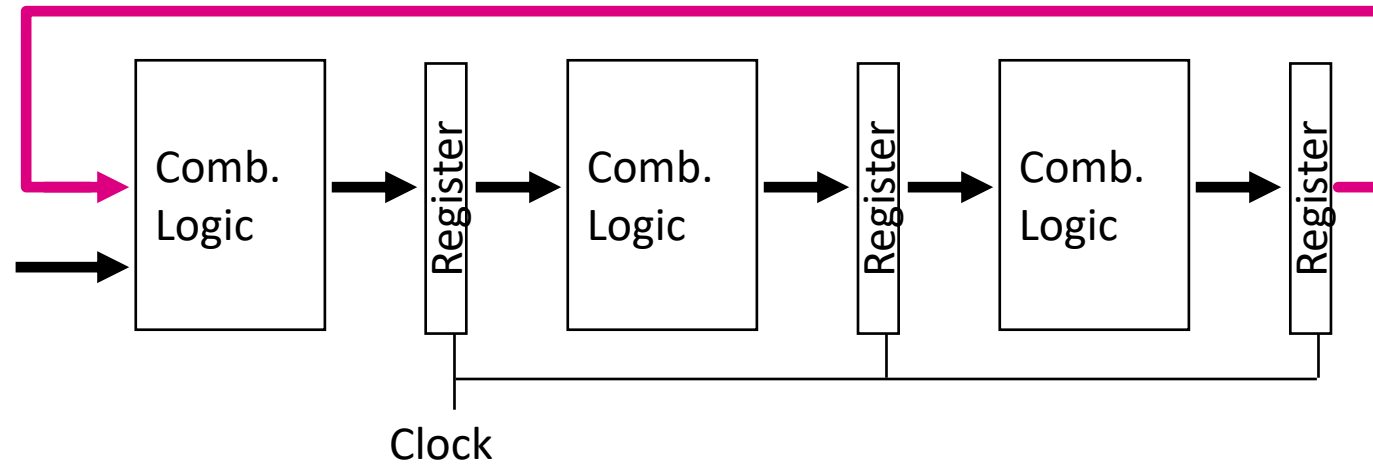
- Throughput limited by slowest stage
- Delay determined by clock period (= max stage latency) × number of stages
- ➔ Must balance stages to maximize performance

Limitation: Deep Pipelines



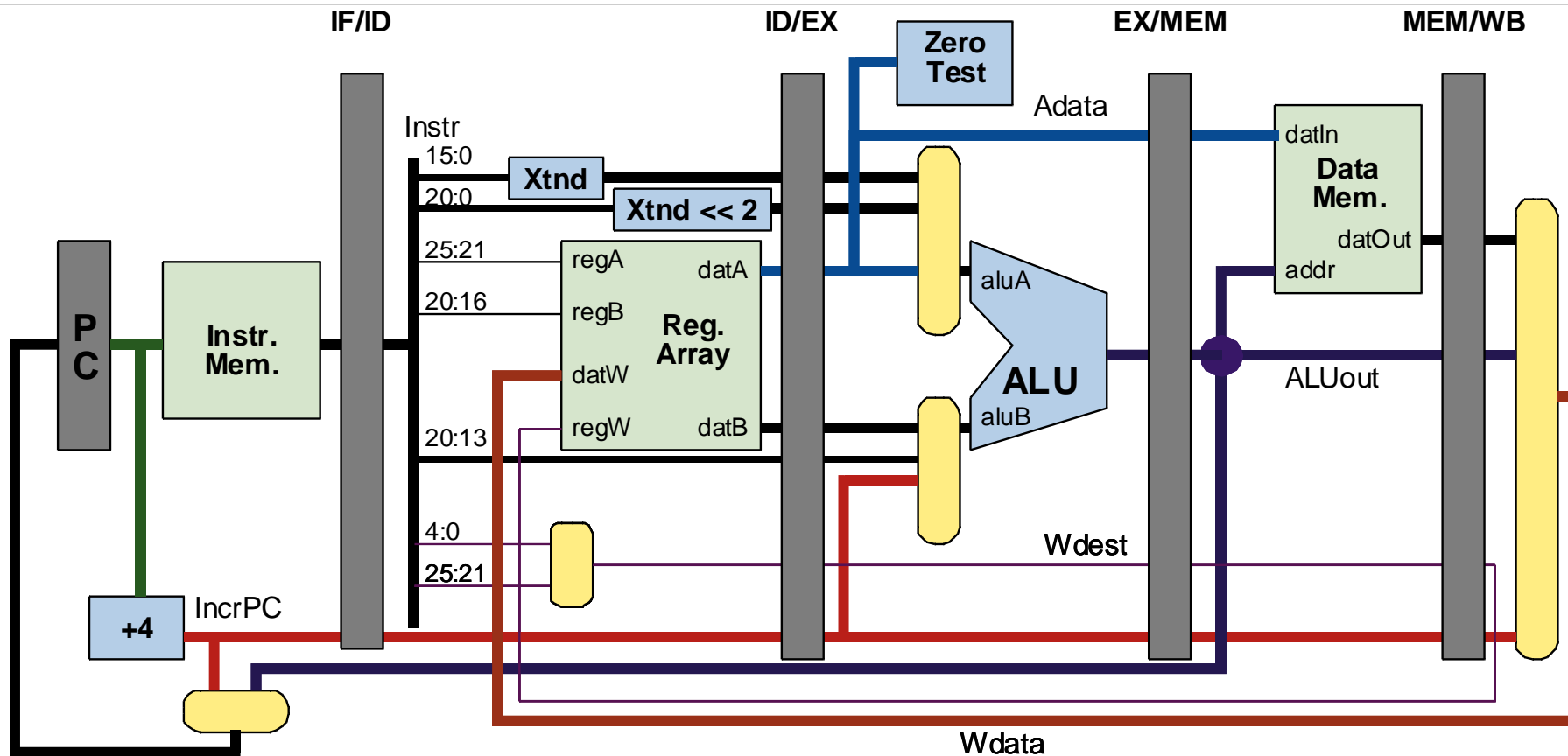
- Diminishing returns as add more pipeline stages
- Register delays become limiting factor
- One of the 3 main reasons sequential performance plateaued
 - Power, diminishing pipeline returns, pipeline hazards (see next slides)

Limitation: Sequential Dependences



- Op4 gets result from Op3!
- *Pipeline Hazard* → *Extra delay*

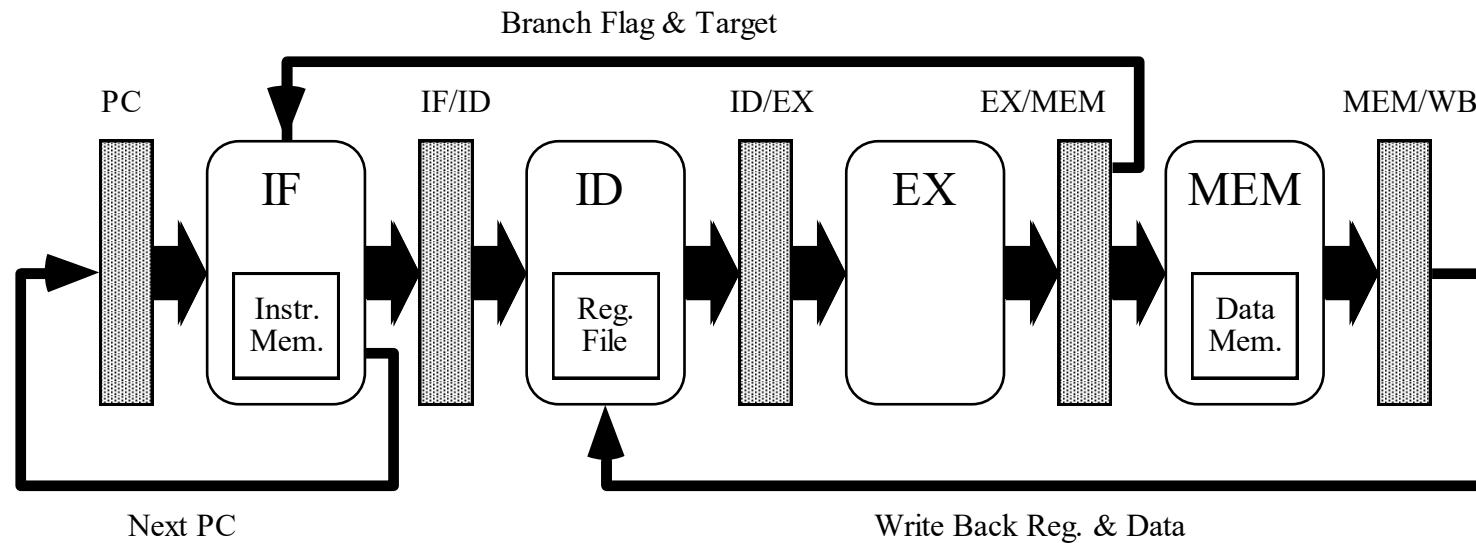
Pipelined Datapath



Pipe Registers

- Inserted between stages
- Labeled by preceding & following stage

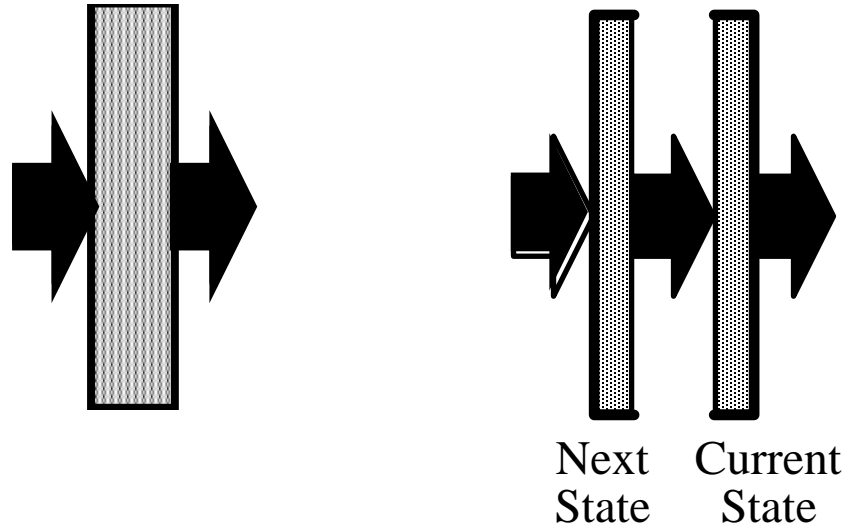
Pipeline Structure



Notes

- Each stage consists of 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

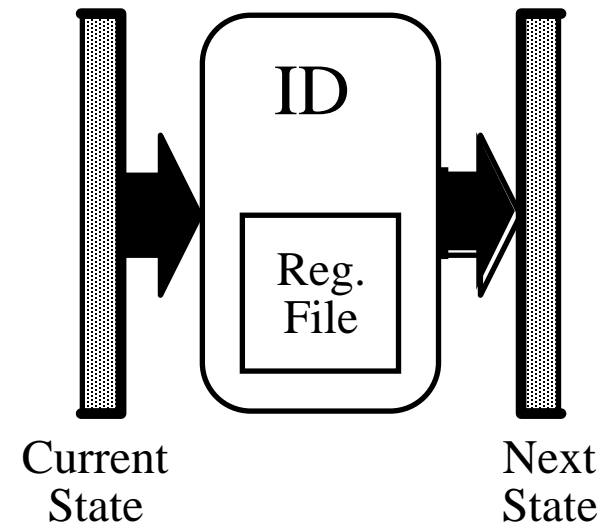
Pipeline Stage

Computes next state based on current

- From/to one or more pipe registers

May have internal memory elements (e.g., Reg File)

- Low-level timing signals control their operation during clock cycle
 - E.g., we assume reg reads get written value in same 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?

\$2	63
\$3	63
\$4	63
\$5	63
\$6	63

Sequential execution

addq \$2, 63, \$2

addq \$2, 0, \$3

addq \$2, 0, \$4

addq \$2, 0, \$5

addq \$2, 0, \$6

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?

\$2	63
\$3	?
\$4	?
\$5	?
\$6	?

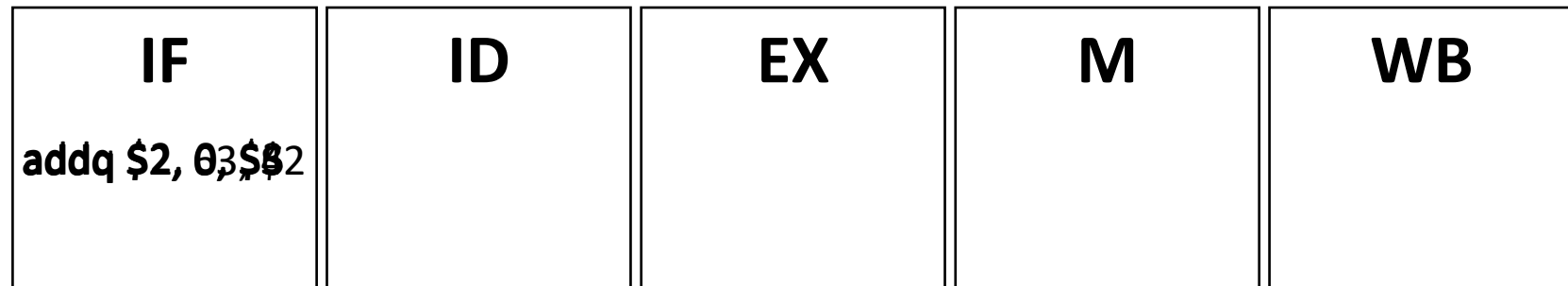
addq \$2, 63, \$2

addq \$2, 0, \$3

addq \$2, 0, \$4

addq \$2, 0, \$5

addq \$2, 0, \$6



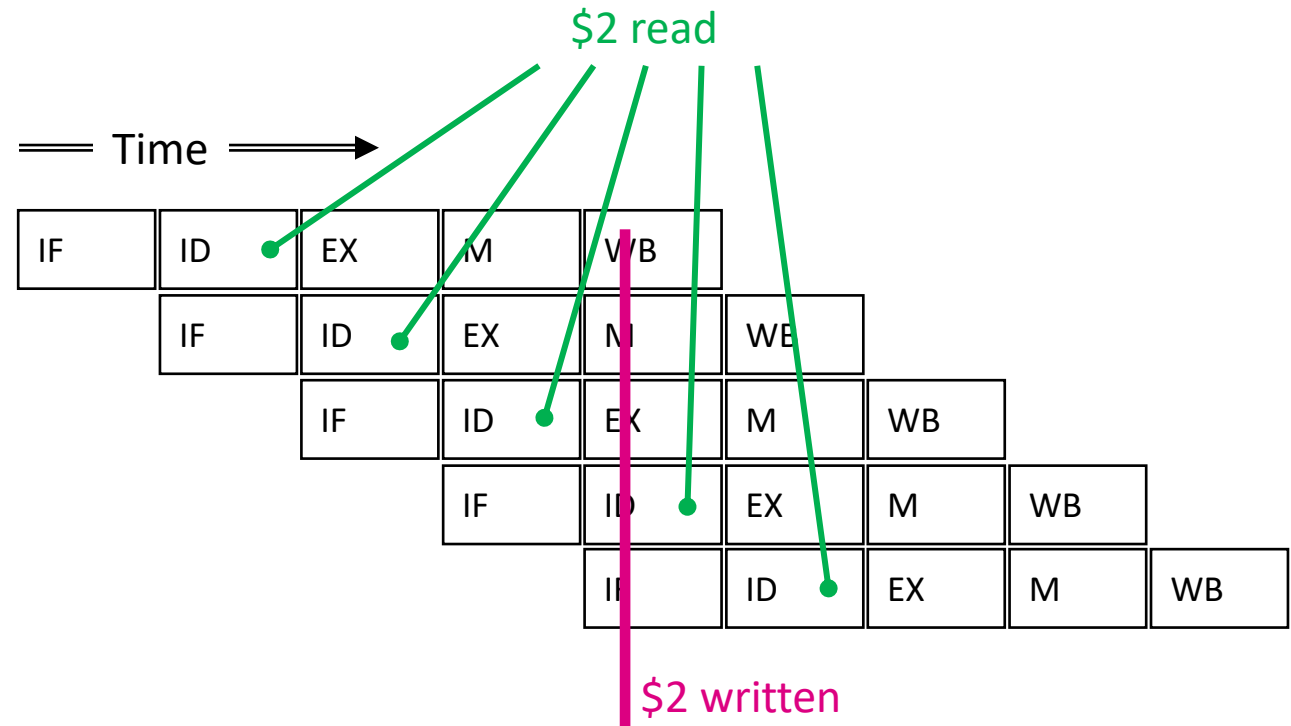
Data Hazards in ALPHA Pipeline

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?

\$2	63
\$3	?
\$4	?
\$5	?
\$6	?

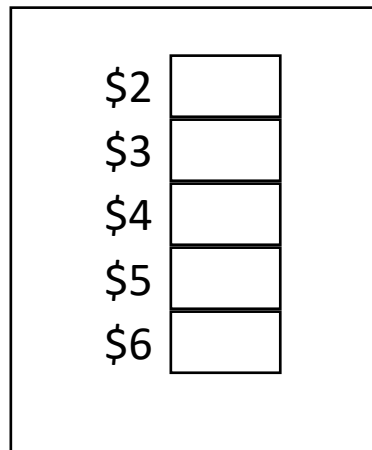
```
addq $2, 63, $2
addq $2, 0, $3
addq $2, 0, $4
addq $2, 0, $5
addq $2, 0, $6
```



Control Hazards in ALPHA Pipeline

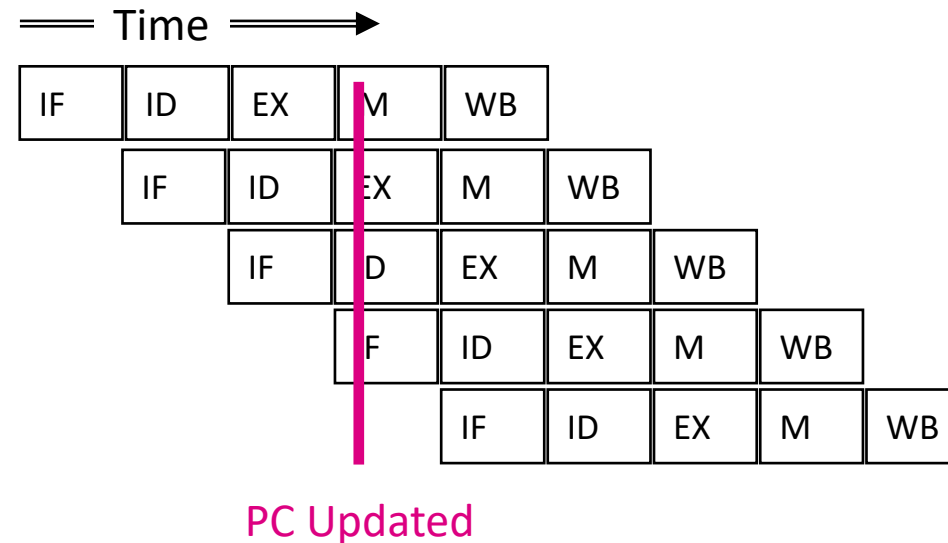
Problem

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



```
beq $0, target
mov 63, $2
mov 63, $3
mov 63, $4
mov 63, $5
```

target: mov 63, \$6



Pipelining Review Thus Far...

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

Pipelining: Dealing with Hazards

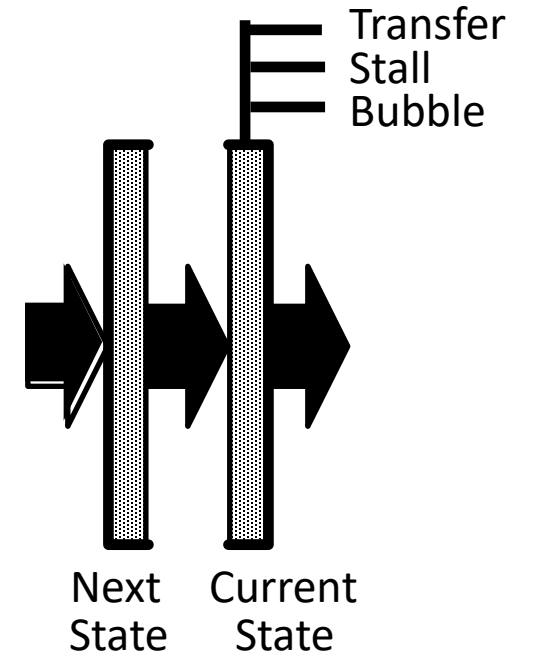
Handling Hazards by Stalling

Idea

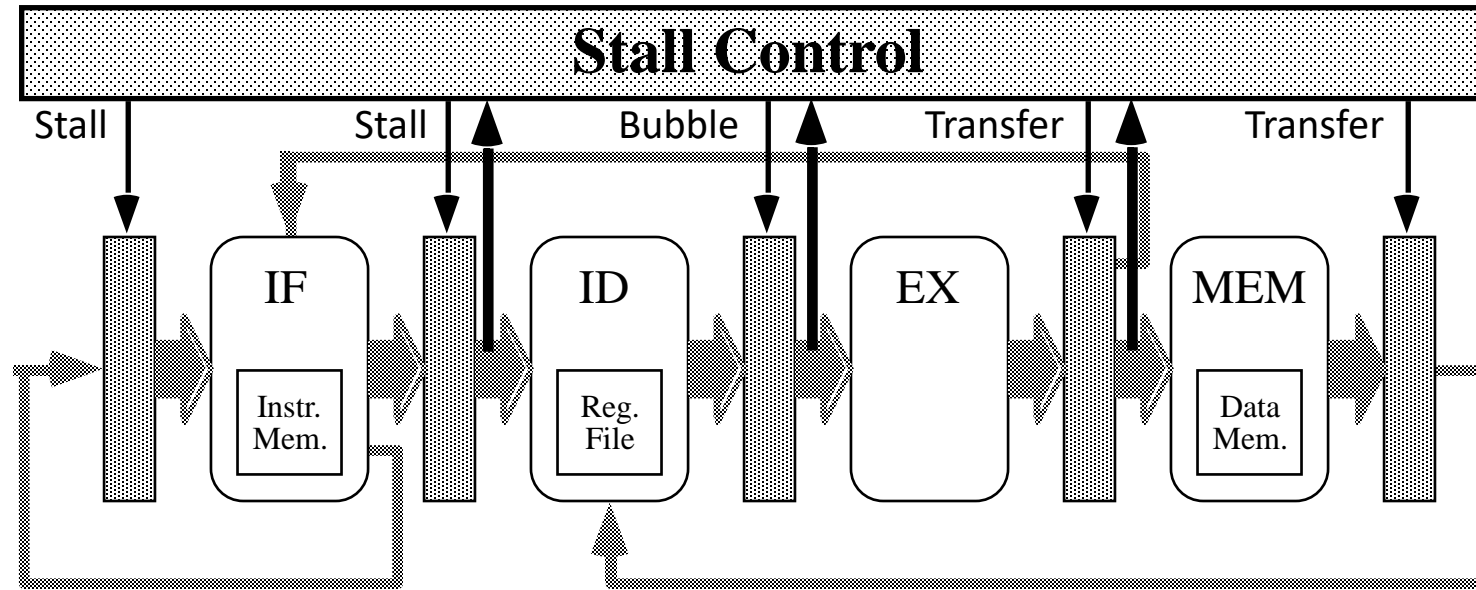
- Delay instruction until hazard eliminated
- Put “bubble” into pipeline
 - Dynamically generated NOP

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



Implementing Stalls



Stall PC, IF/ID

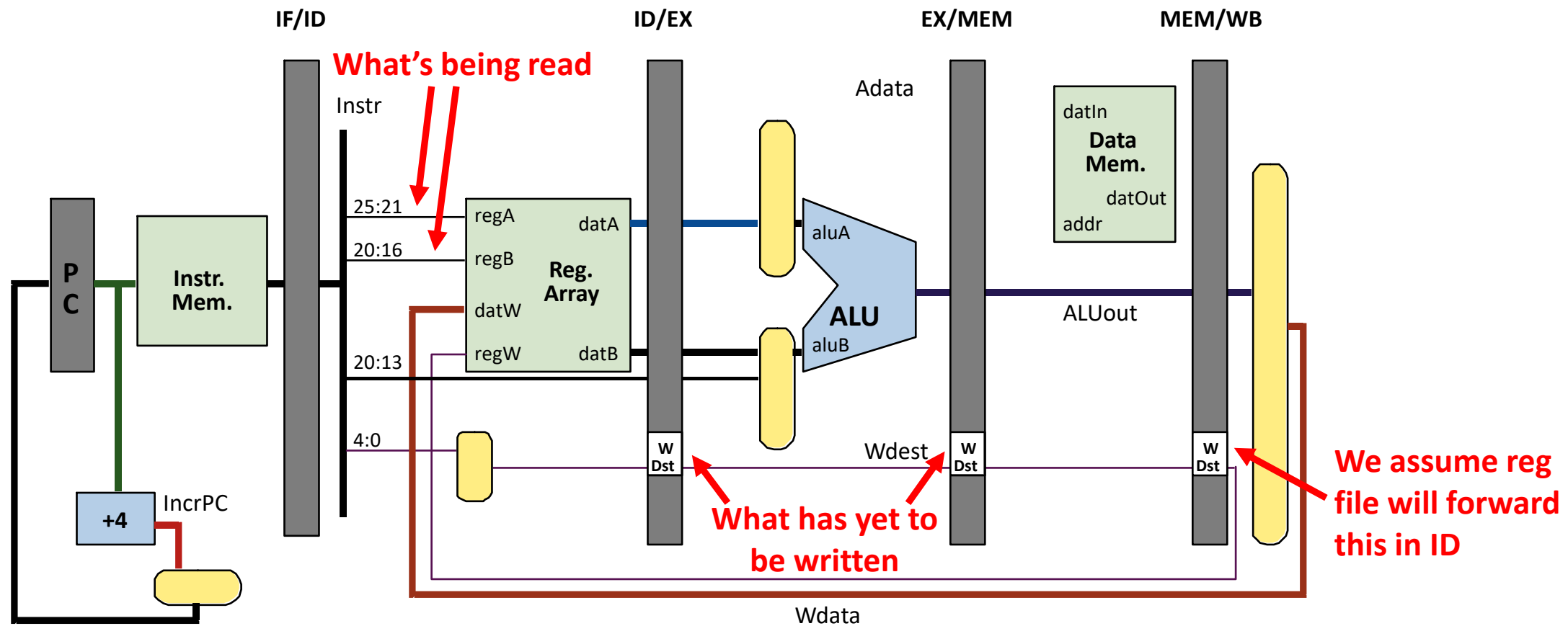
Bubble ID/EX

Transfer EX/MEM, MEM/WB

Effect

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

How do we detect dependencies?

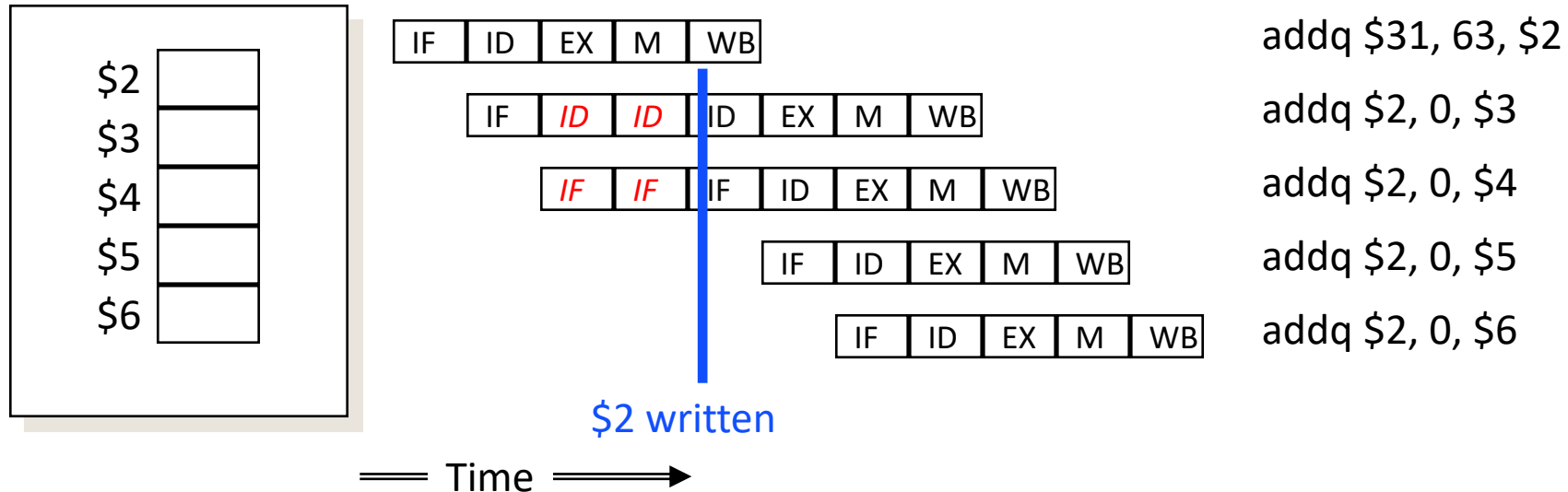


➔ Stall if: $(\text{IF/ID.IR}[25:21] == \text{ID/EX.Wdst}) \ || \ (\text{IF/ID.IR}[25:21] == \text{EX/MEM.Wdst}) \ ||$
 $(\text{IF/ID.IR}[20:16] == \text{ID/EX.Wdst}) \ || \ (\text{IF/ID.IR}[20:16] == \text{EX/MEM.Wdst})$

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

Q: Do we really need to stall?

Forwarding (Bypassing)

Observation: *The data we want is available, but not in the right place!*

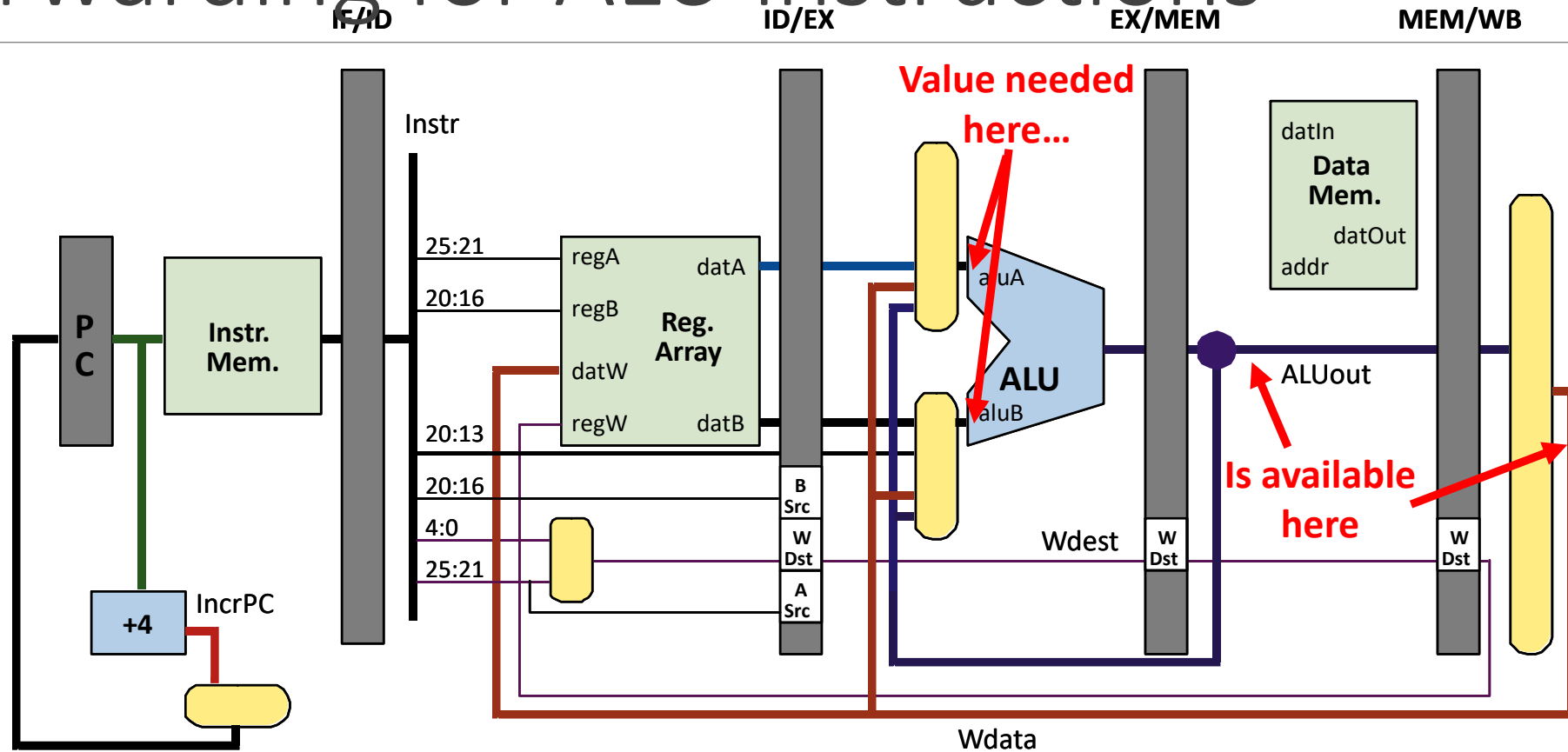
- ALU data generated at end of EX
- ALU data consumed at beginning of EX

➔ *We shouldn't need to stall at all!*

Idea

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

Forwarding for ALU instructions



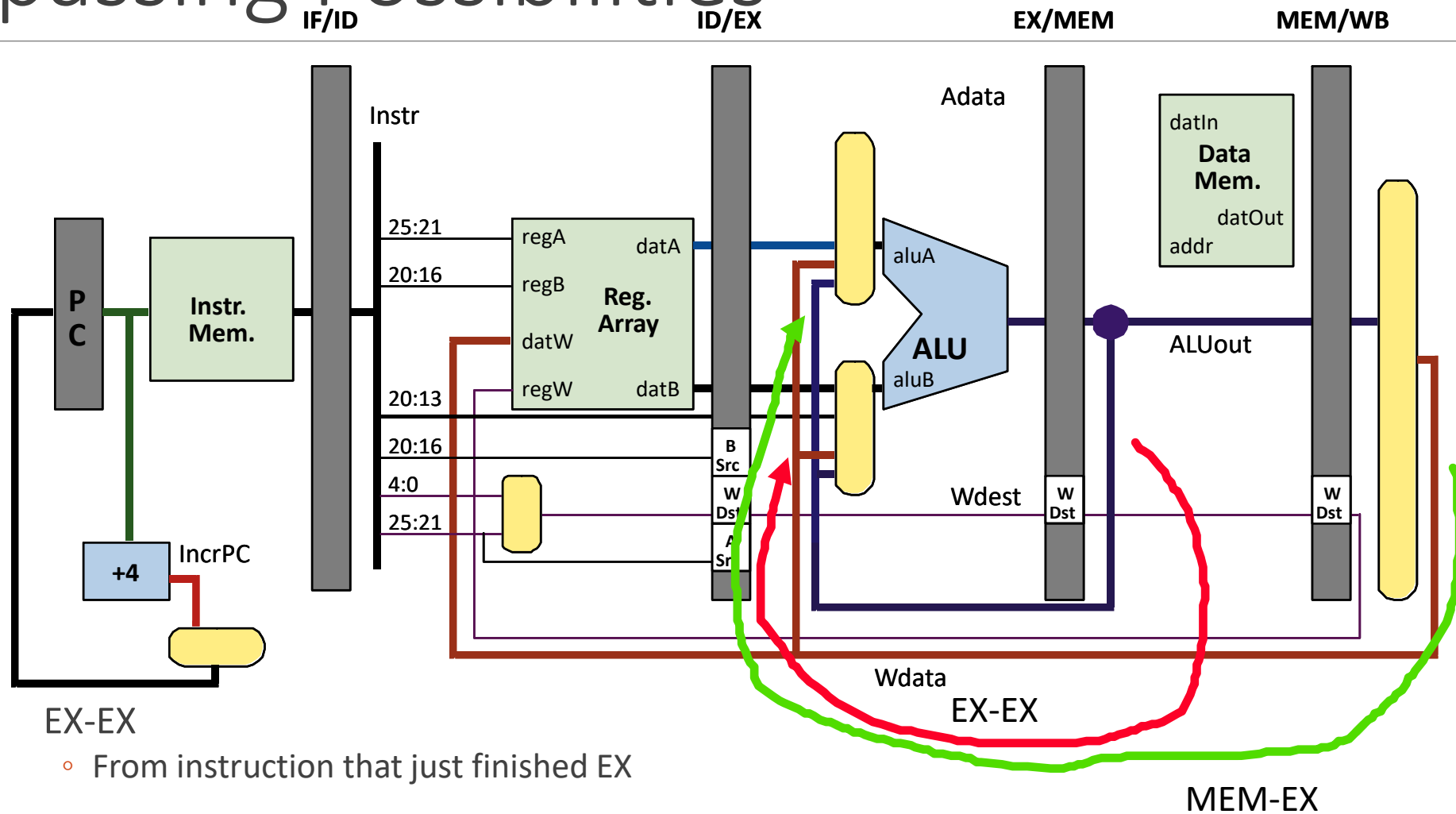
Operand Destinations

- Register ID/EX.ASrc
- Register ID/EX.BSrc

Operand Sources

- EX/MEM.ALUout (pending write to EX/MEM.WDst)
- MEM/WB.ALUout (pending write to MEM/WB.WDst)

Bypassing Possibilities



EX-EX

- From instruction that just finished EX

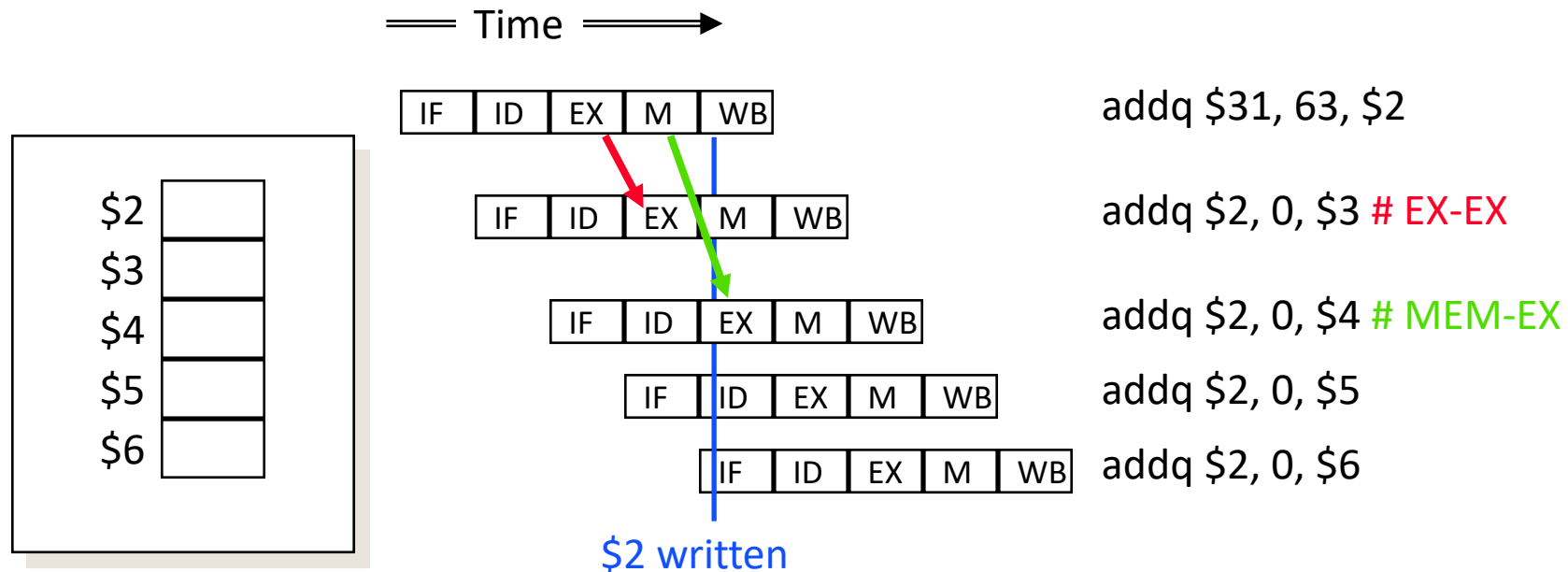
MEM-EX

- 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



Some Hazards with Loads & Stores

Data Generated by Load

Load-Store Data

```
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)
```

*Handled by
memory
system*

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)
```

MEM-MEM Forwarding

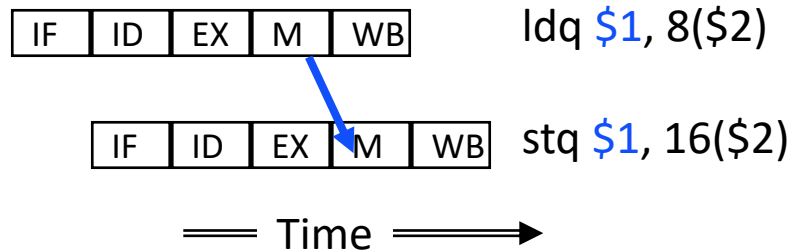
Condition

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

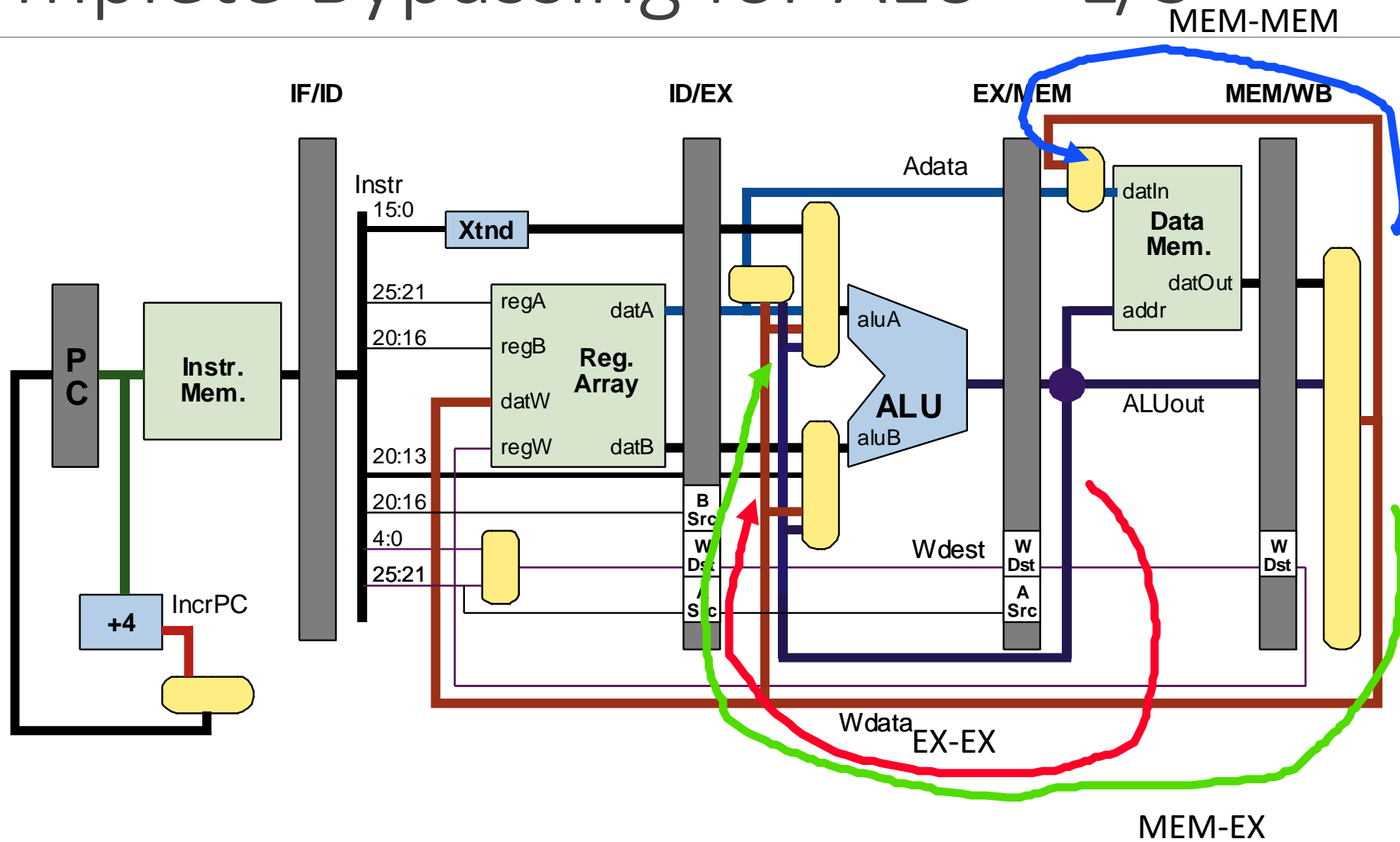
Load-Store Data

ldq \$1, 8(\$2)

stq \$1, 16(\$2)



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)
```

*Handled by
memory
system*

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

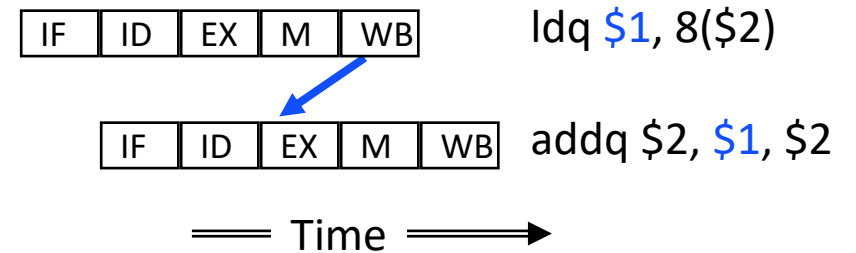
Load-ALU

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

Load-Store (or Load) Addr.

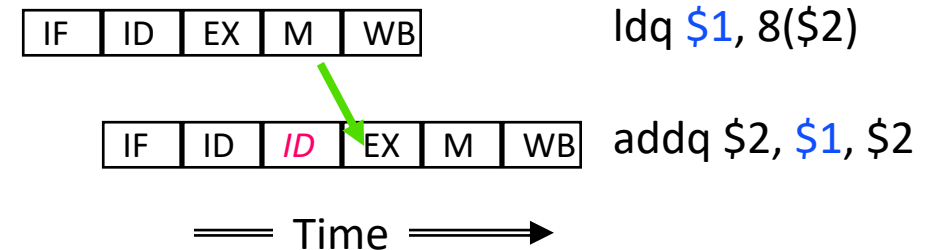
```
ldq $1, 8($2)
stq $2, 16($1)
```

Just Forward?



Value not available soon enough!

With 1 Cycle Stall



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 using EX-EX, MEM-EX

ALU-Branch

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

Distant ALU-Branch

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

Load-Branch

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

Still More Data Hazards

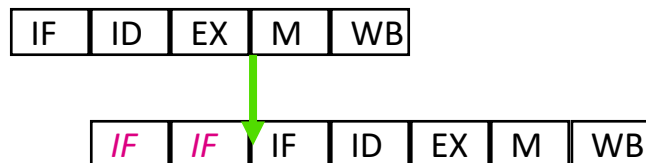
Jump Uses Register Data

- Generated by ALU instruction
- Read from register in ID

Handling

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

...But fetch must stall to handle **control hazard**



ALU-Jump

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

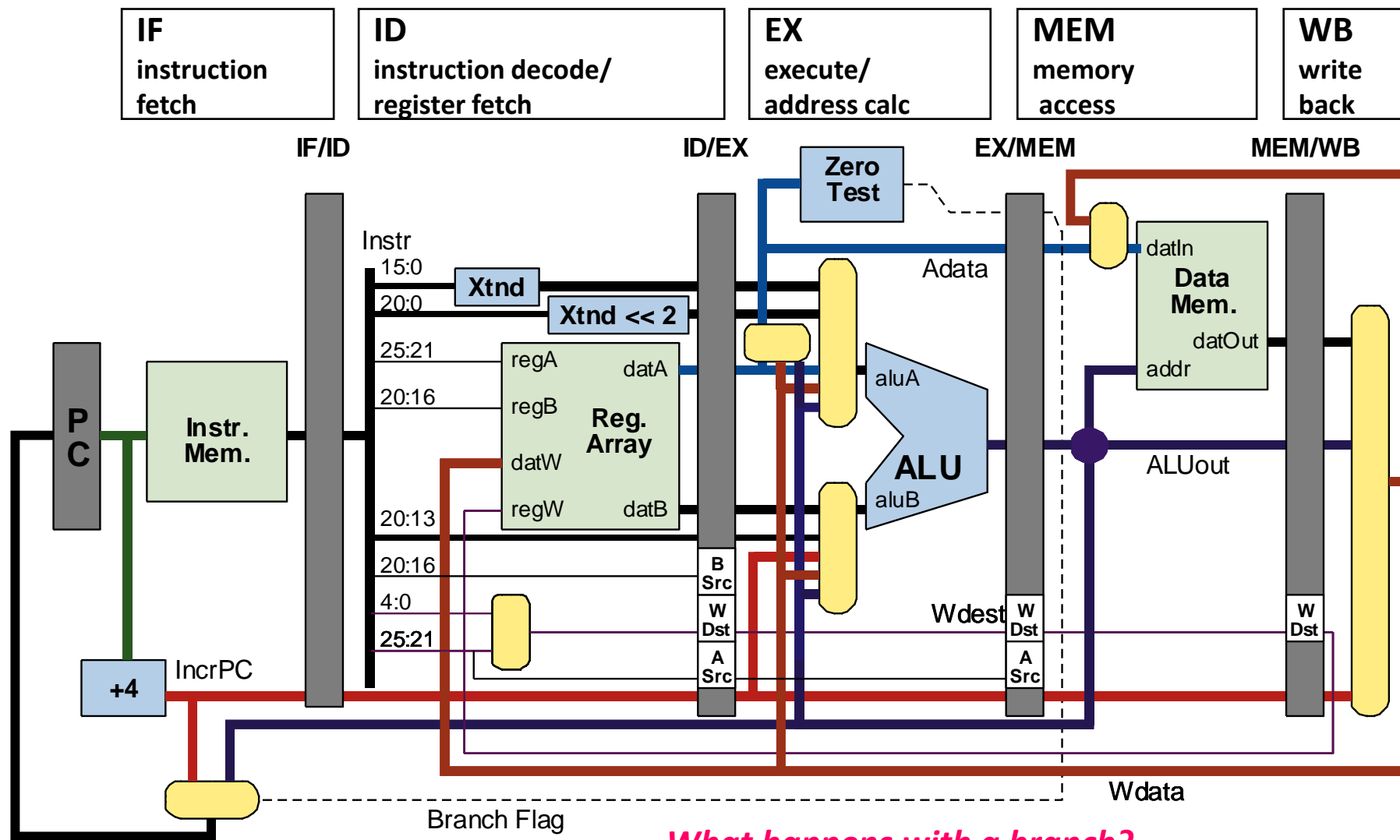
Distant ALU-Jump

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

Load-Jump

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

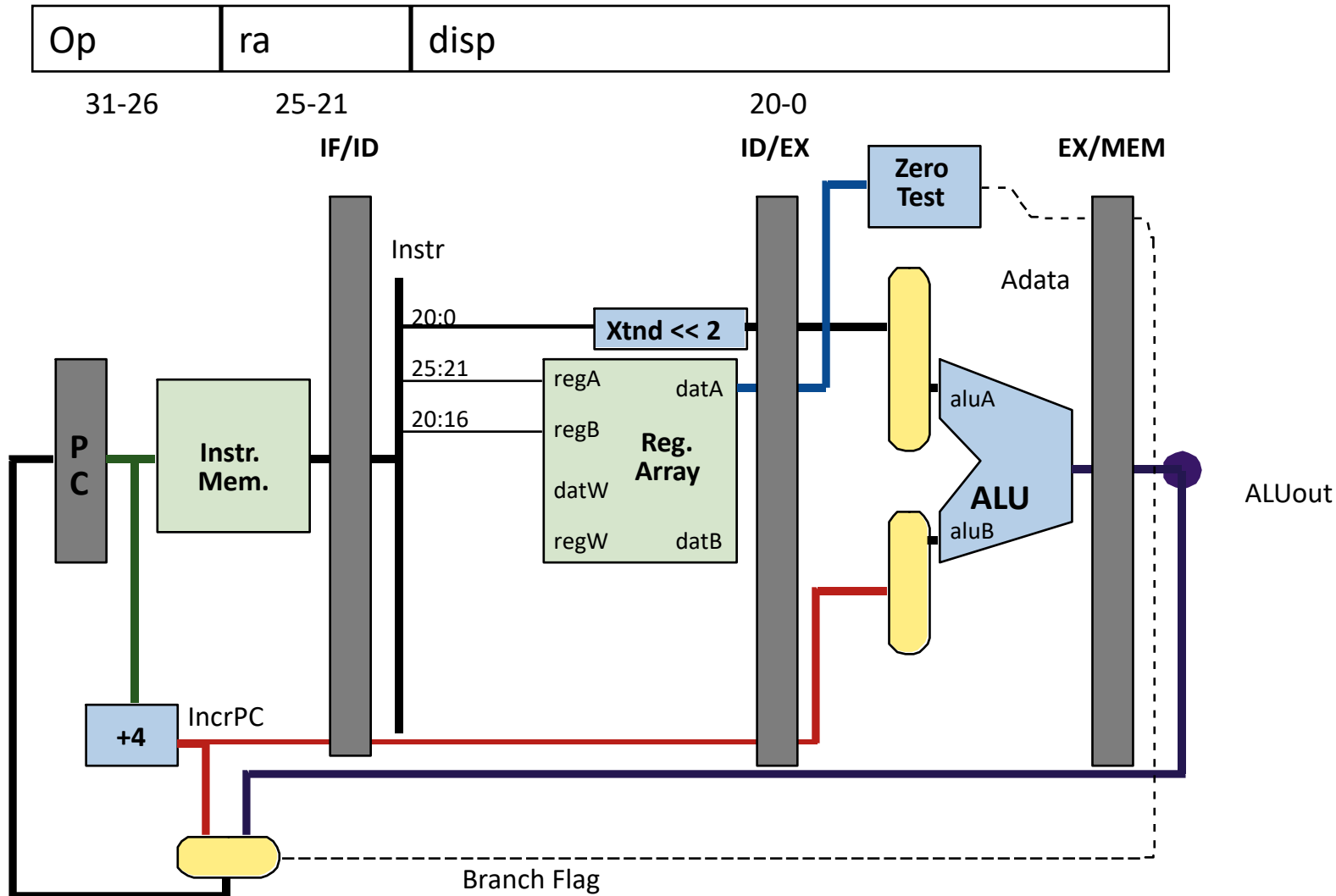
Pipelined Datapath



What happens with a branch?

Conditional Branch Instruction Handling

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



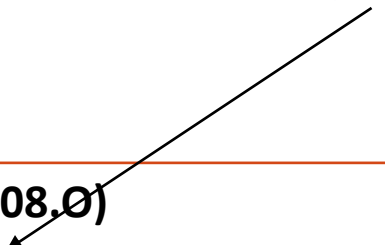
Branch Example

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)



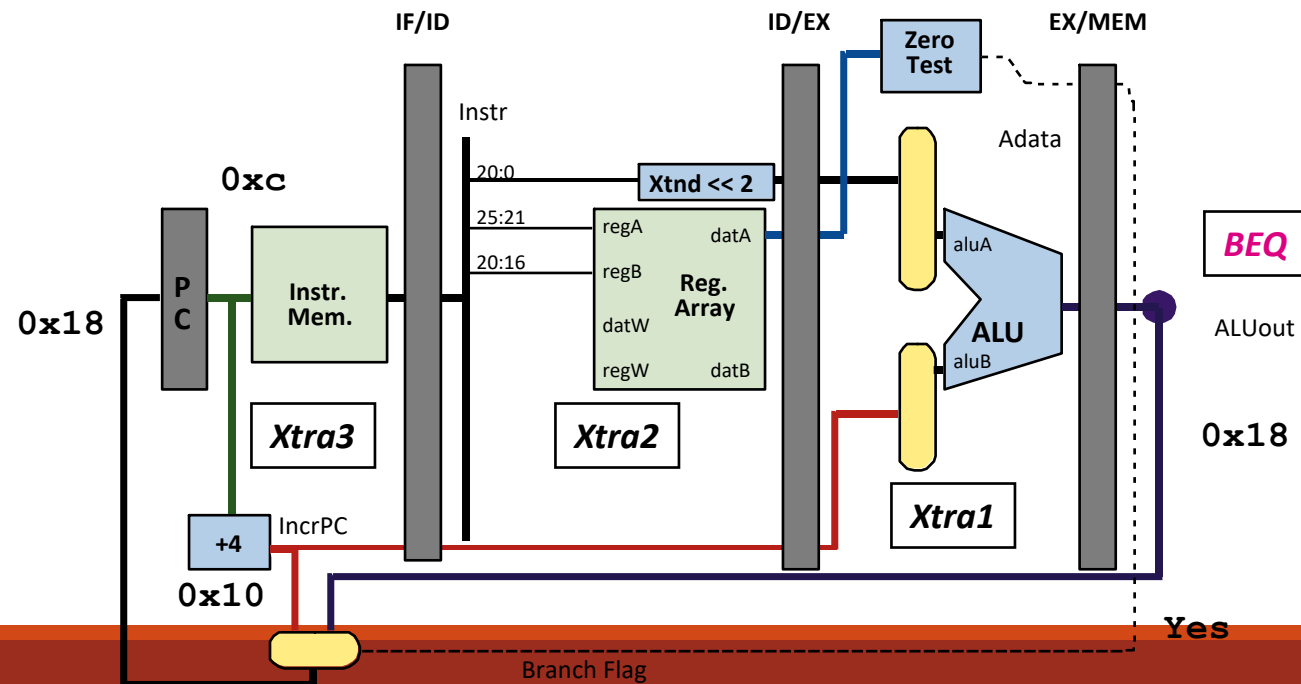
```
0x0:  e7e00005 beq    r31, 0x18    # Take
0x4:  43e7f401 addq   r31, 0x3f, r1   # (Skip)
0x8:  43e7f402 addq   r31, 0x3f, r2   # (Skip)
0xc:  43e7f403 addq   r31, 0x3f, r3   # (Skip)
0x10: 43e7f404 addq   r31, 0x3f, r4   # (Skip)
0x14: 47ff041f bis     r31, r31, r31
0x18: 43e7f405 addq   r31, 0x3f,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

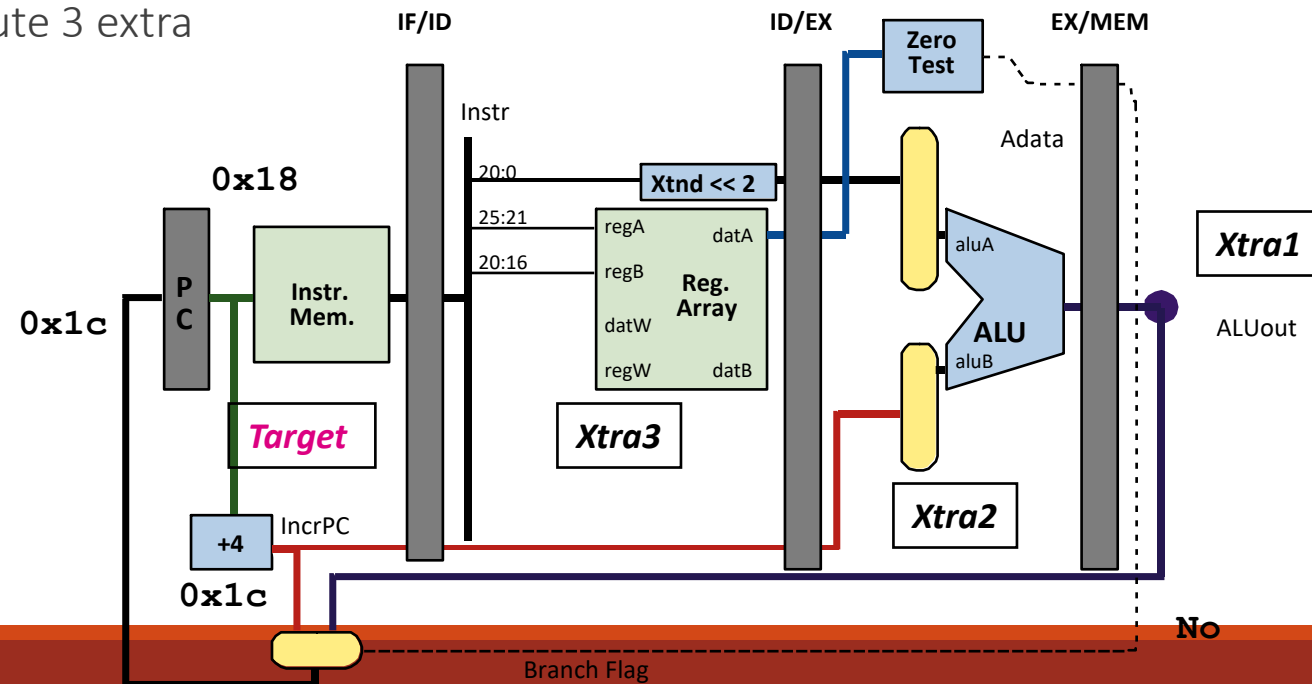


Branch Hazard Example

```
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
```

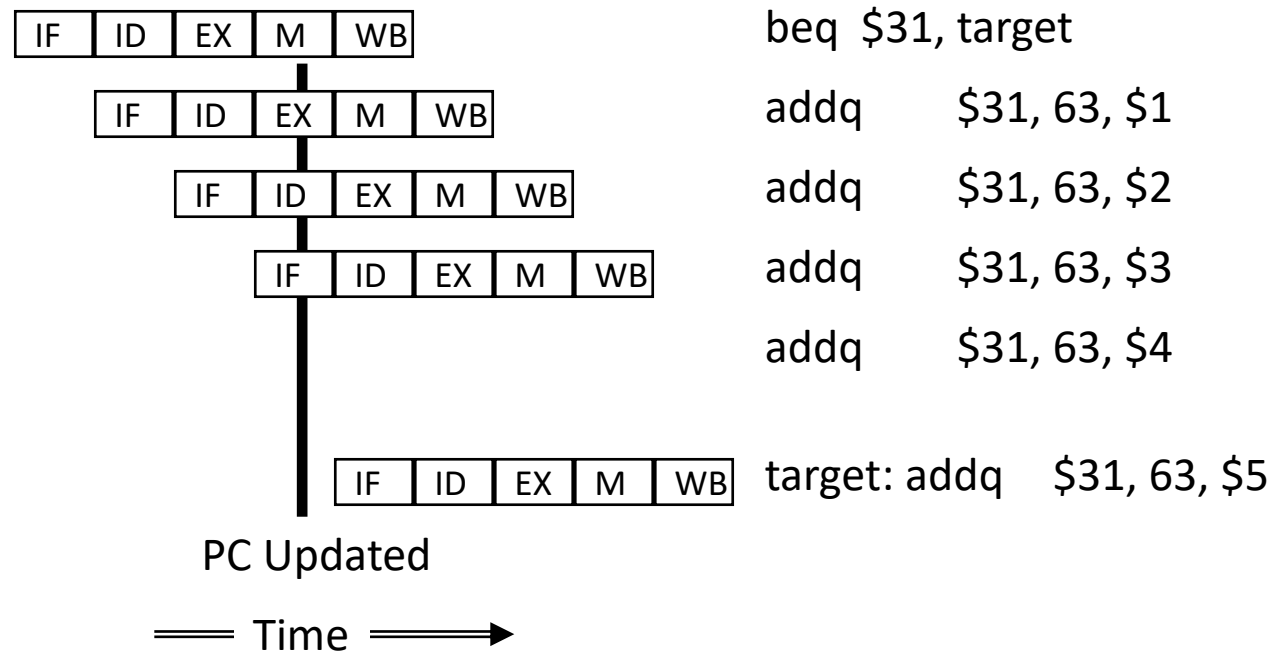
- One cycle later
- Problem: Will execute 3 extra instructions!



Branch Hazard Pipeline Diagram

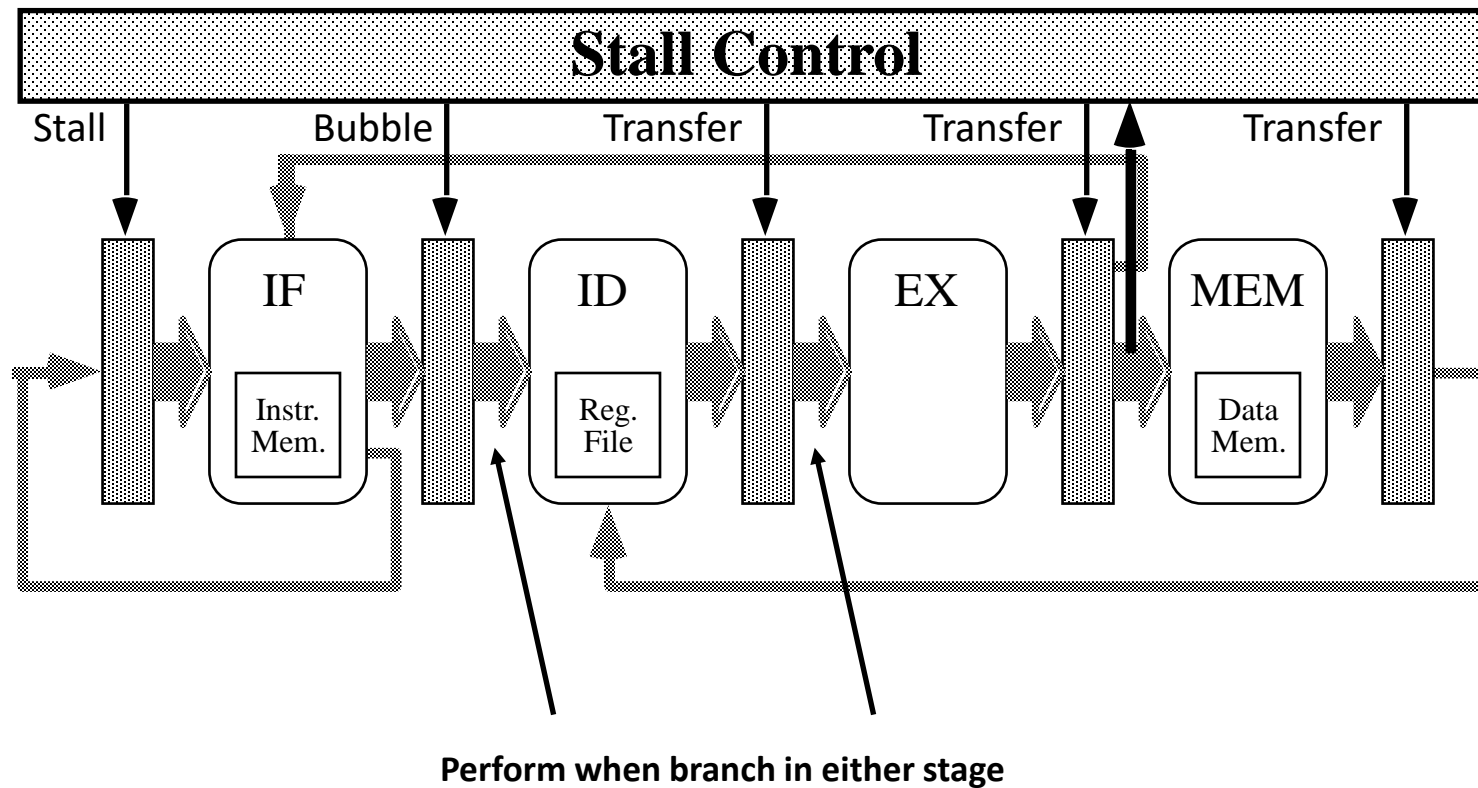
Problem

- Instruction fetched in IF, branch condition set in MEM



Stall Until Resolved Branch

- Detect when branch in stages ID or EX
- Stall fetch
 - Stall PC, bubble IF/ID, transfer others



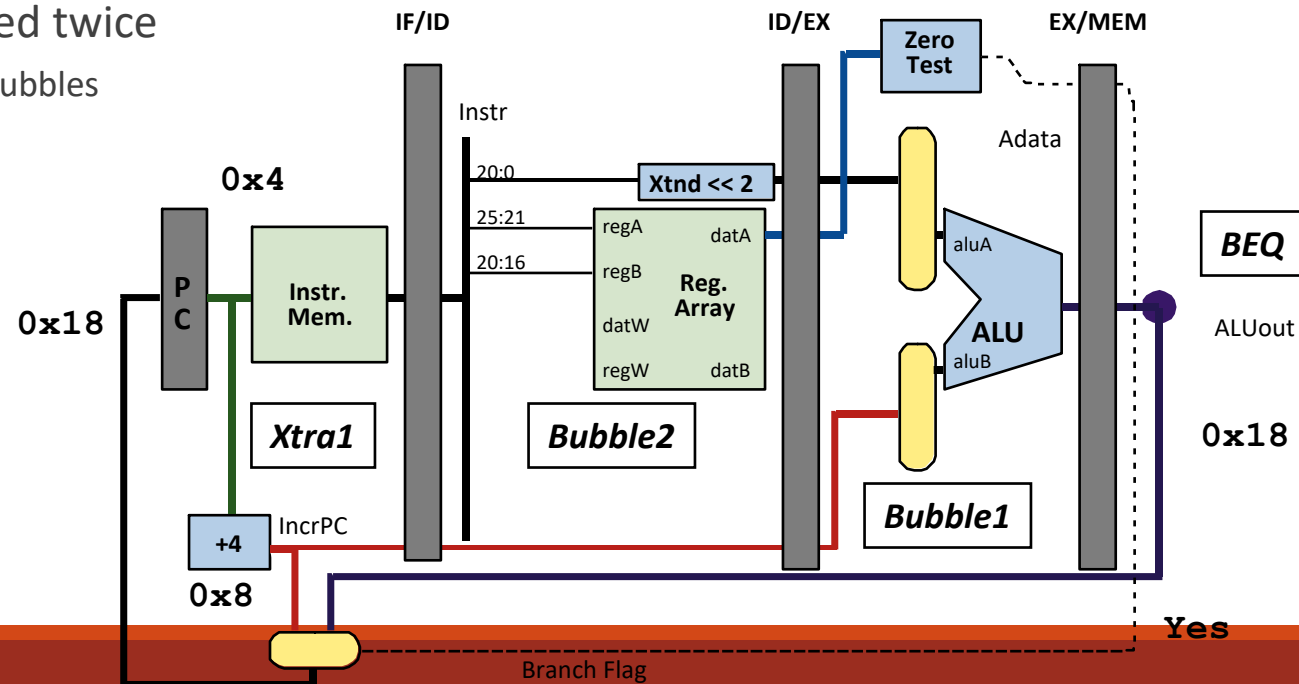
Stalling Branch Example

```

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

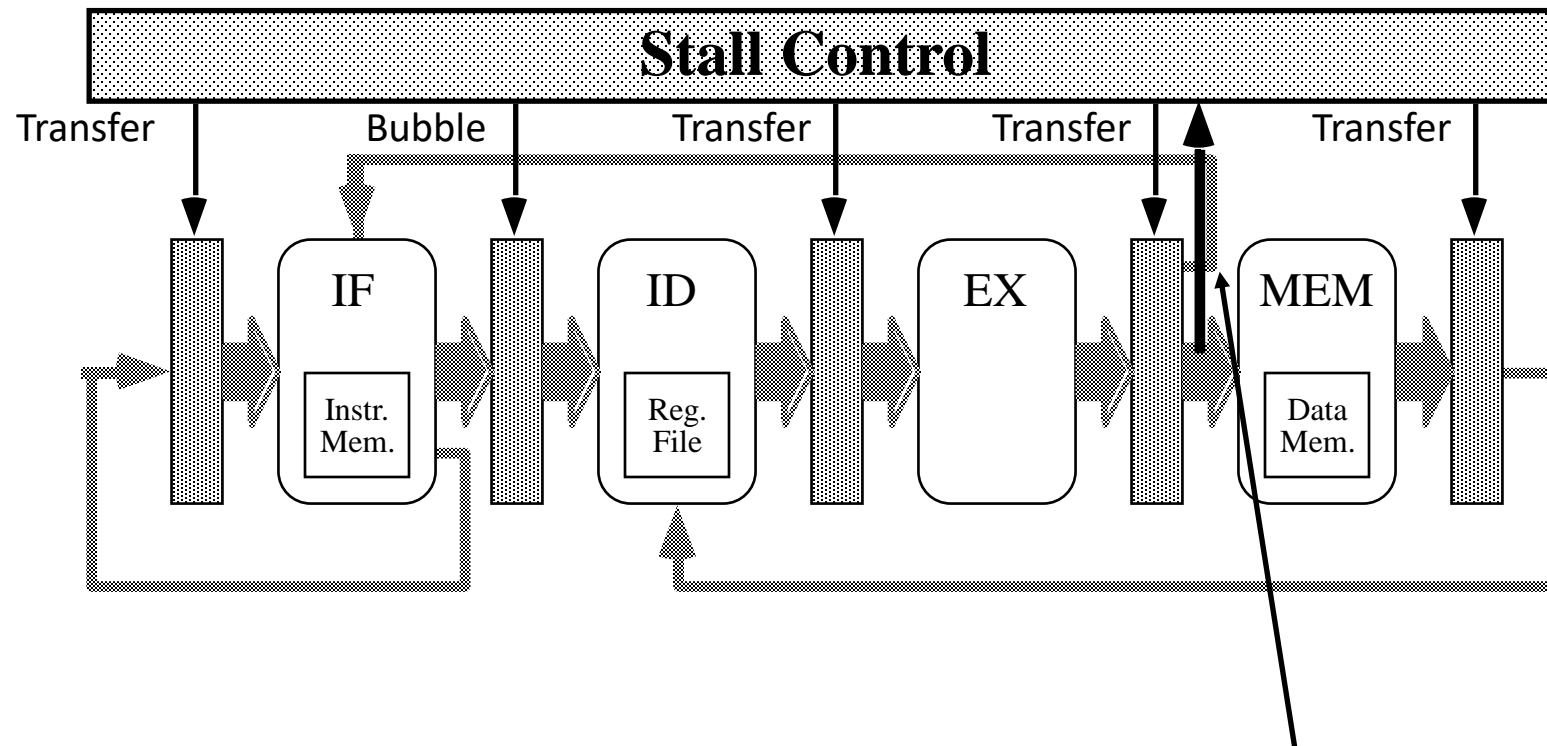
0x18: addq     r31, 0x3f, r5   # Target
    
```

- 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



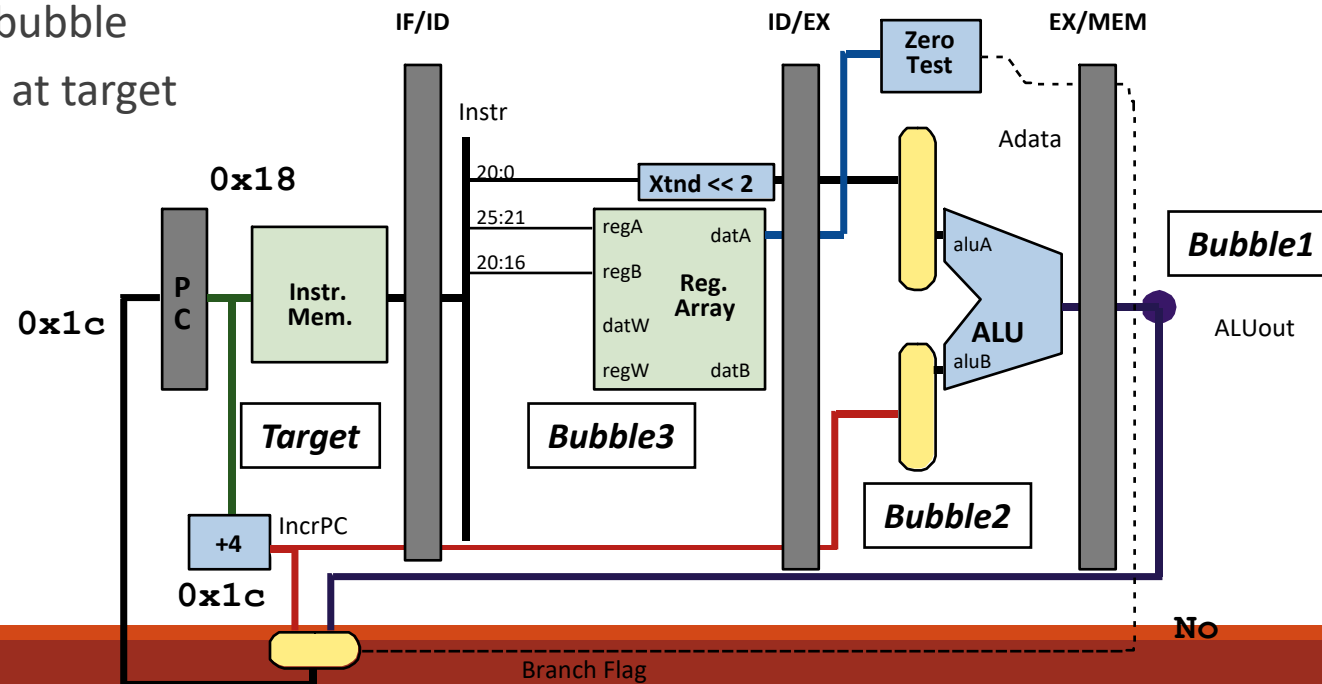
Perform when detect taken branch

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
```

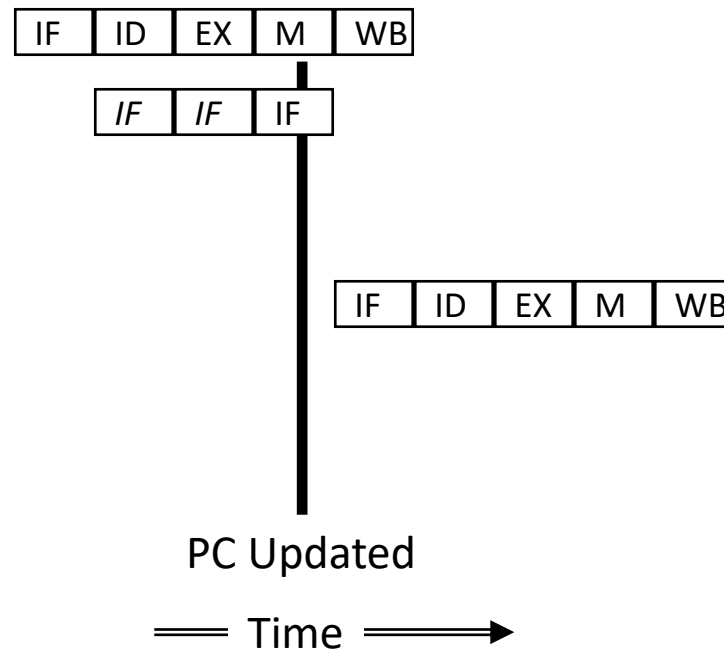
- When branch taken
- Generate 3rd bubble
- Begin fetching at target



Taken Branch Pipeline Diagram

Behavior

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



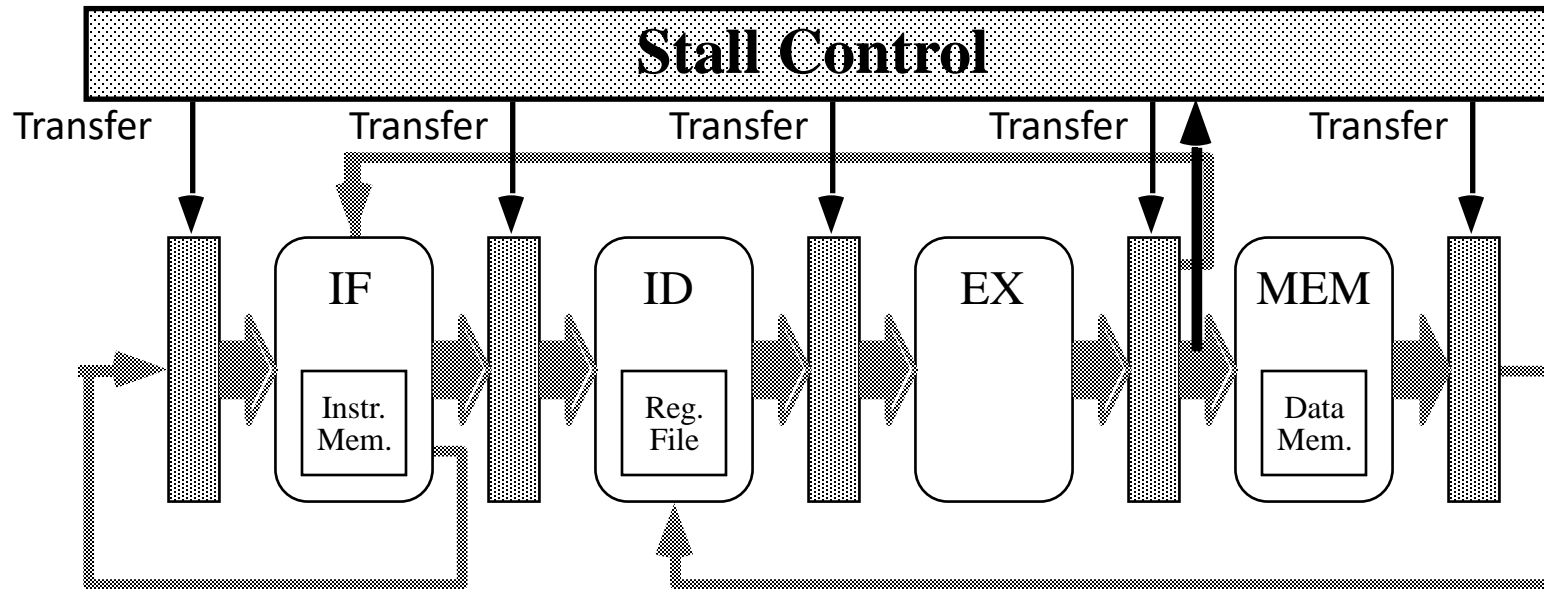
beq \$31, target

addq \$31, 63, \$1 # Xtra1

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

Not Taken Branch Resolution

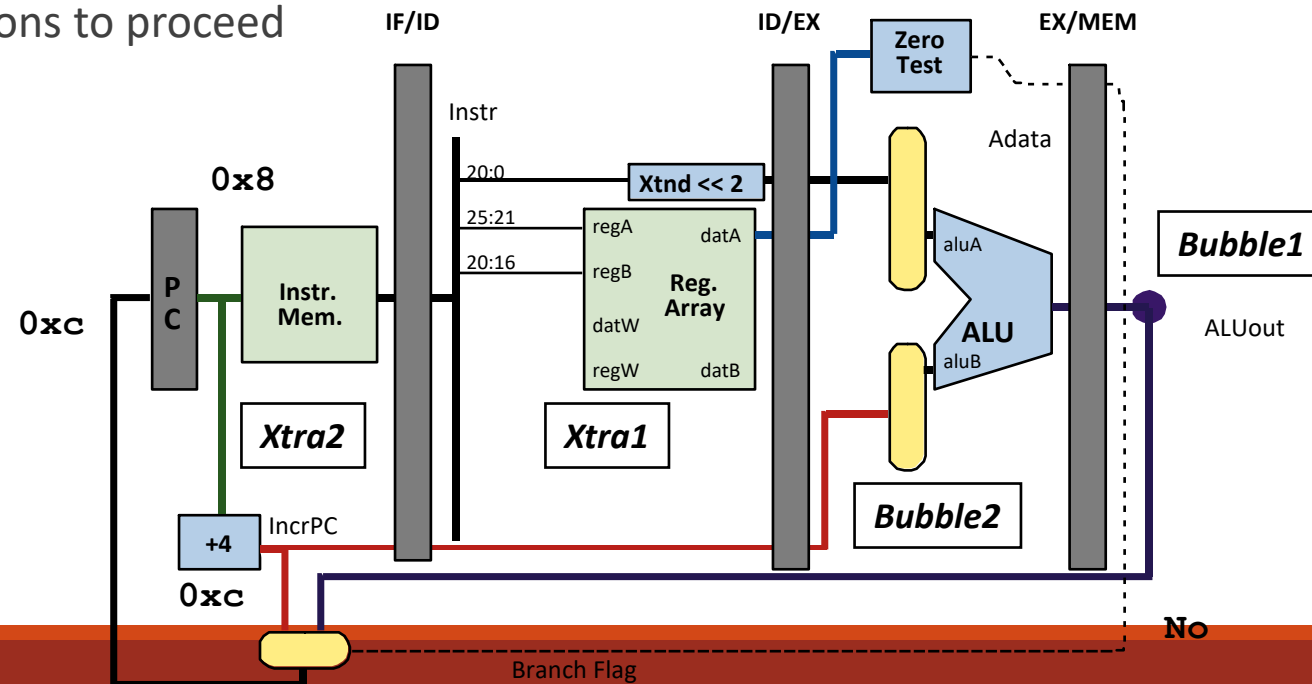
- [Stall two cycles with not-taken branches as well]
- When branch not taken, already have instruction Xtra1 in pipe
- Let it proceed as usual



Not Taken Branch Resolution Example

```
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
```

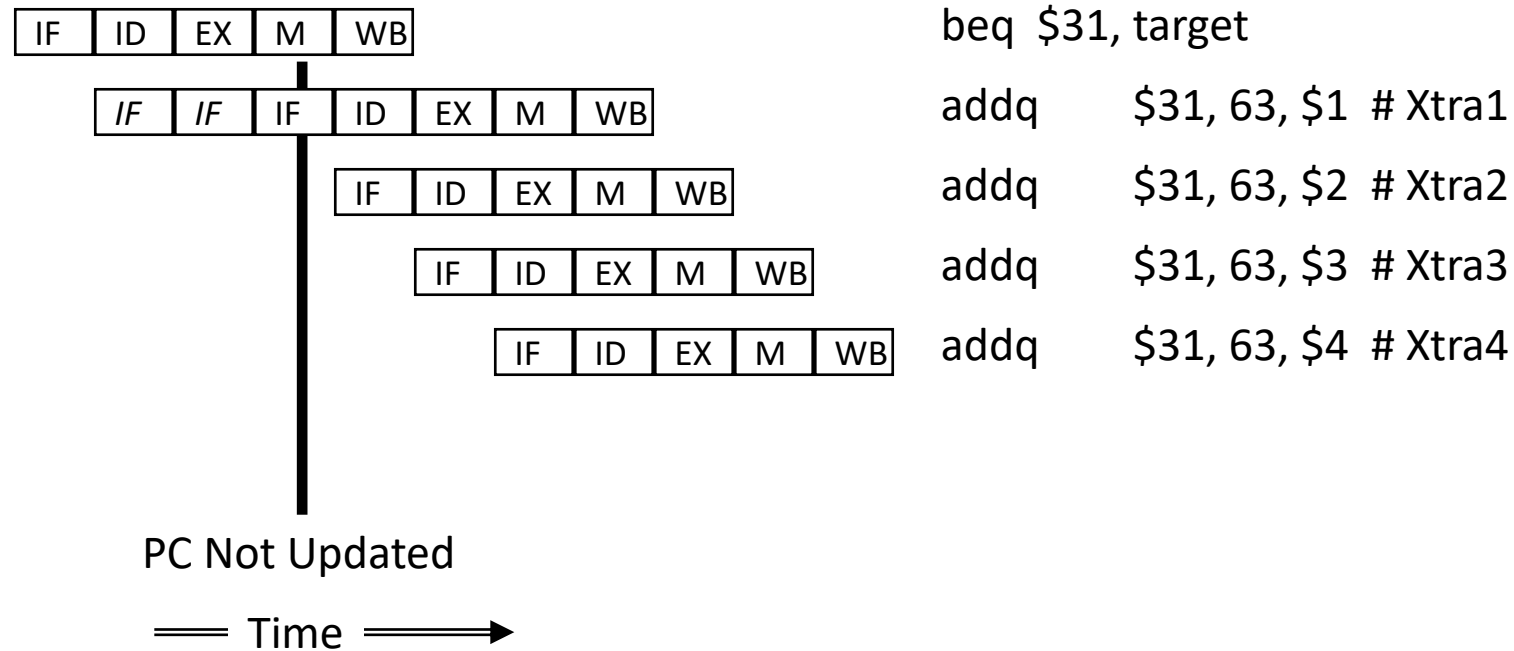
- Branch not taken
- Allow instructions to proceed



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

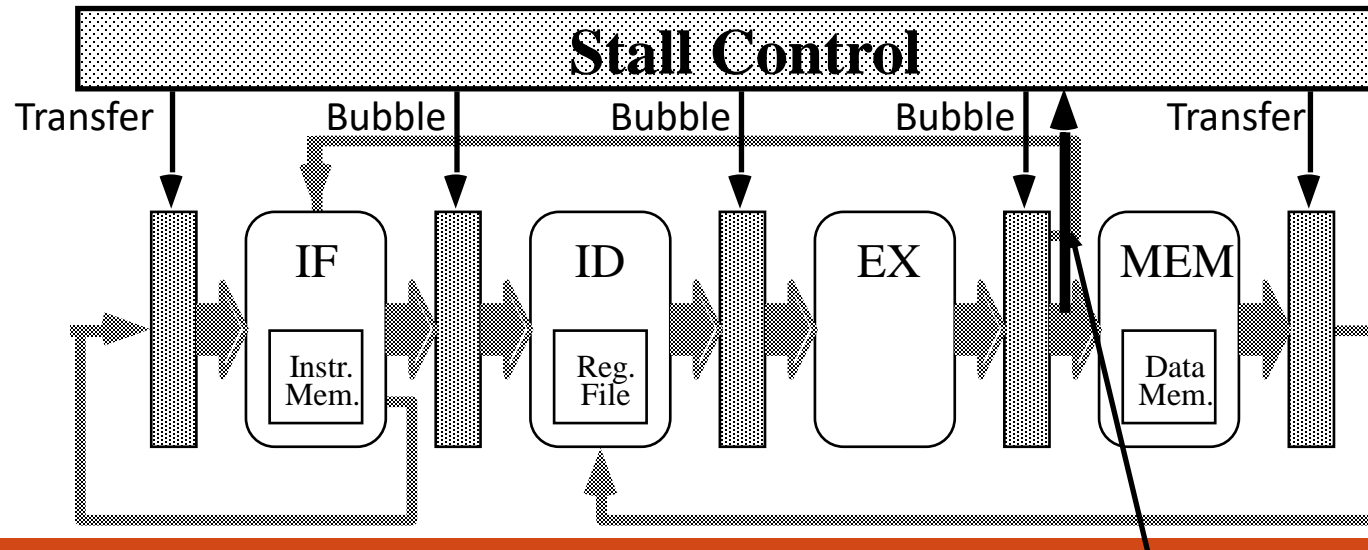
Instruction does not cause any updates until MEM or WB stages!

Instruction can be “cancelled” from pipe up through EX stage

- Replace with bubble

Strategy

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



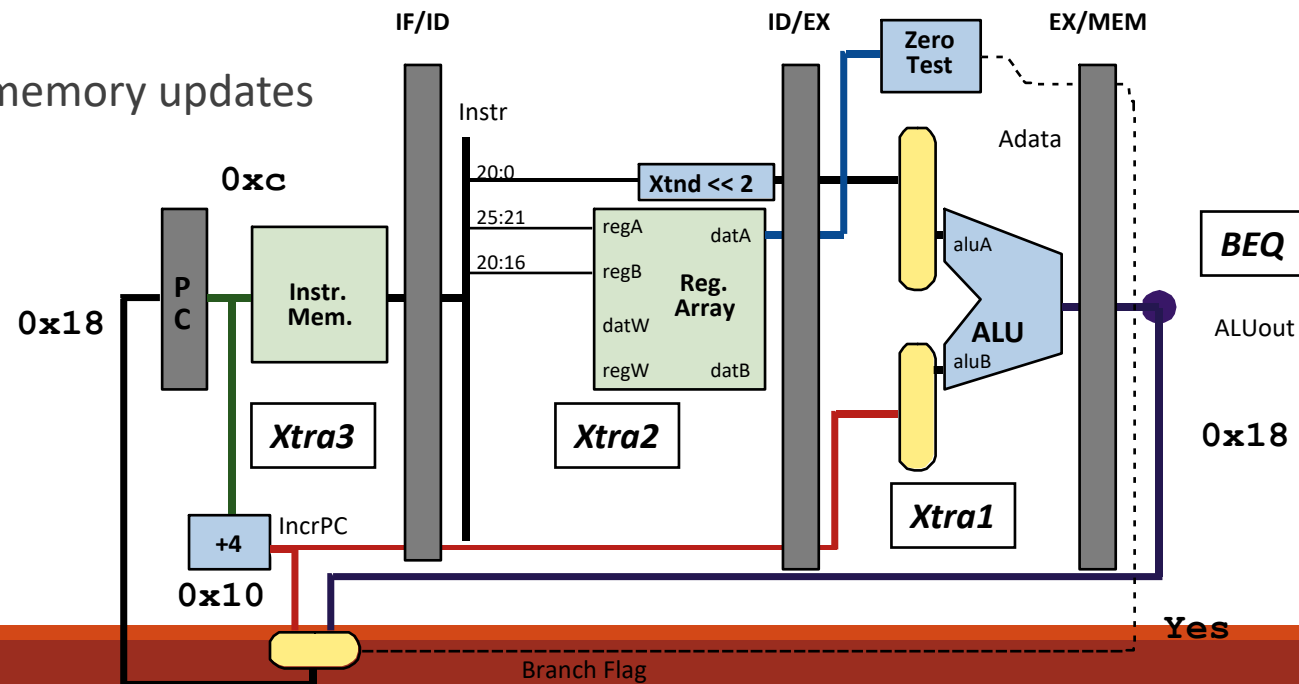
Perform when detect taken branch

Fetch & Cancel Example

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

```
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
```

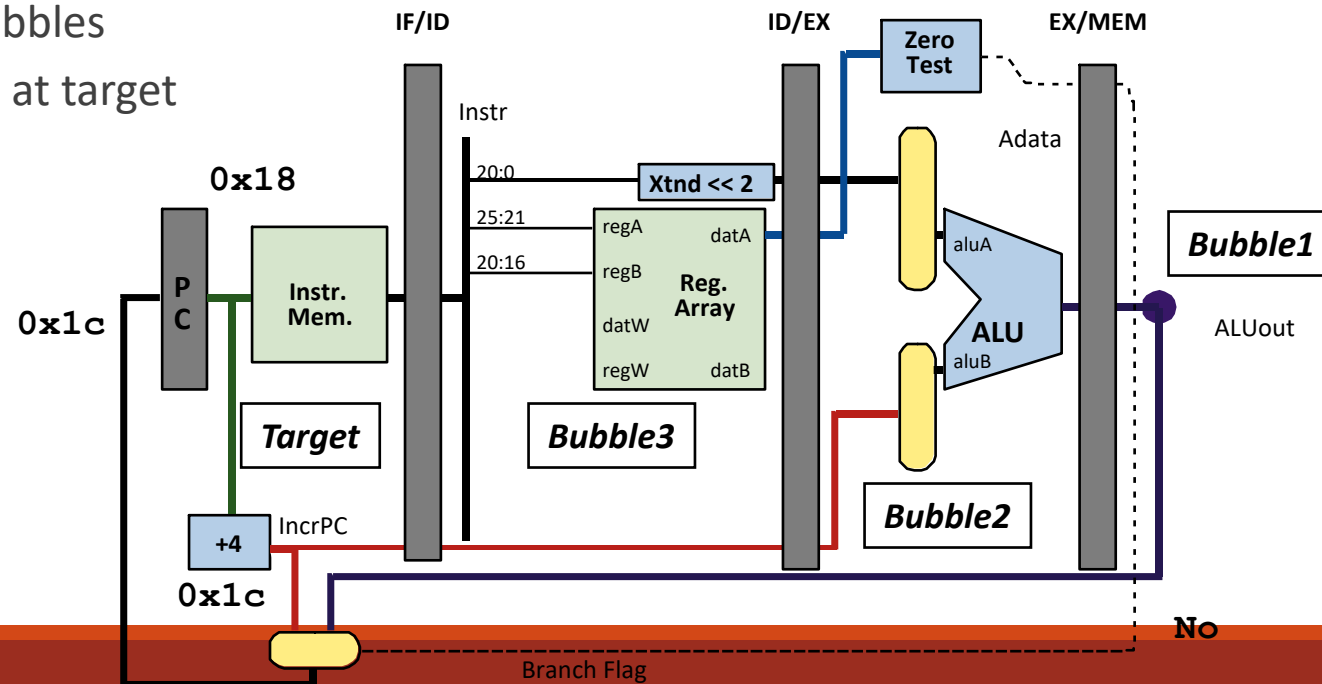


Cancelling Branch Resolution Example

```
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
```

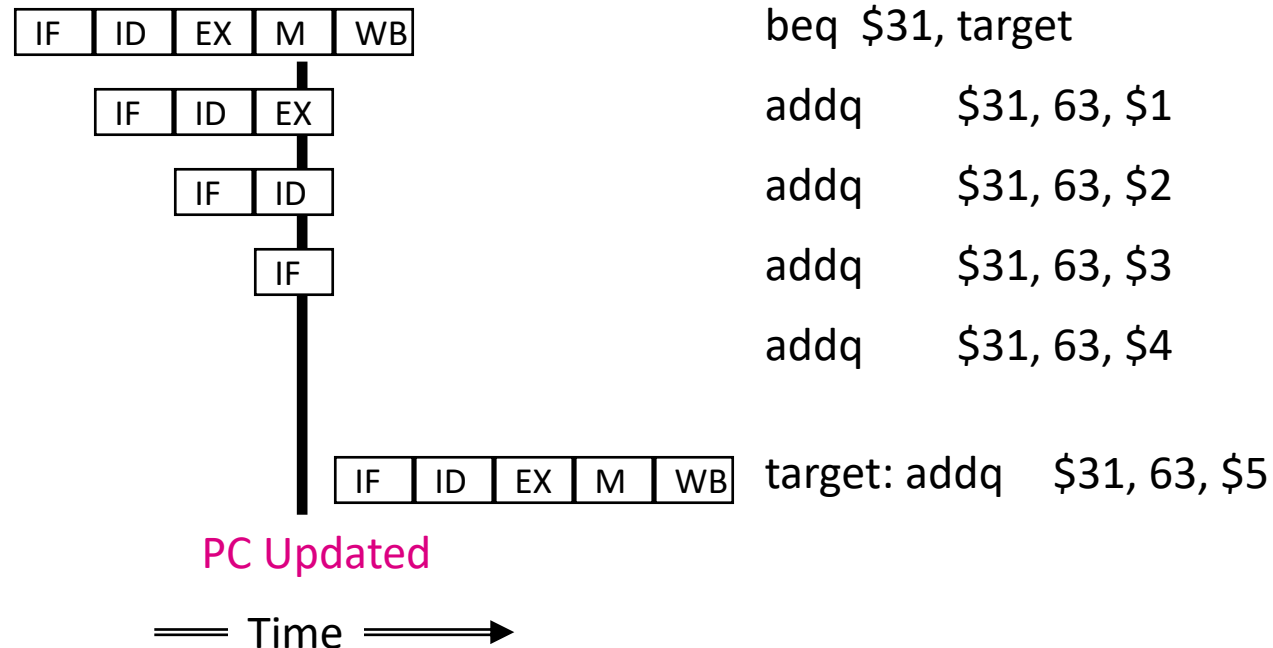
- When branch taken
- Generate 3 bubbles
- Begin fetching at target



Cancelling Branch Pipeline Diagram

Operation

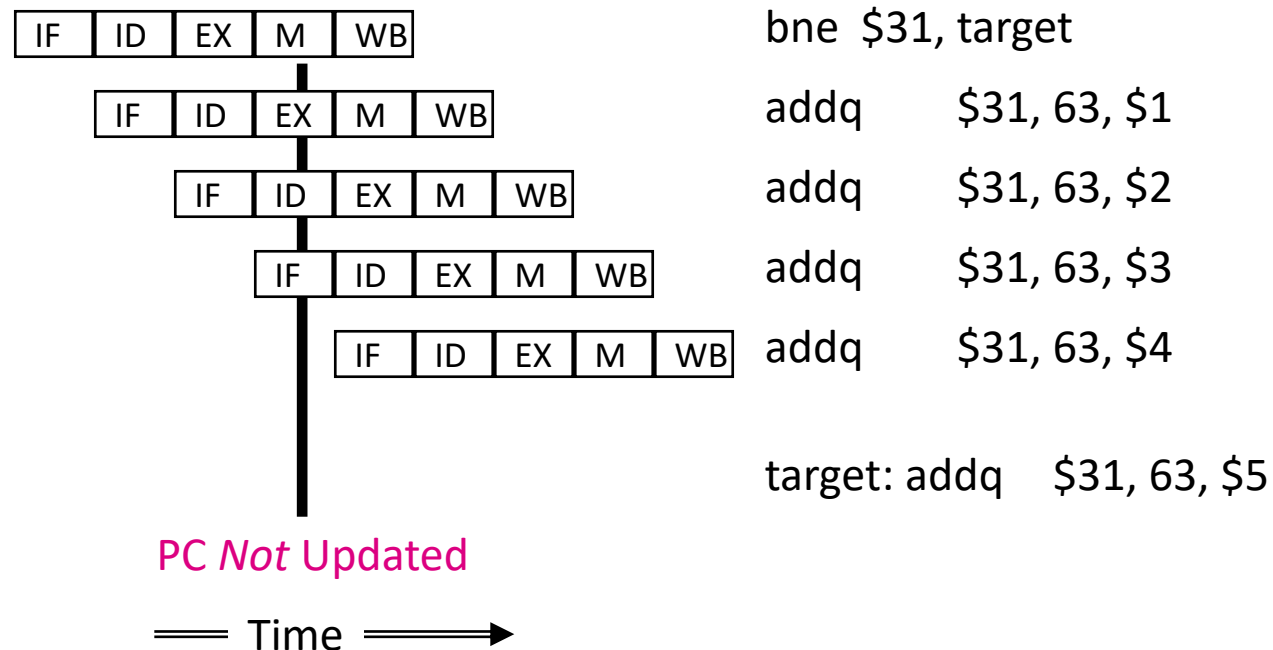
- Process instructions assuming branch will not be taken
- When *is* taken, cancel 3 following instructions



Non-cancelling Branch Pipeline Diagram

Operation

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



Fetch & Cancel Analysis

We have implemented a “static not taken” branch predictor

- 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
 - 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

Self-check questions

How does pipelining improve performance?

What are the causes of non-ideal speedup from pipelining?

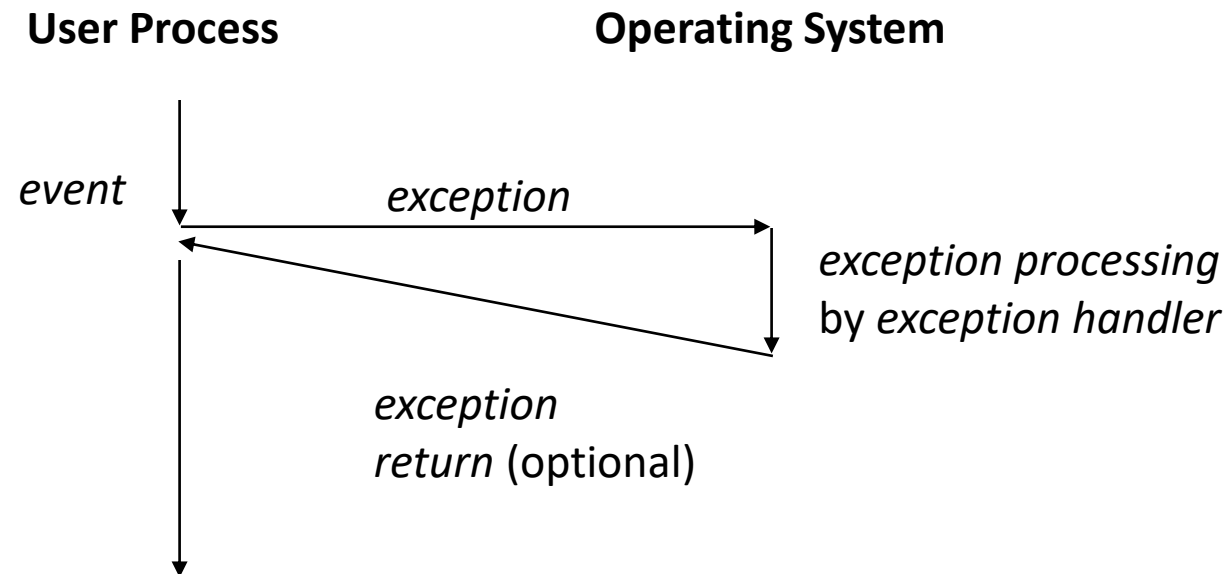
What would happen if the instruction memory took two pipeline stages to return the instruction word? Draw a pipeline diagram showing the impact on performance.

How would we deal with a long-latency operation (e.g., cache miss) in this pipeline?

Exceptions

Exceptions

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



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 (ie, syscall)

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)

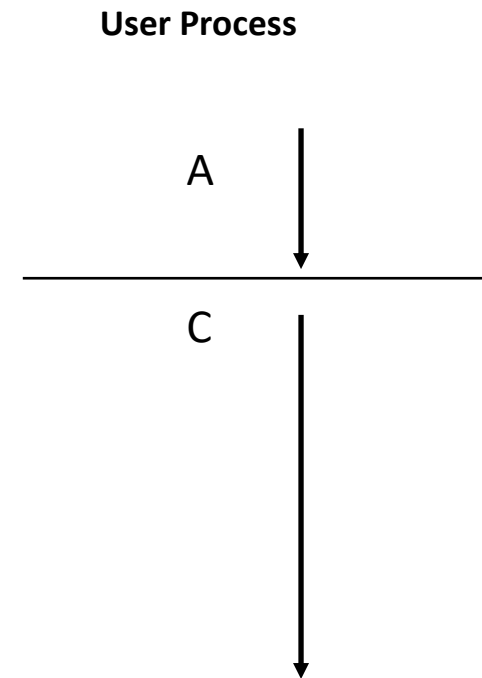
Recognize event(s)

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*”



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

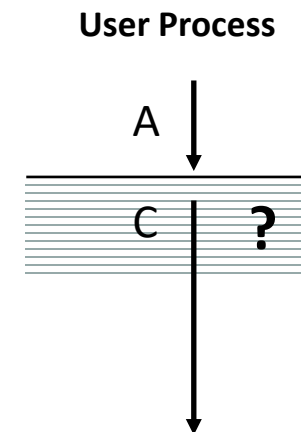
Imprecise exceptions:

- 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?

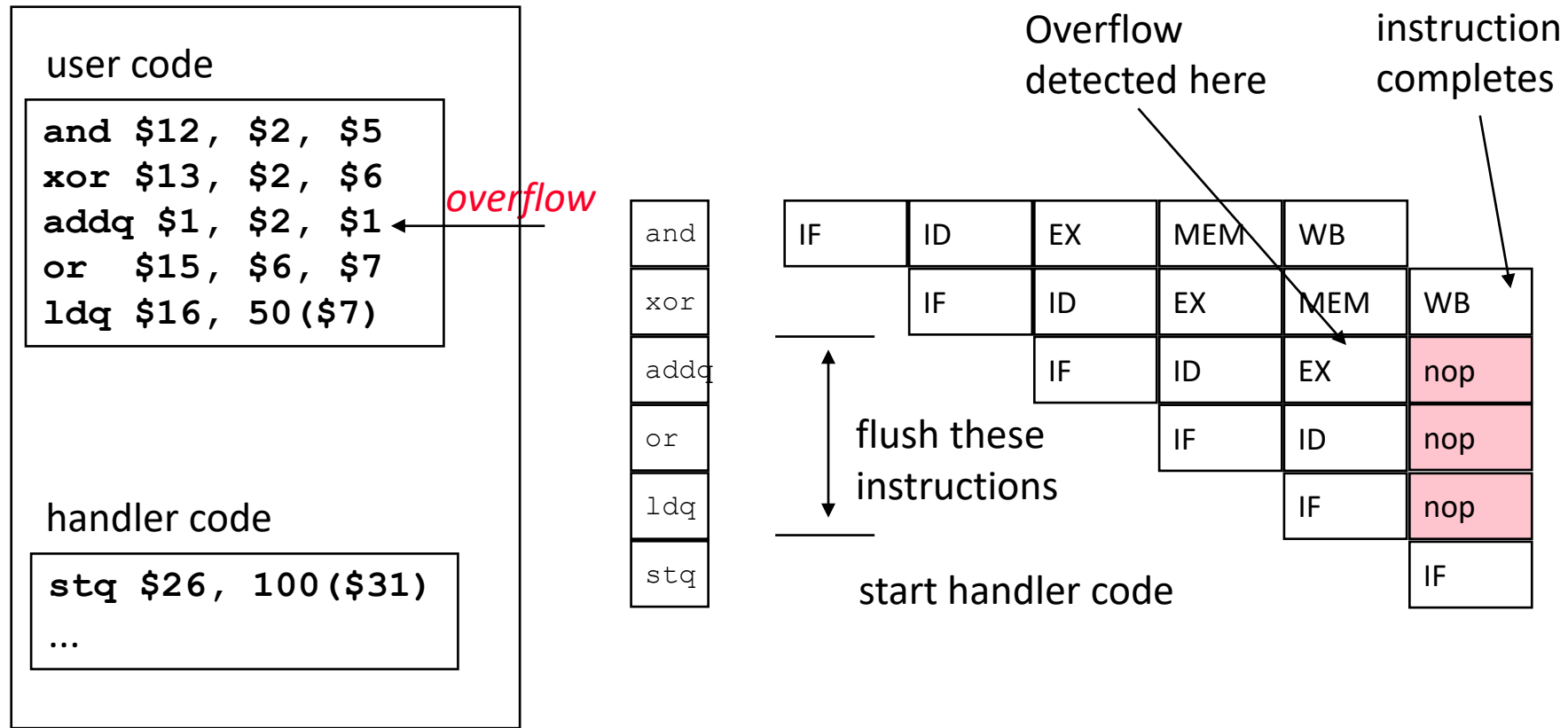
- insert a TRAPB (trap barrier) instruction immediately after
 - stalls until certain that no earlier insts take exceptions

Today systems generally provide precise exceptions to make it easier to reason about program behavior (see: dataflow debugging).



Example: Integer Overflow

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



Multi-Cycle Instructions

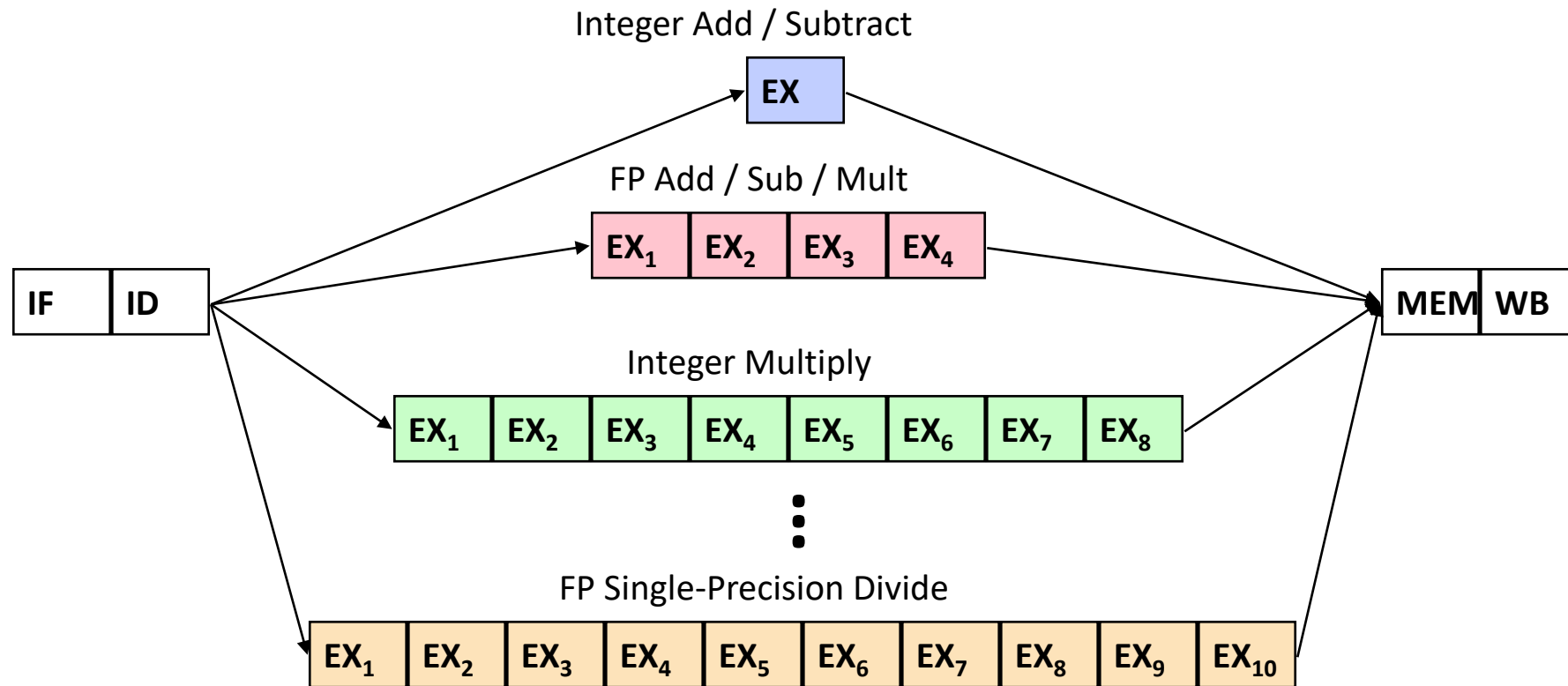
Multicycle instructions

Alpha 21264 Execution Times:

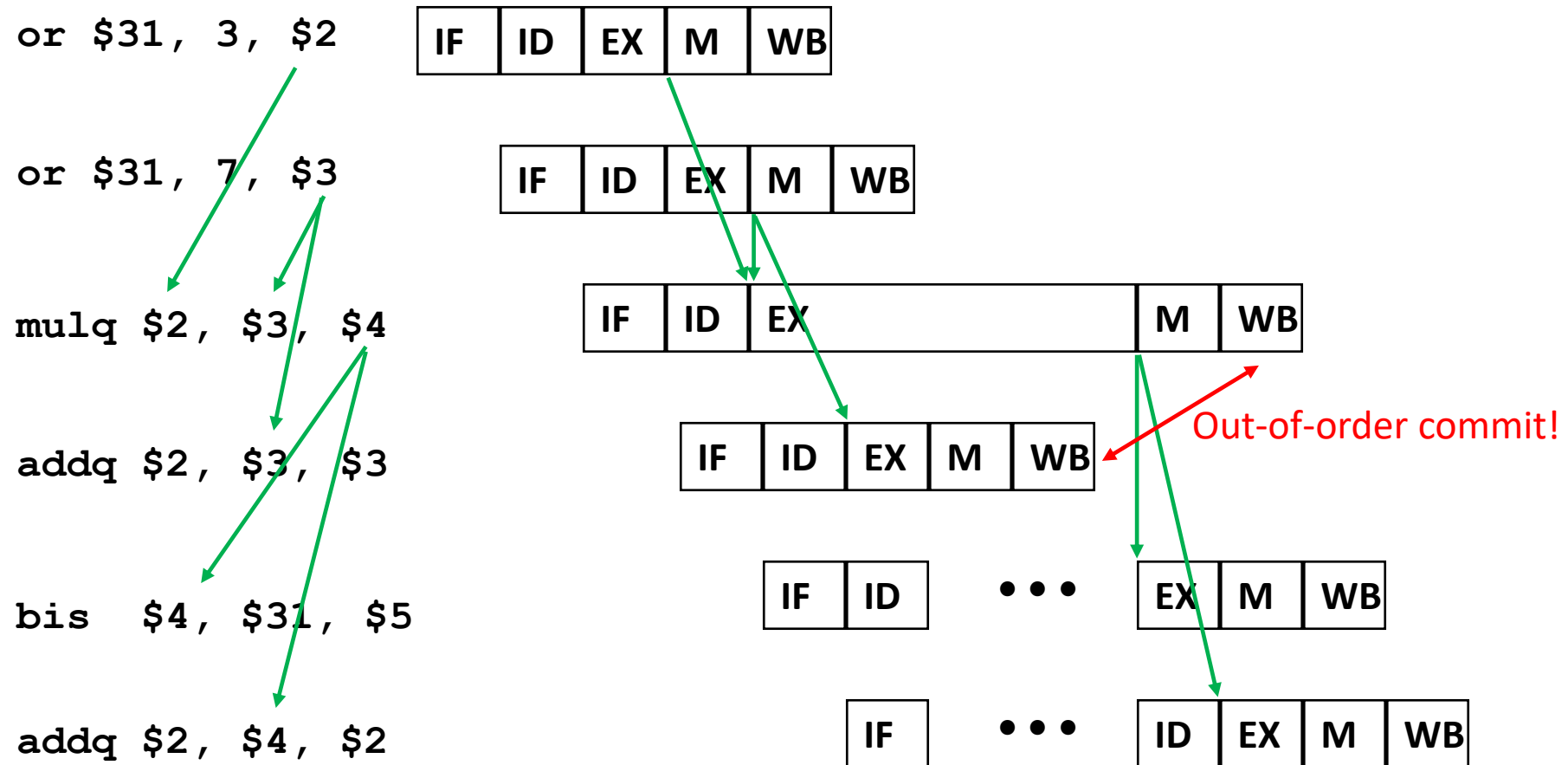
Operation	Integer	FP-Single	FP-Double
add / sub	1	4	4
multiply	8-16	4	4
divide	N / A	10	23

Also: load cache misses

Pipeline Revisited



Multiply Timing Example



Scoreboarding

Simple bookkeeping structure for in-order, multi-cycle issue

- For each register, tracks when value will be available

Issue logic...

- Checks when source registers will be ready & stalls **as long as forwarding paths require**
- Updates ready time of destination register based on op latency

Alternatively

- Scoreboard keeps single “ready bit”
- Pipeline tracks destination register, sets ready bit as early as possible

Scoreboard enables **stall-on-use** to deal with variable latency ops

- E.g., load promotion

Register	Available
R1	<time>
R2	...
R3	
R4	
...	
R31	

Multicycle Instructions Discussion

Pipeline Characteristics for Multi-cycle Instructions

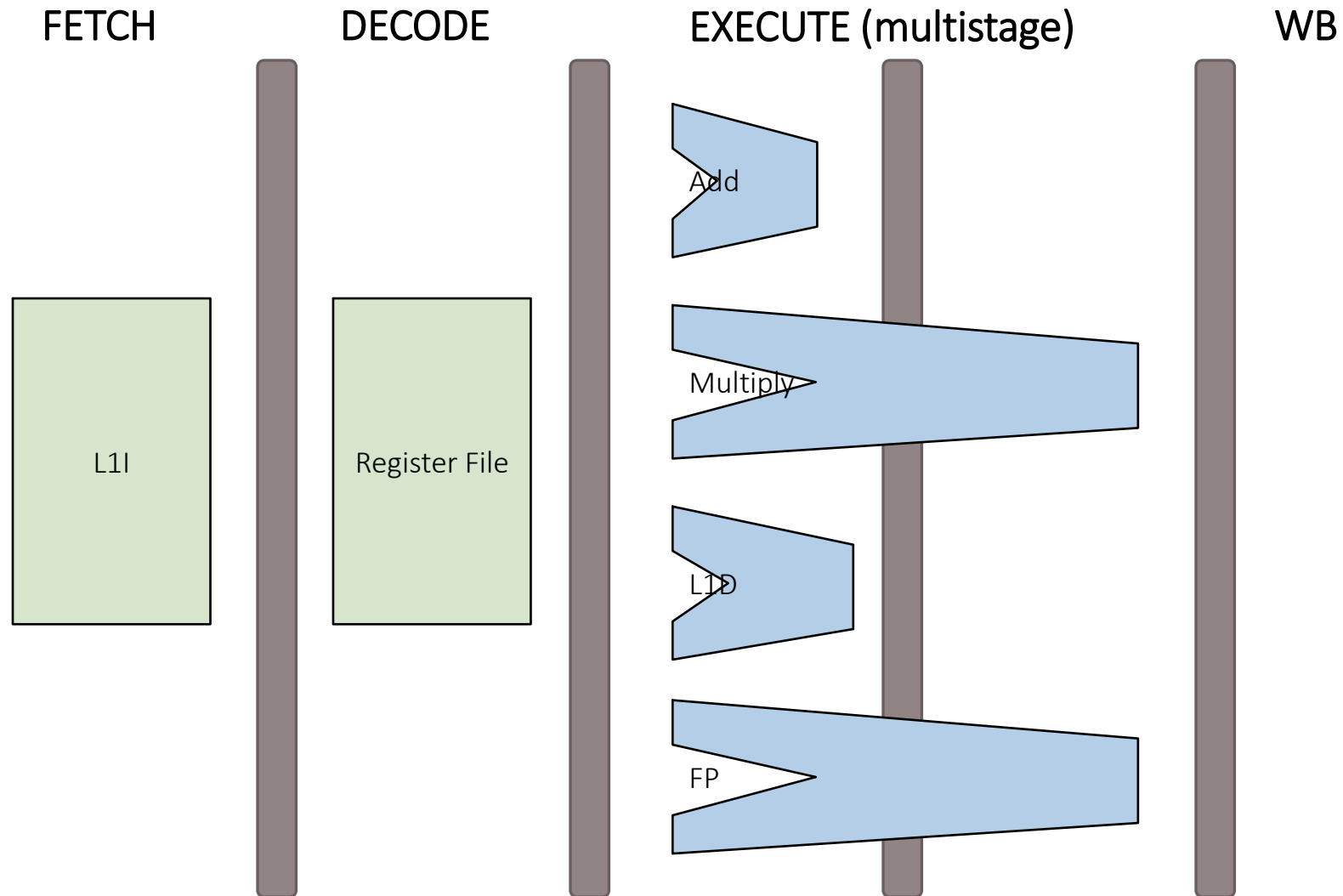
- In-order issue
 - Instructions fetched and decoded in program order
- Out-of-order completion
 - Slow instructions may complete after ones that are later in program order
 - Reason for imprecise exceptions ... but **difficult to reason about**
 - ...will revisit this in OOO / dynamic scheduling

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
 - ...will revisit this in **VLIW**

SUPERSCALAR

Pipelines thus far



Increasing Performance

$$\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Decrease # instructions

Decrease CPI

Decrease cycle time

How well does pipelining do?

Pipelining performance

$$\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Decrease # instructions

- Pipelining has no impact

Decrease CPI

- Pipelining increases CPI due to hazards & stalls (but by less than $N\times$)

Decrease cycle time

- N-stage can give $\approx N\times$ faster clock

Limitations of pipelining

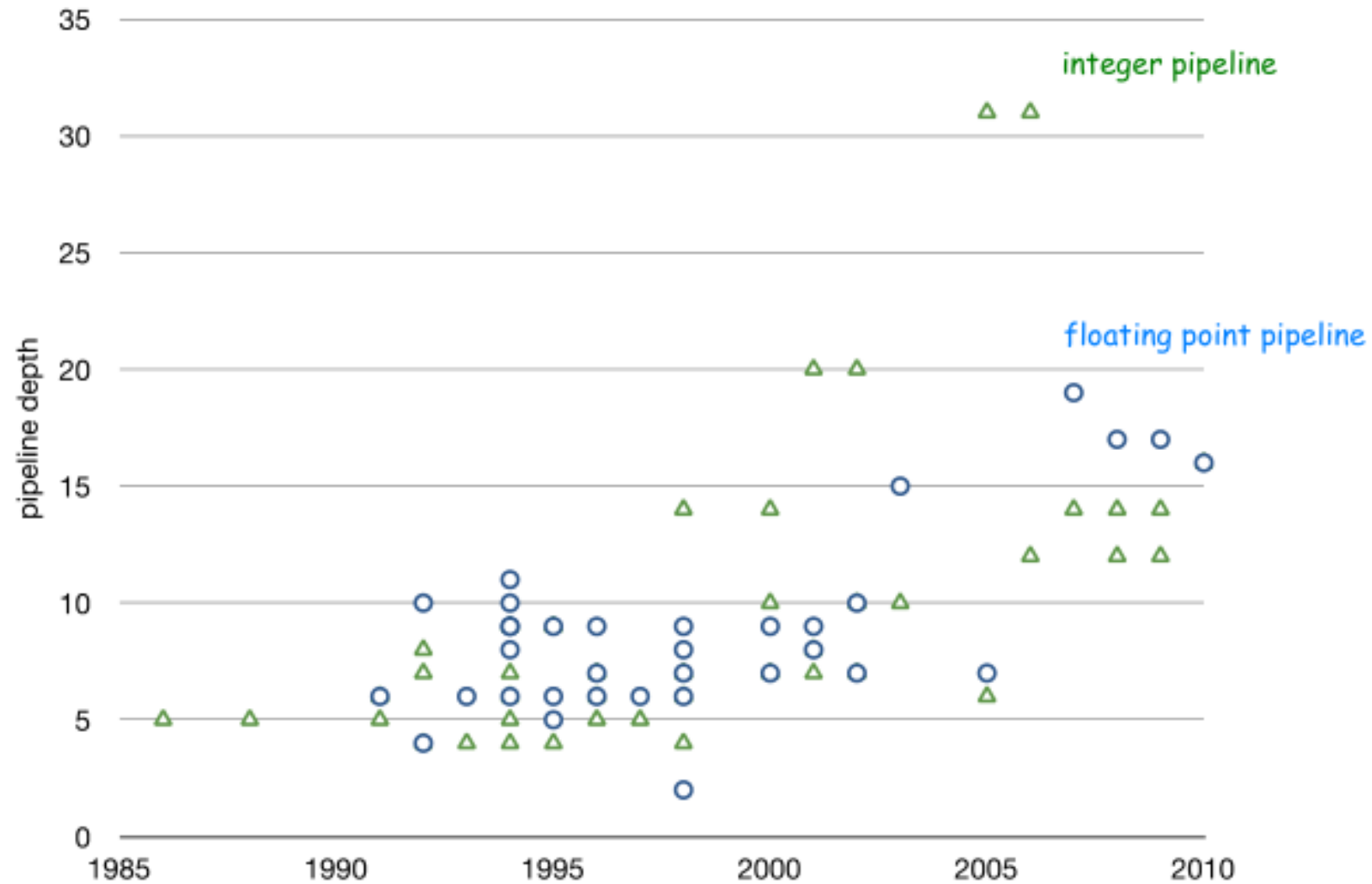
Stages can't be increased forever

Pipeline overheads become significant

Bypassing more expensive, less effective

Flushes due to mis-predicted branches

Pipeline Depth over Time



Going beyond pipelining

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

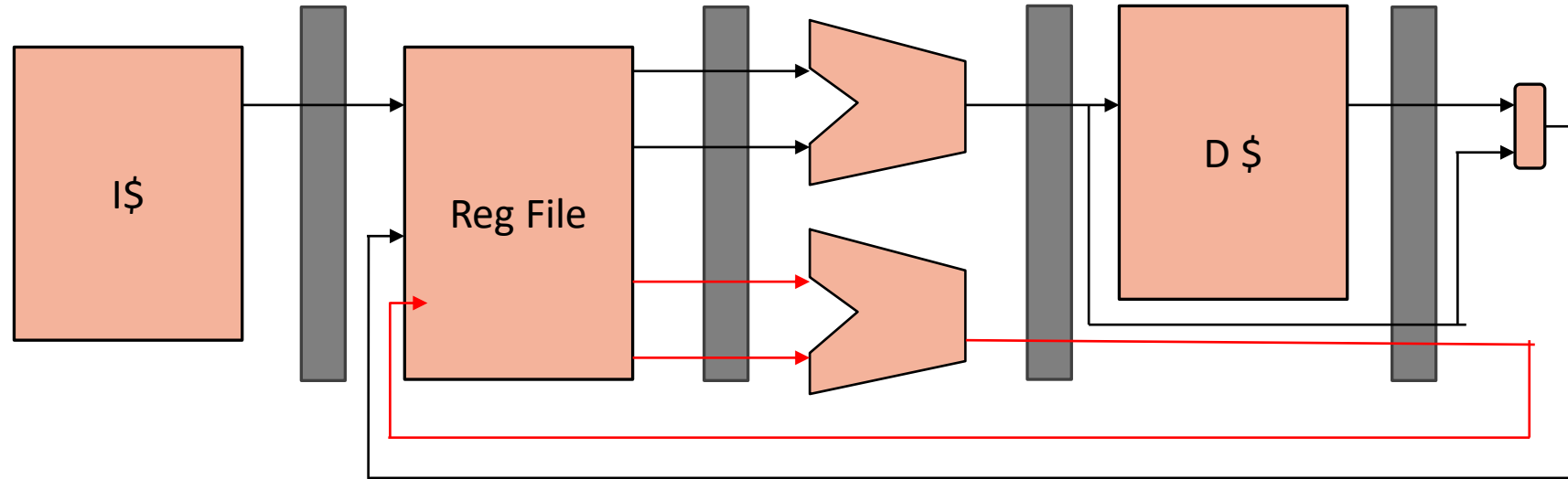
But we have >1 functional unit in execute stage

Why not issue multiple instructions per cycle?

→ *Superscalar processors*

- Instruction-level parallelism (ILP)
- Today processors typically 2- or 4-wide issue

Typical Dual-Issue Pipeline (1/2)



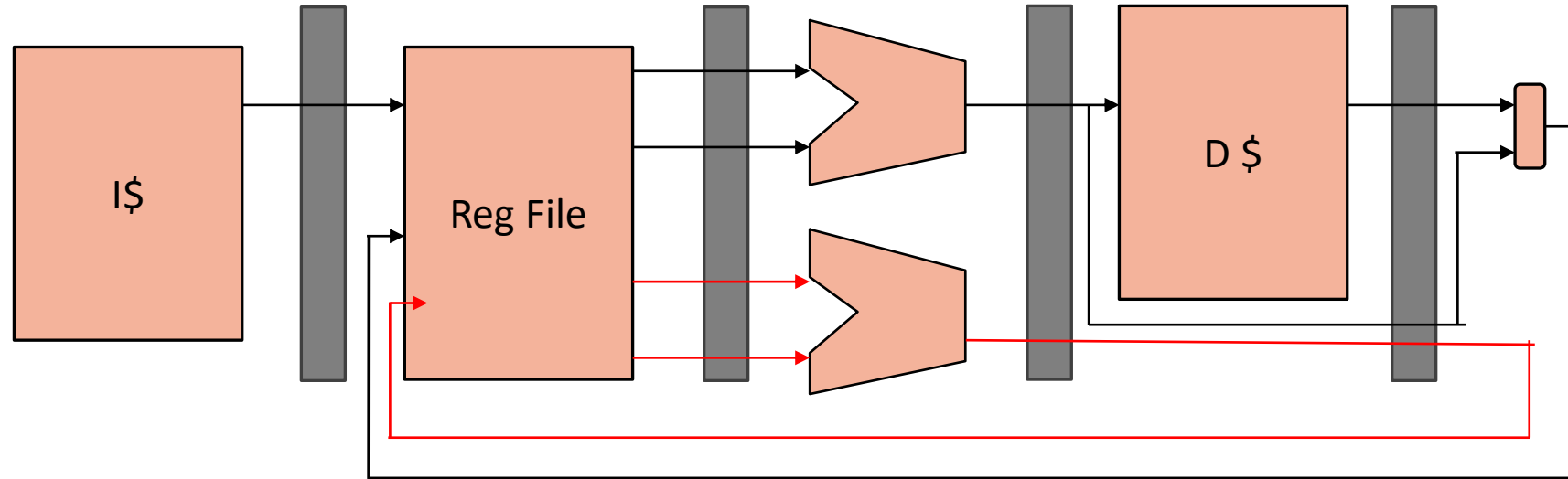
Fetch an entire 16B or 32B cache block

- ≈ 4 to 8 instructions
- Predict a single branch per cycle

Parallel decode

- Check for conflicting instructions
- Other stalls as needed (eg, load-use delay)

Typical Dual-Issue Pipeline (2/2)



Multi-ported register file

- Larger area, latency, power, cost & complexity – bad scaling too!

Multiple execution units

- Simple adders are easy, but bypass paths are expensive

Memory unit

- Single load per cycle is probably OK for dual-issue
- Alternative: Add read port to D\$

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

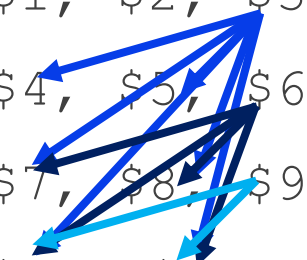
```
graph TD; I1[ADDQ $1, $2, $3] --> I2[ADDQ $4, $5, $6]; I1 --> I3[ADDQ $7, $8, $9]; I1 --> I4[ADDQ $10, $11, $12]; I2 --> I5[ADDQ $13, $14, $15]; I3 --> I5; I4 --> I5; I5 --> I6[ADDQ $16, $17, $18];
```

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, \$9, \$15
ADDQ \$16, \$17, \$18

The diagram illustrates data dependencies between six instructions. Blue arrows show the flow of data: from the first instruction to the third and fourth; from the second to the third; from the third to the fourth and fifth; and from the fourth to the fifth. Red numbers 3 and 9 highlight the superscalar width, indicating that the third and fourth instructions are part of a 3-wide issue.

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 $O(\text{ports}^2)$

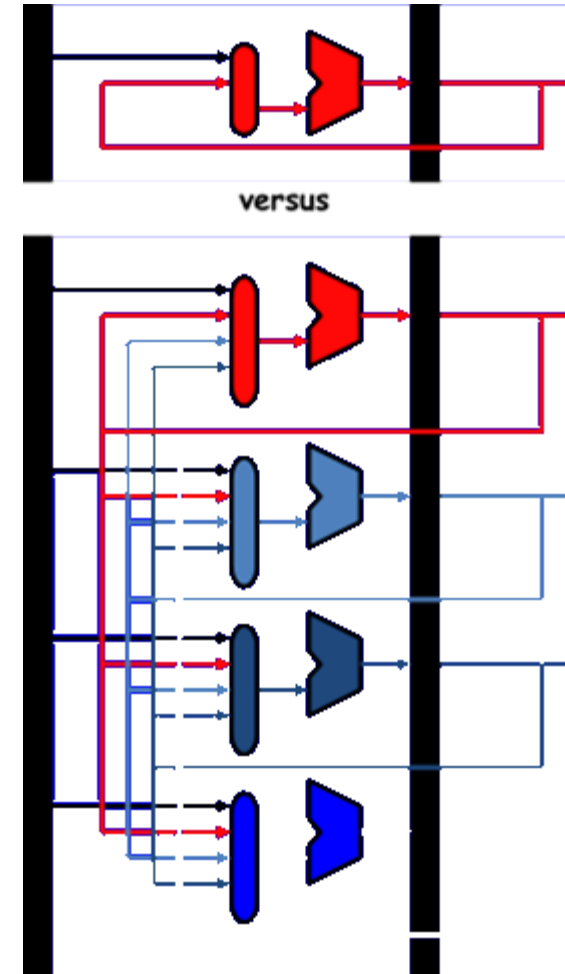
Superscalar Implementation - EX

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

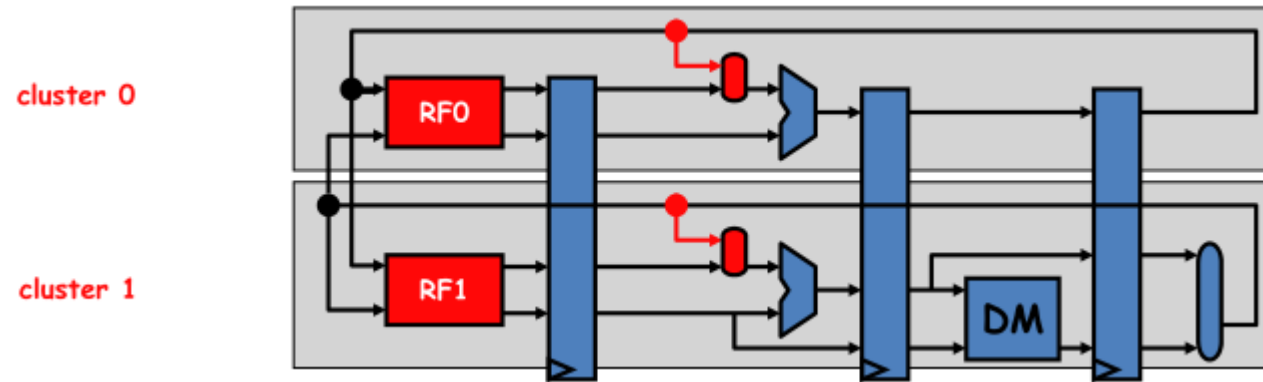
Register file also expensive

Dependency checks are a distant 3rd

Idea: Clustering

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

- No real choice, must execute correctly

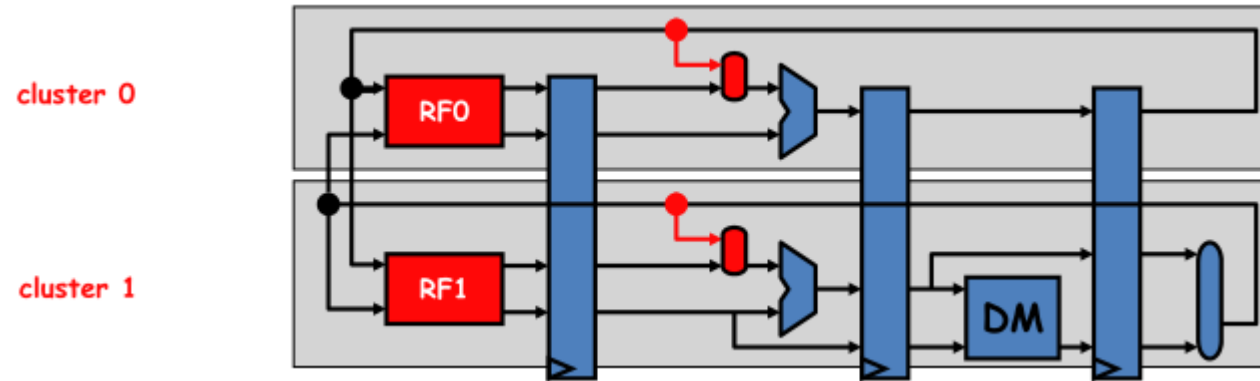


Cluster execution units and register file

- 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?

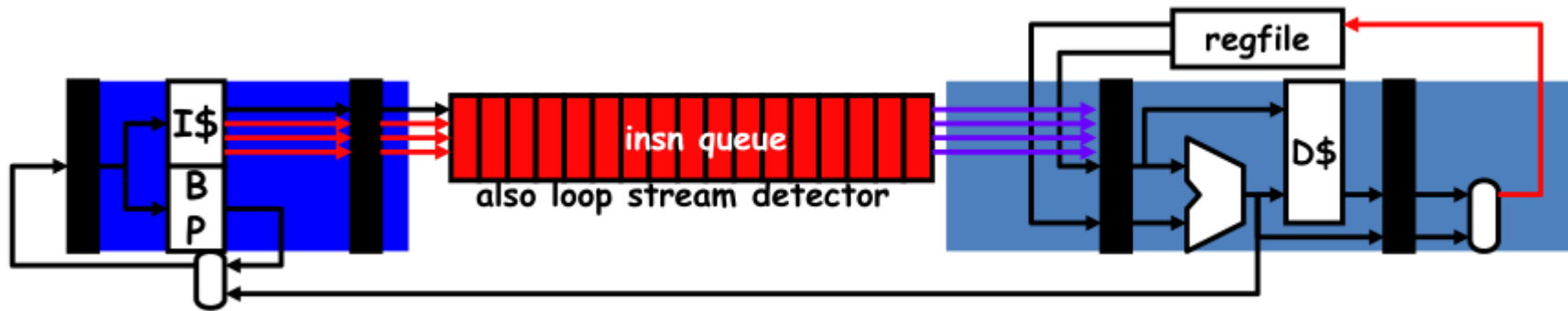
- 20% branches x 50% taken → ~10 instructions between taken branches

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

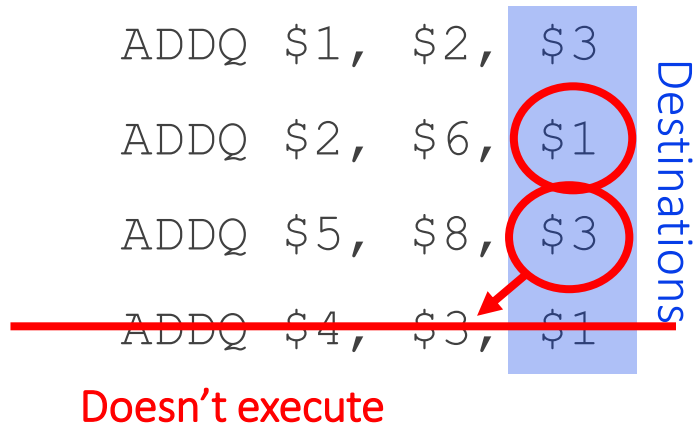
- Add a queue between fetch and decode (18 entries on Core 2)
- Compensates for cycles that fetch less than maximum issue
- Decouples front-end and back-end
 - (see also Decoupled Access Execute [Smith,'82] for a more radical version of this idea)

Or put entire loops in icache

- Any mispredicted branch falls back to normal fetch
- E.g., trace cache in Pentium 4

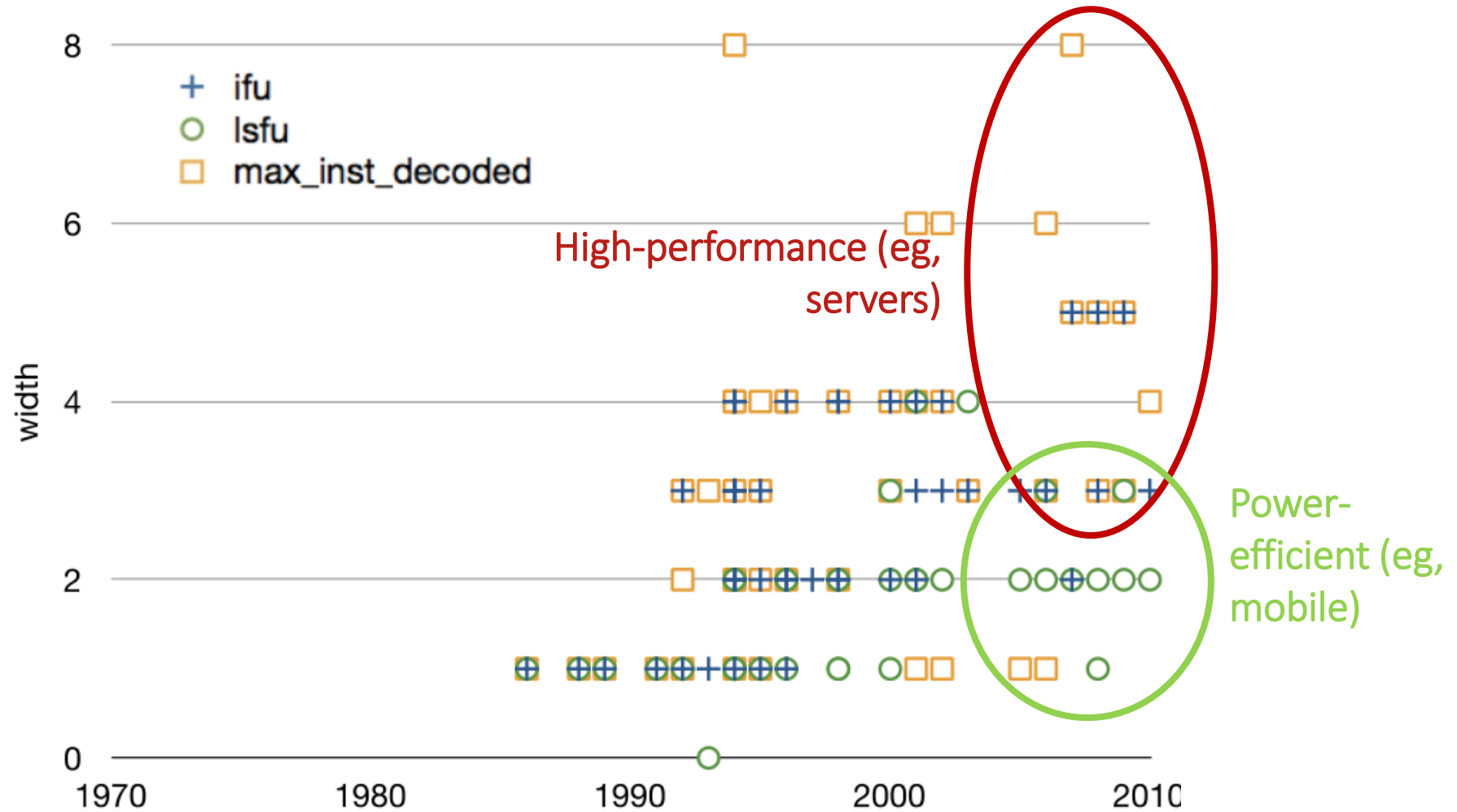
Other challenges: Superscalar Commit

Which instructions write registers in a 4-wide issue?



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 $\mathcal{O}(N^2)$
 - Clustering one solution
- Fetch and decode complicated
 - Buffering, loop streaming, trace cache
- Dependence checks also scales $\mathcal{O}(N^2)$
 - VLIW tries to fix this (next time)

Self-check questions

Why does clustering improve superscalar implementation overheads? (Give a simple equation for the overheads of a clustered design vs. a conventional superscalar.)

What are the pros and cons of superscalar vs. multicore?
How about as part of a multicore design?

If a superscalar processor executes a load and store on the same cycle, how are they ordered in terms of memory consistency?