

Lecture 13: Pipelining

Philipp Koehn

October 5, 2020

601.229 Computer Systems Fundamentals



MIPS overview



- ▶ Developed by MIPS Technologies in 1984, first product in 1986
- ▶ Used in
 - ▶ Silicon Graphics (SGI) Unix workstations
 - ▶ Digital Equipment Corporation (DEC) Unix workstation
 - ▶ Nintendo 64
 - ▶ Sony PlayStation
- ▶ Inspiration for ARM (esp. v8)

Overview

- ▶ 32 bit architecture (registers, memory addresses)
- ▶ 32 registers
- ▶ Multiply and divide instructions
- ▶ Floating point numbers

Example: Addition

- ▶ Mathematical view of addition

$$a = b + c$$

Example: Addition

- ▶ Mathematical view of addition

$$a = b + c$$

- ▶ MIPS instruction

add a,b,c

a, b, c are registers

32 Registers

- ▶ Some are special
 - 0 \$zero always has the value 0
 - 31 \$ra contains return address

32 Registers

- ▶ Some are special
 - 0 `$zero` always has the value 0
 - 31 `$ra` contains return address
- ▶ Some have usage conventions
 - 1 `$at` reserved for pseudo-instructions

32 Registers

- ▶ Some are special
 - 0 \$zero always has the value 0
 - 31 \$ra contains return address
- ▶ Some have usage conventions
 - 1 \$at reserved for pseudo-instructions
 - 2-3 \$v0-\$v1 return values of a function call
 - 4-7 \$a0-\$a3 arguments for a function call

32 Registers

- ▶ Some are special

- 0 \$zero always has the value 0

- 31 \$ra contains return address

- ▶ Some have usage conventions

- 1 \$at reserved for pseudo-instructions

- 2-3 \$v0-\$v1 return values of a function call

- 4-7 \$a0-\$a3 arguments for a function call

- 8-15,24,25 \$t0-\$t9 temporaries, can be overwritten by function

- 16-23 \$s0-\$s7 saved, have to be preserved by function

32 Registers

- ▶ Some are special

0 \$zero always has the value 0

31 \$ra contains return address

- ▶ Some have usage conventions

1 \$at reserved for pseudo-instructions

2-3 \$v0-\$v1 return values of a function call

4-7 \$a0-\$a3 arguments for a function call

8-15,24,25 \$t0-\$t9 temporaries, can be overwritten by function

16-23 \$s0-\$s7 saved, have to be preserved by function

26-27 \$k0-\$k1 reserved for kernel

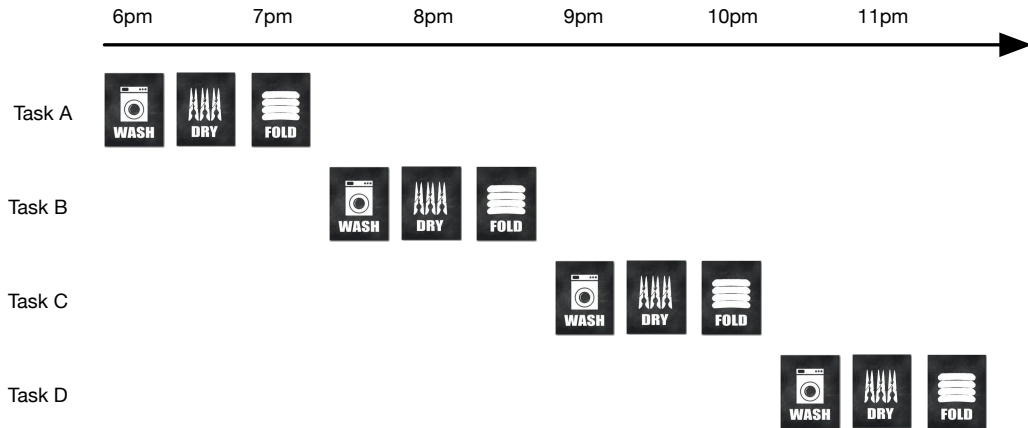
28 \$gp global area pointer

29 \$sp stack pointer

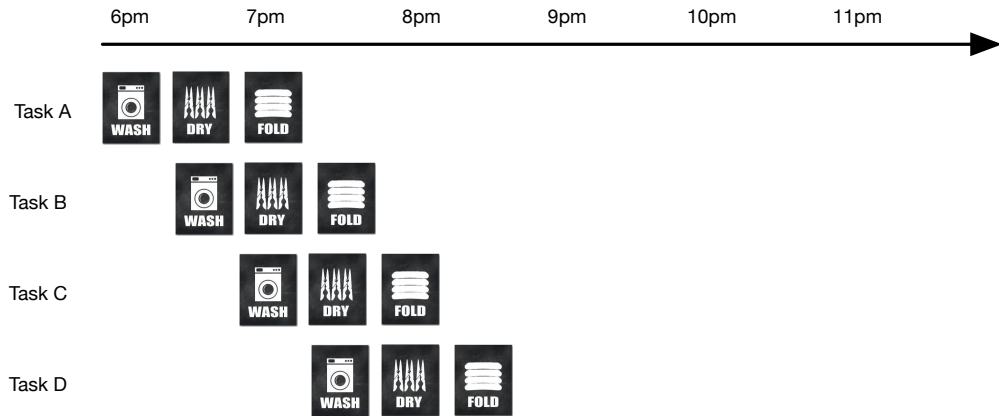
30 \$fp frame pointer

Pipelining

Laundry Analogy



Laundry Pipelined



Speed-up

- ▶ Theoretical speed-up: 3 times
- ▶ Actual speed-up in example: 2 times
 - ▶ sequential: $1:30 + 1:30 + 1:30 + 1:30 = 6$ hours
 - ▶ pipelined: $1:30 + 0:30 + 0:30 + 0:30 = 3$ hours
- ▶ Many tasks \rightarrow speed-up approaches theoretical limit

MIPS instruction pipeline

MIPS Pipeline

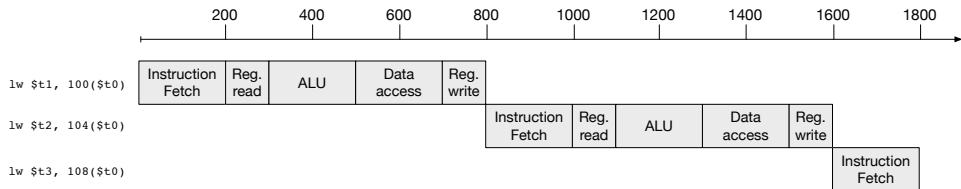
- ▶ Fetch instruction from memory
- ▶ Read registers and decode instruction (note: registers are always encoded in same place in instruction)
- ▶ Execute operation OR calculate an address
- ▶ Access an operand in memory
- ▶ Write result into a register

Time for Instructions

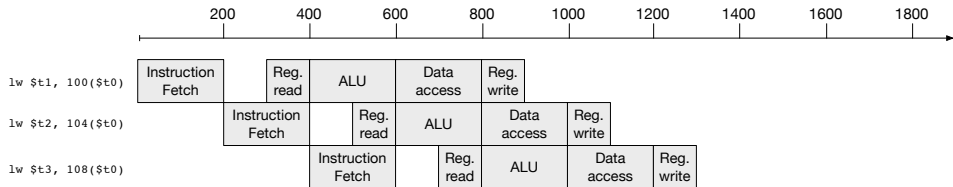
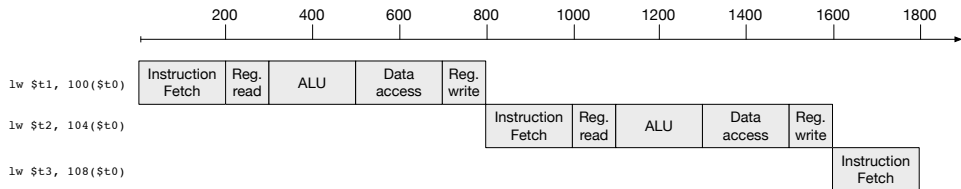
Breakdown for each type of instruction

Instruction class	Instr. fetch	Register read	ALU oper.	Data access	Register write	Total time
Load word (lw)	200ps	100ps	200ps	200ps	100ps	800ps
Store word (sw)	200ps	100ps	200ps	200ps		700ps
R-format (add)	200ps	100ps	200ps		100ps	600ps
Branch (beq)	200ps	100ps	200ps			500ps

Pipeline Execution



Pipeline Execution



Speed-up

- ▶ Theoretical speed-up: 4 times
- ▶ Actual speed-up in example: 1.71 times
 - ▶ sequential: $800\text{ps} + 800\text{ps} + 800\text{ps} = 2400\text{ps}$
 - ▶ pipelined: $1000\text{ps} + 200\text{ps} + 200\text{ps} = 1400\text{ps}$
- ▶ Many tasks \rightarrow speed-up approaches theoretical limit

Design for Pipelining

- ▶ All instructions are 4 bytes
→ easy to fetch next instruction

Design for Pipelining

- ▶ All instructions are 4 bytes
→ easy to fetch next instruction
- ▶ Few instruction formats
→ parallel op decode and register read

Design for Pipelining

- ▶ All instructions are 4 bytes
→ easy to fetch next instruction
- ▶ Few instruction formats
→ parallel op decode and register read
- ▶ Memory access limited to load and store instructions
→ stage 3 used for memory access, otherwise operation execution

Design for Pipelining

- ▶ All instructions are 4 bytes
→ easy to fetch next instruction
- ▶ Few instruction formats
→ parallel op decode and register read
- ▶ Memory access limited to load and store instructions
→ stage 3 used for memory access, otherwise operation execution
- ▶ Words aligned in memory
→ able to read in one instruction
(aligned = memory address multiple of 4)

Hazards

Hazards

- ▶ Hazard = next instruction cannot be executed in next clock cycle
- ▶ Types
 - ▶ structural hazard
 - ▶ data hazard
 - ▶ control hazard

Structural Hazard

- ▶ Definition: instructions overlap in resource use in same stage
- ▶ For instance: memory access conflict

	1	2	3	4	5	6	7
i1	FETCH	DECODE	MEMORY	MEMORY	ALU	REGISTER	
i2		FETCH	DECODE	MEMORY	MEMORY	ALU	REGISTER
				conflict			

- ▶ MIPS designed to avoid structural hazards

Data Hazard

- ▶ Definition: instruction waits on result from prior instruction

- ▶ Example

add \$s0, \$t0, \$t1

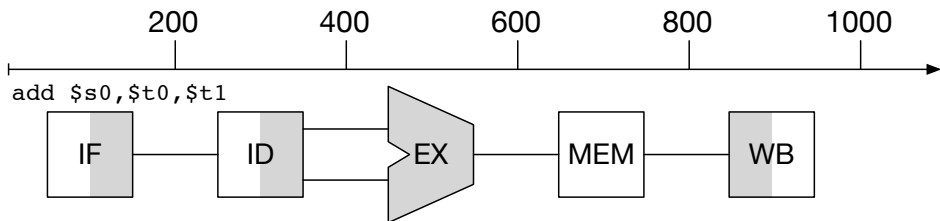
sub \$t0, \$s0, \$t3

- ▶ add instruction writes result to register \$s0 in stage 5
- ▶ sub instruction reads \$s0 in stage 2

⇒ Stage 2 of sub has to be delayed

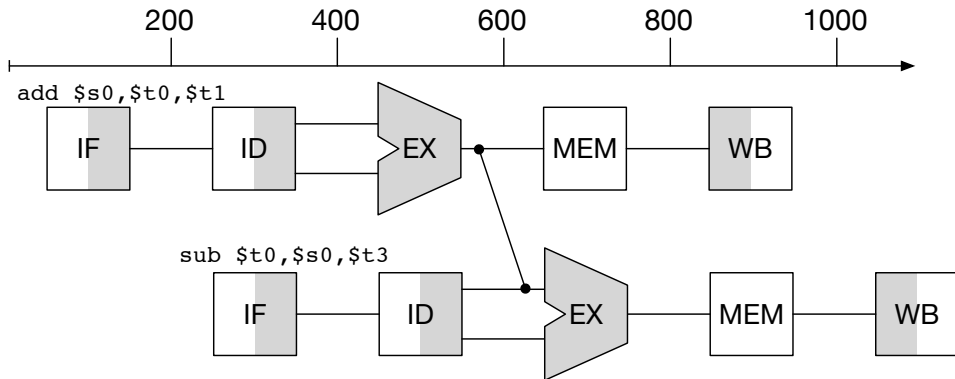
- ▶ We overcome this in hardware

Graphical Representation



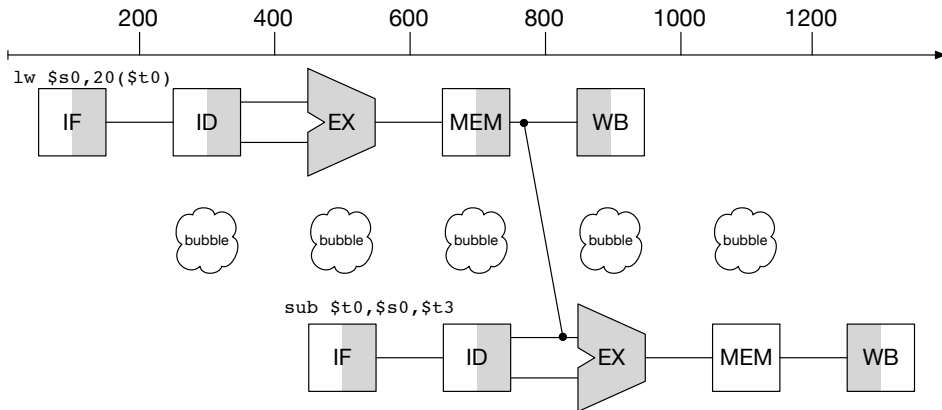
- ▶ IF: instruction fetch
- ▶ ID: instruction decode
- ▶ EX: execution
- ▶ MEM: memory access
- ▶ WB: write-back

Add and Subtract



- Add wiring to circuit to directly connect output of ALU for next instruction

Load and Subtract



- ▶ Add wiring from memory lookup to ALU
- ▶ Still 1 cycle unused: "pipeline stall" or "bubble"

Reorder Code

Code with data hazard

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Reorder Code

Code with data hazard

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Reorder code (may be done by compiler)

Reorder Code

Code with data hazard

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Reorder code (may be done by compiler)

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Load instruction now completed in time

Clicker quiz!

Clicker quiz omitted from public slides

Clicker quiz!

Clicker quiz omitted from public slides

Control Hazard

- ▶ Also called branch hazard
- ▶ Selection of next instruction depends on outcome of previous
- ▶ Example

```
add $s0, $t0, $t1  
beq $s0, $s1, ff40  
sub $t0, $s0, $t3
```

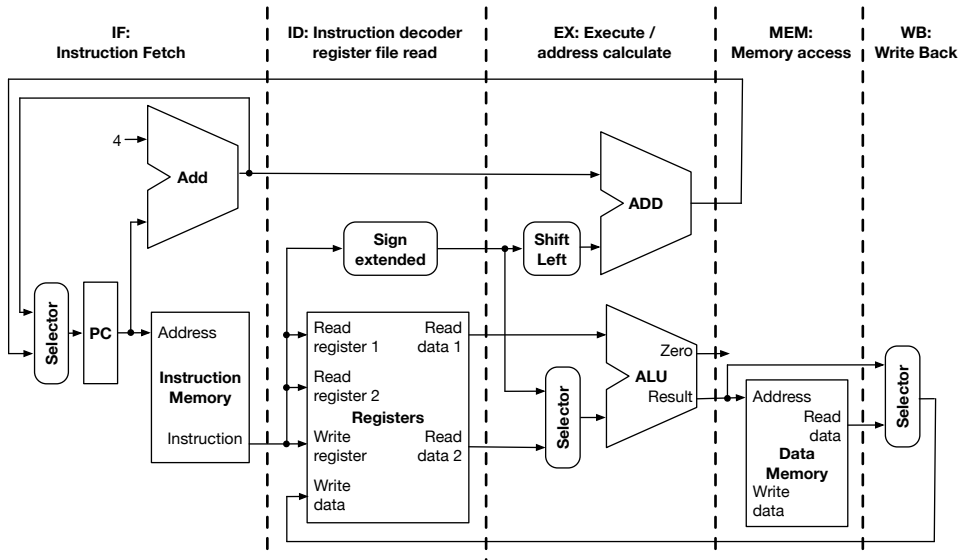
- ▶ sub instruction only executed if branch condition fails
- cannot start until branch condition result known

Branch Prediction

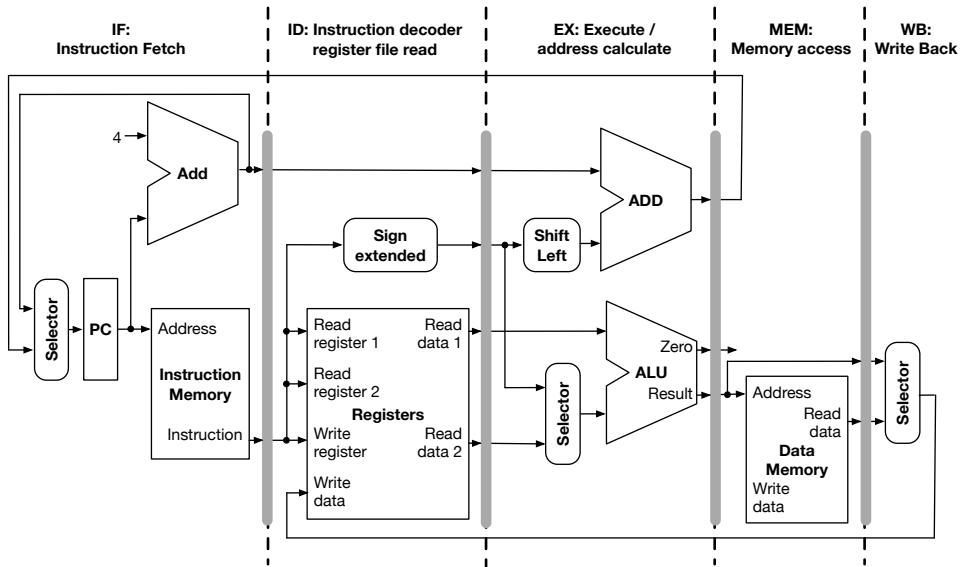
- ▶ Assume that branches are never taken
→ full speed if correct
- ▶ More sophisticated
 - ▶ keep record of branch taken or not
 - ▶ make prediction based on history

Pipelined data path

Datapath

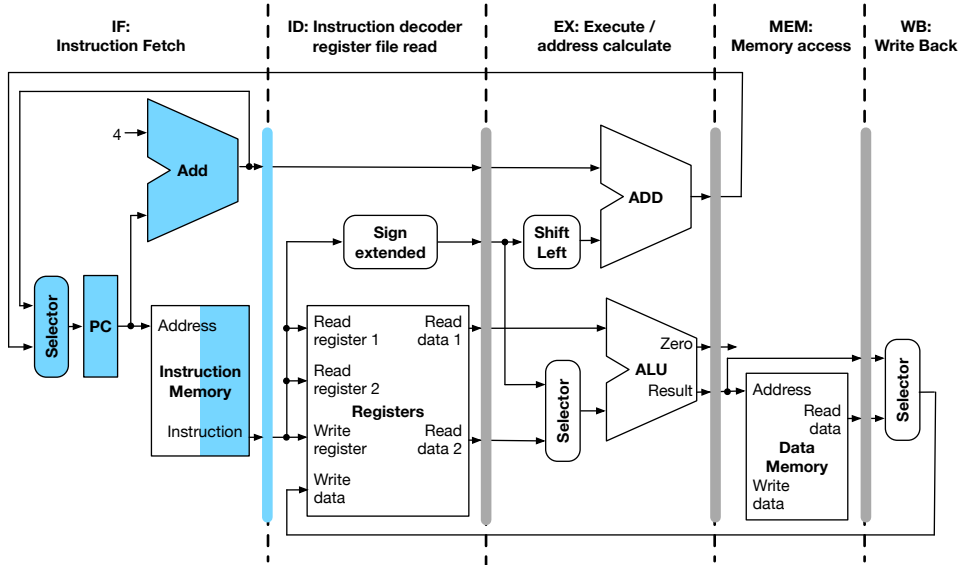


Pipelined Datapath

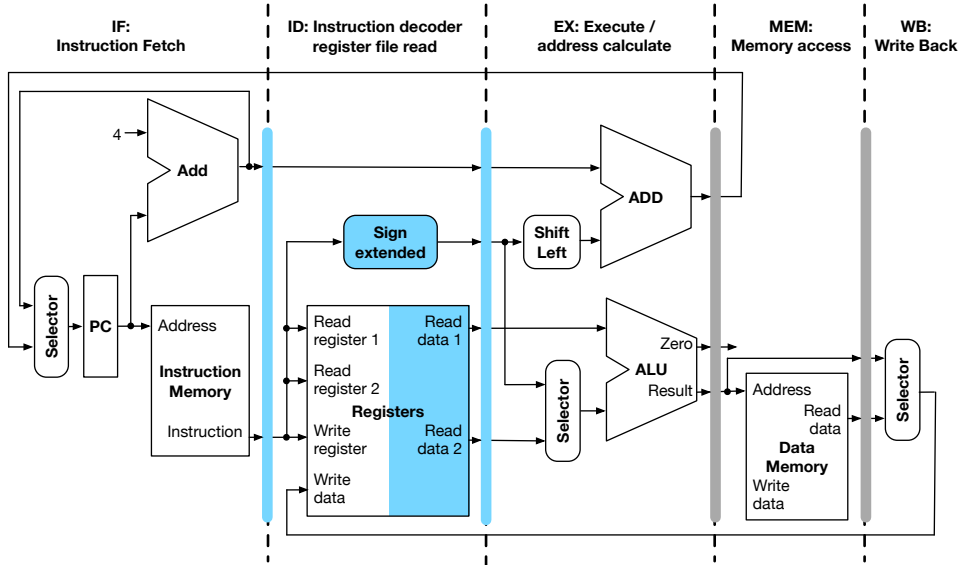


Load

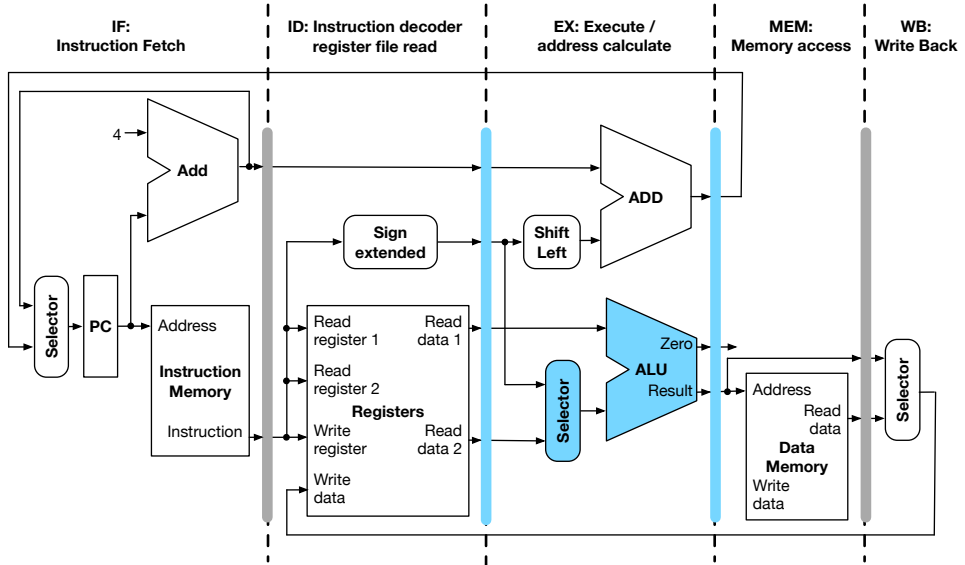
Load: Stage 1



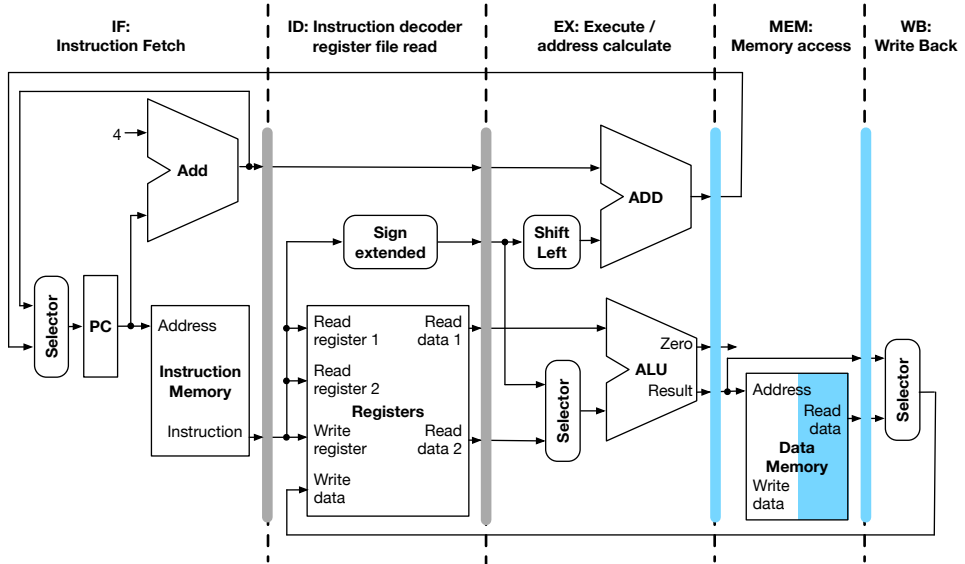
Load: Stage 2



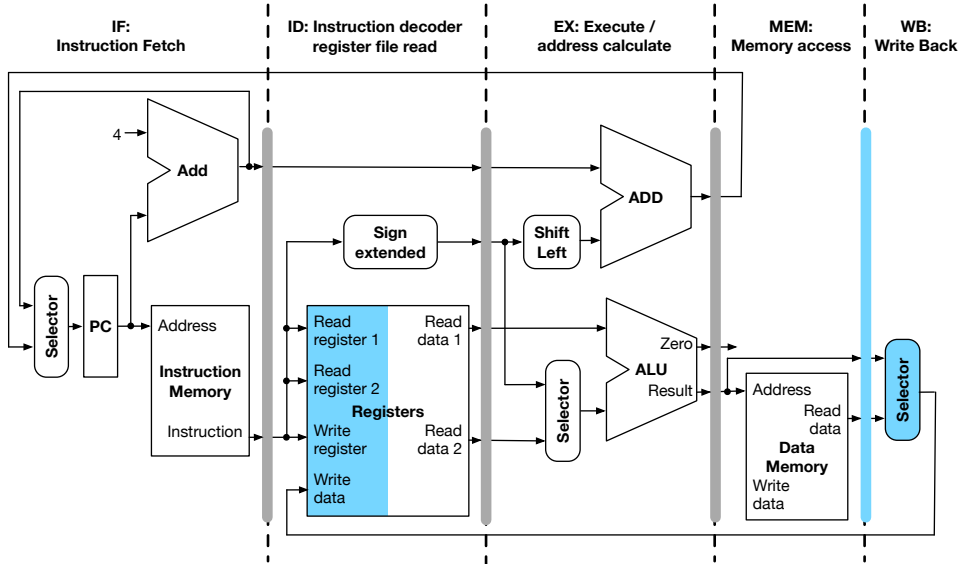
Load: Stage 3



Load: Stage 4

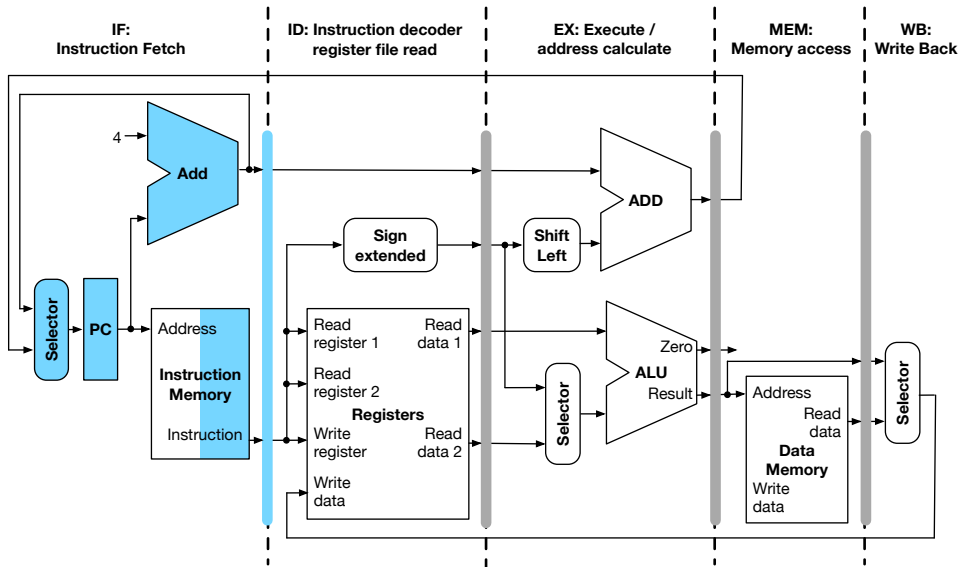


Load: Stage 5

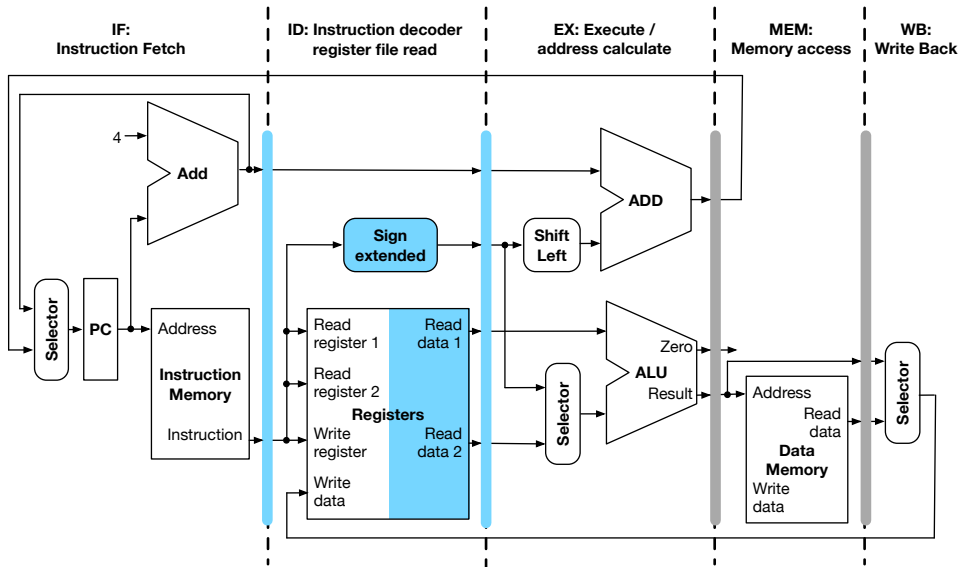


Store

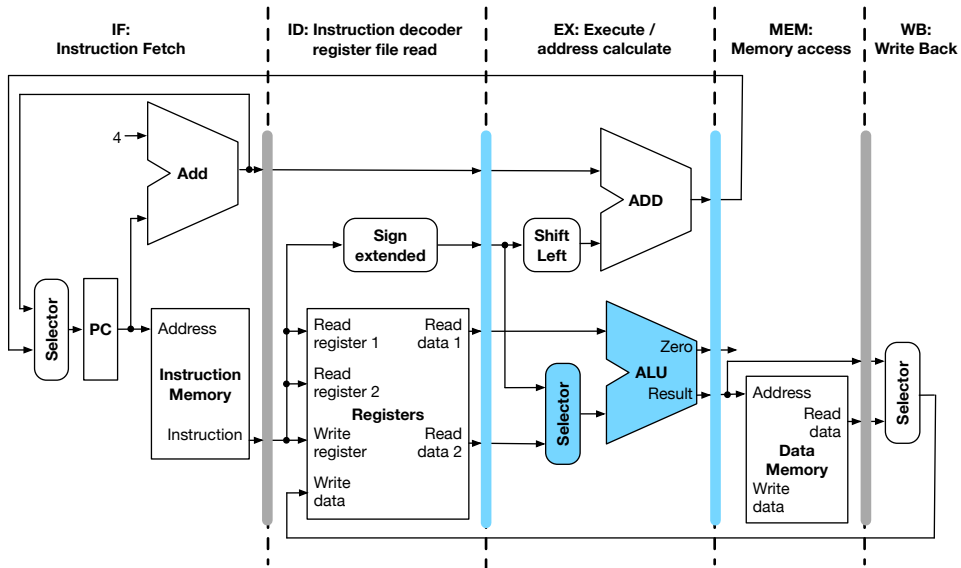
Store: Stage 1



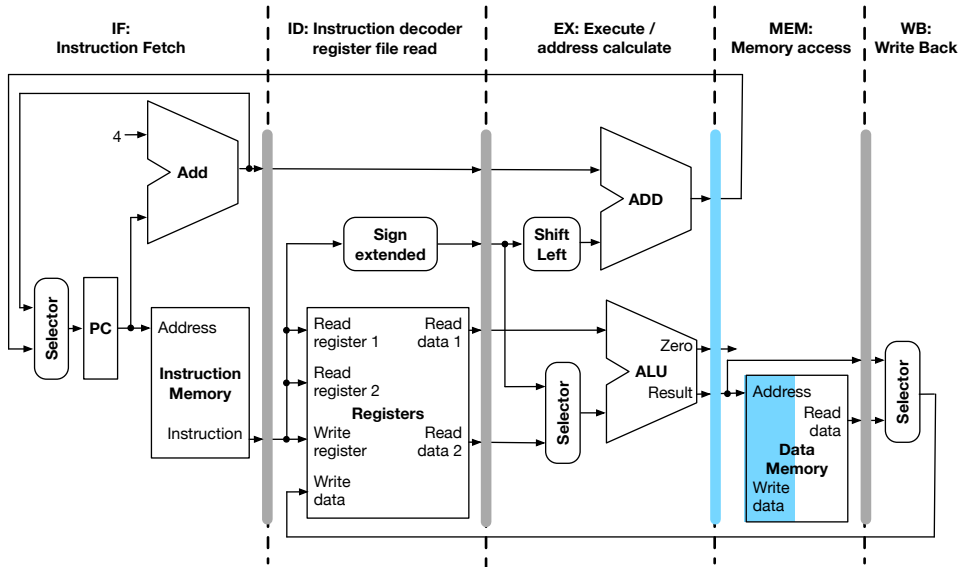
Store: Stage 2



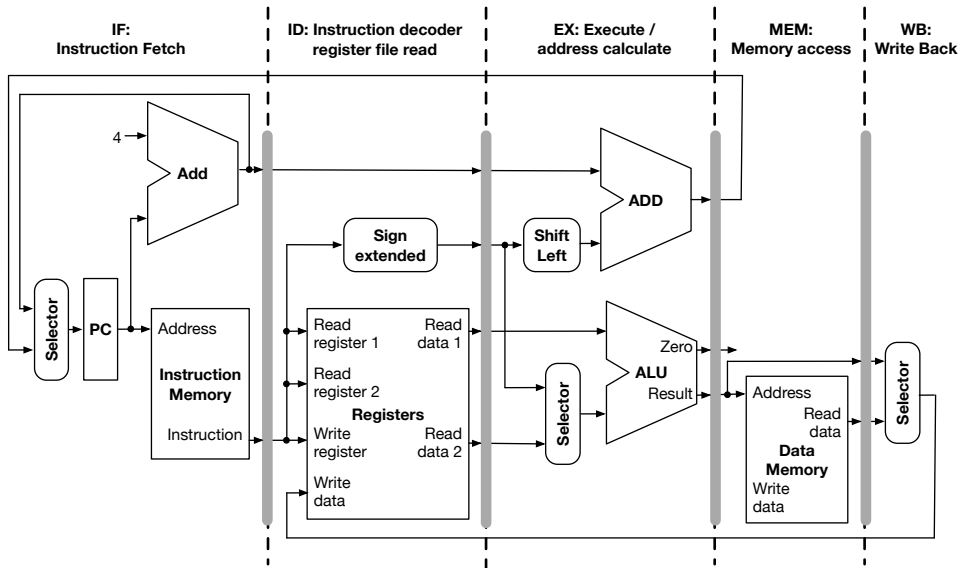
Store: Stage 3



Store: Stage 4

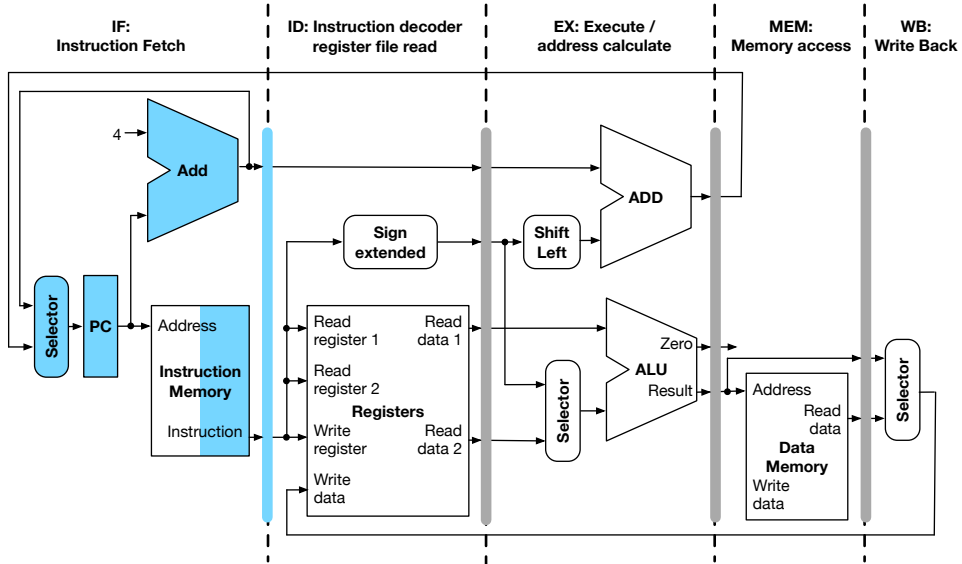


Store: Stage 5

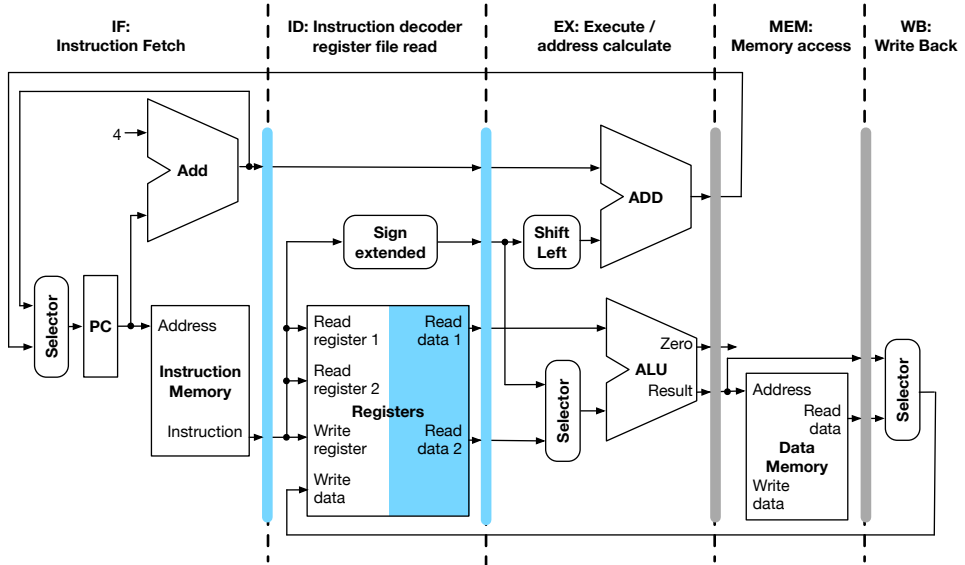


Add

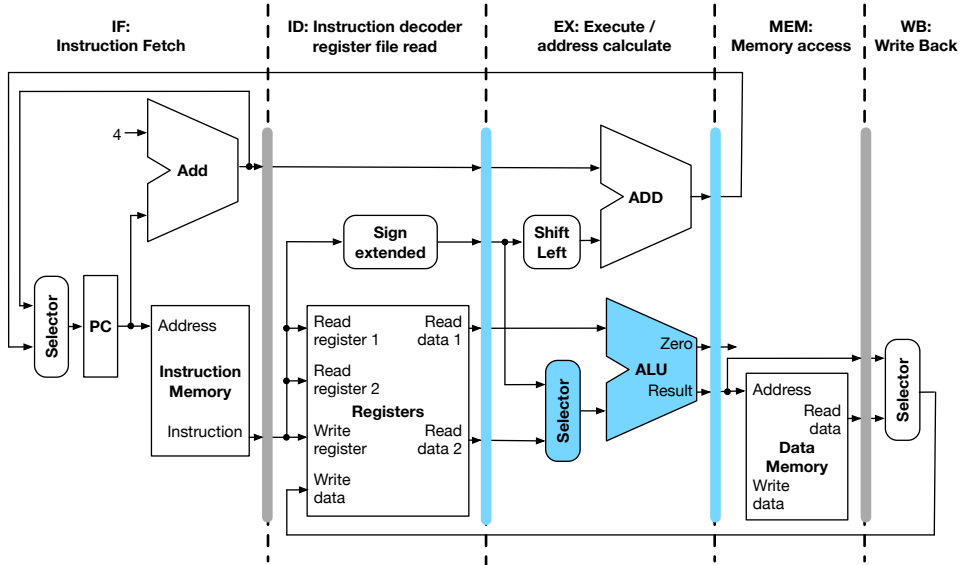
Add: Stage 1



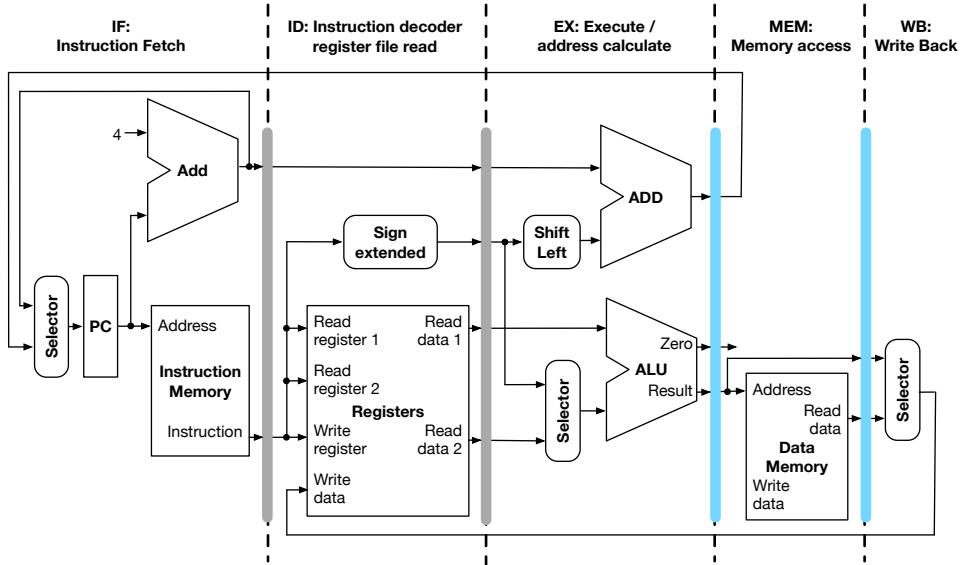
Add: Stage 2



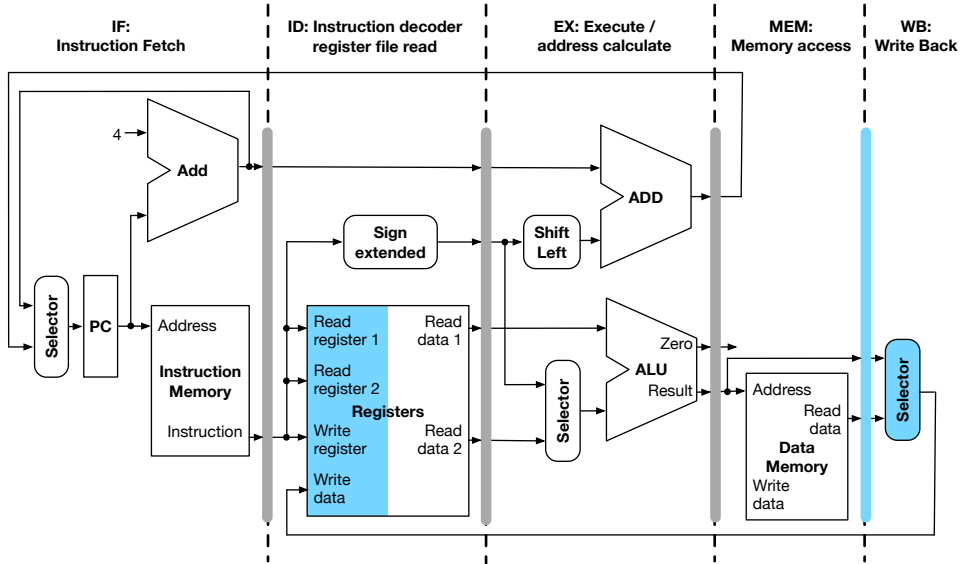
Add: Stage 3



Add: Stage 4

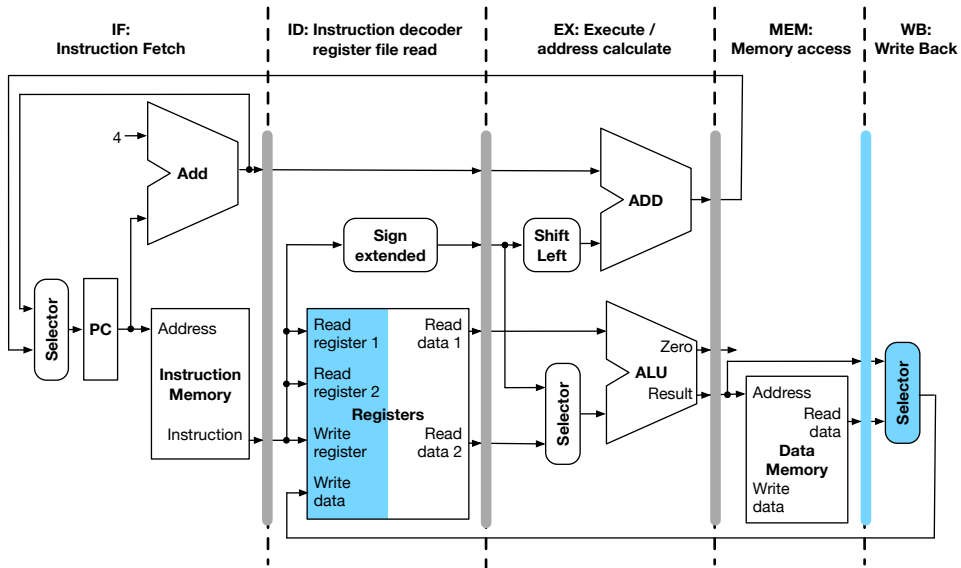


Add: Stage 5



Write to register

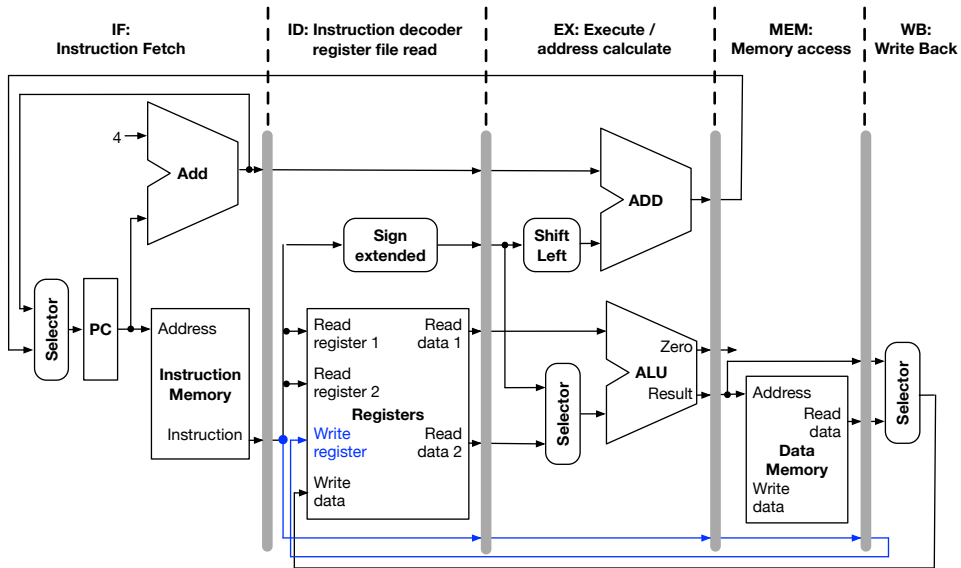
Which Register?



Problem

- ▶ Write register
 - ▶ decoded in stage 2
 - ▶ used in stage 5
- ▶ Identity of register has to be passed along

Data Path for Write Register

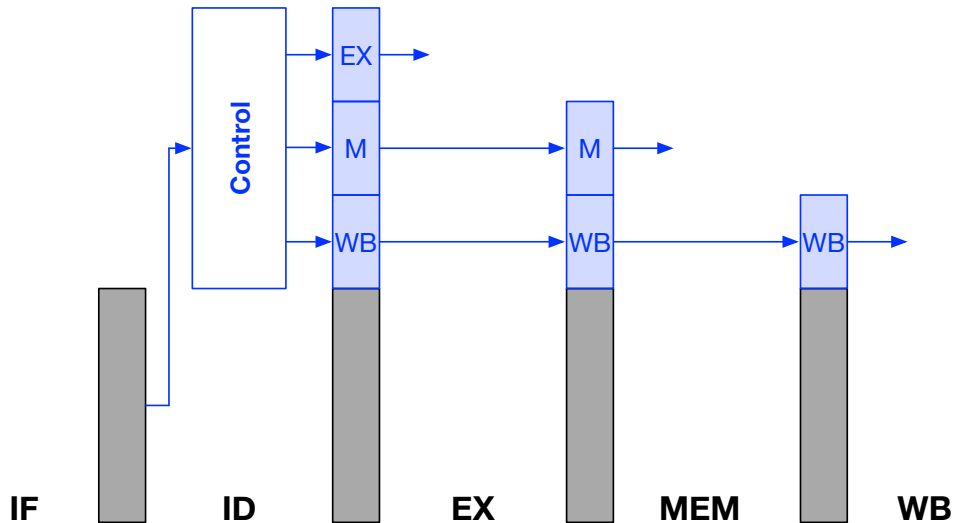


Pipelined control

Pipelined Control

- ▶ At each stage, information from instruction is needed
 - ▶ which ALU operation to execute
 - ▶ which memory address to consult
 - ▶ which register to write to
- ▶ This control information has to be passed through stages

Pipelined Control



Control Flags

