Dependencies and Performance

Computer Architecture

ECE 6913

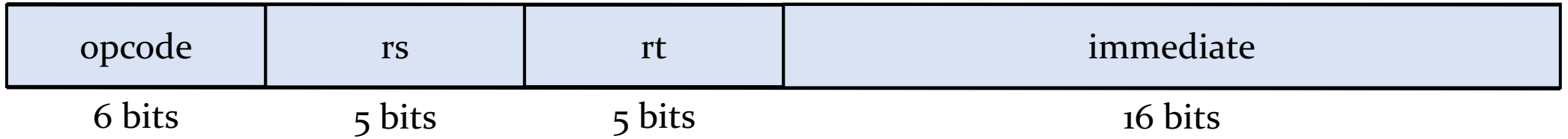
Brandon Reagen

NYU TANDON SCHOOL OF ENGINEERING

What we'll cover today

1) Review:
    1) Control flow example
    2) Stack
2) Dependencies and pipelining
3) Understanding performance
4) Lab 2

MIPS Instructions: I-Type




```
addi    rs, rt, imm     // R[rt]  ← R[rs] + {SignExtend, imm}; MSB of imm is extended to 32 bits
ori     rs, rt, imm     // R[rt]  ← R[rs] | {ZeroExtend, imm}; bit-wise Boolean OR operation

beq     rs, rt, imm     // if{R[rs] == R[rt]} branch to PC + 4 + BranchAddress;  ("PC relative")
                        // Else go to PC+4
                        // BranchAddress = {SignExtend, imm, 00}

lw      rs, rt, imm     // R[rt]  ← Mem[ {SignExtend, imm} +  R[rs] ] ("Displaced/based")
```

MIPS Instructions: J-Type


```
                                // JumpAddress = { (PC+4)[31:28], address, 2'b0 }
                                // (Pseudodirect)
j       address                 // PC ← JumpAddress
jal     address                 // R[31] ← PC+4; PC ← JumpAddress;
```

Branch instructions verses jump?

Why store PC+4?

First quiz

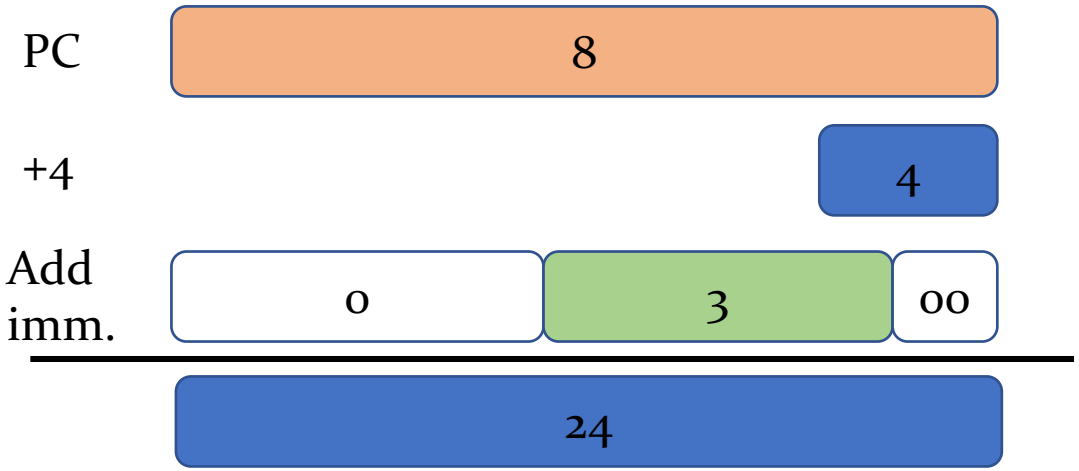
```
for (j=0; j<10; j++)
{
    b = b + j;
}
```

Address	Instruction	Comment
0		
4		
8		
12		
16		
20		
24	…	Done with loop

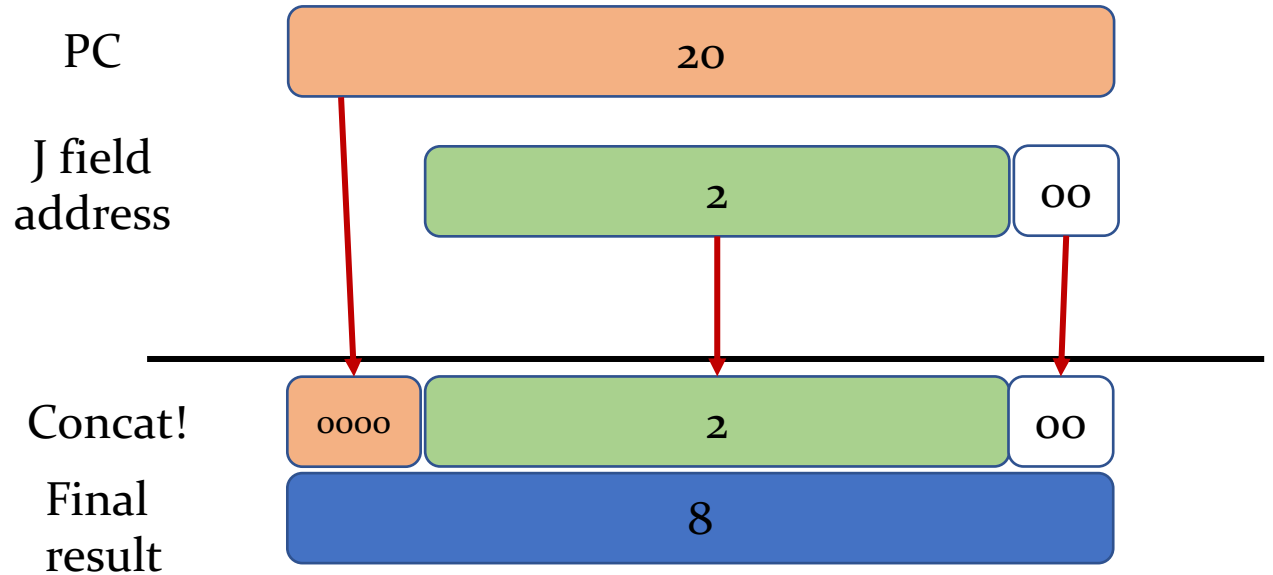
First quiz

```
for (j=0; j<10; j++)
{
    b = b + j;
}
```

Address	Instruction	Comment
0	Addi r5, r0, 0	J = 0
4	Addi r1, r0, 10	R1 = 10; exit condition
8	Beq r5, r1, 3	If j is 10, go to 24
12	Add r6, r6, r5	b = b + j
16	Addi r5, r5, 1	Update j, j = j+1
20	J 2	Goto 8
24	…	Done with loop



Beq: PC = 8 + 4 + (3<<2) = 24

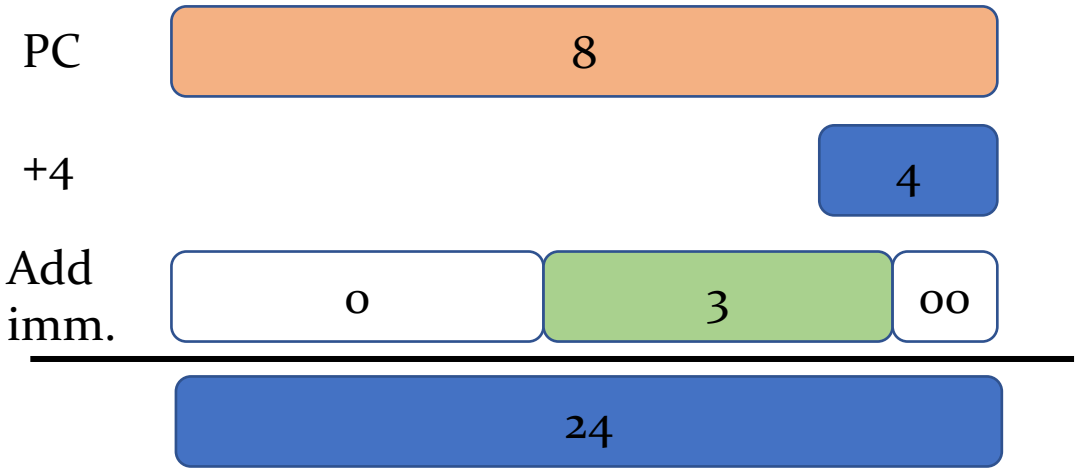


Jump: PC = 0000 || 2 || 00 = 8

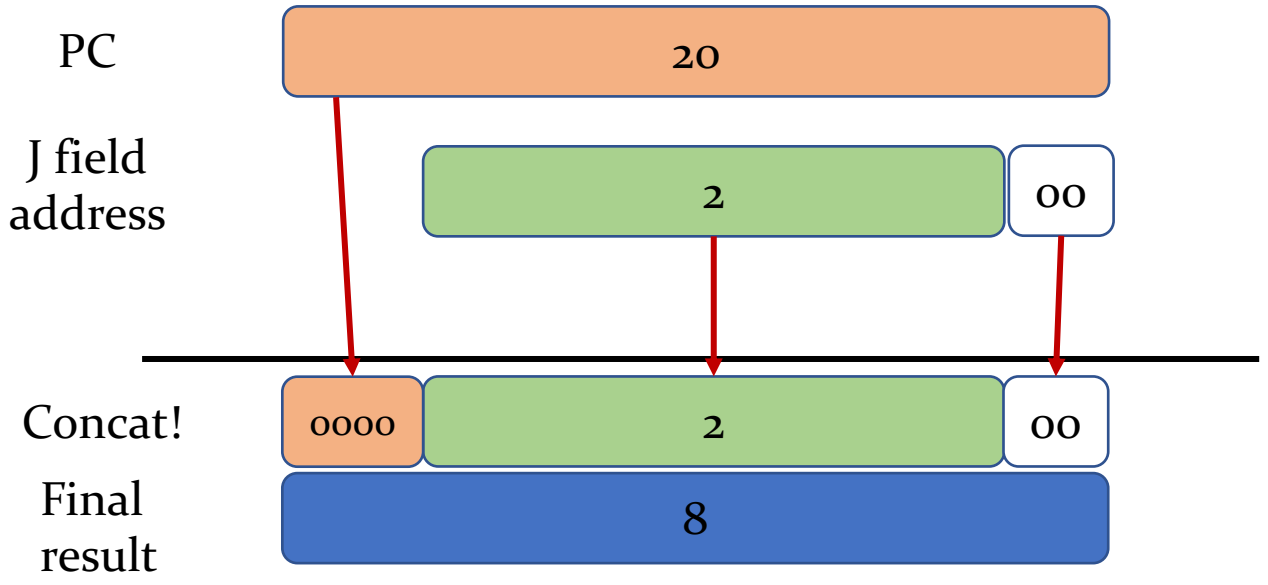
First quiz

Follow-up:

1) Why didn't we add 4 to the jump instruction?
2) What if I want to go further than 28 bits??
3) Do I seriously have to do this arithmetic???



Beq: PC = 8 + 4 + (3<<2) = 24



Jump: PC = 0000 || 2 || 00 = 8

Contrived example

```
Initialize      li $s0, 7
                li $s1, 3
Some compute    sub $s2, $s0, $s1
Jump prep       move $a0, $s0
                move $a1, $s1
Jump!           jal compute_f1
Get return val  move $s3, $v0
Compute..       slt $s4, $s2, $s3
```

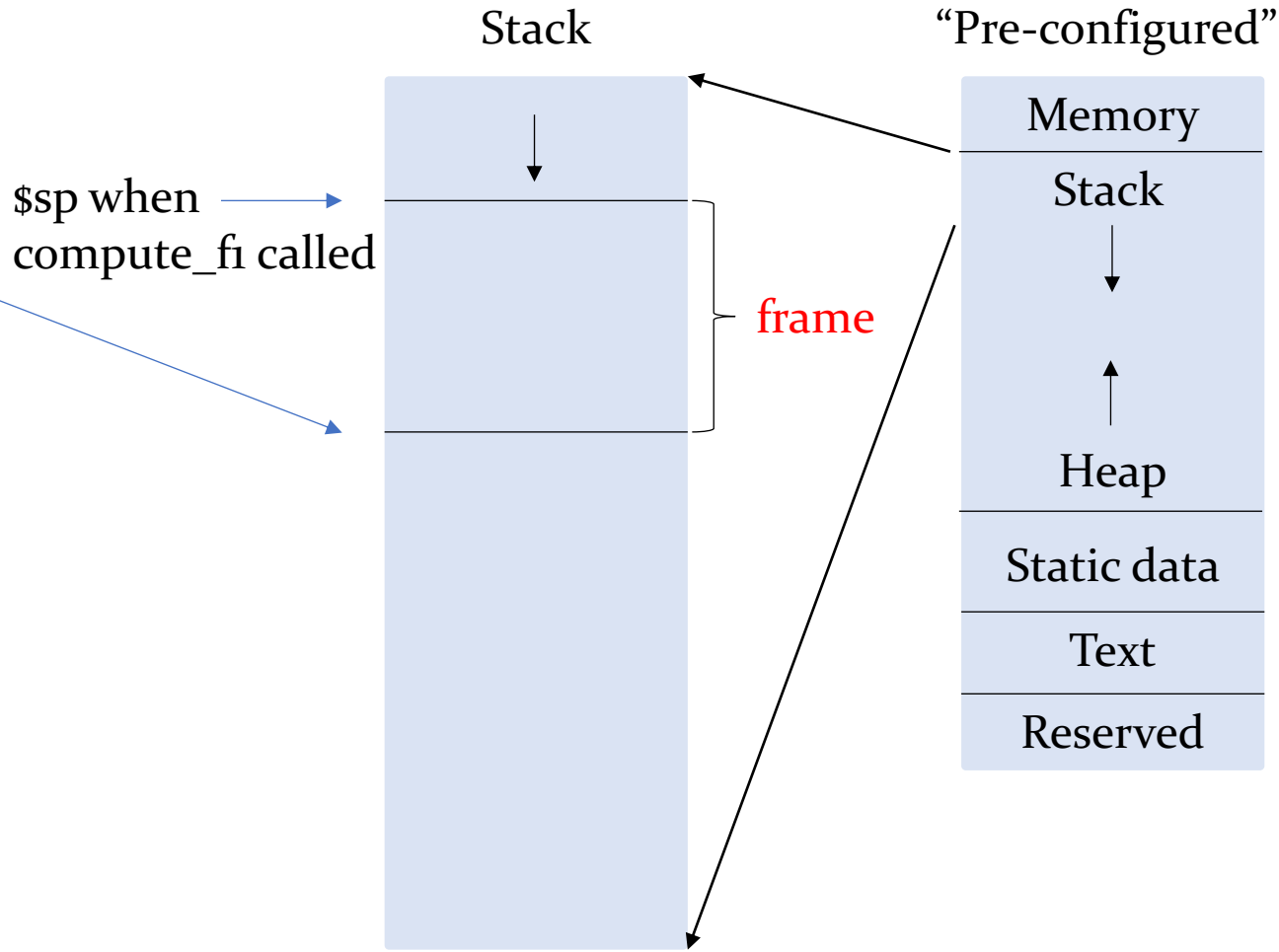
Q: what happens if a function calls another function..?

```
compute_f1:
Procedure prep:
save all used s* regs
                addi $sp, $sp, -12
                sw $s0, 8($sp)
                sw $s1, 4($sp)
                sw $s2, 0($sp)
Free to compute!
                add $s0, $a0, $a1
                addi $s1, $a1, 1
                sub $s2, $a0, $s1
                mult $s2, $s1
Return prep:
save return value
restore s* regs
restore sp!
                mflo $v0, $lo
                lw $s0, 8($sp)
                lw $s1, 4($sp)
                lw $s2, 0($sp)
                addi $sp, $sp, 12
Return control!
(Jump)          jr $ra
```

Visualizing the stack

```
compute_f1:
        addi $sp, $sp, -12
Setup   sw $s0, 8($sp)
        sw $s1, 4($sp)
        sw $s2, 0($sp)

        .....
Teardown lw $s0, 8($sp)
        lw $s1, 4($sp)
        lw $s2, 0($sp)
        addi $sp, $sp, 12
        jr $ra
```

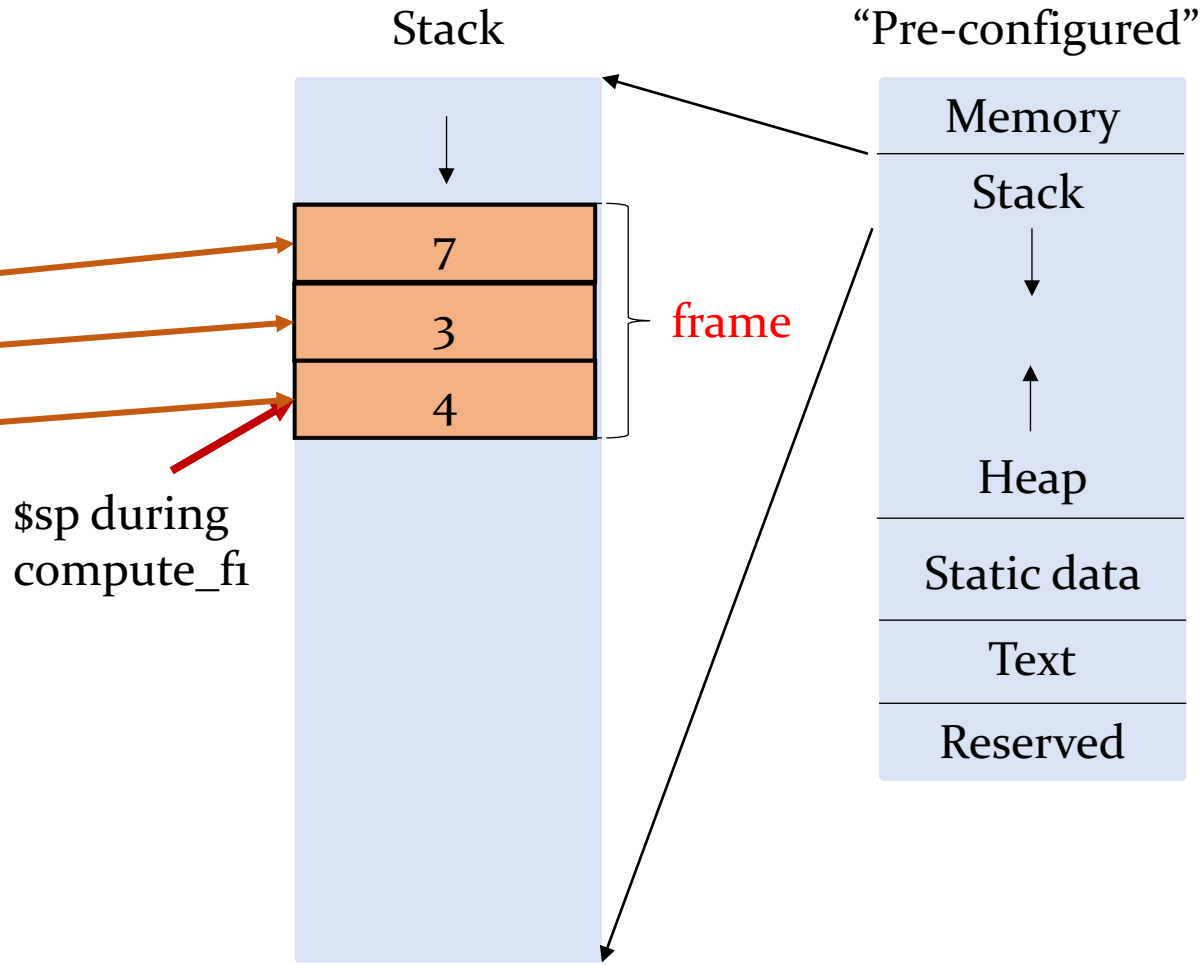


Visualizing the stack

```
compute_f1:
    addi $sp, $sp, -12
    sw $s0, 8($sp)
    sw $s1, 4($sp)
    sw $s2, 0($sp)
    .....
    lw $s0, 8($sp)
    lw $s1, 4($sp)
    lw $s2, 0($sp)
    addi $sp, $sp, 12
    jr $ra
```

Setup

Teardown

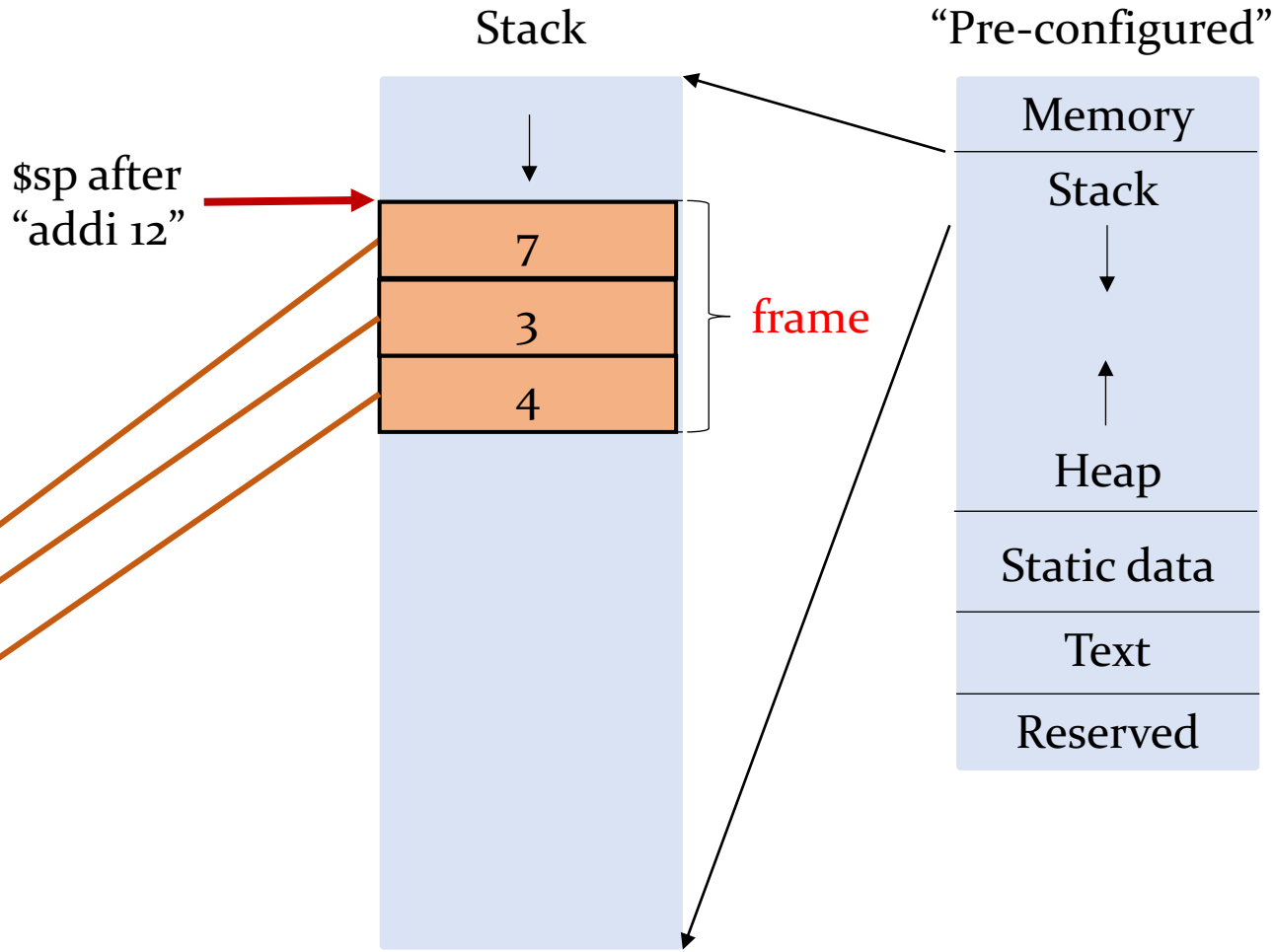


Visualizing the stack

```
compute_f1:
        addi $sp, $sp, -12
Setup   sw $s0, 8($sp)
        sw $s1, 4($sp)
        sw $s2, 0($sp)

        .....

Teardown lw $s0, 8($sp)
        lw $s1, 4($sp)
        lw $s2, 0($sp)
        addi $sp, $sp, 12
        jr $ra
```



Q: what happens if a function calls another function..?

Instruction Set Architecture (ISA)

What is the architectural state of the machine?

1) Registers
2) Memory
3) Program counter

Are pipeline registers part of the architecture or micro architecture?

Why?

What about hazards?

Why is the distinction so important?

MIPS Summary

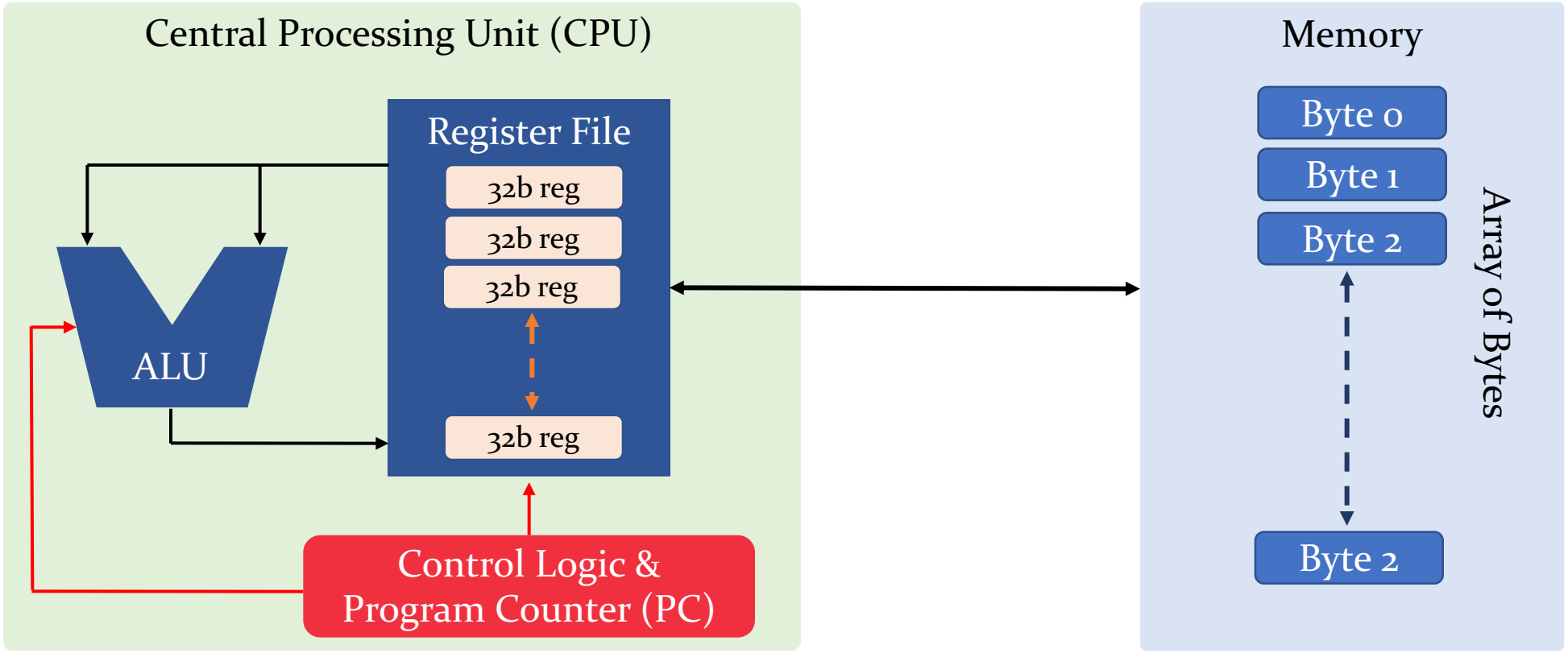
32b RISC architecture
- Fixed instruction length
- What if your simulators supported variable length?
- 32b registers and memory addresses
- Byte addressable memory.. How much?

Load-store architecture
- Only special instructions access memory
- Think about your simulators and the control signals
- Now, what if any instruction could access memory?

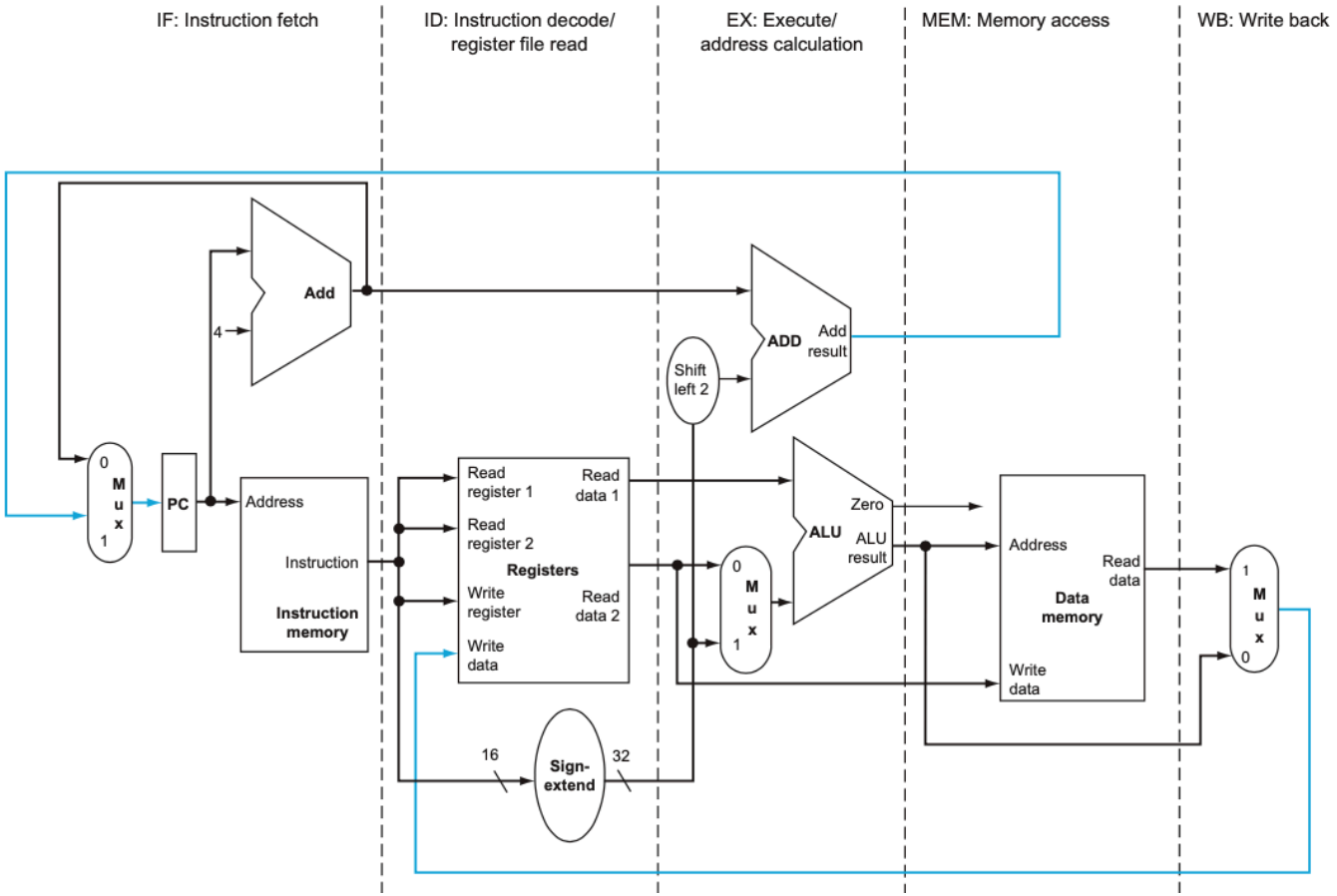
32 general purpose registers
- We will assume the simplified calling convention
- Program counter for finding next instruction
- Hi/lo registers for multiplication
- What if we had 128?

Design principle: smaller is faster.

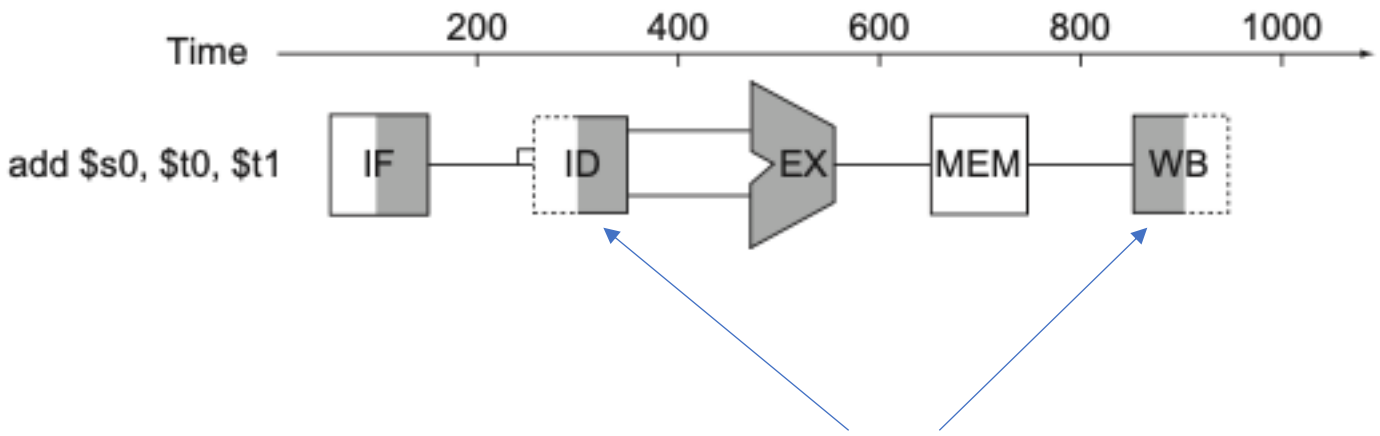
Architectural (ISA) view of a processor



Basic (datapath) pipeline machine



Basic pipeline diagram

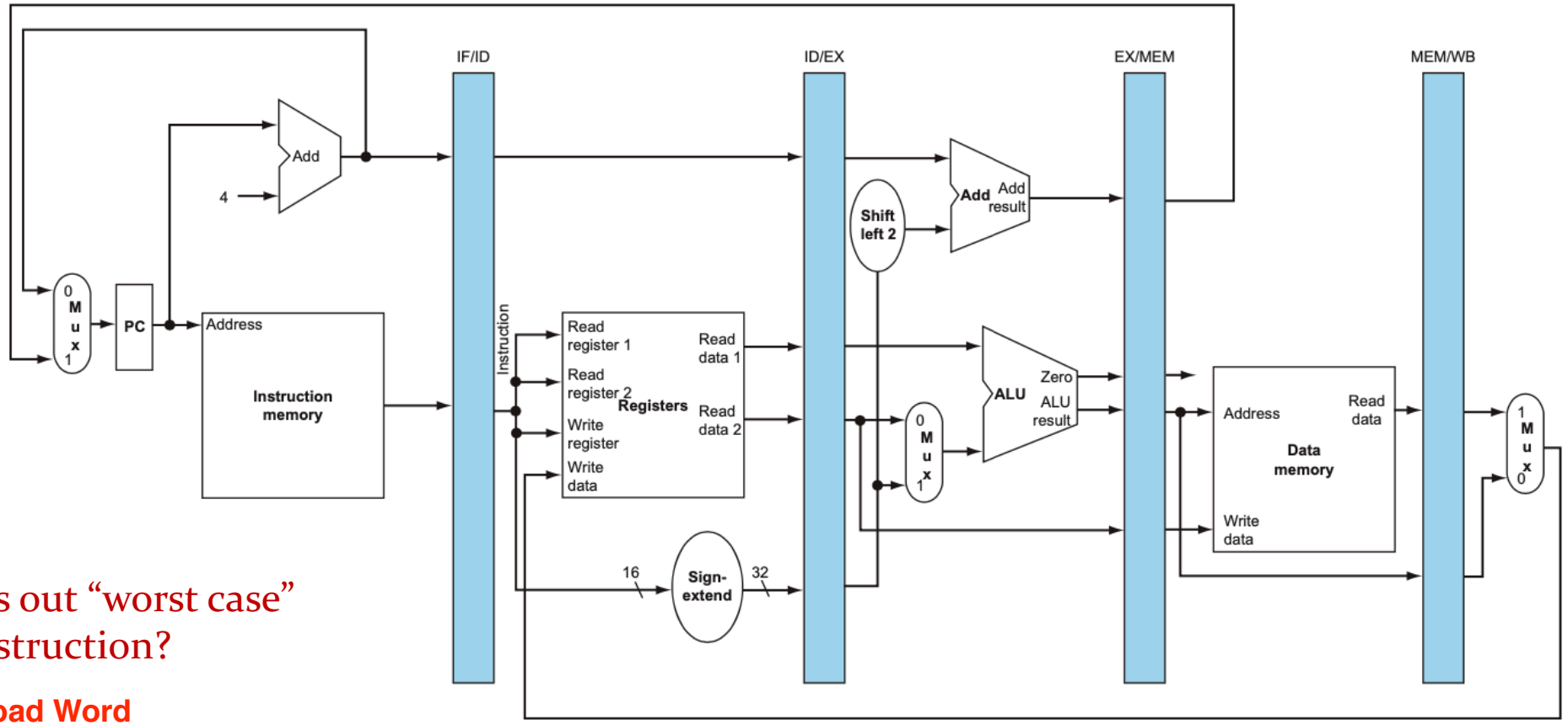


Architectural state written in first half
and read in second half of clock cycle!

More detailed pipeline diagram
Shaded sections imply use
Note: WB and ID both using RF

Question: what's the problem here
and how do we avoid it?

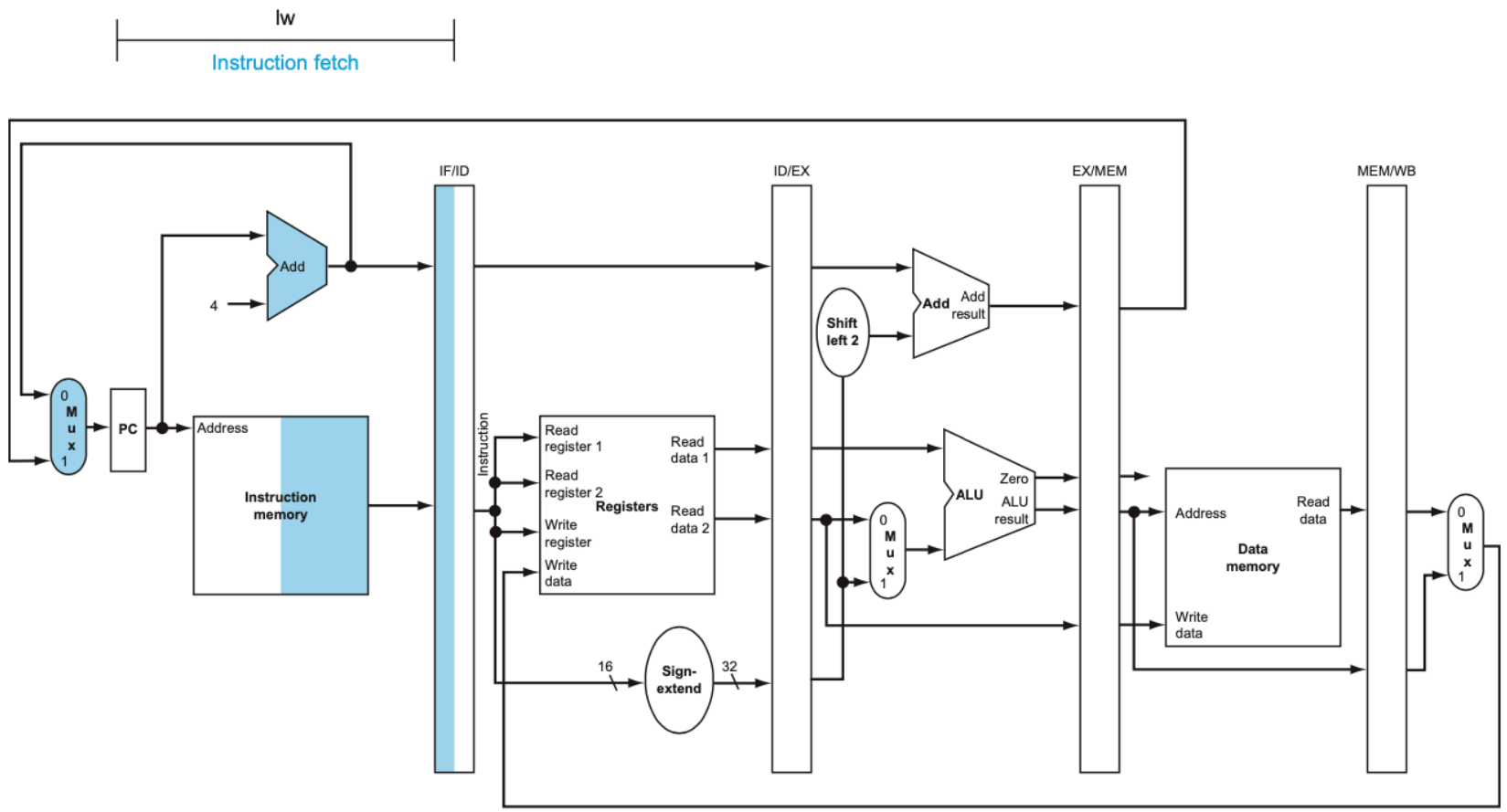
Pipelined datapath



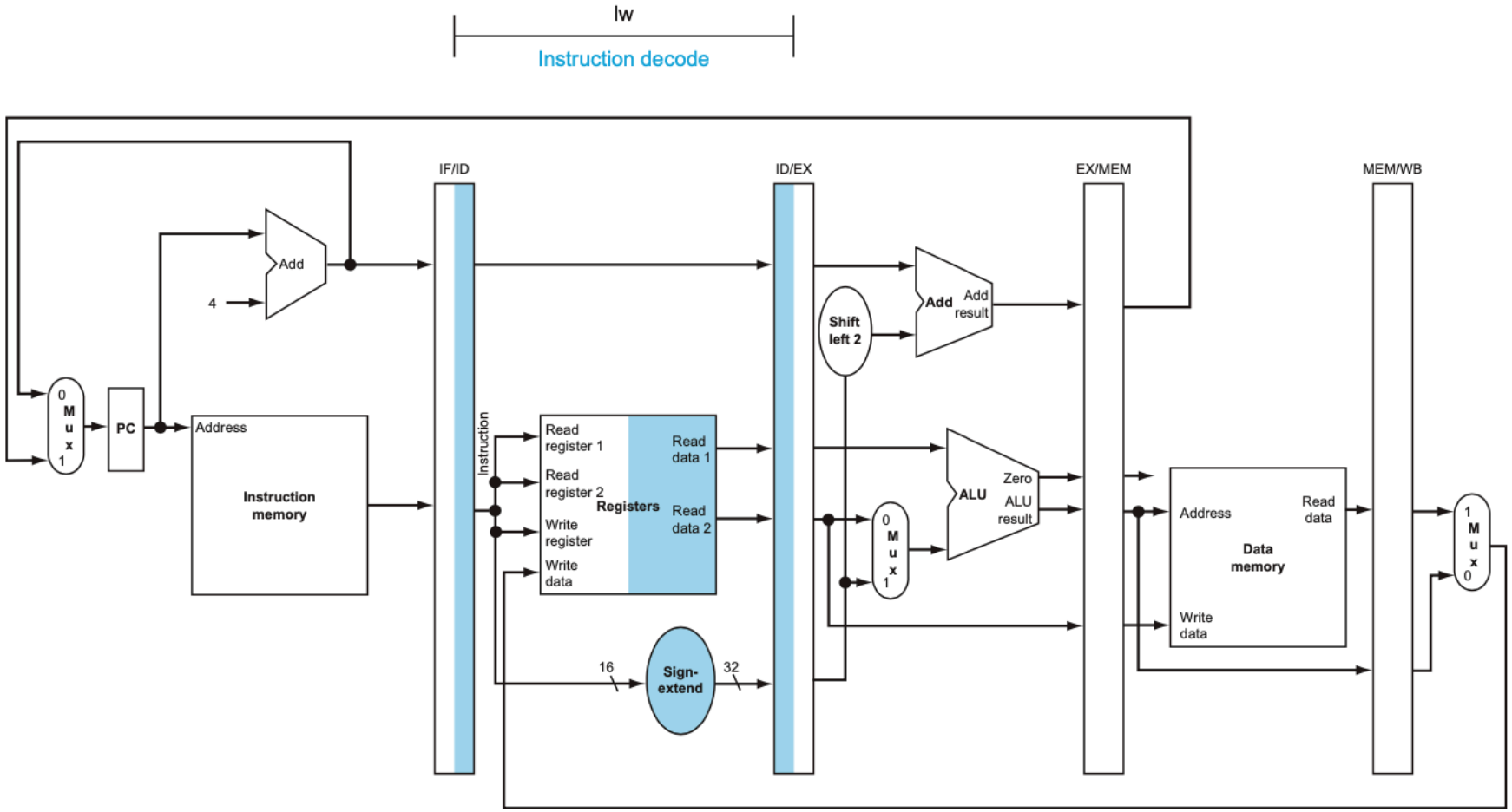
Q: What's out "worst case" instruction?

Load Word

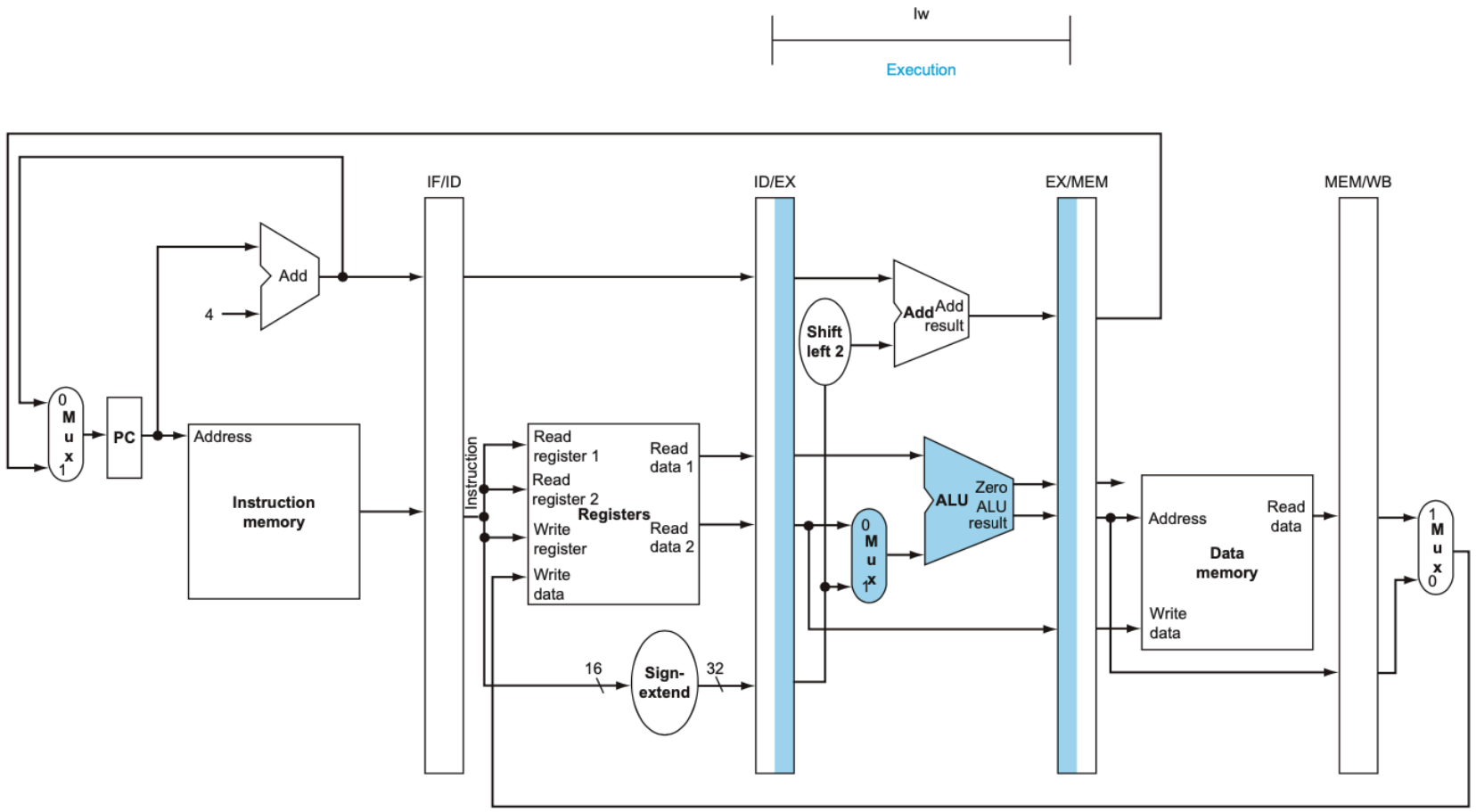
Pipeline: IF


lw
Instruction fetch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
16
Sign-extend
32
Address
Data memory
Read data
Write data

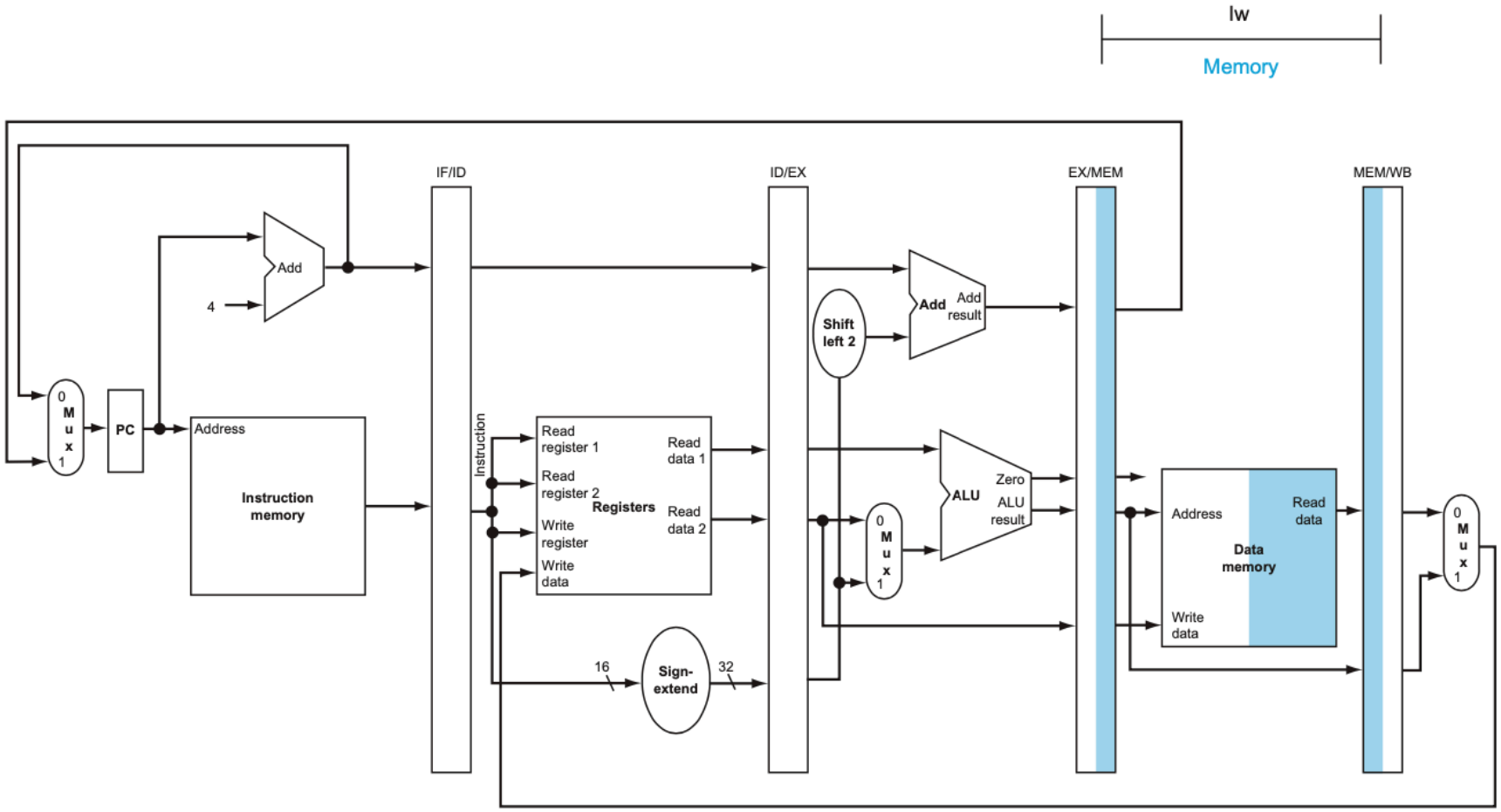
Pipeline: ID


lw
Instruction decode
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

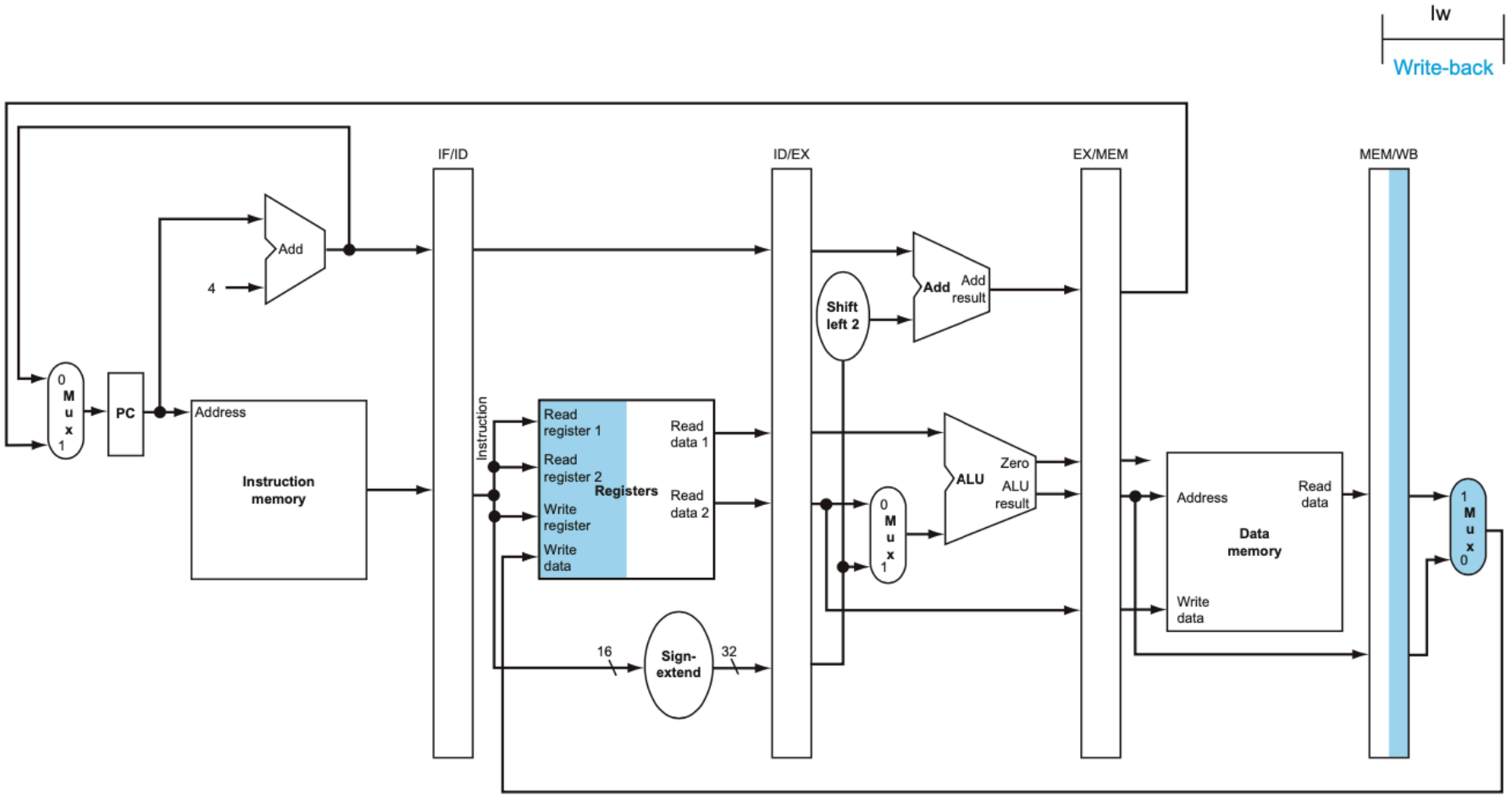
Pipeline: EX


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
16
Sign-extend
32
Address
Data memory
Read data
Write data

Pipeline: Mem

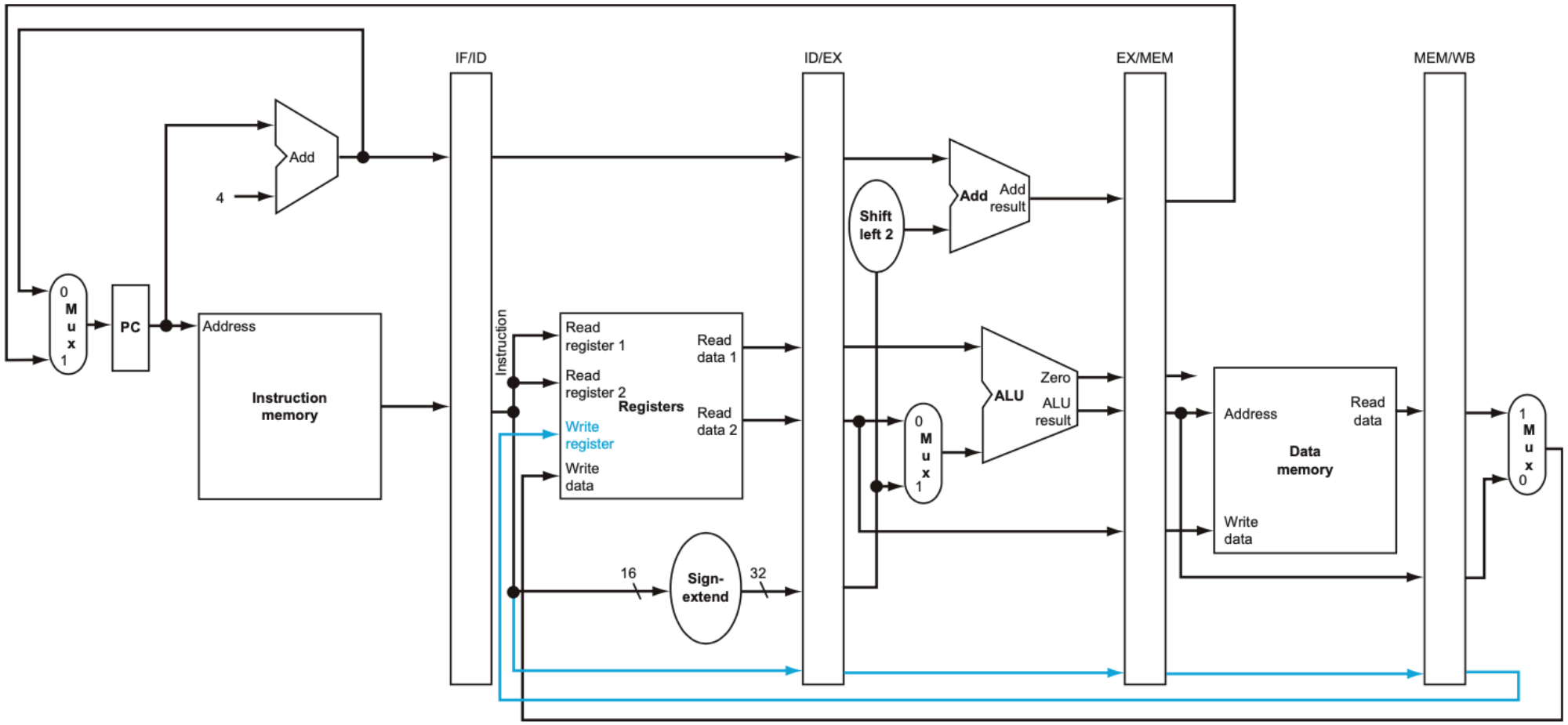


Pipeline: WB



What did we forget?

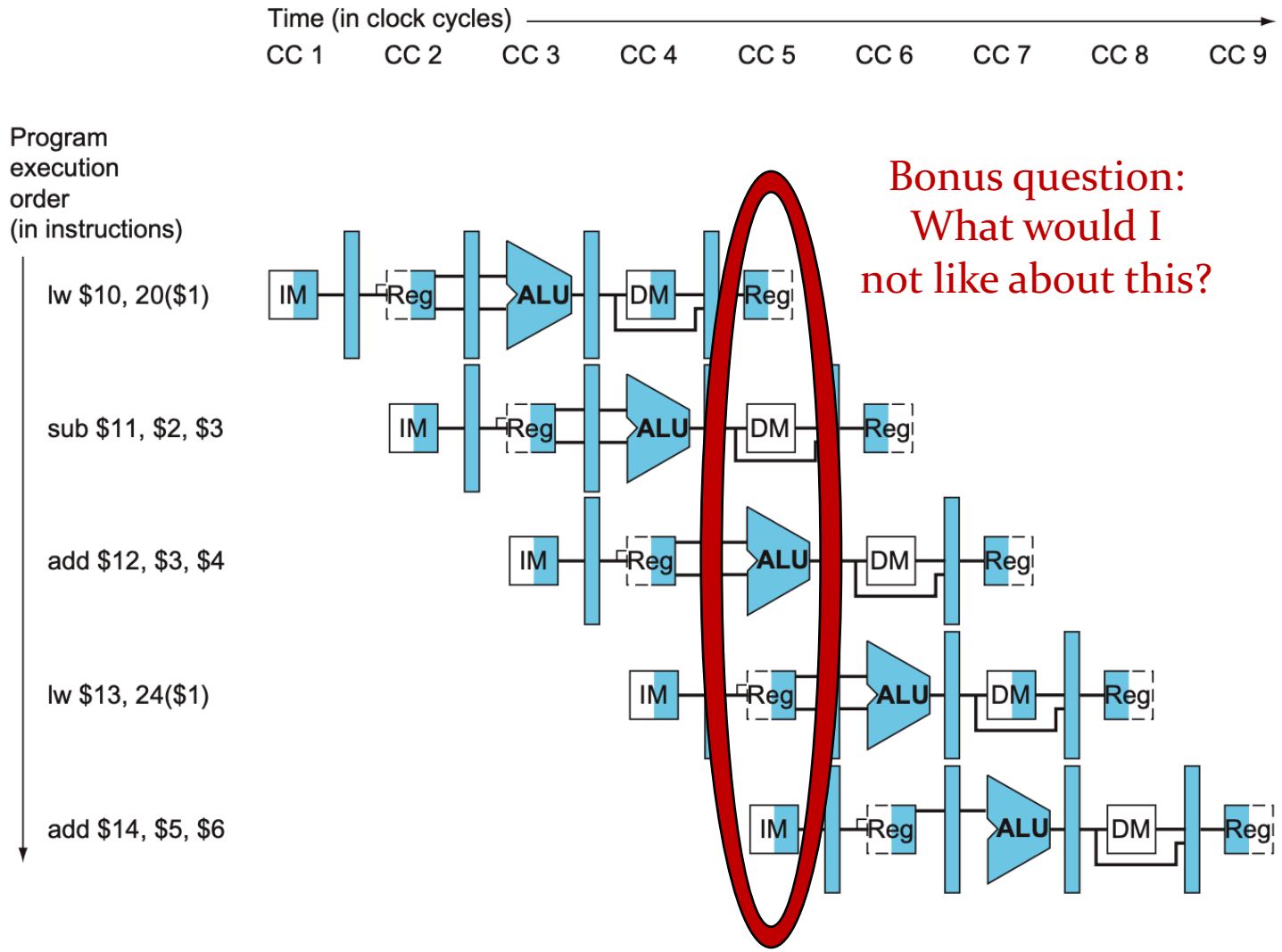
Where to writeback!



Pipeline diagram

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

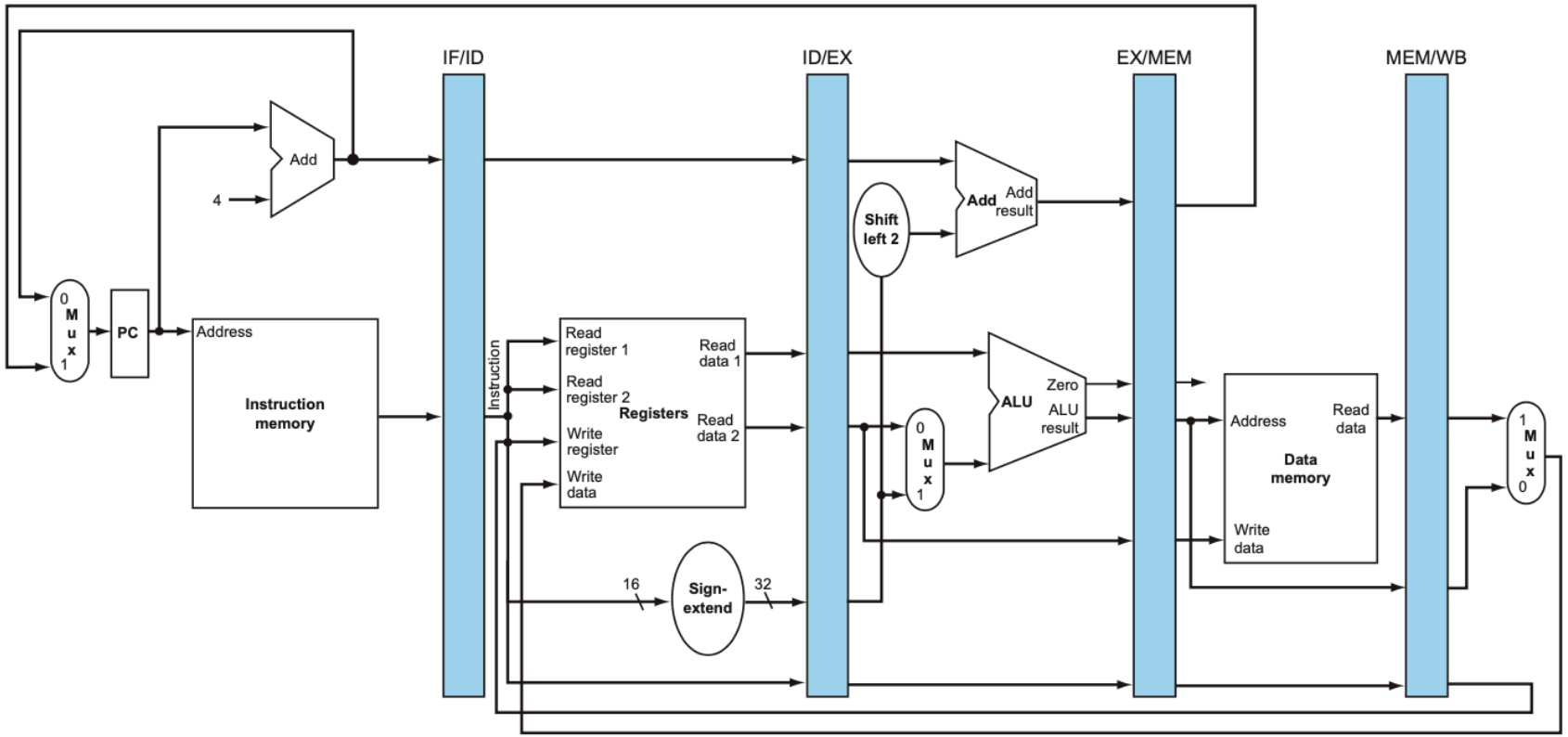
What's great about this program?



Bonus question: What would I not like about this?

Visualizing cycle (cc) 5

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

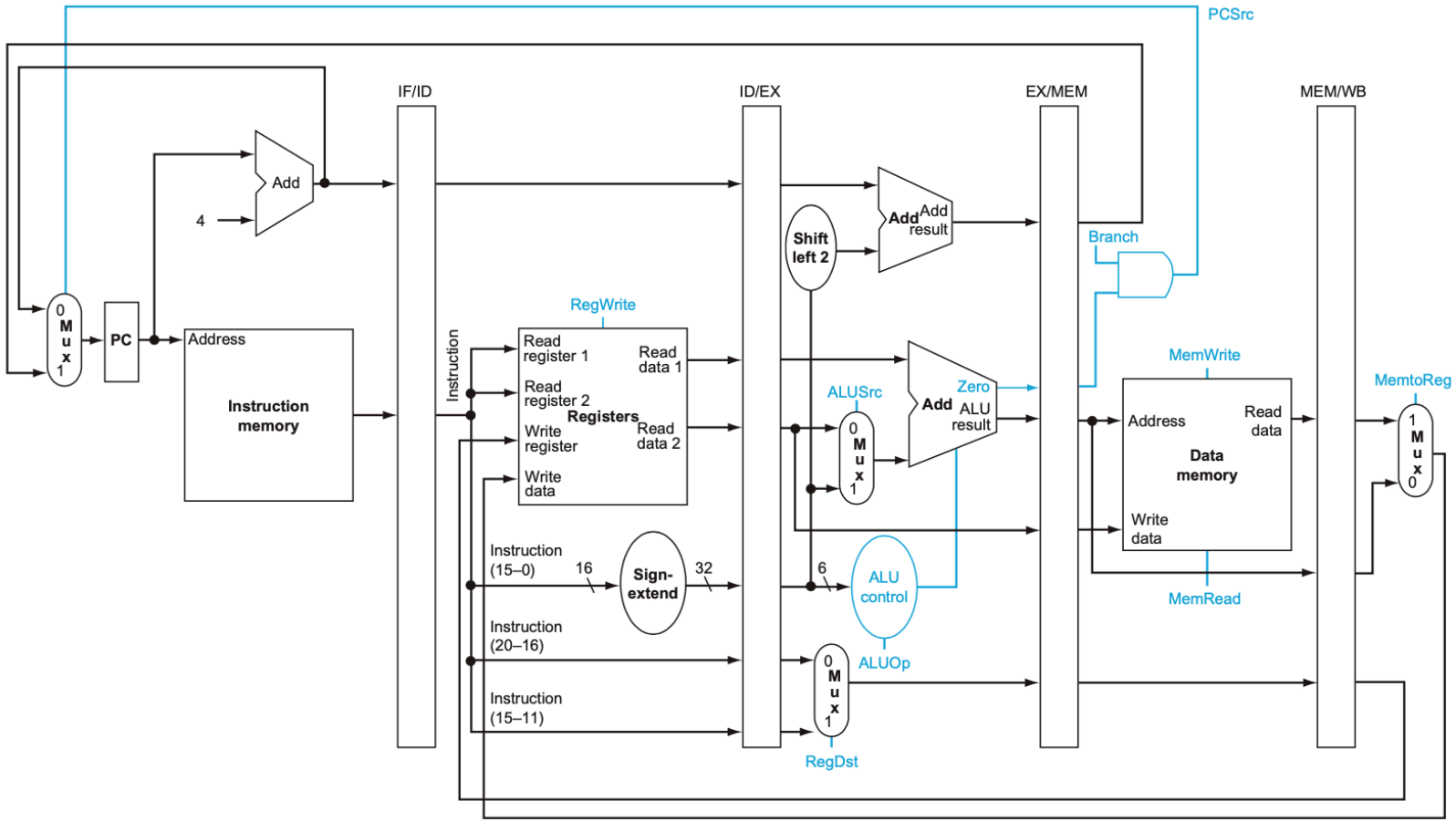


Why is pipelining so powerful?

What happens to:

- Instruction latency?
- Instruction throughput?
- Overall performance?
- Hardware utilization?
- Chip area?
- Chip power?
- Chip energy?

Pipelining control: when are signals needed?



IF stage:

None.

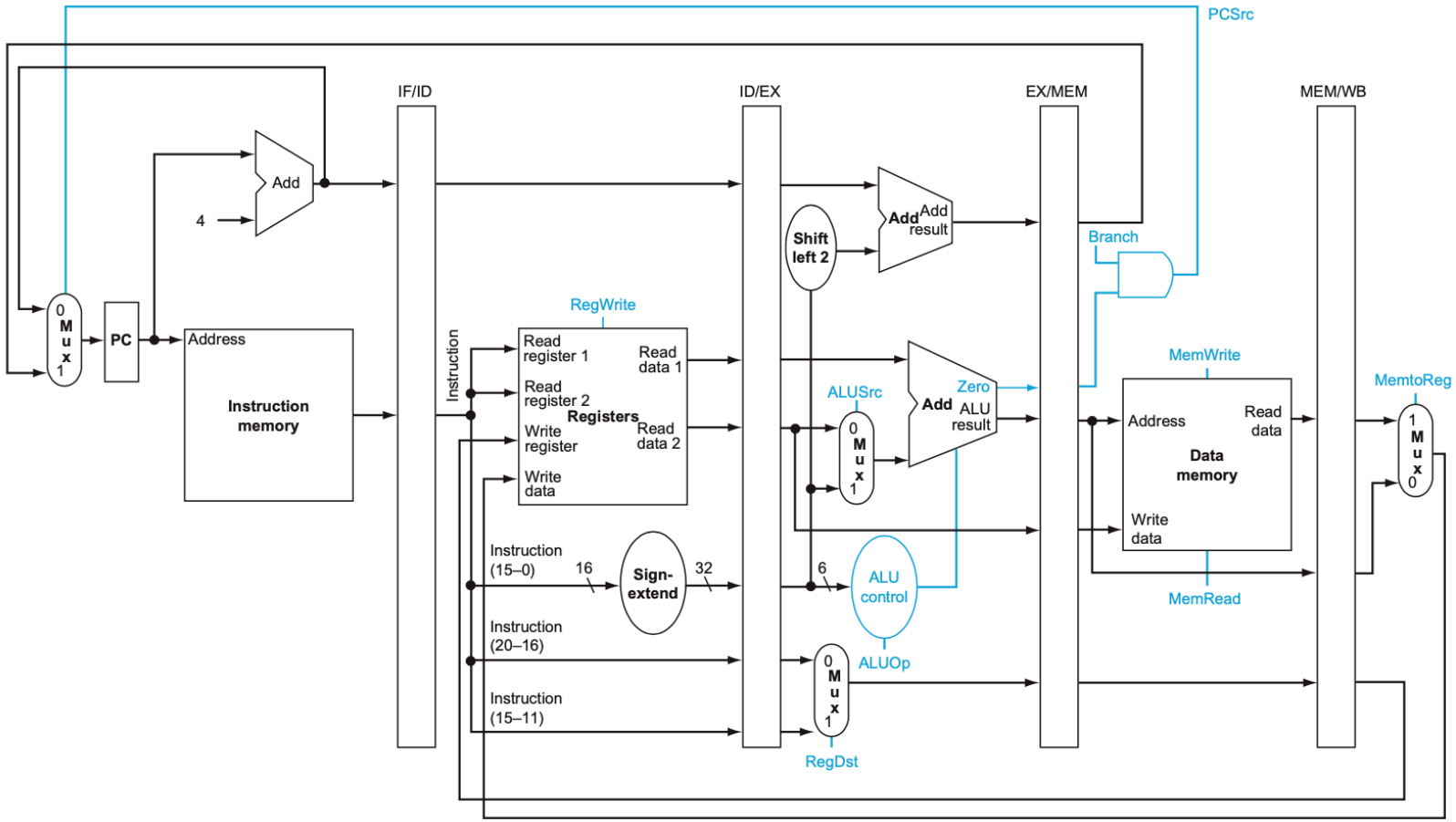
Always read PC memory.

ID stage:

None.

Same thing every cycle.

Pipelining control: when are signals needed?

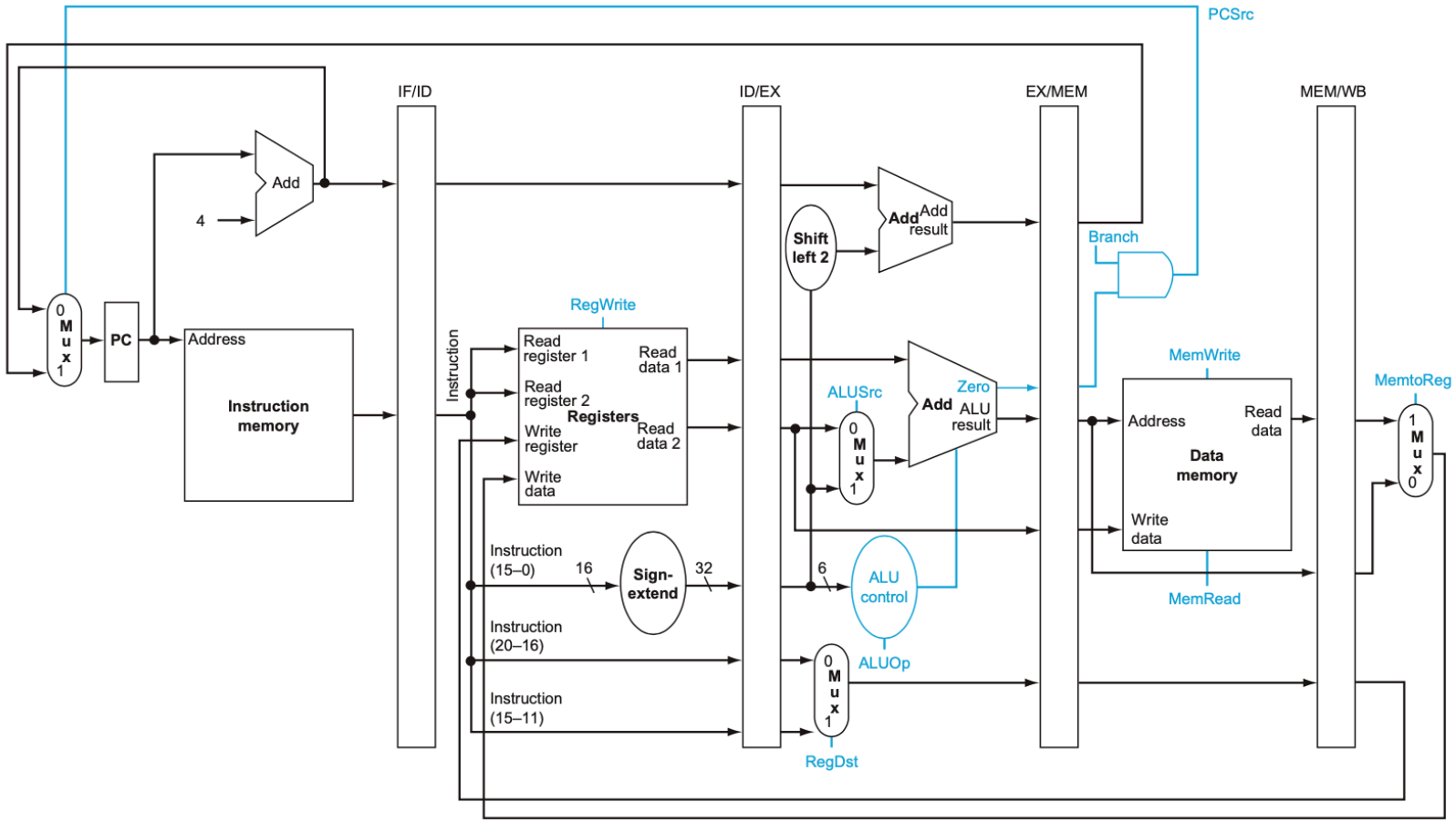


EXE stage:

1) regDest
2) ALUop
3) ALUsrc

regDst: where to write value
ALUop: what ALU should do
ALUsrc: reg2 or immediate value

Pipelining control: when are signals needed?



MEM stage:
1) Branch
2) MemRd
3) MemWr

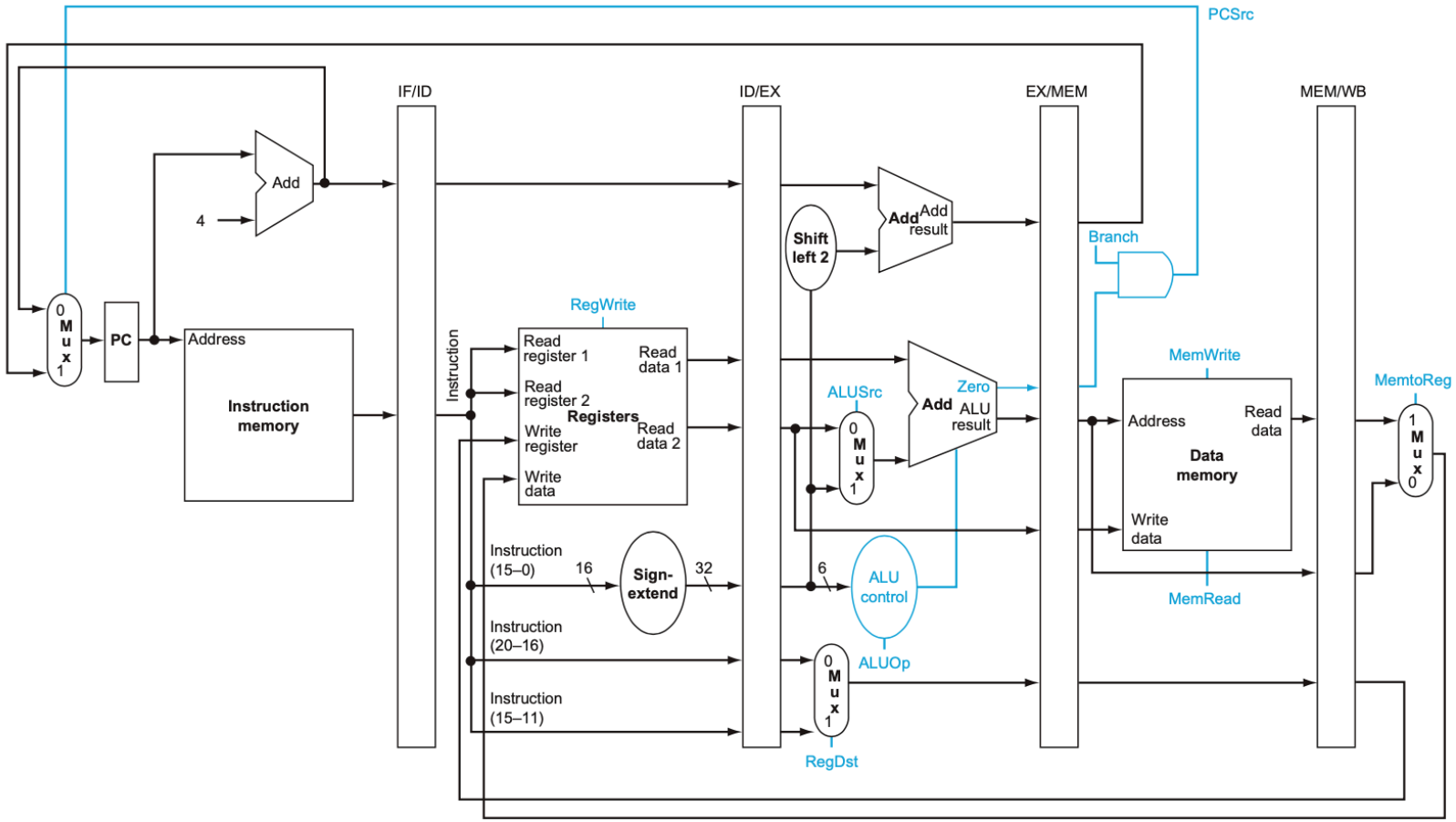
Branch: is branch instruction
MemRd: are we reading memory
MemWr: are we writing memory

Generates PCSrc signal to select next instruction.

What's going on here?

We'll talk about branches in depth in another class.

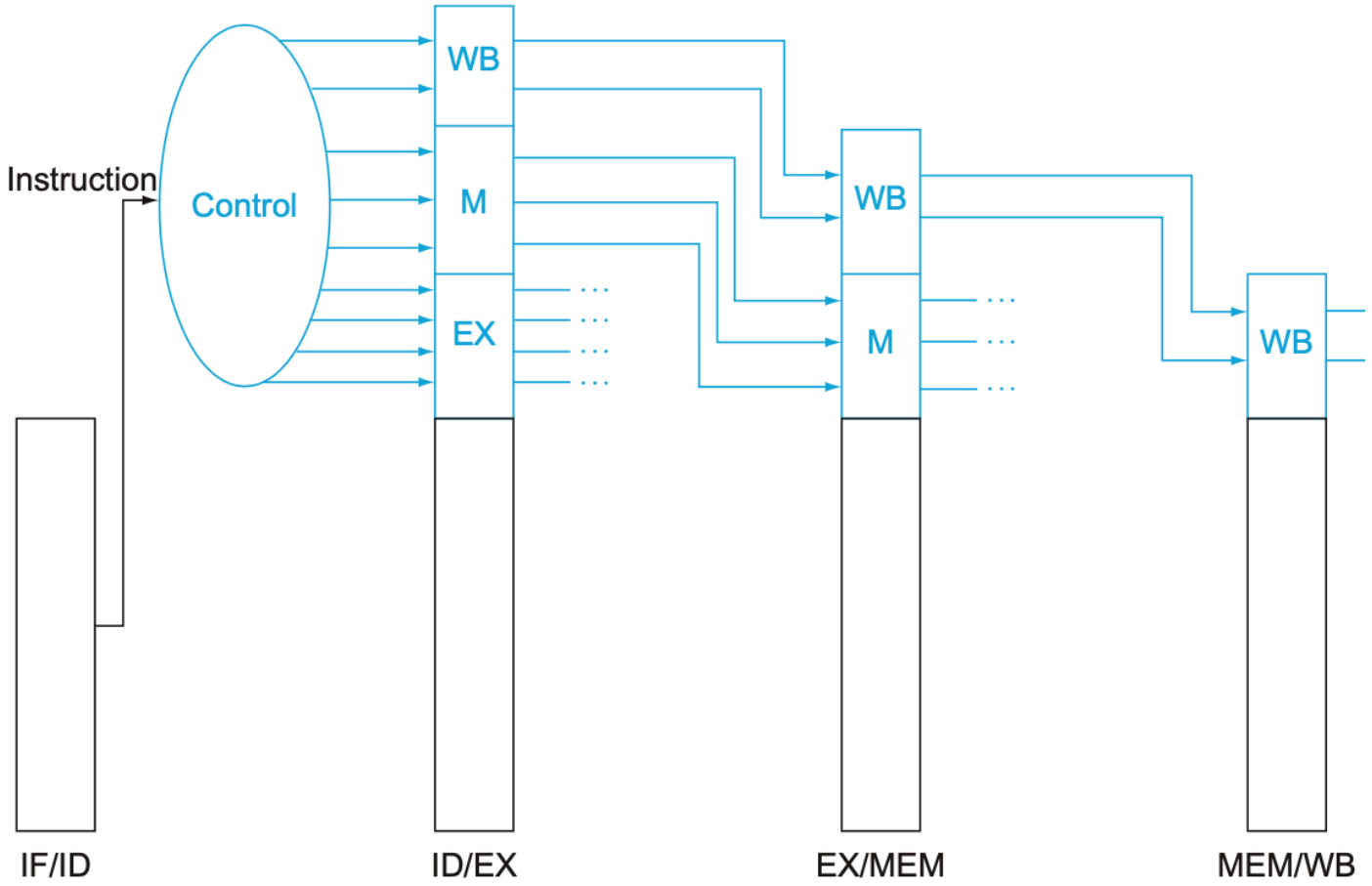
Pipelining control: when are signals needed?



WB stage:
1) MemtoReg
2) RegWrite

MemtoReg: which value to wr. back
RegWrite: enable RF writing

Computing “Control lines” in ID stage

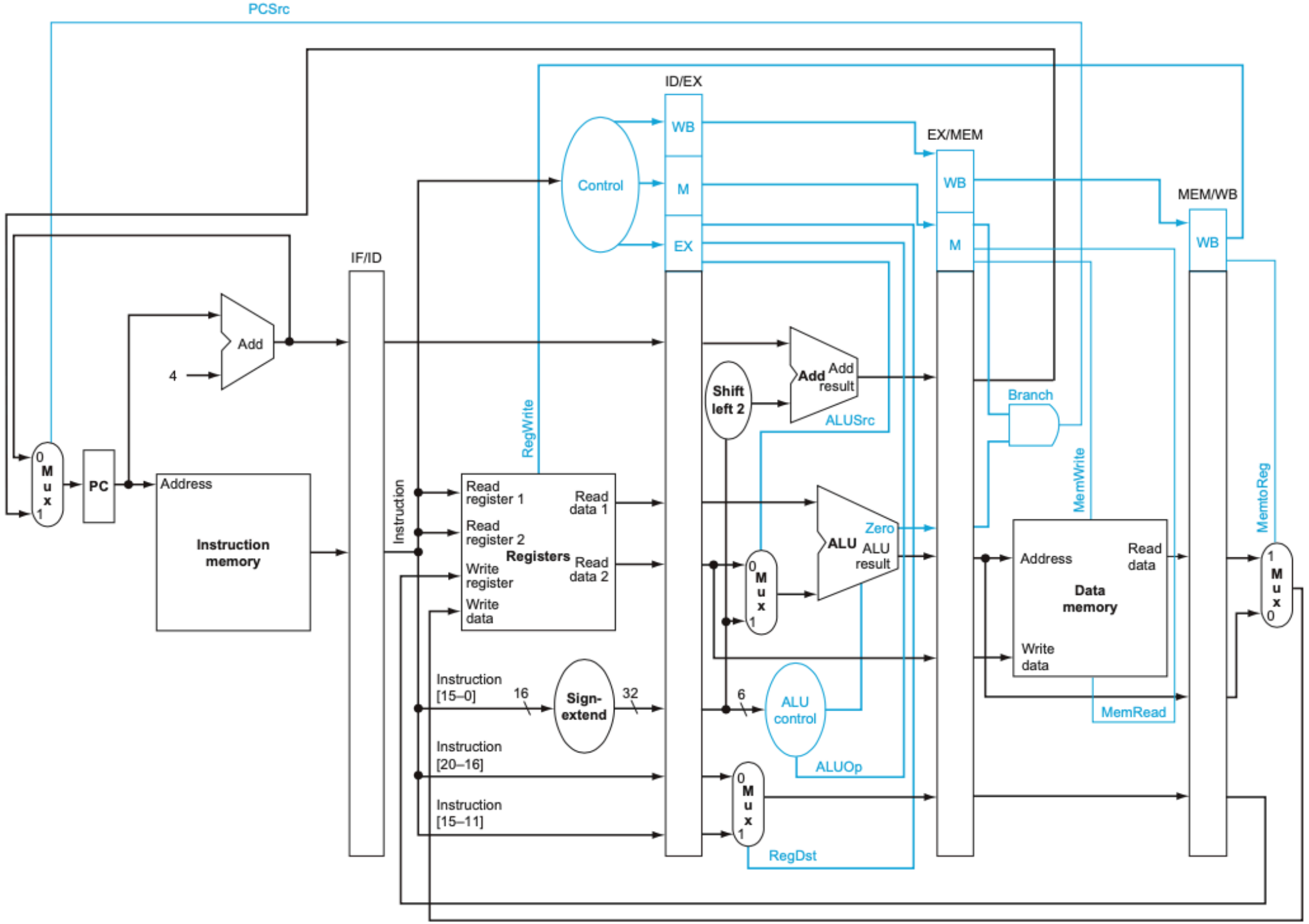


Control signals only needed in EX, MEM, WB stages

All signals depend on instr., which we know in decode (ID). Actually at IF, why not there??

So we compute all control and string it through the pipeline registers!

Our pipeline registers got bigger, But that’s fine b/c it’s not defined in the ISA.


PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Instruction [15–0]
16
Sign-extend
32
Instruction [20–16]
Instruction [15–11]
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
6
ALU control
ALUOp
RegDst
Branch
MemWrite
MemRead
MemtoReg
Address
Read data
Data memory
Write data

Hazards!

Hazard (formal) := “when the next instruction cannot execute in the following clock cycle”

Hazard (me) := “you need something you don’t have”

There are three of them..
Think about what goes into a program and how they execute

1) Structural : hardware resources
2) Data : need result from prior computation
3) Control : need to know where to go next

Hazards!

Hazard (formal) := “when the next instruction cannot execute in the following clock cycle”

Hazard (me) := “you need something you don’t have”

There are three of them..
Think about what goes into a program and how they execute

1) Structural : hardware resources
2) Data : need result from prior computation
3) Control : need to know where to go next

Hazards!

Hazard (formal) := “when the next instruction cannot execute in the following clock cycle”

Hazard (me) := “you need something you don’t have”

There are three of them..
Think about what goes into a program and how they execute

1) Structural : hardware resources : SOLVED!
2) Data : need result from prior computation
3) Control : need to know where to go next => Branch prediction (later)

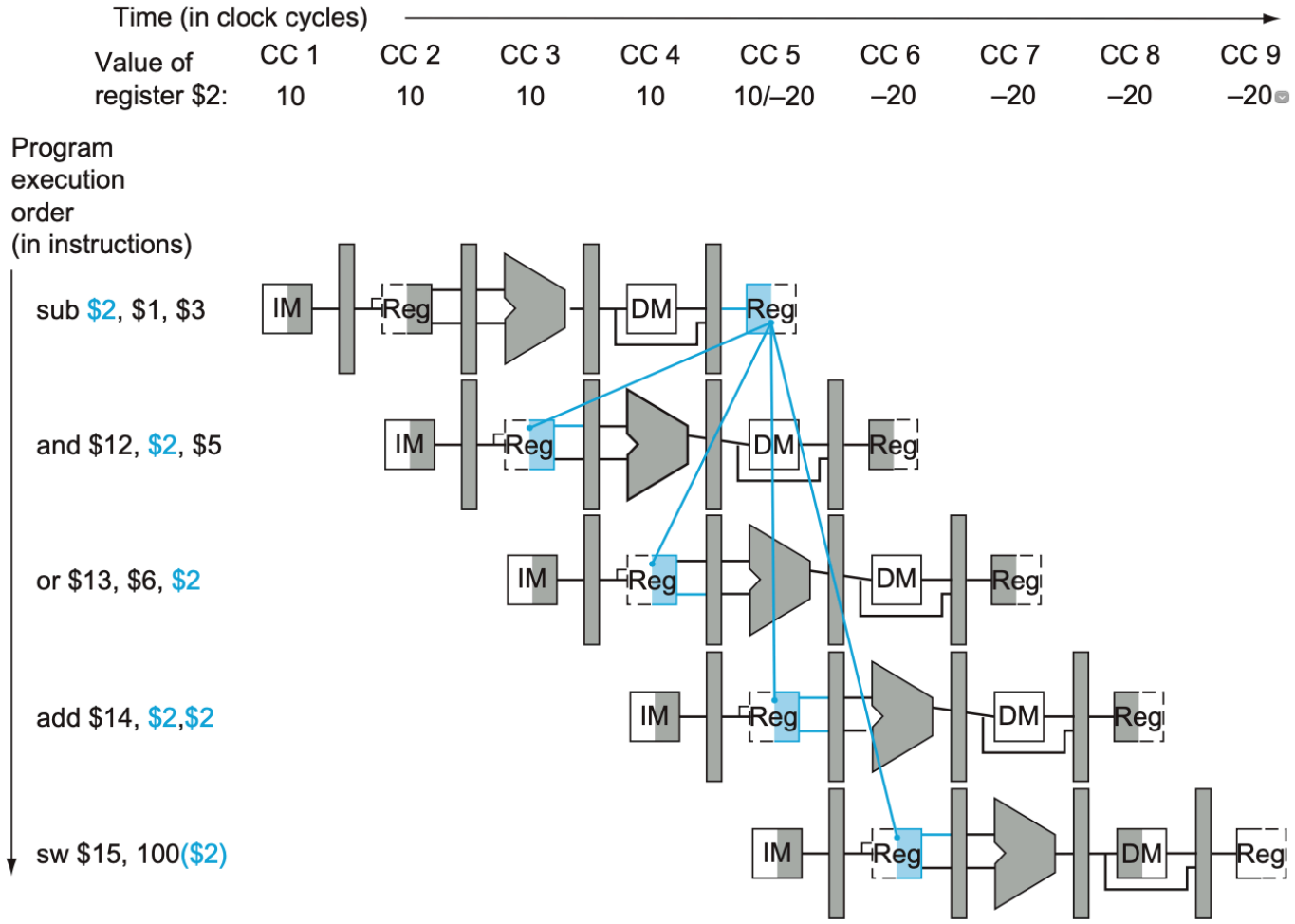
Program with dependencies

```
Sub   $r2, $r1, $r3
And   $r12, $r2, $r5
Or    $r13, $r6, $r2
Add   $r14, $r2, $r2
Sw    $r15, 100($r2)
```

Take a minute to Highlight all RAW Dependencies.

Program with dependencies

```
Sub   $r2, $r1, $r3
And   $r12, $r2, $r5
Or    $r13, $r6, $r2
Add   $r14, $r2, $r2
Sw    $r15, 100($r2)
```



Notation for describing detecting hazards

Notation: <StagePrev> / <StageNext> . <PipelineReg>

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

Ex-Ex hazards

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Mem-Ex hazards

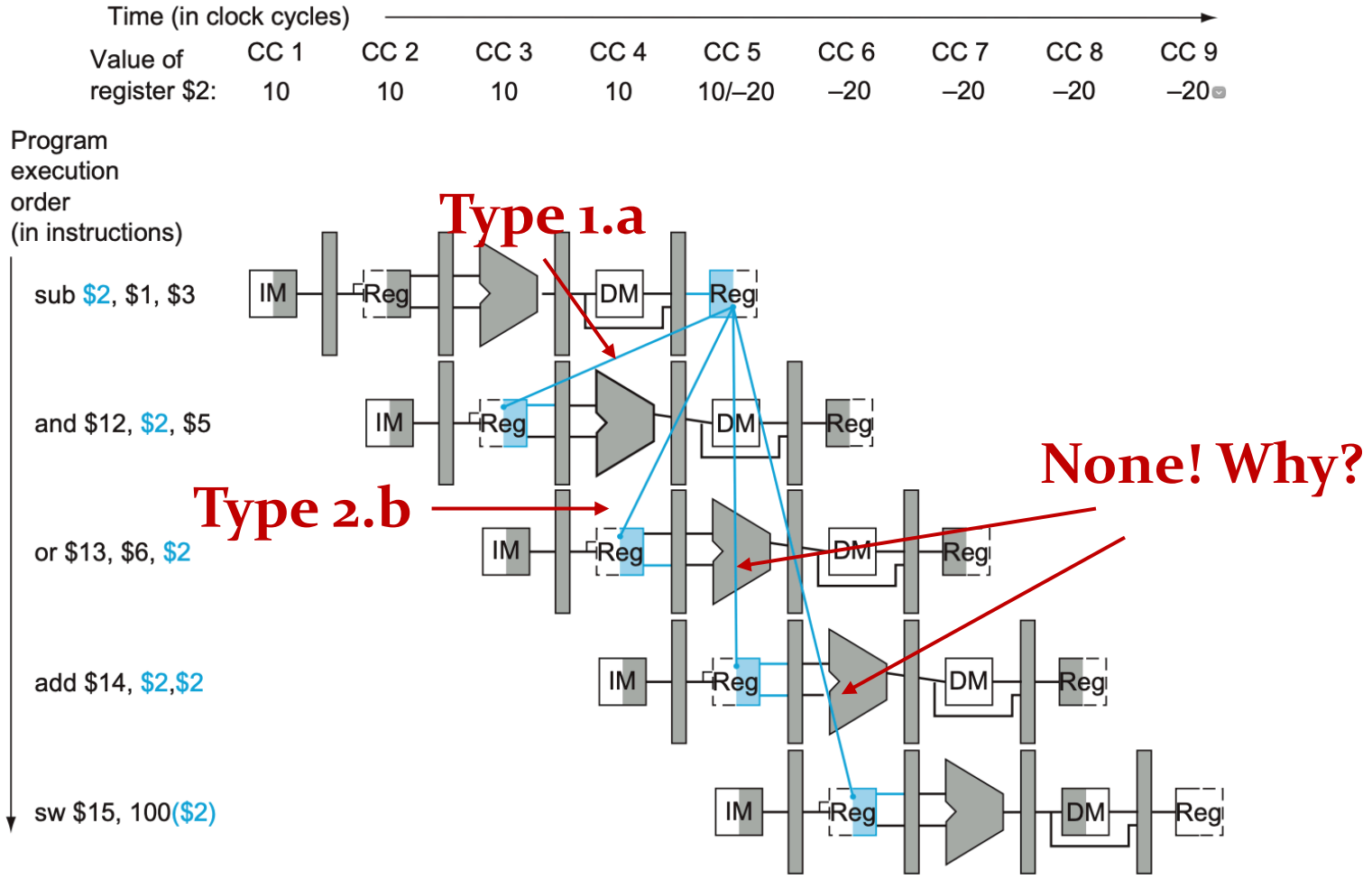
Program with dependencies

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

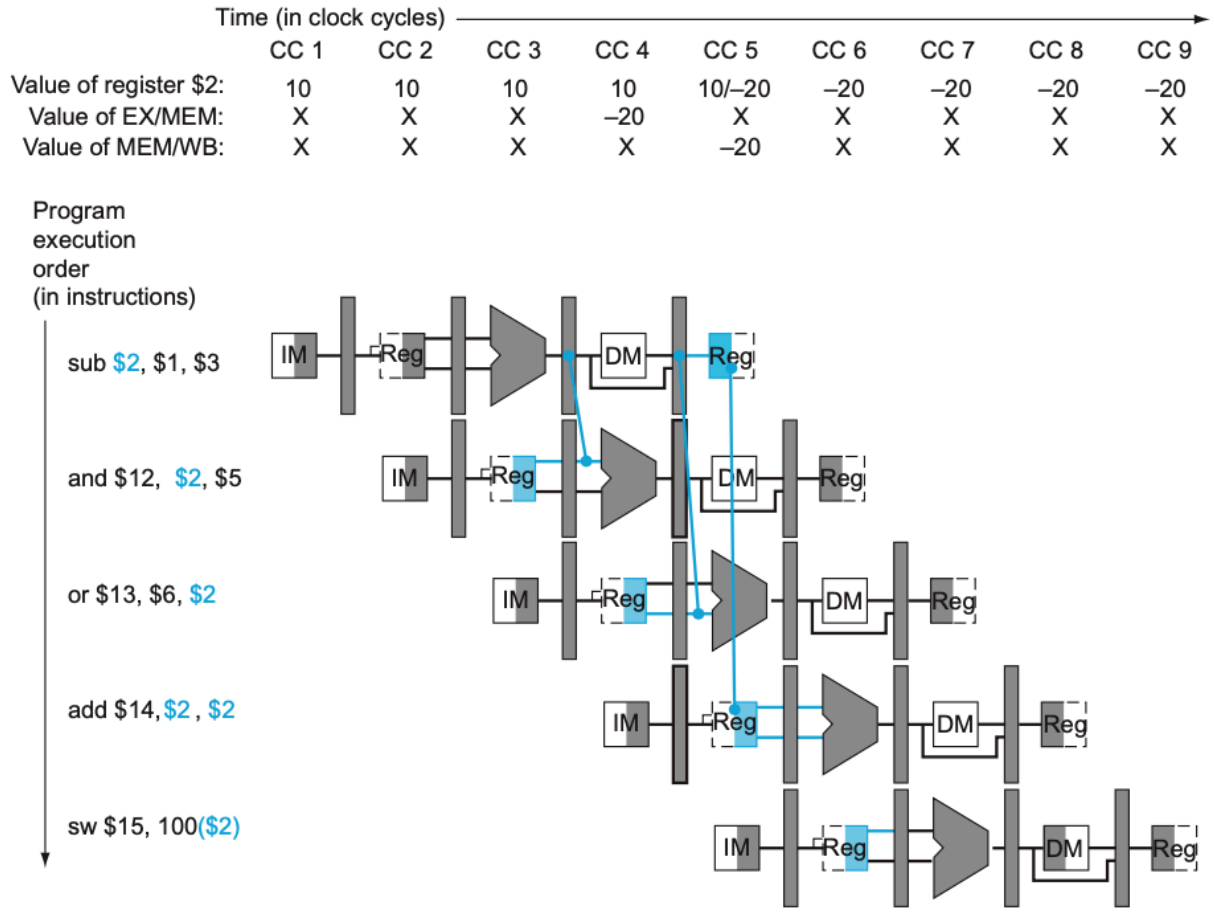
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

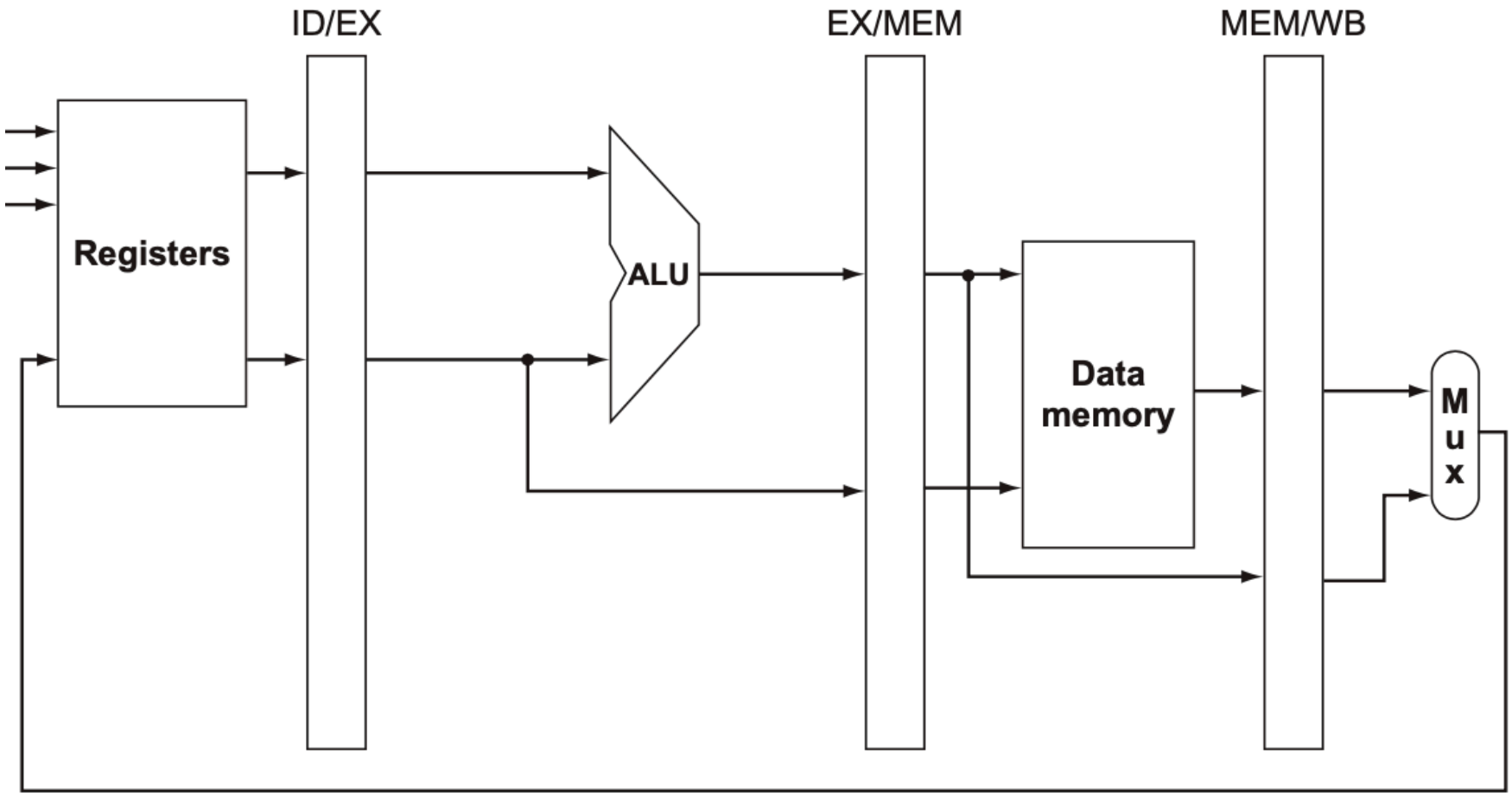
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt



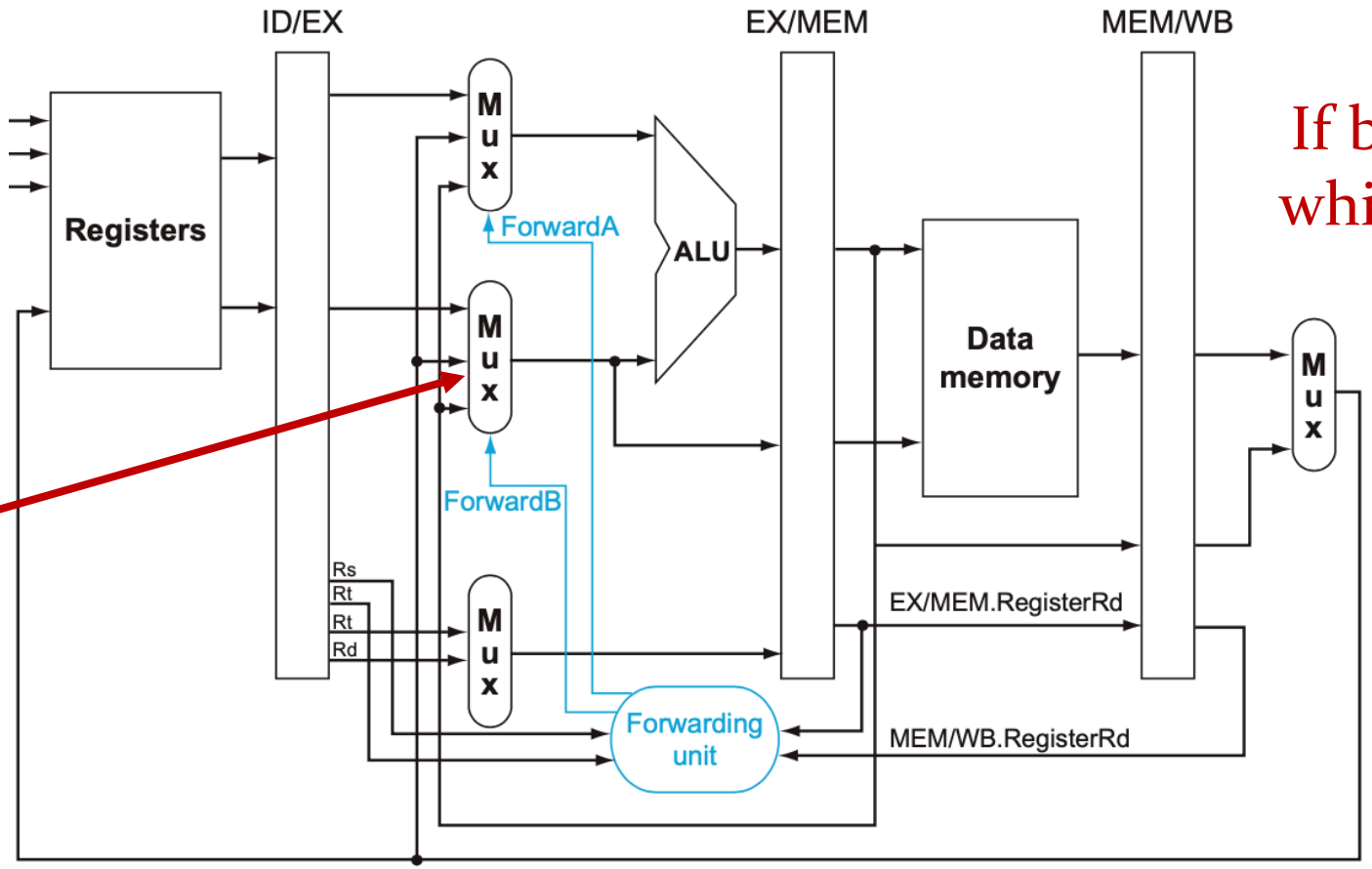
Solve register hazards with forwarding



Implementing forwarding in the pipeline



Forward datapath logic added

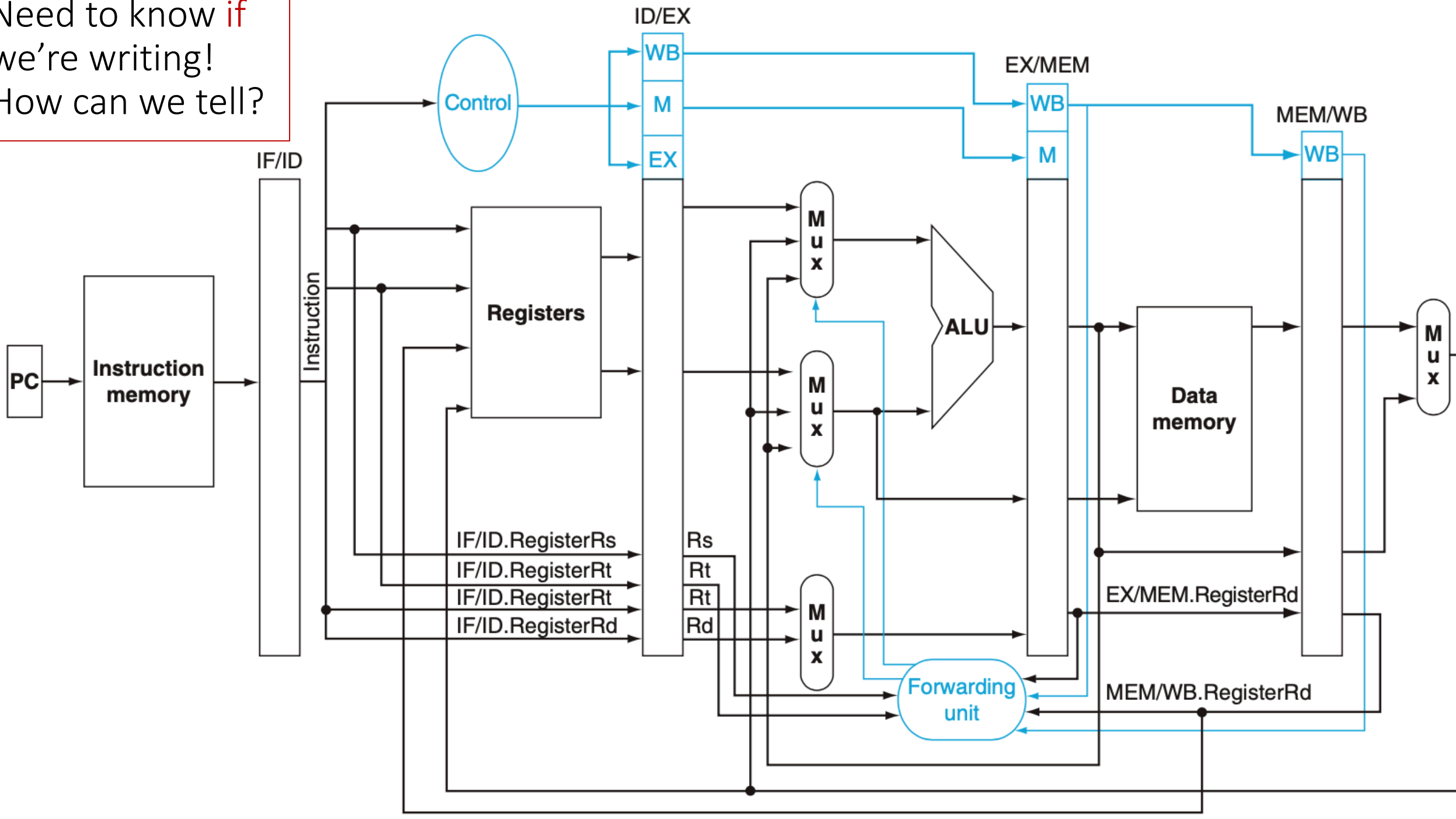


3 sources:
- ID/EX (from RF)
- EX/MEM (from ALU)
- MEM/WB (from Mem)

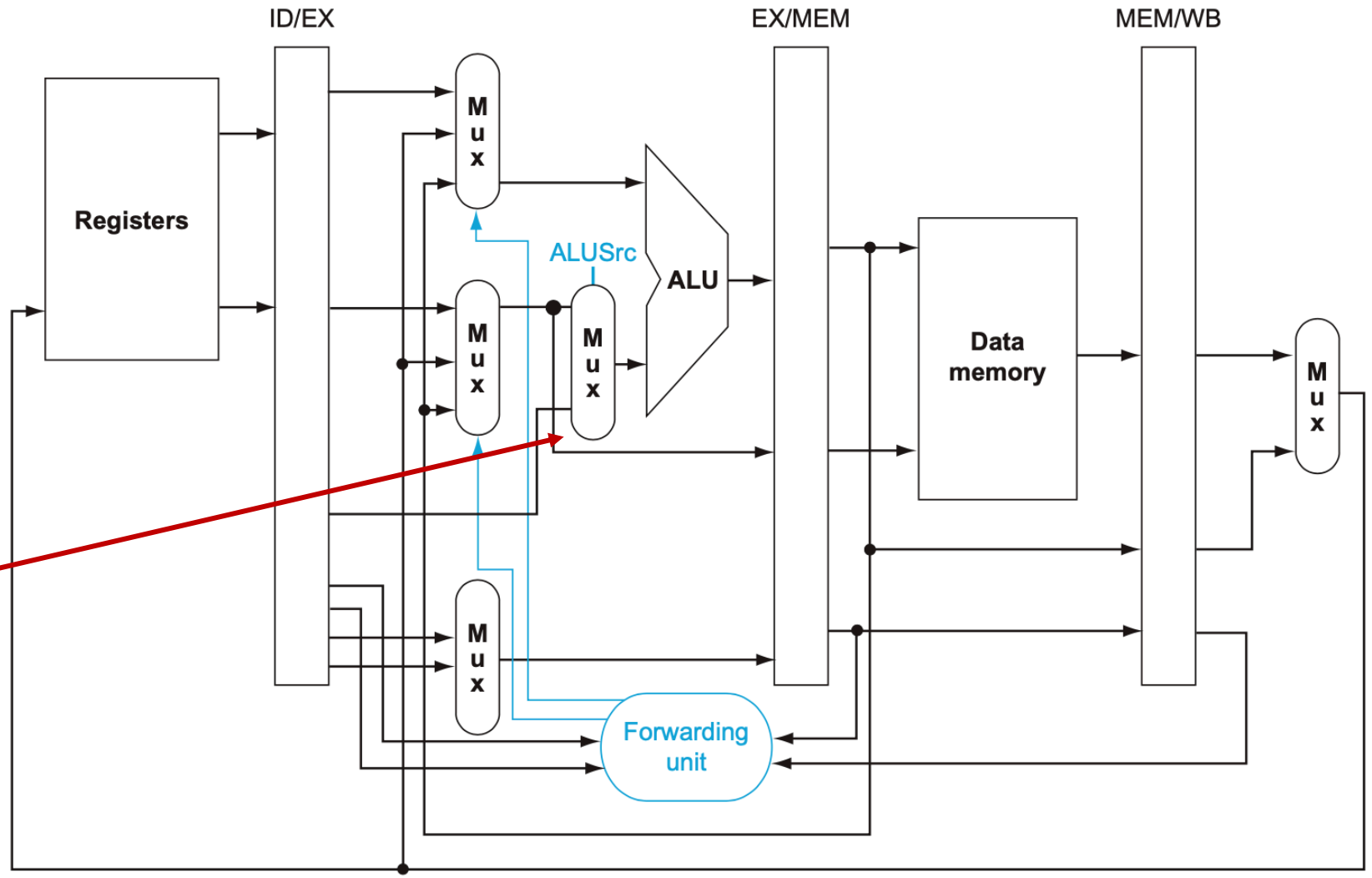
If both cases are true which should we use?

Q: What are we missing here? (Take a min)

Need to know if we're writing! How can we tell?



For completeness..



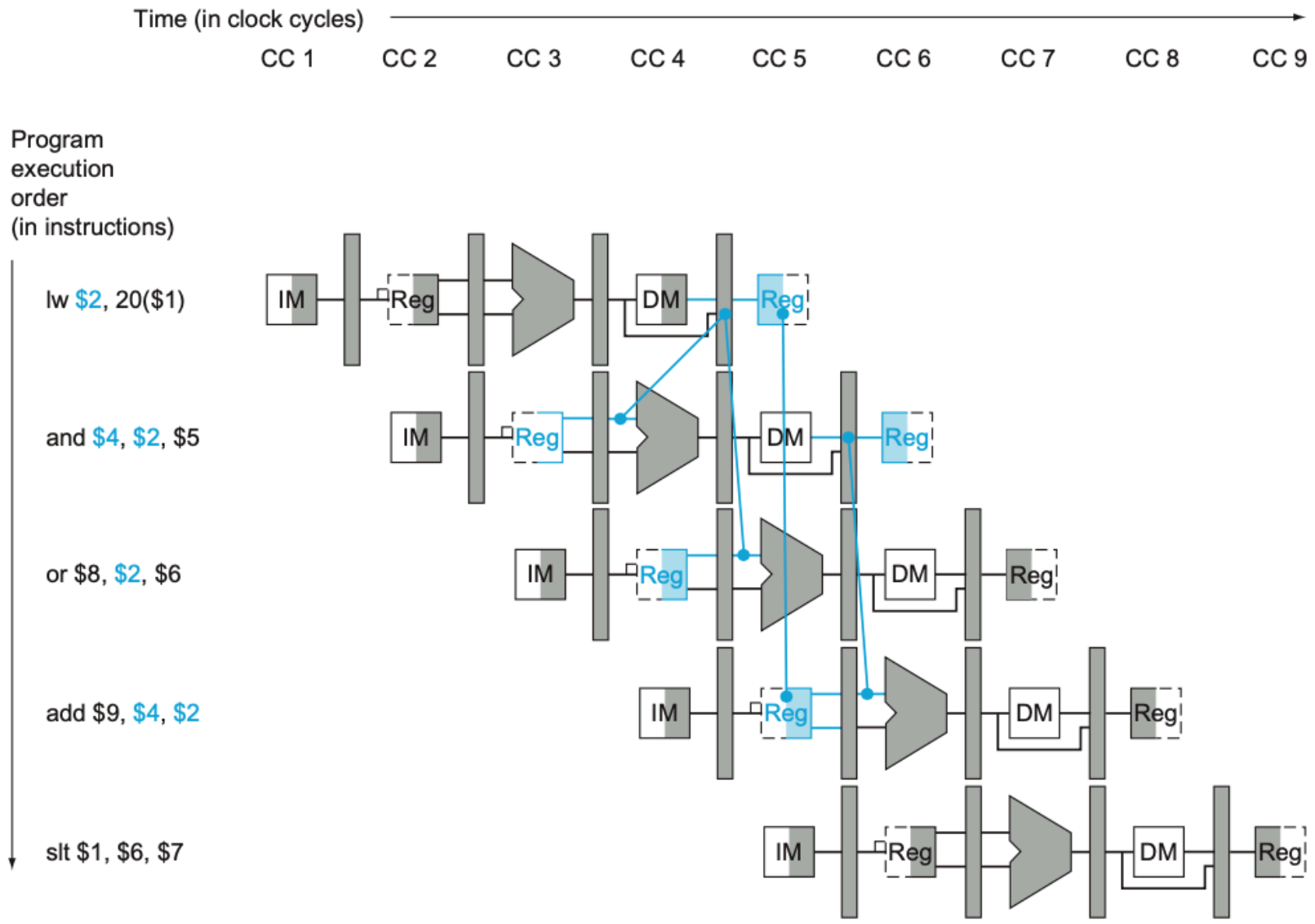
Add another MUX to choose between Immediate and register

When does forwarding fail?

Can't go backward in time.

With a LW instruction, we don't have the value until end of MEM stage.

If we have a dependence between lw and the next instruction, we must stall.



When do we have a hazard?

What stage do we detect the hazard in?
Decode!

The previous instruction was a load

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
      (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      stall the pipeline
```

The previous instruction Destination matches either Source registers.

Dealing with data (lw -> add) hazards

When that condition is met, we have to stall one cycle

First, we disable updating the PC and ID/RF pipeline registers

"Freeze" the front end to stop progressing until hazard dealt with

If we didn't stall PC, what would happen?

Second, insert a NOP into the ID/EX pipeline register

Can zero everything out, or just the control signals

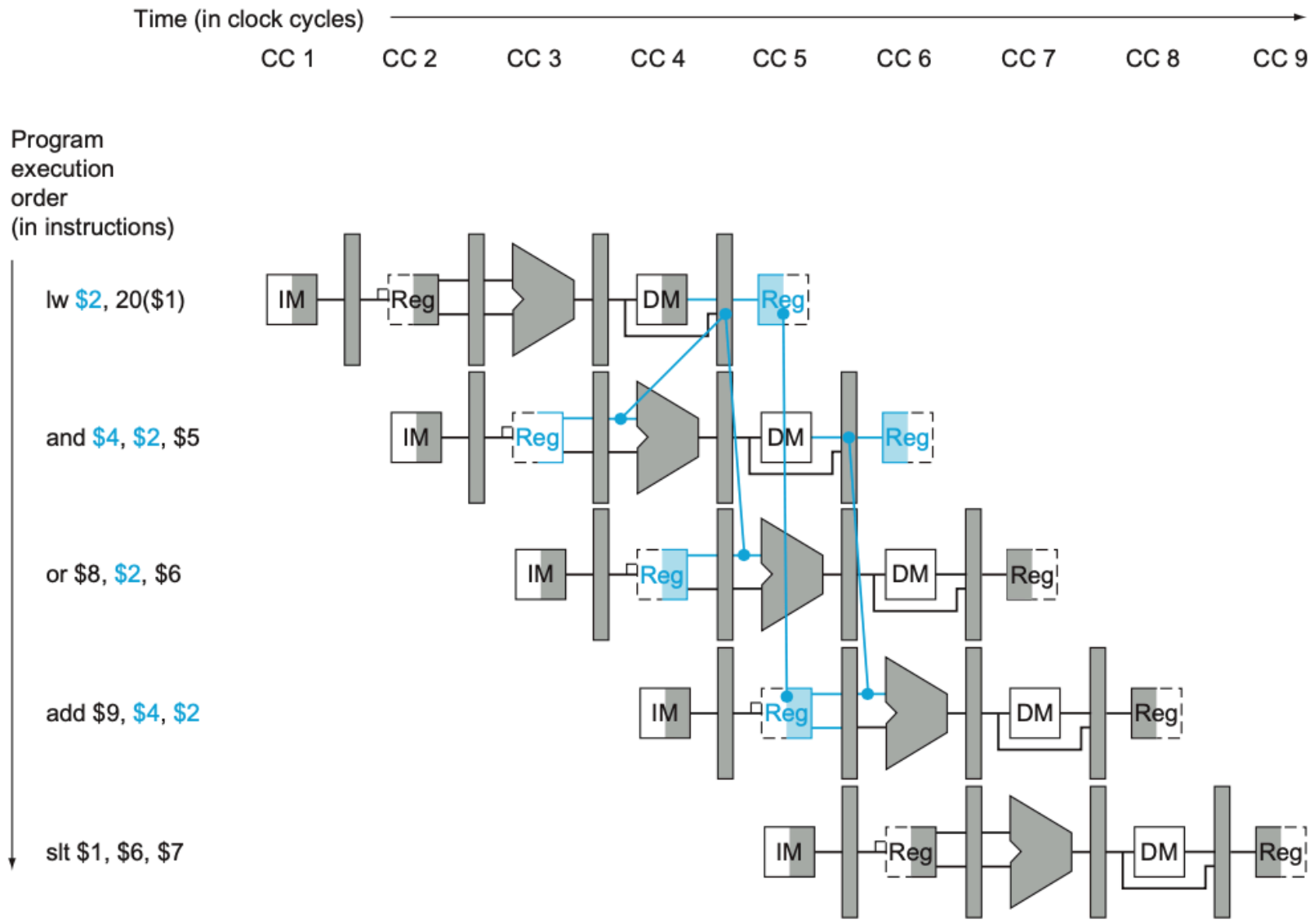
Why?

When does forwarding fail?

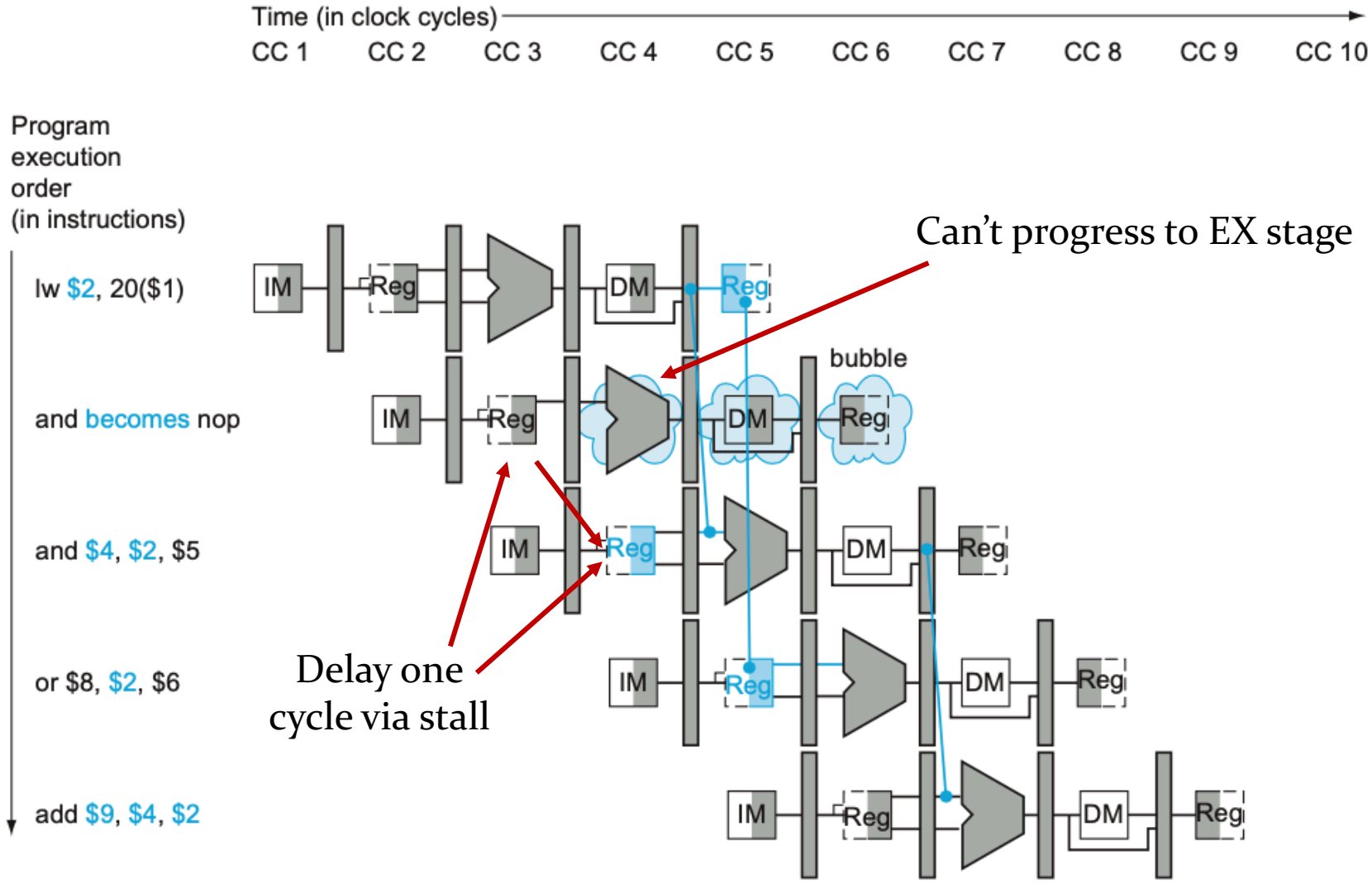
Can't go backward in time.

With a LW instruction, we don't have the value until end of MEM stage.

If we have a dependence between lw and the next instruction, we must stall.



Stall+NOP creates a bubble in the pipeline



QUIZ: Example no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2					
3					
4					
5					
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3					
4					
5					
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4					
5					
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5					
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!	Lw $r1, 0($r2)
6					
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!	Lw $r1, 0($r2)
6	Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!
7					
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!	Lw $r1, 0($r2)
6	Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!
7		Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!
8					
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!	Lw $r1, 0($r2)
6	Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!
7		Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!
8			Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3
9					
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!	Lw $r1, 0($r2)
6	Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!
7		Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!
8			Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3
9				Mul $r3, $r3	Add $r3, $r1, $r2
10					

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!	Lw $r1, 0($r2)
6	Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	BUBBLE!
7		Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!
8			Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3
9				Mul $r3, $r3	Add $r3, $r1, $r2
10					Mul $r3, $r3

Instructions: 4
Cycles: 10

Example
no MEM fwd

Program

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All dependencies

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

All hazards

```
Lw    $r1, 0($r2)
Add   $r2, $r1, $r3
Add   $r3, $r1, $r2
Mul   $r3, $r3
```

Cycle	IF	ID	EX	MEM	WB
1	Lw $r1, 0($r2)				
2	Add $r2, $r1, $r3	Lw $r1, 0($r2)			
3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	Lw $r1, 0($r2)		
4	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)	
5	Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!	Lw $r1, 0($r2)
6		Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3	BUBBLE!
7			Mul $r3, $r3	Add $r3, $r1, $r2	Add $r2, $r1, $r3
8				Mul $r3, $r3	Add $r3, $r1, $r2
9					Mul $r3, $r3

Instructions: 4
Cycles: 9

Performance metrics!

As we'll see now, comparing and computer
computer performance isn't as trivial as it sounds..

Revisit some definitions

Execution/Response time (Latency)

- Elapsed time between the start and completion of an event
- How long did did your project take?
- Where have we seen this in class?

Throughput (Bandwidth)

- Total amount of work done within some amount of time
- How many lines of code (or bugs) did you write per hour?

Some new definitions

Assume we're running a program on a machine X

Performance:

Performance(X) = 1 / Execution time (x)

It's the inverse of the runtime.

"X is n times faster than Y"

Performance(X) / Performance(Y) => speedup factor of n

We evaluate a computer's performance using benchmarks

- (Real) Programs
  - In the form of collection of programs
  - E.g., SPEC, Winstone, SYSMARK, 3D Winbench, EEMBC
- Kernels
  - Small key pieces of real programs
  - E.g., Livermore Fortran Loops Kernels (LFK), Linpack
- Modified (or scripted)
  - To focus on particular aspects (e.g. remove I/O, focus on CPU)
- "Toy" Benchmarks
  - Produce expected results
  - Why are these so important?
- Synthetic Benchmarks:
  - Representative instruction mix
  - E.g., Dhrystone, Whetstone

Important for:

Understanding architectural and microarchitectural design trade-off

Competitive analysis of real products

Check out Mlperf!

On averages..

Arithmetic mean: Use when using the same units

$$\frac{1}{n}\sum_{i=1}^{n} Time_i \quad \text{or} \quad \frac{1}{n}\sum_{i=1}^{n} Weight_i * Time_i$$

Harmonic mean: Use when using data values are ratios of variables with different rates

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{Rate_i}} \quad \text{or} \quad \frac{n}{\sum_{i=1}^{n} \frac{Weight_i}{Rate_i}}$$

Why Harmonic Mean? Driving example

30 mph for the first 10 miles
90 mph for the next 10 miles

Average speed? (30+90)/2 = 60 mph??
Wrong!

Average speed = total distance / total time
(10+10)/(10/30 + 10/90) = 45 mph

Make the common case fast

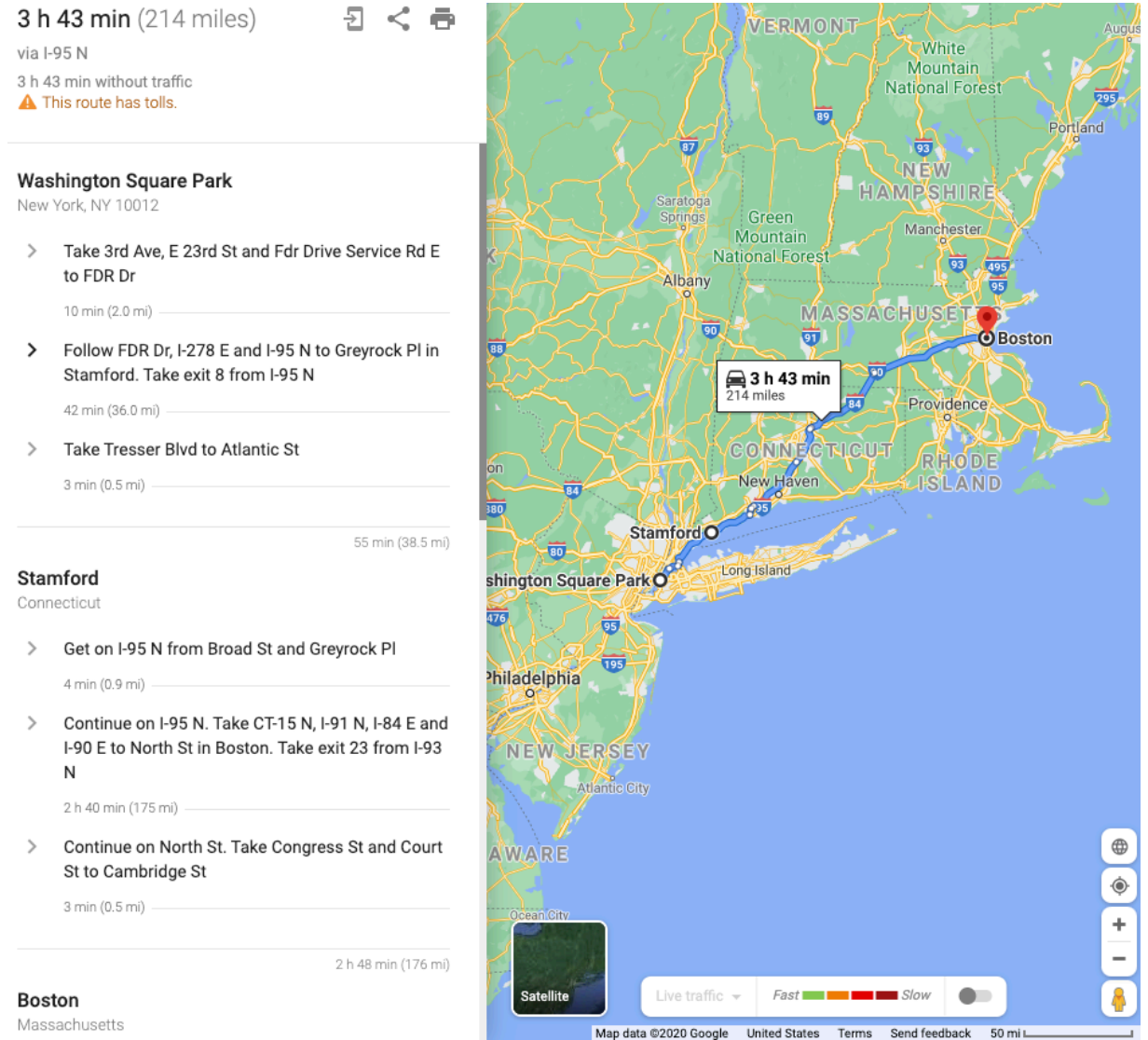
Drive NYC -> Boston
- 130 mph NYC -> Stamford (38 mi)
- 65 mph Stamford -> Boston (176 mi)

- First takes 38/130 = 17.5 minutes
- Second takes 176/65 = 162.5 minutes
- Total = 180 min.

Just go 65 the whole way: 197 minutes

Speedup = 197 / 180 = 1.09x
Not great.. Probably not worth getting arrested.

Make the common case fast!



NYU TANDON SCHOOL OF ENGINEERING

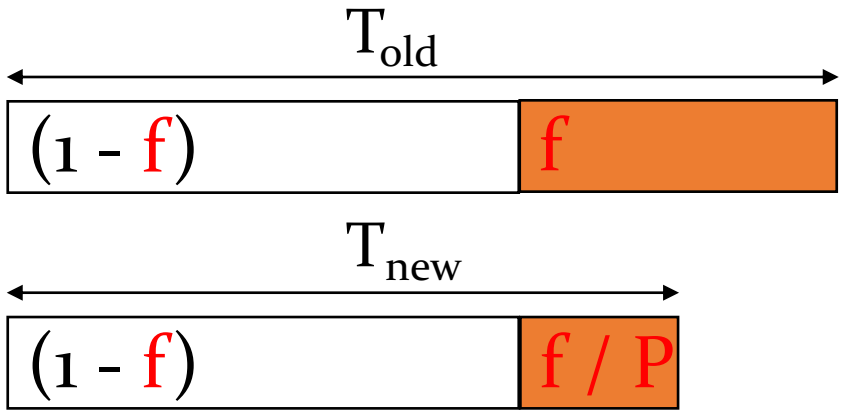
Amdahl's Law (Law of Diminishing Returns)

Performance improvement is limited
by the fraction of time it can be applied

$$\text{Speedup} = \text{Perf}_{new} / \text{Perf}_{old} = T_{old} / T_{new} = \frac{1}{(1-f) + \frac{f}{P}}$$

f = fraction of code sped up

P = speedup factor

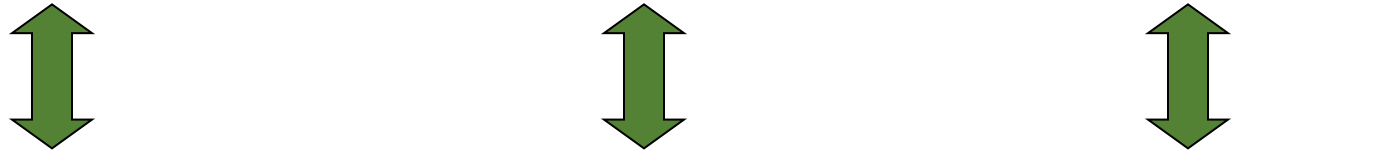


Q: If one function takes 90% of the time,
what's max speedup? (Hint: P -> inf.)

CPU Performance: Iron law of performance

Execution Time = Seconds / Program

$$\frac{\text{Instructions}}{\textbf{program}} \times \frac{\text{cycles}}{\text{Instruction}} \times \frac{\textbf{seconds}}{\text{cycle}}$$



- Programmer
- Algorithms
- ISA
- Compilers

- Microarchitecture
- System architecture

- Microarchitecture, pipeline depth
- Circuit design
- Technology

Example of Performance Evaluation (I)

Operation	Frequency	Clock cycle count
ALU Ops (reg-reg)	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

Assume 25% of the ALU ops directly use a loaded operand that is not used again.
We propose adding ALU instructions that have one src operand in memory.
These new reg-mem instructions take 2 clock cycles.
Also assume that the extended instruction set increases the
branch inst. clock cycle count by 1 but does not impact to cycle time.
Would this change improve performance ?

Example of Performance Evaluation (I)

Operation	Frequency	Clock cycle count
ALU Ops (reg-reg)	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

Assume 25% of the ALU ops directly use a loaded operand that is not used again.
We propose adding ALU instructions that have one src operand in memory.
These new reg-mem instructions take 2 clock cycles.
Also assume that the extended instruction set increases the branch inst. clock cycle count by 1 but does not impact to cycle time.
Would this change improve performance ?

New inst.

$$Cycles_{old} = 0.43*1 + 0.21*2 + 0.12*2 + 0.24*2 = 1.57$$

$$Cycles_{new} = 0.25*0.43*2 + (0.43 - 0.25*0.43)*1 + (0.21 - 0.25*0.43)*2 + 0.12*2 + 0.24*3 = 1.703$$

reg-reg alu

Save load time!

Store and branch stays same.

Example of Performance Evaluation (II)

FP instructions = 25%
Average CPI of FP instructions = 4.0
Average CPI of other instructions = 1.33
FPSQRT = 2% of all instructions, CPI of FPSQRT = 20

Design Option 1: decrease the CPI of FQSQRT to 2

Design Option 2: decease the average CPI of all FP instructions to 2.5

Calculate: original CPI, Des opt 1 CPI, Des. Opt 2 CPI.

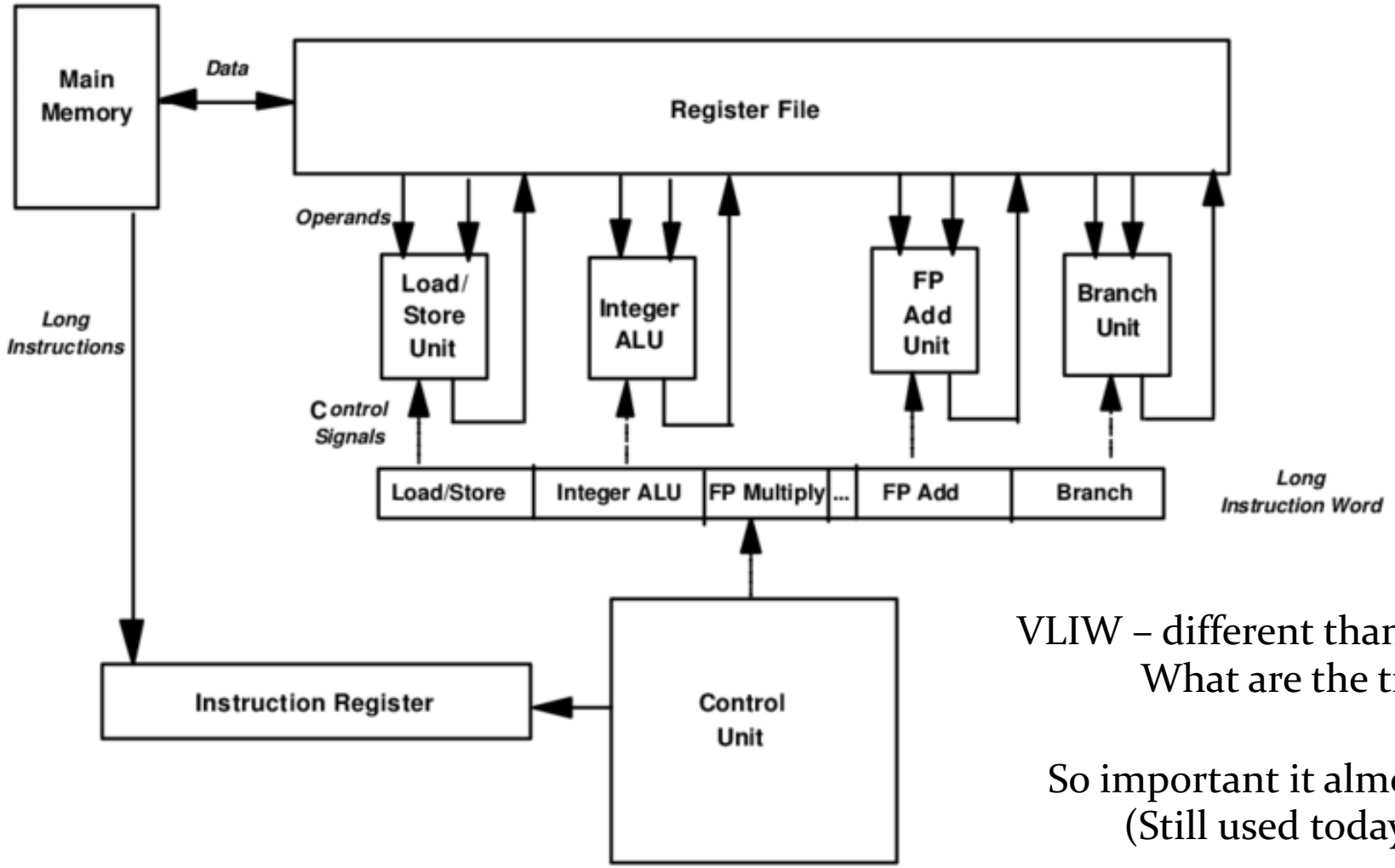
Original CPI = 0.25*4 + 1.33*(1-0.25) = 2.0

Option 1 CPI = 2.0 – 2%*(20-2) = 1.64

Option 2 CPI = 0.25*2.5 + 1.33*(1-0.25) = 1.625

Speedup of Option 1 = 2/1.64 = 1.2195
Speedup of Option 2 = 2/1.625 = 1.2308

Does NYU even do architecture?



VLIW – different than RISC and CISC
What are the tradeoffs?

So important it almost killed Intel
(Still used today in DSPs)

Who invented VLIW?

Josh Fisher!

Highest award in field. Given at ISCA each year.

~Second highest award in field. Given at MICRO each year.

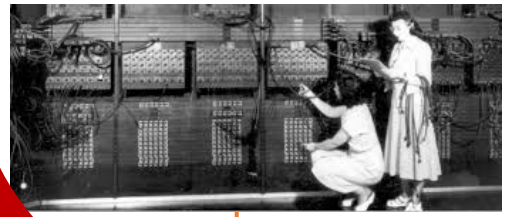
Josh Fisher

Josh Fisher in 2014.

Born	July 22, 1946 (age 74) [1] Bronx, NY, USA [1]
Alma mater	Courant Institute of Mathematical Sciences (New York University)
Known for	The Invention of VLIW Architectures, Instruction-level Parallelism, Trace Scheduling, Co-Founder of Multiflow Computer
Awards	Eckert-Mauchly Award, (IEEE/ACM 2003) B. Ramakrishna Rau Award (IEEE-CS 2012) Connecticut Entrepreneur of the Year (1987) Presidential Young Investigator's Award (NSF 1984)
Scientific career	
Fields	Computer Architecture, Compiling, Embedded Systems
Institutions	Yale University, Multiflow Computer, Hewlett-Packard Laboratories (retired)

Computers over time

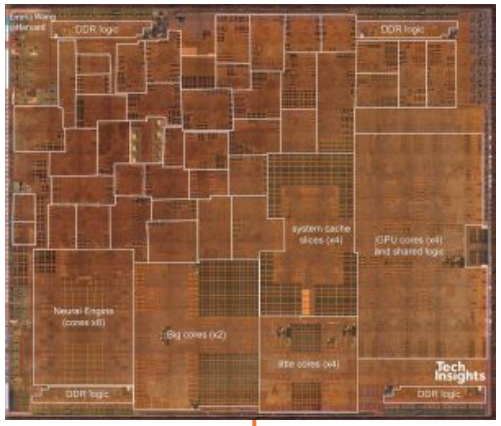
ENIAC
1945

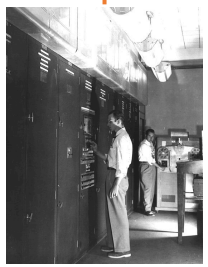


IBM 360
1964

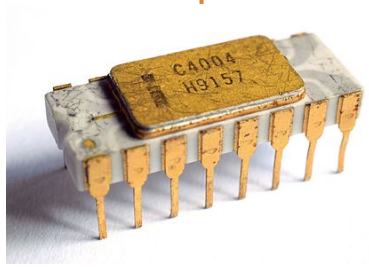
Pentium Pro
1995
P6 cores!

Apple A12 SoC

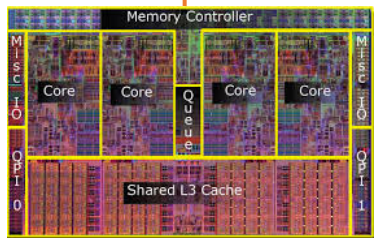




EDVAC
1949



Intel 4004
1971



Intel Nahelem
2008

NYU TANDON SCHOOL OF ENGINEERING

MICRO 2020 happening soon

Type	Early *until 10/10*	Late *after 10/10*
ACM/IEEE Member	EUR 40	EUR 55
ACM/IEEE Lifetime/Retired Member	EUR 25	EUR 35
Non-Member	EUR 60	EUR 80
ACM/IEEE Student Member	EUR 25	EUR 35
Student Non-Member	EUR 40	EUR 55