Chapter 2: Improving Processor's Performance with Pipelining

[ADAPTED FROM MARY JANE IRWIN'S SLIDES (PSU) BASED ON *COMPUTER ORGANIZATION AND DESIGN*, ARM ED. PATTERSON & HENNESSY, © 2017, ELSEVIER]

UNIVERSIDAD DE MÁLAGA

ANDALUCÍA TECH
Campus de Excelencia Internacional
International Campus of Excellence

Index

Introduction

CPU performance factors

- Instruction count
  - Determined by ISA and compiler
- CPI and Cycle time
  - Determined by CPU hardware

We will examine two MIPS implementations

- A simplified version
- A more realistic pipelined version

Simple subset, shows most aspects

- Memory reference: lw, sw
- Arithmetic/logical: add, sub, and, or, slt
- Control transfer: beq, j

MIPS (RISC) Design Principles

Simplicity favors regularity

fixed size instructions

small number of instruction formats

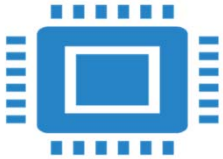
opcode always the first 6 bits

Smaller is faster

limited instruction set

limited number of registers in register file

limited number of addressing modes



Make the common case fast

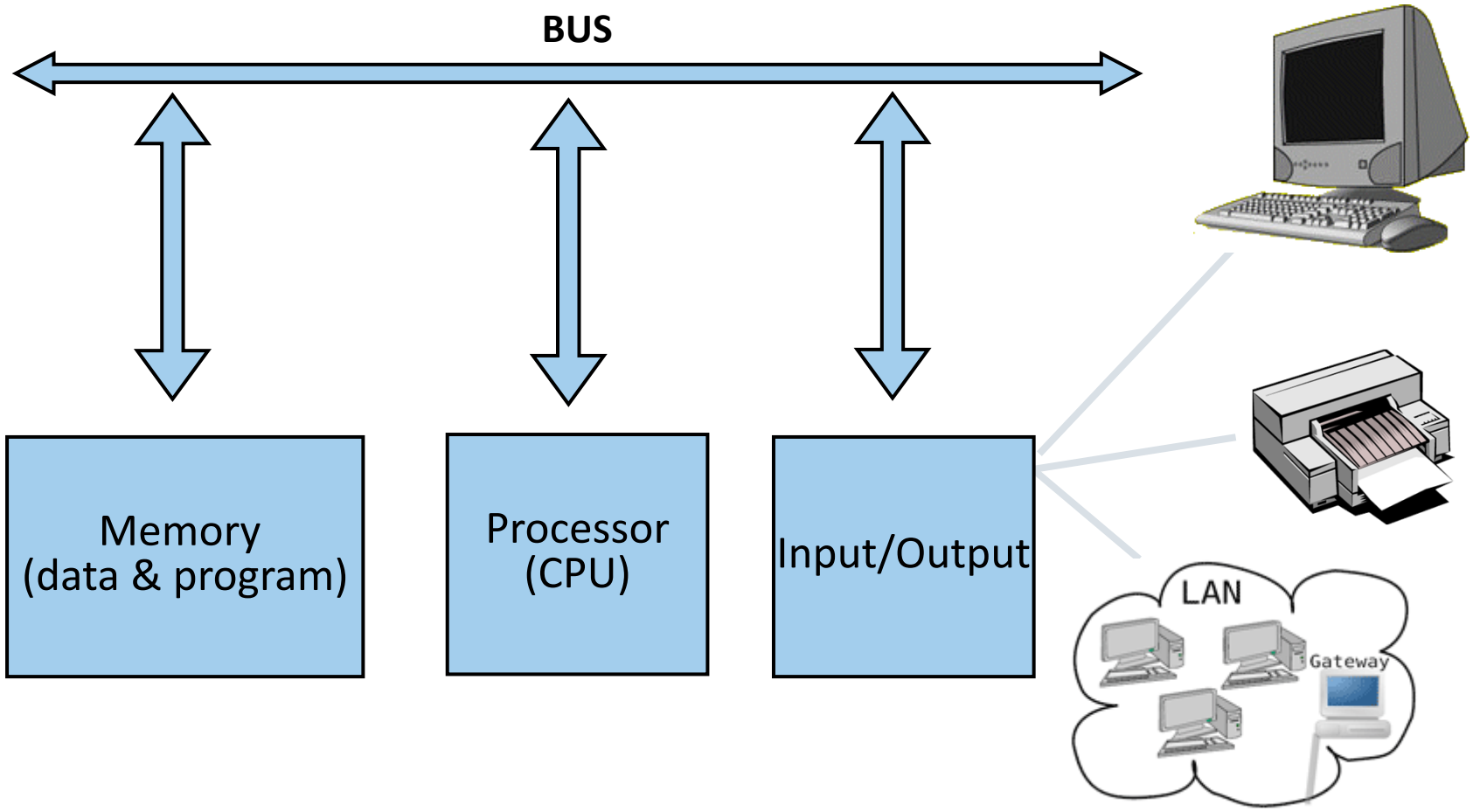
arithmetic operands from the register file (load-store machine)

allow instructions to contain immediate operands

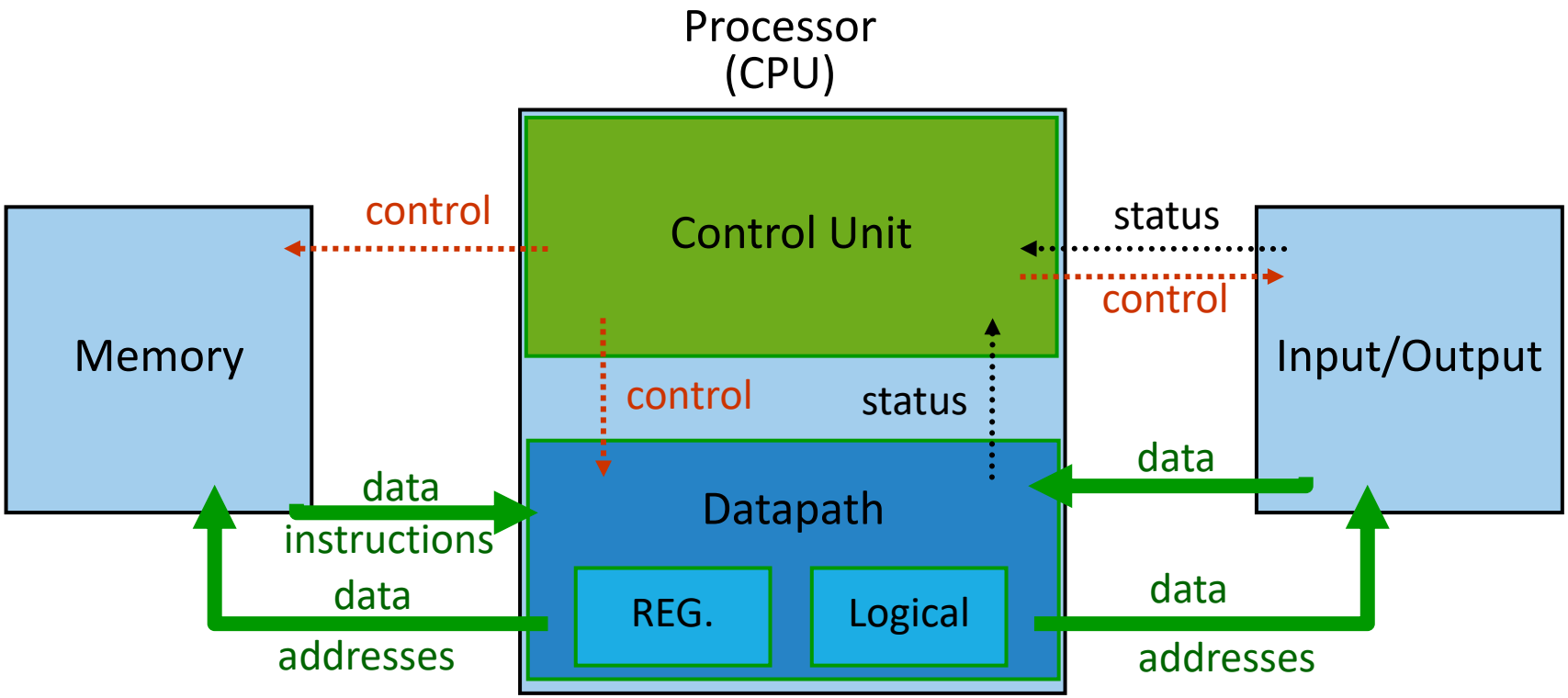
Good design demands good compromises

three instruction formats

Computer's components



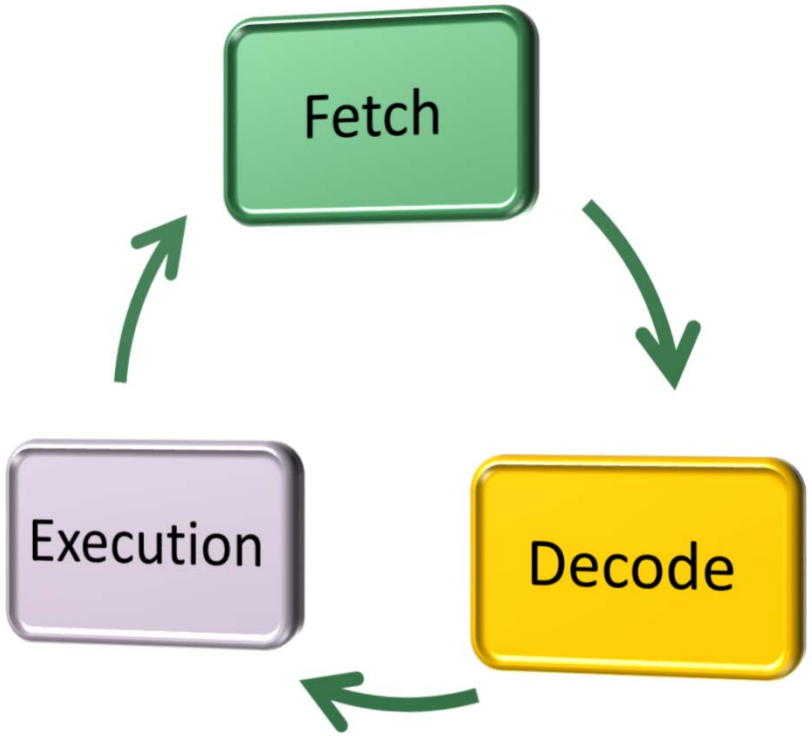
Computer's components



Instruction cycle

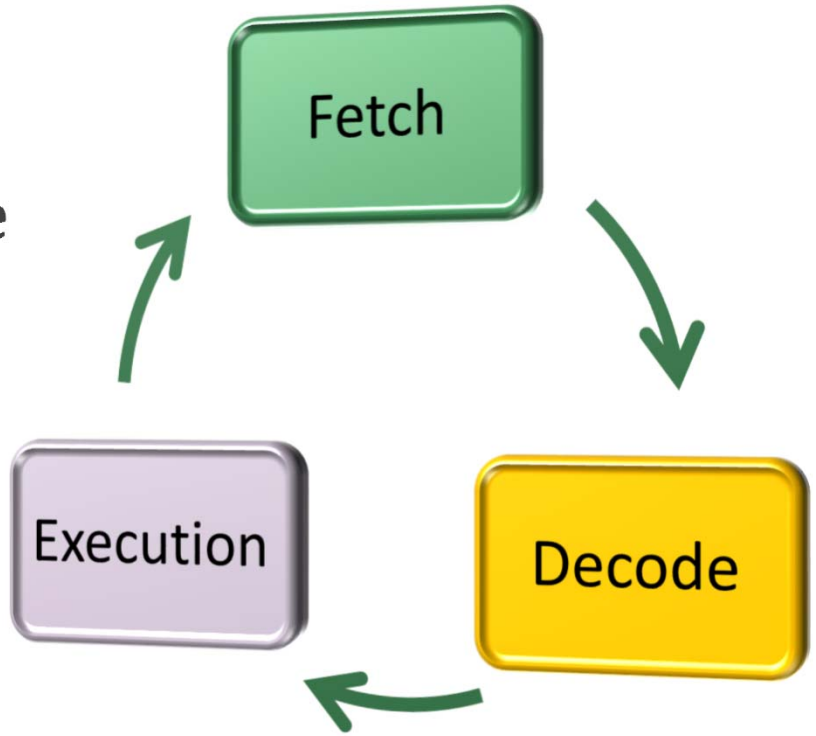
Instruction cycle:

1. Instruction Fetch (IF). The CPU uses the Program Counter (PC) to read the instruction from memory.
2. Instruction Decode (ID). The CPU discover what the instruction does and what operands needs.
3. Execution (EX). The required action is executed.



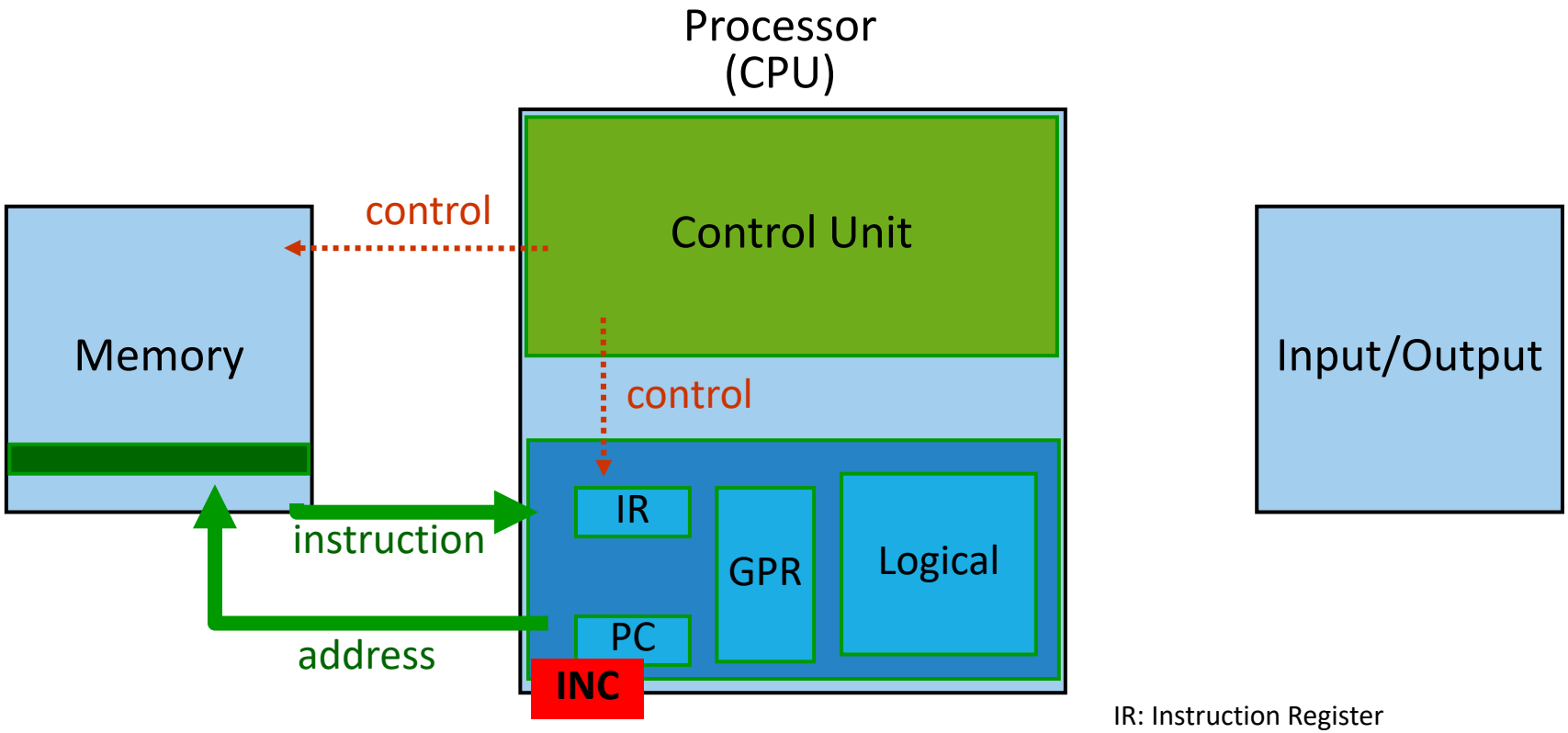
Instruction cycle

1. Instruction Fetch (IF). The CPU uses the Program Counter (PC) to read the instruction from memory.
2. Instruction Decode (ID). The CPU discover what the instruction does and what operands needs.
3. Execution (EX). The required action is executed.



Instruction Cycle

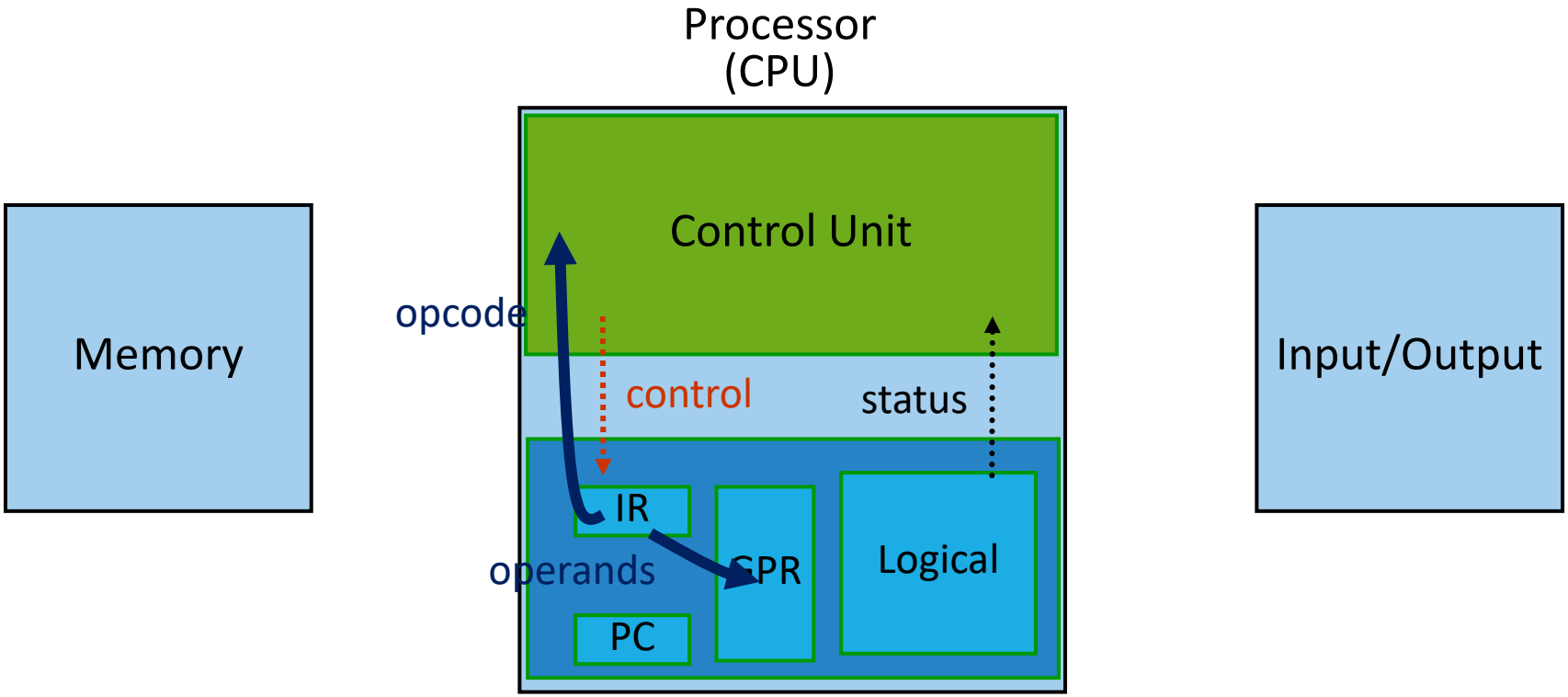
Instruction Fetch (IF):



IR: Instruction Register
PC: Program Counter
GPR: Generic Purpose Registers

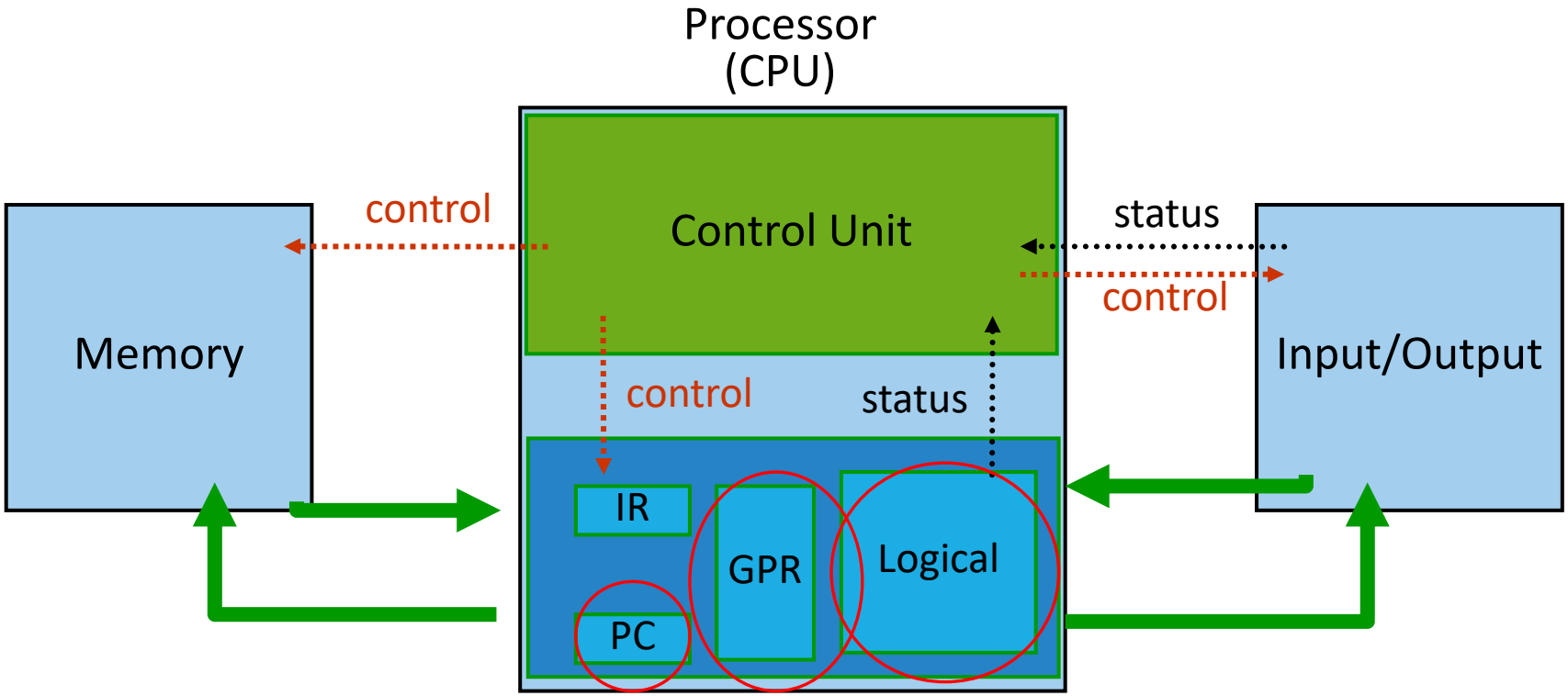
Instruction Cycle

Instruction Decode (ID):



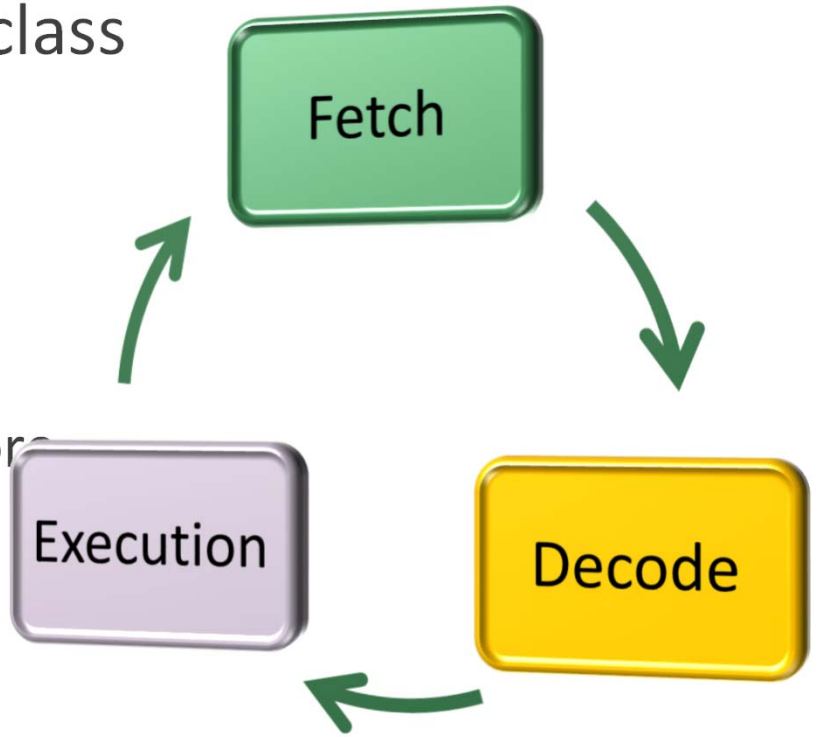
Instruction Cycle

Execution (EX):

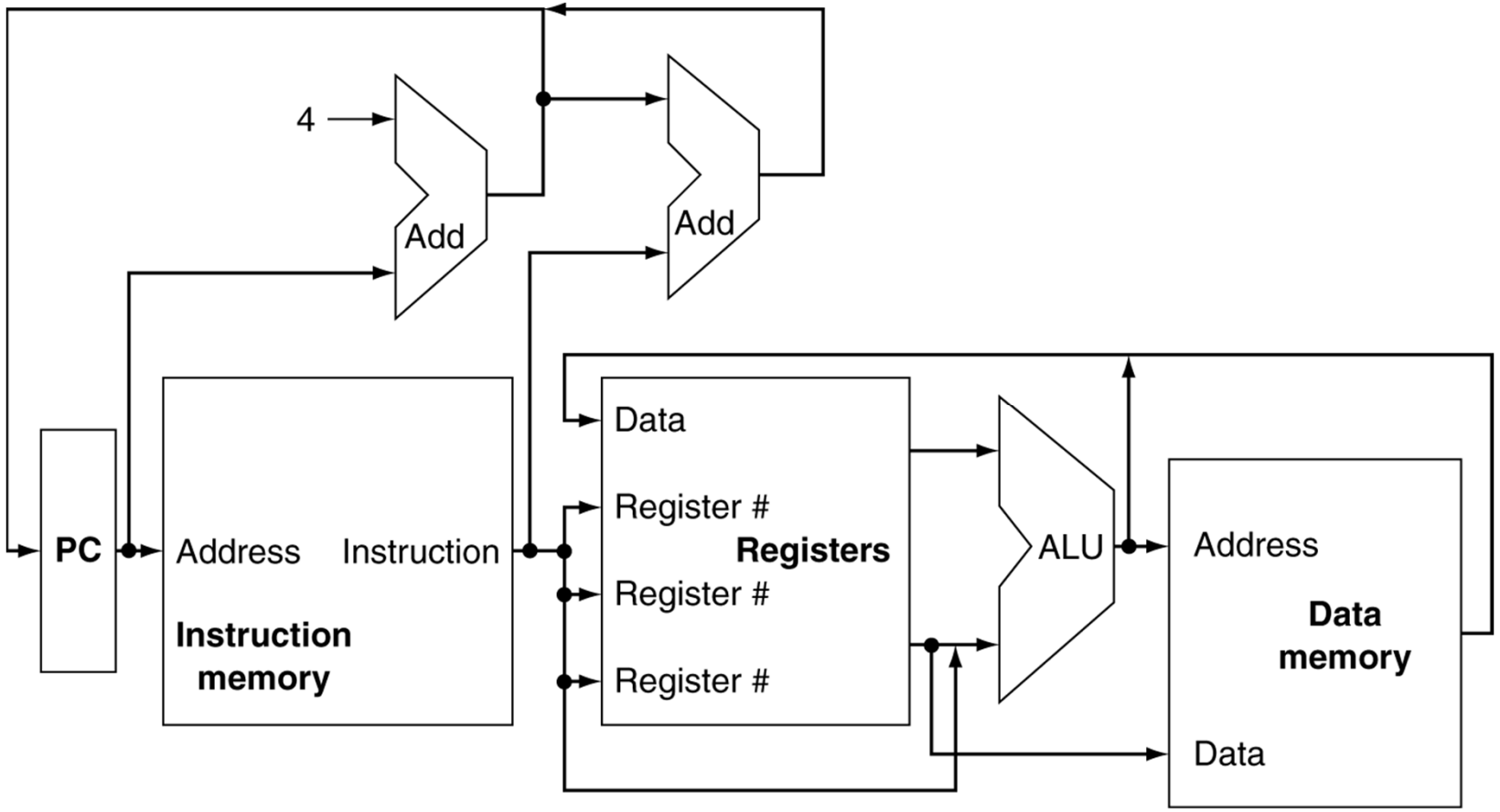


Instruction Cycle in MIPS

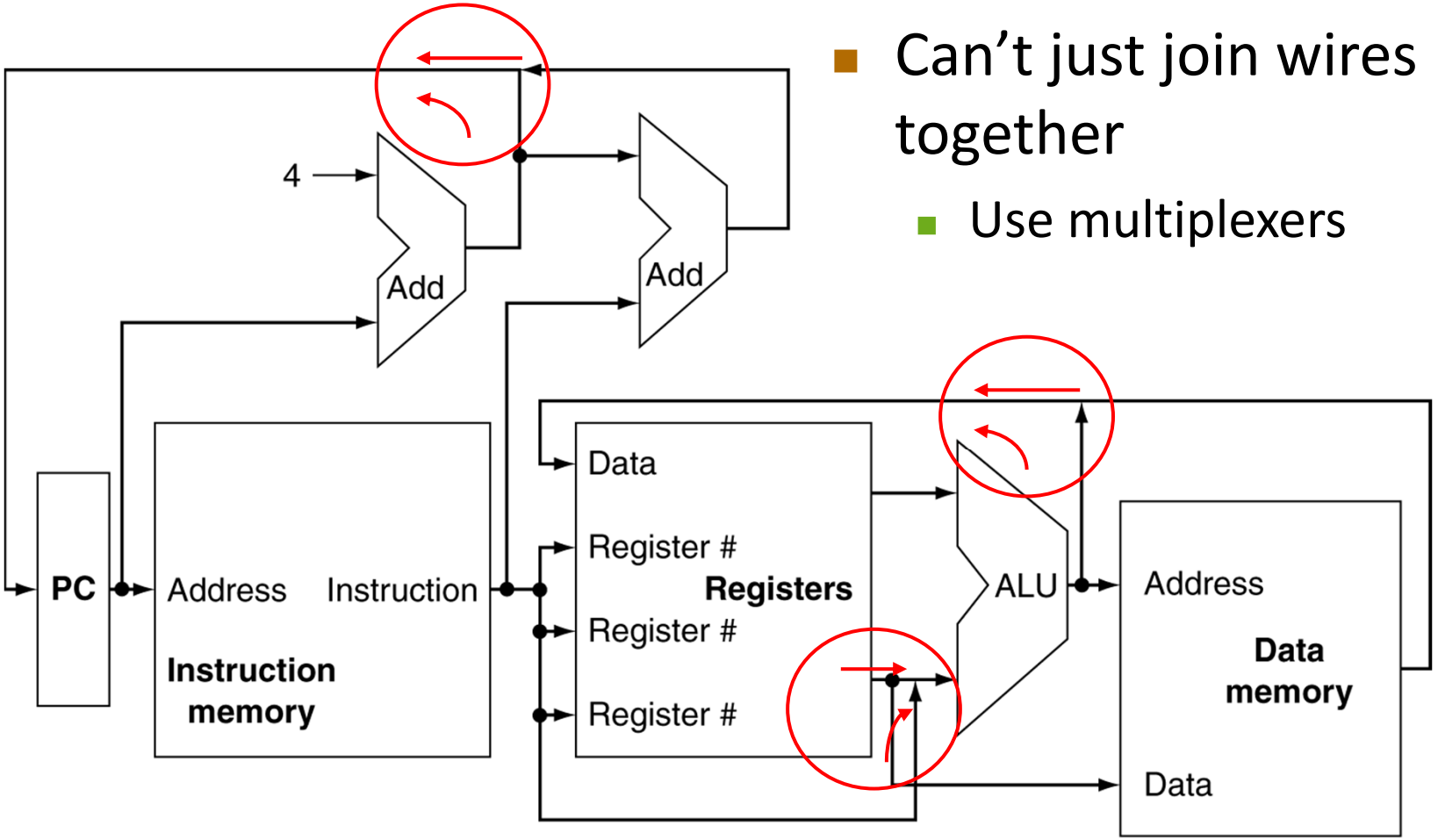
1. PC → instruction memory, fetch instruction
2. Register numbers → register file, read registers
3. Depending on instruction class
   - Use ALU to calculate
     - Arithmetic result
     - Memory address for load/store
     - Branch target address
   - Access data memory for load/stor
   - PC ← target address or PC + 4



MIPS Overview (Datapath)

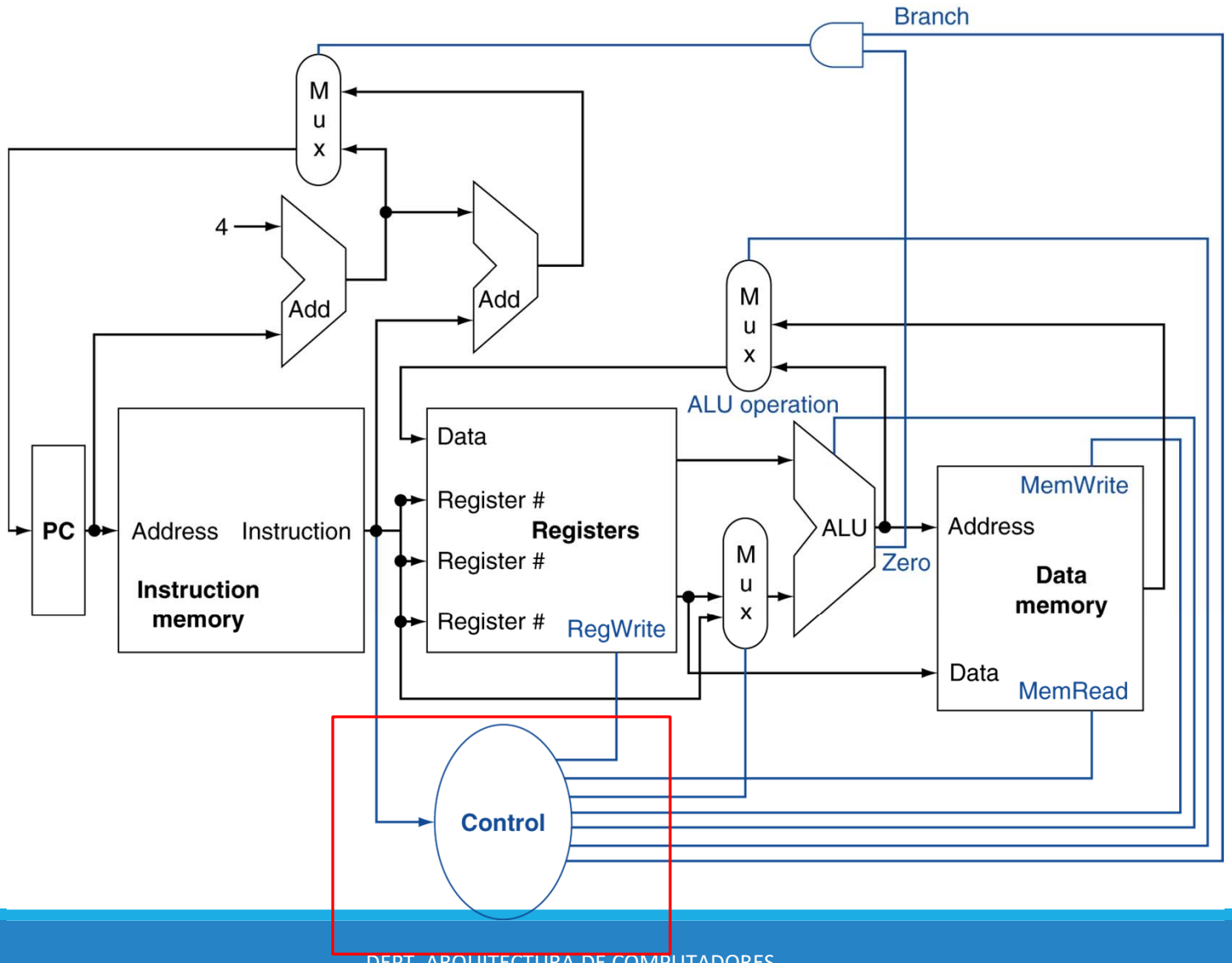


Multiplexers



- Can’t just join wires together
  - Use multiplexers

Control Unit



Index

3.2.- Logic Design Basics

Information encoded in binary

- Low voltage = 0, High voltage = 1
- One wire per bit
- Multi-bit data encoded on multi-wire buses

Combinational element

- Operate on data
- Output is a function of input

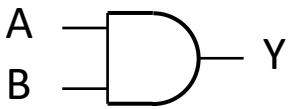
State (sequential) elements

- Store information

Combinational Elements

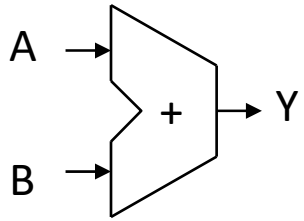
AND-gate

- Y = A & B



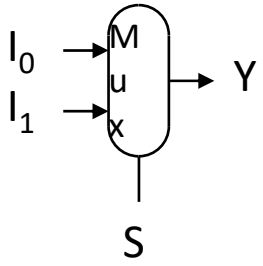
Adder

- Y = A + B



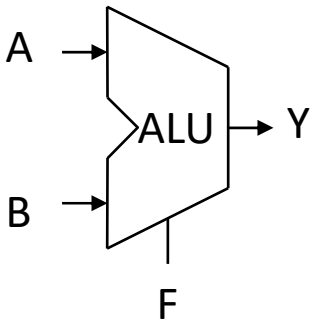
Multiplexer

- Y = S ? I1 : I0



Arithmetic-logic Unit

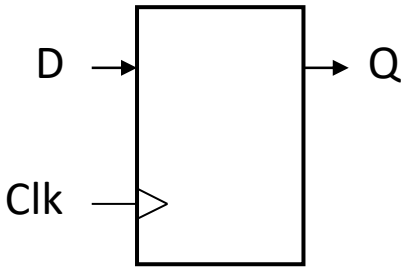
- Y = F(A,B)

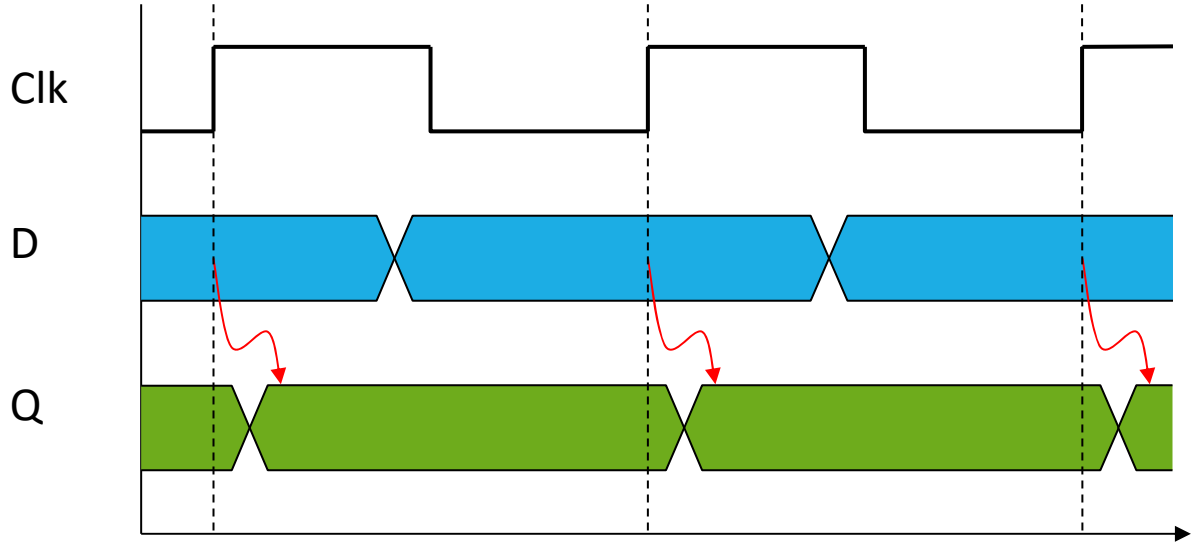


Sequential Elements

Register: stores data in a circuit

- Uses a clock signal to determine when to update the stored value
- Edge-triggered: update when Clk changes from 0 to 1

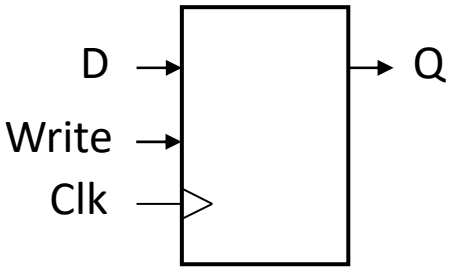


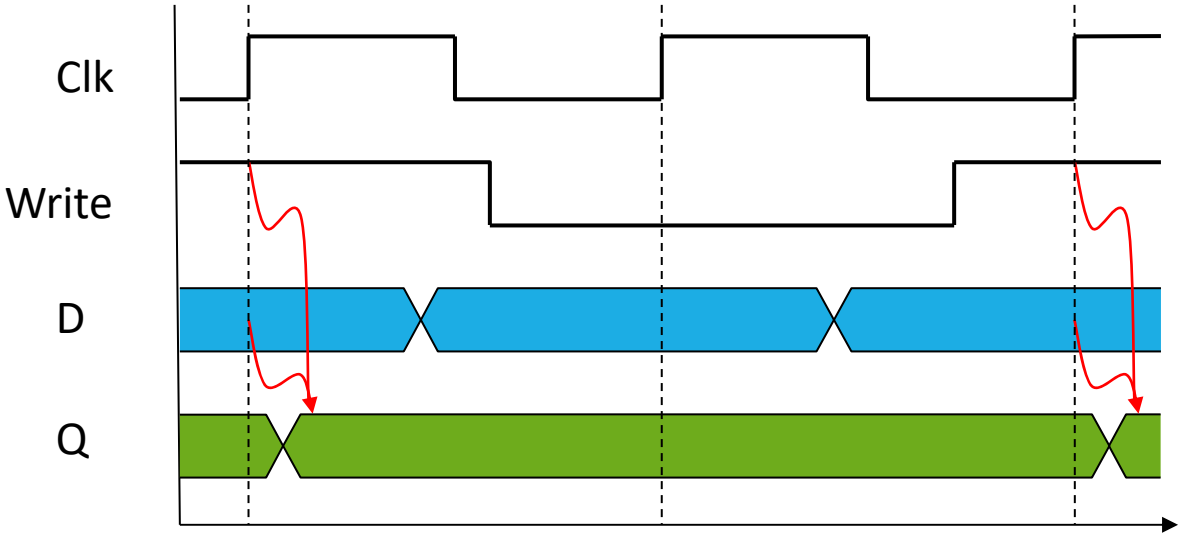


Sequential Elements

Register with write control

- Only updates on clock edge when write control input is 1
- Used when stored value is required later

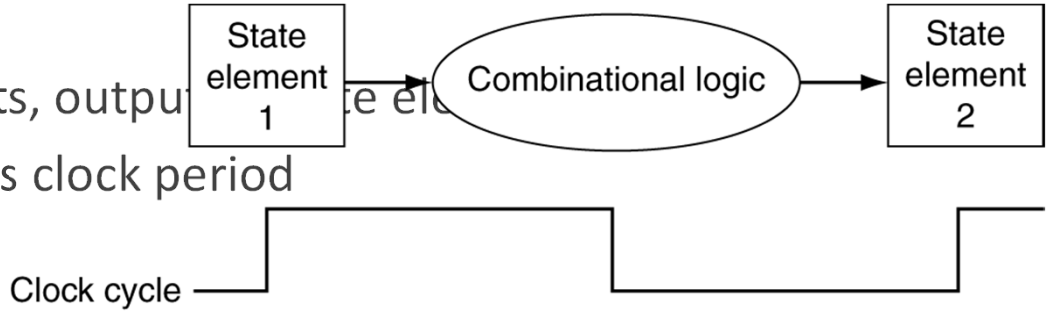


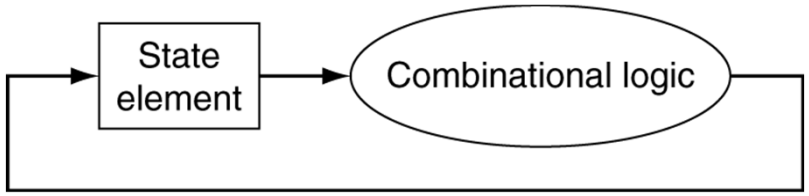


Clocking Methodology

Combinational logic transforms data during clock cycles

- Between clock edges
- Input from state elements, outpu[illegible]
- Longest delay determines clock period





- With write signal there is no worry about feedback

Index

MIPS instructions

Features:

- Encoded as 32-bit instruction words
- Small number of formats encoding operation code (opcode), register numbers, …
- Regularity!

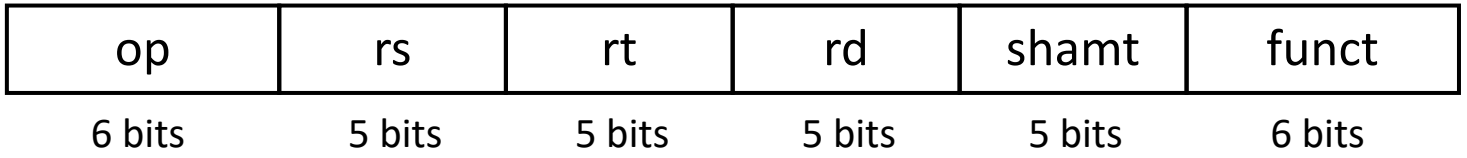
Only 3 different formats:

- R-type
- I-type
- J-type

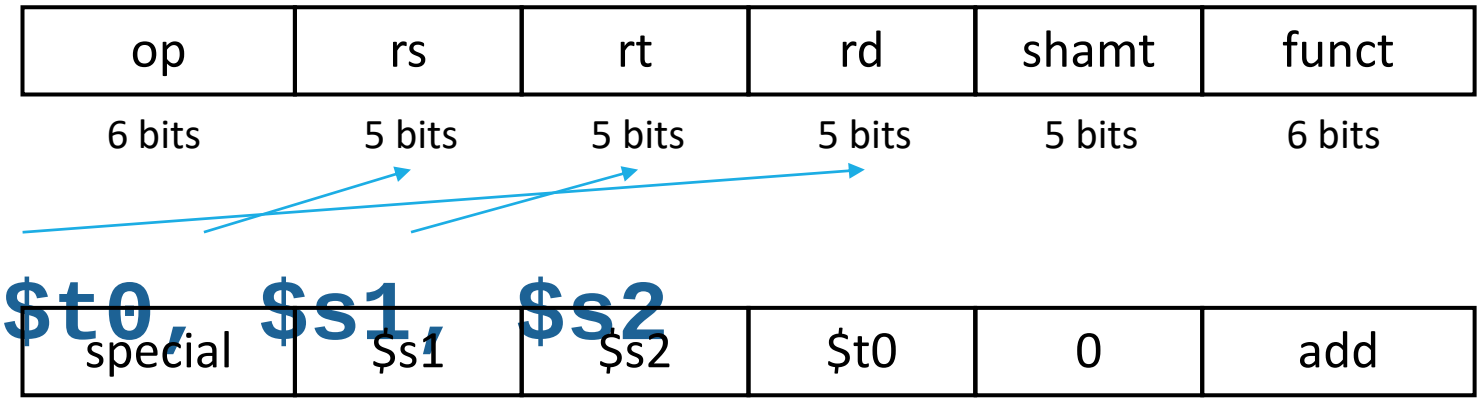
MIPS R-format Instructions

Arithmetic-logic instructions

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

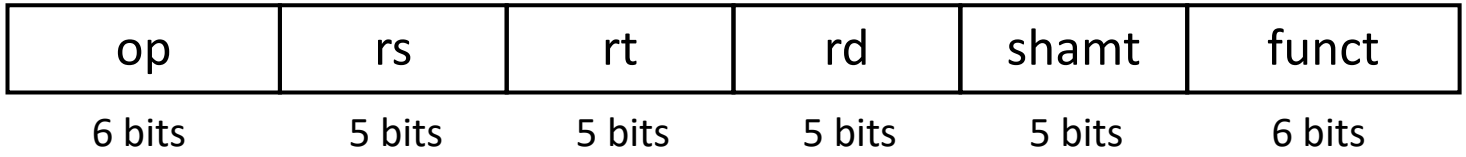


R-format Example



Name	Register Number
$zero	0
$at	1
$v0 - $v1	2-3
$a0 - $a3	4-7
$t0 - $t7	8-15
$s0 - $s7	16-23
$t8 - $t9	24-25
$gp	28
$sp	29
$fp	30
$ra	31

R-format Example



`add $t0, $s1, $s2`

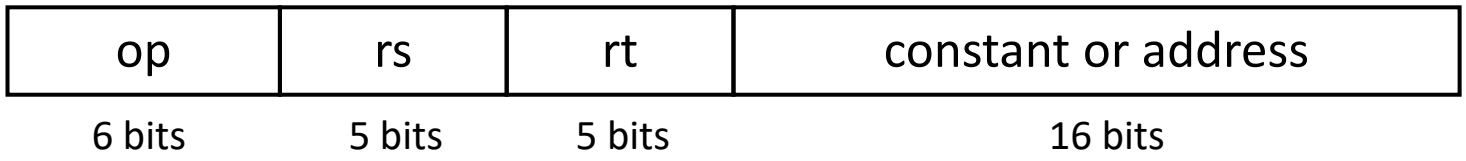
special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions

Immediate arithmetic, load/store instructions and branches

- Constant: -2^{15} to $+2^{15} - 1$
- rt: destination or source register number
- Address: offset added to base address in rs
- Branches: rs and rt are compared
- Jump address: offset added to PC



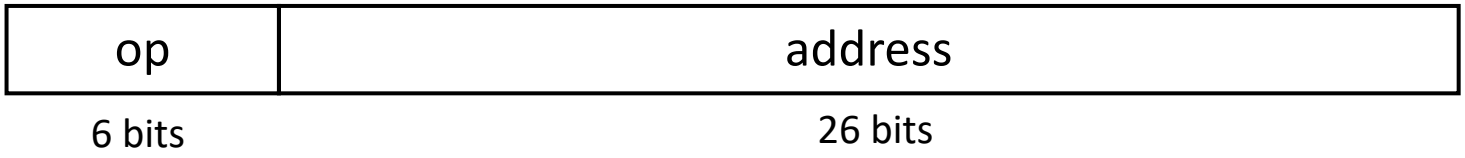
MIPS J-format Instructions

Jump (`j` and `jal`) targets could be anywhere in text segment

- Encode full address in instruction

(Pseudo)Direct jump addressing

- Target address = $PC_{31\ldots28}$: (address × 4)



Classes of Instructions

Arithmetic-logic instructions

`add, sub, or, and, addi, subi, …`

Data transfer instructions

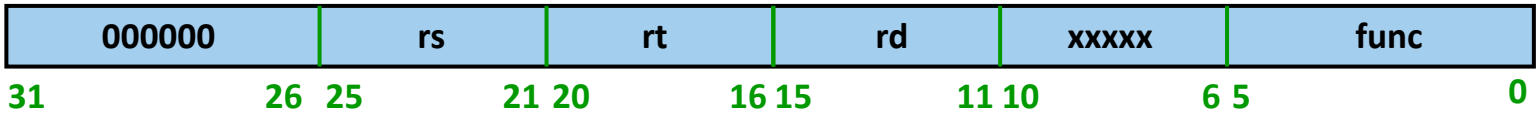
`lw, sw, …`

Branch and jump instructions

`beq, bne, j, jal, jr, …`

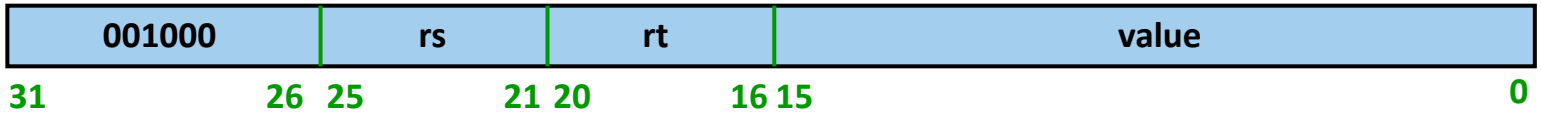
Arithmetic-Logic Instructions

OpAlu rd,rs,rt R[rd] ← R[rs] (OpAlu) R[rt]

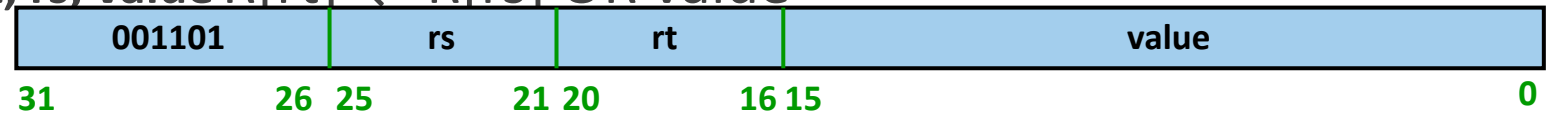


func		func		func	
100000	ADD	100101	OR	000000	SLL
100010	SUB	101010	SLT	000011	SRA
100100	AND	000010	SRL		

ADDI rt, rs, value R[rt] ← R[rs] + value

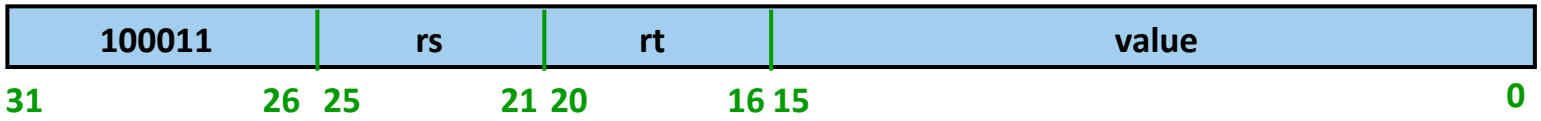


ORI rt, rs, value R[rt] ← R[rs] OR value

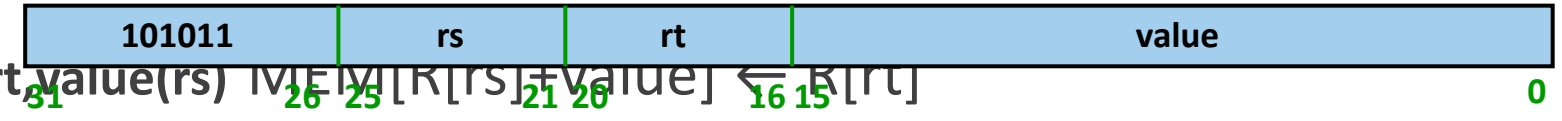


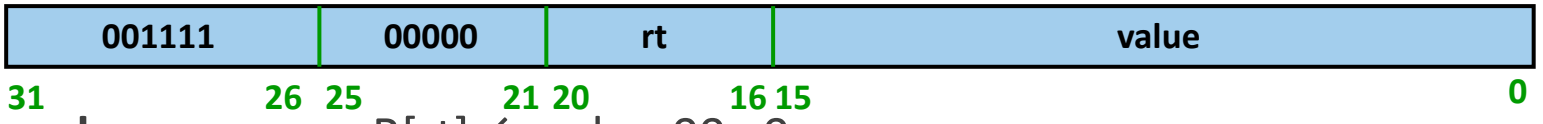
Data Transfer Instructions

LW rt,value(rs) R[rt] ← MEM[R[rs]+value]



SW rt,value(rs) MEM[R[rs]+value] ← R[rt]

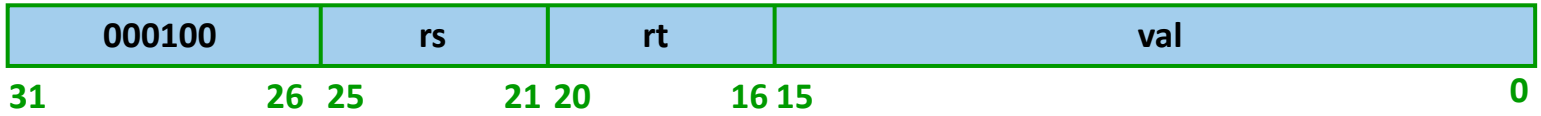


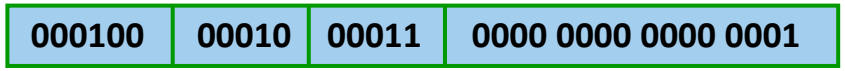


LUI rt, value R[rt] ← value,00…0

Branch Instructions

BEQ rs,rt,val if (R[rs]==R[rt]) PC ← PC+4+[val,00]






```
PC      beq $2, $3, L2
PC+4    add $4, $5, $6
PC+8 L2: sub $7, $8, $9
```

Jump to PC+8:

$$\mathrm{PC} + 8 = \mathrm{PC} + 4 + [\mathrm{val}, 00] = \mathrm{PC} + 4 + \mathrm{val} \times 2^2$$

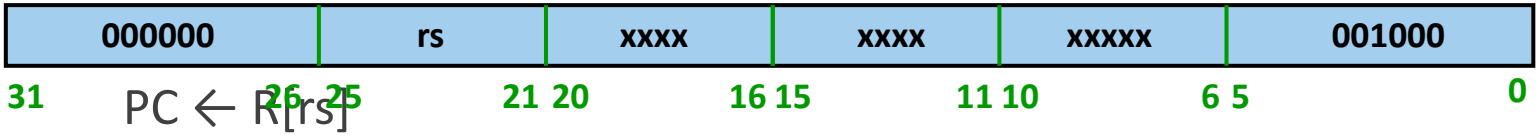
$$\mathrm{val} = \frac{8-4}{4} = 1$$

Jump Instructions

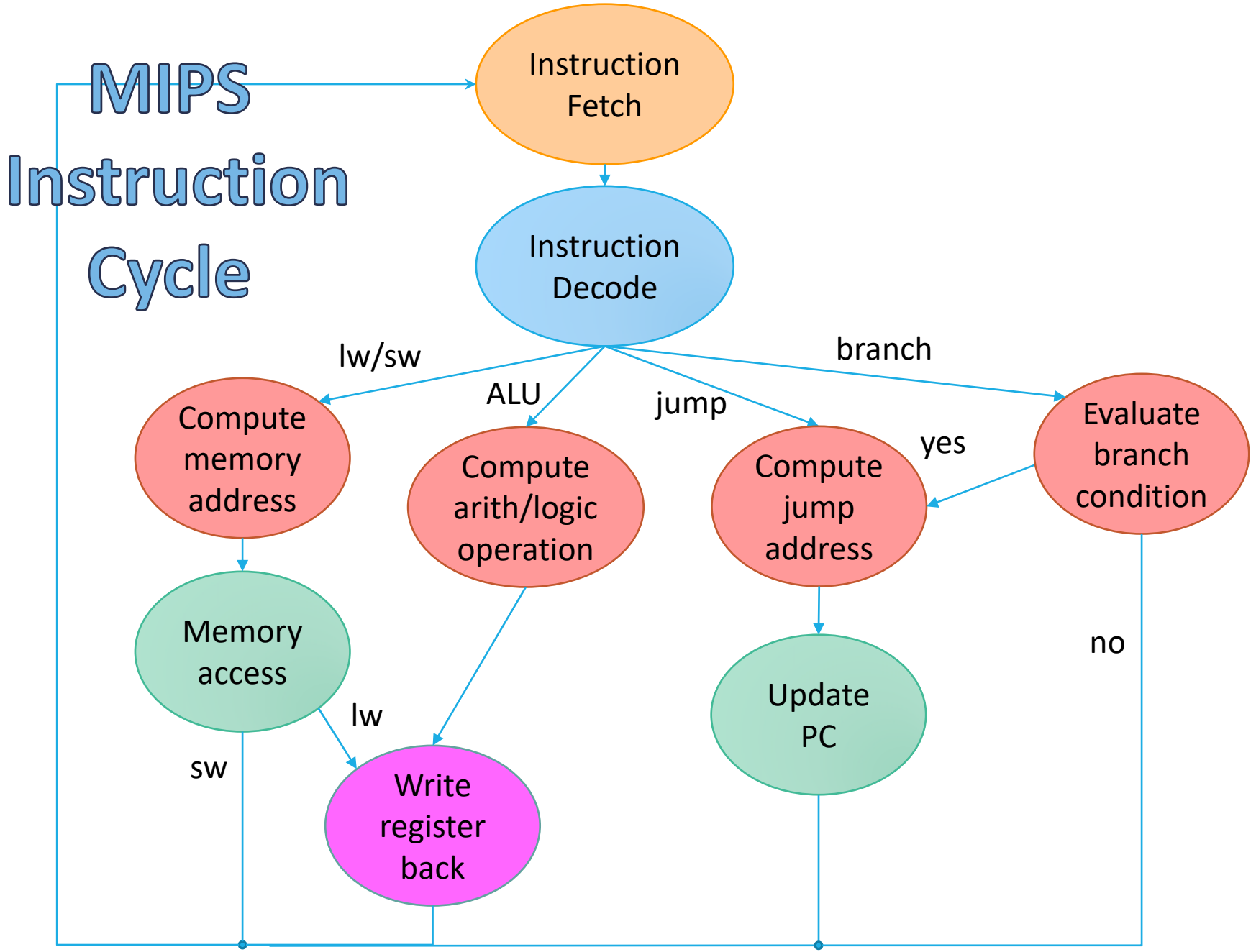
J val PC ← (PC+4)[31:28],val,00

JAL val R[31] ← PC+4
PC ← (PC+4)[31:28],val,00

JR rs PC ← R[rs]



MIPS Instruction Cycle



Index

Datapath Features

Elements that process data and addresses in the CPU

- Registers, ALUs, mux's, memories, …

We will build a MIPS datapath incrementally

- Refining the overview design

Datapath Features

Single-cycle implementation: Each instruction must be completely executed in 1 clock cycle.

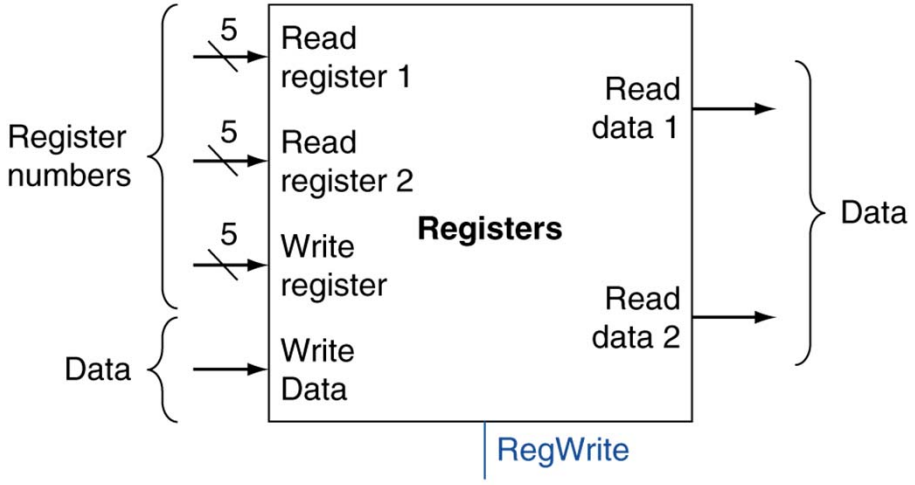
Some components:

- Register File (a set of 32 32-bit GPR)
- Program Counter (PC, 32-bit)
- Adders
- 2 banks of memory: Data & Instructions
  - 2^{32} words of capacity
  - Byte addressed
- 32-bit ALU

Datapath Elements

Register File:

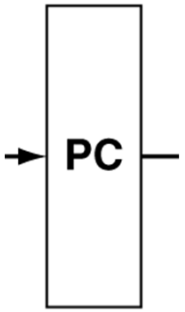
- 32 registers (32-bit)
- 2 read ports + 1 write port
  - 2 reads +1 write simultaneusly



Datapath Elements

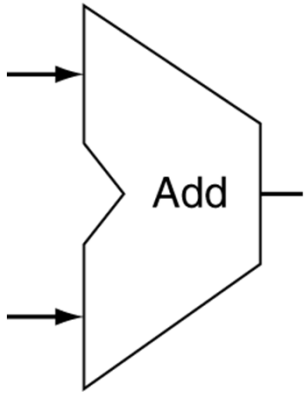
Program Counter

- 32-bits reg.



Adder

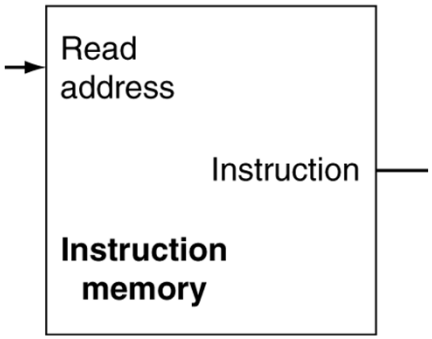
- 32-bits operands



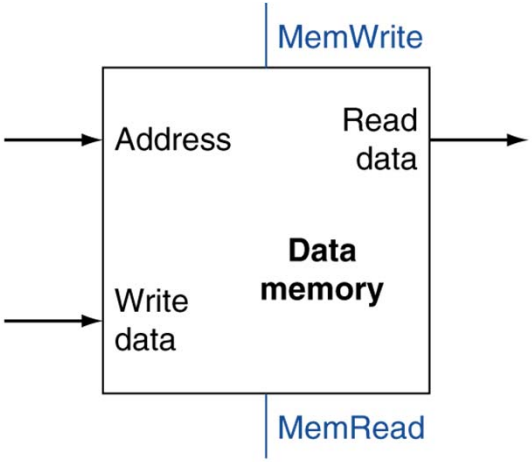
Datapath Elements

Memory: 2 separated banks, 2^{32} words each

- Instruction Memory:



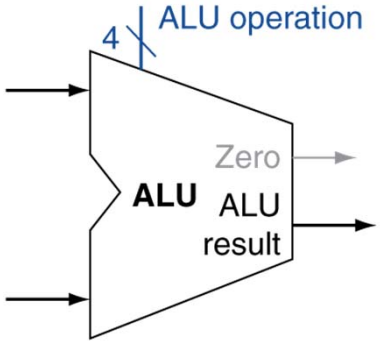
- Data Memory:



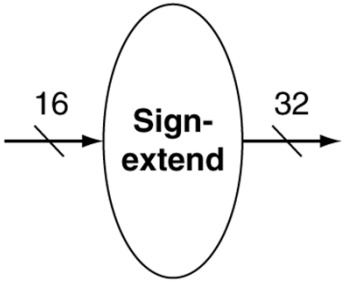
Datapath Elements

ALU, Arithmetic-Logic Unit

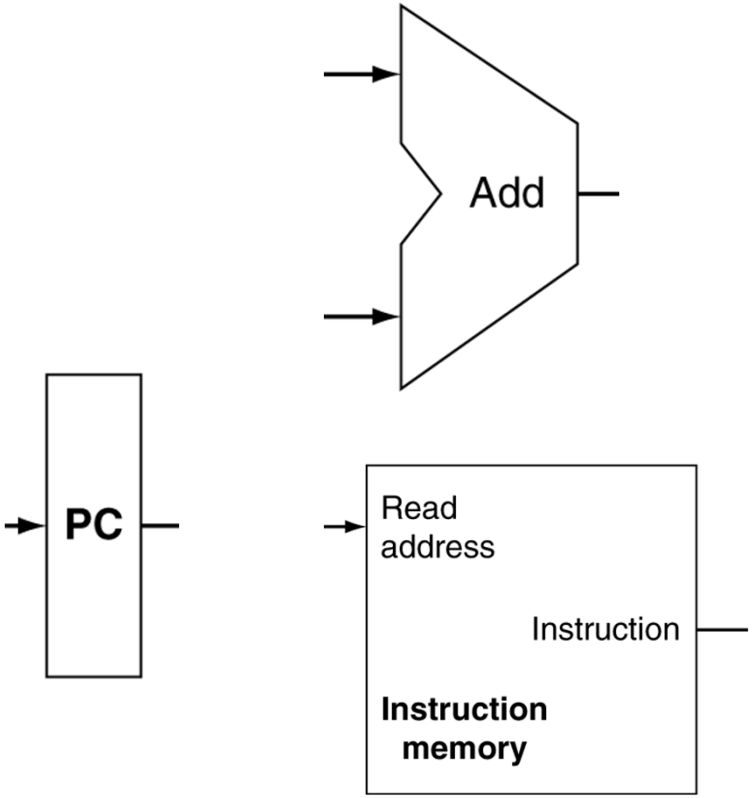
- Two 32-bit inputs, One 32-bit ouput.
- One status output flag (Zero)
- 2^4 different operations



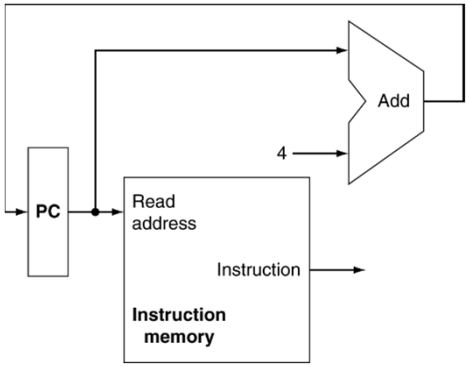
Sign extend



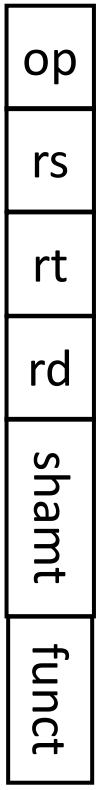
HW Design: Instruction Fetch

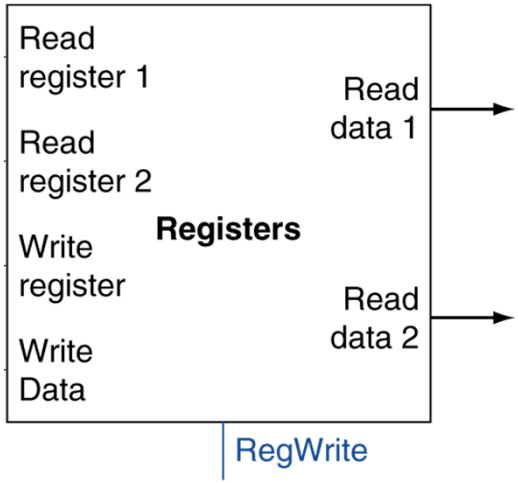


HW Design: Instruction Decode



R-format

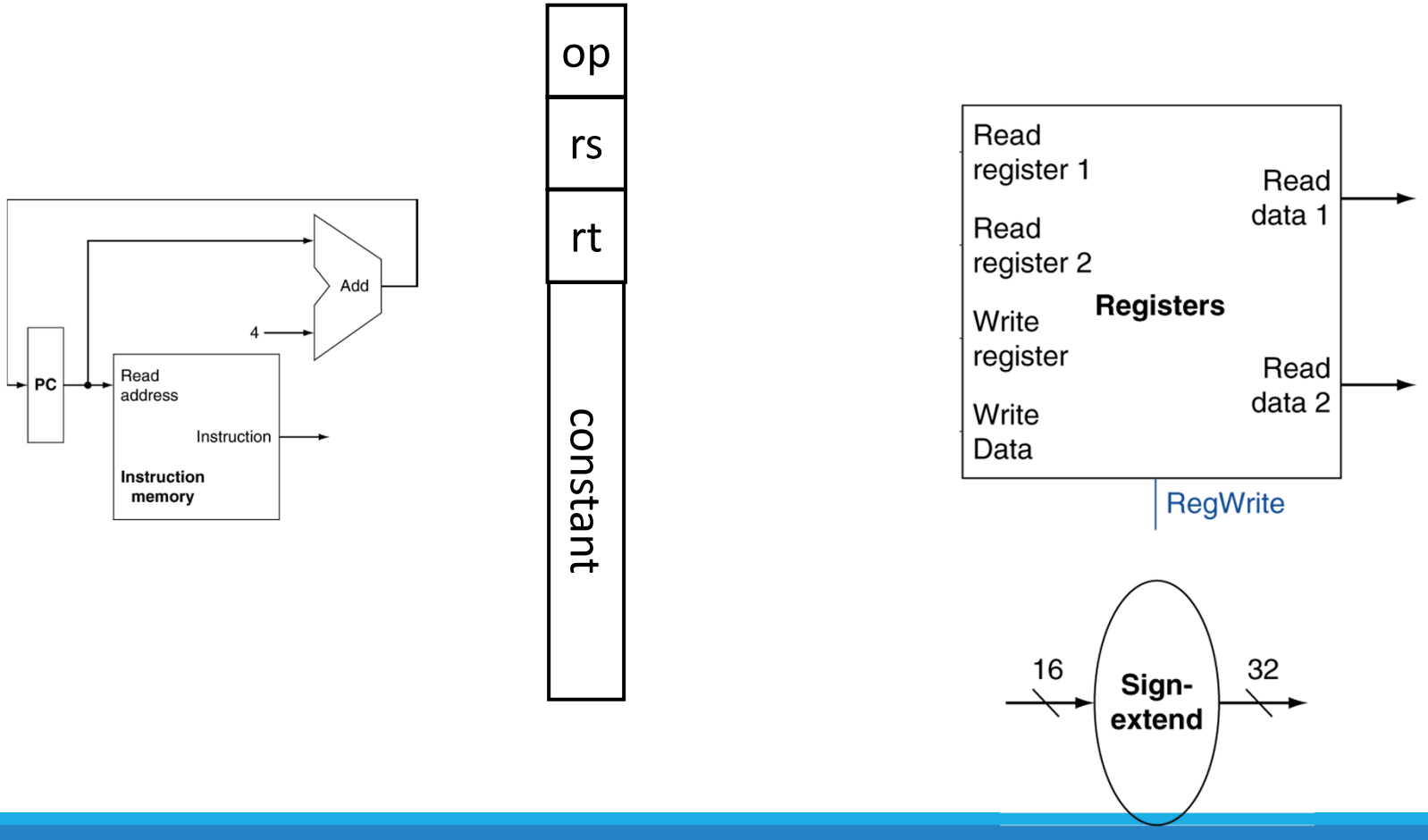




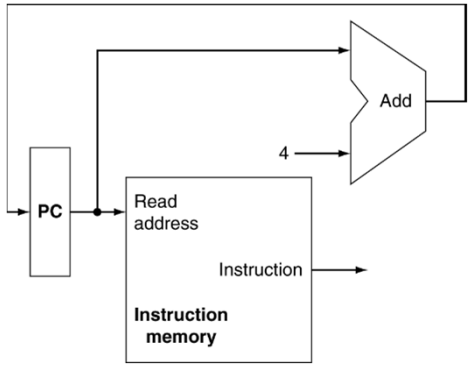
HW Design: Instruction Decode

```
lw rt,value(rs)   MEM[R[rs]+value] → R[rt]
sw rt,value(rs)   R[rt] → MEM[R[rs]+value]
addi rt,rs,val    R[rs]+val → R[rt]
beq rs,rt,label   if (R[rs]==R[rt]) PC=PC+4*val
```

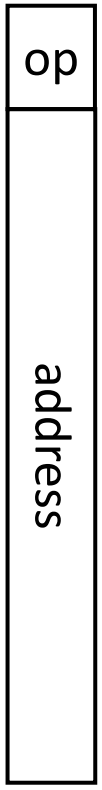
I-format

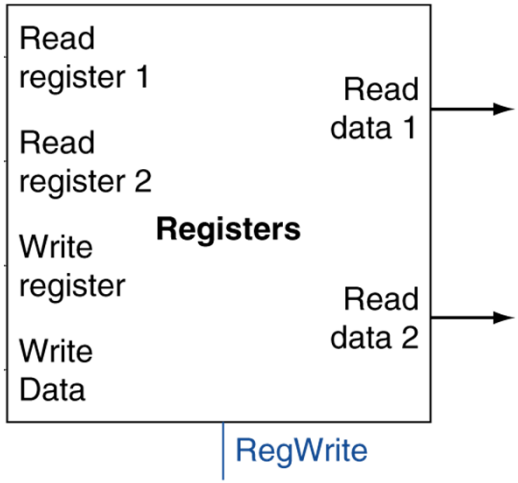


HW Design: Instruction Decode

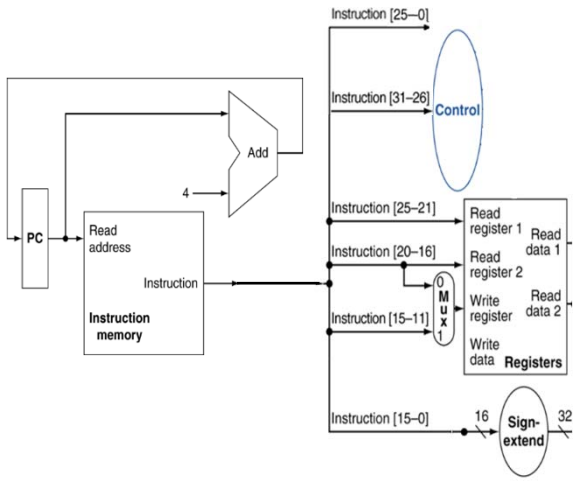


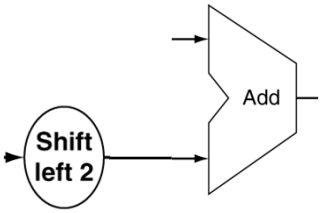
J-format

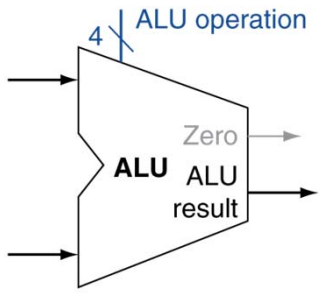


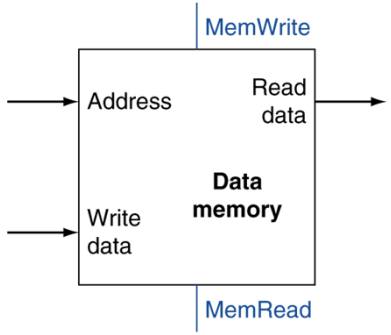


HW Design: Instruction Execution

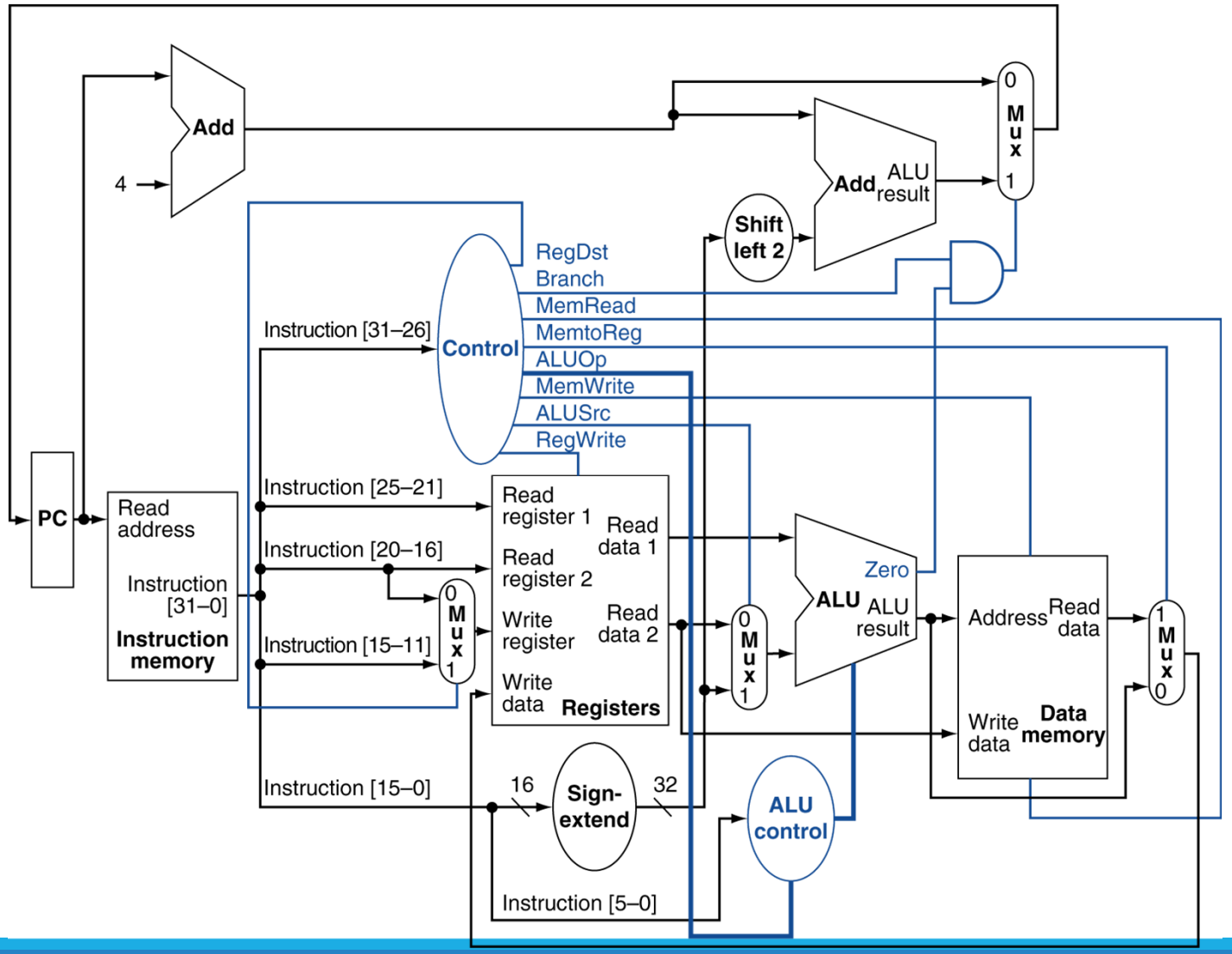




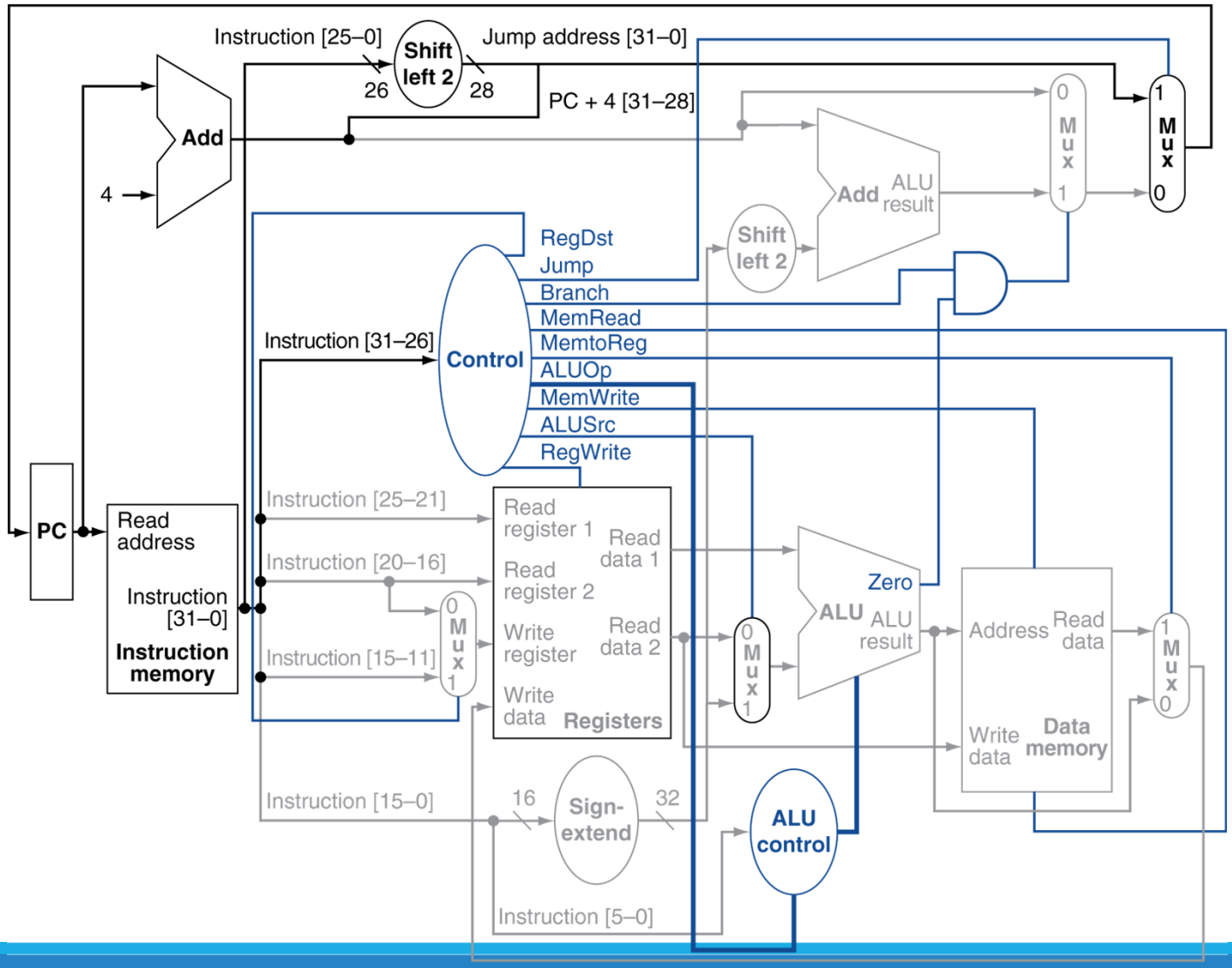




Datapath With Control



Datapath With Jumps Added



Index

The single cycle datapath

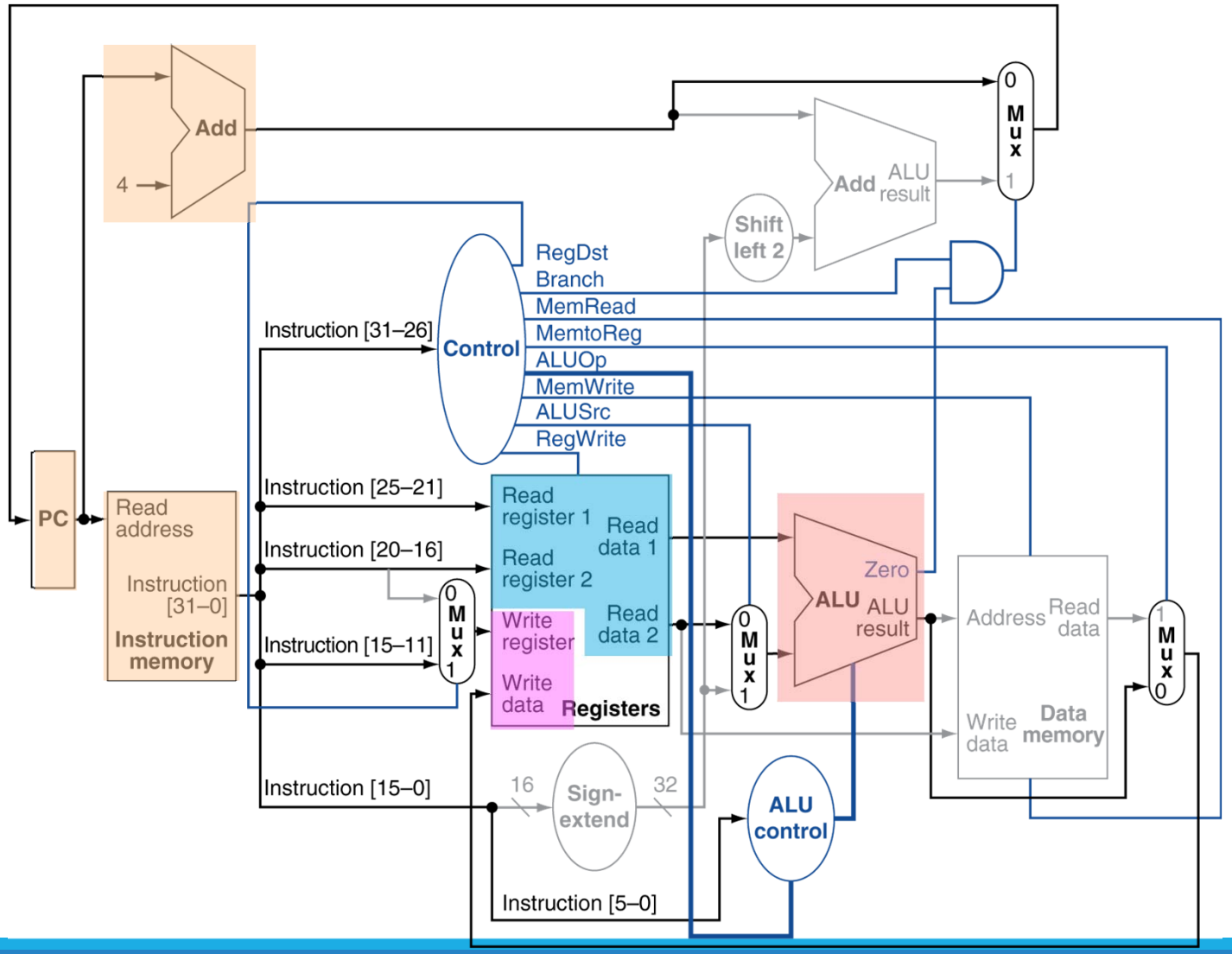
Datapath elements are assembled by wires and multiplexers

No datapath resource can be used more than once per instruction→ replication

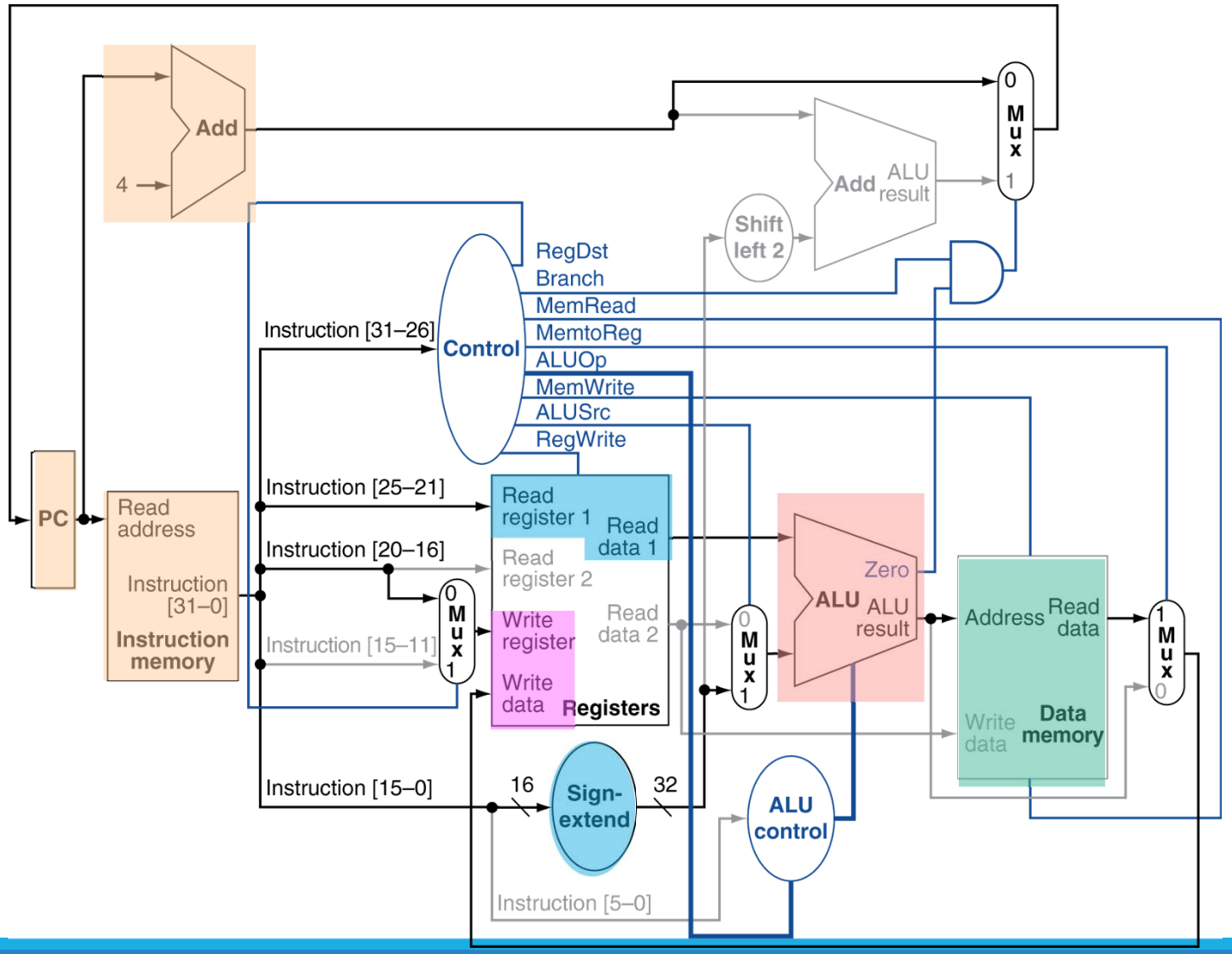
Multiplexors needed to select input to shared elements

Cycle time is determined by the length of the longest path

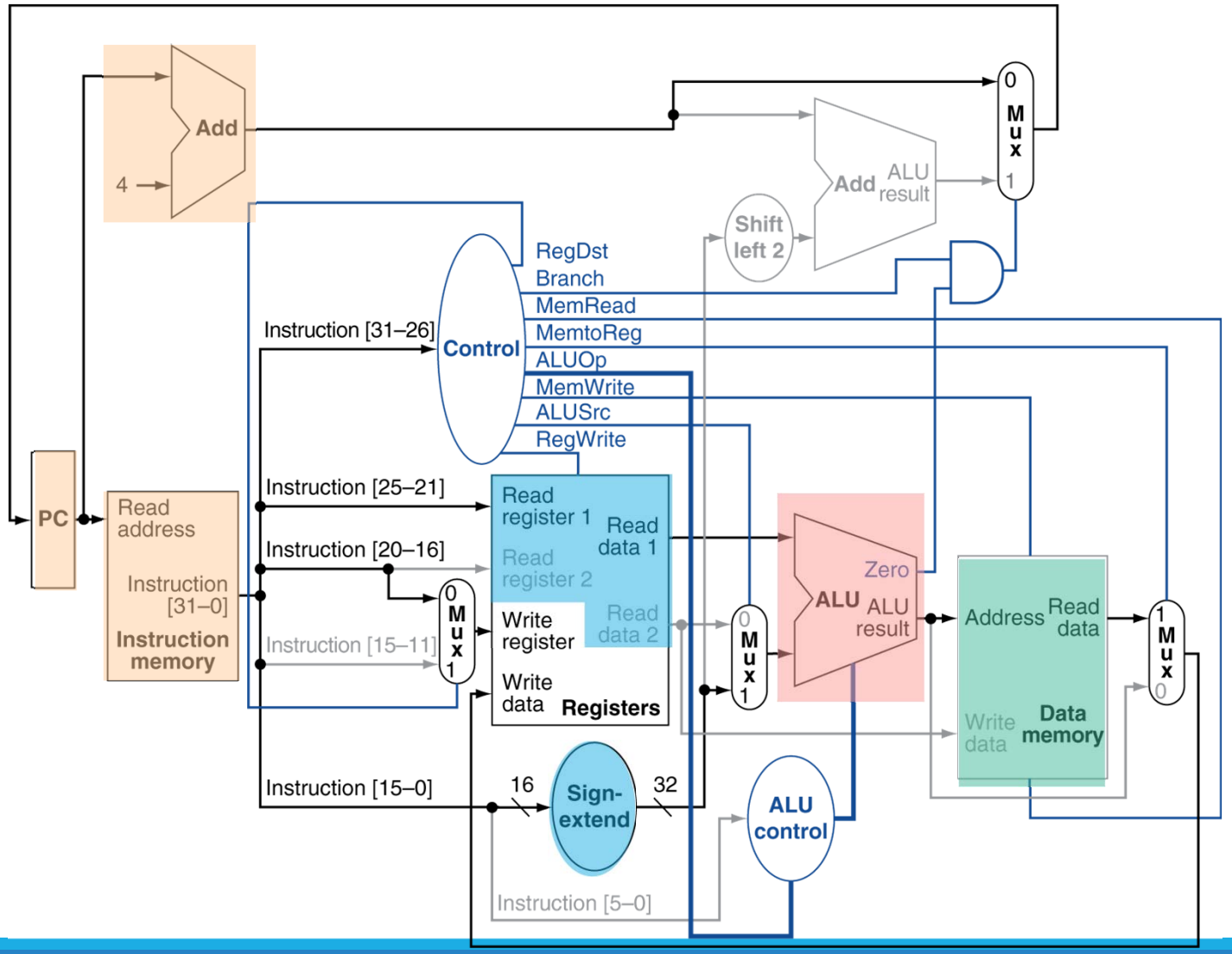
R-Type Instruction



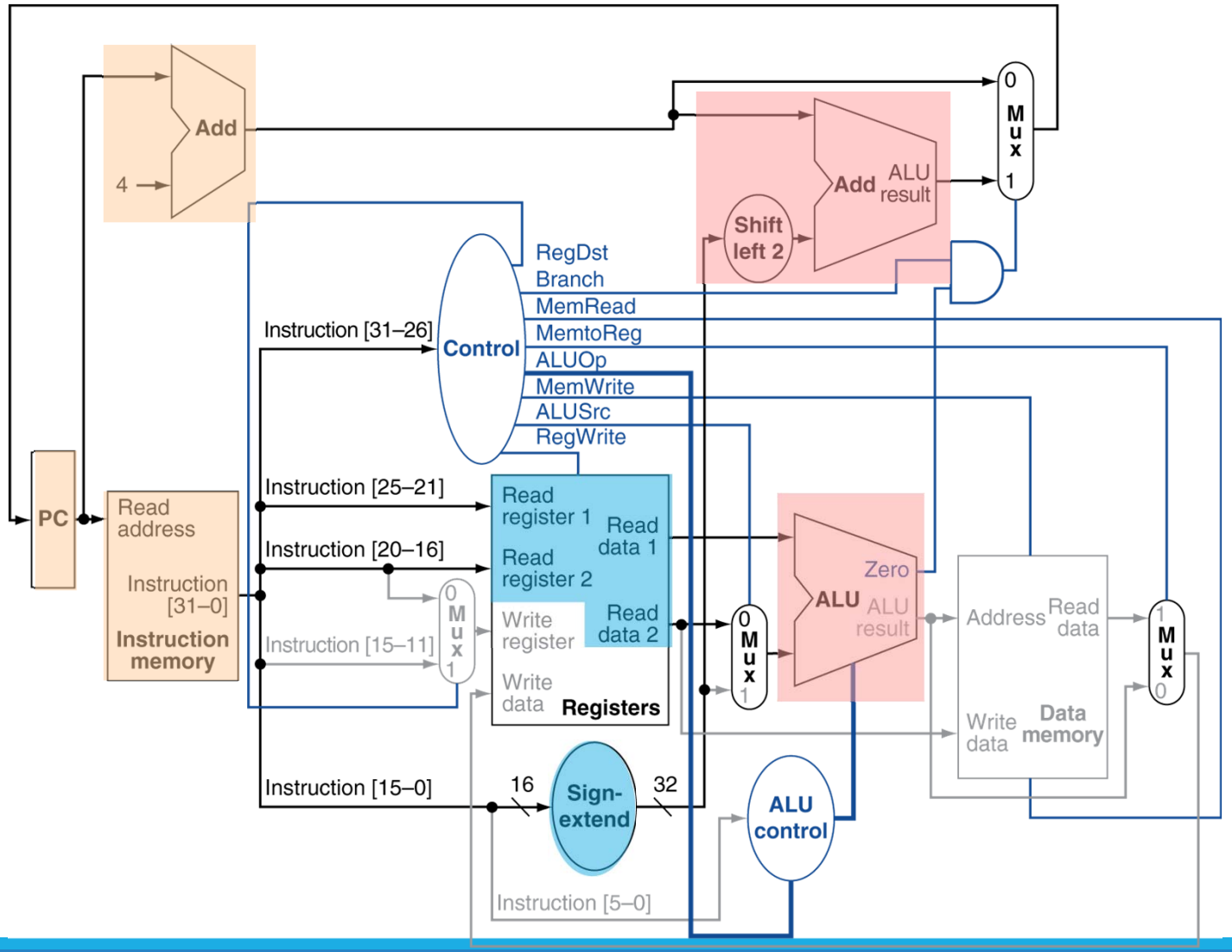
Load Instruction



Store Instruction



Branch-on-Equal Instruction



Instruction Execution Times

	Instruction Memory	Read Registers	ALU Operation	Data Memory	Write Register
Delay	200	100	200	200	100

R-type: $T = 200 + 100 + 200 + 100 = 600$

load: $T = 200 + 100 + 200 + 200 + 100 = 800$

store: $T = 200 + 100 + 200 + 200 = 700$

beq: $T = 200 + 100 + 200 = 500$

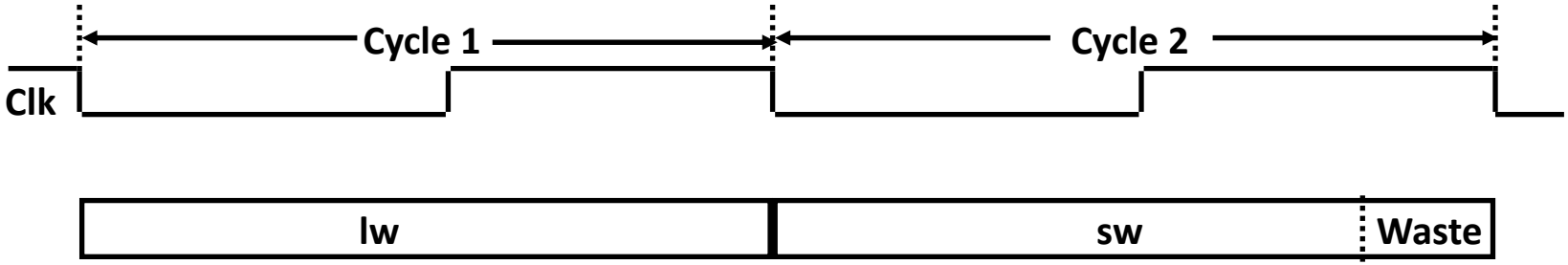
jump: $T = 200 + 100$

Assume negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times

Single Cycle Disadvantages & Advantages

✓ Simple and easy to understand

✗ Clock cycle is used inefficiently



✗ Waste of area because replication

The multi cycle datapath

Instructions last a variable number of cycles

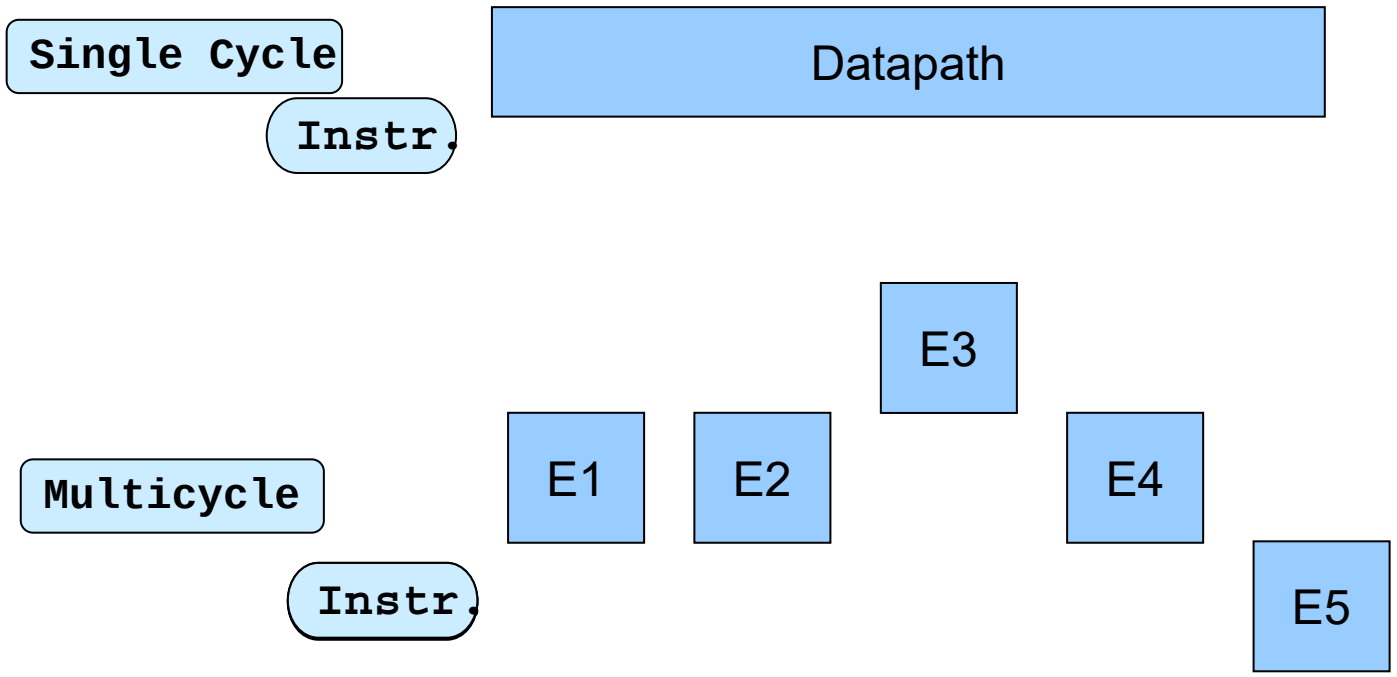
- more complex control
- less waste of cycle time

Datapath resources can be reused if in different stages

Cycle time is determined by the length of the longest stage

	Instruction Memory	Read Registers	ALU Operation	Data Memory	Write Register
Delay	200	100	200	200	100

Execution Comparison



Performance comparison

SPECINT2000 benchmark:

Instruction	Frequency
Loads	25%
Stores	10%
Branches	11%
Jumps	2%
R-type	52%

$$CPI = 0.25 \times 5 + 0.1 \times 4 + 0.11 \times 3 + 0.02 \times 2 + 0.52 \times 4$$

$$CPI = 4.1$$

Performance comparison

SPECINT2000 benchmark:

Single cycle:
- NI instructions
- CPI = 1
- Cycle time = 800 ps

Multi cycle:
- NI instructions
- CPI = 4.1
- Cycle time = 200 ps

$$\text{Speed}-\text{up} = \frac{NI \times 1 \times 800}{NI \times 4.1 \times 200} = 0.975$$

Instruction Level Parallelism

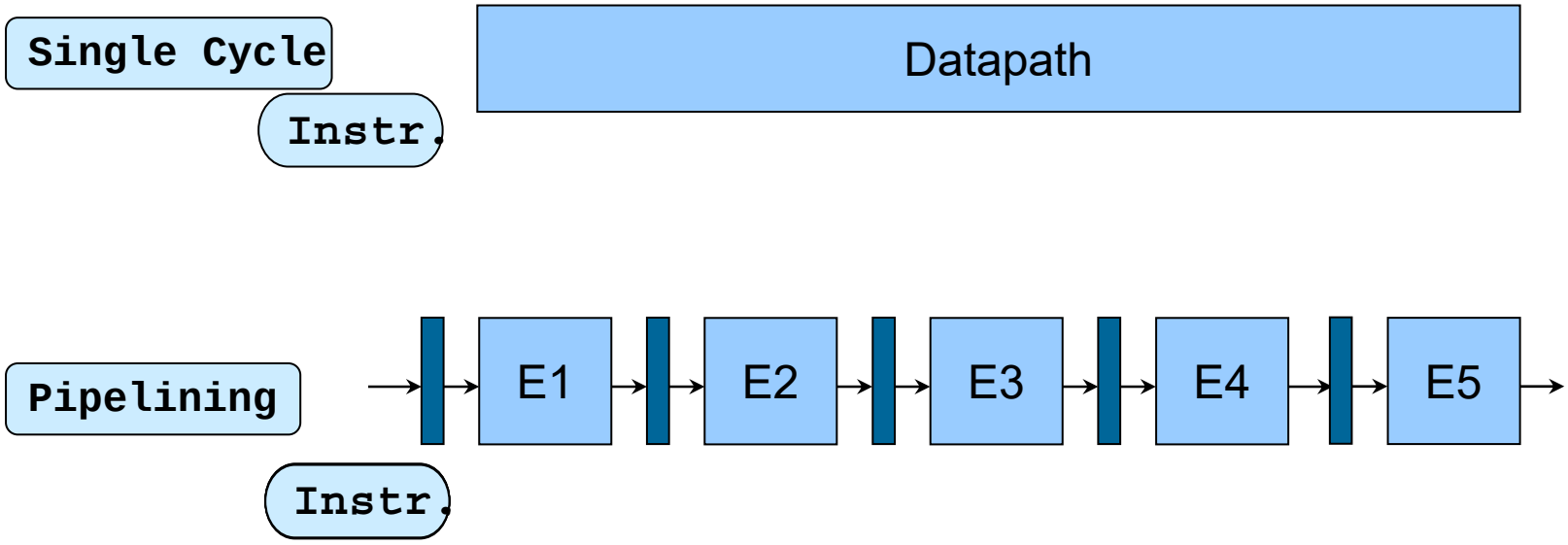
Execute instructions concurrently by

- Starting several instructions at the same time, or
- Partially overlapping execution: pipelining

Pipelining: datapath ≡ assembly line

Charlie Chaplin, Modern Times © Roy Export S.A.S.

Execution Comparison



Performance comparison

SPECINT2000 benchmark:

Single cycle:
- NI instructions
- CPI = 1
- Cycle time = 800 ps

Pipeline:
- NI instructions
- CPI ≈ 1
- Cycle time = 200 ps

$$\text{Speed}-\text{up} = \frac{NI \times 1 \times 800}{NI \times 1 \times 200} = 4$$

MIPS Pipeline

Five stages, one step per stage

IF

Instruction fetch from memory

ID

Instruction decode & register read

EX

Execute operation or calculate address

MEM

Access memory operand

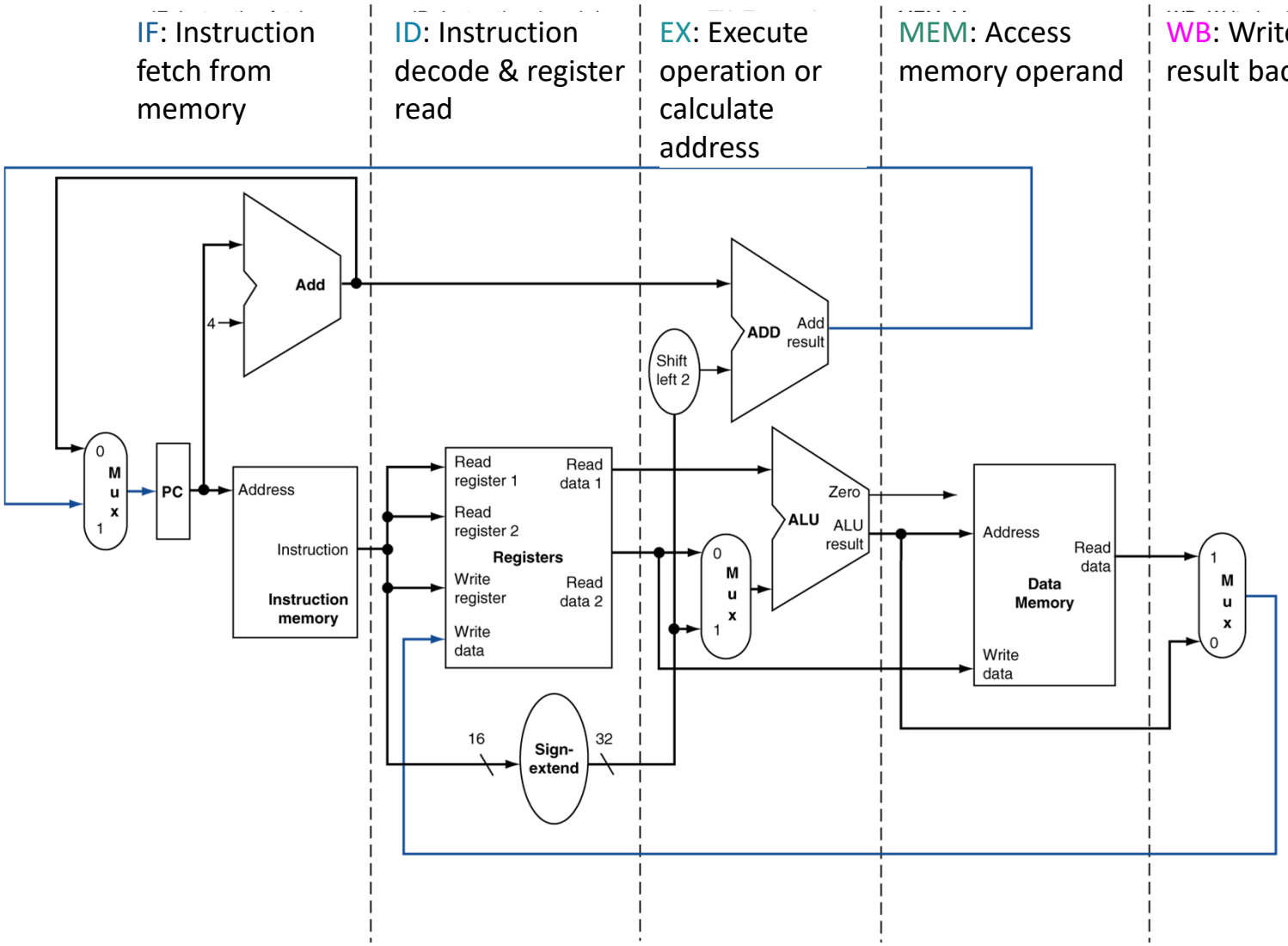
WB

Write result back to register

Pipelining and ISA Design

All MIPS instructions are 32-bits	• Easier to fetch in one cycle
Few and regular instruction formats	• Can decode and read registers in one step
Load/store addressing	• Compute address in 3rd stage, access memory in 4th stage
Alignment of memory operands	• Memory access takes only one cycle

Pipeline Design



Pipeline Design

Let execute next code:

```
lw $1, 0($2)
beq $3, $4, L1
sub $6, $7, $8
sw $9, 0($10)
add $11, $12, $13
```

Pipeline Design

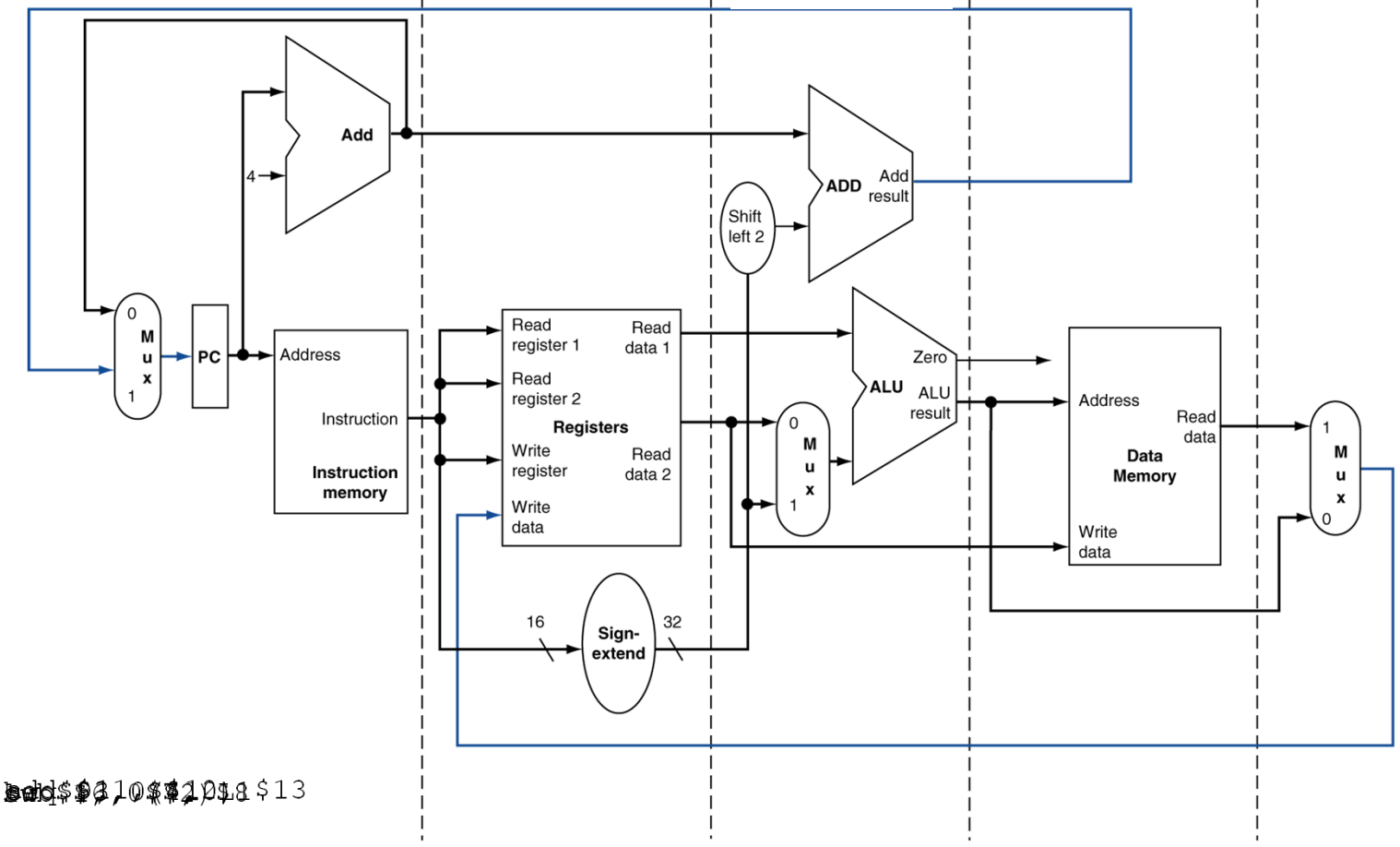
IF: Instruction fetch from memory

ID: Instruction decode & register read

EX: Execute operation or calculate address

MEM: Access memory operand

WB: Write result back



Pipeline Design

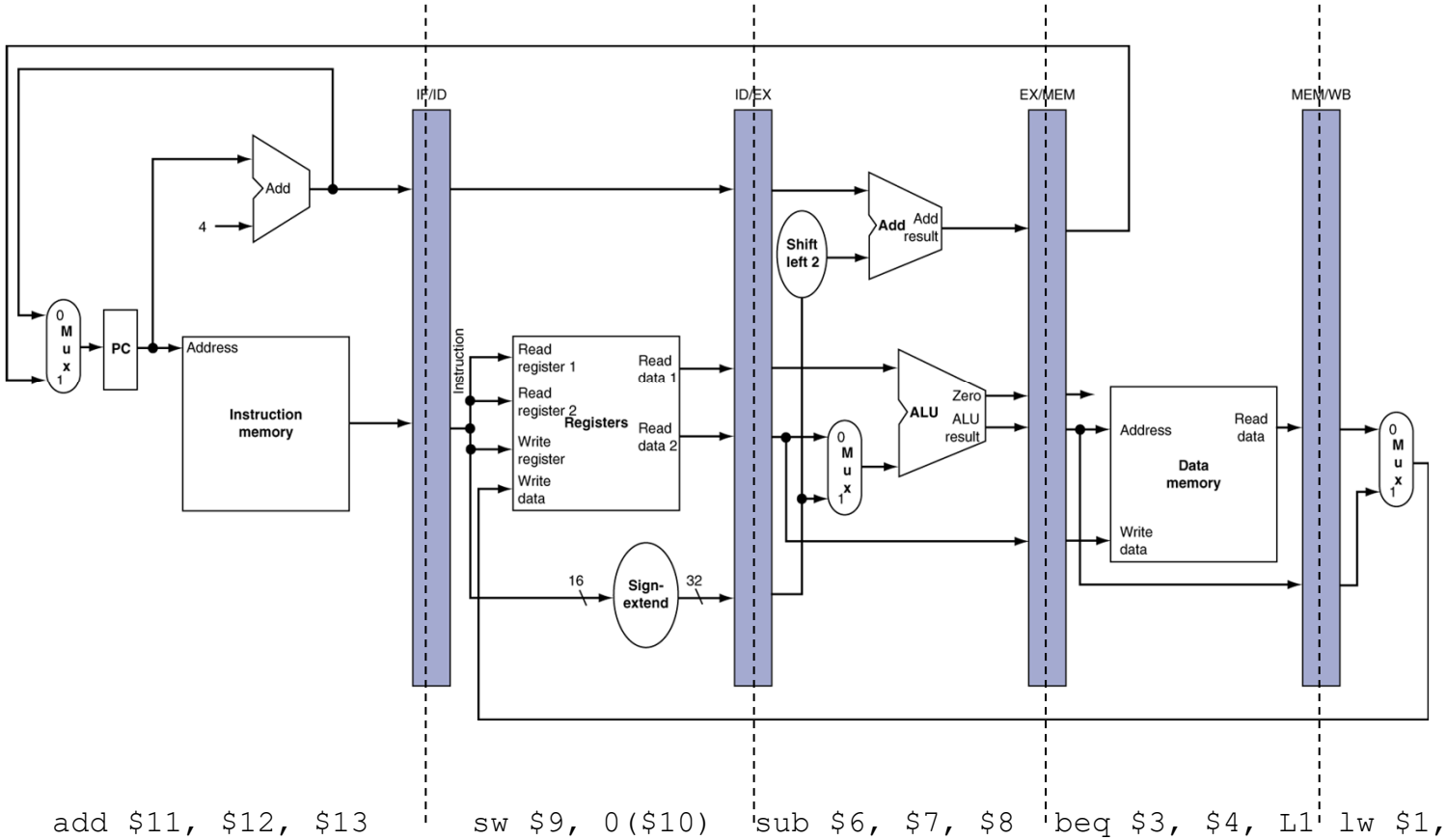
IF: Instruction fetch from memory

ID: Instruction decode & register read

EX: Execute operation or calculate address

MEM: Access memory operand

WB: Write result back




```
add $11, $12, $13    sw $9, 0($10)    sub $6, $7, $8    beq $3, $4, L1    lw $1, 0($2)
```

Pipeline Design

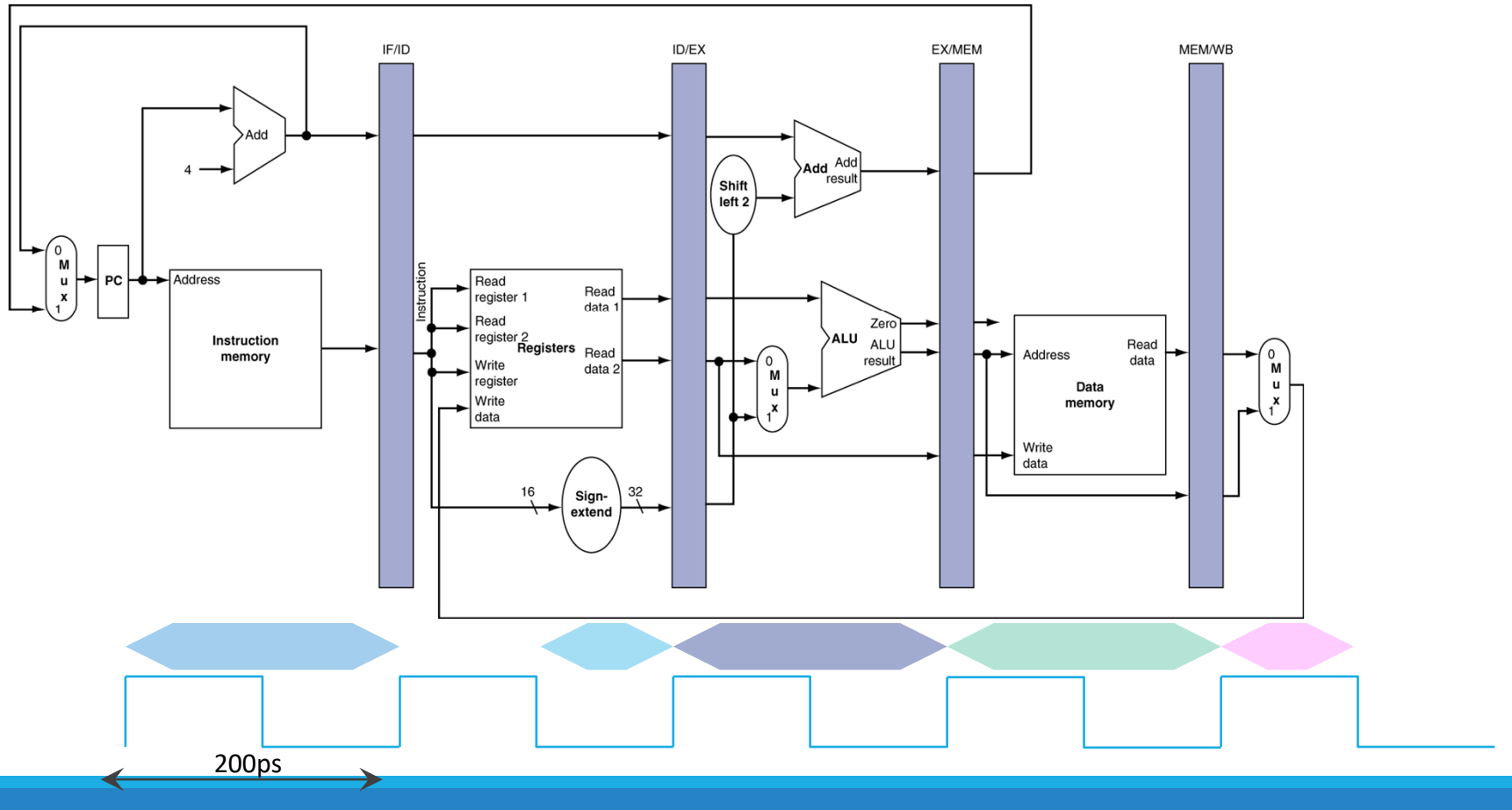
IF: Instruction fetch from memory

ID: Instruction decode & register read

EX: Execute operation or calculate address

MEM: Access memory operand

WB: Write result back



Hazards

Situations that prevent starting the next instruction in the next cycle

Structure hazards

A required resource is busy

Data hazard

Need to wait for previous instruction to complete its data read/write

Control hazard

Deciding on control action depends on previous instruction

Data Hazards

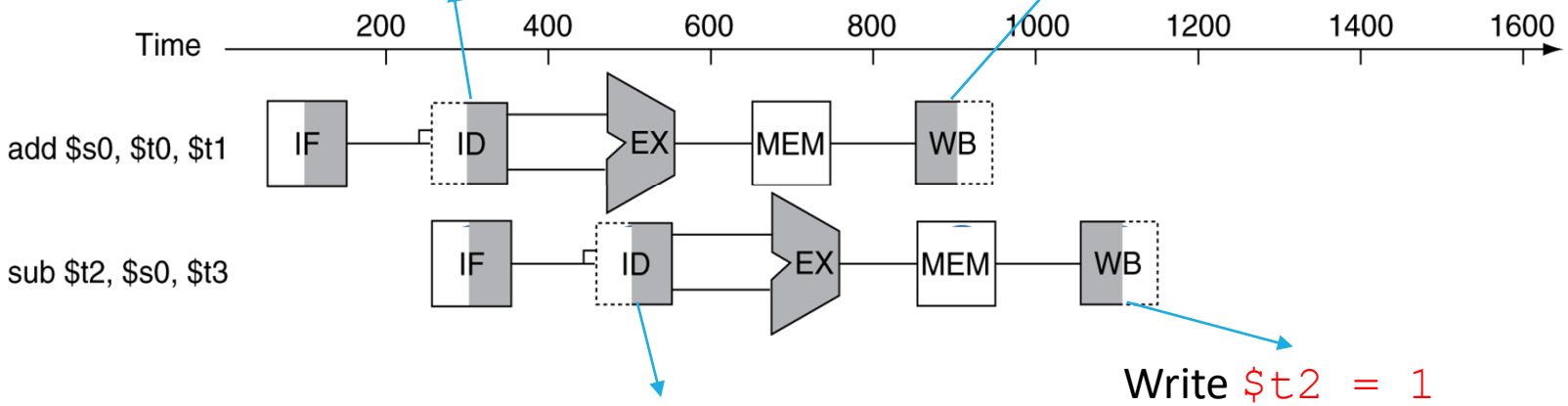
An instruction depends on completion of data access by a previous instruction

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

```
$t0 = 0
$t1 = 1
$t2 = 2
$t3 = 3
$s0 = 4
```

Read $t0 = 0
and $t1 =1

Write $s0 = 1



Read $s0= 4
and $t3 = 3

Write $t2 = 1

Solving data hazards

Software methods

- Inserting *nops*
- Reordering code

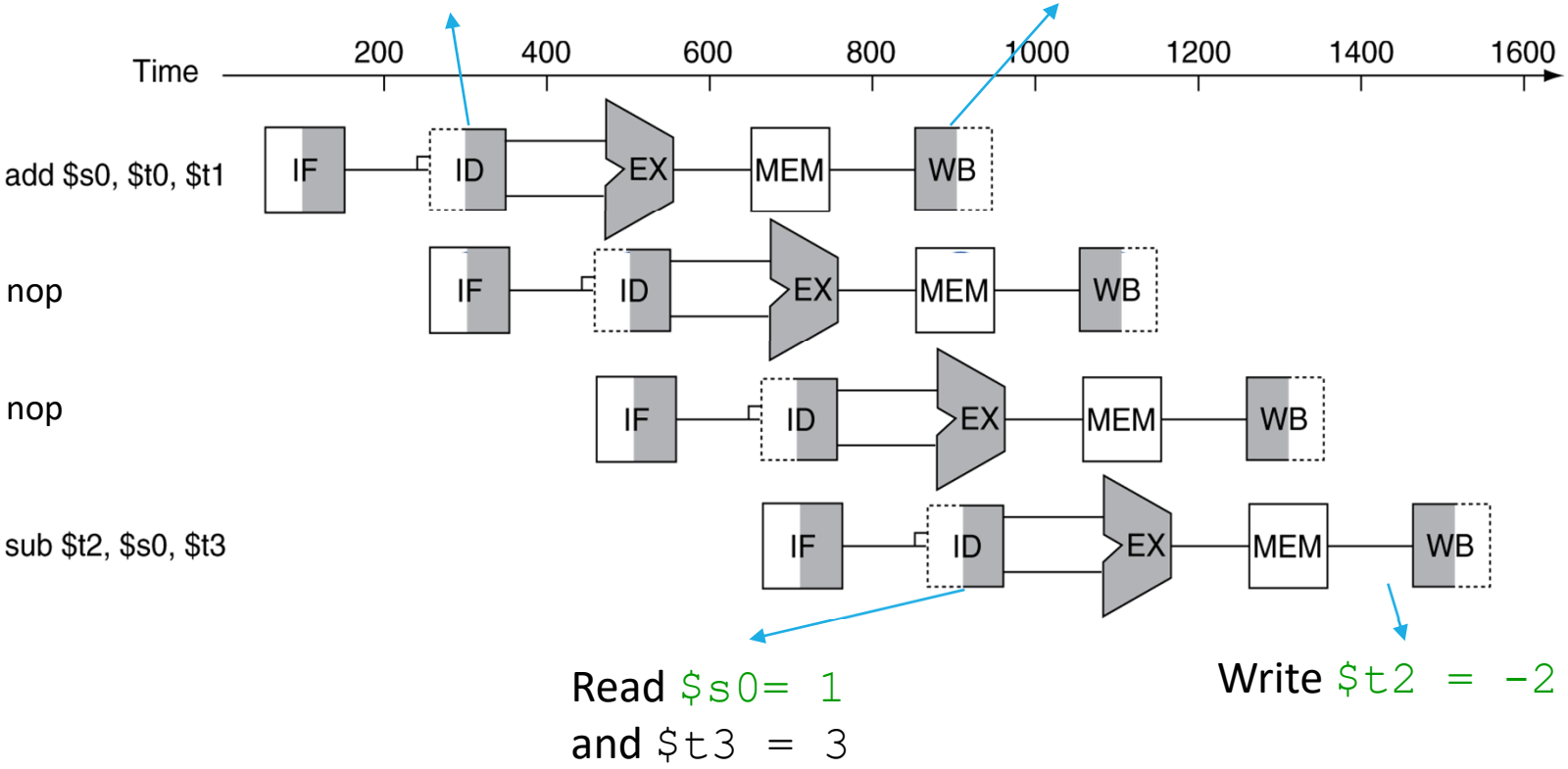
Hardware methods

- Inserting bubbles (*stalling*)
- Forwarding

Inserting nops

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

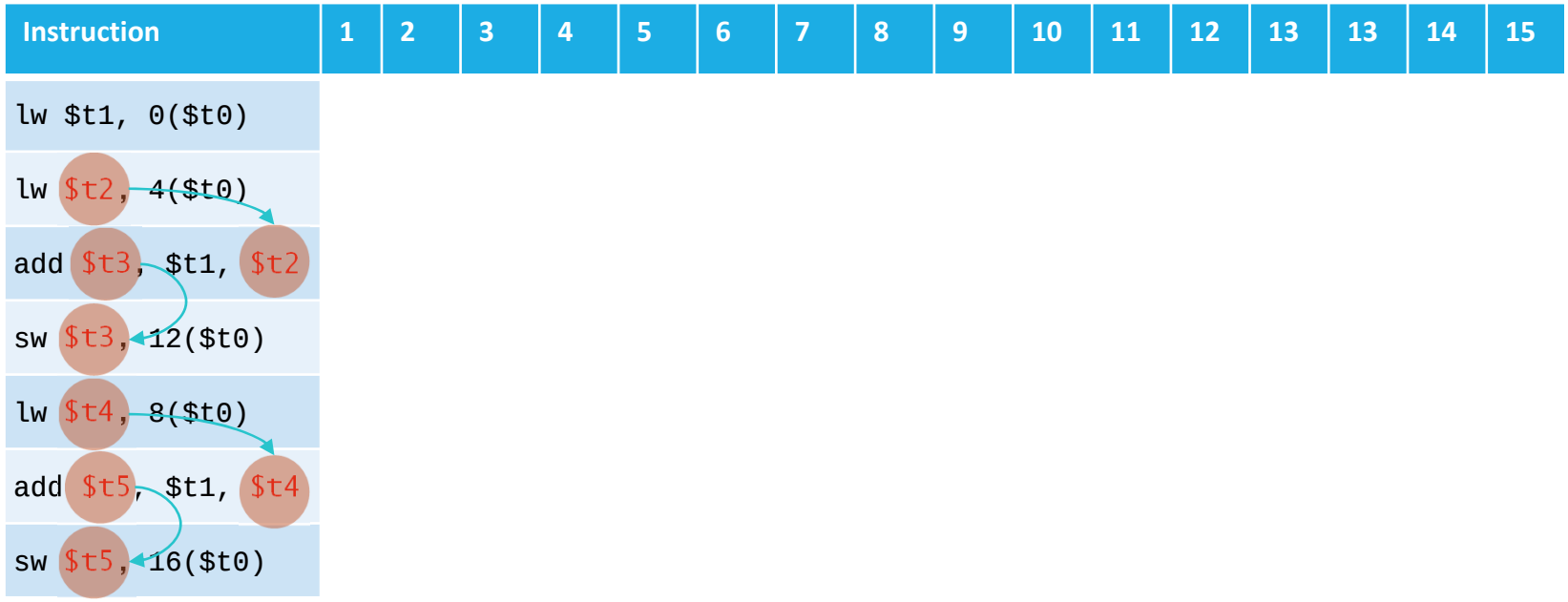
```
$t0 = 0
$t1 = 1
$t2 = 2
$t3 = 3
$s0 = 4
```



Cycles Diagram Representation

Instruction	1
`add $s0, $t0, $t1`	IF
`nop`	
`nop`	
`sub $t2, $s0, $t3`	

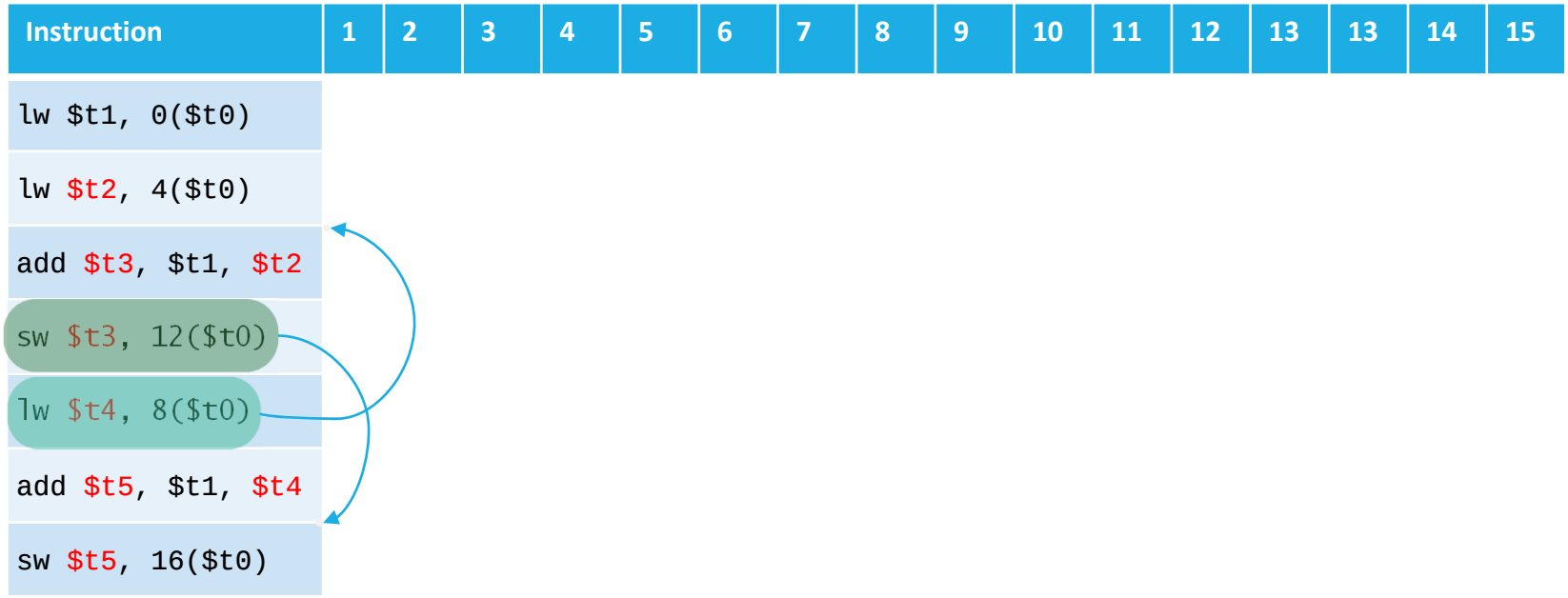
Code Scheduling to Avoid Stalls



Code Scheduling to Avoid Stalls

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`lw $t1, 0($t0)`	IF	ID	EX	M	WB														
`lw $t2, 4($t0)`		IF	ID	EX	M	WB													
`nop`			IF	ID	EX	M	WB												
`nop`				IF	ID	EX	M	WB											
`add $t3, $t1, $t2`					IF	ID	EX	M	WB										
`nop`						IF	ID	EX	M	WB									
`nop`							IF	ID	EX	M	WB								
`sw $t3, 12($t0)`								IF	ID	EX	M	WB							
`lw $t4, 8($t0)`									IF	ID	EX	M	WB						
`nop`										IF	ID	EX	M	WB					
`nop`											IF	ID	EX	M	WB				
`add $t5, $t1, $t4`												IF	ID	EX	M	WB			
`nop`													IF	ID	EX	M	WB		
`nop`														IF	ID	EX	M	WB	
`sw $t5, 16($t0)`															IF	ID	EX	M	WB

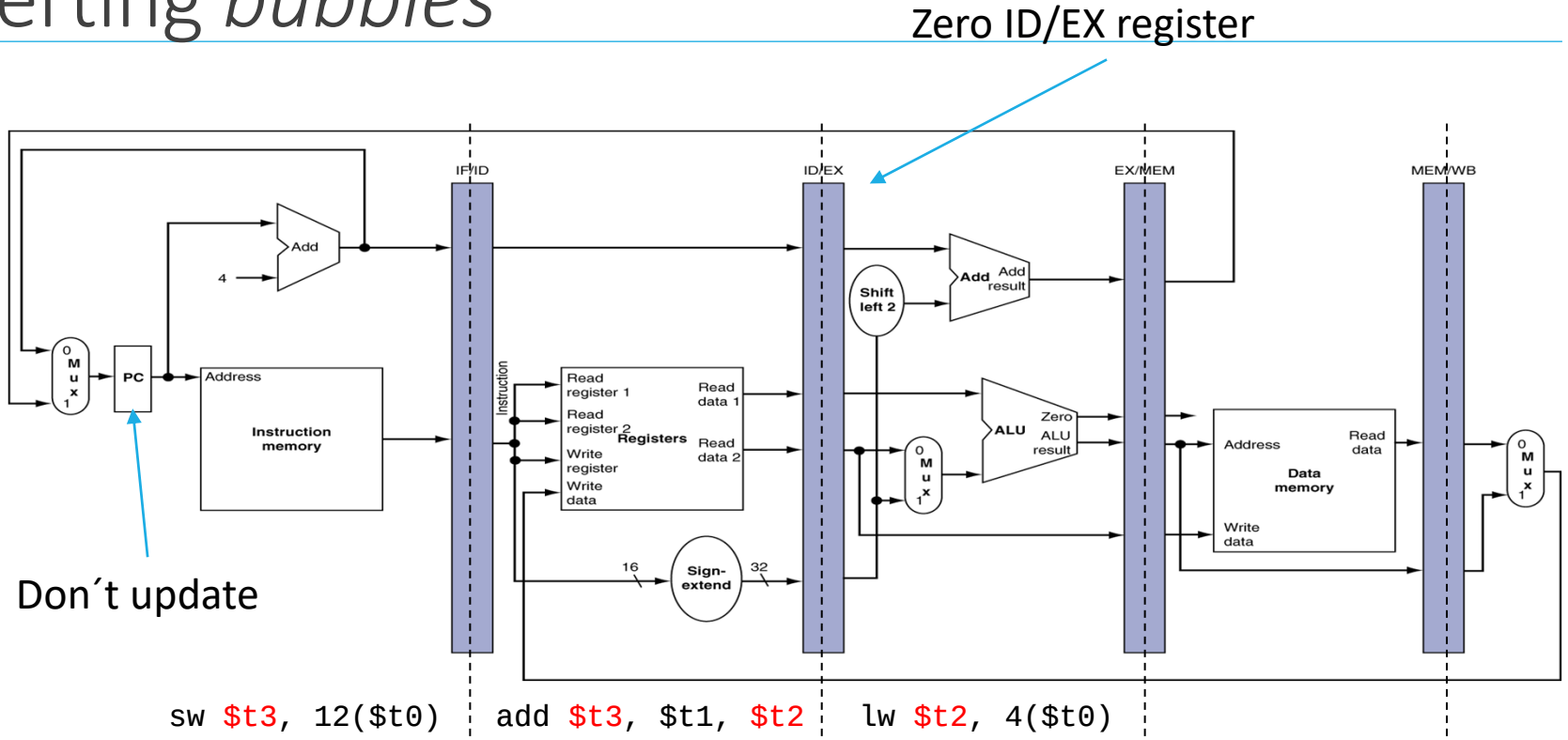
Code Scheduling to Avoid Stalls



Code Scheduling to Avoid Stalls

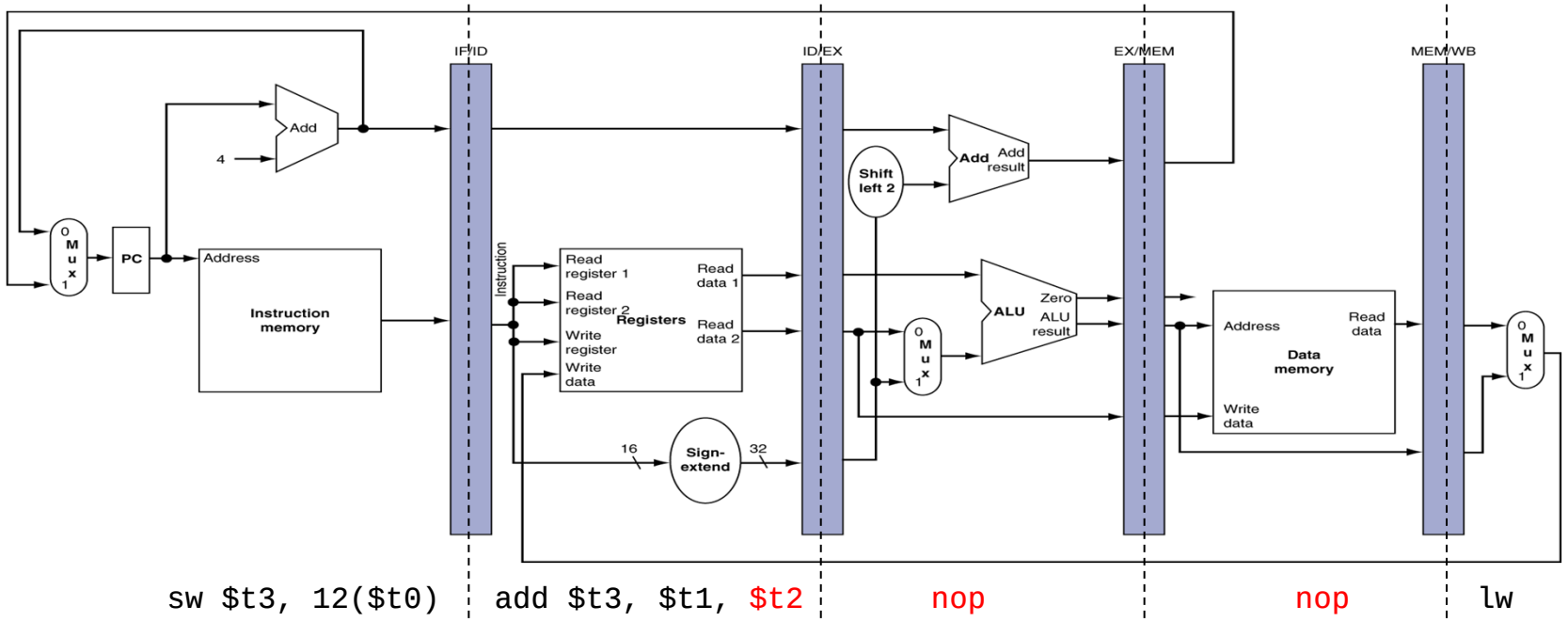
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	13	14	15
`lw $t1, 0($t0)`																
`lw $t2, 4($t0)`																
`lw $t4, 8($t0)`																
`nop`																
`add $t3, $t1, $t2`																
`add $t5, $t1, $t4`																
`nop`																
`sw $t3, 12($t0)`																
`sw $t5, 16($t0)`																

Inserting *bubbles*



Instruction
`lw $t2, 4($t0)`
`add $t3, $t1, $t2`
`sw $t3, 12($t0)`

Inserting *bubbles*



Instruction	1	2	3	4	5	6	7	8
`lw $t2, 4($t0)`	IF	ID	EX	M	WB			
`add $t3, $t1, $t2`		IF			ID	EX	M	WB
`sw $t3, 12($t0)`					IF			ID

Solving data hazards

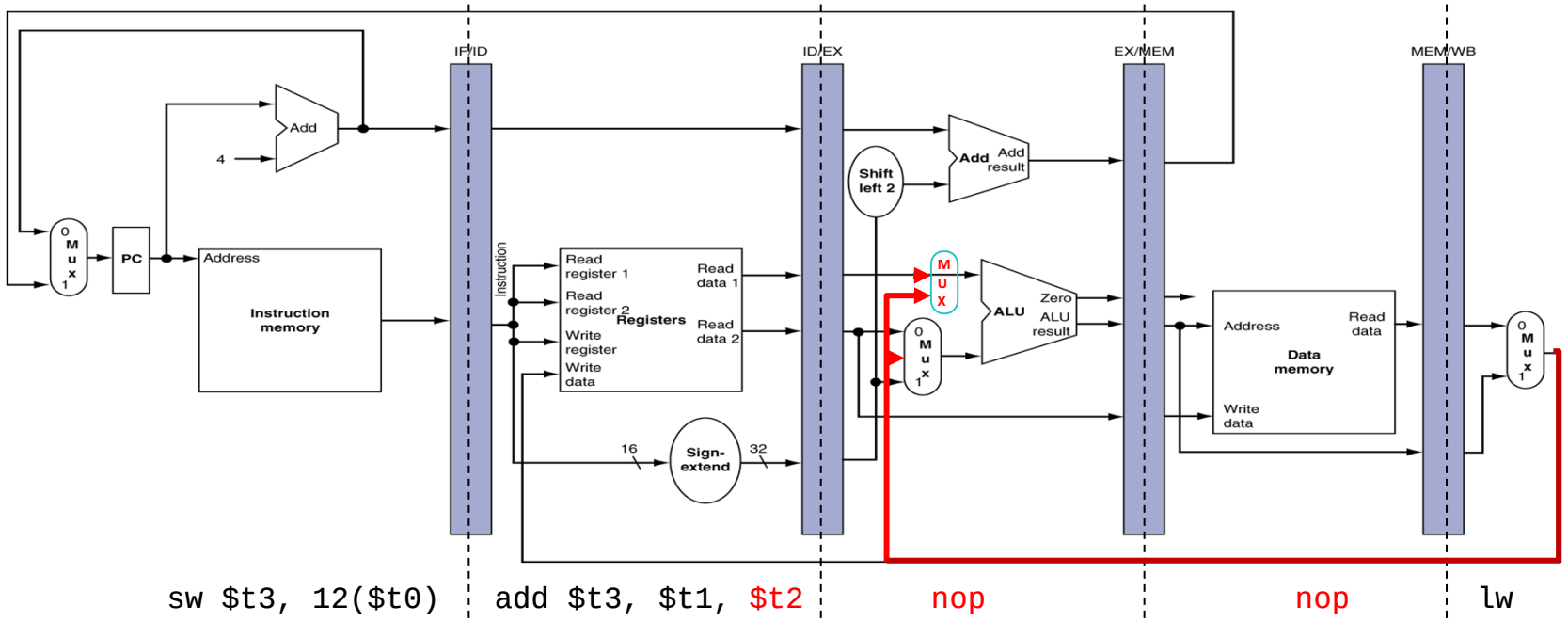
Software methods

- Inserting *nops*
- Reordering code

Hardware methods

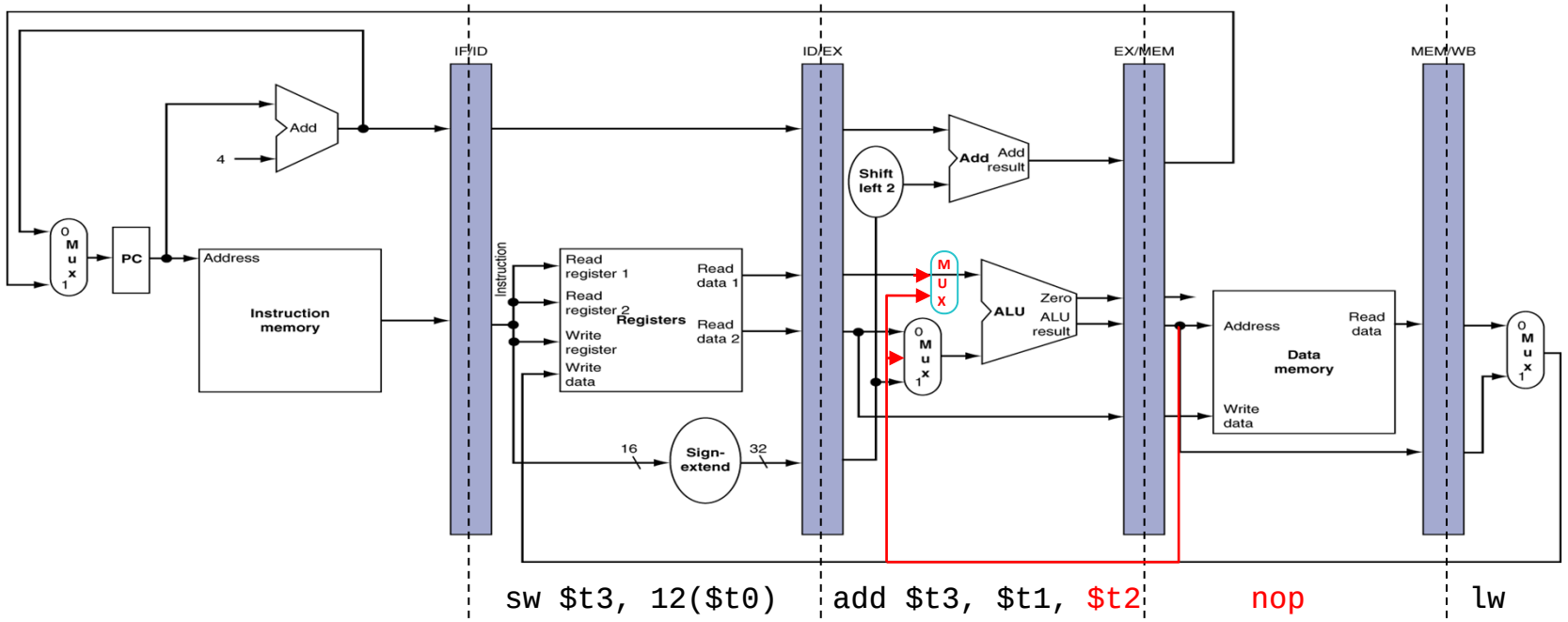
- Inserting bubbles (*stalling*)
- Forwarding

Forwarding



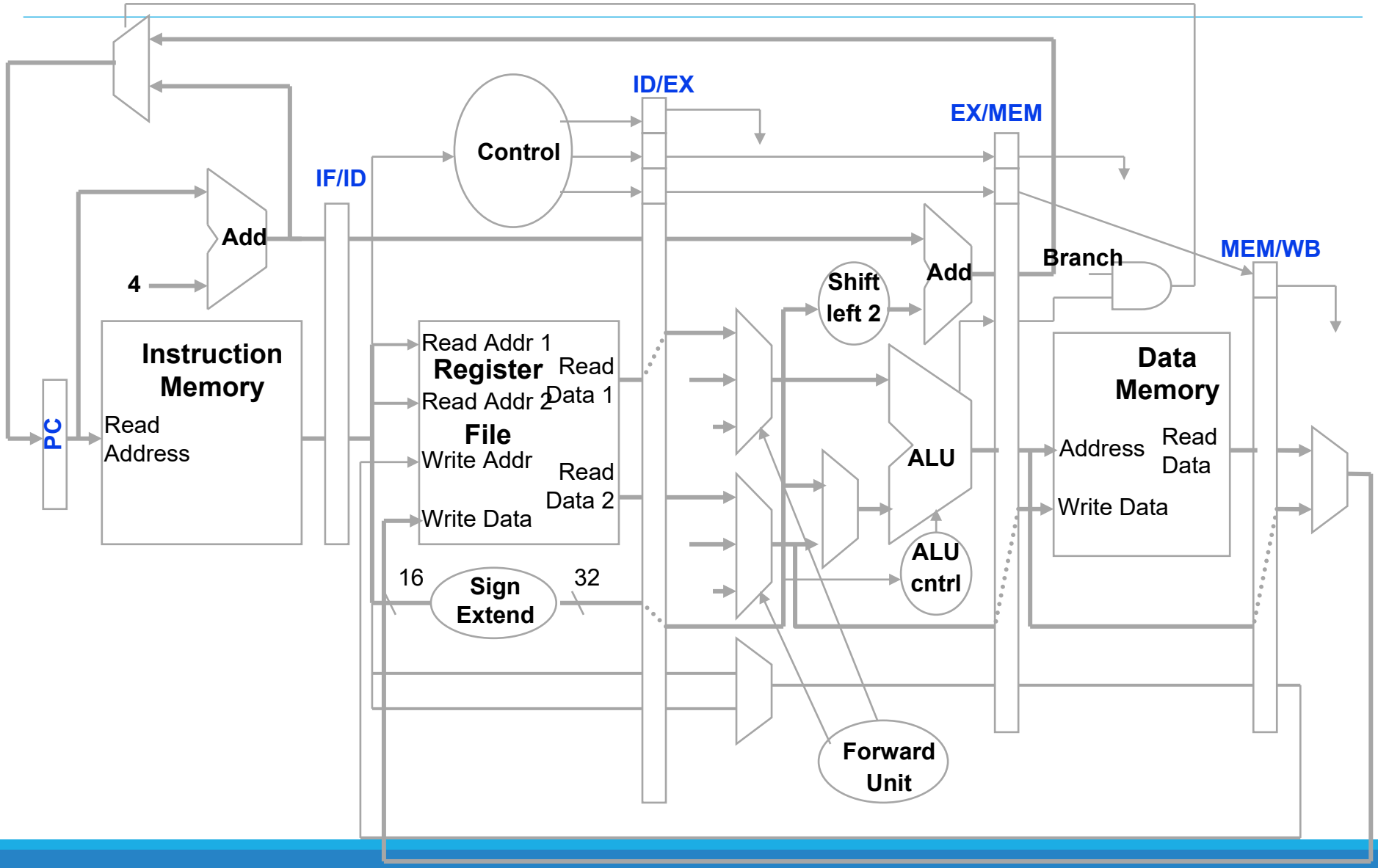
Instruction
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)

Forwarding

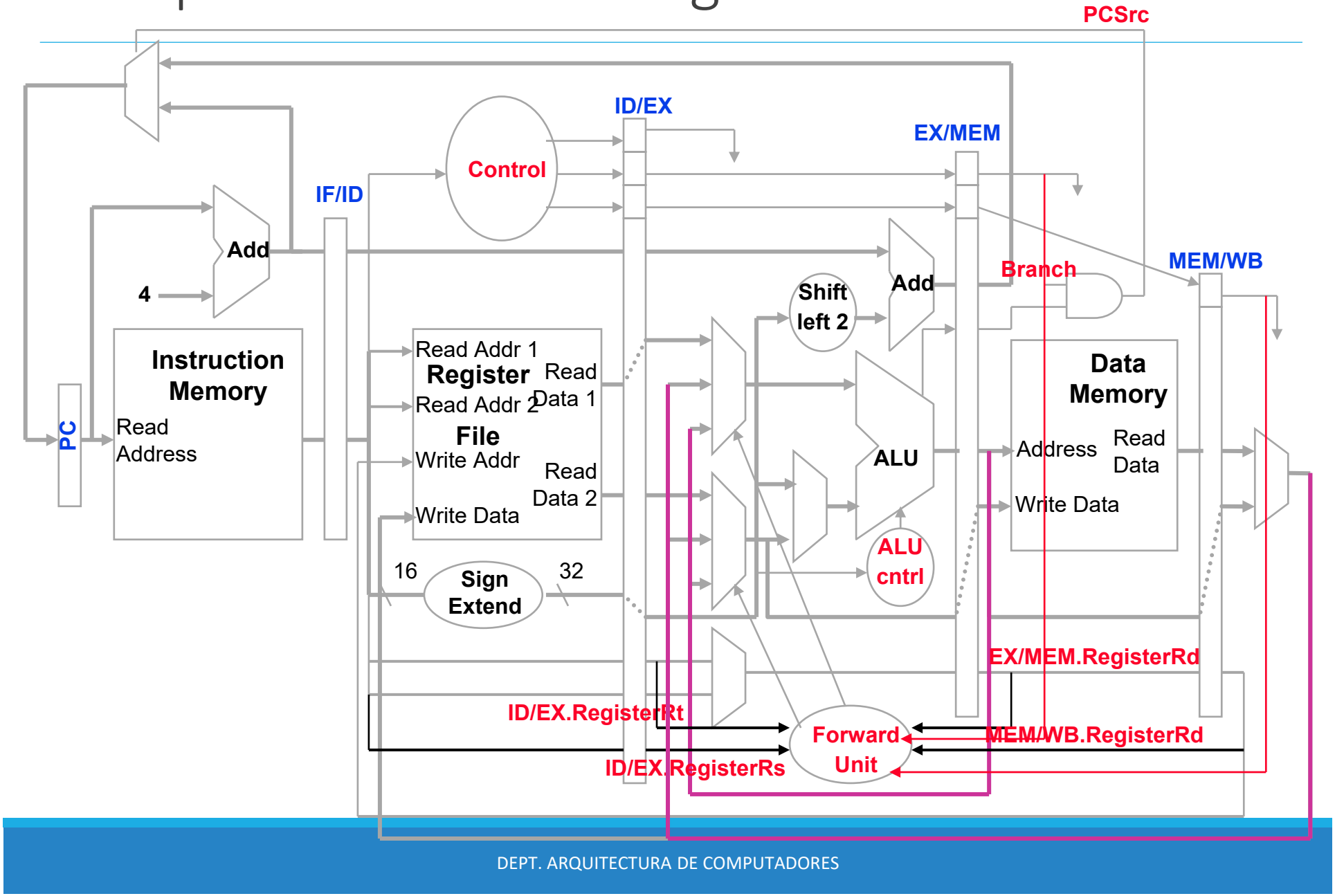


Instruction	1	2	3	4	5	6	7	8
`lw $t2, 4($t0)`	IF	ID	EX	M	WB			
`add $t3, $t1, $t2`		IF		ID	EX	M	WB	
`sw $t3, 12($t0)`				IF	ID	EX	M	WB

Datapath with forwarding hardware



Datapath with forwarding hardware



Data forwarding control conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
     ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
     ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

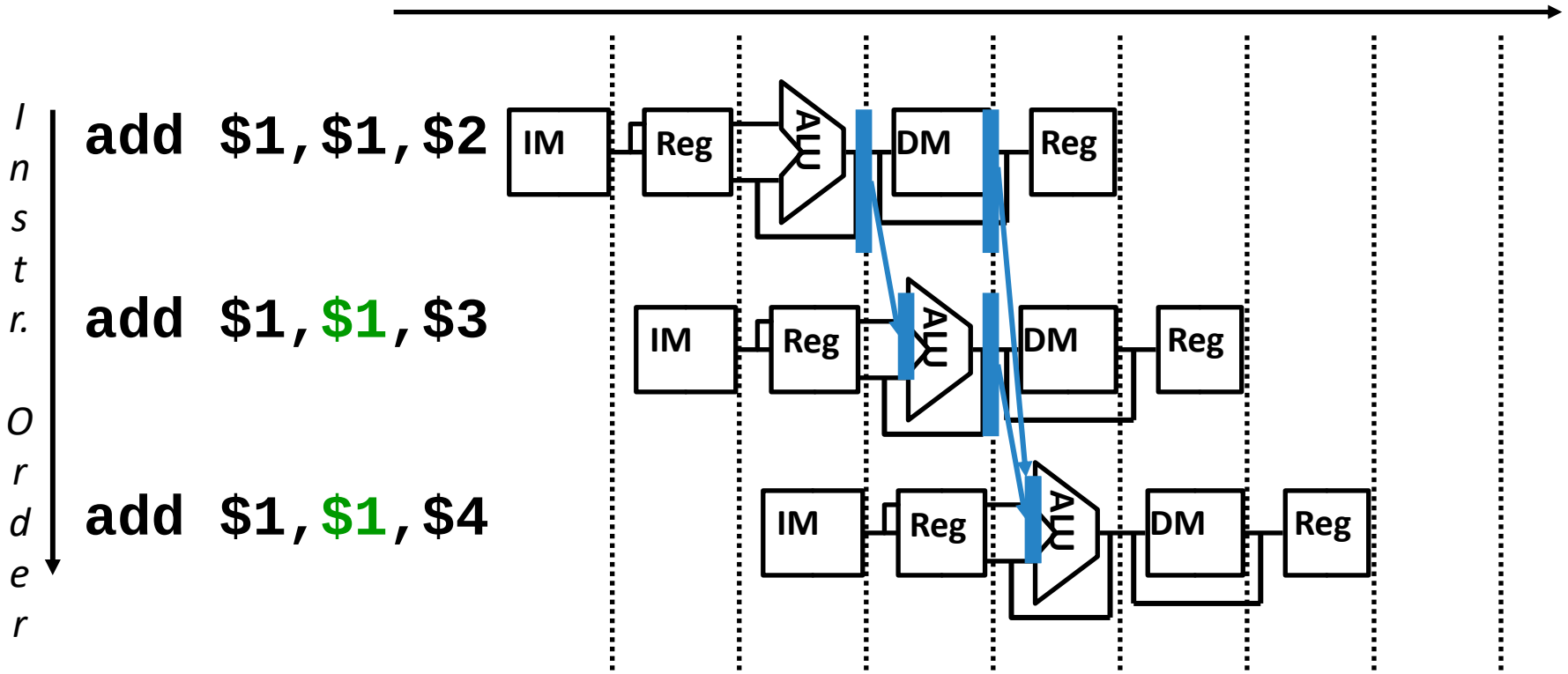
2. MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU

Yet Another Complication!

Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



Data forwarding control conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

2. MEM Forward Unit:

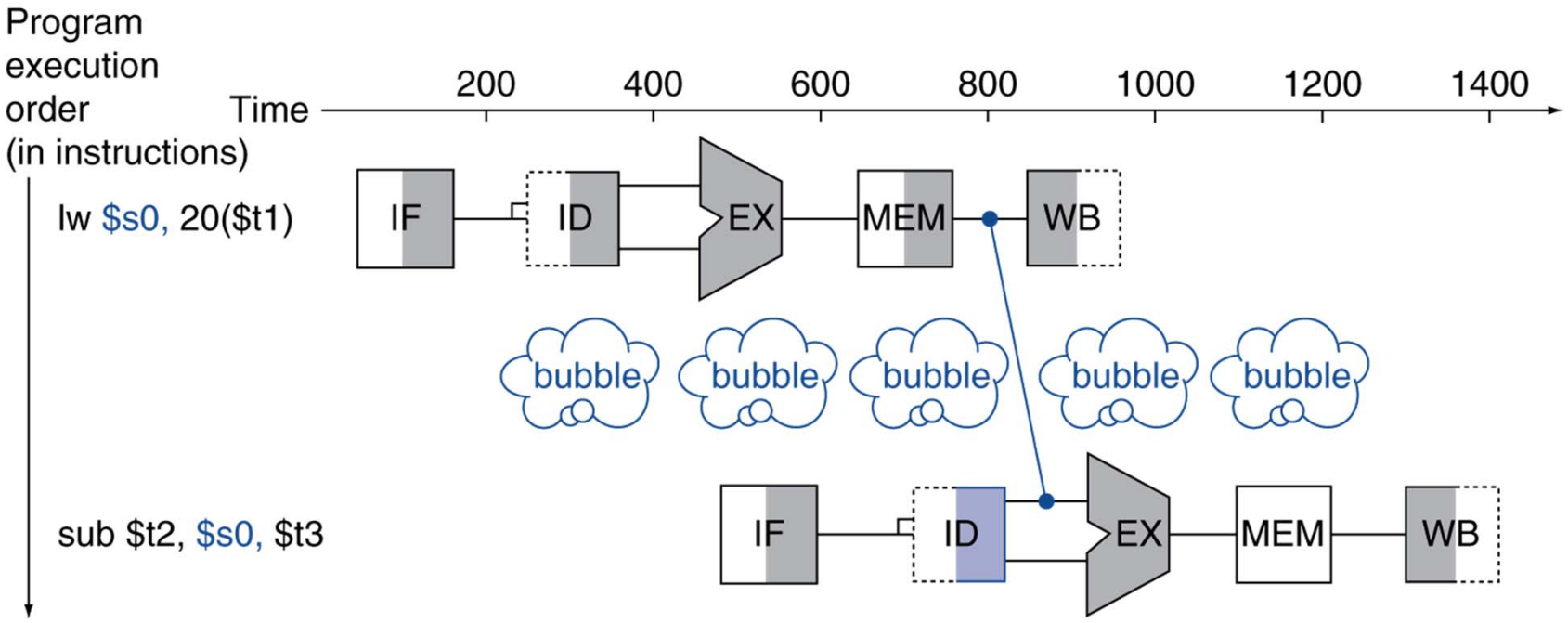
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the previous or second previous instr. to either input of the ALU

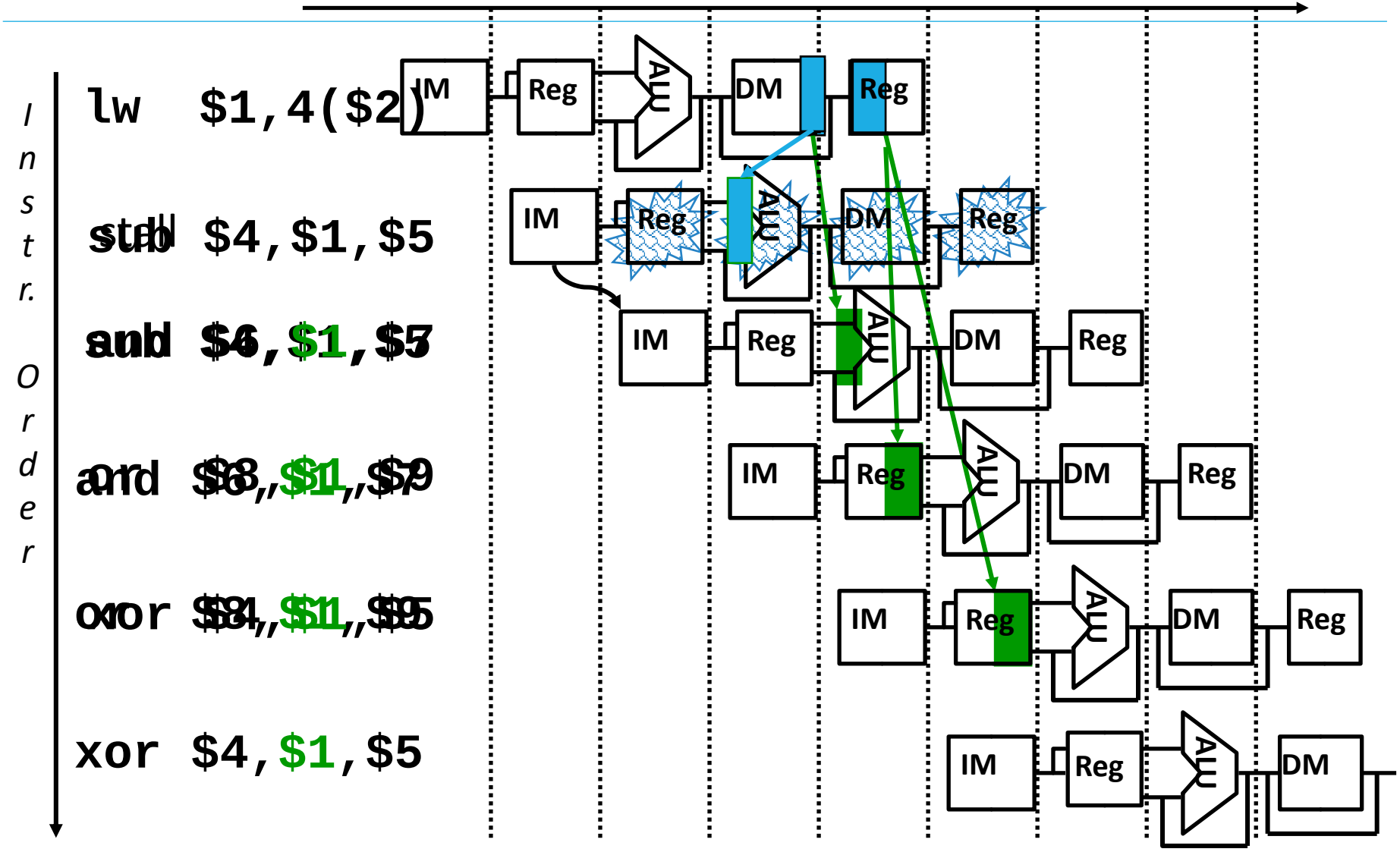
Load-Use Data Hazard

Can’t always avoid stalls by forwarding

- If value not computed when needed, then can’t forward backward in time!



Forwarding with Load-use Data Hazards



Load-use Hazard Detection Unit

Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

3. ID Hazard detection Unit:

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

Hazards

Situations that prevent starting the next instruction in the next cycle

Structure hazards

A required resource is busy

Data hazard

Need to wait for previous instruction to complete its data read/write

Control hazard

Deciding on control action depends on previous instruction

Control Hazards

Delay decision (requires compiler support)

Stall

Branch prediction

- Static prediction
  - Predict always branch not taken

Get outcome earlier, in ID stage

Advanced branch prediction

- Dynamic prediction
  - 1-bit predictor
  - 2-bits predictor
  - Correlated predictor
  - Tournament predictor, etc

Control Hazards: delay decision

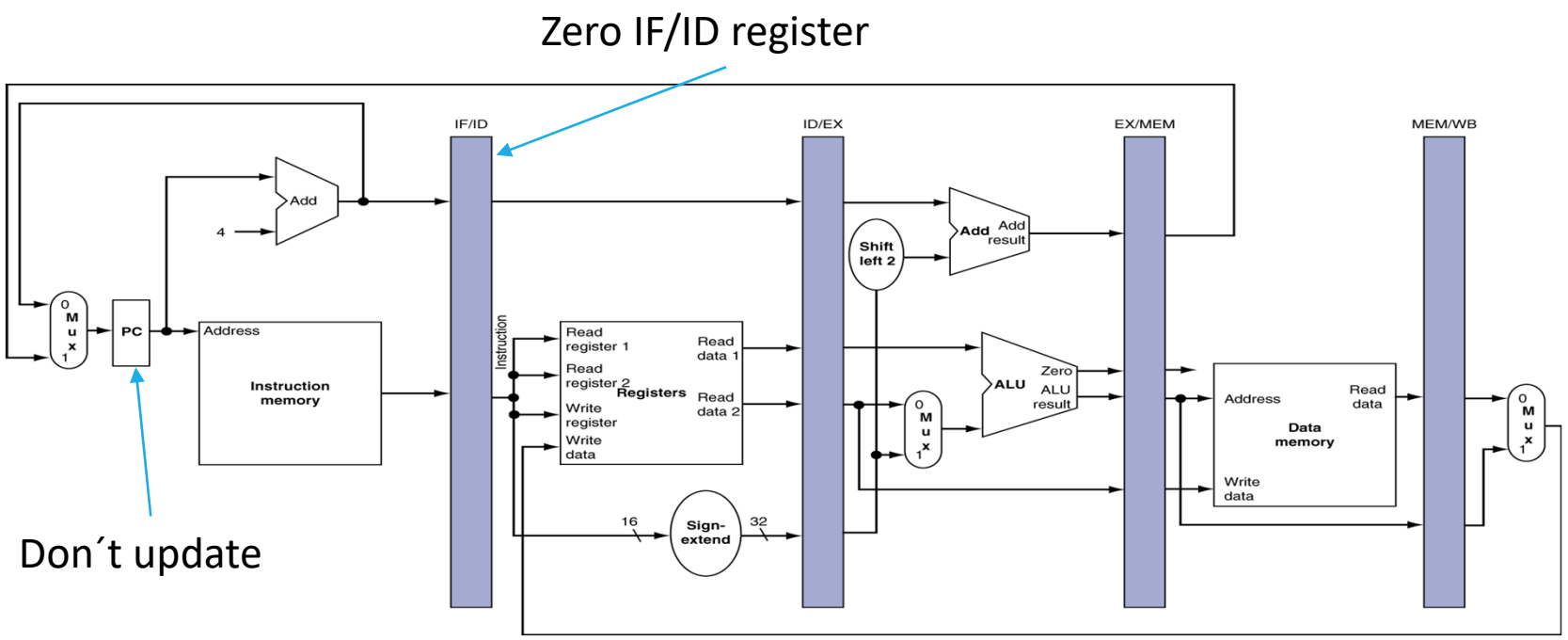
Move unrelated instructions after Branch

If there aren't suitable instructions, insert `nops`

Instruction	1	2	3	4	5	6	7	8
`sub $5, $3, $2`	IF	ID	EX	M	WB			
`add $3, $1, $2`		IF	ID	EX	M	WB		
`beq $2, $4, L1`			IF	ID	EX	M	WB	
…								
`L1:sw $4, 12($2)`								

Instruction	1	2	3	4	5	6	7	8
`beq $2, $4, L1`	IF	ID	EX	M	WB			
`sub $5, $3, $2`		IF	ID	EX	M	WB		
`add $3, $1, $2`			IF	ID	EX	M	WB	
…								
`L1:sw $4, 12($2)`				IF	ID	EX	M	WB

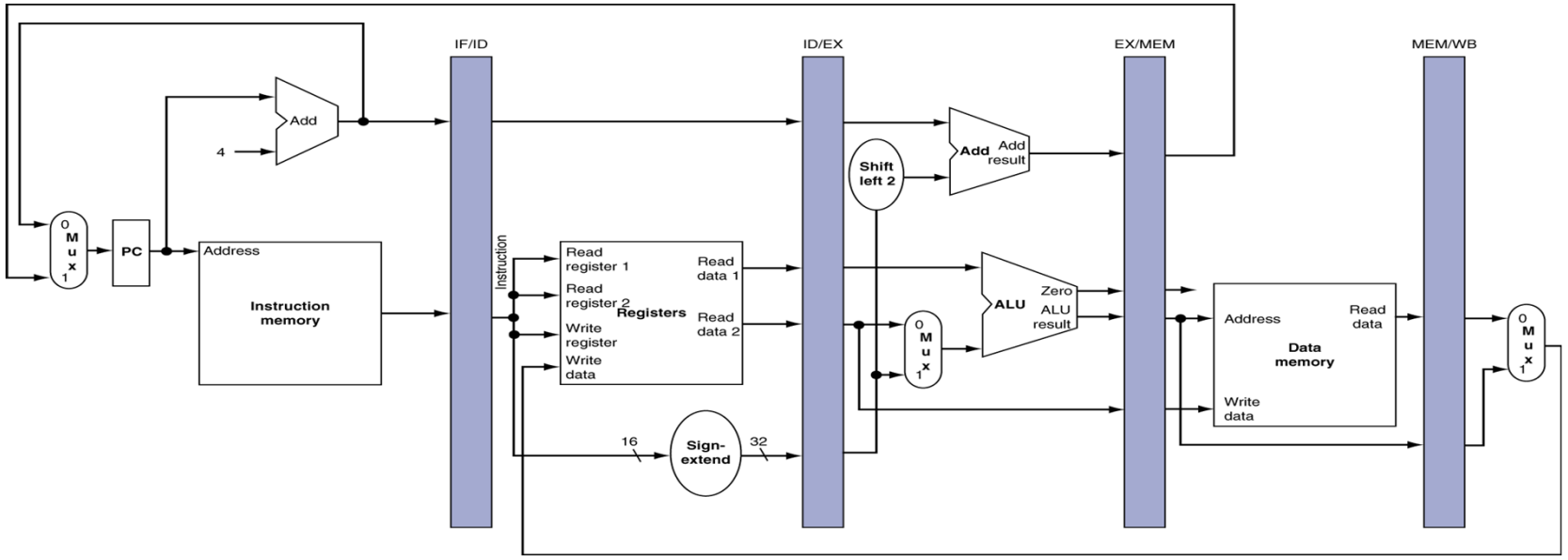
Control Hazards: stall



Branch outcome is taken

Instruction	1	2	3	4	5	6	7	8
`beq $2, $4, L1`	IF	ID						
`add $3, $1, $2`		IF						
`L1:sw $4, 12($2)`								

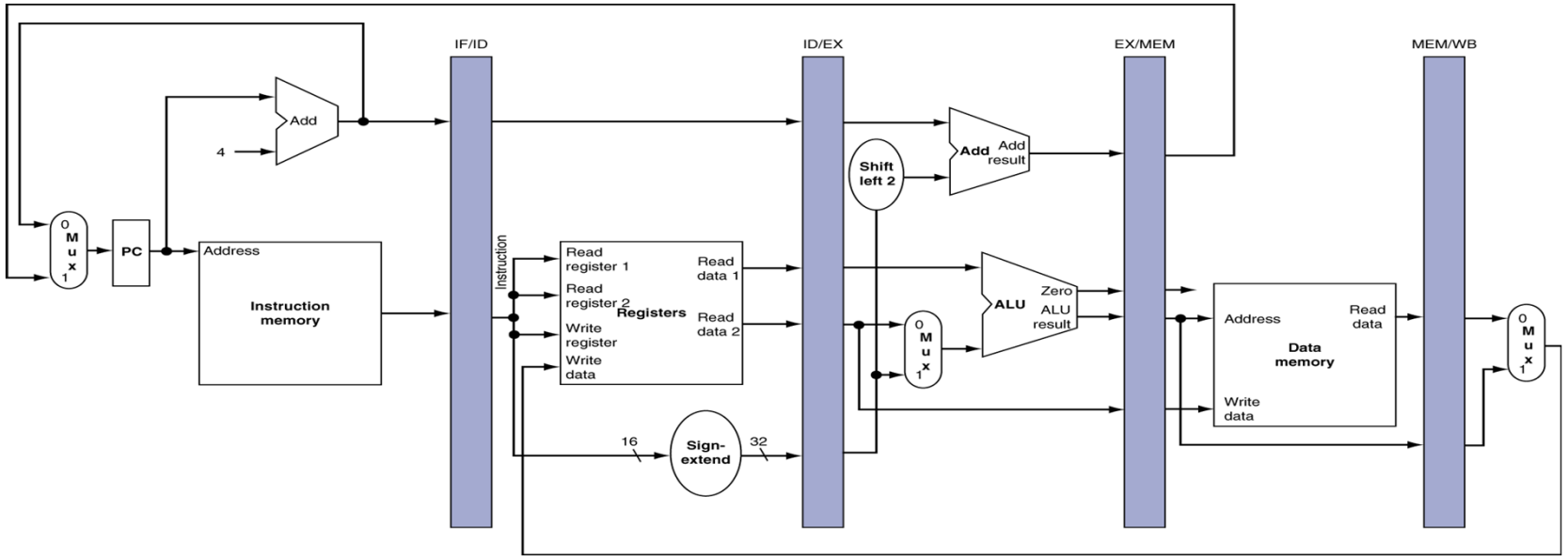
Control Hazards: stall



Branch outcome is not taken

Instruction	1	2	3	4	5	6	7	8
`beq $2, $4, L1`	IF	ID						
`add $3, $1, $2`		IF						
`L1:sw $4, 12($2)`								

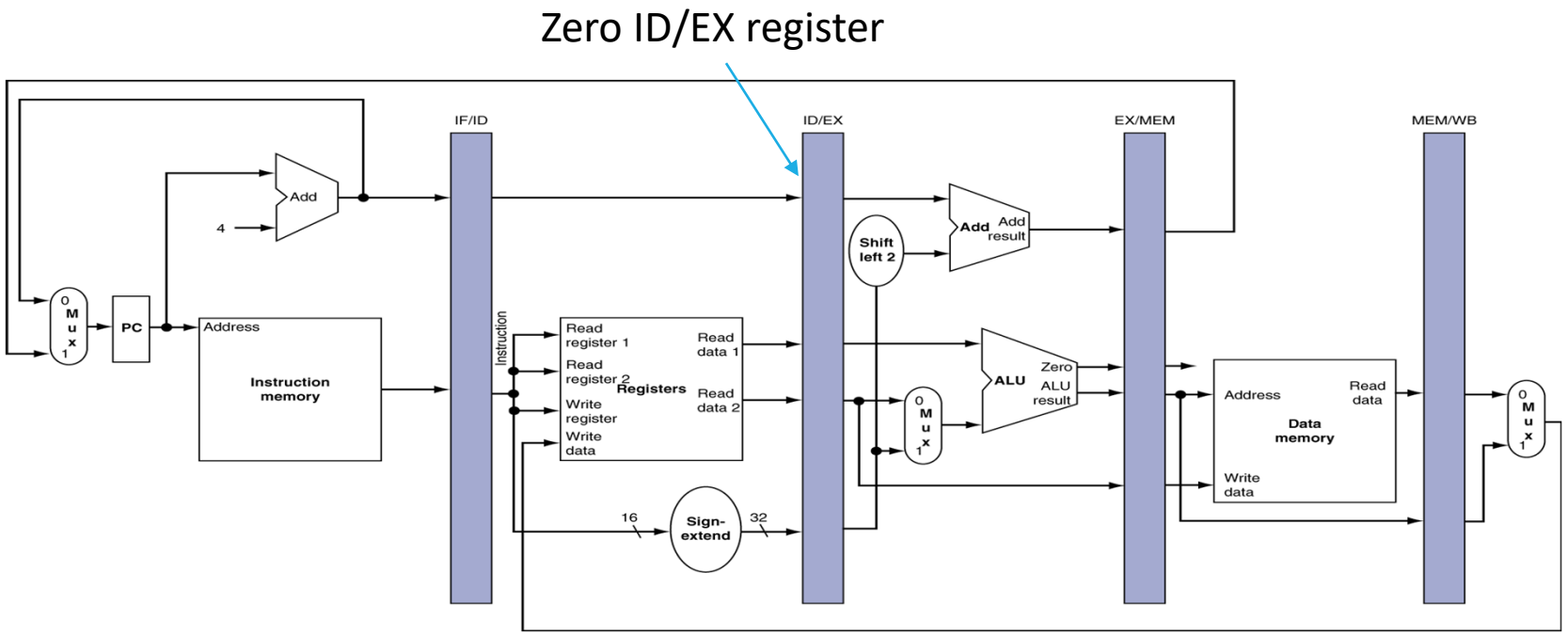
Control Hazards: static prediction



Branch outcome is not taken

Instruction	1	2	3	4	5	6	7	8
`beq $2, $4, L1`	IF	ID	EX	M	WB			
`add $3, $1, $2`		IF	ID	EX	M	WB		
`L1:sw $4, 12($2)`			IF	ID	EX	M	WB	

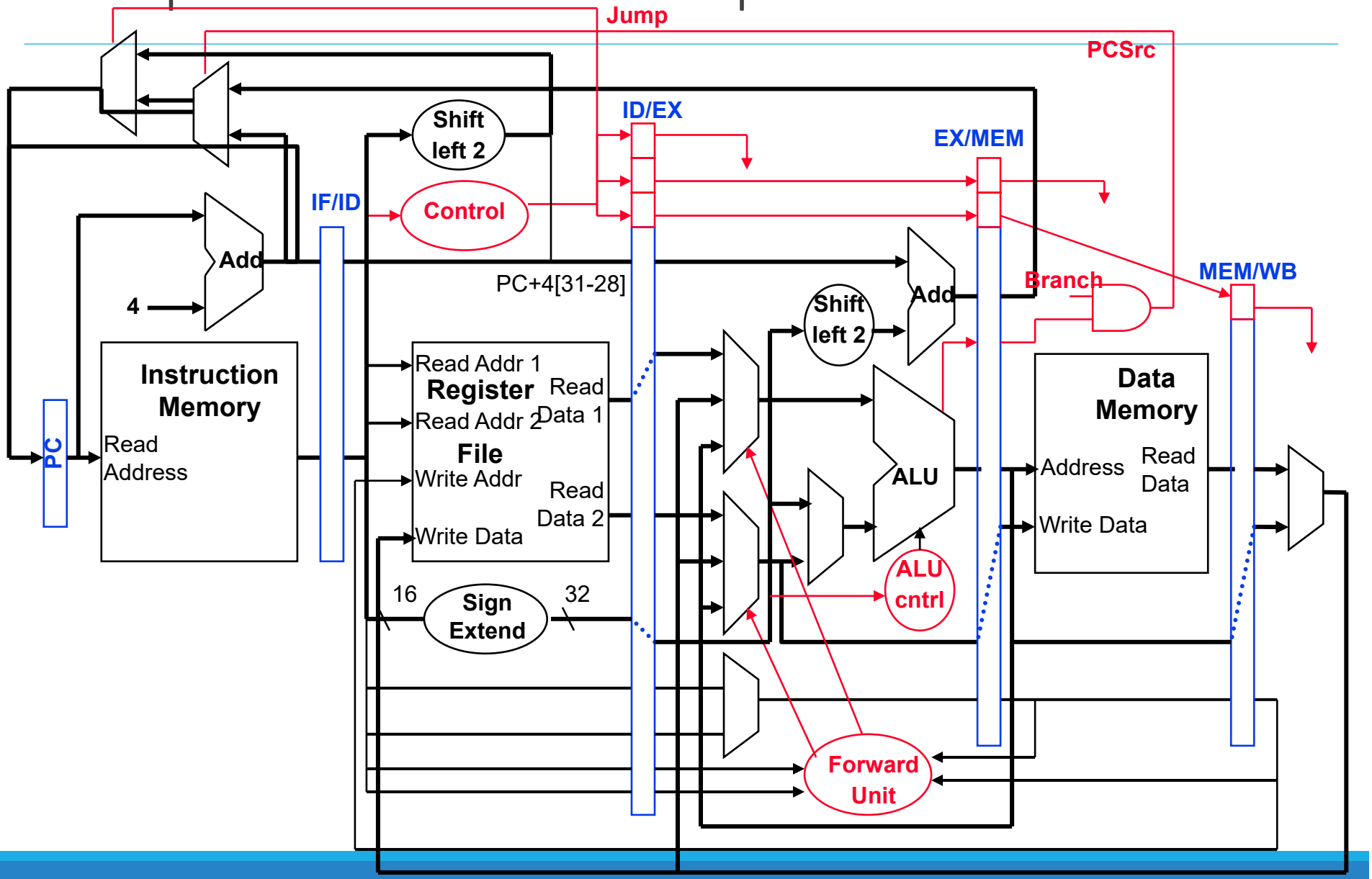
Control Hazards: static prediction



Branch outcome is taken

Instruction	1	2	3	4	5	6	7	8
`beq $2, $4, L1`	IF	ID	EX					
`add $3, $1, $2`		IF	ID					
`L1:sw $4, 12($2)`			IF					

Datapath Branch and Jump Hardware



Control Hazards: earlier outcome

Add hardware to compute the branch target address and to evaluate the branch decision to the ID stage

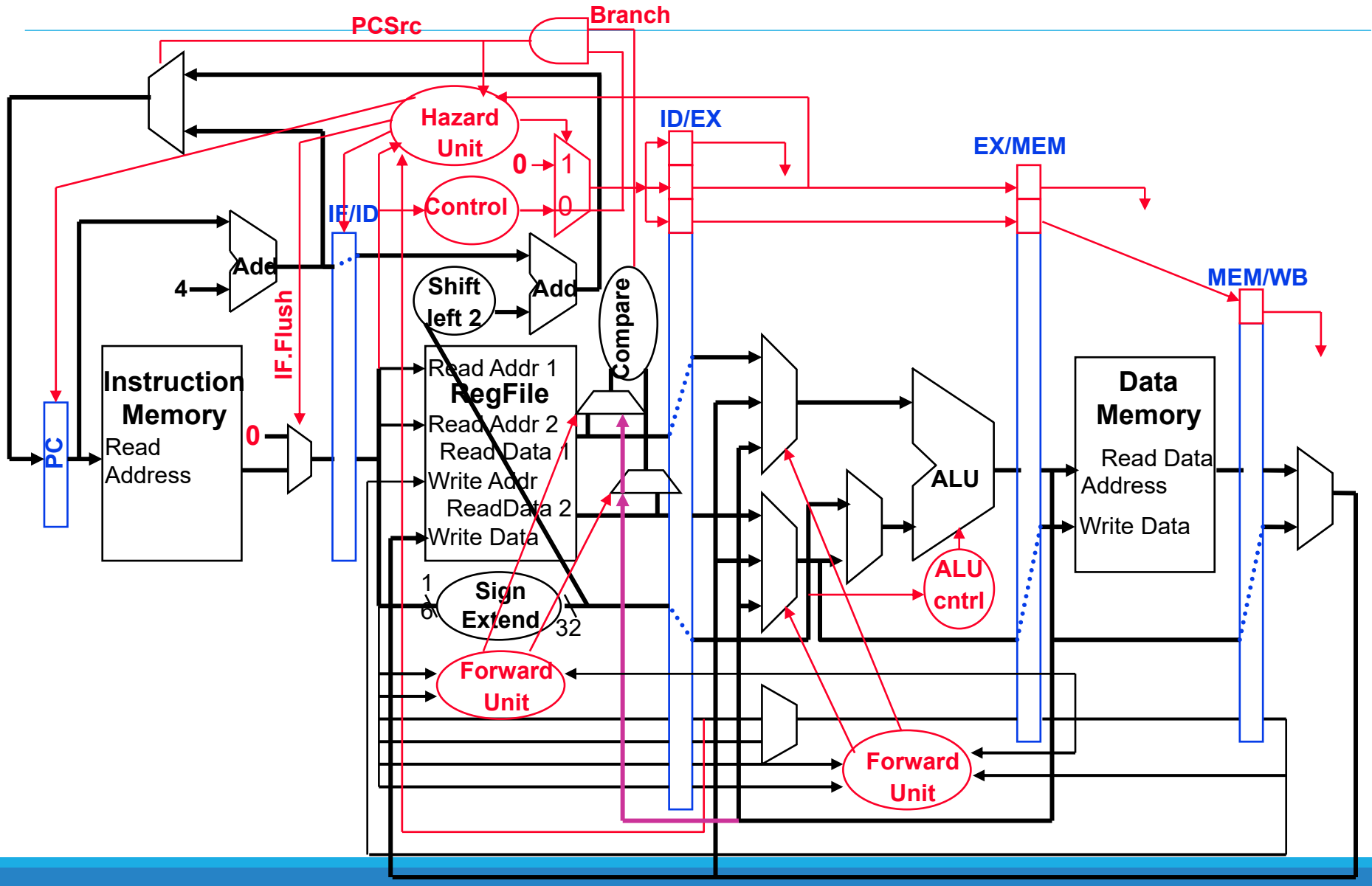
- Reduces the number of stall (flush) cycles to one (like with jumps)

Need to forward from the EX/MEM pipeline stage to the ID comparison hardware

```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
    ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
    ForwardD = 1
```

Forwards the result from the second previous instr. to either input of the compare

Supporting ID Stage Branches



Control Hazards: dynamic prediction

Predict branches at run-time

- It predicts **IF** branch is taken or not, and **WHERE** it is taken to

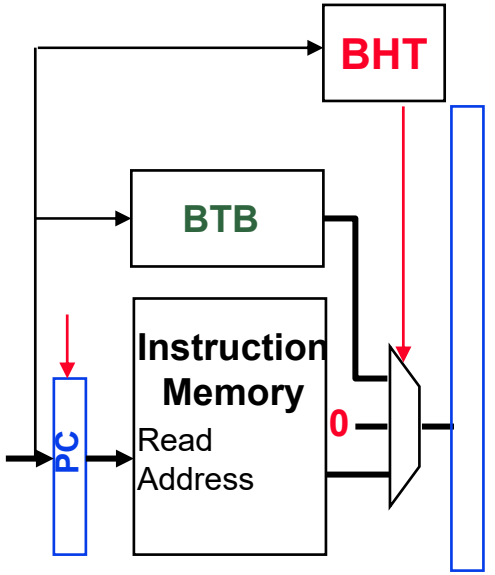
It needs two elements:

- A Branch Prediction Buffer, also known as Branch History Table (**BHT**)
- A Branch Target Buffer (**BTB**)

Both are limited in size

- Addressed with lower bits of PC

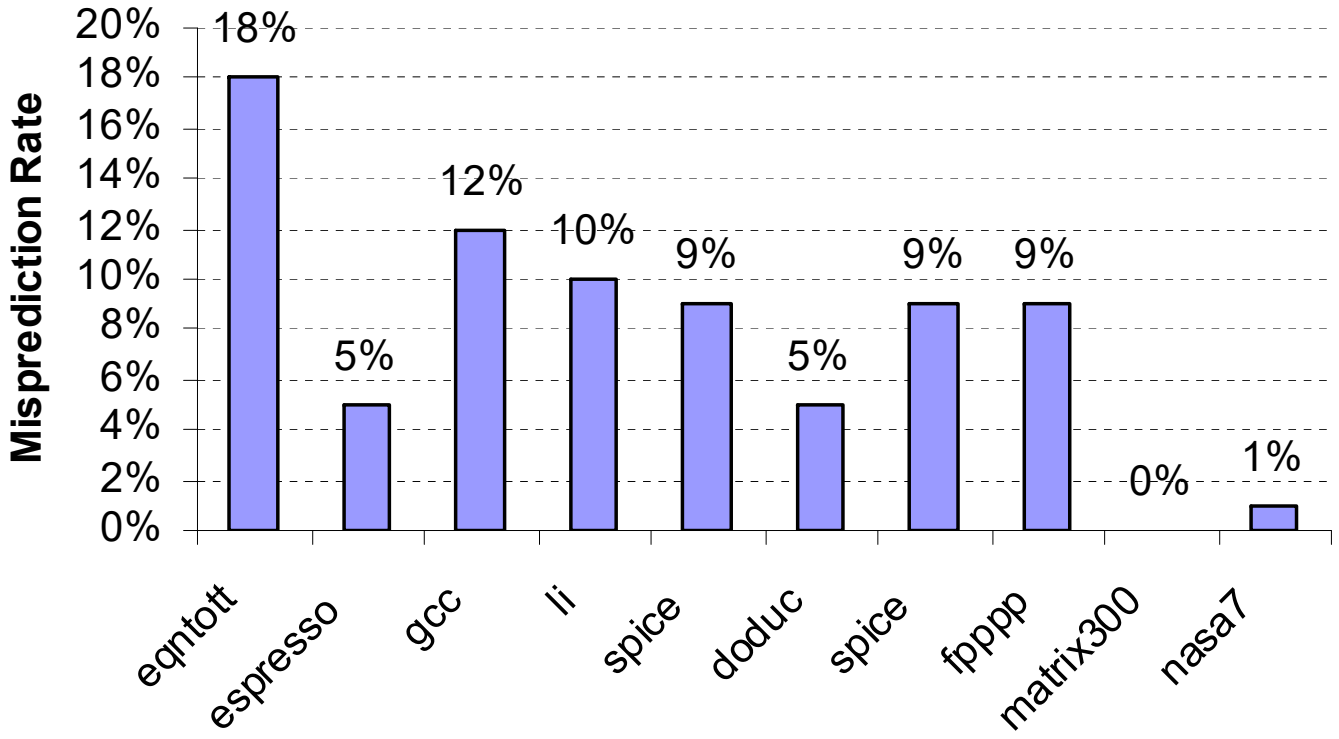
Update both if wrong prediction



Control Hazards: dynamic prediction

1-bit predictor

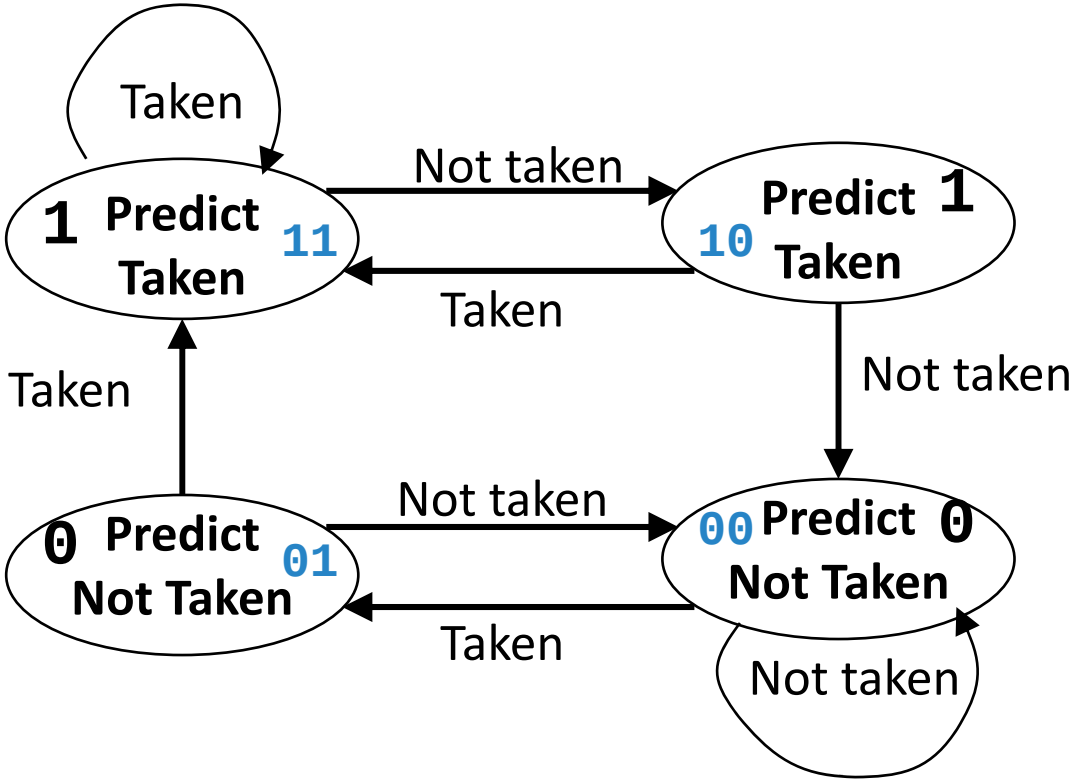
- Each entry in the BHT stores the last outcome of a branch at that address
- Accuracy of a 1-bit predictor with 4096 entries



Control Hazards: dynamic prediction

2-bits predictor

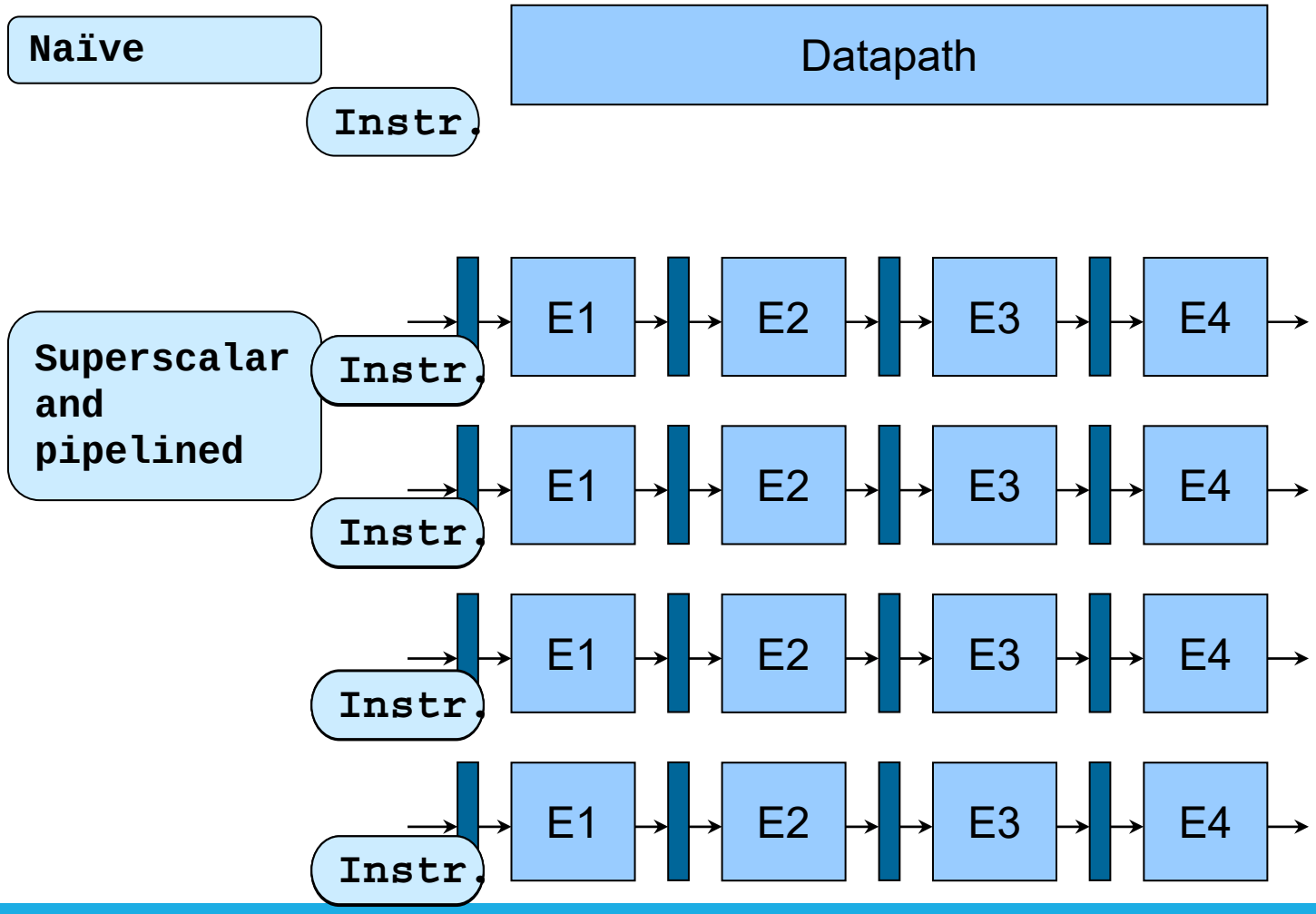
- Each entry in the BHT stores the **two** last outcomes of a branch at that address



Modern processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Superscalar pipelined processor



Pipeline Summary

The BIG Picture

Pipelining improves performance by increasing instruction throughput

- Executes multiple instructions in parallel
- Each instruction has the same latency

Subject to hazards

- Structure, data, control

Instruction set design affects complexity of pipeline implementation