H4
The processor

ELECTA01 Computer architecture

ELECTA01
Hogeschool Rotterdam
Opleiding Elektrotechniek
Minor Embedded Systems
J.W. Peltenburg / J.Z.M. Broeders (brojz@hr.nl)

Planning CTA01

- Week 1: Introduction, Performance & Parallelism
- Week 2: Instructions for arithmetic and memory
- Week 3: Instructions for decisions and procedures
- Week 4: Computer arithmetic
- **Week 5: Single cycle LEGv8, intro pipelining**
- Week 6: Pipelined LEGv8, hazards and forwarding
- Week 7: Memory architecture
- Week 8: Guest lecture (multicore and parallelism)

Chapter 4

The Processor

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two LEGv8 implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `LDUR`, `STUR`
  - Arithmetic/logical: `ADD`, `SUB`, `AND`, `ORR`
  - Control transfer: `CBZ`

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction type
  - Use ALU to calculate
    - Arithmetic or logic result for R-type (ADD, SUB, AND, ORR)
    - Memory address for D-type instructions (LDUR, STUR)
    - Register equals zero for CB-type (CBZ)

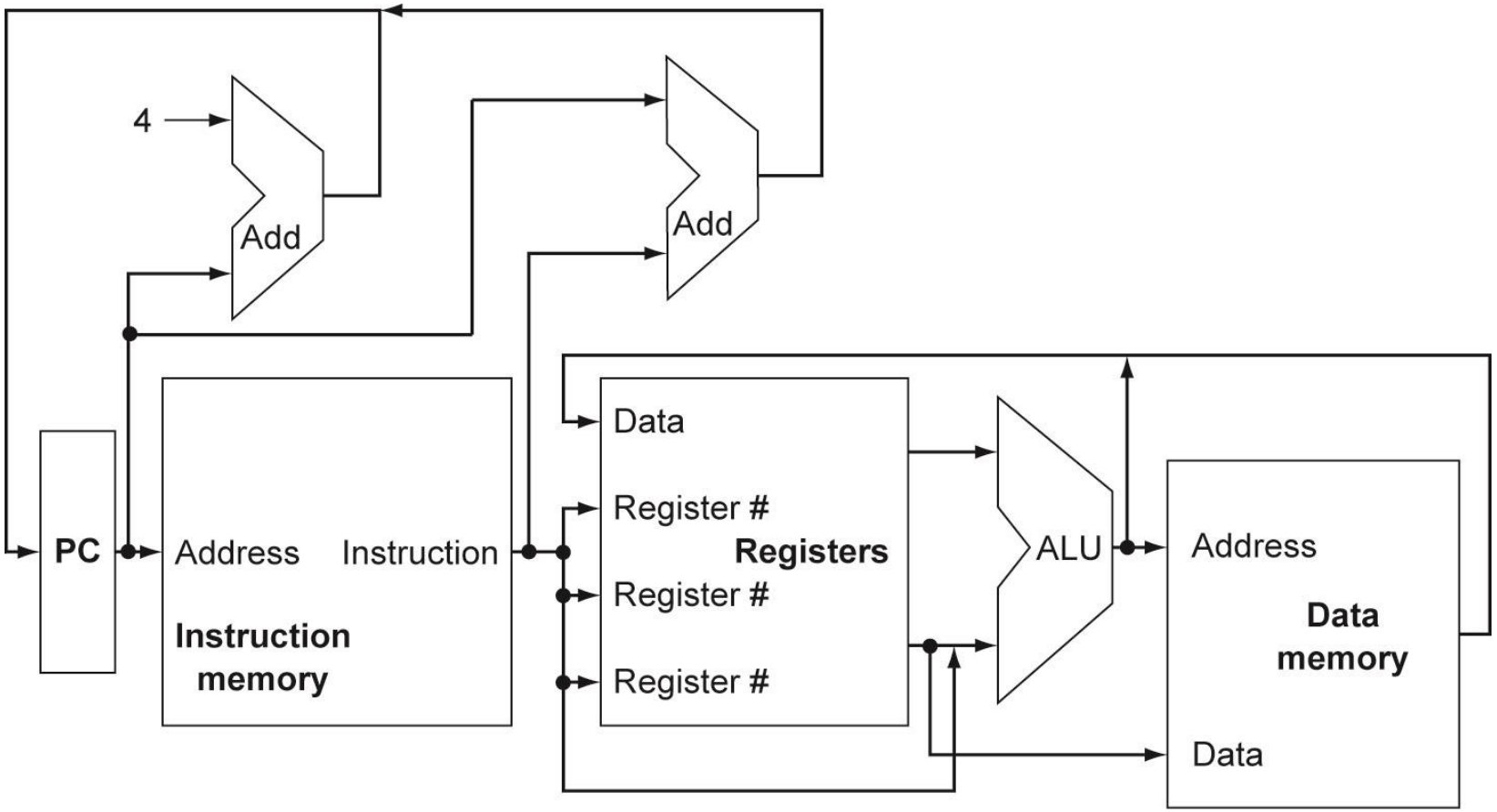
  In parallel

  - For CB-type (CBZ) calculate branch target address
  - Access data memory for D-type (LDUR, STUR)
  - Write to register file for R-type (ADD, SUB, AND, ORR) and load (LDUR)

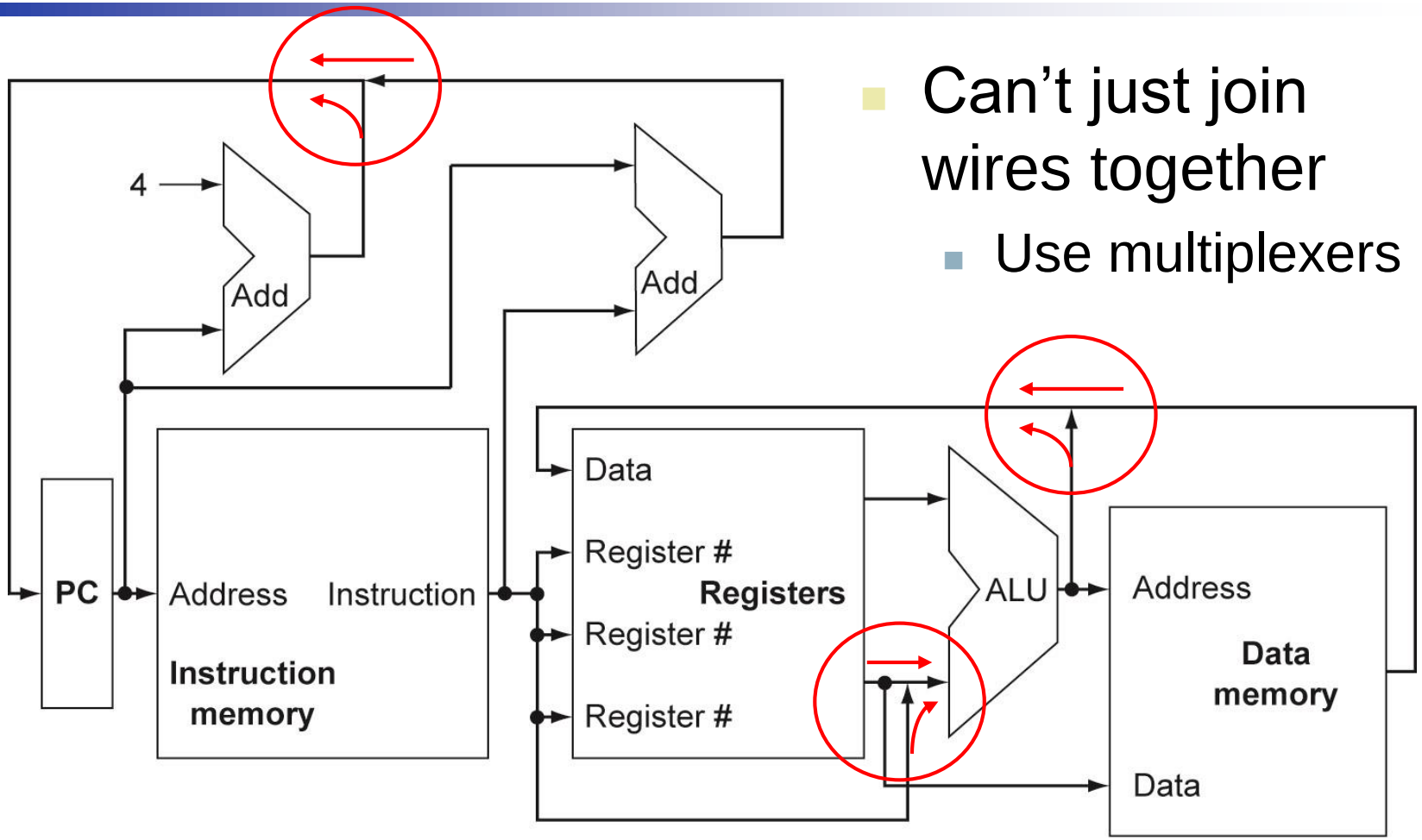
  In parallel

- PC ← PC + 4 or PC ← Branch target address

CPU Overview

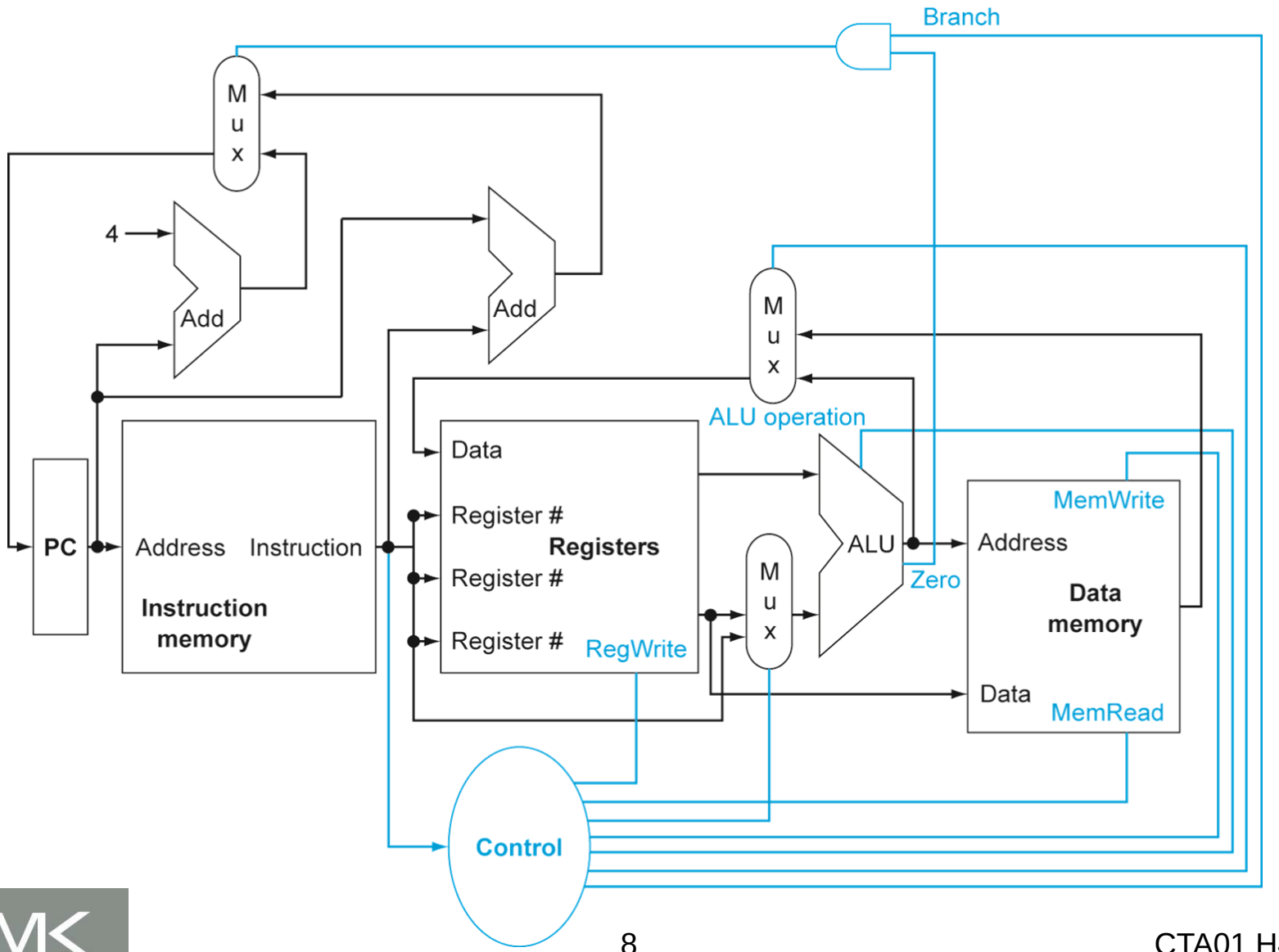


Multiplexers

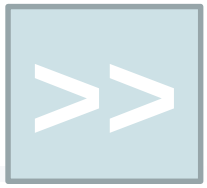


- Can't just join wires together
  - Use multiplexers

Control


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

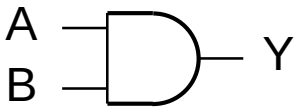
Logic Design Basics



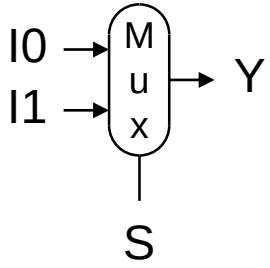
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

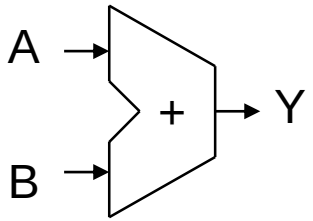
- AND-gate
  - Y = A & B



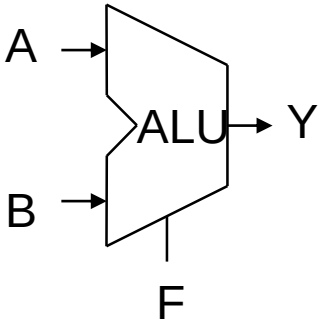
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

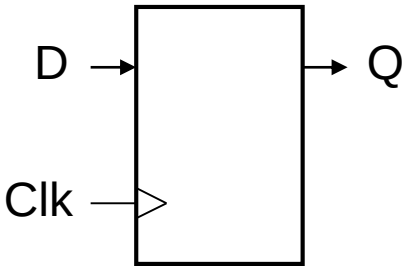


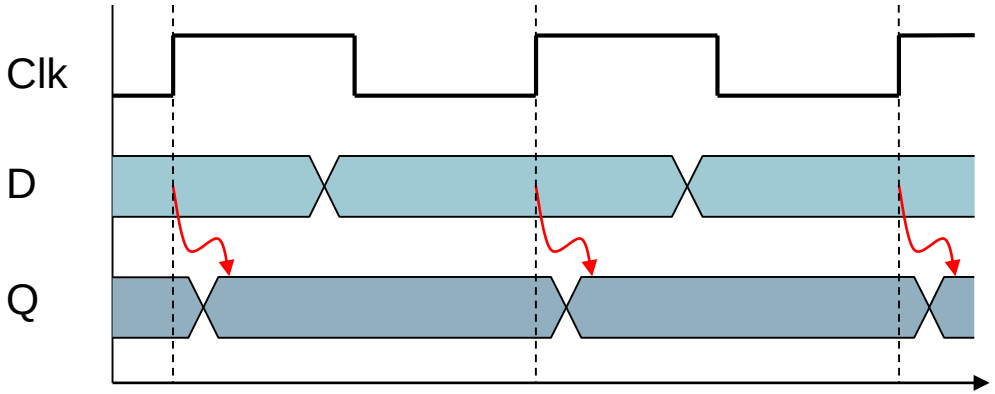
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

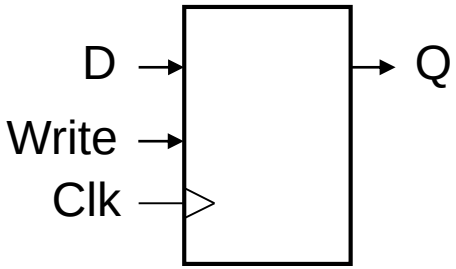
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

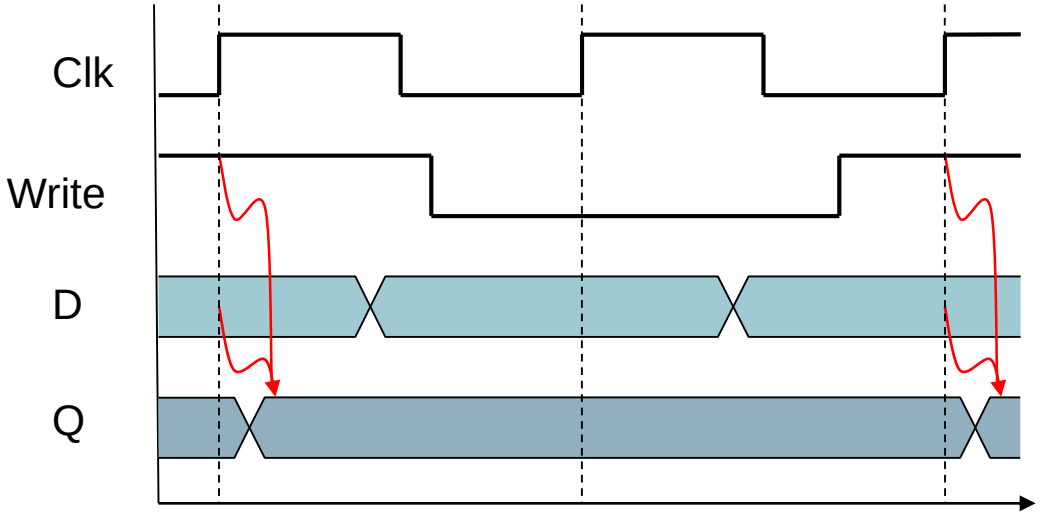




Sequential Elements

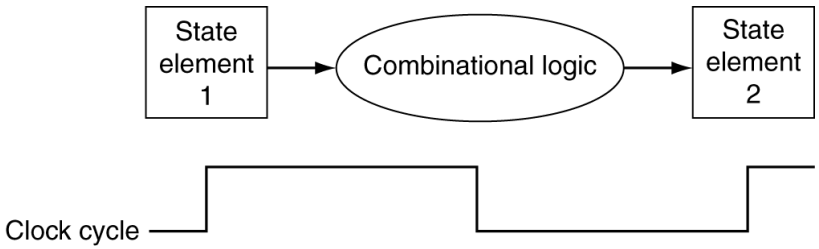
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

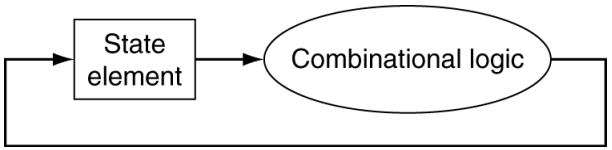




Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

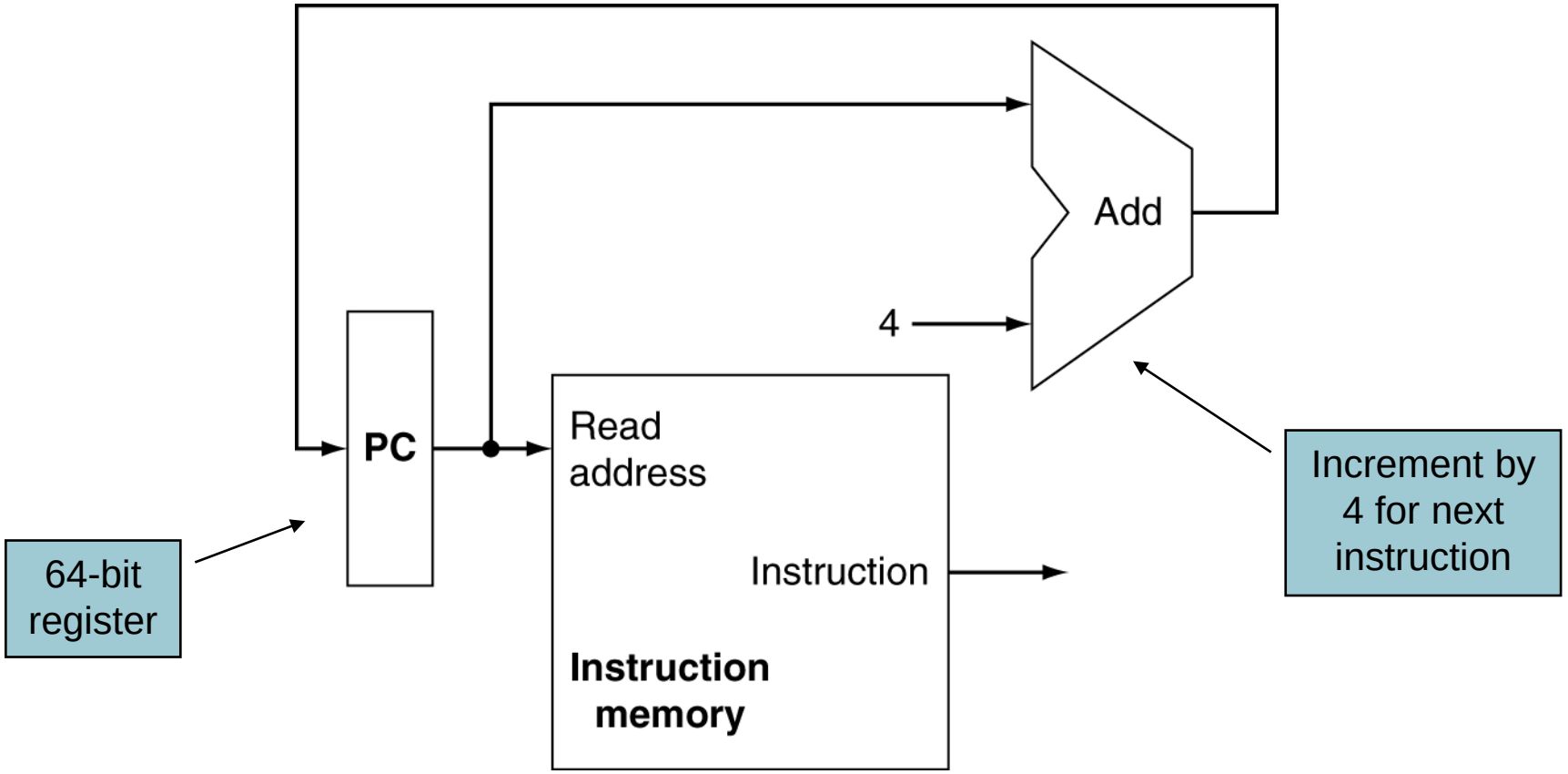




Building a Datapath

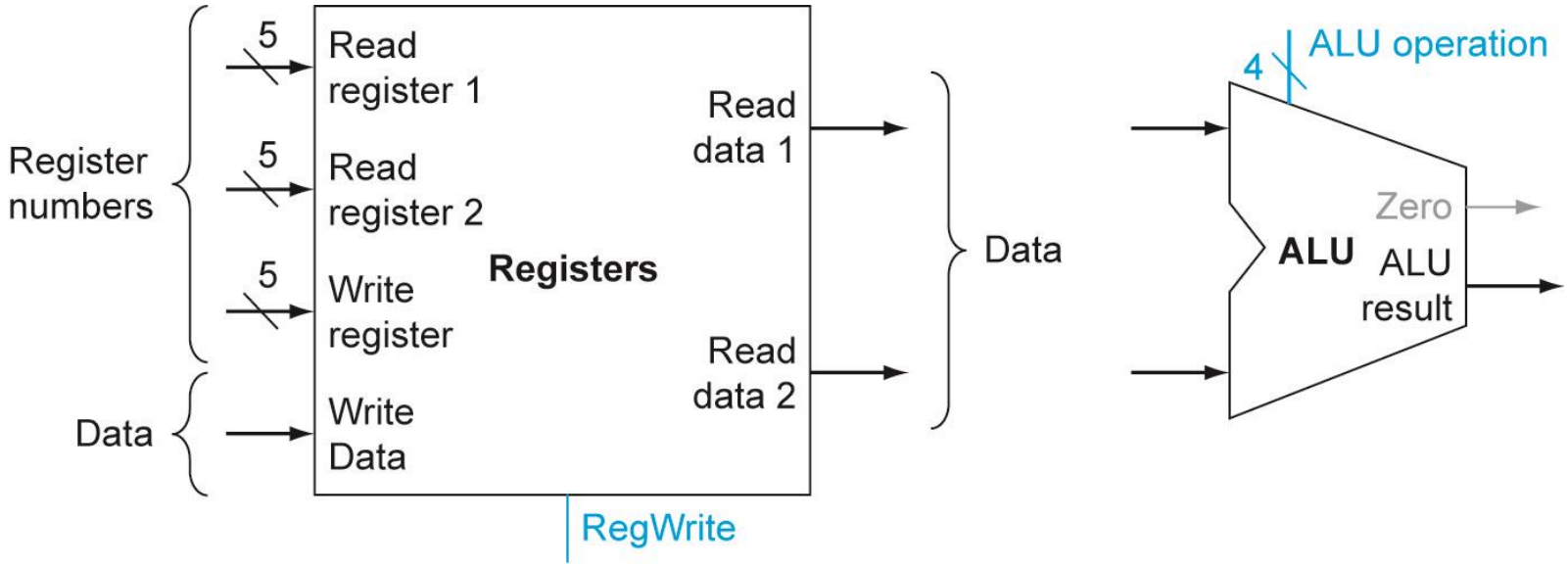
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a LEGv8 datapath incrementally
  - Refining the overview design

Instruction Fetch



R-type Instructions ADD, SUB, AND, ORR

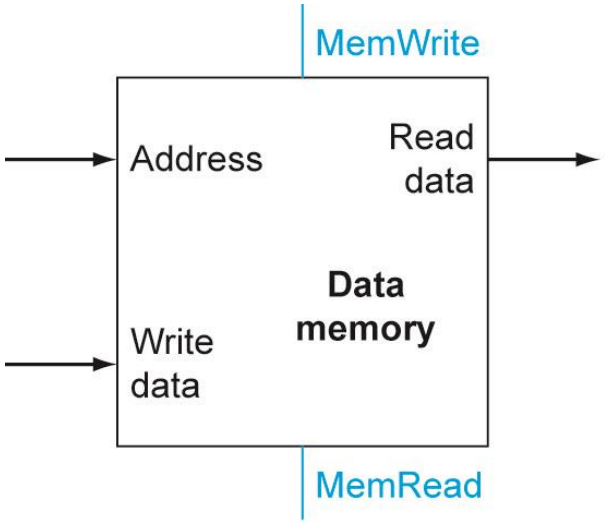
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



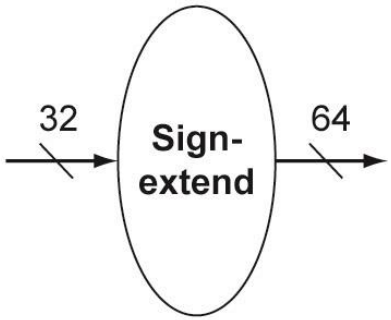
Multiport register file:
2 read ports, 1 write port

D-type Instructions LDUR, STUR

- Read register operands
- Calculate address using 9-bit offset
  - Use ALU, but sign-extend offset
- LDUR: Read memory and update register
- STUR: Write register value to memory



a. Data memory unit



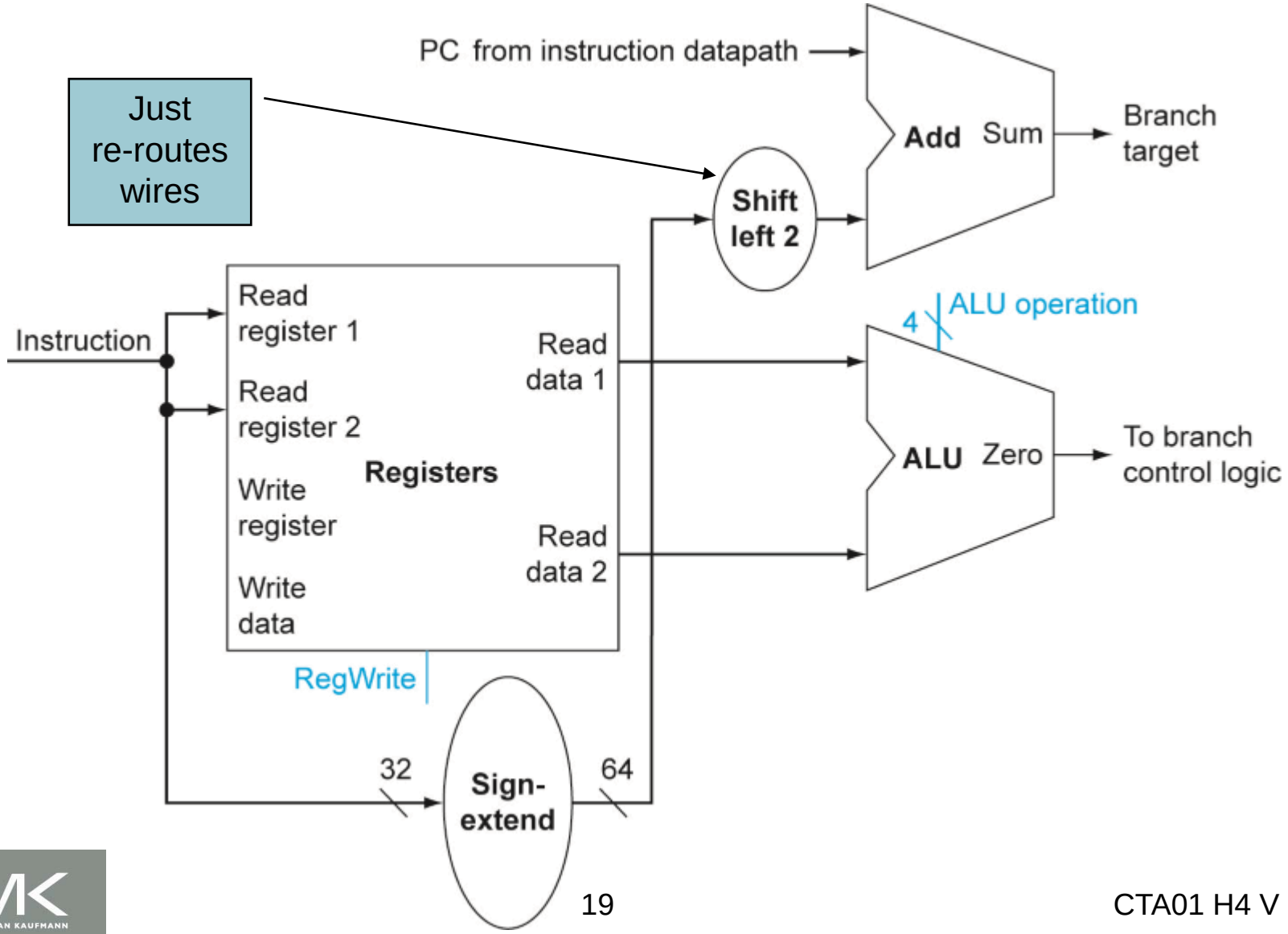
b. Sign extension unit

CB-type Instruction CBZ

- Read register operand
- Compare operand to zero
  - Use ALU pass input and check Zero output
- Calculate target address
  - Sign-extend 19-bit displacement field
  - Shift left 2 places (word displacement)
  - Add to PC

Branch Instructions

Corrected version of Figure 4.9 page 268.

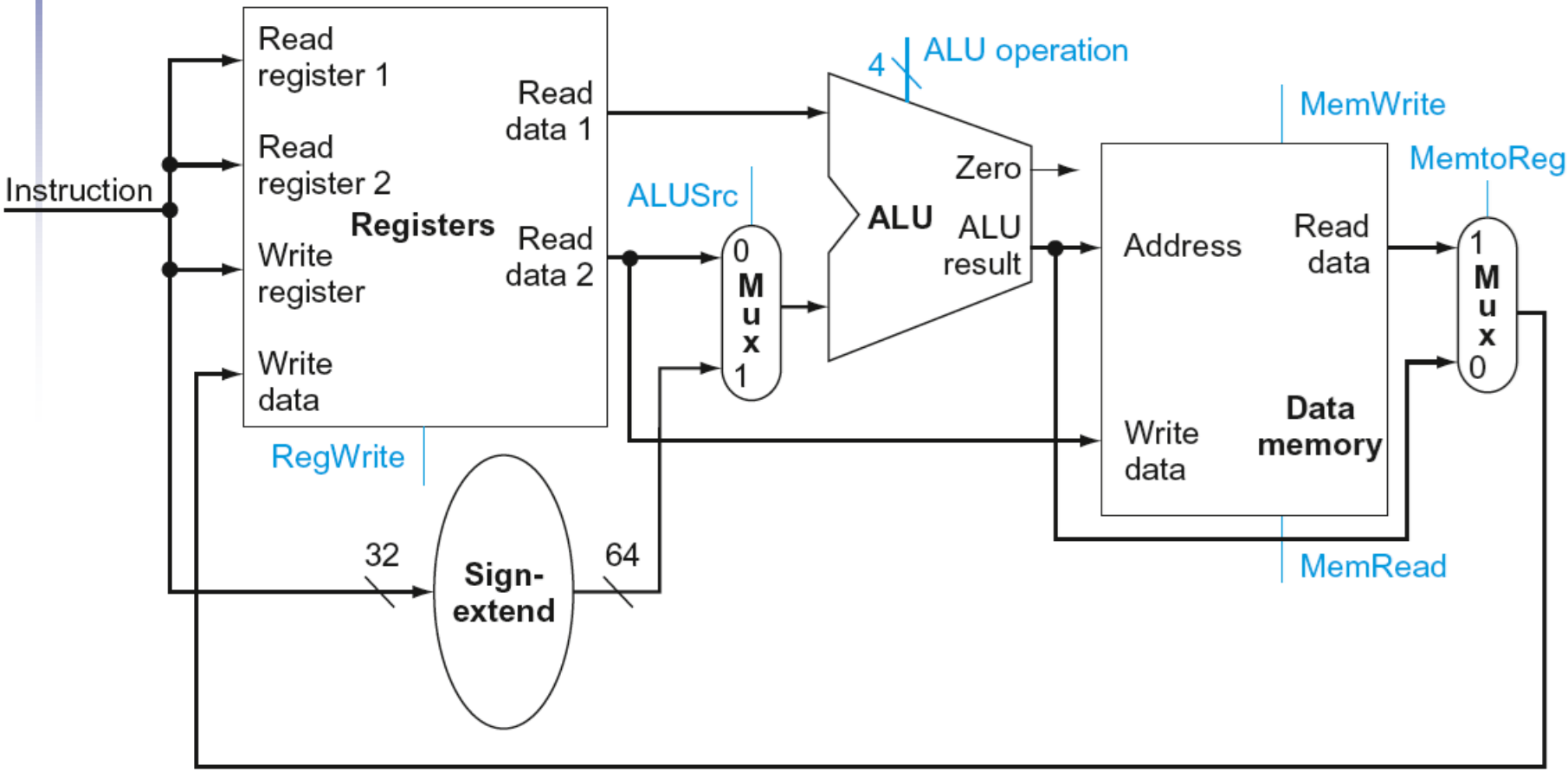


Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

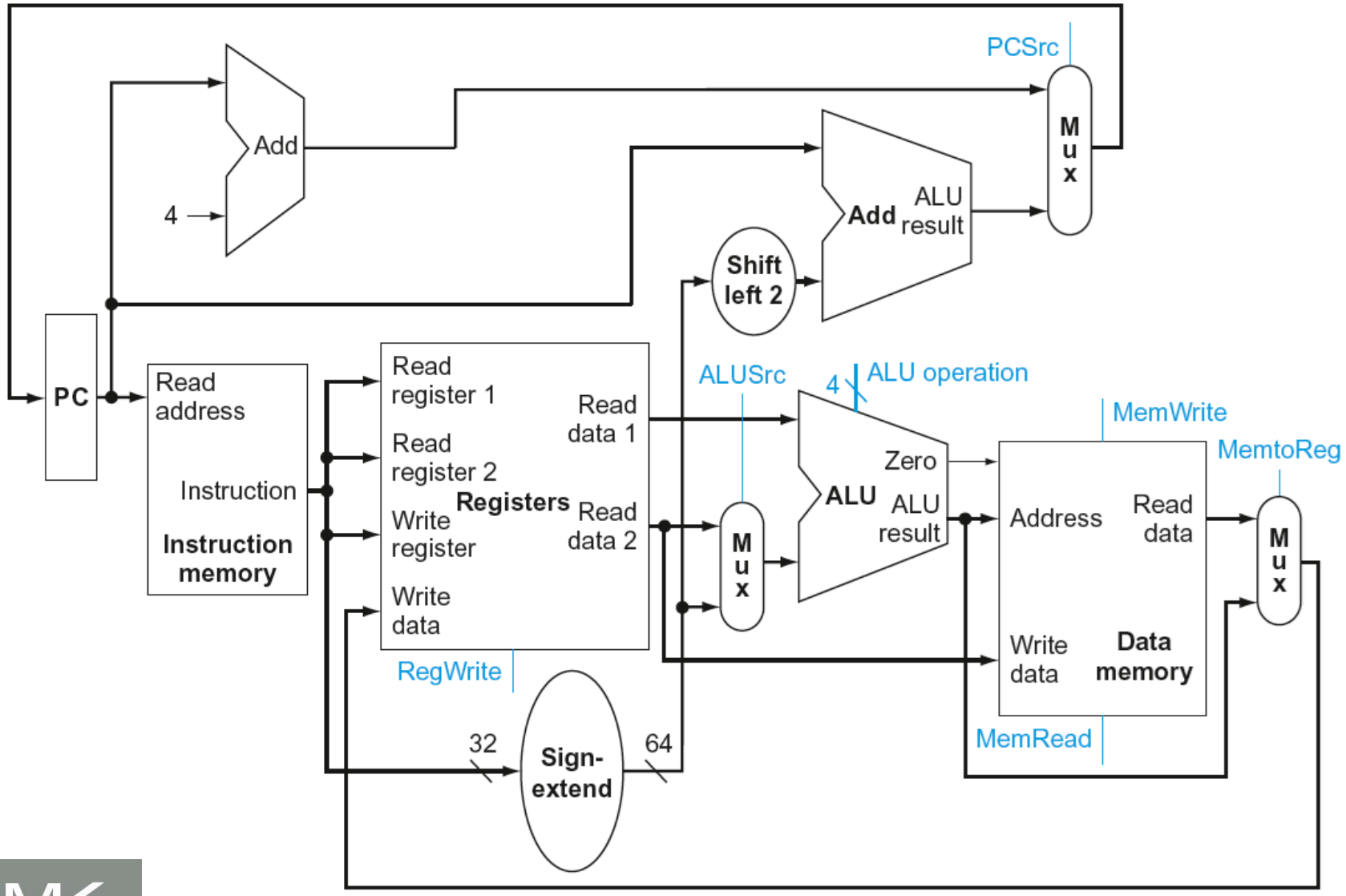
R-Type/D-Type Datapath

ADD, SUB, AND, ORR, LDUR, STUR



Full Datapath

ADD, SUB, AND, ORR, LDUR, STUR, CBZ



ALU Control

See ALU design Figure A.5.12 page A-35

- ALU used for
  - D-type LDUR, STUR: Function = add
  - CB-type CBZ: Function = pass
  - R-type ADD, SUB, AND, ORR: Function depends on opcode

ALU control	F = Function
0000	AND
0001	OR
0010	add
0110	subtract
XX11	pass input b

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	XX11
R-type ADD, SUB, AND, ORR	10	add	10001011000	add	0010
		subtract	11001011000	subtract	0110
		AND	10001010000	AND	0000
		OR	10101010000	OR	0001

The Main Control Unit

- Control signals derived from instruction

Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

a. R-type instruction

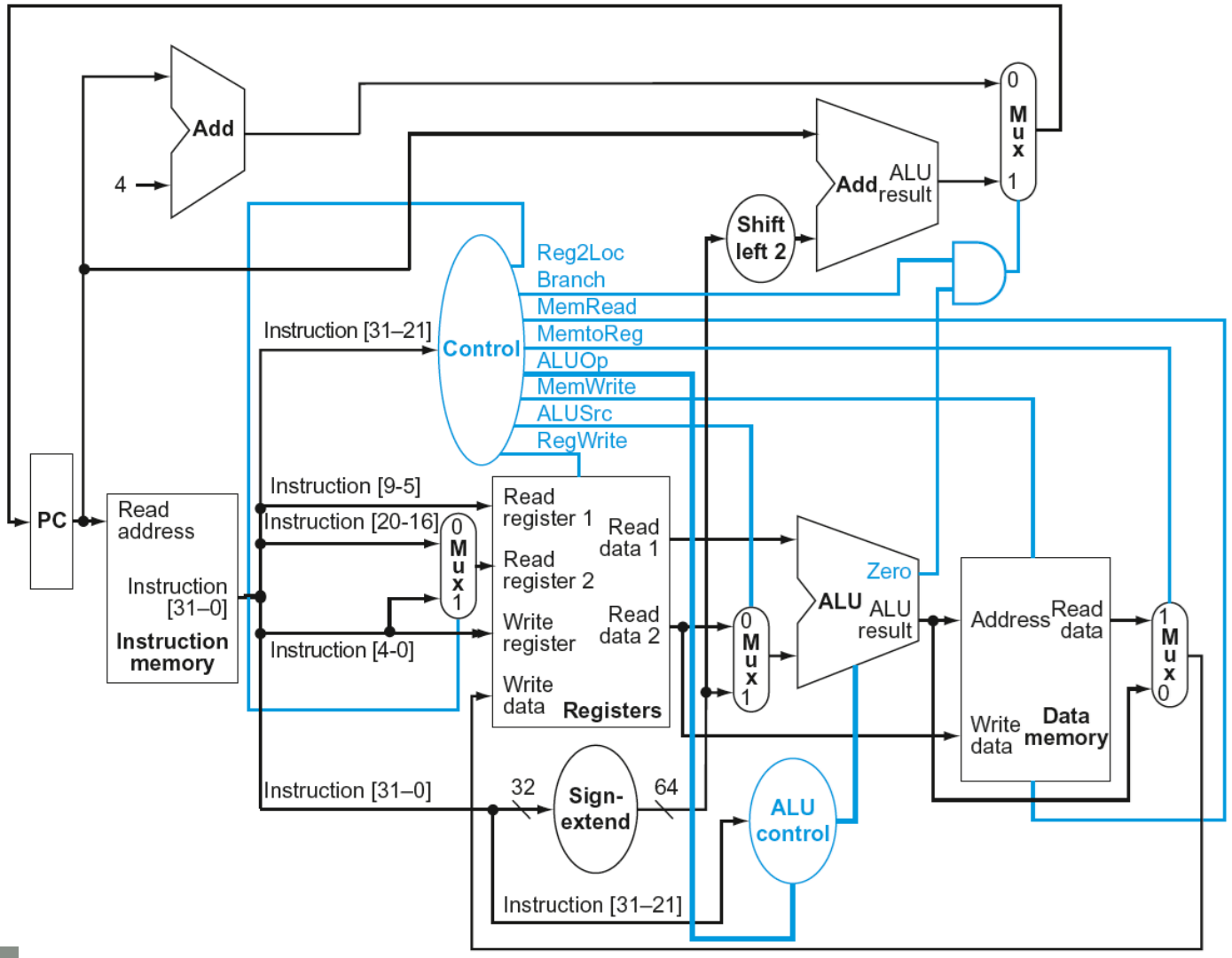
Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

Field	180	address	Rt
Bit positions	31:26	23:5	4:0

c. Conditional branch instruction

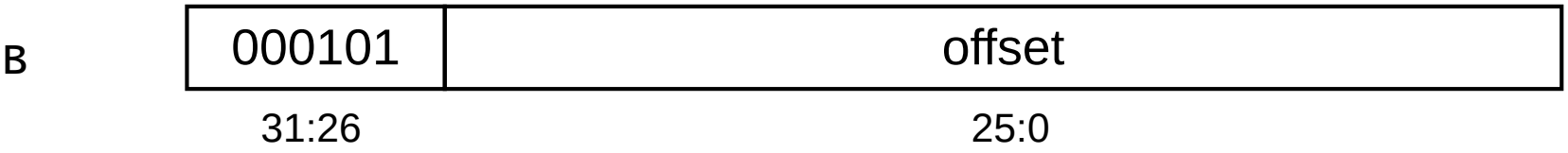
Datapath With Control



Control signals

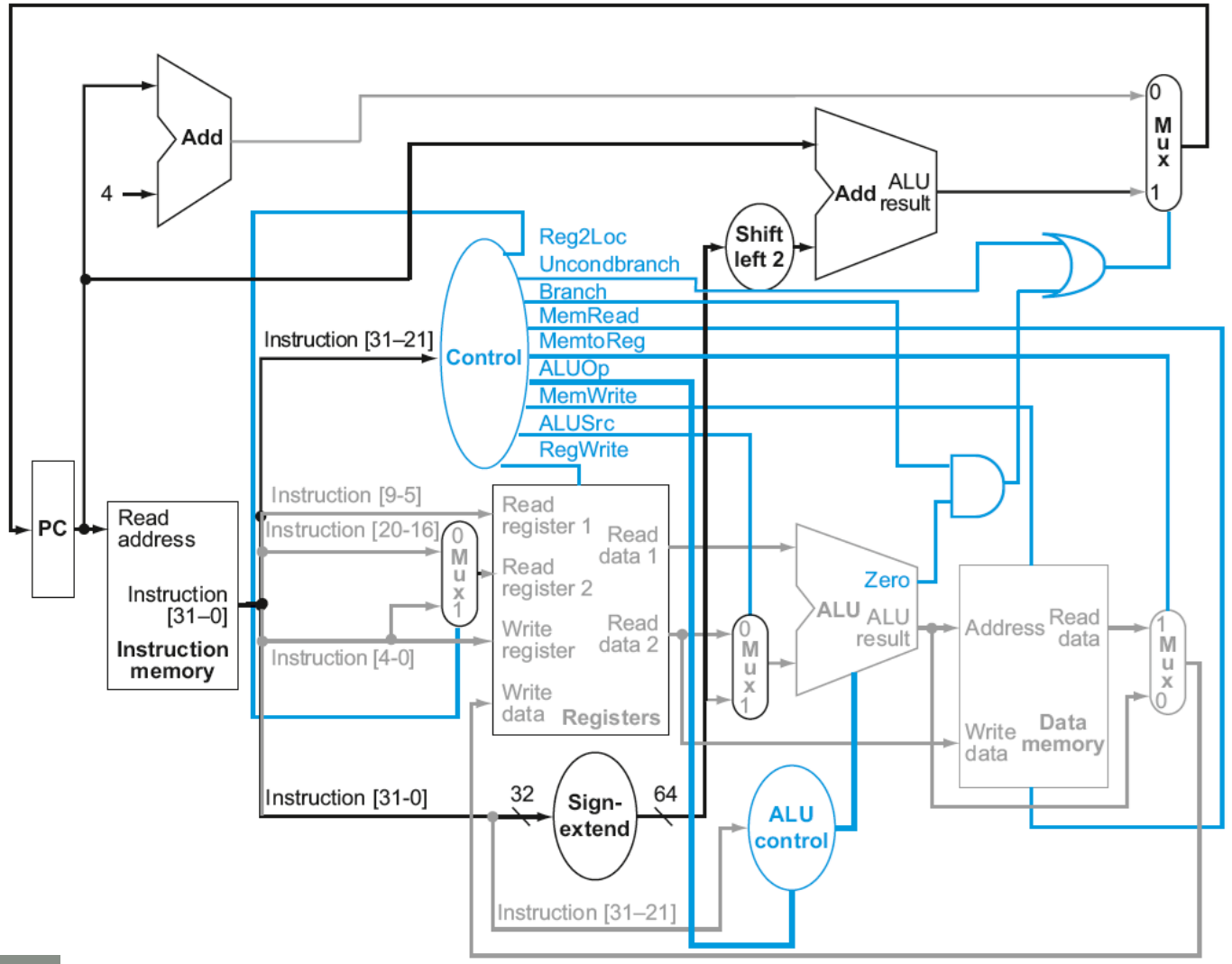
Signal name	Effect when deasserted	Effect when asserted
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 20:16).	The register number for Read register 2 comes from the Rt field (bits 4:0).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Implementing B-type (B)



- Branch uses word offset
- PC ← PC + (sign extended offset) x 4
- Need an extra control signal decoded from opcode

Datapath With B Added

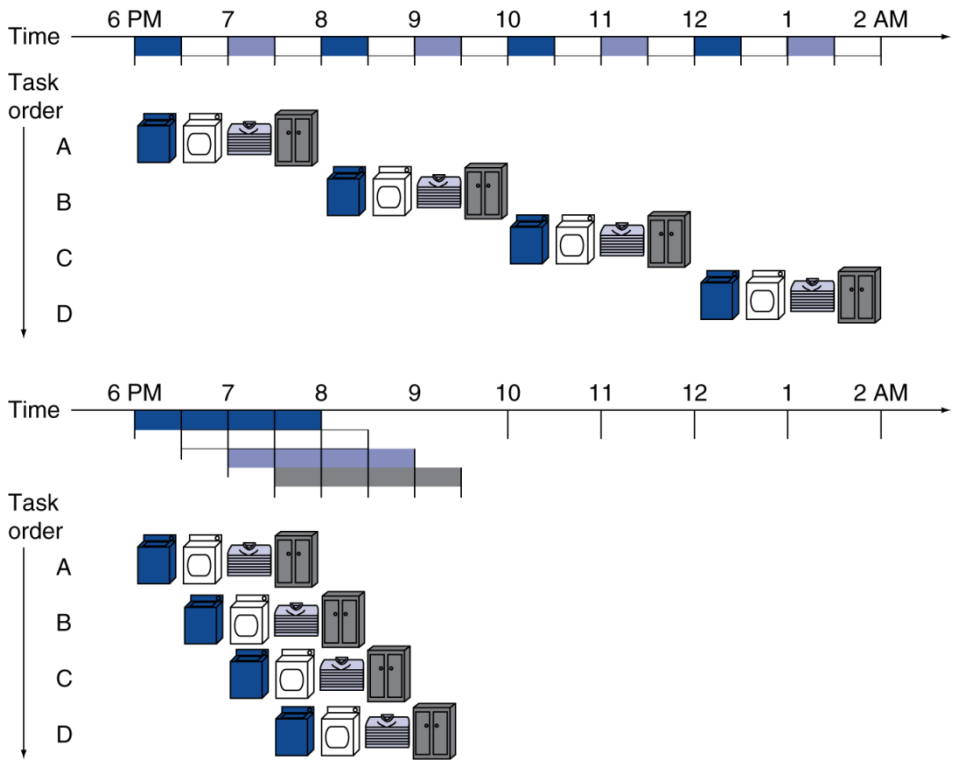


Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



Speedup?

- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

LEGv8 Pipeline

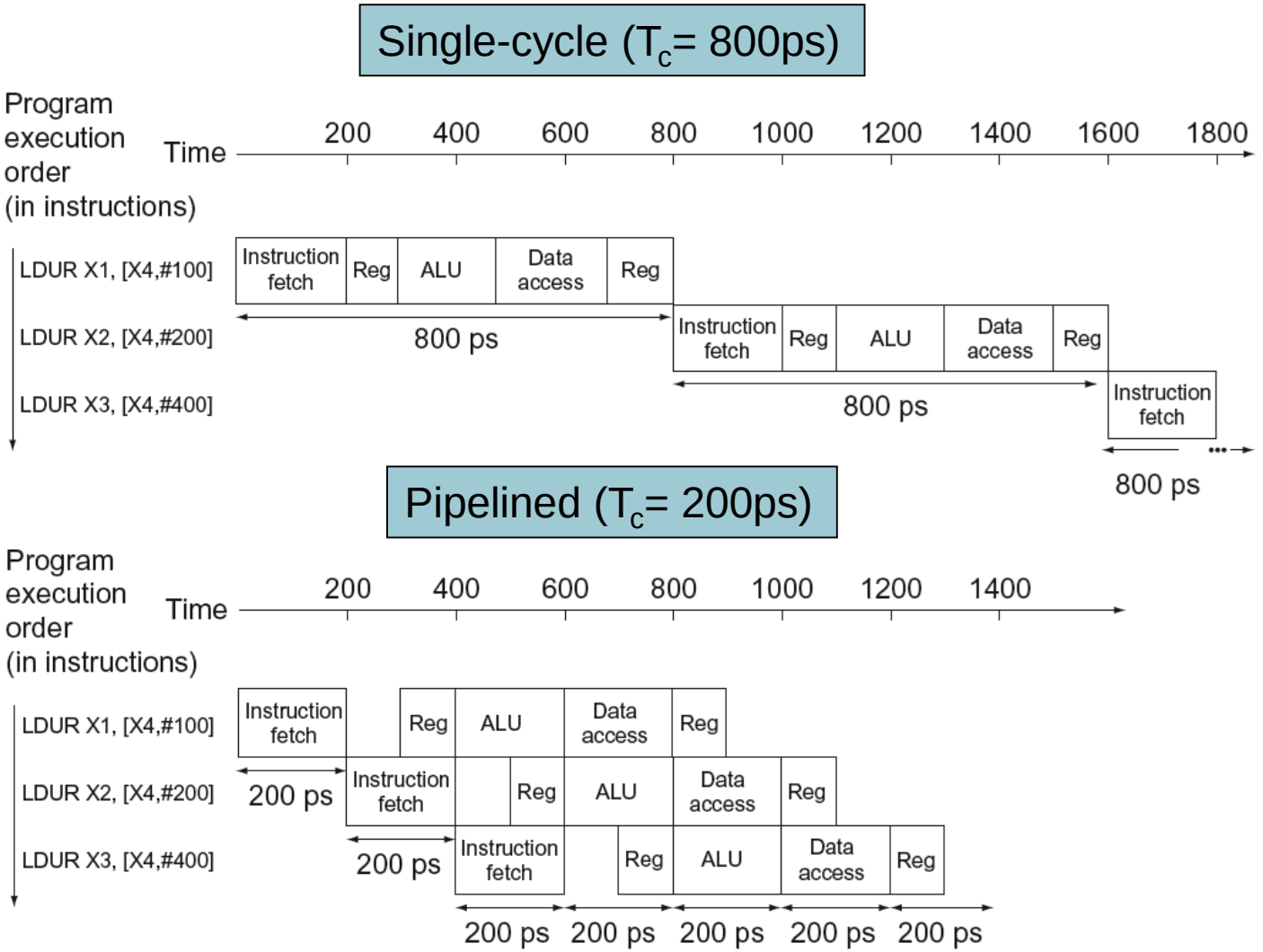
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

- Assume time for stages is
  - 100 ps for register read or write
  - 200 ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
LDUR	200ps	100 ps	200ps	200ps	100 ps	800ps
STUR	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
CBZ	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- LEGv8 ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - 64-bit Memory access takes only one cycle

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

Structure Hazards

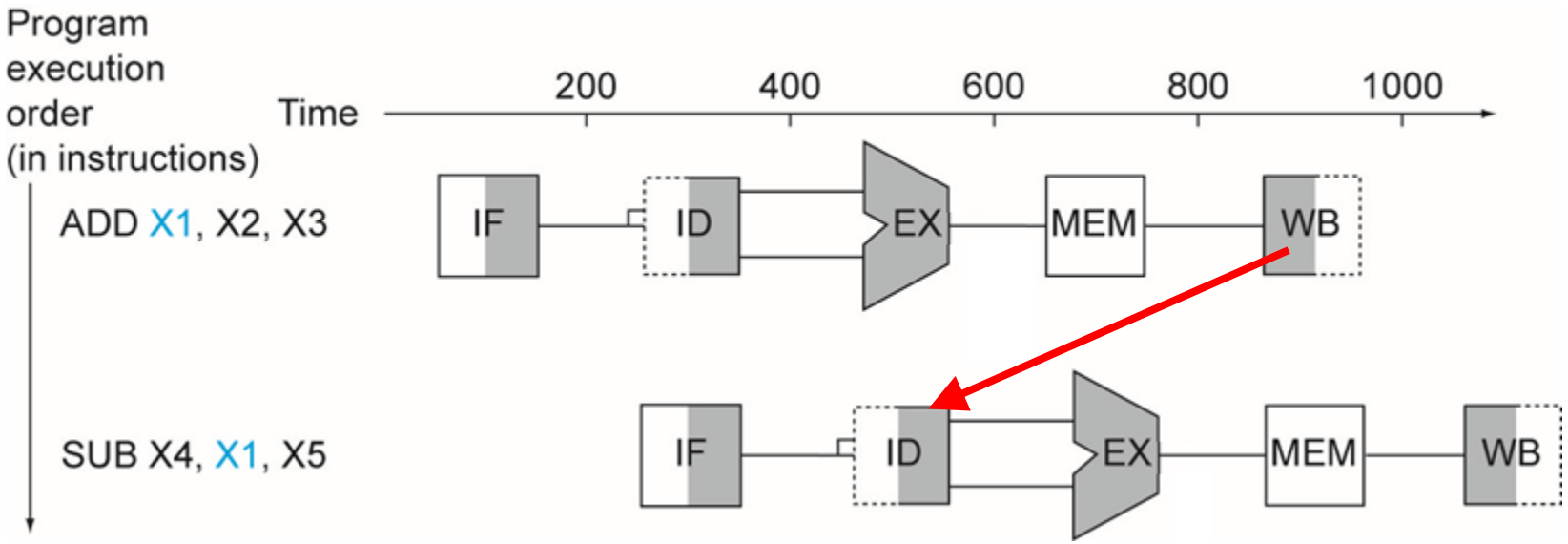
- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Structure Hazards

- Conflict for use of a resource
- In LEGv8 the registers are used in stages ID (read) and WB (write)
- Assumed time for stages is
  - 100 ps for register read or write
  - 200 ps for other stages
- Hence, write and read can be combined in one pipeline slot
  - write in first 100 ps, read in last 100 ps

Data Hazards

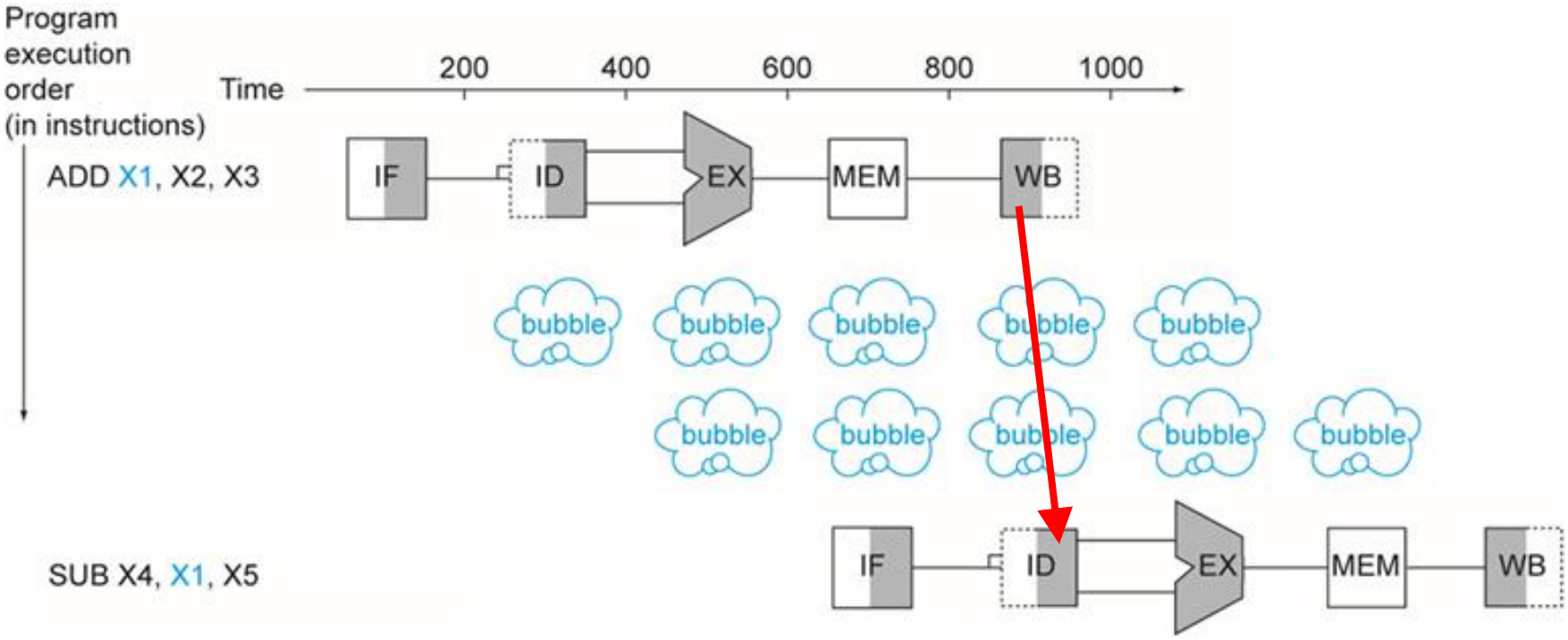
- An instruction depends on completion of data access by a previous instruction
  - ADD X1, X2, X3
    SUB X4, X1, X5



Data can not move backward in time!

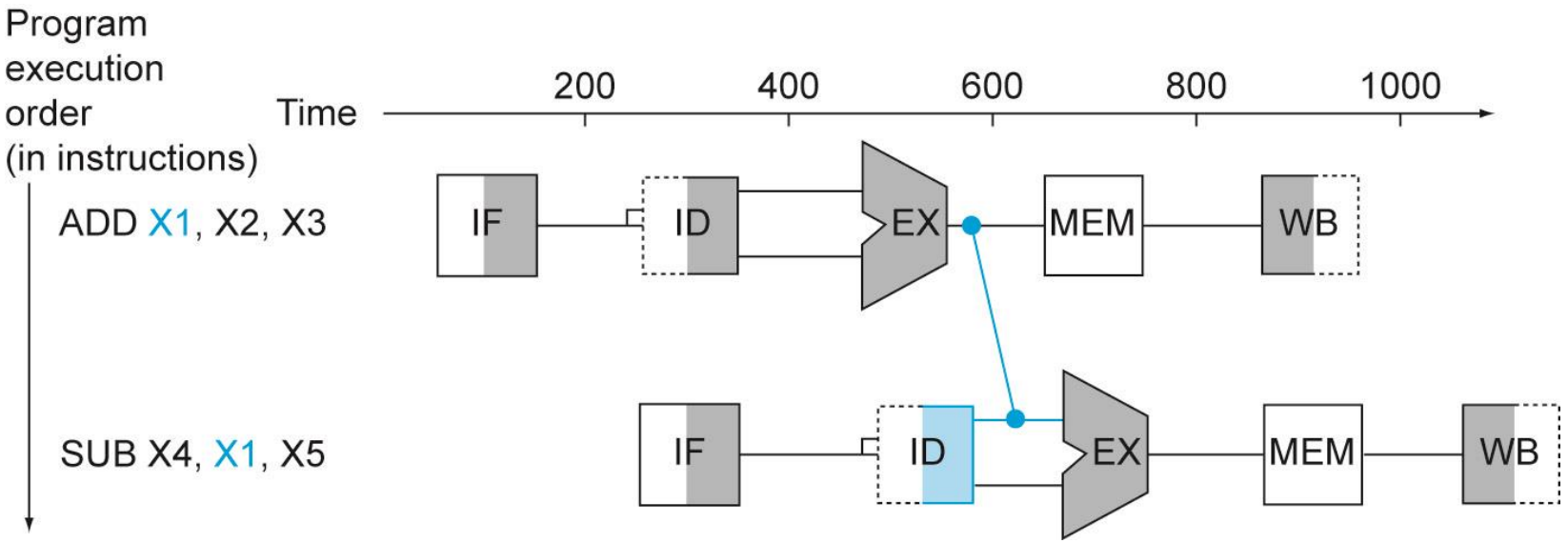
Stalling (aka insert bubbles)

- wait for the data to be stored in a register
  - Requires extra control logic



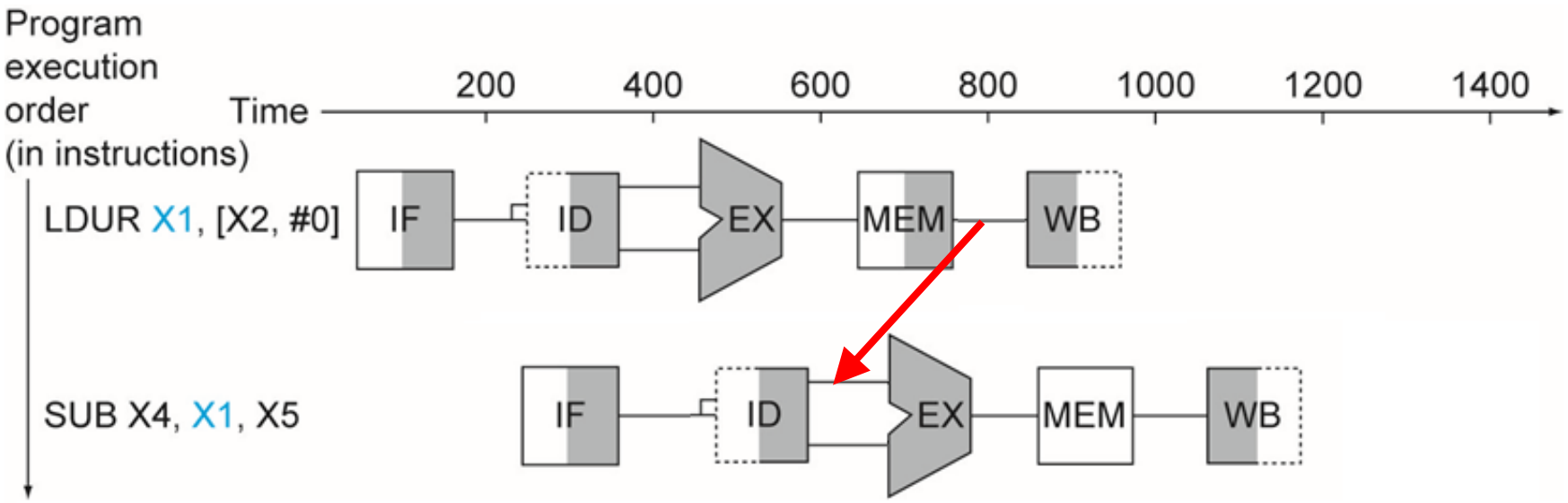
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath and extra control logic



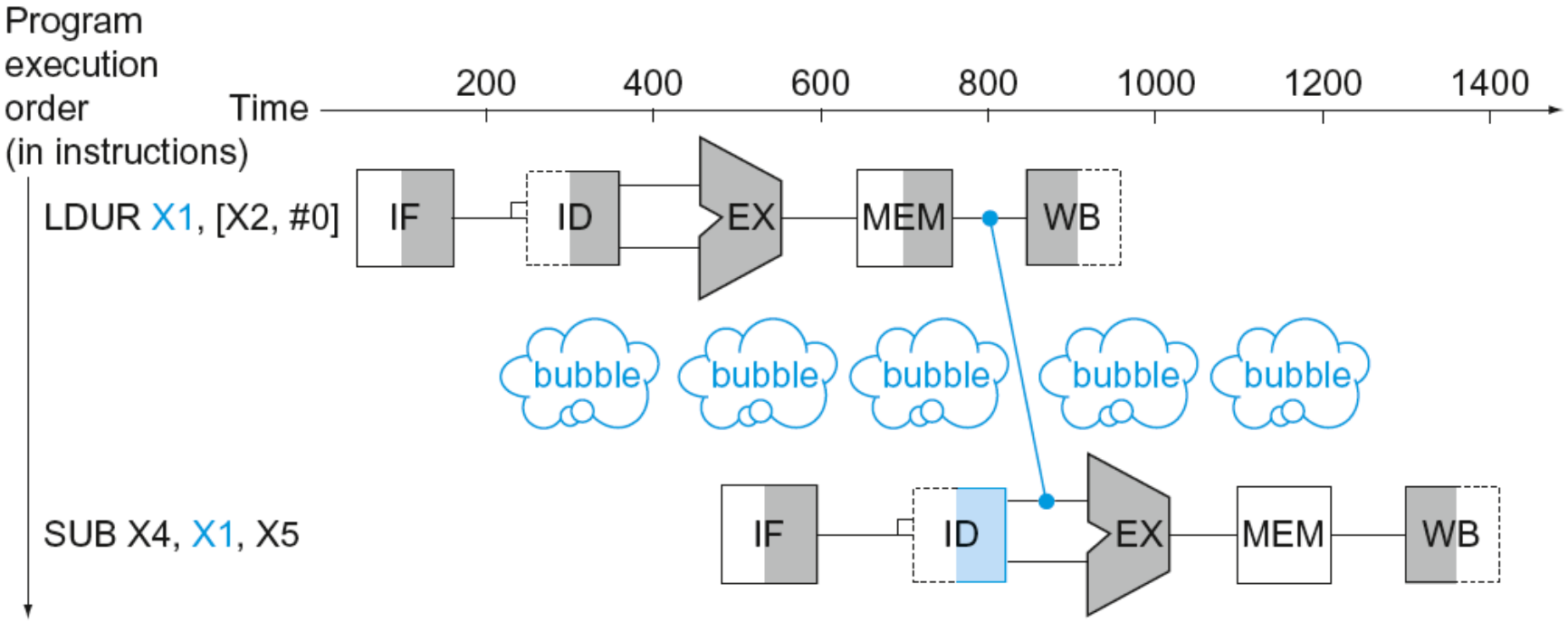
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!



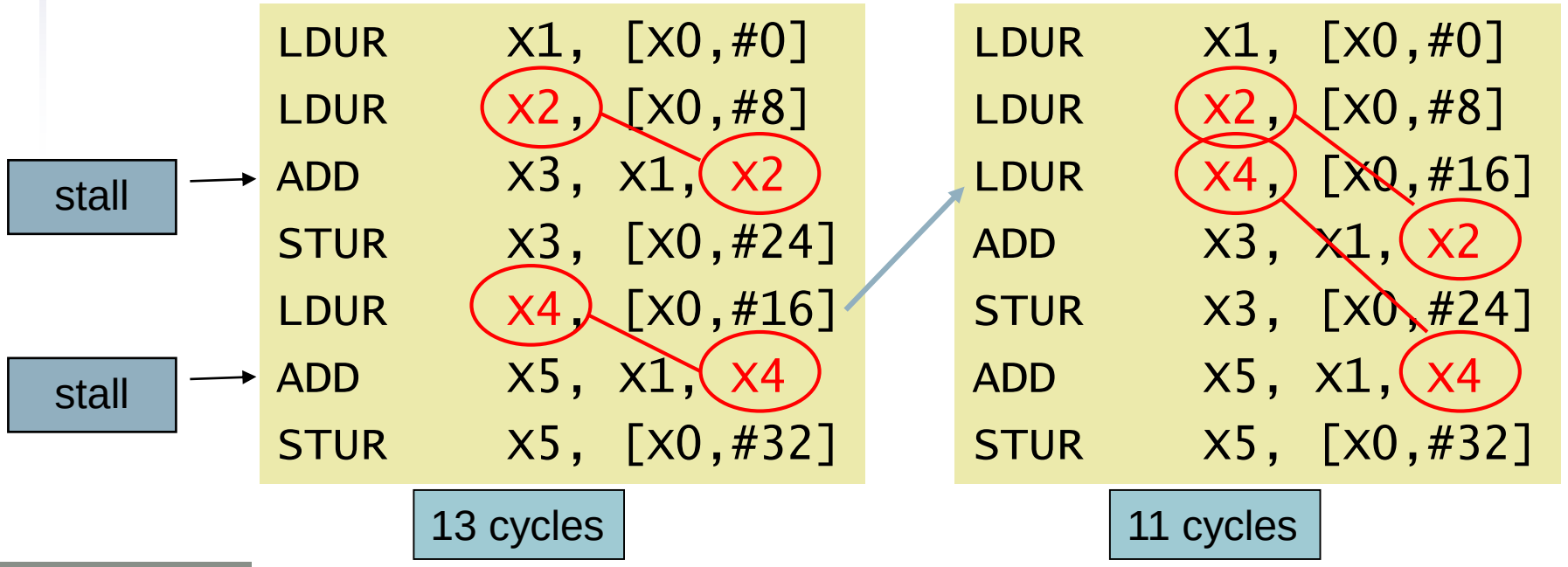
Load-Use Data Hazard

- Can’t avoid stalls by forwarding
  - Stall can be limited to one cycle by using forwarding.



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A[3]=A[0]+A[1]; A[4]=A[0]+A[2];`

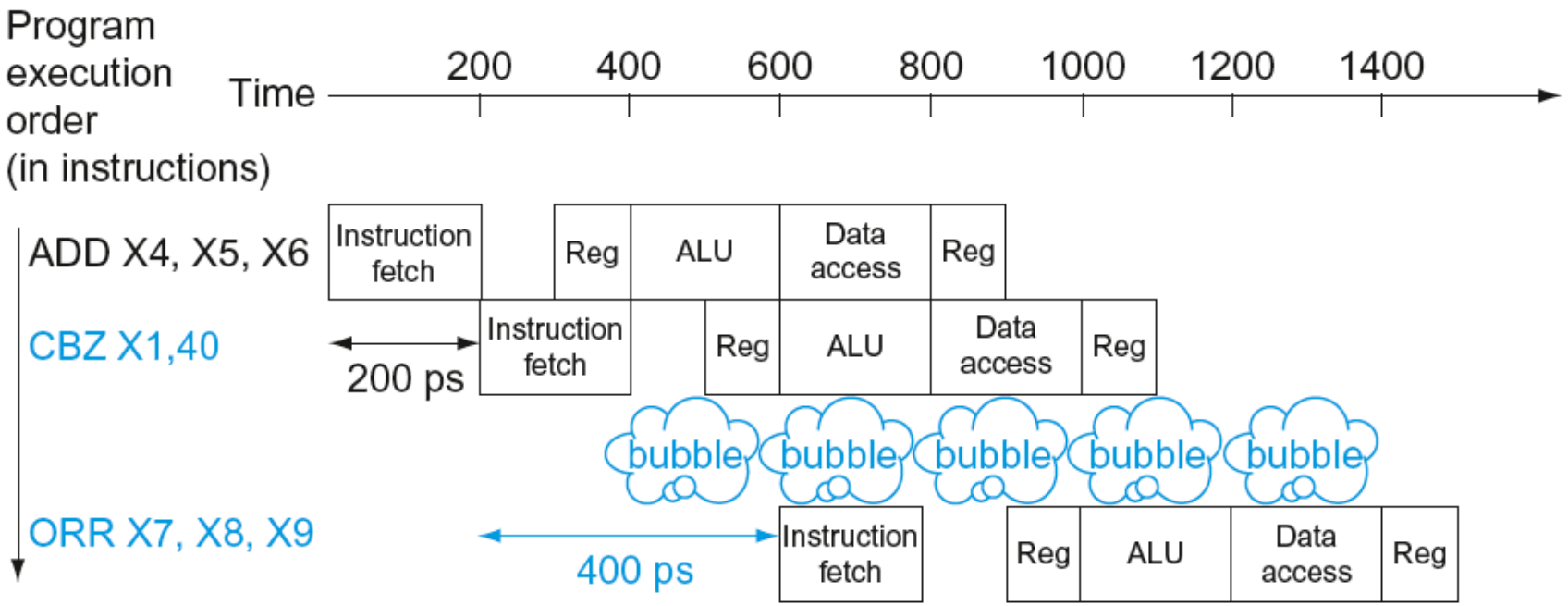


Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In LEGv8 pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In LEGv8 pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

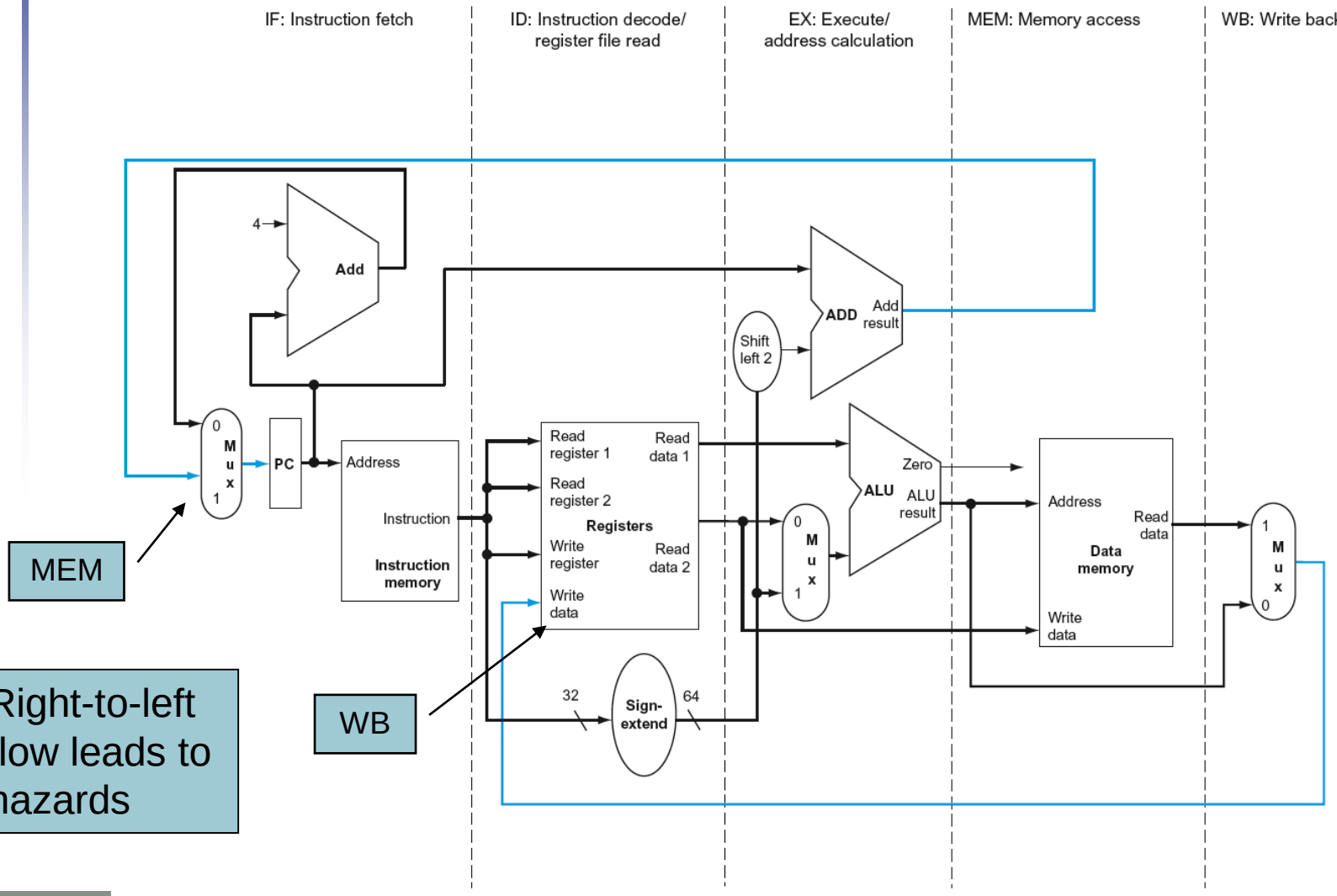
The BIG Picture

- 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

Planning CTA01

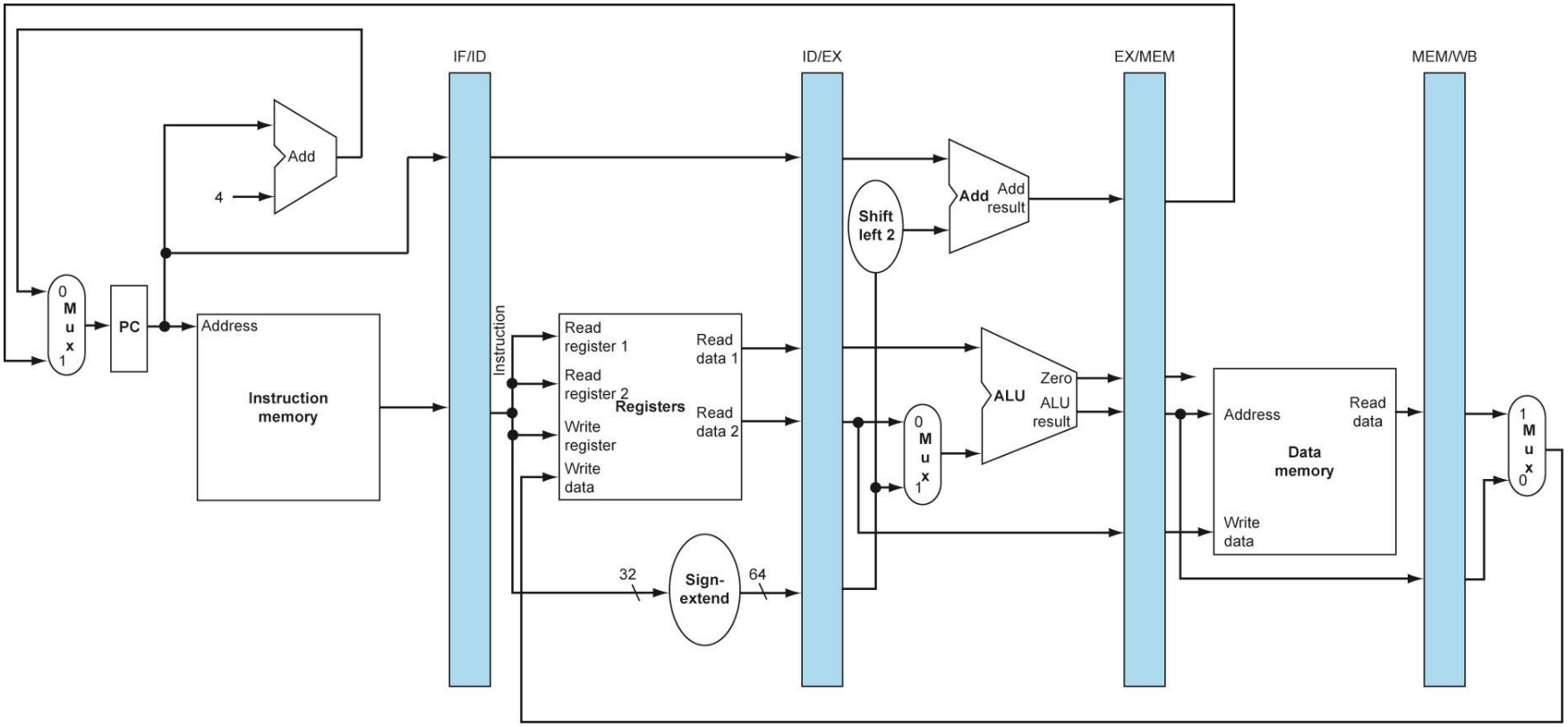
- Week 1: Introduction, Performance & Parallelism
- Week 2: Instructions for arithmetic and memory
- Week 3: Instructions for decisions and procedures
- Week 4: Computer arithmetic
- Week 5: Single cycle LEGv8, intro pipelining
- **Week 6: Pipelined LEGv8, hazards and forwarding**
- Week 7: Memory architecture
- Week 8: Guest lecture (multicore and parallelism)

LEGv8 Pipelined Datapath



Pipeline registers

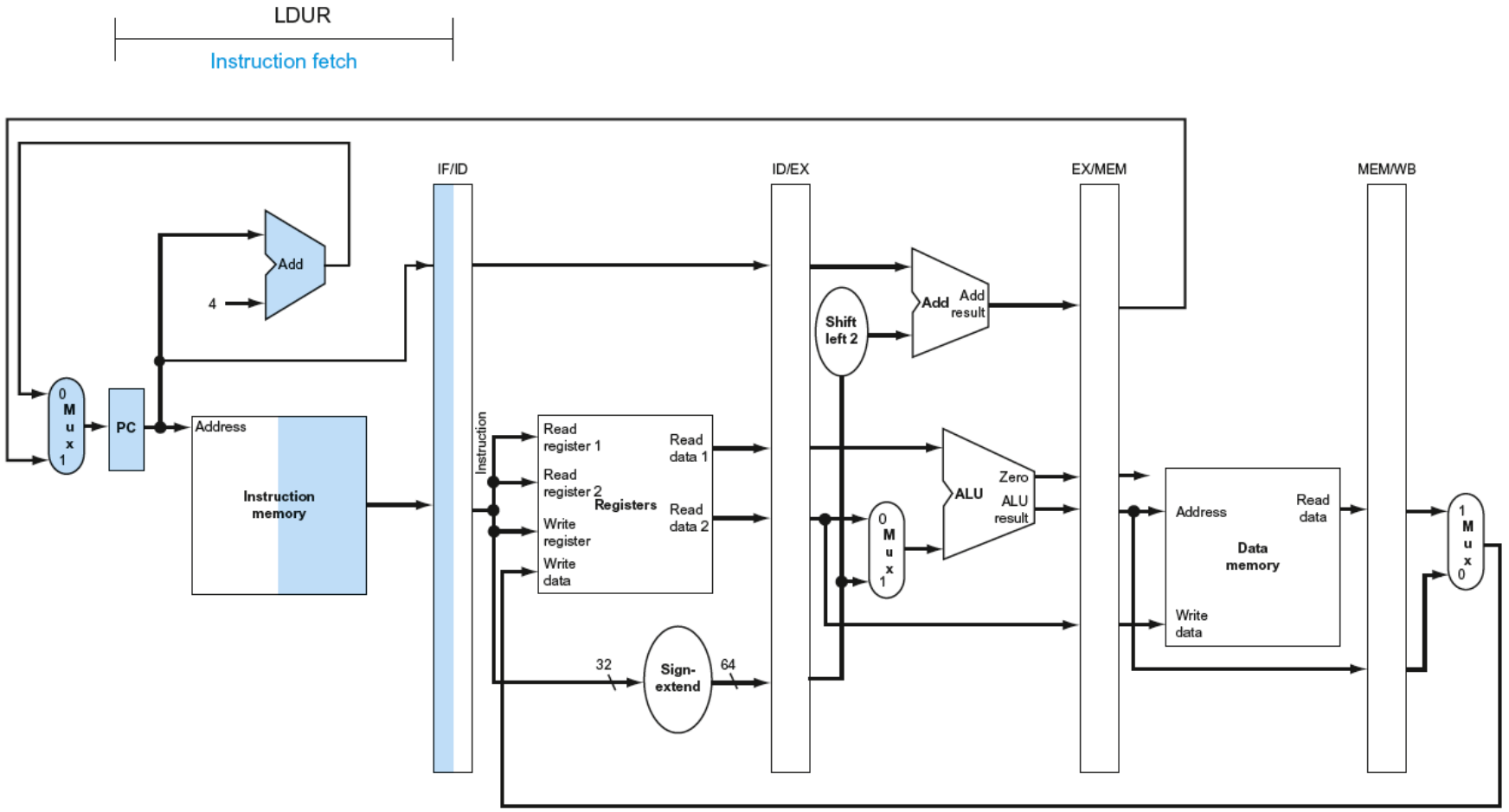
- Need registers between stages
  - To hold information produced in previous cycle



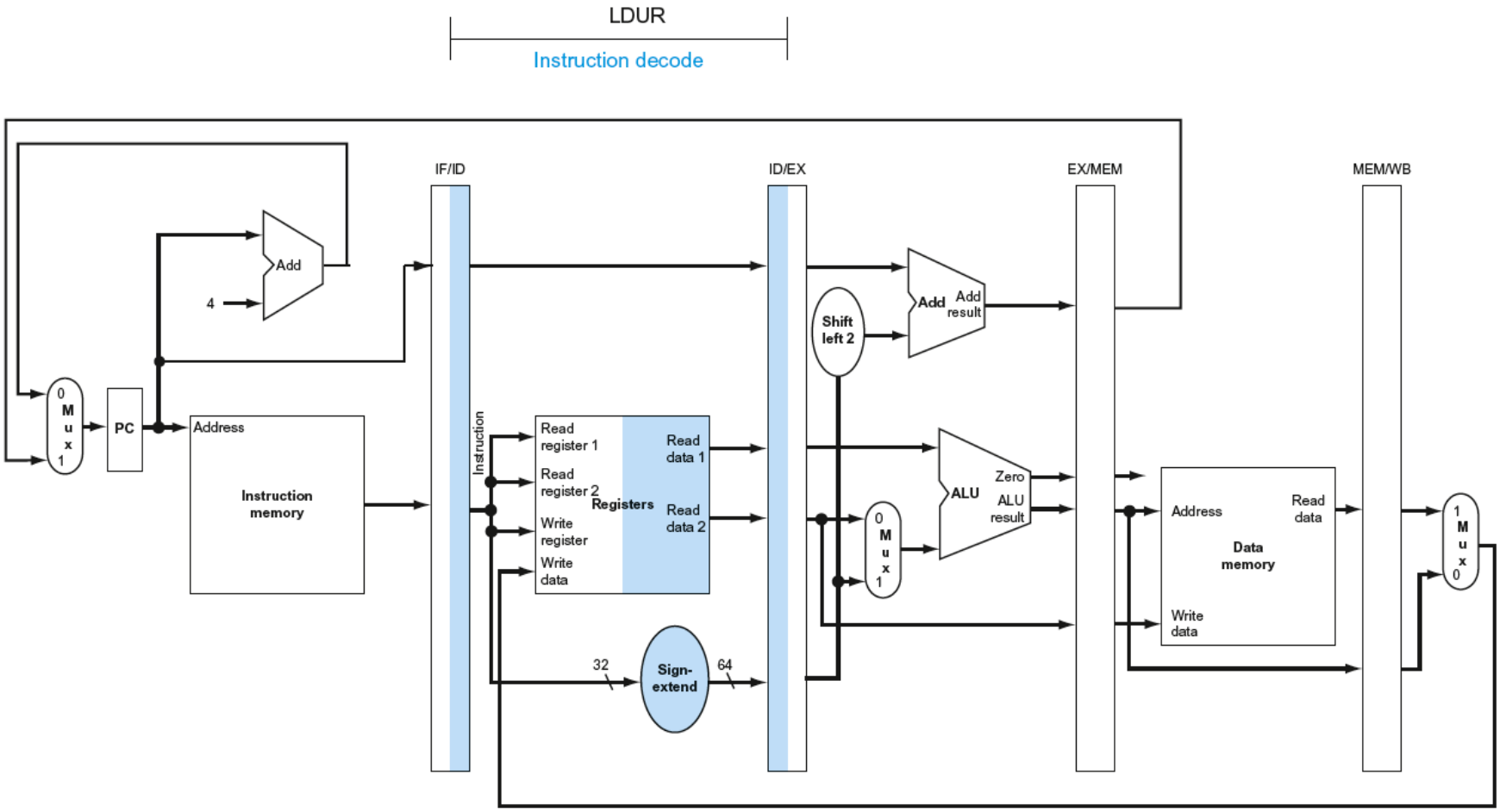
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - “Multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

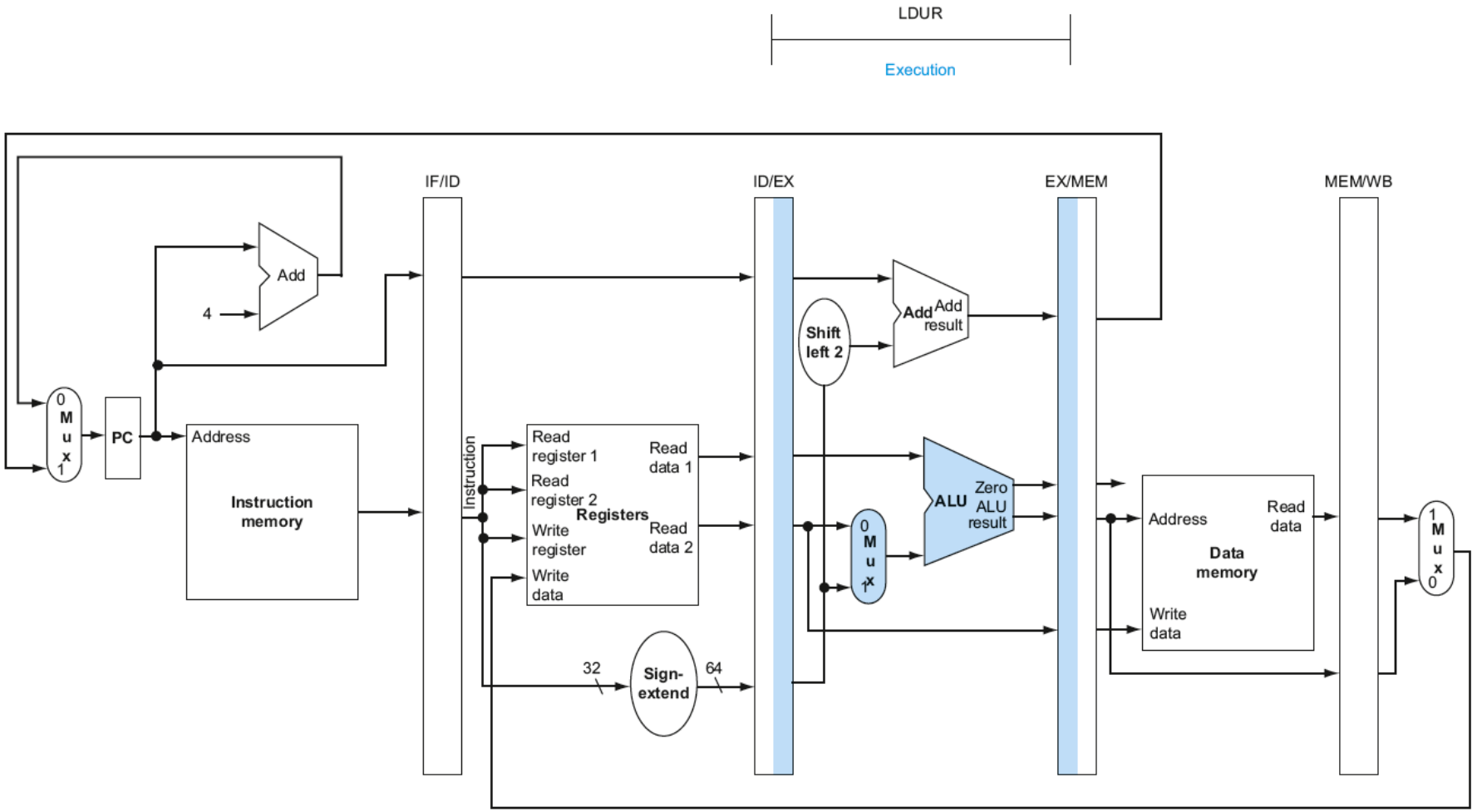
IF for Load, Store, …



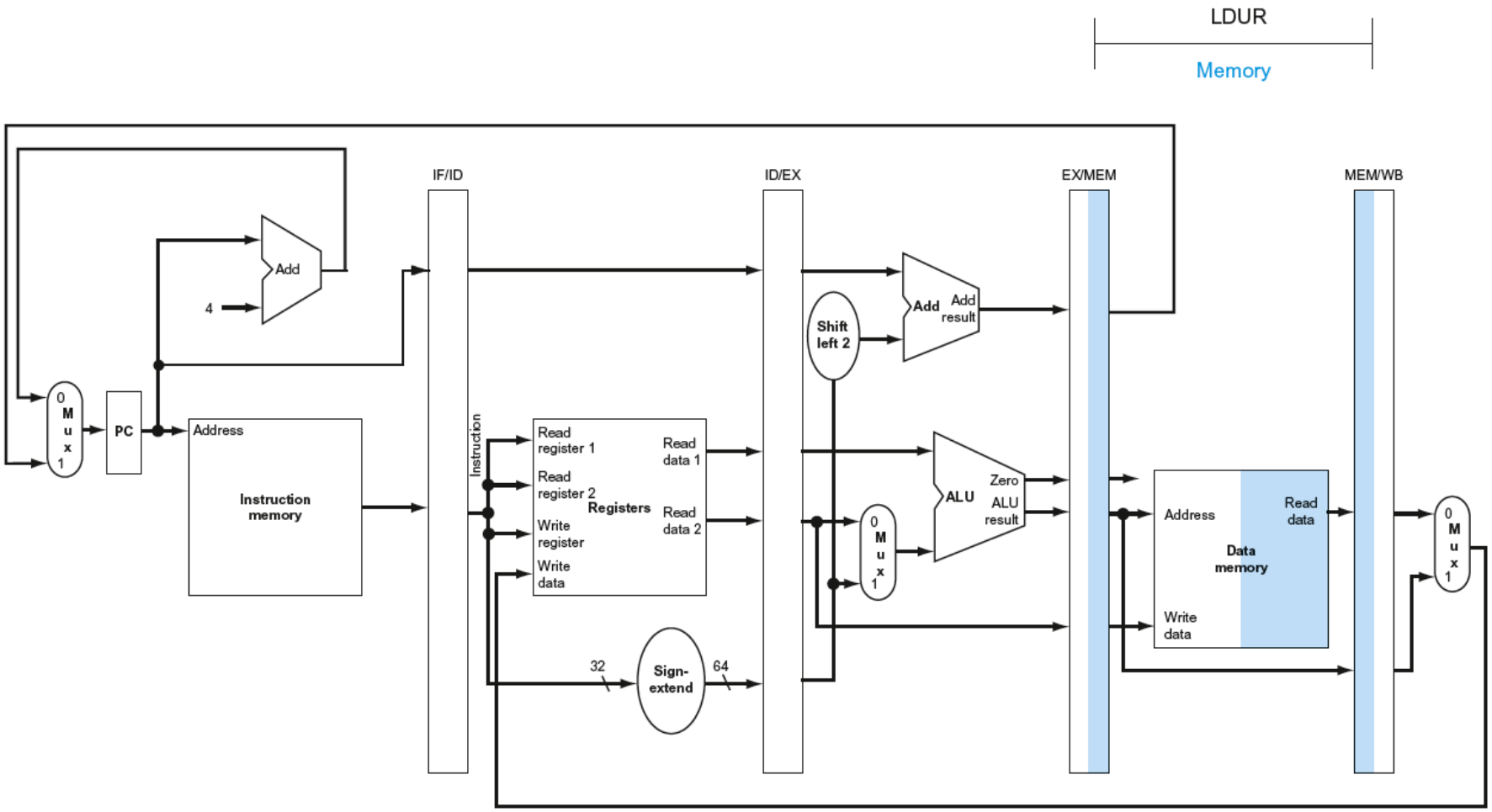
ID for Load, Store, …



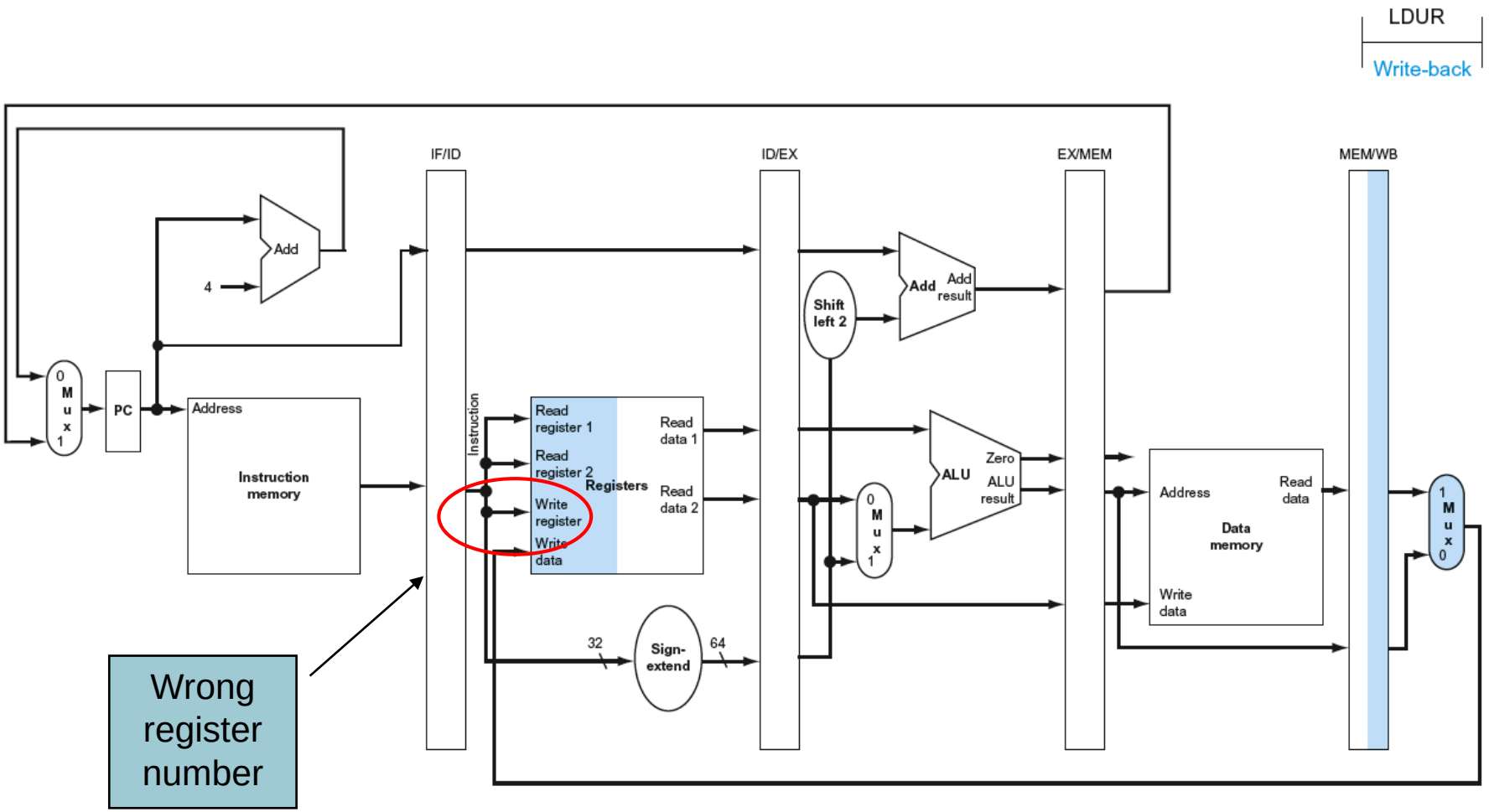
EX for Load


LDUR
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Sign-extend
64
Shift left 2
Add
Add result
0
M
u
x
1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0

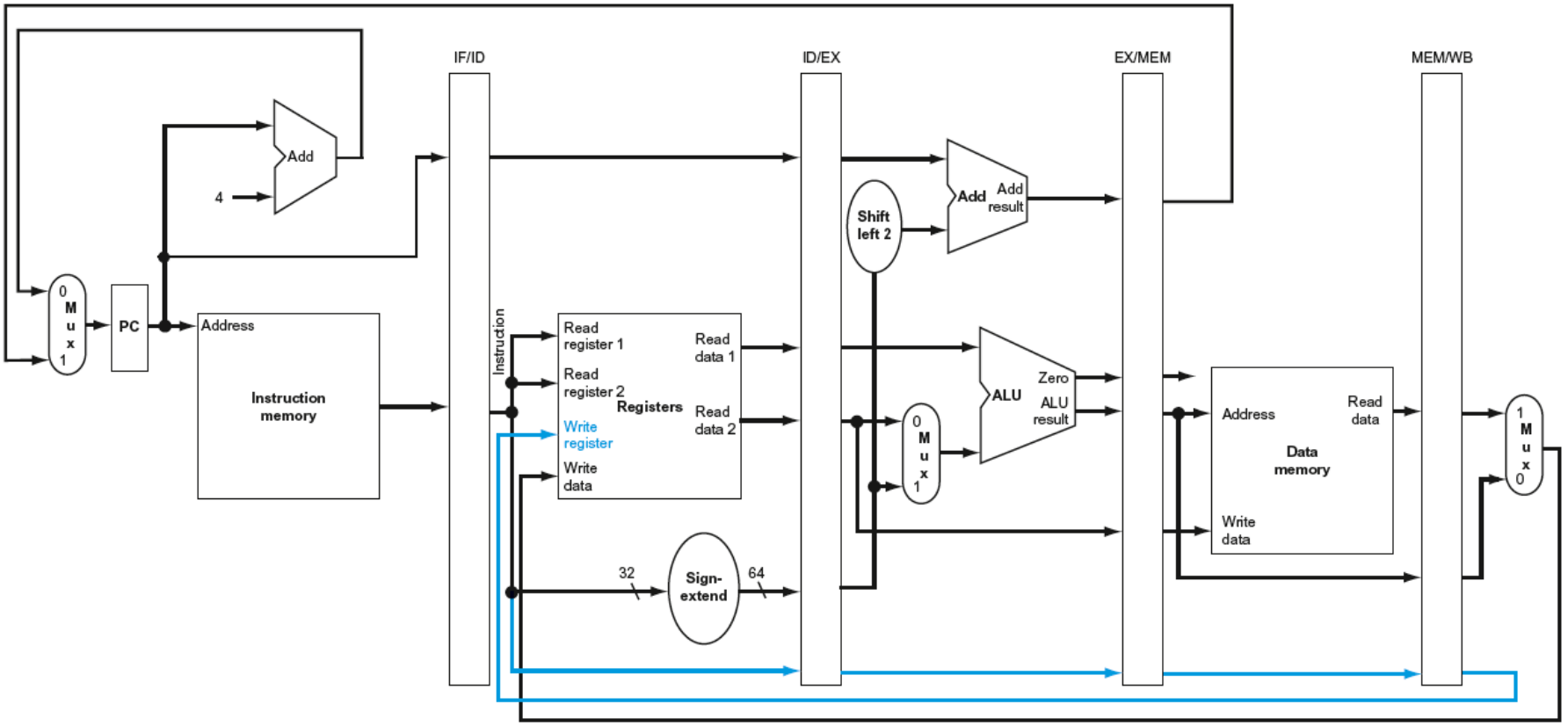
MEM for Load


LDUR
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
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
Data memory
Read data
Write data
32
Sign-extend
64

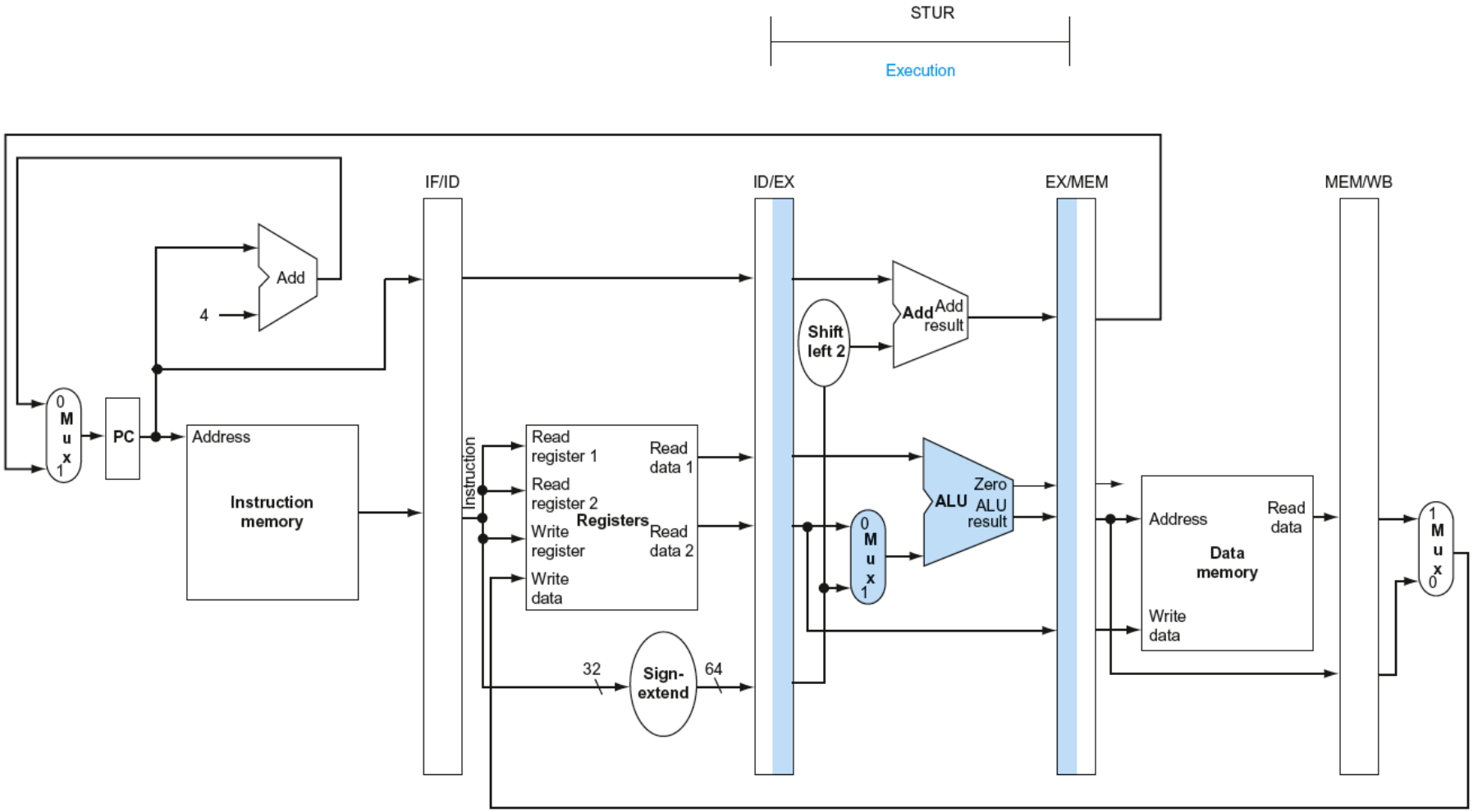
WB for Load


LDUR
Write-back
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
32
Sign-extend
64
Wrong register number

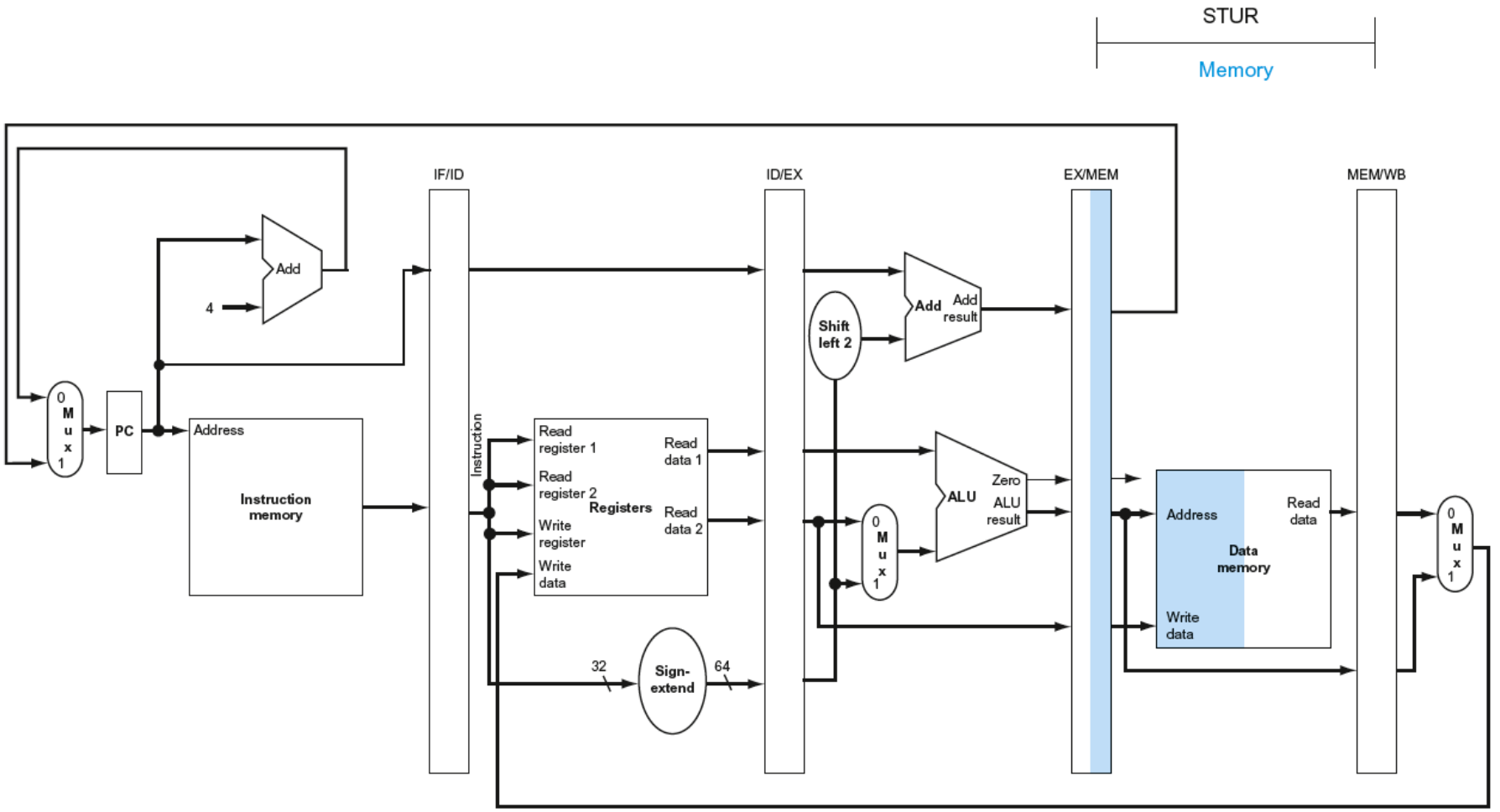
Corrected Datapath for Load



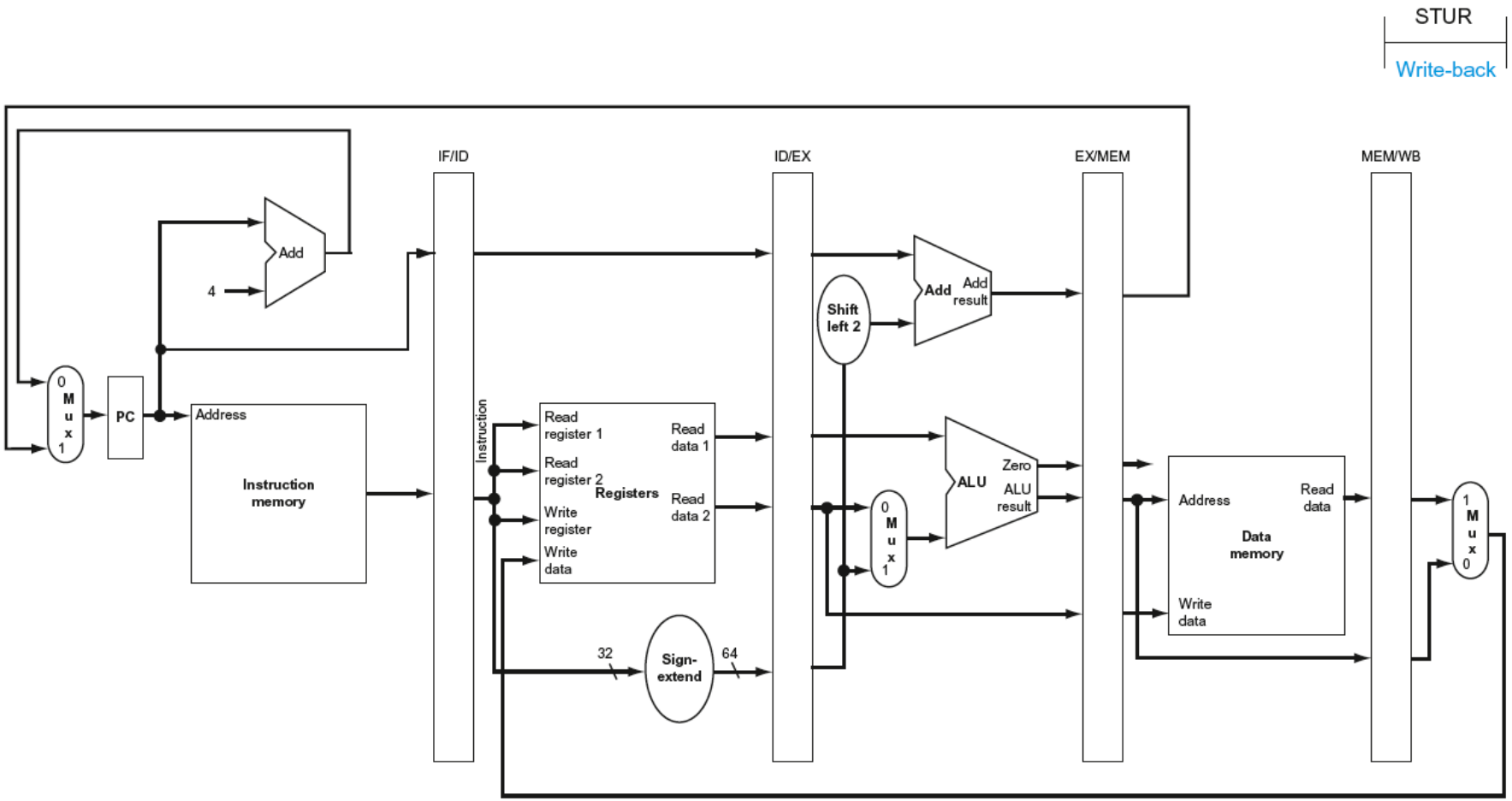
EX for Store


STUR
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
Address
Read data
Data memory
Write data
32
Sign-extend
64

MEM for Store

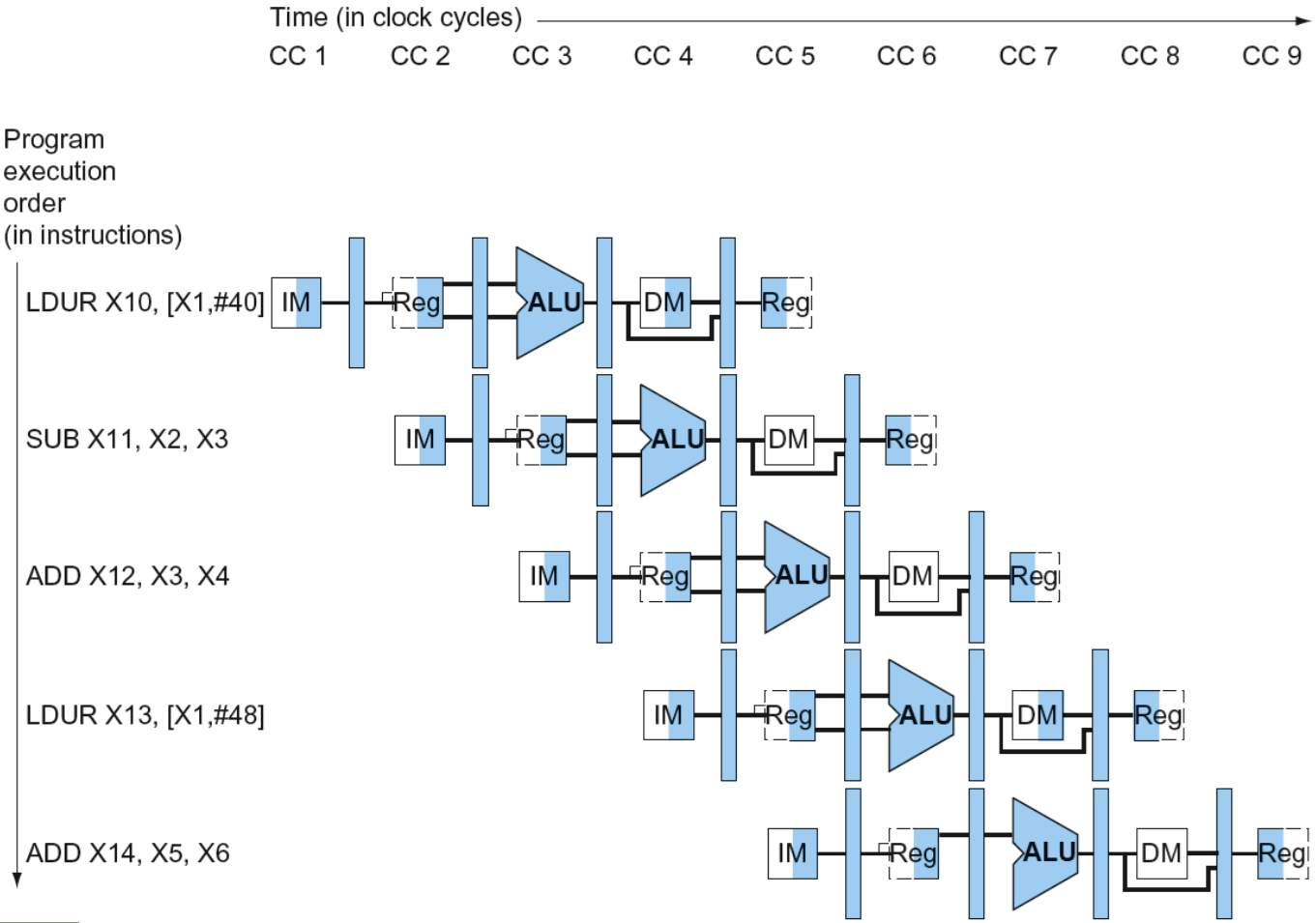


WB for Store


STUR
Write-back
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
32
Sign-extend
64
Address
Read data
Data memory
Write data

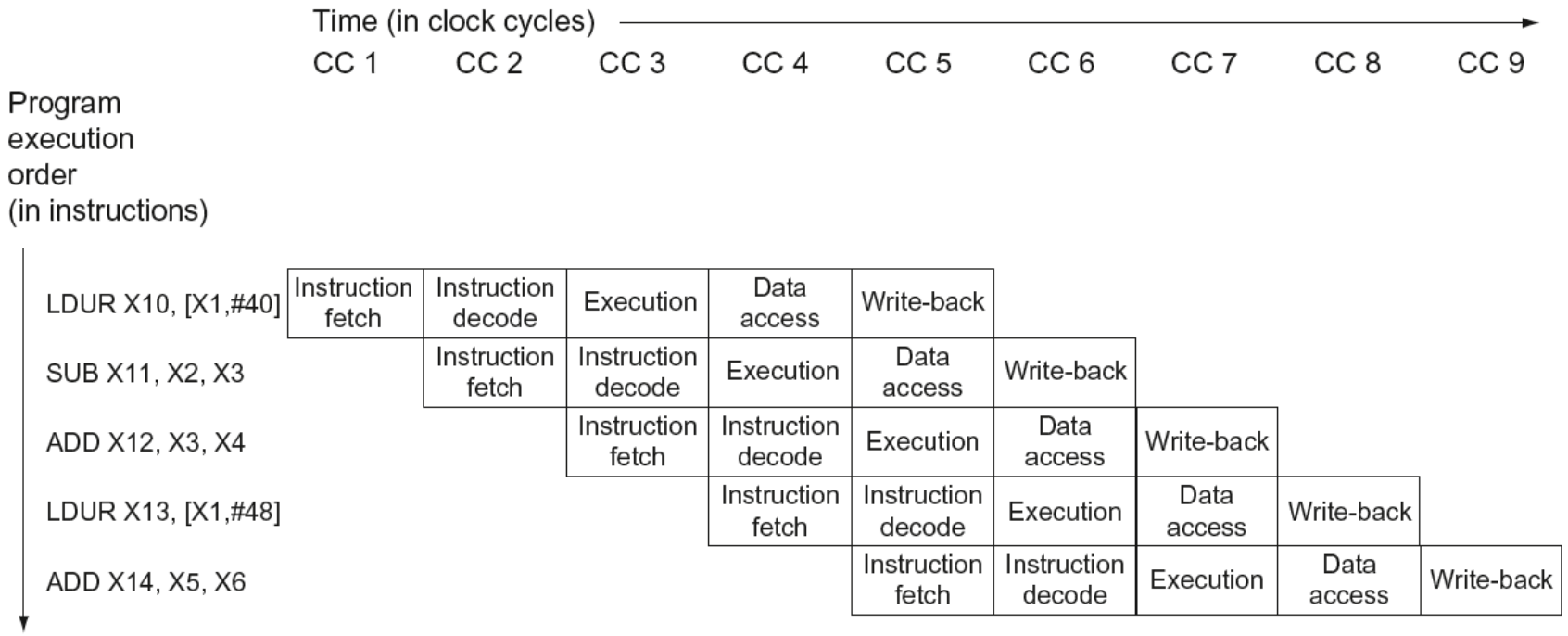
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

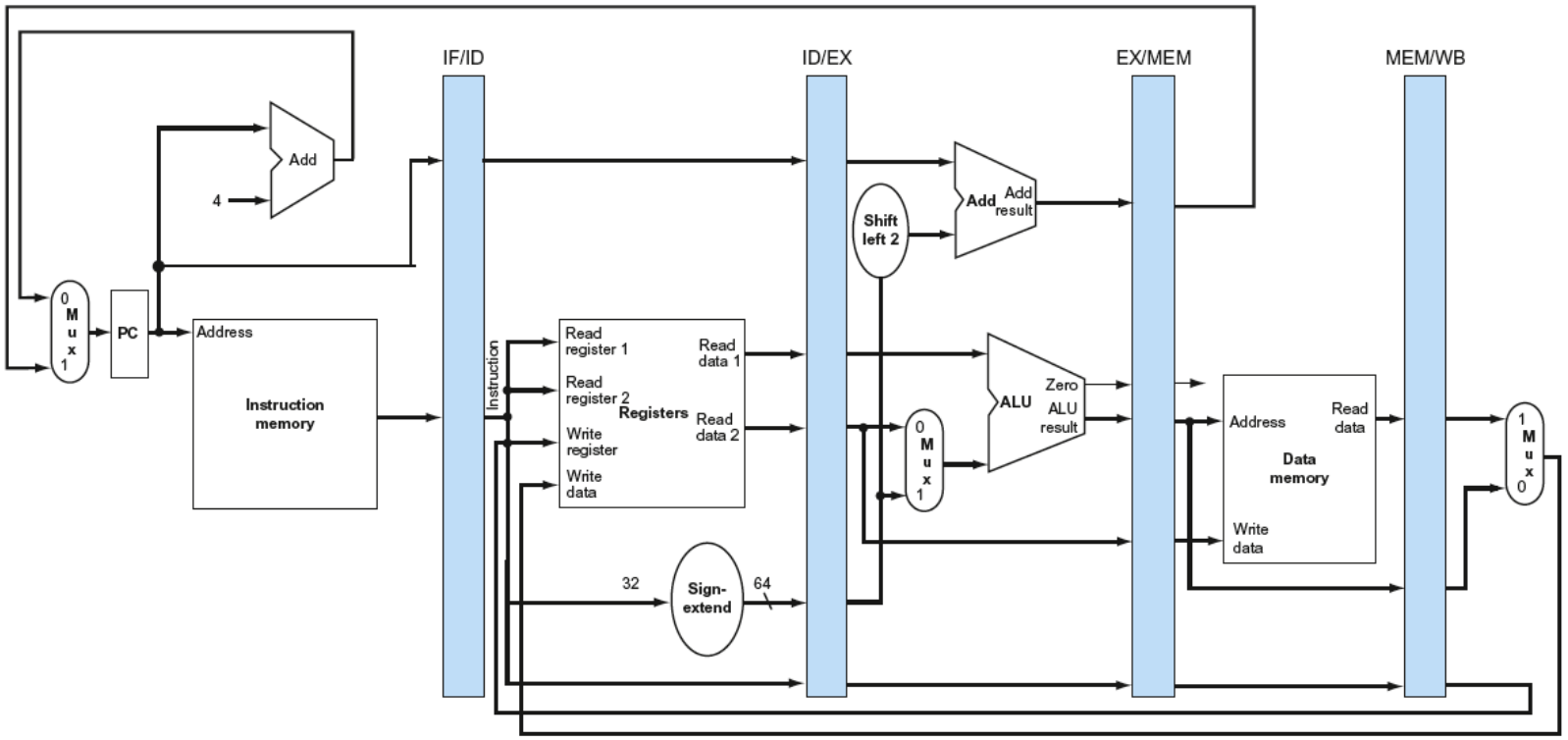
- Traditional form



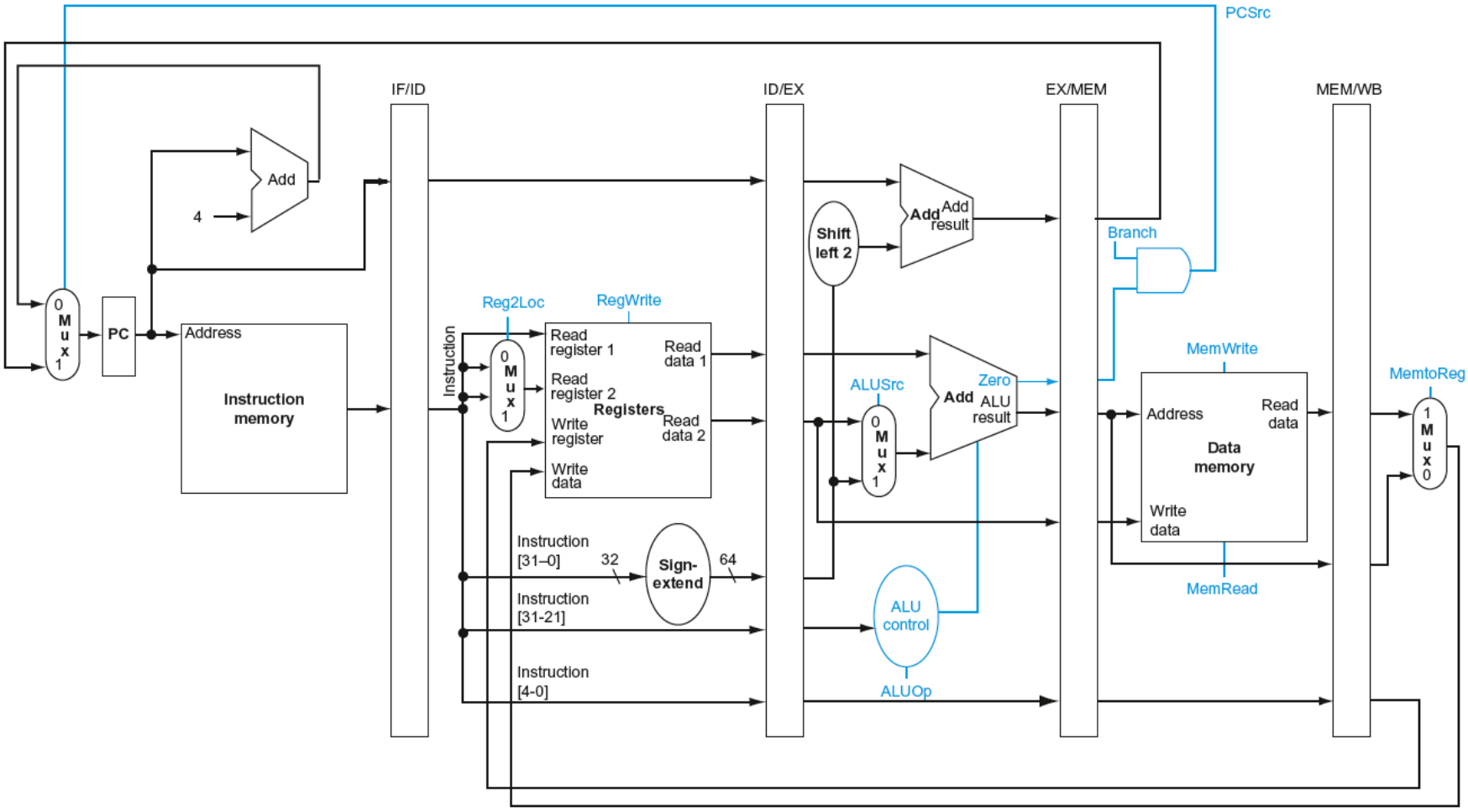
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle, e.g. CC 5

ADD X14, X5, X6	LDUR X13, [X1,48]	ADD X12, X3, X4	SUB X11, X2, X3	LDUR X10, [X1,40]
Instruction fetch	Instruction decode	Execution	Memory	Write-back

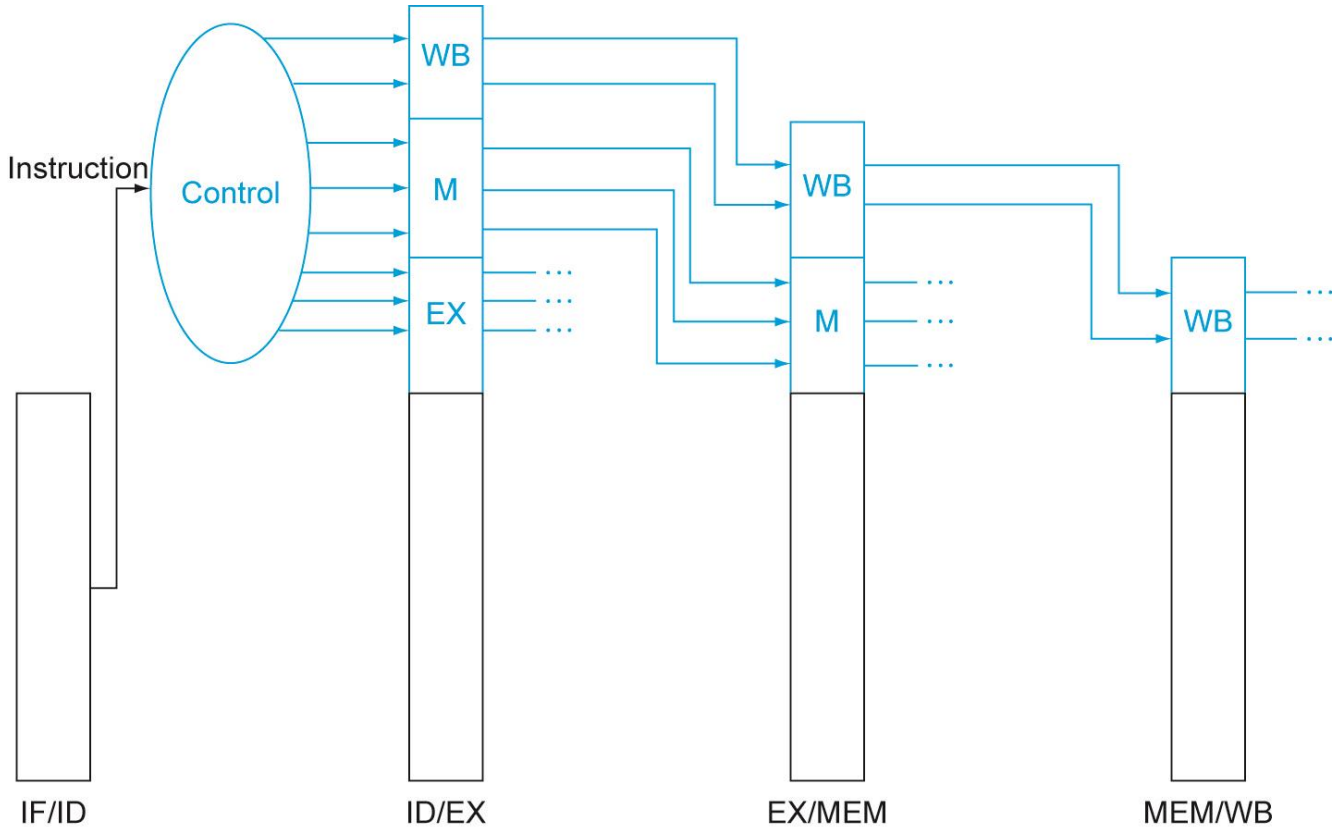


Pipelined Control (Simplified)

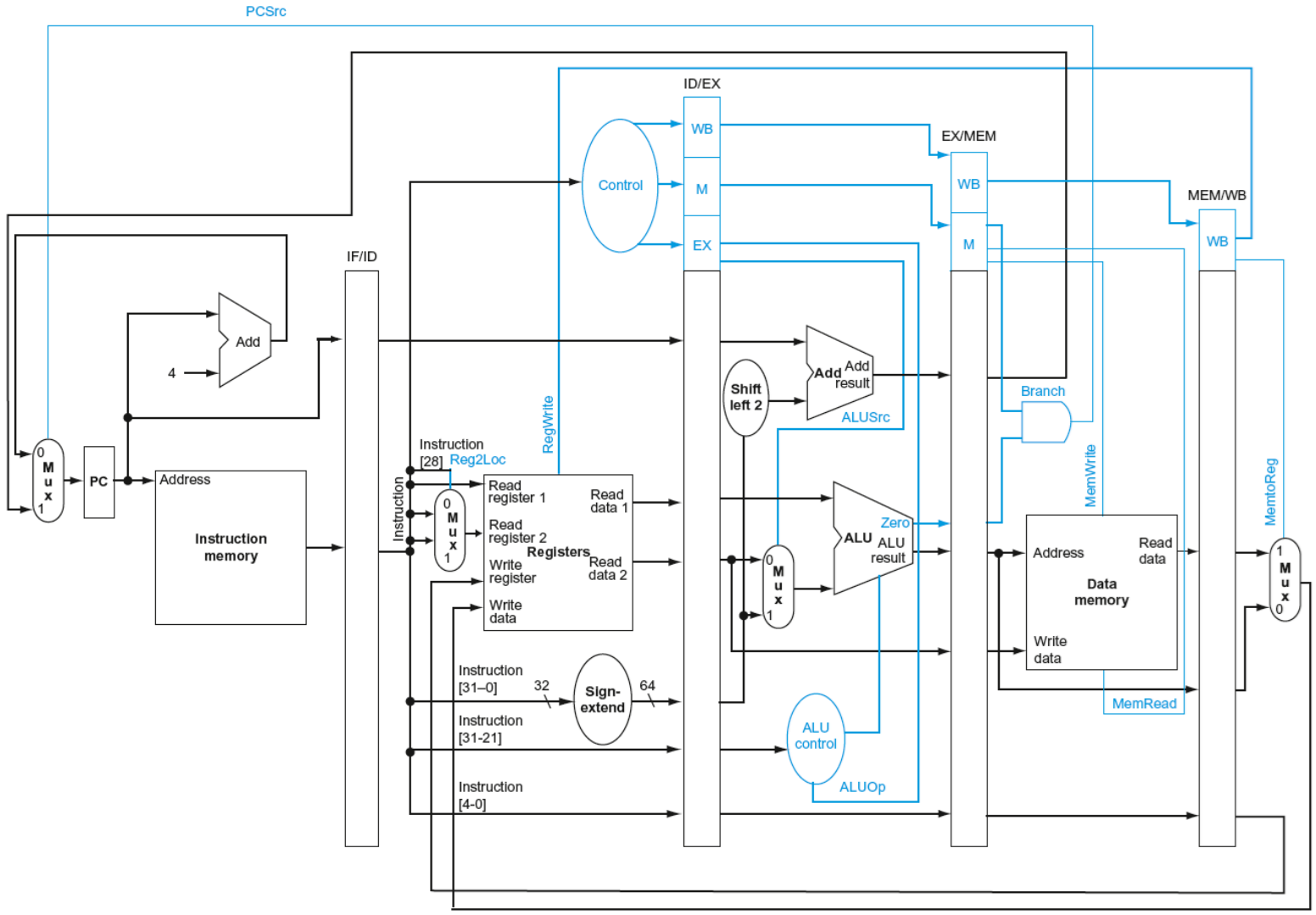


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



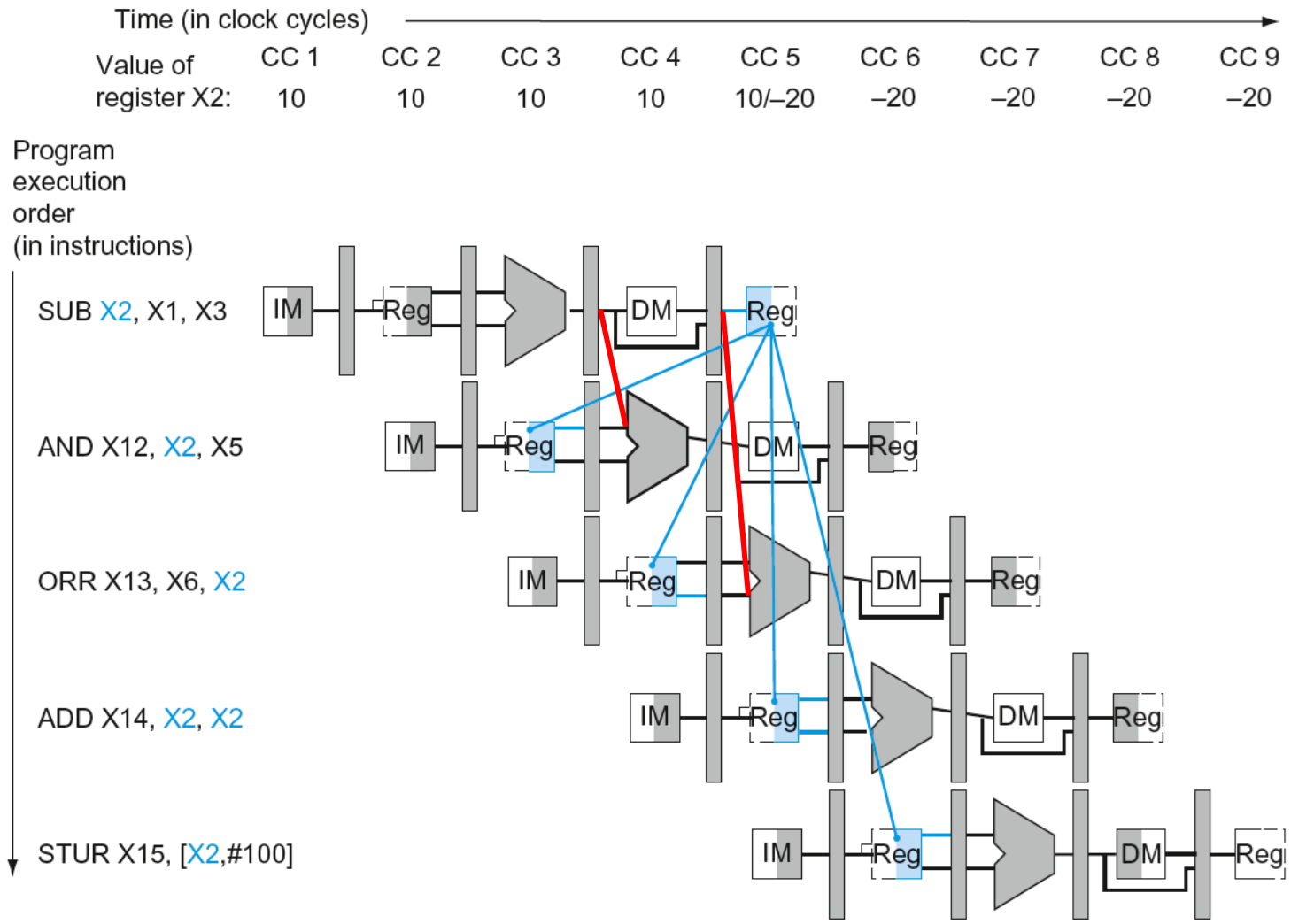
Data Hazards in ALU Instructions

- Consider this sequence:

```
SUB   x2,x1,x3
AND   x12,x2,x5
OR    x13,x6,x2
ADD   x14,x2,x2
STUR  x15,[x2,#100]
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRn1, ID/EX.RegisterRm2
- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRn1
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRm2

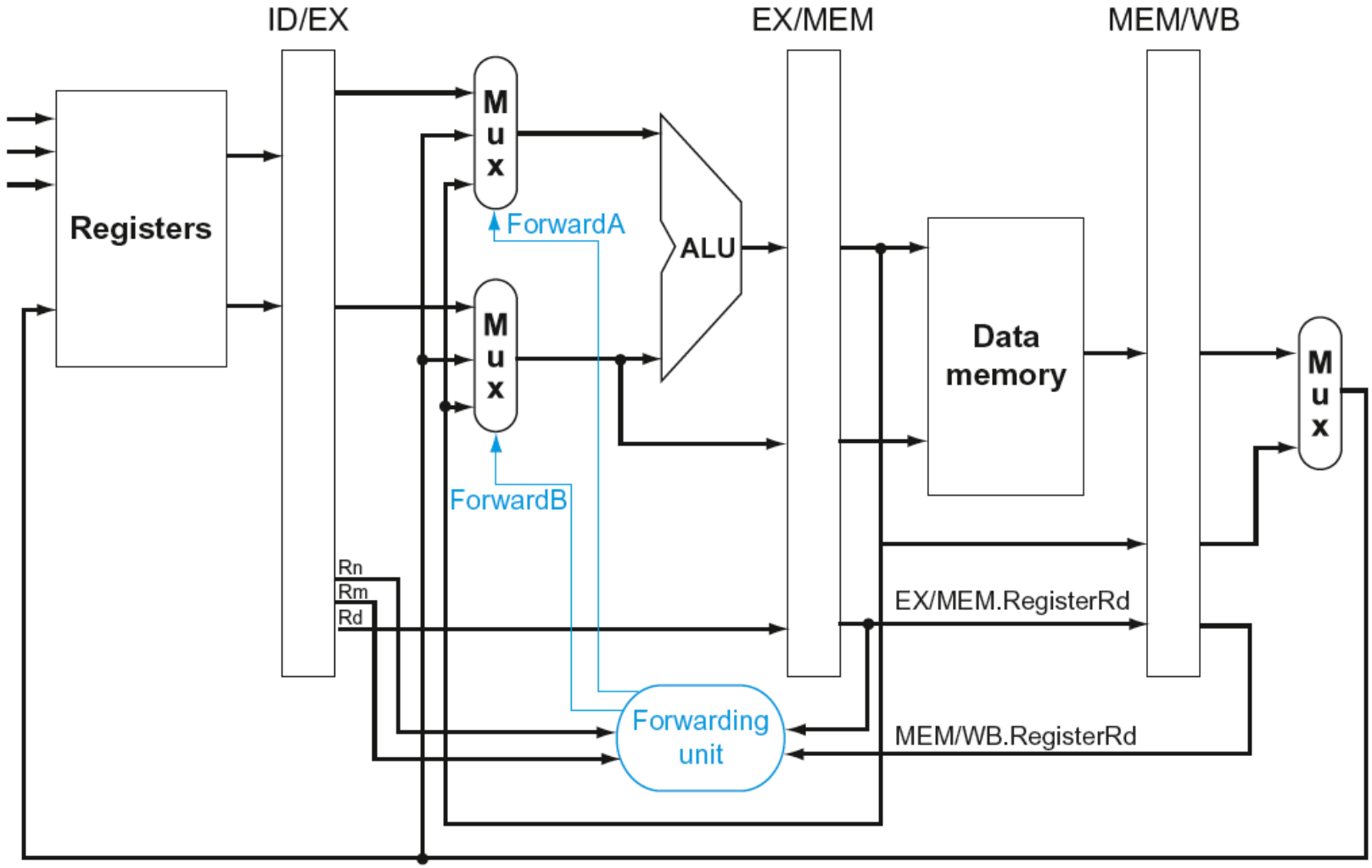
Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not XZR
  - EX/MEM.RegisterRd ≠ 31, MEM/WB.RegisterRd ≠ 31

Forwarding Paths



Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Forwarding Condition EX hazard

- EX hazard
  - if (EX/MEM.RegWrite

    and (EX/MEM.RegisterRd ≠ 31)

    and (EX/MEM.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 10
  - if (EX/MEM.RegWrite

    and (EX/MEM.RegisterRd ≠ 31)

    and (EX/MEM.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 10

Forwarding Condition MEM hazard

- MEM hazard
  - if (MEM/WB.RegWrite

    and (MEM/WB.RegisterRd ≠ 31)

    and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01
  - if (MEM/WB.RegWrite

    and (MEM/WB.RegisterRd ≠ 31)

    and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01

Dit is nog niet helemaal correct!
Zie volgende slide.

Double Data Hazard

- Consider the sequence:

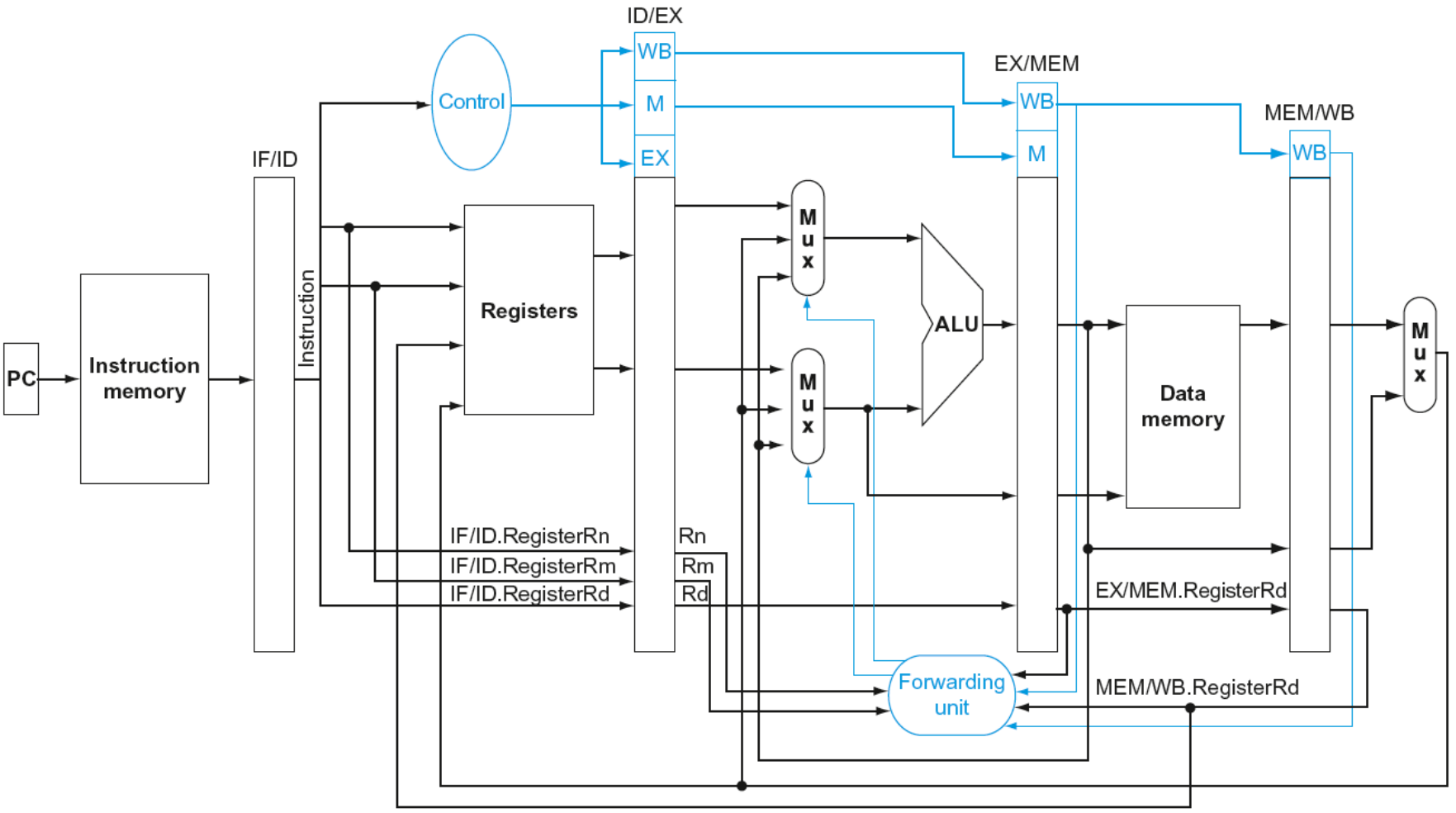
```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

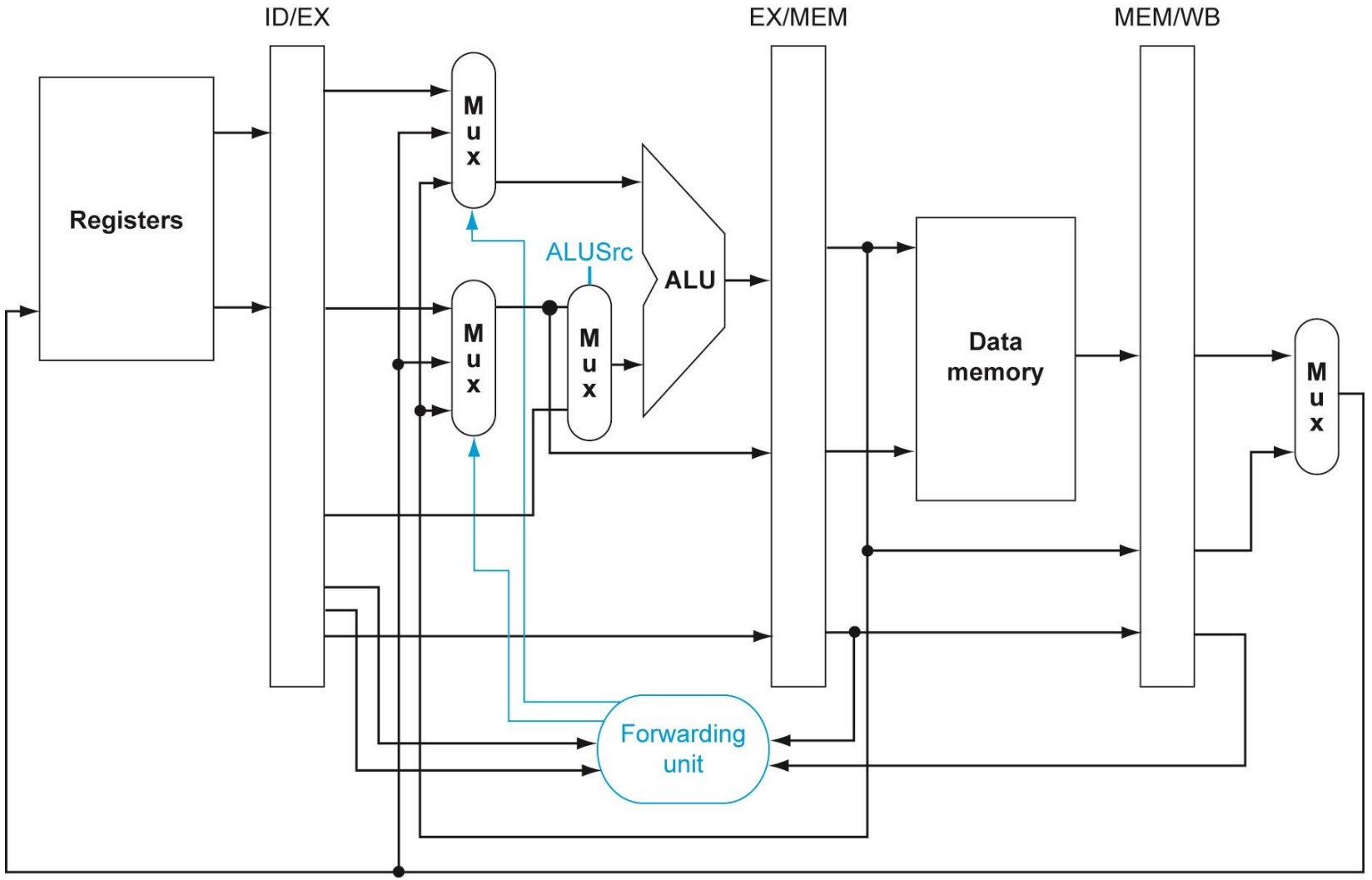
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 31)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRn1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 31)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRm2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01

Datapath with Forwarding



Datapath Detail



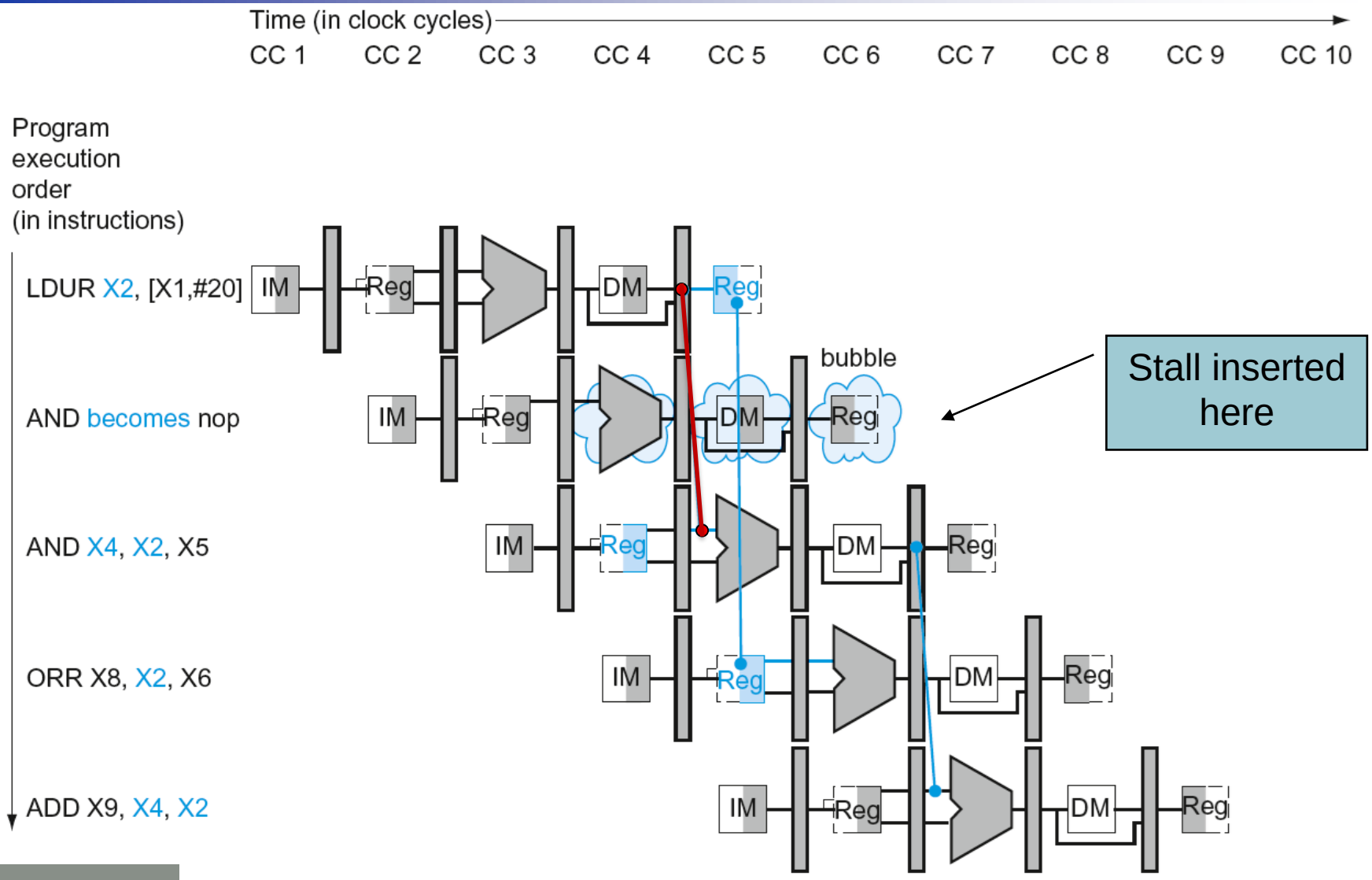
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRn1, IF/ID.RegisterRm2
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRn1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRm2))
- If detected, stall and insert bubble

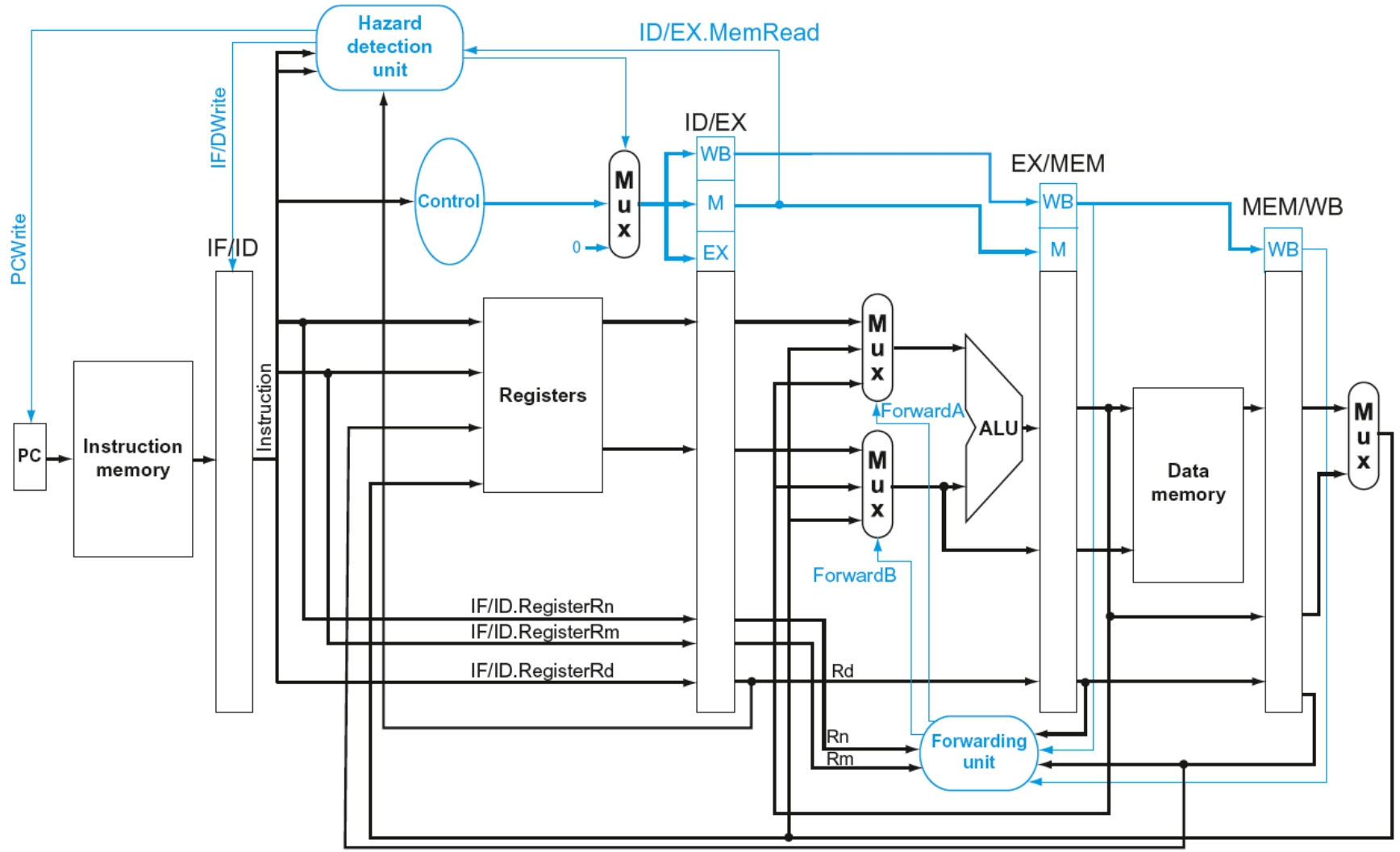
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for LDUR
    - Can subsequently forward to EX stage

Load-Use Data Hazard



Datapath with Hazard Detection



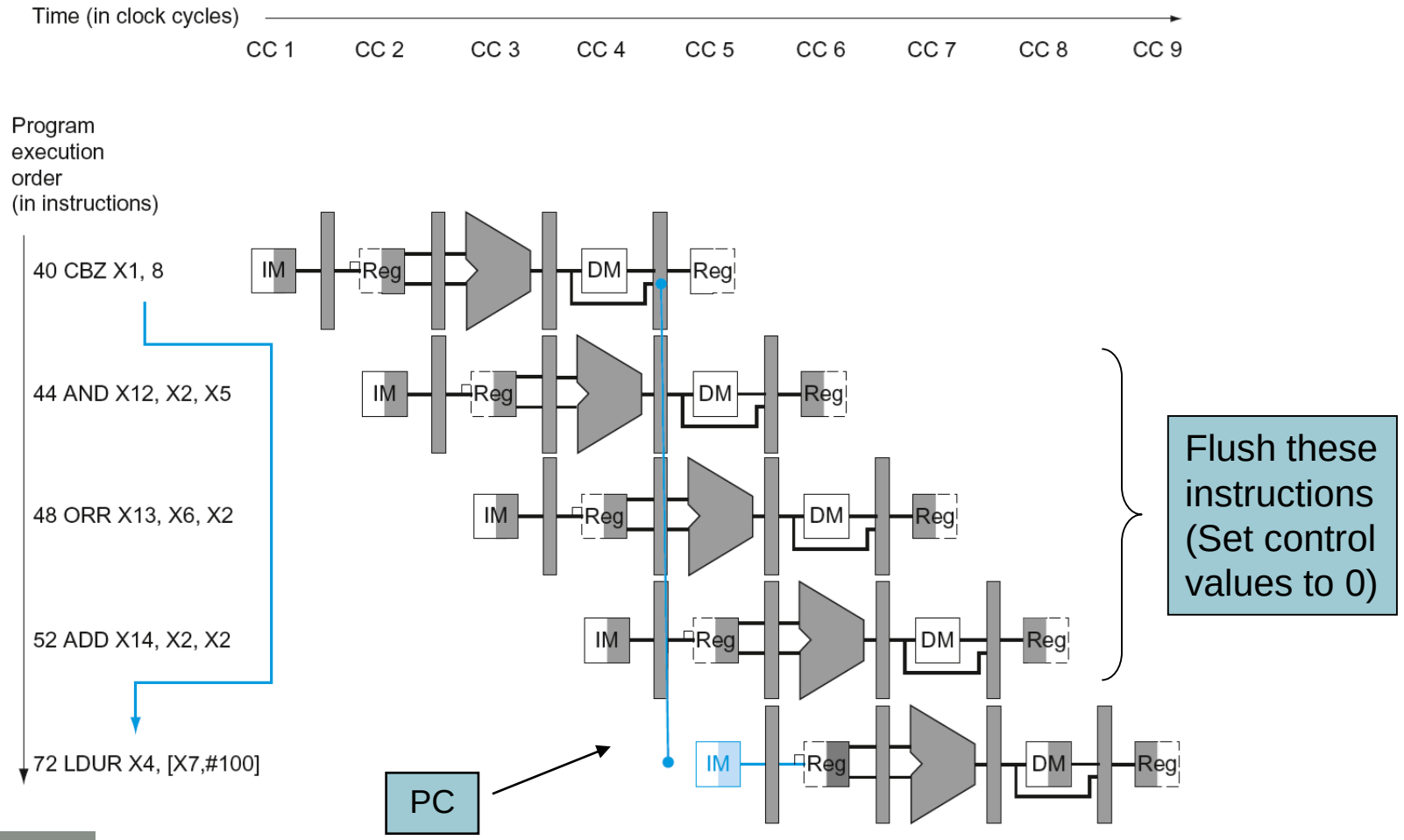
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM

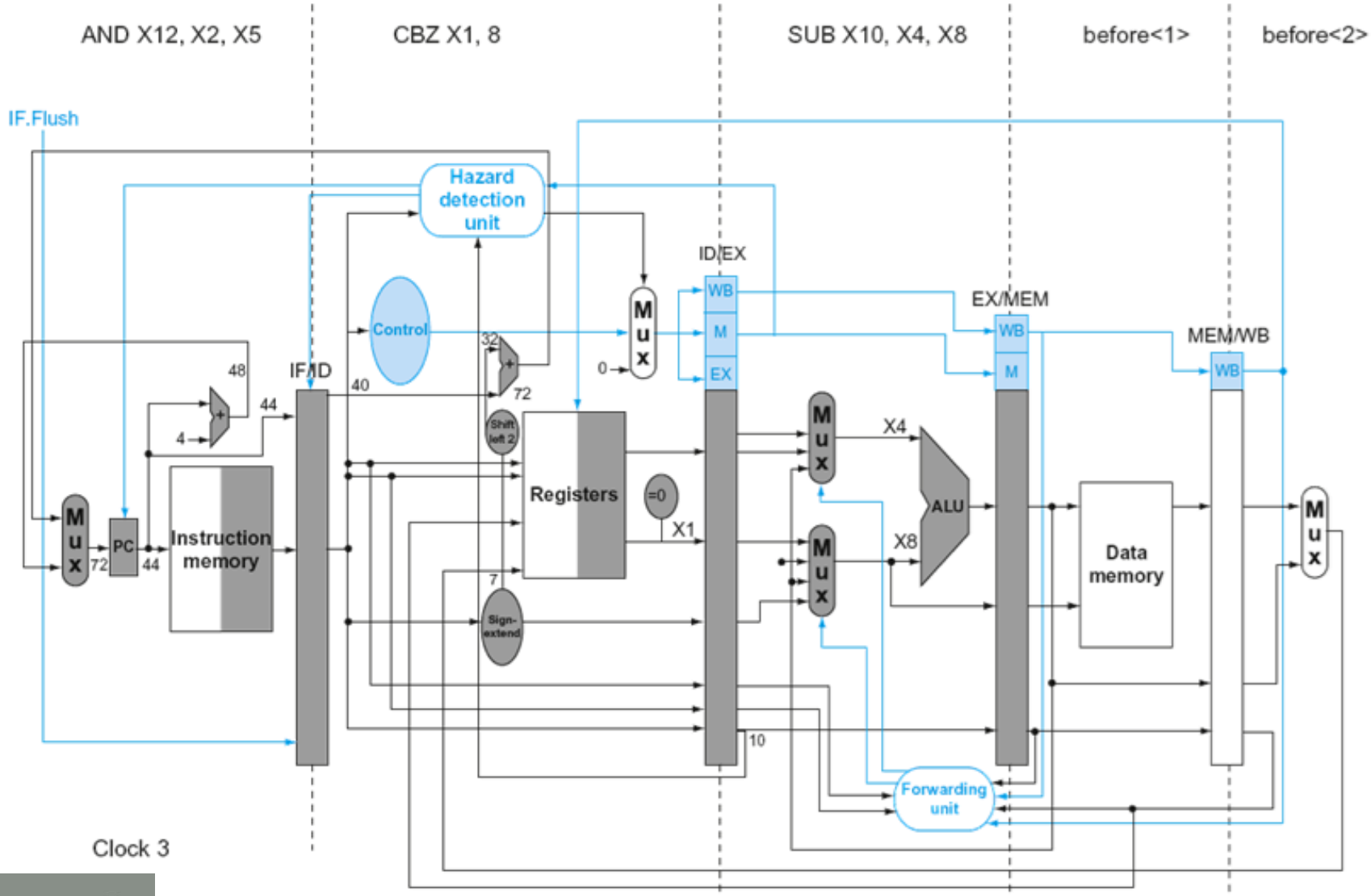


Reducing Branch Delay

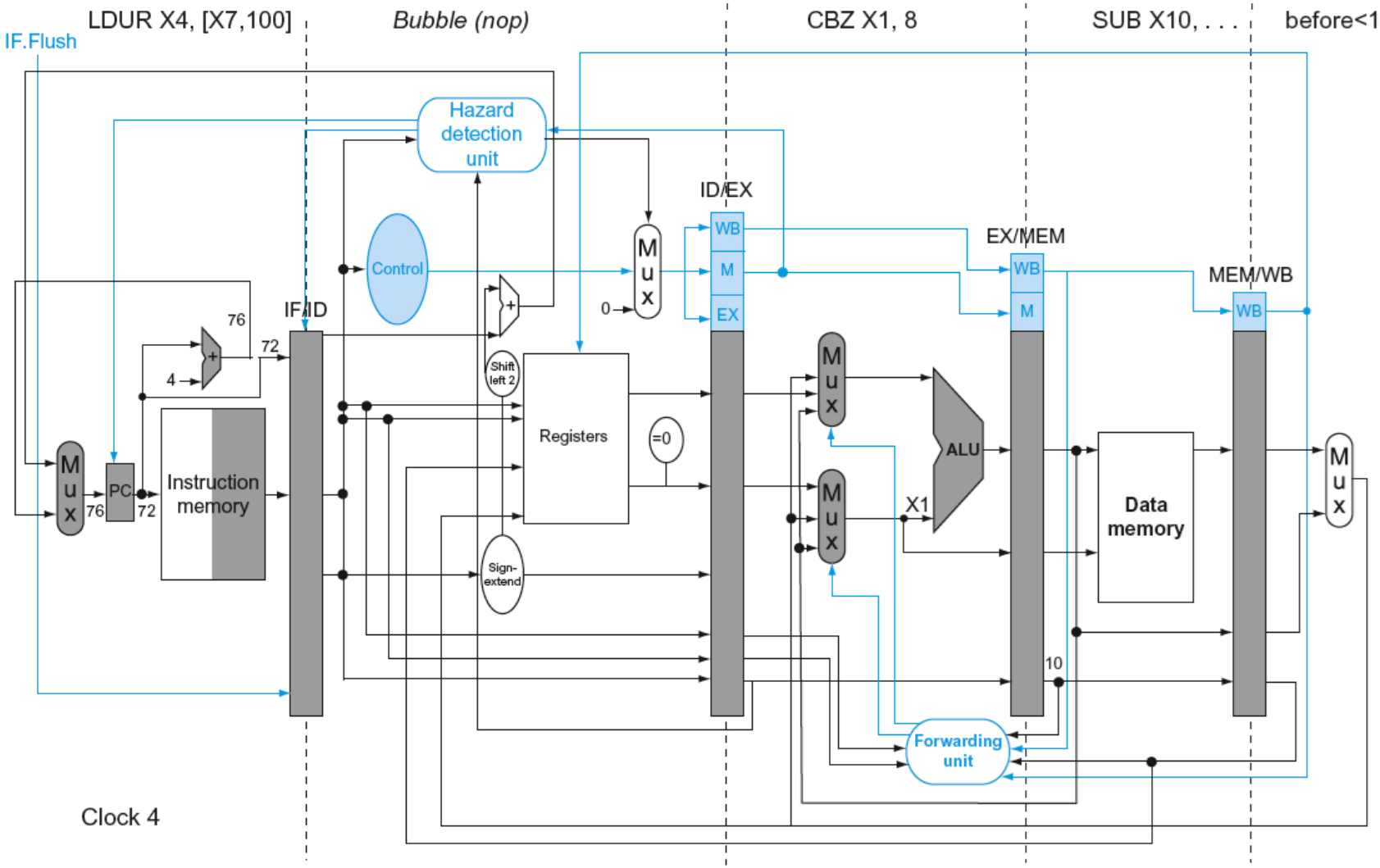
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   SUB  X10, X4, X8
40:   CBZ  X1,  X3, 8
44:   AND  X12, X2, X5
48:   ORR  X13, X2, X6
52:   ADD  X14, X4, X2
56:   SUB  X15, X6, X7
      ...
72:   LDUR X4, [X7,#50]
```

Example: Branch Taken



Example: Branch Taken

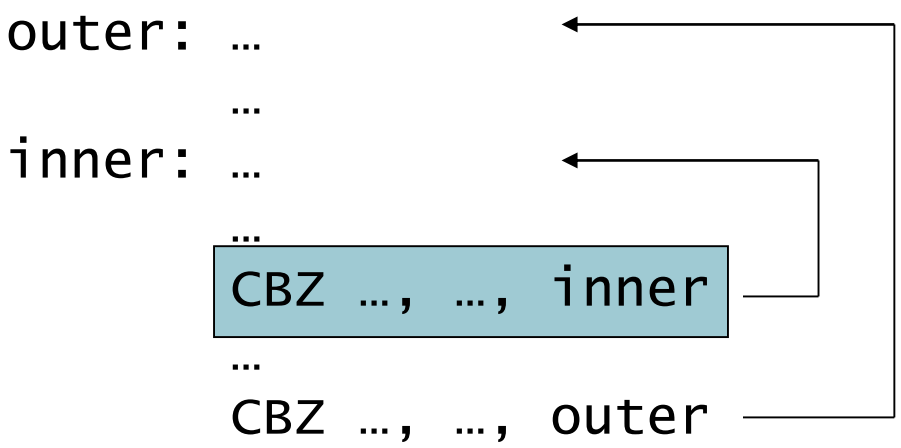


Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

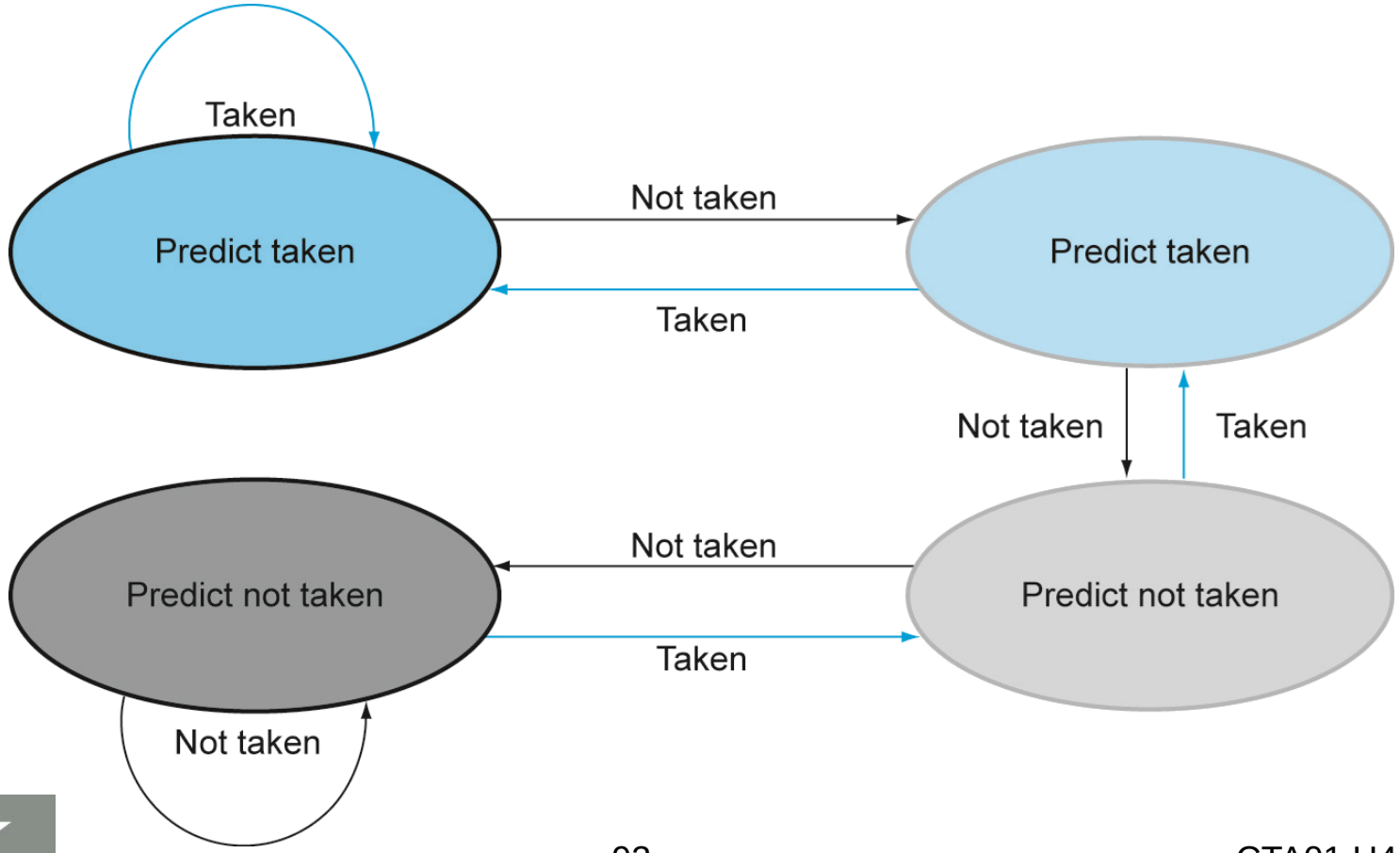
- Inner loop branches mispredicted twice!



  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control