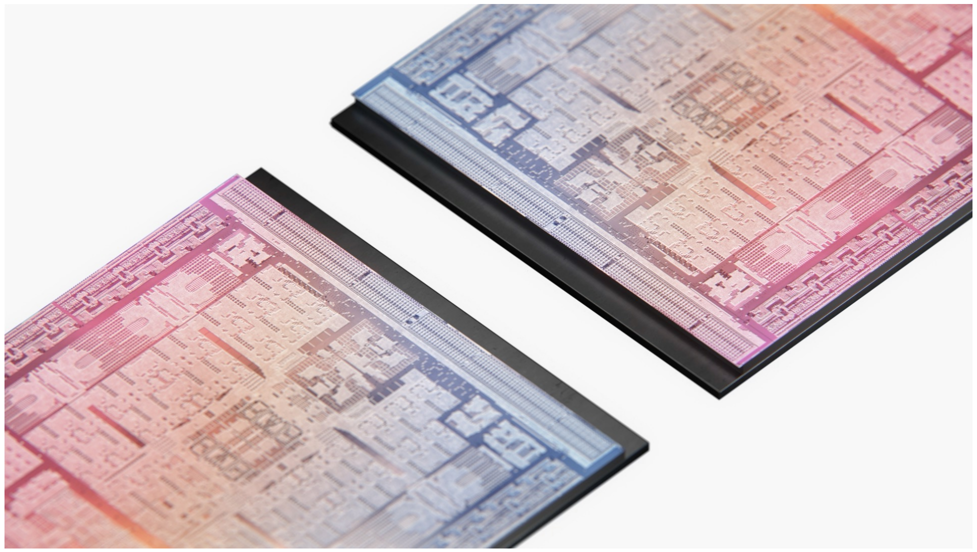
Jae W. Lee (jaewlee@snu.ac.kr), SNU Computer Science and Engineering

***Slide credits**: [CS:APP3e] slides from CMU; [COD:RV2e] slides from Elsevier Inc.*

The Processor (2)

Lecture 5
October 4th, 2023



CompArch Today #2: [Apple introduces M2 Ultra] (06/05/2023)

https://www.apple.com/newsroom/2023/06/apple-introduces-m2-ultra/

M2 Ultra is built using a second-generation 5-nanometer process and uses Apple's UltraFusion technology to connect the die of two M2 Max chips. UltraFusion uses a silicon interposer that connects the dies with more than 10,000 signals, providing over 2.5TB/s of low-latency interprocessor bandwidth. UltraFusion’s architecture enables M2 Ultra to appear as a single chip to software. It has 134 billion transistors, 20 billion more than the M1 Ultra. It also has a unified memory architecture that supports up to 192GB of memory capacity and 800GB/s of memory bandwidth. The M2 Ultra features a more powerful 24-core CPU (16 performance + 8 efficiency cores) that's 20 percent faster than the M1 Ultra, a larger GPU that's up to 30 percent faster, and a Neural Engine (with 31.6 trillion operations per second (TOPS)) that's up to 40 percent faster.

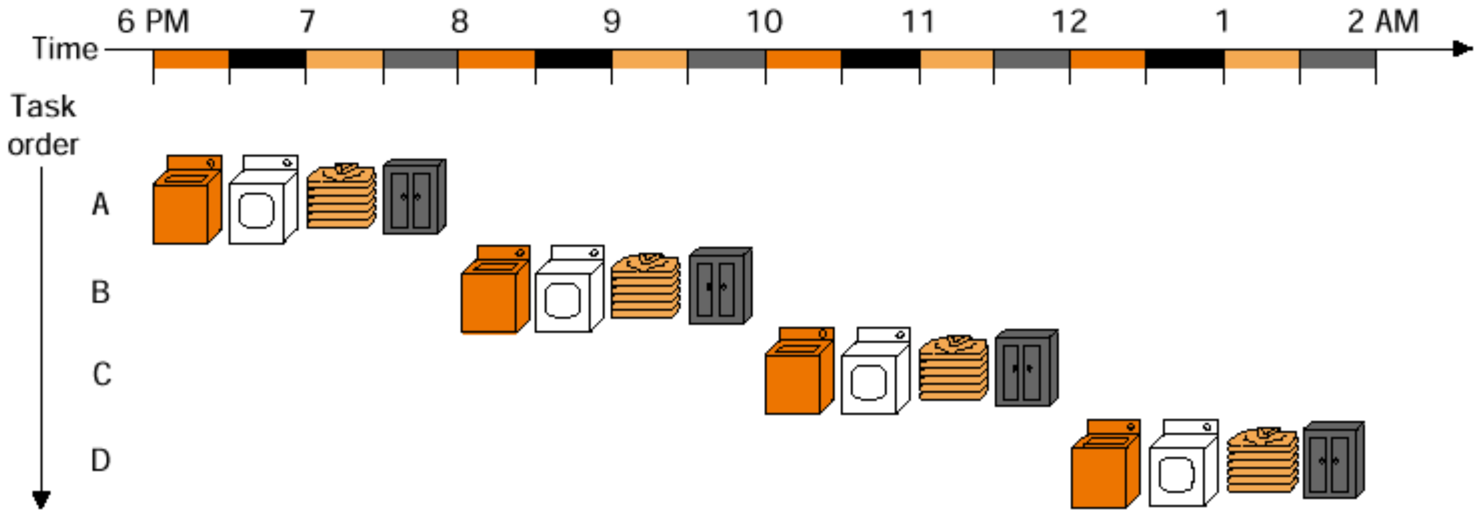
Outline

Textbook: P&H 4.5-4.6

- **An Overview of Pipelining**
- Pipelined Datapath and Control

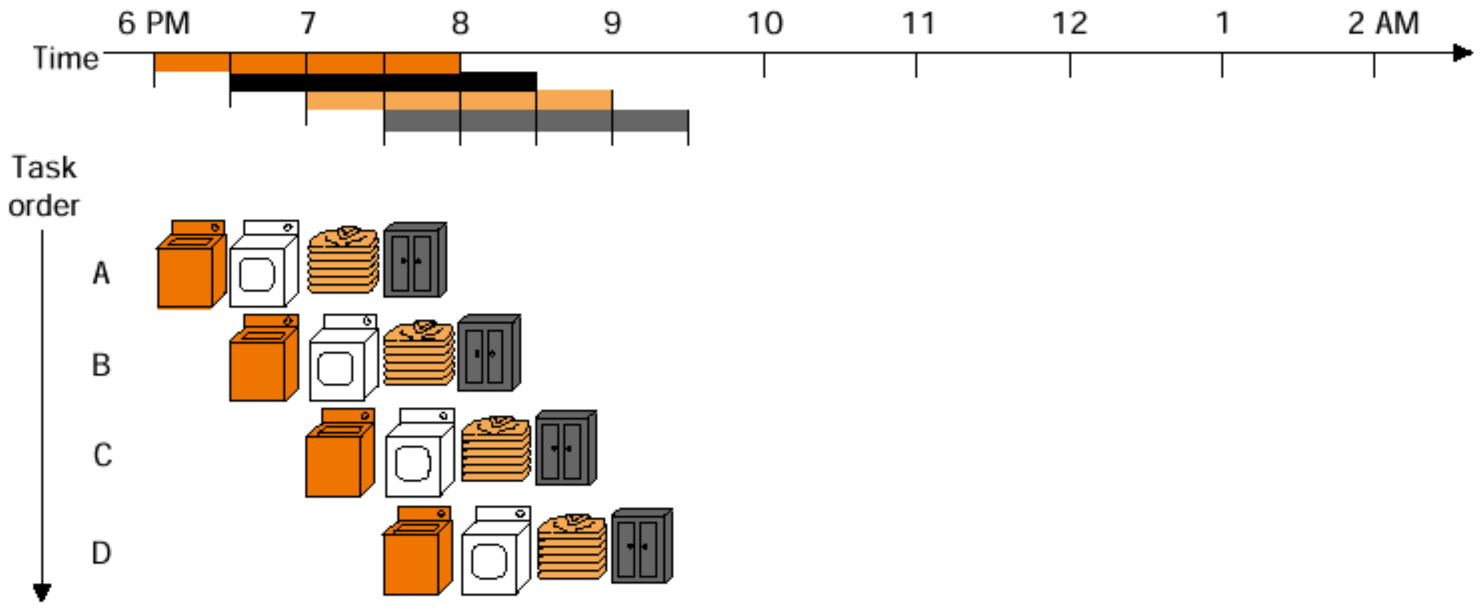
You Already Know Pipelining: Laundry Example

- **Sequential Processing: Wash-Dry-Fold-Store**

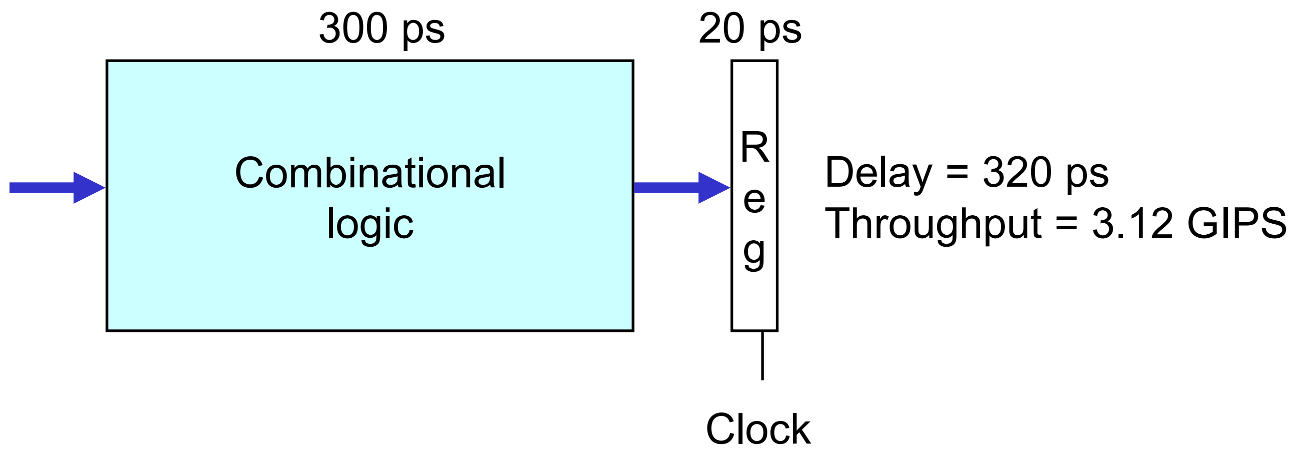


You Already Know Pipelining: Laundry Example

- Pipelined Processing

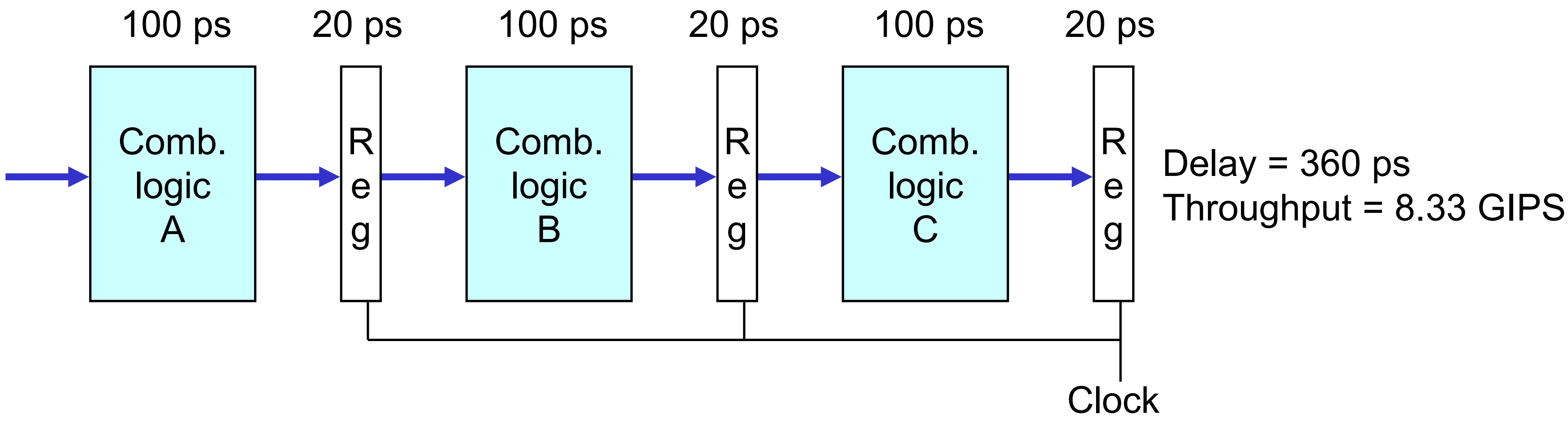


Pipelining for Computation



- **System**
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

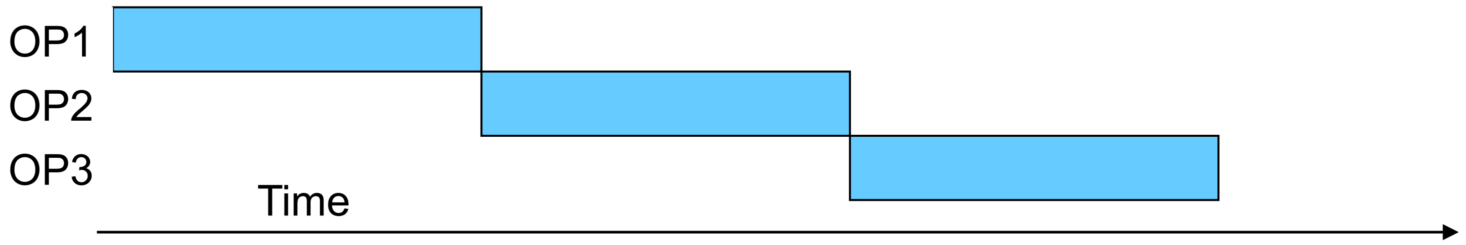
Pipelining for Computation



- **3-Way Pipelined Version**
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

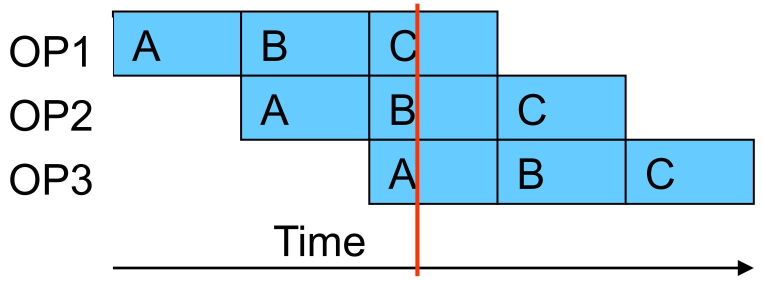
Pipelining for Computation: Pipeline Diagrams

- **Unpipelined**



  - Cannot start new operation until previous one completes

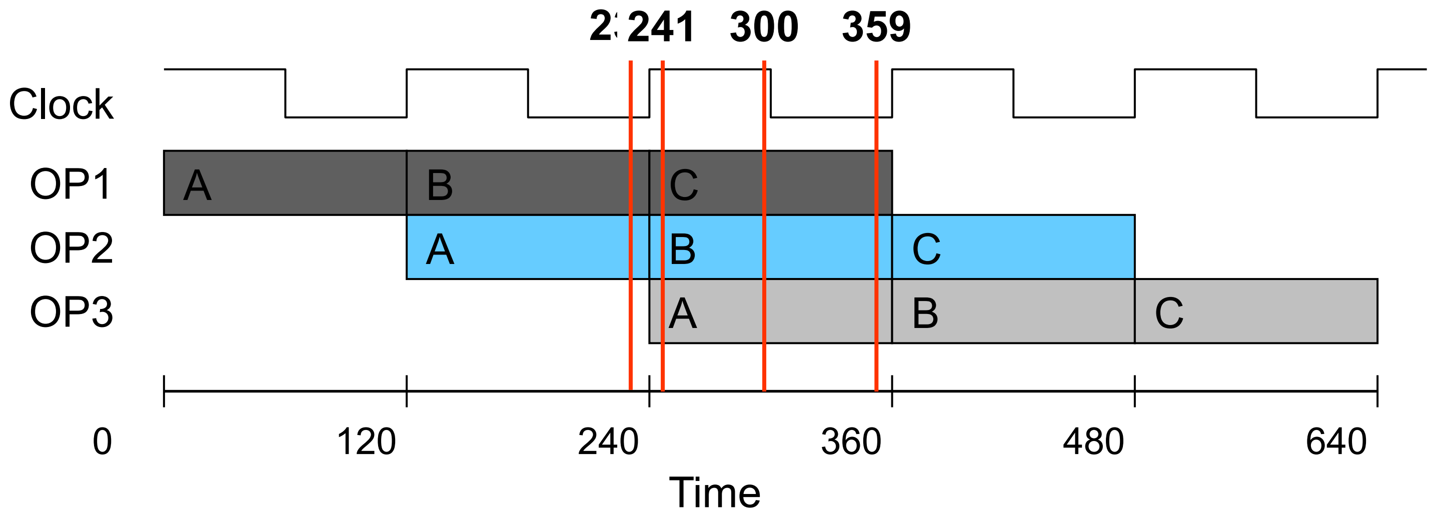
- **3-Way Pipelined**

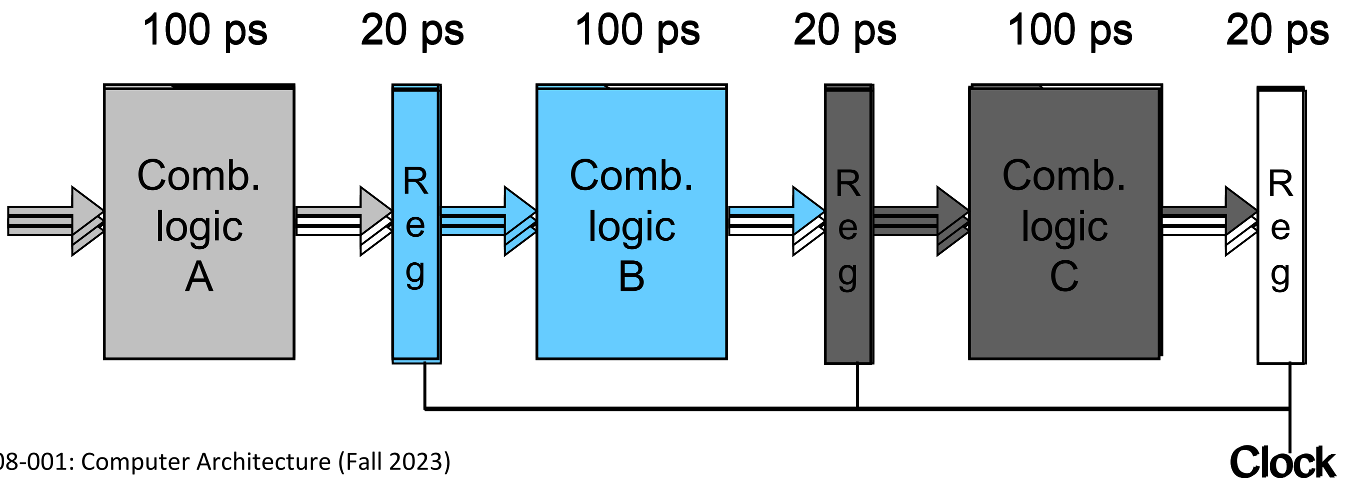


  - Up to 3 operations in process simultaneously

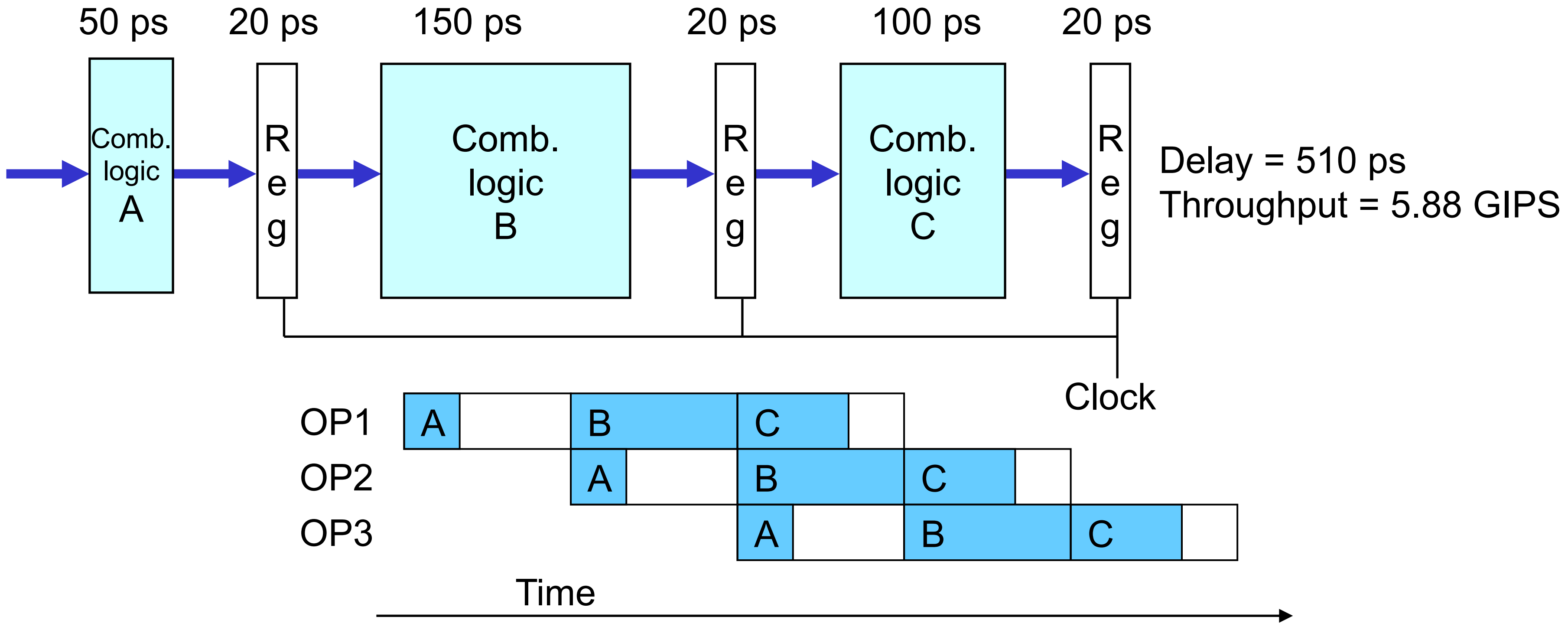
Pipelining for Computation

- 3-Way Pipelined Version: Operation



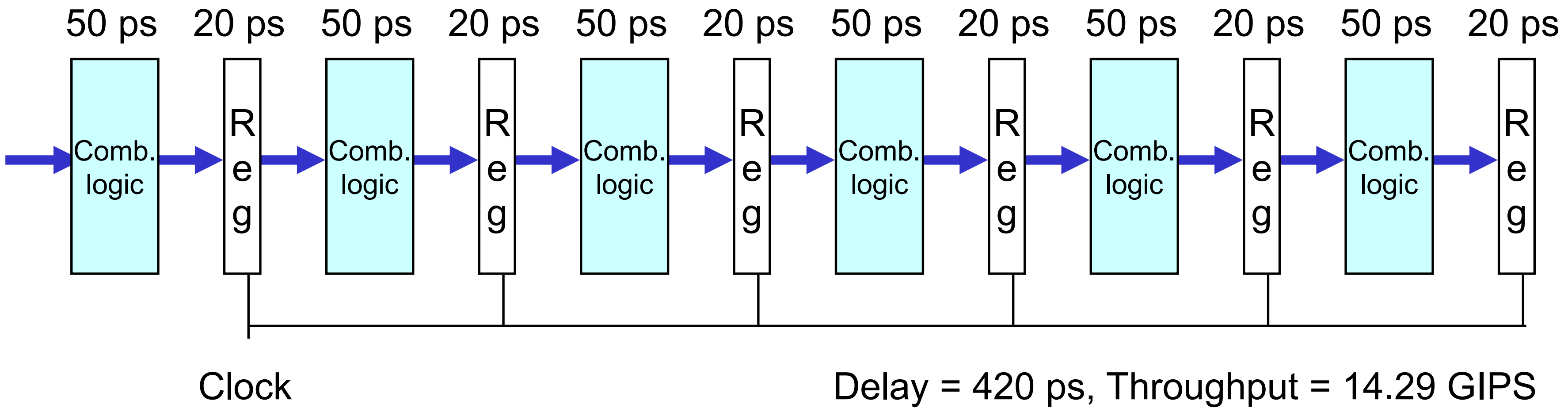


Limitations of Pipelining: Nonuniform Delays



- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations of Pipelining: Register Overhead



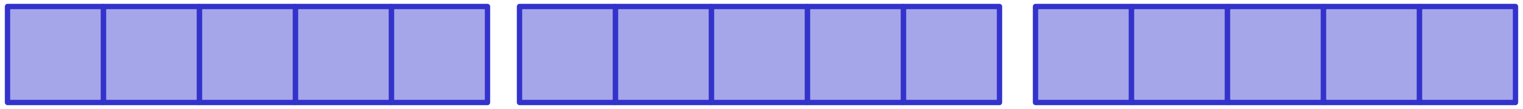
- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

RISC-V Pipeline

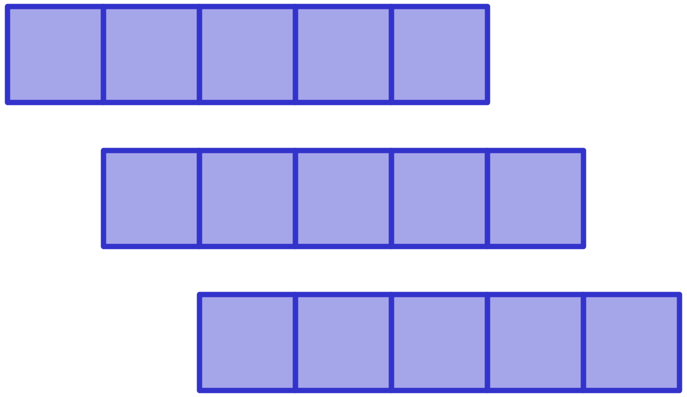
- **Five stages, one step per stage**
  - IF: Instruction fetch from memory
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access memory operand
  - WB: Write result back to register

Pipelined Instruction Execution

- Sequential Execution



- Pipelined Execution



add x21, x12, x0

sub x22, x21, x20

and x16, x22, 0xFFF

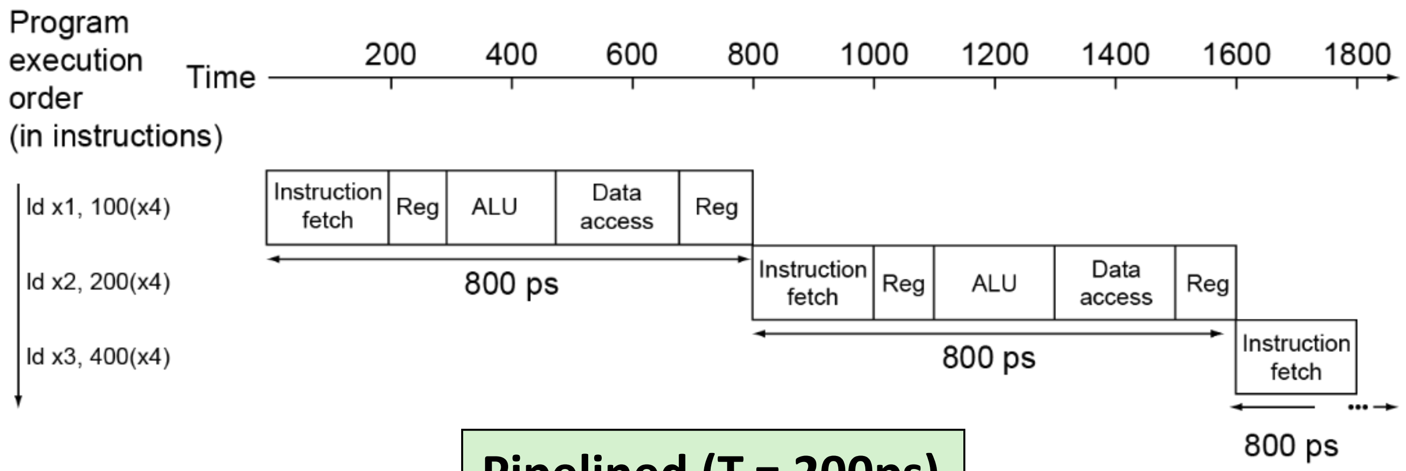
Pipeline Performance

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

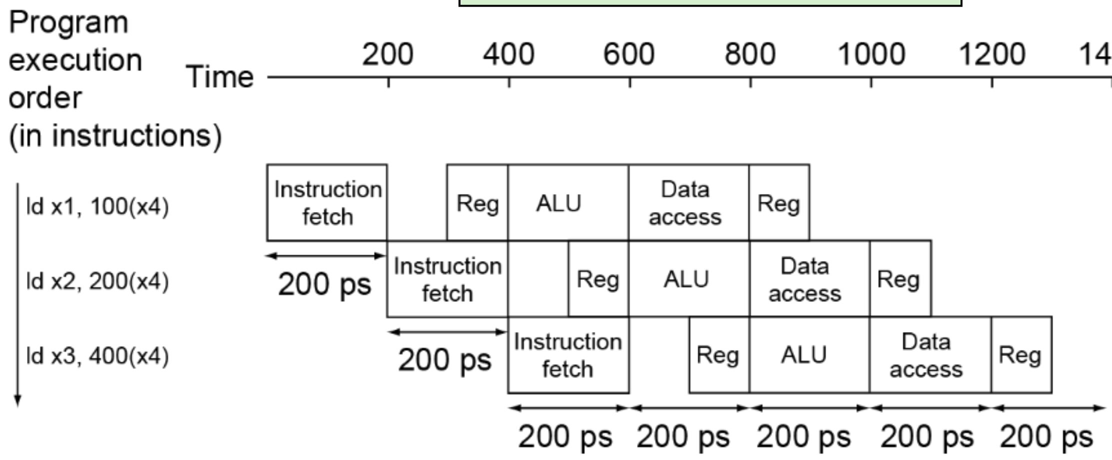
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

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

Pipelining and ISA Design

- **RISC-V ISA designed for pipelining**
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - *cf.* 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

Hazards: Major Hurdles of Pipelining

- **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**
- **Example: in (hypothetical) RISC-V pipeline with a single memory**
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline "bubble"

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Load Instruction	F	D	E	M	W				
Instruction $i+1$		F	D	E	M	W			
Instruction $i+2$			F	D	E	M	W		
Instruction $i+3$				F	D	E	M	W	
Instruction $i+4$					F	D	E	M	W

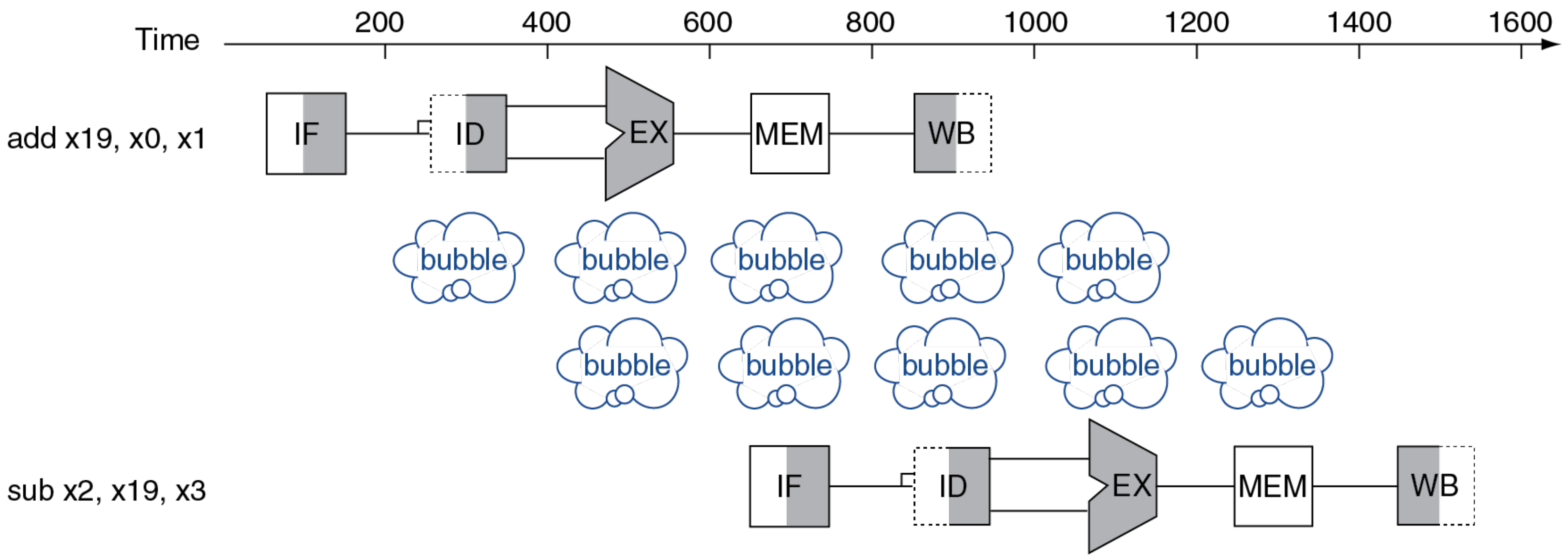
Structure Hazards

- **Solutions to Structural Hazard: Resource Duplication**
  - example
    - Separate I and D caches for memory access conflict
    - Multi-port register file for register file access conflict

Data Hazards

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

```
add   x19, x0, x1
sub   x2, x19, x3
```

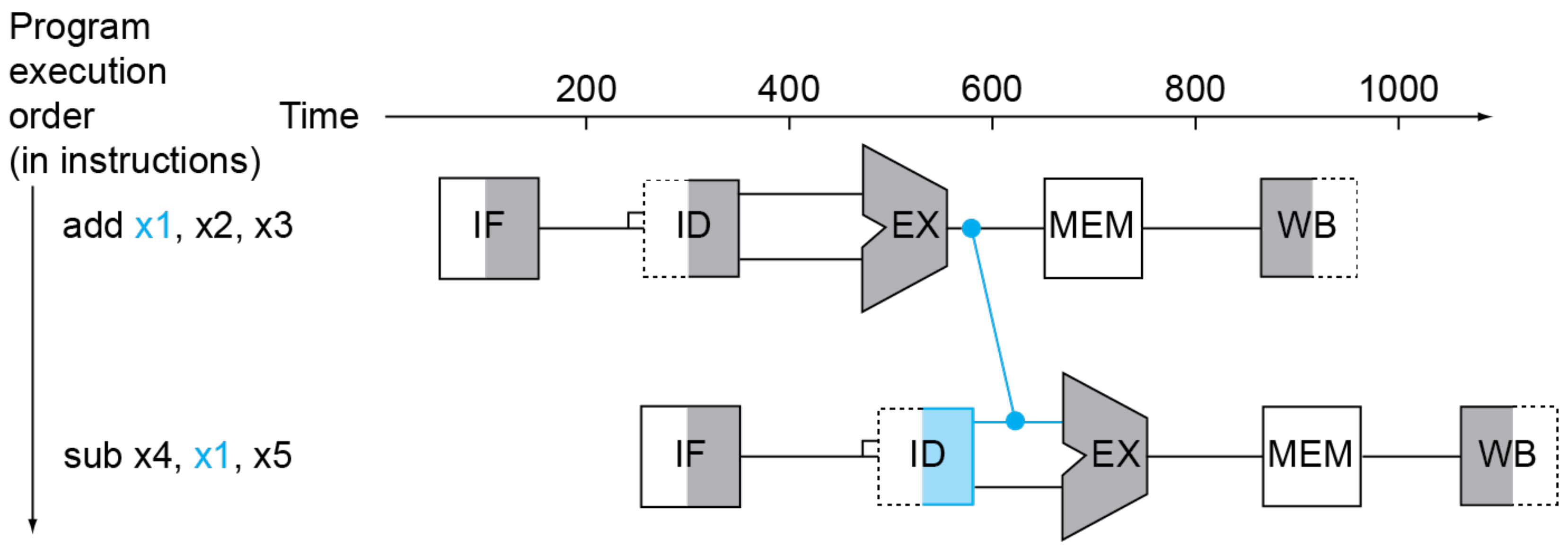


Data Hazards

- **Solutions to Data Hazard**
  1. Freezing the pipeline
  2. (Internal) Forwarding
  3. Compiler scheduling

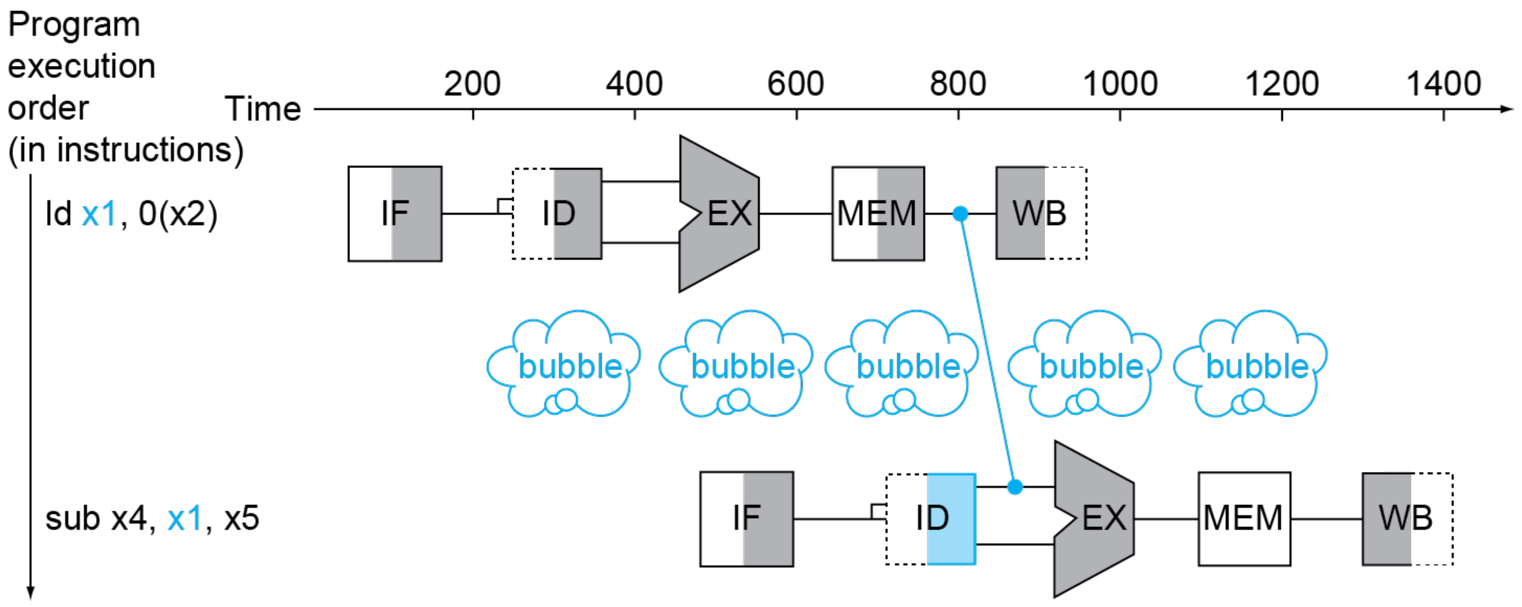
Forwarding (aka Bypassing)

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



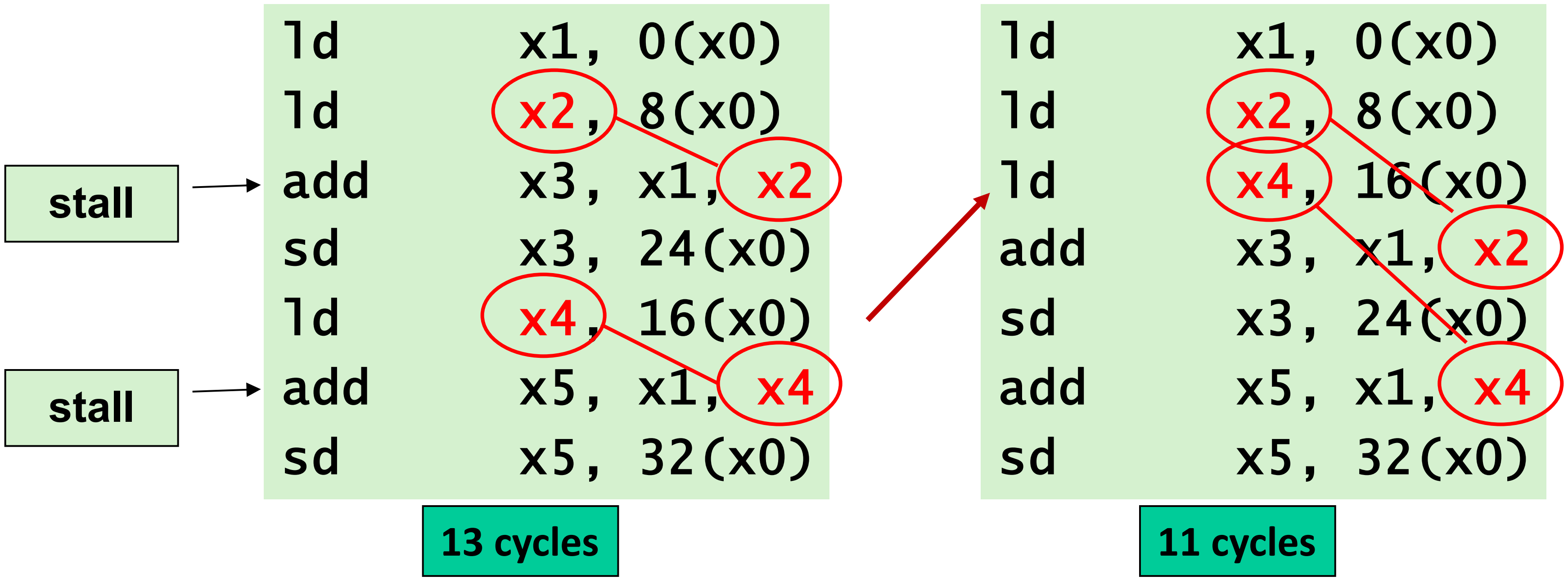
Load-Use Data Hazard

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



Compiler Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`



```
ld      x1, 0(x0)
ld      x2, 8(x0)
add     x3, x1, x2
sd      x3, 24(x0)
ld      x4, 16(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

```
ld      x1, 0(x0)
ld      x2, 8(x0)
ld      x4, 16(x0)
add     x3, x1, x2
sd      x3, 24(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

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

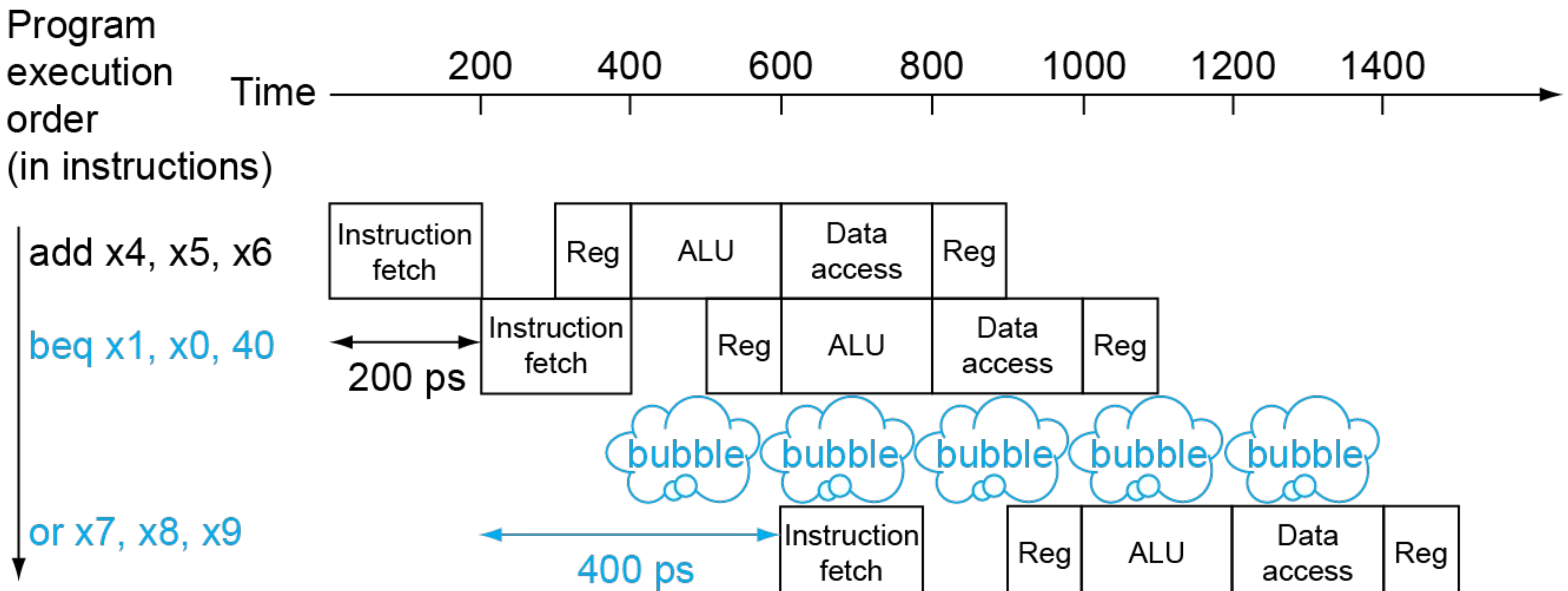
(Example)

Branch Instruction	F	D	E	M	W					
Branch successor		F	*stall*	*stall*	F	D	E	M	W	
Branch successor + 1						F	D	E	M	W
Branch successor + 2							F	D	E	M
Branch successor + 3								F	D	E
Branch successor + 4									F	D
Branch successor + 5										F

For 5-stage pipeline, 3 cycle penalty
15% branch frequency. CPI = 1.45

Control Hazards

- **Solution #1: Stall on branch**
  - Wait until branch outcome determined before fetching next instruction
- **In RISC-V pipeline**
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage



Control Hazards

- **Solution #2: 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 RISC-V 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

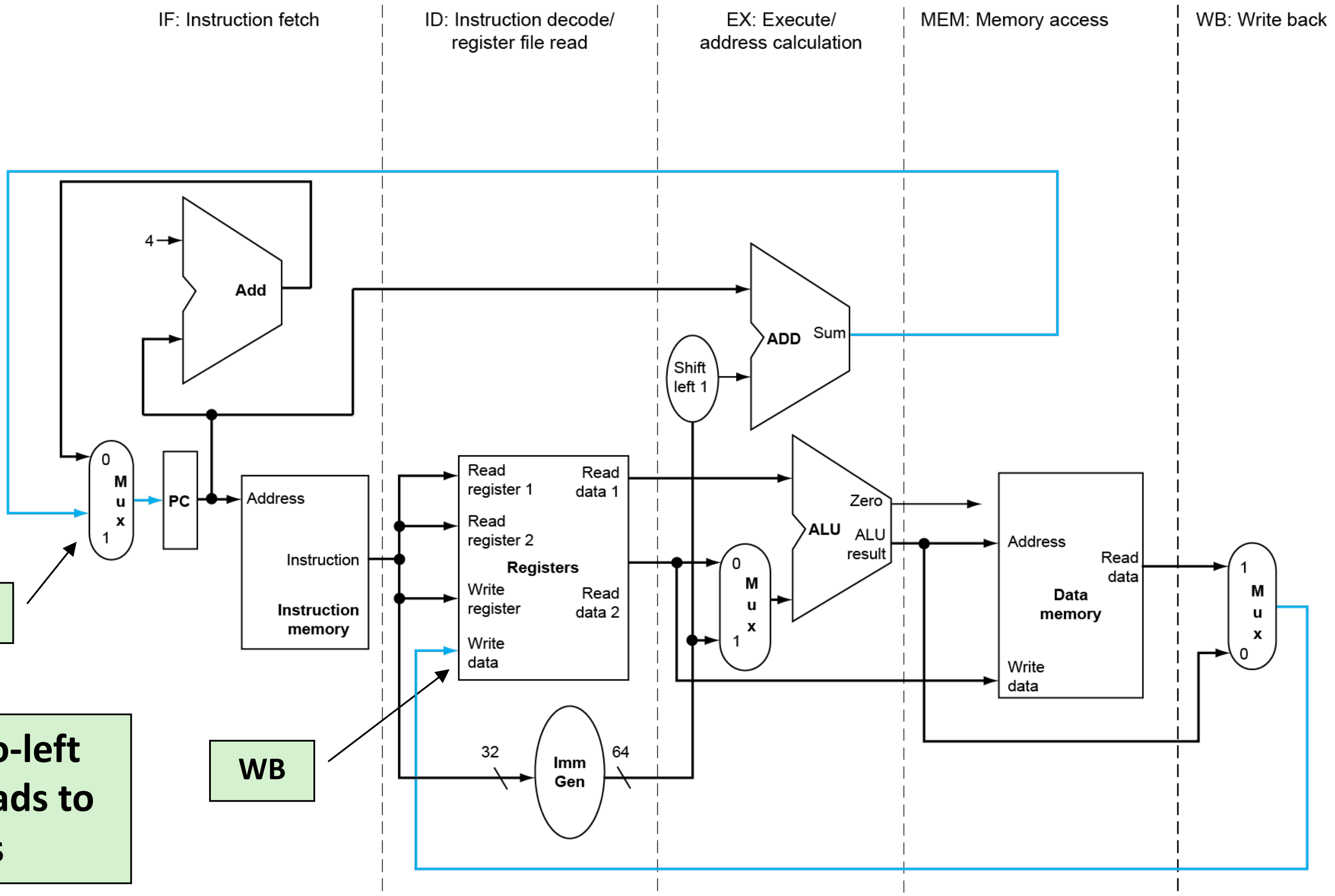
- **Pipelining improves performance by increasing instruction throughput**
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- **Subject to hazards**
  - Structure, data, control
- **Instruction set design affects complexity of pipeline implementation**

Outline

Textbook: P&H 4.5-4.6

- An Overview of Pipelining
- **Pipelined Datapath and Control**

RISC-V Pipelined Datapath



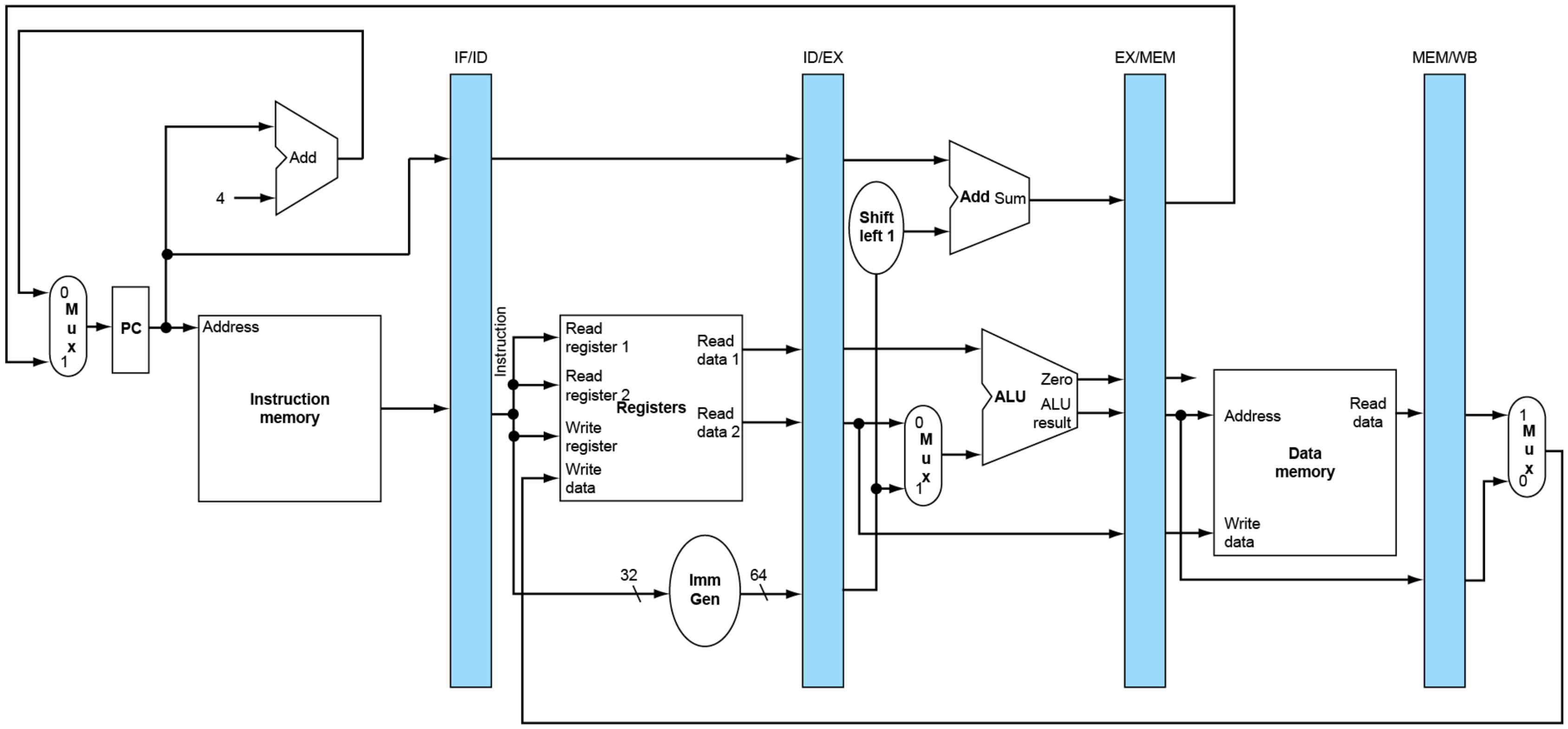
MEM

WB

Right-to-left flow leads to hazards

Pipeline registers

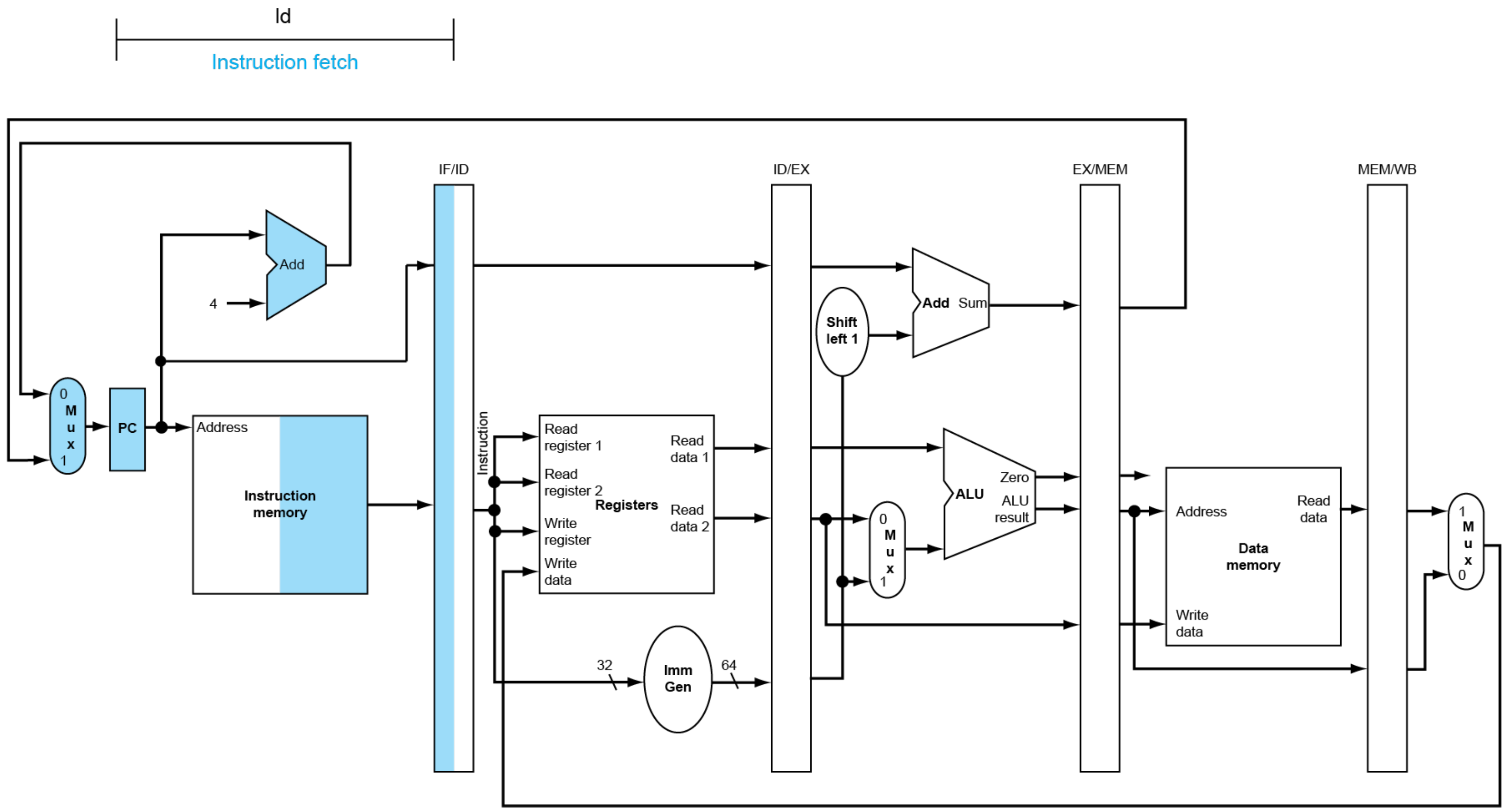
- **Need registers between stages**
  - To hold information produced in previous cycle



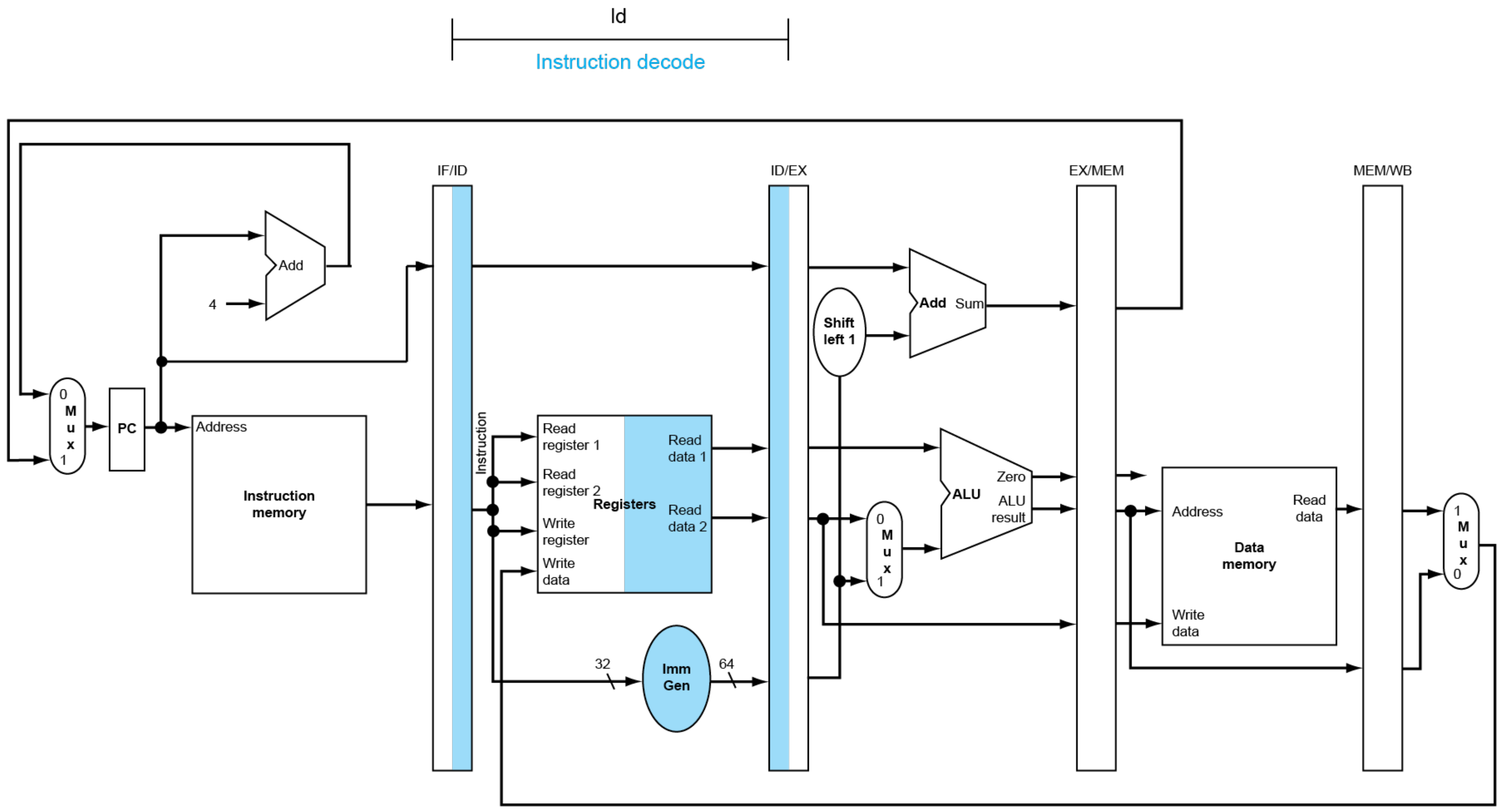
Pipeline Operation

- **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
  - *cf.* "multi-clock-cycle" diagram
    - Graph of operation over time
- **We'll look at "single-clock-cycle" diagrams for load & store**

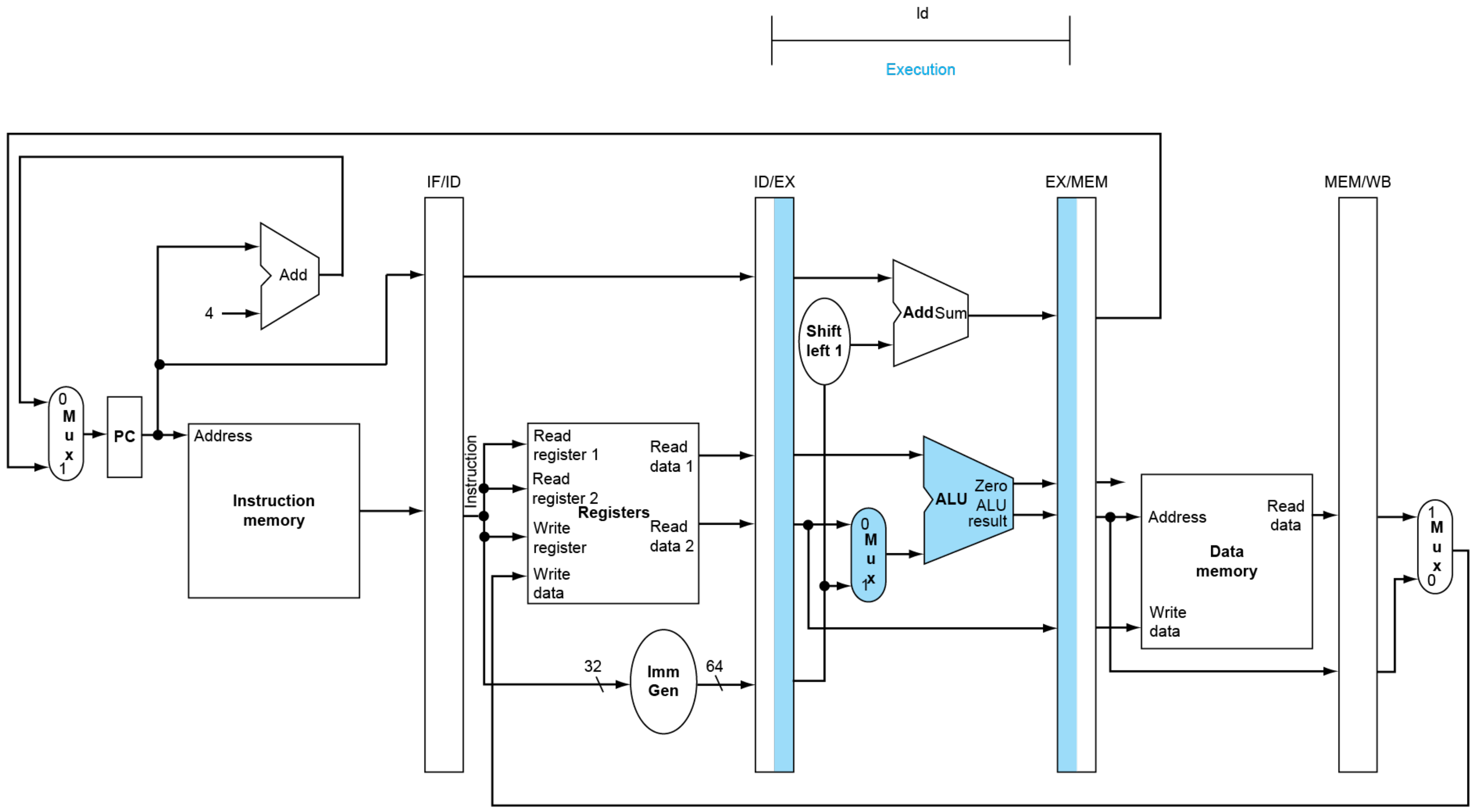
IF for Load, Store, …



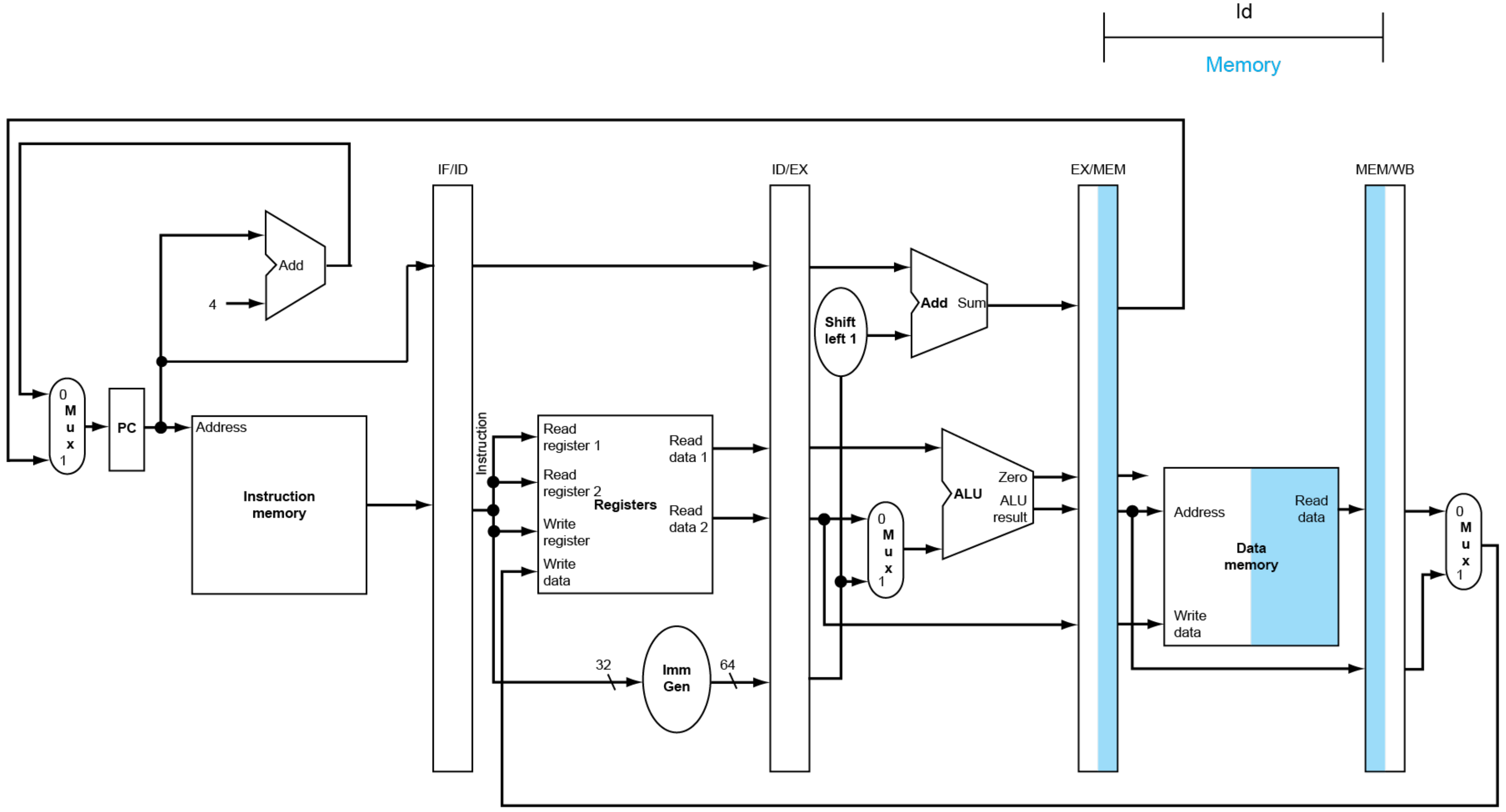
ID for Load, Store, …



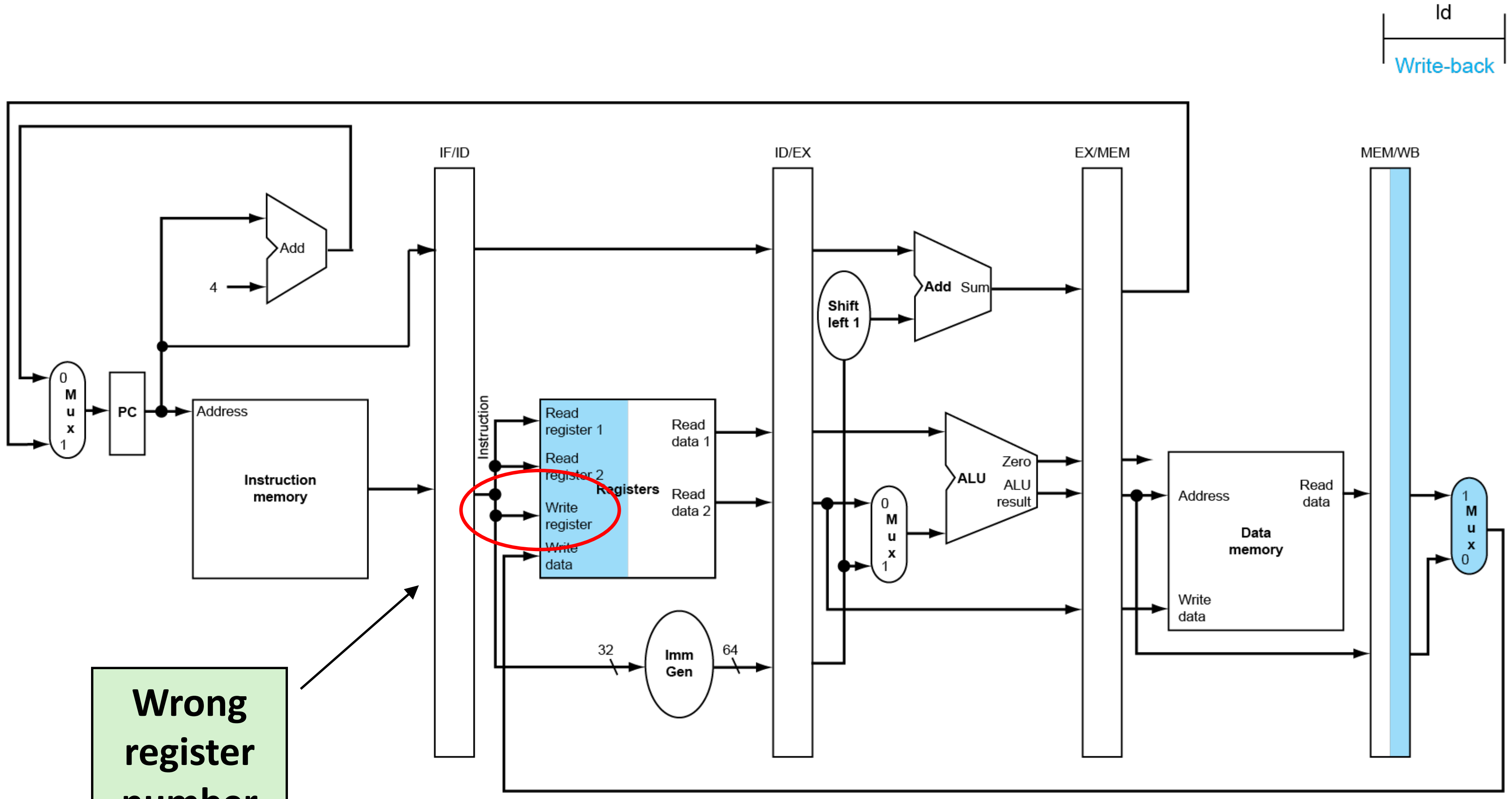
EX for Load


ld
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
Imm Gen
64
Shift left 1
Add Sum
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1 M u x 0

MEM for Load

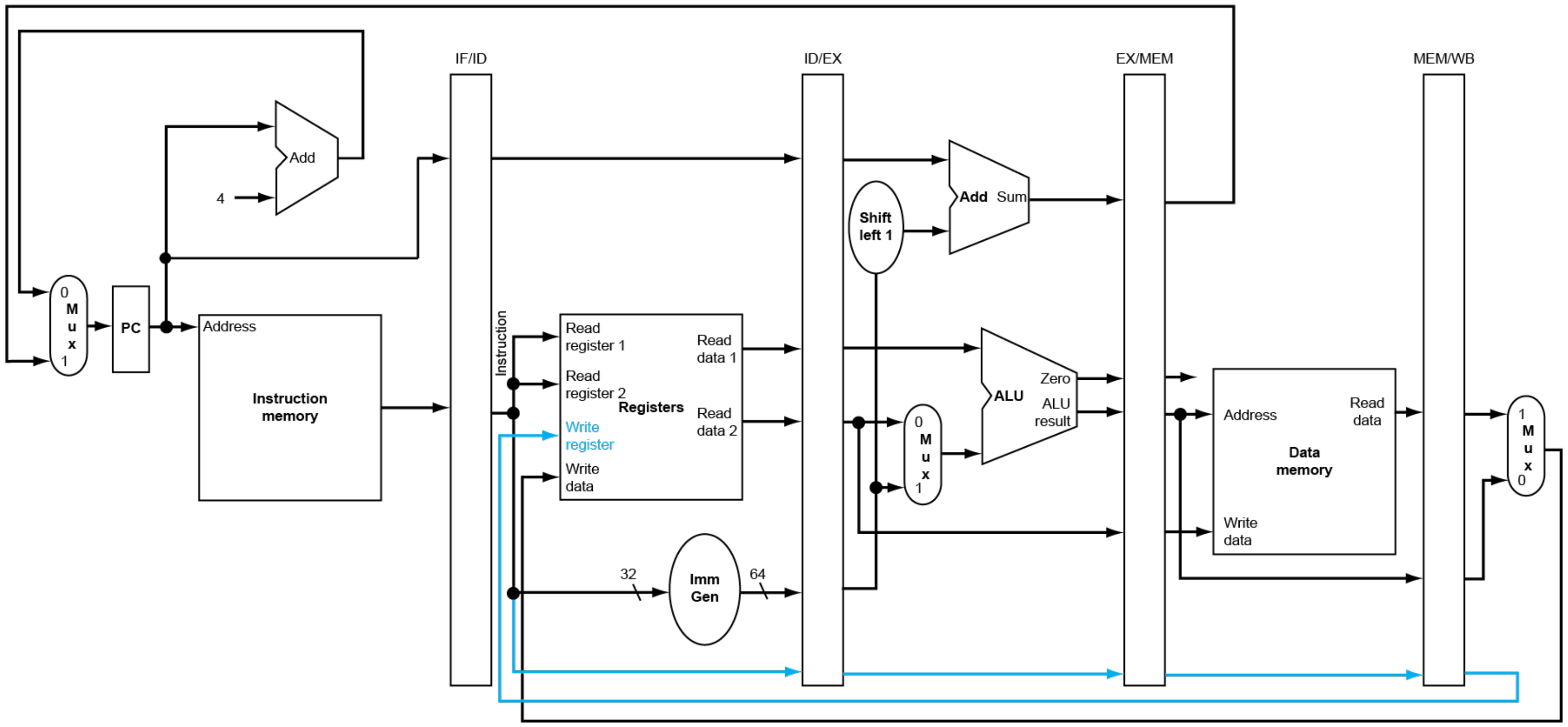

ld
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

WB for Load

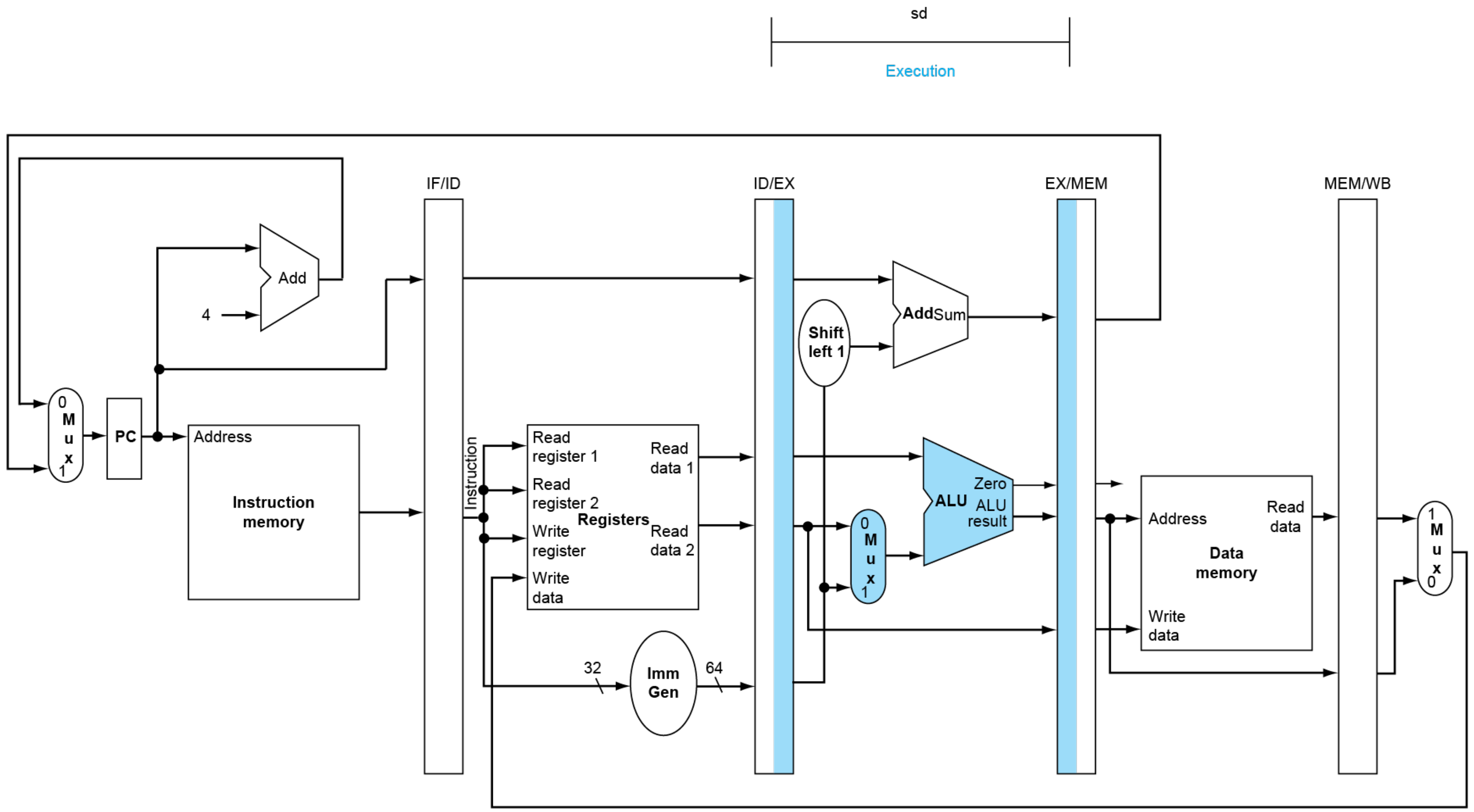


Wrong register number

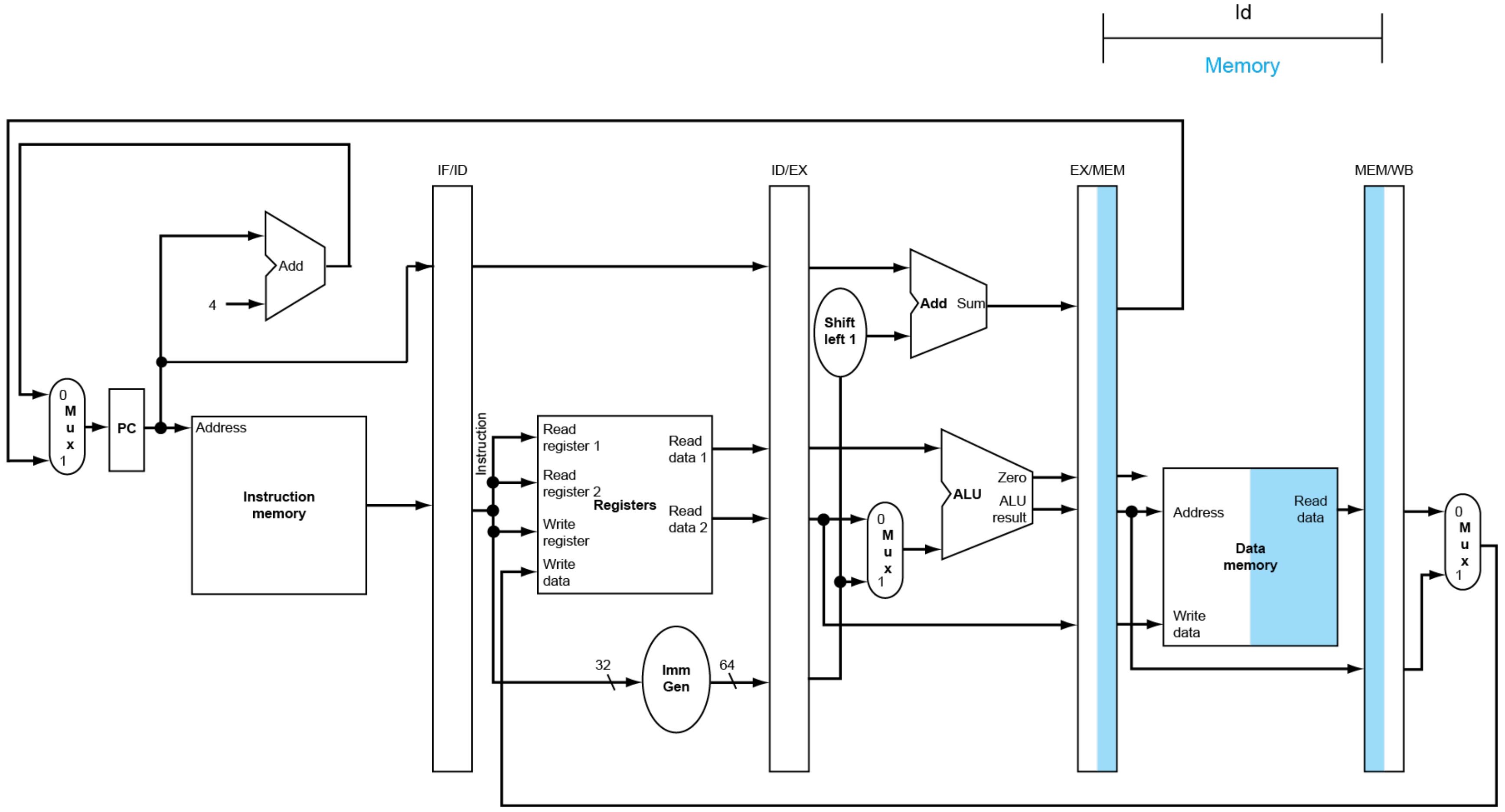
Corrected Datapath for Load



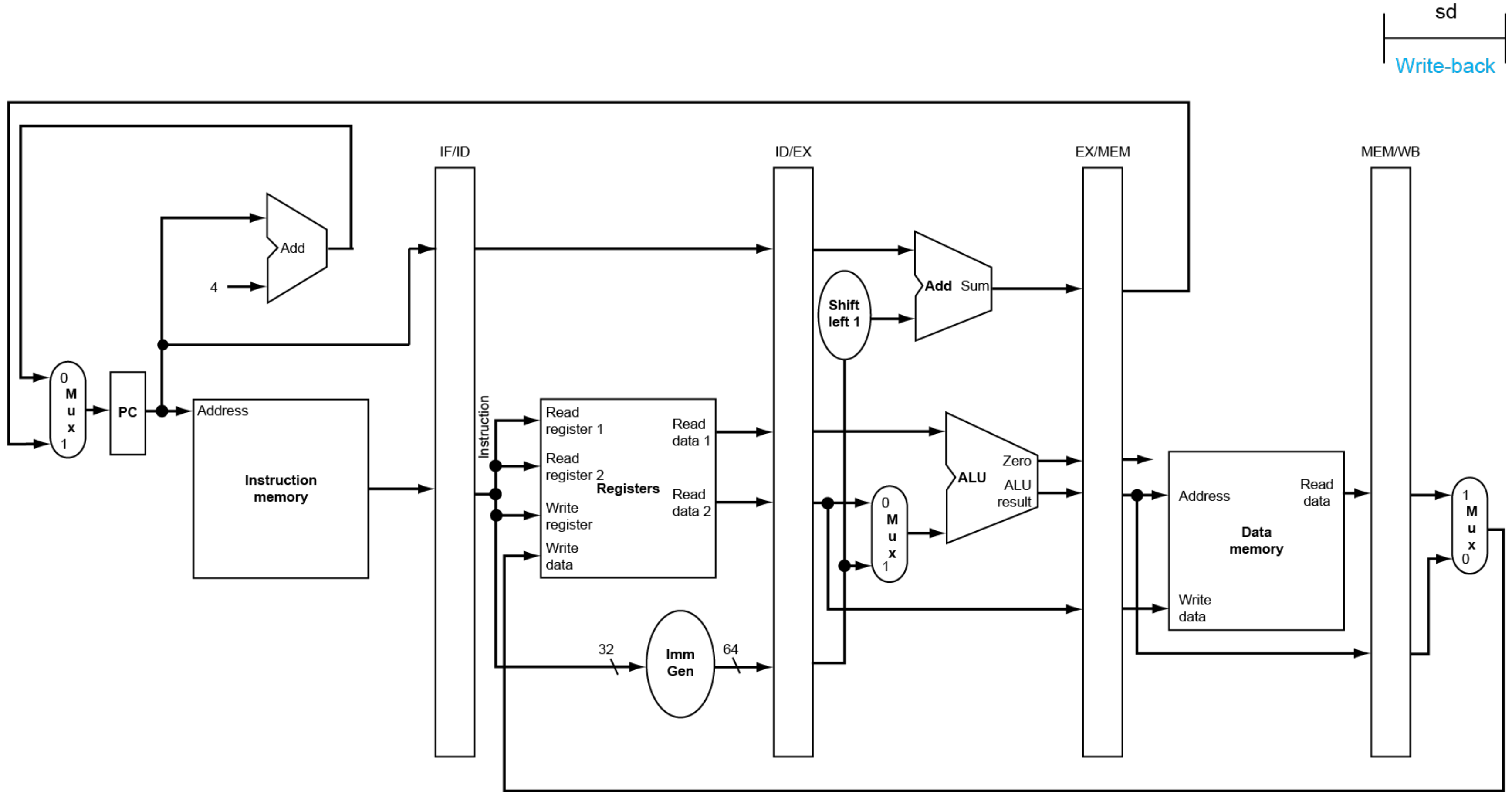
EX for Store


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

MEM for Store

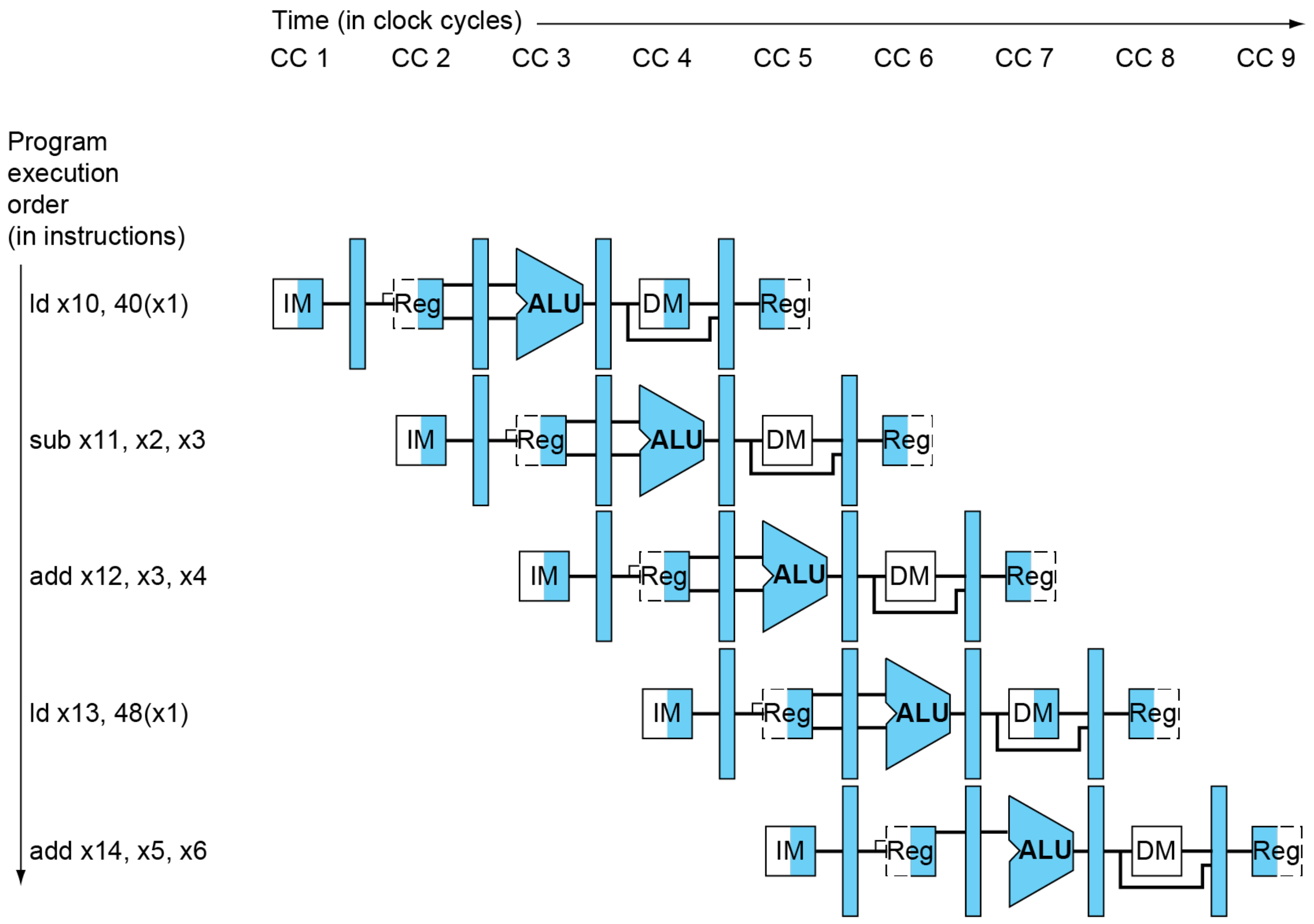


WB for Store


sd
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
Write register
Write data
Registers
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add
Sum
ALU
Zero
ALU result
Address
Data memory
Write data
Read data

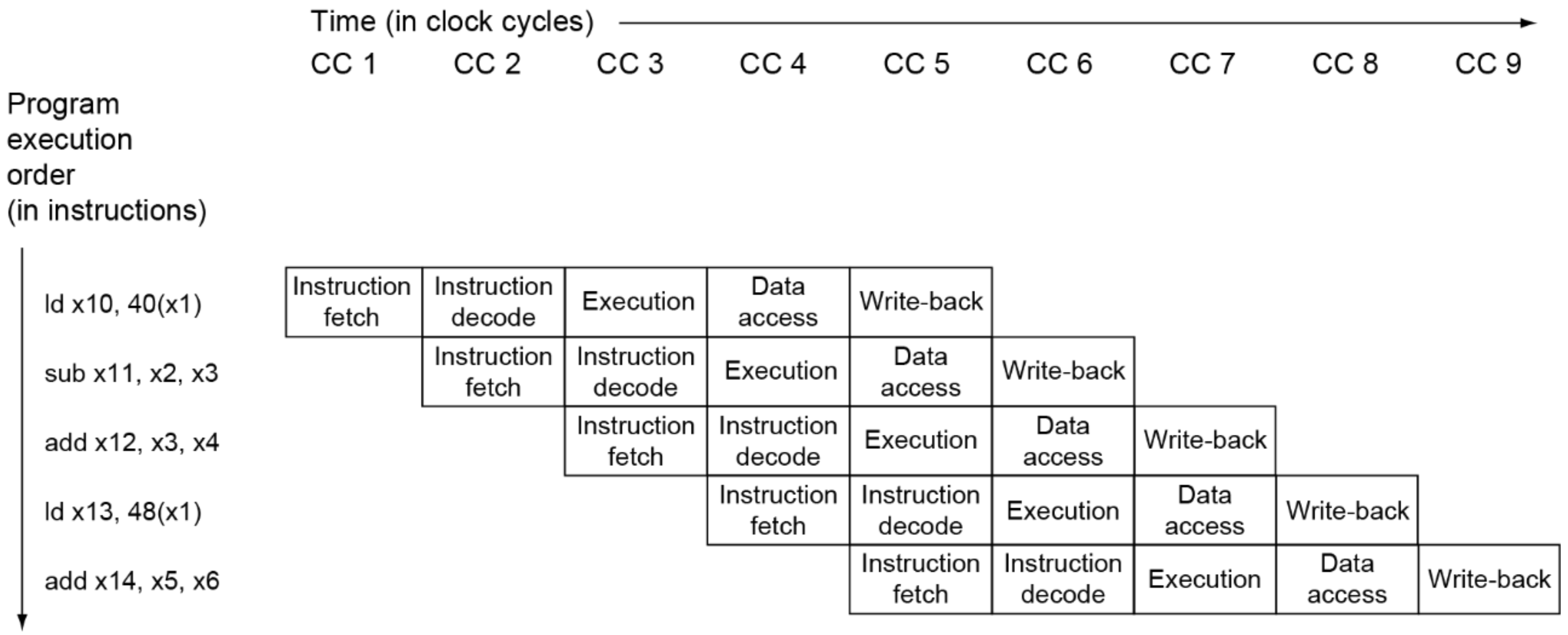
Multi-Cycle Pipeline Diagram

- Form showing resource usage



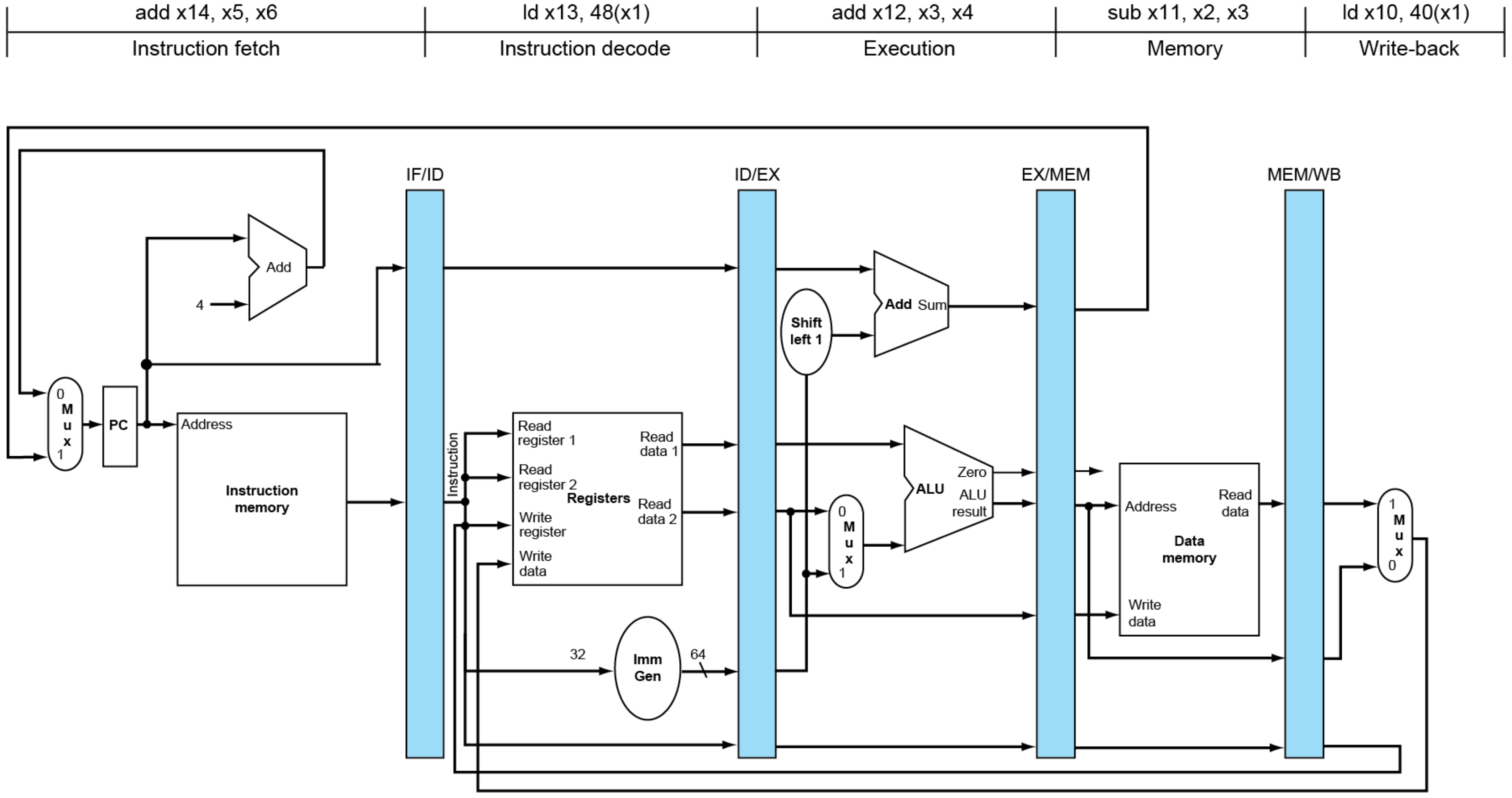
Multi-Cycle Pipeline Diagram

- **Traditional form**

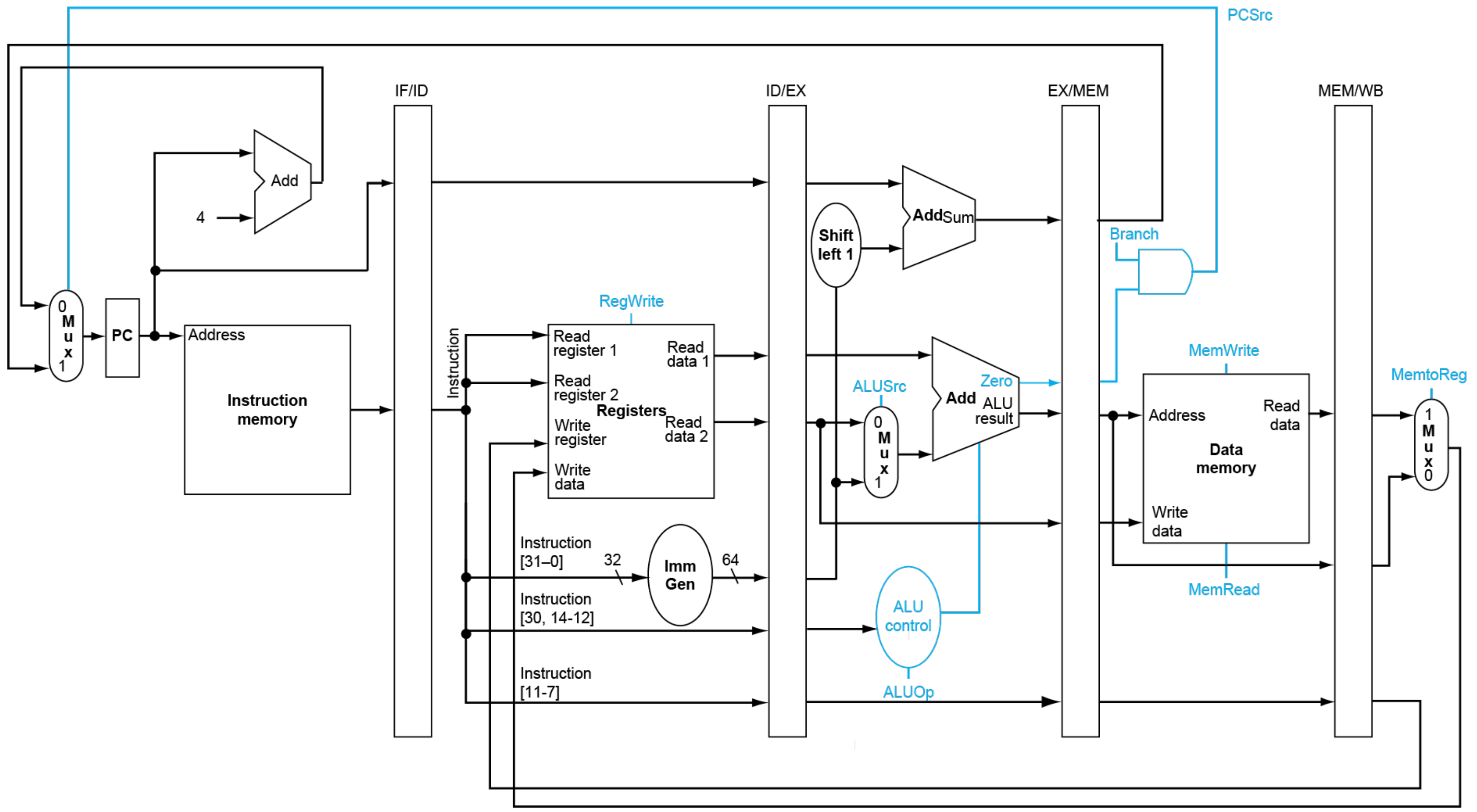


Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

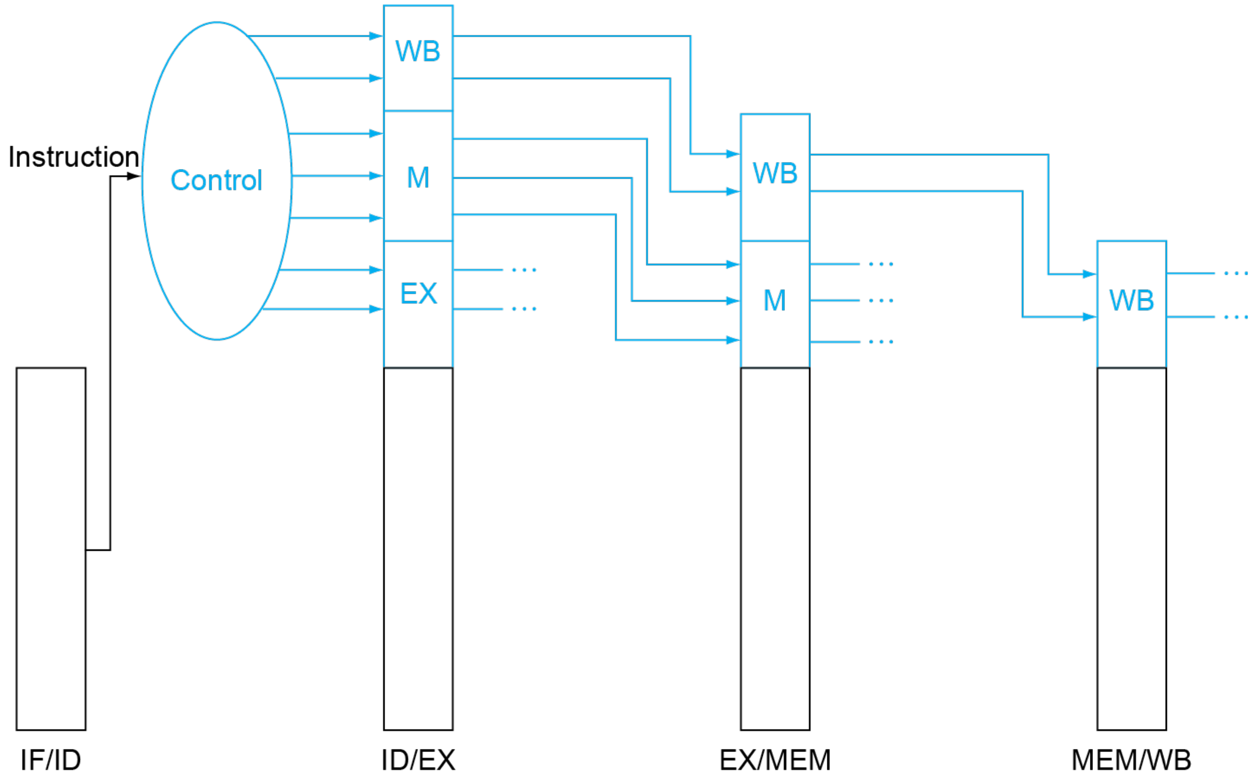


Pipelined Control (Simplified)

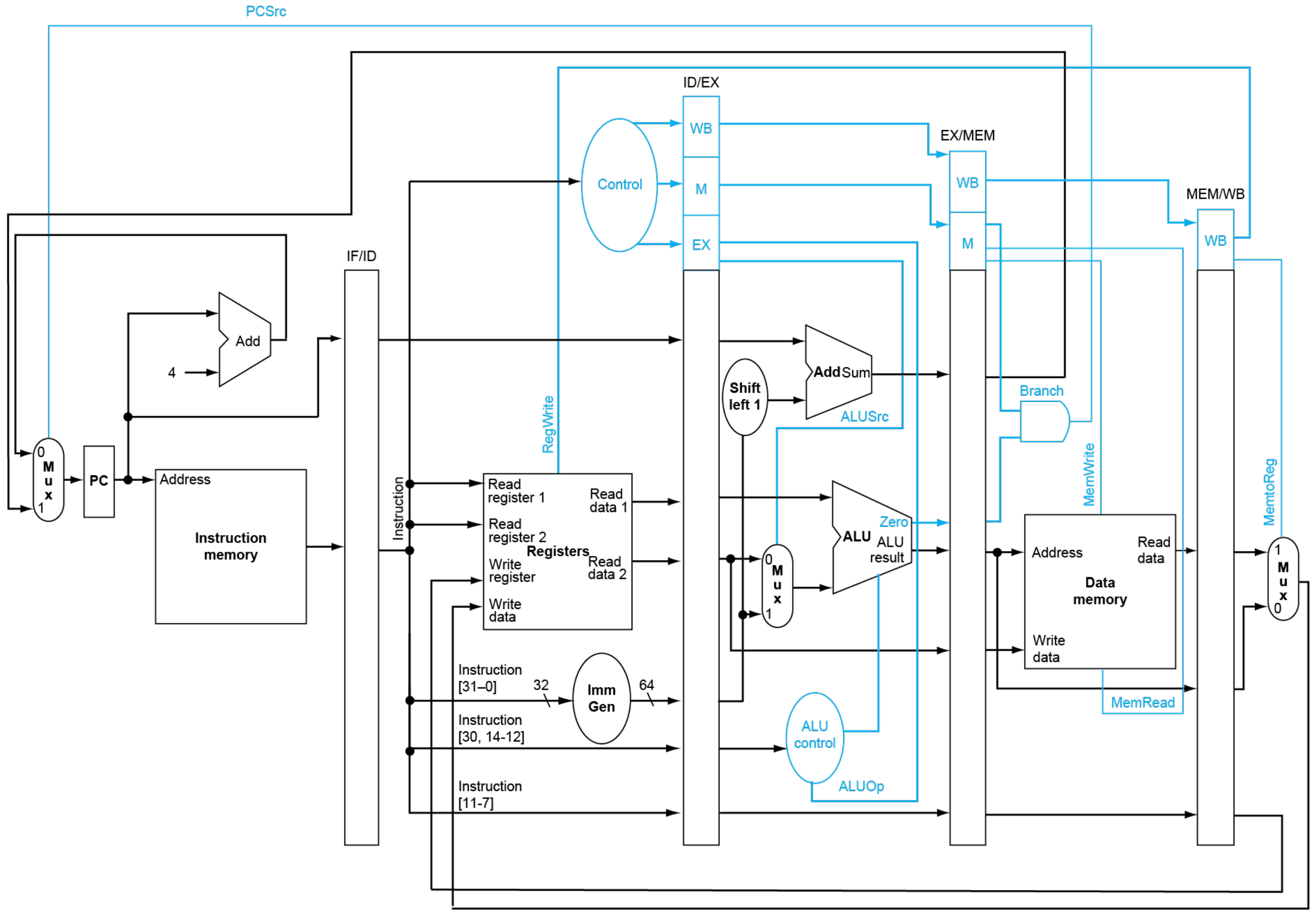


Pipelined Control

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



Pipelined Control


PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
WB
M
WB
Add
4
Add Sum
Shift left 1
ALUSrc
Branch
RegWrite
MemWrite
MemtoReg
Mux
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
MemRead
Instruction [31–0]
32
Imm Gen
64
Instruction [30, 14-12]
ALU control
ALUOp
Instruction [11-7]