

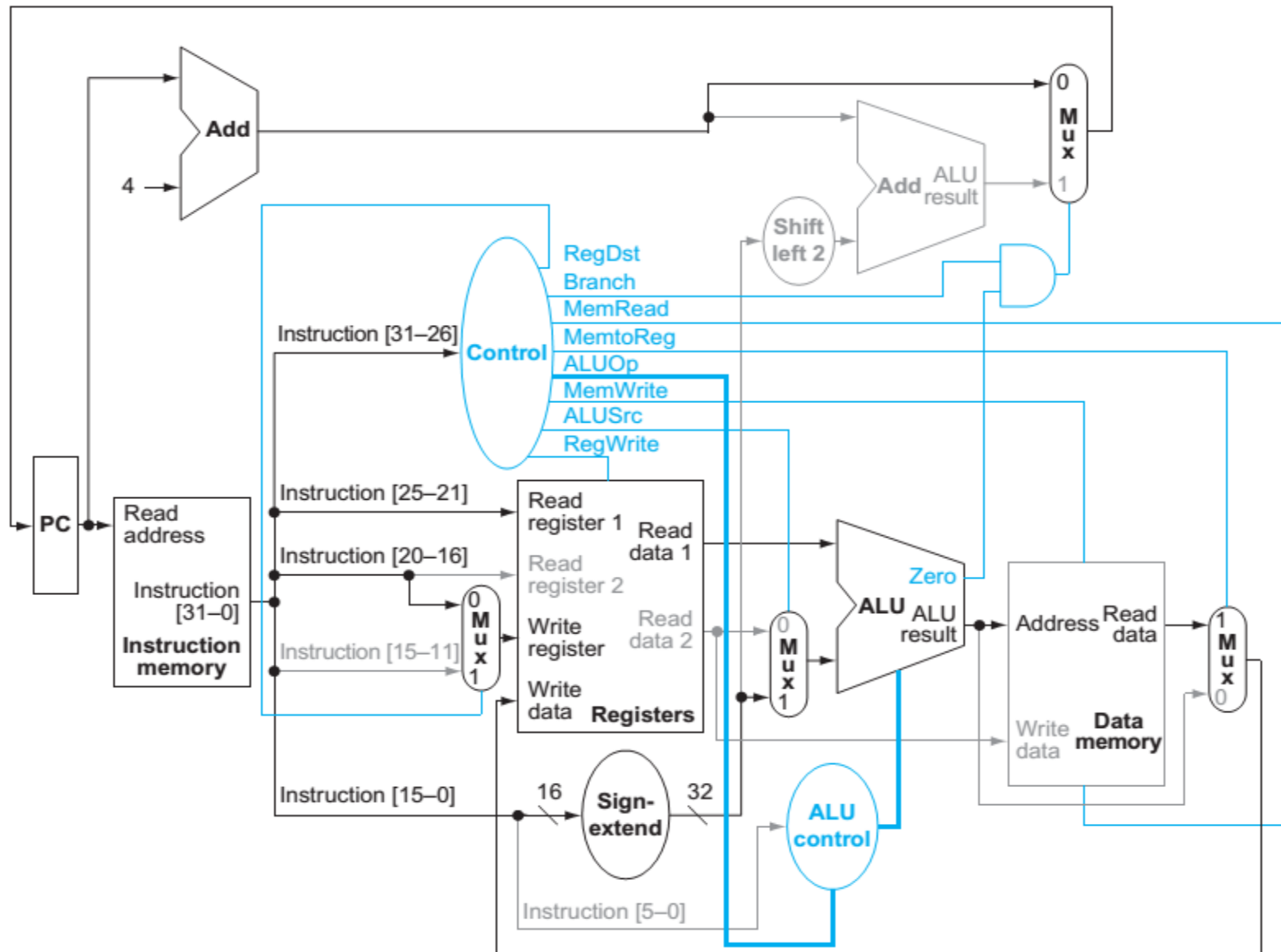
Lecture 9

The Overview of Pipeline

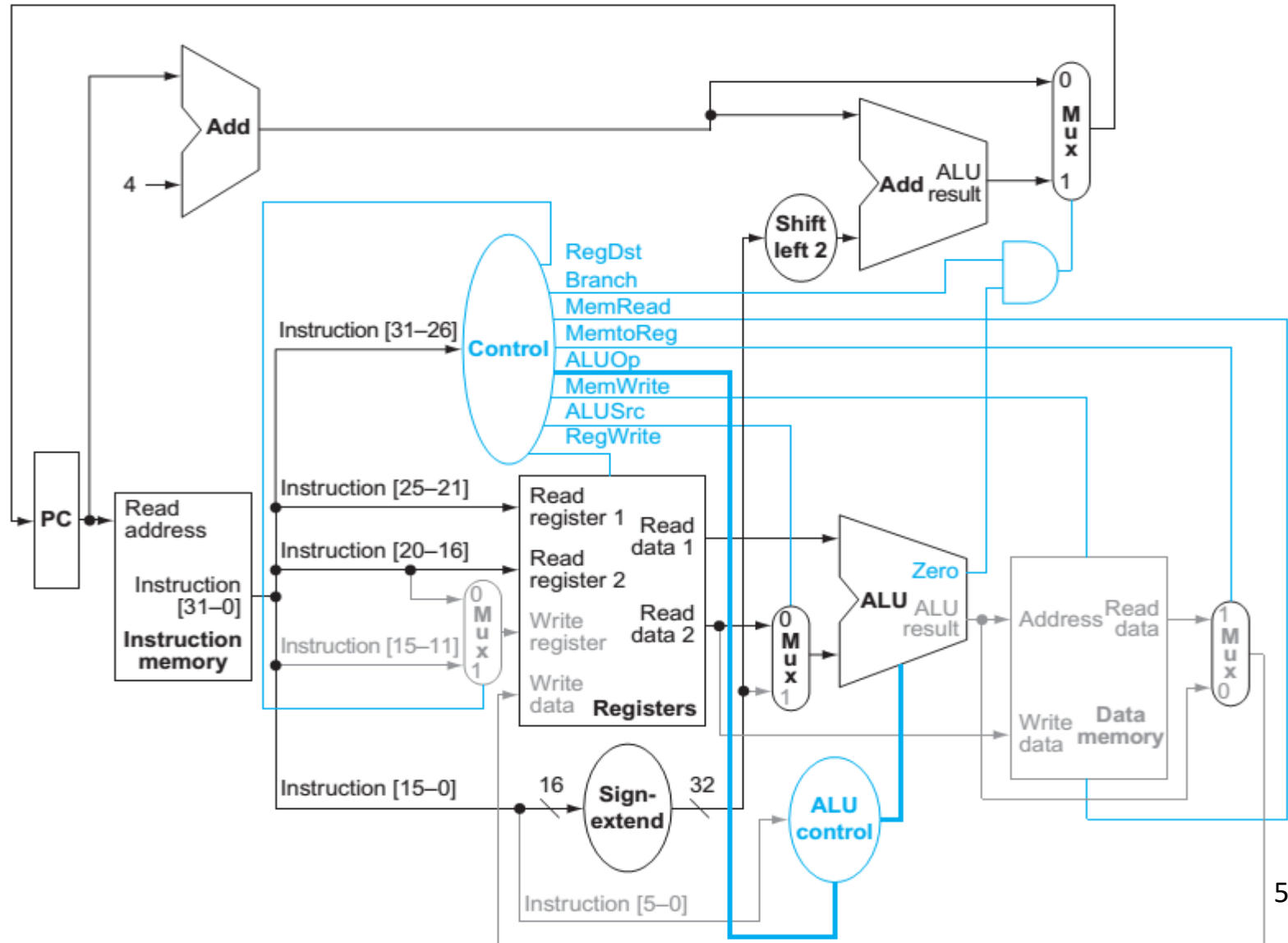
Recap

- We have designed a simple CPU that executes:
 - basic math (add, sub, and, or, slt)
 - memory access (lw and sw)
 - branch and jump instructions (beq and j)
- We will design a more realistic pipelined version of MIPS CPU

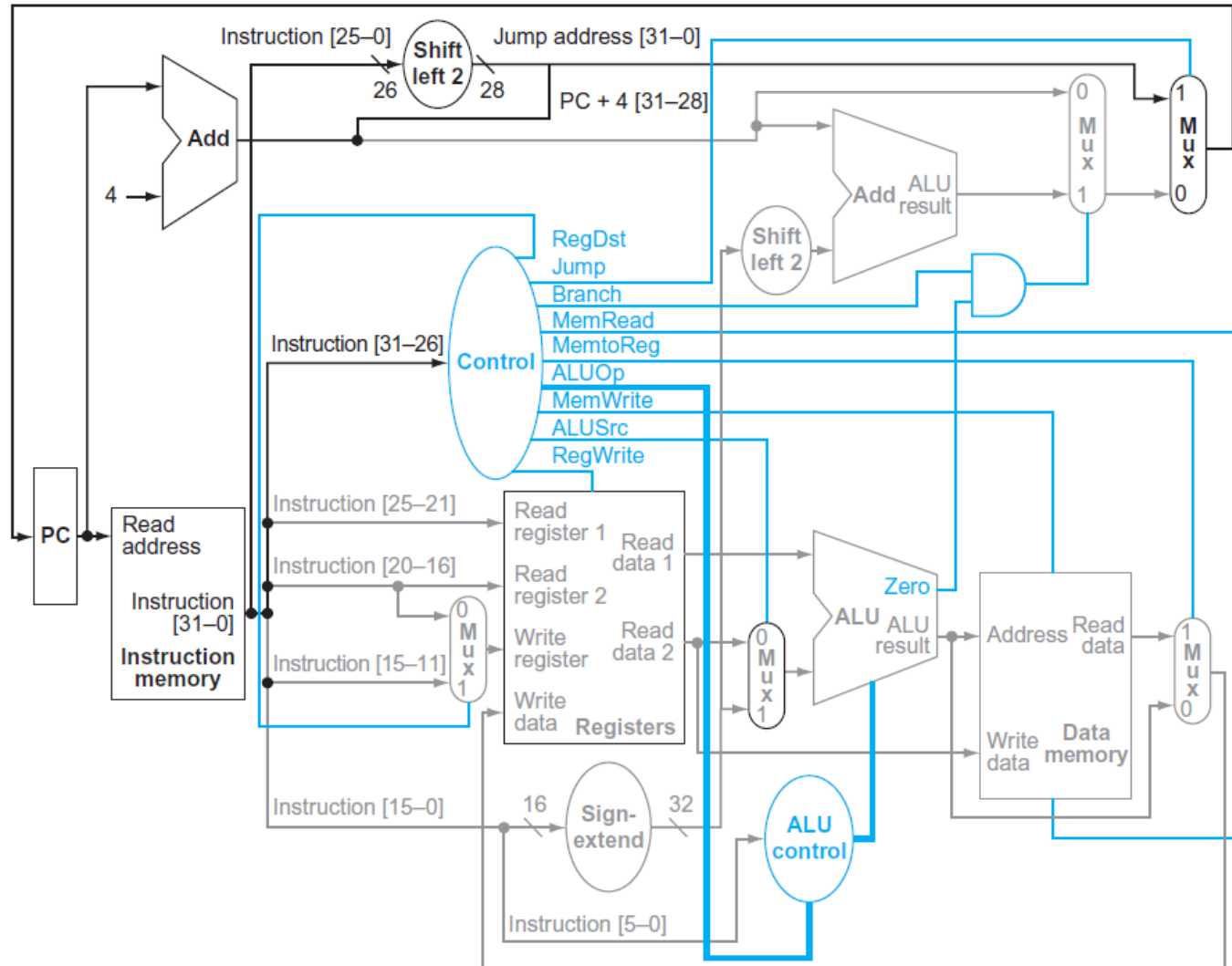
Datapath for a load Instruction



Datapath for a Branch-on-equal Instruction



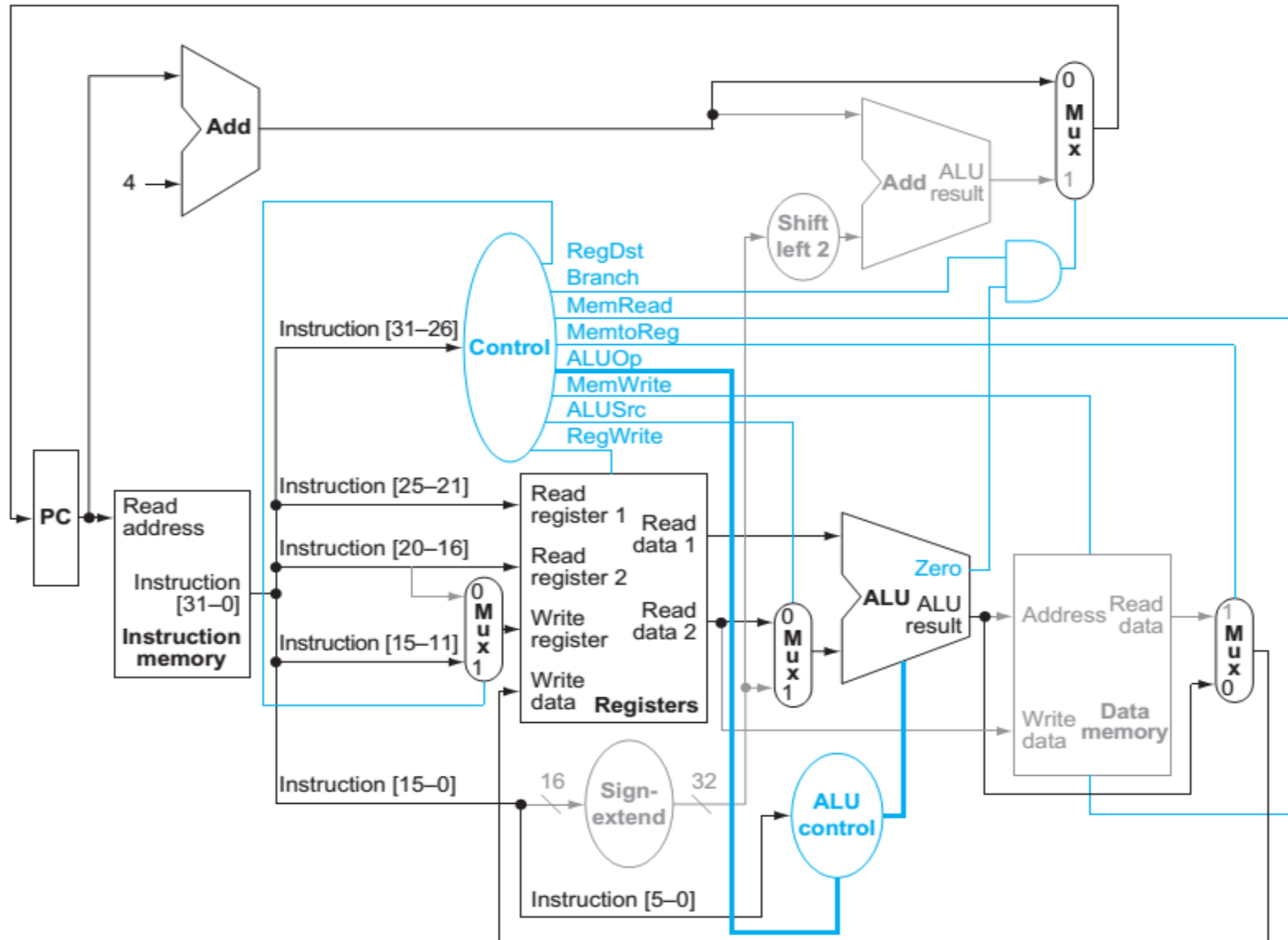
Datapath for Jump



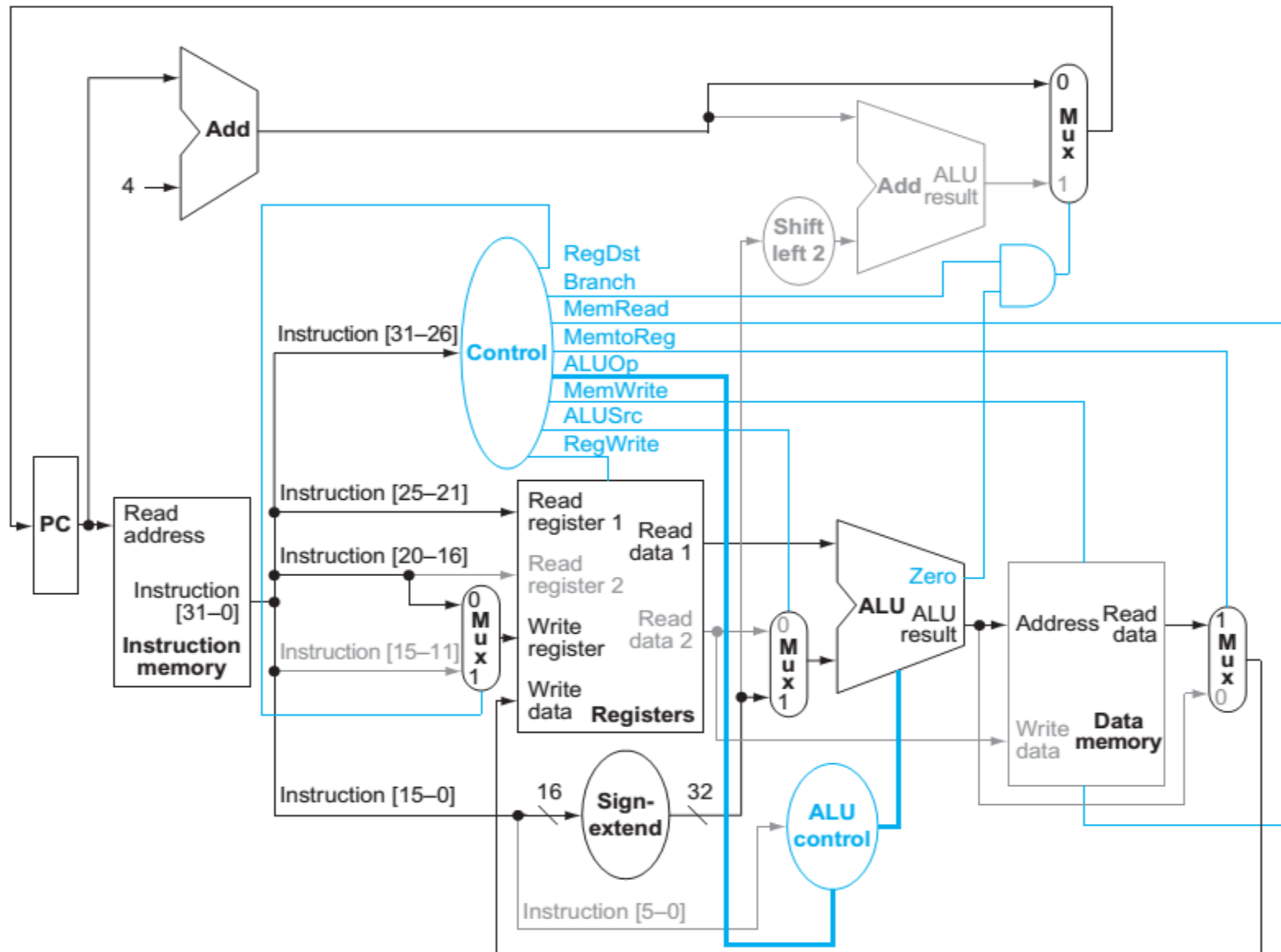
Truth Table for Control Unit

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

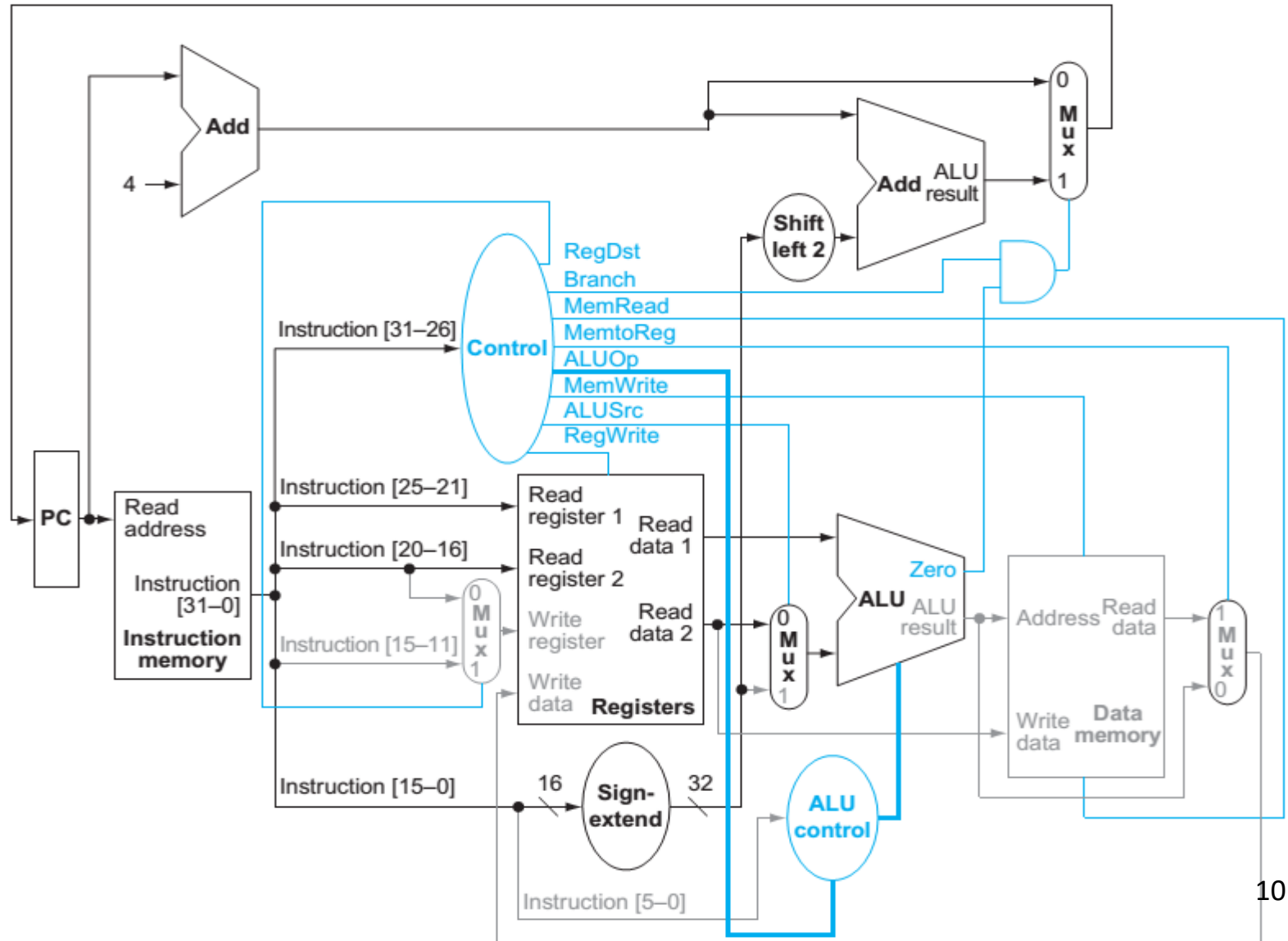
Datapath for an R-type Instruction



Datapath for a load Instruction



Datapath for a Branch-on-equal Instruction



Truth Table for Control Unit

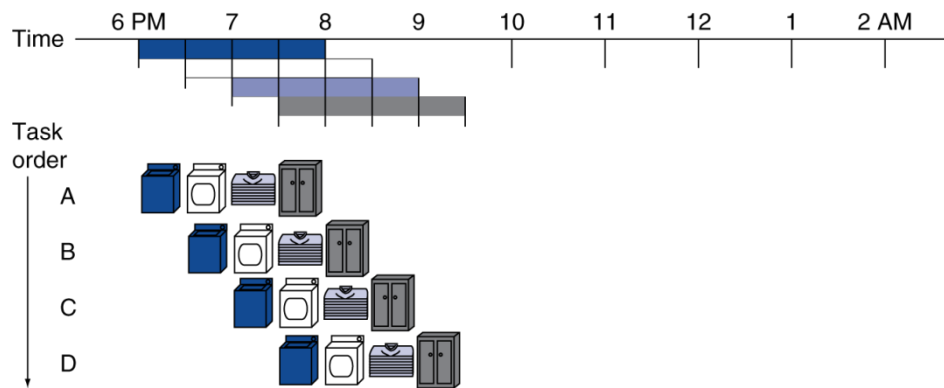
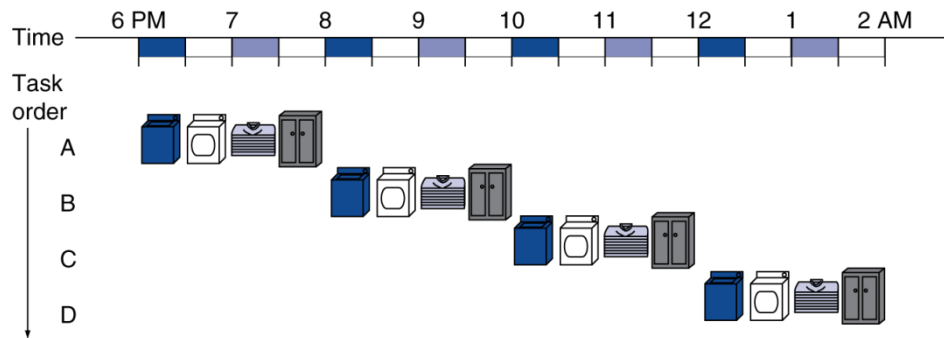
Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

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



- Four loads:
 - Speedup
 $= 16/7 = 2.3$
- Non-stop:
 - Speedup
 $= 4n/(n + 3) \approx 4$
 $= \text{number of stages}$

MIPS Pipeline

- Pipeline: an implementation technique in which **multiple instructions** are **overlapped** in execution.
- 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

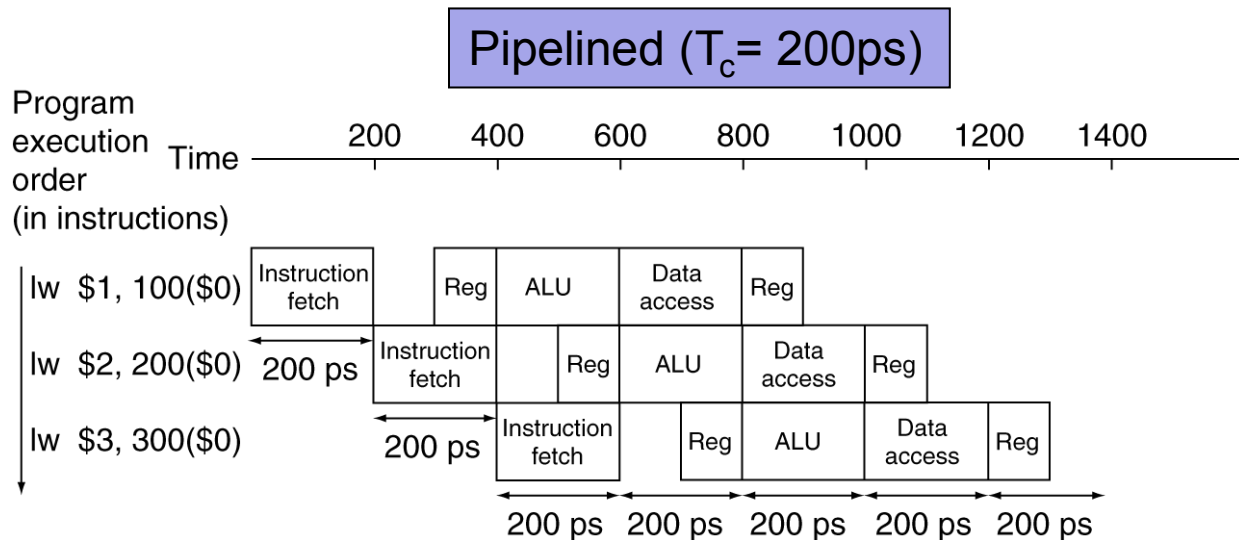
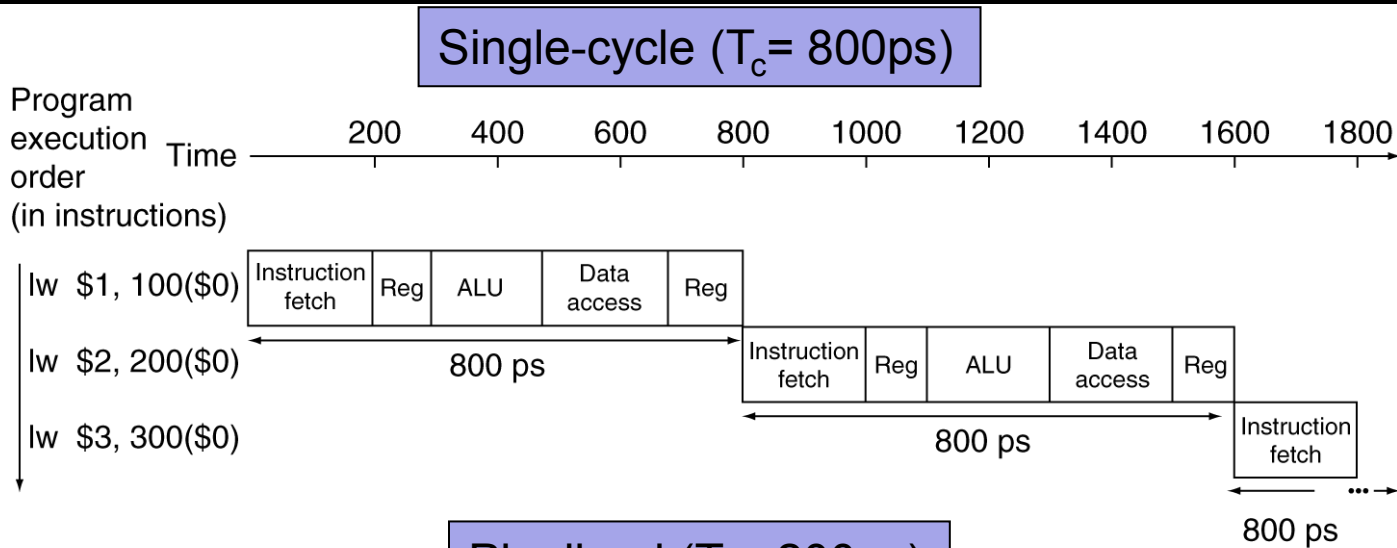
Pipeline Performance

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath *every stage 200ps, every ins 1000ps. { latency ↑ throughput ↑*

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

4x speed up, 因为 Reg Read 和 Write 有 delay

Pipeline Performance



cut as small & average as possible

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

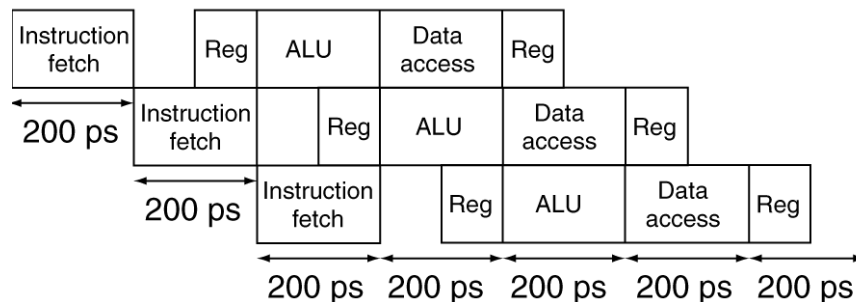
- MIPS ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions *更難 pipeline*
 - Few and regular instruction formats
 - Can decode and read registers in one step
 - Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - 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 hazards
 - Need to wait for previous instruction to complete its data read/write
- Control hazards
 - Decisions of control action depends on the previous instruction

Structure Hazards

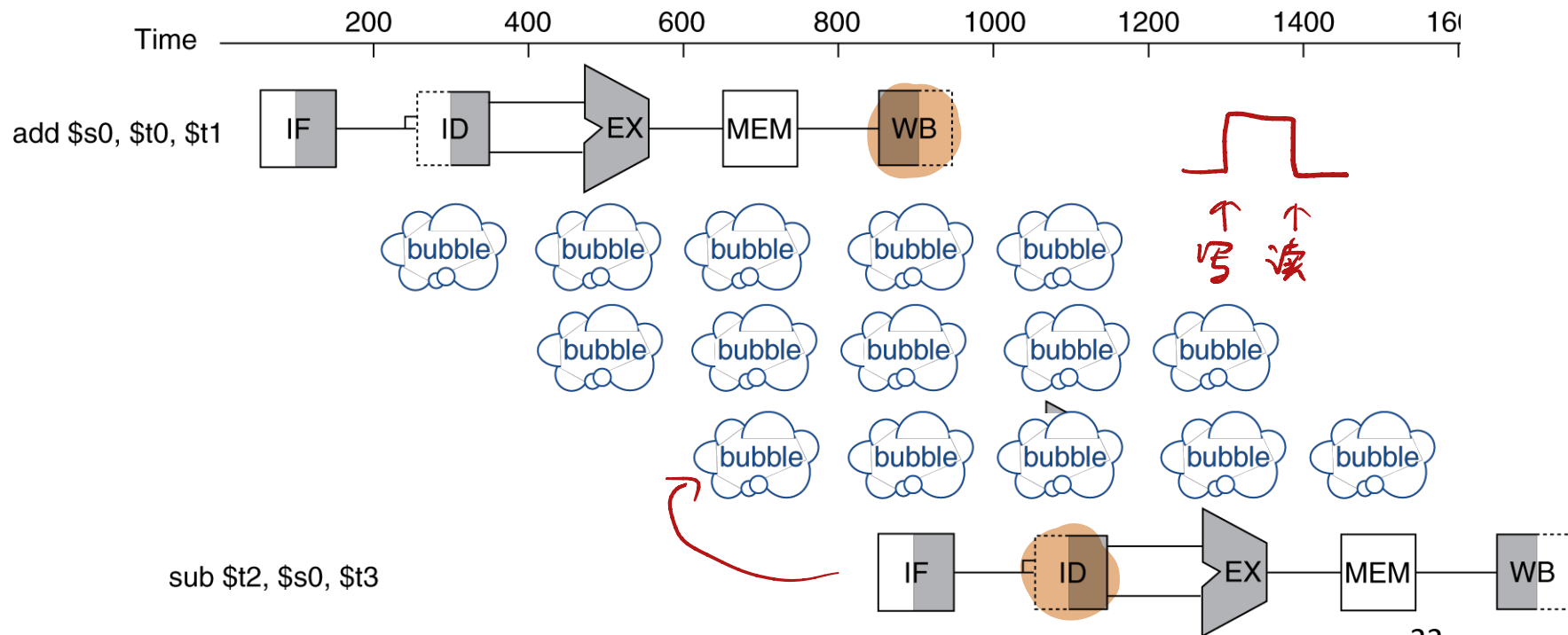
- Conflict for use of a resource
- If MIPS pipeline has only one memory (data and instructions all in one), then
 - Load/store requires data access
 - Instruction fetch would have to *stall* for that cycle
 - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches



Data Hazards

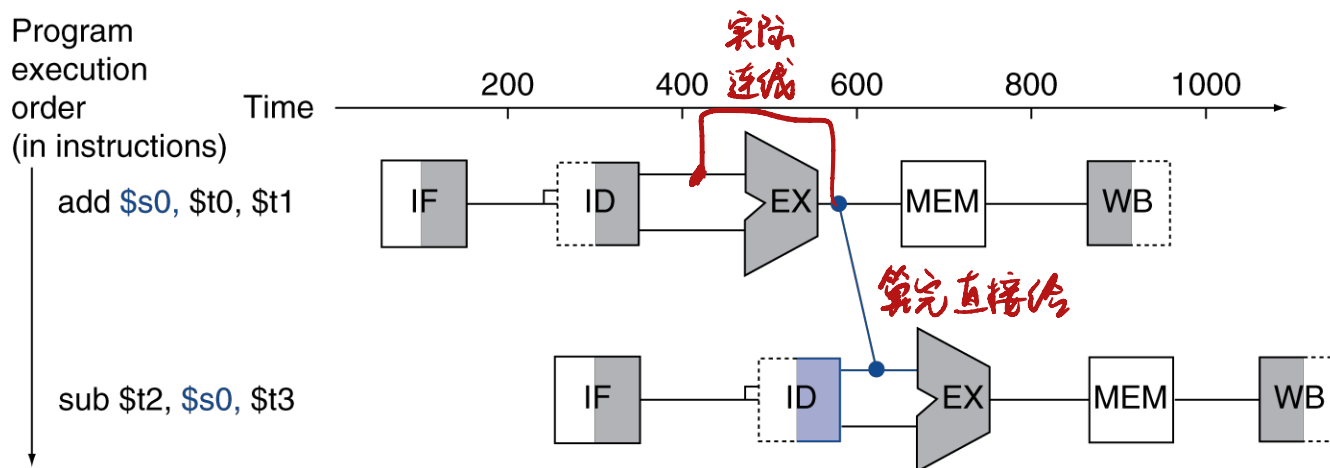
- An instruction depends on completion of data access by a previous instruction

- add $\$s0$, $\$t0$, $\$t1$
 - sub $\$t2$, $\$s0$, $\$t3$



Forwarding (aka Bypassing)

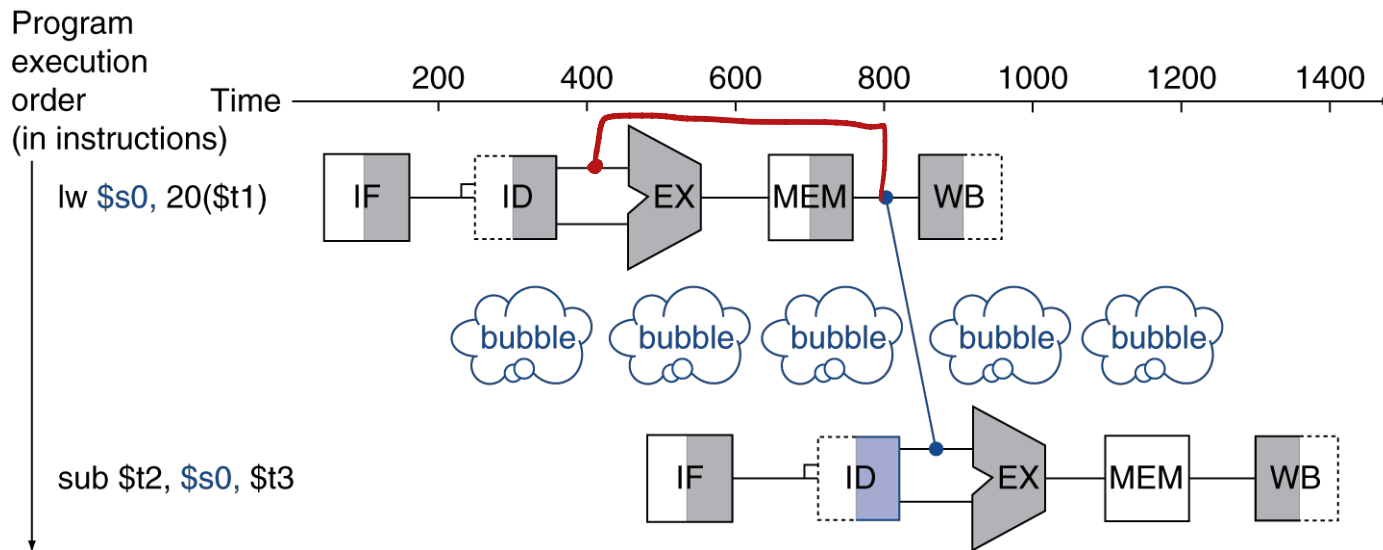
- Forwarding can help to solve data hazard
- Core idea: Use result immediately when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath
 - Add a **bypassing line** to connect the output of EX to the input



EXE - EXE

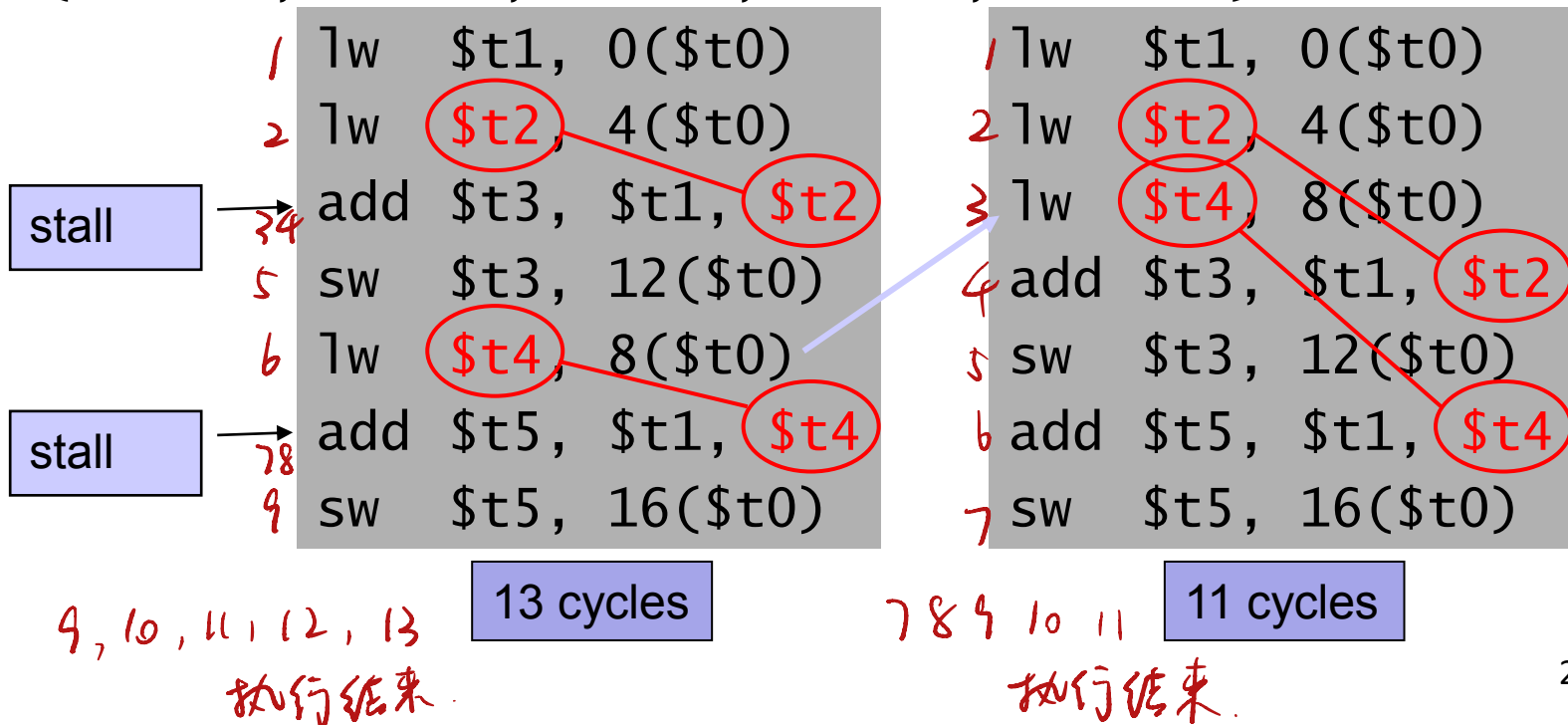
Load-Use Data Hazard

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!
 - E.g. `lw $s0, 20($t1), sub $t2, $s0, $t3`



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction (avoid “load + exe” pattern)
- C code for $A = B + E$; $C = B + F$;
- (t1: B, t2:E, t3:A, t4:F, t5: C)

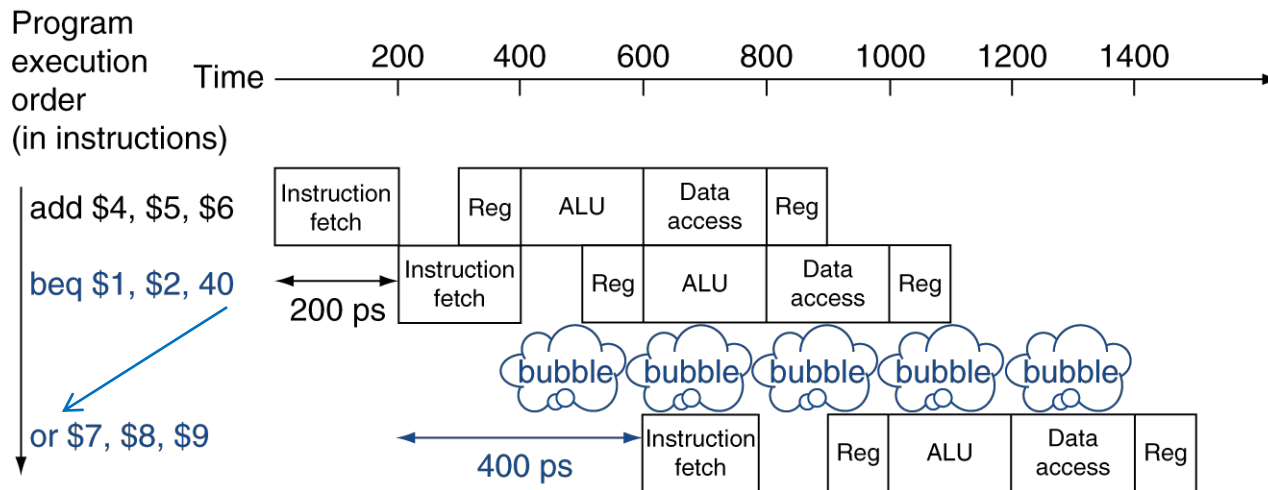


Control Hazards (Branch Hazards)

- 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 MIPS 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

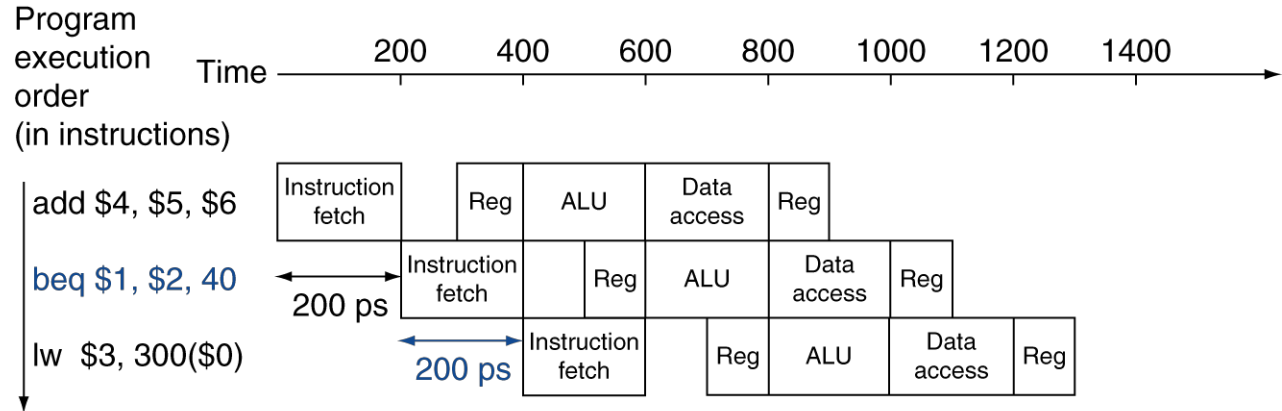


Branch Prediction

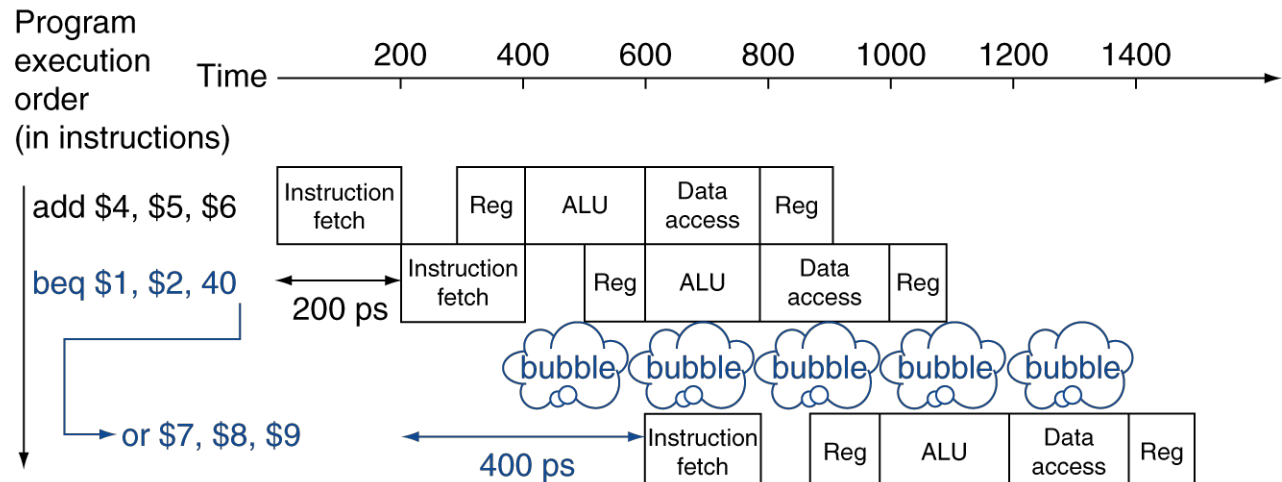
- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Stall only if prediction is wrong
- In MIPS pipeline
 - Can simply predict that branches are not taken
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction
correct



Prediction
incorrect



More-Realistic Branch Prediction

- Static branch prediction
 - for (i=0; i<=100; i++)*
只需要看语句就能猜
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - compiler 决定 (software)*
 - 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