CMPE 200

COMPUTER ARCHITECTURE

Lecture 4 – The Processor – Pipelining

Bo Yu
Computer Engineering Department
SJSU

Adapted from Computer Organization and Design, 5th Edition, 4th edition, Patterson and Hennessy, MK
and Computer Architecture – A Quantitative Approach, 4th edition, Patterson and Hennessy, MK

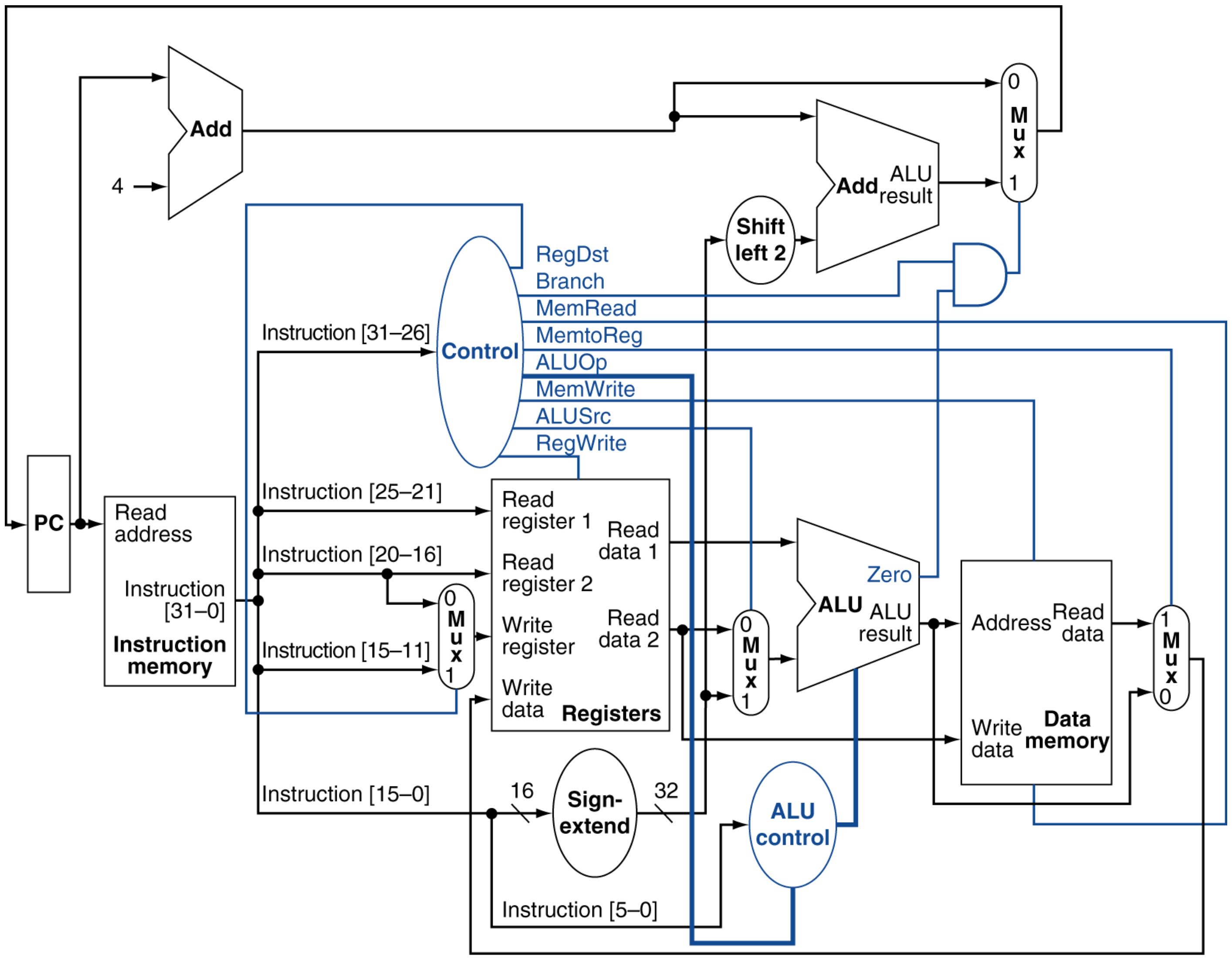
Lecture 3 Key Concepts Review

- Last Lecture:
  - Simple Single-Cycle Data Path implementation
  - Simple Single-Cycle Control Unit Implementation

Lecture 3 Key Concepts Review

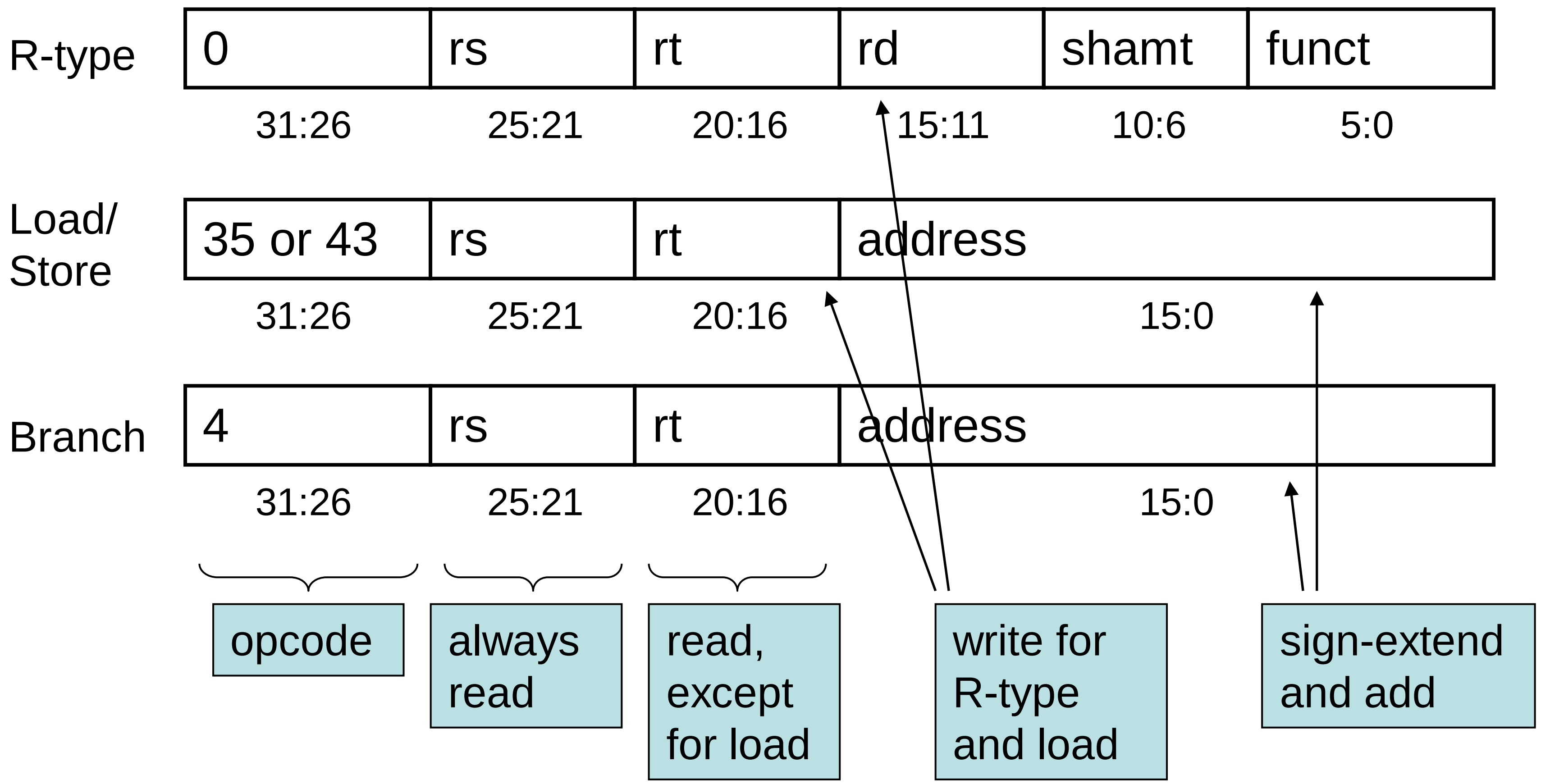
- Datapath Design
  - **Single clock**
  - Pipelining - later
  - Hazards - later
  - Parallelism – ILP - later
- Control Unit Design
  - **Control Unit**
  - **Jumps**
  - CALL/RETURN - later
  - Exceptions - later
- Simple MIPS Implementation
  - Single Clock Implementation
  - Fixed size instructions
  - Uses a subset of core MIPS instruction set (R, I, J-type)
    - Memory reference: lw, sw
    - Arithmetic/logical: add, sub, and, or, slt
    - Control transfer: beq, j
- Instruction Execution (5 steps)
  - Instruction Fetch
  - Operand Read
  - Execute
  - Access data memory for load/store
  - Next Instruction
- How it works – Single-Clock System
  - Clock: Rising edge to start
  - Combinational elements: Operate on data
  - State elements: Store information
- Datapath
  - A group of elements (registers, ALUs, Mux, Memories, …) that process data and addresses in the CPU
- Control
  - A group of signals that control how datapath behaves
  - Instruction 6 higher bits [31-26] and 6 lower bits [5:0] are used as input of control unit

■ Datapath with Control



Lecture 3: The Processor

- The Main Control Unit
  - **Control signals derived from instruction**



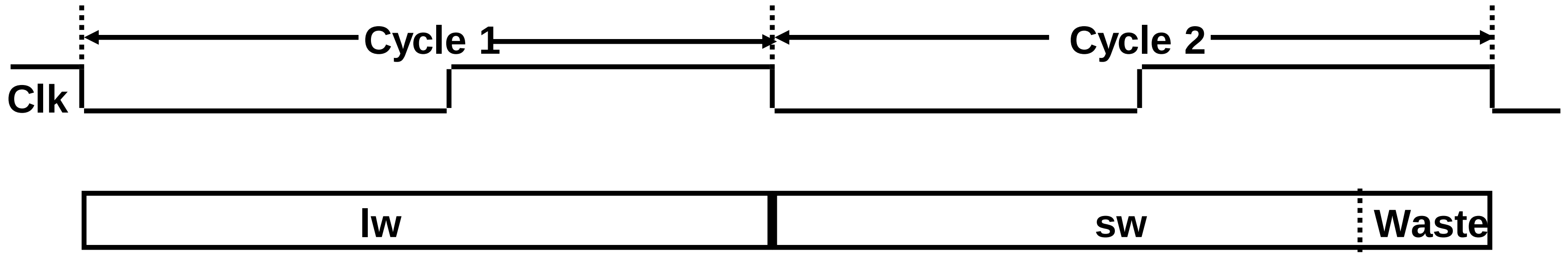
Lecture 3: Key Concepts Review

- Finalizing Control Circuit Design
  - **Combinational Network**

	Inputs				Outputs								
Instruction	Opcode	funct	shamt	ALU func	ALUContrl	RegDst	Branch	JUMP	MemToReg	MemWrite	ALUsrc	RegWrite	zeroExt
LW	xxxxx			ADD	0010	0	0	0	1	0	1	1	0
SW	yyyyyy			ADD	0010	x	0	0	x	1	1	0	0
BEQ	zzzzz			SUB	0110	x	1	0	x	0	0	0	0
R-type	add	100000		ADD	0010	1	0	0	0	0	0	1	0
	sub	100010		SUB	0110	1	0	0	0	0	0	1	0
	and	100100		AND	0000	1	0	0	0	0	0	1	0
	or	100101		OR	0001	1	0	0	0	0	0	1	0
	shift	xxxx	xxxx	SHIFT		1	0	0	0	0	0	1	0
	ori					1	0	0	0	0	0	1	1
Jump					x	x	x	1	x	0	x	0	x

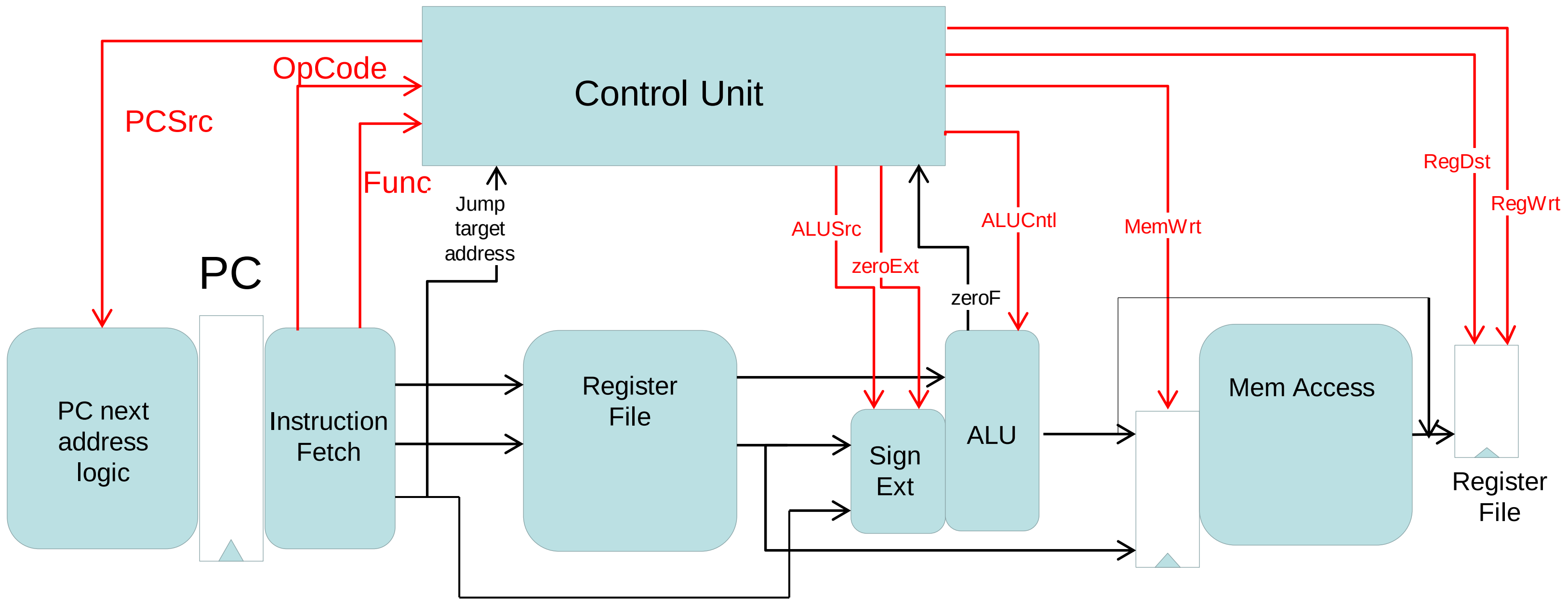
Lecture 3: Key Concepts Review

- Single Clock 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



Lecture 3: Key Concepts Review

- Abstract View of our Single Cycle Processor
  - Looks like a FSM (Finite State Machine) with PC as state



Lecture 3: Key Concepts Review

- Homework Review

Q3:4.1 Consider the following instruction:

Instruction:	AND Rd, Rs, Rt
Interpretation:	Reg[Rd] = Reg[Rs] AND Reg[Rt]

4.1.1 [5] <4.1> What are the values of control signals generated by the control in Figure 4.2 for the above instruction?

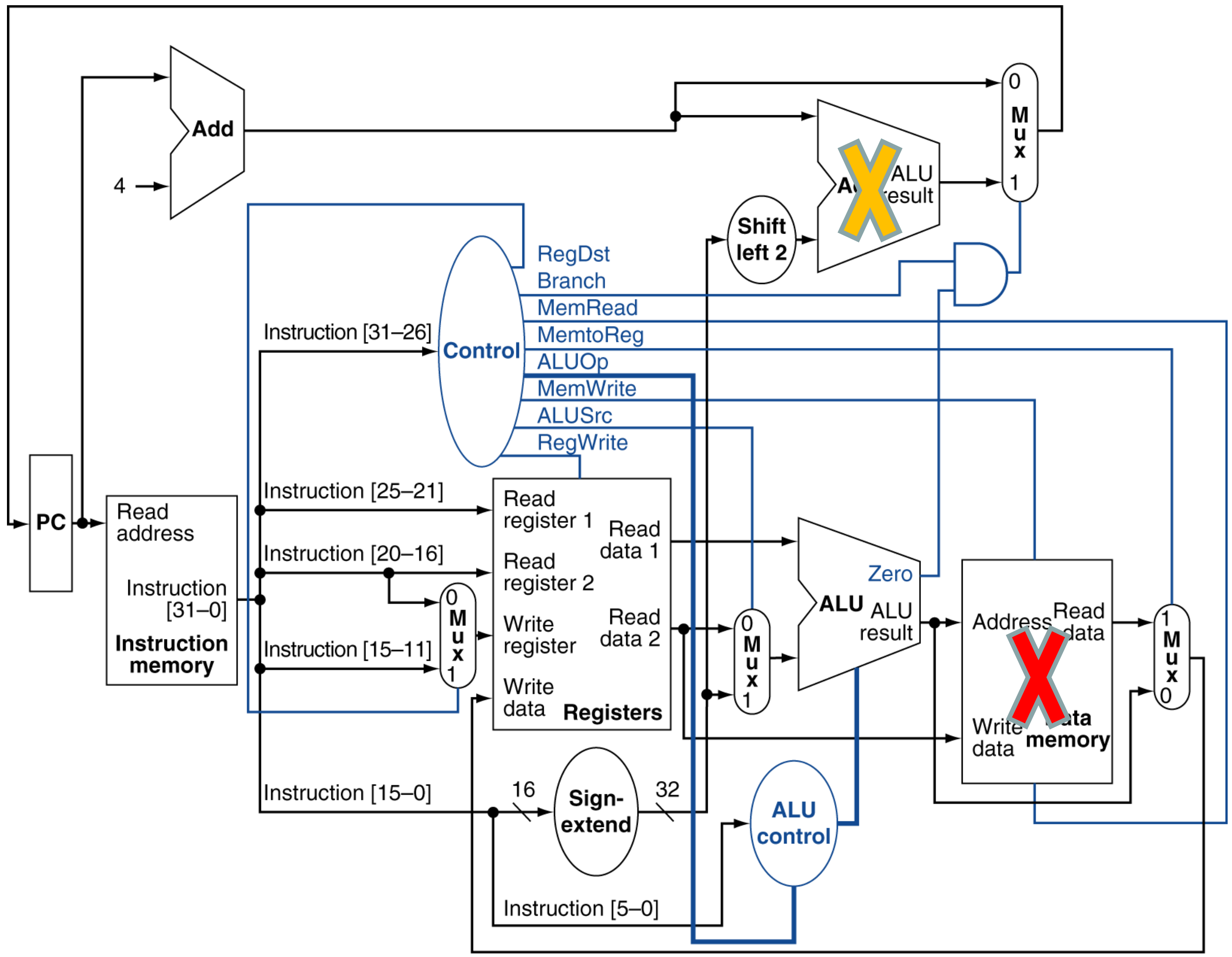
4.1.2 [5] <4.1> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <4.1> Which resources (blocks) produce outputs, but their outputs are not used for this instruction? Which resources produce no outputs for this instruction?

Lecture 3: Key Concepts Review

■ Datapath with Control

RegDst = 1
Branch = 0 (X)
MemRead =0
MemtoReg = 0
ALUOp = AND
MemWrite = 0
ALUSrc = 0
RegWrite = 1

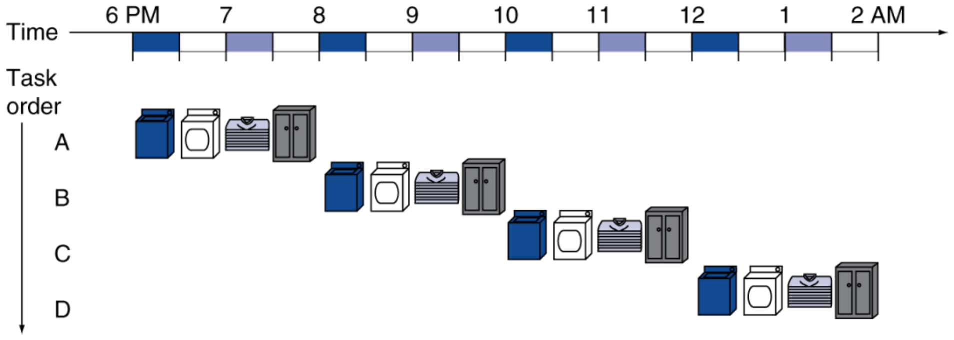


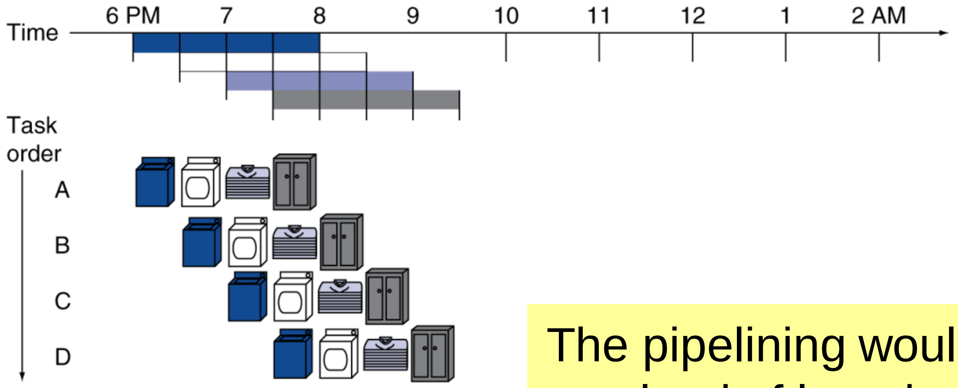
Lecture 4: The Processor - Pipelining

- Today's Lecture (CH4.5-4.6):
  - Pipelining

Lecture 4: The Processor - Pipelining

- Pipelined laundry: overlapping execution
  – Parallelism improves performance



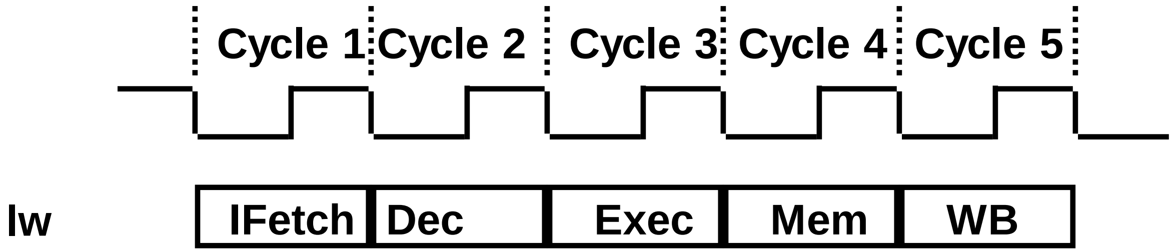


- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/(0.5n + 1.5) ≈ 4 = number of stages

The pipelining would not decrease the time to complete one load of laundry. It improves the **throughput** to decrease the total time of many loads of laundry.

Lecture 4: The Processor - Pipelining

- Five MIPS pipeline stages, one step per stage



- IFetch: Instruction fetch from memory and update PC
- IDec: Instruction decode & registers read
- EXec: Execute R-type or calculate memory address
- MEM: Read/write the data from/to the Data Memory
- WB: Write the result data back into the register file

Lecture 4: The Processor - Pipelining

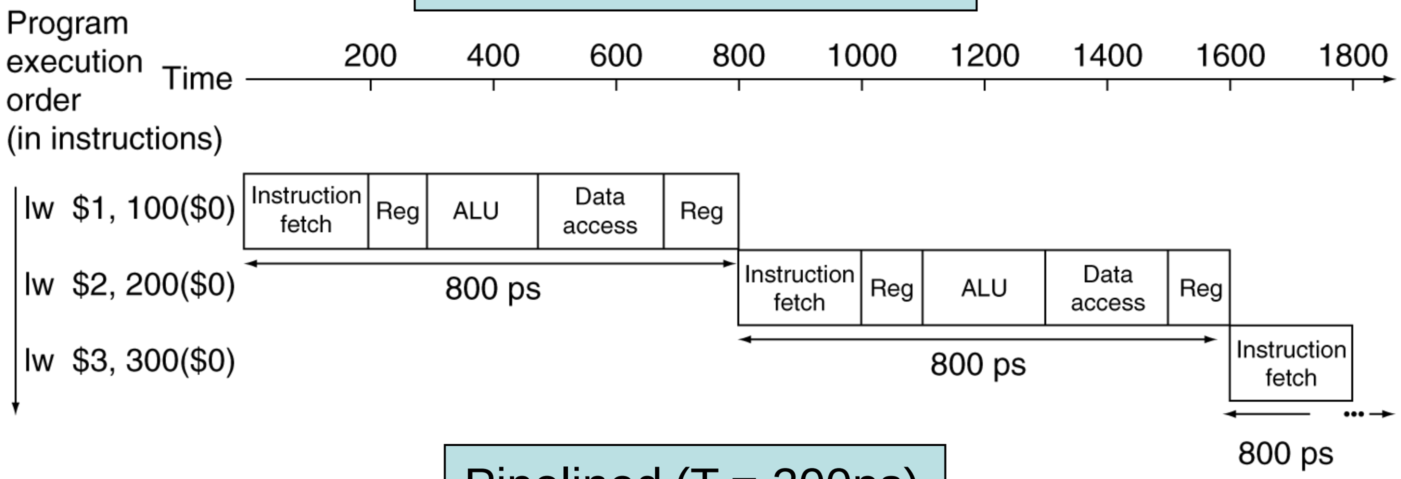
- Pipeline Performance
  - Assume time for stages is
    - 100ps for register read or write
    - 200ps for other stages
  - Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
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

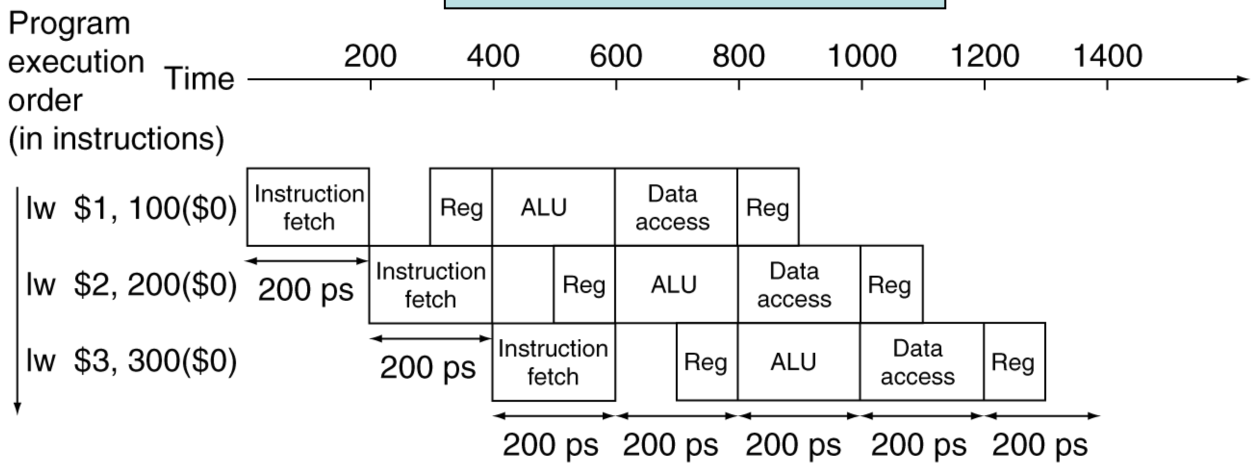
Lecture 4: The Processor - Pipelining

- Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



■ Pipeline Speedup

- If all stages are perfectly balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$

$$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
  - i.e., 800/5=160 vs. 200 in previous page
- 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 15-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage (EX), access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Lecture 4: The Processor - Pipelining

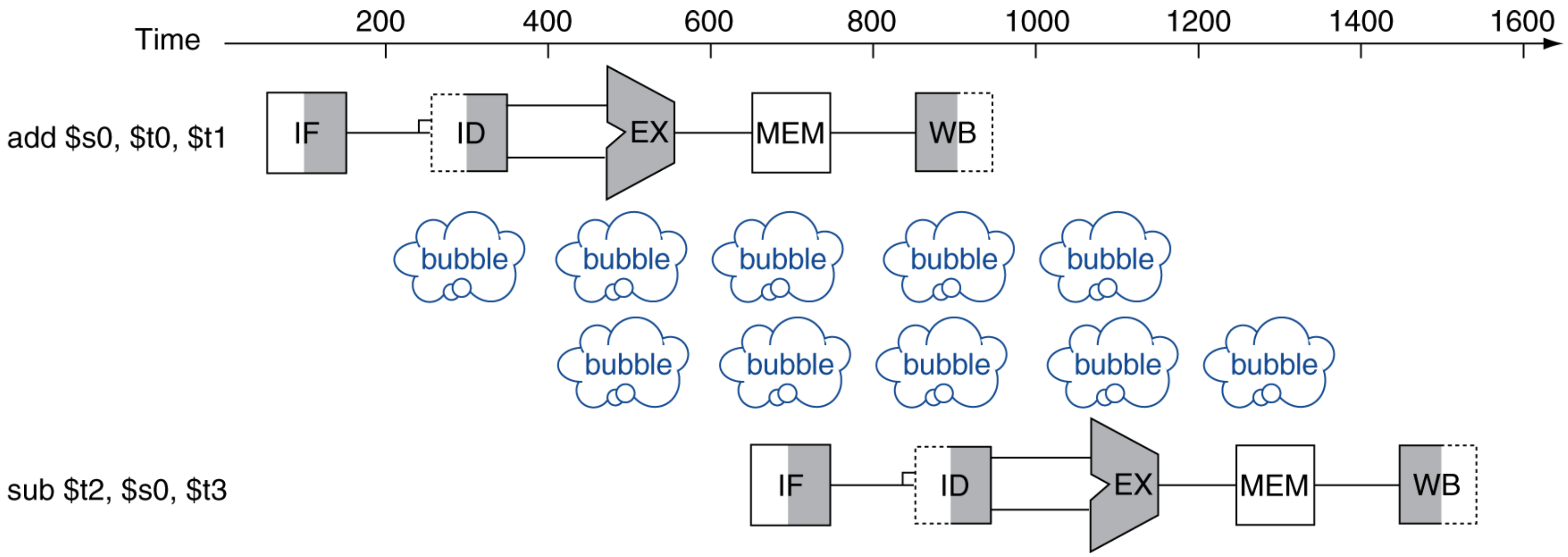
- Pipeline Hazards
  - Situations that prevent starting the next instruction in the next cycle
  - Three different types of hazards
    - Structure hazards
      - A required resource is busy or hardware doesn’t support the combination of instructions to execute
    - Data hazard (aka Pipeline data hazard)
      - Need to wait for previous instruction to complete its data read/write
    - Control hazard (aka Branch hazard)
      - Deciding on control action depends on the results of previous instruction

Structure Hazards

- Conflict for use of a resource
- In MIPS 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

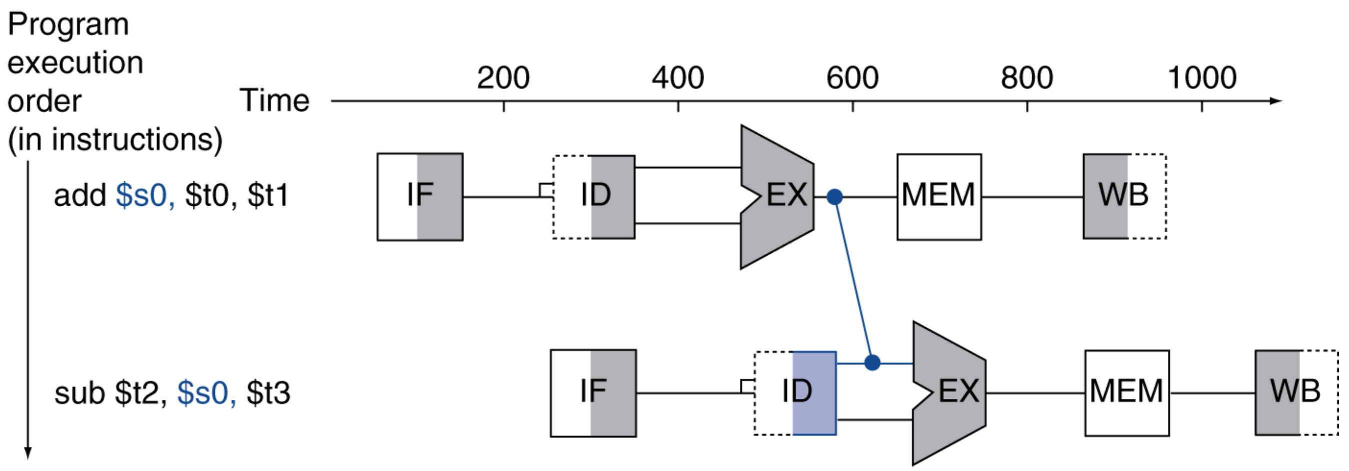
Lecture 4: The Processor - Pipelining

- 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)

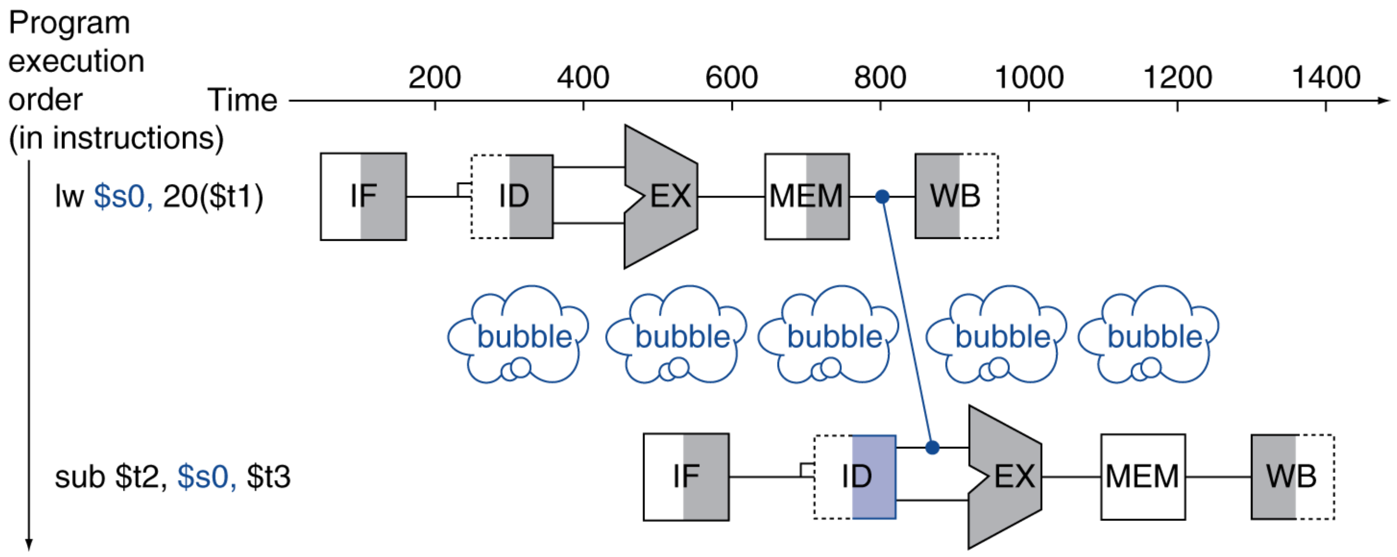
- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Retrieve the missing data from internal buffers
  - Requires extra connections in the datapath



Lecture 4: The Processor - Pipelining

■ Load-Use Data Hazard

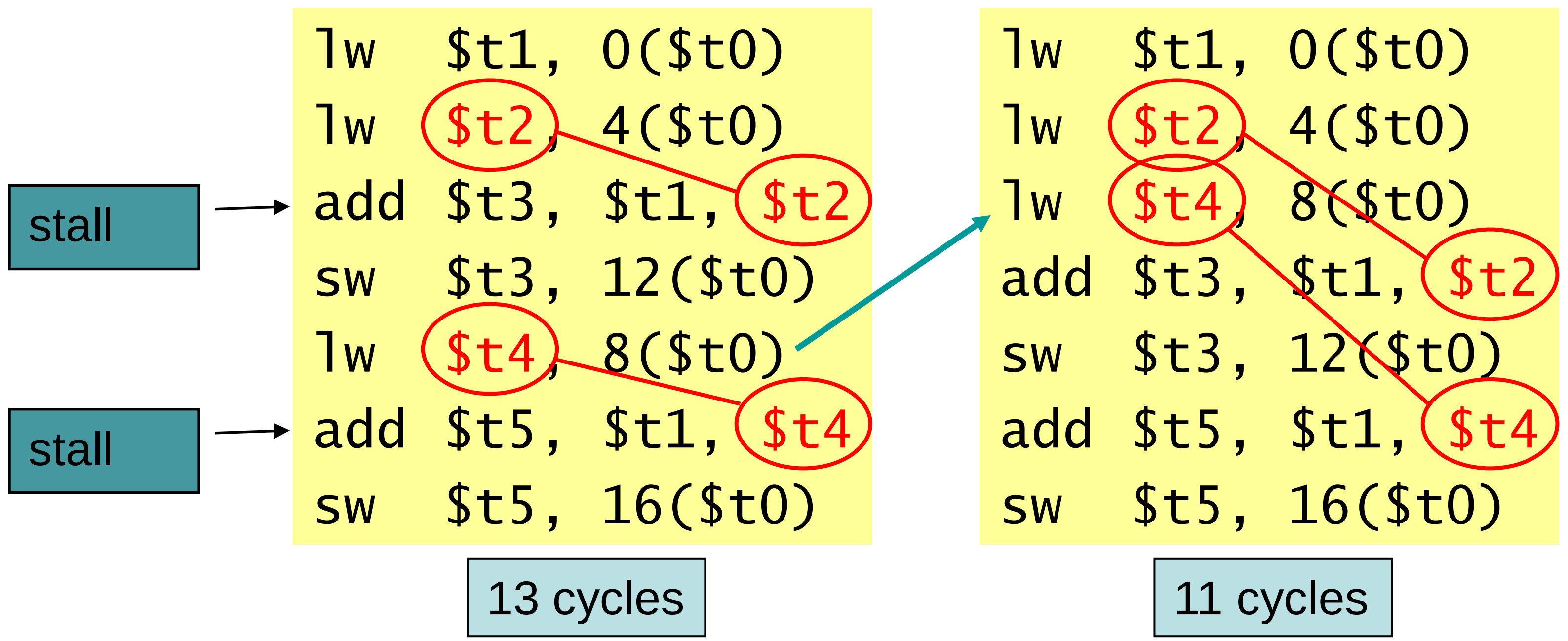
- Data being loaded by load instruction has not become available when it's needed by another instruction
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Bubble (aka pipeline stall): A stall initiated in order to resolve a hazard.

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction, assuming all variables are in memory and are addressable as offsets from $t0
- C code for A `= B + E; C = B + F;`



```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

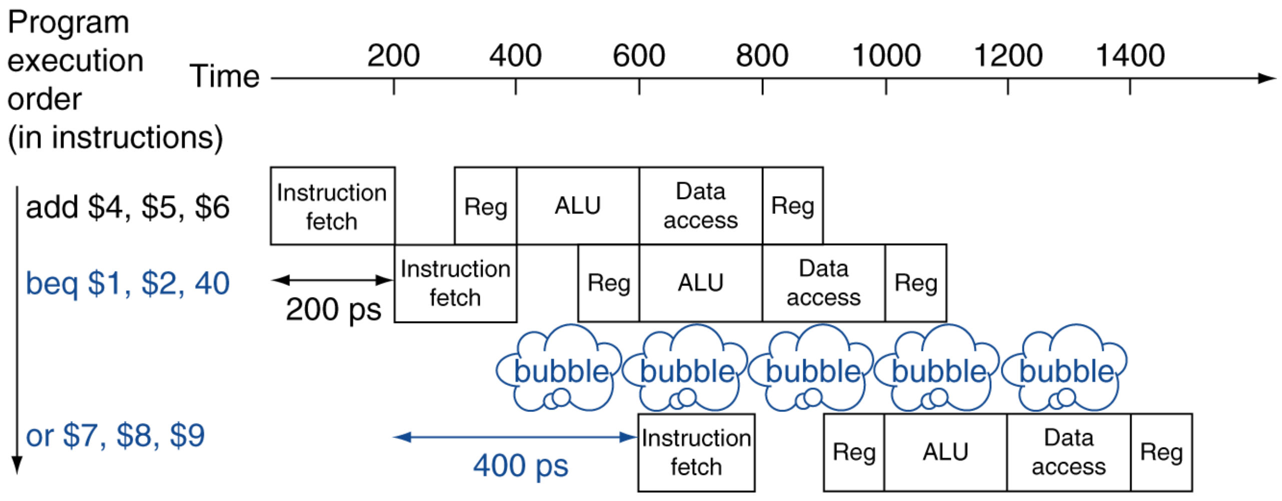
Lecture 4: The Processor - Pipelining

■ Control Hazards (aka 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



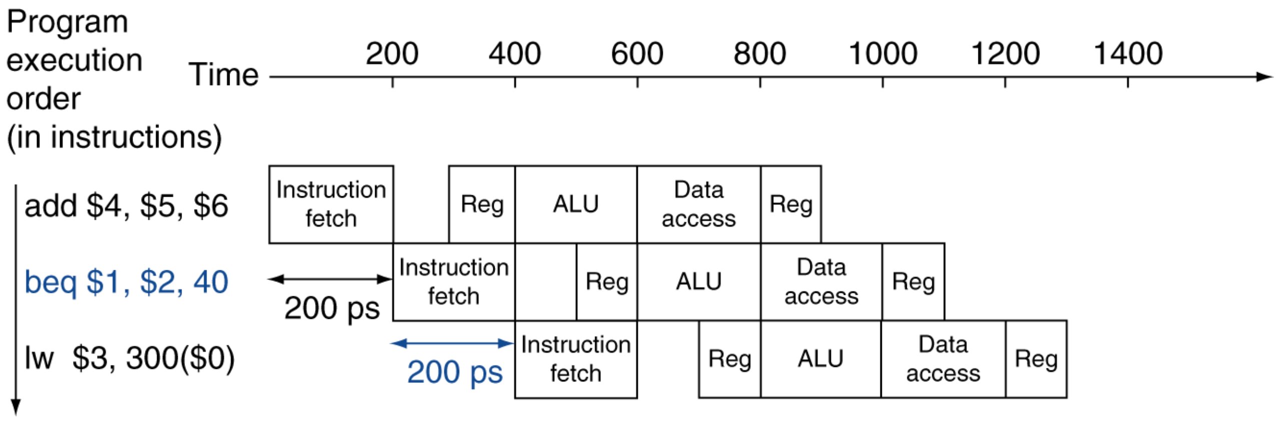
Branch Prediction

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

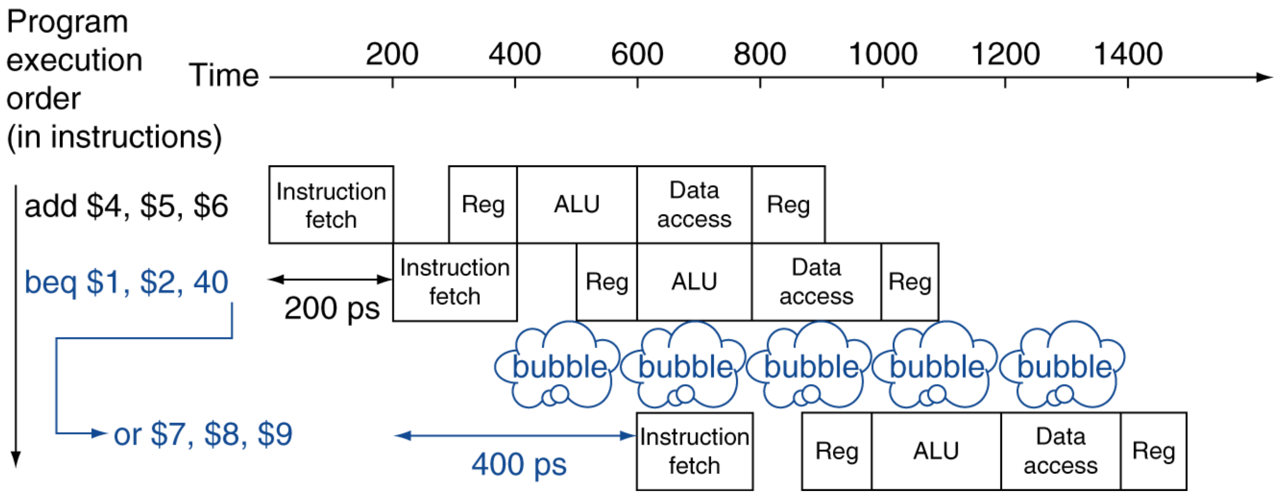
Lecture 4: The Processor - Pipelining

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



Lecture 4: The Processor - Pipelining

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

Lecture 4: The Processor - Pipelining

- Quick Q&A: For each code sequence below, which one is must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

Sequence 1	Sequence 2	Sequence 3

`lw $t0,0($t0)`
`add $t1,$t0,$t0`
	`add $t1,$t0,$t0`
	`addi $t2,$t0,#5`
	`addi $t4,$t1,#5`
		`addi $t1,$t0,#1`
		`addi $t2,$t0,#2`
		`addi $t3,$t0,#2`
		`addi $t3,$t0,#4`
		`addi $t5,$t0,#5`

Must stall on lw result even with forwarding

Can avoid stall using forwarding to bypass the 1st add result written into $t1

Can execute without stalling or forwarding

Lecture 4: The Processor - Pipelining

■ 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