Lecture 10:

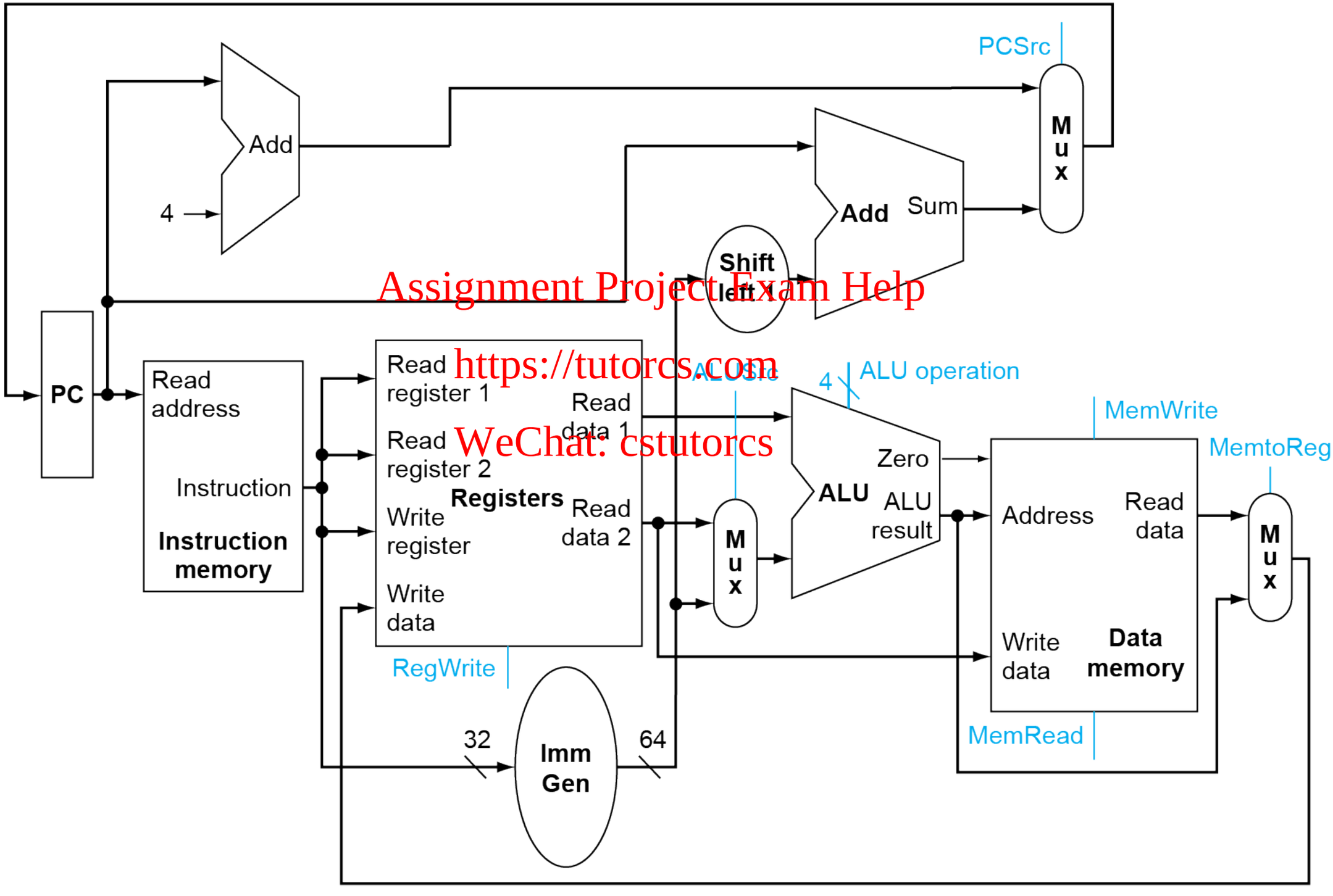
Implementing a Processor 2/5

Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

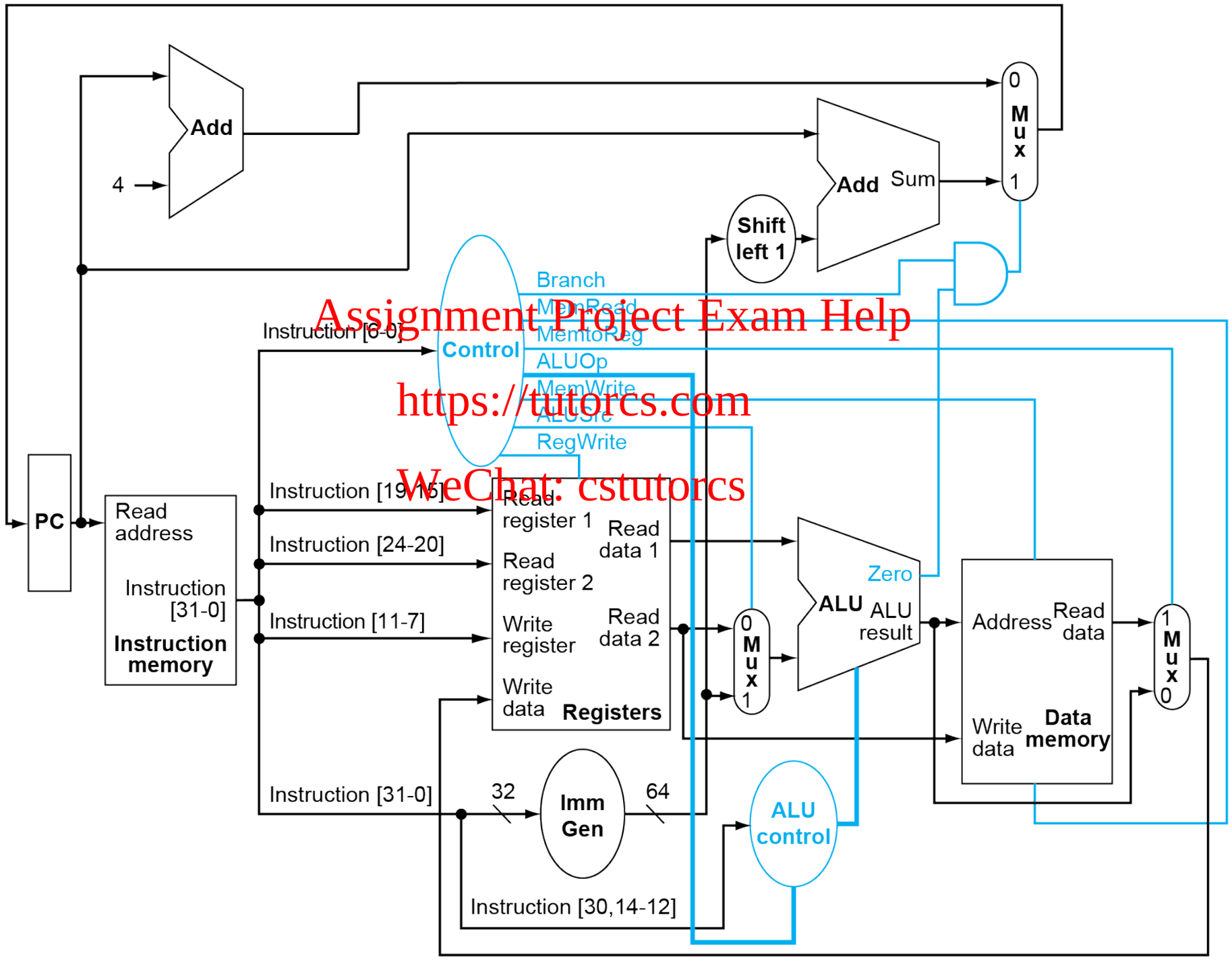
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two RISC-V implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: ld, sd
  - Arithmetic/logical: add, sub, and, or
  - Control transfer: beq

Full Datapath



Datapath With Control

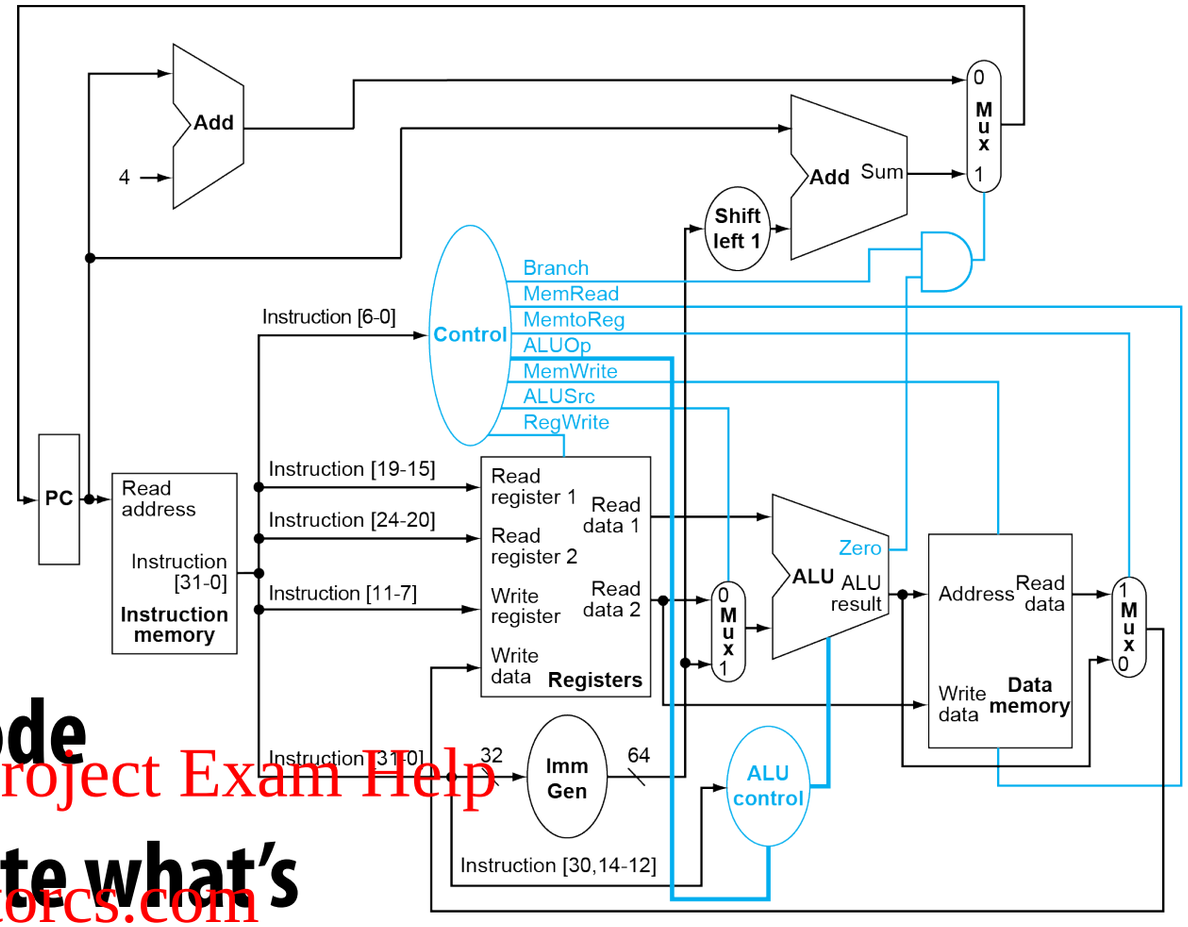


Implementing Logic

- Possibly faster to factor logic into multiple stages
- Recall opcode (6 bits) + funct7 (7 bits) + funct3 (3 bits)
  - Likely a 16-bit function will be very complex!
- Instead, can we do this in stages? Decode the opcode separately from the other bits.
  - Choice of encodings for opcodes can make this much easier

ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode
- We want to eventually generate what's below. How? Next slide.



ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

- Assume 2-bit ALUOp derived from opcode (bits [6:0])
  - Then combinational logic derives ALU control
- Goal here: Set "ALU Control" as a function of ALUOp + funct7/3

opcode	ALUOp	Operation	funct7/3 fields	ALU function	ALU control
ld	00	load word	XXXXXXXXXXXX	add	0010
sd	00	store dword	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	0000000/000	add	0010
		subtract	0100000/000	subtract	0110
		AND	0000000/111	AND	0000
		OR	0000000/110	OR	0001

The Main Control Unit

- Control signals derived from instruction

opcode	ALUOp
ld	00
sd	00
beq	01
R-type	10

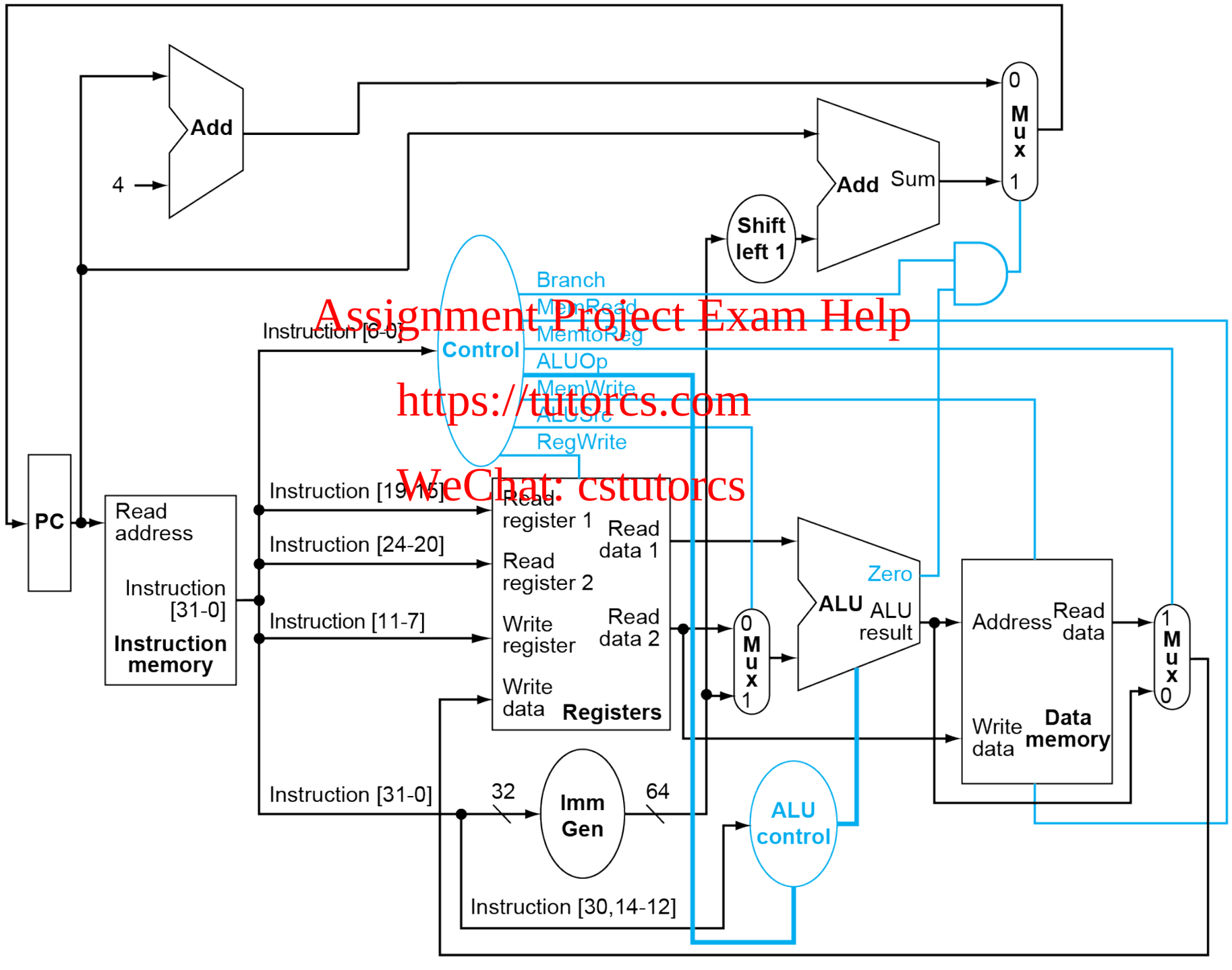
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

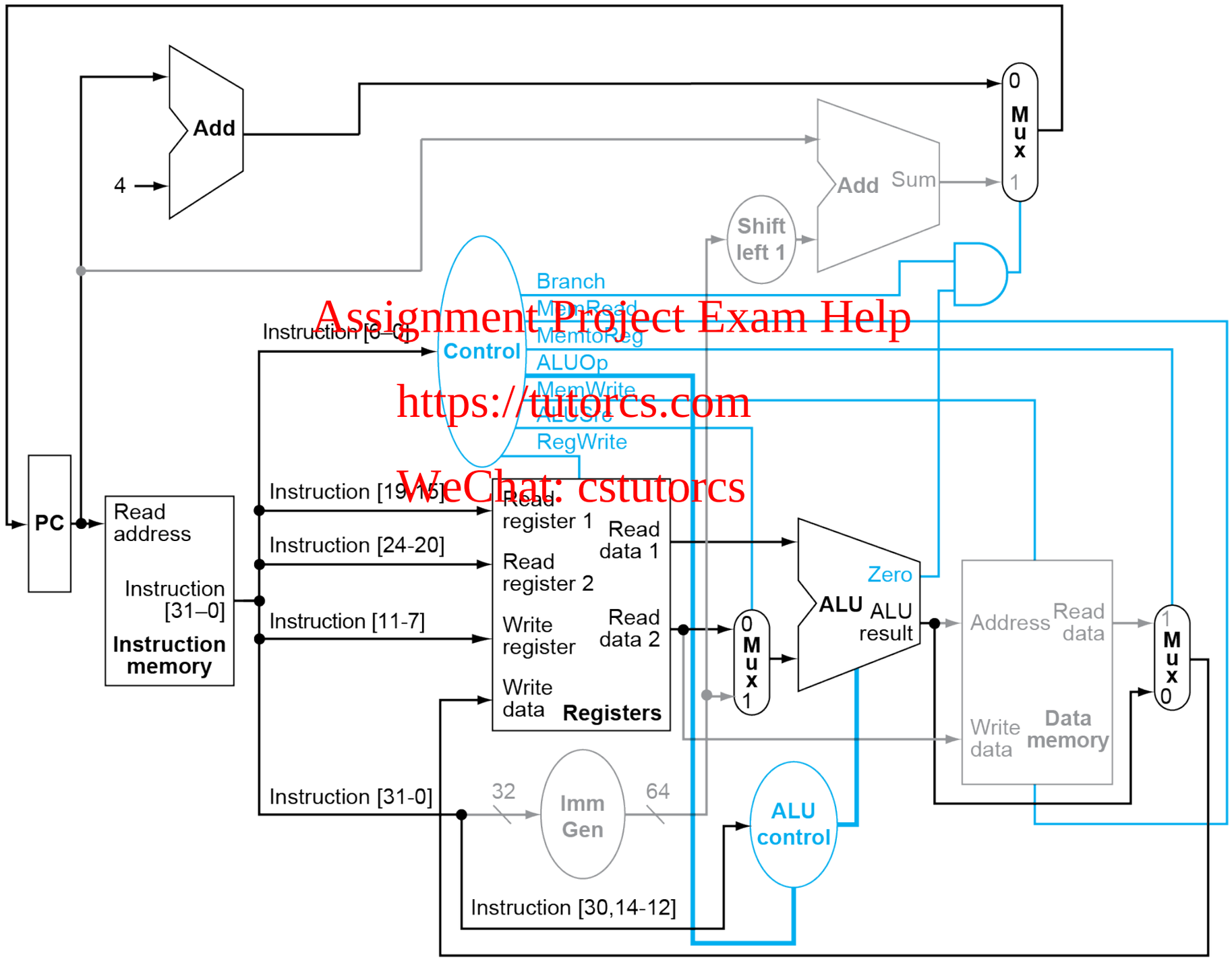
ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

these four bits are the only ones that matter for generating "operation = ALU control"

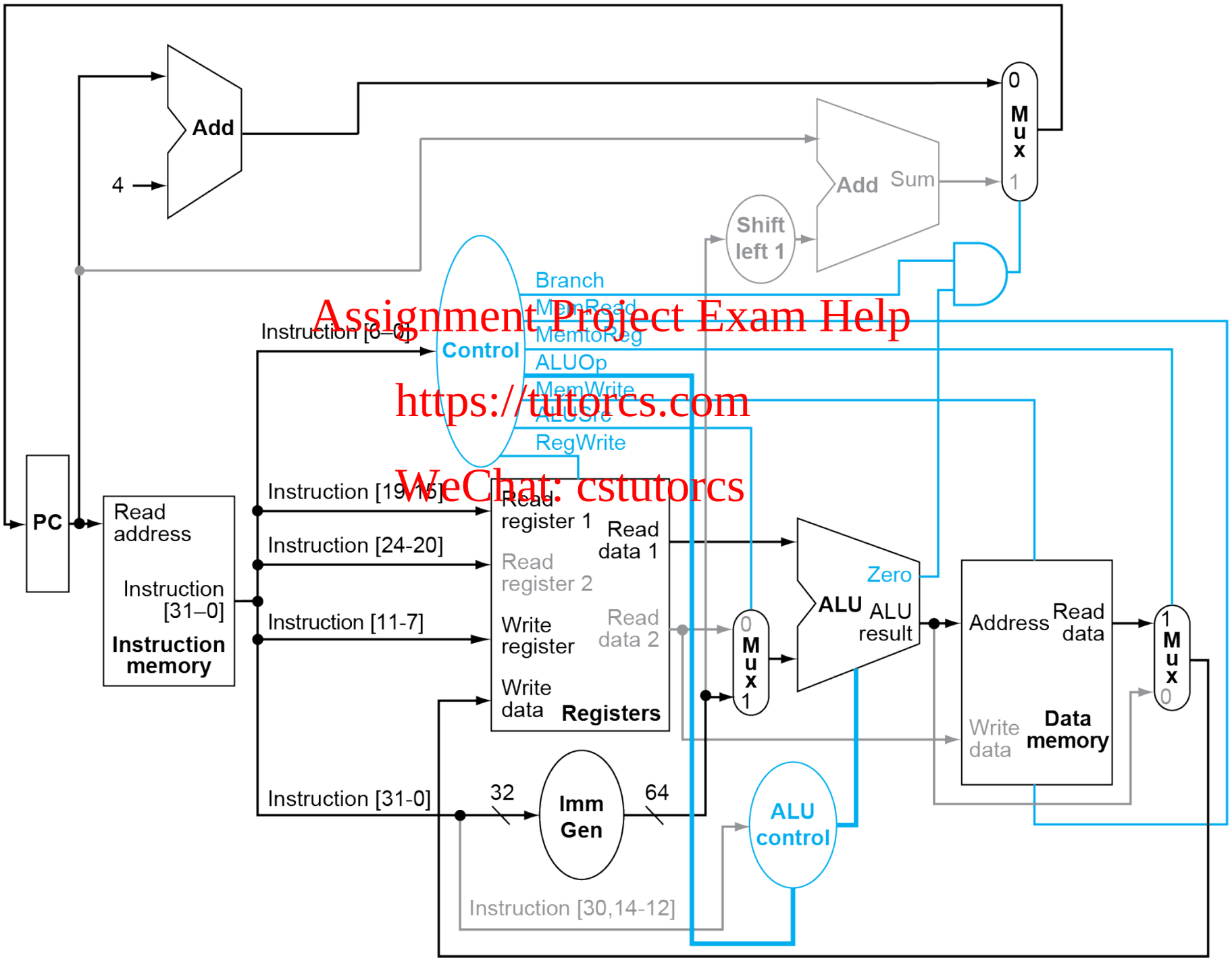
Datapath With Control



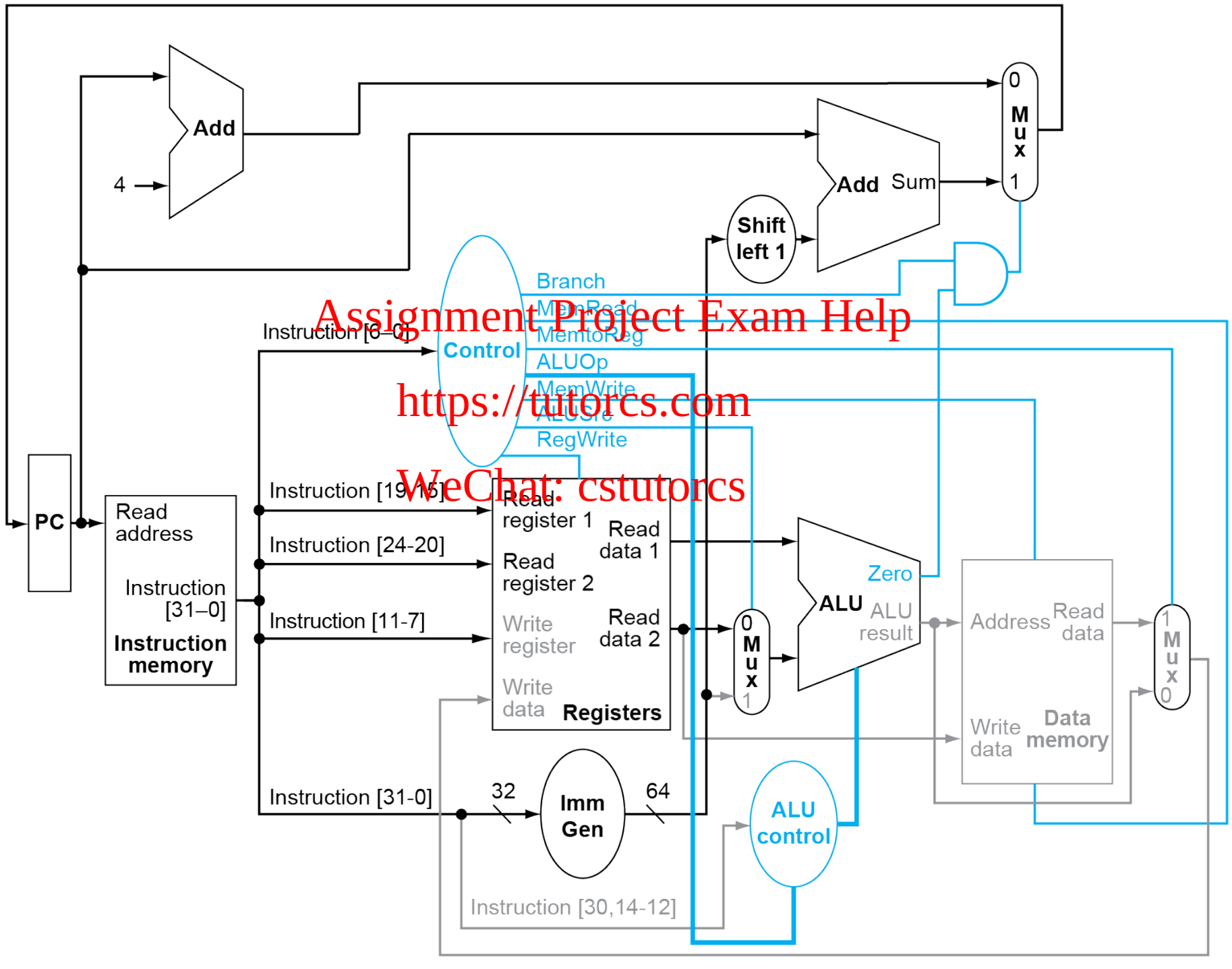
R-Type Instruction



Load Instruction



BEQ Instruction



Building an ISA

- Should we add this other instruction?
- How often would we use it?
- What are the changes to the {datapath, control}?
  - How complicated is the additional logic?
  - Do we have the design time to add it?
  - Would it affect the critical path?

Good Exam Questions

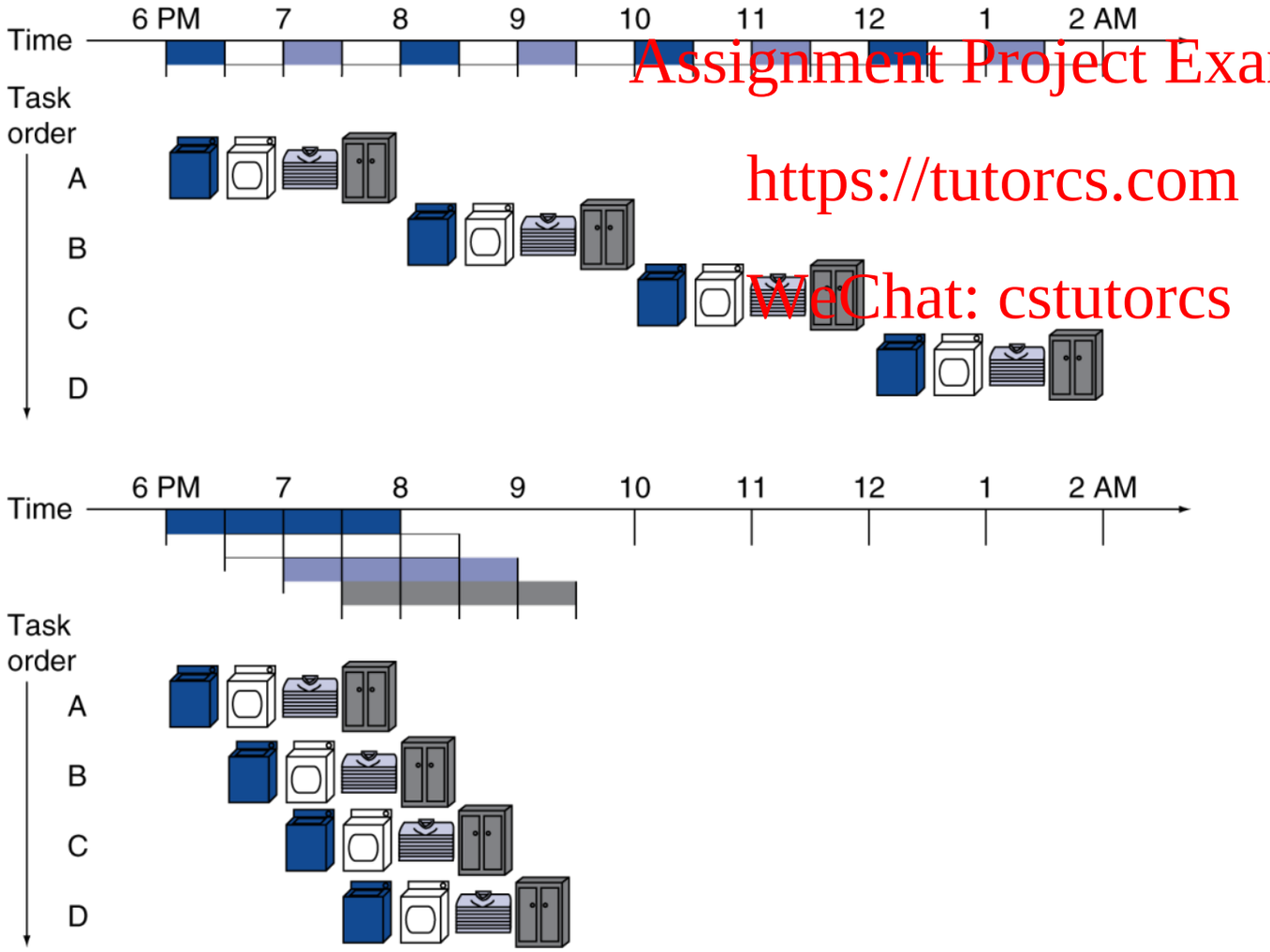
- What are the datapath changes necessary to support new instruction X?
- Given a datapath, what are the control signals necessary to perform instruction Y?
- Given a set of control signals and opcodes, what is the logic for generating control signal Z?

Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
  - But other instructions don't take this long!
- Not feasible to vary period for different instructions
  - Clock distribution, among other issues
- Violates design principle
  - Making the common case fast
- We will improve performance by *pipelining*

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance
- How many "stages" is this laundry process?



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

RISC-V Pipeline

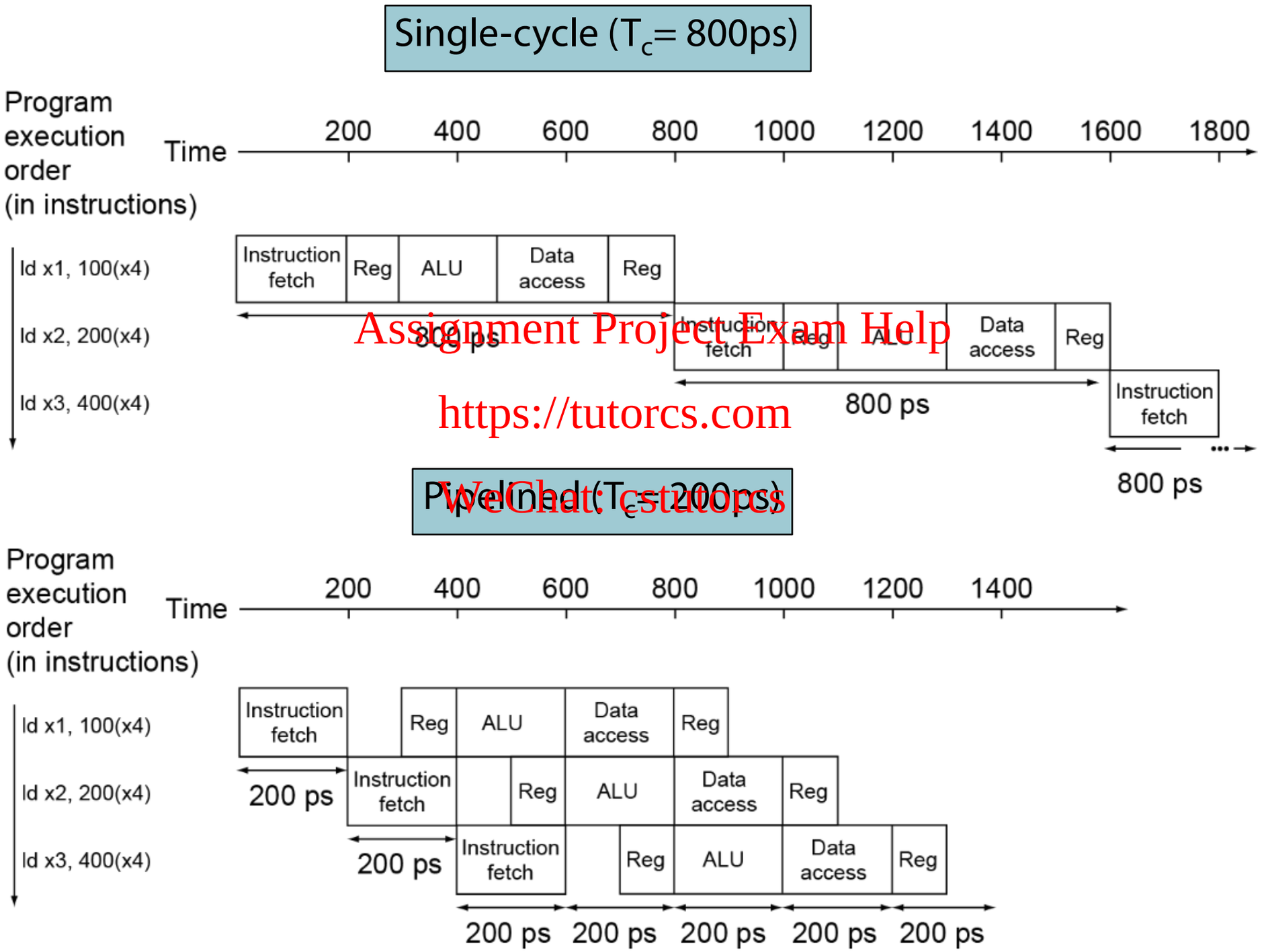
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

- Assume time for stages is
  - 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



Pipeline Speedup

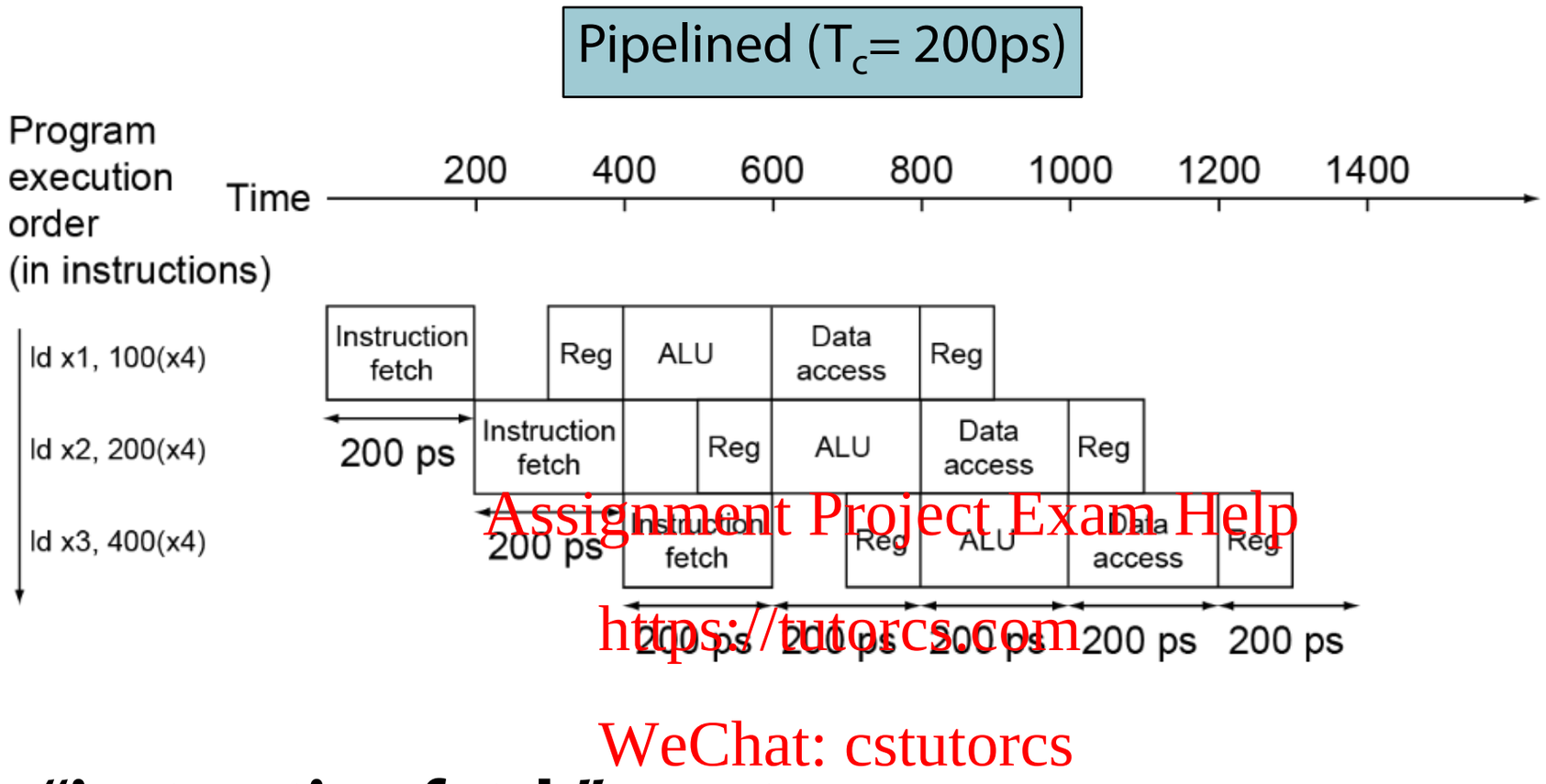
- If all stages are "balanced":
  - i.e., all take the same time
  - then: Time between instructions$_{pipelined}$

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

- If not balanced, speedup is less

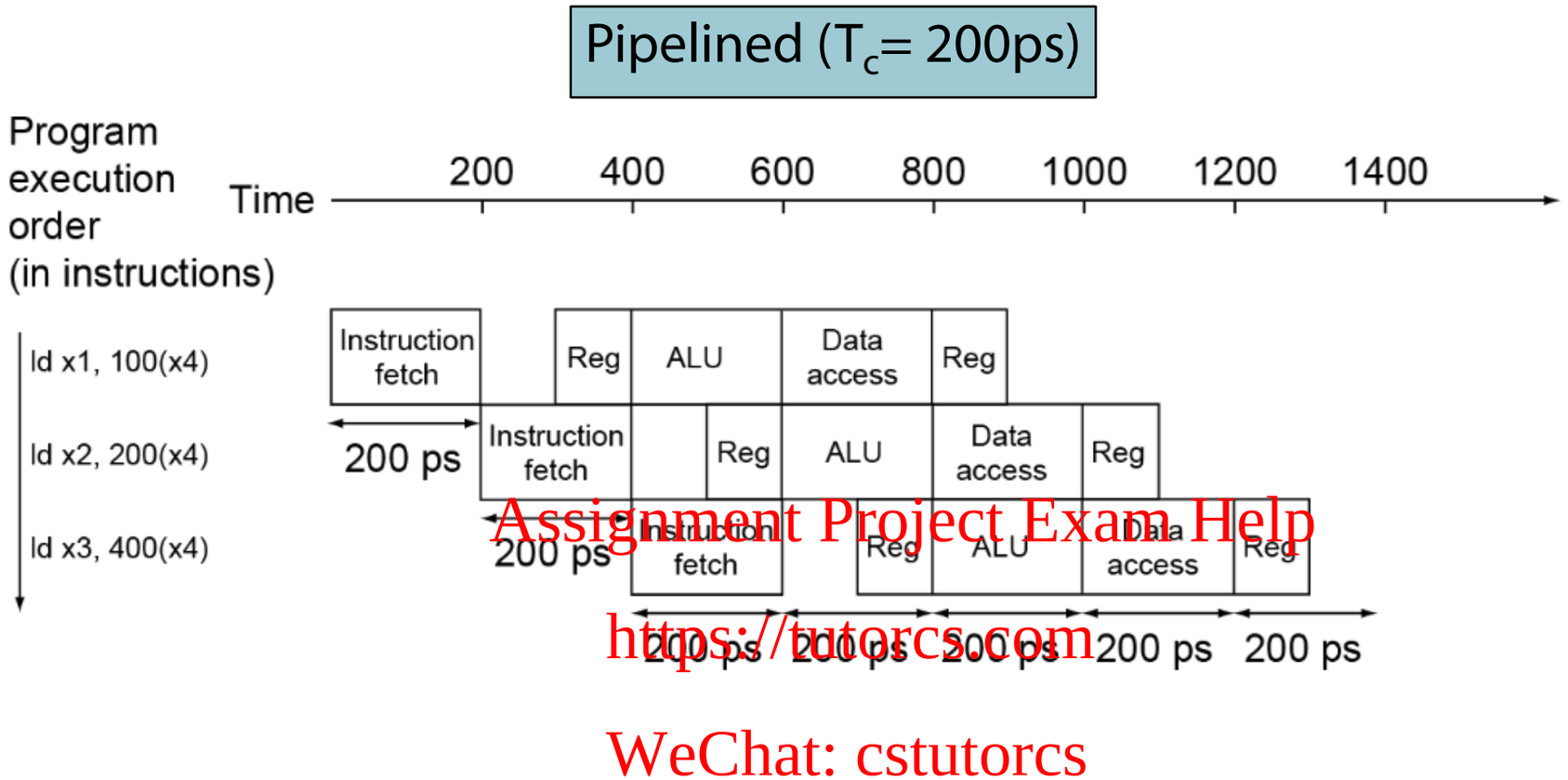
- Speedup is due to increased throughput
  - Latency (time for each instruction) does not decrease
  - *Pipelining helps throughput, not latency*

Fetching the next instruction



- For "instruction fetch", must:
  - Go to memory and get instruction
  - Decode instruction to figure out how long it is
  - Increment program counter appropriately to point to next instruction

Fetching the next instruction



- For "decode", must:
  - Decode instruction to figure out what kind of instruction it is
  - Based on what instruction it is, fetch registers from the RF
    - That's a lot to do! How do we make it fast?

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32 bits
    - This is a critical aspect of the RISC philosophy
    - 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

RISC-V Registers are always in the same place!

If we read one register, it's here

If we read a second register, it's here

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

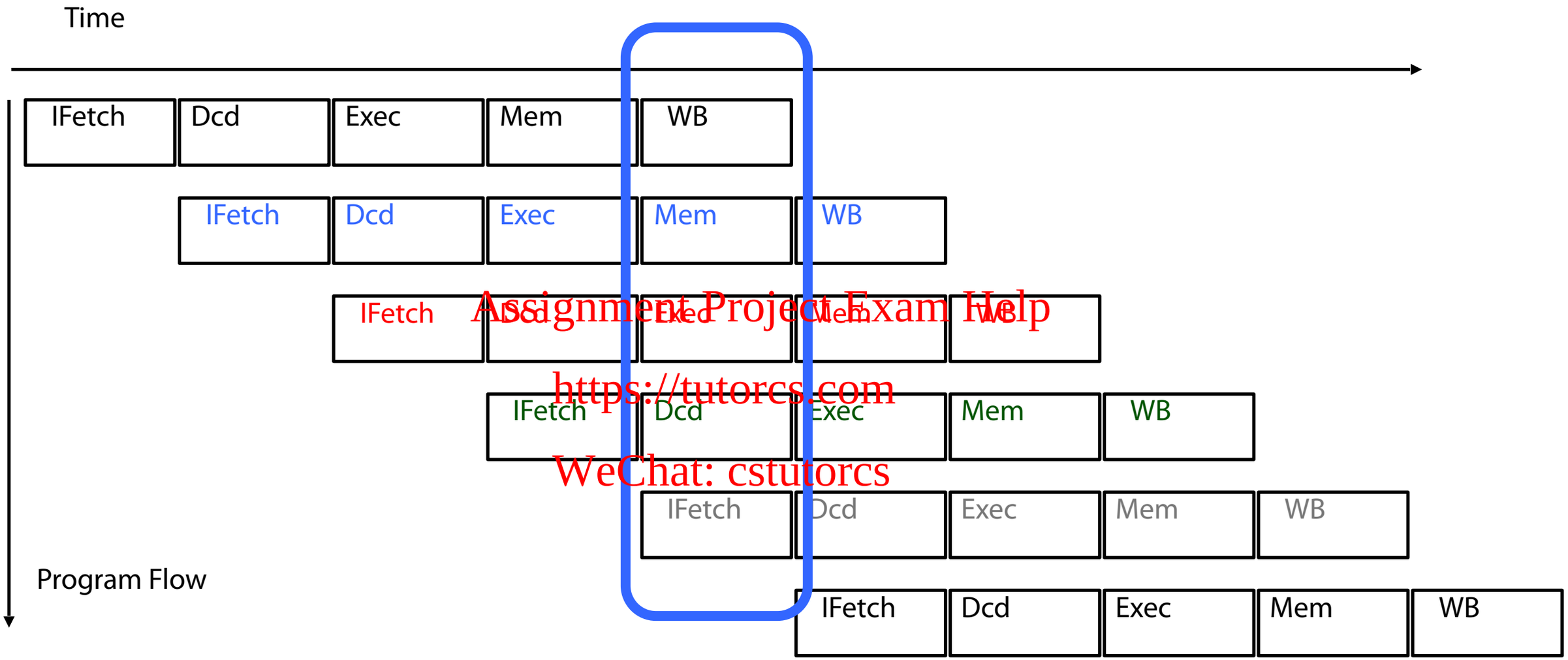
If we write to a register, it's here

- **We don't even have to decode an instruction to start register access!**

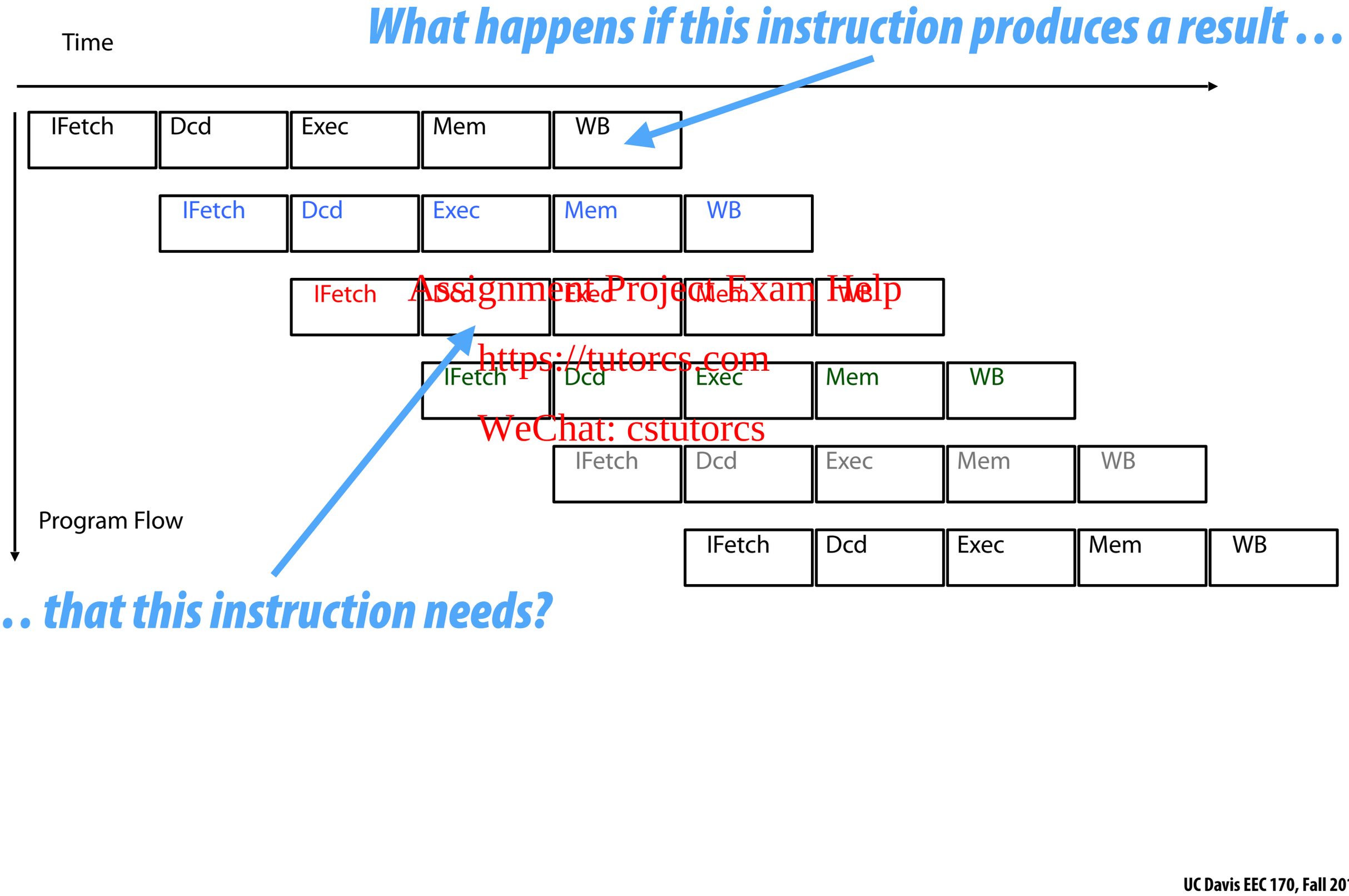
Recap: RISC-V Pipeline

- 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

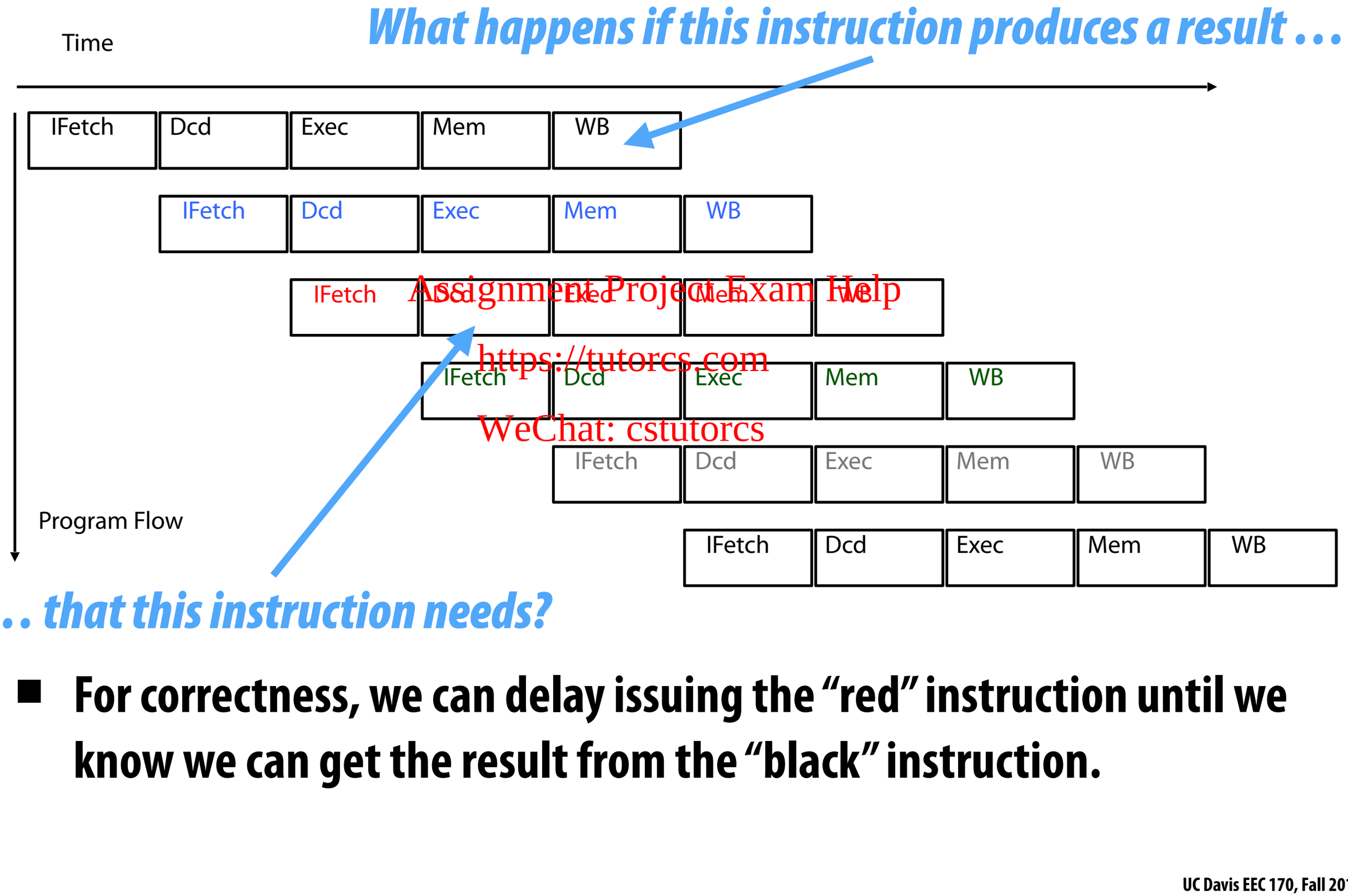
Conventional Pipelined Execution Representation



We Have No Time Machine

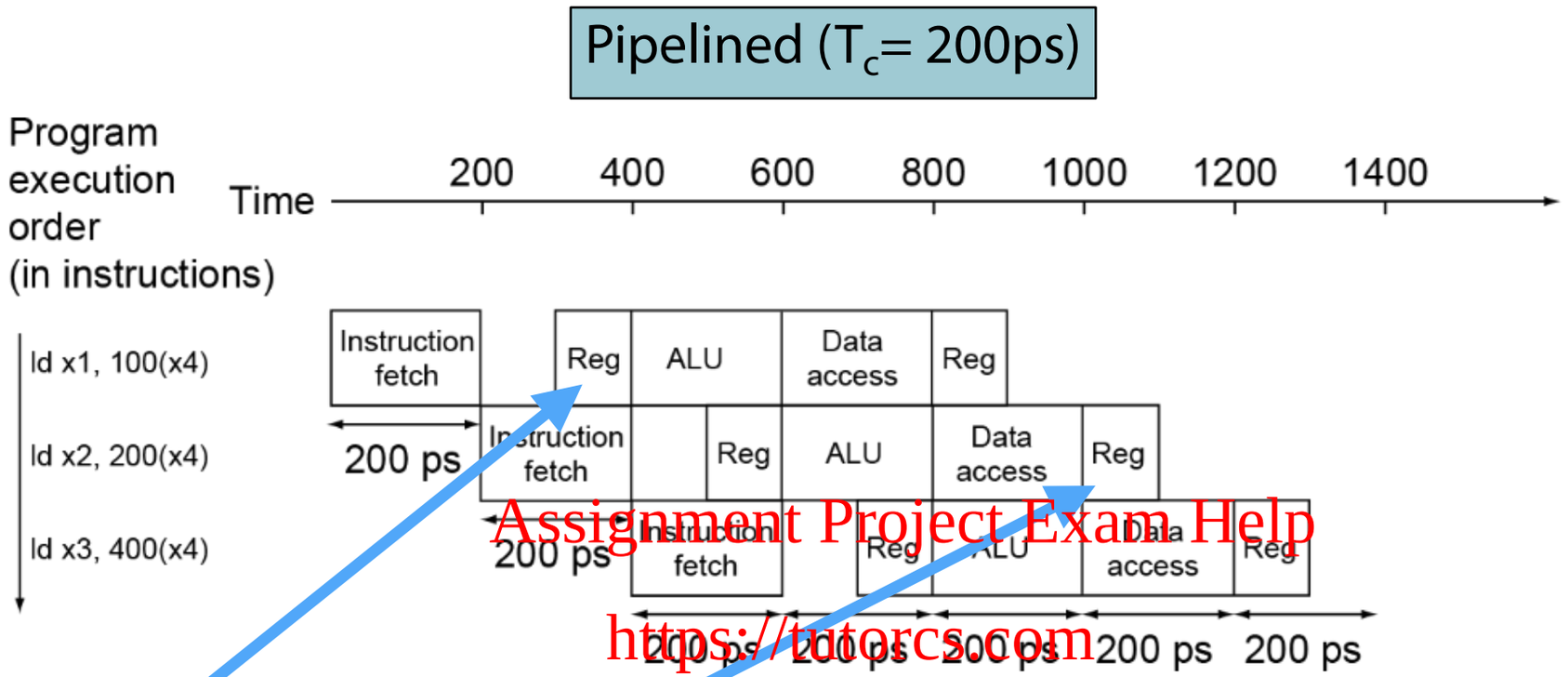


We Have No Time Machine



- For correctness, we can delay issuing the "red" instruction until we know we can get the result from the "black" instruction.

RISC-V write/read order



- The "read" phase of the register file reads in the second half of the cycle
- The "write" phase of the register file writes in the first half of the cycle
- Thus an instruction can write a result then read the same result in the same clock cycle. (This is good!)

Hazards

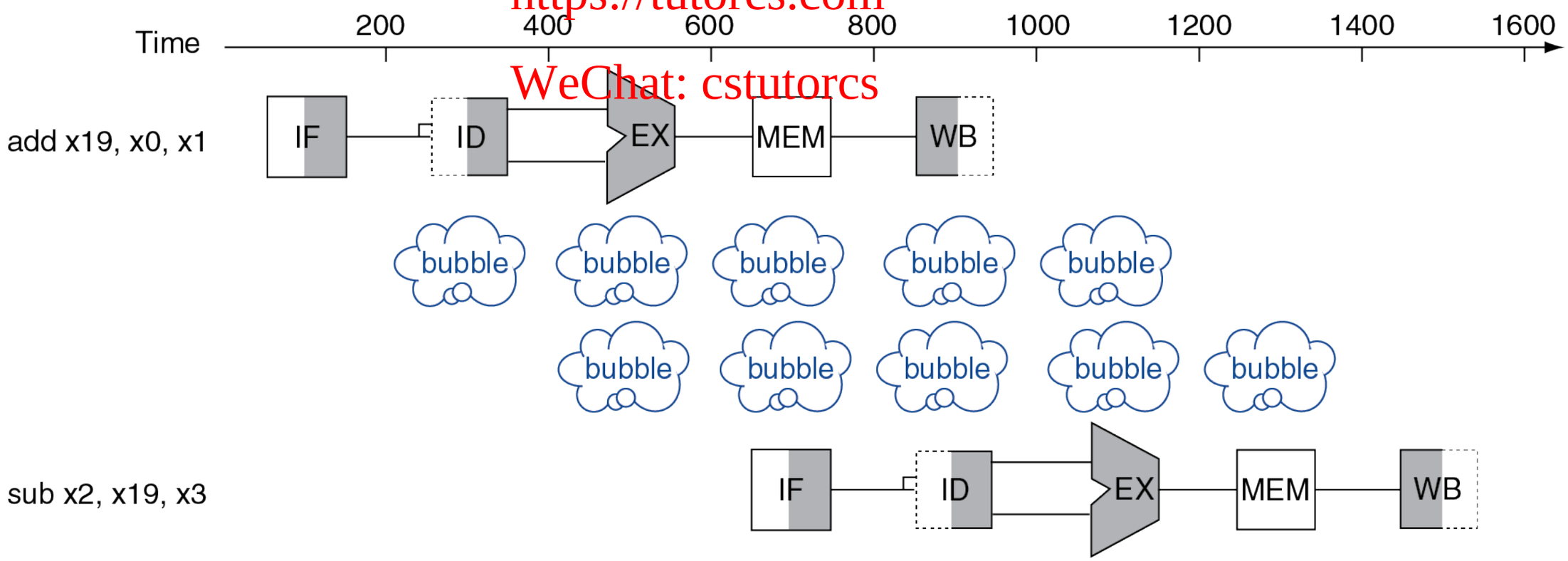
- Situations that prevent starting the next instruction in the next cycle
- Structural hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction
- We can always resolve a hazard by waiting

Structural Hazards

- Conflict for use of a resource
- Example: In 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"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- General strategy for addressing structural hazard: replicate

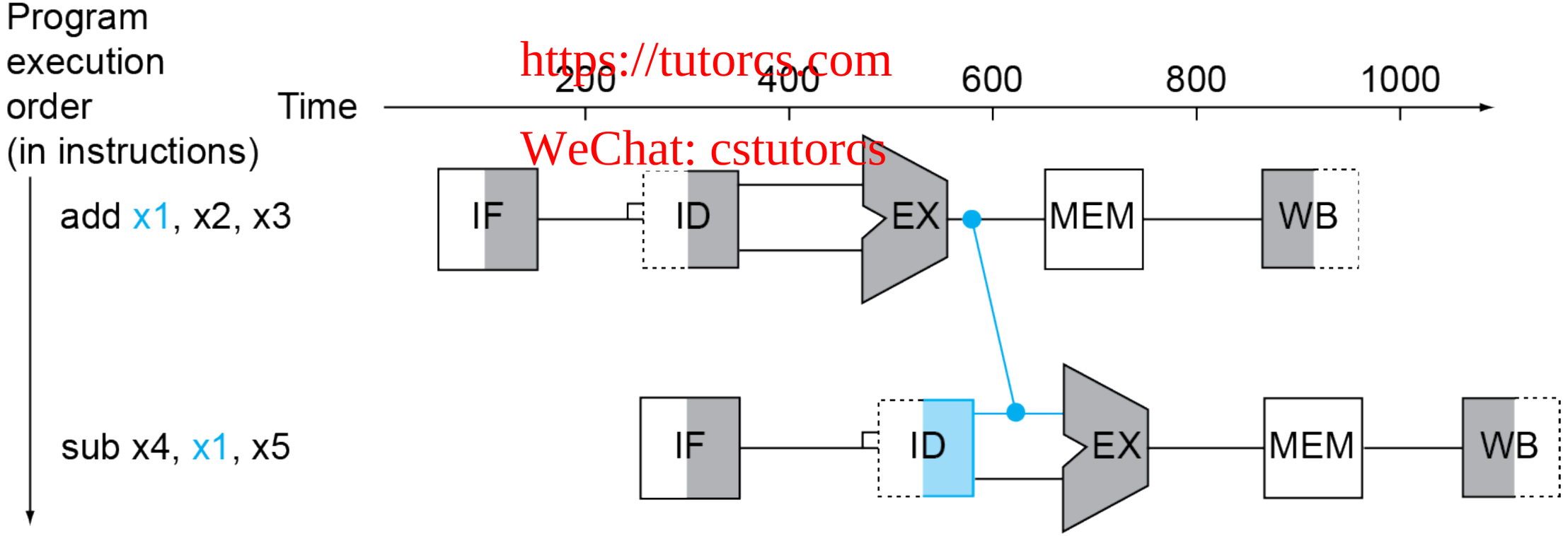
Data Hazards

- **An instruction depends on completion of data access by a previous instruction ("data dependency")**
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`



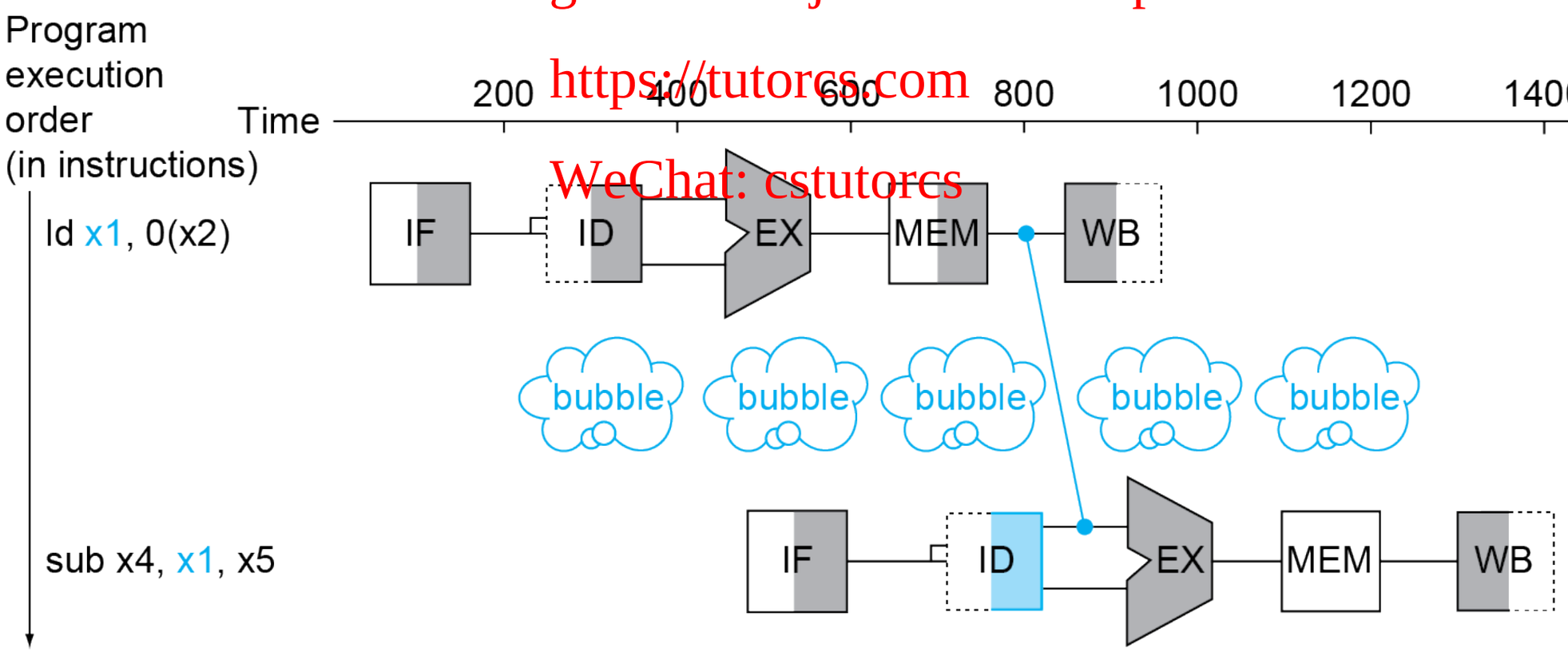
Forwarding (aka Bypassing)

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



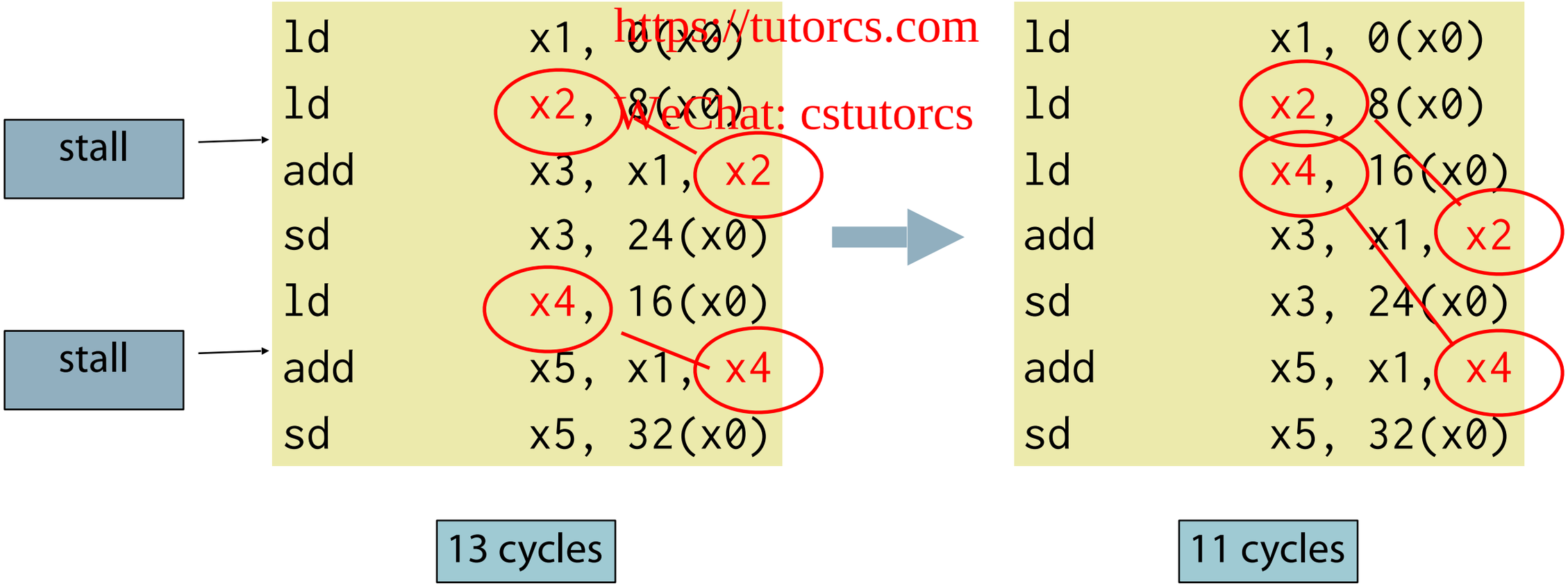
Load-Use Data Hazard

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

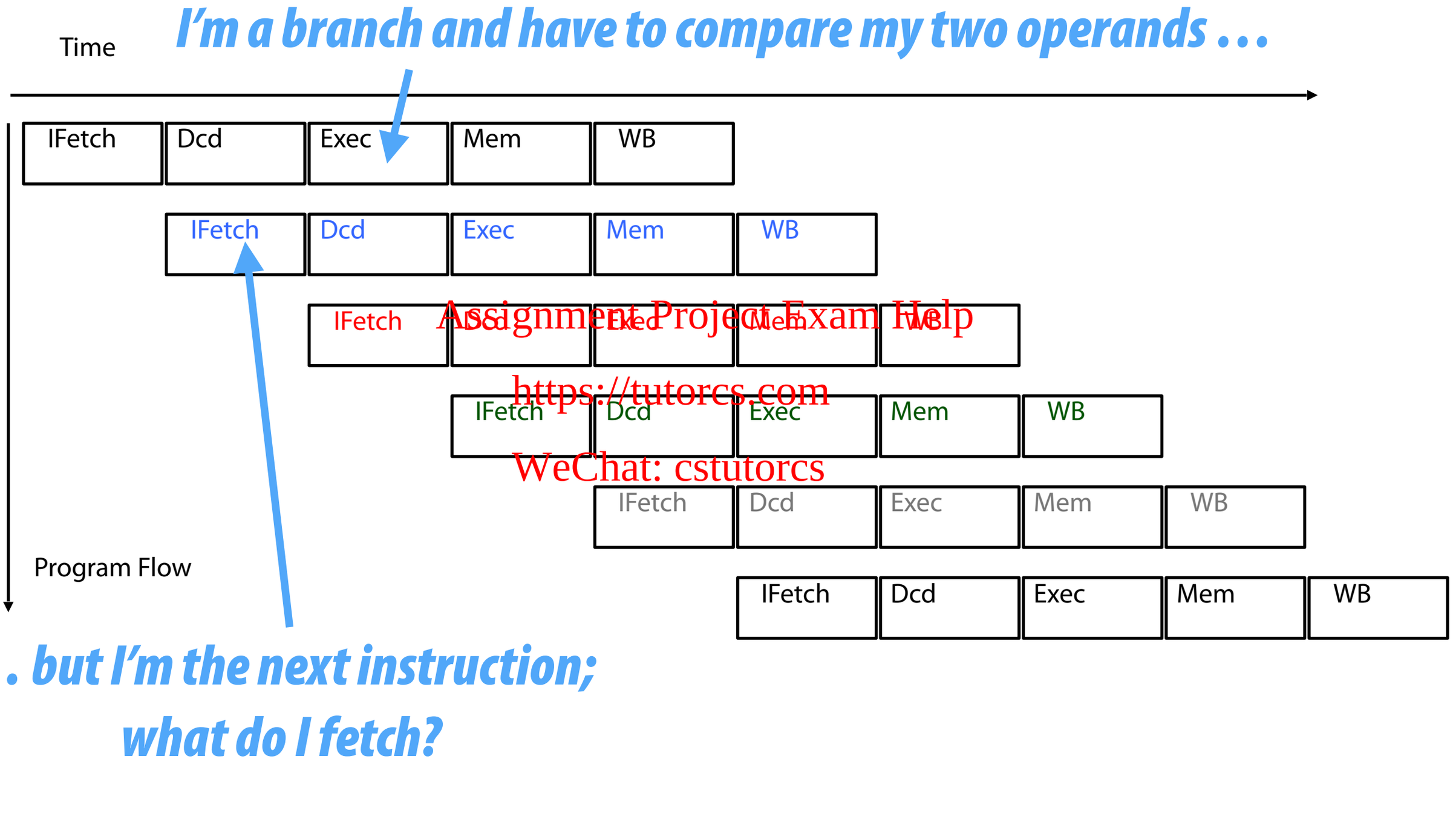


Code Scheduling to Avoid Stalls

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



We Have No Time Machine



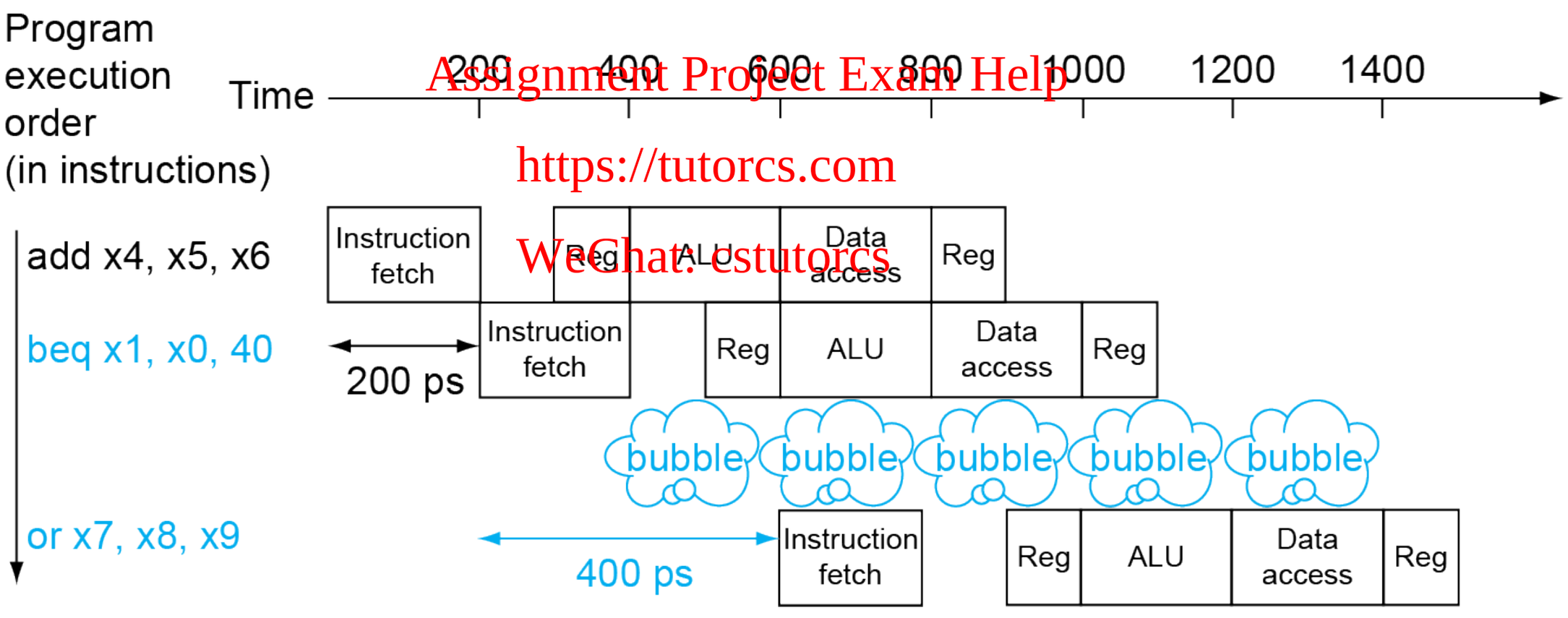
Control Hazards

- 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 RISC-V 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

- There's a lot of work to evaluate a branch!
- High-performance pipelines are usually > 5 cycles
- 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
- If we have time later in the course, we'll spend more time on this

Pipeline Summary

- 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

RISC-V Pipelined Datapath