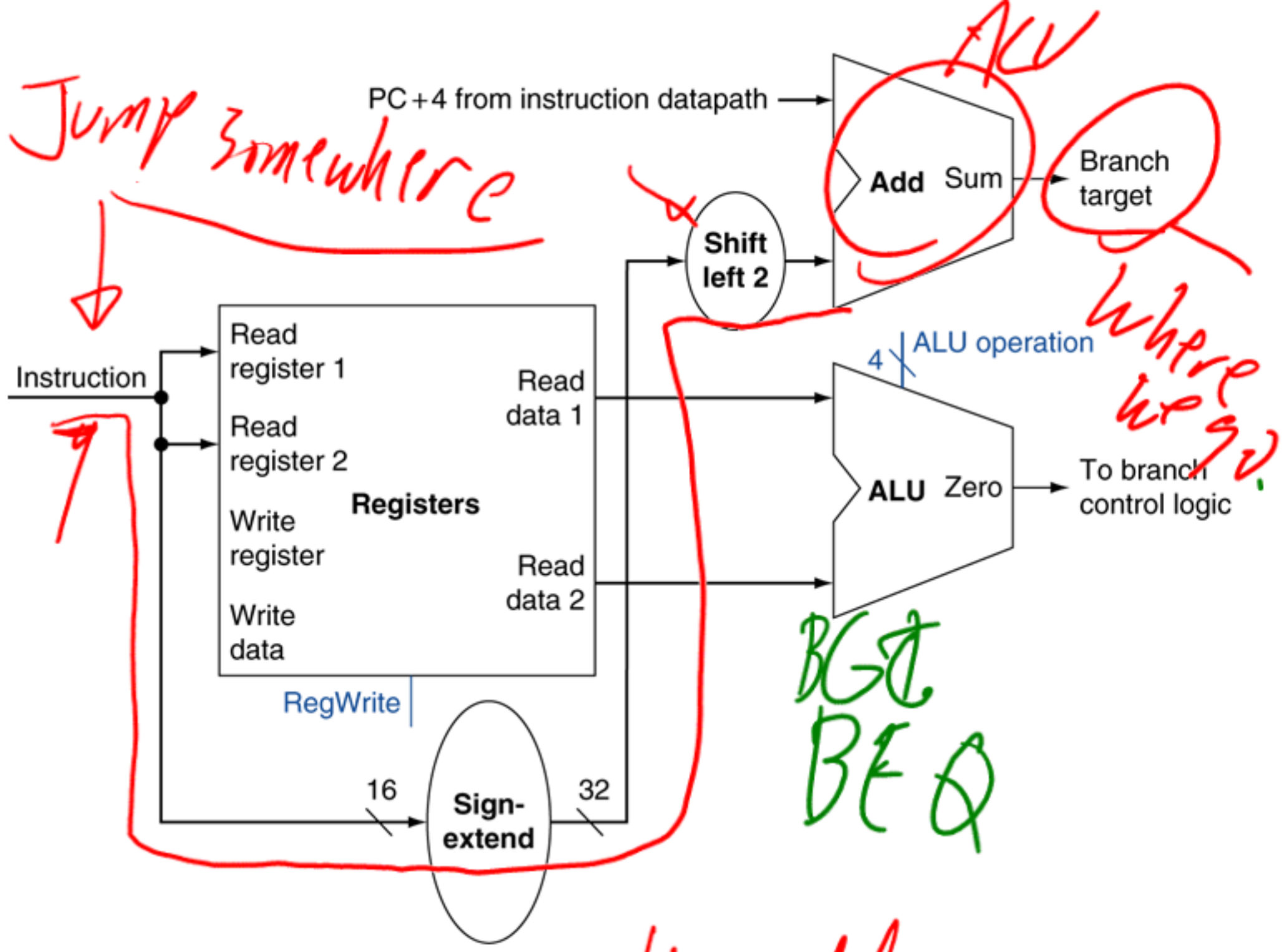
Pipelining

Lecture Objectives:

1) Define pipelining
2) Calculate the speedup achieved by pipelining for a given number of instructions.
3) Define how pipelining improves computer performance.
4) Define structural hazard, data hazard, and control hazard.
5) Define the term stall.
6) Explain the concept of forwarding.

How do we manage a jump?



Relative address:

- The destination address is dependent upon the current program counter.

⇒ Used on branches

⇒

Branch Instructions

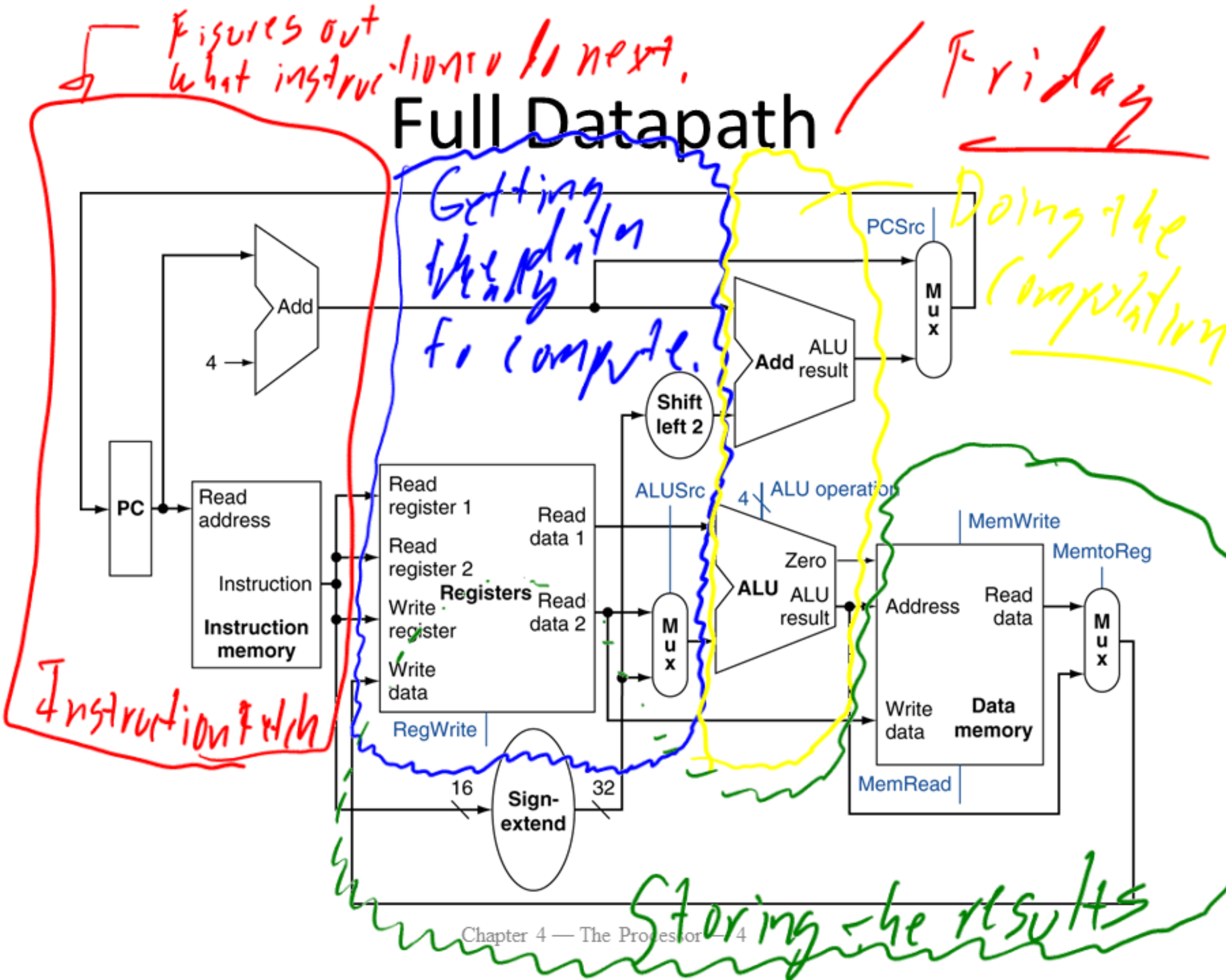
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branches are in terms of instructions, not bytes.

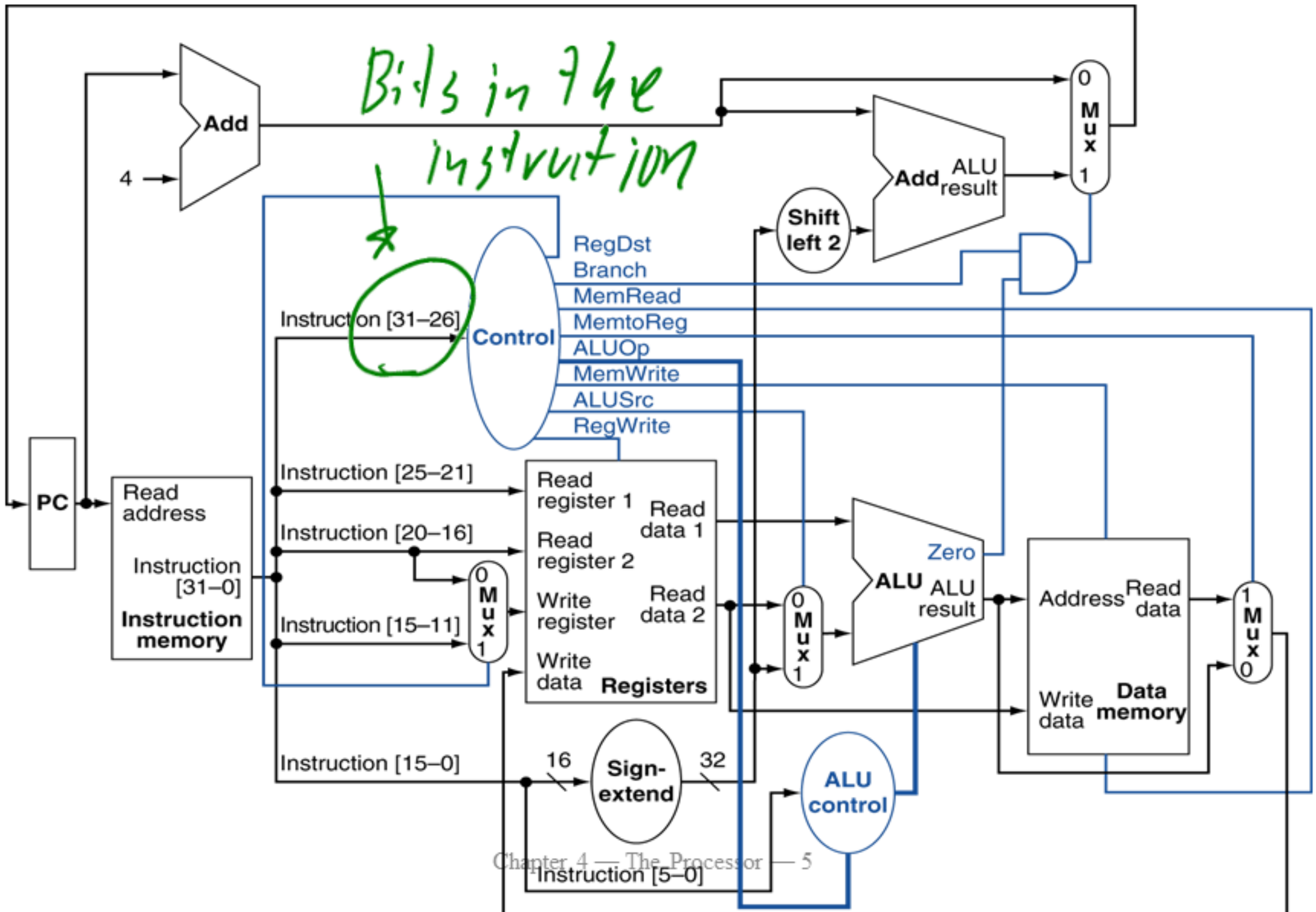
Adder exists.

Multiply by 4.

Full Datapath



Datapath With Control



Automotive Assembly

- Build body
- Paint body
- Build engine assembly
- Attach engine assembly
- Add interior lining
- Add interior seats
- Add dash board
- Test vehicle

Assembly Line

Definitions

- Pipelining
  - An implementation technique in which multiple instructions are overlapped in execution.
- Structural Hazard
  - When a planned instruction can not execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- Control Hazard (aka Branch Hazard)
  - When the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed
- Data Hazard
  - When a planned instruction can not execute in the proper clock cycle because data that is needed to execute the instruction is not yet available

MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
     Figures out the next instruction.
  2. ID: Instruction decode & register read
     => Determine what to do w/ instruction.
  3. EX: Execute operation or calculate address
     => Do it.
     ADD / Subtract etc.
     Branch / Jump.
  4. MEM: Access memory operand
     Read / writing to RAM.
  5. WB: Write result back to register
     Stores results in a register.

Lets compare performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

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

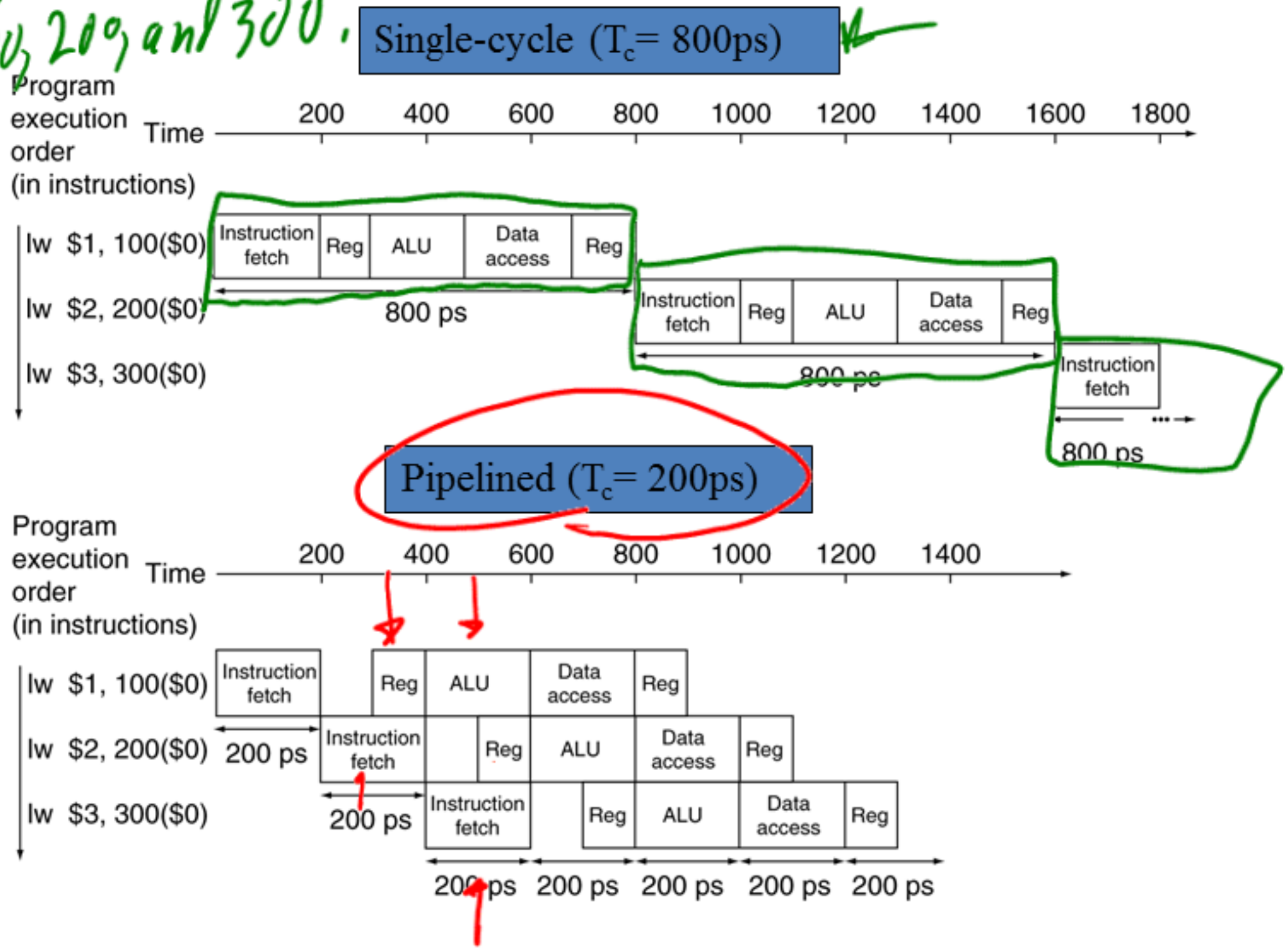
Loading a word in memory

time

Getting the instruction

Faster

Pipeline Performance



Another example

```
Li $t0, 55
Li $t1,53
Li $t3, 51
Addi $t0, $t0, 5
Addi $t2, $t2, -1
```

Pipeline speedup

- Ideal conditions
  - All stages are balanced

All stages take same time

$$\text{Time Between Instructions}_{\text{Pipelined}} = \frac{\text{Time Between Instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- If not balanced, speedup is less

More instructions per unit time.

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Start to finish time.

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ =
    Time between instructions$_{nonpipelined}$ / 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
  - 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

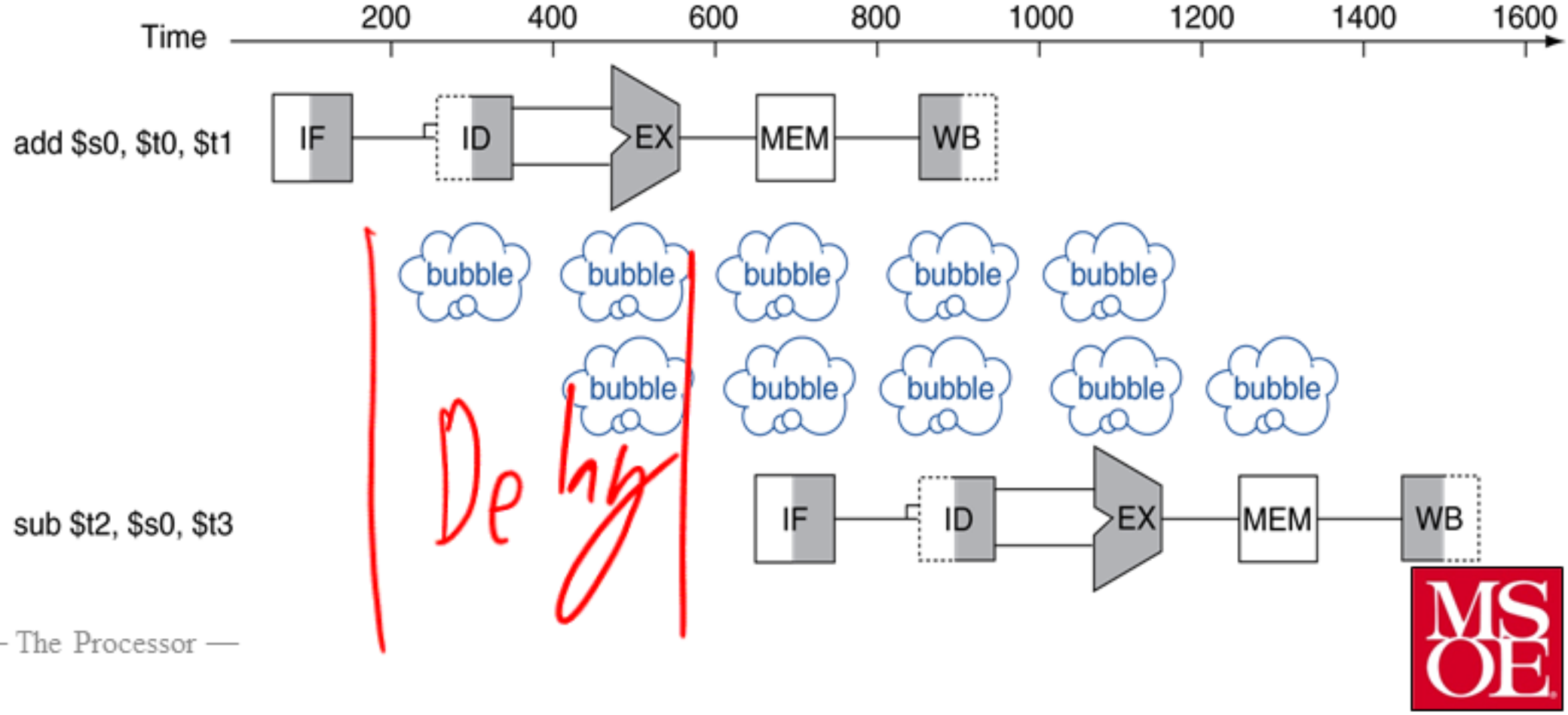
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In 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

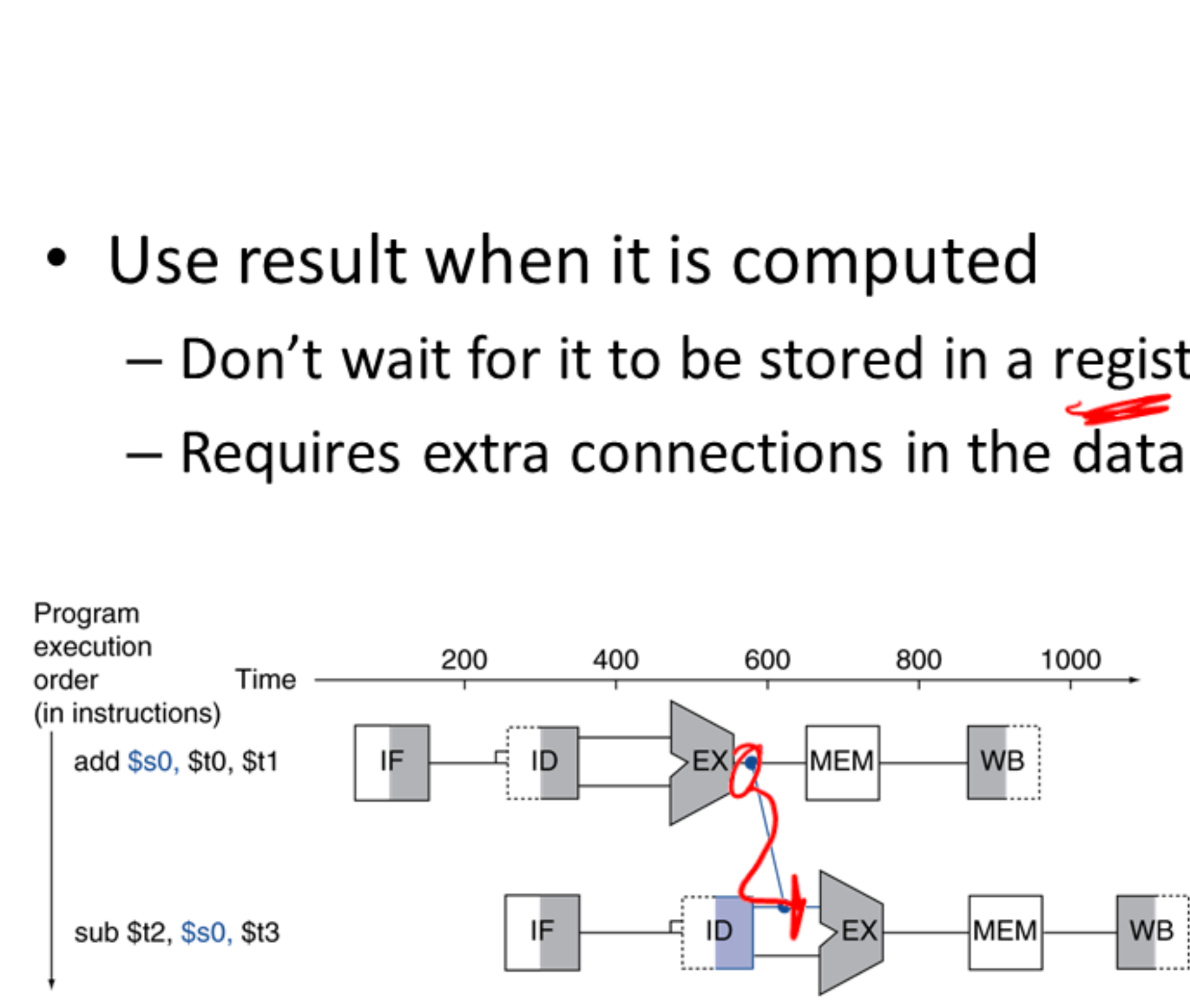
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
    sub $t2, $s0, $t3



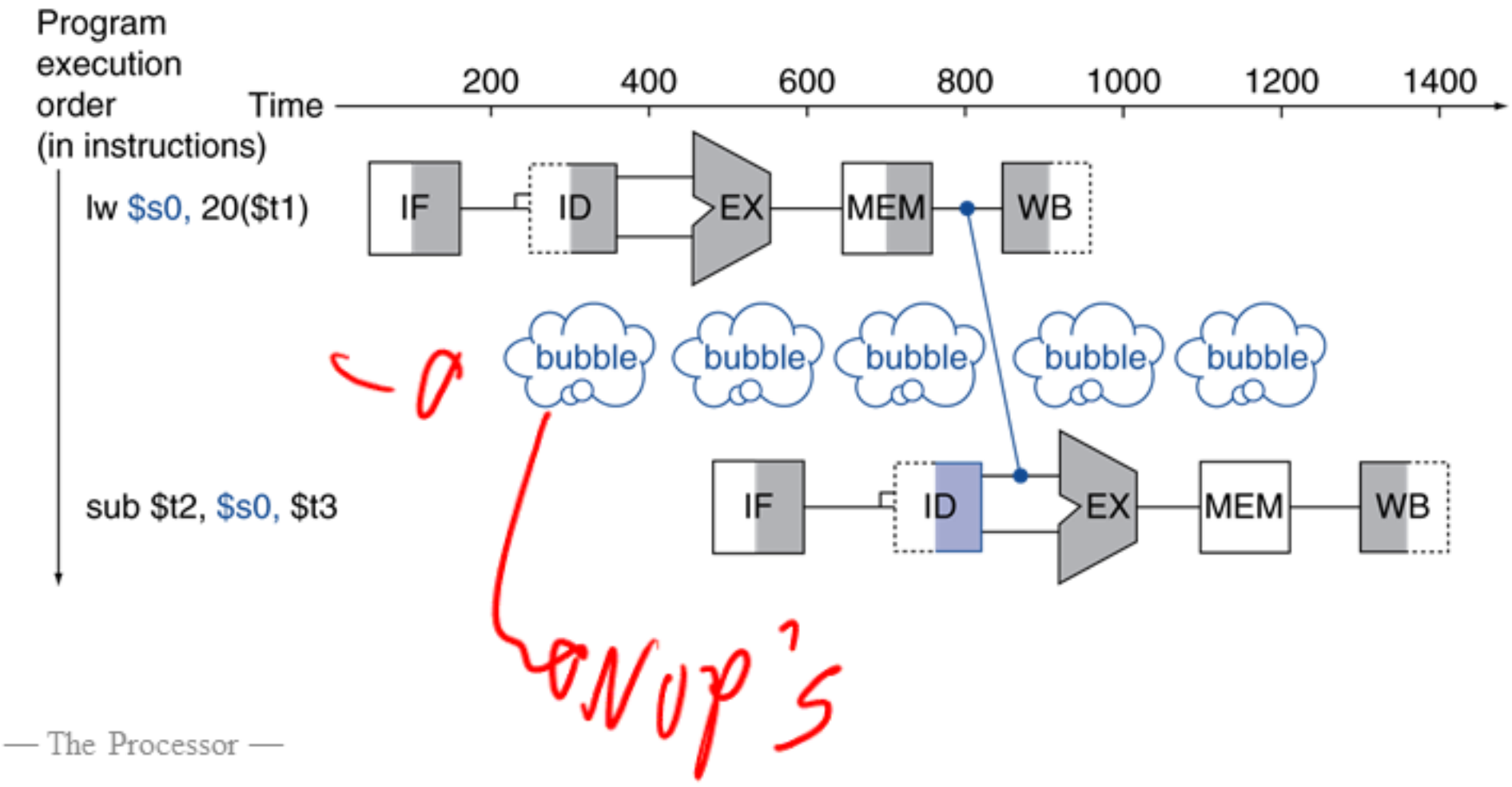
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - 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;

stall

stall

```
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)
```

13 cycles

```
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)
```

11 cycles

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 MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Branch Stall on

- Wait until branch outcome determined before fetching next instruction