# 154B Discussion 4

February 1st, 2023

# Goals

- Assignment 3.1
  - Single cycle CPU -> Pipelined CPU.
  - Instruction life cycle.
  - Stage registers.
  - Debugging pipelined CPU.
- Control hazards & data hazards.
- Week 4 Quiz.

# Logistics

- Assignment 3 template updated.
  - Fixed the diagram: the wire forwarding value from WB to EX.
  - Fixed hexadecimal value errors of the single stepper. Incorrectly displayed only 32 bits out of 64 bits of a 64-bit integer.
  - The piazza assignment 3 post has instructions on how to update the single stepper.

    https://piazza.com/class/lcf5v5re7q51ky/post/90

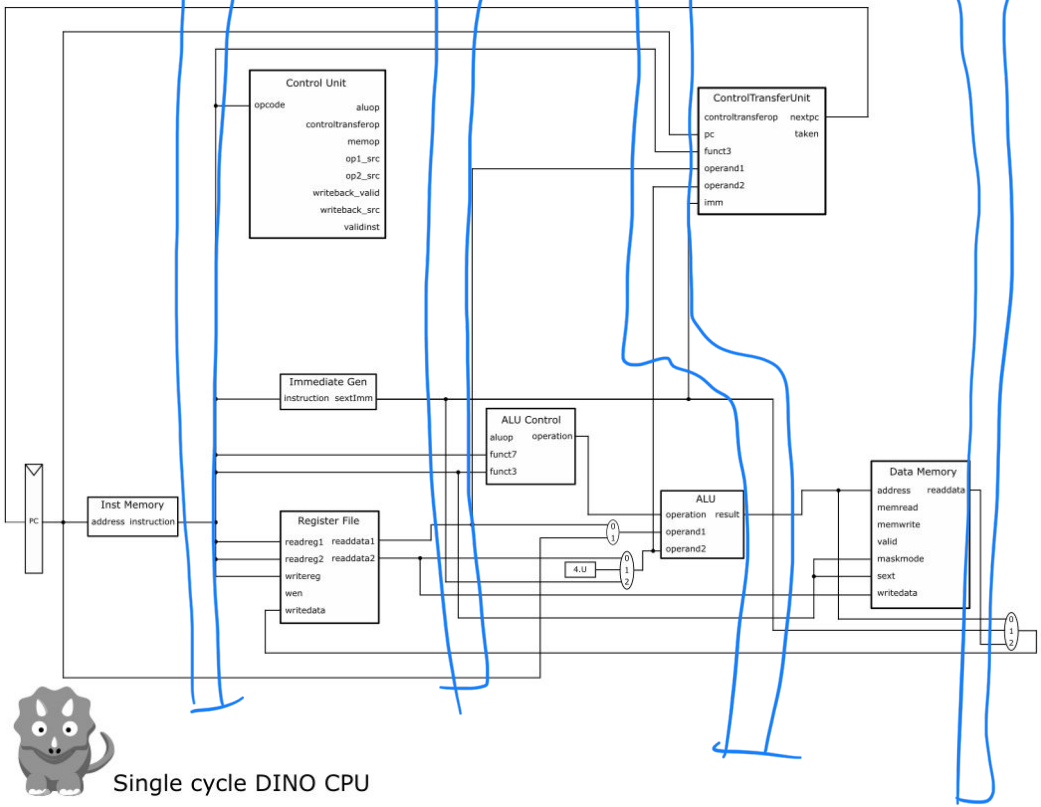
# Assignment 3: 5-stage Pipelined CPU

- Has 5 stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), Writeback (WB).
- Each stage has a different instruction.

# Assignment 3: 5-stage Pipelined CPU

- Has 5 stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), Writeback (WB).
- Each stage has a different instruction.
- DINOCPU pipelined CPU is structurally similar to the single cycle CPU.
  - Most components are the same.
  - Additional components for resolving control hazards and data hazards.

# Assignment 3: Single Cycle CPU -> Pipelined CPU



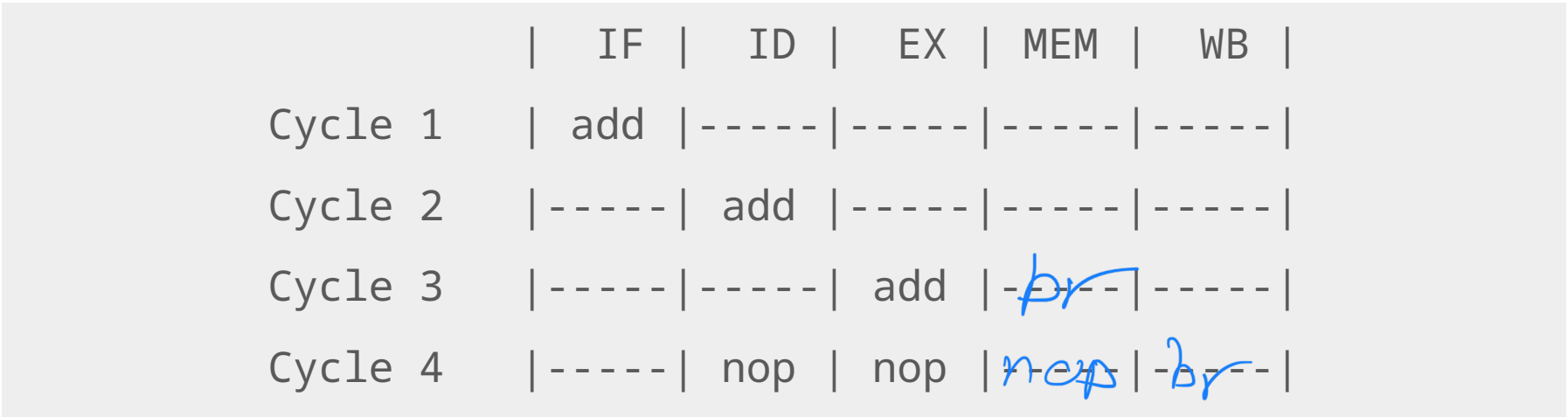
# Assignment 3: Instruction Life Cycle

- Every instruction must go through **all** stages **sequentially**; except when an instruction is flushed due to hazards, it is removed from the pipeline.
- E.g., an ADD instruction must go through the MEM stage even though it does not update the memory.

| | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| Cycle 1 | add | | | | |
| Cycle 2 | | add | | | |
| Cycle 3 | | | add | | |
| Cycle 4 | | | | add | |
| Cycle 5 | | | | | add |

# Assignment 3: Instruction Life Cycle

- Every instruction must go through **all** stages **sequentially**; except when an instruction is flushed due to hazards, it is removed from the pipeline.
- NOP: an instruction that does nothing. In RISC-V, it's equivalent to addi x0, x0, 0.
- E.g., an ADD instruction is flush during EX stage.

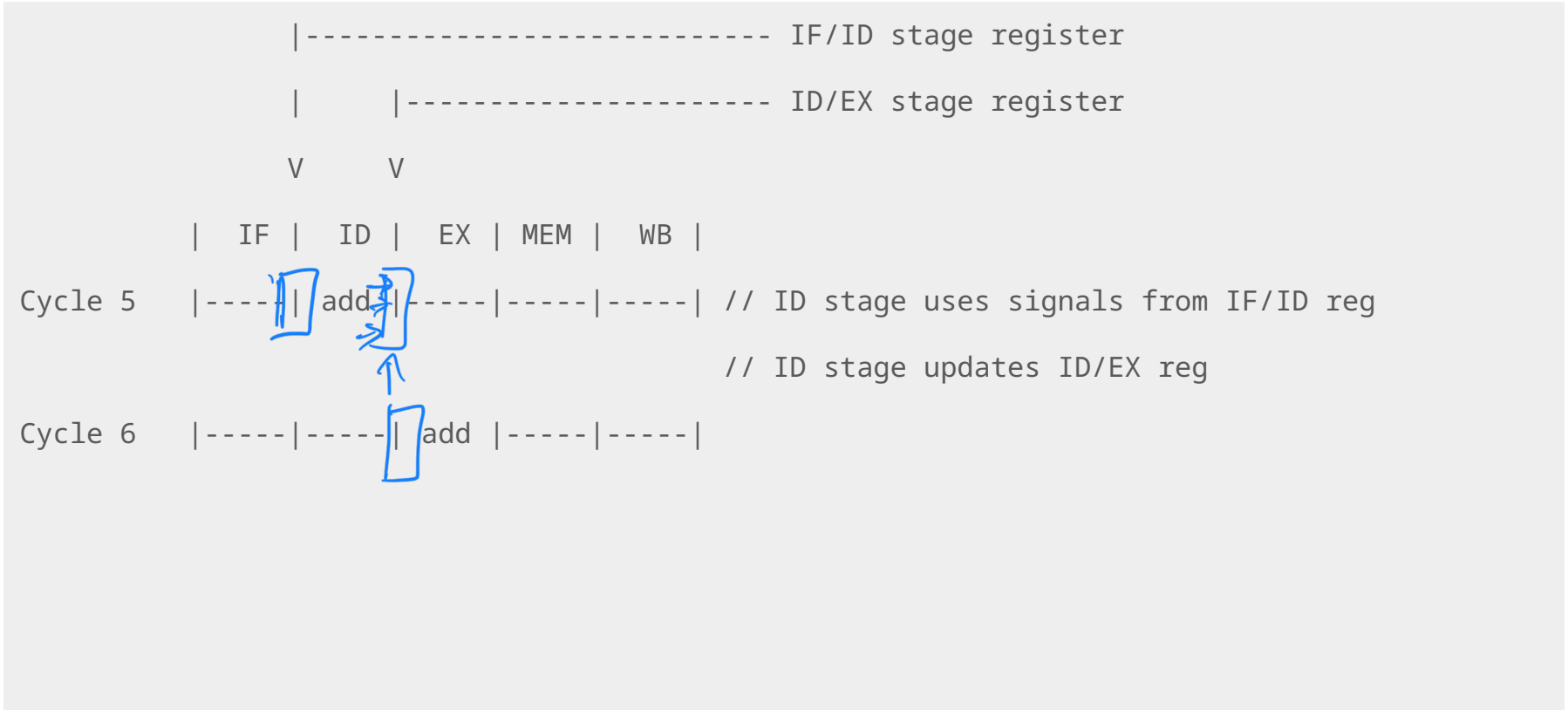


# Assignment 3: Stage Registers

- Transferring the instruction execution context between stages.
- We use 7 stage registers:
  - IF/ID
  - ID/EX
  - ID/EX_ctrl: contains control signals used in the EX stage.
  - EX/MEM
  - EX/MEM_ctrl: contains control signals used in the MEM stage.
  - MEM/WB
  - MEM/WB_ctrl: contains control signals used in the WB stage.
- For debugging purposes, you can add more stage registers.

inst, ctrl signals, pc, ....
[ for MEM stage

# Assignment 3: Stage Registers



# Assignment 3: Stage Register Interface

- They are registers.
- E.g., to update the ID/EX register in the ID stage,

```
id_ex.io.in.sextImm := immGen.io.sextImm
```

- E.g., to use the ID/EX register in the EX stage,

```
controlTransfer.io.imm := id_ex.io.data.sextImm
```

# Assignment 3: Stage Register Interface

stall
flush

- `stage_reg.io.valid:`
  - if true.B, the stage_reg will be updated accordingly to the inputs, i.e., signals from stage_reg.io.in.*
  - if false.B, the stage_reg will not take stage_reg.io.in.* as an input.
- `stage_reg.io.flush:`
  - if true.B, the stage_reg will be zeroed out (equivalent to NOP).
  - if false.B, the stage_reg will not be zeroed out.

| | valid === true.B | valid === false.B |
|---|---|---|
| flush === true.B | zeroed out | zeroed out |
| flush === false.B | stage_reg.io.in.* as input | stage_reg stays the same |

# Assignment 3: Debugging pipelined CPU

- There are 5 instructions in the pipeline at the same time.
- The PC outputted by the single stepper is the PC of the Fetch (IF) stage.
- You can add PC to stage_reg registers to keep track of the PC of instruction at every stage.
- Using both printf and single stepper to keep track the state of the CPU at every cycle.

# Assignment 3: Debugging pipelined CPU

- Regardless of CPU microarchitectures, any in-order CPU must execute instructions in-order (i.e., instructions must be executed in the same order as the *program order*).
- The single cycle CPU and the pipelined CPU are in-order CPUs.
- This means,
  - The order of instructions executed by the single cycle CPU must be exactly the same as the order of instructions appears in the writeback stage of the pipelined CPU.
  - You can generate the correct order (either the order of PC or the order of instructions; PC order might be more useful) from the single cycle CPU, then compare that to the order of instructions appears in the writeback stage of the pipelined CPU. This is useful for debugging big applications.

# Program Order in Out-of-order Execution

- Program Order: the order of instruction execution that is determined by software (users, compilers).
- In-order execution: CPU executes instructions as specified by program order.
- Out-of-order Execution: instructions might be executed out-of-order; however, the instructions must be committed in-order. I.e., regardless of in-order/out-of-order type of execution, the instructions must update the state of the system (updating registers, updating memory, etc.) in order.
    - *Commit (or retire)*: the instruction actually affects the state of the system (updating registers, updating memory, etc.)

# Week 4 Quiz

- Let’s not assume the DINOCPU, but any pipelined CPU.
- The cycle time of a pipelined CPU is the duration of the *longest* stage.
- An instruction cannot enter the execution stage when,
  - The instruction reads from a register that is being written to by one of instructions in the execution stage.
  - The instruction is a load, and there is at least one store instruction in the execution stage.
    - Why? Before execution, we don’t know the effective address of the load, so we don’t know if the load effective address overlap with any of the store execution address.

# Week 4 Quiz

Data hazards: RAW, WAW, WAR

* RAW rd rs1 rs2/imm

inst1: add t0, t1, t2 write to t0

inst2: addi t3, t0, 1 read from t0

in this example:
inst 1&2 : RAW
inst 2&3 : WAR
inst 1&3 : WAW

* WAW rd rs1 rs2/imm

inst1 add t0, t1, t2 -> write to t0 (physical reg 1)
inst 2 add t4, t0, t5
inst 3 addi t0, t3, 1 -> write to t0 (physical reg 2)

# Week 4 Quiz

WAR

inst 2 add t4, t0, t5

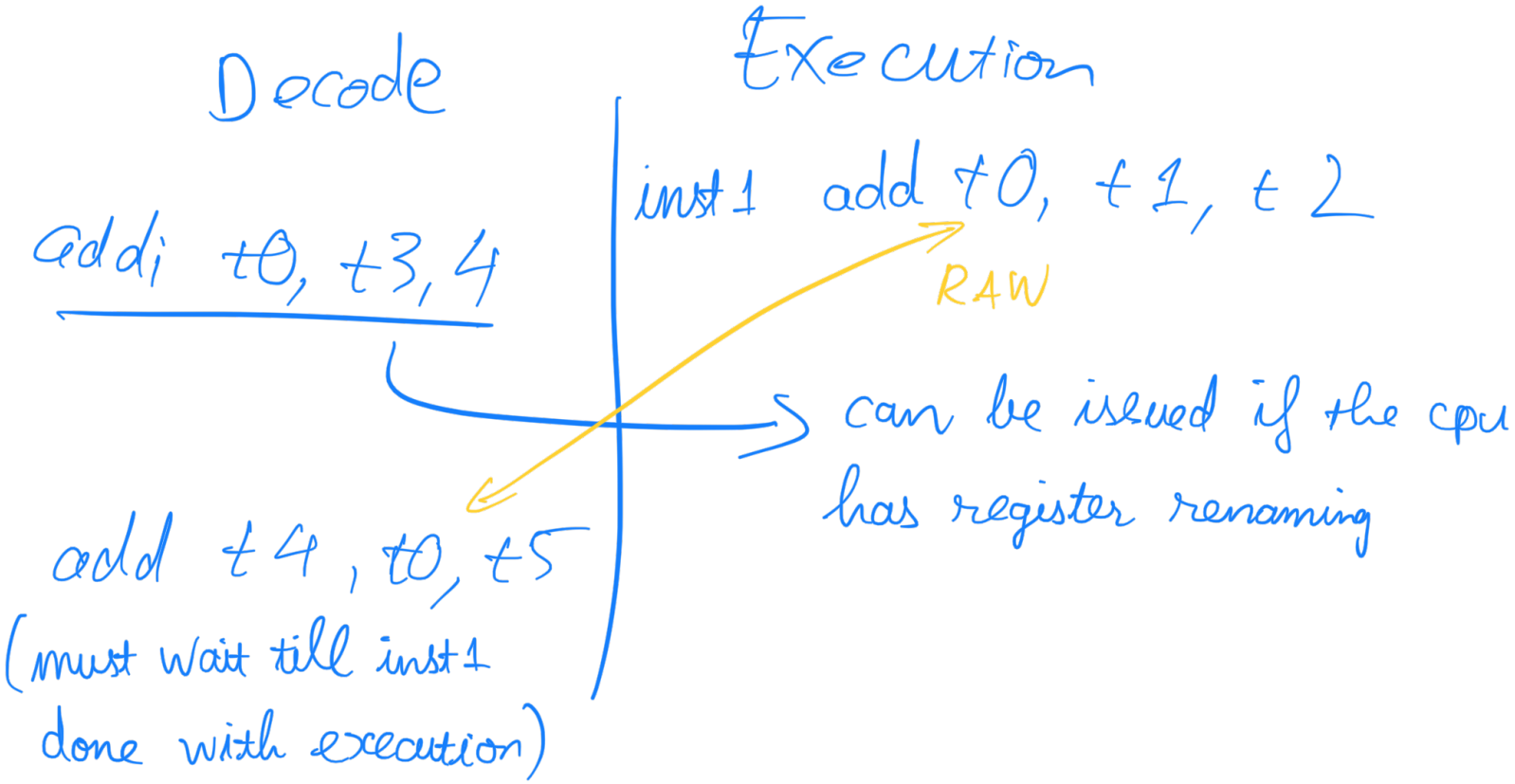
inst 3 addi t0, t3, 1

Mitigation: register renaming

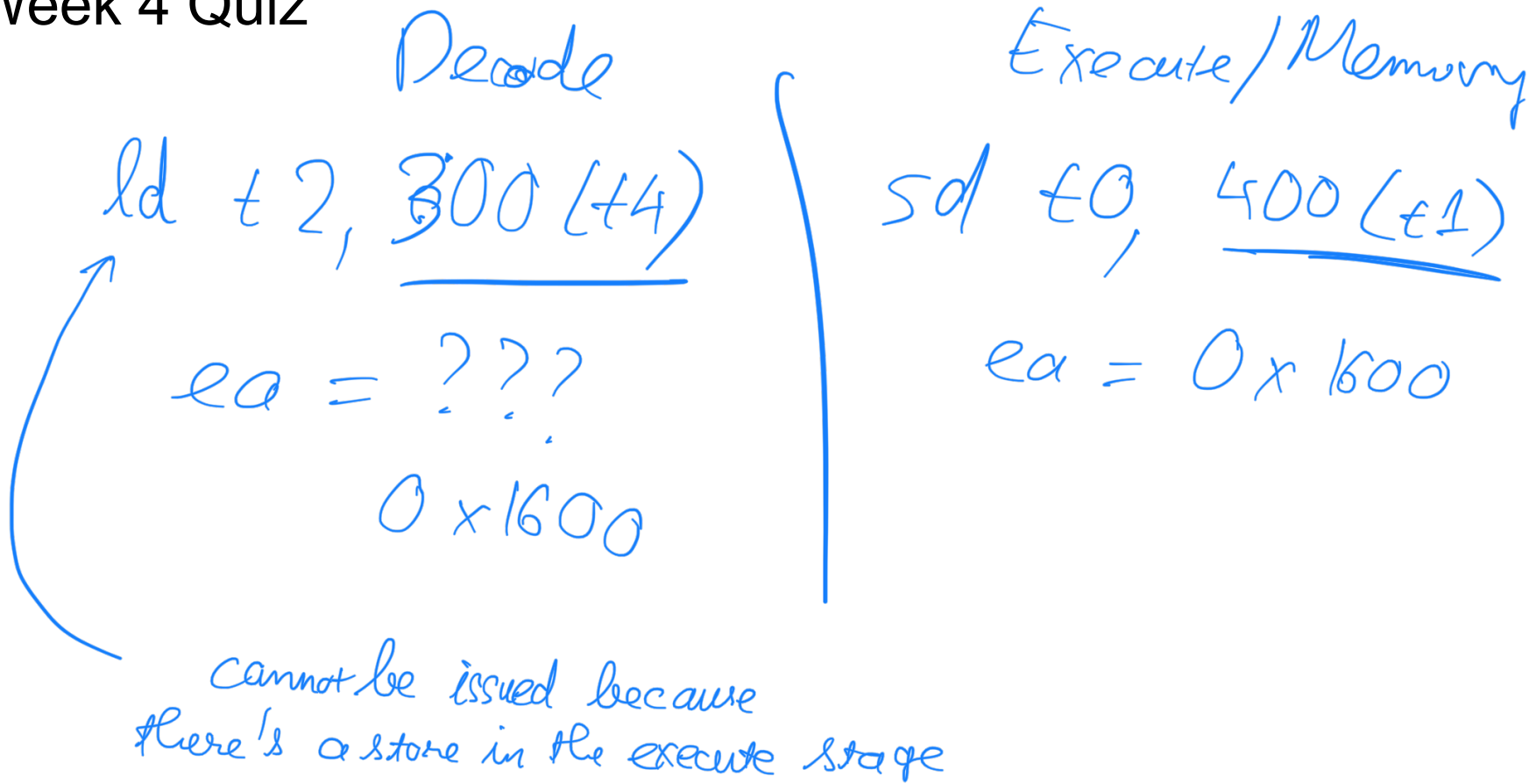
read from t0 (phys reg 1)

write to t0 (physreg 2)

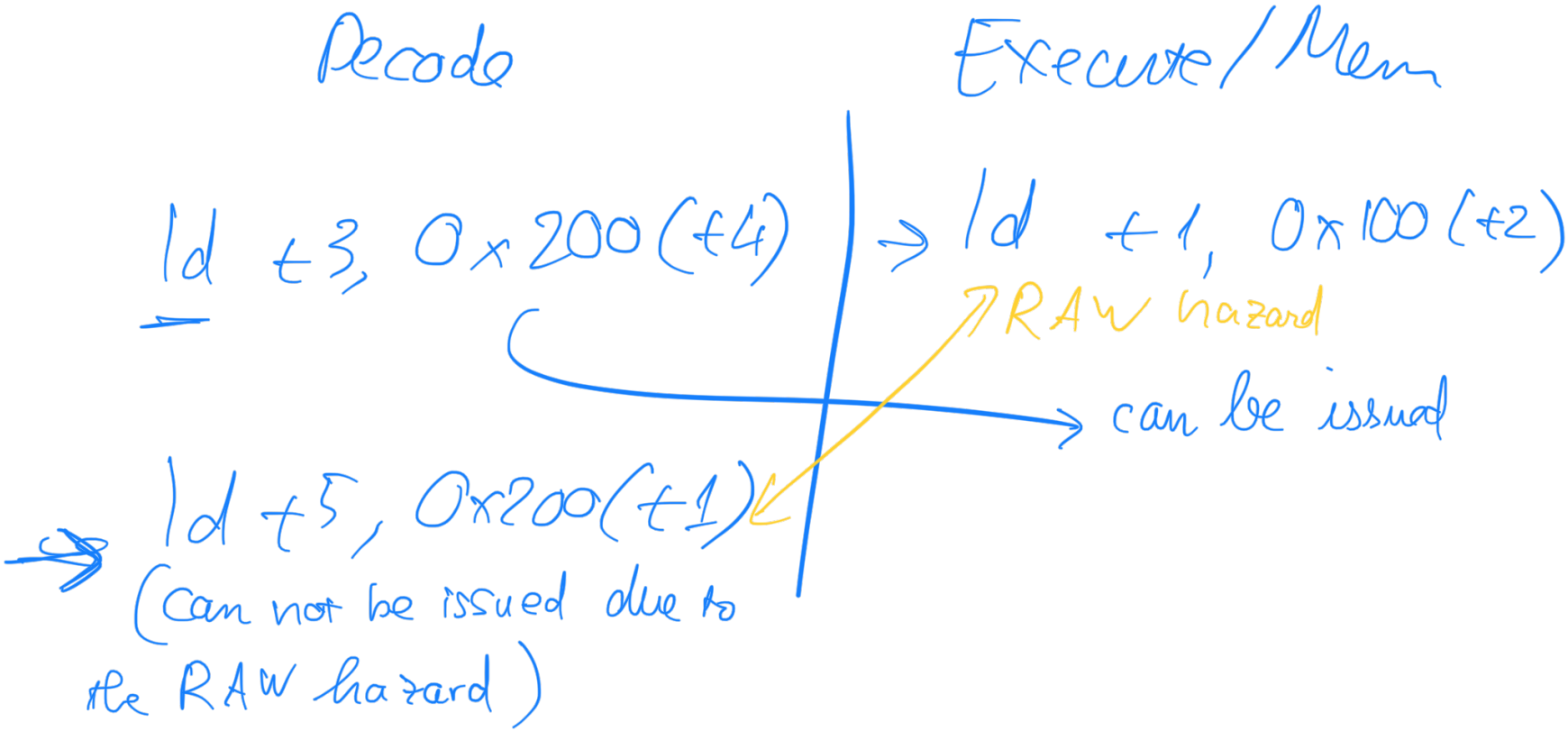
# Week 4 Quiz


Decode
Execution
addi t0, t3, 4
inst 1 add t0, t1, t2
RAW
can be issued if the cpu has register renaming
add t4, t0, t5
(must wait till inst 1 done with execution)

# Week 4 Quiz


Decode
ld t2, 300(t4)
ea = ???
0x1600
Execute/Memory
sd t0, 400(t1)
ea = 0x1600
cannot be issued because
there's a store in the execute stage

# Week 4 Quiz


Decode
Execute / Mem
ld t3, 0x200(t4)
ld t1, 0x100(t2)
RAW hazard
can be issued
ld t5, 0x200(t1)
(Can not be issued due to the RAW hazard)

# Week 4 Quiz

F | D | E | M | WB

600ps < 600ps < 600ps < 600ps < 600ps

$\#\ insts/second = (600 \cdot 10^{-12})^{-1}$

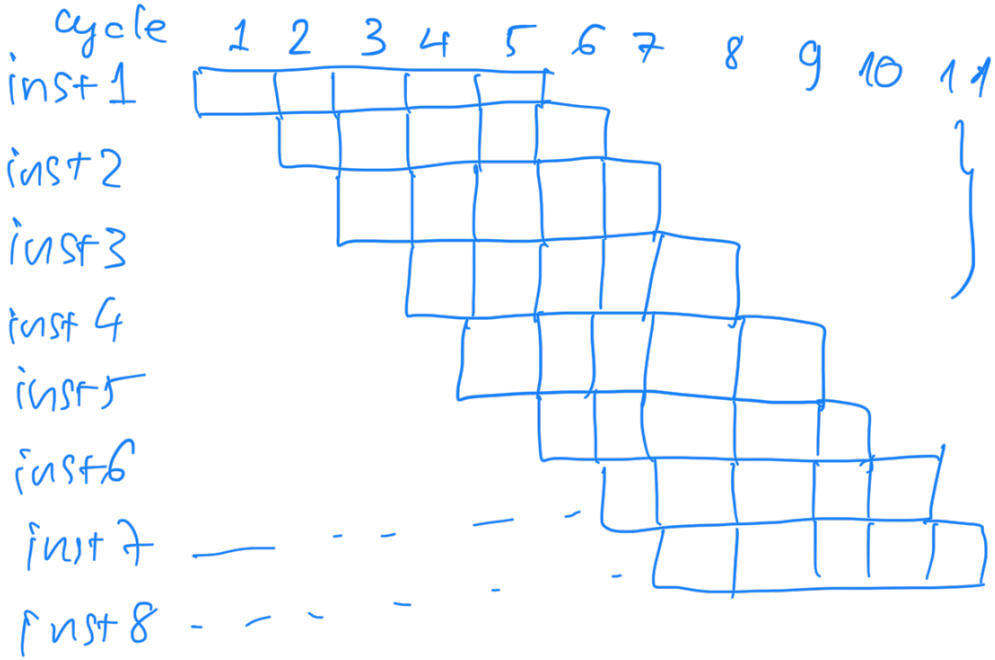
cycle time : $600\,ps = 600 \cdot 10^{-12}\,s$

$\rightarrow 600 \cdot 10^{-12}\,s/inst$ (this is because, in ideal cases, IPC of pipeline cpu is approximately 1 $\rightarrow$ CPI $\simeq$ 1)

$\rightarrow \frac{600 \cdot 10^{-12}\,s}{1\ cycle} \times \frac{1\ cycle}{1\ inst} = \frac{600 \cdot 10^{-12}\,s}{1\ inst}$

# Week 4 Quiz

## Pipeline CPU's IPC / CPI




cycle 1 → 4: no instruction finished (fan-in cycles)

→ Starting from cycle 5, there is 1 instruction finished per cycle (assuming there's no stalling / flushing)

→ So, in the ideal cases, (no stall/flush), N instructions take N+4 cycles to finish.

So, $IPC = \frac{N}{N+4}$.

Usually, # insts of a program is way bigger than 4, so,

$IPC = \frac{N}{N+4} \approx \frac{N}{N} = 1$

→ $CPI \approx 1$

, Note that, each inst still needs 5 cycles to finish. So instruction latency is still 5 cycles.

# Week 4 Quiz

## Q 28 / Q 29

-> Branch & jump insts modify nextpc

=> the cpu doesn't know the correct nextpc till MEM stage (assuming textbook's 5-stage pipeline)

if cpu has a branch predictor

-> the cpu can use the branch predictor to speculate nextpc

-> if the speculation is incorrect, the pipeline is flushed

| | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| cycle 11 | inst 9 | br/j | --- | --- | --- |
| cycle 12 | inst 10 | inst 9 | br/j | --- | --- |
| cycle 13 | inst 11 | inst 10 | inst 9 | br/j | --- |
| cycle 14 | inst 31 | nop | nop | nop | br/j |

if the cpu doesn't have a branch predictor, it must wait till the branch/jump is resolve

| | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| cycle 11 | inst 9 | br/j | --- | --- | --- |
| cycle 12 | inst 9 | nop | br/j | --- | --- |
| cycle 13 | inst 9 | nop | nop | br/j | --- |
| cycle 14 | inst 31 | nop | nop | nop | br/j |