

# Midterm Review

February 6th, 2023

# Quizzes

- Some of quiz questions were discussed 154B discussion sessions.

[https://jlpteaching.github.io/comparch/img/154B\\_discussions/154B-discussion-2.pdf#page=15](https://jlpteaching.github.io/comparch/img/154B_discussions/154B-discussion-2.pdf#page=15)

[https://jlpteaching.github.io/comparch/img/154B\\_discussions/154B-discussion-3.pdf#page=23](https://jlpteaching.github.io/comparch/img/154B_discussions/154B-discussion-3.pdf#page=23)

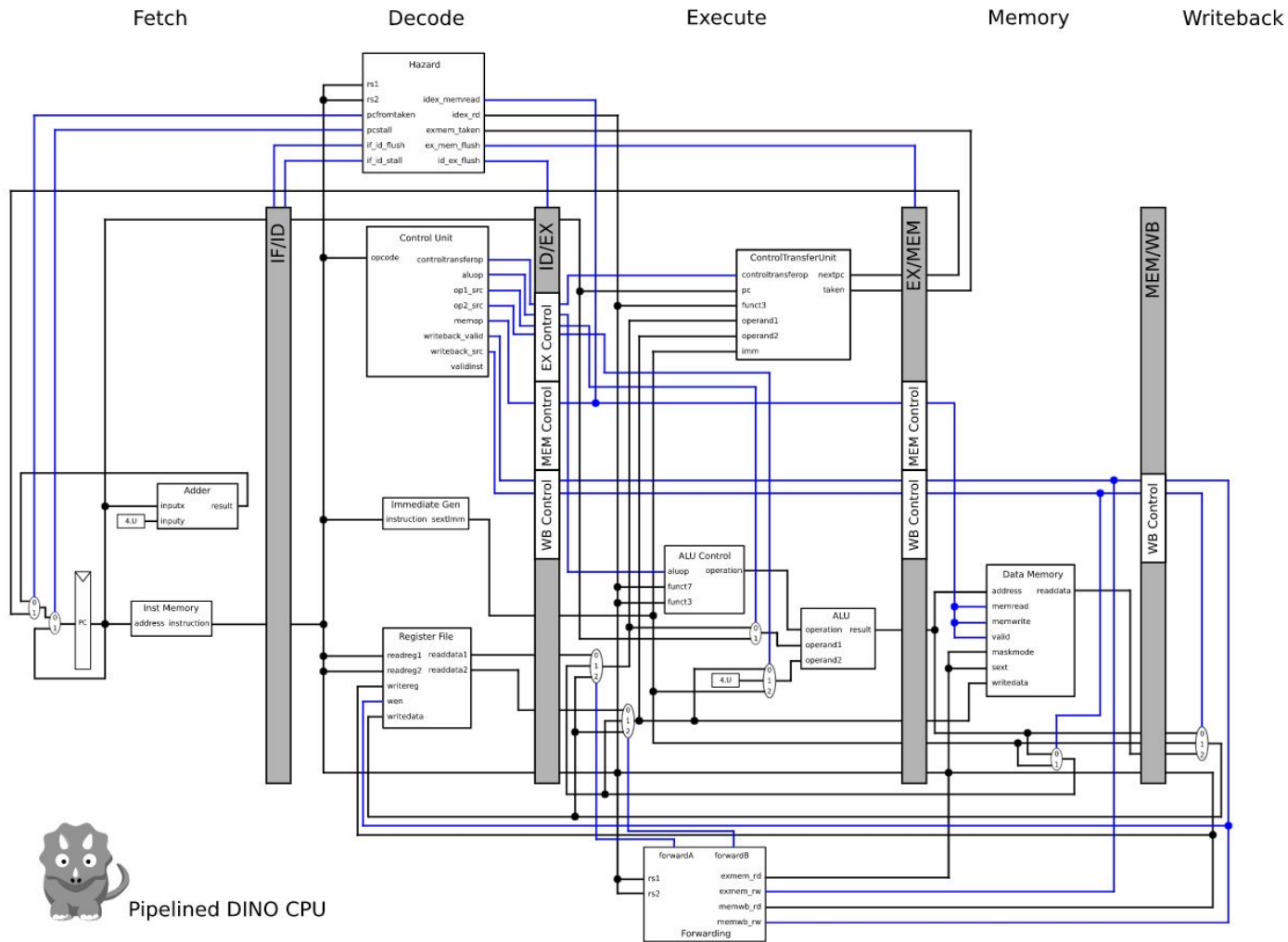
[https://jlpteaching.github.io/comparch/img/154B\\_discussions/154B-discussion-4.pdf#page=16](https://jlpteaching.github.io/comparch/img/154B_discussions/154B-discussion-4.pdf#page=16)

# Textbook's / DINO Pipelined CPU

- 5 stages:
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Execute (EX)
  - Memory (MEM)
  - Writeback (WB)
- Each stage only handles one instruction at a time.
- All instructions must go through all stages, unless they are flushed.

# Textbook's / DINO Pipelined CPU

- <https://jlpteaching.github.io/comparch/img/dinocpu/pipelined.svg>



Pipelined DINO CPU

# Textbook's / DINO Pipelined CPU

- What stage does what?
  - IF: fetches instruction according to the PC register.
  - ID: decodes an instruction, fetches values of registers from the register file, generates the immediate value.
  - EX:
    - Memory instructions: calculates the effective address.
    - Branch/Jump: calculates the next pc (the result of nextpc is only available to use in Memory stage).
    - Arithmetic instructions (R-types, I-types): calculates the result of instruction.
  - MEM:
    - Memory instructions: access data memory.
  - WB:
    - Writes the result back to the register file.

# Control Transfer Instructions

- Control transfer instructions
  - Branch instructions in RISC-V: BEQ, BNE, BLT, BGE, BLTU, BGEU (B prefix)
  - Jump instructions in RISC-V: JAL, JALR (J prefix)
- Branch instruction outcome is dependent on the value of registers.
- Jump instruction outcome is always taken.

# Control Transfer Instructions in Pipelined CPU

- Branch/Jump instruction causes Control hazards in pipelined CPUs because they alternate the next PC to values other than  $PC+4$ .
- Without branch predictor, the pipeline has to be stalled until the control transfer instruction reaches the Memory stage.



# Branch Predictor in Pipelined CPU

- Predict the branch outcome (taken or not taken) as well as the branch target (the nextpc) of all branch/jump instructions.
- Branch prediction is performed at the control transfer instruction's Decode stage.
- The actual outcome of the branch is known at the control transfer instruction's Memory stage.
- If the prediction is correct, no flush happens.
- If the prediction is incorrect, the instructions at Execute, Decode, Fetch stages are not supposed to be in the pipeline. They should be flushed.
- Flush: replacing an instruction with a NOP.
- NOP: an instruction that does nothing.

# Example: Pipelined CPU without a Branch Predictor

$0x10 + 0x14 = 0x24$

Cycle			1	2	3	4	5	6	7	8	9	10	11
PC=0x10 <u>bne</u> t1, t2, <u>0x14</u>			IF	<u>ID </u>	<u>EX </u>	<u>MEM </u>	WB						
PC=0x14 <u>add</u> t5, t3, t4				<u>IF </u>	<u>IF </u>	<del>IF </del>							
PC=0x18 xor t6, t3, t4													
PC=0x1c sub t1, t2, t3													
PC=0x20 and t2, t3, t3													
PC=0x24 or t4, t1, t6							<u>IF </u>	ID	EX	MEM	WB		
PC=0x28 mul t1, t2, t3								IF	ID	EX	MEM	WB	

# Example: Pipelined CPU with a Branch Predictor

↙ branch ~~not~~ taken

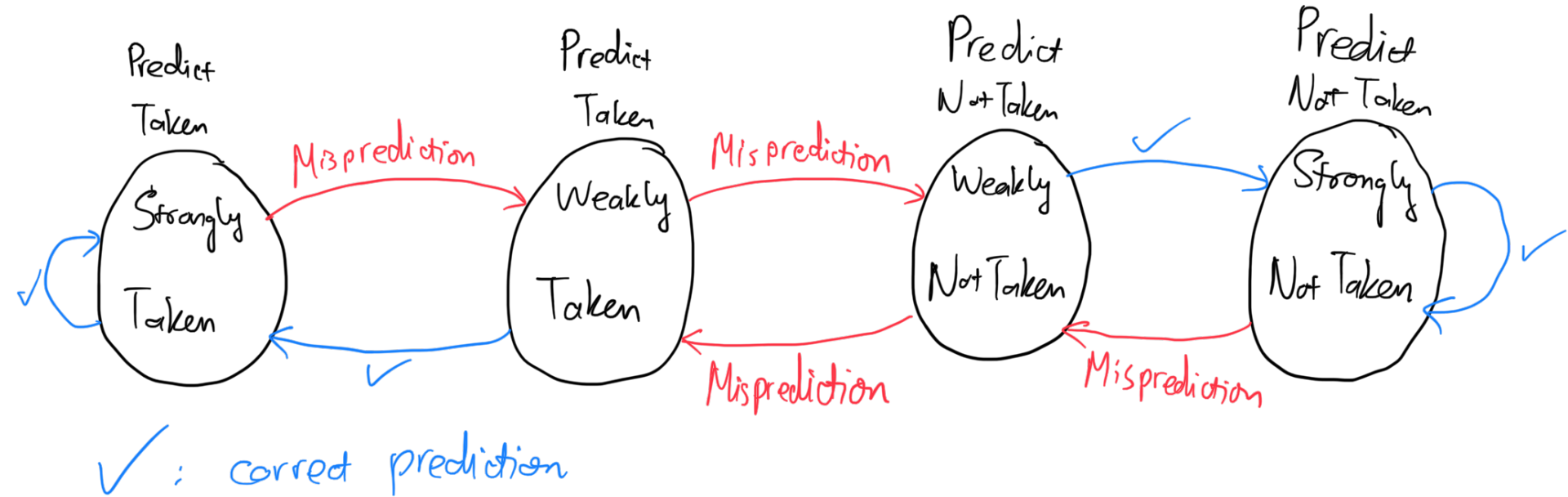
Cycle		1	2	3	4	5	6	7	8	9	10	11
PC=0x10 bne t1, t2, 0x14		IF	ID	EX	MEM	WB						
PC=0x14 add t5, t3, t4			IF	ID	EX	Flushed						
PC=0x18 xor t6, t3, t4				IF	ID	Flushed						
PC=0x1c sub t1, t2, t3					IF	Flushed						
PC=0x20 and t2, t3, t3												
PC=0x24 or t4, t1, t6						IF	ID	EX	MEM	WB		
PC=0x28 mul t1, t2, t3							IF	ID	EX	MEM	WB	

# Example: Pipeline-centric view of instructions

bp: predicted  
not taken  
outcome: taken

Cycle		IF	ID	EX	MEM	WB
PC=0x10 bne t1, t2, 0x14	Cycle=1	<u>bne</u>	----	----	----	----
PC=0x14 add t5, t3, t4	Cycle=2	<u>add</u>	<u>bne</u>	----	----	----
PC=0x18 xor t6, t3, t4	Cycle=3	xor	add	<u>bne</u>	----	----
PC=0x1c sub t1, t2, t3	Cycle=4	sub	xor	add	<u>bne</u>	----
PC=0x20 and t2, t3, t3	Cycle=5	or	<u>nop</u>	<u>nop</u>	<u>nop</u>	bne
PC=0x24 or t4, t1, t6						
PC=0x28 mul t1, t2, t3						

## 2-bit saturating counter



# Data Hazards

- Data dependencies *among registers*.
  - RAW: read-after-write (true dependency), mitigation: forwarding/stalling
  - WAR: write-after-read (anti dependency), mitigation: reg-renaming/stalling
  - WAW: write-after-write (~~anti~~ dependency), mitigation: reg-renaming/stalling
- renaming*

# Data Hazards

- In the 5-stage pipeline, we also have load-to-use hazard.
  - The value loaded from memory is not available to use until the write stage.
  - The next instruction reads the loaded value.
  - Mitigation: inserting a bubble after the load instruction.

Cycle		1	2	3	4	5	6	7
inst 1: ld t0, 0x400(t1)		IF	ID	EX	MEM	WB		
inst 2: add t2, t0, t3			IF	ID	ID	EX	MEM	WB

*Handwritten annotations:*

- A blue arrow points from the WB stage of inst 1 to the EX stage of inst 2, with the text "forwarding" written next to it.
- Below the pipeline, the stages IF, ID, and EX are handwritten in blue.

# Data Hazards: Dynamic Data Dependency

- Assume a pipeline that performs effective address and memory access all in Execute stage. Assume that the order of execution in the Execute stage is out-of-order.
- Each memory instruction requires calculating effective address before accessing memory.
- ~~We can issue a memory instruction to the Execute stage if we issue the instruction and following conditions do not occur in the Execute stage,~~
  - ~~There's a pair of a store instruction and another memory instruction (load/store) that we don't know whether their effective addresses align.~~
  - ~~Essentially, we are avoiding RAW, WAR, WAW hazards when accessing memory.~~

→ Cannot issue a store instruction with other memory instructions in the same cycle if the memory alignment is unknown. It's okay to have them in the Execute stage at the same time, however.



# Data Hazards: Dynamic Data Dependency

- Address alignment:
  - Part of load/store regions overlap.

Cannot be ~~executed~~ <sup>issued</sup>  
at the same  
time

load 8-byte value  
`ld x8, 0x400(t1)`  
store 8-byte value  
`sd x8, 0x400(t2)`

loading from address range  
 $R[t1] + 0x400$  to  $R[t1] + 0x407$

↑ can be overlapped  
storing to address range

$R[t2] + 0x400$  to  $R[t2] + 0x407$

can be ~~executed~~ <sup>issued</sup>  
at the same  
time

`ld x8, 0x400(t1)`  
`sd x8, 0x500(t1)`

loading from address range  
 $R[t1] + 0x400$  to  $R[t1] + 0x407$

↑ always no overlapping ranges

storing to address range

$R[t1] + 0x500$  to  $R[t1] + 0x507$

# Forwarding

- In 5-stage pipeline:
  - Forward from MEM to EX stage.
  - Forward from WB to ~~MEM stage~~ EX stage .
  - Note: There's **no** forwarding from WB stage to MEM stage.
  - Have to stall if there's no forwarding mechanism.

- There's no intra-stage forwarding unless specified otherwise.

Quizzes RISC-V instruction formats:  $rd$ : destination register  
 $rs1, rs2$ : source registers

Arithmetics: (R-types, I-types, LUI, AUIPC):  $imm$ : immediate

(R) add / or / ...  $rd, rs1, rs2$       lui / auipc  $rd, imm$

(I) addi / ori / ...  $rd, rs1, imm$

Jumps/Branches: (J-types, B-types) Branches: blt  $rs1, rs2, imm$

Jumps: jal  $rd, imm$

jalr  $rd, rs1, imm$

Memory (Loads / Stores) loads  $\rightarrow$  ld, lw, lh, lb | ld  $rd, imm(rs1)$   
stores  $\rightarrow$  sd, sw, sh, sb | sd  $rs2, imm(rs1)$

$\hookrightarrow$  Store insts do not write to reg

# Quizzes

