# Out of Order Execution

HW4 Autograder to come;
Check-In 6 on Friday

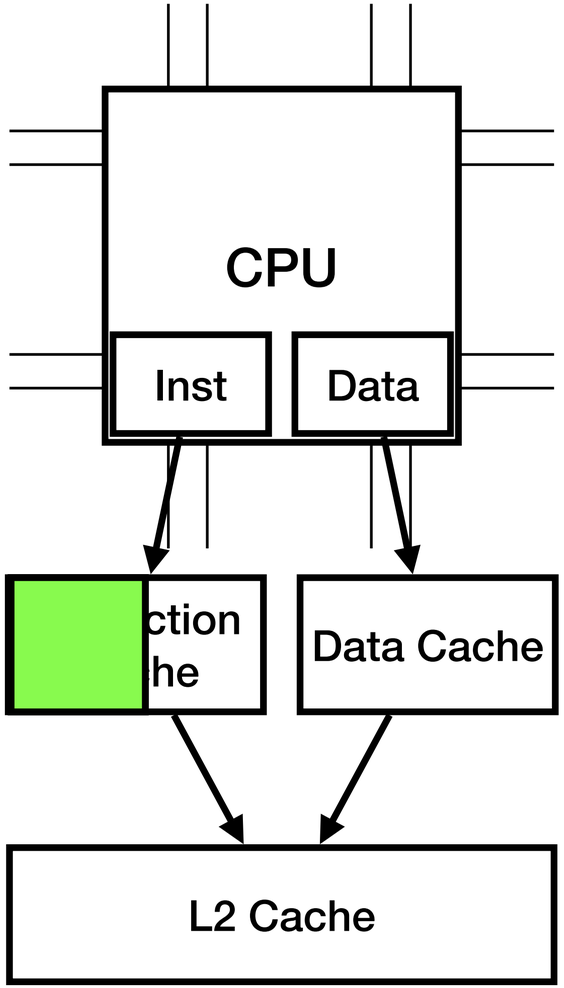
Tying up loose ends in our processor description!


THE CS DEPARTMENT PRESENTS
HOT ONES 6
HOT SAUCE FOR PROFS
ANSWERS FOR STUDENTS
BRING THE HEAT!
Thursday
April 16, 2026
4:15PM
Estella 1051
Argue-Auditorium
Come watch CS Professors eat the HOTTEST sauces
while answering your BURNING questions!
MILD
SPICY
HOT
EXTRA

# Outline

- Identifying the weirdness in our execution model
- Resolving this weirdness!

# Review: the L1/L2 Cache Hierarchy




Takeaway: by decoding all instructions in a cache block, we can eliminate additional instruction cache lookups. Instead, store in an instruction queue in the CPU!

How big are the cache blocks in these caches?

64 bytes!

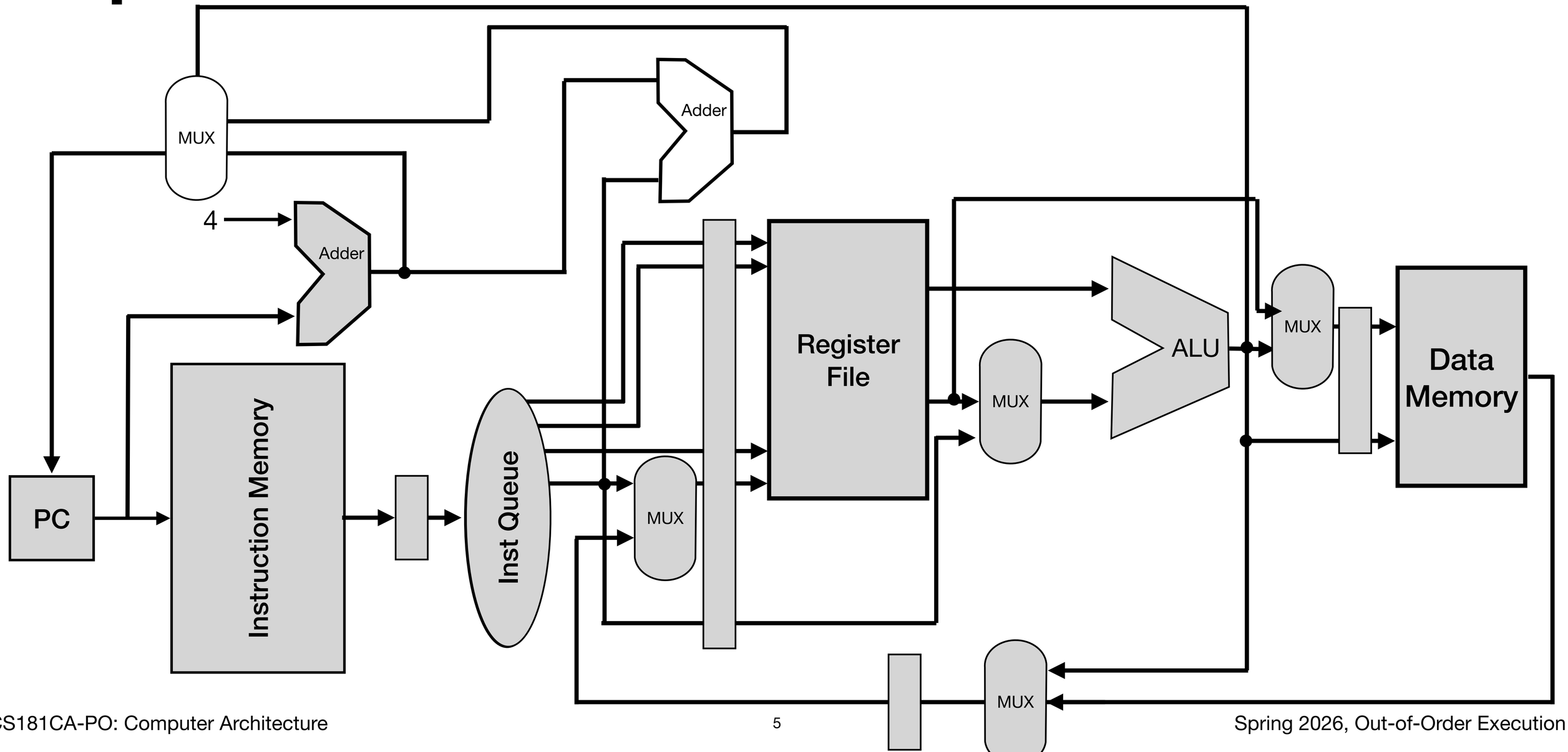
How big are our instructions in RISC-V (standard format)?

32 bits! 16 instructions per cache block

If we fetch one instruction at a time, we *should* benefit from spatial locality… right?

What would happen if we fetch the entire cache block?

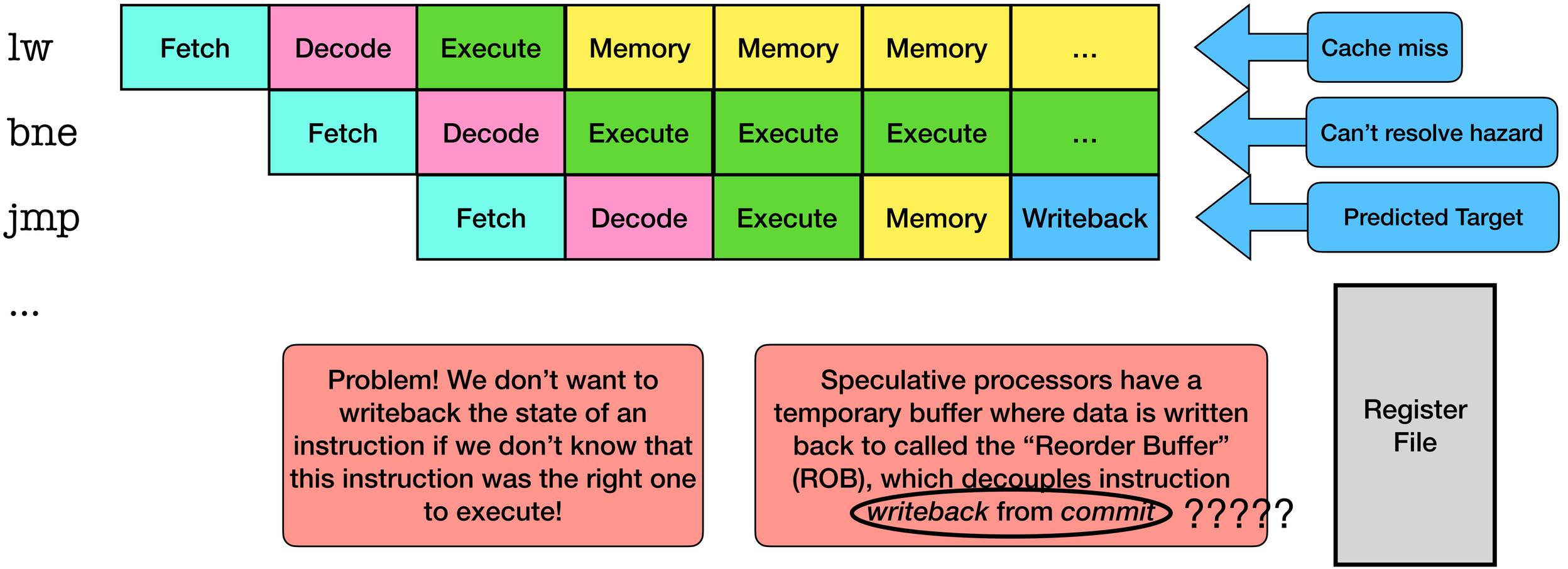
# Updated Data Path with Instruction Queue



# Instruction Queue

- By fetching all instructions in a cache block, we have a sequence of instructions that are ready to be *issued* into the processor data path
- Some of these instructions will be the incorrect instruction relative to the intended program order… detect with the hazard checking unit and resolve accordingly!
- This allows the processor to implement further optimizations, such as *multi-issue* processors if there are multiple functional units (e.g., 3 adders, 2 multipliers) instead of a single ALU for computation ➡️ send the instruction to the functional unit when it's ready!

# Review: Slow to Resolve Branches

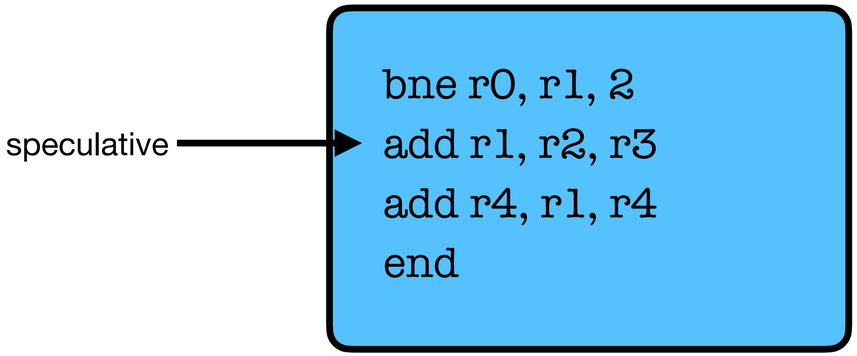



Problem! We don't want to writeback the state of an instruction if we don't know that this instruction was the right one to execute!

Speculative processors have a temporary buffer where data is written back to called the "Reorder Buffer" (ROB), which decouples instruction *writeback* from *commit* ?????

Register File

# Register Renaming




- The register file maintains a map of *logical registers* (e.g., what your program thinks a register is) to *physical registers* ➡️ more physical registers than logical registers!
- When fetching data from a register, the processor reads data from the physical register mapped to the logical register relative to the program state so far
- To commit an instruction, update the renaming map!

| PC | Logical Reg Addr | Physical Reg Addr | Physical Registers |
|---|---|---|---|
| 1 | r1 | 0 | r2 + r3 |
| 2 | r4 | 1 | R1[PC1] + r4 |
| | | | ... |
| | | | 0 |

Suppose this branch was incorrectly predicted not taken. How would you use the register renaming in the register file to “revert” our register state to before the branch? What if the branch was correctly predicted — how would the instruction be committed?

# Running Out of Physical Registers!

- Whenever we want to *write* a register value in a speculative state, we need to allocate a new physical register to that uncommitted instruction…
- Register renaming works well if we have enough physical registers to do the renaming of logical registers!
- If we do not have enough physical registers, then we have hit our maximum speculative depth! We should not “issue” an instruction unless there are enough physical registers available to map the registers for that instruction
- To commit the instruction, ensure that the only mapping of the logical register is the committed instruction’s physical register
- To discard (sometimes referred to as “squash”) an instruction, remove the mapping to its physical registers

# Register Renaming in Practice

- Seeing as the same *logical* register can be mapped to multiple *physical* registers across several speculative instructions, there is a new dependence ➡️ a *register name dependence*: which version of a logical register should be used?

- We want to avoid these as much as possible, so we would like to free these registers whenever we can!

- The majority of uncommitted registers are only used for a few cycles, but some have really long lifetimes… open problem!

**Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors**


Toru Koizumi
Nagoya Institute of Technology
Aichi, Japan
koizumi@nitech.ac.jp

Ryota Shioya
The University of Tokyo
Tokyo, Japan
shioya@ci.i.u-tokyo.ac.jp

Shu Sugita
The University of Tokyo
Tokyo, Japan
sugita@mtl.t.u-tokyo.ac.jp

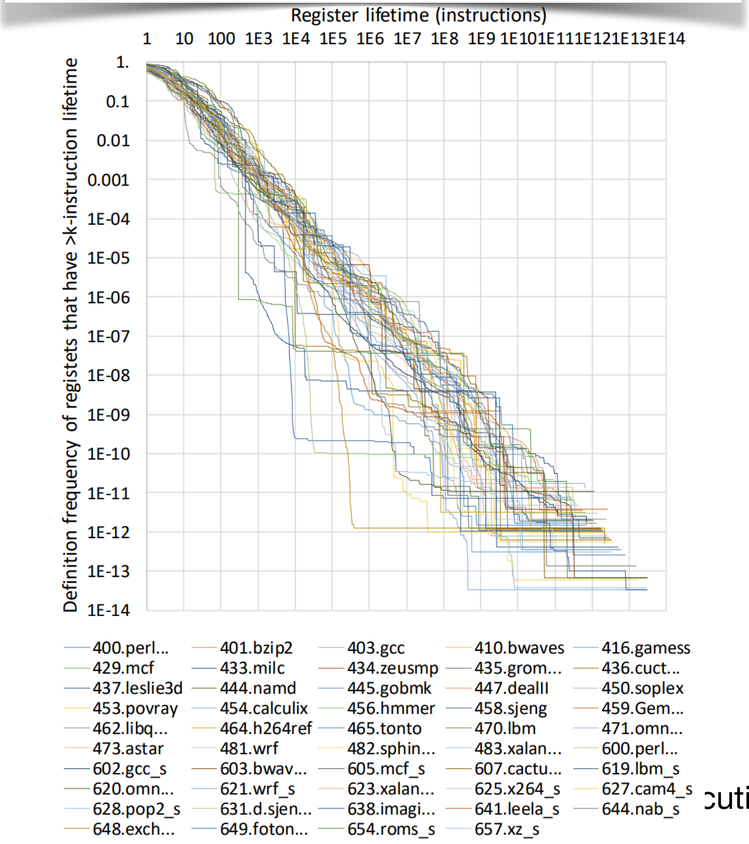
Taichi Amano
The University of Tokyo
Tokyo, Japan
amano@mtl.t.u-tokyo.ac.jp

Yuya Degawa
The University of Tokyo
Tokyo, Japan
degawa@mtl.t.u-tokyo.ac.jp

Junichiro Kadomoto
The University of Tokyo
Tokyo, Japan
kadomoto@mtl.t.u-tokyo.ac.jp

Hidetsugu Irie
The University of Tokyo
Tokyo, Japan
irie@mtl.t.u-tokyo.ac.jp

Shuichi Sakai
The University of Tokyo
Tokyo, Japan
sakai@mtl.t.u-tokyo.ac.jp

# Putting it all together — Out-of-Order (O3)

- If two instructions in the instruction queue are independent, then they can be executed by the processor in any order! This means that an add operation can be performed
- Instructions are fetched in-order (PC 0 is fetched before PC 1) and instructions are committed in-order to give the illusion of in-order execution
- The role of the reorder buffer (ROB) is to track the current sequence of executing instructions so that executed instructions are committed in order once all control hazards are resolved