COM SCI M151B Week 5

Aidan Jan

October 31, 2024

Out of Order Execution

Tradeoffs of Superscalar

Advantages:

- Higher IPC (instructions per cycle)

Disadvantages:

- Higher complexity for dependency checking
- More hardware resources needed

Pipeline is typically never **full** due to frequent dependency stalls!

Summary

- Scalar processors are limited (i.e., best case: IPC=1)
- ILP and superscalar could provide this opportunity to parallelize data processing and achieve IPC > 1.
- In-order fetching prevents us from achieving the full potential of superscalar *since usually there are not* ***enough*** *independent instructions in the pipeline.*
- A solution is out-of-order execution!

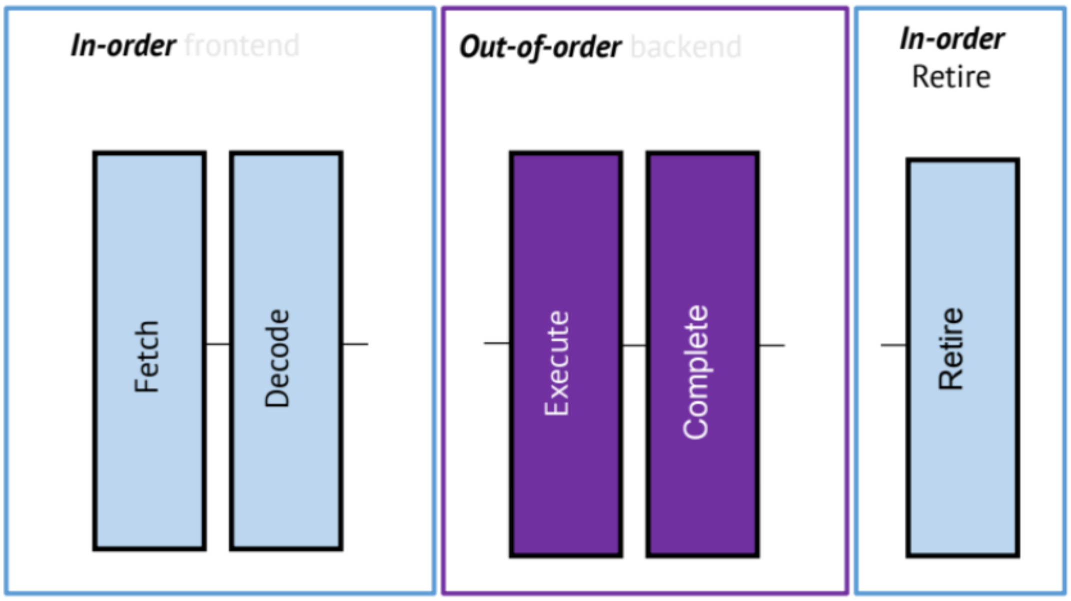
Status Quo

- Superscalar (N = 2 or 4) with pipelining and forwarding and branch prediction!
- Ideally, we can get IPC=N with small CT
- Minimal overhead due to RAW data hazard (due to load-use)
- Minimal stall due to >90% accuracy in branch prediction with only one cycle in miss penalty!
- Bottleneck
  - Not all the pipeline lanes can be always utilized due to *dependencies* and *hazards*!
  - Compilers can help but only in a limited form!
- Instead of always executing the next instruction, why don't we find the first available instruction and feed that into the pipeline?
  - If this instruction window is large enough, then we can always fully utilize our pipeline → IPC = N.

How to execute instructions out-of-order

- A mechanism for *tracking* instructions
  - Later instructions might be **ready**.
- A mechanism for *removing* data hazards
  - New hazards: WAW, WAR, load-store
- A mechanism for *recovery*
  - Speculation (e.g., branch misprediction), etc.

Out of Order Design



Out or Order - Pipeline Recovery

- We need a temporary phase for some instructions so that if they need to be flushed, they can be easily erased!
  - We define a new stage called "retire" (aka. "commit") and separate *completion* from *retire.*

"Retire"/"Commit"

What does it mean for an instruction to finish?

- Writing back data to the register file, or
- Writing data to the memory, or
- Updating the PC (branch)

If the data is already writtenm it is too late to flush, thus we should write somewhere temporarily until we are certain!

Complete vs. Retire

- Idea: to enable recovery, allow instructions to complete (i.e., prepare the final result) out of order, but retire them (i.e., actually writing data permanently) in order.
- We call writing data permanently, writing to architectural state (i.e., registers/memory)

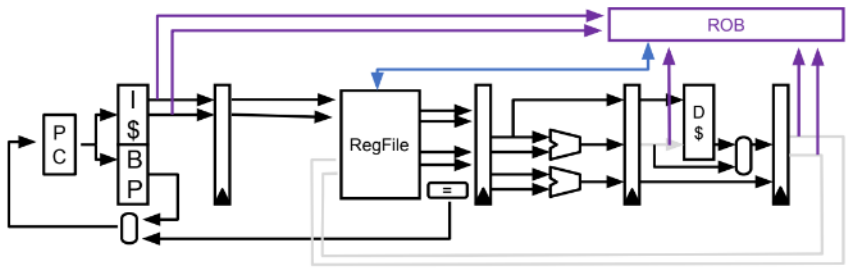
How to Implement retire?

- Need a buffer/storage to store temporary data!
- Need a mechanism to follow instructions in order and out of order to find out when to retire!

How to "retire" in-order?

- Re-order buffer (ROB)
  - ROB is a (circular) table/buffer that holds **completed** instructions.
  - ROB requires an instruction I_x only if *all* I_j (for $j < x$) are retired.

Datapath with ROB:



0.1 What if we start executing out of order?

Examples:

- RAW (Read After Write) = "true dependence"

```
add x2, x1, x3                sub x4, x3, x2
...                   -->     ...
sub x4, x3, x2                add x2, x1, x3
```

- WAW (Write After Write) = "output dependence"

```
add x2, x1, x3                sub x2, x3, x4
...                   -->     ...
sub x2, x3, x4                add x2, x1, x3
```

- WAR (Write After Read) = "anti-dependence"

```
add x3, x1, x2                sub x2, x3, x4
...                   -->     ...
sub x2, x3, x4                add x3, x1, x2
```

The WAW and WAR are called "False Dependencies". These can be fixed by simply changing the register, e.g., changing the `x2` in the sub instructions to `x5` instead.
The RAW case is the true dependence, and in that case, the solution is much more complex. We will discuss this later!

Register Renaming

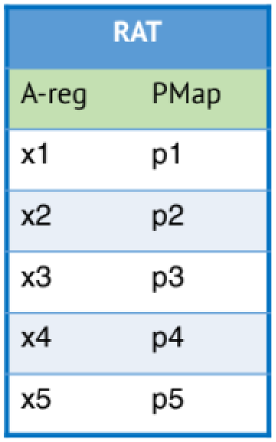
- The problem is that we have **limited architectural registers** (ISA registers, e.g., 32 in RISC-V)
- However, we can have much more **physical** registers (e.g., 128 registers).
- One architectural register (A-reg) can be assigned to **multiple physical registers** (P-reg).

How to do renaming?

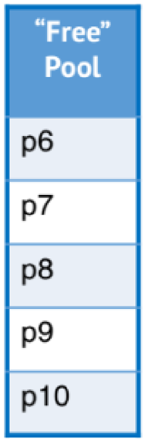
- Register Alias/Map Table (RAT)
  - One entry per architectural register.
  - Each entry stores the *physical location of the most recent* version of the architectural (logical) register.
  - **Algorithm:**
    * For each destination A-reg, the renaming algorithm assigns a new P-reg from a *pool of free (physical) registers.*
    * For each source A-reg, the renaming algorithm accesses RAT and finds the corresponding P-reg.

Renaming Example

- Let's assume that
  - We have 5 architectural registers and 10 physical registers.
  - The *initial* mapping looks like this:



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p3
x4	p4
x5	p5



"Free" Pool
p6
p7
p8
p9
p10

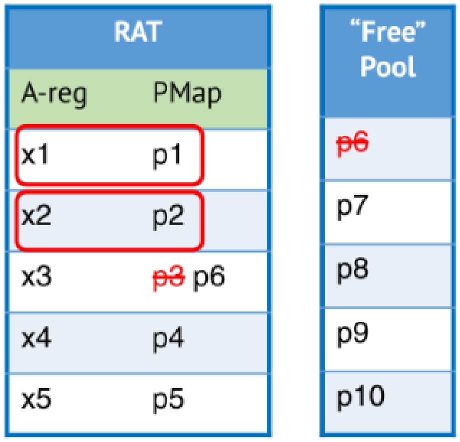
- Consider the instructions:

```
or x3, x2, x1
add x4, x3, x4
sub x3, x5, x2
addi x1, x3, 2
```

- Following the RAT, we get:

```
or x3, x2, x1       --> or p6, p2, p1
add x4, x3, x4
sub x3, x5, x2
addi x1, x3, 2
```

- We assign x3 to point to p6 instead of p3



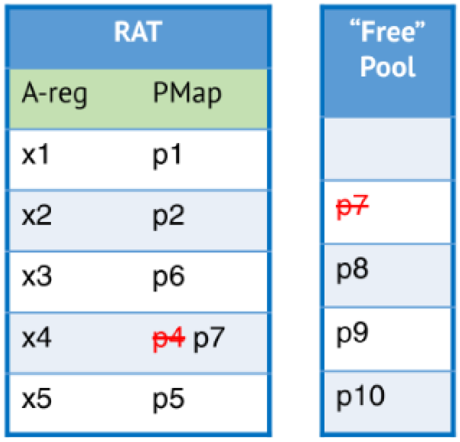
RAT	
A-reg	PMap
x1	p1
x2	p2
x3	~~p3~~ p6
x4	p4
x5	p5

"Free" Pool
~~p6~~
p7
p8
p9
p10

- Now, we fill the second line:

```
or x3, x2, x1        --> or p6, p2, p1
add x4, x3, x4       --> add p7, p6, p4
sub x3, x5, x2
addi x1, x3, 2
```

- Now, in this step, we assign x4 to p7.

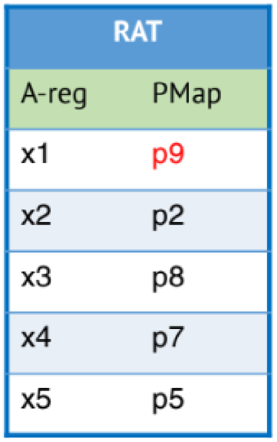


RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p6
x4	~~p4~~ p7
x5	p5

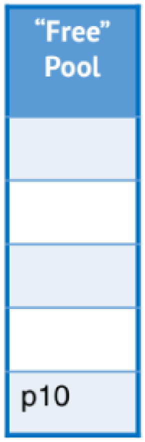
"Free" Pool
~~p7~~
p8
p9
p10

- Et cetera, we do reassignments for the destination registers.

```
or x3, x2, x1        --> or p6, p2, p1
add x4, x3, x4       --> add p7, p6, p4
sub x3, x5, x2       --> sub p8, p5, p2
addi x1, x3, 2       --> addi p9, p8, 2
```



RAT	
A-reg	PMap
x1	p9
x2	p2
x3	p8
x4	p7
x5	p5

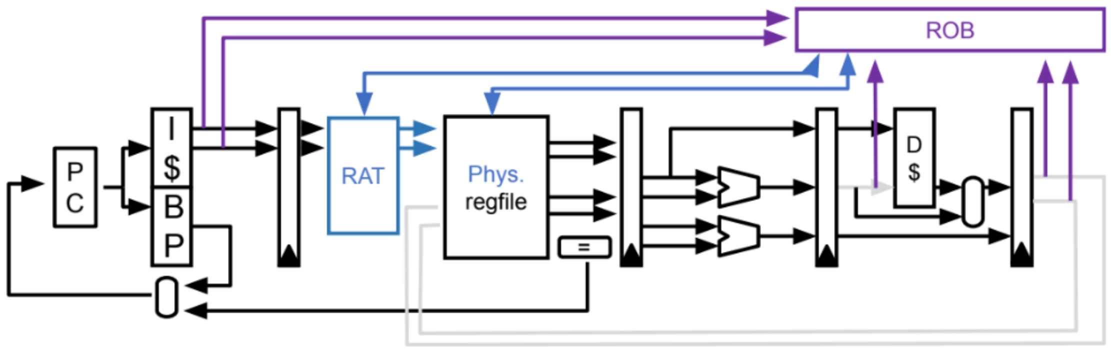


"Free" Pool
p10

When to free?

We free the previous, "old" destination register when the instruction is retired. e.g., once the first instruction is executed (and retired), we can free p3 (not p6!), because that means that the value of p3 is no longer used since it got reassigned.

Datapath with Renaming and ROB



Recap

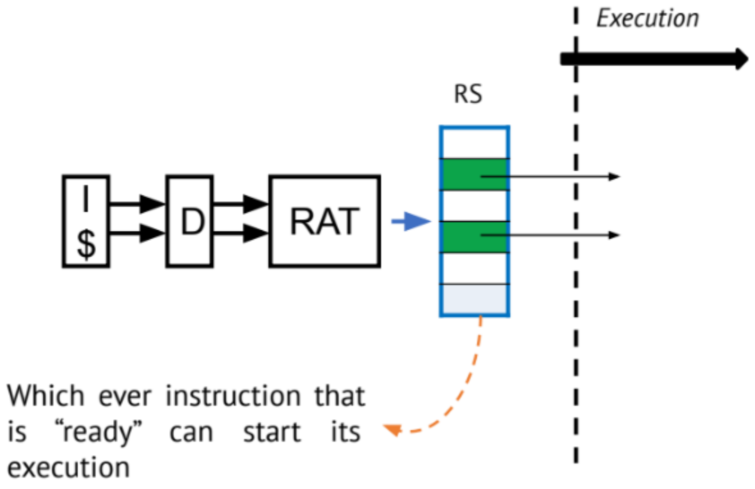
At this point, we have the mechanism for removing data hazards and the mechanism for recovery.

Now, we just need the mechanism for *tracking* instructions, as described earlier.

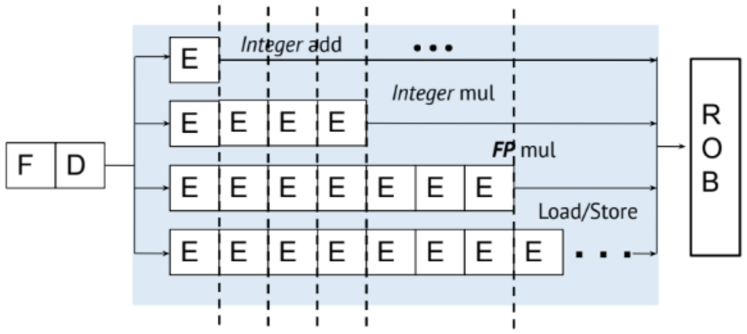
How to Schedule

- We need a mechanism to track who is ready:
  - `I3` is not ready, but `I4` is!
  - Who is ready?
    - This also helps to fix RAW hazards (i.e., forwarding!)

Reservation Station


Execution
RS
I $
D
RAT
Which ever instruction that is "ready" can start its execution

Now, we can remove the MEM stage and consolidate it to one **functional unit.**


F
D
E
Integer add
Integer mul
FP mul
Load/Store
ROB