last time (1)

multiple issue start multiple instructions at a time

out-of-order run instructions as operands ready

OOO pipeline:

in-order beginning: fetch / decode / rename → instruction queue
out-of-order: instruction queue → issue/reg read / execute / writeback
in-order end: commit (aka retire)

last time (2)

register renaming

- multiple versions of architectural register values
- keep each version in separate *physical* registrer
- new register for each change in value
- reuse registers when instructions committed

instruction issuing

- for each physical register: 'is it ready?'
- varied execution units (ALUs, load/store, etc.)
- issue instructions when inputs ready + execution unit ready

quiz Q1

longest stage work + register delay = 400ps

quiz Q2

```
movq 0x1234(%r8), %r9    F  D  E1  E2  M1  M2  W
addq %r9, %r10              F  D   D   D   D   E1  E2  M1  M2
                                                   1   2   3
```

quiz Q3

```
addq %r8, %r9        F D E1 E2 M1 M2 W
                            ^-- [R9 computed]
                            v-- [R9 read]
xorq %r9, %r10         F D  D  E1 E2 M1 M2 W
subq %r9, %r11           F  F  D  E1 E2 M1 M2 W
```

quiz Q4

"a forwarding" was supposed to be "and forwarding"

since 1 billion instructions can time to "ramp up" pipeline

with no stalling, 1 instruction finishes every cycle

1 cycle per instruction PLUS

- 10% take an extra cycle
- 5% take two extra cycles
- 2% take four extra cycles

$(1 + (10\% + 5\% + 2\%))$ times cycle time per instruction

quiz Q5

T = taken; N = not taken

inner jump: TNTNTN

- first time: predicted N (wrong)
- next time: same as previous (wrong)

outer jump: TTN

- first time: predicted N (wrong)
- next time: predicted T (correct!)
- next time: predicted T (wrong)

quiz Q6

```
addq %r9, %r10
movq 0(%r13), %r9
subq %r9, %r11
```

if old %r9 or %r10 slow to compute,
but %r13 and %r11 available quickly
then subq will be ready before addq

anonymous feedback (1)

Qs about OOO homework:

Part 2: data cache can begin new memory instruction when previous instr. at Stage 2?

> yes — pipelined data cache that can do1 access/cycle throughput

Part 3: we assume that the value of registers x01 through x15 are available?

> assume no shortage of available physical registers to cause stalls
> the questions for part 3 are showing you architectural (pre-renaming regs)

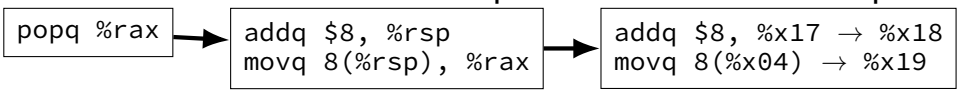
register renaming: missing pieces

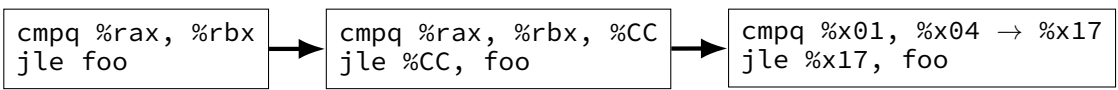
what about "hidden" inputs like %rsp, condition codes?

one solution: translate to intructions with additional register parameters

- making %rsp explicit parameter
- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones





OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

- e.g. Intel Skylake: up to approx. 16 cycles (v. 2 for simple pipelined CPU)

OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

- e.g. Intel Skylake: up to approx. 16 cycles (v. 2 for simple pipelined CPU)

some performance examples

```
example1:
    movq $10000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx 2.65 sec

```
example2:
    movq $10000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```

about 40B instructions

my desktop: approx 2.65 sec

some performance examples

```
example1:
    movq $10000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

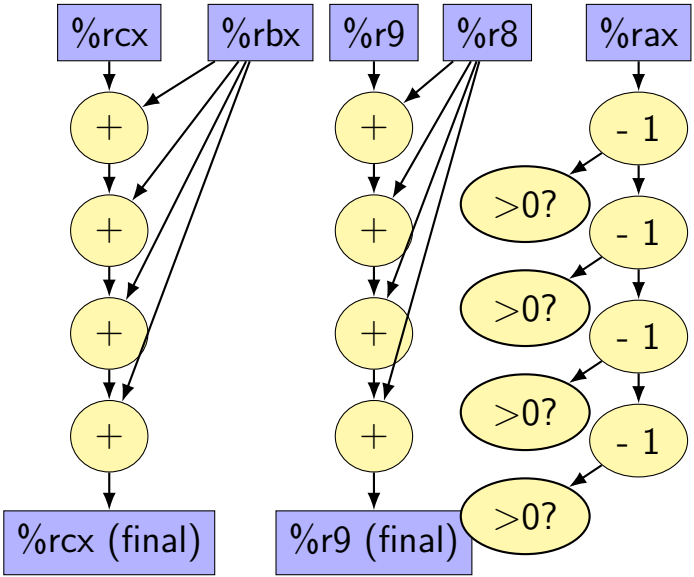
my desktop: approx 2.65 sec

```
example2:
    movq $10000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```

about 40B instructions

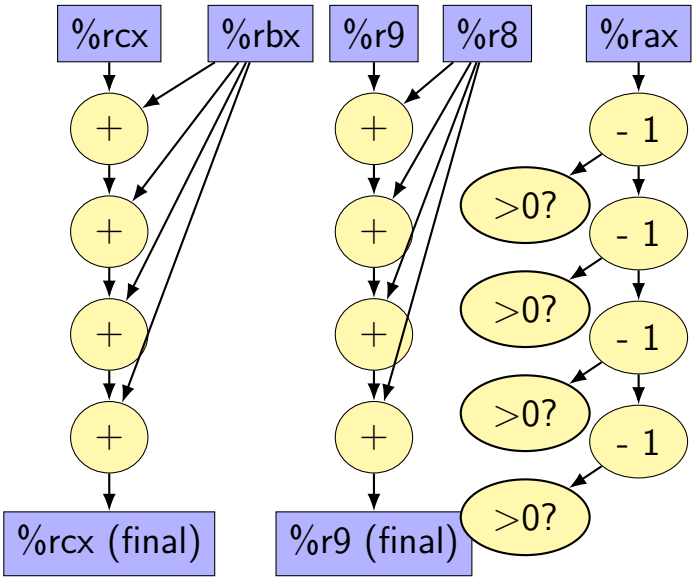
my desktop: approx 2.65 sec

data flow model and limits (1)




```
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
```

data flow model and limits (1)



each yellow box = instruction

arrows = dependences

instructions only executed when dependencies ready

reassociation

with pipelined, 5-cycle latency multiplier; how long does each take to compute?

$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

$(a \times b) \times (c \times d)$

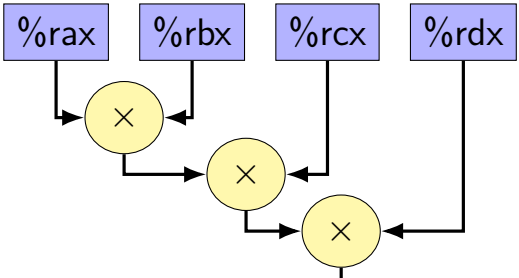
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

reassociation

with pipelined, 5-cycle latency multiplier; how long does each take to compute?

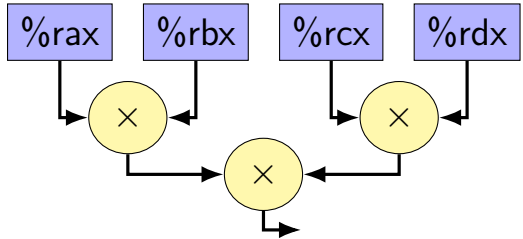
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

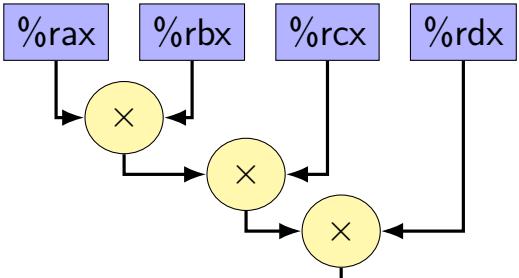


reassociation

with pipelined, 5-cycle latency multiplier; how long does each take to compute?

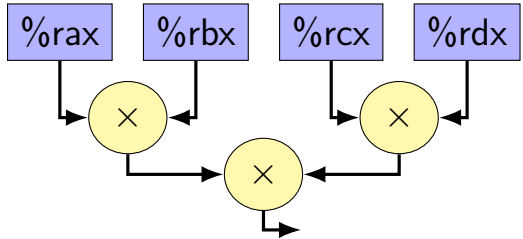
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



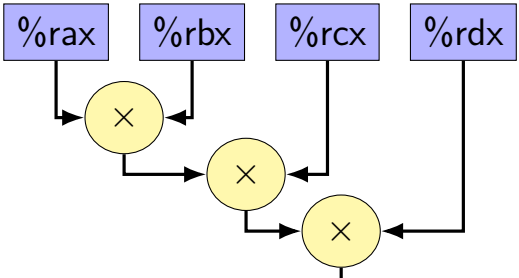
reassociation

with pipelined, 5-cycle latency multiplier; how long does each take to compute?

$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

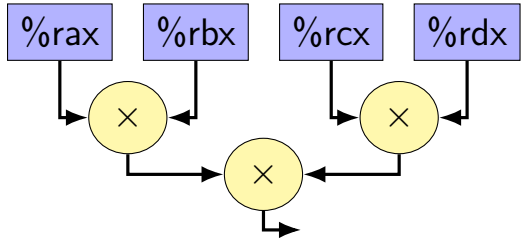
15 cycles



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

11 cycles

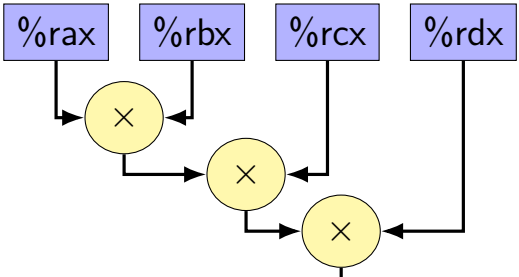


reassociation

with pipelined, 5-cycle latency multiplier; how long does each take to compute?

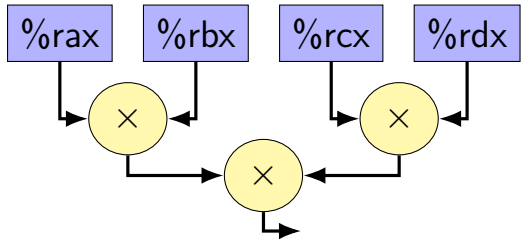
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



Intel Skylake OOO design

2015 Intel design — codename 'Skylake'

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

4 ALU functional units

- but some can handle more/different types of operations than others

2 load functional units

- but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer

check_passphrase

```
int check_passphrase(const char *versus) {
    int i = 0;
    while (passphrase[i] == versus[i] &&
           passphrase[i]) {
        i += 1;
    }
    return (passphrase[i] == versus[i]);
}
```

number of iterations = number matching characters

leaks information about passphrase, oops!

exploiting check_passphrase (1)

guess	measured time
aaaa	100 ± 5
baaa	103 ± 4
caaa	102 ± 6
daaa	111 ± 5
eaaa	99 ± 6
faaa	101 ± 7
gaaa	104 ± 4
...	...

exploiting check_passphrase (2)

guess	measured time
daaa	102 ± 5
dbaa	99 ± 4
dcaa	104 ± 4
ddaa	100 ± 6
deaa	102 ± 4
dfaa	109 ± 7
dgaa	103 ± 4
...	...

timing and cryptography

lots of asymmetric cryptography uses big-integer math

example: multiplying 500+ bit numbers together

how do you implement that?

big integer multiplcation

say we have two 64-bit integers x, y

and want to 128-bit product, but our multiply instruction only does 64-bit products

one way to multiply:

divide x, y into 32-bit parts: $x = x_1 \cdot 2^{32} + x_0$ and $y = y_1 \cdot 2^{32} + y_0$

then $xy = x_1 y_1 2^{64} + x_1 y_0 \cdot 2^{32} + x_0 y_1 \cdot 2^{32} + x_0 y_0$

big integer multiplcation

say we have two 64-bit integers x, y

and want to 128-bit product, but our multiply instruction only does 64-bit products

one way to multiply:

divide x, y into 32-bit parts: $x = x_1 \cdot 2^{32} + x_0$ and $y = y_1 \cdot 2^{32} + y_0$

then $xy = x_1 y_1 2^{64} + x_1 y_0 \cdot 2^{32} + x_0 y_1 \cdot 2^{32} + x_0 y_0$

can extend this idea to arbitrarily large numbers

number of smaller multiplies depends on size of numbers!

big integers and cryptography

naive multiplication idea:

- number of steps depends on size of numbers

problem: sometimes the value of the number is a secret

- e.g. part of the private key

oops! revealed through timing

big integer timing attacks in practice (1)

early versions of OpenSSL (TLS implementation)had timing attack

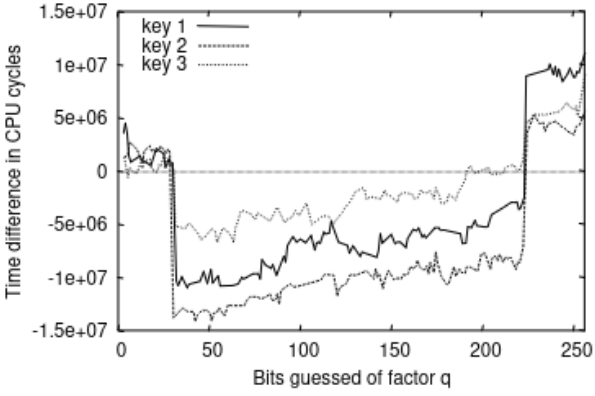
- Brumley and Boneh, "Remote Timing Attacks are Practical" (Usenix Security '03)

attacker could figure out bits of private key from timing

why? variable-time mulitplication and modulus operations

- got faster/slower depending on how input was related to private key

big integer timing attacks in practice (2)



(a) The zero-one gap $T_g - T_{g_{hi}}$ indicates that we can distinguish between bits that are 0 and 1 of the RSA factor q for 3 different randomly-generated keys. For clarity, bits of q that are 1 are omitted, as the x-axis can be used for reference for this case.

Figure 3a from Brumley and Boneh, "Remote Timing Attacks are Practical"

browsers and website leakage

web browsers run code from untrusted webpages

one goal: can't tell what other webpages you visit

some webpage leakage (1)

...as you can see here, here, and here ...

convenient feature 1: browser marks visited links

```
<script>
var the_color = window.getComputedStyle(
    document.querySelector('a[href=~"foo.com"]')
).color
if (color == ...) { ... }
</script>
```

convenient feature 2: scripts can query current color of something

some webpage leakage (1)

...as you can see here, here, and here ...

convenient feature 1: browser marks visited links

```
<script>
var the_color = window.getComputedStyle(
    document.querySelector('a[href=~"foo.com"]')
).color
if (color == ...) { ... }
</script>
```

~~convenient feature 2: scripts can query current color of something~~

fix 1: getComputedStyle lies about the color

fix 2: limited styling options for visited links

some webpage leakage (2)

one idea: script in webpage times loop that writes big array

variation in timing depends on other things running on machine

some webpage leakage (2)

one idea: script in webpage times loop that writes big array

variation in timing depends on other things running on machine

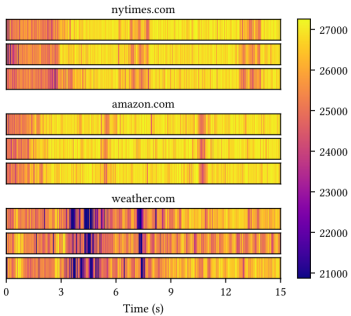


Figure 3: Example loop-counting traces collected over 15 seconds. Darker shades indicate smaller counter values and lower instruction throughput.

turns out, other webpages create distinct "signatures"

Figure from Cook et al, "There's Always a Bigger Fish: Clarifying Analysis of a Machine-Learning-Assisted Side-Channel Attack" (ISCA '22)

inferring cache accesses (1)

suppose I time accesses to array of chars:

reading array[0]: 3 cycles
reading array[64]: 4 cycles
reading array[128]: 4 cycles
reading array[192]: 20 cycles
reading array[256]: 4 cycles
reading array[288]: 4 cycles
...

what could cause this difference?

array[192] not in some cache, but others were

inferring cache accesses (2)

some psuedocode:

```
char array[CACHE_SIZE];
AccessAllOf(array);
*other_address += 1;
TimeAccessingArray();
```

suppose during these accesses I discover that `array[128]` is slower to access

probably because `*other_address` loaded into cache + evicted it

what do we know about `other_address`? (select all that apply)

A. same cache tag B. same cache index C. same cache offset
D. diff. cache tag E. diff. cache index F. diff. cache offset

some complications

caches often use physical, not virtual addresses

- (and need to know about physical address to compare index bits)
- (but can infer physical addresses with measurements/asking OS)
- (and often OS allocates contiguous physical addresses esp. w/'large pages')

storing/processing timings evicts things in the cache

- (but can compare timing with/without access of interest to check for this)

processor "pre-fetching" may load things into cache before access is timed

- (but can arrange accesses to avoid triggering prefetcher
- and make sure to measure with memory barriers)

some L3 caches use a simple hash function to select index instead

exercise: inferring cache accesses (1)

```
char *array;
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
if (mystery) {
    *pointer += 1;
}
if (TimeAccessTo(&array[index]) > THRESHOLD) {
    /* pointer accessed */
}
```

suppose `pointer` is `0x1000188`

and cache (of interest) is direct-mapped, 32768 (2^{15}) byte, 64-byte blocks

what array index should we check?

solution

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
if (mystery) { *pointer = 1; }
if (TimeAccessTo(&array[index]) > THRESHOLD) { /* pointer accessed */ }
```

2^{15} byte direct mapped cache, $64 = 2^6$ byte blocks

9 index bits, 6 offset bits

`0x1000188`: …`0000 0001 1000 1000`

`array[0]` starts at multiple of cache size — index 0, offset 0

to get index 6, offset 0 array[`0b1 1000 0000`] = array[`0x180`]

solution

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
if (mystery) { *pointer = 1; }
if (TimeAccessTo(&array[index]) > THRESHOLD) { /* pointer accessed */ }
```

2^{15} byte direct mapped cache, $64 = 2^6$ byte blocks

9 index bits, 6 offset bits

`0x1000188`: …`0000 0001 1000 1000`

`array[0]` starts at multiple of cache size — index 0, offset 0

to get index 6, offset 0 array[`0b1 1000 0000`] = array[`0x180`]

aside

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
if (mystery) { *pointer += 1; }
if (TimeAccessTo(&array[index]) > THRESHOLD) {
    /* pointer accessed */
}
```

will this detect when pointer accessed? yes

will this detect if mystery is true? not quite

...because branch prediction could started cache access

exercise: inferring cache accesses (2)

```
char *other_array = ...;
char *array;
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
other_array[mystery] += 1;
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (TimeAccessTo(&array[i]) > THRESHOLD) {
        /* found something interesting */
    }
}
```

other_array at 0x200400, and interesting index is i=0x800, then what was mystery?

solution

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
other_array[mystery] += 1;
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (TimeAccessTo(&array[i]) > THRESHOLD) { ... }
}
```

at i=0x800: …0*000 1000 00*00 0000 (cache index = 0x20)

other_array at 0x200400

Q: 0x200400 + X has cache index 0x20?

0x200400	…0	000	0100	00	00	0000
+ X	…?	000	0100	00	??	????
0x200400+X	…?	000	1000	00	??	????

exercise: inferring cache accesses (2)

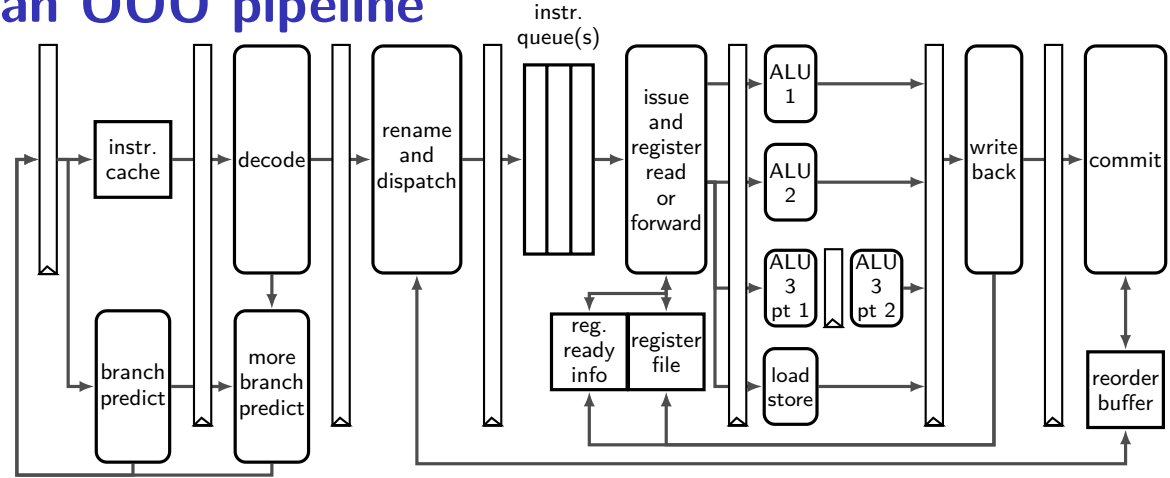
```
char *array;
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
if (mystery) {
    *pointer = 1;
}
if (TimeAccessTo(&array[index1]) > THRESHOLD ||
    TimeAccessTo(&array[index2]) > THRESHOLD) {
    /* pointer accessed */
}
```

pointer is 0x1000188

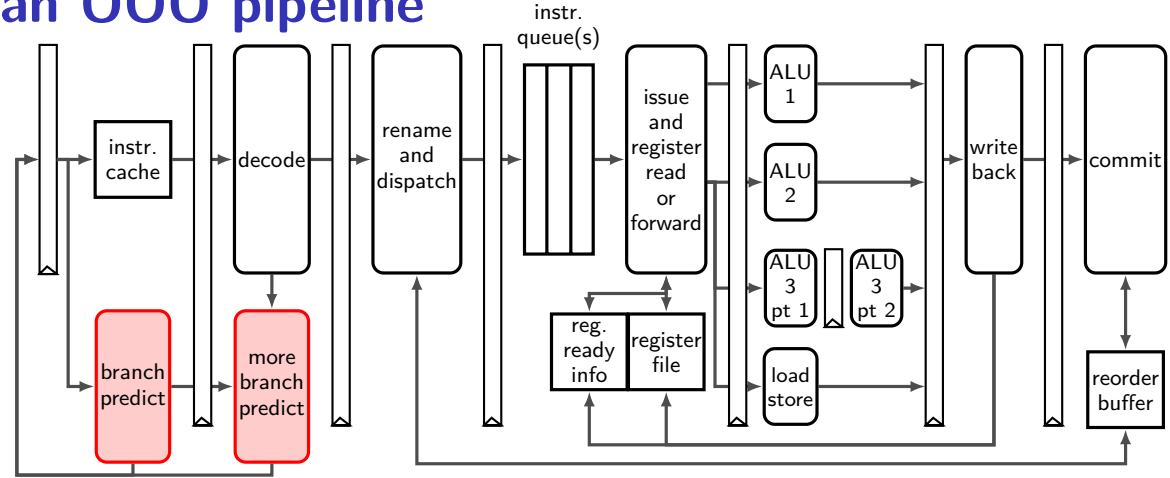
cache is 2-way, 32768 (2^{15}) byte, 64-byte blocks, ???? replacement

what array indexes should we check?

an OOO pipeline

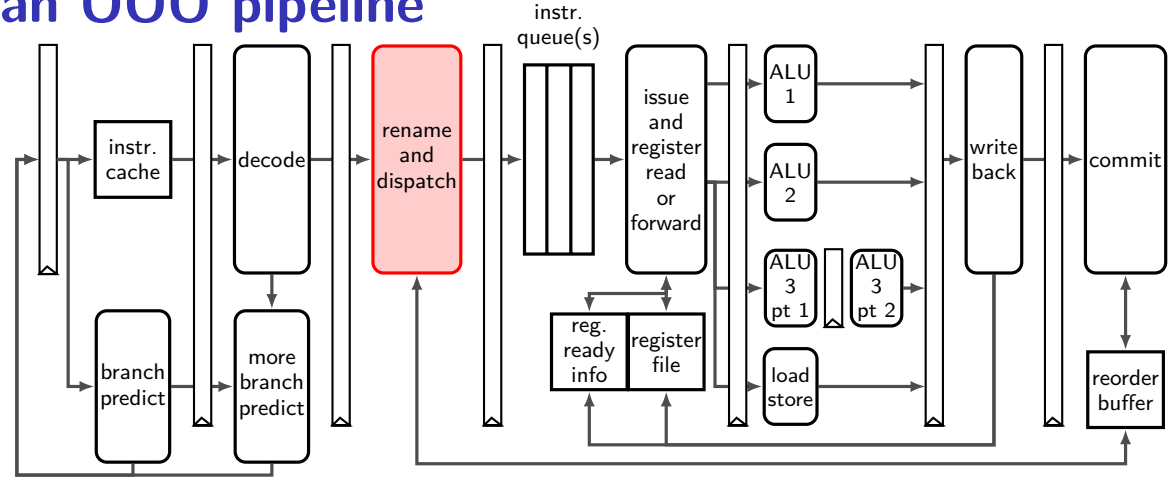


an OOO pipeline



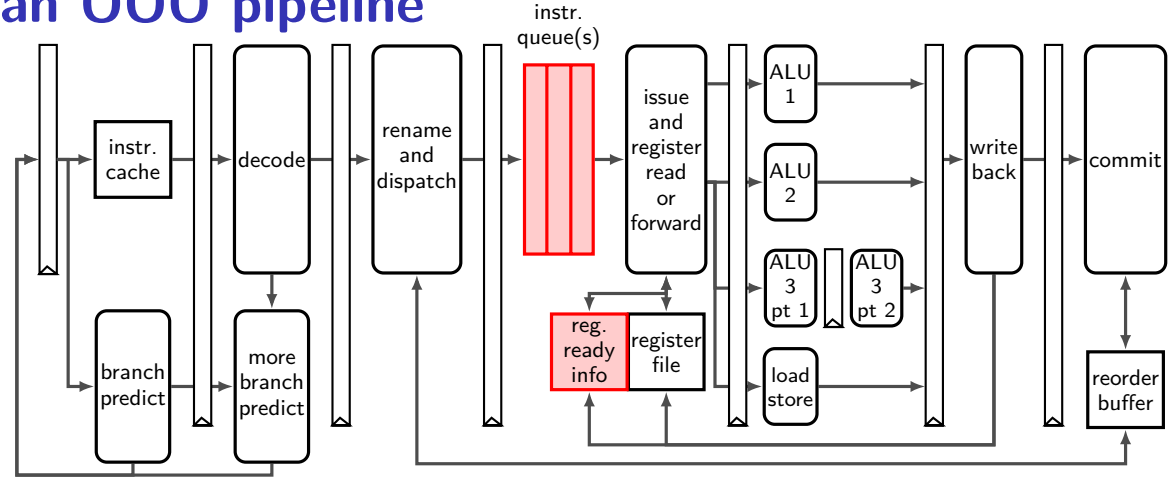
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



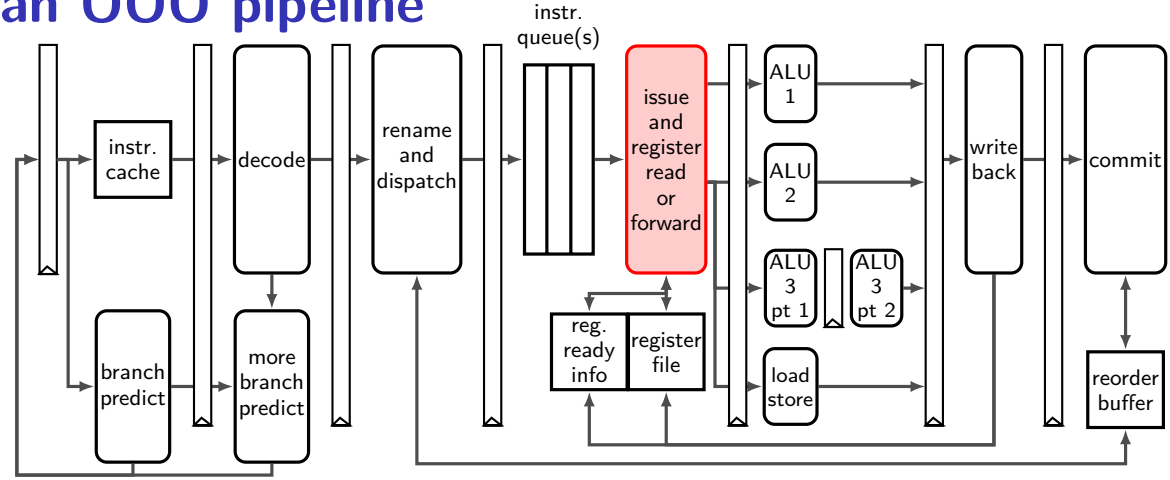
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



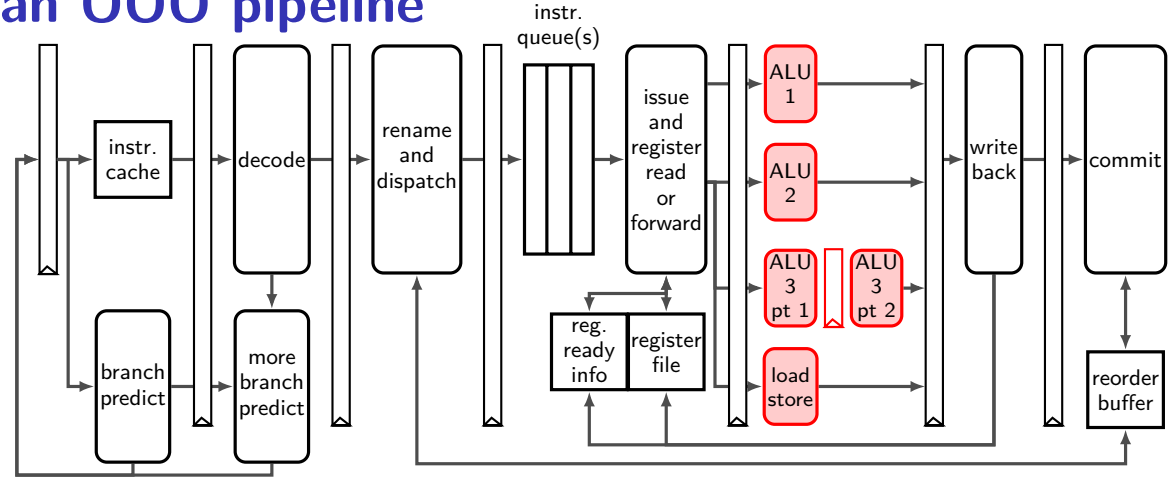
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



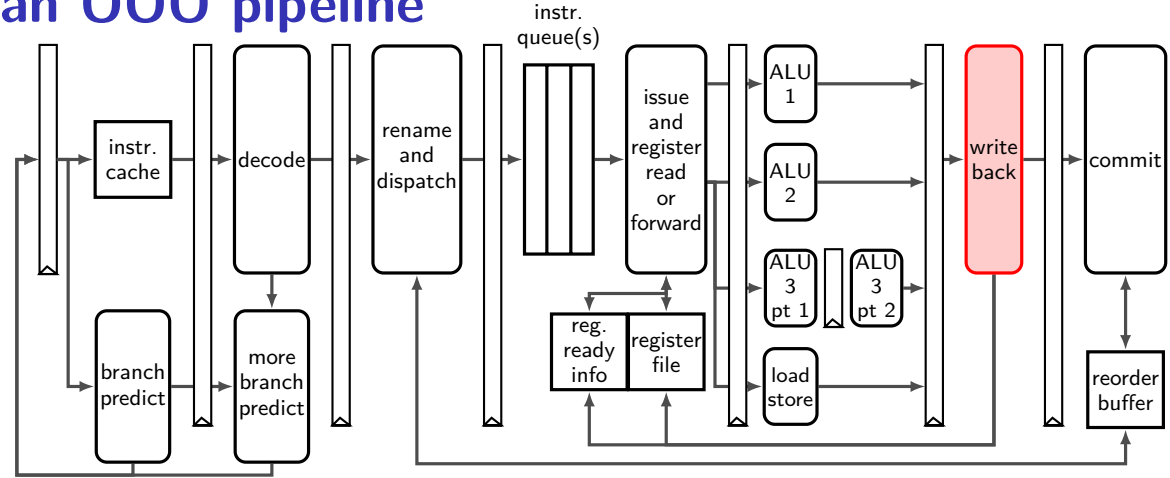
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

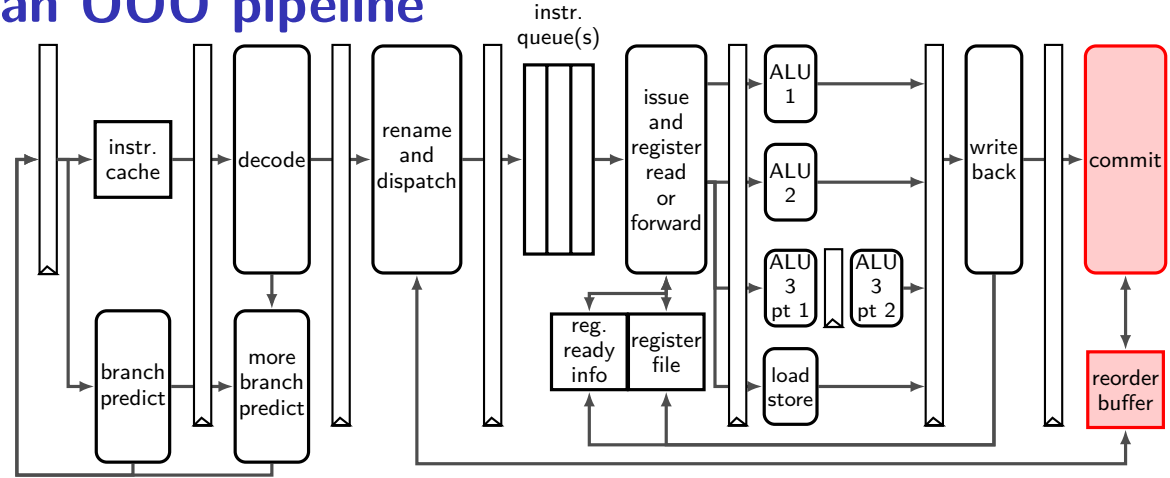
an OOO pipeline



writeback results to physical registers

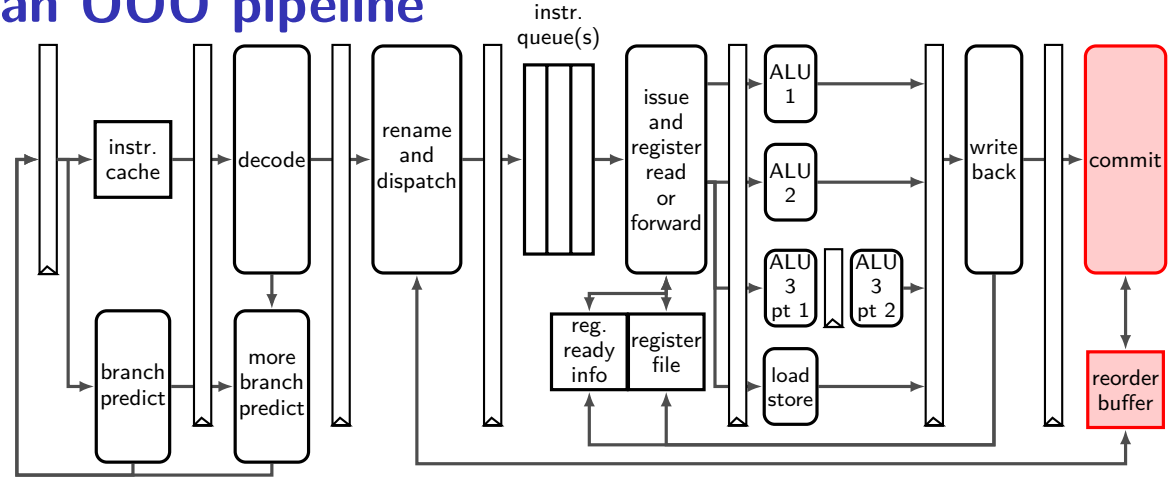
register file: typically support writing 3+ registers at a time

an OOO pipeline



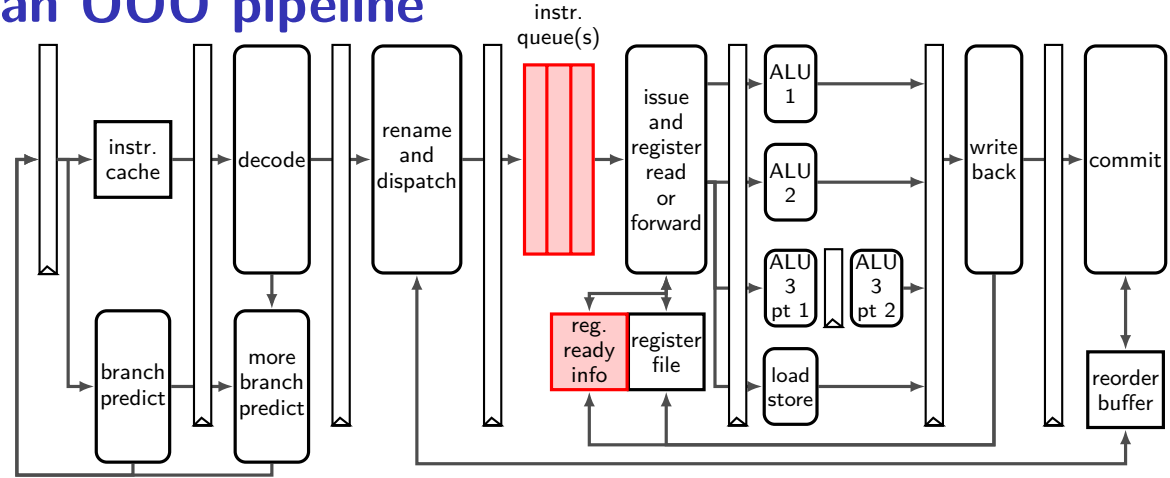
new commit (sometimes *retire*) stage finalizes instruction
figures out when physical registers can be reused again

an OOO pipeline

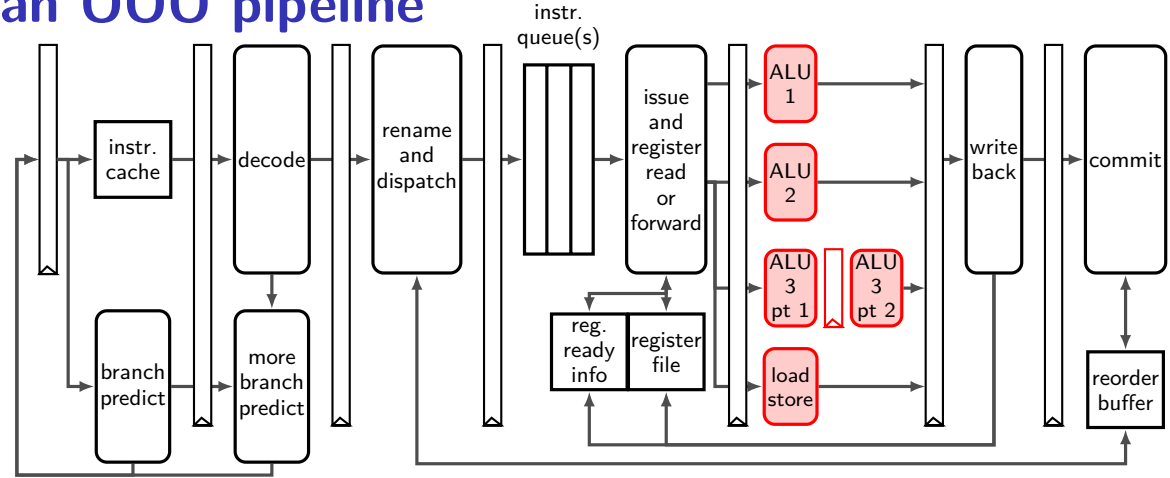


commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.

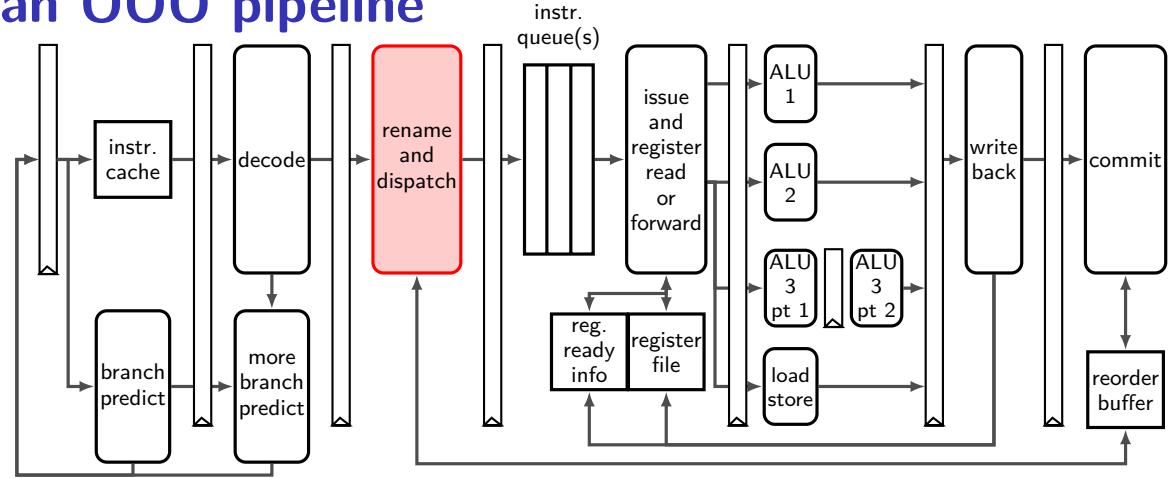
an OOO pipeline



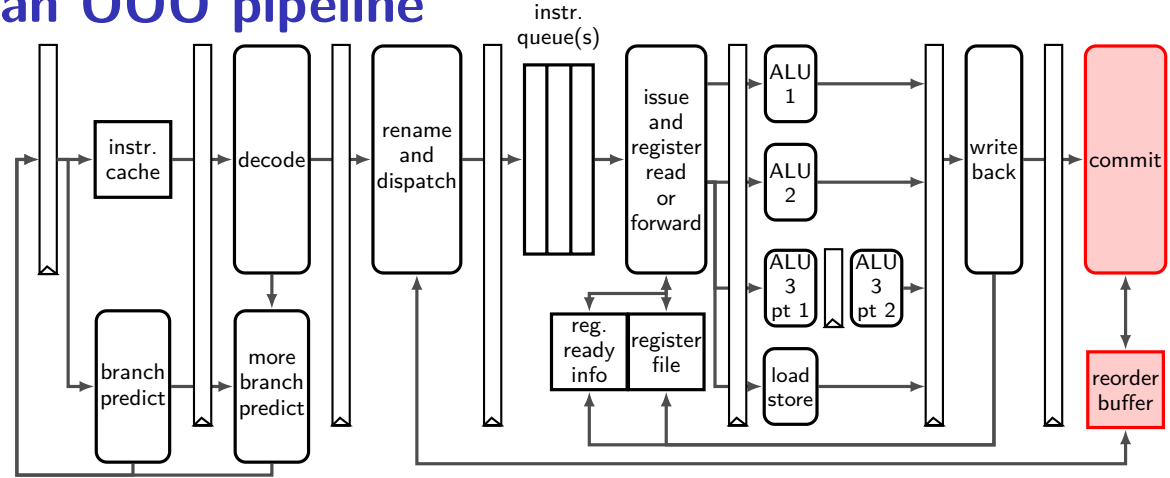
an OOO pipeline



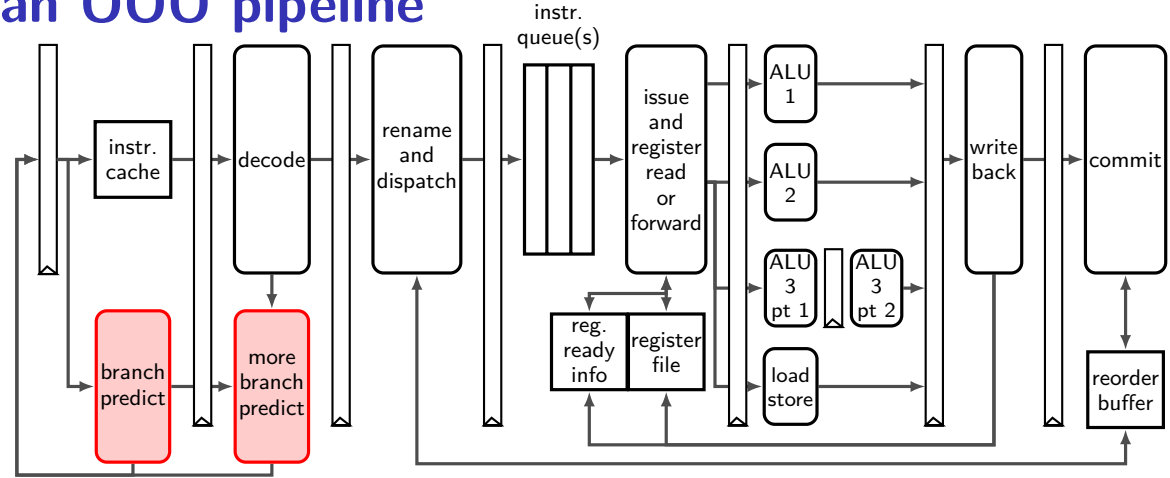
an OOO pipeline



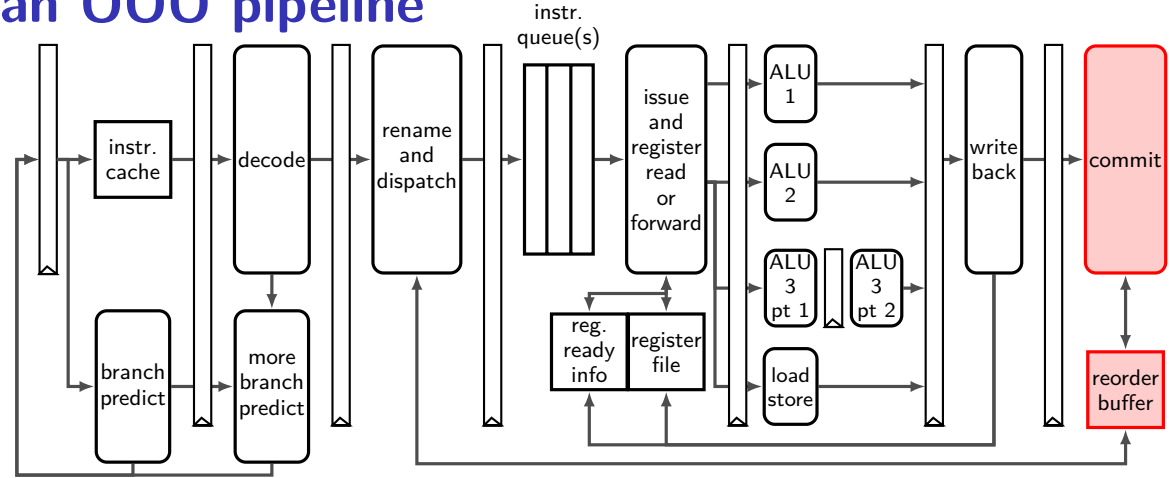
an OOO pipeline



an OOO pipeline



an OOO pipeline

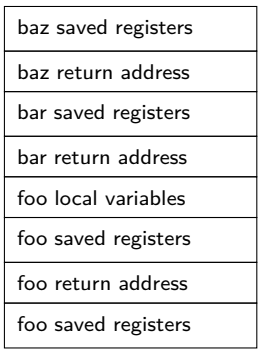


predicting ret: ministack of return addresses

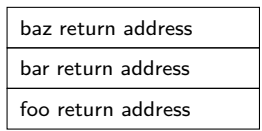
predicting ret — ministack in processor registers

push on ministack on call; pop on ret

ministack overflows? discard oldest, mispredict it later

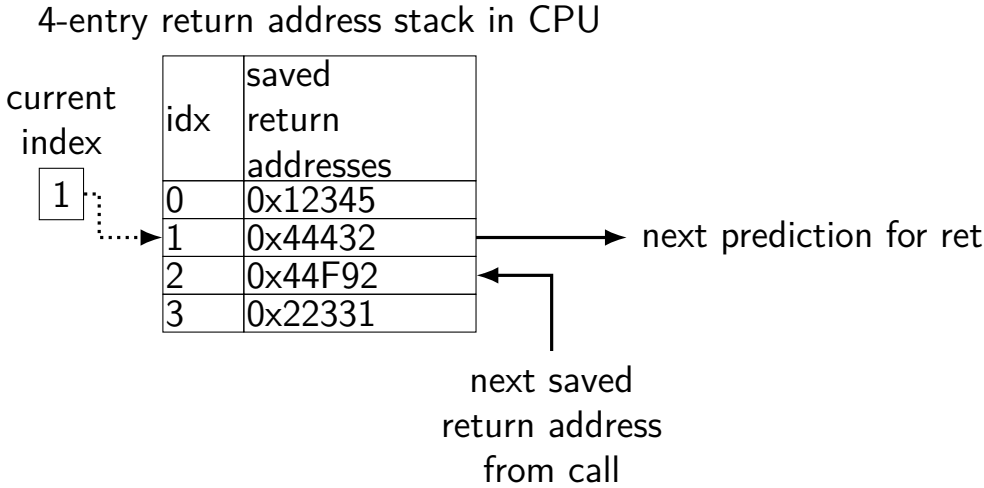


stack in memory



(partial?) stack in CPU registers

4-entry return address stack



on `call`: increment index, save return address in that slot