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

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 do 1 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)
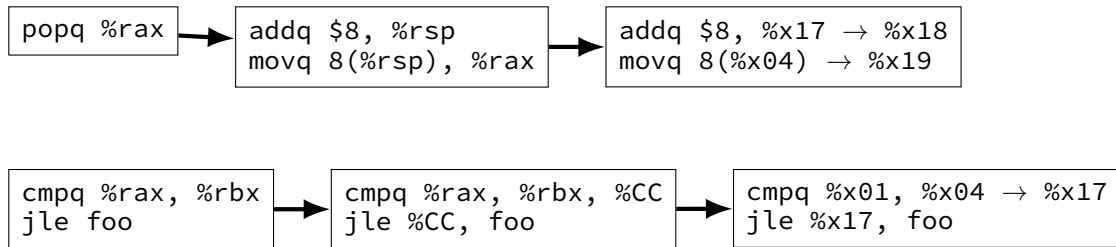
register renaming: missing pieces

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

```
popq %rax
```
→
```
addq $8, %rsp
movq 8(%rsp), %rax
```
→
```
addq $8, %x17 → %x18
movq 8(%x04) → %x19
```

```
cmpq %rax, %rbx
jle foo
```
→
```
cmpq %rax, %rbx, %CC
jle %CC, foo
```
→
```
cmpq %x01, %x04 → %x17
jle %x17, foo
```

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
```
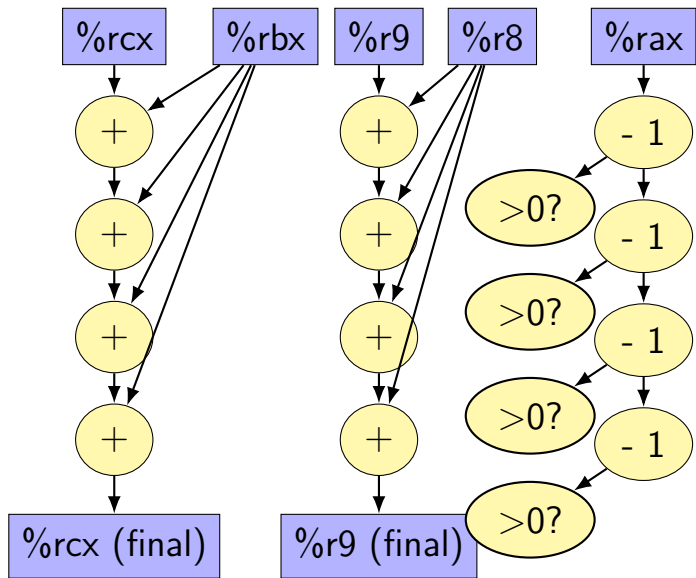
about 40B instructions
my desktop: approx 2.65 sec

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
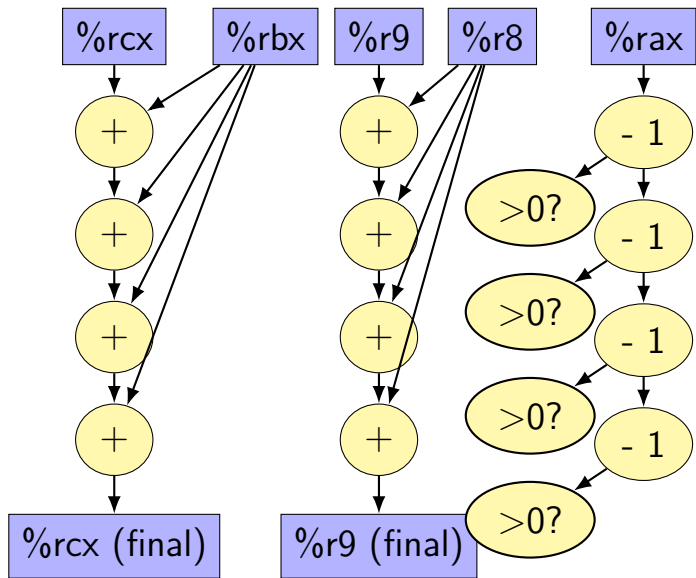my desktop: approx 2.65 sec

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
```

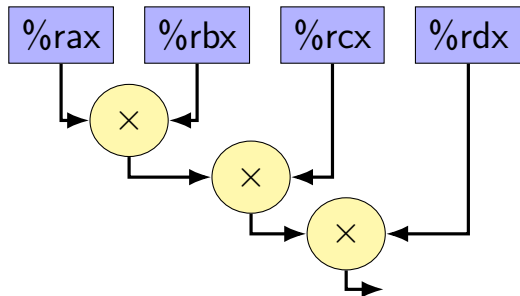$(a \times b) \times (c \times d)$

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

reassociation

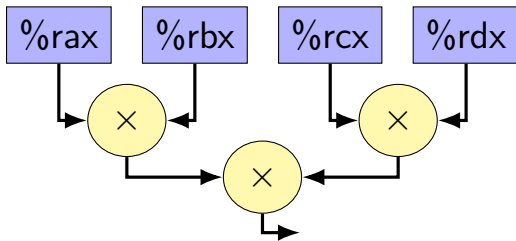with pipelined, 5-cycle latency multiplier; how long does each take to compute?

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

```
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
```

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
  determines how far ahead branch mispredictions, etc. can happen

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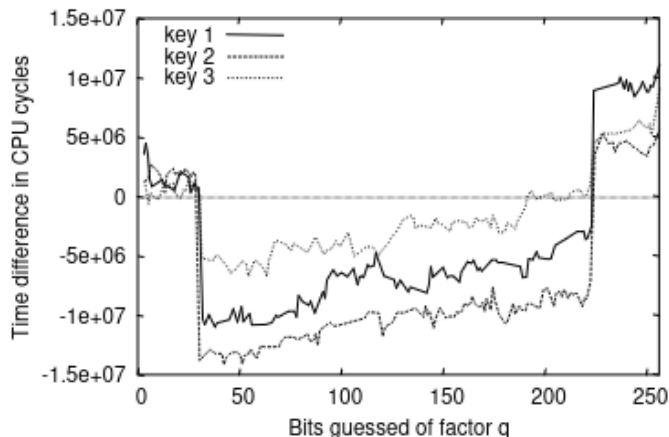
(a) The zero-one gap $T_g - T_{g_{h,i}}$ indicates that we can distinguish between bits that are 0 and 1 of the RSA factor q for 3 different randomly-generated keys. For clarity, bits of q that are 1 are omitted, as the x-axis can be used for reference for this case.

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
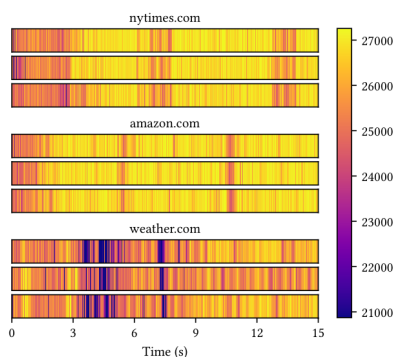
some webpage leakage (2)

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 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 of index bits

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?

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?

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?

PRIME+PROBE

name in literature: PRIME + PROBE

PRIME: fill cache (or part of it) with values

do thing that uses cache

PROBE: access those values again and see if it's slow

(one of several ways to measure how cache is used)

coined in attacks on AES encryption

example: AES (1)

from Osvik, Shamir, and Tromer, "Cache Attacks and Countermeasures: the Case of AES" (2004)

early AES implementation used lookup tables

goal: detect index into lookup table
    index depended on key + data being encrypted

tricks they did to make this work
    vary data being encrypted
    subtract average time to look for what changes
    lots of measurements

example: AES (2)

from Osvik, Shamir, and Tromer, "Cache Attacks and Countermeasures: the Case of AES" (2004)
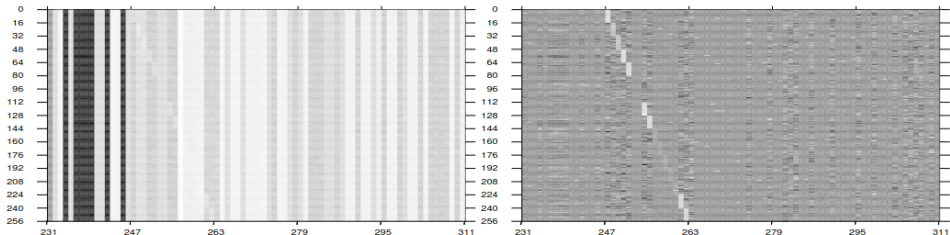
Fig. 5. Prime+Probe attack using 30,000 encryption calls on a 2GHz Athlon 64, attacking Linux 2.6.11 dm-crypt. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0. Left: raw timings (lighter is slower). Right: after subtraction of the average timing of the cache set. The bright diagonal reveals the high nibble of $p_0 = $ 0x00.

reading a value

```
char *array;
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
other_array[mystery * BLOCK_SIZE] += 1;
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) {
        ...
    }
}
```

with 32KB direct-mapped cache

suppose we find out that `array[0x400]` is slow to access

and other_array starts at address 0x100000

what was mystery?

revisiting an earlier example (1)

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

what if mystery is false *but* branch mispredicted?

revisiting an earlier example (2)

```
                   cycle #  0  1  2  3  4  5  6  7  8  9  10 11
movq mystery, %rax          F  D  R  I  E  E  E  W  C
test %rax, %rax             F  D  R           I  E  W  C
jz skip (mispred.)             F  D  R              I  E  W  C
mov pointer, %rax              F  D  R  I  E  E  E  W
mov (%rax), %r8                   F  D  R              I  E  W
add $1, %r8                       F  D  R
mov %r8, %rax                        F  D  R
…
skip: ...                                       F  D  R
```

avoiding/triggering this problem

```
if (something false) {
    access *pointer;
}
```

what can we do to make access more/less likely to happen?

reading a value without really reading it

```
char *array;
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
if (something false) {
    other_array[mystery * BLOCK_SIZE] += 1;
}
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) {
        ...
    }
}
```

if branch mispredicted, cache access may still happen

can find the value of mystery

seeing past a segfault? (1)

```
Prime();
if (something false) {
    triggerSegfault();
    Use(*pointer);
}
Probe();
```

could cache access for *pointer still happen?

yes, if:

    branch for if statement mispredicted, and

    *pointer starts before segfault detected

seeing past a segfault? (2)

operations in virtual memory lookup:
> translate virtual to physical address
> check if access is permitted by permission bits

Intel processors: looks like these were separate steps, so...

```
Prime();
if (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

seeing past a segfault? (2)

operations in virtual memory lookup:
    translate virtual to physical address
    check if access is permitted by permission bits

Intel processors: looks like these were separate steps, so…

```
Prime();
if (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

seeing past a segfault? (2)

operations in virtual memory lookup:
>   translate virtual to physical address
>   check if access is permitted by permission bits

Intel processors: looks like these were separate steps, so...

```
Prime();
if (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

seeing past a segfault? (2)

operations in virtual memory lookup:
    translate virtual to physical address
    check if access is permitted by permission bits

Intel processors: looks like these were separate steps, so...

```
Prime();
if (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
    // %rcx = kernel address
    // %rbx = array to load from to cause eviction
    xor %rax, %rax        // rax <- 0
retry:
    // rax <- memory[kernel address] (segfaults)
        // but check for segfault done out-of-order on Intel
    movb (%rcx), %al
    // rax <- memory[kernel address] * 4096 [speculated]
    shl $0xC, %rax
    jz retry                    // not-taken branch
    // access array[memory[kernel address] * 4096]
    mov (%rbx, %rax), %rbx
```

Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
    // %rcx = ke[     space out accesses by 4096
    // %rbx = ar      ensure separate cache sets and      viction
    xor %rax, %ra     avoid triggering prefetcher
retry:
    // rax <- memory[kernel address] (segfaults)
        // but check for segfault done out-of-order on Intel
    movb (%rcx), %al
    // rax <- memory[kernel address] * 4096 [speculated]
    shl $0xC, %rax
    jz retry                    // not-taken branch
    // access array[memory[kernel address] * 4096]
    mov (%rbx, %rax), %rbx
```

Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
    // %rcx
    // %rbx                                          on
    xor %rax
retry:
    // rax <- memory[kernel address] (segfaults)
        // but check for segfault done out-of-order on Intel
    movb (%rcx), %al
    // rax <- memory[kernel address] * 4096 [speculated]
    shl $0xC, %rax
    jz retry                        // not-taken branch
    // access array[memory[kernel address] * 4096]
    mov (%rbx, %rax), %rbx
```

repeat access if zero
apparently value of zero speculatively read
when real value not yet available

Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
    // %rcx = kernel address
    // %rbx = probe array (base of it in regular location)
    xor %rax
retry:
    // rax <- memory[kernel address] (segfaults)
        // but check for segfault done out-of-order on Intel
    movb (%rcx), %al
    // rax <- memory[kernel address] * 4096 [speculated]
    shl $0xC, %rax
    jz retry                    // not-taken branch
    // access array[memory[kernel address] * 4096]
    mov (%rbx, %rax), %rbx
```

access cache to allow measurement later in paper with FLUSH+RELOAD instead of PRIME+PROBE technique

Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

segfault actually happens eventually
option 1: okay, just start a new process every time
option 2: way of suppressing exception (transactional memory support)

```
// rax <- memory[kernel address] (segfaults)
    // but check for segfault done out-of-order on Intel
movb (%rcx), %al
// rax <- memory[kernel address] * 4096 [speculated]
shl $0xC, %rax
jz retry                    // not-taken branch
// access array[memory[kernel address] * 4096]
mov (%rbx, %rax), %rbx
```

Meltdown fix

HW: permissions check done with/before physical address lookup
was already done by AMD, ARM apparently?
now done by Intel

SW: separate page tables for kernel and user space
don't have sensitive kernel memory pointed to by page table
when user-mode code running
unfortunate performance problem
exceptions start with code that switches page tables

reading a value without really reading it

```
char *array;
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
if (something false) {
    other_array[mystery * BLOCK_SIZE] += 1;
}
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) {
        ...
    }
}
```

if branch mispredicted, cache access may still happen

can find the value of mystery

mistraining branch predictor?

```
if (something) {
    CodeToRunSpeculatively()
}
```

how can we have 'something' be false, but predicted as true

run lots of times with something true

then do actually run with something false

contrived(?) vulnerable code (1)

suppose this C code is run with extra privileges

(e.g. in system call handler, library called from JavaScript in webpage, etc.)

assume x chosen by attacker

(example from original Spectre paper)

```
if (x < array1_size)
        y = array2[array1[x] * 4096];
```

the out-of-bounds access (1)

```
char array1[...];
...
int secret;
...
y = array2[array1[x] * 4096];
```

suppose array1 is at 0x1000000 and

secret is at 0x103F0003;

what x do we choose to make array1[x] access first byte of secret?

the out-of-bounds access (2)

```
char array1[...];
...
int secret;
...
y = array2[array1[x] * 4096];
```

suppose our cache has 64-byte blocks and 8192 sets

and array2[0] is stored in cache set 0

if the above evicts something in cache set 128,
then what do we know about array1[x]?

the out-of-bounds access (2)

```
char array1[...];
...
int secret;
...
y = array2[array1[x] * 4096];
```

suppose our cache has 64-byte blocks and 8192 sets

and array2[0] is stored in cache set 0

if the above evicts something in cache set 128,
then what do we know about array1[x]?
    is 2 or 130

exploit with contrived(?) code

```
/* in kernel: */
int systemCallHandler(int x) {
    if (x < array1_size)
        y = array2[array1[x] * 4096];
    return y;
}
```

```
/* exploiting code */
    /* step 1: mistrain branch predictor */
for (a lot) {
    systemCallHandler(0 /* less than array1_size */);
}
    /* step 2: evict from cache using misprediction */
Prime();
systemCallHandler(targetAddress - array1Address);
int evictedSet = ProbeAndFindEviction();
int targetValue = (evictedSet - array2StartSet) / setsPer4K;
```

really contrived?

```
char *array1; char *array2;
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

times 4096 shifts so we can get lower bits of target value

so all bits effect what cache block is used

really contrived?

```
char *array1; char *array2;
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

times 4096 shifts so we can get lower bits of target value

> so all bits effect what cache block is used

```
int *array1; int *array2;
if (x < array1_size)
    y = array2[array1[x]];
```

will still get *upper* bits of array1[x] (can tell from cache set)

can still read arbitrary memory!
> want memory at 0x10000?
> upper bits of 4-byte integer at 0x3FFFE

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]]);
}
```
our template

```
void SomeSystemCallHandler(int index) {
    if (index > some_table_size)
        return ERROR;
    int kind = table[index];
    switch (other_table[kind].foo) {
        ...
    }
}
```
actual code

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]]);
}
```
our template

```
void SomeSystemCallHandler(int index) {
    if (index > some_table_size)
        return ERROR;
    int kind = table[index];
    switch (other_table[kind].foo) {
        ...
    }
}
```
actual code

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]]);
}
```

our template

```
void SomeSystemCallHandler(int index) {
    if (index > some_table_size)
        return ERROR;
    int kind = table[index];
    switch (other_table[kind].foo) {
        ...
    }
}
```

actual code

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]]);
}
```

our template

```
void SomeSystemCallHandler(int index) {
    if (index > some_table_size)
        return ERROR;
    int kind = table[index];
    switch (other_table[kind].foo) {
        ...
    }
}
```

actual code

privilege levels?

vulnerable code runs with higher privileges

so far: higher privileges $=$ kernel mode

but other common cases of higher privileges

example: scripts in web browsers

JavaScript

JavaScript: scripts in webpages

not supposed to be able to read arbitrary memory, but…

can access arrays to examine caches

and could take advantage of some browser function being vulnerable

JavaScript

JavaScript: scripts in webpages

not supposed to be able to read arbitrary memory, but…

can access arrays to examine caches

and could take advantage of some browser function being vulnerable

or — doesn't even need browser to supply vulnerable code itself!

just-in-time compilation?

for performance, compiled to machine code, run in browser

not supposed to be access arbitrary browser memory

example JavaScript code from paper:

```
if (index < simpleByteArray.length) {
    index = simpleByteArray[index | 0];
    index = (((index * 4096)|0) & (32*1024*1024-1))|0;
    localJunk ^= probeTable[index|0]|0;
}
```

web page runs a lot to train branch predictor

then does run with out-of-bounds index

examines what's evicted by probeTable access

supplying own attack code?

JavaScript: could supply own attack code

turns out also possible with kernel mode scenario

trick: don't need to *actually run* code

...just need branch predictor to fetch it!

other misprediction

so far: talking about mispredicting direction of branch

what about mispredicting target of branch in, e.g.:

```
// possibly from C code like:
//   (*function_pointer)();
jmp *%rax

// possibly from C code like:
//      switch(rcx) { ... }
jmp *(%rax,%rcx,8)
```

an idea for predicting indirect jumps

for jmps like `jmp *%rax` predict target with cache:

bottom 12 bits of jmp address	last seen target
0x0–0x7	0x200000
0x8–0xF	0x440004
0x10-0x18	0x4CD894
0x18-0x20	0x510194
0x20-0x28	0x4FF194
…	…
0xFF8–0xFFF	0x3F8403

Intel Haswell CPU did something similar to this
  uses bits of last several jumps, not just last one

can mistrain this branch predictor

using mispredicted jump

1: find some kernel function with `jmp *%rax`

2: mistrain branch target predictor for it to jump to chosen code
   use code at address that conflicts in "recent jumps cache"

3: have chosen code be attack code (e.g. array access)
   either write special code OR
   find suitable instructions (e.g. array access) in existing kernel code

Spectre variants

showed Spectre variant 1 (array bounds), 2 (indirect jump)
> from original paper

other possible variations:
> could cause other things to be mispredicted
>> prediction of where functions return to?
>> values instead of which code is executed?
> could use side-channel other than data cache changes
>> instruction cache
>> cache of pending stores not yet committed
>> contention for resources on multi-threaded CPU core
>> branch prediction changes
>> …

some Linux kernel mitigations (1)

replace `array[x]` with
`array[x & ComputeMask(x, size)]`

...where ComputeMask() returns
    0 if $x >$ size
    `0xFFFF..F` if $x \le$ size

...and ComputeMask() does not use jumps:

```
mov x, %r8
mov size, %r9
cmp %r9, %r8
sbb %rax, %rax   // sbb = subtract with borrow
     // either 0 or -1
```

some Linux kernel mitigations (2)

for indirect branches:

with hardware help:
    separate indirect (computed) branch prediction for kernel v user mode
    other branch predictor changes to isolate better

without hardware help:
    transform `jmp *(%rax)`, etc. into code that
    will only predicted to jump to safe locations
    (by writing assembly very carefully)

only safe prediction

as replacement for `jmp *(%rax)`

code from Intel's "Retpoline: A Branch Target Injection Mitigation"

```
        call load_label
    capture_ret_spec:      /* <-- want prediction to go here */
        pause
        lfence
        jmp capture_ret_spec
    load_label:
        mov %rax, (%rsp)
        ret
```

backup slides

backup slides

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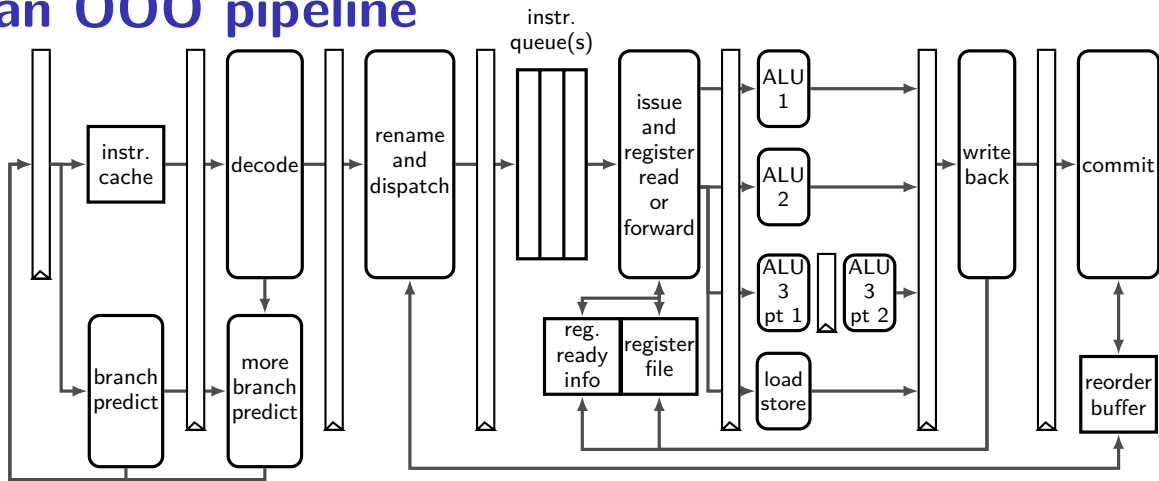
an OOO pipeline
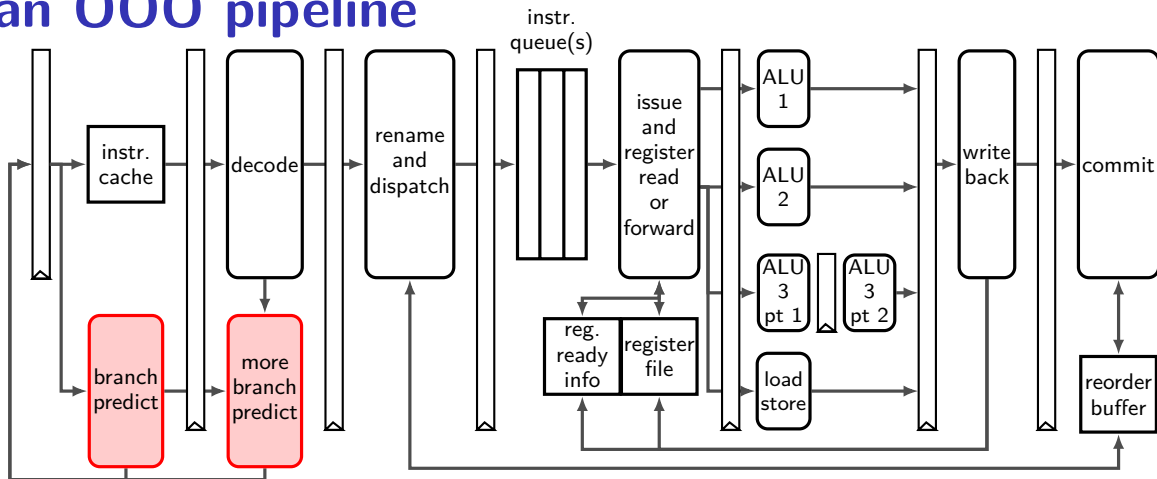
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

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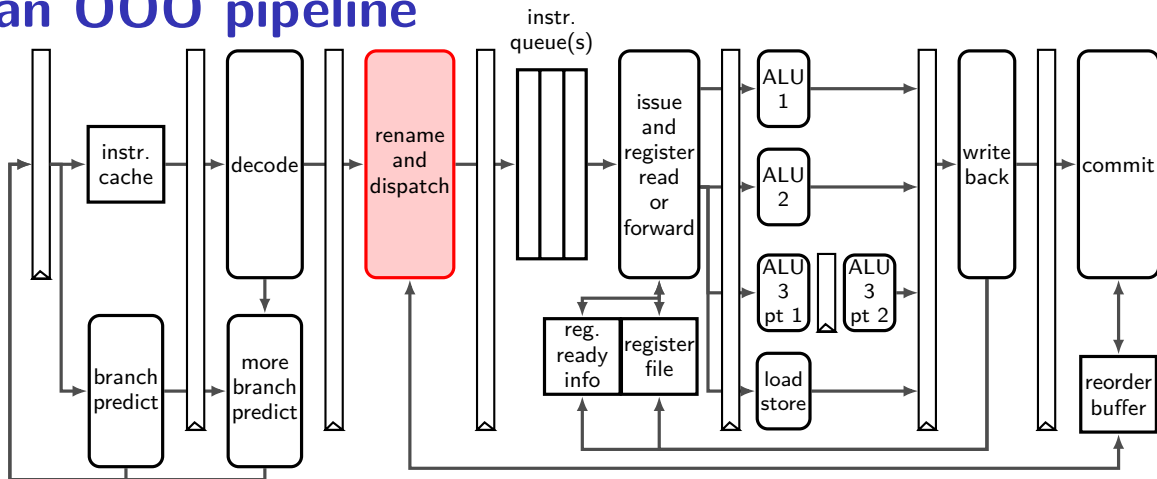
register renaming done here
stage needs to keep mapping from architectural to physical names

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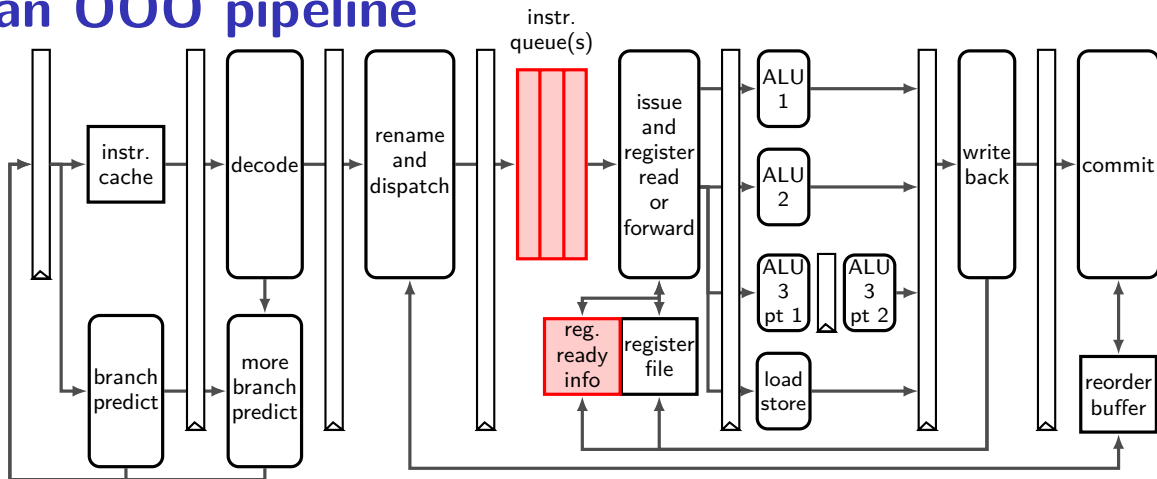
instruction queue holds pending renamed instructions
combined with register-ready info to *issue* instructions
(issue = start executing)

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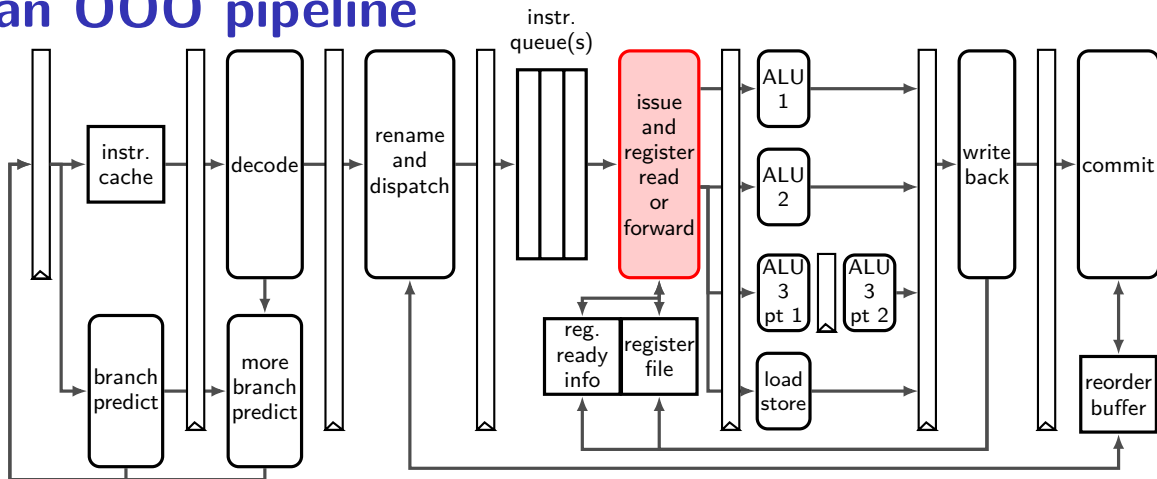
read from much larger register file and handle forwarding
register file: typically read $6+$ registers at a time
(extra data paths wires for forwarding not shown)

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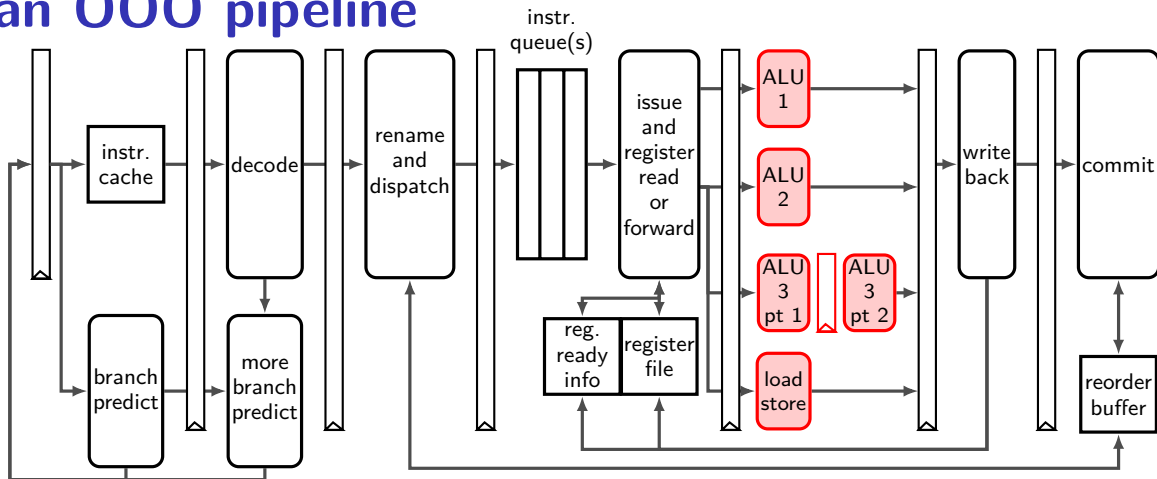
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, …)

an OOO pipeline
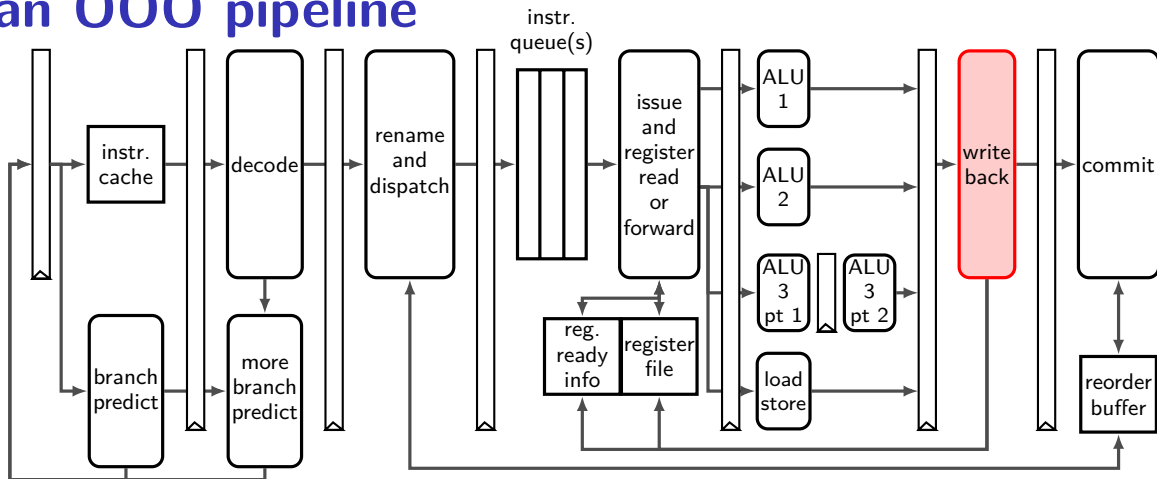
writeback results to physical registers
register file: typically support writing 3+ registers at a time

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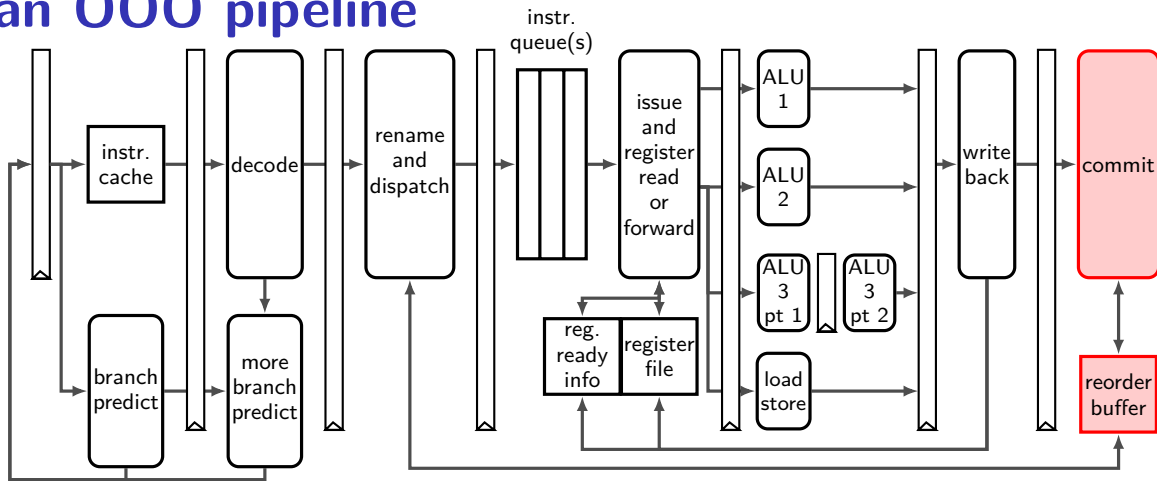
new commit (sometimes *retire*) stage finalizes instruction
figures out when physical registers can be reused again

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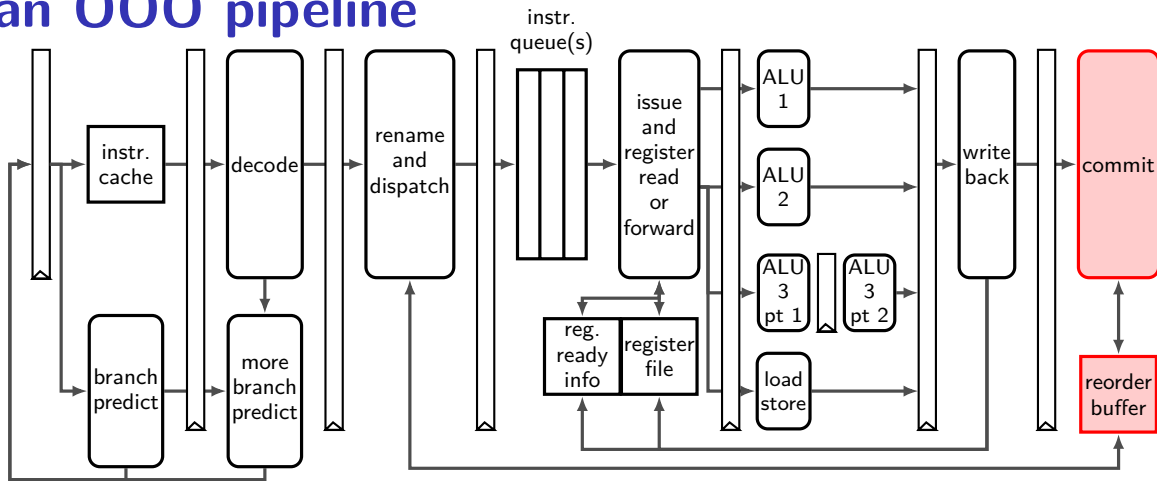
commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.
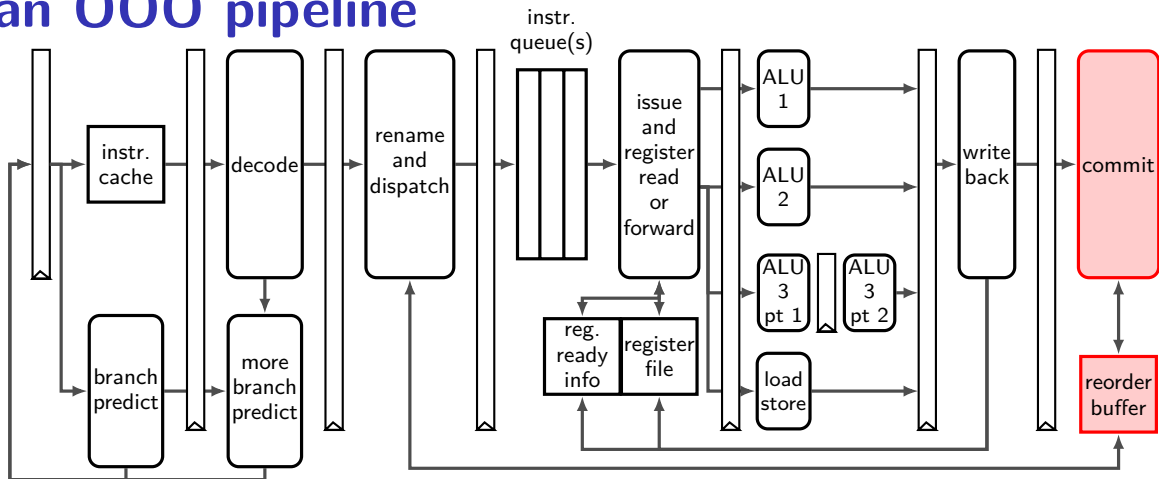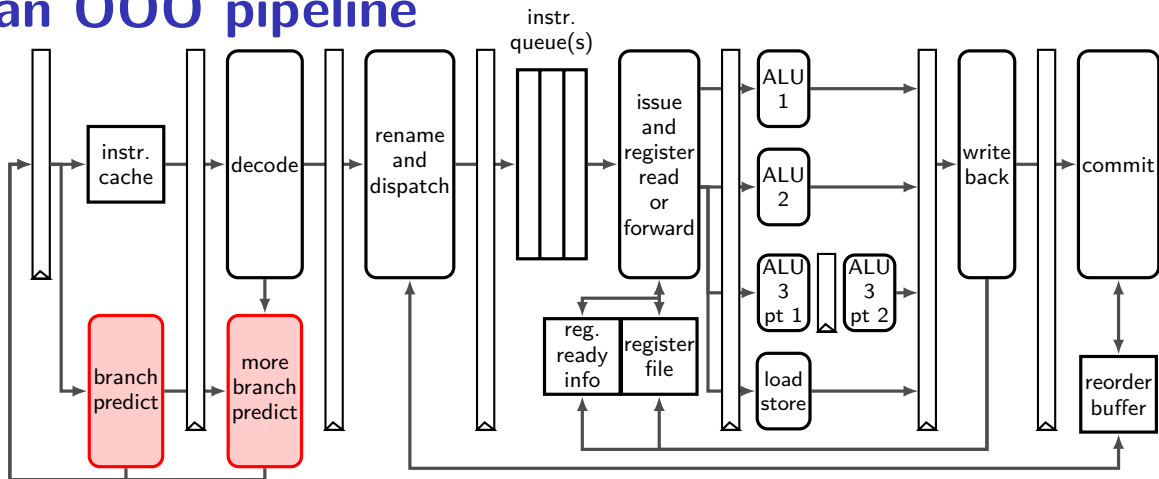
an OOO pipeline
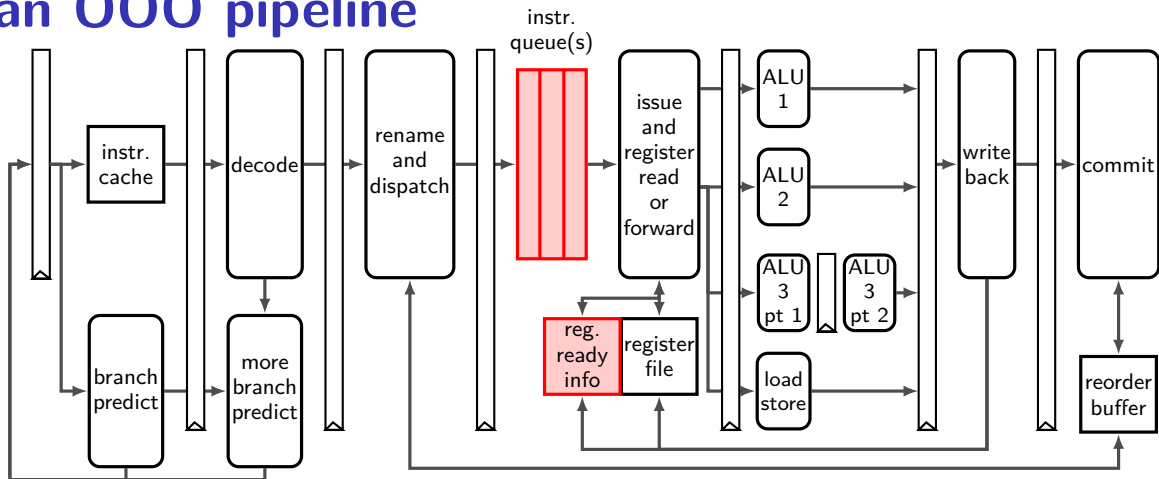
an OOO pipeline
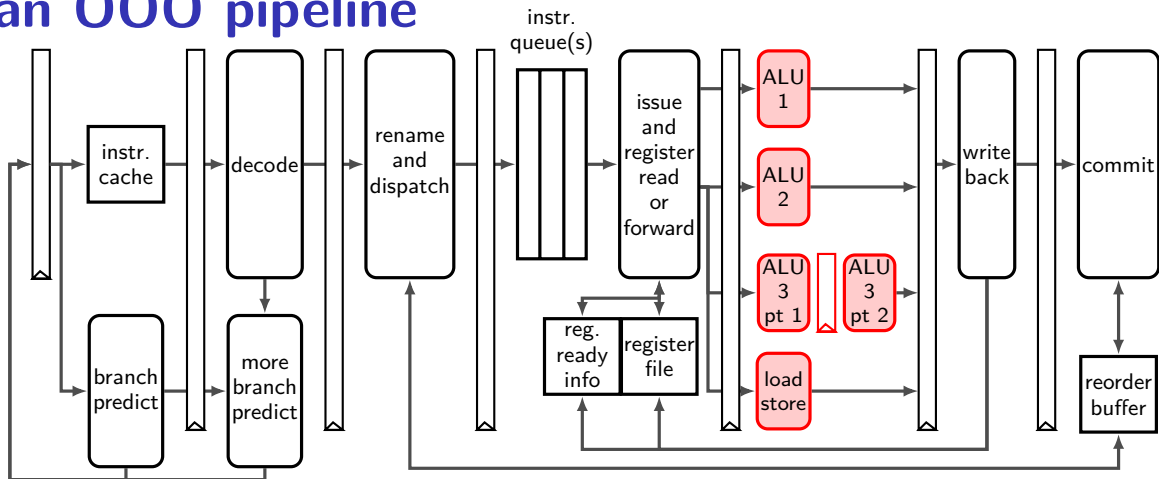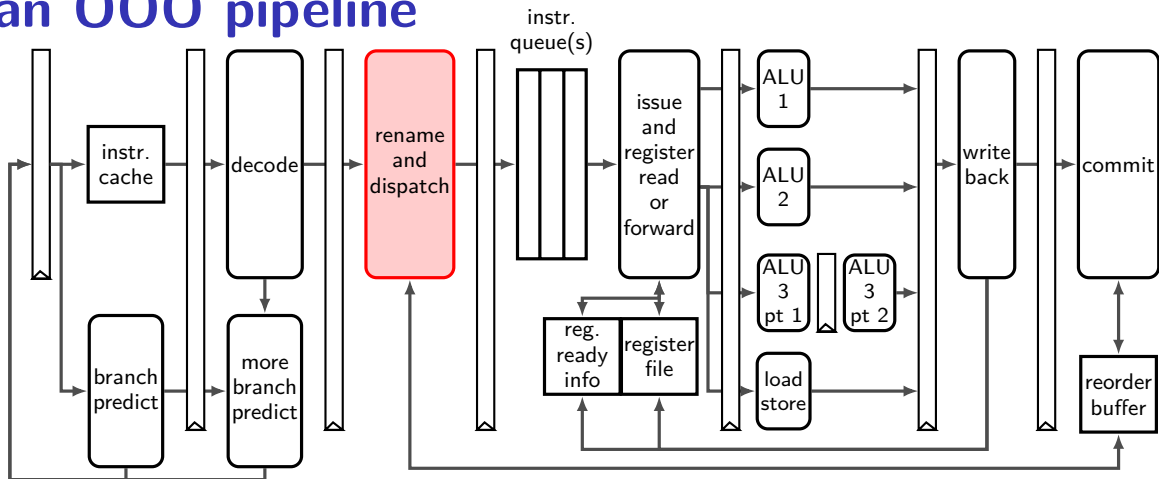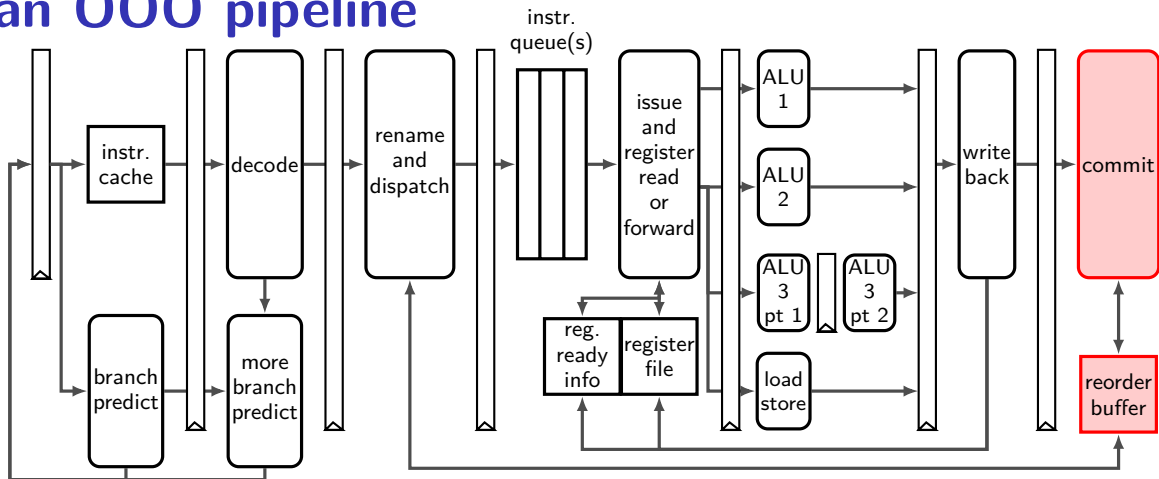
an OOO pipeline

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

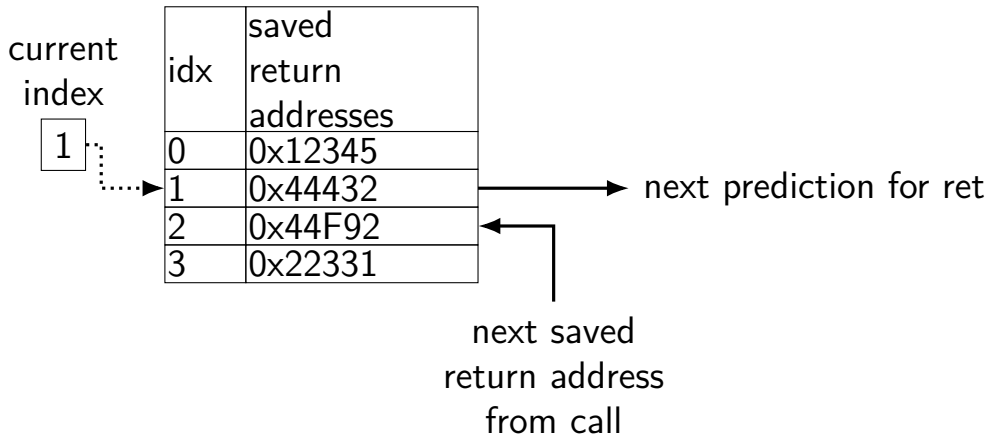| baz saved registers |
| --- |
| baz return address |
| bar saved registers |
| bar return address |
| foo local variables |
| foo saved registers |
| foo return address |
| foo saved registers |

stack in memory

baz return address
bar return address
foo return address

(partial?) stack
in CPU registers

4-entry return address stack

4-entry return address stack in CPU

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

on `ret`: read prediction from index, decrement index