

performance

hypothetical instruction mix

kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
others	92%	1*	1*

performance

hypothetical instruction mix

kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
others	92%	1*	1*

exercise: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

cycle # 0 1 2 3 4 5 6 7 8 9 10

(1) **addq** %rcx, %r9

(2) **jne** foo (not taken)

(3) **subq** %rax, %r9

(4) **call** bar

(5) bar: **pushq** %r9

exercise: with different pipeline

with F/D/E1/E2/M/W

cycle # 0 1 2 3 4 5 6 7 8 9 10

(1) **addq** %rcx, %r9

(2) **jne** foo (not taken)

(3) **subq** %rax, %r9

(4) **call** bar

(5) bar: **pushq** %r9

[solution]: with different pipeline

with F/D/E1/E2/M/W

	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) addq %rcx, %r9		F	D	E1	E2	M	W					
(2) jne foo (not taken)			F	D	E1	E2	M	W				
(2b) mispredicted				F	D	E1	E2	M	W			
(2c) mispredicted					F	D	E1	E2	M	W		
(2d) mispredicted						F	D	E1	E2	M	W	
(3) subq %rax, %r9						F	D	E1	E2	M	W	
(4) call bar							F	D	E1	E2	M	W
(5) bar: pushq %r9								F	D*	D	E1	E2

static branch prediction

forward (target > PC) not taken; backward taken

intuition: loops:

```
LOOP: ...
```

```
    ...
```

```
    je LOOP
```

```
LOOP: ...
```

```
    jne SKIP_LOOP
```

```
    ...
```

```
    jmp LOOP
```

```
SKIP_LOOP:
```

exercise: static prediction

```
.global foo
foo:
    xor %eax, %eax // eax <- 0
foo_loop_top:
    test $0x1, %edi
    je foo_loop_bottom // if (edi & 1 == 0) goto .Lskip
    add %edi, %eax
foo_loop_bottom:
    dec %edi // edi = edi - 1
    jl .Lend_loop // if (edi < 0) goto .Lend_loop
    ret
```

suppose $\%edi = 3$ (initially)

and using forward-taken, backwards-not-taken strategy:

how many mispredictions for je? for jl?

branch target buffer

what can we do to predict `jmp *(%rax)`?

what if we can't decode LABEL from machine code for `jmp LABEL` fast?

will happen in more complex pipelines

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)	valid	...
0x00	1	0x400	5	Jxx	0x3FFFF3	...	1	...
0x01	1	0x401	C	JMP	0x401035	----	0	...
0x02	0	---	---	---	---	---	0	...
0x03	1	0x400	9	RET	---	...	0	...
...
0xFF	1	0x3FF	8	CALL	0x404033	...	0	...

0x3FFFF3: movq %rax, %rsi
0x3FFFF7: pushq %rbx
0x3FFFF8: call 0x404033
0x400001: popq %rbx
0x400003: cmpq %rbx, %rax
0x400005: jle 0x3FFFF3
...
0x400031: ret
...

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	...
0x01	1	0x401	C	JMP	0x401035	----
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	...
...
0xFF	1	0x3FF	8	CALL	0x404033	...

valid	...
1	...
0	...
0	...
0	...
...	...
0	...

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
...
0x400031:  ret
...
```

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	...
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	...
...
0xFF	1	0x3FF	8	CALL	0x404033	...

valid	...
1	...
0	...
0	...
0	...
...	...
0	...

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
...
0x400031:  ret
...
```

beyond pipelining: multiple issue

start **more than one instruction/cycle**

multiple parallel pipelines; many-input/output register file

hazard handling much more complex

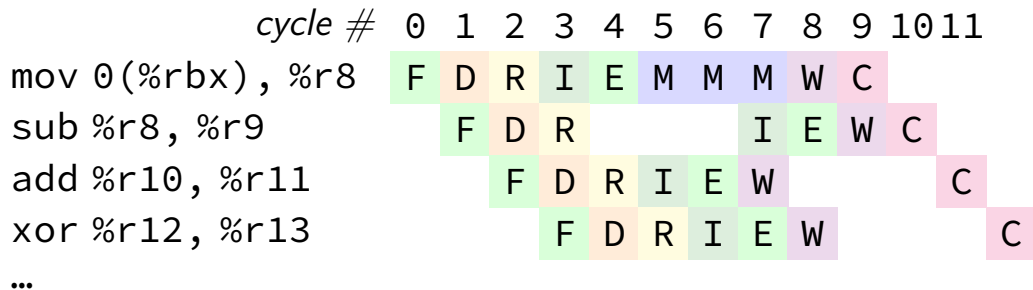
	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %r8, %r9	F	D	E	M	W					
subq %r10, %r11	F	D	E	M	W					
xorq %r9, %r11		F	D	E	M	W				
subq %r10, %rbx		F	D	E	M	W				
...										

beyond pipelining: out-of-order

find **later instructions to do** instead of stalling

lists of available instructions in pipeline registers
take any instruction with available values

provide **illusion that work is still done in order**
much more complicated hazard handling logic



interlude: real CPUs

modern CPUs:

execute **multiple instructions at once**

execute instructions **out of order** — whenever **values available**

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

- value in last stage may not be most up-to-date

- older value may be written back before newer value?

problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

- value in last stage may not be most up-to-date

- older value may be written back before newer value?

problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

read-after-write examples (1)

	cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8		F	D	E	M	W				
addq %r11, %r8			F	D	E	M	W			
addq %r12, %r8				F	D	E	M	W		

normal pipeline: two options for %r8?

choose the one from *earliest stage*

because it's from the most recent instruction

read-after-write examples (1)

out-of-order execution:

%r8 from earliest stage might be from *delayed instruction*
can't use same forwarding logic

addq %r11, %r8
addq %r12, %r8

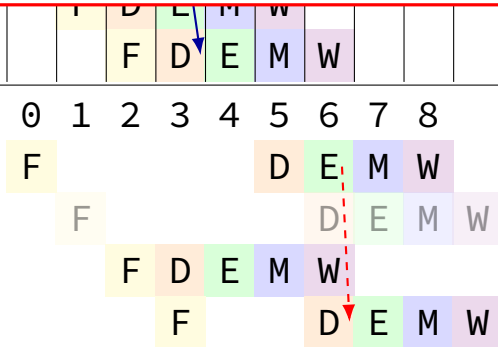
cycle # 0 1 2 3 4 5 6 7 8

addq %r10, %r8

rmmovq %r8, (%rax)

irmovq \$100, %r8

addq %r13, %r8



register version tracking

goal: track **different versions of registers**

out-of-order execution: may compute versions at different times

only forward the **correct version**

strategy for doing this: preprocess instructions represent version info

makes forwarding, etc. lookup easier

rewriting hazard examples (1)

addq %r10, %r8		addq %r10, %r8 _{v1} → %r8 _{v2}
addq %r11, %r8		addq %r11, %r8 _{v2} → %r8 _{v3}
addq %r12, %r8		addq %r12, %r8 _{v3} → %r8 _{v4}

read different version than the one written

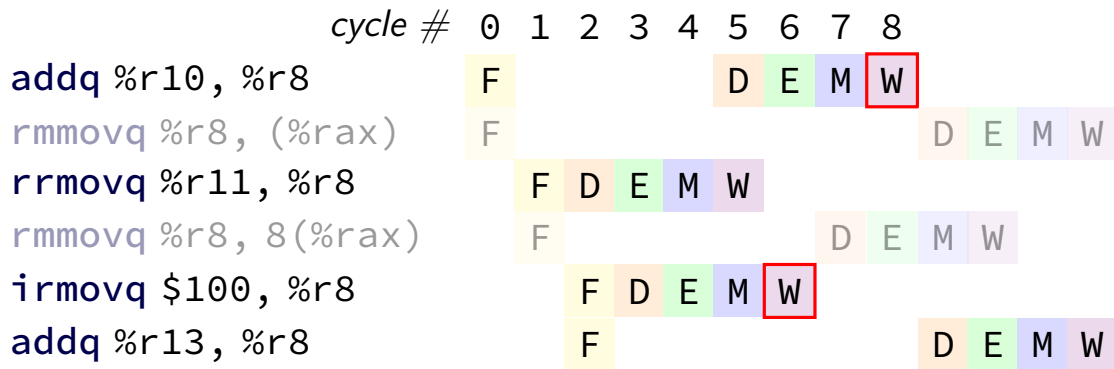
represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

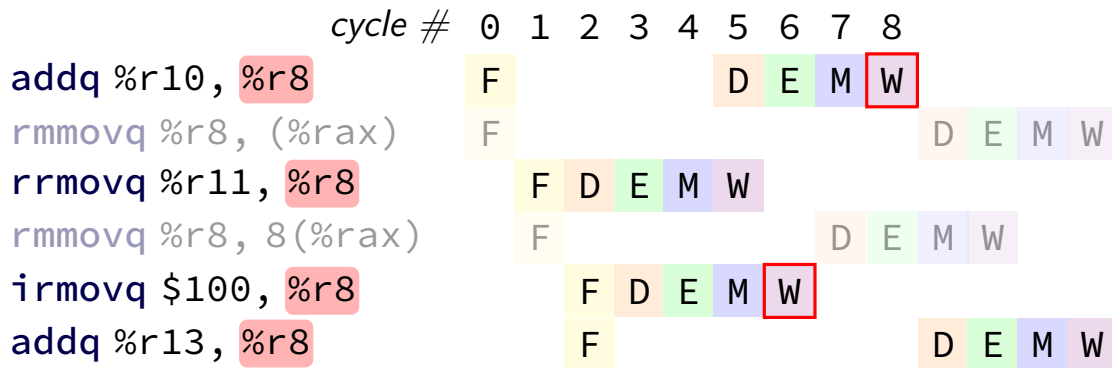
for now: version numbers

later: something simpler to implement

write-after-write example



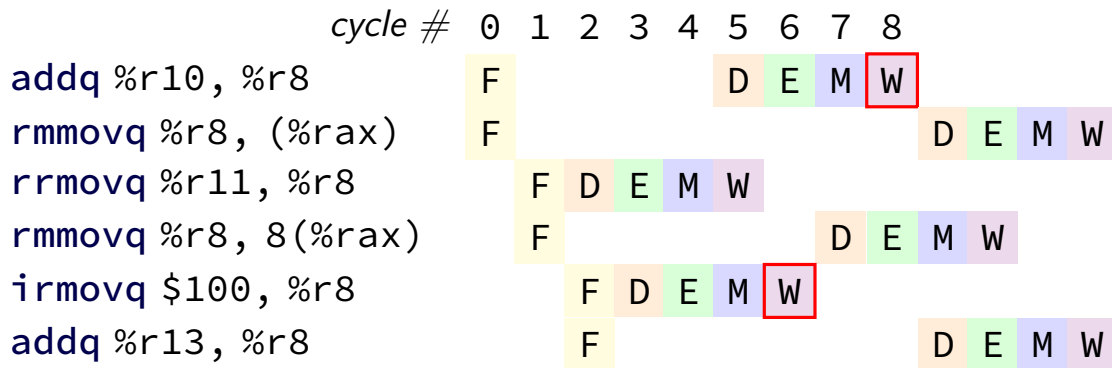
write-after-write example



out-of-order execution:

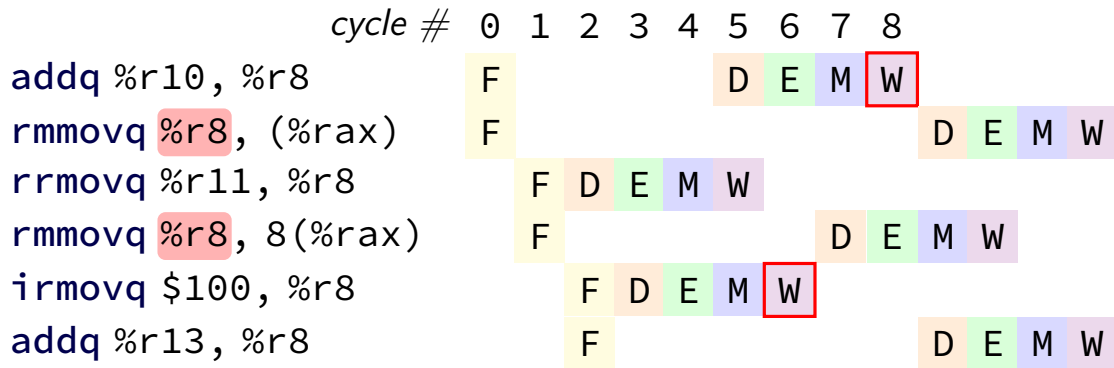
if we don't do something, newest value could be overwritten!

write-after-write example



two instructions that haven't been started
could need *different versions* of %r8!

write-after-write example



keeping multiple versions

for write-after-write problem: need to keep copies of multiple versions

both the new version and the old version needed by delayed instructions

for read-after-write problem: need to distinguish different versions

solution: have lots of extra registers

...and assign each version a new 'real' register

called register renaming

register renaming

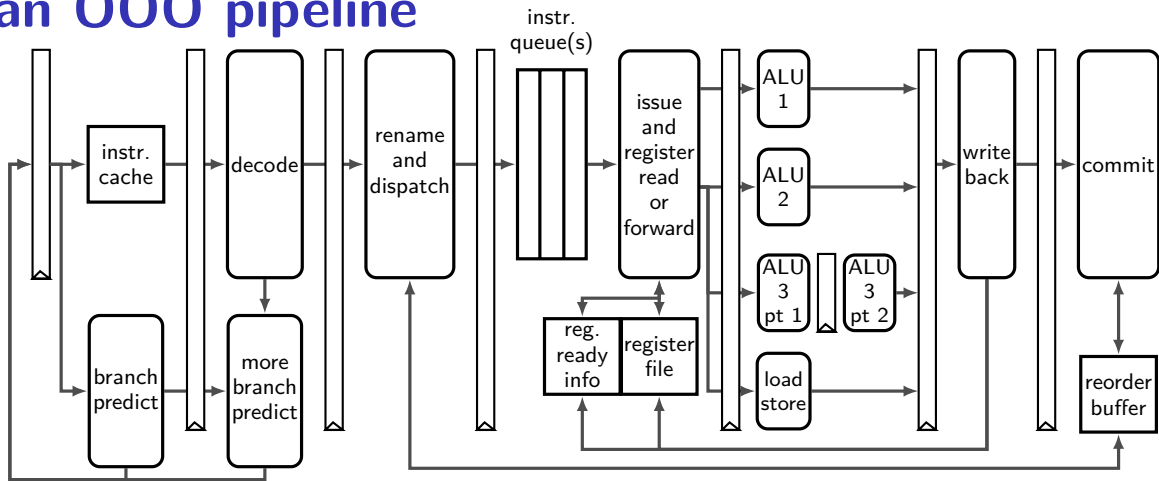
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

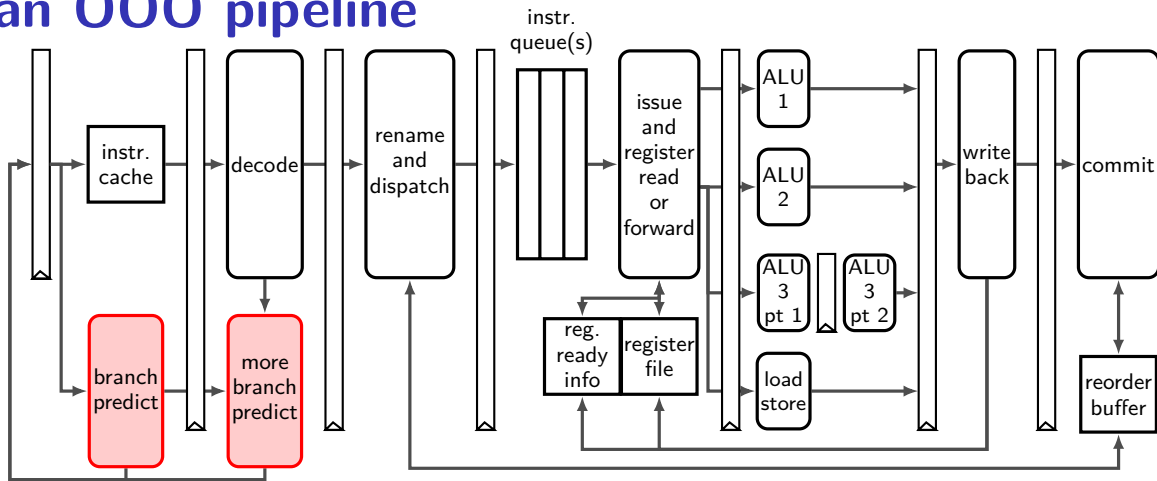
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

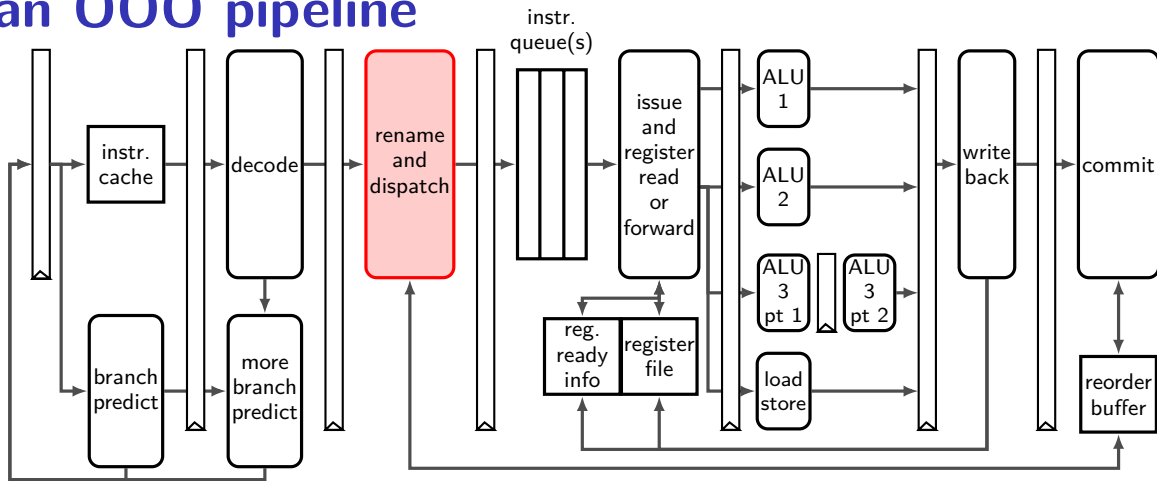


an OOO pipeline



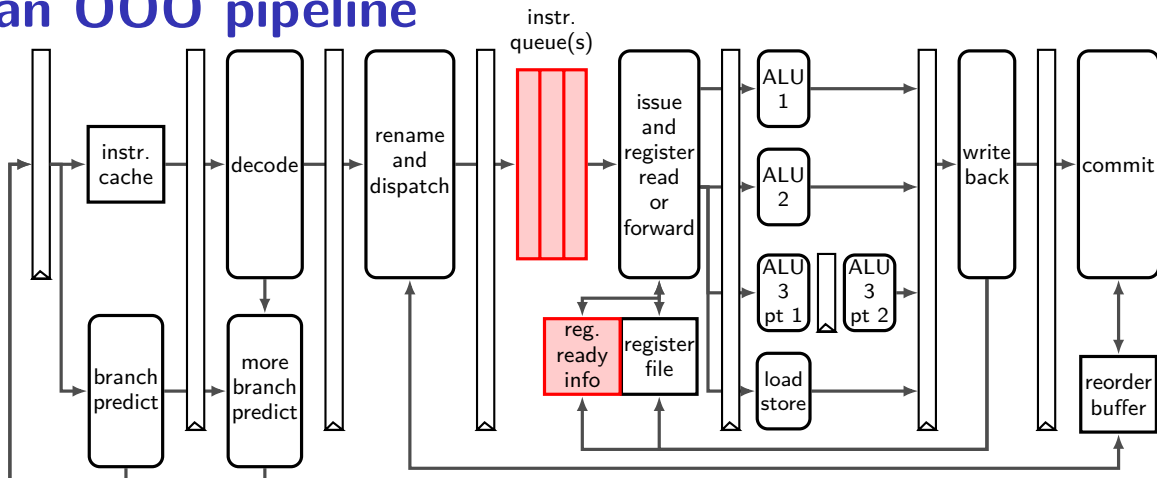
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



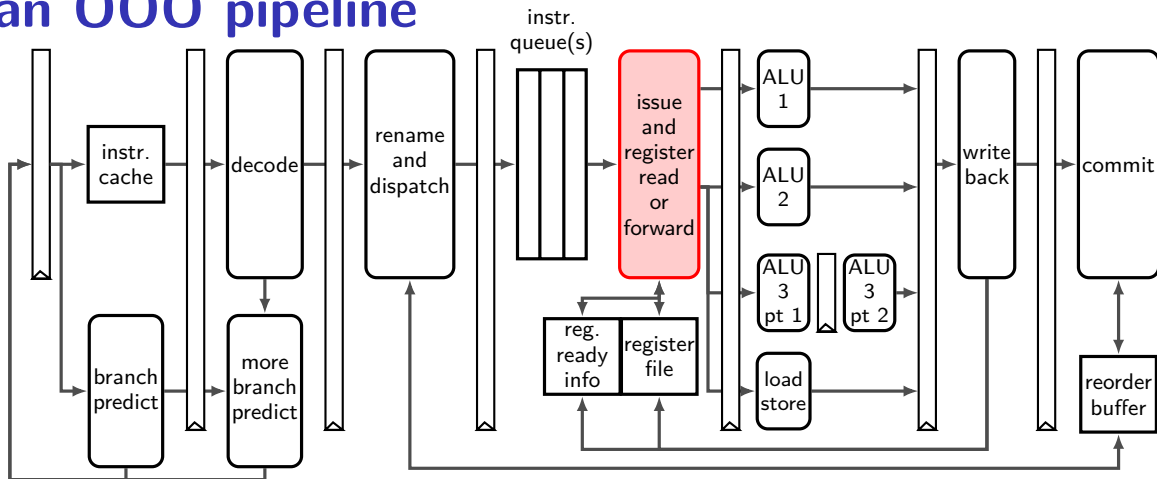
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



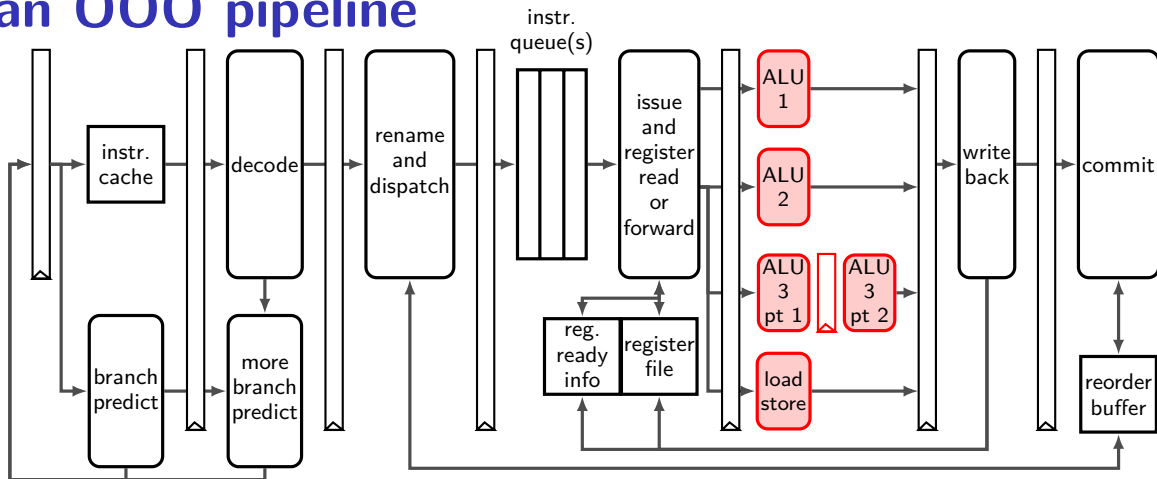
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



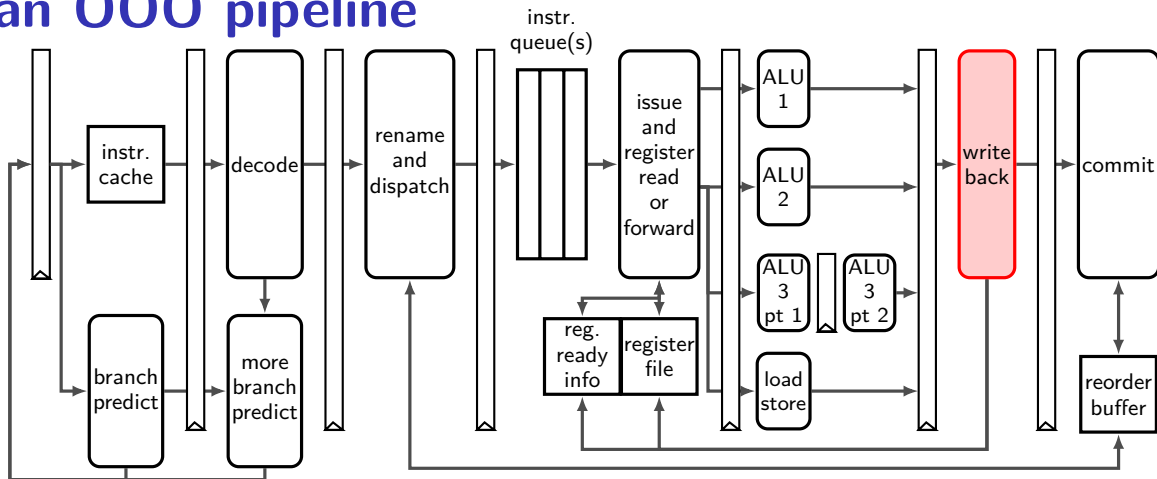
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



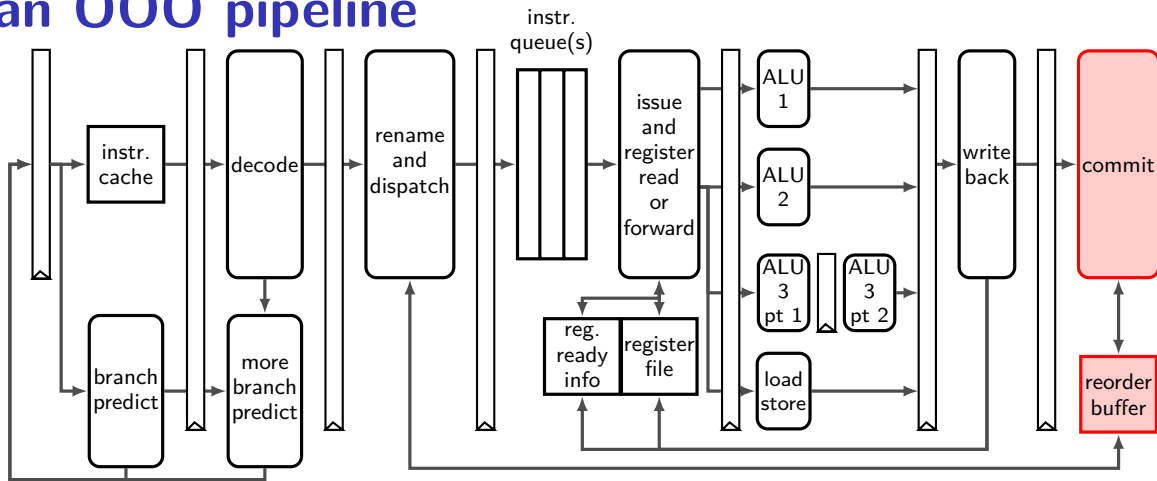
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



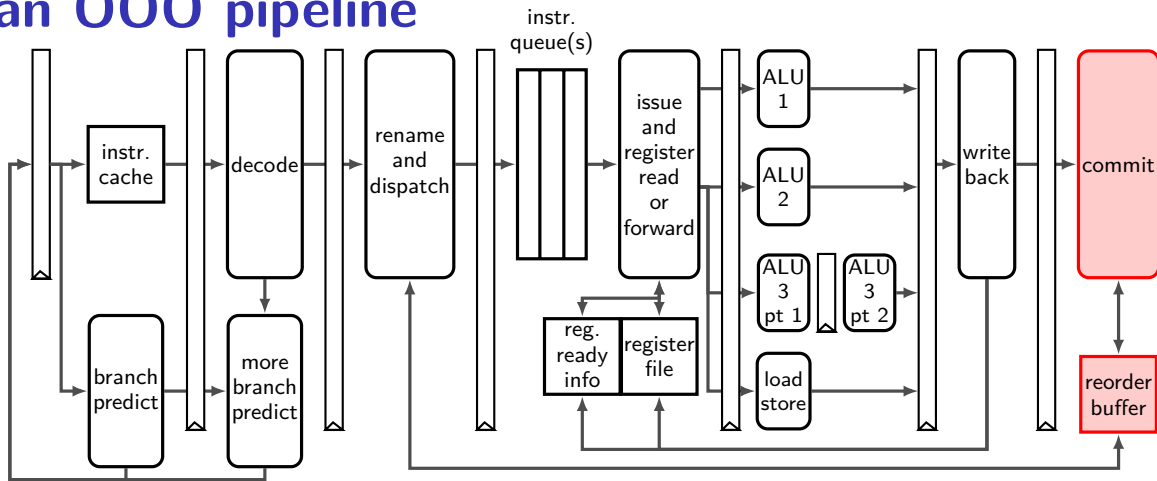
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



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

an OOO pipeline

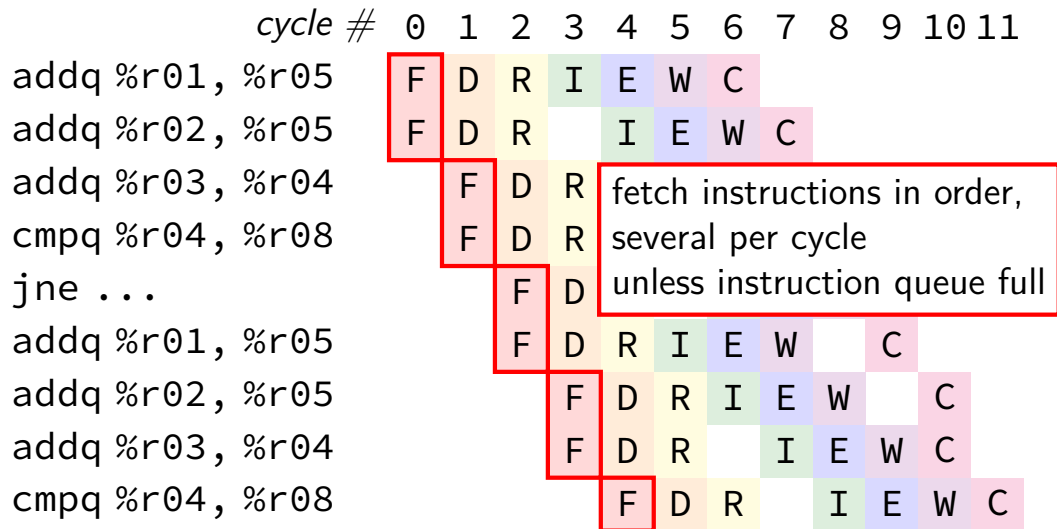


commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.

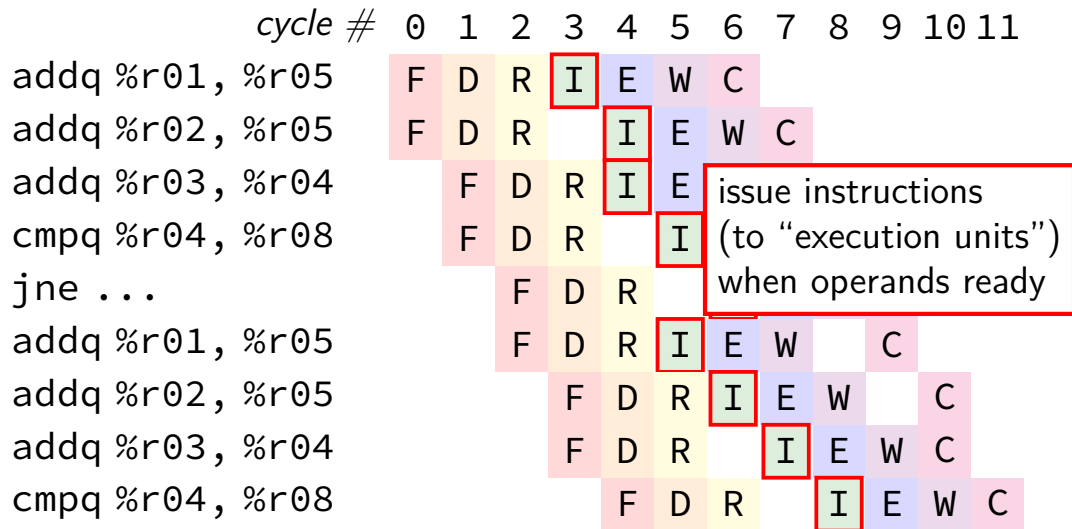
an OOO pipeline diagram

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8	9	10	11
addq %r01, %r05		F	D	R	I	E	W	C					
addq %r02, %r05		F	D	R		I	E	W	C				
addq %r03, %r04			F	D	R	I	E	W	C				
cmpq %r04, %r08			F	D	R		I	E	W	C			
jne ...				F	D	R		I	E	W	C		
addq %r01, %r05				F	D	R	I	E	W		C		
addq %r02, %r05					F	D	R	I	E	W		C	
addq %r03, %r04					F	D	R		I	E	W	C	
cmpq %r04, %r08						F	D	R		I	E	W	C

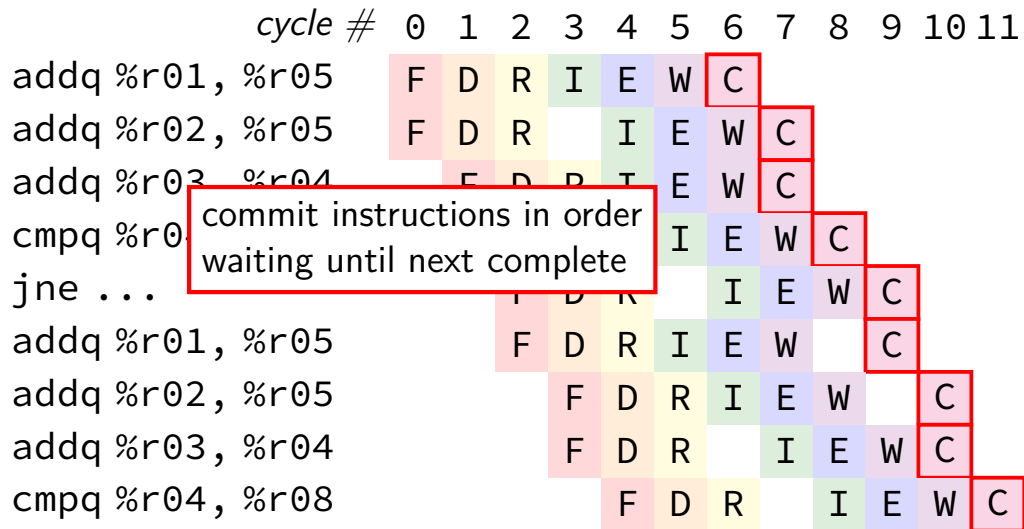
an OOO pipeline diagram



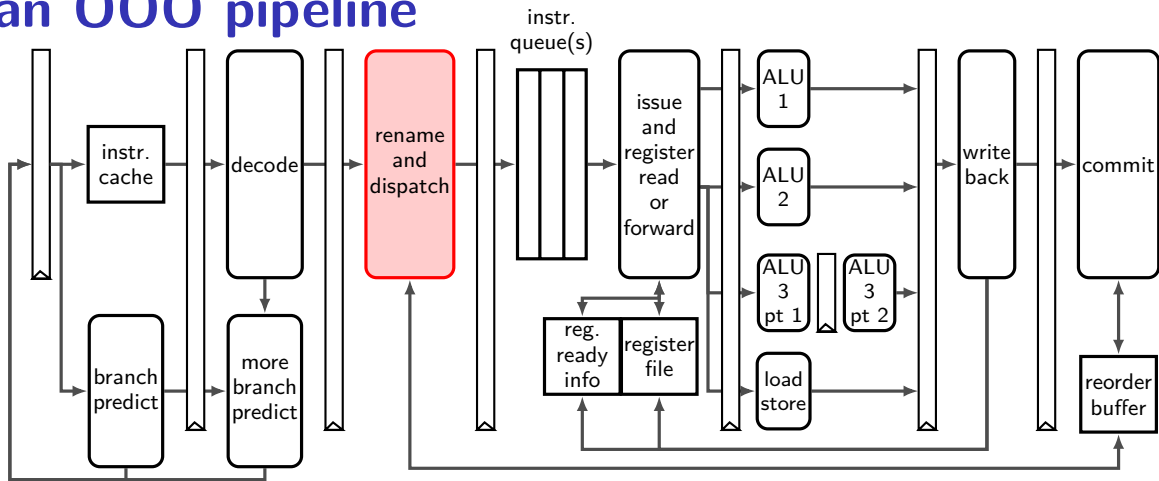
an OOO pipeline diagram



an OOO pipeline diagram



an OOO pipeline



register renaming

rename *architectural registers* to *physical registers*

architectural = part of instruction set architecture

different name for each version of architectural register

register renaming state

original renamed

add %r10, %r8 ...

add %r11, %r8 ...

add %r12, %r8 ...

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming state

original
`add %r10, %r8 ...`
`add %r11, %r8 ...`
`add %r12, %r8 ...`

renamed

table for architectural (external)
and physical (internal) name
(for next instr. to process)

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming state

original
`add %r10, %r8 ...`
`add %r11, %r8 ...`
`add %r12, %r8 ...`

renamed

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

list of available physical registers
added to as instructions finish

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

original

```
add %r10, %r8  
add %r11, %r8  
add %r12, %r8
```

renamed

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

original

```
add %r10, %r8  
add %r11, %r8  
add %r12, %r8
```

renamed

```
add %x19, %x13 → %x18
```

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13 %x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

original	renamed
add %r10, %r8	add %x19, %x13 → %x18
add %r11, %r8	add %x07, %x18 → %x20
add %r12, %r8	

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13% x18 %x20
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

original	renamed
add %r10, %r8	add %x19, %x13 → %x18
add %r11, %r8	add %x07, %x18 → %x20
add %r12, %r8	add %x05, %x20 → %x21

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13%x18% x20 %x21
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

original	renamed
add %r10, %r8	add %x19, %x13 → %x18
add %r11, %r8	add %x07, %x18 → %x20
add %r12, %r8	add %x05, %x20 → %x21

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13%x18%x20%x21
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
```

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13 %x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
rmmovq %x18, (%x04) → (memory)
```

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original	renamed
<code>addq %r10, %r8</code>	<code>addq %x19, %x13 → %x18</code>
<code>rmmovq %r8, (%rax)</code>	<code>rmmovq %x18, (%x04) → (memory)</code>
<code>subq %r8, %r11</code>	
<code>mrmmovq 8(%r11), %r11</code>	
<code>irmovq \$100, %r8</code>	
<code>addq %r11, %r8</code>	

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

could be that $\%rax = 8 + \%r11$
could load before value written!
possible data hazard!

not handled via register renaming

option 1: run load+stores in order

option 2: compare load/store addresses

%x21
%x23
%x24
...

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
rmmovq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
```

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13%x18
%r9	%x17
%r10	%x19
%r11	%x07 %x20
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original	renamed
<code>addq %r10, %r8</code>	<code>addq %x19, %x13 → %x18</code>
<code>rmmovq %r8, (%rax)</code>	<code>rmmovq %x18, (%x04) → (memory)</code>
<code>subq %r8, %r11</code>	<code>subq %x18, %x07 → %x20</code>
<code>mrmmovq 8(%r11), %r11</code>	<code>mrmmovq 8(%x20), (memory) → %x21</code>
<code>irmovq \$100, %r8</code>	
<code>addq %r11, %r8</code>	

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13%x18
%r9	%x17
%r10	%x19
%r11	%x07%x20%x21
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original	renamed
<code>addq %r10, %r8</code>	<code>addq %x19, %x13 → %x18</code>
<code>rmmovq %r8, (%rax)</code>	<code>rmmovq %x18, (%x04) → (memory)</code>
<code>subq %r8, %r11</code>	<code>subq %x18, %x07 → %x20</code>
<code>mrmmovq 8(%r11), %r11</code>	<code>mrmmovq 8(%x20), (memory) → %x21</code>
<code>irmovq \$100, %r8</code>	<code>irmovq \$100 → %x23</code>
<code>addq %r11, %r8</code>	

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13 %x18 %x23
%r9	%x17
%r10	%x19
%r11	%x07% x20 %x21
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
%x24
...

register renaming example (2)

original	renamed
<code>addq %r10, %r8</code>	<code>addq %x19, %x13 → %x18</code>
<code>rmmovq %r8, (%rax)</code>	<code>rmmovq %x18, (%x04) → (memory)</code>
<code>subq %r8, %r11</code>	<code>subq %x18, %x07 → %x20</code>
<code>mrmmovq 8(%r11), %r11</code>	<code>mrmmovq 8(%x20), (memory) → %x21</code>
<code>irmovq \$100, %r8</code>	<code>irmovq \$100 → %x23</code>
<code>addq %r11, %r8</code>	<code>addq %x21, %x23 → %x24</code>

arch → phys
register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13% x18 % x23 %x24
%r9	%x17
%r10	%x19
%r11	%x07%x20%x21
%r12	%x05
%r13	%x02

free
regs

%x18
%x20
%x21
%x23
% x24
...

register renaming exercise

original

```
addq %r8, %r9
movq $100, %r10
subq %r10, %r8
xorq %r8, %r9
andq %rax, %r9
```

arch → phys

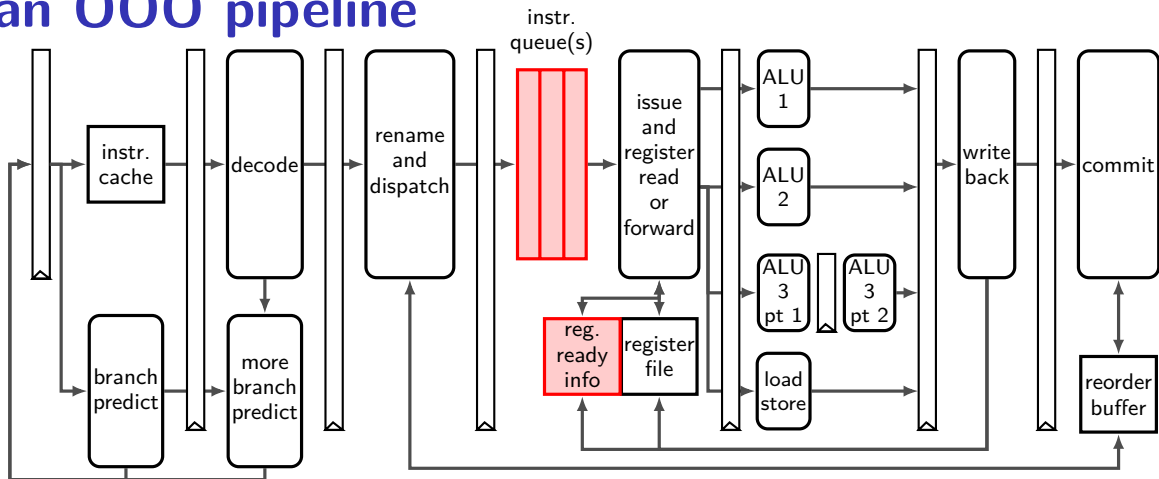
%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x29
%r12	%x05
%r13	%x02
...	...

renamed

free
regs

%x18
%x20
%x21
%x23
%x24
...

an OOO pipeline



instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc

... ..

execution unit

ALU 1

ALU 2

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

...

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit cycle# 1

ALU 1 1

ALU 2

...

instruction queue and dispatch

instruction queue

#	instruction
1	<code>addq %x01, %x05 → %x06</code>
2	<code>addq %x02, %x06 → %x07</code>
3	<code>addq %x03, %x07 → %x08</code>
4	<code>cmpq %x04, %x08 → %x09.cc</code>
5	<code>jne %x09.cc, ...</code>
6	<code>addq %x01, %x08 → %x10</code>
7	<code>addq %x02, %x10 → %x11</code>
8	<code>addq %x03, %x11 → %x12</code>
9	<code>cmpq %x04, %x12 → %x13.cc</code>

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit cycle# 1

ALU 1 1

ALU 2

...

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit cycle# 1

ALU 1 1

ALU 2 —

...

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2
ALU 1		1	2
ALU 2		—	—

...

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2	3	...
ALU 1		1	2	3	
ALU 2		—	—	—	

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2	3
ALU 1		1	2	3
ALU 2		—	—	—

...

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending ready
%x10	pending ready
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2	3	4	...
ALU 1		1	2	3	4	
ALU 2		—	—	—	6	

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending ready
%x10	pending ready
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2	3	4	...
ALU 1		1	2	3	4	
ALU 2		—	—	—	6	

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending ready
%x10	pending ready
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2	3	4	5	...
ALU 1		1	2	3	4	5	
ALU 2		—	—	—	6	7	

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending ready
%x10	pending ready
%x11	pending ready
%x12	pending
%x13	pending
...	...

execution unit	cycle#	1	2	3	4	5	6	...
ALU 1		1	2	3	4	5	8	
ALU 2		—	—	—	6	7	—	

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending ready
%x10	pending ready
%x11	pending ready
%x12	pending ready
%x13	pending
...	...

execution unit	cycle#	1	2	3	4	5	6	7	...
ALU 1		1	2	3	4	5	8	9	
ALU 2		—	—	—	6	7	—	...	

instruction queue and dispatch

instruction queue

#	instruction
1	addq %x01, %x05 → %x06
2	addq %x02, %x06 → %x07
3	addq %x03, %x07 → %x08
4	cmpq %x04, %x08 → %x09.cc
5	jne %x09.cc, ...
6	addq %x01, %x08 → %x10
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	pending ready
%x08	pending ready
%x09	pending ready
%x10	pending ready
%x11	pending ready
%x12	pending ready
%x13	pending ready
...	...

execution unit	cycle#	1	2	3	4	5	6	7	...
ALU 1		1	2	3	4	5	8	9	
ALU 2		—	—	—	6	7	—	...	

instruction queue and dispatch

instruction queue

#	instruction
1	mrmovq (%x04) → %x06
2	mrmovq (%x05) → %x07
3	addq %x01, %x02 → %x08
4	addq %x01, %x06 → %x09
5	addq %x01, %x07 → %x10

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	
%x07	
%x08	
%x09	
%x10	
...	...

execution unit cycle# 1 2 3 4 5 6 7 ...

ALU

data cache



assume

1 cycle/access

register renaming: missing pieces

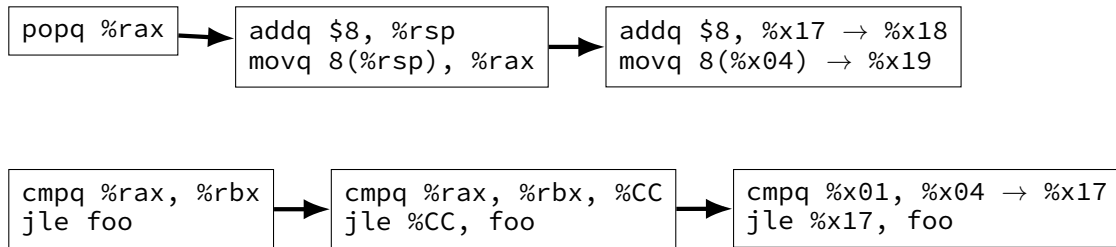
what about “hidden” inputs like `%rsp`, condition codes?

one solution: translate to instructions with additional register parameters

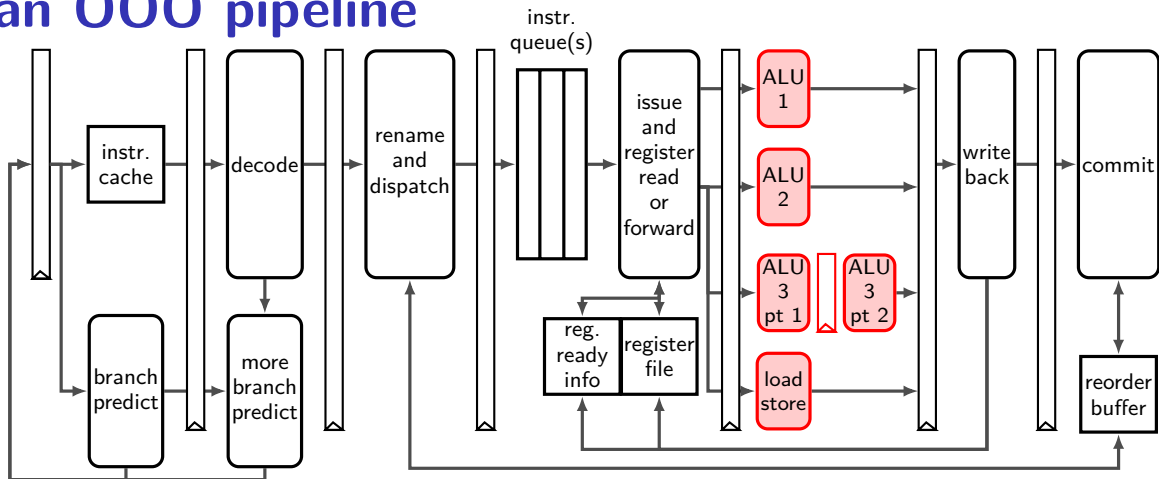
making `%rsp` explicit parameter

turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones



an OOO pipeline



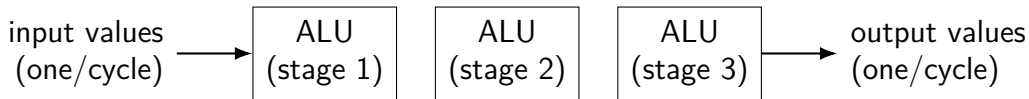
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



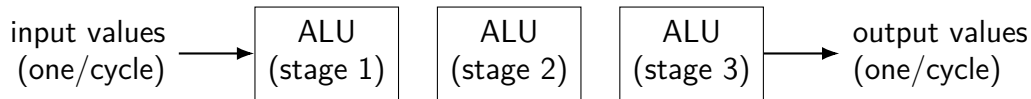
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



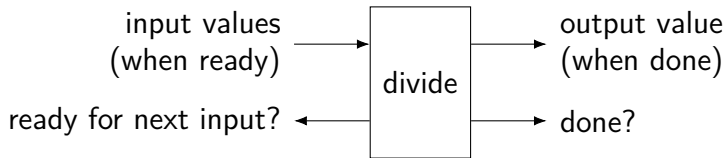
exercise: how long to compute $A \times (B \times (C \times D))$?

execution units AKA functional units (2)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:



instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ..

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ..

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ...

execution unit cycle# 1

ALU 1 (add, cmp, jxx) 1

ALU 2 (add, cmp, jxx) -

ALU 3 (mul) start 2

ALU 3 (mul) end 2

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending (still)
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit	cycle#	1	2
ALU 1 (add, cmp, jxx)	1	6	
ALU 2 (add, cmp, jxx)	—	—	
ALU 3 (mul) start	2	3	
ALU 3 (mul) end		2	3

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	ready
%x08	pending (still)
%x09	pending
%x10	pending
%x11	pending ready
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit	cycle#	1	2	3
ALU 1 (add, cmp, jxx)		1	6	—
ALU 2 (add, cmp, jxx)		—	—	—
ALU 3 (mul) start		2	3	7
ALU 3 (mul) end			2	3

7

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	ready
%x08	pending ready
%x09	pending ready
%x10	pending
%x11	pending ready
%x12	pending (still)
%x13	pending
%x14	pending
...	...

...

execution unit	cycle#	1	2	3	4	...
ALU 1 (add, cmp, jxx)		1	6	—	4	
ALU 2 (add, cmp, jxx)		—	—	—	—	
ALU 3 (mul) start		2	3	7	8	
ALU 3 (mul) end			2	3	7	8

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	ready
%x08	pending ready
%x09	pending ready
%x10	pending
%x11	pending ready
%x12	pending ready
%x13	pending (still)
%x14	pending
...	...

execution unit	cycle#	1	2	3	4	5	...
ALU 1 (add, cmp, jxx)		1	6	—	4	5	
ALU 2 (add, cmp, jxx)		—	—	—	—	—	
ALU 3 (mul) start		2	3	7	8	—	
ALU 3 (mul) end			2	3	7	8	

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	ready
%x08	pending ready
%x09	pending ready
%x10	pending
%x11	pending ready
%x12	pending ready
%x13	pending ready
%x14	pending
...	...

execution unit	cycle#	1	2	3	4	5	...
ALU 1 (add, cmp, jxx)		1	6	—	4	5	
ALU 2 (add, cmp, jxx)		—	—	—	—	—	
ALU 3 (mul) start		2	3	7	8	—	
ALU 3 (mul) end			2	3	7	8	

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ..

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	ready
%x08	pending ready
%x09	pending ready
%x10	pending
%x11	pending ready
%x12	pending ready
%x13	pending ready
%x14	pending ready
...	...

execution unit	cycle#	1	2	3	4	5	6	...
ALU 1 (add, cmp, jxx)		1	6	—	4	5	9	
ALU 2 (add, cmp, jxx)		—	—	—	—	—	—	
ALU 3 (mul) start		2	3	7	8	—		
ALU 3 (mul) end			2	3	7	8		

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
1	add %x01, %x02 → %x03
2	imul %x04, %x05 → %x06
3	imul %x03, %x07 → %x08
4	cmp %x03, %x08 → %x09.cc
5	jle %x09.cc, ...
6	add %x01, %x03 → %x11
7	imul %x04, %x06 → %x12
8	imul %x03, %x08 → %x13
9	cmp %x11, %x13 → %x14.cc
10	jle %x14.cc, ...

... ...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	pending ready
%x04	ready
%x05	ready
%x06	pending ready
%x07	ready
%x08	pending ready
%x09	pending ready
%x10	pending
%x11	pending ready
%x12	pending ready
%x13	pending ready
%x14	pending ready
...	...

execution unit	cycle#	1	2	3	4	5	6	7	...
ALU 1 (add, cmp, jxx)		1	6	—	4	5	9	10	
ALU 2 (add, cmp, jxx)		—	—	—	—	—	—	—	
ALU 3 (mul) start		2	3	7	8	—			
ALU 3 (mul) end			2	3	7	8			

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: approx 16 cycles (v. 2 for 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: approx 16 cycles (v. 2 for pipelined CPU)

multiple cores+threads

core 1:



core 2:



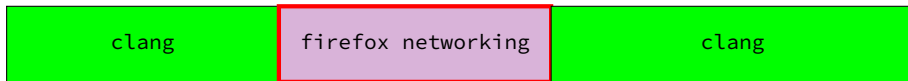
multiple cores? each core still divided up

multiple cores+threads

core 1:

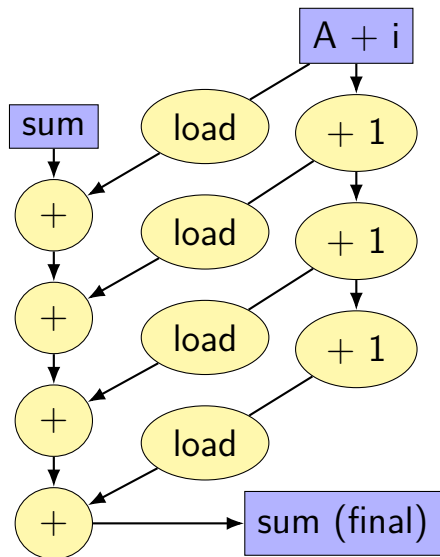


core 2:



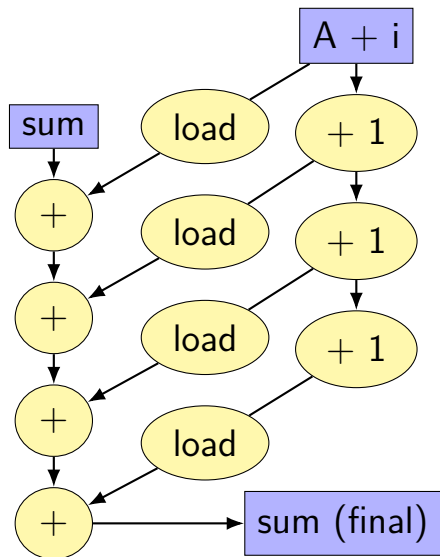
one program with multiple *threads*

data flow model and limits



```
for (int i = 0; i < N; i += K) {  
    sum += A[i];  
    sum += A[i+1];  
    ...  
}
```

data flow model and limits

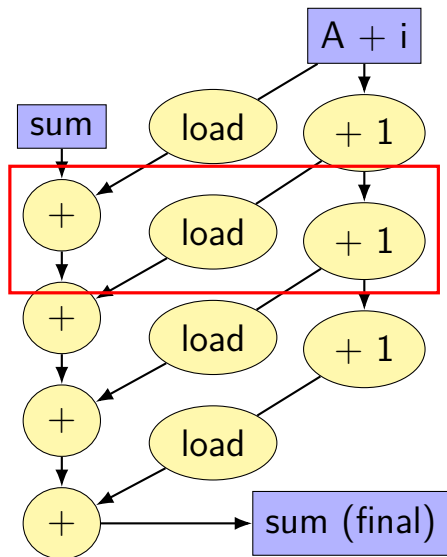


each yellow box = instruction

arrows = dependencies

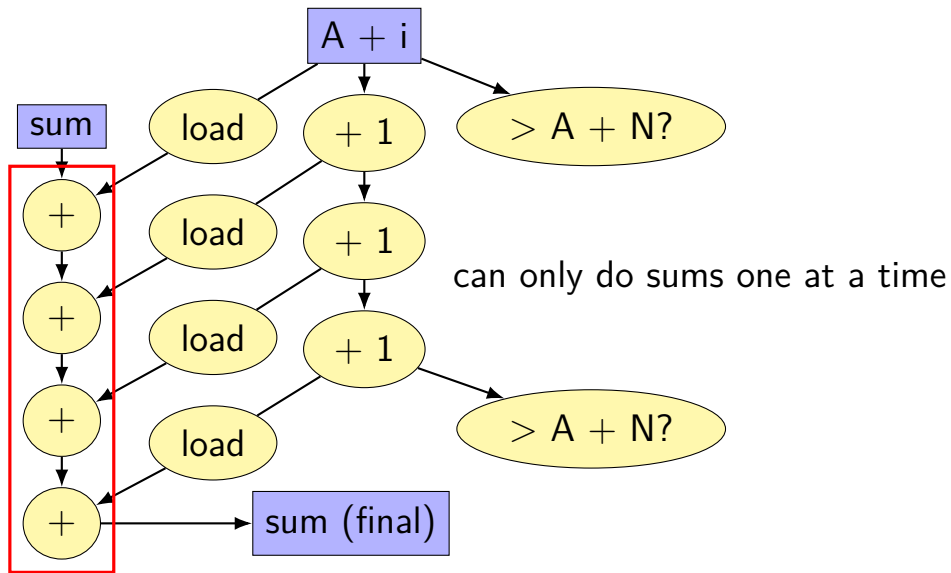
instructions only executed when dependencies

data flow model and limits



three ops/cycle (if each one cycle)

data flow model and limits



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
```

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

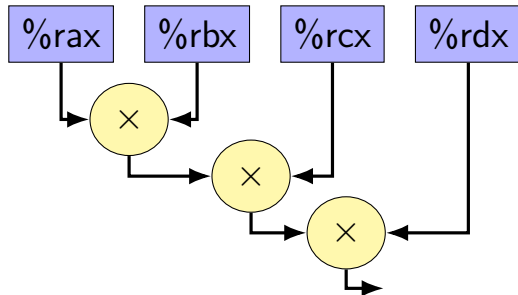
```
imulq %rbx, %rax  
imulq %rcx, %rdx  
imulq %rdx, %rax
```

reassociation

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

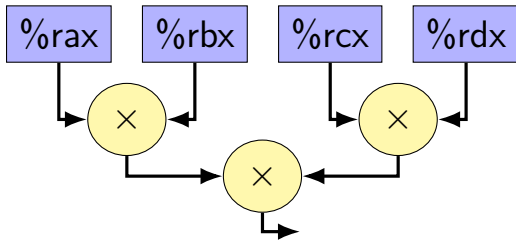
$$((a \times b) \times c) \times d$$

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

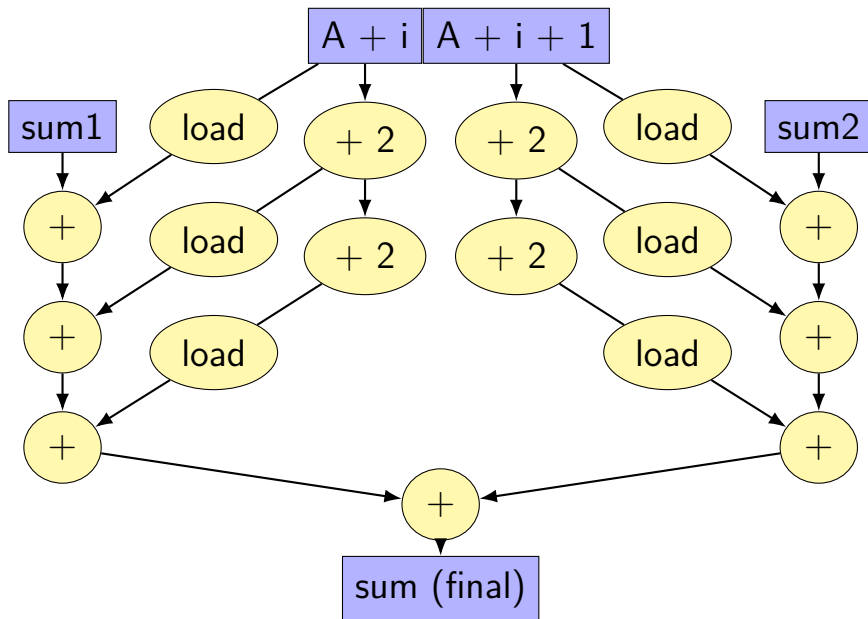


$$(a \times b) \times (c \times d)$$

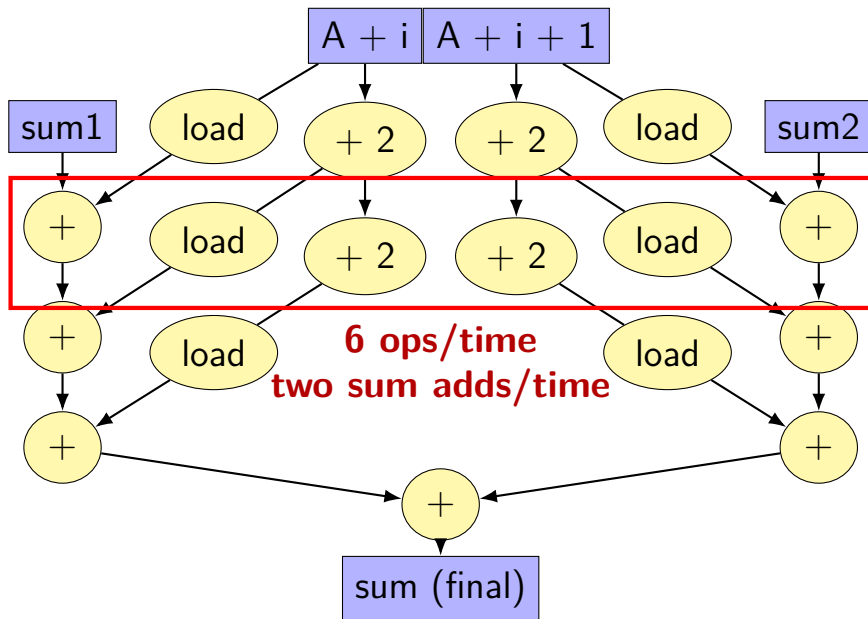
```
imulq %rbx, %rax  
imulq %rcx, %rdx  
imulq %rdx, %rax
```



better data-flow



better data-flow



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

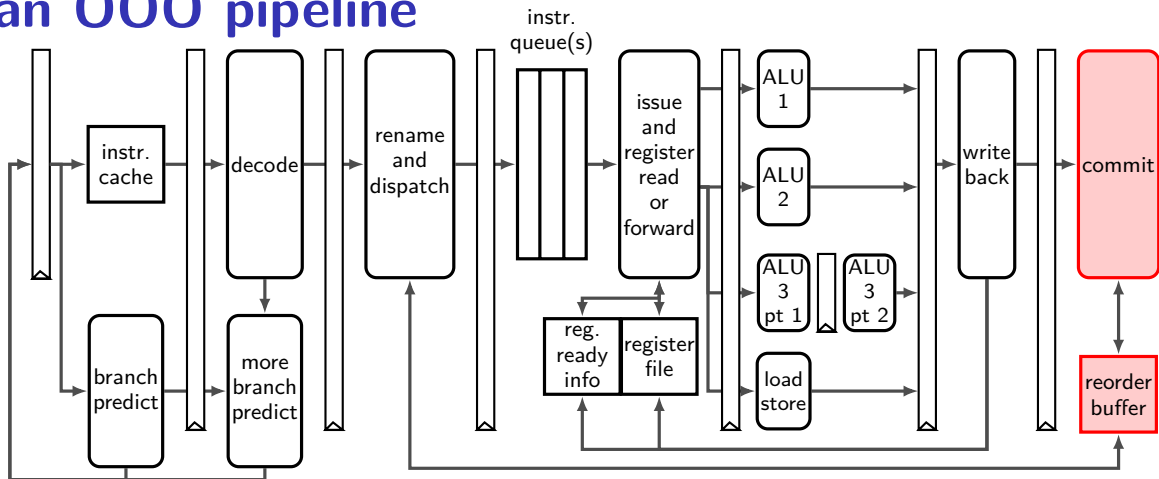
but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer

determines how far ahead branch mispredictions, etc. can happen

an OOO pipeline



reorder buffer: on rename

phys \rightarrow arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
...	...

free list

%x19
%x23
...
...

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
...	...

free list

%x19
%x23
...
...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		

reorder buffer contains instructions started,
but not fully finished new entries created on rename
(not enough space? stall rename stage)

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
...	...

free list

%x19
%x23
...
...

reorder buffer (ROB)

remove
here →
on commit

add here →
on rename

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		

place newly started instruction at end of buffer
remember at least its destination register
(both architectural and physical versions)

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

free list

%x19
%x23
...
...

remove
here
on commit



reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

add here
on rename



next renamed instruction goes in next slot, etc.

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

free list

%x19
%x23
...
...

reorder buffer (ROB)

remove
here
on commit



instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

add here
on rename



reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

free list

%x19
%x13
...
...

remove
here →
on commit

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

free list

%x19
%x13
...
...

remove
here →
on commit

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		✓

instructions marked done in reorder buffer when computed but not removed ('committed') yet

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	%x23
%rdx	%x21
...	...

free list

%x19
%x13
...
...

remove
here →
on commit

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		✓

commit stage tracks architectural to physical register map
for committed instructions

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	%x23 %x24
%rdx	%x21
...	...

free list

%x19
%x13
...
%x23

remove
here →
on commit

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

when next-to-commit instruction is done
update this register map and free register list
and remove instr. from reorder buffer

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07 %x19
...	...

phys → arch. reg remove here
for committed when committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	%x23 %x24
%rdx	%x21
...	...

free list

%x19
%x13
...
%x23

when next-to-commit instruction is done
update this register map and free register list
and remove instr. from reorder buffer

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
...	...

free list

%x19
%x13
...
...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30 %x38
%rcx	%x31 %x32
%rbx	%x23 %x24
%rdx	%x21 %x34
...	...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30	✓	
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32	✓	
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
→ 20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
...	...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30 %x38
%rcx	%x31 %x32
%rbx	%x23 %x24
%rdx	%x21 %x34
...	...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30	✓	
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32	✓	
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
→ 20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

free list

%x19
%x13
...
...

when committing a mispredicted instruction...
this is where we undo mispredicted instructions

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
...	...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30 %x38
%rcx	%x31 %x32
%rbx	%x23 %x24
%rdx	%x21 %x34
...	...

free list

%x19
%x13
...
...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30	✓	
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32	✓	
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

copy commit register map into rename register map
so we can start fetching from the correct PC

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
...	...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30 %x38
%rcx	%x31 %x32
%rbx	%x23 %x24
%rdx	%x21 %x34
...	...



free list

%x19
%x13
...
...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30	✓	
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32	✓	
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
...
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		



...and discard all the mispredicted instructions
(without committing them)

better? alternatives

- can take snapshots of register map on each branch

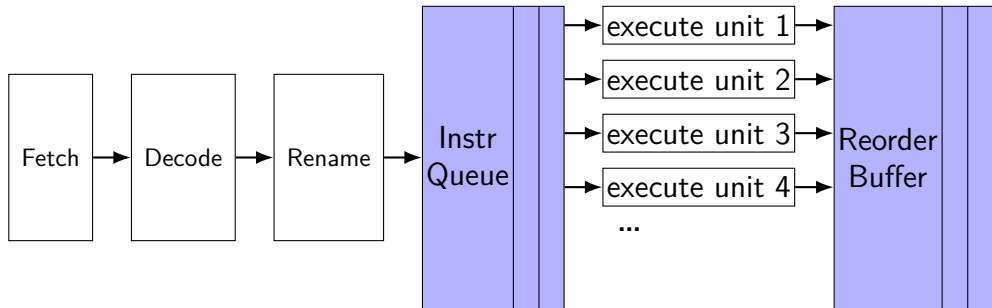
 - don't need to reconstruct the table
(but how to efficiently store them)

- can reconstruct register map before we commit the branch instruction

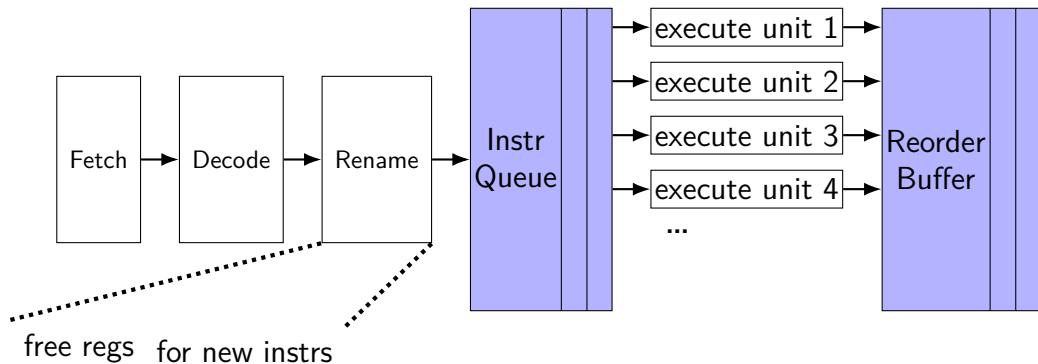
 - need to let reorder buffer be accessed even more?

- can track more/different information in reorder buffer

exceptions and OOO (one strategy)



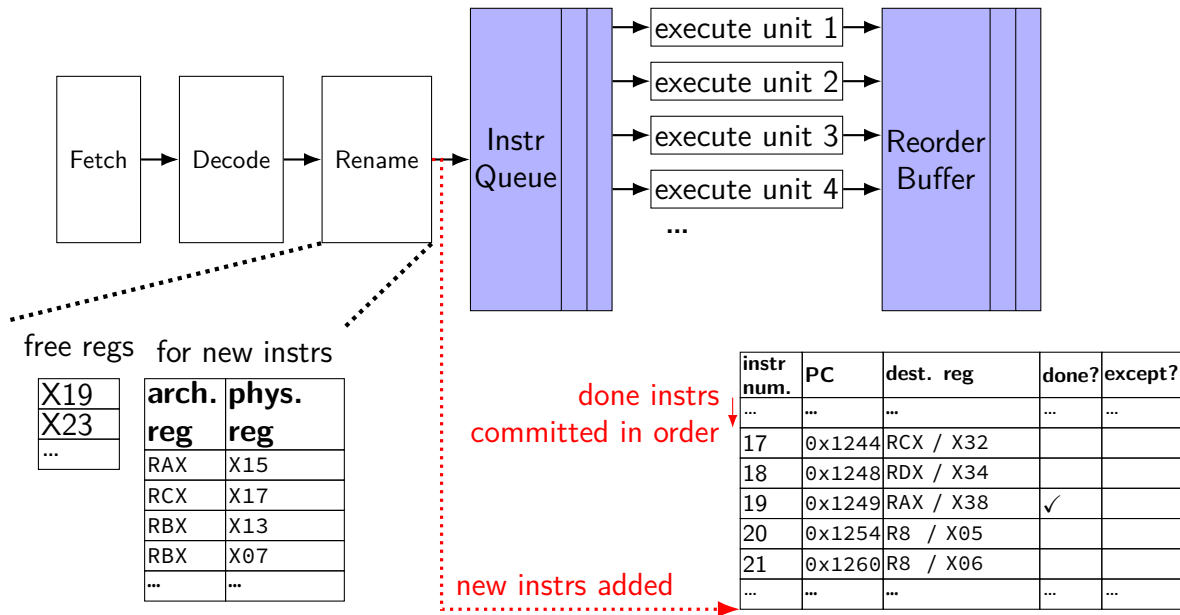
exceptions and OOO (one strategy)



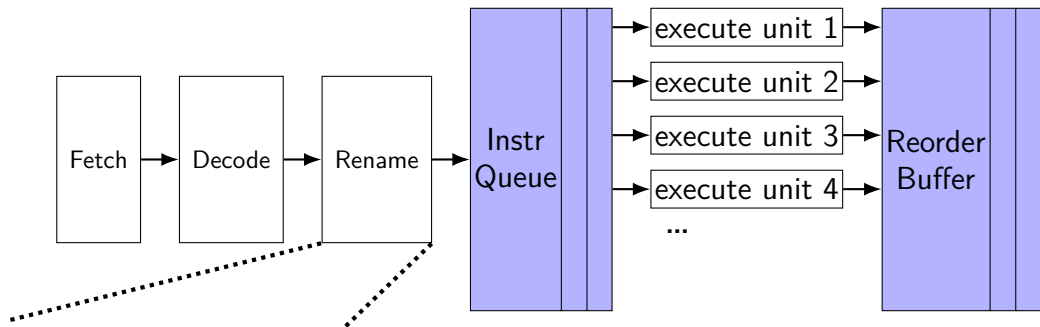
X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



free regs for new instrs for complete instrs

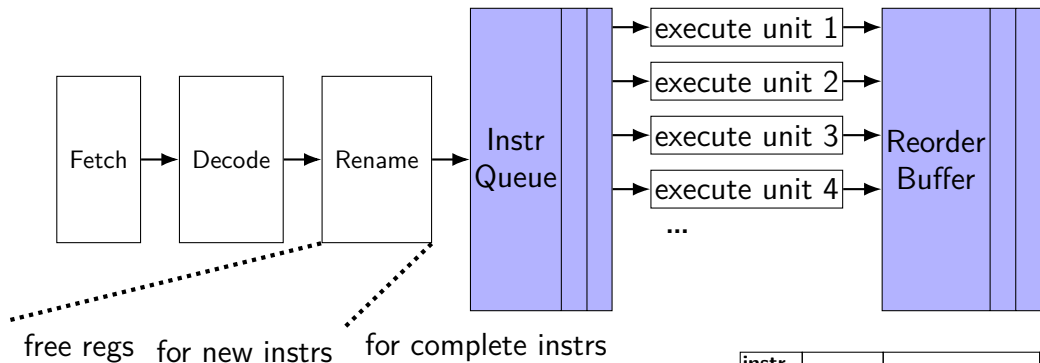
X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

arch. reg	phys. reg
RAX	X21
RCX	X2 X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



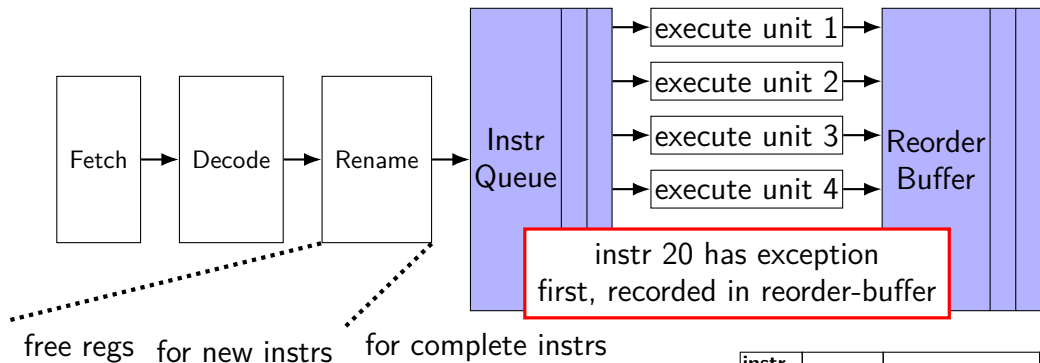
X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

arch. reg	phys. reg
RAX	X21
RCX	X2 X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



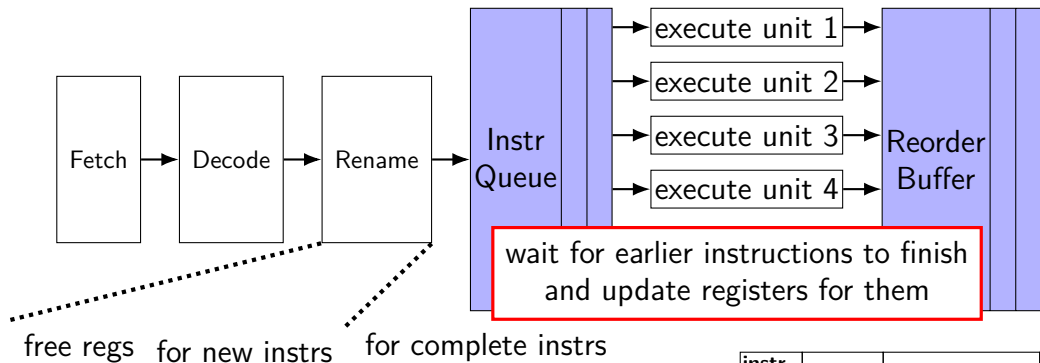
X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

arch. reg	phys. reg
RAX	X21
RCX	X2 X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



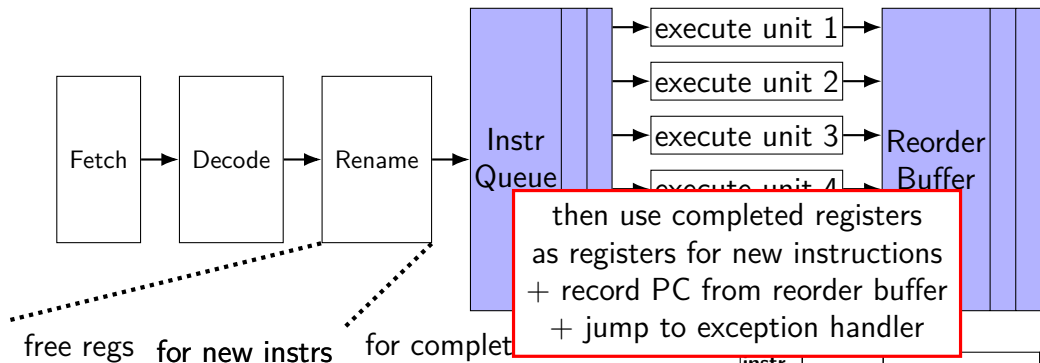
X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

arch. reg	phys. reg
RAX	X21 X38
RCX	X2 X32
RBX	X48
RDX	X37 X34
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



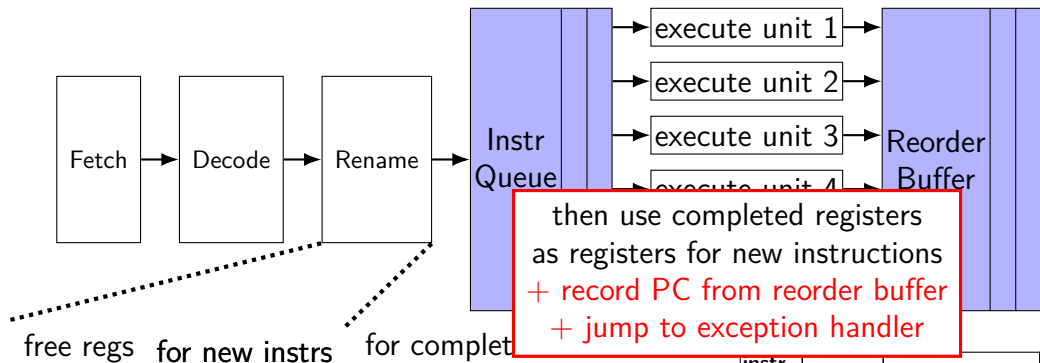
X19
X23
...

arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
...	...

arch. reg	phys. reg
RAX	X21 X38
RCX	X2 X32
RBX	X48
RDX	X37 X34
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



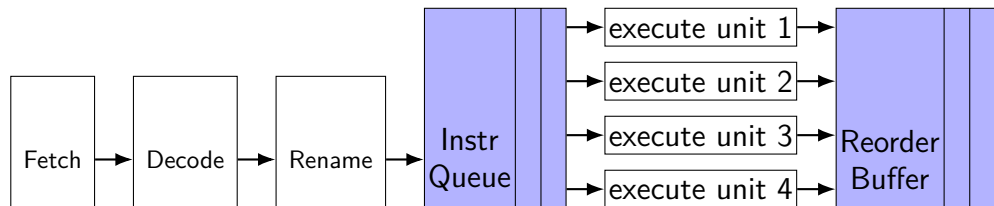
X19
X23
...

arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
...	...

arch. reg	phys. reg
RAX	X21 X38
RCX	X2 X32
RBX	X48
RDX	X37 X34
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



variation: could store architectural reg. values instead of mapping for completed instrs. (and copy values instead of mapping on exception)

free regs for new instrs for complete instrs

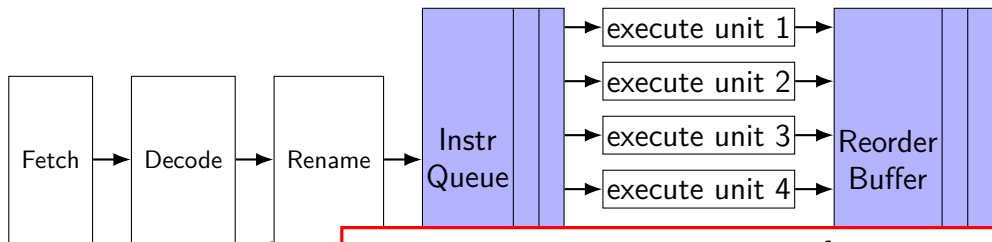
X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

arch. reg	value
RAX	0x12343
RCX	0x234543
RBX	0x56782
RDX	0xF83A4
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...

exceptions and OOO (one strategy)



stopping instructions in progress for exception
similar to how 'squashing' mispredicted instructions

free regs for new instrs for complete instrs

X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

arch. reg	phys. reg
RAX	X21 X38
RCX	X2 X32
RBX	X48
RDX	X37 X34
...	...

instr num.	PC	dest. reg	done?	except?
...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...

handling memory accesses?

one idea:

list of done + uncommitted loads+stores

execute load early + double-check on commit

- have data cache watch for changes to addresses on list
- if changed, treat like branch misprediction

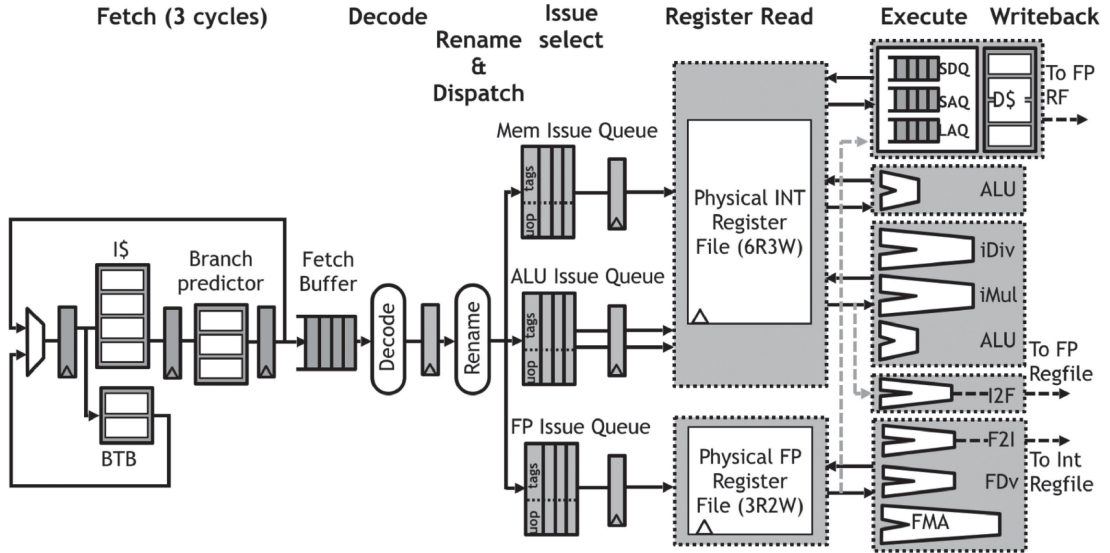
loads check list of stores so you read back own values

actually finish store on commit

- maybe treat like branch misprediction if conflict?

backup slides

the open-source BROOM pipeline



backup slides

exercise: forwarding paths (2)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

subq %r8, %r9

ret (goes to andq)

andq %r10, %r9

in subq, %r8 is _____ addq.

in subq, %r9 is _____ addq.

in andq, %r9 is _____ subq.

in andq, %r9 is _____ addq.

A: not forwarded from

B-D: forwarded to decode from {execute,memory,writeback} stage of