Changelog

2022-11-03: instruction queue + dispatch: fix inconsistencies between registers in first + second iteration of loop

last time (1)

"cache blocking"

- choose subsets of data *fit in cache* to work on
- subset often = part ('block') of a matrix
- reorder operations to iterate over subsets
- within subset, good spatial+temporal locality

```
for each i, j, ...:
becomes
for each starting I, J,...:
    for each i, j in (I, I+size), (J, J+size),...:
```

last time(2)

- loop unrolling
  - make loop body do two or more iterations of work
  - adjust 'bookkeeping' code to account the change
    - increment by two or more instead of one
  - extra code for 'left-over'
  - faster because of less bookkeeping code
- out-of-order and register versions
  - can have multiple active versions of register value
  - can't figure out which is correct based on stage
  - solution: assign version numbers to each one
  - + preprocess instructions to add version numbers

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

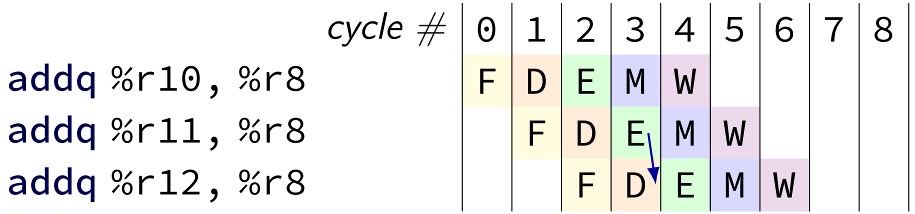
out-of-order execution makes hazards harder to handle

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*

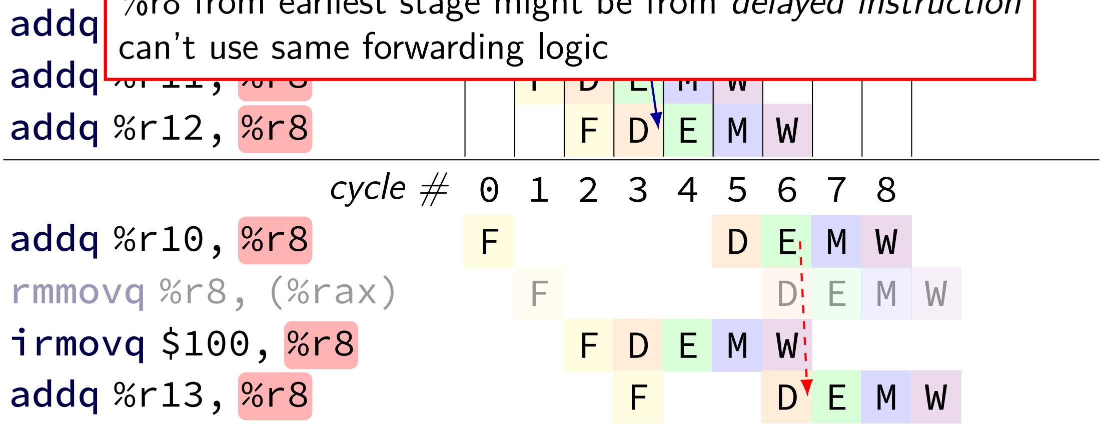
because it's from the most recent instruction

read-after-write examples (1)

out-of-order execution:

%r8 from earliest stage might be from *delayed instruction*

can't use same forwarding logic



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,` $\%r8_{v1} \rightarrow \%r8_{v2}$
`addq %r11, %r8`	`addq %r11,` $\%r8_{v2} \rightarrow \%r8_{v3}$
`addq %r12, %r8`	`addq %r12,` $\%r8_{v3} \rightarrow \%r8_{v4}$

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

cycle #	0	1	2	3	4	5	6	7	8				
`addq %r10, %r8`	F					D	E	M	$\boxed{W}$				
`rmmovq %r8, (%rax)`	F									D	E	M	W
`rrmovq %r11, %r8`		F	D	E	M	W							
`rmmovq %r8, 8(%rax)`		F						D	E	M	W		
`irmovq $100, %r8`			F	D	E	M	$\boxed{W}$						
`addq %r13, %r8`			F							D	E	M	W

write-after-write example

cycle #	0	1	2	3	4	5	6	7	8				
`addq %r10, %r8`	F					D	E	M	W				
`rmmovq %r8, (%rax)`	F									D	E	M	W
`rrmovq %r11, %r8`		F	D	E	M	W							
`rmmovq %r8, 8(%rax)`		F						D	E	M	W		
`irmovq $100, %r8`			F	D	E	M	W						
`addq %r13, %r8`			F							D	E	M	W

out-of-order execution:
if we don't do something, newest value could be overwritten!

write-after-write example

cycle #	0	1	2	3	4	5	6	7	8				
`addq %r10, %r8`	F					D	E	M	$\boxed{\text{W}}$				
`rmmovq %r8, (%rax)`	F									D	E	M	W
`rrmovq %r11, %r8`		F	D	E	M	W							
`rmmovq %r8, 8(%rax)`		F						D	E	M	W		
`irmovq $100, %r8`			F	D	E	M	$\boxed{\text{W}}$						
`addq %r13, %r8`			F							D	E	M	W

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

write-after-write example

cycle #	0	1	2	3	4	5	6	7	8				
`addq %r10, %r8`	F					D	E	M	$\boxed{W}$				
`rmmovq %r8, (%rax)`	F									D	E	M	W
`rrmovq %r11, %r8`		F	D	E	M	W							
`rmmovq %r8, 8(%rax)`		F						D	E	M	W		
`irmovq $100, %r8`			F	D	E	M	$\boxed{W}$						
`addq %r13, %r8`			F							D	E	M	W

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

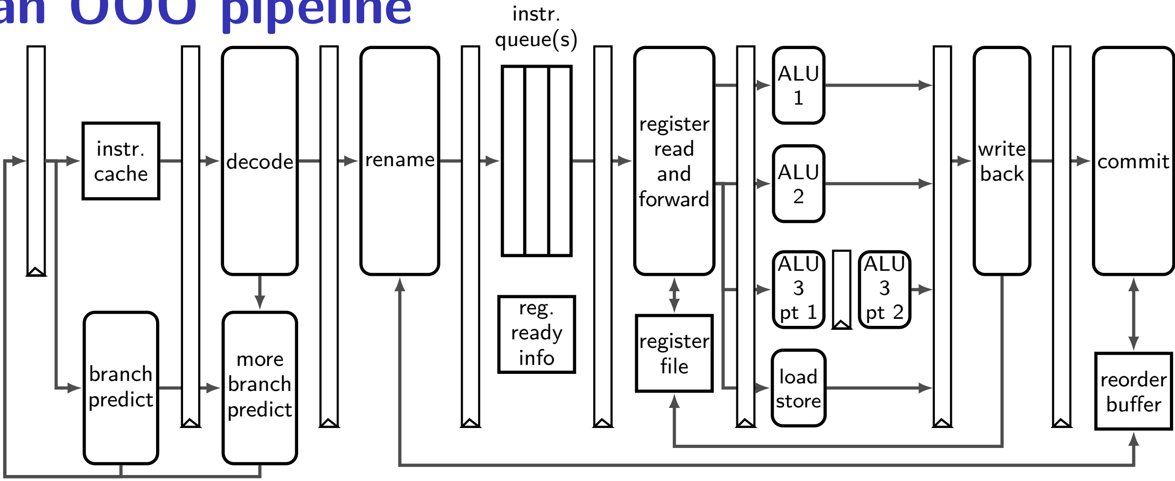
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

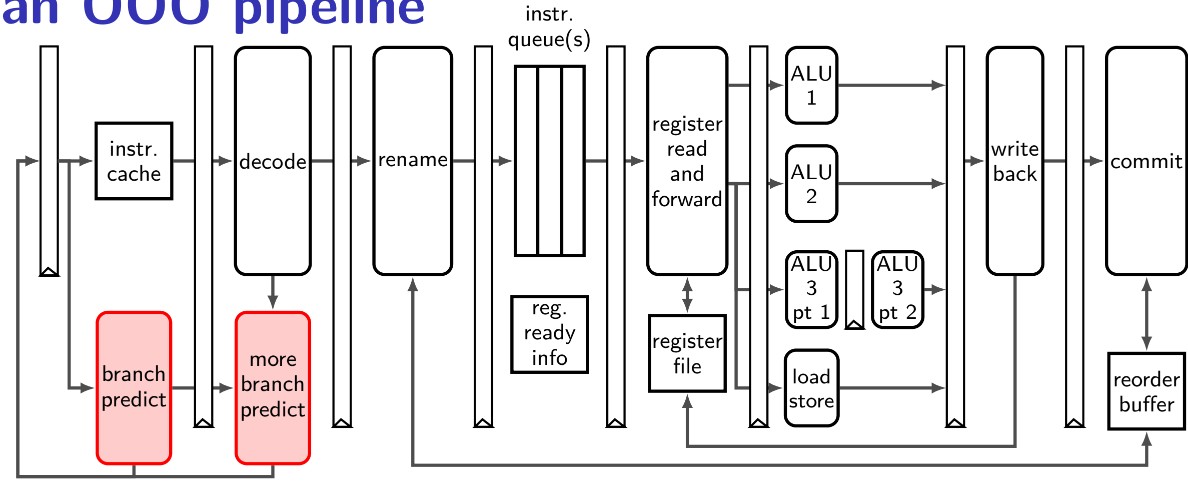
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

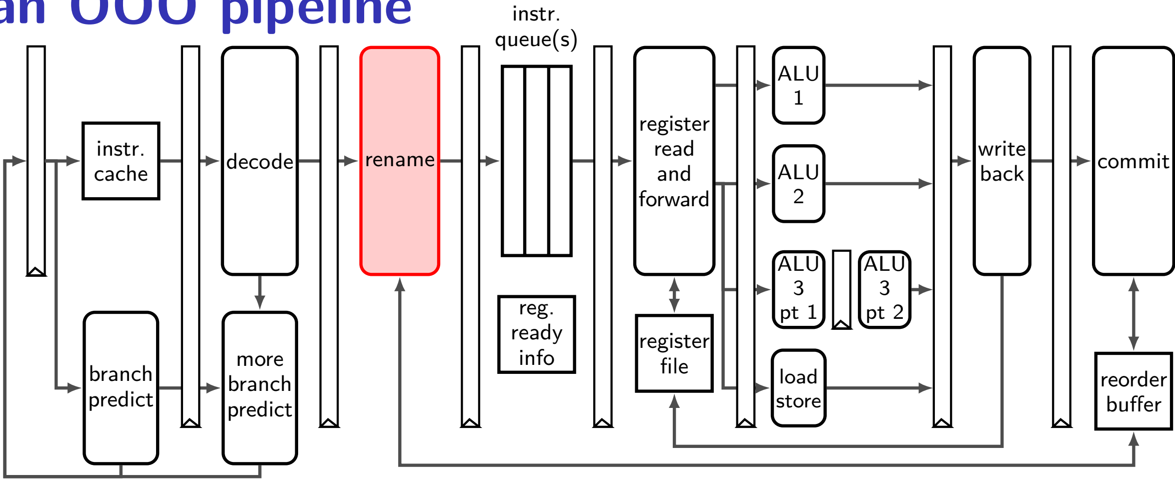


an OOO pipeline



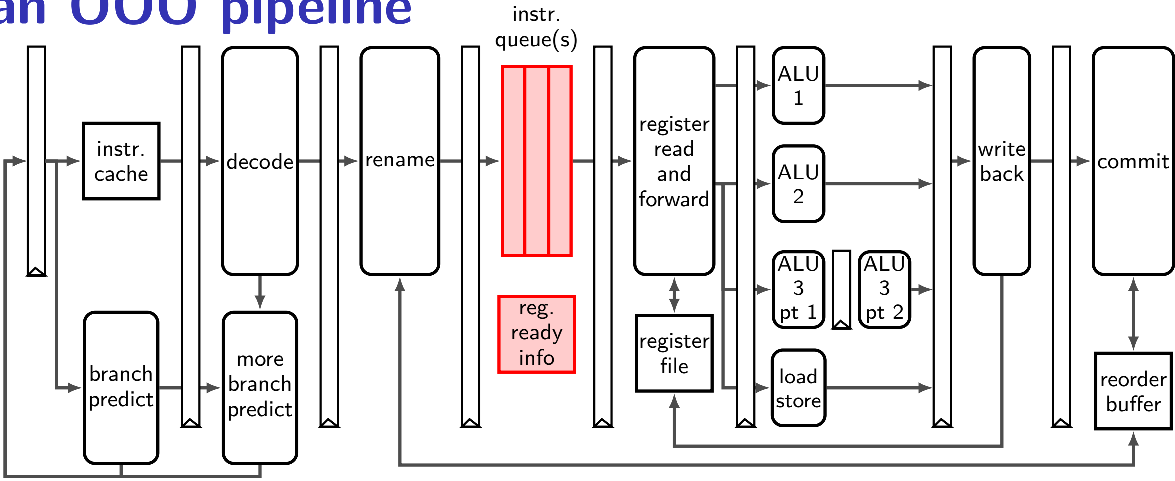
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



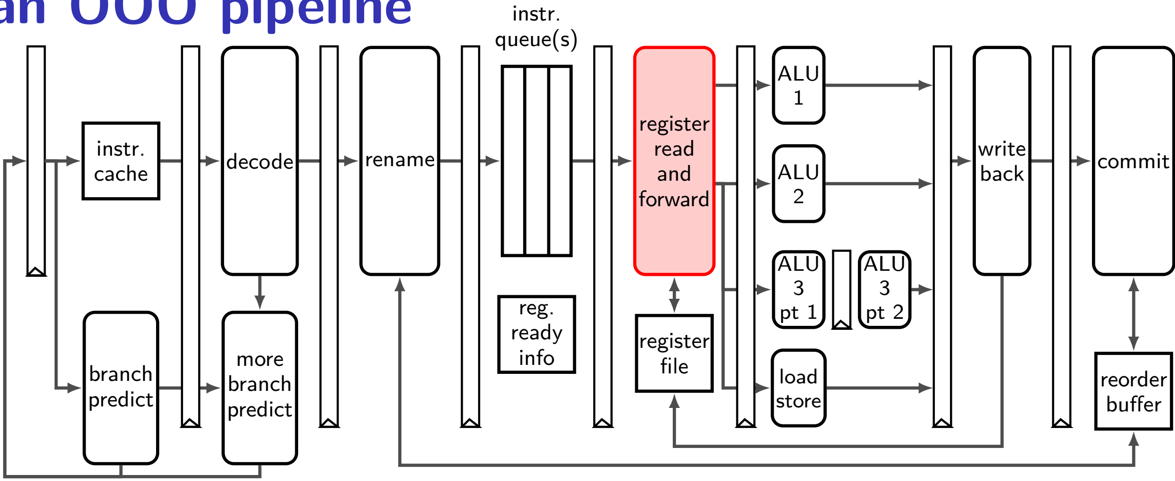
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



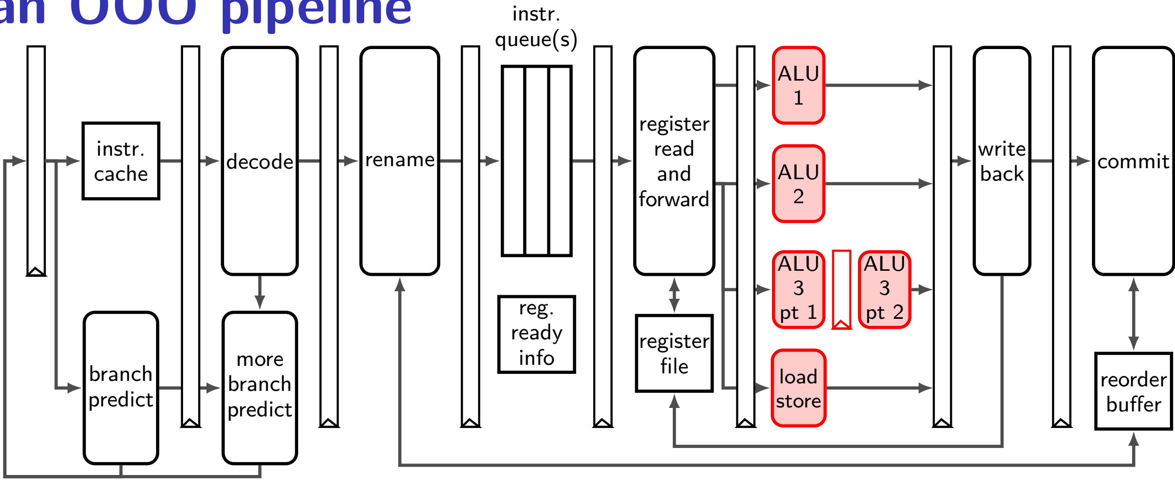
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



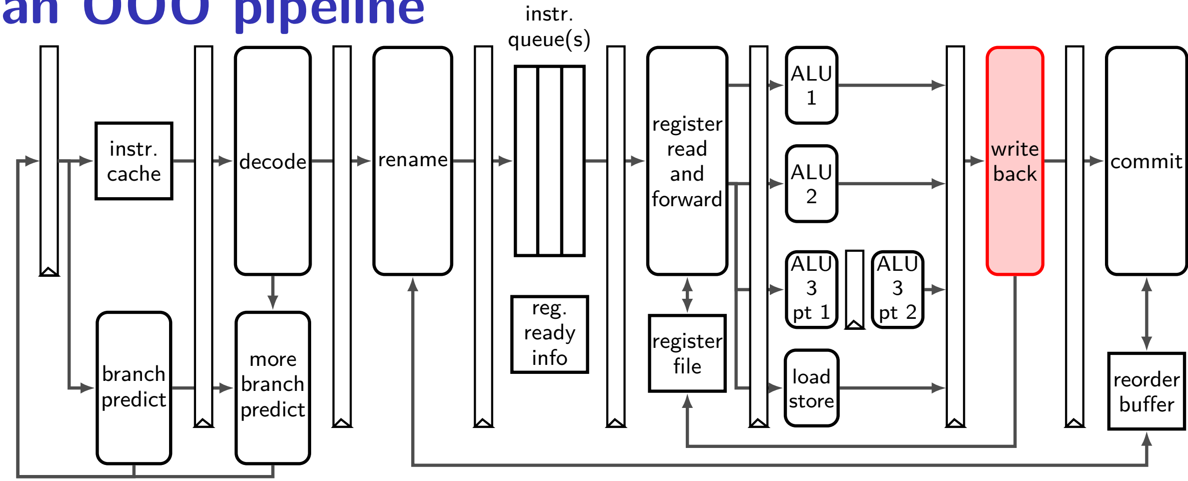
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



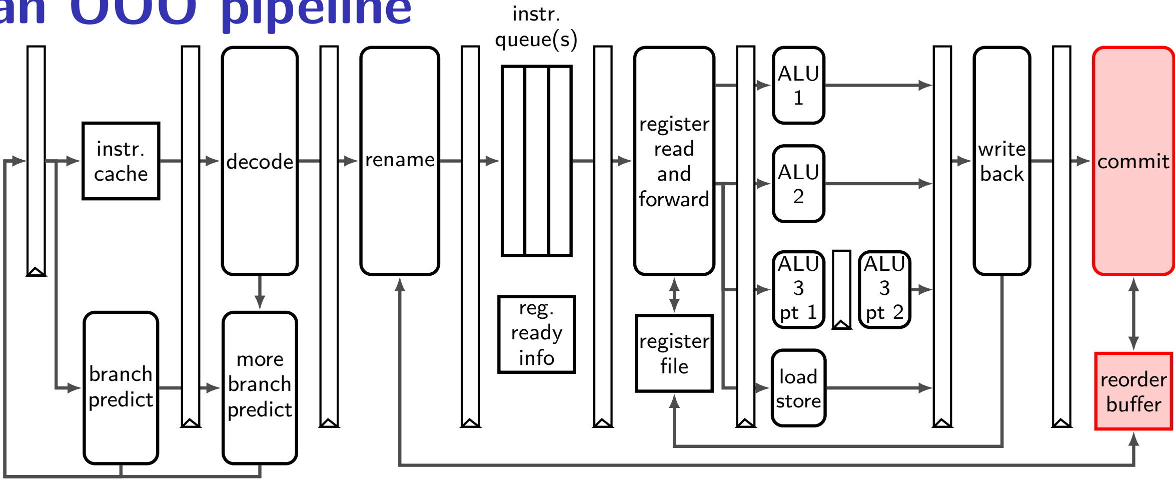
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



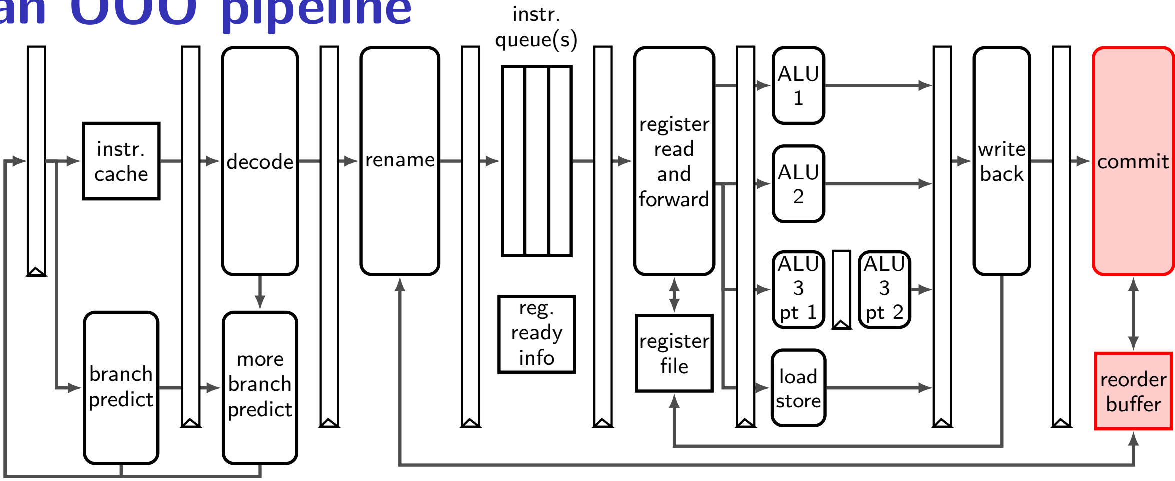
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



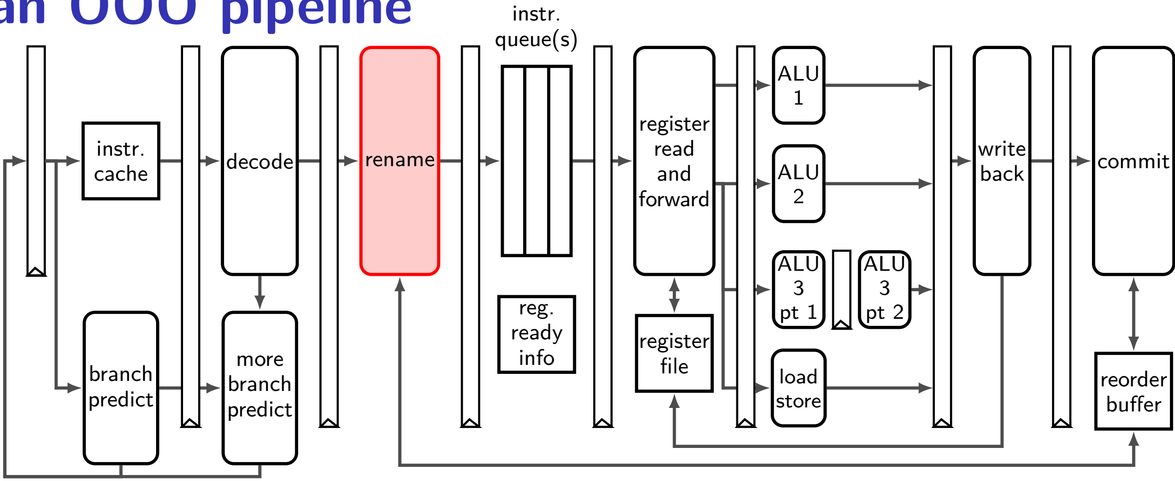
new commit (sometimes *retire*) stage finalizes instruction
figures out when physical registers can be reused again

an OOO pipeline



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

an OOO pipeline



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

```
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 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

$\boxed{\texttt{add \%x19, \%x13} \rightarrow \texttt{\%x18}}$

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

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

renamed

```
add %x19, %x13 → %x18
```

$\boxed{\texttt{add \%x07, \%x18} \to \texttt{\%x20}}$

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` $\rightarrow$ `%x18`
`add %r11, %r8`	`add %x07, %x18` $\rightarrow$ `%x20`
`add %r12, %r8`	$\boxed{\texttt{add \%x05, \%x20} \rightarrow \texttt{\%x21}}$

arch $\rightarrow$ 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

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

renamed

```
add %x19, %x13 → %x18
add %x07, %x18 → %x20
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

```
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

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 addresse

%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
`addq %r10, %r8`	`addq %x19, %x13` $\rightarrow$ `%x18`
`rmmovq %r8, (%rax)`	`rmmovq %x18, (%x04)` $\rightarrow$ (memory)
`subq %r8, %r11`	`subq %x18, %x07` $\rightarrow$ `%x20`
`mrmovq 8(%r11), %r11`	$\boxed{\texttt{mrmovq 8(\%x20), (memory)} \rightarrow \texttt{\%x21}}$
`irmovq $100, %r8`	
`addq %r11, %r8`	

arch $\rightarrow$ 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
`addq %r10, %r8`	`addq %x19, %x13` → `%x18`
`rmmovq %r8, (%rax)`	`rmmovq %x18, (%x04)` → (memory)
`subq %r8, %r11`	`subq %x18, %x07` → `%x20`
`mrmovq 8(%r11), %r11`	`mrmovq 8(%x20)`, (memory) → `%x21`
`irmovq $100, %r8`	$\boxed{\texttt{irmovq \$100} \rightarrow \texttt{\%x23}}$
`addq %r11, %r8`	

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
`addq %r10, %r8`	`addq %x19, %x13` $\rightarrow$ `%x18`
`rmmovq %r8, (%rax)`	`rmmovq %x18, (%x04)` $\rightarrow$ (memory)
`subq %r8, %r11`	`subq %x18, %x07` $\rightarrow$ `%x20`
`mrmovq 8(%r11), %r11`	`mrmovq 8(%x20),` (memory) $\rightarrow$ `%x21`
`irmovq $100, %r8`	`irmovq $100` $\rightarrow$ `%x23`
`addq %r11, %r8`	`addq %x21, %x23` $\rightarrow$ `%x24`

arch $\rightarrow$ 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
```

renamed

arch → phys

register map

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x21
%r12	%x05
%r13	%x02
…	…

free

regs

%x18
%x20
%x21
%x23
%x24
…

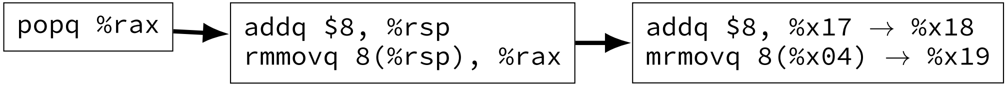
register renaming: missing pieces

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

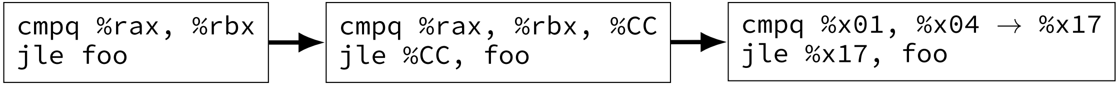
one solution: translate to intructions with additional register parameters

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

bonus: can also translate complex instructions to simpler ones

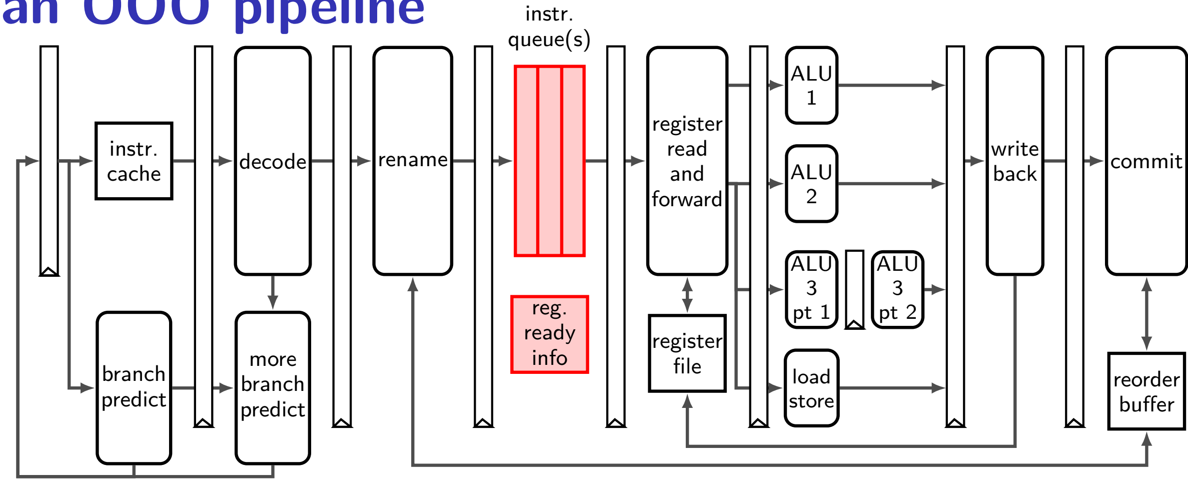


```
popq %rax
```
$\rightarrow$
```
addq $8, %rsp
rmmovq 8(%rsp), %rax
```
$\rightarrow$
```
addq $8, %x17 → %x18
mrmovq 8(%x04) → %x19
```



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

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, %x09 → %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	…
ALU 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, %x09` → `%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	`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, %x09 → %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	`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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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, %x09 → %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`
…	…

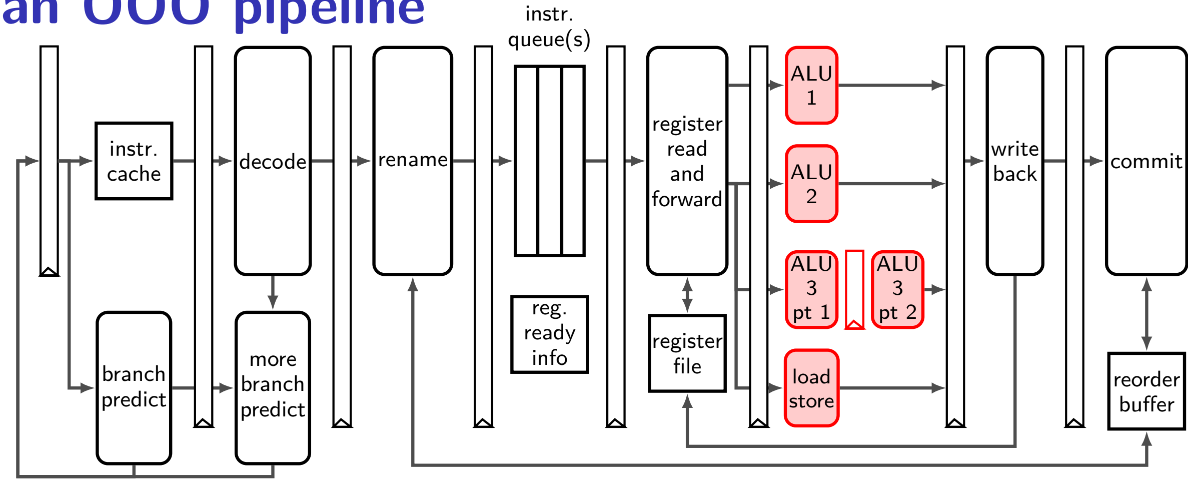
scoreboard

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

an OOO pipeline



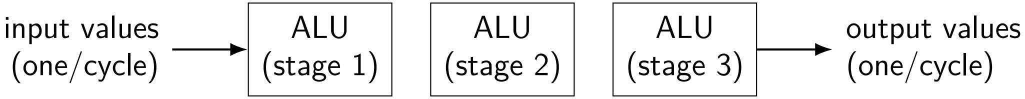
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



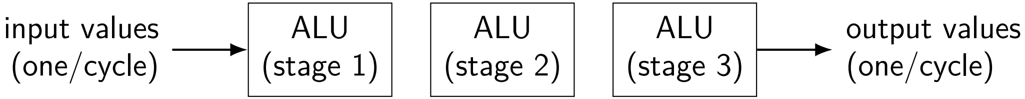
execution units AKA functional units (1)

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 $A \times (B \times (C \times D))$?

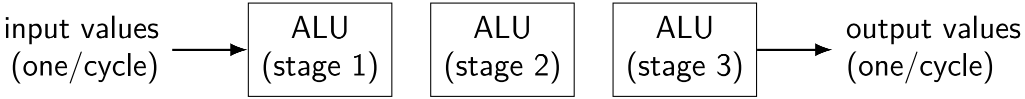
execution units AKA functional units (1)

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 $A \times (B \times (C \times D))$?

3×3 cycles $+$ any time to forward values

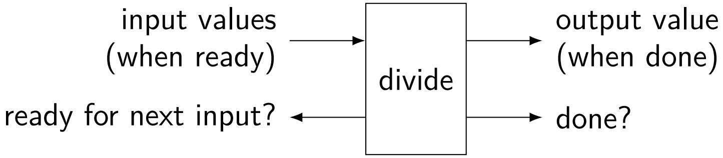
no parallelism!

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, ...`
…	…

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
…	…

…

execution unit
ALU 1 (add, cmp, jxx)
ALU 2 (add, cmp, jxx)
ALU 3 (mul) start
ALU 3 (mul) end

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
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
…	…

…

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

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
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
…	…

…

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

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 pipehw2 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

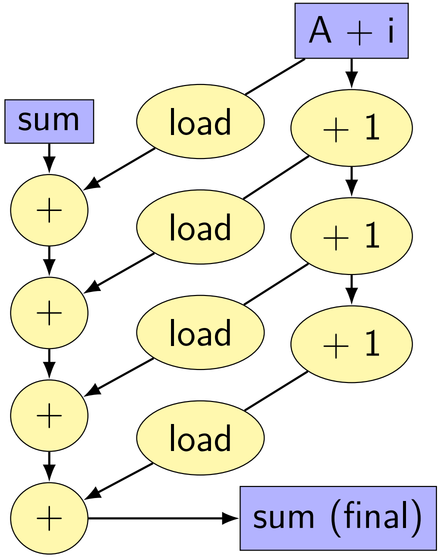
need to track all uncommitted instructions

- 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)

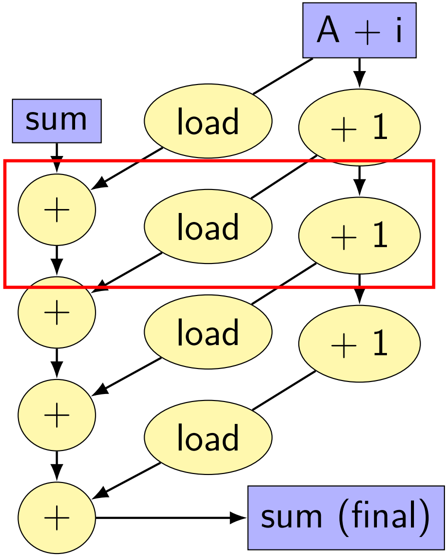
- e.g. Intel Skylake: approx 16 cycles (v. 2 for pipehw2 CPU)

data flow model and limits



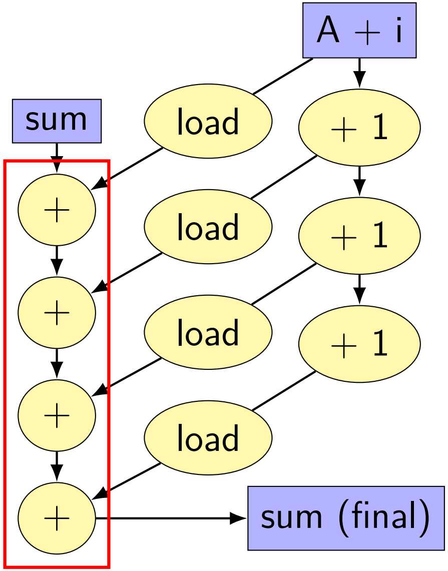
```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



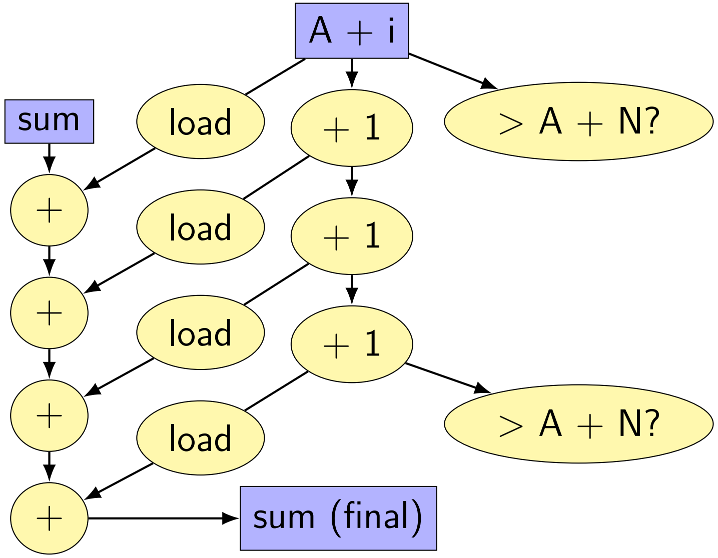
three ops/cycle (if each one cycle)

data flow model and limits



need to do additions
one-at-a-time
book's name: critical path
time needed: sum of latencies

data flow model and limits



reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding.
(hint: think about data-flow graph)

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

```
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

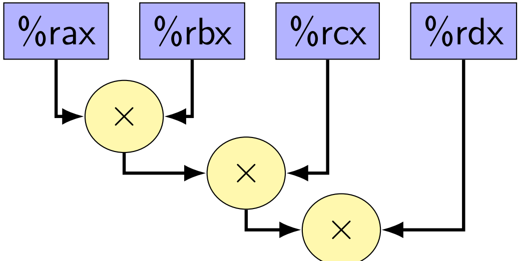
assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding.

(hint: think about data-flow graph)

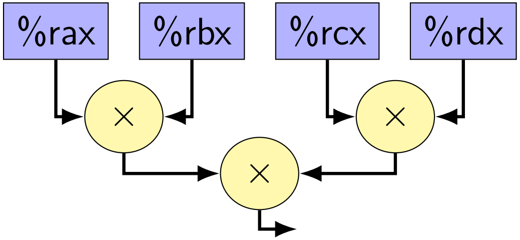
$((a \times b) \times c) \times d$

```
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
```



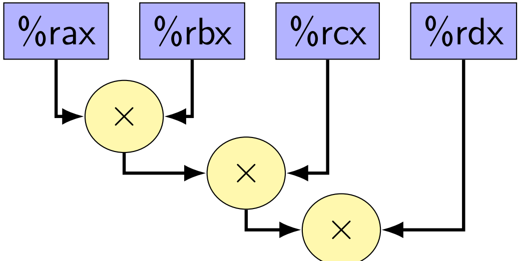
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding.
(hint: think about data-flow graph)

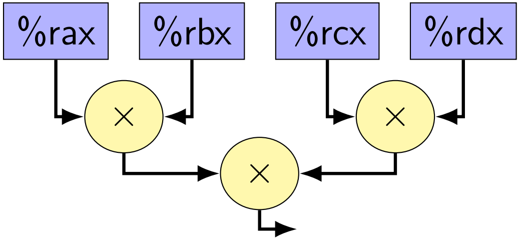
$((a \times b) \times c) \times d$

```
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

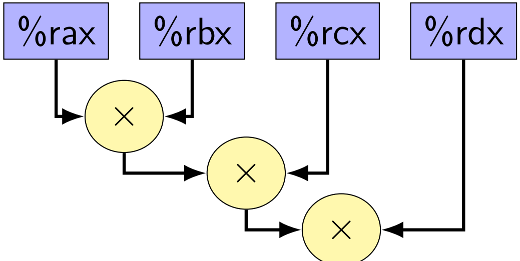
assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding.
(hint: think about data-flow graph)

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

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

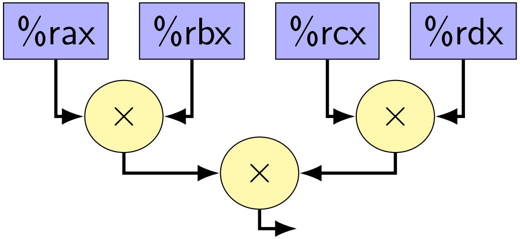
15 cycles



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

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

11 cycles



reassociation

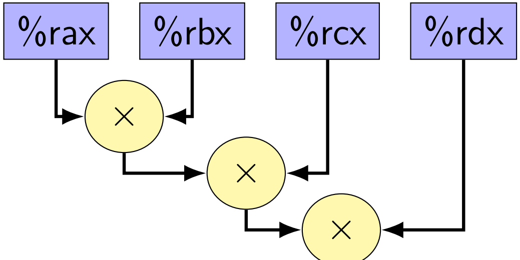
assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding.

(hint: think about data-flow graph)

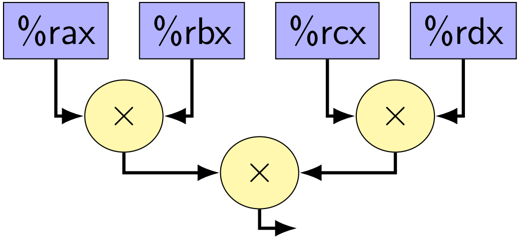
$((a \times b) \times c) \times d$

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

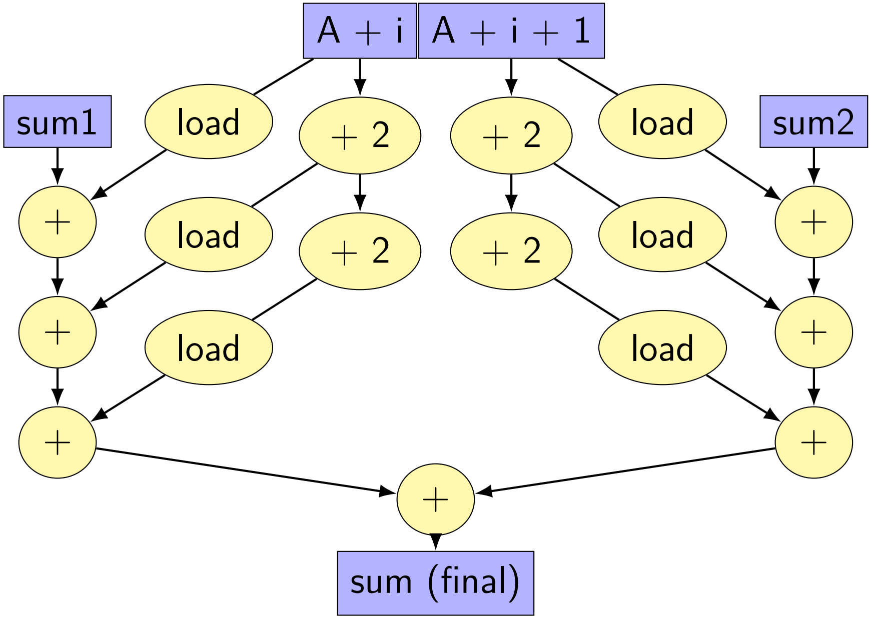


$(a \times b) \times (c \times d)$

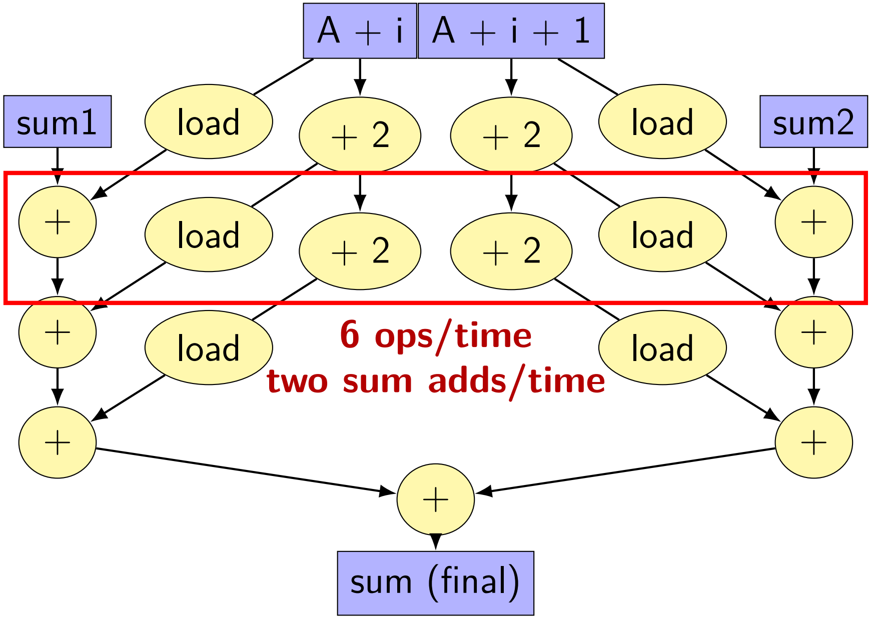
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



better data-flow

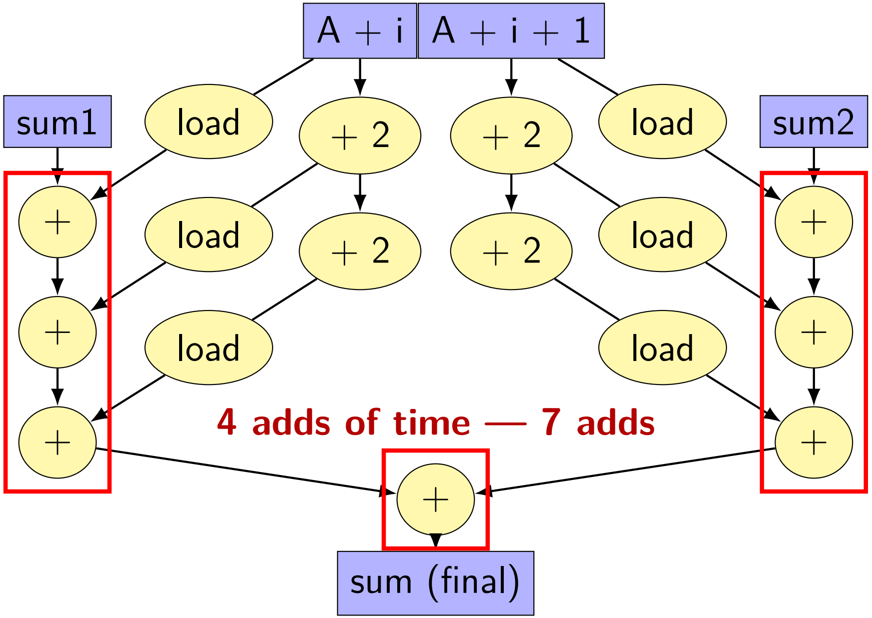


better data-flow



6 ops/time
two sum adds/time

better data-flow



multiple accumulators

```
int i;
long sum1 = 0, sum2 = 0;
for (i = 0; i + 1 < N; i += 2) {
    sum1 += A[i];
    sum2 += A[i+1];
}
// handle leftover, if needed
if (i < N)
    sum1 += A[i];
sum = sum1 + sum2;
```

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

8 accumulator assembly

```
sum1 += A[i + 0];
sum2 += A[i + 1];
...
...
```

```
addq     (%rdx), %rax          // sum1 +=
addq     8(%rdx), %rcx         // sum2 +=
subq     $-128, %rdx           // i +=
addq     -112(%rdx), %rbx      // sum3 +=
addq     -104(%rdx), %r11      // sum4 +=
...
....
cmpq   %r14, %rdx
```

register for each of the `sum1`, `sum2`, …variables:

16 accumulator assembly

compiler runs out of registers

starts to use the stack instead:

```
movq    32(%rdx), %rax  // get A[i+13]
addq    %rax, −48(%rsp) // add to sum13 on stack
```

code does extra cache accesses

also — already using all the adders available all the time

so performance increase not possible

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

maximum performance

2 additions per element:

- one to add to sum
- one to compute address (part of mov)

3/16 add/sub/cmp + 1/16 branch per element:

- over 16 because loop unrolled 16 times
- loop overhead
- compiler not as efficient as it could have been

2+3/16+1/16 = 2+1/4 instructions per element

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

$(2 + 1/4) \div 4 \approx 0.57$ cycles/element

getting over this limit

the $+1/4$ was from loop overhead

solution: more loop unrolling!

common theme with optimization:

fix one bottleneck (need to do adds one after the other)

find another bottleneck

backup slides

exercise: miss estimating (2)

```
for (int k = 0; k < 1000; k += 1)
    for (int i = 0; i < 1000; i += 1)
        for (int j = 0; j < 1000; j += 1)
            A[k*N+j] += B[i*N+j];
```

assuming: 4 elements per block

assuming: cache not close to big enough to hold 1K elements

estimate: *approximately* how many misses for A, B?

simple blocking – with 3?

```
for (int kk = 0; kk < N; kk += 3)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
      C[i*N+j] += A[i*N+kk+2] * B[(kk+2)*N+j];
    }
```

$\frac{N}{3} \cdot N$ j-loop iterations, and (assuming N large):

about 1 misses from A per j-loop iteration

  $N^2/3$ total misses (before blocking: N^2)

about $3N \div$ block size misses from B per j-loop iteration

  $N^3 \div$ block size total misses (same as before)

about $3N \div$ block size misses from C per j-loop iteration

  $N^3 \div$ block size total misses (same as before)

simple blocking – with 3?

```
for (int kk = 0; kk < N; kk += 3)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
      C[i*N+j] += A[i*N+kk+2] * B[(kk+2)*N+j];
    }
```

$\frac{N}{3} \cdot N$ j-loop iterations, and (assuming N large):

about 1 misses from A per j-loop iteration

  $N^2/3$ total misses (before blocking: N^2)

about $3N \div$ block size misses from B per j-loop iteration

  $N^3 \div$ block size total misses (same as before)

about $3N \div$ block size misses from C per j-loop iteration

  $N^3 \div$ block size total misses (same as before)

more than 3?

can we just keep doing this increase from 3 to some large X? ...

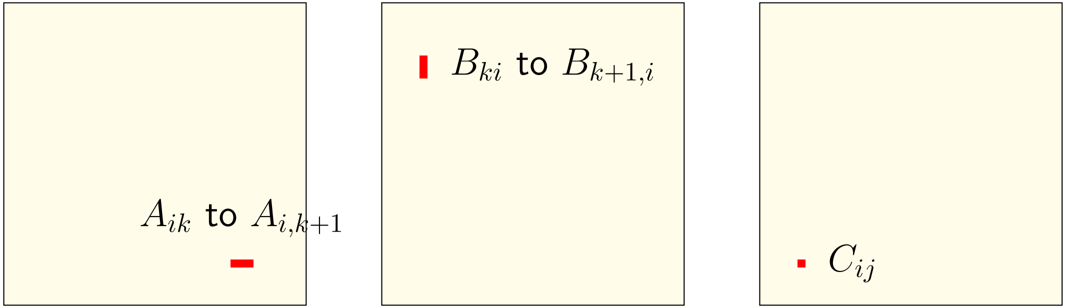
assumption: X values from A would stay in cache

- X too large — cache not big enough

assumption: X blocks from B would help with spatial locality

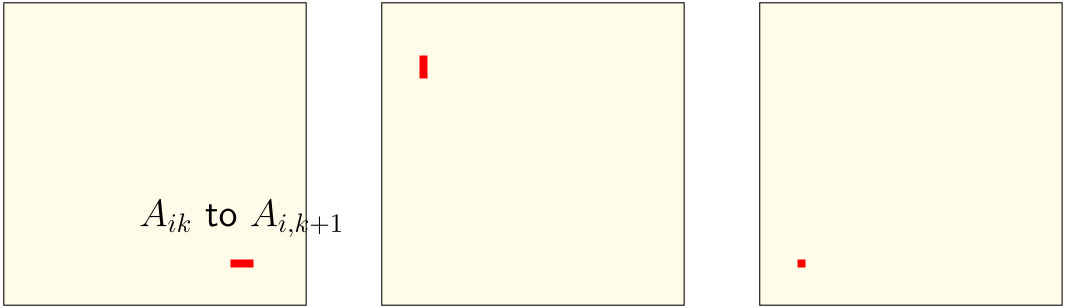
- X too large — evicted from cache before next iteration

array usage (2 k at a time)



```
for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot B_{kj}$
```

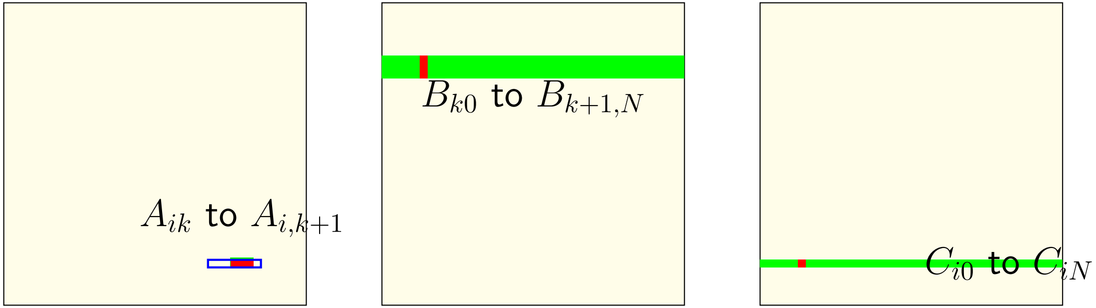
array usage (2 k at a time)



```
for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot B_{kj}$
```

within innermost loop
good spatial locality in A
bad locality in B
good temporal locality in C

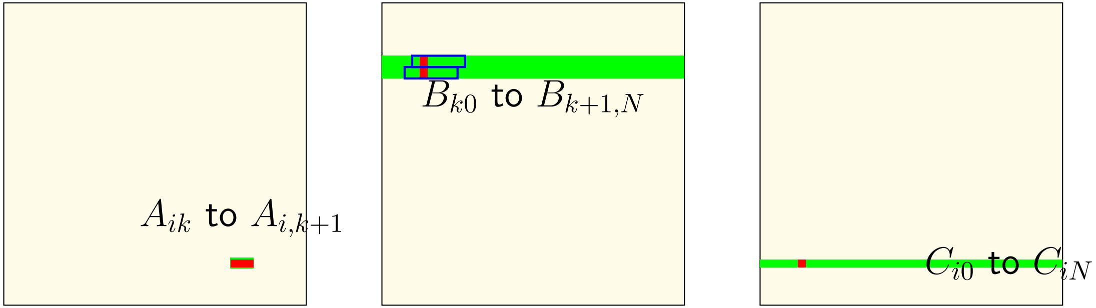
array usage (2 k at a time)



```
for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot B_{kj}$
```

loop over j: better spatial locality over A than before; still good temporal locality for A

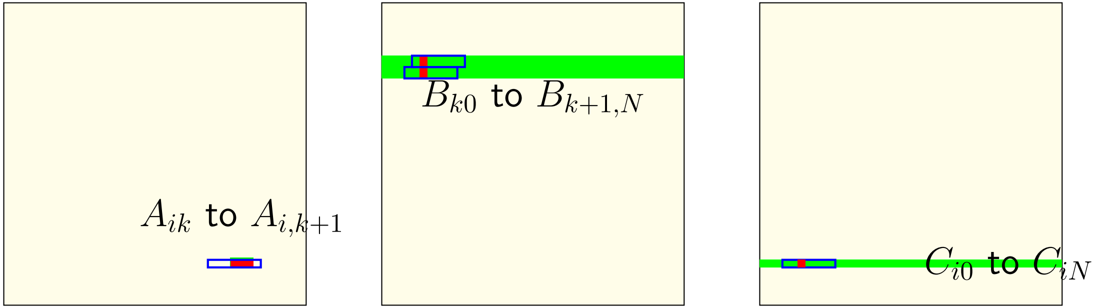
array usage (2 k at a time)



```
for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot B_{kj}$
```

loop over j: spatial locality over B is worse
but probably not more misses
cache needs to keep two cache blocks
for next iter instead of one
(probably has the space left over!)

array usage (2 k at a time)



```
for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot$
```

right now: only really care about
keeping 4 cache blocks in j loop

have more than 4 cache blocks?
increasing kk increment would use more of them

exercise

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        A[i*N+j] += B[i] + C[j]
```

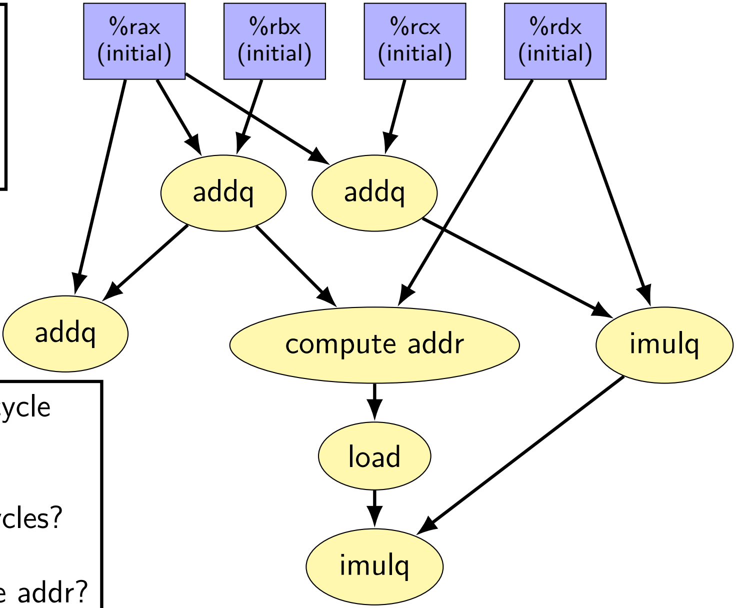
Which of the following suggests changing order of memory accesses?

```
/* version A */
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; j += 2) {
    A[i*N+j] += B[i] + C[j]
    A[i*N+j+1] += B[i] + C[j+1]
  }
```

```
/* version B */
for (int i = 0; i < N; i += 2)
  for (int j = 0; j < N; j += 2) {
    A[i*N+j] += B[i] + C[j];
    A[i*N+j+1] += B[i] + C[j+1];
    A[(i+1)*N+j] += B[i+1] + C[j];
    A[(i+1)*N+j+1] += B[i+1] + C[j+1];
  }
```

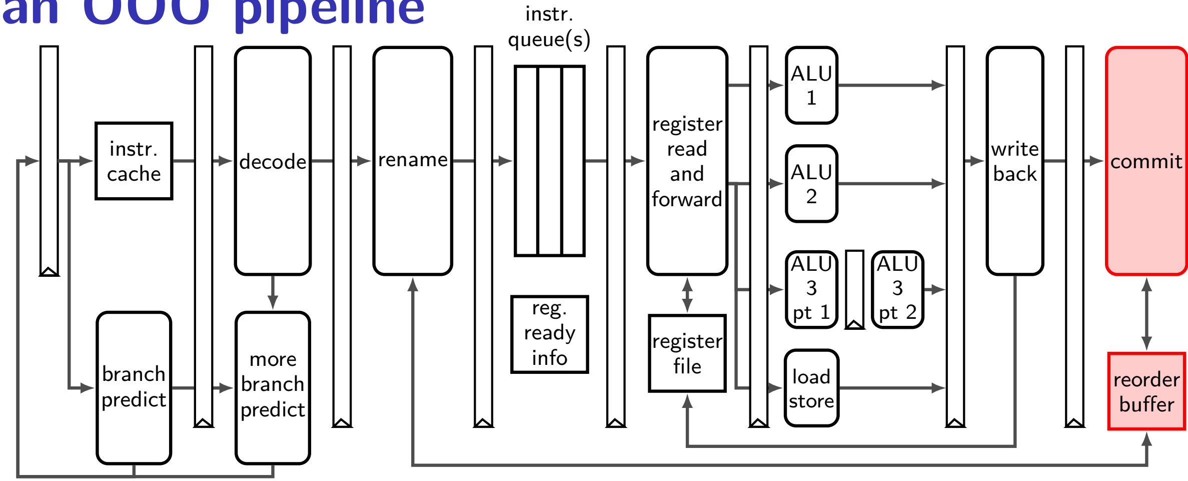
a data flow example

```
addq %rax, %rbx
addq %rax, %rcx
imulq %rdx, %rcx
movq (%rbx, %rdx), %r8
imulq %r8, %rcx
addq %rax, %rbx
```



addq, compute addr: 1 cycle
imulq: 3 cycle latency
load: 3 cycle latency
Q1: latency bound on cycles?
Q2: what can be done
at same time as compute addr?

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
…
…

remove here when committed →

add here on rename →

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		

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
…
…

reorder buffer (ROB)

remove here when committed

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
...
...

remove here
when committed →

add here
on rename →

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		

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
when committed →

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
when committed →

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 result is computed
but not removed from reorder buffer ('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
…	…

free list

~~%x19~~
%x13
…
…

phys → arch. reg
for committed

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

remove here
when committed

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
…	…

free list

~~%x19~~
%x13
…
%x23

phys → arch. reg
for committed

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

remove here
when committed

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
…	…

free list

~~%x19~~
%x13
…
%x23

phys → arch. reg
for committed

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

remove here
when committed

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: 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
…	…

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		

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
...	...

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		

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

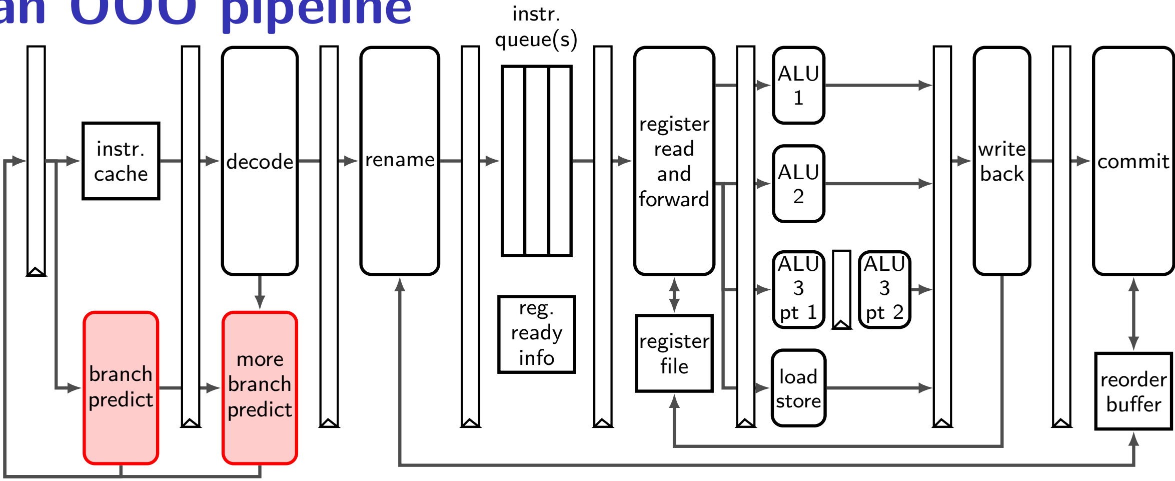
- don't need to reconstruct the table
- (but how to efficiently store them)

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

an OOO pipeline



branch target buffer

can take several cycles to fetch+decode jumps, calls, returns

still want 1-cycle prediction of next thing to fetch

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
…          …
```

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
…          …
```

aside on branch pred. and performance

modern branch predictors are very good

- we might explore how later in semester (if time)

…usually can assume most branches will be predicted

but could be a problem if really no pattern

- e.g. branch based on random number?

generally: measure and see

if branch prediction is bad...

avoiding branches — conditional move, etc.

replace multiple branches with single lookup?

- one misprediction better than K?

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 (stage 1)								
data cache (stage 2)								
data cache (stage 3)								