last time

single-cycle CPU review

- one possible CPU design that runs one instruction per cycle
- PC changes at beginning of cycle, cascades for other components to operation

pipelining idea

- laundry analogy
- opportunity: in single-cycle design, most components mostly idle
- assembly-line: step 1 of instr 1 then {step 2 of instr 1 + step 1 of instr 2}
- then {step 3 of instr 1 + step 2 of instr 2 + step 1 of instr 3} then ...
- adding registers to store values for each *stage*

pipelining limits

- time taken by new registers
- uneven split of stages

data hazards

some notes on the lab (1)

map/reduce division

I expected one loop

> map strategy determines which items you do log(...) operation for
> reduce strategy determines how you do += to answer

okay to two loops, but often likely slower than original code

some notes on the lab (2)

why was atomic update reduce strategy slow?

- processors need to take turns having accumulator (`answer`)
- lots of synchronization time + mostly one thread works at a time

why was task queue strategy slow?

- processors need to take turns grabbing index to use next
- lots of synchronization time

what about few-to-many reduction with array?

- `results[thread_id] +=`
- problem: multiple thread values are in *same cache block*
- cores need to take turns having the block in their cache to write
- workaround: make results array be more spread out
- called "false sharing"

addq processor: data hazard

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

addq processor: data hazard

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

data hazard

```
addq %r8, %r9  // (1)
addq %r9, %r8  // (2)
```

step#	pipeline implementation	ISA specification
1	read r8, r9 for (1)	read r8, r9 for (1)
2	read r9, r8 for (2)	write r9 for (1)
3	write r9 for (1)	read r9, r8 for (2)
4	write r8 for (2)	write r8 ror (2)

pipeline reads older value...

instead of value ISA says was just written

data hazard compiler solution

```
addq %r8, %r9
nop
nop
addq %r9, %r8
```

one solution: change the ISA

- all addqs take effect three instructions later
  (assuming can read register value while it is being written back)

make it compiler's job

problem: recompile everytime processor changes?

data hazard hardware solution

```
addq %r8, %r9
// hardware inserts: nop
// hardware inserts: nop
addq %r9, %r8
```

how about hardware add nops?

called stalling

extra logic:

- sometimes don't change PC
- sometimes put do-nothing values in pipeline registers

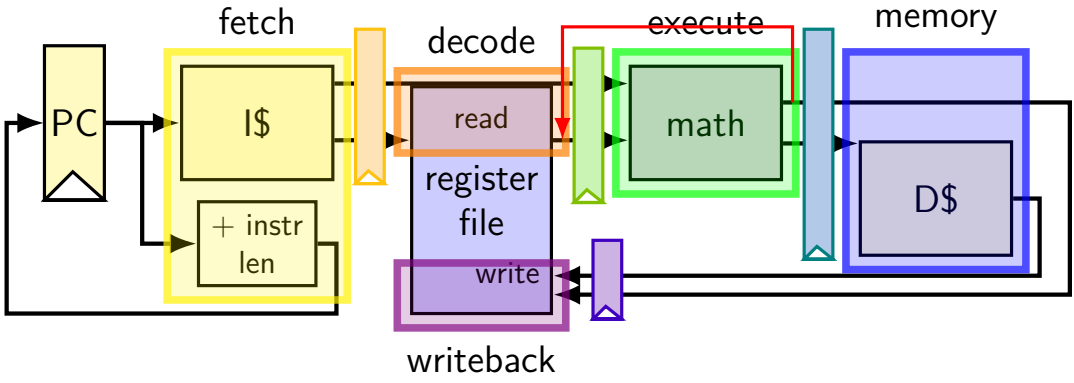
opportunity

```
// initially %r8 = 800,
//           %r9 = 900, etc.
0x0: addq %r8, %r9
0x2: addq %r9, %r8
...
```

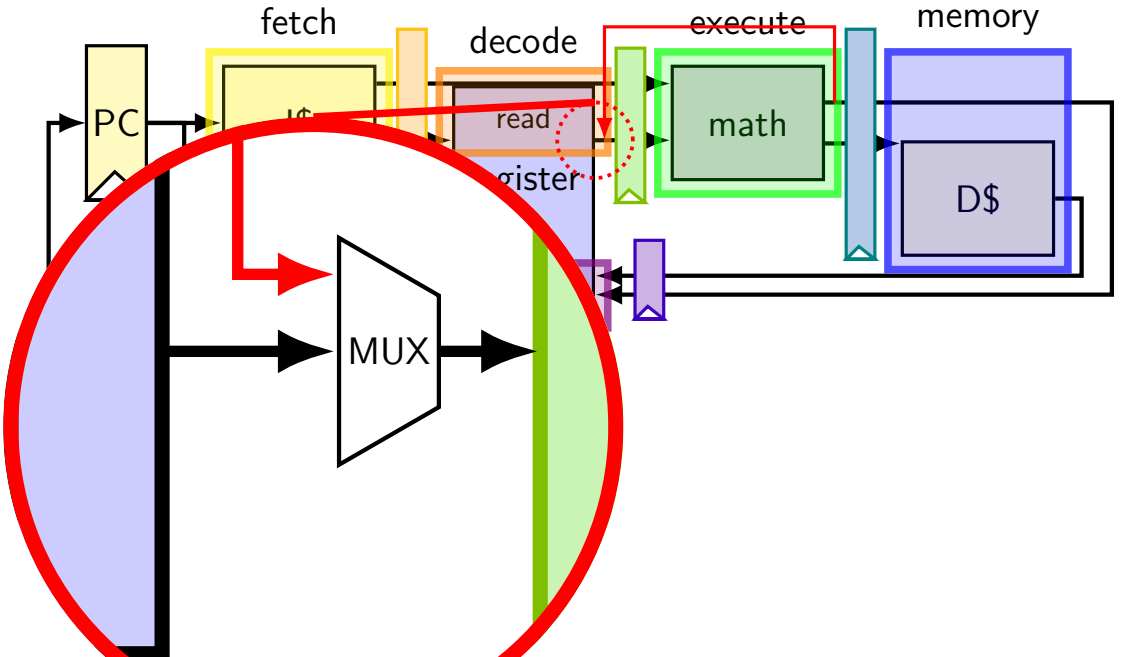
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



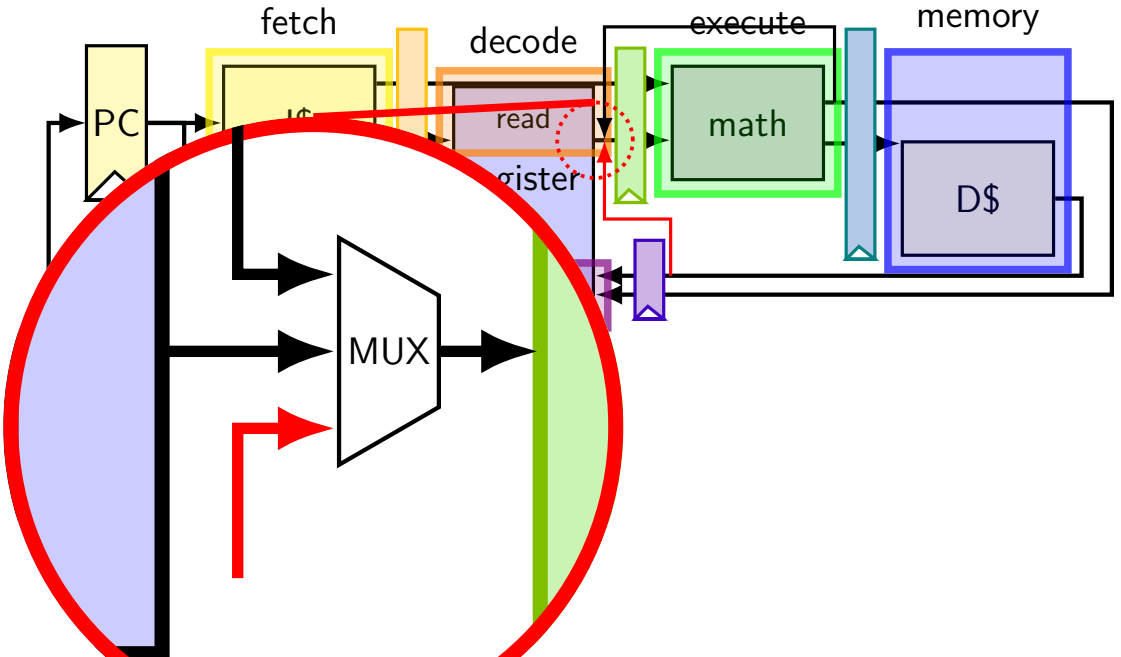
opportunity 2

```
// initially %r8 = 800,
//           %r9 = 900, etc.
0x0: addq %r8, %r9
0x2: nop
0x3: addq %r9, %r8
...
```

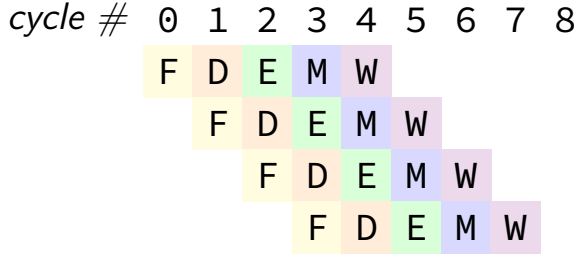
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2	0x3	---	---	800	900	9				
3		9	8	---	---	---	1700	9		
4				900	800	8	---	---	1700	9
5							1700	9	---	---
6									1700	9

should be 1700

exploiting the opportunity



exercise: forwarding paths



in subq, %r8 is _____ addq.

in xorq, %r9 is _____ addq.

in andq, %r9 is _____ addq.

in andq, %r9 is _____ xorq.

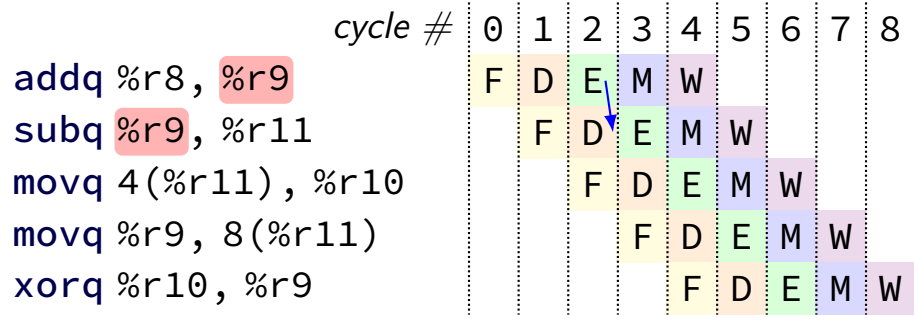
A: not forwarded from

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

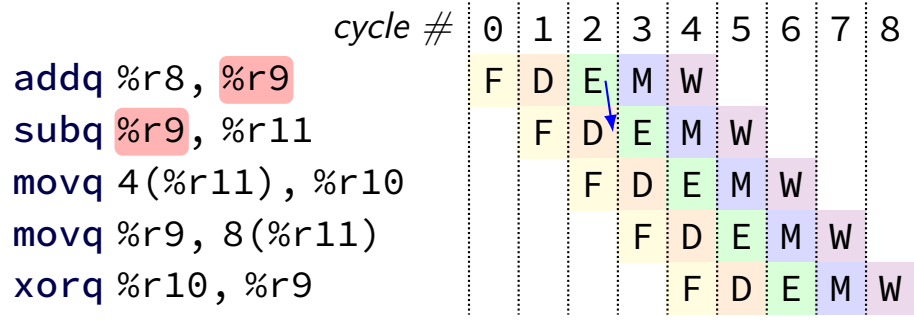
some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`		F	D	E	M	W				
`subq %r9, %r11`			F	D	E	M	W			
`movq 4(%r11), %r10`				F	D	E	M	W		
`movq %r9, 8(%r11)`					F	D	E	M	W	
`xorq %r10, %r9`						F	D	E	M	W

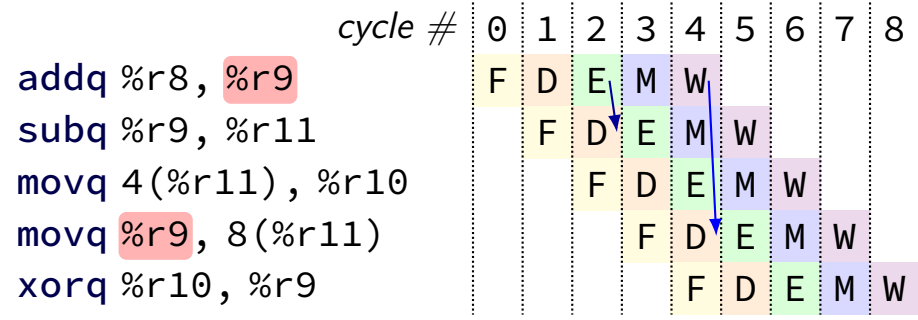
some forwarding paths



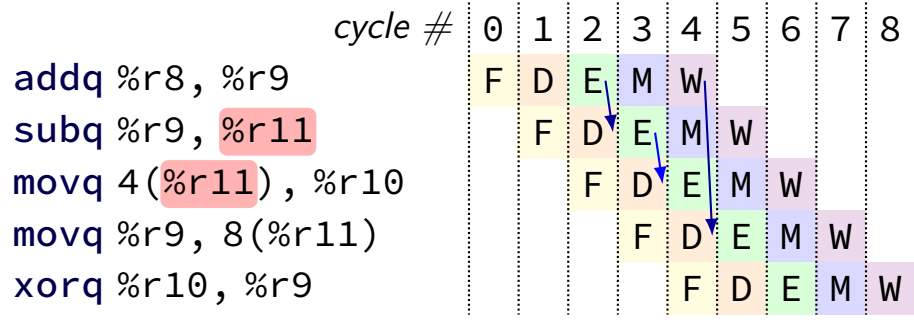
some forwarding paths



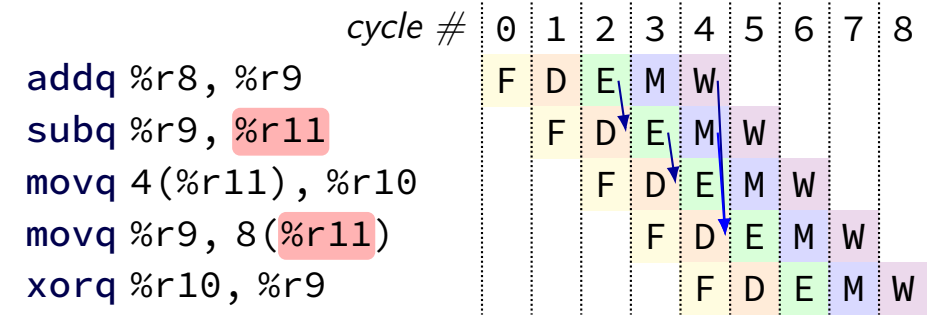
some forwarding paths



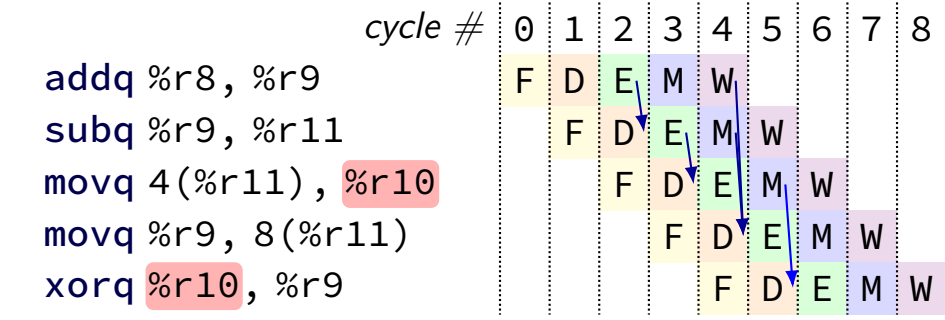
some forwarding paths



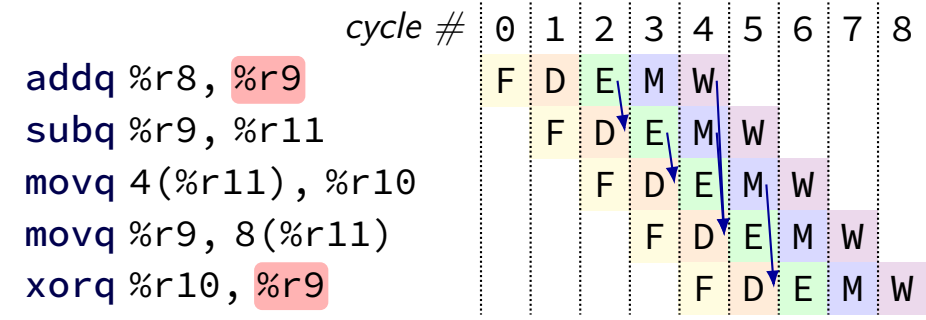
some forwarding paths



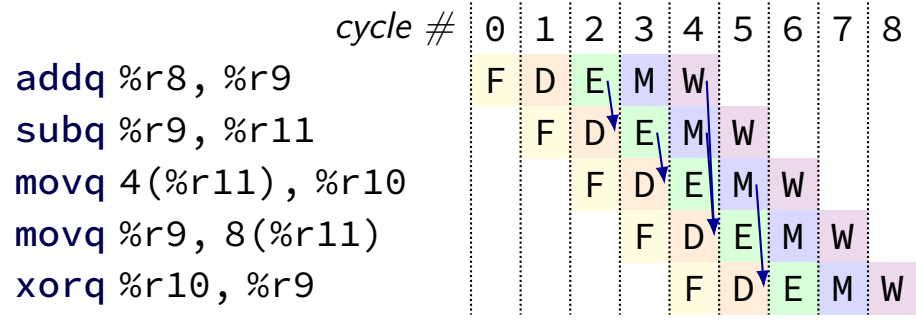
some forwarding paths



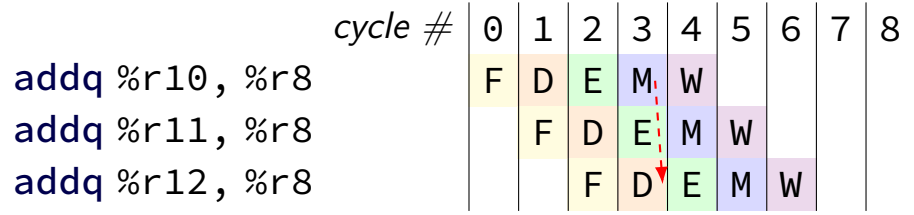
some forwarding paths



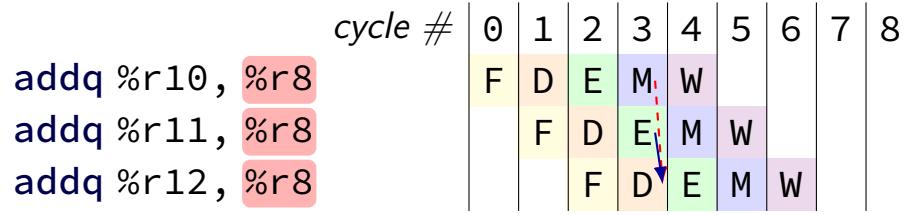
some forwarding paths



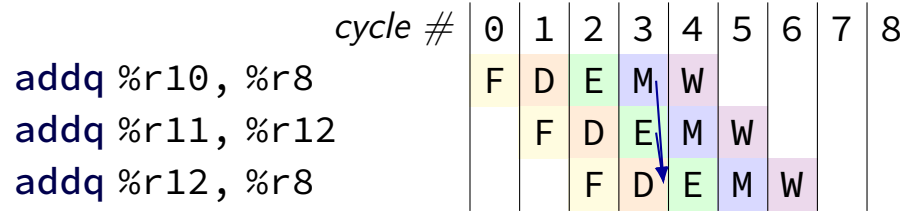
multiple forwarding paths (1)



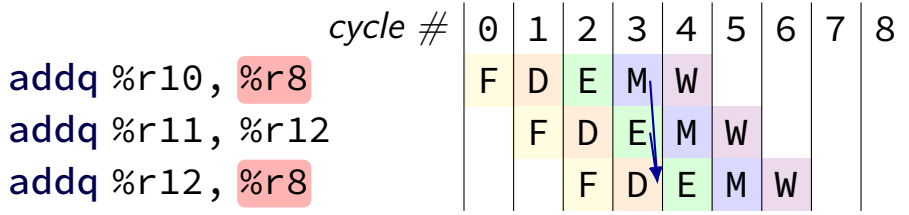
multiple forwarding paths (1)



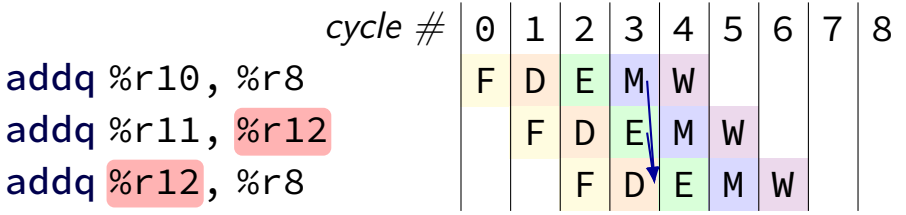
multiple forwarding paths (2)



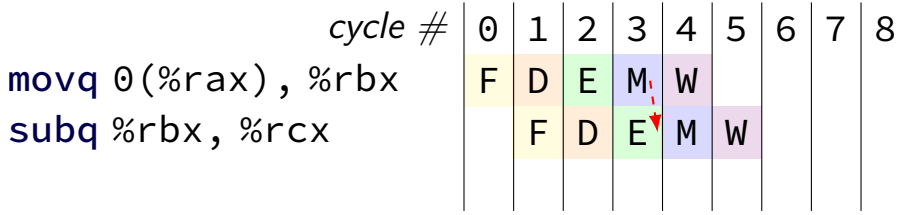
multiple forwarding paths (2)



multiple forwarding paths (2)



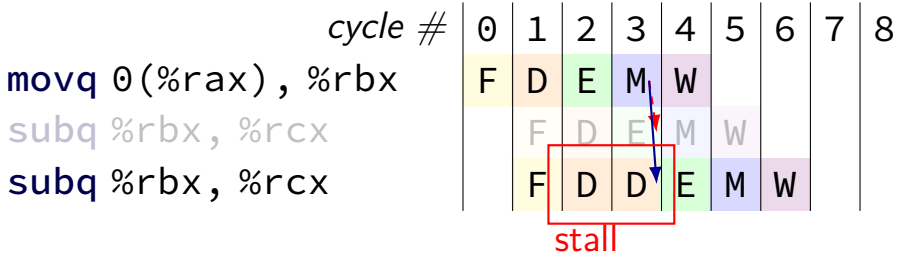
unsolved problem



combine stalling and forwarding to resolve hazard

assumption in diagram: hazard detected in `subq`'s decode stage
(since easier than detecting it in fetch stage)

unsolved problem

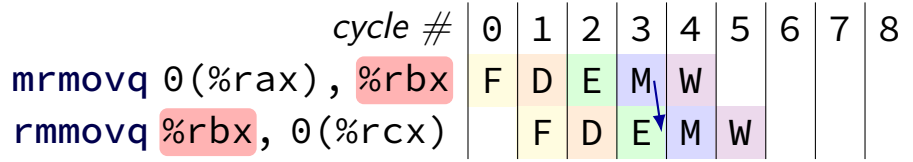


combine stalling and forwarding to resolve hazard

assumption in diagram: hazard detected in `subq`'s decode stage

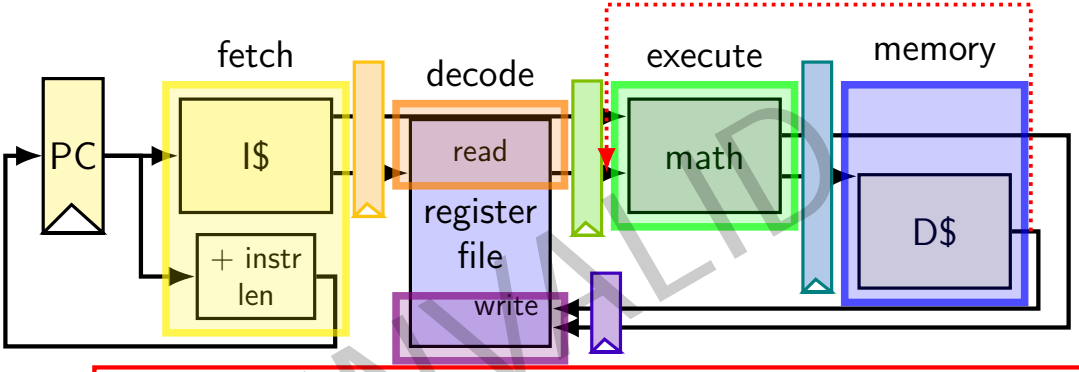
(since easier than detecting it in fetch stage)

solveable problem



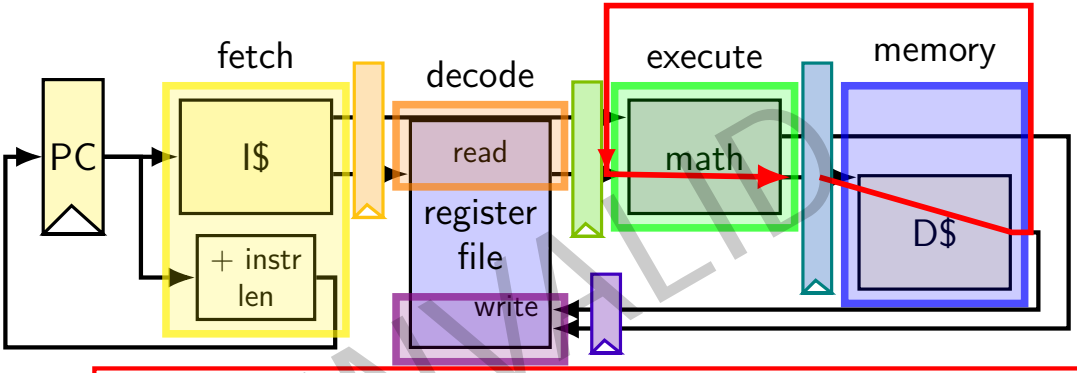
common for real processors to do this
but our textbook only forwards to the end of decode

why can't we...



clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

why can't we...



clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

hazards versus dependencies

dependency — X needs result of instruction Y?

- has potential for being messed up by pipeline
- (since part of X may run before Y finishes)

hazard — will it not work in some pipeline?

- before extra work is done to "resolve" hazards
- multiple kinds: so far, *data hazards*

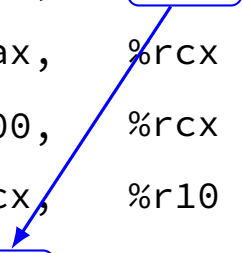
ex.: dependencies and hazards (1)

```
addq    %rax,   %rbx
subq    %rax,   %rcx
movq    $100,   %rcx
addq    %rcx,   %r10
addq    %rbx,   %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

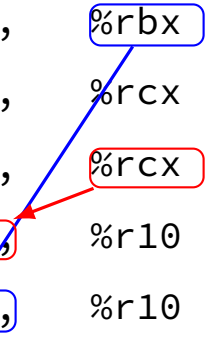
```
addq    %rax, %rbx
subq    %rax, %rcx
movq    $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```



where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

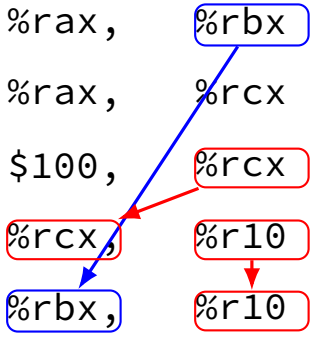
```
addq    %rax, %rbx
subq    %rax, %rcx
movq    $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```



where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq    %rax, %rbx
subq    %rax, %rcx
movq    $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```



where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage   // 5 stage
addq %rax, %r8      //           // W
subq %rax, %r9      // W         // M
xorq %rax, %r10     // EM        // E
andq %r8,  %r11     // D         // D
```

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage   // 5 stage
addq %rax, %r8      //           // W
subq %rax, %r9      // W         // M
xorq %rax, %r10     // EM        // E
andq %r8,  %r11     // D         // D
```

addq/andq is hazard with 5-stage pipeline

addq/andq is **not** a hazard with 4-stage pipeline

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage   // 5 stage
addq %rax, %r8      //           // W
subq %rax, %r9      // W         // M
xorq %rax, %r10     // EM        // E
andq %r8,  %r11     // D         // D
```

more hazards with more pipeline stages

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

result only available near end of second execute stage

where does forwarding, stalls occur?

cycle #	0	1	2	3	4	5	6	7	8
(1) `addq %rcx, %r9`	F	D	E1	E2	M	W			
(2) `addq %r9, %rbx`									
(3) `addq %rax, %r9`									
(4) `movq %r9, (%rbx)`									
(5) `movq %rcx, %r9`									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`	F	D	E1	E2	M	W			
`addq %r9, %rbx`									
`addq %rax, %r9`									
`movq %r9, (%rbx)`									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`	F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	E1	E2	M	W		
`addq %rax, %r9`			F	D	E1	E2	M	W	
`movq %r9, (%rbx)`				F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
`addq %rcx, %r9`	F	D	E1	E2	M	W				
`addq %r9, %rbx`		F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	D	E1	E2	M	W		
`addq %rax, %r9`			F	D	E1	E2	M	W		
`addq %rax, %r9`			F	F	D	E1	E2	M	W	
`movq %r9, (%rbx)`				F	D	E1	E2	M	W	
`movq %r9, (%rbx)`					F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
`addq %rcx, %r9`	F	D	E1	E2	M	W				
`addq %r9, %rbx`		F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	D	E1	E2	M	W		
`addq %rax, %r9`			F	D	E1	E2	M	W		
`addq %rax, %r9`			F	F	D	E1	E2	M	W	
`movq %r9, (%rbx)`				F	D	E1	E2	M	W	
`movq %r9, (%rbx)`					F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8		
`addq %rcx, %r9`	F	D	E1	E2	M	W					
`addq %r9, %rbx`		F	D	E1	E2	M	W				
`addq %r9, %rbx`		F	D	D	E1	E2	M	W			
`addq %rax, %r9`			F	D	E1	E2	M	W			
`addq %rax, %r9`			F	F	D	E1	E2	M	W		
`movq %r9, (%rbx)`				F	D	E1	E2	M	W		
`movq %r9, (%rbx)`					F	D	E1	E2	M	W	
`movq %rcx, %r9`						F	D	E1	E2	M	W

control hazard

```
0x00: cmpq %r8, %r9
0x08: je   0xFFFF
0x10: addq %r10, %r11
```

	fetch	fetch→decode	decode→execute		execute→writeback			
cycle	PC	rA	rB	R[rA]	R[rB]	result	...	...	...
0	0x0								
1	0x8	8	9						
2	???	---	---	800	900				
3	???	---	---	---	---	less than			

control hazard

```
0x00: cmpq %r8, %r9
0x08: je   0xFFFF
0x10: addq %r10, %r11
```

	fetch	fetch→decode	decode→execute		execute→writeback			
cycle	PC	rA	rB	R[rA]	R[rB]	result	...	...	...
0	0x0								
1	0x8	8	9						
2	???	---	---	800	900				
3	???	---	---	---	---	less than			

0xFFFF if R[8] = R[9]; 0x10 otherwise

jXX: stalling?

```
        cmpq %r8, %r9
        jne LABEL        // not taken
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...
```

cmpq %r8, %r9

jne LABEL

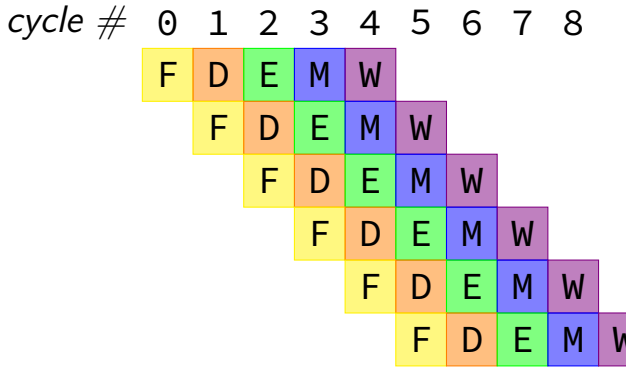
(do nothing)

(do nothing)

xorq %r10, %r11

movq %r11, 0(%r12)

…



jXX: stalling?

```
        cmpq %r8, %r9
        jne LABEL        // not taken
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...
```

cmpq %r8, %r9

jne LABEL

(do nothing)

(do nothing)

xorq %r10, %r11

movq %r11, 0(%r12)

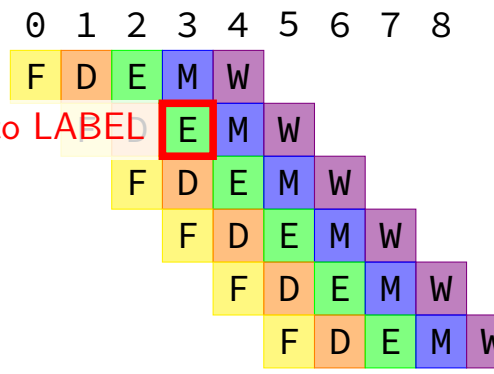
…

jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

```
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
…
```

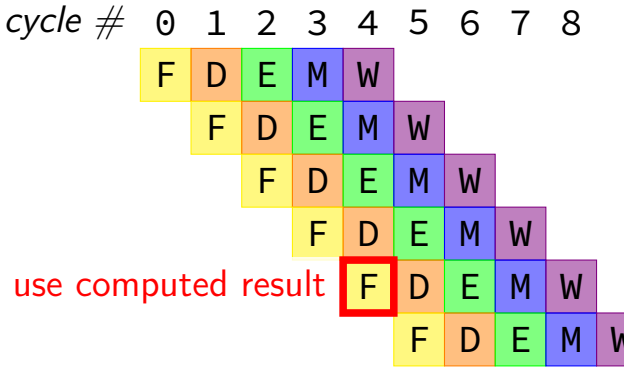
compute jump goes to LABEL



jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
…



making guesses

```
        cmpq %r8, %r9
        jne LABEL
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...

LABEL:  addq %r8, %r9
        imul %r13, %r14
        ...
```

speculate (guess): jne won't go to LABEL

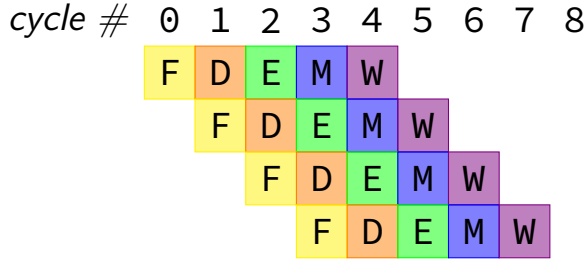
right: 2 cycles faster!; wrong: undo guess before too late

jXX: speculating right (1)

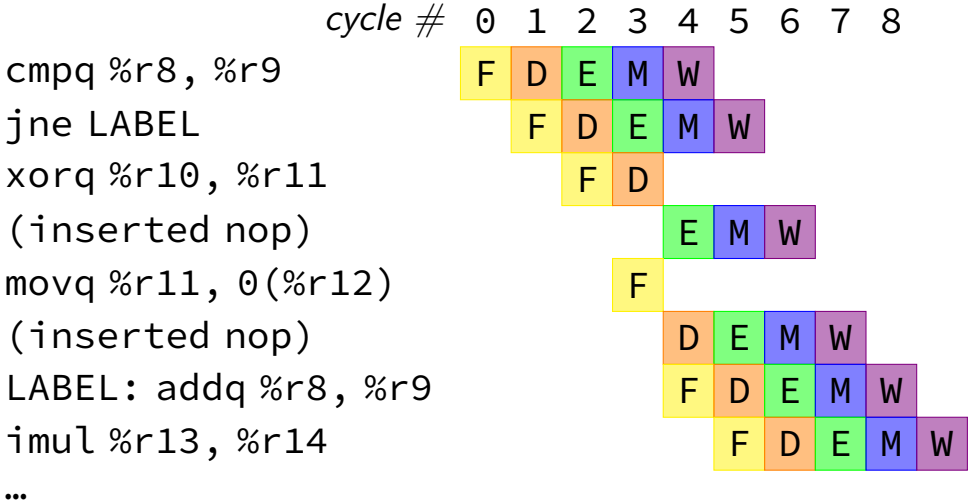
```
        cmpq %r8, %r9
        jne LABEL
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...

LABEL:  addq %r8, %r9
        imul %r13, %r14
        ...
```

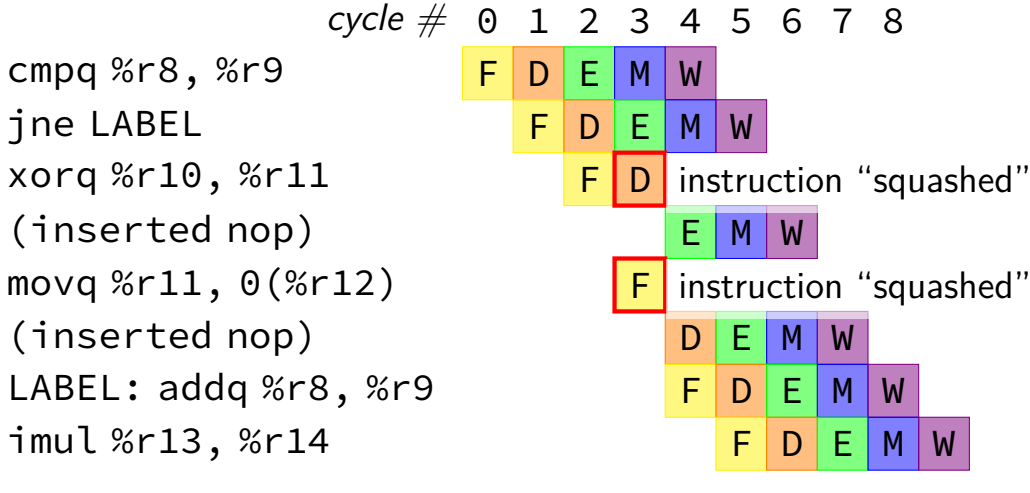
```
cmpq %r8, %r9
jne LABEL
xorq %r10, %r11
movq %r11, 0(%r12)
…
```



jXX: speculating wrong



jXX: speculating wrong



“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

more complicated pipelines: replace written values in cache/registers/etc.

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*

* — ignoring data hazards

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*

* — ignoring data hazards

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

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 mispreditions 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?)
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	…

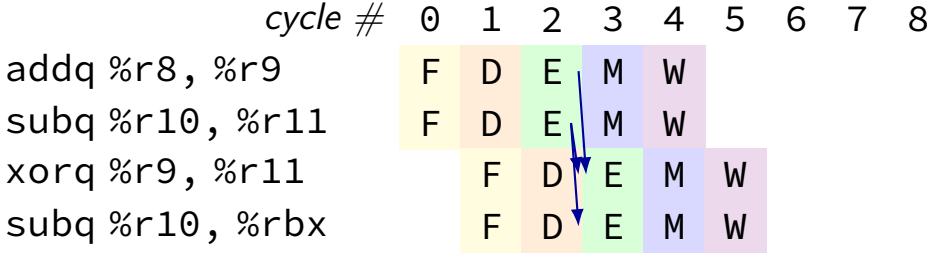
```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

beyond pipelining: multiple issue

start more than one instruction/cycle

multiple parallel pipelines; many-input/output register file

hazard handling much more complex



beyond pipelining: out-of-order

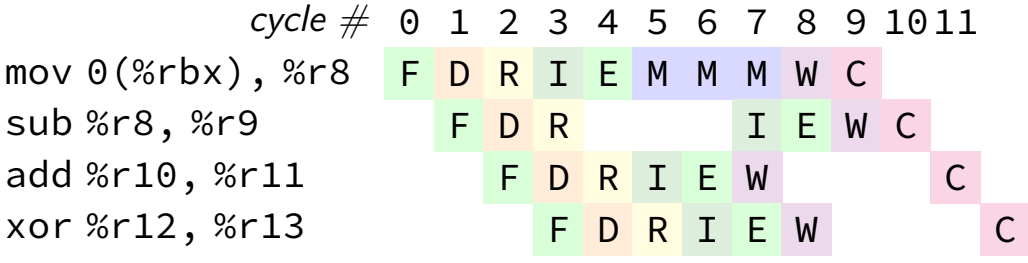
find later instructions to do instead of stalling

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?

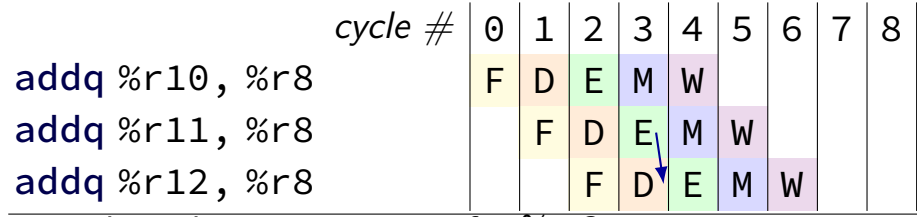
problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

read-after-write examples (1)



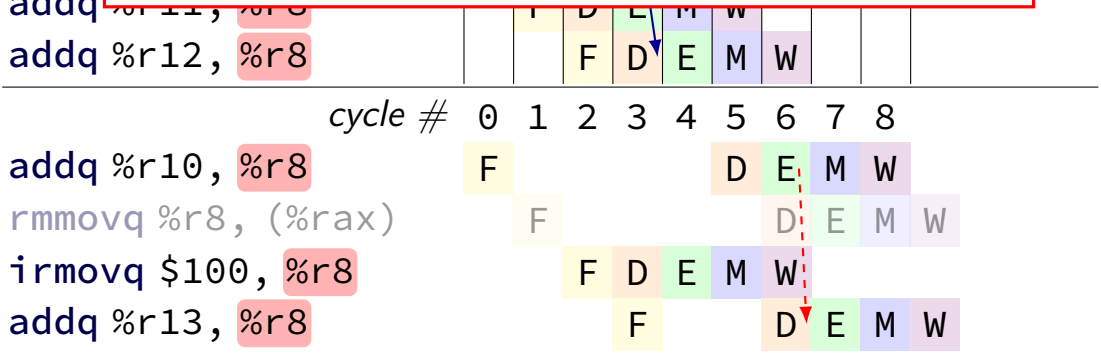
normal pipeline: two options for %r8?

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

read different version than the one written

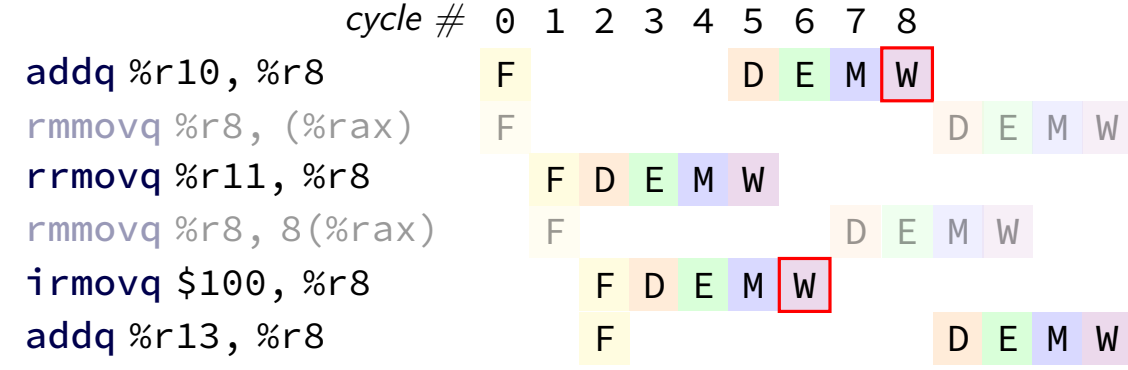
- represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

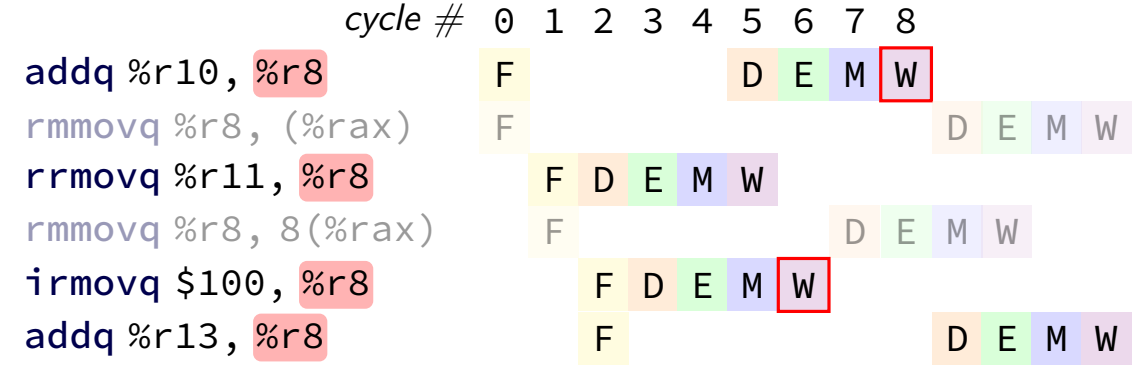
for now: version numbers

later: something simpler to implement

write-after-write example

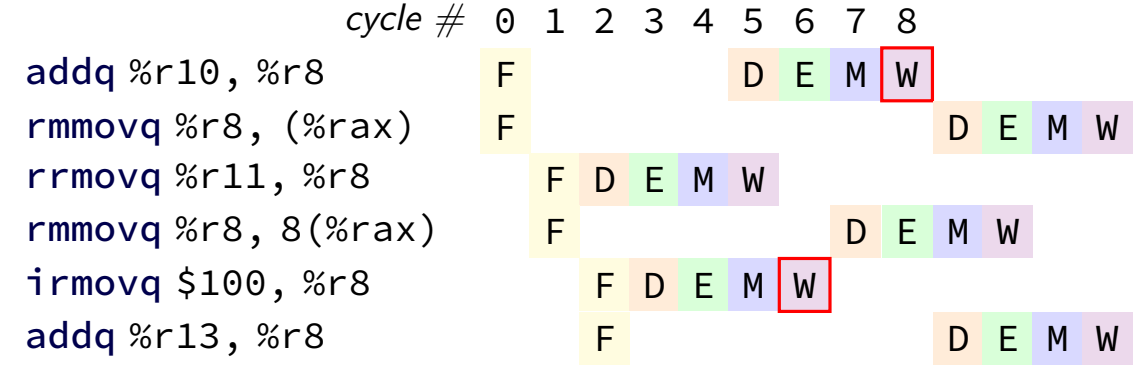


write-after-write example



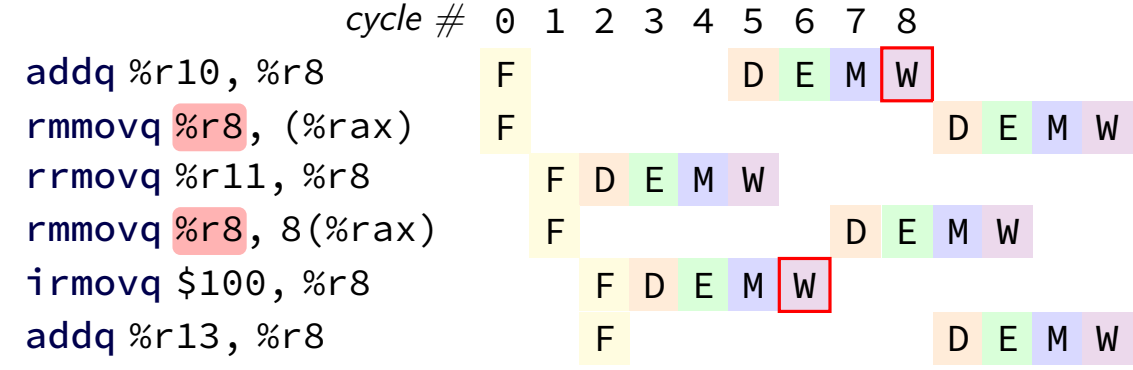
out-of-order execution:
if we don't do something, newest value could be overwritten!

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

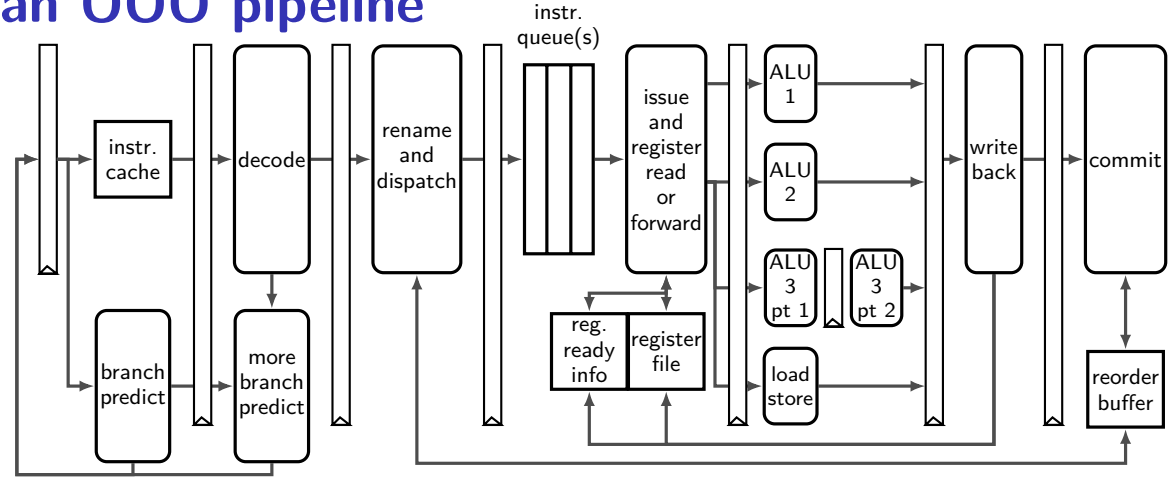
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

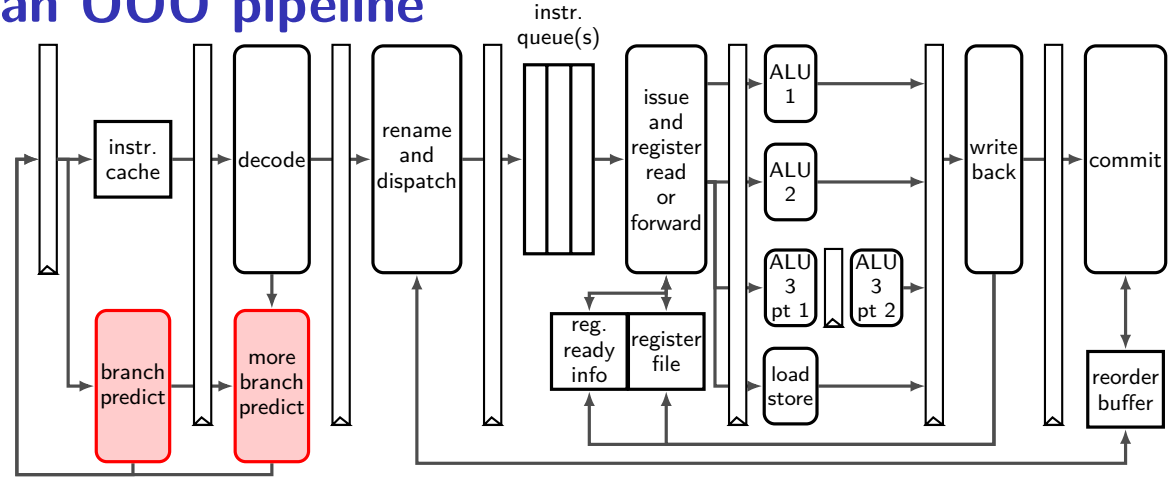
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

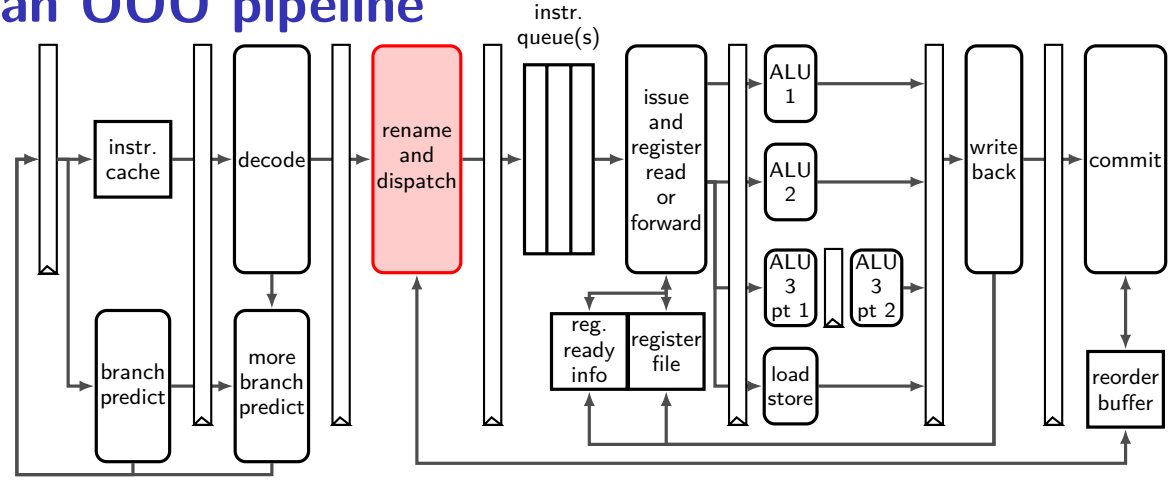


an OOO pipeline



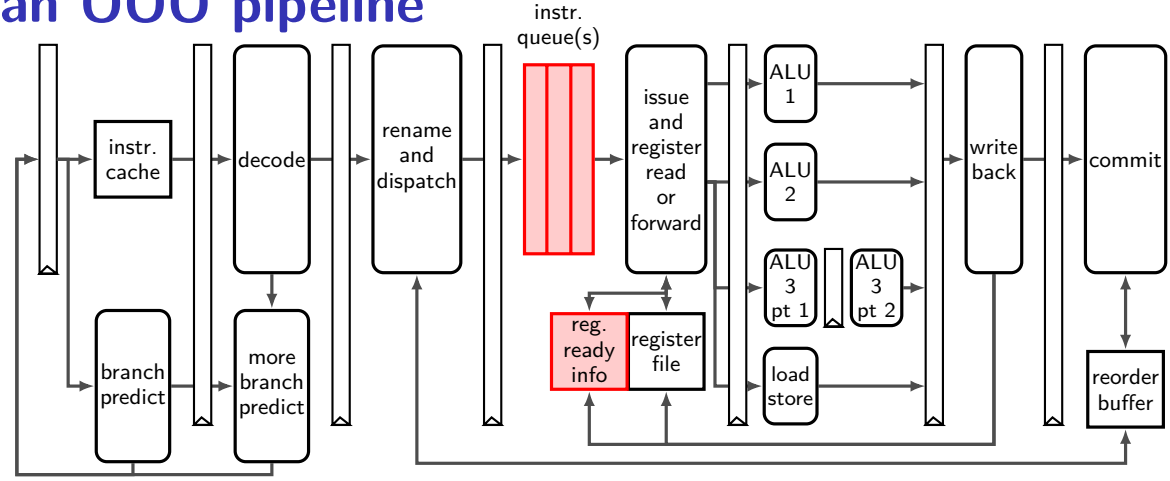
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



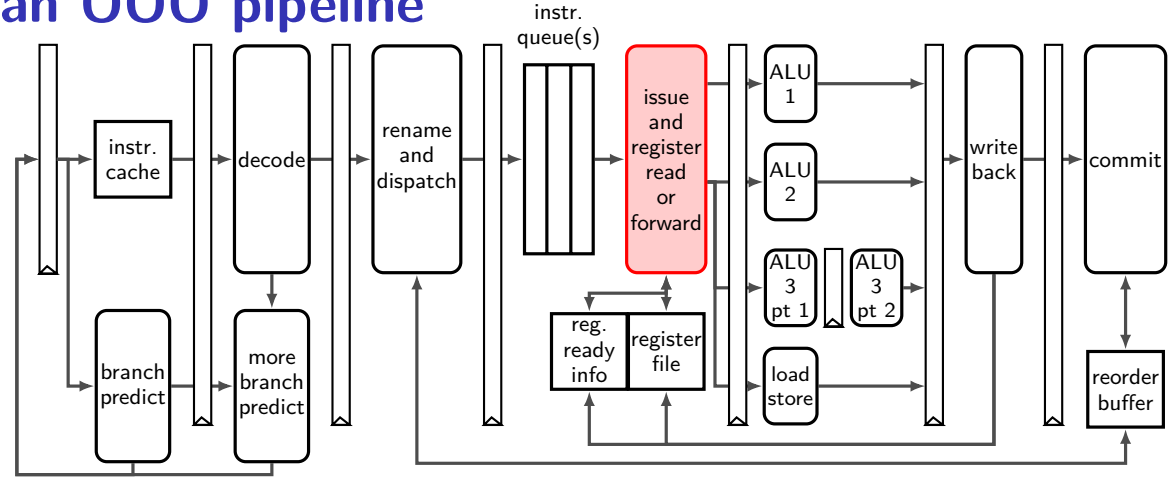
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



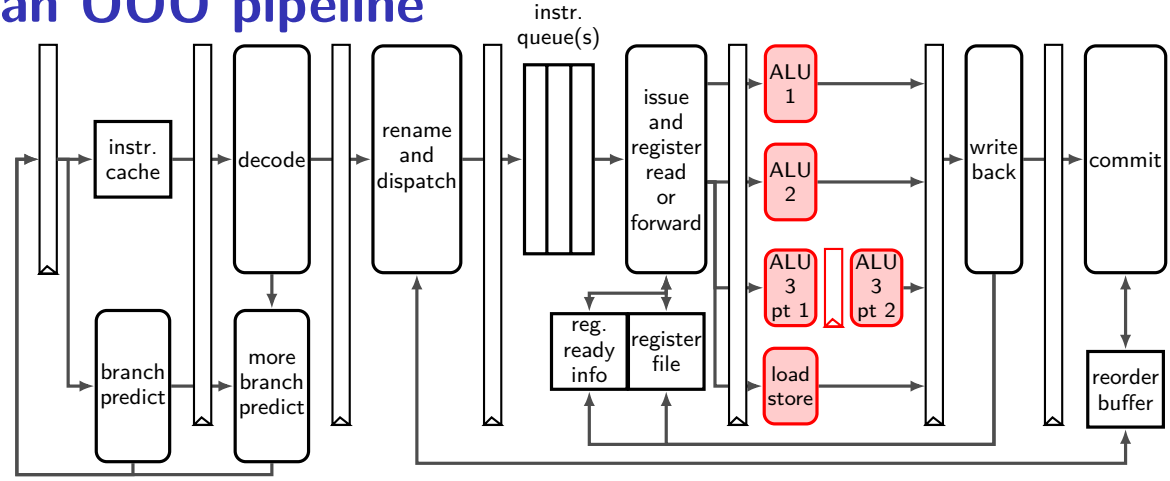
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



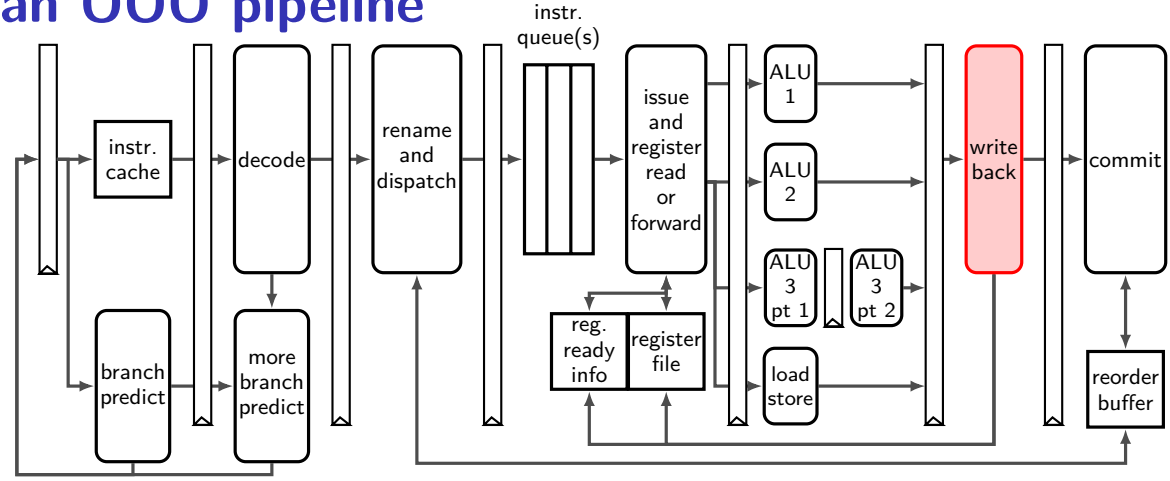
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



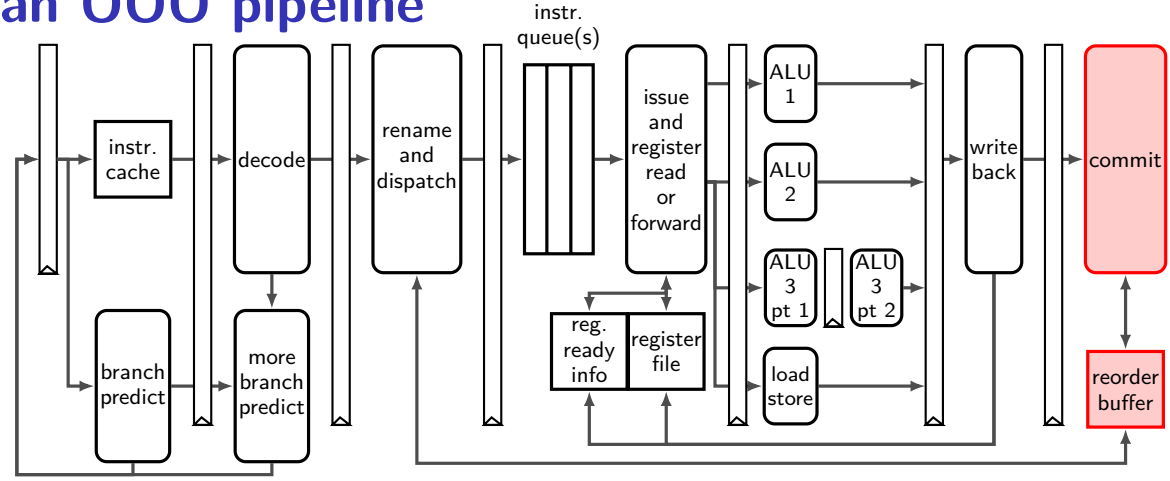
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



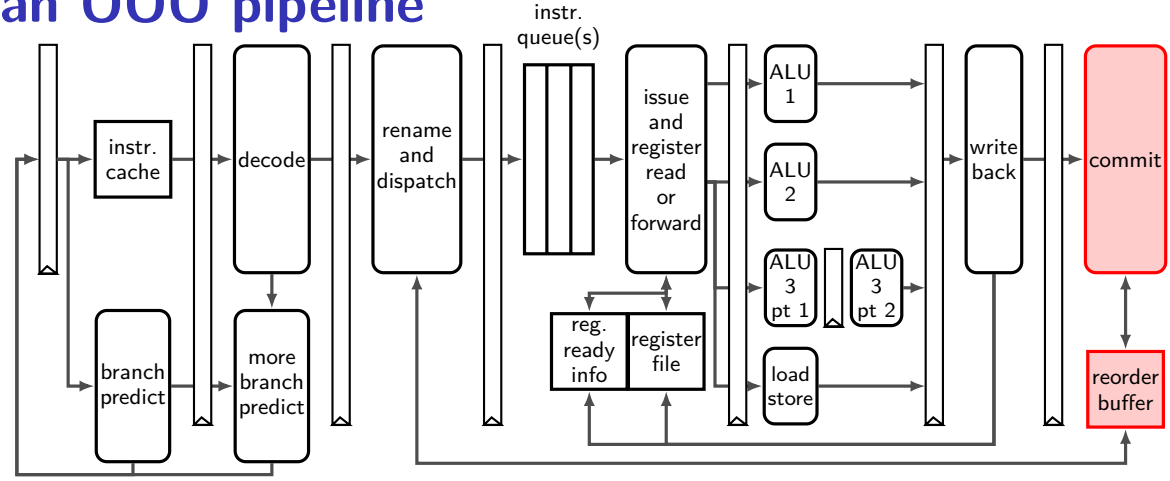
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



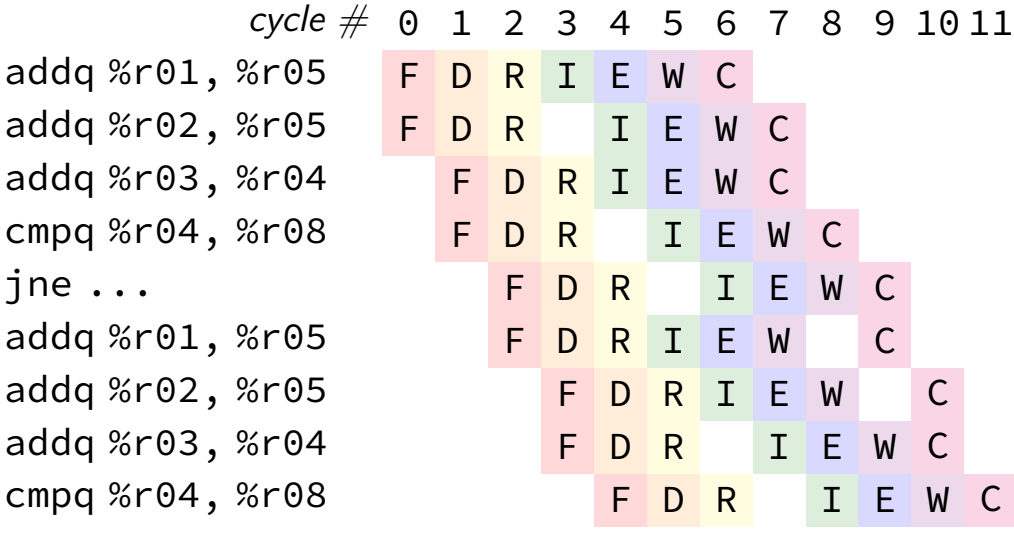
new commit (sometimes *retire*) stage finalizes instruction
figures out when physical registers can be reused again

an OOO pipeline

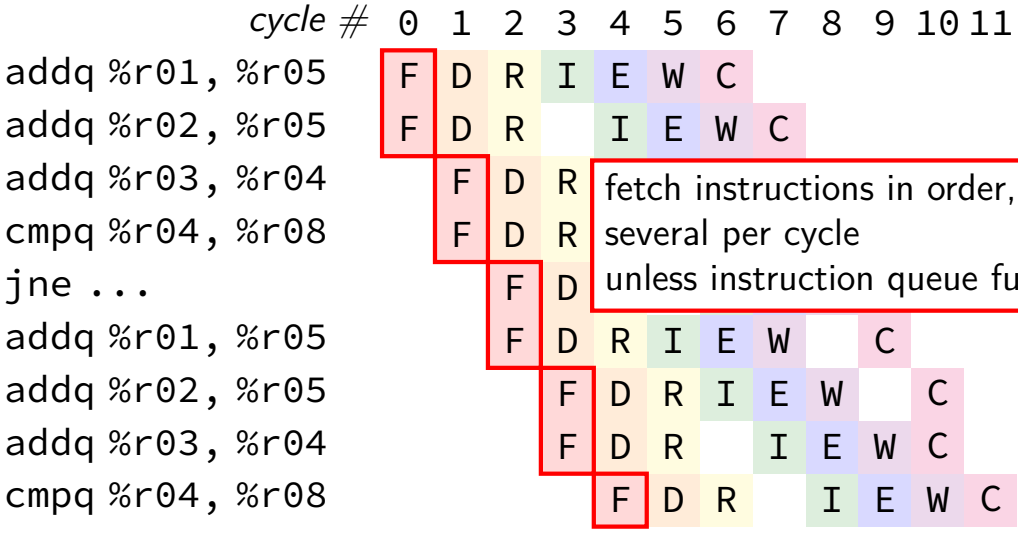


commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.

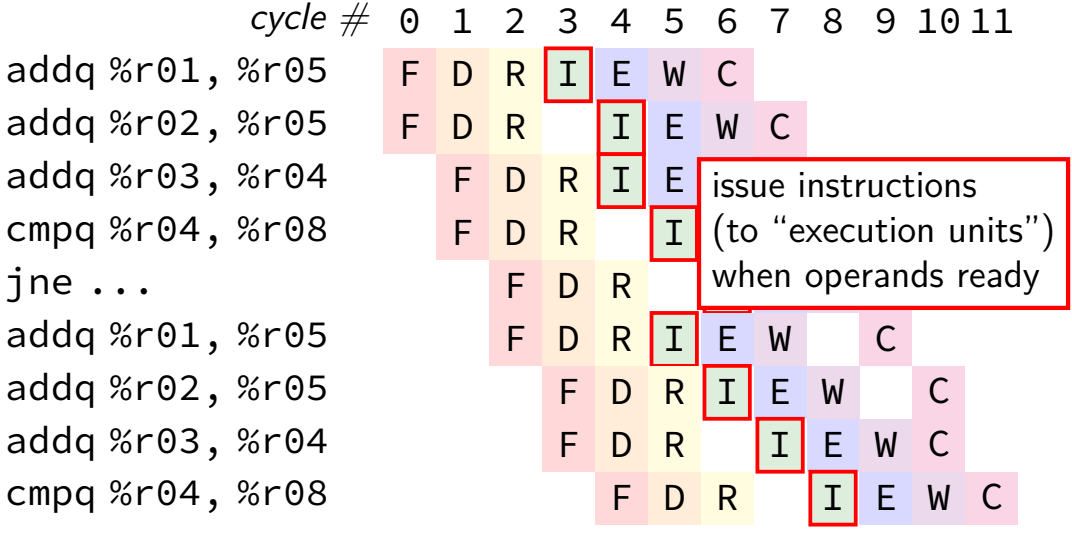
an OOO pipeline diagram



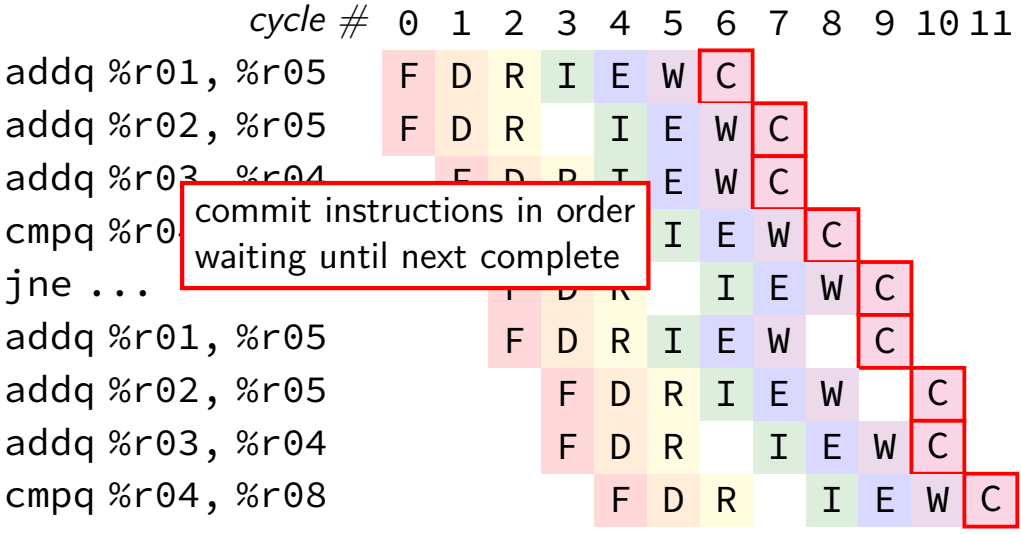
an OOO pipeline diagram



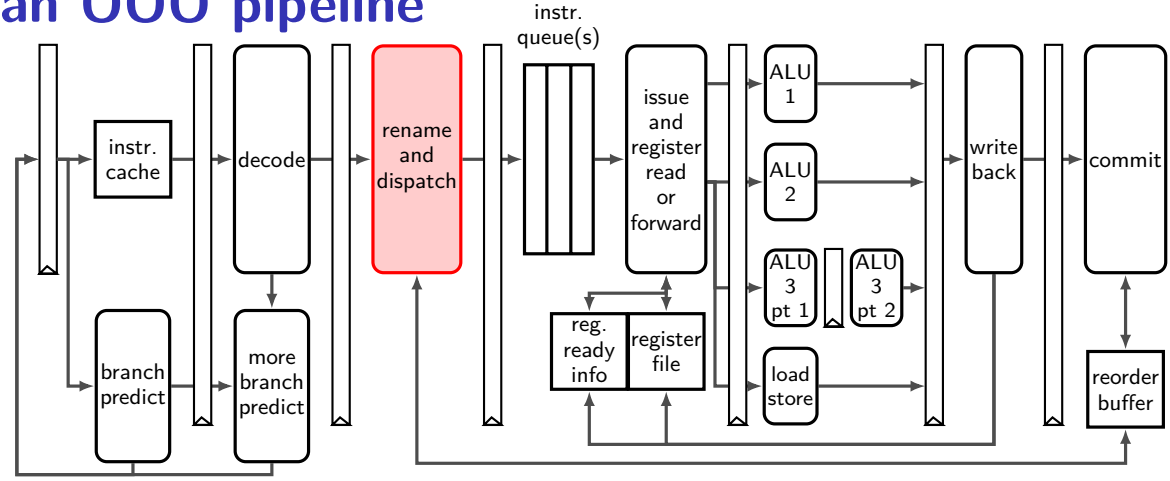
an OOO pipeline diagram



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

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
...	...

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

free reg list

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

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
...	...

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

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

renamed

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

```
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 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
`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`	
`addq %r11, %r8`	

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

```
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
mrmovq 8(%x20), (memory) → %x21
irmovq $100 → %x23
```

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` → `%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`	`irmovq $100` → `%x23`
`addq %r11, %r8`	`addq %x21, %x23` → `%x24`

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
```

renamed

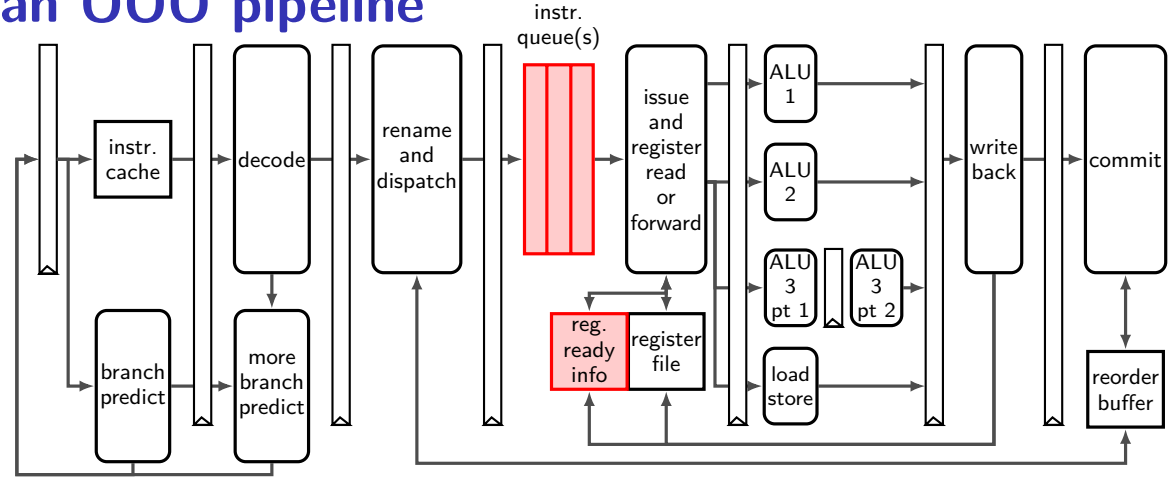
arch → phys

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

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`
...	...

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, %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	`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	`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

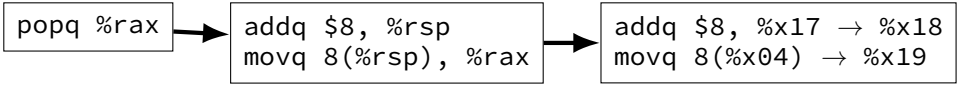
register renaming: missing pieces

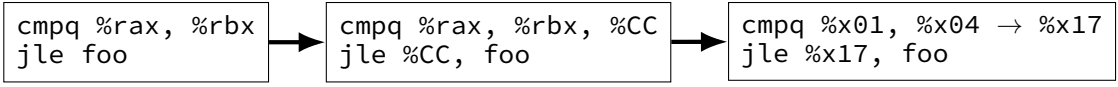
what about "hidden" inputs like `%rsp`, condition codes?

one solution: translate to intructions with additional register parameters

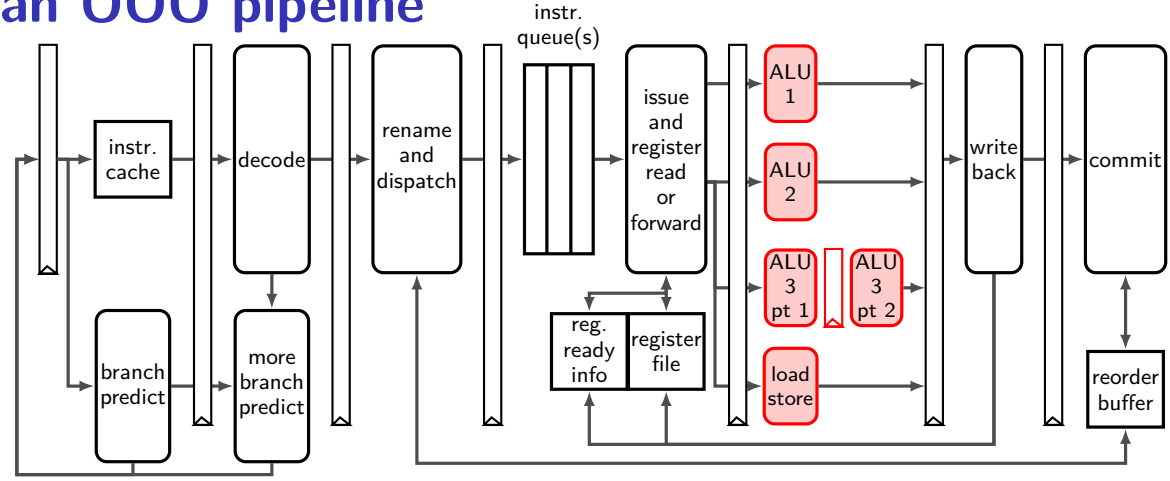
- making %rsp explicit parameter
- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones





an OOO pipeline



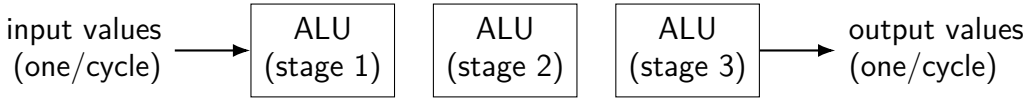
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



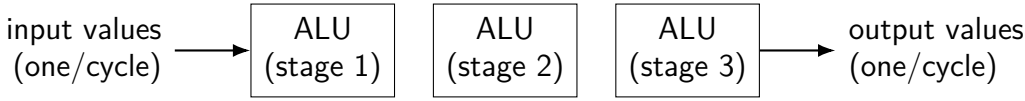
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



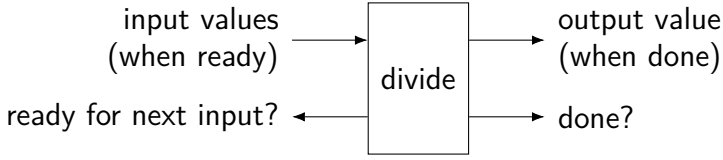
exercise: how long to compute $A \times (B \times (C \times D))$?

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, ...`
…	…

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

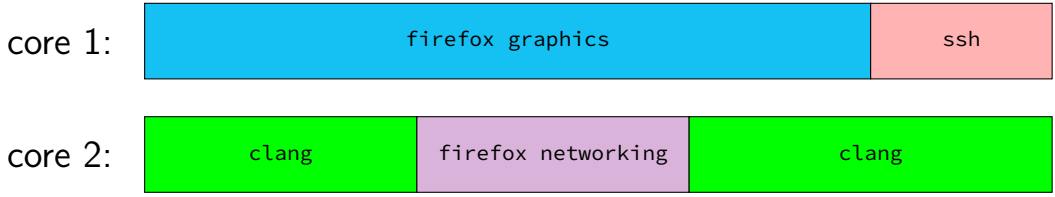
need to track all uncommitted instructions

- 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)

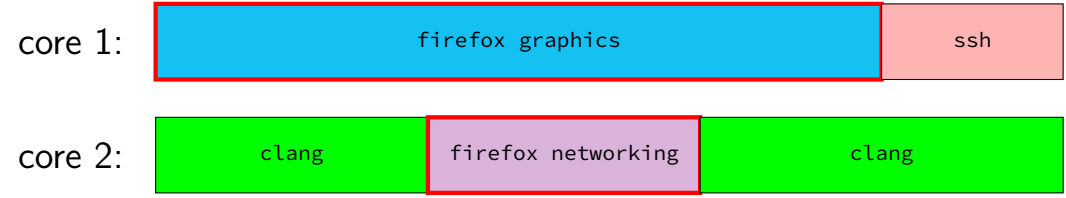
- e.g. Intel Skylake: approx 16 cycles (v. 2 for pipehw2 CPU)

multiple cores+threads



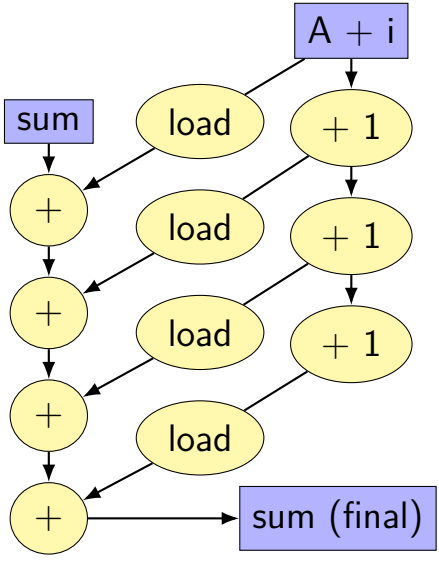
multiple cores? each core still divided up

multiple cores+threads



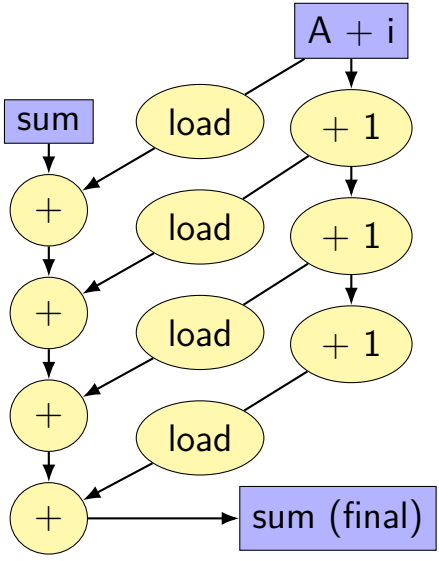
one program with multiple *threads*

data flow model and limits



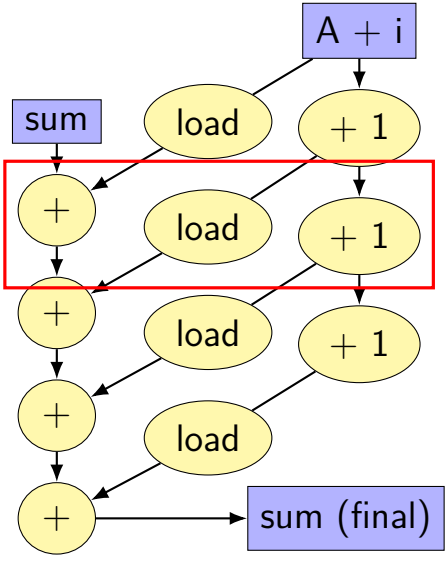

```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



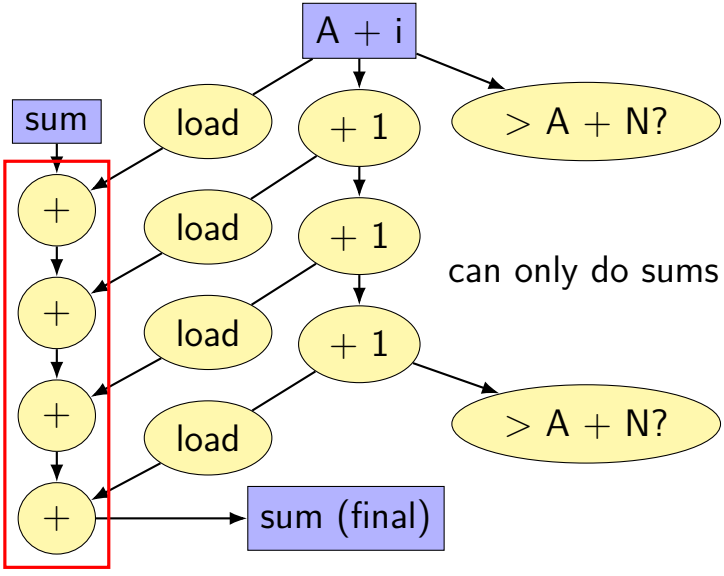
each yellow box = instruction
arrows = dependences
instructions only executed when dependencies

data flow model and limits



three ops/cycle (if each one cycle)

data flow model and limits



can only do sums one at a time

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
```

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

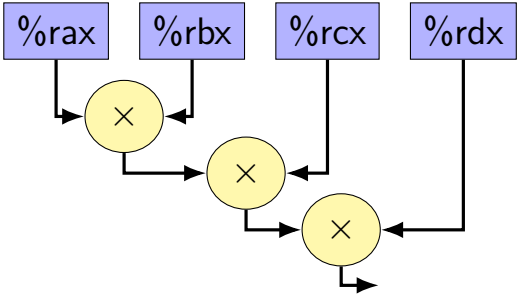
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

reassociation

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

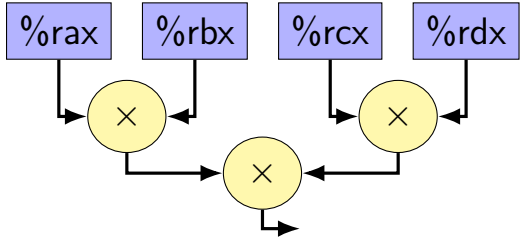
$((a \times b) \times c) \times d$

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

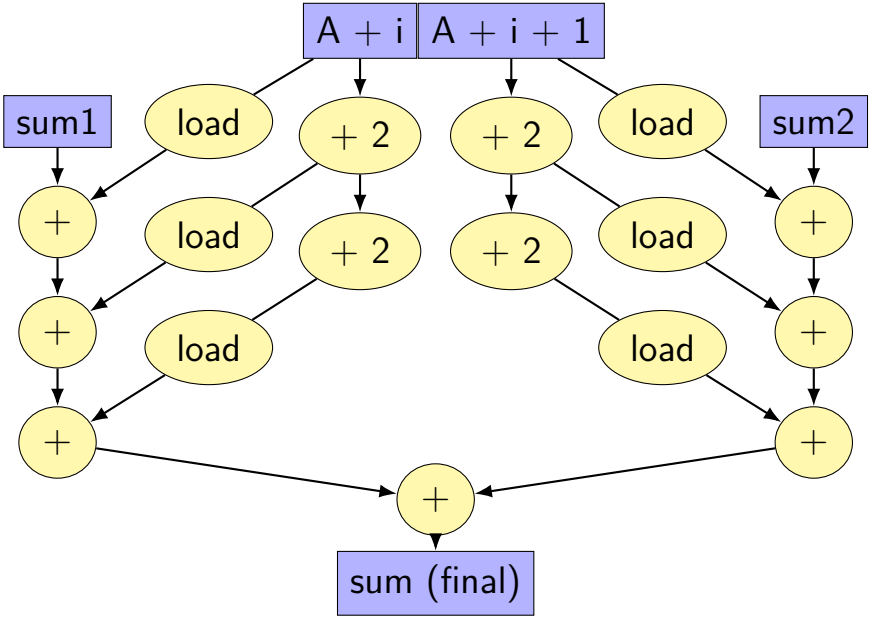


$(a \times b) \times (c \times d)$

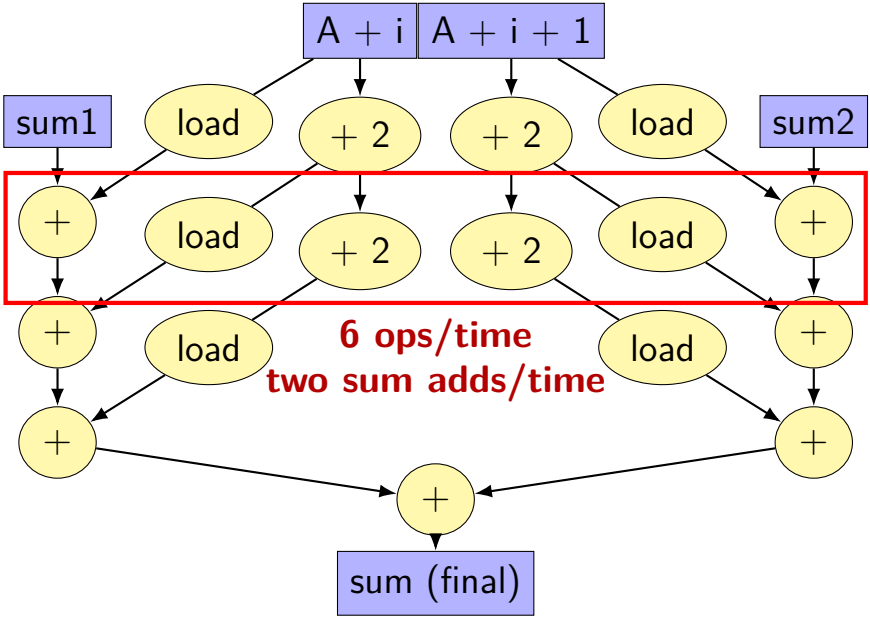
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



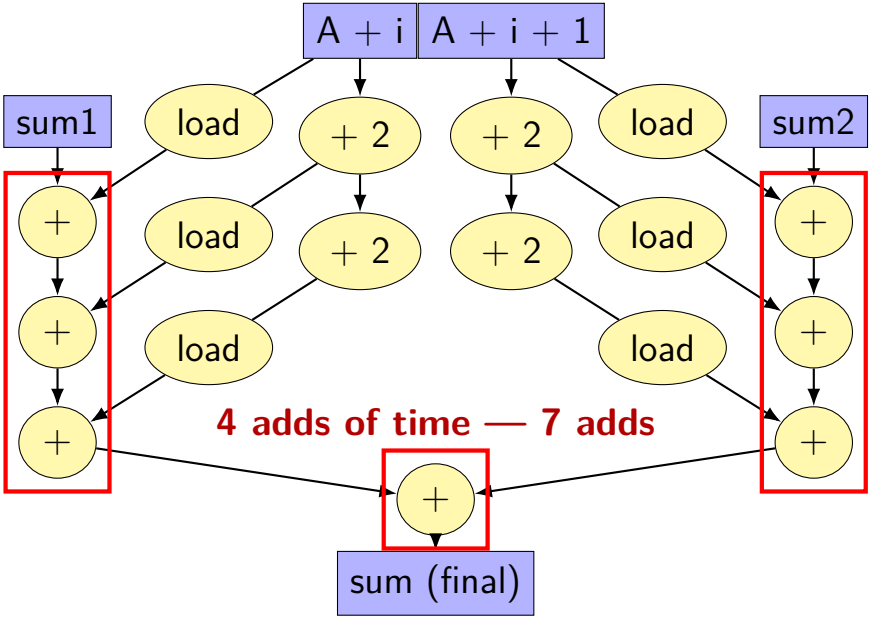
better data-flow



better data-flow



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

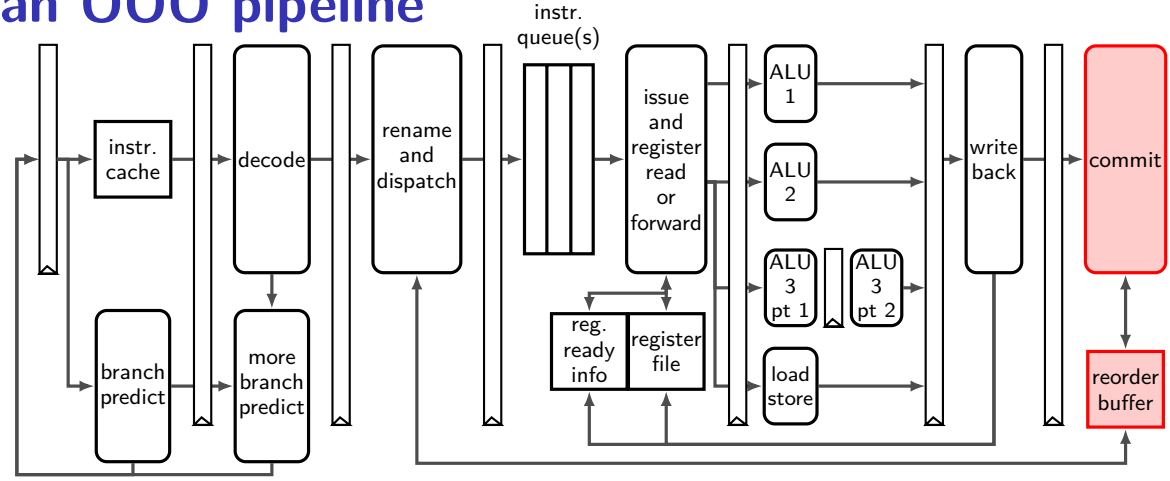
4 ALU functional units
- but some can handle more/different types of operations than others

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 → 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
…
…

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		
32	0x1230	%rdx / %x19		

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 →

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		
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
…
…

reorder buffer (ROB)

remove here → on commit

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
…
…

reorder buffer (ROB)

remove here on commit →

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
…	…

free list

~~%x19~~
%x13
…
…

phys → arch. reg for committed

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

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
…	…

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 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
…	…

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

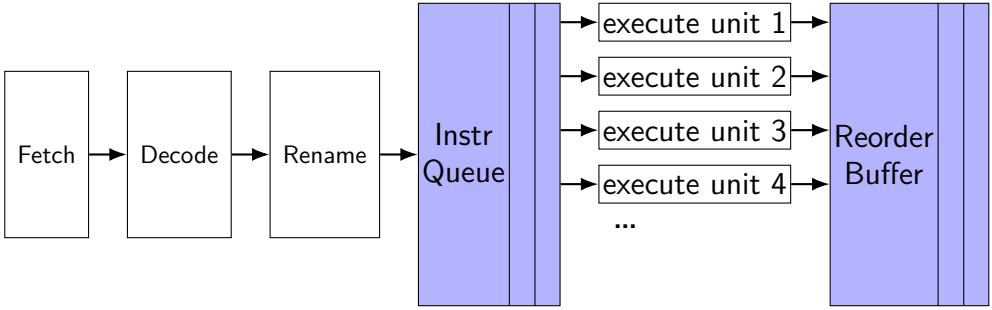
- don't need to reconstruct the table
- (but how to efficiently store them)

can reconstruct register map before we commit the branch instruction

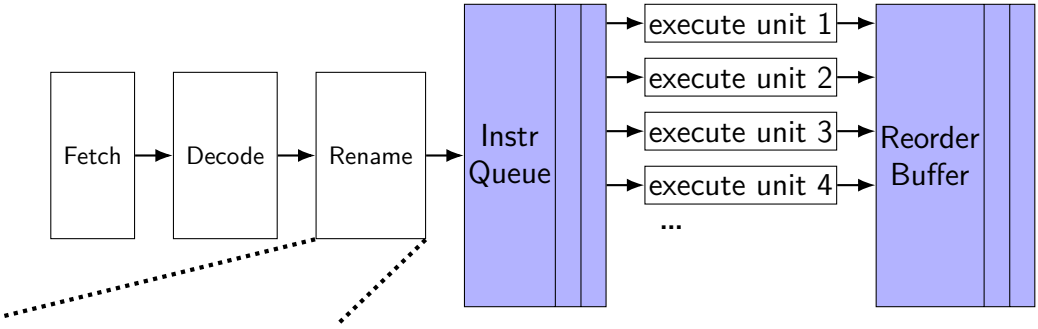
- need to let reorder buffer be accessed even more?

can track more/different information in reorder buffer

exceptions and OOO (one strategy)



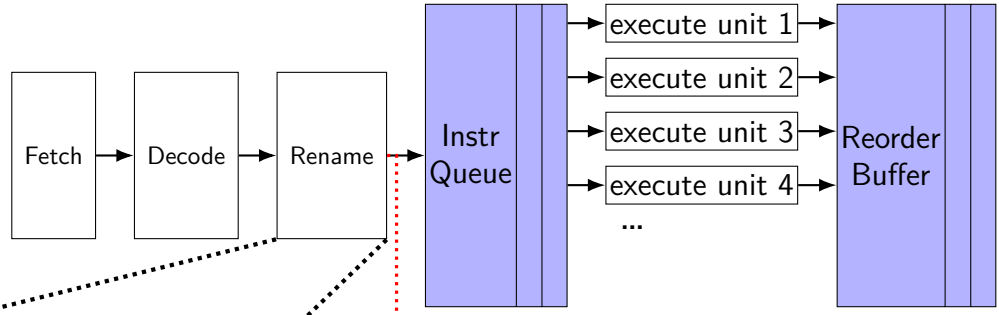
exceptions and OOO (one strategy)



X19
X23
...

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

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

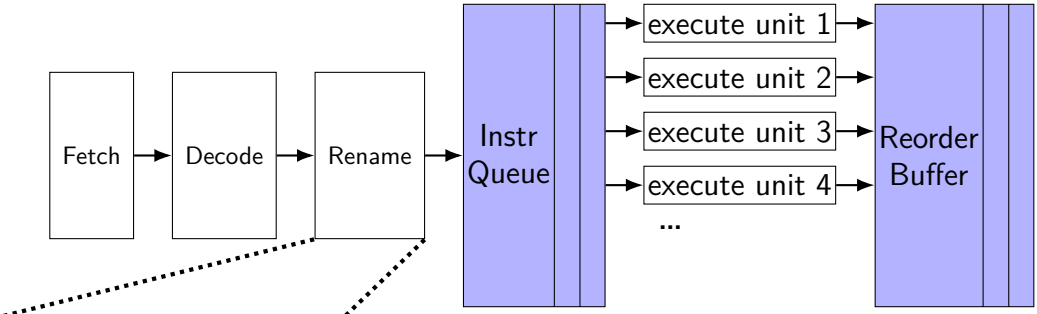
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

done instrs committed in order

new instrs added

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)



free regs

X19
X23
...

for new instrs

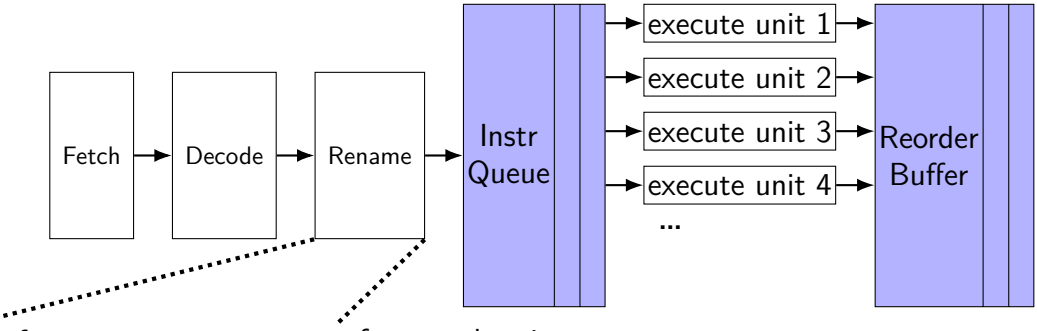
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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)



free regs

X19
X23
...

for new instrs

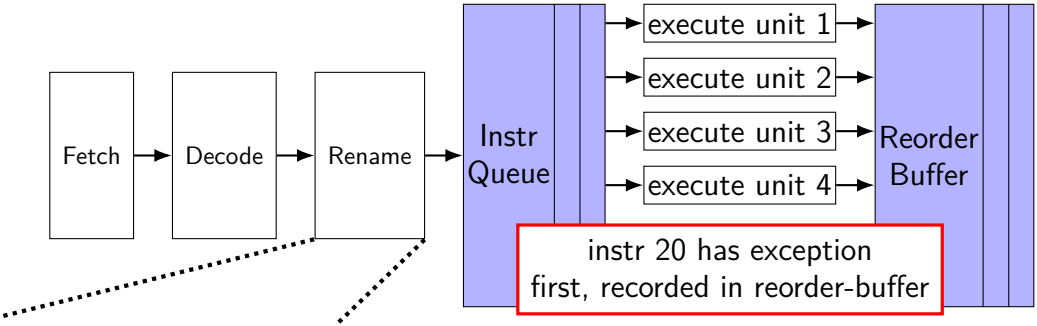
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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)



free regs

X19
X23
...

for new instrs

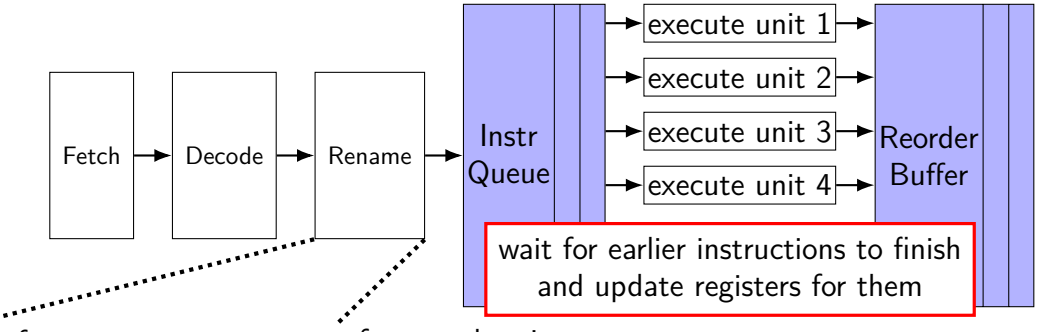
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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)



free regs

X19
X23
...

for new instrs

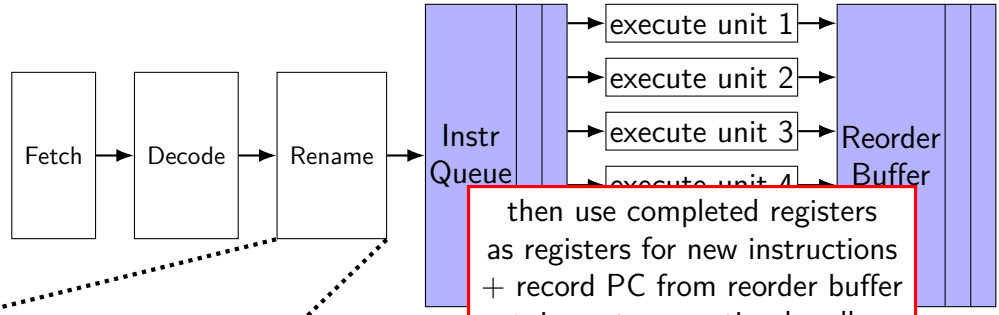
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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)



free regs

X19
X23
...

for new instrs

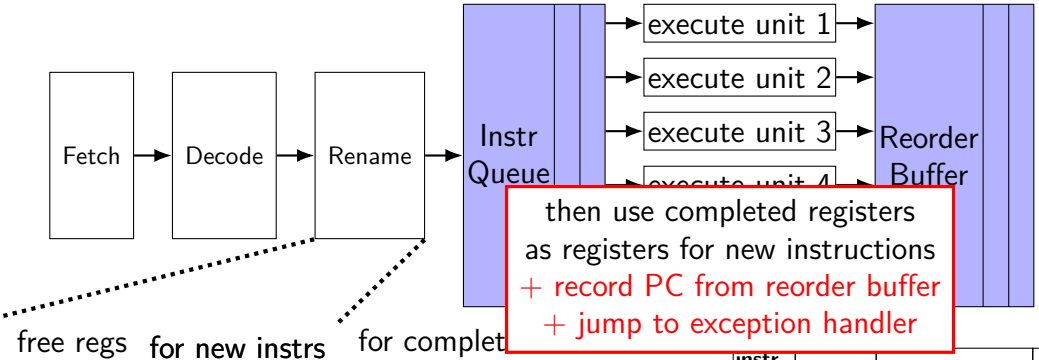
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
...	...

for complet

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)



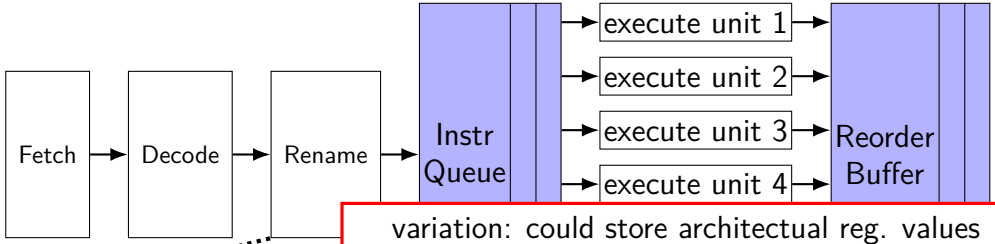
X19
X23
...

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 architectual reg. values instead of mapping for completed instrs.
(and copy values instead of mapping on exception)

free regs

X19
X23
...

for new instrs

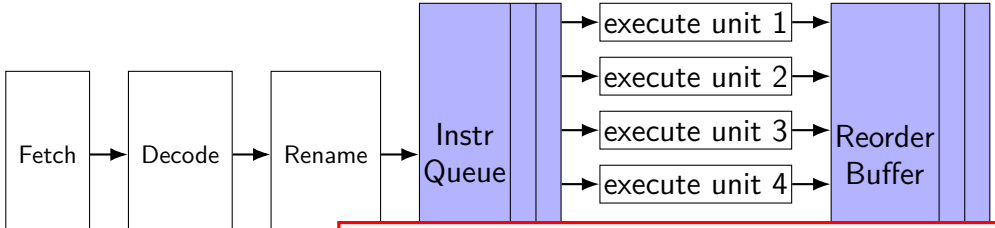
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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

X19
X23
...

for new instrs

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

for complete instrs

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 + uncommited 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

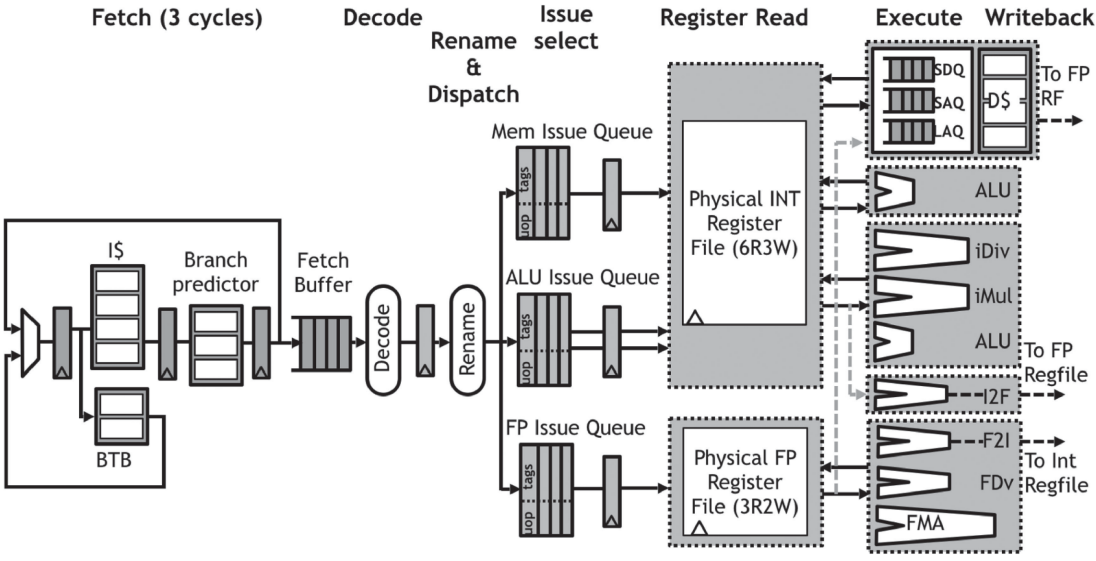


Figure from Celio et al., "BROOM: An Open Source Out-Of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS"

backup slides