connecting things

how to (in hardware) connect A and B?

A

B

connecting things
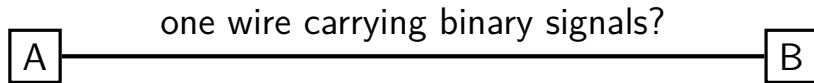
how to (in hardware) connect A and B?

one wire carrying binary signals?

connecting things

how to (in hardware) connect A and B?

A ← - - - - - - - - - - - - - → ← - - - - - - - - - - - - - → B

collision?

connecting things

how to (in hardware) connect A and B?

one-way communication only?
called *simplex*

connecting things

how to (in hardware) connect A and B?

| A | ········▶ | B |

...and later

| A | ◀········ | B |
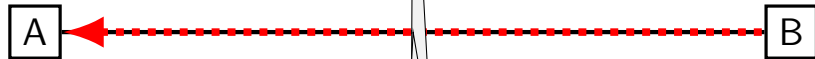
taking turns, but one-way
called *half-duplex*
challenge: how to agree who's turn?
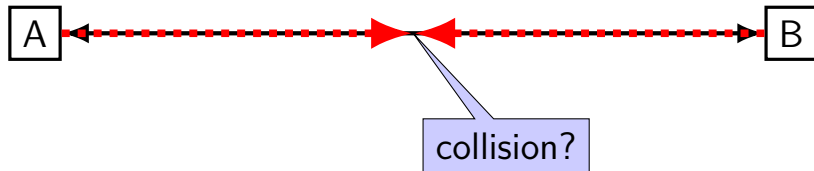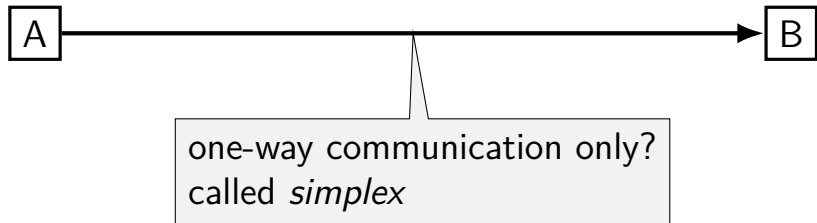
connecting things

how to (in hardware) connect A and B?

A B

both ways at the same time
called *full duplex* (or *duplex*)
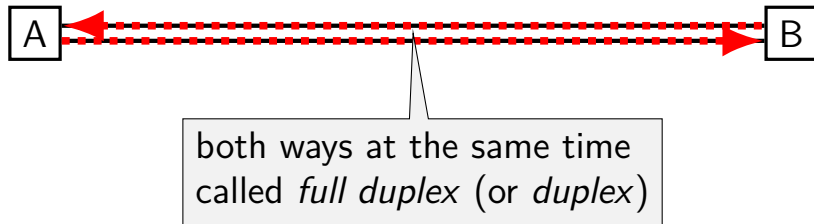
connecting things

how to (in hardware) connect A and B?

A ← - → B

here: duplex via multiple wires (simplest scheme)
can achieve effect electrically/etc. via one wire
example: cable Internet
(how is topic for ECE class)

connecting things

A

D

B

E

C

F

how to connect?

connecting things

all-to-all
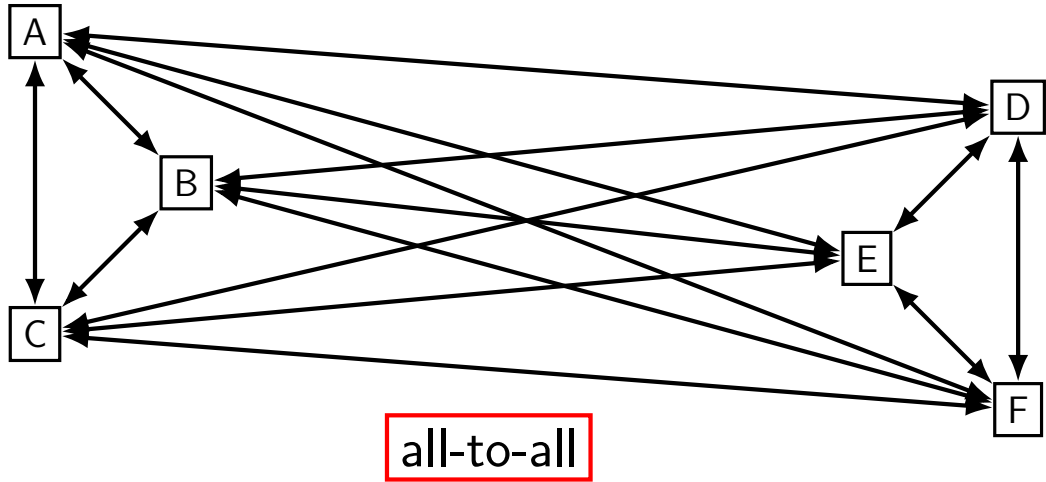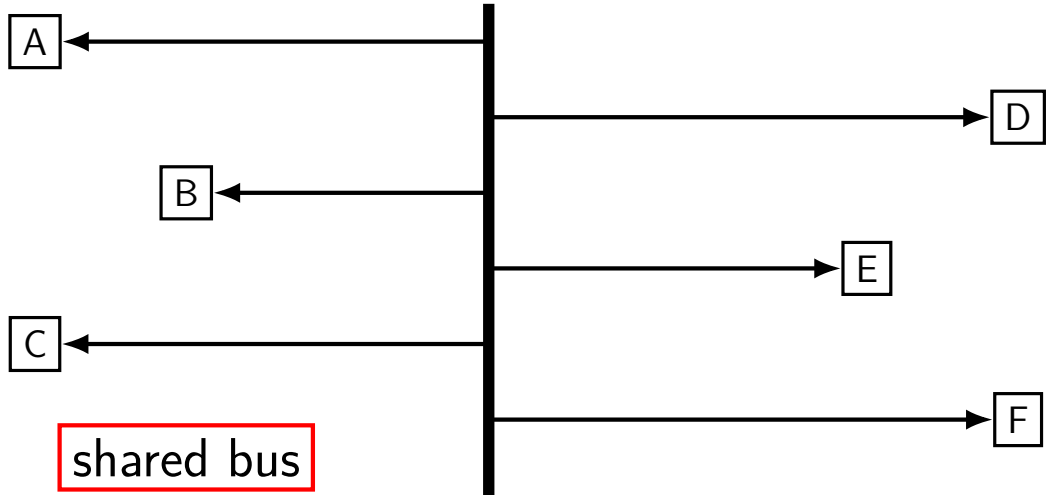
connecting things

shared bus

connecting things

shared wire(s)
need some way to take turns

A

D

B

E

C

F
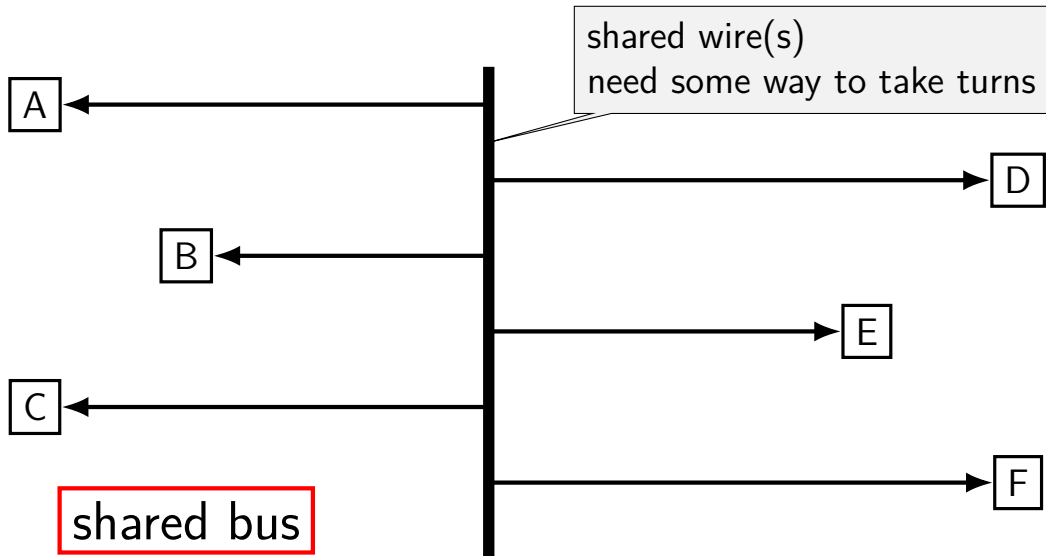
shared bus

shared bus, really?

common for parts of internals of computers (topic later)

model for wifi
    radio "channel" kinda similar to shared wire

how the early versions of Ethernet worked
    "vampire taps" physically attached to shared cable

shared bus, messages for who?

Figure 6-1: Data Link Layer Frame Format
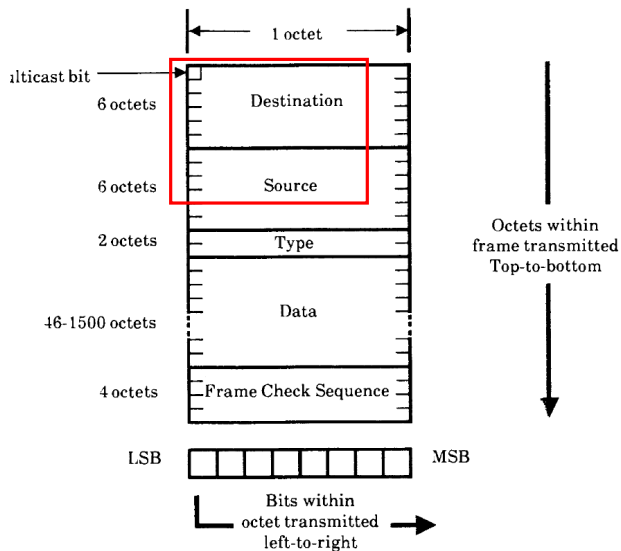
messages needs a 'header' to tell who it's to/from

everyone needs to filter out messages that aren't theirs

taking turns on shared bus?

token ring
    one machine has a 'token' = can send
    send special message to pass to another machine

free-for-all: collision detection + retry
    detect if you're transmitting when someone else is
    wait (usually randomized amount of time) and retry

coordinating machine transmits timeslots
    part of common cellphone design (TDMA: time division multiple access)

make bus support multiple transmitters?
    requires understanding how interference works
    another part of common cell phone design

connecting things
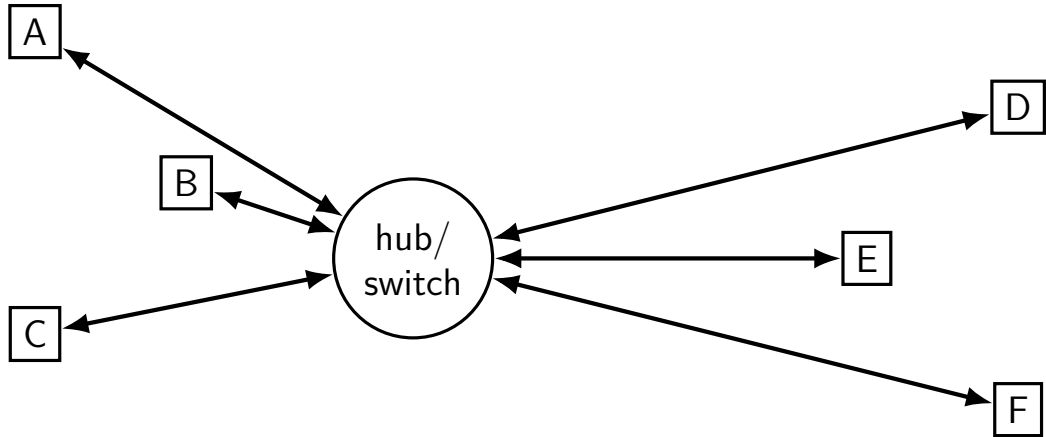
what does the hub do?

simple version:

> imitate shared bus: copy messages to everyone else
>
> something to handle two messages sent at once

less simple:

> read "header" on message + send to destination only
>
> requires some way to figure out destinations
>
> queue of messages waiting to be sent

connecting things
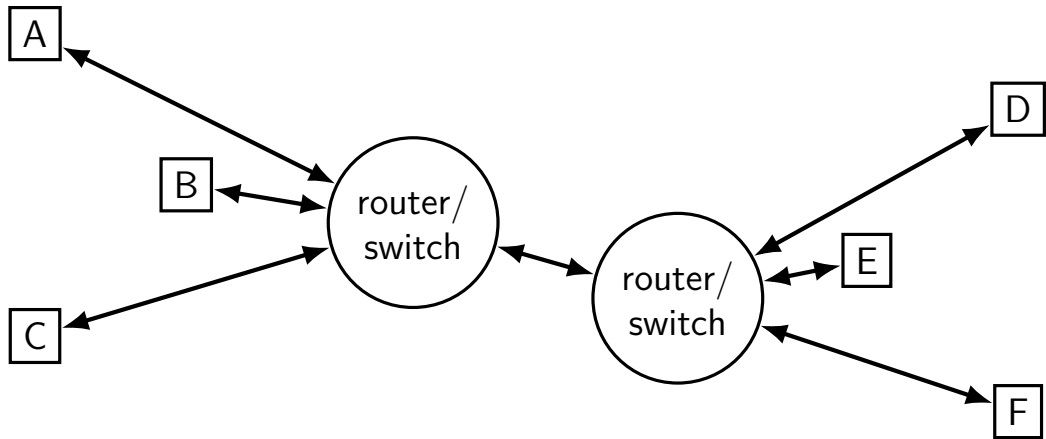
more complicated designs

hierarchies

networks of networks
    "internetworks"

so far still have single points of failure

connecting things
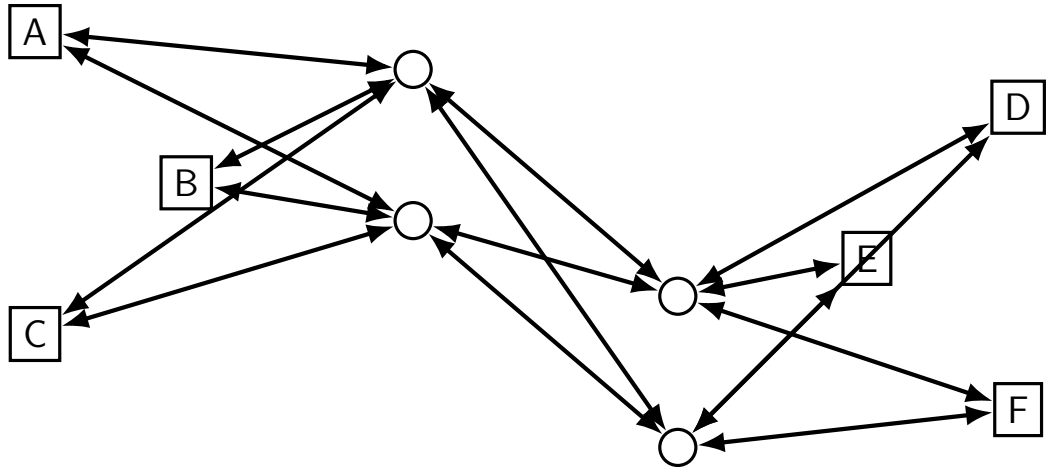
individual computers are networks

individual computers are (kinda) networks of...
>   processors
>   memories
>   I/O devices

so what topology (layout) do those networks have?

the "bus"
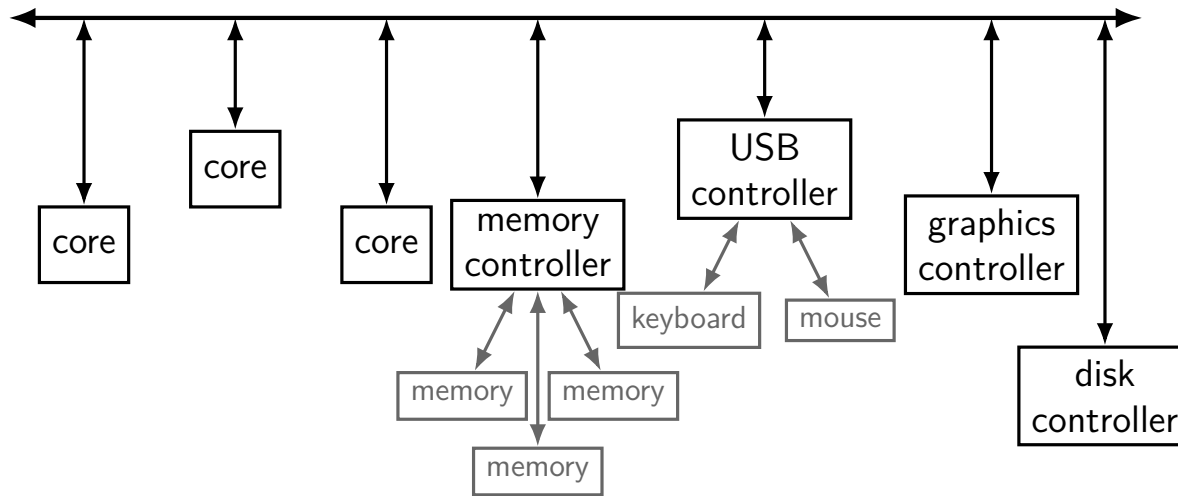
example: 80386 signal pins

name	purpose	
CLK2	clock for bus	timing
W/R#	write or read?	metadata
D/C#	data or control?	
M/IO#	memory or I/O?	
INTR	interrupt request	
...	other metadata signals	
BE0#-BE3#	(4) byte enable	address
A2-A31	(30) address bits	
DO-D31	(32) data signals	data

example: AMD EPYC (1 socket)
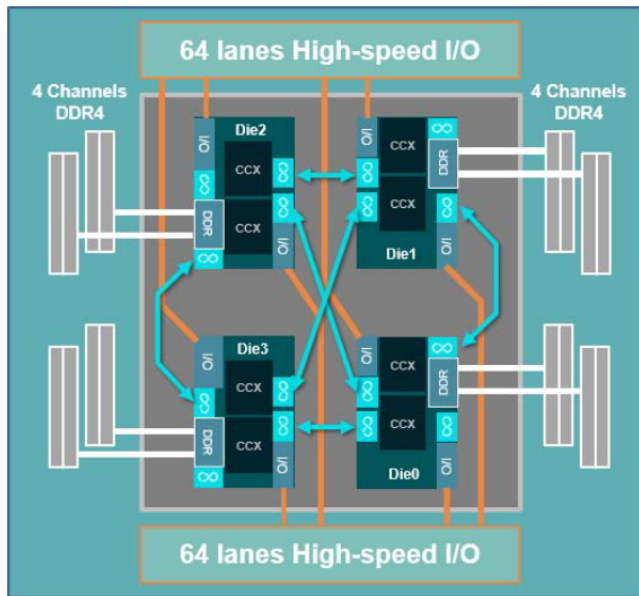
Fig. 21. Single-socket AMD EPYC™ system (SP3).

21

example: Intel Skylake-SP
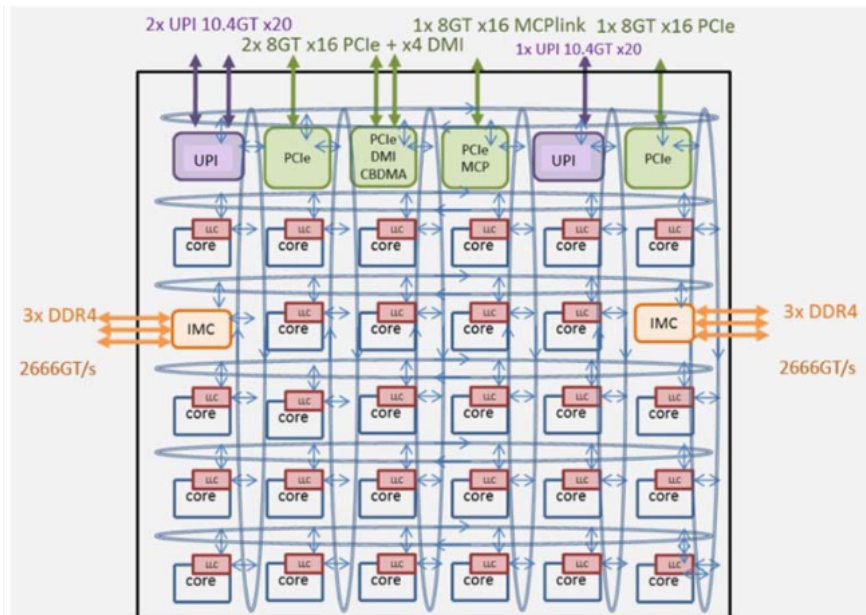
Figure from Tam et al, "SkyLake-SP: A 14nm 28-Core Xeon® Processor" (ISSCC 2018)   22
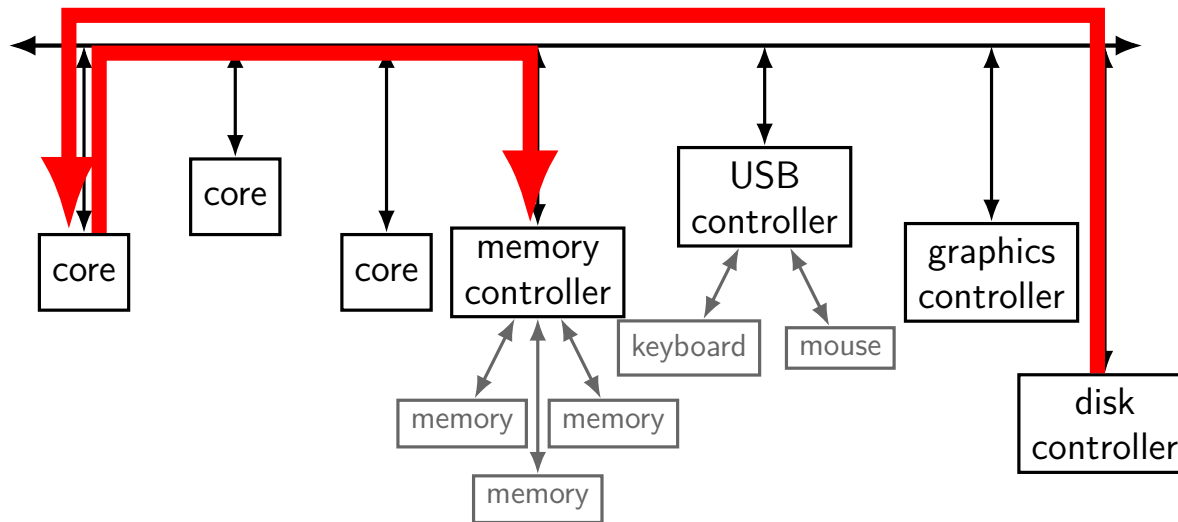
extra trips to CPU

extra trips to CPU

DMA
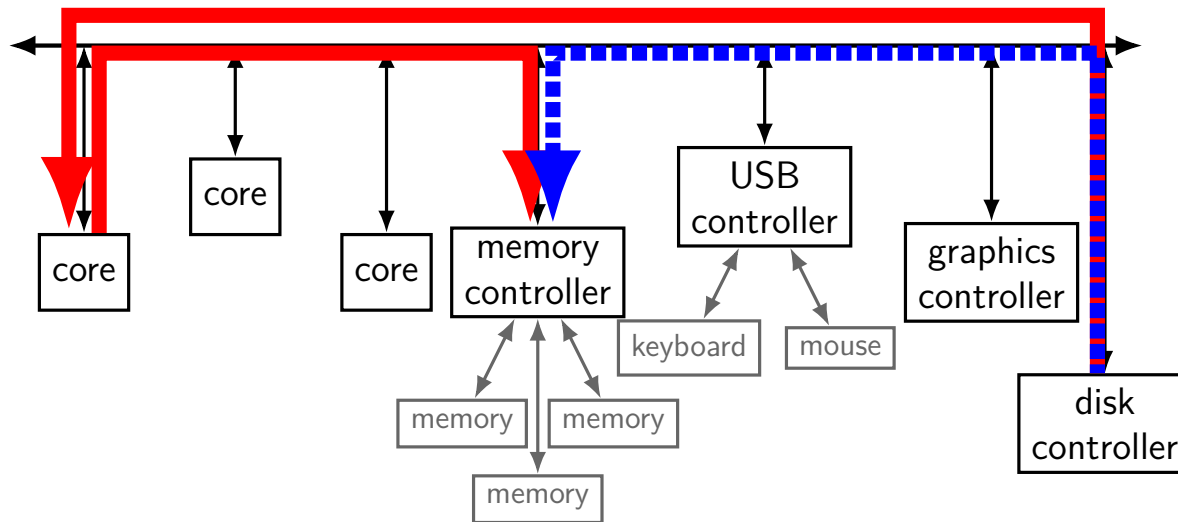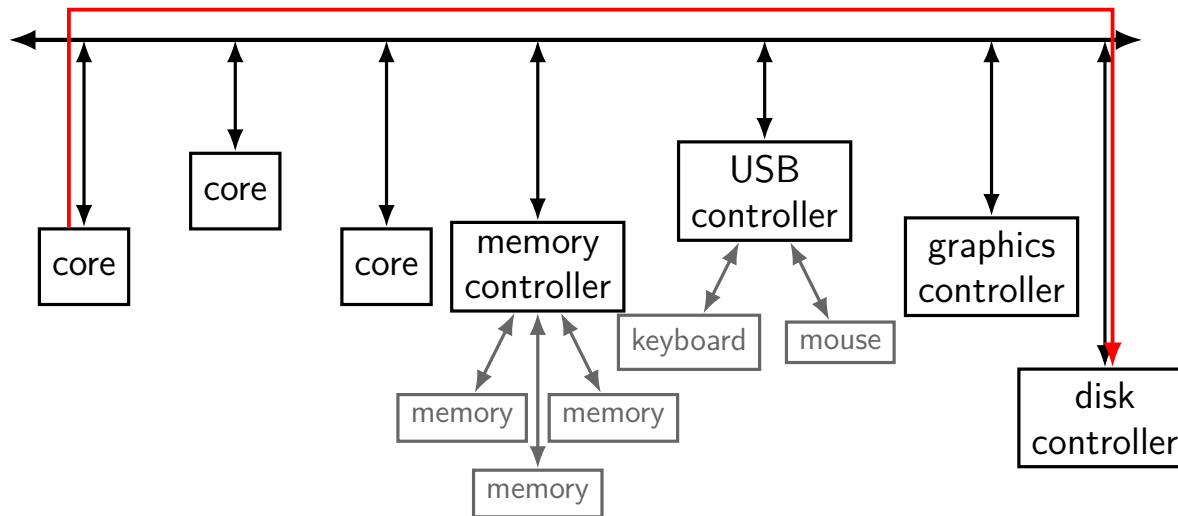
"place data at 0xABCD"

DMA

"place data at 0xABCD"

"write to 0xABCD"

core

core

core

memory controller

USB controller

keyboard

mouse

graphics controller

memory

memory

memory

disk controller

DMA

"place data at 0xABCD"

"write to 0xABCD"

core

core

core

memory controller

USB controller

keyboard

mouse

graphics controller

disk controller

"direct memory access"
controller communicates with memory itself

memory
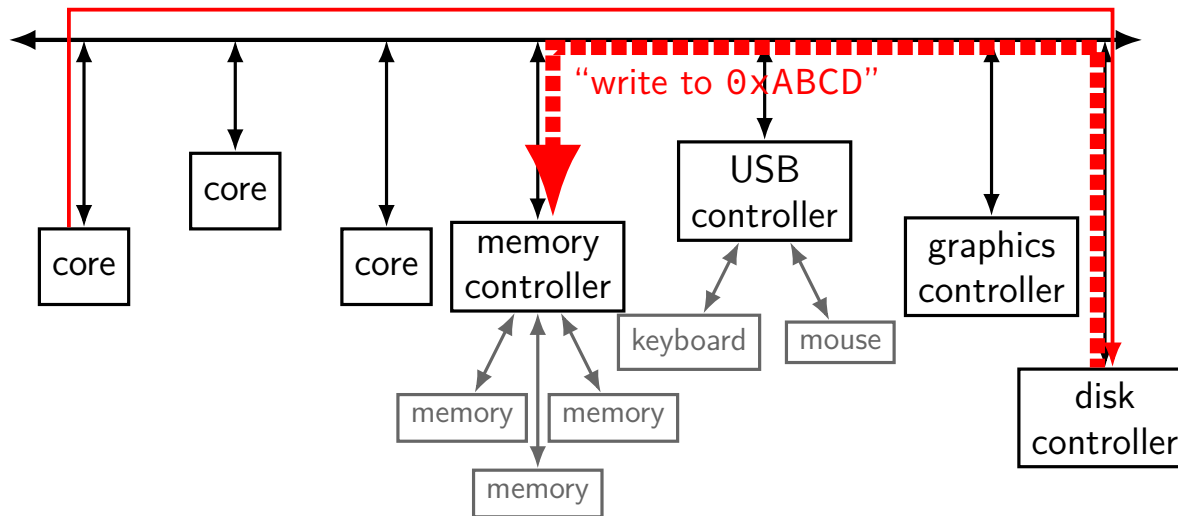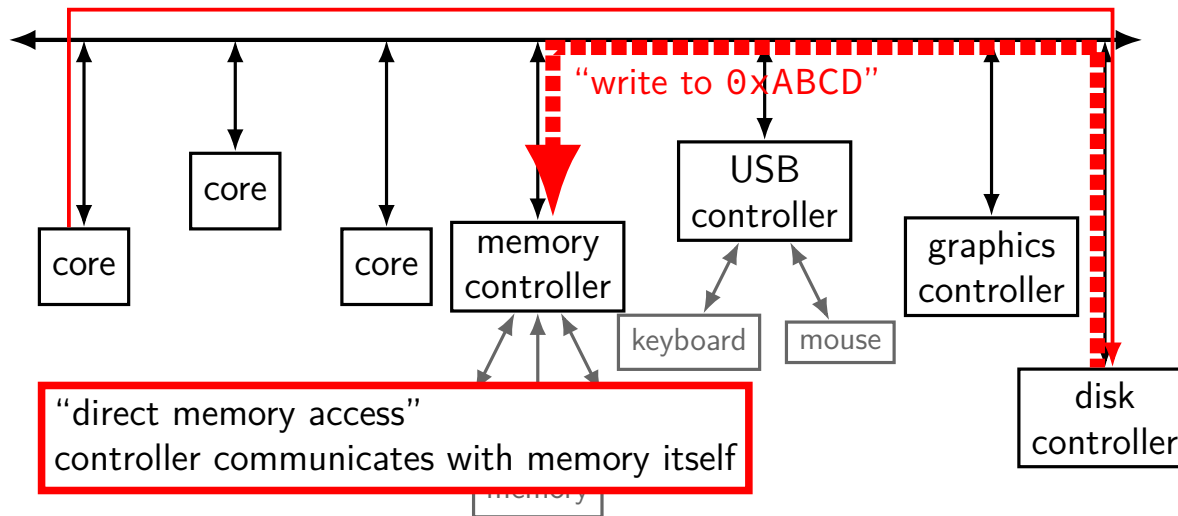
beyond pipelining: multiple issue

start more than one instruction/cycle

multiple parallel pipelines; many-input/output register file

hazard handling much more complex

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

beyond pipelining: out-of-order

find later instructions to do instead of stalling

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

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

cycle #	0	1	2	3	4	5	6	7	8	9	10	11
mov 0(%rbx), %r8	F	D	R	I	E	M	M	M	W	C		
sub %r8, %r9		F	D	R				I	E	W	C	
add %r10, %r11			F	D	R	I	E	W			C	
xor %r12, %r13				F	D	R	I	E	W			C
...												

29
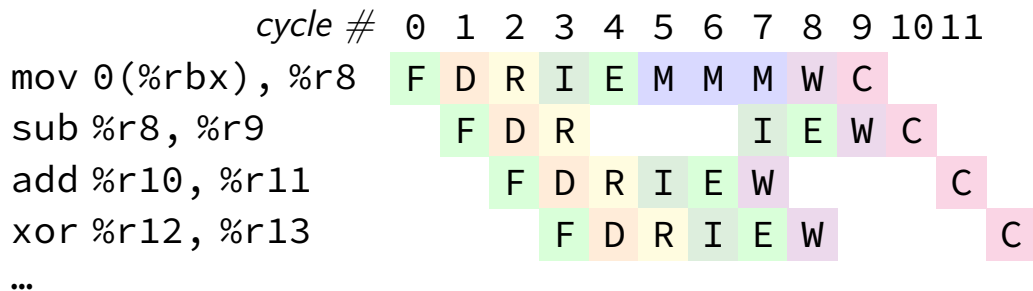
interlude: real CPUs

modern CPUs:

execute multiple instructions at once

execute instructions out of order — whenever values available

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:
    value in last stage may not be most up-to-date
    older value may be written back before newer value?

problems for branch prediction:
    mispredicted instructions may complete execution before squashing

which instructions to dispatch?
    how to quickly find instructions that are ready?

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:
    value in last stage may not be most up-to-date
    older value may be written back before newer value?

problems for branch prediction:
    mispredicted instructions may complete execution before squashing

which instructions to dispatch?
    how to quickly find instructions that are ready?

read-after-write examples (1)

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

normal pipeline: two options for %r8?

choose the one from *earliest stage*

because it's from the most recent instruction

read-after-write examples (1)

out-of-order execution:
%r8 from earliest stage might be from *delayed instruction*
can't use same forwarding logic

addq				F	D	E	M	W		
addq %r11, %r8				F	D	E	M	W		
addq %r12, %r8					F	D	E	M	W	

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

33
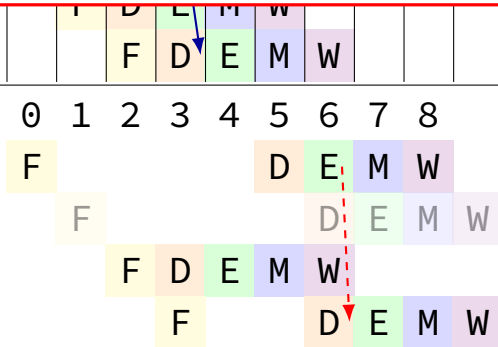
register version tracking

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	W
`movq %r8, (%rax)`		F									D E M W
`movq %r11, %r8`			F	D	E	M	W			
`movq %r8, 8(%rax)`			F							D E M W
`movq $100, %r8`			F	D	E	M	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	
`movq %r8, (%rax)`		F								D	E M W
`movq %r11, %r8`			F	D	E	M	W				
`movq %r8, 8(%rax)`			F					D	E	M	W
`movq $100, %r8`				F	D	E	M	W			
`addq %r13, %r8`				F						D	E M W
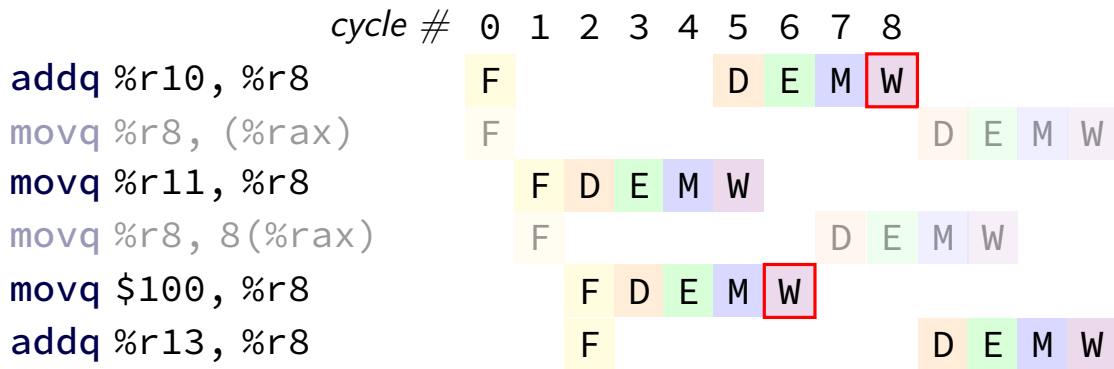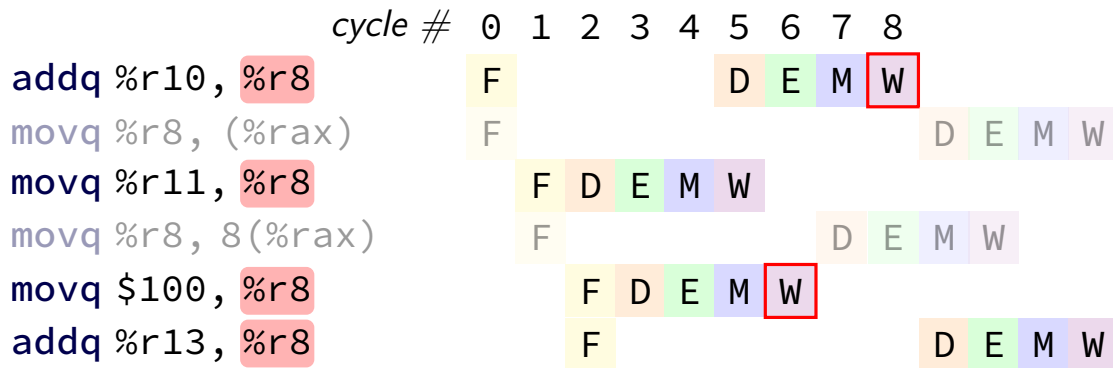
out-of-order execution:
if we don't do something, newest value could be overwritten!

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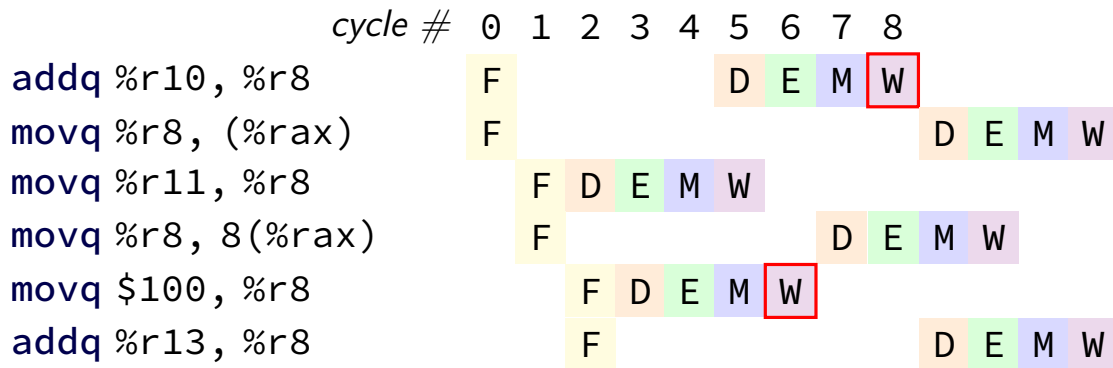
two instructions that haven't been started
could need *different versions* of %r8!

write-after-write example

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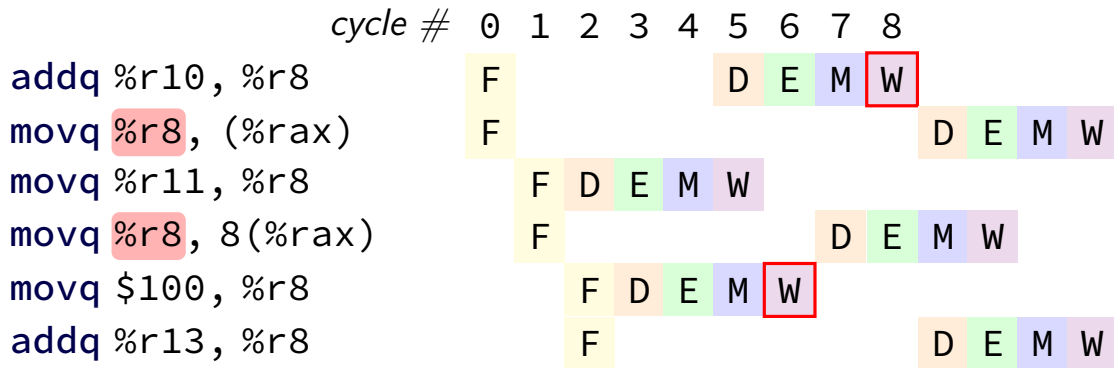
keeping multiple versions

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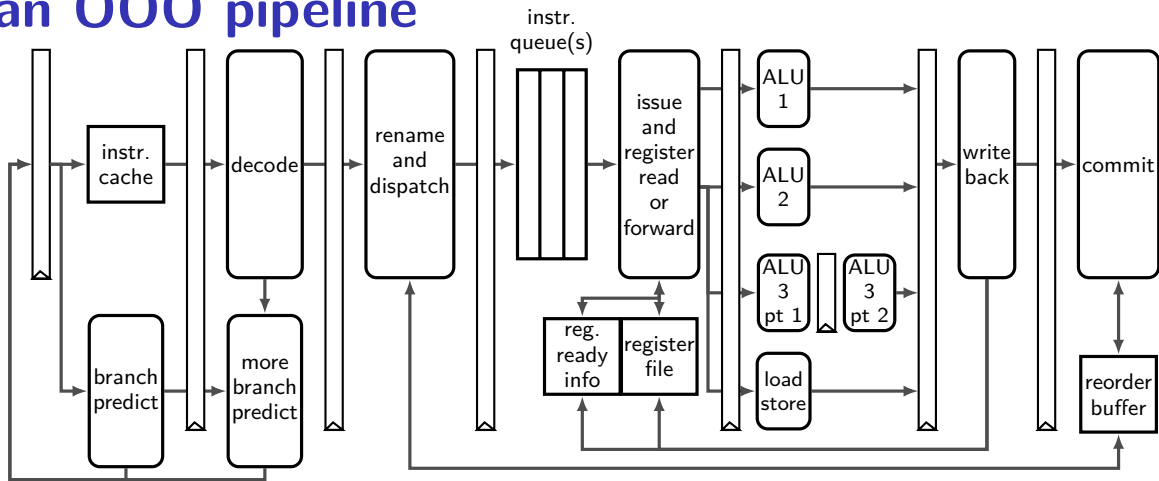
an OOO pipeline
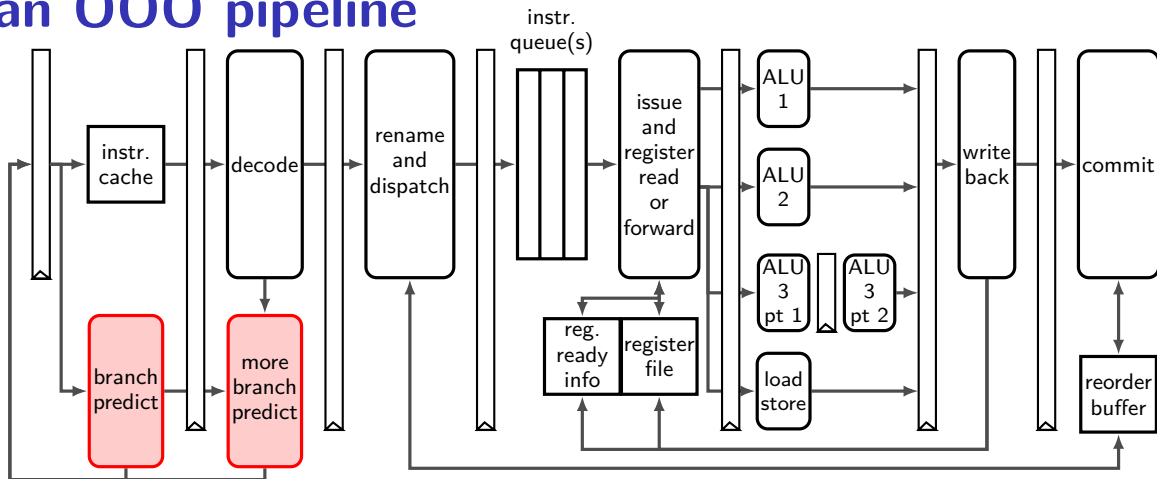
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

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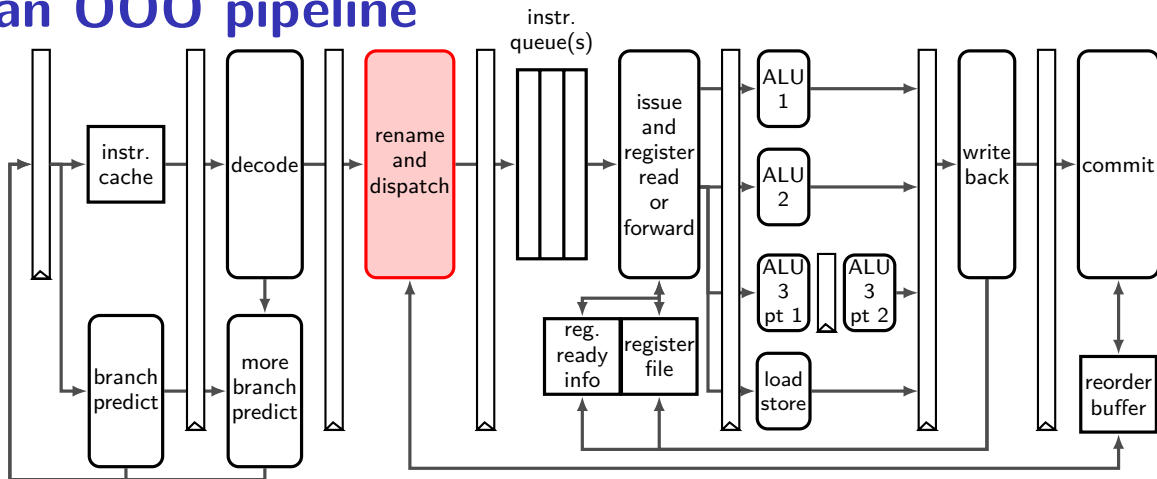
register renaming done here
stage needs to keep mapping from architectural to physical names

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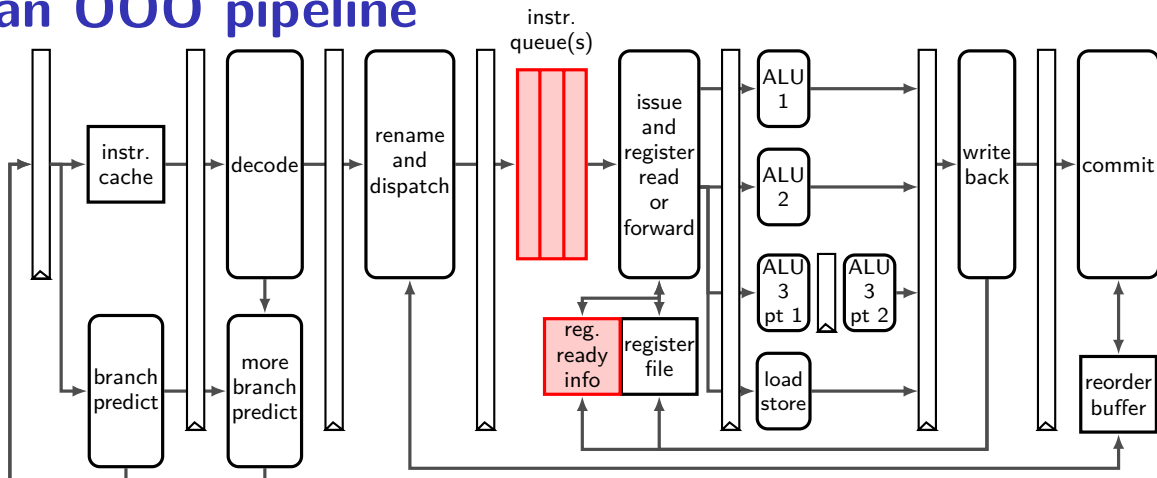
instruction queue holds pending renamed instructions
combined with register-ready info to *issue* instructions
(issue = start executing)

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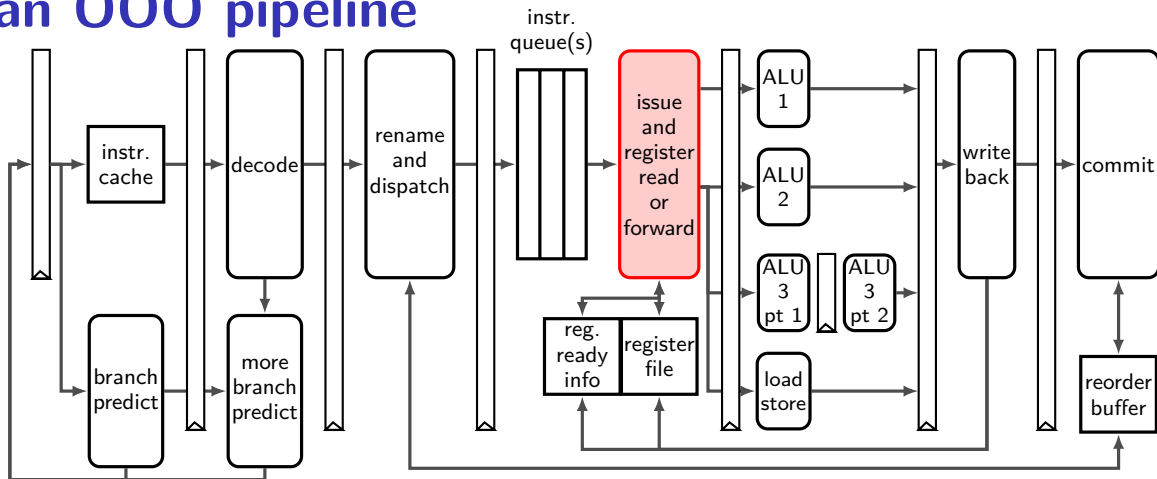
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

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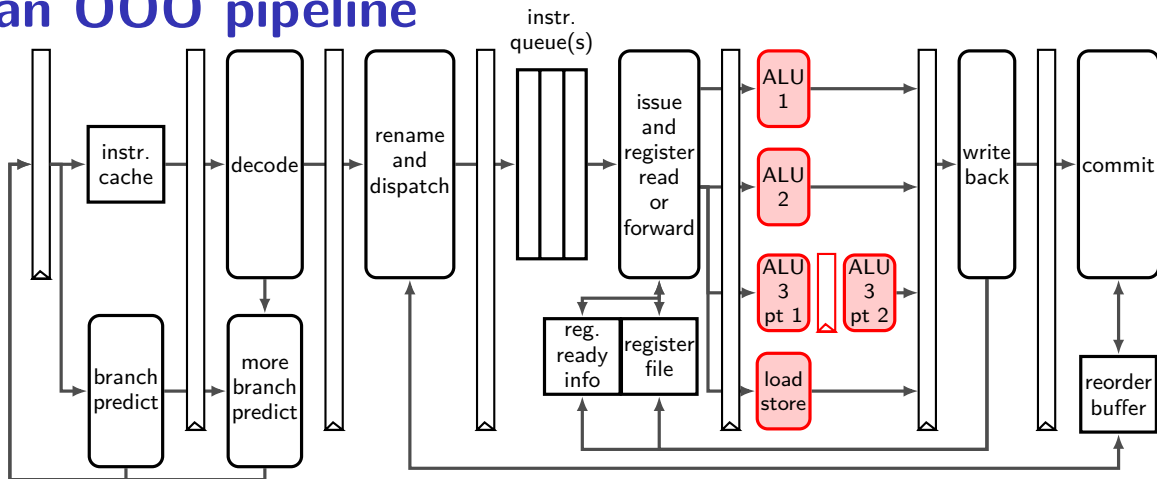
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, …)

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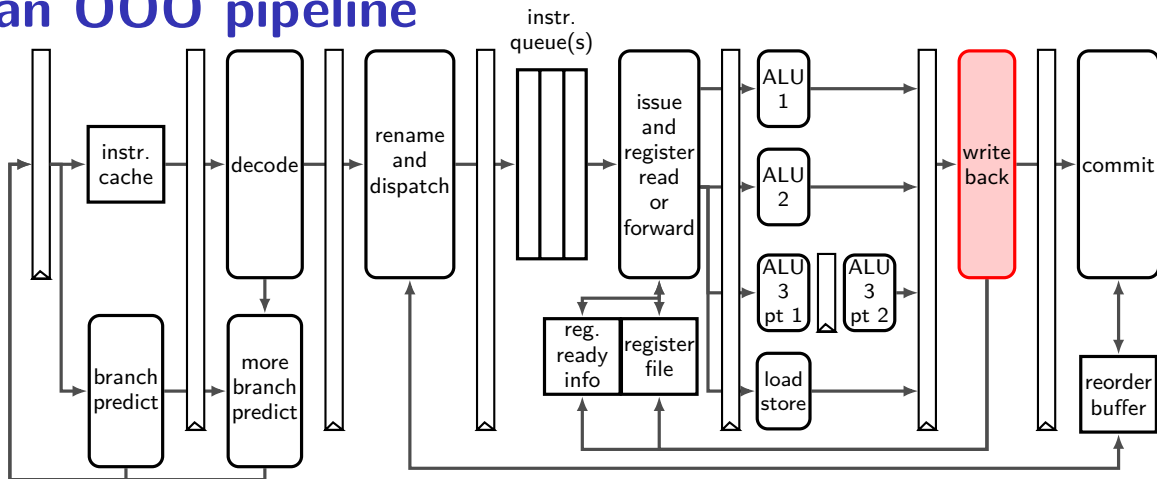
writeback results to physical registers
register file: typically support writing 3+ registers at a time

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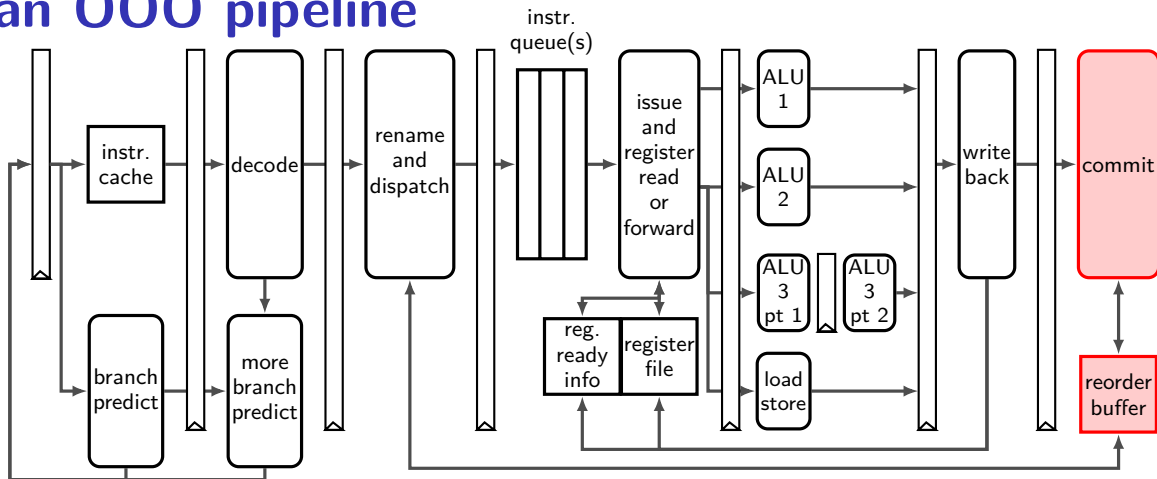
new commit (sometimes *retire*) stage finalizes instruction figures out when physical registers can be reused again

an OOO pipeline

instr. queue(s)

instr. cache

decode

rename and dispatch

issue and register read or forward

ALU 1

ALU 2

ALU 3 pt 1

ALU 3 pt 2

write back

commit

branch predict

more branch predict

reg. ready info

register file

load store
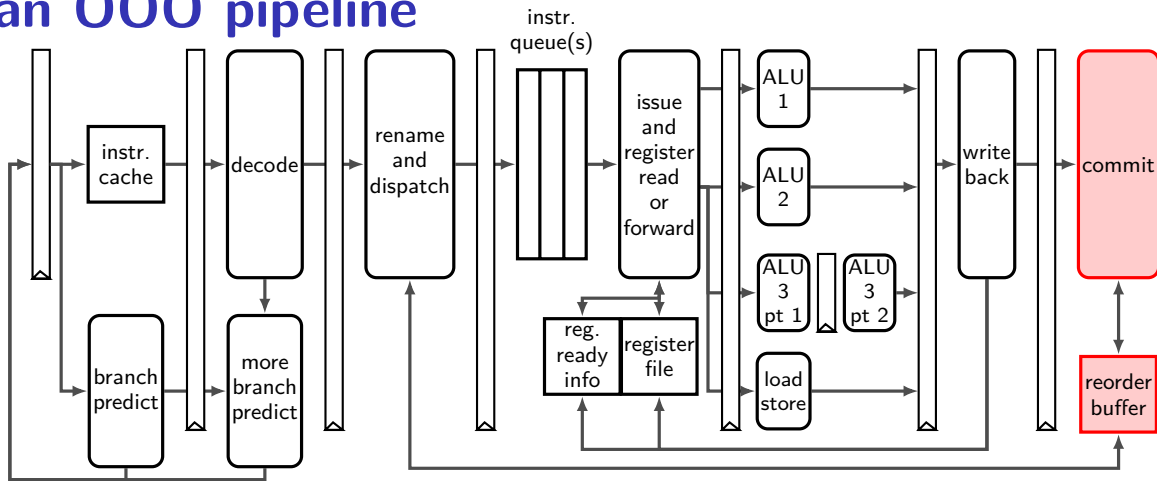
reorder buffer

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

an OOO pipeline diagram

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

an OOO pipeline diagram

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

fetch instructions in order,
several per cycle
unless instruction queue full

41

an OOO pipeline diagram

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

issue instructions
(to "execution units")
when operands ready

41

an OOO pipeline diagram

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

> commit instructions in order
> waiting until next complete

41

1-cycle fetch?

assumption so far:

1 cycle to fetch instruction $+$ identify if jmp, etc.

often not really practical

especially if:
    complex machine code format
    many pipeline stages
    more complex instruction cache
    (future idea) fetching $2+$ instructions/cycle

branch target buffer

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

    will happen in more complex pipelines

what if we can't decode that there's a RET, CALL, etc. fast?

BTB: cache for branch targets

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

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

BTB: cache for branch targets

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

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

BTB: cache for branch targets

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

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

indirect branch prediction

`jmp *%rax` or `jmp *(%rax, %rcx, 8)`

BTB can provide a prediction

but can do better with more context

example—predict based on other recent computed jumps
  good for polymophic method calls

table lookup with Hash(last few jmps)
instead of Hash(this jmp)

an OOO pipeline
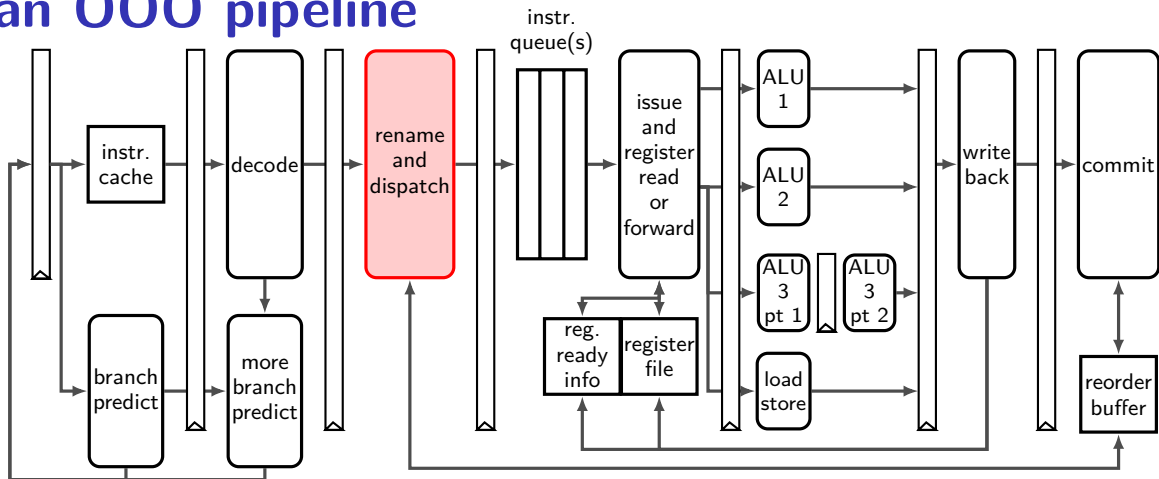
register renaming

rename *architectural registers* to *physical registers*
    architectural = part of instruction set architecture

different name for each version of architectural register

register renaming state

original   renamed
```
add %r10, %r8   …
add %r11, %r8   …
add %r12, %r8   …
```

arch → phys register map

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

free reg list

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

register renaming state

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

renamed

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

arch → phys register map	
%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

%x18
%x20
%x21
%x23
%x24
…

register renaming state

original

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

renamed

list of available physical registers
added to as instructions finish

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   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 example (1)

original   renamed

```
add %r10, %r8    add %x19, %x13 → %x18
add %r11, %r8
add %r12, %r8
```

arch → phys register map

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

free reg list

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

register renaming example (1)

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

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

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	renamed
`addq %r10, %r8`	`addq %x19, %x13 → %x18`
`movq %r8, (%rax)`	
`subq %r8, %r11`	
`movq 8(%r11), %r11`	
`movq $100, %r8`	
`addq %r11, %r8`	

arch → phys register map

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

free regs

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

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13 → %x18`
`movq %r8, (%rax)`	`movq %x18, (%x04) → (memory)`
`subq %r8, %r11`	
`movq 8(%r11), %r11`	
`movq $100, %r8`	
`addq %r11, %r8`	

arch → phys register map

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

free
regs

%x18
%x20
%x21
%x23
%x24
•••

register renaming example (2)

	original		renamed
addq	%r10, %r8	addq	%x19, %x13 → %x18
movq	%r8, (%rax)	movq	%x18, (%x04) → (memory)
subq	%r8, %r11		
movq	8(%r11), %r11		
movq	$100, %r8		
addq	%r11, %r8		

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
...

50

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13 → %x18`
`movq %r8, (%rax)`	`movq %x18, (%x04) → (memory)`
`subq %r8, %r11`	`subq %x18, %x07 → %x20`
`movq 8(%r11), %r11`	
`movq $100, %r8`	
`addq %r11, %r8`	

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`
`movq %r8, (%rax)`	`movq %x18, (%x04) → (memory)`
`subq %r8, %r11`	`subq %x18, %x07 → %x20`
`movq 8(%r11), %r11`	`movq 8(%x20), (memory) → %x21`
`movq $100, %r8`	
`addq %r11, %r8`	

arch → phys register map

%rax	%x04
%rcx	%x09
•••	•••
%r8	%x̶1̶3̶ %x18
%r9	%x17
%r10	%x19
%r11	%x̶0̶7̶ %x̶2̶0̶ %x21
%r12	%x05
%r13	%x02
•••	•••

free regs

%x̶1̶8̶
%x̶2̶0̶
%x̶2̶1̶
%x23
%x24
•••

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13 → %x18`
`movq %r8, (%rax)`	`movq %x18, (%x04) → (memory)`
`subq %r8, %r11`	`subq %x18, %x07 → %x20`
`movq 8(%r11), %r11`	`movq 8(%x20), (memory) → %x21`
`movq $100, %r8`	`movq $100 → %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 → %x18
	movq %r8, (%rax)		movq %x18, (%x04) → (memory)
	subq %r8, %r11		subq %x18, %x07 → %x20
	movq 8(%r11), %r11		movq 8(%x20), (memory) → %x21
	movq $100, %r8		movq $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	renamed

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

arch → phys

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

free regs

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

an OOO pipeline

instruction queue and dispatch

scoreboard

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

instruction queue

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

execution unit
ALU 1
ALU 2

...

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

scoreboard

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

instruction queue

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

execution unit	cycle# 1
ALU 1	1
ALU 2	

...

instruction queue and dispatch

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
...	...

instruction queue

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

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

53

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	—	—	—	

53

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
…	…

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

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
...	...

instruction queue

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

execution unit	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

↑

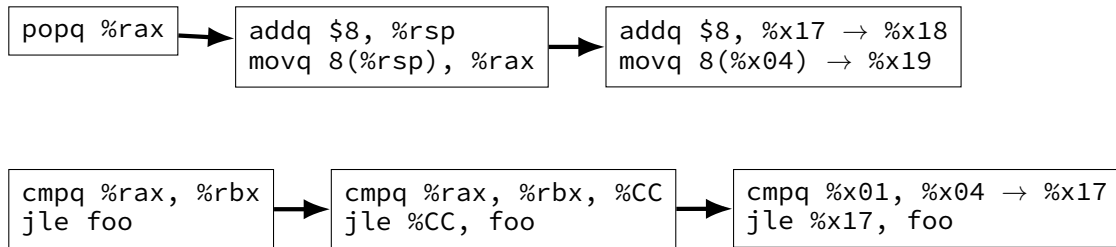assume

1 cycle/access

register renaming: missing pieces

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

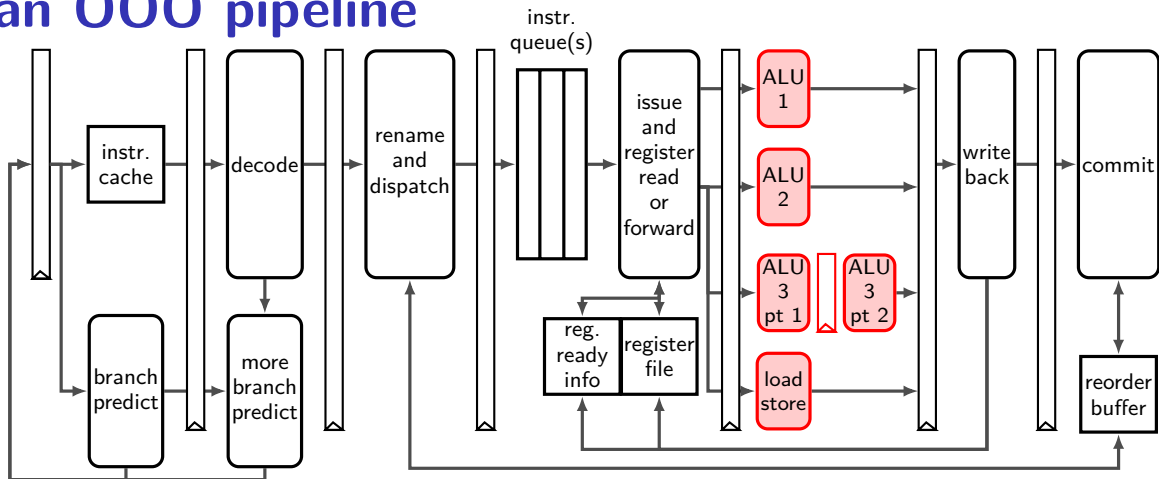one solution: translate to intructions with additional register parameters
    making %rsp explicit parameter
    turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones

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

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

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)

input values $\longrightarrow$ | ALU (stage 1) | | ALU (stage 2) | | ALU (stage 3) | $\longrightarrow$ output values
(one/cycle)   (one/cycle)
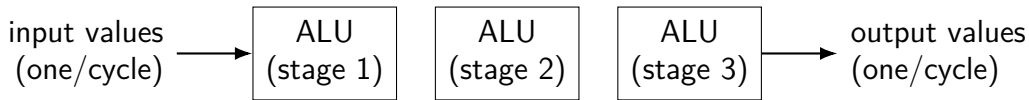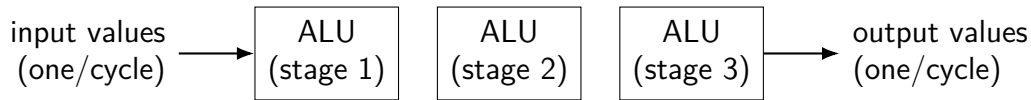
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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

input values $\longrightarrow$ | ALU (stage 1) | | ALU (stage 2) | | ALU (stage 3) | $\longrightarrow$ output values (one/cycle)
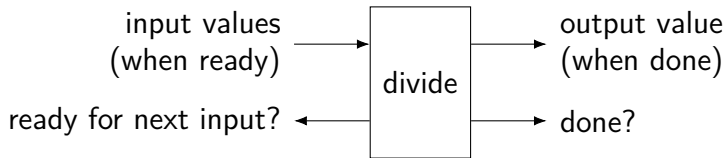(one/cycle)

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, ...
…	…

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, ...
…	…

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, ...
...	...

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, ...
…	…

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

59

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, ...
…	…

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, ...
…	…

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, ...
...	...

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, ...
…	…

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, ...
…	…

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	

59

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, ...~~
...	...

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			

59

OOO limitations

can't always find instructions to run
> plenty of instructions, but all depend on unfinished ones
> programmer can adjust program to help this

need to track all uncommitted instructions
> can only go so far ahead
> e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)
> e.g. Intel Skylake: up to approx. 16 cycles (v. 2 for simple pipelined CPU)

OOO limitations

can't always find instructions to run

    plenty of instructions, but all depend on unfinished ones

    programmer can adjust program to help this

need to track all uncommitted instructions

    can only go so far ahead

    e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

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

some performance examples

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

about 30B instructions
my desktop: approx 2.65 sec

```
example2:
    movq $10000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```
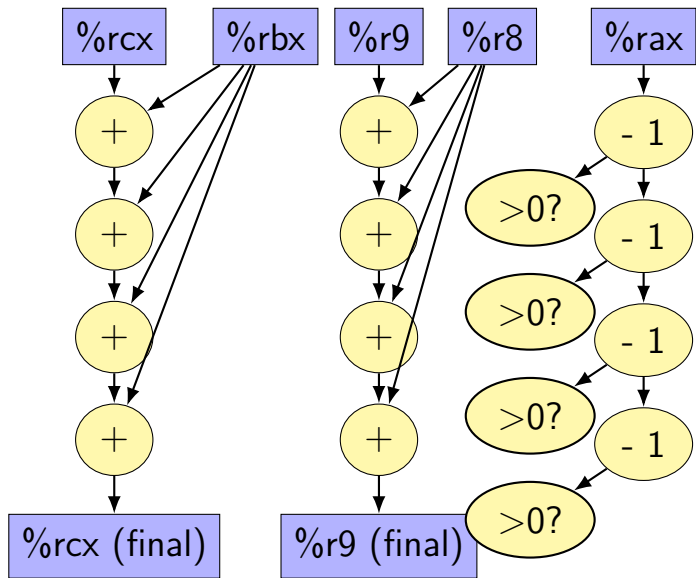
about 40B instructions
my desktop: approx 2.65 sec

some performance examples

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

about 30B instructions
my desktop: approx 2.65 sec

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

about 40B instructions
my desktop: approx 2.65 sec

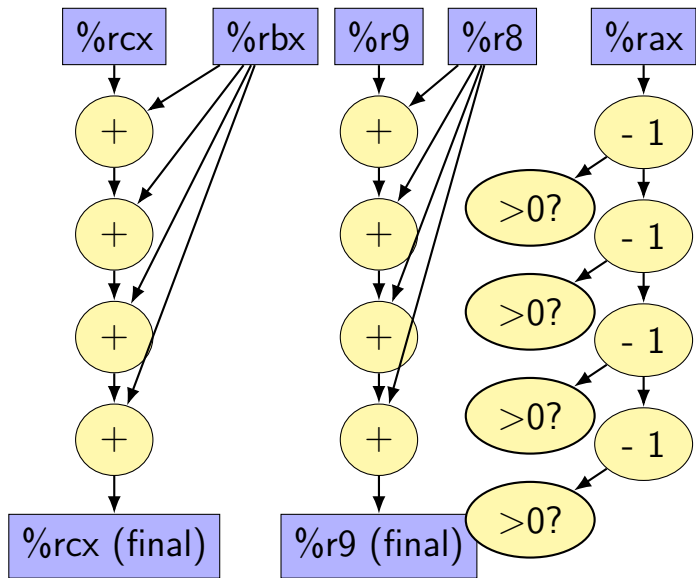# data flow model and limits (1)


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

data flow model and limits (1)

each yellow box = instruction

arrows = dependences

instructions only executed when dependencies ready

reassociation

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

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

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```
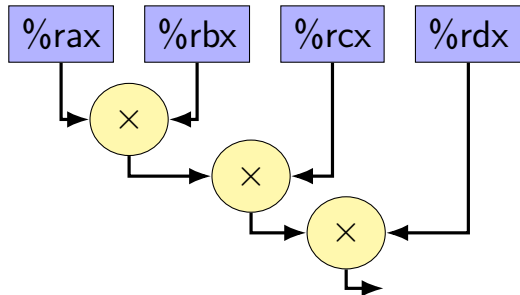
$(a \times b) \times (c \times d)$

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

reassociation

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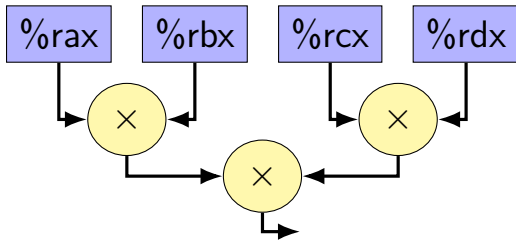to compute?

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

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

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

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

Intel Skylake OOO design

2015 Intel design — codename 'Skylake'

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

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

2 load functional units
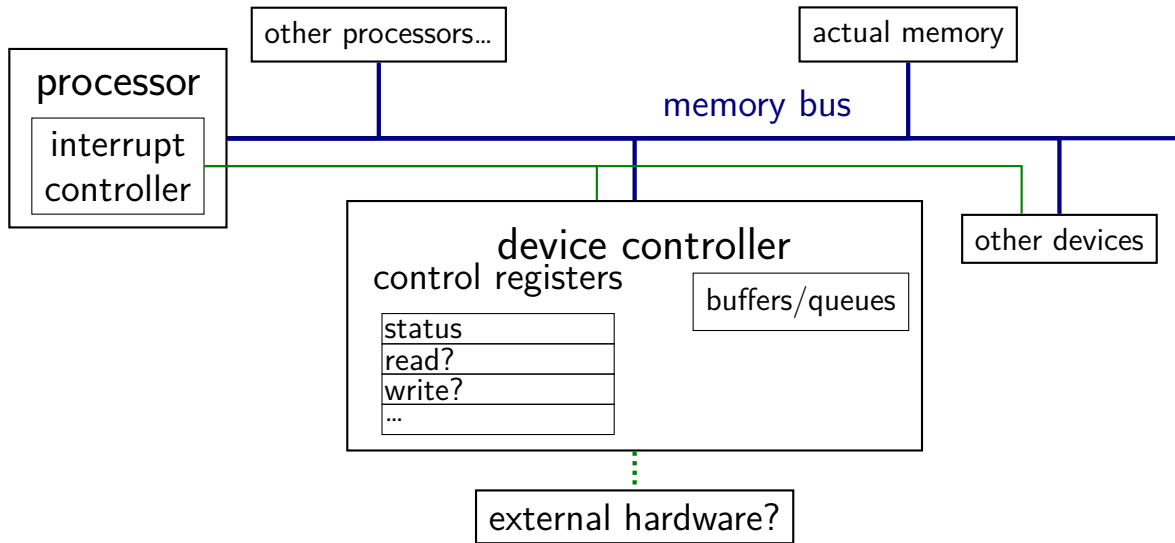   but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer
    determines how far ahead branch mispredictions, etc. can happen

backup slides

connecting devices

connecting devices

control registers have memory addresses
looks like write to memory
actually changes value in device controller

pr...

in...
controller

...ry bus

actual memory

other devices

device controller
control registers

buffers/queues

0x80004800:	status
0x80004808:	read?
0x80004810:	write?
…:	…
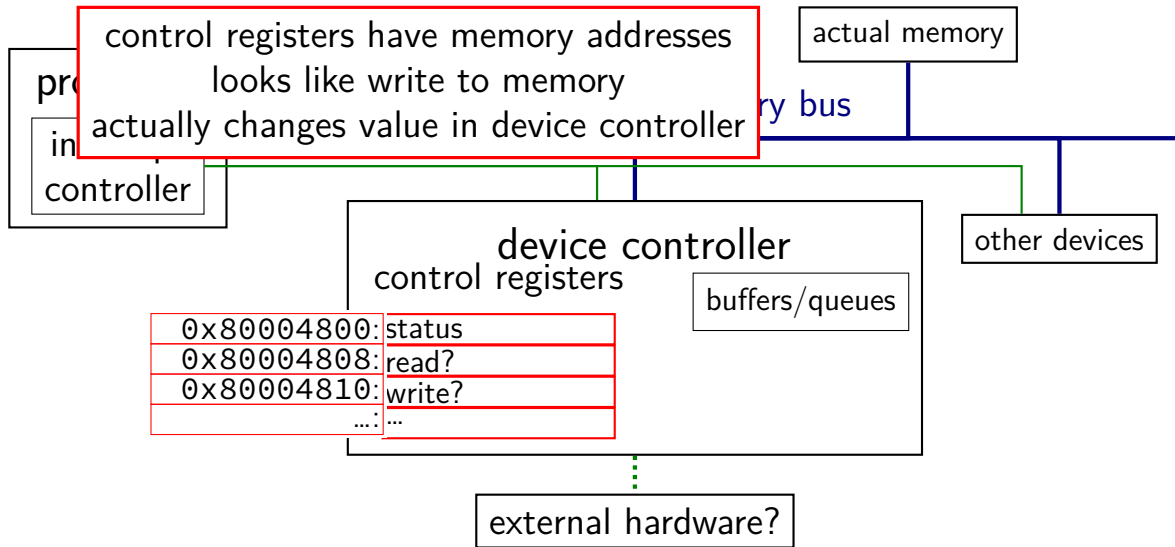
external hardware?

connecting devices
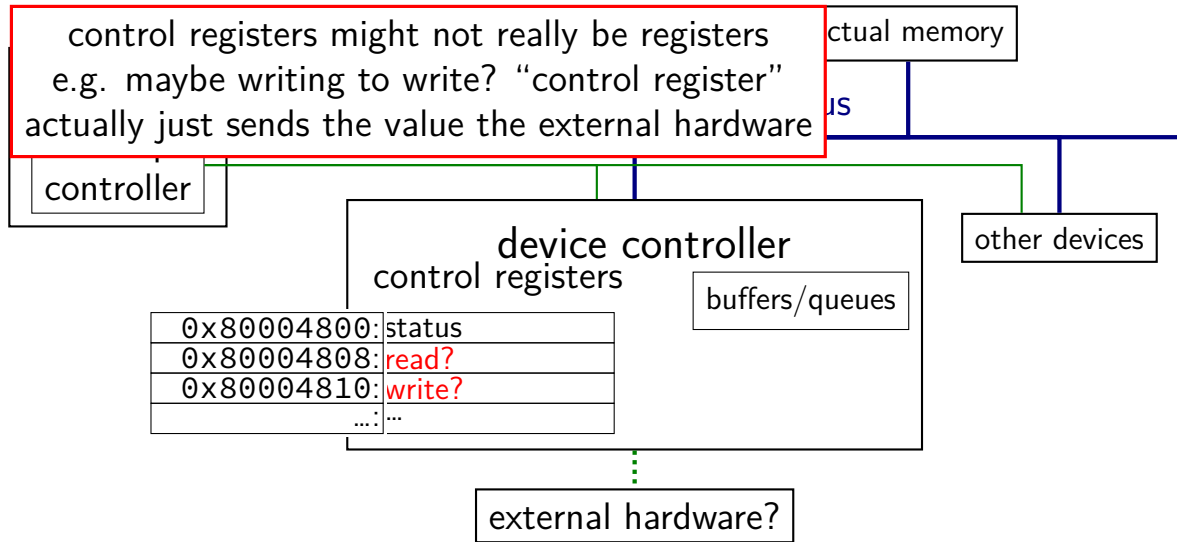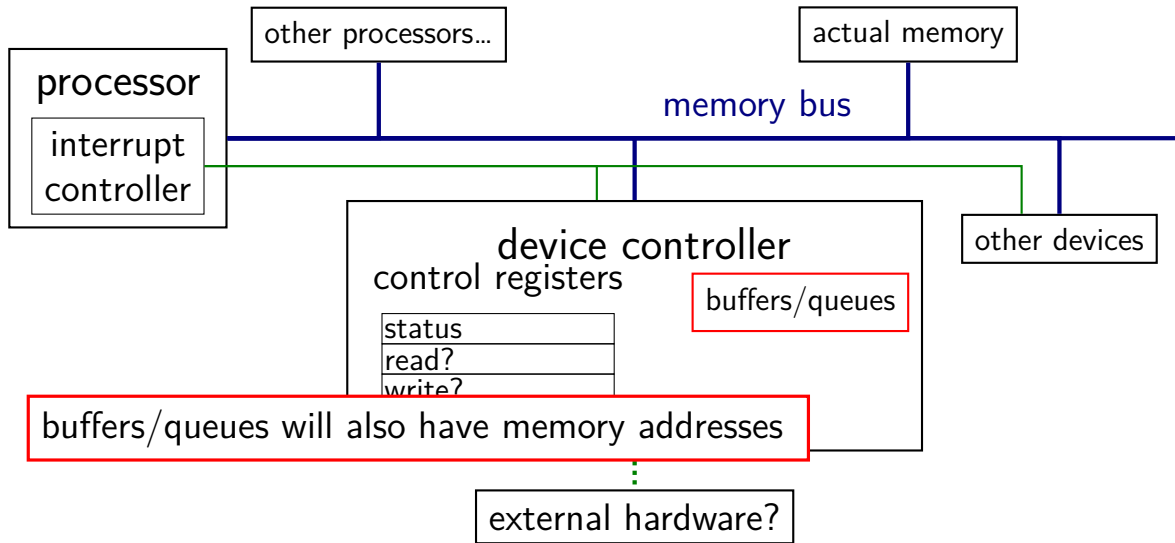
control registers might not really be registers
e.g. maybe writing to write? "control register"
actually just sends the value the external hardware

ctual memory

us

controller

device controller
control registers

buffers/queues

0x80004800:	status
0x80004808:	read?
0x80004810:	write?
…:	…

other devices

external hardware?

connecting devices

processor

interrupt controller

other processors...

actual memory

memory bus

device controller

control registers

status
read?
write?

buffers/queues

other devices

buffers/queues will also have memory addresses

external hardware?

connecting devices

processor

interrupt controller

other processors…

actual memory

memory bus

device controller
control registers

buffers/queues

status
read?
~~write?~~

other devices

way to send "please interrupt" signal
component of processor decides when to handle
(deals with ordering, interrupt disabling,
which of several processors handles it, …, etc.)