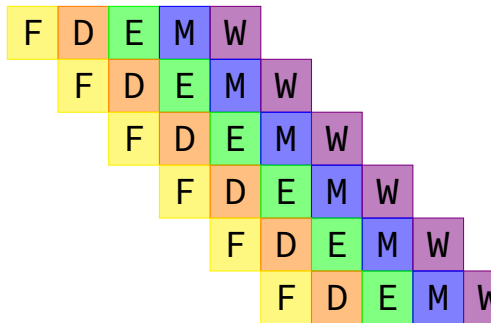


jXX: stalling?

```
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)
...
```

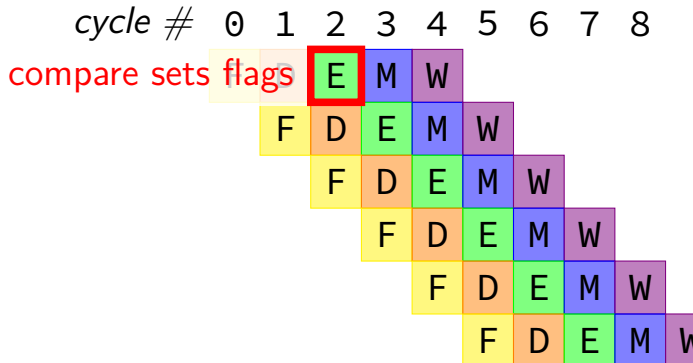
cycle # 0 1 2 3 4 5 6 7 8



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
```

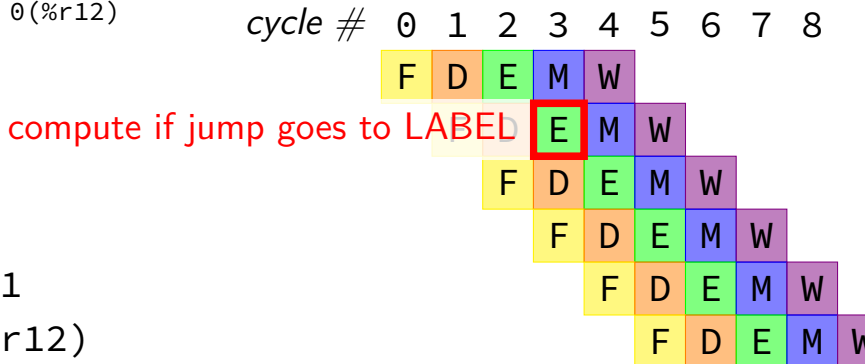
```
(do nothing)
```

```
(do nothing)
```

```
xorq %r10, %r11
```

```
movq %r11, 0(%r12)
```

```
...
```

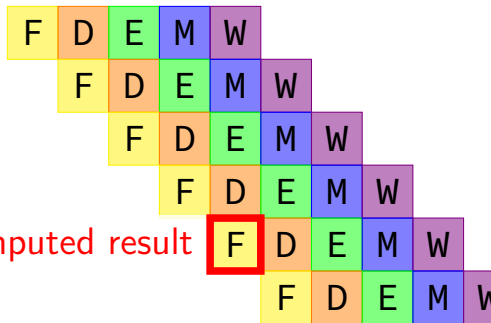


jXX: stalling?

```
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)
...
```

cycle # 0 1 2 3 4 5 6 7 8



use computed result

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

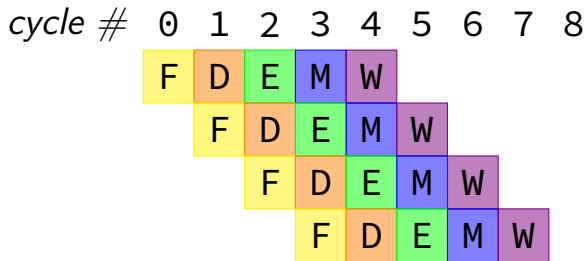
right: 2 cycles faster!; wrong: undo guess before too late

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

cycle # 0 1 2 3 4 5 6 7 8

cmpq %r8, %r9

jne LABEL

xorq %r10, %r11

(inserted nop)

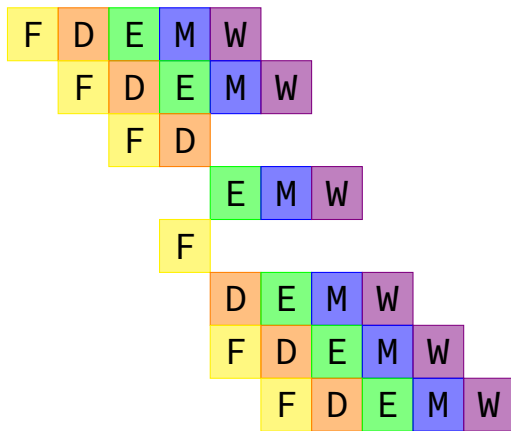
movq %r11, 0(%r12)

(inserted nop)

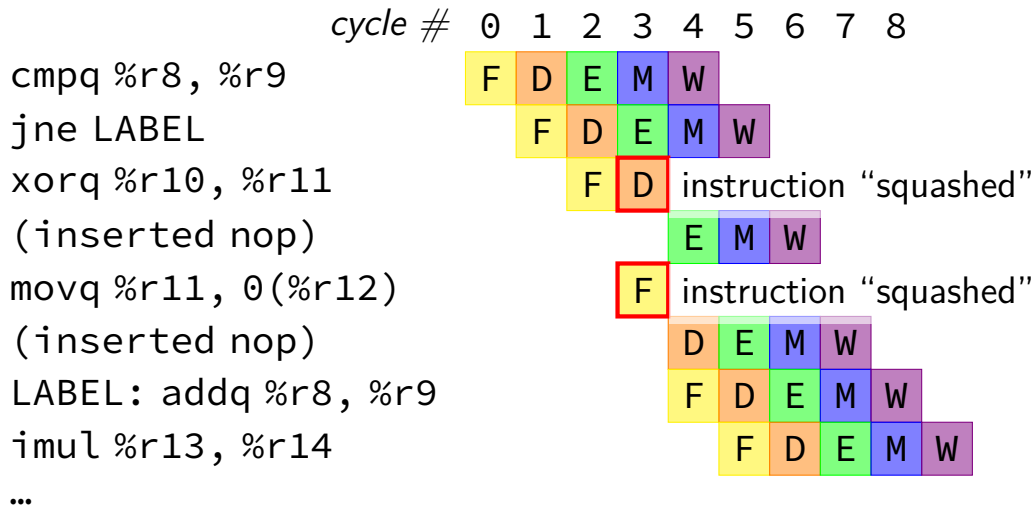
LABEL: addq %r8, %r9

imul %r13, %r14

...



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 not-taken)	cycles (stall)
taken jXX	3%	3	3
non-taken jXX	5%	1	3
others	92%	1*	1*

performance

hypothetical instruction mix

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

exercise: predict+forward (1)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

cmpq %r9, %r10

jle foo (taken)

...

...

...

foo: andq %r9, %r8

if jle is *correctly predicted*:

in cmpq, %r9 is _____ addq.

in andq, %r9 is _____ addq.

A: not forwarded from

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



exercise: predict+forward (1)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

cmpq %r9, %r10

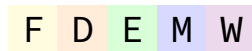
jle foo (taken)

...

...

...

foo: andq %r9, %r8



if jle is *correctly predicted*:

in cmpq, %r9 is _____ addq.

in andq, %r9 is _____ addq.

A: not forwarded from

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

exercise: predict+forward (2)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

cmpq %r9, %r10

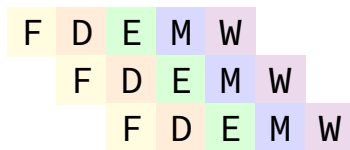
jle foo (taken)

...

...

...

foo: andq %r9, %r8



if jle is *mispredicted* + resolved after jle's execute:

in cmpq, %r9 is _____ addq.

in andq, %r9 is _____ addq.

A: not forwarded from

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

exercise: predict+forward (2)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

cmpq %r9, %r10

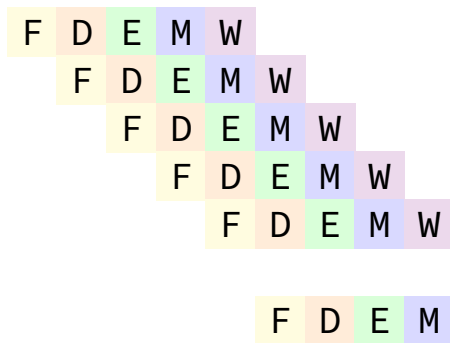
jle foo (taken)

(mispredicted)

(mispredicted)

...

foo: andq %r9, %r8



if jle is *mispredicted* + resolved after jle's execute:

in cmpq, %r9 is _____ addq.

in andq, %r9 is _____ addq.

A: not forwarded from

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

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**

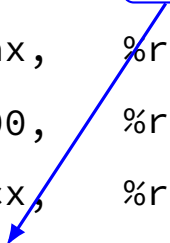
where are dependencies?

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

The diagram illustrates data dependencies between instructions. A blue arrow points from the `%rbx` operand in the first instruction (`addq %rax, %rbx`) to the `%rbx` operand in the fifth instruction (`addq %rbx, %r10`). A red arrow points from the `%rcx` operand in the third instruction (`movq $100, %rcx`) to the `%rcx` operand in the fourth instruction (`addq %rcx, %r10`). The `%rbx` and `%rcx` operands in the first, third, and fifth instructions are enclosed in blue boxes, while the `%rcx` operands in the third and fourth instructions are enclosed in red boxes.

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

```
graph TD; I1[addq %rax, %rbx] -- blue --> I4[addq %rcx, %r10]; I3[movq $100, %rcx] -- red --> I4; I4 -- red --> I5[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

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

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

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

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

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?

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

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
    jg for_loop_top // if (edi > 0) goto for_loop_top
    ret
```

suppose `%edi = 3` (initially)

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

how many mispredictions for `je`? for `jl`?

predict: repeat last

PC of branch

0x40042A

hash function

index *prediction/
last result?*

0 taken (1)

1 not taken (0)

2 taken (1)

3 taken (1)

...

14 not taken (0)

15 taken (1)

predict: repeat last

PC of branch

0x40042A

hash function

index *prediction/
last result?*

0 taken (1)

1 not taken (0)

2 taken (1)

3 taken (1)

...

14 not taken (0)

typical choice: some bits of branch address
for our example: will use bits 4-7

predict: repeat last

PC of branch

0x40042A

hash function

<i>index</i>	<i>prediction/ last result?</i>
0	taken (1)
1	not taken (0)
2	taken (1)
3	taken (1)
...	...
14	not taken (0)
15	taken (1)

predict: repeat last

PC of branch

0x40042A

hash function

<i>index</i>	<i>prediction/ last result?</i>
0	taken (1)
1	not taken (0)
2	taken (1)
3	taken (1)
...	...
14	not taken (0)
15	taken (1)

prediction
to fetch stage

predict: repeat last

PC of branch

0x40042A

hash function

index prediction/
last result?

0

taken (1)

1

not taken (0)

2

taken (1)

3

taken (1)

...

...

14

not taken (0)

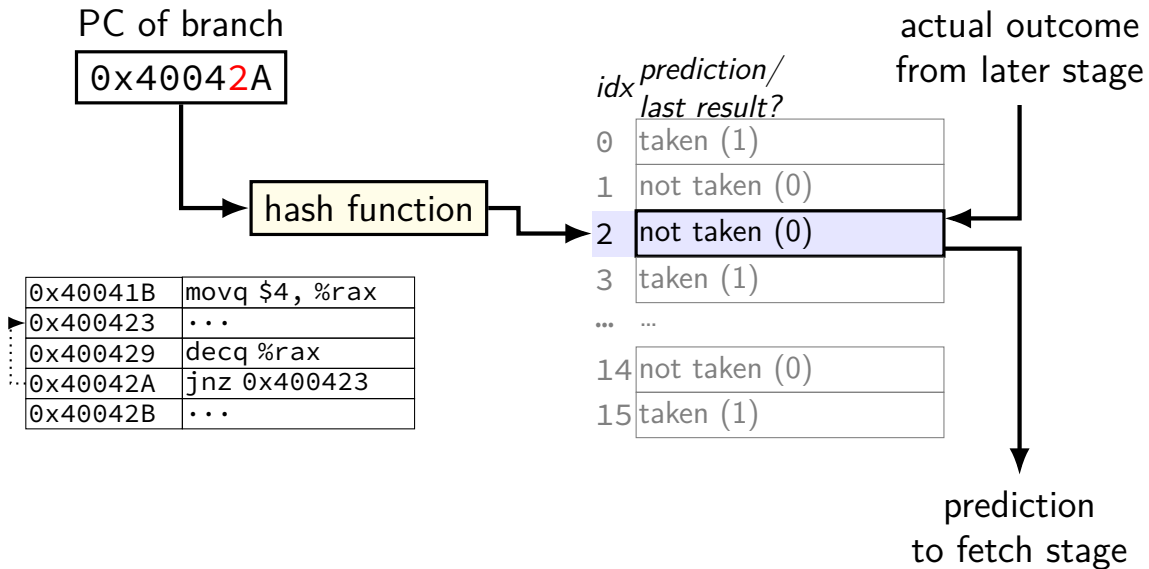
15

taken (1)

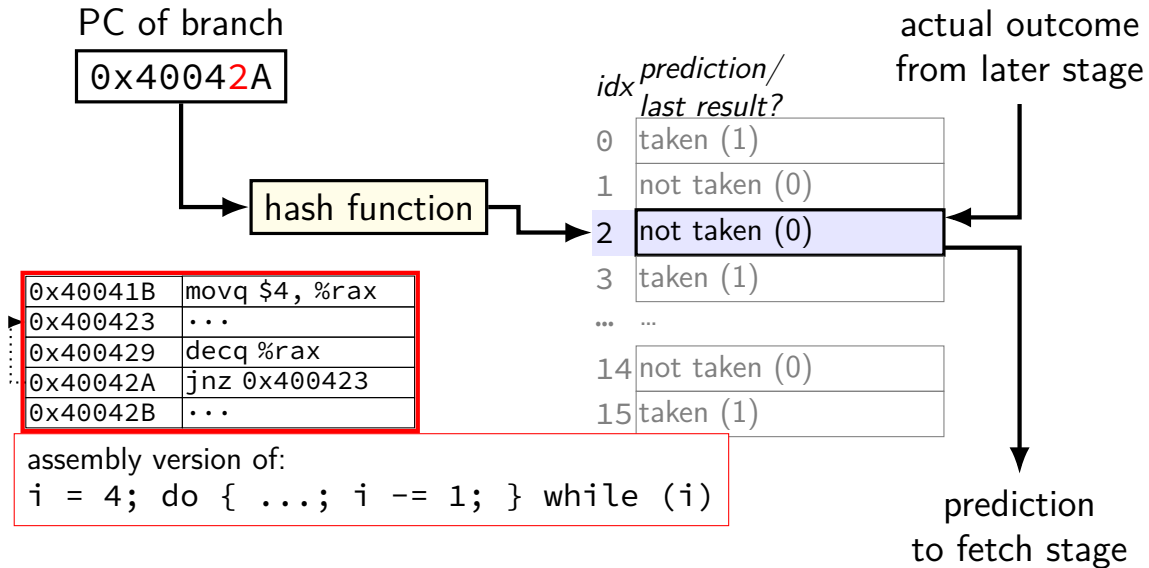
actual outcome
(from later stage)

prediction
to fetch stage

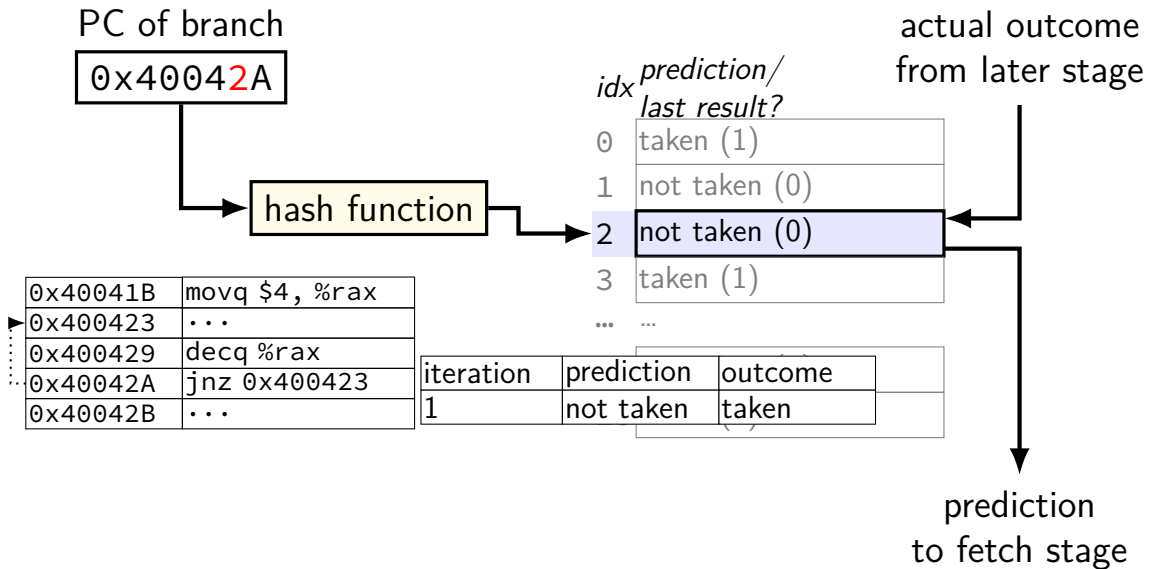
example



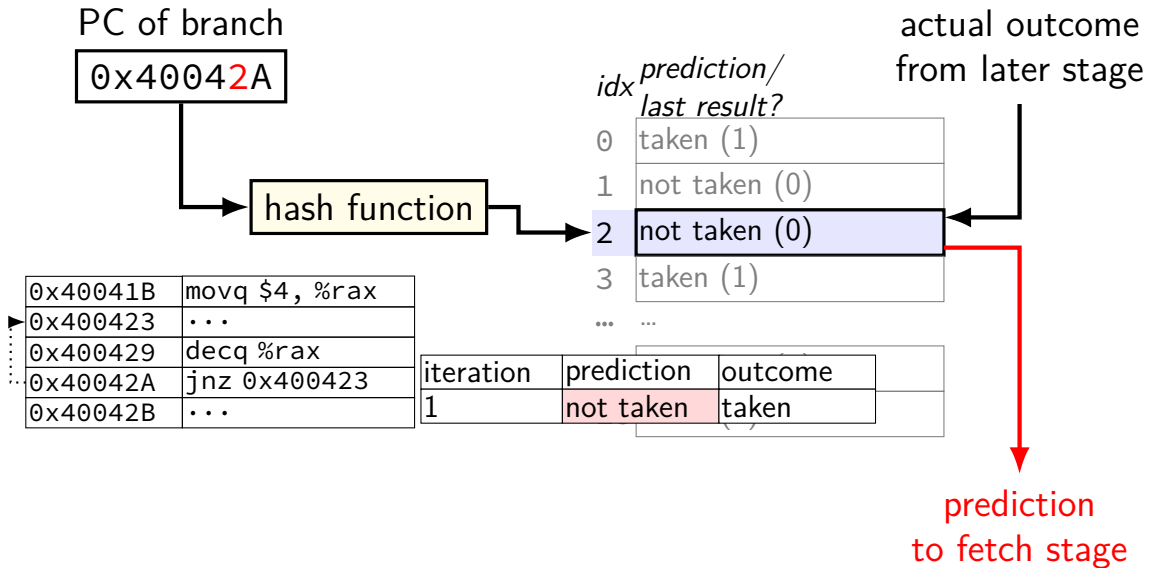
example



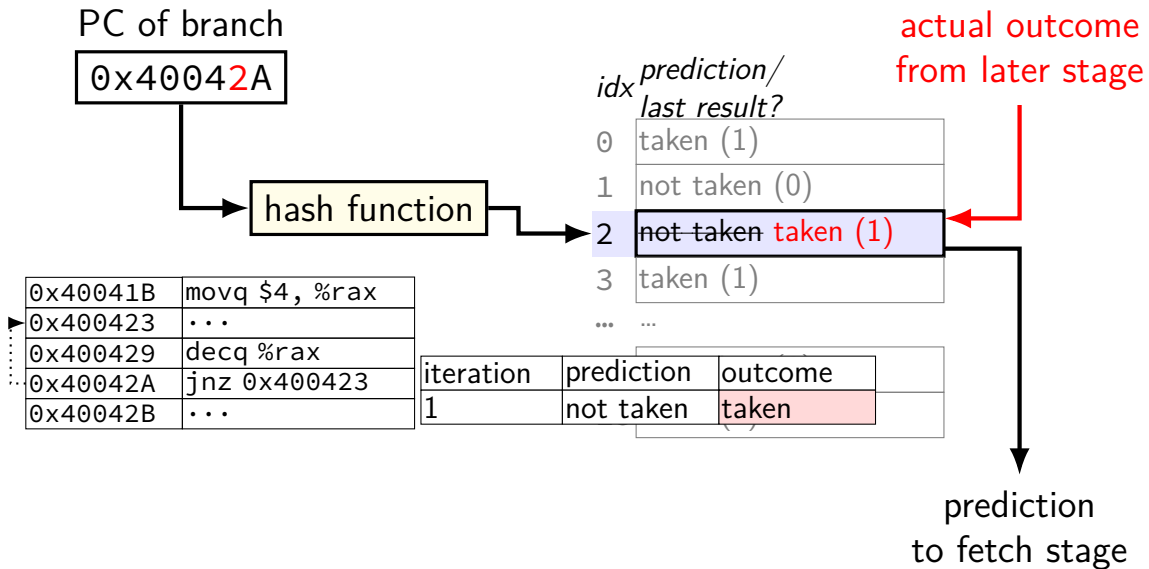
example



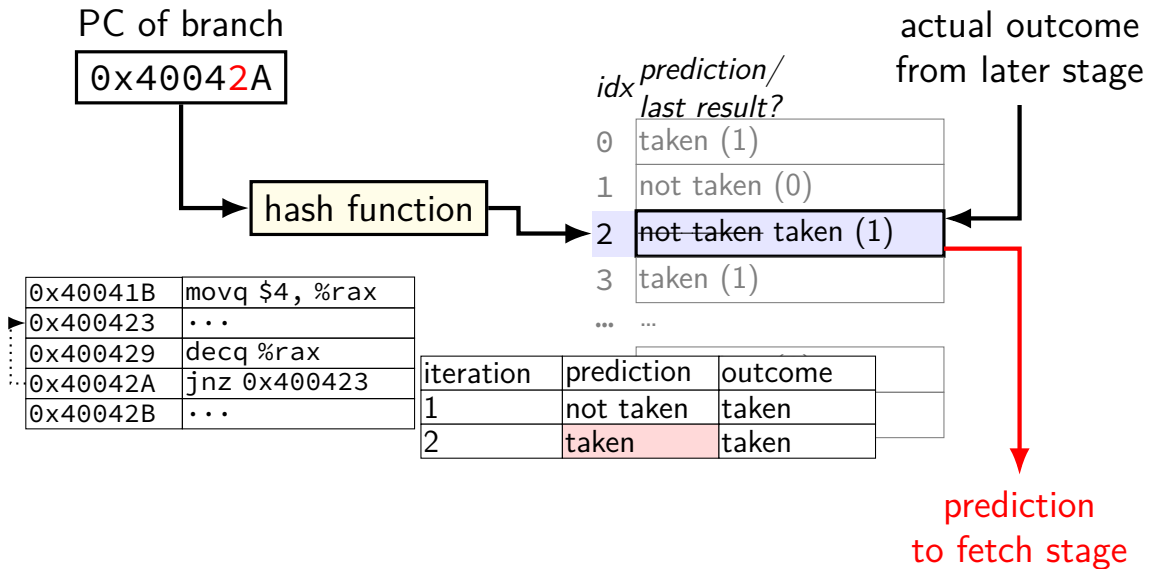
example



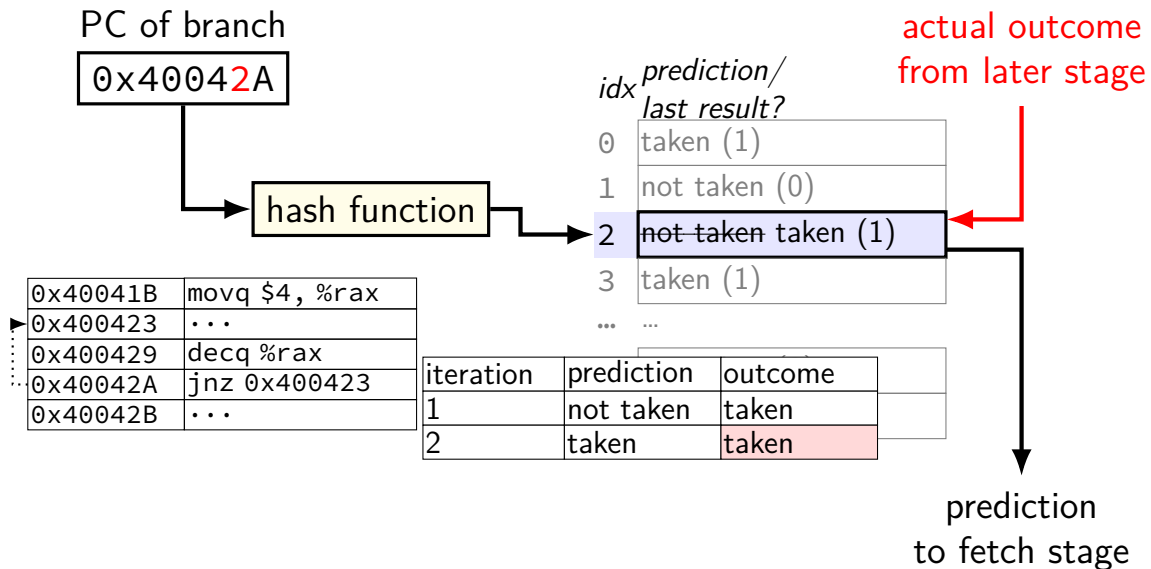
example



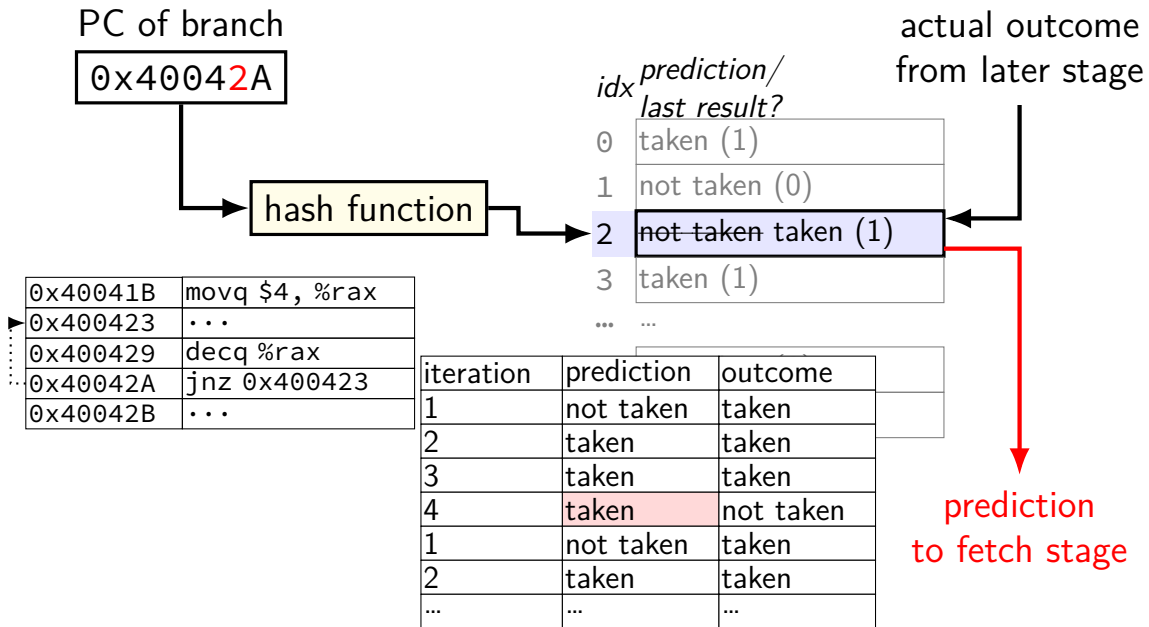
example



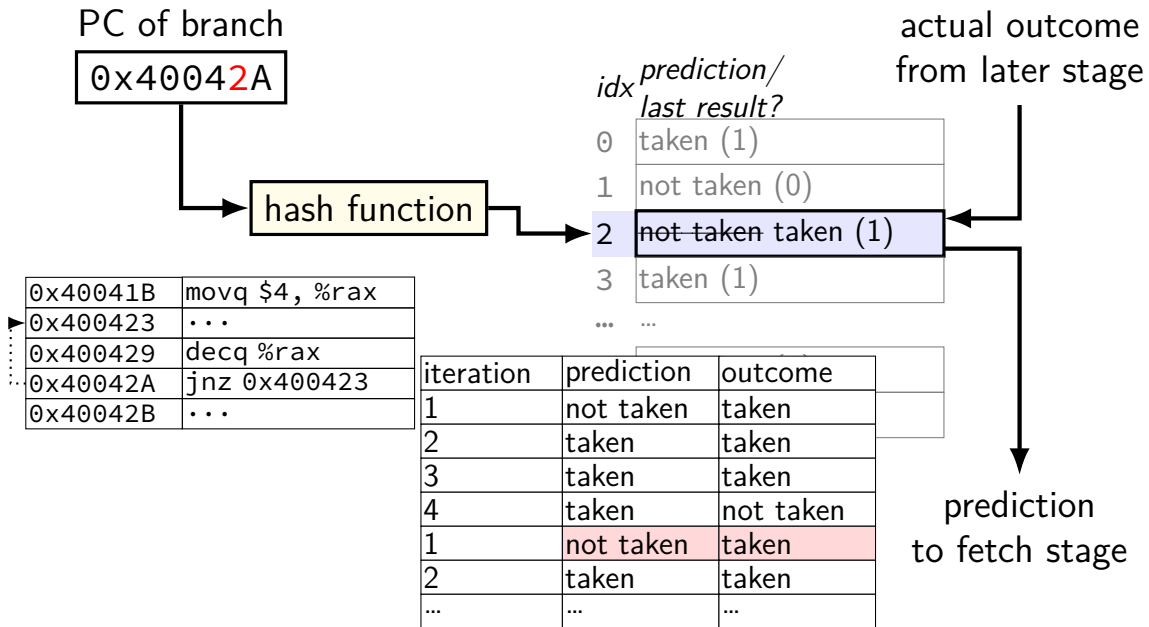
example



example



example



exercise

use 1-bit predictor on this loop

executed in outer loop (not shown) many, many times

what is the conditional branch misprediction rate?

```
int i = 0;
while (true) {
    if (i % 3 == 0) goto next;
    ...
next:
    i += 1;
    if (i == 50) break;
}
```


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?)
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 branch targets

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 branch targets

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

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

```

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


beyond pipelining: multiple issue

start **more than one instruction/cycle**

multiple parallel pipelines; many-input/output register file

hazard handling much more complex

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

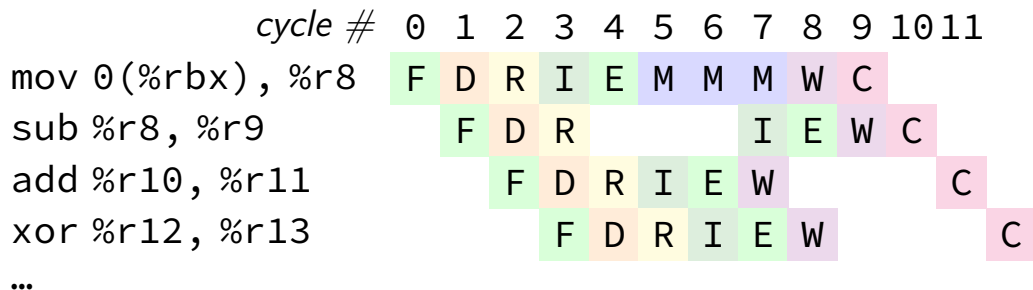


beyond pipelining: out-of-order

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

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

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



interlude: real CPUs

modern CPUs:

execute **multiple instructions at once**

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

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

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

- older value may be written back before newer value?

problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

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

- older value may be written back before newer value?

problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

read-after-write examples (1)

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

normal pipeline: two options for %r8?

choose the one from *earliest stage*

because it's from the most recent instruction

read-after-write examples (1)

out-of-order execution:

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

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

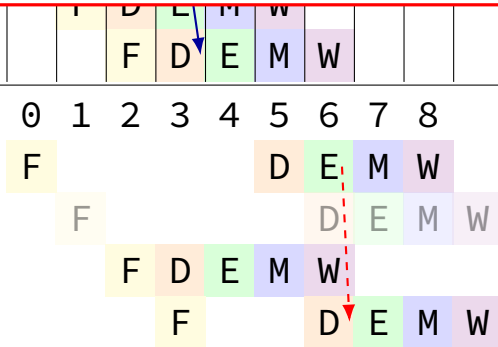
cycle # 0 1 2 3 4 5 6 7 8

addq %r10, %r8

movq %r8, (%rax)

movq \$100, %r8

addq %r13, %r8



register version tracking

goal: track **different versions of registers**

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

only forward the **correct version**

strategy for doing this: preprocess instructions represent version info

makes forwarding, etc. lookup easier

rewriting hazard examples (1)

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

read different version than the one written

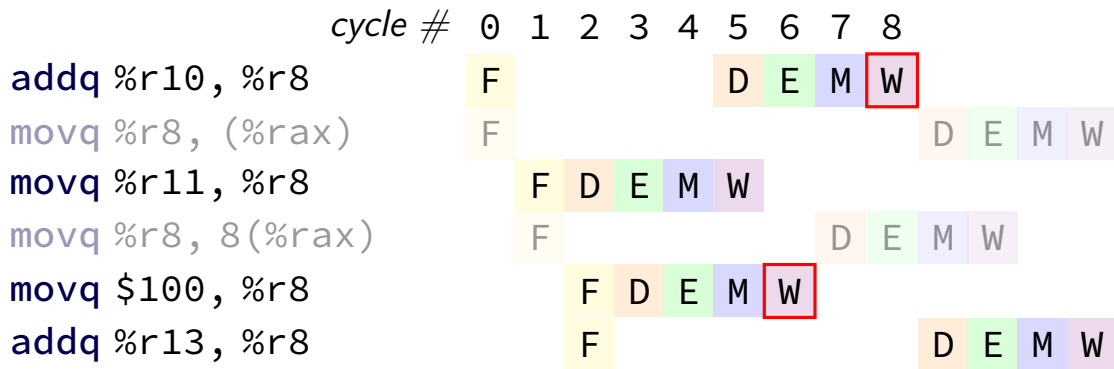
represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

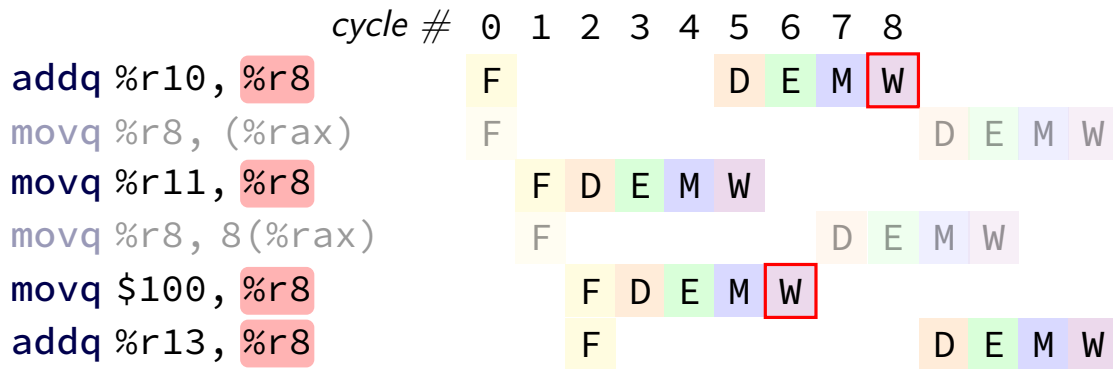
for now: version numbers

later: something simpler to implement

write-after-write example



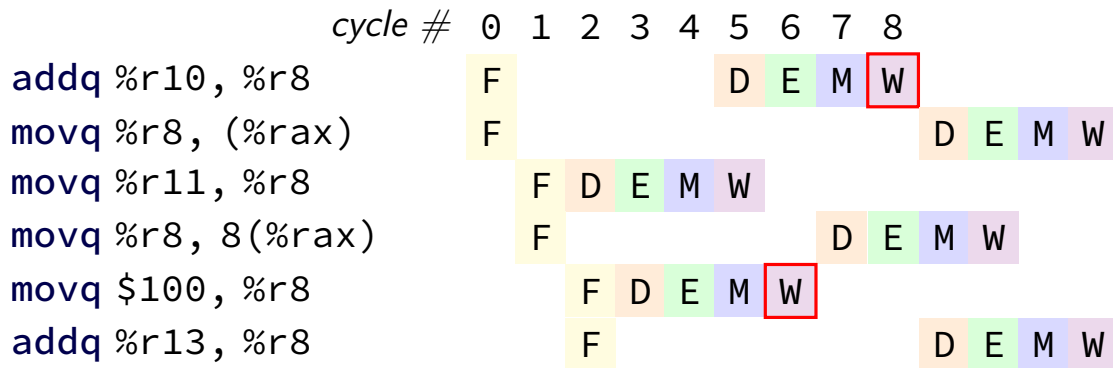
write-after-write example



out-of-order execution:

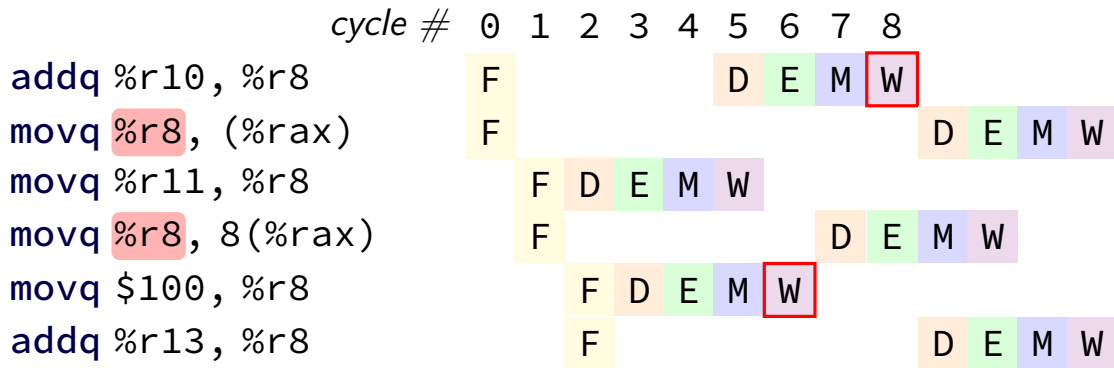
if we don't do something, newest value could be overwritten!

write-after-write example



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

write-after-write example



keeping multiple versions

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

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

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

solution: have lots of extra registers

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

called register renaming

register renaming

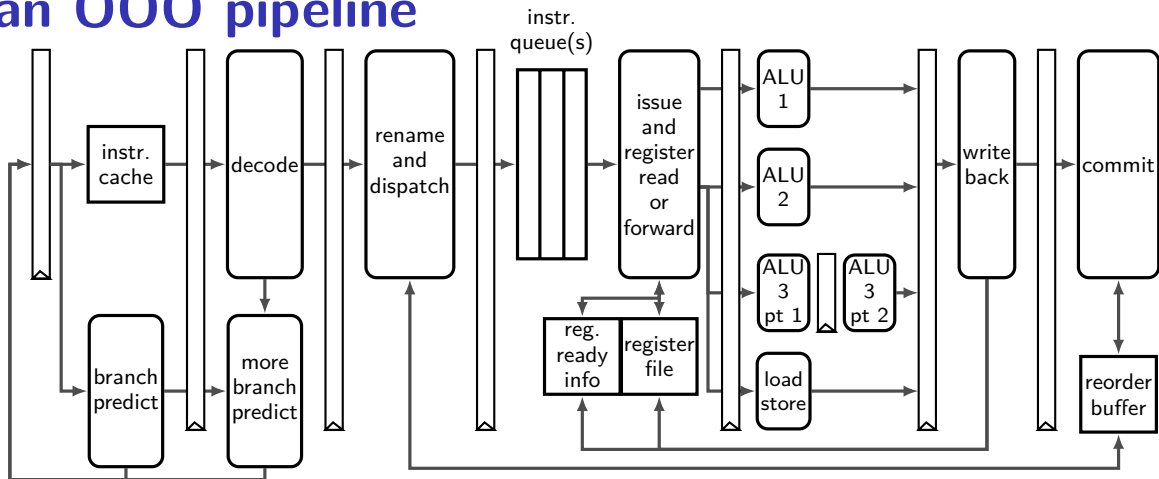
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

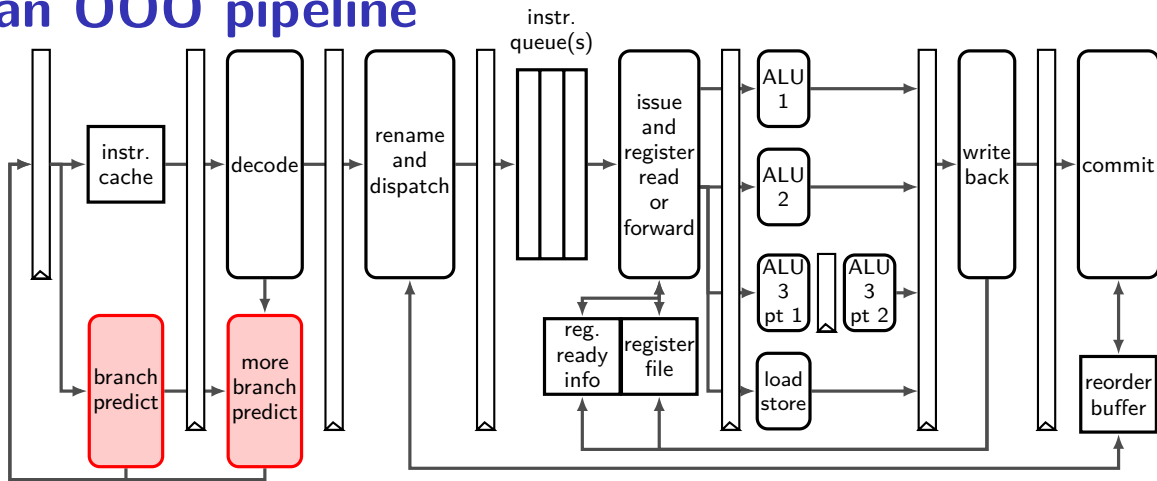
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

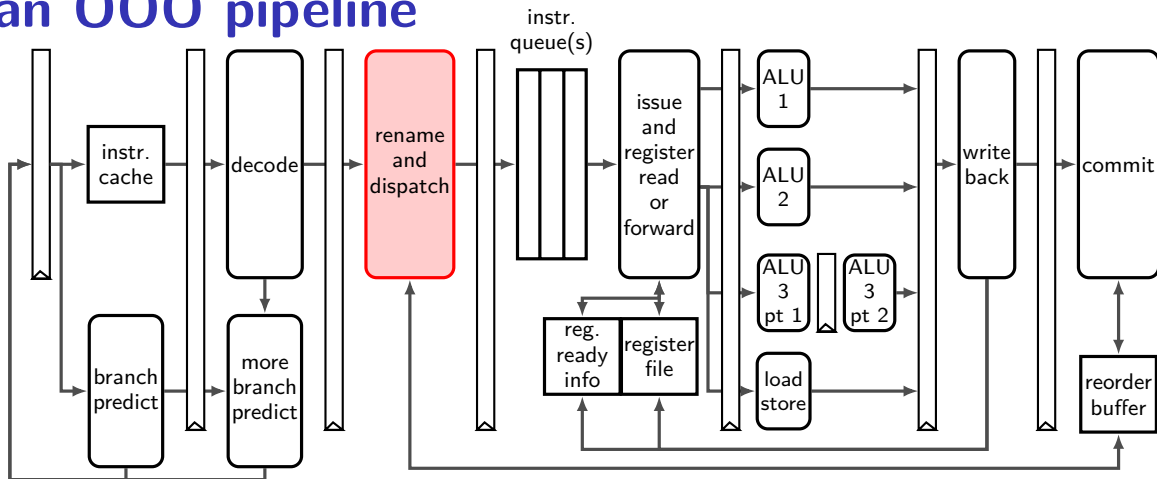


an OOO pipeline



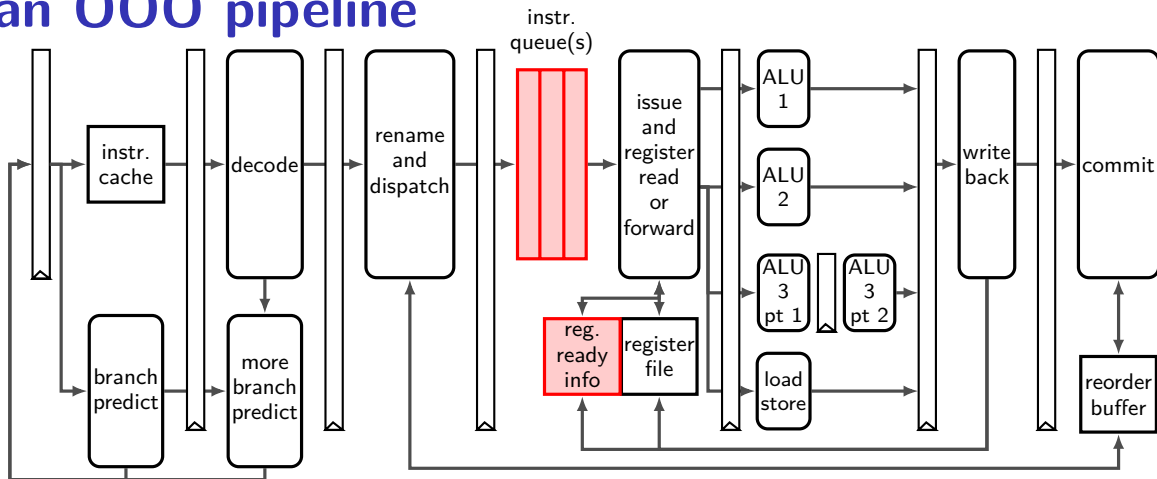
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



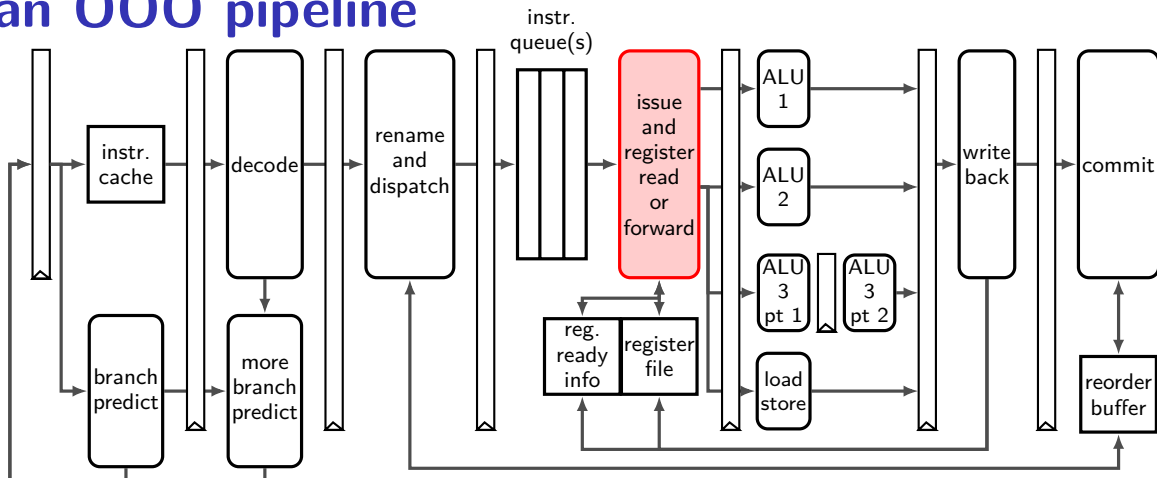
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



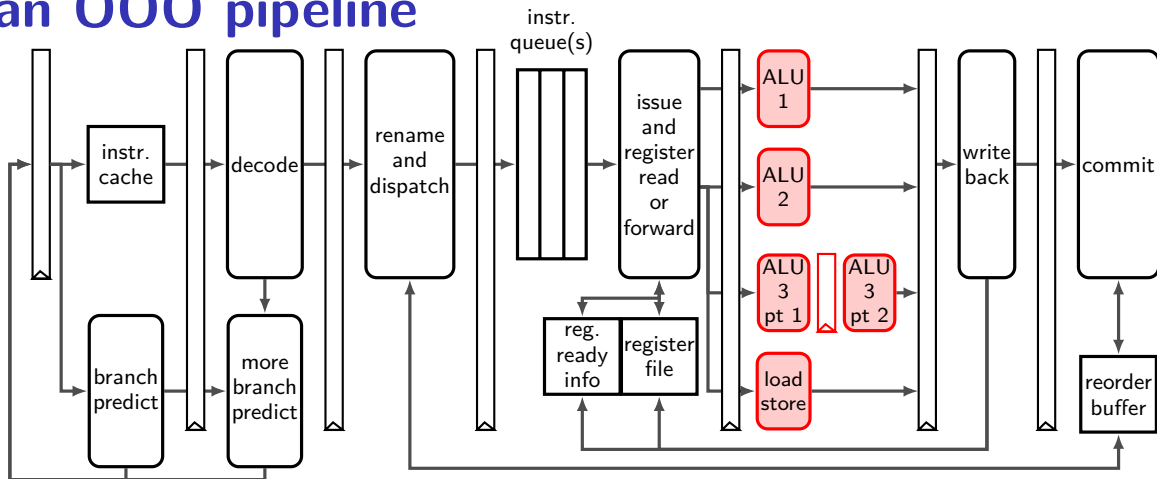
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



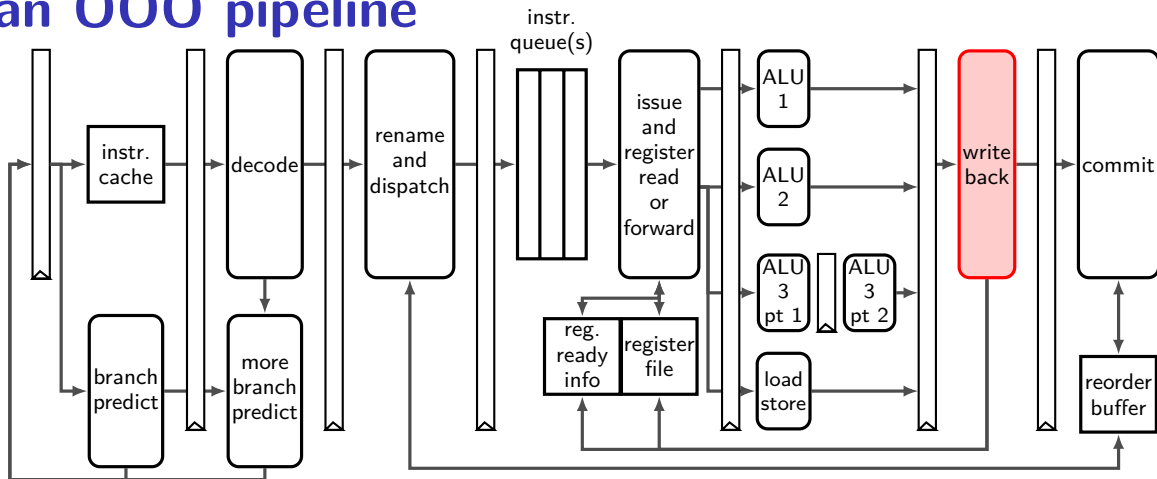
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



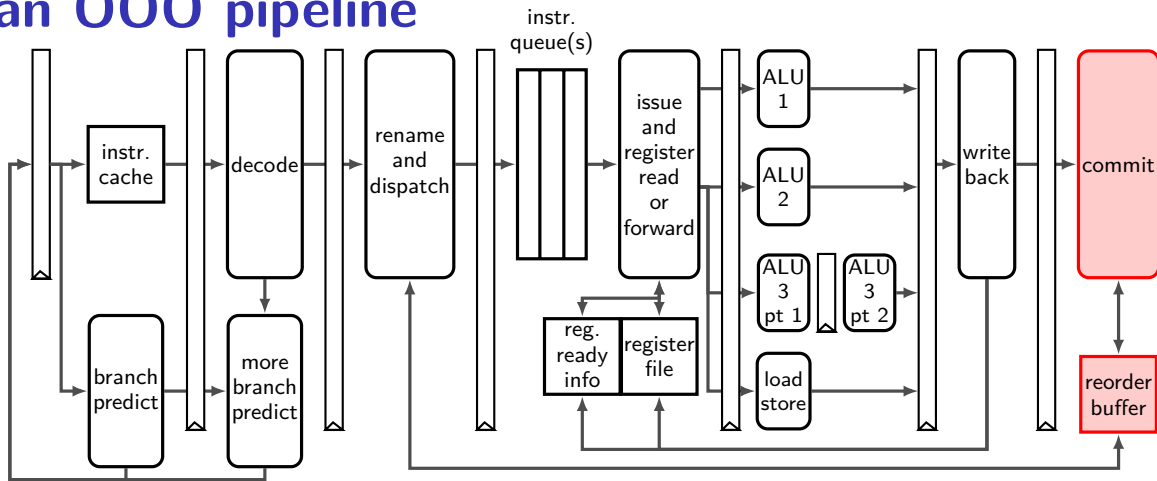
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



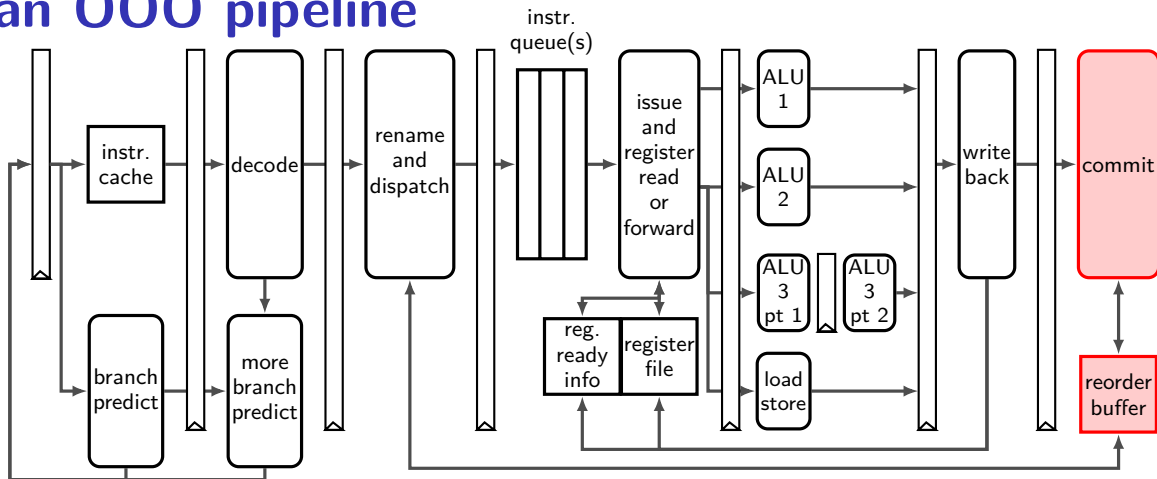
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



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

an OOO pipeline

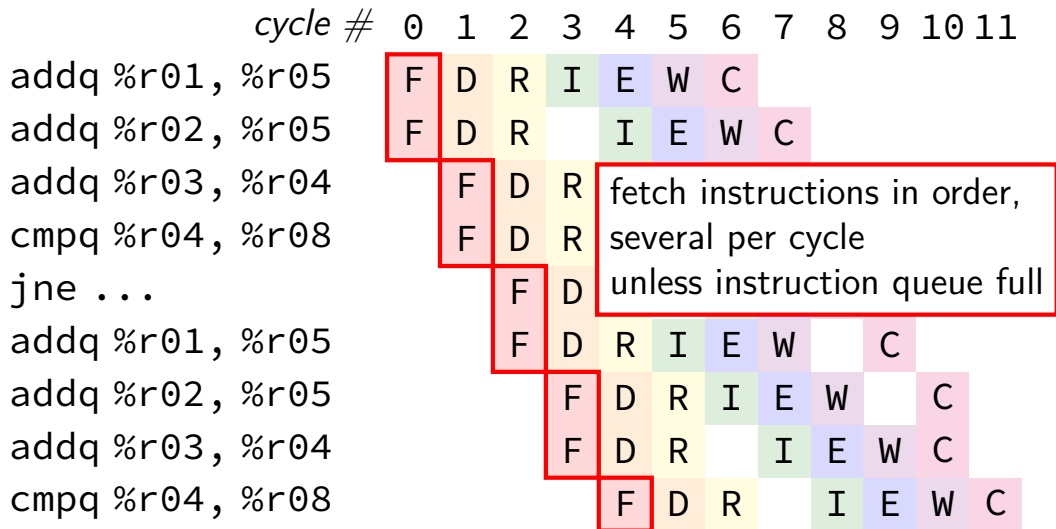


commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.

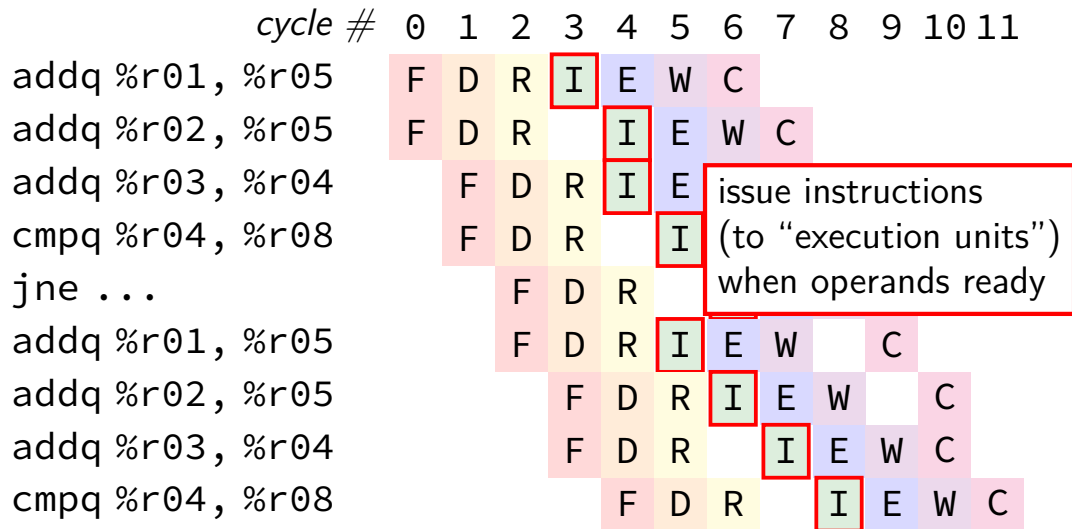
an OOO pipeline diagram

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

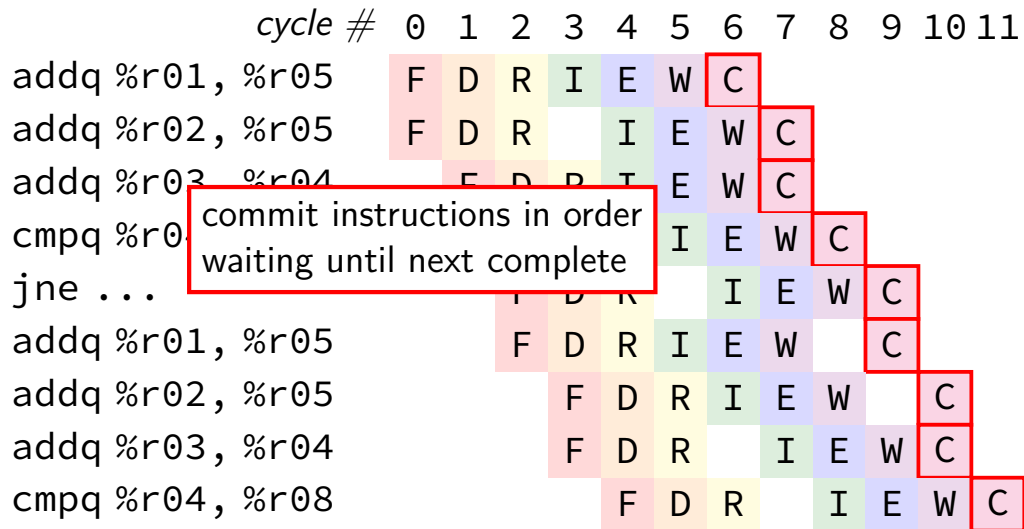
an OOO pipeline diagram



an OOO pipeline diagram



an OOO pipeline diagram



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?)
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 branch targets

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 branch targets

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
...
```

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 polymorphic method calls

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

an OOO pipeline diagram

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

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
<code>add %r10, %r8 ...</code>	
<code>add %r11, %r8 ...</code>	
<code>add %r12, %r8 ...</code>	

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

```
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	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	

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

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

renamed

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

arch → phys register map

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

free
regs

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

register renaming example (2)

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

arch → phys register map

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

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

not handled via register renaming

option 1: run load+stores in order

option 2: compare load/store addresses

%x21
%x23
%x24
...

register renaming example (2)

original

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

renamed

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

arch → phys register map

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

free
regs

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

register renaming example (2)

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

arch → phys register map

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

free
regs

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

register renaming example (2)

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

arch → phys register map

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

free
regs

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

register renaming example (2)

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

arch → phys register map

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

free
regs

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

register renaming exercise

original

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

arch → phys

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

renamed

free
regs

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

an OOO pipeline diagram

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

instruction queue and dispatch

instruction queue

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

... ..

execution unit

ALU 1

ALU 2

scoreboard

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

...

instruction queue and dispatch

instruction queue

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

... ..

scoreboard

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

execution unit cycle# 1

ALU 1 1

ALU 2

...

instruction queue and dispatch

instruction queue

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

... ..

scoreboard

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

execution unit cycle# 1

ALU 1 1

ALU 2

...

instruction queue and dispatch

instruction queue

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

... ..

scoreboard

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

execution unit cycle# 1

ALU 1 1

ALU 2 —

...

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

...

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

... ..

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

... ..

scoreboard

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

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

ALU

data cache



assume

1 cycle/access

register renaming: missing pieces

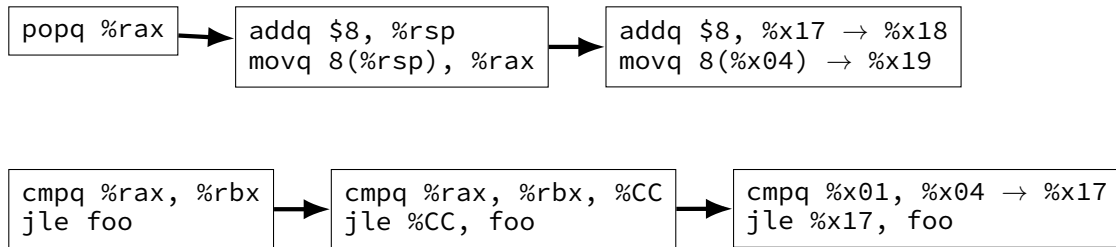
what about “hidden” inputs like `%rsp`, condition codes?

one solution: translate to instructions with additional register parameters

- making `%rsp` explicit parameter

- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones



an OOO pipeline diagram

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

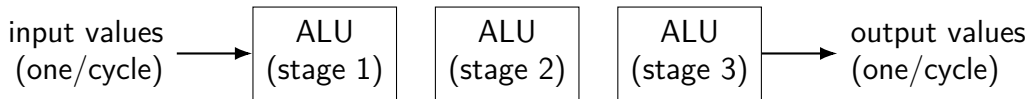
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



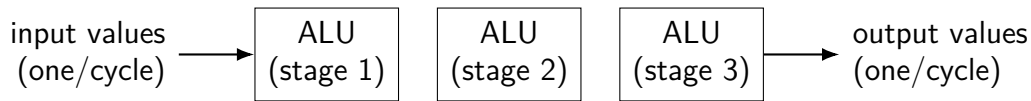
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



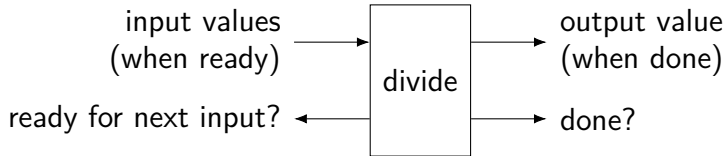
exercise: how long to compute $A \times (B \times (C \times D))$?

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)

scoreboard

instruction queue

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

... ..

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

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

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

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

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

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

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

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

execution unit	cycle#	1	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

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
...

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

execution unit	cycle#	1	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

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
...

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

execution unit	cycle#	1	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

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
...

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

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

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
...

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

execution unit	cycle#	1	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	

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
...

instruction queue and dispatch (multicycle)

scoreboard

instruction queue

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

... ..

execution unit	cycle#	1	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	

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
...	...

6
9
—

instruction queue and dispatch (multicycle)

scoreboard

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
6	7... ..

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	

9 10
— —

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 $1000000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx 2.65 sec

example2:

```
    movq $1000000000000, %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 $1000000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx 2.65 sec

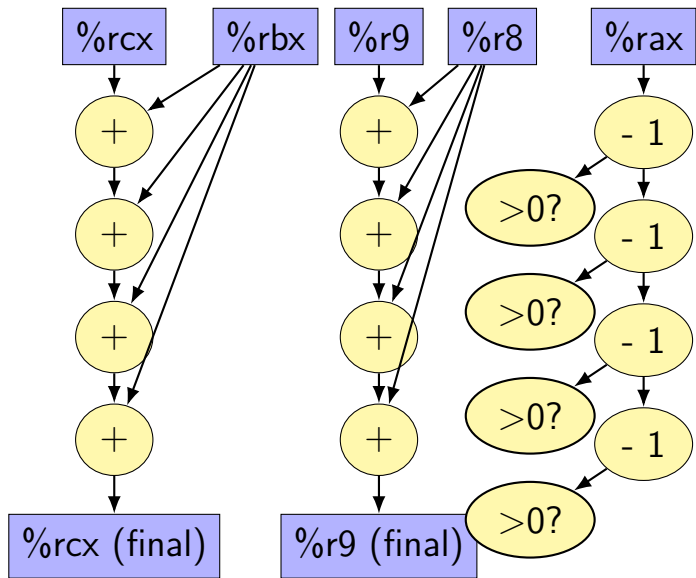
example2:

```
    movq $1000000000000, %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:

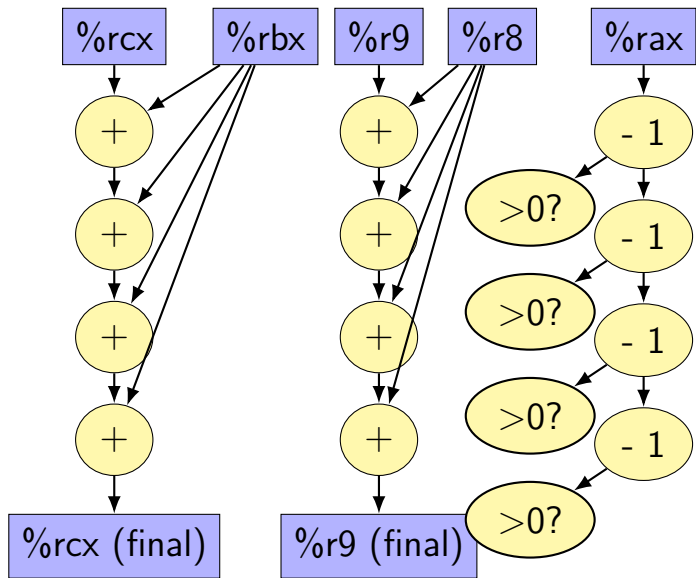
```
addq %rbx, %rcx
```

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

```
decq %rax
```

```
jge loop2
```

data flow model and limits (1)



each yellow box =
instruction

arrows = dependencies

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
```

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

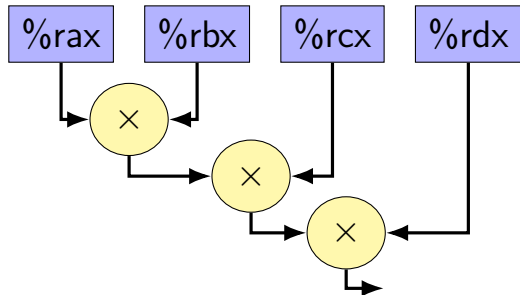
```
imulq %rbx, %rax  
imulq %rcx, %rdx  
imulq %rdx, %rax
```

reassociation

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

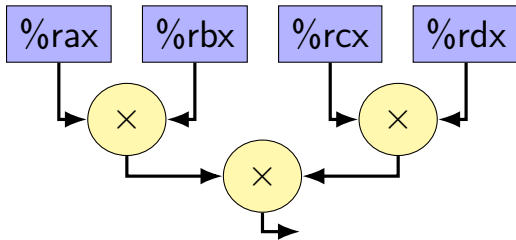
$$((a \times b) \times c) \times d$$

```
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 things

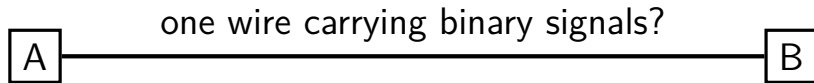
how to (in hardware) connect A and B?

A

B

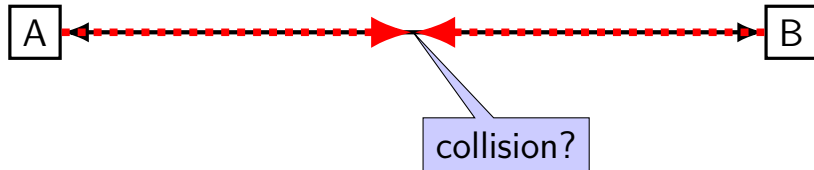
connecting things

how to (in hardware) connect A and B?



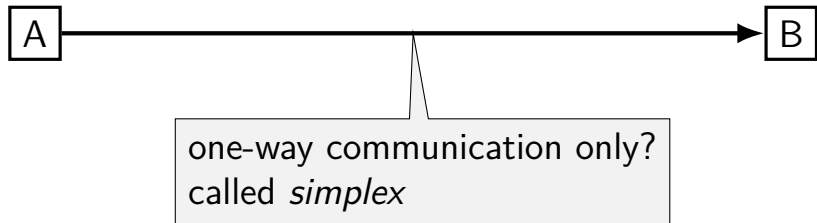
connecting things

how to (in hardware) connect A and B?



connecting things

how to (in hardware) connect A and B?



connecting things

how to (in hardware) connect A and B?



...and later



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?



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

connecting things

how to (in hardware) connect A and 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

B

C

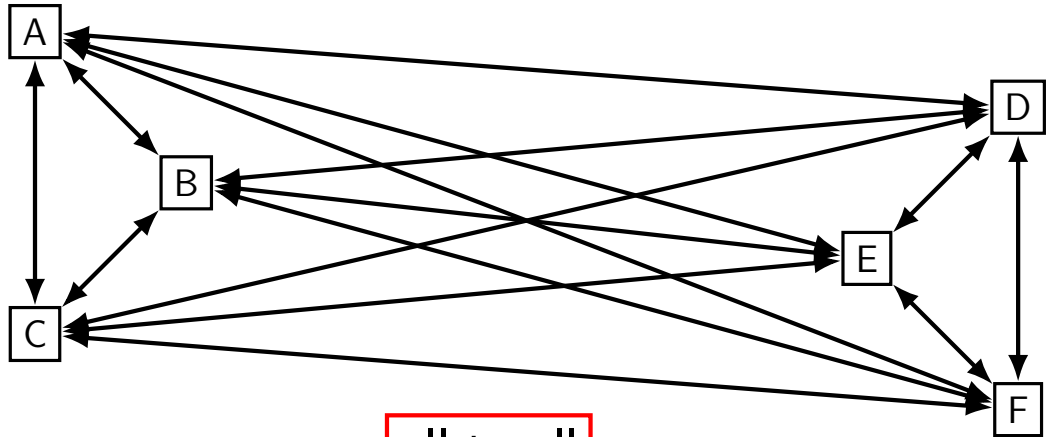
D

E

F

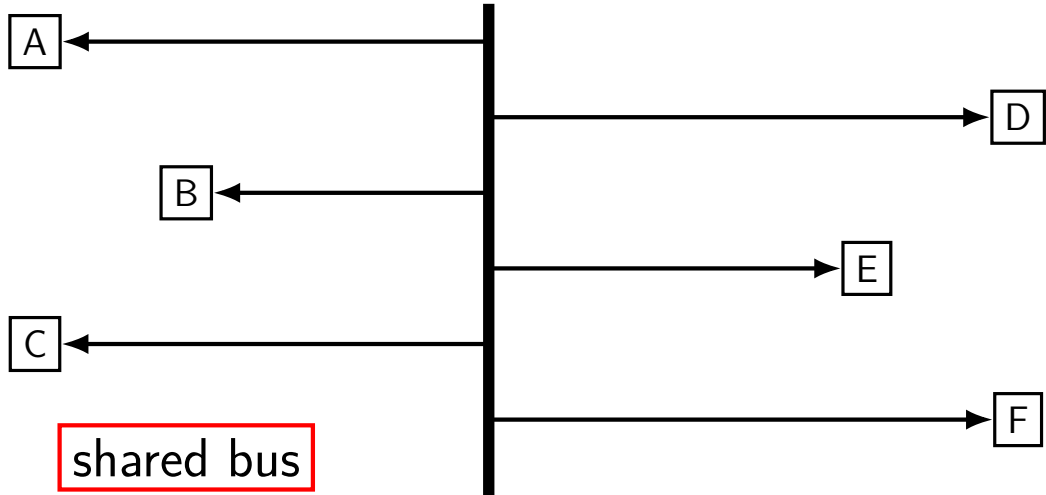
how to connect?

connecting things

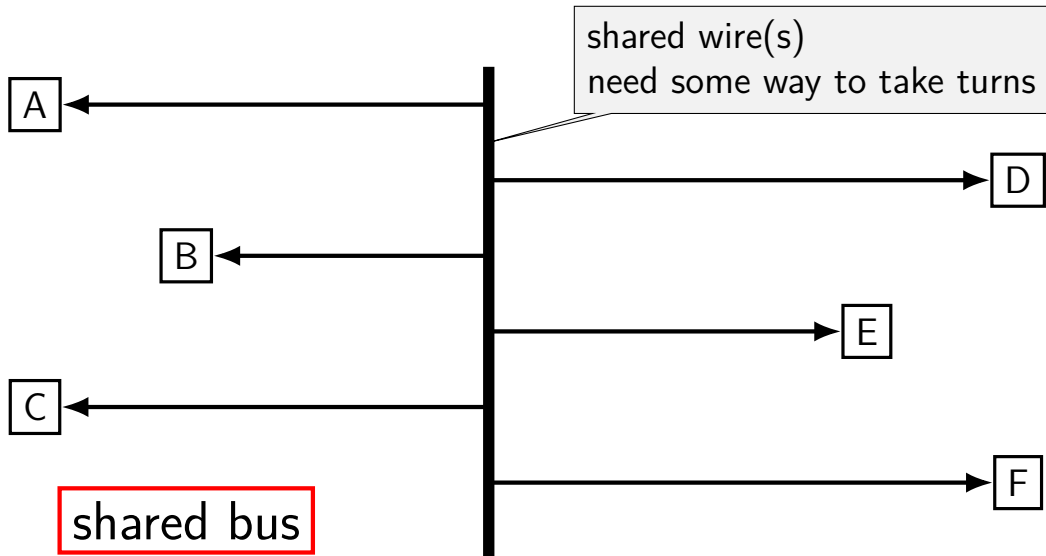


all-to-all

connecting things



connecting things



shared bus, really?

common for parts of internals of computers (topic later)

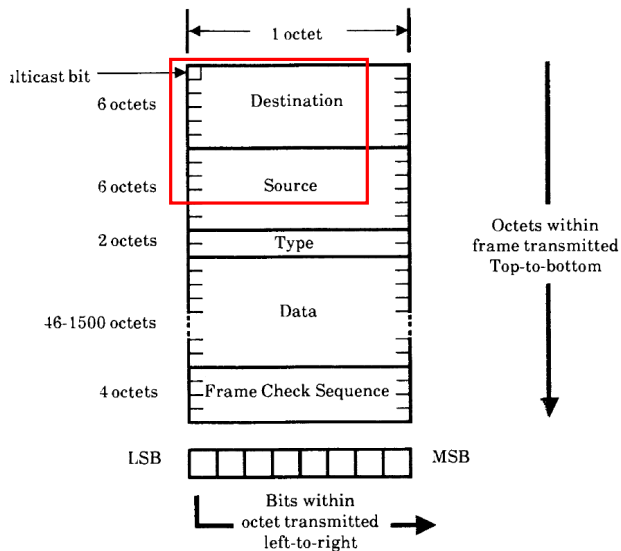
model for wifi

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?



messages need a '**header**' to tell who it's to/from

everyone needs to filter out messages that aren't theirs

Figure 6-1: Data Link Layer Frame Format

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

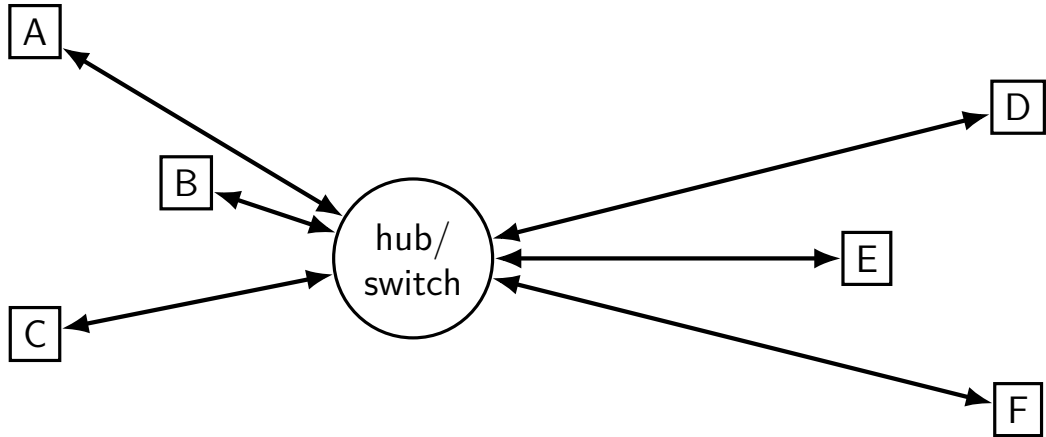
coordinating machine transmits timeslots

- 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?

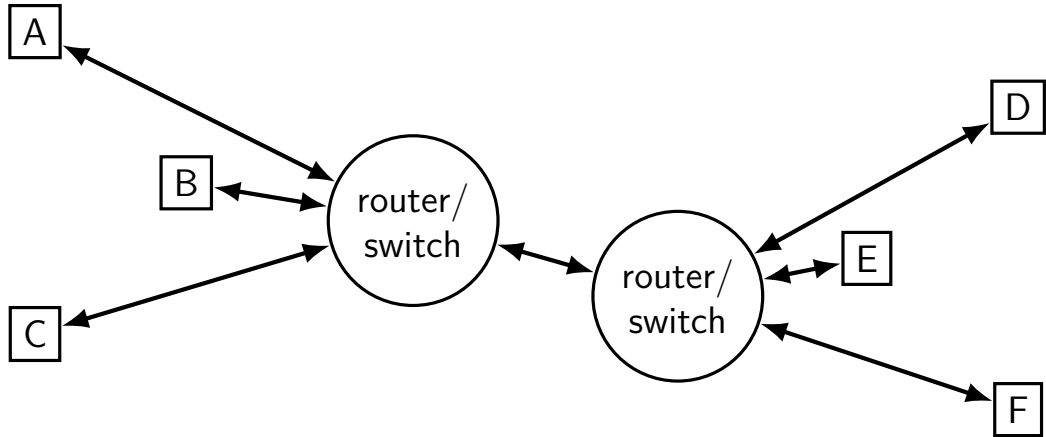
simple version:

- 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

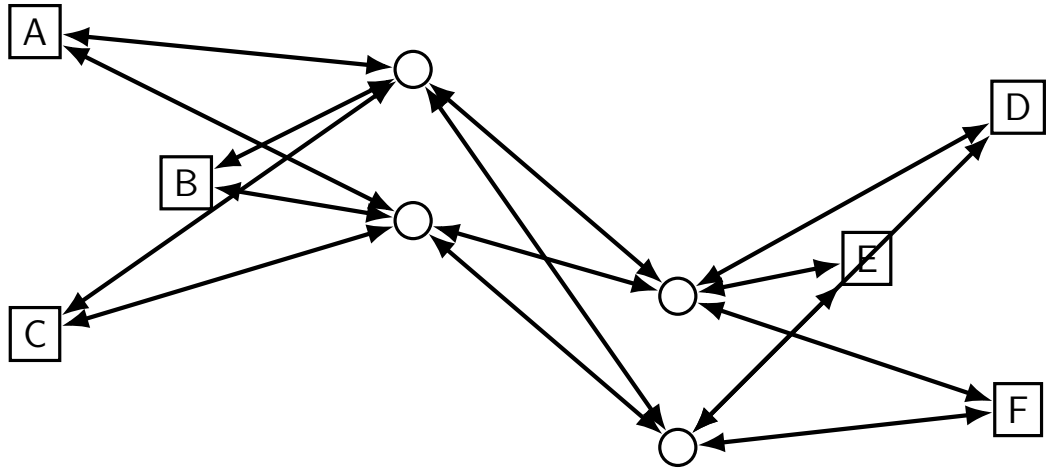
hierarchies

networks of networks

“internetworks”

so far still have single points of failure

connecting things



individual computers are networks

individual computers are (kinda) networks of...

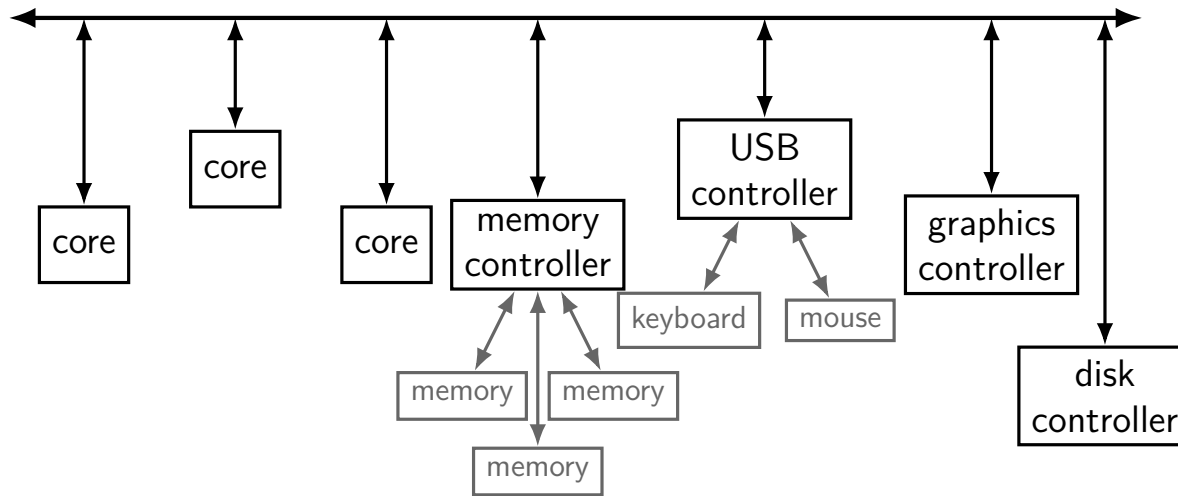
- processors

- 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)

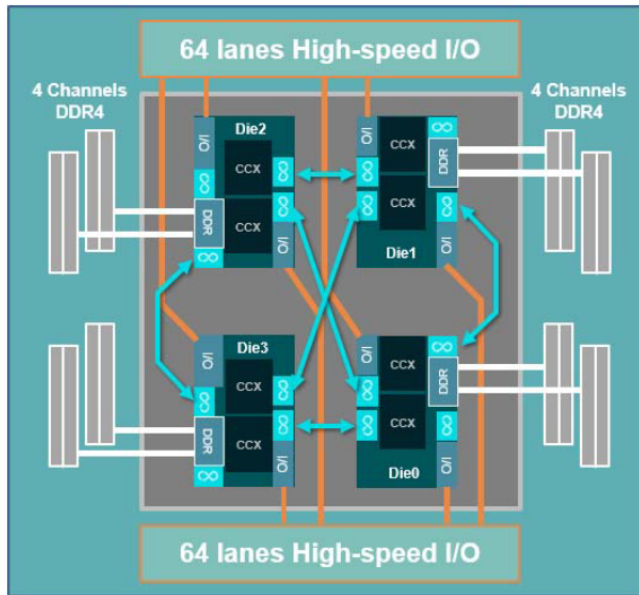


Fig. 21. Single-socket AMD EPYC™ system (SP3).

Figure from Burd et al, " 'Zeppelin': An SoC for Multichip Architectures" (IEEE JSSC Vol 54, No 1)

example: Intel Skylake-SP

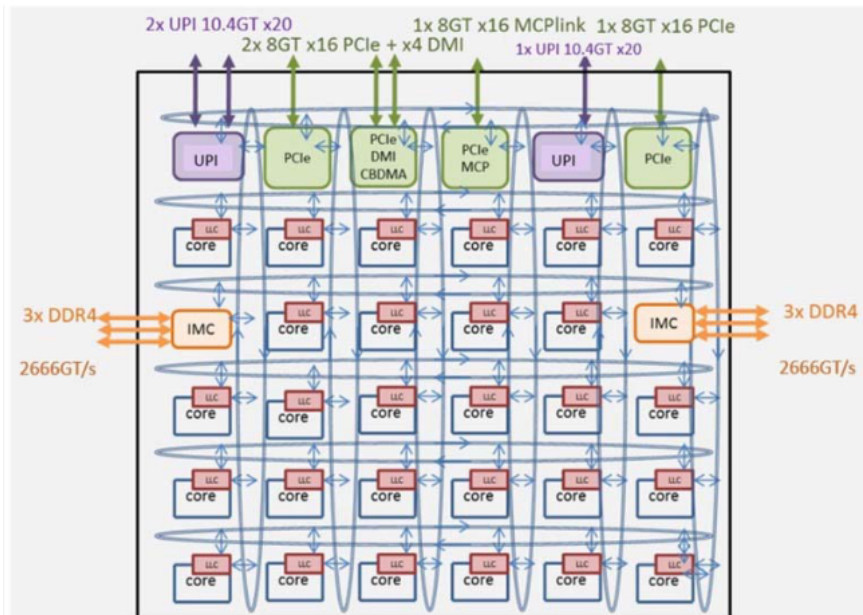
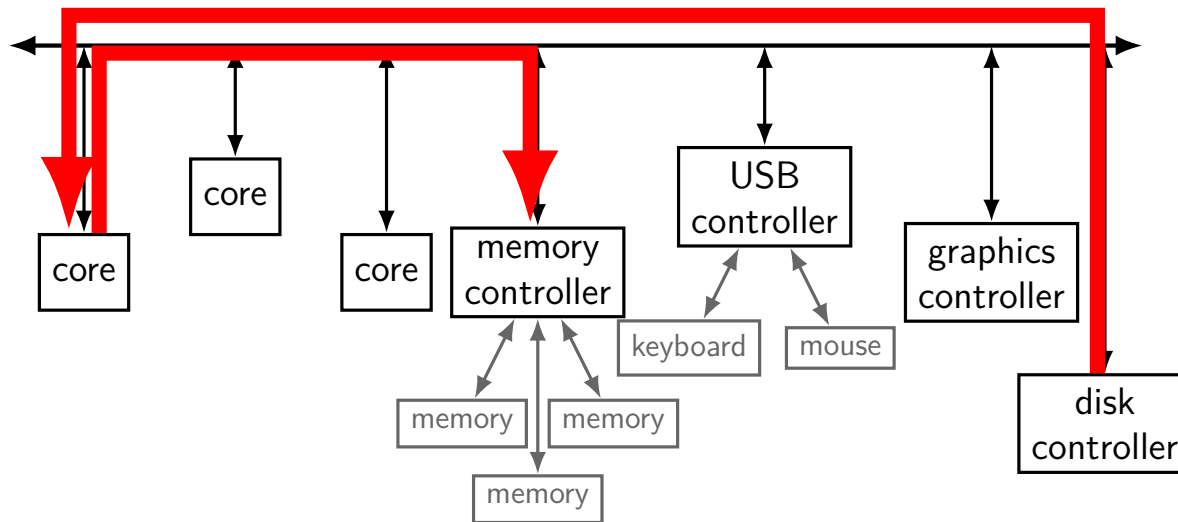
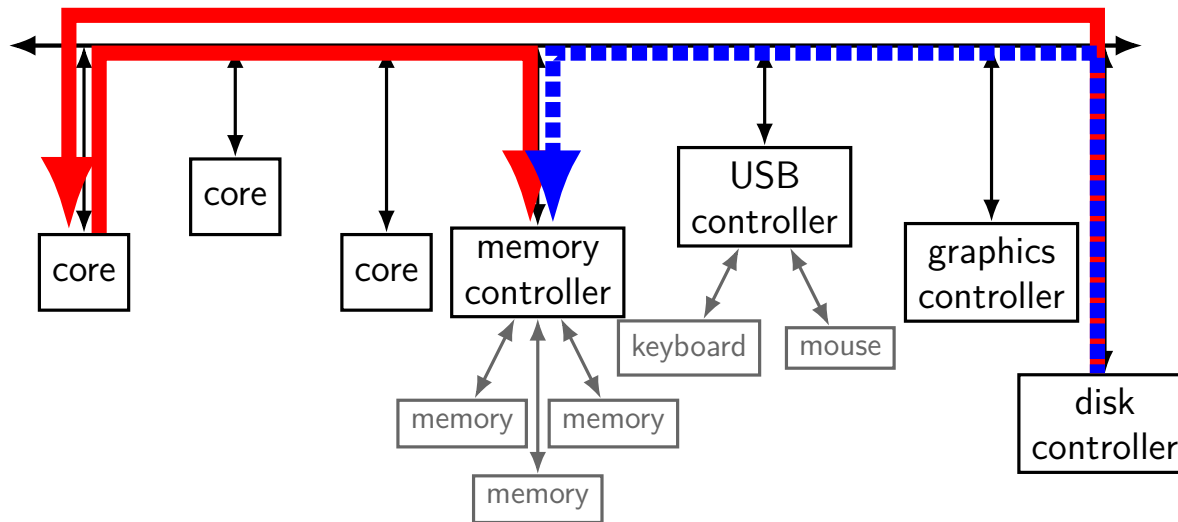


Figure from Tam et al, "SkyLake-SP: A 14nm 28-Core Xeon® Processor" (ISSCC 2018)

extra trips to CPU

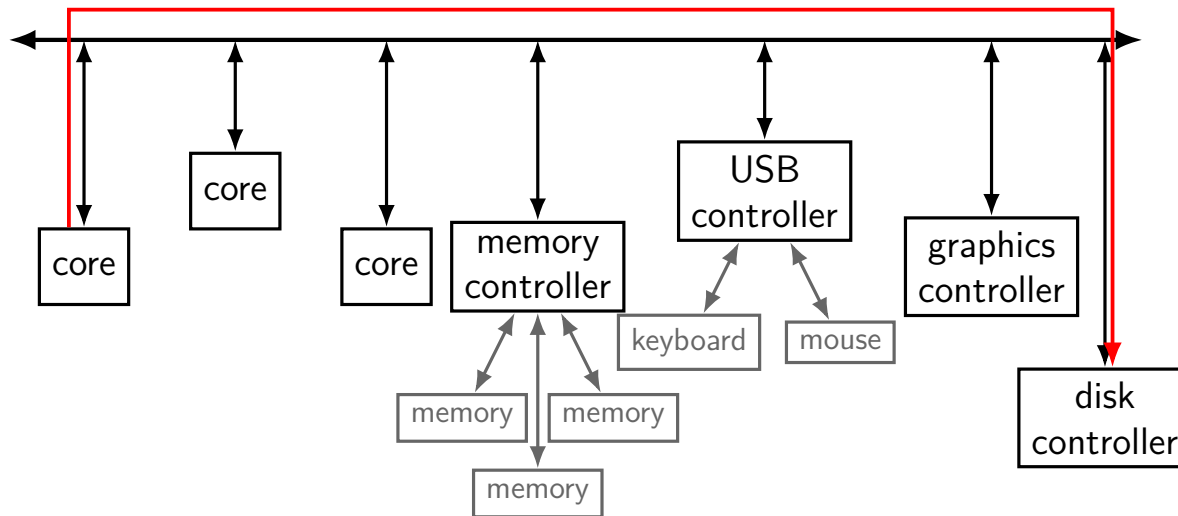


extra trips to CPU



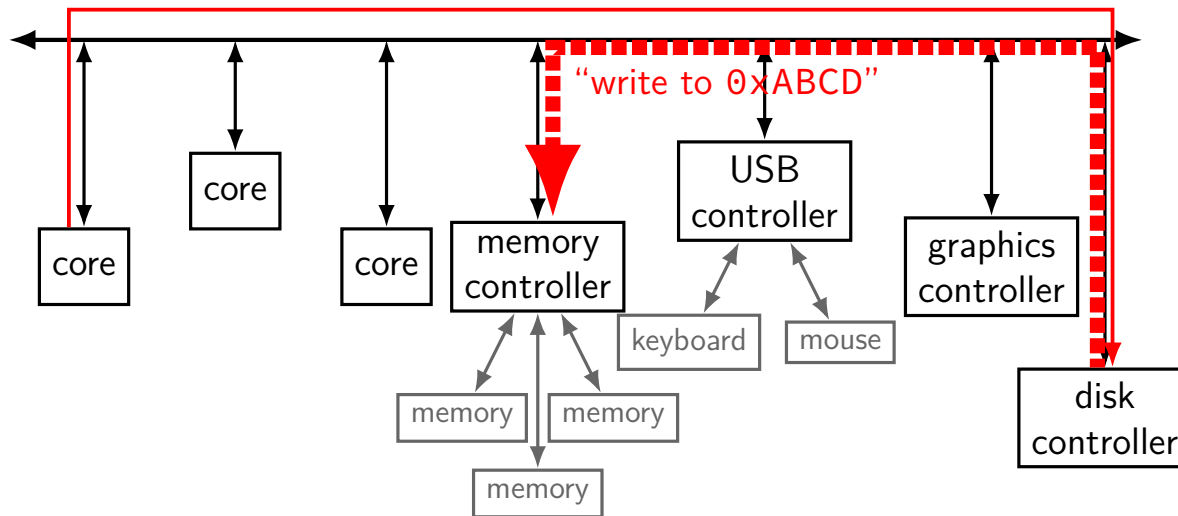
DMA

“place data at 0xABCD”

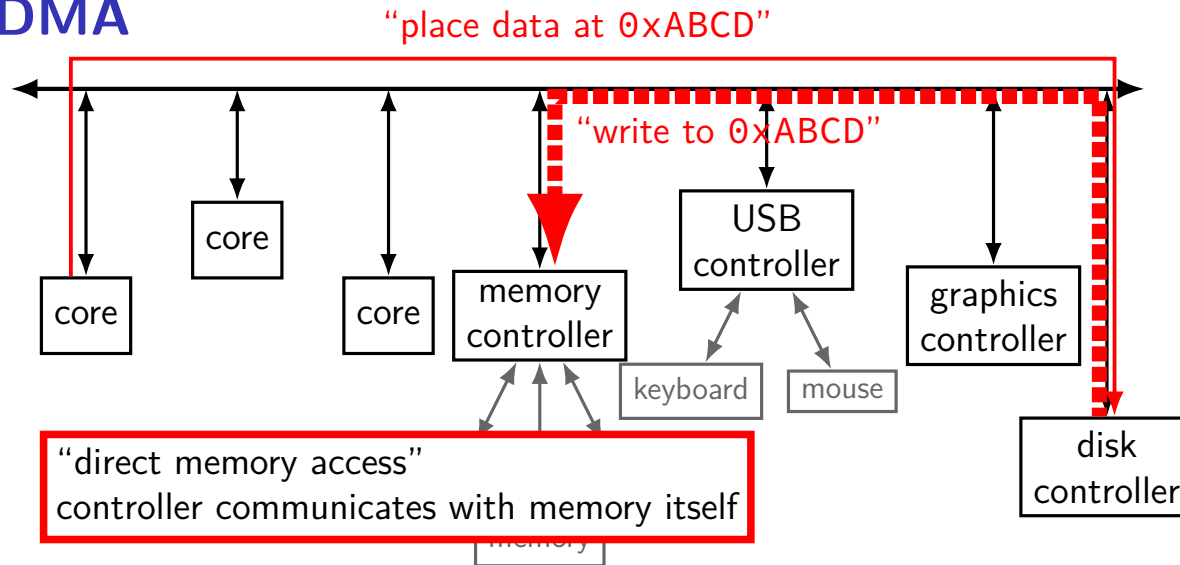


DMA

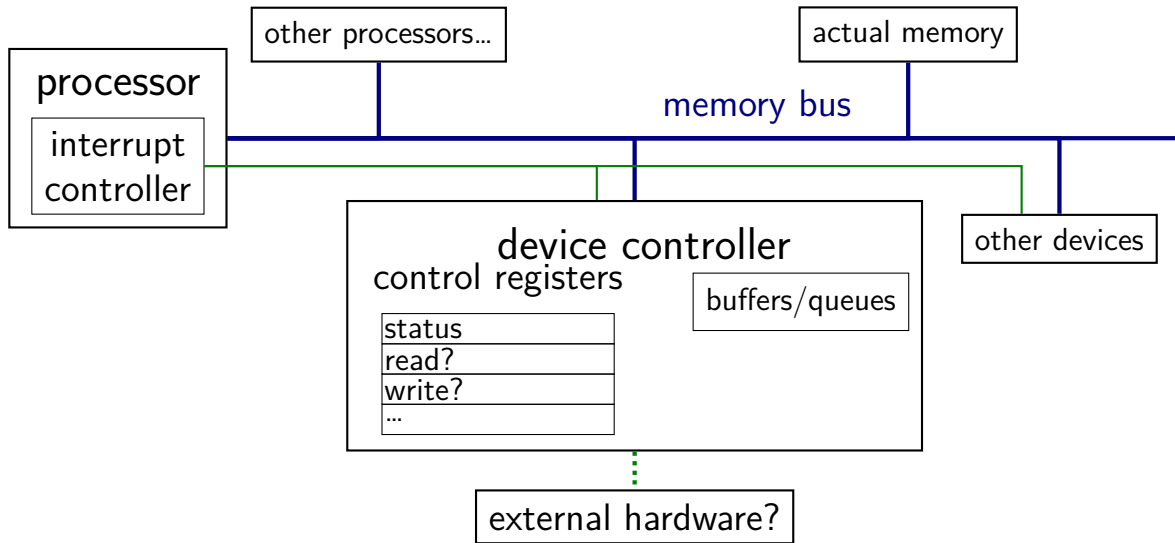
“place data at 0xABCD”



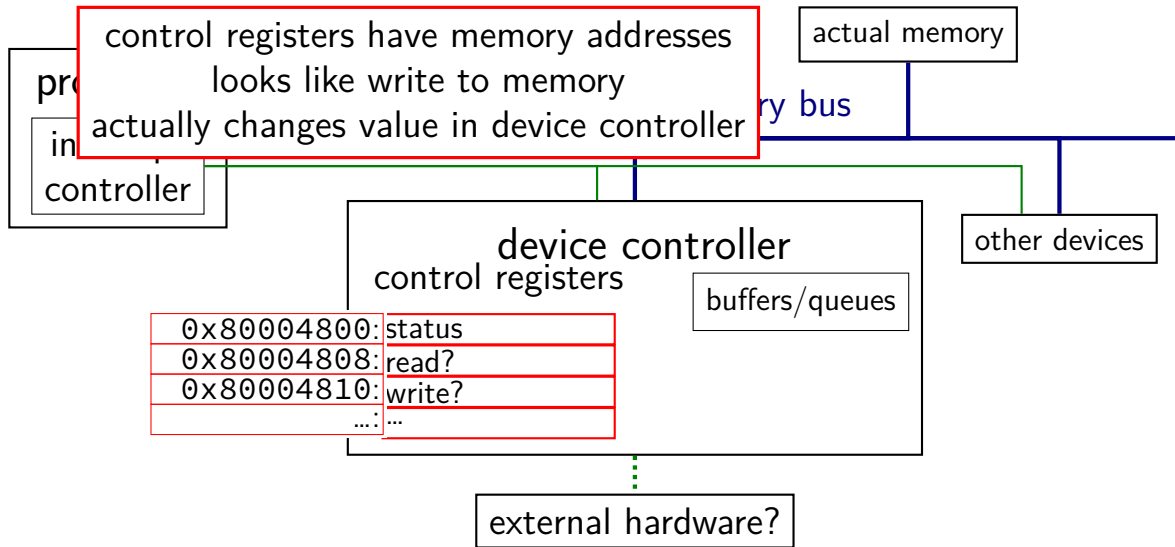
DMA



connecting devices

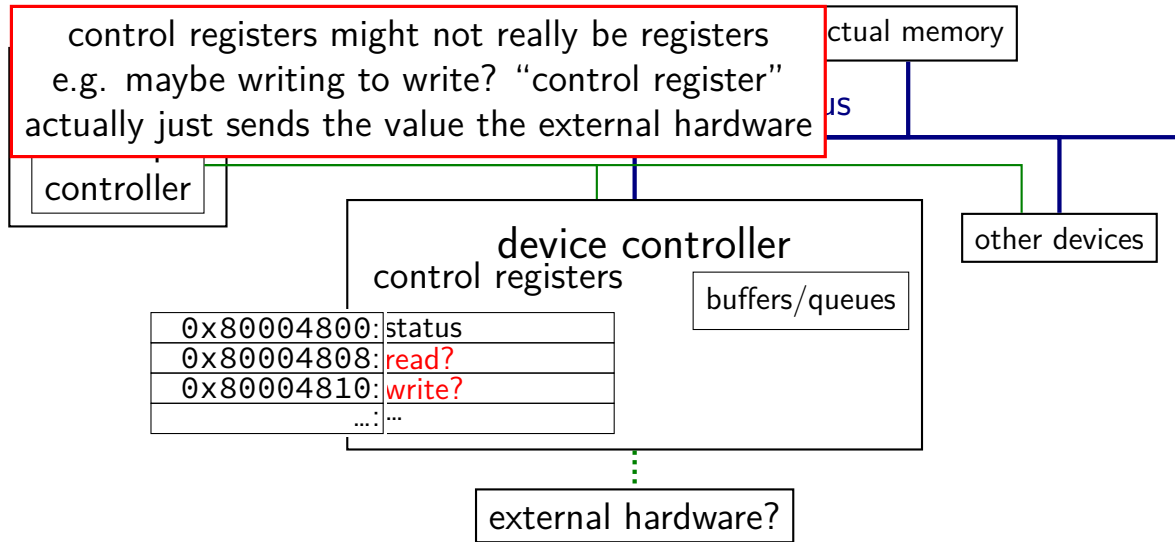


connecting devices

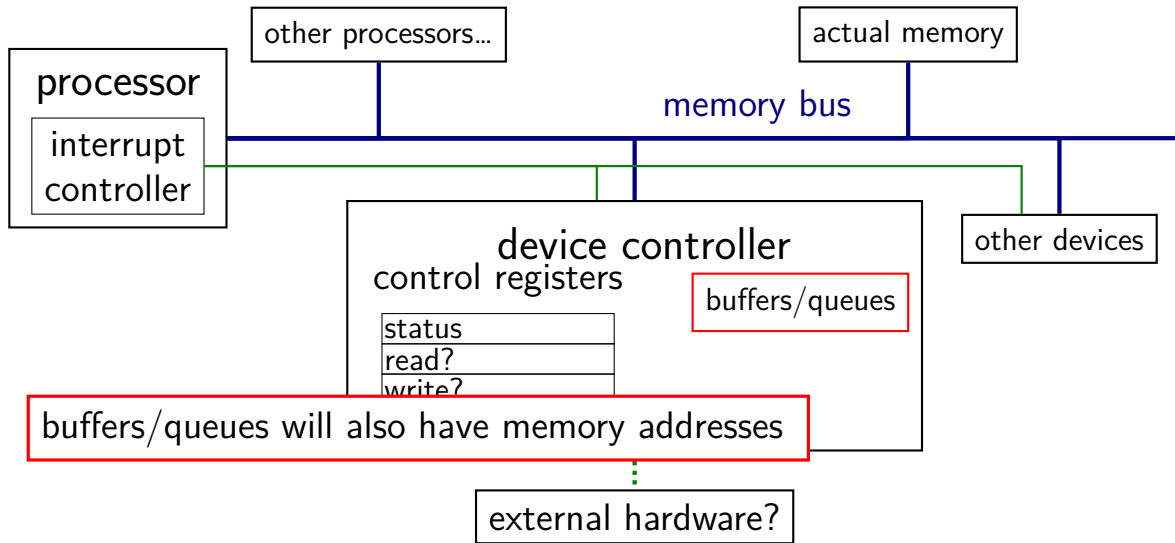


connecting devices

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



connecting devices



connecting devices

