

last time

branch prediction

- replace idle cycles with trying guess

- worst case: just as slow as waiting for branch

- squashing

making predictions

- static: forwards-not-taken, backwards-taken strategy

- dynamic: table with historical results

- typically lookup by hash of jump address

- special predictor for ret (store recent return addresses from call)

missing topic: connecting processors + devices

talked about how individual processors work

but no place to communicate with I/O devices, other CPUs

how do we do that?

individual computers are networks

individual computers are (kinda) networks of...

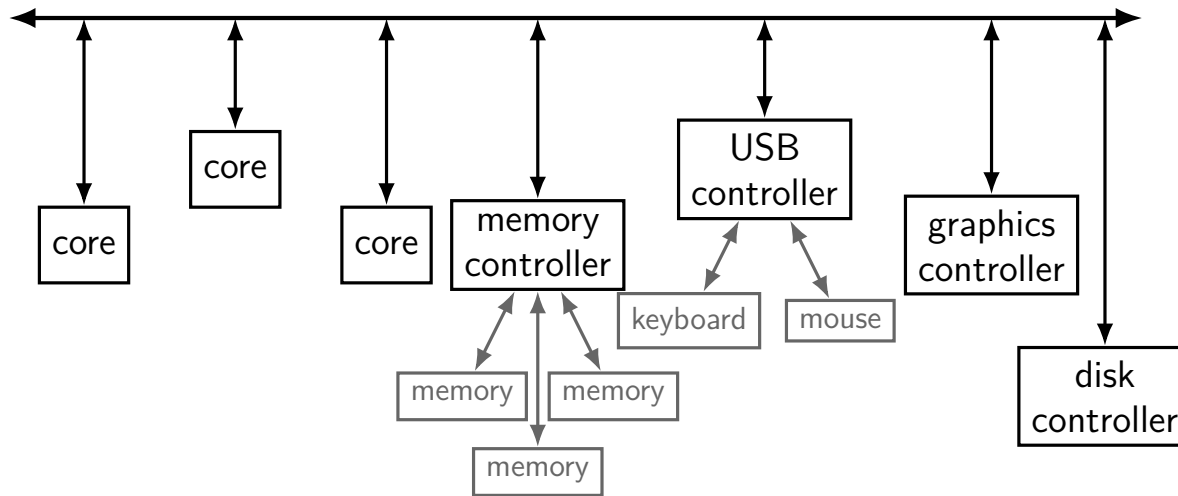
- processors

- 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)

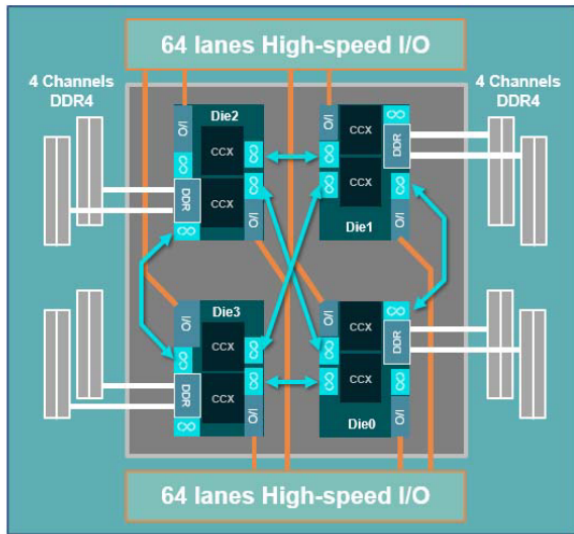


Fig. 21. Single-socket AMD EPYC™ system (SP3).

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

example: Intel Skylake-SP

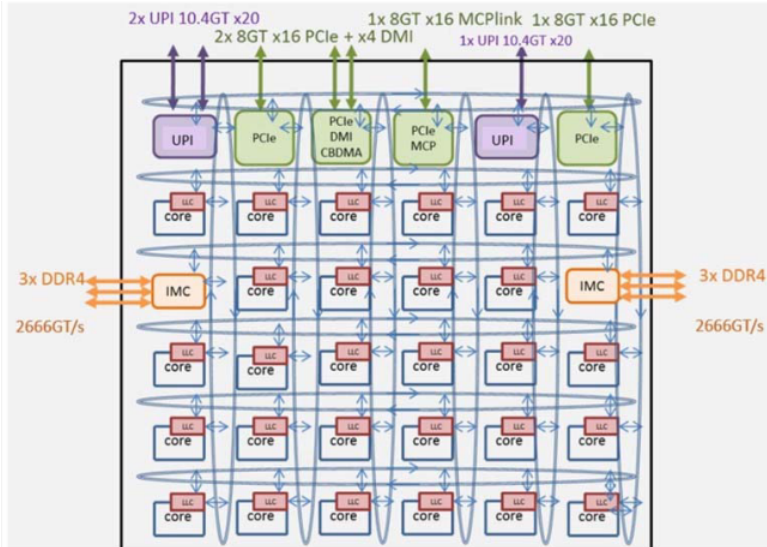
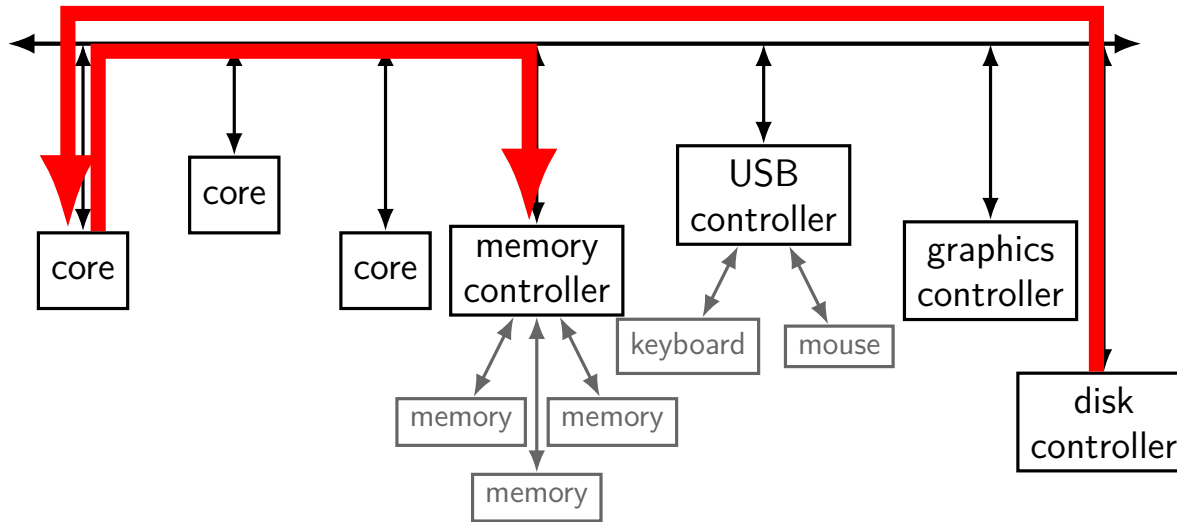
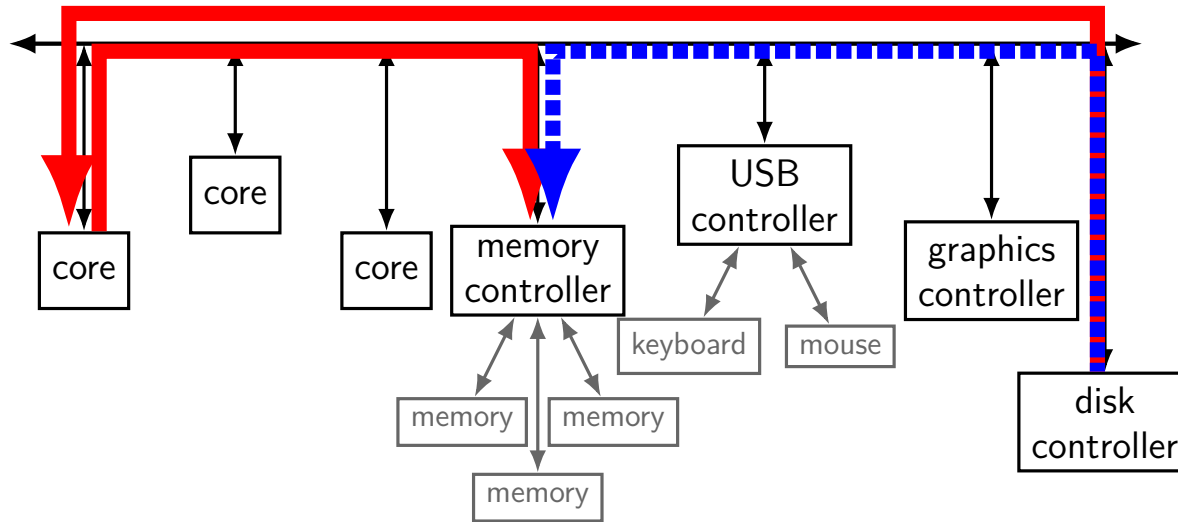


Figure from Tam et al, "SkyLake-SP: A 14nm 28-Core Xeon® Processor" (ISSCC 2018)

extra trips to CPU

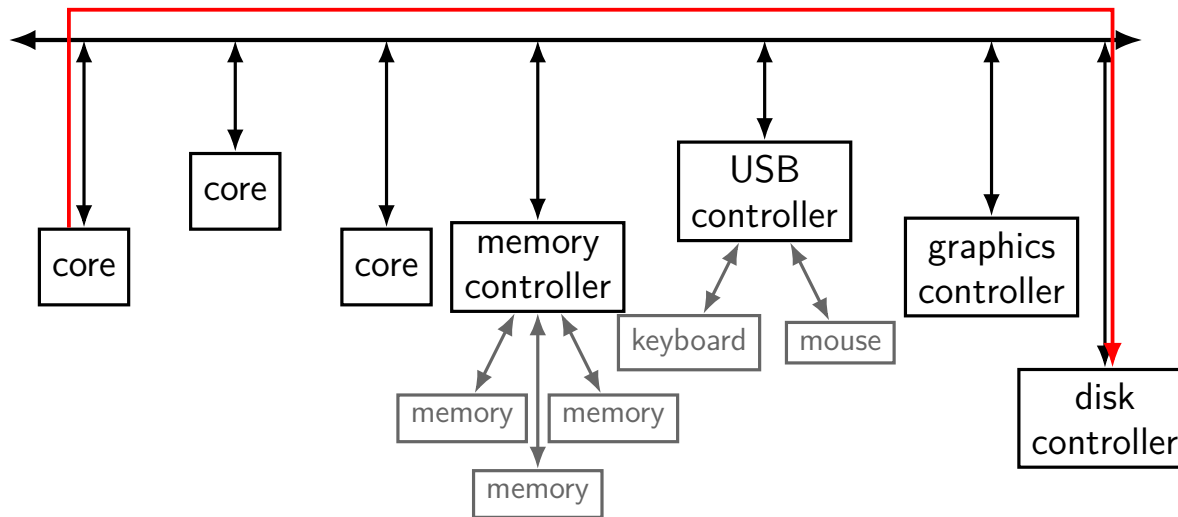


extra trips to CPU



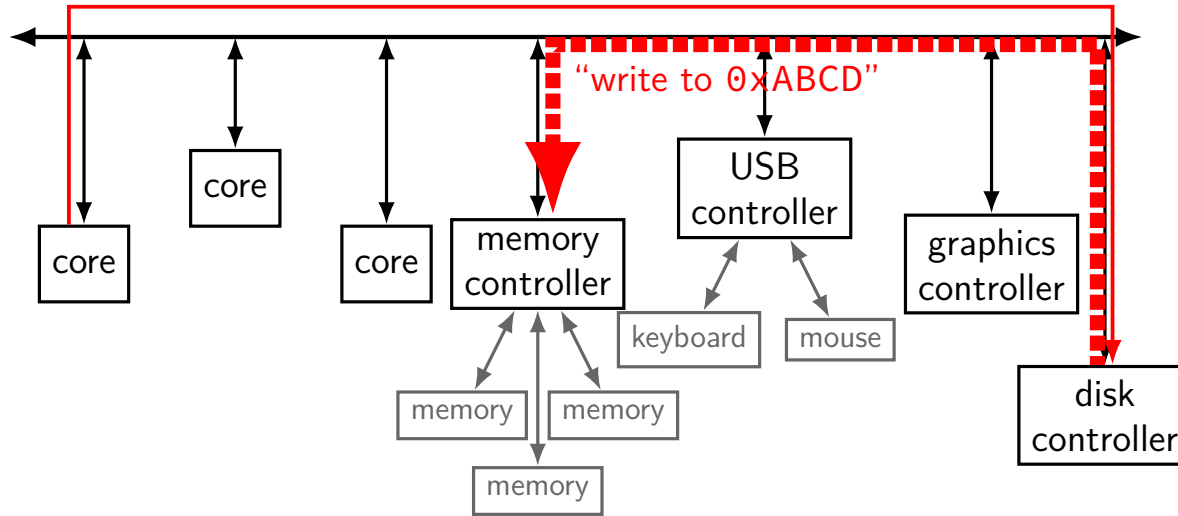
DMA

“place data at 0xABCD”



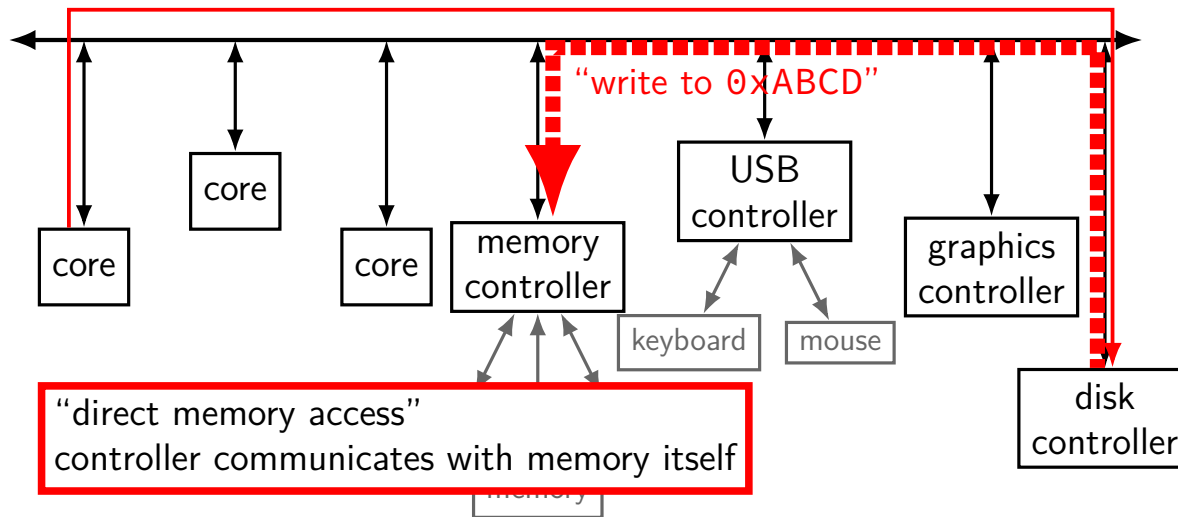
DMA

“place data at 0xABCD”



DMA

“place data at 0xABCD”



instruction-level parallelism

with pipelining: ran multiple instructions at once

but started/finished at most one at a time

and one slow instruction can slow everything down

we can often do better

beyond pipelining: multiple issue

start **more than one instruction/cycle**

multiple parallel pipelines; many-input/output register file

hazard handling much more complex

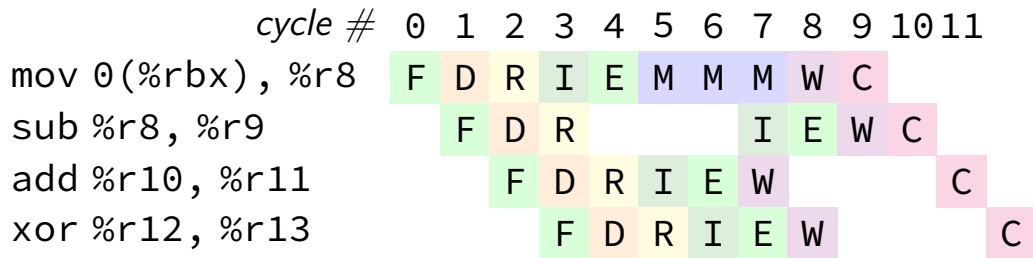
	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %r8, %r9		F	D	E	M	W				
subq %r10, %r11		F	D	E	M	W				
xorq %r9, %r11			F	D	E	M	W			
subq %r10, %rbx			F	D	E	M	W			
...										

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

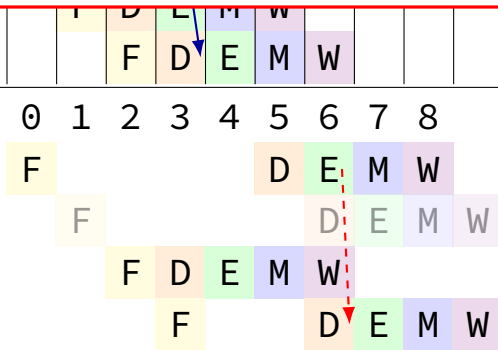
cycle # 0 1 2 3 4 5 6 7 8

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

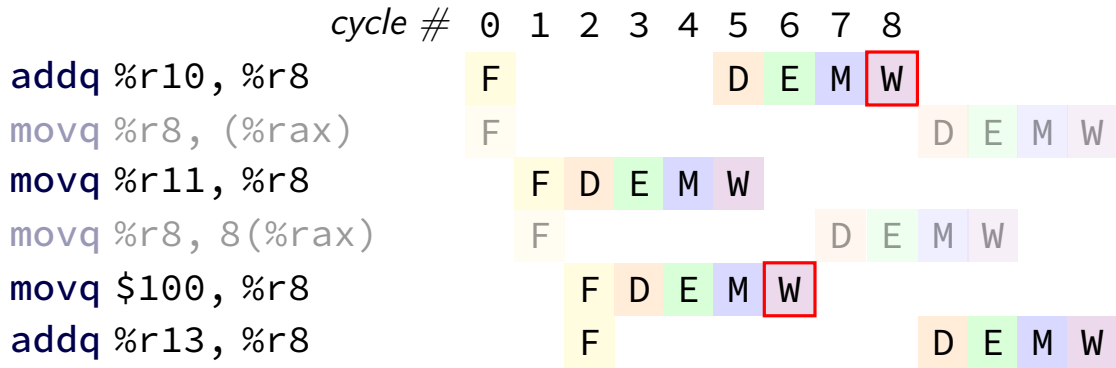
represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

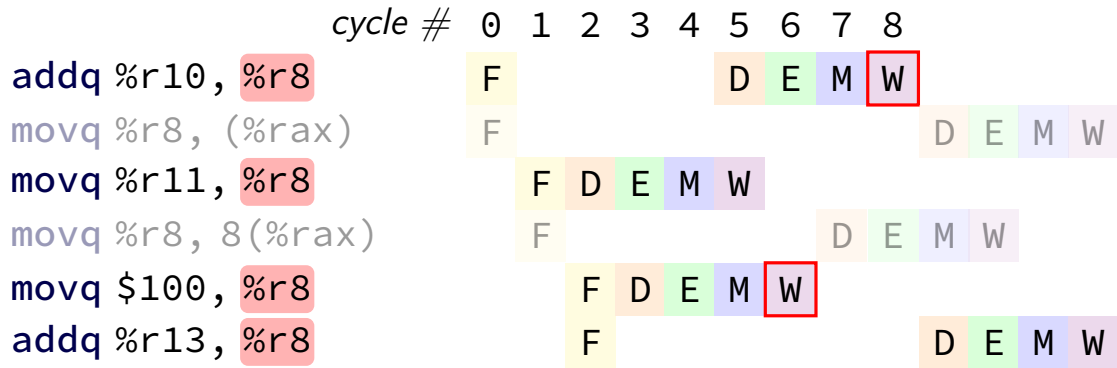
for now: version numbers

later: something simpler to implement

write-after-write example



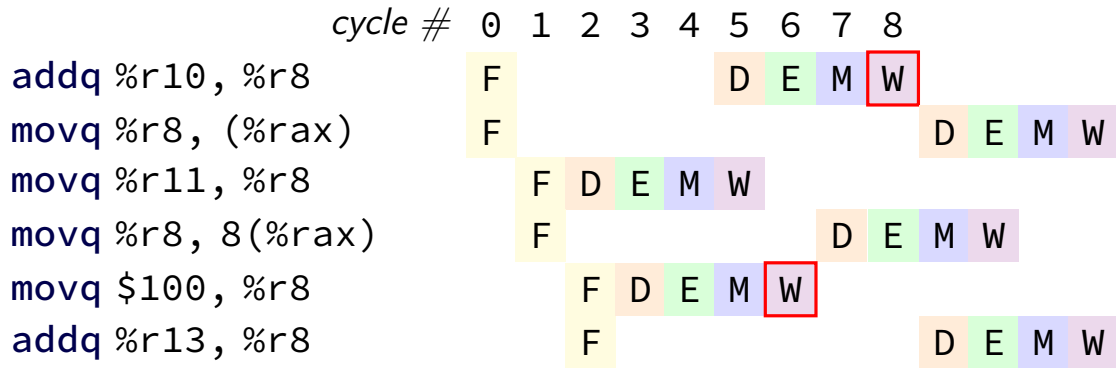
write-after-write example



out-of-order execution:

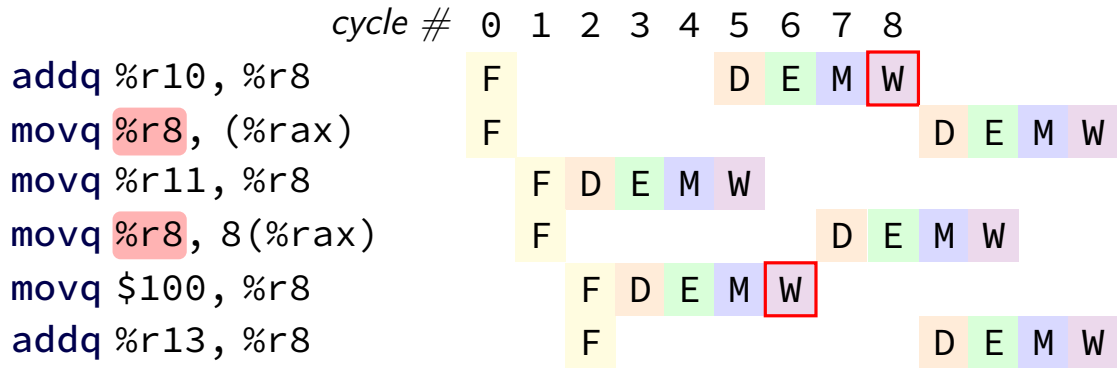
if we don't do something, newest value could be overwritten!

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

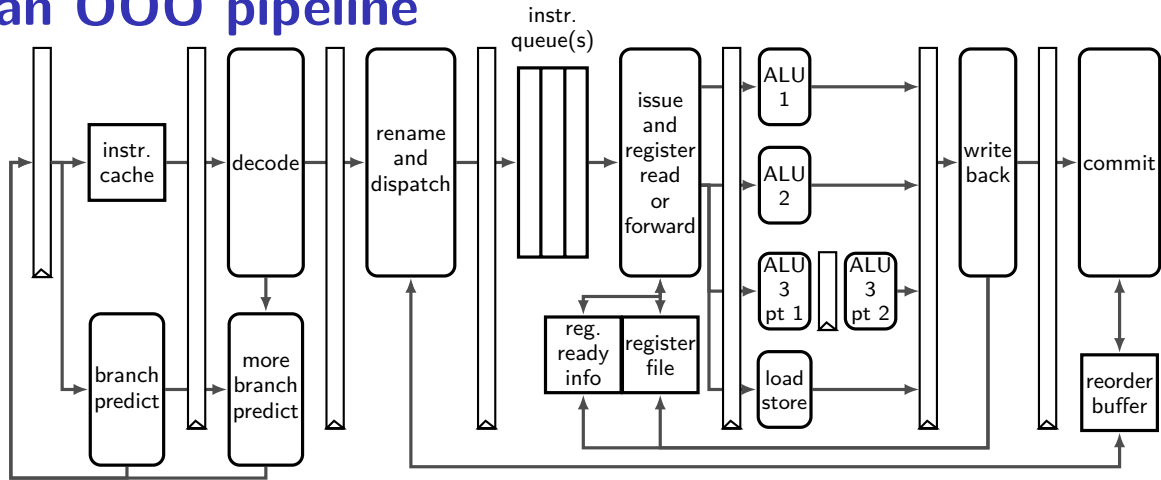
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

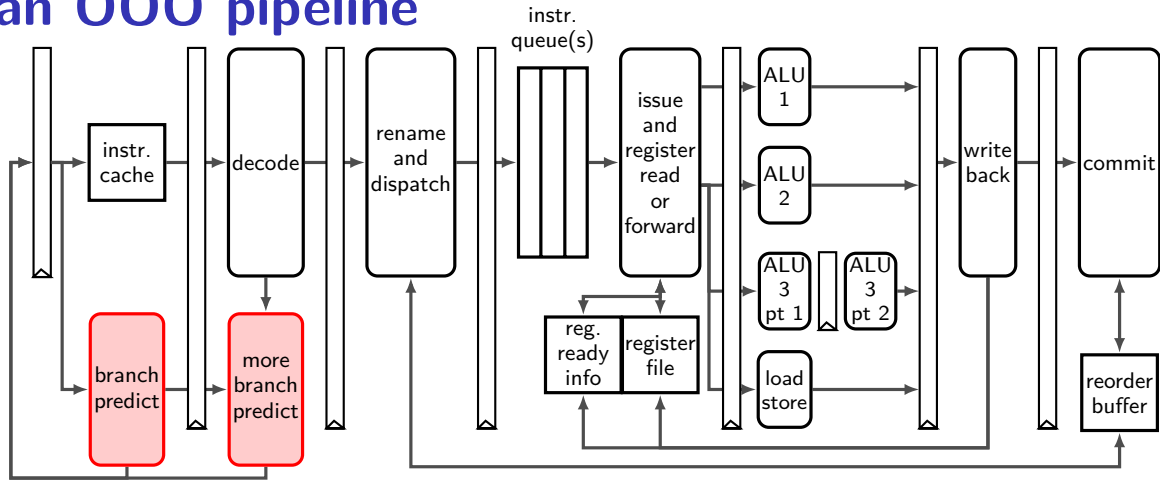
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

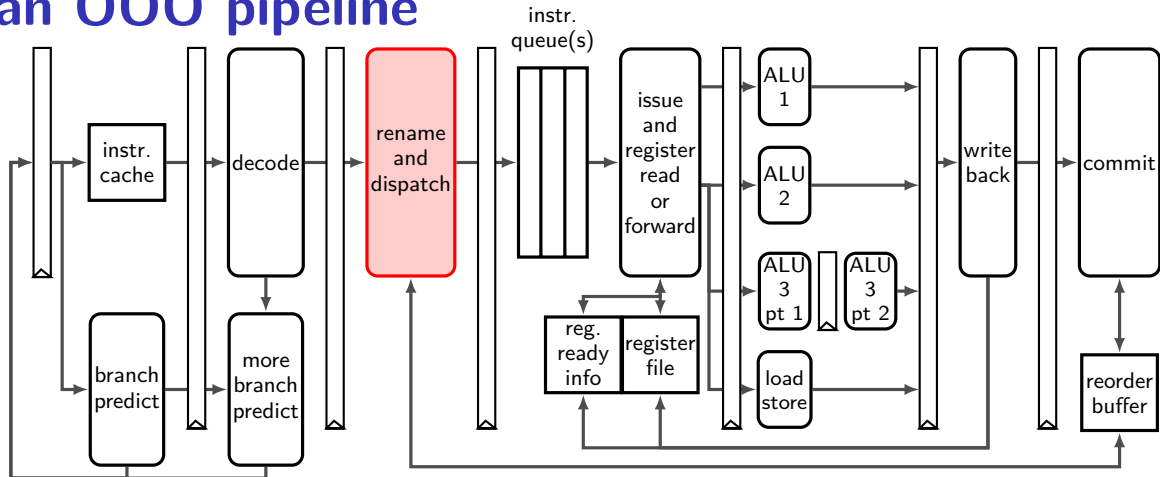


an OOO pipeline



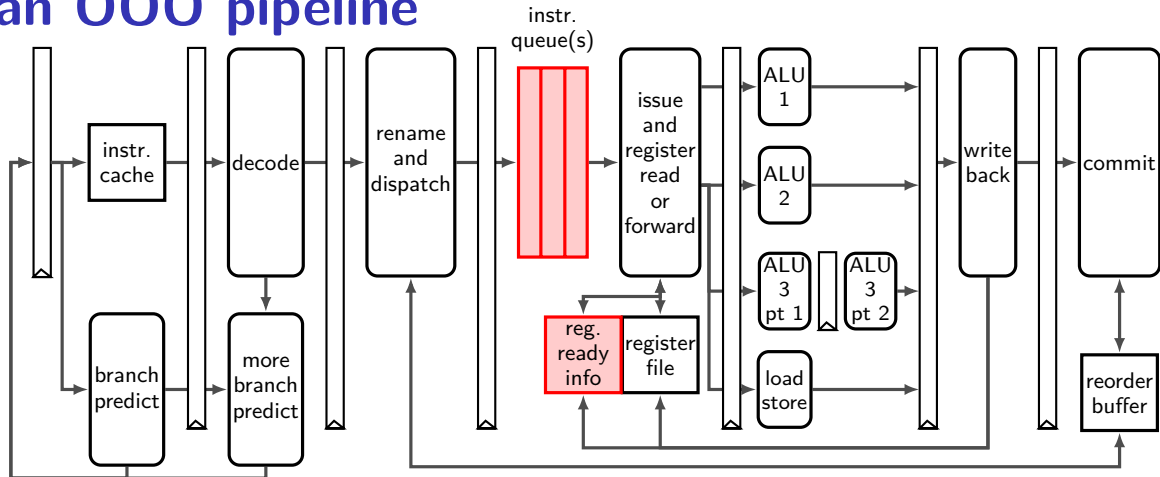
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



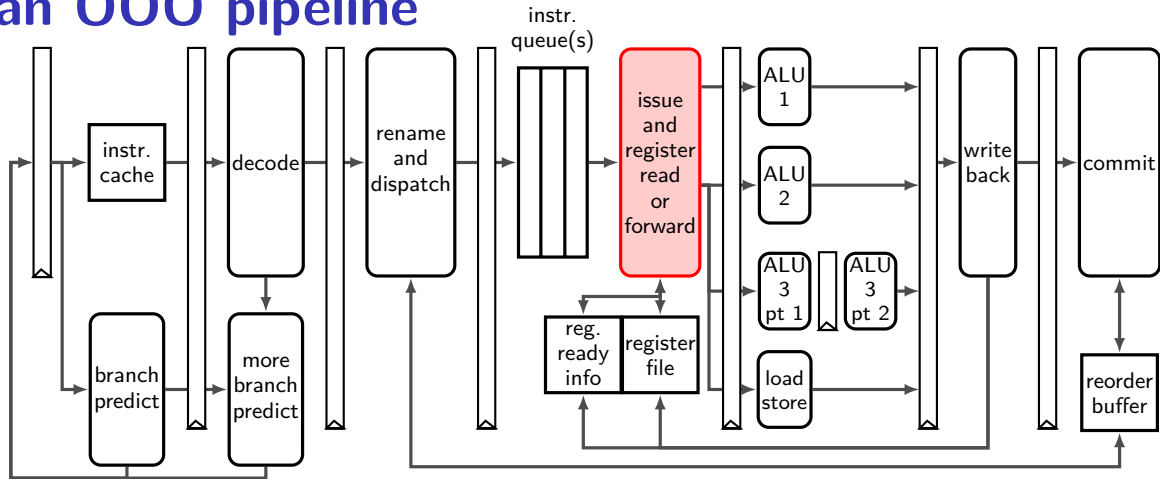
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



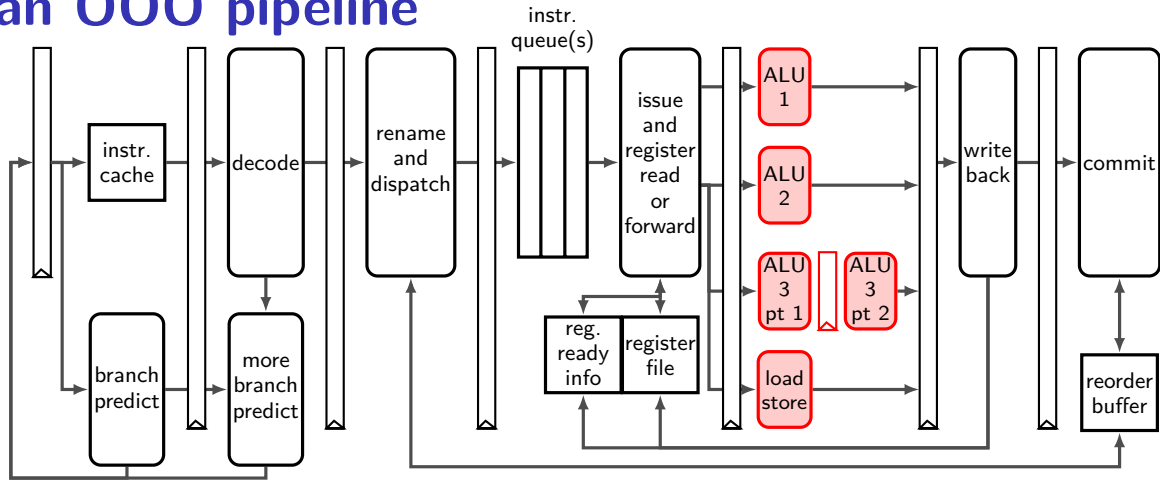
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



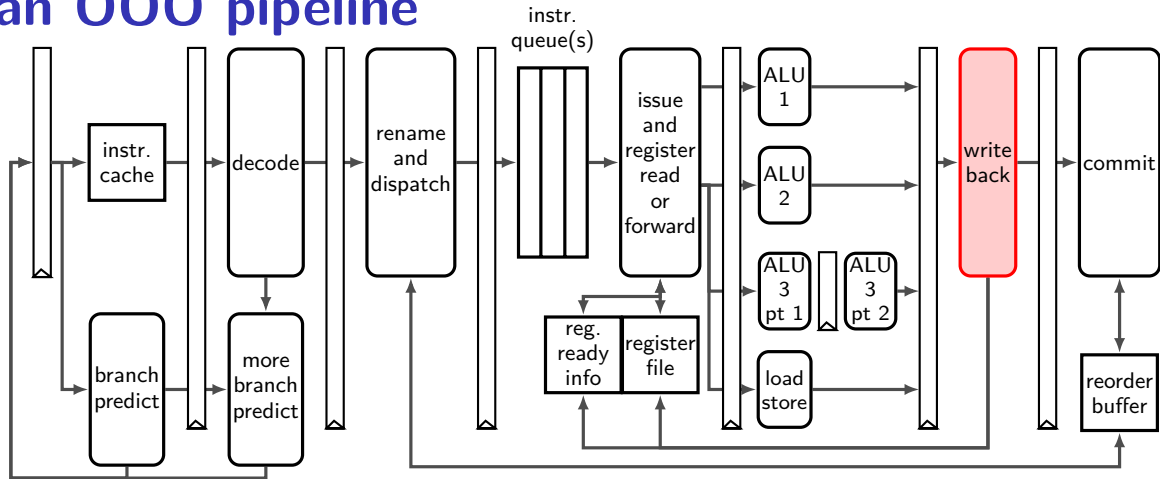
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

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...)

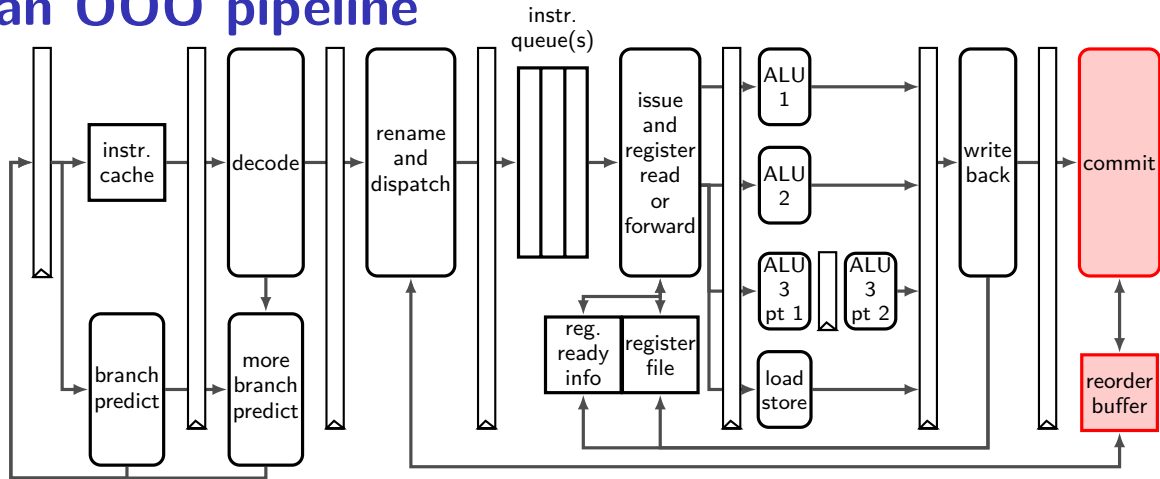
an OOO pipeline



writeback results to physical registers

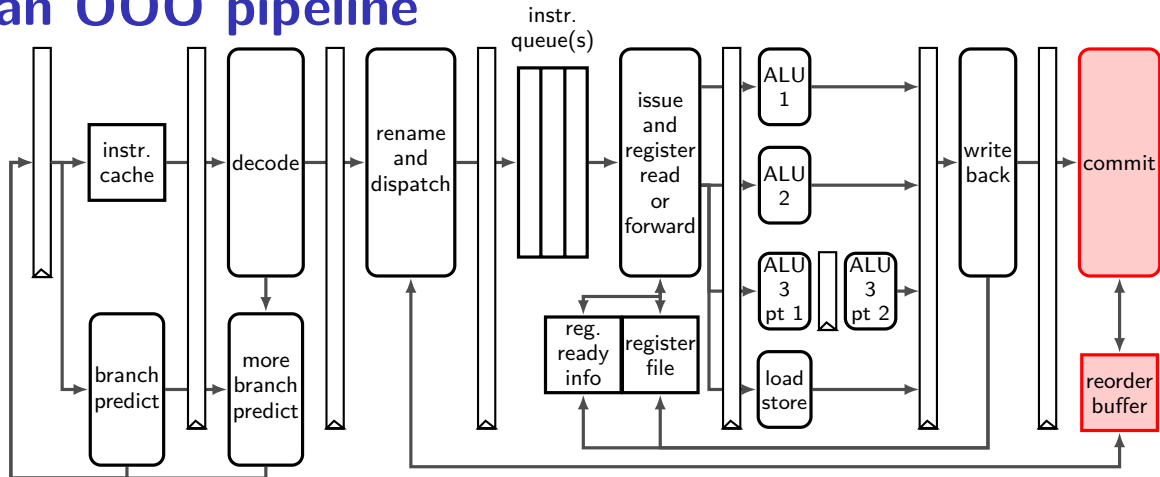
register file: typically support writing 3+ registers at a time

an OOO pipeline



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

an OOO pipeline

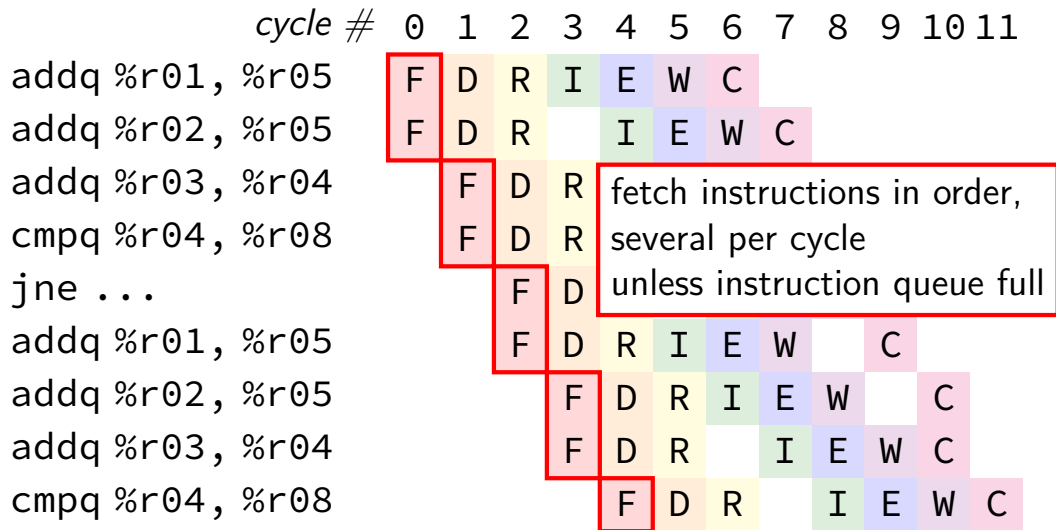


commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.

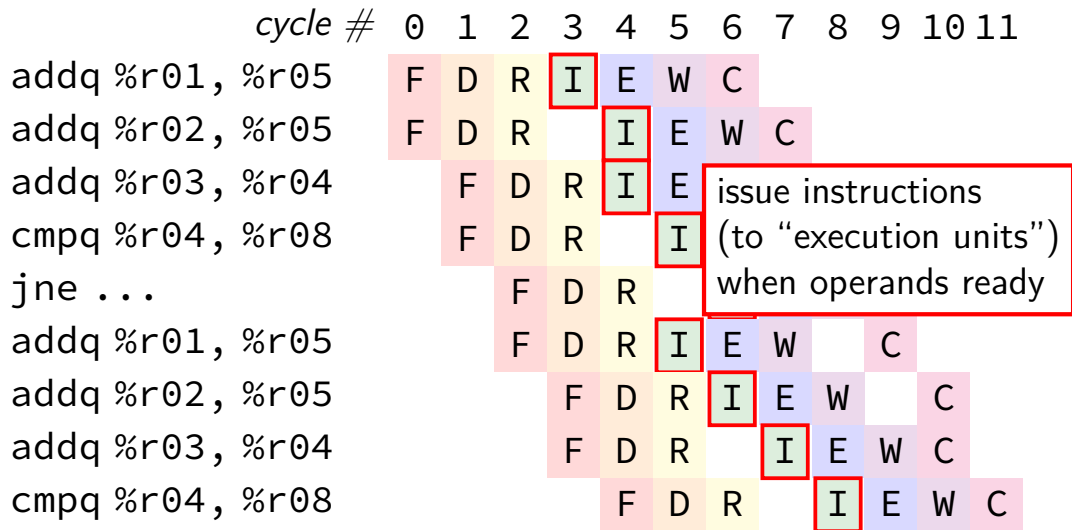
an OOO pipeline diagram

	<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

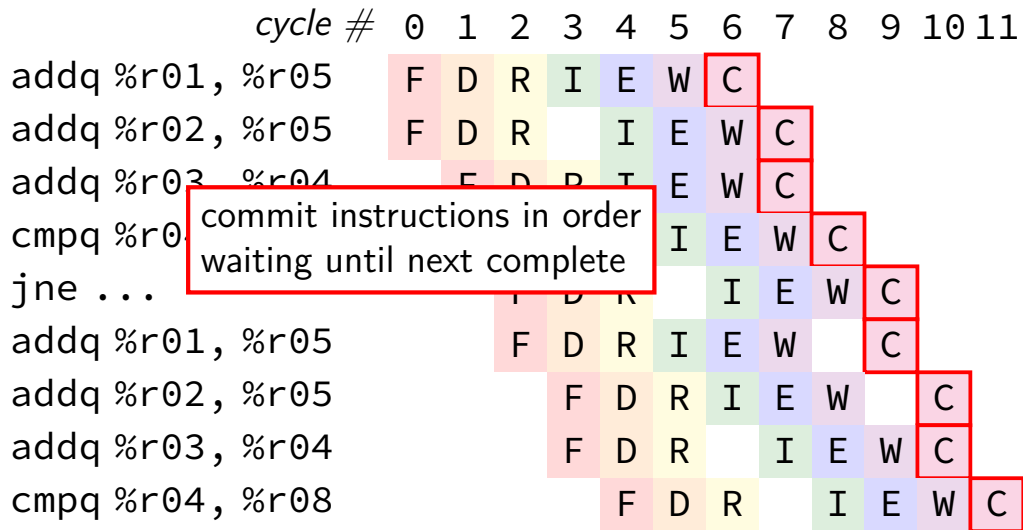
an OOO pipeline diagram



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
0x3FFF8:   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

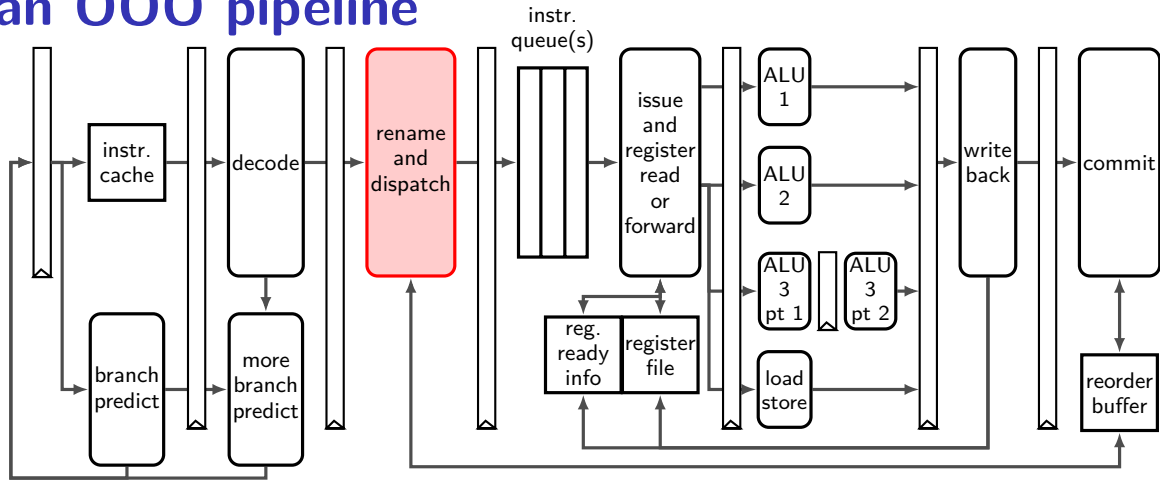
example—predict based on other recent computed jumps

good for polymorphic method calls

table lookup with `Hash(last few jmps)`

instead of `Hash(this jmp)`

an OOO pipeline



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

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

renamed

arch → phys register map

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

free reg list

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

register renaming example (1)

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

renamed
add %x19, %x13 → %x18

arch → phys register map

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

free reg list

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

register renaming example (1)

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

arch → phys register map

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

free reg list

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

register renaming example (1)

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

arch → phys register map

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

free reg list

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

register renaming example (1)

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

arch → phys register map

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

free reg list

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

register renaming example (2)

original

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

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

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

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

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

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)
subq %x18, %x07 → %x20
movq 8(%x20), (memory) → %x21
```

arch → phys register map

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

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)
subq %x18, %x07 → %x20
movq 8(%x20), (memory) → %x21
movq $100 → %x23
```

arch → phys register map

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

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

free
regs

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

register renaming exercise

original

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

arch → phys

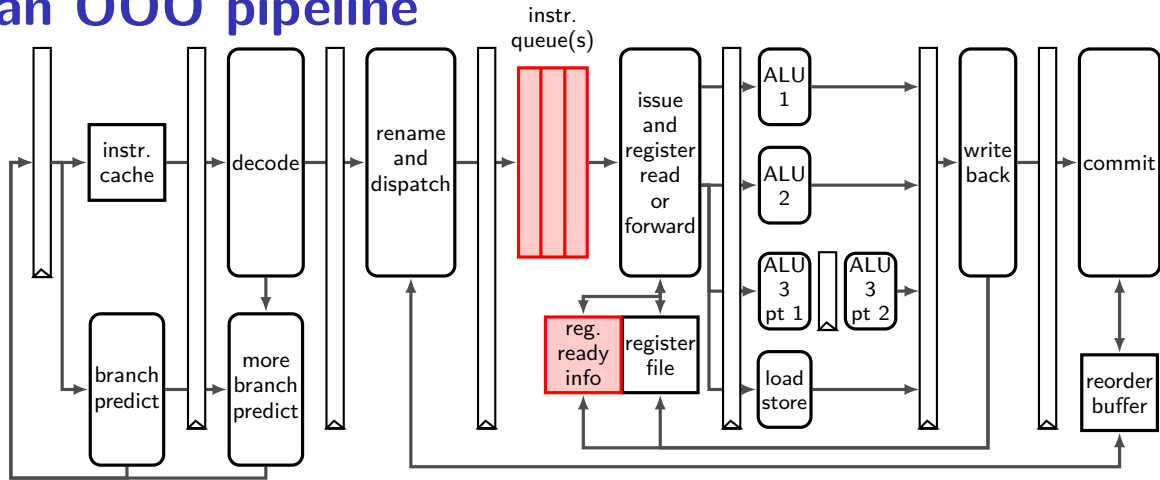
%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x29
%r12	%x05
%r13	%x02
...	...

renamed

free
regs

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

an OOO pipeline



instruction queue and dispatch

instruction queue

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

... ...

execution unit

ALU 1

ALU 2

scoreboard

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

...

instruction queue and dispatch

instruction queue

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

... ..

execution unit *cycle#* 1

ALU 1 **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

... ...

execution unit cycle# 1

ALU 1 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

... ...

execution unit cycle# 1

ALU 1 1

ALU 2

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 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

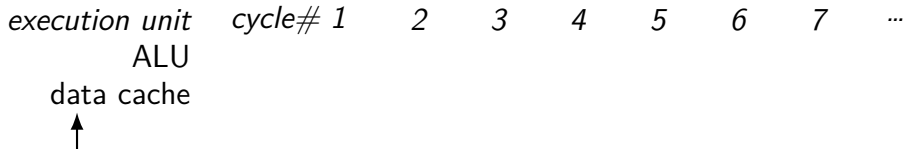
instruction queue

#	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	
...	...



register renaming: missing pieces

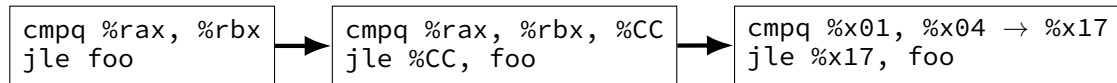
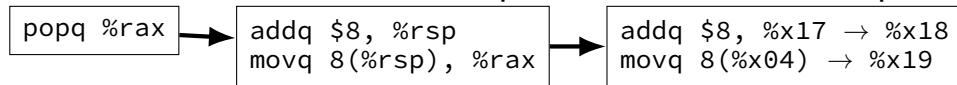
what about “hidden” inputs like `%rsp`, condition codes?

one solution: translate to instructions with additional register parameters

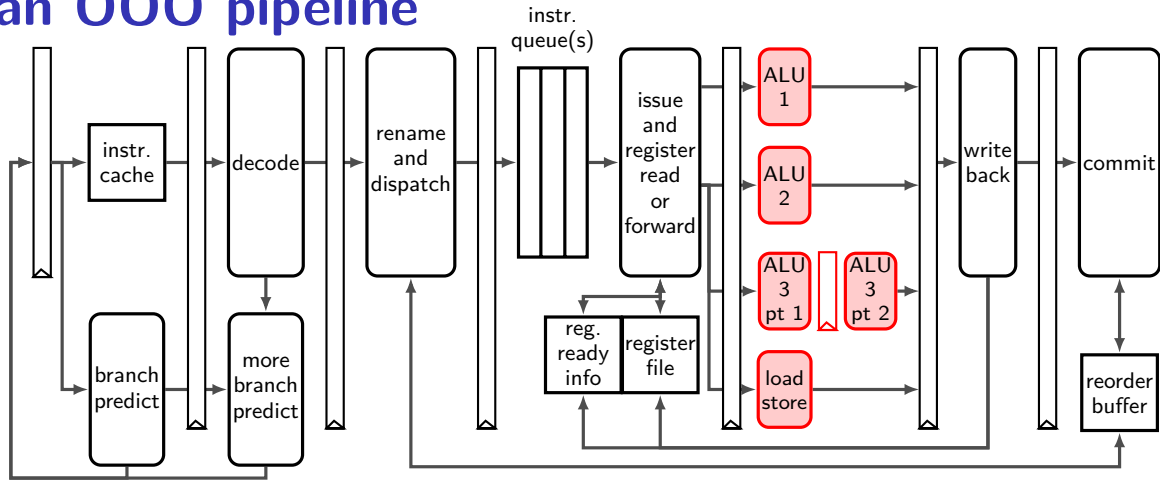
- making `%rsp` explicit parameter

- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones



an OOO pipeline



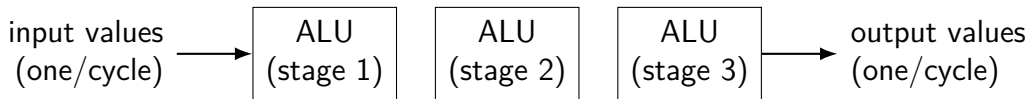
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



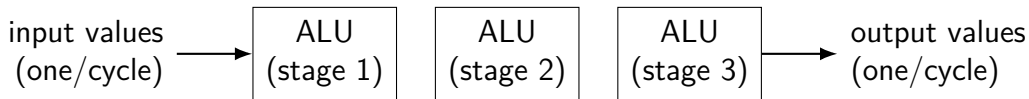
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



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

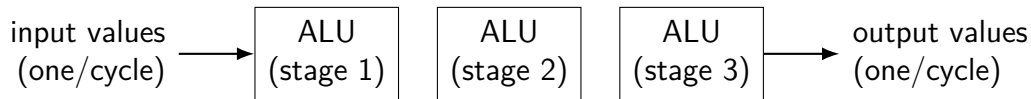
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



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

3×3 cycles + any time to forward values

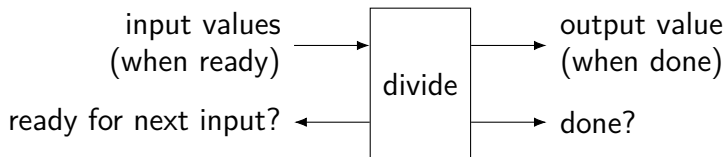
no parallelism!

execution units AKA functional units (2)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:



instruction queue and dispatch (multicycle)

instruction queue

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

... ...

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

scoreboard

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

...

instruction queue and dispatch (multicycle)

instruction queue

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

... ...

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

scoreboard

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

...

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

execution unit **cycle# 1**

ALU 1 (add, cmp, jxx) **1**

ALU 2 (add, cmp, jxx) **—**

ALU 3 (mul) start **2**

scoreboard

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

...

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

...

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

...

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

...

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

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

OOO limitations

- can't always find instructions to run

 - plenty of instructions, but all depend on unfinished ones

 - programmer can adjust program to help this

- need to track all uncommitted instructions

 - can only go so far ahead

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

- branch misprediction has a big cost (relative to pipelined)

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

OOO limitations

can't always find instructions to run

plenty of instructions, but all depend on unfinished ones

programmer can adjust program to help this

need to track all uncommitted instructions

can only go so far ahead

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

branch misprediction has a big cost (relative to pipelined)

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

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

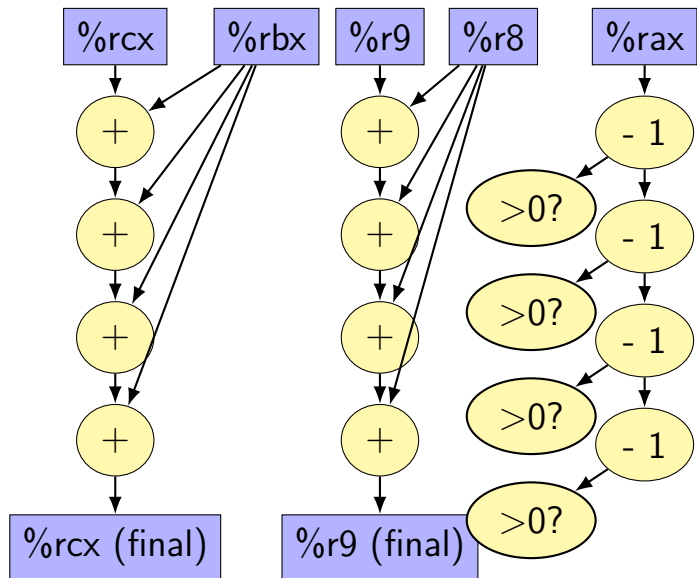
```
example1:  
    movq $1000000000000, %rax  
loop1:  
    addq %rbx, %rcx  
    decq %rax  
    jge loop1  
    ret
```

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

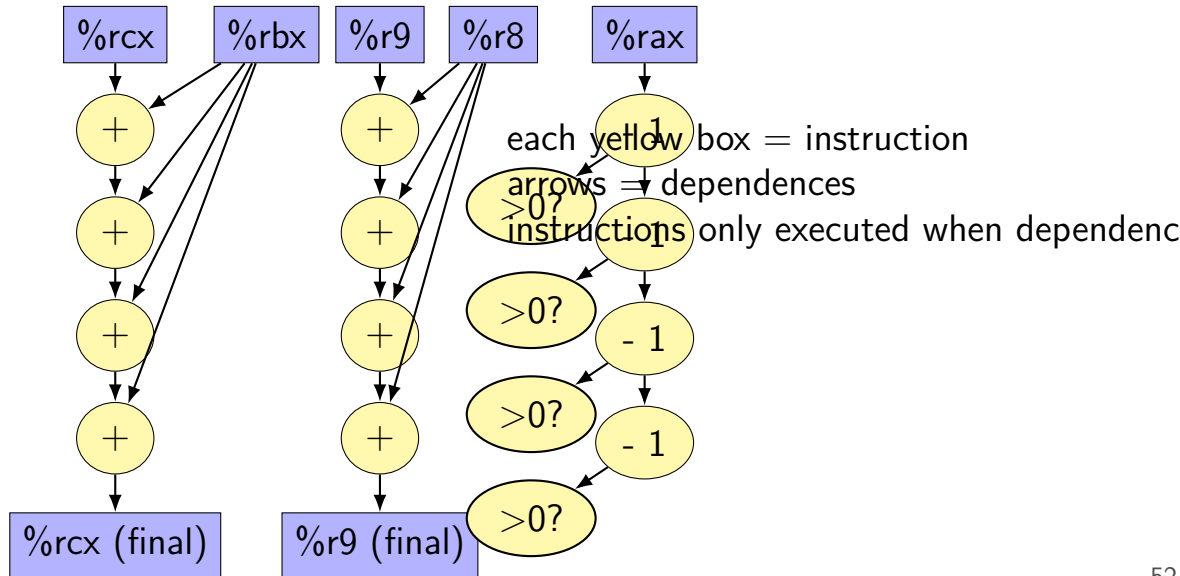
data flow model and limits (1)



loop2:

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

data flow model and limits (1)



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
```

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

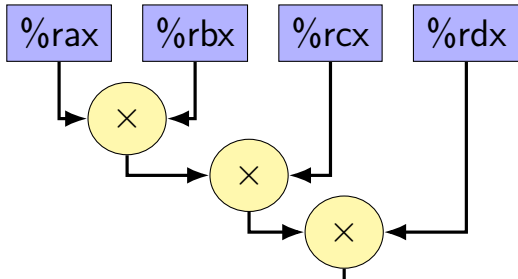
```
imulq %rbx, %rax  
imulq %rcx, %rdx  
imulq %rdx, %rax
```

reassociation

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

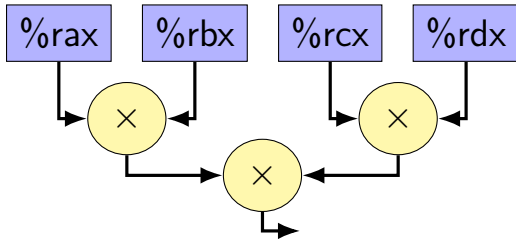
$$((a \times b) \times c) \times d$$

```
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
```

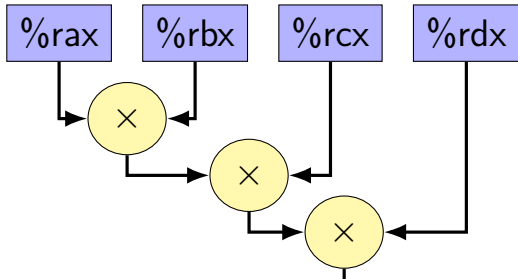


reassociation

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

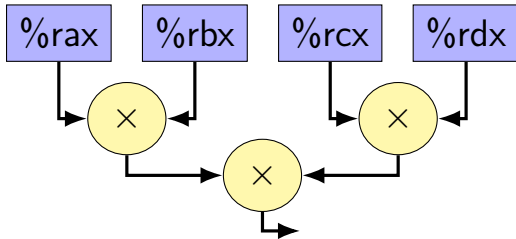
$$((a \times b) \times c) \times d$$

```
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
```



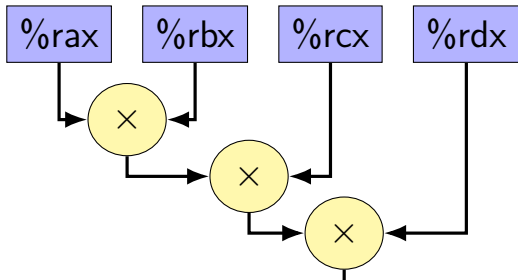
reassociation

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
```

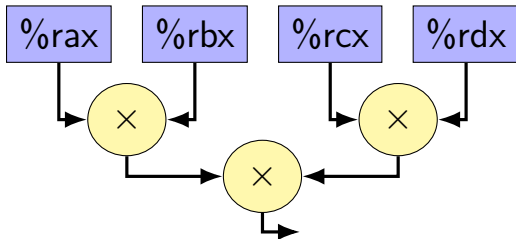
15
cycles



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

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

11
cycles

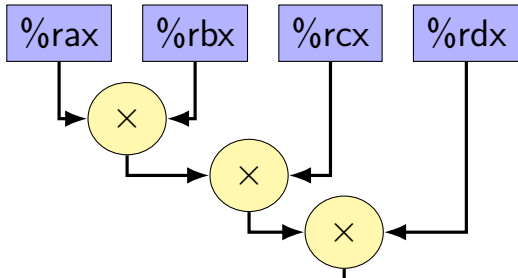


reassociation

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

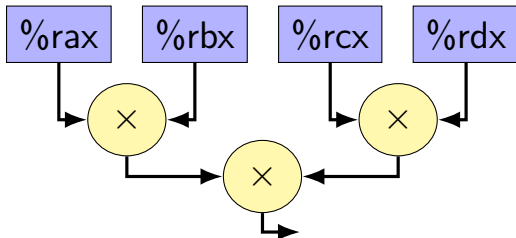
$$((a \times b) \times c) \times d$$

```
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

backup slides

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 `jg`?

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

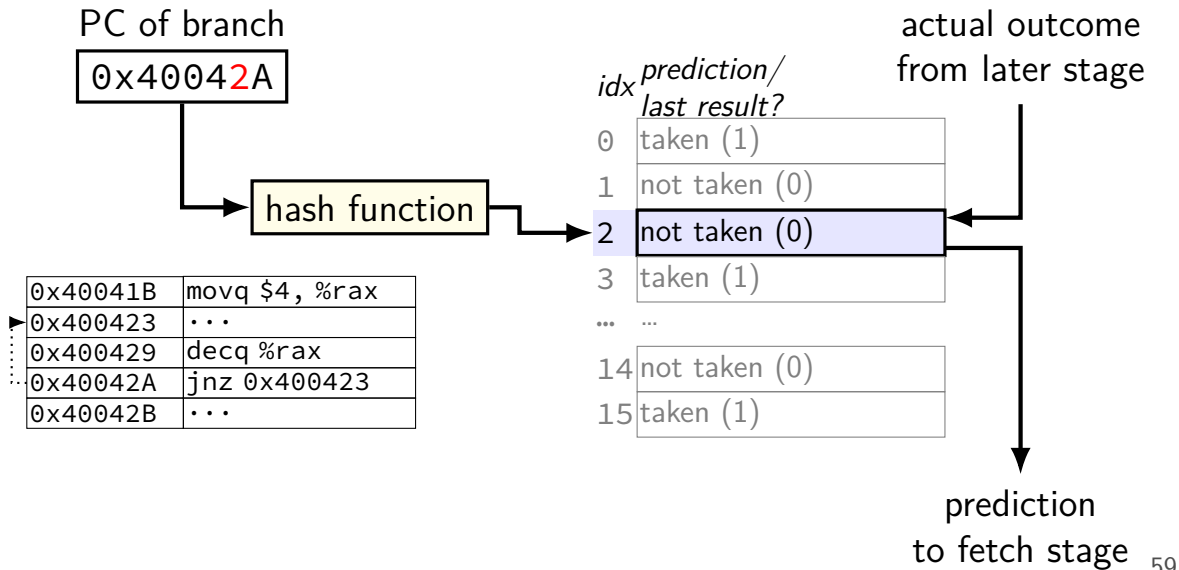
*index prediction/
last result?*

0	taken (1)
1	not taken (0)
2	taken (1)
3	taken (1)
...	...
14	not taken (0)
15	taken (1)

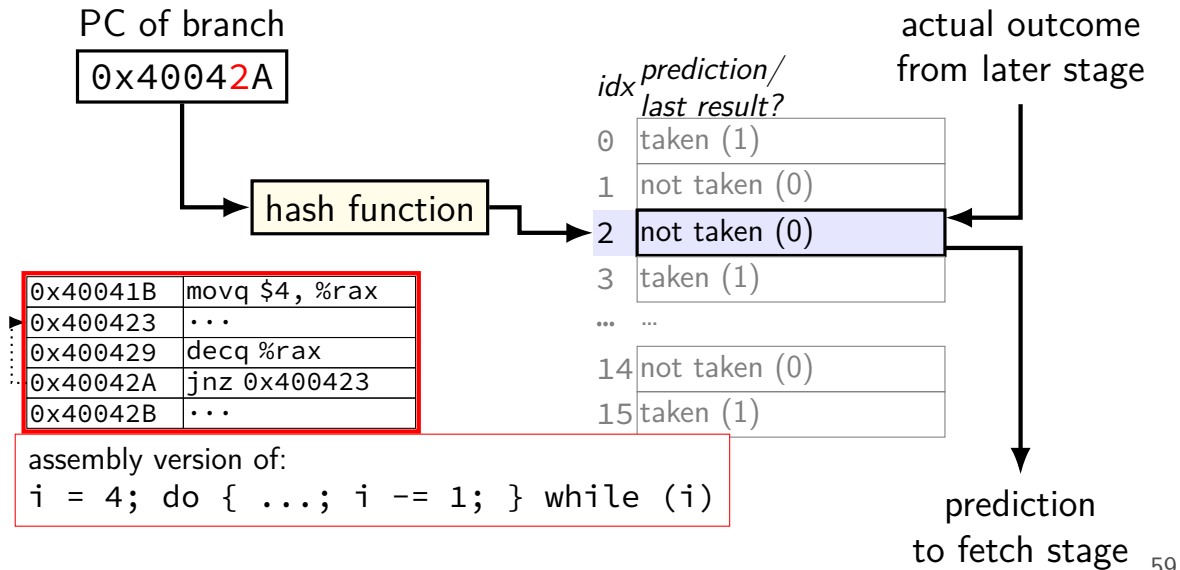
actual outcome
(from later stage)

prediction
to fetch stage

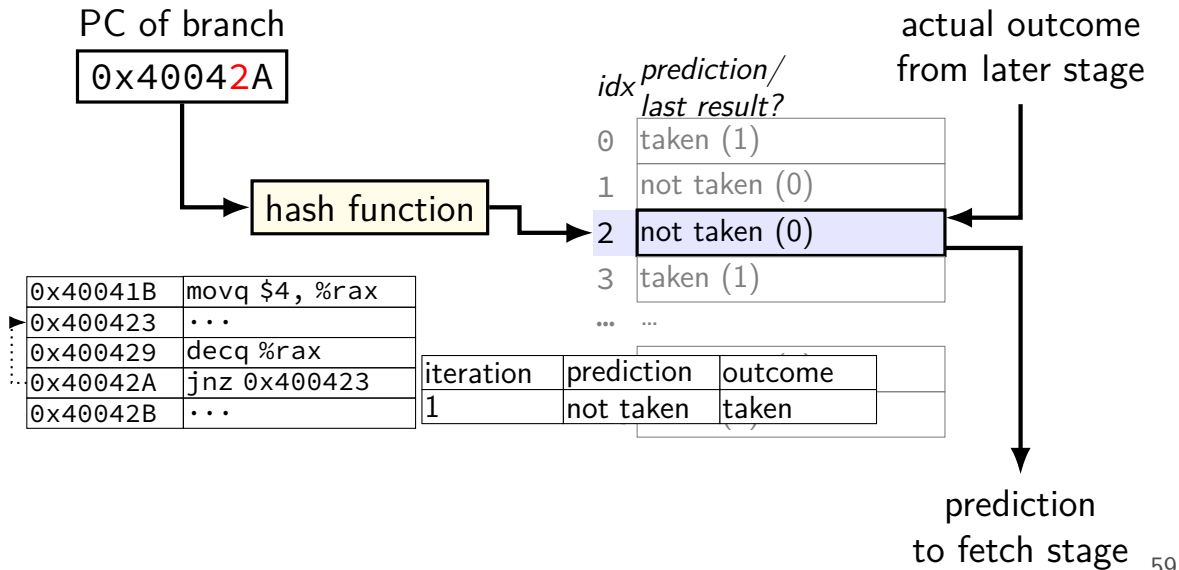
example



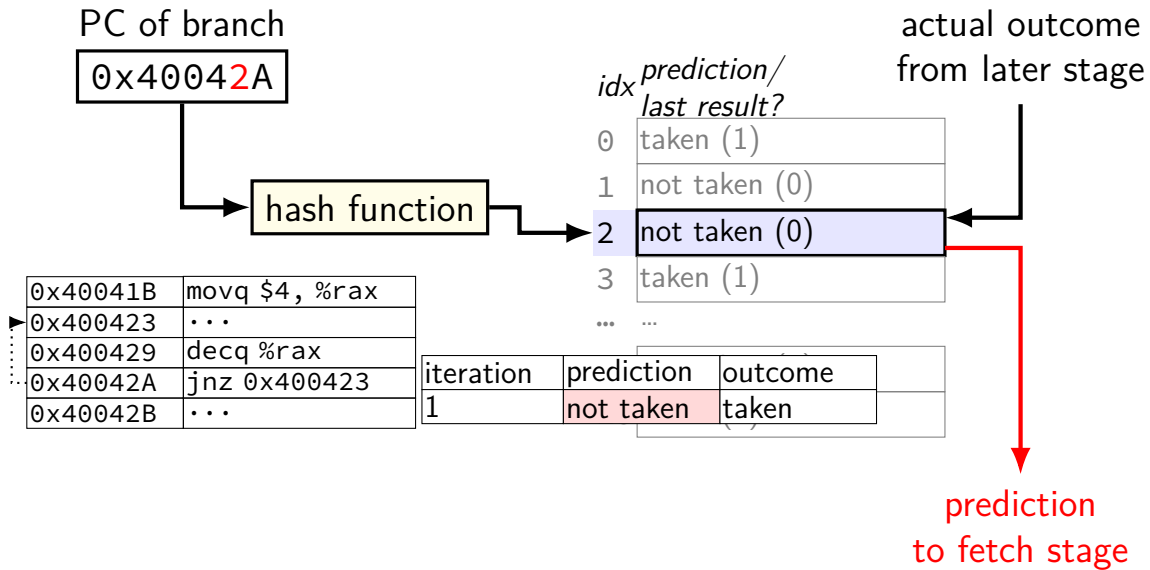
example



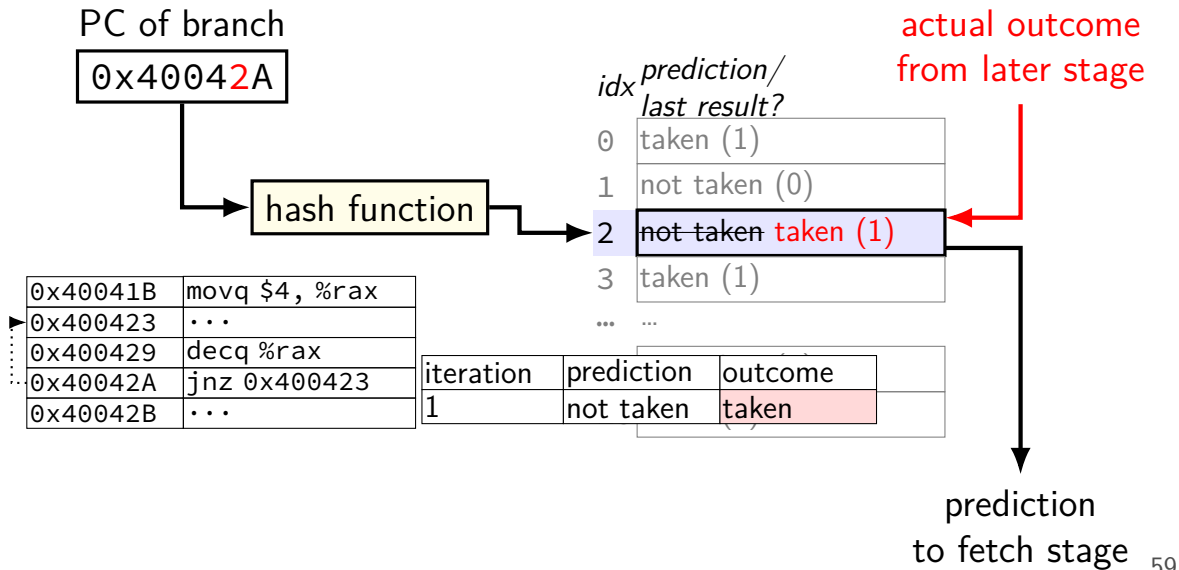
example



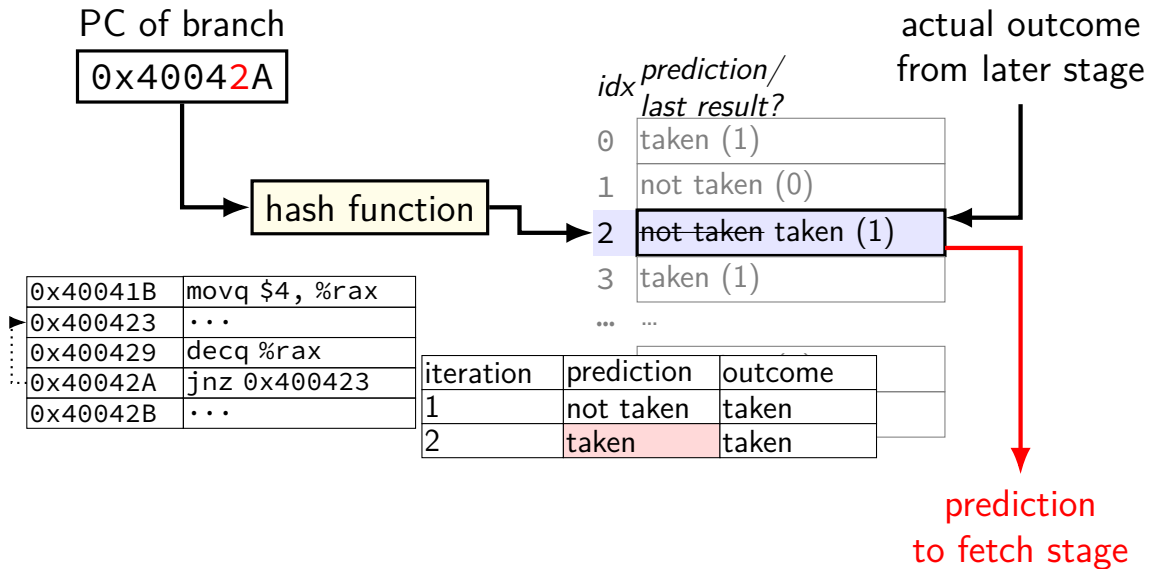
example



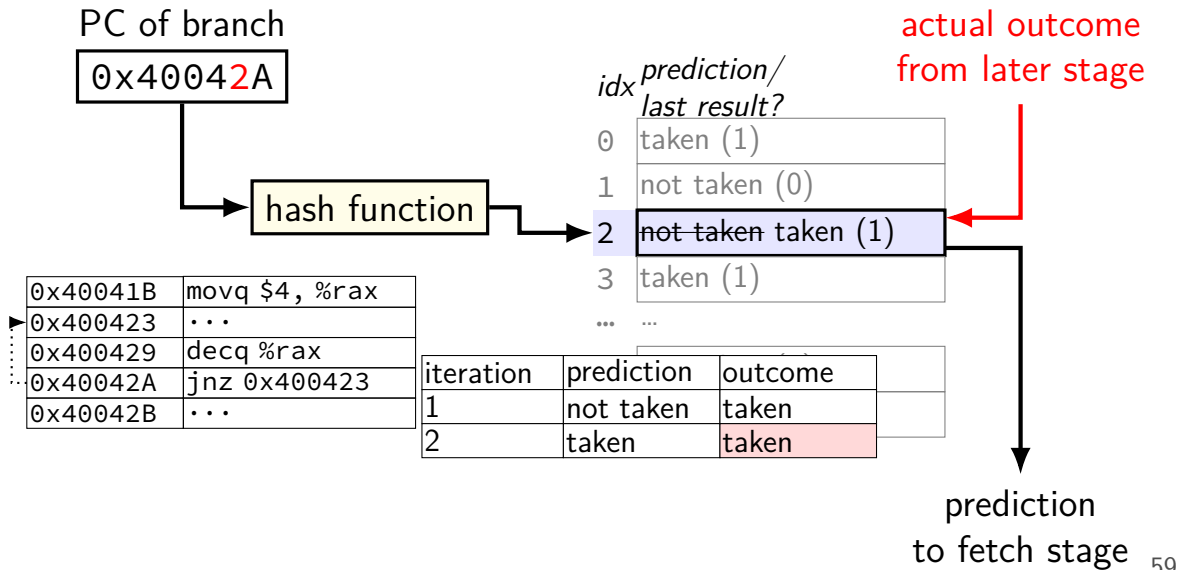
example



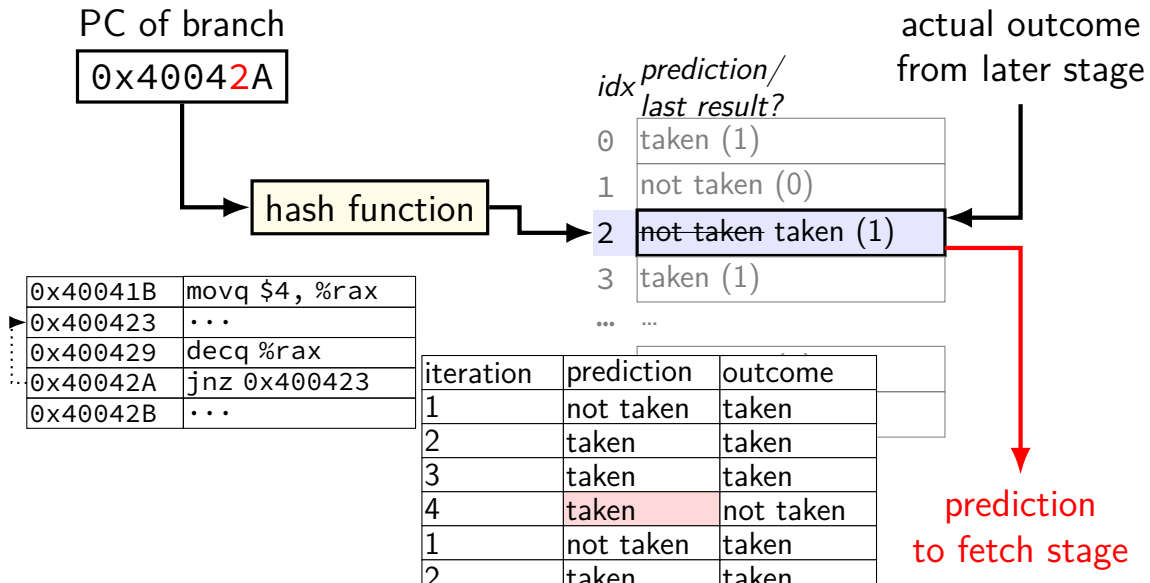
example



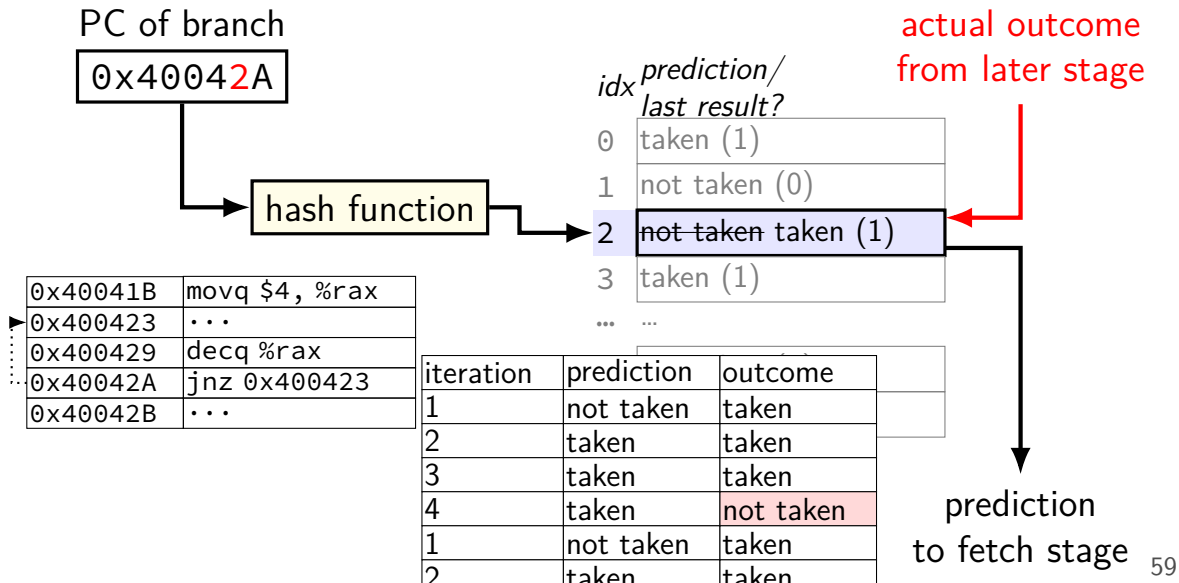
example



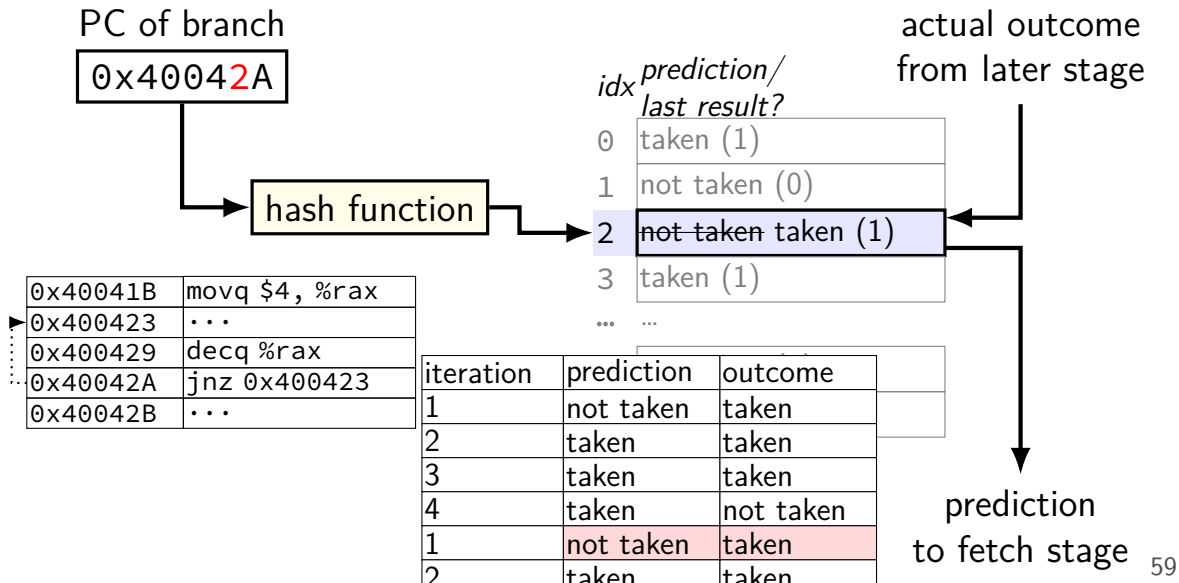
example



example



example



collisions?

two branches could have same hashed PC

nothing in table tells us about this

versus direct-mapped cache: had *tag bits* to tell

is it worth it?

adding tag bits makes table *much* larger and/or slower

but does anything go wrong when there's a collision?

collision results

possibility 1: both branches usually taken

no actual conflict — prediction is better(!)

possibility 2: both branches usually not taken

no actual conflict — prediction is better(!)

possibility 3: one branch taken, one not taken

performance probably worse

1-bit predictor for loops

predicts first and last iteration wrong

example: branch to beginning — but same for branch from beginning to end

everything else correct

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;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
...
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	
1	break	T	N	

mod 3: correct for i=2,5,8,...,49 (16/50)
break: correct for i=2,3,...,48 (48/50)
overall: 64/100

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

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
...
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	
1	break	T	N	

mod 3: correct for i=2,5,8,...,49 (16/50)
break: correct for i=2,3,...,48 (48/50)
overall: 64/100

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

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
...
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	
1	break	T	N	

mod 3: correct for i=2,5,8,...,49 (16/50)
break: correct for i=2,3,...,48 (48/50)
overall: 64/100

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

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
...
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	
1	break	T	N	

mod 3: correct for i=2,5,8,...,49 (16/50)
break: correct for i=2,3,...,48 (48/50)
overall: 64/100

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


exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
...
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	
1	break	T	N	

mod 3: correct for i=2,5,8,...,49 (16/50)
break: correct for i=2,3,...,48 (48/50)
overall: 64/100

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

beyond local 1-bit predictor

can predict using more historical info

whether taken last several times

example: taken 3 out of 4 last times → predict taken

pattern of how taken recently

example: if last few are T, N, T, N, T, N; next is probably T
makes two branches hashing to same entry not so bad

outcomes of last N conditional jumps (“global history”)

take into account conditional jumps in surrounding code

example: loops with if statements will have regular patterns

predicting ret: minstack of return addresses

predicting ret — minstack in processor registers

push on minstack on call; pop on ret

minstack overflows? discard oldest, mispredict it later

baz saved registers
baz return address
bar saved registers
bar return address
foo local variables
foo saved registers
foo return address
foo saved registers

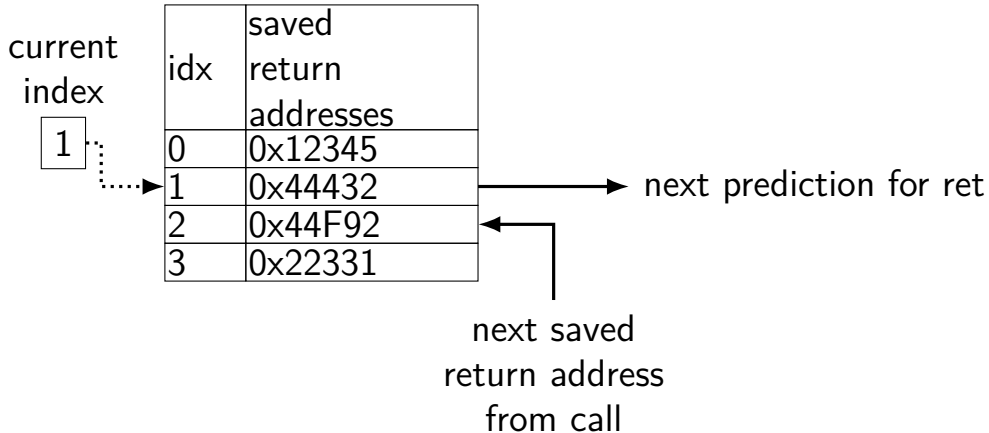
baz return address
bar return address
foo return address

(partial?) stack
in CPU registers

stack in memory

4-entry return address stack

4-entry return address stack in CPU



on call: increment index, save return address in that slot

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
0x3FFF8:   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

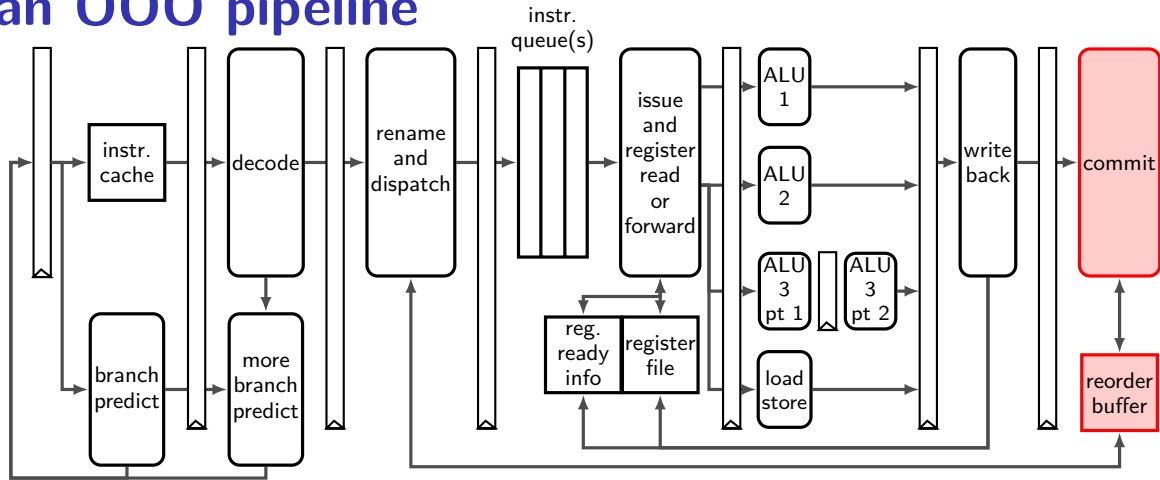
good for polymorphic method calls

table lookup with `Hash(last few jmps)`

instead of `Hash(this jmp)`

backup slides

an OOO pipeline



reorder buffer: on rename

phys \rightarrow arch. reg
for new instrs

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

free list

%x19
%x23
...
...

reorder buffer: on rename

phys → arch. reg
for new instrs

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

free list

%x19
%x23
...
...

reorder buffer (ROB)

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

reorder buffer contains instructions started,
but not fully finished new entries created on rename

reorder buffer: on rename

phys → arch. reg
for new instrs

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

free list

%x19
%x23
...
...

reorder buffer (ROB)

remove
here
on commit



add here
on rename



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

place newly started instruction at end of buffer
remember at least its destination register

reorder buffer: on rename

phys → arch. reg
for new instrs

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

free list

%x19
%x23
...
...

remove
here
on commit



reorder buffer (ROB)

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

add here
on rename



next renamed instruction goes in next slot, etc.

reorder buffer: on rename

phys \rightarrow arch. reg
for new instrs

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

free list

%x19
%x23
...
...

reorder buffer (ROB)

remove
here
on commit



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

add here
on rename



reorder buffer: on commit

phys → arch. reg
for new instrs

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

free list

%x19
%x13
...
...

remove
here
on commit



reorder buffer (ROB)

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

reorder buffer: on commit

phys → arch. reg
for new instrs

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

free list

%x19
%x13
...
...

remove
here →
on commit

reorder buffer (ROB)

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

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

reorder buffer: on commit

phys → arch. reg
for new instrs

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

phys → arch. reg
for committed

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

free list

%x19
%x13
...
...

remove
here →
on commit

reorder buffer (ROB)

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

commit stage tracks architectural to physical register map
for committed instructions

reorder buffer: on commit

phys → arch. reg
for new instrs

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

phys → arch. reg
for committed

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

free list

%x19
%x13
...
%x23

remove
here →
on commit

reorder buffer (ROB)

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

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

reorder buffer: on commit

phys → arch. reg
for new instrs

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

free list

%x19
%x13
...
%x23

phys → arch. reg remove here
for committed when committed

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

reorder buffer (ROB)

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

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

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

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

free list

%x19
%x13
...
...

phys → arch. reg
for committed

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

reorder buffer (ROB)

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

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

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

phys → arch. reg
for committed

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

reorder buffer (ROB)

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

free list

%x19
%x13
...
...

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

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

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

phys → arch. reg
for committed

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

reorder buffer (ROB)

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

free list

%x19
%x13
...
...

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

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

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

phys → arch. reg
for committed

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



free list

%x19
%x13
...
...

reorder buffer (ROB)

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



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

better? alternatives

- can take snapshots of register map on each branch

 - don't need to reconstruct the table

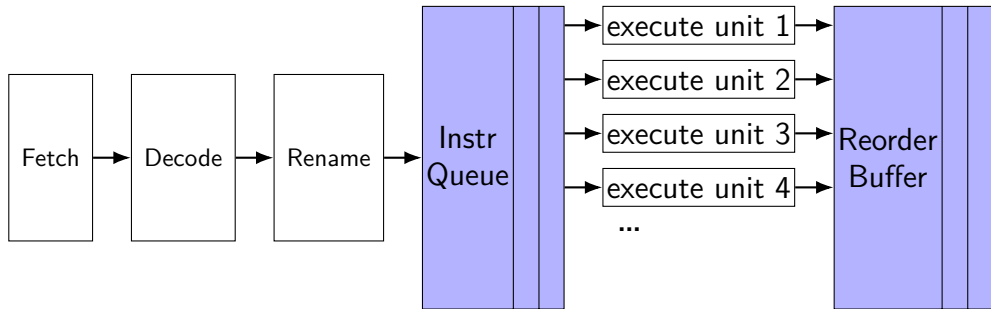
 - (but how to efficiently store them)

- can reconstruct register map before we commit the branch instruction

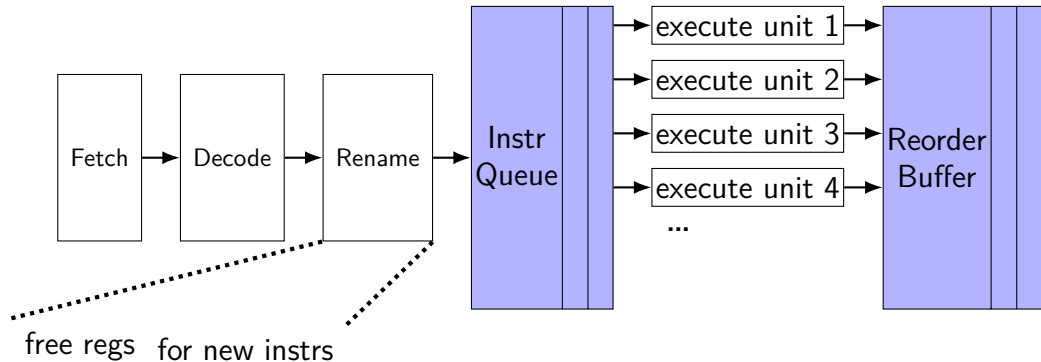
 - need to let reorder buffer be accessed even more?

- can track more/different information in reorder buffer

exceptions and OOO (one strategy)



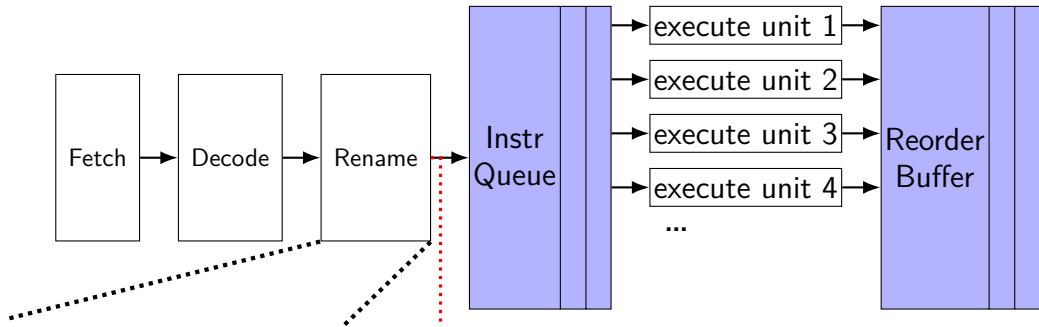
exceptions and OOO (one strategy)



X19
X23
...

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

exceptions and OOO (one strategy)



free regs for new instrs

X19
X23
...

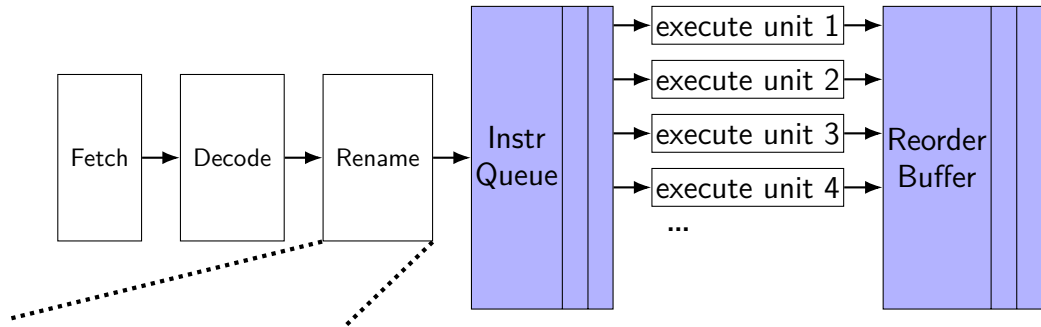
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

done instrs
committed in order

new instrs added

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

exceptions and OOO (one strategy)



free regs for new instrs for complete instrs

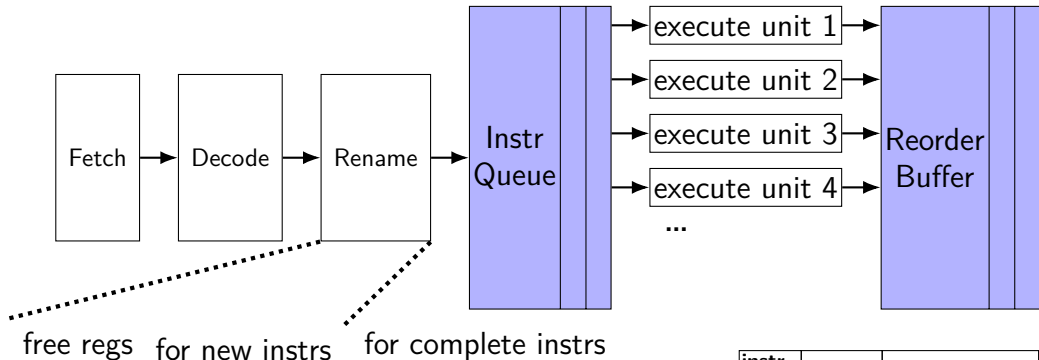
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



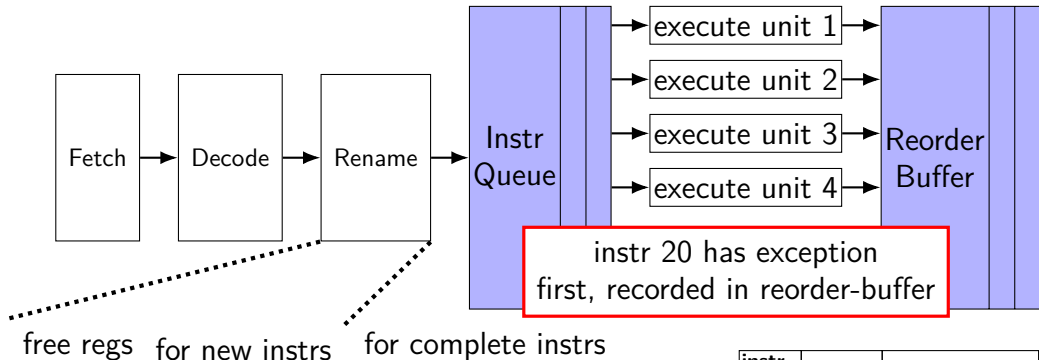
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



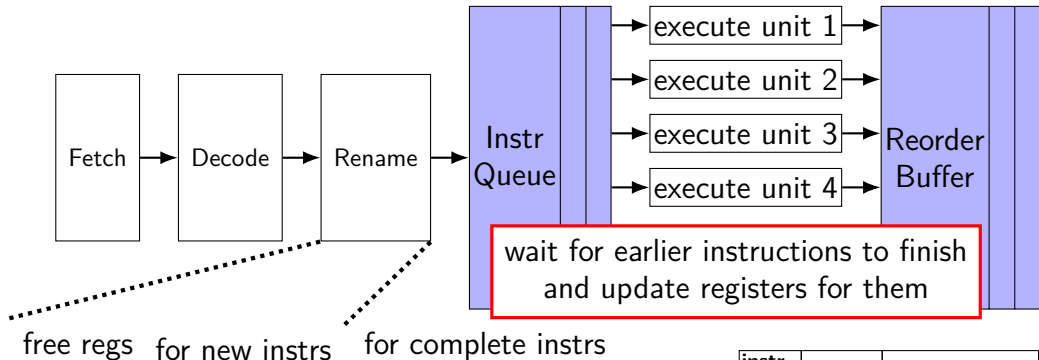
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



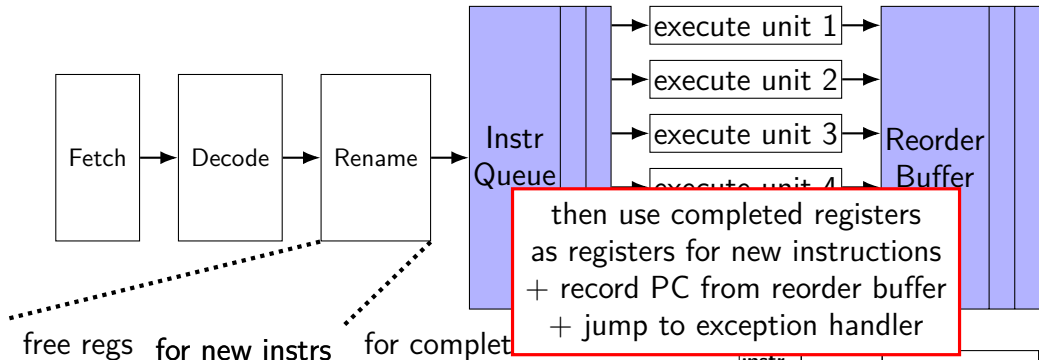
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



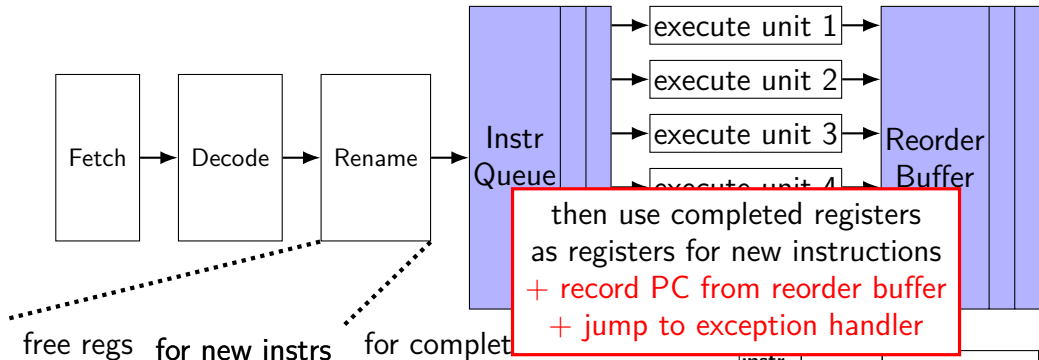
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



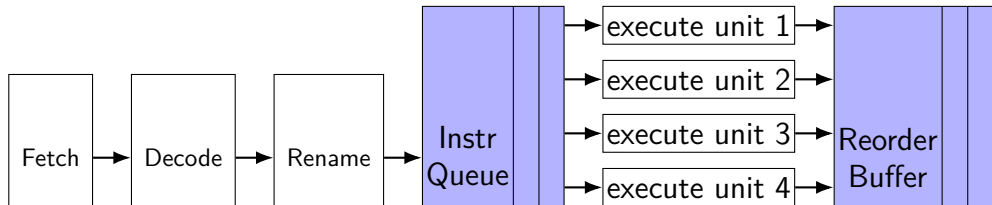
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



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

free regs for new instrs for complete instrs

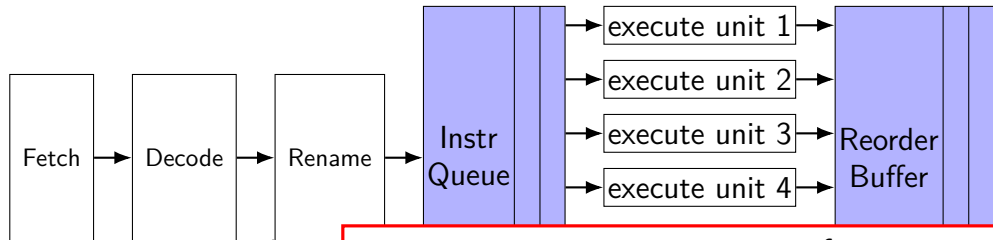
X19
X23
...

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

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

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

exceptions and OOO (one strategy)



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

free regs for new instrs for complete instrs

X19
X23
...

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

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

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

handling memory accesses?

one idea:

list of done + uncommitted loads+stores

execute load early + double-check on commit

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

loads check list of stores so you read back own values

actually finish store on commit

- maybe treat like branch misprediction if conflict?

the open-source BROOM pipeline

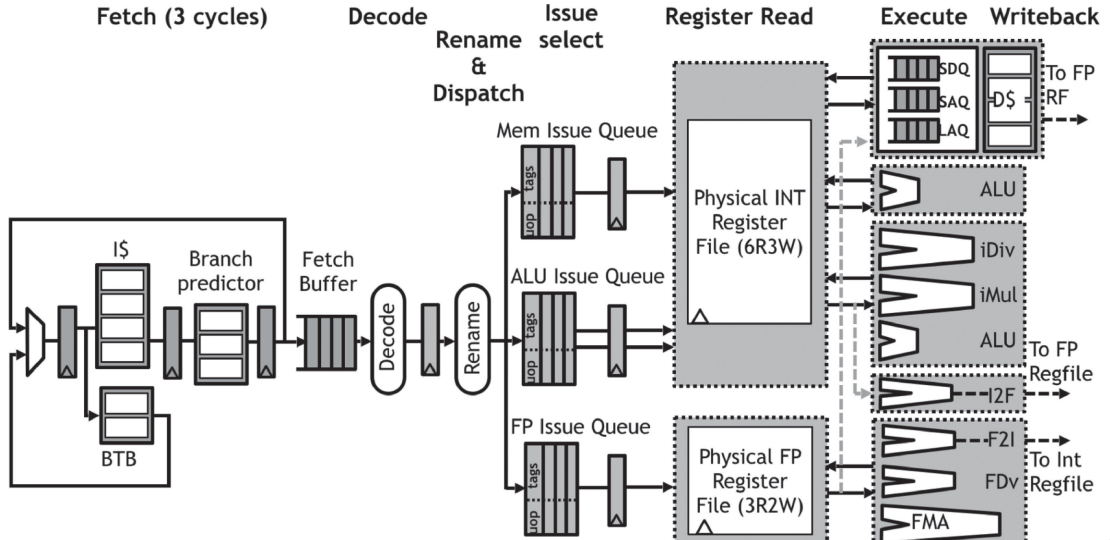
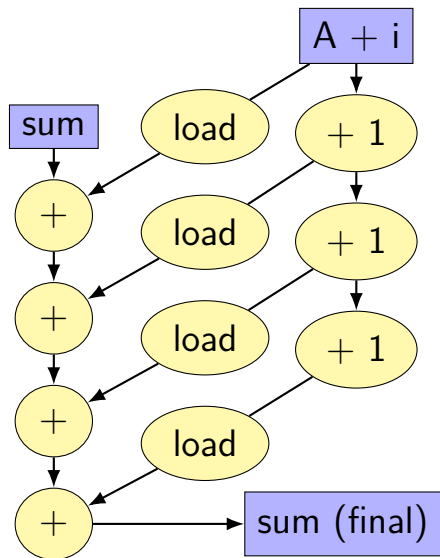


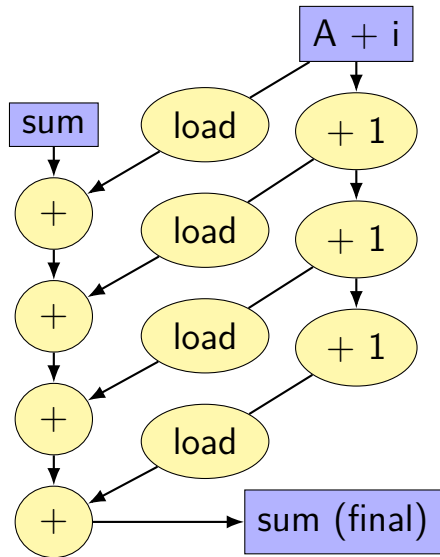
Figure from Celio et al., "BROOM: An Open Source Out-Of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS"

data flow model and limits



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

data flow model and limits

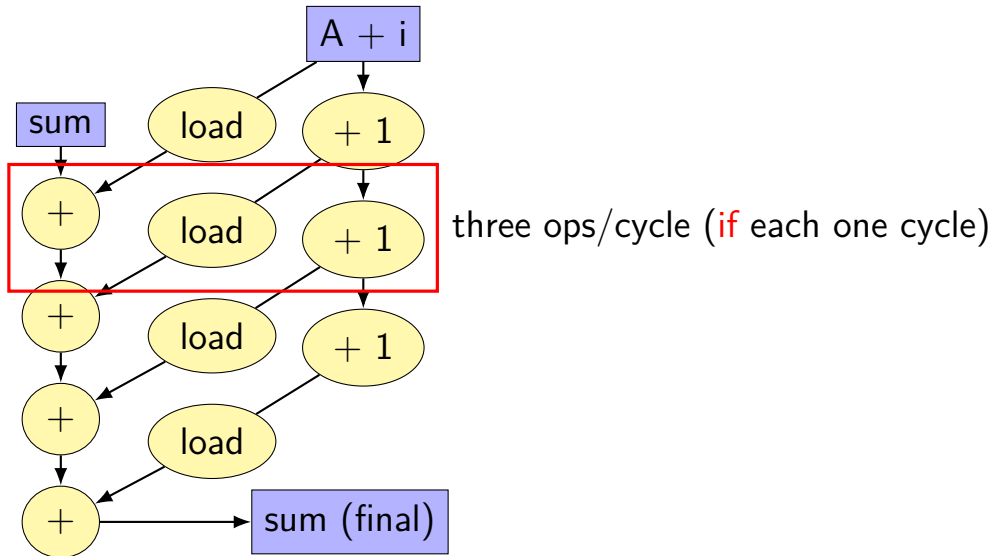


each yellow box = instruction

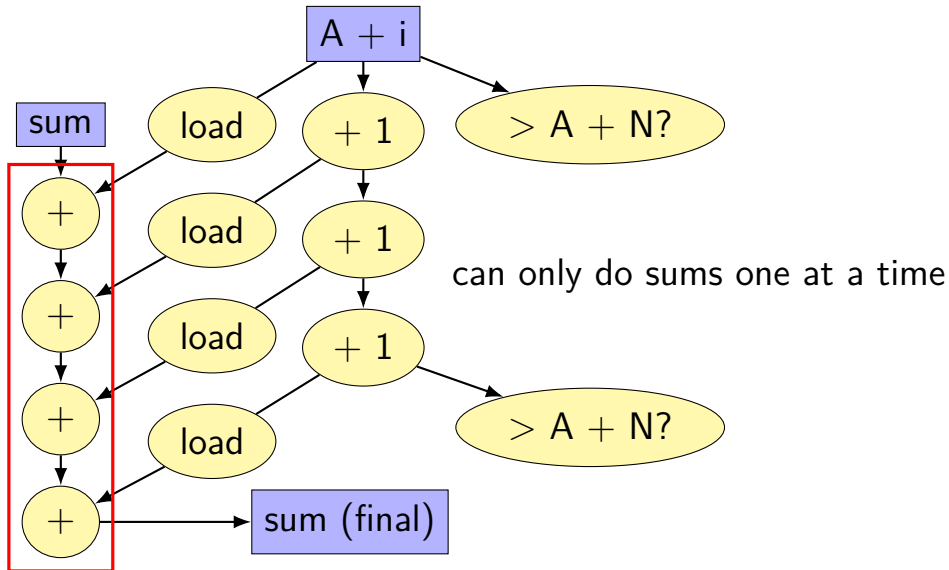
arrows = dependences

instructions only executed when dependencies

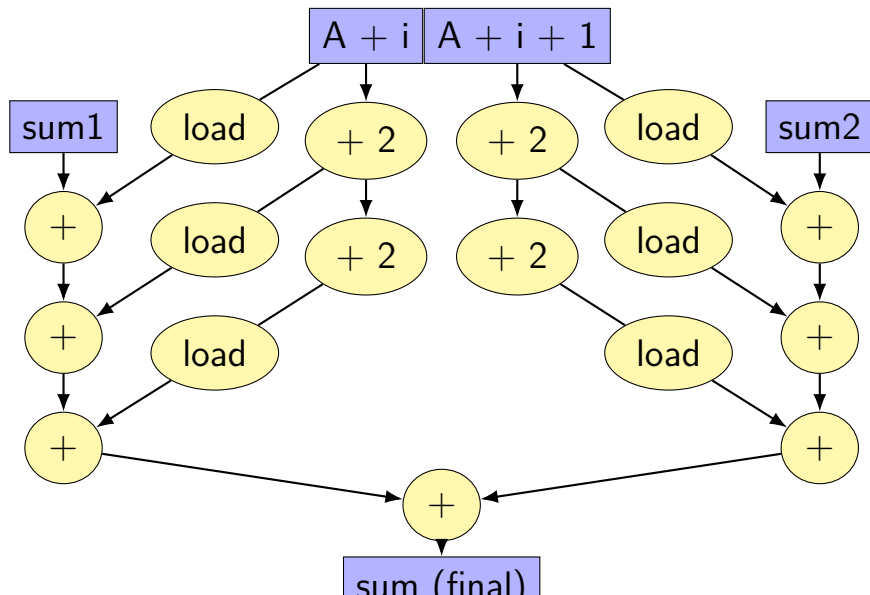
data flow model and limits



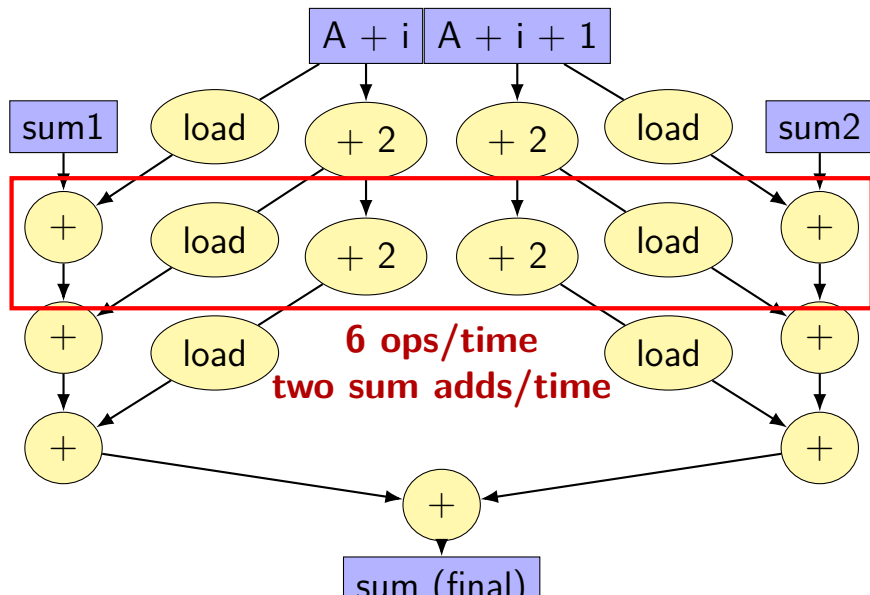
data flow model and limits



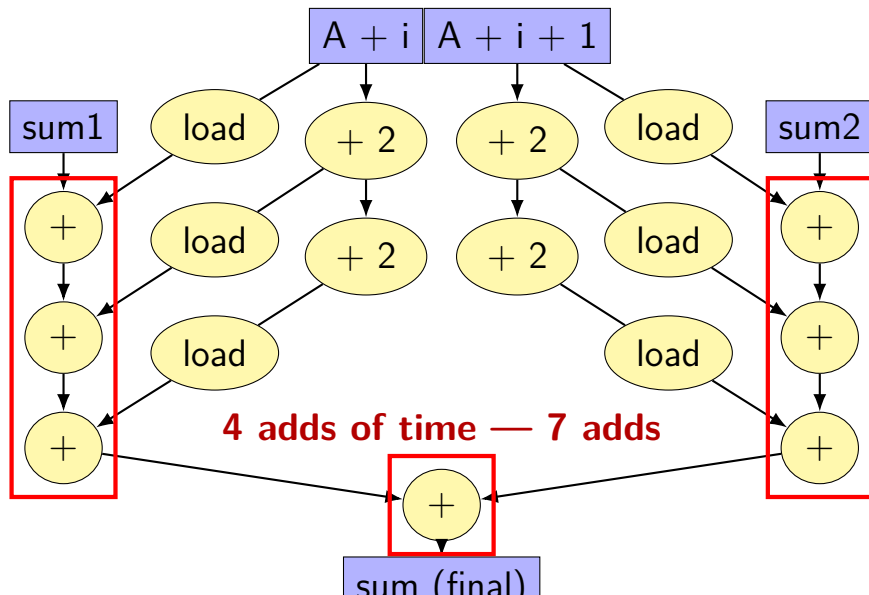
better data-flow



better data-flow



better data-flow



beyond 1-bit predictor

devote *more space* to storing history

main goal: rare exceptions don't immediately change prediction

example: branch taken 99% of the time

1-bit predictor: wrong about 2% of the time

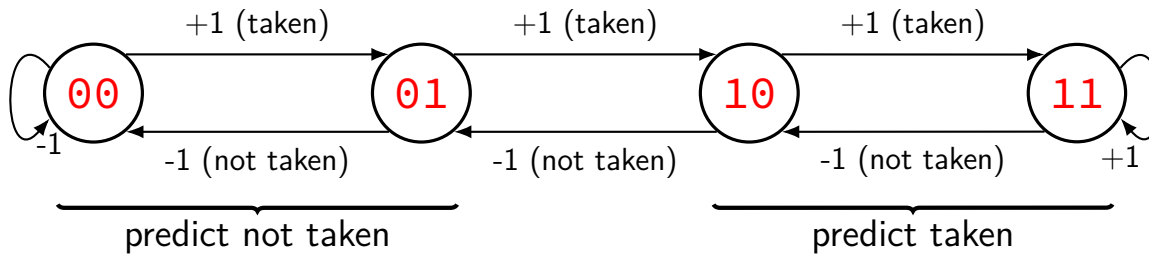
- 1% when branch not taken

- 1% of taken branches right after branch not taken

new predictor: wrong about 1% of the time

- 1% when branch not taken

2-bit saturating counter



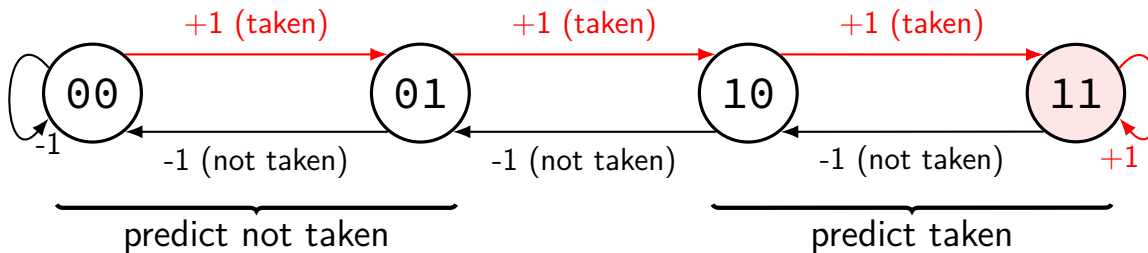
PC of branch

0x40042A

hash function

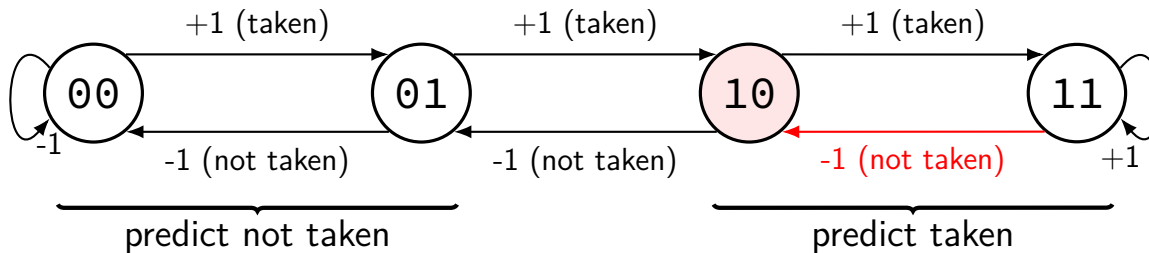
index	counter
0	11
1	01
2	11
...	...
14	10

2-bit saturating counter



branch always taken:
value increases to 'strongest' taken value

2-bit saturating counter



branch almost always taken, then not taken once:
still predicted as taken

example

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

iter.	table before	prediction	outcome	table after
1	01	not taken	taken	10
2	10	taken	taken	11
3	11	taken	taken	11
4	11	taken	not taken	10
1	10	taken	taken	11
2	11	taken	taken	11
3	11	taken	taken	11
4	11	taken	not taken	10
1	10	taken	taken	11
...

generalizing saturating counters

2-bit counter: ignore one exception to taken/not taken

3-bit counter: ignore more exceptions

000 \leftrightarrow 001 \leftrightarrow 010 \leftrightarrow 011 \leftrightarrow 100 \leftrightarrow 101 \leftrightarrow 110 \leftrightarrow 111

000-011: not taken

100-111: taken

exercise

use 2-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;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	01 (N)	T	
1	break	01 (N)	N	✓
1	mod 3	10 (T)	N	
2	break	00 (N)	N	✓
2	mod 3	01 (N)	N	✓
3	break	00 (N)	N	✓
3	mod 3	00 (N)	T	
4	break	00 (N)	N	✓
...
48	mod 3	00 (N)	T	
49	break	00 (N)	N	✓
49	mod 3	01 (N)	N	✓
50	break	00 (N)	T	
0	mod 3	00 (N)	T	
1	break	01 (N)	N	✓

mod 3: correct for i=1,2,4,5,7,8,...,49
(33/50)

mod 3: ends up always predicting not taken

break: correct for i=2,3,...,48

(49/50)

break: ends up always predicting not taken

overall: 82/100

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

branch patterns

```
i = 4;  
do {  
    ...  
    i -= 1;  
} while (i != 0);
```

typical pattern for jump to top of do-while above:

TTTN TTTN TTTN TTTN TTTN...(T = taken, N = not taken)

goal: take advantage of recent pattern to make predictions

just saw 'NTTTNT'? predict T next

'TNTTTN'? predict T; 'TTNTTT'? predict N next

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

4-iter loop: TTTN TTTN TTTN ...

idx *recent pattern*

0	NNNNNN
1	NNTNTT
2	TTTTNT
3	TTTTTT
...	...
14	NTNTTN
15	NNTTTT

actual outcome

from commit(?) stage

???

convert to
prediction
???

prediction
to fetch stage

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

4-iter loop: TTTN TTTN TTTN ...

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT

idx recent
pattern

0	NNNNNN
1	NNTNTT
2	TTTTNT
3	TTTTTT
...	...
14	NTNTTN
15	NNTTTT

actual outcome

from commit(?) stage

???

convert to
prediction

???

prediction
to fetch stage

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

4-iter loop: TTTN TTTN TTTN ...

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT

idx recent
pattern

0	NNNNNN
1	NNTNTT
2	TTTTNT
3	TTTTTT
...	...
14	NTNTTN
15	NNTTTT

actual outcome
from commit(?) stage

???

convert to
prediction
???

prediction
to fetch stage

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

idx recent
pattern

0	NNNNNN
1	NNTNTT
2	TTTNTT
3	TTTTTT
...	...
14	NTNTTN
15	NNTTTT

actual outcome

from commit(?) stage

???

convert to
prediction

???

prediction
to fetch stage

4-iter loop: TTTN TTTN TTTN ...

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT
2	TTTNTT	???	taken	TTNTTT

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

idx recent
pattern

0	NNNNNN
1	NNTNTT
2	TTTNTTT
3	TTTTTT
...	...
14	NTNTTN
15	NNTTTT

actual outcome

from commit(?) stage

???

convert to
prediction

???

prediction
to fetch stage

4-iter loop: TTTN TTTN TTTN ...

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT
2	TTTNTT	???	taken	TTNTTT
3	TTNTTT	???	taken	TNTTTT

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

0x40041B	movq \$4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

idx recent
pattern

0	NNNNNN
1	NNTNTT
2	TTTNTTT
3	TTTTTT
...	...
14	NTNTTN
15	NNTTTT

actual outcome

from commit(?) stage

???

convert to
prediction

???

prediction
to fetch stage

4-iter loop: TTTN TTTN TTTN ...

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT
2	TTTNTT	???	taken	TTNTTT
3	TTNTTT	???	taken	TNTTTT
4	TNTTTT	???	not taken	NTTTTN

recent pattern to prediction?

easy cases:

just saw TTTTTT: predict T

just saw NNNNNN: predict N

just saw TNTNTN: predict T

hard cases:

TTNTTTT

predict T? loop with many iterations
(NTTTTTTTNTTTTTTTNTTTTTT...)

predict T? if statement mostly taken
(TTTTNTTNTTTTTTTTTTTNTTTT...)

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTN
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern
NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter
00
00
...
10
...
11
...
01
01
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTNT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern

NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter

00
00
...
10
...
11
...
01
01 10
11

prediction
to fetch sta

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTNT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern
NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter
00
00
...
10
...
11
...
01
01 10
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTNTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern
NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter
00
00
...
10
...
11
...
01 10
01 10
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx recent
pattern

0	NNNN
1	TNTT
2	TTTNTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern

NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter

00
00
...
10
...
11
...
01 10
01 10
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTT TNTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern	counter
NNNN	00
NNNT	00
...	...
NTTT	10
...	...
TNTT	11
...	...
TTNT	01 10
TTTN	01 10
TTTT	11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTT NTTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern
NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter
00
00
...
10
...
11
...
01 10
01 10
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTNTTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern

NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter

00
00
...
10 11
...
11
...
01 10
01 10
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTT TNTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern
NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTT

counter
00
00
...
10 11
...
11
...
01 10
01 10
11

prediction
to fetch sta

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTT	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

history of history

PC of branch

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTT TNTT
3	TTTT
...	...
14	NTTN
15	TTTT

actual outcome
from commit(?) stage

pattern
NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter
00
00
...
10 11
...
11
...
01 10
01 10 11
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

prediction
to fetch sta

history of history

PC of branch

actual outcome
from commit(?) stage

0x40042A

hash

idx	recent pattern
0	NNNN
1	TNTT
2	TTTT NTTT
3	TTTT
...	...
14	NTTN
15	TTTT

pattern

NNNN
NNNT
...
NTTT
...
TNTT
...
TTNT
TTTN
TTTT

counter

00
00
...
10 11
...
11
...
01 10
01 10 11
11

iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

prediction
to fetch sta

local patterns and collisions (1)

```
i = 10000;  
do {  
    p = malloc(...);  
    if (p == NULL) goto error;  // BRANCH 1  
    ...  
} while (i-- != 0); // BRANCH 2
```

what if branch 1 and branch 2 hash to same table entry?

local patterns and collisions (1)

```
i = 10000;  
do {  
    p = malloc(...);  
    if (p == NULL) goto error; // BRANCH 1  
    ...  
} while (i-- != 0); // BRANCH 2
```

what if branch 1 and branch 2 hash to same table entry?

pattern: TNTNTNTNTNTNTNTNT...

actually no problem to predict!

local patterns and collisions (2)

```
i = 10000;  
do {  
    if (i % 2 == 0) goto skip; // BRANCH 1  
    ...  
    p = malloc(...);  
    if (p == NULL) goto error; // BRANCH 2  
skip: ...  
} while (i-- != 0); // BRANCH 3
```

what if branch 1 and branch 2 and branch 3 hash to same table entry?

local patterns and collisions (2)

```
i = 10000;  
do {  
    if (i % 2 == 0) goto skip; // BRANCH 1  
    ...  
    p = malloc(...);  
    if (p == NULL) goto error; // BRANCH 2  
skip: ...  
} while (i-- != 0); // BRANCH 3
```

what if branch 1 and branch 2 and branch 3 hash to same table entry?

pattern: TTNNTTNNTTNNTTNNTT

also no problem to predict!

local patterns and collisions (3)

```
i = 10000;  
do {  
    if (A) goto one  // BRANCH 1  
    ...  
one:  
    if (B) goto two  // BRANCH 2  
    ...  
two:  
    if (A or B) goto three // BRANCH 3  
    ...  
    if (A and B) goto three // BRANCH 4  
    ...  
three:  
    ... // changes A, B  
} while (i-- != 0);
```

what if branch 1-4 hash to same table entry?

better for prediction of branch 3 and 4

global history predictor: idea

one predictor idea: ignore the PC

just record taken/not-taken pattern for all branches

lookup in big table like for local patterns

global history predictor (1)

branch history register

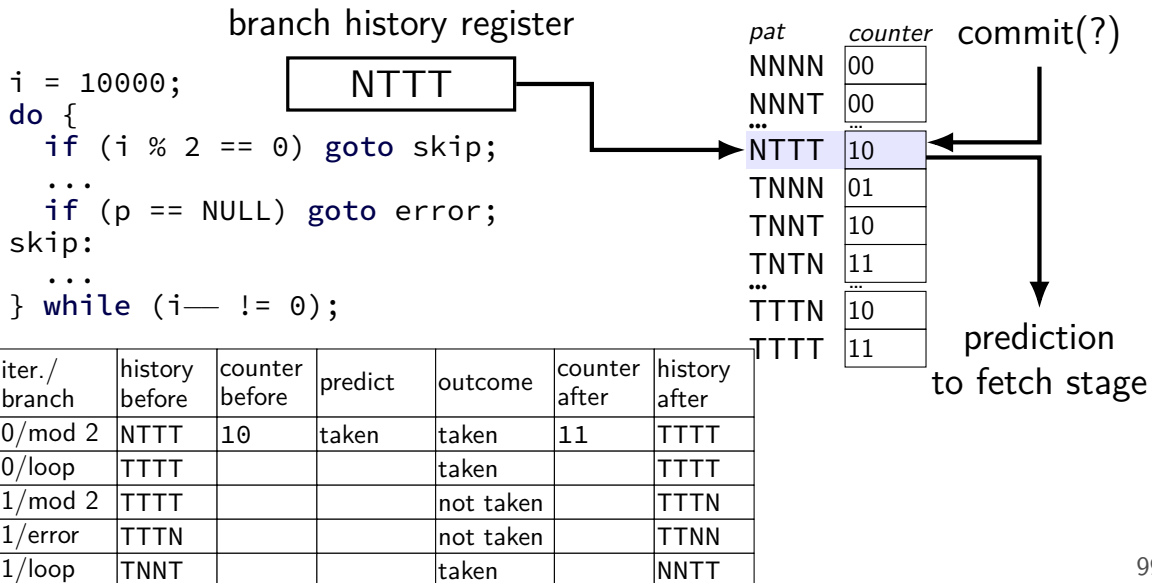
NTTT

<i>pat</i>	<i>counter</i>
NNNN	00
NNNT	00
...	...
NTTT	10
TNNN	01
TNNT	10
TNTN	11
...	...
TTTN	10
TTTT	11

outcome
from
commit(?)

prediction
to fetch stage

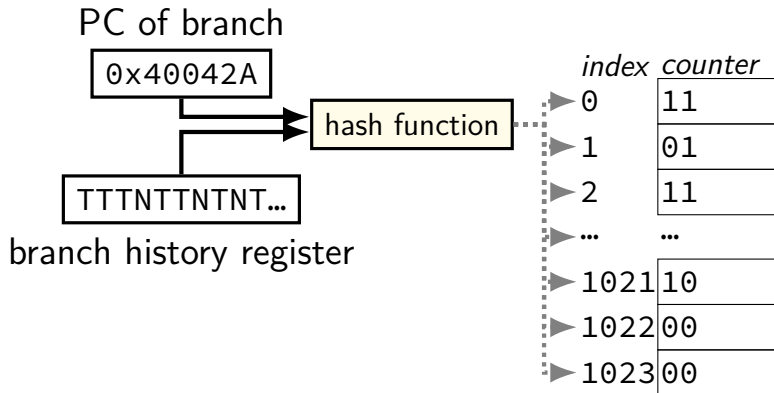
global history predictor (1)



correlating predictor

global history *and* local info good together

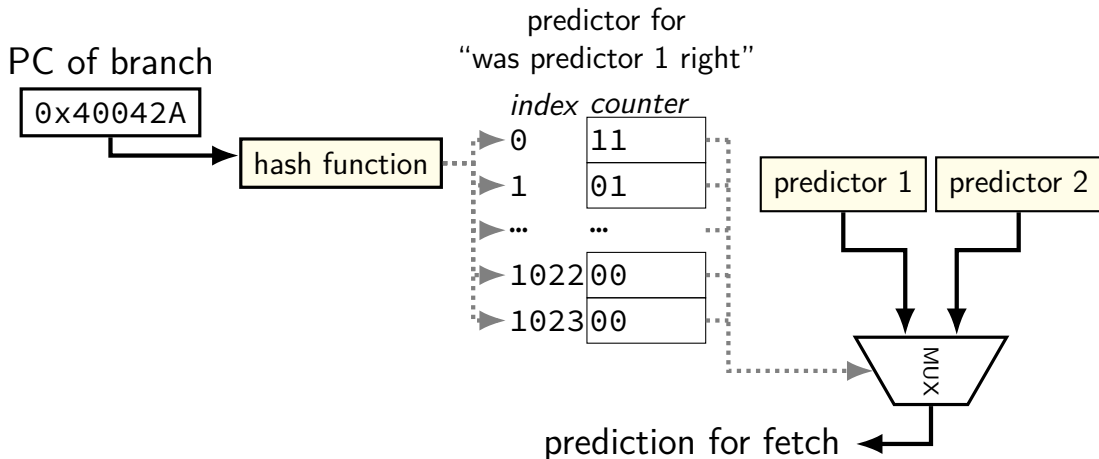
one idea: **combine history register + PC** (“gshare”)



mixing predictors

different predictors good at different times

one idea: have two predictors, + predictor to predict which is right



loop count predictors (1)

```
for (int i = 0; i < 64; ++i)  
    ...
```

can we predict this perfectly with predictors we've seen

yes — local or global history with 64 entries

but this is very important — more efficient way?

loop count predictors (2)

loop count predictor idea: look for NNNNNNT+repeat (or TTTTTTN+repeat)

track for each possible loop branch:

- how many repeated Ns (or Ts) so far

- how many repeated Ns (or Ts) last time before one T (or N)

- something to indicate this pattern is useful?

known to be used on Intel

benchmark results

from 1993 paper

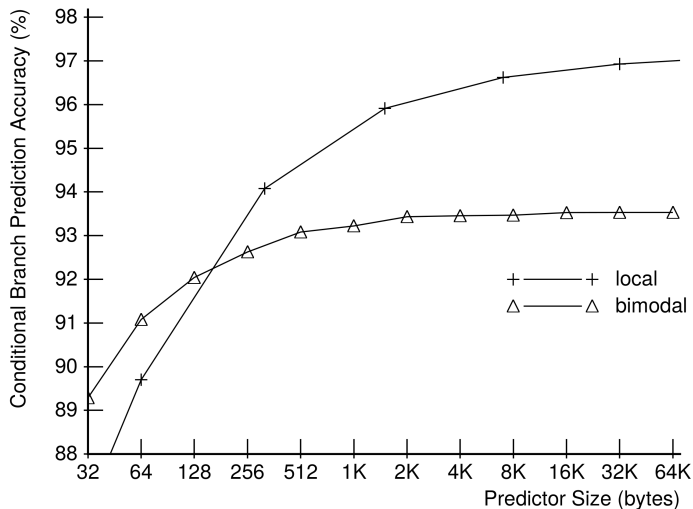
(not representative of modern workloads?)

rate for conditional branches on benchmark

variable table sizes

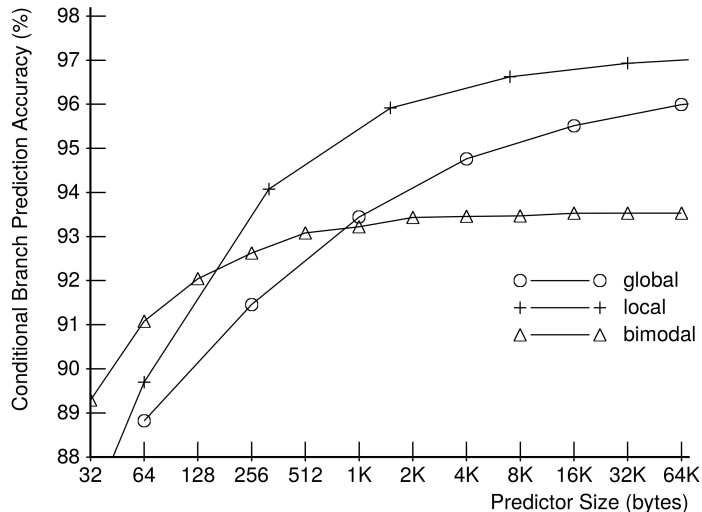
2-bit ctr + local history

from McFarling, “Combining Branch Predictors” (1993)



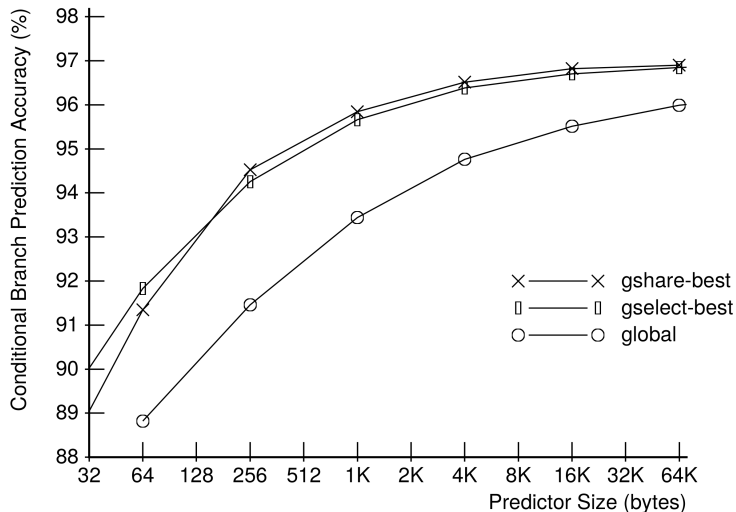
2-bit (bimodal) + local + global hist

from McFarling, "Combining Branch Predictors" (1993)



global + hash(global+PC) (gshare/gselect)

from McFarling, "Combining Branch Predictors" (1993)



real BP?

details of modern CPU's branch predictors often not public

but...

Google Project Zero blog post with reverse engineered details

<https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>

for RE'd BTB size:

<https://xania.org/201602/haswell-and-ivy-btb>

reverse engineering Haswell BPs

branch target buffer

- 4-way, 4096 entries

- ignores bottom 4 bits of PC?

- hashes PC to index by shifting + XOR

- seems to store 32 bit offset from PC (not all 48+ bits of virtual addr)

indirect branch predictor

- like the global history + PC predictor we showed, but...

- uses history of recent branch addresses instead of taken/not taken

- keeps some info about last 29 branches

what about conditional branches??? loops???

- couldn't find a reasonable source