exercise: different pipeline

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

result only available near end of second execute stage

where does forwarding, stalls occur?

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

last time

diminishing returns for pipelines

hazards

- pipeline does not work because value not ready

stalling to resolve hazards

- insert no-operations to wait

forwarding

- take value from elsewhere in pipeline
- replace value just read
- make decision using MUX

jXX: stalling?

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

	cycle # 0	1	2	3	4	5	6	7	8
`cmpq %r8, %r9`	F	D	E	M	W				
`jne LABEL`		F	D	E	M	W			
(do nothing)			F	D	E	M	W		
(do nothing)				F	D	E	M	W	
`xorq %r10, %r11`					F	D	E	M	W
`movq %r11, 0(%r12)`						F	D	E	M
...									

jXX: stalling?

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

cmpq %r8, %r9

jne LABEL

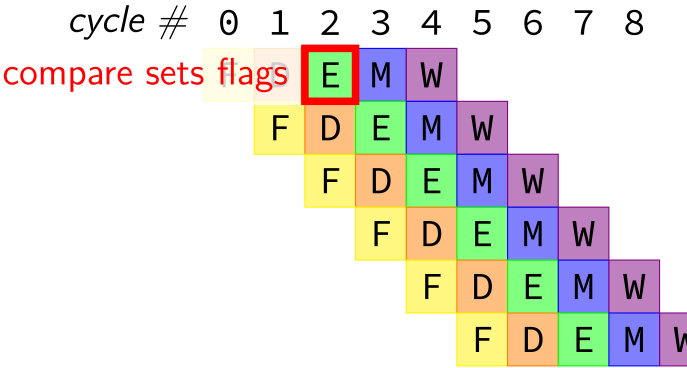
(do nothing)

(do nothing)

xorq %r10, %r11

movq %r11, 0(%r12)

...



jXX: stalling?

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

	cycle #	0	1	2	3	4	5	6	7	8
cmpq %r8, %r9		F	D	E	M	W				
jne LABEL	compute if jump goes to LABEL		F	D	$\boxed{E}$	M	W			
(do nothing)				F	D	E	M	W		
(do nothing)					F	D	E	M	W	
xorq %r10, %r11						F	D	E	M	W
movq %r11, 0(%r12)							F	D	E	M
...										

jXX: stalling?

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

	cycle #	0	1	2	3	4	5	6	7	8
cmpq %r8, %r9		F	D	E	M	W				
jne LABEL			F	D	E	M	W			
(do nothing)				F	D	E	M	W		
(do nothing)					F	D	E	M	W	
xorq %r10, %r11	use computed result					$\boxed{F}$	D	E	M	W
movq %r11, 0(%r12)							F	D	E	M
...										

making guesses

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

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

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

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

jXX: speculating right (1)

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

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

	cycle # 0	1	2	3	4	5	6	7	8
`cmpq %r8, %r9`	F	D	E	M	W				
`jne LABEL`		F	D	E	M	W			
`xorq %r10, %r11`			F	D	E	M	W		
`movq %r11, 0(%r12)`				F	D	E	M	W	
…									

jXX: speculating wrong

	cycle # 0	1	2	3	4	5	6	7	8	
`cmpq %r8, %r9`	F	D	E	M	W					
`jne LABEL`		F	D	E	M	W				
`xorq %r10, %r11`			F	D						
`(inserted nop)`					E	M	W			
`movq %r11, 0(%r12)`				F						
`(inserted nop)`					D	E	M	W		
`LABEL: addq %r8, %r9`					F	D	E	M	W	
`imul %r13, %r14`						F	D	E	M	W
…										

jXX: speculating wrong

cycle #	0	1	2	3	4	5	6	7	8	
`cmpq %r8, %r9`	F	D	E	M	W					
`jne LABEL`		F	D	E	M	W				
`xorq %r10, %r11`			F	D	instruction “squashed”					
(inserted nop)					E	M	W			
`movq %r11, 0(%r12)`				F	instruction “squashed”					
(inserted nop)					D	E	M	W		
`LABEL: addq %r8, %r9`					F	D	E	M	W	
`imul %r13, %r14`						F	D	E	M	W
…										

“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

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

static branch prediction

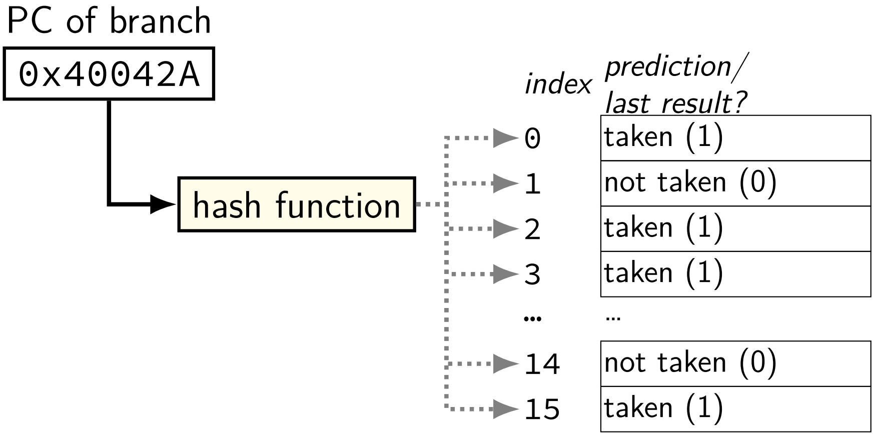
forward (target $>$ PC) not taken; backward taken

intuition: loops:

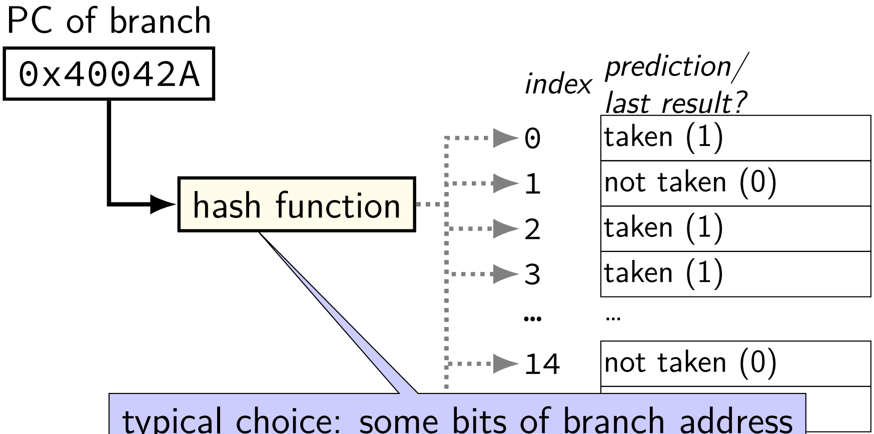
```
LOOP: ...
      ...
      je LOOP

LOOP: ...
      jne SKIP_LOOP
      ...
      jmp LOOP
SKIP_LOOP:
```

predict: repeat last

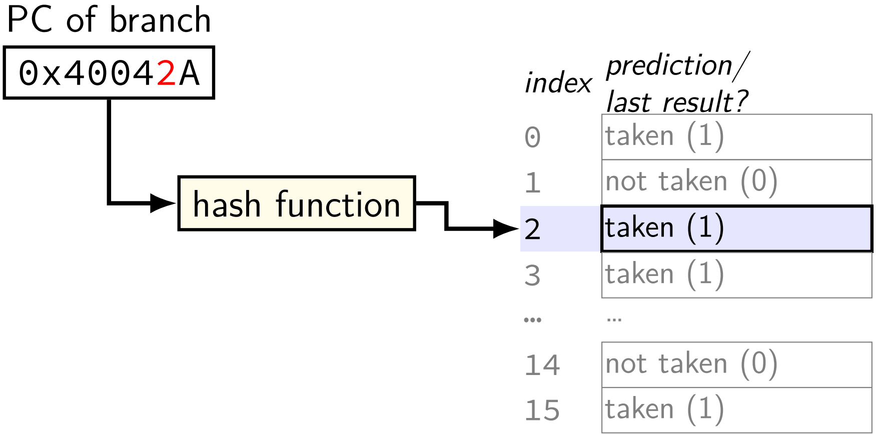


predict: repeat last

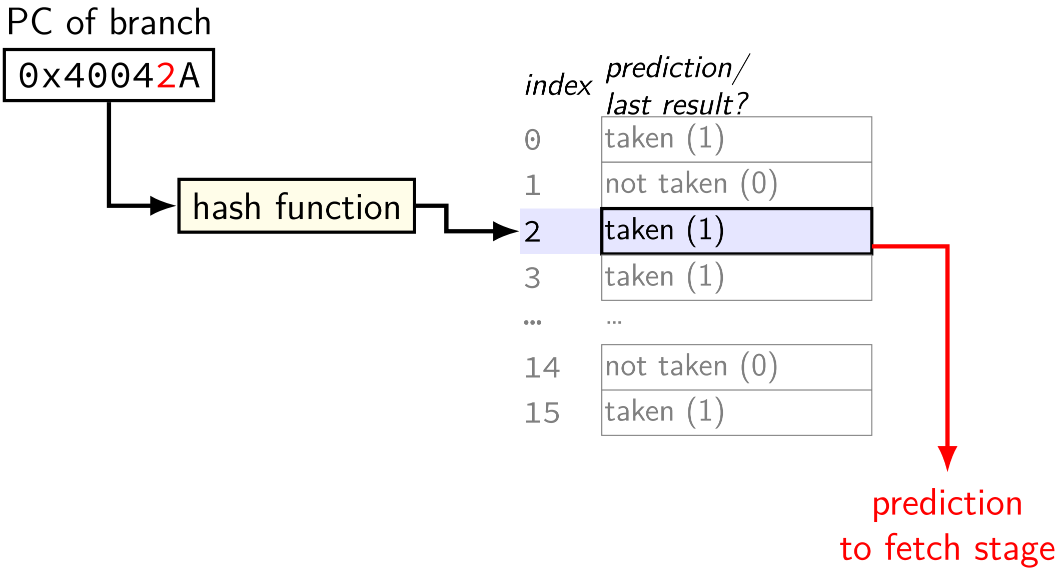


typical choice: some bits of branch address
for our example: will use bits 4-7

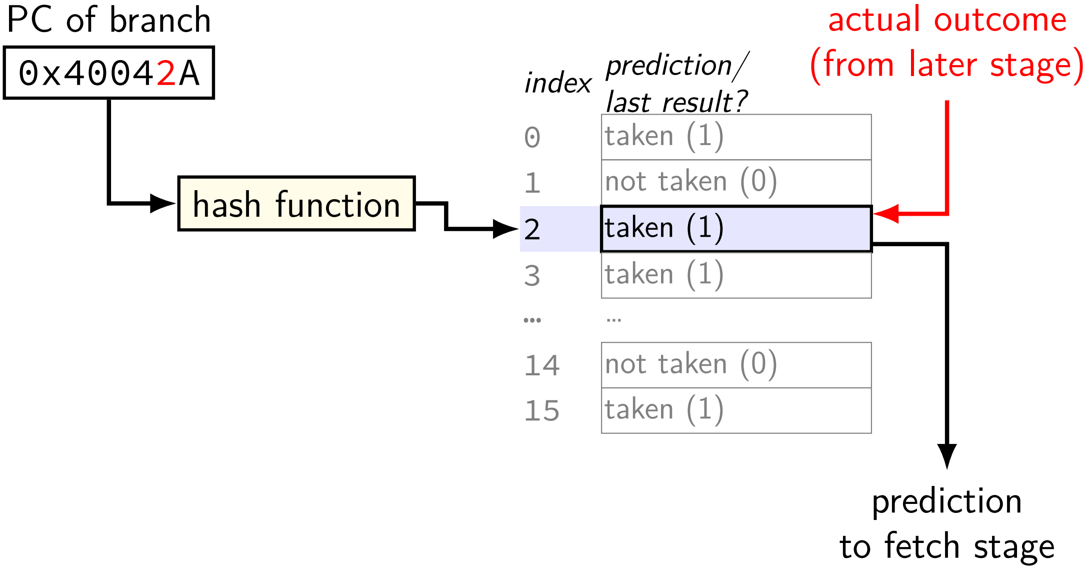
predict: repeat last



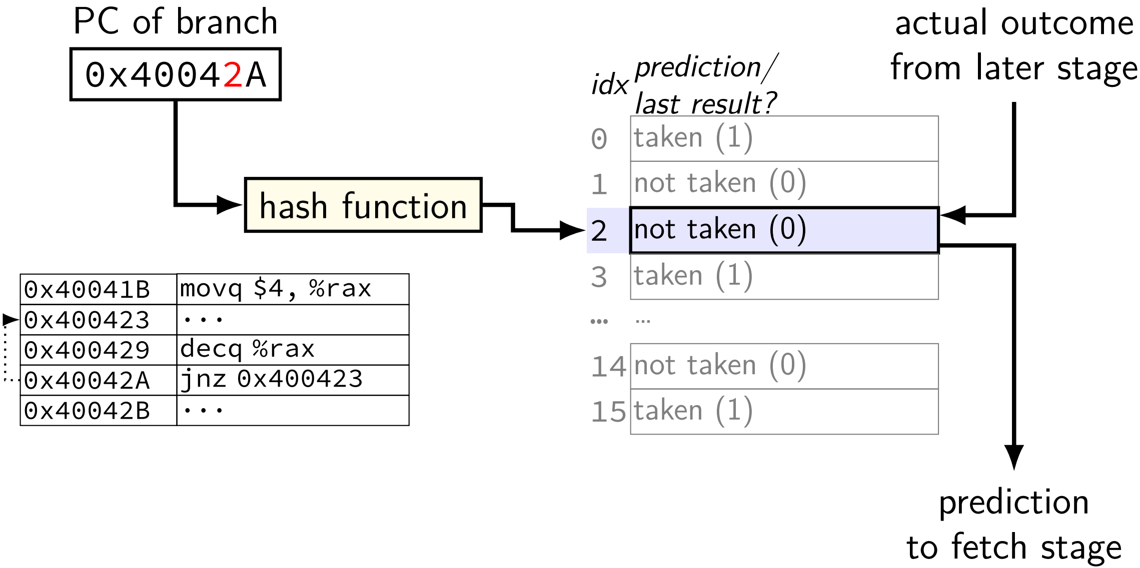
predict: repeat last



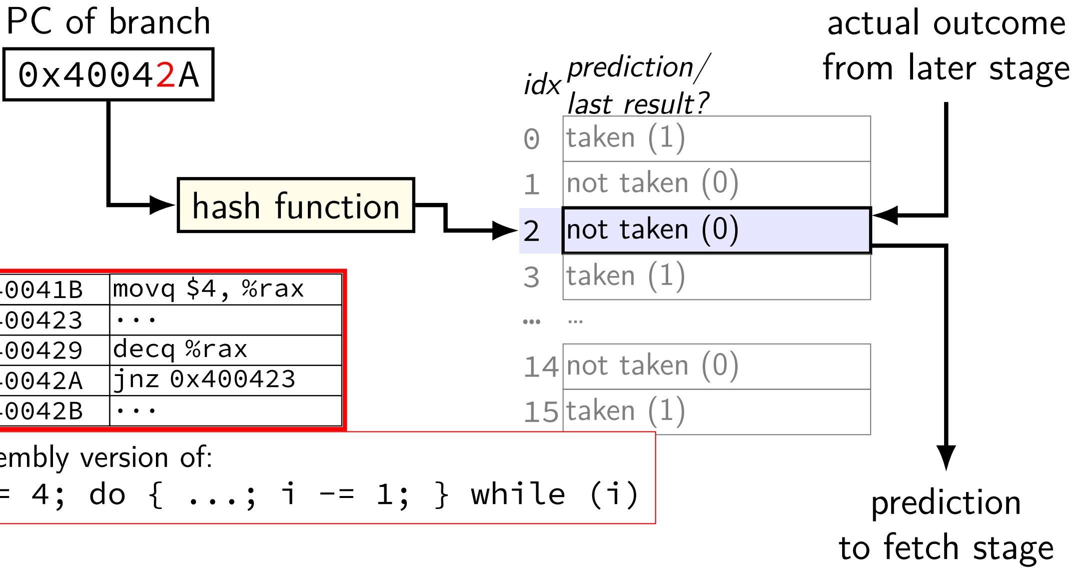
predict: repeat last



example


PC of branch
0x40042A
hash function
idx
prediction/ last result?
0 taken (1)
1 not taken (0)
2 not taken (0)
3 taken (1)
...
14 not taken (0)
15 taken (1)
actual outcome from later stage
prediction to fetch stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jnz 0x400423
0x40042B ...

example

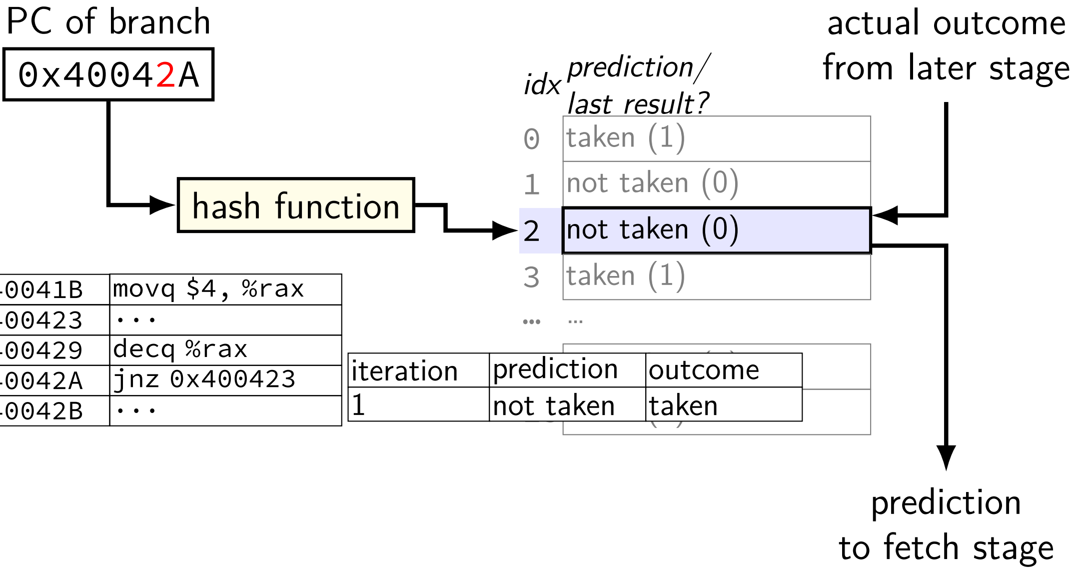


0x40041B	movq $4, %rax
0x400423	···
0x400429	decq %rax
0x40042A	jnz 0x400423
0x40042B	···

assembly version of:

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

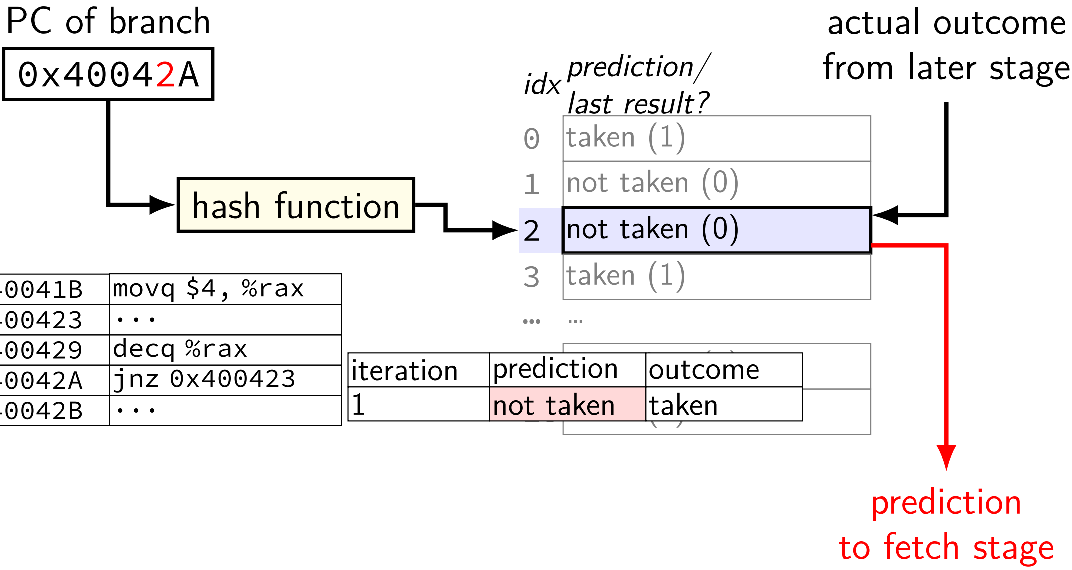
example



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

iteration	prediction	outcome
1	not taken	taken

example

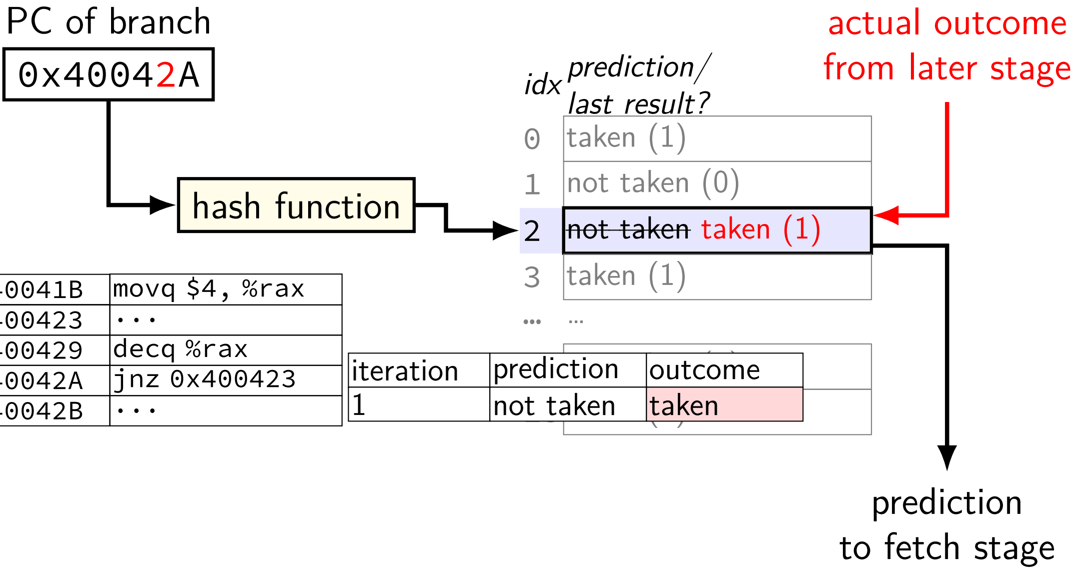


idx	*prediction/ last result?*
0	taken (1)
1	not taken (0)
2	not taken (0)
3	taken (1)
…	…

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

iteration	prediction	outcome
1	not taken	taken

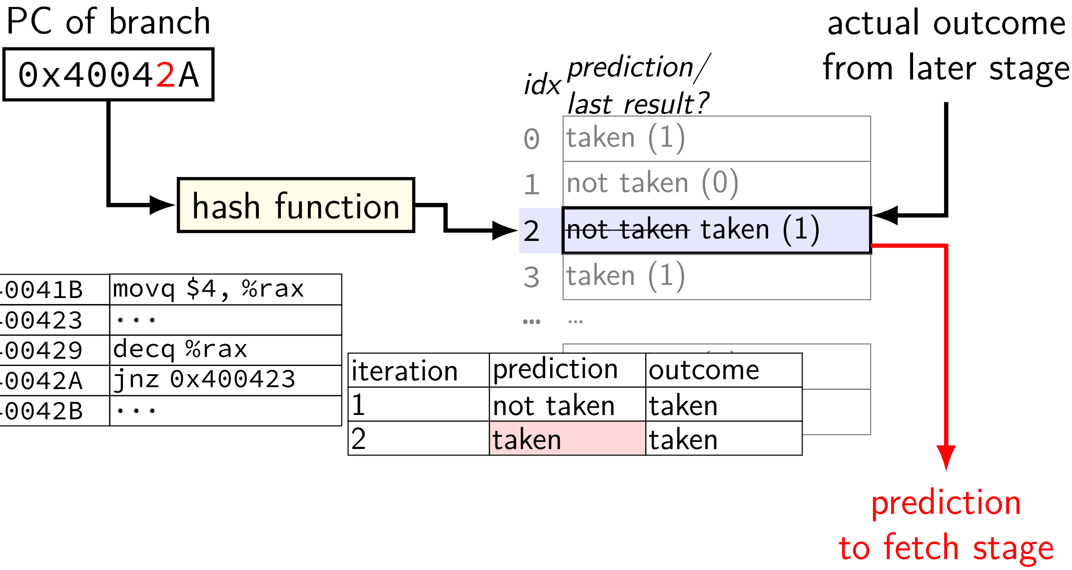
example



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

iteration	prediction	outcome
1	not taken	taken

example



0x40041B	movq $4, %rax
0x400423	···
0x400429	decq %rax
0x40042A	jnz 0x400423
0x40042B	···

iteration	prediction	outcome
1	not taken	taken
2	taken	taken

example

PC of branch

`0x40042A`

hash function

actual outcome from later stage

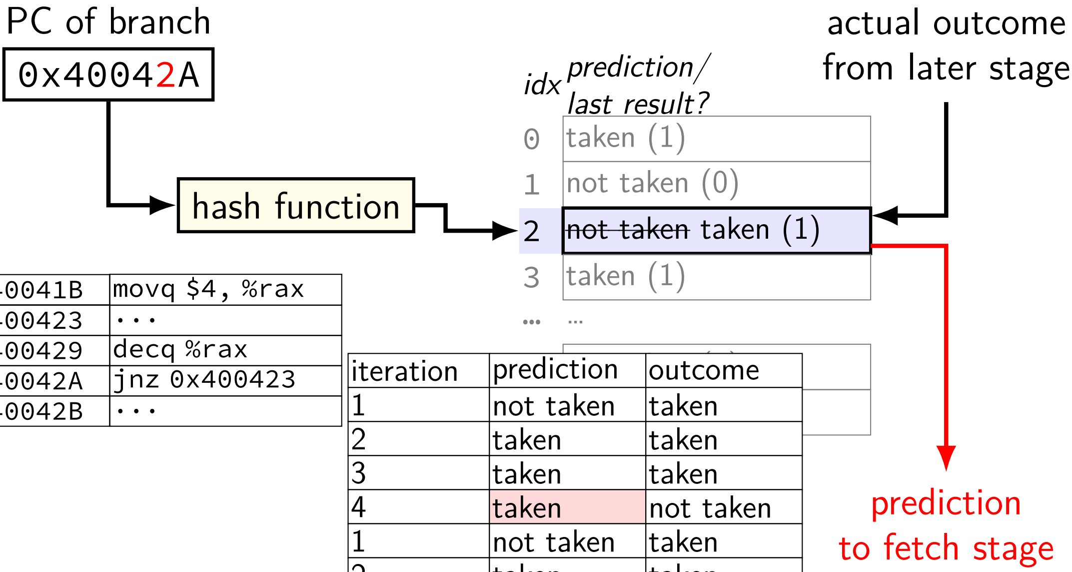
idx	*prediction/ last result?*
0	taken (1)
1	not taken (0)
2	~~not taken~~ taken (1)
3	taken (1)
...	...

prediction to fetch stage

0x40041B	`movq $4, %rax`
0x400423	...
0x400429	`decq %rax`
0x40042A	`jnz 0x400423`
0x40042B	...

iteration	prediction	outcome
1	not taken	taken
2	taken	taken

example



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

iteration	prediction	outcome
1	not taken	taken
2	taken	taken
3	taken	taken
4	taken	not taken
1	not taken	taken
2	taken	taken
...	...	...

example

PC of branch

`0x40042A`

hash function

idx	prediction/ last result?
0	taken (1)
1	not taken (0)
2	~~not taken~~ taken (1)
3	taken (1)
…	…

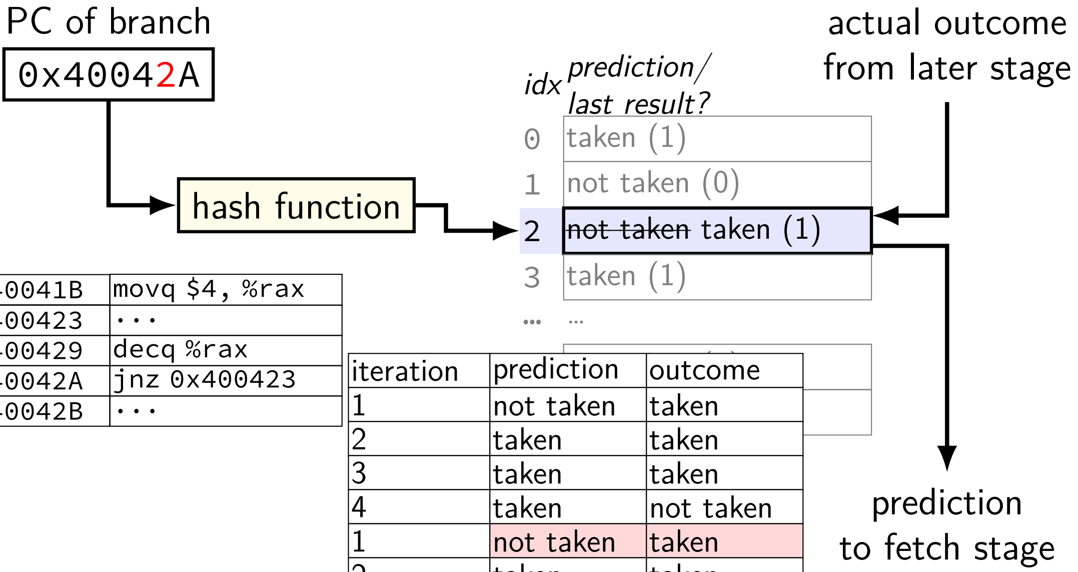
actual outcome from later stage

prediction to fetch stage

0x40041B	movq $4, %rax
0x400423	···
0x400429	decq %rax
0x40042A	jnz 0x400423
0x40042B	···

iteration	prediction	outcome
1	not taken	taken
2	taken	taken
3	taken	taken
4	taken	not taken
1	not taken	taken
2	taken	taken
…	…	…

example



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

iteration	prediction	outcome
1	not taken	taken
2	taken	taken
3	taken	taken
4	taken	not taken
1	not taken	taken
2	taken	taken
...	...	...

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 (pt 1)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate for `i % 3 == 0`?

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

exercise (pt 1)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate for `i % 3 == 0`?

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

i =	branch	pred	outcome	correct?
0	mod 3	???	T	???
1	mod 3	T	F	no
2	mod 3	F	F	yes
3	mod 3	F	T	no
...	...	...	...	...

exercise (pt 1)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate for `i % 3 == 0`?

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

i =	branch	pred	outcome	correct?
0	mod 3	???	T	???
1	mod 3	T	F	no
2	mod 3	F	F	yes
3	mod 3	F	T	no
…	…	…	…	…

exercise (pt 1)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate for `i % 3 == 0`?

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

i =	branch	pred	outcome	correct?
0	mod 3	???	T	???
1	mod 3	T	F	no
2	mod 3	F	F	yes
3	mod 3	F	T	no
...	...	...	...	...

exercise (pt 2)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate for `i == 50`?

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

exercise (full)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate?

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

exercise (full)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate?

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

i =	branch	pred	outcome	correct?
0	mod 3	???	T	???
1	== 50	???	F	???
1	mod 3	T	F	—
2	== 50	F	F	✓
…	…	…	…	…

exercise (full)

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate?

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

i =	branch	pred	outcome	correct?
0	mod 3	???	T	???
1	== 50	???	F	???
1	mod 3	T	F	—
2	== 50	F	F	✓
…	…	…	…	…

branch target buffer

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

- will happen in more complex pipelines

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

BTB: cache for branch targets

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

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

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

BTB: cache for branch targets

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

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

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

BTB: cache for branch targets

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

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

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

indirect branch prediction

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

BTB can provide a prediction

but can do better with more context

example—predict based on other recent computed jumps

- good for polymophic method calls

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

beyond pipelining: multiple issue

start more than one instruction/cycle

multiple parallel pipelines; many-input/output register file

hazard handling much more complex

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

beyond pipelining: out-of-order

find later instructions to do instead of stalling

lists of available instructions in pipeline registers

- take any instruction with available values

provide illusion that work is still done in order

- much more complicated hazard handling logic

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

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

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

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

register version tracking

goal: track different versions of registers

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

only forward the correct version

strategy for doing this: preprocess instructions represent version info

makes forwarding, etc. lookup easier

rewriting hazard examples (1)

`addq %r10, %r8`	`addq %r10,` $\%r8_{v1} \rightarrow \%r8_{v2}$
`addq %r11, %r8`	`addq %r11,` $\%r8_{v2} \rightarrow \%r8_{v3}$
`addq %r12, %r8`	`addq %r12,` $\%r8_{v3} \rightarrow \%r8_{v4}$

read different version than the one written

represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

for now: version numbers

later: something simpler to implement

write-after-write example

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

write-after-write example

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

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

write-after-write example

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

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

write-after-write example

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

keeping multiple versions

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

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

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

solution: have lots of extra registers

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

called register renaming

register renaming

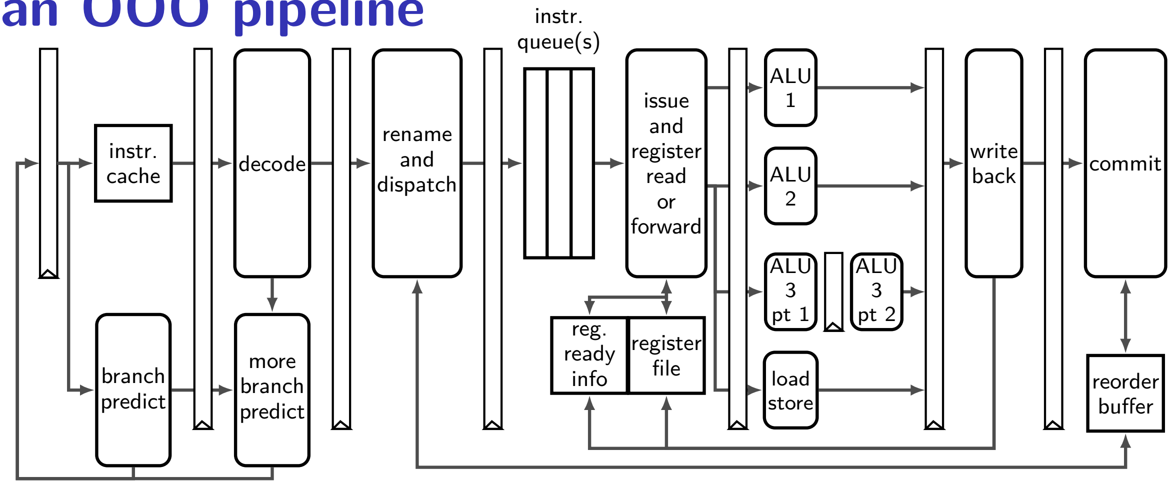
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

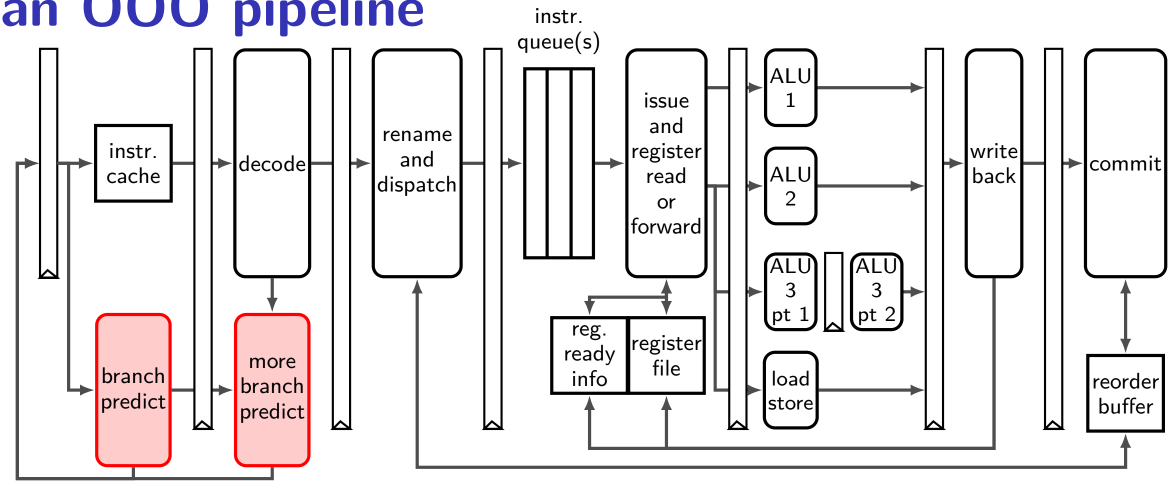
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

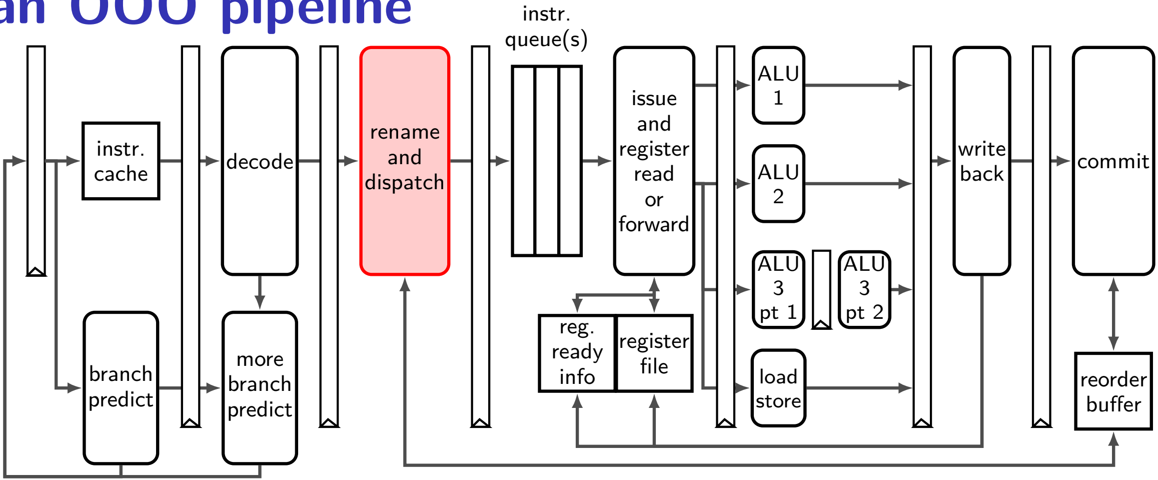


an OOO pipeline



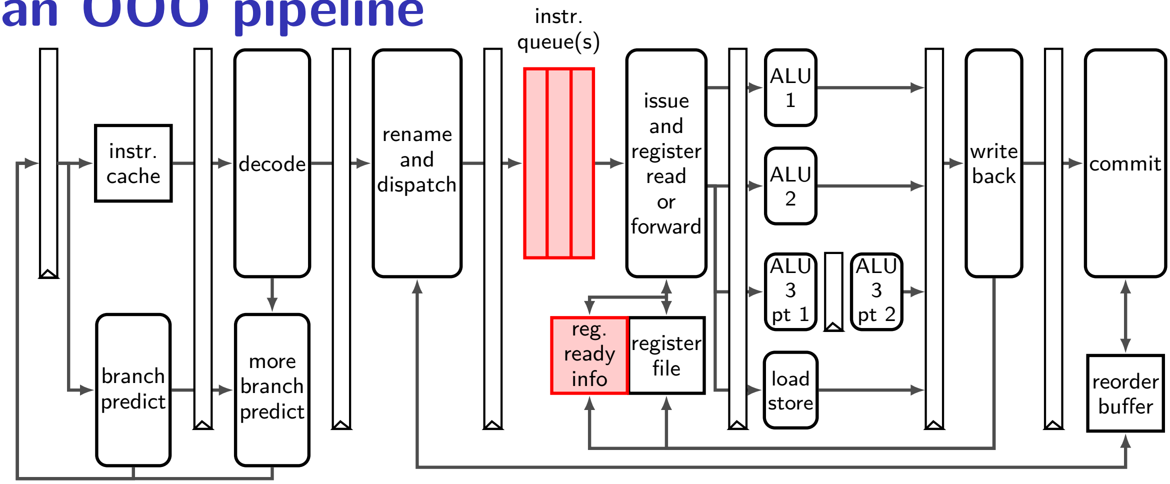
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



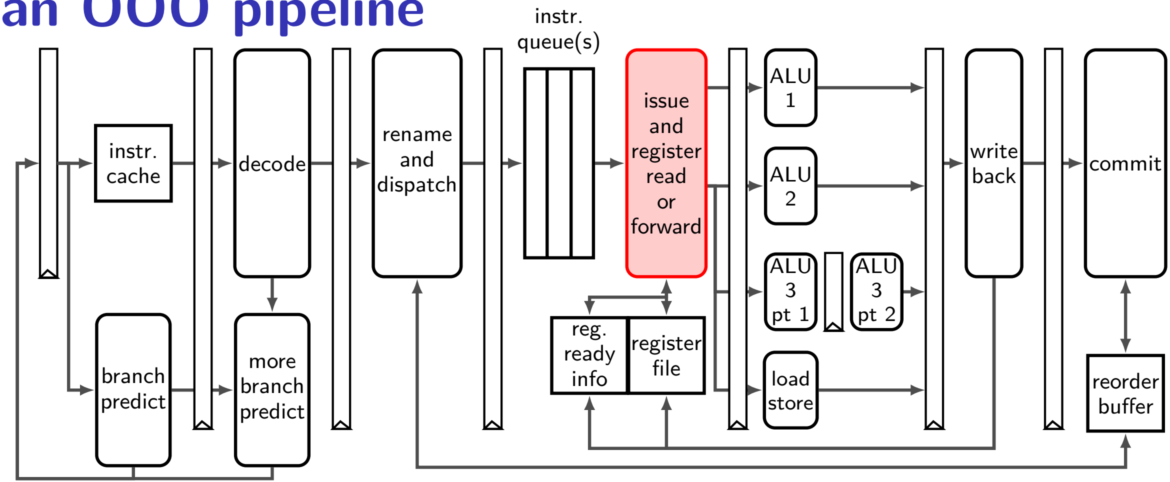
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



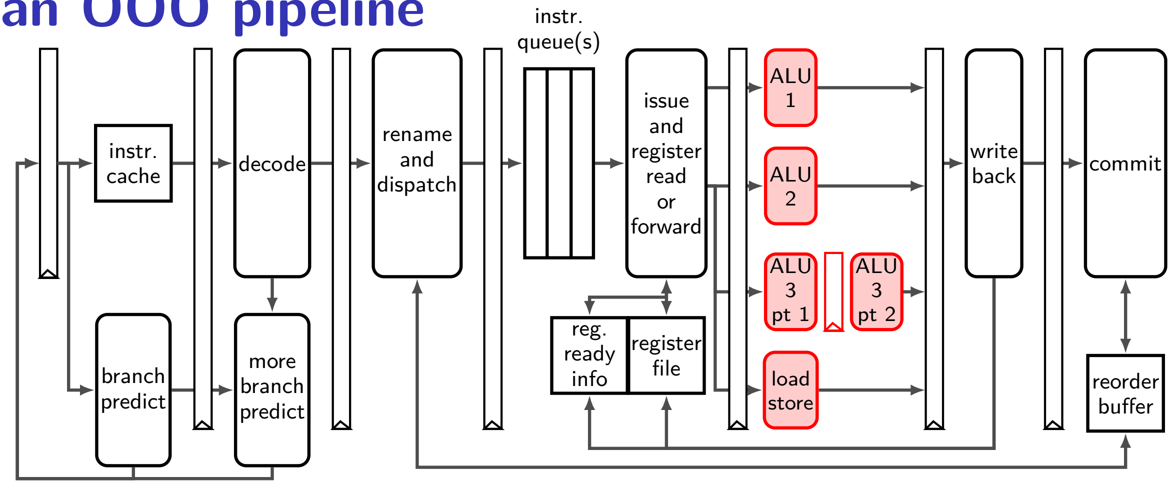
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



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

an OOO pipeline

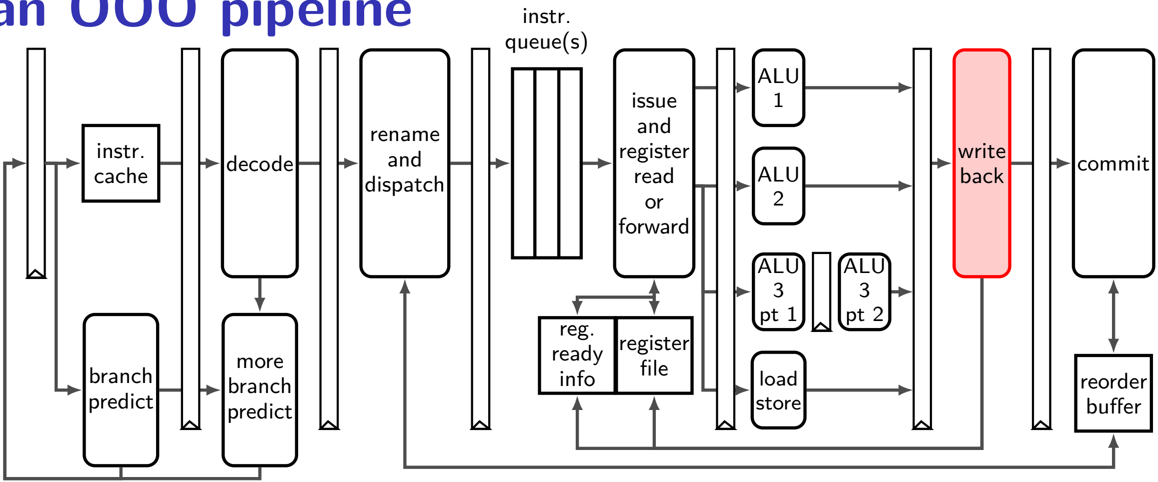


many *execution units* actually do math or memory load/store

some may have multiple pipeline stages

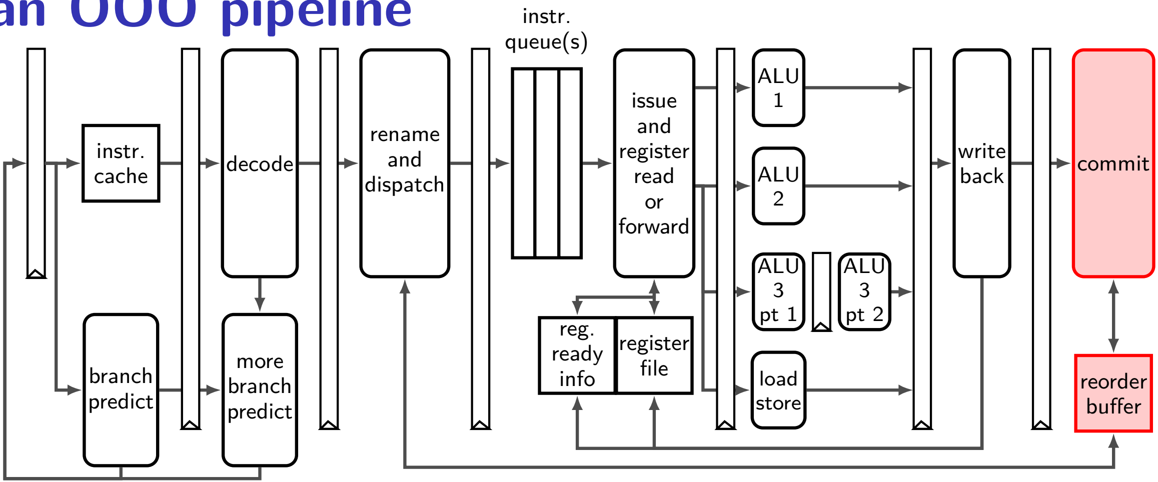
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



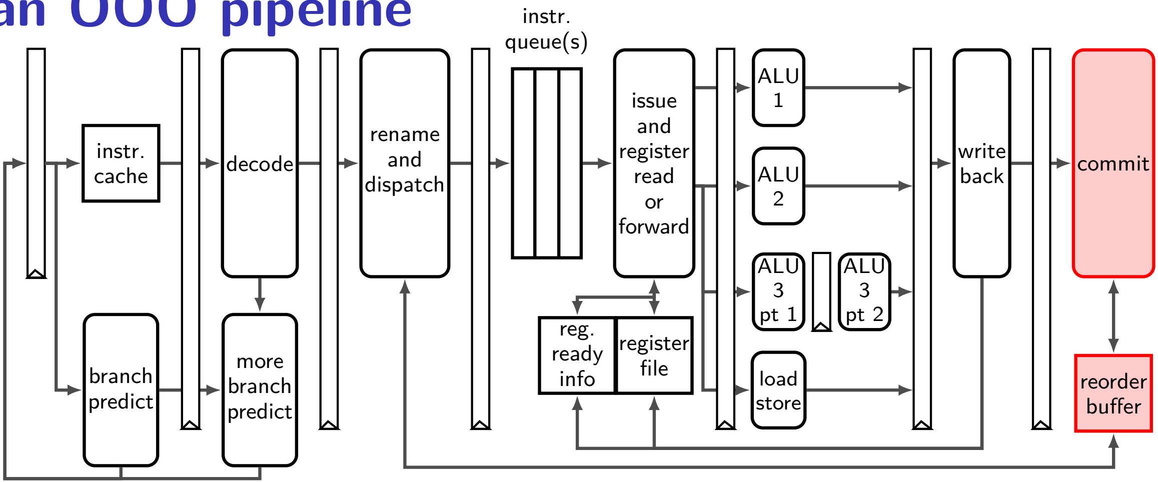
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



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

an OOO pipeline



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

an OOO pipeline diagram

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

an OOO pipeline diagram

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

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

an OOO pipeline diagram

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

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

an OOO pipeline diagram

cycle #	0	1	2	3	4	5	6	7	8	9	10	11
addq %r01, %r05	F	D	R	I	E	W	C					
addq %r02, %r05	F	D	R		I	E	W	C				
addq %r03, %r04		F	D	R	I	E	W	C				
cmpq %r0						I	E	W	C			
jne ...			F	D	R		I	E	W	C		
addq %r01, %r05			F	D	R	I	E	W		C		
addq %r02, %r05				F	D	R	I	E	W		C	
addq %r03, %r04				F	D	R		I	E	W	C	
cmpq %r04, %r08					F	D	R		I	E	W	C

commit instructions in order
waiting until next complete

an OOO pipeline diagram

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

register renaming

rename *architectural registers* to *physical registers*

architectural = part of instruction set architecture

different name for each version of architectural register

register renaming state

original

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

renamed

arch → phys register map

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

free reg list

%x18
%x20
%x21
%x23
%x24
…

register renaming state

original

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

renamed

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

arch → phys register map

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

free reg list

%x18
%x20
%x21
%x23
%x24
…

register renaming state

original

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

renamed

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

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

arch → phys register map

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

free reg list

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (1)

original

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

renamed

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

arch → phys register map

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

free reg list

~~%x18~~
~~%x20~~
%x21
%x23
%x24
…

register renaming example (1)

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

arch → phys register map

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

free reg list

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (1)

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

arch → phys register map

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

free reg list

~~`%x18`~~
~~`%x20`~~
~~`%x21`~~
`%x23`
`%x24`
…

register renaming example (2)

original

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

renamed

arch → phys register map

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

free regs

%x18
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

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

renamed

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

arch → phys register map

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

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

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

renamed

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

arch → phys register map

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

free regs

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

register renaming example (2)

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

arch → phys register map

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

could be that %rax = 8+%r11
could load before value written!
possible data hazard!
not handled via register renaming
option 1: run load+stores in order
option 2: compare load/store addresses

%x21
%x23
%x24
…

register renaming example (2)

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

arch → phys register map

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

free regs

~~%x18~~
~~%x20~~
%x21
%x23
%x24
…

register renaming example (2)

original

```
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
%r13	%x02
…	…

free regs

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (2)

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

arch → phys register map

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

free regs

~~%x18~~
~~%x20~~
~~%x21~~
~~%x23~~
%x24
…

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13` $\rightarrow$ `%x18`
`movq %r8, (%rax)`	`movq %x18, (%x04)` $\rightarrow$ (memory)
`subq %r8, %r11`	`subq %x18, %x07` $\rightarrow$ `%x20`
`movq 8(%r11), %r11`	`movq 8(%x20),` (memory) $\rightarrow$ `%x21`
`movq $100, %r8`	`movq $100` $\rightarrow$ `%x23`
`addq %r11, %r8`	$\boxed{\texttt{addq \%x21, \%x23} \rightarrow \texttt{\%x24}}$

arch $\rightarrow$ phys register map

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

free regs

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

register renaming exercise

original

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

renamed

arch → phys

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

free regs

%x18
%x20
%x21
%x23
%x24
…

an OOO pipeline diagram

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit

ALU 1

ALU 2

…

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit	*cycle# 1*	…
ALU 1	**1**	
ALU 2		

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit	*cycle# 1*	…
ALU 1	1	
ALU 2		

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit	*cycle#* 1	2	3	…
ALU 1	1	2	**3**	
ALU 2	—	—	—	

instruction queue and dispatch

instruction queue

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

scoreboard

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	ready
`%x04`	ready
`%x05`	ready
`%x06`	~~pending~~ ready
`%x07`	~~pending~~ ready
`%x08`	~~pending~~ ready
`%x09`	pending
`%x10`	pending
`%x11`	pending
`%x12`	pending
`%x13`	pending
…	…

execution unit	*cycle# 1*	*2*	*3*	…
ALU 1	1	2	3	
ALU 2	—	—	—	

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit	*cycle#* 1	2	3	4	…
ALU 1	1	2	3	**4**	
ALU 2	—	—	—	**6**	

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit	*cycle# 1*	*2*	*3*	*4*	…
ALU 1	1	2	3	4	
ALU 2	—	—	—	6	

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	…
ALU								
data cache								

↑
assume
1 cycle/access

an OOO pipeline diagram

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

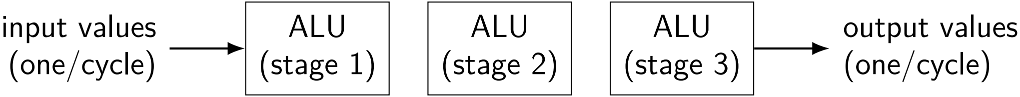
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



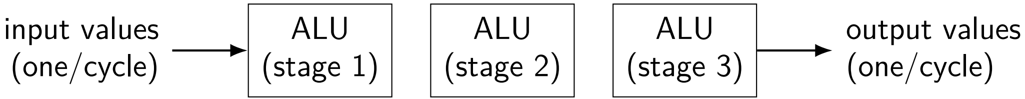
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



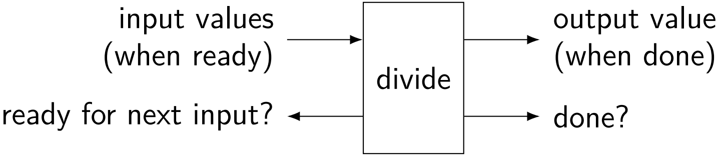
exercise: how long to compute $A \times (B \times (C \times D))$?

execution units AKA functional units (2)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:



instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

register renaming: missing pieces

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

one solution: translate to intructions with additional register parameters

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

bonus: can also translate complex instructions to simpler ones

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

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

OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

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

OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

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

some performance examples

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

about 30B instructions

my desktop: approx 2.65 sec

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

about 40B instructions

my desktop: approx 2.65 sec

some performance examples

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

about 30B instructions

my desktop: approx 2.65 sec

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

about 40B instructions

my desktop: approx 2.65 sec

backup slides

beyond local 1-bit predictor

can predict using more historical info

whether taken last several times

- example: taken 3 out of 4 last times $\rightarrow$ 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: ministack of return addresses

predicting ret — ministack in processor registers

- push on ministack on call; pop on ret

ministack 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

stack in memory

baz return address
bar return address
foo return address

(partial?) stack
in CPU registers

4-entry return address stack

4-entry return address stack in CPU

current index

$\boxed{1}$

idx	saved return addresses
0	0x12345
1	0x44432
2	0x44F92
3	0x22331

next prediction for ret

next saved return address from call

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

on `ret`: read prediction from index, decrement index

pipeline with different hazards

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

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

pipeline with different hazards

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

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

addq/andq is hazard with 5-stage pipeline

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

pipeline with different hazards

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

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

more hazards with more pipeline stages

performance

hypothetical instruction mix

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

* — ignoring data hazards

performance

hypothetical instruction mix

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

* — ignoring data hazards

hazards versus dependencies

dependency — X needs result of instruction Y?

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

hazard — will it not work in some pipeline?

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

ex.: dependencies and hazards (1)

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

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

ex.: dependencies and hazards (1)

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

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

ex.: dependencies and hazards (1)

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

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

ex.: dependencies and hazards (1)

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

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

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

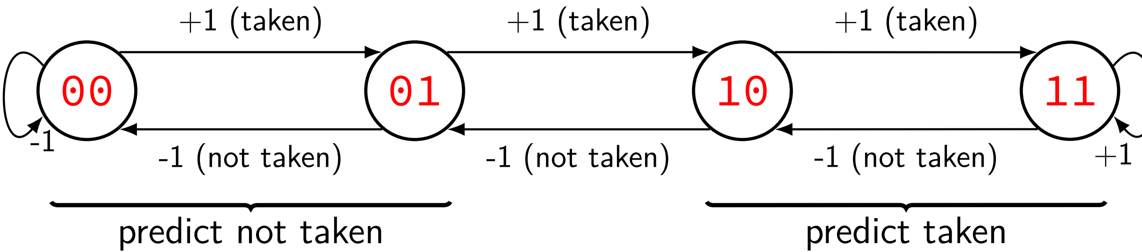
1-bit predictor: wrong about 2% of the time

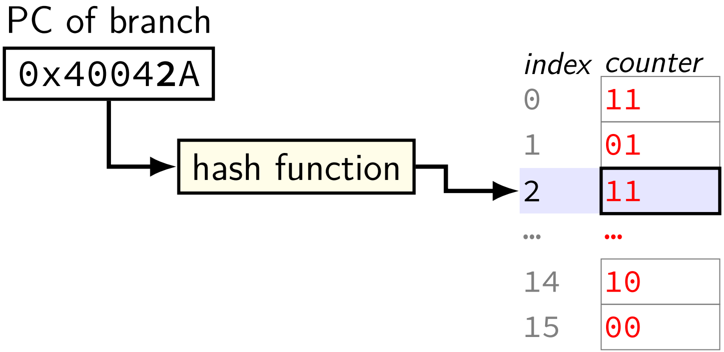
- 1% when branch not taken
- 1% of taken branches right after branch not taken

new predictor: wrong about 1% of the time

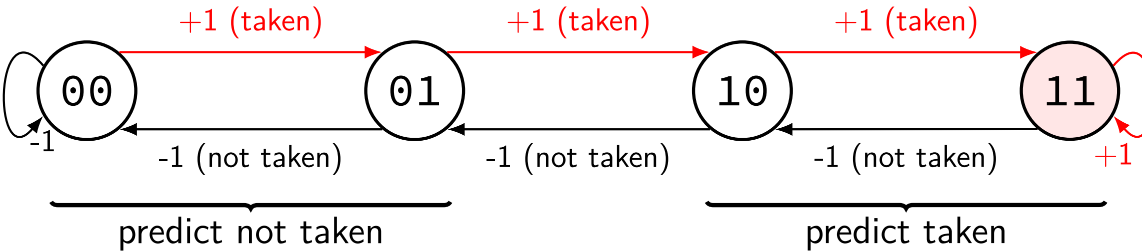
- 1% when branch not taken

2-bit saturating counter



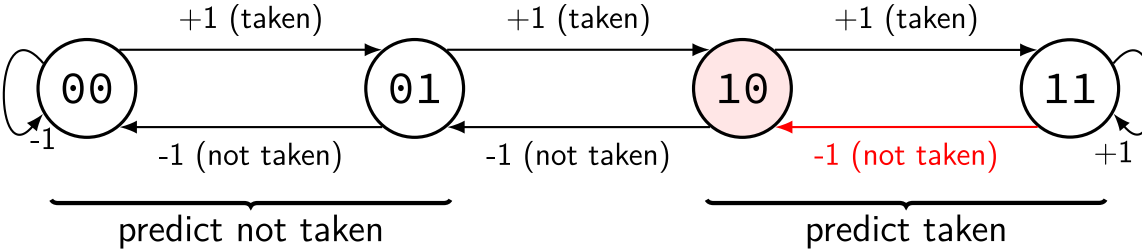


2-bit saturating counter



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;
}
```

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)

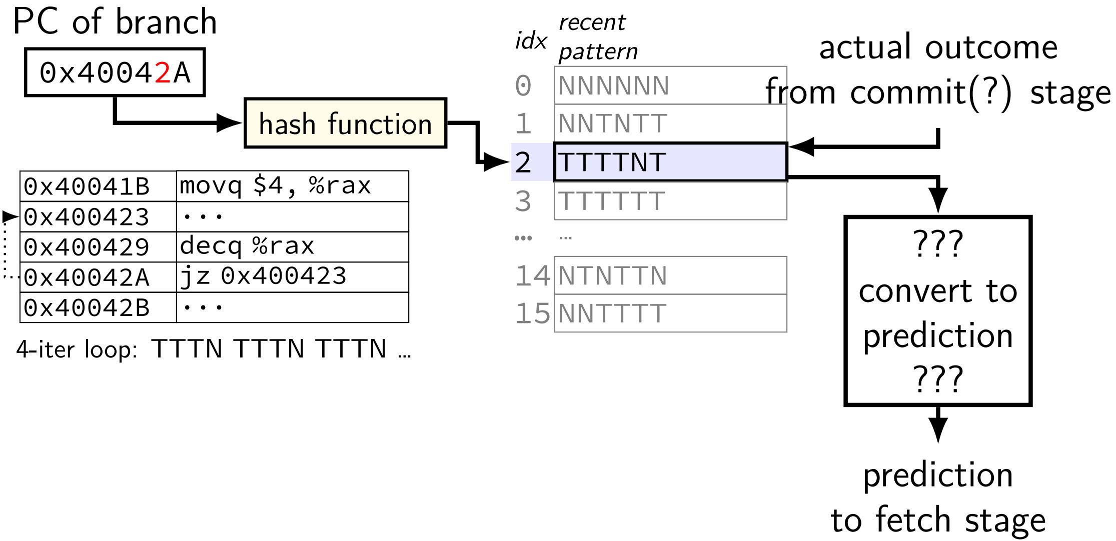
goal: take advantage of recent pattern to make predictions

just saw 'NTTTNT'? predict T next

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

...

local pattern predictor (incomplete)



local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

idx	recent pattern
0	NNNNNN
1	NNTNTT
2	TTTTNT
3	TTTTTT
…	…
14	NTNTTN
15	NNTTTT

actual outcome from commit(?) stage

??? convert to prediction ???

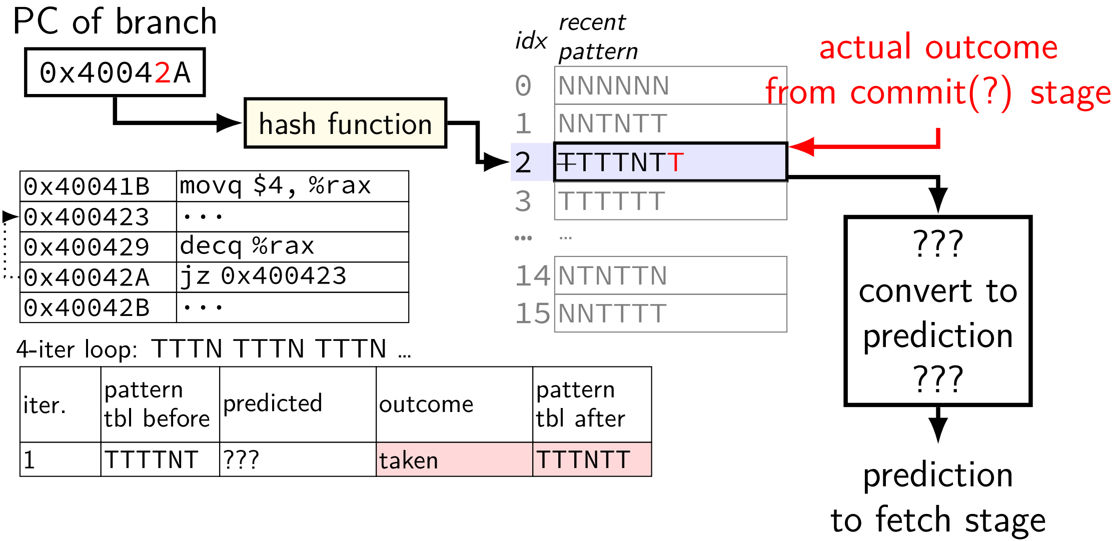
prediction to fetch stage

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

local pattern predictor (incomplete)



idx	recent pattern
0	NNNNNN
1	NNTNTT
2	~~T~~TTTNTT
3	TTTTTT
…	…
14	NTNTTN
15	NNTTTT

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

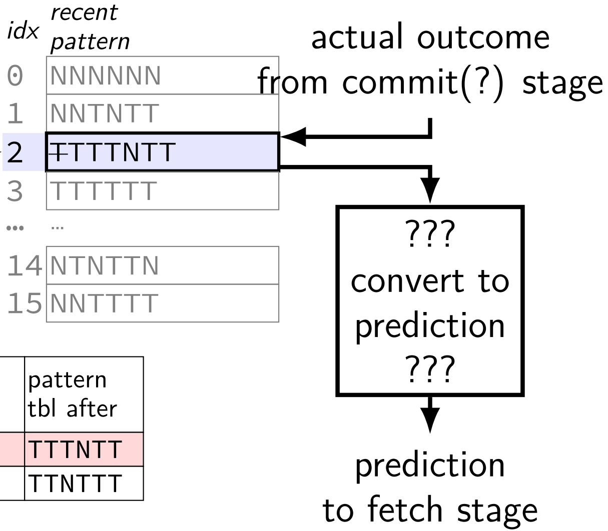
4-iter loop: TTTN TTTN TTTN …

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

local pattern predictor (incomplete)

PC of branch

0x40042A → hash function

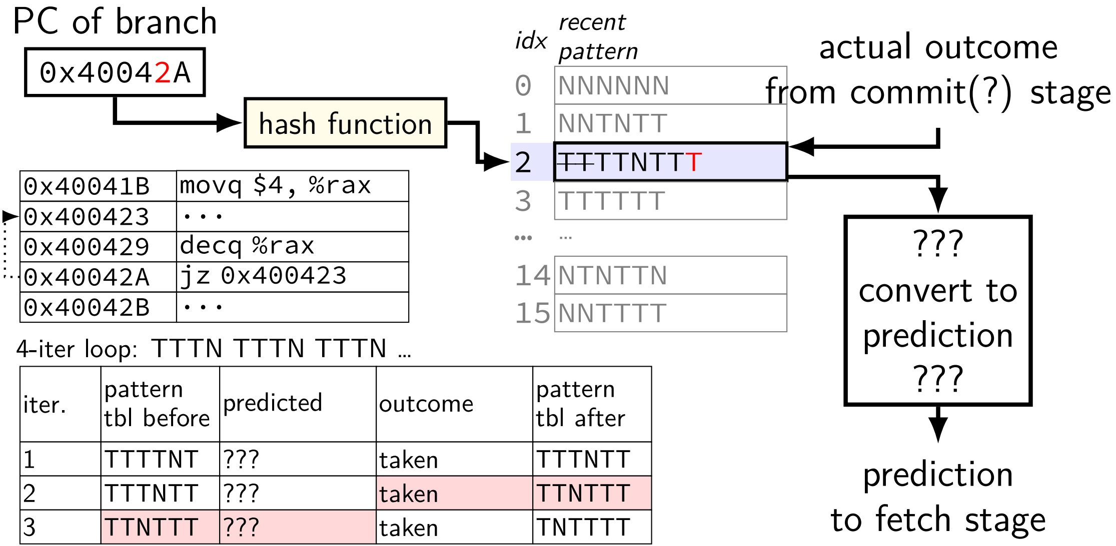


0x40041B	movq $4, %rax
0x400423	...
0x400429	decq %rax
0x40042A	jz 0x400423
0x40042B	...

4-iter loop: TTTN TTTN TTTN ...

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

local pattern predictor (incomplete)



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
2	TTTNTT	???	taken	TTNTTT
3	TTNTTT	???	taken	TNTTTT

local pattern predictor (incomplete)

PC of branch

`0x40042A`

hash function

idx	*recent pattern*
0	NNNNNN
1	NNTNTT
2	~~TT~~TTNTTT
3	TTTTTT
…	…
14	NTNTTN
15	NNTTTT

actual outcome from commit(?) stage

???
convert to prediction
???

prediction to fetch stage

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
2	TTTNTT	???	taken	TTNTTT
3	TTNTTT	???	taken	TNTTTT
4	TNTTTT	???	not taken	NTTTTN
1	NTTTTN	???	taken	TTTTNT
2	TTTTNT	???	taken	TTTNTT

recent pattern to prediction?

easy cases:

just saw TTTTTT: predict T

just saw NNNNNN: predict N

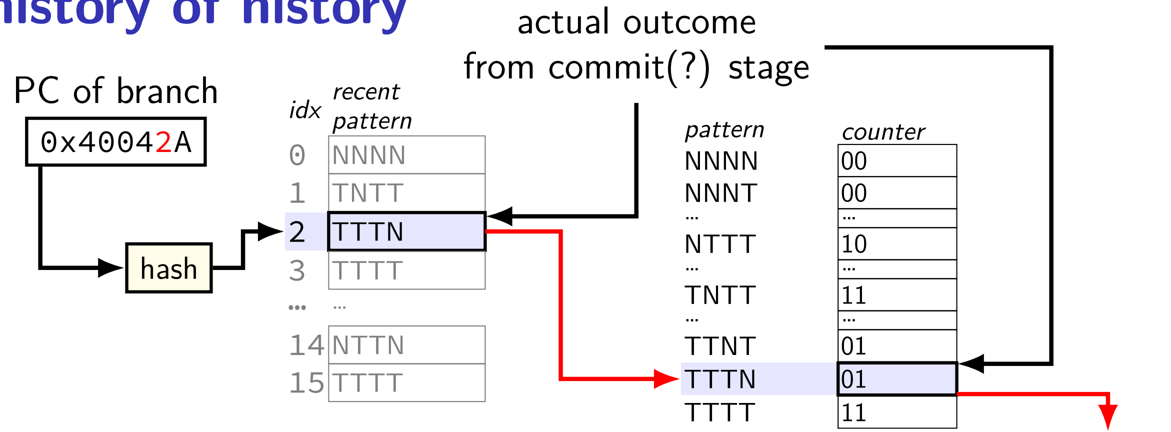
just saw TNTNTN: predict T

hard cases:

TTNTTTT

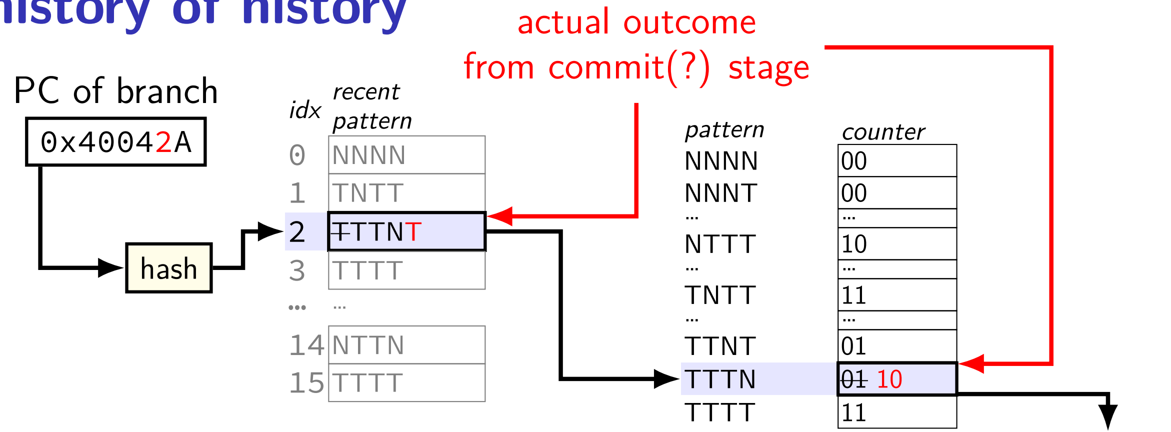
- predict T? loop with many iterations (NTTTTTTTNTTTTTTTNTTTTTT...)
- predict T? if statement mostly taken (TTTTNTTNTTTTTTTTTTTNTTTT...)
- predict N? loop with 5 iterations (NTTTTNTTTTNTTTTNTTTTNTT...)

history of history



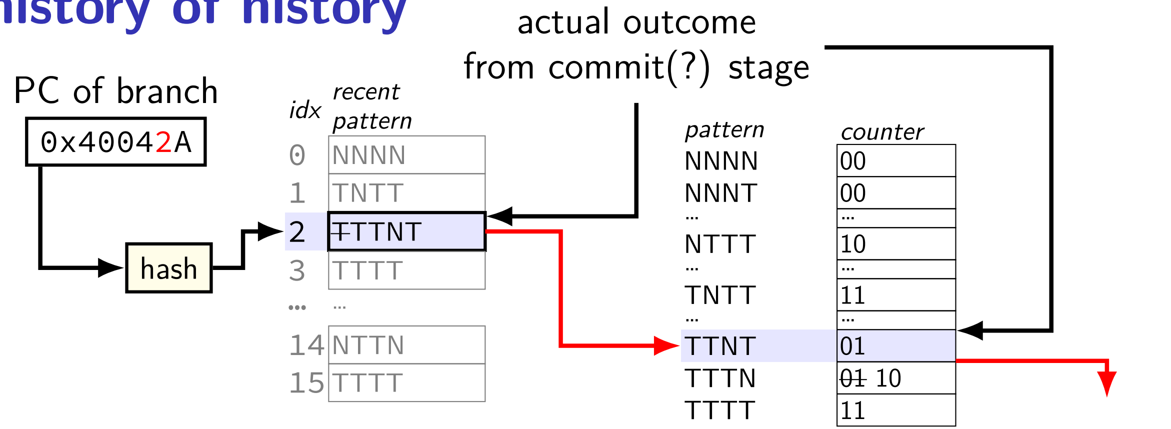
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT

history of history



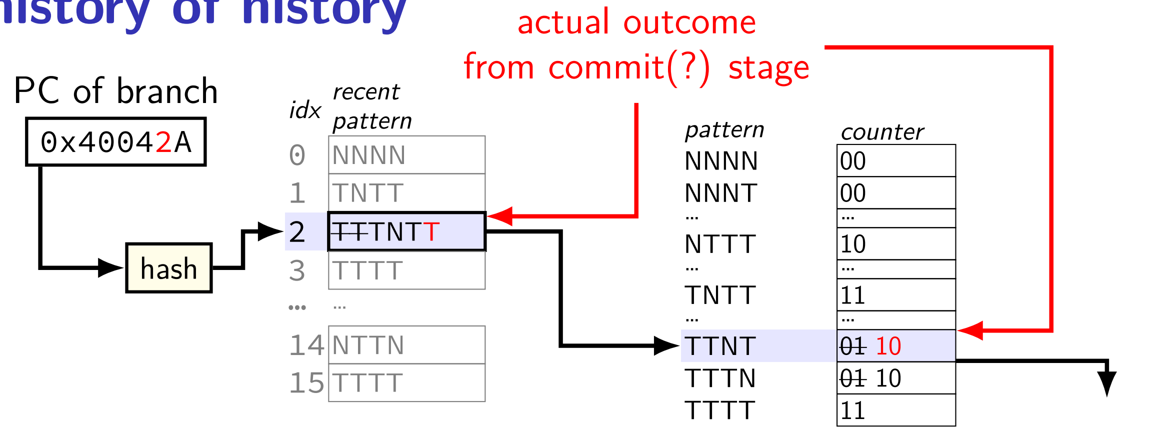
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT

history of history



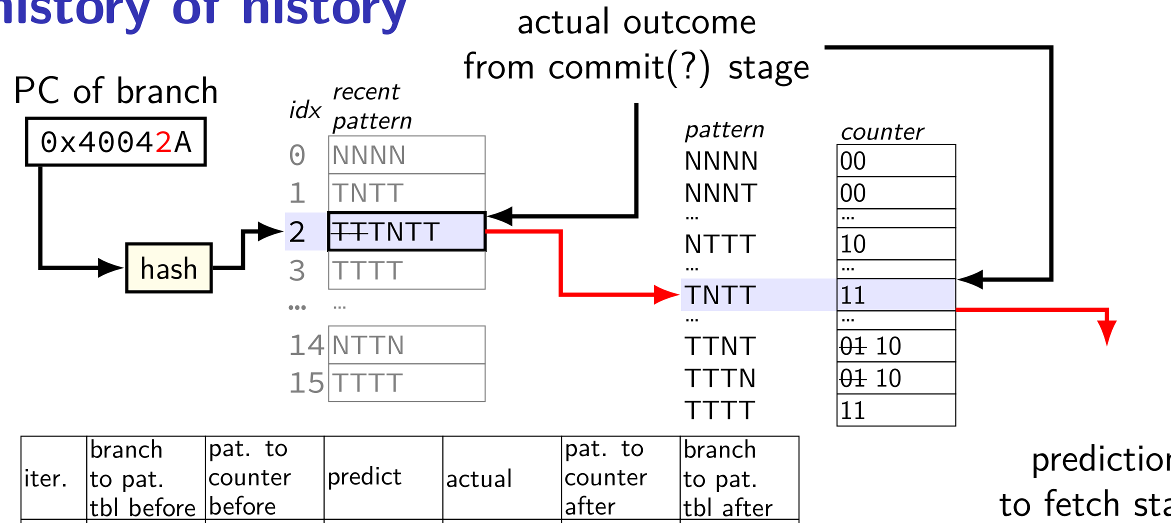
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT

history of history



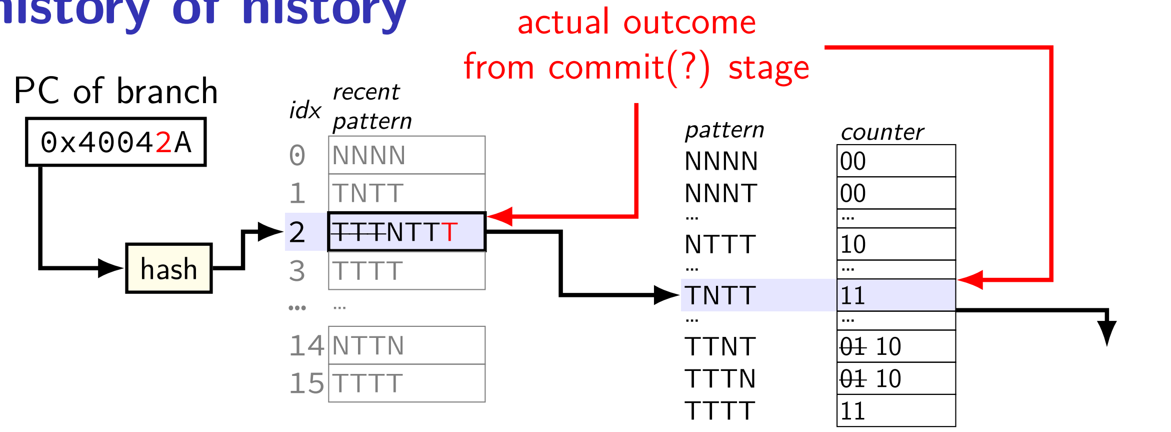
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT

history of history



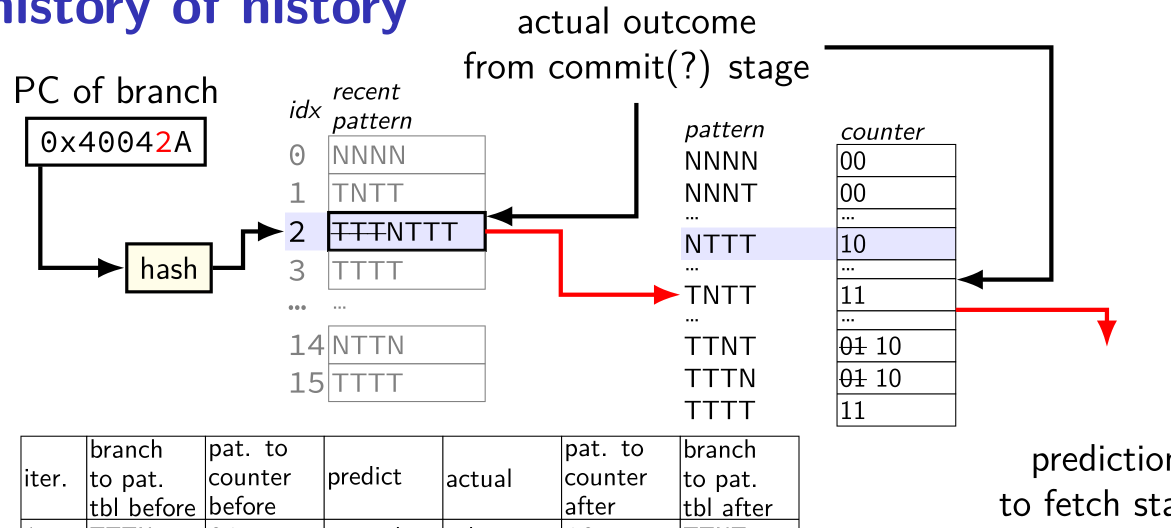
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

history of history



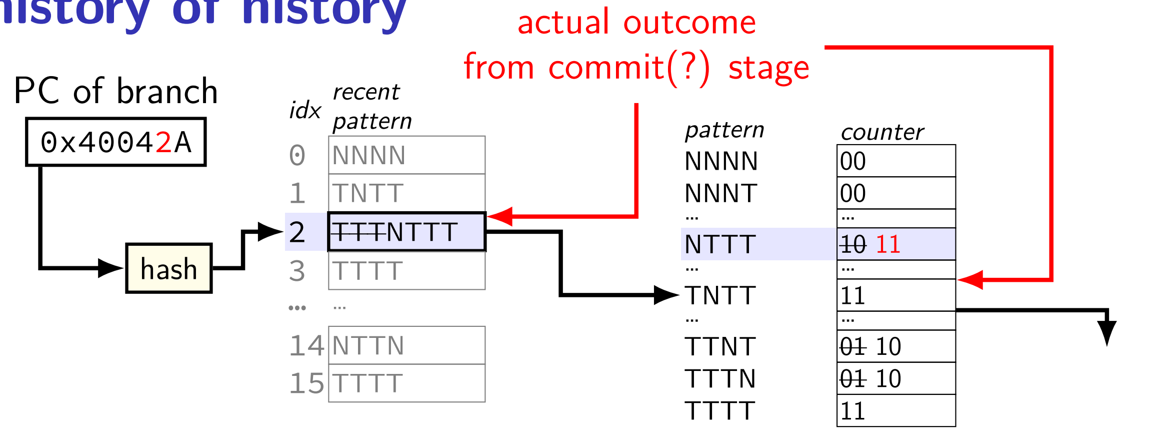
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

history of history



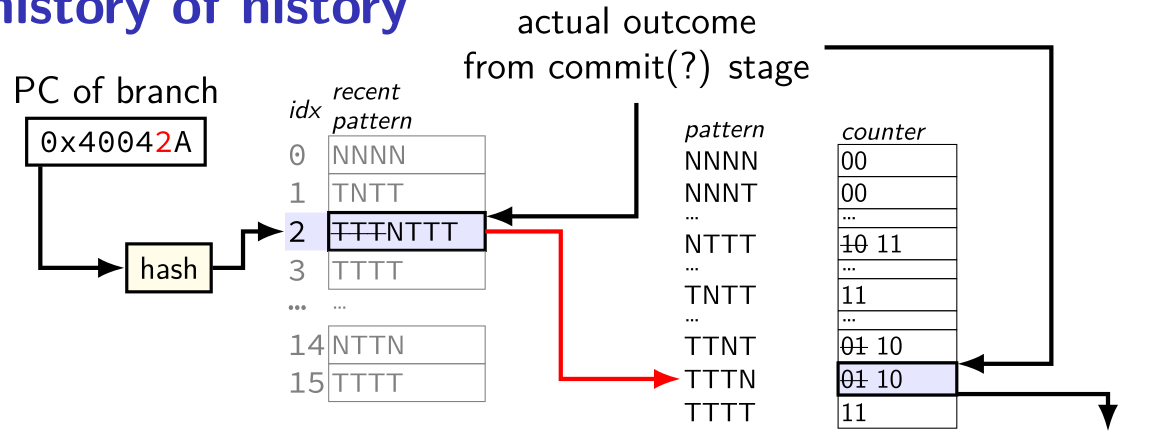
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

history of history



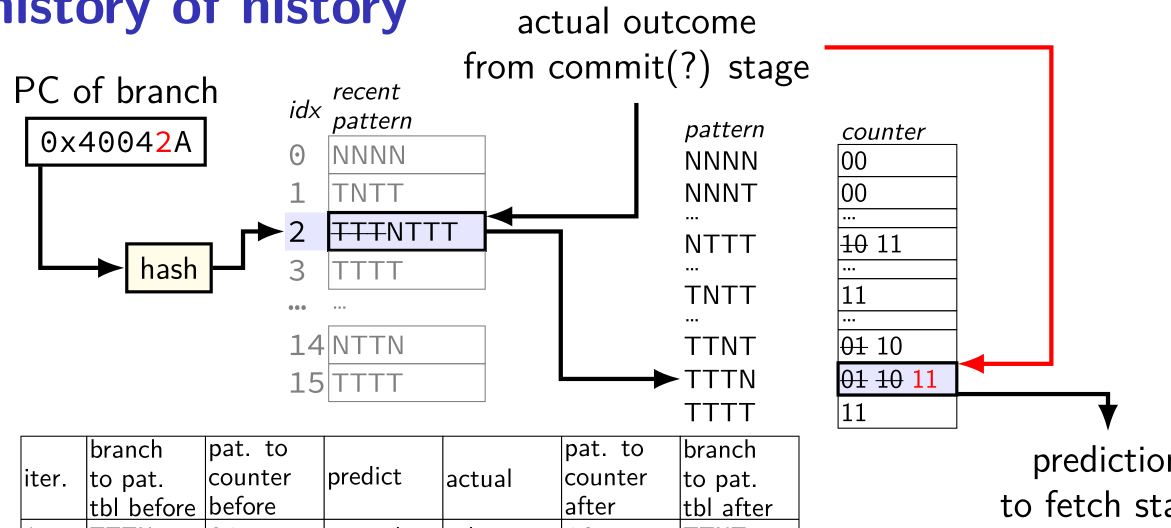
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT

history of history



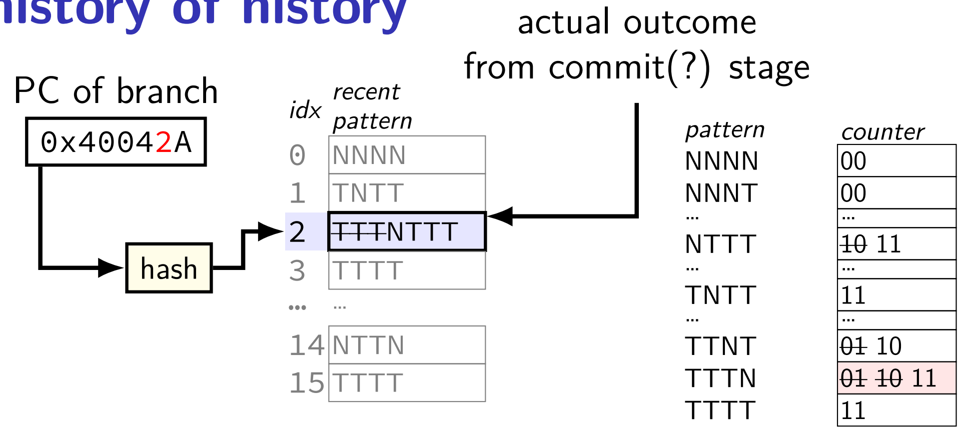
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT
4	NTTT	01	not taken	taken	10	TTTT
1	TTTN	10	taken	taken	11	TTNT

history of history



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
1	TTTN	10	taken	taken	11	TTNT

history of history



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
1	TTTN	10	taken	taken	11	TTNT

predictior
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

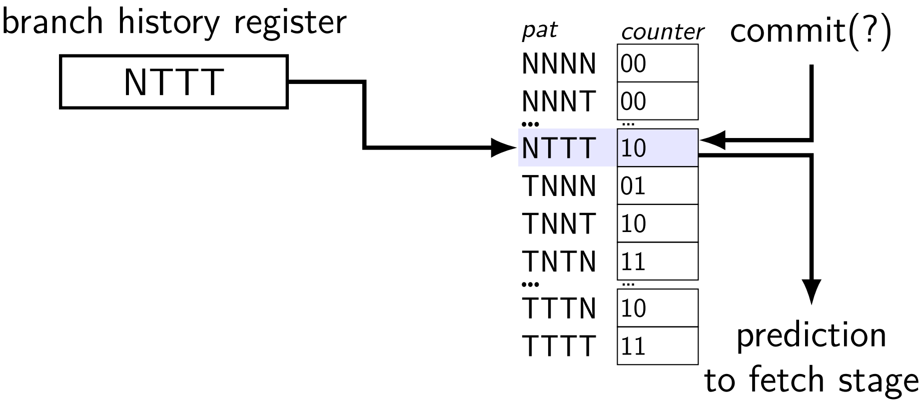
global history predictor: idea

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)



global history predictor (1)

branch history register

NTTT

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

pat	*counter*
NNNN	00
NNNT	00
...	...
NTTT	10
TNNN	01
TNNT	10
TNTN	11
...	...
TTTN	10
TTTT	11

outcome from commit(?)

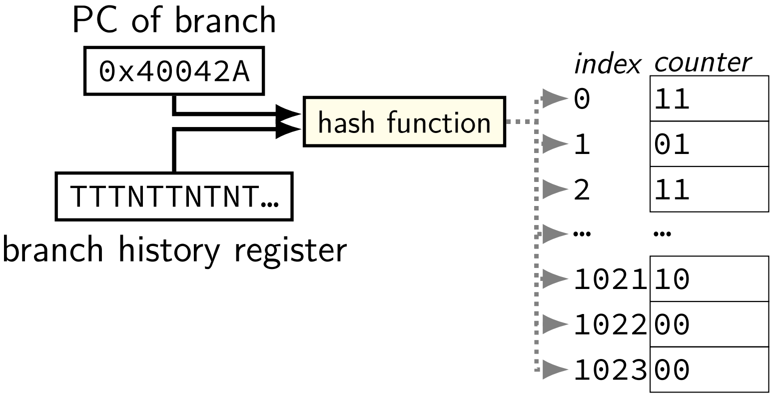
prediction to fetch stage

iter./ branch	history before	counter before	predict	outcome	counter after	history after
0/mod 2	NTTT	10	taken	taken	11	TTTT
0/loop	TTTT			taken		TTTT
1/mod 2	TTTT			not taken		TTTN
1/error	TTTN			not taken		TTNN
1/loop	TNNT			taken		NNTT
2/mod 2	NNTT			taken		NTTT
2/loop	TTTT			taken		TTTT

correlating predictor

global history *and* local info good together

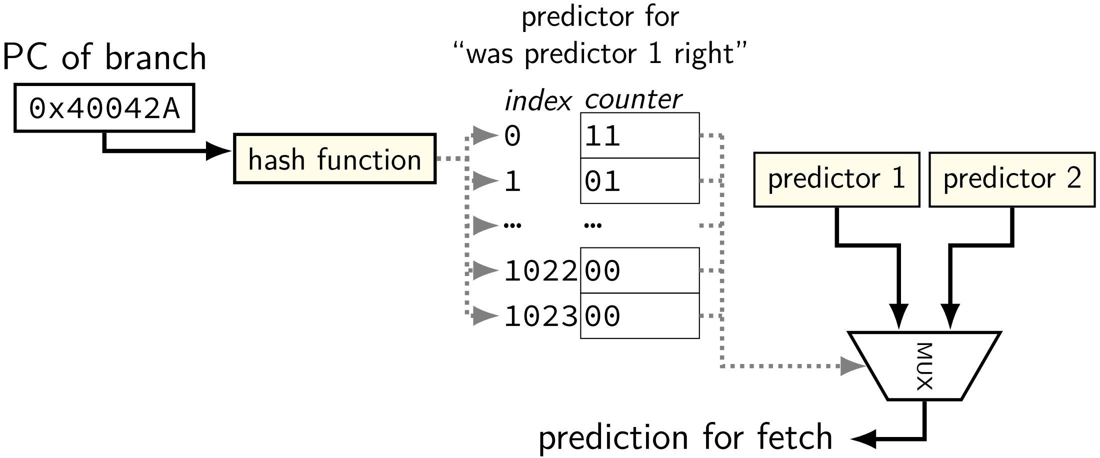
one idea: combine history register + PC ("gshare")



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

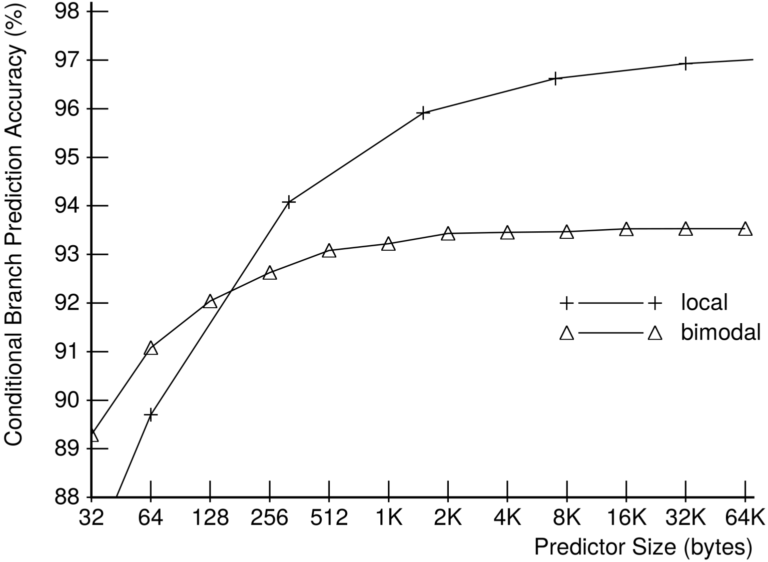
(not representative of modern workloads?)

rate for conditional branches on benchmark

variable table sizes

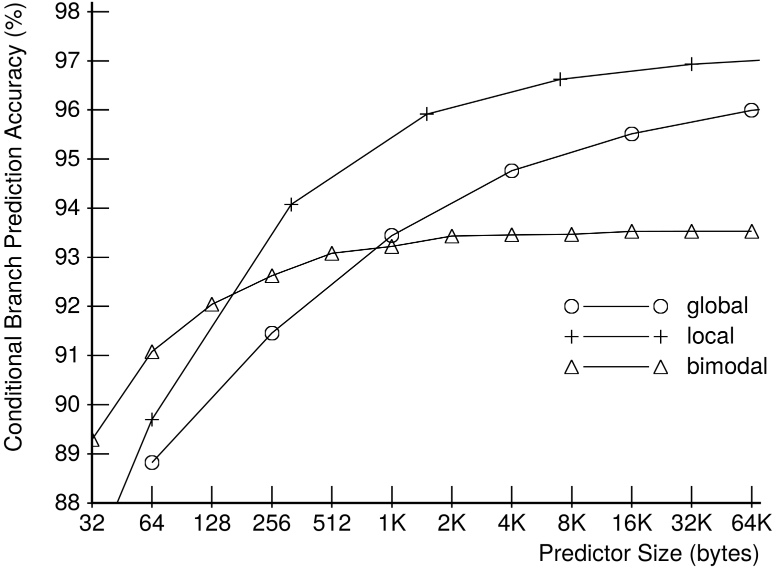
2-bit ctr + local history

from McFarling, "Combining Branch Predictors" (1993)



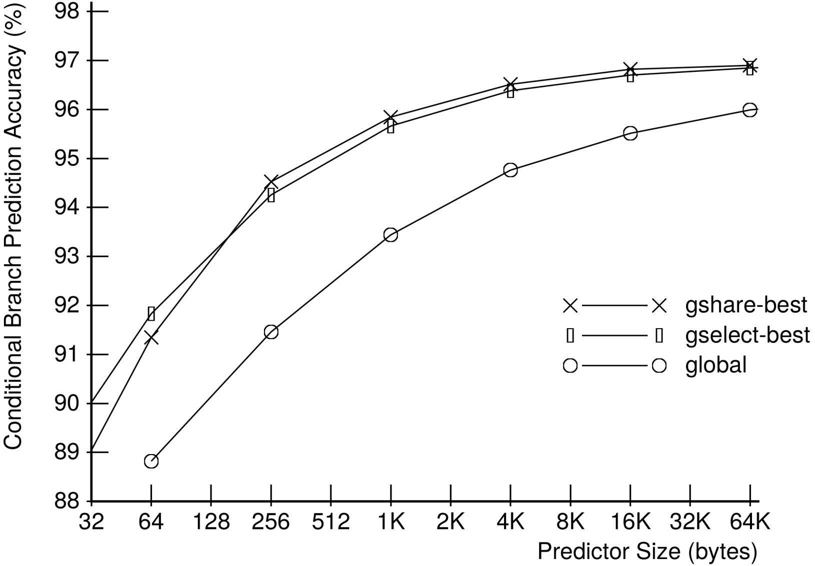
2-bit (bimodal) + local + global hist

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

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 for_loop_bottom
    add %edi, %eax
foo_loop_bottom:
    dec %edi            // edi = edi - 1
    jg for_loop_top // if (edi > 0) goto for_loop_top
    ret
```

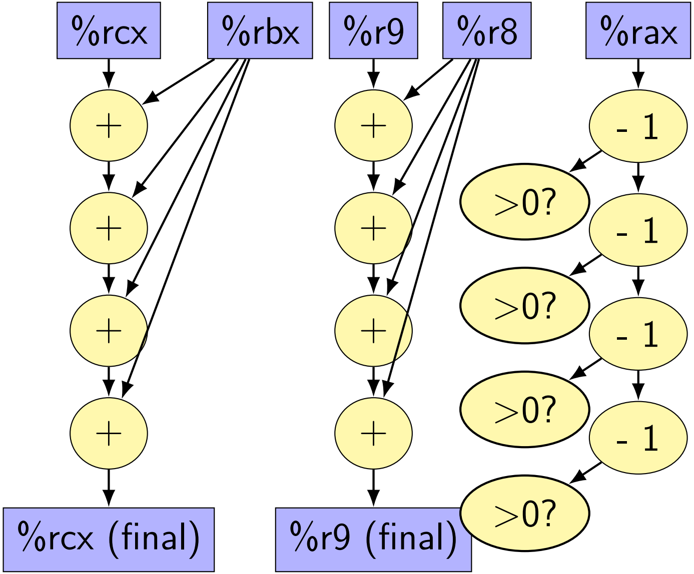
suppose %edi $= 3$ (initially)

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

how many mispreditions for je? for jg?

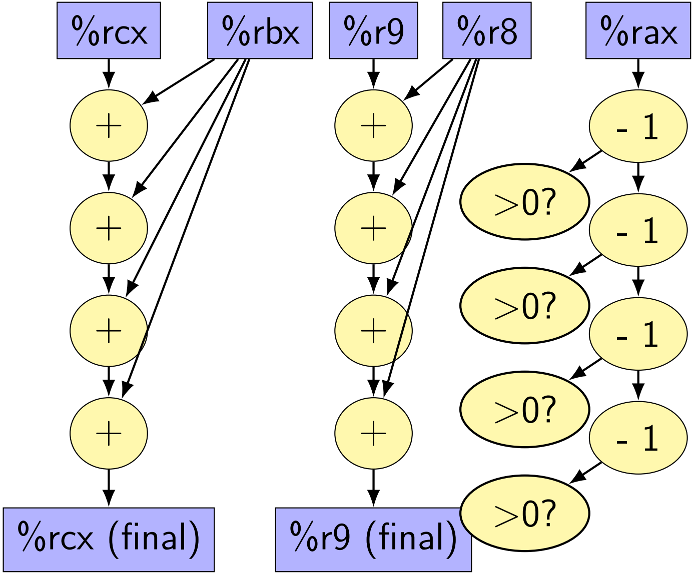
backup slides

data flow model and limits (1)




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

data flow model and limits (1)



each yellow box = instruction

arrows = dependences

instructions only executed when dependencies ready

reassociation

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

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

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

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

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

reassociation

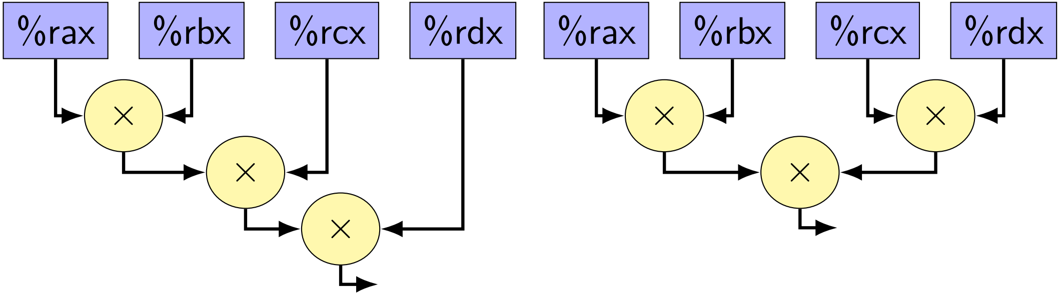
with pipelined, 5-cycle latency multiplier; how long does each take to compute?

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

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

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

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



Intel Skylake OOO design

2015 Intel design — codename 'Skylake'

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

4 ALU functional units

- but some can handle more/different types of operations than others

2 load functional units

- but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer

- determines how far ahead branch mispredictions, etc. can happen

indirect branch prediction

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

BTB can provide a prediction

but can do better with more context

example—predict based on other recent computed jumps

- good for polymophic method calls

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

an OOO pipeline diagram

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

reorder buffer: on rename

arch → phys reg
for new instrs

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

free list

%x19
%x23
…
…

reorder buffer: on rename

arch → phys reg
for new instrs

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

free list

%x19
%x23
...
...

reorder buffer (ROB)

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

reorder buffer contains instructions started,
but not fully finished new entries created on rename
(not enough space? stall rename stage)

reorder buffer: on rename

arch → phys 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 →

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		

add here on rename →

place newly started instruction at end of buffer
remember at least its destination register
(both architectural and physical versions)

reorder buffer: on rename

arch → phys 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 →

next renamed instruction goes in next slot, etc.

reorder buffer: on rename

arch → phys 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

add here on rename

reorder buffer (ROB)

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

reorder buffer: on commit

arch → phys. 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

arch → phys. 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

arch → phys. reg
for new instrs

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

free list

~~%x19~~
%x13
…
…

arch → phys reg
for committed

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

remove
here →
on commit

reorder buffer (ROB)

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

commit stage tracks architectural to physical register map for committed instructions

reorder buffer: on commit

arch → phys. reg
for new instrs

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

free list

~~%x19~~
%x13
…
%x23

arch → phys reg
for committed

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

remove
here →
on commit

reorder buffer (ROB)

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

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

reorder buffer: on commit

arch → phys. reg
for new instrs

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

free list

~~%x19~~
%x13
…
%x23

arch → phys reg
for committed

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

remove here
when committed

reorder buffer (ROB)

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

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

reorder buffer: commit mispredict (one way)

arch → phys reg
for new instrs

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

arch → phys reg
for committed

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

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

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

reorder buffer: commit mispredict (one way)

arch → phys reg
for new instrs

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

arch → phys reg
for committed

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

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

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

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

reorder buffer: commit mispredict (one way)

arch → phys reg
for new instrs

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

arch → phys 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)

arch → phys reg
for new instrs

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

arch → phys 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
…
…

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

better? alternatives

can take snapshots of register map on each branch

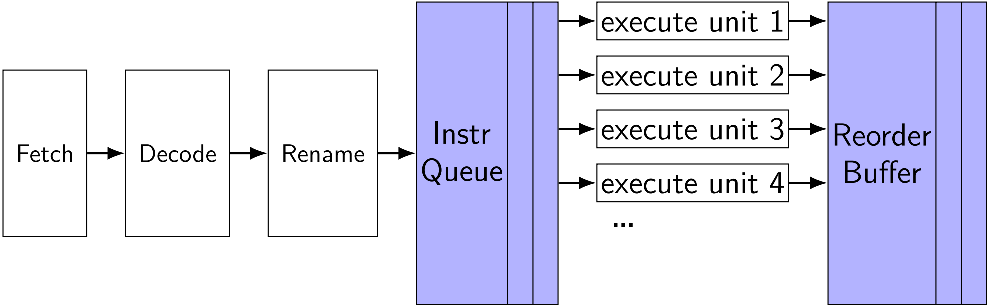
- don't need to reconstruct the table
- (but how to efficiently store them)

can reconstruct register map before we commit the branch instruction

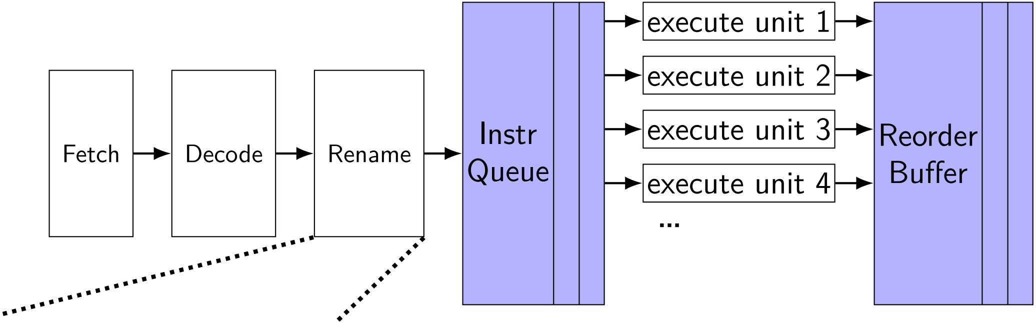
- need to let reorder buffer be accessed even more?

can track more/different information in reorder buffer

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



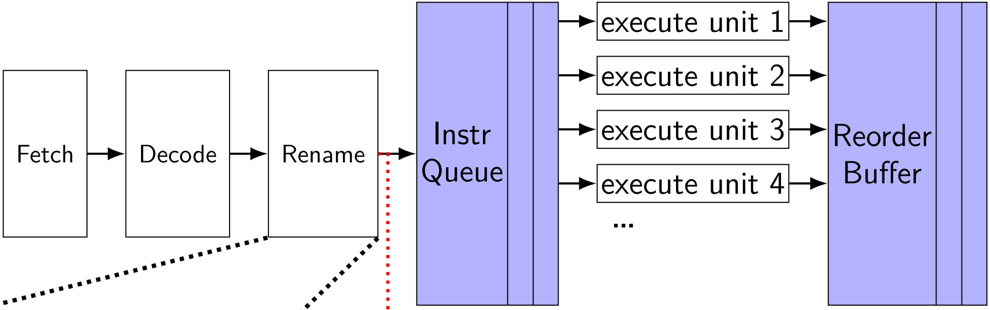
free regs

X19
X23
...

for new instrs

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

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

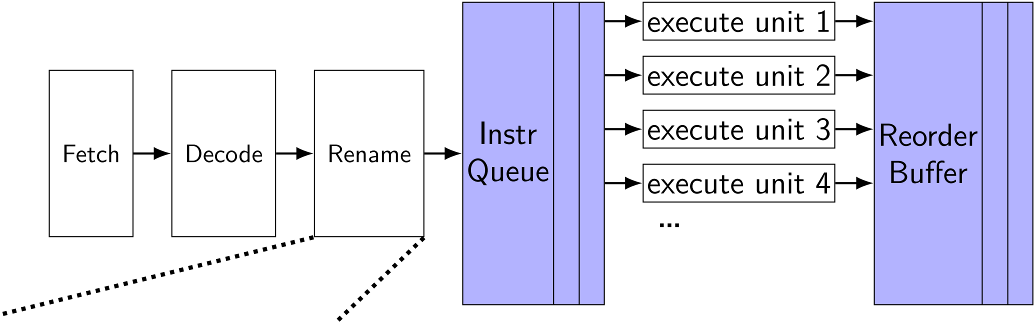
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

done instrs committed in order

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		
...	...	...	...	...

new instrs added

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

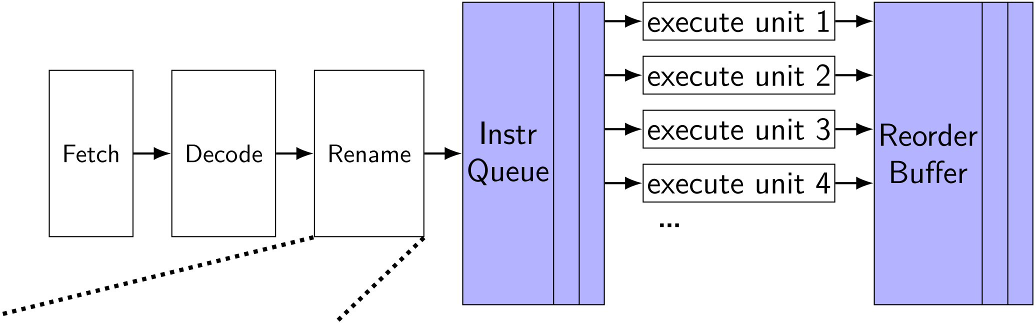
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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

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

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

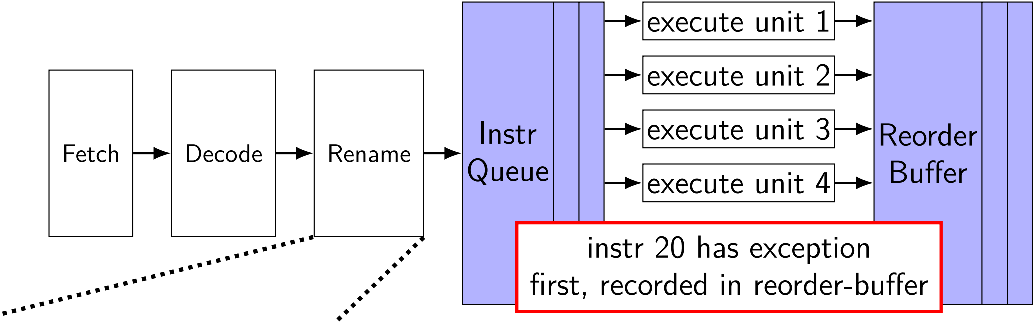
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

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

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

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

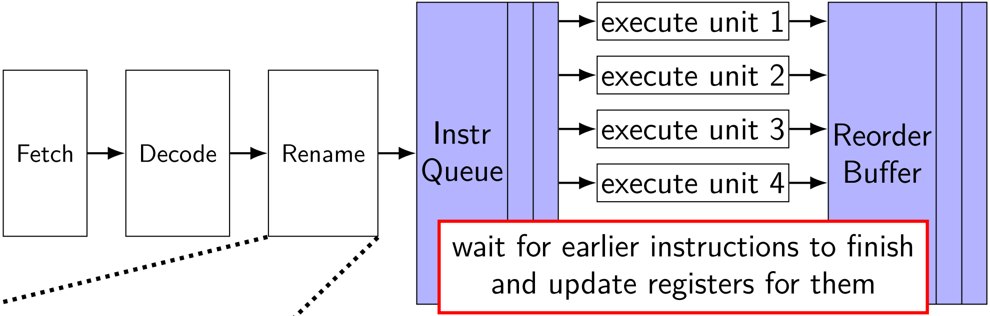
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

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

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

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

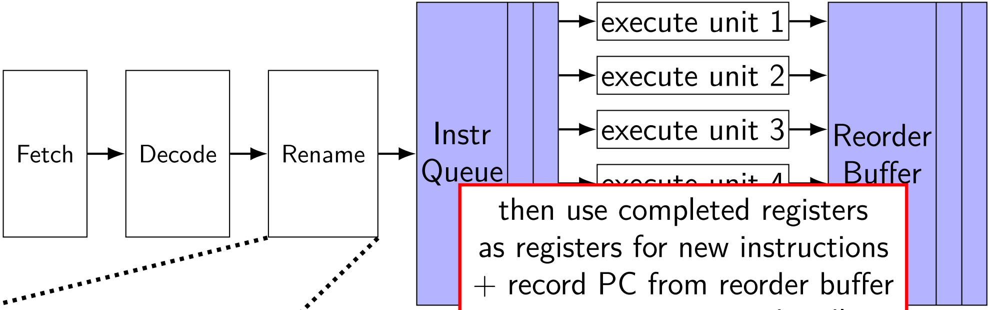
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

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

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

exceptions and OOO (one strategy)



then use completed registers
as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

X19
X23
…

for new instrs

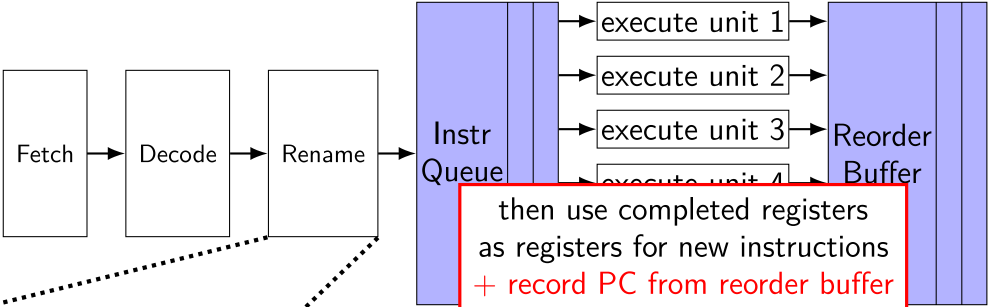
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complet

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

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

exceptions and OOO (one strategy)



then use completed registers
as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

X19
X23
...

for new instrs

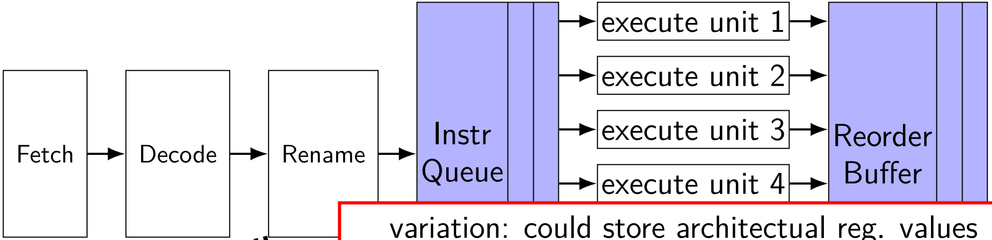
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
...	...

for complet

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

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

exceptions and OOO (one strategy)



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

free regs

X19
X23
…

for new instrs

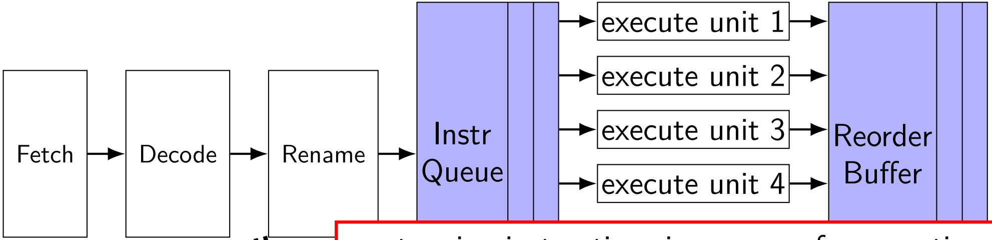
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

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

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

exceptions and OOO (one strategy)



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

free regs

X19
X23
…

for new instrs

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

for complete instrs

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

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

handling memory accesses?

one idea:

list of done + uncommited loads+stores

execute load early + double-check on commit
- have data cache watch for changes to addresses on list
- if changed, treat like branch misprediction

loads check list of stores so you read back own values

actually finish store on commit
- maybe treat like branch misprediction if conflict?

the open-source BROOM pipeline

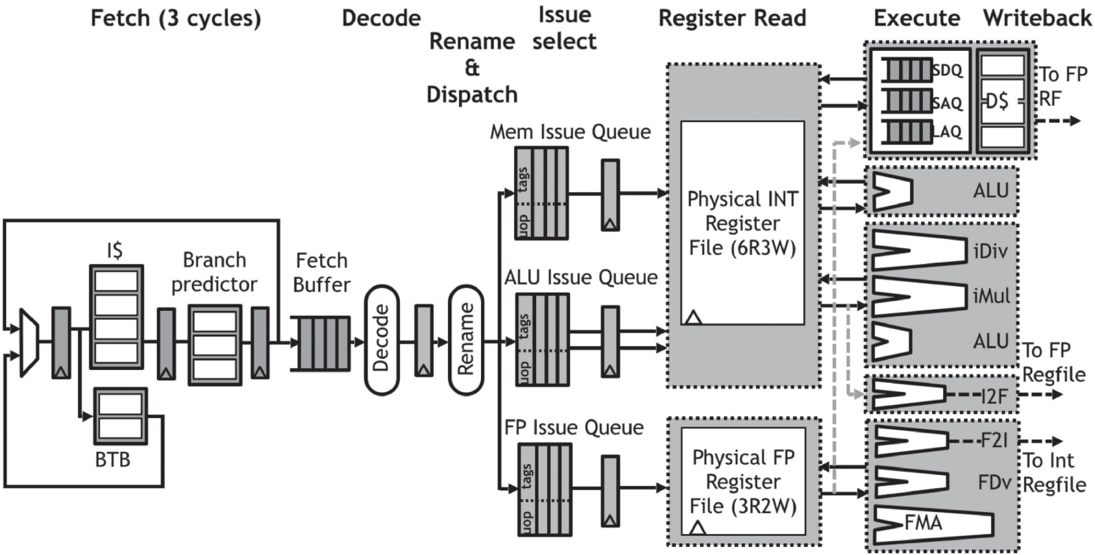
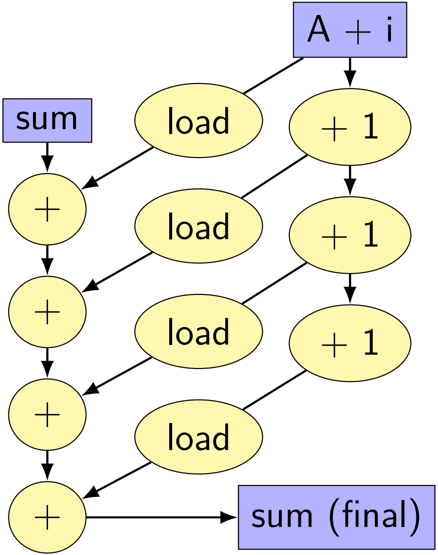


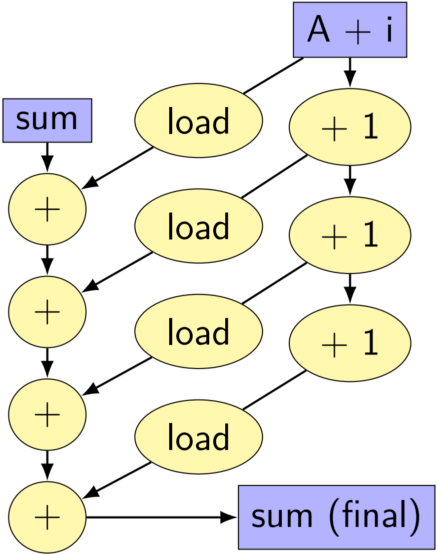
Figure from Celio et al., "BROOM: An Open Source Out-Of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS"

data flow model and limits



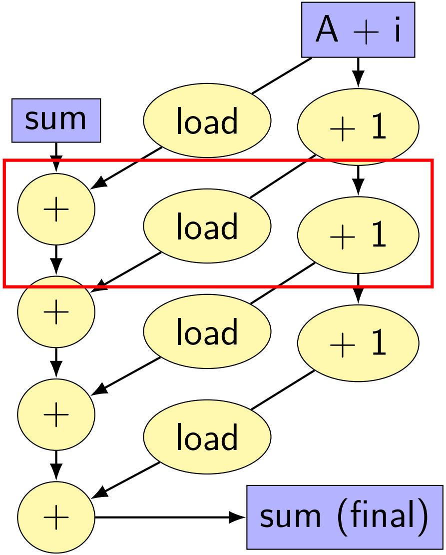

```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



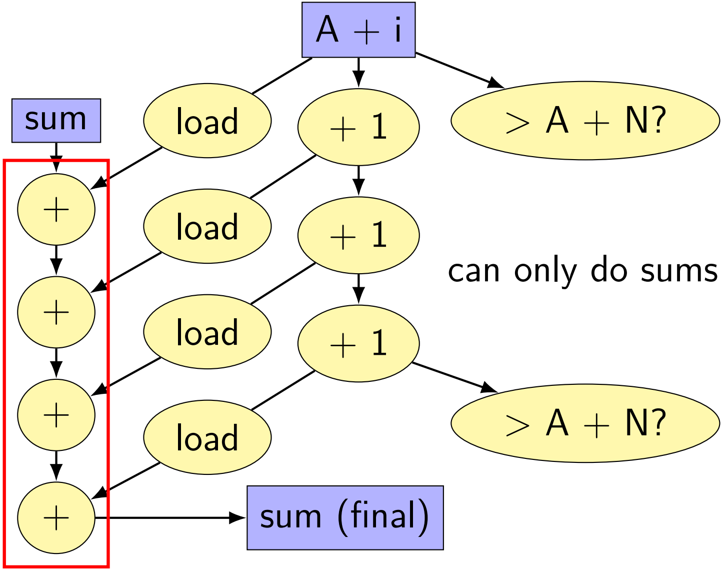
each yellow box = instruction
arrows = dependences
instructions only executed when dependencies

data flow model and limits



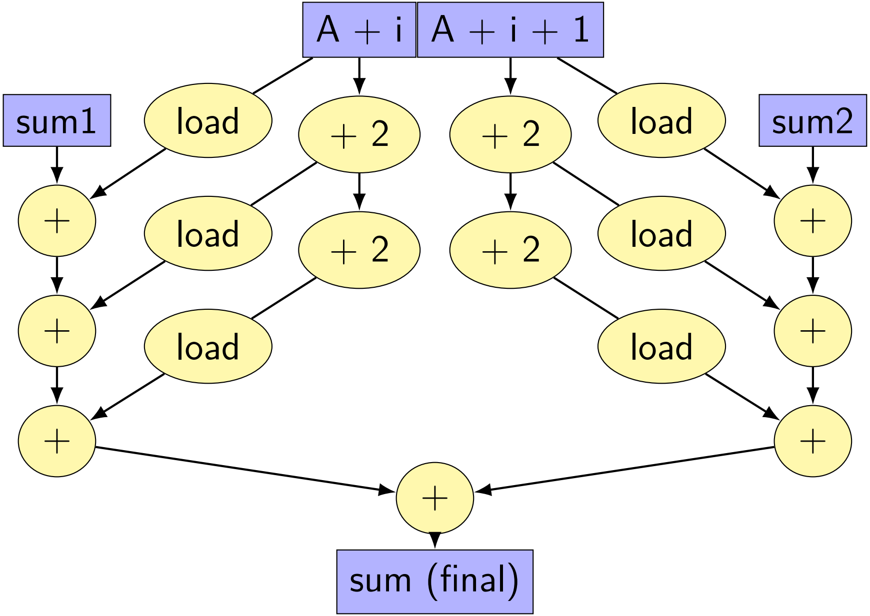
three ops/cycle (if each one cycle)

data flow model and limits

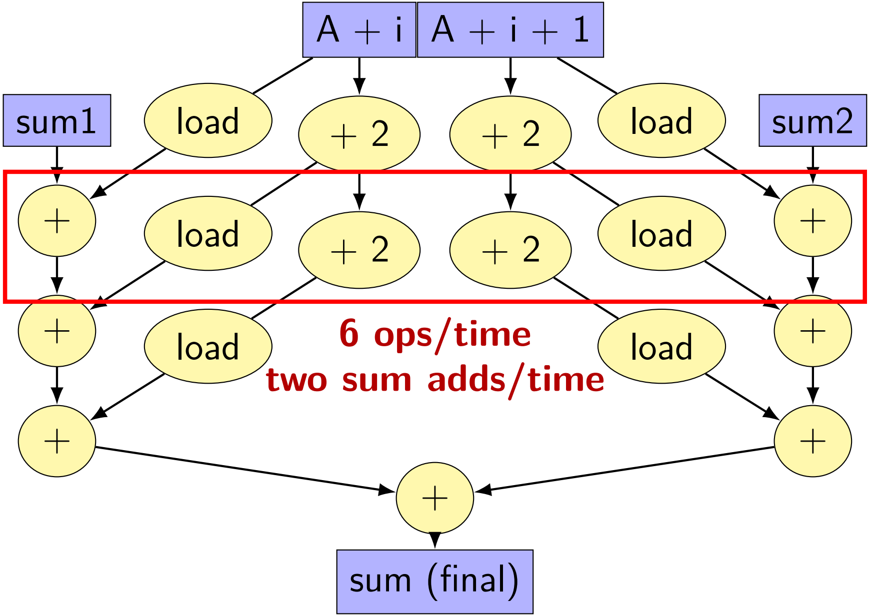


can only do sums one at a time

better data-flow



better data-flow



better data-flow