

bpred

branch prediction

guess where jumps, etc. go

very important for out-of-order processor performance

- takes many cycles before misprediction detected

- takes some time to undo misprediction

- lots of instructions per cycle “missed” on misprediction

modern laptop/desktop CPUs devote a lot of space to branch prediction

static branch prediction

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

intuition: loops:

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

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

exercise: static prediction

```
.global foo
foo:
    xor %eax, %eax // eax ← 0
foo_loop_top:
    test $0x1, %edi
    je foo_loop_bottom // if (edi & 1 == 0) goto 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
```

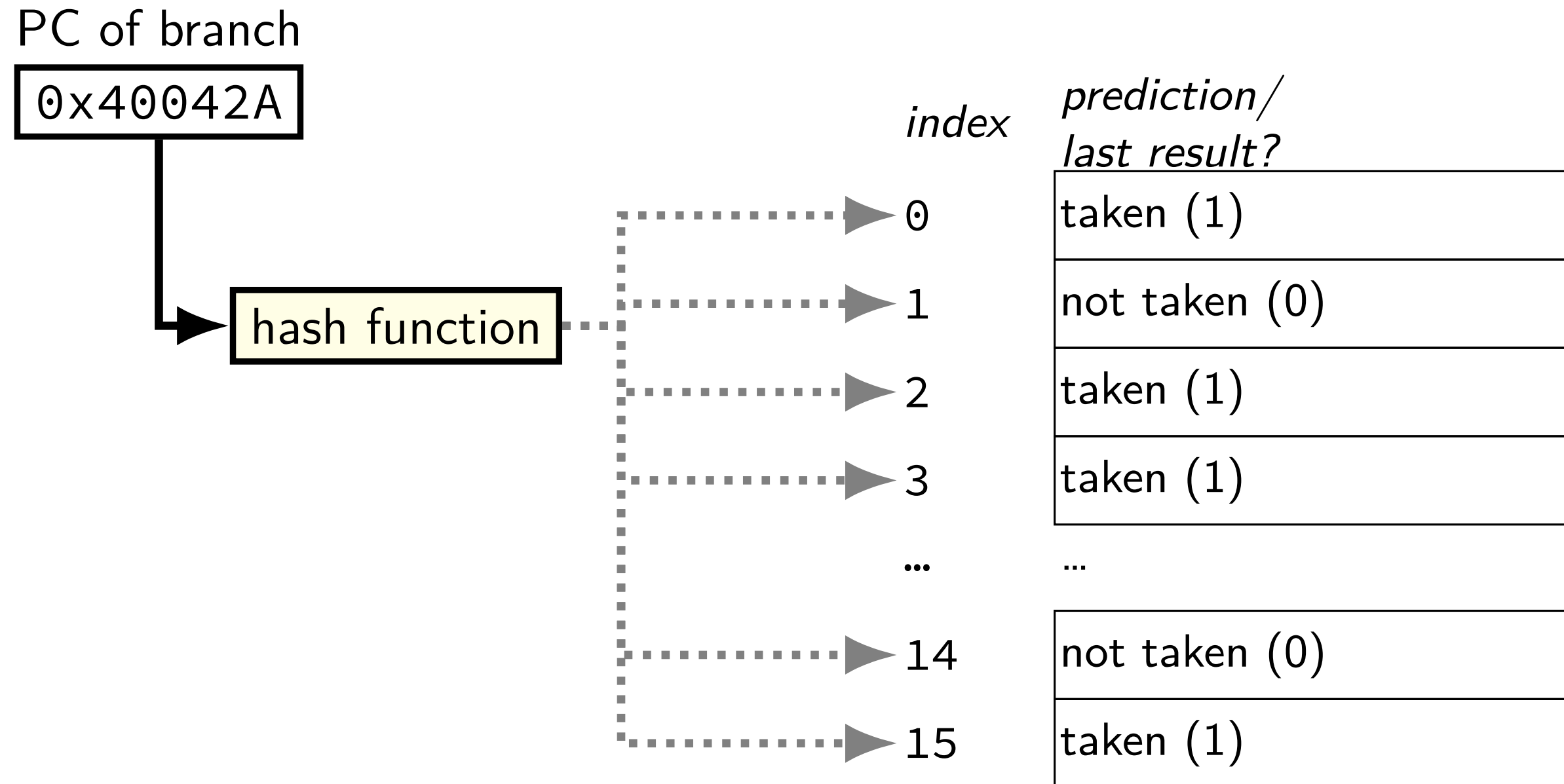
suppose %edi = 3 (initially)

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

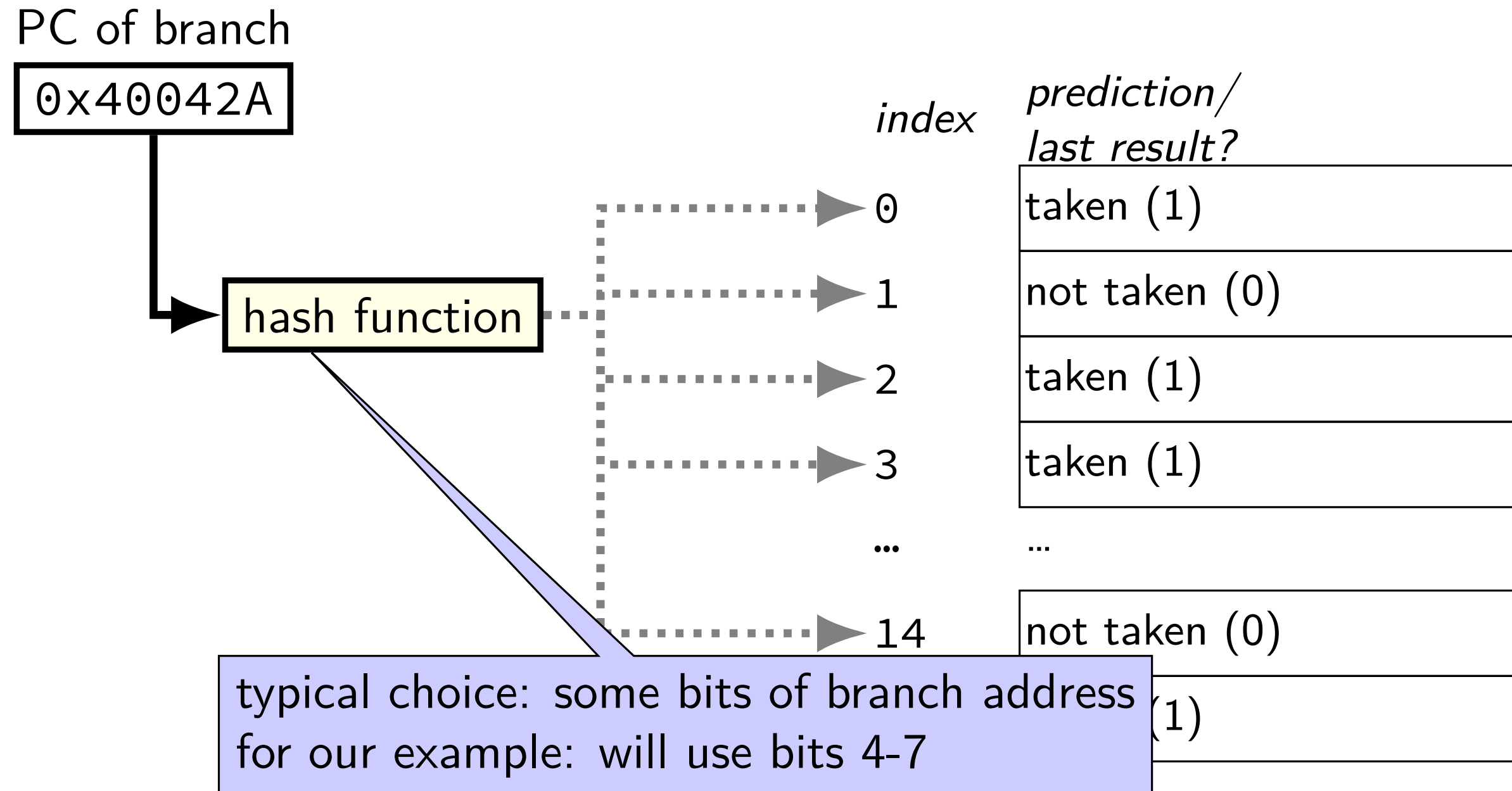
how many mispredictions for je? for jg?

predict: repeat last

predict: repeat last



predict: repeat last



predict: repeat last

PC of branch

0x40042A

hash function

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

predict: repeat last

PC of branch

0x40042A

hash function

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

prediction
to fetch stage

predict: repeat last

PC of branch

0x40042A

hash function

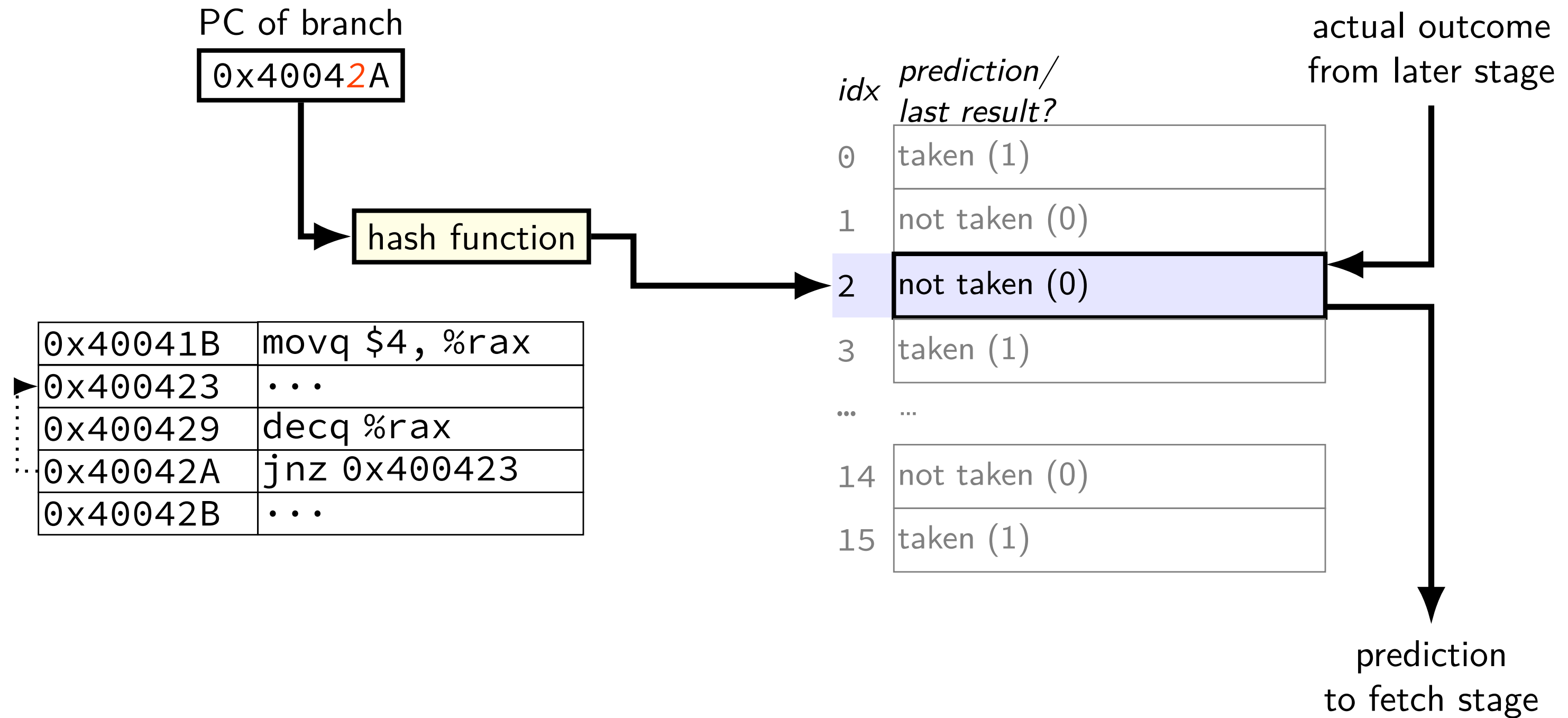
<i>index</i>	<i>prediction/ last result?</i>
0	taken (1)
1	not taken (0)
2	taken (1)
3	taken (1)
...	...
14	not taken (0)
15	taken (1)

actual outcome
(from later stage)

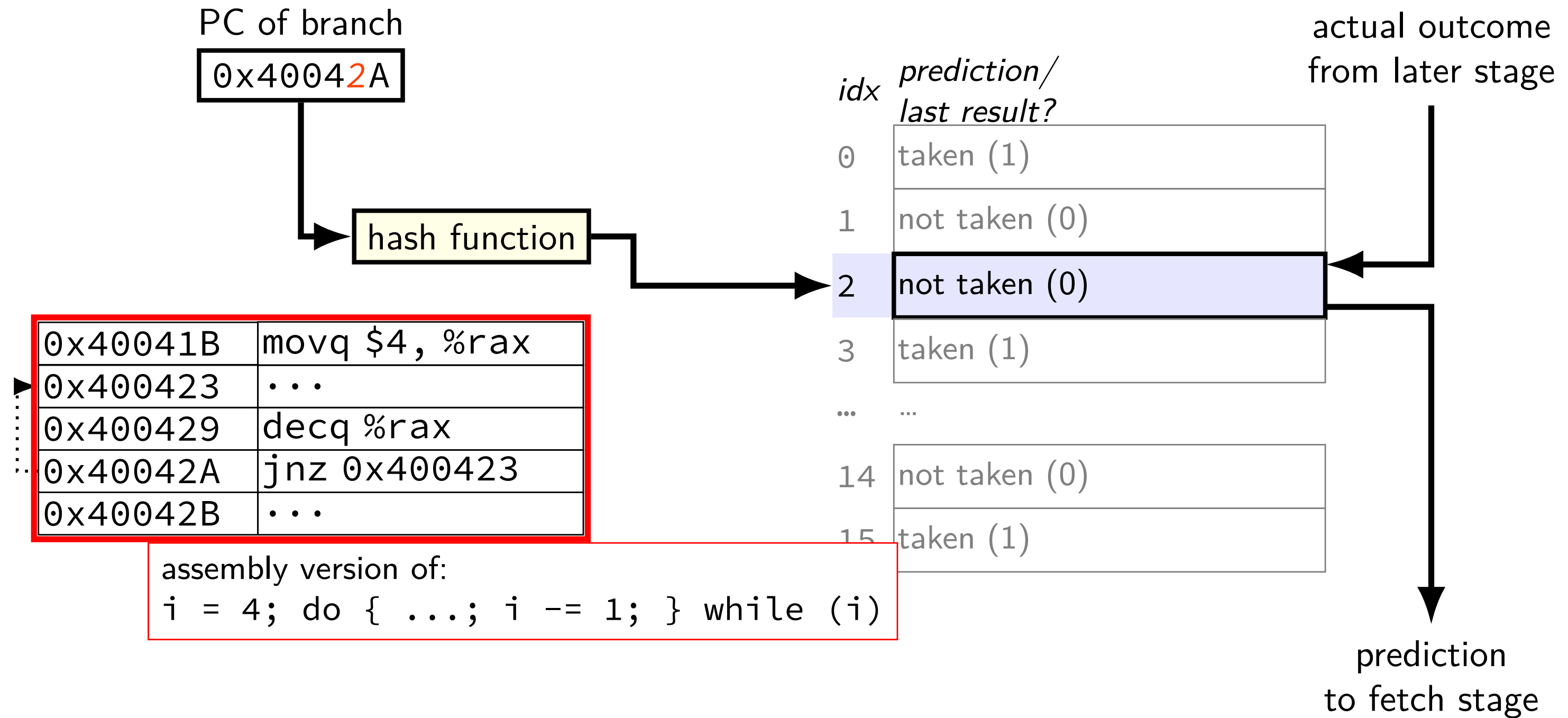
prediction
to fetch stage

example

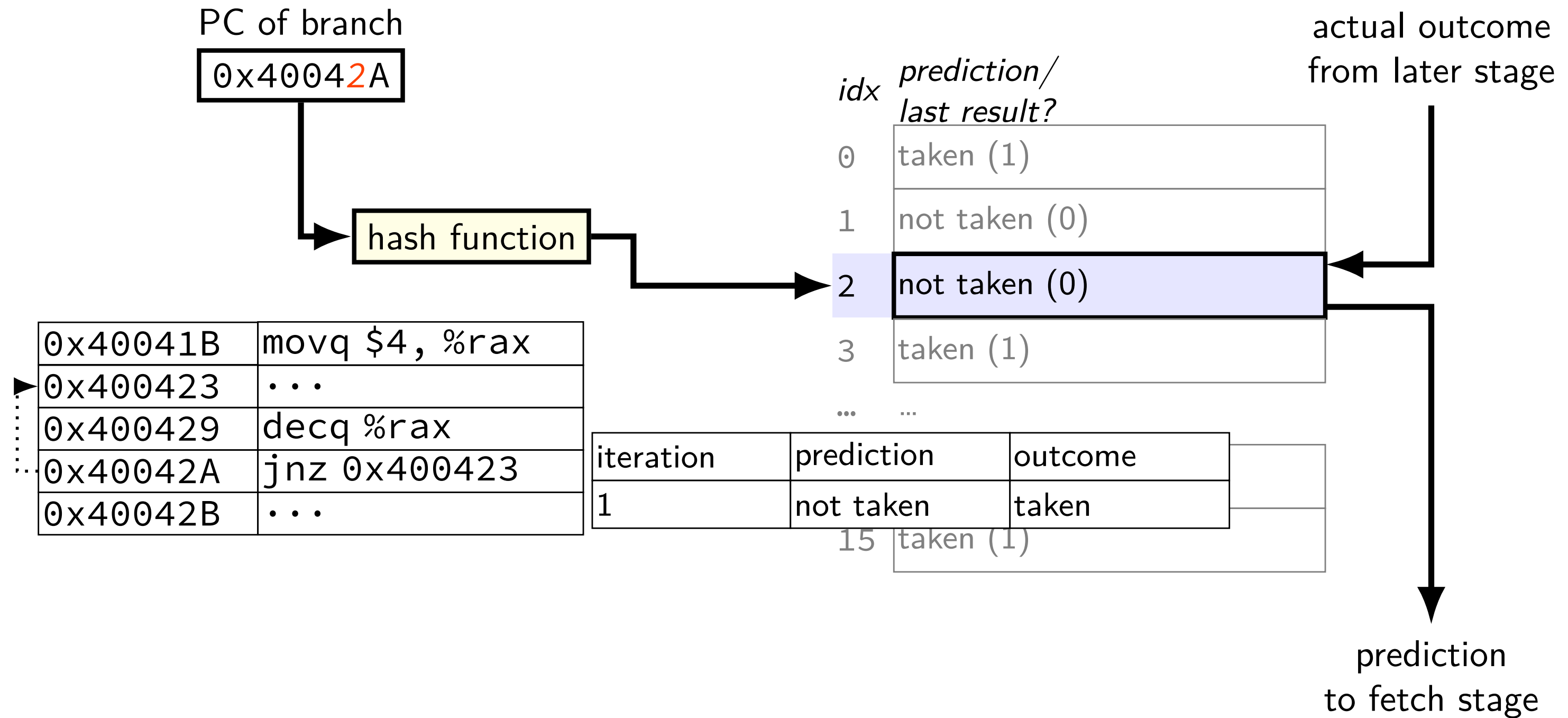
example



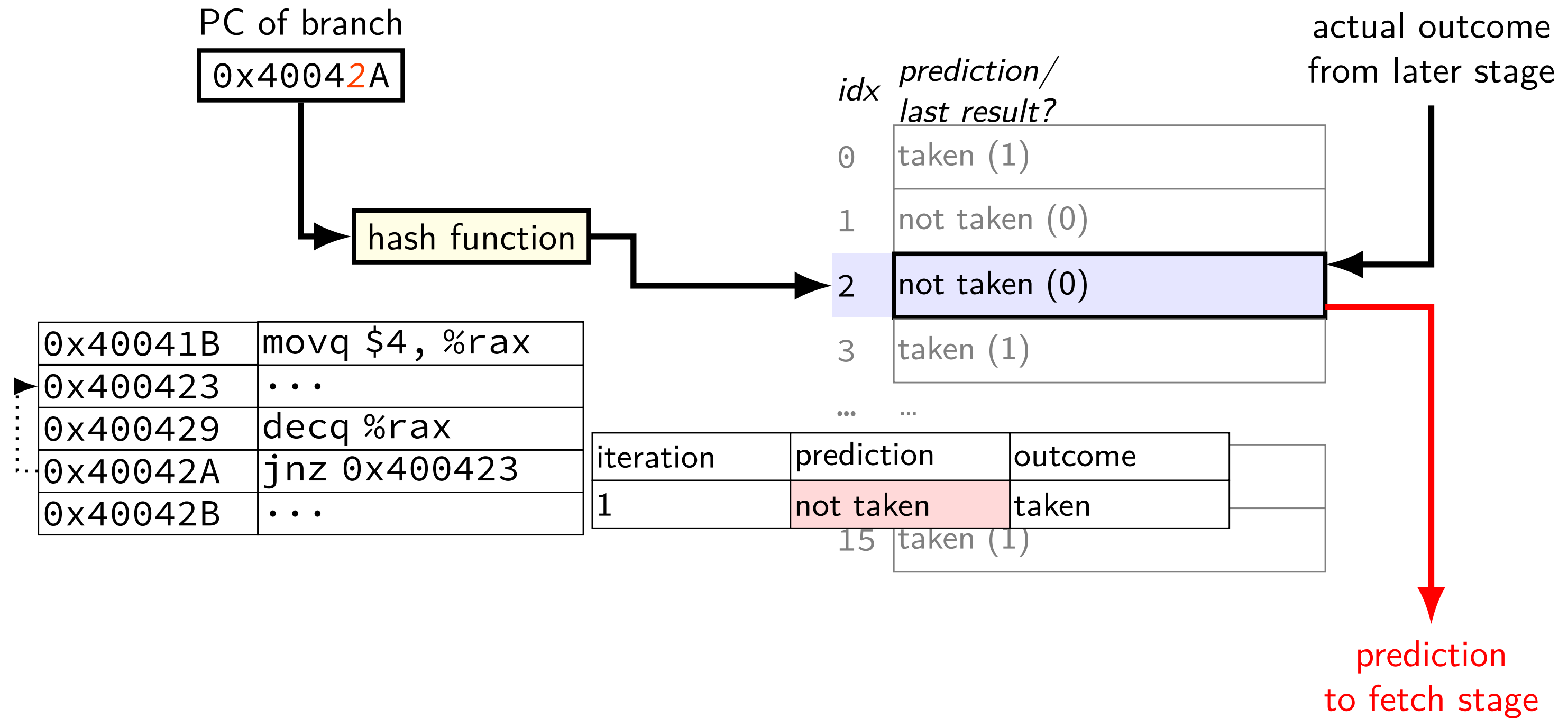
example



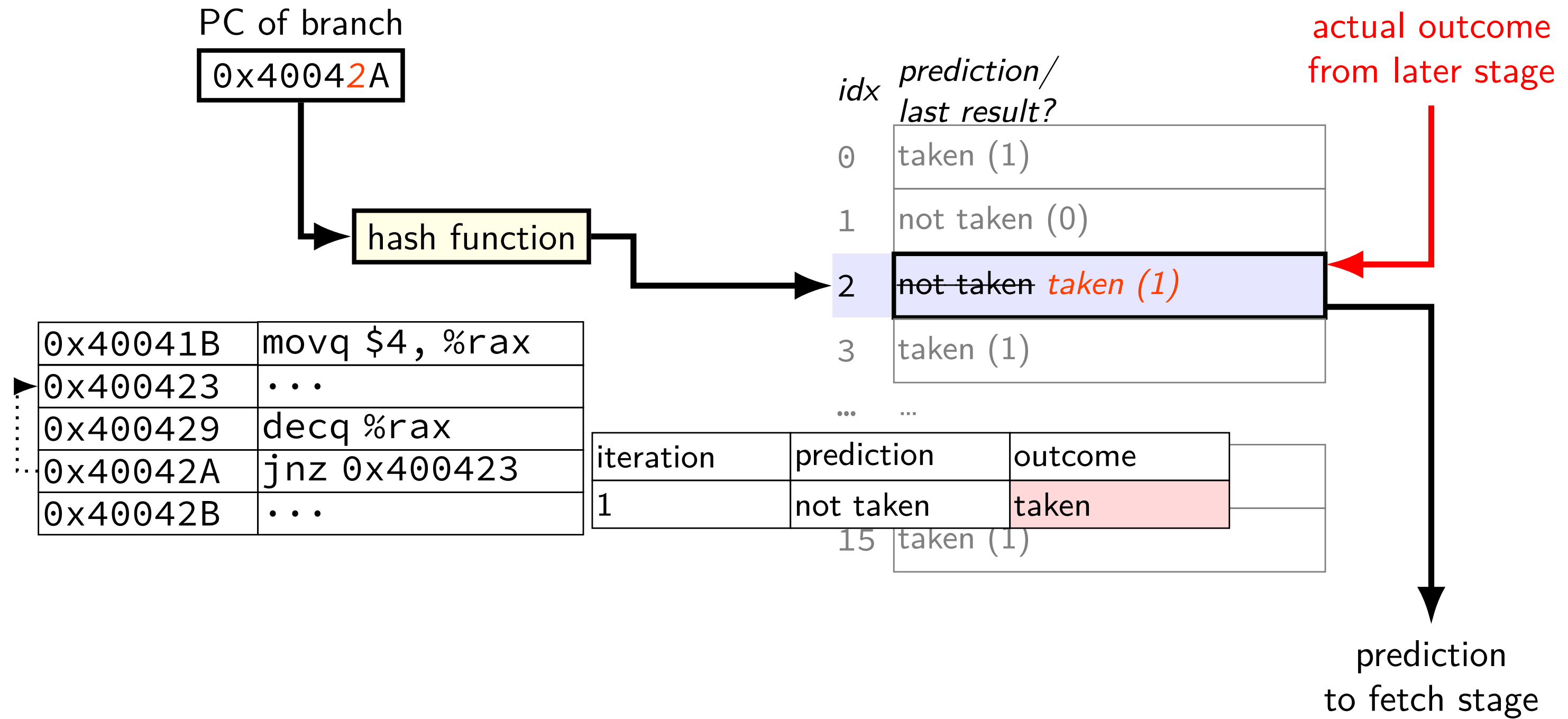
example



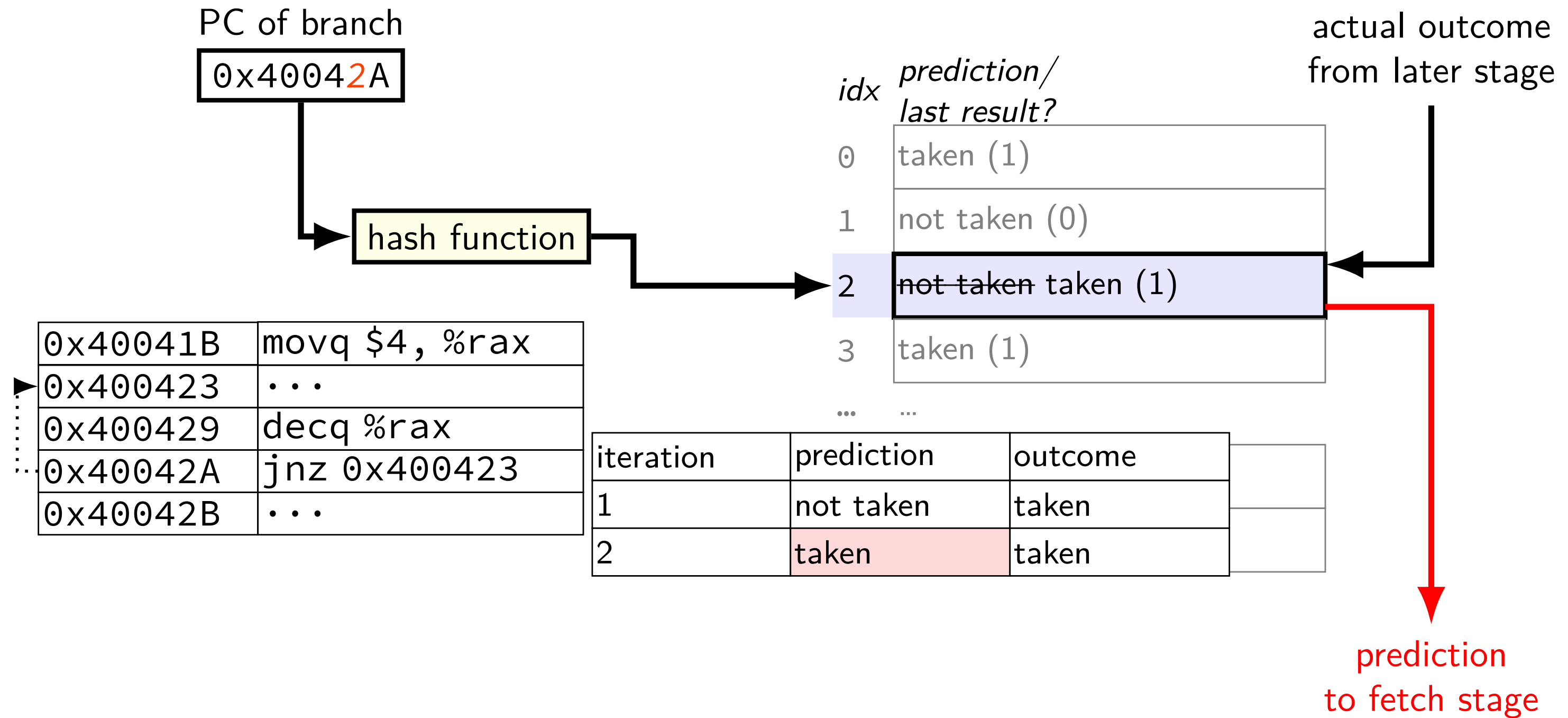
example



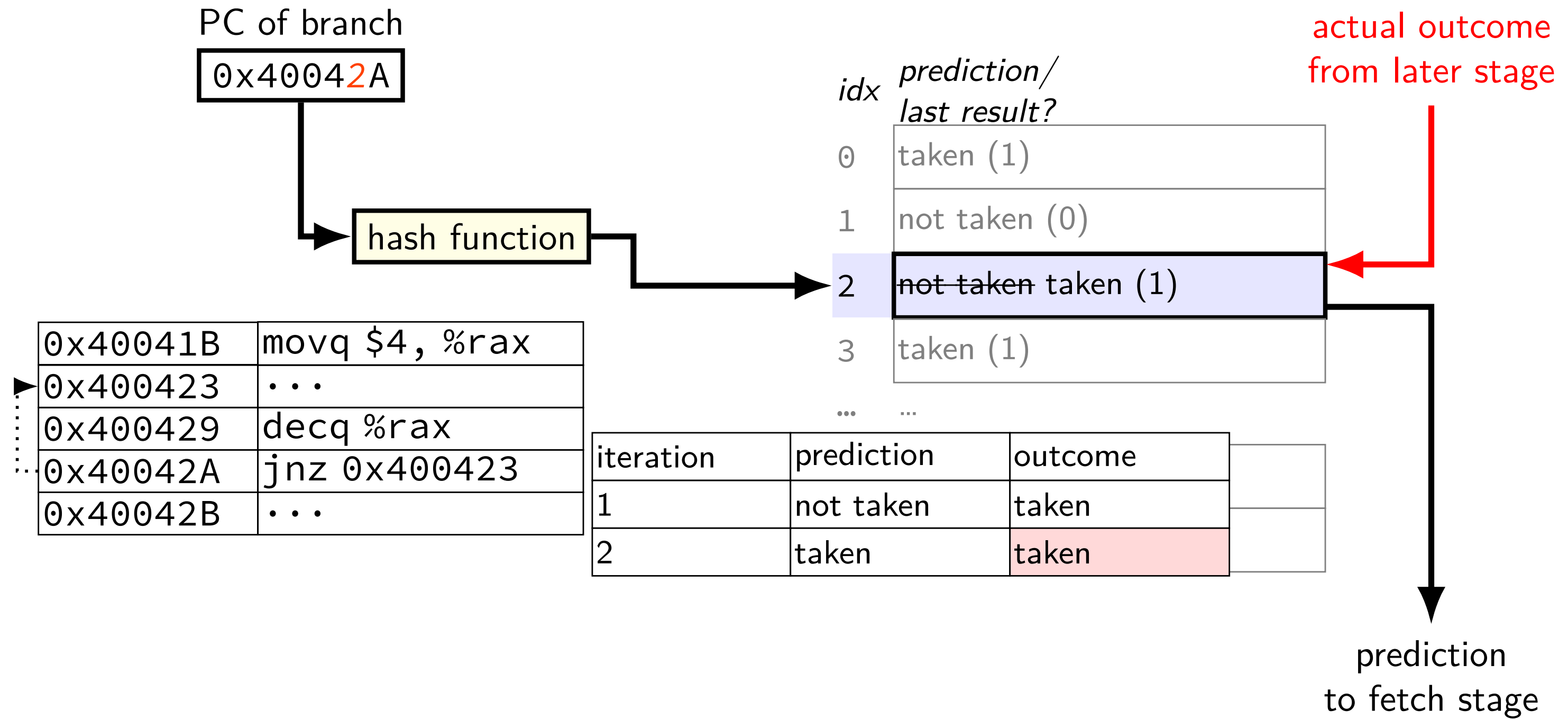
example



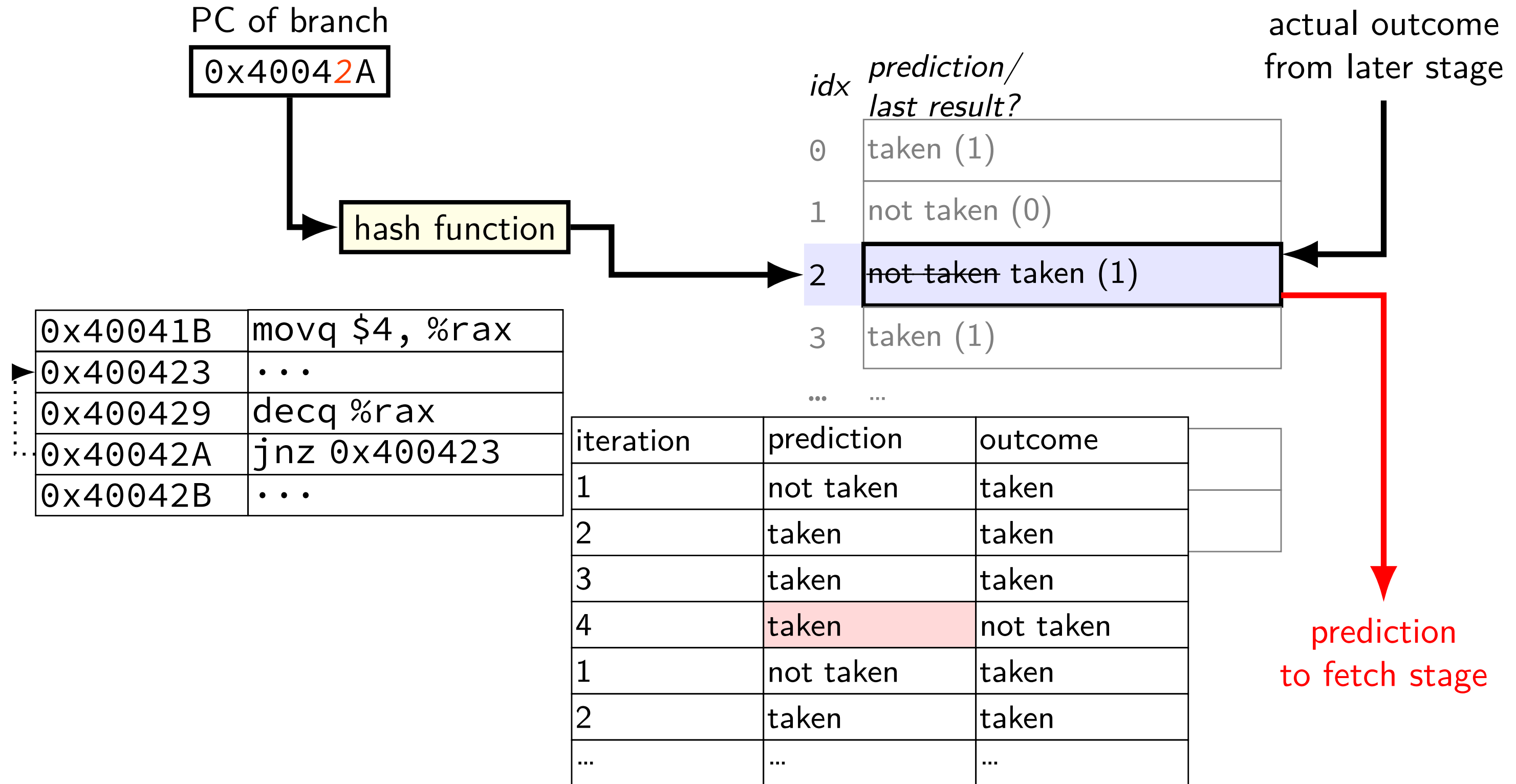
example



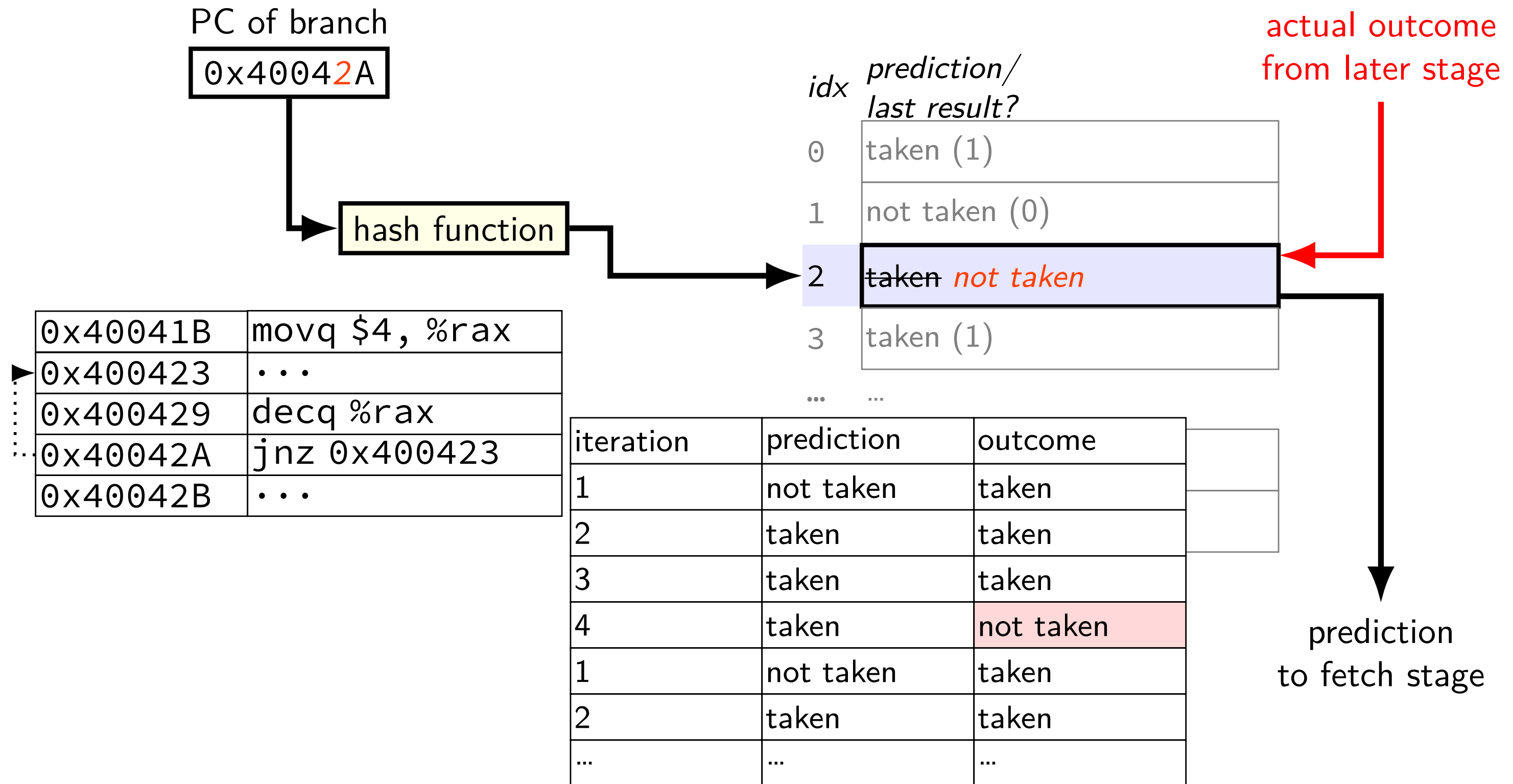
example



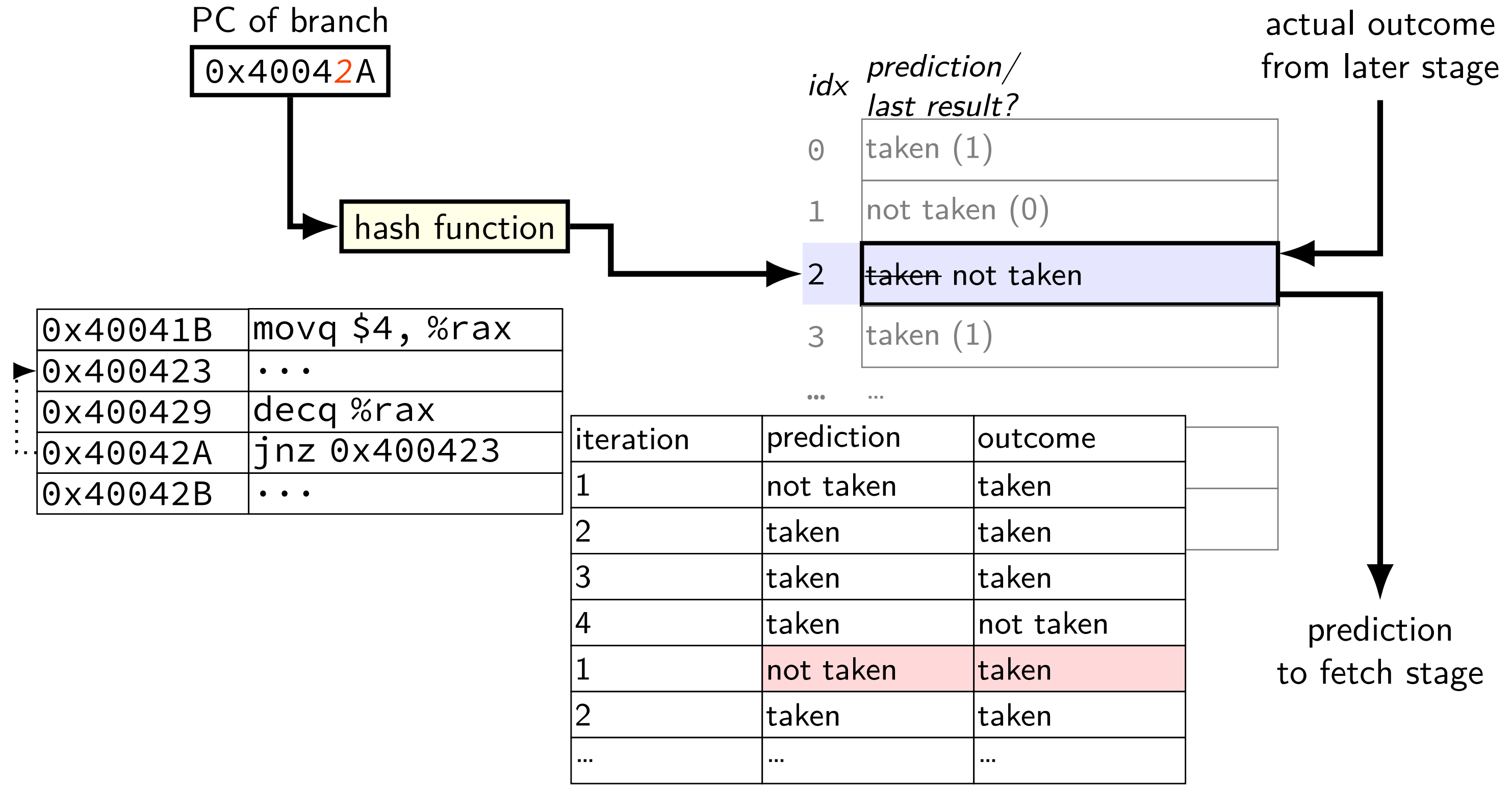
example



example



example



collisions?

two branches could have same hashed PC

nothing in table tells us about this

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

is it worth it?

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

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

collision results

possibility 1: both branches usually taken

no actual conflict — prediction is better(!)

possibility 2: both branches usually not taken

no actual conflict — prediction is better(!)

possibility 3: one branch taken, one not taken

performance probably worse

1-bit predictor and loops

loops have jump at beginning/end
static prediction got these 100%

1-bit prediction: predicts first and last iteration wrong
first wrong: last time loop ran, did not continue, but should
last wrong: last time loop ran, did continue, but should not
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 $i \% 3 == 0$? for $i == 50$? overall?

```
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 $i \% 3 == 0$? for $i == 50$? overall?

```
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 soln (1)

exercise soln (1)

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

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

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

exercise soln (1)

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

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

```

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

exercise soln (1)

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

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

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

exercise soln (1)

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

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

```

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

exercise soln (1)

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

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

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

beyond 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

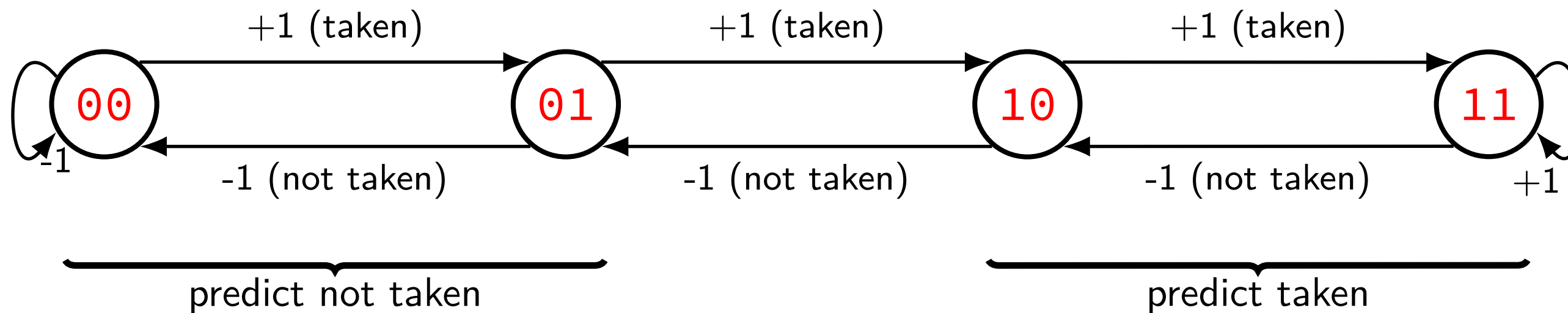
- 1% of taken branches right after branch not taken

new predictor: wrong about 1% of the time

- 1% when branch not taken

2-bit saturating counter

2-bit saturating counter



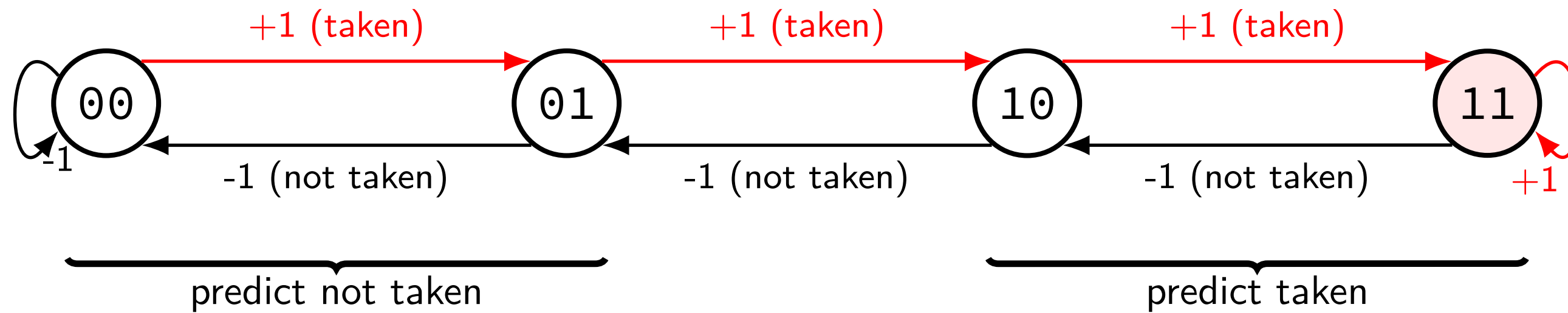
PC of branch

0x40042A

hash function

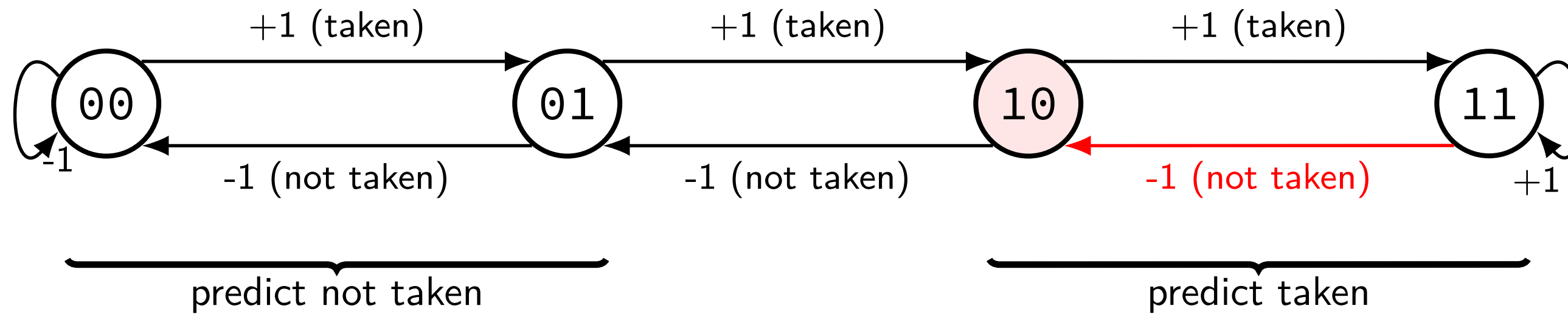
<i>index</i>	<i>counter</i>
0	11
1	01
2	11
...	...
14	10
15	00

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 ↔ 001 ↔ 010 ↔ 011 ↔ 100 ↔ 101 ↔ 110 ↔ 111

000-011: not taken

100-111: taken

exercise

use 2-bit predictor on this loop

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

what is the conditional branch misprediction rate?

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

exercise soln (1)

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

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

mod 3: ends up always predicting not taken

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

(49/50)

break: ends up always predicting not taken

overall: 82/100

```

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

```

briefly: other prediction ideas

branches often have cyclic behavior

example: `for (int i = 0; i < 5; ++i)`

example: `if (first or last item) {...}`

prediction idea: try to find/count fixed loops

prediction idea: identify patterns like TTNNNTTNNNT→T

branches are correlated

example: flag checked in multiple places

example: special case for first iteration of loop

prediction idea: identify patterns across multiple branches

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

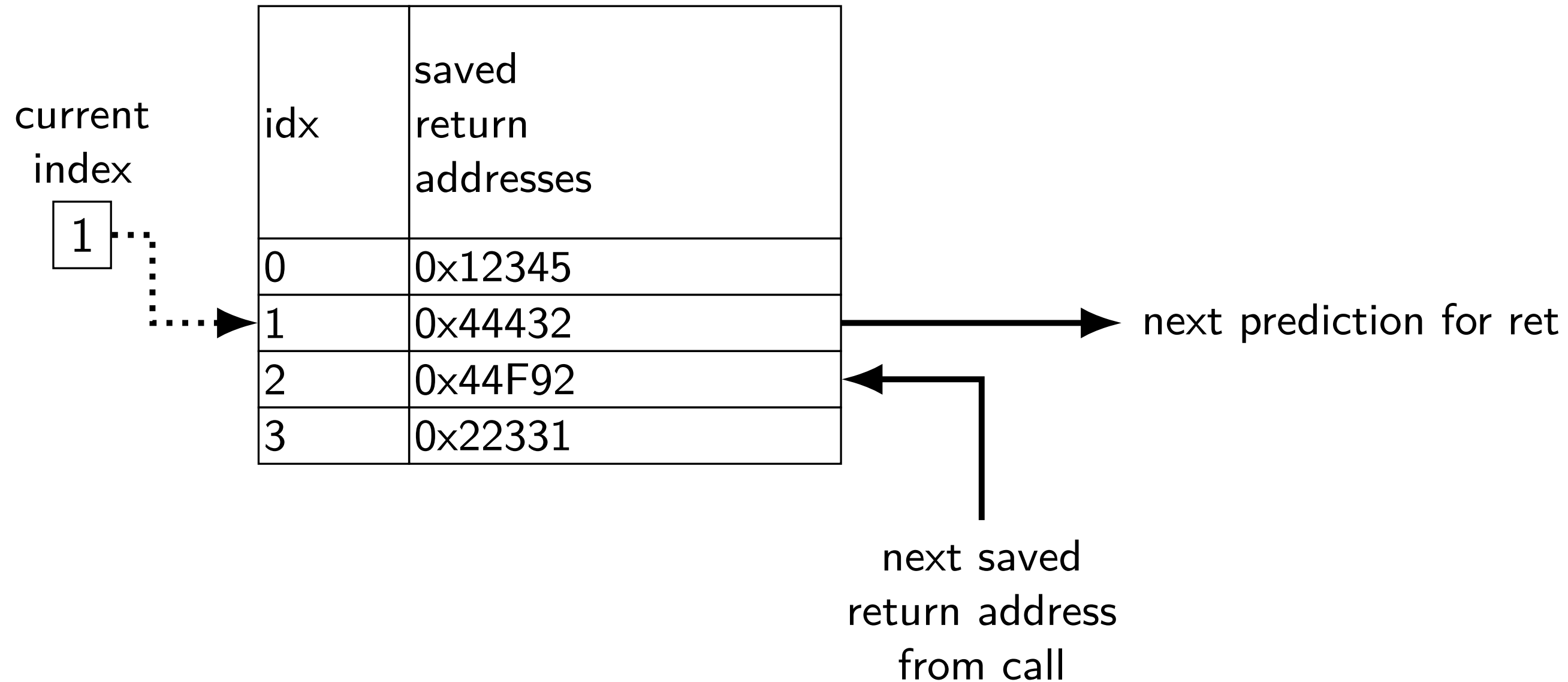
stack in memory

baz return address
bar return address
foo return address

(partial?) stack
in CPU registers

4-entry return address stack

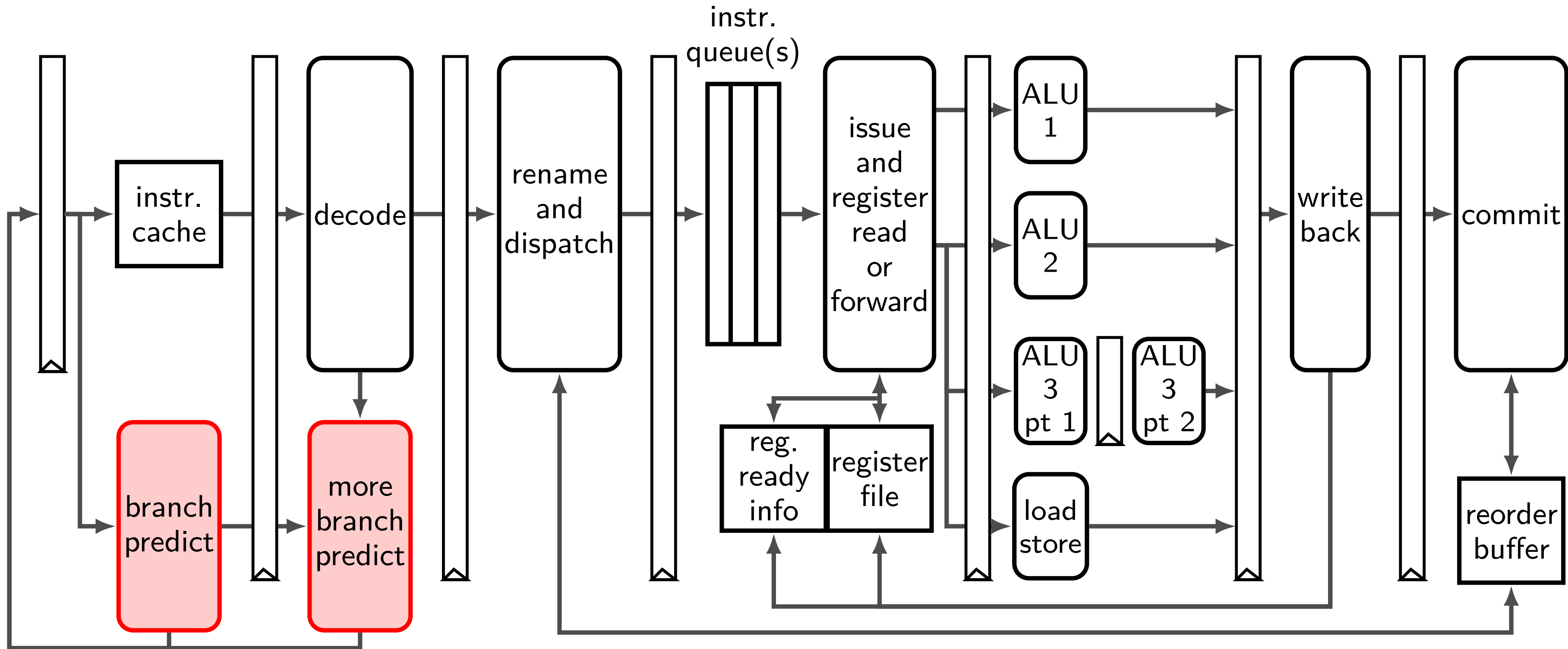
4-entry return address stack in CPU



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

on ret: read prediction from index, decrement index

branch prediction before fetch/decode



branch target buffer

suppose we have OOO processor that fetches 4 instructions/cycle

given `jmp LABEL` is 1st instruction we fetch

want to fetch code at LABEL *ideally in same cycle* to avoid slowdown

if not same cycle, in next cycle

likely we won't decode LABEL for several cycles

if it's `jle LABEL` instead...

also want to predict if it will go to LABEL just as fast

may not figure out its a `jle` for several cycles

solution: special predictor for what instruction will be

BTB: cache for branch targets

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	0x3FFFFFF3	...
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	...

```

0x3FFFFFF3:  movq %rax, %rsi
0x3FFFFFF7:  pushq %rbx
0x3FFFFFF8:  call 0x404033
0x4000001:   popq %rbx
0x4000003:   cmpq %rbx, %rax
0x4000005:   jle 0x3FFFFFF3
...
0x4000031:   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)` or `call *%rax` or ...

example: implementing switch statement

example: implementing polymorphic method call

want to predict *target address*

BTB could store one possibility

really want to take advantage of context

example: guess whether we'll call `Rectangle.GetArea` or `Circle.GetArea`

prediction idea: instead of tables containing taken/not taken...

... have table containing predicted target

one implementation: hashtable keyed by recent branches taken

branch patterns

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

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

(T = taken, N = not taken)

goal: take advantage of recent pattern to make predictions

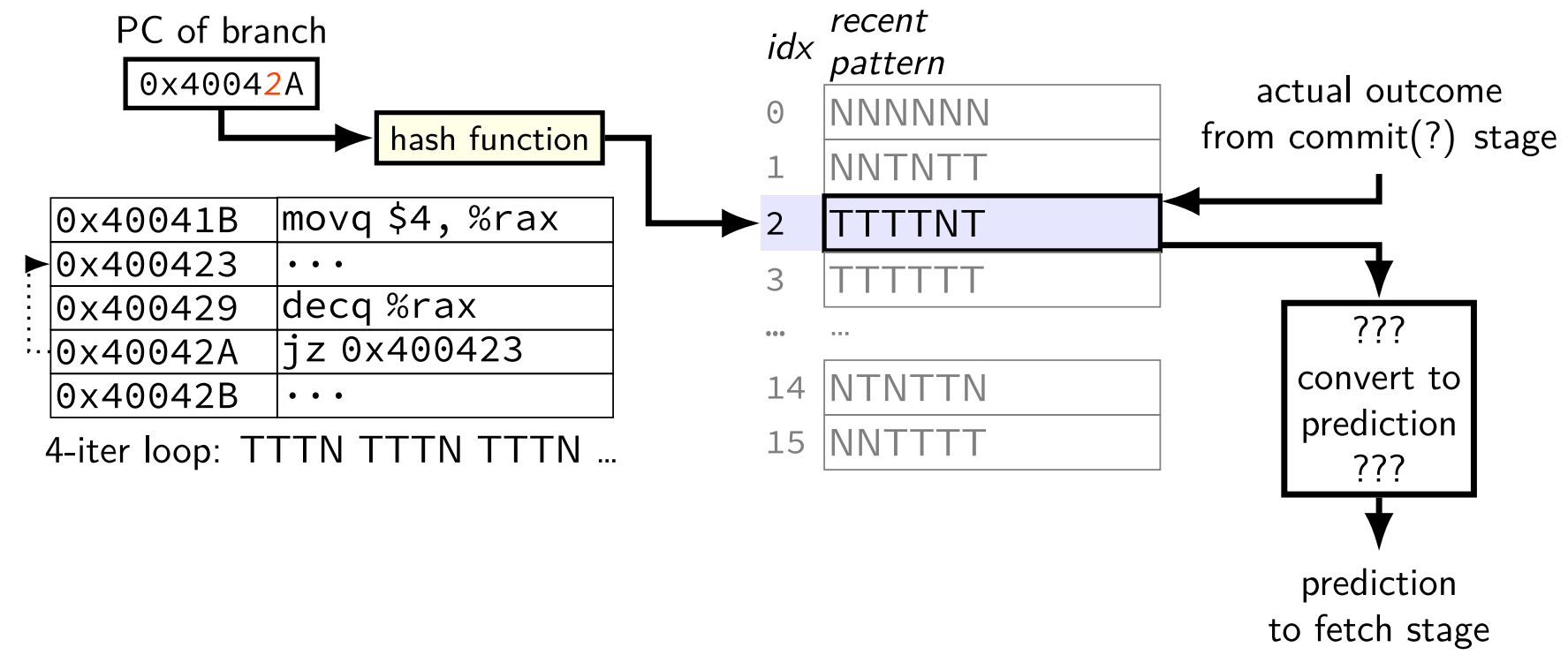
just saw 'NTTTNT'? predict T next

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

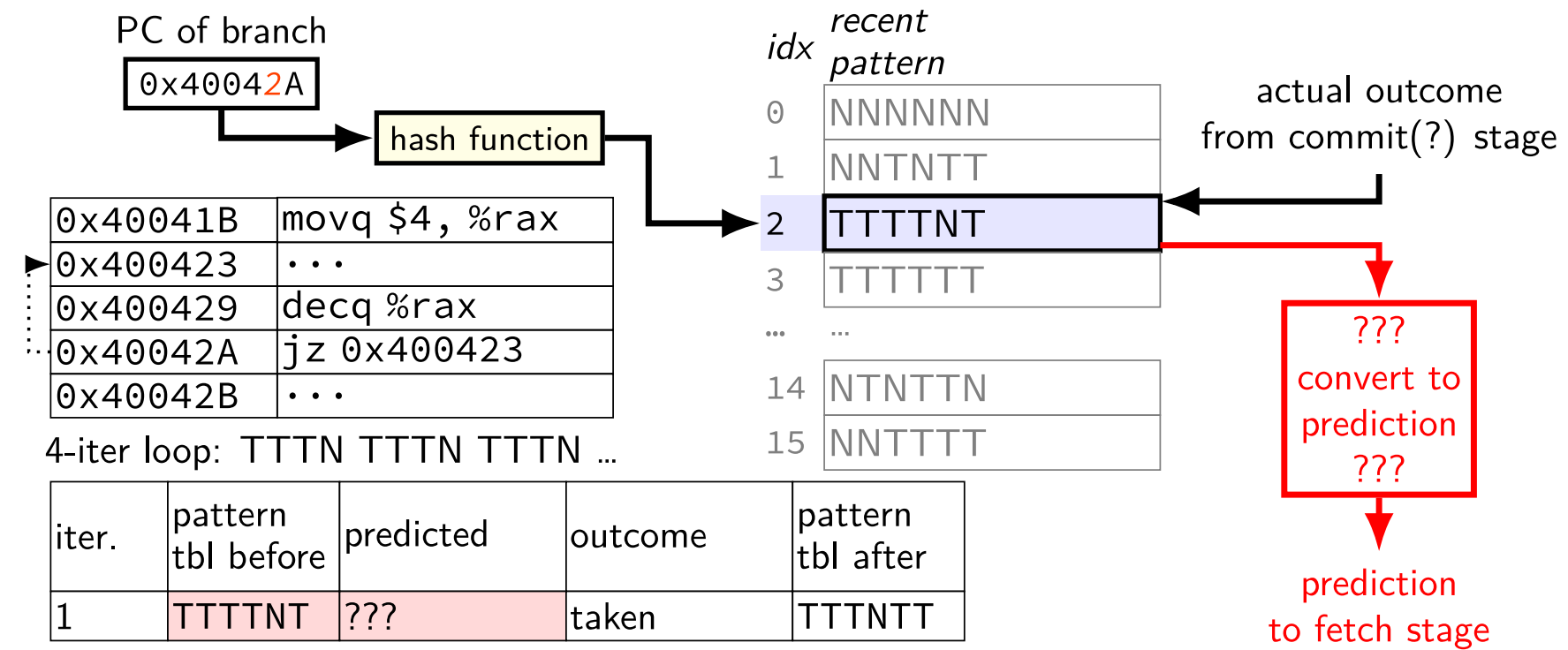
...

local pattern predictor (incomplete)

local pattern predictor (incomplete)

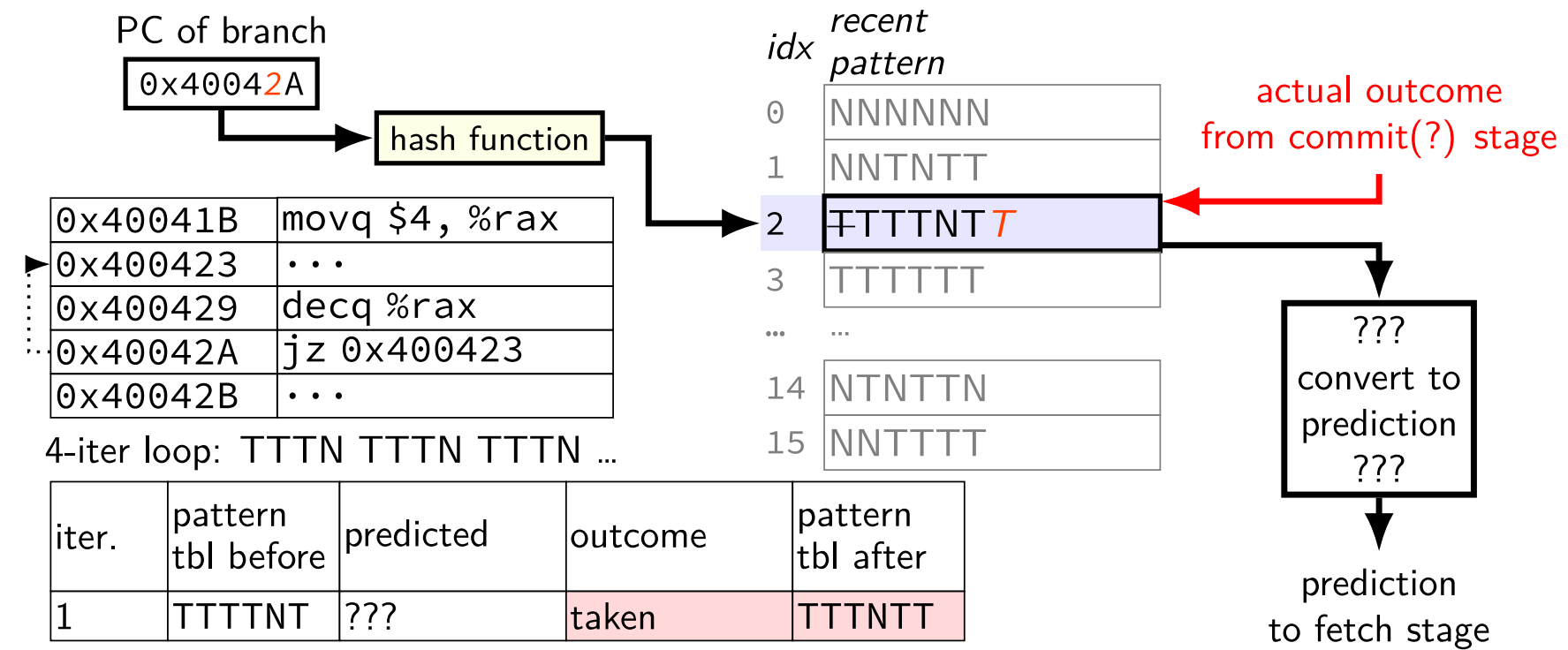


local pattern predictor (incomplete)



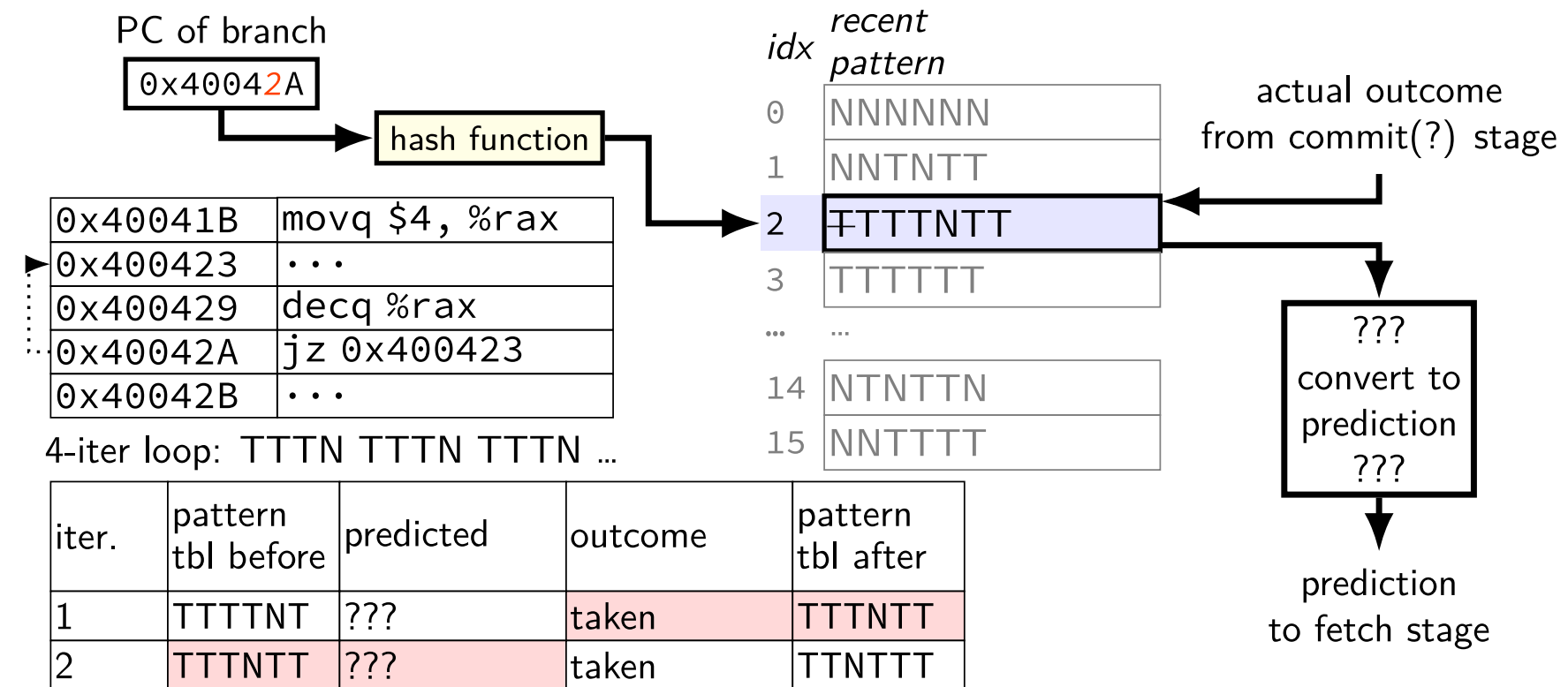
...
-----	-----	-----	-----	-----

local pattern predictor (incomplete)



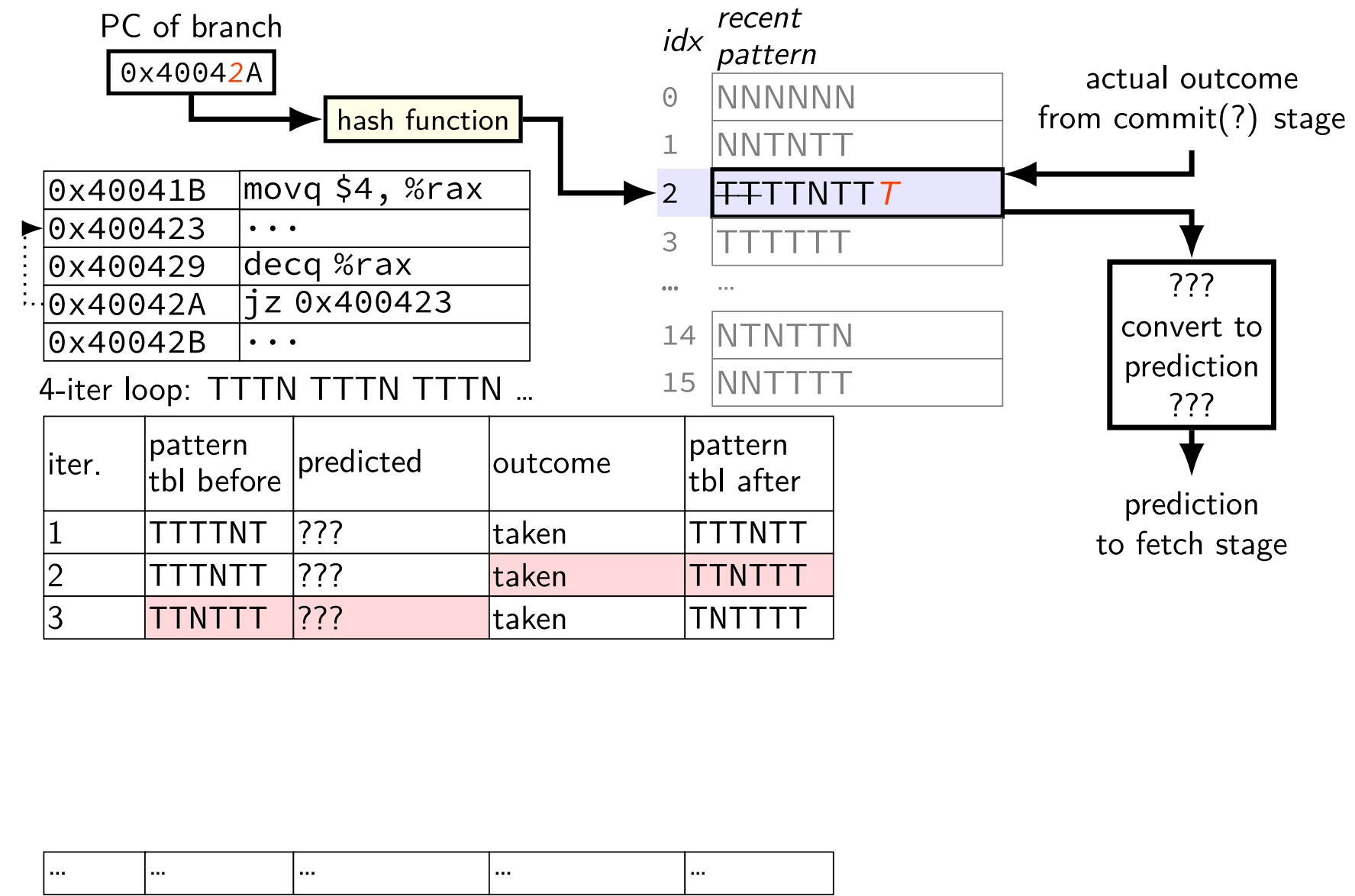
...
-----	-----	-----	-----	-----

local pattern predictor (incomplete)

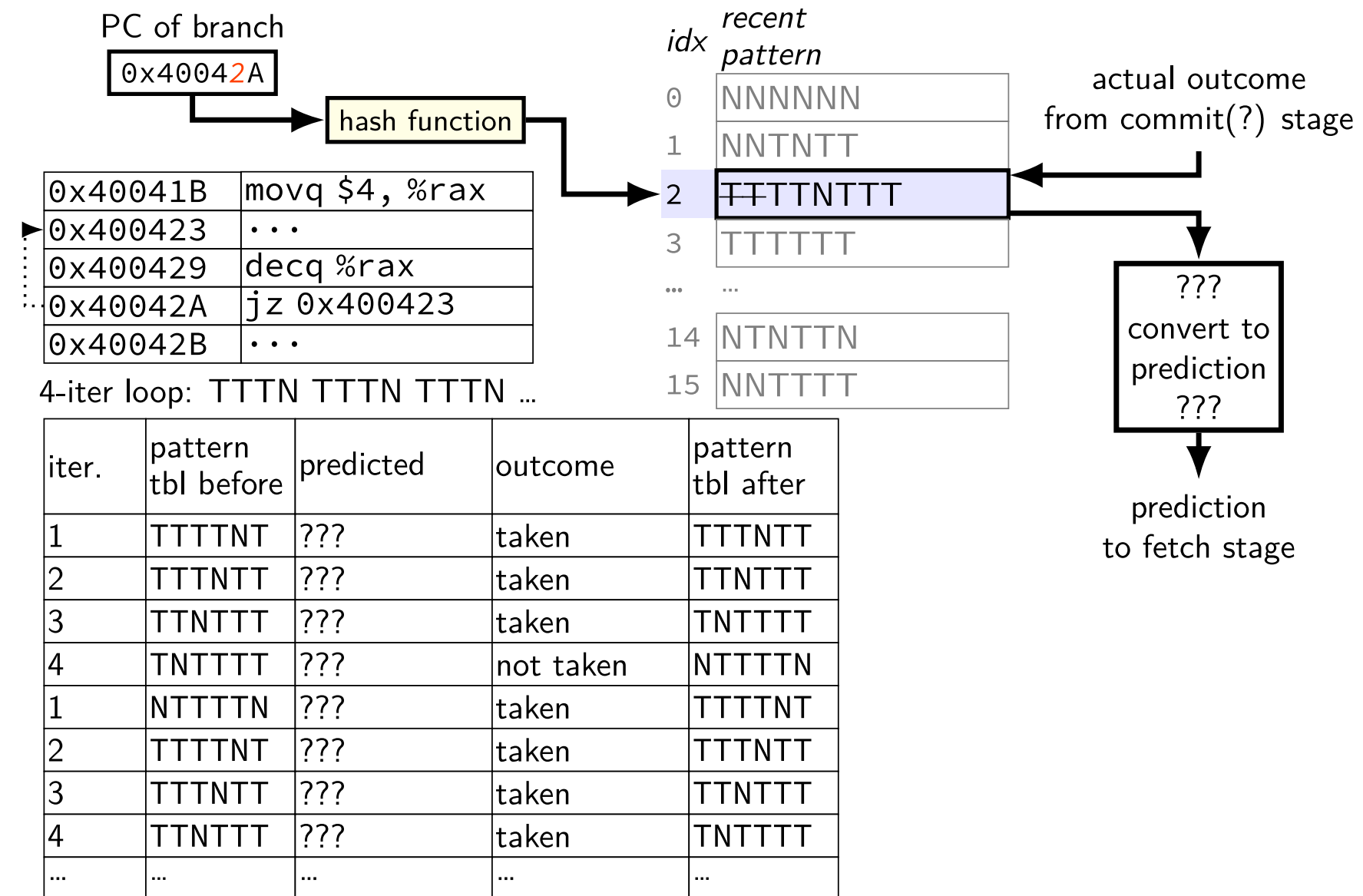


...
-----	-----	-----	-----	-----

local pattern predictor (incomplete)



local pattern predictor (incomplete)



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 (NTTTTTTTNTTTTTTTNTTTTTTT...)

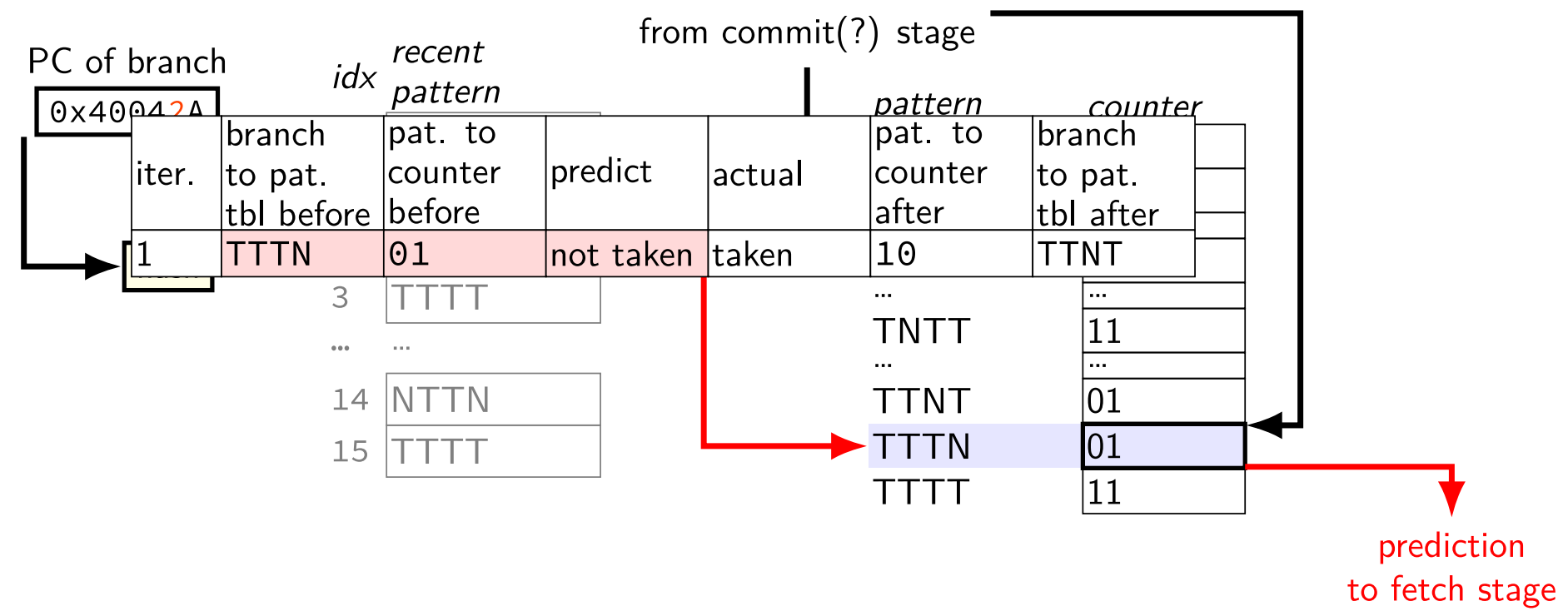
predict T? if statement mostly taken (TTTTNTTNTTTTTTTTTTTNTTTT...)

predict N? loop with 5 iterations (NTTTTNTTTTNTTTTNTTTTNTT...)

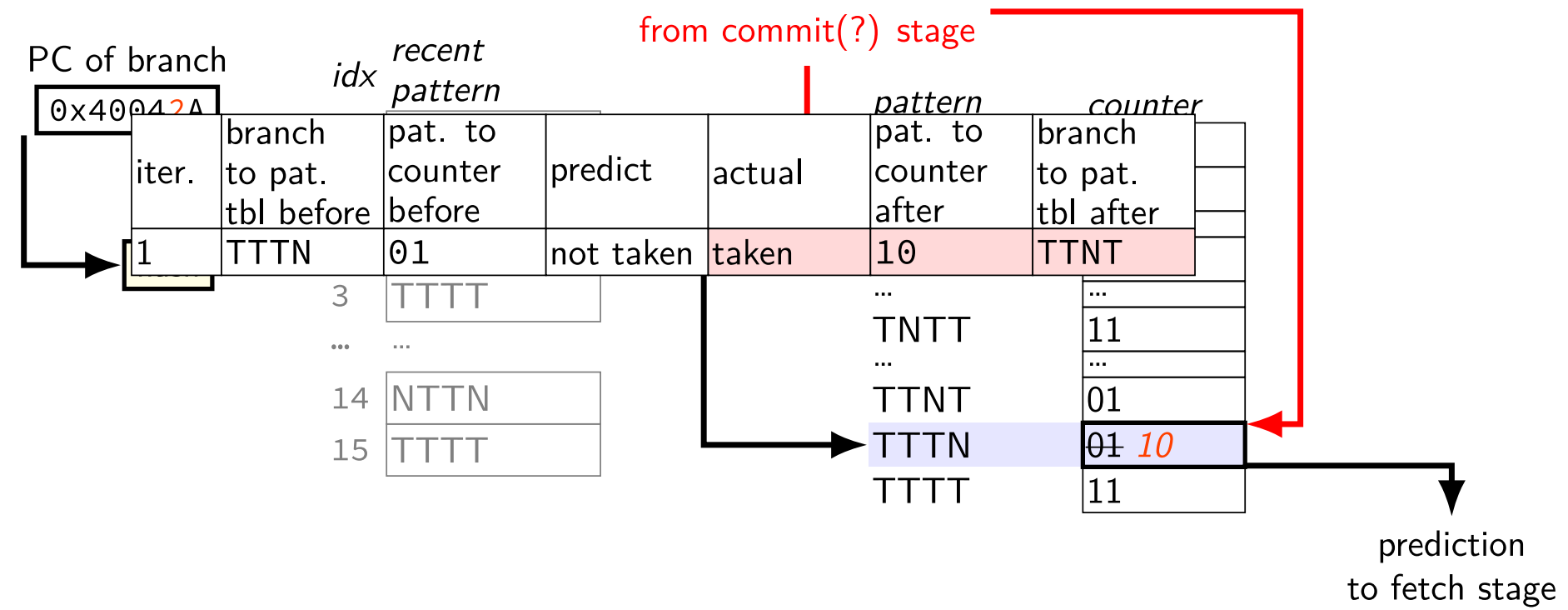
(many more)

history of history

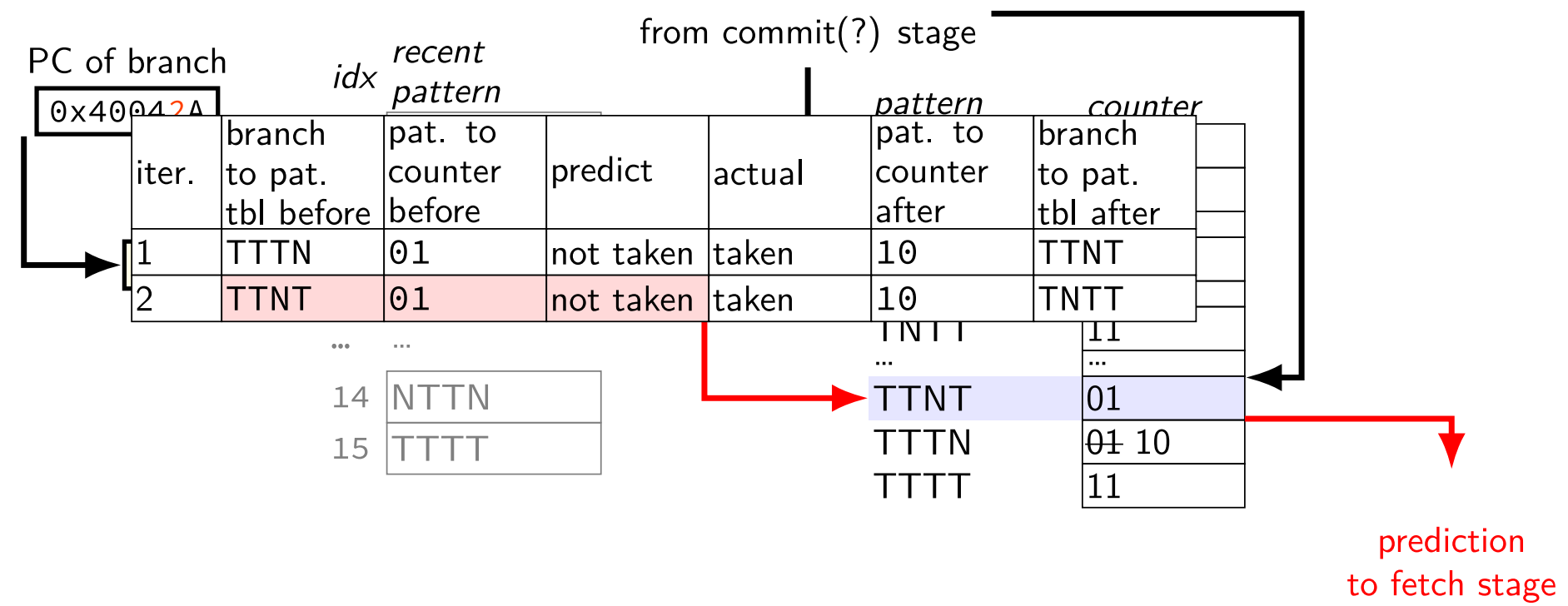
history of history



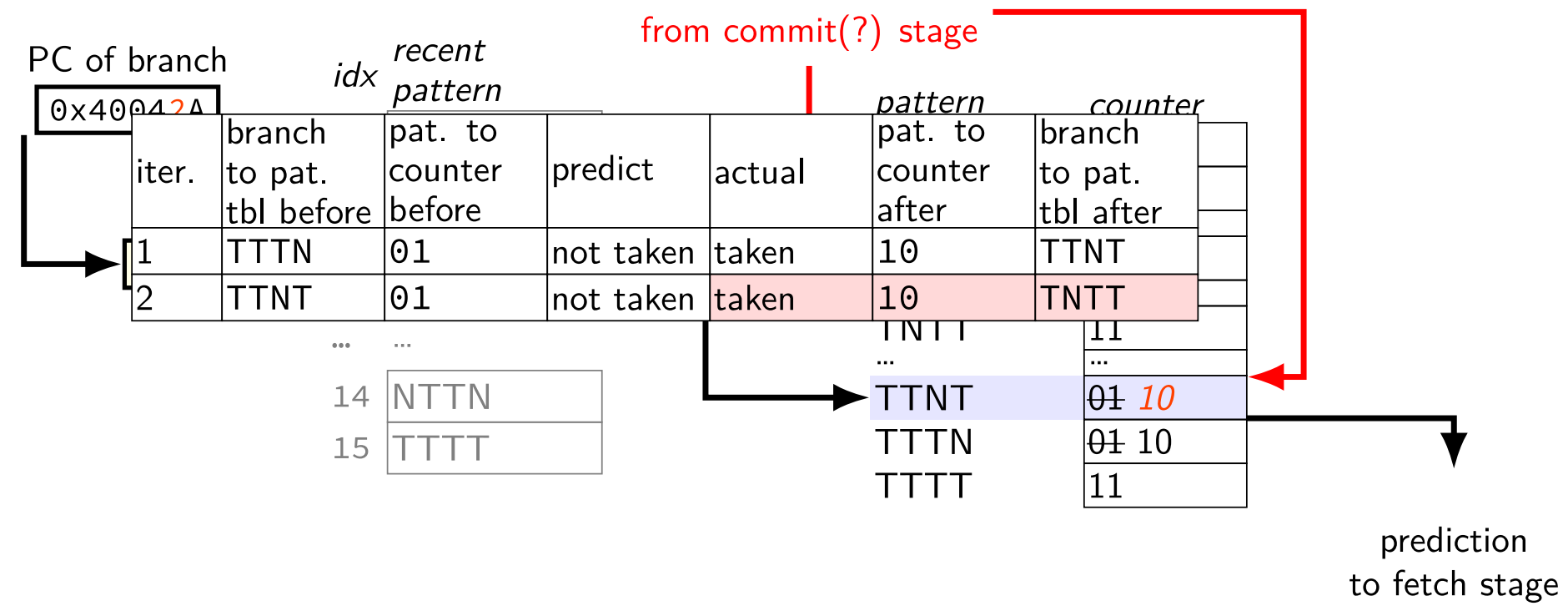
history of history



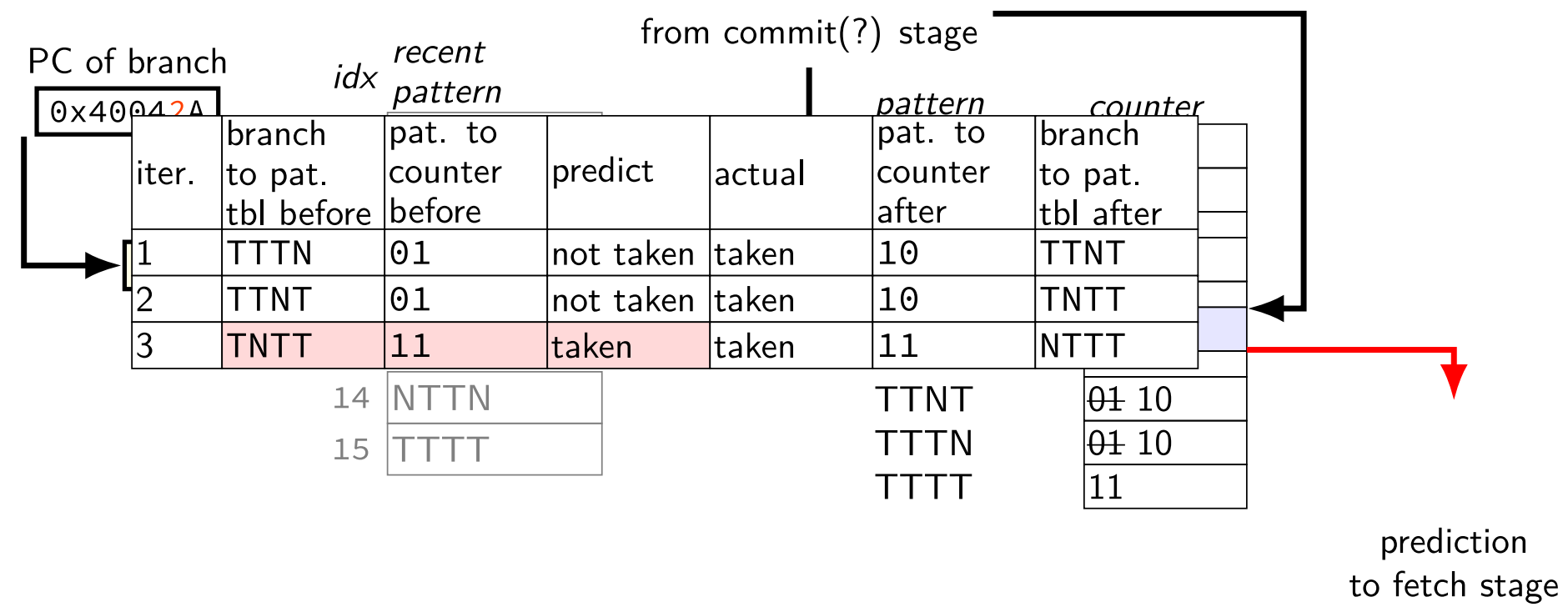
history of history



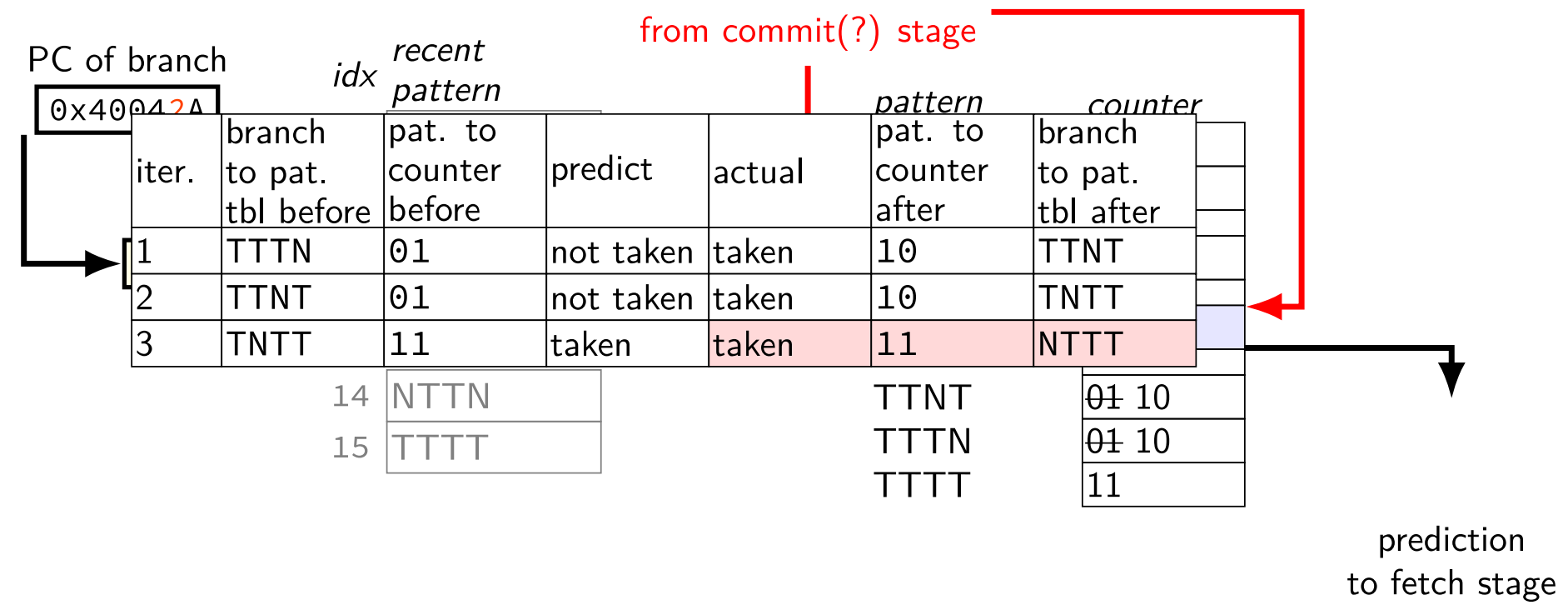
history of history



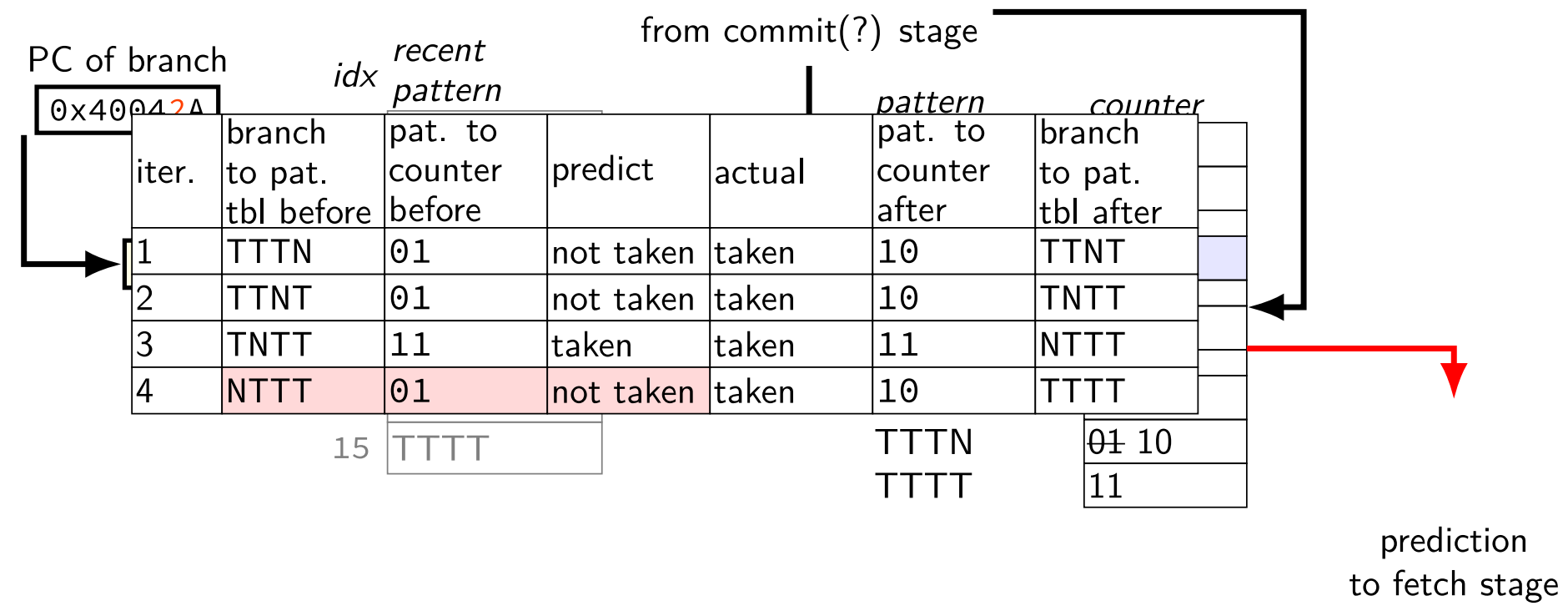
history of history



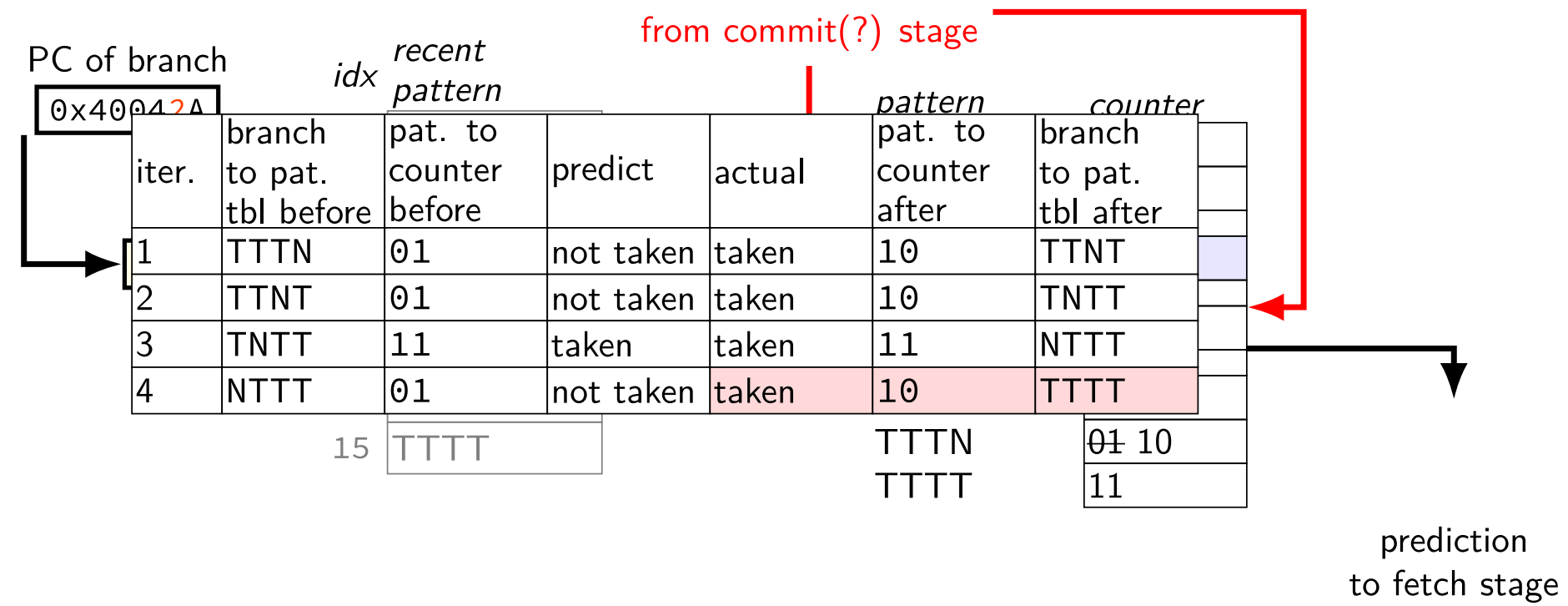
history of history



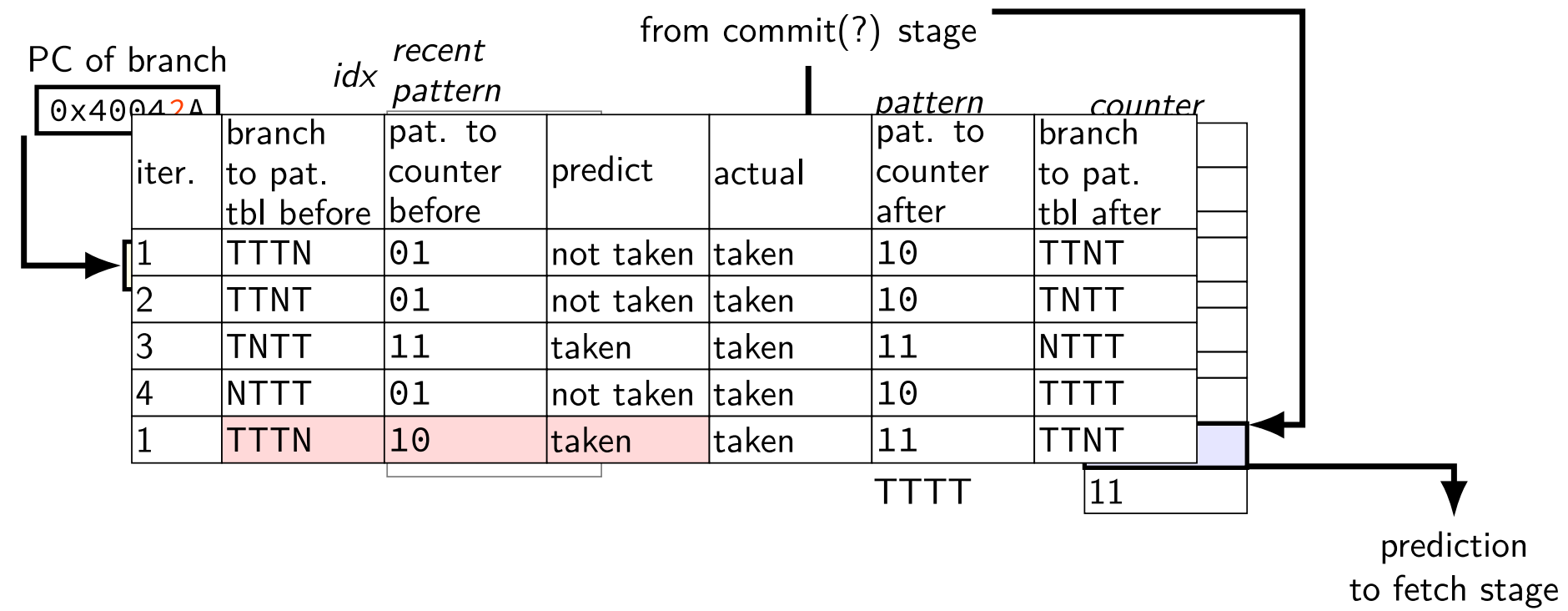
history of history



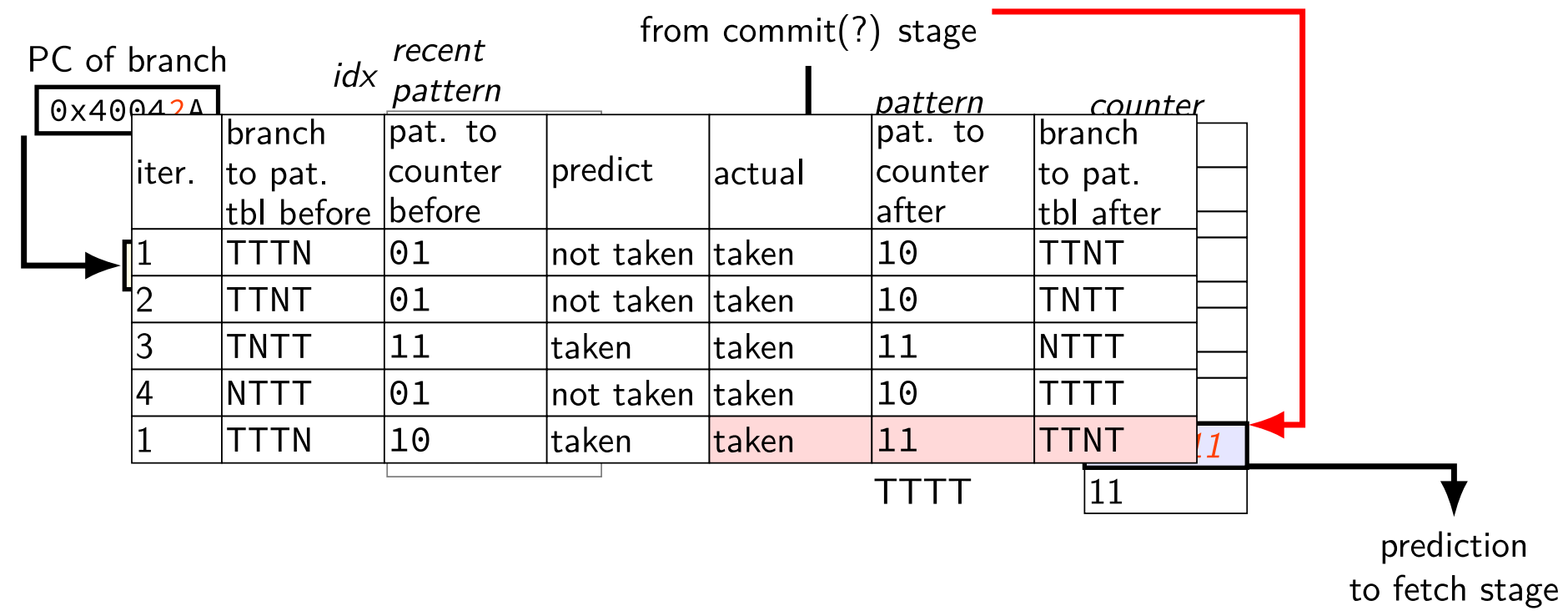
history of history



history of history



history of history



history of history

PC of branch		idx	recent	from commit(?) stage			branch to pat. tbl after	counter
0x40042A			pattern	pat. to counter before	predict	actual		
iter.	branch to pat. tbl before							
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	1
						TTTT	11	

prediction
to fetch stage

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?

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

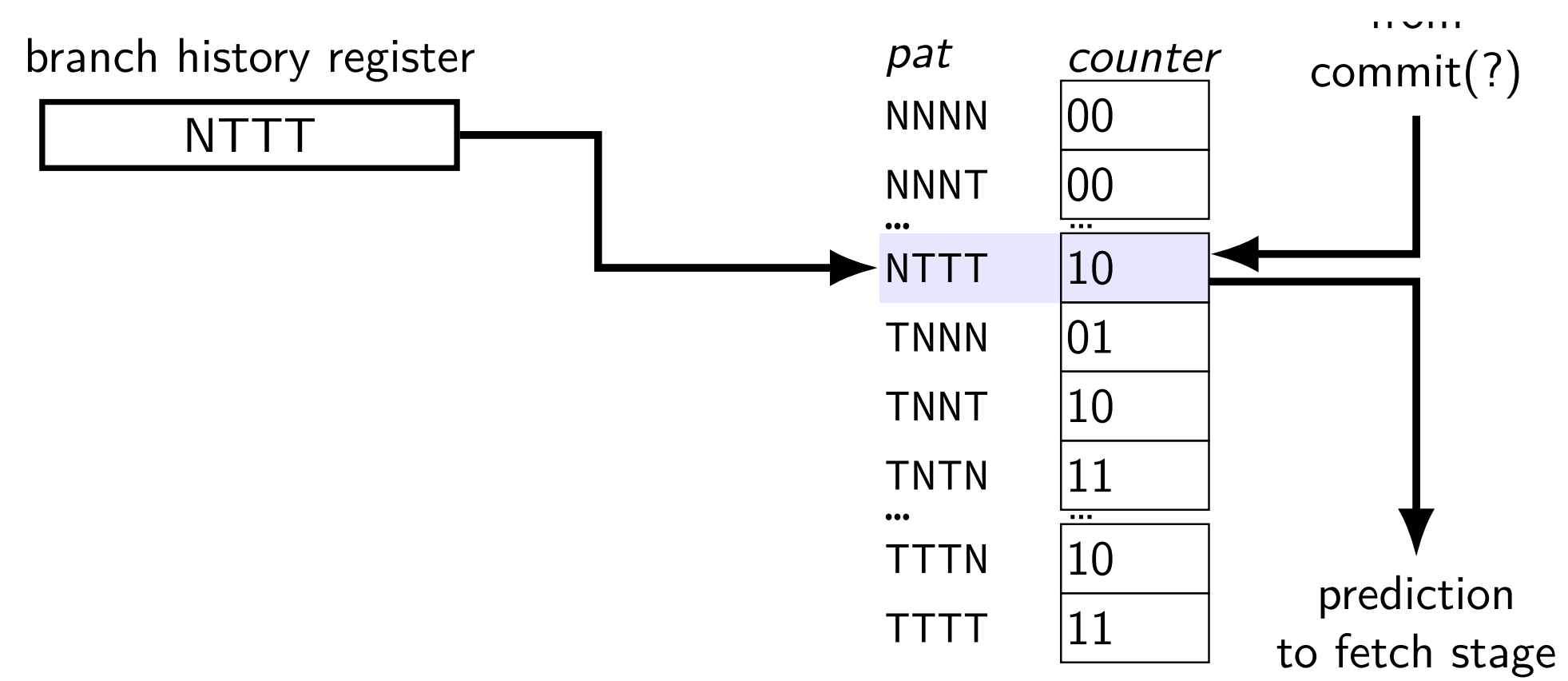
one predictor idea: ignore the PC

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)



global history predictor (1)

```

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

```

branch history register

NTTT

pat

NNNN
NNNT
...
NTTT
TNNN
TNNT
TNTN
...
TTTN
TTTT

counter

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

commit(?)

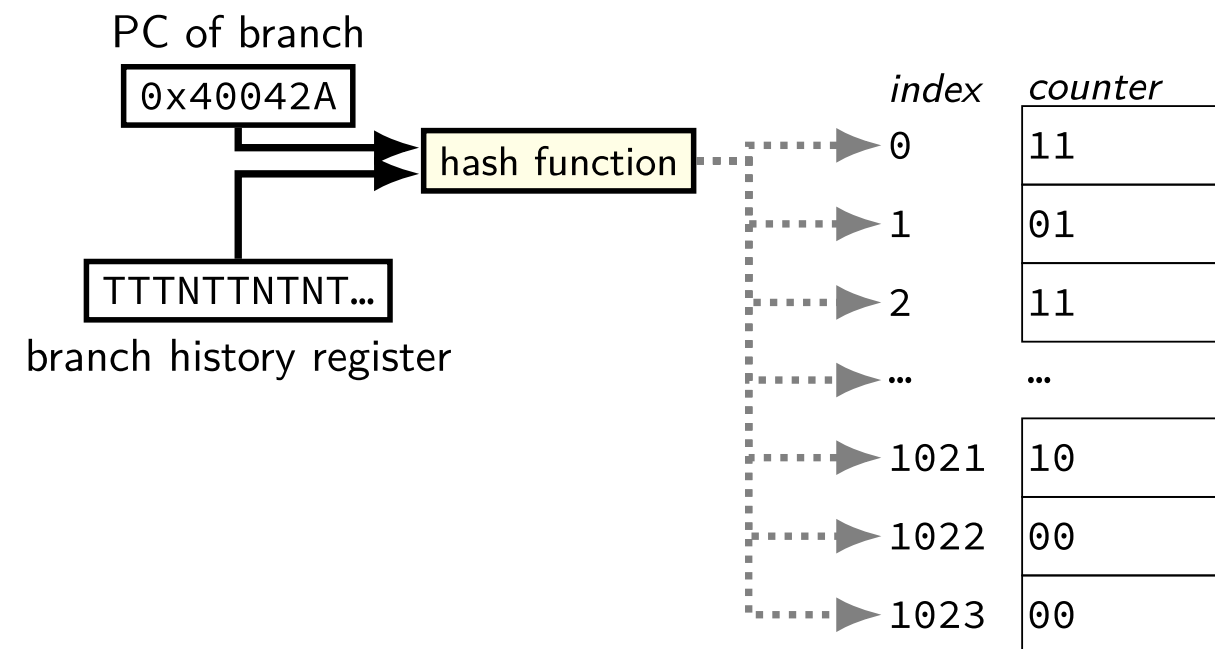
prediction
to fetch stage

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
3/mod 2	TTTT			not taken		TTTN

correlating predictor

global history *and* local info good together

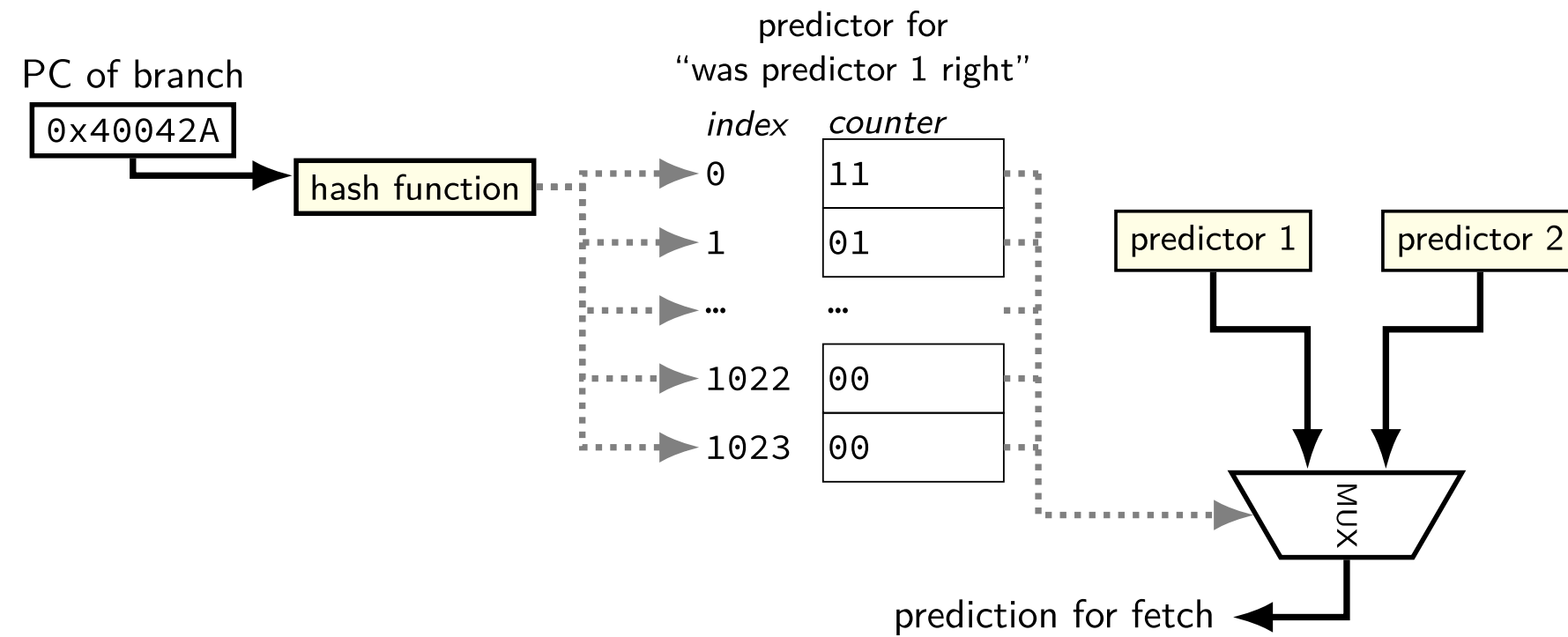
one idea: *combine history register + PC* (“gshare”)



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 TTTTNTN+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

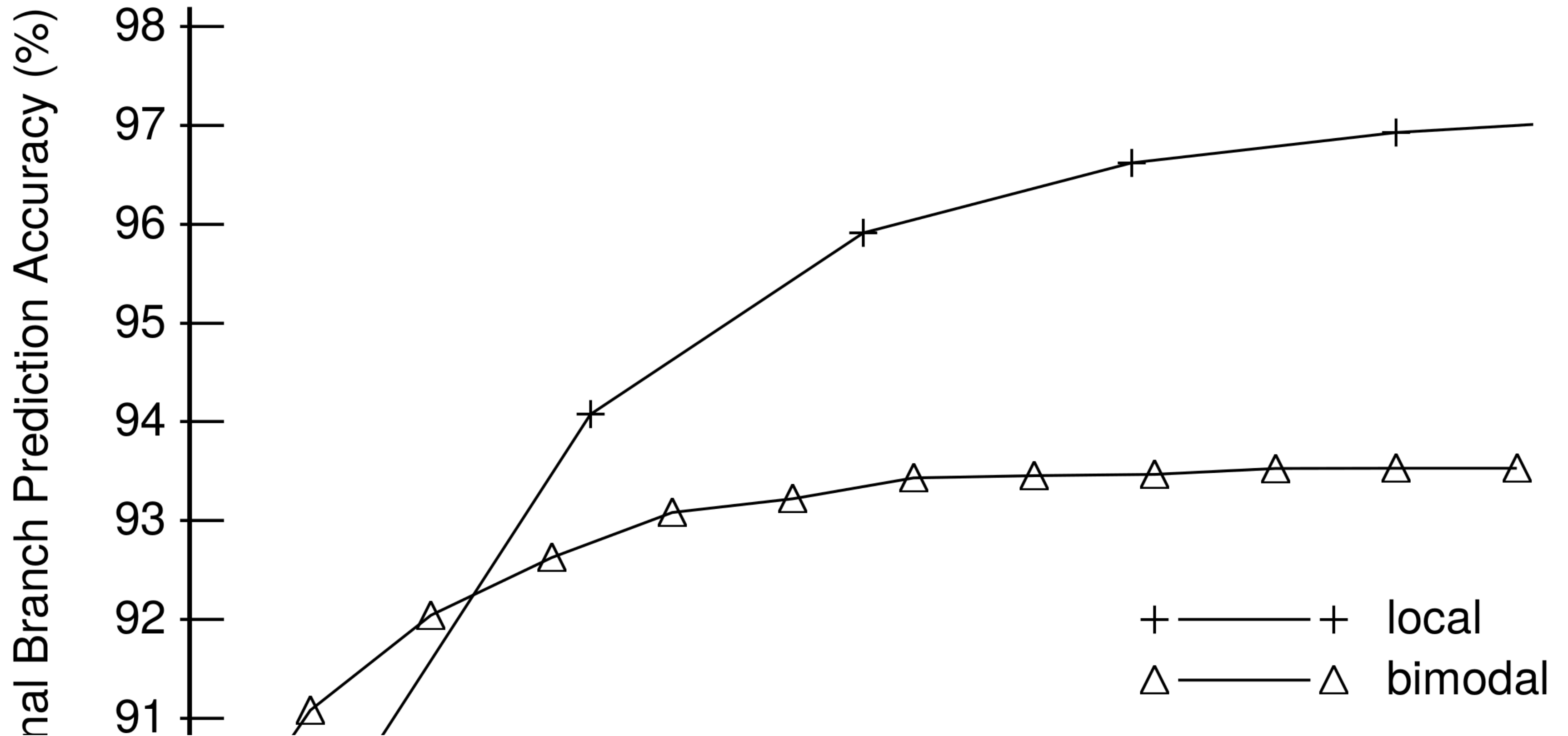
(not representative of modern workloads?)

rate for conditional branches on benchmark

variable table sizes

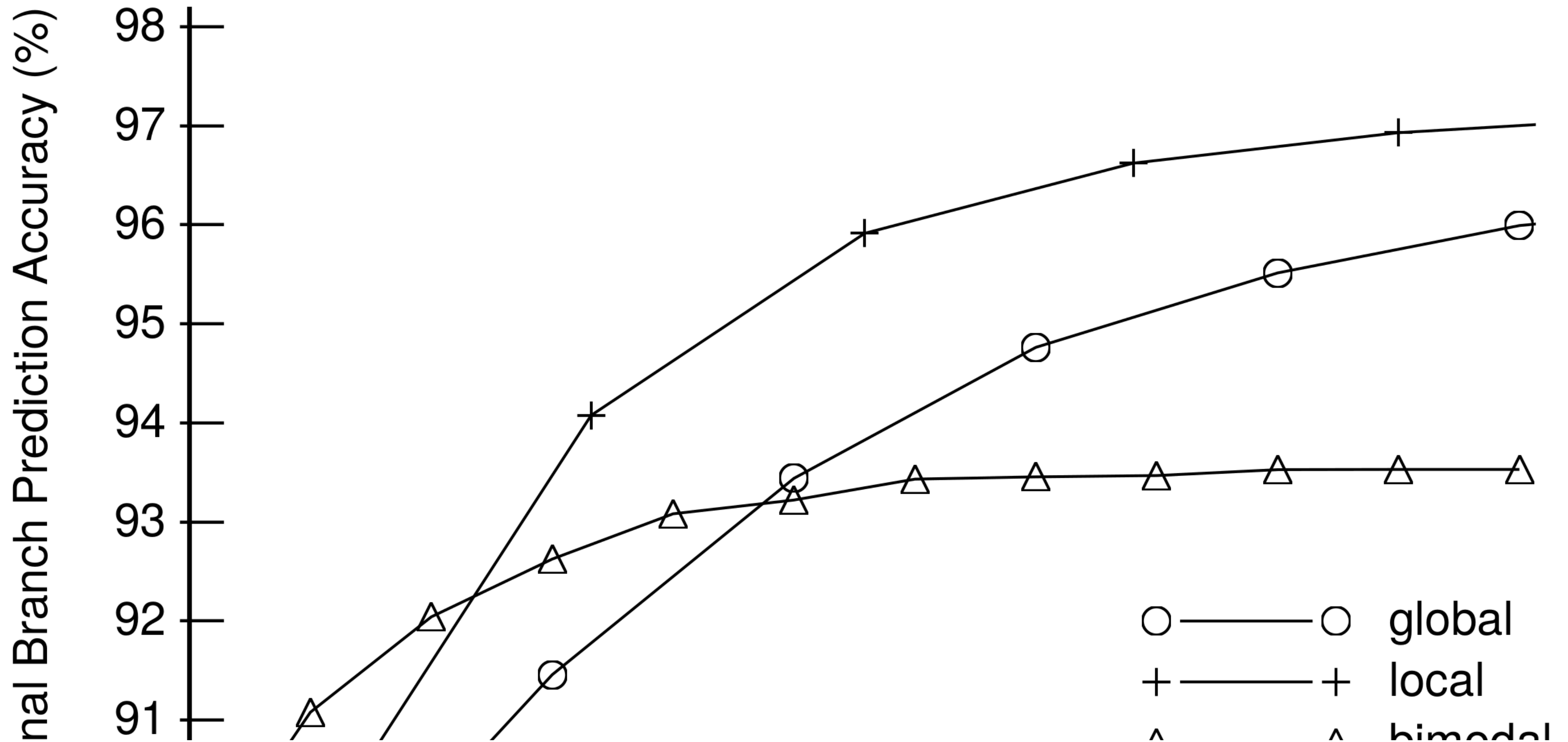
2-bit ctr + local history

from McFarling, "Combining Branch Predictors" (1993)



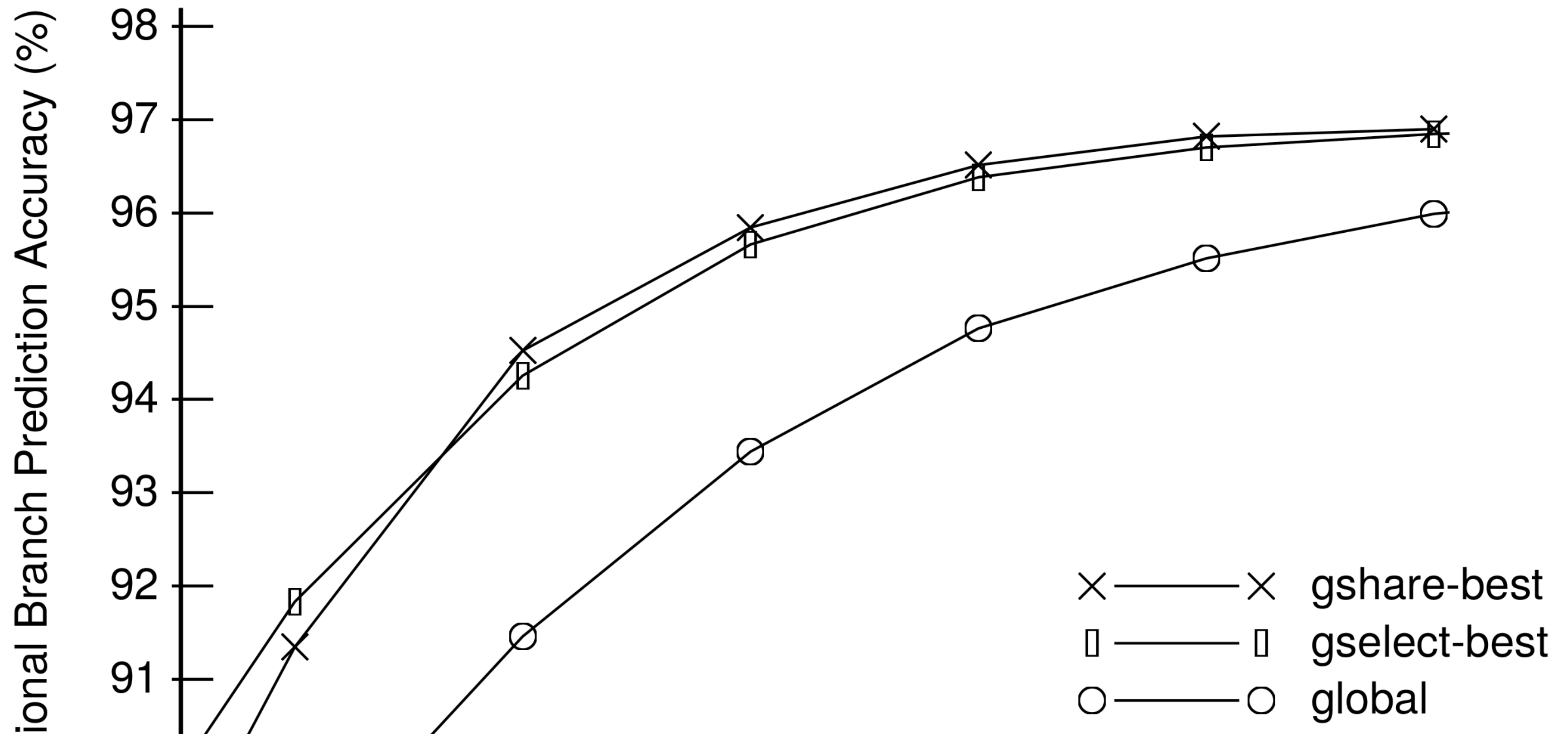
2-bit (bimodal) + local + global hist

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

backup slides

Backup slides