

pipelining 2

last time

cryptographic hashes

- impractical to find values with particular hash
- suitable summary for signatures

asymmetric key agreement

- specialized protocol (even though not strictly needed)
- private data \rightarrow key share
- my key share mixed with their private data = our key

TLS — asymmetric to do symmetric + certificates

pipelining

- divide into steps
- instr. 2 starts step A when instr. 1 starts step B
- instr. 3 starts step A when instr. 2 starts step B
- etc.

anonymous feedback (1)

“Can the TA Office Hour Queue actually be a queue and be first come first serve? It is very frustrating to be at office hours (one of the first people there) and then have people essentially cut you in line because (as a TA explained - the queue prioritizes people who have not gone to office hours before).”

anonymous feedback (2)

“I’ve gone to almost every lecture this semester and I felt a lot less prepared for the quiz questions on the secure channels unit than the other units. Not sure if the quiz was harder than others or the lecture was just harder for me to understand, but either way I think the lectures on this unit should be slowed down a bit in general, and in particular spend more time comparing and contrasting different security methods/attack types and include more diagrams illustrating exchanges of messages and keys.”

exercise: throughput/latency (1)

	cycle #	0	1	2	3	4	5	6	7	8
0x100: add %r8, %r9		F	D	E	M	W				
0x108: mov 0x1234(%r10), %r11			F	D	E	M	W			
0x110:					

suppose cycle time is 500 ps

exercise: latency of one instruction?

- A. 100 ps B. 500 ps C. 2000 ps D. 2500 ps E. something else

exercise: throughput/latency (1)

	cycle #	0	1	2	3	4	5	6	7	8
0x100: add %r8, %r9		F	D	E	M	W				
0x108: mov 0x1234(%r10), %r11			F	D	E	M	W			
0x110:					

suppose cycle time is 500 ps

exercise: latency of one instruction?

- A. 100 ps B. 500 ps C. 2000 ps D. 2500 ps E. something else

exercise: throughput overall?

- A. 1 instr/100 ps B. 1 instr/500 ps C. 1 instr/2000ps D. 1 instr/2500 ps
E. something else

exercise: throughput/latency (2)

0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: ...

cycle #	0	1	2	3	4
	F	D	E	M	W
		F	D	E	M
				...	

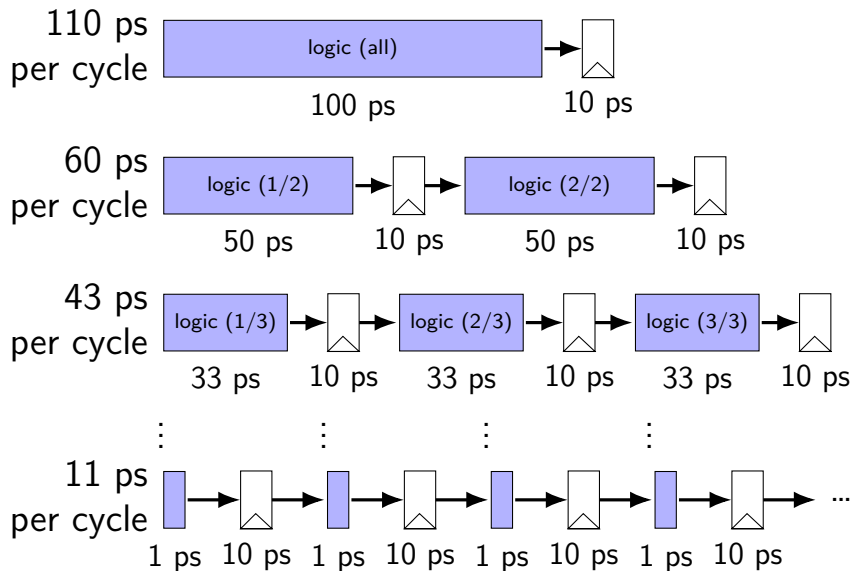
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: ...

cycle #	0	1	2	3	4	5	6	7	8
	F1	F2	D1	D2	E1	E2	M1	M2	W1
		F1	F2	D1	D2	E1	E2	M1	M2
				...					

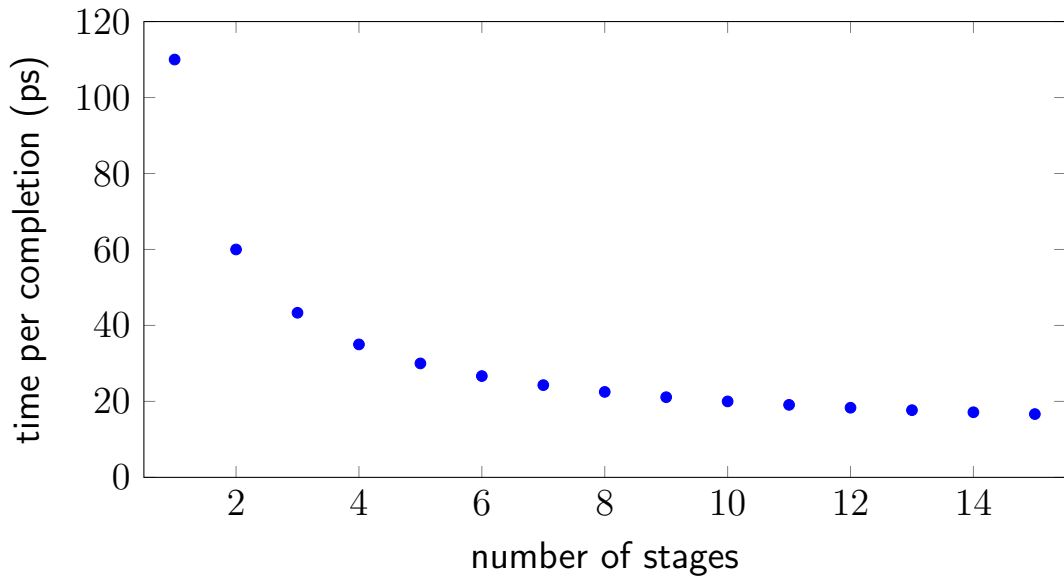
double number of pipeline stages (to 10) + decrease cycle time from 500 ps to 250 ps — throughput?

- A. 1 instr/100 ps B. 1 instr/250 ps C. 1 instr/1000ps D. 1 instr/5000 ps
E. something else

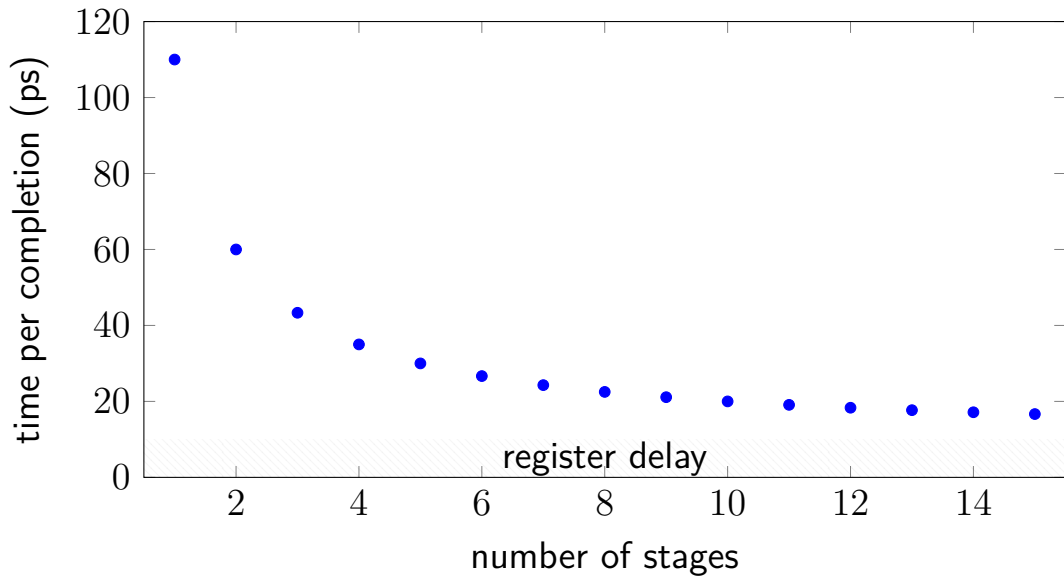
diminishing returns: register delays



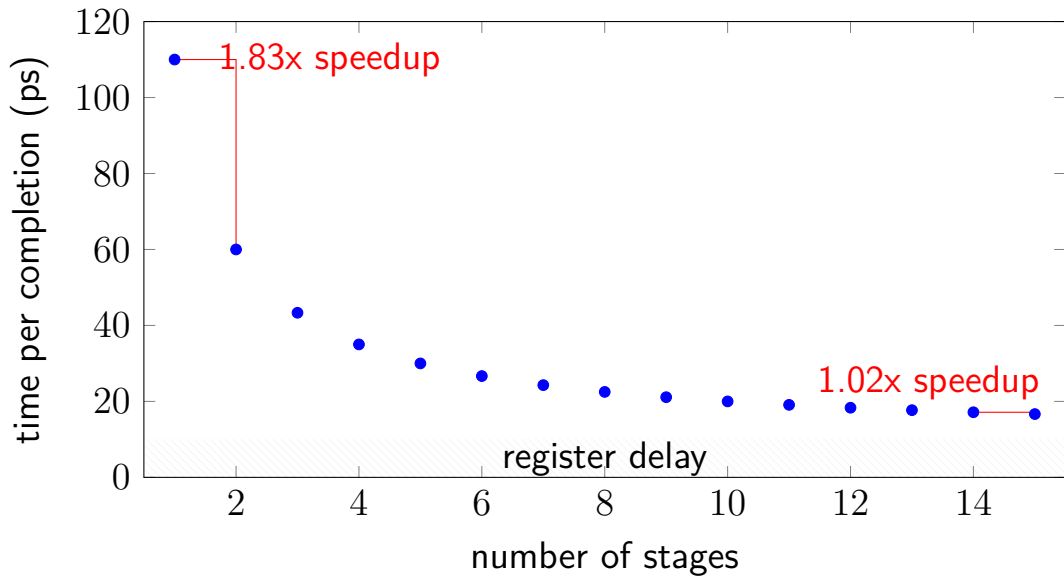
diminishing returns: register delays



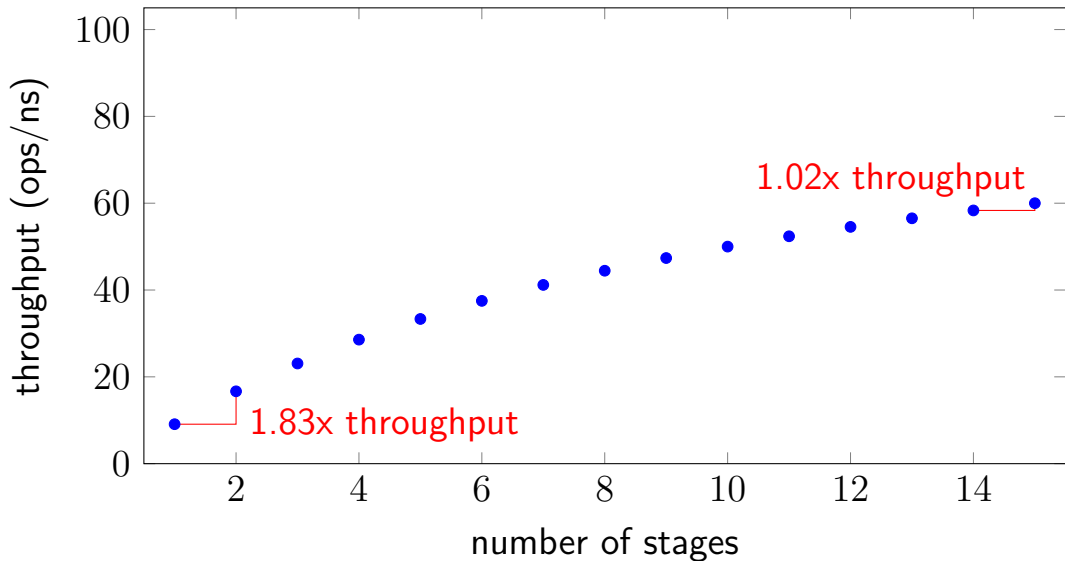
diminishing returns: register delays



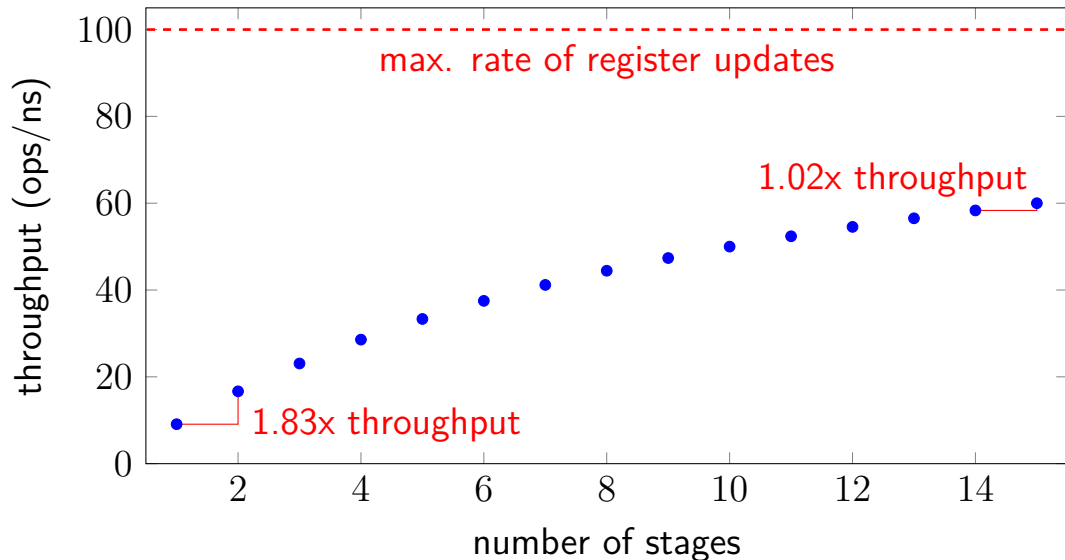
diminishing returns: register delays



diminishing returns: register delays



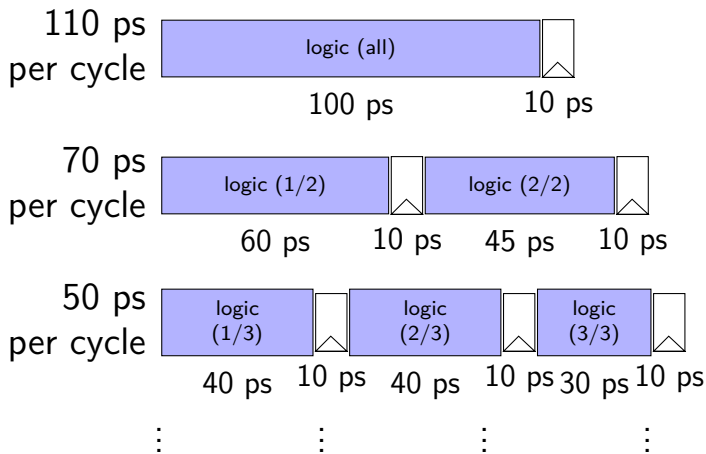
diminishing returns: register delays



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

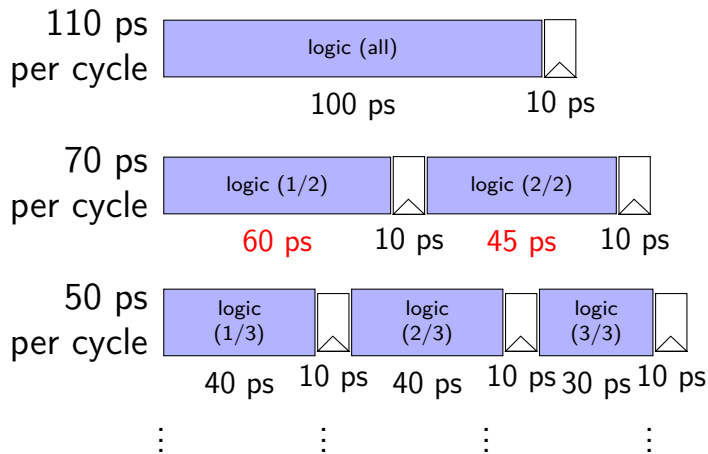
Probably not...



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

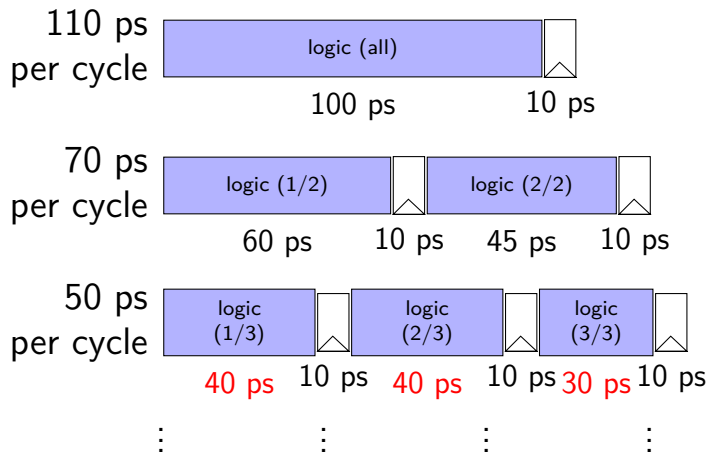
Probably not...



diminishing returns: uneven split

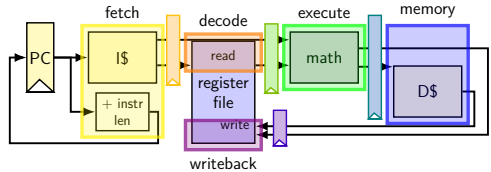
Can we split up some logic (e.g. adder) arbitrarily?

Probably not...



a data hazard

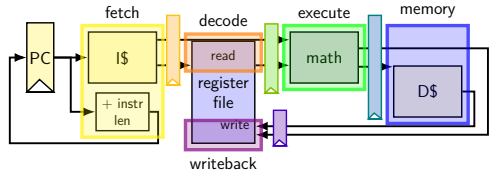
```
// initially %r8 = 800,
//                %r9 = 900, etc.
addq %r8, %r9    // R8 + R9 -> R9
addq %r9, %r8    // R9 + R8 -> R9
addq ...
addq ...
```



	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

a data hazard

```
// initially %r8 = 800,
//                %r9 = 900, etc.
addq %r8, %r9    // R8 + R9 -> R9
addq %r9, %r8    // R9 + R8 -> R9
addq ...
addq ...
```



	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

data hazard

addq %r8, %r9 // (1)

addq %r9, %r8 // (2)

step#	pipeline implementation	ISA specification
1	read r8, r9 for (1)	read r8, r9 for (1)
2	read r9, r8 for (2)	write r9 for (1)
3	write r9 for (1)	read r9, r8 for (2)
4	write r8 for (2)	write r8 for (2)

pipeline reads **older value**...

instead of value ISA says was just written

data hazard compiler solution

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

one solution: **change the ISA**

all addqs take effect **three instructions later**

(assuming can read register value while it is being written back)

make it **compiler's job**

problem: recompile everytime processor changes?

data hazard compiler solution

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

one solution: **change the ISA**

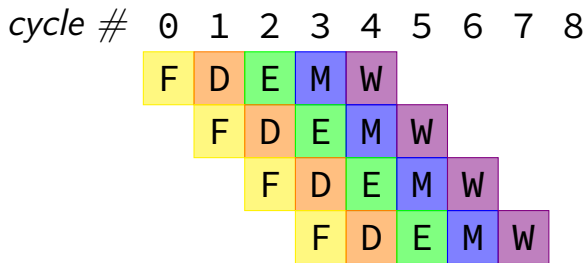
all addqs take effect **three instructions later**
(assuming can read register value while it is being written back)

make it **compiler's job**

problem: recompile everytime processor changes?

stalling/nop pipeline diagram (1)

add %r8, %r9
nop
nop
addq %r9, %r8



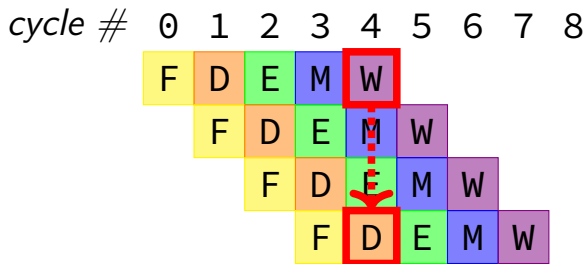
stalling/nop pipeline diagram (1)

add %r8, %r9

nop

nop

addq %r9, %r8



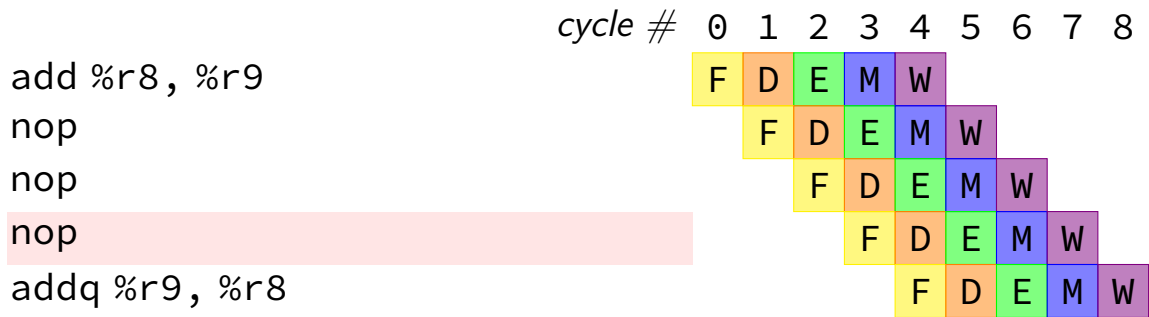
assumption:

if writing register value

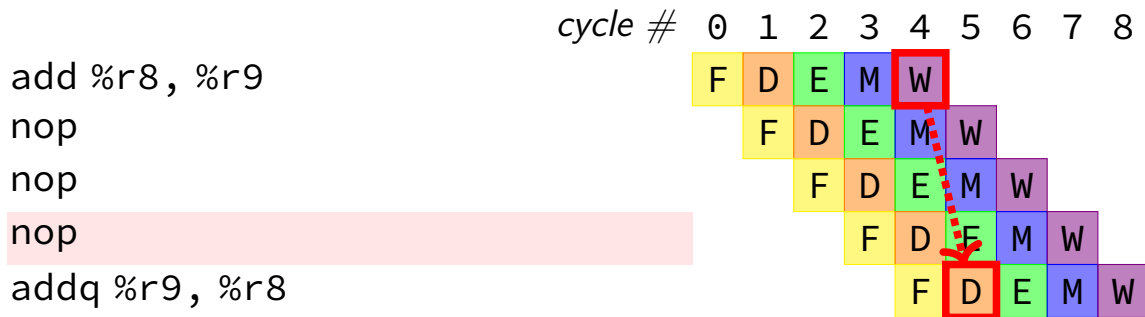
register file will return that value for reads

not actually way register file worked in single-cycle CPU
(e.g. can read old %r9 while writing new %r9)

stalling/nop pipeline diagram (2)



stalling/nop pipeline diagram (2)



if we didn't modify the register file, we'd need an extra cycle

data hazard hardware solution

```
addq %r8, %r9  
// hardware inserts: nop  
// hardware inserts: nop  
addq %r9, %r8
```

how about hardware add nops?

called **stalling**

extra logic:

- sometimes don't change PC

- sometimes put do-nothing values in pipeline registers

opportunity

```
// initially %r8 = 800,  
//                %r9 = 900, etc.
```

```
0x0: addq %r8, %r9
```

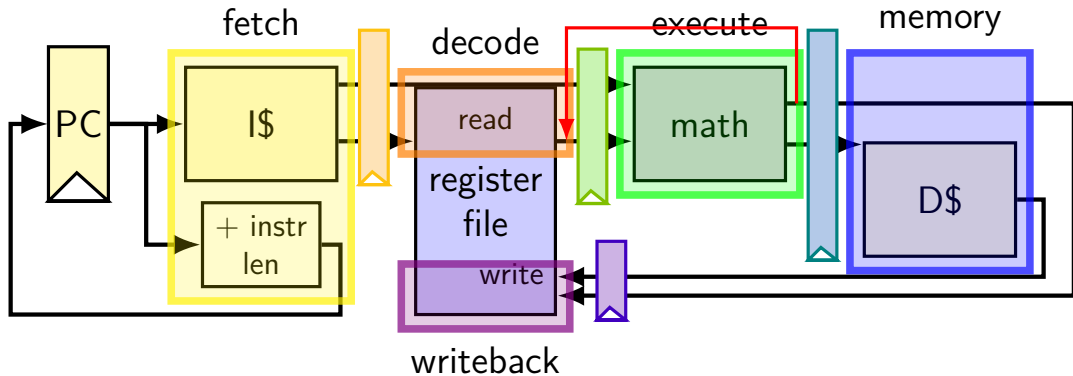
```
0x2: addq %r9, %r8
```

...

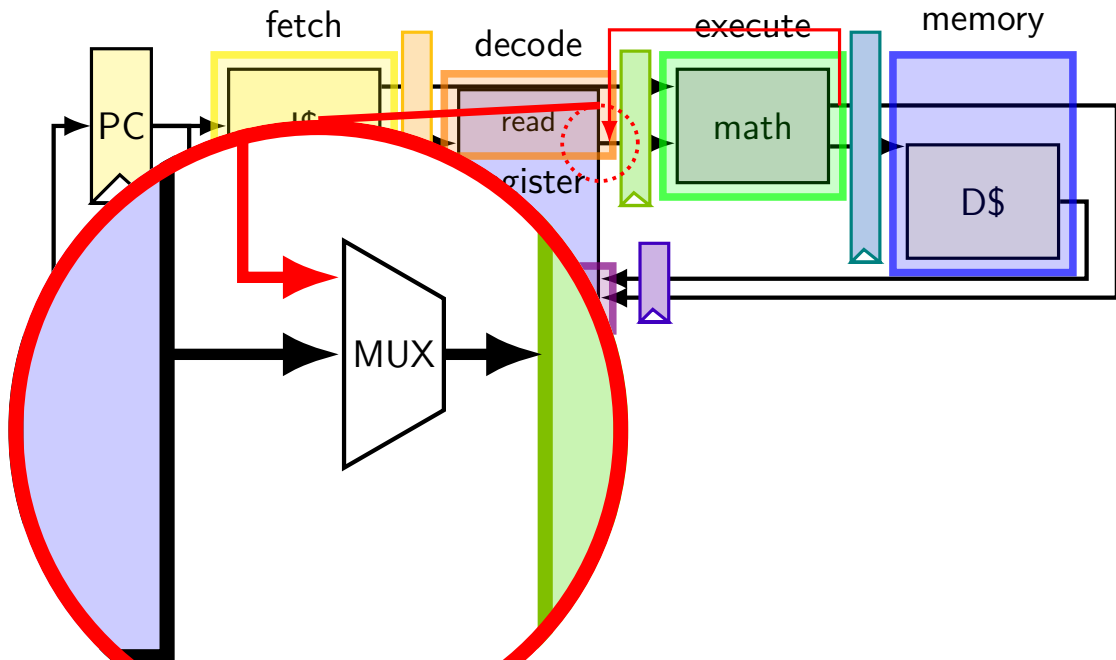
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



opportunity 2

```
// initially %r8 = 800,  
//                %r9 = 900, etc.
```

```
0x0: addq %r8, %r9
```

```
0x2: nop
```

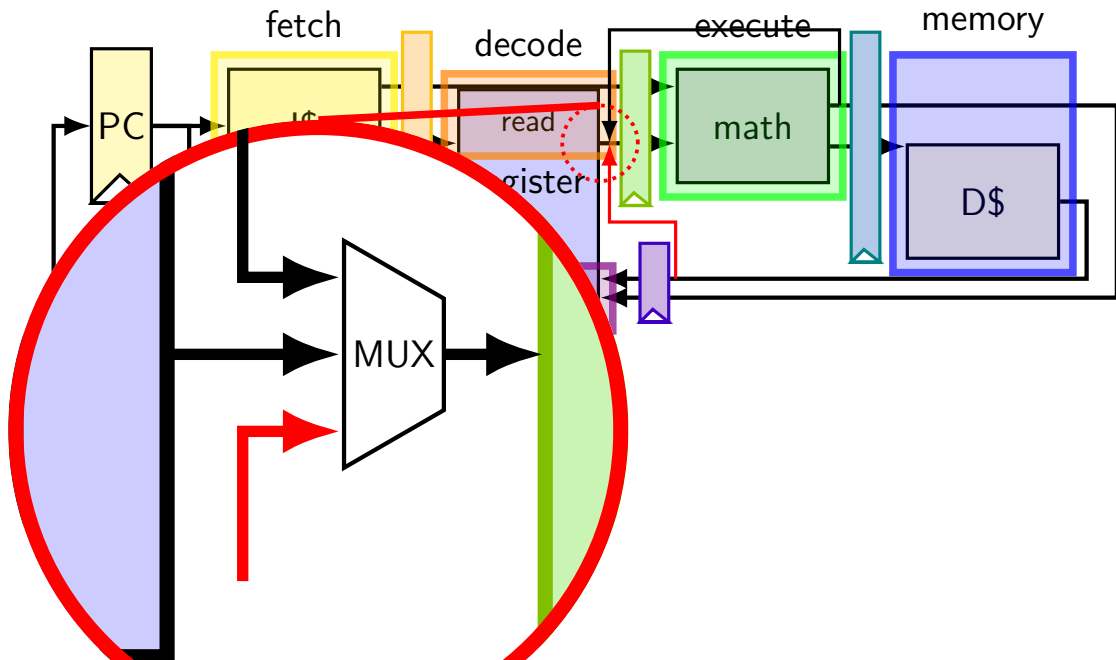
```
0x3: addq %r9, %r8
```

```
...
```

	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2	0x3	---	---	800	900	9				
3		9	8	---	---	---	1700	9		
4				900	800	8	---	---	1700	9
5							1700	9	---	---
6									1700	9

should be 1700

exploiting the opportunity



exercise: forwarding paths

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

subq %r8, %r10

xorq %r8, %r9

andq %r9, %r8

F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	

in subq, %r8 is _____ addq.

in xorq, %r9 is _____ addq.

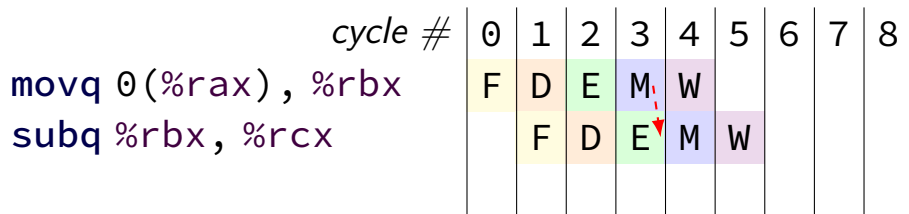
in andq, %r9 is _____ addq.

in andq, %r9 is _____ xorq.

A: not forwarded from

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

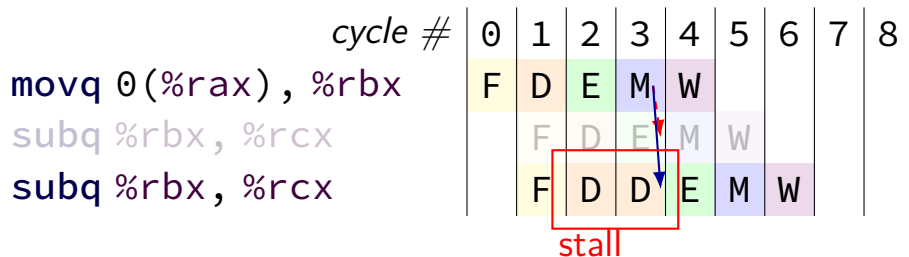
unsolved problem



combine stalling and forwarding to resolve hazard

assumption in diagram: hazard detected in subq's decode stage
(since easier than detecting it in fetch stage)

unsolved problem



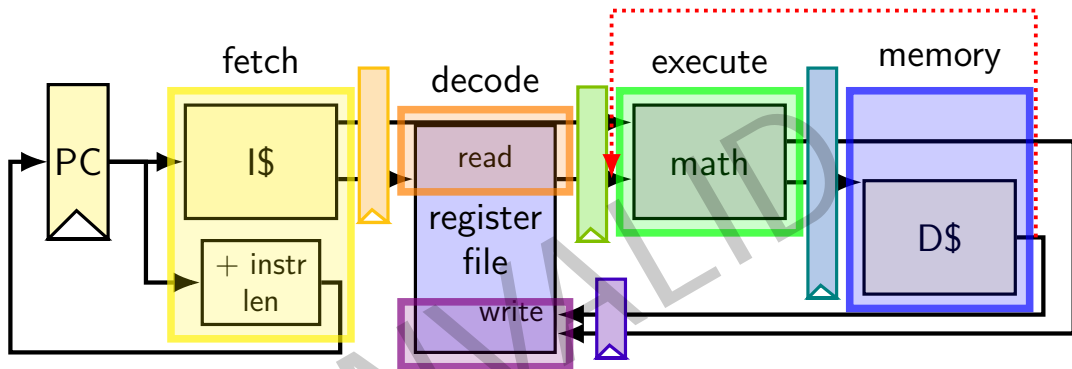
combine stalling and forwarding to resolve hazard

assumption in diagram: hazard detected in subq's decode stage
(since easier than detecting it in fetch stage)

solveable problem

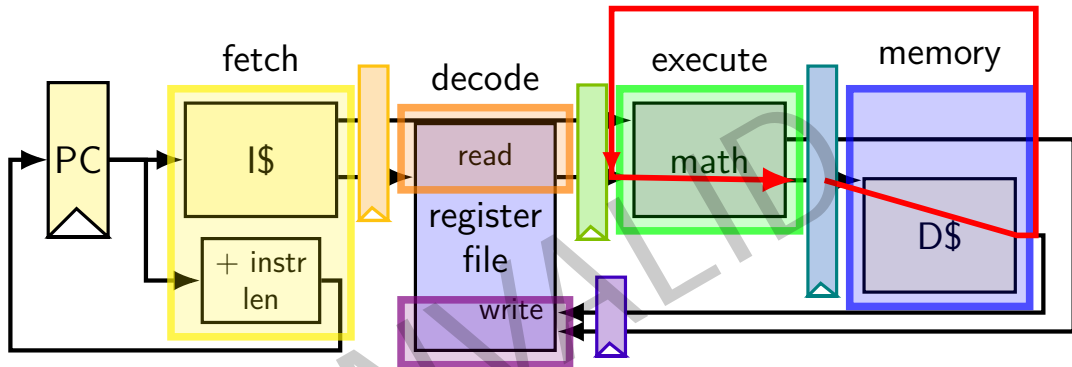
	cycle #	0	1	2	3	4	5	6	7	8
movq 0(%rax), %rbx		F	D	E	M	W				
movq %rbx, 0(%rcx)			F	D	E	M	W			

why can't we...



clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

why can't we...



clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

control hazard

0x00: cmpq %r8, %r9

0x08: je 0xFFFF

0x10: addq %r10, %r11

	fetch	fetch→decode		decode→execute		execute→write	execute→writeback		...
cycle	PC	rA	rB	R[rA]	R[rB]	result
0	0x0								
1	0x8	8	9						
2	???	---	---	800	900				
3	???	---	---	---	---	less than			

control hazard

0x00: cmpq %r8, %r9

0x08: je 0xFFFF

0x10: addq %r10, %r11

	fetch	fetch→decode		decode→execute		execute→write	execute→writeback		...
cycle	PC	rA	rB	R[rA]	R[rB]	result
0	0x0								
1	0x8	8	9						
2	???	---	---	800	900				
3	???	---	---	---	---	less than			

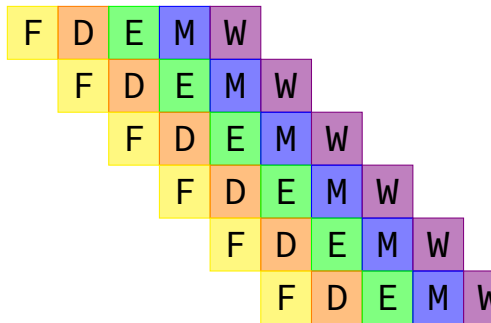
0xFFFF if $R[8] = R[9]$; 0x10 otherwise

jXX: stalling?

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

```
...
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

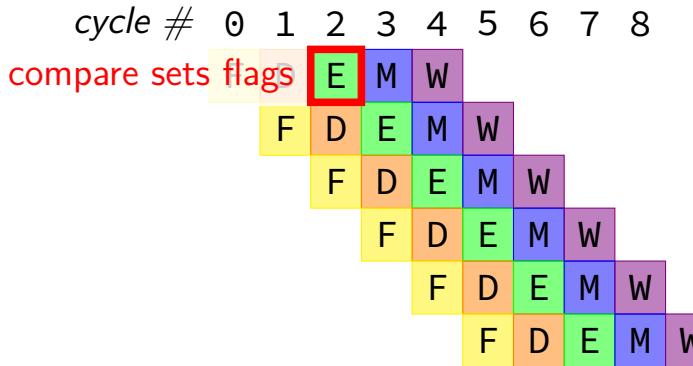
cycle # 0 1 2 3 4 5 6 7 8



jXX: stalling?

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

```
...
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



jXX: stalling?

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

...

```
cmpq %r8, %r9
```

```
jne LABEL
```

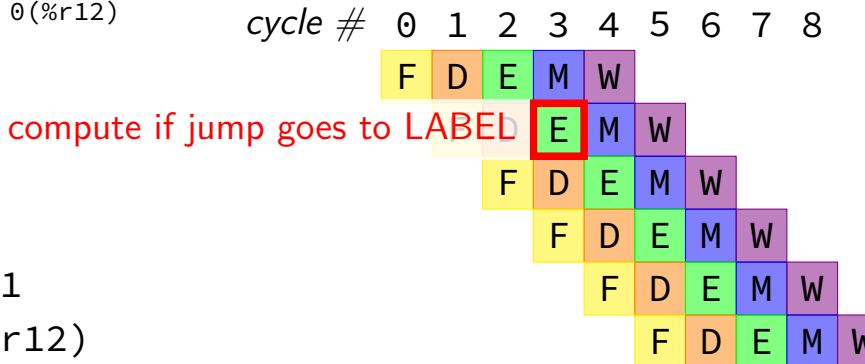
```
(do nothing)
```

```
(do nothing)
```

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

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

...

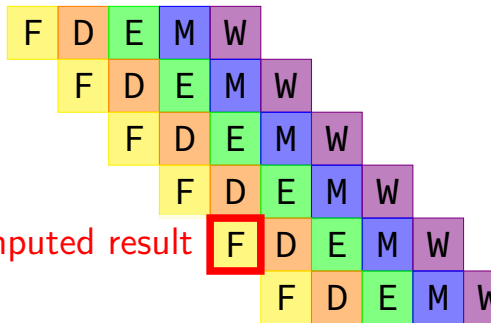


jXX: stalling?

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

```
...
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

cycle # 0 1 2 3 4 5 6 7 8



use computed result

making guesses

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

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

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

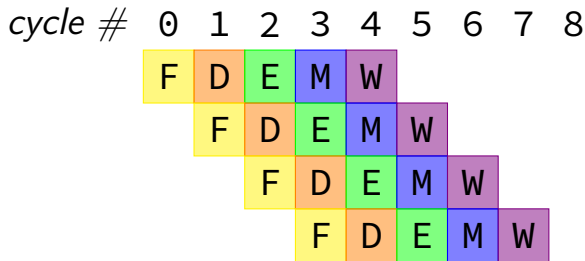
right: 2 cycles faster!; wrong: undo guess before too late

jXX: speculating right (1)

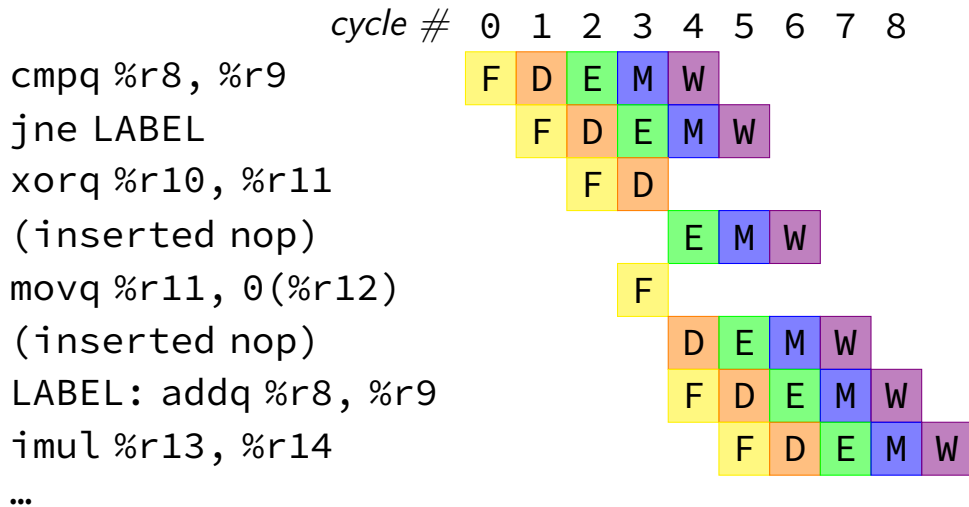
```
cmpq %r8, %r9
jne LABEL
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

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

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



jXX: speculating wrong



jXX: speculating wrong

cycle # 0 1 2 3 4 5 6 7 8

cmpq %r8, %r9

jne LABEL

xorq %r10, %r11

(inserted nop)

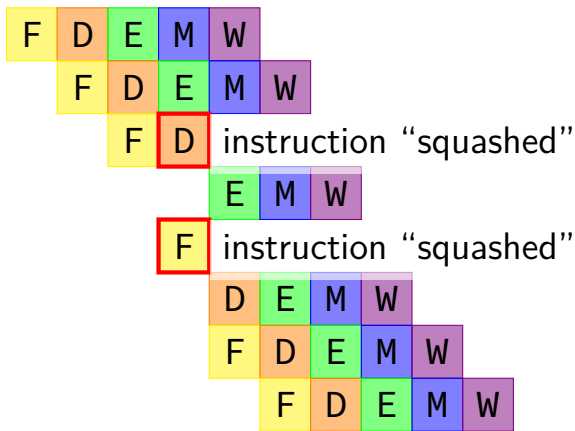
movq %r11, 0(%r12)

(inserted nop)

LABEL: addq %r8, %r9

imul %r13, %r14

...



“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

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

performance

hypothetical instruction mix

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

performance

hypothetical instruction mix

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

hazards versus dependencies

dependency — X needs result of instruction Y?

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

hazard — will it not work in some pipeline?

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

ex.: dependencies and hazards (1)

addq %rax, %rbx

subq %rax, %rcx

movq \$100, %rcx

addq %rcx, %r10

addq %rbx, %r10

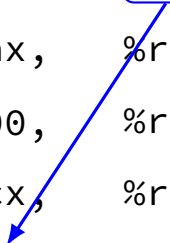
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

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



where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

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

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

where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

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

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

where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

`addq/andq` is hazard with 5-stage pipeline

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

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

more hazards with more pipeline stages

exercise: different pipeline

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

result only available near end of second execute stage

where does forwarding, stalls occur?

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

	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

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

PC of branch

0x40042A

hash function

index *prediction/
last result?*

0

taken (1)

1

not taken (0)

2

taken (1)

3

taken (1)

...

...

14

not taken (0)

15

taken (1)

predict: repeat last

PC of branch

0x40042A

hash function

index *prediction/
last result?*

0 taken (1)

1 not taken (0)

2 taken (1)

3 taken (1)

...

14 not taken (0)

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

predict: repeat last

PC of branch

0x40042A

hash function

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

predict: repeat last

PC of branch

0x40042A

hash function

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

prediction
to fetch stage

predict: repeat last

PC of branch

0x40042A

hash function

index prediction/
last result?

0

taken (1)

1

not taken (0)

2

taken (1)

3

taken (1)

...

...

14

not taken (0)

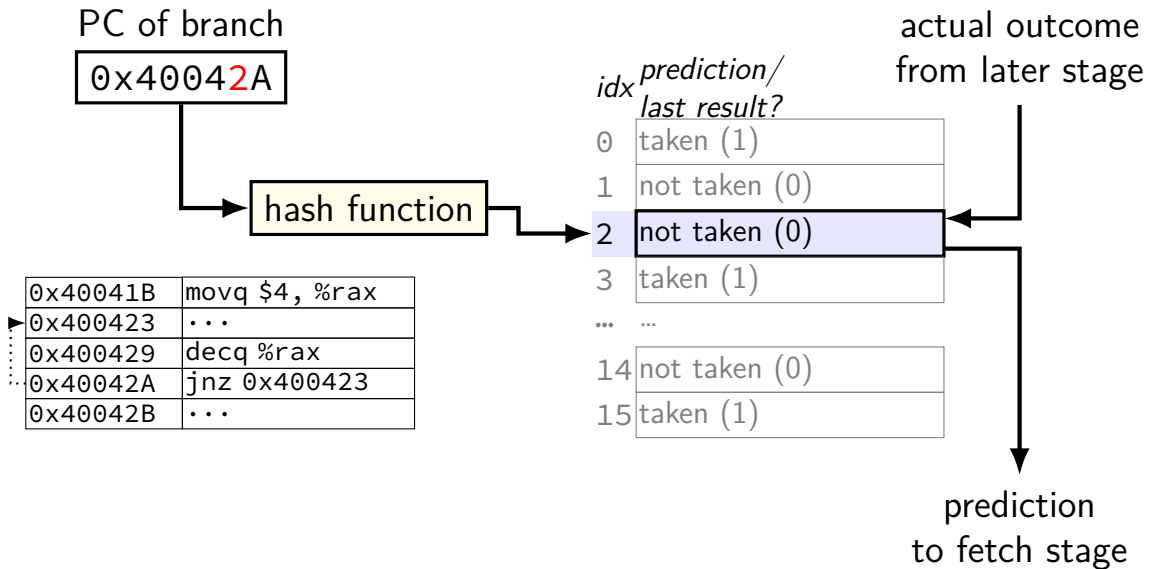
15

taken (1)

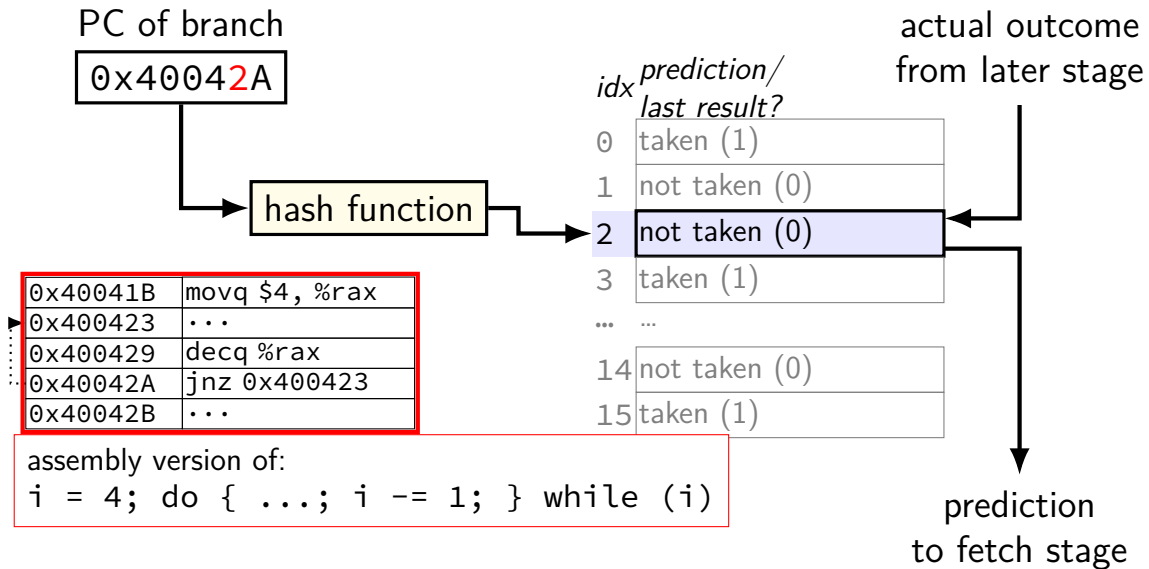
actual outcome
(from later stage)

prediction
to fetch stage

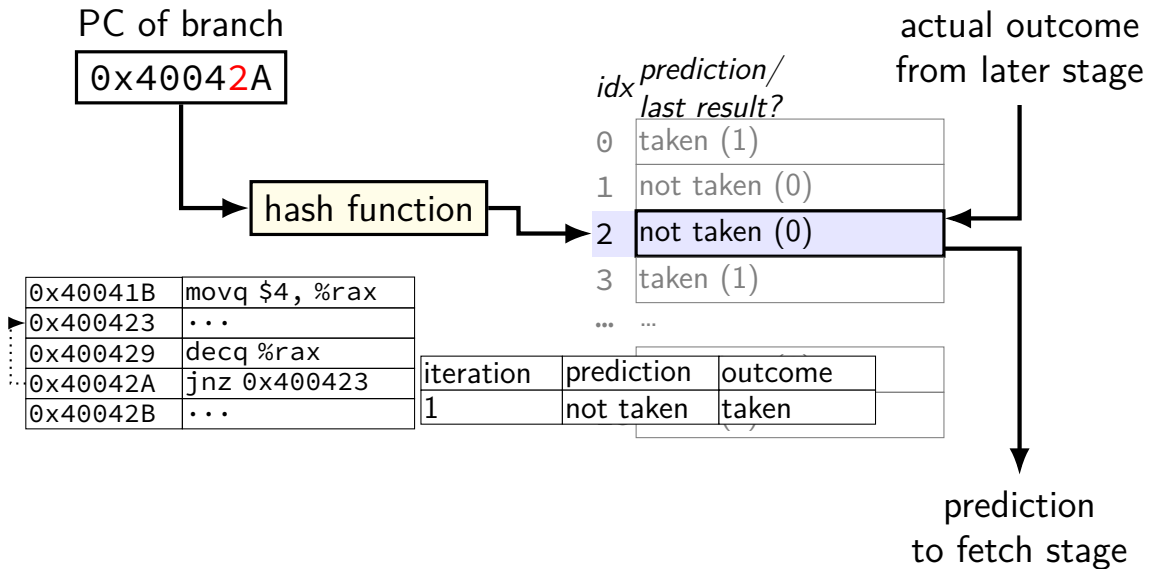
example



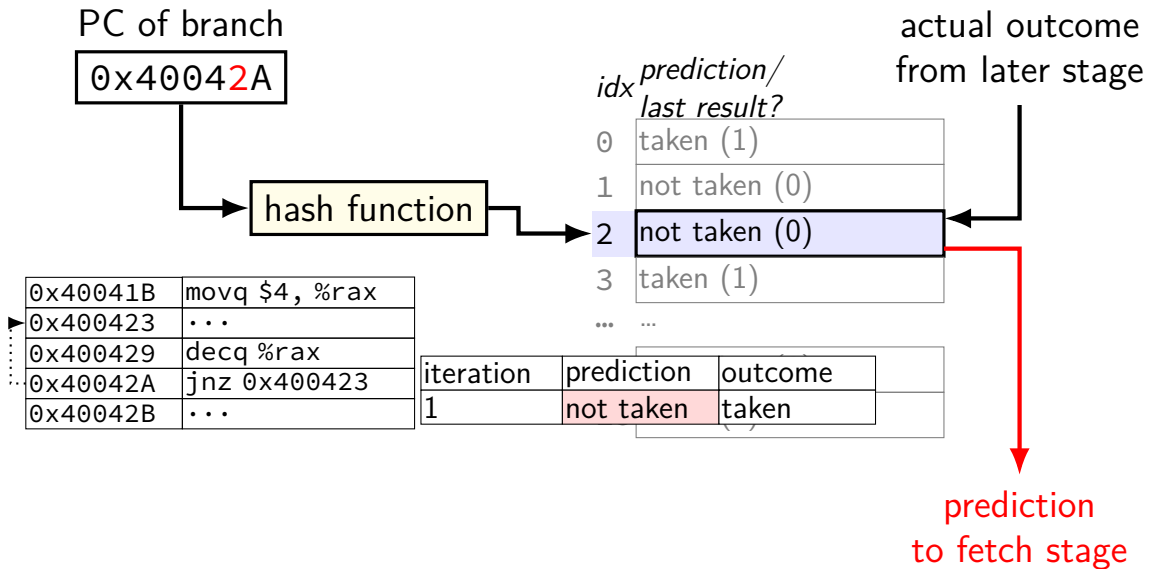
example



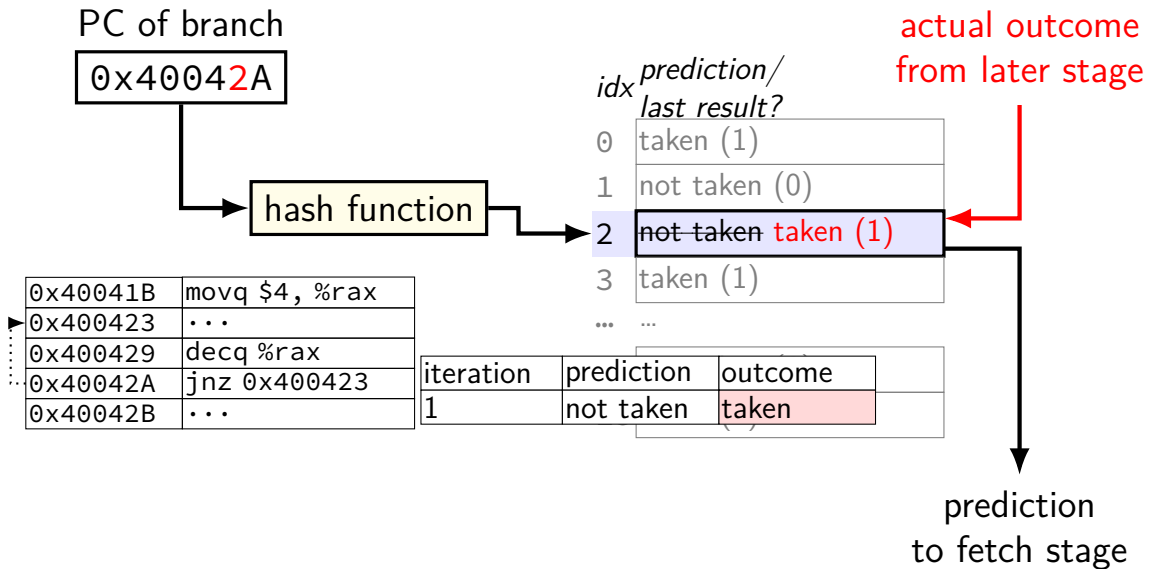
example



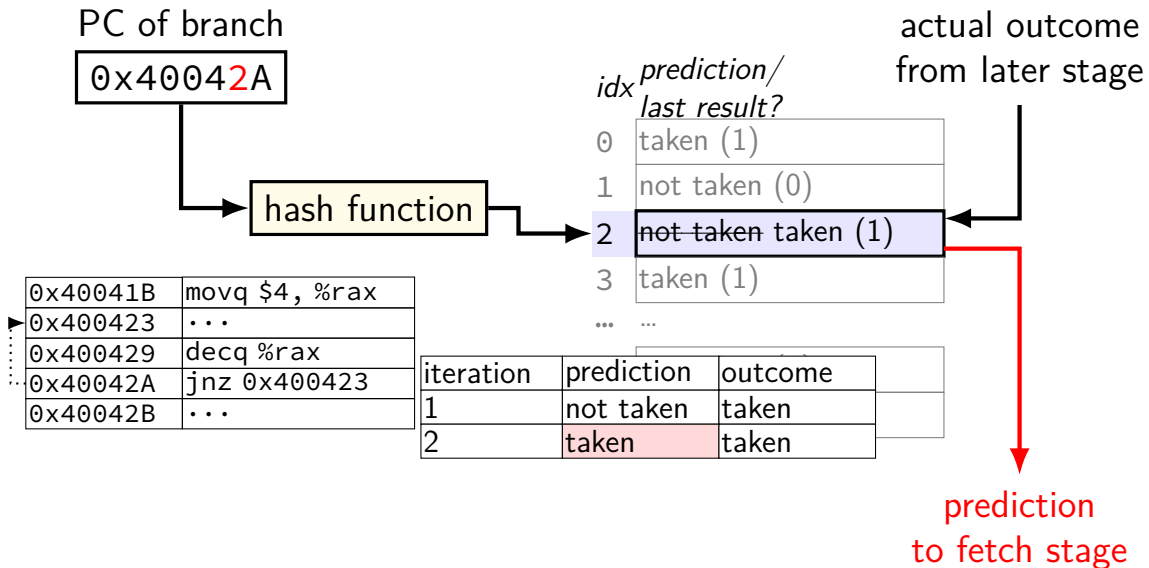
example



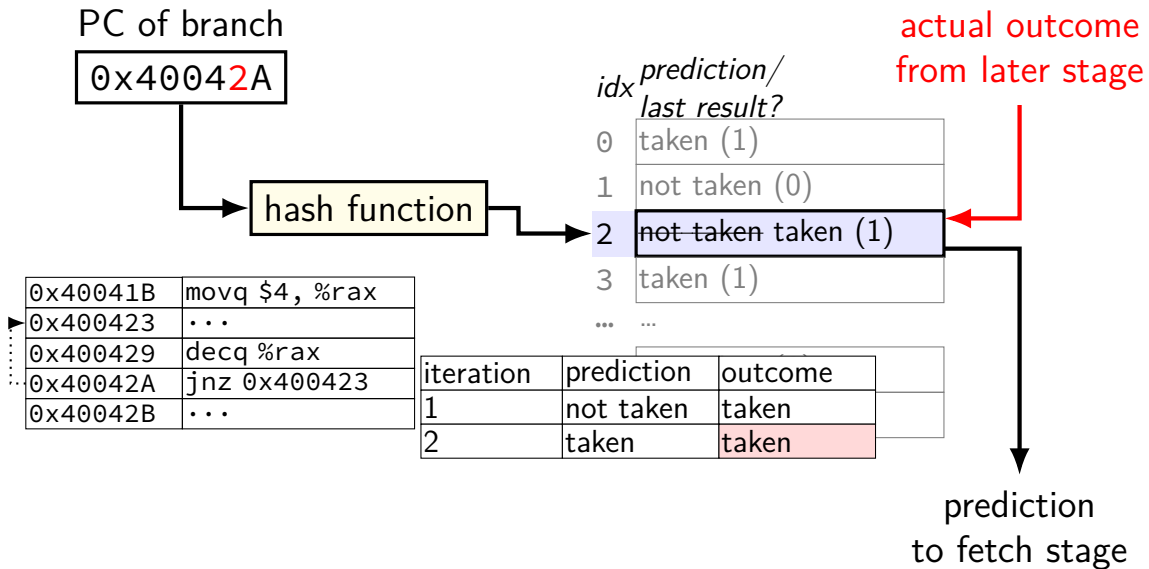
example



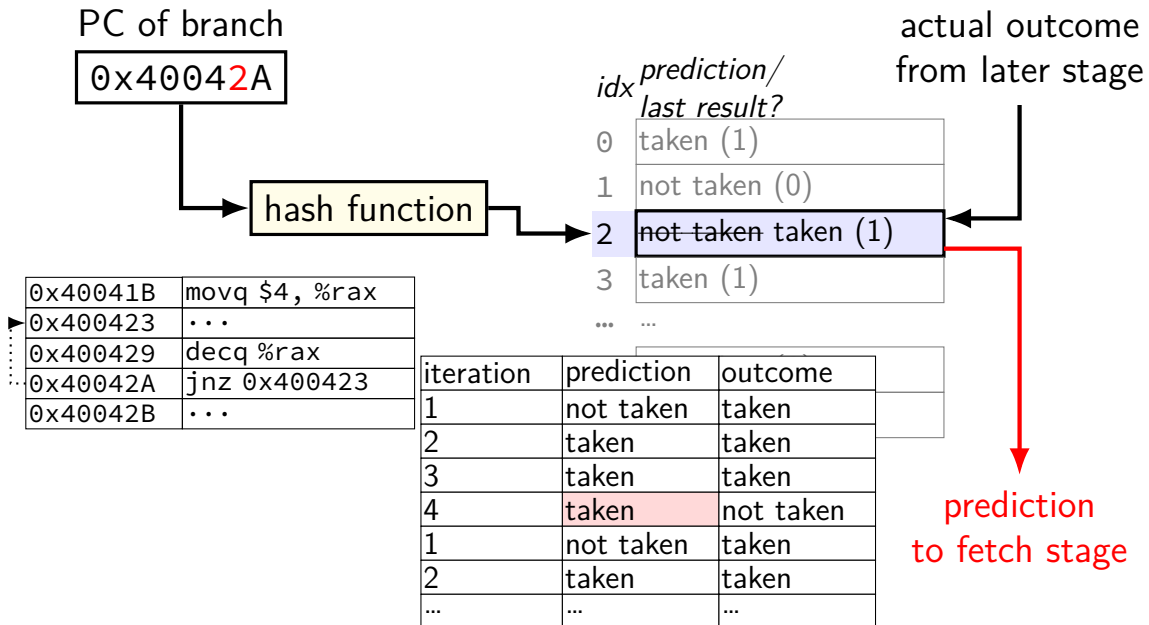
example



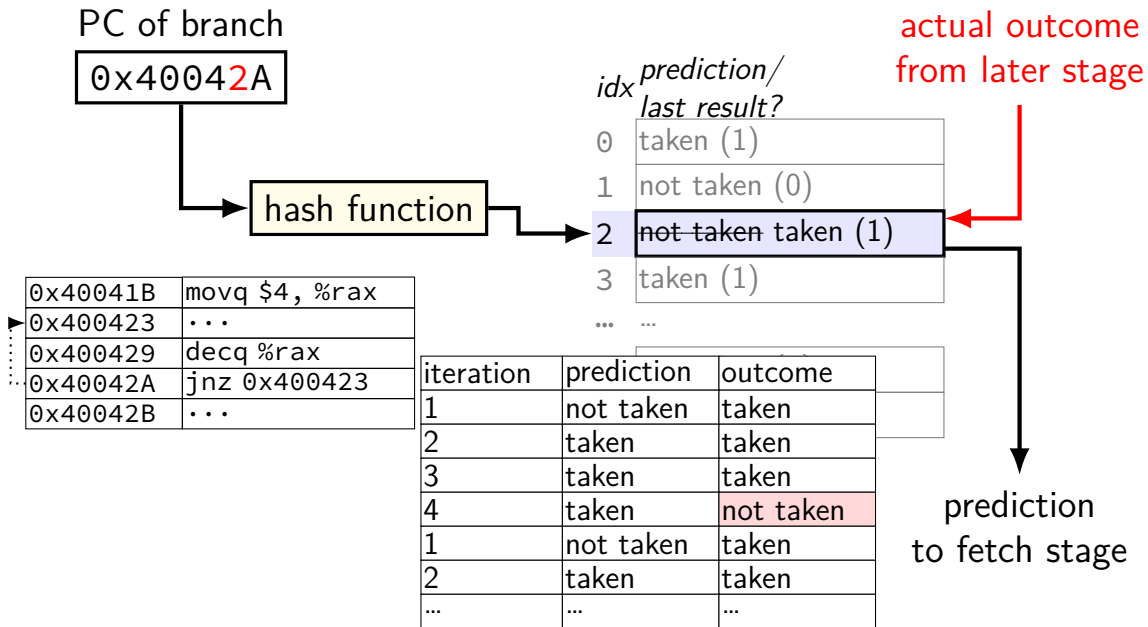
example



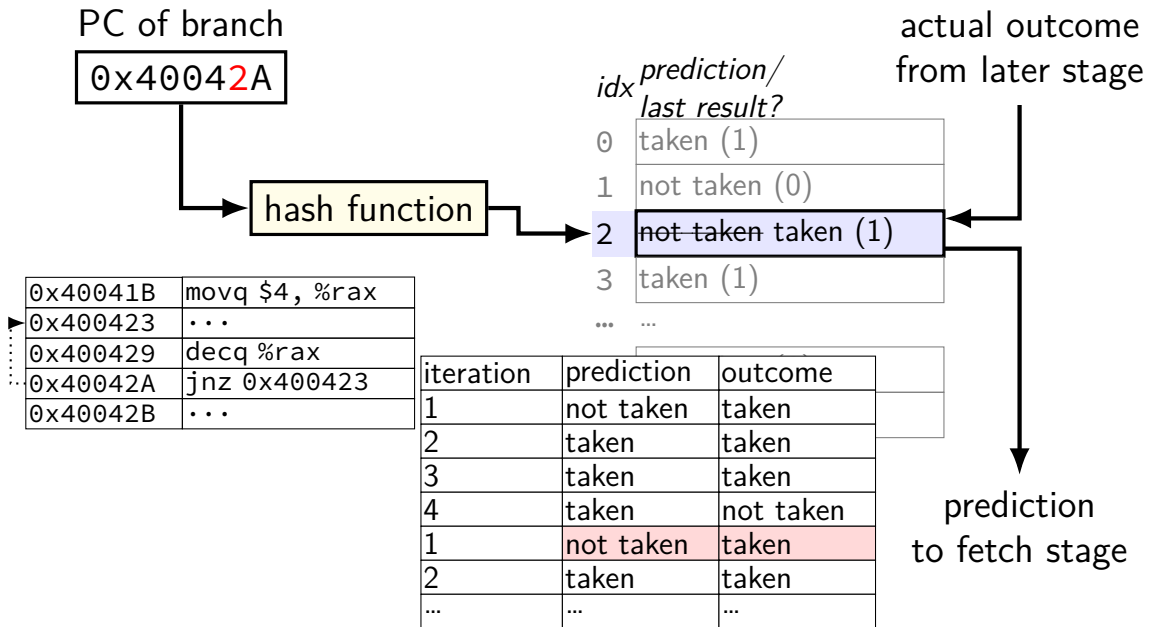
example



example



example



collisions?

two branches could have same hashed PC

nothing in table tells us about this

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

is it worth it?

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

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

collision results

possibility 1: both branches usually taken

no actual conflict — prediction is better(!)

possibility 2: both branches usually not taken

no actual conflict — prediction is better(!)

possibility 3: one branch taken, one not taken

performance probably worse

1-bit predictor for loops

predicts first and last iteration wrong

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

everything else correct

exercise

use 1-bit predictor on this loop

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

what is the conditional jump misprediction rate?

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

exercise

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

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	✓
...

beyond local 1-bit predictor

can predict using more historical info

whether taken last several times

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

pattern of how taken recently

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

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

take into account conditional jumps in surrounding code

example: loops with if statements will have regular patterns

predicting ret: minstack of return addresses

predicting ret — minstack in processor registers

push on minstack on call; pop on ret

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

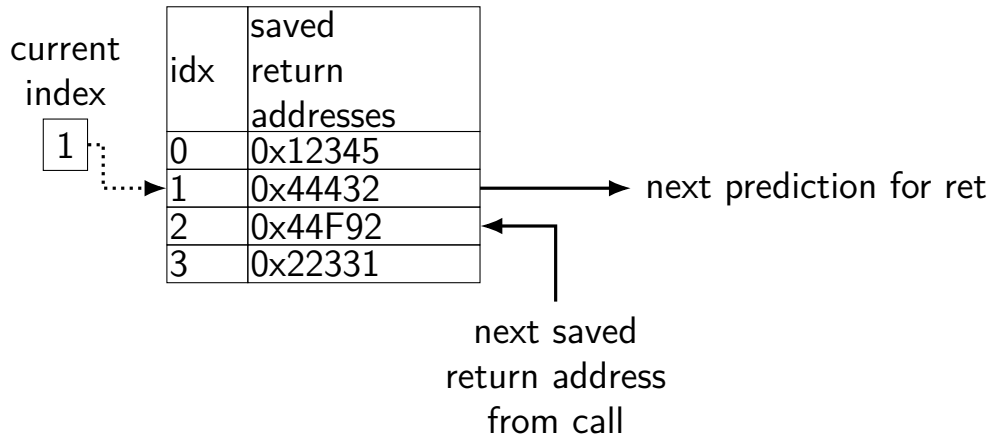
baz return address
bar return address
foo return address

(partial?) stack
in CPU registers

stack in memory

4-entry return address stack

4-entry return address stack in CPU



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

on ret: read prediction from index, decrement index

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	0x3FFFFF3	...
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	...

```
0x3FFFFF3:  movq %rax, %rsi
0x3FFFFF7:  pushq %rbx
0x3FFFFF8:  call 0x404033
0x4000001:  popq %rbx
0x4000003:  cmpq %rbx, %rax
0x4000005:  jle 0x3FFFFF3
...
0x400031:  ret
...
```

BTB: cache for branch targets

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

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

```

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

BTB: cache for branch targets

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

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

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

indirect branch prediction

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

BTB can provide a prediction

but can do better with more context

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

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

backup slides

beyond 1-bit predictor

devote *more space* to storing history

main goal: rare exceptions don't immediately change prediction

example: branch taken 99% of the time

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

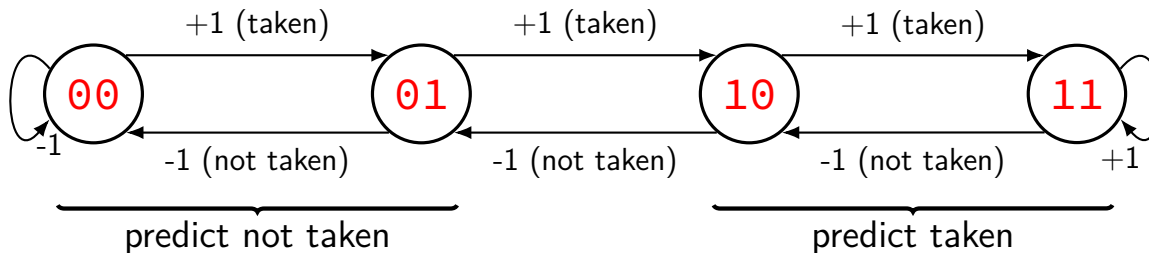
- 1% when branch not taken

- 1% of taken branches right after branch not taken

new predictor: wrong about 1% of the time

- 1% when branch not taken

2-bit saturating counter



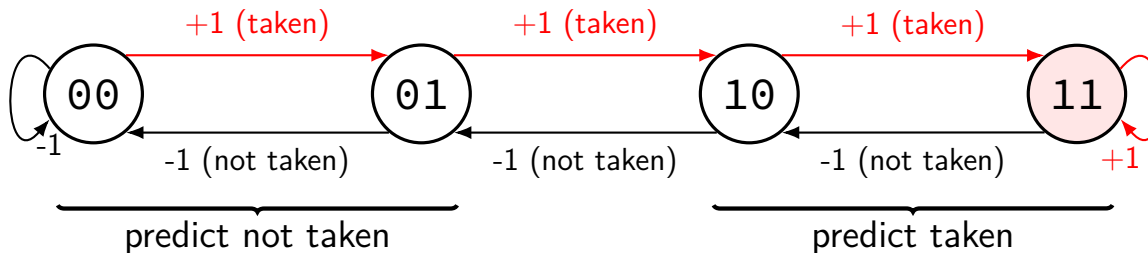
PC of branch

0x40042A

hash function

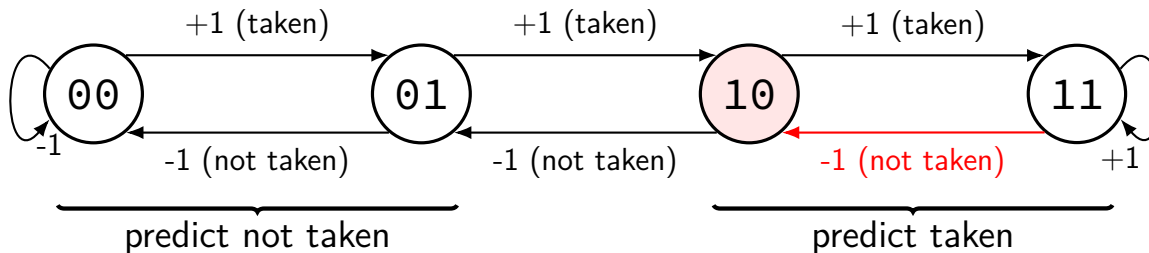
index	counter
0	11
1	01
2	11
...	...
14	10
15	00

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 \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)

PC of branch

0x40042A

hash function

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

4-iter loop: TTTN TTTN TTTN ...

idx recent
pattern

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

actual outcome

from commit(?) stage

???

convert to
prediction
???

prediction
to fetch stage

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

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

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

4-iter loop: TTTN TTTN TTTN ...

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

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

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

4-iter loop: TTTN TTTN TTTN ...

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

idx recent
pattern

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

actual outcome
from commit(?) stage

???

convert to
prediction
???

prediction
to fetch stage

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

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

idx recent
pattern

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

actual outcome

from commit(?) stage

???

convert to
prediction
???

prediction
to fetch stage

4-iter loop: TTTN TTTN TTTN ...

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

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

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

idx recent
pattern

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

actual outcome

from commit(?) stage

???

convert to
prediction
???

prediction
to fetch stage

4-iter loop: TTTN TTTN TTTN ...

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

local pattern predictor (incomplete)

PC of branch

0x40042A

hash function

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

idx recent
pattern

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

actual outcome

from commit(?) stage

???

convert to
prediction

???

prediction
to fetch stage

4-iter loop: TTTN TTTN TTTN ...

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

TTNTTTTT

predict T? loop with many iterations

(NTTTTTTTNTTTTTTTNTTTTTT...)

predict T? if statement mostly taken

(TTTTNTTNTTTTTTTTTTTNTTTT...)

predict N? loop with 5 iterations

(NTTTTNTTTTNTTTTNTTTTNTT...)

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

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

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

pattern

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

counter

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

prediction
to fetch sta

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

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

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

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

pattern

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

counter

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

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

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

pattern

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

counter

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

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

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

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

actual outcome
from commit(?) stage

pattern

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

counter

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

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

prediction
to fetch sta

history of history

PC of branch

0x40042A

hash

idx recent
pattern

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

actual outcome
from commit(?) stage

pattern

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

counter

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

prediction
to fetch sta

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

PC of branch

0x40042A

hash

idx recent
pattern

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

actual outcome
from commit(?) stage

pattern

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

counter

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

prediction
to fetch sta

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

PC of branch

actual outcome
from commit(?) stage

0x40042A

hash

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

pattern

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

counter

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

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

prediction
to fetch sta

local patterns and collisions (1)

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

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

local patterns and collisions (1)

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

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

pattern: TNTNTNTNTNTNTNTNT...

actually no problem to predict!

local patterns and collisions (2)

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

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

local patterns and collisions (2)

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

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

pattern: TTNNTTNNTTNNTTNNTT

also no problem to predict!

local patterns and collisions (3)

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

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

better for prediction of branch 3 and 4

global history predictor: idea

one predictor idea: ignore the PC

just record taken/not-taken pattern for all branches

lookup in big table like for local patterns

global history predictor (1)

branch history register

NTTT

pat

NNNN

NNNT

...

NTTT

TNNN

TNNT

TNTN

...

TTTN

TTTT

counter

00

00

...

10

01

10

11

...

10

11

outcome
from
commit(?)

prediction
to fetch stage

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

outcome
from
commit(?)

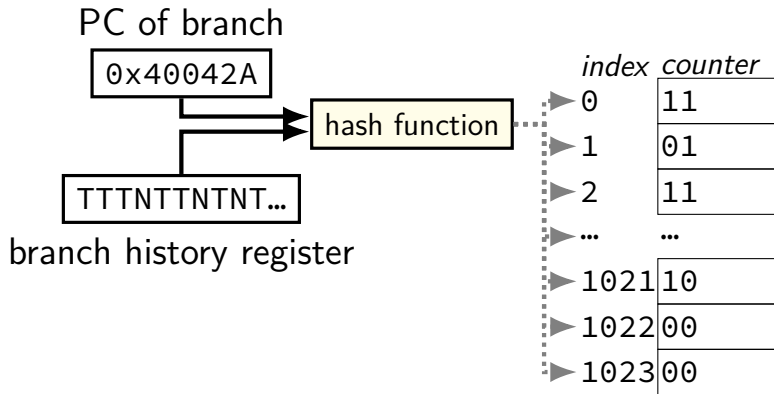
prediction
to fetch stage

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		NNTT
2/loop	TTTT			taken		TTTT

correlating predictor

global history *and* local info good together

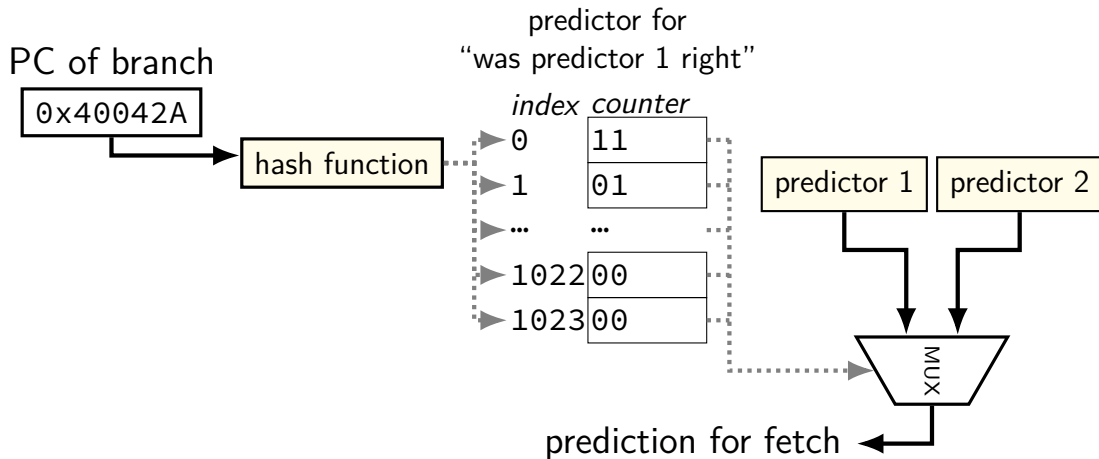
one idea: **combine history register + PC** (“gshare”)



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

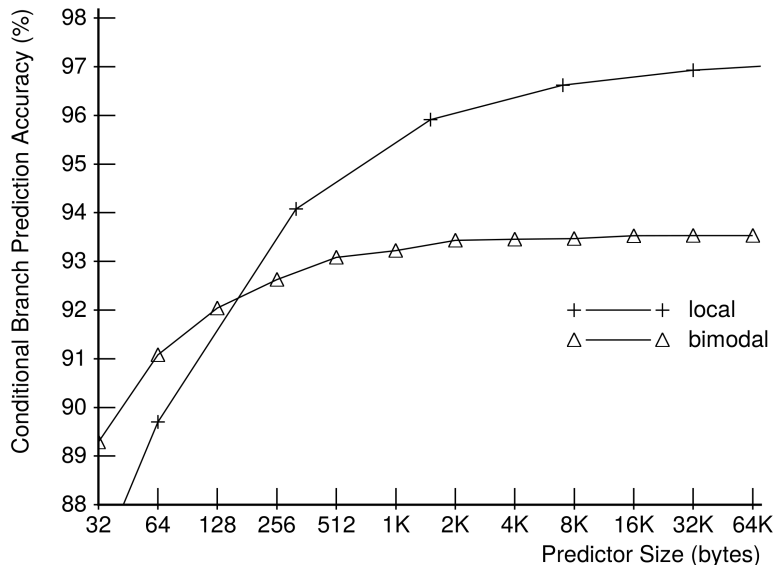
(not representative of modern workloads?)

rate for conditional branches on benchmark

variable table sizes

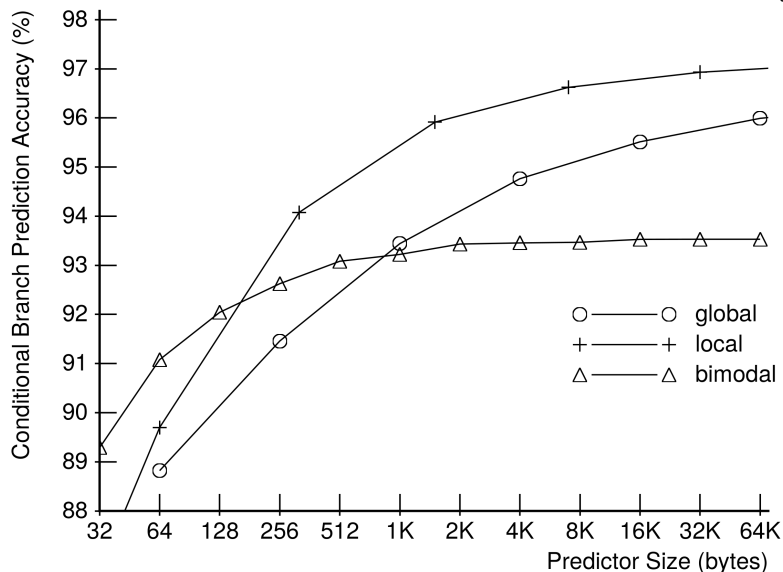
2-bit ctr + local history

from McFarling, "Combining Branch Predictors" (1993)



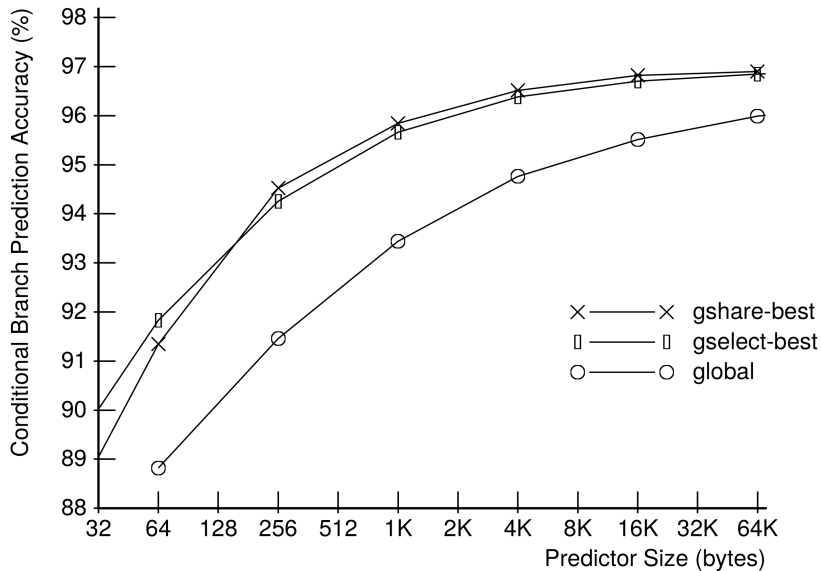
2-bit (bimodal) + local + global hist

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