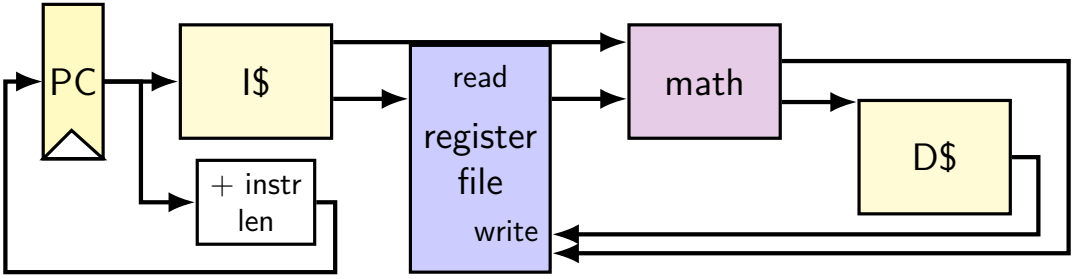
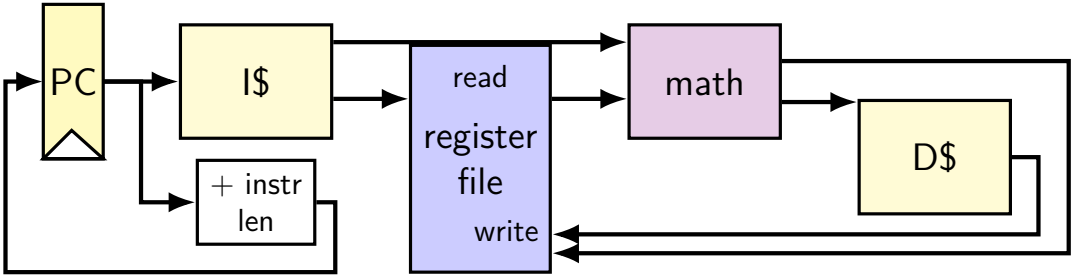
simple CPU



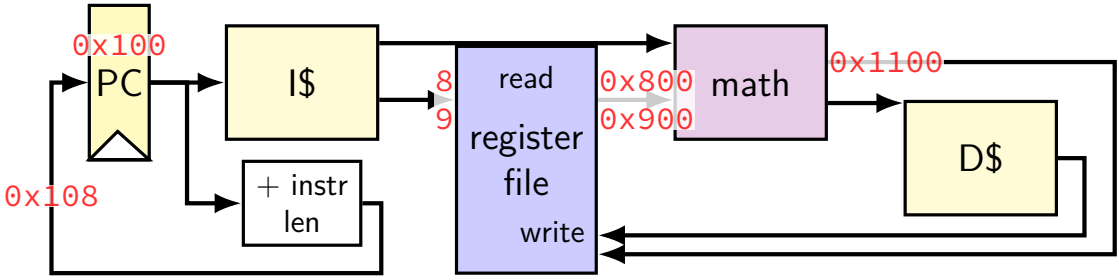
running instructions




```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

```
…
%r8: 0x800
%r9: 0x900
%r10: 0x1000
%r11: 0x1100
…
```

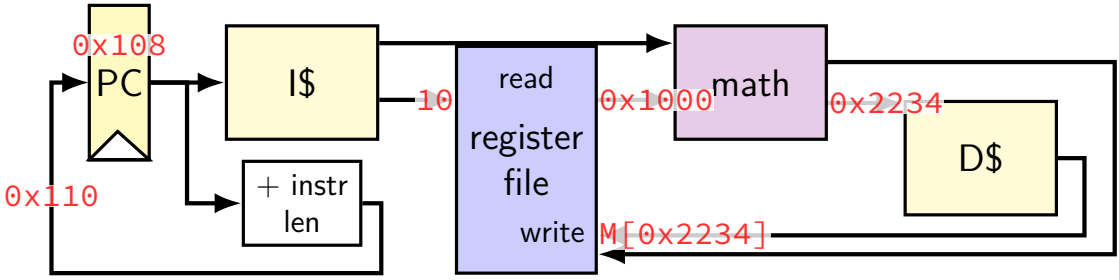
running instructions




```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

```
…
%r8: 0x800
%r9: 0x1700
%r10: 0x1000
%r11: 0x1100
…
```

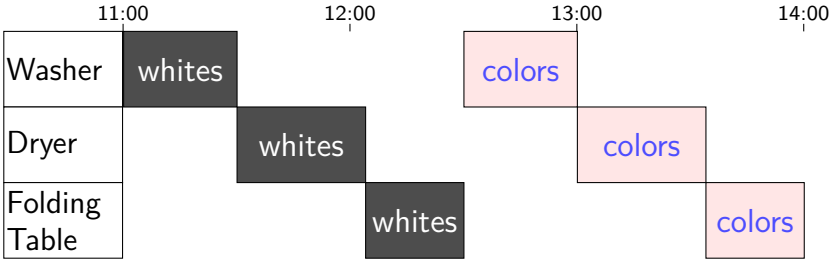
running instructions



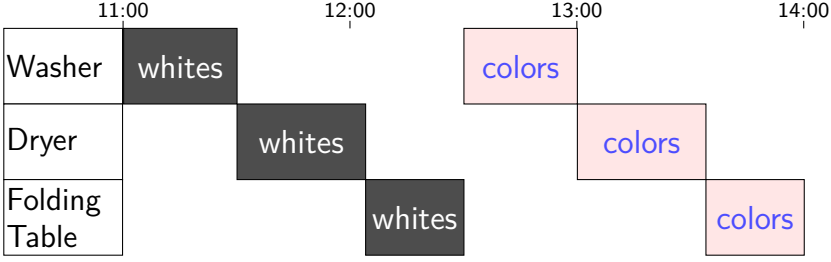

```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

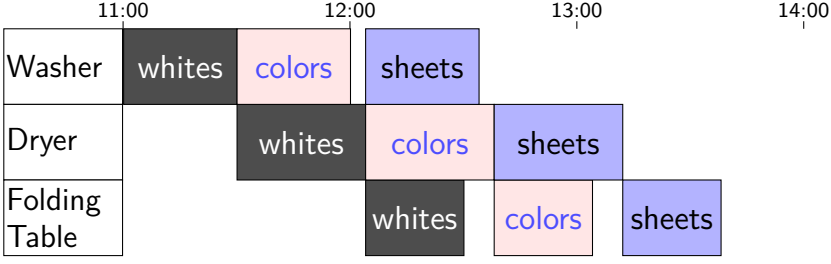
```
…
%r8: 0x800
%r9: 0x1700
%r10: 0x1000
%r11: M[0x2234]
…
```

Human pipeline: laundry

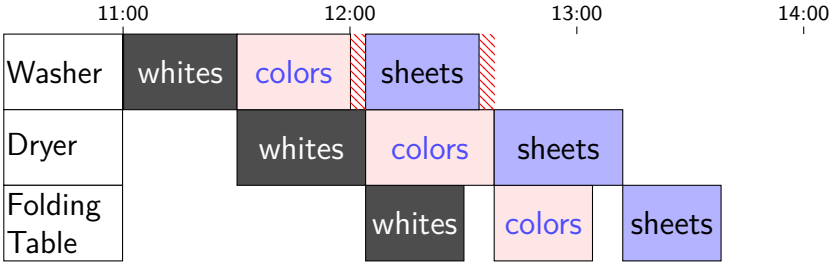


Human pipeline: laundry

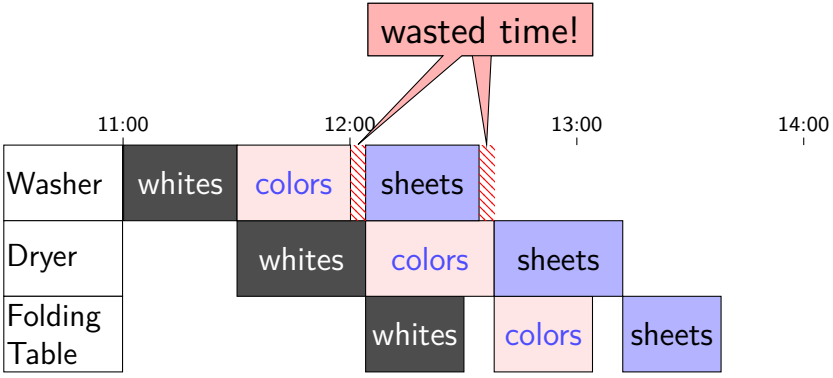




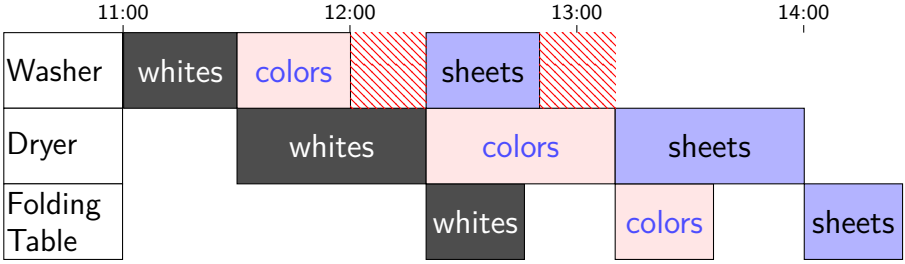
Waste (1)


11:00
12:00
13:00
14:00
Washer
Dryer
Folding Table
whites
colors
sheets
whites
colors
sheets
whites
colors
sheets

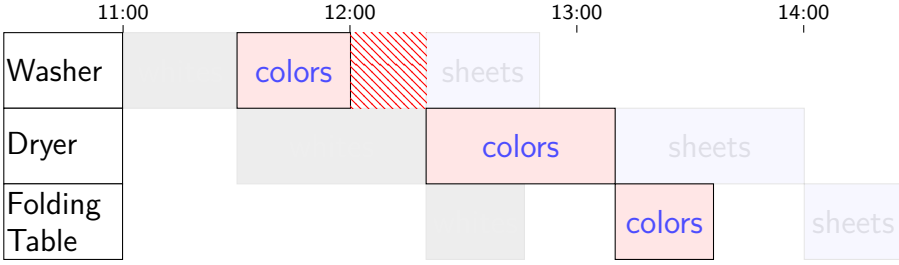
Waste (1)


wasted time!
11:00
12:00
13:00
14:00
Washer
whites
colors
sheets
Dryer
whites
colors
sheets
Folding Table
whites
colors
sheets

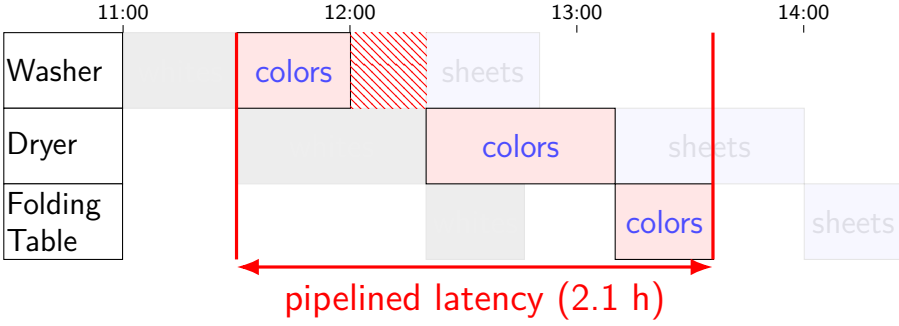
Waste (2)


11:00
12:00
13:00
14:00
Washer
Dryer
Folding Table
whites
colors
sheets
whites
colors
sheets
whites
colors
sheets

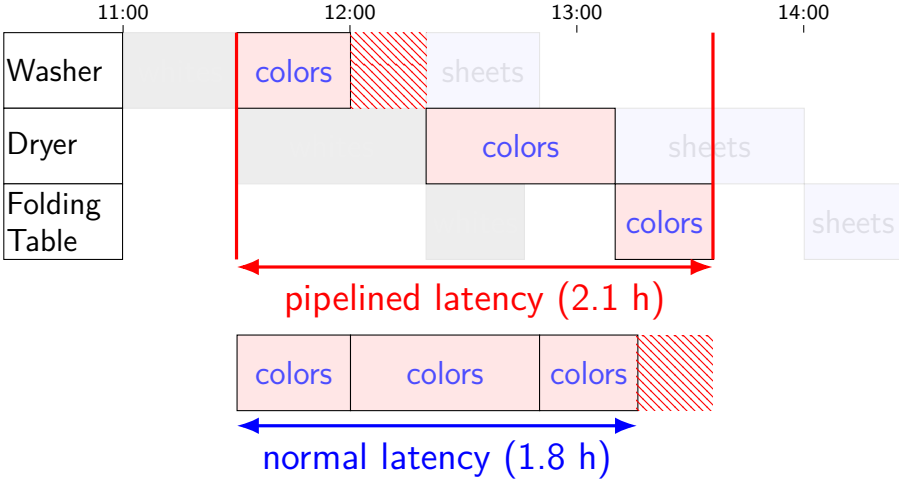
Latency — Time for One



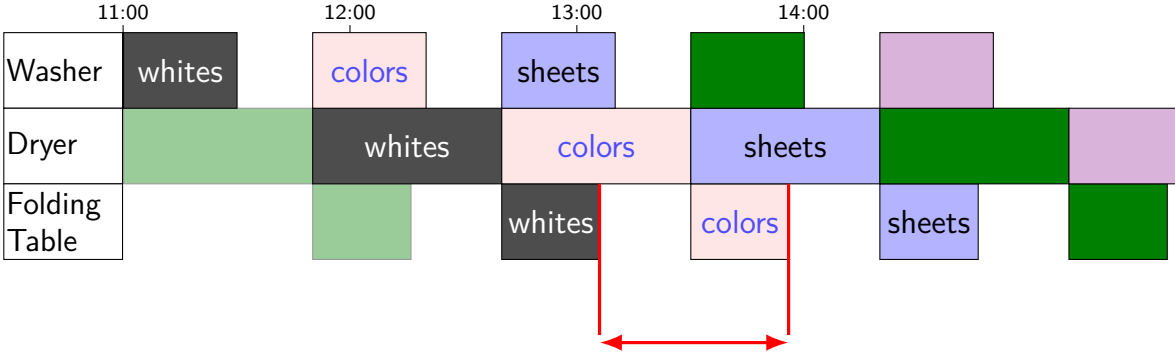
Latency — Time for One



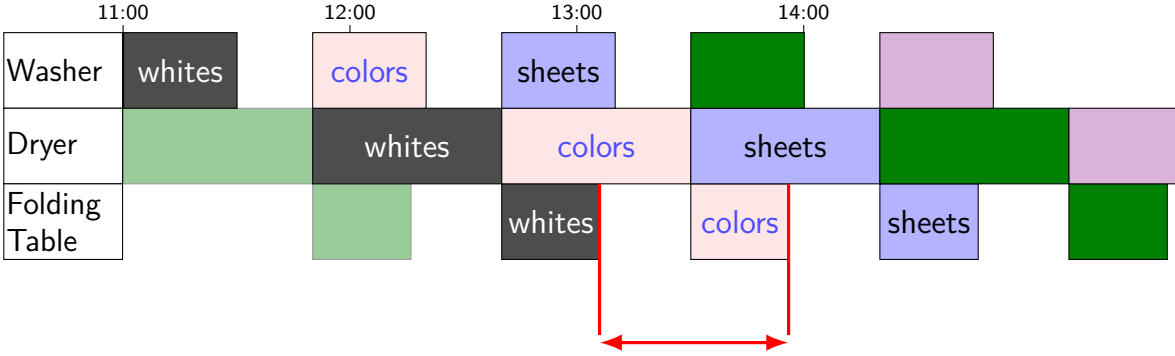
Latency — Time for One



Throughput — Rate of Many

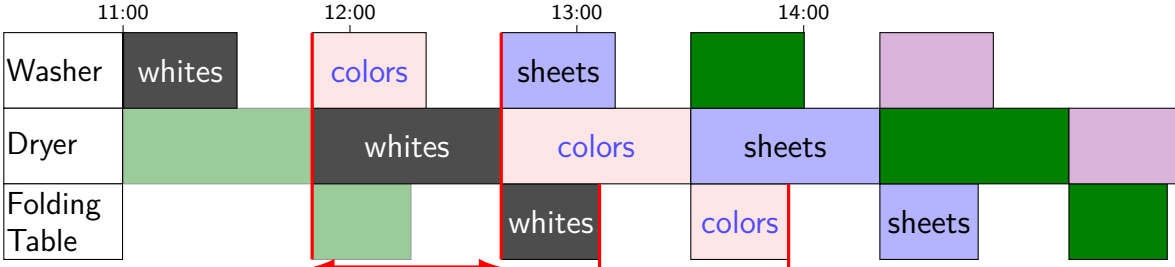


Throughput — Rate of Many



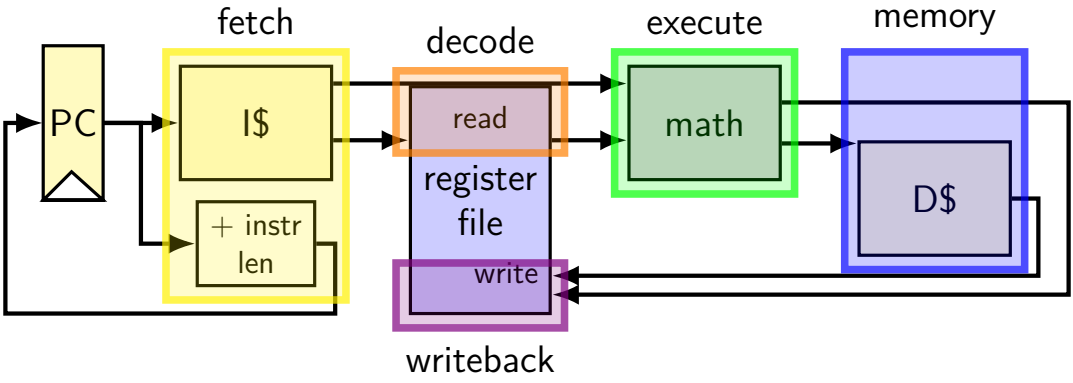
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many



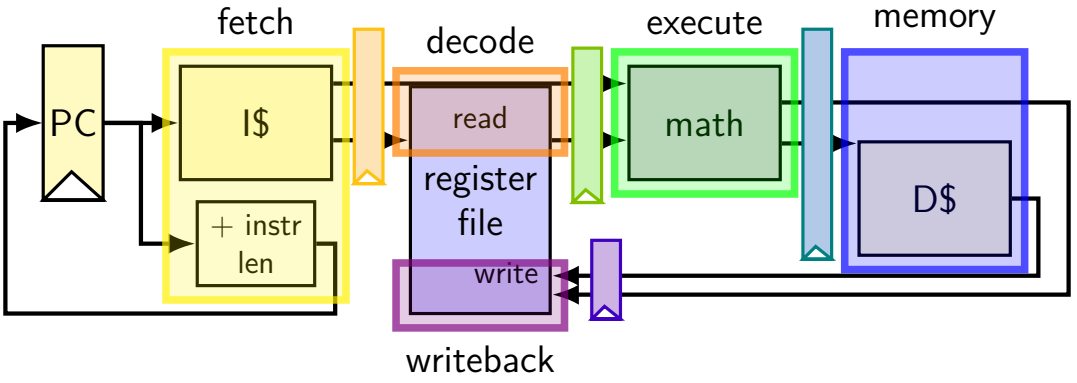
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

adding stages (one way)



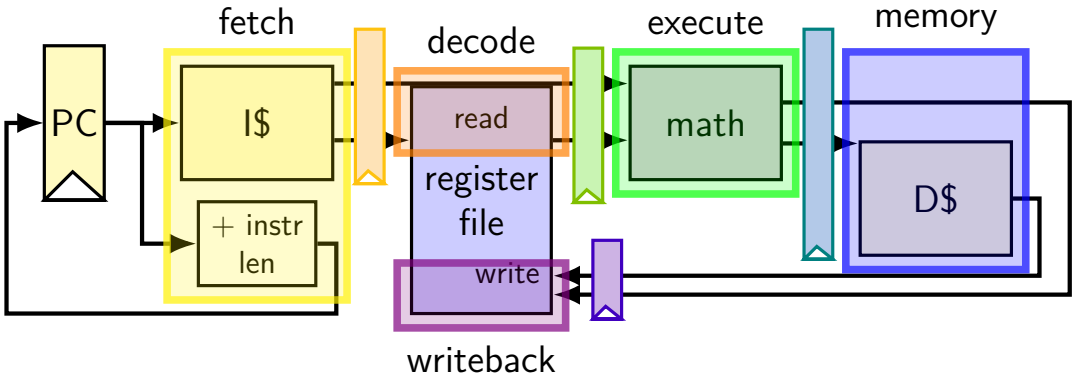
divide running instruction into steps
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

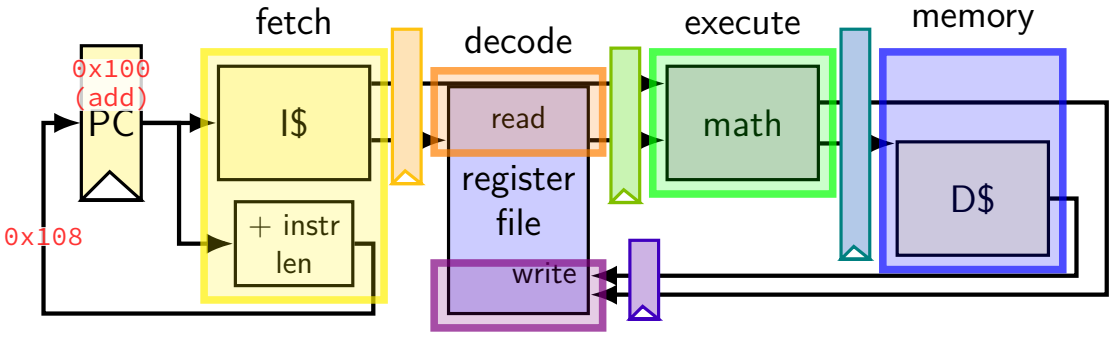


add ‘pipeline registers’ to hold values from instruction

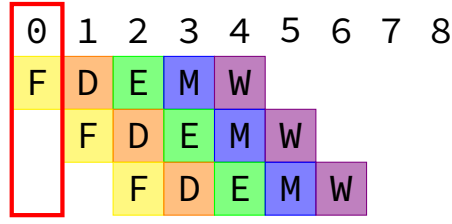
running some instructions



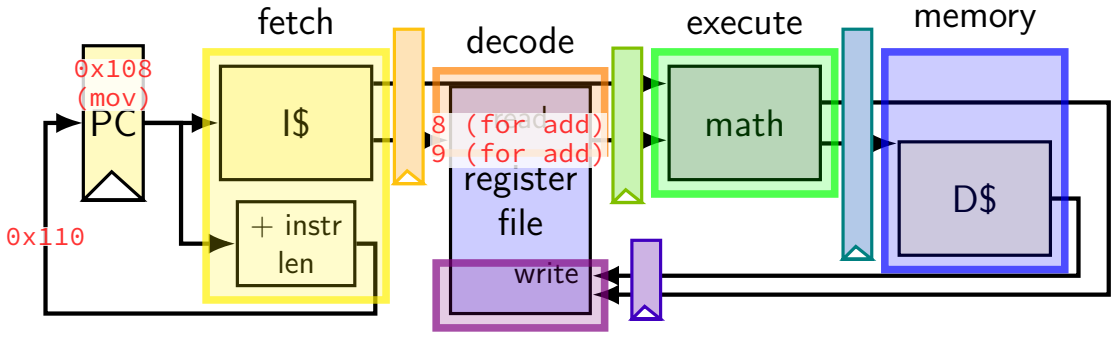
running some instructions




```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```



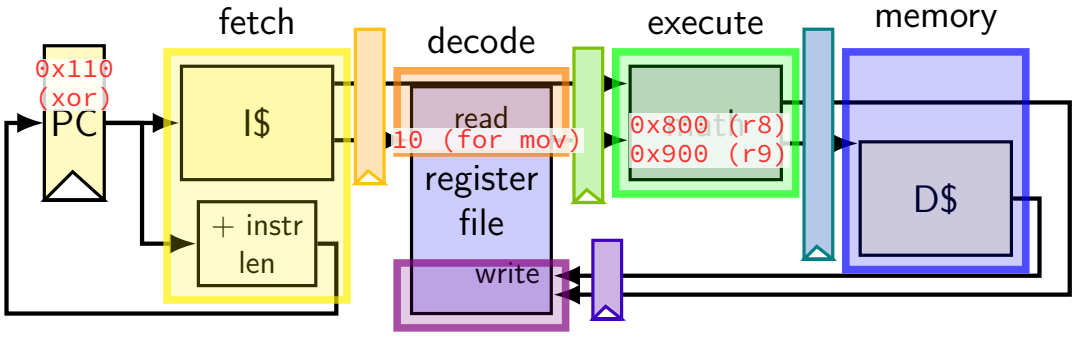
running some instructions




```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

cycle #	0	1	2	3	4	5	6	7	8
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		

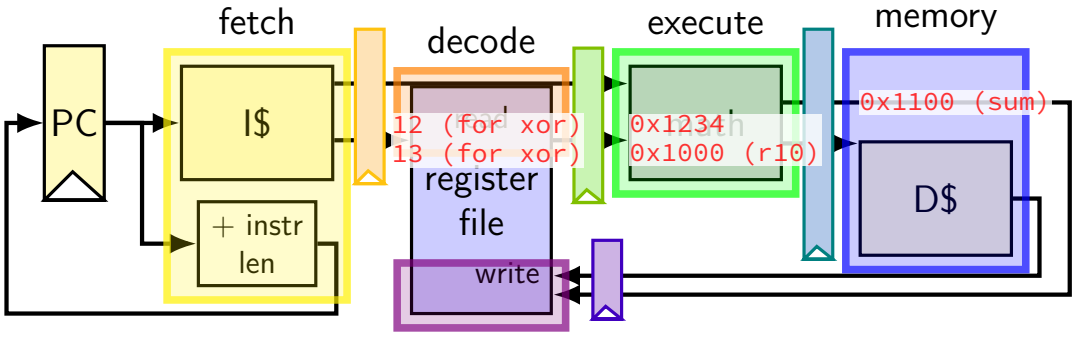
running some instructions




```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

cycle #	0	1	2	3	4	5	6	7	8
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		

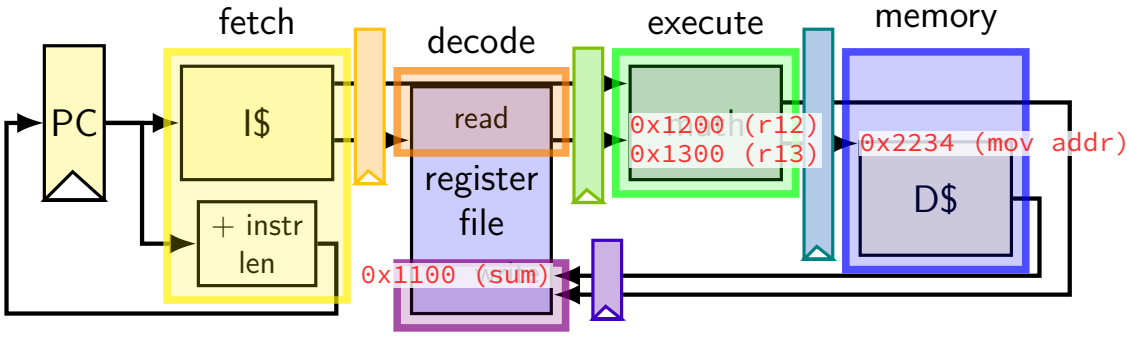
running some instructions



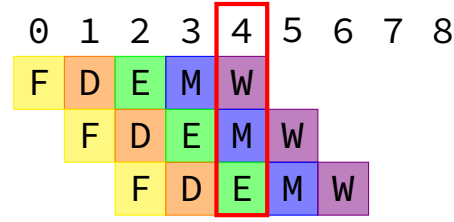

```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

cycle #	0	1	2	3	4	5	6	7	8
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		

running some instructions




```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

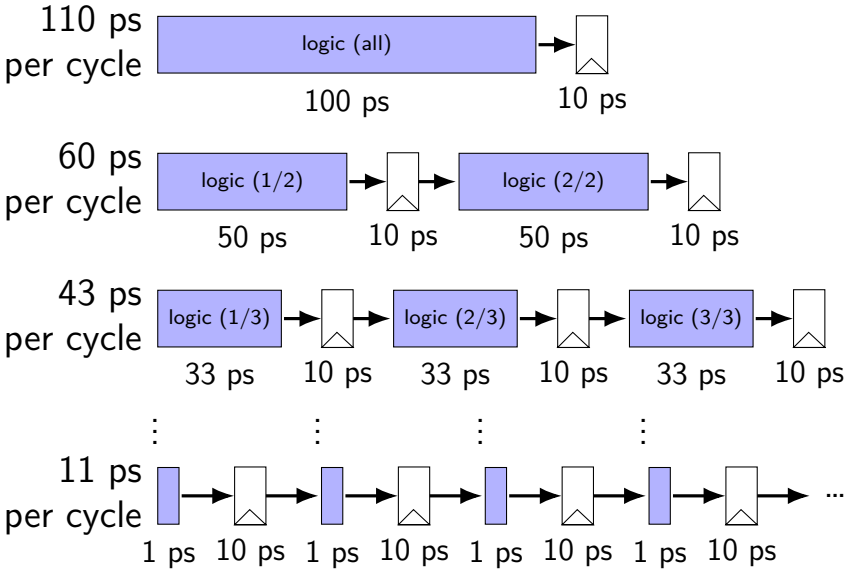


why registers?

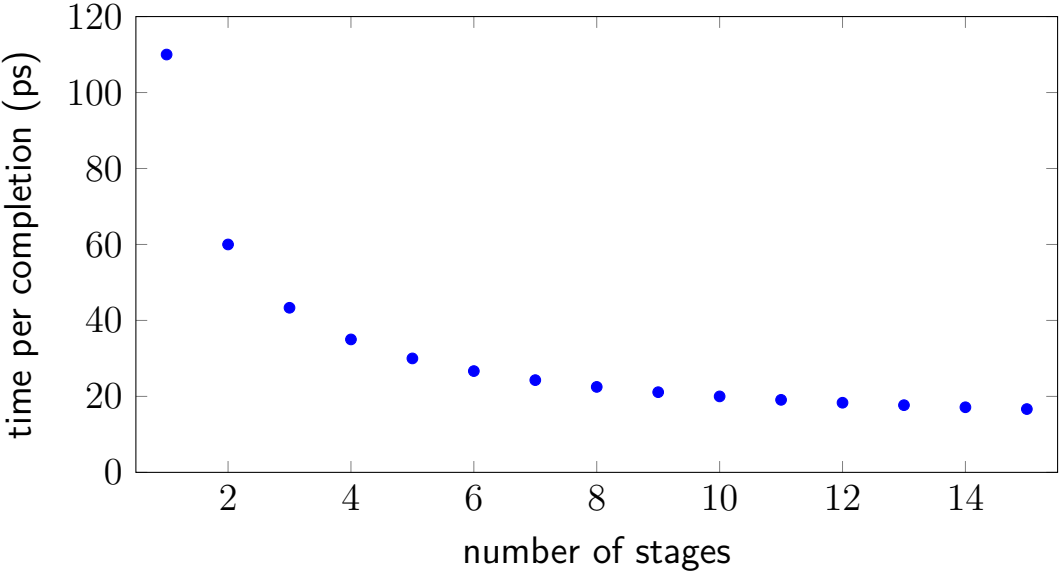
example: fetch/decode

need to store current instruction somewhere ...while fetching next one

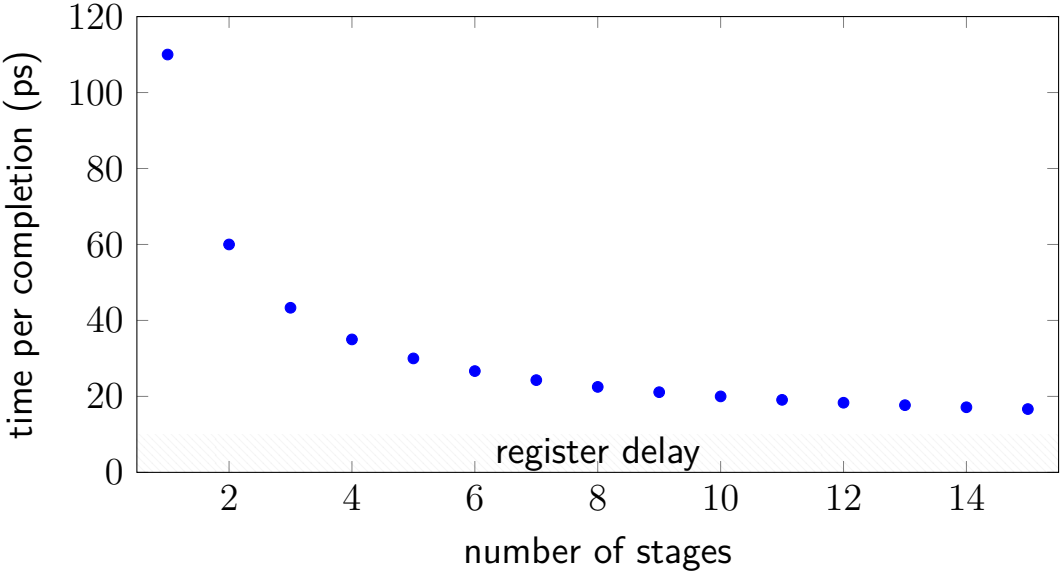
diminishing returns: register delays



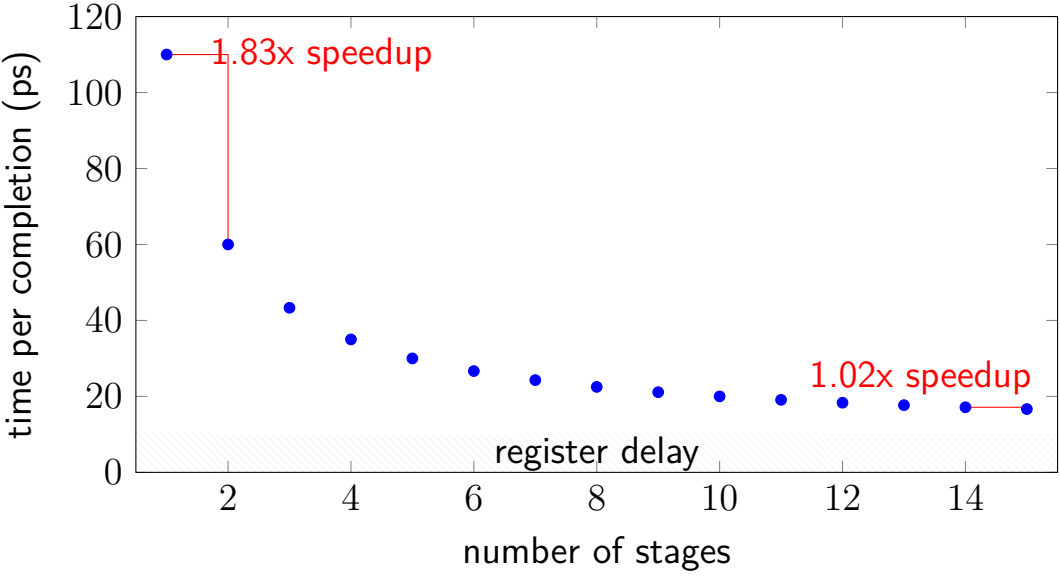
diminishing returns: register delays



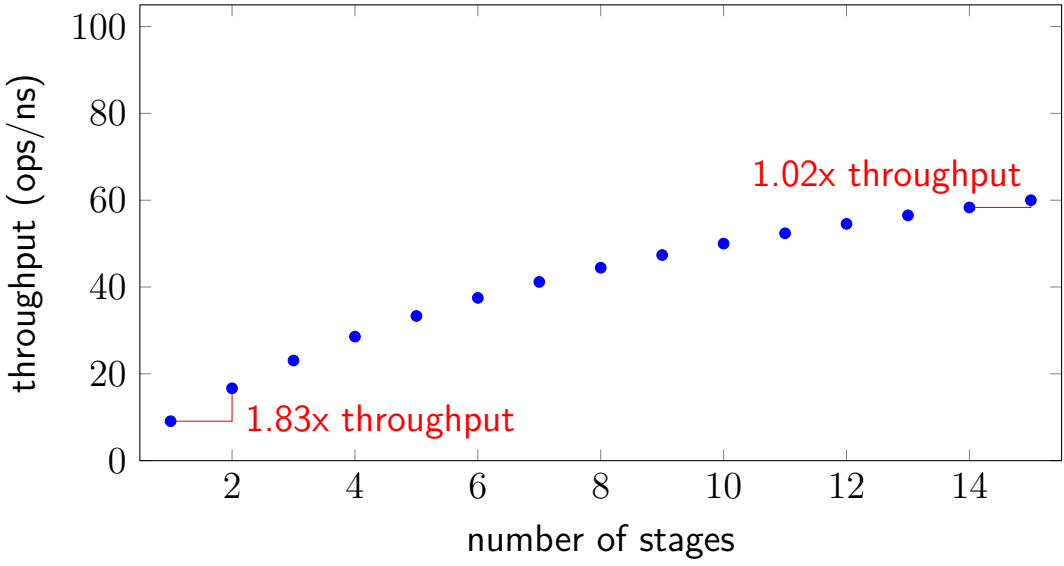
diminishing returns: register delays



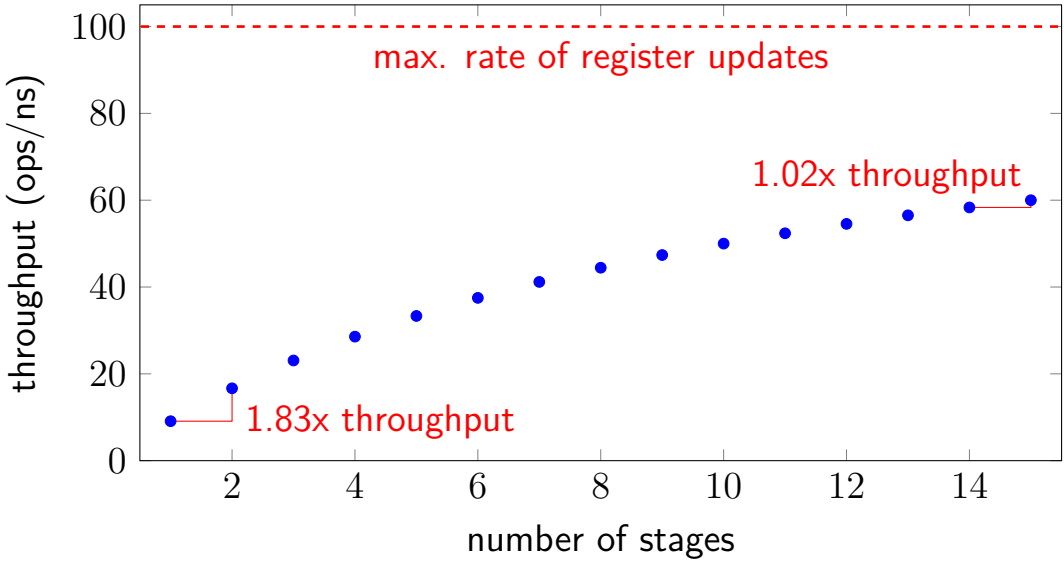
diminishing returns: register delays



diminishing returns: register delays



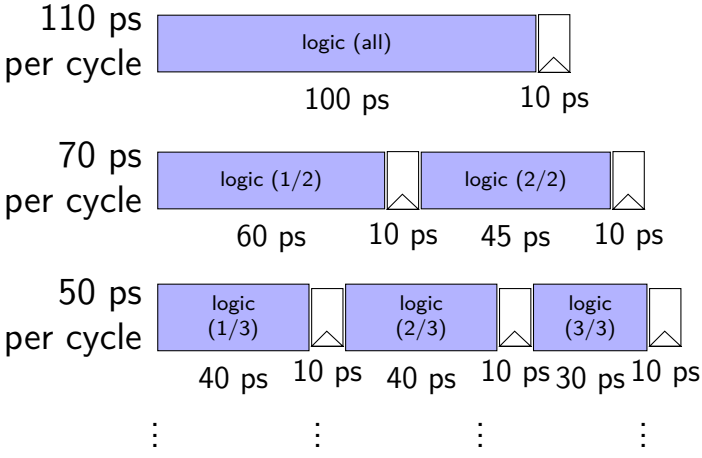
diminishing returns: register delays



diminishing returns: uneven split

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

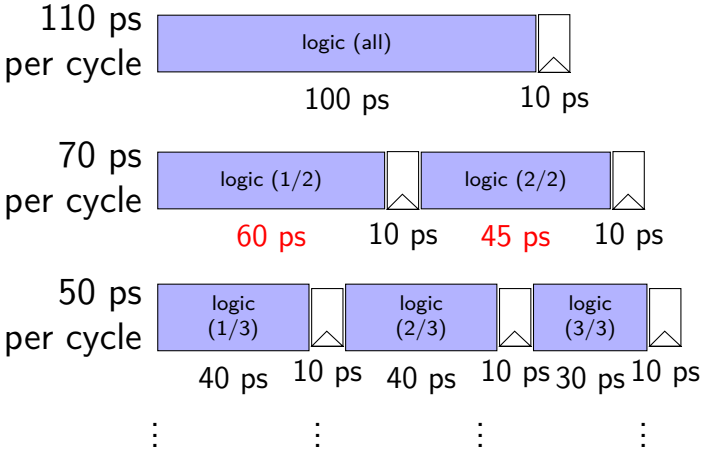
Probably not...



diminishing returns: uneven split

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

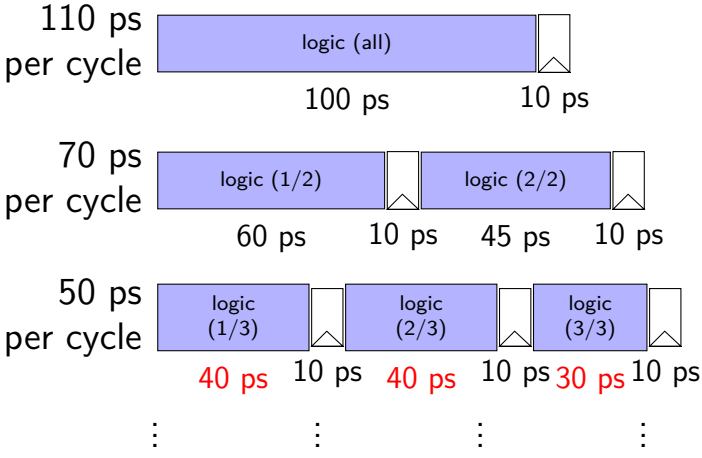
Probably not...



diminishing returns: uneven split

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

Probably not...



addq processor: data hazard

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

addq processor: data hazard

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

make it compiler's job

problem: recompile everytime processor changes?

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

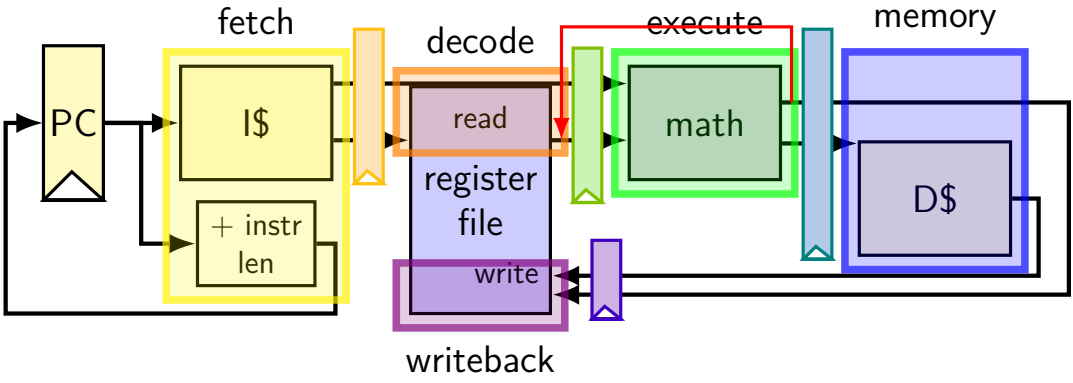
opportunity

```
// initially %r8 = 800,
//           %r9 = 900, etc.
0x0: addq %r8, %r9
0x2: addq %r9, %r8
...
```

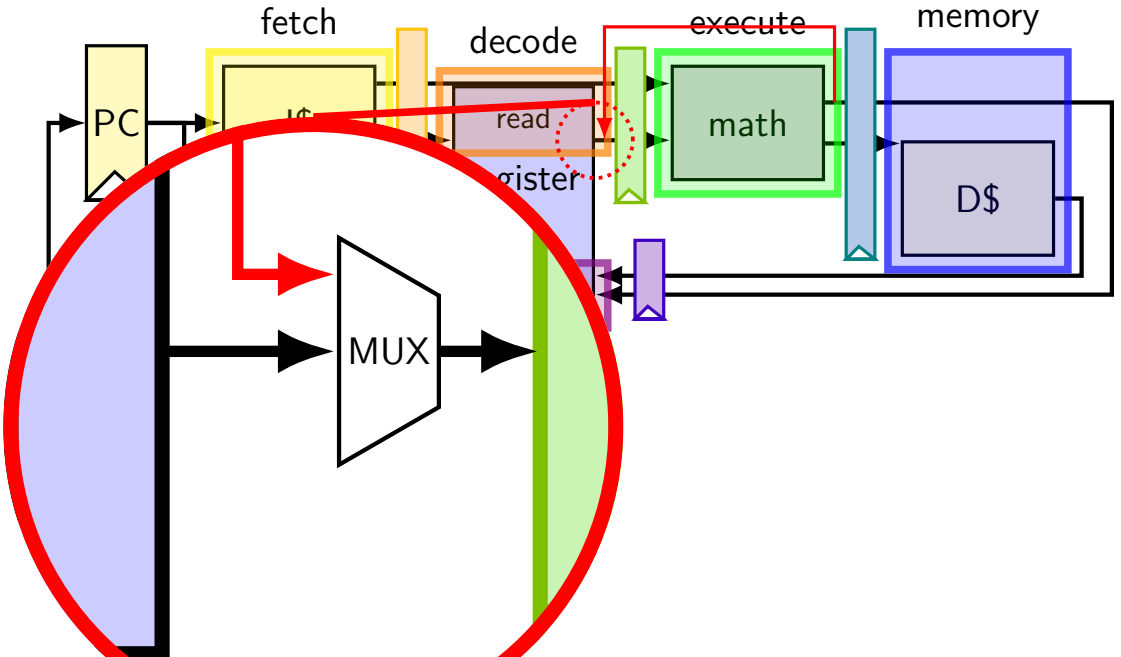
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



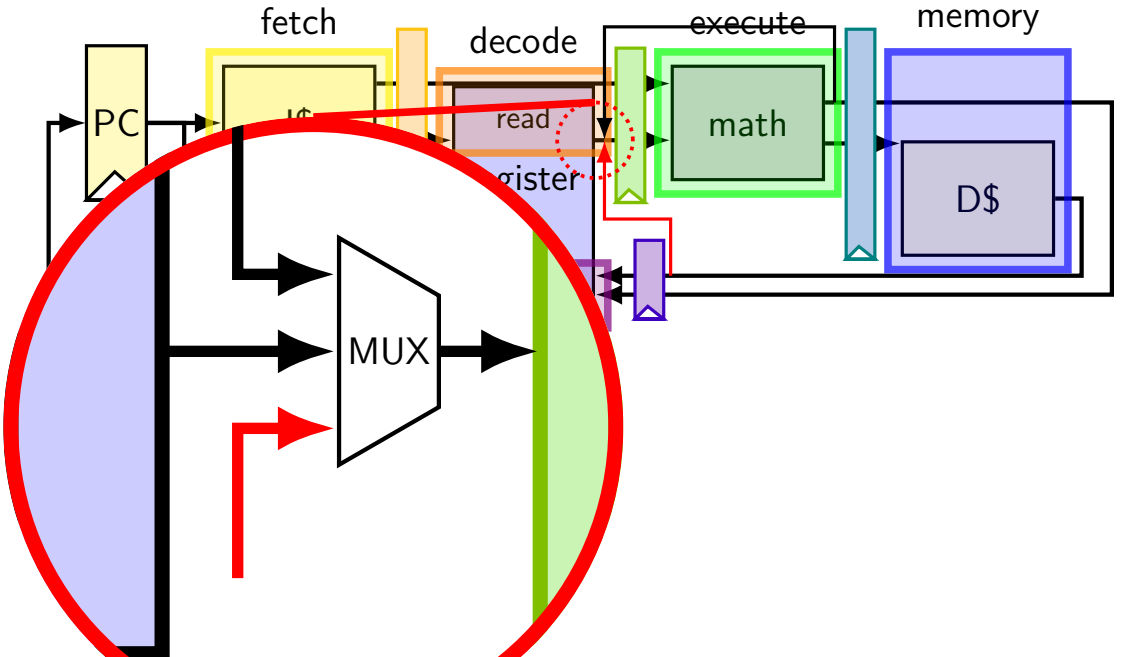
opportunity 2

```
// initially %r8 = 800,
//           %r9 = 900, etc.
0x0: addq %r8, %r9
0x2: nop
0x3: addq %r9, %r8
...
```

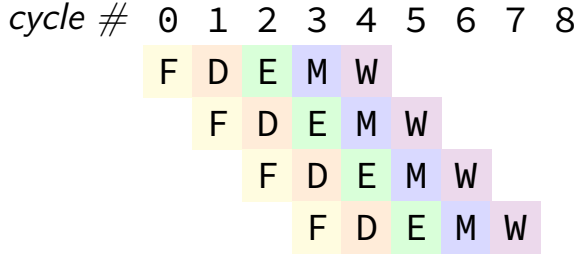
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2	0x3	---	---	800	900	9				
3		9	8	---	---	---	1700	9		
4				900	800	8	---	---	1700	9
5							1700	9	---	---
6									1700	9

should be 1700

exploiting the opportunity



exercise: forwarding paths



in subq, %r8 is _____ addq.

in xorq, %r9 is _____ addq.

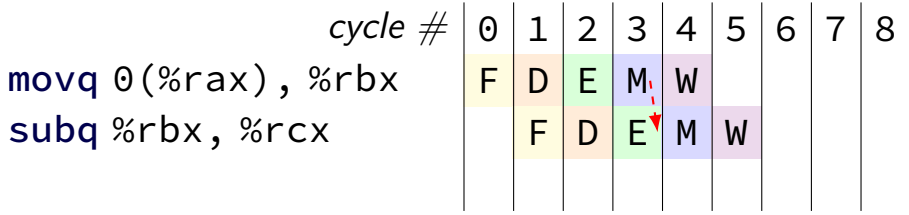
in andq, %r9 is _____ addq.

in andq, %r9 is _____ xorq.

A: not forwarded from

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

unsolved problem

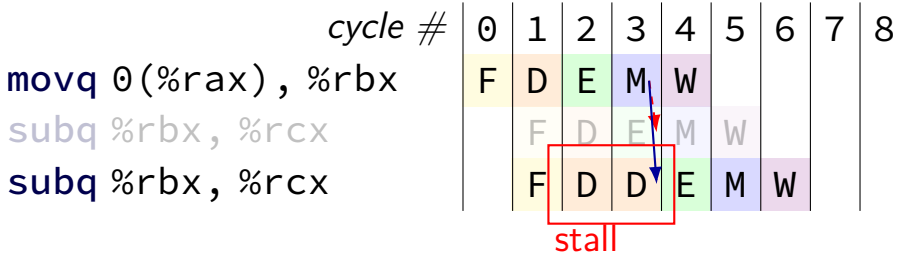


combine stalling and forwarding to resolve hazard

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)

control hazard

```
cmpq %r8, %r9
je   0xFFFF
addq %r10, %r11
```

	fetch	fetch→decode	decode→execute	execute→v	execute→writeback	···

cycle	PC	rA	rB	R[rA]	R[rB]	result	···	···	···
0	0x0								
1	0x2	8	9						
2	???	---	---	800	900				
2	???	---	---	---	---	less than			

control hazard

```
cmpq %r8, %r9
je   0xFFFF
addq %r10, %r11
```

	fetch	fetch→decode	decode→execute	execute→v	execute→writeback	···

cycle	PC	rA	rB	R[rA]	R[rB]	result	···	···	···
0	0x0								
1	0x2	8	9						
2	???	---	---	800	900				
2	???	---	---	---	---	less			

0xFFFF if R[8] = R[9], 0x12 otherwise

making guesses

```
        subq   %rcx, %rax
        jne    LABEL
        xorq   %r10, %r11
        xorq   %r12, %r13
        ...
LABEL:  addq   %r8, %r9
        rmmovq %r10, 0(%r11)
```

speculate: jne will goto LABEL

right: 2 cycles faster!

wrong: forget before execute finishes

when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

to "undo" instruction during fetch/decode:

forget everything in pipeline registers

jXX: speculating right

```
        subq %r8, %r8
        jne LABEL
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
        irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	irmovq	rmmovq	addq	jne (done)	OPq

jXX: speculating right

```
        subq %r8, %r8
        jne LABEL
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
        irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	j were waiting/nothing		
5	irmovq	rmmovq	addq	jne (done)	OPq

jXX: speculating wrong

```
        subq %r8, %r8
        jne LABEL
        xorq %r10, %r11
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

jXX: speculating wrong

```
        subq %r8, %r8
        jne LABEL
        xorq %r10, %r11
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne				
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

"squash" wrong guesses

jXX: speculating wrong

```
        subq %r8, %r8
        jne LABEL
        xorq %r10, %r11
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	j			
4	rmmovq [?]	addq [?]	jne (use 2?)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

fetch correct next instruction

static branch prediction

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

intuition: loops:

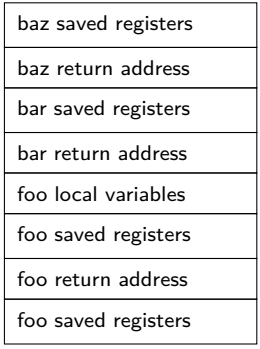
```
LOOP: ...
      ...
      je LOOP

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

predicting ret: extra copy of stack

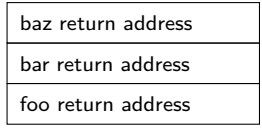
predicting ret — ministack in processor registers

push on ministack on call; pop on ret

ministack overflows? discard oldest, mispredict it later

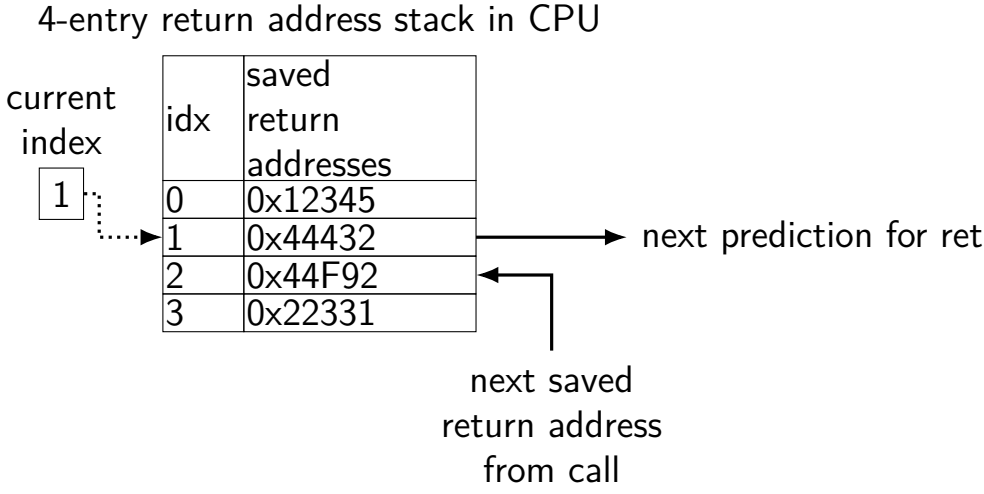


stack in memory



(partial?) stack
in CPU registers

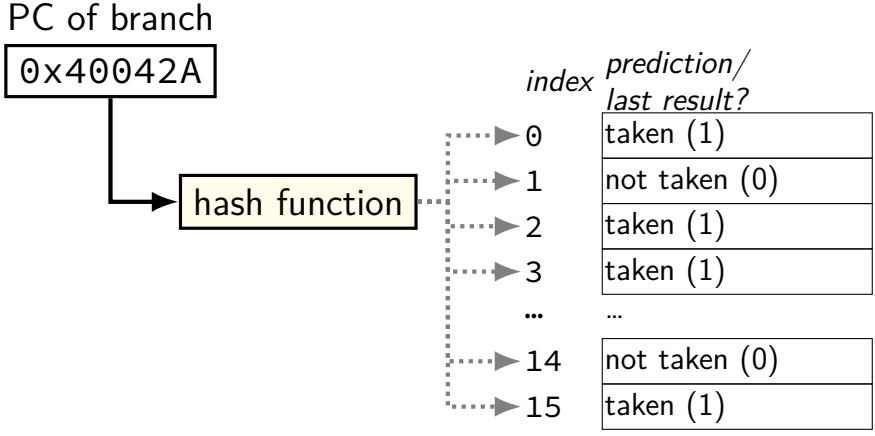
4-entry return address stack



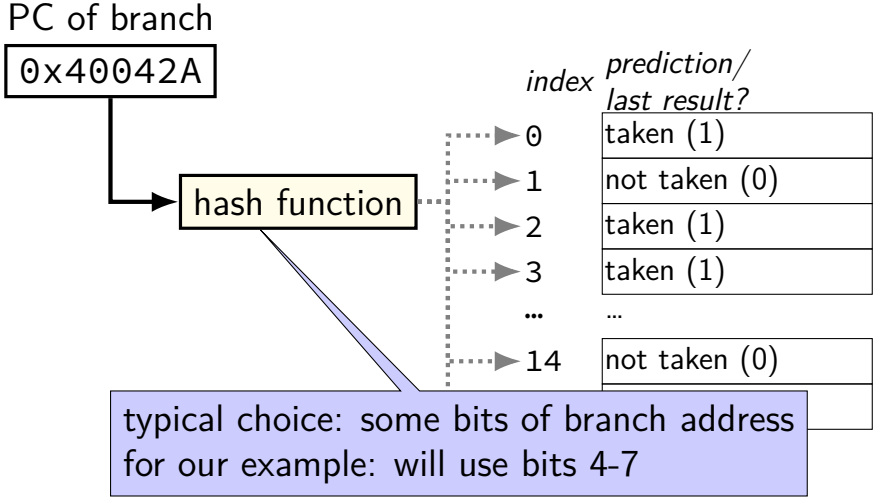
on `call`: increment index, save return address in that slot

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

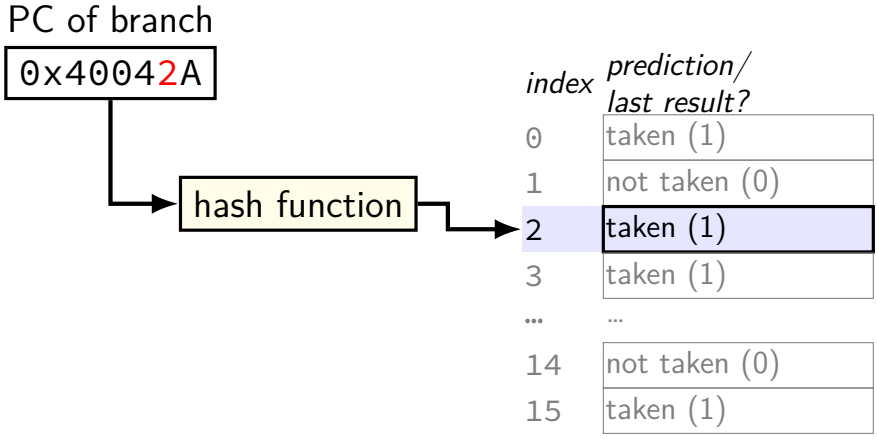
predict: repeat last



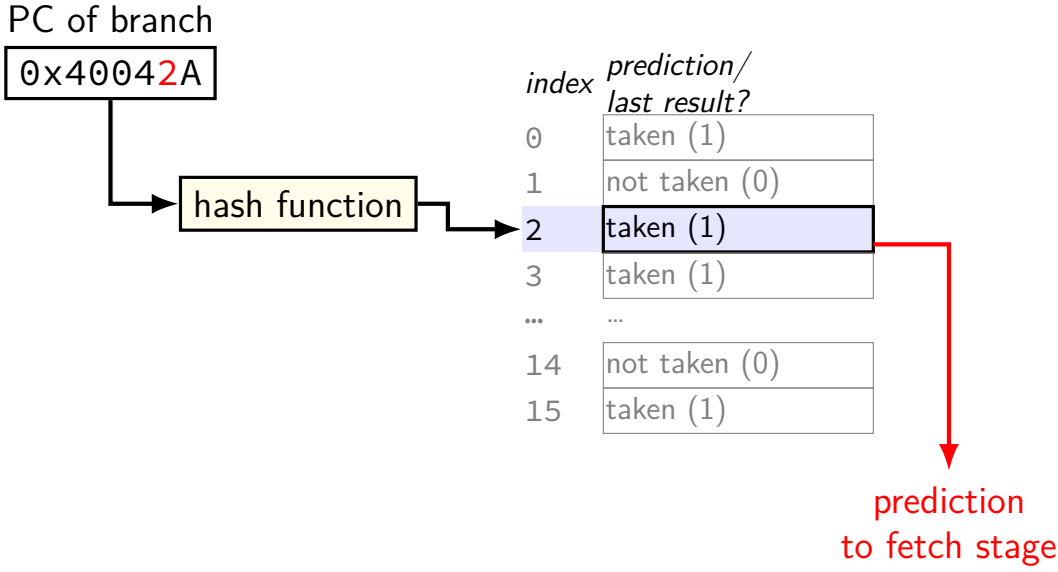
predict: repeat last



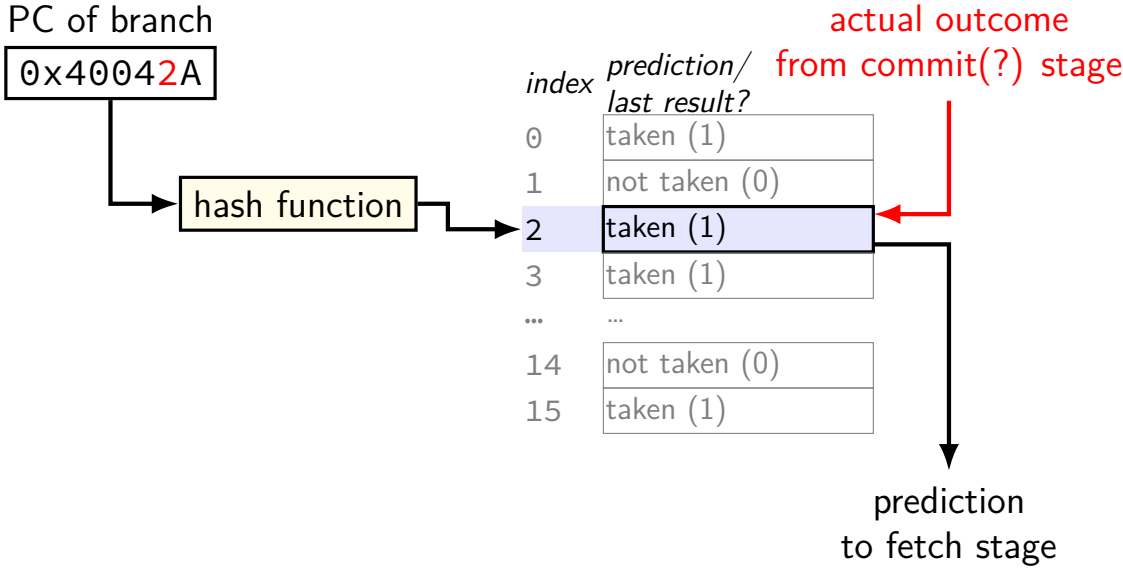
predict: repeat last



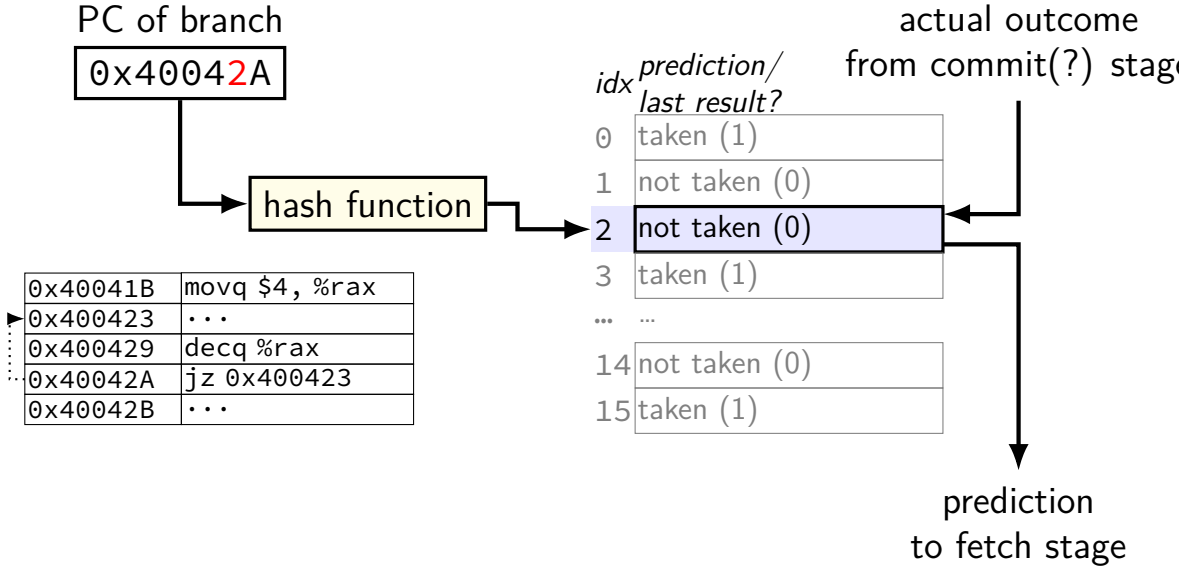
predict: repeat last



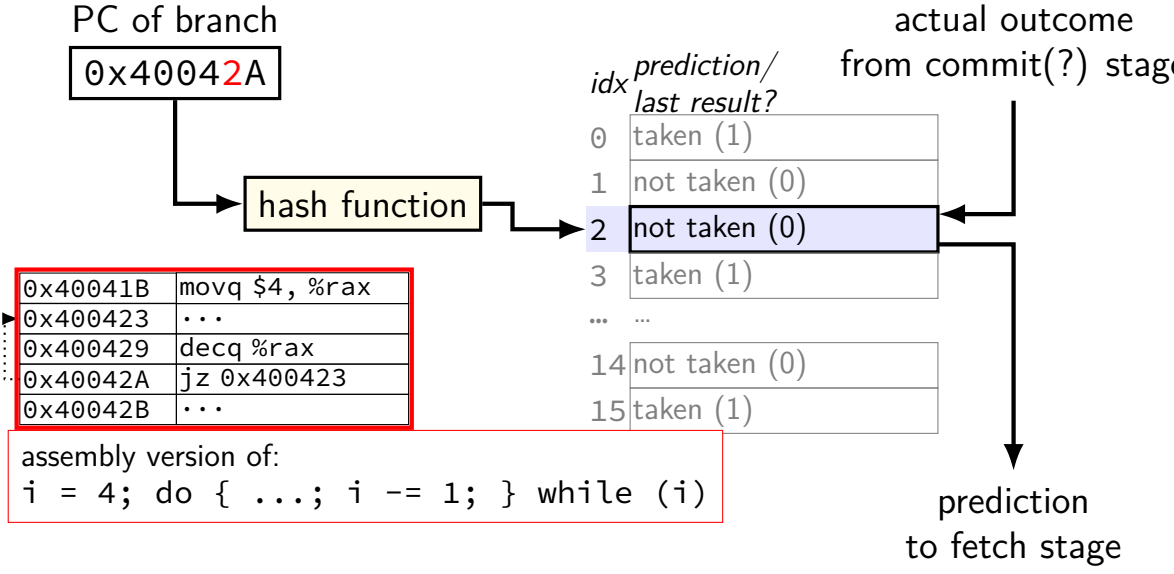
predict: repeat last



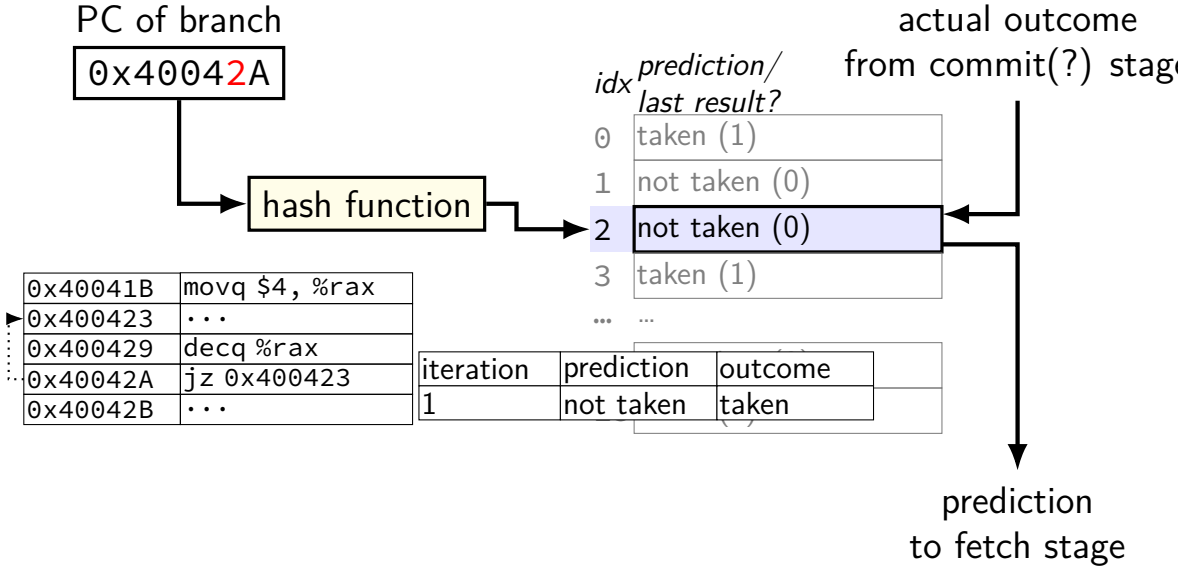
example


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 commit(?) stage
prediction
to fetch stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jz 0x400423
0x40042B ...

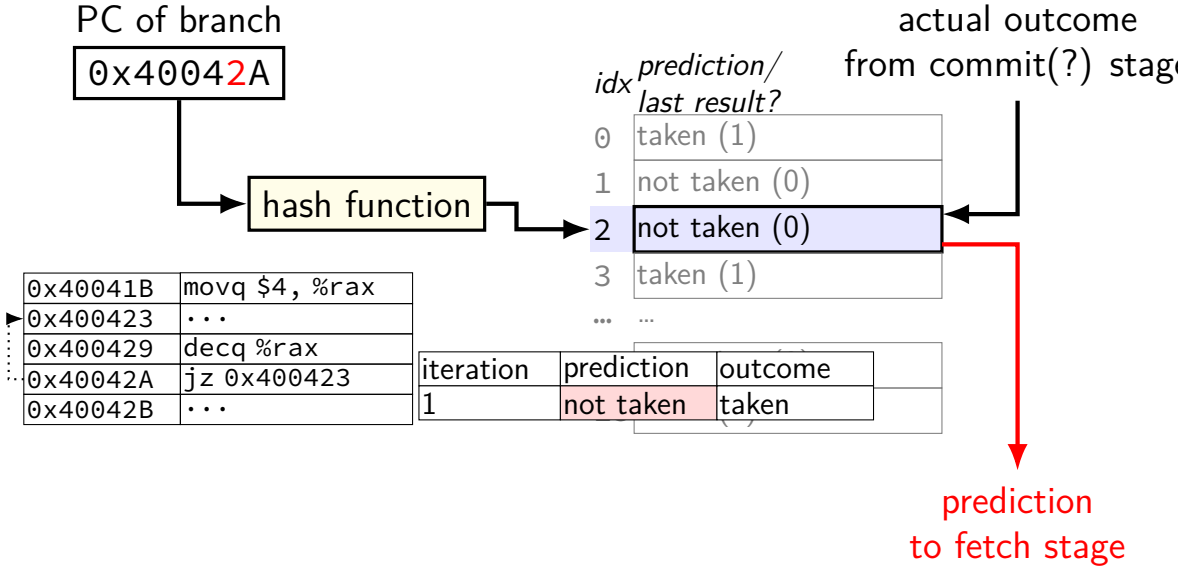
example


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 commit(?) stage
prediction
to fetch stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jz 0x400423
0x40042B ...
assembly version of:
i = 4; do { ...; i -= 1; } while (i)

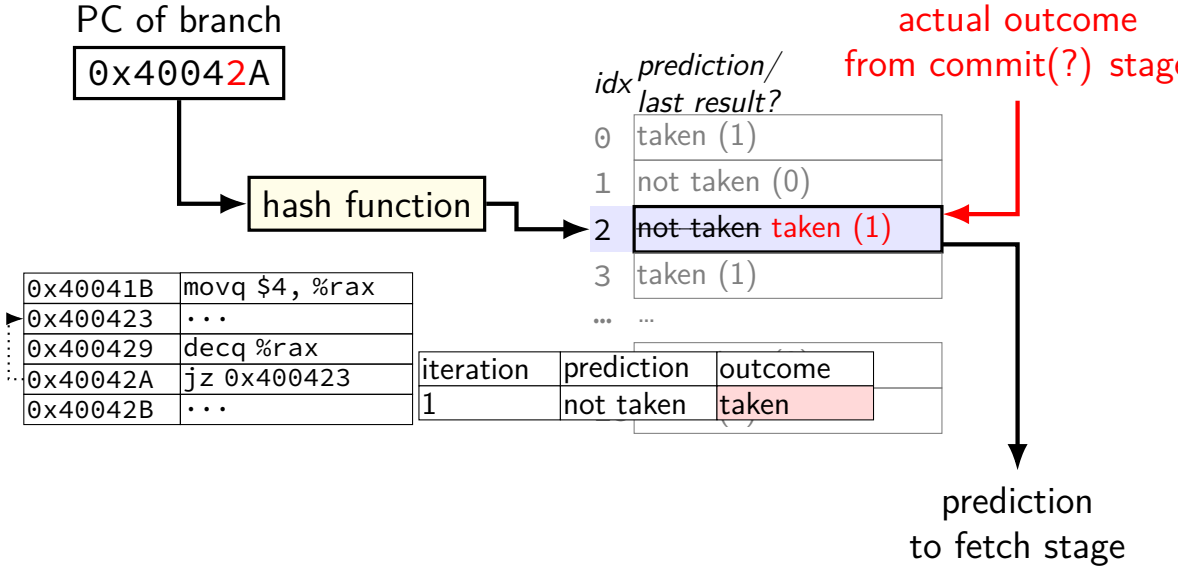
example


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

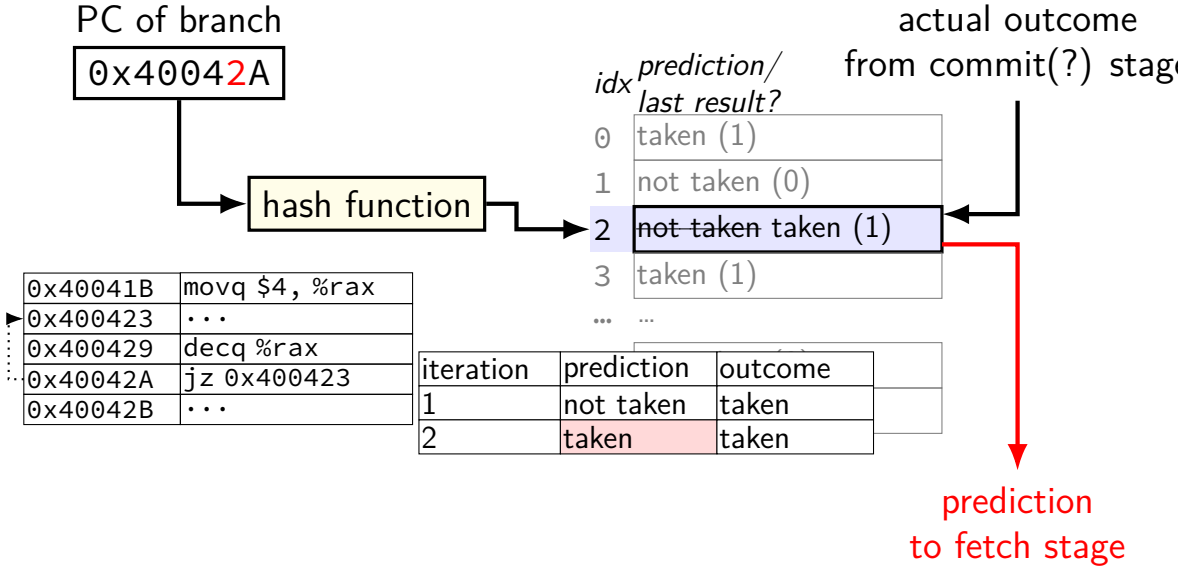
example


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

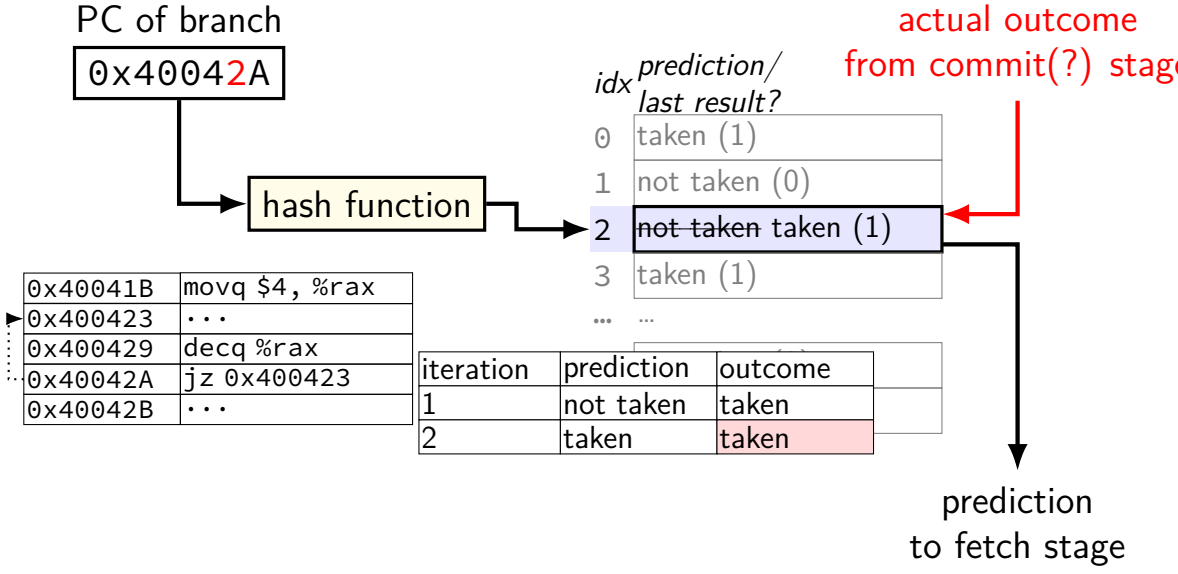
example


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 commit(?) stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jz 0x400423
0x40042B ...
iteration prediction outcome
1 not taken taken
prediction
to fetch stage

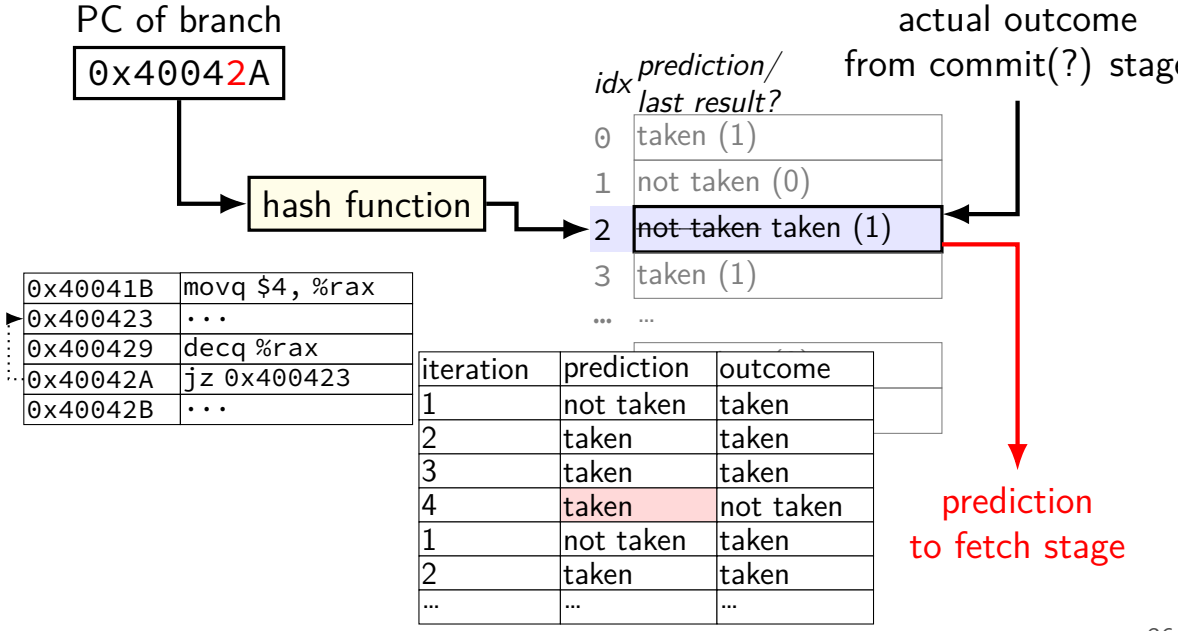
example


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 commit(?) stage
0x40041B
movq $4, %rax
0x400423
...
0x400429
decq %rax
0x40042A
jz 0x400423
0x40042B
...
iteration
prediction
outcome
1
not taken
taken
2
taken
taken
prediction
to fetch stage

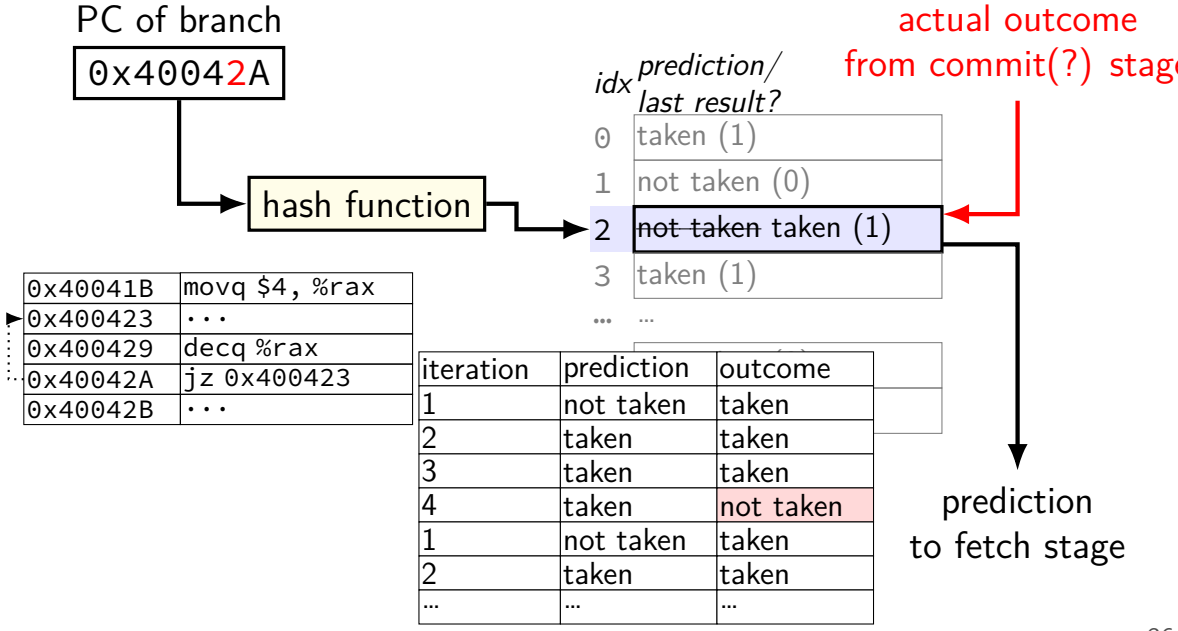
example


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 commit(?) stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jz 0x400423
0x40042B ...
iteration prediction outcome
1 not taken taken
2 taken taken
prediction
to fetch stage

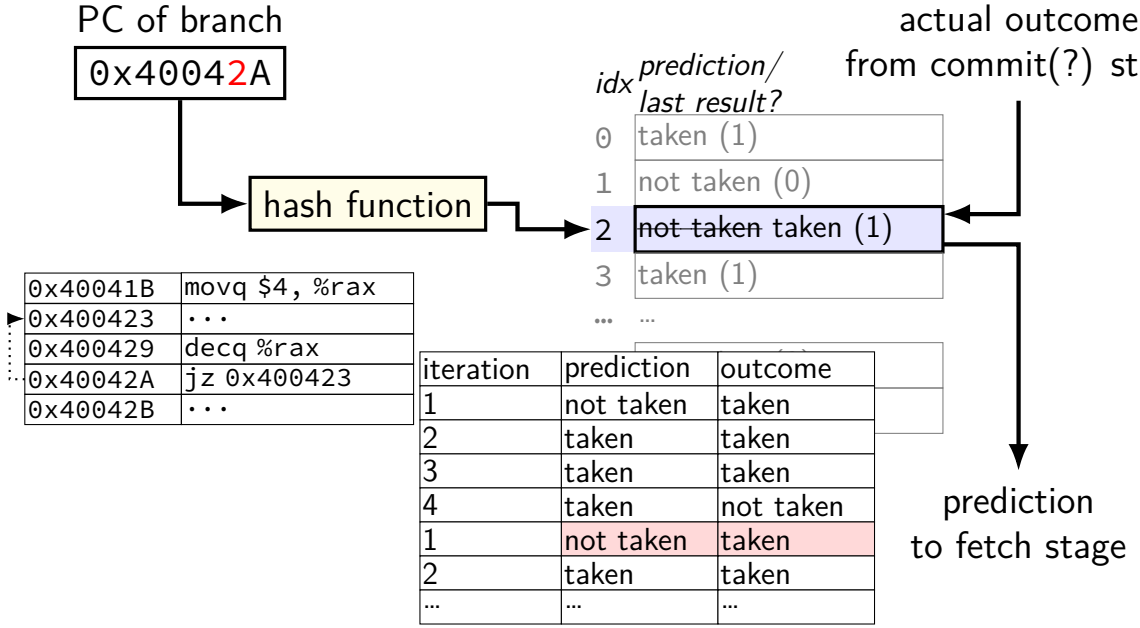
example


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 commit(?) stage
prediction to fetch stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jz 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 commit(?) stage
prediction
to fetch stage
0x40041B
movq $4, %rax
0x400423
...
0x400429
decq %rax
0x40042A
jz 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 commit(?) stage
prediction to fetch stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jz 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* smaller 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

later: we'll find a way to do better

exercise

use 1-bit predictor on this loop

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

what is the conditional branch misprediction rate?

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

backup slides

backup slides

exercise: forwarding paths (2)

cycle # `0 1 2 3 4 5 6 7 8`

```
addq %r8, %r9
subq %r8, %r9
ret (goes to andq)
andq %r10, %r9
```

in subq, %r8 is _____ addq.

in subq, %r9 is _____ addq.

in andq, %r9 is _____ subq.

in andq, %r9 is _____ addq.

A: not forwarded from

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