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

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 really like the analogy you used at the end of last class with the washing machine. For a student like me who doesn’t always intuitively get the examples/motivation behind what you speak about in class, the analogy really helped me understand the purpose of overlapping steps and the terminology such as latency and throughput.”

that analogy’s courtesy of Patterson and Hennessy, *Computer Organiztion and Design*

anonymous feedback (3)

“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.”

quiz Q1 (0a)

student $\rightarrow$ registrar

no protection (attacker can know/replace)

- name (copy also public-key encrypted, but attacker can still replace)
- number of items in list of requested courses

hashed + signed (attacker can check guesses but cannot replace/add)

- course identities being registered for (copy also public-key encrypted, but attacker can still replace)

public-key encrypted (attacker cannot read, but can replace)

- course section requested

quiz Q1 (0b)

registrar $\rightarrow$ student

signed (attacker cannot replace)

- name (known from other direction of protocol)
- time

public-key encrypted (attacker can replace)

- list of courses registered for

quiz Q1 (1)

re: authenticity

attacker can't forge signatures, but...

- student signature only over identity of course (not section)
- registrar signature only over open time + name

yes C: tamper message to omit classes

no D: cannot get student signature

yes E: signature doesn't protect section requested

yes F+G: can make up own list of courses, encrypt to student

quiz Q1 (2)

re: confidentiality

section encrypted to registrar, course numbers not

yes A, no B

quiz Q2

A: dropped — only one certificate authority actually verifies public key in chain, but can use information about different certificate authorities in chain

B+C: chain is bigger than if we just had trusted cert verify directly

D: can only have top-level certificate authorities, using chain to fill in

E: can trust certificate authority to to tell use to trust other certificate authority without persistently remembering other certificate authorities

quiz Q3

A: A can see that B successfully decrypted, so B got A's message somehow (even if in different context)

C: protocol has B decrypting arbitrary 'key' (encrypted to B) and sending it back

- B has no way of knowing it is key versus something else (without additional steps)

D: A can reuse old message

E: B's reply needs to correspond to A's message

quiz Q4

	0	1	2	3	4	5	6
A	X	X	X	X			
B		X	X	X	X		
C			X	X	X	X	

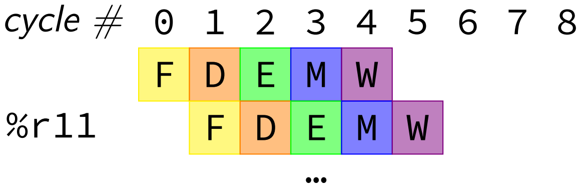
quiz Q5

$1000 + 7$ stages total because of overlap

easily less than 1500 ns

exercise: throughput/latency (1)

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

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

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

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
				…	

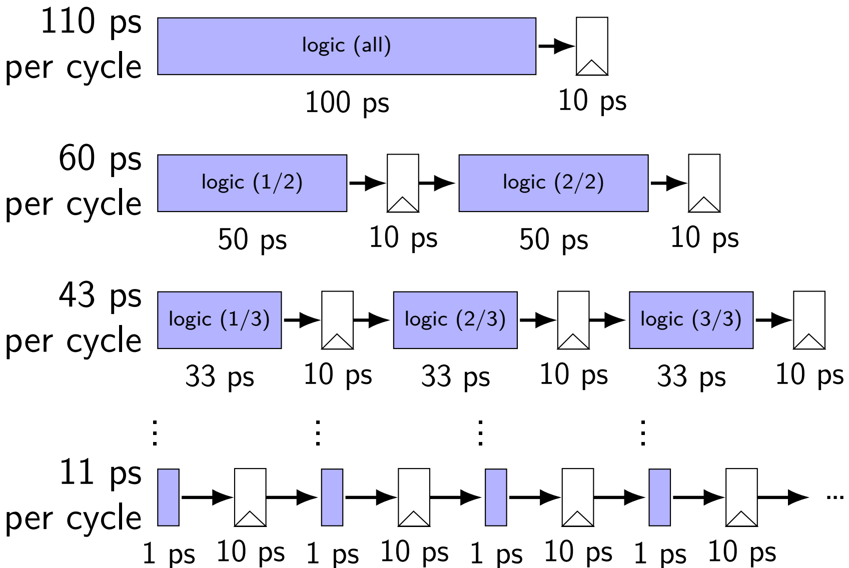
```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: …
```

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

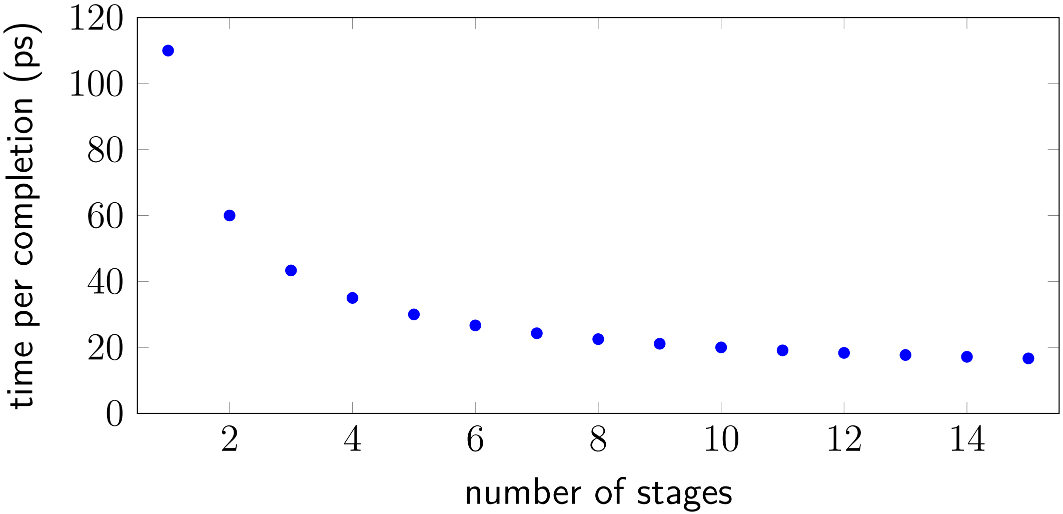
double number of pipeline stages (to 10) + decrease cycle time from 500 ps to 250 ps — throughput?

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

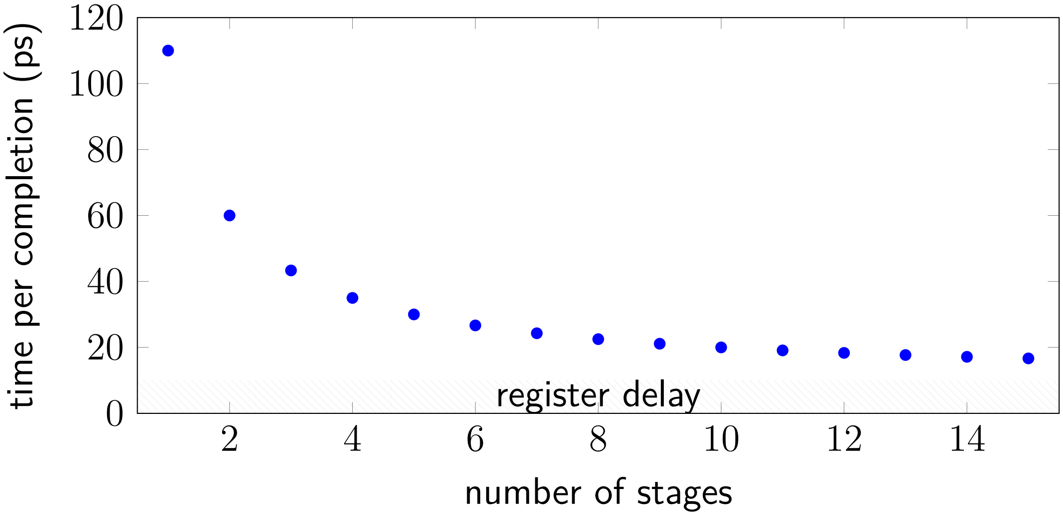
diminishing returns: register delays



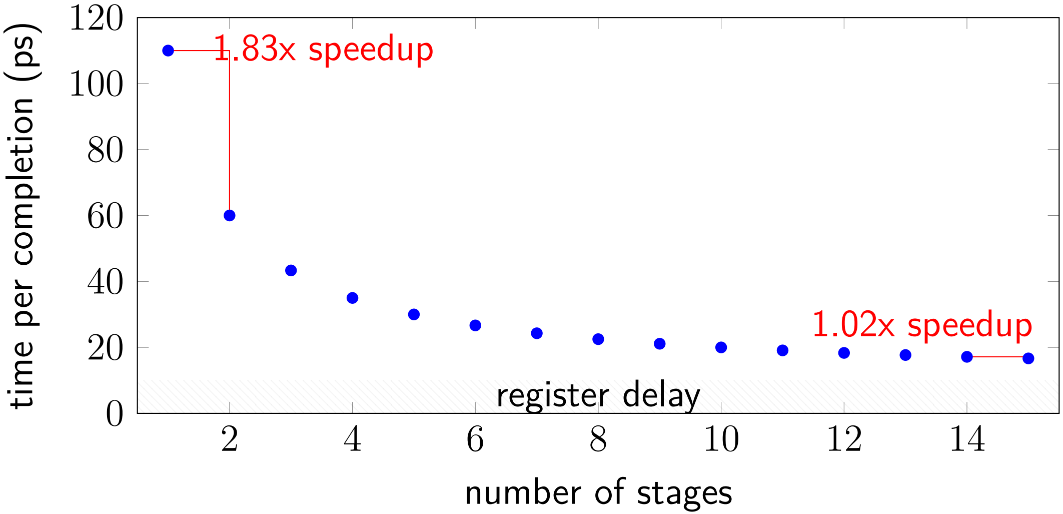
diminishing returns: register delays



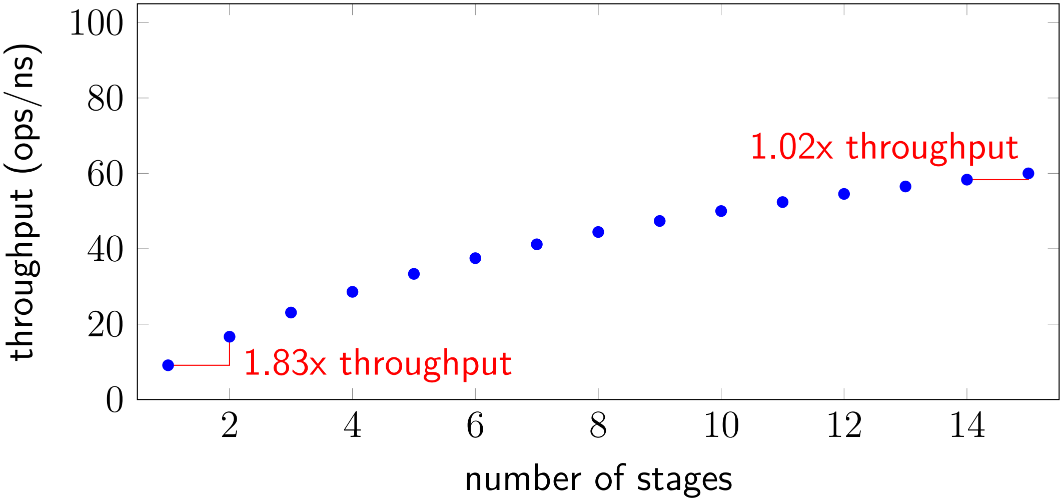
diminishing returns: register delays



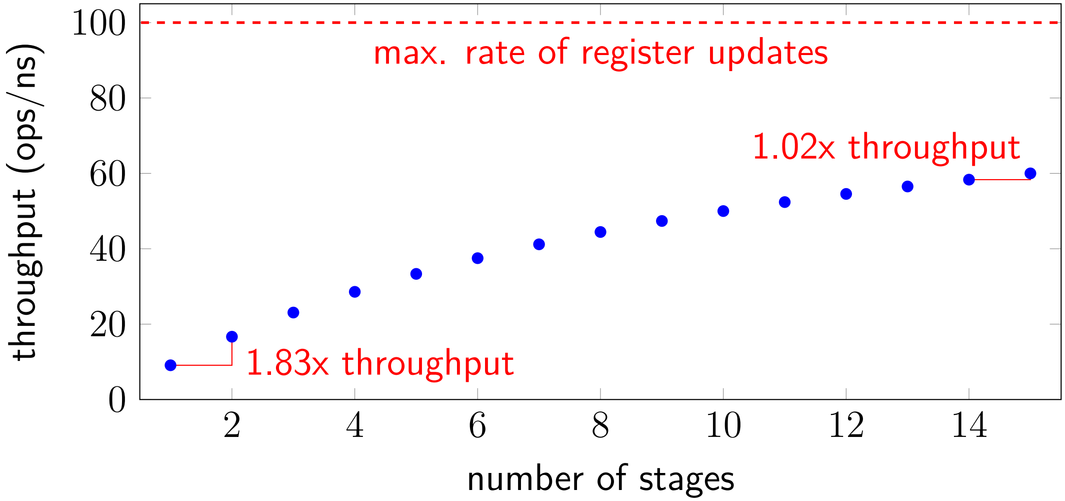
diminishing returns: register delays



diminishing returns: register delays



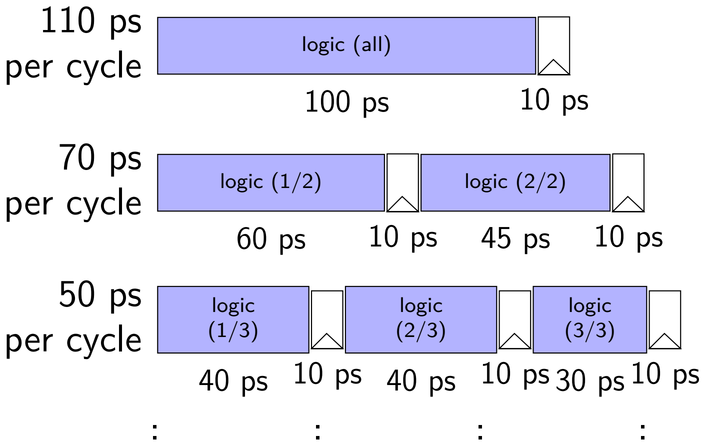
diminishing returns: register delays



diminishing returns: uneven split

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

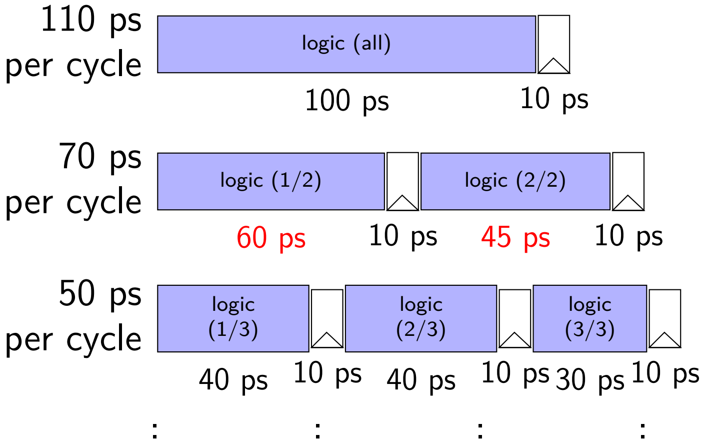
Probably not...



diminishing returns: uneven split

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

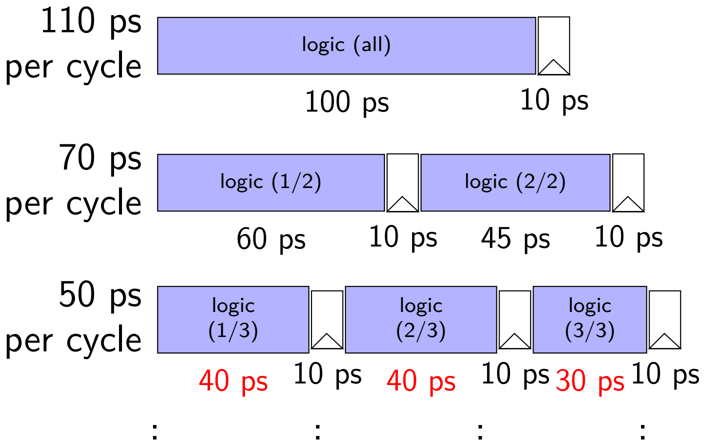
Probably not...



diminishing returns: uneven split

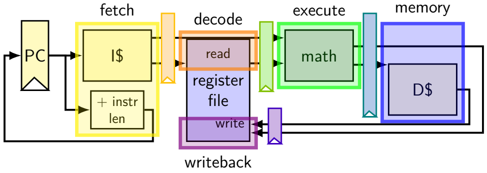
Can we split up some logic (e.g. adder) arbitrarily?

Probably not...



a data hazard

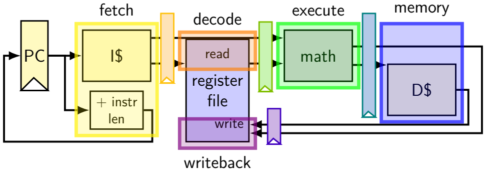
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9   // R8 + R9 -> R9
addq %r9, %r8   // R9 + R8 -> R9
addq ...
addq ...
```



	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 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
- (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)

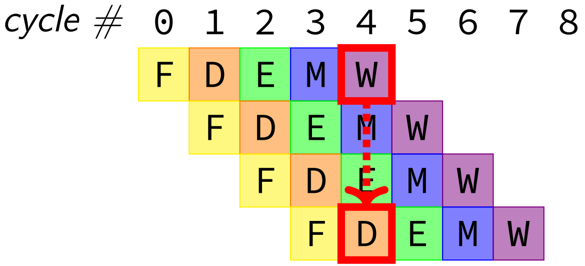
make it compiler's job

problem: recompile everytime processor changes?

stalling/nop pipeline diagram (1)

cycle #	0	1	2	3	4	5	6	7	8
`add %r8, %r9`	F	D	E	M	W				
`nop`		F	D	E	M	W			
`nop`			F	D	E	M	W		
`addq %r9, %r8`				F	D	E	M	W	

stalling/nop pipeline diagram (1)



	cycle # 0	1	2	3	4	5	6	7	8
`add %r8, %r9`	F	D	E	M	W				
`nop`		F	D	E	M	W			
`nop`			F	D	E	M	W		
`addq %r9, %r8`				F	D	E	M	W	

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)

cycle #	0	1	2	3	4	5	6	7	8
`add %r8, %r9`	F	D	E	M	W				
`nop`		F	D	E	M	W			
`nop`			F	D	E	M	W		
`nop`				F	D	E	M	W	
`addq %r9, %r8`					F	D	E	M	W

stalling/nop pipeline diagram (2)

cycle #	0	1	2	3	4	5	6	7	8
`add %r8, %r9`	F	D	E	M	W				
`nop`		F	D	E	M	W			
`nop`			F	D	E	M	W		
`nop`				F	D	E	M	W	
`addq %r9, %r8`					F	D	E	M	W

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

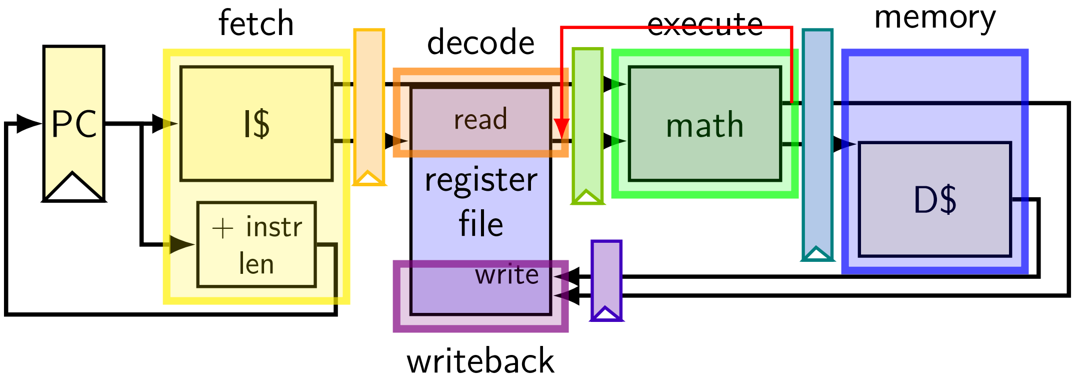
opportunity

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

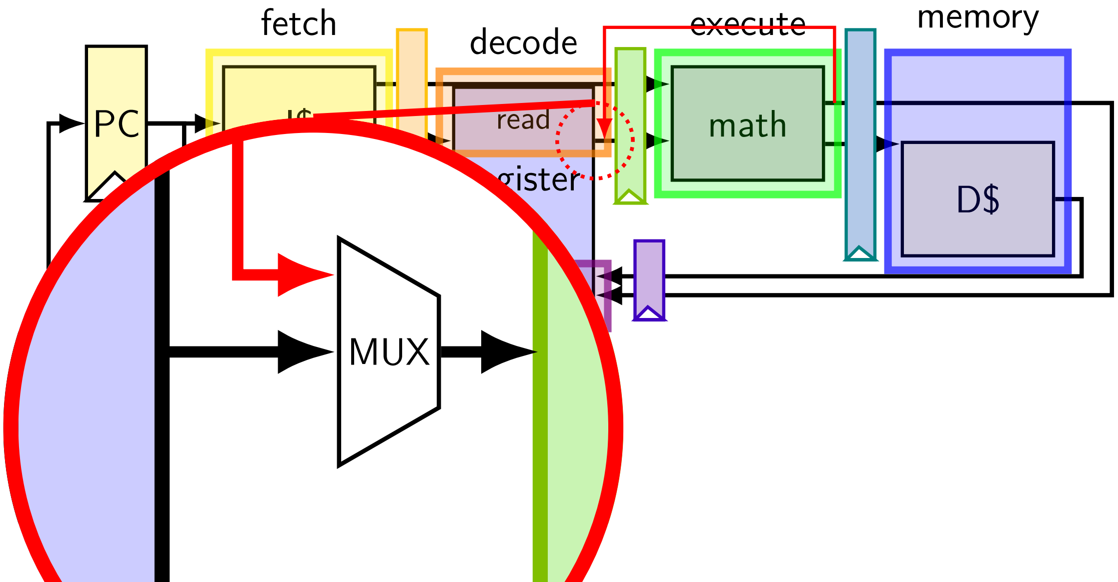
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



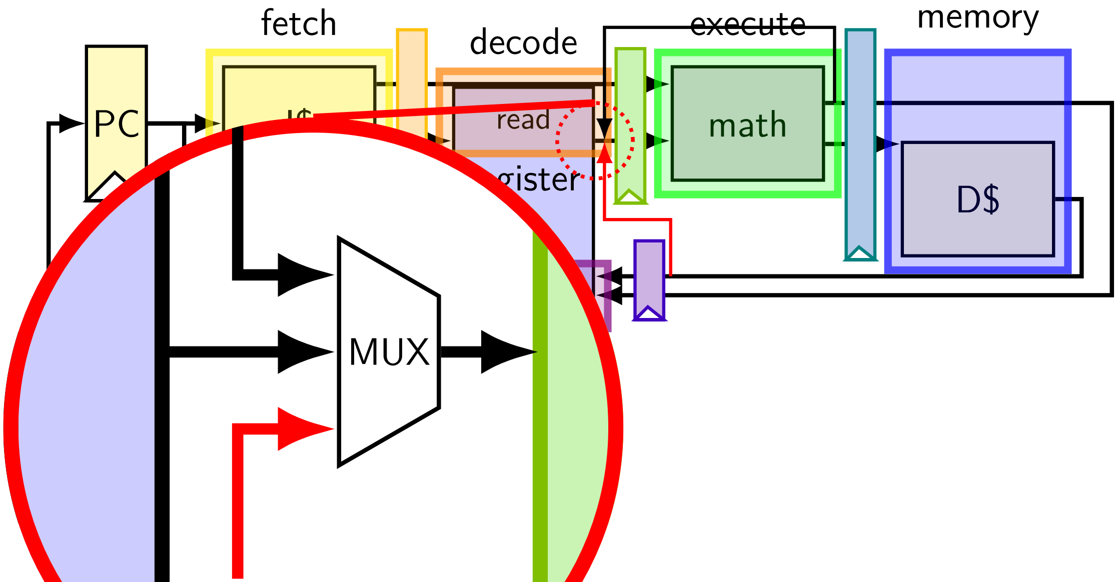
opportunity 2

```
// 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`	F	D	E	M	W				
`subq %r8, %r10`		F	D	E	M	W			
`xorq %r8, %r9`			F	D	E	M	W		
`andq %r9, %r8`				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

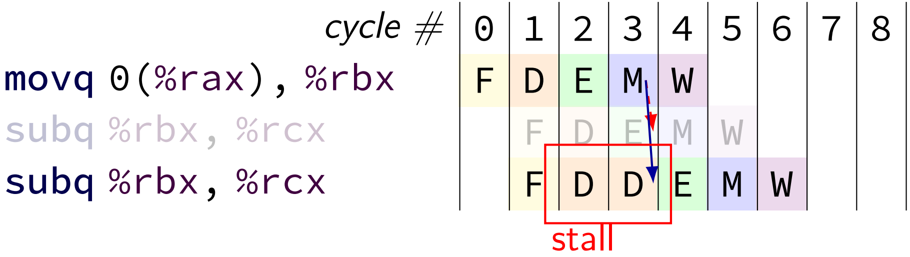
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	E	M	W			

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



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

combine stalling and forwarding to resolve hazard

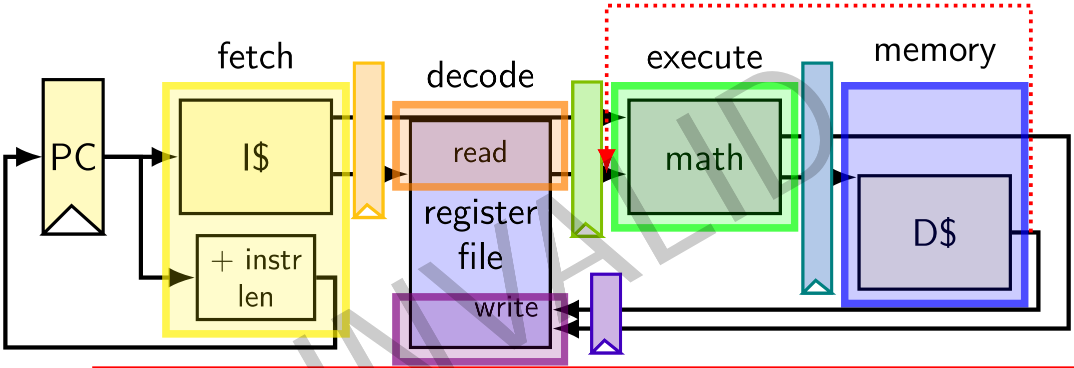
assumption in diagram: hazard detected in `subq`'s decode stage

(since easier than detecting it in fetch stage)

solveable problem

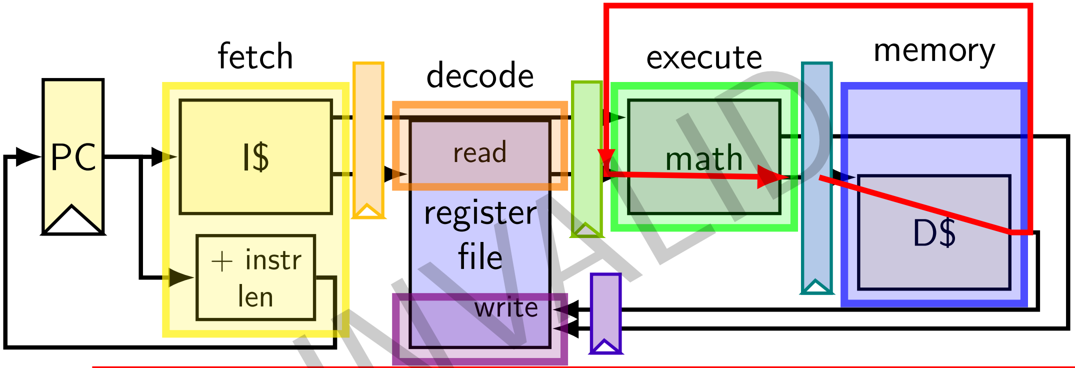
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`movq %rbx, 0(%rcx)`		F	D	E	M	W			

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→writeb	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

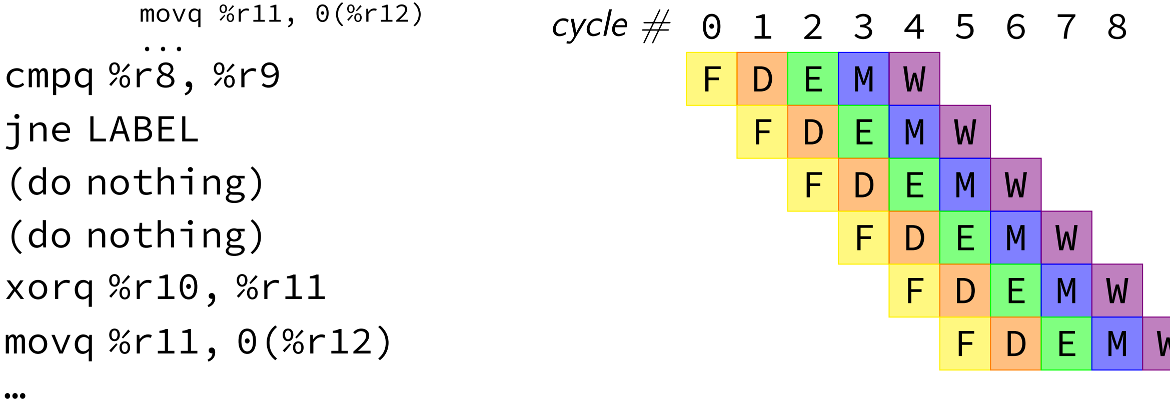
```
0x00: cmpq %r8, %r9
0x08: je   0xFFFF
0x10: addq %r10, %r11
```

	fetch	fetch→decode		decode→execute		execute→writeb	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?

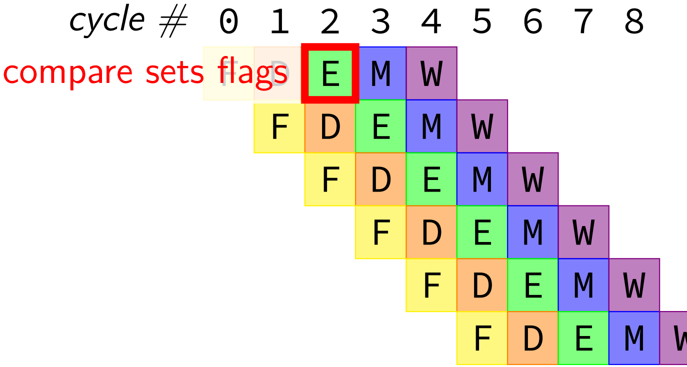
```
cmpq %r8, %r9
jne LABEL          // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



jXX: stalling?

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

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



jXX: stalling?

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

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

compute if jump goes to LABEL

jXX: stalling?

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

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

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
        ...
```

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

jXX: speculating wrong

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

jXX: speculating wrong

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

“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

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

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