last time

public/private key pairs

- give public key to (potentially) everyone
- keep private key secret (even from correspondents)

asymmetric encryption

- encrypt with public key / decrypt with private key

digital signatures

- public-key version of message authentication code (‘keyed checksum’)
- ‘sign’ with private key / verify with public key

replay attacks

- encrypted/signed/MAC’d/etc. messages can be used out-of-context
- fix: include needed context/prevent reuse

getting public keys?

browser talking to websites
needs public keys of every single website?

not really feasible, but...

certificate idea

let's say A has B's public key already.

if C wants B's public key and knows A's already:

A can send C:

> "B's public key is XXX" AND
> Sign(A's private key, "B's public key is XXX")

if C trusts A, now C has B's public key

> if C does not trust A, well, can't trust this either

certificate authorities

instead, have public keys of trusted *certificate authorities*

only 10s of them, probably

websites go to certificates authorities with their public key

certificate authorities sign messages like:
“The public key for foo.com is XXX.”

these signed messages called “certificates”

example web certificate (1)

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            81:13:c9:49:90:8c:81:bf:94:35:22:cf:e0:25:20:33
        Signature Algorithm: sha256WithRSAEncryption
        Issuer:
            commonName                = InCommon RSA Server CA
            organizationalUnitName    = InCommon
            organizationName          = Internet2
            localityName              = Ann Arbor
            stateOrProvinceName       = MI
            countryName               = US
        Validity
            Not Before: Feb 28 00:00:00 2022 GMT
            Not After : Feb 28 23:59:59 2023 GMT
        Subject:
            commonName                = collab.its.virginia.edu
            organizationalUnitName    = Information Technology and Communication
            organizationName          = University of Virginia
            stateOrProvinceName       = Virginia
            countryName               = US
.....
```

example web certificate (1)

```
Certificate:
    Data:
....
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:a2:fb:5a:fb:2d:d2:a7:75:7e:eb:f4:e4:d4:6c:
                    94:be:91:a8:6a:21:43:b2:d5:9a:48:b0:64:d9:f7:
                    f1:88:fa:50:cf:d0:f3:3d:8b:cc:95:f6:46:4b:42:
....
        X509v3 extensions:
....
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
....
            X509v3 Subject Alternative Name:
                DNS:collab.its.virginia.edu
                DNS:collab-prod.its.virginia.edu
                DNS:collab.itc.virginia.edu
    Signature Algorithm: sha256WithRSAEncryption
         39:70:70:77:2d:4d:0d:0a:6d:d5:d1:f5:0e:4c:e3:56:4e:31:
....
```

certificate chains

That certificate signed by “InCommon RSA Server CA”

CA = certificate authority

so their public key, comes with my OS/browser?

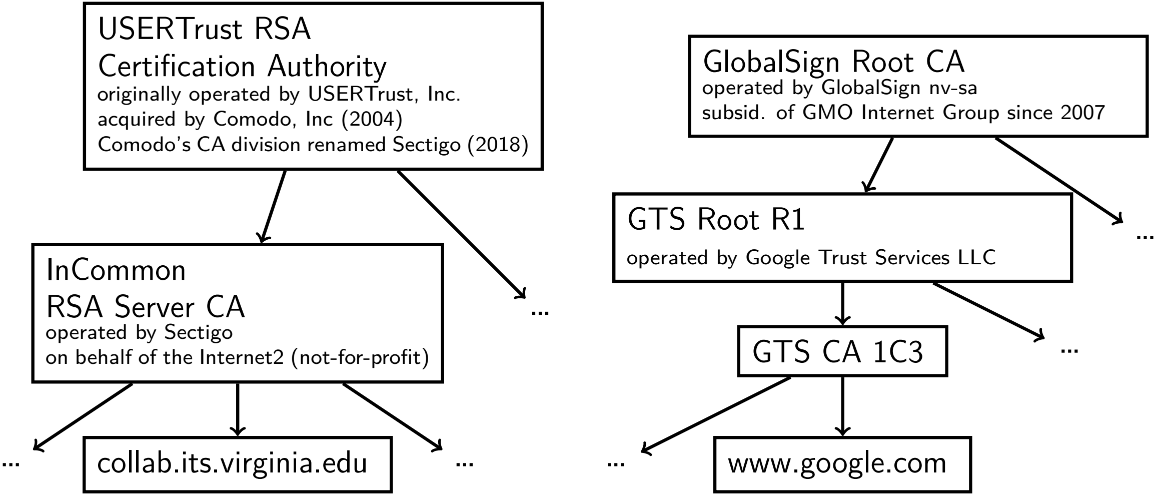
not exactly…

they have their own certificate signed by “USERTrust RSA Certification Authority”

and their public key comes with your OS/browser?

(but both CAs now operated by UK-based Sectigo)

certificate hierarchy



USERTrust RSA Certification Authority
originally operated by USERTrust, Inc.
acquired by Comodo, Inc (2004)
Comodo's CA division renamed Sectigo (2018)

InCommon RSA Server CA
operated by Sectigo
on behalf of the Internet2 (not-for-profit)

collab.its.virginia.edu

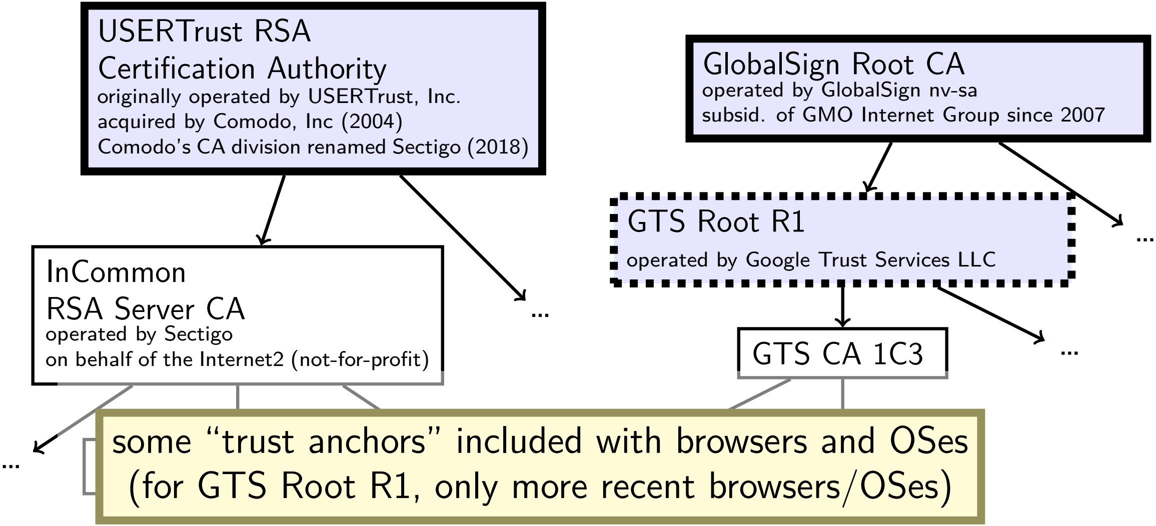
GlobalSign Root CA
operated by GlobalSign nv-sa
subsid. of GMO Internet Group since 2007

GTS Root R1
operated by Google Trust Services LLC

GTS CA 1C3

www.google.com

certificate hierarchy



USERTrust RSA Certification Authority
originally operated by USERTrust, Inc.
acquired by Comodo, Inc (2004)
Comodo's CA division renamed Sectigo (2018)

GlobalSign Root CA
operated by GlobalSign nv-sa
subsid. of GMO Internet Group since 2007

GTS Root R1
operated by Google Trust Services LLC

InCommon RSA Server CA
operated by Sectigo
on behalf of the Internet2 (not-for-profit)

GTS CA 1C3

...

...

...

...

some "trust anchors" included with browsers and OSes
(for GTS Root R1, only more recent browsers/OSes)

how many trust anchors?

Mozilla Firefox (as of 27 Feb 2023)

155 trust anchors
operated by 55 distinct entities

Microsoft Windows (as of 27 Feb 2023)

237 trust anchors
operated by 86 distinct entities

public-key infrastructure

ecosystem with certificate authorities
and certificates for everyone

called “public-key infrastructure”

several of these:

for verifying identity of websites
for verifying origin of domain name records (kind-of)
for verifying origin of applications in some OSes/app stores/etc.
for encrypted email in some organizations
…

exercise

exercise: how should website certificates verify identity?

how do certificate authorities verify

for web sites, set by CA/Browser Forum

organization of:

- everyone who ships code with list of valid certificate authorities
  - Apple, Google, Microsoft, Mozilla, Opera, Cisco, Qihoo 360, Brave, …
- certificate authorities

decide on rules (“baseline requirements”) for what CAs do

BR domain name identity validation

options involve CA choosing random value and:

sending it to domain contact (with domain registrar) and receive response with it, or

observing it placed in DNS or website or sent from server in other specific way

exercise: problems this doesn't deal with?

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

- some browsers check these, sometimes

certificate transparency

- public logs of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate which CAs are allowed to issue certificates in DNS
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

- some browsers check these, sometimes

certificate transparency

- public logs of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate which CAs are allowed to issue certificates in DNS
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

- some browsers check these, sometimes

certificate transparency

- public logs of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate which CAs are allowed to issue certificates in DNS
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

- some browsers check these, sometimes

certificate transparency

- public logs of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate which CAs are allowed to issue certificates in DNS
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

motivation: summary for signature

digital signatures typically have size limit

…but we want to sign very large messages

solution: get secure “summary” of message

cryptographic hash

hash(M) = X

given X:

hard to find message other than by guessing

given X, M:

hard to find second message so that hash(second message) = H

example uses:

substitute for original message in digital signature

building message authentication codes

password hashing

cryptographic hash functions need (basically) guessing to ‘reverse’

idea: store cryptographic hash of password instead of password

- attacker who gets hash doesn’t get password
- but can still check entered password is correct

password hashing

cryptographic hash functions need (basically) guessing to ‘reverse’

idea: store cryptographic hash of password instead of password

- attacker who gets hash doesn’t get password
- but can still check entered password is correct

problem: with fast hash function, can try lots of guesses fast

password hashing

cryptographic hash functions need (basically) guessing to ‘reverse’

idea: store cryptographic hash of password instead of password

- attacker who gets hash doesn’t get password
- but can still check entered password is correct

problem: with fast hash function, can try lots of guesses fast

fix: special slow/resource-intensive cryptograph hash functions

- Argon2i
- scrypt
- PBKDF2

random numbers

want keys, etc. to be unguessable and evenly distributed

solution: random numbers

but: many random number functions are not cryptographically secure!

- example NOT SECURE: ~~C `rand()`; Python's `random.random`~~
- better: Python's `secrets`, `os.urandom`; Linux `getrandom()`, `/dev/urandom`

extra effort to ensure not guessable

need to incorporate "entropy" from unpredictable sources

- deliberately unstable circuit; exact timing of input/output; etc.

just asymmetric?

given public-key encryption + digital signatures…

why bother with the symmetric stuff?

symmetric stuff much faster

symmetric stuff much better at supporting larger messages

key agreement

problem: A has B's public encryption key
wants to choose shared secret

some ideas:

- A chooses a key, sends it encrypted to B
- A sends a public key encrypted B, B chooses a key and sends it back

key agreement

problem: A has B's public encryption key
wants to choose shared secret

some ideas:

- A chooses a key, sends it encrypted to B
- A sends a public key encrypted B, B chooses a key and sends it back

alternate model:

- both sides generate random values
- derive public-key like "key shares" from values
- use math to combine "key shares"
- kinda like A + B both sending each other public encryption keys

Diffie-Hellman key agreement (2)

A and B want to agree on shared secret

A chooses random value Y

A sends public value derived from Y (“key share”)

B chooses random value Z

B sends public value derived from Z (“key share”)

A combines Y with public value from B to get number

B combines Z with public value from A to get number

and b/c of math chosen, both get same number

Diffie-Hellman key agreement (1)

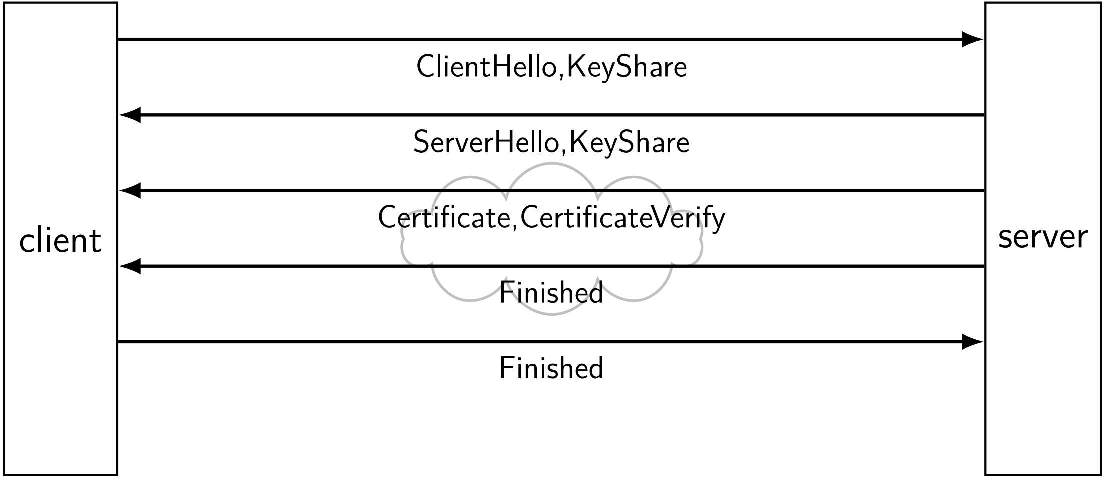
math requirement:

some f, so $f(f(X,Y),Z) = f(f(X,Z),Y)$
(that's hard to invert, etc.)

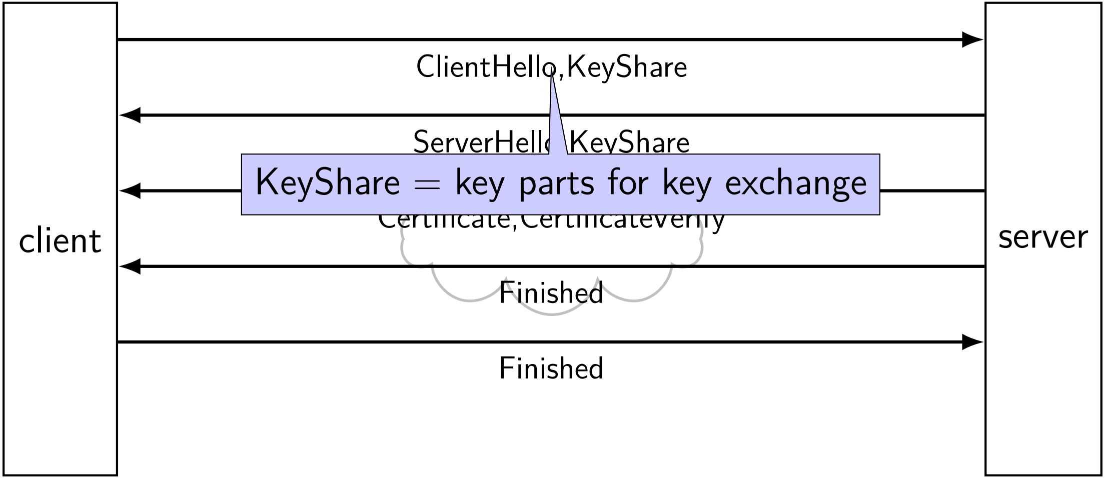
choose X in advance and:

A randomly chooses Y	B randomly chooses Z
A sends $f(X,Y)$ to B	B sends $f(X,Z)$ to A
A computes $f(f(X,Z),Y)$	B computes $f(f(X,Y),Z)$

typical TLS handshake



typical TLS handshake



client

server

ClientHello,KeyShare

ServerHello,KeyShare

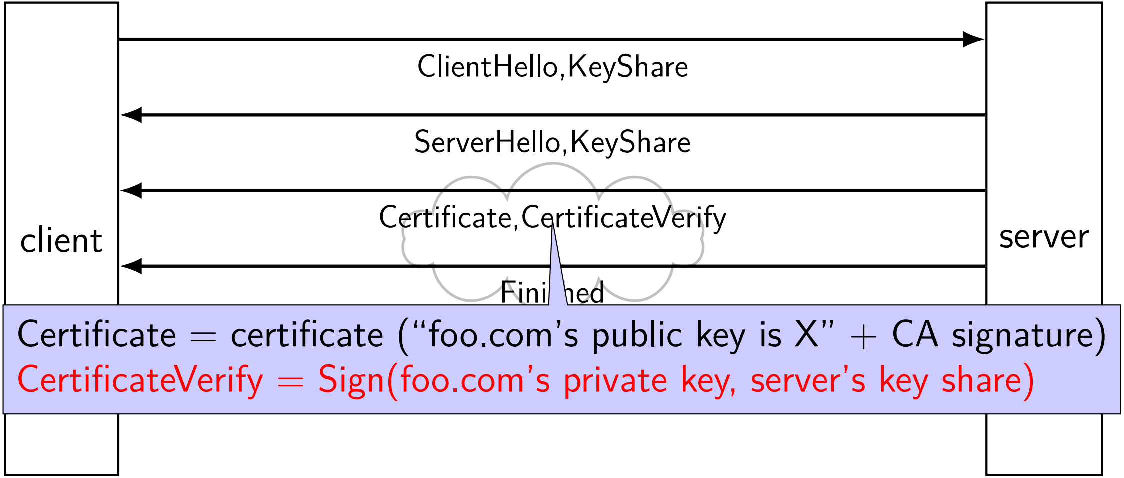
Certificate,CertificateVerify

Finished

Finished

KeyShare = key parts for key exchange

typical TLS handshake



client

server

ClientHello,KeyShare

ServerHello,KeyShare

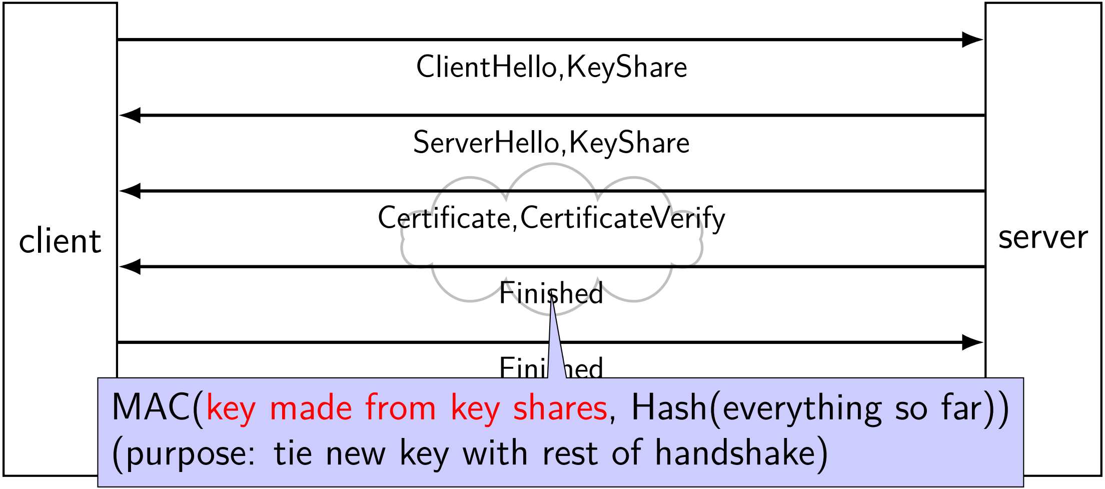
Certificate,CertificateVerify

Finished

Certificate = certificate (“foo.com’s public key is X” + CA signature)

CertificateVerify = Sign(foo.com’s private key, server’s key share)

typical TLS handshake



client

ClientHello,KeyShare

ServerHello,KeyShare

Certificate,CertificateVerify

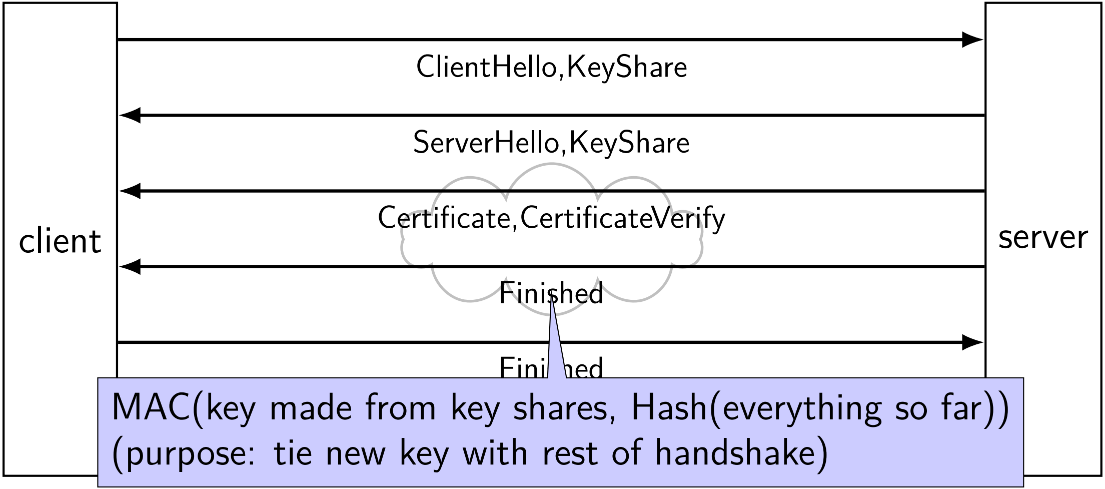
Finished

Finished

server

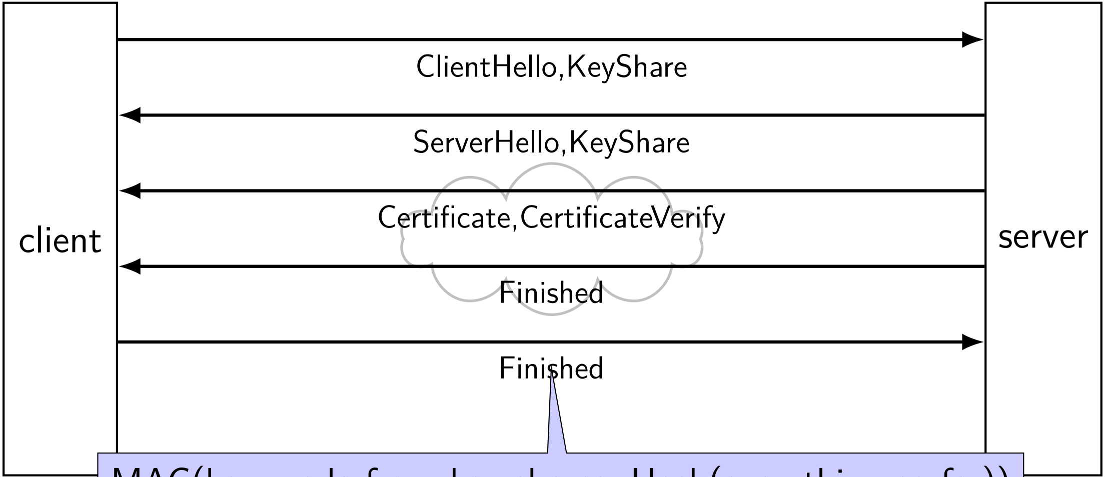
MAC(key made from key shares, Hash(everything so far))
(purpose: tie new key with rest of handshake)

typical TLS handshake



MAC(key made from key shares, Hash(everything so far))
(purpose: tie new key with rest of handshake)

typical TLS handshake



client

server

ClientHello,KeyShare

ServerHello,KeyShare

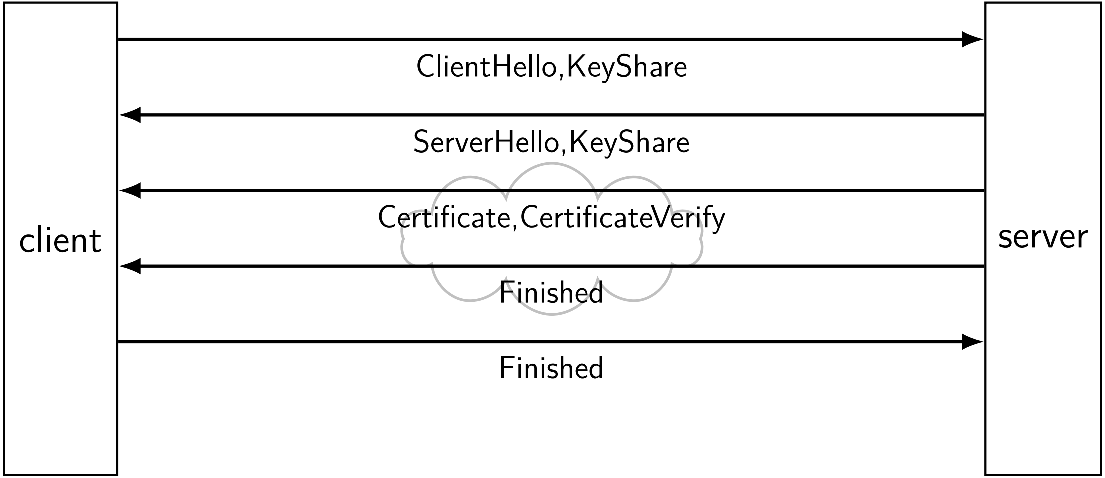
Certificate,CertificateVerify

Finished

Finished

MAC(key made from key shares, Hash(everything so far))
(purpose: tie new key with rest of handshake)

typical TLS handshake



TLS: after handshake

use key shares results to get **several** keys

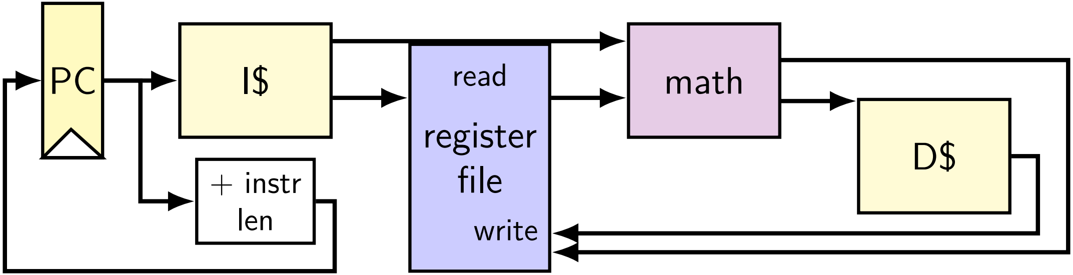
take hash(something + shared secret) to derive each key

separate keys for each direction (server → client and vice-versa)

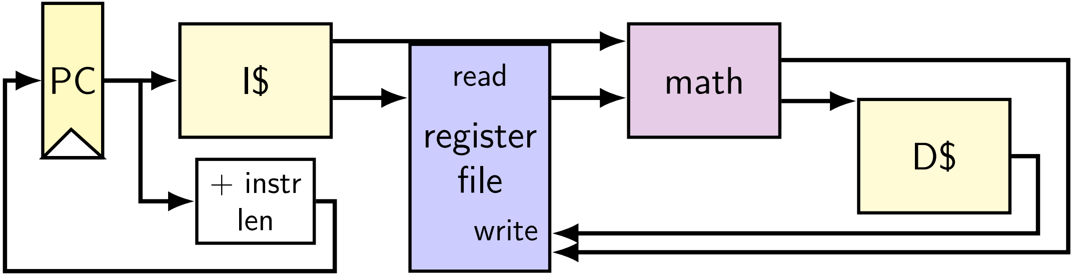
often separate keys for encryption and MAC

later messages use encryption + MAC + nonces

simple CPU



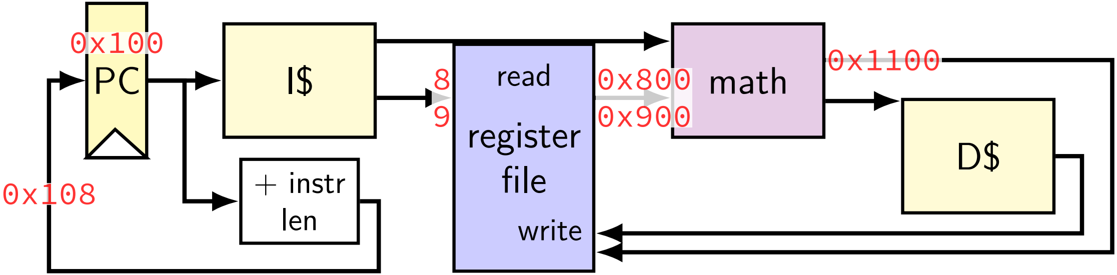
running instructions



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

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

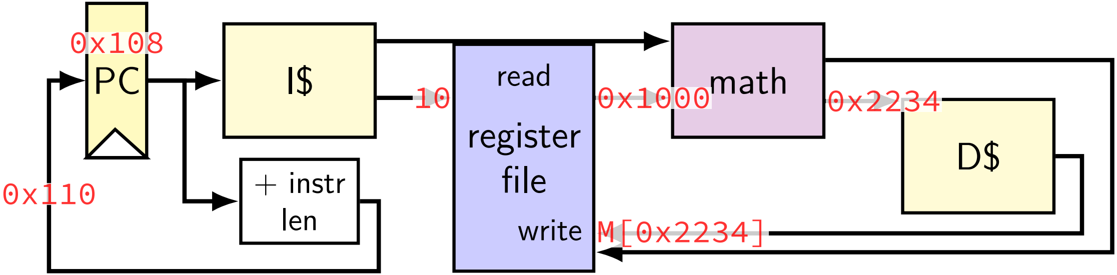
running instructions



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

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

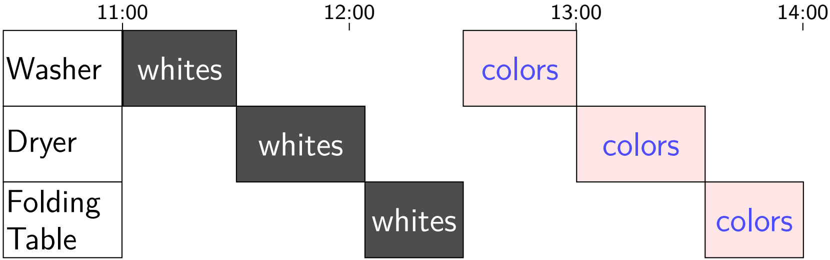
running instructions



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

```
…
%r8: 0x800
%r9: 0x1100
%r10: 0x1000
%r11: M[0x2234]
…
```

Human pipeline: laundry



11:00

12:00

13:00

14:00

Washer

whites

colors

Dryer

whites

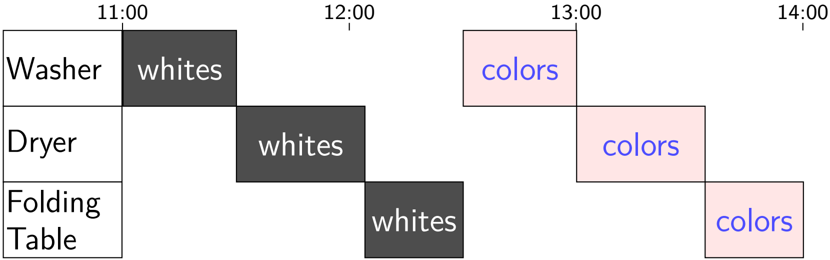
colors

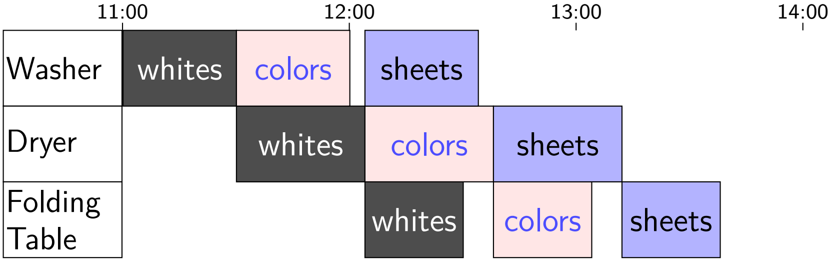
Folding Table

whites

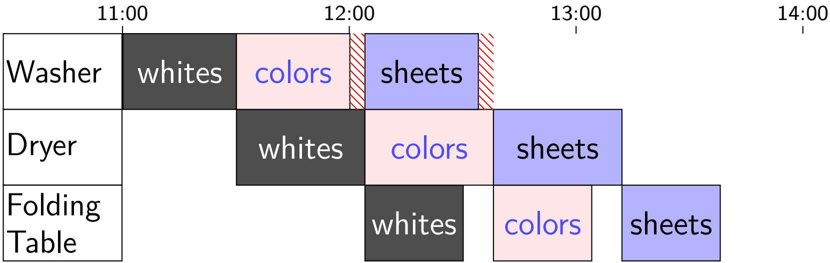
colors

Human pipeline: laundry





Waste (1)



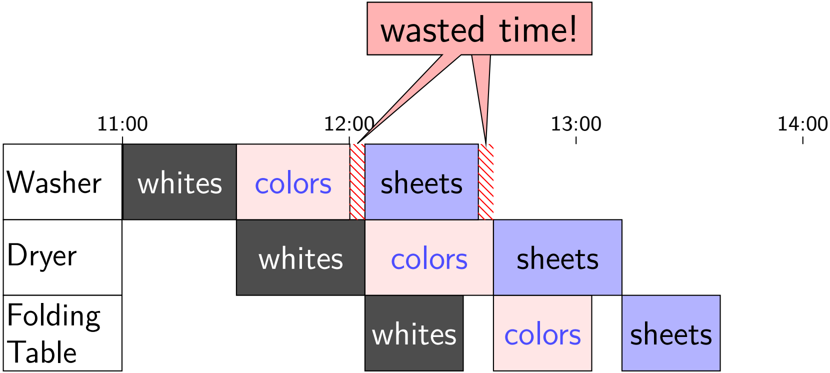
11:00
12:00
13:00
14:00

Washer
whites
colors
sheets

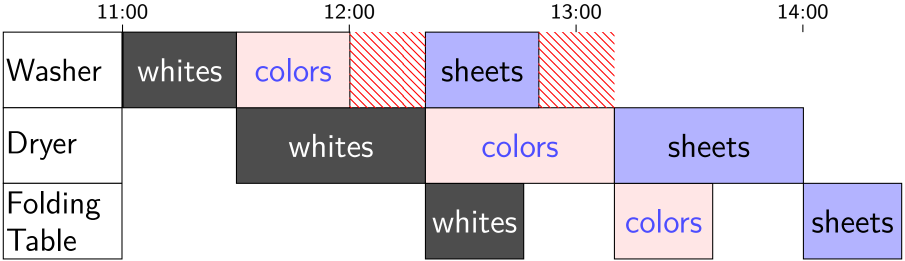
Dryer
whites
colors
sheets

Folding Table
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)



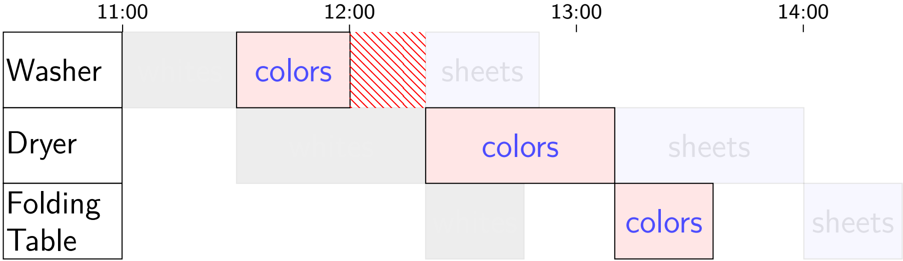
11:00
12:00
13:00
14:00

Washer
whites
colors
sheets

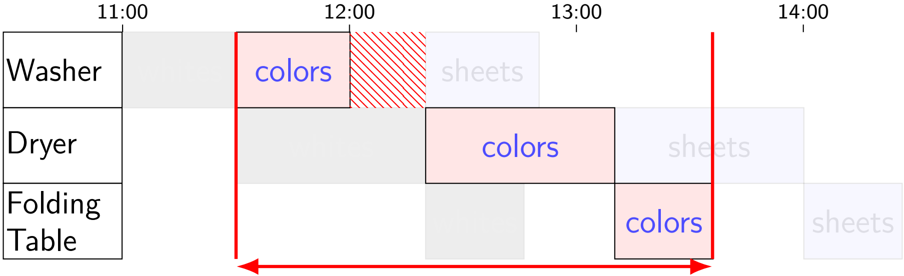
Dryer
whites
colors
sheets

Folding Table
whites
colors
sheets

Latency — Time for One

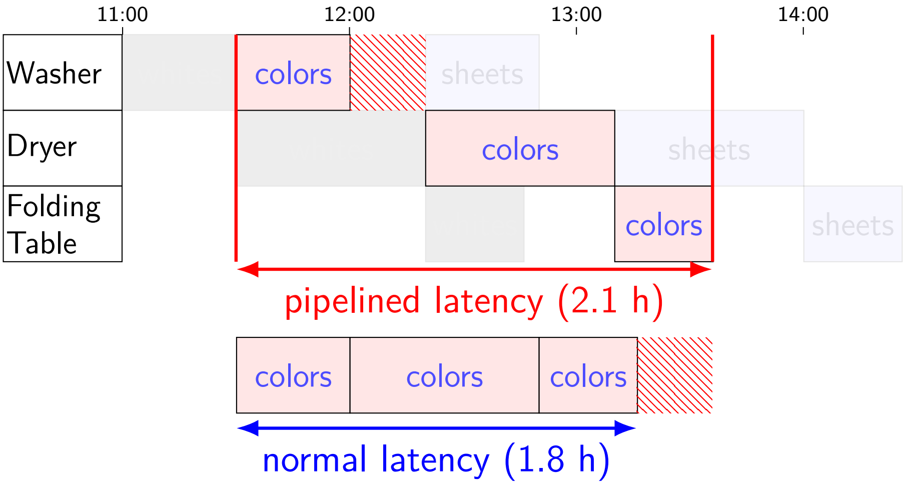


Latency — Time for One

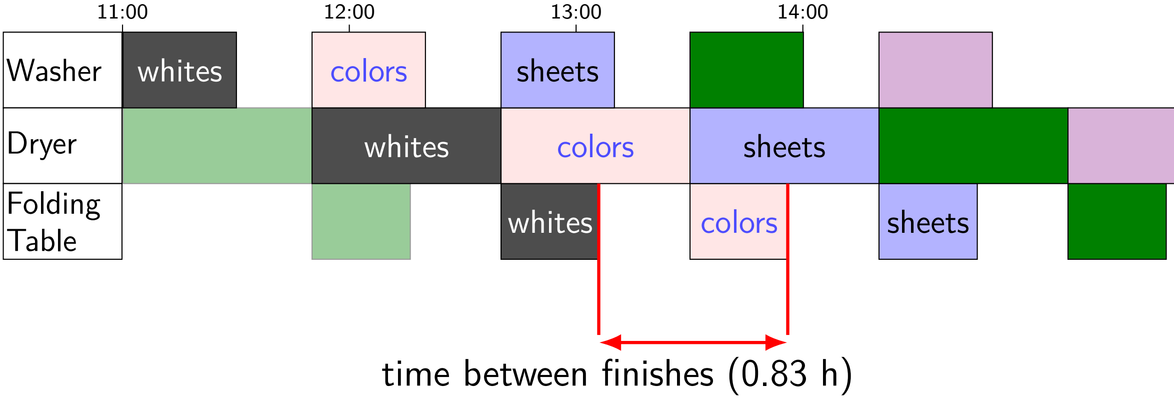


pipelined latency (2.1 h)

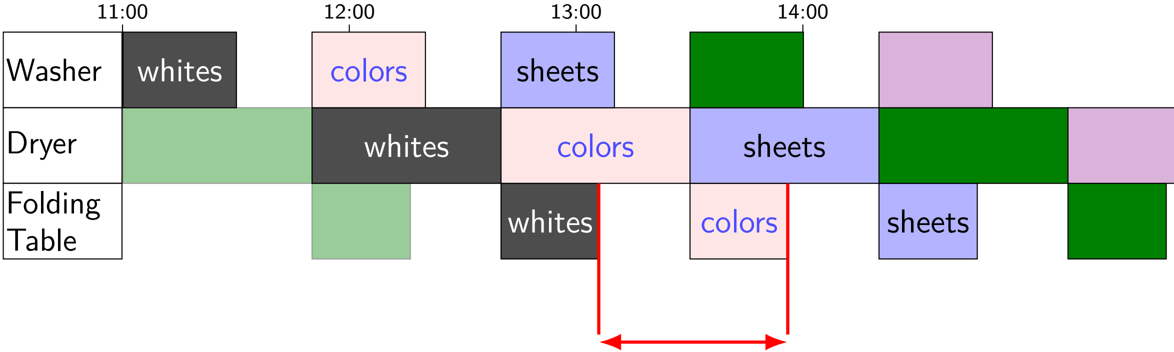
Latency — Time for One



Throughput — Rate of Many



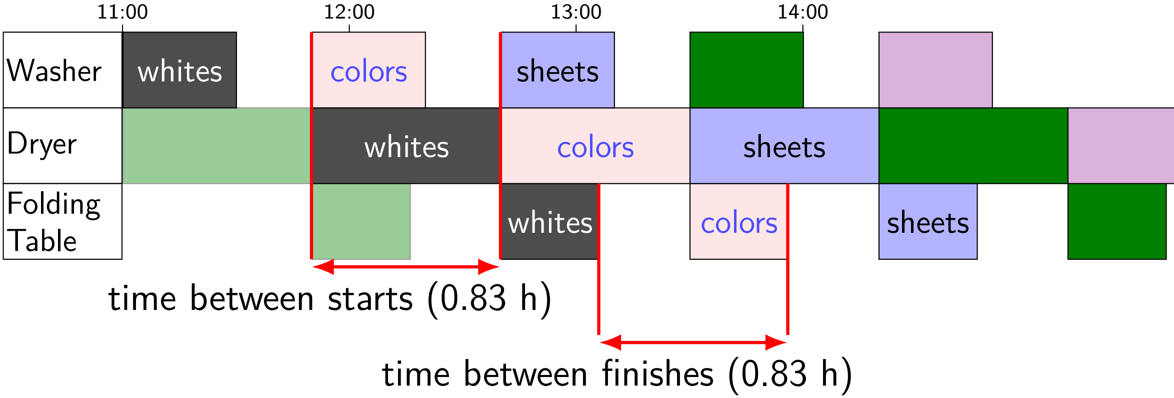
Throughput — Rate of Many



time between finishes (0.83 h)

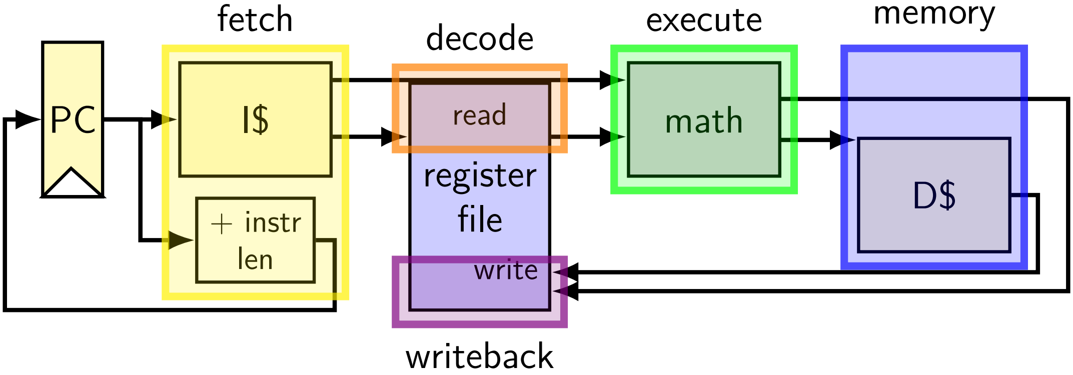
$$\frac{1 \text{ load}}{0.83\text{h}} = \color{red}{1.2 \text{ loads/h}}$$

Throughput — Rate of Many



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

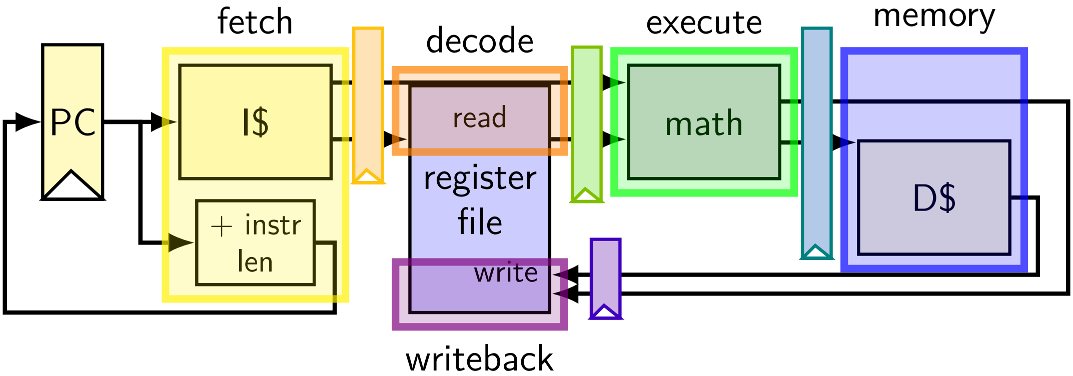
adding stages (one way)



divide running instruction into steps

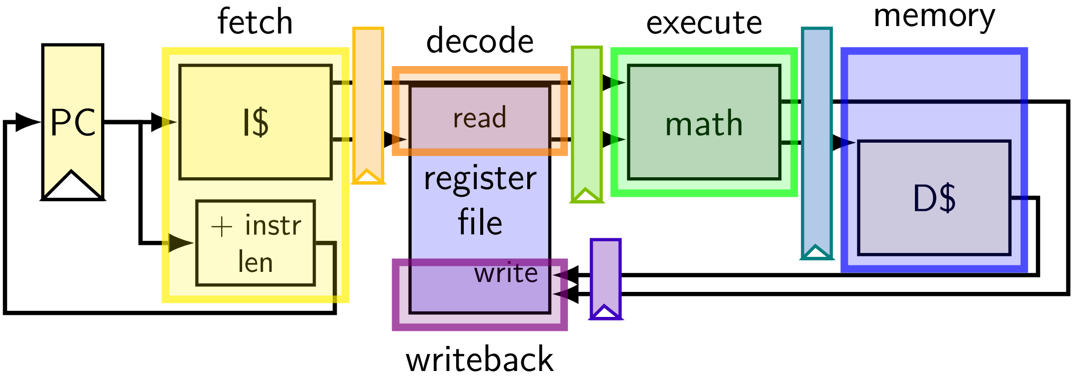
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

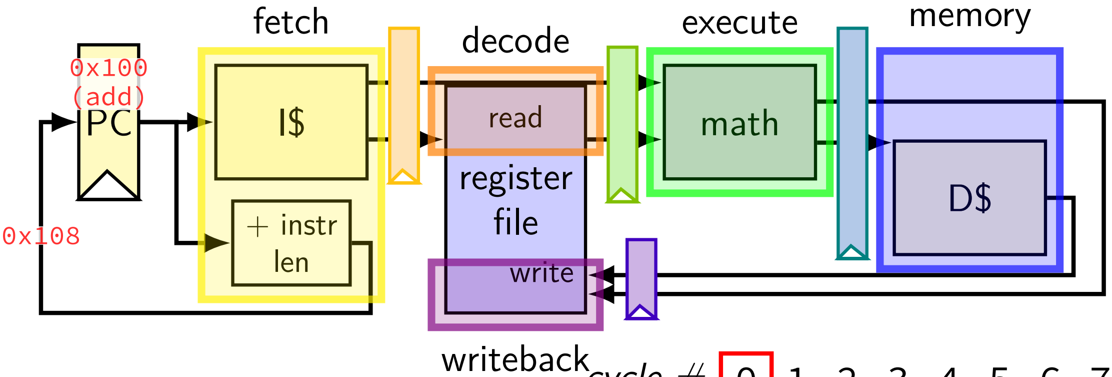


add ‘pipeline registers’ to hold values from instruction

running some instructions



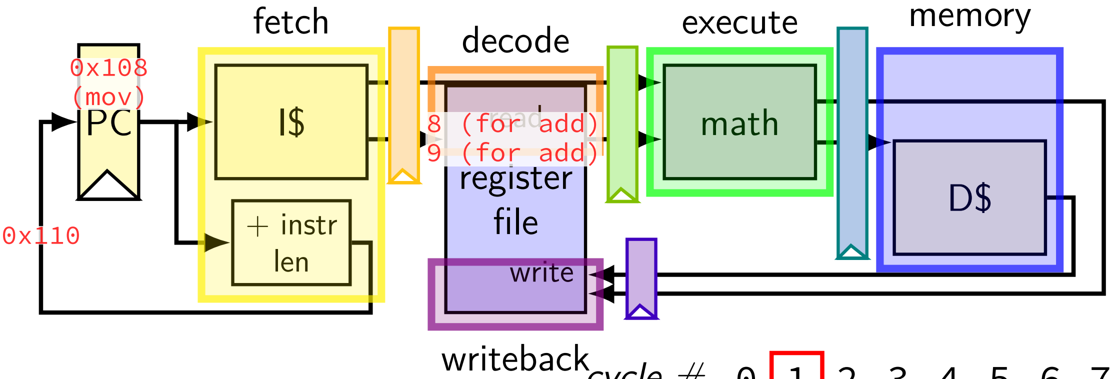
running some instructions



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

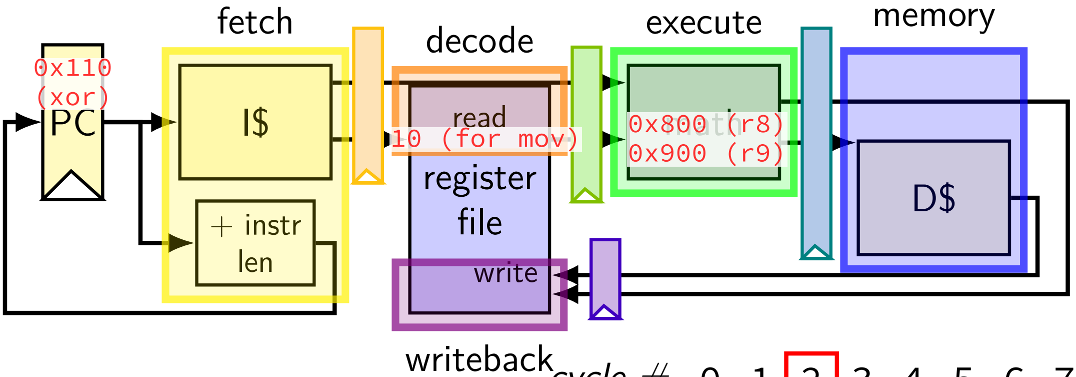
cycle #	0	1	2	3	4	5	6	7	8
0x100: add %r8, %r9	F	D	E	M	W				
0x108: mov 0x1234(%r10), %r11		F	D	E	M	W			
0x110: xor %r12, %r13			F	D	E	M	W		

running some instructions



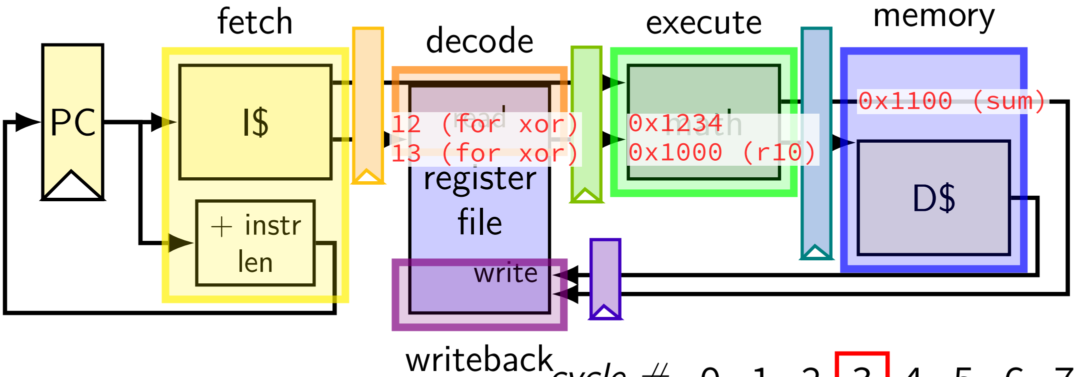
	cycle # 0	1	2	3	4	5	6	7	8
0x100: add %r8, %r9	F	D	E	M	W				
0x108: mov 0x1234(%r10), %r11		F	D	E	M	W			
0x110: xor %r12, %r13			F	D	E	M	W		

running some instructions



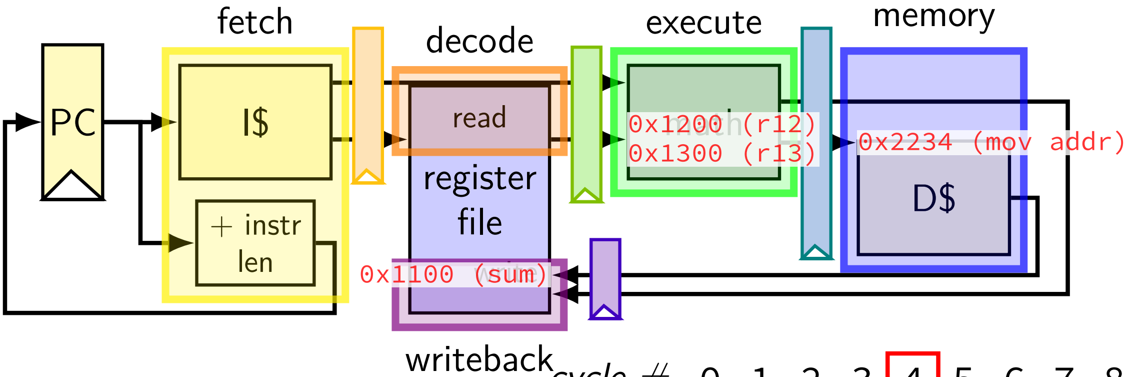
	cycle # 0	1	2	3	4	5	6	7	8
`0x100: add %r8, %r9`	F	D	E	M	W				
`0x108: mov 0x1234(%r10), %r11`		F	D	E	M	W			
`0x110: xor %r12, %r13`			F	D	E	M	W		

running some instructions



	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: xor %r12, %r13`			F	D	E	M	W		

running some instructions



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: xor %r12, %r13`			F	D	E	M	W		

why registers?

example: fetch/decode

need to store current instruction somewhere …while fetching next one

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
				…	

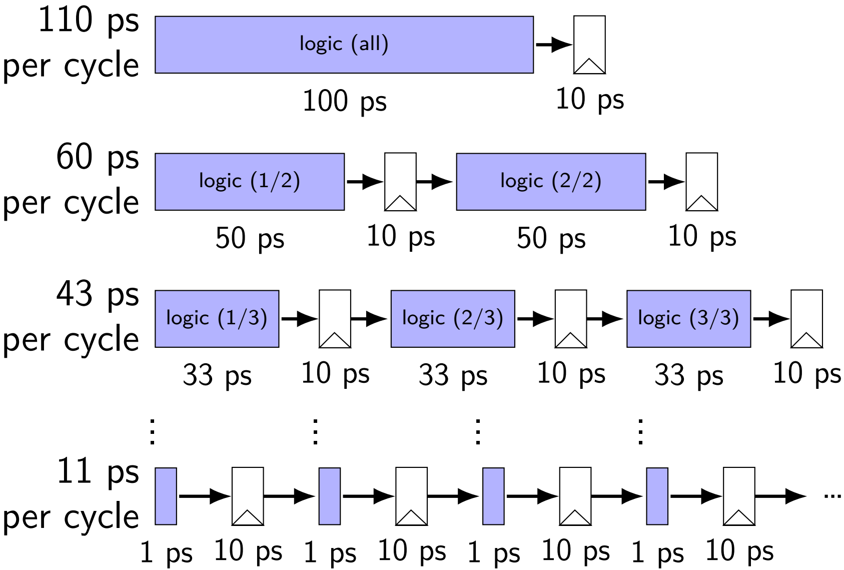
```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: …
```

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

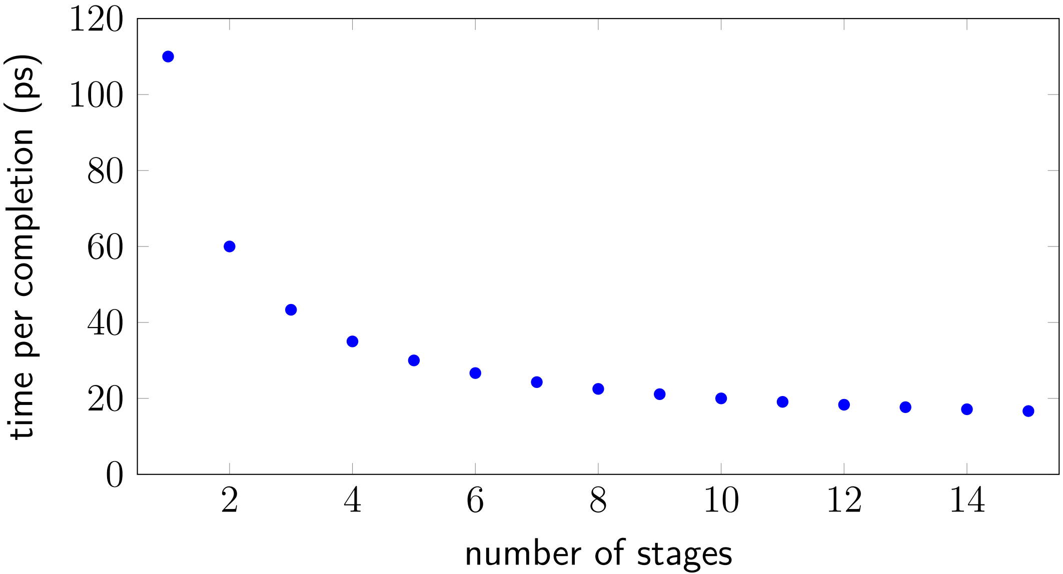
double number of pipeline stages (to 10) + decrease cycle time from 500 ps to 250 ps — throughput?

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

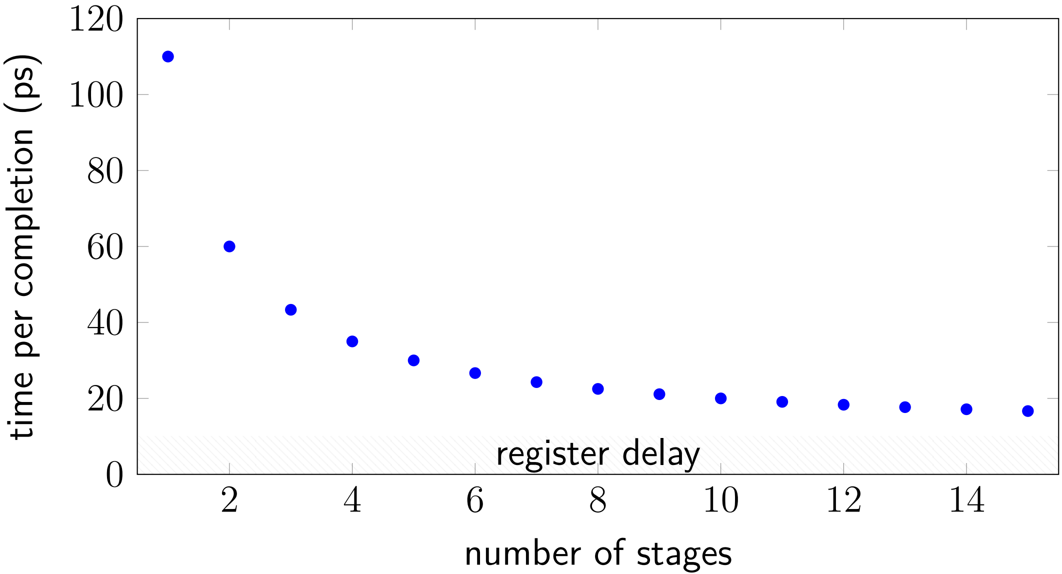
diminishing returns: register delays



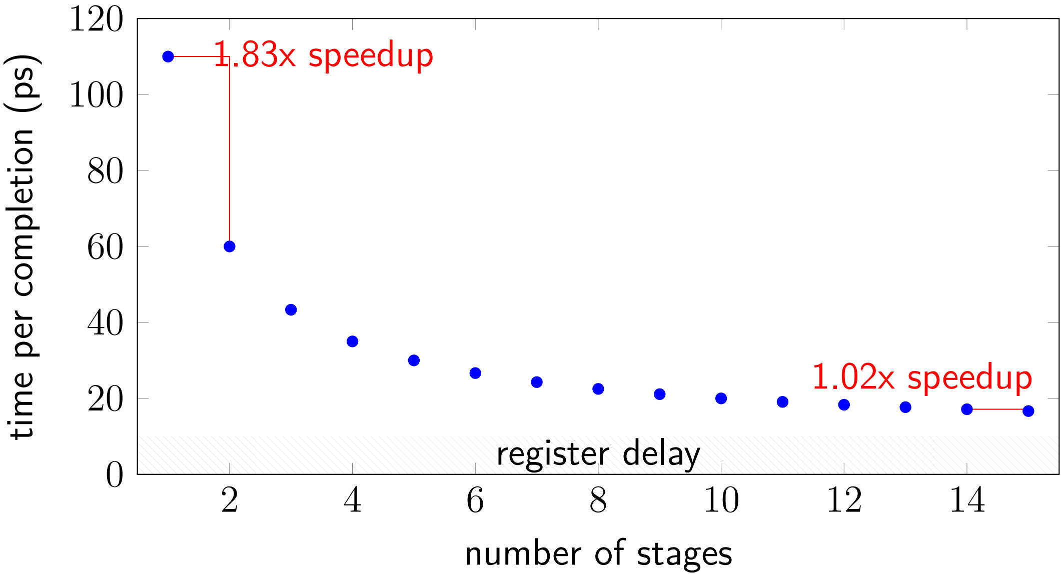
diminishing returns: register delays



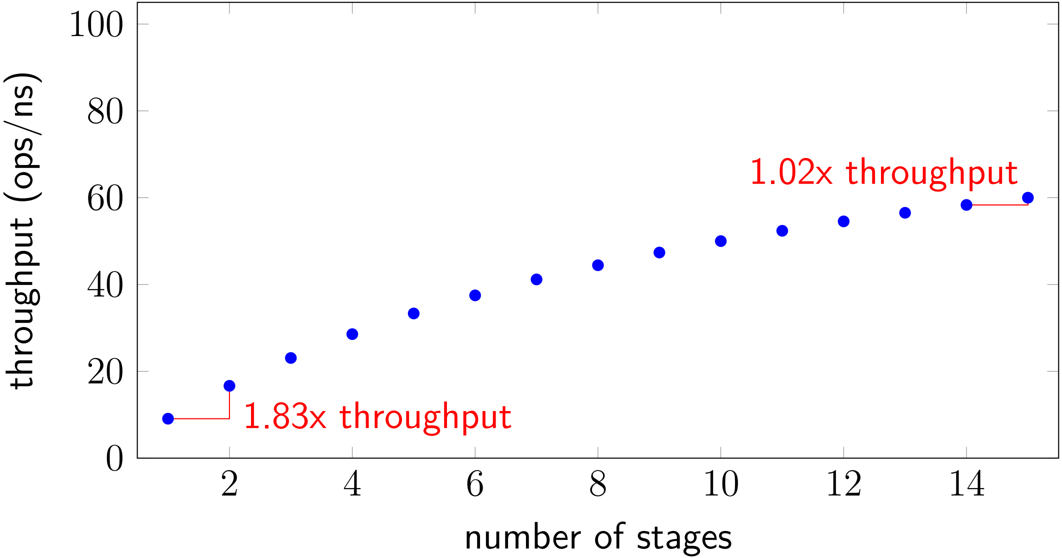
diminishing returns: register delays



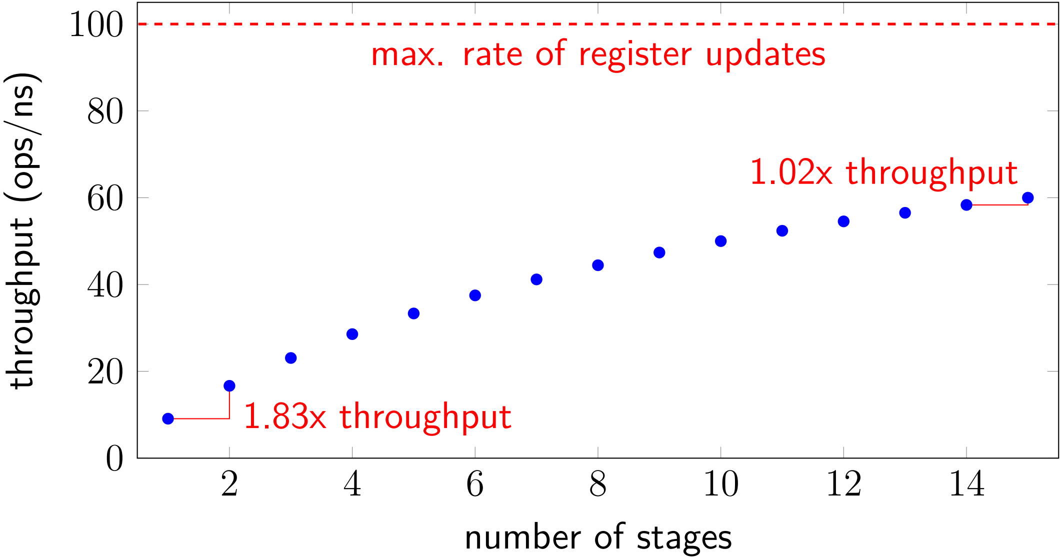
diminishing returns: register delays



diminishing returns: register delays



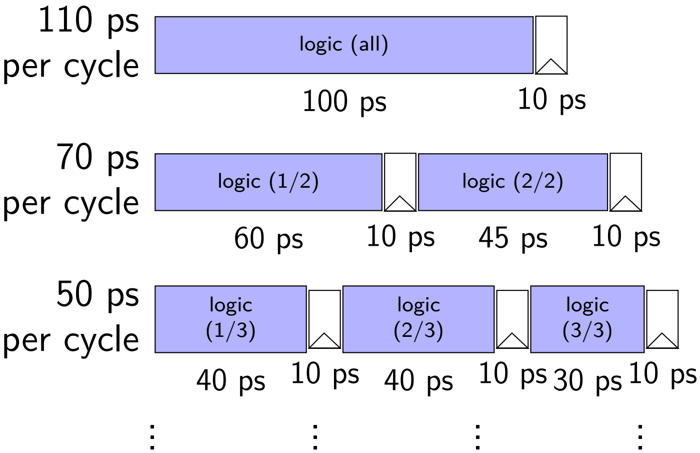
diminishing returns: register delays



diminishing returns: uneven split

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

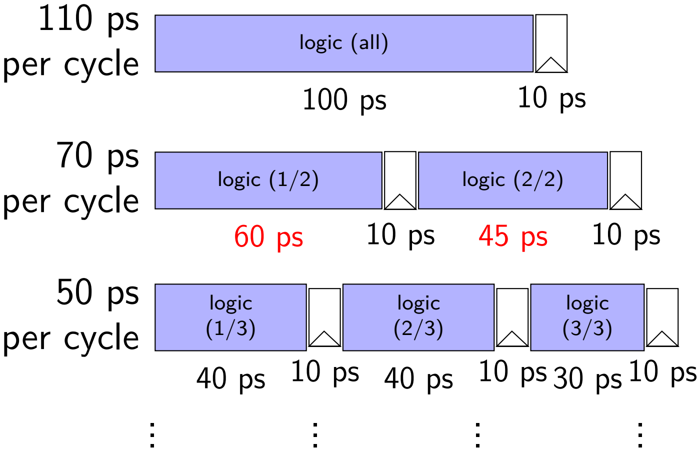
Probably not...



diminishing returns: uneven split

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

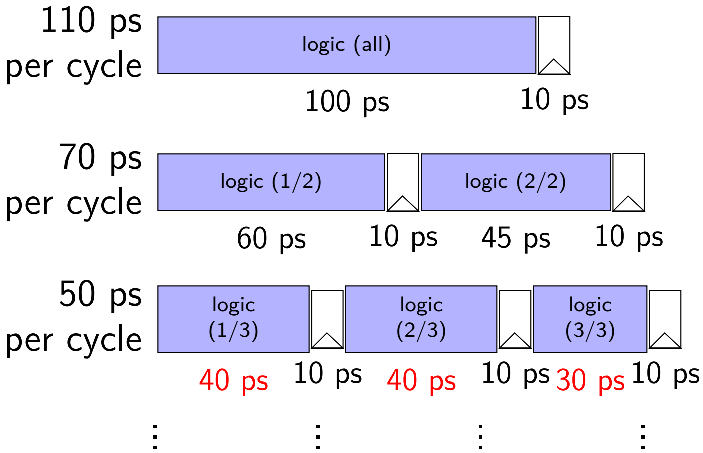
Probably not...



diminishing returns: uneven split

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

Probably not...



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

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

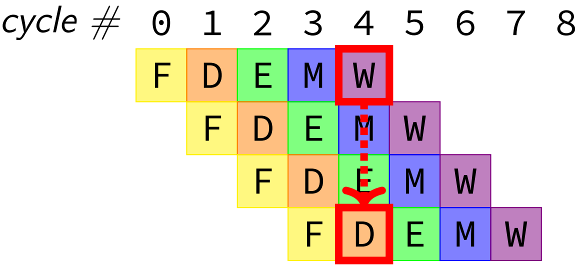
extra logic:

- sometimes don't change PC
- sometimes put do-nothing values in pipeline registers

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	

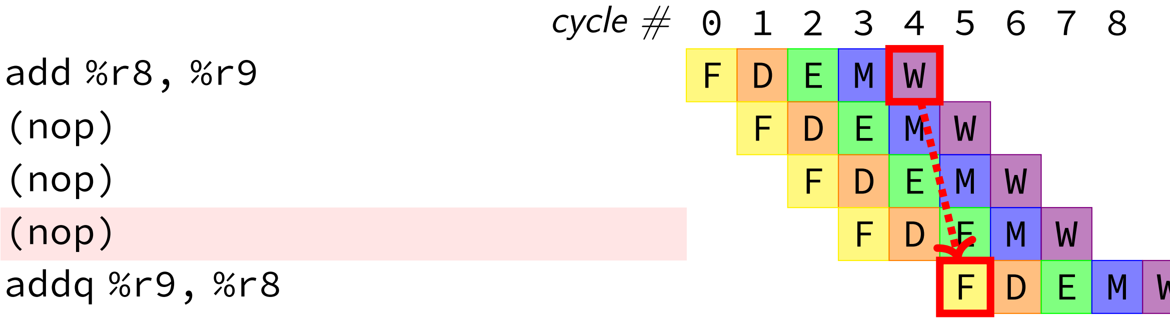
assumption:
if writing register value
register file will return that value for reads

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

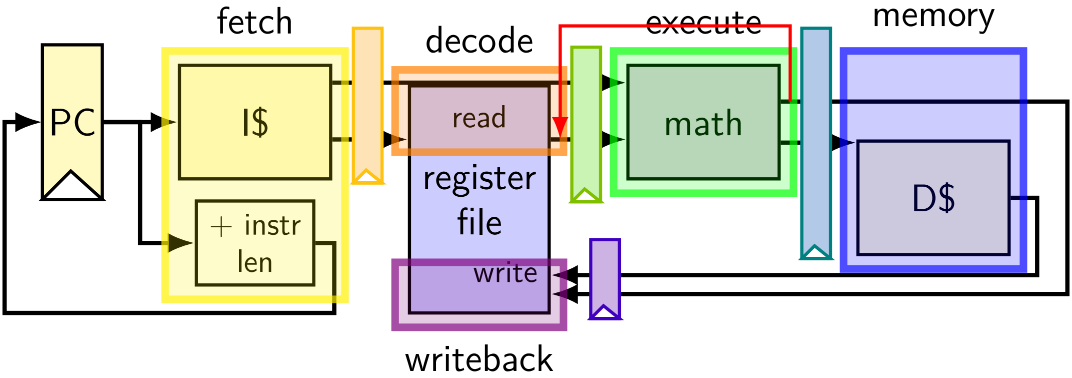
opportunity

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

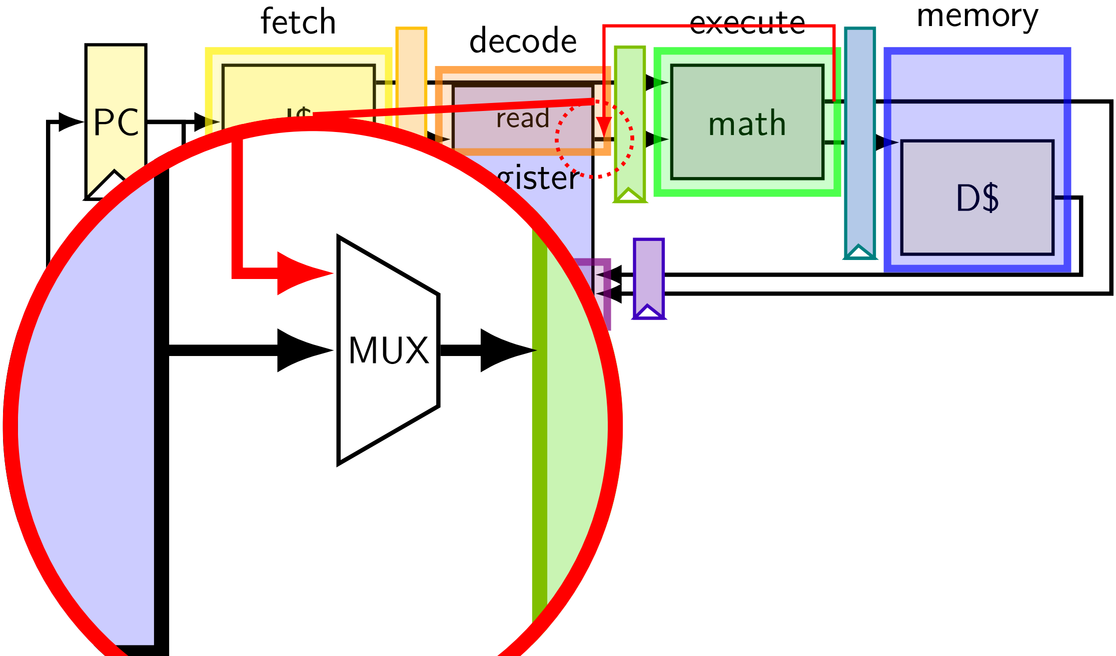
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



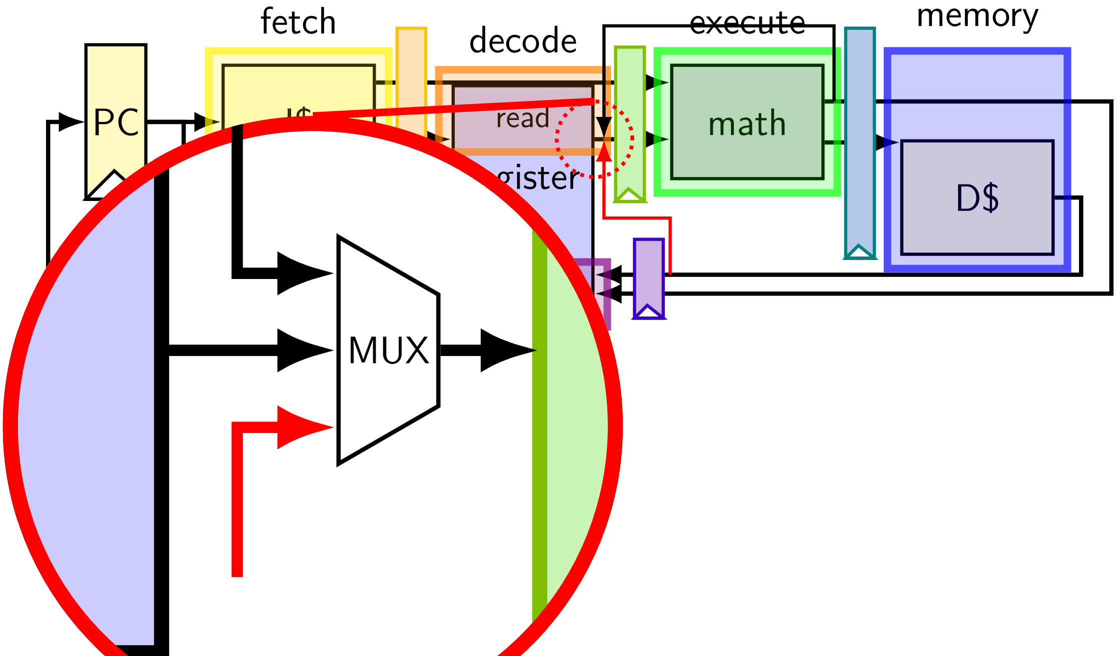
opportunity 2

```
// 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.

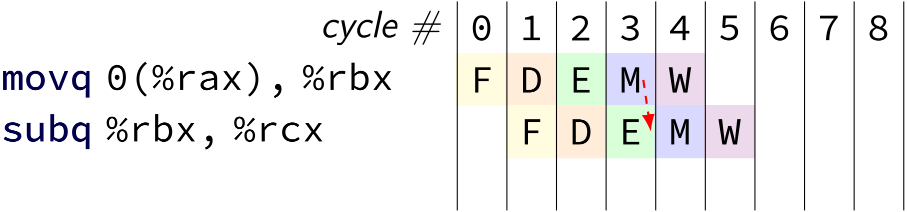
in xorq, %r9 is ________________ addq.

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



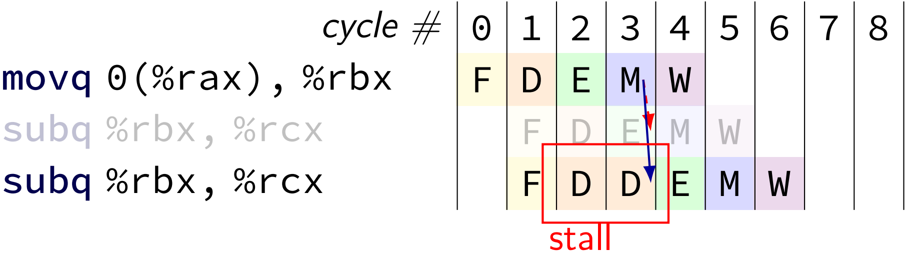
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	E	M	W			

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



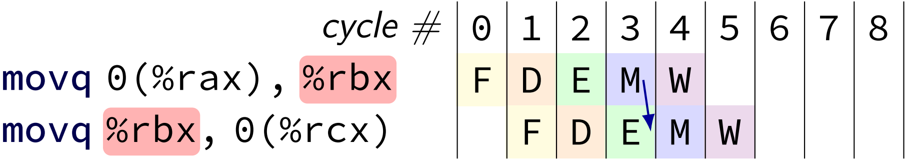
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	E	M	W			
`subq %rbx, %rcx`		F	D	D	E	M	W		

combine stalling and forwarding to resolve hazard

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

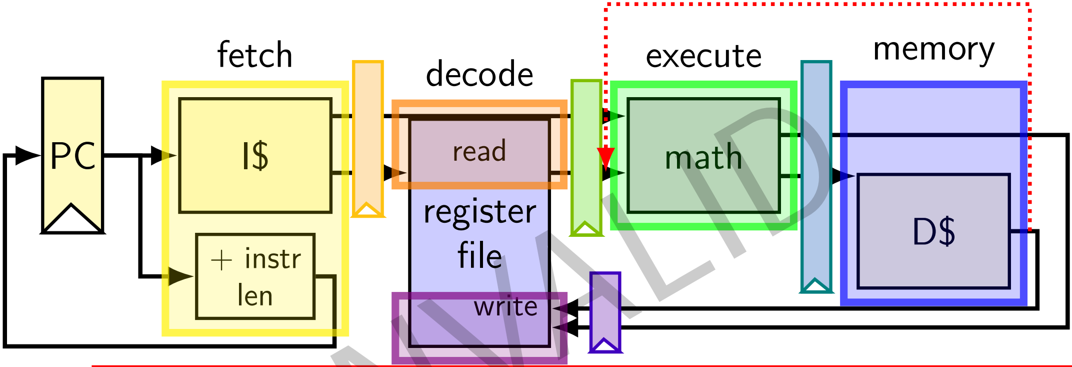
(since easier than detecting it in fetch stage)

solveable problem



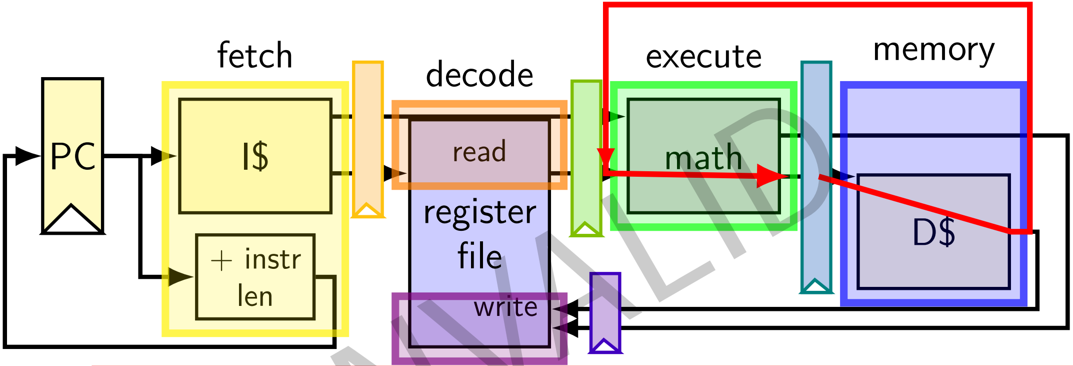
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`movq %rbx, 0(%rcx)`		F	D	E	M	W			

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)

hazards versus dependencies

dependency — X needs result of instruction Y?

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

hazard — will it not work in some pipeline?

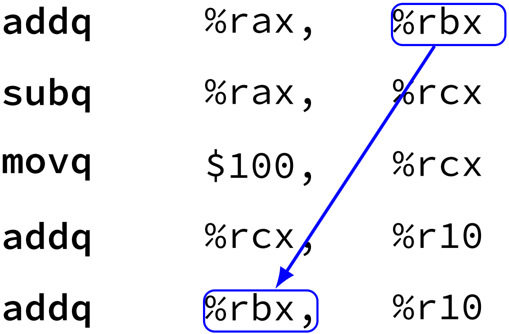
- before extra work is done to "resolve" hazards
- multiple kinds: so far, *data hazards*

ex.: dependencies and hazards (1)

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

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

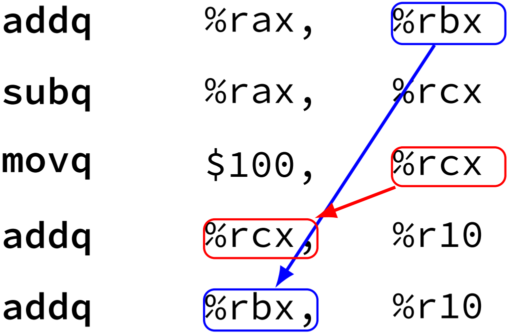
ex.: dependencies and hazards (1)



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

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

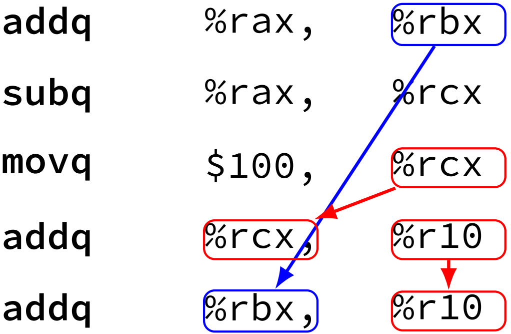
ex.: dependencies and hazards (1)



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

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

ex.: dependencies and hazards (1)



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

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

pipeline with different hazards

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

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

pipeline with different hazards

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

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

addq/andq is hazard with 5-stage pipeline

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

pipeline with different hazards

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

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

more hazards with more pipeline stages

exercise: different pipeline

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

result only available near end of second execute stage

where does forwarding, stalls occur?

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

exercise: different pipeline

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

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

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)
...
```

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	W
…										

jXX: stalling?

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

compare sets flags

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

jXX: stalling?

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

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	W
...										

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.

performance

hypothetical instruction mix

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

* — ignoring data hazards

performance

hypothetical instruction mix

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

* — ignoring data hazards

backup slides

key agreement and asym. encryption

can construct public-key encryption from key agreeement

private key: generated random value Y

public key: key share generated from that Y

key agreement and asym. encryption

can construct public-key encryption from key agreeement

private key: generated random value Y

public key: key share generated from that Y

PE(public key, message) $=$

- generate random value Z
- combine with public key to get shared secret
- use symmetric encryption $+$ MAC using shared secret as keys
- output: (key share generated from Z) (sym. encrypted data) (mac tag)

key agreement and asym. encryption

can construct public-key encryption from key agreeement

private key: generated random value Y

public key: key share generated from that Y

PE(public key, message) =

    generate random value Z
    combine with public key to get shared secret
    use symmetric encryption + MAC using shared secret as keys
    output: (key share generated from Z) (sym. encrypted data) (mac tag)

PD(private key, message) =

    extract (key share generated from Z)
    combine with private key to get shared secret, …

random numbers

need a lot of keys that no one else knows

common task: choose a *random* number

question: what does *random* mean here?

cryptographically secure random numbers

security properties we might want for random numbers:

attacker cannot guess (part of) number better than chance

knowing prior ‘random’ numbers shouldn’t help predict next ‘random’ numbers

compromising machine now shouldn’t reveal older random numbers

exercise: how to generate?

/dev/urandom

Linux kernel random number generator

collects “entropy” from hard-to-predict events

- e.g. exact timing of I/O interrupts
- e.g. some processor’s built-in random number circuit

turned into as many random bytes as you want

turning ‘entropy’ into random bytes

lots of ways to do this; one (rough/incomplete) idea:

internal variable *state*

to add ‘entropy’

state ← SecureHash(state + entropy)

to extract value:

random bytes ← SecureHash(1 + state)
give bytes that can’t be reversed to compute state

state ← SecureHash(2 + state)
change state so attacker can’t take us back to old state if compromised

things modern TLS usually does

(not all these properties provided by all TLS versions and modes)

confidentiality/authenticity

- server = one ID'd by certificate
- client = same throughout whole connection

forward secrecy

- can't decrypt old conversations (data for KeyShares is temporary)

fast

- most communication done with more efficient symmetric ciphers
- 1 set of messages back and forth to setup connection

denial of service (1)

so far: worried about network attacker disrupting confidentiality/authenticity

what if we're just worried about just breaking things

well, if they control network, nothing we can do...

but often worried about less

denial of service (2)

if you just want to inconvenience...

attacker just sends lots of stuff to my server

my server becomes overloaded?

my network becomes overloaded?

but: doesn't this require a lot of work for attacker?

exercise: why is this often not a big obstacle

denial of service: asymmetry

work for attacker $>$ work for defender

how much computation per message?

- complex search query?
- something that needs tons of memory?
- something that needs to read tons from disk?

how much sent back per message?

resources for attacker $>$ resources of defender

how many machines can attacker use?

denial of service: reflection/amplification

instead of sending messages directly…attacker can send messages "from" you to third-party

third-party sends back replies that overwhelm network

example: short DNS query with lots of things in response

"amplification" =

third-party inadvertantly turns small attack into big one

firewalls

don't want to expose network service to everyone?

solutions:

- service picky about who it accepts connections from
- filters in OS on machine with services
- filters on router

later two called "firewalls"

firewall rules examples?

ALLOW tcp port 443 (https) FROM everyone

ALLOW tcp port 22 (ssh) FROM my desktop's IP address

BLOCK tcp port 22 (ssh) FROM everyone else

ALLOW from address X to address Y

...

network security summary (1)

communicating securely with math

- secret value (shared key, public key) that attacker can't have
- symmetric: shared keys used for ed/encryption + auth/verify; fast
- asymmetric: public key used by any for encrypt + verify; slower
- asymmetric: private key used by holder for decrypt + sign; slower

protocol attacks — repurposing encrypt/signed/etc. messages

certificates — verifiable forwarded public keys

key agreement — for generated shared-secret "in public"

- publish key shares from private data
- combine private data with key share for shared secret

network security summary (2)

TLS: combine all cryptography stuff to make “secure channel”

denial-of-service — attacker just disrupts/overloads (not subtle)

firewalls

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