The death of optimizing compilers

Daniel J. Bernstein
University of Illinois at Chicago &
Technische Universiteit Eindhoven

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

(Donald E. Knuth,
"Structured programming
with go to statements", 1974)

th of optimizing compilers

. Bernstein

ty of Illinois at Chicago &

che Universiteit Eindhoven

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

(Donald E. Knuth,
"Structured programming
with go to statements", 1974)

The ove

Once up

CPUs w

Software

Software

hand-tu

mizing compilers

n

is at Chicago &

siteit Eindhoven

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

(Donald E. Knuth,
"Structured programming
with go to statements", 1974)

The oversimplified

Once upon a time
CPUs were painful
Software speed ma
Software was caref
hand-tuned in mac

mpilers

ago &
hoven

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

(Donald E. Knuth,
"Structured programming
with go to statements", 1974)

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine lang

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

(Donald E. Knuth, "Structured programming with go to statements", 1974)

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully hand-tuned in machine language.

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

(Donald E. Knuth, "Structured programming with go to statements", 1974)

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully hand-tuned in machine language.

Today:
CPUs are so fast that software speed is irrelevant.
"Unoptimized" is fast enough.
Programmers have stopped thinking about performance.
Compilers will do the same: easier to write, test, verify.

mmers waste enormous
s of time thinking about,
ying about, the speed
ritical parts of their
s, and these attempts at
y actually have a strong
impact when debugging
ntenance are considered.
uld forget about small
ies, say about 97% of
e; premature optimization
oot of all evil.

E. Knuth,
ured programming
to statements", 1974)

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine language.

Today:
CPUs are so fast that
software speed is irrelevant.
"Unoptimized" is fast enough.
Programmers have stopped
thinking about performance.
Compilers will do the same:
easier to write, test, verify.

The actu

Wait! It

Software

Users ar
for their

te enormous
hinking about,
, the speed
s of their
ese attempts at
have a strong
hen debugging
are considered.
about small
out 97% of
re optimization
vil.

”
amming
ents”, 1974)

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine language.

Today:
CPUs are so fast that
software speed is irrelevant.
“Unoptimized” is fast enough.
Programmers have stopped
thinking about performance.
Compilers will do the same:
easier to write, test, verify.

The actual story

Wait! It’s not tha

Software speed sti

Users are often wa
for their computer

us
bout,
d

ts at
rong
gging
ered.
all
of
ation

4)

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine language.

Today:
CPUs are so fast that
software speed is irrelevant.
"Unoptimized" is fast enough.
Programmers have stopped
thinking about performance.
Compilers will do the same:
easier to write, test, verify.

The actual story

Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine language.

Today:
CPUs are so fast that
software speed is irrelevant.
"Unoptimized" is fast enough.
Programmers have stopped
thinking about performance.
Compilers will do the same:
easier to write, test, verify.

The actual story

Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine language.

Today:
CPUs are so fast that
software speed is irrelevant.
"Unoptimized" is fast enough.
Programmers have stopped
thinking about performance.
Compilers will do the same:
easier to write, test, verify.

The actual story

Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow
computations, users are often
limiting what they compute.

The oversimplified story

Once upon a time:
CPUs were painfully slow.
Software speed mattered.
Software was carefully
hand-tuned in machine language.

Today:
CPUs are so fast that
software speed is irrelevant.
"Unoptimized" is fast enough.
Programmers have stopped
thinking about performance.
Compilers will do the same:
easier to write, test, verify.

The actual story

Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow
computations, users are often
limiting what they compute.

Example: In your favorite
sword-fighting video game,
are light reflections affected
realistically by sword vibration?

rsimplified story

on a time:

ere painfully slow.

e speed mattered.

e was carefully

hed in machine language.

re so fast that

speed is irrelevant.

mized” is fast enough.

nmers have stopped

about performance.

rs will do the same:

write, test, verify.

The actual story

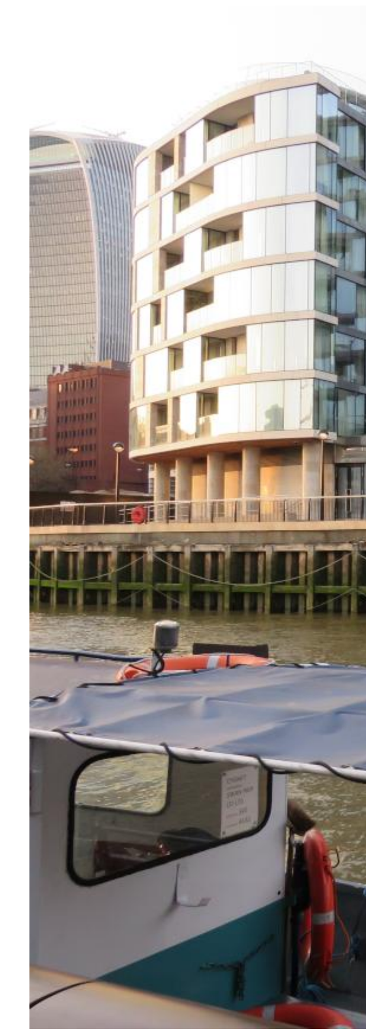
Wait! It’s not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow
computations, users are often
limiting what they compute.

Example: In your favorite
sword-fighting video game,
are light reflections affected
realistically by sword vibration?



story

:

lly slow.

attered.

fully

chine language.

that

rrelevant.

fast enough.

e stopped

rformance.

the same:

st, verify.

The actual story

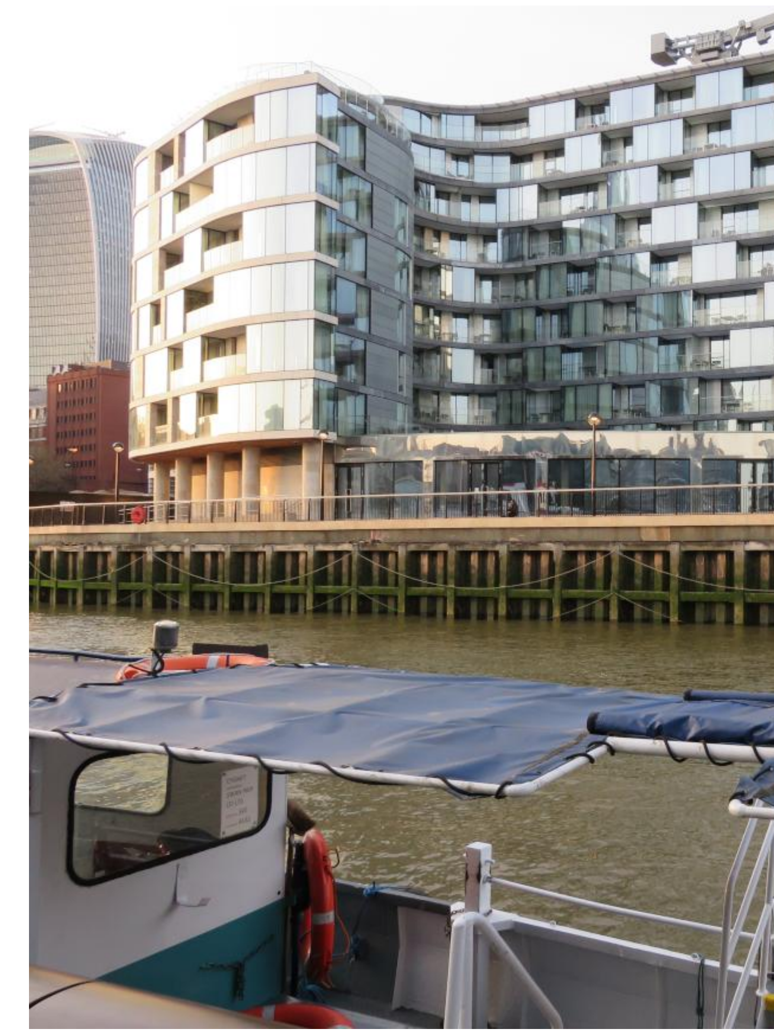
Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow computations, users are often limiting what they compute.

Example: In your favorite sword-fighting video game, are light reflections affected realistically by sword vibration?



uage.

gh.

The actual story

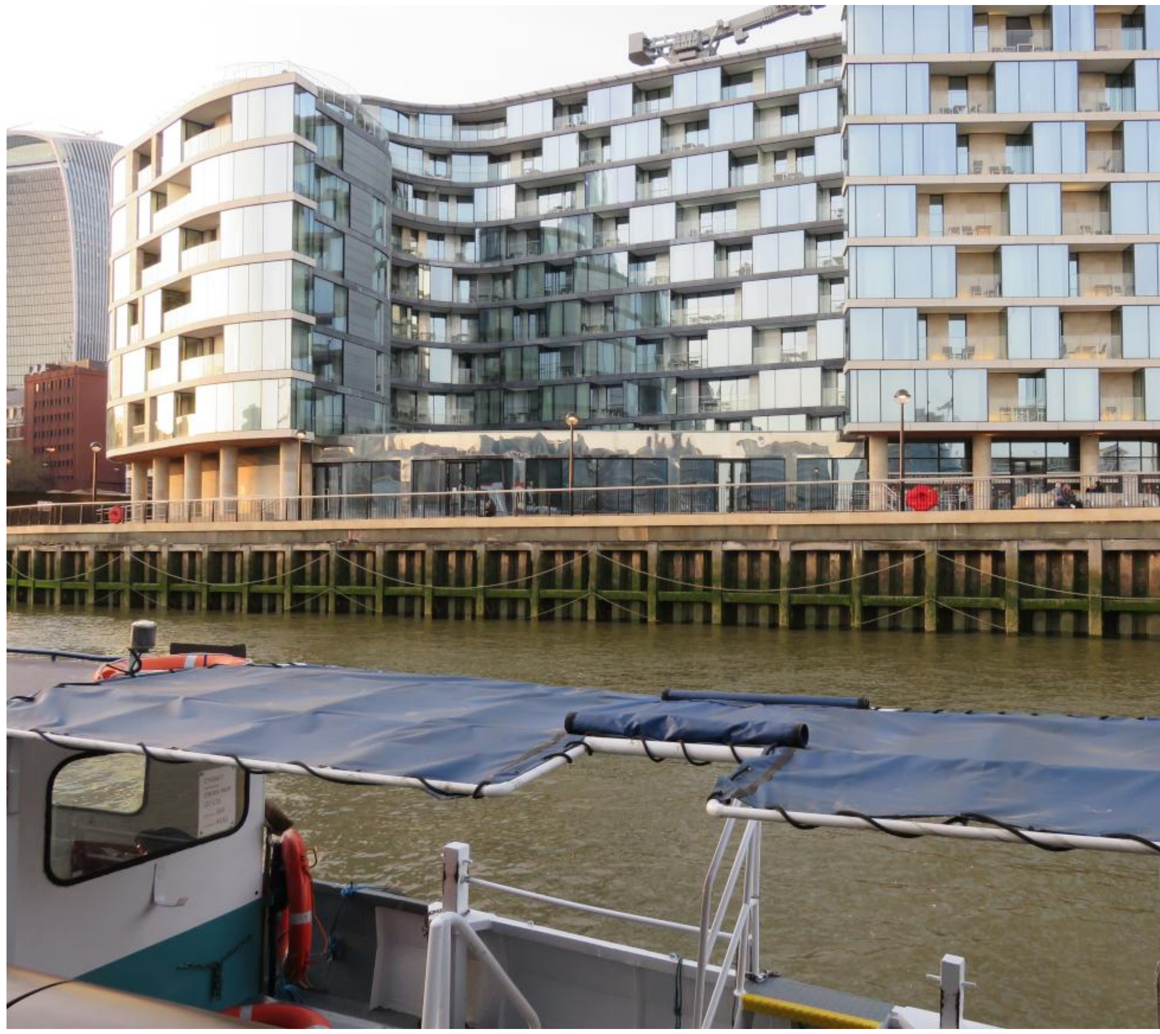
Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow
computations, users are often
limiting what they compute.

Example: In your favorite
sword-fighting video game,
are light reflections affected
realistically by sword vibration?



The actual story

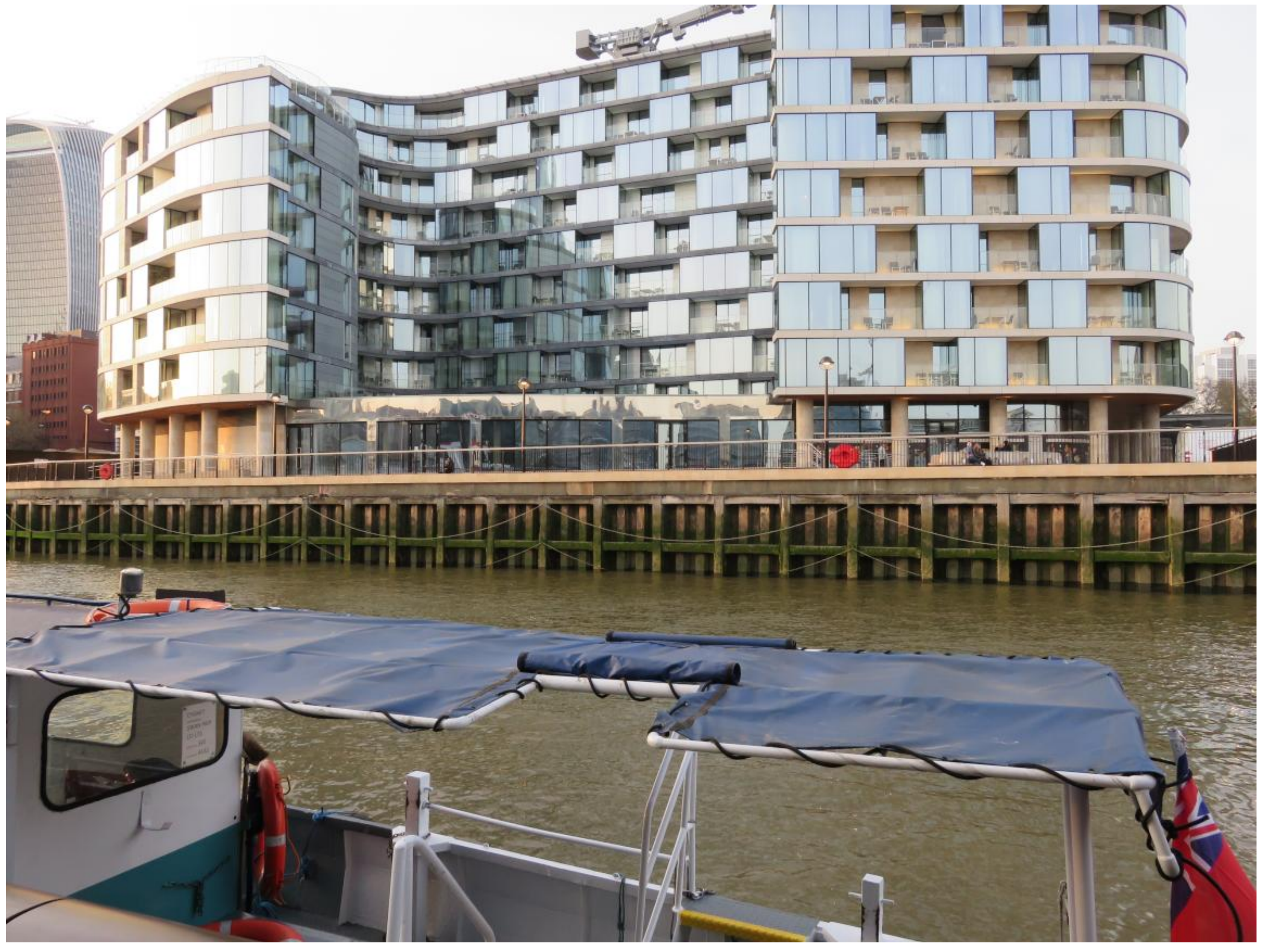
Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow computations, users are often limiting what they compute.

Example: In your favorite sword-fighting video game, are light reflections affected realistically by sword vibration?



The actual story

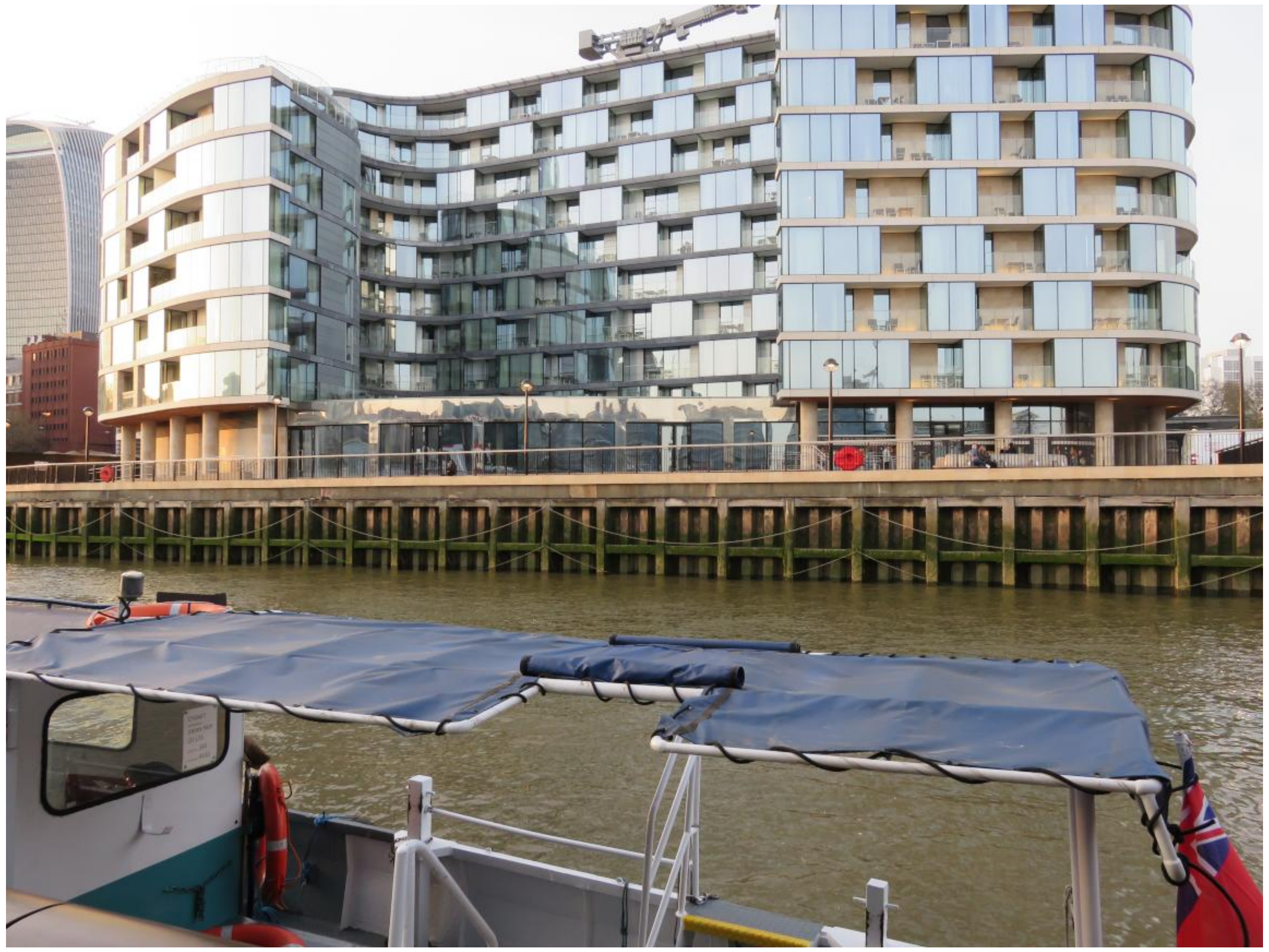
Wait! It's not that simple.

Software speed still matters.

Users are often waiting
for their computers.

To avoid unacceptably slow
computations, users are often
limiting what they compute.

Example: In your favorite
sword-fighting video game,
are light reflections affected
realistically by sword vibration?



ual story

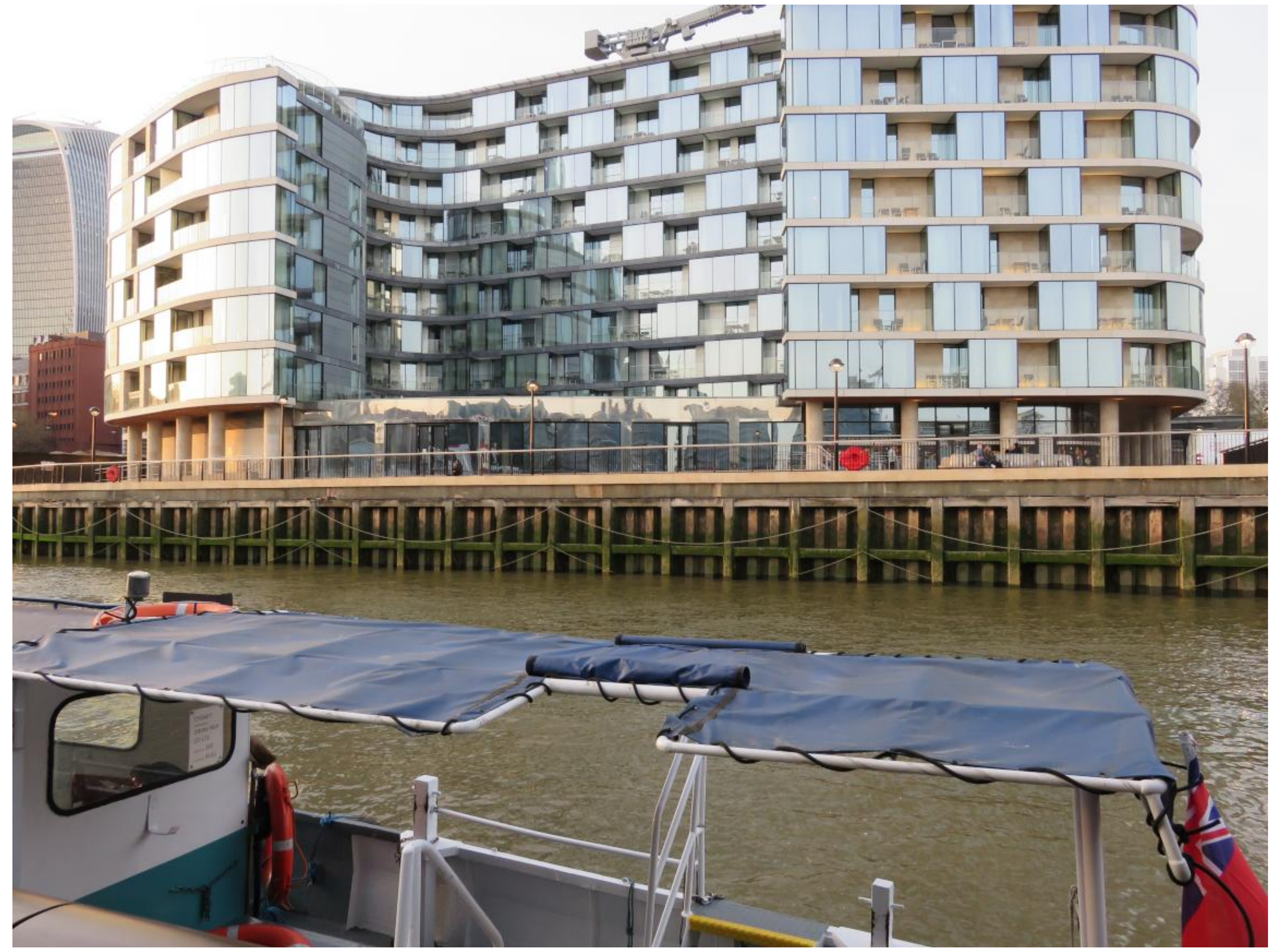
’s not that simple.

e speed still matters.

e often waiting
 computers.

d unacceptably slow
ations, users are often
what they compute.

e: In your favorite
ghting video game,
 reflections affected
ally by sword vibration?



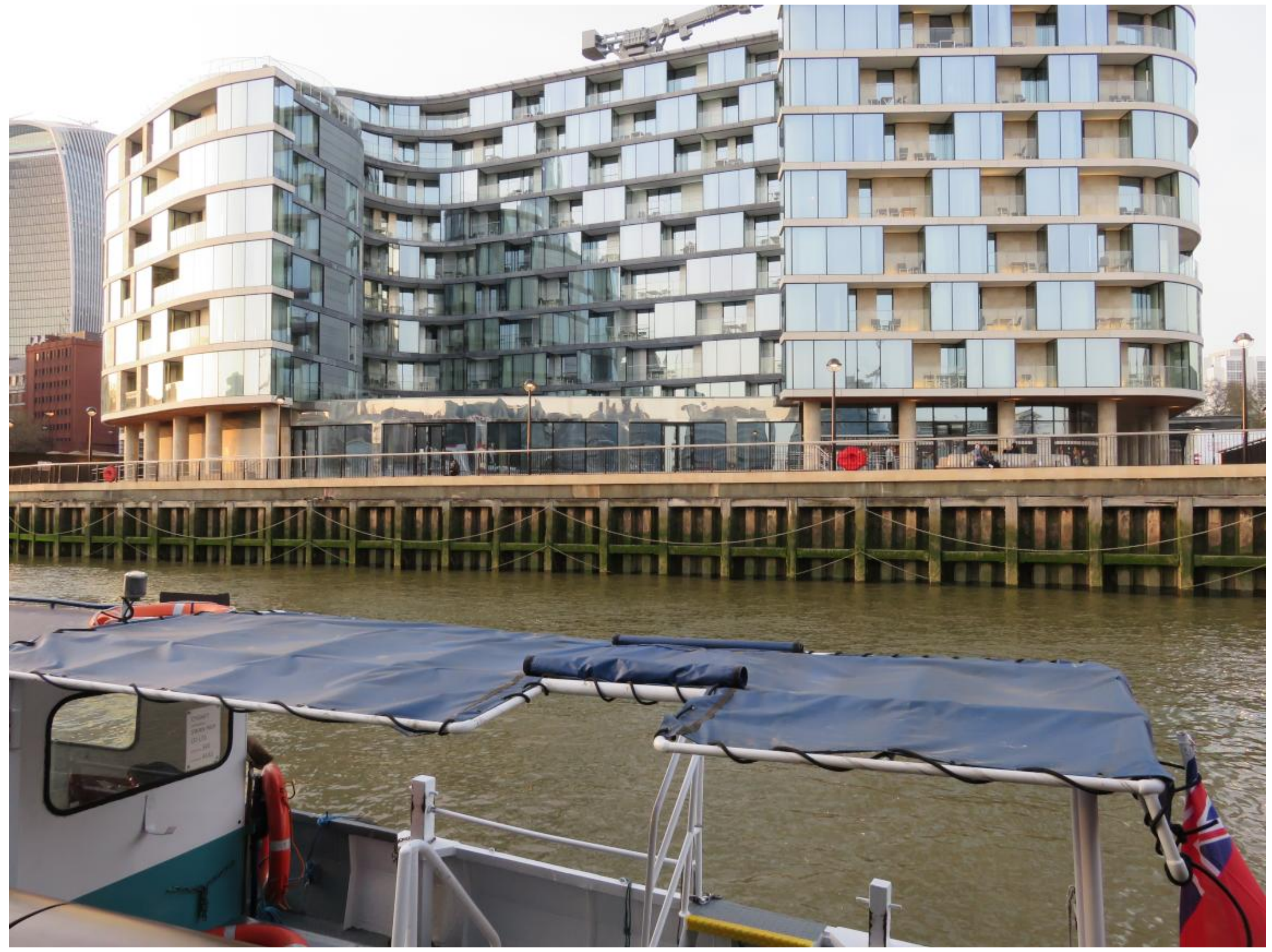
t simple.

ll matters.

iting
s.

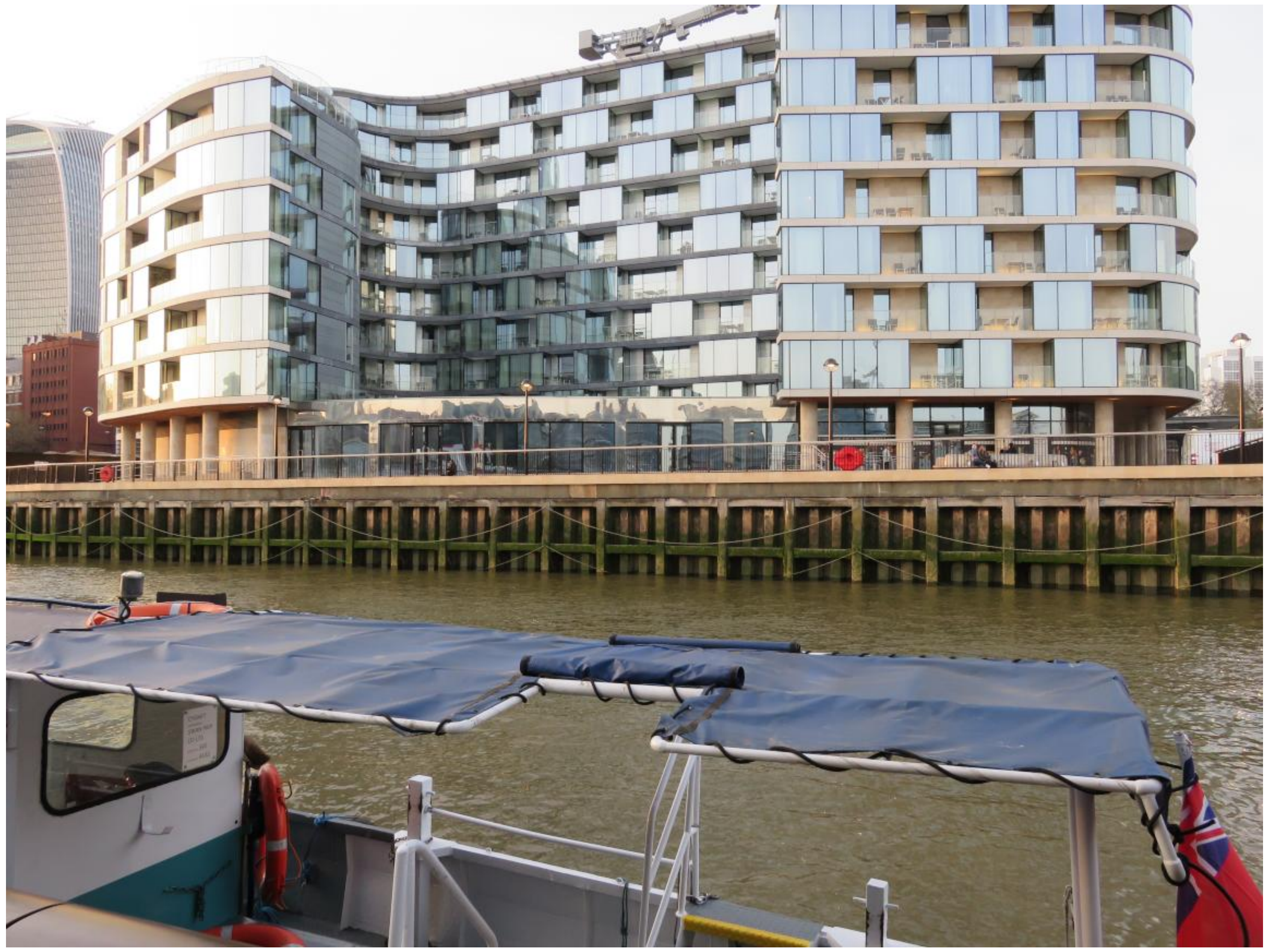
tably slow
rs are often
compute.

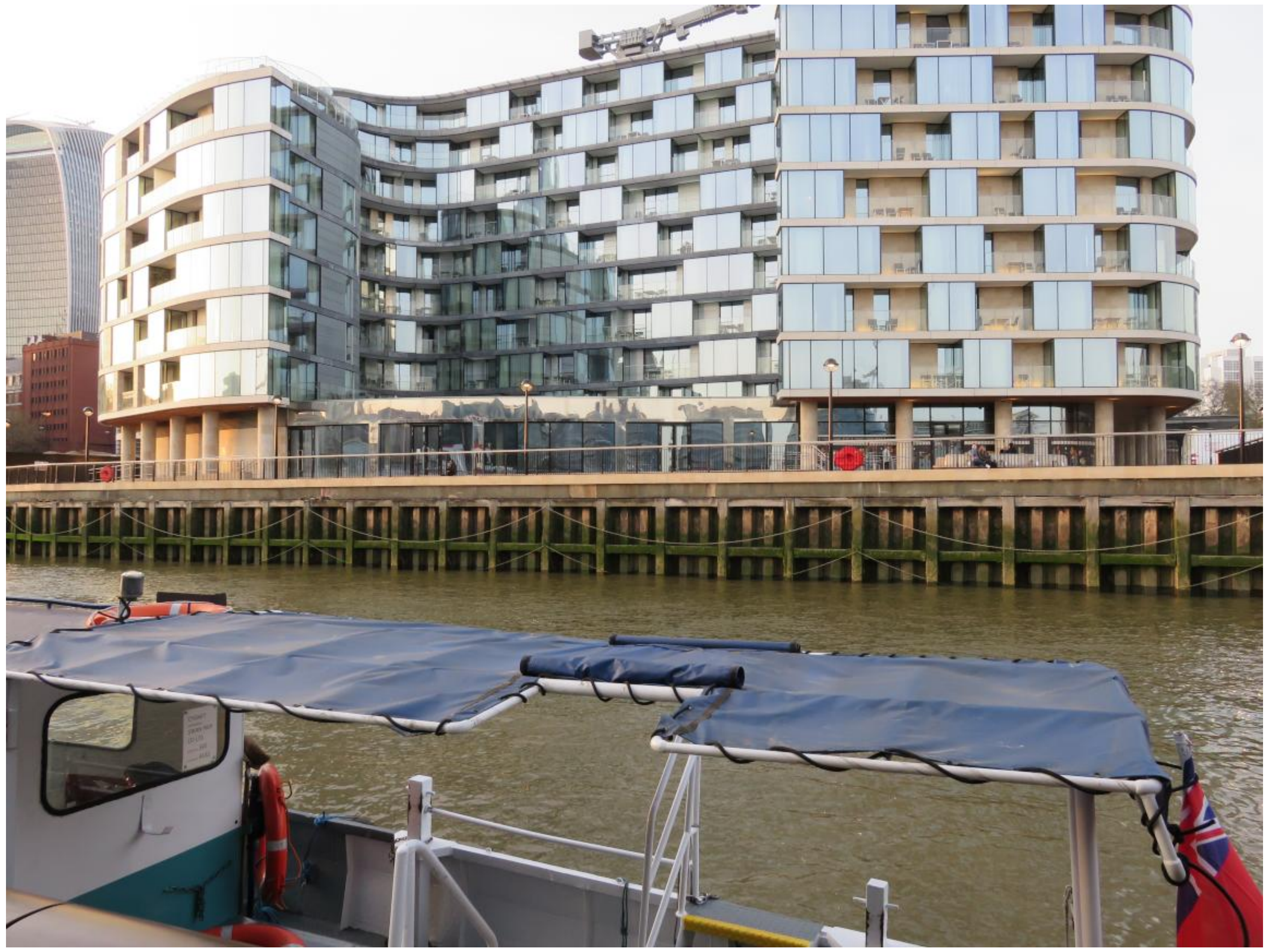
favorite
eo game,
s affected
ord vibration?

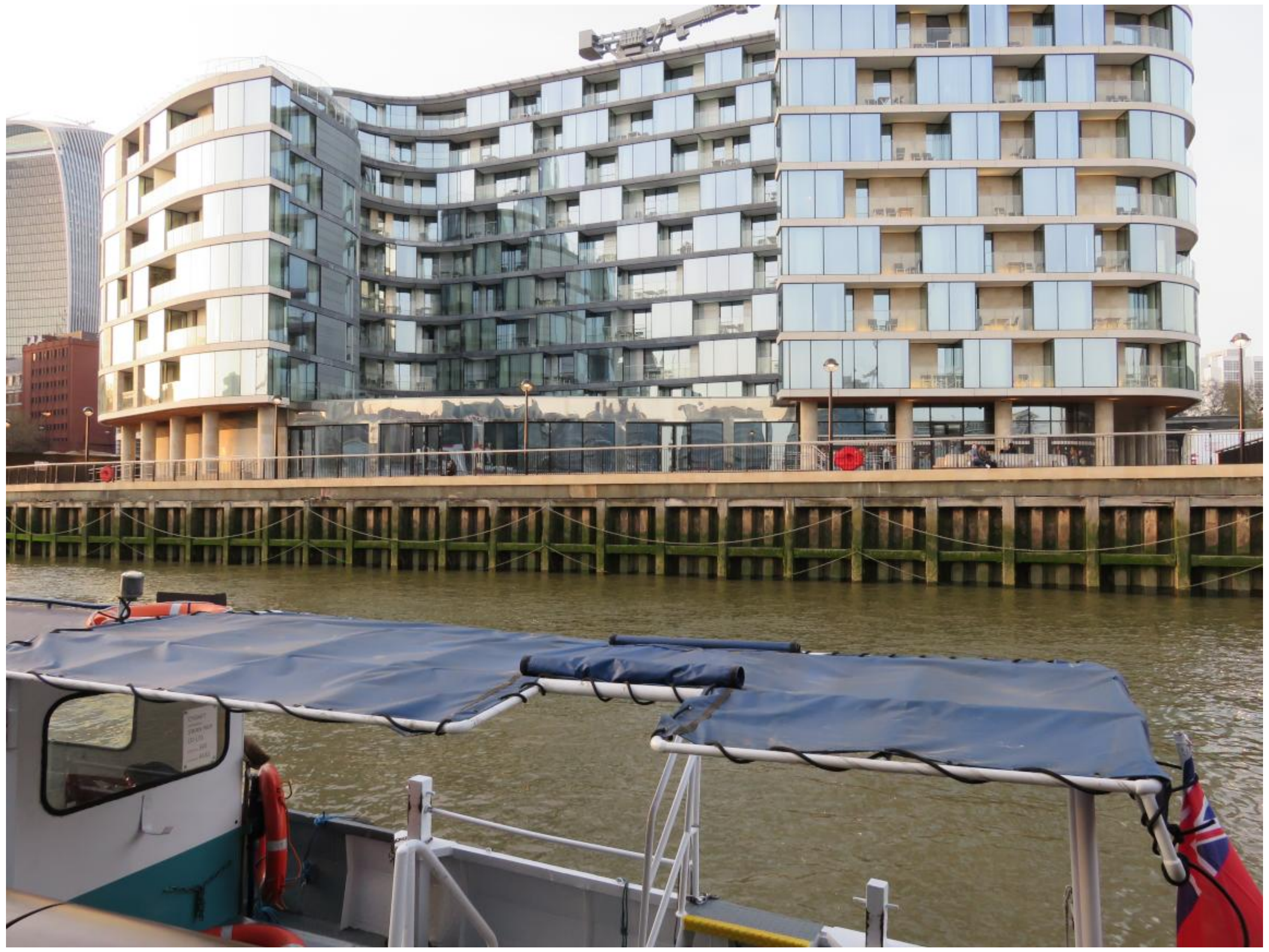


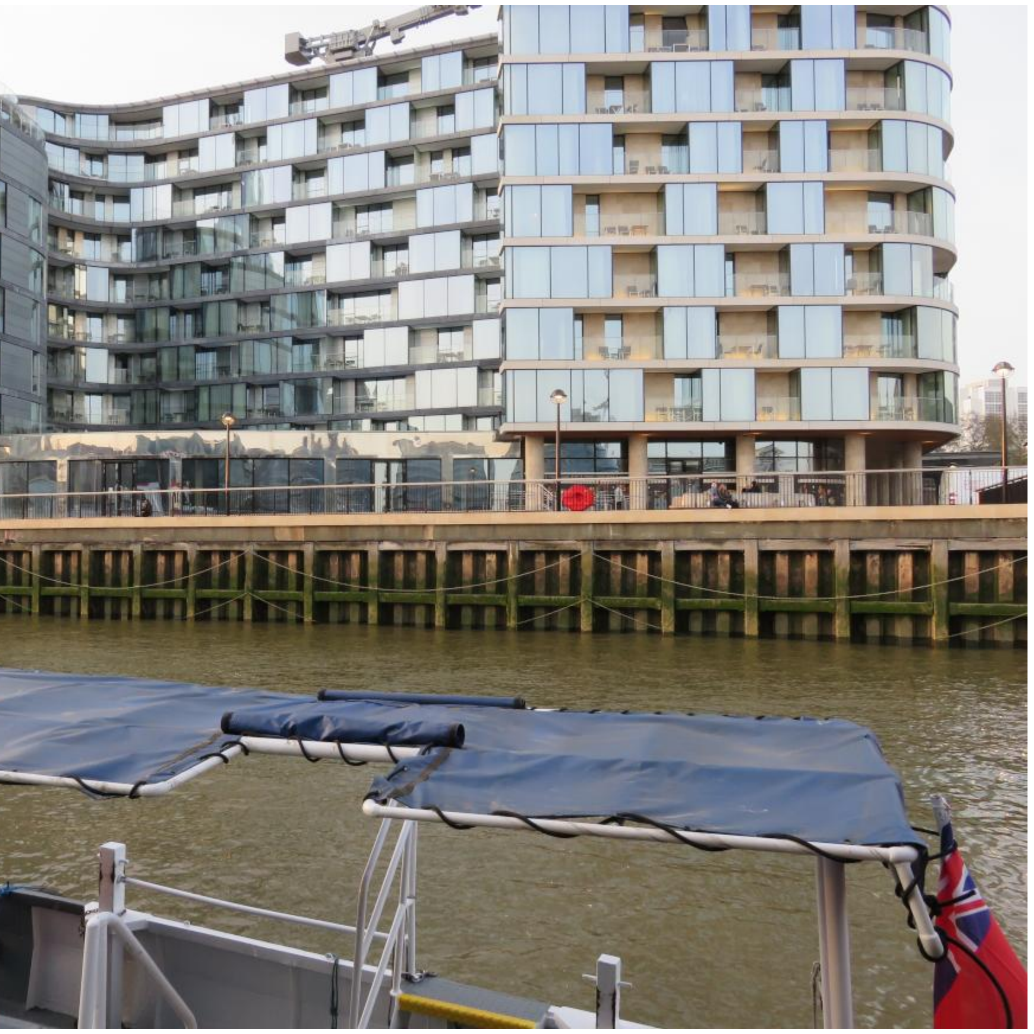
en

on?







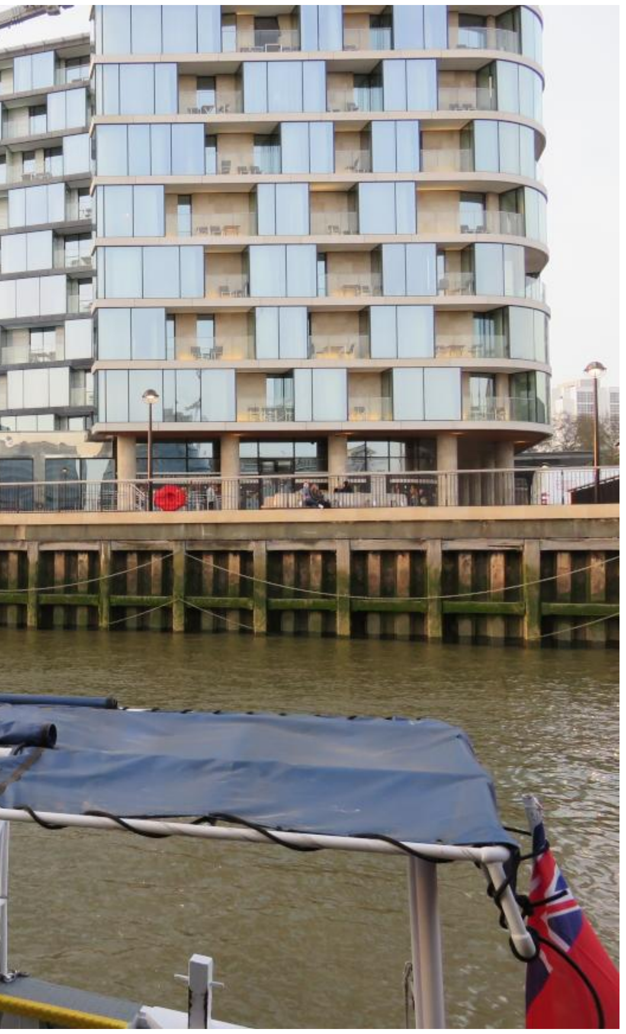
Old CPU

0ms: St

400ms:

1200ms:

1600ms:



Old CPU displayin

0ms: Start openin

400ms: Start disp

1200ms: Start cle

1600ms: Finish.




Old CPU displaying a file:

0ms: Start opening file.

400ms: Start displaying con

1200ms: Start cleaning up.

1600ms: Finish.

Old CPU displaying a file:

0ms: Start opening file.

400ms: Start displaying contents.

1200ms: Start cleaning up.

1600ms: Finish.

CPUs become faster:

0ms: Start opening file.

350ms: Start displaying contents.

1050ms: Start cleaning up.

1400ms: Finish.

CPUs become faster:

0ms: Start opening file.

300ms: Start displaying contents.

900ms: Start cleaning up.

1200ms: Finish.

CPUs become faster:

0ms: Start opening file.

250ms: Start displaying contents.

800ms: Start cleaning up.

1000ms: Finish.

CPUs become faster:

0ms: Start opening file.

200ms: Start displaying contents.

600ms: Start cleaning up.

800ms: Finish.

User displays bigger file:

0ms: Start opening file.

200ms: Start displaying contents.

1000ms: Start cleaning up.

1200ms: Finish.

CPUs become faster:

0ms: Start opening file.

175ms: Start displaying contents.

875ms: Start cleaning up.

1050ms: Finish.

CPUs become faster:

0ms: Start opening file.

150ms: Start displaying contents.

750ms: Start cleaning up.

900ms: Finish.

CPUs become faster:

0ms:	Start opening file.
125ms:	Start displaying contents.
625ms:	Start cleaning up.
750ms:	Finish.

CPUs become faster:

0ms: Start opening file.

100ms: Start displaying contents.

500ms: Start cleaning up.

600ms: Finish.

User displays bigger file:

0ms: Start opening file.
100ms: Start displaying contents.

900ms: Start cleaning up.
1000ms: Finish.

User displays bigger file:

100ms: Start displaying contents.

1000ms: Finish.

CPUs become faster:

87.5ms: Start displaying contents.

875ms: Finish.

CPUs become faster:

75.0ms: Start displaying contents.

750ms: Finish.

CPUs become faster:

62.5ms: Start displaying contents.

625ms: Finish.

CPUs become faster:

50ms: Start displaying contents.

500ms: Finish.

User displays bigger file:

50ms: Start displaying contents.

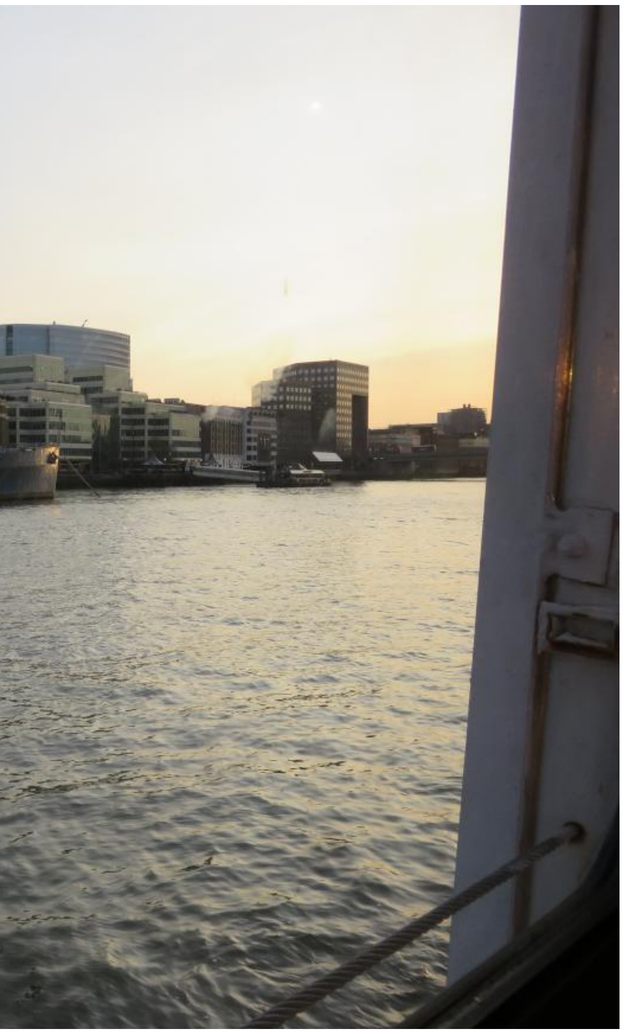
900ms: Finish.

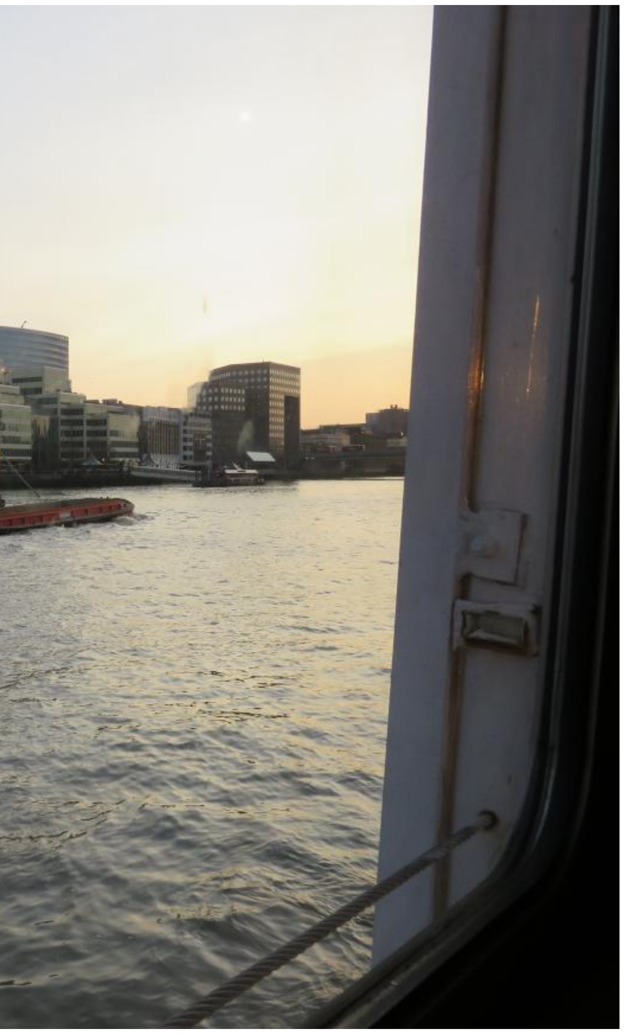
User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper
users pro





User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computat
users process more

User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computation $\Rightarrow$ users process more data.

User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computation $\Rightarrow$ users process more data.

User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computation $\Rightarrow$ users process more data.

Performance issues disappear for most operations. e.g. open file, clean up.

User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computation ⇒
users process more data.

Performance issues disappear
for most operations.
e.g. open file, clean up.

Inside the top operations:
Performance issues disappear
for most subroutines.

User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computation $\Rightarrow$
users process more data.

Performance issues disappear
for most operations.
e.g. open file, clean up.

Inside the top operations:
Performance issues disappear
for most subroutines.

Performance remains important
for occasional **hot spots**:
small segments of code
applied to tons of data.

splays bigger file:

---

tart displaying contents.

Finish.

Cheaper computation ⇒ users process more data.

Performance issues disappear for most operations. e.g. open file, clean up.

Inside the top operations: Performance issues disappear for most subroutines.

Performance remains important for occasional **hot spots**: small segments of code applied to tons of data.

"Except
applicati
mostly f
a lot of

ger file:

ying contents.

Cheaper computation ⇒
users process more data.

Performance issues disappear
for most operations.
e.g. open file, clean up.

Inside the top operations:
Performance issues disappear
for most subroutines.

Performance remains important
for occasional **hot spots**:
small segments of code
applied to tons of data.

"Except, uh, a lot
applications whose
mostly flat, becaus
a lot of time optin

ents.

Cheaper computation ⇒
users process more data.

Performance issues disappear
for most operations.
e.g. open file, clean up.

Inside the top operations:
Performance issues disappear
for most subroutines.

Performance remains important
for occasional **hot spots**:
small segments of code
applied to tons of data.

"Except, uh, a lot of people
applications whose profiles a
mostly flat, because they've
a lot of time optimizing ther

Cheaper computation $\Rightarrow$
users process more data.

Performance issues disappear
for most operations.
e.g. open file, clean up.

Inside the top operations:
Performance issues disappear
for most subroutines.

Performance remains important
for occasional **hot spots**:
small segments of code
applied to tons of data.

"Except, uh, a lot of people have applications whose profiles are mostly flat, because they've spent a lot of time optimizing them."

Cheaper computation $\Rightarrow$
users process more data.

Performance issues disappear
for most operations.
e.g. open file, clean up.

Inside the top operations:
Performance issues disappear
for most subroutines.

Performance remains important
for occasional **hot spots**:
small segments of code
applied to tons of data.

"Except, uh, a lot of people have
applications whose profiles are
mostly flat, because they've spent
a lot of time optimizing them."

— This view is obsolete.

Flat profiles are dying.
Already dead for most programs.
Larger and larger fraction
of code runs freezingly cold,
while hot spots run hotter.

Underlying phenomena:
Optimization tends to converge.
Data volume tends to diverge.

computation ⇒
ocess more data.

ance issues disappear
: operations.
n file, clean up.

ne top operations:
ance issues disappear
: subroutines.

ance remains important
sional **hot spots**:
gments of code
to tons of data.

"Except, uh, a lot of people have applications whose profiles are mostly flat, because they've spent a lot of time optimizing them."

— This view is obsolete.

Flat profiles are dying.
Already dead for most programs.
Larger and larger fraction
of code runs freezingly cold,
while hot spots run hotter.

Underlying phenomena:
Optimization tends to converge.
Data volume tends to diverge.

Speed m

2015.02.
"Do the
performa
(boldfac
Google s
major sit
supporte
Now all
support
Poly1305
than AE
devices.
decryptio
rendering

tion $\Rightarrow$
e data.

s disappear
ns.
n up.

rations:
s disappear
es.

ins important
spots:
code
data.

“Except, uh, a lot of people have applications whose profiles are mostly flat, because they’ve spent a lot of time optimizing them.”

— This view is obsolete.

Flat profiles are dying.
Already dead for most programs.
Larger and larger fraction of code runs freezingly cold, while hot spots run hotter.

Underlying phenomena:
Optimization tends to converge.
Data volume tends to diverge.

Speed matters: ar

2015.02.23 CloudF
“Do the ChaCha:
performance with
(boldface added):
Google services we
major sites on the
supported this new
Now all sites on C
support it, too. . .
Poly1305 is **three**
than AES-128-GC
devices. Spending
decryption means
rendering and bett

r

r

ant

“Except, uh, a lot of people have applications whose profiles are mostly flat, because they’ve spent a lot of time optimizing them.”

— This view is obsolete.

Flat profiles are dying.
Already dead for most programs.
Larger and larger fraction
of code runs freezingly cold,
while hot spots run hotter.

Underlying phenomena:
Optimization tends to converge.
Data volume tends to diverge.

Speed matters: an example

2015.02.23 CloudFlare blog
“Do the ChaCha: better mo
performance with cryptograp
(boldface added): “Until tod
Google services were the onl
major sites on the Internet t
supported this new algorithm
Now all sites on CloudFlare
support it, too. . . . ChaCha2
Poly1305 is **three times fas**
than AES-128-GCM on mob
devices. Spending less time
decryption means faster pag
rendering and better battery

"Except, uh, a lot of people have applications whose profiles are mostly flat, because they've spent a lot of time optimizing them."

— This view is obsolete.

Flat profiles are dying.
Already dead for most programs.
Larger and larger fraction of code runs freezingly cold, while hot spots run hotter.

Underlying phenomena:
Optimization tends to converge.
Data volume tends to diverge.

Speed matters: an example

2015.02.23 CloudFlare blog post "Do the ChaCha: better mobile performance with cryptography" (boldface added): "Until today, Google services were the only major sites on the Internet that supported this new algorithm. Now all sites on CloudFlare support it, too. . . . ChaCha20-Poly1305 is **three times faster** than AES-128-GCM on mobile devices. Spending less time on decryption means faster page rendering and better battery life."

, uh, a lot of people have
ions whose profiles are
lat, because they've spent
time optimizing them."

view is obsolete.

files are dying.
dead for most programs.
nd larger fraction
runs freezingly cold,
t spots run hotter.

ng phenomena:
ation tends to converge.
lume tends to diverge.

Speed matters: an example

2015.02.23 CloudFlare blog post "Do the ChaCha: better mobile performance with cryptography" (boldface added): "Until today, Google services were the only major sites on the Internet that supported this new algorithm. Now all sites on CloudFlare support it, too. . . . ChaCha20-Poly1305 is **three times faster** than AES-128-GCM on mobile devices. Spending less time on decryption means faster page rendering and better battery life."

What ab

CloudFla
"In orde
HTTPS
we have
usage is
performa
source **a**
of ChaC
engineer
that has
servers'
the cost
this new

of people have
e profiles are
se they've spent
nizing them."

solete.

ving.
most programs.
fraction
ingly cold,
n hotter.

mena:
s to converge.
s to diverge.

Speed matters: an example

2015.02.23 CloudFlare blog post "Do the ChaCha: better mobile performance with cryptography" (boldface added): "Until today, Google services were the only major sites on the Internet that supported this new algorithm. Now all sites on CloudFlare support it, too. . . . ChaCha20-Poly1305 is **three times faster** than AES-128-GCM on mobile devices. Spending less time on decryption means faster page rendering and better battery life."

What about the se

CloudFlare blog po
"In order to suppo
HTTPS sites on o
we have to make s
usage is low. To h
performance we ar
source **assembly c**
of ChaCha/Poly b
engineer Vlad Kra
that has been **opt**
servers' Intel CP
the cost of encryp
this new cipher to

have
re
spent
m."

ams.

erge.
ge.

Speed matters: an example

2015.02.23 CloudFlare blog post "Do the ChaCha: better mobile performance with cryptography" (boldface added): "Until today, Google services were the only major sites on the Internet that supported this new algorithm. Now all sites on CloudFlare support it, too. . . . ChaCha20-Poly1305 is **three times faster** than AES-128-GCM on mobile devices. Spending less time on decryption means faster page rendering and better battery life."

What about the servers?

CloudFlare blog post, contin
"In order to support over a
HTTPS sites on our servers,
we have to make sure CPU
usage is low. To help impro
performance we are using ar
source **assembly code versi**
of ChaCha/Poly by CloudFla
engineer Vlad Krasnov and
that has been **optimized fo**
servers' Intel CPUs. This
the cost of encrypting data
this new cipher to a minimu

Speed matters: an example

2015.02.23 CloudFlare blog post "Do the ChaCha: better mobile performance with cryptography" (boldface added): "Until today, Google services were the only major sites on the Internet that supported this new algorithm. Now all sites on CloudFlare support it, too. . . . ChaCha20-Poly1305 is **three times faster** than AES-128-GCM on mobile devices. Spending less time on decryption means faster page rendering and better battery life."

What about the servers?

CloudFlare blog post, continued: "In order to support over a million HTTPS sites on our servers, we have to make sure CPU usage is low. To help improve performance we are using an open source **assembly code version** of ChaCha/Poly by CloudFlare engineer Vlad Krasnov and others that has been **optimized for our servers' Intel CPUs**. This keeps the cost of encrypting data with this new cipher to a minimum."

natters: an example

123 CloudFlare blog post
ChaCha: better mobile
ance with cryptography"
e added): "Until today,
services were the only
tes on the Internet that
ed this new algorithm.
sites on CloudFlare
it, too. ... ChaCha20-
5 is **three times faster**
S-128-GCM on mobile
Spending less time on
on means faster page
g and better battery life."

What about the servers?

CloudFlare blog post, continued: "In order to support over a million HTTPS sites on our servers, we have to make sure CPU usage is low. To help improve performance we are using an open source **assembly code version** of ChaCha/Poly by CloudFlare engineer Vlad Krasnov and others that has been **optimized for our servers' Intel CPUs**. This keeps the cost of encrypting data with this new cipher to a minimum."

Typical c
inner loo

```
vpaddd $
vpxor $a
vpshufb
vpaddd $
vpxor $c
vpslld $
vpsrld $
vpxor $f
```

Mobile c
heavy ve

n example

Flare blog post
better mobile
cryptography”
“Until today,
ere the only
Internet that
v algorithm.
loudFlare
. ChaCha20-
times faster
M on mobile
less time on
faster page
ter battery life.”

What about the servers?

CloudFlare blog post, continued: “In order to support over a million HTTPS sites on our servers, we have to make sure CPU usage is low. To help improve performance we are using an open source **assembly code version** of ChaCha/Poly by CloudFlare engineer Vlad Krasnov and others that has been **optimized for our servers’ Intel CPUs**. This keeps the cost of encrypting data with this new cipher to a minimum.”

Typical excerpt fro
inner loop of serve

```
vpaddd $b, $a, $
vpxor $a, $d, $d
vpshufb .rol16(%
vpaddd $d, $c, $
vpxor $c, $b, $b
vpslld \$12, $b,
vpsrld \$20, $b,
vpxor $tmp, $b,
```

Mobile code simila
heavy vectorizatio

post
obile
phy"
day,
y
that
m.

20-
ster
bile
on
e
life."

What about the servers?

CloudFlare blog post, continued: "In order to support over a million HTTPS sites on our servers, we have to make sure CPU usage is low. To help improve performance we are using an open source **assembly code version** of ChaCha/Poly by CloudFlare engineer Vlad Krasnov and others that has been **optimized for our servers' Intel CPUs**. This keeps the cost of encrypting data with this new cipher to a minimum."

Typical excerpt from inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d,
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has heavy vectorization + asm.

What about the servers?

CloudFlare blog post, continued: "In order to support over a million HTTPS sites on our servers, we have to make sure CPU usage is low. To help improve performance we are using an open source **assembly code version** of ChaCha/Poly by CloudFlare engineer Vlad Krasnov and others that has been **optimized for our servers' Intel CPUs**. This keeps the cost of encrypting data with this new cipher to a minimum."

Typical excerpt from inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has heavy vectorization + asm.

bout the servers?

are blog post, continued:
r to support over a million
sites on our servers,
to make sure CPU
low. To help improve
ance we are using an open
ssembly code version
ha/Poly by CloudFlare
Vlad Krasnov and others
been **optimized for our**
Intel CPUs. This keeps
of encrypting data with
cipher to a minimum."

Typical excerpt from
inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has
heavy vectorization + asm.

Hand-tu

Wikiped
even per
optimizi
performa

ervers?

ost, continued:

rt over a million

ur servers,

sure CPU

elp improve

e using an open

code version

y CloudFlare

snov and others

imized for our

Us. This keeps

ting data with

a minimum."

Typical excerpt from inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has heavy vectorization + asm.

Hand-tuned? In 2

Wikipedia: "By th

even performance

optimizing compile

performance of hu

ued:
million
,
ve
n open
ion
are
others
r our
keeps
with
m.”

Typical excerpt from
inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has
heavy vectorization + asm.

Hand-tuned? In 2015? Serio

Wikipedia: “By the late 199
even performance sensitive c
optimizing compilers exceed
performance of human expe

Typical excerpt from inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has heavy vectorization + asm.

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

Typical excerpt from inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has heavy vectorization + asm.

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

— [citation needed]

Typical excerpt from
inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has
heavy vectorization + asm.

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

— The experts disagree,
and hold the speed records.

Typical excerpt from
inner loop of server code:

```
vpaddd $b, $a, $a
vpxor $a, $d, $d
vpshufb .rol16(%rip), $d, $d
vpaddd $d, $c, $c
vpxor $c, $b, $b
vpslld \$12, $b, $tmp
vpsrld \$20, $b, $b
vpxor $tmp, $b, $b
```

Mobile code similarly has heavy vectorization + asm.

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

— The experts disagree, and hold the speed records.

Mike Pall, LuaJIT author, 2011: "If you write an interpreter loop in assembler, you can do much better . . . There's just no way you can reasonably expect even the most advanced C compilers to do this on your behalf."

excerpt from
op of server code:

```
$b, $a, $a
a, $d, $d
 .rol16(%rip), $d, $d
$d, $c, $c
c, $b, $b
\$12, $b, $tmp
\$20, $b, $b
tmp, $b, $b
```

code similarly has
ectorization + asm.

Hand-tuned? In 2015? Seriously?

Wikipedia: “By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts.”

— The experts disagree, and hold the speed records.

Mike Pall, LuaJIT author, 2011: “If you write an interpreter loop in assembler, you can do much better … There’s just no way you can reasonably expect even the most advanced C compilers to do this on your behalf.”

— “We
on most
can’t do
NP com
of heuris
get little
where th
wrong a

om
er code:

a

```
rip), $d, $d
c

 $tmp
 $b
$b
```

arly has
n + asm.

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

— The experts disagree, and hold the speed records.

Mike Pall, LuaJIT author, 2011: "If you write an interpreter loop in assembler, you can do much better . . . There's just no way you can reasonably expect even the most advanced C compilers to do this on your behalf."

— "We come so c
on most architectu
can't do much mo
NP complete algo
of heuristics. We
get little niggles h
where the heuristi
wrong answers."

$d

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

— The experts disagree, and hold the speed records.

Mike Pall, LuaJIT author, 2011: "If you write an interpreter loop in assembler, you can do much better . . . There's just no way you can reasonably expect even the most advanced C compilers to do this on your behalf."

— "We come so close to op
on most architectures that w
can't do much more without
NP complete algorithms inst
of heuristics. We can only t
get little niggles here and th
where the heuristics get sligl
wrong answers."

Hand-tuned? In 2015? Seriously?

Wikipedia: “By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts.”

— The experts disagree, and hold the speed records.

Mike Pall, LuaJIT author, 2011: “If you write an interpreter loop in assembler, you can do much better . . . There’s just no way you can reasonably expect even the most advanced C compilers to do this on your behalf.”

— “We come so close to optimal on most architectures that we can’t do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers.”

Hand-tuned? In 2015? Seriously?

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

— The experts disagree, and hold the speed records.

Mike Pall, LuaJIT author, 2011: "If you write an interpreter loop in assembler, you can do much better . . . There's just no way you can reasonably expect even the most advanced C compilers to do this on your behalf."

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

ned? In 2015? Seriously?

ia: "By the late 1990s for
rformance sensitive code,
ng compilers exceeded the
ance of human experts."

experts disagree,
l the speed records.

ll, LuaJIT author, 2011:
write an interpreter loop
bler, you can do much
. . There's just no way
reasonably expect even
t advanced C compilers to
on your behalf."

— "We come so close to optimal
on most architectures that we
can't do much more without using
NP complete algorithms instead
of heuristics. We can only try to
get little niggles here and there
where the heuristics get slightly
wrong answers."

— "Which compiler is this which
can, for instance, take Netlib
LAPACK and run serial Linpack
as fast as OpenBLAS on recent
x86-64? (Other common hotspots
are available.) Enquiring HPC
minds want to know."

The algo

Context:
that we'

CS 101

015? Seriously?

he late 1990s for
sensitive code,
ers exceeded the
man experts."

sagree,
d records.

author, 2011:
terpreter loop
can do much
just no way
y expect even
d C compilers to
ehalf."

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

The algorithm des

Context: What's t
that we're trying t

CS 101 view: "Ti

ously?

90s for
code,
ed the
rts."

011:
loop
uch
ay
ven
lers to

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

The algorithm designer's job

Context: What's the metric that we're trying to optimize

CS 101 view: "Time".

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

<u>The algorithm designer's job</u>

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

— "We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— "Which compiler is this which can, for instance, take Netlib LAPACK and run serial Linpack as fast as OpenBLAS on recent x86-64? (Other common hotspots are available.) Enquiring HPC minds want to know."

<u>The algorithm designer's job</u>

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

come so close to optimal
architectures that we
much more without using
plete algorithms instead
stics. We can only try to
e niggles here and there
he heuristics get slightly
nswers."

ich compiler is this which
instance, take Netlib
K and run serial Linpack
ns OpenBLAS on recent
(Other common hotspots
able.) Enquiring HPC
ant to know."

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

Many ot
space, c

close to optimal
ures that we
re without using
rithms instead
can only try to
ere and there
cs get slightly

er is this which
take Netlib
serial Linpack
AS on recent
ommon hotspots
quiring HPC
ow."

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

Many other choice
space, cache utiliz

ptimal
ve
t using
tead
ry to
ere
htly

which
o
back
cent
tspots
PC

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

Many other choices of metri
space, cache utilization, etc.

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

Many other choices of metrics: space, cache utilization, etc.

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

Many other choices of metrics: space, cache utilization, etc.

Many physical metrics such as real time and energy defined by physical machines: e.g., my smartphone; my laptop; a cluster; a data center; the entire Internet.

The algorithm designer's job

Context: What's the metric that we're trying to optimize?

CS 101 view: "Time".

What exactly does this mean? Need to specify machine model in enough detail to analyze.

Simple defn of "RAM" model has pathologies: e.g., can factor integers in poly "time".

With more work can build more reasonable "RAM" model.

Many other choices of metrics: space, cache utilization, etc.

Many physical metrics such as real time and energy defined by physical machines: e.g., my smartphone; my laptop; a cluster; a data center; the entire Internet.

Many other abstract models. e.g. Simplify: Turing machine. e.g. Allow parallelism: PRAM.

orithm designer's job

: What's the metric
re trying to optimize?

view: "Time".

xactly does this mean?
specify machine model
gh detail to analyze.

defn of "RAM" model
nologies: e.g., can
tegers in poly "time".

ore work can build
asonable "RAM" model.

Many other choices of metrics:
space, cache utilization, etc.

Many physical metrics
such as real time and energy
defined by physical machines:
e.g., my smartphone;
my laptop;
a cluster;
a data center;
the entire Internet.

Many other abstract models.
e.g. Simplify: Turing machine.
e.g. Allow parallelism: PRAM.

Output
an algor
of instru

Try to m
cost of t
in the sp
(or comb

igner's job

the metric
to optimize?

me".

s this mean?
achine model
o analyze.

AM" model
.g., can
oly "time".

an build
RAM" model.

Many other choices of metrics:
space, cache utilization, etc.

Many physical metrics
such as real time and energy
defined by physical machines:
e.g., my smartphone;
my laptop;
a cluster;
a data center;
the entire Internet.

Many other abstract models.
e.g. Simplify: Turing machine.
e.g. Allow parallelism: PRAM.

Output of algorith
an algorithm—spe
of instructions for

Try to minimize
cost of the algorit
in the specified me
(or combinations o

o

e?

n?
odel

el

e".

odel.

Many other choices of metrics:
space, cache utilization, etc.

Many physical metrics
such as real time and energy
defined by physical machines:
e.g., my smartphone;
my laptop;
a cluster;
a data center;
the entire Internet.

Many other abstract models.
e.g. Simplify: Turing machine.
e.g. Allow parallelism: PRAM.

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics)

Many other choices of metrics:
space, cache utilization, etc.

Many physical metrics
such as real time and energy
defined by physical machines:
e.g., my smartphone;
my laptop;
a cluster;
a data center;
the entire Internet.

Many other abstract models.
e.g. Simplify: Turing machine.
e.g. Allow parallelism: PRAM.

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics).

Many other choices of metrics:
space, cache utilization, etc.

Many physical metrics
such as real time and energy
defined by physical machines:
e.g., my smartphone;
my laptop;
a cluster;
a data center;
the entire Internet.

Many other abstract models.
e.g. Simplify: Turing machine.
e.g. Allow parallelism: PRAM.

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics).

Input to algorithm design:
specification of function
that we want to compute.
Typically a simpler algorithm
in a higher-level language:
e.g., a mathematical formula.

ther choices of metrics:
ache utilization, etc.

nysical metrics
real time and energy
by physical machines:
smartphone;
op;
r;
enter;
re Internet.

ther abstract models.
nplify: Turing machine.
ow parallelism: PRAM.

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics).

Input to algorithm design:
specification of function
that we want to compute.
Typically a simpler algorithm
in a higher-level language:
e.g., a mathematical formula.

Algorith

Massive
State of
extremel

Some ge
with bro
(e.g., dy
but mos
heavily **c**
Karatsul
Strassen
the Boye
the Ford
Shor's a

es of metrics:
ation, etc.
trics
and energy
l machines:
ne;
.
act models.
ing machine.
ism: PRAM.

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics).

Input to algorithm design:
specification of function
that we want to compute.
Typically a simpler algorithm
in a higher-level language:
e.g., a mathematical formula.

Algorithm design i

Massive research t
State of the art is
extremely complic

Some general tech
with broad applic
(e.g., dynamic pro
but most progress
heavily **domain-sp**
Karatsuba's algori
Strassen's algorith
the Boyer–Moore
the Ford–Fulkerso
Shor's algorithm,

cs:

y

s:

.

ne.

M.

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics).

Input to algorithm design:
specification of function
that we want to compute.
Typically a simpler algorithm
in a higher-level language:
e.g., a mathematical formula.

Algorithm design is hard.

Massive research topic.
State of the art is
extremely complicated.

Some general techniques
with broad applicability
(e.g., dynamic programming
but most progress is
heavily **domain-specific**:
Karatsuba's algorithm,
Strassen's algorithm,
the Boyer–Moore algorithm,
the Ford–Fulkerson algorithr
Shor's algorithm, . . .

Output of algorithm design:
an algorithm—specification
of instructions for machine.

Try to minimize
cost of the algorithm
in the specified metric
(or combinations of metrics).

Input to algorithm design:
specification of function
that we want to compute.
Typically a simpler algorithm
in a higher-level language:
e.g., a mathematical formula.

Algorithm design is hard.

Massive research topic.
State of the art is
extremely complicated.

Some general techniques
with broad applicability
(e.g., dynamic programming)
but most progress is
heavily **domain-specific**:
Karatsuba's algorithm,
Strassen's algorithm,
the Boyer–Moore algorithm,
the Ford–Fulkerson algorithm,
Shor's algorithm, . . .

of algorithm design:
ithm—specification
ctions for machine.

inimize
the algorithm
pecified metric
binations of metrics).

algorithm design:
tion of function
want to compute.
y a simpler algorithm
her-level language:
mathematical formula.

Algorithm design is hard.

Massive research topic.
State of the art is
extremely complicated.

Some general techniques
with broad applicability
(e.g., dynamic programming)
but most progress is
heavily **domain-specific**:
Karatsuba's algorithm,
Strassen's algorithm,
the Boyer–Moore algorithm,
the Ford–Fulkerson algorithm,
Shor's algorithm, . . .

Algorith

Wikiped
compiler
tries to
some att
compute

— So th
(viewed
is an opt

m design:
cification
machine.

hm
etric
of metrics).

design:
nction
ompute.
r algorithm
nguage:
cal formula.

Algorithm design is hard.

Massive research topic.
State of the art is
extremely complicated.

Some general techniques
with broad applicability
(e.g., dynamic programming)
but most progress is
heavily **domain-specific**:
Karatsuba's algorithm,
Strassen's algorithm,
the Boyer–Moore algorithm,
the Ford–Fulkerson algorithm,
Shor's algorithm, . . .

Algorithm designe

Wikipedia: "An op
compiler is a comp
tries to minimize o
some attributes of
computer program

— So the algorith
(viewed as a mach
is an optimizing co

).

n

a.

Algorithm design is hard.

Massive research topic.
State of the art is
extremely complicated.

Some general techniques
with broad applicability
(e.g., dynamic programming)
but most progress is
heavily **domain-specific**:
Karatsuba's algorithm,
Strassen's algorithm,
the Boyer–Moore algorithm,
the Ford–Fulkerson algorithm,
Shor's algorithm, . . .

Algorithm designer vs. comp

Wikipedia: "An optimizing
compiler is a compiler that
tries to minimize or maximiz
some attributes of an execut
computer program."

— So the algorithm designe
(viewed as a machine)
is an optimizing compiler?

Algorithm design is hard.

Massive research topic. State of the art is extremely complicated.

Some general techniques with broad applicability (e.g., dynamic programming) but most progress is heavily **domain-specific**: Karatsuba's algorithm, Strassen's algorithm, the Boyer–Moore algorithm, the Ford–Fulkerson algorithm, Shor's algorithm, . . .

Algorithm designer vs. compiler

Wikipedia: "An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program."

— So the algorithm designer (viewed as a machine) is an optimizing compiler?

Algorithm design is hard.

Massive research topic.
State of the art is
extremely complicated.

Some general techniques
with broad applicability
(e.g., dynamic programming)
but most progress is
heavily **domain-specific**:
Karatsuba's algorithm,
Strassen's algorithm,
the Boyer–Moore algorithm,
the Ford–Fulkerson algorithm,
Shor's algorithm, . . .

Algorithm designer vs. compiler

Wikipedia: "An optimizing
compiler is a compiler that
tries to minimize or maximize
some attributes of an executable
computer program."

— So the algorithm designer
(viewed as a machine)
is an optimizing compiler?

Nonsense. Compiler designers
have narrower focus. Example:
"A compiler will not change an
implementation of bubble sort to
use mergesort." — Why not?

m design is hard.

research topic.
the art is
ly complicated.

eneral techniques
ad applicability
namic programming)
t progress is
domain-specific:
ba's algorithm,
's algorithm,
er–Moore algorithm,
l–Fulkerson algorithm,
lgorithm, . . .

Algorithm designer vs. compiler

Wikipedia: "An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program."

— So the algorithm designer (viewed as a machine) is an optimizing compiler?

Nonsense. Compiler designers have narrower focus. Example: "A compiler will not change an implementation of bubble sort to use mergesort." — Why not?

In fact, c
take resp
"machin
Outside
freely bl

Functio

Source
machin
opt

Objec
mach
opt

s hard.

opic.

ated.

niques
bility
gramming)
is
pecific:
thm,
m,
algorithm,
n algorithm,
. . .

Algorithm designer vs. compiler

Wikipedia: "An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program."

— So the algorithm designer (viewed as a machine) is an optimizing compiler?

Nonsense. Compiler designers have narrower focus. Example: "A compiler will not change an implementation of bubble sort to use mergesort." — Why not?

In fact, compiler d
take responsibility
"machine-specific
Outside this bailiw
freely blame algori

Function specifica

Algori

Source code with
machine-indepen
optimizations

Optim

Object code wi
machine-specif
optimizations

)

m,

Algorithm designer vs. compiler

Wikipedia: "An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program."

— So the algorithm designer (viewed as a machine) is an optimizing compiler?

Nonsense. Compiler designers have narrower focus. Example: "A compiler will not change an implementation of bubble sort to use mergesort." — Why not?

In fact, compiler designers take responsibility only for "machine-specific optimizati
Outside this bailiwick they freely blame algorithm desig

Function specification

↓ Algorithm desig

Source code with all machine-independent optimizations

↓ Optimizing com

Object code with machine-specific optimizations

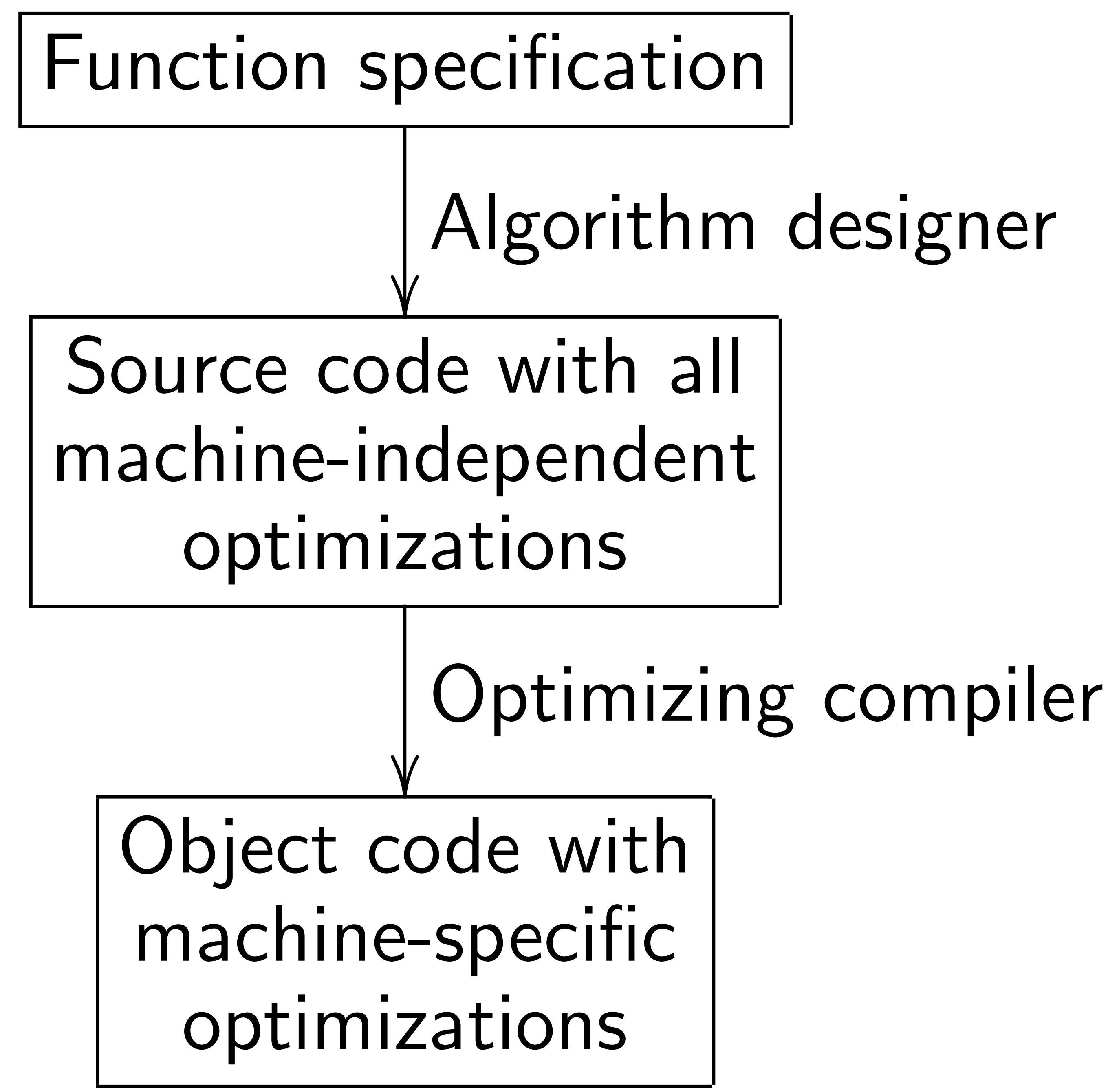
Algorithm designer vs. compiler

Wikipedia: "An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program."

— So the algorithm designer (viewed as a machine) is an optimizing compiler?

Nonsense. Compiler designers have narrower focus. Example: "A compiler will not change an implementation of bubble sort to use mergesort." — Why not?

In fact, compiler designers take responsibility only for "machine-specific optimization". Outside this bailiwick they freely blame algorithm designers:



m designer vs. compiler

ia: "An optimizing
r is a compiler that
minimize or maximize
tributes of an executable
er program."

ne algorithm designer
as a machine)
timizing compiler?

e. Compiler designers
rrower focus. Example:
piler will not change an
ntation of bubble sort to
gesort." — Why not?

In fact, compiler designers take responsibility only for "machine-specific optimization". Outside this bailiwick they freely blame algorithm designers:

Function specification

↓ Algorithm designer

Source code with all machine-independent optimizations

↓ Optimizing compiler

Object code with machine-specific optimizations

Output
is algorit

Algorith
targeted
Why bui
compiler

r vs. compiler

ptimizing
piler that
or maximize
an executable
n."

m designer
ine)
ompiler?

er designers
us. Example:
ot change an
bubble sort to
– Why not?

In fact, compiler designers take responsibility only for "machine-specific optimization". Outside this bailiwick they freely blame algorithm designers:

Function specification

↓ Algorithm designer

Source code with all machine-independent optimizations

↓ Optimizing compiler

Object code with machine-specific optimizations

Output of optimiz
is algorithm for ta

Algorithm designe
targeted this mach
Why build a new c
compiler $\circ$ old des

piler

ze

table

r

rs

ble:

an

ort to

t?

In fact, compiler designers take responsibility only for "machine-specific optimization". Outside this bailiwick they freely blame algorithm designers:

Function specification

↓ Algorithm designer

Source code with all machine-independent optimizations

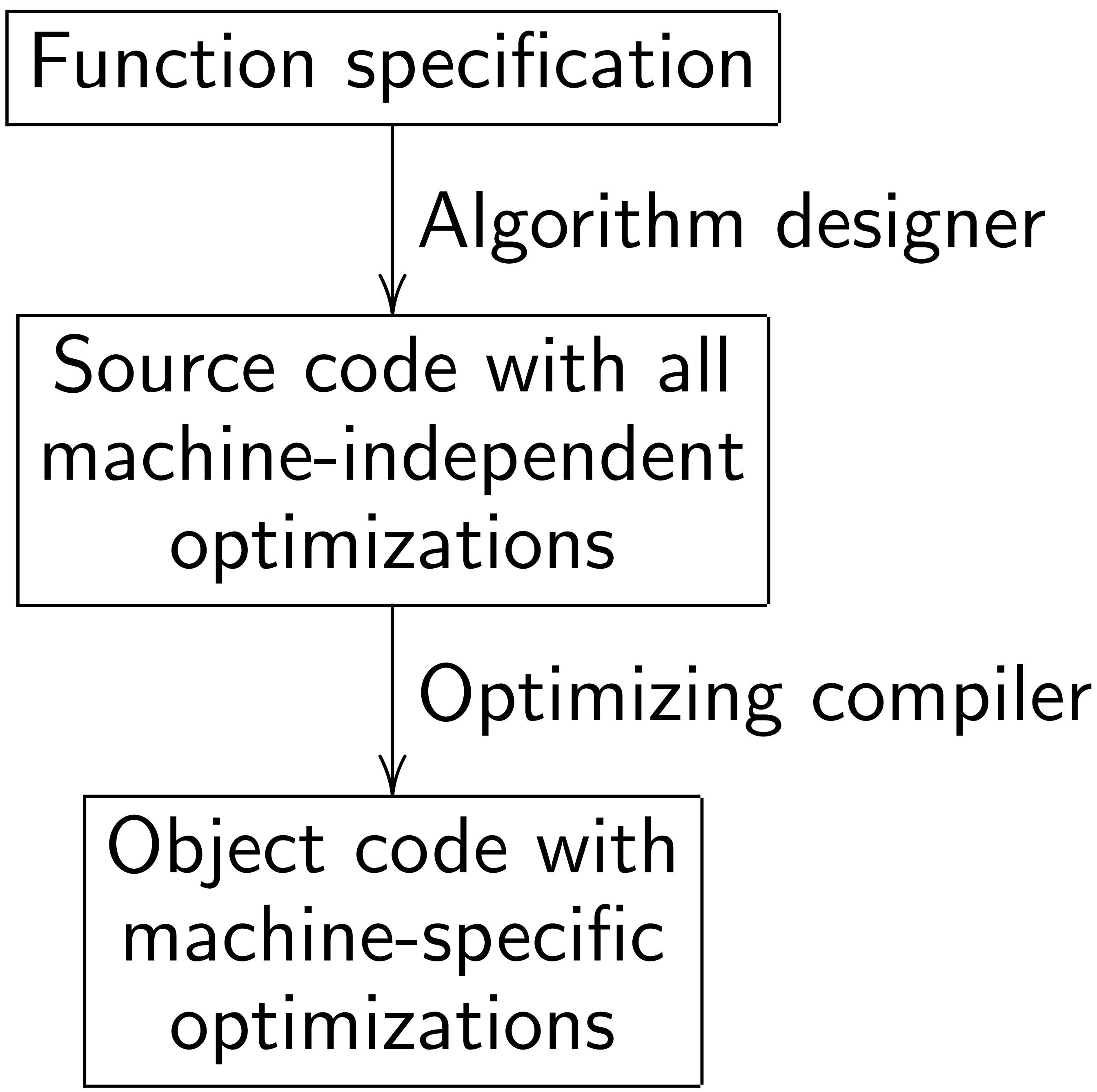
↓ Optimizing compiler

Object code with machine-specific optimizations

Output of optimizing compi
is algorithm for target mach

Algorithm designer could ha
targeted this machine direct
Why build a new designer as
compiler ∘ old designer?

In fact, compiler designers take responsibility only for "machine-specific optimization". Outside this bailiwick they freely blame algorithm designers:



Output of optimizing compiler is algorithm for target machine.

Algorithm designer could have targeted this machine directly. Why build a new designer as compiler ∘ old designer?

In fact, compiler designers take responsibility only for "machine-specific optimization". Outside this bailiwick they freely blame algorithm designers:

$$\begin{array}{c} \boxed{\text{Function specification}} \\ \Big\downarrow \ \text{Algorithm designer} \\ \boxed{\begin{array}{c}\text{Source code with all}\\ \text{machine-independent}\\ \text{optimizations}\end{array}} \\ \Big\downarrow \ \text{Optimizing compiler} \\ \boxed{\begin{array}{c}\text{Object code with}\\ \text{machine-specific}\\ \text{optimizations}\end{array}} \end{array}$$

Output of optimizing compiler is algorithm for target machine.

Algorithm designer could have targeted this machine directly. Why build a new designer as compiler $\circ$ old designer?

Advantages of this composition:
(1) save designer's time in handling complex machines;
(2) save designer's time in handling many machines.

Optimizing compiler is general-purpose, used by many designers.

compiler designers
ponsibility only for
e-specific optimization".
this bailiwick they
ame algorithm designers:

n specification

↓ Algorithm designer

code with all
e-independent
imizations

↓ Optimizing compiler

ct code with
hine-specific
imizations

Output of optimizing compiler
is algorithm for target machine.

Algorithm designer could have
targeted this machine directly.
Why build a new designer as
compiler ∘ old designer?

Advantages of this composition:
(1) save designer's time
in handling complex machines;
(2) save designer's time
in handling many machines.

Optimizing compiler is general-
purpose, used by many designers.

And the
say the
Rememb
"We cor
on most
only try
and ther
get sligh

lesigners
only for
optimization".
vick they
thm designers:

tion

thm designer

n all
dent

nizing compiler

th
ic

Output of optimizing compiler is algorithm for target machine.

Algorithm designer could have targeted this machine directly. Why build a new designer as compiler ∘ old designer?

Advantages of this composition:
(1) save designer's time
in handling complex machines;
(2) save designer's time
in handling many machines.

Optimizing compiler is general-purpose, used by many designers.

And the compiler
say the results are
Remember the typ
"We come so clos
on most architectu
only try to get litt
and there where th
get slightly wrong

on".

ners:

ner

piler

Output of optimizing compiler is algorithm for target machine.

Algorithm designer could have targeted this machine directly. Why build a new designer as compiler $\circ$ old designer?

Advantages of this composition: (1) save designer's time in handling complex machines; (2) save designer's time in handling many machines.

Optimizing compiler is general-purpose, used by many designers.

And the compiler designers say the results are great!

Remember the typical quote

"We come so close to optim on most architectures . . . W only try to get little niggles and there where the heuristi get slightly wrong answers."

Output of optimizing compiler is algorithm for target machine.

Algorithm designer could have targeted this machine directly. Why build a new designer as compiler $\circ$ old designer?

Advantages of this composition:
(1) save designer's time in handling complex machines;
(2) save designer's time in handling many machines.

Optimizing compiler is general-purpose, used by many designers.

And the compiler designers say the results are great!
Remember the typical quote:
"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

Output of optimizing compiler
is algorithm for target machine.

Algorithm designer could have
targeted this machine directly.
Why build a new designer as
compiler $\circ$ old designer?

Advantages of this composition:
(1) save designer's time
in handling complex machines;
(2) save designer's time
in handling many machines.

Optimizing compiler is general-purpose, used by many designers.

And the compiler designers
say the results are great!
Remember the typical quote:
"We come so close to optimal
on most architectures . . . We can
only try to get little niggles here
and there where the heuristics
get slightly wrong answers."

— But they're wrong.
Their results are becoming
less and less satisfactory,
despite clever compiler research;
more CPU time for compilation;
extermination of many targets.

of optimizing compiler
thm for target machine.

m designer could have
this machine directly.
ild a new designer as
r ∘ old designer?

ges of this composition:
designer's time
ing complex machines;
designer's time
ing many machines.

ing compiler is general-
used by many designers.

And the compiler designers
say the results are great!
Remember the typical quote:
"We come so close to optimal
on most architectures . . . We can
only try to get little niggles here
and there where the heuristics
get slightly wrong answers."

— But they're wrong.
Their results are becoming
less and less satisfactory,
despite clever compiler research;
more CPU time for compilation;
extermination of many targets.

How the

Fastest c
hot spot
by algori
using do
Mediocr
output c
hot spot
algorithr

ing compiler
rget machine.

r could have
nine directly.
designer as
igner?

s composition:
s time
ex machines;
s time
machines.

er is general-
many designers.

And the compiler designers
say the results are great!
Remember the typical quote:
"We come so close to optimal
on most architectures . . . We can
only try to get little niggles here
and there where the heuristics
get slightly wrong answers."

— But they're wrong.
Their results are becoming
less and less satisfactory,
despite clever compiler research;
more CPU time for compilation;
extermination of many targets.

How the code bas

Fastest code:
hot spots targeted
by algorithm desig
using domain-spec

Mediocre code:
output of optimizi
hot spots not yet
algorithm designer

ler
ine.

ve
ly.
s

tion:

es;

ral-
gners.

And the compiler designers
say the results are great!
Remember the typical quote:
"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.
Their results are becoming **less and less satisfactory**,
despite clever compiler research;
more CPU time for compilation;
extermination of many targets.

How the code base is evolvi

Fastest code:
hot spots targeted directly
by algorithm designers,
using domain-specific tools.

Mediocre code:
output of optimizing compil
hot spots not yet reached by
algorithm designers.

And the compiler designers
say the results are great!
Remember the typical quote:
"We come so close to optimal
on most architectures . . . We can
only try to get little niggles here
and there where the heuristics
get slightly wrong answers."

— But they're wrong.
Their results are becoming
less and less satisfactory,
despite clever compiler research;
more CPU time for compilation;
extermination of many targets.

How the code base is evolving:

Fastest code:
hot spots targeted directly
by algorithm designers,
using domain-specific tools.

Mediocre code:
output of optimizing compilers;
hot spots not yet reached by
algorithm designers.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.
Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.
Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code:
hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code:
output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code:
code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code:
hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code:
output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code:
code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote: "We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong. Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote: "We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong. Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote: "We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong. Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code: hot spots targeted directly by algorithm designers, using domain-specific tools.

Mediocre code: output of optimizing compilers; hot spots not yet reached by algorithm designers.

Slowest code: code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great!

Remember the typical quote:

"We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong.

Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code:
hot spots targeted directly by algorithm designers, using domain-specific tools.

Slowest code:
code with optimization turned off; so cold that optimization isn't worth the costs.

And the compiler designers say the results are great! Remember the typical quote: "We come so close to optimal on most architectures . . . We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

— But they're wrong. Their results are becoming **less and less satisfactory**, despite clever compiler research; more CPU time for compilation; extermination of many targets.

How the code base is evolving:

Fastest code (most CPU time): hot spots targeted directly by algorithm designers, using domain-specific tools.

Slowest code (almost all code): code with optimization turned off; so cold that optimization isn't worth the costs.

---

compiler designers
results are great!
ber the typical quote:
me so close to optimal
architectures . . . We can
to get little niggles here
re where the heuristics
tly wrong answers."

they're wrong.
sults are becoming
l less satisfactory,
clever compiler research;
PU time for compilation;
nation of many targets.

How the code base is evolving:

Fastest code (most CPU time):
hot spots targeted directly
by algorithm designers,
using domain-specific tools.

Slowest code (almost all code):
code with optimization turned off;
so cold that optimization
isn't worth the costs.

2013 Wa
"AUGEM
high per
algebra k

"Many D
are man
assembly
Our tem
[allows]
optimiza
applicati
allows th
how bes
kernels t
integrate

designers
great!
pical quote:
e to optimal
ures . . . We can
le niggles here
ne heuristics
answers.”

ong.
ecoming
sfactory,
piler research;
r compilation;
many targets.

How the code base is evolving:

Fastest code (most CPU time):
hot spots targeted directly
by algorithm designers,
using domain-specific tools.

Slowest code (almost all code):
code with optimization turned off;
so cold that optimization
isn’t worth the costs.

2013 Wang–Zhang
“AUGEM: automa
high performance
algebra kernels on

“Many DLA kerne
are manually imple
assembly by doma
Our template-base
[allows] multiple m
optimizations in a
application specifi
allows the expert k
how best to optim
kernels to be seam
integrated in the p

e:

al

/e can

here

cs

rch;

tion;

ets.

How the code base is evolving:

Fastest code (most CPU time):
hot spots targeted directly
by algorithm designers,
using domain-specific tools.

Slowest code (almost all code):
code with optimization turned off;
so cold that optimization
isn't worth the costs.

2013 Wang–Zhang–Zhang–Y
"AUGEM: automatically ger
high performance dense line
algebra kernels on x86 CPUs

"Many DLA kernels in ATLA
are manually implemented in
assembly by domain experts
Our template-based approac
[allows] multiple machine-lev
optimizations in a domain/
application specific setting a
allows the expert knowledge
how best to optimize varying
kernels to be seamlessly
integrated in the process."

How the code base is evolving:

Fastest code (most CPU time):
hot spots targeted directly
by algorithm designers,
using domain-specific tools.

Slowest code (almost all code):
code with optimization turned off;
so cold that optimization
isn't worth the costs.

2013 Wang–Zhang–Zhang–Yi "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs":

"Many DLA kernels in ATLAS are manually implemented in assembly by domain experts . . . Our template-based approach [allows] multiple machine-level optimizations in a domain/ application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process."

e code base is evolving:

---

code (most CPU time):
ts targeted directly
ithm designers,
omain-specific tools.

---

code (almost all code):
th optimization turned off;
that optimization
rth the costs.

2013 Wang–Zhang–Zhang–Yi "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs":

"Many DLA kernels in ATLAS are manually implemented in assembly by domain experts . . . Our template-based approach [allows] multiple machine-level optimizations in a domain/ application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process."

Why thi

The actu
farther a
from the

e is evolving:

t CPU time):
directly
gners,
cific tools.

ost all code):
ation turned off;
nization
sts.

2013 Wang–Zhang–Zhang–Yi "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs":

"Many DLA kernels in ATLAS are manually implemented in assembly by domain experts ... Our template-based approach [allows] multiple machine-level optimizations in a domain/ application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process."

Why this is happe

The actual machin
farther and farther
from the source m

ng:

ne):

de):
ed off;

2013 Wang–Zhang–Zhang–Yi "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs":

"Many DLA kernels in ATLAS are manually implemented in assembly by domain experts . . . Our template-based approach [allows] multiple machine-level optimizations in a domain/ application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process."

Why this is happening

The actual machine is evolv farther and farther away from the source machine.

2013 Wang–Zhang–Zhang–Yi "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs":

"Many DLA kernels in ATLAS are manually implemented in assembly by domain experts . . . Our template-based approach [allows] multiple machine-level optimizations in a domain/ application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process."

Why this is happening

The actual machine is evolving farther and farther away from the source machine.

2013 Wang–Zhang–Zhang–Yi "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs":

"Many DLA kernels in ATLAS are manually implemented in assembly by domain experts . . . Our template-based approach [allows] multiple machine-level optimizations in a domain/ application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process."

Why this is happening

The actual machine is evolving farther and farther away from the source machine.

Minor optimization challenges:

- Pipelining.
- Superscalar processing.

Major optimization challenges:

- Vectorization.
- Many threads; many cores.
- The memory hierarchy; the ring; the mesh.
- Larger-scale parallelism.
- Larger-scale networking.

ang–Zhang–Zhang–Yi
M: automatically generate
formance dense linear
kernels on x86 CPUs":

DLA kernels in ATLAS
ually implemented in
y by domain experts . . .
plate-based approach
multiple machine-level
tions in a domain/
ion specific setting and
he expert knowledge of
t to optimize varying
to be seamlessly
ed in the process."

Why this is happening

The actual machine is evolving farther and farther away from the source machine.

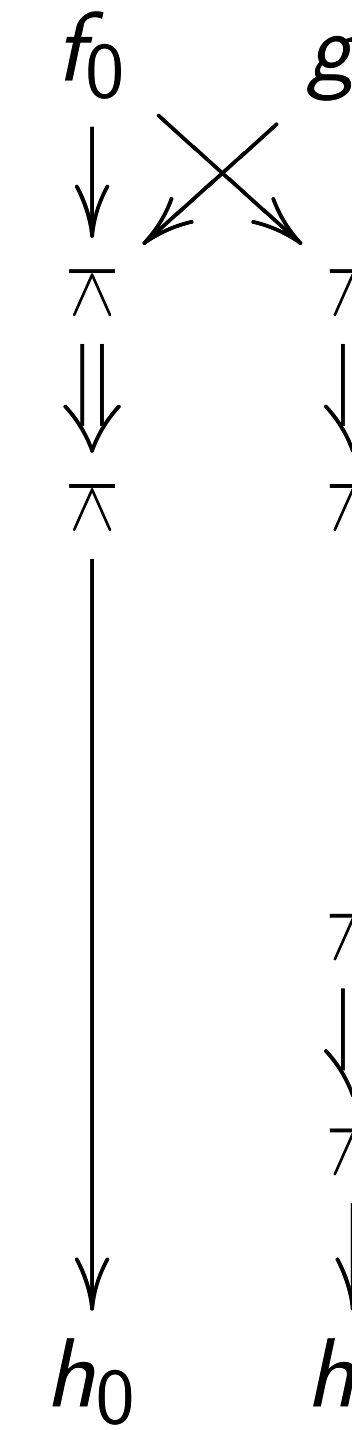
Minor optimization challenges:

- Pipelining.
- Superscalar processing.

Major optimization challenges:

- Vectorization.
- Many threads; many cores.
- The memory hierarchy; the ring; the mesh.
- Larger-scale parallelism.
- Larger-scale networking.

CPU des



Gates $\barwedge$
product
of intege

g–Zhang–Yi
ntically generate
dense linear
x86 CPUs”:

els in ATLAS
emented in
in experts . . .
ed approach
nachine-level
domain/
c setting and
knowledge of
nize varying
nlessly
process.”

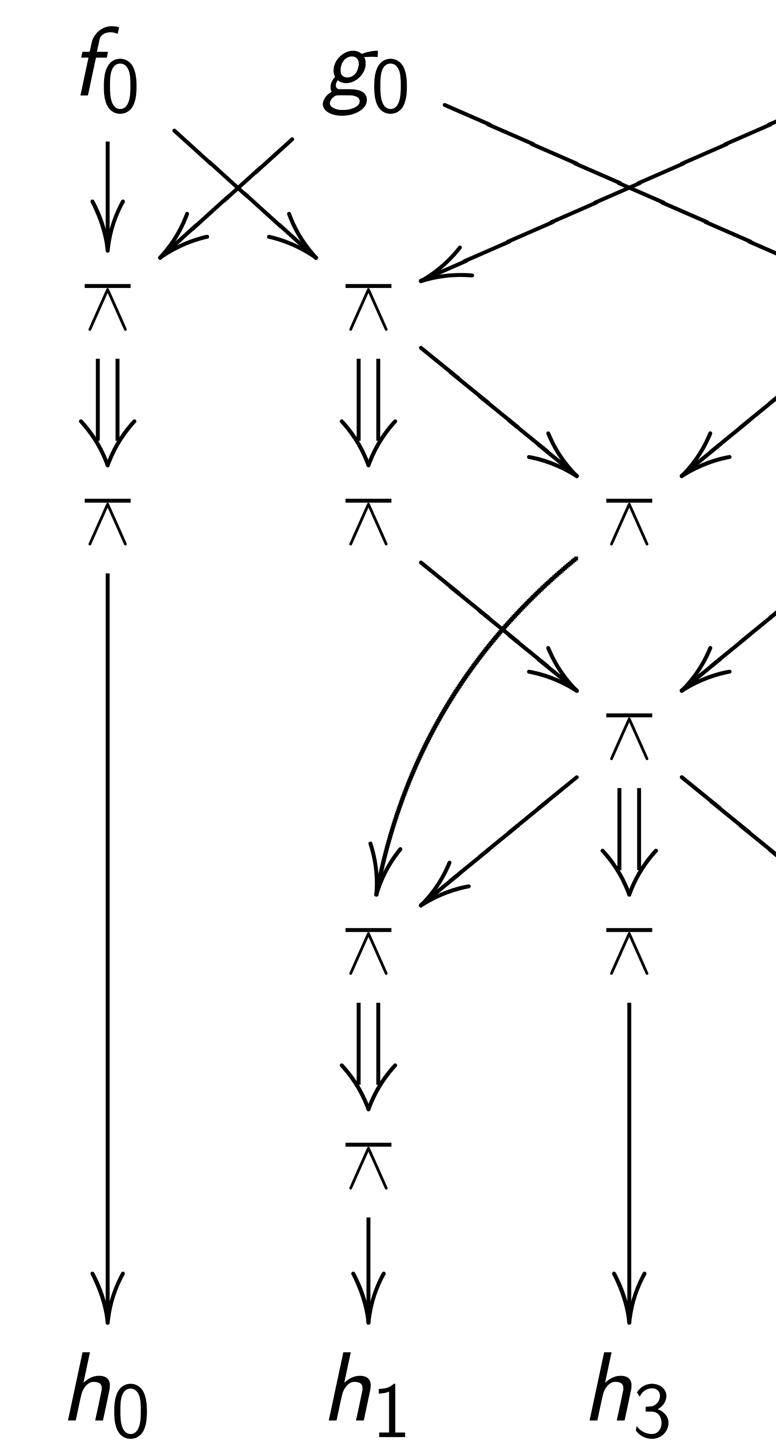
Why this is happening

The actual machine is evolving farther and farther away from the source machine.

Minor optimization challenges:
- Pipelining.
- Superscalar processing.

Major optimization challenges:
- Vectorization.
- Many threads; many cores.
- The memory hierarchy; the ring; the mesh.
- Larger-scale parallelism.
- Larger-scale networking.

CPU design in a n



Gates $\barwedge : a, b \mapsto 1$
product $h_0 + 2h_1 +$
of integers $f_0 + 2f$

Yi
nerate
ar
s":

AS
n
...
ch
vel

nd
of
g

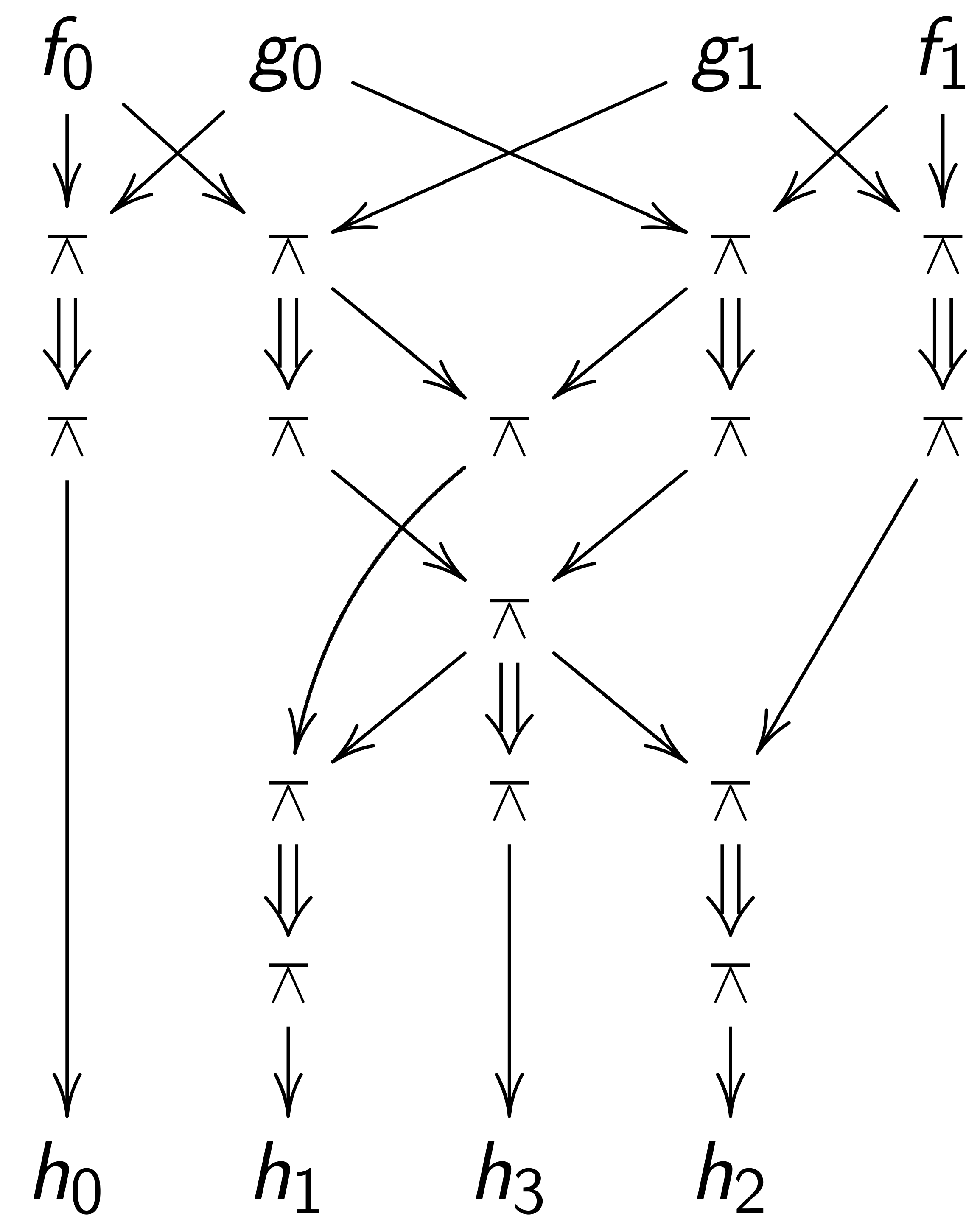
Why this is happening

The actual machine is evolving farther and farther away from the source machine.

Minor optimization challenges:
- Pipelining.
- Superscalar processing.

Major optimization challenges:
- Vectorization.
- Many threads; many cores.
- The memory hierarchy; the ring; the mesh.
- Larger-scale parallelism.
- Larger-scale networking.

CPU design in a nutshell



Gates $\barwedge : a, b \mapsto 1 - ab$ com
product $h_0 + 2h_1 + 4h_2 + 8$
of integers $f_0 + 2f_1, g_0 + 2g$

Why this is happening

The actual machine is evolving farther and farther away from the source machine.

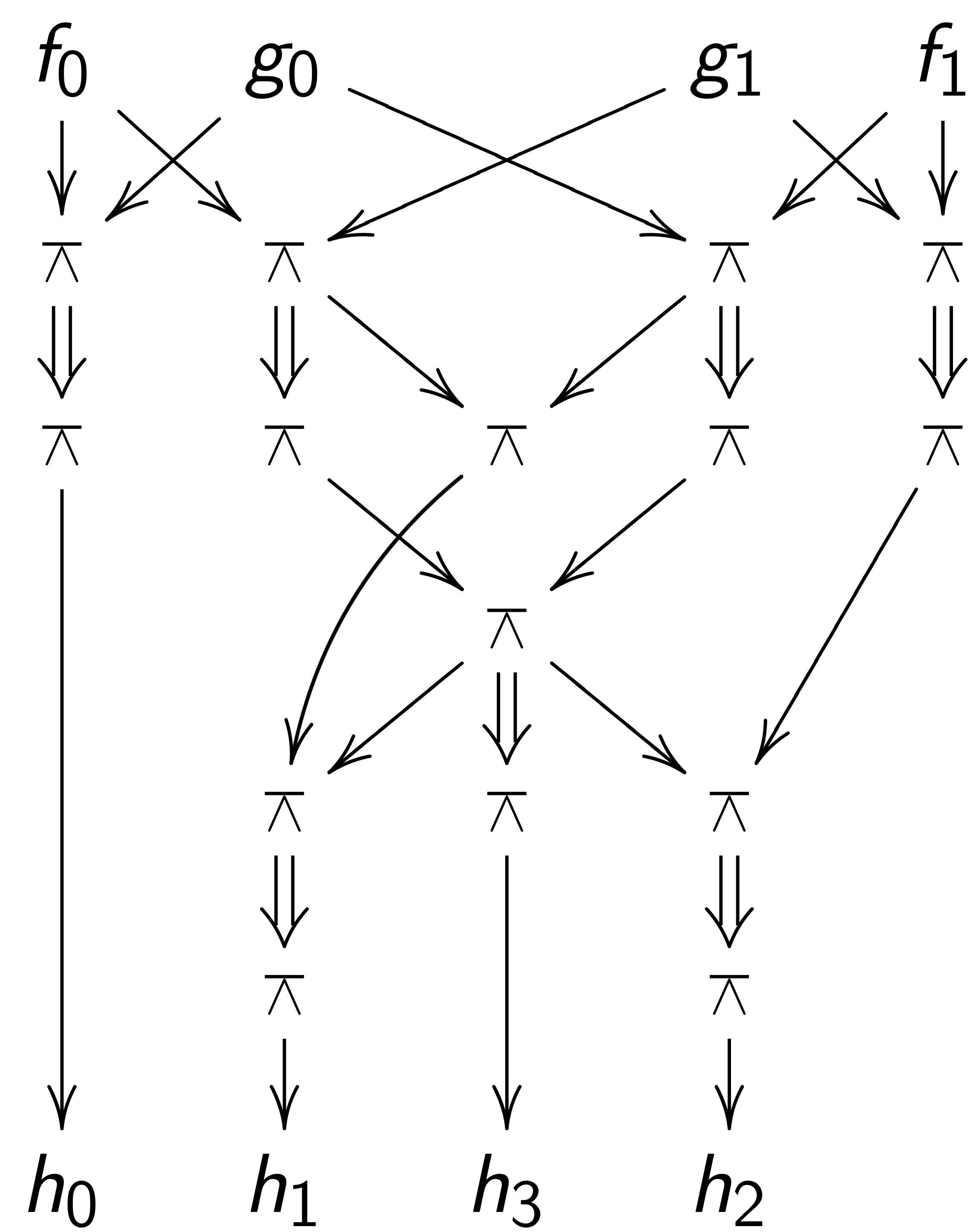
Minor optimization challenges:

- Pipelining.
- Superscalar processing.

Major optimization challenges:

- Vectorization.
- Many threads; many cores.
- The memory hierarchy; the ring; the mesh.
- Larger-scale parallelism.
- Larger-scale networking.

CPU design in a nutshell



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

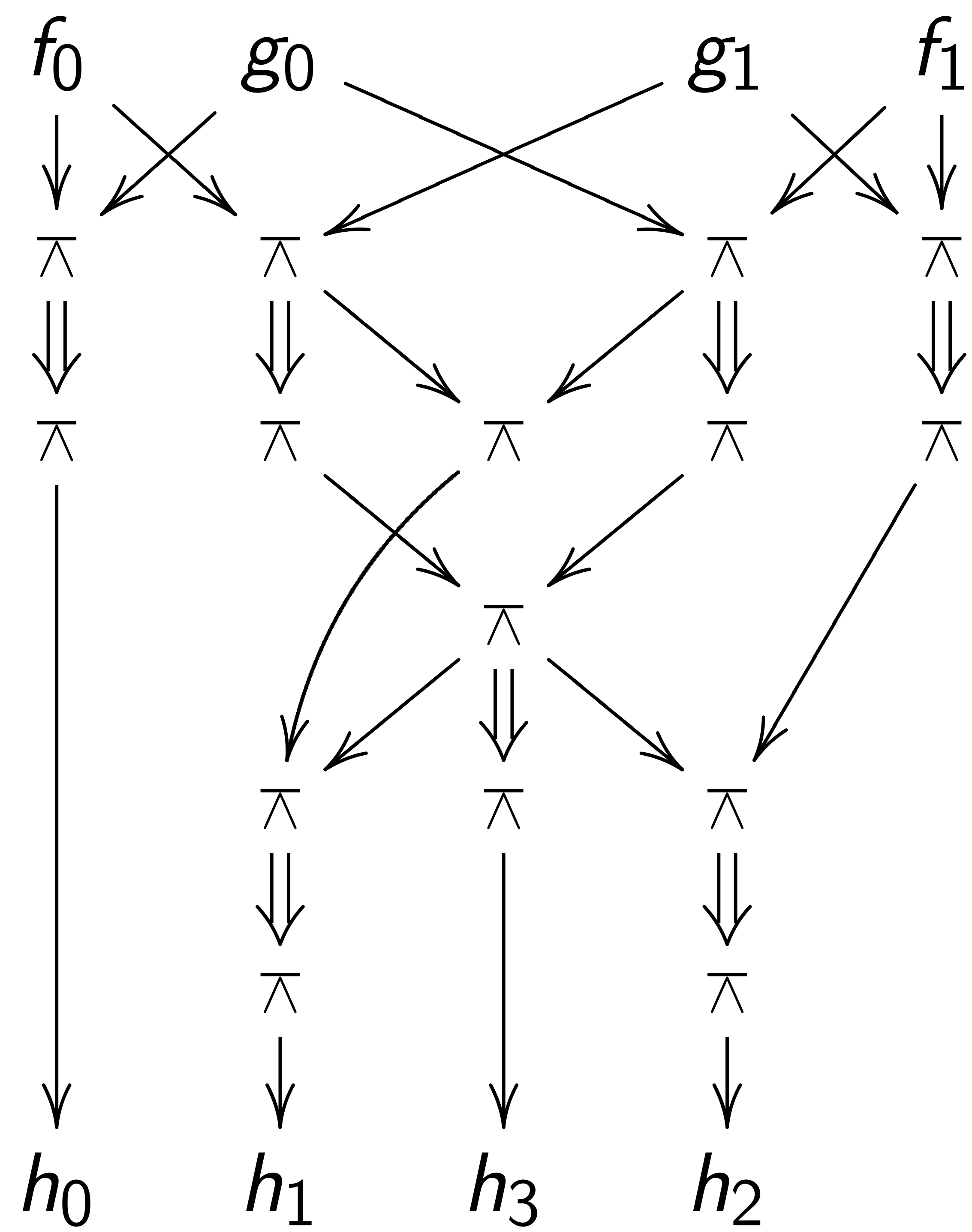
s is happening

ual machine is evolving
nd farther away
e source machine.

ptimization challenges:
ning.
scalar processing.

ptimization challenges:
rization.
threads; many cores.
nemory hierarchy;
ng; the mesh.
-scale parallelism.
-scale networking.

CPU design in a nutshell



Gates $\barwedge : a, b \mapsto 1 - ab$ computing
product $h_0 + 2h_1 + 4h_2 + 8h_3$
of integers $f_0 + 2f_1, g_0 + 2g_1$.

Electrici
percolat
If $f_0, f_1,$
then $h_0,$
a few m

ning

ne is evolving
r away
achine.

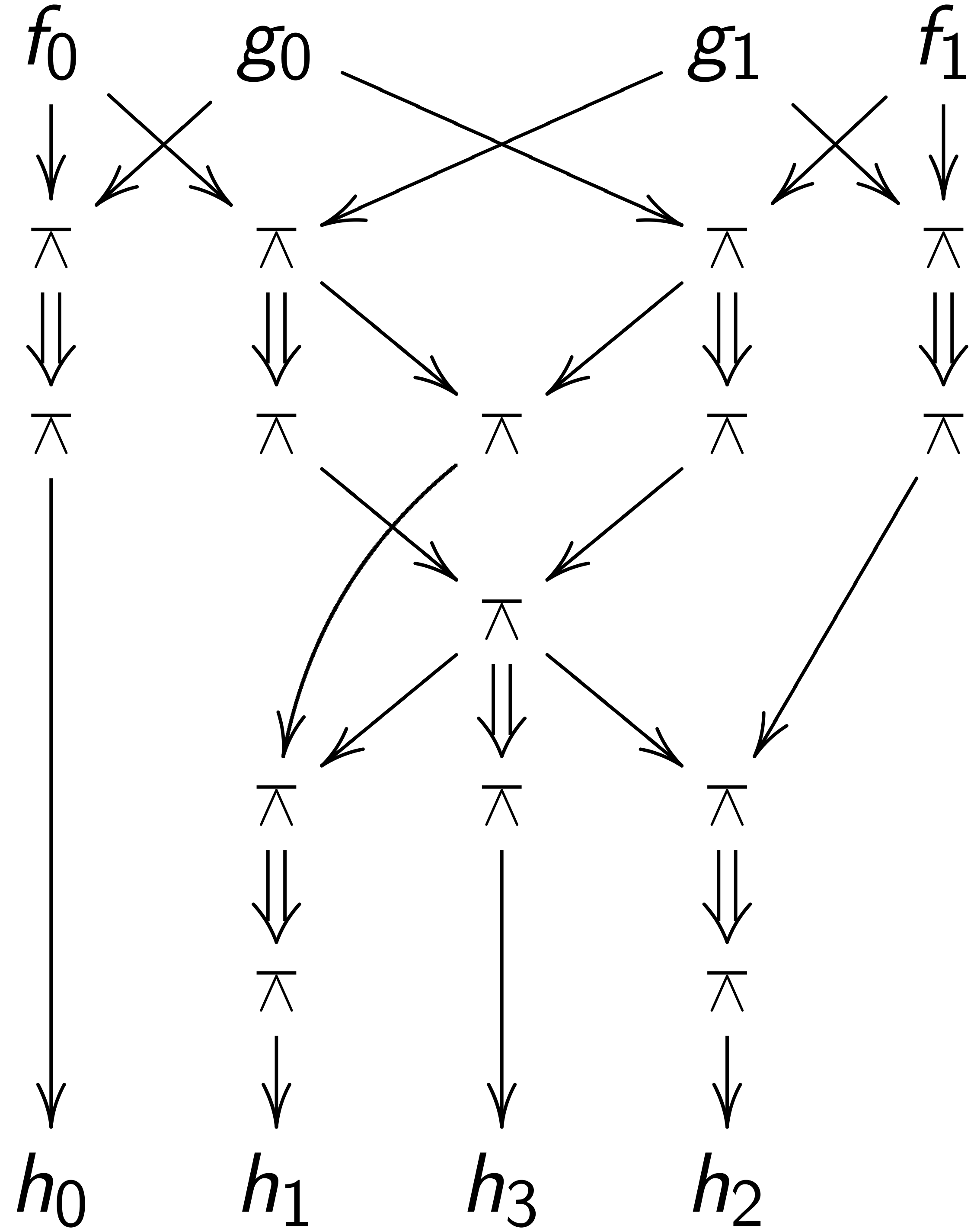
n challenges:

cessing.

n challenges:

many cores.
erarchy;
sh.
allelism.
working.

CPU design in a nutshell



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

Electricity takes ti
percolate through
If f_0, f_1, g_0, g_1 are
then h_0, h_1, h_2, h_3
a few moments lat

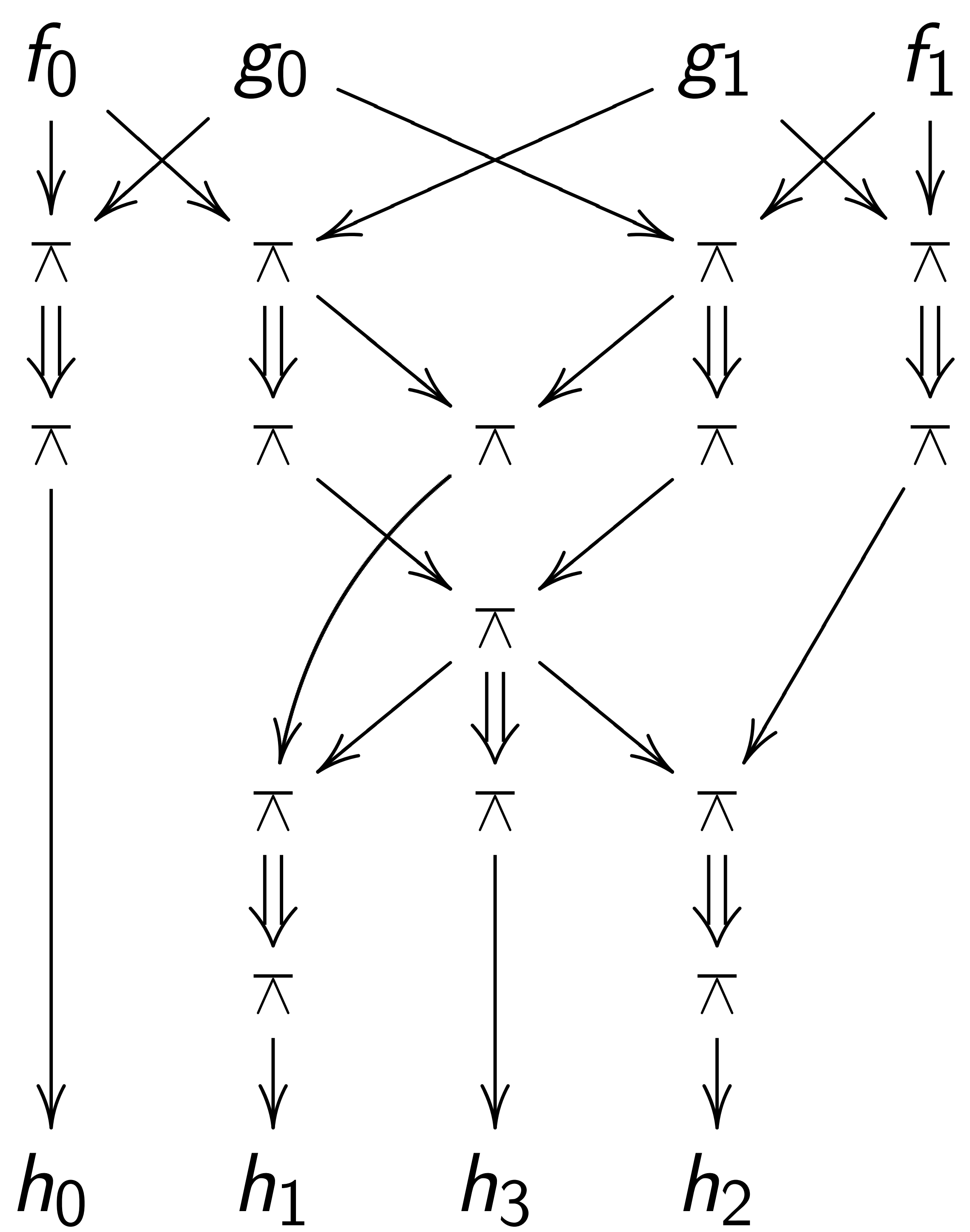
ing

es:

es:

s.

CPU design in a nutshell

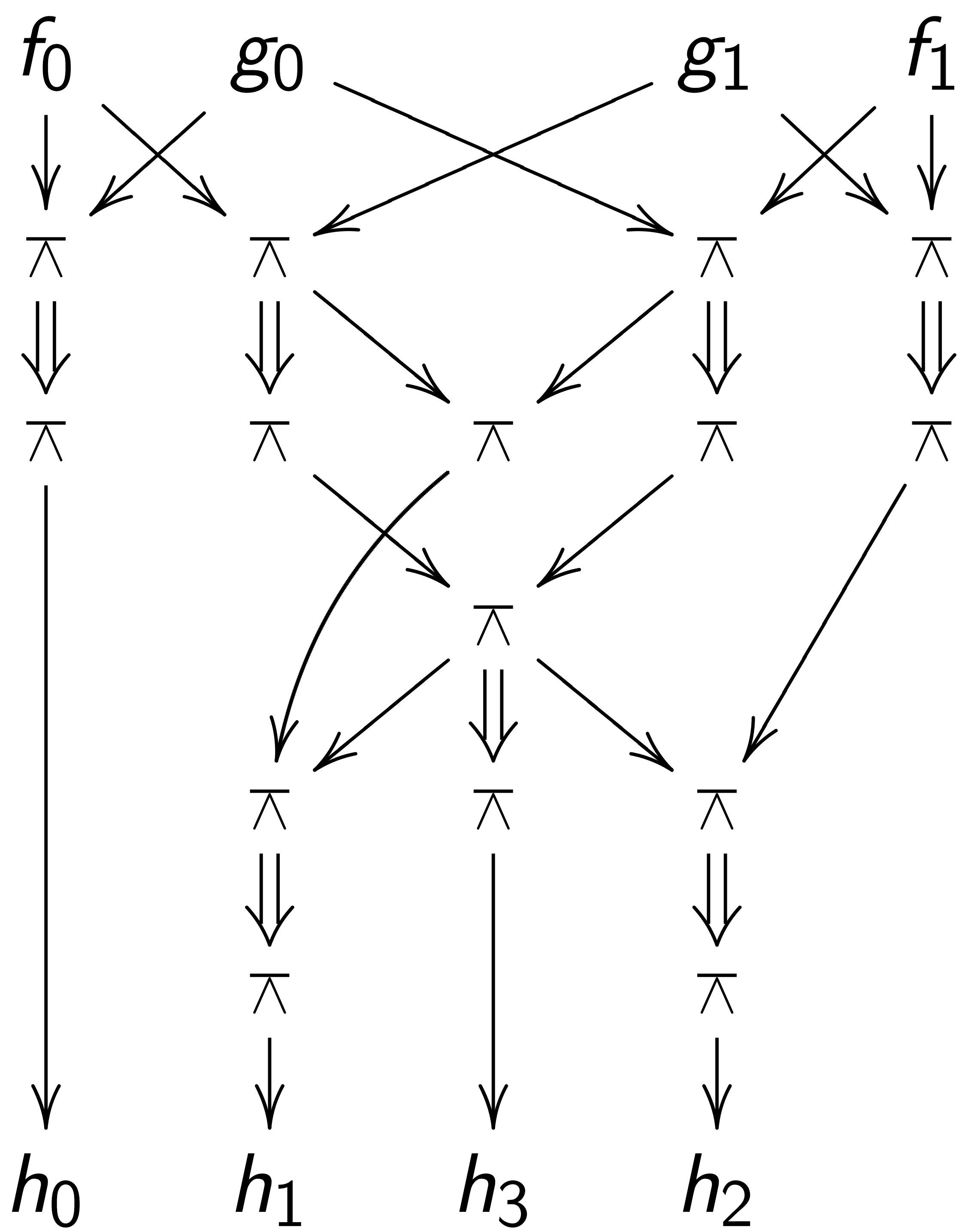


Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

Electricity takes time to percolate through wires and

If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

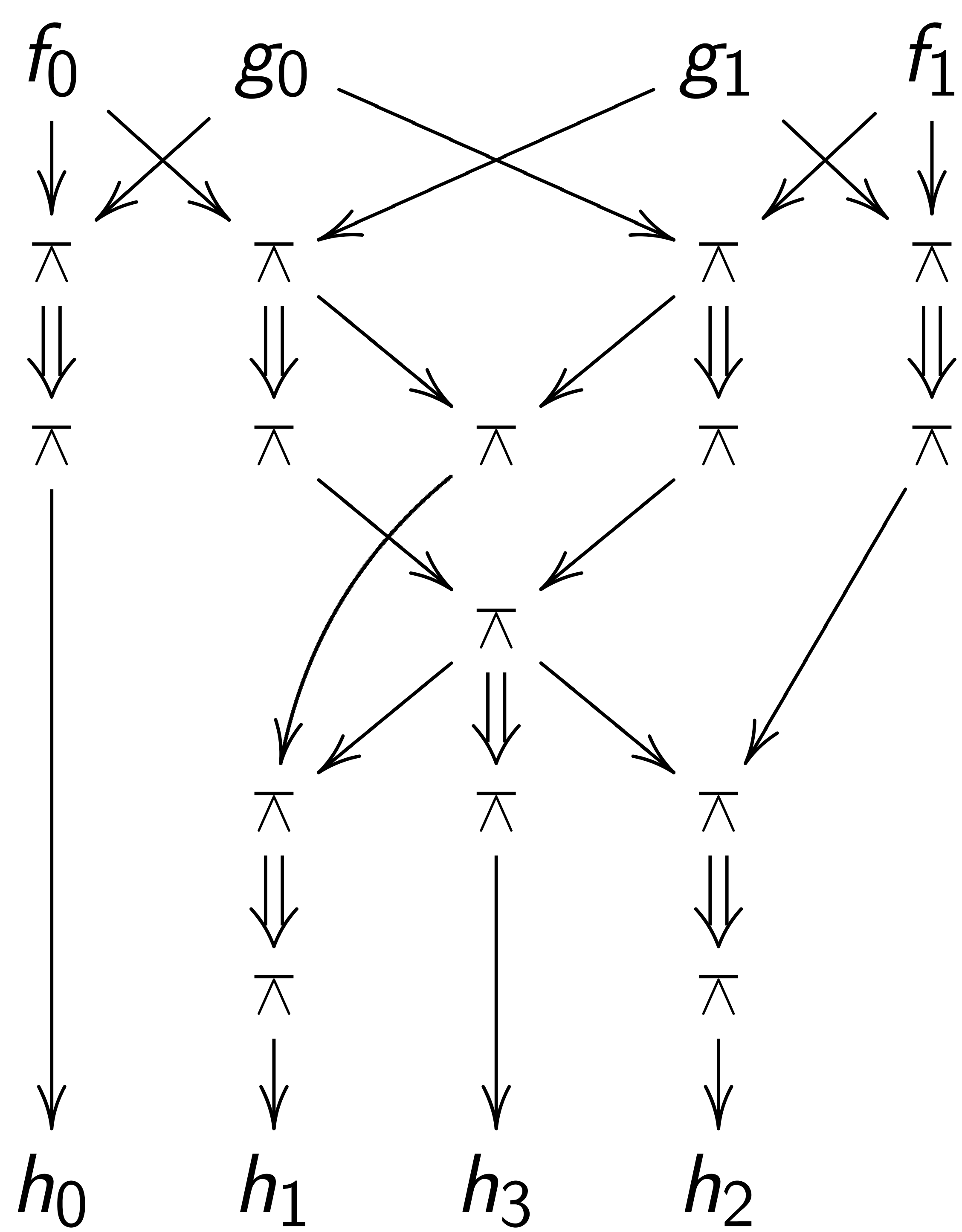
CPU design in a nutshell



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

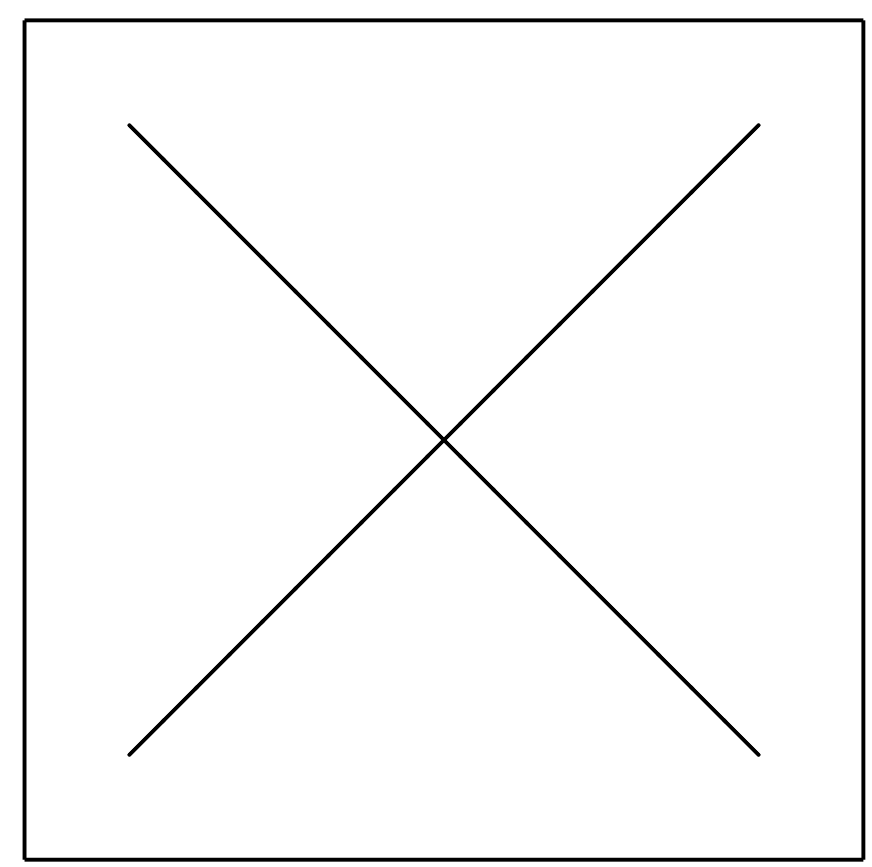
CPU design in a nutshell



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

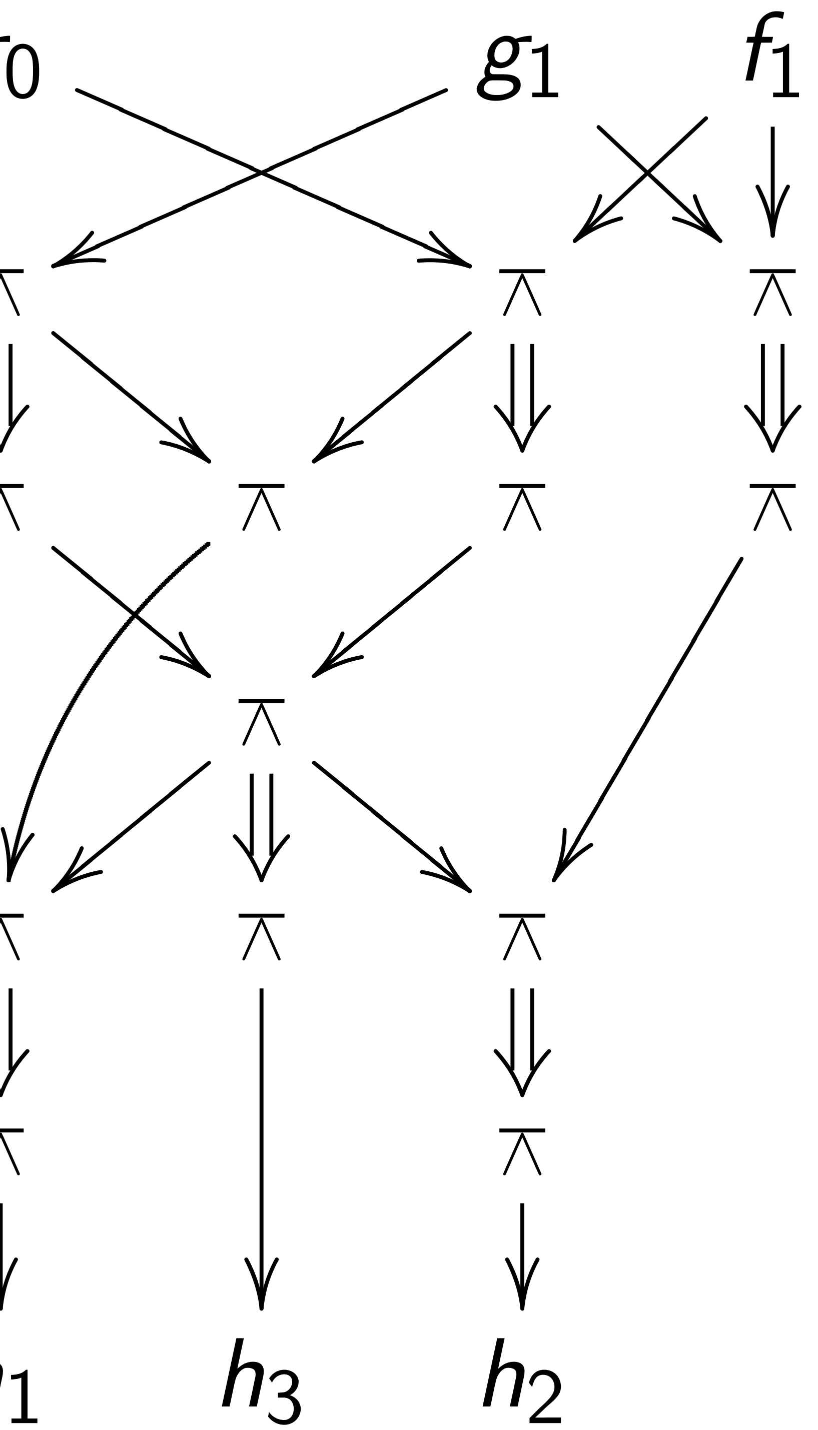
Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

Build circuit with more gates to multiply (e.g.) 32-bit integers:



(Details omitted.)

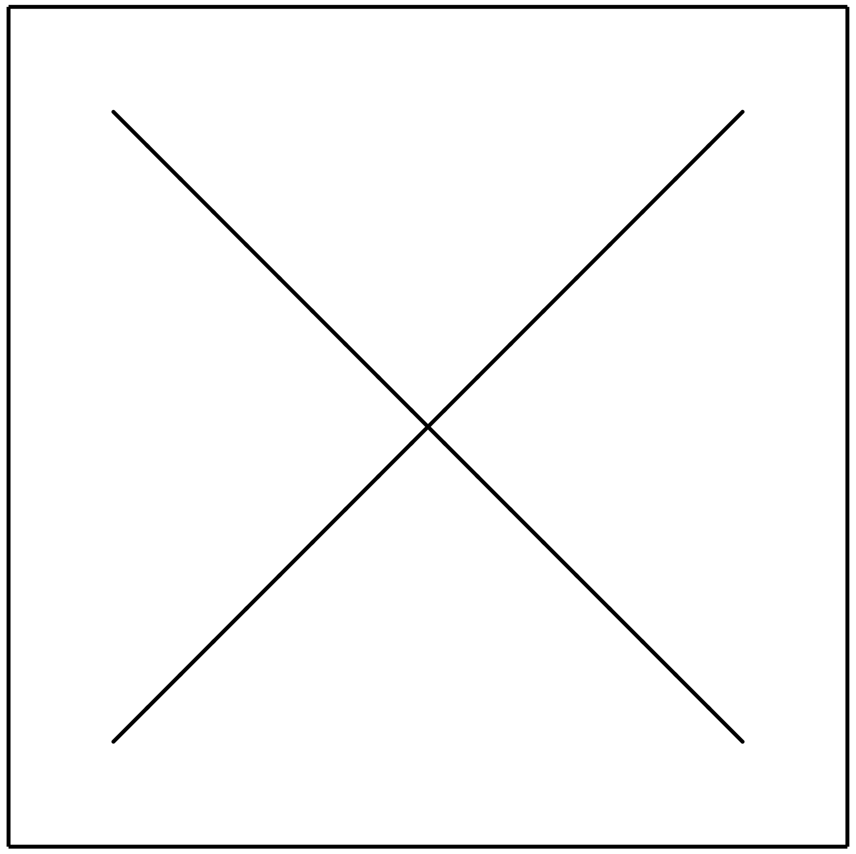
sign in a nutshell



: $a, b \mapsto 1 - ab$ computing
$h_0 + 2h_1 + 4h_2 + 8h_3$
ers $f_0 + 2f_1, g_0 + 2g_1$.

Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

Build circuit with more gates to multiply (e.g.) 32-bit integers:

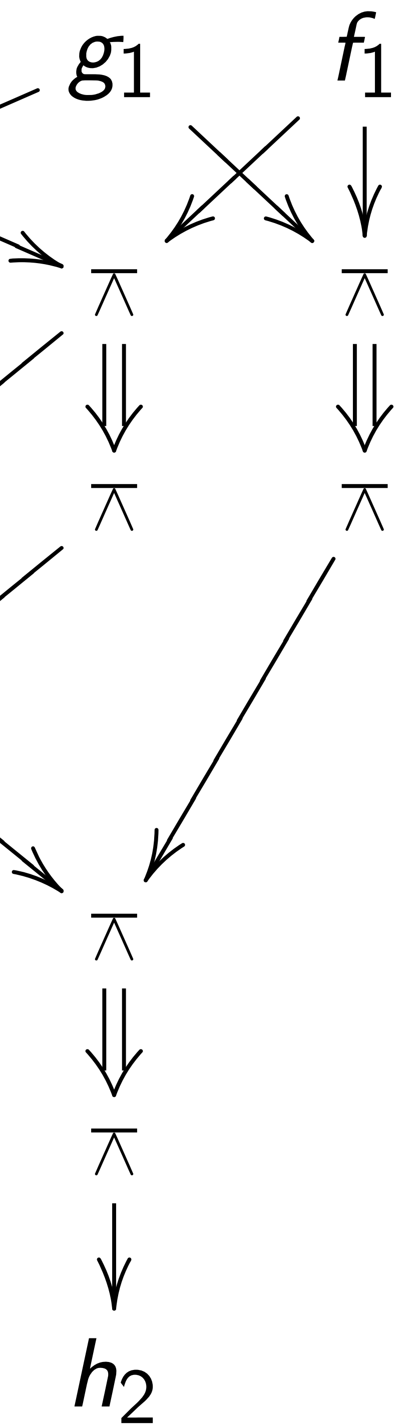


(Details omitted.)

Build cir
32-bit in
given 4-
and 32-b

reg
re

utshell



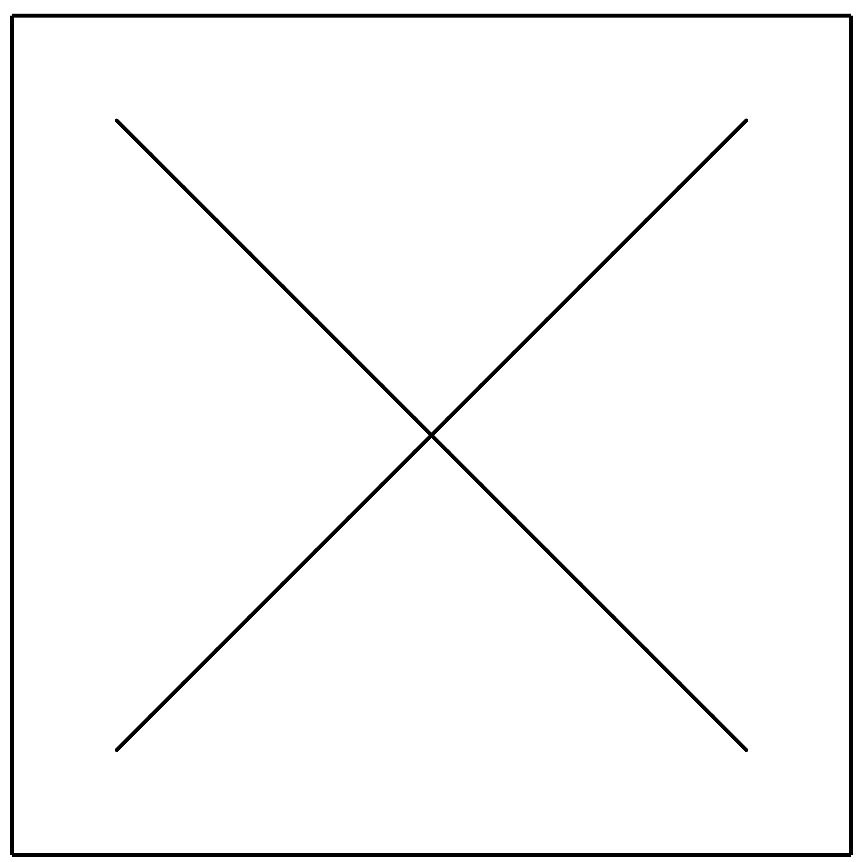
$- ab$ computing

$+ 4h_2 + 8h_3$

$_1, g_0 + 2g_1.$

Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

Build circuit with more gates to multiply (e.g.) 32-bit integers:



(Details omitted.)

Build circuit to co

32-bit integer r_i

given 4-bit integer

and 32-bit integers

register

read

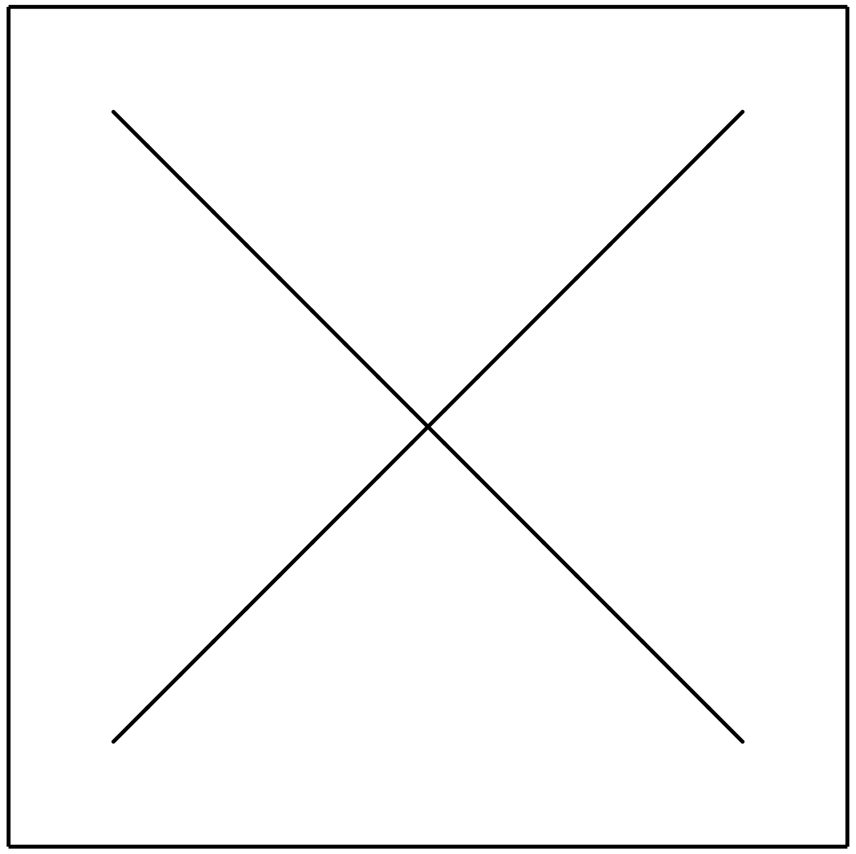
puting

h_3

$_1$.

Electricity takes time to
percolate through wires and gates.
If f_0, f_1, g_0, g_1 are stable
then h_0, h_1, h_2, h_3 are stable
a few moments later.

Build circuit with more gates
to multiply (e.g.) 32-bit integers:



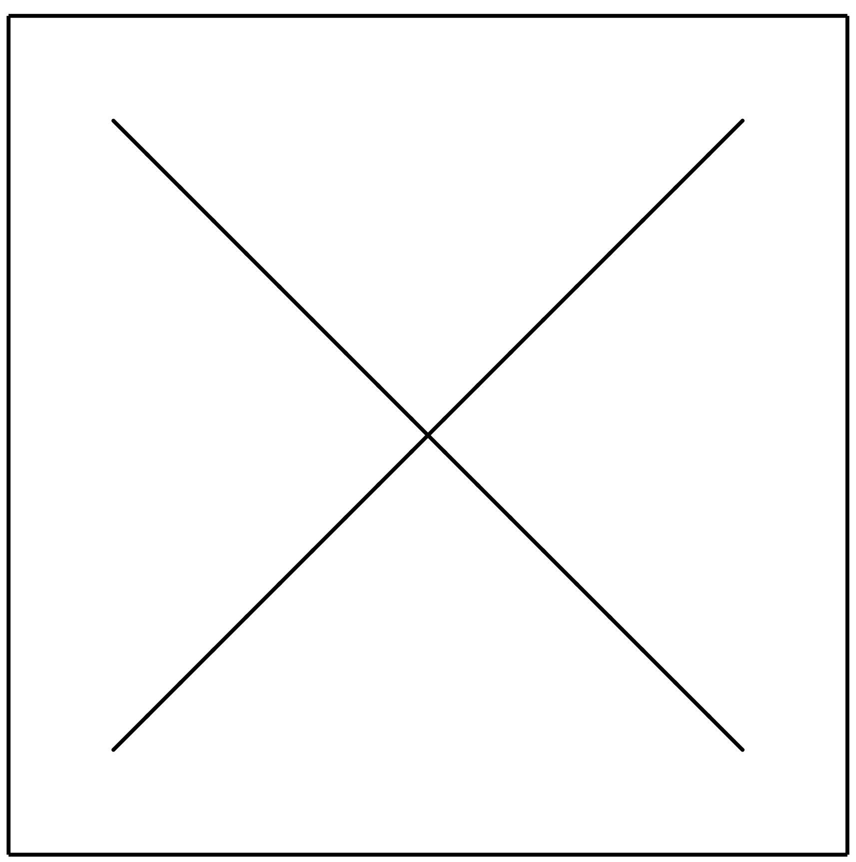
(Details omitted.)

Build circuit to compute
32-bit integer r_i
given 4-bit integer i
and 32-bit integers $r_0, r_1, \ldots$

register
read

Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

Build circuit with more gates to multiply (e.g.) 32-bit integers:

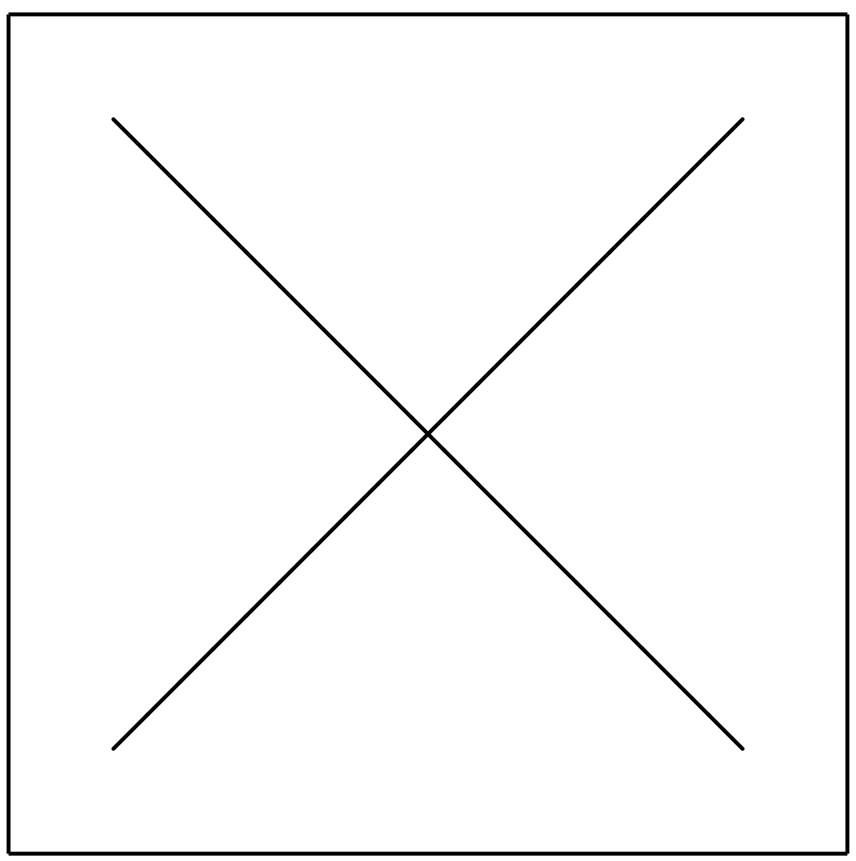


(Details omitted.)

Build circuit to compute 32-bit integer r_i given 4-bit integer i and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

Build circuit with more gates to multiply (e.g.) 32-bit integers:



(Details omitted.)

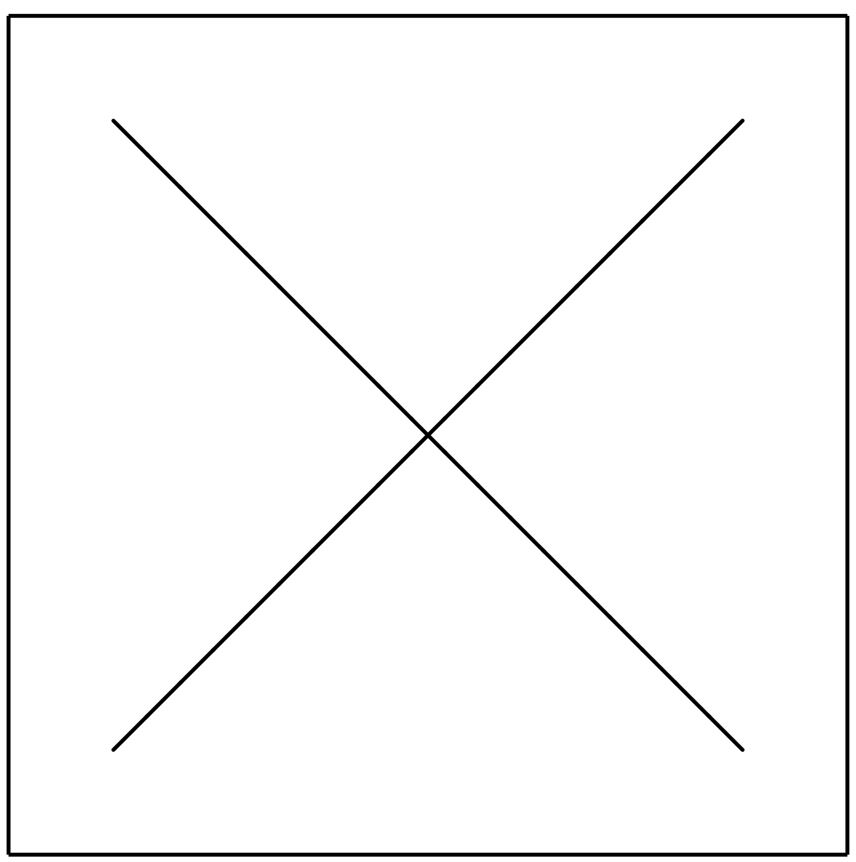
Build circuit to compute 32-bit integer r_i given 4-bit integer i and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

register read

Build circuit for "register write": $r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$ where $r'_j = r_j$ except $r'_i = s$.

Electricity takes time to percolate through wires and gates. If f_0, f_1, g_0, g_1 are stable then h_0, h_1, h_2, h_3 are stable a few moments later.

Build circuit with more gates to multiply (e.g.) 32-bit integers:



(Details omitted.)

Build circuit to compute 32-bit integer r_i given 4-bit integer i and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

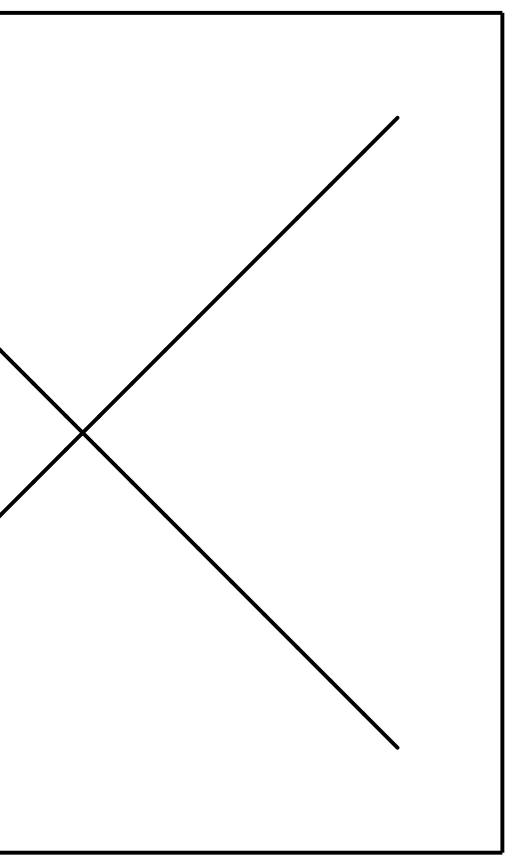
register read

Build circuit for "register write": $r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$ where $r'_j = r_j$ except $r'_i = s$.

Build circuit for addition. Etc.

ty takes time to
e through wires and gates.
g_0, g_1 are stable
h_1, h_2, h_3 are stable
oments later.

rcuit with more gates
ply (e.g.) 32-bit integers:



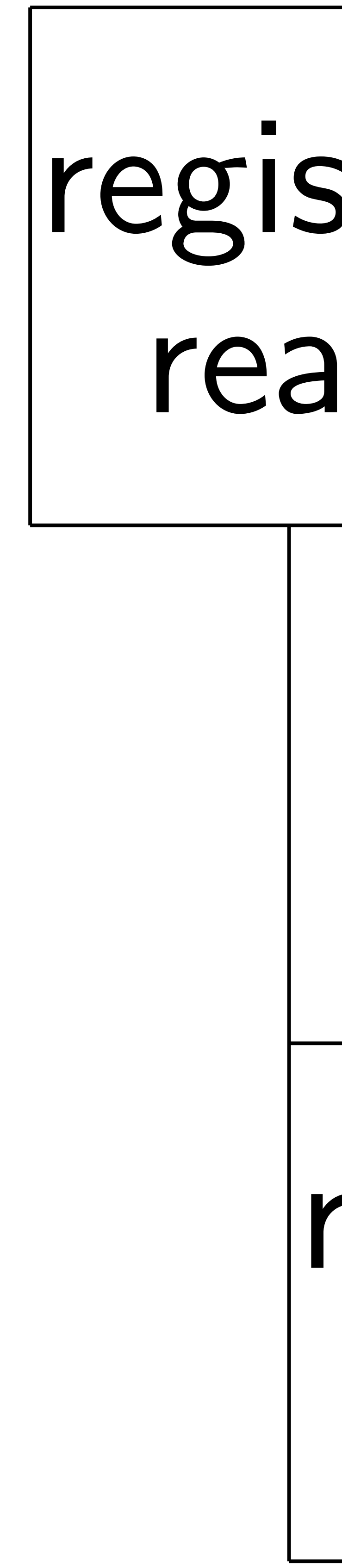
omitted.)

Build circuit to compute
32-bit integer r_i
given 4-bit integer i
and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

register
read

Build circuit for "register write":
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.
Build circuit for addition. Etc.

$r_0, \ldots, r$
where r'_ℓ



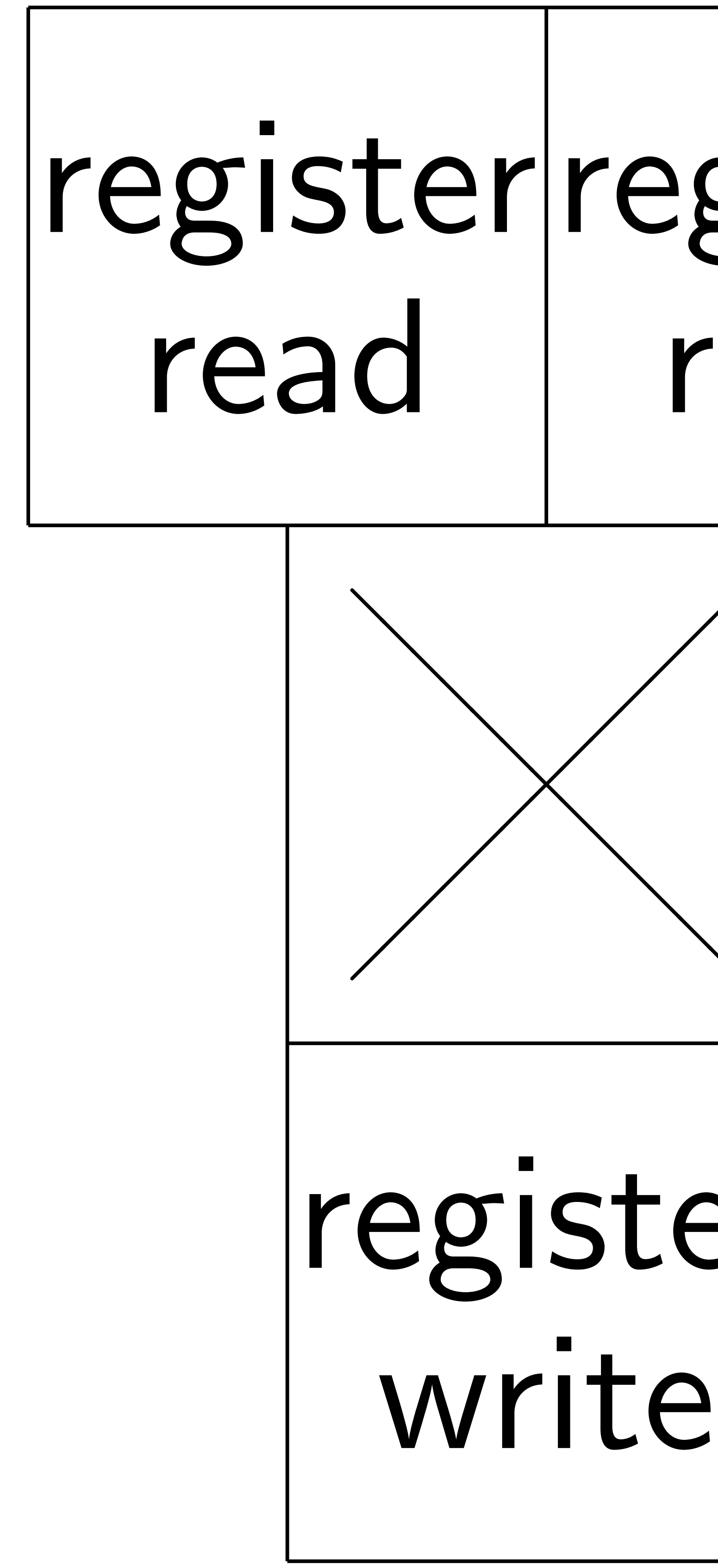
me to
wires and gates.
stable
are stable
ter.

more gates
32-bit integers:

Build circuit to compute
32-bit integer r_i
given 4-bit integer i
and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

Build circuit for “register write”:
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.
Build circuit for addition. Etc.

$r_0, \ldots, r_{15}, i, j, k \vdash$
where $r'_\ell = r_\ell$ exce



gates.

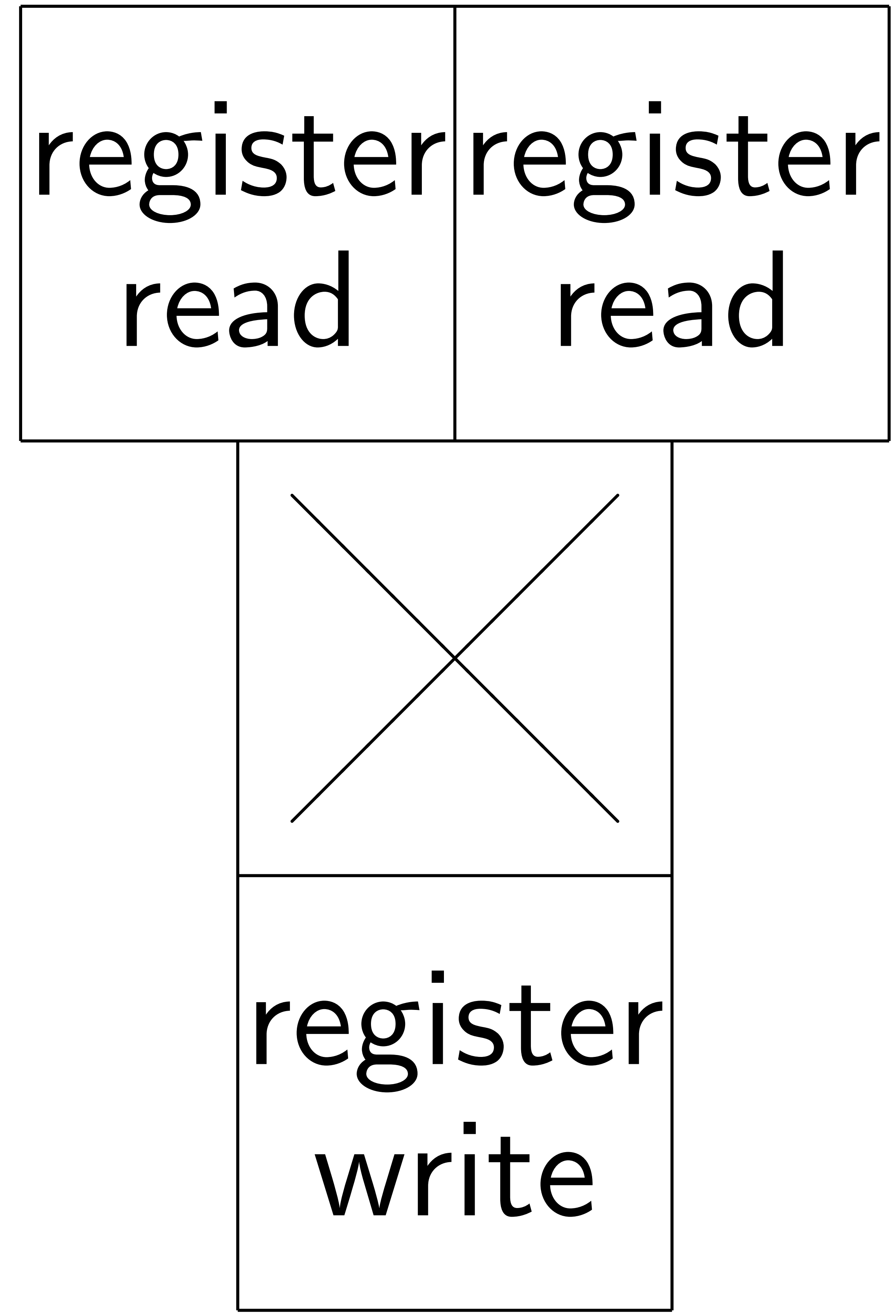
s

egers:

Build circuit to compute
32-bit integer r_i
given 4-bit integer i
and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

Build circuit for "register write":
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.
Build circuit for addition. Etc.

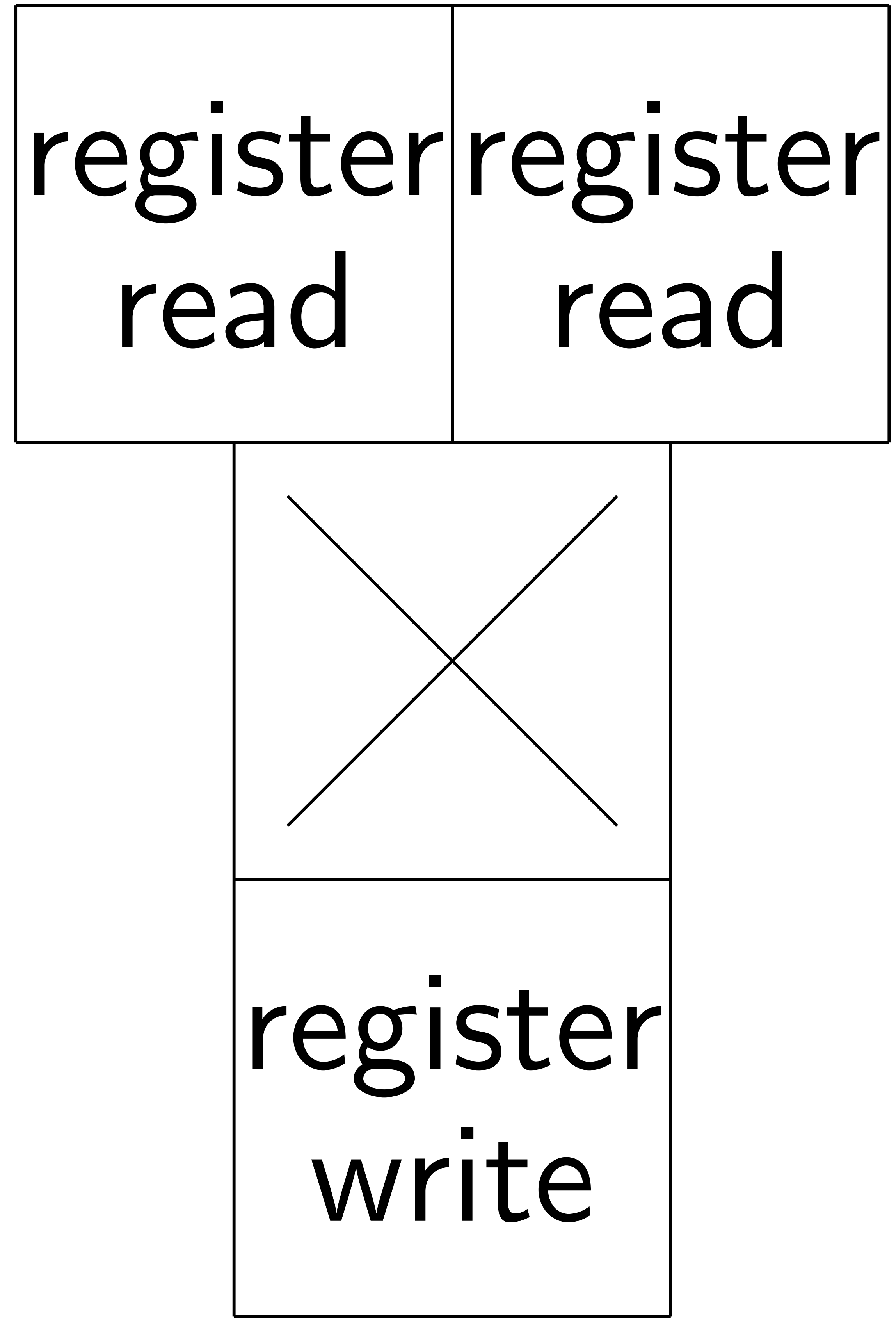
$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r$
where $r'_\ell = r_\ell$ except $r'_i = r_j$



Build circuit to compute
32-bit integer r_i
given 4-bit integer i
and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

Build circuit for "register write":
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.
Build circuit for addition. Etc.

$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



rcuit to compute

nteger r_i

bit integer i

bit integers $r_0, r_1, \ldots, r_{15}$:

rcuit for "register write":

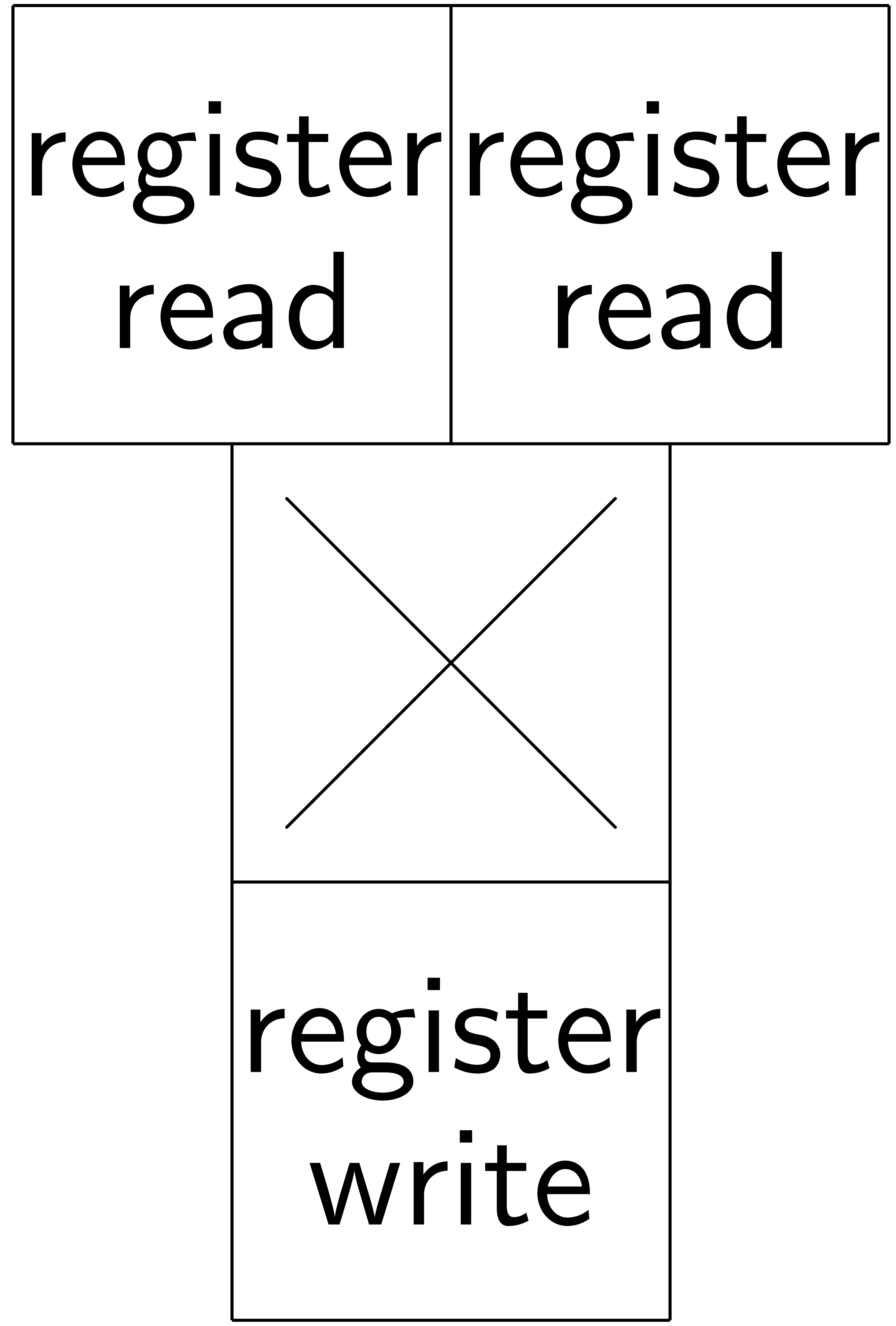
$_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$

$= r_j$ except $r'_i = s$.

rcuit for addition. Etc.

$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$

where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



Add mor

More ari

replace (

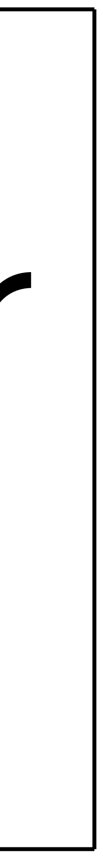
("$\times$", i,

("$+$", i,

mpute

i

s $r_0, r_1, \ldots, r_{15}$:



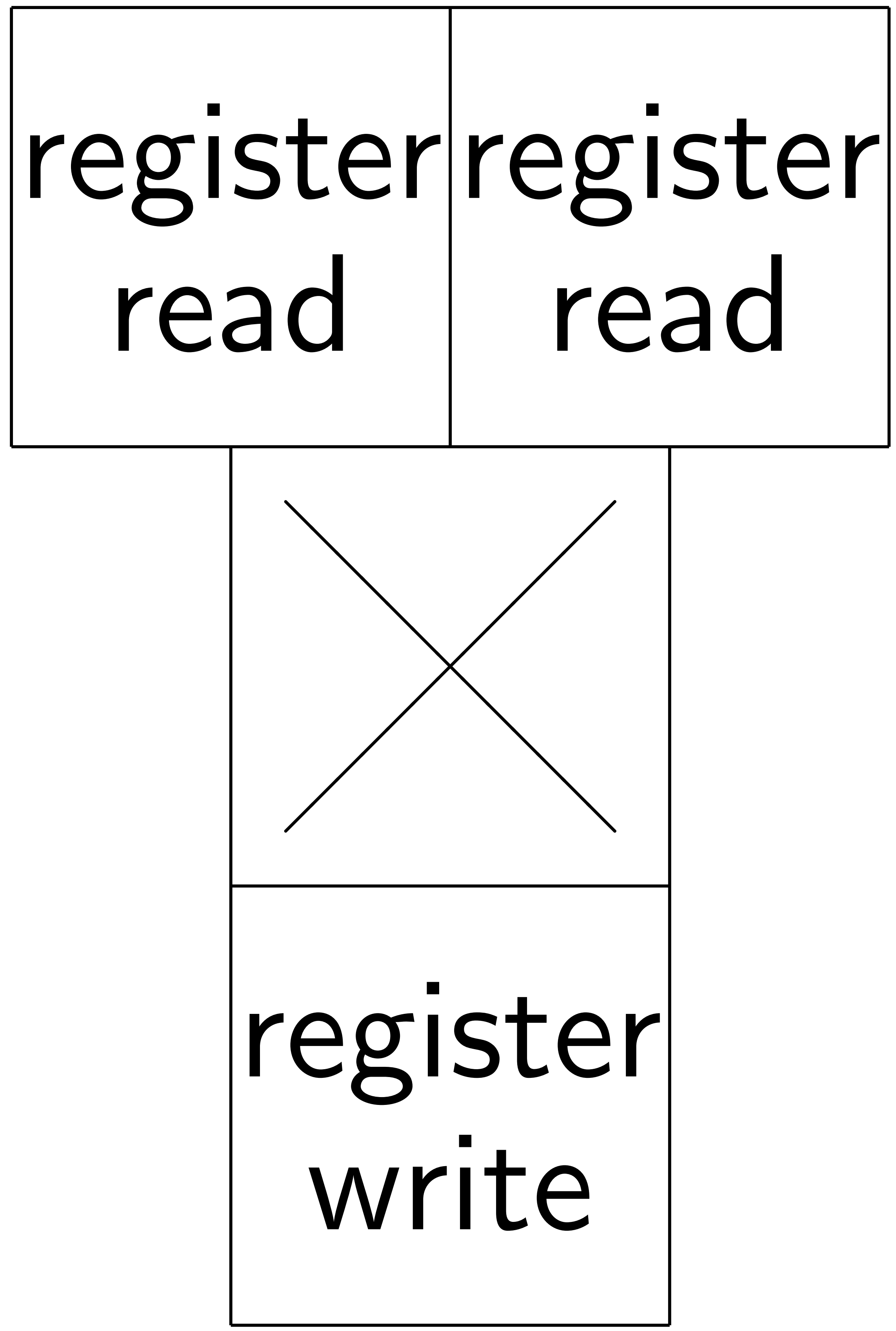
register write":

$r'_0, \ldots, r'_{15}$

pt $r'_i = s$.

ddition. Etc.

$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



Add more flexibilit

More arithmetic:

replace (i, j, k) wit

$(\text{“}\times\text{”}, i, j, k)$ and

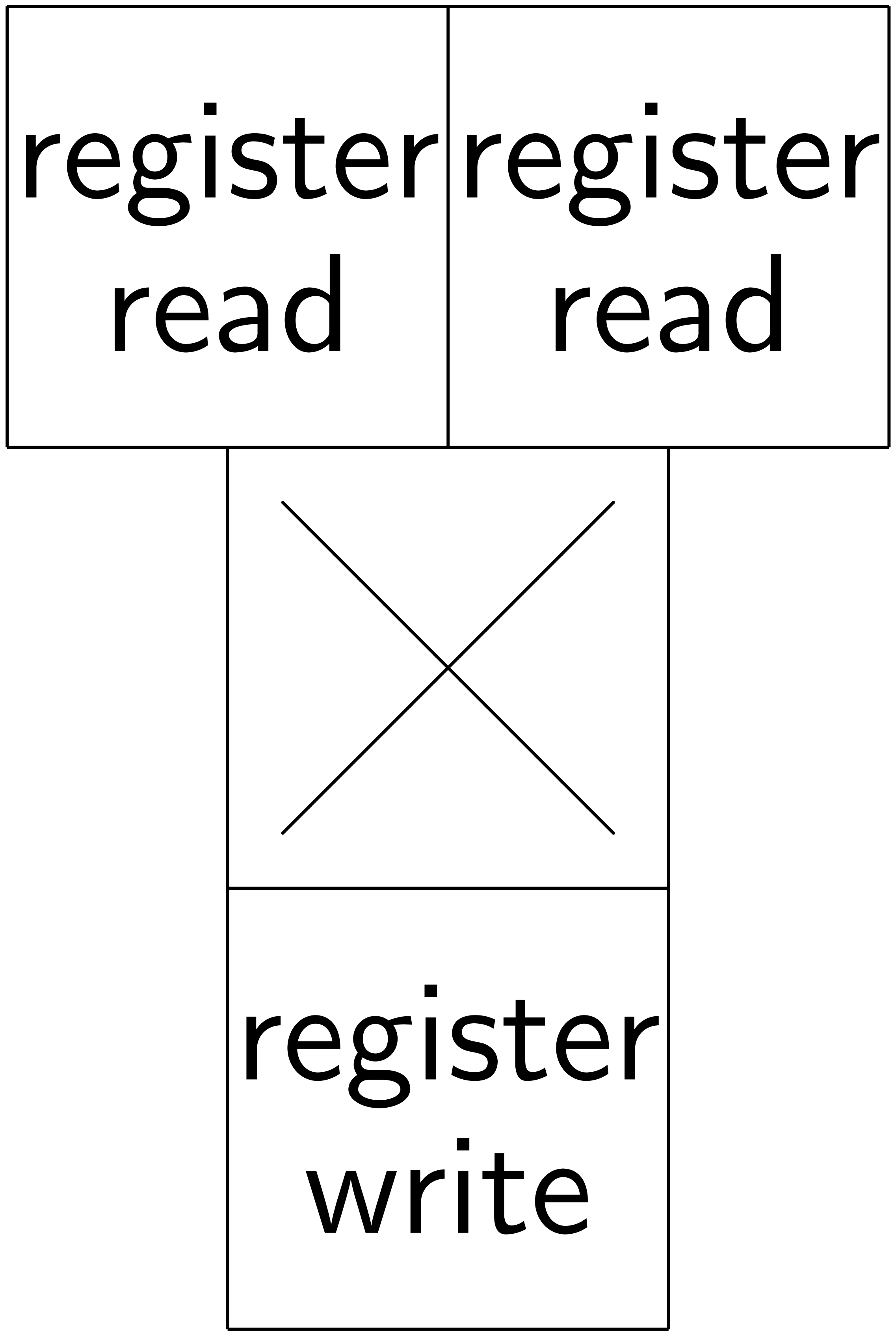
$(\text{“}+\text{”}, i, j, k)$ and r

$, r_{15}$:

ite":

tc.

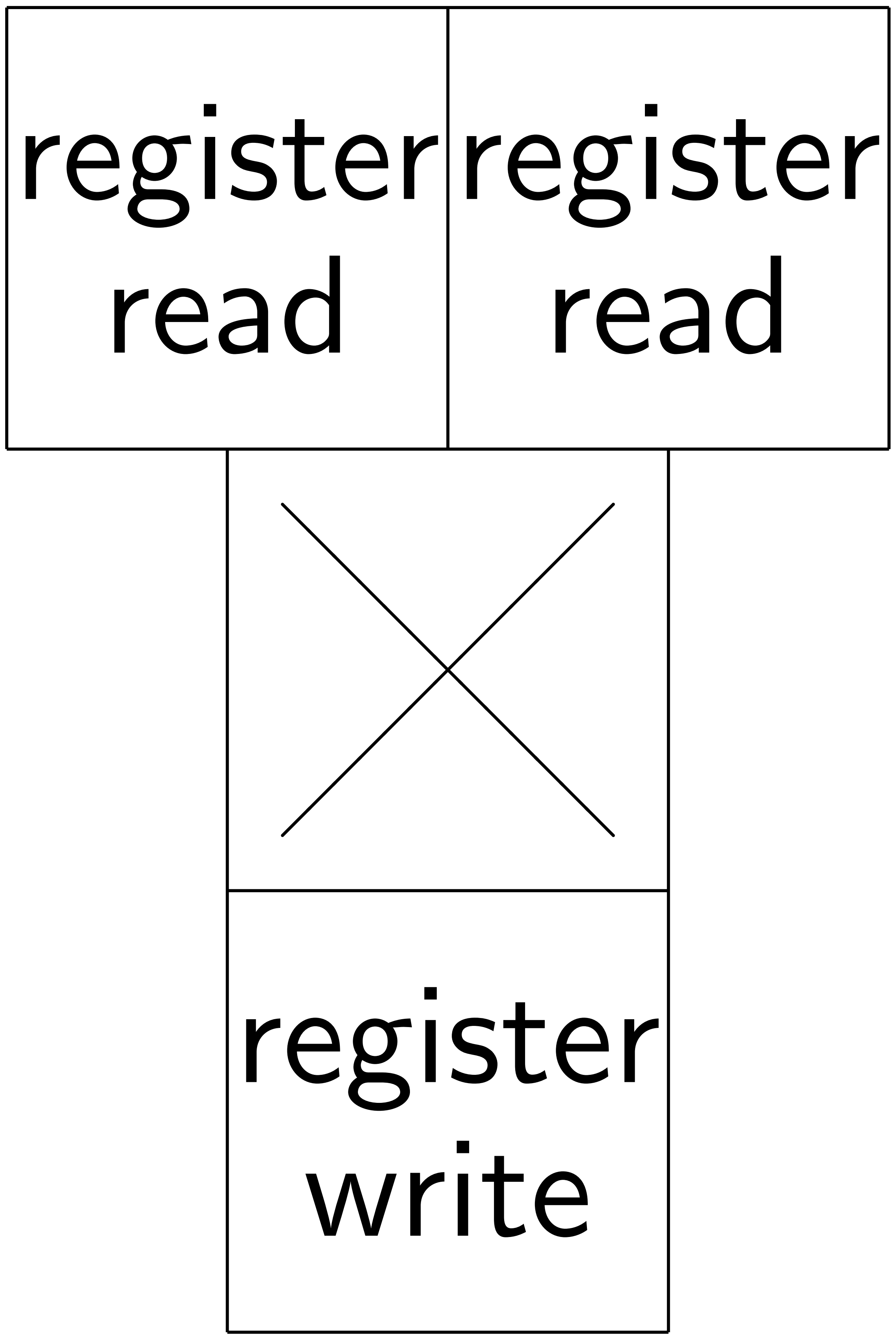
$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more optio

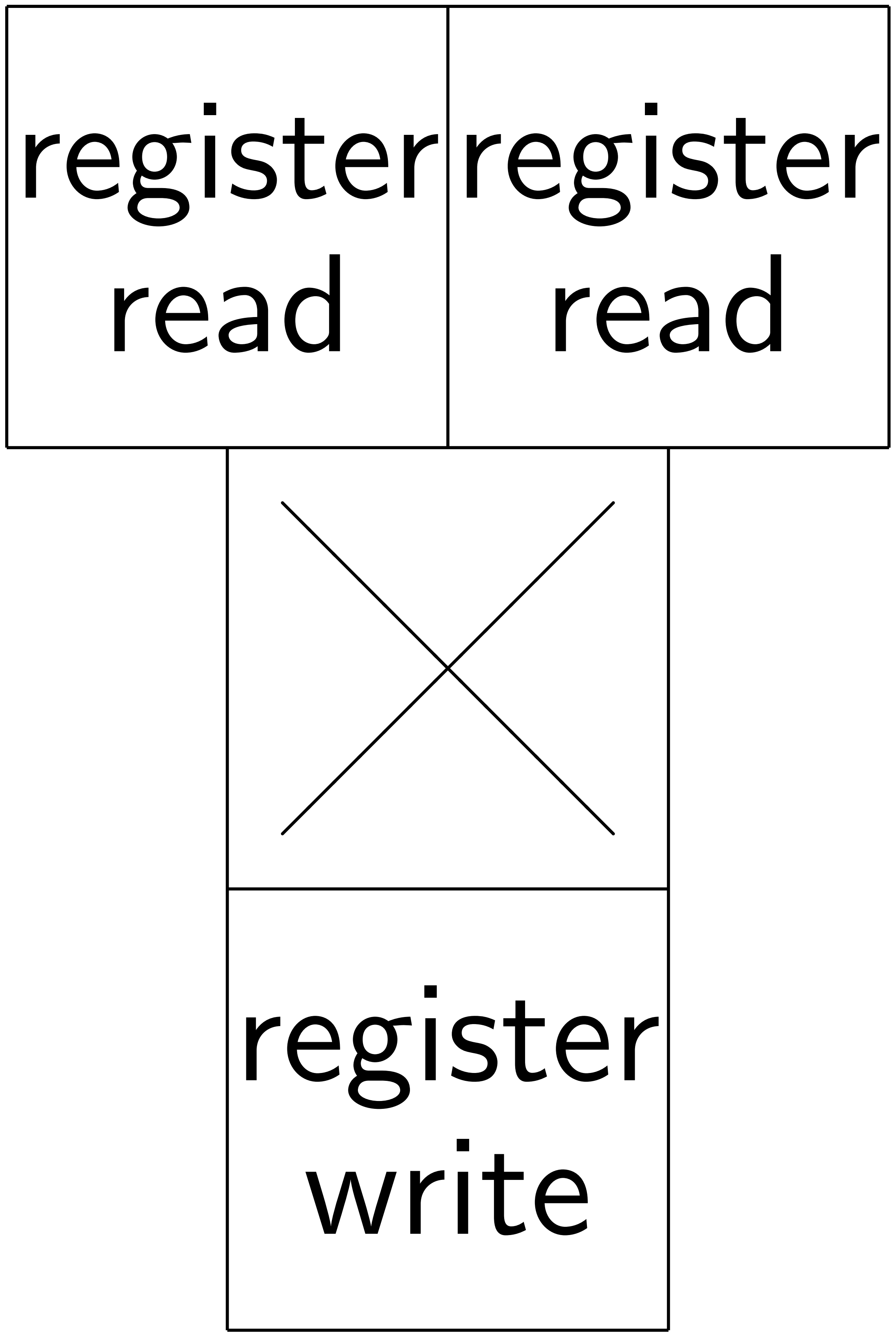
$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more options.

$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:

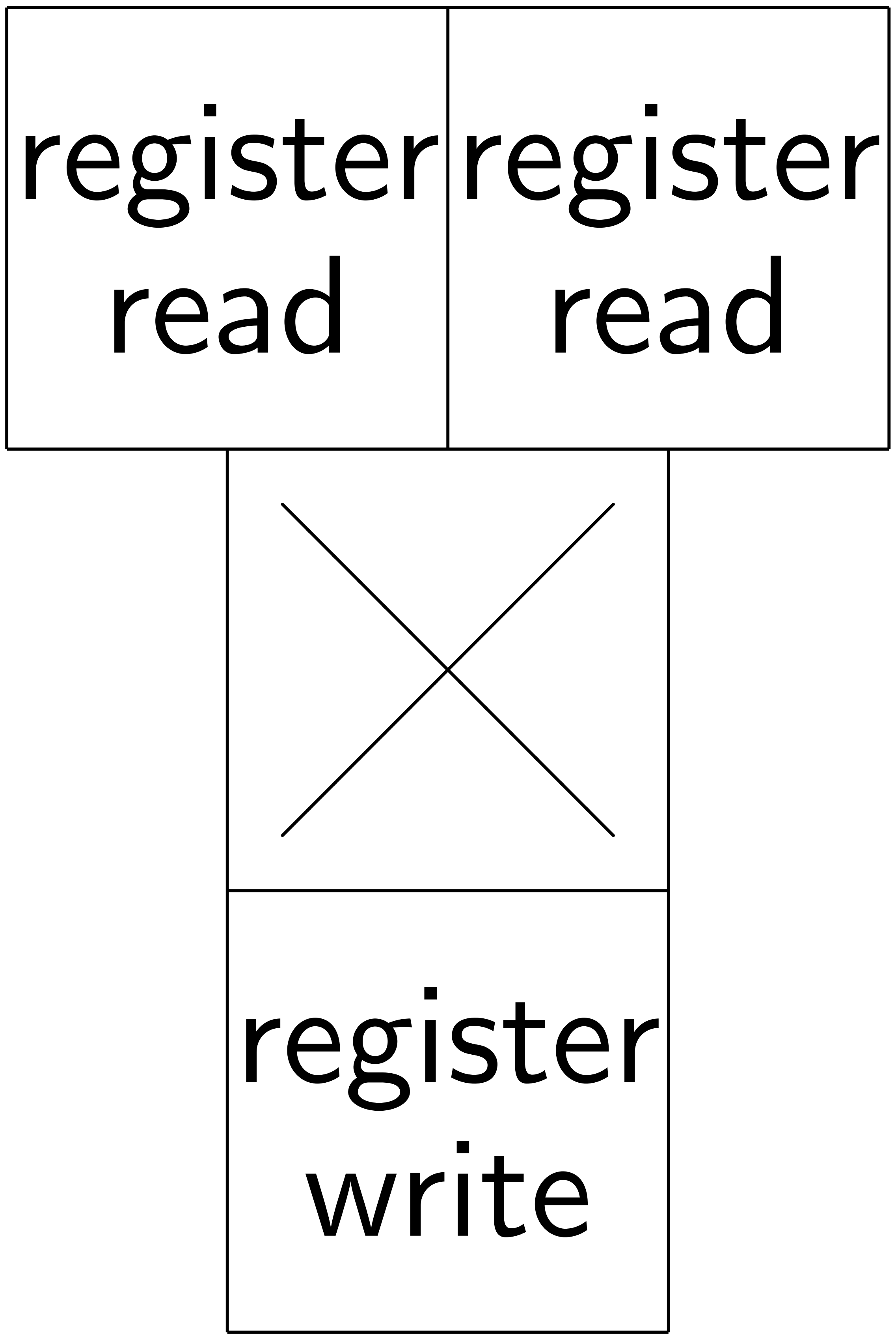


Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more options.

“Instruction fetch”:
$p \mapsto o_p, i_p, j_p, k_p, p'$.

$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



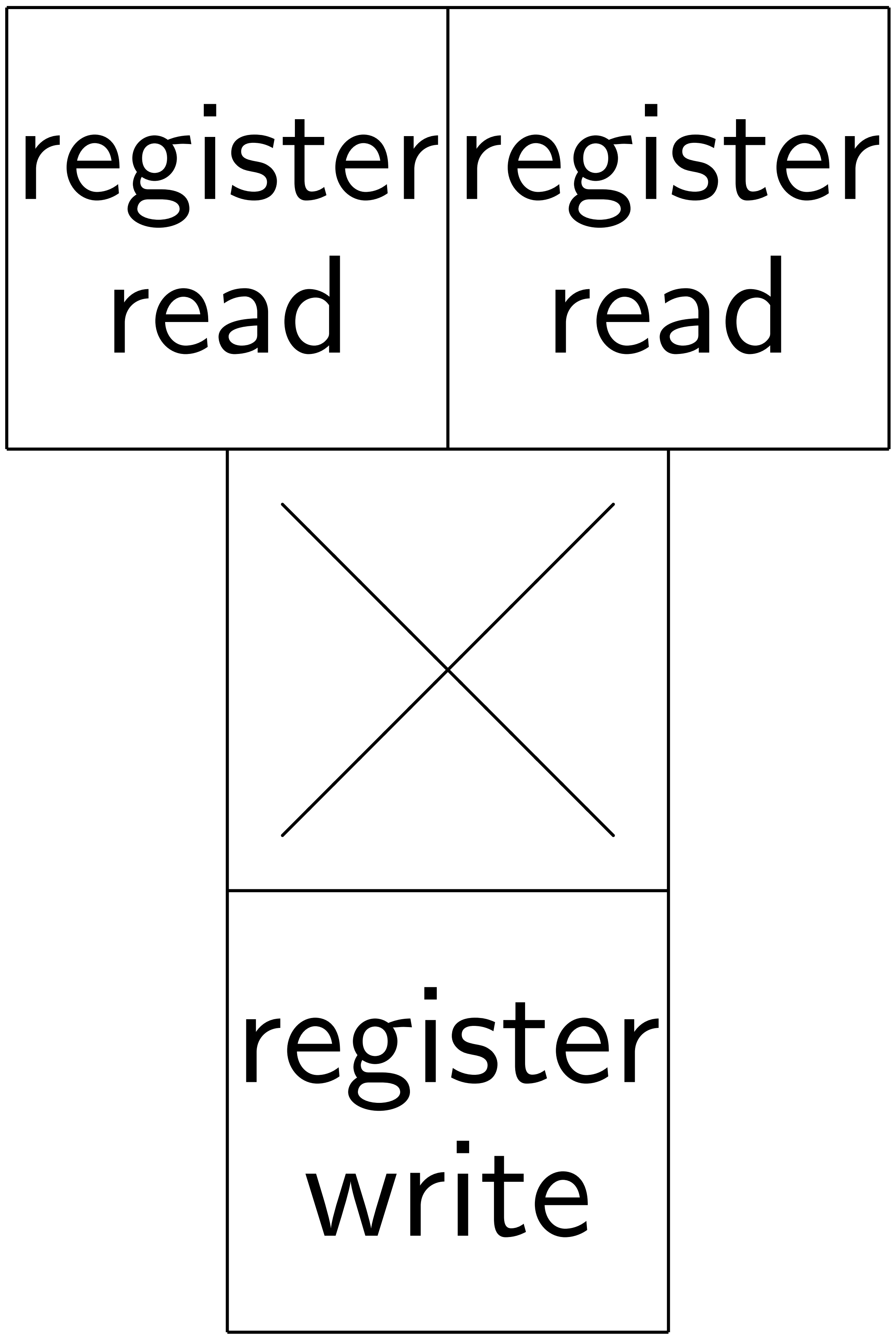
Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more options.

“Instruction fetch”:
$p \mapsto o_p, i_p, j_p, k_p, p'$.

“Instruction decode”:
decompression of compressed
format for o_p, i_p, j_p, k_p, p'.

$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more options.

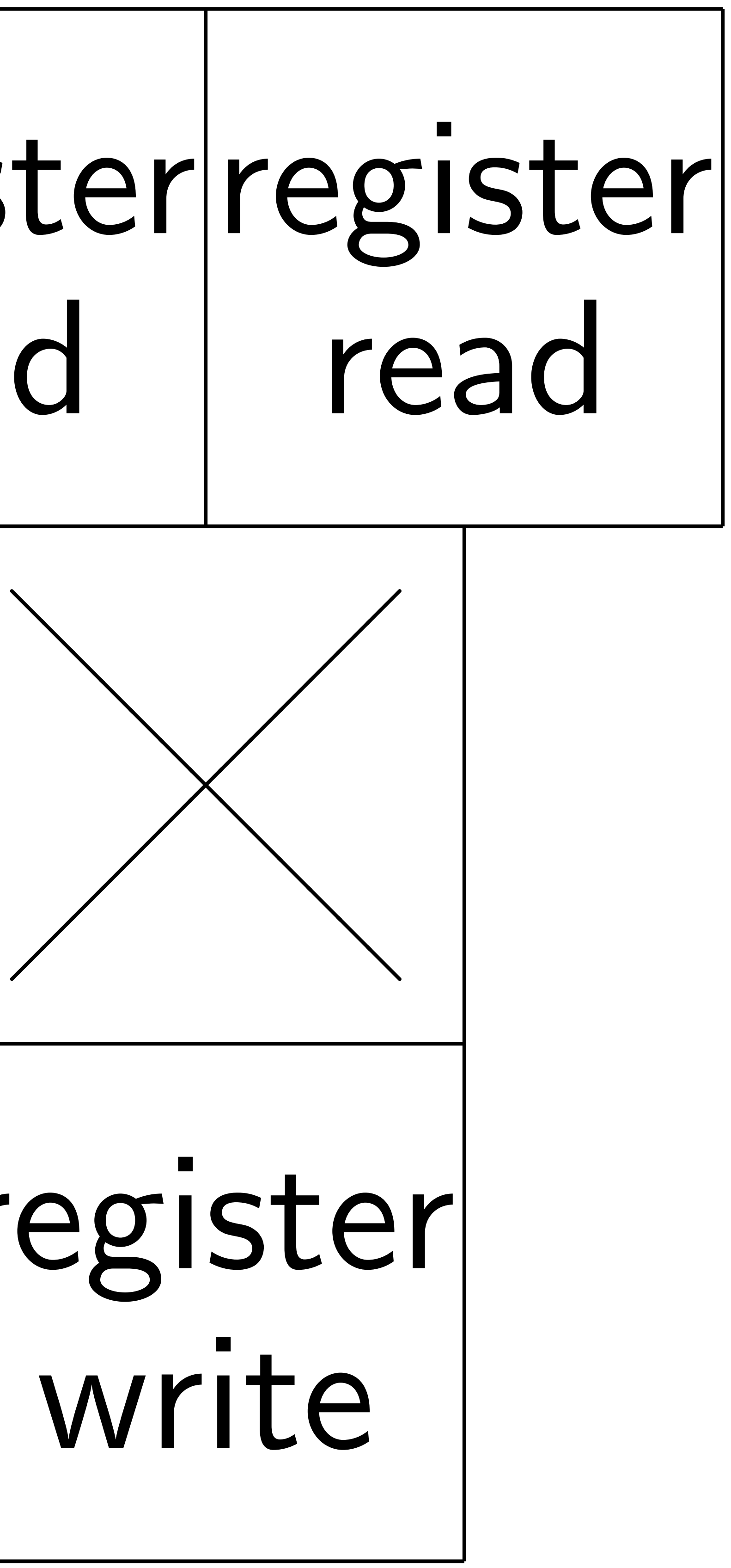
“Instruction fetch”:
$p \mapsto o_p, i_p, j_p, k_p, p'$.

“Instruction decode”:
decompression of compressed
format for o_p, i_p, j_p, k_p, p'.

More (but slower) storage:
“load” from and “store” to
larger “RAM” arrays.

$_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$

$= r_\ell$ except $r'_i = r_j r_k$:



Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more options.

“Instruction fetch”:
$p \mapsto o_p, i_p, j_p, k_p, p'$.

“Instruction decode”:
decompression of compressed format for o_p, i_p, j_p, k_p, p'.

More (but slower) storage:
“load” from and “store” to larger “RAM” arrays.

Build “fl
storing (

Hook (p
flip-flops

Hook ou
into the

At each
flip-flops
with the

Clock ne
for elect
all the w
from flip

$\to r'_0, \ldots, r'_{15}$

ept $r'_i = r_j r_k$:

gister
ead

er

Add more flexibility.

More arithmetic:
replace (i, j, k) with
("$\times$", i, j, k) and
("$+$", i, j, k) and more options.

"Instruction fetch":
$p \mapsto o_p, i_p, j_p, k_p, p'$.

"Instruction decode":
decompression of compressed
format for o_p, i_p, j_p, k_p, p'.

More (but slower) storage:
"load" from and "store" to
larger "RAM" arrays.

Build "flip-flops"
storing $(p, r_0, \ldots,$

Hook $(p, r_0, \ldots, r_1$
flip-flops into circu

Hook outputs $(p',$
into the same flip-

At each "clock tic
flip-flops are overw
with the outputs.

Clock needs to be
for electricity to p
all the way throug
from flip-flops to f

$'_{15}$

r_k:

Add more flexibility.

More arithmetic:
replace (i, j, k) with
$(\text{“}\times\text{”}, i, j, k)$ and
$(\text{“}+\text{”}, i, j, k)$ and more options.

“Instruction fetch”:
$p \mapsto o_p, i_p, j_p, k_p, p'$.

“Instruction decode”:
decompression of compressed
format for o_p, i_p, j_p, k_p, p'.

More (but slower) storage:
“load” from and “store” to
larger “RAM” arrays.

Build “flip-flops”
storing $(p, r_0, \ldots, r_{15})$.

Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

Hook outputs $(p', r'_0, \ldots, r'_{15}$
into the same flip-flops.

At each “clock tick”,
flip-flops are overwritten
with the outputs.

Clock needs to be slow enou
for electricity to percolate
all the way through the circu
from flip-flops to flip-flops.

Add more flexibility.

More arithmetic: replace (i, j, k) with $(\text{“}\times\text{”}, i, j, k)$ and $(\text{“}+\text{”}, i, j, k)$ and more options.

“Instruction fetch”: $p \mapsto o_p, i_p, j_p, k_p, p'$.

“Instruction decode”: decompression of compressed format for o_p, i_p, j_p, k_p, p'.

More (but slower) storage: “load” from and “store” to larger “RAM” arrays.

Build “flip-flops” storing $(p, r_0, \ldots, r_{15})$.

Hook $(p, r_0, \ldots, r_{15})$ flip-flops into circuit inputs.

Hook outputs $(p', r'_0, \ldots, r'_{15})$ into the same flip-flops.

At each “clock tick”, flip-flops are overwritten with the outputs.

Clock needs to be slow enough for electricity to percolate all the way through the circuit, from flip-flops to flip-flops.

re flexibility.

thmetic:

(i, j, k) with

$j, k)$ and

$j, k)$ and more options.

tion fetch":

i_p, j_p, k_p, p'.

tion decode":

ression of compressed

or o_p, i_p, j_p, k_p, p'.

ut slower) storage:

rom and "store" to

RAM" arrays.

Build "flip-flops"

storing $(p, r_0, \ldots, r_{15})$.

Hook $(p, r_0, \ldots, r_{15})$

flip-flops into circuit inputs.

Hook outputs $(p', r_0', \ldots, r_{15}')$

into the same flip-flops.

At each "clock tick",

flip-flops are overwritten

with the outputs.

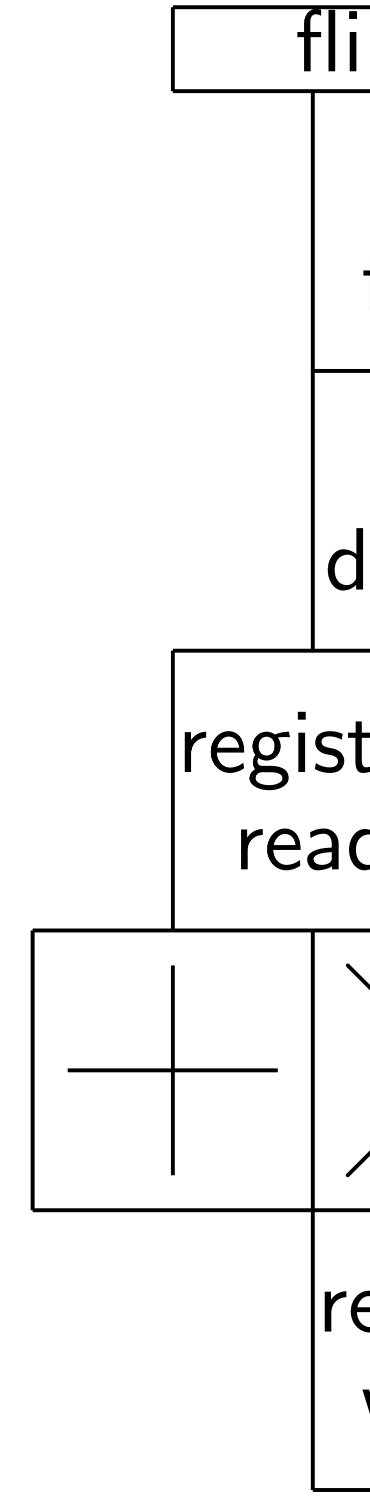
Clock needs to be slow enough

for electricity to percolate

all the way through the circuit,

from flip-flops to flip-flops.

Now hav



Further

but orth

ty.

ch

more options.

':

p'.

le":

compressed

$_p$, k_p, p'.

storage:

"store" to

ays.

Build "flip-flops"
storing $(p, r_0, \ldots, r_{15})$.

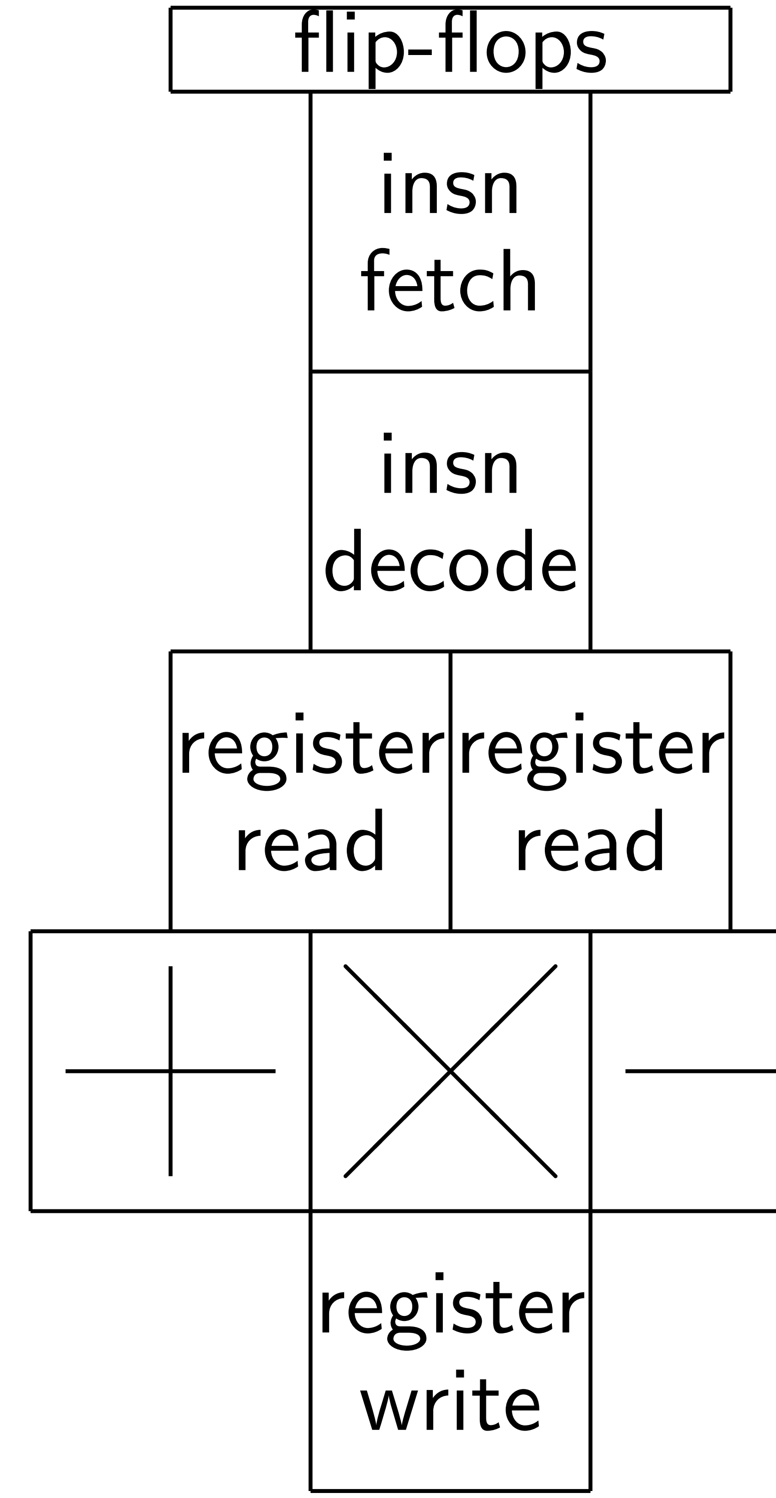
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

Hook outputs $(p', r'_0, \ldots, r'_{15})$
into the same flip-flops.

At each "clock tick",
flip-flops are overwritten
with the outputs.

Clock needs to be slow enough
for electricity to percolate
all the way through the circuit,
from flip-flops to flip-flops.

Now have semi-fle



Further flexibility i
but orthogonal to

ns.

d

Build "flip-flops" storing $(p, r_0, \ldots, r_{15})$.

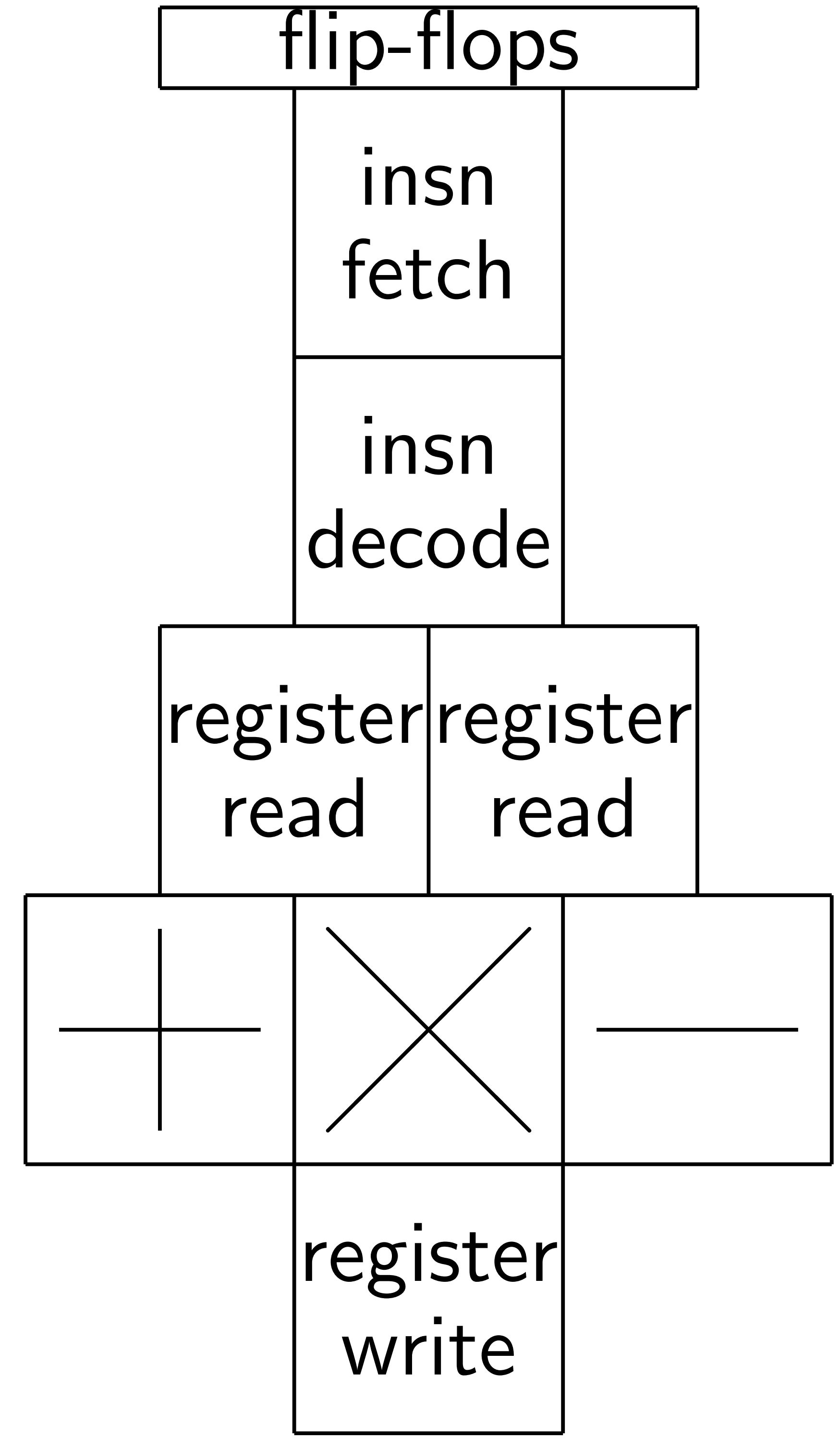
Hook $(p, r_0, \ldots, r_{15})$ flip-flops into circuit inputs.

Hook outputs $(p', r'_0, \ldots, r'_{15})$ into the same flip-flops.

At each "clock tick", flip-flops are overwritten with the outputs.

Clock needs to be slow enough for electricity to percolate all the way through the circuit, from flip-flops to flip-flops.

Now have semi-flexible CPU:



Further flexibility is useful but orthogonal to this talk.

Build “flip-flops”
storing $(p, r_0, \ldots, r_{15})$.

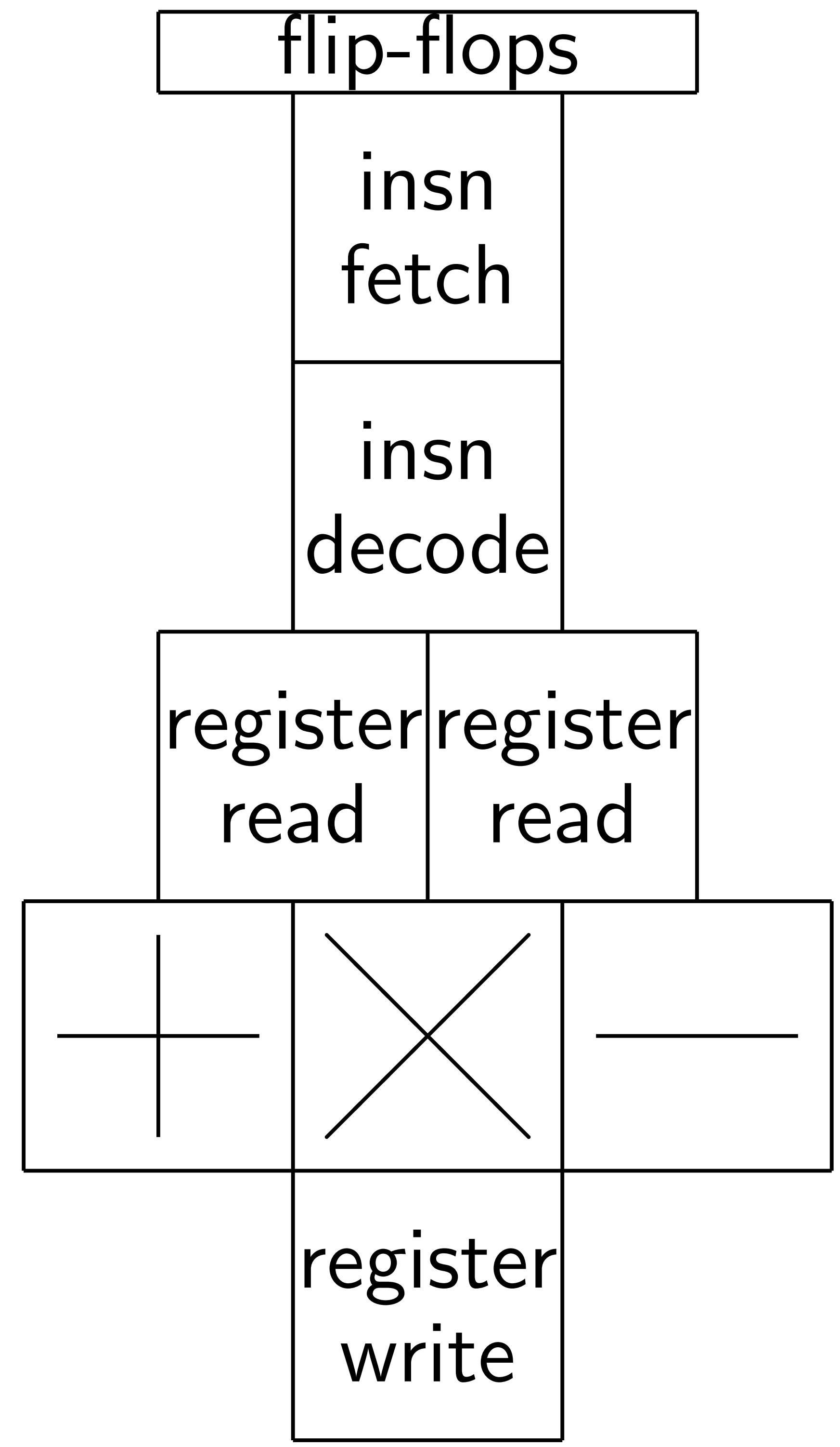
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

Hook outputs $(p', r'_0, \ldots, r'_{15})$
into the same flip-flops.

At each “clock tick”,
flip-flops are overwritten
with the outputs.

Clock needs to be slow enough
for electricity to percolate
all the way through the circuit,
from flip-flops to flip-flops.

Now have semi-flexible CPU:



Further flexibility is useful
but orthogonal to this talk.

lip-flops”

$(p, r_0, \ldots, r_{15})$.

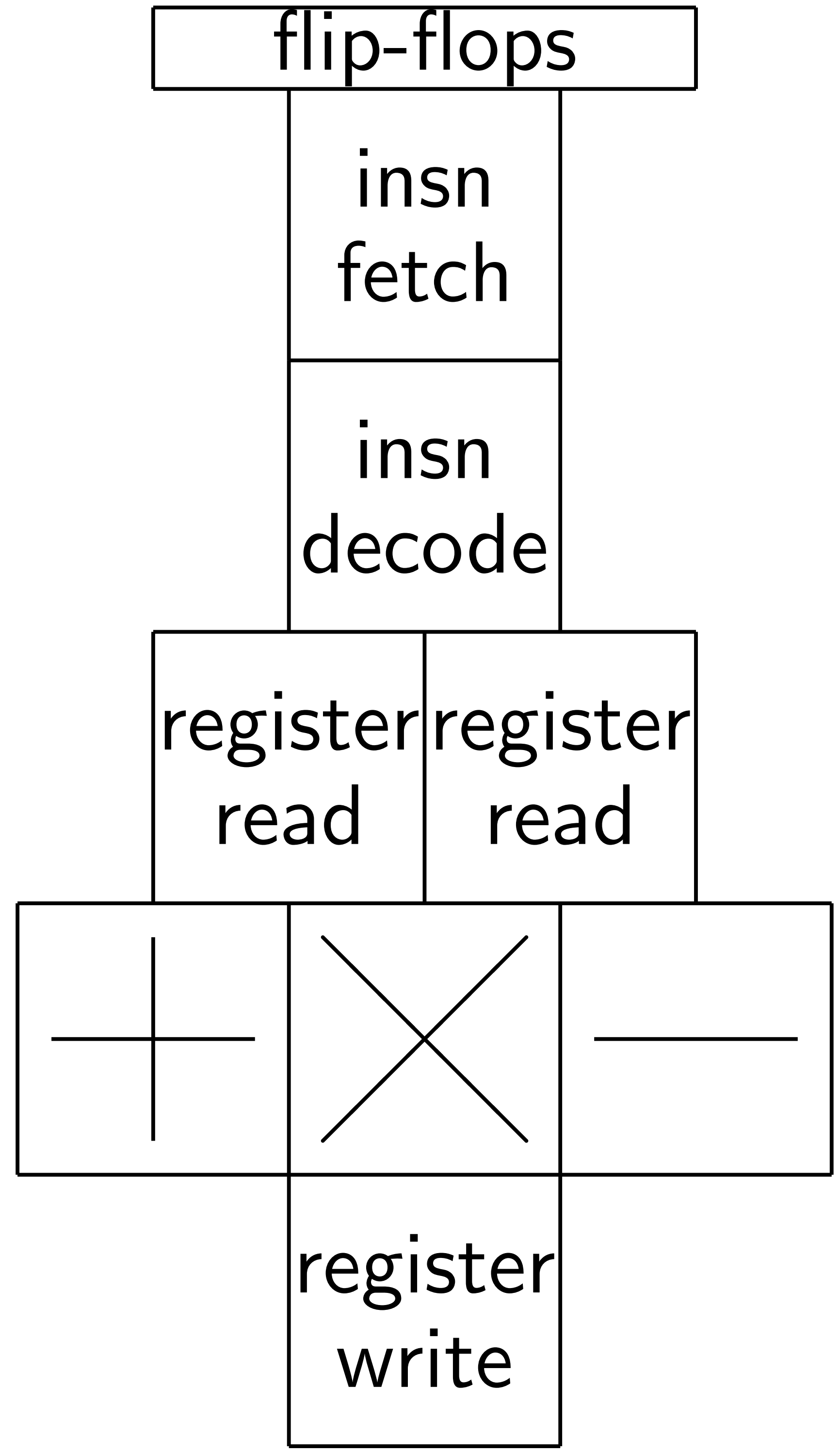
$p, r_0, \ldots, r_{15})$
s into circuit inputs.

utputs $(p', r'_0, \ldots, r'_{15})$
same flip-flops.

“clock tick”,
s are overwritten
outputs.

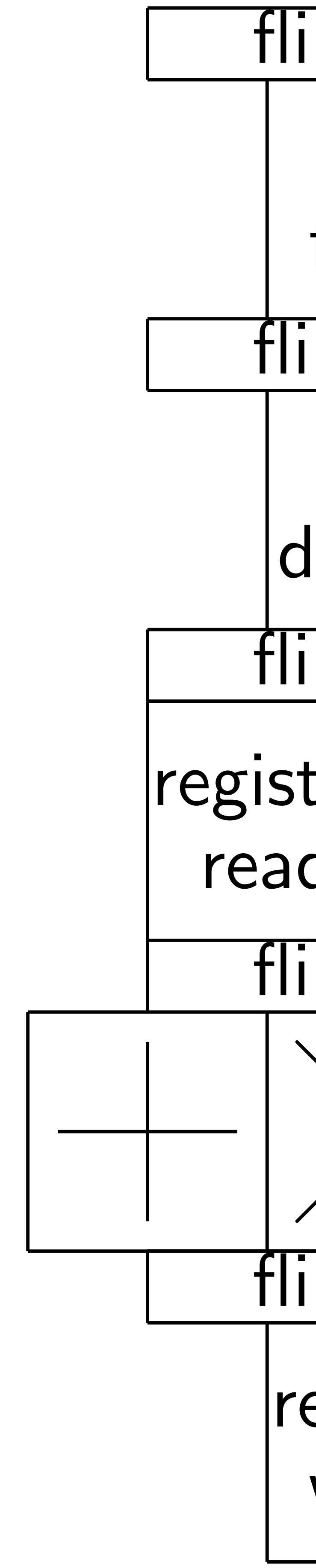
eeds to be slow enough
ricity to percolate
vay through the circuit,
-flops to flip-flops.

Now have semi-flexible CPU:



Further flexibility is useful
but orthogonal to this talk.

“Pipelin



r_{15}).

$_{15}$)

uit inputs.

$r'_0, \ldots, r'_{15}$)

-flops.

k”,

vritten

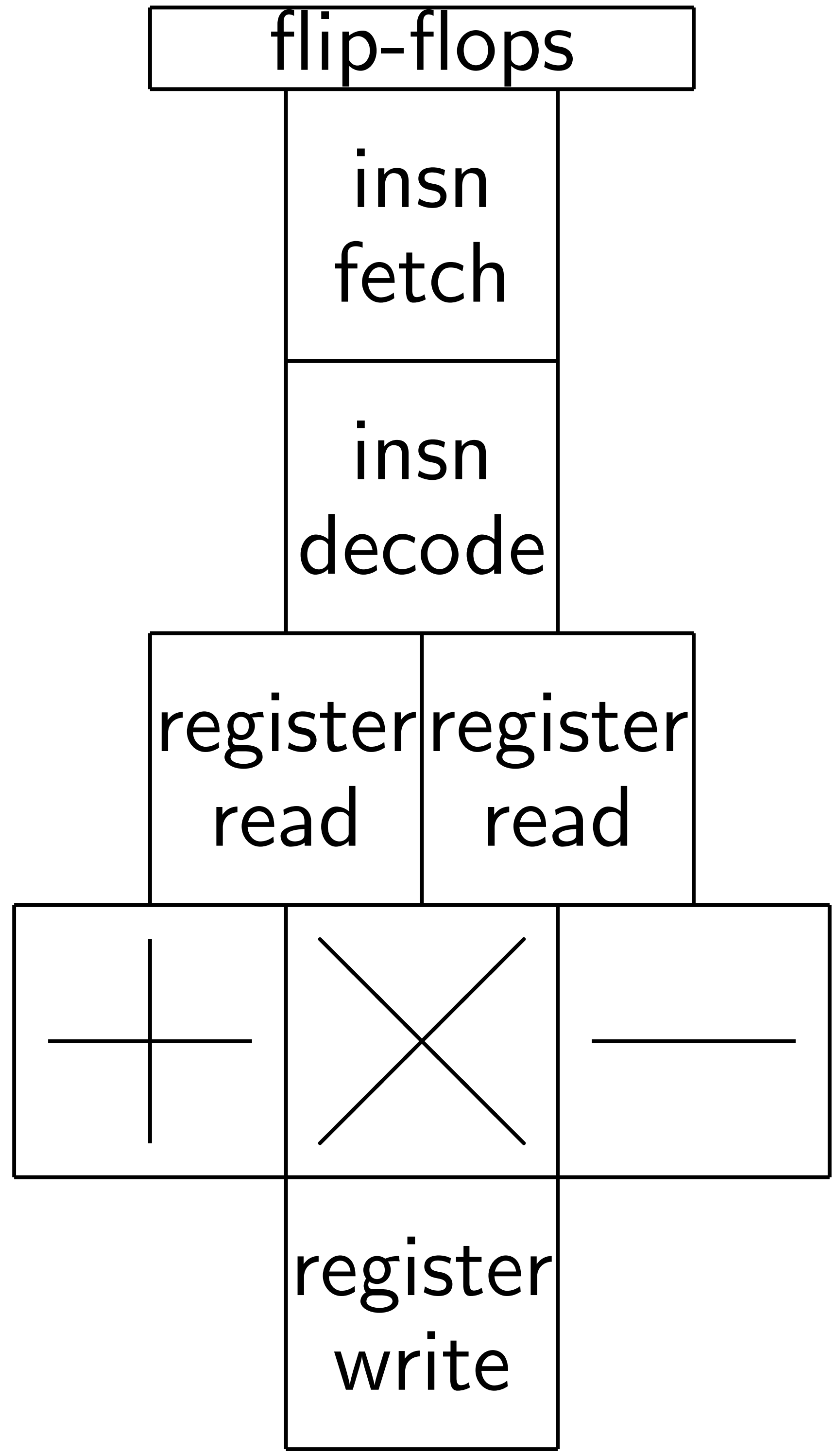
slow enough

ercolate

h the circuit,

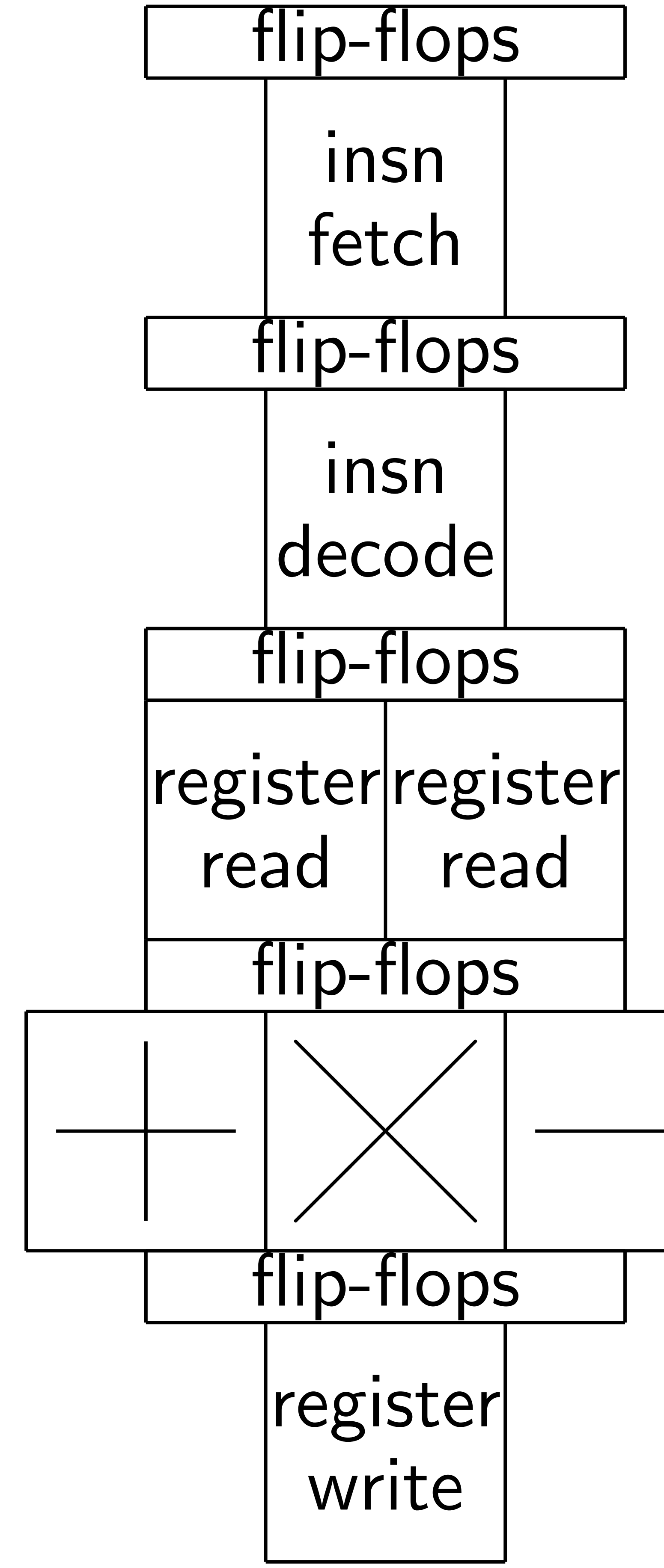
flip-flops.

Now have semi-flexible CPU:



Further flexibility is useful but orthogonal to this talk.

“Pipelining” allow

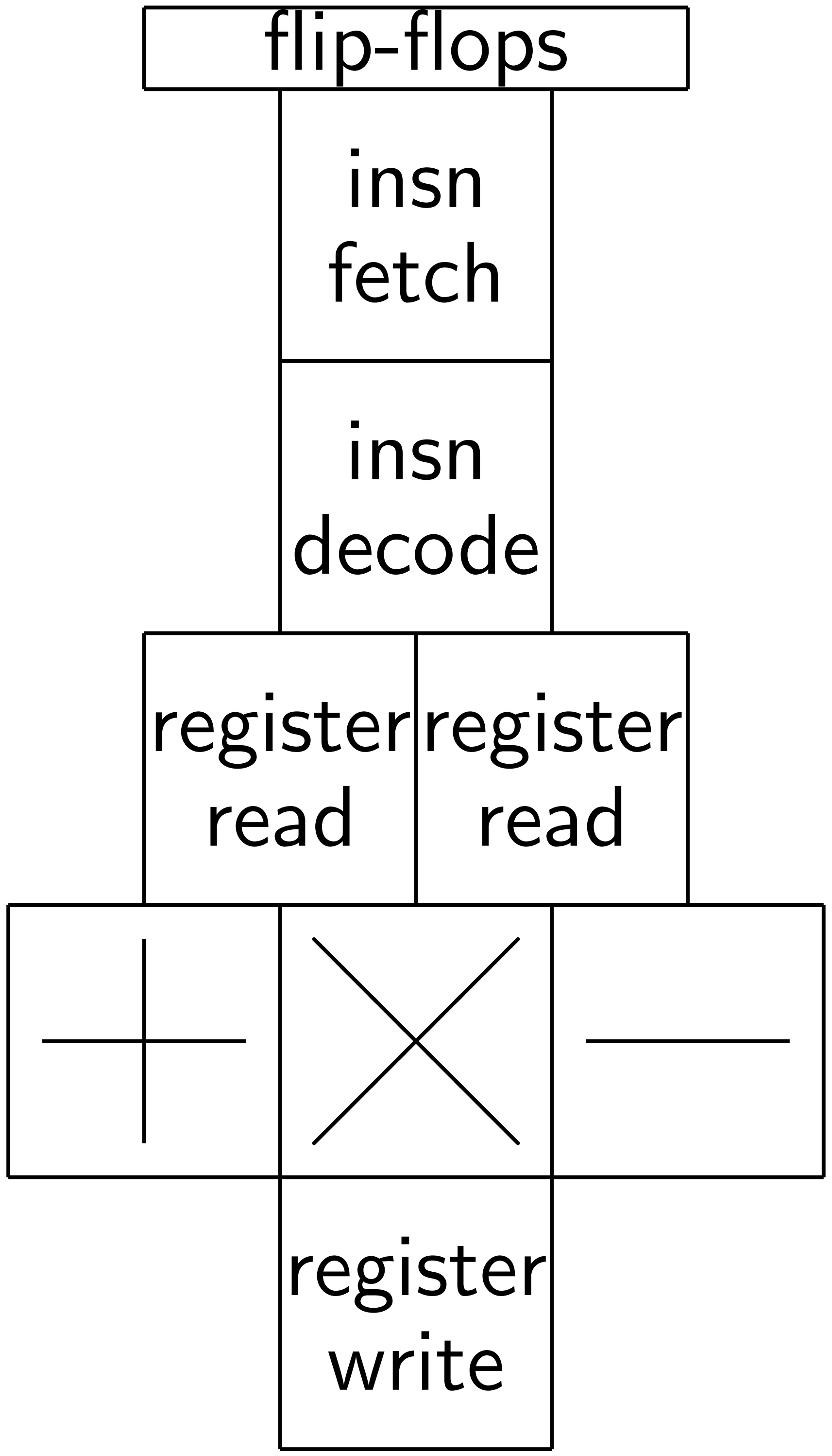


s)

ugh

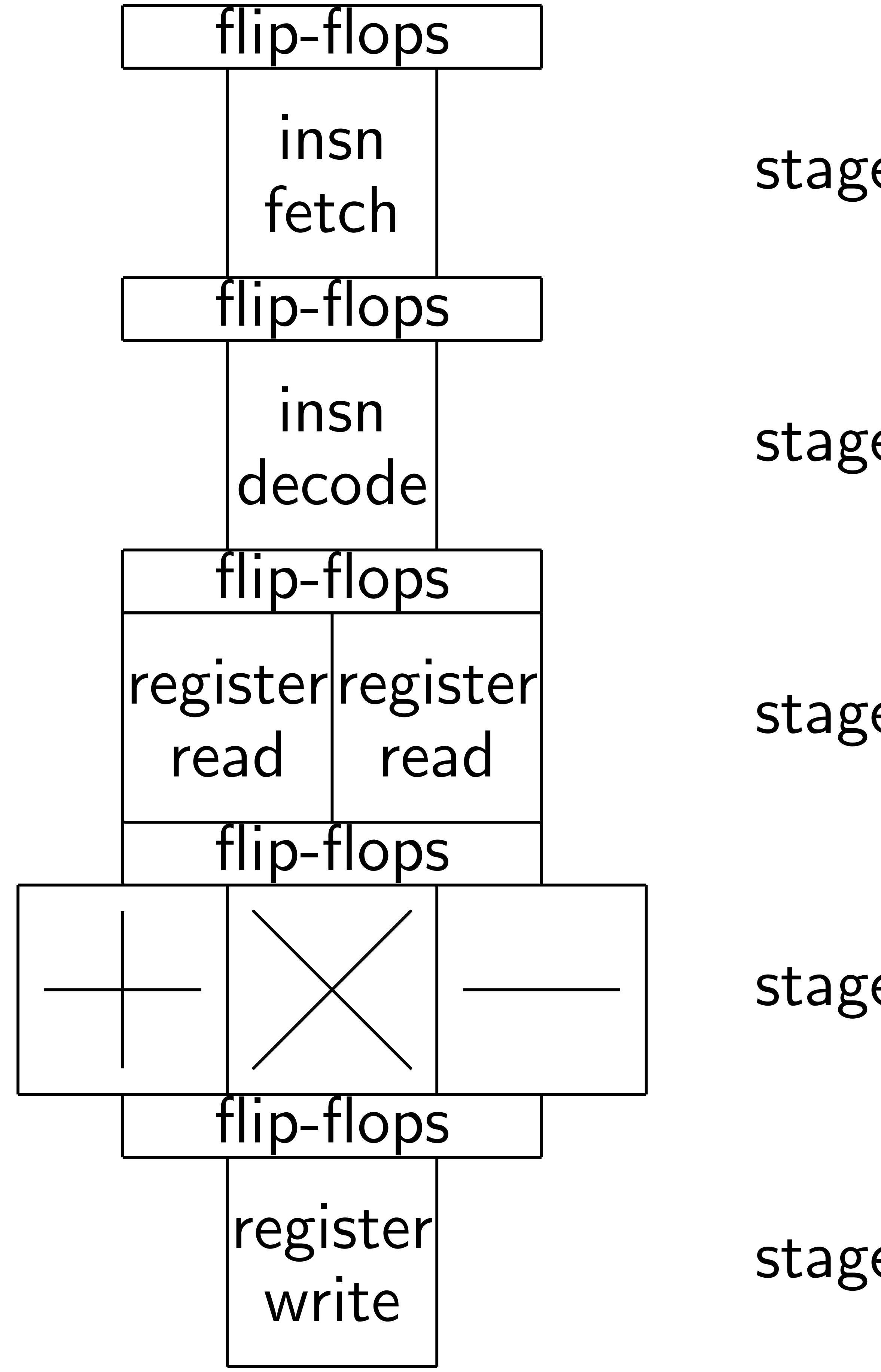
uit,

Now have semi-flexible CPU:

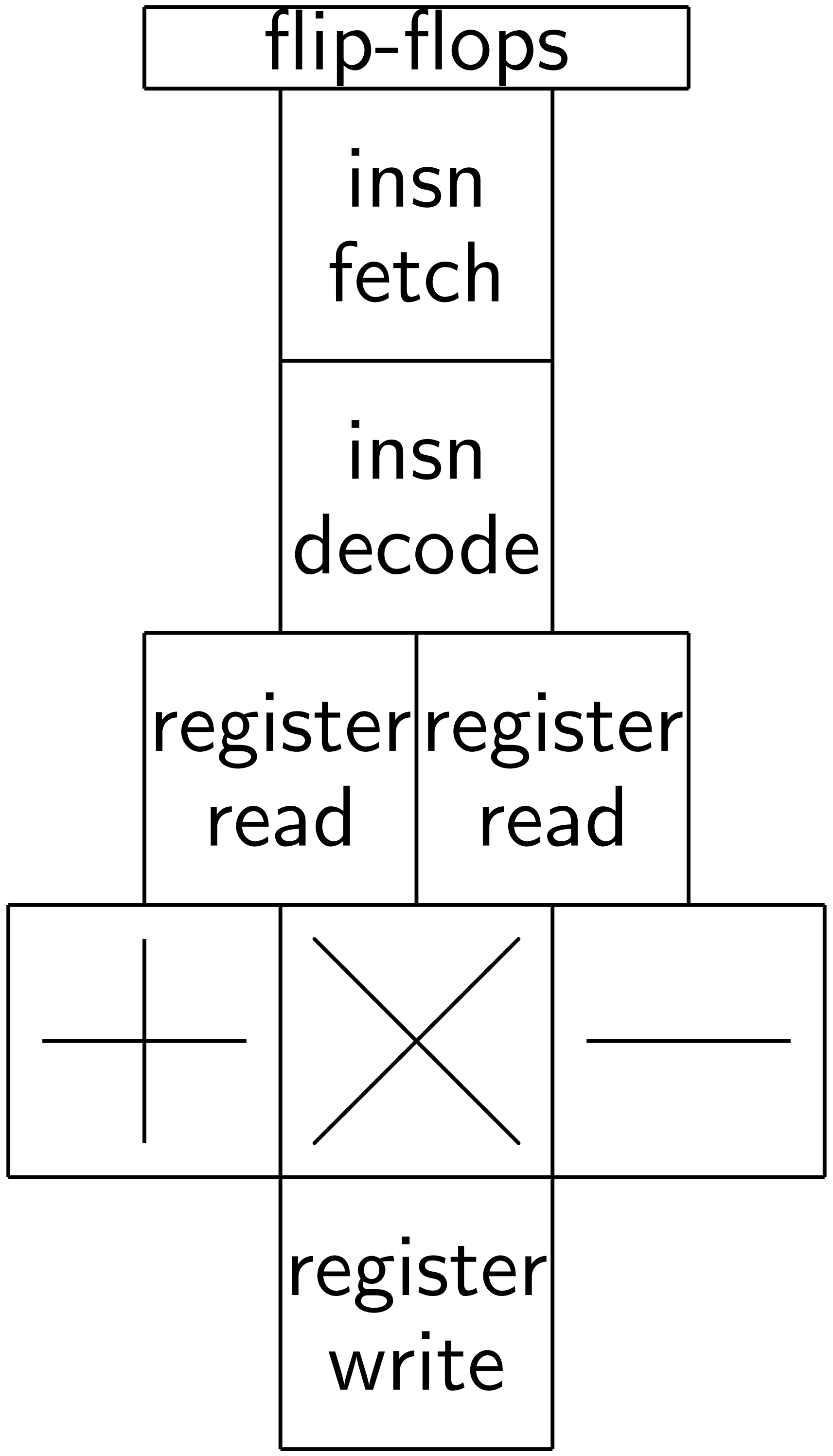


Further flexibility is useful but orthogonal to this talk.

“Pipelining” allows faster cl

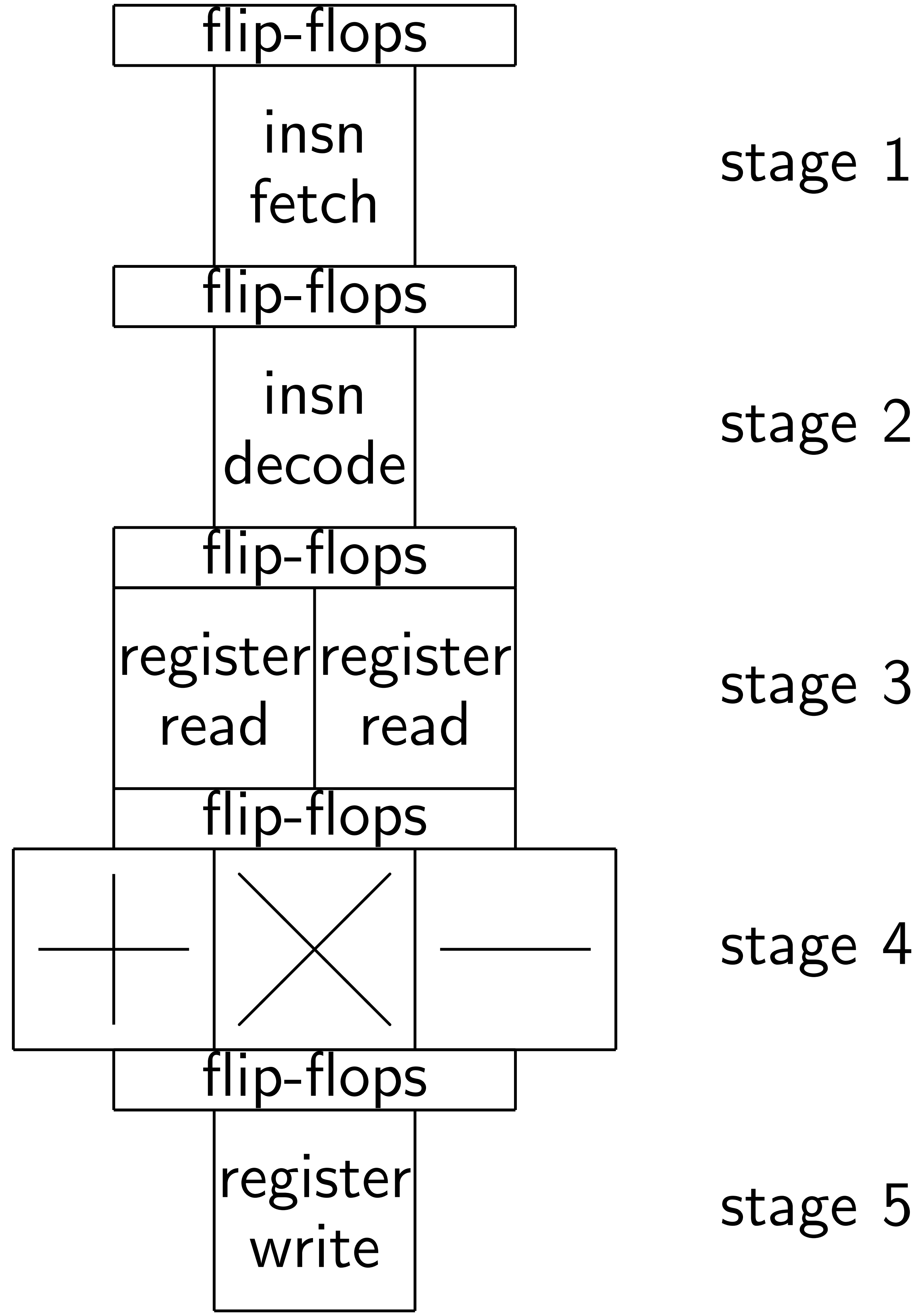


Now have semi-flexible CPU:

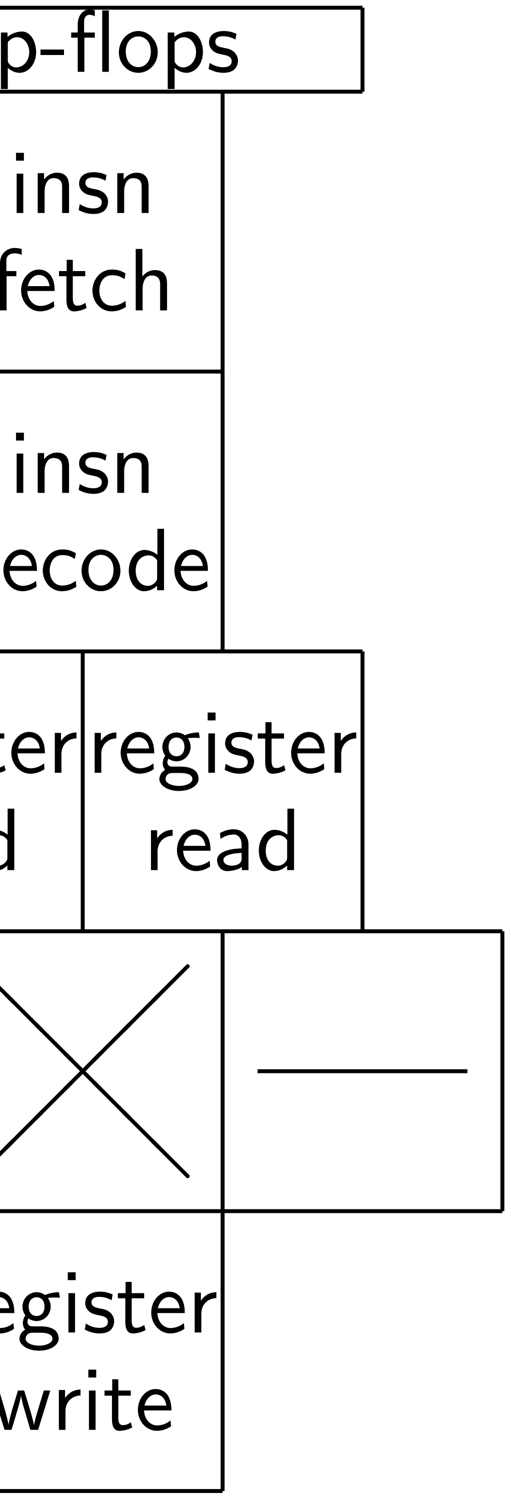


Further flexibility is useful but orthogonal to this talk.

"Pipelining" allows faster clock:

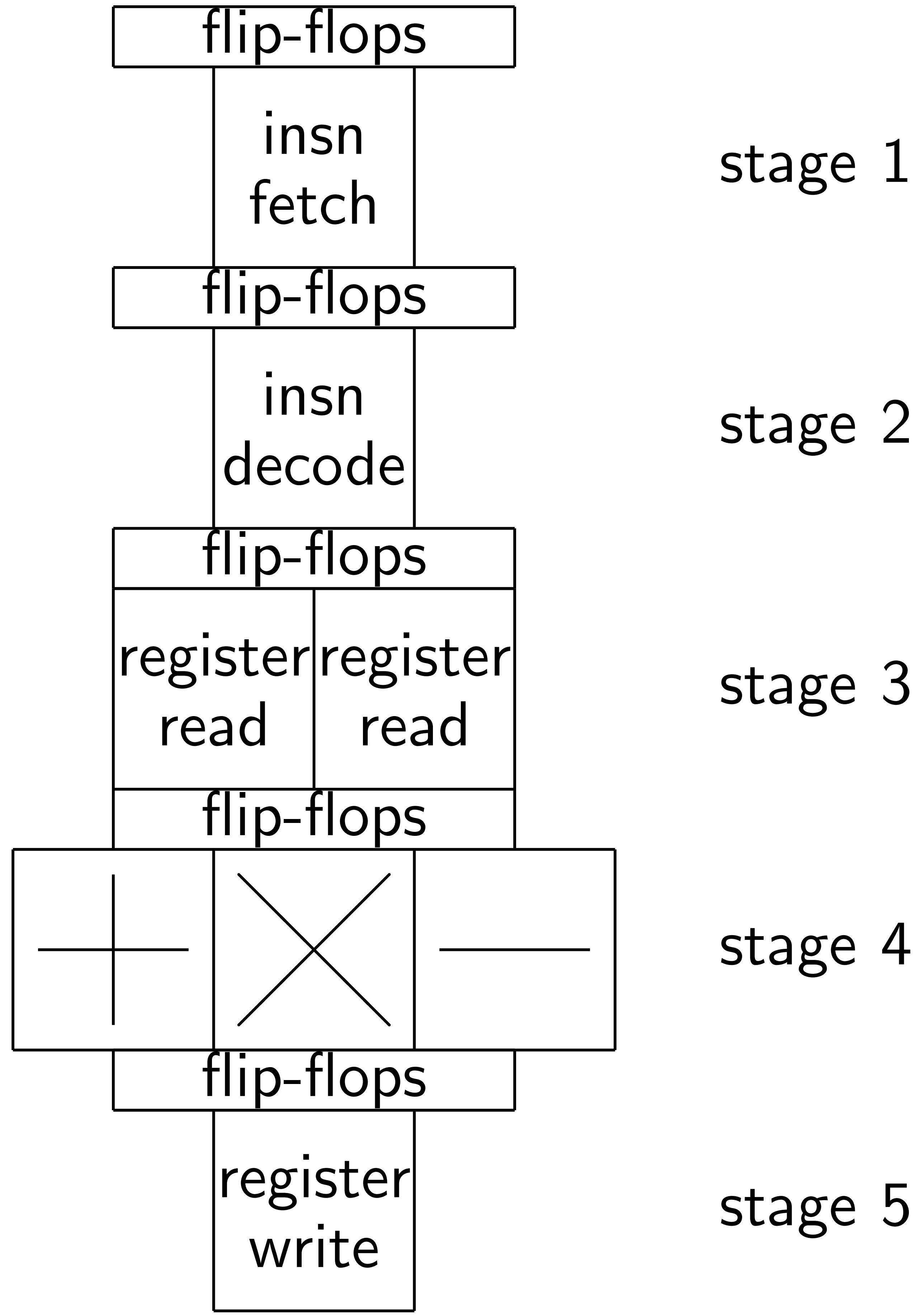


ve semi-flexible CPU:



flexibility is useful
ogonal to this talk.

“Pipelining” allows faster clock:



Goal: St
one tick

Instructi
reads ne
feeds p'

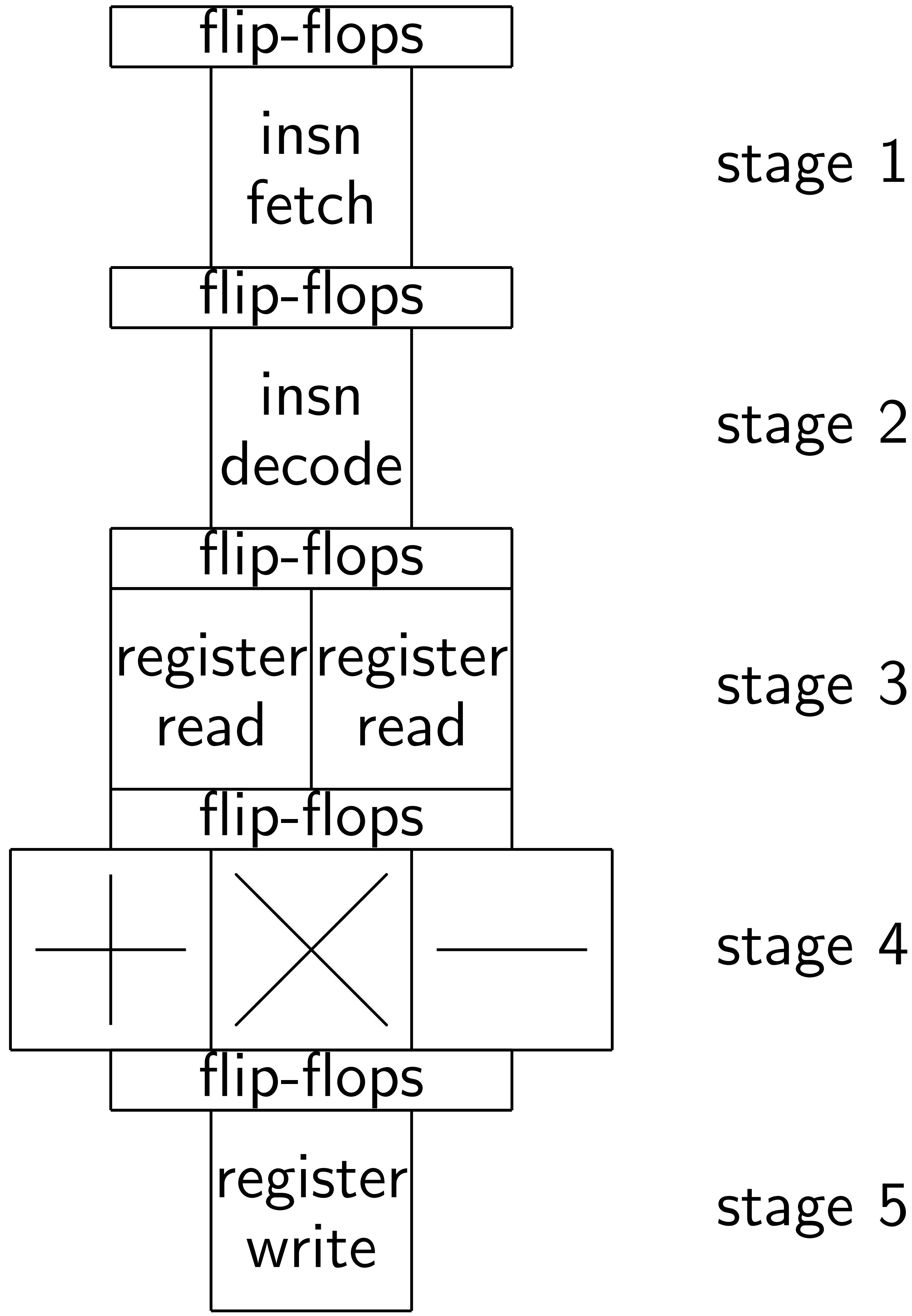
After ne
instructi
uncompr
while ins
reads an

Some ex
Also ext
preserve
e.g., stal

xible CPU:

is useful
this talk.

"Pipelining" allows faster clock:



Goal: Stage n han
one tick after stag

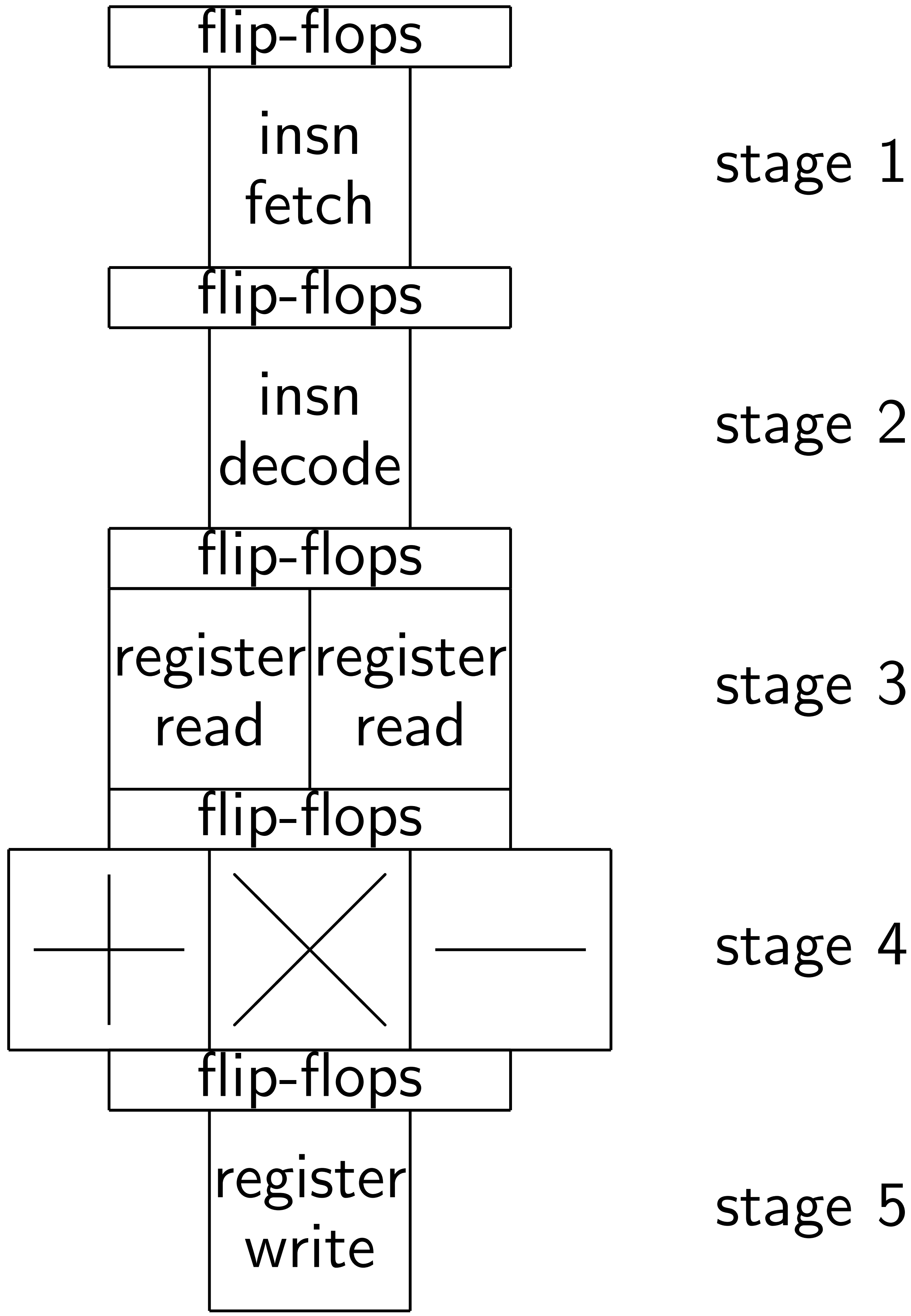
Instruction fetch
reads next instruct
feeds p' back, sen

After next clock ti
instruction decode
uncompresses this
while instruction f
reads another instr

Some extra flip-flo
Also extra area to
preserve instructio
e.g., stall on read-

:

“Pipelining” allows faster clock:



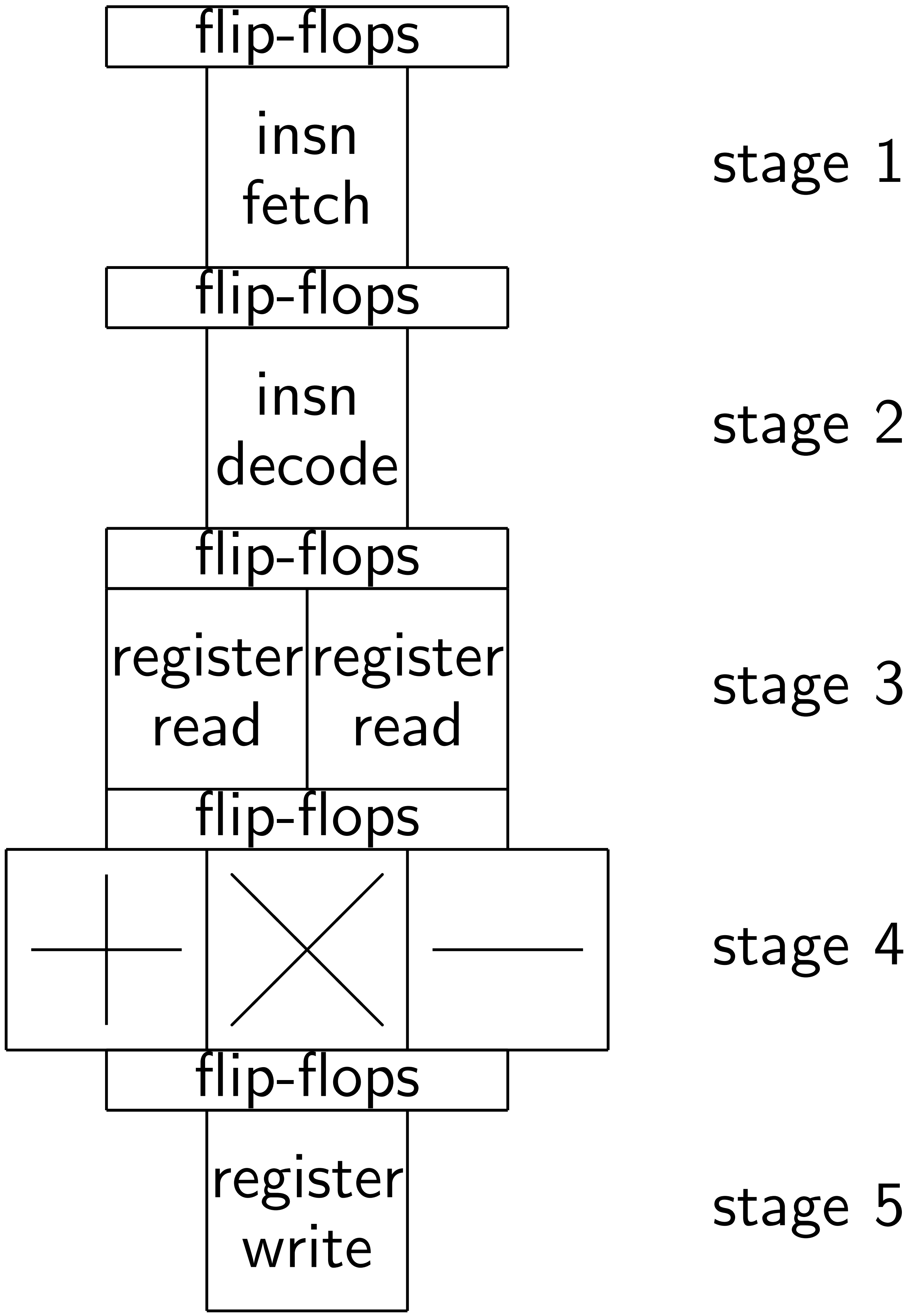
Goal: Stage n handles instru
one tick after stage $n-1$.

Instruction fetch
reads next instruction,
feeds p' back, sends instruct

After next clock tick,
instruction decode
uncompresses this instructio
while instruction fetch
reads another instruction.

Some extra flip-flop area.
Also extra area to
preserve instruction semanti
e.g., stall on read-after-write

"Pipelining" allows faster clock:



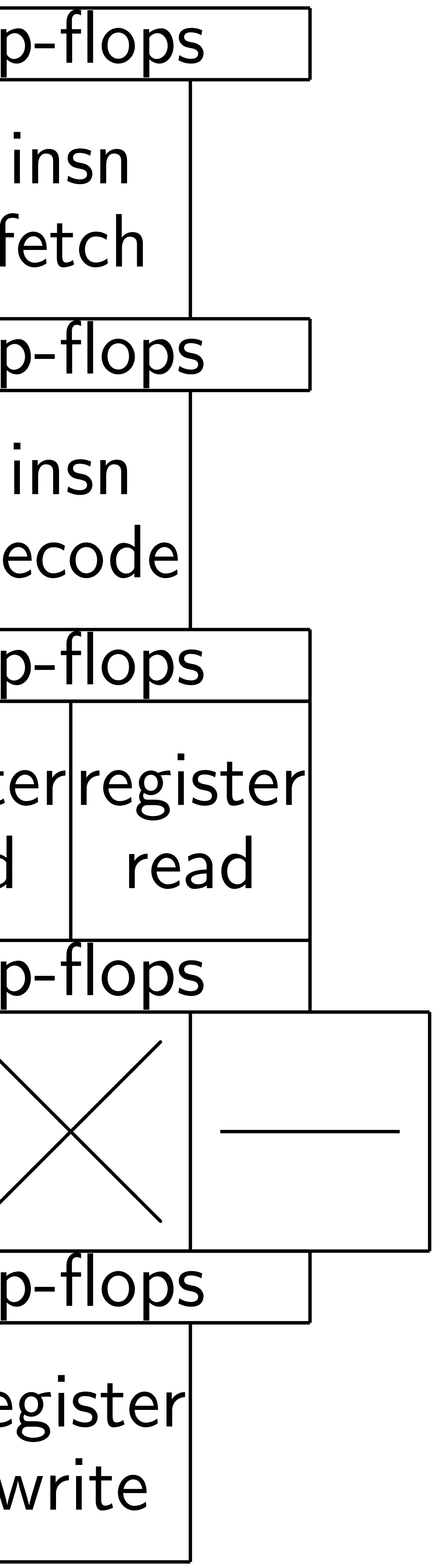
Goal: Stage n handles instruction one tick after stage $n-1$.

Instruction fetch
reads next instruction,
feeds p' back, sends instruction.

After next clock tick,
instruction decode
uncompresses this instruction,
while instruction fetch
reads another instruction.

Some extra flip-flop area.
Also extra area to
preserve instruction semantics:
e.g., stall on read-after-write.

ing" allows faster clock:



stage 1

stage 2

stage 3

stage 4

stage 5

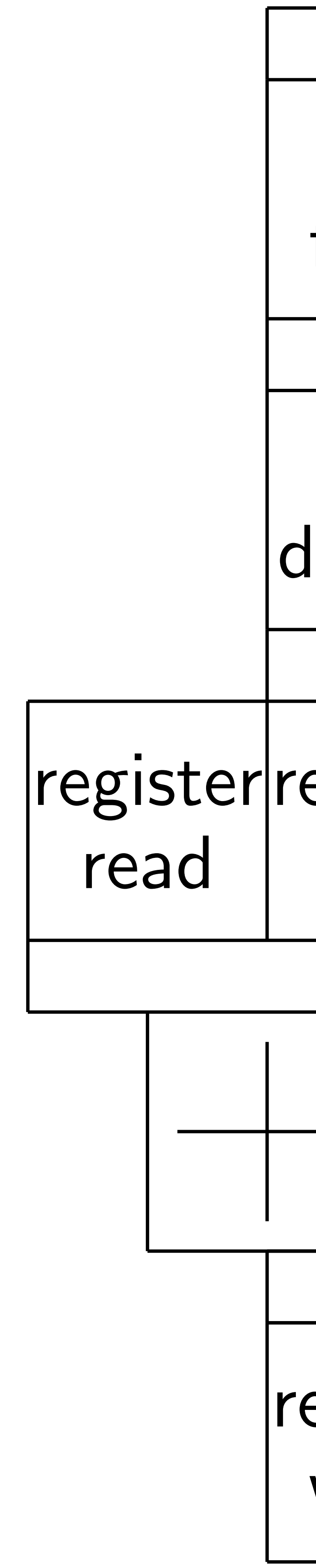
Goal: Stage n handles instruction one tick after stage $n-1$.

Instruction fetch
reads next instruction,
feeds p' back, sends instruction.

After next clock tick,
instruction decode
uncompresses this instruction,
while instruction fetch
reads another instruction.

Some extra flip-flop area.
Also extra area to
preserve instruction semantics:
e.g., stall on read-after-write.

"Supers

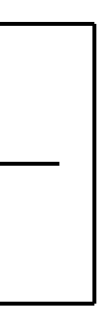


s faster clock:

stage 1

stage 2

stage 3



stage 4

stage 5

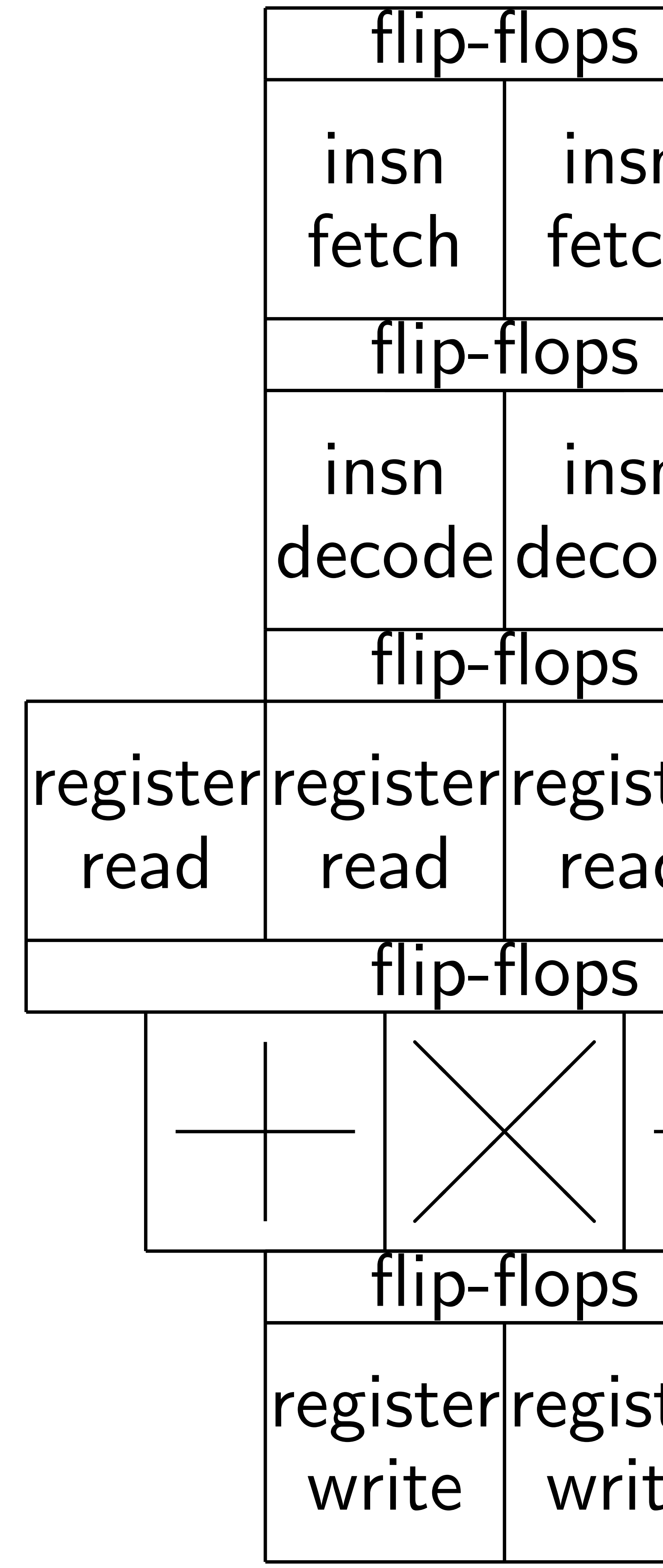
Goal: Stage n handles instruction one tick after stage $n-1$.

Instruction fetch
reads next instruction,
feeds p' back, sends instruction.

After next clock tick,
instruction decode
uncompresses this instruction,
while instruction fetch
reads another instruction.

Some extra flip-flop area.
Also extra area to
preserve instruction semantics:
e.g., stall on read-after-write.

"Superscalar" proc



ock:

e 1

e 2

e 3

e 4

e 5

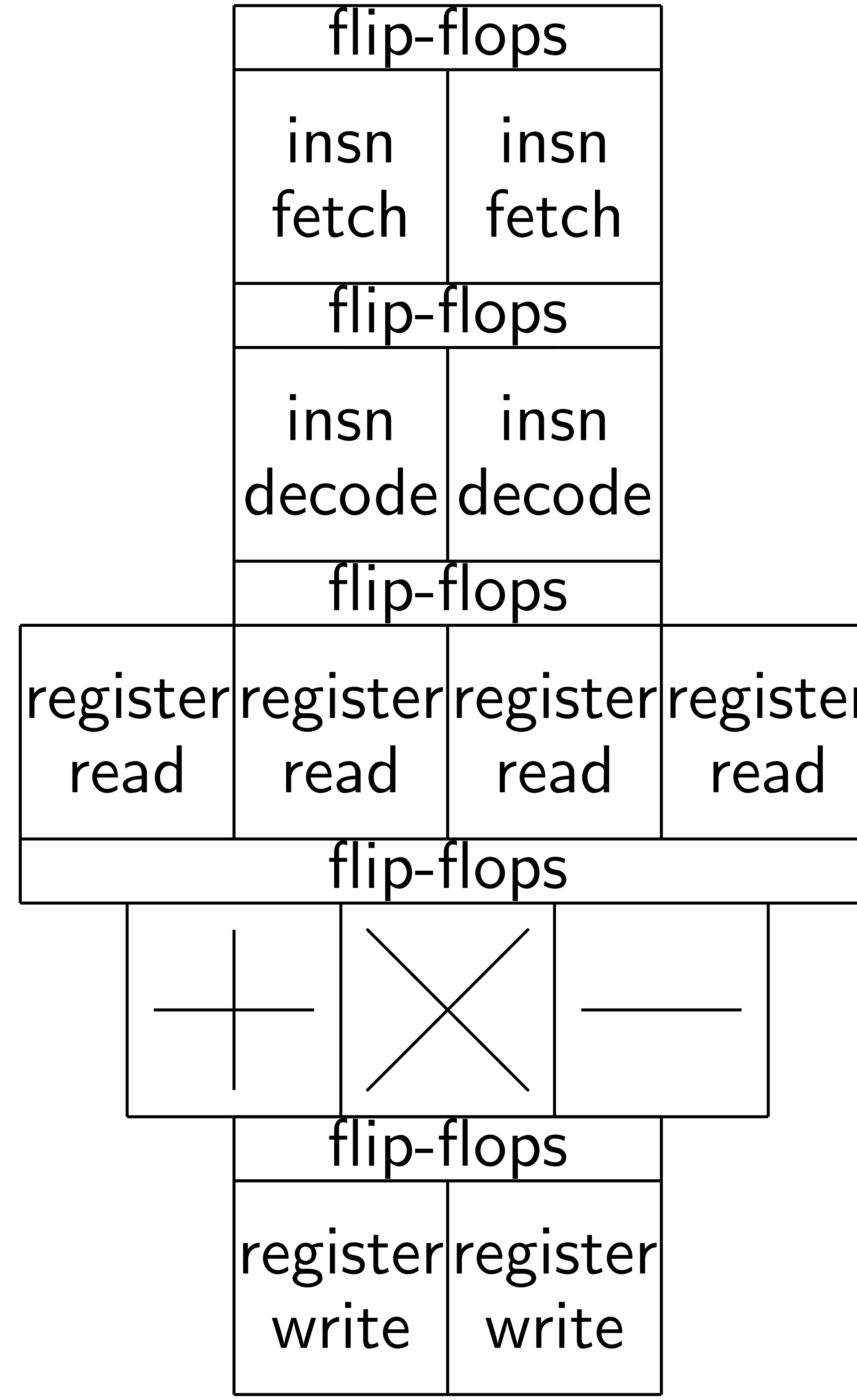
Goal: Stage n handles instruction one tick after stage $n-1$.

Instruction fetch
reads next instruction,
feeds p' back, sends instruction.

After next clock tick,
instruction decode
uncompresses this instruction,
while instruction fetch
reads another instruction.

Some extra flip-flop area.
Also extra area to
preserve instruction semantics:
e.g., stall on read-after-write.

"Superscalar" processing:



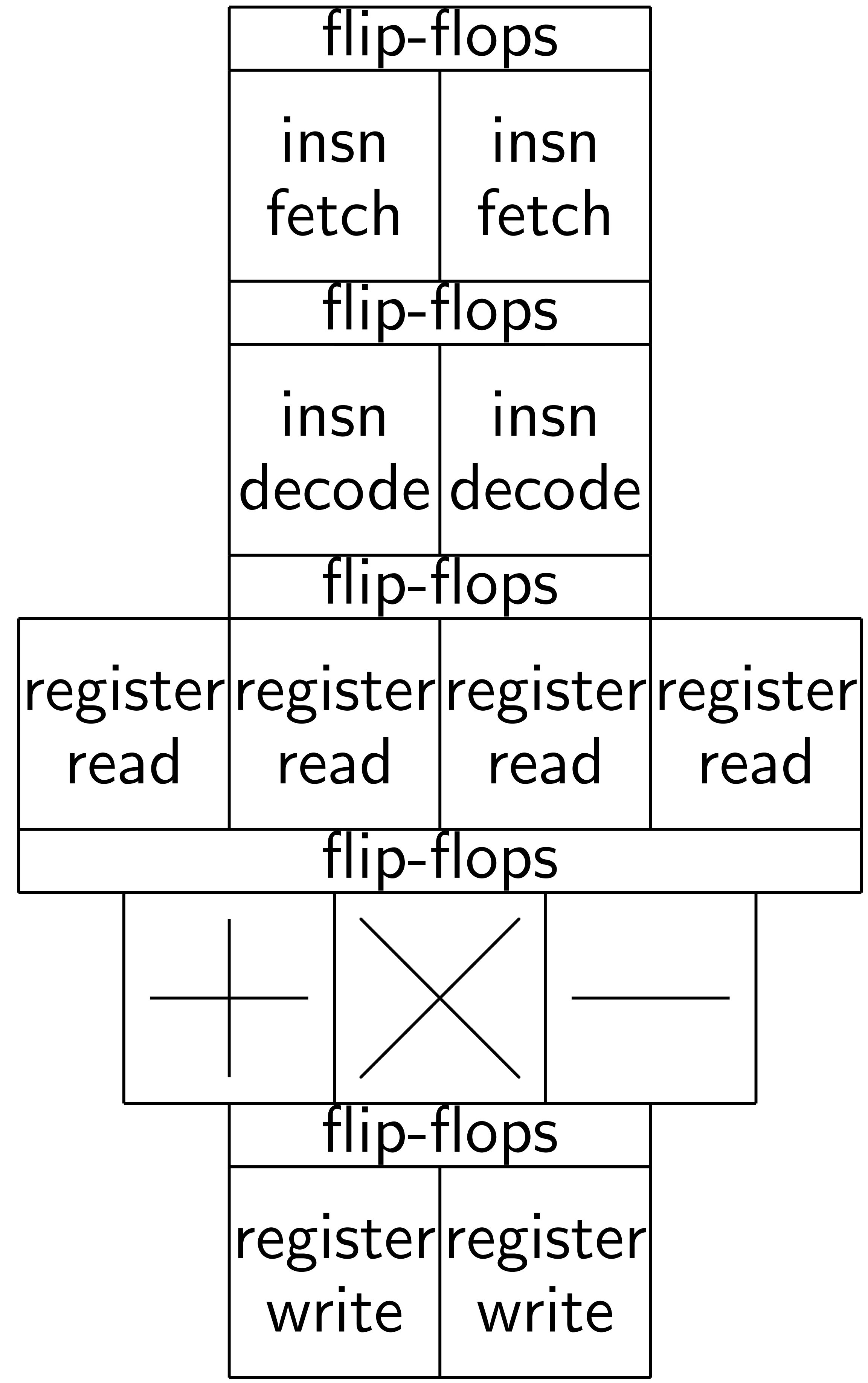
Goal: Stage n handles instruction one tick after stage $n-1$.

Instruction fetch
reads next instruction,
feeds p' back, sends instruction.

After next clock tick,
instruction decode
uncompresses this instruction,
while instruction fetch
reads another instruction.

Some extra flip-flop area.
Also extra area to
preserve instruction semantics:
e.g., stall on read-after-write.

"Superscalar" processing:



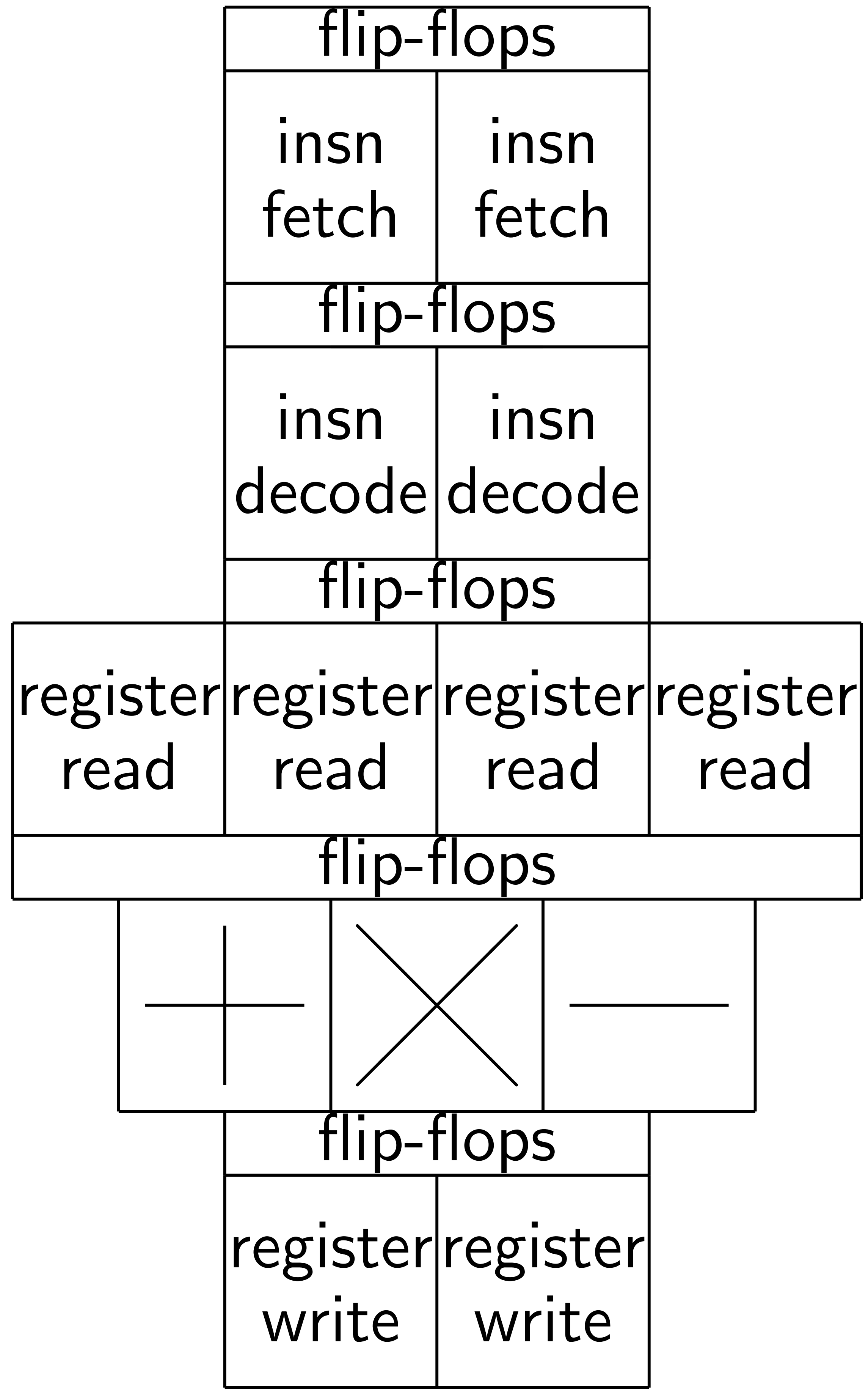
tage n handles instruction
after stage $n-1$.

on fetch
xt instruction,
back, sends instruction.

xt clock tick,
on decode
resses this instruction,
struction fetch
other instruction.

tra flip-flop area.
ra area to
instruction semantics:
ll on read-after-write.

“Superscalar” processing:



“Vector”

Expand
into n-ve
ARM “N
Intel “AV
Intel “AV
GPUs ha

ndles instruction
ge $n-1$.

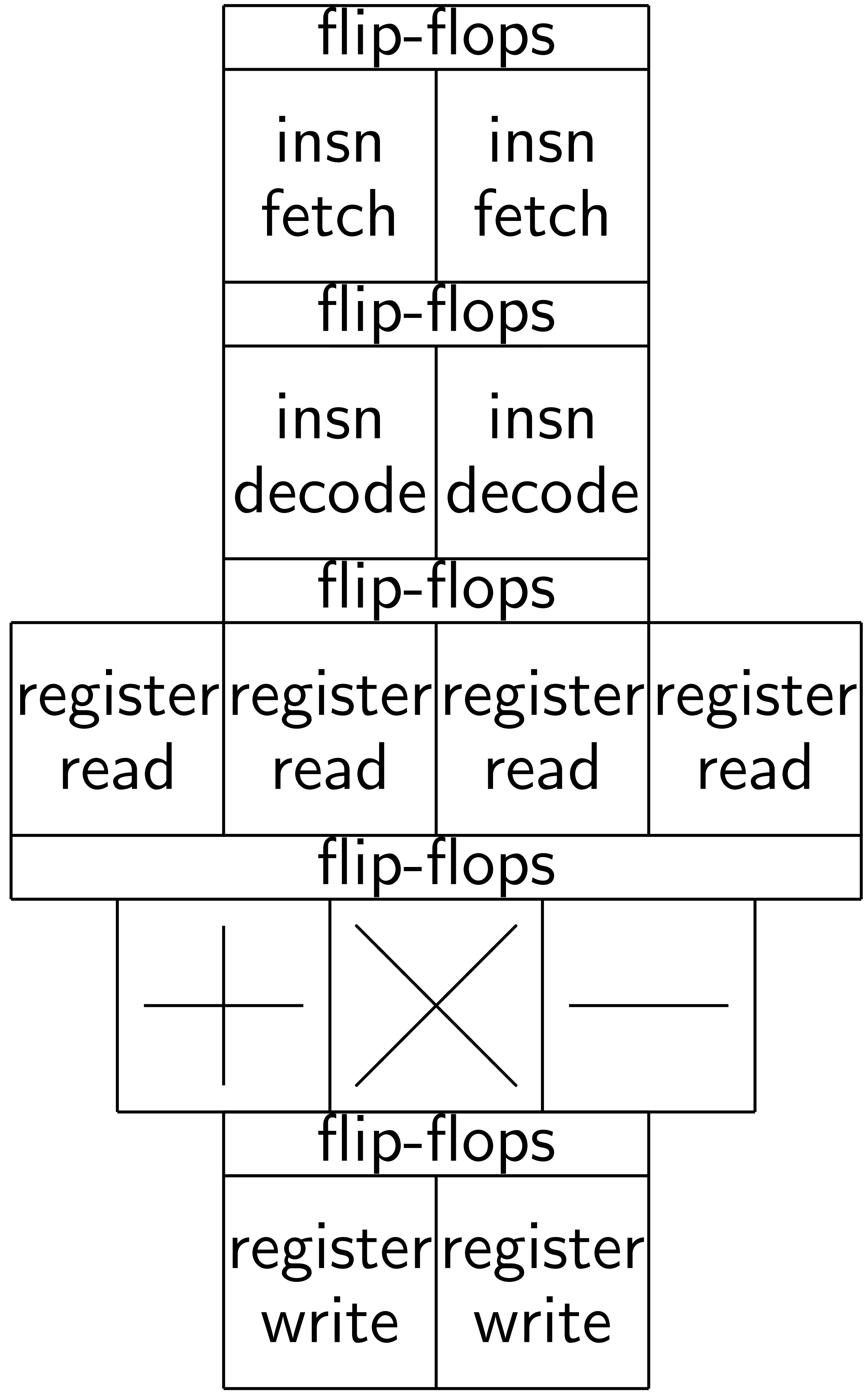
tion,
ds instruction.

ck,
e
instruction,
etch
ruction.

p area.

n semantics:
after-write.

“Superscalar” processing:



“Vector” processin

Expand each 32-bi
into n-vector of 32
ARM “NEON” ha
Intel “AVX2” has
Intel “AVX-512” h
GPUs have larger

uction

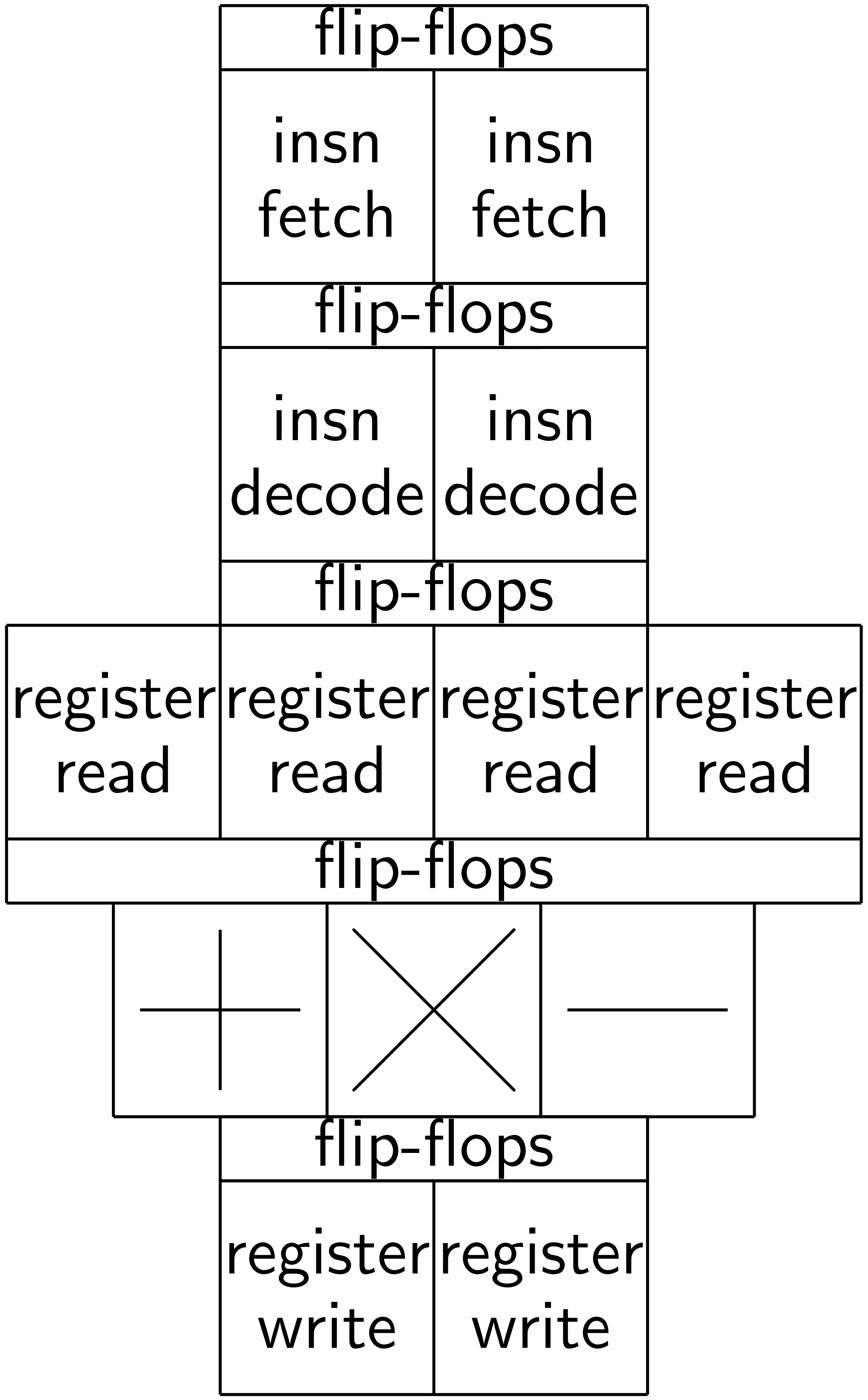
tion.

n,

cs:

e.

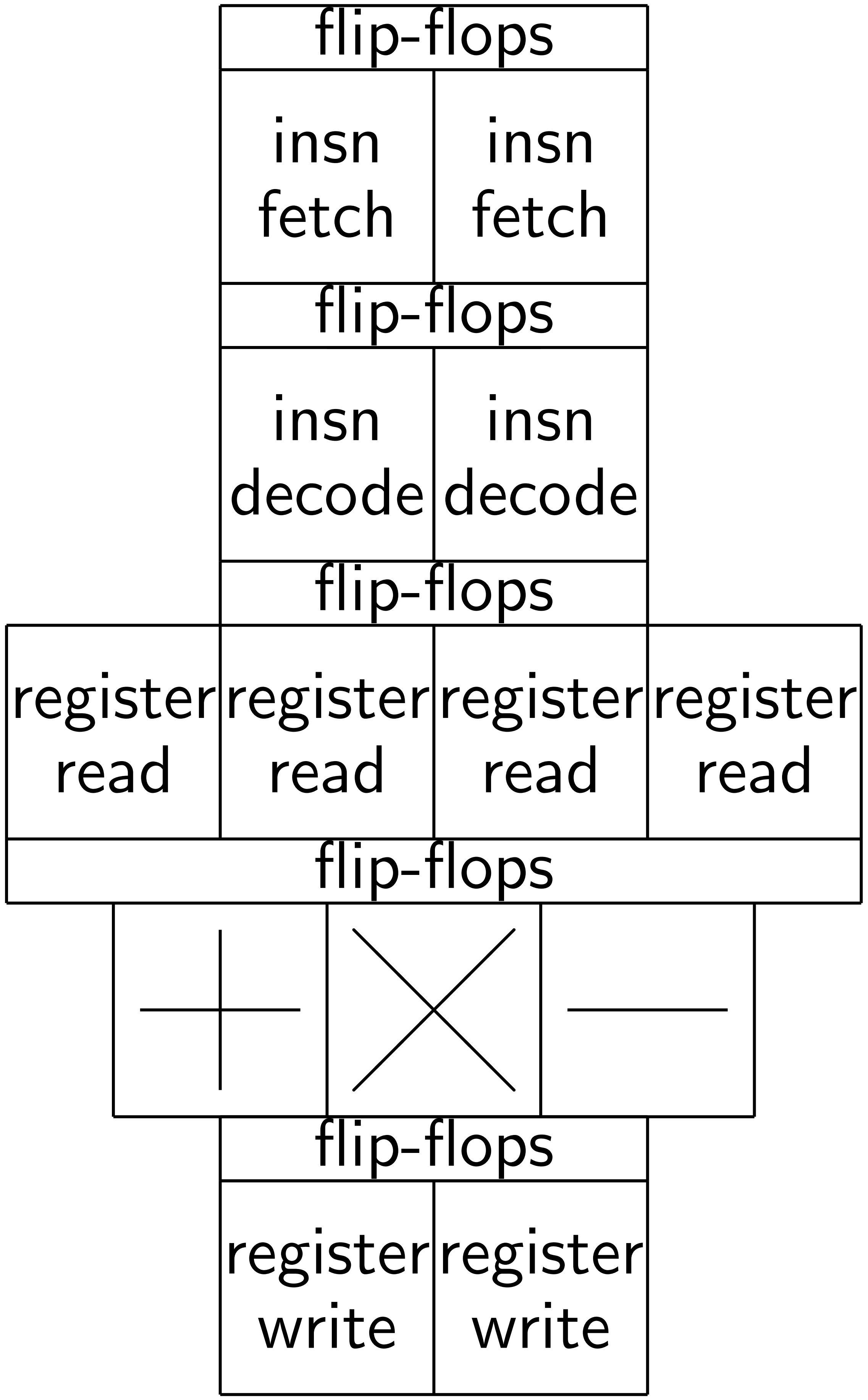
“Superscalar” processing:



“Vector” processing:

Expand each 32-bit integer
into n-vector of 32-bit integ
ARM “NEON” has $n = 4$;
Intel “AVX2” has $n = 8$;
Intel “AVX-512” has $n = 16$
GPUs have larger n.

“Superscalar” processing:



“Vector” processing:

Expand each 32-bit integer into n-vector of 32-bit integers.

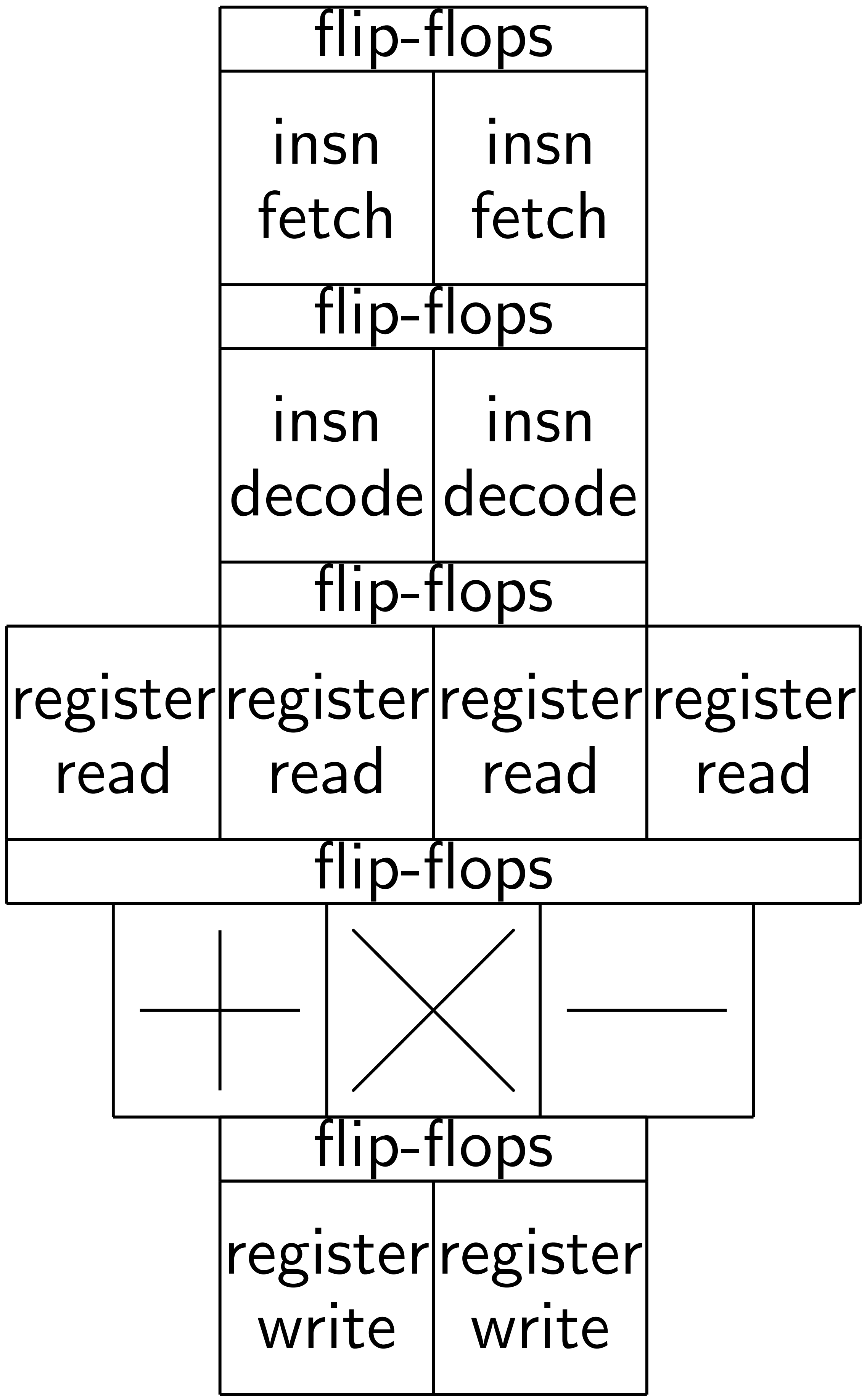
ARM “NEON” has $n = 4$;

Intel “AVX2” has $n = 8$;

Intel “AVX-512” has $n = 16$;

GPUs have larger n.

“Superscalar” processing:

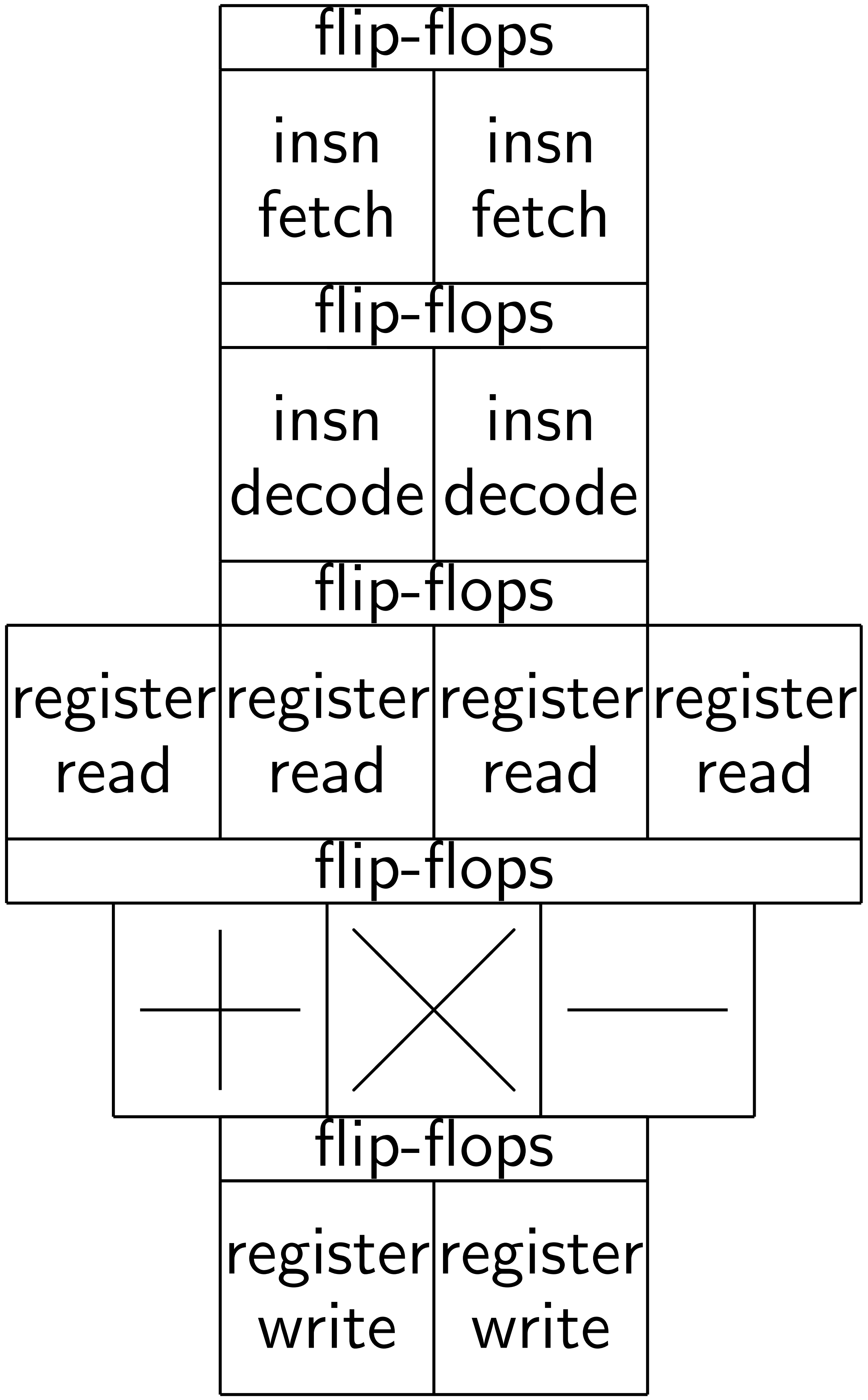


“Vector” processing:

Expand each 32-bit integer into n-vector of 32-bit integers. ARM “NEON” has $n = 4$; Intel “AVX2” has $n = 8$; Intel “AVX-512” has $n = 16$; GPUs have larger n.

$n\times$ speedup if $n\times$ arithmetic circuits, $n\times$ read/write circuits. Benefit: Amortizes insn circuits.

“Superscalar” processing:



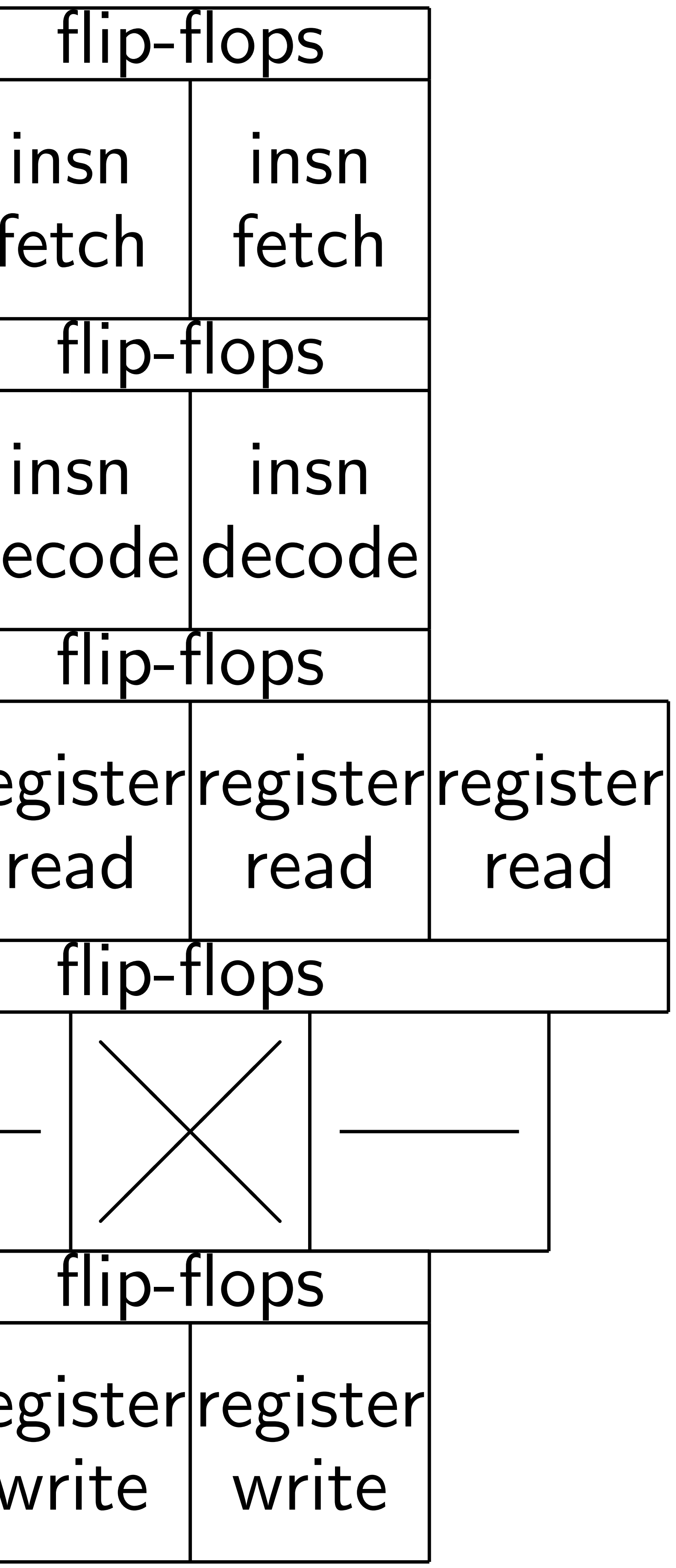
“Vector” processing:

Expand each 32-bit integer
into n-vector of 32-bit integers.
ARM “NEON” has $n = 4$;
Intel “AVX2” has $n = 8$;
Intel “AVX-512” has $n = 16$;
GPUs have larger n.

$n\times$ speedup if
$n\times$ arithmetic circuits,
$n\times$ read/write circuits.
Benefit: Amortizes insn circuits.

Huge effect on higher-level
algorithms and data structures.

calar” processing:



“Vector” processing:

Expand each 32-bit integer into *n*-vector of 32-bit integers.
ARM “NEON” has $n = 4$;
Intel “AVX2” has $n = 8$;
Intel “AVX-512” has $n = 16$;
GPUs have larger n.

$n\times$ speedup if
$n\times$ arithmetic circuits,
$n\times$ read/write circuits.
Benefit: Amortizes insn circuits.

Huge effect on higher-level algorithms and data structures.

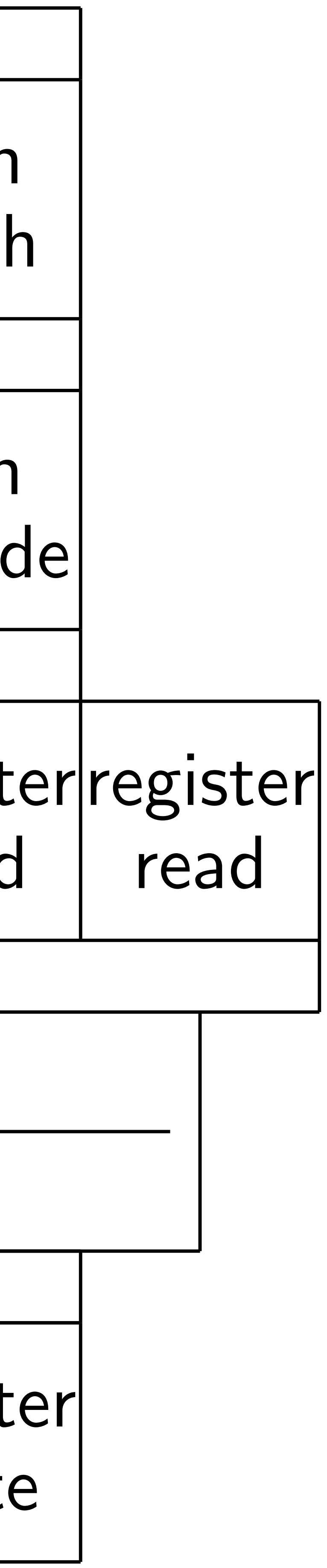
<u>Network</u>

How exp

Input: a
Each nu
represen

Output:
in increa
represen
same mu

cessing:



"Vector" processing:

Expand each 32-bit integer
into n-vector of 32-bit integers.
ARM "NEON" has $n = 4$;
Intel "AVX2" has $n = 8$;
Intel "AVX-512" has $n = 16$;
GPUs have larger n.

$n\times$ speedup if
$n\times$ arithmetic circuits,
$n\times$ read/write circuits.
Benefit: Amortizes insn circuits.

Huge effect on higher-level
algorithms and data structures.

Network on chip:

How expensive is s

Input: array of n n
Each number in $\{$
represented in bina

Output: array of n
in increasing order
represented in bina
same multiset as i

“Vector” processing:

Expand each 32-bit integer into n-vector of 32-bit integers.
ARM “NEON” has $n = 4$;
Intel “AVX2” has $n = 8$;
Intel “AVX-512” has $n = 16$;
GPUs have larger n.

$n\times$ speedup if
$n\times$ arithmetic circuits,
$n\times$ read/write circuits.
Benefit: Amortizes insn circuits.

Huge effect on higher-level algorithms and data structures.

Network on chip: the mesh

How expensive is sorting?

Input: array of n numbers.
Each number in $\{1, 2, \ldots, n$
represented in binary.

Output: array of n numbers
in increasing order,
represented in binary;
same multiset as input.

"Vector" processing:

Expand each 32-bit integer
into n-vector of 32-bit integers.
ARM "NEON" has $n = 4$;
Intel "AVX2" has $n = 8$;
Intel "AVX-512" has $n = 16$;
GPUs have larger n.

$n\times$ speedup if
$n\times$ arithmetic circuits,
$n\times$ read/write circuits.
Benefit: Amortizes insn circuits.

Huge effect on higher-level
algorithms and data structures.

Network on chip: the mesh

How expensive is sorting?

Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

Output: array of n numbers,
in increasing order,
represented in binary;
same multiset as input.

"Vector" processing:

Expand each 32-bit integer into n-vector of 32-bit integers.
ARM "NEON" has $n = 4$;
Intel "AVX2" has $n = 8$;
Intel "AVX-512" has $n = 16$;
GPUs have larger n.

$n\times$ speedup if
$n\times$ arithmetic circuits,
$n\times$ read/write circuits.
Benefit: Amortizes insn circuits.

Huge effect on higher-level algorithms and data structures.

Network on chip: the mesh

How expensive is sorting?

Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$, represented in binary.

Output: array of n numbers, in increasing order, represented in binary; same multiset as input.

Metric: seconds used by circuit of area $n^{1+o(1)}$.

For simplicity assume $n = 4^k$.

" processing:

each 32-bit integer
ector of 32-bit integers.
NEON" has $n = 4$;
VX2" has $n = 8$;
VX-512" has $n = 16$;
ave larger n.

dup if
metic circuits,
/write circuits.
Amortizes insn circuits.

fect on higher-level
ms and data structures.

Network on chip: the mesh

How expensive is sorting?

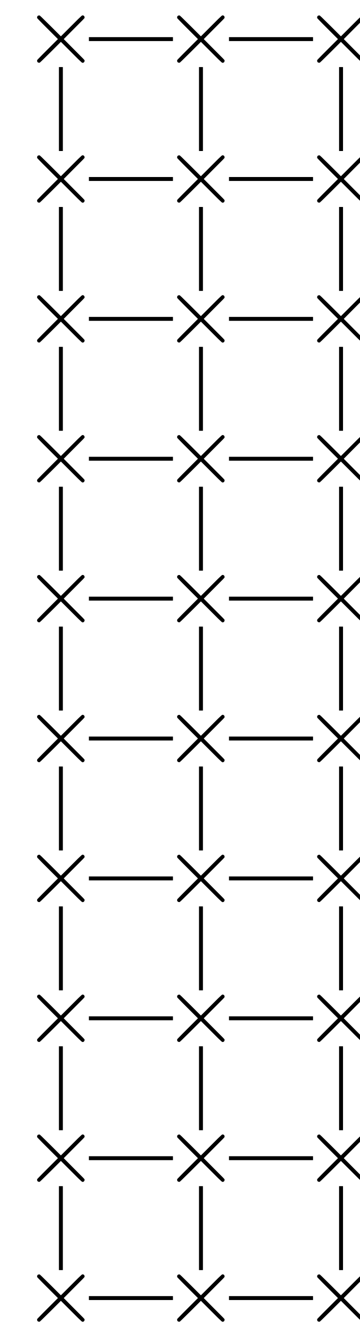
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

Output: array of n numbers,
in increasing order,
represented in binary;
same multiset as input.

Metric: seconds used by
circuit of area $n^{1+o(1)}$.

For simplicity assume $n = 4^k$.

Spread a
square n
each of
with nea



ng:

it integer
2-bit integers.
s $n = 4$;
$n = 8$;
has $n = 16$;
n.

cuits,
cuits.
s insn circuits.

gher-level
ta structures.

Network on chip: the mesh

How expensive is sorting?

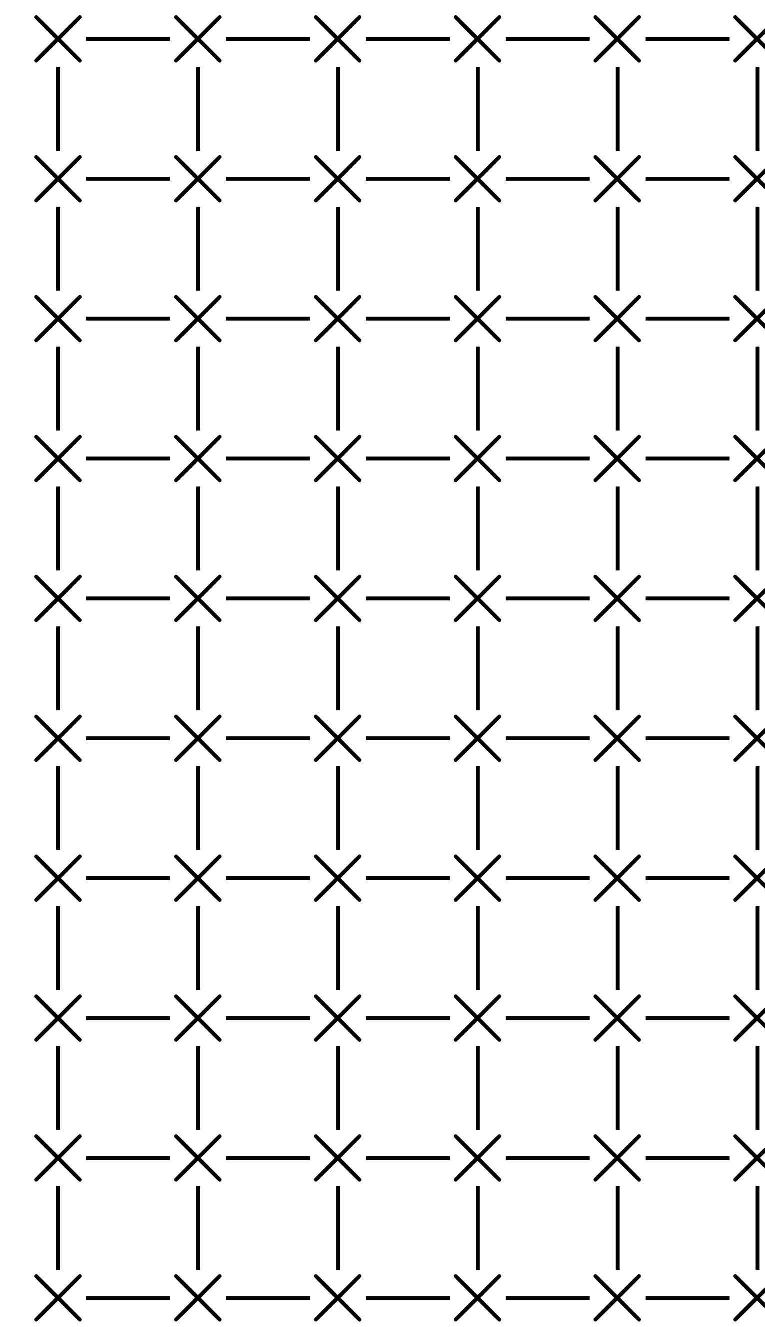
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

Output: array of n numbers,
in increasing order,
represented in binary;
same multiset as input.

Metric: seconds used by
circuit of area $n^{1+o(1)}$.

For simplicity assume $n = 4^k$.

Spread array acros
square mesh of n
each of area $n^{o(1)}$
with near-neighbor



ers.

;

uits.

res.

Network on chip: the mesh

How expensive is sorting?

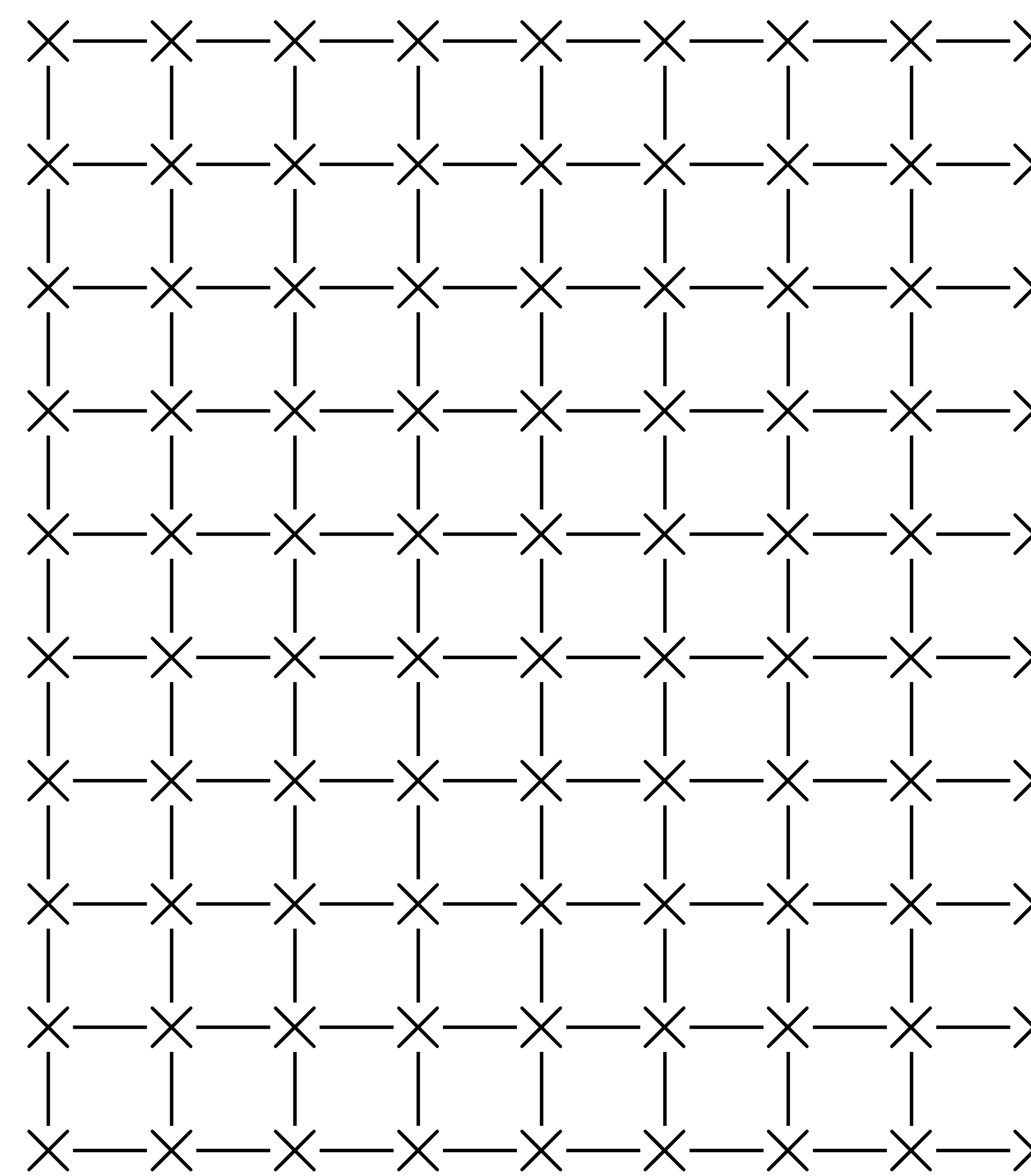
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

Output: array of n numbers,
in increasing order,
represented in binary;
same multiset as input.

Metric: seconds used by
circuit of area $n^{1+o(1)}$.

For simplicity assume $n = 4^k$.

Spread array across
square mesh of n small cells
each of area $n^{o(1)}$,
with near-neighbor wiring:



Network on chip: the mesh

How expensive is sorting?

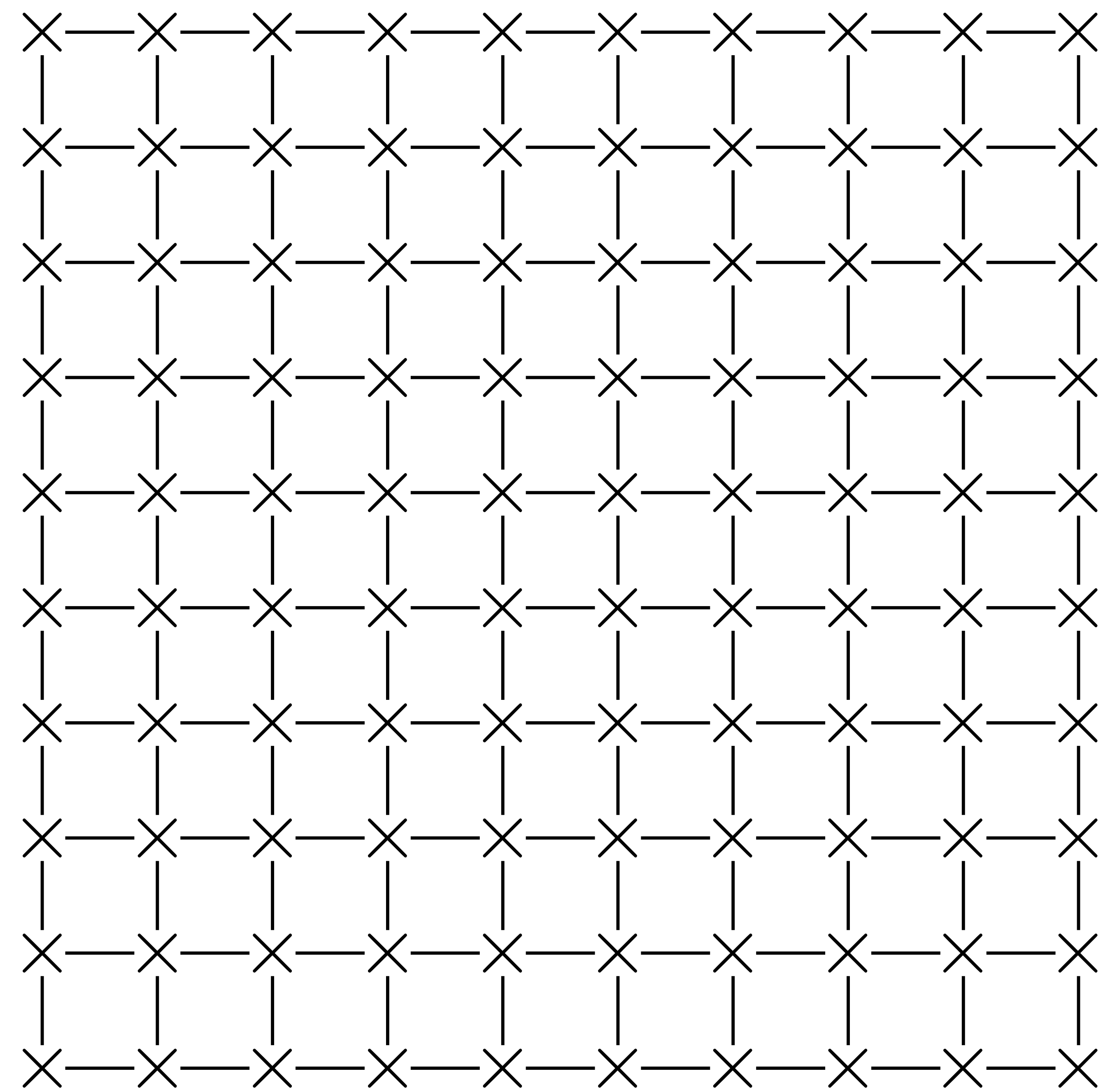
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

Output: array of n numbers,
in increasing order,
represented in binary;
same multiset as input.

Metric: seconds used by
circuit of area $n^{1+o(1)}$.

For simplicity assume $n = 4^k$.

Spread array across
square mesh of n small cells,
each of area $n^{o(1)}$,
with near-neighbor wiring:



on chip: the mesh

pensive is sorting?

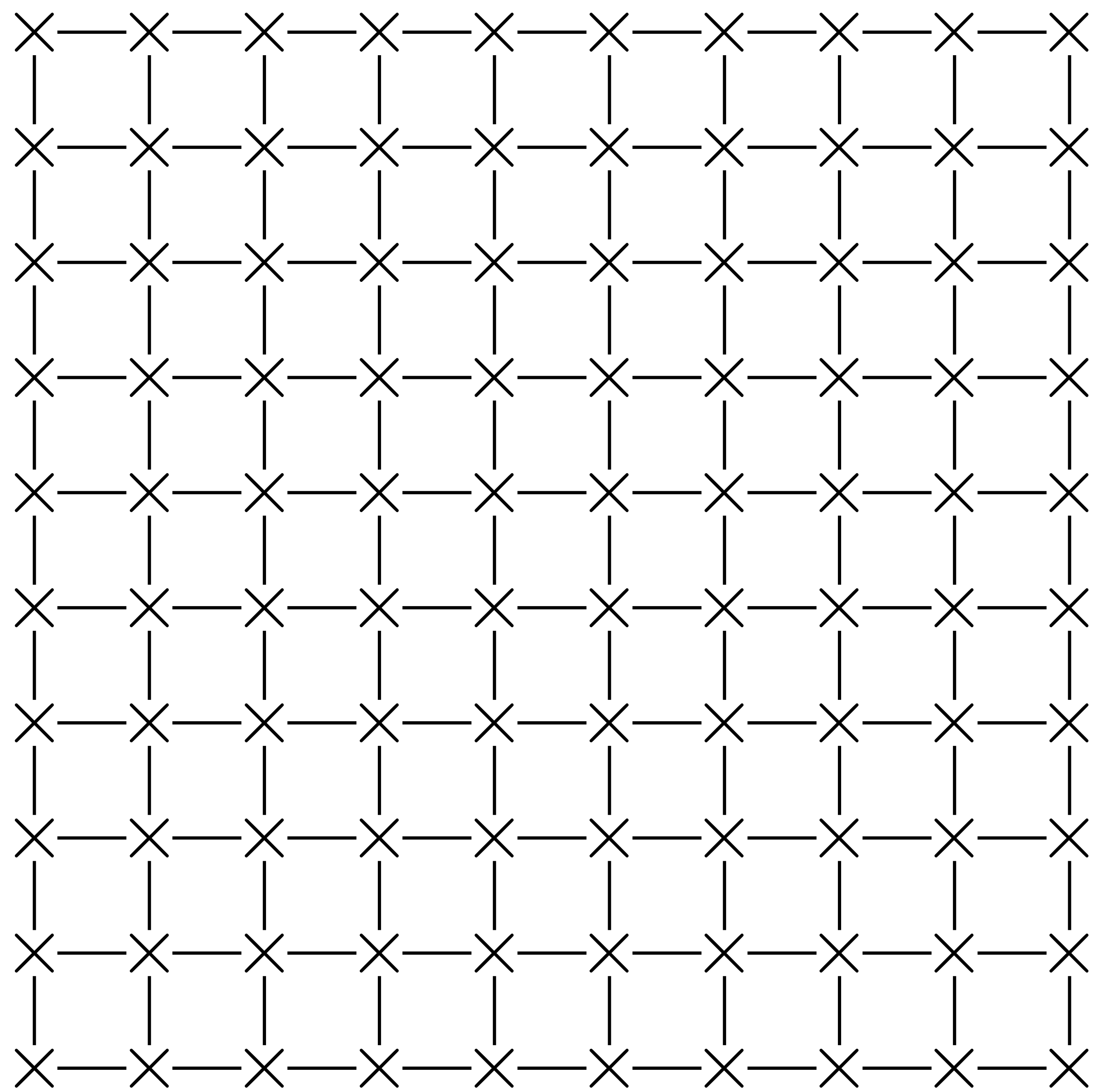
rray of n numbers.
mber in $\{1, 2, \ldots, n^2\}$,
ted in binary.

array of n numbers,
nsing order,
ted in binary;
ultiset as input.

seconds used by
f area $n^{1+o(1)}$.

plicity assume $n = 4^k$.

Spread array across
square mesh of n small cells,
each of area $n^{o(1)}$,
with near-neighbor wiring:



Sort row
in $n^{0.5+o}$

- Sort e
  3 1 4
  1 3 1

- Sort a
  1 3 1
  1 1 3

- Repeat
  equals

the mesh

sorting?

numbers.

$1, 2, \ldots, n^2\}$,

ary.

n numbers,

,

ary;

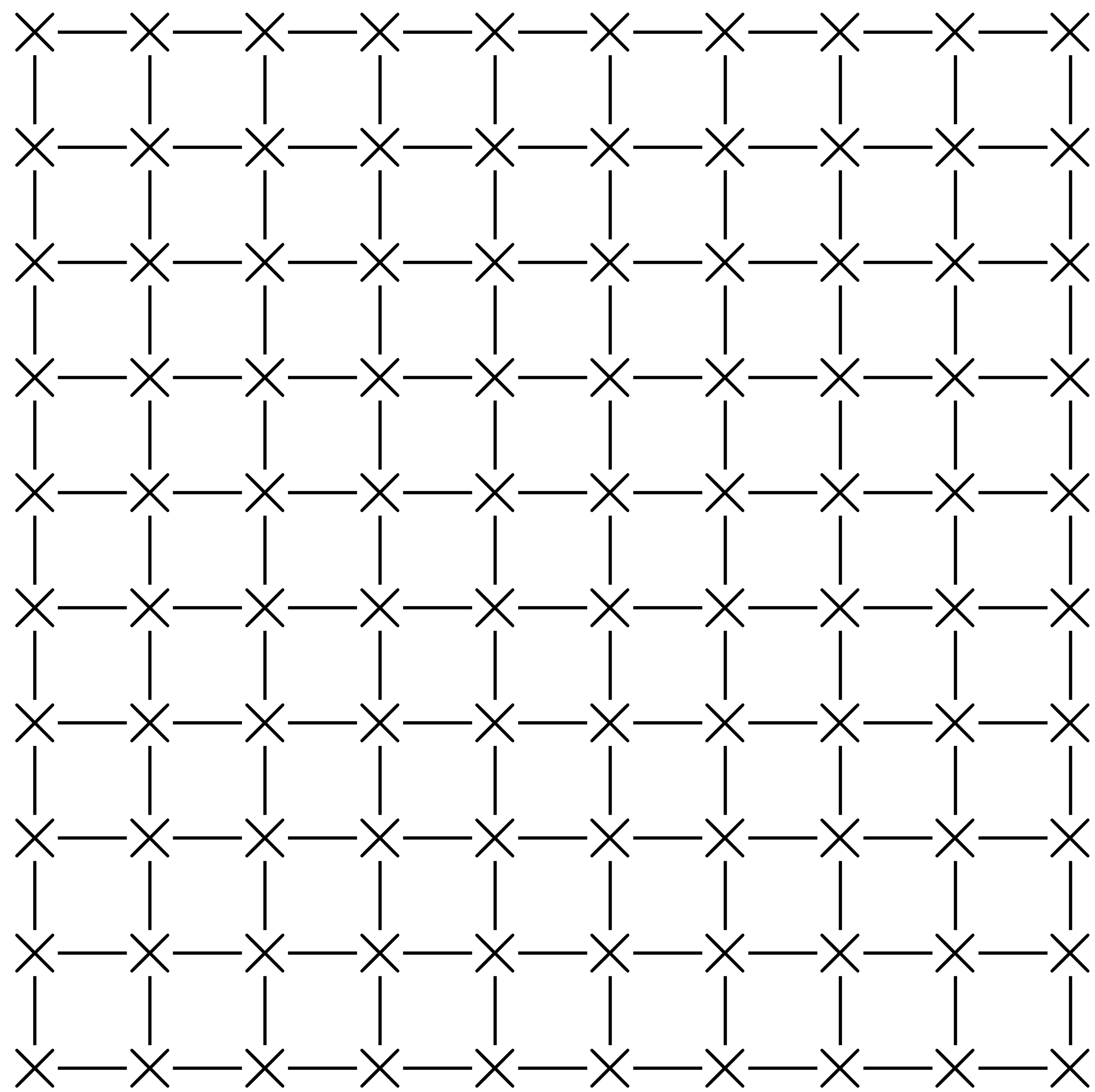
nput.

sed by

$o(1)$.

ume $n = 4^k$.

Spread array across
square mesh of n small cells,
each of area $n^{o(1)}$,
with near-neighbor wiring:



Sort row of $n^{0.5}$ ce
in $n^{0.5+o(1)}$ secon

- Sort each pair in

  3 1 4 1 5 9 2 6

  1 3 1 4 5 9 2 6

- Sort alternate pa

  1 3 1 4 5 9 2 6

  1 1 3 4 5 2 9 6

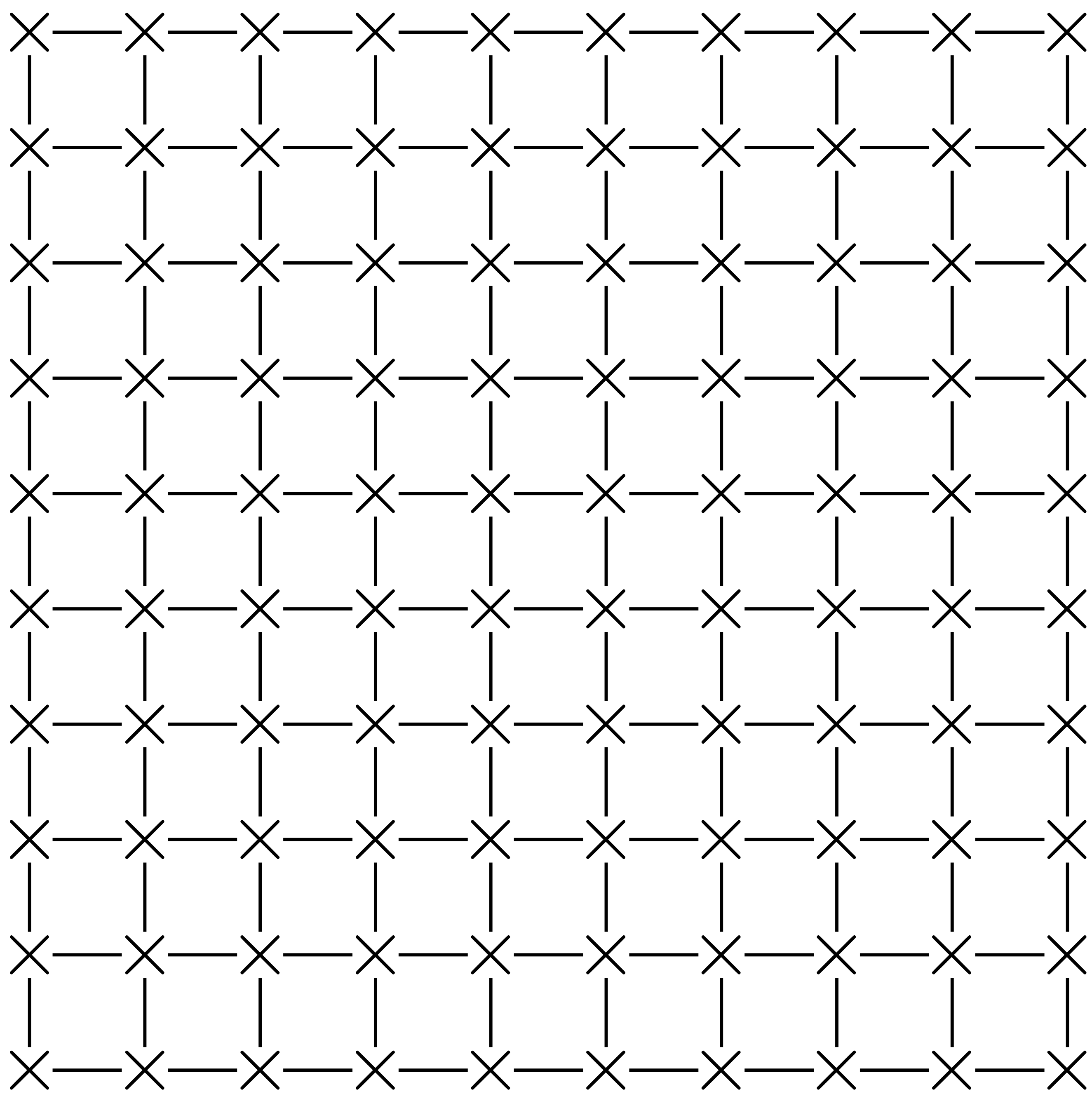
- Repeat until nu
  equals row lengt

$^2\}$,

,

k.

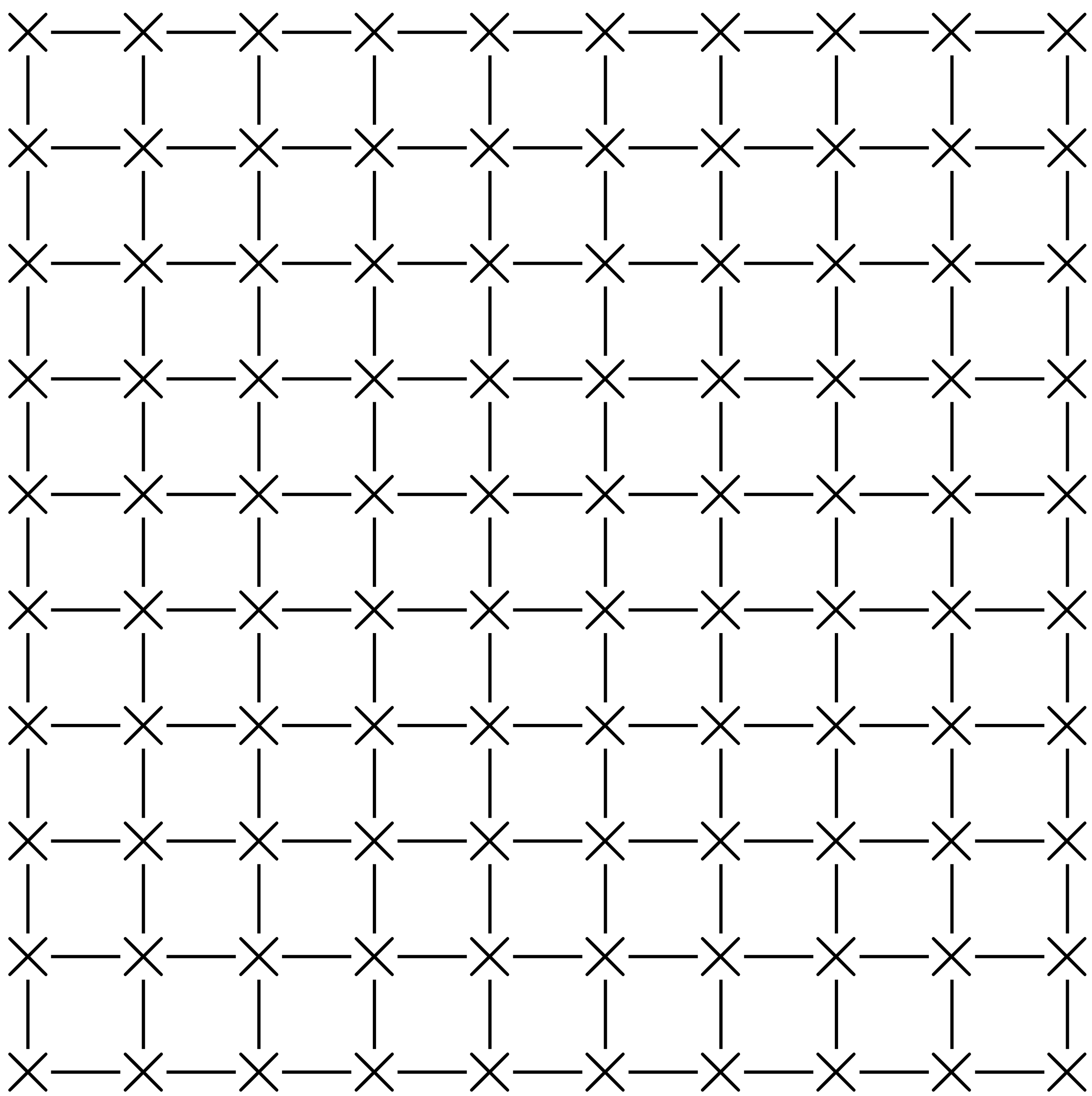
Spread array across square mesh of n small cells, each of area $n^{o(1)}$, with near-neighbor wiring:



Sort row of $n^{0.5}$ cells in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  $1\ 3\ 1\ 4\ 5\ 9\ 2\ 6$
- Sort alternate pairs in para
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  $1\ 1\ 3\ 4\ 5\ 2\ 9\ 6$
- Repeat until number of ste
  equals row length.

Spread array across
square mesh of n small cells,
each of area $n^{o(1)}$,
with near-neighbor wiring:



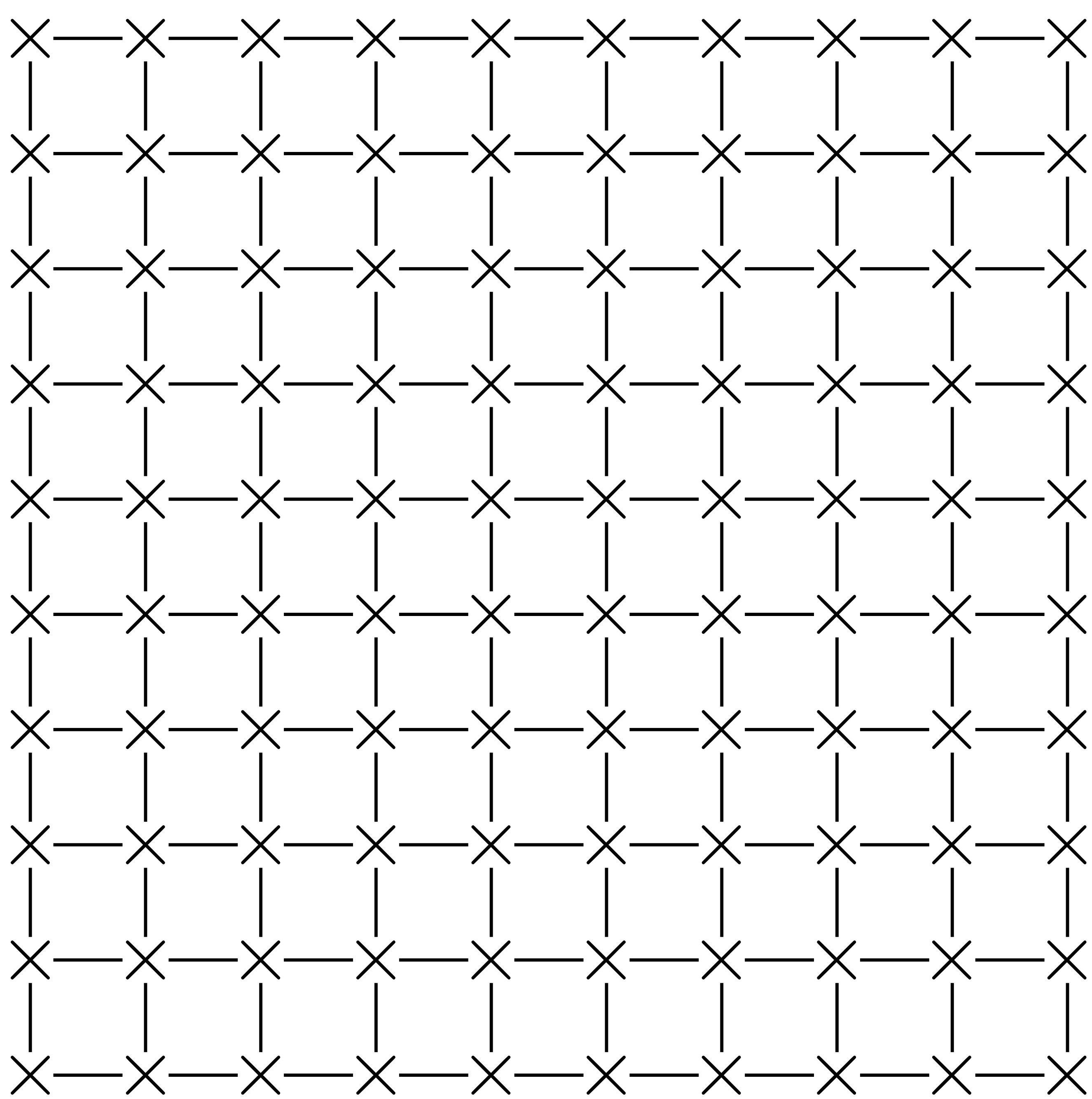
Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  $1\ 3\ 1\ 4\ 5\ 9\ 2\ 6$

- Sort alternate pairs in parallel.
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  $1\ 1\ 3\ 4\ 5\ 2\ 9\ 6$

- Repeat until number of steps
  equals row length.

Spread array across
square mesh of n small cells,
each of area $n^{o(1)}$,
with near-neighbor wiring:

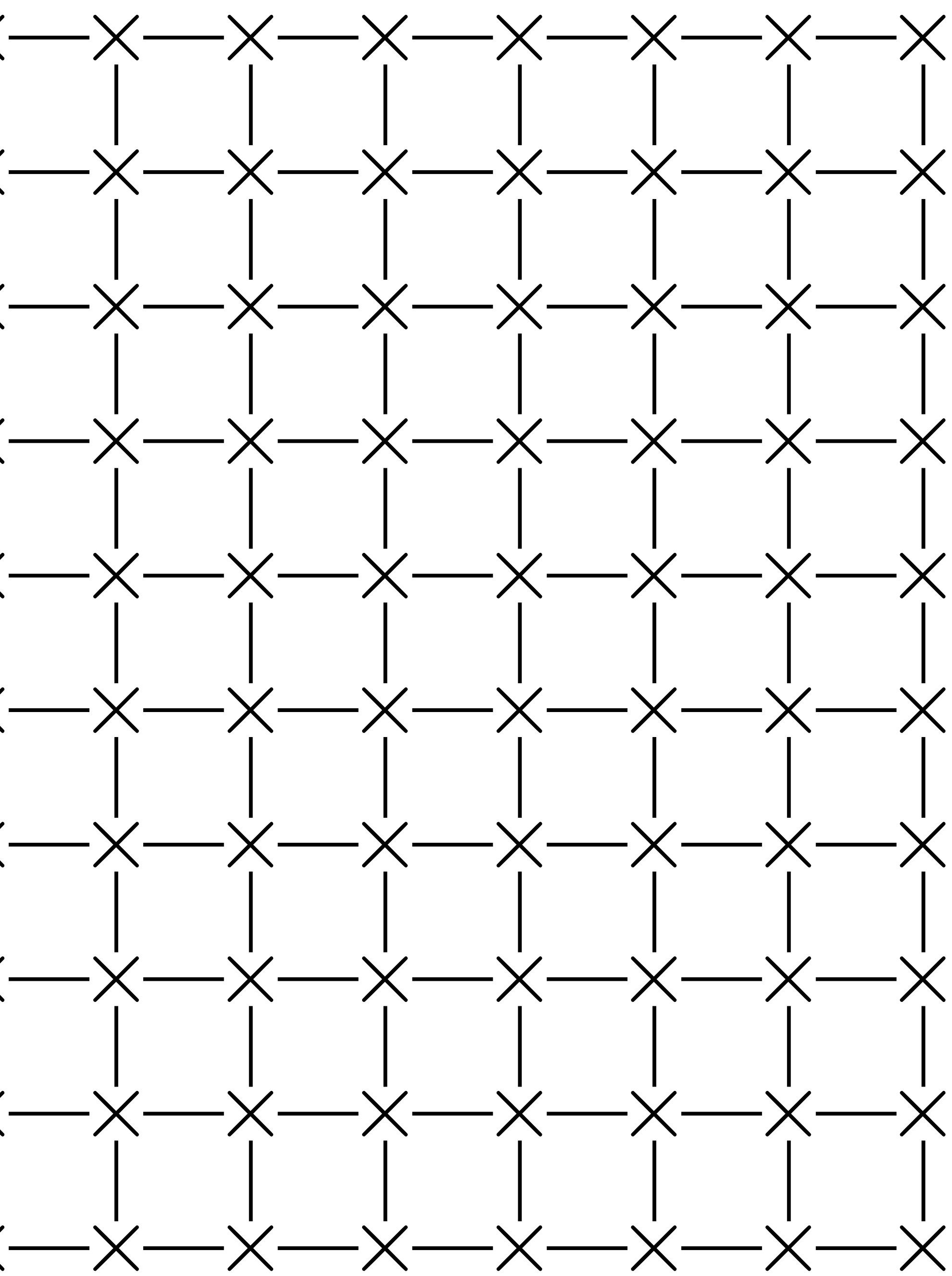


Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  $1\ 3\ 1\ 4\ 5\ 9\ 2\ 6$
- Sort alternate pairs in parallel.
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  $1\ 1\ 3\ 4\ 5\ 2\ 9\ 6$
- Repeat until number of steps equals row length.

Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

array across
nesh of n small cells,
area $n^{o(1)}$,
ar-neighbor wiring:



Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  $1\ 3\ 1\ 4\ 5\ 9\ 2\ 6$

- Sort alternate pairs in parallel.
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  $1\ 1\ 3\ 4\ 5\ 2\ 9\ 6$

- Repeat until number of steps equals row length.

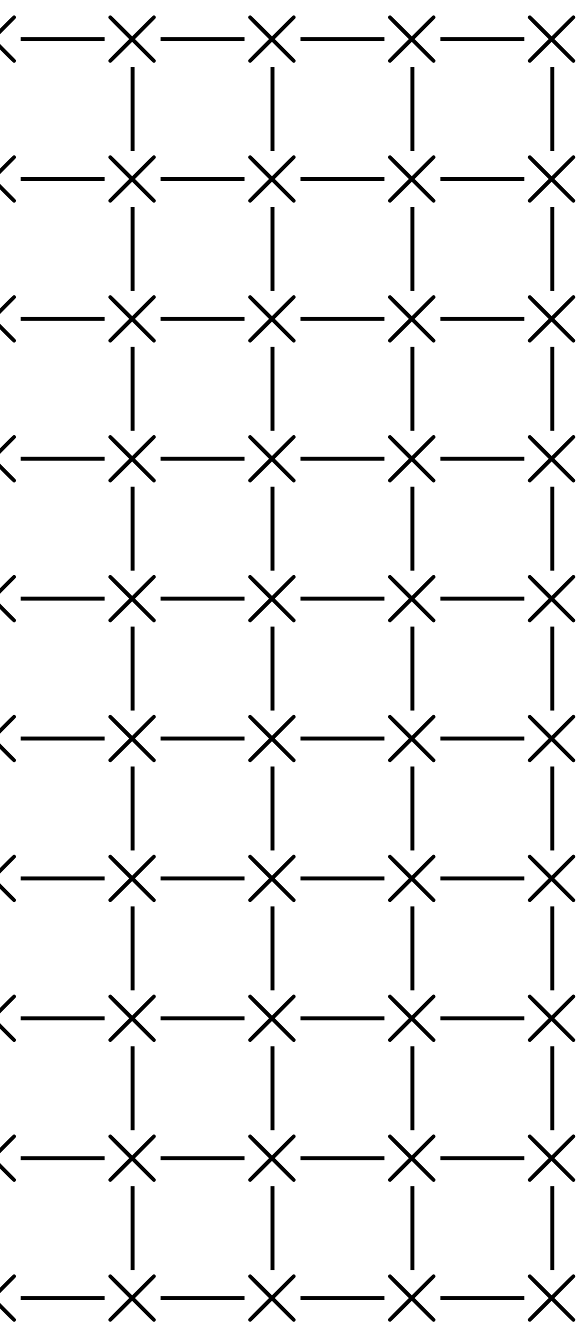
Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

Sort all
in $n^{0.5+o}$

- Recurs
  in para
- Sort ea
- Sort ea
- Sort ea
- Sort ea

With pro
left-to-ri
for each
that this

ss
small cells,
’
r wiring:



Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  1 3 1 4 5 9 2 6
- Sort alternate pairs in parallel.
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  1 1 3 4 5 2 9 6
- Repeat until number of steps equals row length.

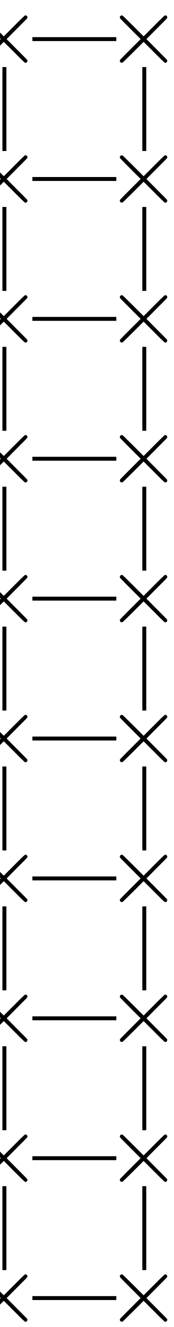
Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

Sort all n cells
in $n^{0.5+o(1)}$ secon

- Recursively sort
  in parallel, if $n >$
- Sort each colum
- Sort each row in
- Sort each colum
- Sort each row in

With proper choic
left-to-right/right-
for each row, can
that this sorts who

,



Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  $1\ 3\ 1\ 4\ 5\ 9\ 2\ 6$
- Sort alternate pairs in parallel.
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  $1\ 1\ 3\ 4\ 5\ 2\ 9\ 6$
- Repeat until number of steps equals row length.

Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

Sort all n cells
in $n^{0.5+o(1)}$ seconds:

- Recursively sort quadrants in parallel, if $n > 1$.
- Sort each column in parall
- Sort each row in parallel.
- Sort each column in parall
- Sort each row in parallel.

With proper choice of
left-to-right/right-to-left
for each row, can prove
that this sorts whole array.

Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- Sort each pair in parallel.
  $\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$
  1 3 1 4 5 9 2 6
- Sort alternate pairs in parallel.
  $1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$
  1 1 3 4 5 2 9 6
- Repeat until number of steps equals row length.

Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

Sort all n cells
in $n^{0.5+o(1)}$ seconds:

- Recursively sort quadrants in parallel, if $n > 1$.
- Sort each column in parallel.
- Sort each row in parallel.
- Sort each column in parallel.
- Sort each row in parallel.

With proper choice of
left-to-right/right-to-left
for each row, can prove
that this sorts whole array.

of $n^{0.5}$ cells
$^{o(1)}$ seconds:

ach pair in parallel.

$\underline{1}$ $\underline{5\ 9}$ $\underline{2\ 6}$ $\mapsto$

4 5 9 2 6

lternate pairs in parallel.

$\underline{4\ 5}$ $\underline{9\ 2}$ 6 $\mapsto$

4 5 2 9 6

t until number of steps
row length.

h row, in parallel,
al of $n^{0.5+o(1)}$ seconds.

Sort all n cells
in $n^{0.5+o(1)}$ seconds:

- Recursively sort quadrants in parallel, if $n > 1$.
- Sort each column in parallel.
- Sort each row in parallel.
- Sort each column in parallel.
- Sort each row in parallel.

With proper choice of
left-to-right/right-to-left
for each row, can prove
that this sorts whole array.

For exar
this $8 \times$

3	1	4
5	3	5
2	3	8
3	3	8
0	2	8
1	6	9
5	1	0
7	4	9

ells
ds:

n parallel.

$\mapsto$

airs in parallel.

$\mapsto$

mber of steps
ch.

parallel,
$o(1)$ seconds.

Sort all n cells
in $n^{0.5+o(1)}$ seconds:

- Recursively sort quadrants in parallel, if $n > 1$.
- Sort each column in parallel.
- Sort each row in parallel.
- Sort each column in parallel.
- Sort each row in parallel.

With proper choice of
left-to-right/right-to-left
for each row, can prove
that this sorts whole array.

For example, assu
this 8×8 array is

3	1	4	1	5	9
5	3	5	8	9	7
2	3	8	4	6	2
3	3	8	3	2	7
0	2	8	8	4	1
1	6	9	3	9	9
5	1	0	5	8	2
7	4	9	4	4	5

allel.

eps

ds.

Sort all n cells
in $n^{0.5+o(1)}$ seconds:

- Recursively sort quadrants in parallel, if $n > 1$.
- Sort each column in parallel.
- Sort each row in parallel.
- Sort each column in parallel.
- Sort each row in parallel.

With proper choice of
left-to-right/right-to-left
for each row, can prove
that this sorts whole array.

For example, assume that
this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Sort all n cells
in $n^{0.5+o(1)}$ seconds:

- Recursively sort quadrants in parallel, if $n > 1$.
- Sort each column in parallel.
- Sort each row in parallel.
- Sort each column in parallel.
- Sort each row in parallel.

With proper choice of left-to-right/right-to-left for each row, can prove that this sorts whole array.

For example, assume that this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

n cells

$^{o(1)}$ seconds:

sively sort quadrants

allel, if $n > 1$.

ach column in parallel.

ach row in parallel.

ach column in parallel.

ach row in parallel.

oper choice of

ght/right-to-left

row, can prove

s sorts whole array.

For example, assume that this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Recursiv

top $\rightarrow$, l

1	1	2
3	3	3
3	4	4
5	8	8
1	1	0
4	4	3
7	6	5
9	9	8

ds:

quadrants

> 1.

n in parallel.

n parallel.

n in parallel.

n parallel.

e of

to-left

prove

ble array.

For example, assume that this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Recursively sort qu

top $\rightarrow$, bottom $\leftarrow$

1	1	2	3	2	2
3	3	3	3	4	5
3	4	4	5	6	6
5	8	8	8	9	9
1	1	0	0	2	2
4	4	3	2	5	4
7	6	5	5	9	8
9	9	8	8	9	9

el.

el.

For example, assume that this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Recursively sort quadrants, top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

For example, assume that this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Recursively sort quadrants, top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

mple, assume that
8 array is in cells:

1	5	9	2	6
8	9	7	9	3
4	6	2	6	4
3	2	7	9	5
8	4	1	9	7
3	9	9	3	7
5	8	2	0	9
4	4	5	9	2

Recursively sort quadrants,
top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

Sort eac
in parall

1	1	0
1	1	2
3	3	3
3	4	3
4	4	4
5	6	5
7	8	8
9	9	8

me that
in cells:

2	6
9	3
6	4
9	5
9	7
3	7
0	9
9	2

Recursively sort quadrants,
top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

Sort each column
in parallel:

1	1	0	0	2	2
1	1	2	2	2	2
3	3	3	3	4	4
3	4	3	3	5	5
4	4	4	5	6	6
5	6	5	5	9	8
7	8	8	8	9	9
9	9	8	8	9	9

Recursively sort quadrants,
top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

Sort each column
in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Recursively sort quadrants,
top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

Sort each column
in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

vely sort quadrants,

bottom ←:

2	3	2	2	2	3
3	3	4	5	5	6
4	5	6	6	7	7
8	8	9	9	9	9
0	0	2	2	1	0
3	2	5	4	4	3
5	5	9	8	7	7
8	8	9	9	9	9

Sort each column

in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Sort eac

alternate

0	0	0
3	2	2
3	3	3
6	5	5
4	4	4
9	8	7
7	8	8
9	9	9

uadrants,

:

2	3
5	6
7	7
9	9
1	0
4	3
7	7
9	9

Sort each column in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Sort each row in p
alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1
3	2	2	2	2	2
3	3	3	3	3	4
6	5	5	5	4	3
4	4	4	5	6	6
9	8	7	7	6	5
7	8	8	8	9	9
9	9	9	9	9	9

Sort each column in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Sort each row in parallel, alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

Sort each column in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Sort each row in parallel, alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

h column

el:

0	0	2	2	1	0
2	2	2	2	2	3
3	3	4	4	4	3
3	3	5	5	5	6
4	5	6	6	7	7
5	5	9	8	7	7
8	8	9	9	9	9
8	8	9	9	9	9

Sort each row in parallel,

alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

Sort eac

in parall

0	0	0
3	2	2
3	3	3
4	4	4
6	5	5
7	8	7
9	8	8
9	9	9

1	0
2	3
4	3
5	6
7	7
7	7
9	9
9	9

Sort each row in parallel, alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

Sort each column in parallel:

0	0	0	1	1	1
3	2	2	2	2	2
3	3	3	3	3	3
4	4	4	5	4	4
6	5	5	5	6	5
7	8	7	7	6	6
9	8	8	8	9	9
9	9	9	9	9	9

Sort each row in parallel, alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

Sort each column in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

Sort each row in parallel,
alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

Sort each column
in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

h row in parallel,

ely ←, →:

1	1	1	2	2
2	2	2	1	1
3	3	4	4	4
5	4	3	3	3
5	6	6	7	7
7	6	5	5	5
8	9	9	9	9
9	9	9	8	8

Sort each column

in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

Sort eac

← or →

0	0
2	2
3	3
4	4
5	5
6	6
8	8
9	9

parallel,

2	2
1	1
4	4
3	3
7	7
5	5
9	9
8	8

Sort each column in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

Sort each row in p

$\leftarrow$ or $\rightarrow$ as desire

0	0	0	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
6	6	7	7	7	7
8	8	8	8	8	9
9	9	9	9	9	9

Sort each column in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

Sort each row in parallel, $\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Sort each column in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

Sort each row in parallel, $\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

h column
el:

0	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	5	4	4	4	4
5	5	6	5	5	5
7	7	6	6	7	7
8	8	9	9	8	8
9	9	9	9	9	9

Sort each row in parallel,
$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Chips ar
towards
parallelis

GPUs: p
Old Xeo
New Xe

1	1
2	2
3	3
4	4
5	5
7	7
8	8
9	9

Sort each row in parallel,
$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Chips are in fact e
towards having thi
parallelism and co

GPUs: parallel +
Old Xeon Phi: par
New Xeon Phi: pa

Sort each row in parallel,
$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Chips are in fact evolving
towards having this much
parallelism and communicati

GPUs: parallel + global RA
Old Xeon Phi: parallel + rir
New Xeon Phi: parallel + m

Sort each row in parallel,
$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Sort each row in parallel,
$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Algorithm designers don't even get the right exponent without taking this into account.

Sort each row in parallel,
$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Algorithm designers don't even get the right exponent without taking this into account.

Shock waves into high levels of domain-specific algorithm design: e.g., for "NFS" factorization, replace "sieving" with "ECM".

h row in parallel,
as desired:

0	1	1	1	1	1
2	2	2	2	2	3
3	3	3	3	3	3
4	4	4	4	4	5
5	5	5	5	6	6
7	7	7	7	7	8
8	8	8	9	9	9
9	9	9	9	9	9

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Algorithm designers
don't even get the right exponent without taking this into account.

Shock waves into high levels of domain-specific algorithm design: e.g., for "NFS" factorization, replace "sieving" with "ECM".

The futu

At this p
say, "Bu
P, and a
will proc
"No, the
would ha
(much n
we have
be unrel
alternati
class of
far bette

parallel,

d:

1	1
2	3
3	3
4	5
6	6
7	8
9	9
9	9

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Algorithm designers
don't even get the right exponent without taking this into account.

Shock waves into high levels of domain-specific algorithm design: e.g., for "NFS" factorization, replace "sieving" with "ECM".

The future of com

At this point man
say, "But he shou
P, and an optimiz
will produce Q." T
"No, the optimizi
would have to be
(much more so th
we have now) tha
be unreliable." I h
alternative to prop
class of software w
far better. . . .

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Algorithm designers don't even get the right exponent without taking this into account.

Shock waves into high levels of domain-specific algorithm design: e.g., for "NFS" factorization, replace "sieving" with "ECM".

The future of compilers

At this point many readers w say, "But he should only wri P, and an optimizing compil will produce Q." To this I sa "No, the optimizing compile would have to be so complic (much more so than anythin we have now) that it will in be unreliable." I have anoth alternative to propose, a new class of software which will far better. . . .

Chips are in fact evolving towards having this much parallelism and communication.

GPUs: parallel + global RAM.
Old Xeon Phi: parallel + ring.
New Xeon Phi: parallel + mesh.

Algorithm designers don't even get the right exponent without taking this into account.

Shock waves into high levels of domain-specific algorithm design: e.g., for "NFS" factorization, replace "sieving" with "ECM".

The future of compilers

At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q." To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable." I have another alternative to propose, a new class of software which will be far better. . . .

e in fact evolving
having this much
sm and communication.

parallel + global RAM.
n Phi: parallel + ring.
on Phi: parallel + mesh.

m designers
en get the right exponent
taking this into account.

vaves into high levels of
specific algorithm design:
"NFS" factorization,
"sieving" with "ECM".

The future of compilers

At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q." To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable." I have another alternative to propose, a new class of software which will be far better. . . .

For 15 y
trying to
compiler
quality o
of the M
are cons
than any
compilin
to produ
various t
coder lik
them int
automat
ago, sev
at a typi

evolving
is much
mmunication.

global RAM.
rallel + ring.
arallel + mesh.

rs
e right exponent
s into account.

high levels of
gorithm design:
ctorization,
with "ECM".

The future of compilers

At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q." To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable." I have another alternative to propose, a new class of software which will be far better. . . .

For 15 years or so
trying to think of
compiler that real
quality code. For
of the Mix progra
are considerably m
than any of today
compiling scheme
to produce. I've t
various technique
coder like myself u
them into some sy
automatic system.
ago, several stude
at a typical sampl

ion.

M.

ng.

nesh.

onent

ount.

s of

esign:

n,

M".

The future of compilers

At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q." To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable." I have another alternative to propose, a new class of software which will be far better. . . .

For 15 years or so I have bee
trying to think of how to wr
compiler that really produce
quality code. For example, r
of the Mix programs in my b
are considerably more efficie
than any of today's most vis
compiling schemes would be
to produce. I've tried to stu
various techniques that a ha
coder like myself uses, and t
them into some systematic a
automatic system. A few ye
ago, several students and I l
at a typical sample of FORT

The future of compilers

At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q." To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable." I have another alternative to propose, a new class of software which will be far better. . . .

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN ure of compilers

point many readers will
ut he should only write
an optimizing compiler
duce Q." To this I say,
e optimizing compiler
ave to be so complicated
more so than anything
now) that it will in fact
iable." I have another
ive to propose, a new
software which will be
er. . . .

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN

program
hard to
could pr
compete
optimize
found ou
up agair
compiler
with the
know pr
whether
etc. And
good lar
such a d

pilers

y readers will
ld only write
ing compiler
To this I say,
ng compiler
so complicated
an anything
t it will in fact
have another
pose, a new
which will be

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN

programs [51], and
hard to see how a
could produce cod
compete with our
optimized object p
found ourselves al
up against the sa
compiler needs to
with the programm
know properties o
whether certain ca
etc. And we could
good language in
such a dialog.

will
ite
ler
ay,
er
cated
ng
fact
her
w
be

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN

programs [51], and we all tri
hard to see how a machine
could produce code that wo
compete with our best hand
optimized object programs.
found ourselves always runn
up against the same problen
compiler needs to be in a di
with the programmer; it nee
know properties of the data,
whether certain cases can ar
etc. And we couldn't think
good language in which to h
such a dialog.

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.

ears or so I have been
o think of how to write a
r that really produces top
code. For example, most
Mix programs in my books
iderably more efficient
y of today's most visionary
ng schemes would be able
uce. I've tried to study the
techniques that a hand-
ke myself uses, and to fit
to some systematic and
tic system. A few years
eral students and I looked
ical sample of FORTRAN

programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.

For som
me) had
optimiza
always r
the-scen
in the m
the prog
to know
lifted fro
ran acro
[42] tha
should b
optimizi
its optin
languag

I have been
how to write a
ly produces top
example, most
ms in my books
nore efficient
's most visionary
s would be able
ried to study the
s that a hand-
uses, and to fit
ystematic and
A few years
nts and I looked
e of FORTRAN

programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.

For some reason w
me) had a mental
optimization, nam
always regarded it
the-scenes activity
in the machine lar
the programmer is
to know. This vei
lifted from my eye
ran across a remar
[42] that, ideally, a
should be designed
optimizing compile
its optimizations i
language. Of cour

en
rite a
s top
most
books
ent
sionary
e able
dy the
nd-
to fit
and
ears
looked
TRAN

programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.

For some reason we all (esp
me) had a mental block abo
optimization, namely that w
always regarded it as a behi
the-scenes activity, to be do
in the machine language, wh
the programmer isn't suppo
to know. This veil was first
lifted from my eyes . . . whe
ran across a remark by Hoa
[42] that, ideally, a languag
should be designed so that a
optimizing compiler can des
its optimizations in the sour
language. Of course! . . .

programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes . . . when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the source language. Of course! . . .

s [51], and we all tried
see how a machine
oduce code that would
e with our best hand-
ed object programs. We
urselves always running
nst the same problem: the
r needs to be in a dialog
e programmer; it needs to
operties of the data, and
certain cases can arise,
d we couldn't think of a
nguage in which to have
dialog.

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes . . . when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the source language. Of course! . . .

The tim
for prog
systems
using su
his beau
possibly
then he
transfor
efficient
much m
than a c
one. . . .
certainly
exciting
becomes

d we all tried
machine
le that would
best hand-
programs. We
ways running
me problem: the
be in a dialog
mer; it needs to
f the data, and
ases can arise,
ln't think of a
which to have

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes . . . when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the source language. Of course! . . .

The time is clearly
for program-manip
systems . . . The pr
using such a syste
his beautifully-stru
possibly inefficient
then he will intera
transformations th
efficient. Such a s
much more power
than a completely
one. . . . As I say,
certainly isn't my
exciting I hope tha
becomes aware of

ied

uld

l-

We

ing

n: the

alog

ds to

and

ise,

of a

have

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes . . . when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the source language. Of course! . . .

The time is clearly ripe
for program-manipulation
systems . . . The programme
using such a system will wri
his beautifully-structured, bu
possibly inefficient, program
then he will interactively spe
transformations that make i
efficient. Such a system will
much more powerful and rel
than a completely automati
one. . . . As I say, this idea
certainly isn't my own; it is
exciting I hope that everyon
becomes aware of its possib

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes . . . when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the source language. Of course! . . .

The time is clearly ripe for program-manipulation systems . . . The programmer using such a system will write his beautifully-structured, but possibly inefficient, program P; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one. . . . As I say, this idea certainly isn't my own; it is so exciting I hope that everyone *soon becomes aware of its possibilities.*