# Engineering cryptographic software

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

---

This is easy, right?

1. Take general principles of software engineering.
2. Apply principles to crypto.

Let's try some examples . . .



1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.

ring
aphic software

. Bernstein

ty of Illinois at Chicago &
che Universiteit Eindhoven

easy, right?

general principles
ftware engineering.
y principles to crypto.

y some examples . . .



1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.



Another
of softw
Make th
and the

ware

n

is at Chicago &
siteit Eindhoven

---

?

rinciples
ineering.
s to crypto.

mples . . .



1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.



Another general pr
of software engine
Make the right thi
and the wrong thi

ago &
hoven

o.

1

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as
`#define ROUNDS 20`
then it is easy to change.

2

Another general principle of software engineering:
Make the right thing simple
and the wrong thing comple

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.



Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.



Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

1972 Parnas "On the criteria to be used in decomposing systems into modules":

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

e.g. If number of cipher rounds is properly modularized as

```
#define ROUNDS 20
```

then it is easy to change.



Another general principle of software engineering: Make the right thing simple and the wrong thing complex.

e.g. Make it difficult to ignore invalid authenticators.

Do not design APIs like this: "The sample code used in this manual omits the checking of status values for clarity, but when using cryptlib you should check return values, particularly for critical functions . . . "

rnas "On the criteria
ed in decomposing
into modules":

pose instead that
ins with a list of
design decisions or
ecisions which are
change. Each module
designed to hide such
on from the others."

umber of cipher rounds
rly modularized as
`ROUNDS 20`
s easy to change.



Another general principle
of software engineering:
Make the right thing simple
and the wrong thing complex.

e.g. Make it difficult to
ignore invalid authenticators.

Do not design APIs like this:
"The sample code used in
this manual omits the checking
of status values for clarity, but
when using cryptlib you should
check return values, particularly
for critical functions . . ."



Not so e

1970s: –
compare
against s
one char
stopping

- AAAAA
- SAAAA
- SEAAA

Attacker
deduces
A few hu
reveal se

the criteria
omposing
ules":

ead that
list of
cisions or
hich are
Each module
o hide such
e others."

cipher rounds
rized as
0
change.



Another general principle
of software engineering:
Make the right thing simple
and the wrong thing complex.

e.g. Make it difficult to
ignore invalid authenticators.

Do not design APIs like this:
"The sample code used in
this manual omits the checking
of status values for clarity, but
when using cryptlib you should
check return values, particularly
for critical functions . . . "



Not so easy: Timi

1970s: TENEX op
compares user-sup
against secret pass
one character at a
stopping at first d

- `AAAAAA` vs. `SECF`
- `SAAAAA` vs. `SECF`
- `SEAAAA` vs. `SECF`

Attacker sees com
deduces position o
A few hundred trie
reveal secret passw

a

ule

h

nds



Another general principle
of software engineering:
Make the right thing simple
and the wrong thing complex.

e.g. Make it difficult to
ignore invalid authenticators.

Do not design APIs like this:
"The sample code used in
this manual omits the checking
of status values for clarity, but
when using cryptlib you should
check return values, particularly
for critical functions . . ."



Not so easy: Timing attacks

1970s: TENEX operating sy
compares user-supplied strin
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `SECRET`: stop
- `SAAAAA` vs. `SECRET`: stop
- `SEAAAA` vs. `SECRET`: stop

Attacker sees comparison ti
deduces position of differenc
A few hundred tries
reveal secret password.

Another general principle
of software engineering:
Make the right thing simple
and the wrong thing complex.

e.g. Make it difficult to
ignore invalid authenticators.

Do not design APIs like this:
"The sample code used in
this manual omits the checking
of status values for clarity, but
when using cryptlib you should
check return values, particularly
for critical functions . . ."



## Not so easy: Timing attacks

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `SECRET`: stop at 1.
- `SAAAAA` vs. `SECRET`: stop at 2.
- `SEAAAA` vs. `SECRET`: stop at 3.

Attacker sees comparison time,
deduces position of difference.
A few hundred tries
reveal secret password.

general principle
are engineering:
e right thing simple
wrong thing complex.

ke it difficult to
nvalid authenticators.

design APIs like this:
mple code used in
nual omits the checking
s values for clarity, but
ing cryptlib you should
turn values, particularly
cal functions . . . ”





## Not so easy: Timing attacks

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `SECRET`: stop at 1.
- `SAAAAA` vs. `SECRET`: stop at 2.
- `SEAAAA` vs. `SECRET`: stop at 3.

Attacker sees comparison time,
deduces position of difference.
A few hundred tries
reveal secret password.

How typ
16-byte

```
for
  if
retu
```

Fix, elim
from sec

```
diff
for
  di
retu
```

Notice t
makes th
and the

rinciple
ering:
ng simple
ng complex.

ult to
nenticators.

ls like this:
used in
the checking
r clarity, but
b you should
es, particularly
ns . . .”



## Not so easy: Timing attacks

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference:

- AAAAAA vs. SECRET: stop at 1.
- SAAAAA vs. SECRET: stop at 2.
- SEAAAA vs. SECRET: stop at 3.

Attacker sees comparison time,
deduces position of difference.
A few hundred tries
reveal secret password.



How typical softwa
16-byte authentica

```
for (i = 0;i
  if (x[i] !=
return 1;
```

Fix, eliminating in
from secrets to tin

```
diff = 0;
for (i = 0;i
  diff |= x[i
return 1 & ((
```

Notice that the lar
makes the wrong t
and the right thing

ex.

s.

:

ing

ut

uld

arly



## Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `SECRET`: stop at 1.
- `SAAAAA` vs. `SECRET`: stop at 2.
- `SEAAAA` vs. `SECRET`: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.



How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) re
return 1;
```

Fix, eliminating information from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >
```

Notice that the language makes the wrong thing simp and the right thing complex

## Not so easy: Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- `AAAAAA` vs. `SECRET`: stop at 1.
- `SAAAAA` vs. `SECRET`: stop at 2.
- `SEAAAA` vs. `SECRET`: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.



How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

## easy: Timing attacks

TENEX operating system
es user-supplied string
secret password
racter at a time,
g at first difference:

A vs. SECRET: stop at 1.
A vs. SECRET: stop at 2.
A vs. SECRET: stop at 3.

r sees comparison time,
position of difference.
undred tries
ecret password.



How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Languag
“right” i

So mista

ng attacks

erating system
plied string
sword
time,
ifference:

RET: stop at 1.
RET: stop at 2.
RET: stop at 3.

parison time,
f difference.
es
vord.



How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer
“right” is too wea

So mistakes contin

s

ystem

g

at 1.

at 2.

at 3.

me,

ce.



How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.



Language designer's notion o
"right" is too weak for secur

So mistakes continue to hap

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.



Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.



Language designer's notion of
"right" is too weak for security.

So mistakes continue to happen.

One of many examples,
part of the reference software for
CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

ical software checks
authenticator:

```
(i = 0;i < 16;++i)
 (x[i] != y[i]) return 0;
rn 1;
```

inating information flow
rets to timings:

```
 = 0;
(i = 0;i < 16;++i)
ff |= x[i] ^ y[i];
rn 1 & ((diff-1) >> 8);
```

hat the language
e wrong thing simple
right thing complex.



Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples,
part of the reference software for
CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

Do timin

Objectio

are checks
ator:

```
< 16;++i)
 y[i]) return 0;
```

formation flow
nings:

```
< 16;++i)
] ^ y[i];
diff-1) >> 8);
```

nguage
thing simple
g complex.



Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```



Do timing attacks

Objection: "Timir

```
turn 0;
```

flow

```
> 8);
```

le

.



Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

Do timing attacks really wor

Objection: "Timings are noi

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```



Do timing attacks really work?

Objection: "Timings are noisy!"

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```



Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```



## Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```



## Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

ge designer's notion of
is too weak for security.

akes continue to happen.

many examples,
the reference software for
R candidate CLOC:

```
are the tag */

 0;i < CRYPTO_ABYTES;i++)
g[i] != c[(*mlen) + i]){
urn RETURN_TAG_NO_MATCH;

RETURN_SUCCESS;
```



## Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender
must block *all* information flow.

Answer #2: Attacker uses
statistics to eliminate noise.

Answer #3, what the
1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

## Defende

Some of

1996 Ko
attacks

Briefly n
Kocher a
Schneier
secret ar
affect tir

2002 Pa
Suzaki–S
timing a

’s notion of
k for security.

nue to happen.

nples,
ce software for
e CLOC:

```
ag */

YPTO_ABYTES;i++)
[(*mlen) + i]){
N_TAG_NO_MATCH;

CCESS;
```



Do timing attacks really work?

Objection: “Timings are noisy!”

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.



Defenders don’t le

Some of the litera

1996 Kocher point
attacks on cryptog

Briefly mentioned
Kocher and by 19
Schneier–Wagner–
secret array indice
affect timing via c

2002 Page, 2003
Suzaki–Shigeri–Mi
timing attacks on

of

rity.

pen.

re for

```
ES;i++)
 + i]){
MATCH;
```



## Do timing attacks really work?

Objection: “Timings are noisy!”

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender
must block *all* information flow.

Answer #2: Attacker uses
statistics to eliminate noise.

Answer #3, what the
1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.



## Defenders don’t learn

Some of the literature:

1996 Kocher pointed out tir
attacks on cryptographic key

Briefly mentioned by
Kocher and by 1998 Kelsey–
Schneier–Wagner–Hall:
secret array indices can
affect timing via cache misse

2002 Page, 2003 Tsunoo–Sa
Suzaki–Shigeri–Miyauchi:
timing attacks on DES.

## Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.



## Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

ng attacks really work?

on: "Timings are noisy!"

#1:
ise stop *all* attacks?
antee security, defender
ock *all* information flow.

#2: Attacker uses
s to eliminate noise.

#3, what the
ttackers actually did:
nge boundary,
page faults,
fy timing signal.



## Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaran
load ent

really work?

ngs are noisy!”

ll attacks?
rity, defender
ormation flow.

cker uses
ate noise.

the
ctually did:
ary,
ts,
signal.



## Defenders don’t learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

“Guaranteed” cou
load entire table i

rk?

sy!"

der
low.

:



## Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.

"Guaranteed" countermeasu

load entire table into cache.

## Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.



"Guaranteed" countermeasure: load entire table into cache.

## Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.



"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

Briefly mentioned by Kocher and by 1998 Kelsey–Schneier–Wagner–Hall: secret array indices can affect timing via cache misses.

2002 Page, 2003 Tsunoo–Saito–Suzaki–Shigeri–Miyauchi: timing attacks on DES.



"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

rs don't learn

' the literature:

ocher pointed out timing
on cryptographic key bits.

nentioned by
and by 1998 Kelsey–
–Wagner–Hall:
rray indices can
ming via cache misses.

ge, 2003 Tsunoo–Saito–
Shigeri–Miyauchi:
ttacks on DES.



"Guaranteed" countermeasure:
load entire table into cache.

2004.11/2005.04 Bernstein:
Timing attacks on AES.
Countermeasure isn't safe;
e.g., secret array indices can affect
timing via cache-bank collisions.
What *is* safe: kill all data flow
from secrets to array indices.

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.

Intel rec
OpenSS
counterr
from kno

arn

ture:

ted out timing
graphic key bits.

by
98 Kelsey–
-Hall:
s can
ache misses.

Tsunoo–Saito–
yauchi:
DES.



“Guaranteed” countermeasure:
load entire table into cache.

2004.11/2005.04 Bernstein:
Timing attacks on AES.
Countermeasure isn’t safe;
e.g., secret array indices can affect
timing via cache-bank collisions.
What *is* safe: kill all data flow
from secrets to array indices.

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.



Intel recommends,
OpenSSL integrat
countermeasure: a
from known *lines*

ming
y bits.

–

es.

aito–



“Guaranteed” countermeasure:
load entire table into cache.

2004.11/2005.04 Bernstein:
Timing attacks on AES.
Countermeasure isn’t safe;
e.g., secret array indices can affect
timing via cache-bank collisions.
What *is* safe: kill all data flow
from secrets to array indices.

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.



Intel recommends, and
OpenSSL integrates, cheape
countermeasure: always loac
from known *lines* of cache.

"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.



Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

"Guaranteed" countermeasure: load entire table into cache.

2004.11/2005.04 Bernstein: Timing attacks on AES. Countermeasure isn't safe; e.g., secret array indices can affect timing via cache-bank collisions. What *is* safe: kill all data flow from secrets to array indices.

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.



Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Same issues described in 2004.

"Guaranteed" countermeasure:
load entire table into cache.

2004.11/2005.04 Bernstein:
Timing attacks on AES.
Countermeasure isn't safe;
e.g., secret array indices can affect
timing via cache-bank collisions.
What *is* safe: kill all data flow
from secrets to array indices.

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.



Intel recommends, and
OpenSSL integrates, cheaper
countermeasure: always loading
from known *lines* of cache.

2013 Bernstein–Schwabe
"A word of warning":
This countermeasure isn't safe.
Same issues described in 2004.

2016 Yarom–Genkin–Heninger
"CacheBleed" steals RSA secret
key via timings of OpenSSL.

teed" countermeasure:
ire table into cache.

/2005.04 Bernstein:
attacks on AES.
measure isn't safe;
ret array indices can affect
ia cache-bank collisions.
safe: kill all data flow
crets to array indices.

omer–Osvik–Shamir:
steal Linux AES key
hard-disk encryption.



Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RF
Layer Se
Version
small tin
performa
extent o
fragmen
be large
due to t
existing
of the ti

ntermeasure:
nto cache.

Bernstein:
AES.
sn't safe;
ndices can affect
bank collisions.
all data flow
ray indices.

ik–Shamir:
ux AES key
encryption.



Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

2008 RFC 5246 "T
Layer Security (TL
Version 1.2": "Th
small timing chann
performance depe
extent on the size
fragment, but it is
be large enough to
due to the large bl
existing MACs and
of the timing signa

re:

affect

ons.

ow

.

t

y

n.

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.



2008 RFC 5246 "The Trans
Layer Security (TLS) Protoc
Version 1.2": "This leaves a
small timing channel, since
performance depends to som
extent on the size of the dat
fragment, but it is not belie
be large enough to be exploi
due to the large block size o
existing MACs and the smal
of the timing signal."

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.



2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

Intel recommends, and OpenSSL integrates, cheaper countermeasure: always loading from known *lines* of cache.

2013 Bernstein–Schwabe "A word of warning": This countermeasure isn't safe. Same issues described in 2004.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.



2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols": exploit these timings; steal plaintext.

ommends, and
L integrates, cheaper
measure: always loading
own *lines* of cache.

rnstein–Schwabe
of warning":
untermeasure isn't safe.
sues described in 2004.

rom–Genkin–Heninger
Bleed" steals RSA secret
timings of OpenSSL.



2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols": exploit these timings; steal plaintext.

How to

If possib
to contr

Look for
identifyi
"Divisio
when th
complet
cycles re
values o

Measure
trusting

and
es, cheaper
always loading
of cache.

chwabe
g":
ure isn't safe.
bed in 2004.

in–Heninger
als RSA secret
OpenSSL.



2008 RFC 5246 "The Transport Layer Security (TLS) Protocol, Version 1.2": "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols": exploit these timings; steal plaintext.

How to write cons

If possible, write c
to control instruct

Look for documen
identifying variabil
"Division operatio
when the divide op
completes, with th
cycles required dep
values of the input

Measure cycles rat
trusting CPU docu

er
ding

afe.
04.

ger
ecret
r



2008 RFC 5246 “The Transport Layer Security (TLS) Protocol, Version 1.2”: “This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.”

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols”: exploit these timings; steal plaintext.



How to write constant-time

If possible, write code in asm to control instruction selecti

Look for documentation identifying variability: e.g., “Division operations termina when the divide operation completes, with the number cycles required dependent o values of the input operands

Measure cycles rather than trusting CPU documentatio

2008 RFC 5246 “The Transport Layer Security (TLS) Protocol, Version 1.2”: “This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.”

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols”: exploit these timings; steal plaintext.



## How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., “Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands.”

Measure cycles rather than trusting CPU documentation.

FC 5246 "The Transport
ecurity (TLS) Protocol,
1.2": "This leaves a
ning channel, since MAC
ance depends to some
n the size of the data
t, but it is not believed to
enough to be exploitable,
he large block size of
MACs and the small size
ming signal."

Fardan–Paterson "Lucky
n: breaking the TLS and
ecord protocols": exploit
mings; steal plaintext.



## How to write constant-time code

If possible, write code in asm
to control instruction selection.

Look for documentation
identifying variability: e.g.,
"Division operations terminate
when the divide operation
completes, with the number of
cycles required dependent on the
values of the input operands."

Measure cycles rather than
trusting CPU documentation.

Cut off a
secrets t

Cut off a
secrets t

Cut off a
secrets t

Prefer lo

Prefer ve

Watch o
variable-
Cortex-M

The Transport
LS) Protocol,
is leaves a
nel, since MAC
nds to some
of the data
not believed to
be exploitable,
lock size of
l the small size
al."

terson "Lucky
the TLS and
ocols": exploit
al plaintext.



How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., "Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands."

Measure cycles rather than trusting CPU documentation.

Cut off all data flo
secrets to branch

Cut off all data flo
secrets to array in

Cut off all data flo
secrets to shift/ro

Prefer logic instru

Prefer vector instr

Watch out for CP
variable-time mult
Cortex-M3 and mo

port
col,
n
MAC
he
ta
ved to
itable,
of
l size

ucky
and
ploit
t.



How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., "Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands."

Measure cycles rather than trusting CPU documentation.

Cut off all data flow from secrets to branch conditions

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distar

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g Cortex-M3 and most PowerF

## How to write constant-time code

If possible, write code in asm to control instruction selection.

Look for documentation identifying variability: e.g., "Division operations terminate when the divide operation completes, with the number of cycles required dependent on the values of the input operands."

Measure cycles rather than trusting CPU documentation.



Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.

write constant-time code

le, write code in asm
ol instruction selection.

r documentation
ng variability: e.g.,
n operations terminate
e divide operation
es, with the number of
equired dependent on the
f the input operands."

e cycles rather than
CPU documentation.



Cut off all data flow from
secrets to branch conditions.

Cut off all data flow from
secrets to array indices.

Cut off all data flow from
secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with
variable-time multipliers: e.g.,
Cortex-M3 and most PowerPCs.

Software

Almost a
much slo

stant-time code

ode in asm
ion selection.

tation
ity: e.g.,
ns terminate
peration
ne number of
pendent on the
t operands."

ther than
umentation.



Cut off all data flow from
secrets to branch conditions.

Cut off all data flow from
secrets to array indices.

Cut off all data flow from
secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with
variable-time multipliers: e.g.,
Cortex-M3 and most PowerPCs.

Software optimiza

Almost all softwar
much slower than

code

n
on.

ate

of
n the
s.”

n.

Cut off all data flow from
secrets to branch conditions.

Cut off all data flow from
secrets to array indices.

Cut off all data flow from
secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with
variable-time multipliers: e.g.,
Cortex-M3 and most PowerPCs.



Software optimization

Almost all software is
much slower than it could b

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.



## Software optimization

Almost all software is much slower than it could be.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.



Software optimization

Almost all software is much slower than it could be.

Is software applied to much data? Usually not. Usually the wasted CPU time is negligible.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

Cut off all data flow from secrets to shift/rotate distances.

Prefer logic instructions.

Prefer vector instructions.

Watch out for CPUs with variable-time multipliers: e.g., Cortex-M3 and most PowerPCs.



## Software optimization

Almost all software is much slower than it could be.

Is software applied to much data? Usually not. Usually the wasted CPU time is negligible.

But *crypto software* should be applied to all communication.

Crypto that's too slow $\Rightarrow$ fewer users $\Rightarrow$ fewer cryptanalysts $\Rightarrow$ less attractive for everybody.

all data flow from
o branch conditions.

all data flow from
o array indices.

all data flow from
o shift/rotate distances.

gic instructions.

ector instructions.

ut for CPUs with
time multipliers: e.g.,
M3 and most PowerPCs.





## Software optimization

Almost all software is
much slower than it could be.

Is software applied to much data?
Usually not. Usually the
wasted CPU time is negligible.

But *crypto software* should be
applied to all communication.

Crypto that's too slow ⇒
fewer users ⇒ fewer cryptanalysts
⇒ less attractive for everybody.

Typical
You wan
software
as efficie

Starting
You hav
reference

You hav
(Can rep

You mea
impleme

ow from
conditions.

ow from
dices.

ow from
tate distances.

ctions.

uctions.

Us with
ipliers: e.g.,
ost PowerPCs.



## Software optimization

Almost all software is much slower than it could be.

Is software applied to much data? Usually not. Usually the wasted CPU time is negligible.

But *crypto software* should be applied to all communication.

Crypto that's too slow ⇒ fewer users ⇒ fewer cryptanalysts ⇒ less attractive for everybody.

Typical situation:
You want (constar
software that com
as efficiently as po

Starting point:
You have written a
reference impleme

You have chosen a
(Can repeat for ot

You measure perfo
implementation. N

.

nces.

g.,
PCs.



Software optimization

Almost all software is much slower than it could be.

Is software applied to much data? Usually not. Usually the wasted CPU time is negligible.

But *crypto software* should be applied to all communication.

Crypto that's too slow ⇒ fewer users ⇒ fewer cryptanalysts ⇒ less attractive for everybody.



Typical situation:
You want (constant-time) software that computes ciph as efficiently as possible.

Starting point:
You have written a reference implementation of

You have chosen a target C
(Can repeat for other CPUs

You measure performance o implementation. Now what?

## Software optimization

Almost all software is much slower than it could be.

Is software applied to much data? Usually not. Usually the wasted CPU time is negligible.

But *crypto software* should be applied to all communication.

Crypto that's too slow $\Rightarrow$ fewer users $\Rightarrow$ fewer cryptanalysts $\Rightarrow$ less attractive for everybody.



Typical situation:
You want (constant-time) software that computes cipher $X$ as efficiently as possible.

Starting point:
You have written a reference implementation of $X$.

You have chosen a target CPU. (Can repeat for other CPUs.)

You measure performance of the implementation. Now what?

e optimization

all software is
ower than it could be.

are applied to much data?
not. Usually the
CPU time is negligible.

*pto software* should be
to all communication.

that's too slow ⇒
ers ⇒ fewer cryptanalysts
attractive for everybody.



Typical situation:
You want (constant-time)
software that computes cipher $X$
as efficiently as possible.

Starting point:
You have written a
reference implementation of $X$.

You have chosen a target CPU.
(Can repeat for other CPUs.)

You measure performance of the
implementation. Now what?

A simpli

Target C
microcor
one ARM

Referenc

```
int sum
{
  int re
  int i
  for (i
    resu
  return
}
```

tion

e is
it could be.

l to much data?
lly the
is negligible.

*re* should be
munication.

slow ⇒
ver cryptanalysts
for everybody.



Typical situation:
You want (constant-time)
software that computes cipher $X$
as efficiently as possible.

Starting point:
You have written a
reference implementation of $X$.

You have chosen a target CPU.
(Can repeat for other CPUs.)

You measure performance of the
implementation. Now what?

A simplified examp

Target CPU: TI LI
microcontroller co
one ARM Cortex-M

Reference impleme

```
int sum(int *x)
{
  int result = 0
  int i;
  for (i = 0;i <
    result += x[
  return result;
}
```

e.

data?

le.

be

n.

alysts

ody.



Typical situation:
You want (constant-time) software that computes cipher $X$ as efficiently as possible.

Starting point:
You have written a reference implementation of $X$.

You have chosen a target CPU. (Can repeat for other CPUs.)

You measure performance of the implementation. Now what?



## A simplified example

Target CPU: TI LM4F120H microcontroller containing one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i
    result += x[i];
  return result;
}
```

Typical situation:
You want (constant-time)
software that computes cipher $X$
as efficiently as possible.

Starting point:
You have written a
reference implementation of $X$.

You have chosen a target CPU.
(Can repeat for other CPUs.)

You measure performance of the
implementation. Now what?



## A simplified example

Target CPU: TI LM4F120H5QR
microcontroller containing
one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += x[i];
  return result;
}
```

situation:
nt (constant-time)
that computes cipher $X$
ently as possible.

point:
e written a
e implementation of $X$.

e chosen a target CPU.
peat for other CPUs.)

asure performance of the
ntation. Now what?



## A simplified example

Target CPU: TI LM4F120H5QR
microcontroller containing
one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += x[i];
  return result;
}
```

Counting

```
static v
  *const
  = (voi
...

int befo
int resu
int afte
UARTprin
  result
```

Output s
Change

nt-time)
putes cipher $X$
ossible.

a
ntation of $X$.

a target CPU.
her CPUs.)

ormance of the
Now what?



## A simplified example

Target CPU: TI LM4F120H5QR microcontroller containing one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += x[i];
  return result;
}
```

Counting cycles:

```
static volatile
  *const DWT_CYC
  = (void *) 0xE
...

int beforesum =
int result = sum
int aftersum = *
UARTprintf("sum
  result,aftersu
```

Output shows 801

Change 1000 to 50

er $X$

$X$.

PU.

.)

f the

?



## A simplified example

Target CPU: TI LM4F120H5QR microcontroller containing one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += x[i];
  return result;
}
```

Counting cycles:

```
static volatile unsigned
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCC
int result = sum(x);
int aftersum = *DWT_CYCCN
UARTprintf("sum %d %d\n",
  result,aftersum-befores
```

Output shows 8012 cycles.

Change 1000 to 500: 4012.

# A simplified example

Target CPU: TI LM4F120H5QR microcontroller containing one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += x[i];
  return result;
}
```



Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.
Change 1000 to 500: 4012.

fied example

CPU: TI LM4F120H5QR
ntroller containing
M Cortex-M4F core.

ce implementation:

```
(int *x)

esult = 0;
;
i = 0;i < 1000;++i)
ult += x[i];
n result;
```



Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.
Change 1000 to 500: 4012.

"Okay, 8
Um, are
really th

ple

M4F120H5QR
ntaining
M4F core.

entation:

```
;

 1000;++i)
i];
```



Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.
Change 1000 to 500: 4012.

“Okay, 8 cycles pe
Um, are microcon
really this slow at

5QR

)

Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.

Change 1000 to 500: 4012.



“Okay, 8 cycles per addition

Um, are microcontrollers

really this slow at addition?”

Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.

Change 1000 to 500: 4012.



"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.
Change 1000 to 500: 4012.



"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.
Change 1000 to 500: 4012.



"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Good practice:
Figure out lower bound for
cycles spent on arithmetic etc.
Understand gap between
lower bound and observed time.

g cycles:

```
volatile unsigned int
t DWT_CYCCNT
id *) 0xE0001004;



oresum = *DWT_CYCCNT;
ult = sum(x);
ersum = *DWT_CYCCNT;
ntf("sum %d %d\n",
t,aftersum-beforesum);
```

shows 8012 cycles.
1000 to 500: 4012.



"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Good practice:
Figure out lower bound for
cycles spent on arithmetic etc.
Understand gap between
lower bound and observed time.

Find "A
Technica
Rely on
M4F = I

Manual
"implem
architect

Points to
Architec
which de
e.g., "AI

First ma
ADD tal

```
unsigned int
CNT
0001004;

*DWT_CYCCNT;
(x);
DWT_CYCCNT;
%d %d\n",
m-beforesum);
```

2 cycles.
00: 4012.



"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Good practice:
Figure out lower bound for
cycles spent on arithmetic etc.
Understand gap between
lower bound and observed time.

Find "ARM Corte
Technical Referen
Rely on Wikipedia
M4F = M4 + float

Manual says that
"implements the A
architecture profile

Points to the "AR
Architecture Refer
which defines instr
e.g., "ADD" for 3

First manual says
ADD takes just 1

int

NT;

T;

um);



"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Good practice:
Figure out lower bound for
cycles spent on arithmetic etc.
Understand gap between
lower bound and observed time.



Find "ARM Cortex-M4 Proc
Technical Reference Manual
Rely on Wikipedia comment
M4F = M4 + floating-point

Manual says that Cortex-M4
"implements the ARMv7E-M
architecture profile".

Points to the "ARMv7-M
Architecture Reference Man
which defines instructions:
e.g., "ADD" for 32-bit addit

First manual says that
ADD takes just 1 cycle.

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Good practice:
Figure out lower bound for
cycles spent on arithmetic etc.
Understand gap between
lower bound and observed time.



Find "ARM Cortex-M4 Processor
Technical Reference Manual".
Rely on Wikipedia comment that
M4F $=$ M4 $+$ floating-point unit.

Manual says that Cortex-M4
"implements the ARMv7E-M
architecture profile".

Points to the "ARMv7-M
Architecture Reference Manual",
which defines instructions:
e.g., "ADD" for 32-bit addition.

First manual says that
ADD takes just 1 cycle.

3 cycles per addition.
microcontrollers
is slow at addition?”

ctice:
ndom “optimizations”
eak compiler options)
u get bored.
e fastest results.

actice:
ut lower bound for
pent on arithmetic etc.
and gap between
ound and observed time.



Find “ARM Cortex-M4 Processor Technical Reference Manual”. Rely on Wikipedia comment that M4F = M4 + floating-point unit.

Manual says that Cortex-M4 “implements the ARMv7E-M architecture profile”.

Points to the “ARMv7-M Architecture Reference Manual”, which defines instructions: e.g., “ADD” for 32-bit addition.

First manual says that ADD takes just 1 cycle.

Inputs a
“integer
has 16 i
special-p
and “pro

Each ele
be “load

Basic lo
Manual
a note a
Then mo
instructi
address
then it s

er addition.
trollers
addition?"

otimizations"
ler options)
d.
esults.

ound for
ithmetic etc.
etween
bserved time.



Find "ARM Cortex-M4 Processor Technical Reference Manual". Rely on Wikipedia comment that M4F $=$ M4 $+$ floating-point unit.

Manual says that Cortex-M4 "implements the ARMv7E-M architecture profile".

Points to the "ARMv7-M Architecture Reference Manual", which defines instructions: e.g., "ADD" for 32-bit addition.

First manual says that ADD takes just 1 cycle.

Inputs and output
"integer registers"
has 16 integer reg
special-purpose "s
and "program cou

Each element of x
be "loaded" into a

Basic load instruct
Manual says 2 cyc
a note about "pipe
Then more explan
instruction is also
address not based
then it saves 1 cyc

.

”

ns”

s)

tc.

me.



Find “ARM Cortex-M4 Processor Technical Reference Manual”. Rely on Wikipedia comment that M4F $=$ M4 $+$ floating-point unit.

Manual says that Cortex-M4 “implements the ARMv7E-M architecture profile”.

Points to the “ARMv7-M Architecture Reference Manual”, which defines instructions: e.g., “ADD” for 32-bit addition.

First manual says that ADD takes just 1 cycle.



Inputs and output of ADD a
“integer registers”. ARMv7-
has 16 integer registers, incl
special-purpose “stack point
and “program counter”.

Each element of `x` array nee
be “loaded” into a register.

Basic load instruction: LDR
Manual says 2 cycles but ad
a note about “pipelining”.
Then more explanation: if n
instruction is also LDR (wit
address not based on first L
then it saves 1 cycle.

Find “ARM Cortex-M4 Processor Technical Reference Manual”. Rely on Wikipedia comment that M4F = M4 + floating-point unit.

Manual says that Cortex-M4 “implements the ARMv7E-M architecture profile”.

Points to the “ARMv7-M Architecture Reference Manual”, which defines instructions: e.g., “ADD” for 32-bit addition.

First manual says that ADD takes just 1 cycle.



Inputs and output of ADD are “integer registers”. ARMv7-M has 16 integer registers, including special-purpose “stack pointer” and “program counter”.

Each element of `x` array needs to be “loaded” into a register.

Basic load instruction: LDR. Manual says 2 cycles but adds a note about “pipelining”. Then more explanation: if next instruction is also LDR (with address not based on first LDR) then it saves 1 cycle.

RM Cortex-M4 Processor
al Reference Manual".
Wikipedia comment that
M4 + floating-point unit.

says that Cortex-M4
ents the ARMv7E-M
ture profile".

o the "ARMv7-M
ture Reference Manual",
efines instructions:
DD" for 32-bit addition.

nual says that
kes just 1 cycle.



Inputs and output of ADD are "integer registers". ARMv7-M has 16 integer registers, including special-purpose "stack pointer" and "program counter".

Each element of x array needs to be "loaded" into a register.

Basic load instruction: LDR. Manual says 2 cycles but adds a note about "pipelining". Then more explanation: if next instruction is also LDR (with address not based on first LDR) then it saves 1 cycle.

$n$ consec
takes on
("more
pipelined

Can ach
in other
but noth

Lower b
$2n+1$ c
including

Why obs
non-cons
costs of

x-M4 Processor
ce Manual".
comment that
ting-point unit.

Cortex-M4
ARMv7E-M
e".

Mv7-M
rence Manual",
ructions:
2-bit addition.

that
cycle.



Inputs and output of ADD are "integer registers". ARMv7-M has 16 integer registers, including special-purpose "stack pointer" and "program counter".

Each element of x array needs to be "loaded" into a register.

Basic load instruction: LDR. Manual says 2 cycles but adds a note about "pipelining". Then more explanation: if next instruction is also LDR (with address not based on first LDR) then it saves 1 cycle.

$n$ consecutive LDF
takes only $n + 1$ c
("more multiple L
pipelined together

Can achieve this s
in other ways (LD
but nothing seems

Lower bound for $n$
$2n + 1$ cycles,
including $n$ cycles

Why observed tim
non-consecutive L
costs of manipulat

cessor
”.
that
unit.

4
M

ual”,

tion.



Inputs and output of ADD are “integer registers”. ARMv7-M has 16 integer registers, including special-purpose “stack pointer” and “program counter”.

Each element of `x` array needs to be “loaded” into a register.

Basic load instruction: LDR. Manual says 2 cycles but adds a note about “pipelining”. Then more explanation: if next instruction is also LDR (with address not based on first LDR) then it saves 1 cycle.

$n$ consecutive LDRs takes only $n + 1$ cycles (“more multiple LDRs can b pipelined together”).

Can achieve this speed in other ways (LDRD, LDM but nothing seems faster.

Lower bound for $n$ LDR $+$ $n$ $2n + 1$ cycles, including $n$ cycles of arithme

Why observed time is higher non-consecutive LDRs; costs of manipulating `i`.

Inputs and output of ADD are "integer registers". ARMv7-M has 16 integer registers, including special-purpose "stack pointer" and "program counter".

Each element of `x` array needs to be "loaded" into a register.

Basic load instruction: LDR. Manual says 2 cycles but adds a note about "pipelining". Then more explanation: if next instruction is also LDR (with address not based on first LDR) then it saves 1 cycle.



$n$ consecutive LDRs takes only $n+1$ cycles ("more multiple LDRs can be pipelined together").

Can achieve this speed in other ways (LDRD, LDM) but nothing seems faster.

Lower bound for $n$ LDR $+$ $n$ ADD: $2n+1$ cycles, including $n$ cycles of arithmetic.

Why observed time is higher: non-consecutive LDRs; costs of manipulating `i`.

nd output of ADD are
registers". ARMv7-M
nteger registers, including
purpose "stack pointer"
ogram counter".

ement of `x` array needs to
led" into a register.

ad instruction: LDR.
says 2 cycles but adds
bout "pipelining".
ore explanation: if next
on is also LDR (with
not based on first LDR)
aves 1 cycle.



$n$ consecutive LDRs
takes only $n+1$ cycles
("more multiple LDRs can be
pipelined together").

Can achieve this speed
in other ways (LDRD, LDM)
but nothing seems faster.

Lower bound for $n$ LDR + $n$ ADD:
$2n+1$ cycles,
including $n$ cycles of arithmetic.

Why observed time is higher:
non-consecutive LDRs;
costs of manipulating `i`.

```
int sum
{
  int r
  int *
  int x
      x
  while
    x0
    x1
    x2
    x3
    x4
    x5
    x6
```

of ADD are
. ARMv7-M
isters, including
tack pointer"
nter".

array needs to
a register.

tion: LDR.
les but adds
elining".
ation: if next
LDR (with
on first LDR)
cle.



$n$ consecutive LDRs
takes only $n + 1$ cycles
("more multiple LDRs can be
pipelined together").

Can achieve this speed
in other ways (LDRD, LDM)
but nothing seems faster.

Lower bound for $n$ LDR + $n$ ADD:
$2n + 1$ cycles,
including $n$ cycles of arithmetic.

Why observed time is higher:
non-consecutive LDRs;
costs of manipulating `i`.

```
int sum(int *x)
{
  int result = 0
  int *y = x + 1
  int x0,x1,x2,x
      x5,x6,x7,x

  while (x != y)
    x0 = 0[(vola
    x1 = 1[(vola
    x2 = 2[(vola
    x3 = 3[(vola
    x4 = 4[(vola
    x5 = 5[(vola
    x6 = 6[(vola
```

are
-M
uding
ter"

ds to

.
ds

ext
n
DR)



$n$ consecutive LDRs
takes only $n+1$ cycles
("more multiple LDRs can be pipelined together").

Can achieve this speed
in other ways (LDRD, LDM)
but nothing seems faster.

Lower bound for $n$ LDR + $n$ ADD:
$2n+1$ cycles,
including $n$ cycles of arithmetic.

Why observed time is higher:
non-consecutive LDRs;
costs of manipulating `i`.

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int
    x1 = 1[(volatile int
    x2 = 2[(volatile int
    x3 = 3[(volatile int
    x4 = 4[(volatile int
    x5 = 5[(volatile int
    x6 = 6[(volatile int
```

$n$ consecutive LDRs
takes only $n+1$ cycles
("more multiple LDRs can be
pipelined together").

Can achieve this speed
in other ways (LDRD, LDM)
but nothing seems faster.

Lower bound for $n$ LDR + $n$ ADD:
$2n+1$ cycles,
including $n$ cycles of arithmetic.

Why observed time is higher:
non-consecutive LDRs;
costs of manipulating `i`.



```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int *)x];
    x1 = 1[(volatile int *)x];
    x2 = 2[(volatile int *)x];
    x3 = 3[(volatile int *)x];
    x4 = 4[(volatile int *)x];
    x5 = 5[(volatile int *)x];
    x6 = 6[(volatile int *)x];
```

cutive LDRs
ly $n+1$ cycles
multiple LDRs can be
d together").

ieve this speed
ways (LDRD, LDM)
hing seems faster.

ound for $n$ LDR + $n$ ADD:
cycles,
g $n$ cycles of arithmetic.

served time is higher:
secutive LDRs;
manipulating `i`.



```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int *)x];
    x1 = 1[(volatile int *)x];
    x2 = 2[(volatile int *)x];
    x3 = 3[(volatile int *)x];
    x4 = 4[(volatile int *)x];
    x5 = 5[(volatile int *)x];
    x6 = 6[(volatile int *)x];
```

```
x7 =
x8 =
x9 =
resu
resu
resu
resu
resu
resu
resu
resu
resu
resu
x0 =
x1 =
```

Rs
ycles
DRs can be
").

peed
RD, LDM)
faster.

$n$ LDR + $n$ ADD:

of arithmetic.

e is higher:
DRs;
ting i.



```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int *)x];
    x1 = 1[(volatile int *)x];
    x2 = 2[(volatile int *)x];
    x3 = 3[(volatile int *)x];
    x4 = 4[(volatile int *)x];
    x5 = 5[(volatile int *)x];
    x6 = 6[(volatile int *)x];
```

```
    x7 = 7[(vola
    x8 = 8[(vola
    x9 = 9[(vola
    result += x0
    result += x1
    result += x2
    result += x3
    result += x4
    result += x5
    result += x6
    result += x7
    result += x8
    result += x9
    x0 = 10[(vol
    x1 = 11[(vol
```

be

)

ADD:

etic.

r:



```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int *)x];
    x1 = 1[(volatile int *)x];
    x2 = 2[(volatile int *)x];
    x3 = 3[(volatile int *)x];
    x4 = 4[(volatile int *)x];
    x5 = 5[(volatile int *)x];
    x6 = 6[(volatile int *)x];
```

```
x7 = 7[(volatile int
x8 = 8[(volatile int
x9 = 9[(volatile int
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
x0 = 10[(volatile int
x1 = 11[(volatile int
```

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int *)x];
    x1 = 1[(volatile int *)x];
    x2 = 2[(volatile int *)x];
    x3 = 3[(volatile int *)x];
    x4 = 4[(volatile int *)x];
    x5 = 5[(volatile int *)x];
    x6 = 6[(volatile int *)x];
```



```
    x7 = 7[(volatile int *)x];
    x8 = 8[(volatile int *)x];
    x9 = 9[(volatile int *)x];
    result += x0;
    result += x1;
    result += x2;
    result += x3;
    result += x4;
    result += x5;
    result += x6;
    result += x7;
    result += x8;
    result += x9;
    x0 = 10[(volatile int *)x];
    x1 = 11[(volatile int *)x];
```

```
(int *x)

esult = 0;
y = x + 1000;
0,x1,x2,x3,x4,
5,x6,x7,x8,x9;

 (x != y) {
= 0[(volatile int *)x];
= 1[(volatile int *)x];
= 2[(volatile int *)x];
= 3[(volatile int *)x];
= 4[(volatile int *)x];
= 5[(volatile int *)x];
= 6[(volatile int *)x];
```



```
x7 = 7[(volatile int *)x];
x8 = 8[(volatile int *)x];
x9 = 9[(volatile int *)x];
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
x0 = 10[(volatile int *)x];
x1 = 11[(volatile int *)x];
```

```
x2 =
x3 =
x4 =
x5 =
x6 =
x7 =
x8 =
x9 =
x +=
resu
resu
resu
resu
resu
resu
```

```
;
000;
3,x4,
8,x9;

 {
tile int *)x];
tile int *)x];
tile int *)x];
tile int *)x];
tile int *)x];
tile int *)x];
tile int *)x];
```



```
x7 = 7[(volatile int *)x];
x8 = 8[(volatile int *)x];
x9 = 9[(volatile int *)x];
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
x0 = 10[(volatile int *)x];
x1 = 11[(volatile int *)x];
```

```
x2 = 12[(vol
x3 = 13[(vol
x4 = 14[(vol
x5 = 15[(vol
x6 = 16[(vol
x7 = 17[(vol
x8 = 18[(vol
x9 = 19[(vol
x += 20;
result += x0
result += x1
result += x2
result += x3
result += x4
result += x5
```

22

```
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
```

```
x7 = 7[(volatile int *)x];
x8 = 8[(volatile int *)x];
x9 = 9[(volatile int *)x];
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
x0 = 10[(volatile int *)x];
x1 = 11[(volatile int *)x];
```

23

```
x2 = 12[(volatile int
x3 = 13[(volatile int
x4 = 14[(volatile int
x5 = 15[(volatile int
x6 = 16[(volatile int
x7 = 17[(volatile int
x8 = 18[(volatile int
x9 = 19[(volatile int
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```

```
x7 = 7[(volatile int *)x];
x8 = 8[(volatile int *)x];
x9 = 9[(volatile int *)x];
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
x0 = 10[(volatile int *)x];
x1 = 11[(volatile int *)x];
```



```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```

```
= 7[(volatile int *)x];
= 8[(volatile int *)x];
= 9[(volatile int *)x];
ult += x0;
ult += x1;
ult += x2;
ult += x3;
ult += x4;
ult += x5;
ult += x6;
ult += x7;
ult += x8;
ult += x9;
= 10[(volatile int *)x];
= 11[(volatile int *)x];
```



```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```

```
    resu
    resu
    resu
    resu
  }

  returr
}
```

```
tile int *)x];
tile int *)x];
tile int *)x];
;
;
;
;
;
;
;
;
;
;
atile int *)x];
atile int *)x];
```



```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```

```
    result += x6
    result += x7
    result += x8
    result += x9
  }

  return result;
}
```

```
*)x];
*)x];
*)x];
. *)x];
. *)x];
```



```
  x2 = 12[(volatile int *)x];
  x3 = 13[(volatile int *)x];
  x4 = 14[(volatile int *)x];
  x5 = 15[(volatile int *)x];
  x6 = 16[(volatile int *)x];
  x7 = 17[(volatile int *)x];
  x8 = 18[(volatile int *)x];
  x9 = 19[(volatile int *)x];
  x += 20;
  result += x0;
  result += x1;
  result += x2;
  result += x3;
  result += x4;
  result += x5;
```

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```



```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```



```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```



```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

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

```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```



```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

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

```
= 12[(volatile int *)x];
= 13[(volatile int *)x];
= 14[(volatile int *)x];
= 15[(volatile int *)x];
= 16[(volatile int *)x];
= 17[(volatile int *)x];
= 18[(volatile int *)x];
= 19[(volatile int *)x];
= 20;
ult += x0;
ult += x1;
ult += x2;
ult += x3;
ult += x4;
ult += x5;
```



```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

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

A real ex

Salsa20

30.25 cy

Lower bo

64 bytes

$21 \cdot 16$ 1

$20 \cdot 16$ 1

so at lea

ARMv7-

includes

as part o

(Compile

```
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];


;
;
;
;
;
;
```



```
      result += x6;
      result += x7;
      result += x8;
      result += x9;
    }

    return result;
}
```

2526 cycles. Even better in asm.

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

A real example

Salsa20 reference
30.25 cycles/byte

Lower bound for a
64 bytes require
$21 \cdot 16$ 1-cycle AD
$20 \cdot 16$ 1-cycle XO
so at least 10.25 c

ARMv7-M instruct
includes free rotat
as part of XOR ins
(Compiler knows t

```
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
```

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

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



## A real example

Salsa20 reference software:
30.25 cycles/byte on this CP

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

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



## A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)

```
ult += x6;
ult += x7;
ult += x8;
ult += x9;

n result;
```

cles. Even better in asm.

ia: “By the late 1990s for
rformance sensitive code,
ng compilers exceeded the
ance of human experts.”
tion needed]



A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)

Detailed
several c
`load_li`
`store_l`

Can repl
(Compil

Then ob
18 cycle
plus 5 cy
Still far

;

;

;

;

better in asm.

ne late 1990s for
sensitive code,
ers exceeded the
man experts.”
d]



## A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)

Detailed benchma
several cycles/byte
`load_littleendia`
`store_littleendi`

Can replace with L
(Compiler doesn’t

Then observe 23 c
18 cycles/byte for
plus 5 cycles/byte
Still far above 10.2

asm.

0s for
code,
ed the
rts."



A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)



Detailed benchmarks show
several cycles/byte spent on
`load_littleendian` and
`store_littleendian`.

Can replace with LDR and S
(Compiler doesn't see this.)

Then observe 23 cycles/byte
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/

## A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)



Detailed benchmarks show
several cycles/byte spent on
`load_littleendian` and
`store_littleendian`.

Can replace with LDR and STR.
(Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

## A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)



Detailed benchmarks show
several cycles/byte spent on
`load_littleendian` and
`store_littleendian`.

Can replace with LDR and STR.
(Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

Gap is mostly loads, stores.
Minimize load/store cost by
choosing "spills" carefully.

xample

reference software:
cles/byte on this CPU.

ound for arithmetic:
require
-cycle ADDs,
-cycle XORs,
st 10.25 cycles/byte.

M instruction set
free rotation
of XOR instruction.
er knows this.)



Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`.

Can replace with LDR and STR. (Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

Gap is mostly loads, stores.
Minimize load/store cost by choosing "spills" carefully.

Which o
should b
Don't tr
optimize

Make lo
Don't tr
optimize

Spill to
Don't tr
optimize

On bigg
selecting
is critica

software:
on this CPU.

rithmetic:

Ds,
Rs,
cycles/byte.

tion set
ion
struction.
this.)



Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`.

Can replace with LDR and STR. (Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

Gap is mostly loads, stores.
Minimize load/store cost by choosing "spills" carefully.

Which of the 16 S
should be in regist
Don't trust compi
optimize register a

Make loads consec
Don't trust compi
optimize instructio

Spill to FPU inste
Don't trust compi
optimize instructio

On bigger CPUs,
selecting vector in
is critical for perfo

PU.

e.



Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`.

Can replace with LDR and STR. (Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

Gap is mostly loads, stores.
Minimize load/store cost by choosing "spills" carefully.



Which of the 16 Salsa20 wo
should be in registers?
Don't trust compiler to
optimize register allocation.

Make loads consecutive?
Don't trust compiler to
optimize instruction schedul

Spill to FPU instead of stac
Don't trust compiler to
optimize instruction selectio

On bigger CPUs,
selecting vector instructions
is critical for performance.

Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`.

Can replace with LDR and STR. (Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

Gap is mostly loads, stores.
Minimize load/store cost by choosing "spills" carefully.



Which of the 16 Salsa20 words should be in registers?
Don't trust compiler to optimize register allocation.

Make loads consecutive?
Don't trust compiler to optimize instruction scheduling.

Spill to FPU instead of stack?
Don't trust compiler to optimize instruction selection.

On bigger CPUs,
selecting vector instructions is critical for performance.

benchmarks show
cycles/byte spent on
ttleendian and
ittleendian.

ace with LDR and STR.
er doesn't see this.)

serve 23 cycles/byte:
s/byte for rounds,
ycles/byte overhead.
above 10.25 cycles/byte.

nostly loads, stores.
e load/store cost by
g "spills" carefully.



Which of the 16 Salsa20 words should be in registers? Don't trust compiler to optimize register allocation.

Make loads consecutive? Don't trust compiler to optimize instruction scheduling.

Spill to FPU instead of stack? Don't trust compiler to optimize instruction selection.

On bigger CPUs, selecting vector instructions is critical for performance.

The big

CPUs ar
farther a
from nai

rks show
e spent on
n and
an.

LDR and STR.
see this.)

cycles/byte:
rounds,
overhead.
25 cycles/byte.

ds, stores.
re cost by
carefully.



Which of the 16 Salsa20 words
should be in registers?
Don't trust compiler to
optimize register allocation.

Make loads consecutive?
Don't trust compiler to
optimize instruction scheduling.

Spill to FPU instead of stack?
Don't trust compiler to
optimize instruction selection.

On bigger CPUs,
selecting vector instructions
is critical for performance.

<u>The big picture</u>

CPUs are evolving
farther and farther
from naive models

STR.

e:

byte.



Which of the 16 Salsa20 words
should be in registers?
Don't trust compiler to
optimize register allocation.

Make loads consecutive?
Don't trust compiler to
optimize instruction scheduling.

Spill to FPU instead of stack?
Don't trust compiler to
optimize instruction selection.

On bigger CPUs,
selecting vector instructions
is critical for performance.



## The big picture

CPUs are evolving
farther and farther away
from naive models of CPUs.

Which of the 16 Salsa20 words should be in registers? Don't trust compiler to optimize register allocation.

Make loads consecutive? Don't trust compiler to optimize instruction scheduling.

Spill to FPU instead of stack? Don't trust compiler to optimize instruction selection.

On bigger CPUs, selecting vector instructions is critical for performance.



## The big picture

CPUs are evolving farther and farther away from naive models of CPUs.

Which of the 16 Salsa20 words should be in registers?
Don't trust compiler to optimize register allocation.

Make loads consecutive?
Don't trust compiler to optimize instruction scheduling.

Spill to FPU instead of stack?
Don't trust compiler to optimize instruction selection.

On bigger CPUs, selecting vector instructions is critical for performance.



## The big picture

CPUs are evolving farther and farther away from naive models of CPUs.

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.

f the 16 Salsa20 words
e in registers?
ust compiler to
register allocation.

ads consecutive?
ust compiler to
instruction scheduling.

FPU instead of stack?
ust compiler to
instruction selection.

er CPUs,
vector instructions
l for performance.



The big picture

CPUs are evolving
farther and farther away
from naive models of CPUs.

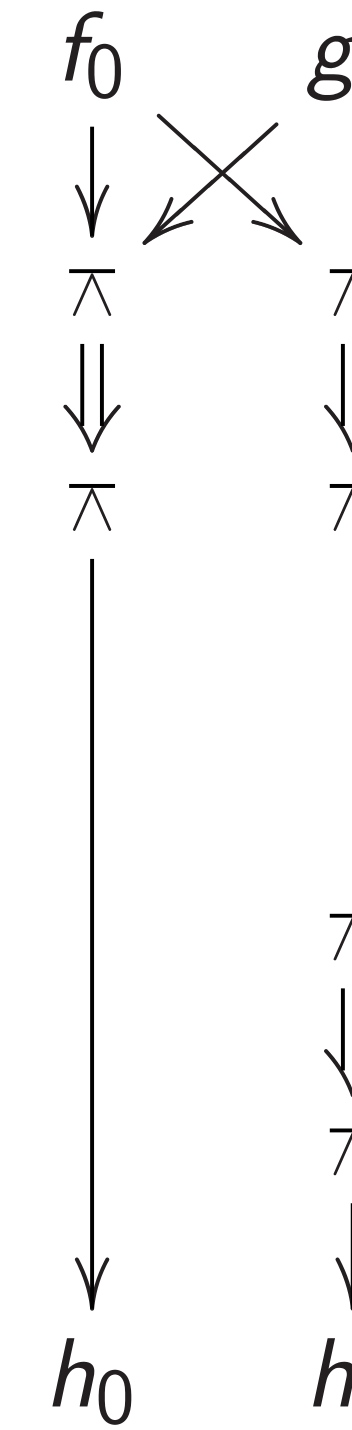
Minor optimization challenges:

- 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

alsa20 words
ers?
ler to
llocation.

cutive?
ler to
on scheduling.

ad of stack?
ler to
on selection.

structions
rmance.



## The big picture

CPUs are evolving
farther and farther away
from naive models of CPUs.

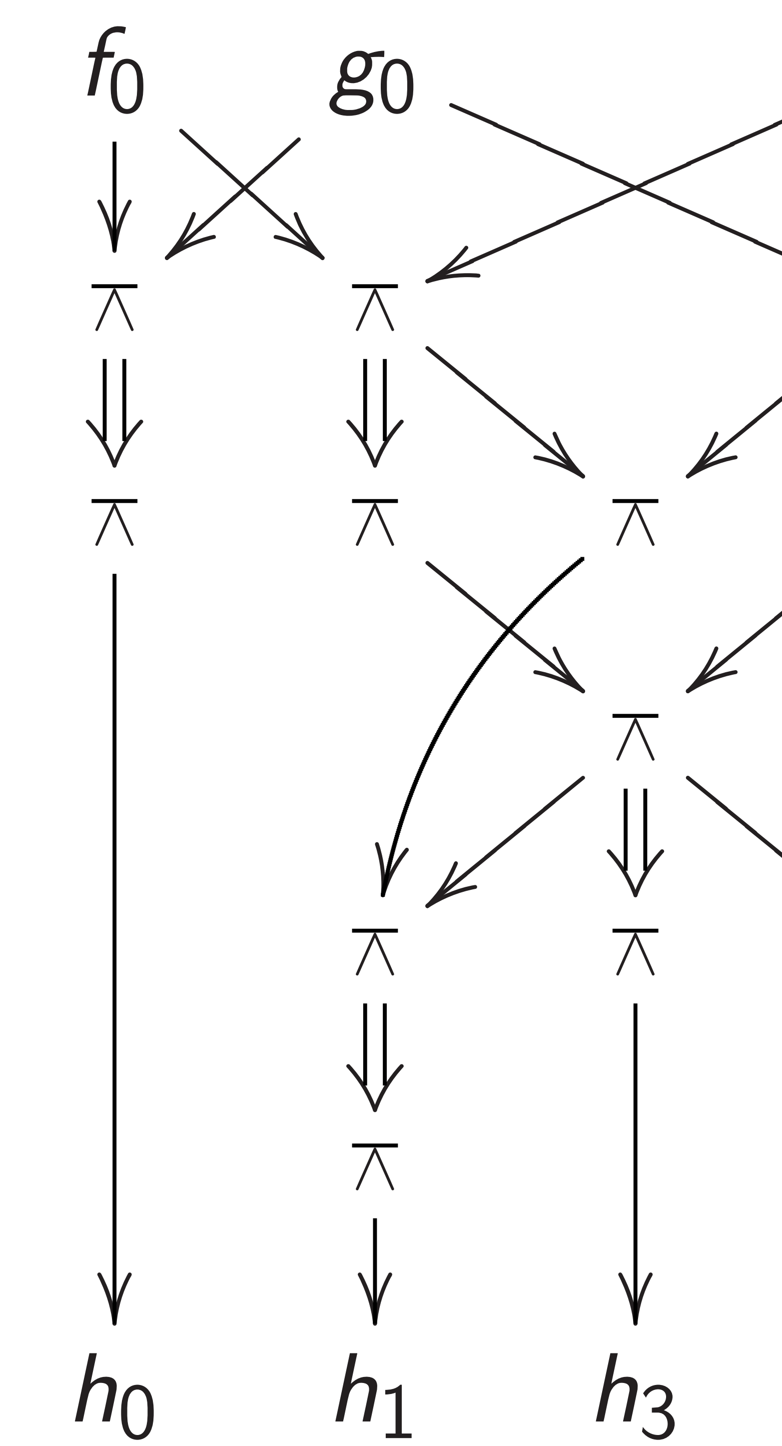
Minor optimization challenges:

- 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 $f_0 + 2f$

rds

ing.

k?

n.



## The big picture

CPUs are evolving
farther and farther away
from naive models of CPUs.

Minor optimization challenges:

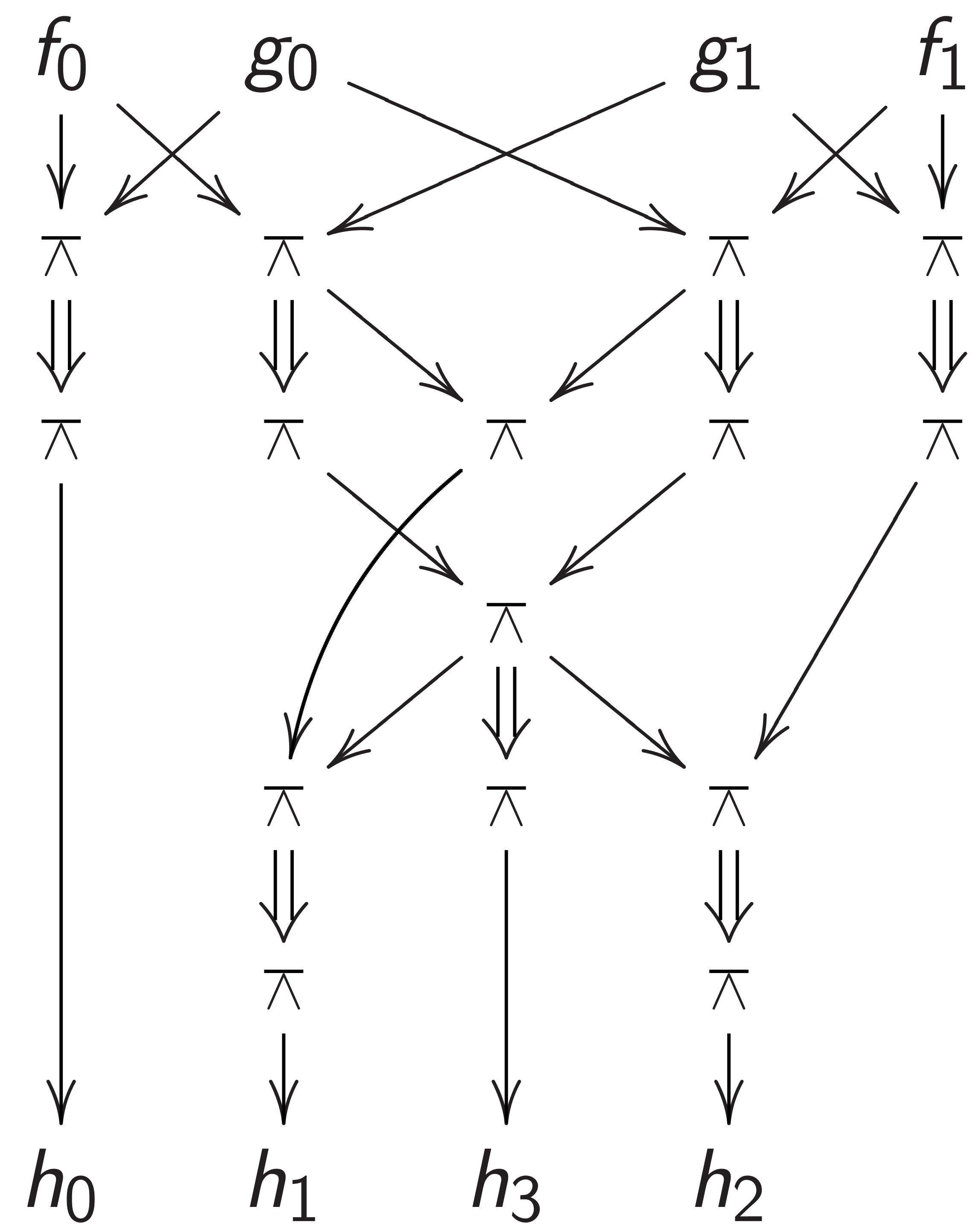
- Pipelining.
- Superscalar processing.

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 $f_0 + 2f_1, g_0 + 2g$

## The big picture

CPUs are evolving
farther and farther away
from naive models of CPUs.

Minor optimization challenges:

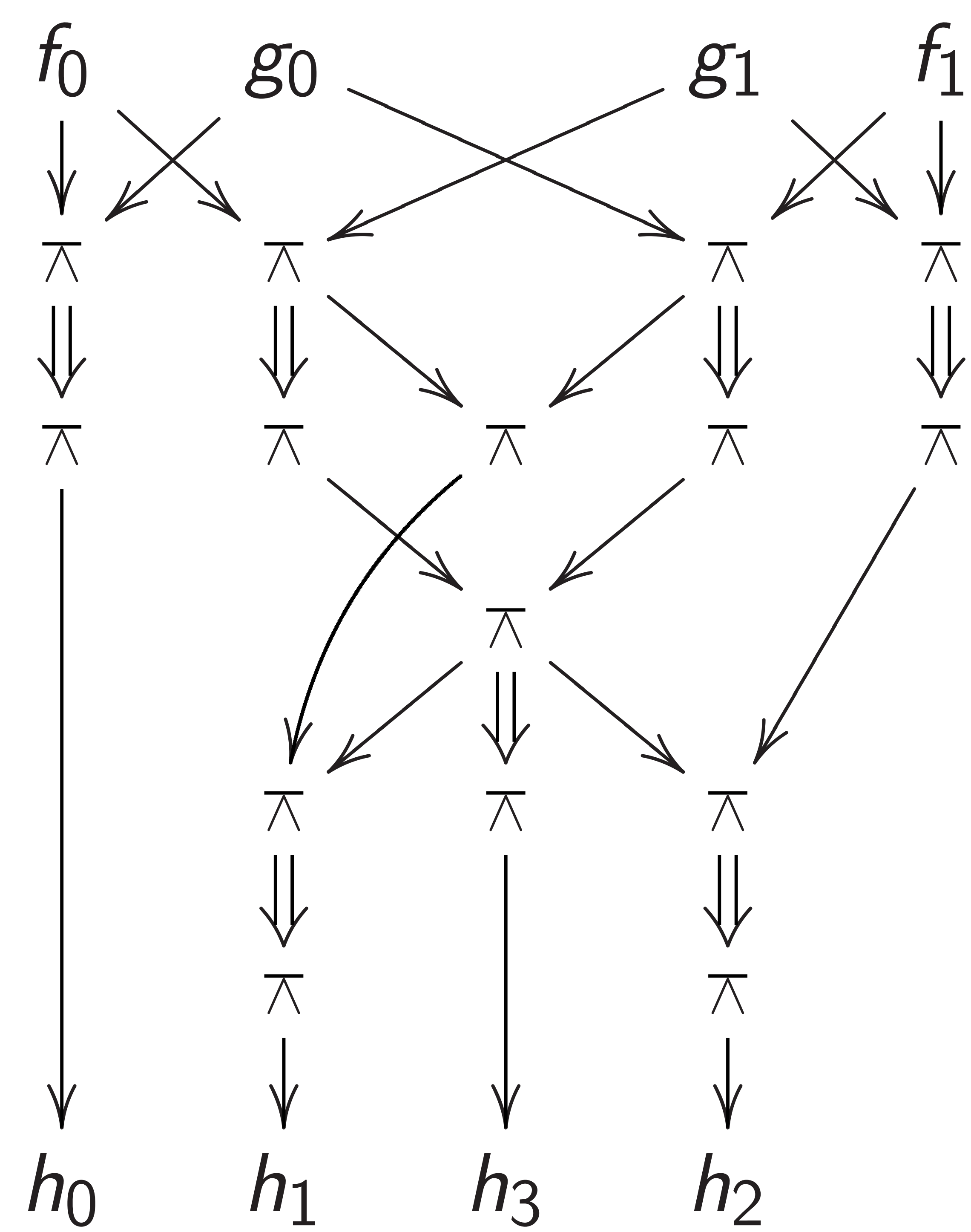
- Pipelining.
- Superscalar processing.

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 $f_0 + 2f_1, g_0 + 2g_1$.

picture

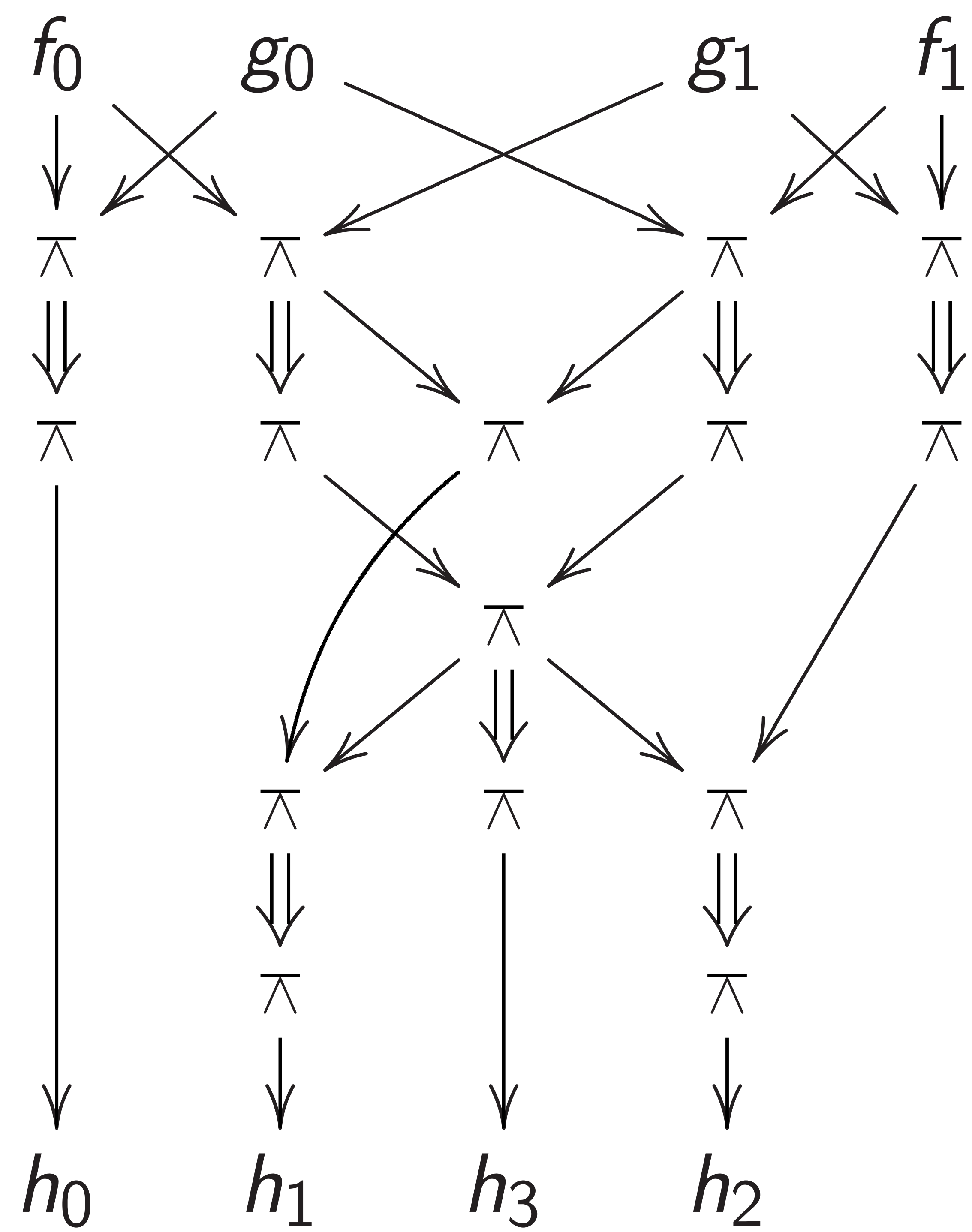
re evolving
and farther away
ive models of CPUs.

ptimization challenges:
ning.
scalar processing.

ptimization challenges:
rization.
threads; many cores.
memory 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

r away
s of CPUs.

n challenges:

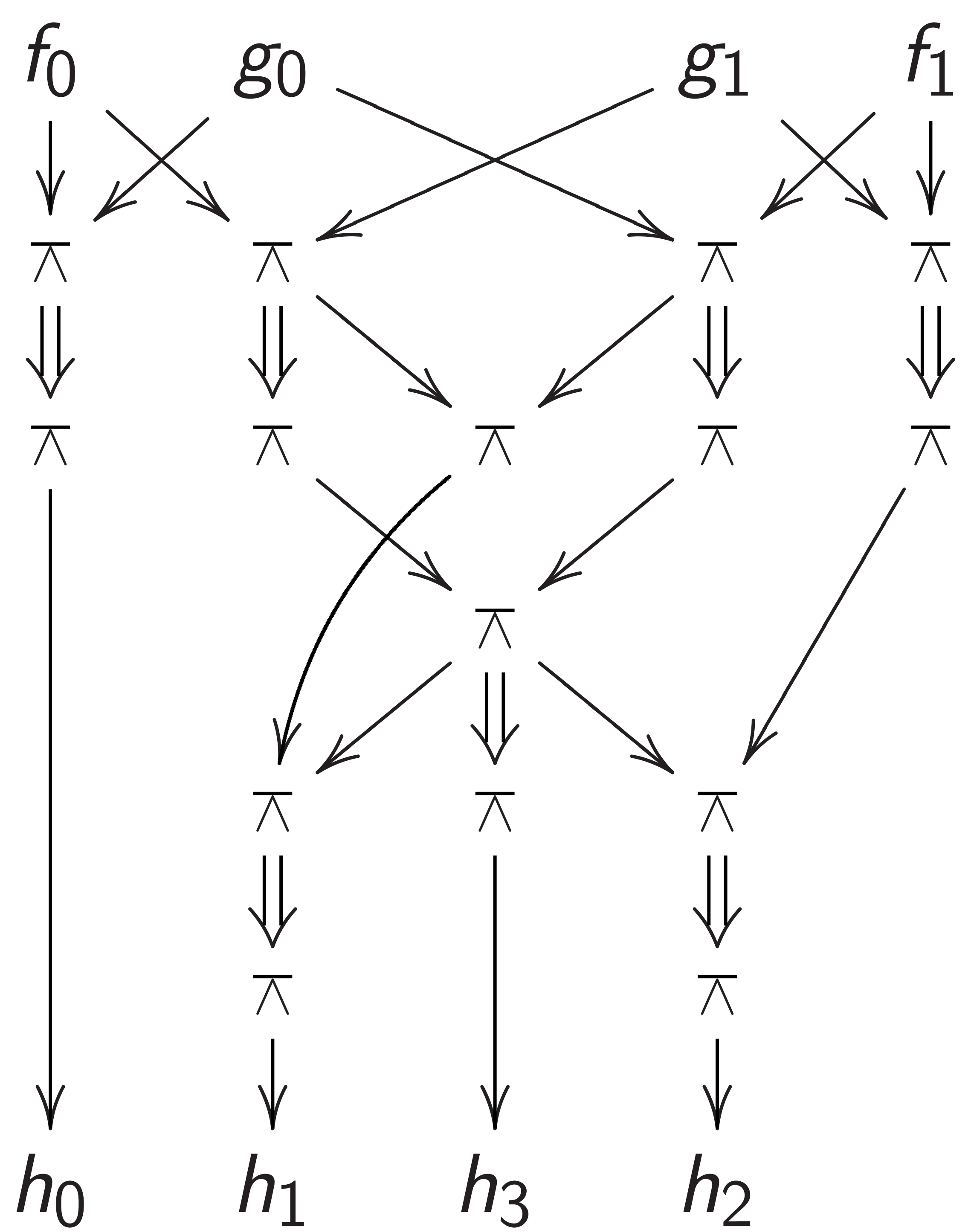
cessing.

n challenges:

nany 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

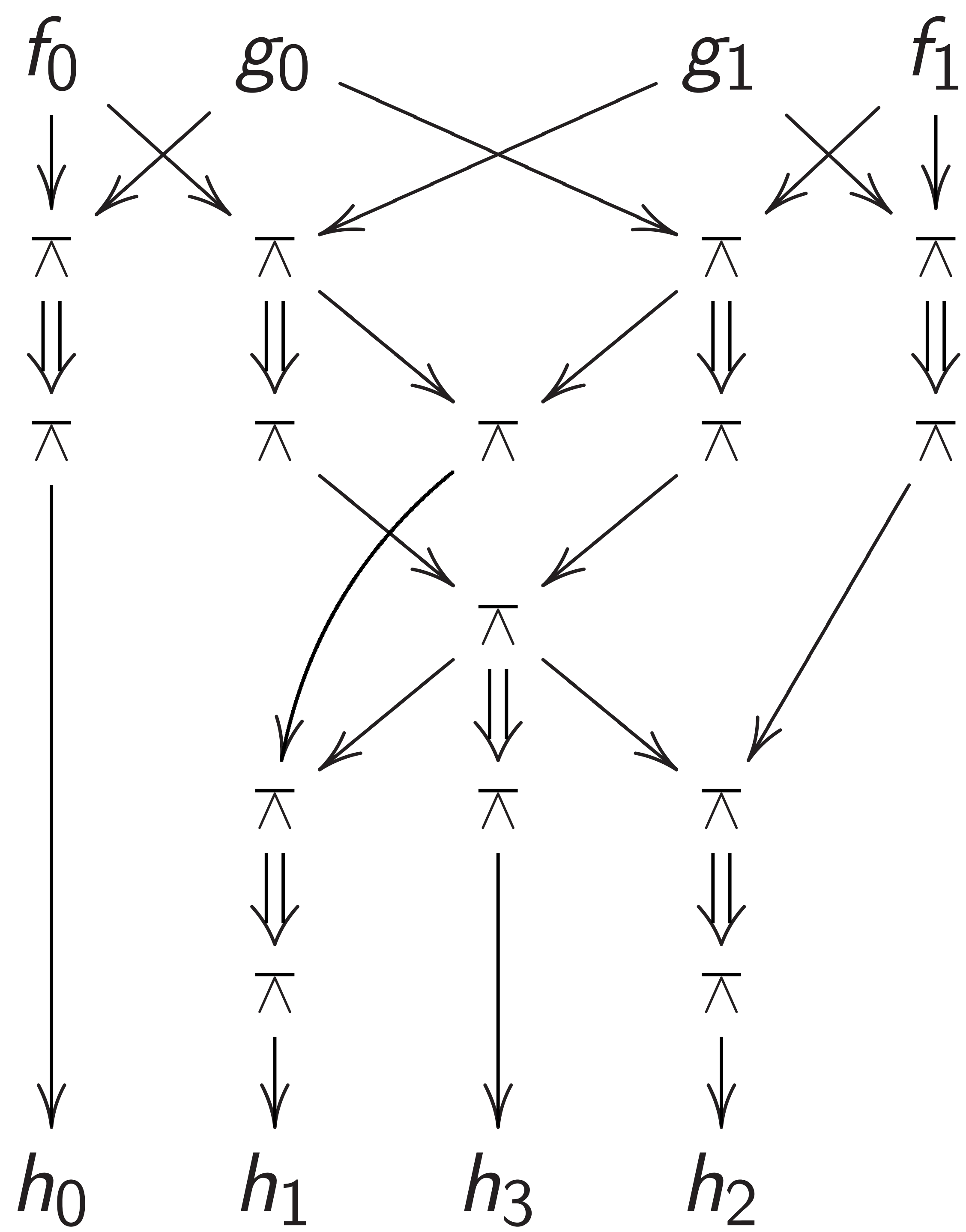
es:

es:

s.



CPU design in a nutshell

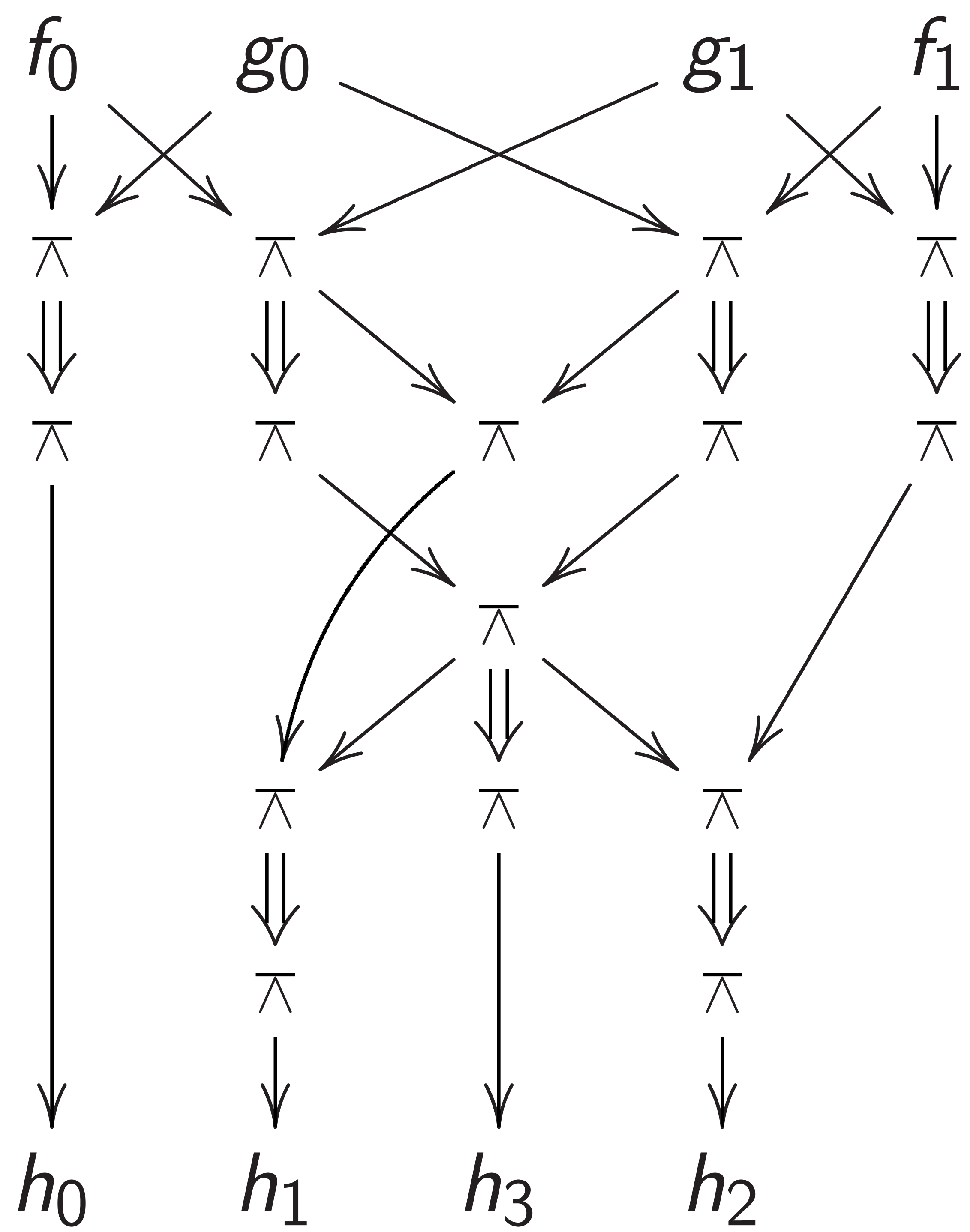



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.



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

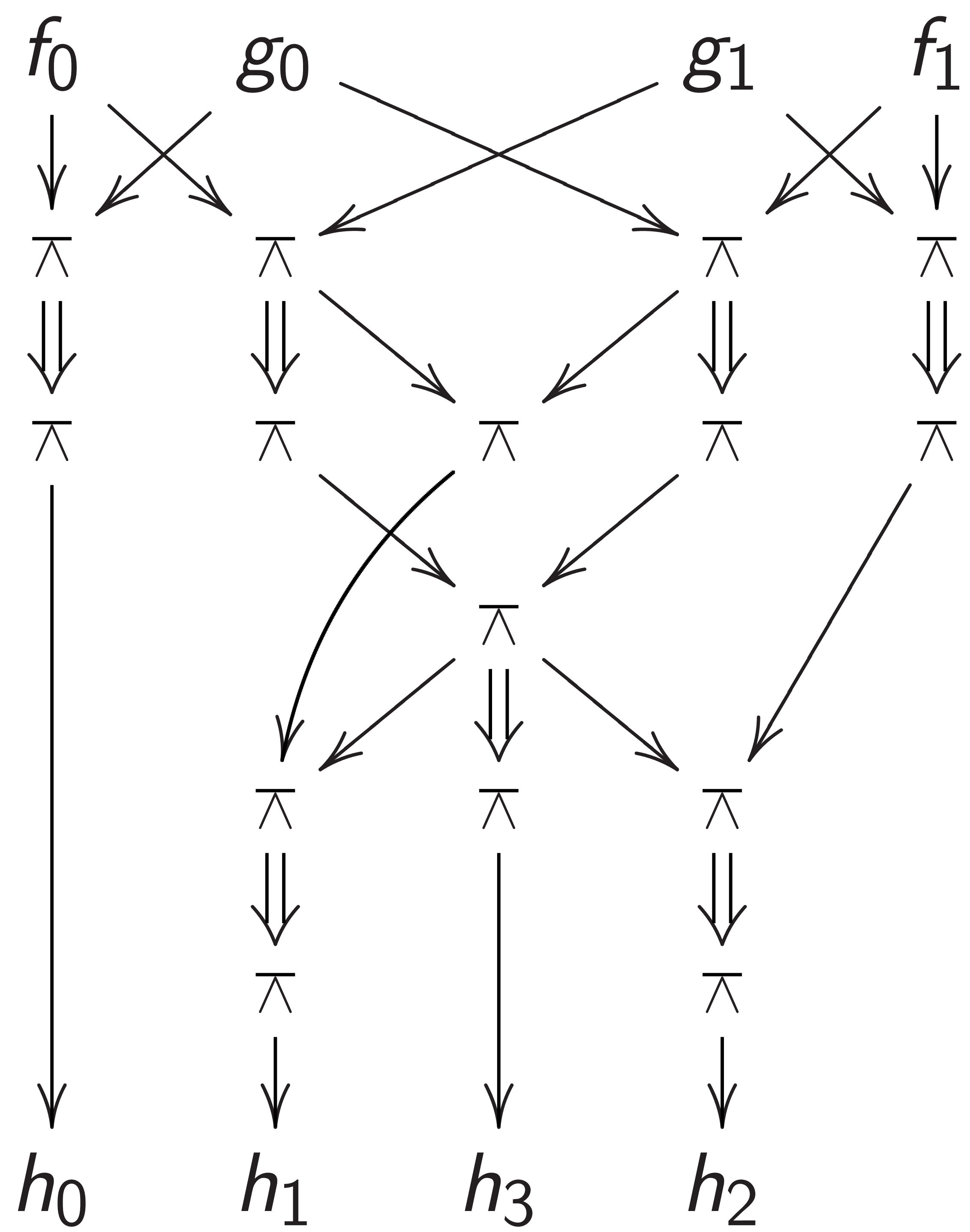



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.



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

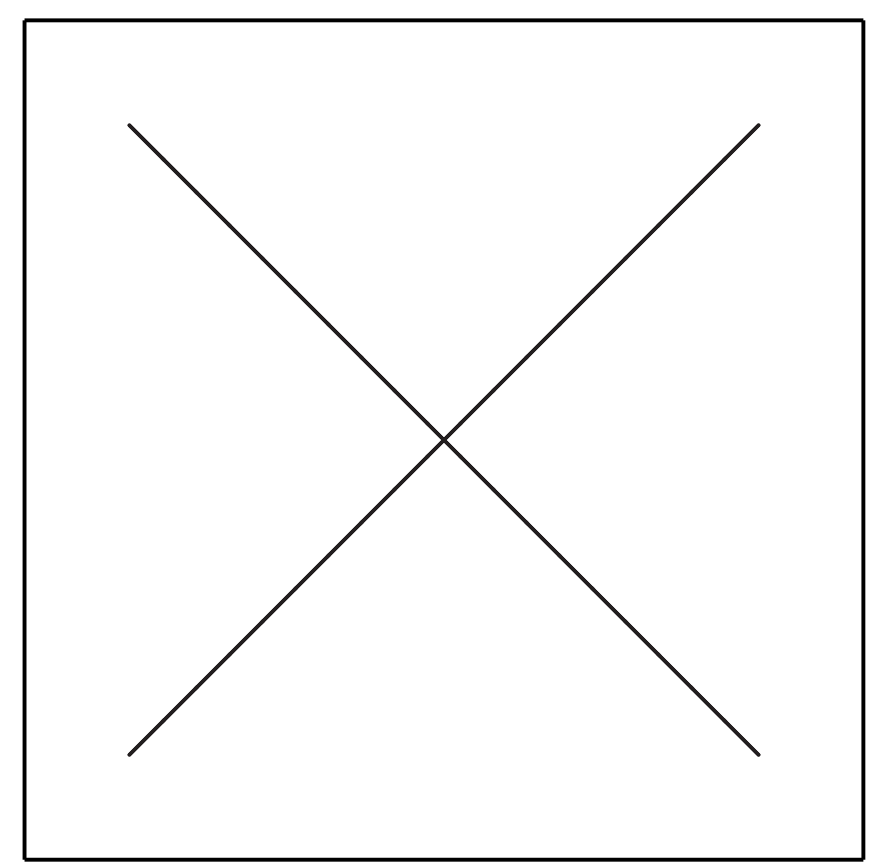



Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.



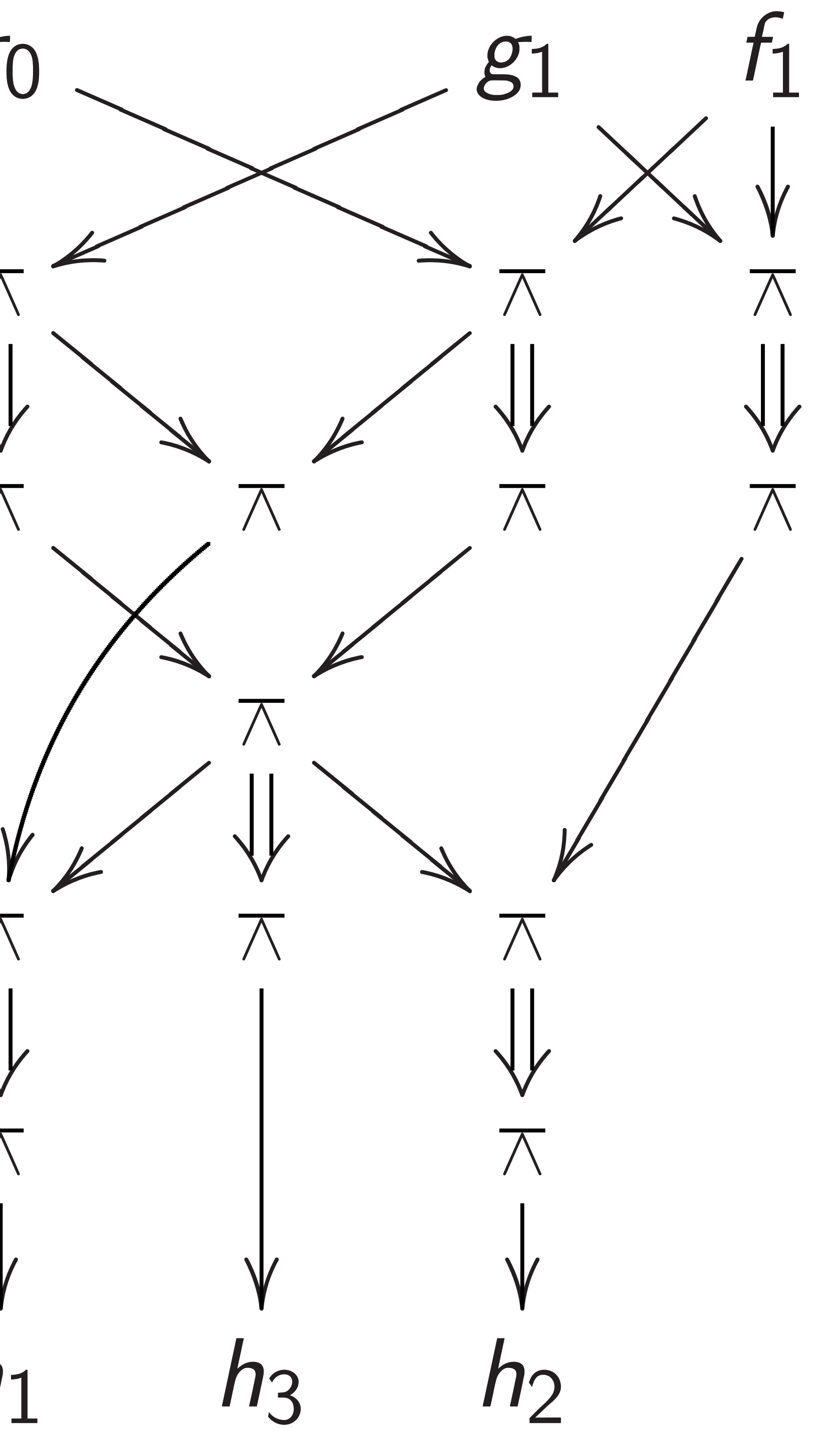
Electricity takes time to percolate through wires and gates. If $f_0, f_1, g_0, g_1$ are stable then $h_0, h_1, h_2, h_3$ are stable a few moments later.

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



(Details omitted.)

sign in a nutshell

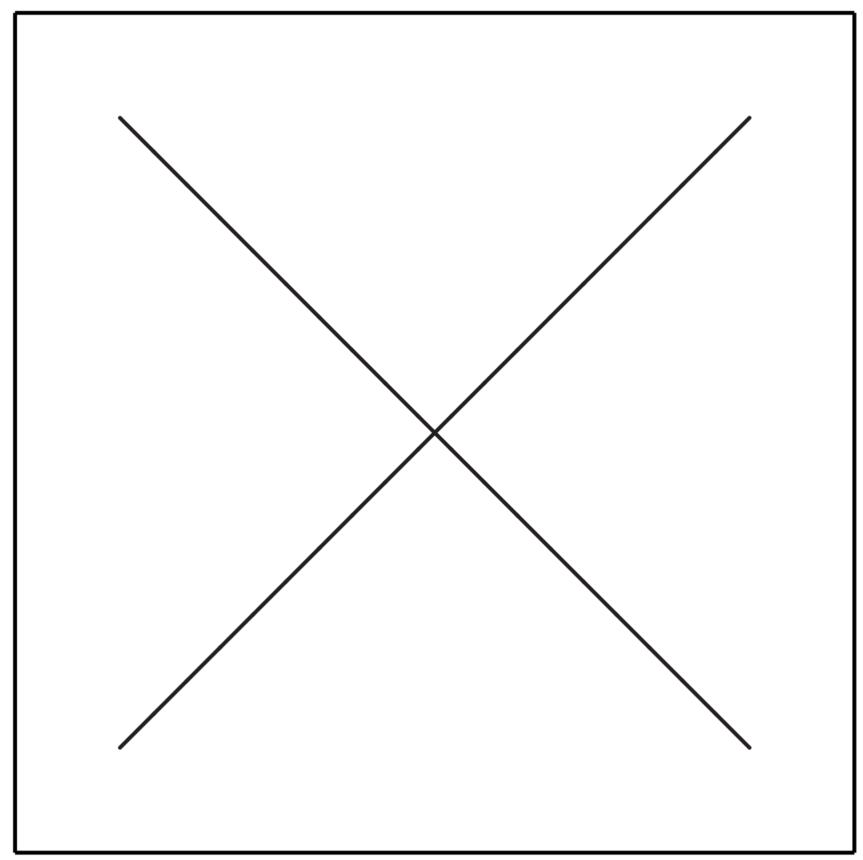



: $a, b \mapsto 1 - ab$ computing
$h_0 + 2h_1 + 4h_2 + 8h_3$
ers $f_0 + 2f_1, g_0 + 2g_1$.



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:

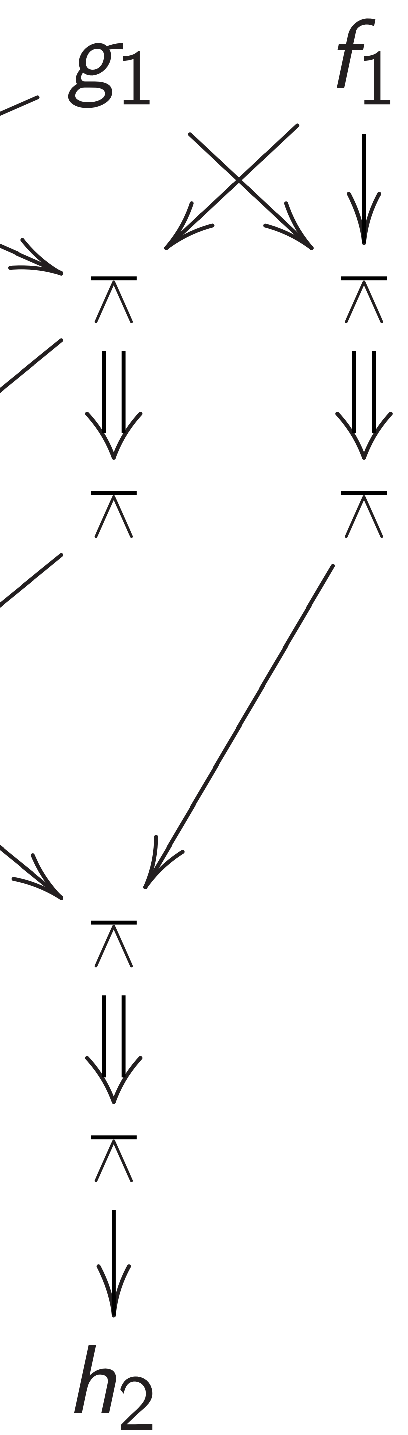


(Details omitted.)

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

reg
re

utshell




$-\ ab$ computing

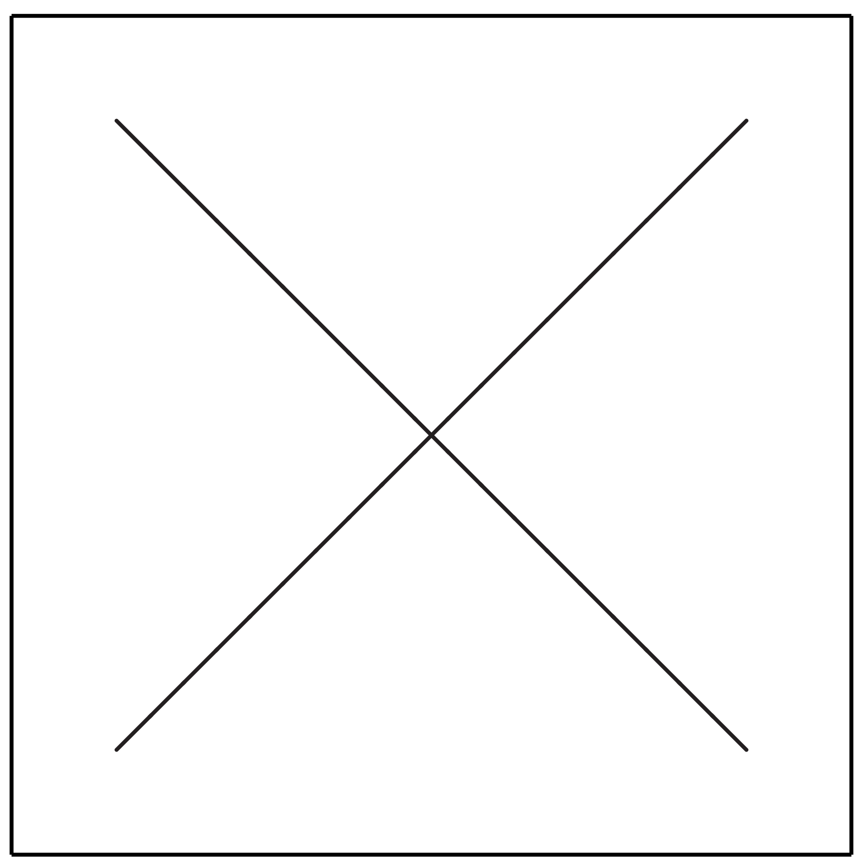
$+\ 4h_2 + 8h_3$

$1, g_0 + 2g_1$.



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

$h_3$

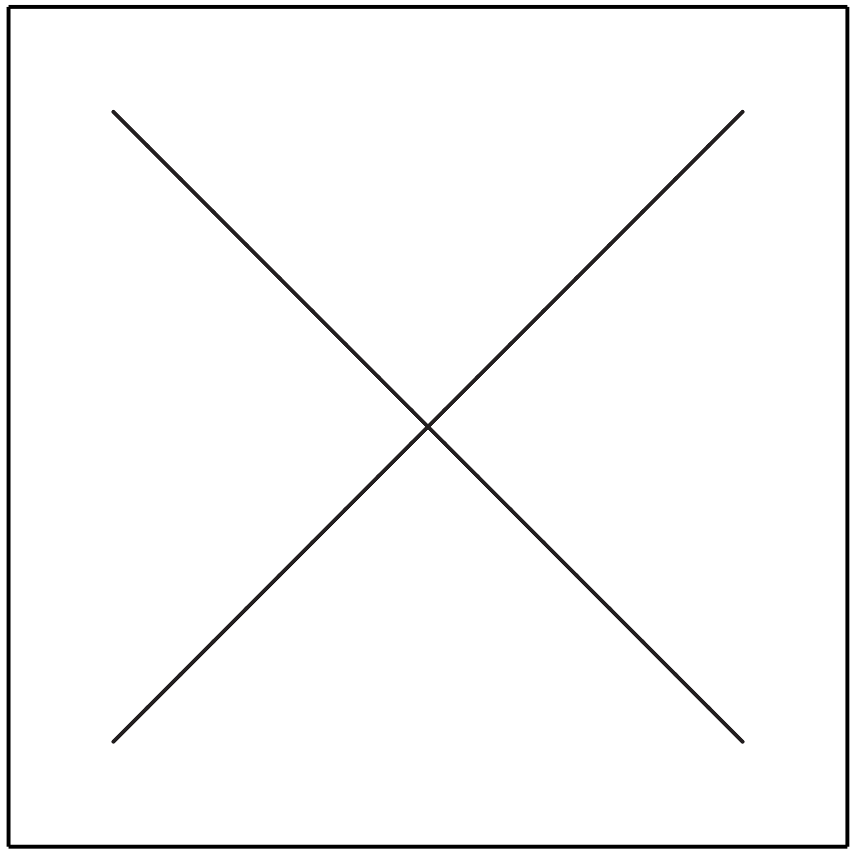
$_1$.



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.)

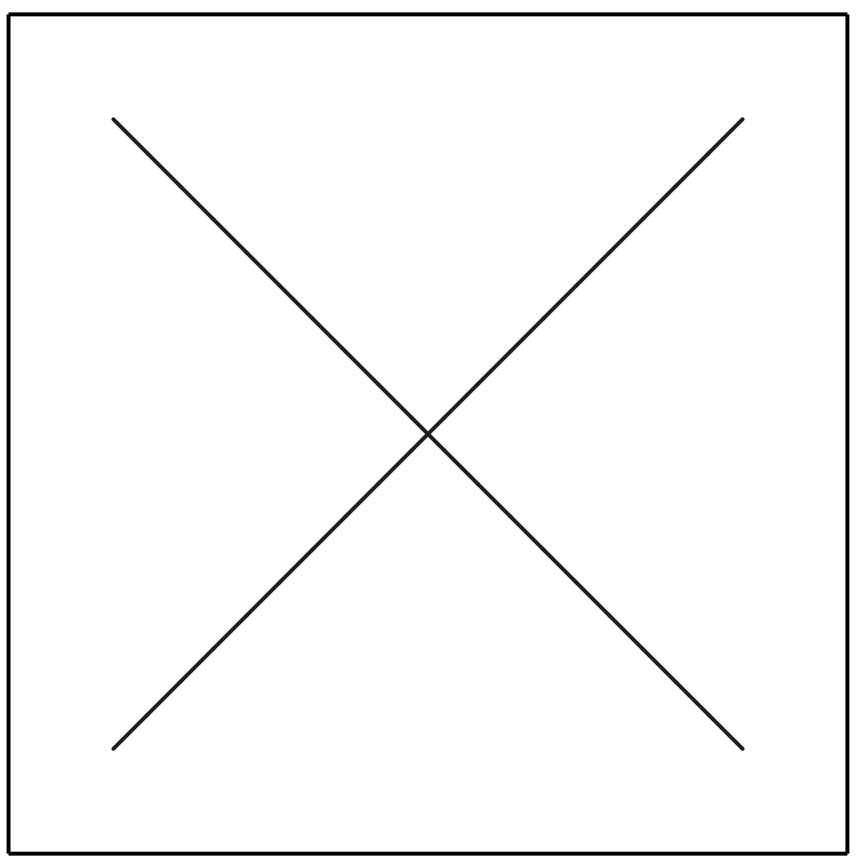


Build circuit to compute 32-bit integer $r_i$ given 4-bit integer $i$ and 32-bit integers $r_0, r_1, \ldots$

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.)

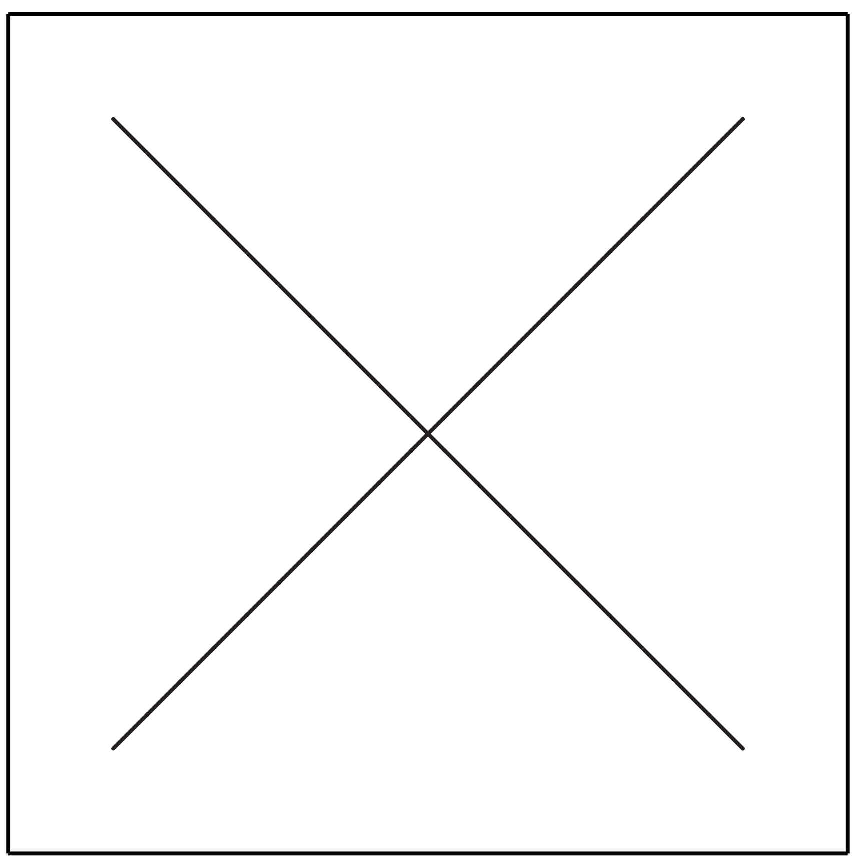


Build circuit to compute 32-bit integer $r_i$ given 4-bit integer $i$ and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

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

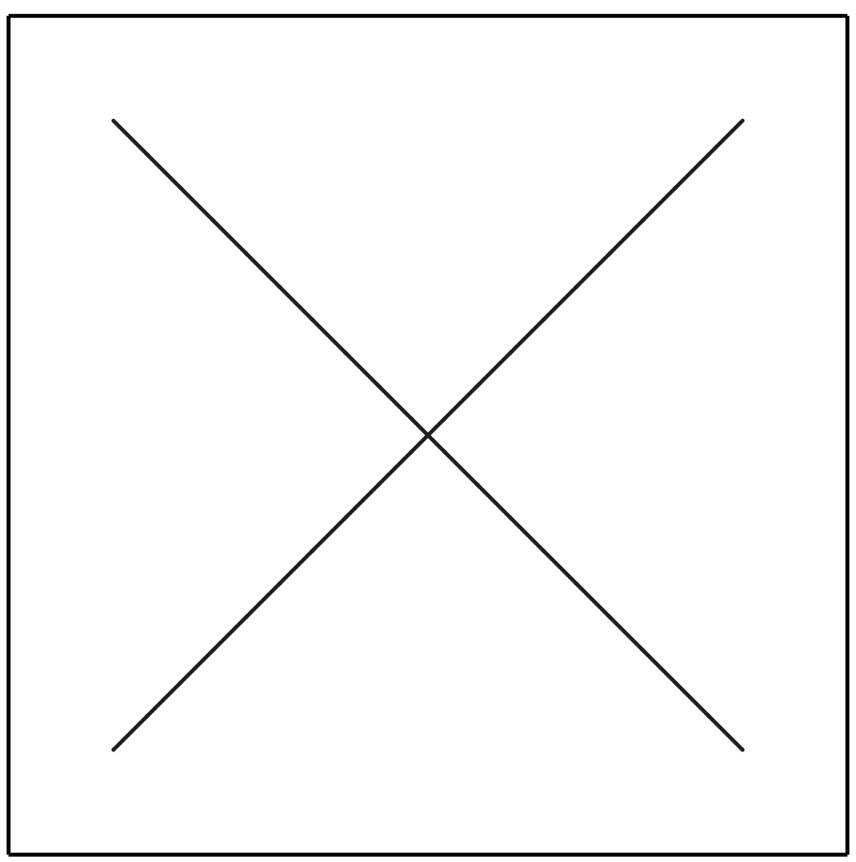
Build circuit for "register write":

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

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.)

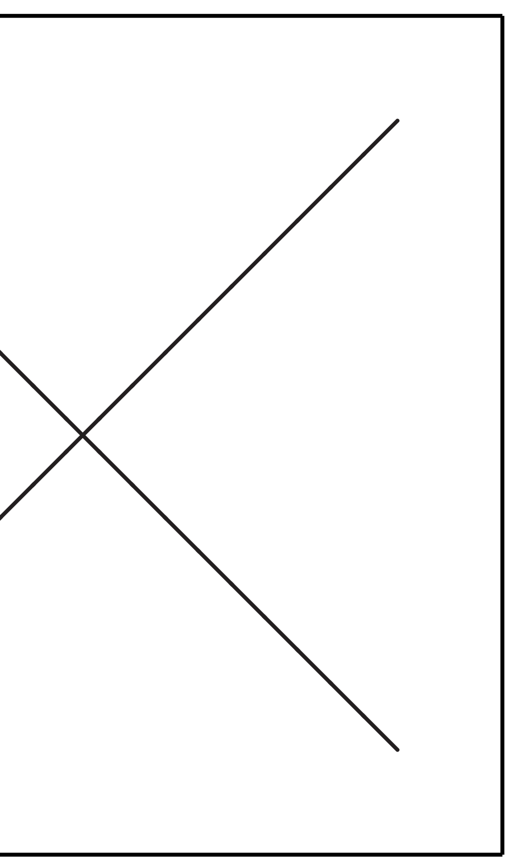


Build circuit to compute 32-bit integer $r_i$ given 4-bit integer $i$ and 32-bit integers $r_0, r_1, \ldots, r_{15}$:


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.)

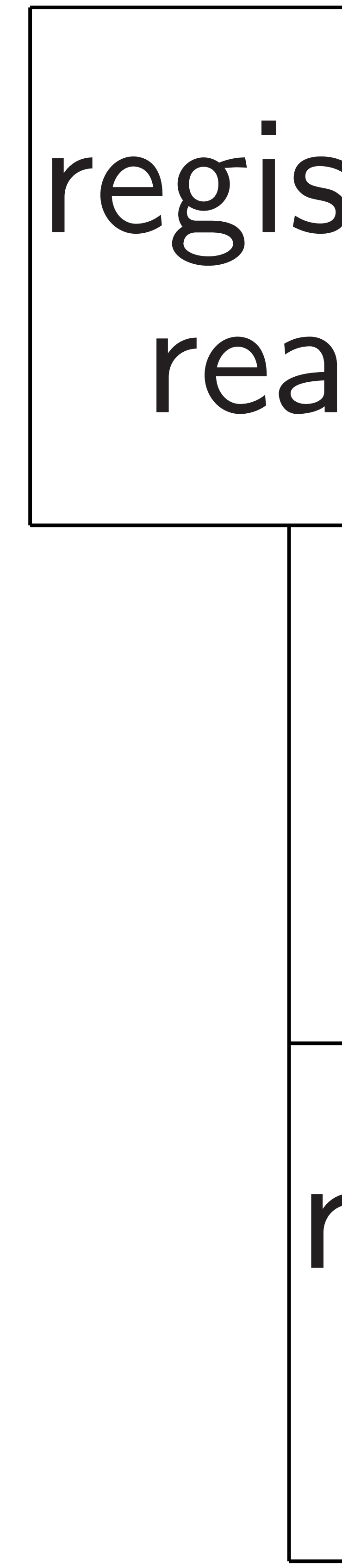


Build circuit to compute
32-bit integer $r_i$
given 4-bit integer $i$
and 32-bit integers $r_0, r_1, \ldots, r_{15}$:

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 $r'_\ell$



me to
wires and gates.
stable
are stable
ter.

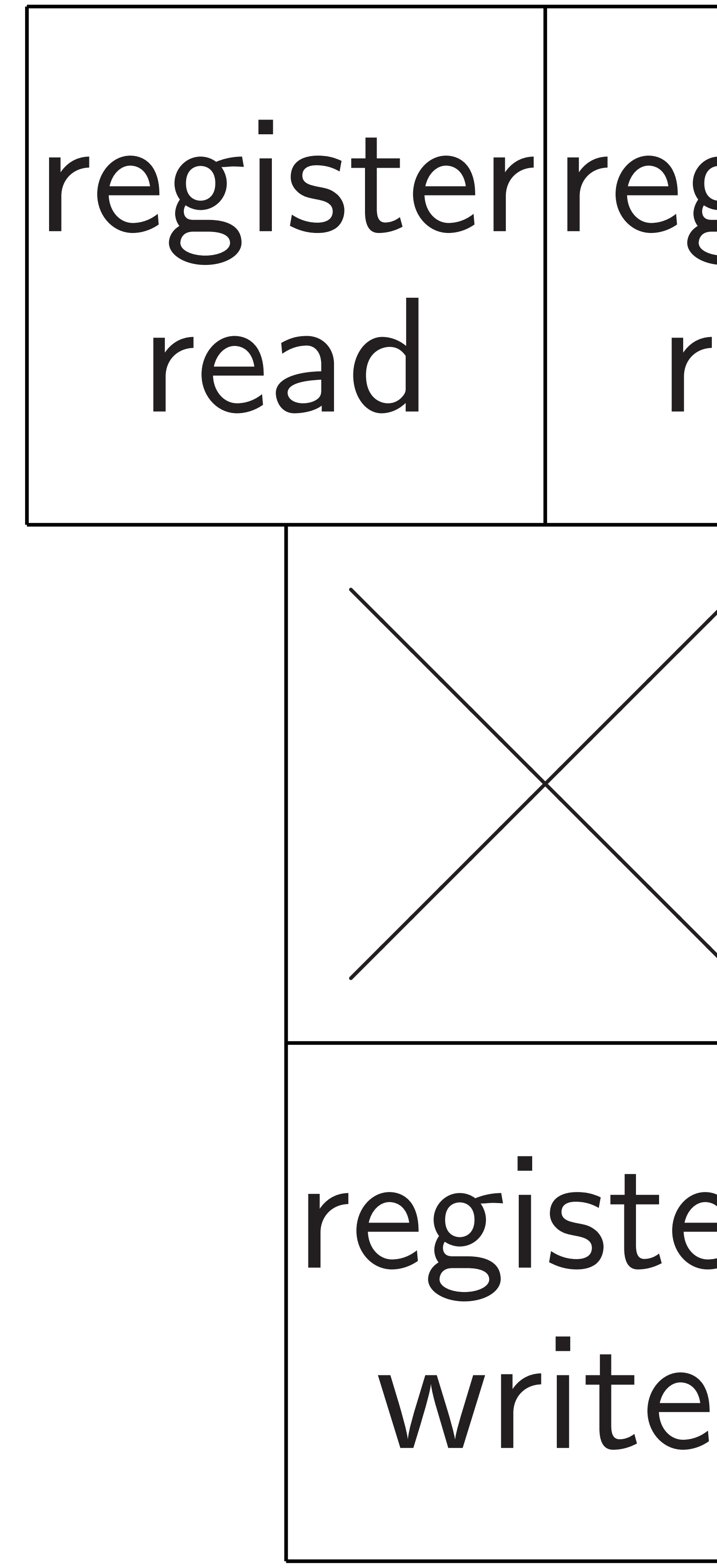
more gates
32-bit integers:



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.

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



gates.

s

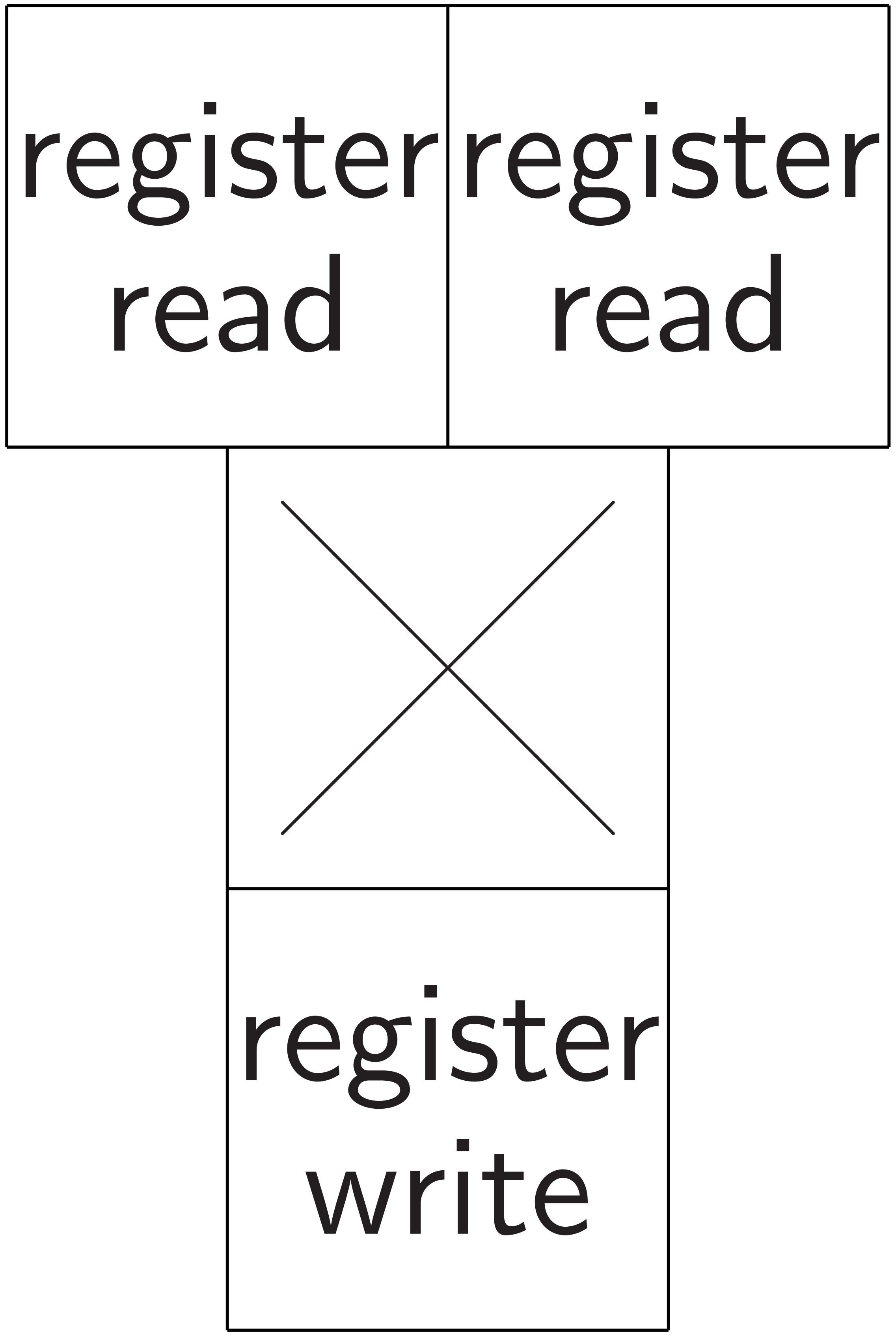
egers:



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 $r'_i = r_j$

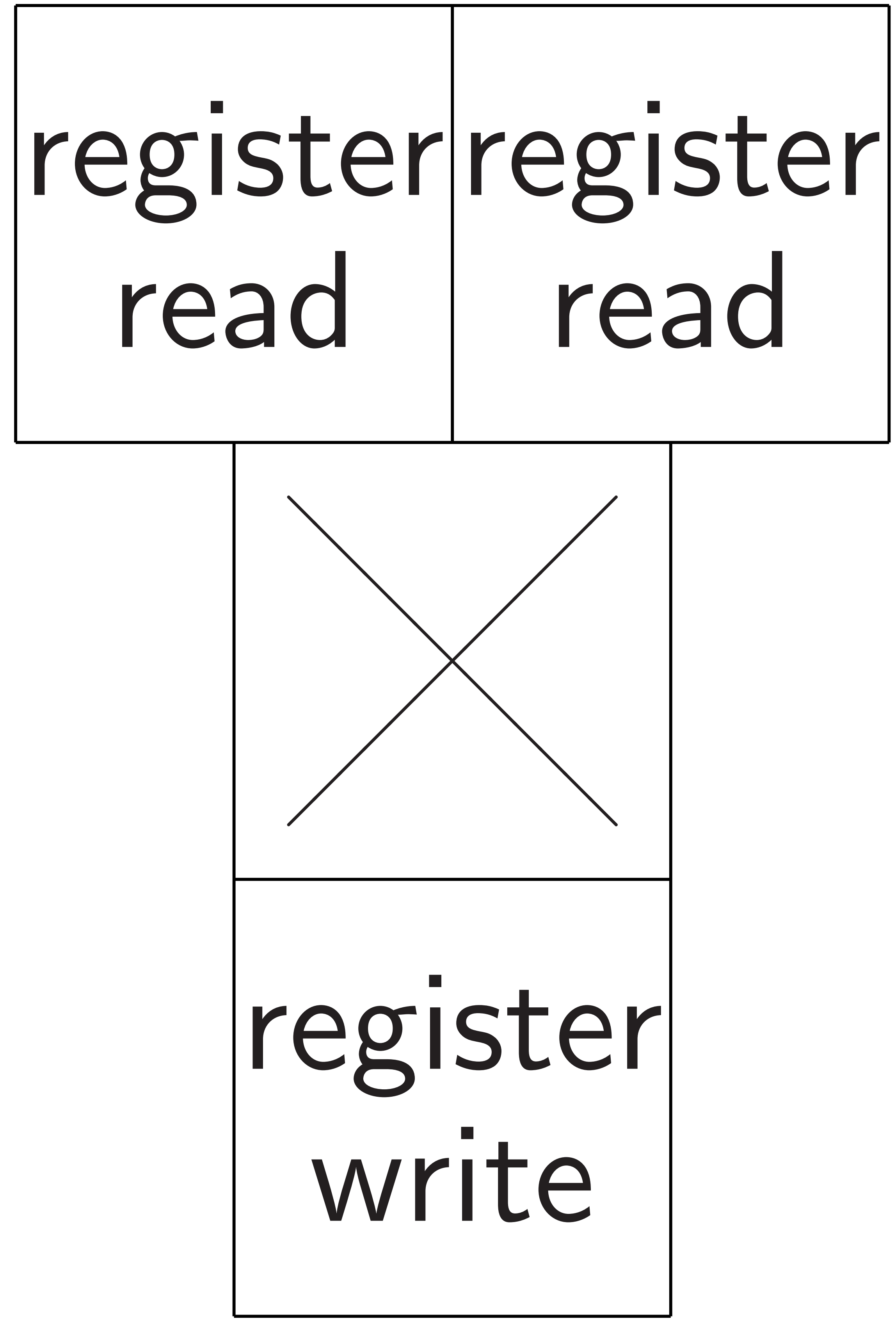


Build circuit to compute
32-bit integer $r_i$
given 4-bit integer $i$
and 32-bit integers $r_0, r_1, \ldots, r_{15}$:


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 $r_0, r_1, \dots, r_{15}$:


rcuit for “register write”:

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

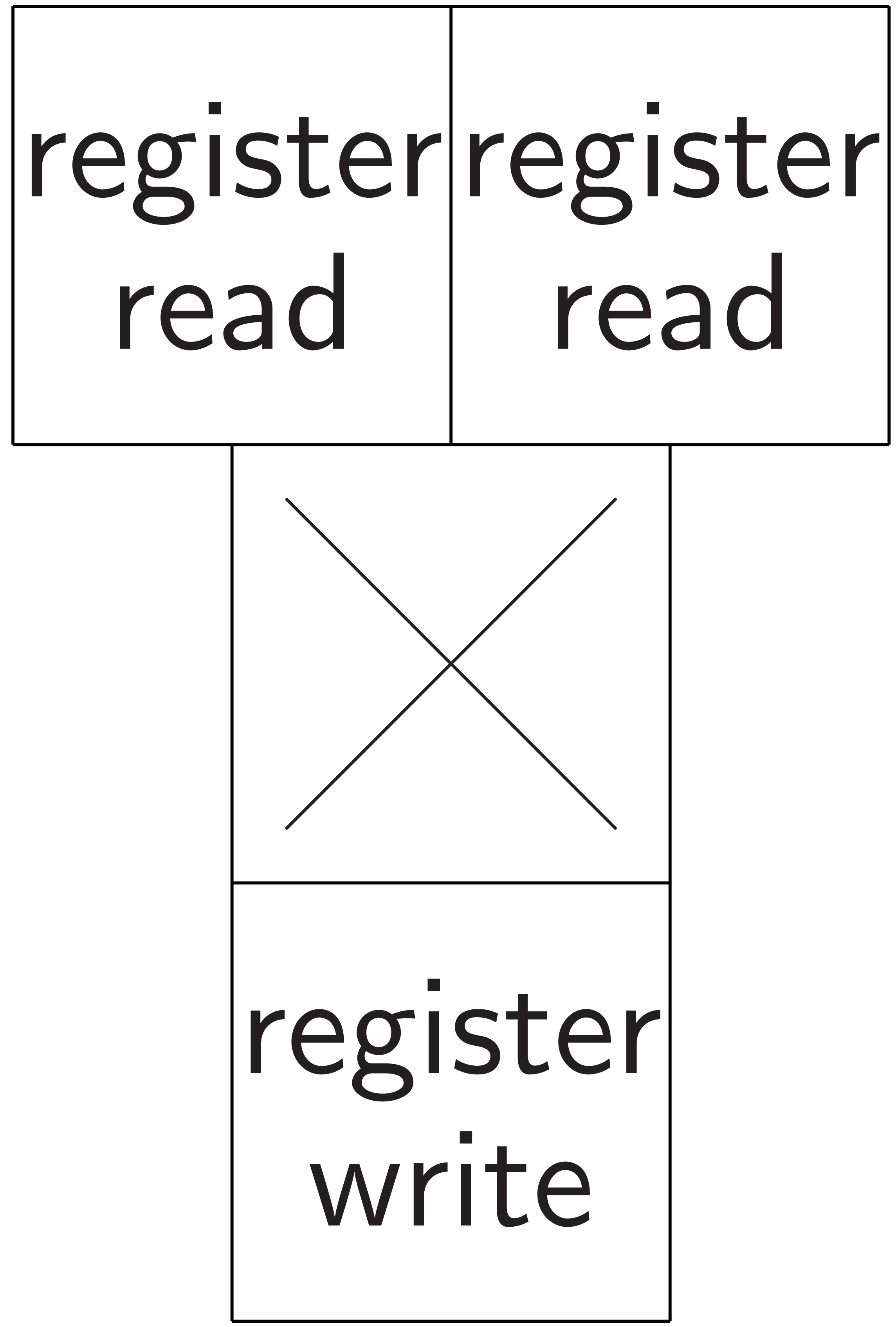
$= r_j$ except $r'_i = s$.

rcuit for addition. Etc.



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

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




Add mor

More ari

replace (

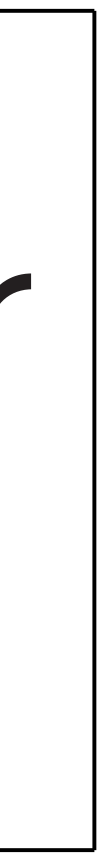
(“$\times$”, $i$,

(“$+$”, $i$,

mpute

$i$

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



register write":

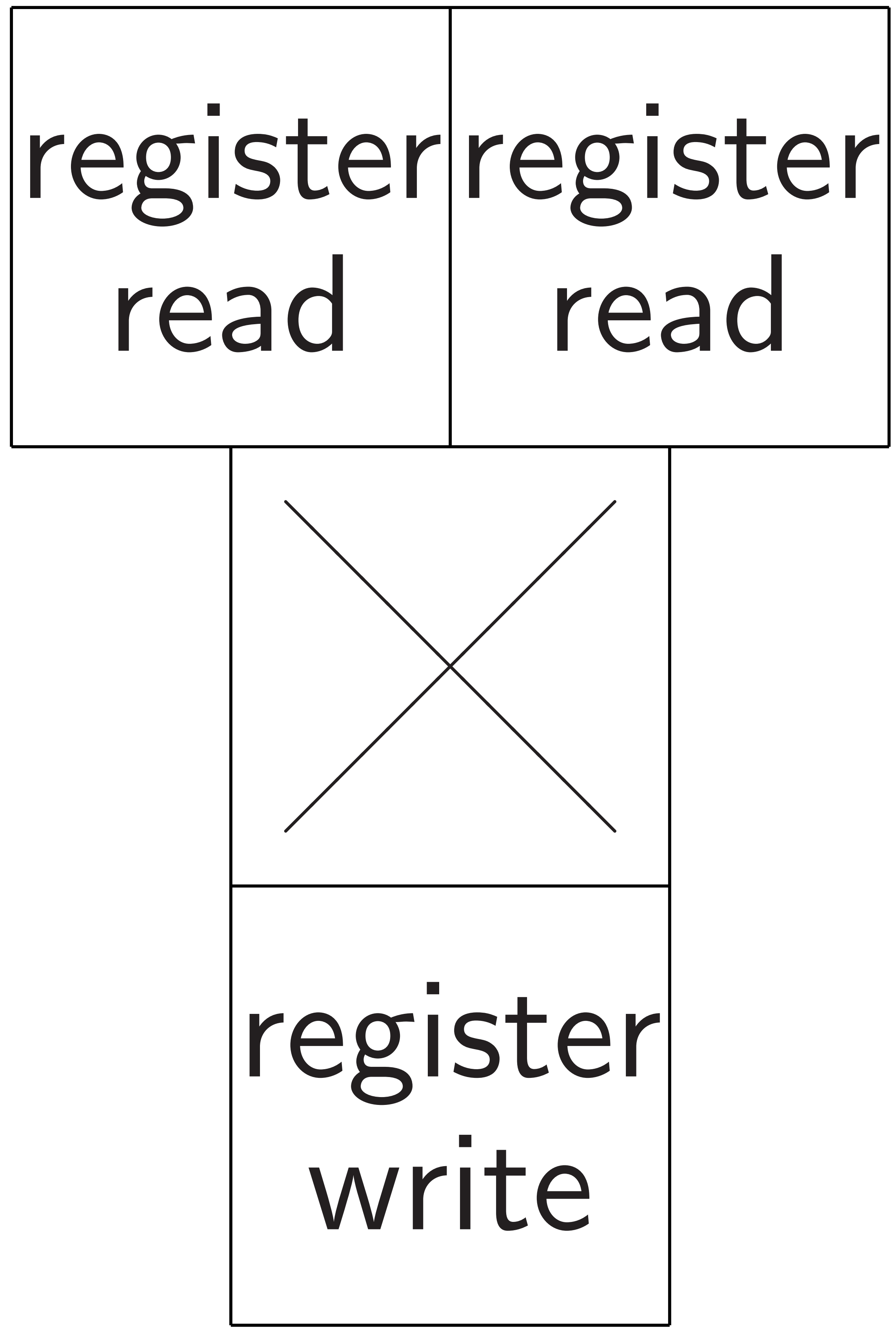
$r'_0, \ldots, r'_{15}$

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

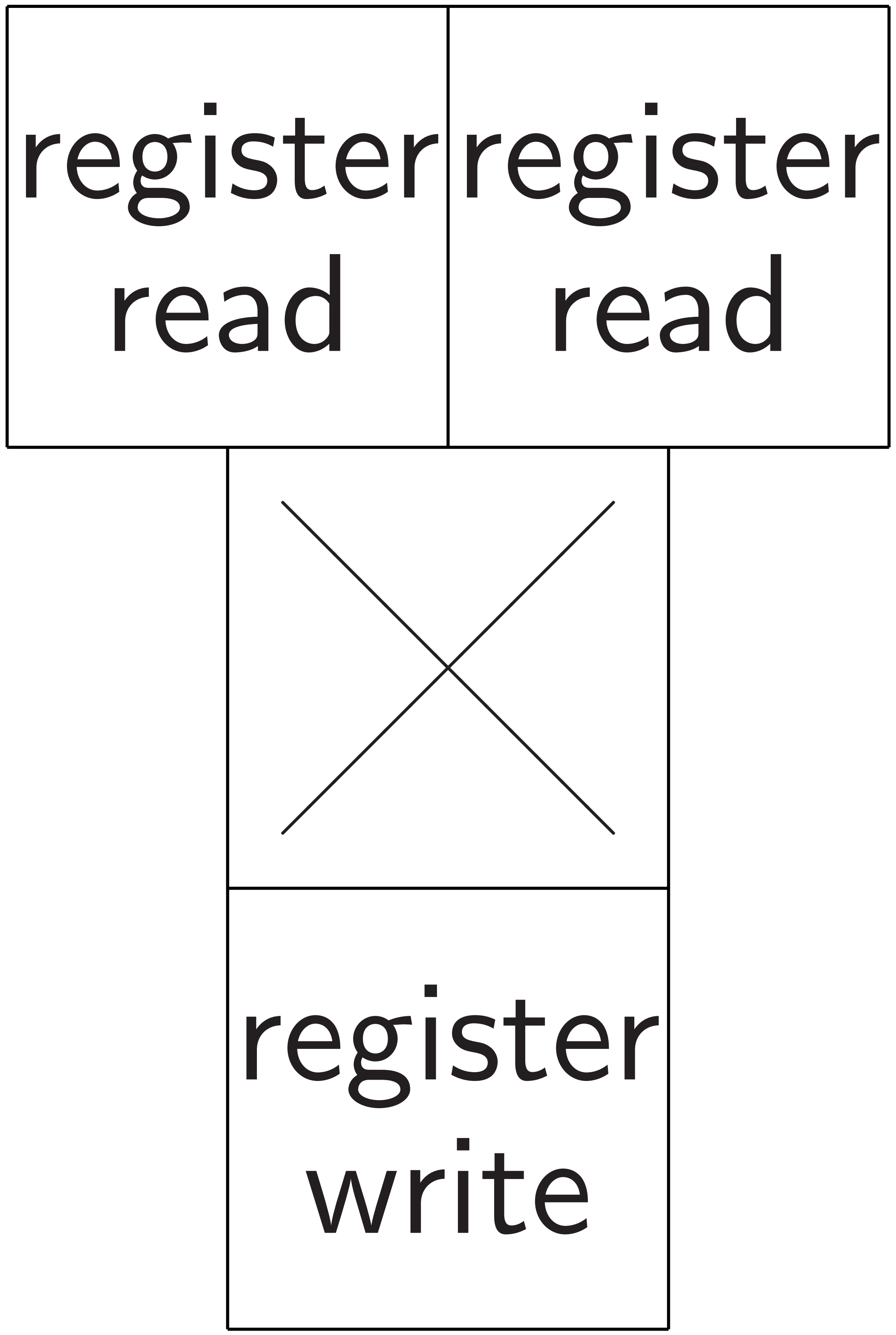
, $r_{15}$:

rite”:

tc.



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

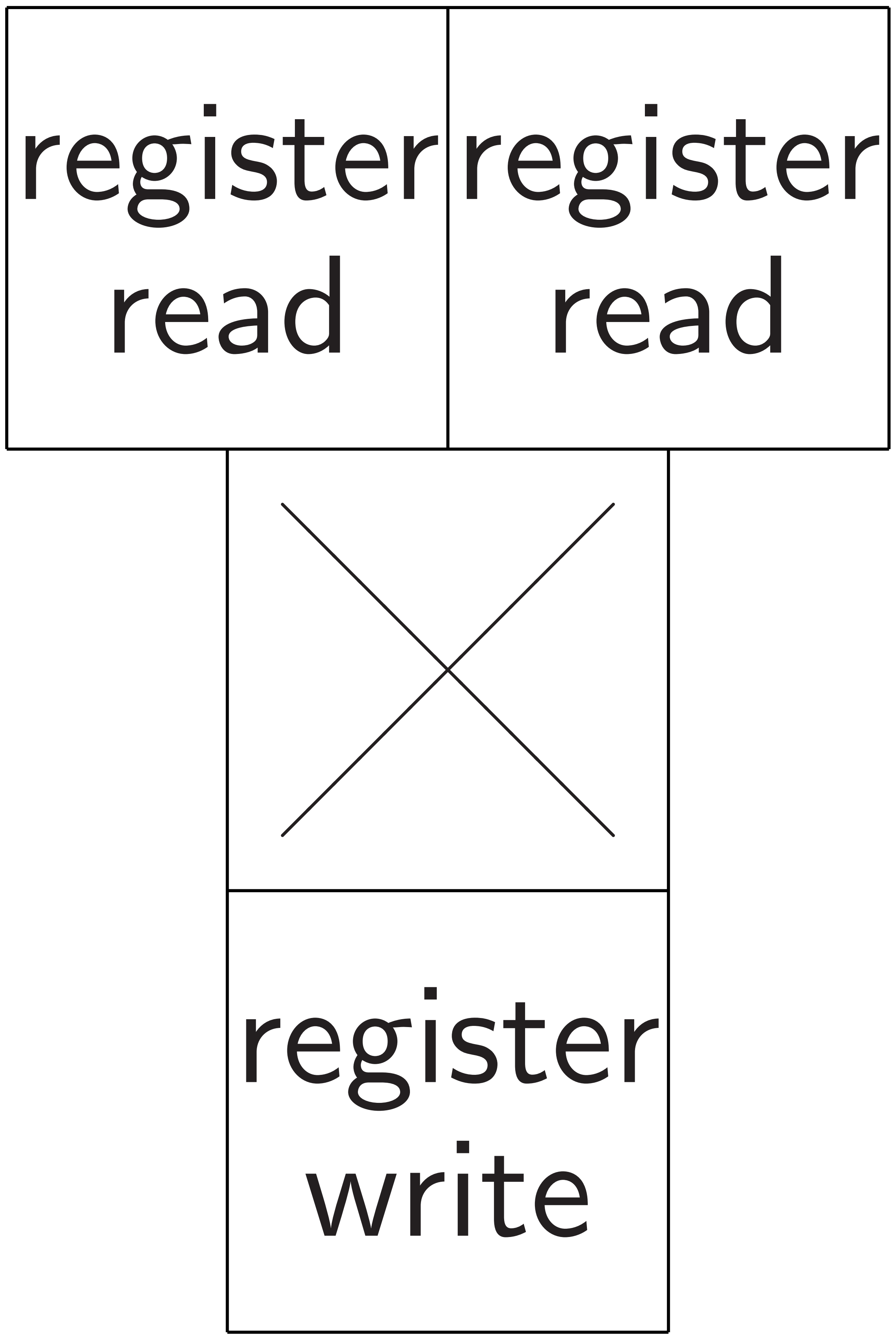





Add more flexibility.

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

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

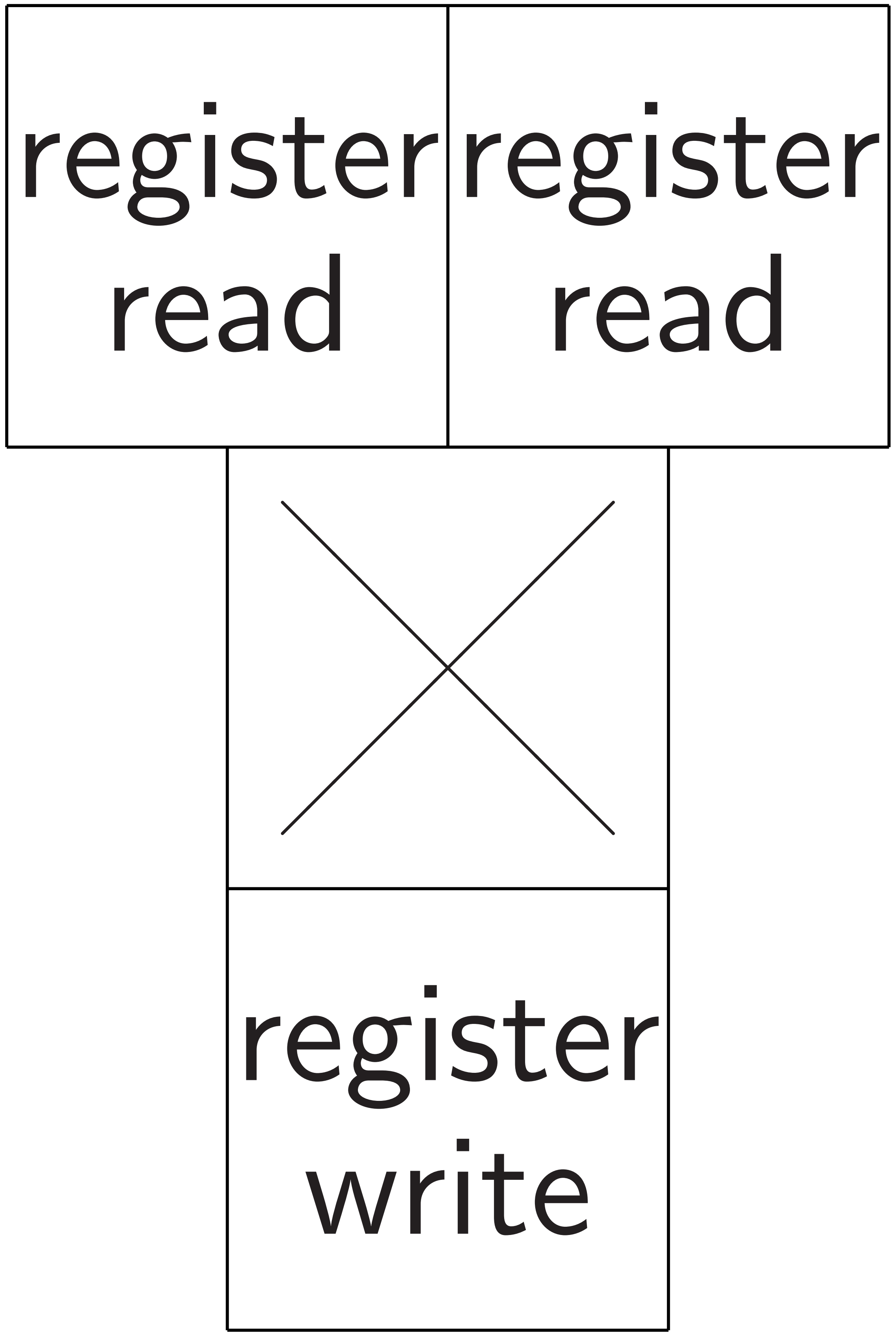





Add more flexibility.

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 $r'_i = r_j r_k$:



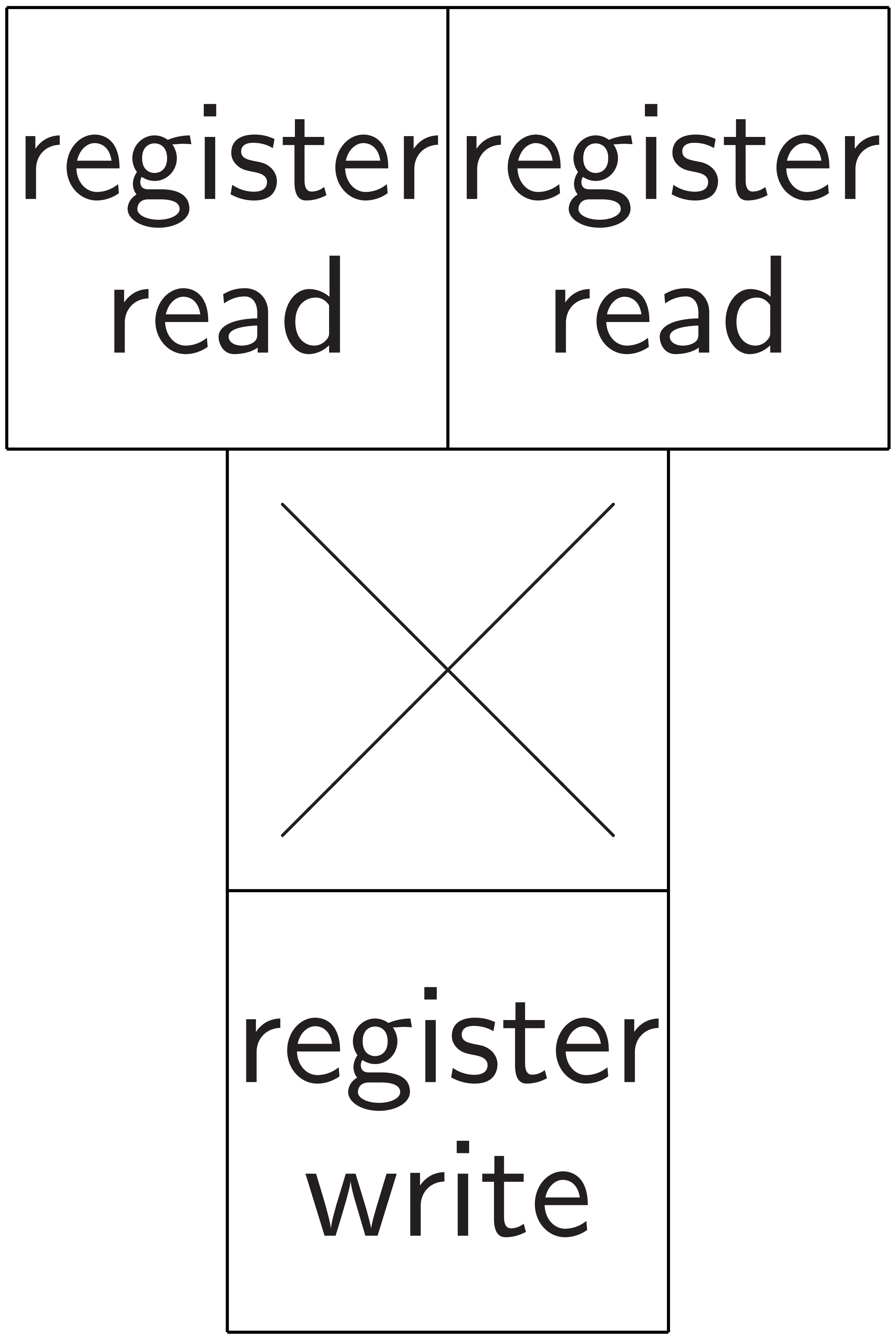



Add more flexibility.

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 $r'_i = r_j r_k$:





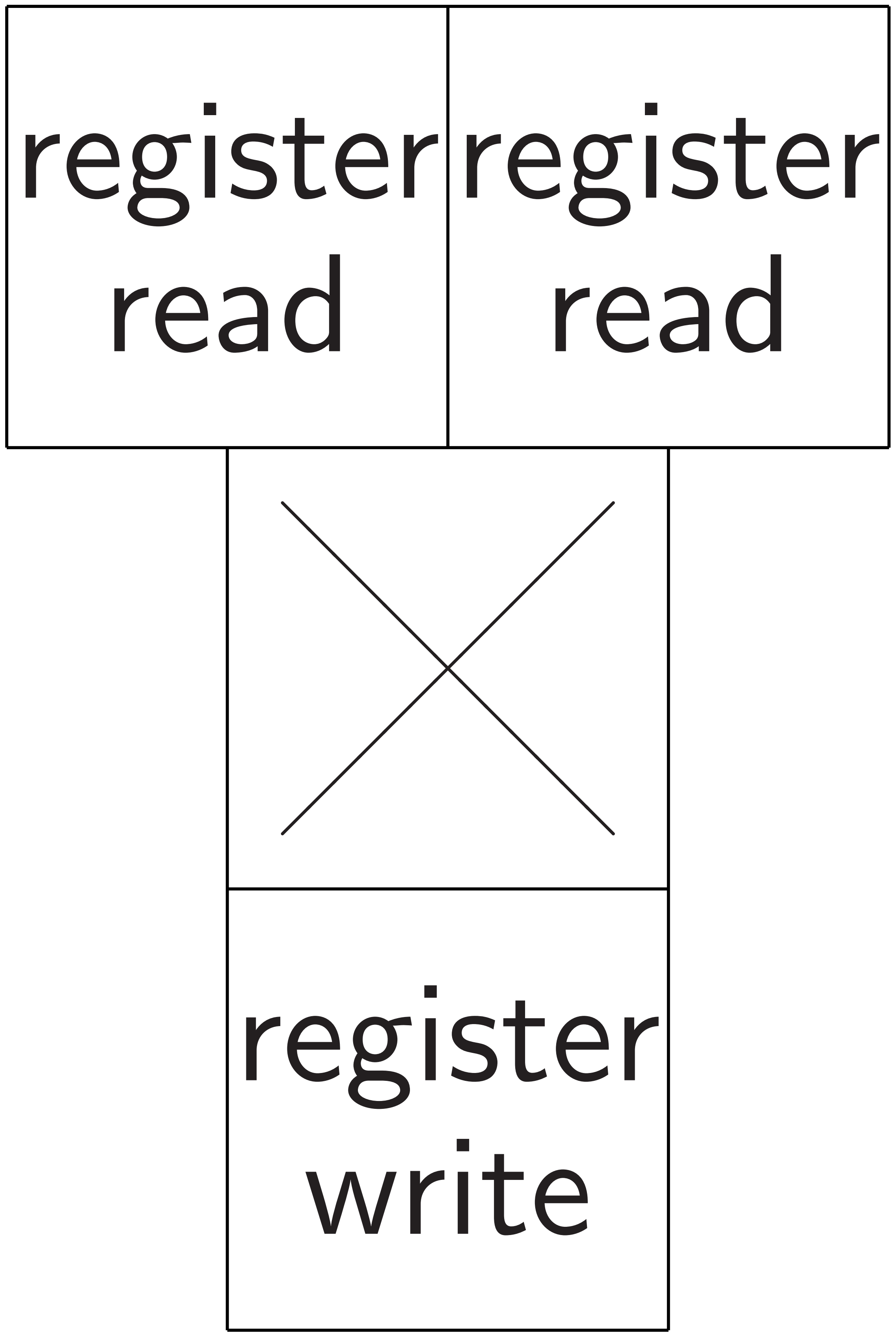

Add more flexibility.

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.

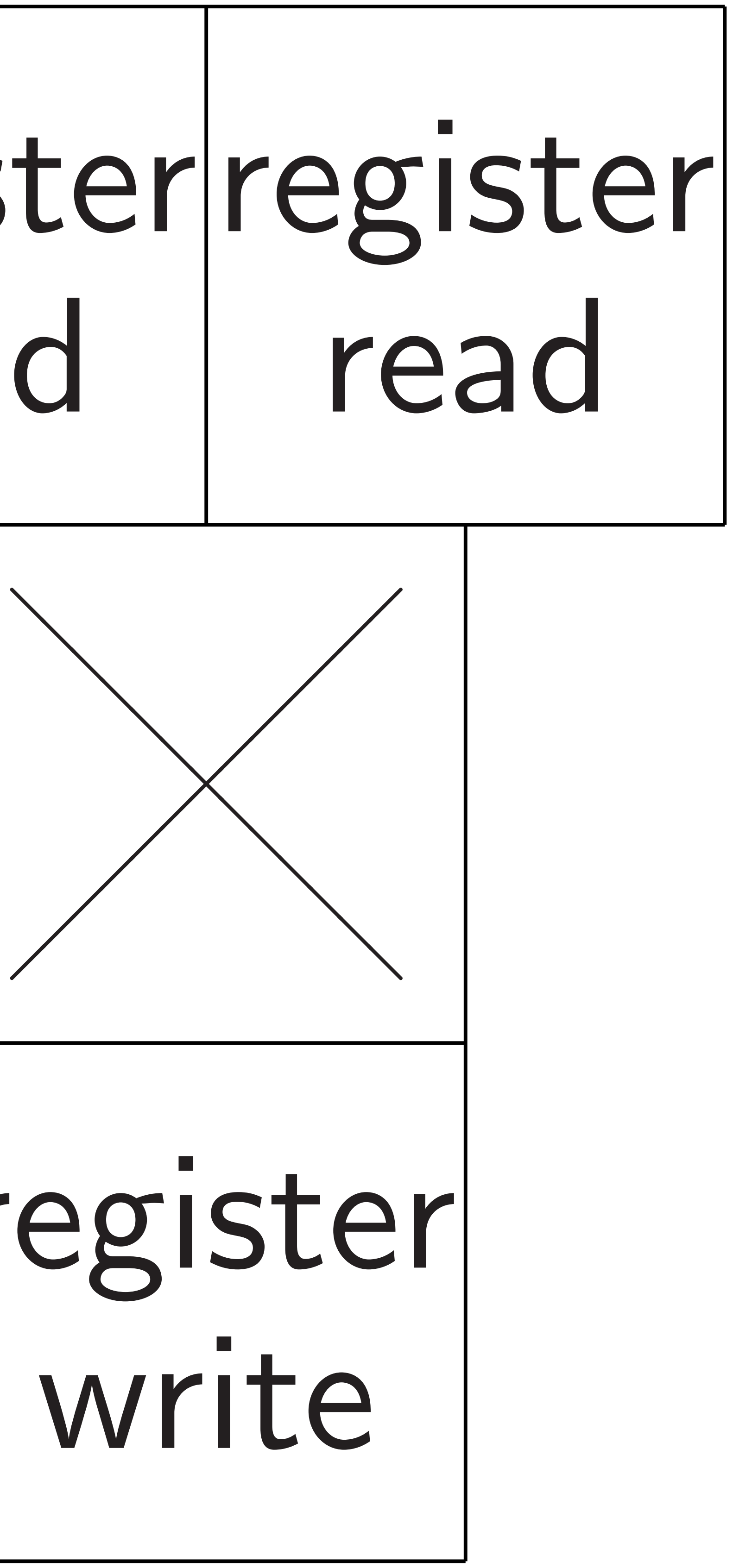
“Instruction fetch”:
$p \mapsto o_p, i_p, j_p, k_p, p'$.

“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
("$\times$", $i, j, k$) 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 "fl
storing (

Hook ($p$
flip-flops

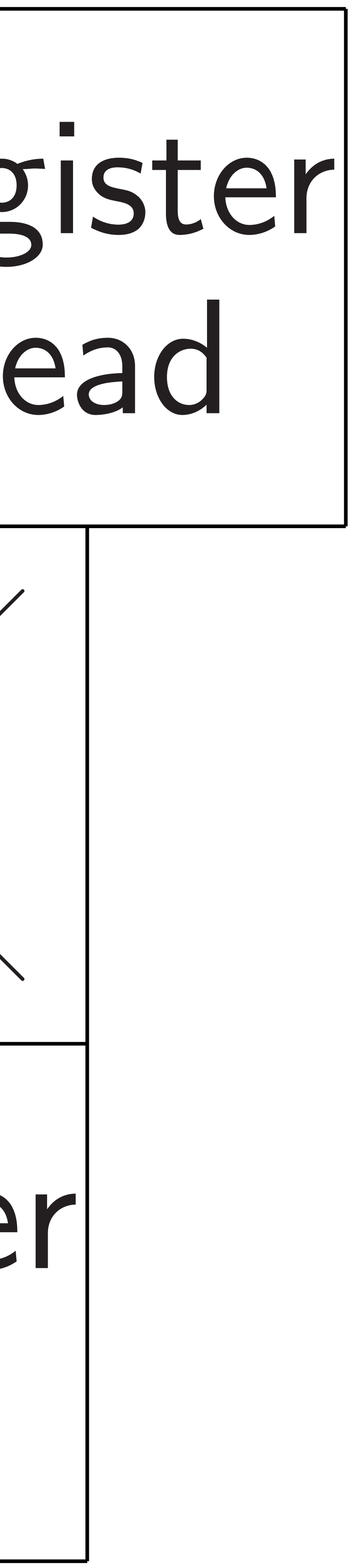
Hook ou
into the

At each
flip-flops
with the

Clock ne
for elect
all the w
from flip

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

ept $r'_i = r_j r_k$:






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 $(p, r_0, \ldots,$

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

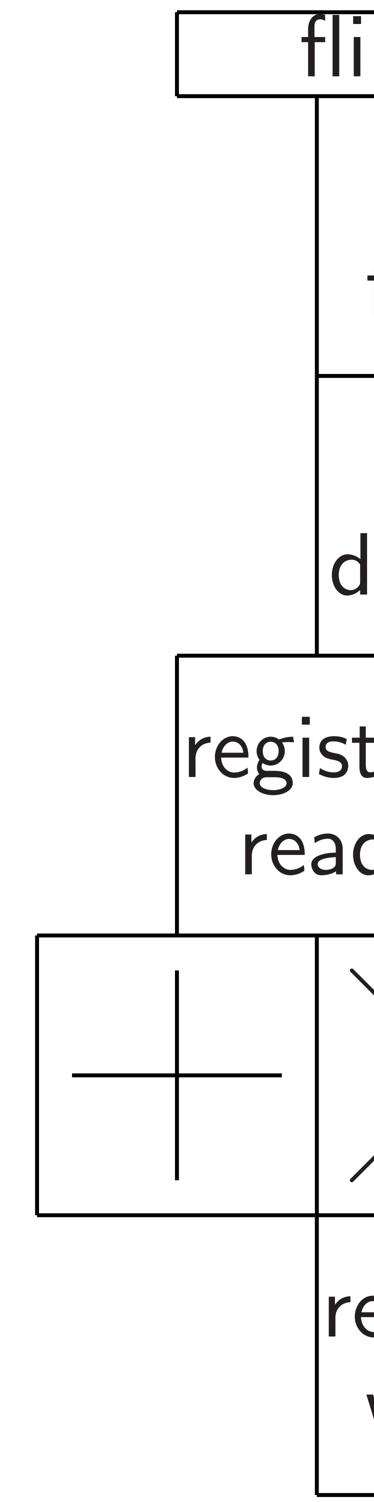
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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

e.g., rota

ty.

th

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.

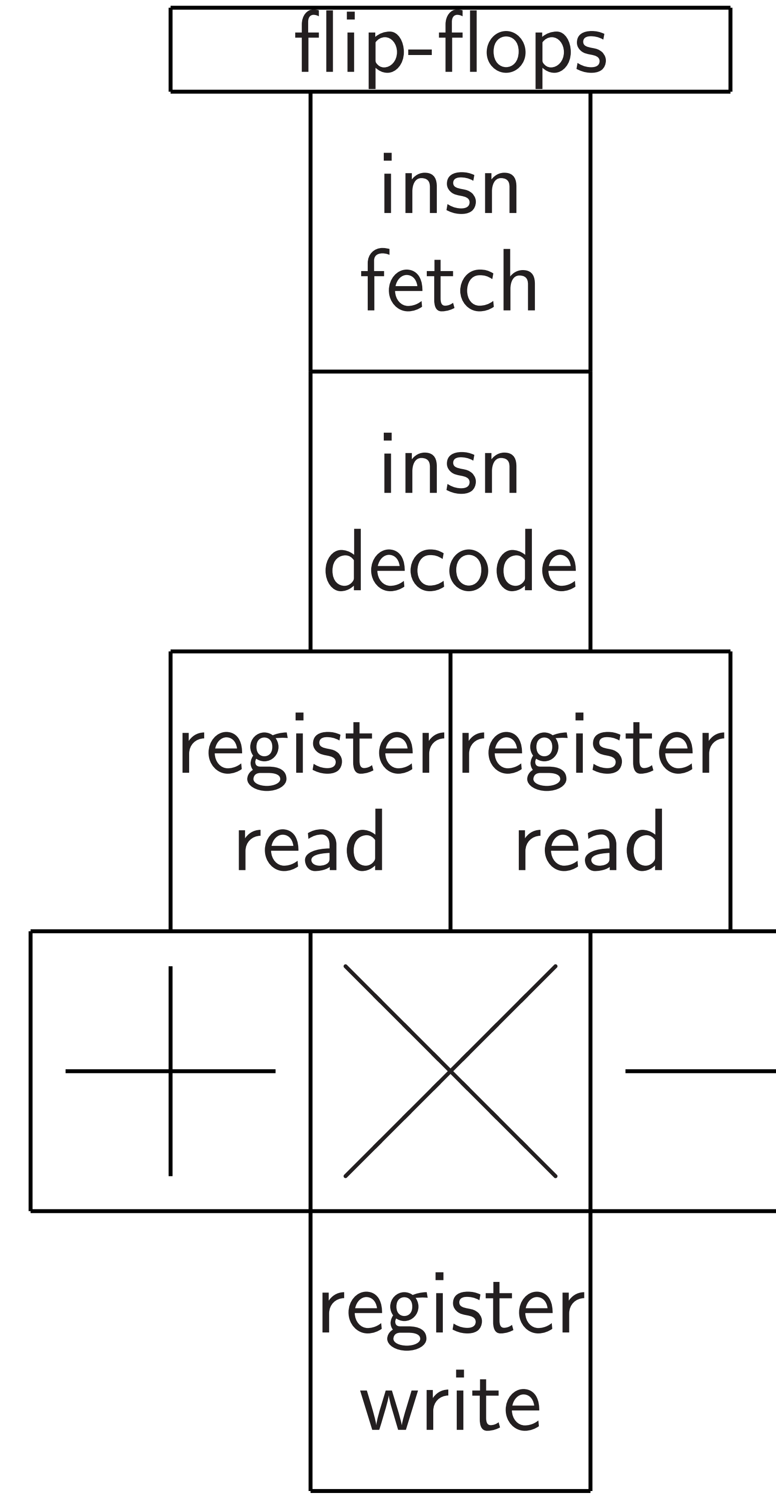
Hook outputs $(p', r'_0, \ldots, r'_{15})$
into the same flip-flops.

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
e.g., rotation instr

ns.

d



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

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

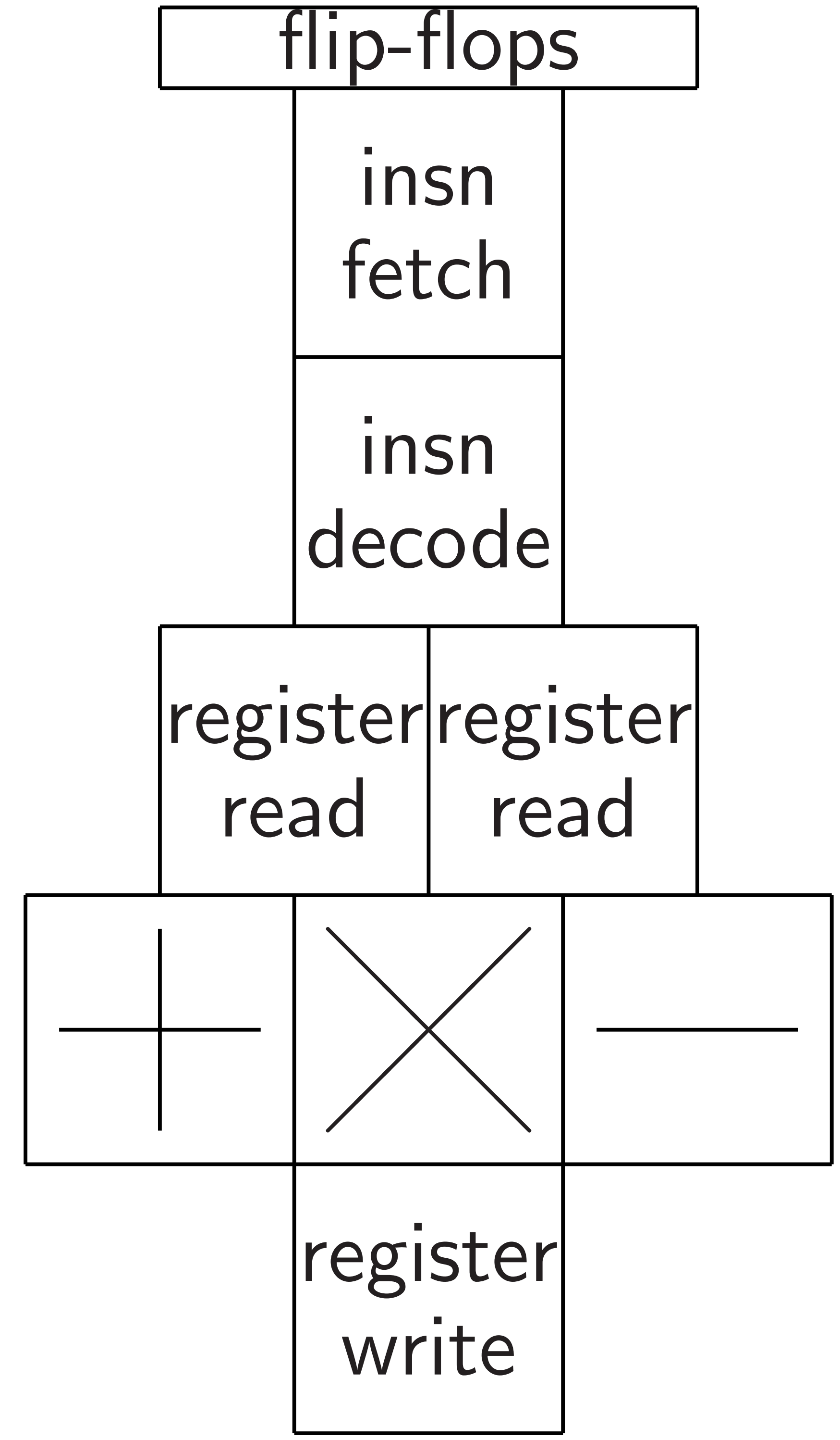
Hook outputs $(p', r'_0, \ldots, r'_{15})$
into the same flip-flops.

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:
e.g., rotation instructions.

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

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

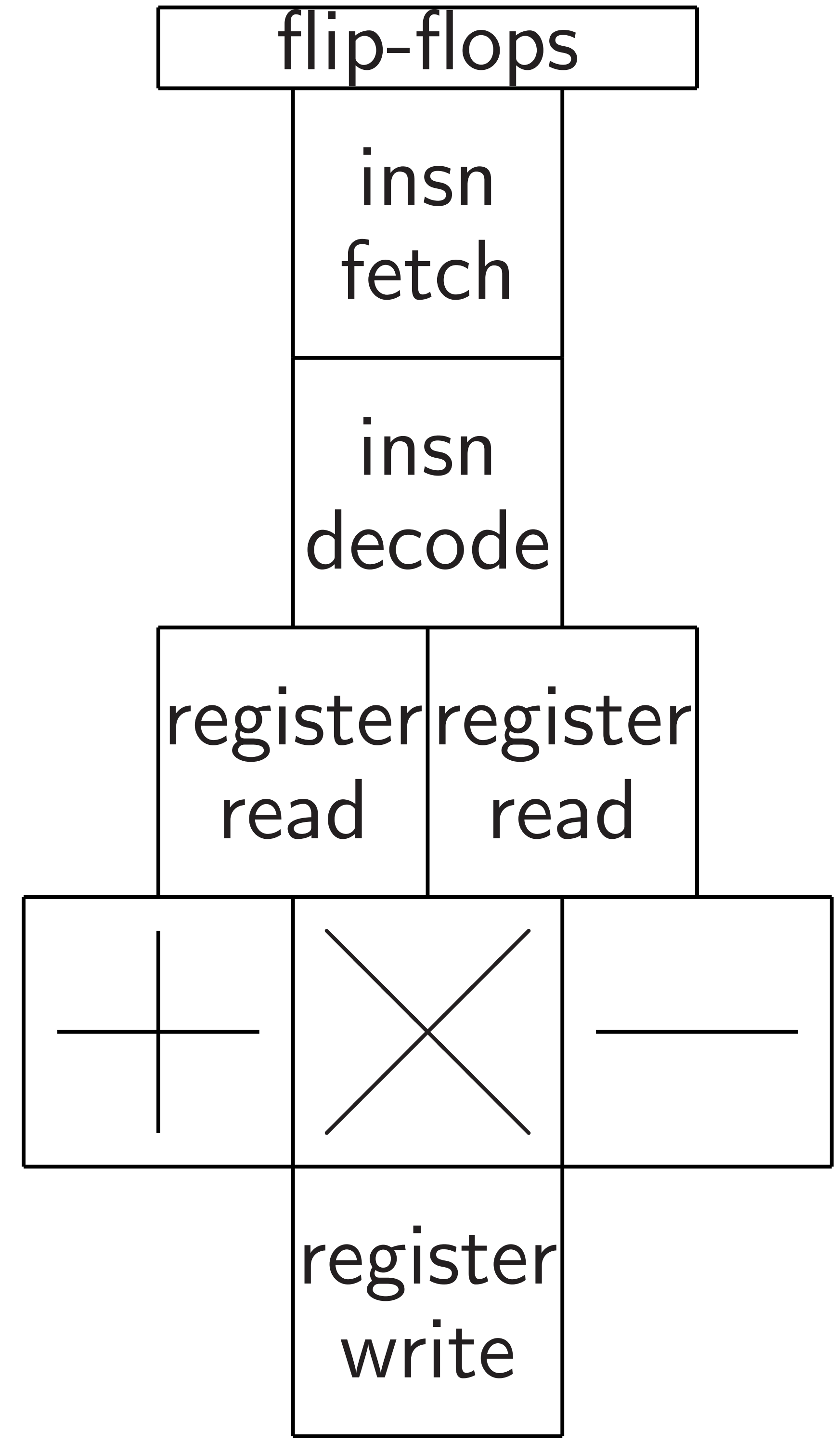
Hook outputs $(p', r'_0, \ldots, r'_{15})$
into the same flip-flops.

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:
e.g., rotation instructions.

lip-flops"

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

$, r_0, \ldots, r_{15})$

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

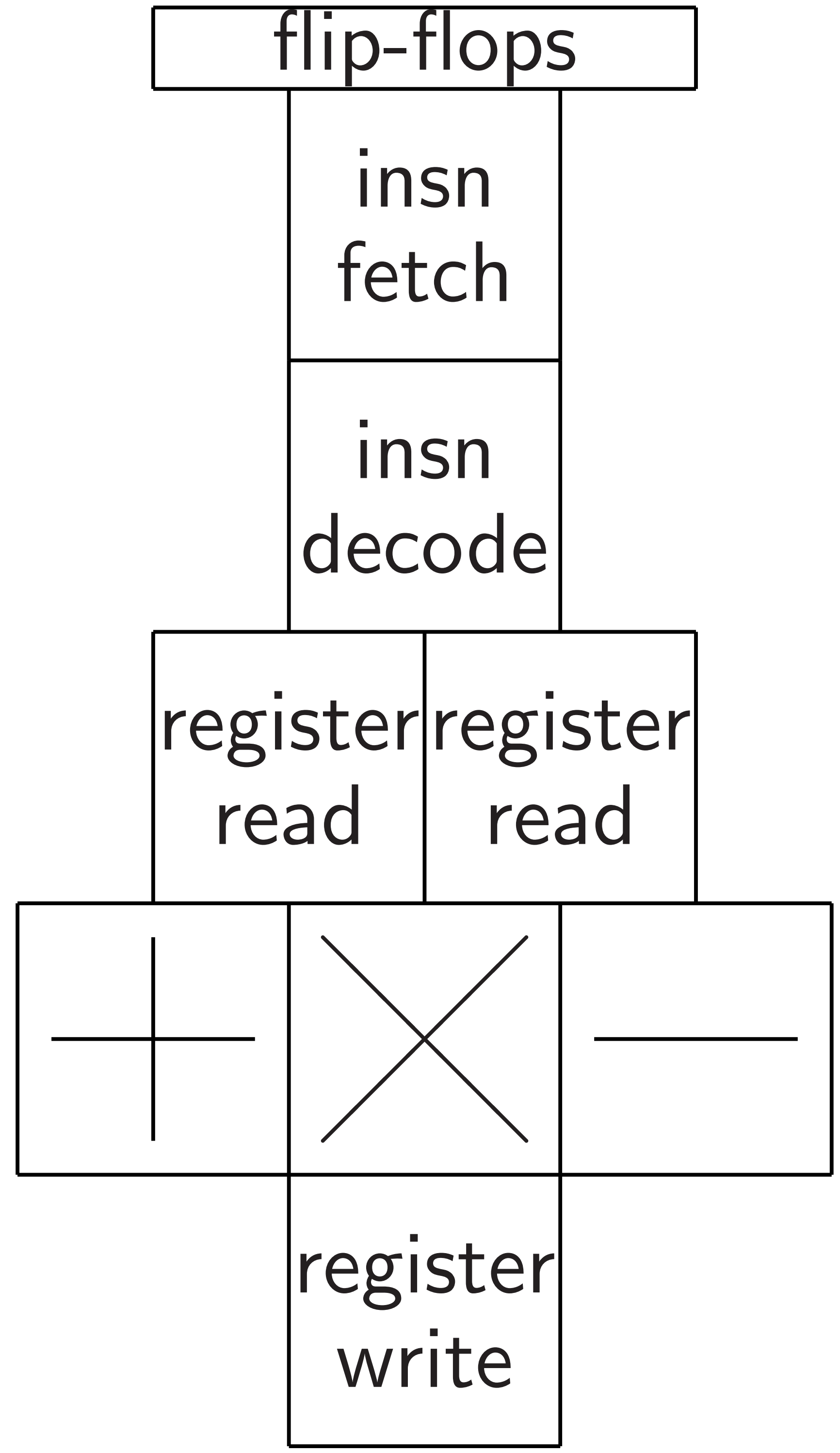
ricity to percolate

vay through the circuit,

-flops to flip-flops.



Now have semi-flexible CPU:




Further flexibility is useful:

e.g., rotation instructions.

"Pipelin

$r_{15}$).

$_{15}$)

uit inputs.

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

flops.

k”,

written

slow enough

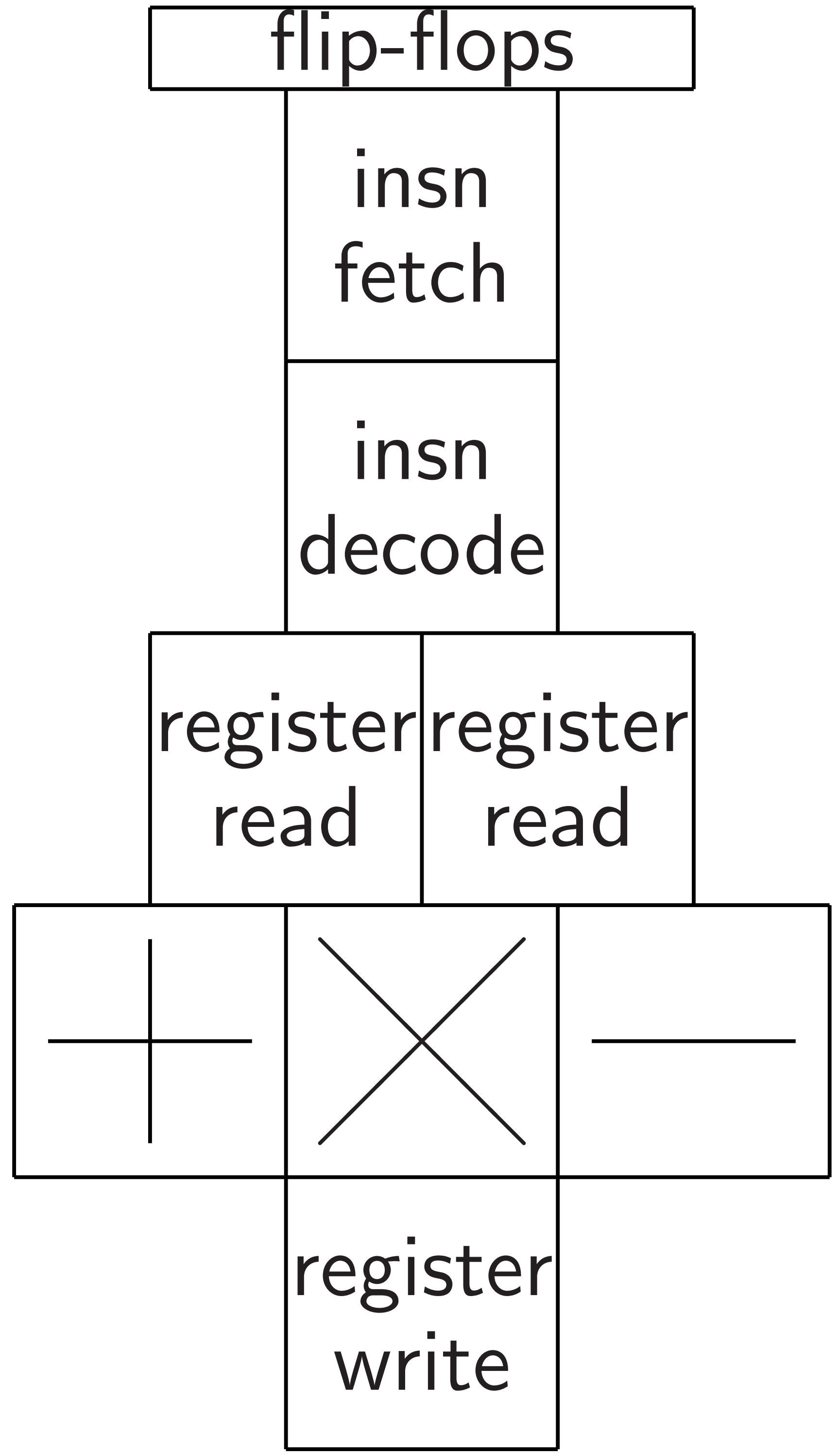
ercolate

h the circuit,

flip-flops.

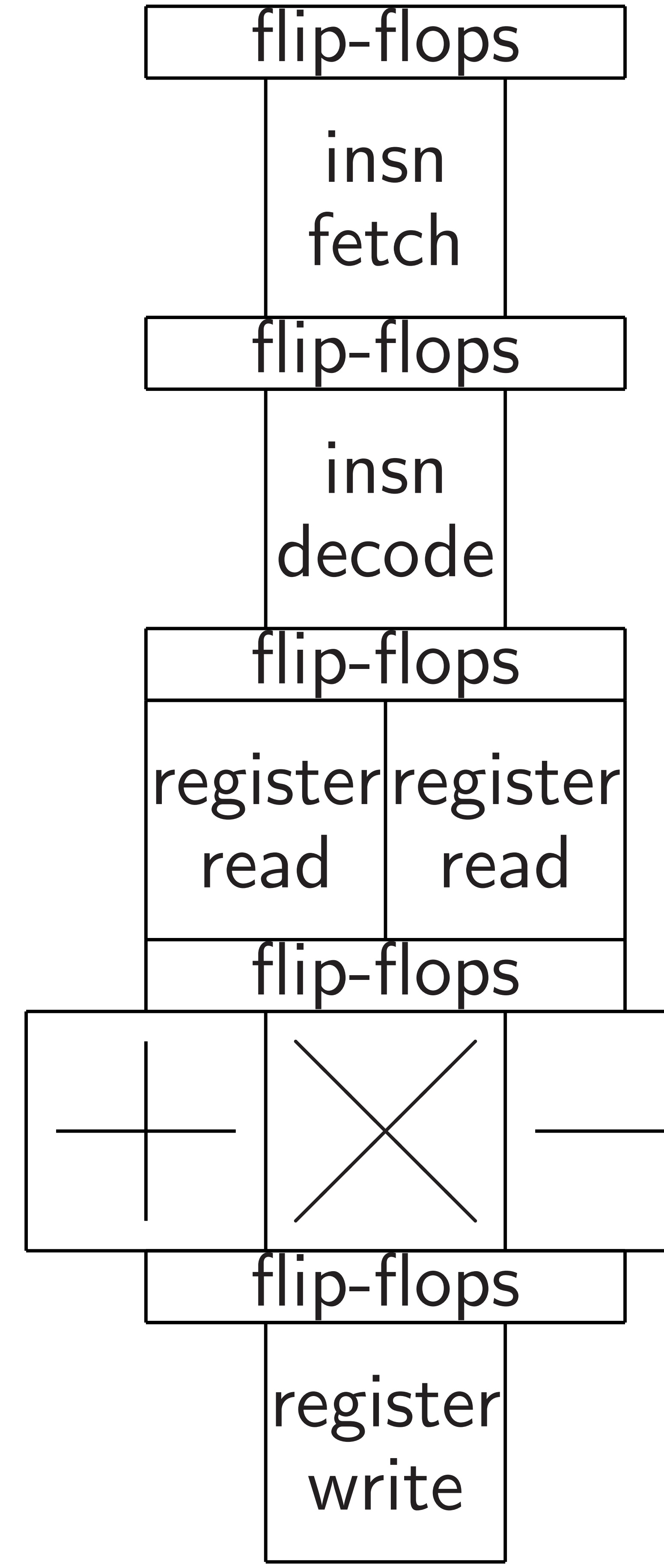


Now have semi-flexible CPU:




Further flexibility is useful: e.g., rotation instructions.

“Pipelining” allow



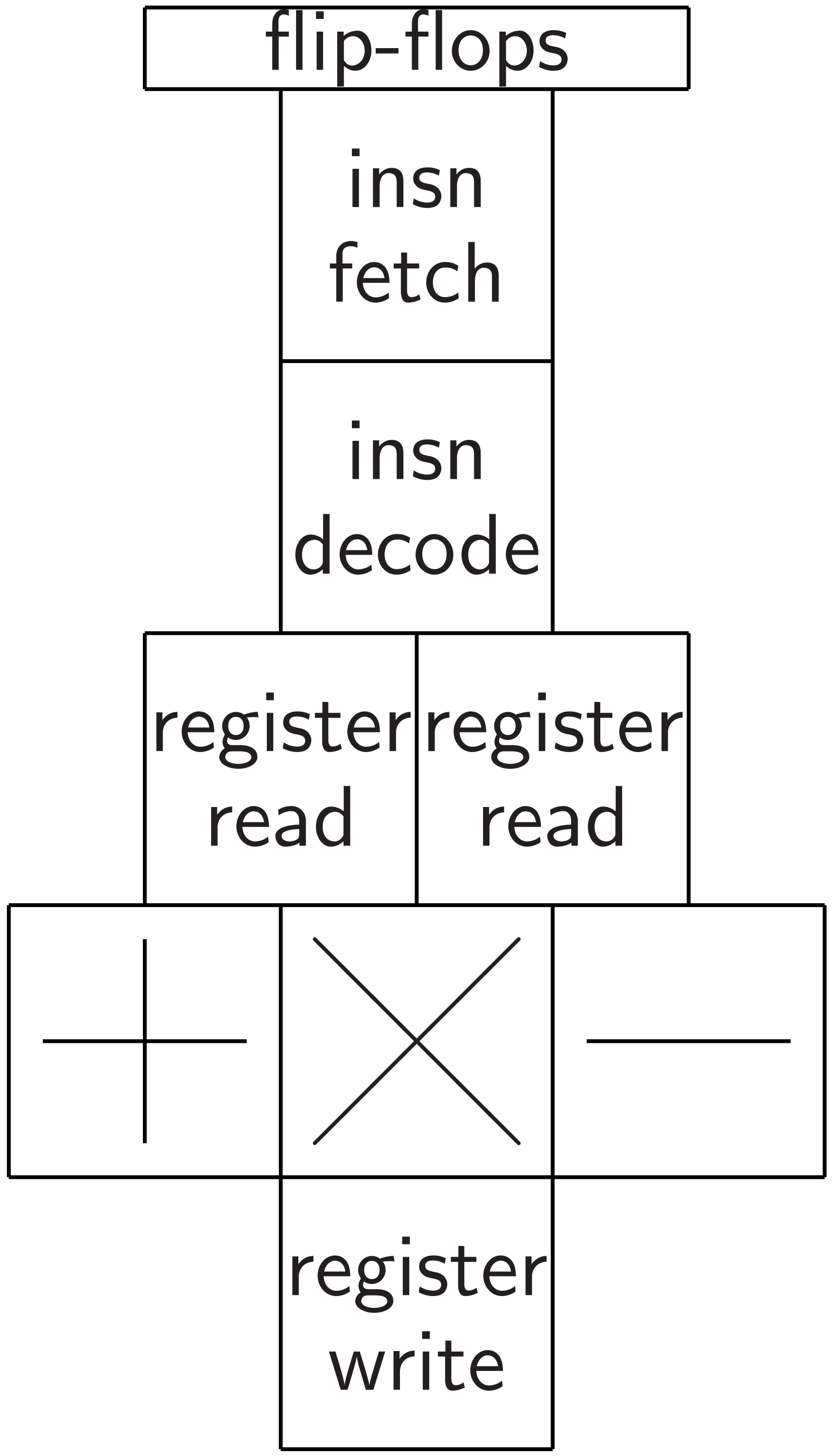
)

ugh

uit,



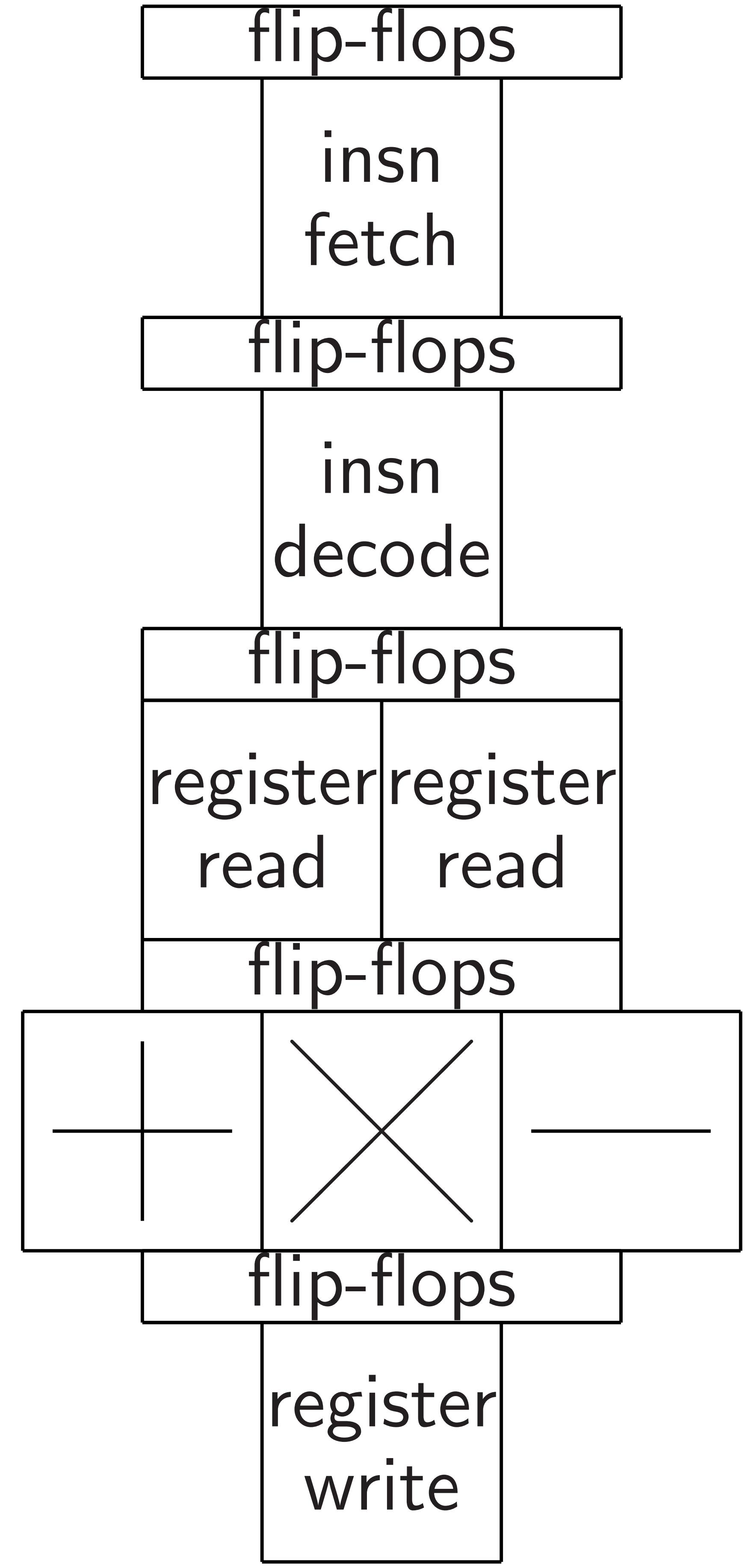
Now have semi-flexible CPU:




Further flexibility is useful:
e.g., rotation instructions.



“Pipelining” allows faster cl




stage

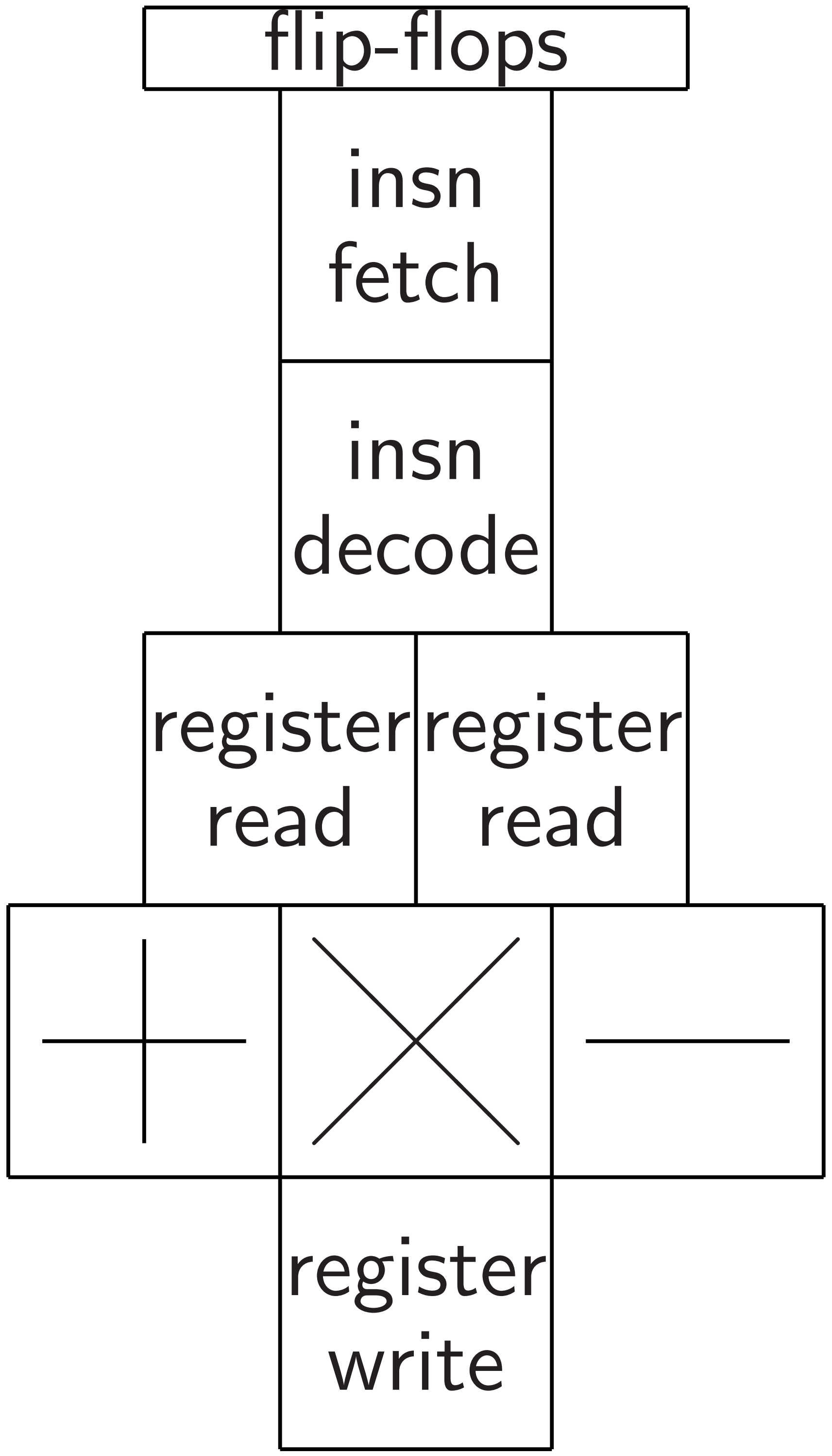
stage

stage

stage

stage

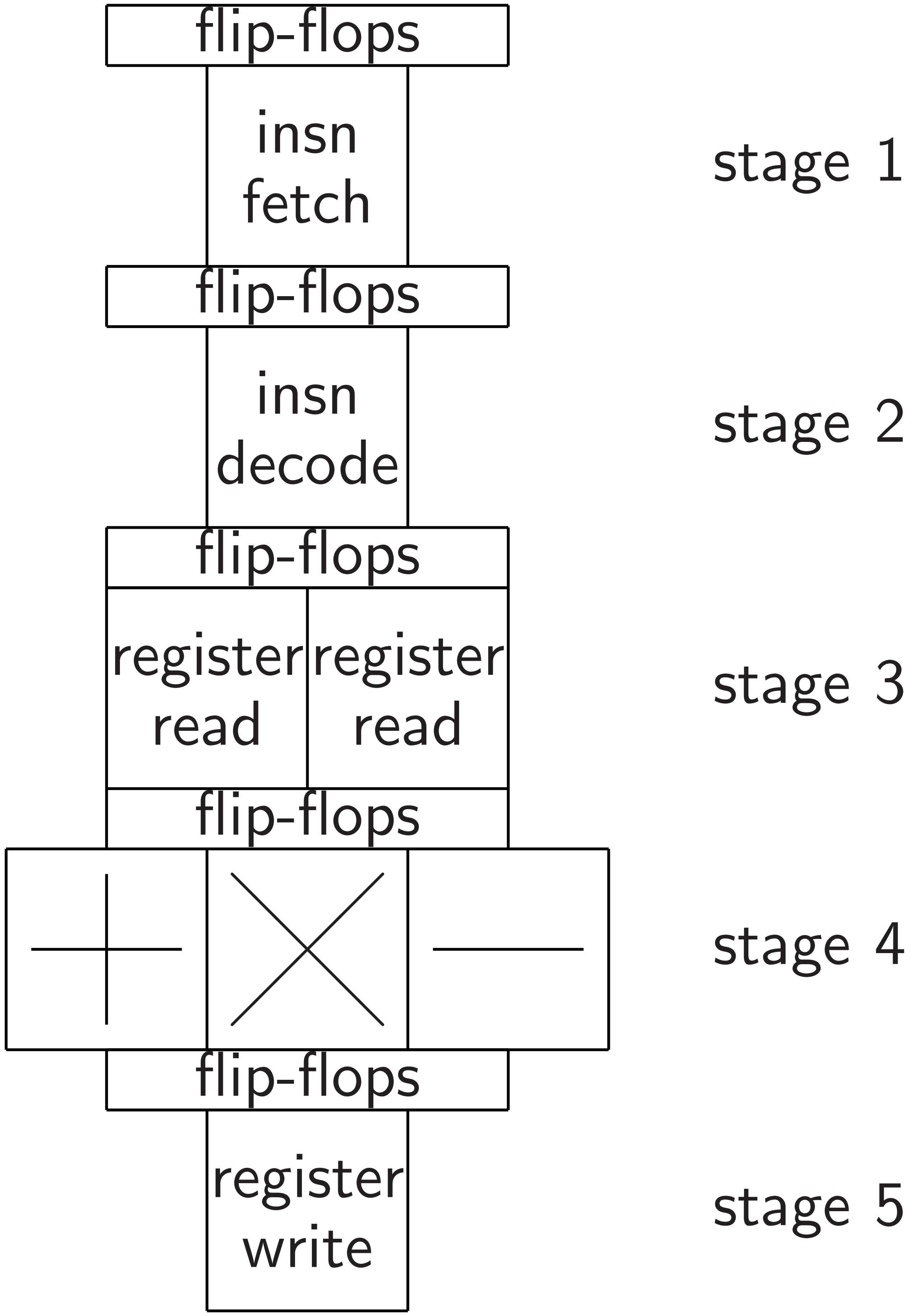
Now have semi-flexible CPU:



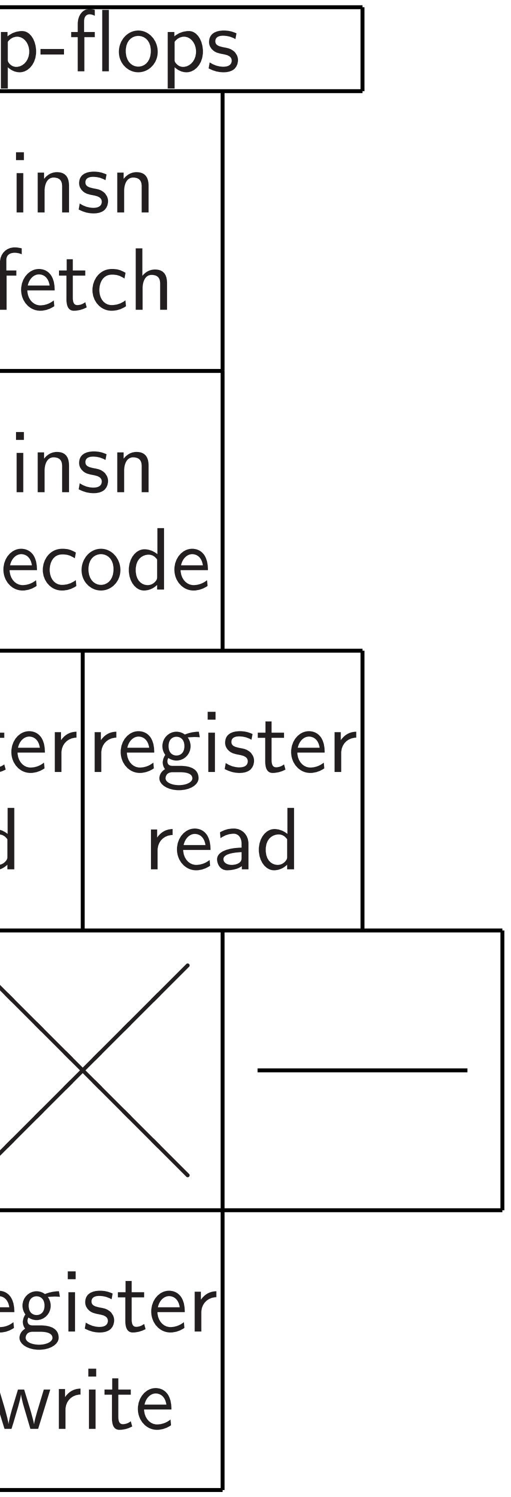

Further flexibility is useful: e.g., rotation instructions.



"Pipelining" allows faster clock:



ve semi-flexible CPU:

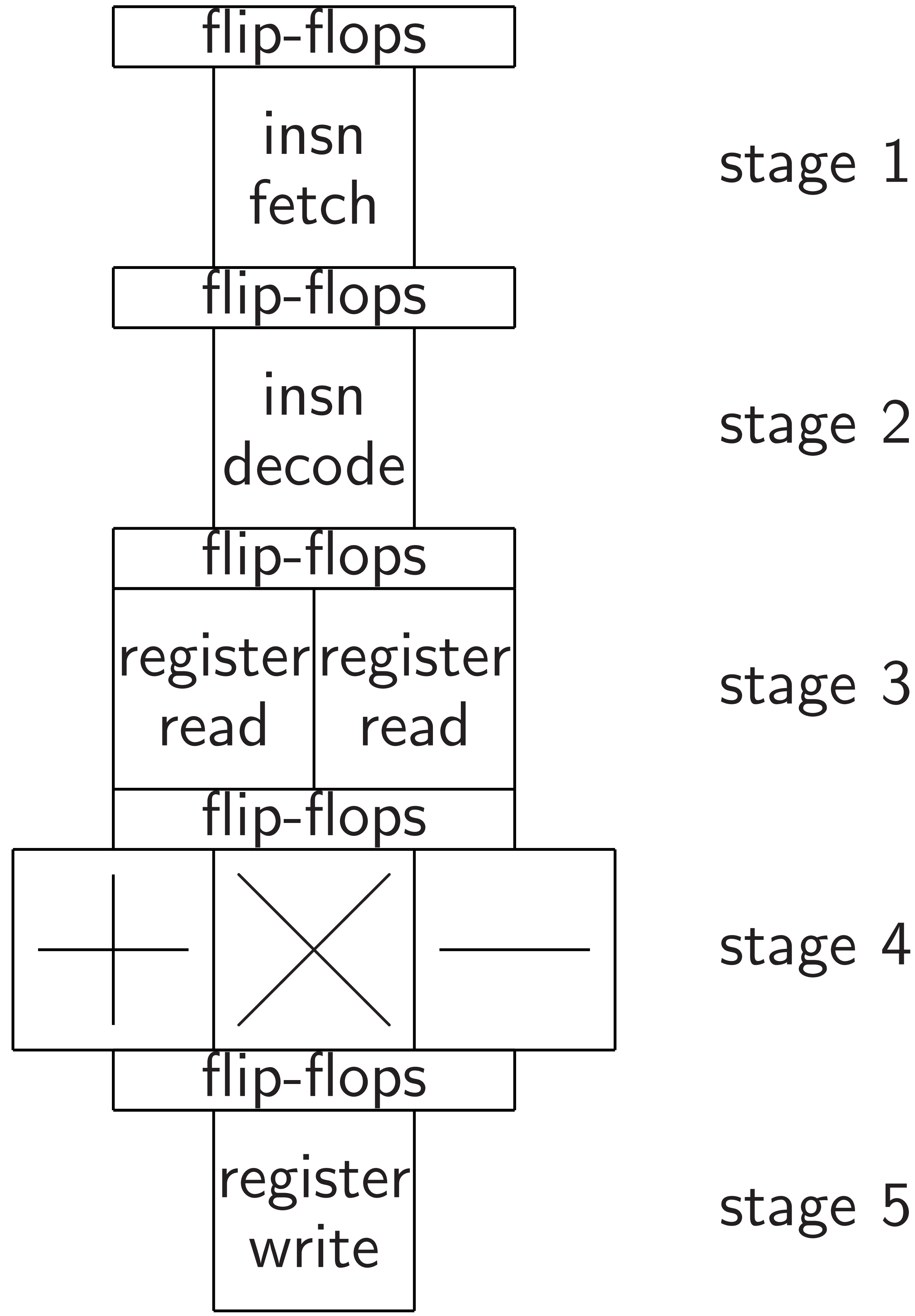



flexibility is useful:

ation instructions.



“Pipelining” allows faster clock:




Goal: St

one tick

Instructi

reads ne

feeds $p'$

After ne

instructi

uncompr

while ins

reads an

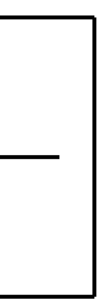
Some ex

Also ext

preserve

e.g., stal

xible CPU:

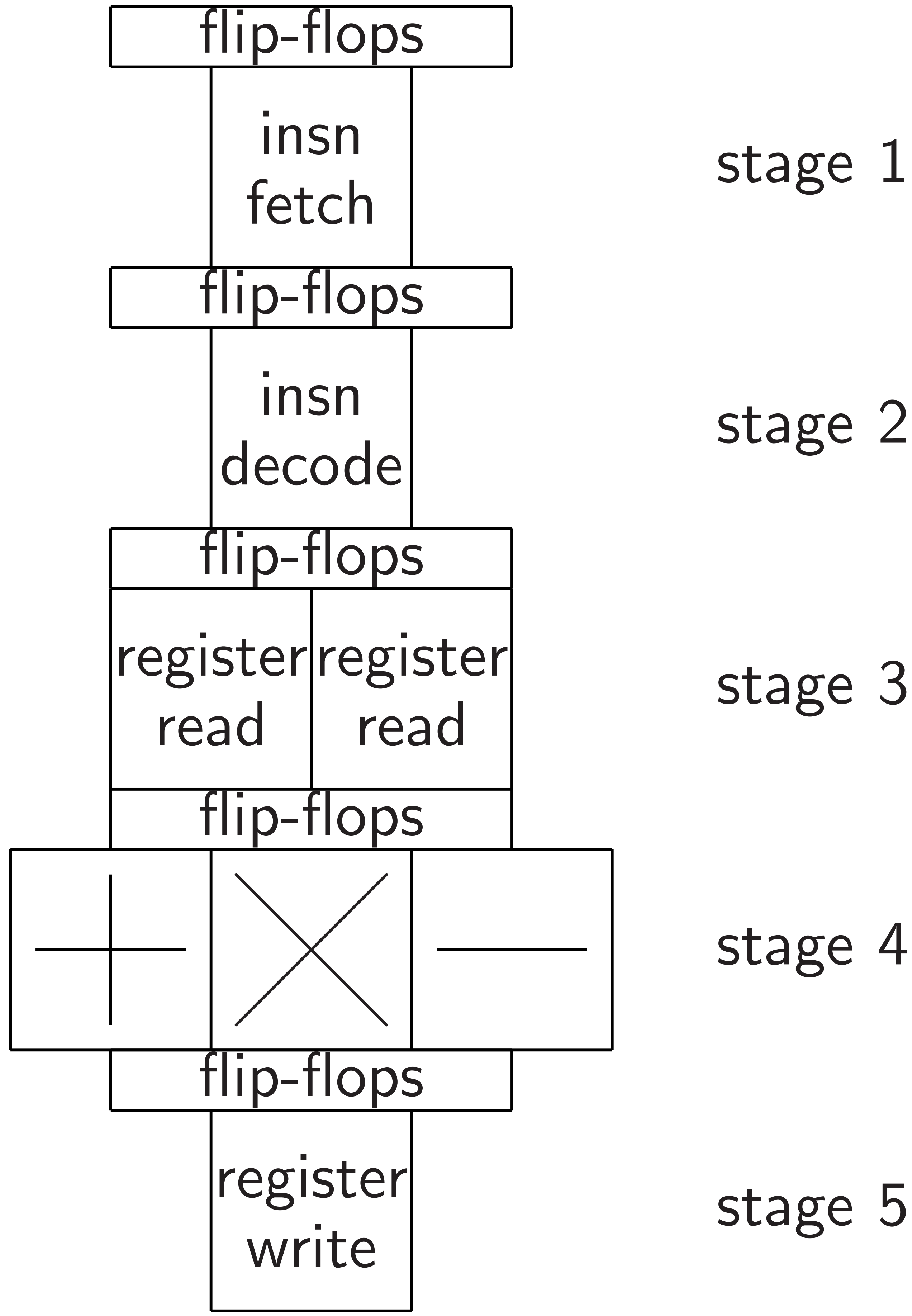


is useful:

uctions.



“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

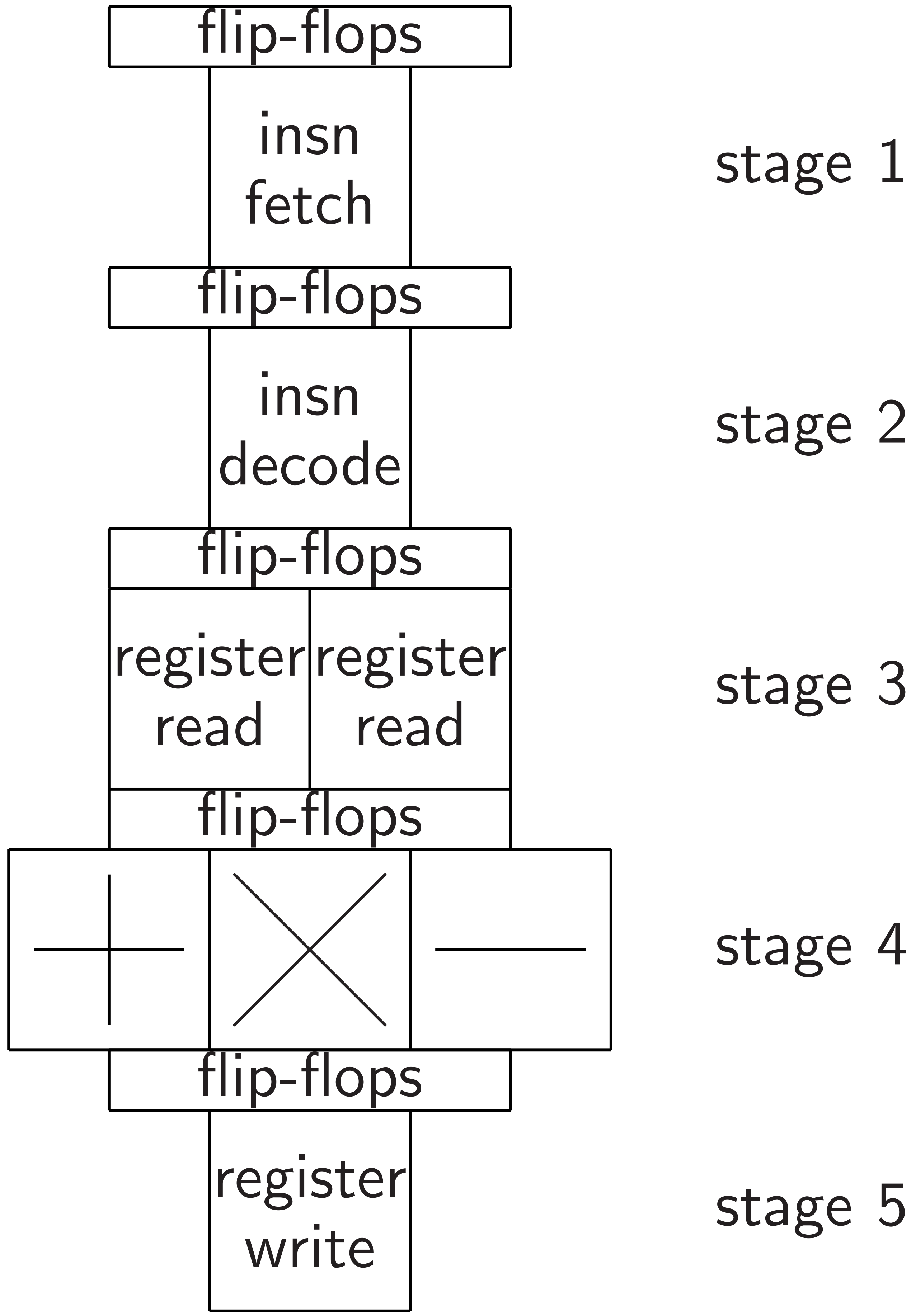
preserve instructio

e.g., stall on read-

:



"Pipelining" allows faster clock:





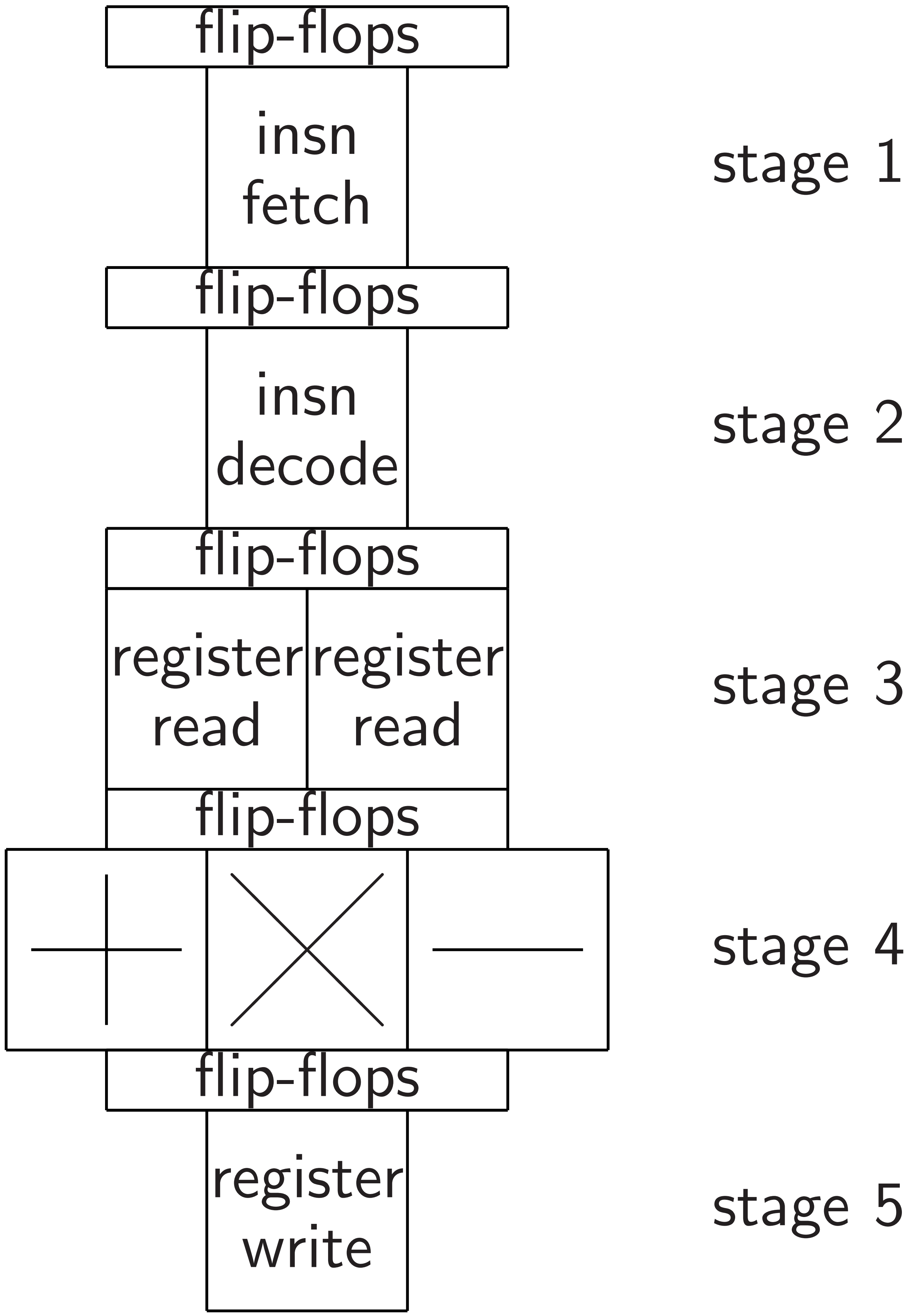

Goal: Stage $n$ handles instru
one tick after stage $n - 1$.

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:





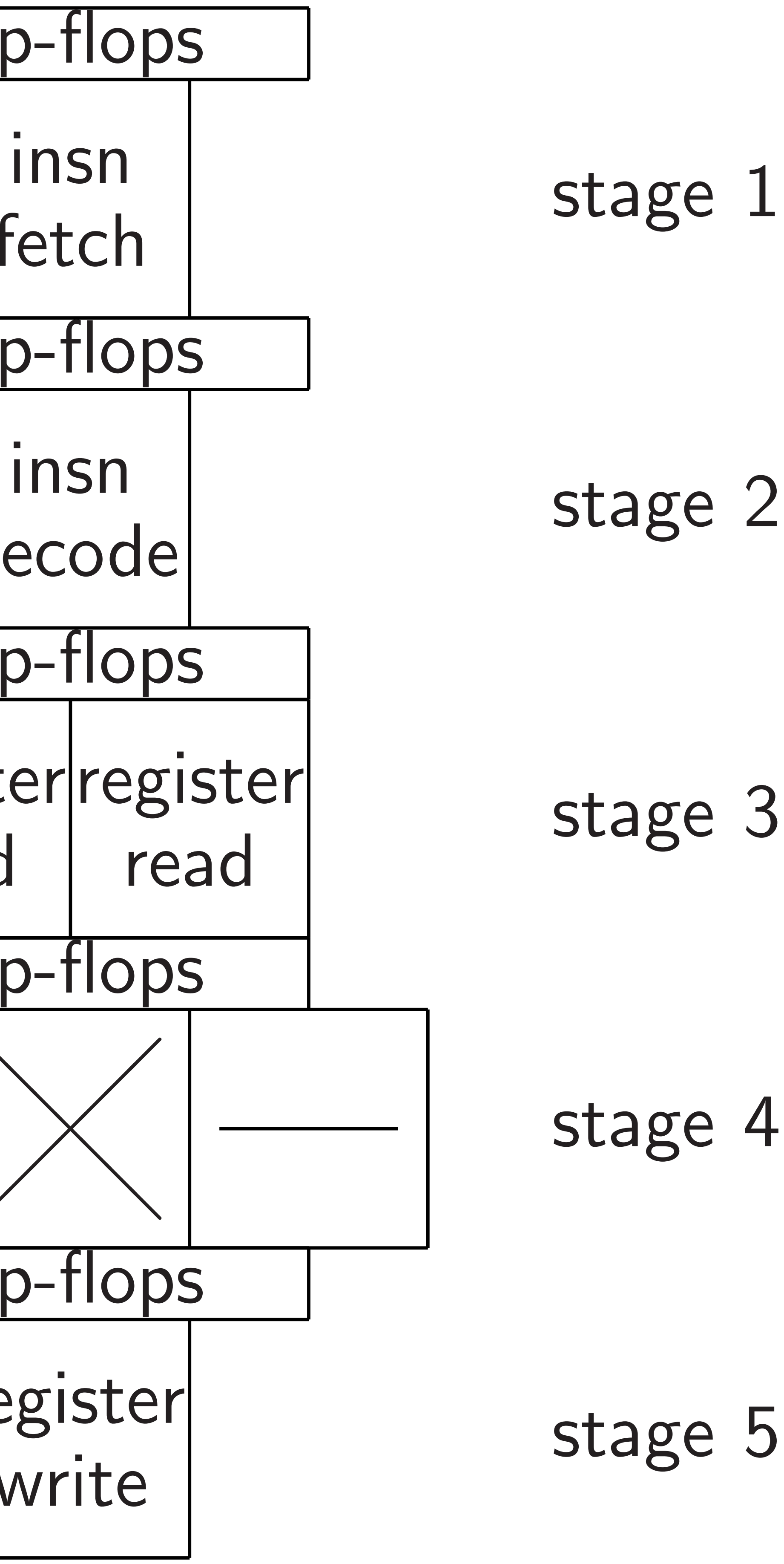

Goal: Stage $n$ handles instruction one tick after stage $n - 1$.

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:





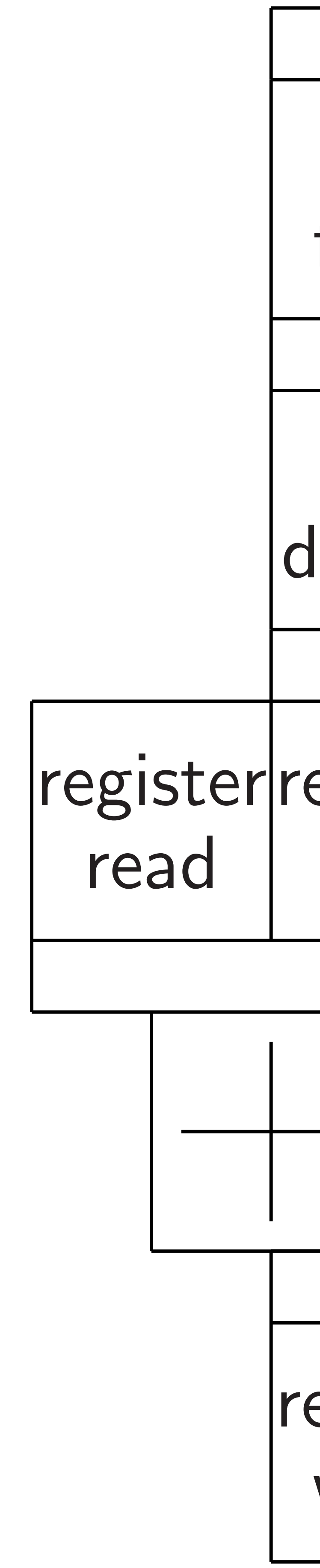

Goal: Stage $n$ handles instruction one tick after stage $n-1$.

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

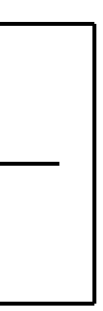


s faster clock:

stage 1

stage 2

stage 3



stage 4

stage 5



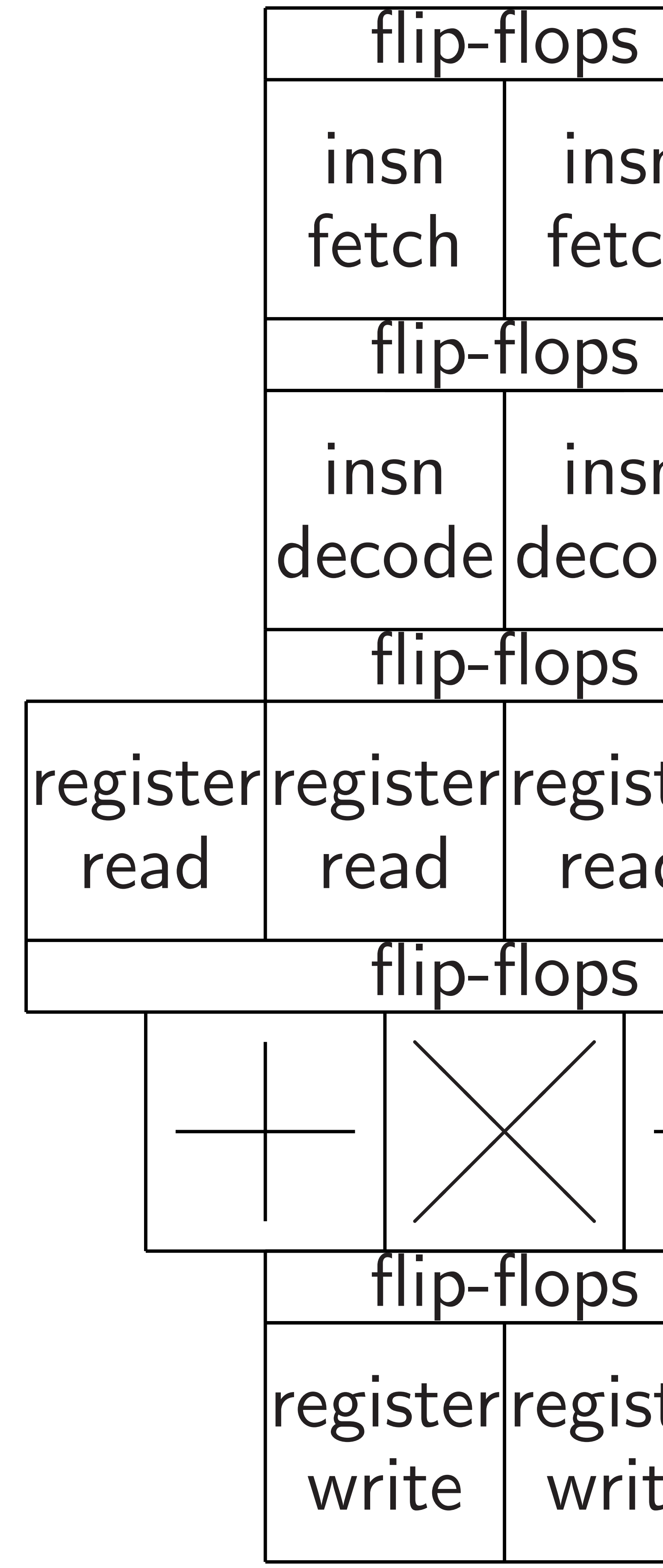
Goal: Stage $n$ handles instruction one tick after stage $n-1$.

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 $n - 1$.

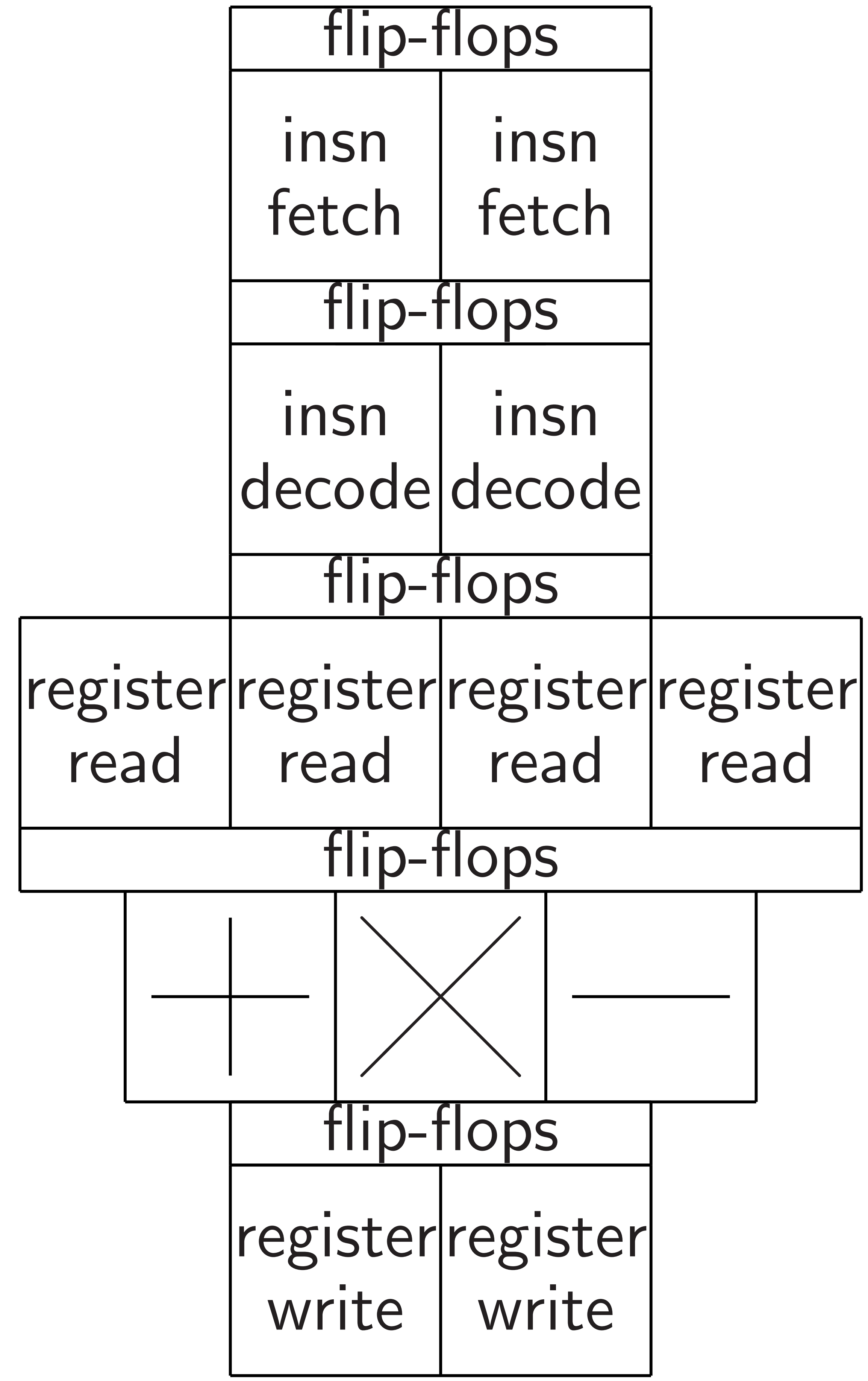
Instruction fetch reads next instruction, feeds $p'$ back, sends instruction.

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 $n - 1$.

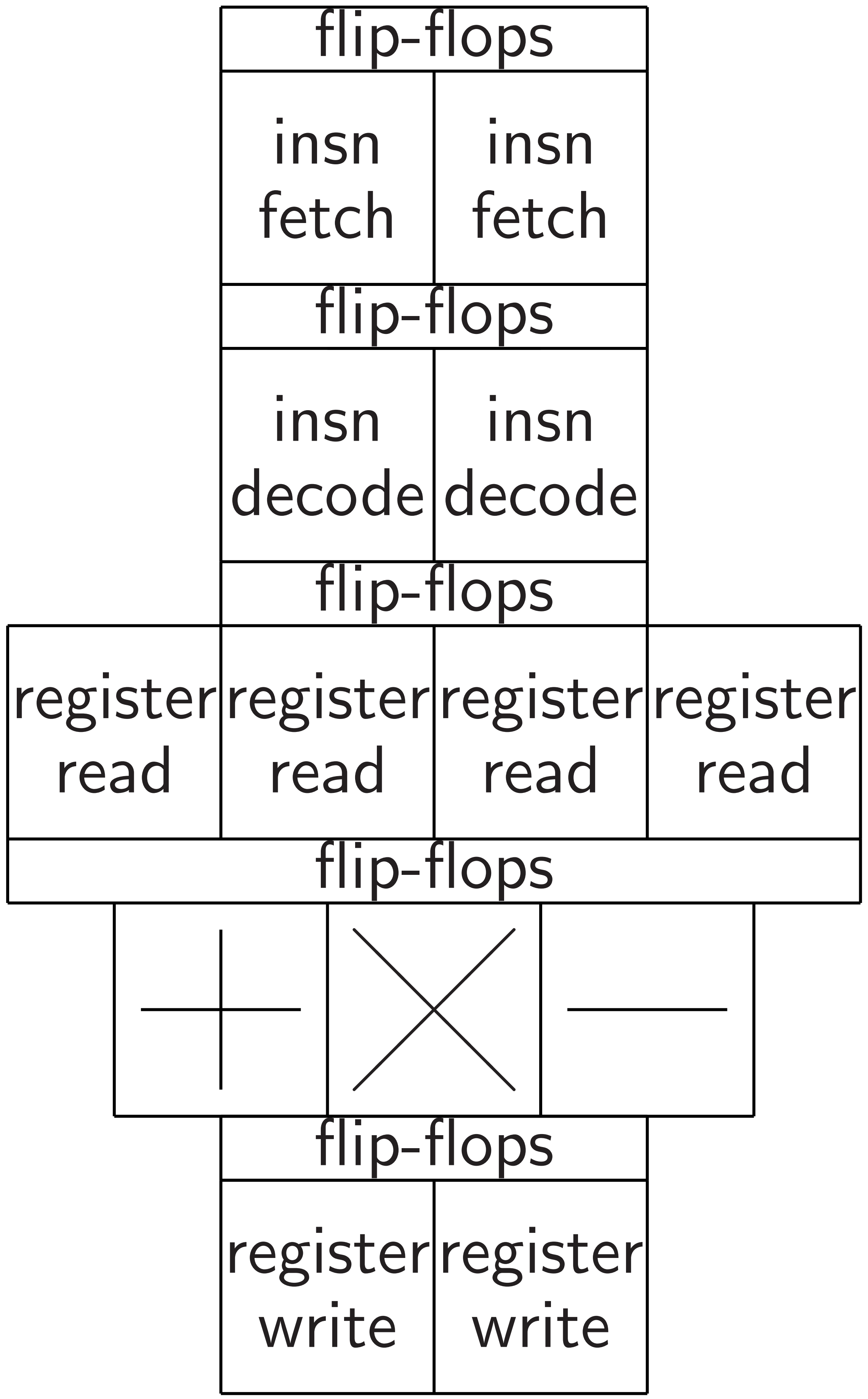
Instruction fetch reads next instruction, feeds $p'$ back, sends instruction.

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 $n-1$.

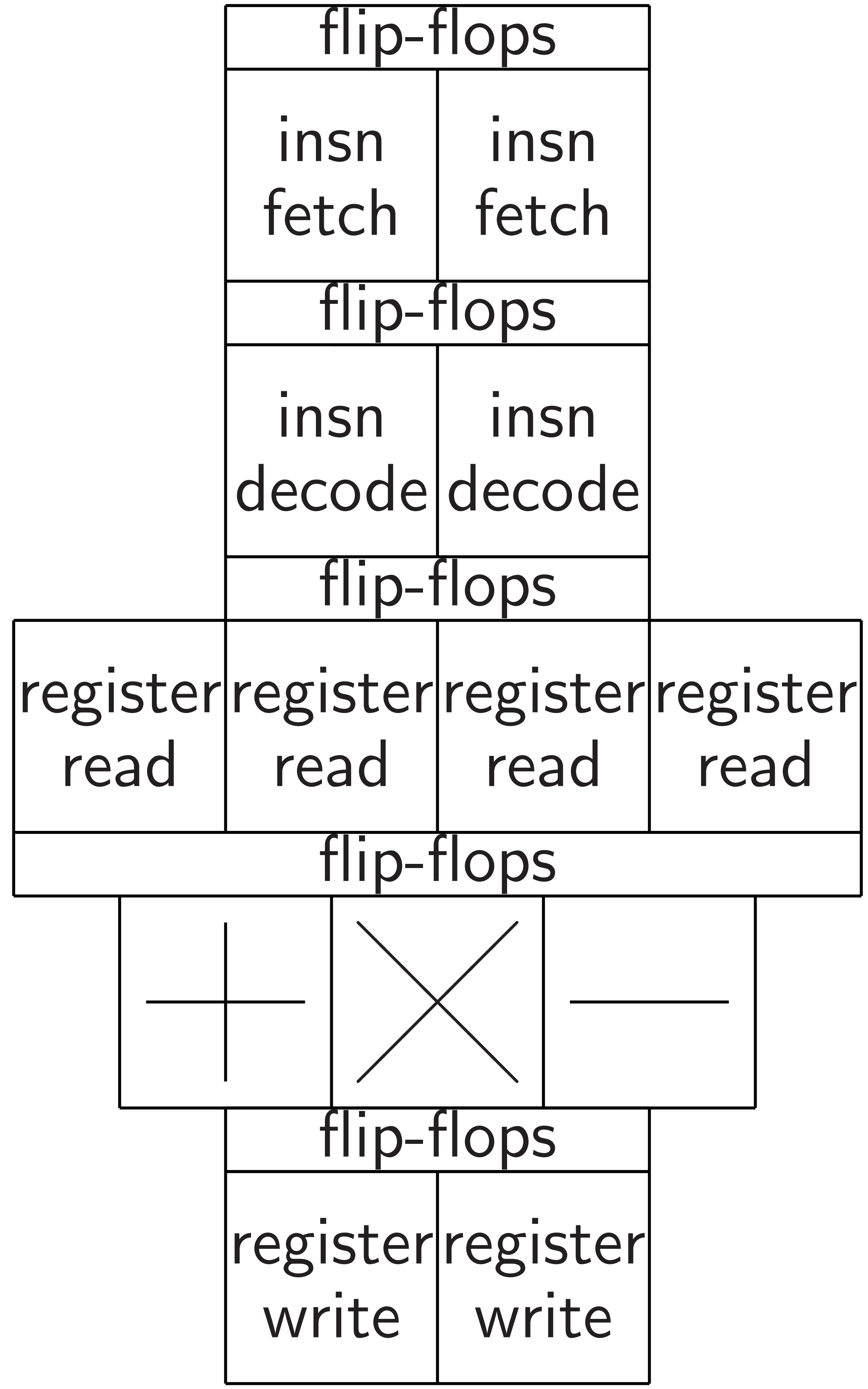
on fetch
xt instruction,
back, sends instruction.

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.

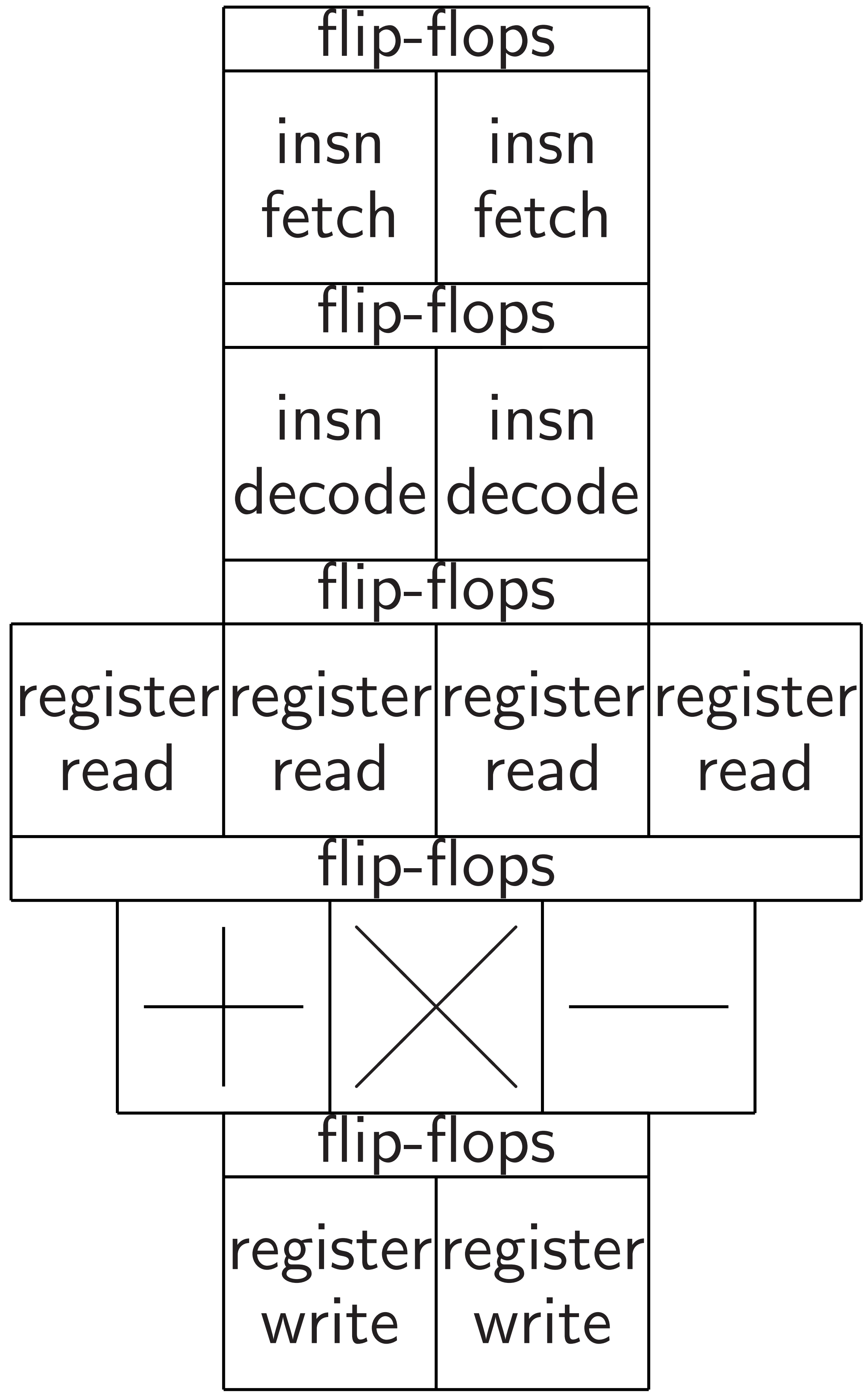
ck,
e
instruction,
etch
ruction.

op 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

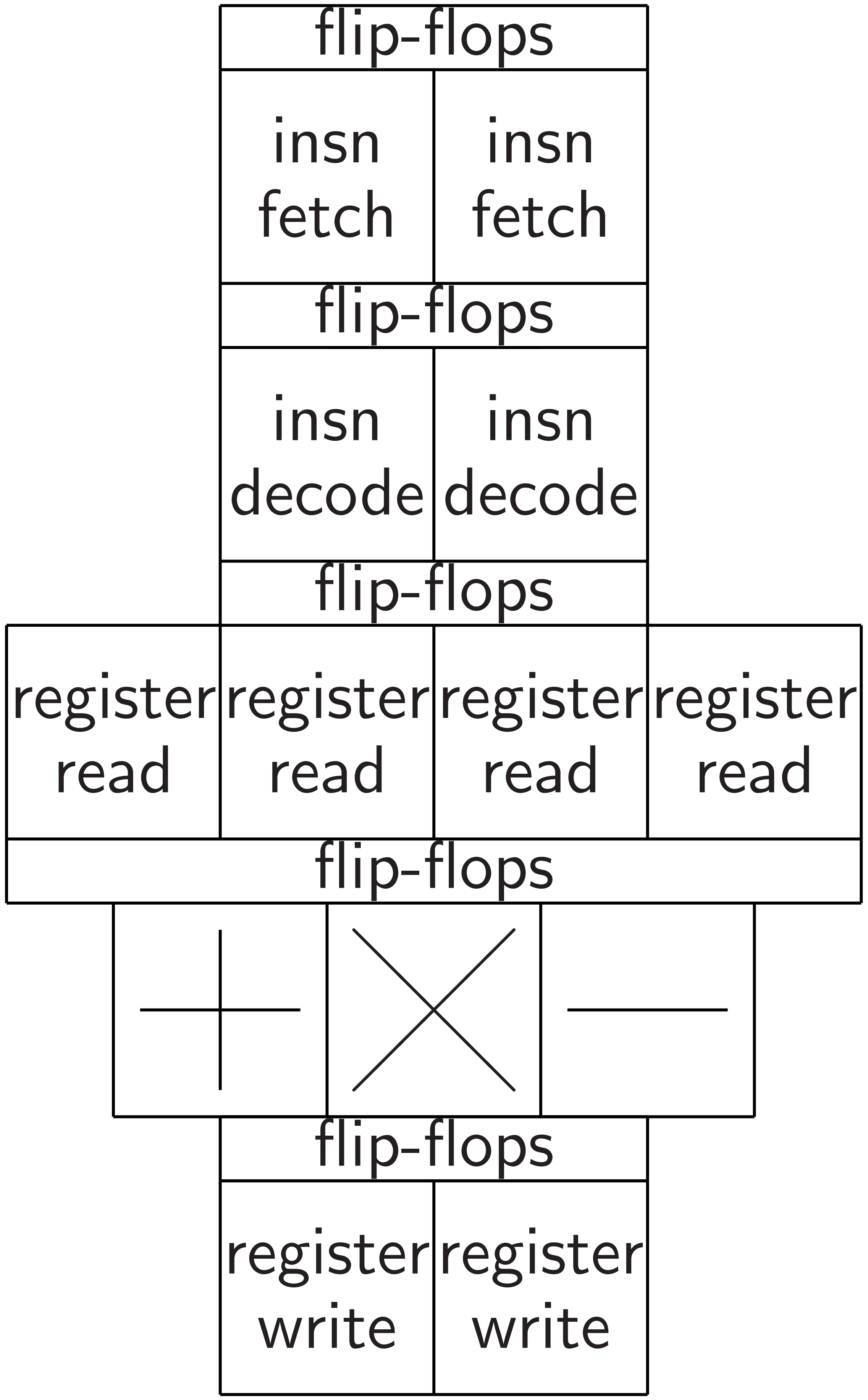
tion.

n,

cs:

e.

“Superscalar” processing:

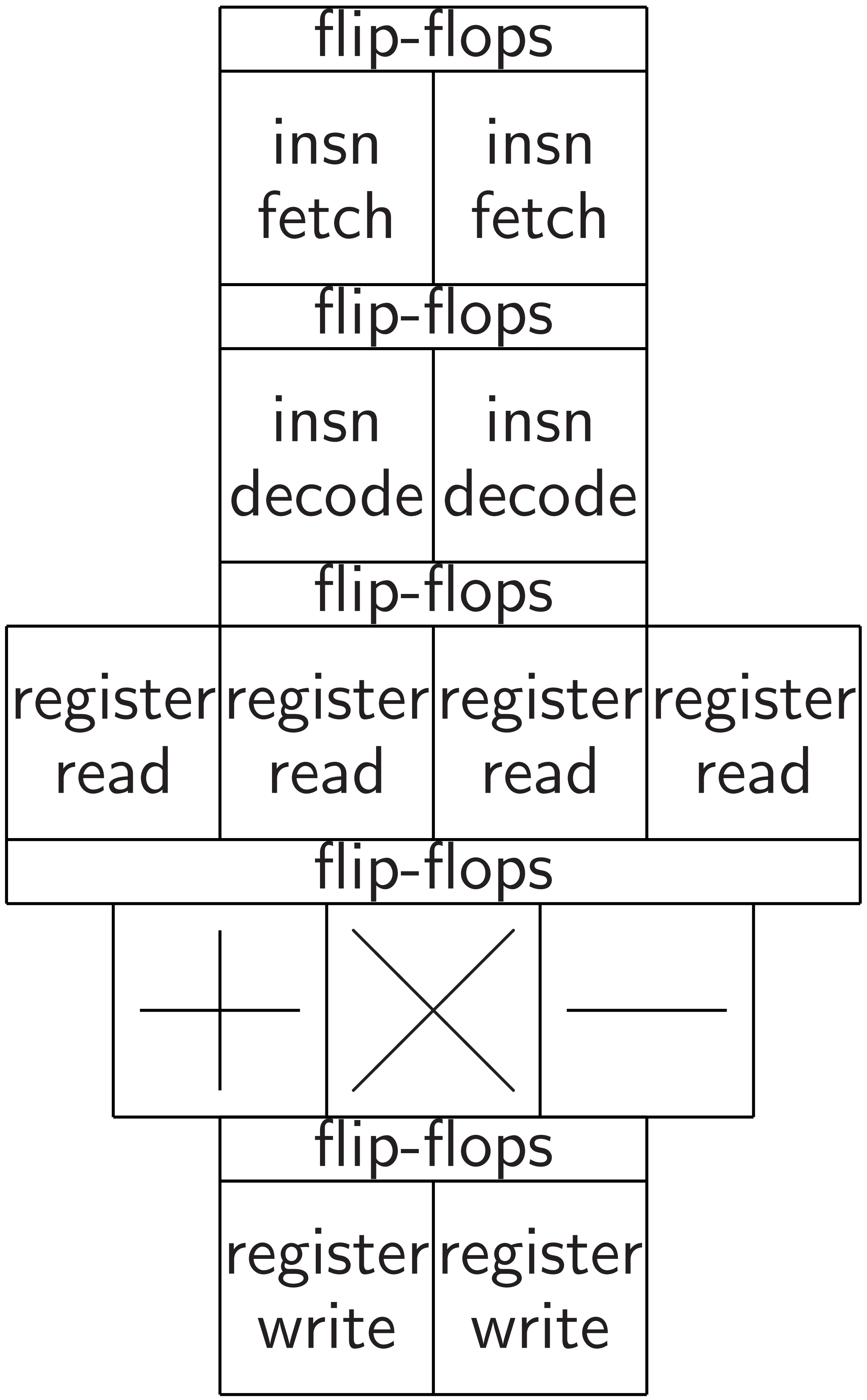





“Vector” processing:

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:

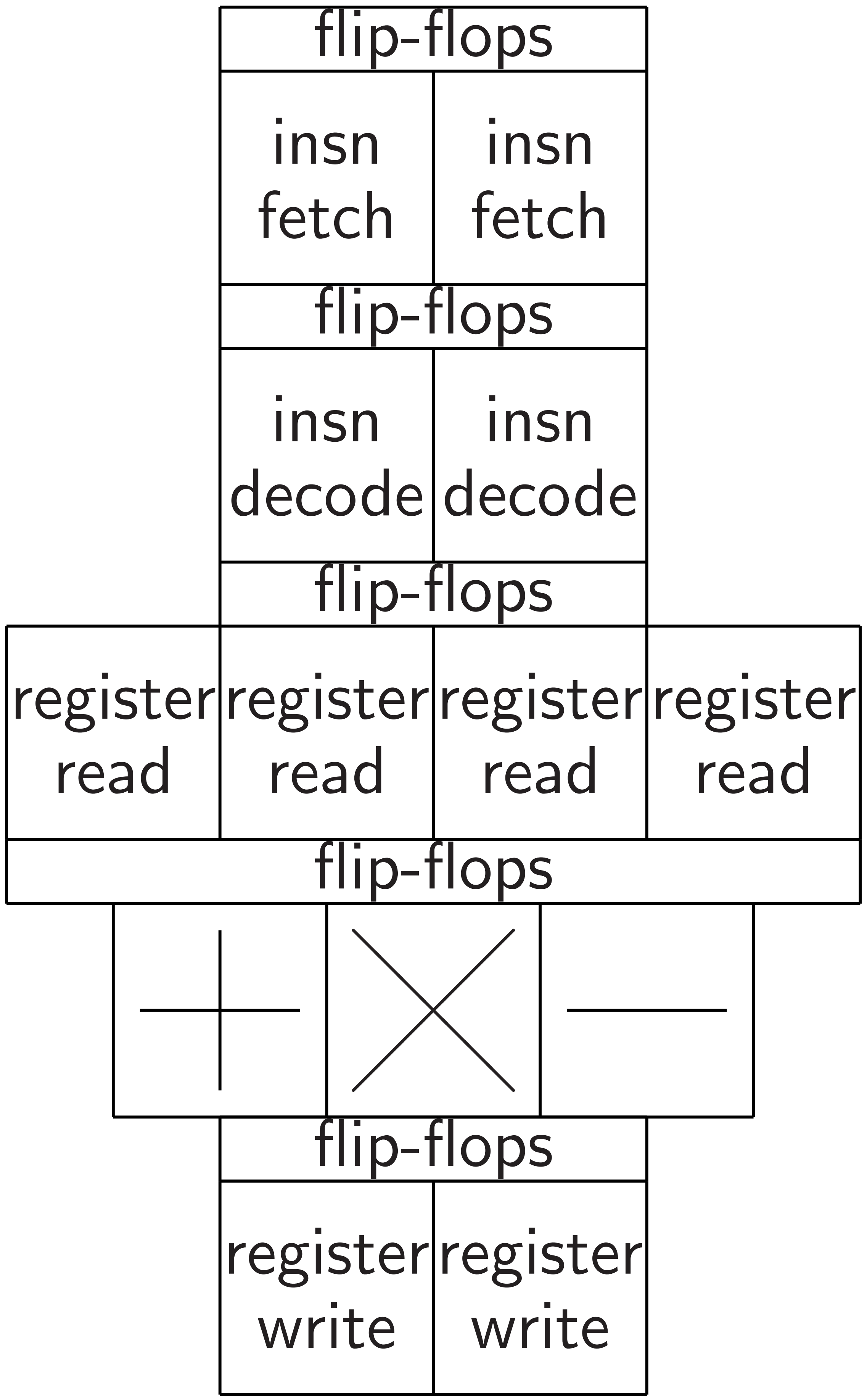





“Vector” processing:

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:



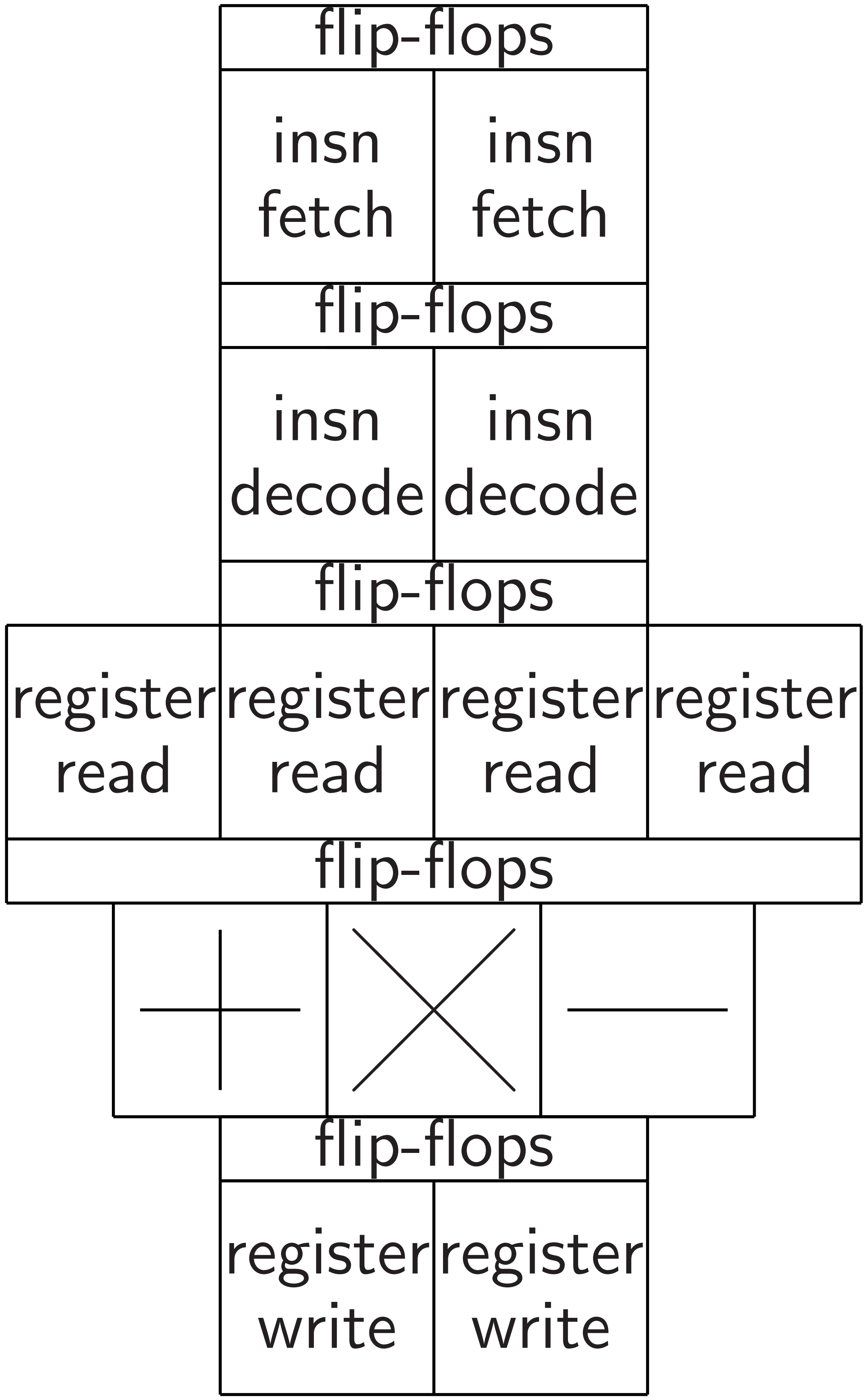



“Vector” processing:

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:





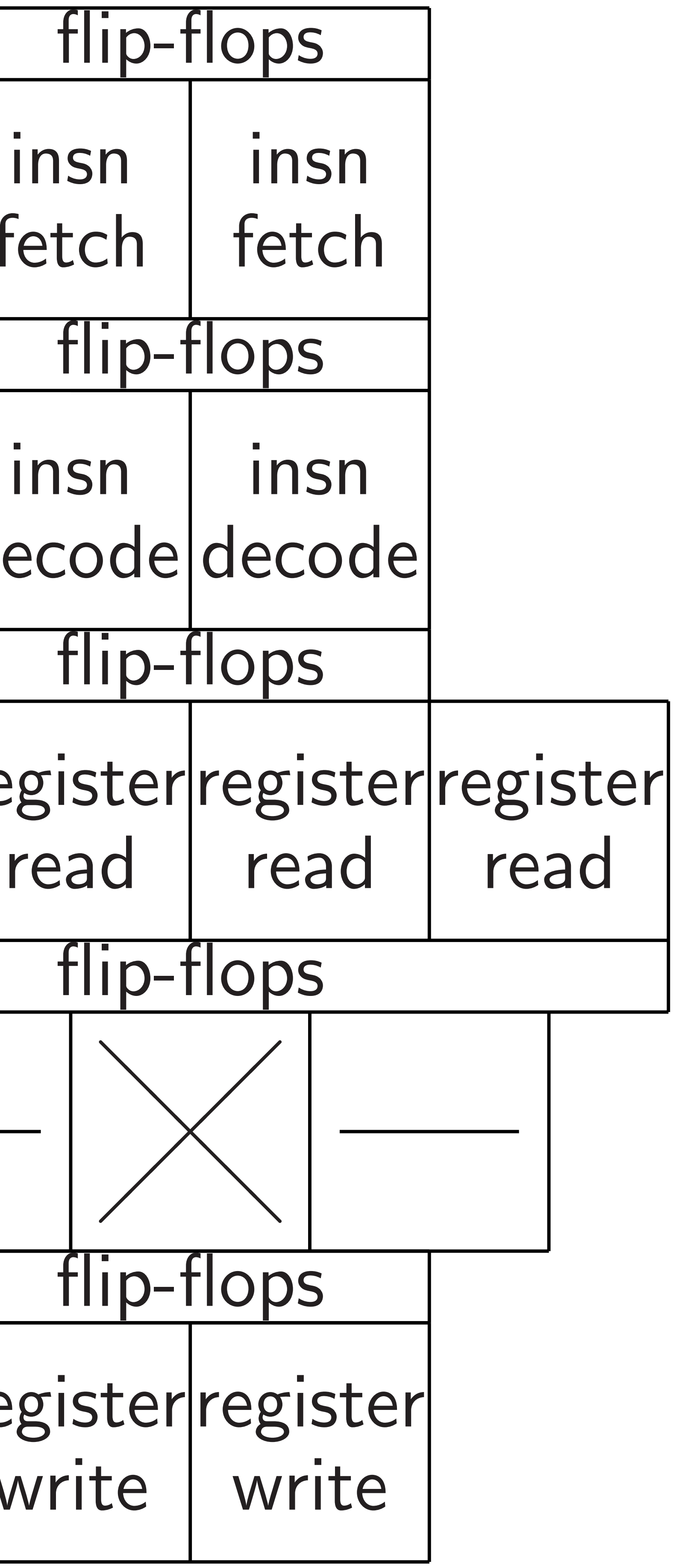

“Vector” processing:

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.

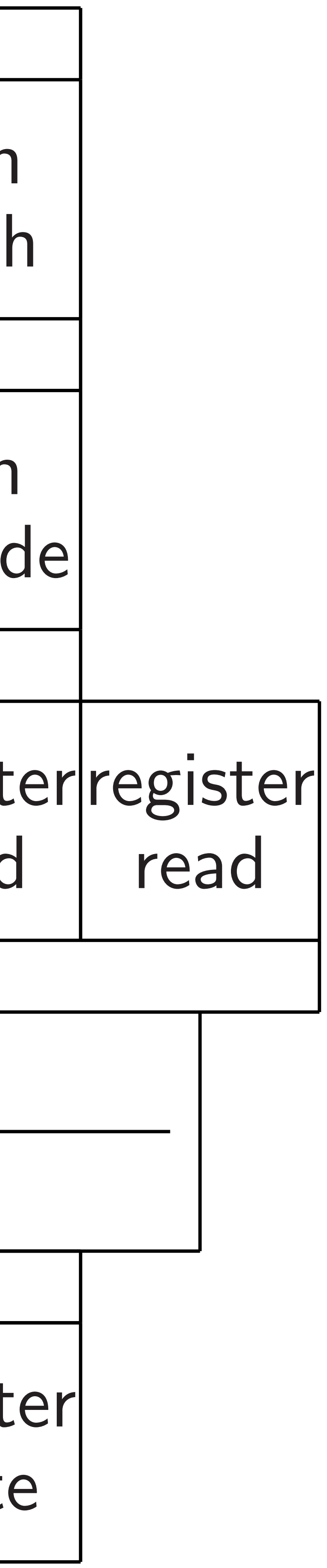
Network

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?

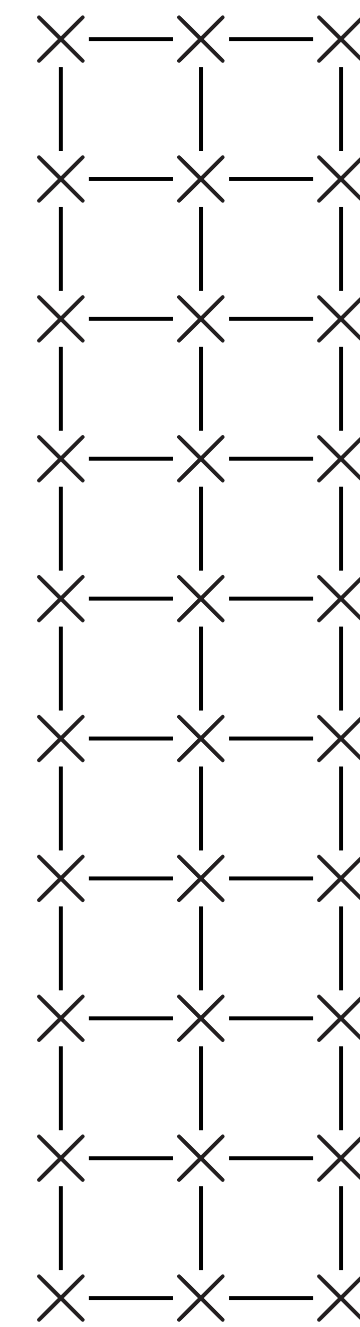
Input: array of $n$ numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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?

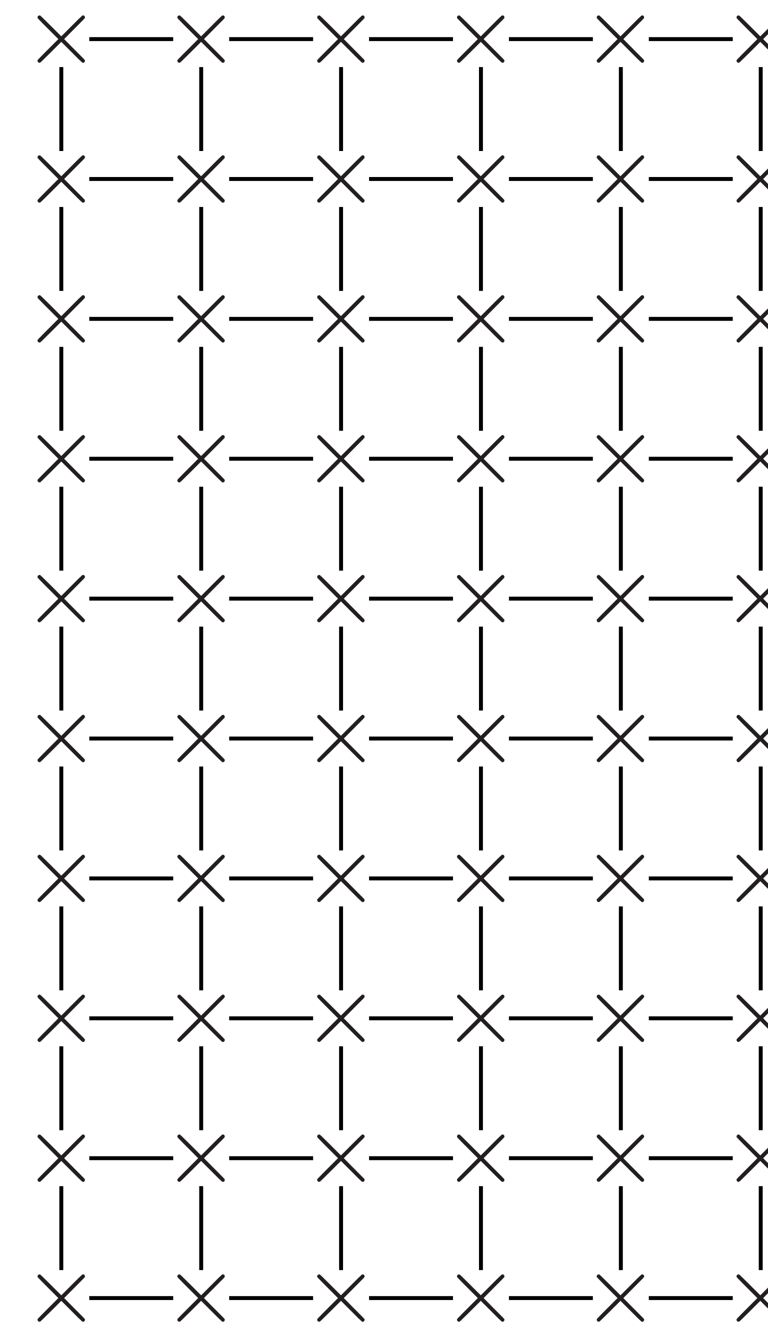
Input: array of $n$ numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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 $n$ s
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.

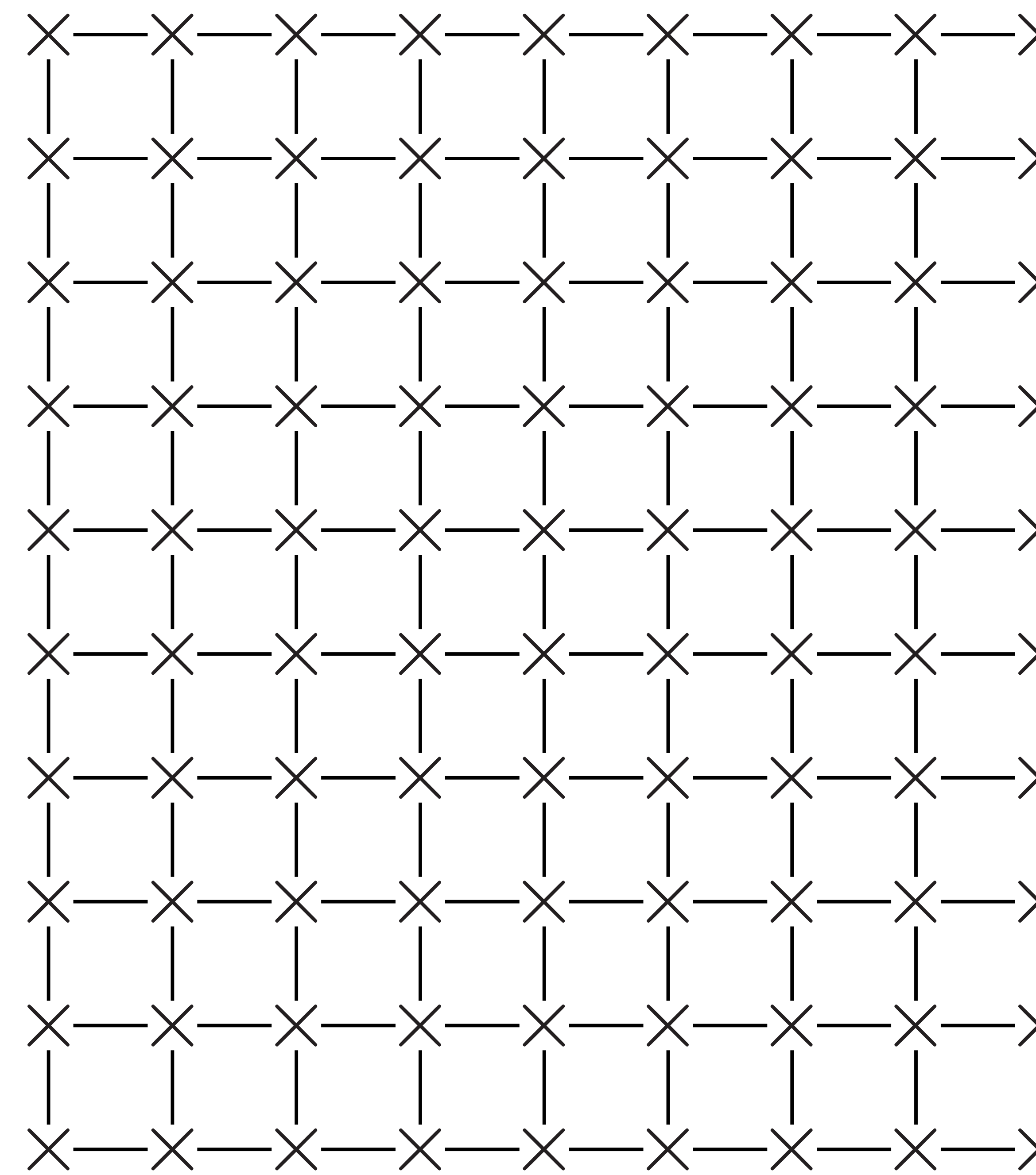
Output: array of $n$ numbers,
in increasing order,
represented in binary;
same multiset as input.

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.

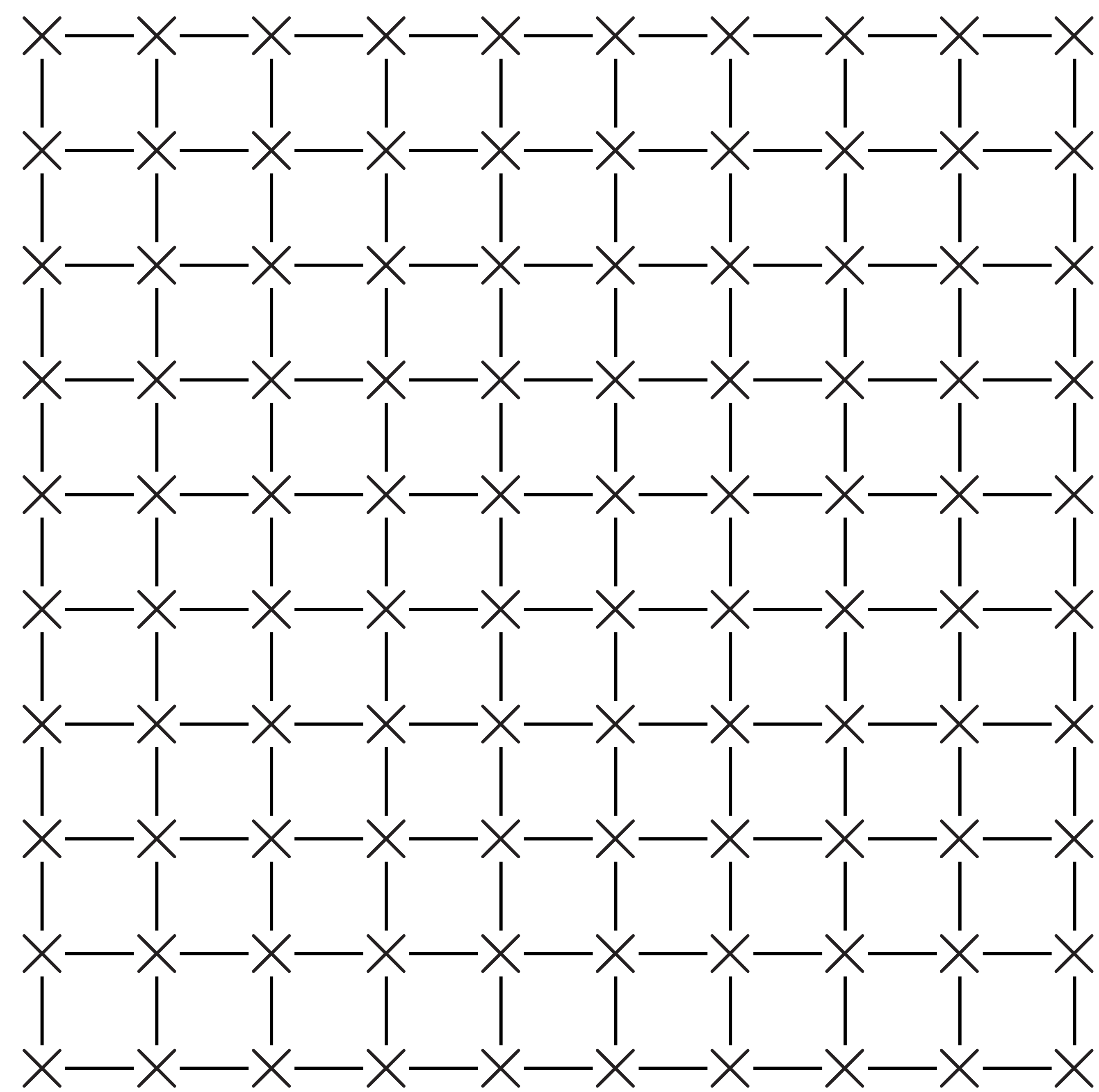
Output: array of $n$ numbers,
in increasing order,
represented in binary;
same multiset as input.

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.

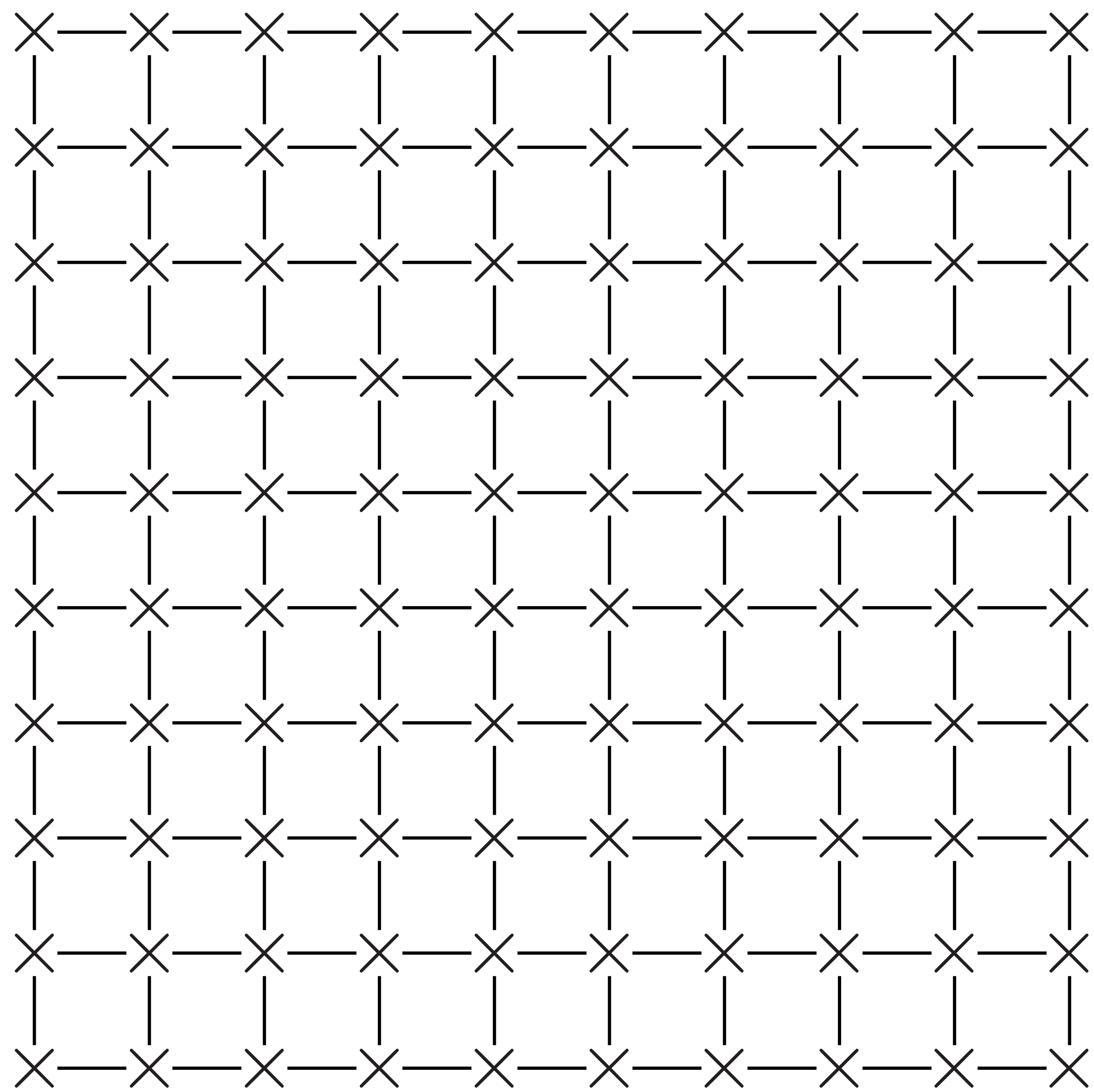
array of $n$ numbers,
nsing order,
ted in binary;
ultiset as input.

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 4
- Sort a
  1 3 1 4
  1 1 3 4
- Repea
  equals

the mesh

sorting?

numbers.

1, 2, . . . , $n^2$},

ary.

n numbers,

,

ary;

nput.

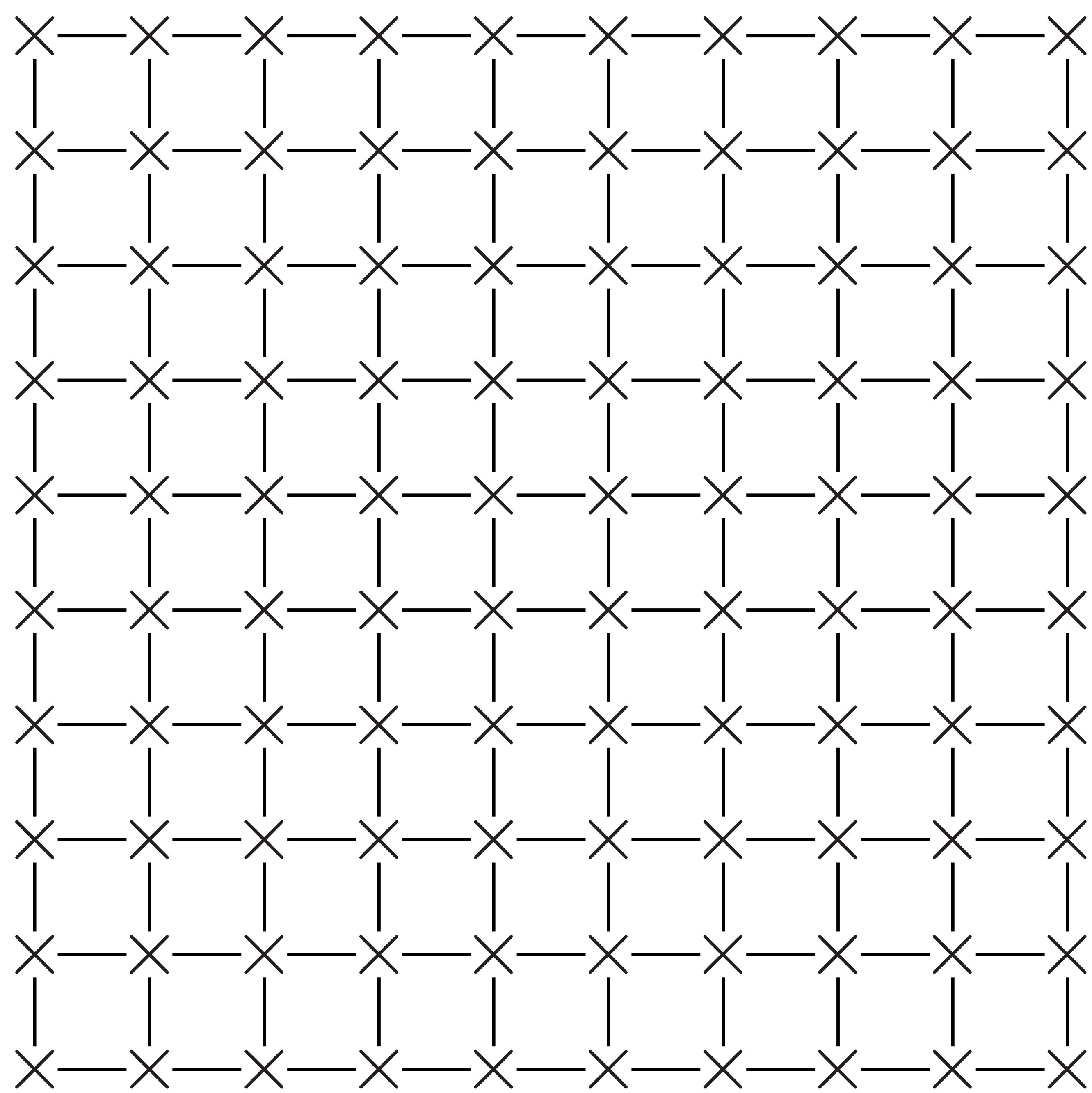
sed by

$+o(1)$.

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 ir
  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 nur
  equals row lengt

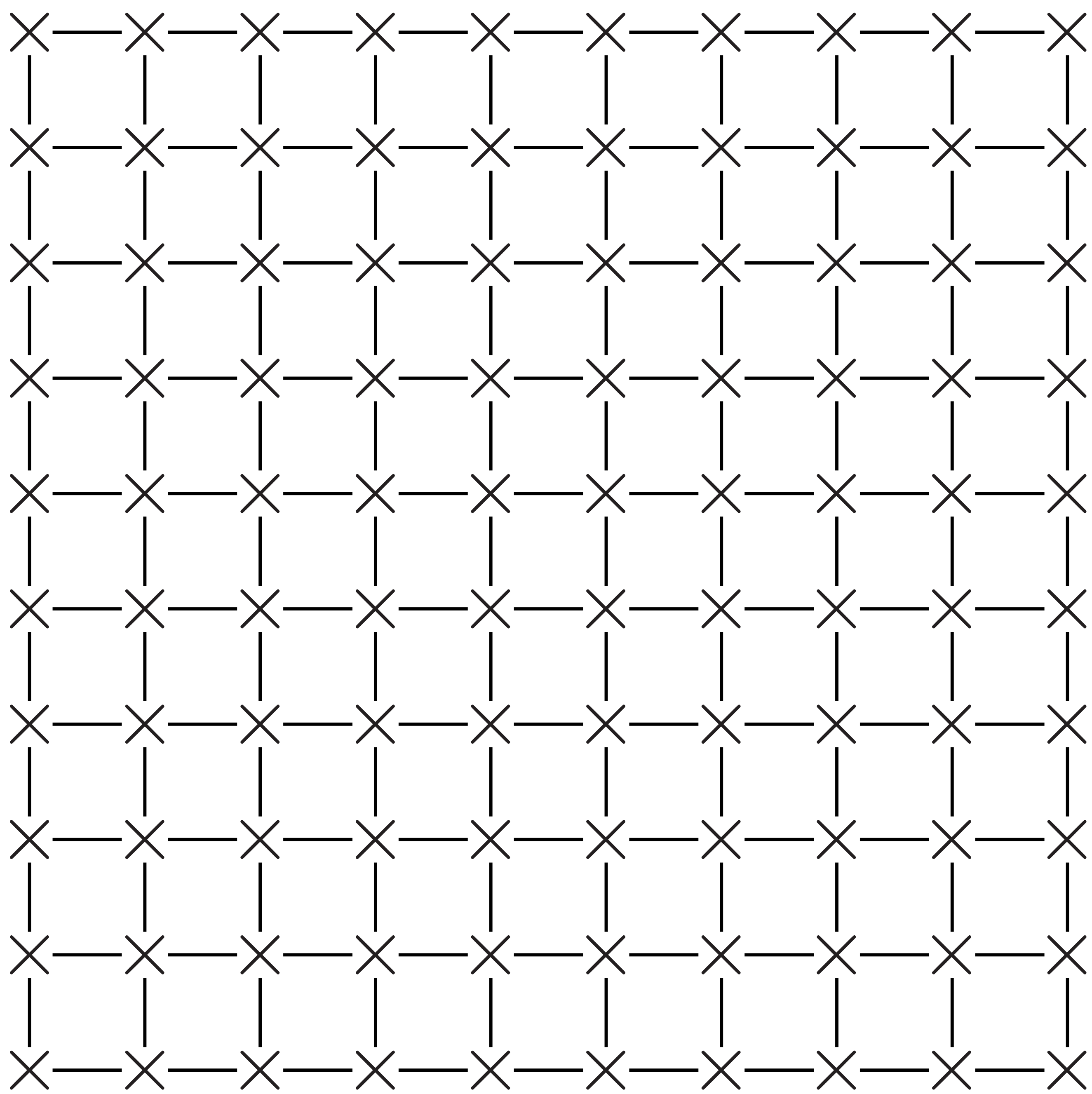
$^2\}$,

,

$k$ .



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





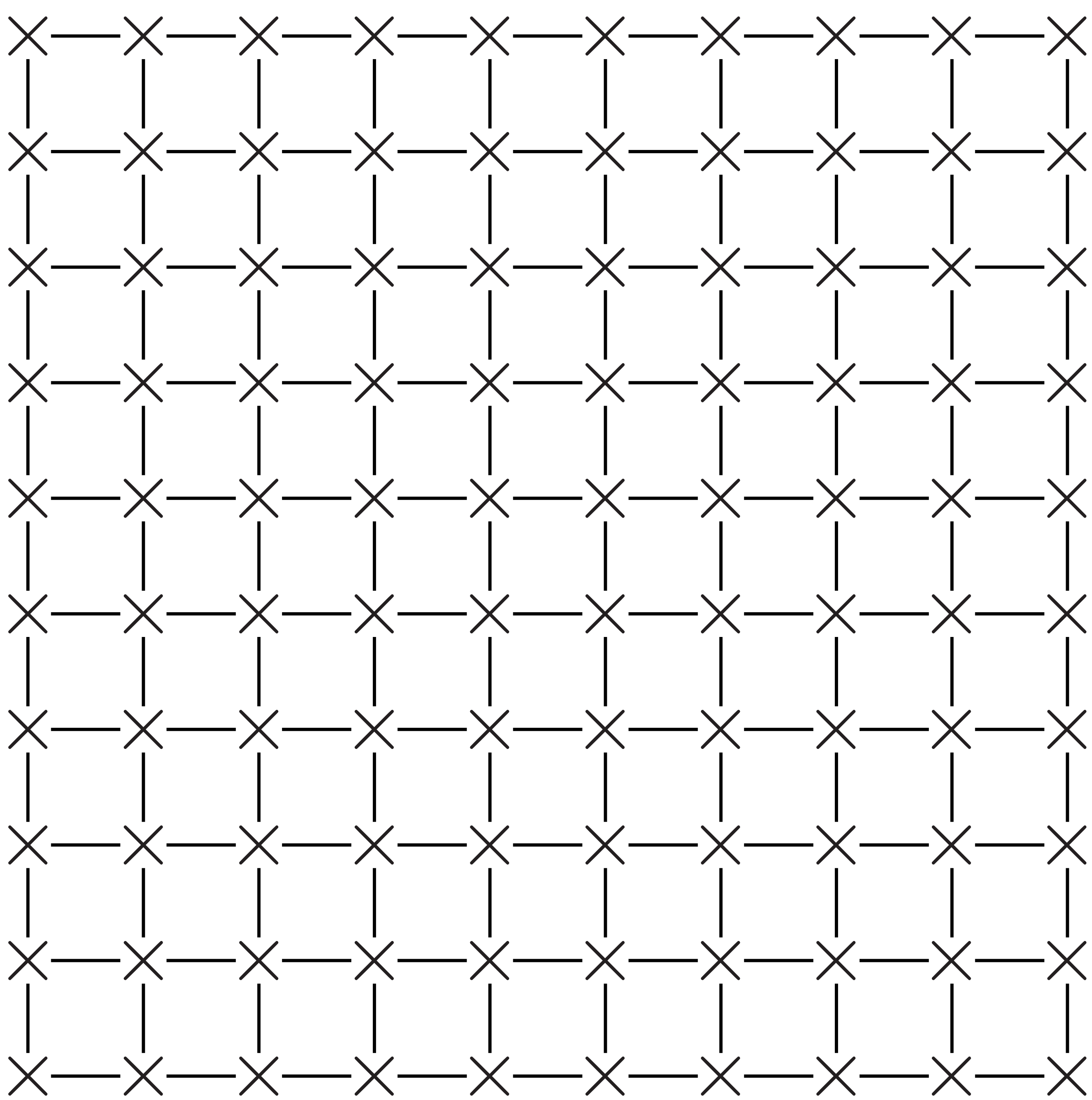
Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- 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 st
  equals row length.

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





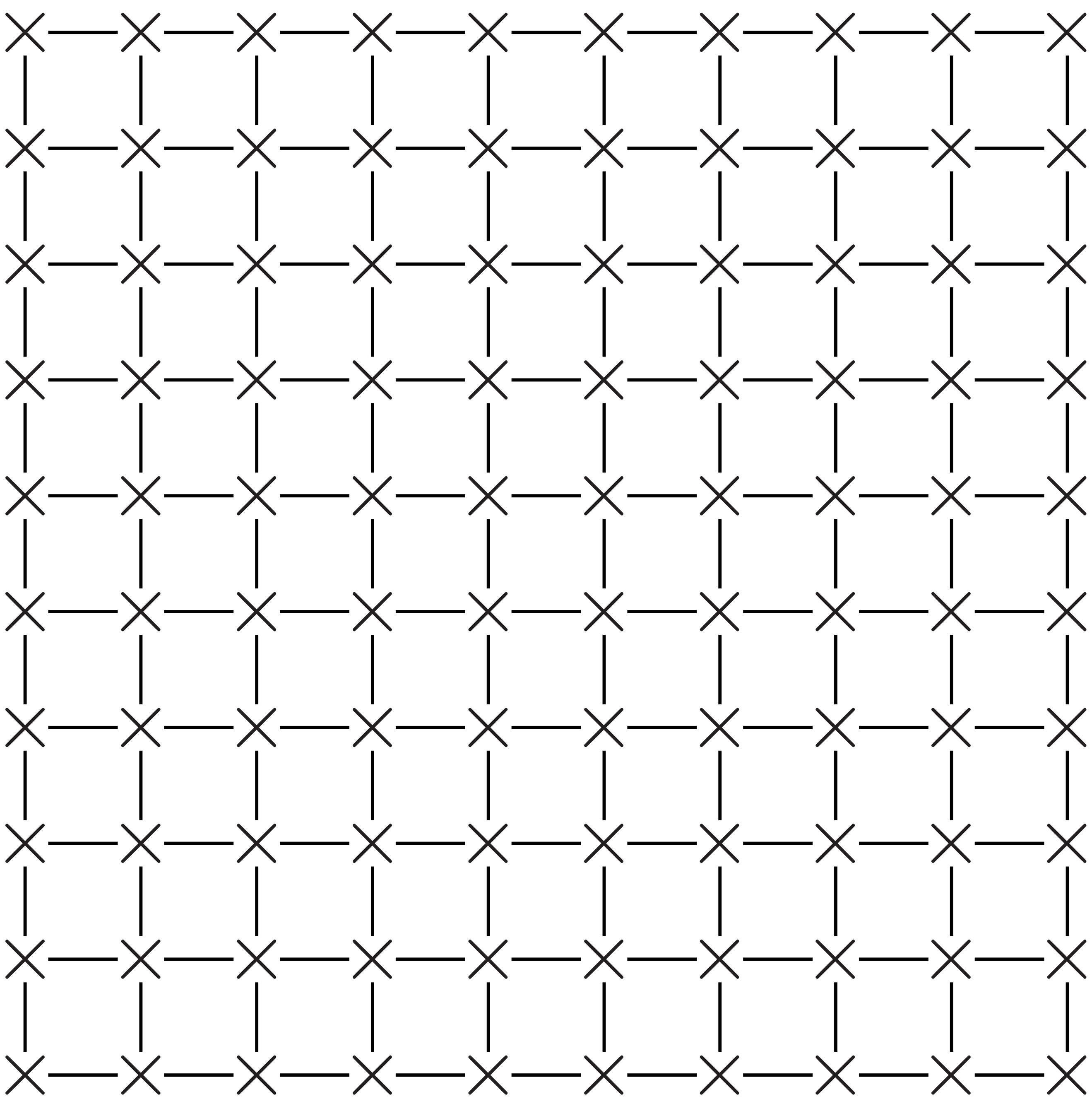
Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- 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:



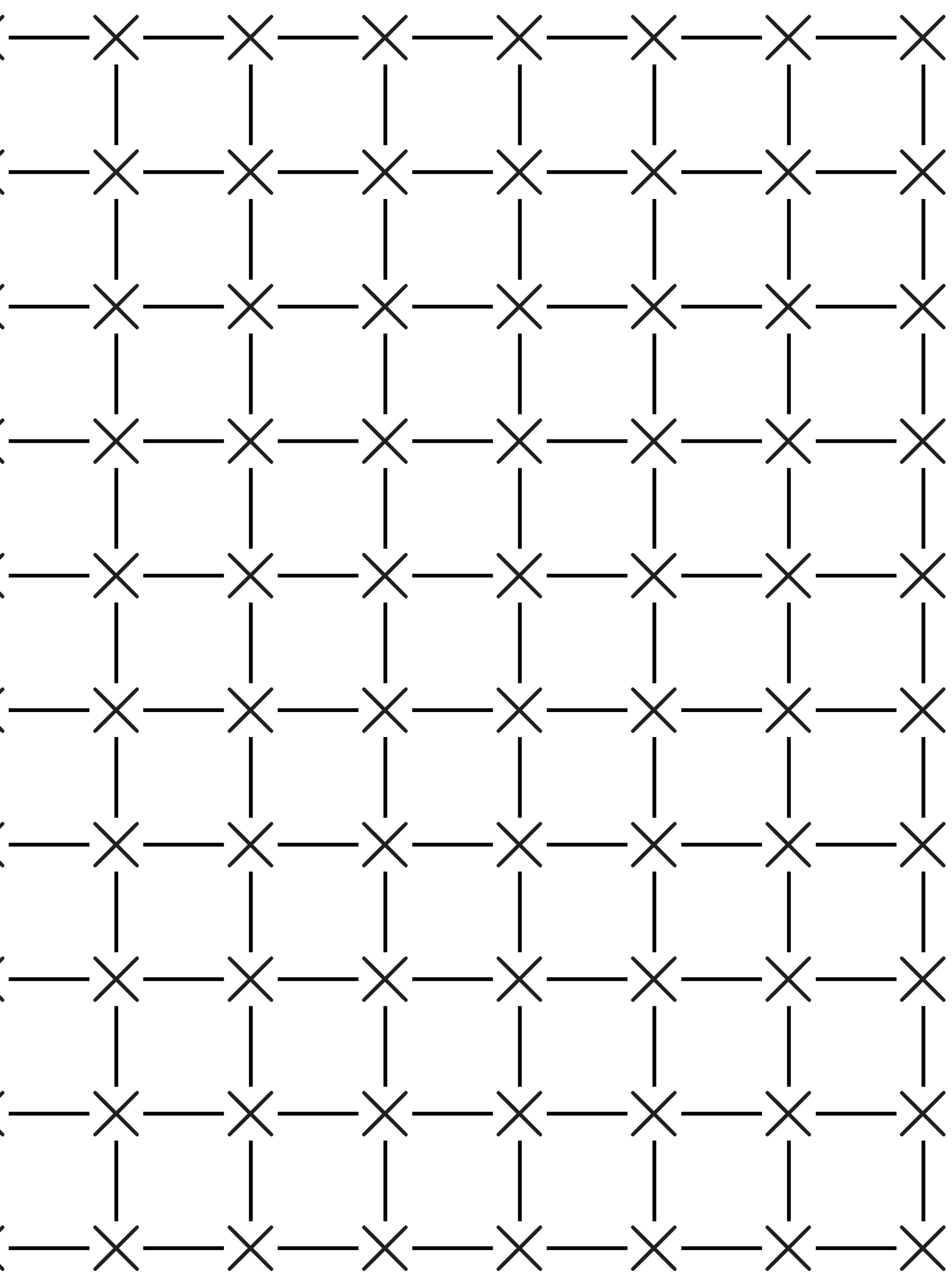


Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

- 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
mesh of $n$ small cells,
area $n^{o(1)}$,
r-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 e
- Sort e
- Sort e
- Sort e

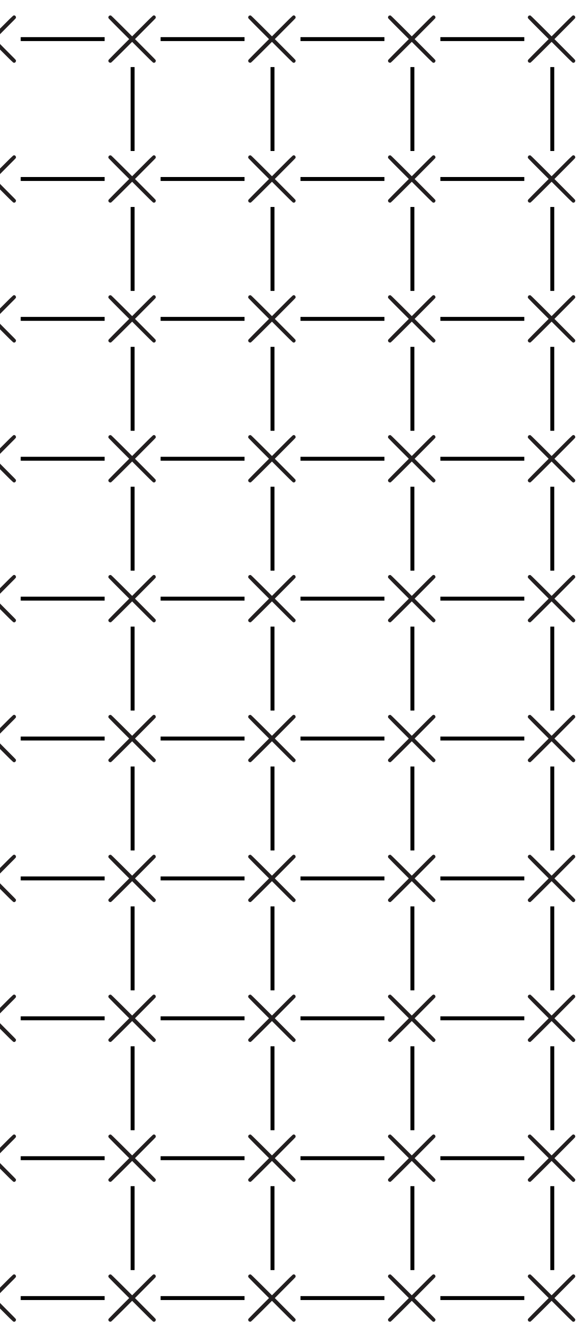
With pro
left-to-ri
for each
that this

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.

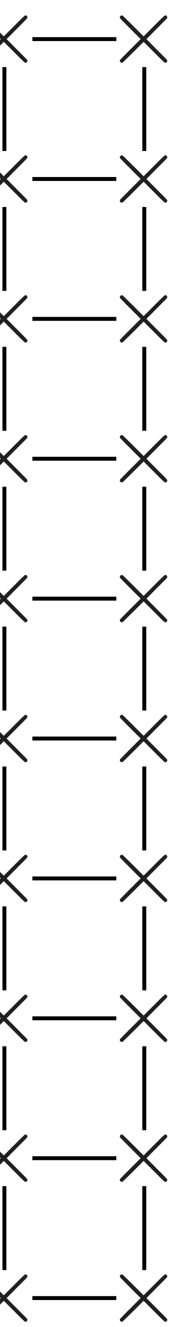
Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

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.

1 5 9 2 6 $\mapsto$

4 5 9 2 6

lternate pairs in parallel.

4 5 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 exan
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 $8 \times 8$ 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 $8 \times 8$ 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 $8 \times 8$ 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 $8 \times 8$ 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 →,

| | | |
|---|---|---|
| 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

ole array.



For example, assume that this $8 \times 8$ 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 $8 \times 8$ 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 $8 \times 8$ 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:

| | | | | | |
|---|---|---|---|---|---|
| 4 | 1 | 5 | 9 | 2 | 6 |
| 5 | 8 | 9 | 7 | 9 | 3 |
| 3 | 4 | 6 | 2 | 6 | 4 |
| 3 | 3 | 2 | 7 | 9 | 5 |
| 3 | 8 | 4 | 1 | 9 | 7 |
| 9 | 3 | 9 | 9 | 3 | 7 |
| 0 | 5 | 8 | 2 | 0 | 9 |
| 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 →, bottom ←:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

ely sort quadrants,

bottom ←:

| | | | | | |
|---|---|---|---|---|---|
| 2 | 3 | 2 | 2 | 2 | 3 |
| 3 | 3 | 4 | 5 | 5 | 6 |
| 4 | 5 | 6 | 6 | 7 | 7 |
| 3 | 8 | 9 | 9 | 9 | 9 |
| 0 | 0 | 2 | 2 | 1 | 0 |
| 3 | 2 | 5 | 4 | 4 | 3 |
| 5 | 5 | 9 | 8 | 7 | 7 |
| 3 | 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 ←, →:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 ←, →:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 from subroutines into high-level algorithm design.