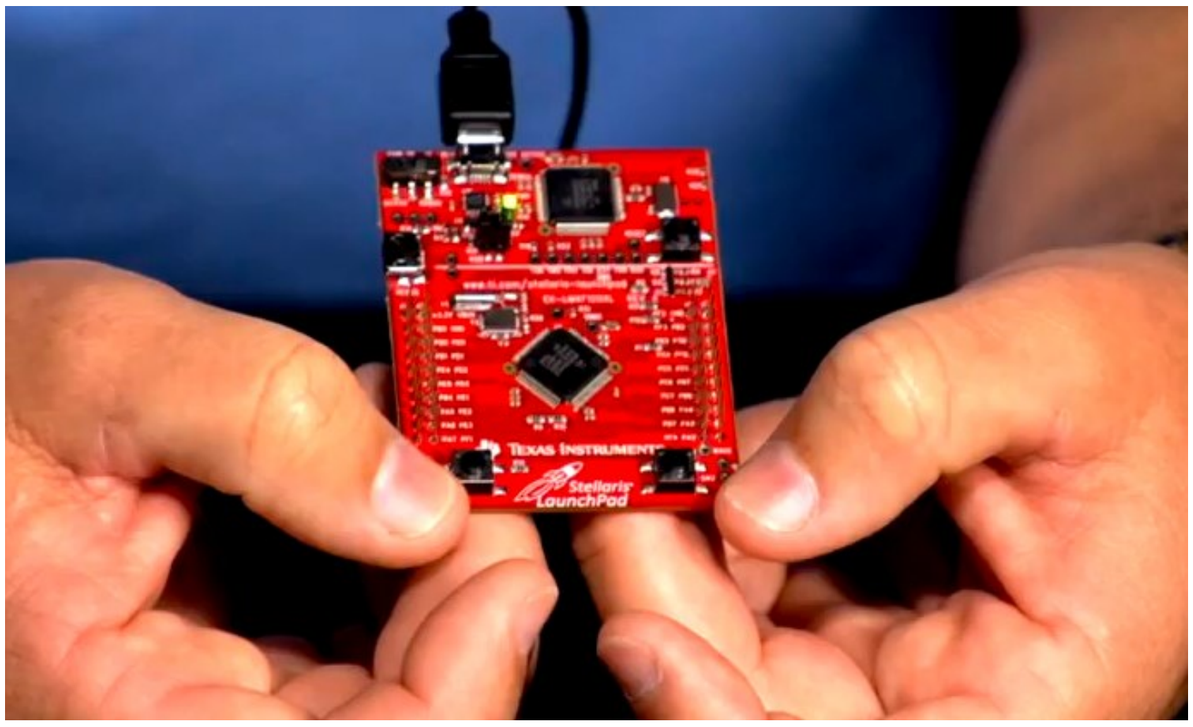
CPU-specific optimization

Example of a target CPU core:
ARM Cortex-M4F core inside
LM4F120H5QR microcontroller
in Stellaris LM4F120 Launchpad.



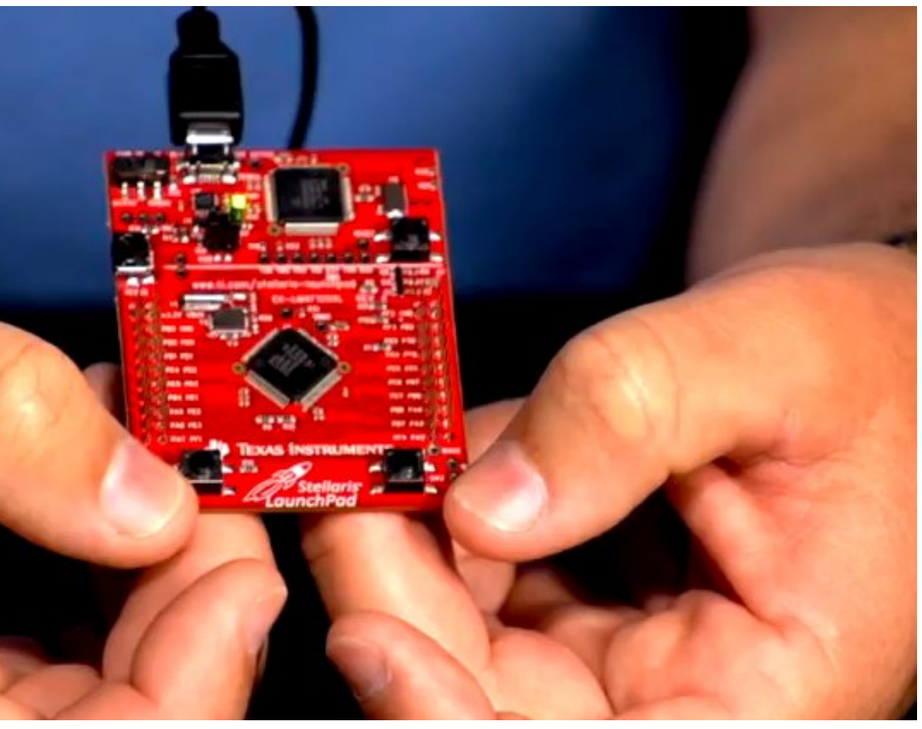
Example of a function
that we want to optimize:
adding 1000 integers mod 2^{32}.

Reference implementation:

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

ecific optimization

e of a target CPU core:
ortex-M4F core inside
20H5QR microcontroller
ris LM4F120 Launchpad.



Example of a function
that we want to optimize:
adding 1000 integers mod 2^{32}.

Reference implementation:

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

Counting

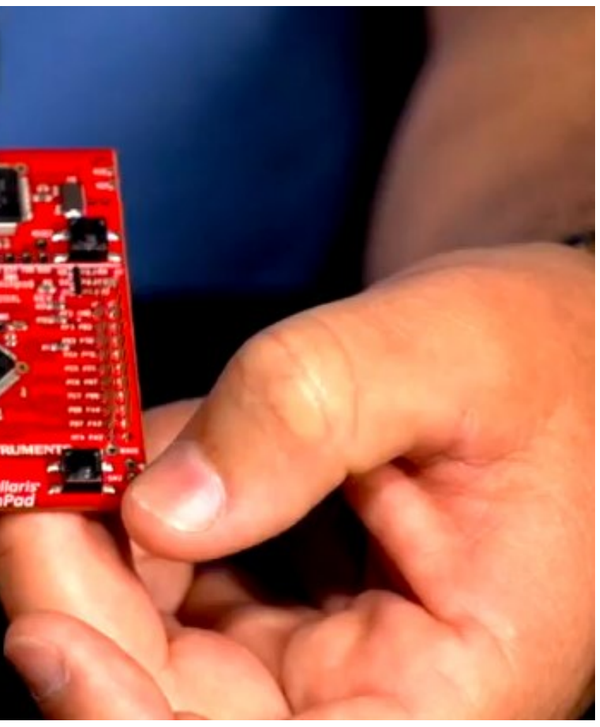
```
static
  *cons
  = (vo
...

int bef
int res
int aft
UARTpri
  resul
```

Output
Change

mization

et CPU core:
core inside
icrocontroller
20 Launchpad.



Example of a function
that we want to optimize:
adding 1000 integers mod 2^{32}.

Reference implementation:

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


Counting cycles:

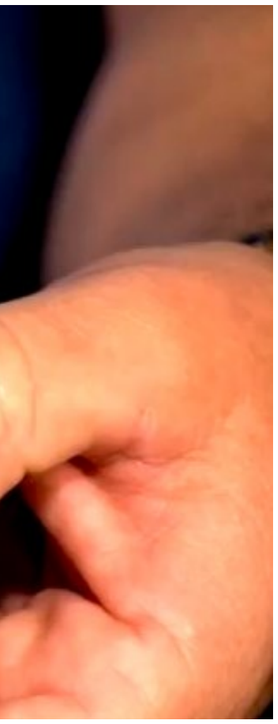
```
static volatile
  *const DWT_CYC
  = (void *) 0xE
...

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

Output shows 801

Change 1000 to 5

ore:
e
oller
npad.



Example of a function
that we want to optimize:
adding 1000 integers mod 2^{32}.

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.

Example of a function
that we want to optimize:
adding 1000 integers mod 2^{32}.

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.

e of a function
want to optimize:
1000 integers mod 2^{32}.

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

tion

ptimize:

ers mod 2^{32}.

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 microcont

really this slow at

32

)

2

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.

3

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

Try `-Os`: 8012 cycles.

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

Try `-Os`: 8012 cycles.

Try `-O1`: 8012 cycles.

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

Try `-Os`: 8012 cycles.

Try `-O1`: 8012 cycles.

Try `-O2`: 8012 cycles.

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

Try `-Os`: 8012 cycles.

Try `-O1`: 8012 cycles.

Try `-O2`: 8012 cycles.

Try `-O3`: 8012 cycles.

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

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.
Try `-O3`: 8012 cycles.

Try mov

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

```
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 approach:
Apply random "optimizations"
(and tweak compiler options)
until you get bored/frustrated.
Keep the fastest results.

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.
Try `-O3`: 8012 cycles.

Try moving the po

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

```
int

NT;

T;

um);
```


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

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

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.
Try `-O3`: 8012 cycles.

Try moving the pointer:

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

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

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

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.
Try `-O3`: 8012 cycles.

Try moving the pointer:

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

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

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

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.
Try `-O3`: 8012 cycles.

Try moving the pointer:

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

8010 cycles.

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

proach:
ndom "optimizations"
eak compiler options)
u get bored/frustrated.
e fastest results.

: 8012 cycles.
: 8012 cycles.
: 8012 cycles.
: 8012 cycles.

Try moving the pointer:

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

8010 cycles.

Try coun

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

er addition.
trollers
addition?"

otimizations"
ler options)
d/frustrated.
esults.

cles.
cles.
cles.
cles.

4

Try moving the pointer:

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

8010 cycles.

5

Try counting down

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

.

,

ns"

s)

ed.

Try moving the pointer:

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

8010 cycles.

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i
    result += *x++;
  return result;
}
```

Try moving the pointer:

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

8010 cycles.

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

Try moving the pointer:

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

8010 cycles.

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

ving the pointer:

```
(int *x)


esult = 0;

;

i = 0;i < 1000;++i)

ult += *x++;

n result;
```

cles.

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try usin

```
int sum
{
  int re
  int *
  while
    resu
  retur
}
```

pinter:

```
;

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


Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try using an end p

```
int sum(int *x)
{
  int result = 0
  int *y = x + 1
  while (x != y)
    result += *x
  return result;
}
```

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

nting down:

```
(int *x)


esult = 0;

;

i = 1000;i > 0;--i)

ult += *x++;

n result;
```

cles.

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

Back to

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

n:

```
;


i > 0;--i)
++;
```


Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

Back to original. T

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

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

Back to original. Try unrolli

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i +
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

5016 cycles.

g an end pointer:

```
(int *x)


esult = 0;
y = x + 1000;
 (x != y)
ult += *x++;
n result;
```

cles.

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

5016 cycles.

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

pointer:

```
;
000;

++;
```


Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

5016 cycles.

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

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

5016 cycles.


```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i +
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

5016 cycles.


```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

original. Try unrolling:

```
(int *x)


esult = 0;
;
i = 0;i < 1000;i += 2) {
ult += x[i];
ult += x[i + 1];


n result;
```

cles.

8

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

9

4016 cyc

Try unrolling:

```
;


 1000;i += 2) {
i];
i + 1];
```



```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

4016 cycles. Are v

ng:

```
= 2) {
```

8

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

9

4016 cycles. Are we done n

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```


4016 cycles. Are we done now?

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```


4016 cycles. Are we done now?

Most random "optimizations"
that we tried seem useless.
Can spend time trying more.
Does frustration level tell us
that we're close to optimal?

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```


4016 cycles. Are we done now?

Most random "optimizations"
that we tried seem useless.
Can spend time trying more.
Does frustration level tell us
that we're close to optimal?

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

Let's try this approach.

```
(int *x)


esult = 0;
;
i = 0;i < 1000;i += 5) {
ult += x[i];
ult += x[i + 1];
ult += x[i + 2];
ult += x[i + 3];
ult += x[i + 4];

n result;
```


4016 cycles. Are we done now?

Most random "optimizations"
that we tried seem useless.
Can spend time trying more.
Does frustration level tell us
that we're close to optimal?

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

Let's try this approach.

Find "A
Technica
Rely on
M4F =

Manual
"implem
architect

Points t
Architec
which de
e.g., "Al

First ma
ADD tal

```
;


 1000;i += 5) {
i];
i + 1];
i + 2];
i + 3];
i + 4];
```


4016 cycles. Are we done now?

Most random "optimizations"
that we tried seem useless.
Can spend time trying more.
Does frustration level tell us
that we're close to optimal?

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

Let's try this approach.

Find "ARM Cortex
Technical Referenc
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

```
= 5) {
```


4016 cycles. Are we done now?

Most random "optimizations"
that we tried seem useless.
Can spend time trying more.
Does frustration level tell us
that we're close to optimal?

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

Let's try this approach.

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.

4016 cycles. Are we done now?

Most random "optimizations"
that we tried seem useless.
Can spend time trying more.
Does frustration level tell us
that we're close to optimal?

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

Let's try this approach.

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.

cles. Are we done now?

ndom "optimizations"
tried seem useless.
nd time trying more.
ustration level tell us
re close to optimal?

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

this approach.

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 loa
Manual
a note a
Then mo
instructi
address
then it s

we done now?

timizations"
n useless.
ying more.
evel tell us
o optimal?

ound for
ithmetic etc.
etween
bserved time.

oach.

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 "pip
Then more explan
instruction is also
address not based
then it saves 1 cyc

ow?

s"

.

tc.

me.

10

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.

11

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 (with
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 r
pipelined

Can ach
in other
but noth

Lower b
$2n+1$ c
n cycles

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 LDR
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, incl
n cycles of arithme

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 arithmetic.

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
urpose "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`.

2281 cyc

```
y = x +
p = x
result =

loop:

  xi9 =
  xi8 =
  xi7 =
  xi6 =
  xi5 =
  xi4 =
  xi3 =
  xi2 =
```

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

array needs to
a register.

tion: LDR.
cles 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`.

2281 cycles using

```
y = x + 4000
p = x
result = 0

loop:

  xi9 = *(uint32
  xi8 = *(uint32
  xi7 = *(uint32
  xi6 = *(uint32
  xi5 = *(uint32
  xi4 = *(uint32
  xi3 = *(uint32
  xi2 = *(uint32
```

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`.

2281 cycles using `ldr.w`:

```
y = x + 4000
p = x
result = 0


loop:


  xi9 = *(uint32 *) (p +
  xi8 = *(uint32 *) (p +
  xi7 = *(uint32 *) (p +
  xi6 = *(uint32 *) (p +
  xi5 = *(uint32 *) (p +
  xi4 = *(uint32 *) (p +
  xi3 = *(uint32 *) (p +
  xi2 = *(uint32 *) (p +
```

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`.

2281 cycles using `ldr.w`:

```
y = x + 4000
p = x
result = 0


loop:


  xi9 = *(uint32 *) (p + 76)
  xi8 = *(uint32 *) (p + 72)
  xi7 = *(uint32 *) (p + 68)
  xi6 = *(uint32 *) (p + 64)
  xi5 = *(uint32 *) (p + 60)
  xi4 = *(uint32 *) (p + 56)
  xi3 = *(uint32 *) (p + 52)
  xi2 = *(uint32 *) (p + 48)
```

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

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

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

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

2281 cycles using `ldr.w`:

```
y = x + 4000
p = x
result = 0


loop:


  xi9 = *(uint32 *) (p + 76)
  xi8 = *(uint32 *) (p + 72)
  xi7 = *(uint32 *) (p + 68)
  xi6 = *(uint32 *) (p + 64)
  xi5 = *(uint32 *) (p + 60)
  xi4 = *(uint32 *) (p + 56)
  xi3 = *(uint32 *) (p + 52)
  xi2 = *(uint32 *) (p + 48)
```

```
  xi1 =
  xi0 =
  result
  result
  result
  result
  result
  result
  result
  result
  result
  xi9 =
  xi8 =
  xi7 =
```

Rs

ycles

DRs can be

").

peed

RD, LDM)

s faster.

n LDR + n ADD:

uding

etic.

e is higher:

DRs;

ting i.

2281 cycles using `ldr.w`:

```
y = x + 4000
p = x
result = 0

loop:

  xi9 = *(uint32 *) (p + 76)
  xi8 = *(uint32 *) (p + 72)
  xi7 = *(uint32 *) (p + 68)
  xi6 = *(uint32 *) (p + 64)
  xi5 = *(uint32 *) (p + 60)
  xi4 = *(uint32 *) (p + 56)
  xi3 = *(uint32 *) (p + 52)
  xi2 = *(uint32 *) (p + 48)
```

```
  xi1 = *(uint32
  xi0 = *(uint32
  result += xi9
  result += xi8
  result += xi7
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0
  xi9 = *(uint32
  xi8 = *(uint32
  xi7 = *(uint32
```

2281 cycles using `ldr.w`:

```
y = x + 4000
p = x
result = 0

loop:

  xi9 = *(uint32 *) (p + 76)
  xi8 = *(uint32 *) (p + 72)
  xi7 = *(uint32 *) (p + 68)
  xi6 = *(uint32 *) (p + 64)
  xi5 = *(uint32 *) (p + 60)
  xi4 = *(uint32 *) (p + 56)
  xi3 = *(uint32 *) (p + 52)
  xi2 = *(uint32 *) (p + 48)
```



```
  xi1 = *(uint32 *) (p +
  xi0 = *(uint32 *) (p +
  result += xi9
  result += xi8
  result += xi7
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0
  xi9 = *(uint32 *) (p +
  xi8 = *(uint32 *) (p +
  xi7 = *(uint32 *) (p +
```

2281 cycles using `ldr.w`:

```
y = x + 4000
p = x
result = 0

loop:

  xi9 = *(uint32 *) (p + 76)
  xi8 = *(uint32 *) (p + 72)
  xi7 = *(uint32 *) (p + 68)
  xi6 = *(uint32 *) (p + 64)
  xi5 = *(uint32 *) (p + 60)
  xi4 = *(uint32 *) (p + 56)
  xi3 = *(uint32 *) (p + 52)
  xi2 = *(uint32 *) (p + 48)
```



```
  xi1 = *(uint32 *) (p + 44)
  xi0 = *(uint32 *) (p + 40)
  result += xi9
  result += xi8
  result += xi7
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0
  xi9 = *(uint32 *) (p + 36)
  xi8 = *(uint32 *) (p + 32)
  xi7 = *(uint32 *) (p + 28)
```

cles using `ldr.w`:

```
 4000


= 0




 *(uint32 *) (p + 76)
*(uint32 *) (p + 72)
*(uint32 *) (p + 68)
*(uint32 *) (p + 64)
*(uint32 *) (p + 60)
*(uint32 *) (p + 56)
*(uint32 *) (p + 52)
*(uint32 *) (p + 48)
```



```
xi1 = *(uint32 *) (p + 44)
xi0 = *(uint32 *) (p + 40)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi9 = *(uint32 *) (p + 36)
xi8 = *(uint32 *) (p + 32)
xi7 = *(uint32 *) (p + 28)
```

```
xi6 =
xi5 =
xi4 =
xi3 =
xi2 =
xi1 =
xi0 =
resul
resul
resul
resul
resul
resul
resul
```

```
ldr.w:

 *) (p + 76)
*) (p + 72)
*) (p + 68)
*) (p + 64)
*) (p + 60)
*) (p + 56)
*) (p + 52)
*) (p + 48)
```



```
    xi1 = *(uint32 *) (p + 44)
    xi0 = *(uint32 *) (p + 40)
    result += xi9
    result += xi8
    result += xi7
    result += xi6
    result += xi5
    result += xi4
    result += xi3
    result += xi2
    result += xi1
    result += xi0
    xi9 = *(uint32 *) (p + 36)
    xi8 = *(uint32 *) (p + 32)
    xi7 = *(uint32 *) (p + 28)
```

```
    xi6 = *(uint32
    xi5 = *(uint32
    xi4 = *(uint32
    xi3 = *(uint32
    xi2 = *(uint32
    xi1 = *(uint32
    xi0 = *(uint32
    result += xi9
    result += xi8
    result += xi7
    result += xi6
    result += xi5
    result += xi4
    result += xi3
    result += xi2
```

```
76)
72)
68)
64)
60)
56)
52)
48)
```

```
    xi1 = *(uint32 *) (p + 44)
    xi0 = *(uint32 *) (p + 40)
    result += xi9
    result += xi8
    result += xi7
    result += xi6
    result += xi5
    result += xi4
    result += xi3
    result += xi2
    result += xi1
    result += xi0
    xi9 = *(uint32 *) (p + 36)
    xi8 = *(uint32 *) (p + 32)
    xi7 = *(uint32 *) (p + 28)
```

```
    xi6 = *(uint32 *) (p +
    xi5 = *(uint32 *) (p +
    xi4 = *(uint32 *) (p +
    xi3 = *(uint32 *) (p +
    xi2 = *(uint32 *) (p +
    xi1 = *(uint32 *) (p +
    xi0 = *(uint32 *) p; p
    result += xi9
    result += xi8
    result += xi7
    result += xi6
    result += xi5
    result += xi4
    result += xi3
    result += xi2
```

```
xi1 = *(uint32 *) (p + 44)
xi0 = *(uint32 *) (p + 40)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi9 = *(uint32 *) (p + 36)
xi8 = *(uint32 *) (p + 32)
xi7 = *(uint32 *) (p + 28)
```



```
xi6 = *(uint32 *) (p + 24)
xi5 = *(uint32 *) (p + 20)
xi4 = *(uint32 *) (p + 16)
xi3 = *(uint32 *) (p + 12)
xi2 = *(uint32 *) (p + 8)
xi1 = *(uint32 *) (p + 4)
xi0 = *(uint32 *) p; p += 160
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
```

```
 *(uint32 *) (p + 44)
 *(uint32 *) (p + 40)
t += xi9
t += xi8
t += xi7
t += xi6
t += xi5
t += xi4
t += xi3
t += xi2
t += xi1
t += xi0
 *(uint32 *) (p + 36)
 *(uint32 *) (p + 32)
 *(uint32 *) (p + 28)
```



```
    xi6 = *(uint32 *) (p + 24)
    xi5 = *(uint32 *) (p + 20)
    xi4 = *(uint32 *) (p + 16)
    xi3 = *(uint32 *) (p + 12)
    xi2 = *(uint32 *) (p + 8)
    xi1 = *(uint32 *) (p + 4)
    xi0 = *(uint32 *) p; p += 160
    result += xi9
    result += xi8
    result += xi7
    result += xi6
    result += xi5
    result += xi4
    result += xi3
    result += xi2
```

```
    result
    result
    xi9 =
    xi8 =
    xi7 =
    xi6 =
    xi5 =
    xi4 =
    xi3 =
    xi2 =
    xi1 =
    xi0 =
    resul
    result
    result
```

```
*) (p + 44)
*) (p + 40)
*) (p + 36)
*) (p + 32)
*) (p + 28)
```



```
  xi6 = *(uint32 *) (p + 24)
  xi5 = *(uint32 *) (p + 20)
  xi4 = *(uint32 *) (p + 16)
  xi3 = *(uint32 *) (p + 12)
  xi2 = *(uint32 *) (p + 8)
  xi1 = *(uint32 *) (p + 4)
  xi0 = *(uint32 *) p; p += 160
  result += xi9
  result += xi8
  result += xi7
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
```

```
  result += xi1
  result += xi0
  xi9 = *(uint32
  xi8 = *(uint32
  xi7 = *(uint32
  xi6 = *(uint32
  xi5 = *(uint32
  xi4 = *(uint32
  xi3 = *(uint32
  xi2 = *(uint32
  xi1 = *(uint32
  xi0 = *(uint32
  result += xi9
  result += xi8
  result += xi7
```

```
44)
40)
36)
32)
28)
```

15

```
    xi6 = *(uint32 *) (p + 24)
    xi5 = *(uint32 *) (p + 20)
    xi4 = *(uint32 *) (p + 16)
    xi3 = *(uint32 *) (p + 12)
    xi2 = *(uint32 *) (p + 8)
    xi1 = *(uint32 *) (p + 4)
    xi0 = *(uint32 *) p; p += 160
    result += xi9
    result += xi8
    result += xi7
    result += xi6
    result += xi5
    result += xi4
    result += xi3
    result += xi2
```

16

```
    result += xi1
    result += xi0
    xi9 = *(uint32 *) (p -
    xi8 = *(uint32 *) (p -
    xi7 = *(uint32 *) (p -
    xi6 = *(uint32 *) (p -
    xi5 = *(uint32 *) (p -
    xi4 = *(uint32 *) (p -
    xi3 = *(uint32 *) (p -
    xi2 = *(uint32 *) (p -
    xi1 = *(uint32 *) (p -
    xi0 = *(uint32 *) (p -
    result += xi9
    result += xi8
    result += xi7
```

```
xi6 = *(uint32 *) (p + 24)
xi5 = *(uint32 *) (p + 20)
xi4 = *(uint32 *) (p + 16)
xi3 = *(uint32 *) (p + 12)
xi2 = *(uint32 *) (p + 8)
xi1 = *(uint32 *) (p + 4)
xi0 = *(uint32 *) p; p += 160
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
```



```
result += xi1
result += xi0
xi9 = *(uint32 *) (p - 4)
xi8 = *(uint32 *) (p - 8)
xi7 = *(uint32 *) (p - 12)
xi6 = *(uint32 *) (p - 16)
xi5 = *(uint32 *) (p - 20)
xi4 = *(uint32 *) (p - 24)
xi3 = *(uint32 *) (p - 28)
xi2 = *(uint32 *) (p - 32)
xi1 = *(uint32 *) (p - 36)
xi0 = *(uint32 *) (p - 40)
result += xi9
result += xi8
result += xi7
```

16

```
 *(uint32 *) (p + 24)
 *(uint32 *) (p + 20)
 *(uint32 *) (p + 16)
 *(uint32 *) (p + 12)
 *(uint32 *) (p + 8)
 *(uint32 *) (p + 4)
 *(uint32 *) p; p += 160
t += xi9
t += xi8
t += xi7
t += xi6
t += xi5
t += xi4
t += xi3
t += xi2
```

17

```
   result += xi1
   result += xi0
   xi9 = *(uint32 *) (p - 4)
   xi8 = *(uint32 *) (p - 8)
   xi7 = *(uint32 *) (p - 12)
   xi6 = *(uint32 *) (p - 16)
   xi5 = *(uint32 *) (p - 20)
   xi4 = *(uint32 *) (p - 24)
   xi3 = *(uint32 *) (p - 28)
   xi2 = *(uint32 *) (p - 32)
   xi1 = *(uint32 *) (p - 36)
   xi0 = *(uint32 *) (p - 40)
   result += xi9
   result += xi8
   result += xi7
```

```
   resul
   resul
   resul
   resul
   resul
   resul
   resul
   xi9 =
   xi8 =
   xi7 =
   xi6 =
   xi5 =
   xi4 =
   xi3 =
   xi2 =
```

```
*) (p + 24)
*) (p + 20)
*) (p + 16)
*) (p + 12)
*) (p + 8)
*) (p + 4)
*) p; p += 160
```



```
  result += xi1
  result += xi0
  xi9 = *(uint32 *) (p - 4)
  xi8 = *(uint32 *) (p - 8)
  xi7 = *(uint32 *) (p - 12)
  xi6 = *(uint32 *) (p - 16)
  xi5 = *(uint32 *) (p - 20)
  xi4 = *(uint32 *) (p - 24)
  xi3 = *(uint32 *) (p - 28)
  xi2 = *(uint32 *) (p - 32)
  xi1 = *(uint32 *) (p - 36)
  xi0 = *(uint32 *) (p - 40)
  result += xi9
  result += xi8
  result += xi7
```

```
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0
  xi9 = *(uint32
  xi8 = *(uint32
  xi7 = *(uint32
  xi6 = *(uint32
  xi5 = *(uint32
  xi4 = *(uint32
  xi3 = *(uint32
  xi2 = *(uint32
```

```
24)
20)
16)
12)
8)
4)
+= 160
```

```
   result += xi1
   result += xi0
   xi9 = *(uint32 *) (p - 4)
   xi8 = *(uint32 *) (p - 8)
   xi7 = *(uint32 *) (p - 12)
   xi6 = *(uint32 *) (p - 16)
   xi5 = *(uint32 *) (p - 20)
   xi4 = *(uint32 *) (p - 24)
   xi3 = *(uint32 *) (p - 28)
   xi2 = *(uint32 *) (p - 32)
   xi1 = *(uint32 *) (p - 36)
   xi0 = *(uint32 *) (p - 40)
   result += xi9
   result += xi8
   result += xi7
```

```
   result += xi6
   result += xi5
   result += xi4
   result += xi3
   result += xi2
   result += xi1
   result += xi0
   xi9 = *(uint32 *) (p -
   xi8 = *(uint32 *) (p -
   xi7 = *(uint32 *) (p -
   xi6 = *(uint32 *) (p -
   xi5 = *(uint32 *) (p -
   xi4 = *(uint32 *) (p -
   xi3 = *(uint32 *) (p -
   xi2 = *(uint32 *) (p -
```

```
result += xi1
result += xi0
xi9 = *(uint32 *) (p - 4)
xi8 = *(uint32 *) (p - 8)
xi7 = *(uint32 *) (p - 12)
xi6 = *(uint32 *) (p - 16)
xi5 = *(uint32 *) (p - 20)
xi4 = *(uint32 *) (p - 24)
xi3 = *(uint32 *) (p - 28)
xi2 = *(uint32 *) (p - 32)
xi1 = *(uint32 *) (p - 36)
xi0 = *(uint32 *) (p - 40)
result += xi9
result += xi8
result += xi7
```



```
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi9 = *(uint32 *) (p - 44)
xi8 = *(uint32 *) (p - 48)
xi7 = *(uint32 *) (p - 52)
xi6 = *(uint32 *) (p - 56)
xi5 = *(uint32 *) (p - 60)
xi4 = *(uint32 *) (p - 64)
xi3 = *(uint32 *) (p - 68)
xi2 = *(uint32 *) (p - 72)
```

17

```
t += xi1
t += xi0
 *(uint32 *) (p - 4)
 *(uint32 *) (p - 8)
 *(uint32 *) (p - 12)
 *(uint32 *) (p - 16)
 *(uint32 *) (p - 20)
 *(uint32 *) (p - 24)
 *(uint32 *) (p - 28)
 *(uint32 *) (p - 32)
 *(uint32 *) (p - 36)
 *(uint32 *) (p - 40)
t += xi9
t += xi8
t += xi7
```

18

```
   result += xi6
   result += xi5
   result += xi4
   result += xi3
   result += xi2
   result += xi1
   result += xi0
   xi9 = *(uint32 *) (p - 44)
   xi8 = *(uint32 *) (p - 48)
   xi7 = *(uint32 *) (p - 52)
   xi6 = *(uint32 *) (p - 56)
   xi5 = *(uint32 *) (p - 60)
   xi4 = *(uint32 *) (p - 64)
   xi3 = *(uint32 *) (p - 68)
   xi2 = *(uint32 *) (p - 72)
```

```
   xi1 =
   xi0 =
   result
   result
   result
   result
   result
   result
   result
   result
   result
   result
 goto loo
```

17

```
*) (p - 4)
*) (p - 8)
*) (p - 12)
*) (p - 16)
*) (p - 20)
*) (p - 24)
*) (p - 28)
*) (p - 32)
*) (p - 36)
*) (p - 40)
```

```
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi9 = *(uint32 *) (p - 44)
xi8 = *(uint32 *) (p - 48)
xi7 = *(uint32 *) (p - 52)
xi6 = *(uint32 *) (p - 56)
xi5 = *(uint32 *) (p - 60)
xi4 = *(uint32 *) (p - 64)
xi3 = *(uint32 *) (p - 68)
xi2 = *(uint32 *) (p - 72)
```

18

```
xi1 = *(uint32
xi0 = *(uint32
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
                =?
goto loop if !=
```

```
4)
8)
12)
16)
20)
24)
28)
32)
36)
40)
```



```
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi9 = *(uint32 *) (p - 44)
xi8 = *(uint32 *) (p - 48)
xi7 = *(uint32 *) (p - 52)
xi6 = *(uint32 *) (p - 56)
xi5 = *(uint32 *) (p - 60)
xi4 = *(uint32 *) (p - 64)
xi3 = *(uint32 *) (p - 68)
xi2 = *(uint32 *) (p - 72)
```



```
  xi1 = *(uint32 *) (p -
  xi0 = *(uint32 *) (p -
  result += xi9
  result += xi8
  result += xi7
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0

                 =? p - y
goto loop if !=
```

```
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi9 = *(uint32 *) (p - 44)
xi8 = *(uint32 *) (p - 48)
xi7 = *(uint32 *) (p - 52)
xi6 = *(uint32 *) (p - 56)
xi5 = *(uint32 *) (p - 60)
xi4 = *(uint32 *) (p - 64)
xi3 = *(uint32 *) (p - 68)
xi2 = *(uint32 *) (p - 72)
```



```
xi1 = *(uint32 *) (p - 76)
xi0 = *(uint32 *) (p - 80)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0

                =? p - y
goto loop if !=
```

```
t += xi6
t += xi5
t += xi4
t += xi3
t += xi2
t += xi1
t += xi0
 *(uint32 *) (p - 44)
*(uint32 *) (p - 48)
*(uint32 *) (p - 52)
*(uint32 *) (p - 56)
*(uint32 *) (p - 60)
*(uint32 *) (p - 64)
*(uint32 *) (p - 68)
*(uint32 *) (p - 72)
```



```
  xi1 = *(uint32 *) (p - 76)
  xi0 = *(uint32 *) (p - 80)
  result += xi9
  result += xi8
  result += xi7
  result += xi6
  result += xi5
  result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0

                 =? p - y
goto loop if !=
```

Wikiped
even per
optimizi
performa

```
*) (p - 44)
*) (p - 48)
*) (p - 52)
*) (p - 56)
*) (p - 60)
*) (p - 64)
*) (p - 68)
*) (p - 72)
```



```
xi1 = *(uint32 *) (p - 76)
xi0 = *(uint32 *) (p - 80)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0

                =? p - y
goto loop if !=
```

Wikipedia: “By th
even performance
optimizing compil
performance of hu

```
18                                                                  19
        xi1 = *(uint32 *) (p - 76)
        xi0 = *(uint32 *) (p - 80)
        result += xi9
        result += xi8
        result += xi7
        result += xi6
        result += xi5
44)     result += xi4
48)       result += xi3
52)       result += xi2
56)       result += xi1
60)       result += xi0
64)
68)                  =? p - y
72)   goto loop if !=
```

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

```
xi1 = *(uint32 *) (p - 76)
xi0 = *(uint32 *) (p - 80)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0

            =? p - y
goto loop if !=
```


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

```
xi1 = *(uint32 *) (p - 76)
xi0 = *(uint32 *) (p - 80)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
  result += xi3
  result += xi2
  result += xi1
  result += xi0

                =? p - y
goto loop if !=
```


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

Reality: The fastest software today relies on human experts understanding the CPU.

Cannot trust compiler to optimize instruction selection.

Cannot trust compiler to optimize instruction scheduling.

Cannot trust compiler to optimize register allocation.

```
 *(uint32 *) (p - 76)
 *(uint32 *) (p - 80)
t += xi9
t += xi8
t += xi7
t += xi6
t += xi5
t += xi4
t += xi3
t += xi2
t += xi1
t += xi0

       =? p - y
op if !=
```


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

Reality: The fastest software today relies on human experts understanding the CPU.

Cannot trust compiler to optimize instruction selection.

Cannot trust compiler to optimize instruction scheduling.

Cannot trust compiler to optimize register allocation.

The big

CPUs ar
farther a
from nai

```
*) (p - 76)
*) (p - 80)
```

```
p - y
```

19

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

Reality: The fastest software today relies on human experts understanding the CPU.

Cannot trust compiler to optimize instruction selection.

Cannot trust compiler to optimize instruction scheduling.

Cannot trust compiler to optimize register allocation.

20

The big picture

CPUs are evolving farther and farther from naive models

76)

80)

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

Reality: The fastest software today relies on human experts understanding the CPU.

Cannot trust compiler to optimize instruction selection.

Cannot trust compiler to optimize instruction scheduling.

Cannot trust compiler to optimize register allocation.

The big picture

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

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

Reality: The fastest software today relies on human experts understanding the CPU.

Cannot trust compiler to optimize instruction selection.

Cannot trust compiler to optimize instruction scheduling.

Cannot trust compiler to optimize register allocation.

The big picture

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

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

Reality: The fastest software today relies on human experts understanding the CPU.

Cannot trust compiler to optimize instruction selection.

Cannot trust compiler to optimize instruction scheduling.

Cannot trust compiler to optimize register allocation.

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.

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

The fastest software
lies on human experts
nding the CPU.

trust compiler to
e instruction selection.

trust compiler to
e instruction scheduling.

trust compiler to
e register allocation.

The big picture

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

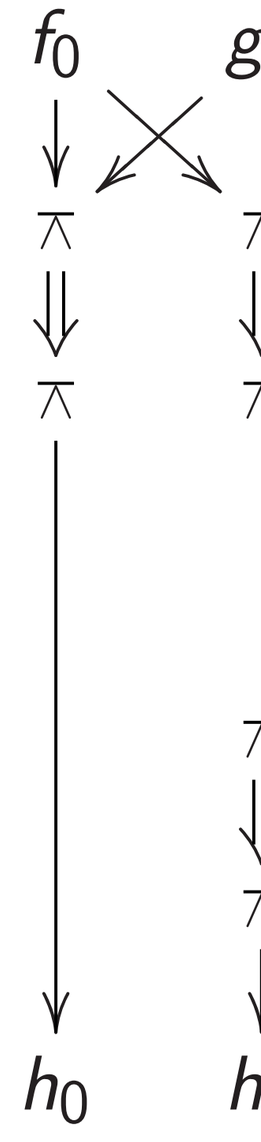
Minor optimization challenges:

- 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 $\curlywedge$
product
of intege

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

st software
man experts
CPU.

piler to
on selection.

piler to
on scheduling.

piler to
llocation.

The big picture

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

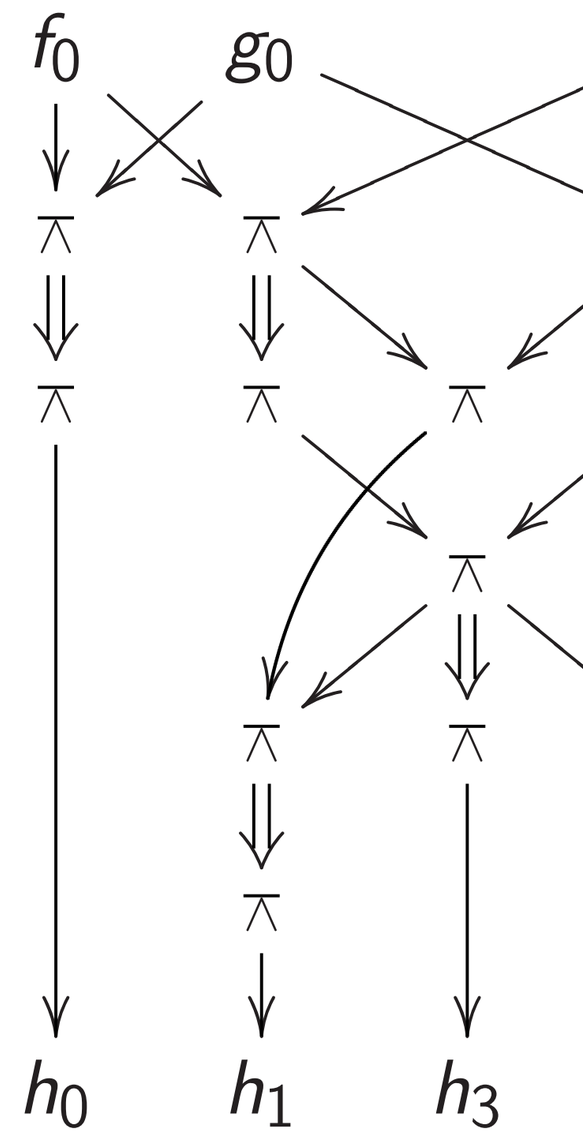
Minor optimization challenges:

- 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 $f_0 + 2f$

90s for
code,
ed the
rts."

e
rts

n.

ing.

The big picture

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

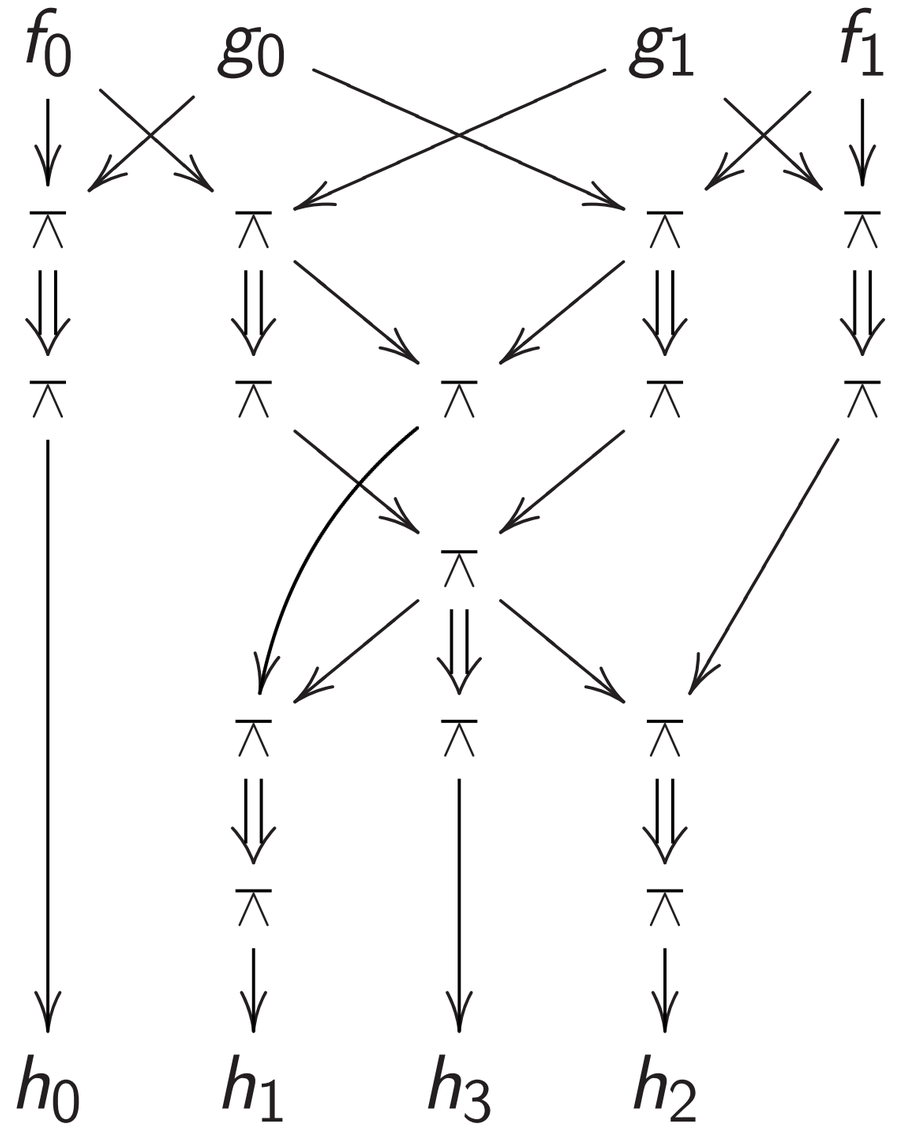
Minor optimization challenges:

- 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 $\curlywedge : a, b \mapsto 1 - ab$ 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.

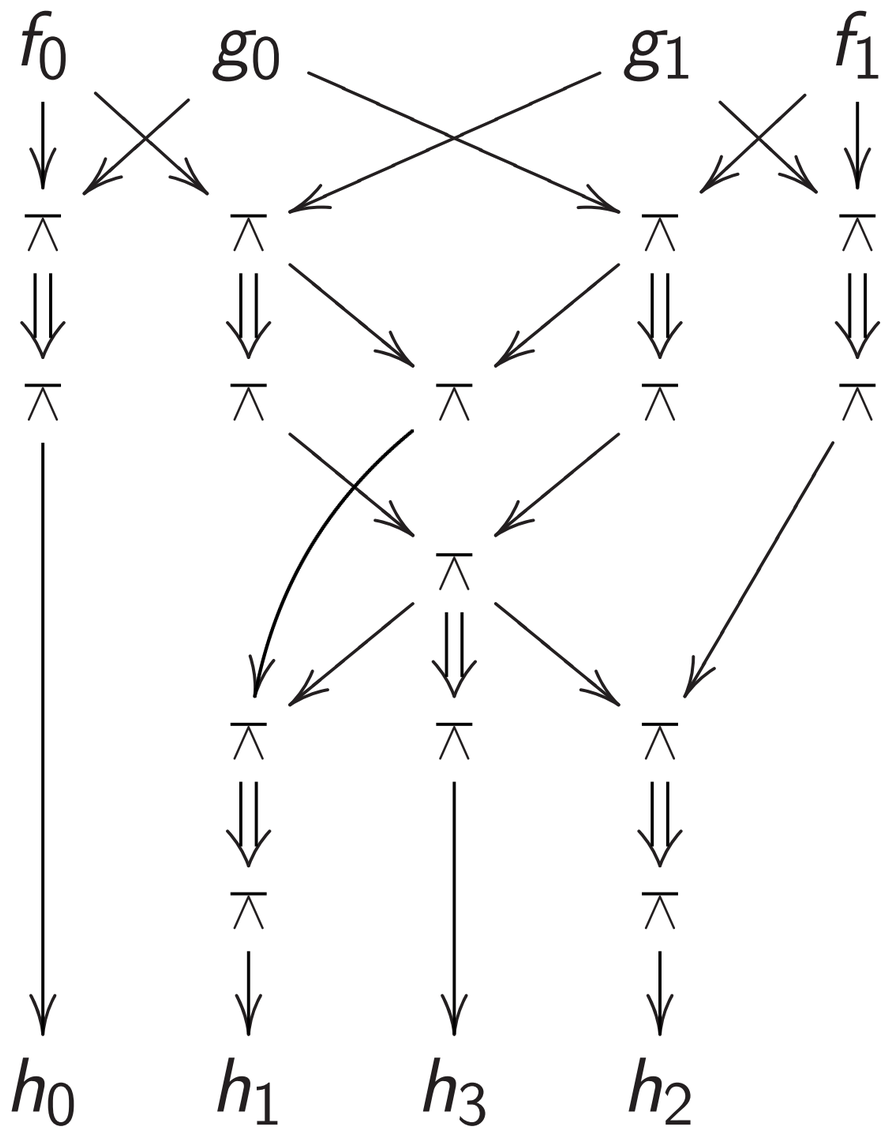
Minor optimization challenges:

- 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

nd farther away

ve models of CPUs.

ptimization challenges:

ning.

scalar processing.

ptimization challenges:

rization.

threads; many cores.

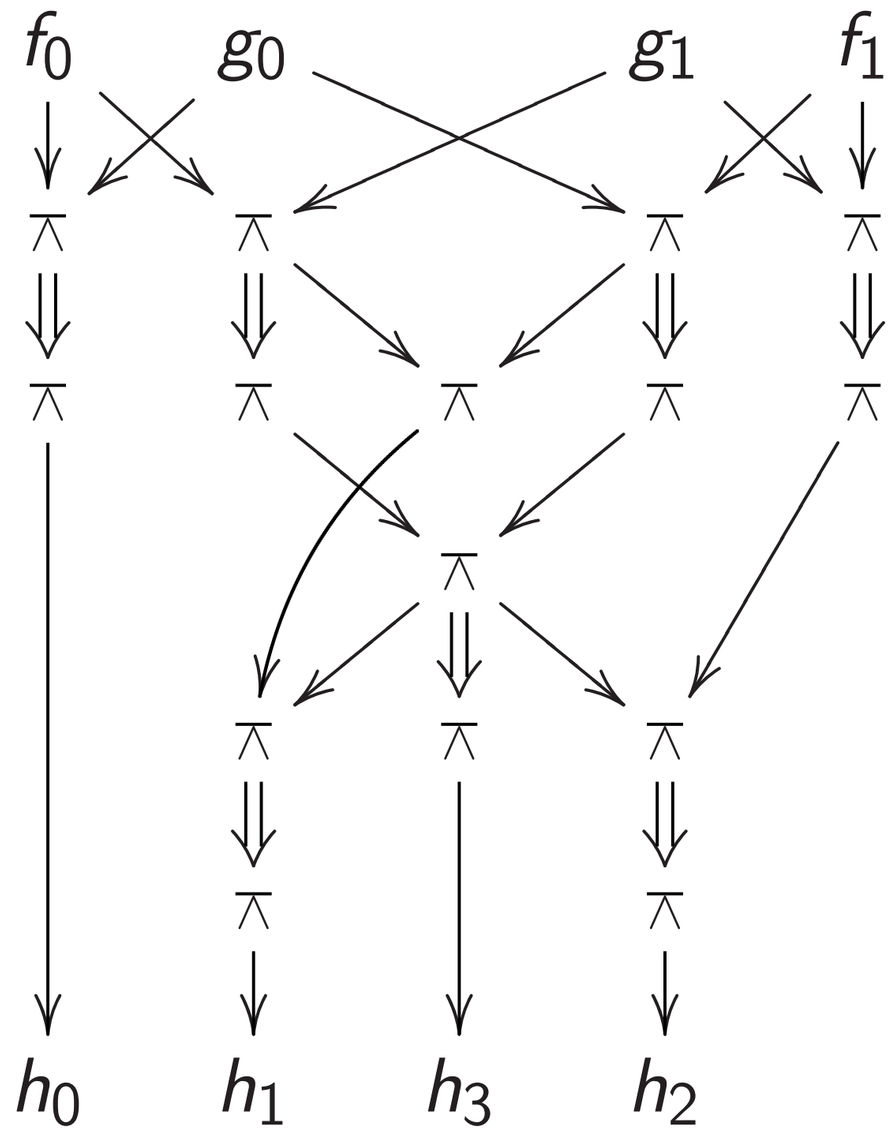
nemory hierarchy;

ng; the mesh.

-scale parallelism.

-scale networking.

CPU design in a nutshell



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

Electrici

percolat

If $f_0, f_1,$ &

then $h_0,$

a few m

r away

s of CPUs.

n challenges:

cessing.

n challenges:

many cores.

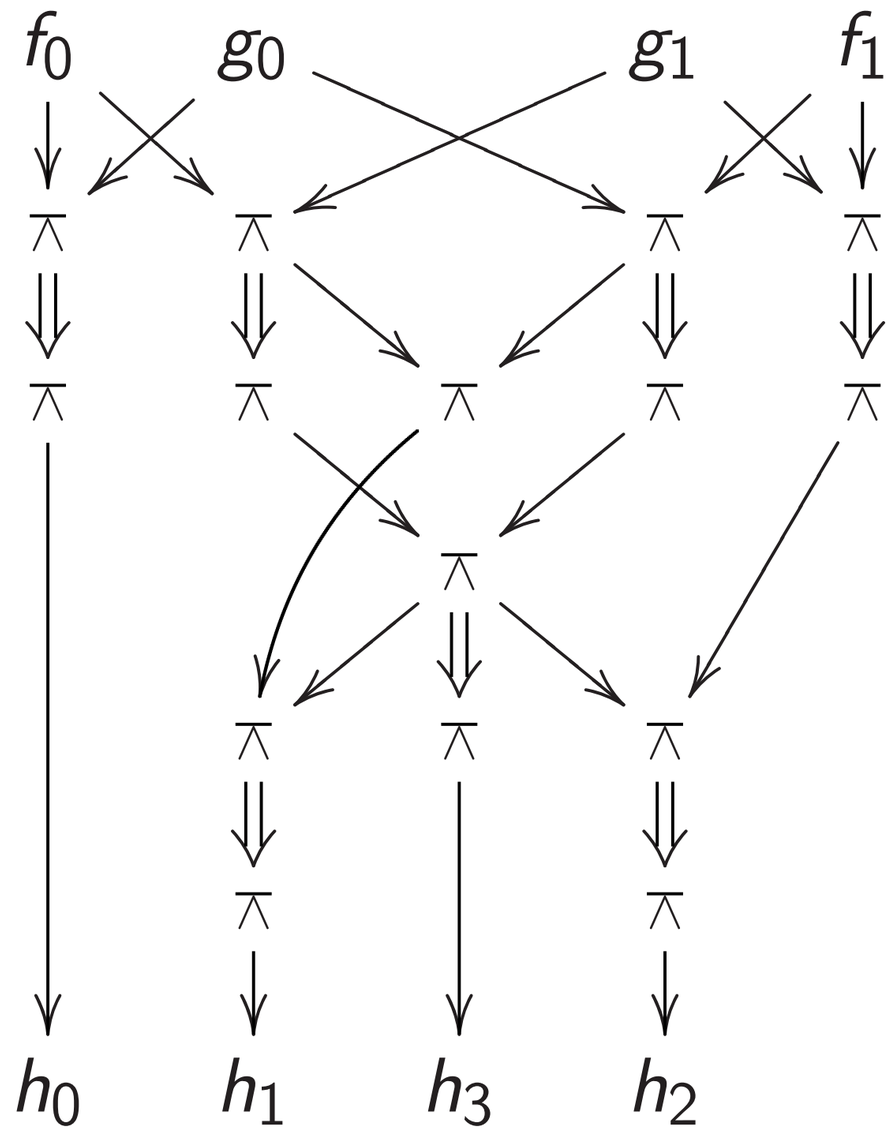
erarchy;

sh.

allelism.

working.

CPU design in a nutshell



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

Electricity takes ti

percolate through

If f_0, f_1, g_0, g_1 are

then h_0, h_1, h_2, h_3

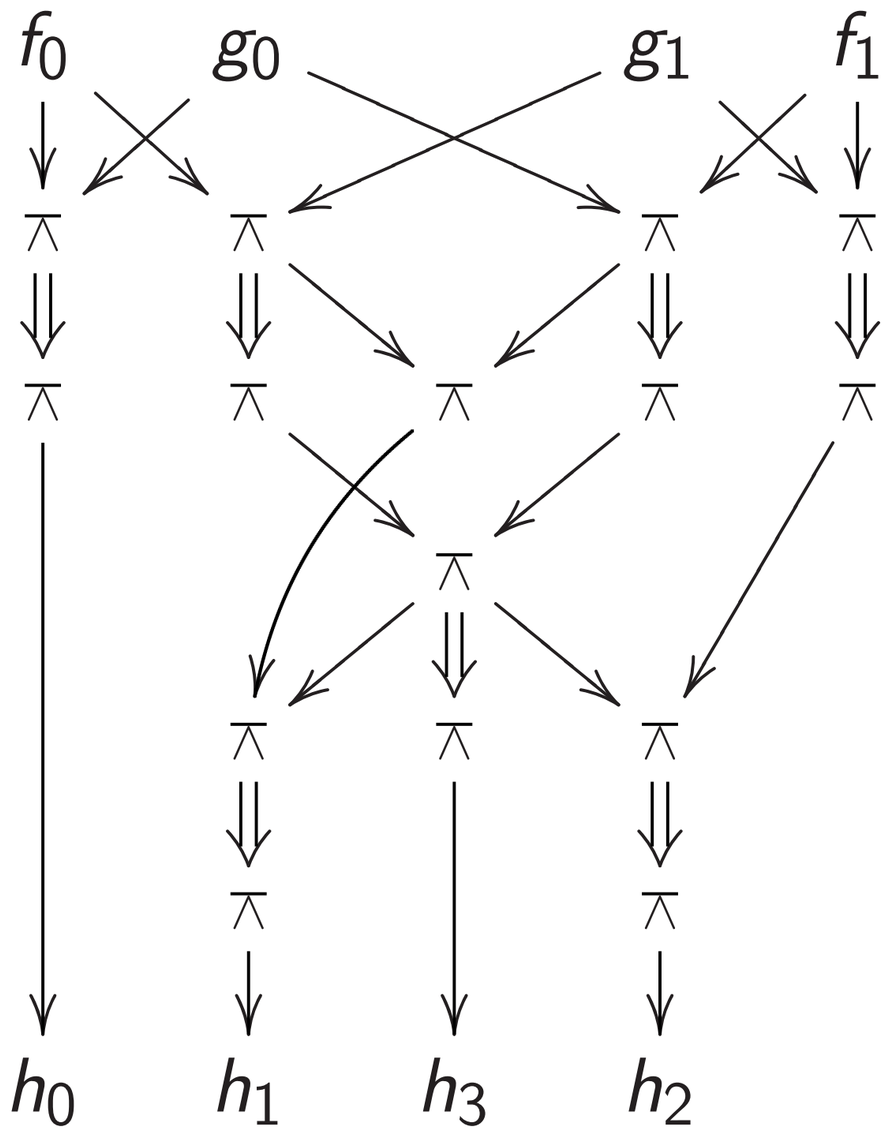
a few moments lat

es:

es:

s.

CPU design in a nutshell

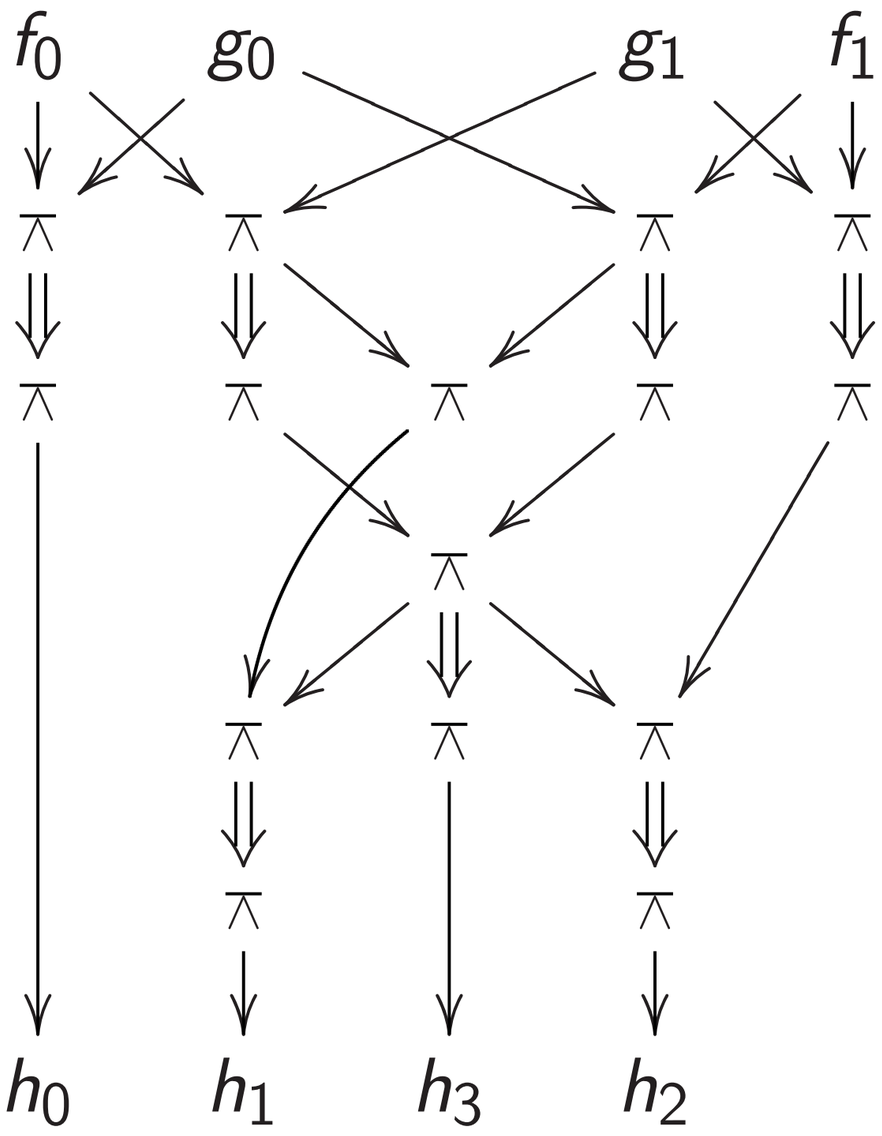


Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

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.

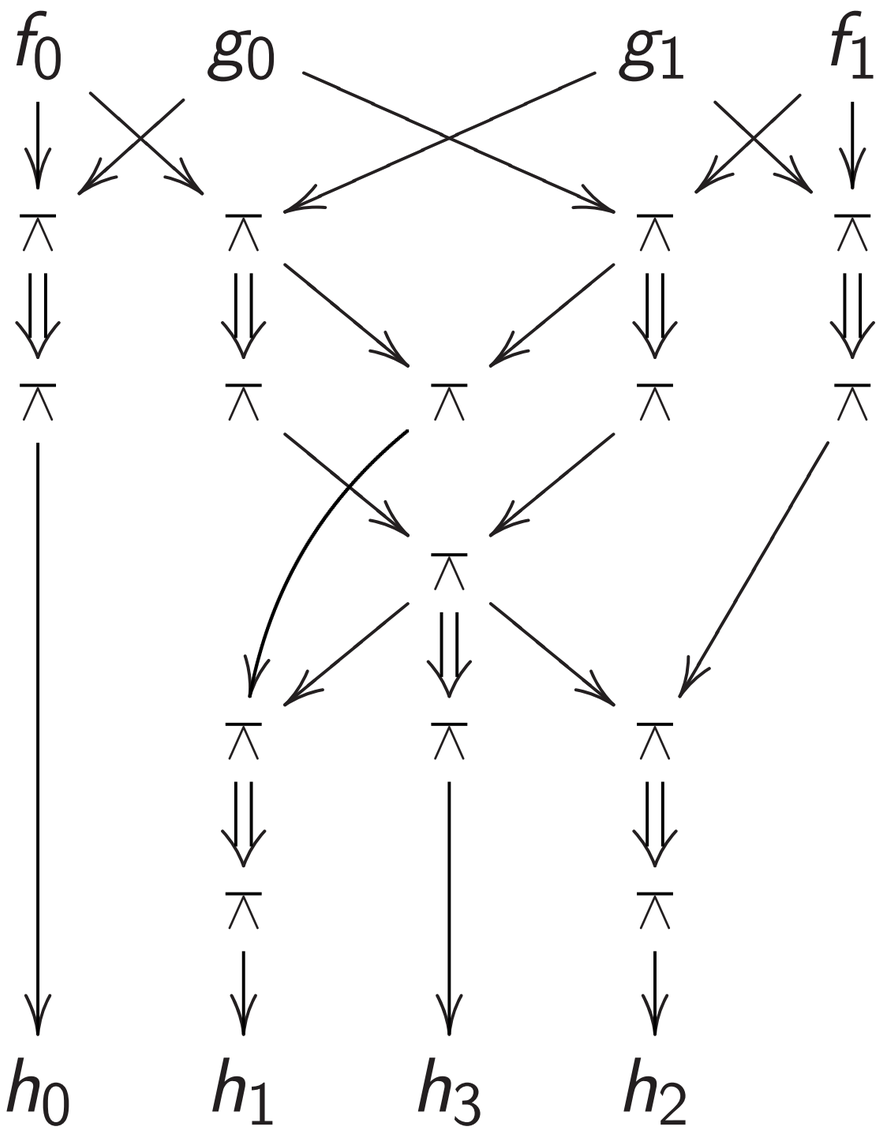
CPU design in a nutshell



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.

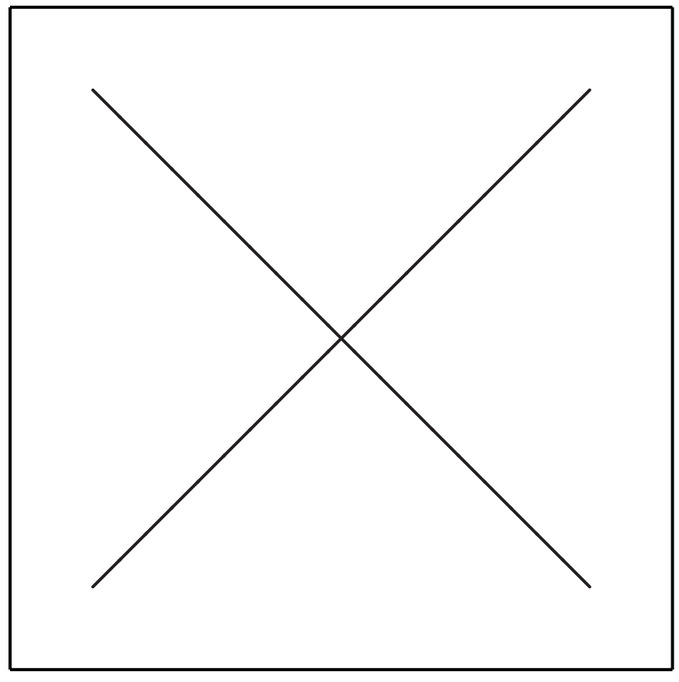
CPU design in a nutshell



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

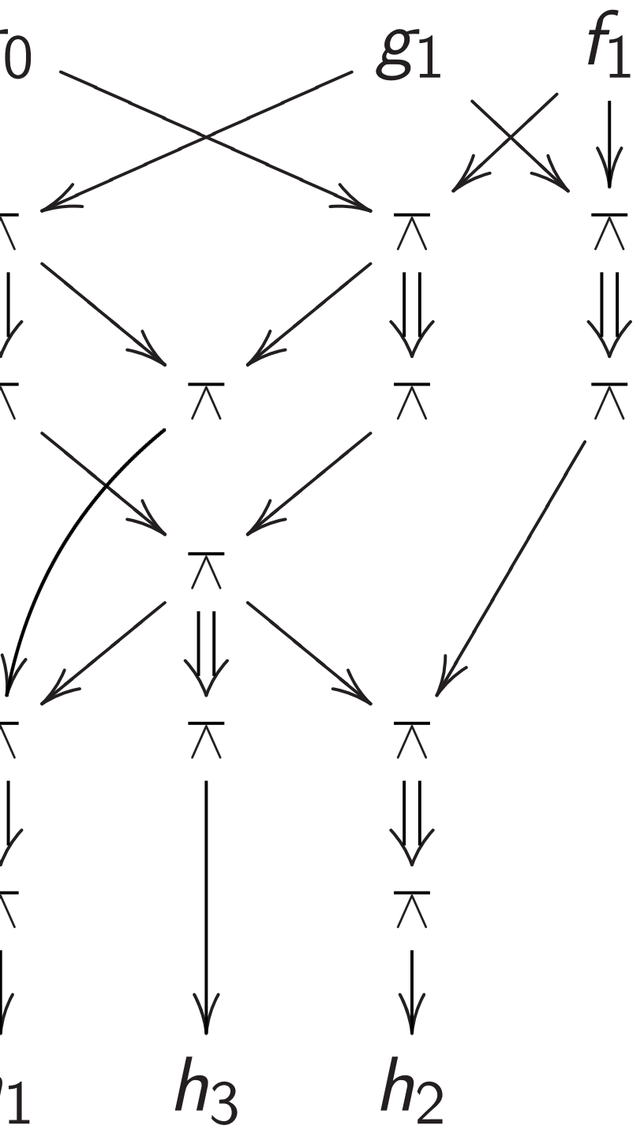
Electricity takes time to
percolate through wires and gates.
If f_0, f_1, g_0, g_1 are stable
then h_0, h_1, h_2, h_3 are stable
a few moments later.

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



(Details omitted.)

sign in a nutshell



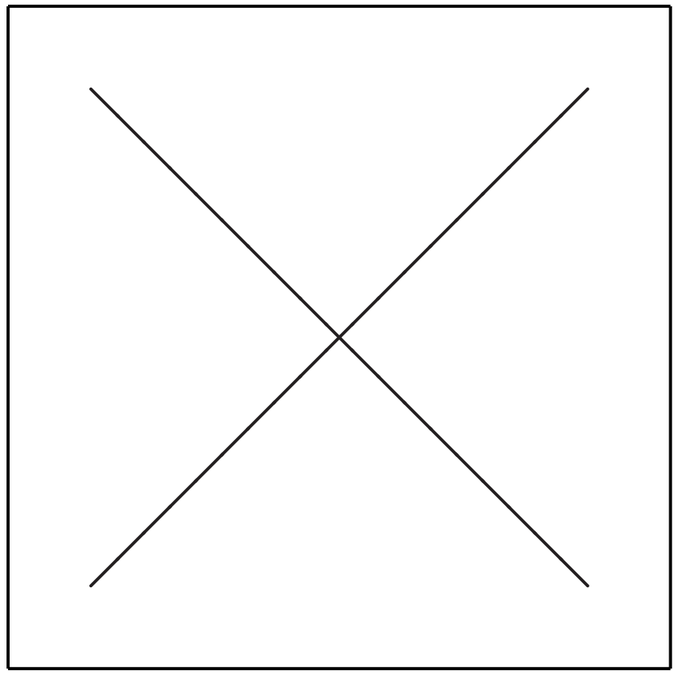
: $a, b \mapsto 1 - ab$ computing

$h_0 + 2h_1 + 4h_2 + 8h_3$

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:

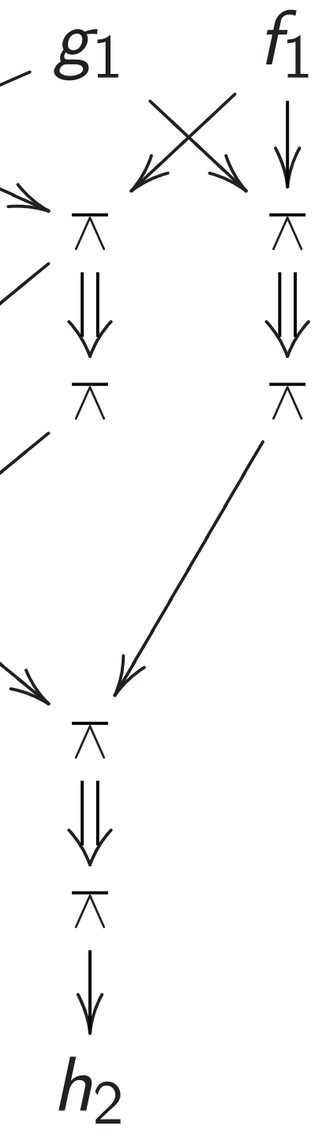


(Details omitted.)

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

reg
re

utshell



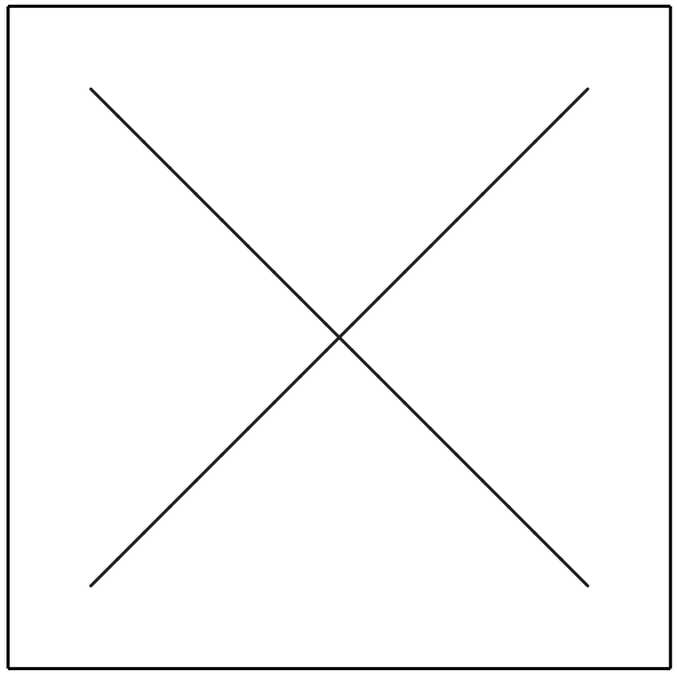
$- ab$ computing

$+ 4h_2 + 8h_3$

$_1, g_0 + 2g_1$.

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

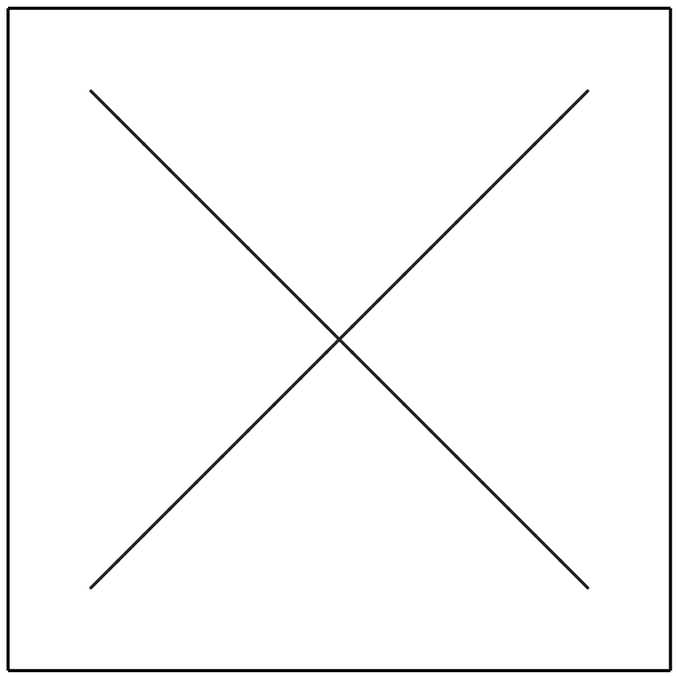
register

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:

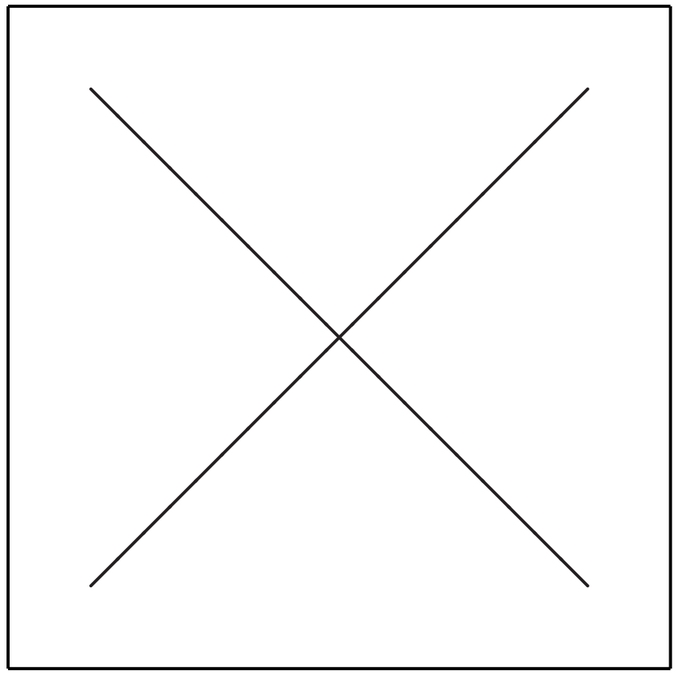


(Details omitted.)

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

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:

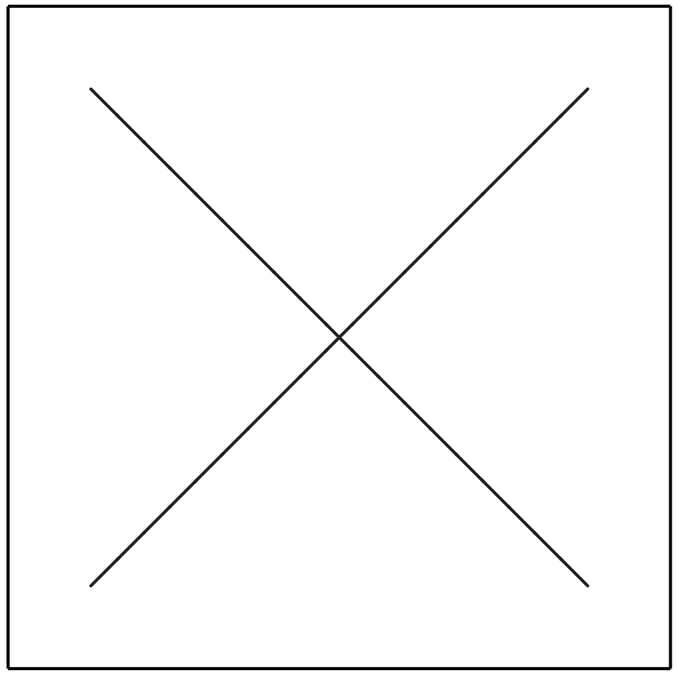


(Details omitted.)

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:



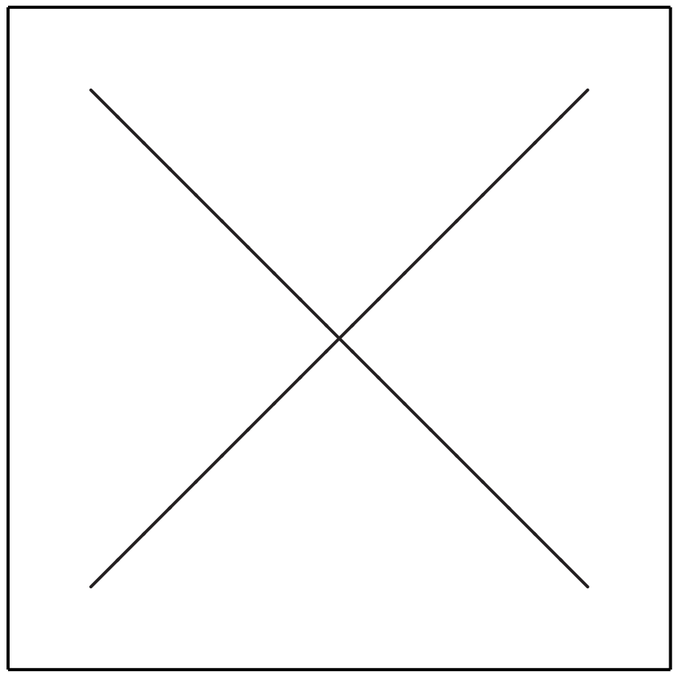
(Details omitted.)

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

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:



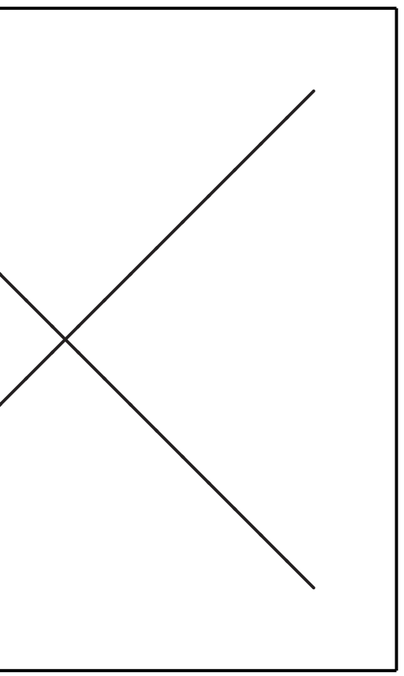
(Details omitted.)

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:

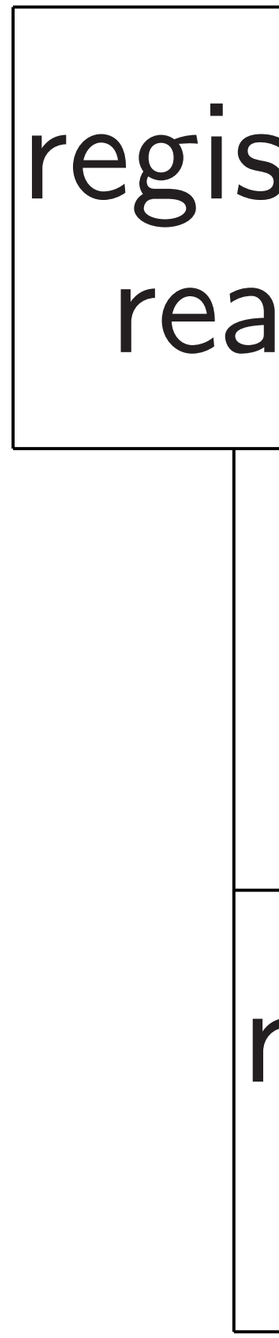


omitted.)

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

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

$r_0, \ldots, r$
where r'_ℓ



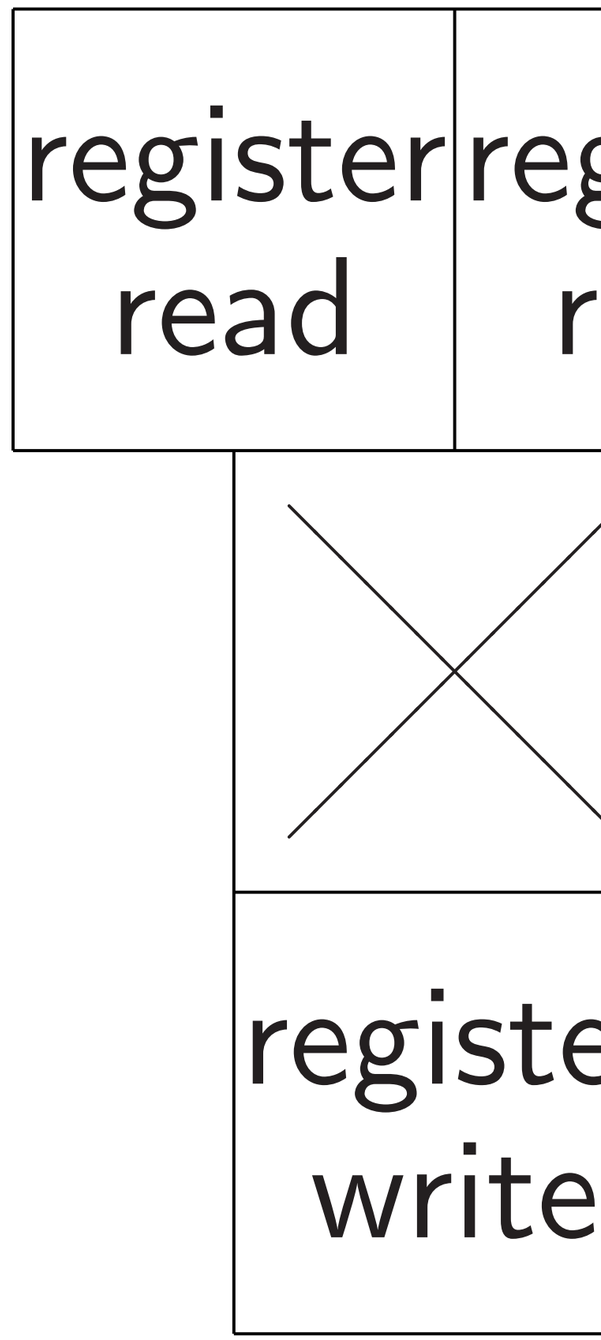
me to
wires and gates.
stable
are stable
ter.

more gates
32-bit integers:

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

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

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



gates.

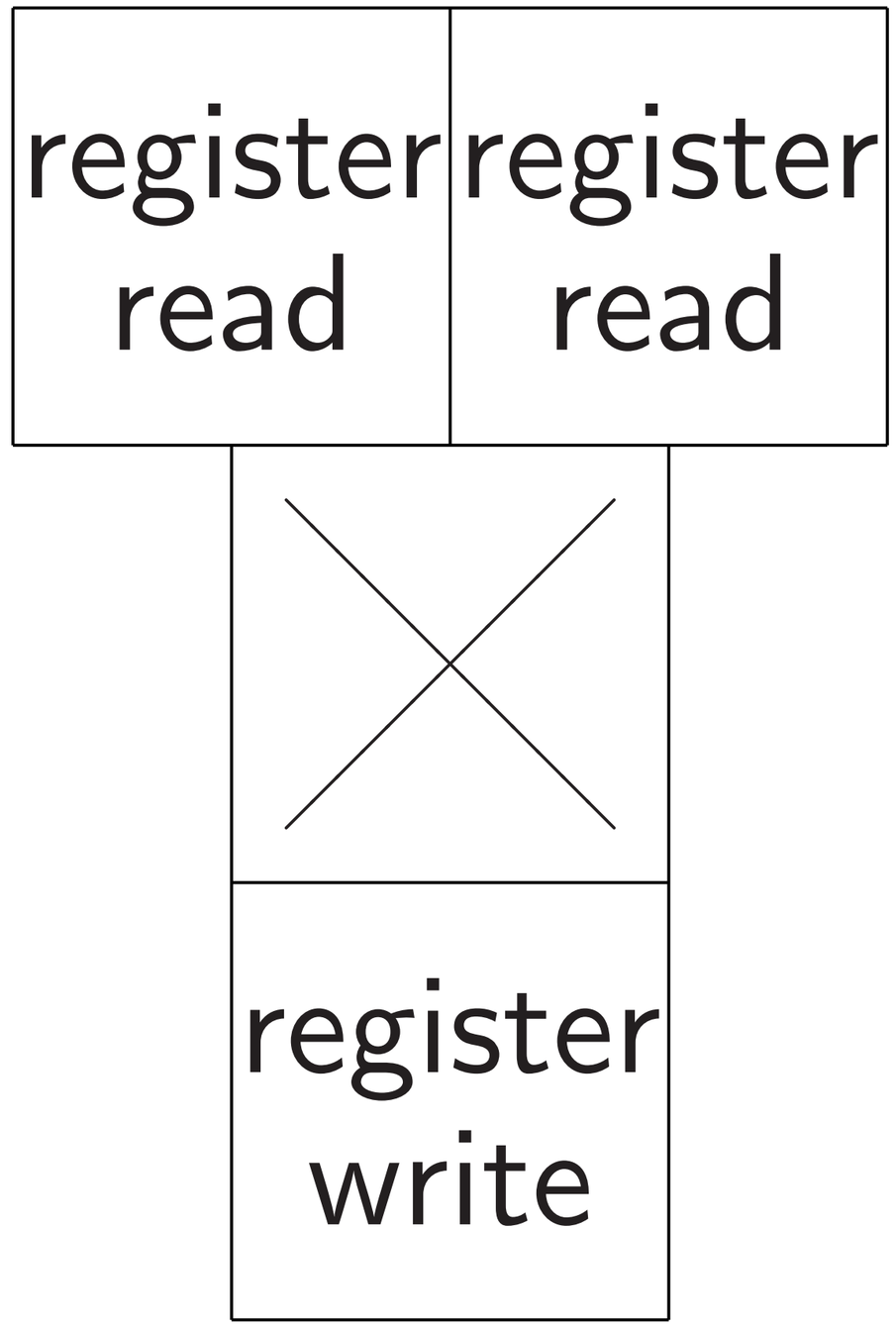
s

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.

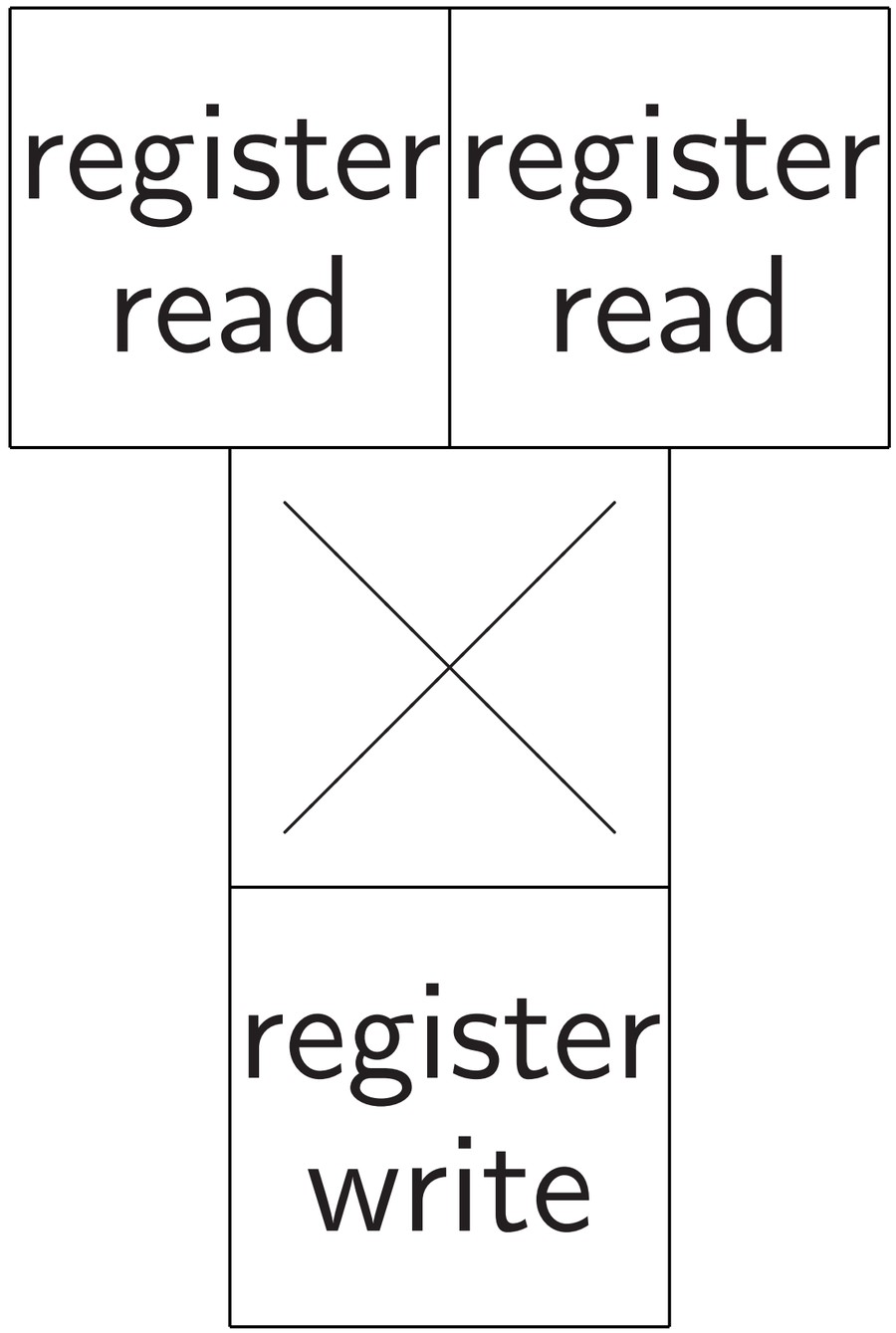
$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r$
where $r'_\ell = r_\ell$ except $r'_i = r_j$



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.

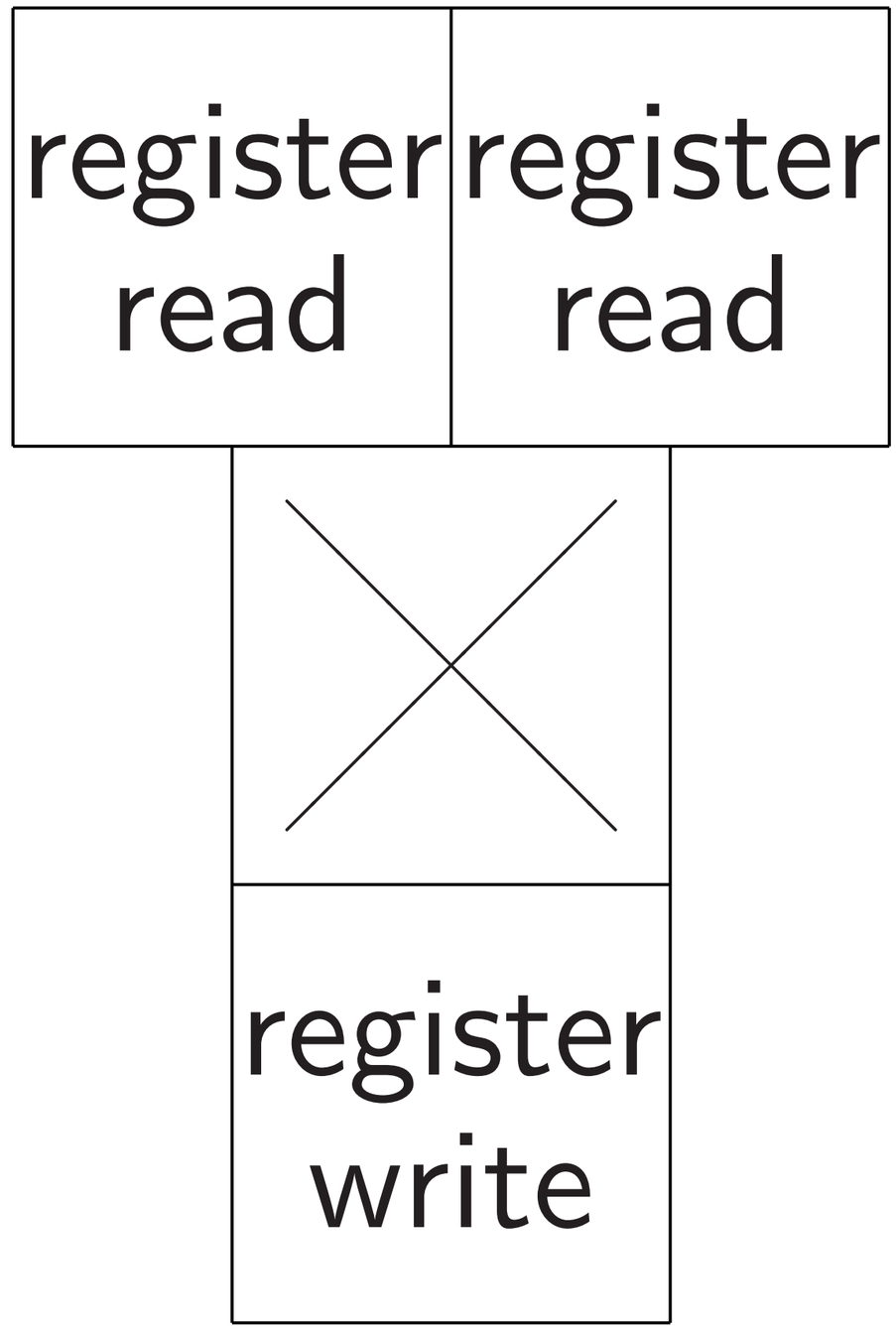
$r_0, \ldots, r_{15}, i, j, k \mapsto r'_0, \ldots, r'_{15}$
where $r'_\ell = r_\ell$ except $r'_i = r_j r_k$:



rcuit to compute
nteger r_i
bit integer i
bit integers $r_0, r_1, \ldots, r_{15}$:

rcuit for "register write":
$_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
$= r_j$ except $r'_i = s$.
rcuit for addition. Etc.

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



Add mo

More ari

replace (
("×", i,
("+", i,

mpute

i

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

r

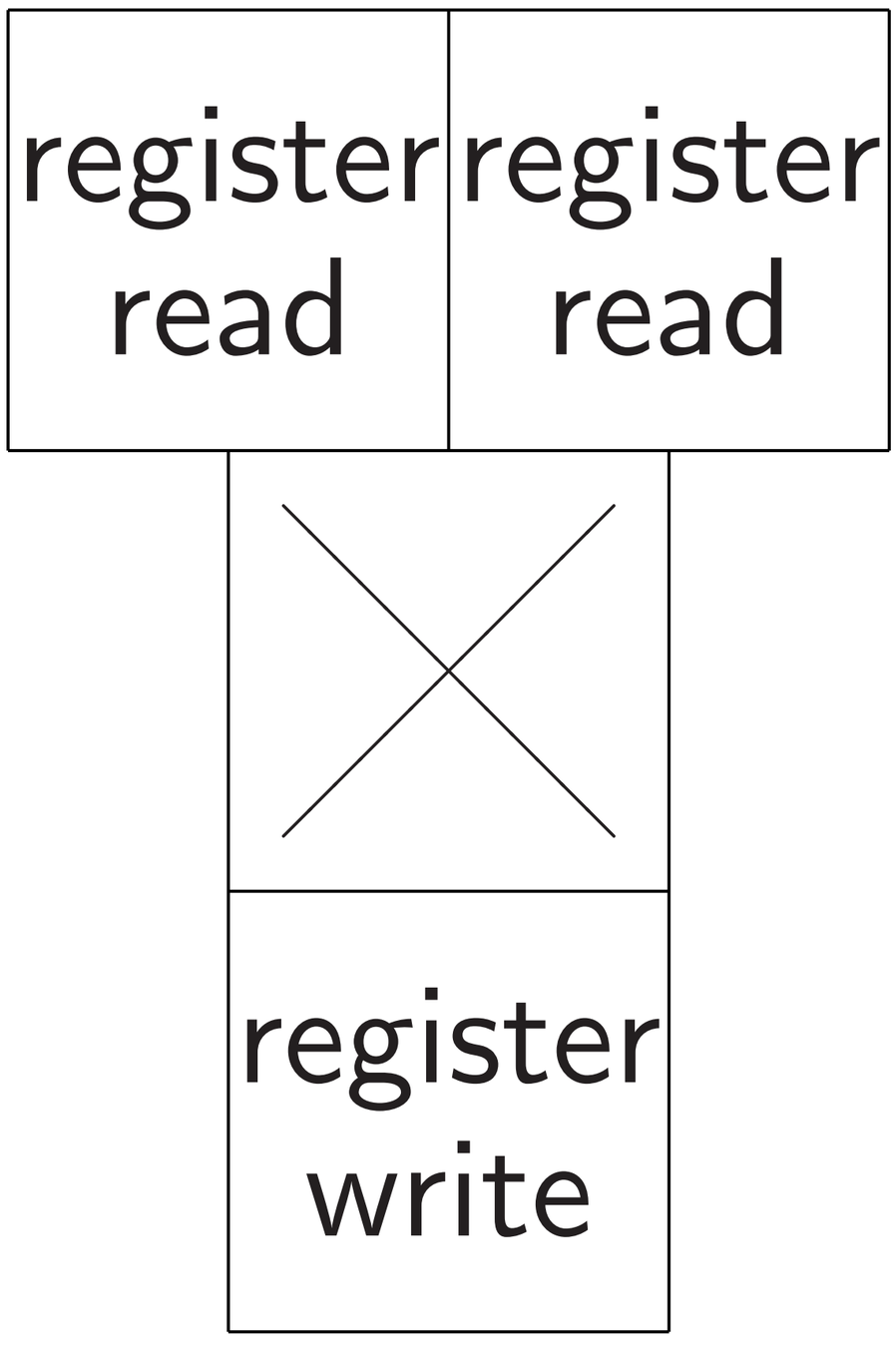
egister write":

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

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

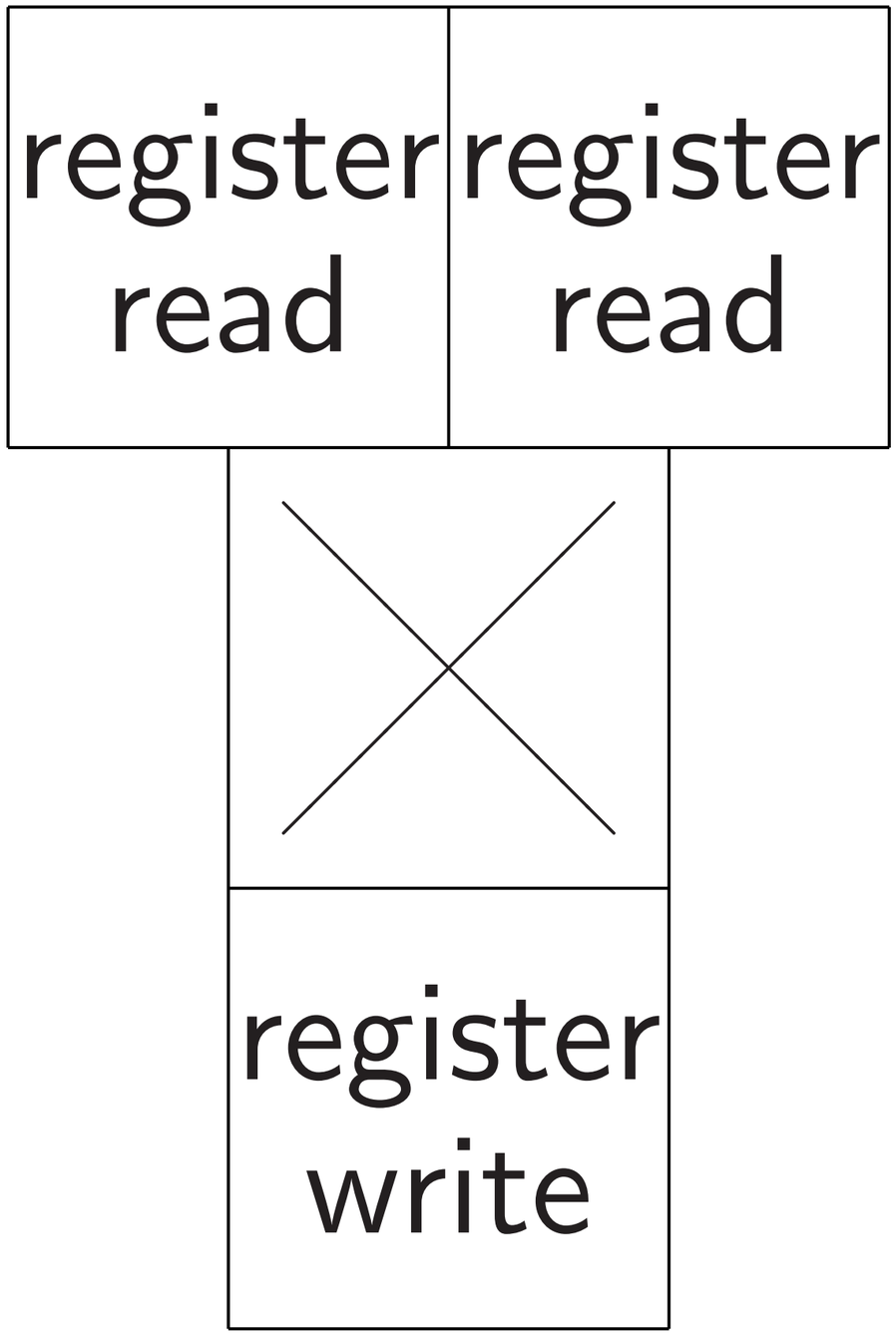
More arithmetic:
replace (i, j, k) wit
("×", i, j, k) and
("+", i, j, k) and n

, r_{15}:

ite" :

tc.

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

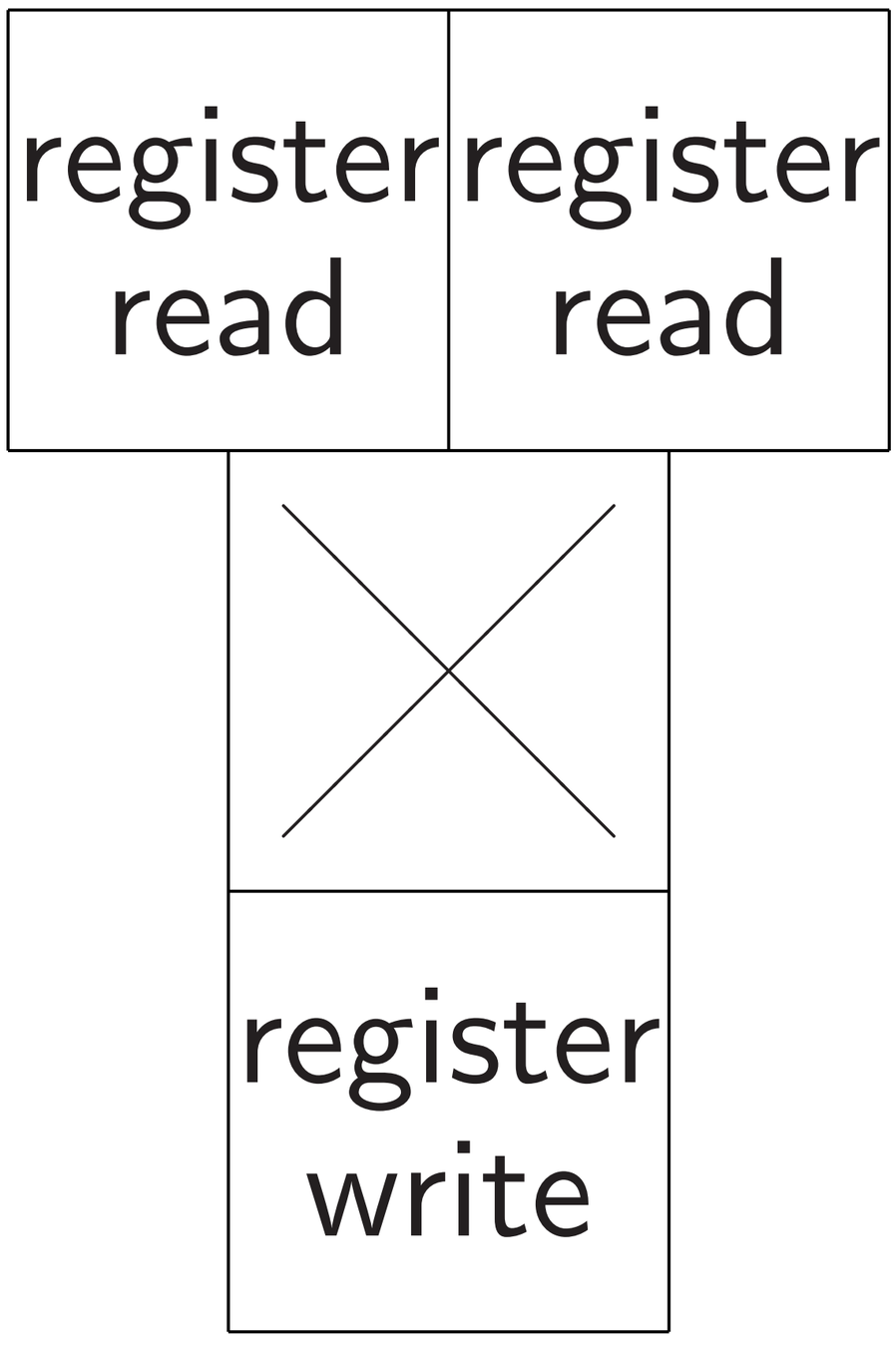


Add more flexibility.

More arithmetic:

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

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

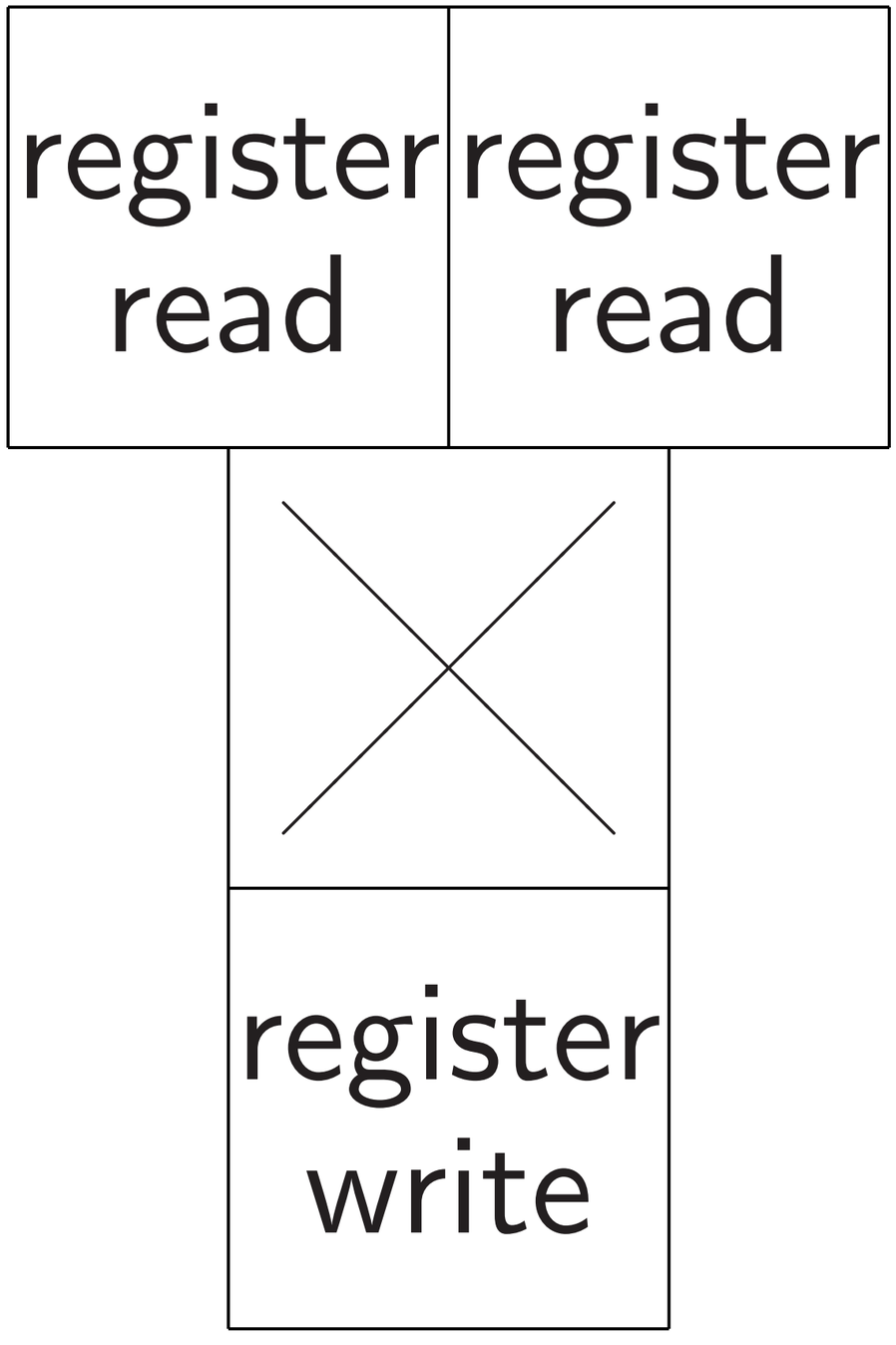


Add more flexibility.

More arithmetic:

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

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

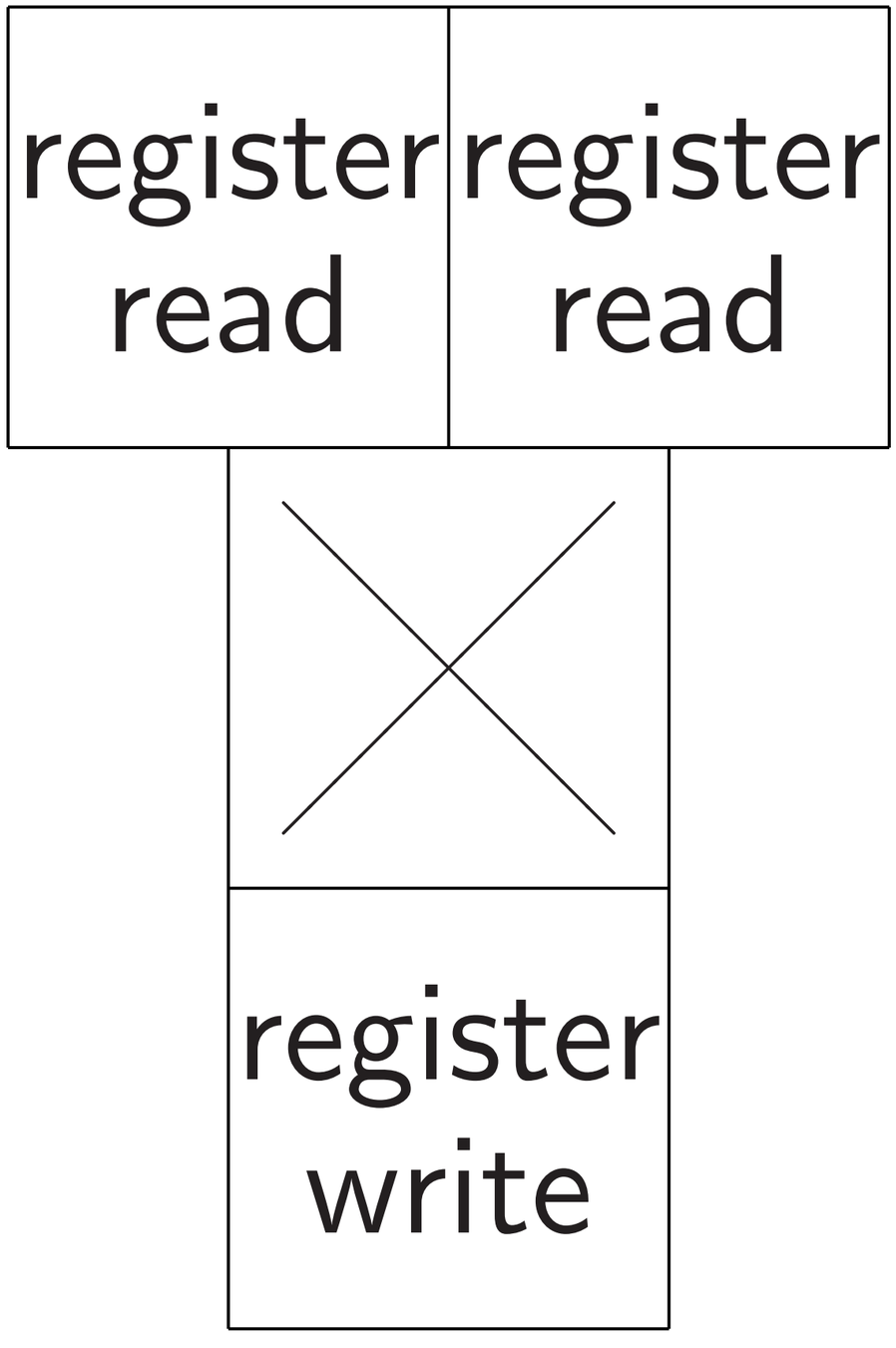


Add more flexibility.

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

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

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



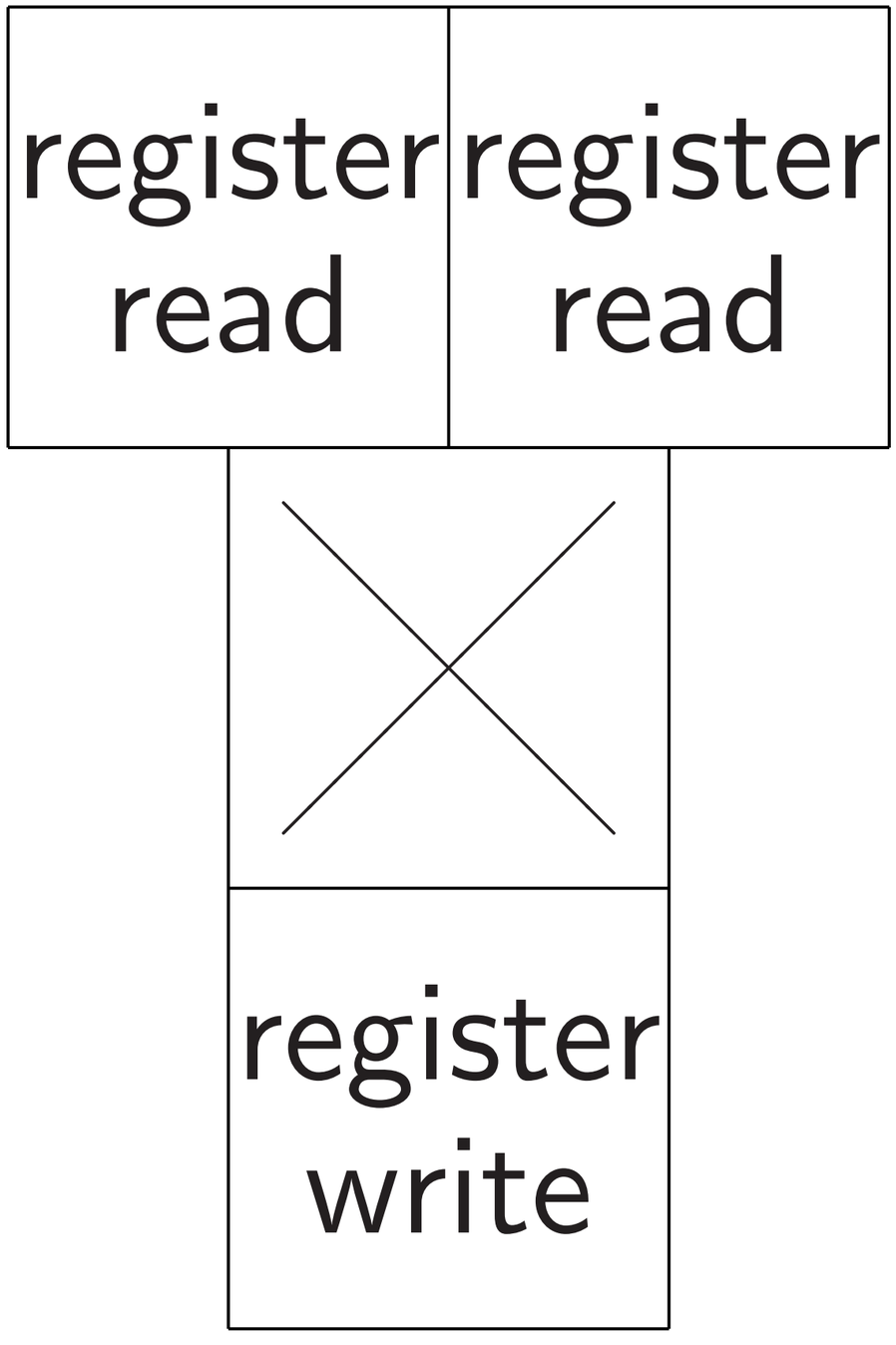
Add more flexibility.

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

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

“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.

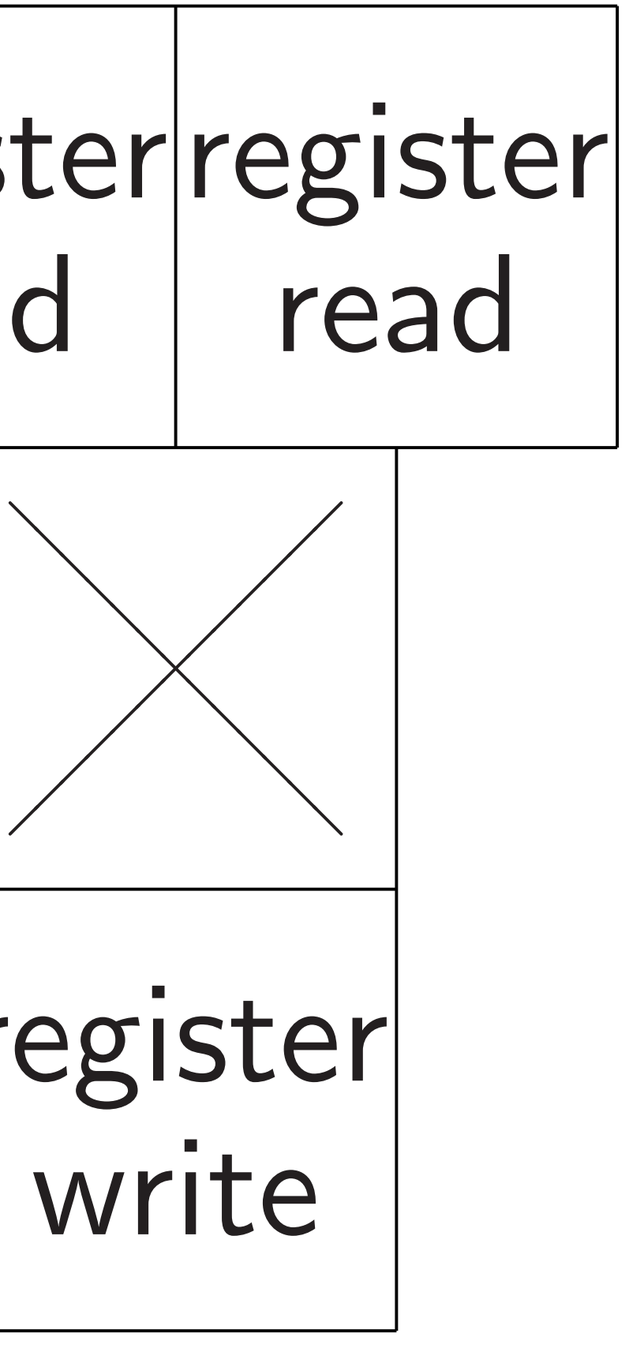
More (but slower) storage:
“load” from and “store” to
larger “RAM” arrays.

“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'.

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

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



Add more flexibility.

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

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

“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'.

Build “fl
storing (

Hook (p
flip-flops

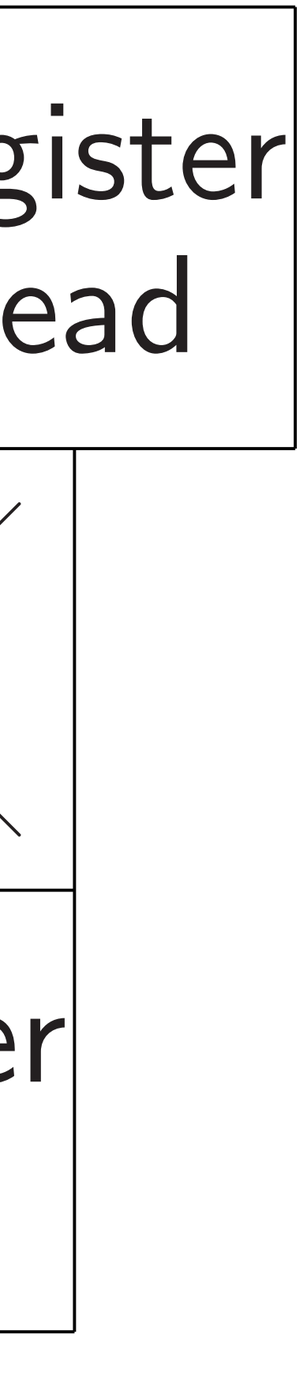
Hook ou
into the

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.

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

“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'.

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

r'_{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.

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

“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'.

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.

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

“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'.

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.

ut slower) storage:
rom and "store" to
RAM" arrays.

tion fetch":
i_p, j_p, k_p, p'.

tion decode":
ression of compressed
or o_p, i_p, j_p, k_p, p'.

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., log

ty.

th

more options.

storage:
"store" to
ays.

":
p'.

le":
compressed
$_p$, k_p, p'.

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

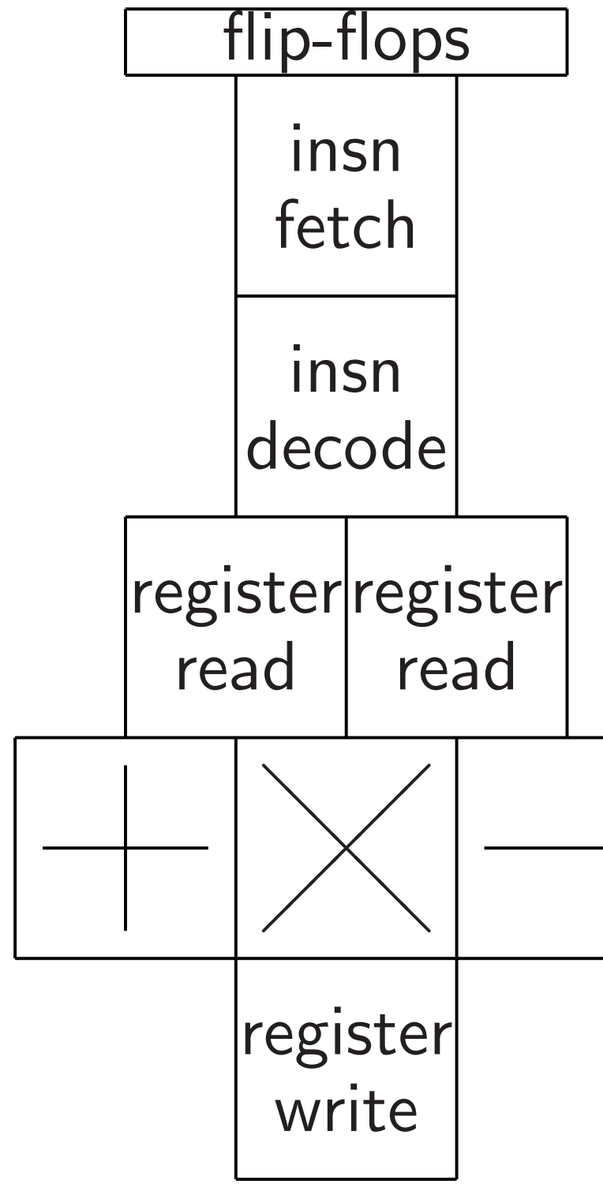
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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., logic instruct

ns.

d

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

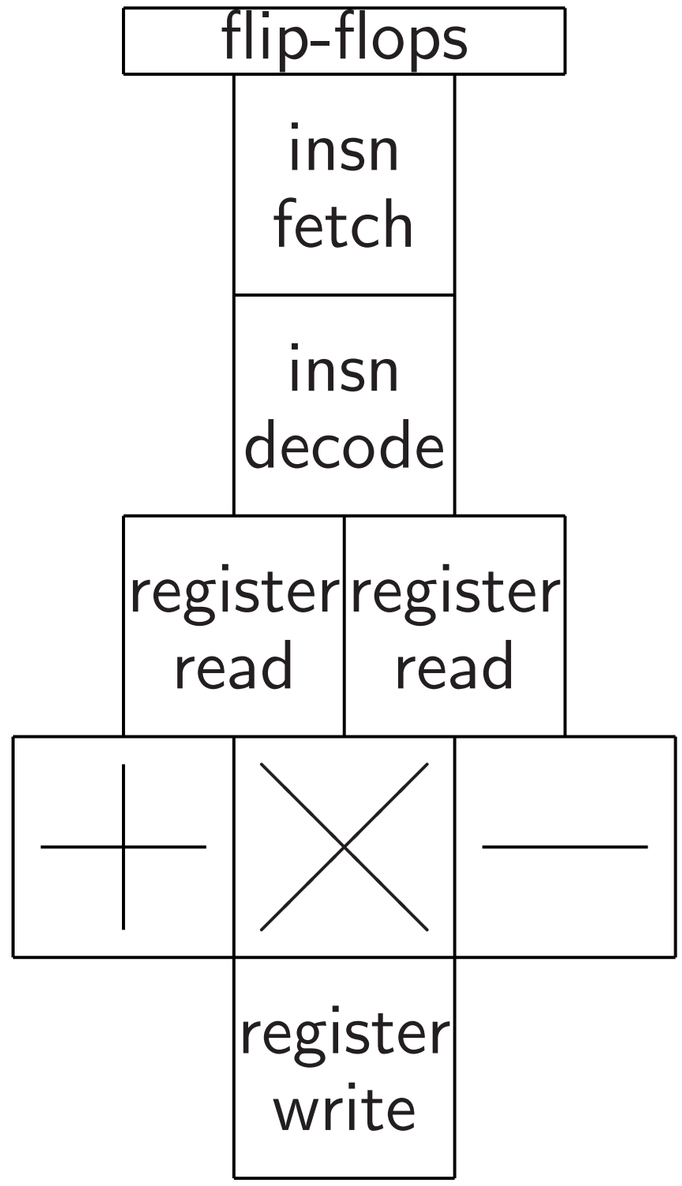
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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., logic instructions.

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

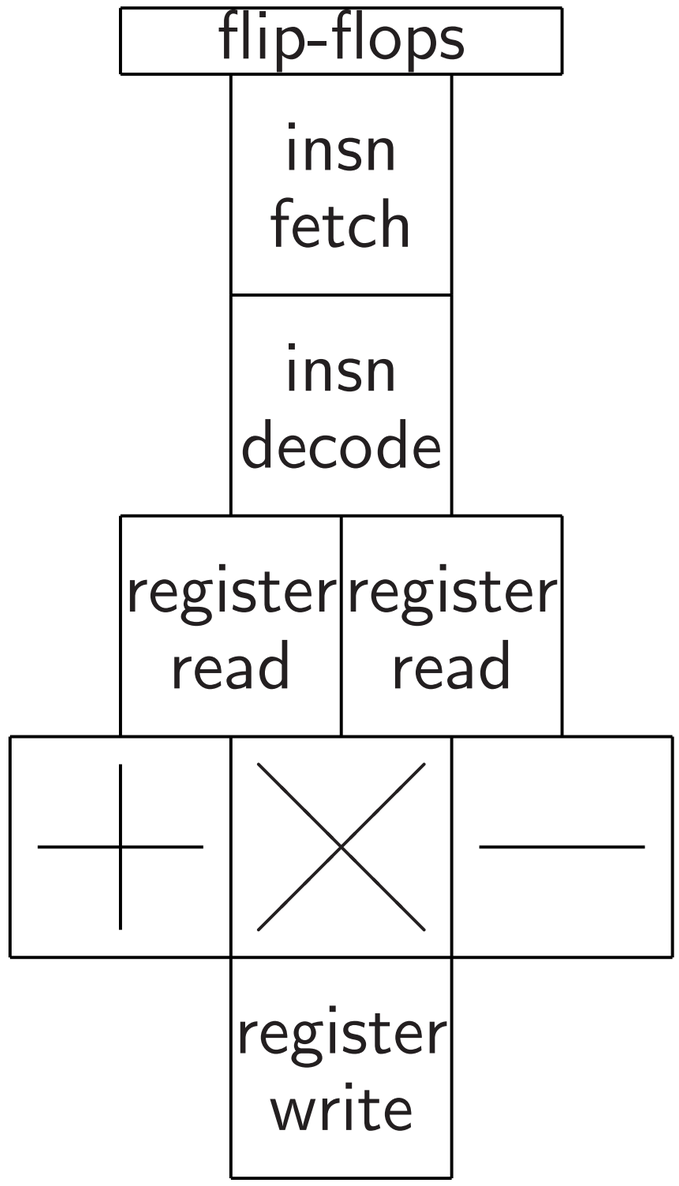
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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., logic instructions.

lip-flops"

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

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

s into circuit inputs.

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

same flip-flops.

"clock tick",

s are overwritten

outputs.

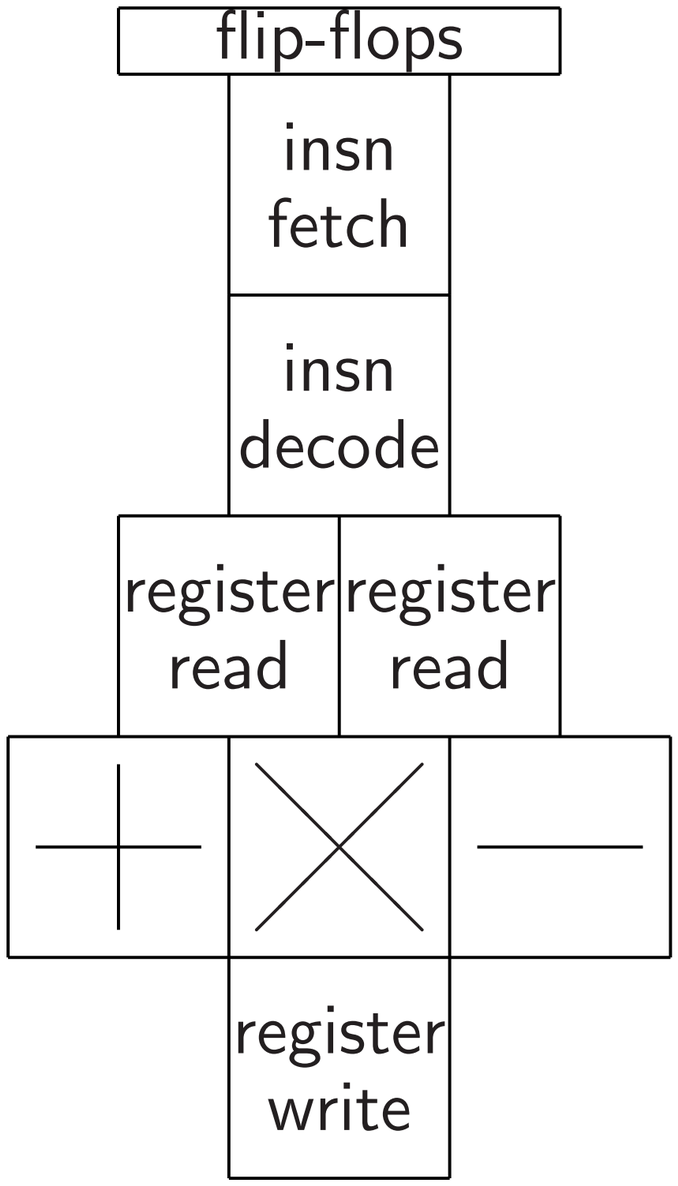
eeds to be slow enough

ricity to percolate

way through the circuit,

p-flops to flip-flops.

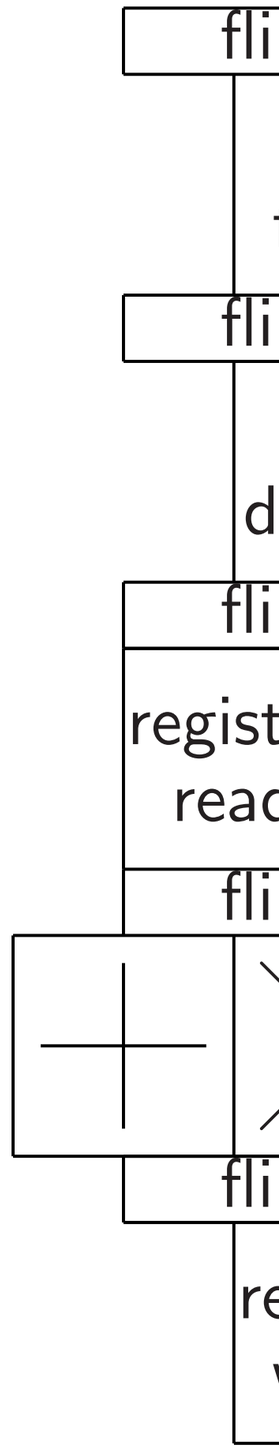
Now have semi-flexible CPU:



Further flexibility is useful:

e.g., logic instructions.

"Pipelin



r_{15}).

$_{15}$)

uit inputs.

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

-flops.

k",

written

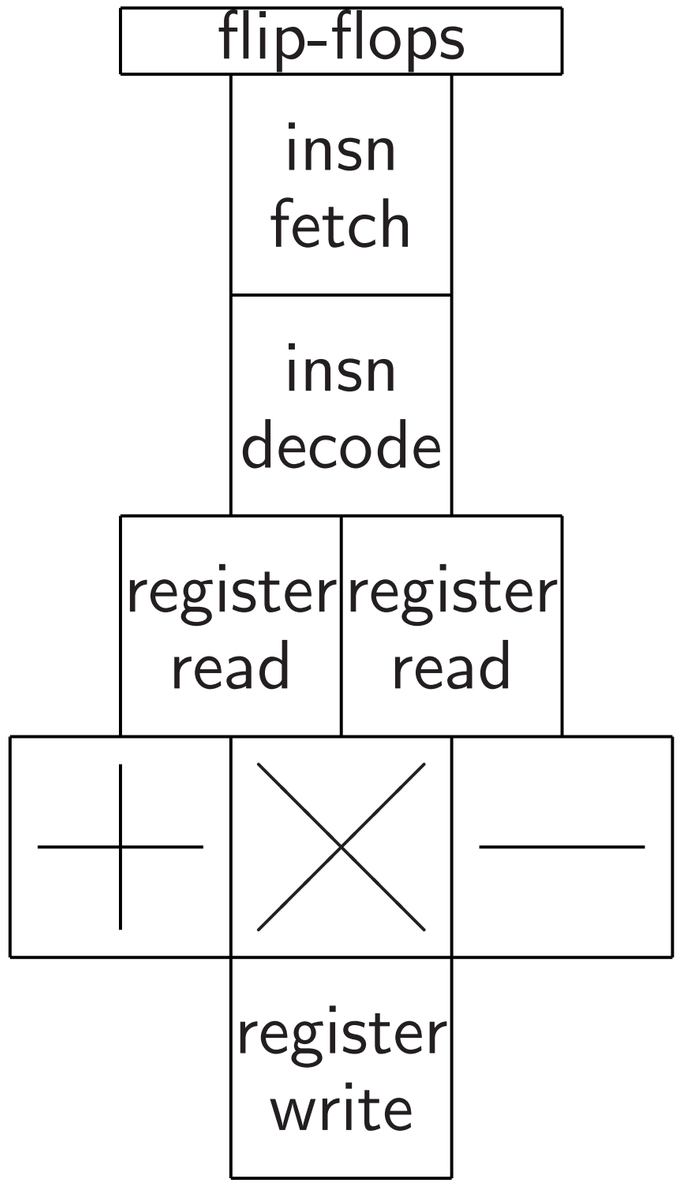
slow enough

ercolate

h the circuit,

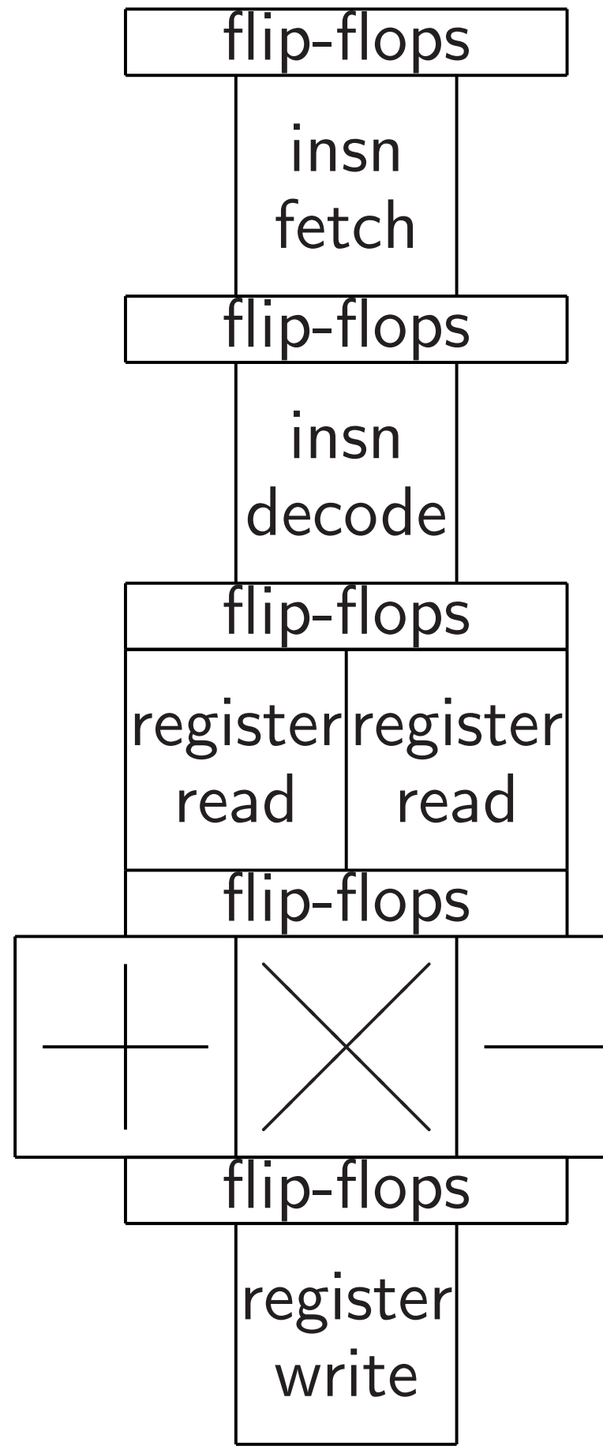
flip-flops.

Now have semi-flexible CPU:

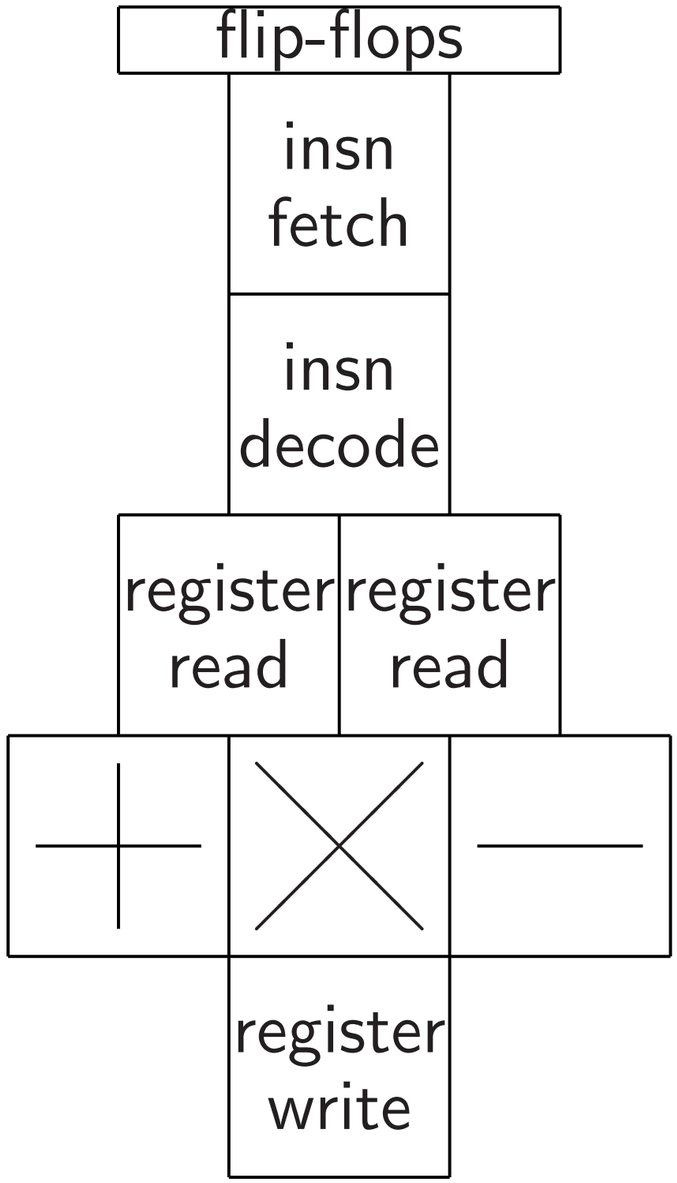


Further flexibility is useful:
e.g., logic instructions.

"Pipelining" allow

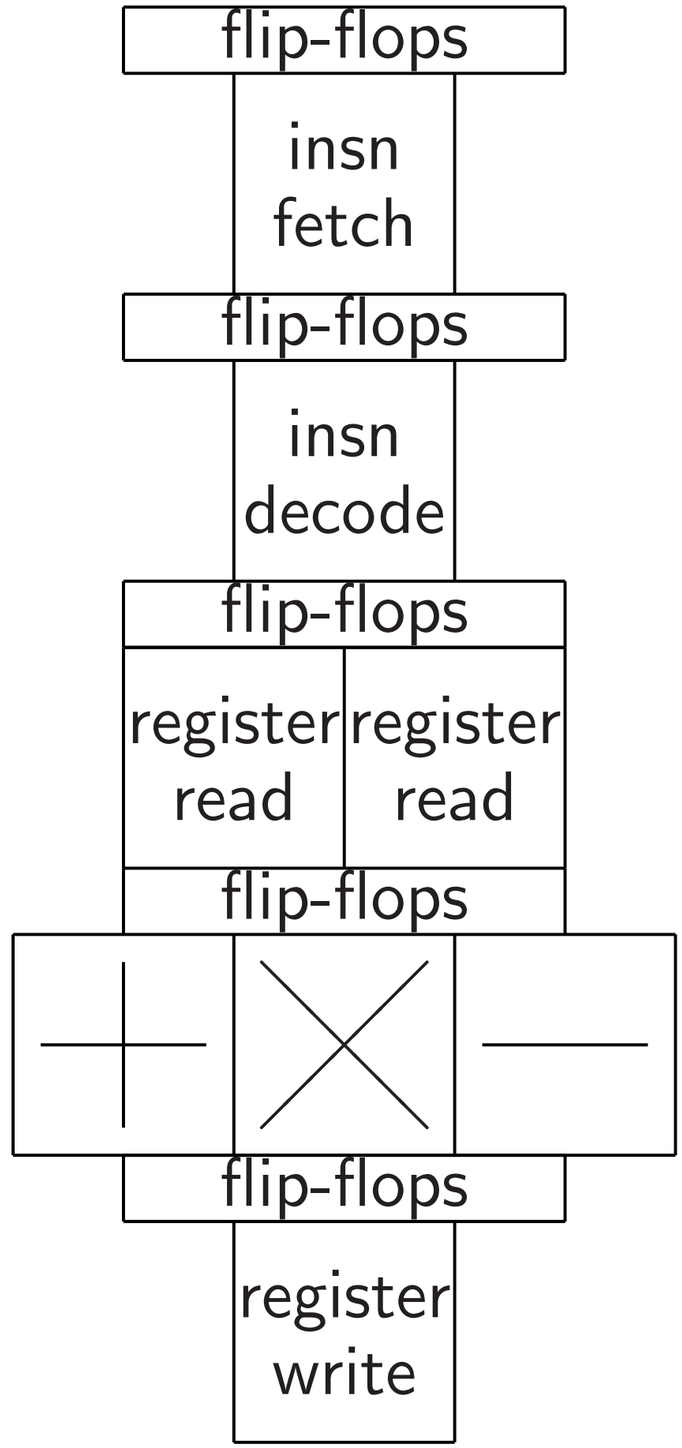


Now have semi-flexible CPU:



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

"Pipelining" allows faster cl



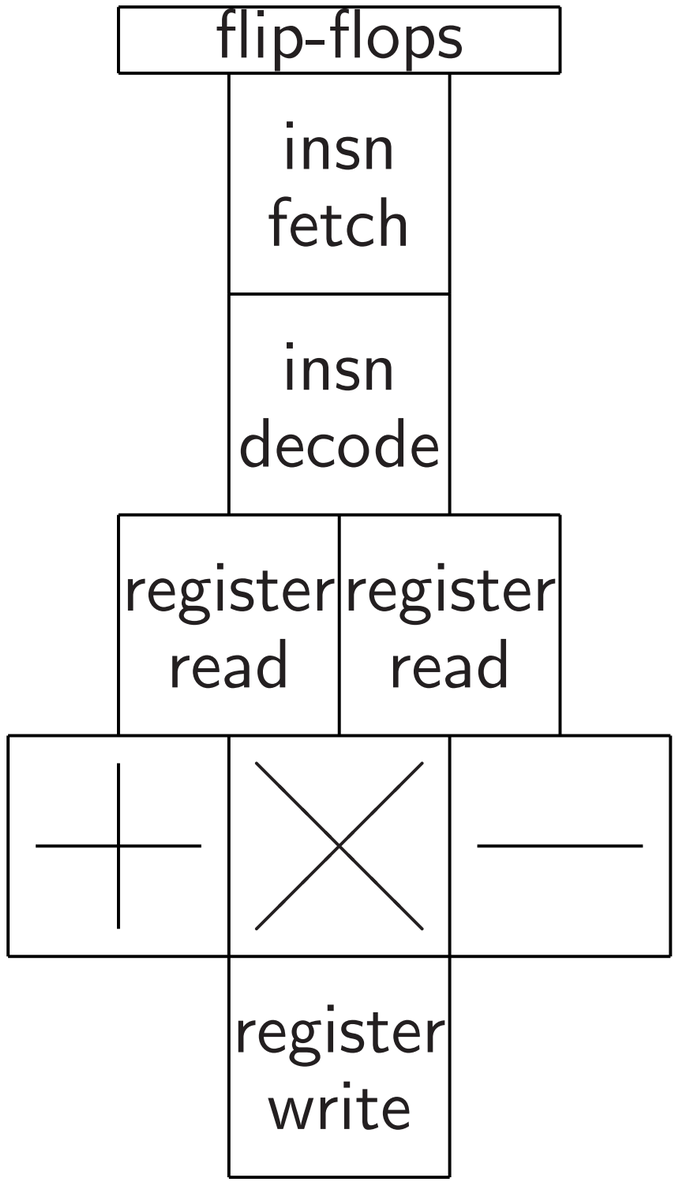
stage

stage

stage

stage

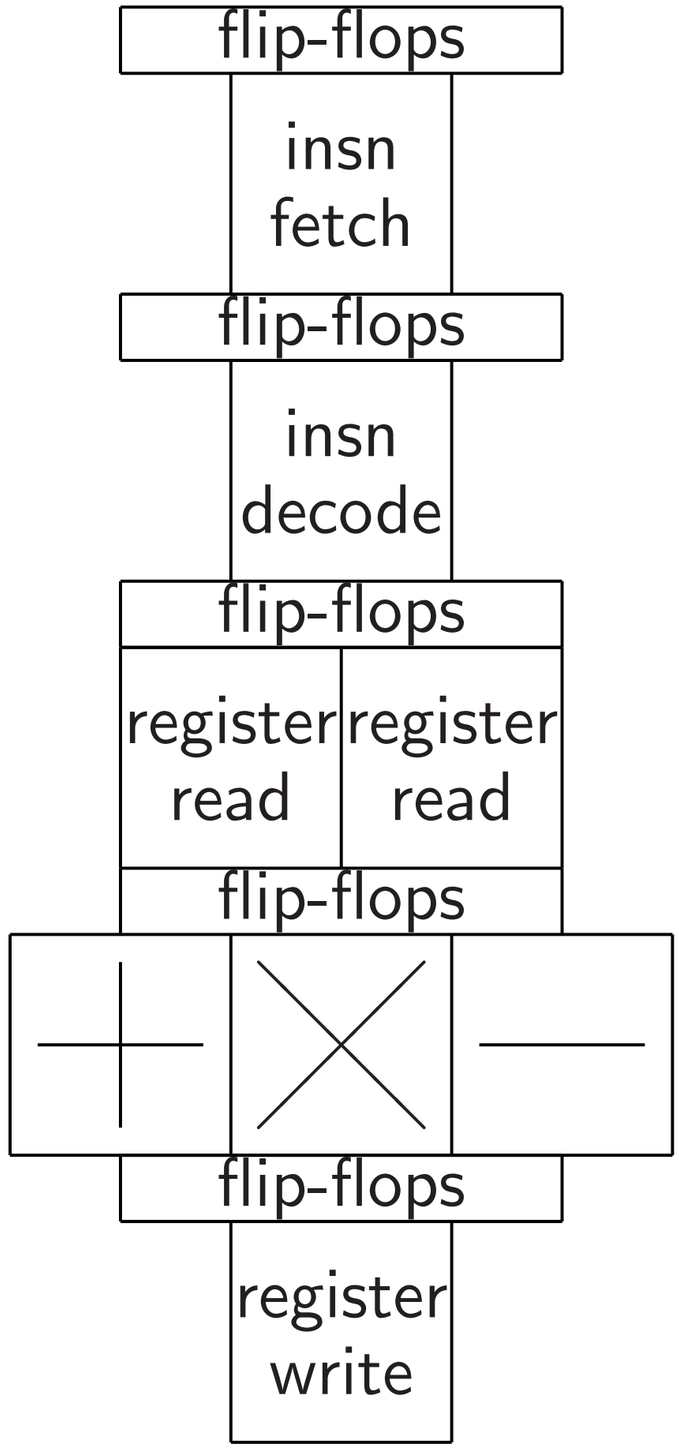
Now have semi-flexible CPU:



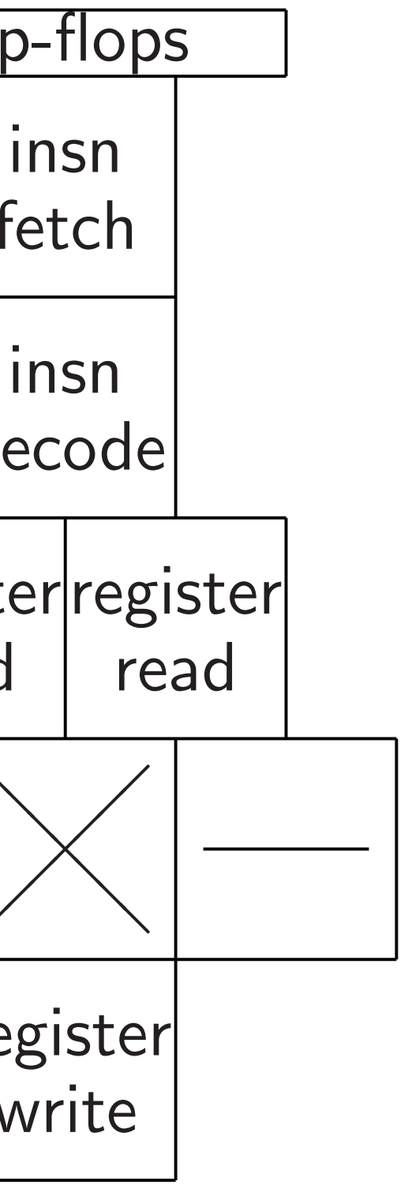
Further flexibility is useful:

e.g., logic instructions.

"Pipelining" allows faster clock:



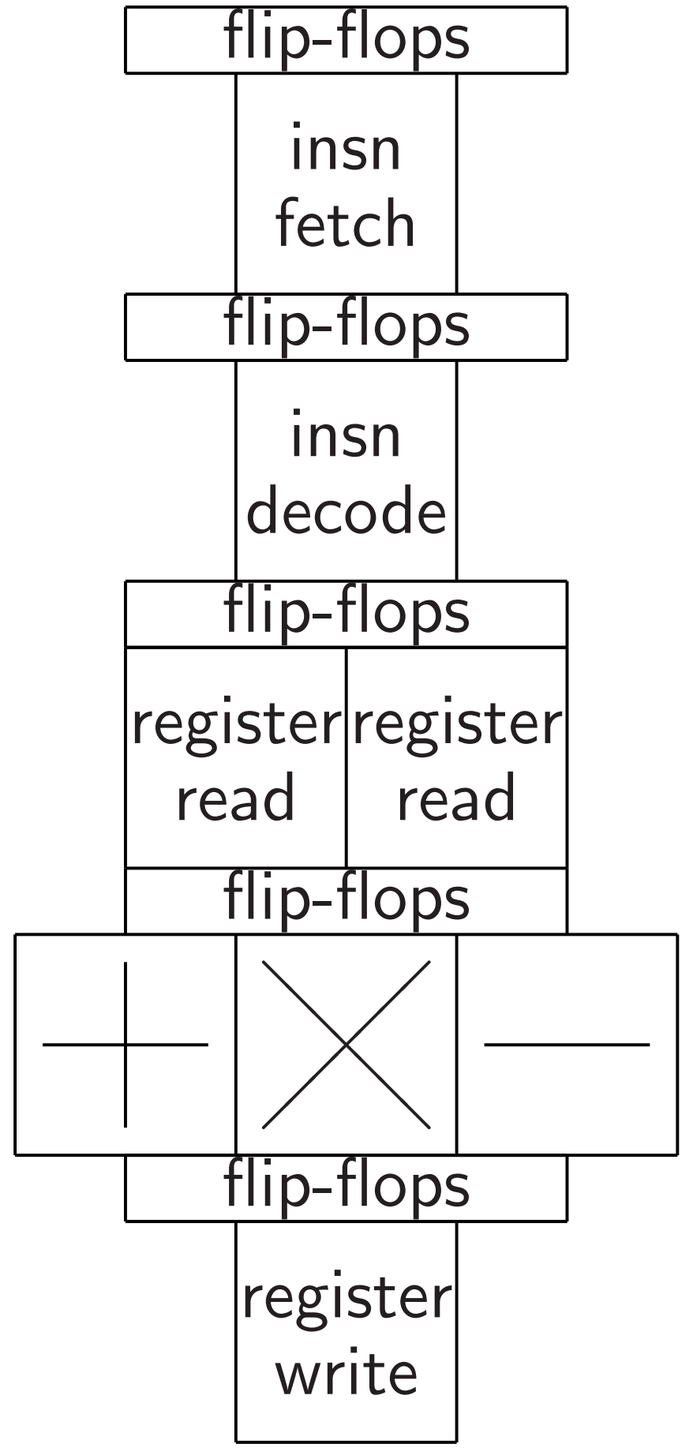
ve semi-flexible CPU:



flexibility is useful:

ic instructions.

"Pipelining" allows faster clock:



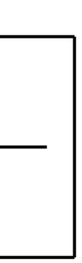
Goal: St
one tick

Instructi
reads ne
feeds p'

After ne
instructi
uncompr
while ins
reads an

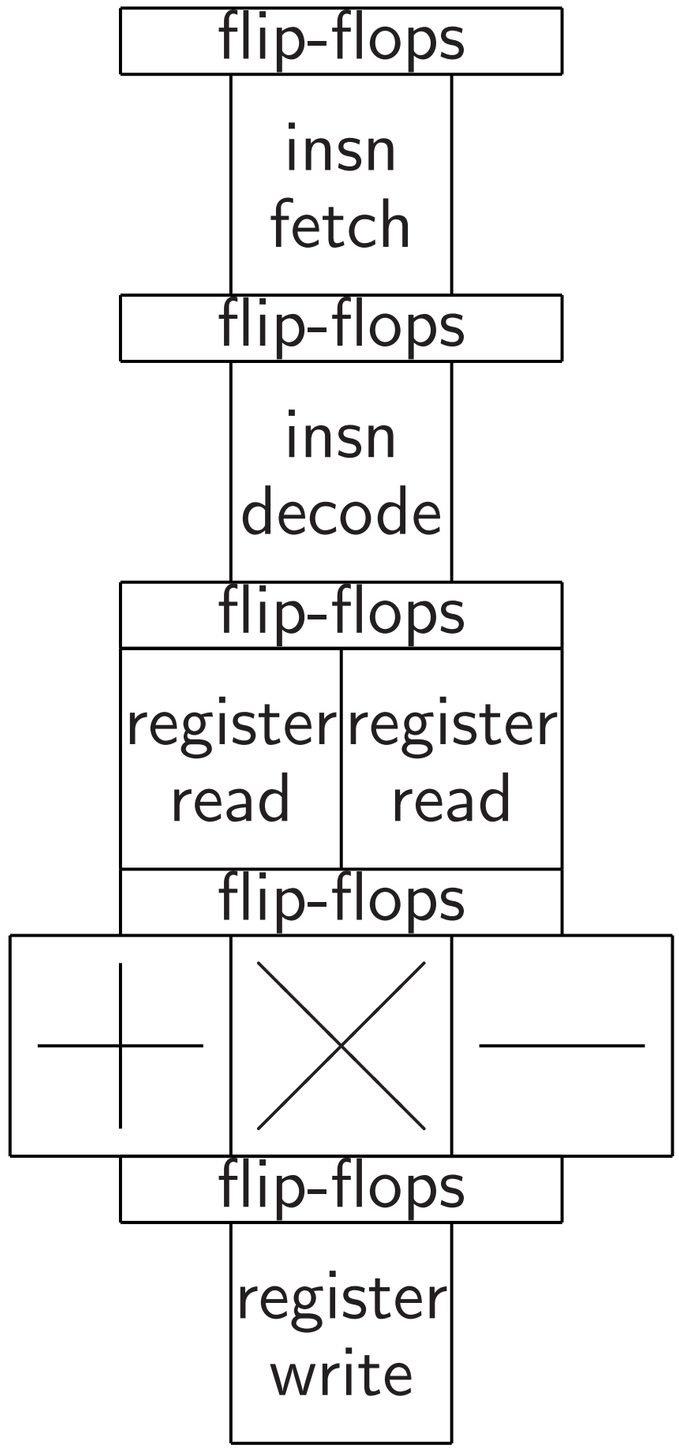
Some ex
Also ext
preserve
e.g., stal

xible CPU:



is useful:
ions.

"Pipelining" allows faster clock:



Goal: Stage n han
one tick after stag

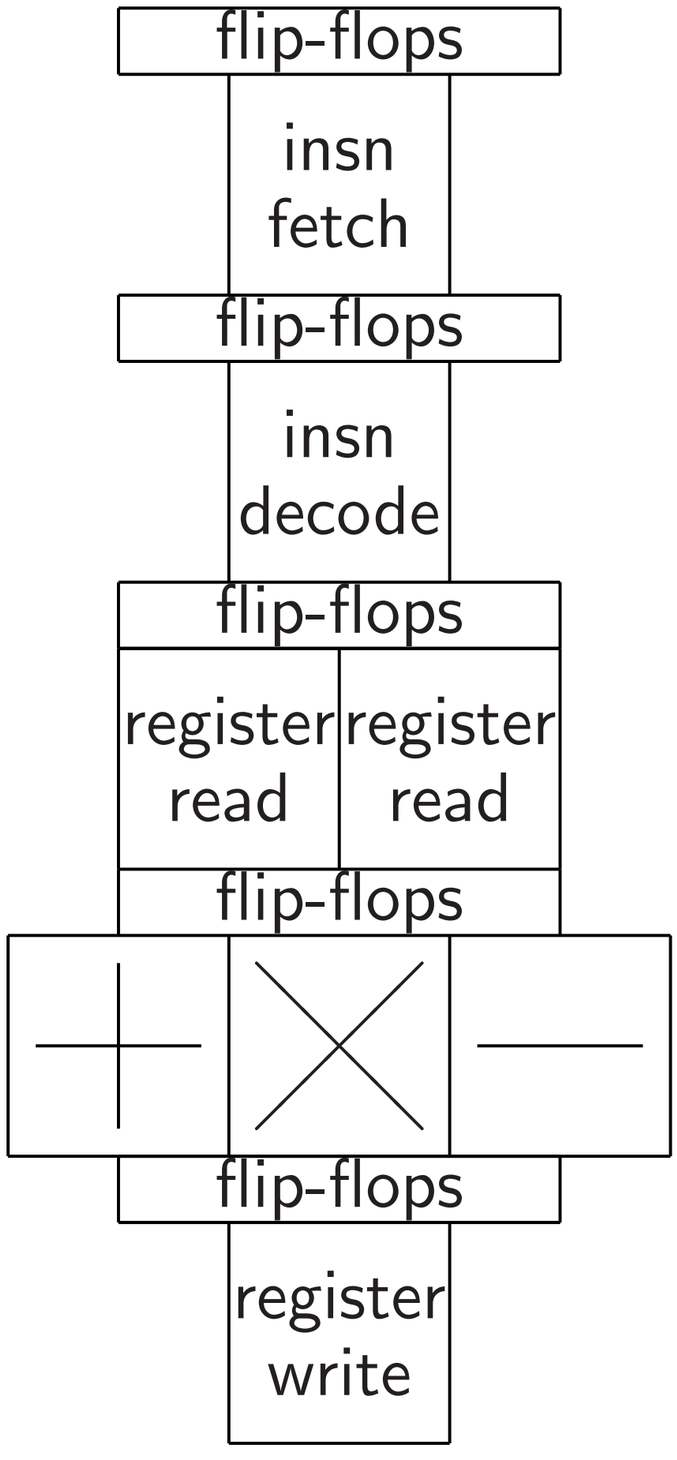
Instruction fetch
reads next instruct
feeds p' back, sen

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

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

:

"Pipelining" allows faster clock:



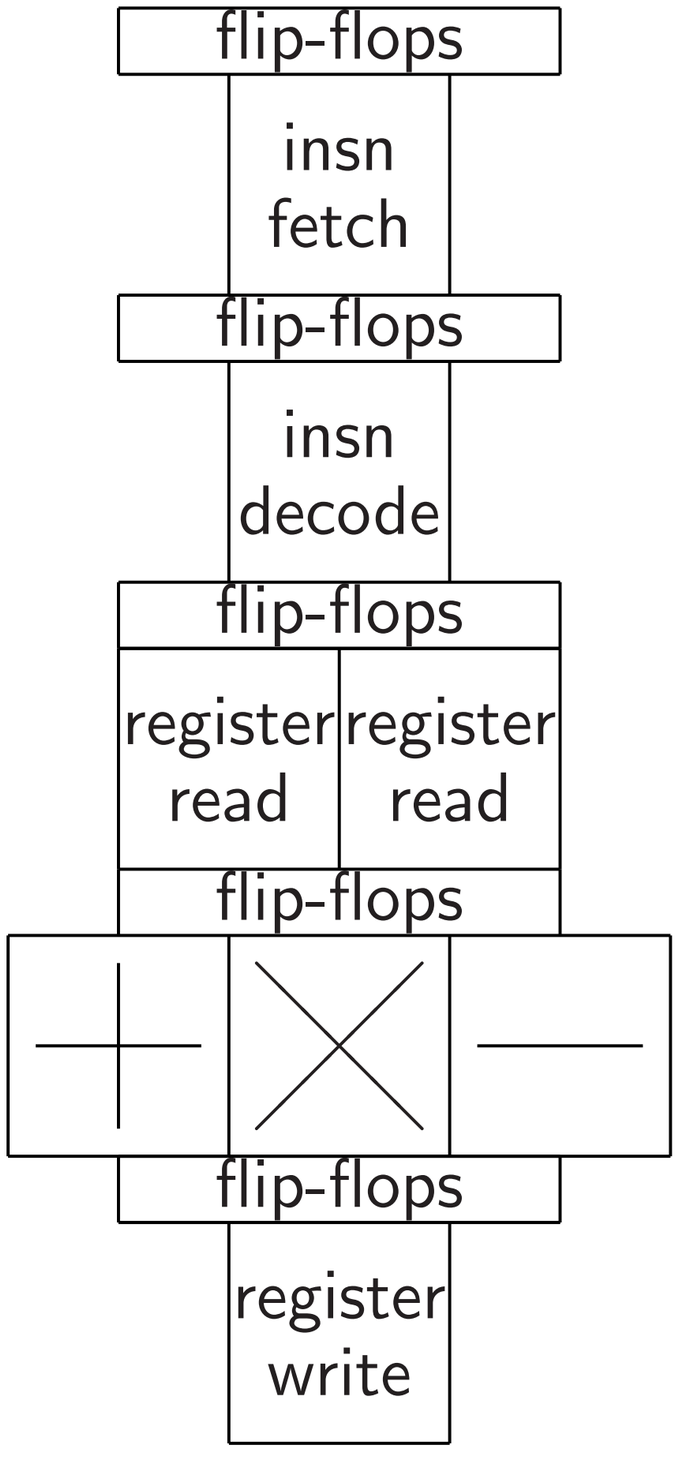
Goal: Stage n handles instru
one tick after stage $n-1$.

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

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

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

"Pipelining" allows faster clock:



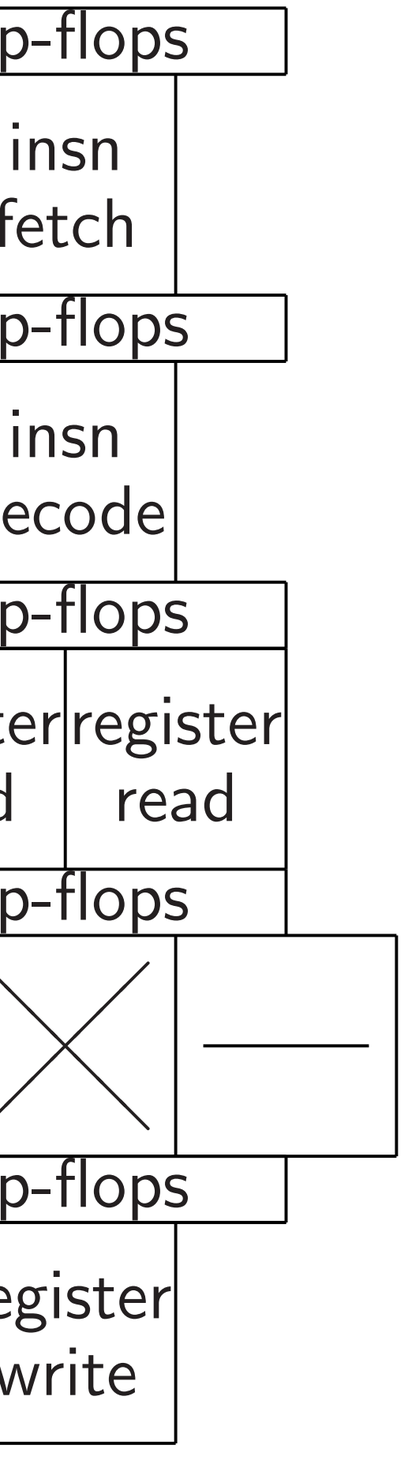
Goal: Stage n handles instruction one tick after stage $n-1$.

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

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

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

ing" allows faster clock:



stage 1

stage 2

stage 3

stage 4

stage 5

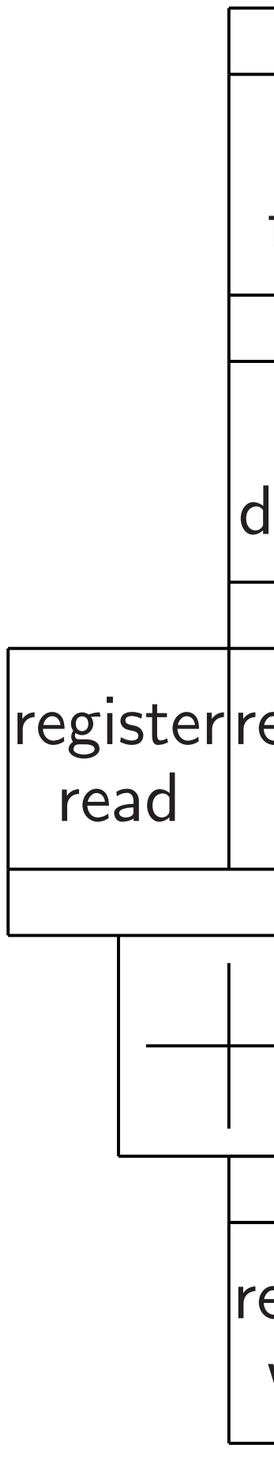
Goal: Stage n handles instruction one tick after stage $n-1$.

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

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

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

"Supersc



s faster clock:

stage 1

stage 2

stage 3

stage 4

stage 5

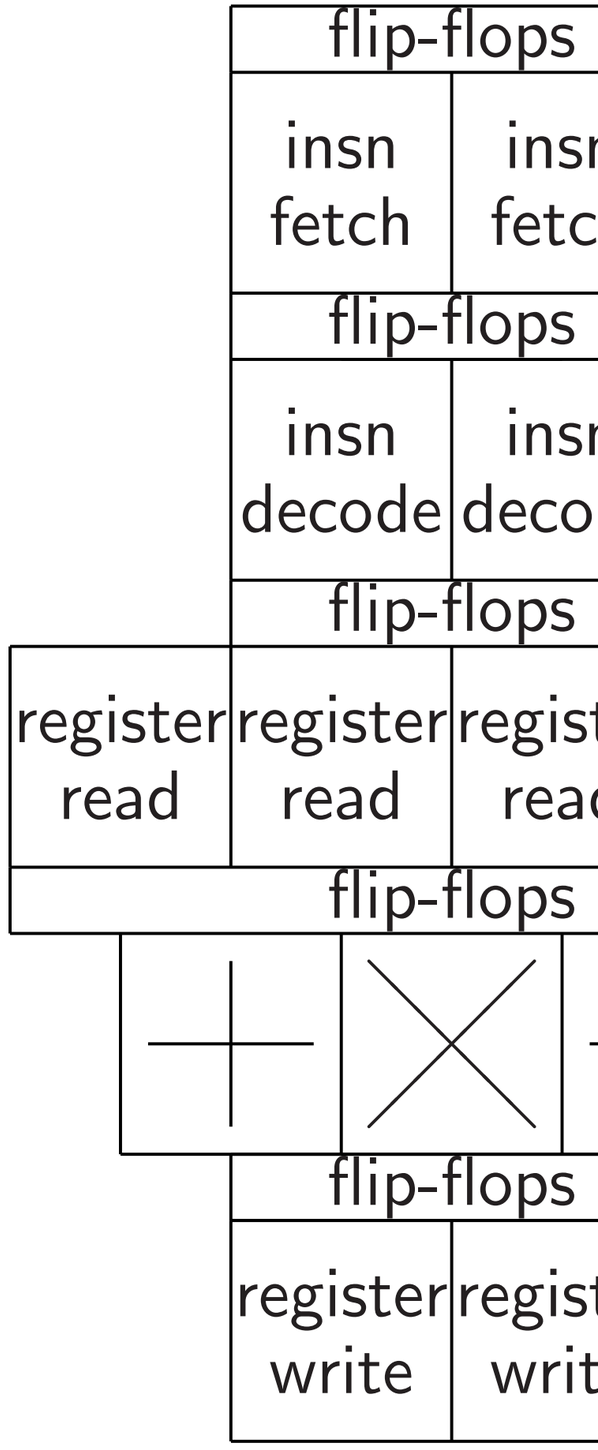
Goal: Stage n handles instruction one tick after stage $n - 1$.

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

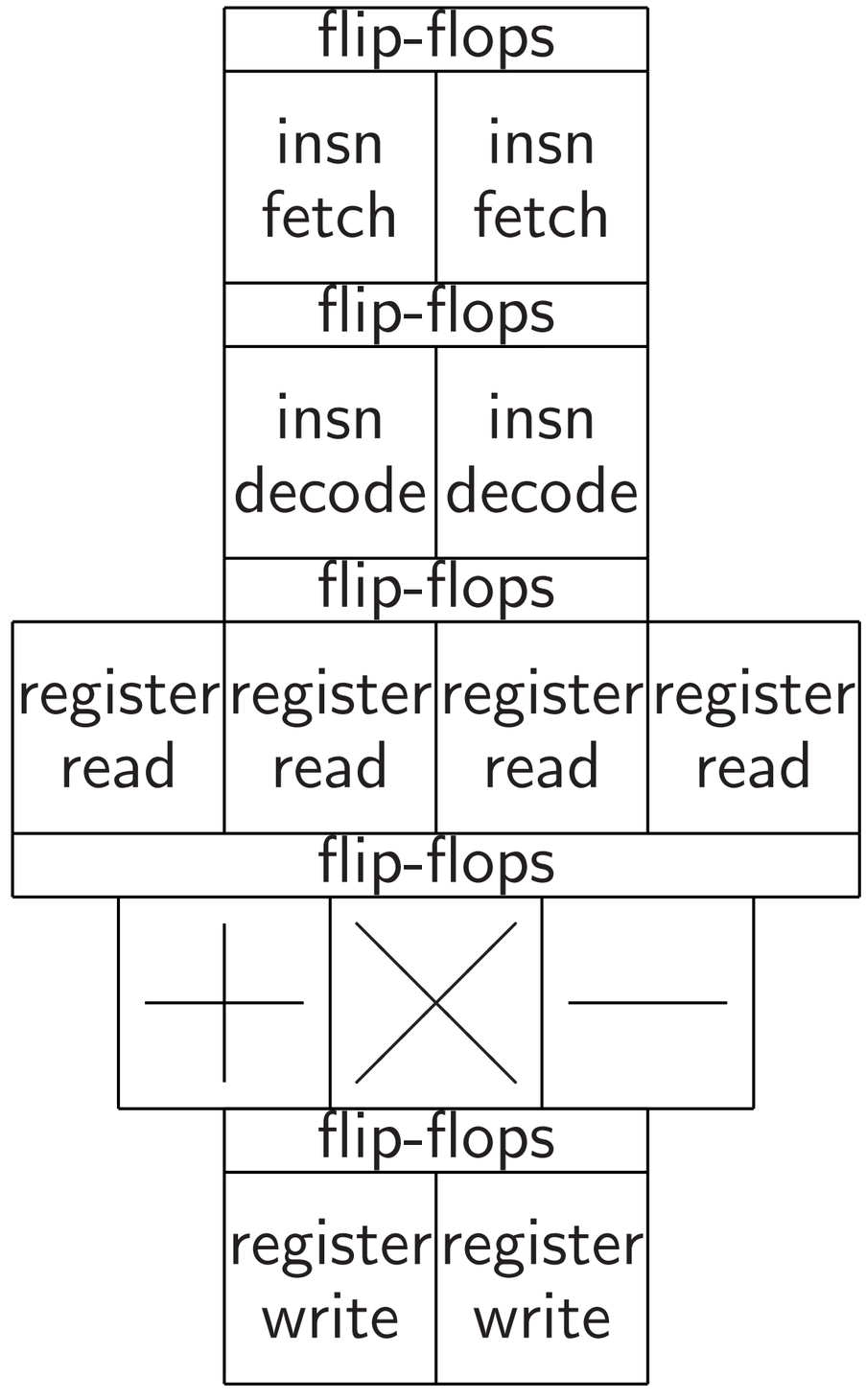
Goal: Stage n handles instruction one tick after stage $n - 1$.

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:



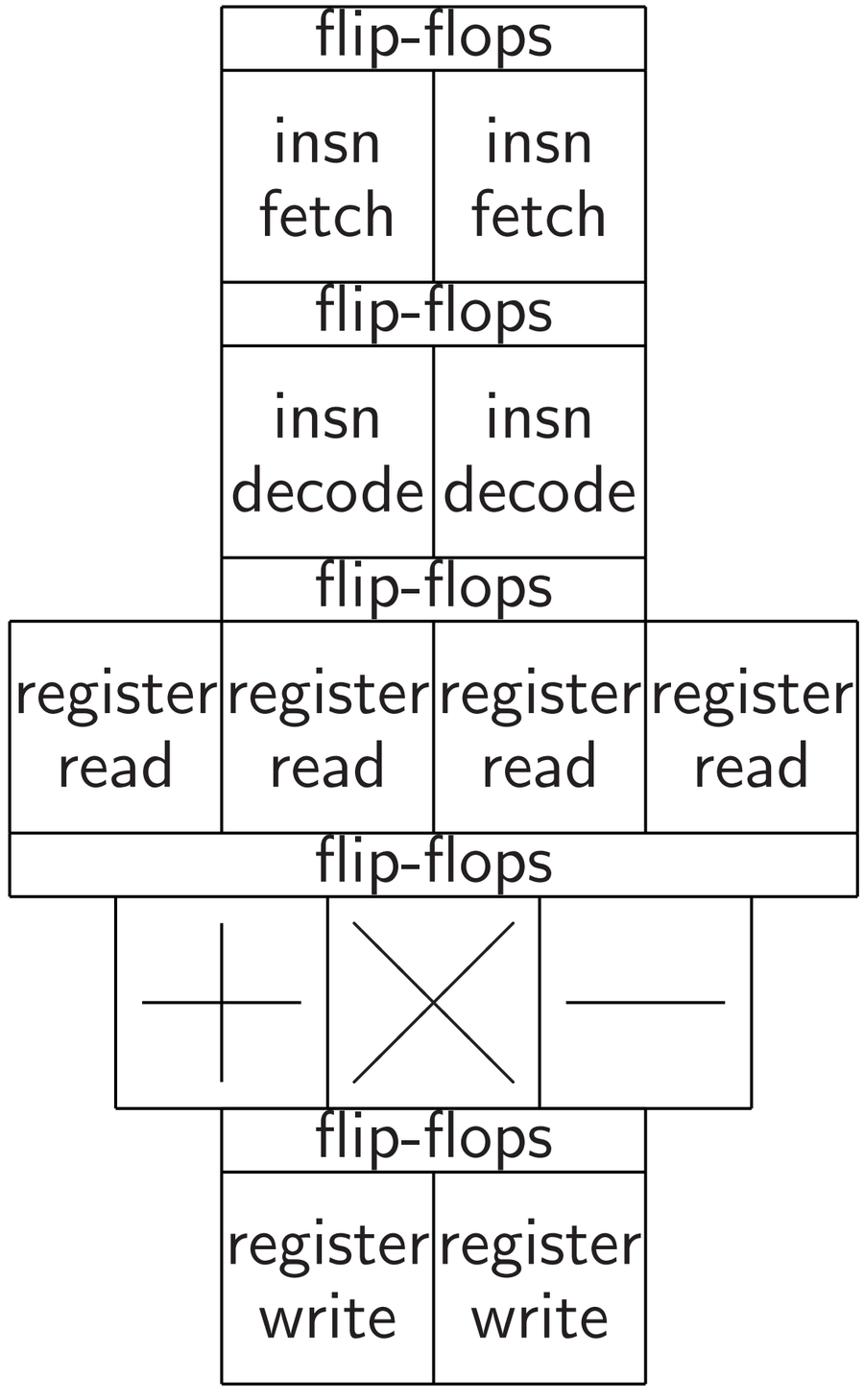
Goal: Stage n handles instruction one tick after stage $n - 1$.

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:



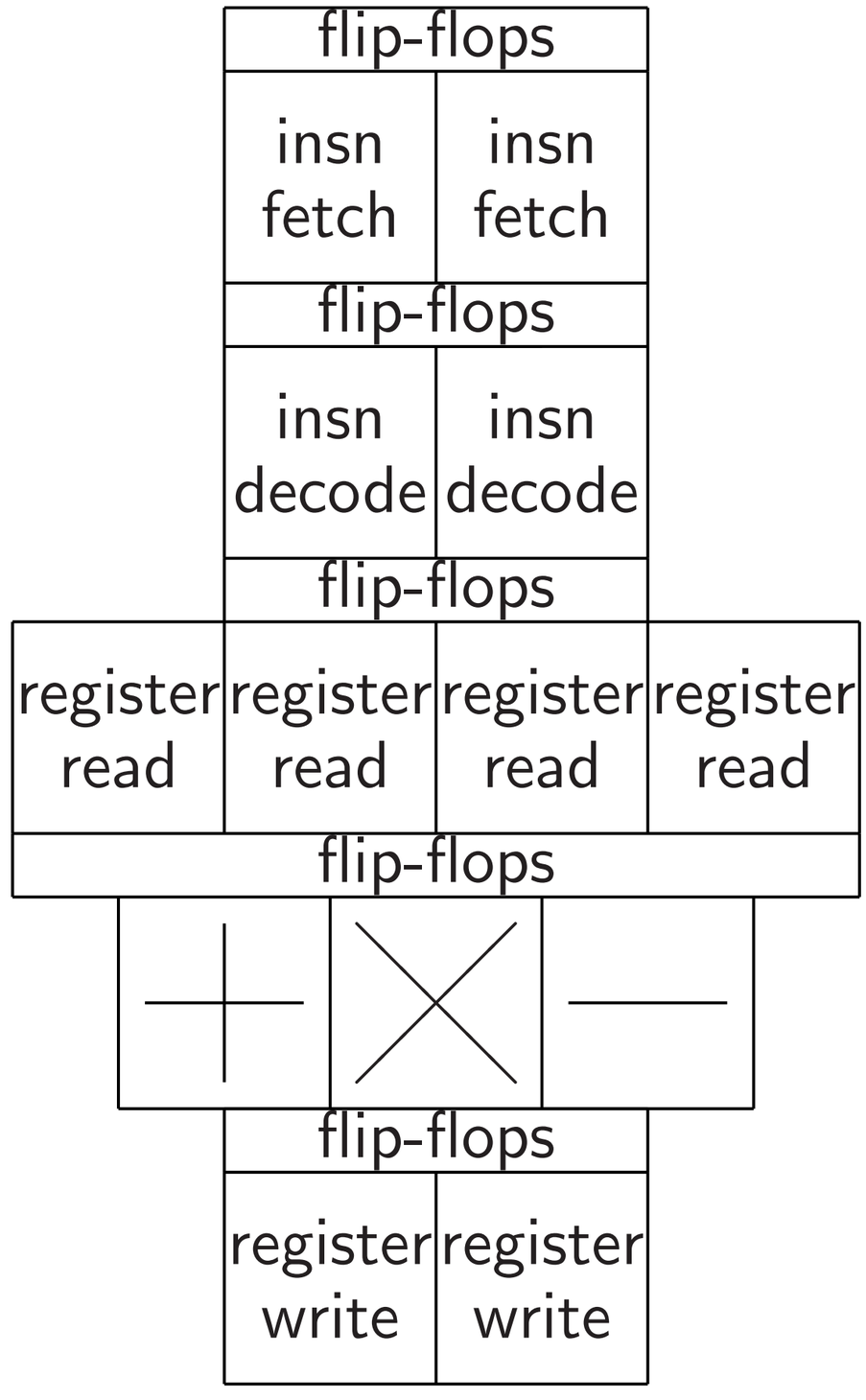
tage n handles instruction
after stage $n-1$.

on fetch
xt instruction,
back, sends instruction.

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

xtra 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 $n-1$.

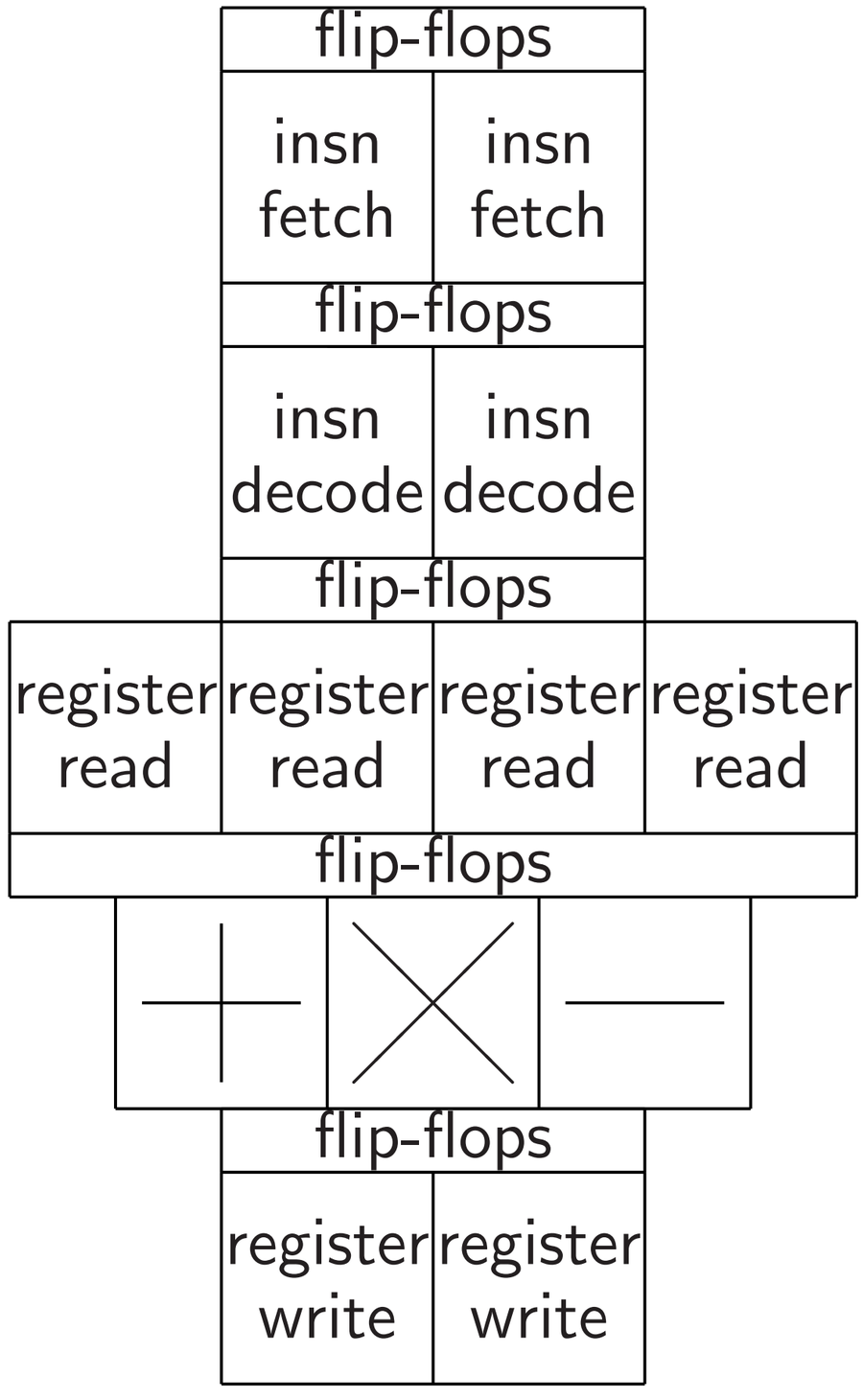
tion,
ds instruction.

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

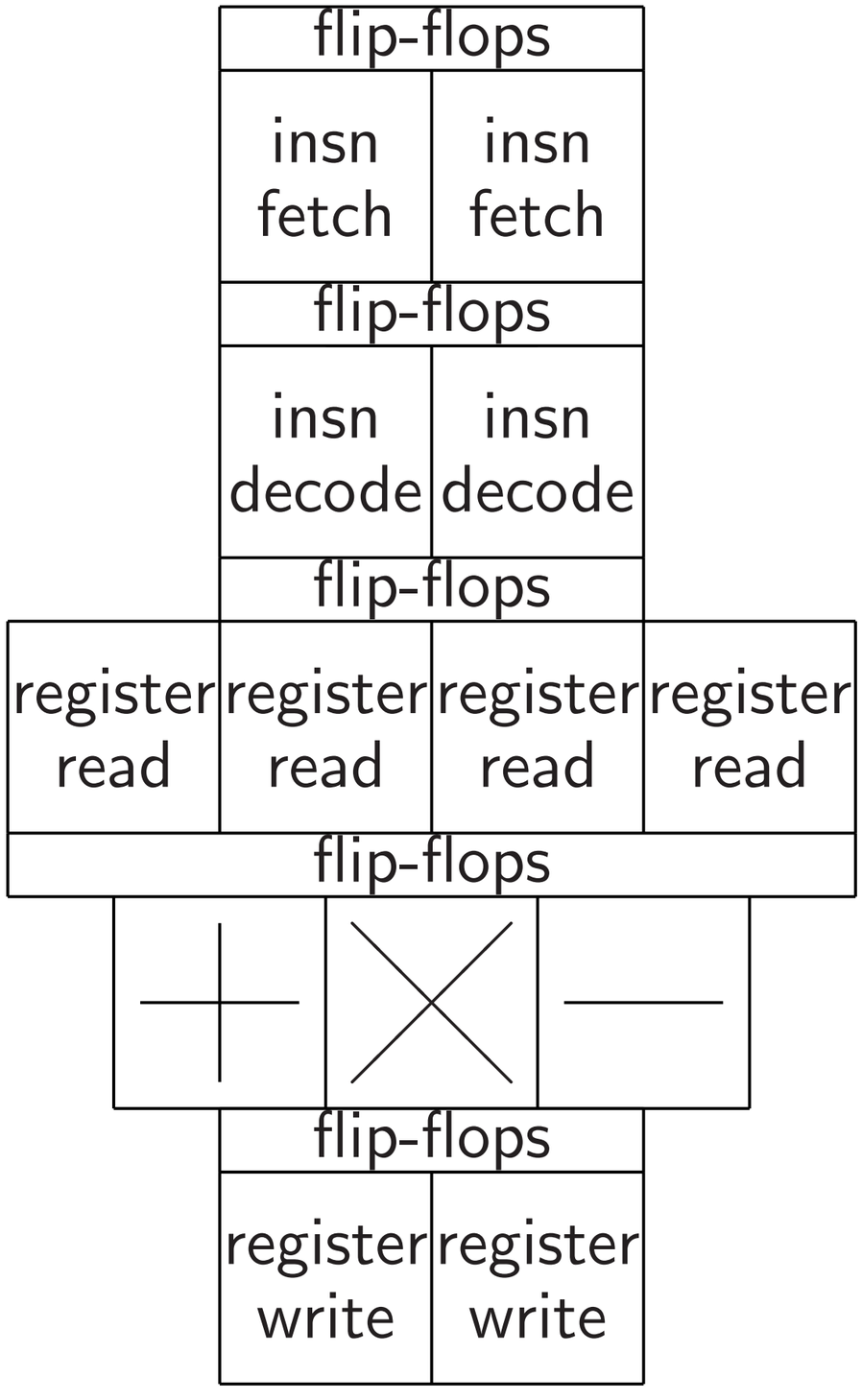
tion.

n,

cs:

e.

"Superscalar" processing:



"Vector" processing:

Expand each 32-bit integer

into n-vector of 32-bit integ

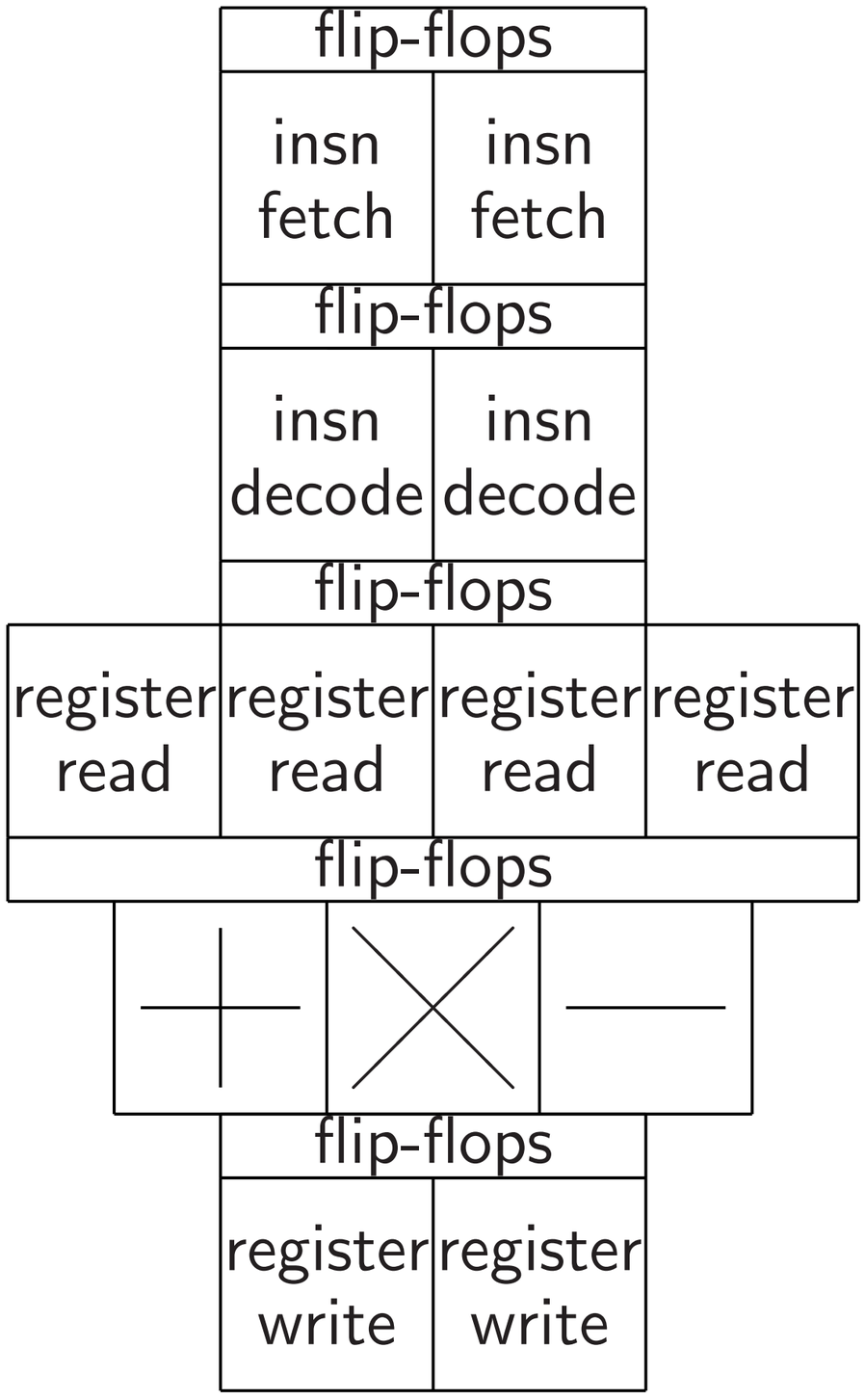
ARM "NEON" has $n = 4$;

Intel "AVX2" has $n = 8$;

Intel "AVX-512" has $n = 16$

GPUs have larger n.

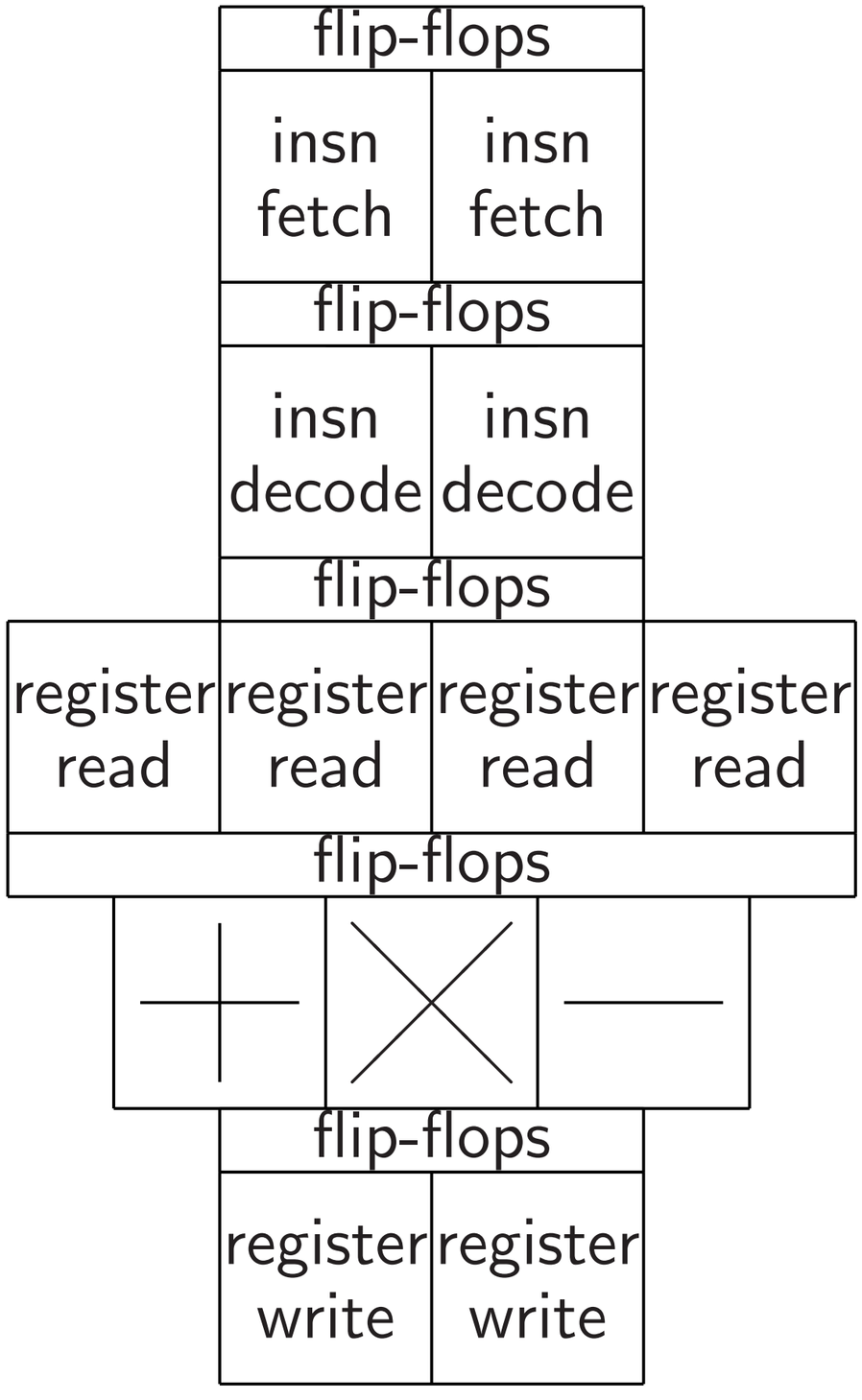
“Superscalar” processing:



“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:

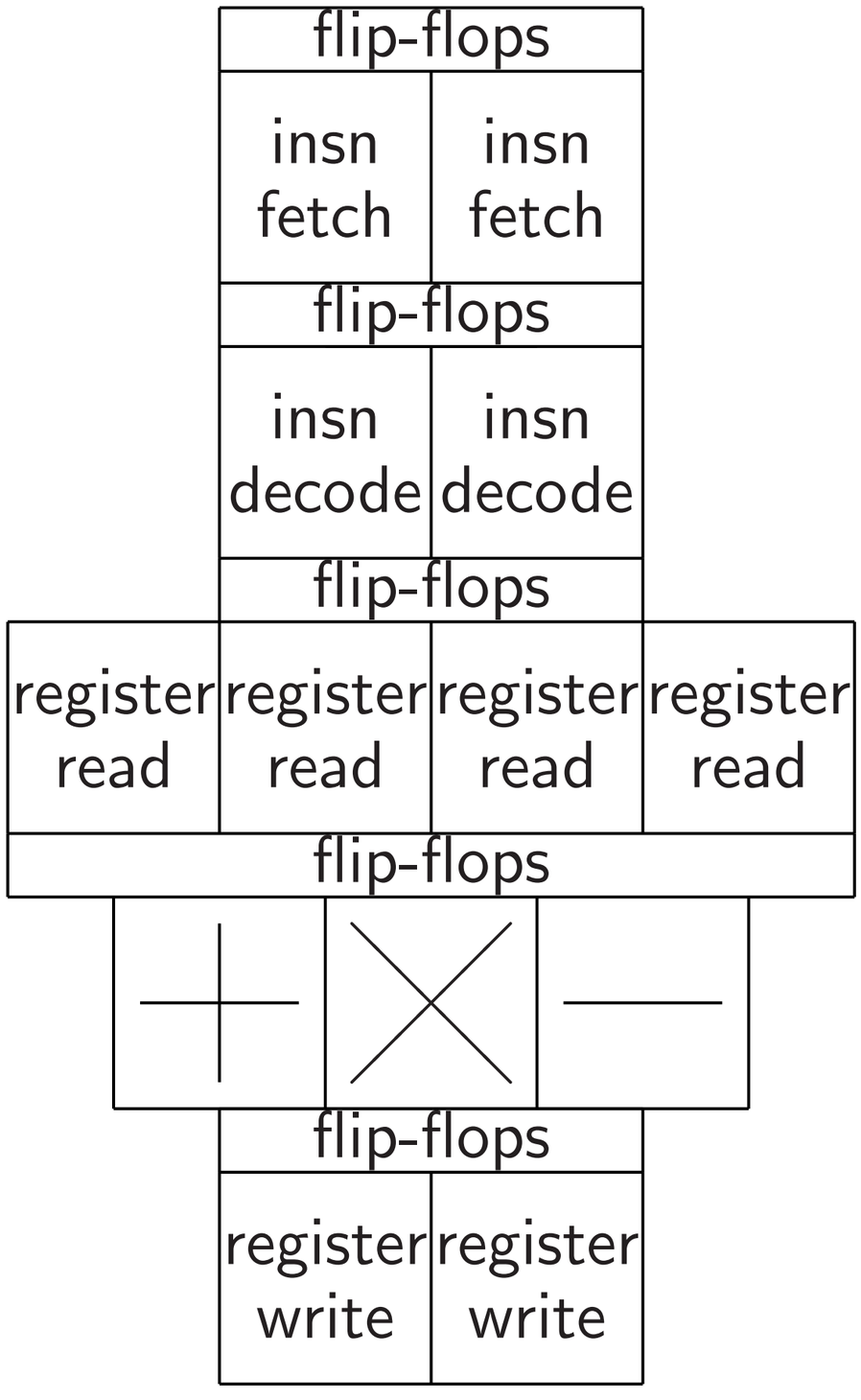


“Vector” processing:

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:



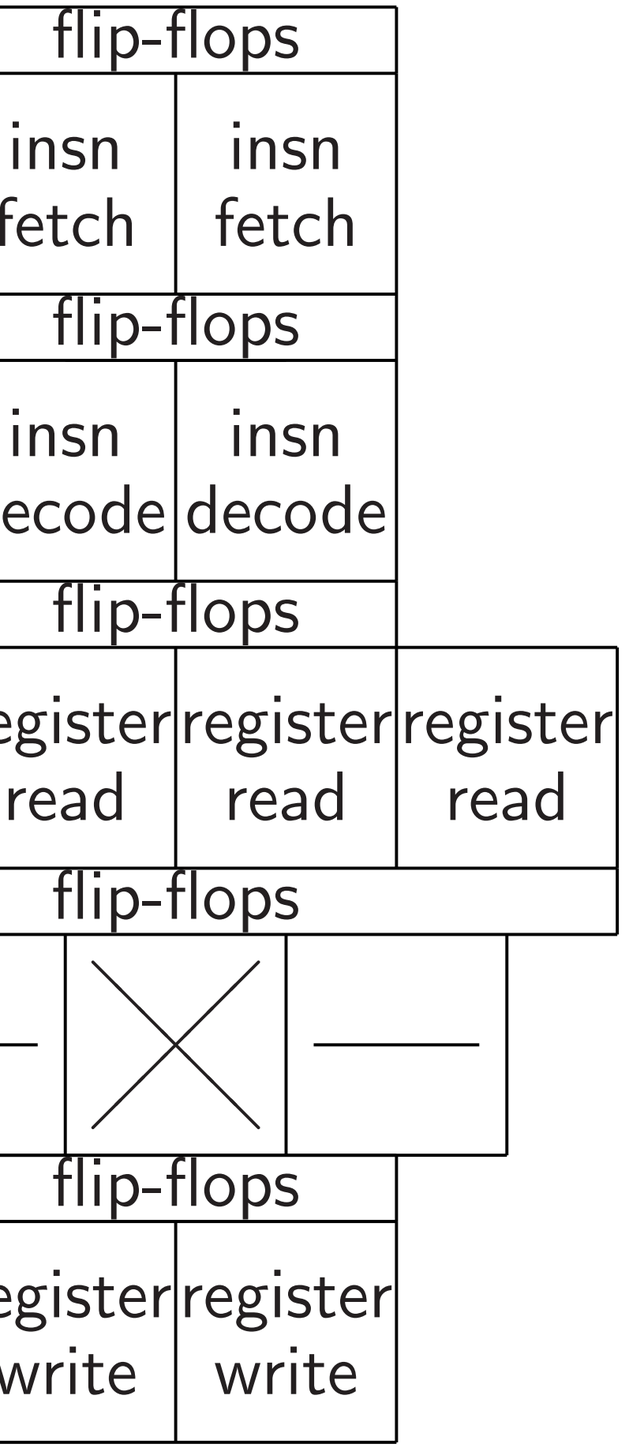
“Vector” processing:

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.

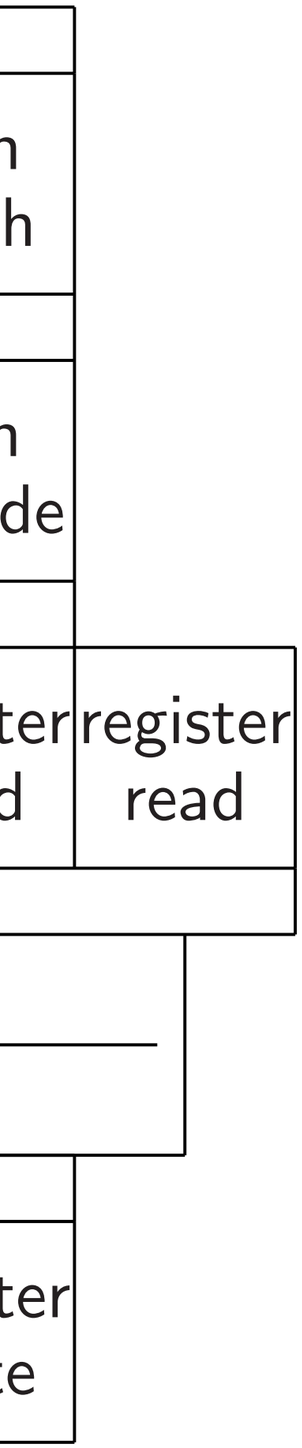
Network

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?

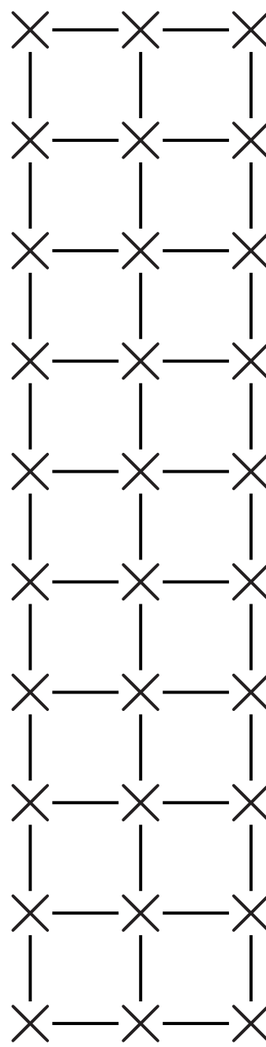
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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 m
each of
with nea



ng:

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

cuits,
cuits.
s insn circuits.

gher-level
ta structures.

Network on chip: the mesh

How expensive is sorting?

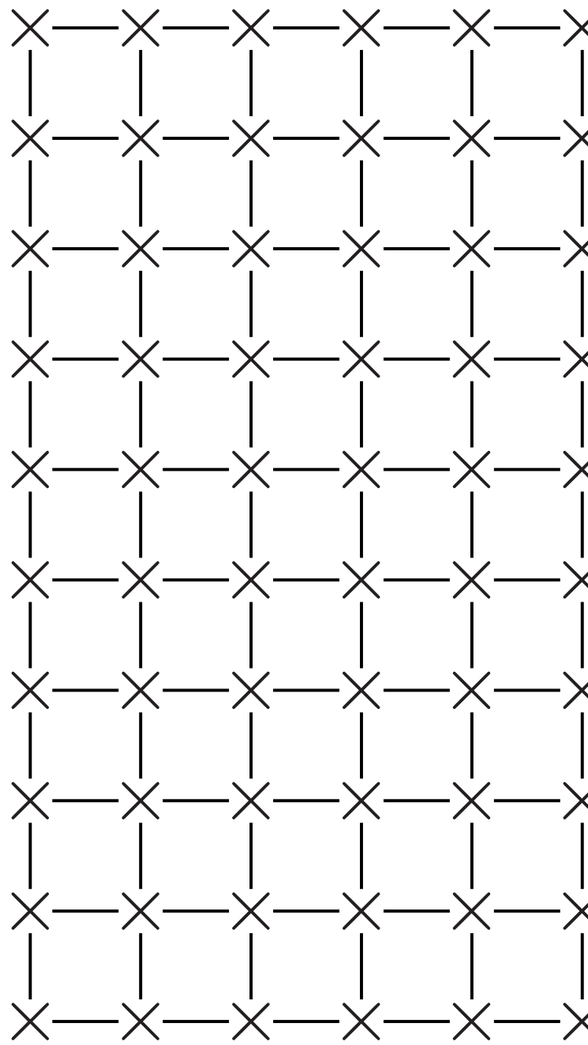
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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-neighbo



ers.

;

uits.

res.

Network on chip: the mesh

How expensive is sorting?

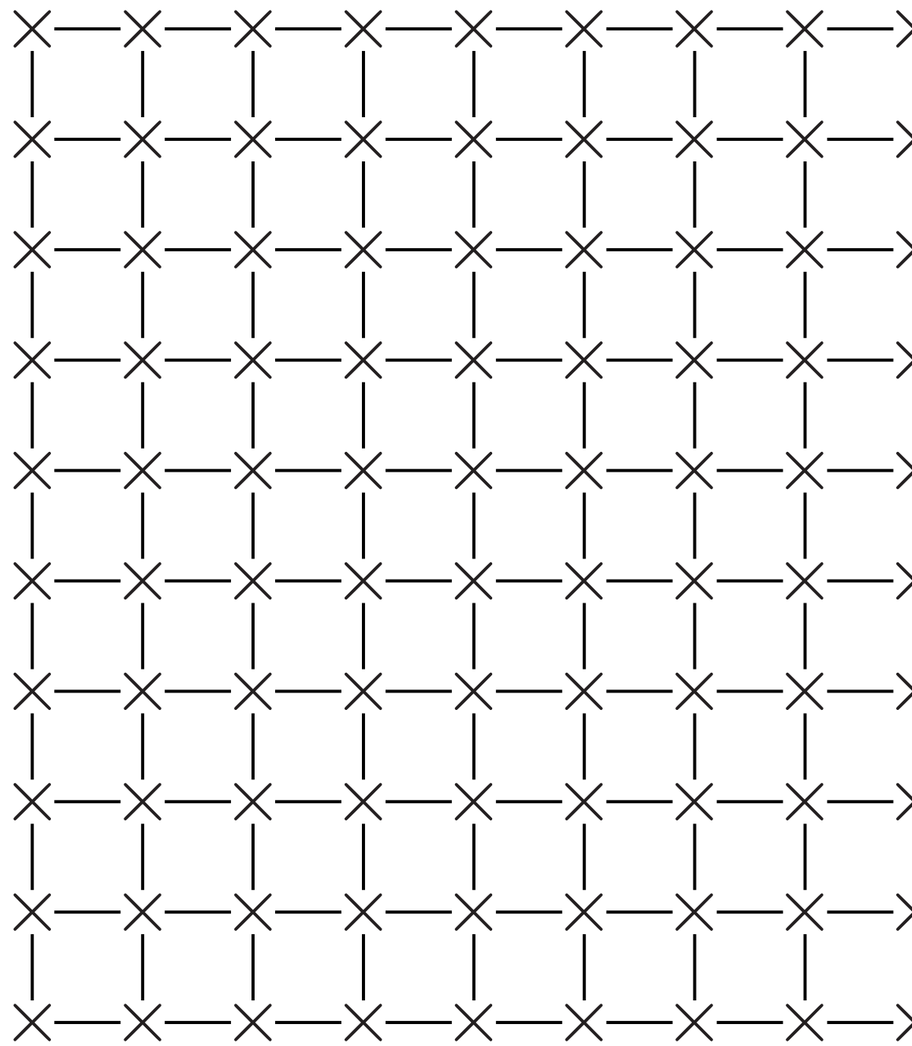
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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?

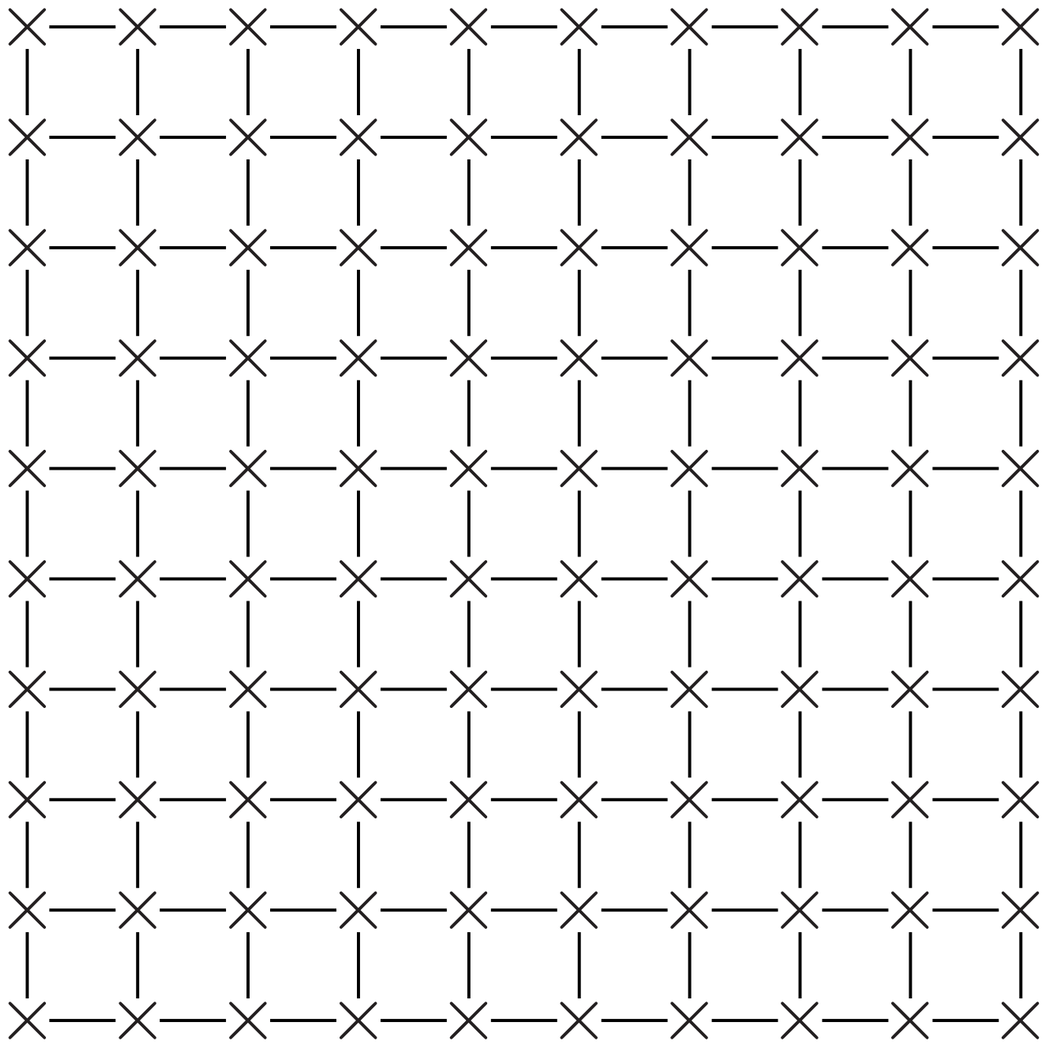
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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?

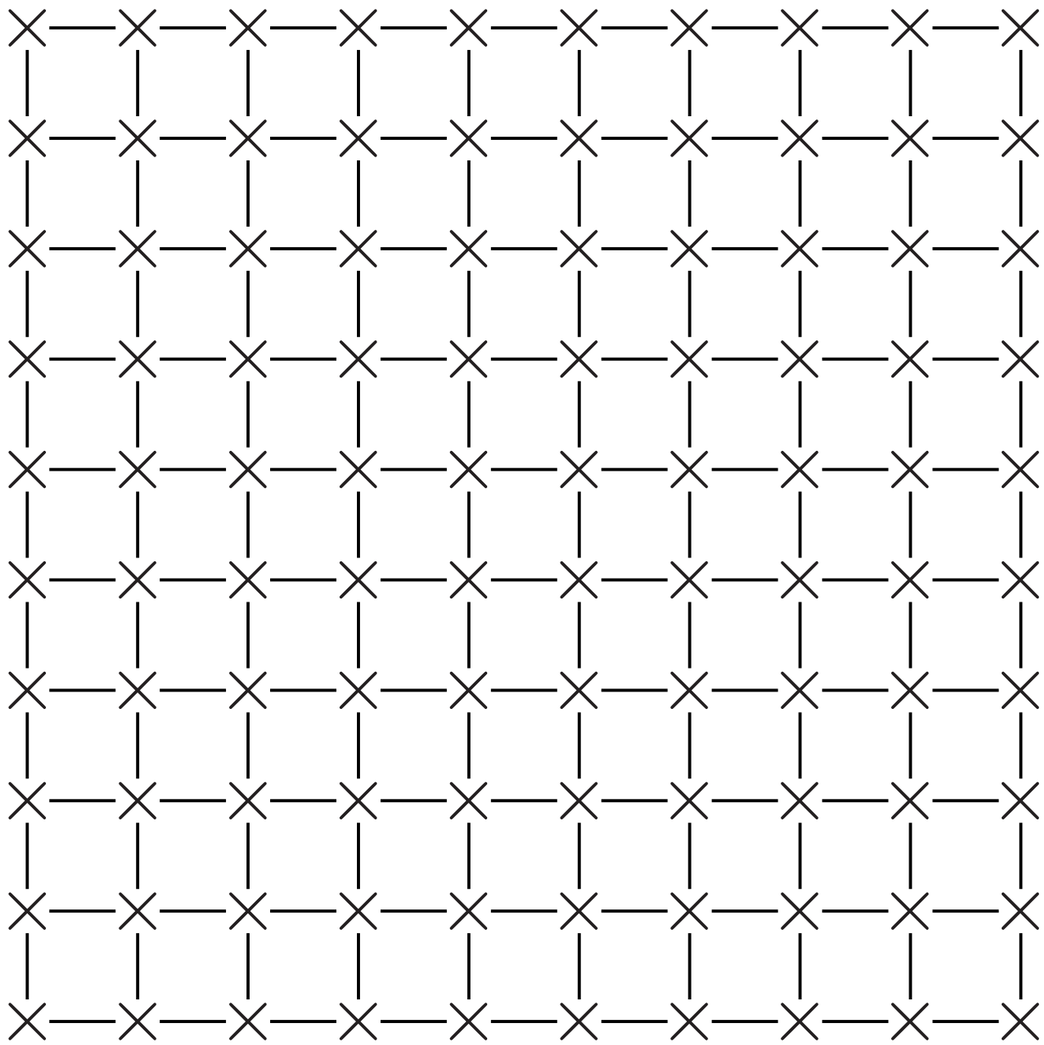
rray of n numbers.
mber in $\{1, 2, \ldots, n^2\}$,
ted in binary.

array of n numbers,
sing 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 $n^{0.5+}$

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

the mesh

sorting?

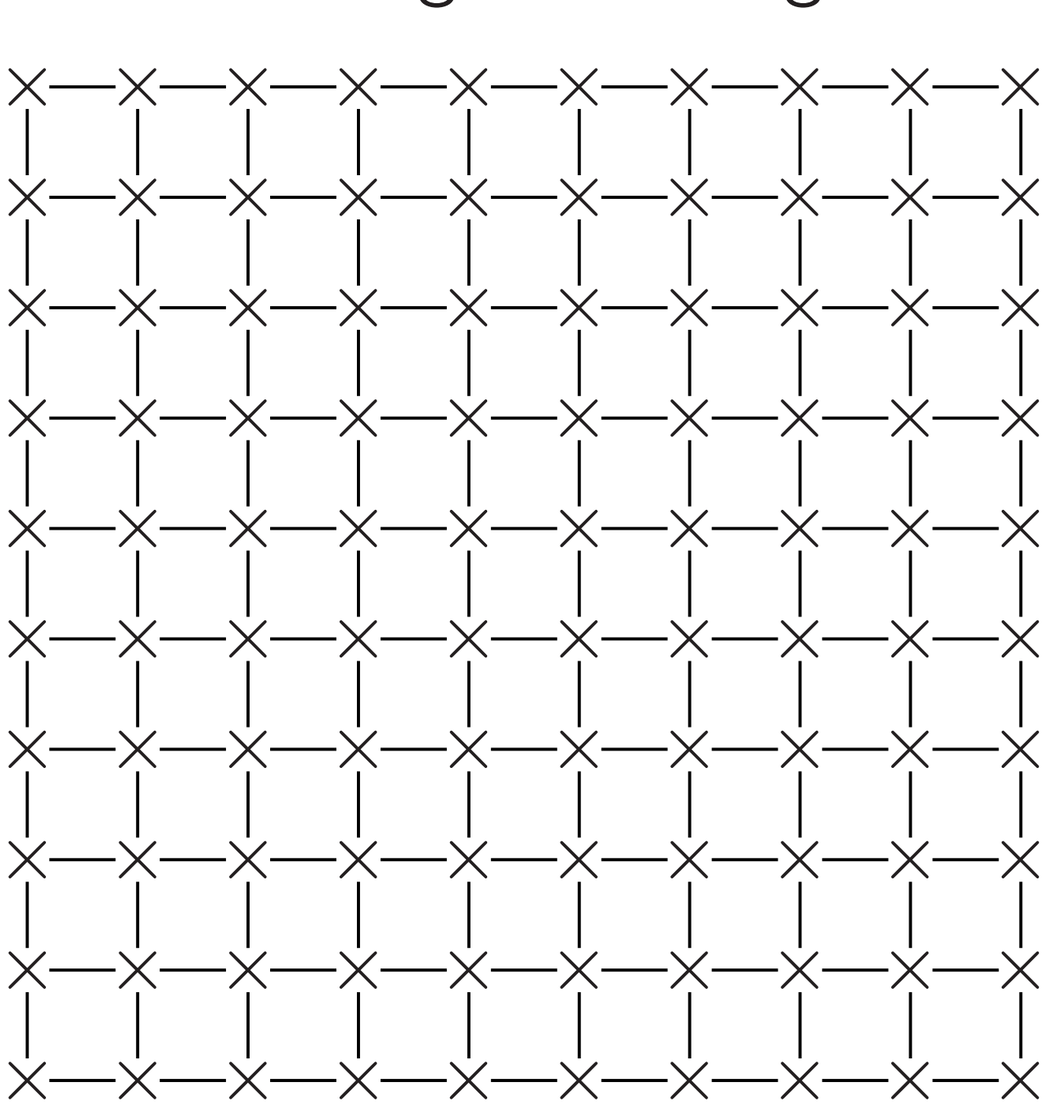
numbers.
$1, 2, \ldots, n^2\}$,
ary.

n numbers,
,
ary;
nput.

sed by
$^{o(1)}$.

me $n = 4^k$.

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 $n^{0.5+o(1)}$ second

- Sort each pair in
  3 1 4 1 5 9 2 6
  1 3 1 4 5 9 2 6
- Sort alternate pa
  1 3 1 4 5 9 2 6
  1 1 3 4 5 2 9 6
- Repeat until num
  equals row lengt

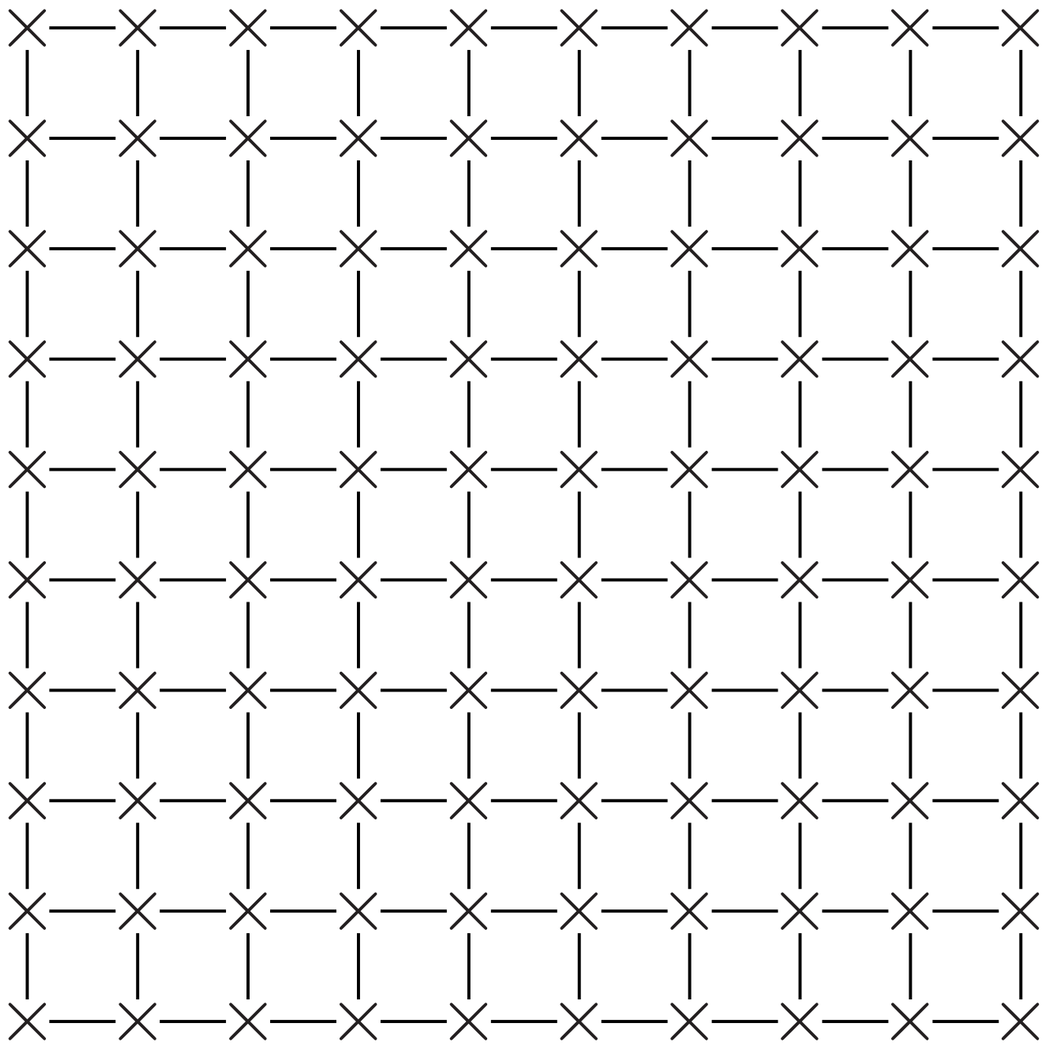
2},

,

k.

33

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

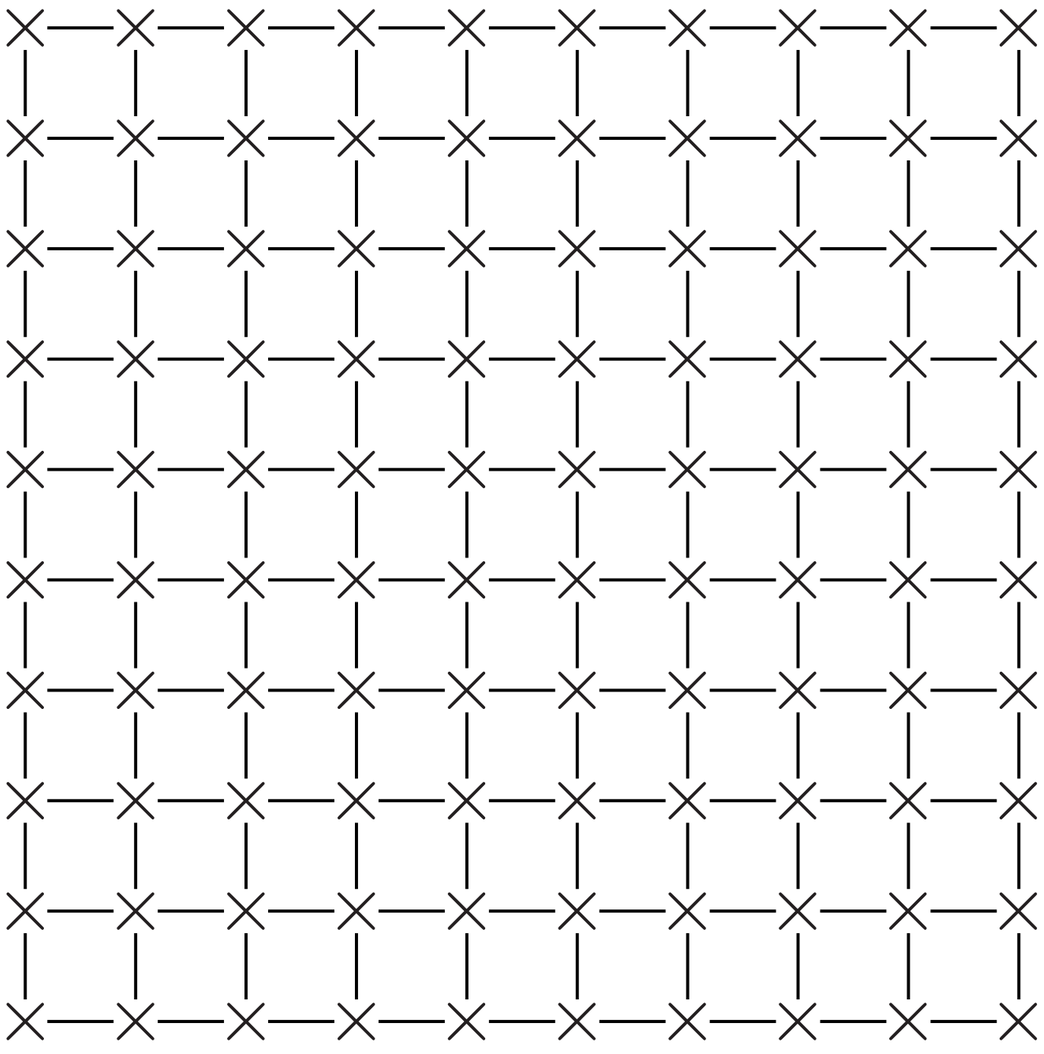


34

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.

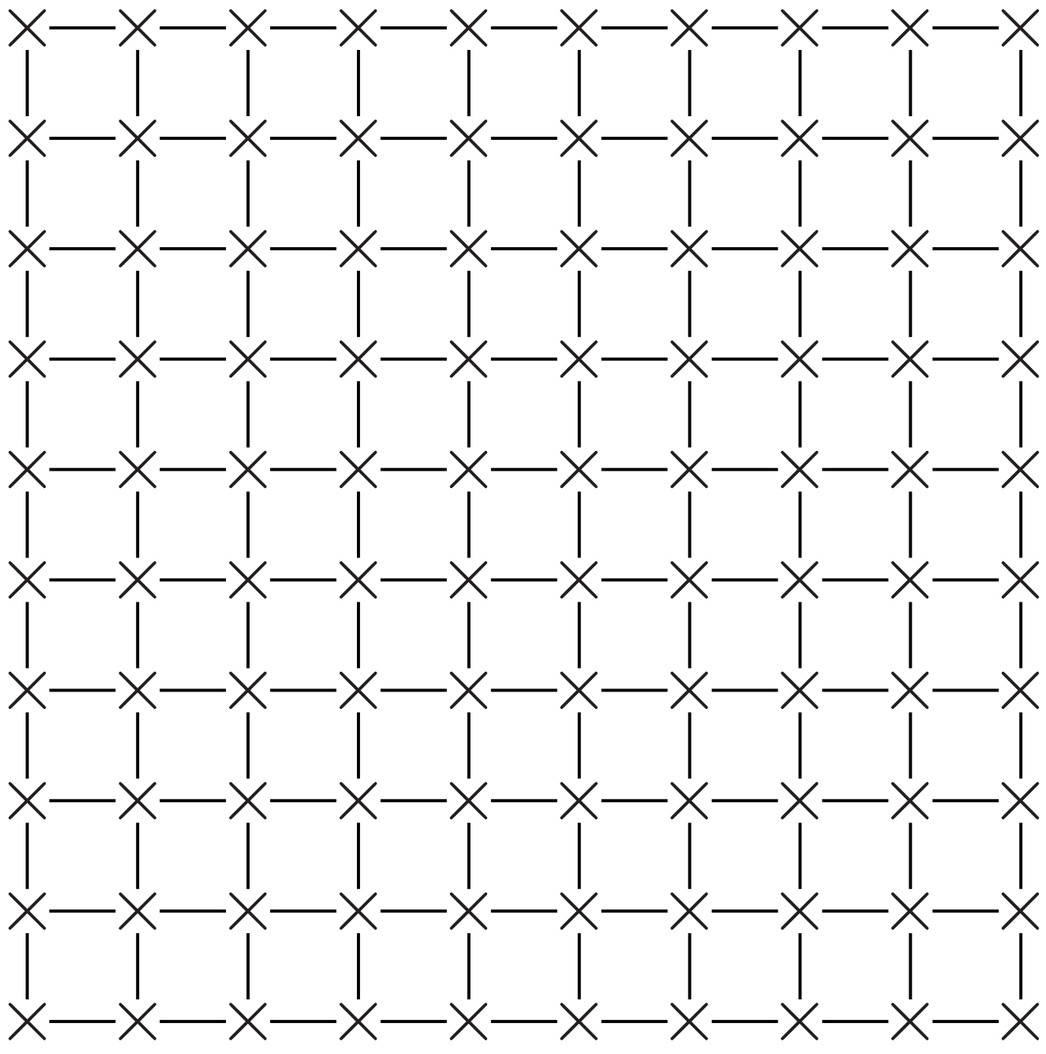
Spread array across
square mesh of n small cells,
each of area $n^{o(1)}$,
with near-neighbor wiring:



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

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

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

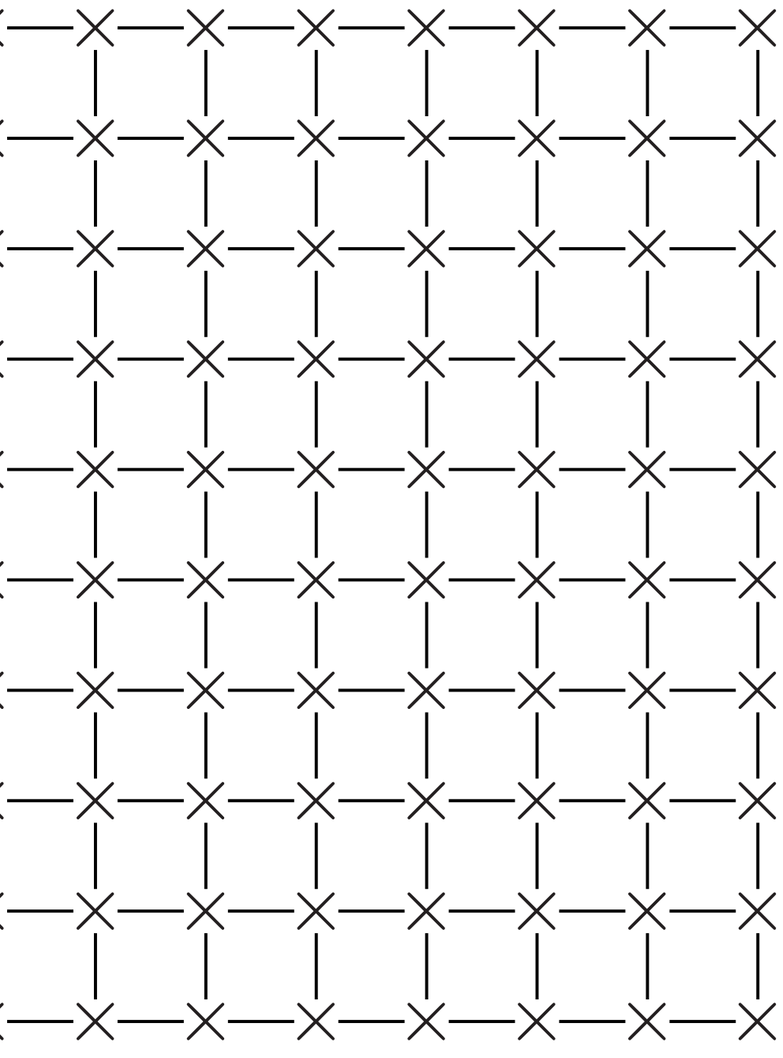


Sort row of $n^{0.5}$ cells
in $n^{0.5+o(1)}$ seconds:

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

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

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



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

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

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

Sort all
in $n^{0.5+$

- Recurs
  in para
- Sort e
- Sort e
- Sort e
- Sort e

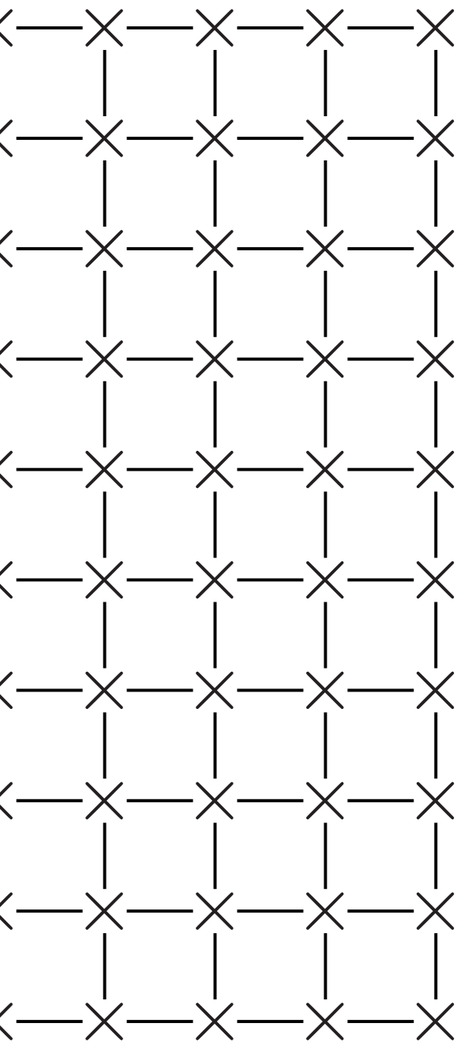
With pr
left-to-ri
for each
that this

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.

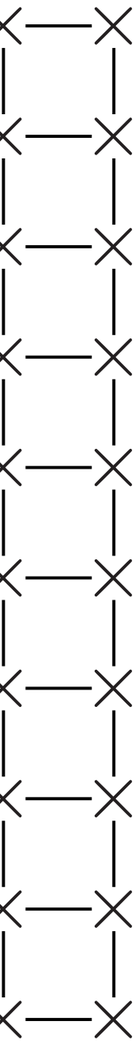
Sort *each* row, in parallel,
in a *total* of $n^{0.5+o(1)}$ seconds.

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

- 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 wh

,



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 exam
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

th.

parallel,

$^{o(1)}$ seconds.

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

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

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

For example, assu

this 8×8 array is

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

allel.

eps

ds.

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

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

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

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

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

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

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

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

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

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

n cells
$^{(1)}$ seconds:

sively sort quadrants
allel, if $n > 1$.
ach column in parallel.
ach row in parallel.
ach column in parallel.
ach row in parallel.

oper choice of
ght/right-to-left
row, can prove
s sorts whole array.

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

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

Recursiv
top $\rightarrow$,

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.
parallel.
n in parallel.
parallel.

e of
to-left
prove
ble array.

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

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

Recursively sort qu

top $\rightarrow$, bottom $\leftarrow$

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

el.

el.

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

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

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

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

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

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

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

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

mple, assume that
8 array is in cells:

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 paralle

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:

0	2	6
7	9	3
2	6	4
7	9	5
	9	7
0	3	7
2	0	9
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 each column
in parallel:

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

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

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

Sort each column
in parallel:

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

Recursively sort quadrants,

top $\rightarrow$, bottom $\leftarrow$:

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

Sort each column

in parallel:

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

vely sort quadrants,
bottom ←:

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

Sort each column
in parallel:

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

Sort eac
alternate

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

uadrants,

:

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

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

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

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

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

Sort each column in parallel:

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

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

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

Sort each column
in parallel:

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

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

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

Sort each column
in parallel:

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

h row in parallel,

ely $\leftarrow$, $\rightarrow$:

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

$\leftarrow$ or $\rightarrow$

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

parallel,

	2	2
2	1	1
4	4	4
3	3	3
6	7	7
5	5	5
0	9	9
0	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	2
3	3	3
4	4	4
5	5	5
6	7	7
8	8	8
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 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 + rin
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.