Engineering cryptographic software

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

This is easy, right?

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

Let's try some examples . . .

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

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

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

```
#define ROUNDS 20
```

then it is easy to change.

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

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

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

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

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

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

Not so easy: Timing attacks

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

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

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

How typical software checks
16-byte authenticator:

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

Fix, eliminating information flow
from secrets to timings:

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

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

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

So mistakes continue to happen.

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

So mistakes continue to happen.

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

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

Do timing attacks really work?

Objection: "Timings are noisy!"

Do timing attacks really work?

Objection: "Timings are noisy!"

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

Do timing attacks really work?

Objection: "Timings are noisy!"

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

Answer #2: Attacker uses statistics to eliminate noise.

Do timing attacks really work?

Objection: "Timings are noisy!"

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

Answer #2: Attacker uses
statistics to eliminate noise.

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

Defenders don't learn

Some of the literature:

1996 Kocher pointed out timing attacks on cryptographic key bits.

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

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

“Guaranteed” countermeasure:
load entire table into cache.

"Guaranteed" countermeasure:
load entire table into cache.

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

“Guaranteed” countermeasure: load entire table into cache.

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

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

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

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

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

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

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

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

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

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

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

How to write constant-time code

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

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

Measure cycles rather than trusting CPU documentation.

Cut off all data flow from secrets to branch conditions.

Cut off all data flow from secrets to array indices.

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

Prefer logic instructions.

Prefer vector instructions.

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

Software optimization

Almost all software is
much slower than it could be.

Software optimization

Almost all software is
much slower than it could be.

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

Software optimization

Almost all software is much slower than it could be.

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

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

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

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

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

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

You measure performance of the implementation. Now what?

A simplified example

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

Reference implementation:

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

Counting cycles:

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

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

Output shows 8012 cycles.

Change 1000 to 500: 4012.

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

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

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

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

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

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

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

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

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

First manual says that ADD takes just 1 cycle.

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`.

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

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

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

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

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

  return result;
}
```

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

  return result;
}
```

2526 cycles. Even better in asm.

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

  return result;
}
```

2526 cycles. Even better in asm.

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

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

  return result;
}
```

2526 cycles. Even better in asm.

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

A real example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The big picture

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

The big picture

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

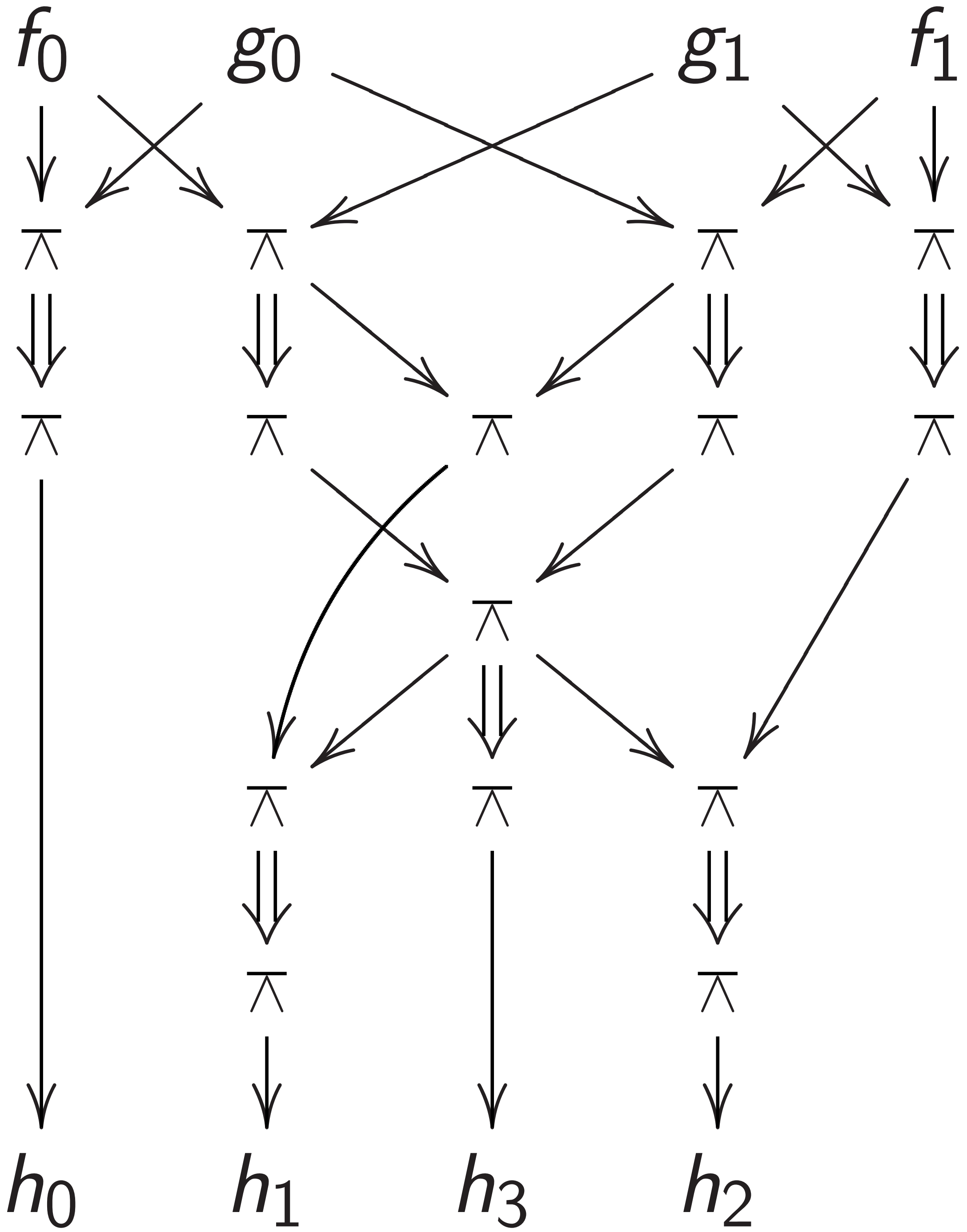
Minor optimization challenges:

- 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

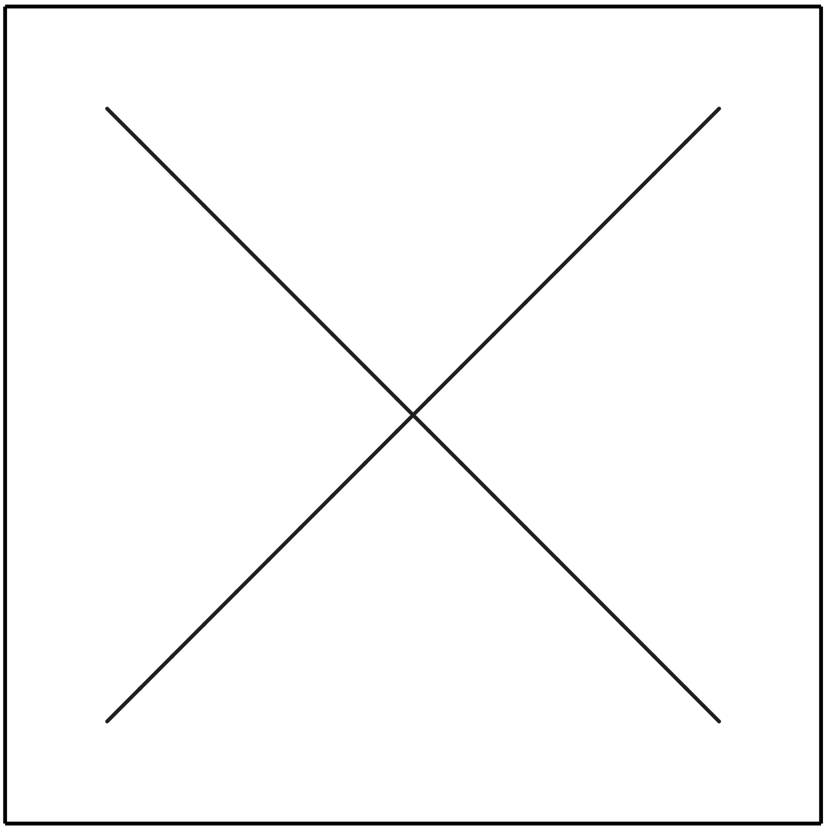


Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

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.

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

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



(Details omitted.)

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

register read

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

register
read

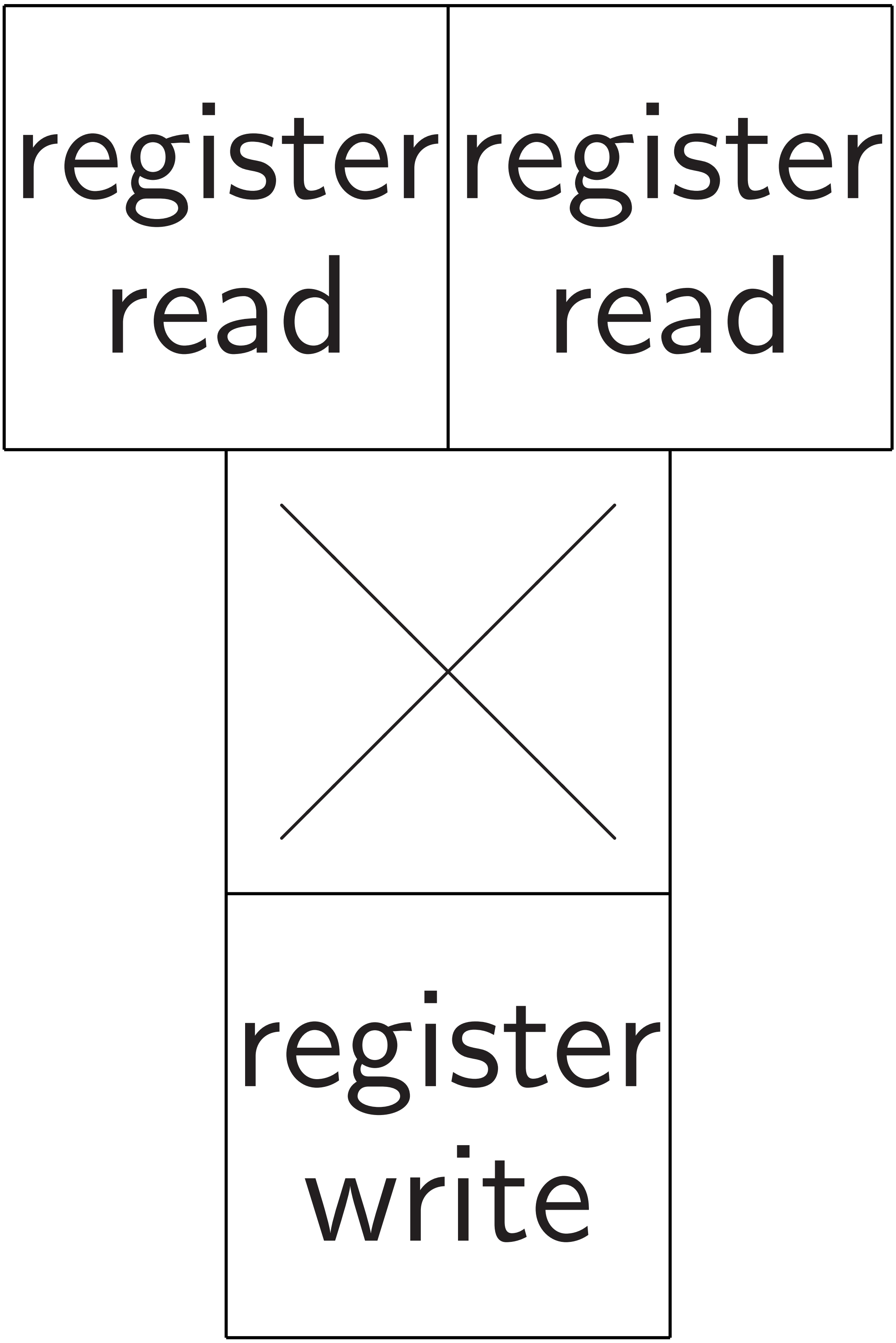
Build circuit for "register write":
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.

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

register
read

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

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

Add more flexibility.

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

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

Add more flexibility.

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

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

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

Add more flexibility.

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

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

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

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

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

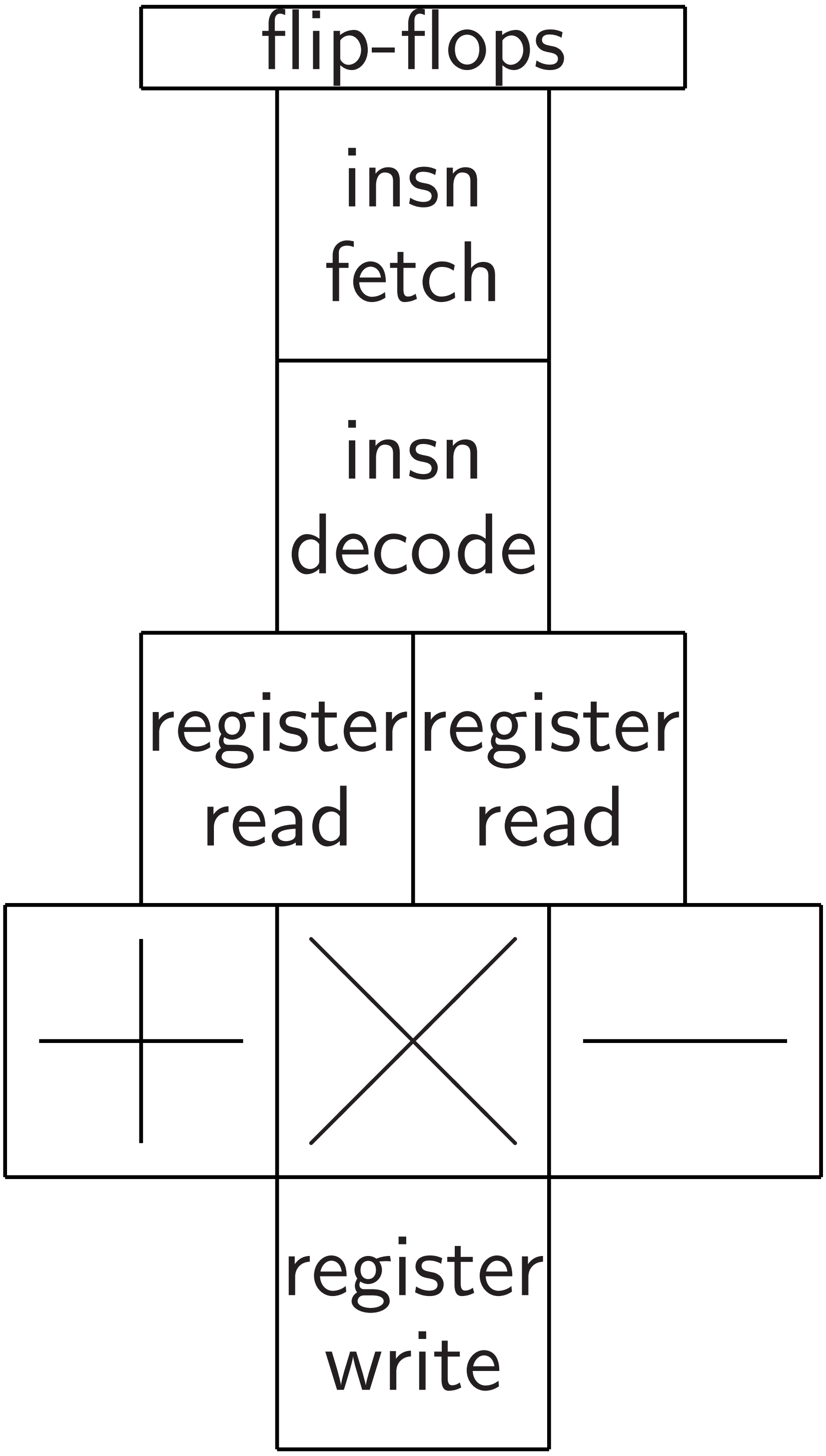
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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

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

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

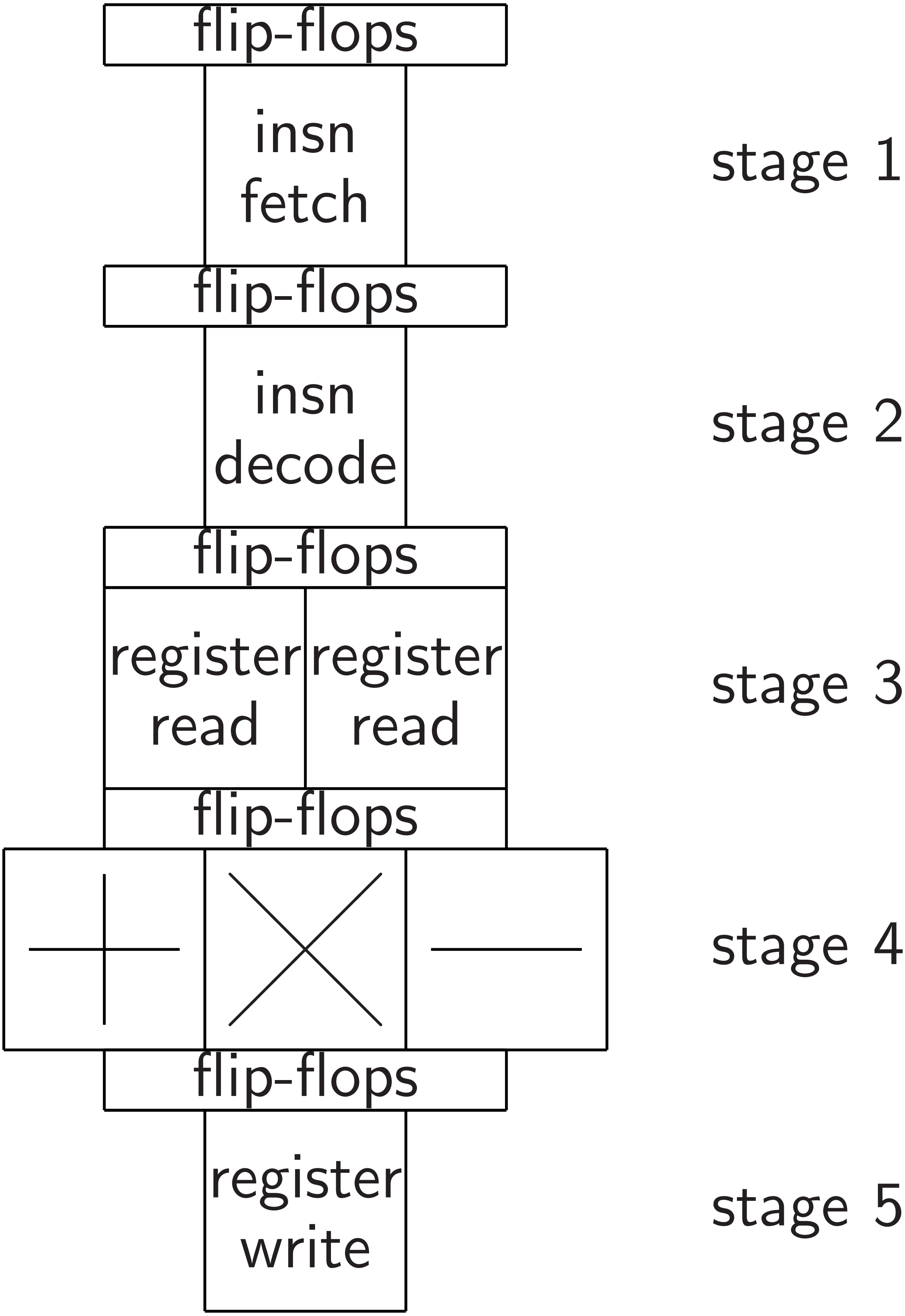
Now have semi-flexible CPU:



Further flexibility is useful:

e.g., rotation instructions.

“Pipelining” allows faster clock:



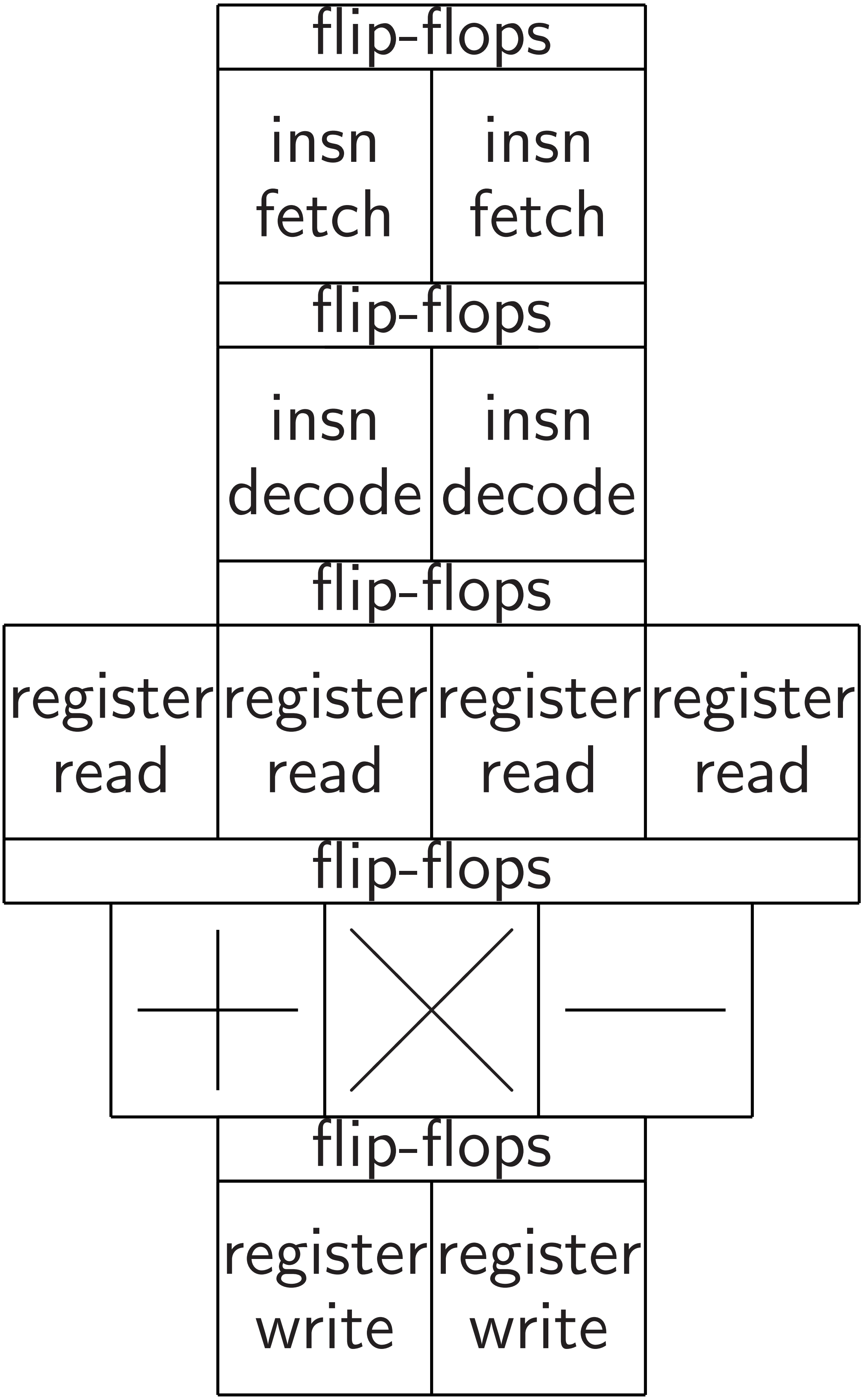
Goal: Stage n handles instruction one tick after stage $n-1$.

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.

“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.

“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.

“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.

Network on chip: the mesh

How expensive is sorting?

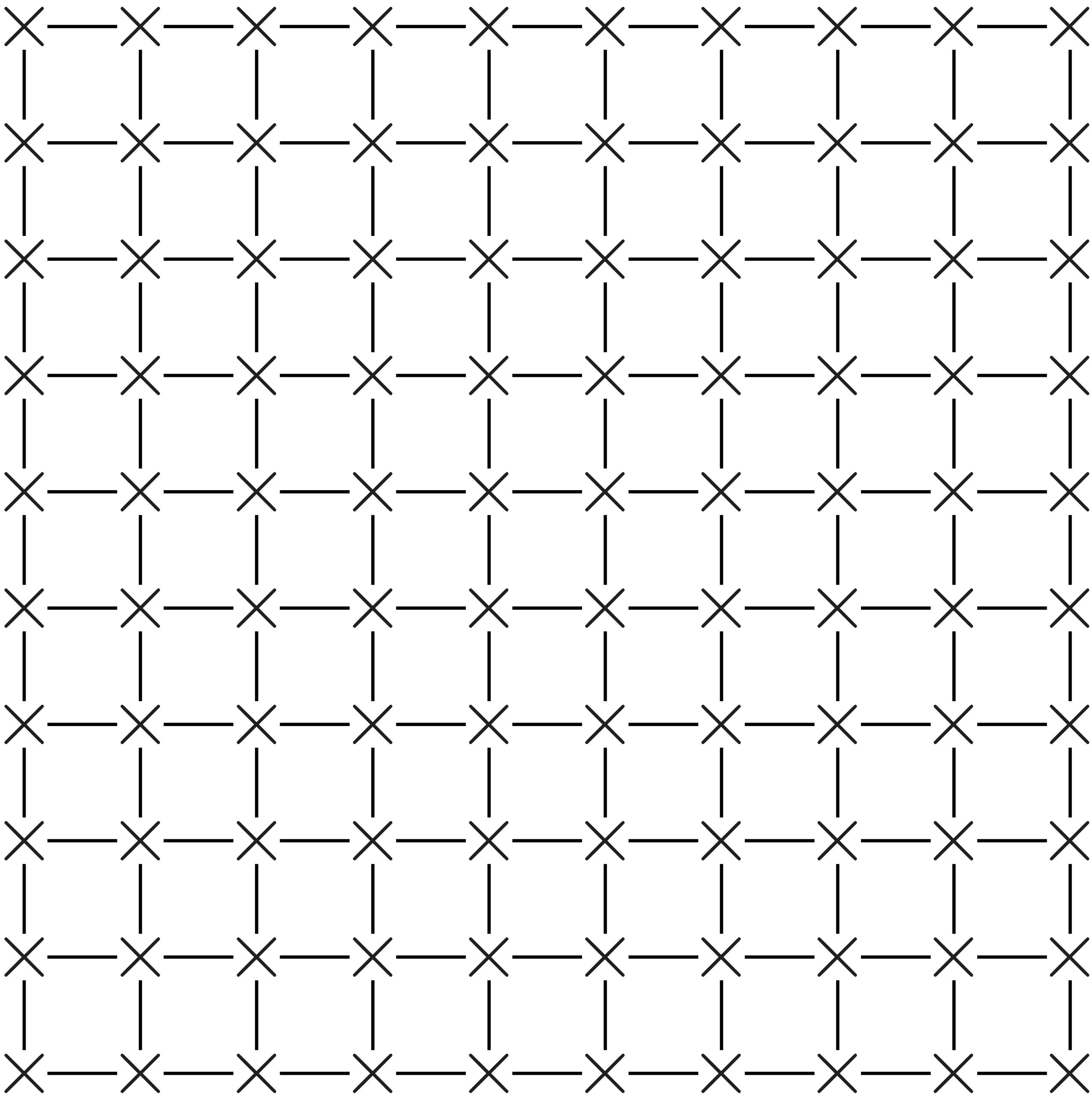
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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:



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 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.

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

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 ←, →:

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,
$\leftarrow$ 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.

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.

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.