The death of optimizing compilers

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

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

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

The oversimplified story

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

The oversimplified story

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

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

The actual story

Wait! It’s not that simple.

Software speed still matters.

Users are often waiting
for their computers.

The actual story

Wait! It’s not that simple.

Software speed still matters.

Users are often waiting
for their computers.

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

The actual story

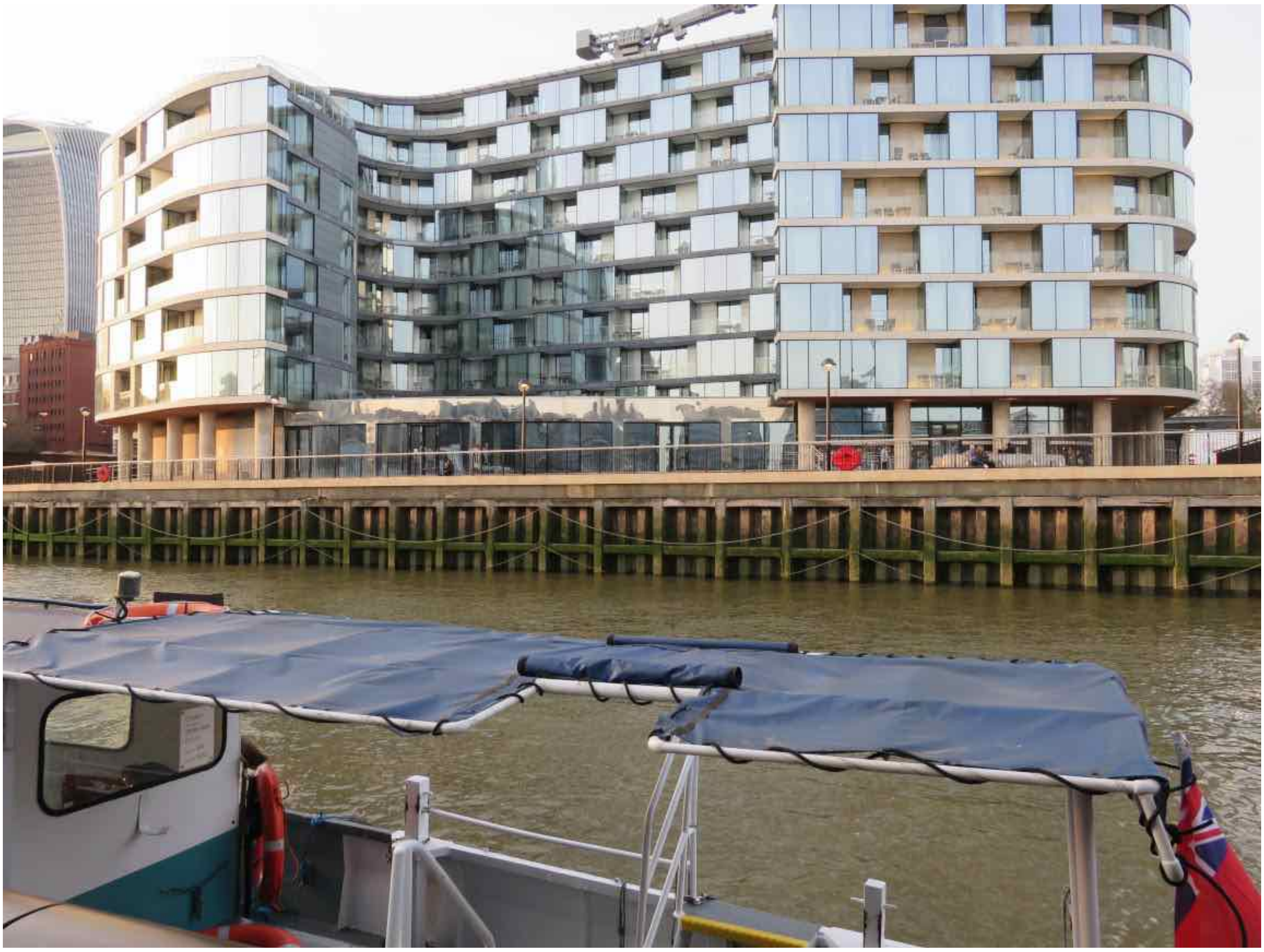
Wait! It's not that simple.

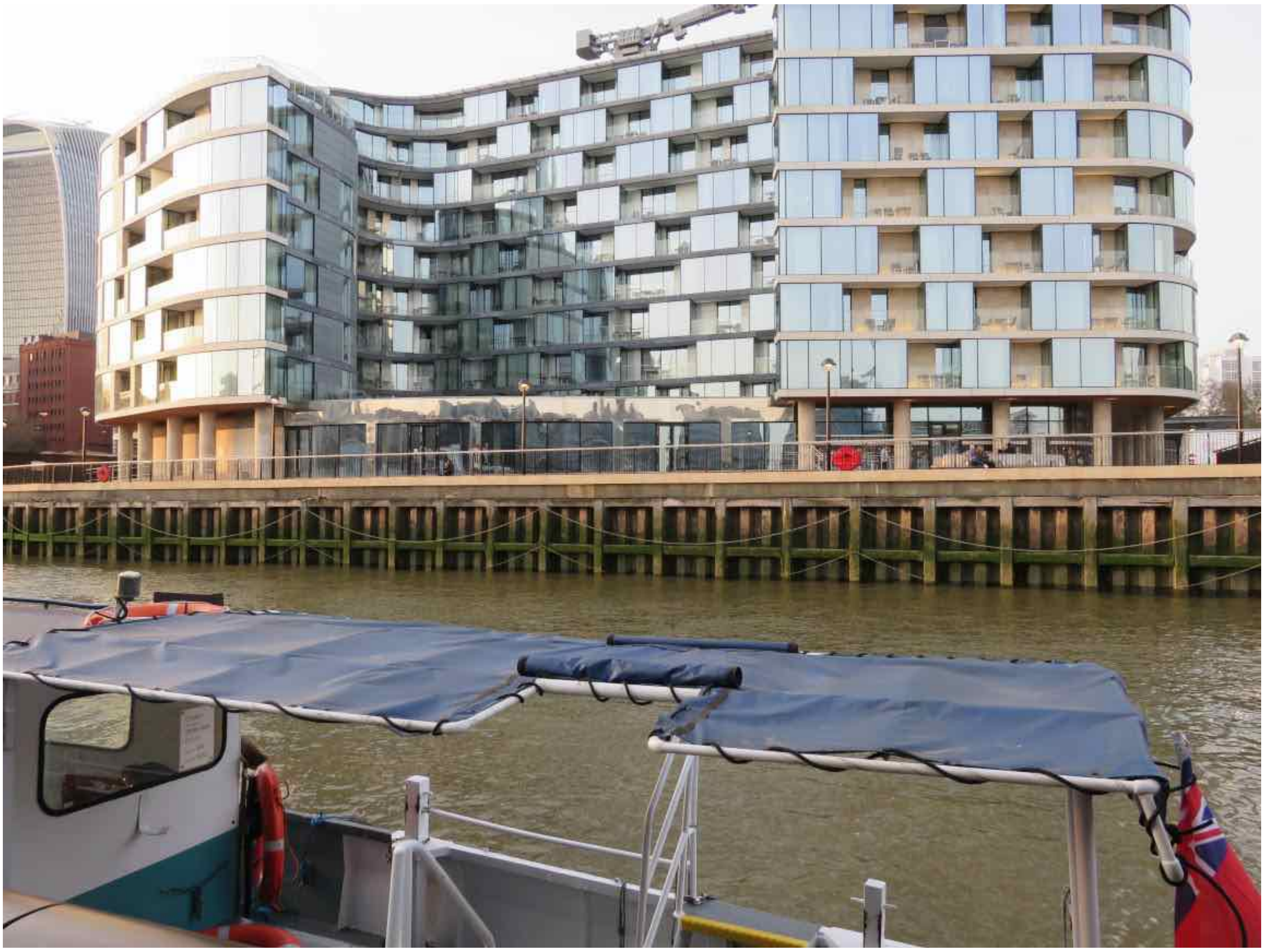
Software speed still matters.

Users are often waiting
for their computers.

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

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





Old CPU displaying a file:

0ms: Start opening file.

400ms: Start displaying contents.

1200ms: Start cleaning up.

1600ms: Finish.

CPUs become faster:

0ms: Start opening file.

350ms: Start displaying contents.

1050ms: Start cleaning up.

1400ms: Finish.

CPUs become faster:

0ms: Start opening file.

300ms: Start displaying contents.

900ms: Start cleaning up.

1200ms: Finish.

CPUs become faster:

0ms: Start opening file.

250ms: Start displaying contents.

800ms: Start cleaning up.

1000ms: Finish.

CPUs become faster:

0ms: Start opening file.

200ms: Start displaying contents.

600ms: Start cleaning up.

800ms: Finish.

User displays bigger file:

0ms: Start opening file.

200ms: Start displaying contents.

1000ms: Start cleaning up.

1200ms: Finish.

CPUs become faster:

0ms: Start opening file.

175ms: Start displaying contents.

875ms: Start cleaning up.

1050ms: Finish.

CPUs become faster:

0ms: Start opening file.

150ms: Start displaying contents.

750ms: Start cleaning up.

900ms: Finish.

CPUs become faster:

0ms: Start opening file.

125ms: Start displaying contents.

625ms: Start cleaning up.

750ms: Finish.

CPUs become faster:

0ms: Start opening file.

100ms: Start displaying contents.

500ms: Start cleaning up.

600ms: Finish.

User displays bigger file:

0ms: Start opening file.

100ms: Start displaying contents.

900ms: Start cleaning up.

1000ms: Finish.

User displays bigger file:

100ms: Start displaying contents.

1000ms: Finish.

CPUs become faster:

87.5ms: Start displaying contents.

875ms: Finish.

CPUs become faster:

75.0ms: Start displaying contents.

750ms: Finish.

CPUs become faster:

62.5ms: Start displaying contents.

625ms: Finish.

CPUs become faster:

50ms: Start displaying contents.

500ms: Finish.

User displays bigger file:

50ms: Start displaying contents.

900ms: Finish.

Cheaper computation $\Rightarrow$
users process more data.

Cheaper computation $\Rightarrow$
users process more data.

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

Cheaper computation $\Rightarrow$
users process more data.

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

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

Cheaper computation $\Rightarrow$
users process more data.

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

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

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

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

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

— This view is obsolete.

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

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

Speed matters: an example

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

What about the servers?

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

Typical excerpt from inner loop of server code:

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

Mobile code similarly has heavy vectorization + asm.

Hand-tuned? In 2015? Seriously?

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

Hand-tuned? In 2015? Seriously?

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

— [citation needed]

Hand-tuned? In 2015? Seriously?

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

— The experts disagree, and hold the speed records.

Hand-tuned? In 2015? Seriously?

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

— The experts disagree, and hold the speed records.

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

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

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

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

The algorithm designer’s job

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

CS 101 view: “Time”.

The algorithm designer's job

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

CS 101 view: "Time".

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

The algorithm designer's job

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

CS 101 view: "Time".

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

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

The algorithm designer's job

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

CS 101 view: "Time".

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm design is hard.

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

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

Algorithm designer vs. compiler

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

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

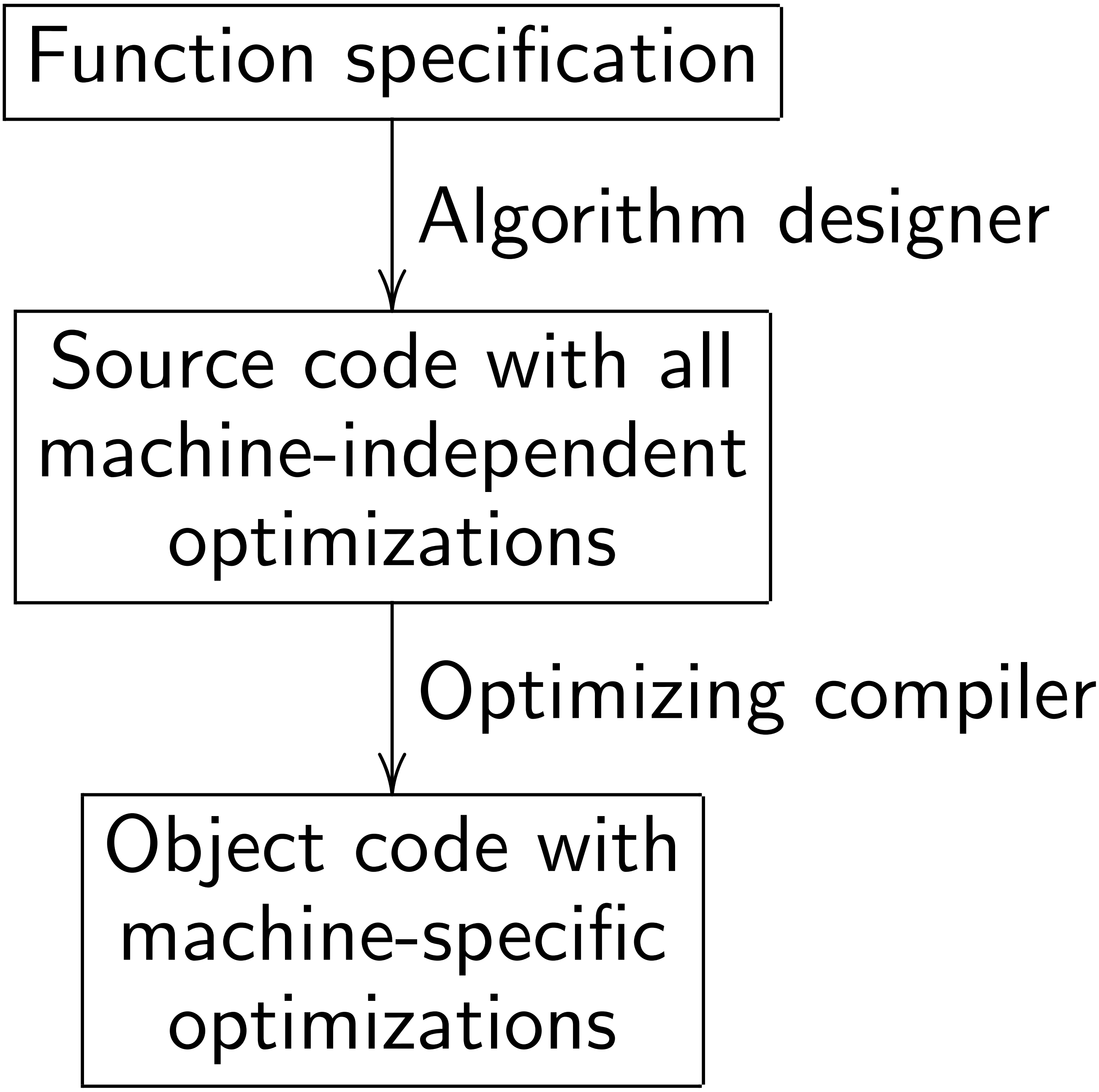
Algorithm designer vs. compiler

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

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

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

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



Function specification

Algorithm designer

Source code with all machine-independent optimizations

Optimizing compiler

Object code with machine-specific optimizations

Output of optimizing compiler is algorithm for target machine.

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

Output of optimizing compiler
is algorithm for target machine.

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

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

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

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

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

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

How the code base is evolving:

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

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

How the code base is evolving:

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

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

How the code base is evolving:

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

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

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

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

Why this is happening

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

Why this is happening

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

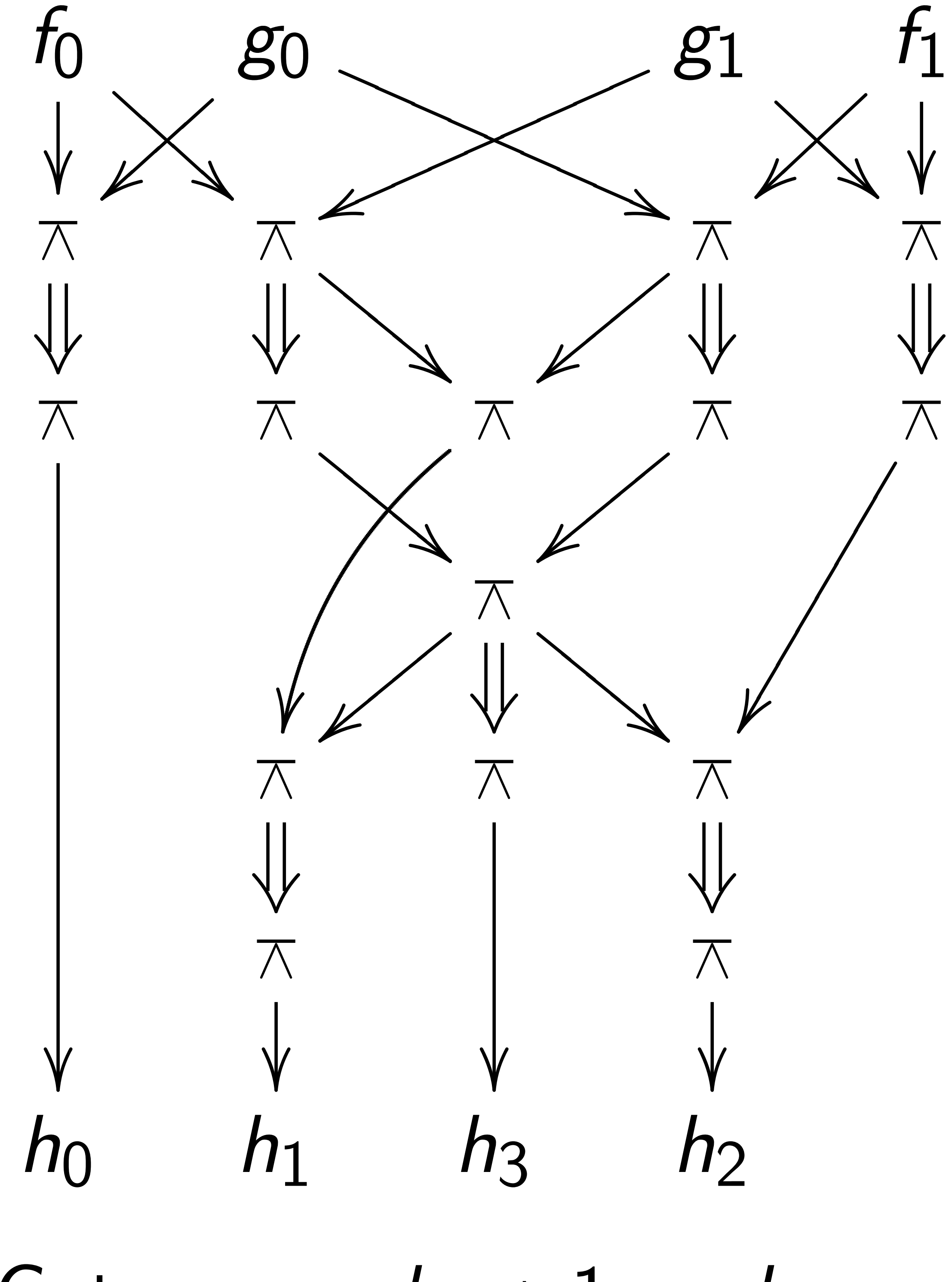
Minor optimization challenges:

- Pipelining.
- Superscalar processing.

Major optimization challenges:

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

CPU design in a nutshell

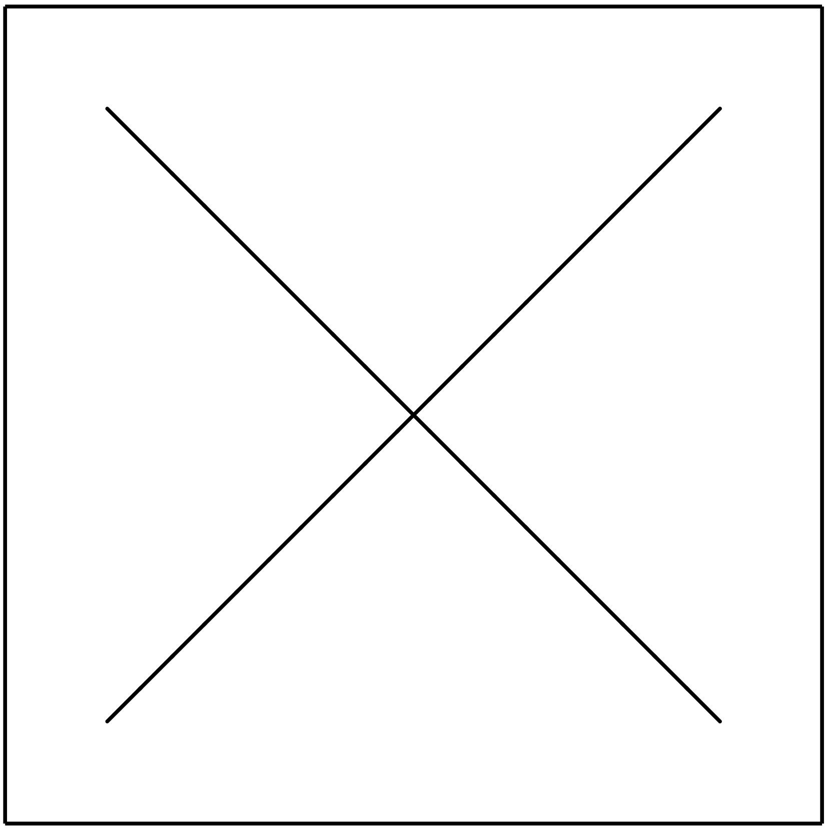


Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

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

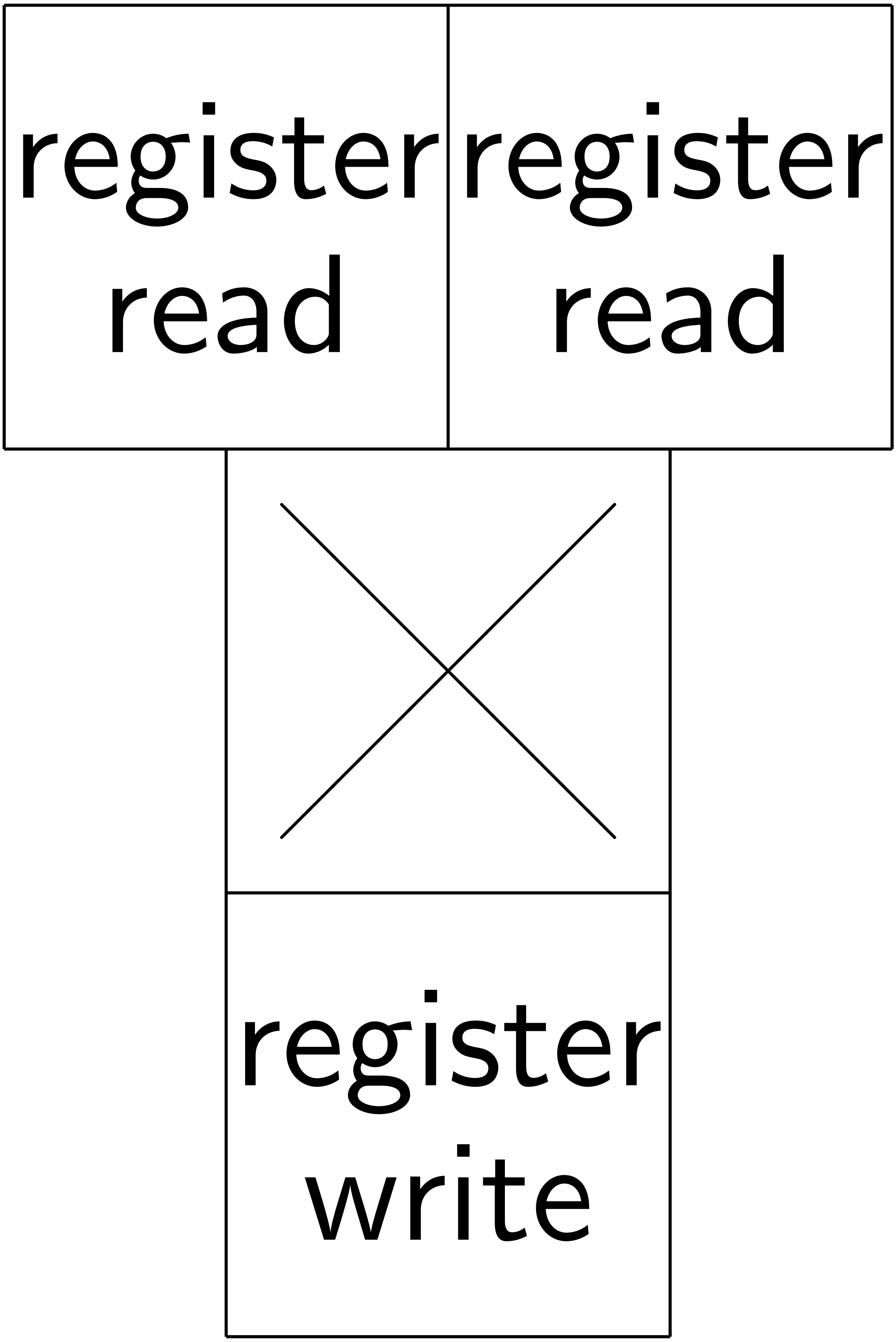
Build circuit for "register write":
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.

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 $r'_i = r_j r_k$:



register read

register read

register write

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

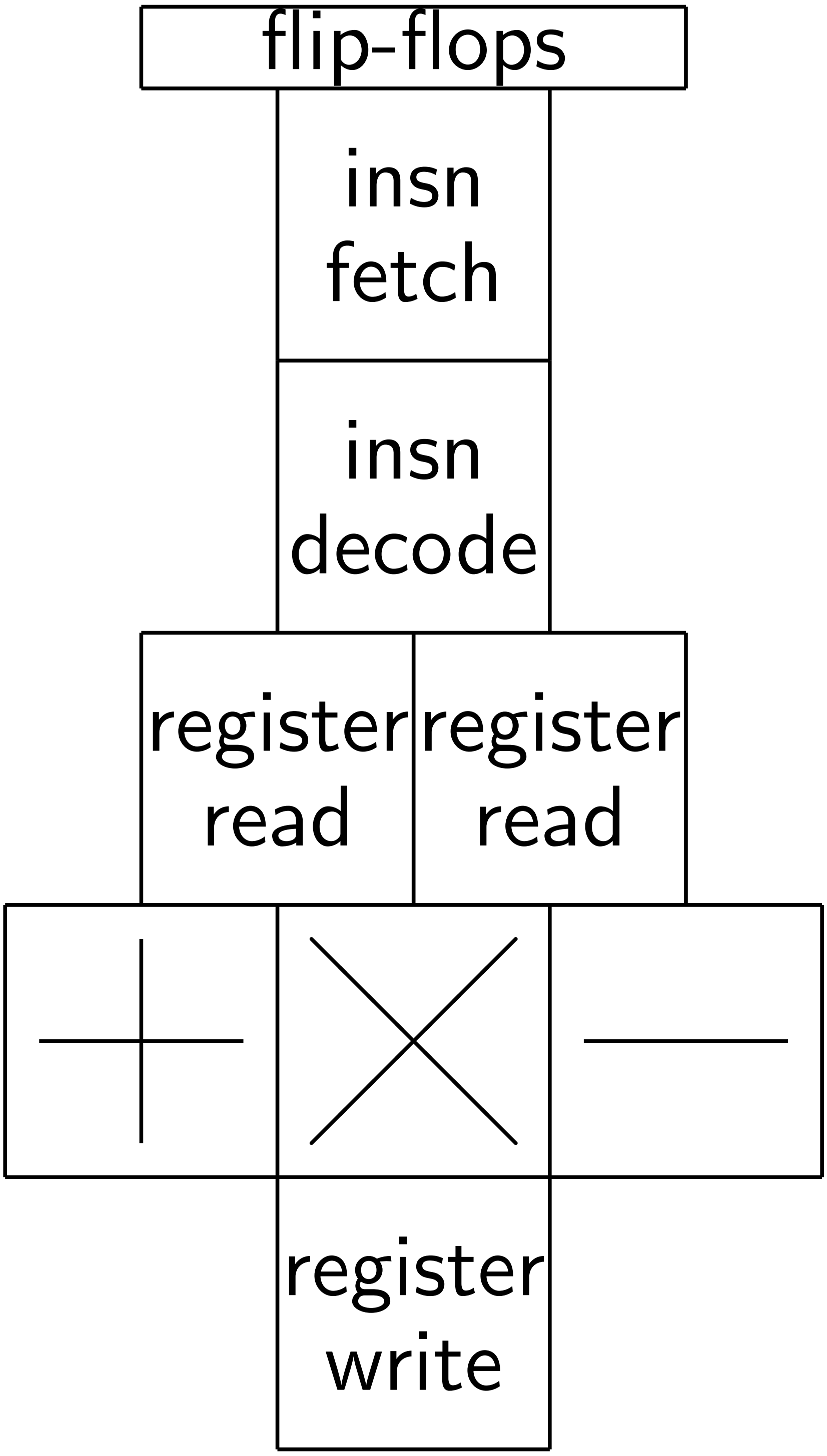
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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

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

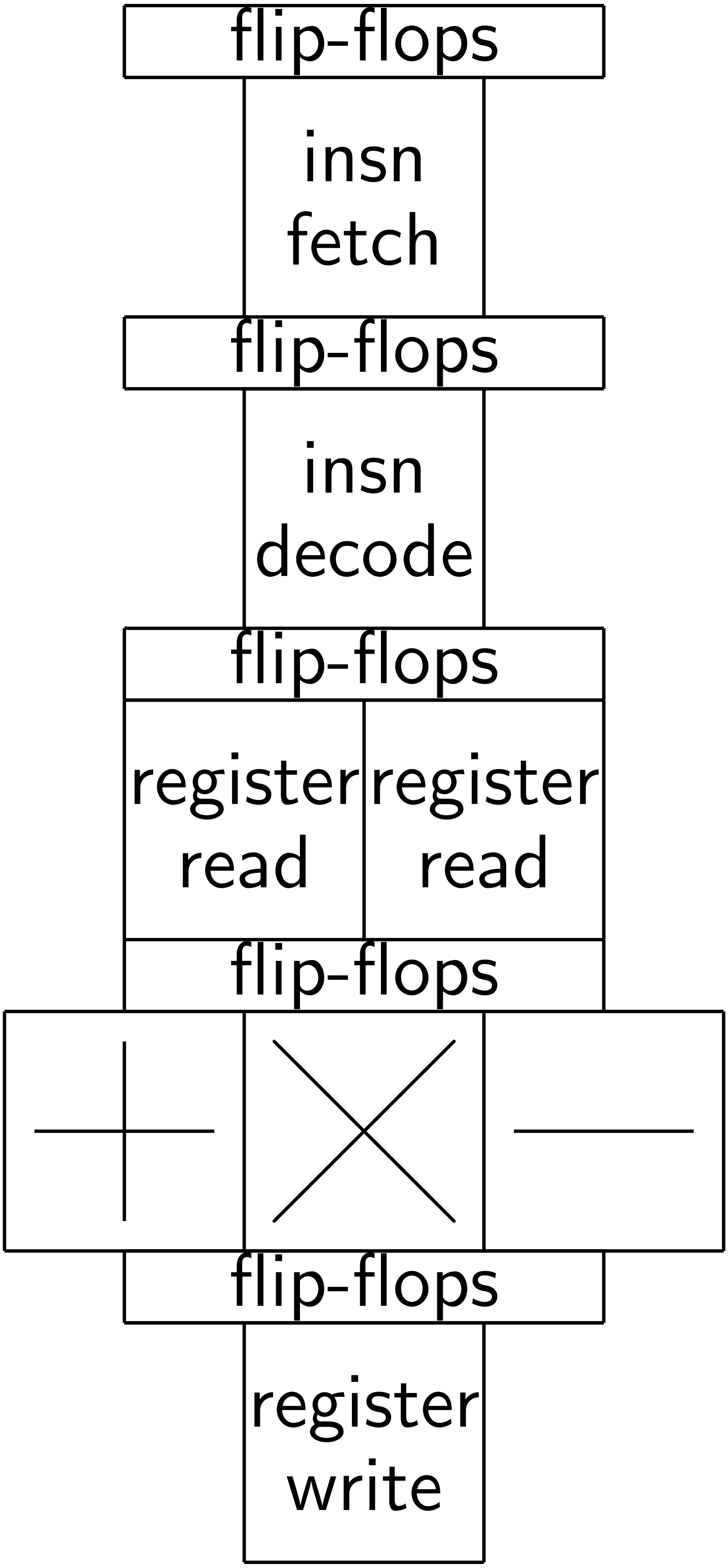
Clock needs to be slow enough
for electricity to percolate
all the way through the circuit,
from flip-flops to flip-flops.

Now have semi-flexible CPU:



Further flexibility is useful but orthogonal to this talk.

“Pipelining” allows faster clock:



stage 1

stage 2

stage 3

stage 4

stage 5

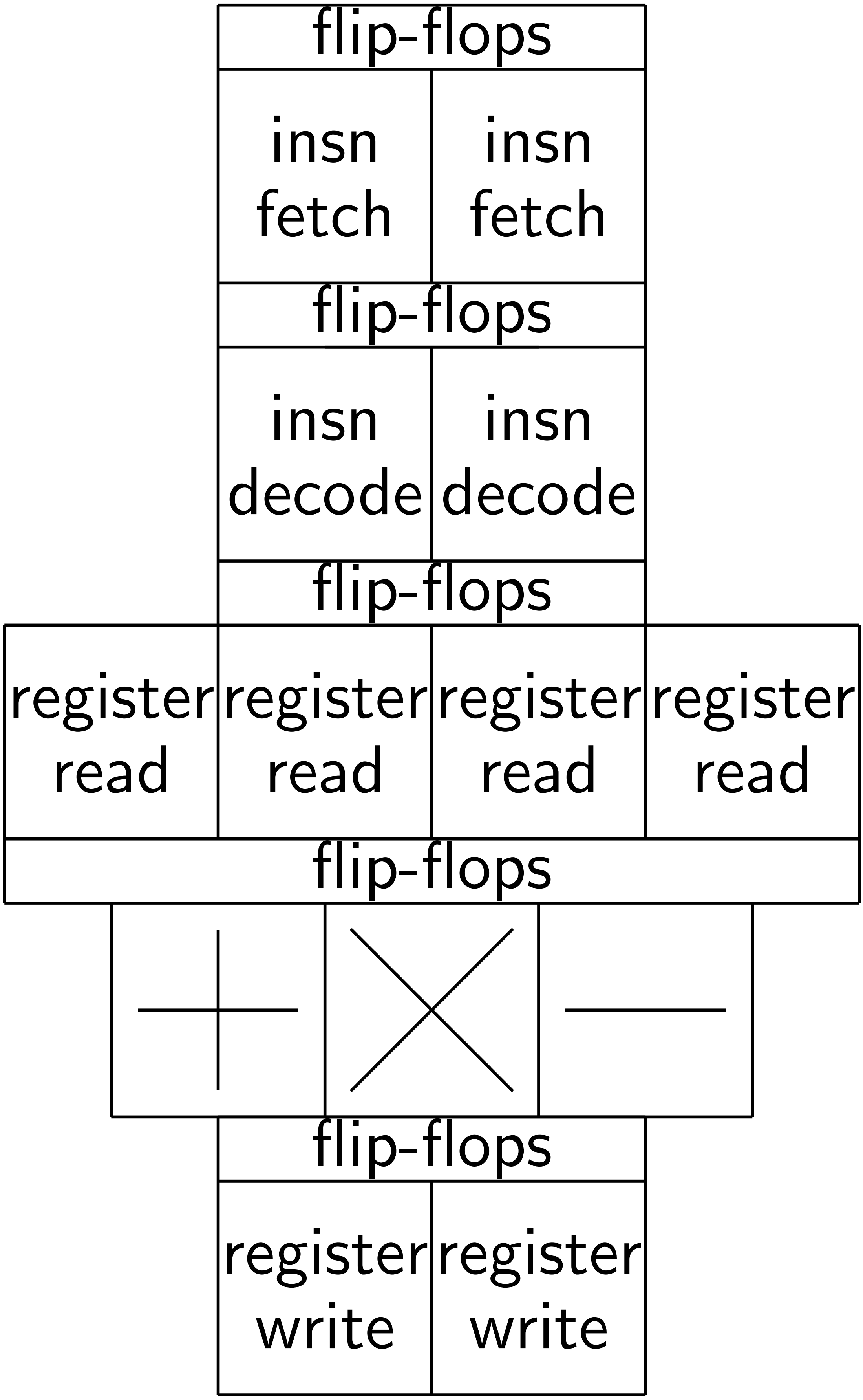
Goal: Stage n handles instruction one tick after stage $n-1$.

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.

“Superscalar” processing:



flip-flops

insn fetch | insn fetch

flip-flops

insn decode | insn decode

flip-flops

register read | register read | register read | register read

flip-flops

+ | × | −

flip-flops

register write | register write

“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 $\left\{1, 2, \ldots, n^2\right\}$,
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?

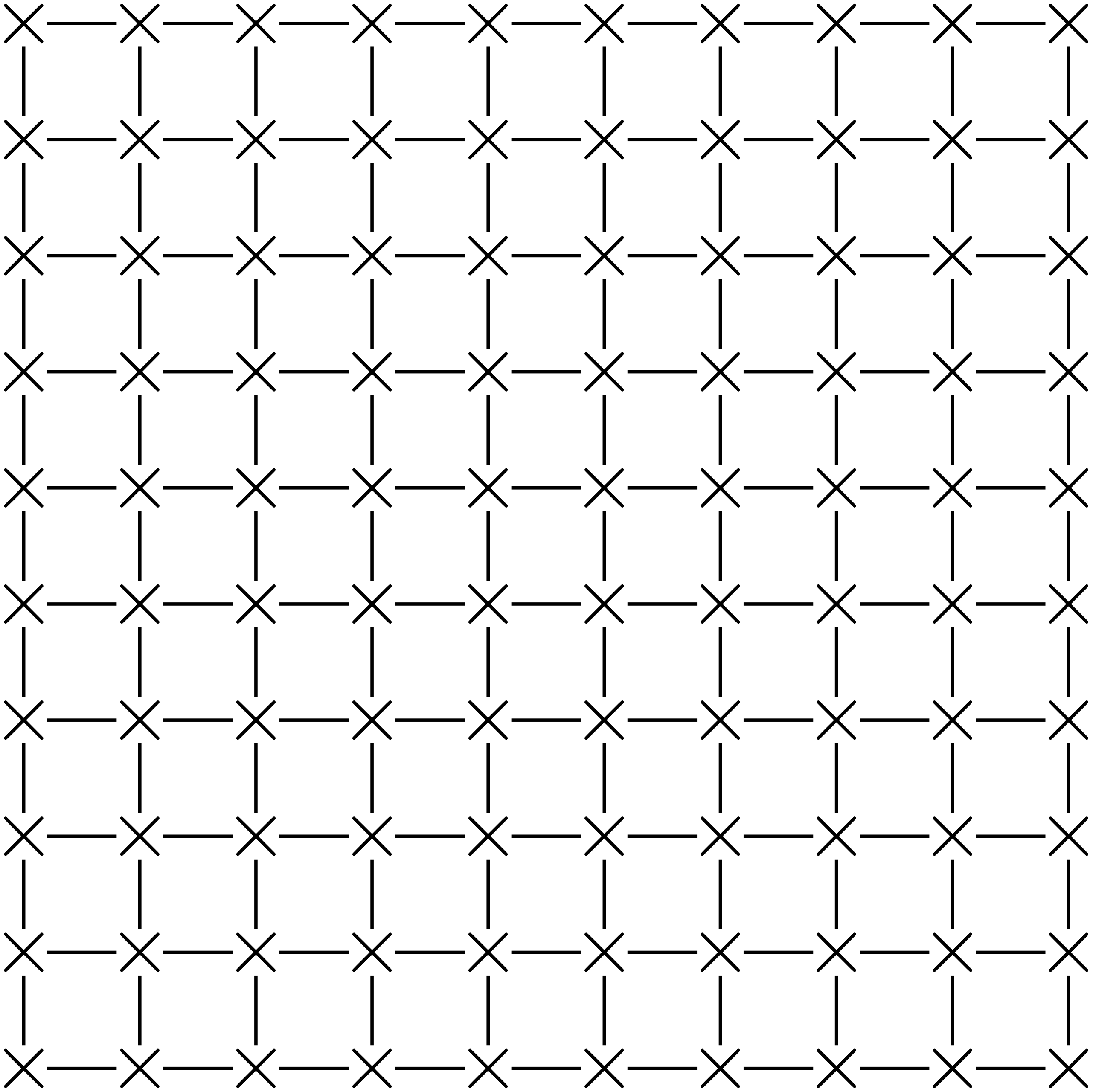
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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 →, bottom ←:

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

Sort each column
in parallel:

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

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,
← or → 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 into high levels of domain-specific algorithm design: e.g., for "NFS" factorization, replace "sieving" with "ECM".

The future of compilers

At this point many readers will say, “But he should only write P, and an optimizing compiler will produce Q.” To this I say, “No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable.” I have another alternative to propose, a new class of software which will be far better. . . .

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN *programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.*

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes . . . when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the source language. Of course! . . .

The time is clearly ripe for program-manipulation systems . . . The programmer using such a system will write his beautifully-structured, but possibly inefficient, program P; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one. . . . As I say, this idea certainly isn't my own; it is so exciting I hope that everyone *soon becomes aware of its possibilities.*