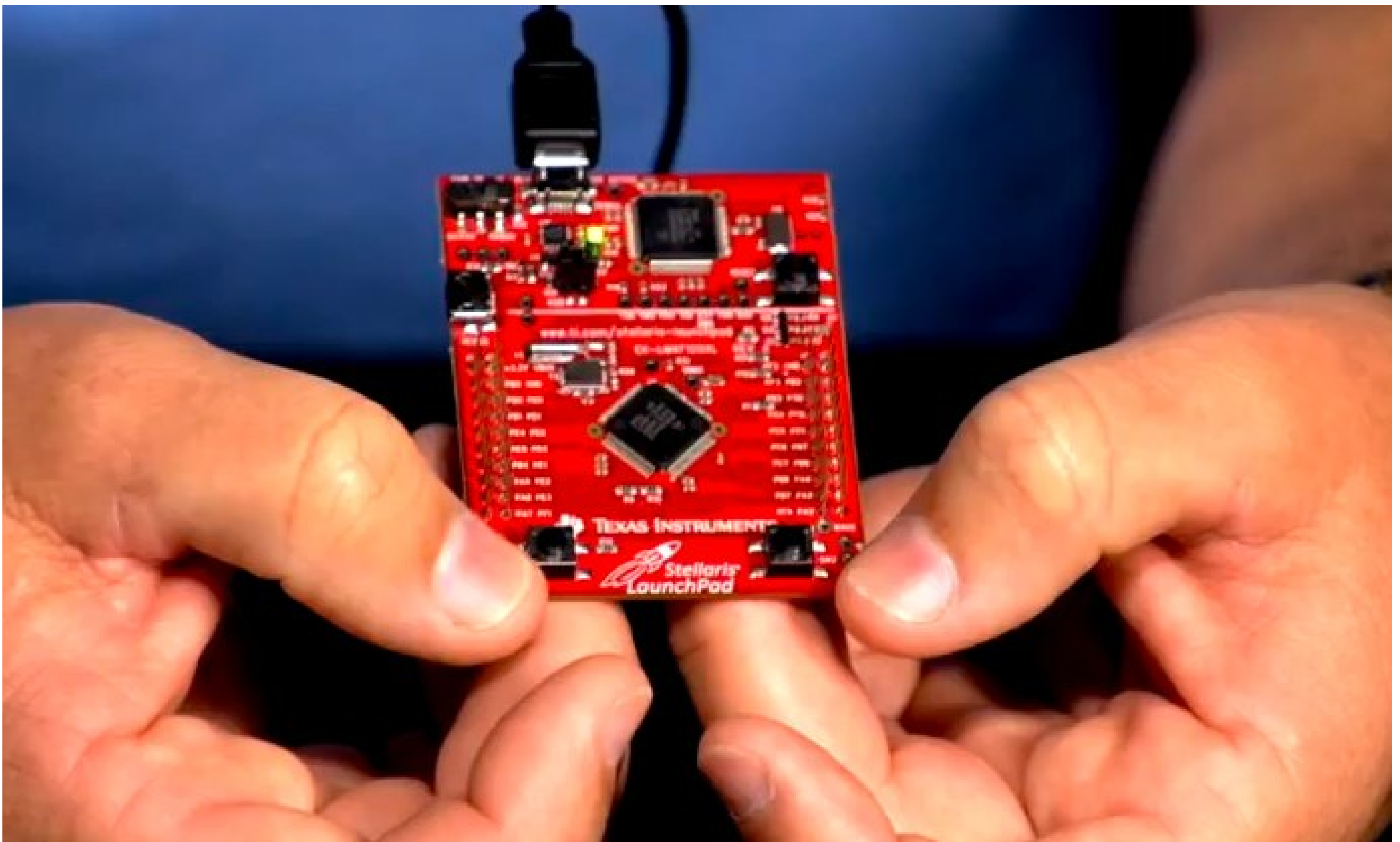
CPU-specific optimization

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;
}
```

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

"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.

"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.

"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.

"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;
}
```

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 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 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;
}
```

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;
}
```

4016 cycles. Are we done now?

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?

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.

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`.

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)
```

```
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
```

```
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 !=
```

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

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.

The big picture

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

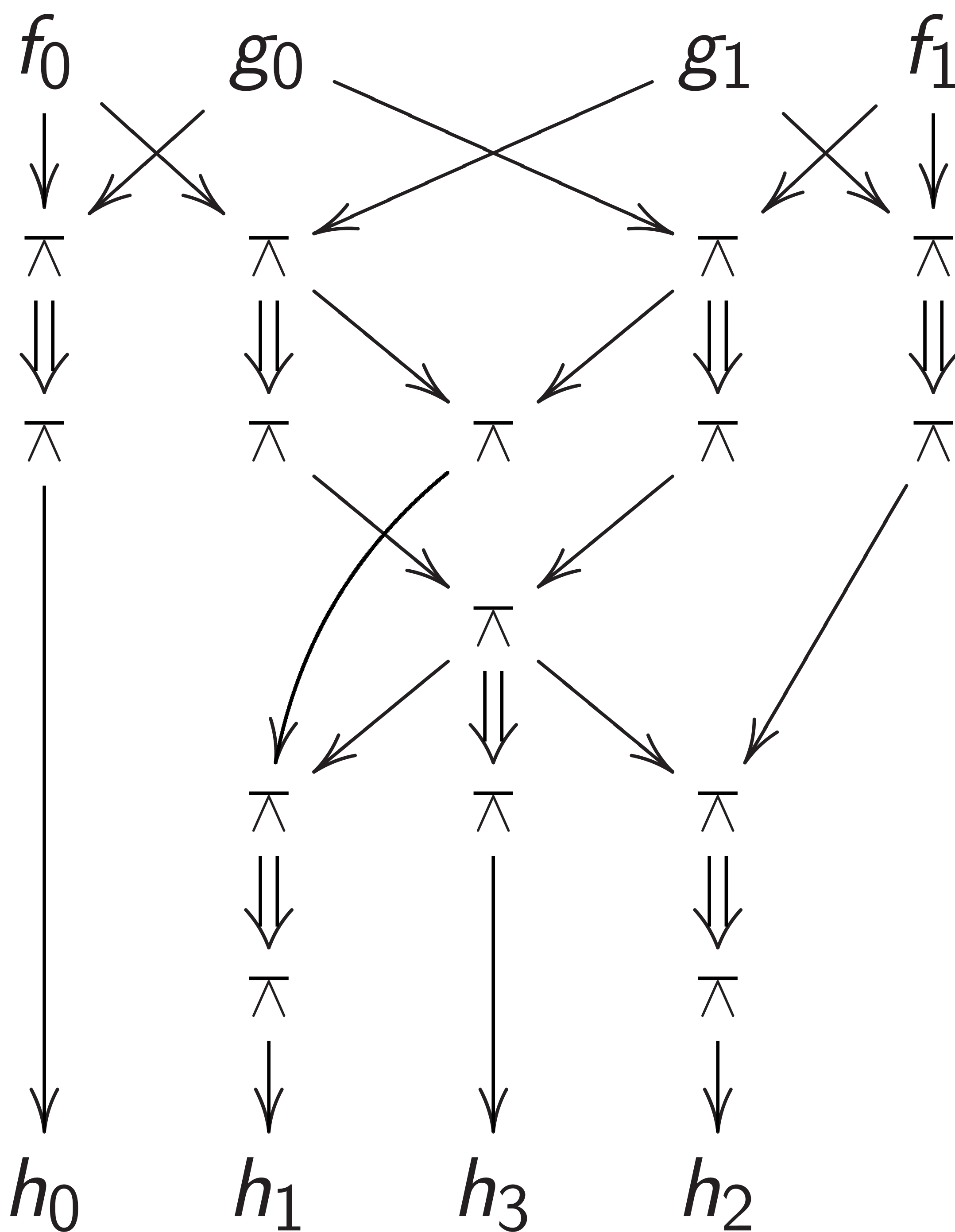
Minor optimization challenges:

- 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

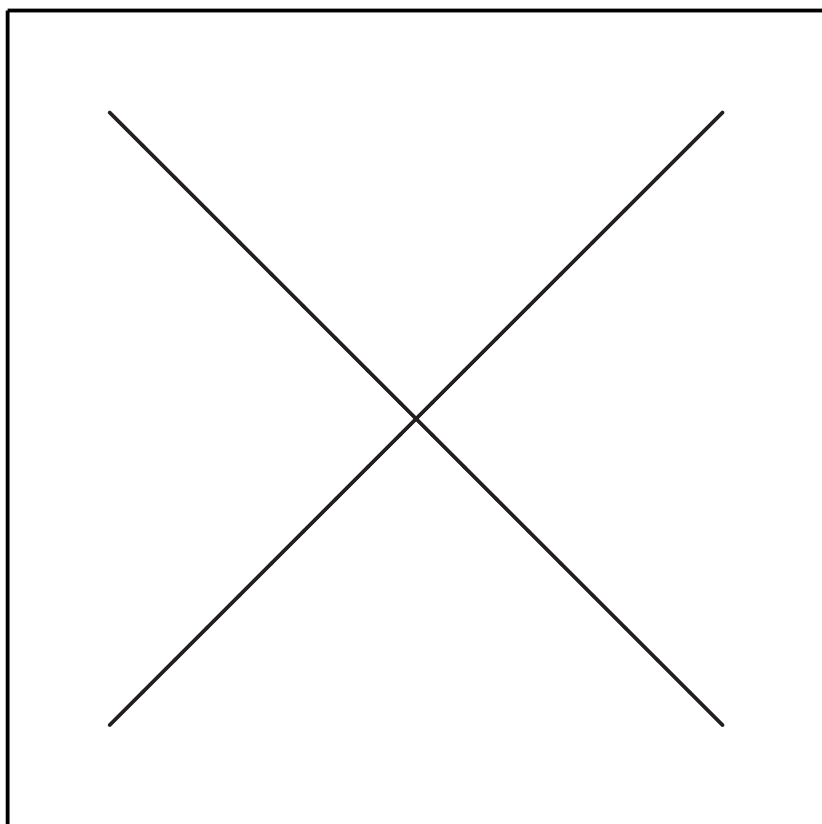


Gates $\barwedge : a, b \mapsto 1 - ab$ computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1, g_0 + 2g_1$.

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

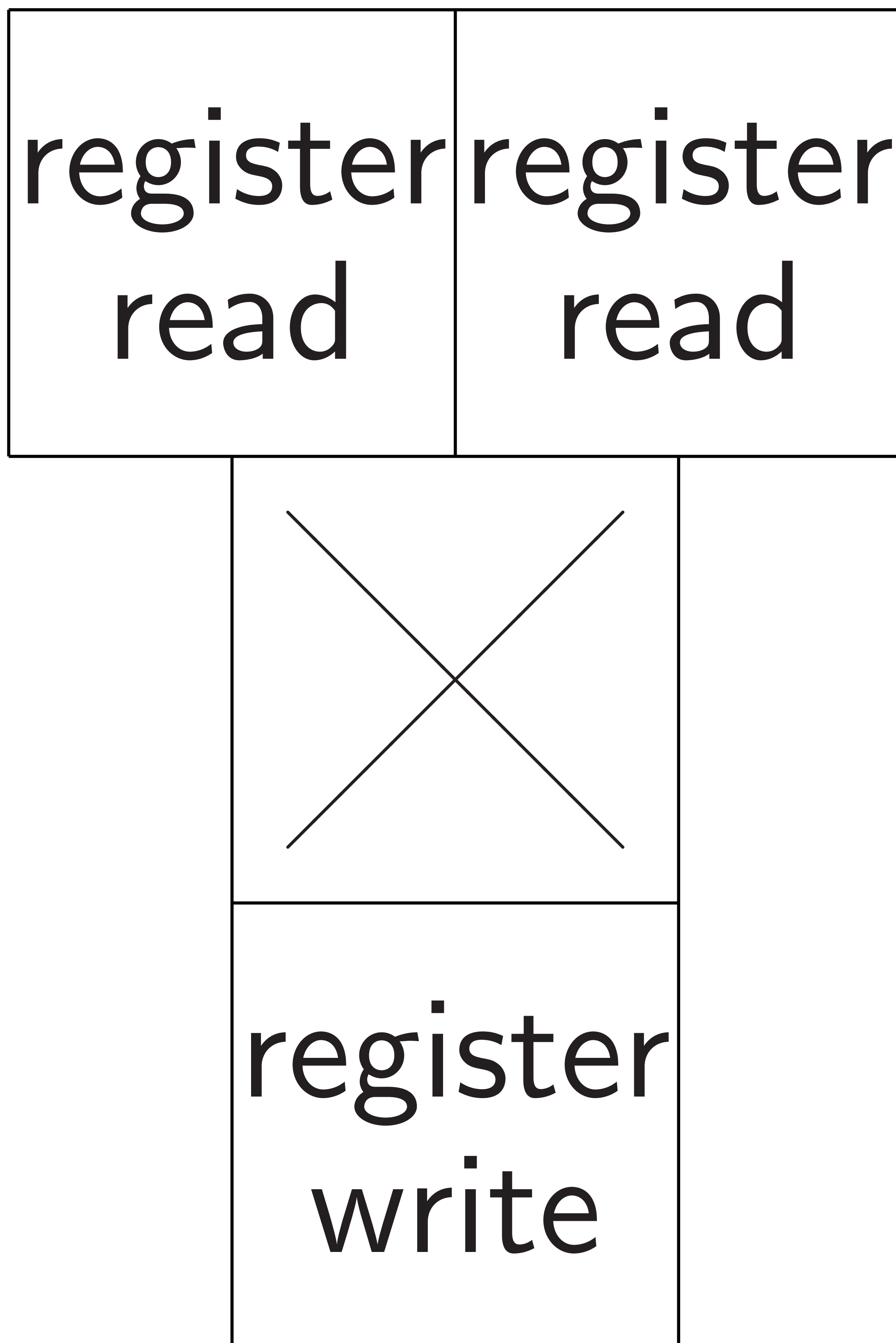
Build circuit for “register write”:
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
where $r'_j = r_j$ except $r'_i = s$.

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.

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

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

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

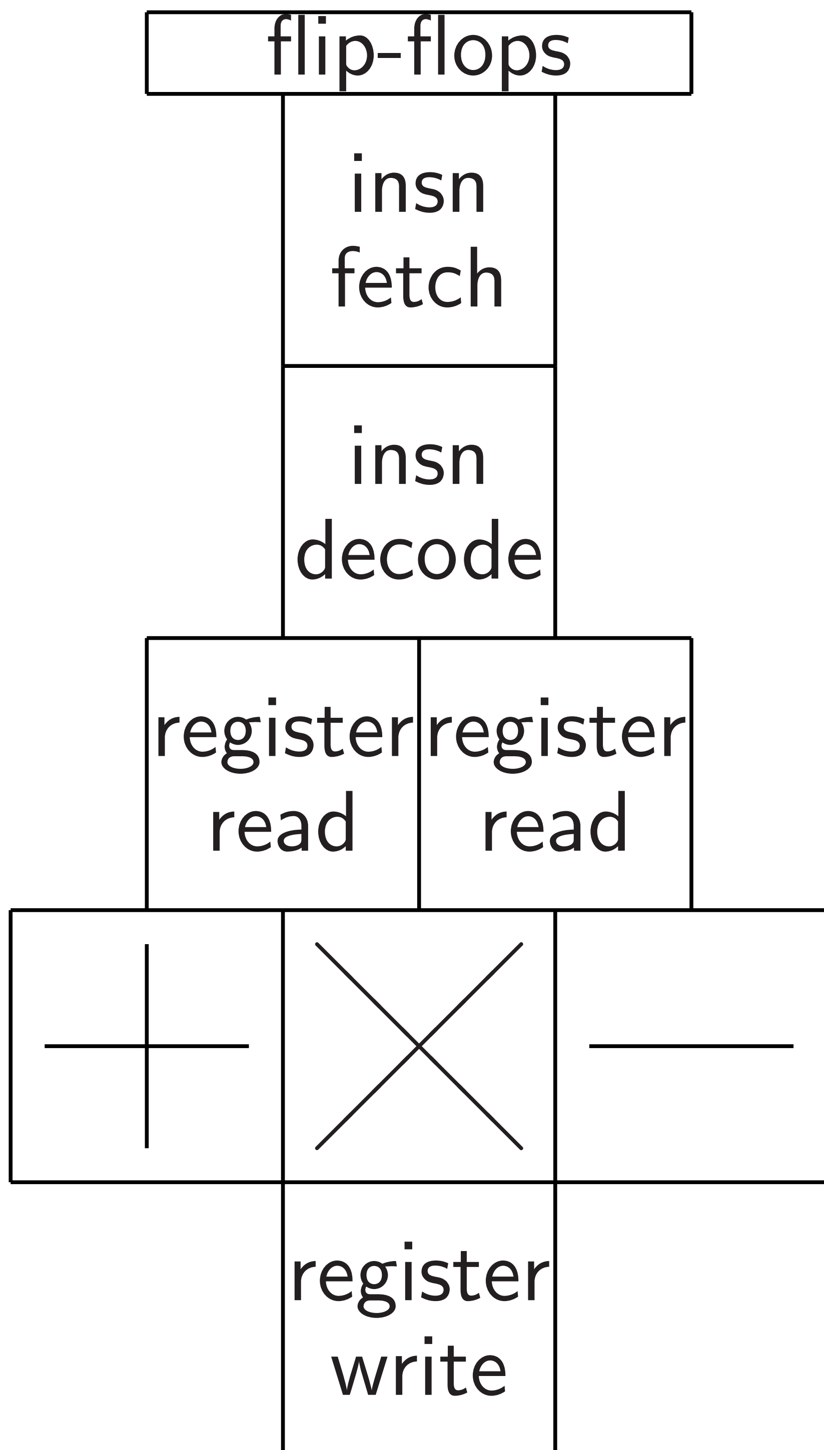
Hook $(p, r_0, \ldots, r_{15})$
flip-flops into circuit inputs.

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

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

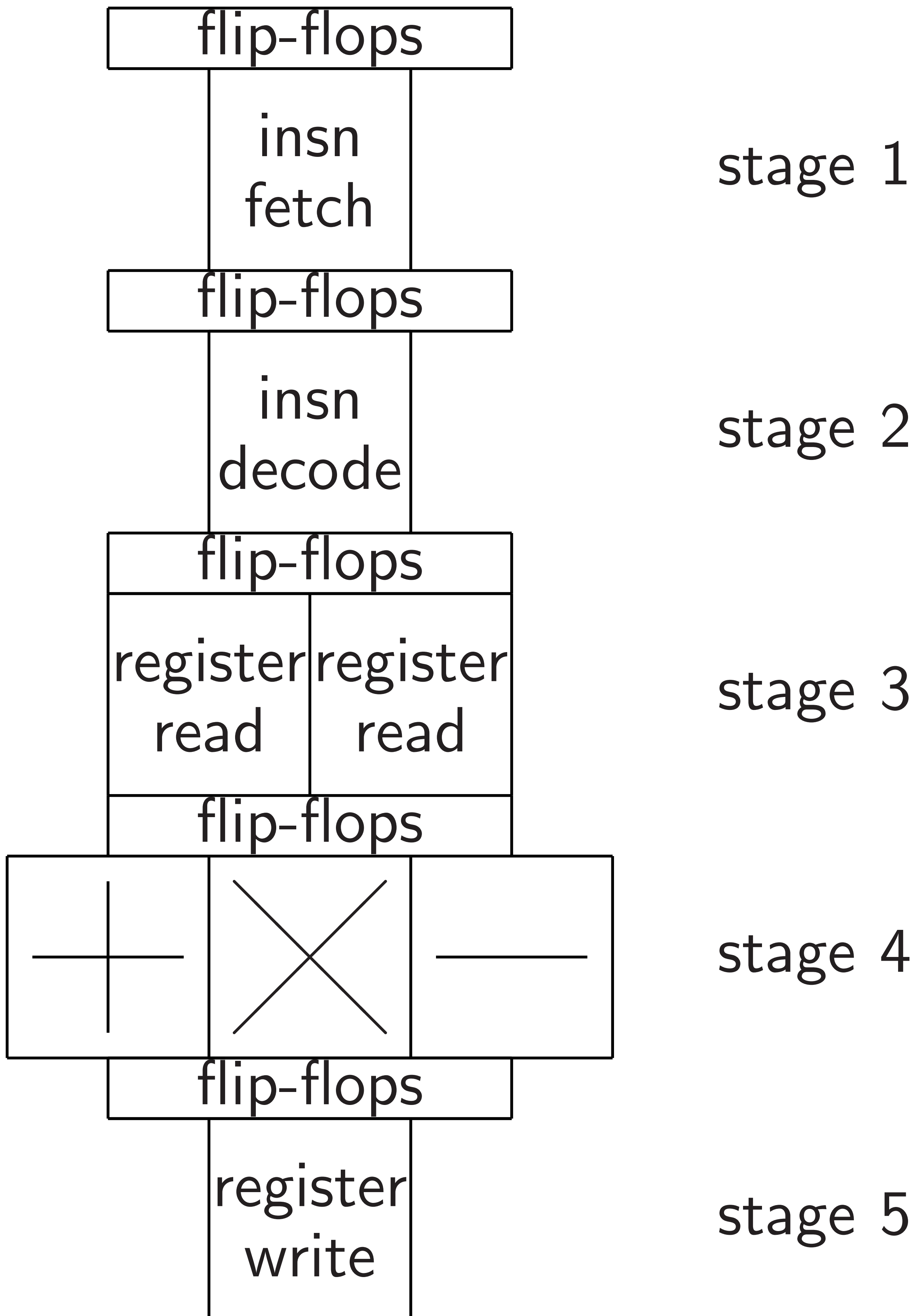
Clock needs to be slow enough
for electricity to percolate
all the way through the circuit,
from flip-flops to flip-flops.

Now have semi-flexible CPU:



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

“Pipelining” allows faster clock:



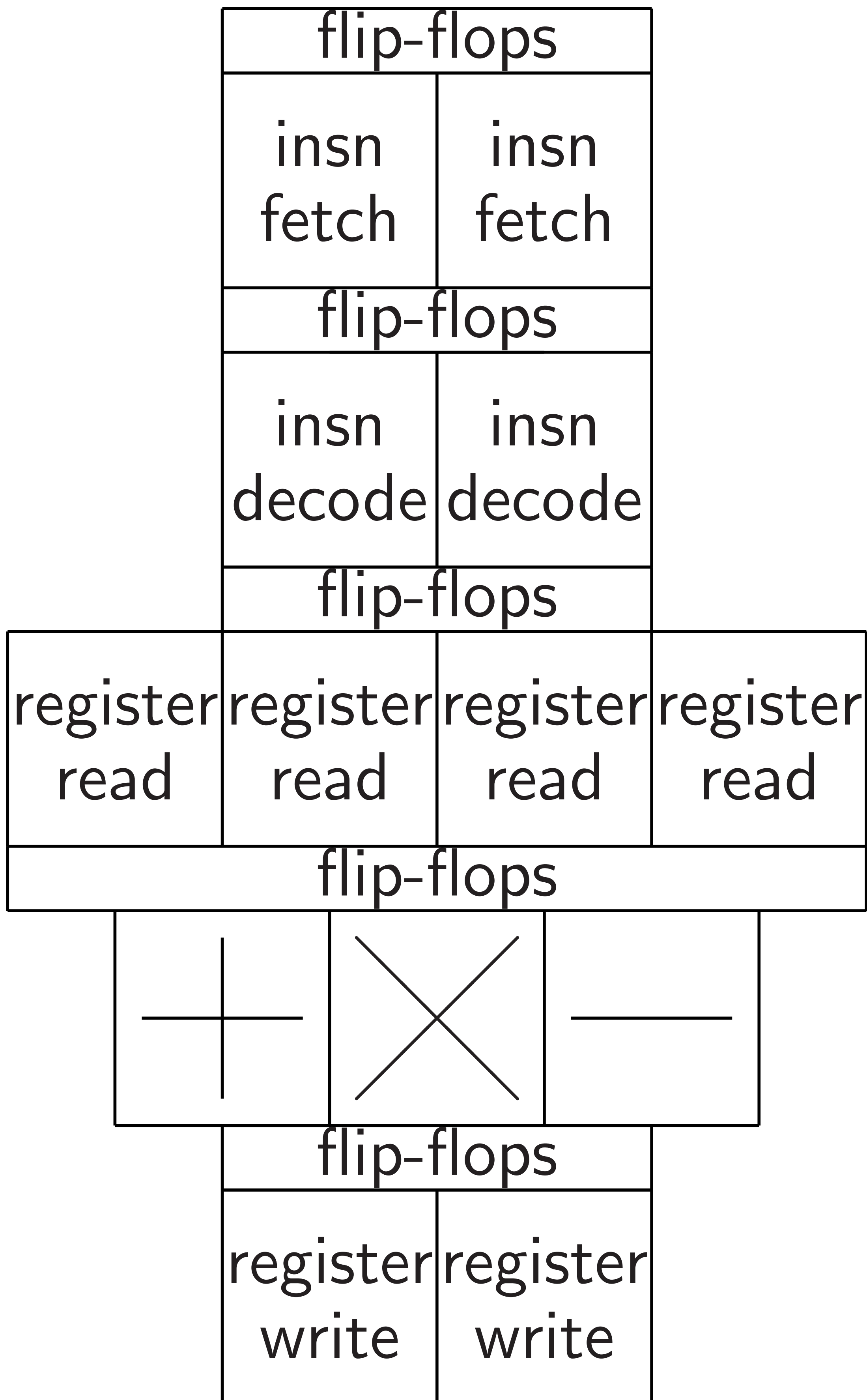
Goal: Stage n handles instruction
one tick after stage $n - 1$.

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?

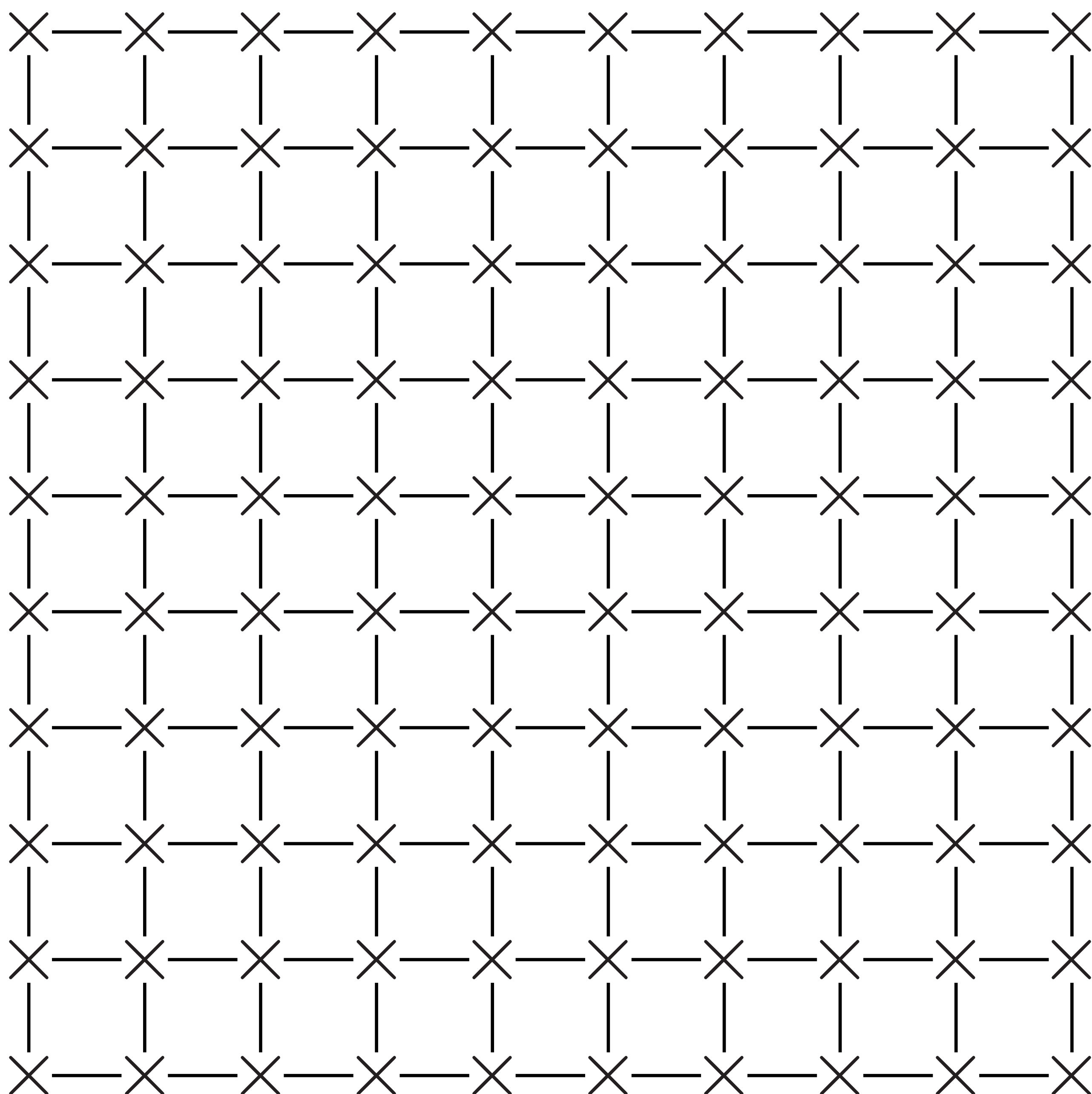
Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

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.