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:

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

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

Counting cycles:

```c
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
", result, aftersum - beforesum);
```
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:

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

Counting cycles:

```c
static volatile unsigned int *const DWT_CYCCNT = (void *) 0xE0001004;
...
int beforesum = *
int result = sum(x);
int aftersum = *
UARTprintf("sum %d %d
", result,aftersum-beforesum);
```

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:

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

Counting cycles:

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

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

Reference implementation:

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

Counting cycles:

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

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

Counting cycles:

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

Reference implementation:

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

Counting cycles:

```c
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?
Example of a function that we want to optimize: adding 1000 integers mod 2.

Reference implementation:

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

Counting cycles:

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

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

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

“Okay, 8 cycles per addition. Um, are microcontrollers really this slow at addition?”
Counting cycles:

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

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

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

“Okay, 8 cycles per addition. Um, are microcontrollers really this slow at addition?”
Counting cycles:

```c
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
",
  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:

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

```c
int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d
", 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:

```c
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.
Counting cycles:

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

```c
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.

Try moving the pointer:

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

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

```c
int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d
", 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.

Try moving the pointer:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    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?”

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 -0s: 8012 cycles.
Try -01: 8012 cycles.
Try -02: 8012 cycles.
Try -03: 8012 cycles.

Try moving the pointer:

```c
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:

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

8010 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:

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

8010 cycles.

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000;i > 0;--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:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
```
8010 cycles.

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000; i > 0; --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:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
8010 cycles.
```

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000; i > 0; --i)
        result += *x++;
    return result;
}
```
Try moving the pointer:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
8010 cycles.
```

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000; i > 0; --i)
        result += *x++;
    return result;
}
```
Try moving the pointer:

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

8010 cycles.

Try counting down:

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

8010 cycles.
Try moving the pointer:

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

8010 cycles.

Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
8010 cycles.
```
Try counting down:

```c
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:

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

8010 cycles.

Back to original. Try unrolling:

```c
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 counting down:

```c
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:

```c
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:

```c
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 counting down:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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.
Try using an end pointer:

```c
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:

```c
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.

Also try unrolling by 5:

```c
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;
}
```
Try using an end pointer:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; i += 2)
        result += x[i];
    return result;
}
5016 cycles.
```

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; i += 5)
        result += x[i];
    return result;
}
```
Try using an end pointer:

```c
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:

```c
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.

Back to original. Try unrolling:

```c
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;
}
```
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.
Try unrolling:

```c
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;
}

4016 cycles.
```

Are we done now?
Try unrolling:

```c
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.

4016 cycles. Are we done now?
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;
}

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?
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;
}
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;
}
int sum(int *x) {
  int result = 0;
  for (int 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.

Rely on Wikipedia comment that M4F = M4 + floating-point unit.
Manual says that Cortex-M4 “implements the ARMv7E-M architecture”.
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.
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.

Rely on Wikipedia comment that M4F = M4 + floating-point.

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.
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
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.
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.

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.
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.
10

Are we done now?

Most random “optimizations”
that we tried seem useless.
Can spend time trying more.

Can 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.

11

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 $x$ need 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.

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.

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.
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.
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. 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. n consecutive LDRs takes only n + 1 cycle ("more multiple LDRs can be pipelined together"). Can achieve this speed in other ways (LDRD, LDM) but nothing seems faster.

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.
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 $i$. 
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. Read instruction: LDR. Manual says 2 cycles but adds a note about “pipelining”.

More explanation: if next instruction is also LDR (with address not based on first LDR) 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)
```
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 array needs to be "loaded" into a register.

Operation: LDR. Takes 1 cycle but adds "pipelining".

Operation: 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)
```

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:

$$xi9 = *(uint32 *) (p + 76)$$
$$xi8 = *(uint32 *) (p + 72)$$
$$xi7 = *(uint32 *) (p + 68)$$
$$xi6 = *(uint32 *) (p + 64)$$
$$xi5 = *(uint32 *) (p + 60)$$
$$xi4 = *(uint32 *) (p + 56)$$
$$xi3 = *(uint32 *) (p + 52)$$
$$xi2 = *(uint32 *) (p + 48)$$
$n$ 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 $\text{ldr.w}$:

\begin{align*}
y &= x + 4000 \\
p &= x \\
\text{result} &= 0
\end{align*}

\begin{verbatim}
loop:
  \text{xi9} = \ast(\text{uint32 } \ast) (p + 76) \\
  \text{xi8} = \ast(\text{uint32 } \ast) (p + 72) \\
  \text{xi7} = \ast(\text{uint32 } \ast) (p + 68) \\
  \text{xi6} = \ast(\text{uint32 } \ast) (p + 64) \\
  \text{xi5} = \ast(\text{uint32 } \ast) (p + 60) \\
  \text{xi4} = \ast(\text{uint32 } \ast) (p + 56) \\
  \text{xi3} = \ast(\text{uint32 } \ast) (p + 52) \\
  \text{xi2} = \ast(\text{uint32 } \ast) (p + 48)
\end{verbatim}
consecutive LDRs
takes only \( n + 1 \) cycles
(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:

2 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:

\[
\begin{align*}
\xi_9 &= *(\text{uint32 } *)(p + 76) \\
\xi_8 &= *(\text{uint32 } *)(p + 72) \\
\xi_7 &= *(\text{uint32 } *)(p + 68) \\
\xi_6 &= *(\text{uint32 } *)(p + 64) \\
\xi_5 &= *(\text{uint32 } *)(p + 60) \\
\xi_4 &= *(\text{uint32 } *)(p + 56) \\
\xi_3 &= *(\text{uint32 } *)(p + 52) \\
\xi_2 &= *(\text{uint32 } *)(p + 48) \\
\end{align*}
\]
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:

\[
xi9 = *(\text{uint32} *) (p + 76)
\]
\[
xi8 = *(\text{uint32} *) (p + 72)
\]
\[
xi7 = *(\text{uint32} *) (p + 68)
\]
\[
xi6 = *(\text{uint32} *) (p + 64)
\]
\[
xi5 = *(\text{uint32} *) (p + 60)
\]
\[
xi4 = *(\text{uint32} *) (p + 56)
\]
\[
xi3 = *(\text{uint32} *) (p + 52)
\]
\[
xi2 = *(\text{uint32} *) (p + 48)
\]
\[
xi1 = *(\text{uint32} *) (p + 44)
\]
\[
xi0 = *(\text{uint32} *) (p + 40)
\]
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0

xi9 = *(\text{uint32} *) (p + 36)
xi8 = *(\text{uint32} *) (p + 32)
xi7 = *(\text{uint32} *) (p + 28)
13

2281 cycles using ldr.w:

\[ y = x + 4000 \]
\[ p = x \]
\[ \text{result} = 0 \]

\textbf{loop:}

\begin{align*}
\text{x19} & = *(\text{uint32 *})(p + 76) \\
\text{x18} & = *(\text{uint32 *})(p + 72) \\
\text{x17} & = *(\text{uint32 *})(p + 68) \\
\text{x16} & = *(\text{uint32 *})(p + 64) \\
\text{x15} & = *(\text{uint32 *})(p + 60) \\
\text{x14} & = *(\text{uint32 *})(p + 56) \\
\text{x13} & = *(\text{uint32 *})(p + 52) \\
\text{x12} & = *(\text{uint32 *})(p + 48) \\
\end{align*}

14

\begin{align*}
\text{x11} & = *(\text{uint32 *})(p + 44) \\
\text{x10} & = *(\text{uint32 *})(p + 40) \\
\text{result} & += \text{x19} \\
\text{result} & += \text{x18} \\
\text{result} & += \text{x17} \\
\text{result} & += \text{x16} \\
\text{result} & += \text{x15} \\
\text{result} & += \text{x14} \\
\text{result} & += \text{x13} \\
\text{result} & += \text{x12} \\
\text{result} & += \text{x11} \\
\end{align*}

\begin{align*}
\text{x9} & = *(\text{uint32 *})(p + 36) \\
\text{x8} & = *(\text{uint32 *})(p + 32) \\
\text{x7} & = *(\text{uint32 *})(p + 28) \\
\text{x6} & = *(\text{uint32 *})(p + 24) \\
\text{x5} & = *(\text{uint32 *})(p + 20) \\
\text{x4} & = *(\text{uint32 *})(p + 16) \\
\text{x3} & = *(\text{uint32 *})(p + 12) \\
\text{x2} & = *(\text{uint32 *})(p + 8) \\
\text{x1} & = *(\text{uint32 *})(p + 4) \\
\text{x0} & = *(\text{uint32 *})(p + 0) \\
\end{align*}
2281 cycles using ldr.w:

\[ y = x + 4000 \]
\[ p = x \]
\[ \text{result} = 0 \]

\textbf{loop:}

\[ \text{x19} = *(\text{uint32} *) (p + 76) \]
\[ \text{x18} = *(\text{uint32} *) (p + 72) \]
\[ \text{x17} = *(\text{uint32} *) (p + 68) \]
\[ \text{x16} = *(\text{uint32} *) (p + 64) \]
\[ \text{x15} = *(\text{uint32} *) (p + 60) \]
\[ \text{x14} = *(\text{uint32} *) (p + 56) \]
\[ \text{x13} = *(\text{uint32} *) (p + 52) \]
\[ \text{x12} = *(\text{uint32} *) (p + 48) \]
\[ \text{x11} = *(\text{uint32} *) (p + 44) \]
\[ \text{x10} = *(\text{uint32} *) (p + 40) \]
\[ \text{result} += \text{x19} \]
\[ \text{result} += \text{x18} \]
\[ \text{result} += \text{x17} \]
\[ \text{result} += \text{x16} \]
\[ \text{result} += \text{x15} \]
\[ \text{result} += \text{x14} \]
\[ \text{result} += \text{x13} \]
\[ \text{result} += \text{x12} \]
\[ \text{result} += \text{x11} \]
\[ \text{result} += \text{x10} \]
\[ \text{x19} = *(\text{uint32} *) (p + 36) \]
\[ \text{x18} = *(\text{uint32} *) (p + 32) \]
\[ \text{x17} = *(\text{uint32} *) (p + 28) \]
14 cycles using ldr.w:

\[ y = x + 4000 \]
\[ p = x \]
\[ \text{result} = 0 \]

**loop:**

\[ \text{xi9} = *(\text{uint32} *) (p + 76) \]
\[ \text{xi8} = *(\text{uint32} *) (p + 72) \]
\[ \text{xi7} = *(\text{uint32} *) (p + 68) \]
\[ \text{xi6} = *(\text{uint32} *) (p + 64) \]
\[ \text{xi5} = *(\text{uint32} *) (p + 60) \]
\[ \text{xi4} = *(\text{uint32} *) (p + 56) \]
\[ \text{xi3} = *(\text{uint32} *) (p + 52) \]
\[ \text{xi2} = *(\text{uint32} *) (p + 48) \]
\[ \text{xi1} = *(\text{uint32} *) (p + 44) \]
\[ \text{xi0} = *(\text{uint32} *) (p + 40) \]

\[ \text{result} += \text{xi9} \]
\[ \text{result} += \text{xi8} \]
\[ \text{result} += \text{xi7} \]
\[ \text{result} += \text{xi6} \]
\[ \text{result} += \text{xi5} \]
\[ \text{result} += \text{xi4} \]
\[ \text{result} += \text{xi3} \]
\[ \text{result} += \text{xi2} \]
\[ \text{result} += \text{xi1} \]
\[ \text{result} += \text{xi0} \]

\[ \text{xi9} = *(\text{uint32} *) (p + 36) \]
\[ \text{xi8} = *(\text{uint32} *) (p + 32) \]
\[ \text{xi7} = *(\text{uint32} *) (p + 28) \]
\[ \text{xi6} = *(\text{uint32} *) (p + 24) \]
\[ \text{xi5} = *(\text{uint32} *) (p + 20) \]
\[ \text{xi4} = *(\text{uint32} *) (p + 16) \]
\[ \text{xi3} = *(\text{uint32} *) (p + 12) \]
\[ \text{xi2} = *(\text{uint32} *) (p + 8) \]
\[ \text{xi1} = *(\text{uint32} *) (p + 4) \]
\[ \text{result} += \text{xi9} \]
\[ \text{result} += \text{xi8} \]
\[ \text{result} += \text{xi7} \]
\[ \text{result} += \text{xi6} \]
\[ \text{result} += \text{xi5} \]
\[ \text{result} += \text{xi4} \]
\[ \text{result} += \text{xi3} \]
\[ \text{result} += \text{xi2} \]
\[ \text{result} += \text{xi1} \]
\[ \text{result} += \text{xi0} \]

\[ \text{result} += \text{xi9} \]
\[ \text{result} += \text{xi8} \]
\[ \text{result} += \text{xi7} \]
\[ \text{result} += \text{xi6} \]
\[ \text{result} += \text{xi5} \]
\[ \text{result} += \text{xi4} \]
\[ \text{result} += \text{xi3} \]
\[ \text{result} += \text{xi2} \]
\[ \text{result} += \text{xi1} \]
\[ \text{result} += \text{xi0} \]

15
14 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
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
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)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
xi0 = *(uint32 *) p; p += 160
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
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

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
\[ \begin{align*}
\text{xi6} &= *(\text{uint32} \ *) (p + 24) \\
\text{xi5} &= *(\text{uint32} \ *) (p + 20) \\
\text{xi4} &= *(\text{uint32} \ *) (p + 16) \\
\text{xi3} &= *(\text{uint32} \ *) (p + 12) \\
\text{xi2} &= *(\text{uint32} \ *) (p + 8) \\
\text{xi1} &= *(\text{uint32} \ *) (p + 4) \\
\text{xi0} &= *(\text{uint32} \ *) p; p += 160 \\
\text{result} &= +\text{xi9} \\
\text{result} &= +\text{xi8} \\
\text{result} &= +\text{xi7} \\
\text{result} &= +\text{xi6} \\
\text{result} &= +\text{xi5} \\
\text{result} &= +\text{xi4} \\
\text{result} &= +\text{xi3} \\
\text{result} &= +\text{xi2} \\
\text{result} &= +\text{xi1} \\
\text{result} &= +\text{xi0} \\
\text{result} &= +\text{xi9} \\
\text{result} &= +\text{xi8} \\
\text{result} &= +\text{xi7} \\
\end{align*} \]
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
*(uint32 *) (p + 24)
*(uint32 *) (p + 20)
*(uint32 *) (p + 16)
*(uint32 *) (p + 12)
*(uint32 *) (p + 8)
*(uint32 *) (p + 4)
*(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 += xi1
result += xi0
xi9 = *(uint32 *) (p - 4)
xi8 = *(uint32 *) (p - 8)
xi7 = *(uint32 *) (p - 12)
xi6 = *(uint32 *) (p - 16)
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)
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)
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)
result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2
result += xi1
result += xi0
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)
result += xi1
result += 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

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)
\[ \text{result} += x_{i1} \]
\[ \text{result} += x_{i0} \]
\[ x_{i9} = *(\text{uint32} *) \quad (p - 4) \]
\[ x_{i8} = *(\text{uint32} *) \quad (p - 8) \]
\[ x_{i7} = *(\text{uint32} *) \quad (p - 12) \]
\[ x_{i6} = *(\text{uint32} *) \quad (p - 16) \]
\[ x_{i5} = *(\text{uint32} *) \quad (p - 20) \]
\[ x_{i4} = *(\text{uint32} *) \quad (p - 24) \]
\[ x_{i3} = *(\text{uint32} *) \quad (p - 28) \]
\[ x_{i2} = *(\text{uint32} *) \quad (p - 32) \]
\[ x_{i1} = *(\text{uint32} *) \quad (p - 36) \]
\[ x_{i0} = *(\text{uint32} *) \quad (p - 40) \]
\[ \text{result} += x_{i9} \]
\[ \text{result} += x_{i8} \]
\[ \text{result} += x_{i7} \]
\[ \text{result} += x_{i6} \]
\[ \text{result} += x_{i5} \]
\[ \text{result} += x_{i4} \]
\[ \text{result} += x_{i3} \]
\[ \text{result} += x_{i2} \]
\[ \text{result} += x_{i1} \]
\[ \text{result} += x_{i0} \]
\[ x_{i1} = *(\text{uint32} *) \quad (p - 44) \]
\[ x_{i0} = *(\text{uint32} *) \quad (p - 48) \]
\[ x_{i9} = *(\text{uint32} *) \quad (p - 52) \]
\[ x_{i8} = *(\text{uint32} *) \quad (p - 56) \]
\[ x_{i7} = *(\text{uint32} *) \quad (p - 60) \]
\[ x_{i6} = *(\text{uint32} *) \quad (p - 64) \]
\[ x_{i5} = *(\text{uint32} *) \quad (p - 68) \]
\[ x_{i4} = *(\text{uint32} *) \quad (p - 72) \]
\[ \text{goto loop if } \neq \]
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)

= 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
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 !=
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
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)

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

result += xi9
result += xi8
result += xi7
result += xi6
result += xi5
result += xi4
result += xi3
result += xi2

=? p - y
goto loop if !=
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

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

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.
19. *) (p - 76)  
   *) (p - 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.
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

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

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.
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.

Gates $\lor: a, b \mapsto 1$, computing product $h_0 + 2h_1 + 4h_2 + 8h_3$ of integers $f_0 + 2f_1; g_0 + 2g_1$. 
By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts. The fastest software today relies on human experts understanding the CPU.

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

$\wedge$
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.

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.

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

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.

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

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.

Gates \( \land : a, b \mapsto 1 - ab \) 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.

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

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:

\[
\begin{align*}
\text{Gates } \land : a, b &\mapsto 1 - ab \text{ computing product } h_0 + 2h_1 + 4h_2 + 8h_3 \\
of \text{integers } f_0 + 2f_1, g_0 + 2g_1.
\end{align*}
\]
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:

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

Details omitted.)

Register read

reg
CPU design in a nutshell

- 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:
  \[
  f_0 + 2f_1 + g_0 + 2g_1.
  \]
  
  (Details omitted.)
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:

\[ \text{register read} \]

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

\[
\begin{array}{c}
\text{Build circuit for “register write”: } \\
r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15} \\
\text{where } r'_j = r_j \text{ except } r'_i = s.
\end{array}
\]

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

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

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

\[
\begin{align*}
\text{register read} \\
\end{align*}
\]

where \( r_j' = r_j \) except \( r_i' = s \).

Build circuit for “register write”:

\[
\begin{align*}
\text{register write} \\
\end{align*}
\]

\[
\begin{align*}
r_0, ..., r_{15}, s, i \mapsto r_0', ..., r_{15}' \\
\end{align*}
\]

where \( r_j' = r_j \) except \( r_i' = s \).

Build circuit for addition. Etc.
Electricity takes time to percolate through wires and gates. If \( f_0; f_1; g_0; g_1 \) are stable then \( h_0; h_1; \ldots \) are stable a few moments later.

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

\[
\begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  & & & \\
\end{array}
\]

Details omitted.

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

\[
\begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  & & & \\
\end{array}
\]

Build circuit for “register write”:

\[
\begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  & & & \\
\end{array}
\]

\[
r_0, \ldots, r_{15}, i, j, k \mapsto r_0', \ldots, r_{15}'
\]

where \( r'_\ell = r_\ell \) except \( r'_i = s \).

Build circuit for addition. Etc.
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.
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”:
\[
\begin{align*}
r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}
\end{align*}
\]
where \( r'_j = r_j \) except \( r'_i = s \).
Build circuit for addition. Etc.
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

Register read

Register write

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

Add more flexibility.

More arithmetic:
replace $(i, j, k)$ with $(\times, i, j, k)$ and $(+, i, j, k)$ and more options.

Add circuit for “register write”:
$r_0, \ldots, r_{15}, s, i \mapsto r'_0, \ldots, r'_{15}$
$r'_j = r_j$ except $r'_i = s$.
Circuit for addition. Etc.
Build circuit to compute 32-bit integer \( r \) given 4-bit integer \( i \) and 32-bit integers \( r_0, r_1, \ldots, r_{15} \):

\[
\begin{align*}
r_0, \ldots, r_{15}, i, j, k &\mapsto r'_0, \ldots, r'_ {15} \\
\text{where } r'_\ell &= r_\ell \text{ except } r'_i = r_j r_k:
\end{align*}
\]

Add more flexibility.
More arithmetic:
replace \((i, j, k)\) with \(("\times", i, j, k)\) and \(("+", i, j, k)\) and more options.
Build circuit to compute 32-bit integer $r_i$ given 4-bit integer $i$ and 32-bit integers $r_0, r_1, \ldots, r_{15}$.

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

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

Add more flexibility.

More arithmetic:
replace $(i, j, k)$ with
("$\times$", $i, j, k$) and
("+$", $i, j, k$) and more options.
Add more flexibility.
More arithmetic:
replace \((i, j, k)\) with
\((\times, i, j, k)\) and
\((+ , i, j, k)\) and more options.
Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with
\(("\times", i, j, k)\) and
\(("+", i, j, k)\) 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
\((\times, i, j, k)\) and
\((+, i, j, k)\) 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 \( r'_i = r_j r_k \):

<table>
<thead>
<tr>
<th>register read</th>
<th>register read</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i \times j )</td>
<td>( i + j )</td>
</tr>
</tbody>
</table>

Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with
\(("\times", i, j, k)\) and
\(("+", i, j, k)\) 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 \( o_p, i_p, j_p, k_p, p' \).
\[ r_{i} = r_{\ell} \text{ except } r'_{i} = r_{j} r_{k}; \]

Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with \((\times, i, j, k)\) and \((+, i, j, k)\) and more options.

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.
Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with
\((\times, i, j, k)\) and
\((+, i, j, k)\) 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.
Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with
\((\times, i, j, k)\) and
\((+, i, j, k)\) 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.
Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with
\((\times, i, j, k)\) and
\((+, i, j, k)\) 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”
storng \((p, r_0, \ldots, r_{15})\).

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.

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

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.
Add more flexibility.

More arithmetic:
replace \((i, j, k)\) with
\(" \times \); i; j; k\) and
\("+\); i; j; k\) and more options.

More (but slower) storage:
"load" from and "store" to
"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.

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

More arithmetic:
replace \((i; j; k)\) with
\((×; i; j; k)\) and
\((+; i; j; k)\) and more options.

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

“Instruction fetch”:
\(p \mapsto oo; i_p; j_p; k_p; p'\).

“Instruction decode”:
decompression of compressed format for \(oo; i_p; j_p; k_p; p'\).

26

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:
flip-flops
insn
fetch
insn
decode

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.
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.

Outputs $(p', r'_0, \ldots, r'_{15})$ hook back 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.

“Pipelining” allows faster clock:
Now have semi-flexible CPU:

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

“Pipelining” allows faster clock:

stage 1

stage 2

stage 3

stage 4

stage 5
Now have semi-flexible CPU:

```
Now have semi-flexible CPU:
flip-flops
insn
fetch
insn
decode
register
read
register
read


?
?
?
?
?
?
?

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

“Pipelining” allows faster clock:

```
“Pipelining” allows faster clock:
flip-flops
insn
fetch
stage 1
flip-flops
insn
decode
stage 2
flip-flops
register
read
register
read
stage 3
flip-flops
register
write
stage 4
flip-flops
register
write
stage 5
```
Now have semi-flexible CPU:

- flip-flops
- insn
- fetch
- insn
- decode
- register
  - read
  - read
- register
- write

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

"Pipelining" allows faster clock:

- flip-flops
- insn
- fetch
- insn
- decode
- stage 1

- flip-flops
- insn
- decode
- stage 2

- flip-flops
- register
  - read
  - read
- stage 3

- flip-flops
- register
  - read
  - write
- stage 4

- flip-flops
- register
  - write
- stage 5

Goal: Stage \( n \) handles instruction one tick after stage \( n - 1 \).

Instruction fetch reads next instruction, feeds \( p' \), 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.
Flexible CPU:

“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 decoder 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.
“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.
“Pipelining” allows faster clock:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>2</td>
<td>Instruction decode</td>
</tr>
<tr>
<td>3</td>
<td>Register read</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Register write</td>
</tr>
</tbody>
</table>

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.
“Pipelining” allows faster clock:

- Stage 1: Instruction fetch
- Stage 2: Instruction decode
- Stage 3: Register read
- Stage 4: Register read
- Stage 5: Register write

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:
Pipelining allows faster clock:

- Stage 1: Flip-flops
- Stage 2: Instruction fetch
- Stage 3: Decode
- Stage 4: Some extra flip-flop area.
- Stage 5: Register write

**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:

<table>
<thead>
<tr>
<th>flip-flops</th>
<th>flip-flops</th>
<th>flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn fetch</td>
<td>insn fetch</td>
<td>insn decode</td>
</tr>
<tr>
<td>insn decode</td>
<td>insn decode</td>
<td>insn decode</td>
</tr>
<tr>
<td>flip-flops</td>
<td>flip-flops</td>
<td>flip-flops</td>
</tr>
<tr>
<td>register read</td>
<td>register read</td>
<td>register read</td>
</tr>
<tr>
<td>register read</td>
<td>register read</td>
<td>register read</td>
</tr>
<tr>
<td>flip-flops</td>
<td>flip-flops</td>
<td>flip-flops</td>
</tr>
<tr>
<td>register write</td>
<td>register write</td>
<td>register write</td>
</tr>
</tbody>
</table>
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:

```
<table>
<thead>
<tr>
<th>flip-flops</th>
<th>flip-flops</th>
<th>flip-flops</th>
<th>flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn fetch</td>
<td>insn fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insn decode</td>
<td>insn decode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register read</td>
<td>register read</td>
<td>register read</td>
<td>register read</td>
</tr>
</tbody>
</table>
```

"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:

<table>
<thead>
<tr>
<th>flip-flops</th>
<th>flip-flops</th>
<th>flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn fetch</td>
<td>insn fetch</td>
<td>insn decode</td>
</tr>
<tr>
<td>insn decode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“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 $n$. 

Goal: Stage $n$ handles instruction one tick after stage $n - 1$.
Instruction fetch reads next instruction, feeds instruction.
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.
Goal: Stage $n$ handles instruction one tick after stage $n-1$.

Instruction fetch reads next instruction, feeds 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:

<table>
<thead>
<tr>
<th>flip-flops</th>
<th>insn fetch</th>
<th>insn fetch</th>
<th>flip-flops</th>
<th>insn decode</th>
<th>insn decode</th>
<th>flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>register read</td>
<td>register read</td>
<td>register read</td>
<td>register read</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“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:

<table>
<thead>
<tr>
<th>flip-flops</th>
<th>insn fetch</th>
<th>insn fetch</th>
<th>flip-flops</th>
<th>insn decode</th>
<th>insn decode</th>
<th>flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>register read</td>
<td>register read</td>
<td>register read</td>
<td>register read</td>
<td>flip-flops</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### “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:

```
<table>
<thead>
<tr>
<th>flip-flops</th>
<th>insn fetch</th>
<th>insn fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>flip-flops</td>
<td>insn decode</td>
<td>insn decode</td>
</tr>
<tr>
<td>flip-flops</td>
<td>registration read</td>
<td>registration read</td>
</tr>
</tbody>
</table>

```

“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:

<table>
<thead>
<tr>
<th>flip-flops</th>
<th>insn fetch</th>
<th>insn fetch</th>
<th>flip-flops</th>
<th>insn decode</th>
<th>insn decode</th>
<th>flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>register read</td>
<td>register read</td>
<td></td>
<td>register read</td>
<td>register read</td>
<td></td>
</tr>
<tr>
<td>flip-flops</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### "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.
“Superscalar” processing:

<table>
<thead>
<tr>
<th>flip-flops</th>
<th>insn</th>
<th>insn</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>fetch</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flip-flops</td>
<td>insn</td>
<td>insn</td>
</tr>
<tr>
<td>decode</td>
<td>decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flip-flops</td>
<td>register</td>
<td>register</td>
</tr>
<tr>
<td>read</td>
<td>read</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flip-flops</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“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 \( \{1, 2, \ldots, n\} \) 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 $\{1, 2, \ldots, n\}$ 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 \( n = 4^k \).
"Vector" processing:

- Expand each 32-bit integer to an n-vector of 32-bit integers.
- ARM "NEON" has $n = 4$;
- "AVX2" has $n = 8$;
- "AVX-512" has $n = 16$;
- GPUs have larger $n$.

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 $n = 4^k$.

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

Expand each 32-bit integer into a $n$-vector of 32-bit integers.

- ARM "NEON" has $n = 4$;
- Intel "AVX2" has $n = 8$;
- Intel "AVX-512" has $n = 16$;
- GPUs have larger $n$.

$\mathbf{n \times \text{speedup}}$ if $\mathbf{n \times \text{arithmetic circuits},}$ $\mathbf{n \times \text{read/write circuits}.}$

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 $\{1, 2, \ldots, \sqrt{n} \}$, 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 a 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:
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; 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.
  
  $3 1 4 1 5 9 2 6 \mapsto 1 3 1 4 5 9 2 6$

- Sort alternate pairs in parallel.
  
  $1 3 1 4 5 2 9 6$

- Repeat until number of steps equals row length.
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.
  3 1 4 1 5 9 2 6 ↦→ 1 3 1 4 5 9 2 6
- Sort alternate pairs in parallel.
  1 3 1 4 5 9 2 6 ↦→ 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.
  \[
  3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ \mapsto \ 1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6
  \]
- 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.
  
  $3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ \rightarrow \ 1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6$

- Sort alternate pairs in parallel.
  
  $1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6 \ \rightarrow \ 1 \ 1 \ 3 \ 4 \ 5 \ 2 \ 9 \ 6$

- 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.
  $3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ \Rightarrow \ 1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6$

- Sort alternate pairs in parallel.
  $1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6 \ \Rightarrow \ 1 \ 1 \ 3 \ 4 \ 5 \ 2 \ 9 \ 6$

- Repeat until number of steps equals row length.

Sort each row, in parallel, in a total of $n^{0.5+o(1)}$ seconds.
Spread array across square mesh of \( n \) small cells, each of area \( n^o(1) \), with near-neighbor wiring:

\[
\begin{array}{cccccccccccc}
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\end{array}
\]

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

- Sort each pair in parallel.
  \[
  3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ \rightarrow \\
  1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6 
  \]

- Sort alternate pairs in parallel.
  \[
  1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6 \ \rightarrow \\
  1 \ 1 \ 3 \ 4 \ 5 \ 2 \ 9 \ 6 
  \]

- 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 the whole array.
Spread array across square mesh of \(n\) small cells, each of area \(n^o(1)\), with near-neighbor wiring:

\[
\begin{array}{cccccccccccccccc}
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\end{array}
\]

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

- Sort each pair in parallel.
  
  \[3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \ \Rightarrow \ 1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6\]

- Sort alternate pairs in parallel.
  
  \[1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6 \ \Rightarrow \ 1 \ 1 \ 3 \ 4 \ 5 \ 2 \ 9 \ 6\]

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

\[
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\times \times \times \times \times \times \times \times \times \times
\]

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

- Sort each pair in parallel.
  \[
  3 \ 1 \ 4 \ 1 \ 5 \ 9 \ 2 \ 6 \quad \mapsto \quad 1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6
  \]

- Sort alternate pairs in parallel.
  \[
  1 \ 3 \ 1 \ 4 \ 5 \ 9 \ 2 \ 6 \quad \mapsto \quad 1 \ 1 \ 3 \ 4 \ 5 \ 2 \ 9 \ 6
  \]

- 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.
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.

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

- Sort each pair in parallel.
  \[ \begin{array}{cccccc}
  3 & 1 & 4 & 1 & 5 & 9 \\
  2 & 6
  \end{array} \rightarrow
  \begin{array}{cccccc}
  1 & 3 & 1 & 4 & 5 & 9 \\
  2 & 6
  \end{array} \]
- Sort alternate pairs in parallel.
  \[ \begin{array}{cccccc}
  1 & 3 & 1 & 4 & 5 & 9 \\
  2 & 6
  \end{array} \rightarrow
  \begin{array}{cccccc}
  1 & 1 & 3 & 4 & 5 & 2 & 9 & 6
  \end{array} \]
- 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:
\[
\begin{array}{ccccccc}
3 & 1 & 4 & 1 & 5 & 9 & 2 \\
5 & 3 & 5 & 8 & 9 & 7 & 9 \\
2 & 3 & 8 & 4 & 6 & 2 & 6 \\
3 & 3 & 8 & 3 & 2 & 7 & 9 \\
0 & 2 & 8 & 8 & 4 & 1 & 9 \\
1 & 6 & 9 & 3 & 9 & 9 & 3 \\
5 & 1 & 0 & 5 & 8 & 2 & 0 \\
7 & 4 & 9 & 4 & 4 & 5 & 9
\end{array}
\]
Sort all $n$ cells in $n^{0.5+o(1)}$ seconds:

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

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

For example, assume that this $8 \times 8$ array is in cells:

\[
\begin{array}{cccccccc}
3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 \\
5 & 3 & 5 & 8 & 9 & 7 & 9 & 3 \\
2 & 3 & 8 & 4 & 6 & 2 & 6 & 4 \\
3 & 3 & 8 & 3 & 2 & 7 & 9 & 5 \\
0 & 2 & 8 & 8 & 4 & 1 & 9 & 7 \\
1 & 6 & 9 & 3 & 9 & 9 & 3 & 7 \\
5 & 1 & 0 & 5 & 8 & 2 & 0 & 9 \\
7 & 4 & 9 & 4 & 4 & 5 & 9 & 2 \\
\end{array}
\]
Sort all $n$ cells in $n^{0.5+o(1)}$ seconds:

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

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

For example, assume that this $8 \times 8$ array is in cells:

```
3 1 4 1 5 9 2 6
5 3 5 8 9 7 9 3
2 3 8 4 6 2 6 4
3 3 8 3 2 7 9 5
0 2 8 8 4 1 9 7
1 6 9 3 9 9 3 7
5 1 0 5 8 2 0 9
7 4 9 4 4 5 9 2
```
Sort all $n$ cells in $n^{0.5+o(1)}$ seconds:

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

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

For example, assume that this $8 \times 8$ array is in cells:

```
3 1 4 1 5 9 2 6
5 3 5 8 9 7 9 3
2 3 8 4 6 2 6 4
3 3 8 3 2 7 9 5
0 2 8 8 4 1 9 7
1 6 9 3 9 9 3 7
5 1 0 5 8 2 0 9
7 4 9 4 4 5 9 2
```
For example, assume that this $8 \times 8$ array is in cells:

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

Sort all \(n\) cells in \(n^0\) seconds:

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

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

For example, assume that this \(8 \times 8\) array is in cells:

\[
\begin{array}{cccc}
3 & 1 & 4 & 1 \\
5 & 3 & 5 & 8 \\
2 & 3 & 8 & 4 \\
3 & 3 & 8 & 3 \\
0 & 2 & 8 & 8 \\
1 & 6 & 9 & 3 \\
5 & 1 & 0 & 5 \\
7 & 4 & 9 & 4 \\
\end{array}
\]

Recursively sort quadrants, top →, bottom ←:
Sort all \( n \) cells in \( n^0 \) : 5+ o (1) seconds:

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

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

For example, assume that this \( 8 \times 8 \) array is in cells:

\[
\begin{array}{cccc}
3 & 1 & 4 & 1 \\
5 & 3 & 5 & 8 \\
2 & 3 & 8 & 4 \\
3 & 3 & 8 & 3 \\
0 & 2 & 8 & 8 \\
1 & 6 & 9 & 3 \\
5 & 1 & 0 & 5 \\
7 & 4 & 9 & 4 \\
\end{array}
\]

Recursively sort quadrants, top \( \rightarrow \), bottom \( \leftarrow \):

\[
\begin{array}{cccc}
1 & 1 & 2 & 3 \\
3 & 3 & 3 & 3 \\
3 & 4 & 4 & 5 \\
5 & 8 & 8 & 8 \\
1 & 1 & 0 & 0 \\
4 & 4 & 3 & 2 \\
7 & 6 & 5 & 5 \\
9 & 9 & 8 & 8 \\
\end{array}
\]

\[
\begin{array}{cccc}
2 & 2 & 2 & 3 \\
4 & 5 & 5 & 6 \\
6 & 6 & 7 & 7 \\
9 & 9 & 9 & 9 \\
2 & 2 & 1 & 0 \\
5 & 4 & 4 & 3 \\
9 & 8 & 7 & 7 \\
9 & 9 & 9 & 9 \\
\end{array}
\]
For example, assume that this $8 \times 8$ array is in cells:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
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
For example, assume that this 8 × 8 array is in cells:

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>2</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

Recursively sort quadrants, top →, bottom ←:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each column in parallel:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
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 | 0 2 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 →, bottom ←:

<table>
<thead>
<tr>
<th>1 1 2 3</th>
<th>2 2 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 3 3</td>
<td>4 5 5 6</td>
</tr>
<tr>
<td>3 4 4 5</td>
<td>6 6 7 7</td>
</tr>
<tr>
<td>5 8 8 8</td>
<td>9 9 9 9</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>2 2 1 0</td>
</tr>
<tr>
<td>4 4 3 2</td>
<td>5 4 4 3</td>
</tr>
<tr>
<td>7 6 5 5</td>
<td>9 8 7 7</td>
</tr>
<tr>
<td>9 9 8 8</td>
<td>9 9 9 9</td>
</tr>
</tbody>
</table>

Sort each column in parallel:

<table>
<thead>
<tr>
<th>1 1 0 0 2 2 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 2 2 2 2 3 3</td>
</tr>
<tr>
<td>3 3 3 3 4 4 4 3</td>
</tr>
<tr>
<td>3 4 3 3 5 5 5 6</td>
</tr>
<tr>
<td>4 4 4 5 6 6 7 7</td>
</tr>
<tr>
<td>5 6 5 5 9 8 7 7</td>
</tr>
<tr>
<td>7 8 8 8 9 9 9 9</td>
</tr>
<tr>
<td>9 9 8 8 9 9 9 9</td>
</tr>
</tbody>
</table>
Recursively 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
8 2 9 8 7 7
8 8 9 9 9 9
```
Recursively sort quadrants, top $\rightarrow$, bottom $\leftarrow$:

$$
\begin{array}{cccc}
1 & 1 & 2 & 3 \\
2 & 2 & 2 & 3 \\
3 & 3 & 3 & 3 \\
4 & 5 & 5 & 6 \\
3 & 4 & 4 & 5 \\
6 & 6 & 7 & 7 \\
5 & 8 & 8 & 8 \\
9 & 9 & 9 & 9 \\
1 & 1 & 0 & 0 \\
2 & 2 & 1 & 0 \\
4 & 4 & 3 & 2 \\
5 & 4 & 4 & 3 \\
7 & 6 & 5 & 5 \\
9 & 8 & 7 & 7 \\
9 & 9 & 8 & 8 \\
9 & 9 & 9 & 9
\end{array}
$$

Sort each column in parallel:

$$
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
1 & 1 & 2 & 2 \\
3 & 3 & 3 & 4 \\
3 & 4 & 4 & 3 \\
3 & 4 & 3 & 3 \\
3 & 4 & 3 & 3 \\
4 & 4 & 4 & 3 \\
4 & 4 & 4 & 3 \\
5 & 6 & 5 & 5 \\
5 & 6 & 5 & 5 \\
5 & 6 & 5 & 5 \\
6 & 5 & 5 & 6 \\
4 & 4 & 5 & 6 \\
4 & 4 & 5 & 6 \\
9 & 8 & 7 & 7 \\
7 & 8 & 8 & 9 \\
9 & 9 & 9 & 9 \\
9 & 9 & 9 & 9 \\
9 & 9 & 9 & 9
\end{array}
$$

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

$$
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
3 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
3 & 3 & 3 & 3 \\
6 & 5 & 5 & 4 \\
4 & 4 & 5 & 3 \\
4 & 4 & 4 & 3 \\
4 & 4 & 4 & 3 \\
9 & 8 & 7 & 7 \\
7 & 8 & 8 & 9 \\
9 & 9 & 9 & 9 \\
9 & 9 & 9 & 9 \\
9 & 9 & 9 & 9
\end{array}
$$
Recursively sort quadrants, top → , bottom ← :

\[
\begin{array}{cccc}
1 & 1 & 2 & 3 \\
1 & 1 & 2 & 3 \\
3 & 3 & 3 & 3 \\
3 & 4 & 3 & 3 \\
4 & 4 & 4 & 5 \\
5 & 6 & 5 & 5 \\
7 & 8 & 8 & 8 \\
9 & 9 & 8 & 8 \\
\end{array}
\quad
\begin{array}{cccc}
2 & 2 & 2 & 3 \\
2 & 2 & 2 & 3 \\
3 & 4 & 4 & 3 \\
3 & 4 & 3 & 3 \\
5 & 6 & 6 & 7 \\
9 & 8 & 7 & 7 \\
7 & 8 & 8 & 8 \\
9 & 9 & 9 & 9 \\
\end{array}
\]

Sort each column in parallel:

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
1 & 1 & 2 & 2 \\
3 & 3 & 3 & 3 \\
3 & 4 & 3 & 3 \\
4 & 4 & 4 & 5 \\
5 & 6 & 5 & 5 \\
7 & 8 & 8 & 8 \\
9 & 9 & 8 & 8 \\
\end{array}
\quad
\begin{array}{cccc}
2 & 2 & 1 & 0 \\
2 & 2 & 2 & 2 \\
4 & 4 & 3 & 3 \\
4 & 4 & 5 & 6 \\
6 & 5 & 5 & 5 \\
9 & 8 & 7 & 7 \\
9 & 8 & 8 & 8 \\
9 & 9 & 9 & 9 \\
\end{array}
\]

Sort each row in parallel, alternately ← , → :

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
3 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
6 & 5 & 5 & 5 \\
4 & 4 & 4 & 5 \\
9 & 8 & 7 & 7 \\
7 & 8 & 8 & 8 \\
9 & 9 & 9 & 9 \\
\end{array}
\quad
\begin{array}{cccc}
1 & 1 & 1 & 2 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
4 & 3 & 3 & 3 \\
5 & 6 & 6 & 7 \\
6 & 5 & 5 & 5 \\
9 & 9 & 9 & 9 \\
8 & 8 & 8 & 8 \\
\end{array}
\]
Sort each column in parallel:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, alternately ←, →:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Sort each column in parallel:

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, alternately ←, →:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
### Sort each row in parallel, alternately ←, →:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>...</td>
</tr>
</tbody>
</table>
Sort each row in parallel, alternately ←, →:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Sort each column in parallel:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each row in parallel, alternately ←, →:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Sort each column in parallel:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each row in parallel, alternately ←, →:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Sort each column in parallel:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each row in parallel,
alternately $\leftarrow$, $\rightarrow$:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each column in parallel:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, $\leftarrow$ or $\rightarrow$ as desired:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each column in parallel:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, \( \leftarrow \) or \( \rightarrow \) as desired:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each column in parallel:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, ← or → as desired:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each column in parallel:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, ← or → as desired:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

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 column in parallel:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Sort each row in parallel, ← or → as desired:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

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, ← or → as desired:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

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, ← 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 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, ← or → as desired:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort each row in parallel, ← or → as desired:

<table>
<thead>
<tr>
<th>0 0 0 1 1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 2 2 2 2 2 2 3 3</td>
</tr>
<tr>
<td>3 3 3 3 3 3 3 3 3</td>
</tr>
<tr>
<td>4 4 4 4 4 4 4 5 5</td>
</tr>
<tr>
<td>5 5 5 5 5 5 6 6 6</td>
</tr>
<tr>
<td>6 6 7 7 7 7 8 8 8</td>
</tr>
<tr>
<td>8 8 8 8 9 9 9 9 9</td>
</tr>
<tr>
<td>9 9 9 9 9 9 9 9 9</td>
</tr>
</tbody>
</table>

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.