Lecture 14
Concurrency

CS213 – Intro to Computer Systems
Branden Ghena – Winter 2024

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Attack Lab due Today

- Homework 4 is out now
    - Due next week Thursday, only covers cache materials

- SETI Lab releases soon
    - Today’s lecture has 90%+ of the information you need for it
        - And everything you need to get started
    - Tuesday’s lecture will add a few more details about optimization
    - Due last Thursday of class

Today’s Goals

- Discuss goals of concurrency and how it is achieved in software
- Understand the challenges of writing parallel software
- Explore how to practically use parallelism for simple examples

Outline

- **Need for Parallelism**
- Processes and Threads
- Concurrency Challenges
- Using Threads

It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years.

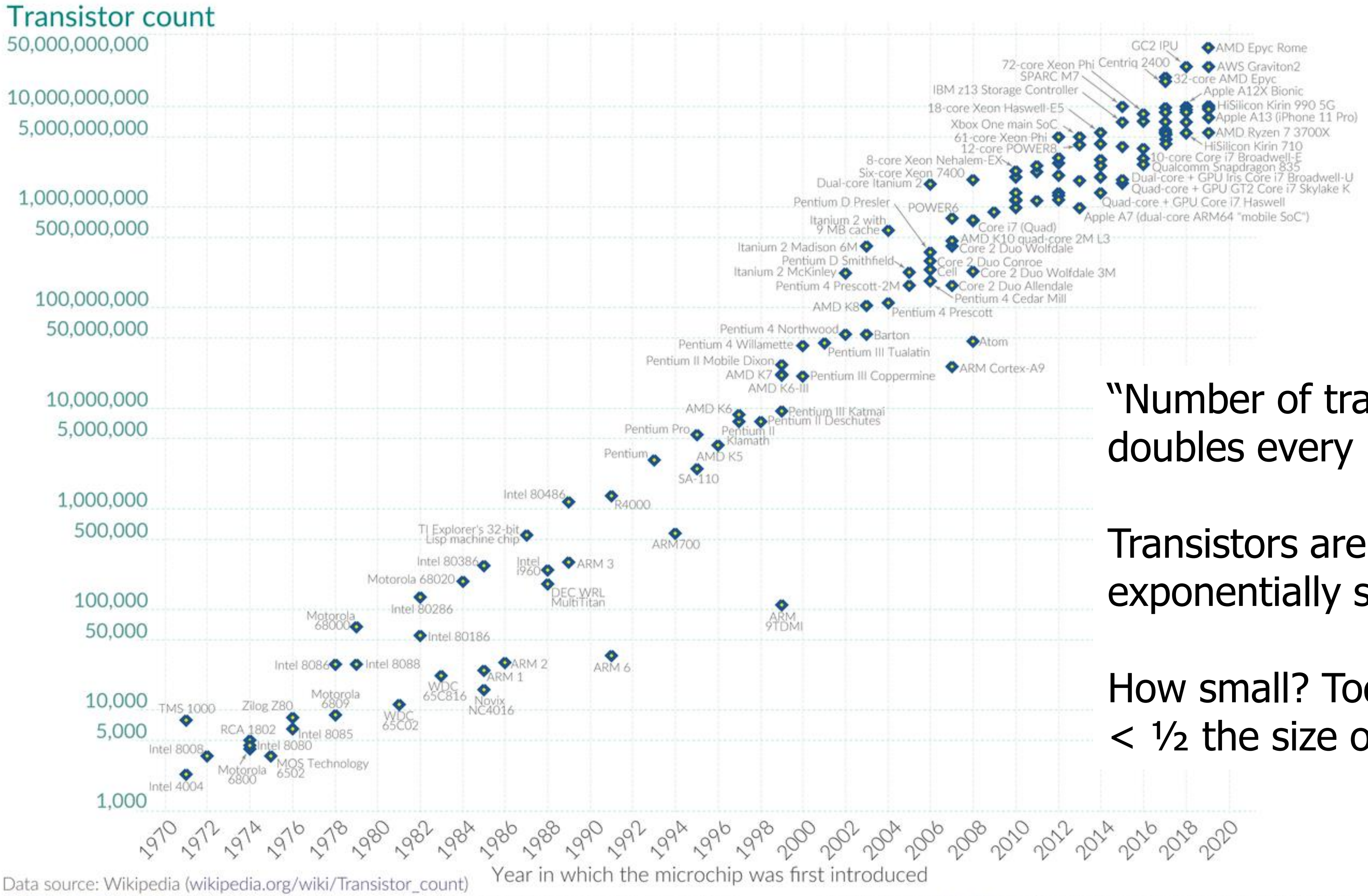
What do you do?

It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years.

What do you do? **Take a vacation**

Moore’s Law – CPU transistors counts



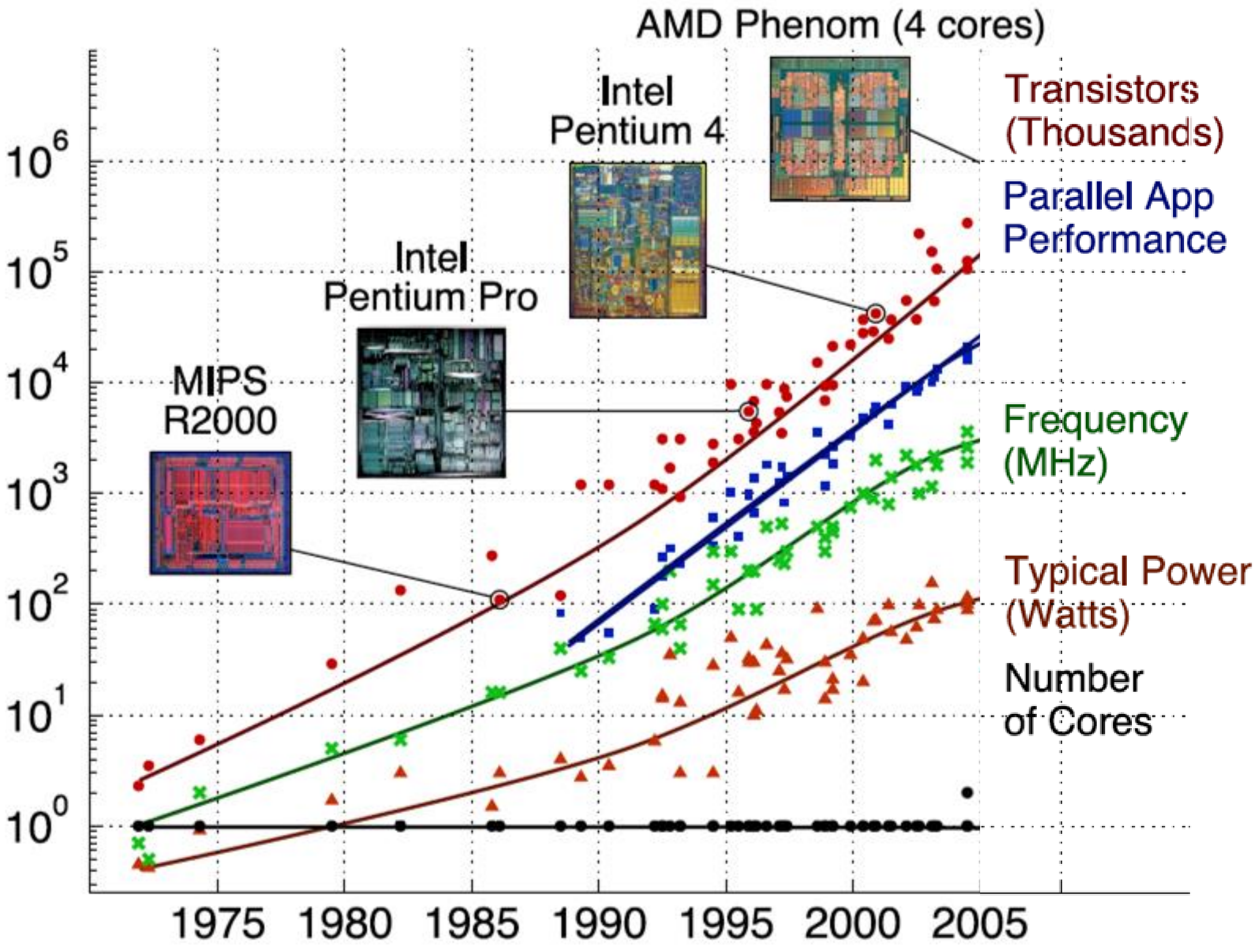
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world’s largest problems.

“Number of transistors in a chip doubles every 18 months”

Transistors are getting exponentially smaller!

How small? Today: 7nm!
< ½ the size of most viruses!

Processors kept getting faster too



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olu

Power is a major limiting factor on speed

- We could make processors go very fast
  - But doing so uses more and more power

- More power means more heat generated
  - And chips typically work up to around 100°C
  - Hotter than that and things stop working

- We add heat sinks and fans and water coolers to keep chips cool
  - But it’s hard to remove heat quickly enough from chips

- So, power consumption ends up limiting processor speed

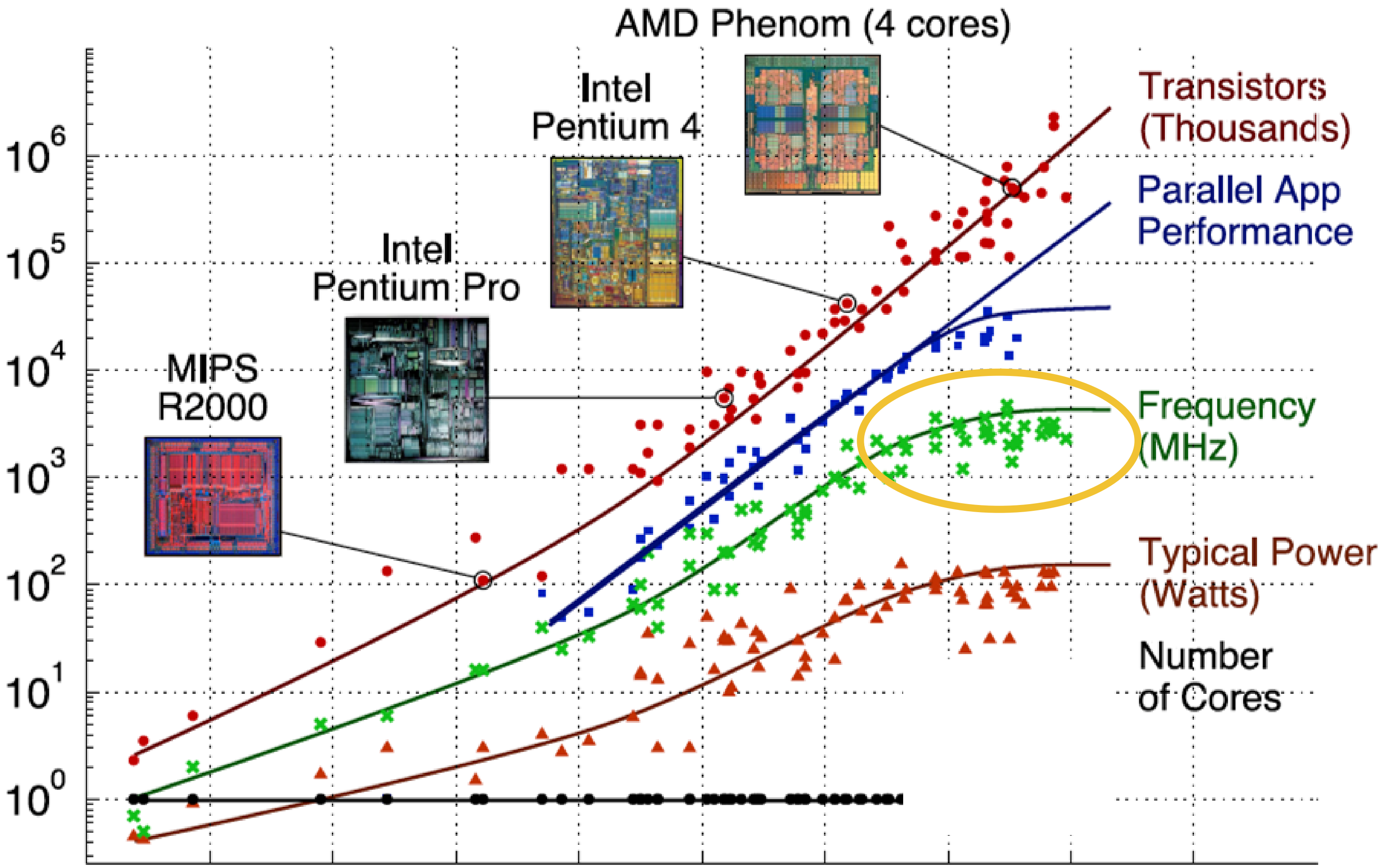
Denard Scaling

- Moore’s Law corollary: Denar Scaling
  - As transistors get smaller, the power density stays the same
  - Which is to say that the power-per-transistor decreases!

- Making the processor clock speed faster uses more power
  - But the two balance out for roughly net even power
  - So not only do we get *more* transistors, but chip speed can be *faster* too

- From our Excel example:
  - In two years, new hardware would run the existing software twice as fast

Then they stopped getting faster



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

~2006: Leakage current becomes significant

Now smaller transistors doesn't mean lower power

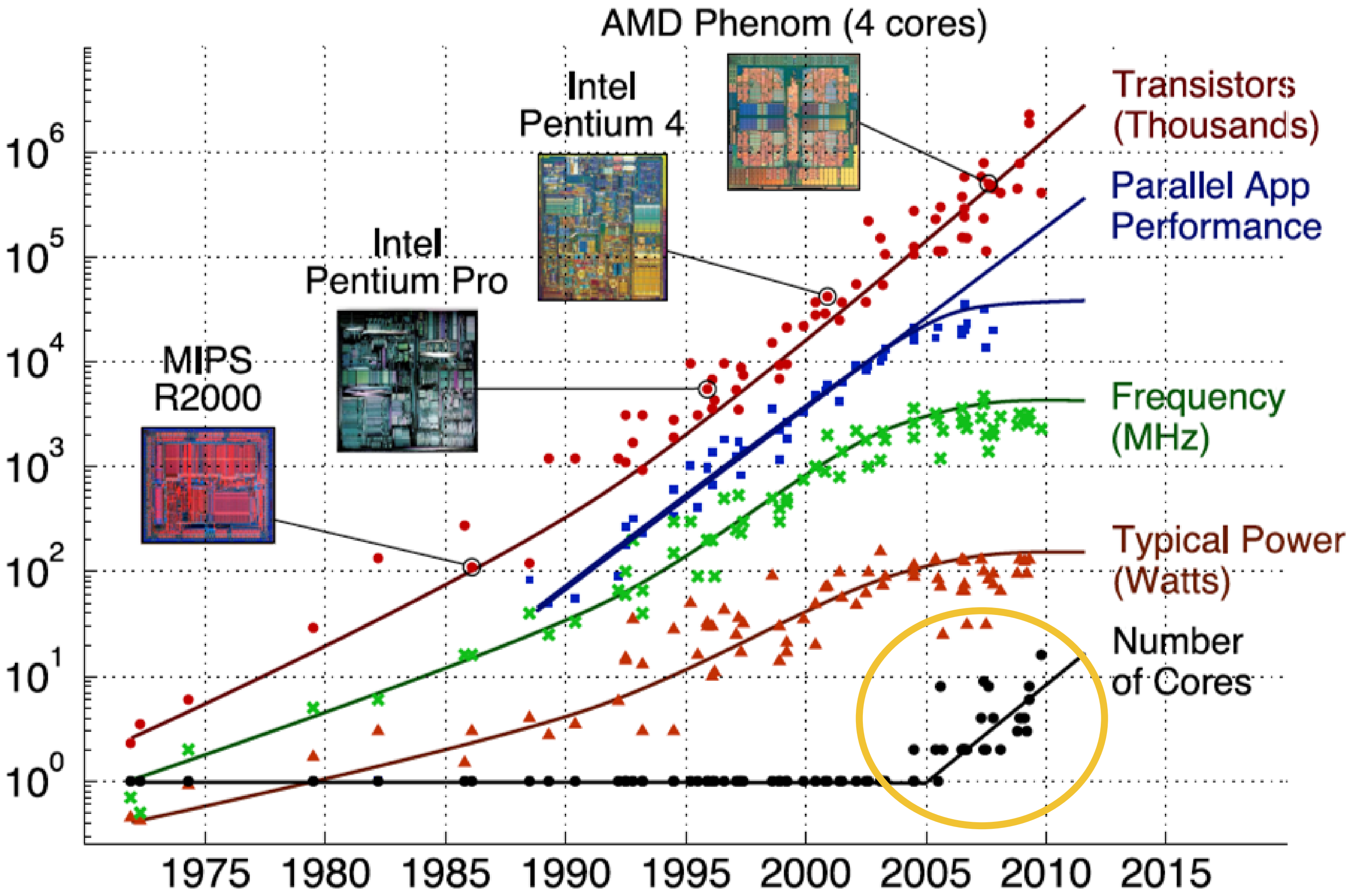
So… now what?

In summary:

- Making transistors smaller doesn’t make them lower power,
- so if we were to make them faster, they would take more power,
- which will eventually lead to our processors melting…
- and because of that, *we can’t reliably make performance better by waiting for clock speeds to increase.*

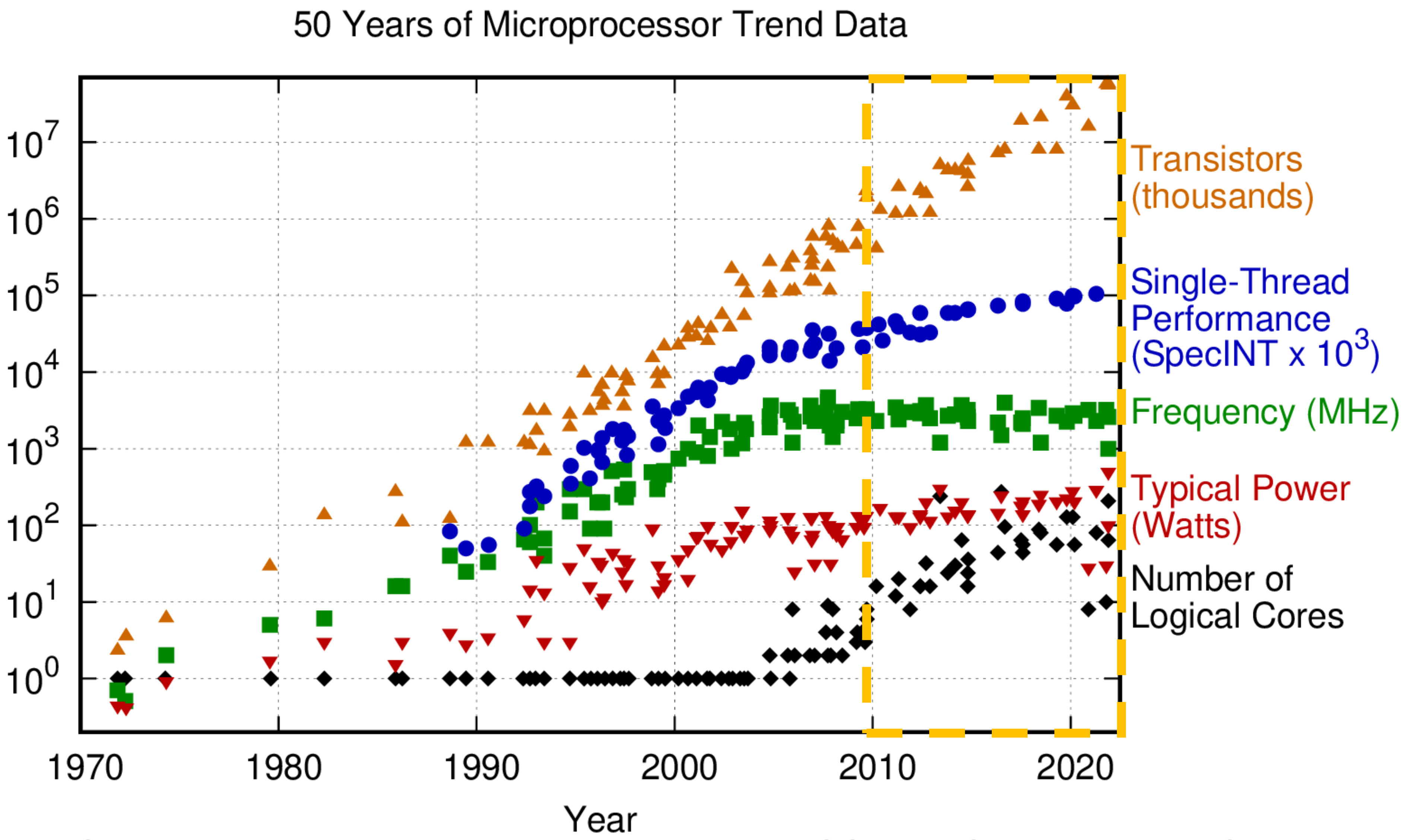
How do we continue to get better computation performance?

Exploit parallelism!



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Update: 2010-2021



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Still growing!

Small improvement

Stable

Stable

Still growing for servers

Parallelism Analogy

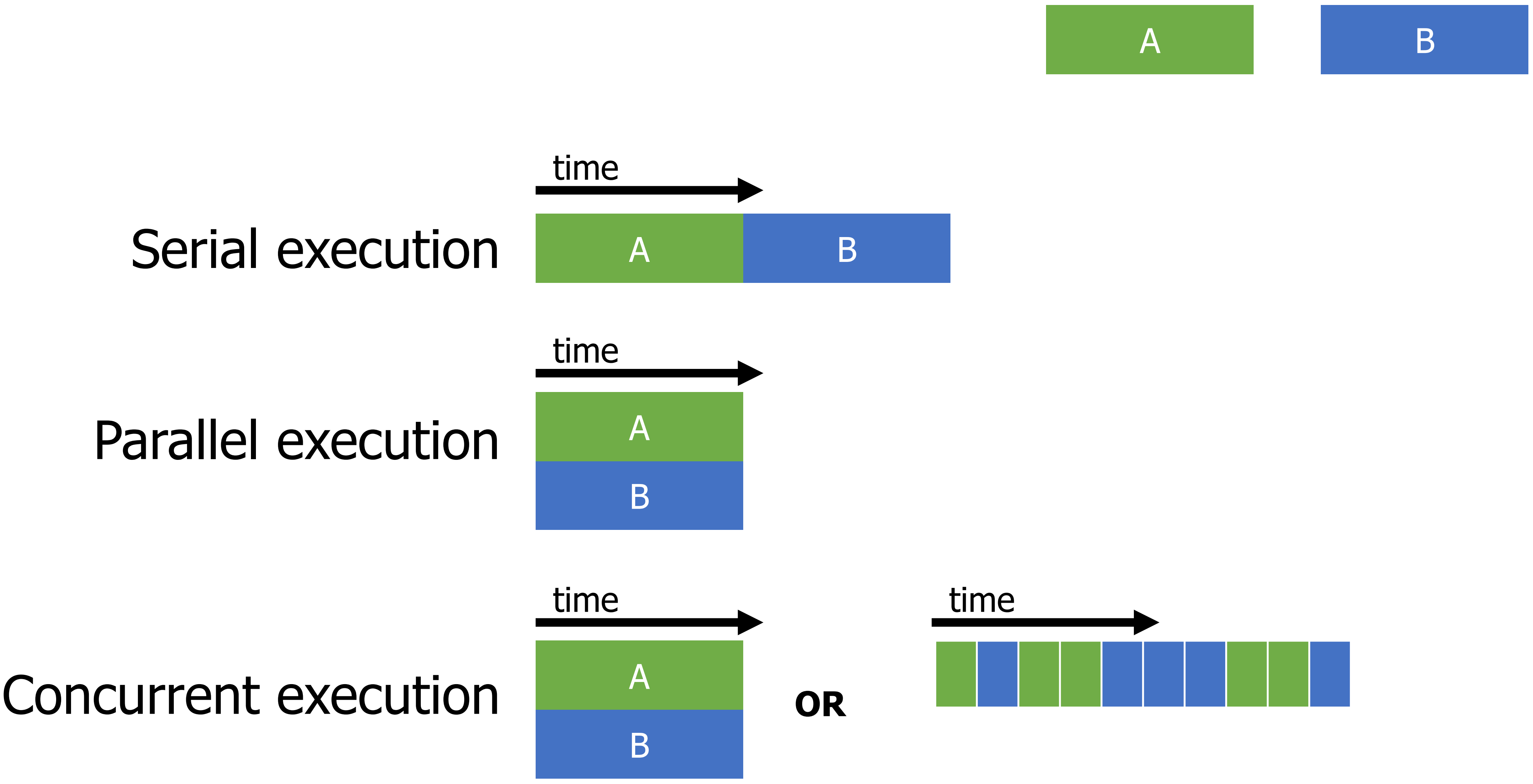
- I want to peel 100 potatoes as fast as possible:
  - I can learn to peel potatoes faster

  OR

  - I can get 99 friends to help me
- Whenever one result doesn’t depend on another, doing the task in parallel can be a big win!

Parallelism versus Concurrency

Two processes A and B



Serial execution

Parallel execution

Concurrent execution

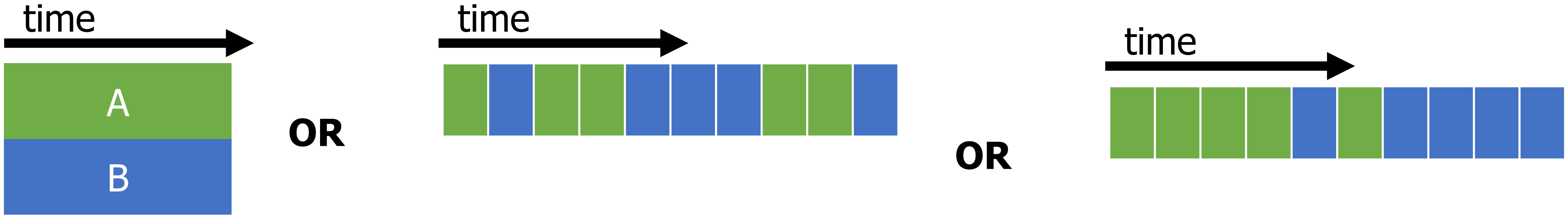
OR

Parallelism versus Concurrency

- Parallelism
  - Two things happen strictly simultaneously

- Concurrency
  - More general term
  - Two things happen in the same time window
    - Could be simultaneous, could be interleaved

  - Concurrent execution occurs whenever two processes are both active



Outline

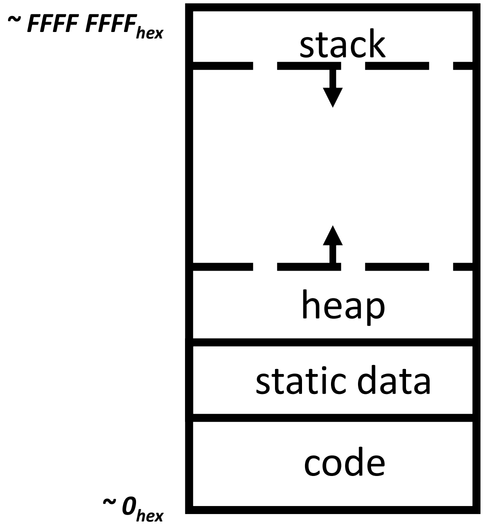
- Need for Parallelism
- **Processes and Threads**
- Concurrency Challenges
- Using Threads

How do we apply parallelism to software?

- Goal: make computer faster by performing multiple tasks
- Need multiple different software tasks
- Two particular ways of creating a software task
  - Processes
  - Threads

View of a process

- Process: a program that is currently being run
- Contents:
  - Address Space



  - Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

  - Instruction Pointer
  - Condition Codes
  - Etc.

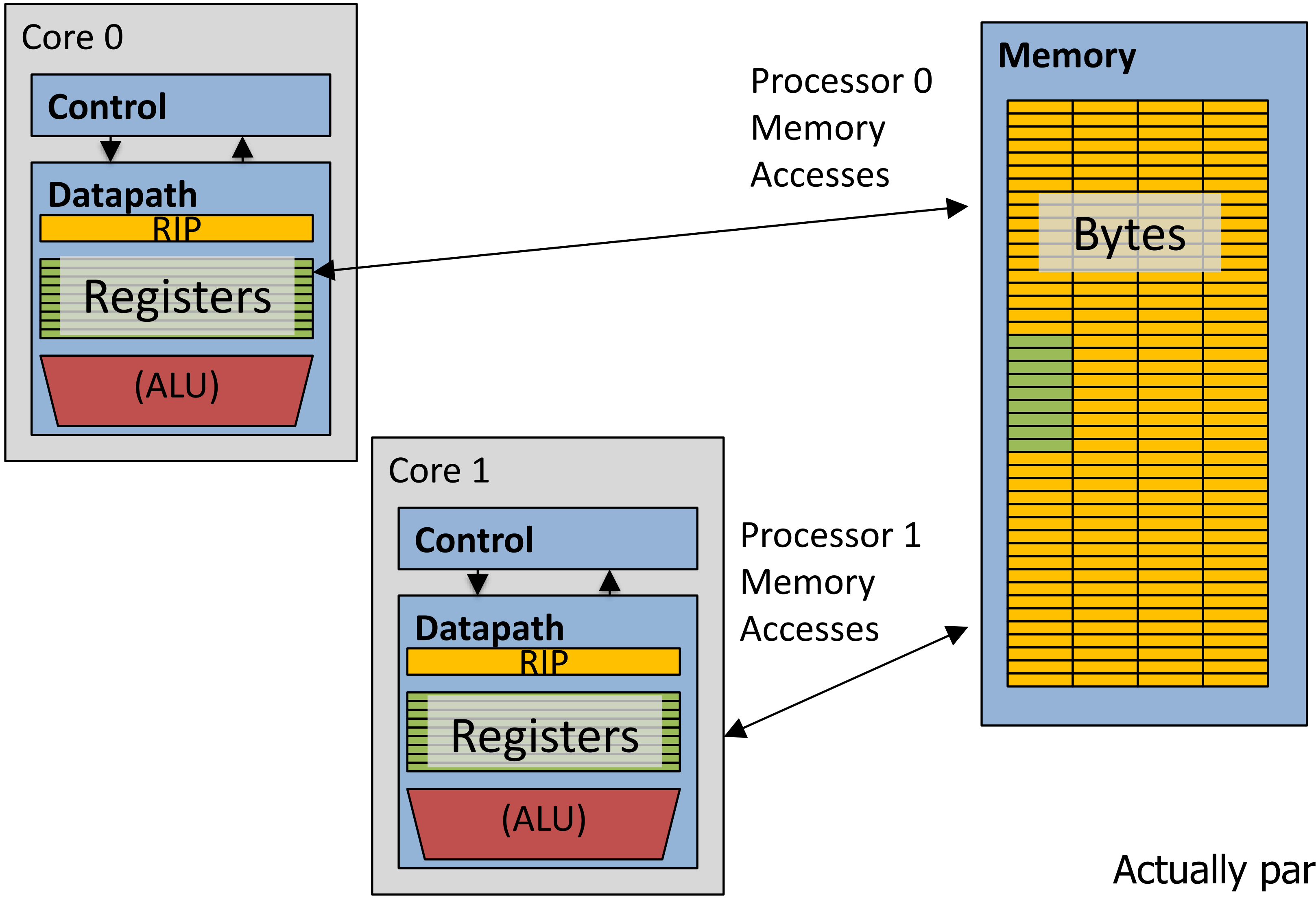
Process use case: separate programs

- Right now I am running:
  - Powerpoint
  - Chrome
  - Notion

- Each is a separate process
  - Have their own memory
  - Have their own registers
  - Operating System manages them

- No need for communication between them

Multiprocessor Systems (in pictures)

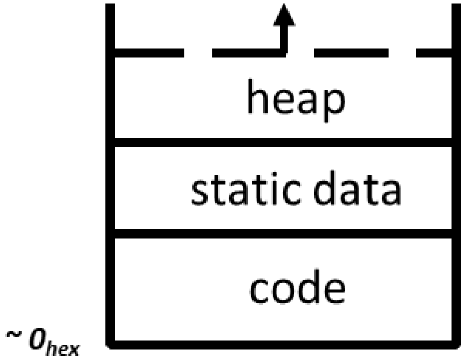


Actually parallel!

Alternate view of a process

- Process: code and data, plus a **thread**
- Thread: execution state
  - Each process has *at least* one thread

- Code and Data

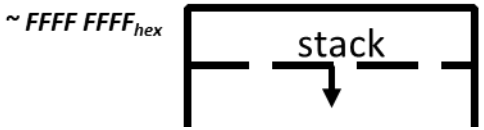


- Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

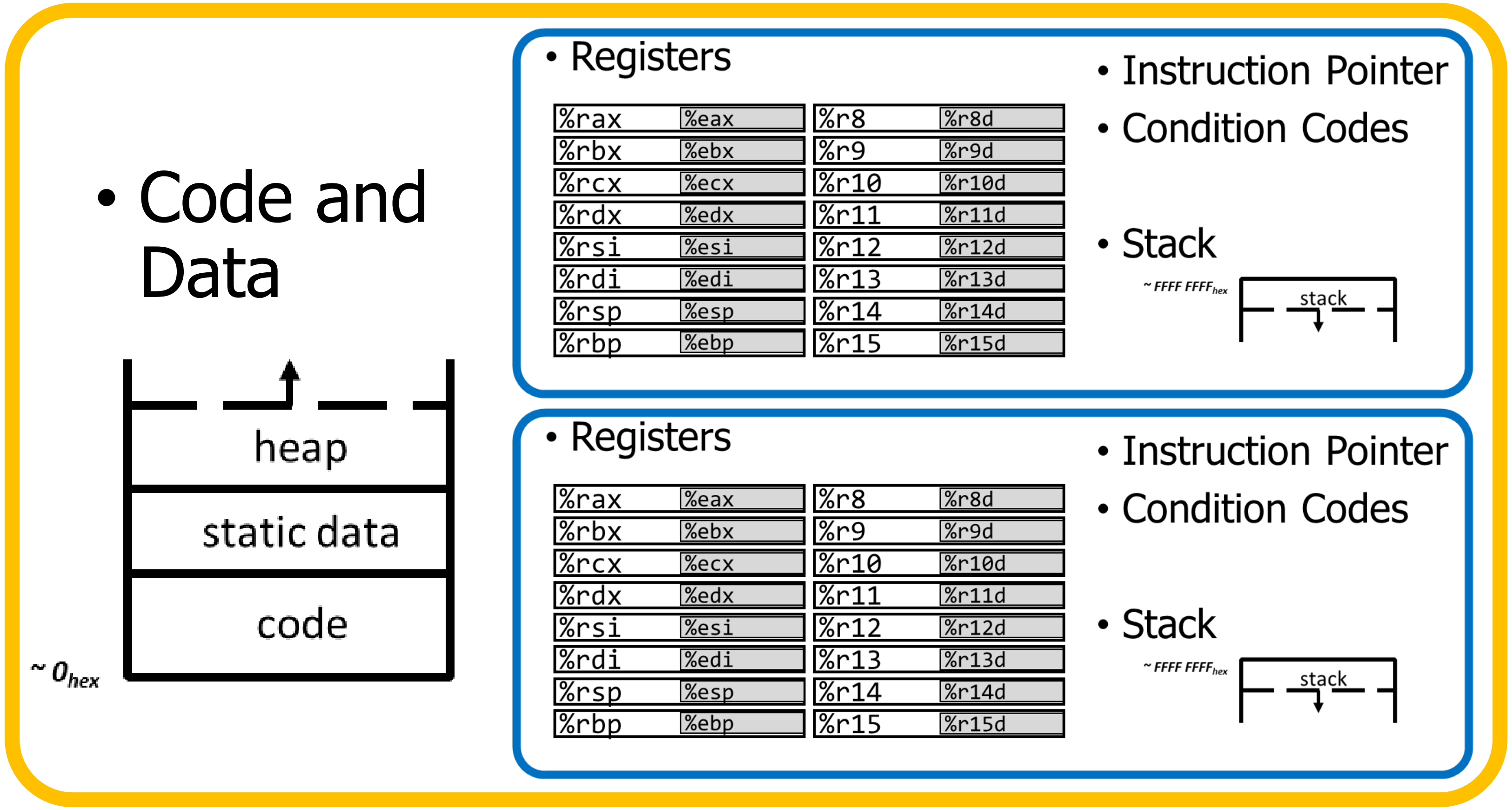
- Instruction Pointer
- Condition Codes

- Stack



Alternate view of a process

- A process could have multiple threads
  - Each with its own registers and stack



- Code and Data

- Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes
- Stack

- Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes
- Stack

Threads have separate:

- Instruction Pointer
- Registers
- Stack Memory
- Condition Codes

Threads share:

- Code
- Global variables

Thread use case: web browser

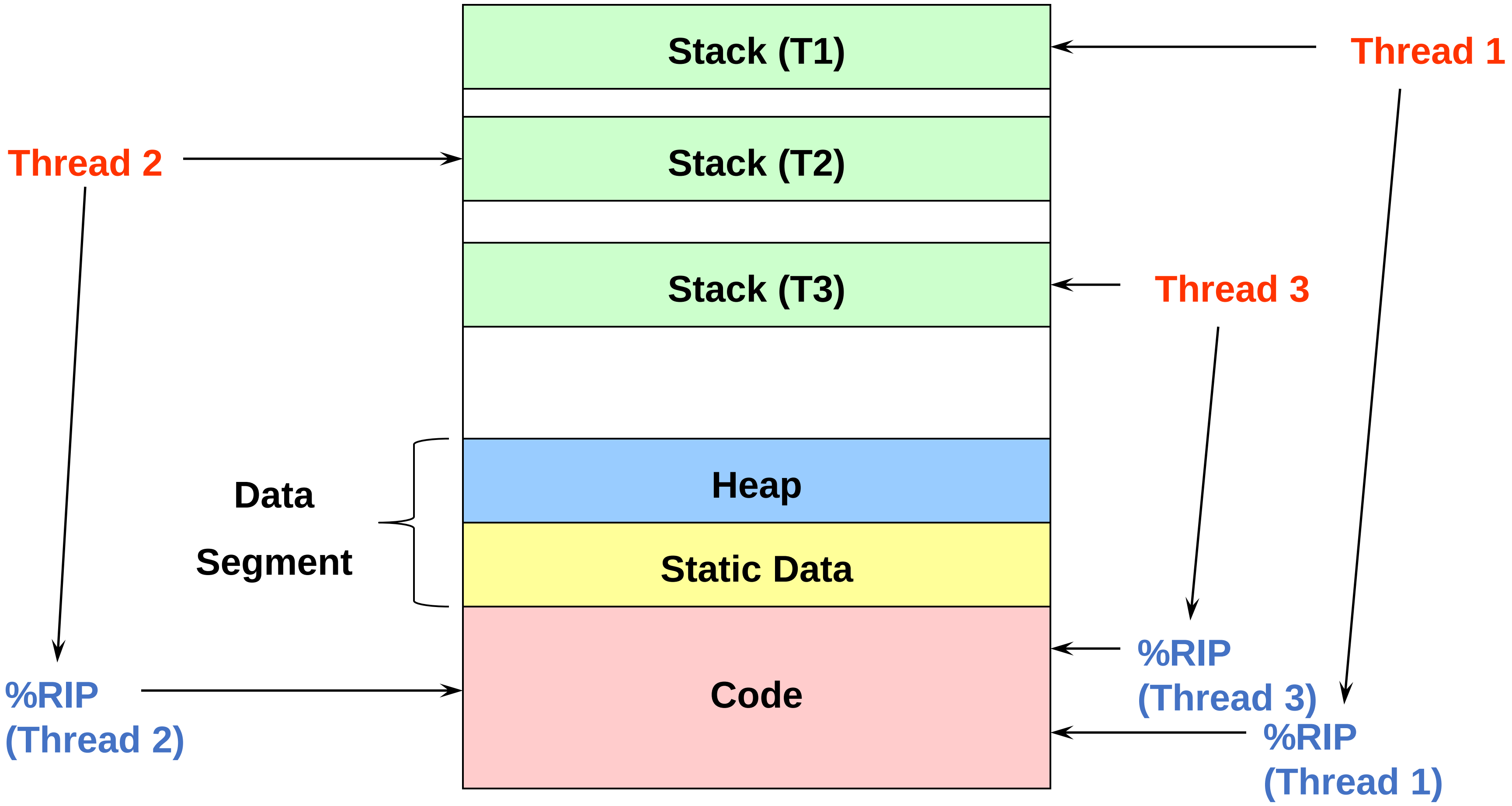
Let’s say you’re implementing a web browser:

You want a tab for each web page you open:

- Each tab is its own **thread**
- The same code loads each website (shared code section)
- The same global settings are shared by each tab (shared data section)
- Each tab does have separate state (separate stack and registers)

Disclaimer: Actually, modern browsers use separate processes for each tab for a variety of reasons including performance and security. But they used to use threads.

Process address space with multiple threads



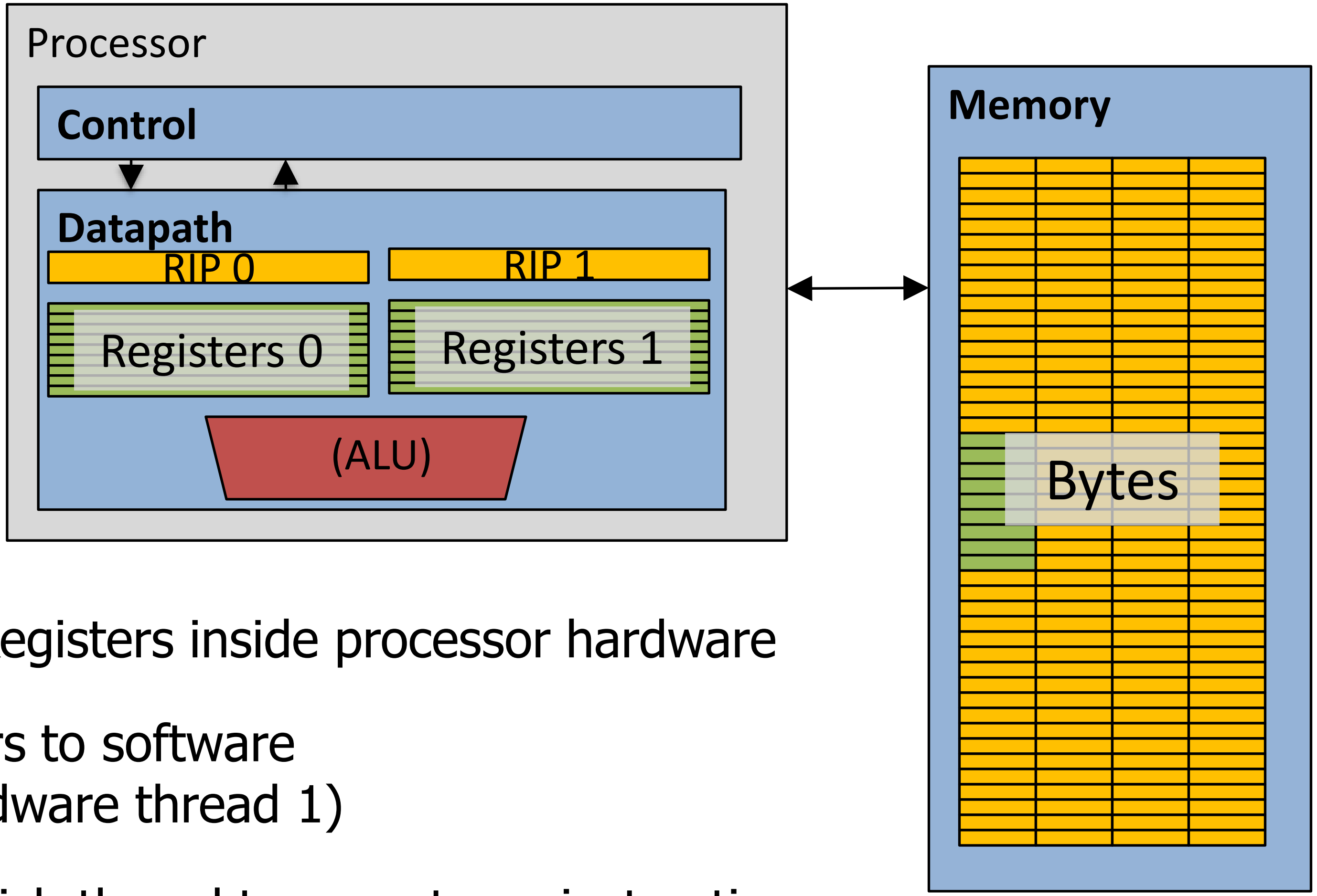
Multithreading processors

Basic idea: Processor resources are expensive and should not be left idle

Long memory latency to memory on cache miss?

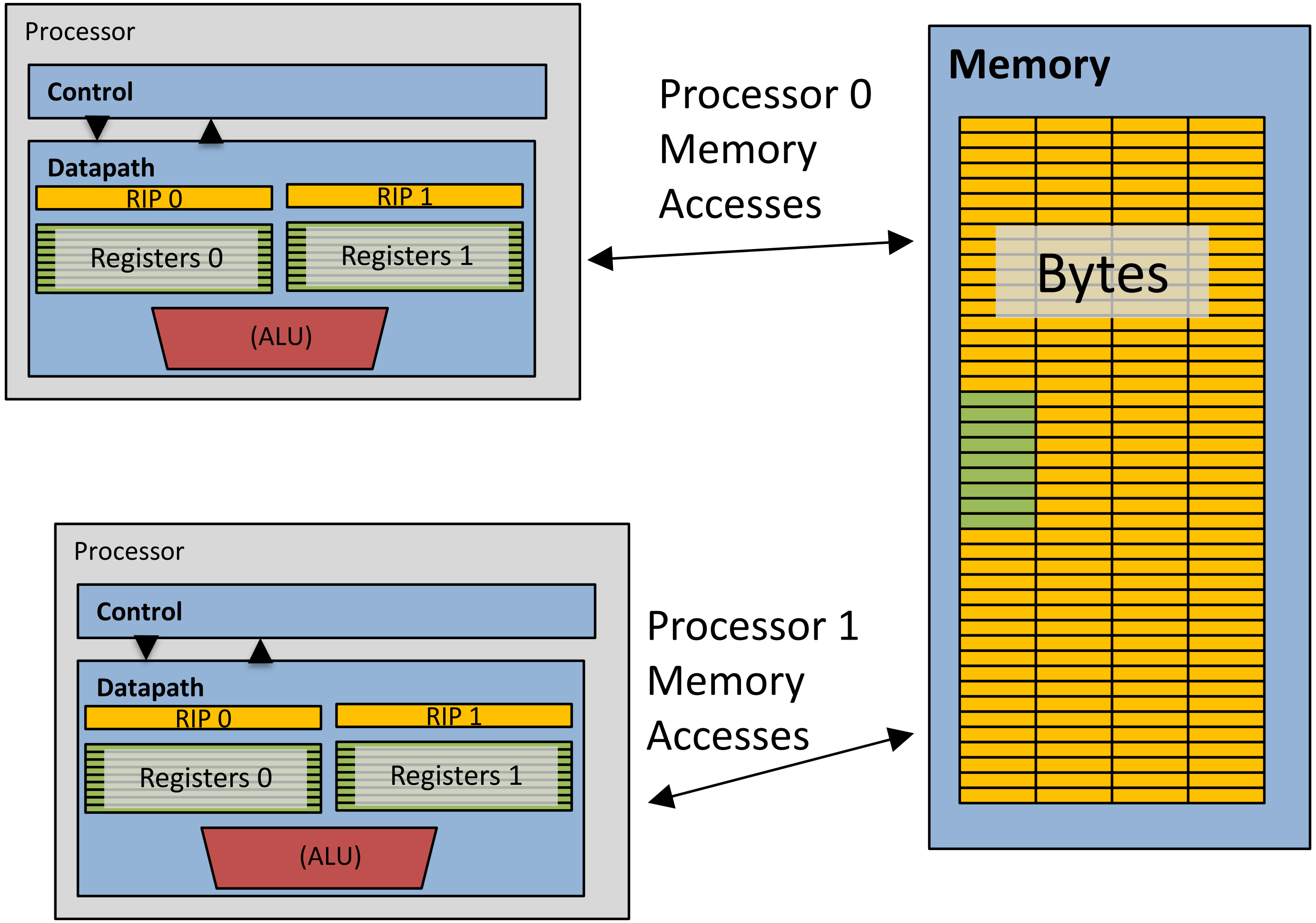
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Swapping between threads in a process takes less time than waiting for memory, so we get back to work sooner!

Multithreading processor



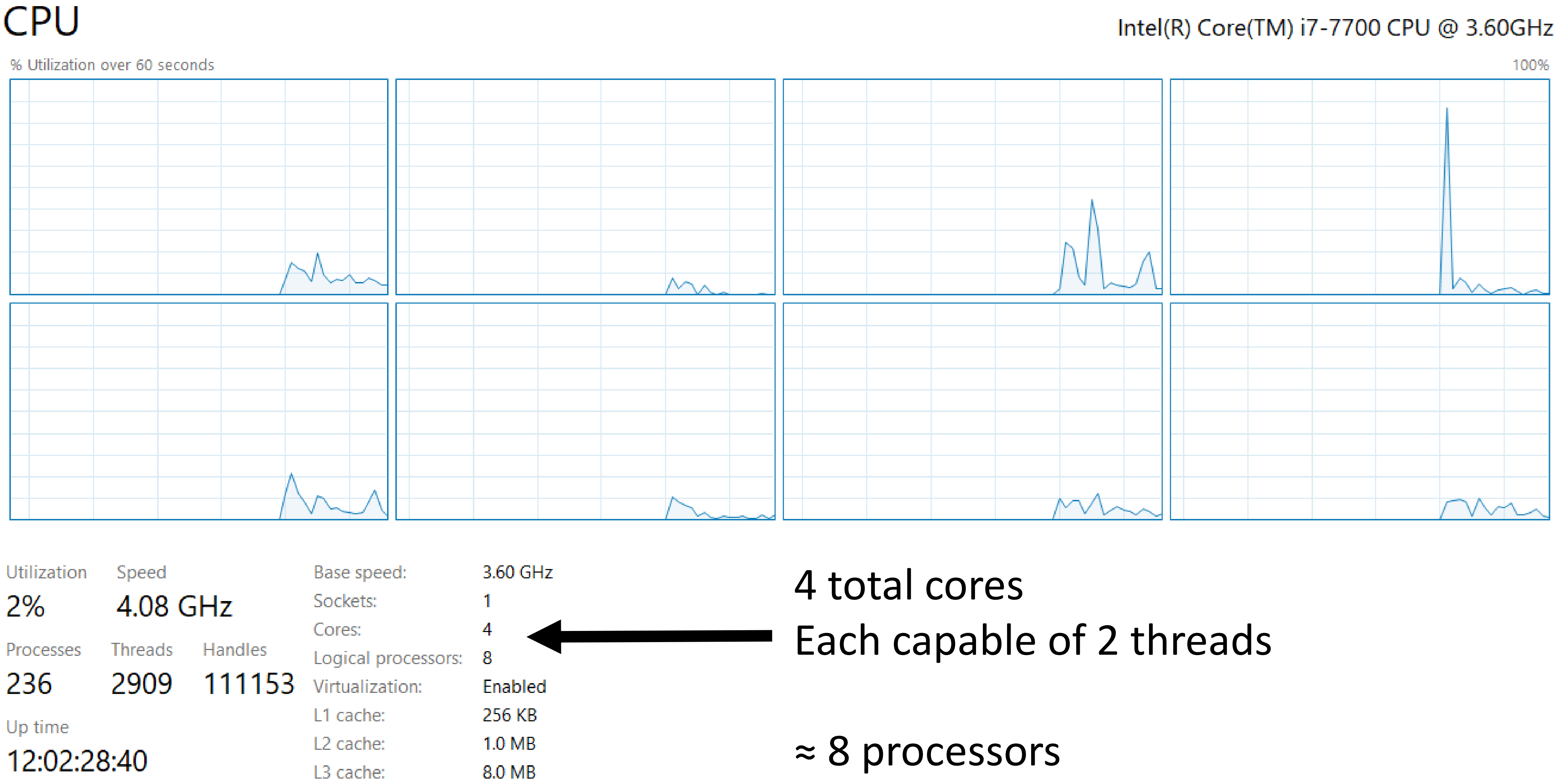
- Two copies of RIP and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next (concurrent, but NOT parallel)

Multithreading, multicore processors



- Combine capabilities of both designs
- Run two processes each with two threads
- Or run one process with four threads

Example: i7 processor



CPU

Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz

Utilization
2%

Speed
4.08 GHz

Processes
236

Threads
2909

Handles
111153

Up time
12:02:28:40

Base speed: 3.60 GHz
Sockets: 1
Cores: 4
Logical processors: 8
Virtualization: Enabled
L1 cache: 256 KB
L2 cache: 1.0 MB
L3 cache: 8.0 MB

4 total cores
Each capable of 2 threads

≈ 8 processors

Break + Open Question

- How many “cores” does a computer need?

Break + Open Question

- How many “cores” does a computer need?
  - Depends on the workload
  - Personal computer
    - ~2-10 processes running at once in the foreground
    - Plus ~100 in the background
  - Server
    - Could be serving thousands of requests simultaneously
    - Moore: 48 cores, Hanlon: 40 cores

Outline

- Need for Parallelism
- Processes and Threads
- **Concurrency Challenges**
- Using Threads

Challenges to concurrency

Concurrency is great! We can do so many things!!

But what’s the downside…?

1. How much speedup can we get from it?
2. How hard is it to write parallel programs?

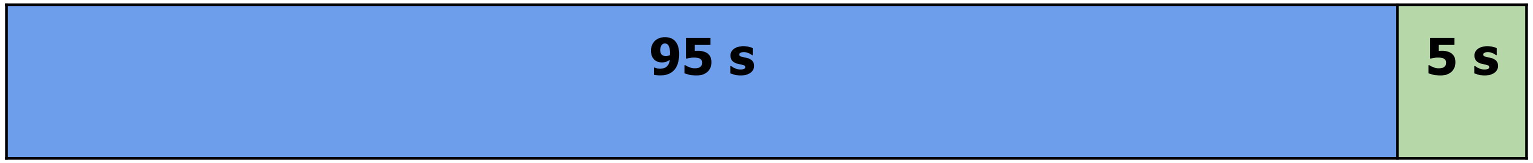
Challenges to concurrency

Concurrency is great! We can do so many things!!

But what’s the downside…?

1. **How much speedup can we get from it?**
2. How hard is it to write parallel programs?

Speedup Example



Imagine a program that takes 100 seconds to run

- 95 seconds in the blue part
- 5 seconds in the green part

We're going to speed up the green part and take a look at the net result

Speedup from improvements

$$\text{Speedup with Improvement} = \frac{\text{Execution time without improvement}}{\text{Execution time with improvement}}$$

5 s -> 2.5 s: Speedup = 100/97.5 = 1.026

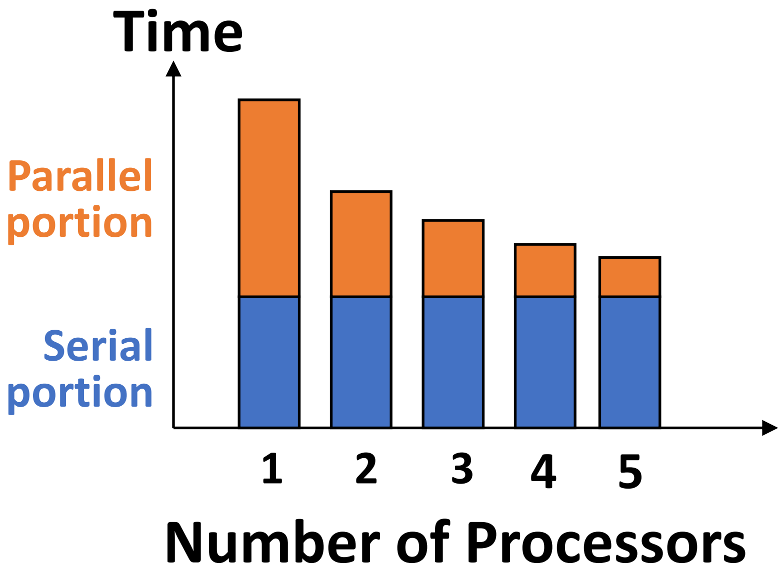
5 s -> 1 s: Speedup = 100/96 = 1.042

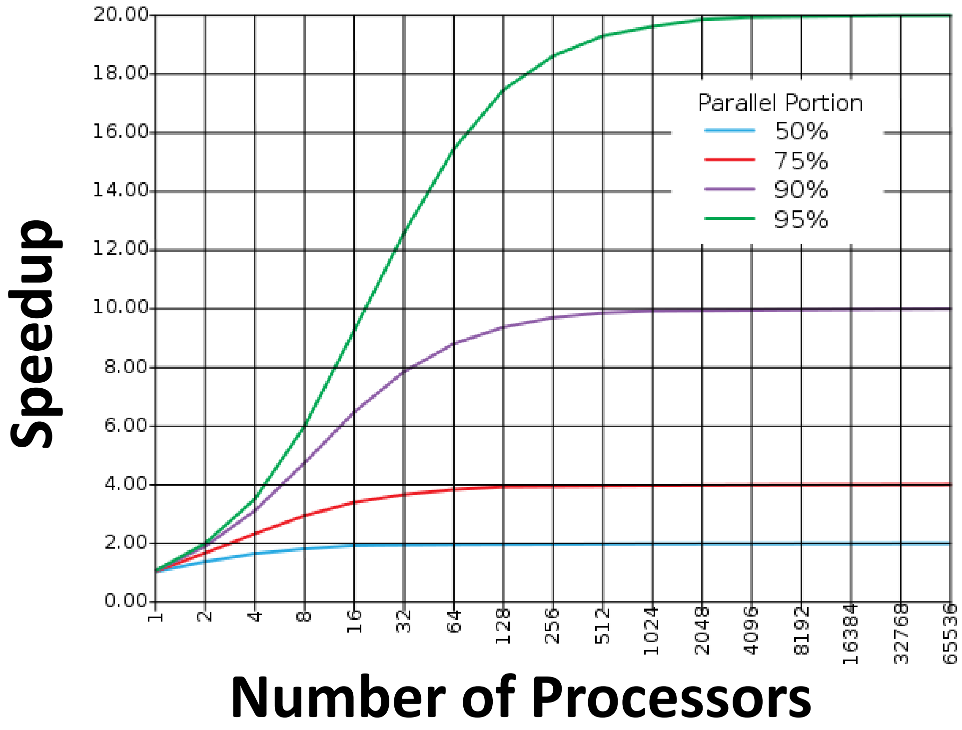
5 s -> 0.001s: Speedup = 100/95.001 = 1.053

The impact of a performance improvement is relative to the importance of the part being improved!

Amdahl’s Law (in pictures)

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program! 😭
  - And every program has at least *some* non-parallel parts





Challenges to concurrency

Concurrency is great! We can do so many things!!

But what’s the downside…?

1. How much speedup can we get from it?
2. **How hard is it to write parallel programs?**

Concurrency problem: data races

Consider two threads with a shared global variable: `int count = 0`

Thread 1:

```
void thread_fn(){
  count += 1;
}
```

Thread 2:

```
void thread_fn(){
  count += 1;
}
```

`count` could end up with a final value of 1 or 2. How?

Concurrency problem: data races

Consider two threads with a shared global variable: `int count = 0`

Thread 1:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

Thread 2:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

Assuming “count” is in memory location 0x8049a1c

`count` could end up with a final value of 1 or 2. How?

These instructions could be interleaved in any way.

Data race example – Count = 2

Before this code starts

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1	
Register	**Value**
`%eax`	???

Thread 2	
Register	**Value**
`%eax`	???

Memory	
Variable	**Value**
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 2

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	**Value**
`%eax`	1

Thread 2

Register	**Value**
`%eax`	???

Memory

Variable	**Value**
`count`	0

Assuming “count” is in memory location pointed to by `%edi`

Data race example – Count = 2

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1	
Register	**Value**
`%eax`	1

Thread 2	
Register	**Value**
`%eax`	???

Memory	
Variable	**Value**
`count`	1

Assuming “count” is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	**Value**
`%eax`	1

Thread 2

Register	**Value**
`%eax`	1

Memory

Variable	**Value**
`count`	1

Assuming “count” is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	**Value**
`%eax`	1

Thread 2

Register	**Value**
`%eax`	2

Memory

Variable	**Value**
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	**Value**
`%eax`	1

Thread 2

Register	**Value**
`%eax`	2

Memory

Variable	**Value**
`count`	2

Assuming “count” is in memory location pointed to by **`%edi`**

Theads do not have guaranteed ordering

BUT, there’s no guarantee that the instructions occur in that order!

Since the two threads are running in parallel, the instructions could be interleaved in any way
(both threads are really running simultaneously)

Data race example – Count = 1

Remember, each thread has its
own separate registers!

Before this code starts

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	???

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is
in memory location
pointed to by **`%edi`**

Data race example – Count = 1

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming “count” is in memory location pointed to by `%edi`

Data race example – Count = 1

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	0

Memory

Variable	Value
`count`	0

Assuming “count” is in memory location pointed to by `%edi`

Data race example – Count = 1

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	**Value**
`%eax`	0

Thread 2

Register	**Value**
`%eax`	1

Memory

Variable	**Value**
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	**Value**
`%eax`	0

Thread 2

Register	**Value**
`%eax`	1

Memory

Variable	**Value**
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	**Value**
`%eax`	1

Thread 2

Register	**Value**
`%eax`	1

Memory

Variable	**Value**
`count`	1

Assuming “count” is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	**Value**
`%eax`	1

Thread 2

Register	**Value**
`%eax`	1

Memory

Variable	**Value**
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race comparison

Assuming "count" is in memory location pointed to by **`%edi`**

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Final value of count: 2

Thread 1	**Thread 2**
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Final value of count: 1

Data race explanation

- Thread scheduling is **non-deterministic**
  - There is no guarantee that any thread will go first or last or not be interrupted at any point

- If different threads write to the **same** variable
  - The final value of the variable is also non-deterministic
  - This is a *data race*

- Avoid incorrect results by:
  1. Not writing to the same memory address!!

     OR

  2. Synchronizing reading and writing to get deterministic behavior

Data race explanation

- Thread scheduling is **non-deterministic**
  - There is no guarantee that any thread will go first or last or not be interrupted at any point

- If different threads write to the **same** variable
  - The final value of the variable is also non-deterministic
  - This is a *data race*

- Avoid incorrect results by:
  1. Not writing to the same memory address!! ←

     OR

  2. Synchronizing reading and writing to get deterministic behavior

We'll pick this one for CS213

CS343 explores this in depth

Avoiding shared memory data races

- Ensure that no two threads write to the same memory address

- Multiple threads reading from the same memory address is fine
  - As long as no thread writes to that memory

- Where do you put results then? Simple solution:
  - Make an array with a slot for each thread
  - Each thread only writes to their own slot in the array

  - After all threads are done, main thread iterates the array and determines the final result

Question + Break

Consider three threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 1;
}
```

Thread 2:

```
void main(){
  count -= 1;
}
```

Thread 3:

```
void main(){
  count += 2;
}
```

What are the possible values of count?

Question + Break

Consider three threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 1;
}
```

Thread 2:

```
void main(){
  count -= 1;
}
```

Thread 3:

```
void main(){
  count += 2;
}
```

What are the possible values of count? **-1, 0, 1, 2, 3**

How are you supposed to reason about this?!
Need mechanisms for sharing memory.

Outline

- Need for Parallelism

- Processes and Threads

- Concurrency Challenges

- **Using Threads**

Thread operations

- **`Create threads`**
  - ***Shares*** all memory with all threads of the process.
  - Scheduled independently of parent

- **`Join thread`**
  - Waits for a particular thread to finish
  - Can’t continue computation until all threads finish

- That’s it! Don’t really need anything else (for this class)
  - Library also includes synchronization primitives to solve data races

- Can communicate between threads by reading/writing (shared) global variables
  - But we’re only going to ***read*** from shared variables for safety
  - We’ll write to separate memory locations

POSIX Threads Library: pthreads

- https://man7.org/linux/man-pages/man7/pthreads.7.html

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr,
void* (*start_routine)(void*), void* arg);
```

  - Thread is created executing *start_routine* with *arg* as its sole argument.
  - Returning from the start routine exits the thread

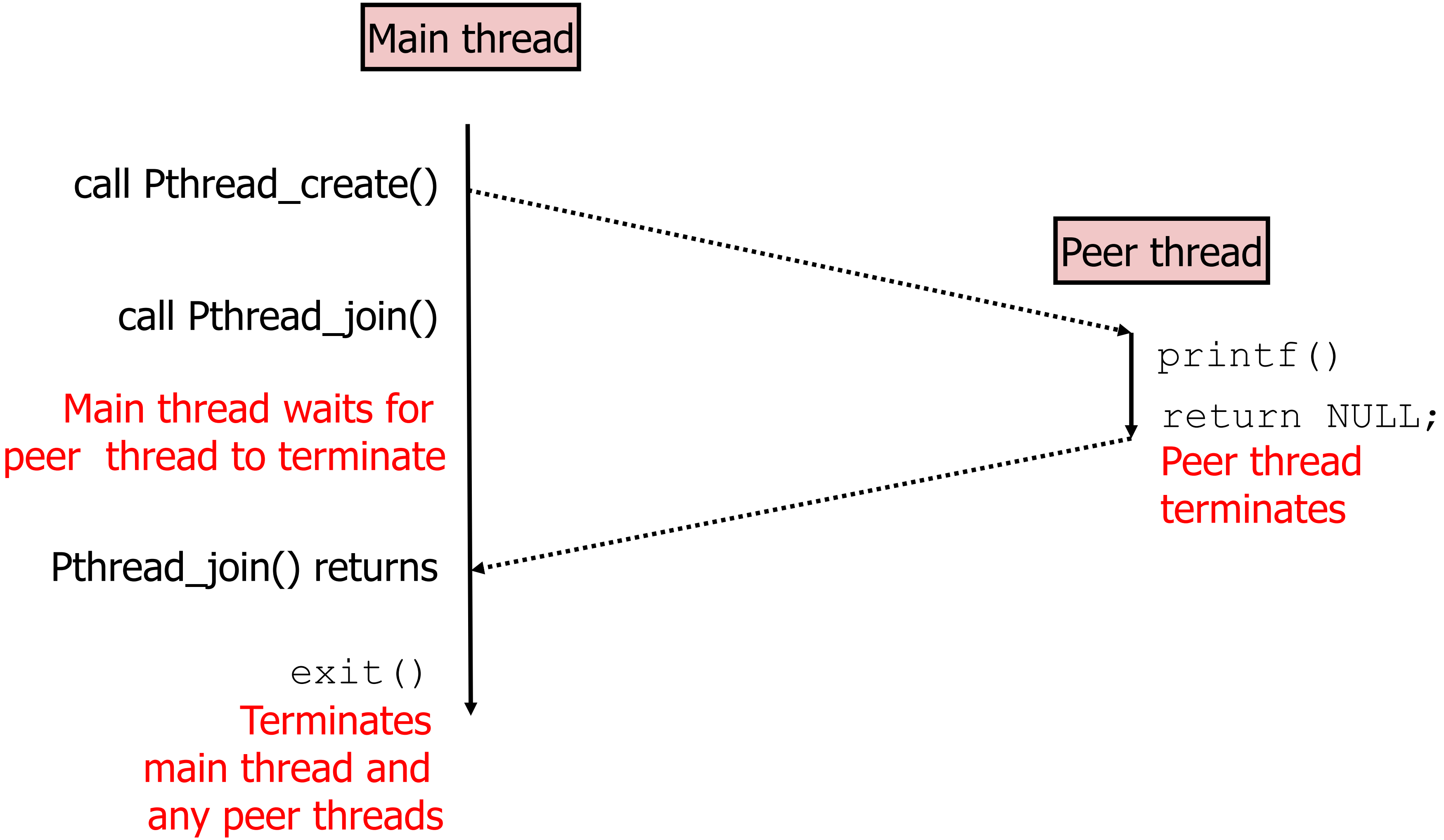
```
void pthread_exit(void* value_ptr);
```

  - Terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_join(pthread_t thread, void** value_ptr);
```

  - Suspends execution of the calling thread until the target *thread* terminates.
  - On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

Basic thread example



Main thread

call Pthread_create()

call Pthread_join()

Main thread waits for peer thread to terminate

Pthread_join() returns

`exit()`

Terminates main thread and any peer threads

Peer thread

`printf()`

`return NULL;`

Peer thread terminates

Example: parallel sum of array

```
double array[LEN] = {1, 2, 3, …, LEN};


// determine result sequentially
double sequential_sum = 0;

for (int i=0; i<LEN; i++) {

  sequential_sum += array[i];

}
```

Example: parallel sum of array

```
double array[LEN] = {1, 2, 3, …, LEN};
```

Parallelization Plan

1. Create num_threads different threads

2. Threads create "partial" sums for their portion of the work
    - Each thread does **`(LEN/num_threads)`** work
    - Create an array for results with one slot per thread

3. Wait until done, then sum the partial results
    - Main thread calls `join()` to wait for each thread to complete
    - Main thread adds up results

Example: parallel sum of array

1. Create num_threads different threads

Main Thread

```
pthread_t tid[num_threads];
for (long i=0; i<num_threads; i++) {
  pthread_create(&(tid[i]), NULL, worker, (void*)i);
}
```

- Arguments to pthread_create
  - thread_handle, attributes, thread_function, function_argument

Example: parallel sum of array

2. Threads create “partial” sums for their portion of the work

Worker Threads

```
void* worker(void* arg) {
  long i = (long)arg;
  int mystart = i * (LEN/num_threads);
  int myend = (i+1) * (LEN/num_threads);
  partial_sum[i] = 0;
  for (int j=mystart; j<myend; j++) {
    partial_sum[i] += array[j];
  }
  pthread_exit(NULL); // Thread work is complete
}
```

Decide which portion of work this thread should do

Example: parallel sum of array

2. Threads create “partial” sums for their portion of the work

Worker Threads

```
void* worker(void* arg) {
  long i = (long)arg;
  int mystart = i * (LEN/num_threads);
  int myend = (i+1) * (LEN/num_threads);
  partial_sum[i] = 0;
  for (int j=mystart; j<myend; j++) {
    partial_sum[i] += array[j];
  }
  pthread_exit(NULL); // Thread work is complete
}
```

Do the work

Puts result in its own slot in the `partial_sum` array (avoids data races)

Example: parallel sum of array

3. Wait until done, then sum the partial results

Main Thread

```
for (int j=0; j<num_threads; j++) {
  pthread_join(tid[j], NULL); // second argument is return result
}

double parallel_sum = 0;
for (int k=0; k<num_threads; k++) {
  parallel_sum += partial_sum[k];
}
```

Trying this out for yourself

- See SETI Lab for example code you can run yourself

- We just went through a slightly simplified version of `parallel-sum-ex.c`

Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum:   19999999900000000 ( 878576632 cycles)
Parallel sum:     0 (44 cycles)
```

Array of 200 million length

No threads created

Only the sequential version is run. Takes about ~878 million cycles to run

Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum:   19999999900000000 ( 878576632 cycles)
Parallel sum:     0 (44 cycles)

$ ./parallel-sum-ex 1 1 200000000
Sequential sum:   19999999900000000 ( 902438479 cycles)
Parallel sum:     19999999900000000 (1169222739 cycles)
```

Array of 200 million length

1 to thread created. No parallelism for a speedup. Actually, it takes LONGER to run.

Starting threads takes time! Need to make sure they’re doing enough work to be worth it.

Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum:   19999999900000000 ( 878576632 cycles)
Parallel sum:     0 (44 cycles)

$ ./parallel-sum-ex 1 1 200000000
Sequential sum:   19999999900000000 ( 902438479 cycles)
Parallel sum:     19999999900000000 (1169222739 cycles)

$ ./parallel-sum-ex 8 8 200000000
Sequential sum:   19999999900000000 ( 888810917 cycles)
Parallel sum:     19999999900000000 (1033659530 cycles)
```

Array of 200 million length

8 threads actually has some parallelism

Starts doing better than one thread but needs more parallelism for a big win.

Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum:   19999999900000000 ( 878576632 cycles)
Parallel sum:     0 (44 cycles)

$ ./parallel-sum-ex 1 1 200000000
Sequential sum:   19999999900000000 ( 902438479 cycles)
Parallel sum:     19999999900000000 (1169222739 cycles)

$ ./parallel-sum-ex 8 8 200000000
Sequential sum:   19999999900000000 ( 888810917 cycles)
Parallel sum:     19999999900000000 (1033659530 cycles)

$ ./parallel-sum-ex 16 16 200000000
Sequential sum:   19999999900000000 ( 895258209 cycles)
Parallel sum:     19999999900000000 ( 693511997 cycles)
```

Array of 200 million length

16 threads starts to win!

I don’t actually have that many cores, but the system is swapping threads whenever memory reads stall to improve performance

Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum:   19999999900000000 ( 878576632 cycles)
Parallel sum:     0 (44 cycles)

$ ./parallel-sum-ex 1 1 200000000
Sequential sum:   19999999900000000 ( 902438479 cycles)
Parallel sum:     19999999900000000 (1169222739 cycles)

$ ./parallel-sum-ex 8 8 200000000
Sequential sum:   19999999900000000 ( 888810917 cycles)
Parallel sum:     19999999900000000 (1033659530 cycles)

$ ./parallel-sum-ex 16 16 200000000
Sequential sum:   19999999900000000 ( 895258209 cycles)
Parallel sum:     19999999900000000 ( 693511997 cycles)

$ ./parallel-sum-ex 32 32 200000000
Sequential sum:   19999999900000000 ( 886174224 cycles)
Parallel sum:     19999999900000000 ( 609774231 cycles)

$ ./parallel-sum-ex 64 64 200000000
Sequential sum:   19999999900000000 ( 898098616 cycles)
Parallel sum:     19999999900000000 ( 426420305 cycles)
```

Array of 200 million length

32 and 64 threads are really cruising

Down to half the time for the computation

Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum:   19999999900000000 ( 878576632 cycles)
Parallel sum:     0 (44 cycles)

$ ./parallel-sum-ex 1 1 200000000
Sequential sum:   19999999900000000 ( 902438479 cycles)
Parallel sum:     19999999900000000 (1169222739 cycles)

$ ./parallel-sum-ex 8 8 200000000
Sequential sum:   19999999900000000 ( 888810917 cycles)
Parallel sum:     19999999900000000 (1033659530 cycles)

$ ./parallel-sum-ex 16 16 200000000
Sequential sum:   19999999900000000 ( 895258209 cycles)
Parallel sum:     19999999900000000 ( 693511997 cycles)

$ ./parallel-sum-ex 32 32 200000000
Sequential sum:   19999999900000000 ( 886174224 cycles)
Parallel sum:     19999999900000000 ( 609774231 cycles)

$ ./parallel-sum-ex 64 64 200000000
Sequential sum:   19999999900000000 ( 898098616 cycles)
Parallel sum:     19999999900000000 ( 426420305 cycles)

$ ./parallel-sum-ex 128 128 200000000
Sequential sum:   19999999900000000 ( 891919128 cycles)
Parallel sum:     19999999900000000 ( 493951974 cycles)
```

Array of 200 million length

128 threads is basically the same as 64 threads

Further parallelism isn’t helping very much. (technically worse than 64, but it’s within error bounds on timing)

Outline

- Need for Parallelism
- Processes and Threads
- Concurrency Challenges
- Using Threads

Outline

- Bonus: SIMD Instructions

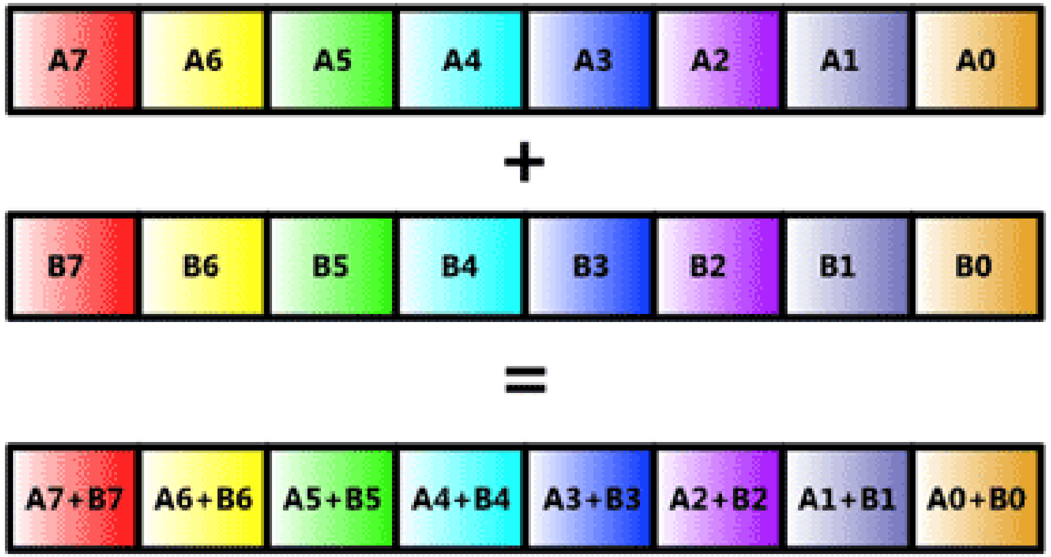
SIMD Architectures

- *Data-Level Parallelism (DLP):* Executing one operation on multiple data streams
  - SIMD: Single Instruction Multiple Data
- **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

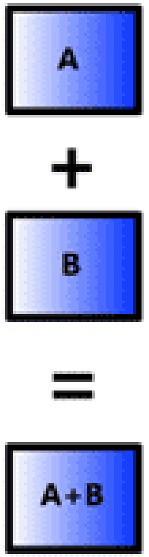
$$y[i] := c[i] \times x[i],\ 0 \le i < n$$

- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well

SIMD Mode

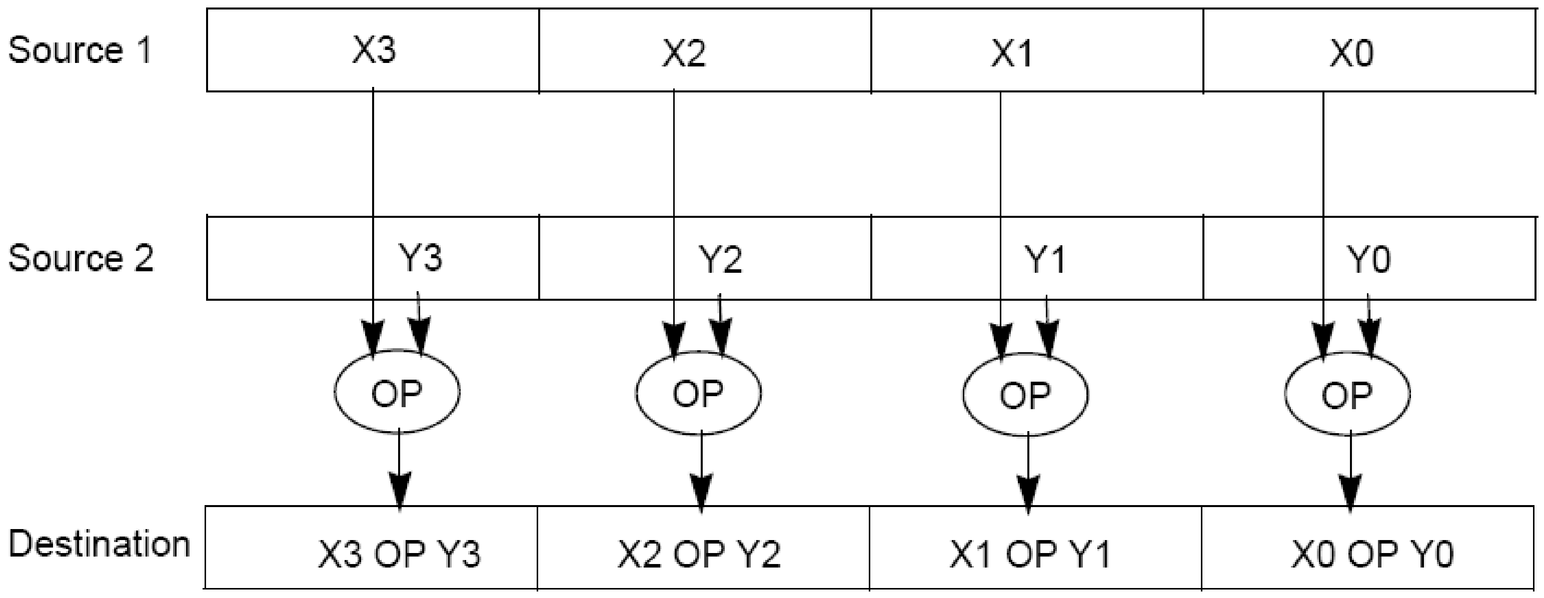


Scalar Mode



Example SIMD Instructions

- To improve performance, Intel's SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - *SSE (Streaming SIMD Extension, Pentium III and beyond)*



Example: SIMD Array Processing

```
for each f in array
    f = sqrt(f)
```

pseudocode

```
for each f in array {
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

SISD

```
for each 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

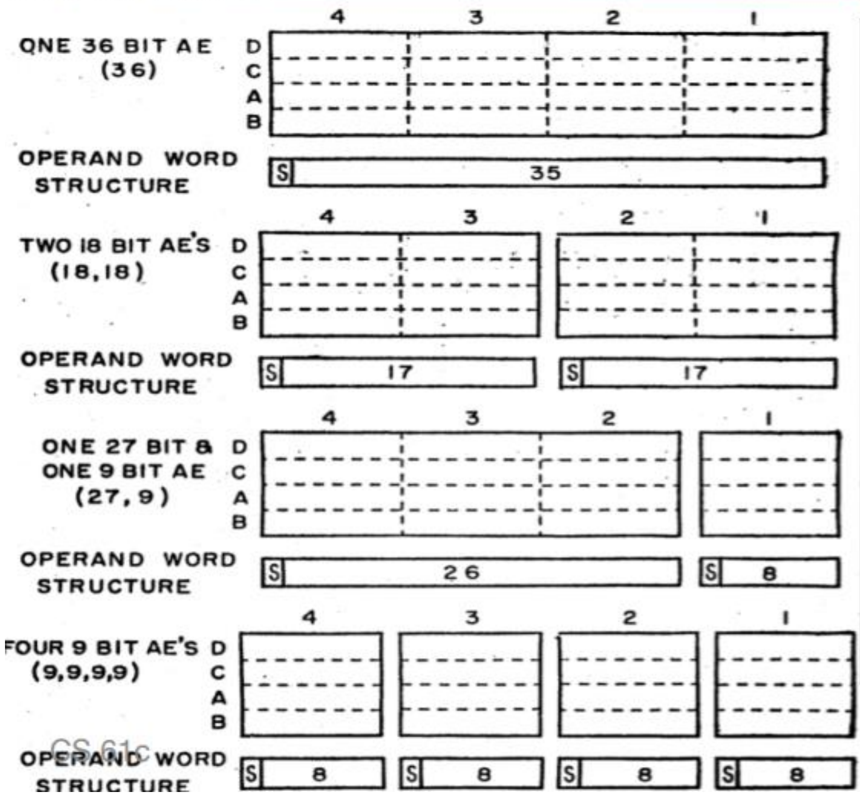
SIMD

SIMD in the Real World

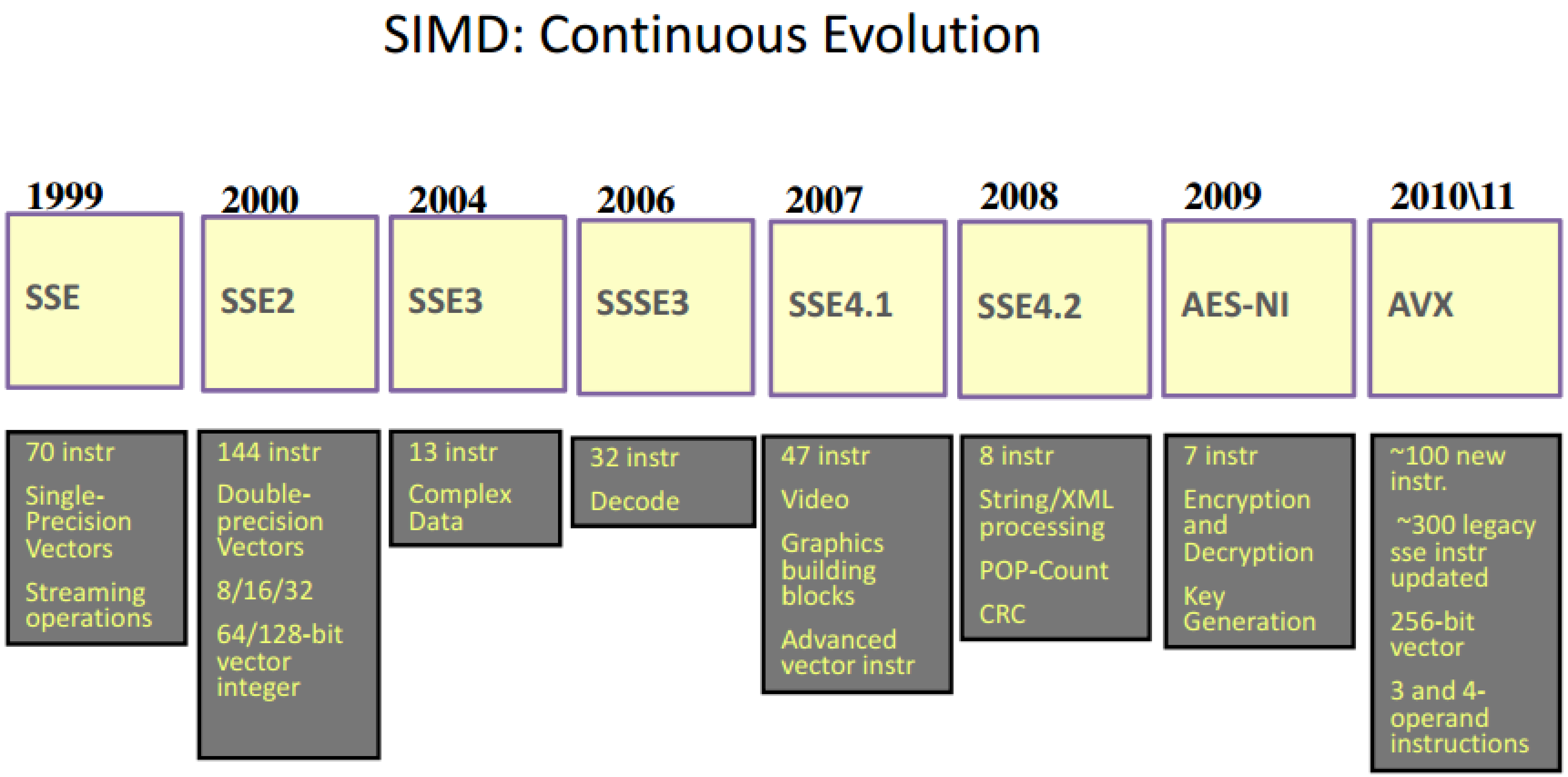
- Today’s compilers can generate SIMD code!
  - But in some cases we get better results by hand

- Intel’s x86 implements many SIMD instructions
  - Which have the benefit of being usable on lab machines
  - (and most of our own personal computers)

First SIMD Extensions:
MIT Lincoln Labs TX-2, 1957



Intel SIMD has been continuously extended



SIMD: Continuous Evolution

1999	2000	2004	2006	2007	2008	2009	2010\11
SSE	**SSE2**	**SSE3**	**SSSE3**	**SSE4.1**	**SSE4.2**	**AES-NI**	**AVX**
70 instr Single-Precision Vectors Streaming operations	144 instr Double-precision Vectors 8/16/32 64/128-bit vector integer	13 instr Complex Data	32 instr Decode	47 instr Video Graphics building blocks Advanced vector instr	8 instr String/XML processing POP-Count CRC	7 instr Encryption and Decryption Key Generation	~100 new instr. ~300 legacy sse instr updated 256-bit vector 3 and 4-operand instructions

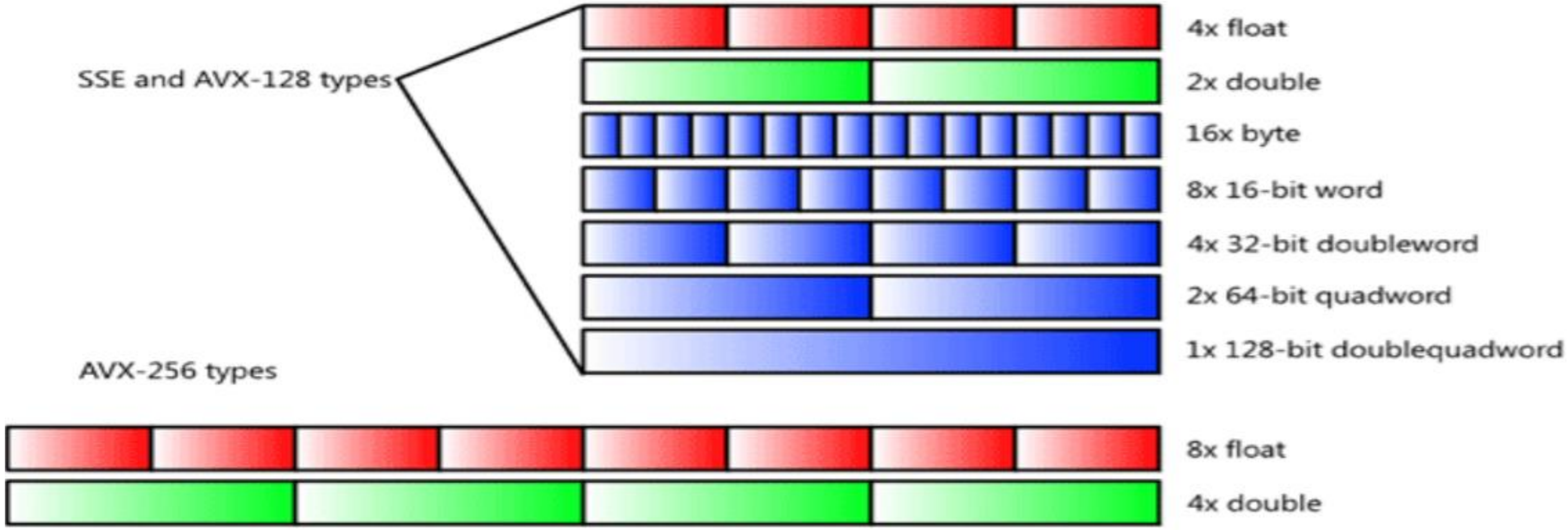
And it has increased in size a lot

Intel **A**dvanced **V**ector e**X**tensions

AVX Registers getting wider, instruction set getting richer

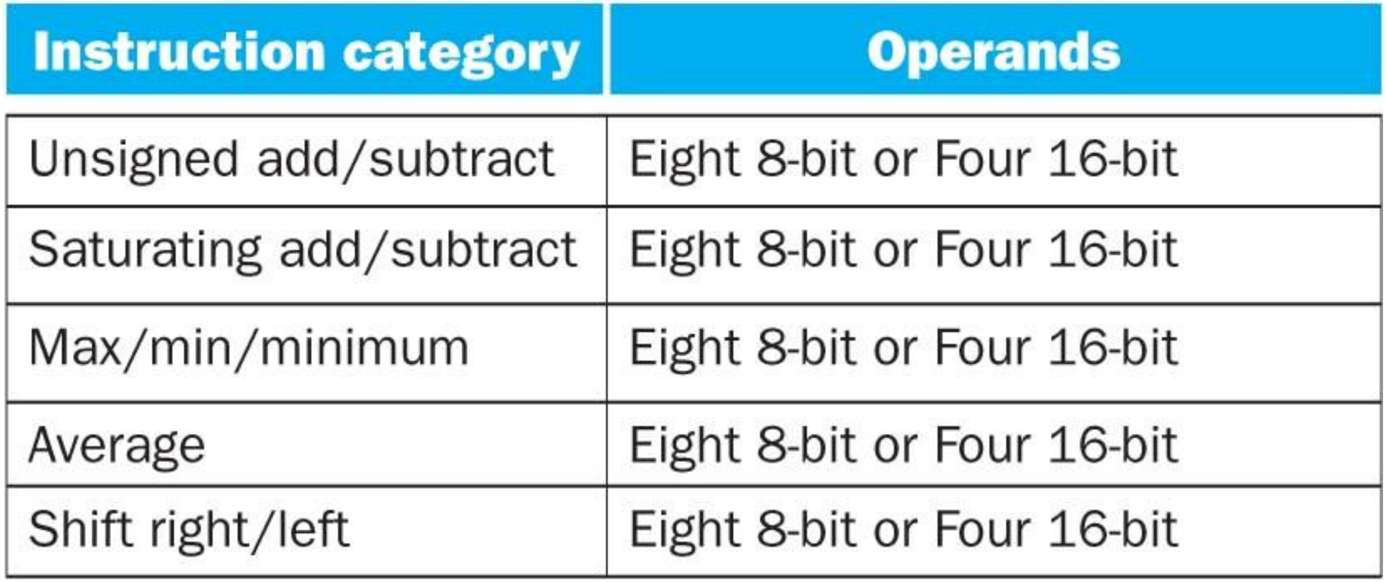
2011	2012	2013	2014	2015	Future
87 GFLOPS	**185 GFLOPS**	**~225 GFLOPS**	**~500 GFLOPS**	**tbd GFLOPS**	**tbd GFLOPS**
Westmere	**Sandy Bridge**	**Ivy Bridge**	**Haswell**	**Broadwell**	**Skylake**
32 nm SSE 4.2 DDR3 PCIe2	32 nm AVX **(256 bit registers)** DDR3 PCIe3	22 nm	22 nm AVX2 **(new instructions)** DDR4 PCIe3	14 nm	14 nm AVX 3.2 **(512 bit registers)** DDR4 PCIe4

Intel SIMD Data Types



(Now also AVX-512 available (but not on Hive): 16x float and 8x double)

SSE Instruction Categories for Multimedia Support



Instruction category	Operands
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

- SSE-2+ supports wider data types to allow 16 × 8-bit and 8 × 16-bit operands

How do we use these SIMD instructions?

- Intrinsics:
  - “function calls” that actually just execute an assembly instruction

Example:

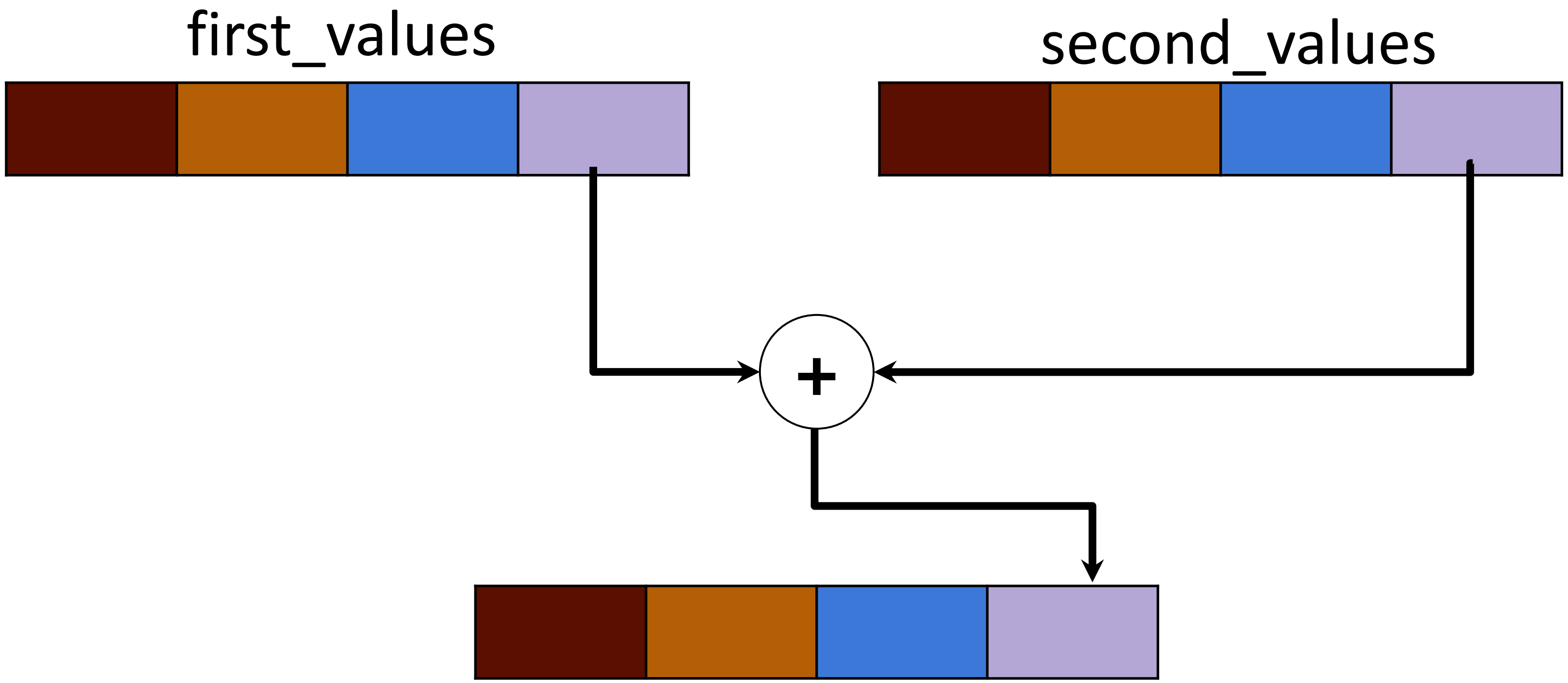
```
_mm_add_epi32(first_values, second_values);
```

WHAT????

`_mm_add_epi32(first_values, second_values)`

Multi**M**edia extension
(They all start with this)

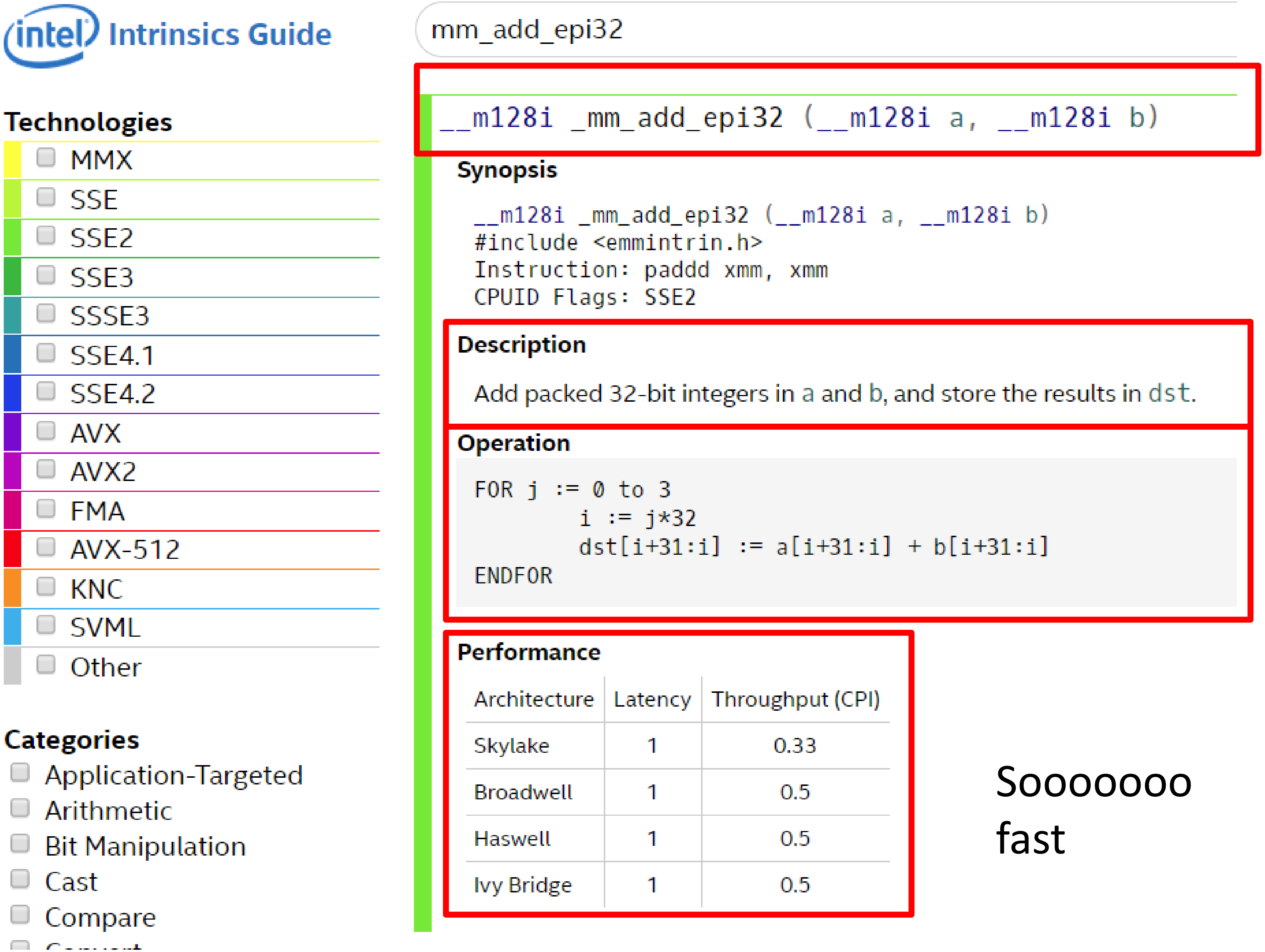
Arguments are
Extended **P**acked **I**ntegers,
each **32**-bits in size
(signed)



first_values

second_values

+



Intrinsics Guide

mm_add_epi32

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
```

Synopsis

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddd xmm, xmm
CPUID Flags: SSE2
```

Description

Add packed 32-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 3
        i := j*32
        dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
```

Performance

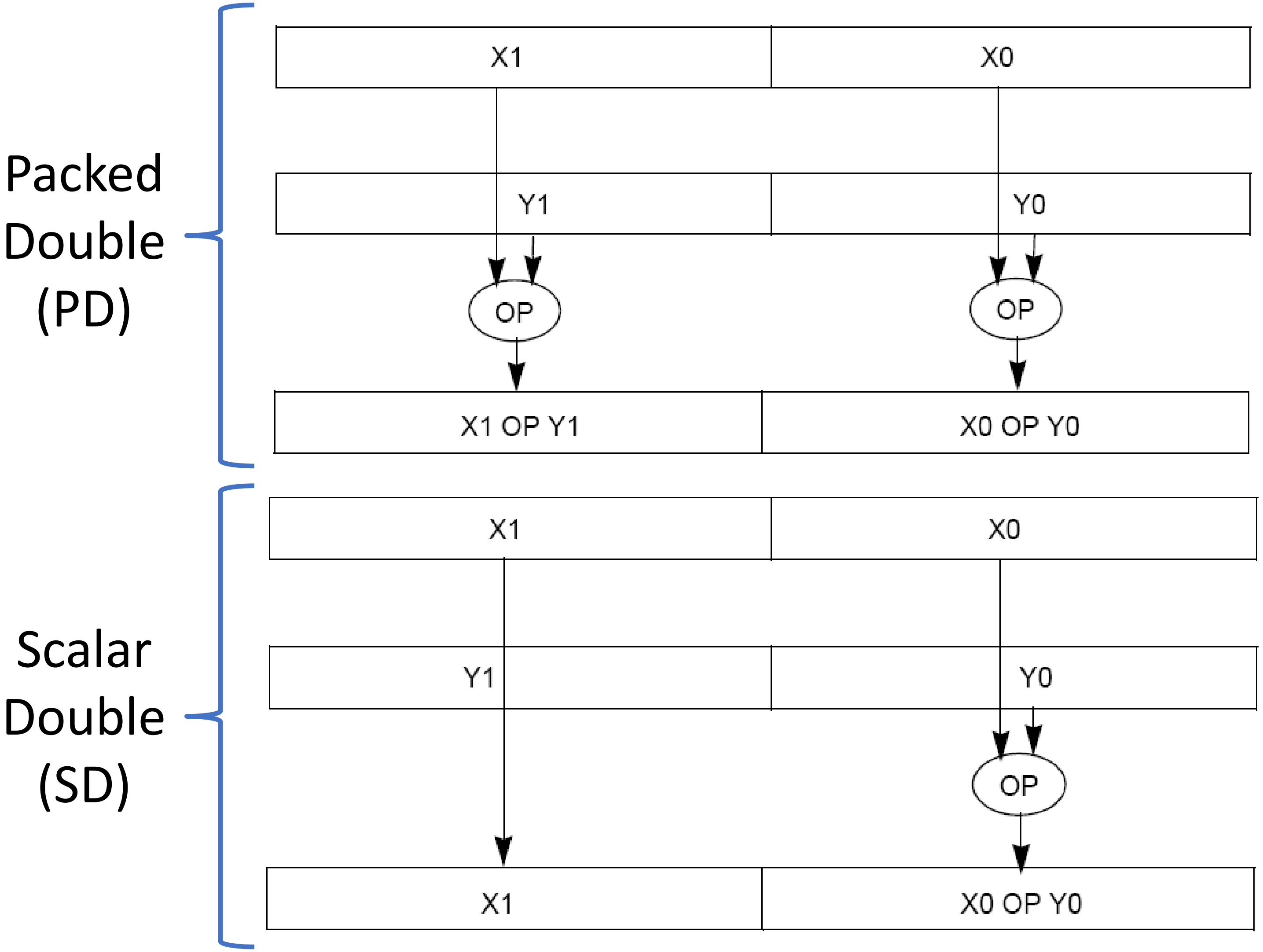
Architecture	Latency	Throughput (CPI)
Skylake	1	0.33
Broadwell	1	0.5
Haswell	1	0.5
Ivy Bridge	1	0.5

Sooooooo
fast

```
int add_no_SSE(int size, int *first_array, int *second_array) {
    for (int i = 0; i < size; ++i) {
        first_array[i] += second_array[i];
    }
}

int add_SSE(int size, int *first_array, int *second_array) {
    for (int i=0; i + 4 <= size; i+=4) { // only works if (size%4) == 0
        // load 128-bit chunks of each array
        __m128i first_values = _mm_loadu_si128((__m128i*) &first_array[i]);
        __m128i second_values = _mm_loadu_si128((__m128i*) &second_array[i]);

        // add each pair of 32-bit integers in the 128-bit chunks
        first_values = _mm_add_epi32(first_values, second_values);

        // store 128-bit chunk to first array
        _mm_storeu_si128((__m128i*) &first_array[i], first_values);
    }
        ...
}
```

https://www.cs.virginia.edu/~cr4bd/3330/F2017/simdref.html

You can do this with floating point numbers too!



Example: Reversing an array in 7 steps (animated)