

# Lecture 15

# Concurrency

CS213 – Intro to Computer Systems  
Branden Ghen a – Winter 2022

Slides adapted from:

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

# Administrivia

- Attack Lab due today
- Homework 4 will be released tonight
  - Due next week Thursday
- SETI Lab releases on Friday. Due in two weeks (last Thursday of class)
  - Today's lecture has most of the information you need for it
  - Tuesday's lecture will add a few more details about optimization

# 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 time.

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 time.

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

## Transistor count

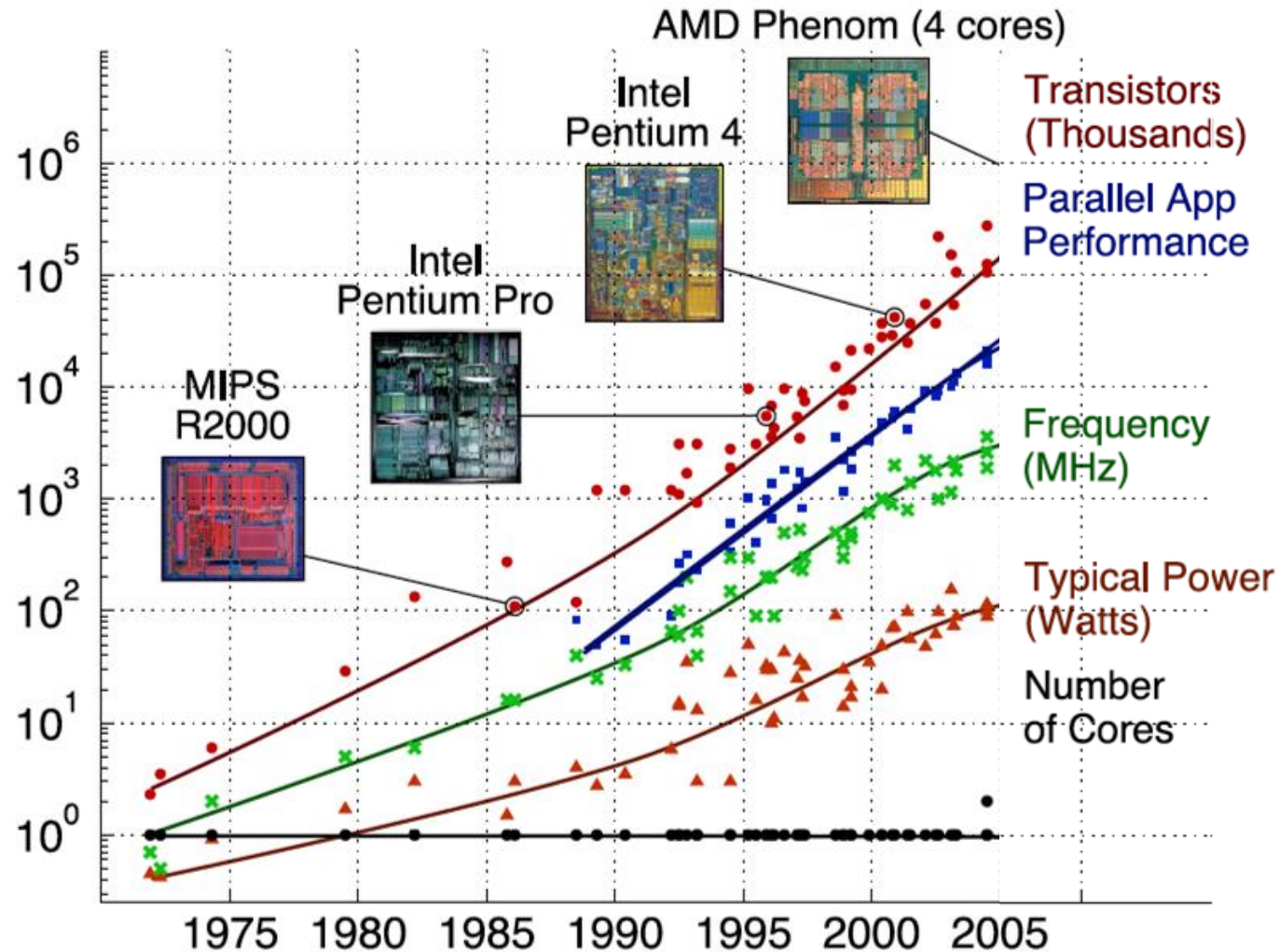


# Transistors are getting exponentially smaller!

How small? Today: 7nm!  
< 1/2 the size of most viruses!

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

# Processors kept getting faster too



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



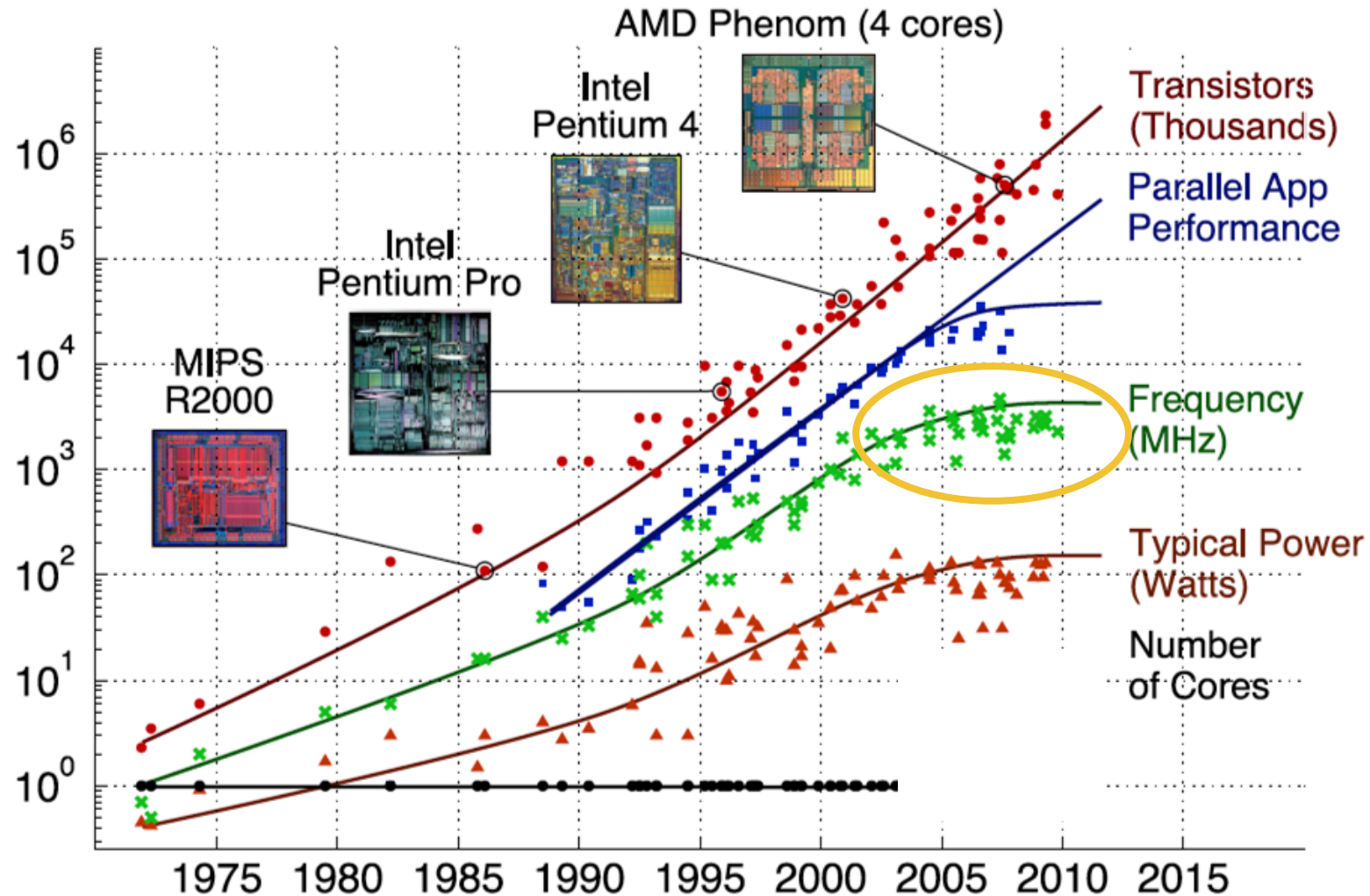
# Power is a major limiting factor on speed

- 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**

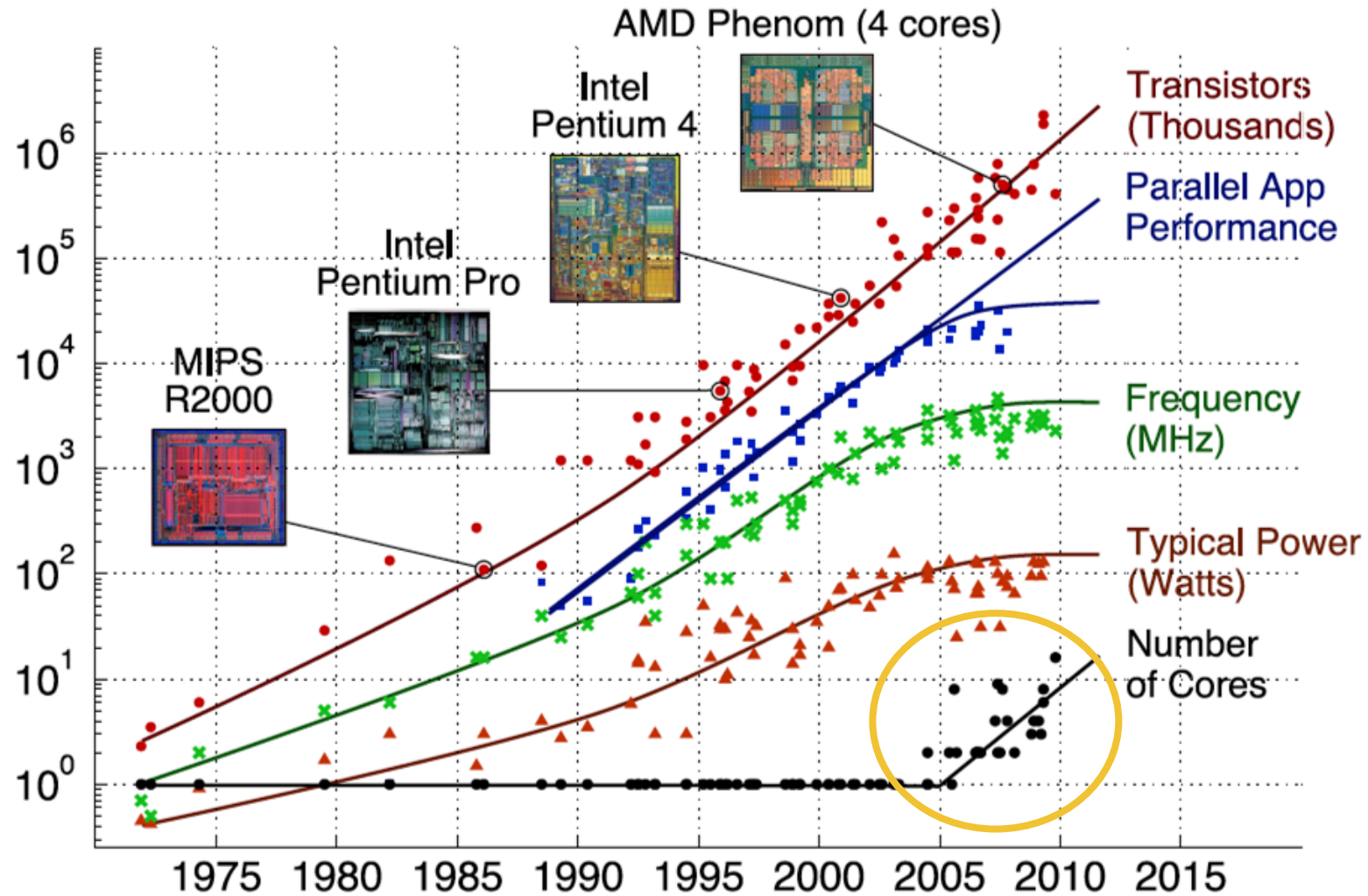
# So... now what?

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

# Parallelism Analogy

- I want to peel 100 potatoes as fast as possible:

- I can learn to peel potatoes faster

OR

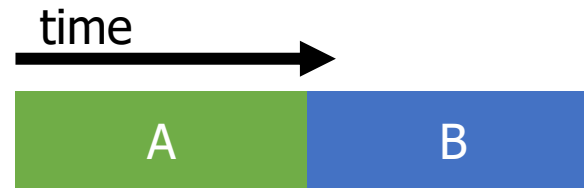
- I can get 99 friends to help me
- Whenever one result doesn't depend on another, doing the task in parallel can be a big win!

# Parallelism versus Concurrency

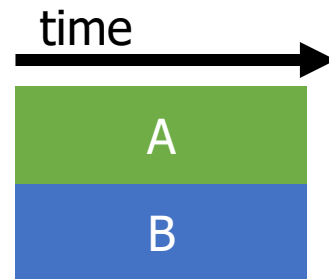
Two processes A and B



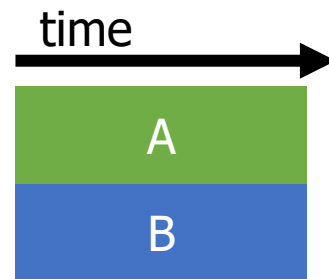
Serial execution



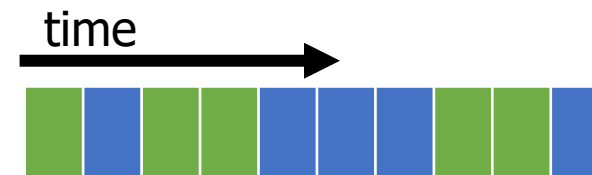
Parallel execution



Concurrent execution

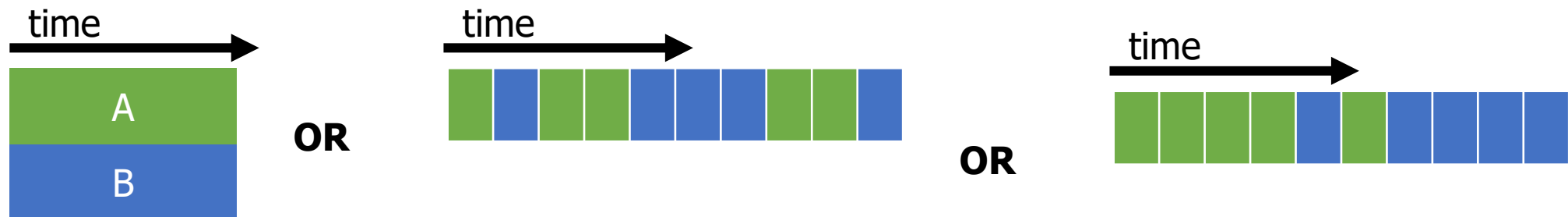


OR



# 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

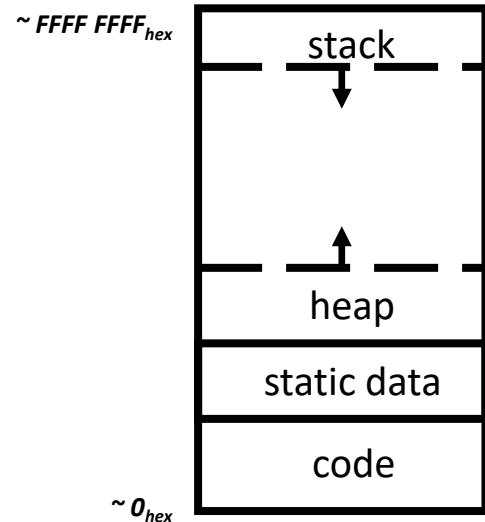
# How do we apply parallelism to software?

- 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



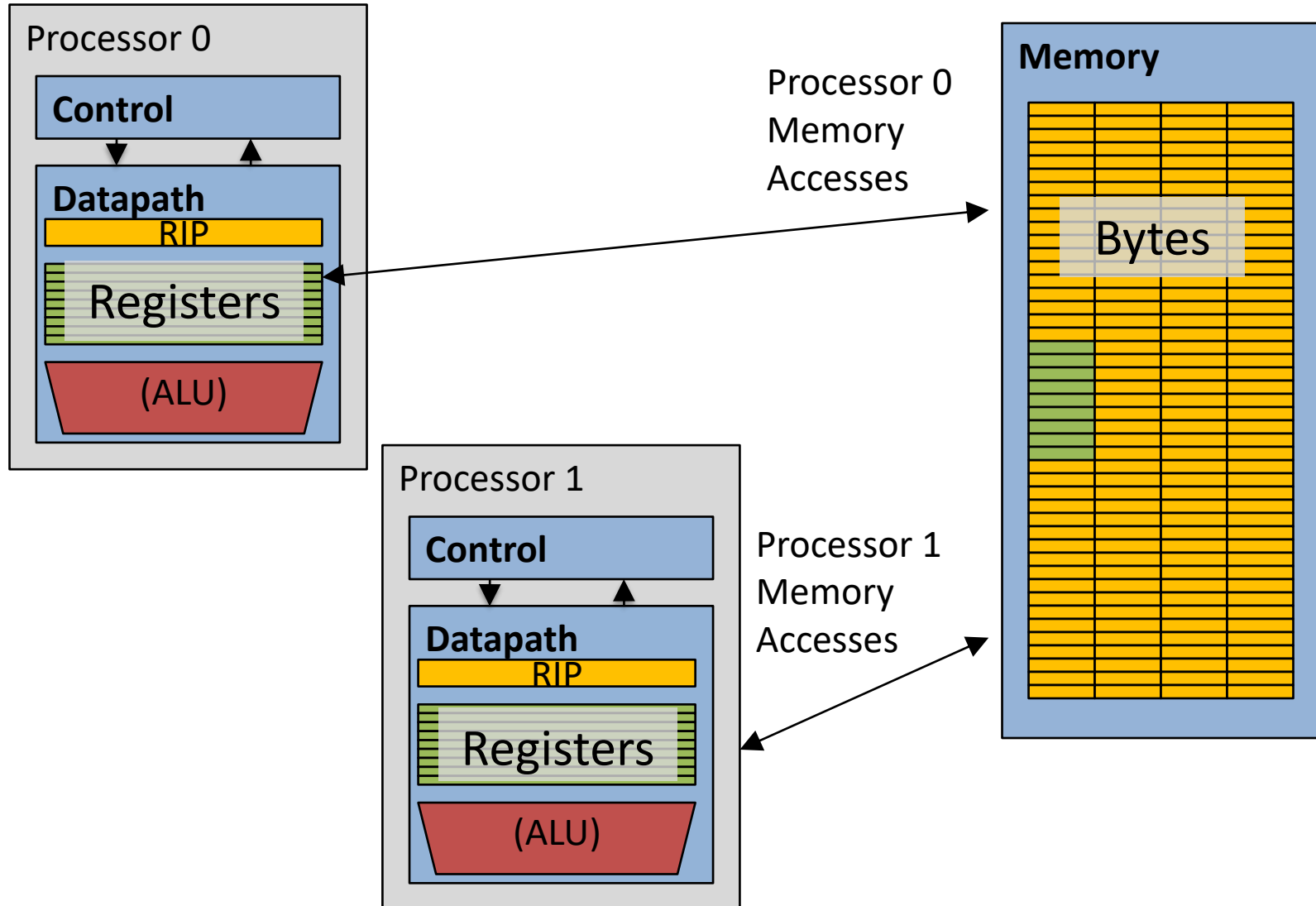
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes
- Etc.

# Process use case: separate programs

- Right now I am running:
  - Powerpoint
  - Chrome
  - CLion
- Each is a separate process
  - Have their own memory
  - Have their own registers
  - Operating System manages them
- No need for communication between them

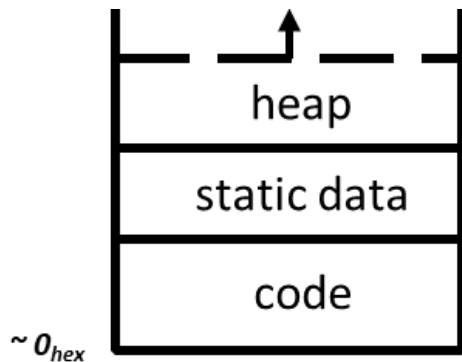
# Multiprocessor Systems (in pictures)



# 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

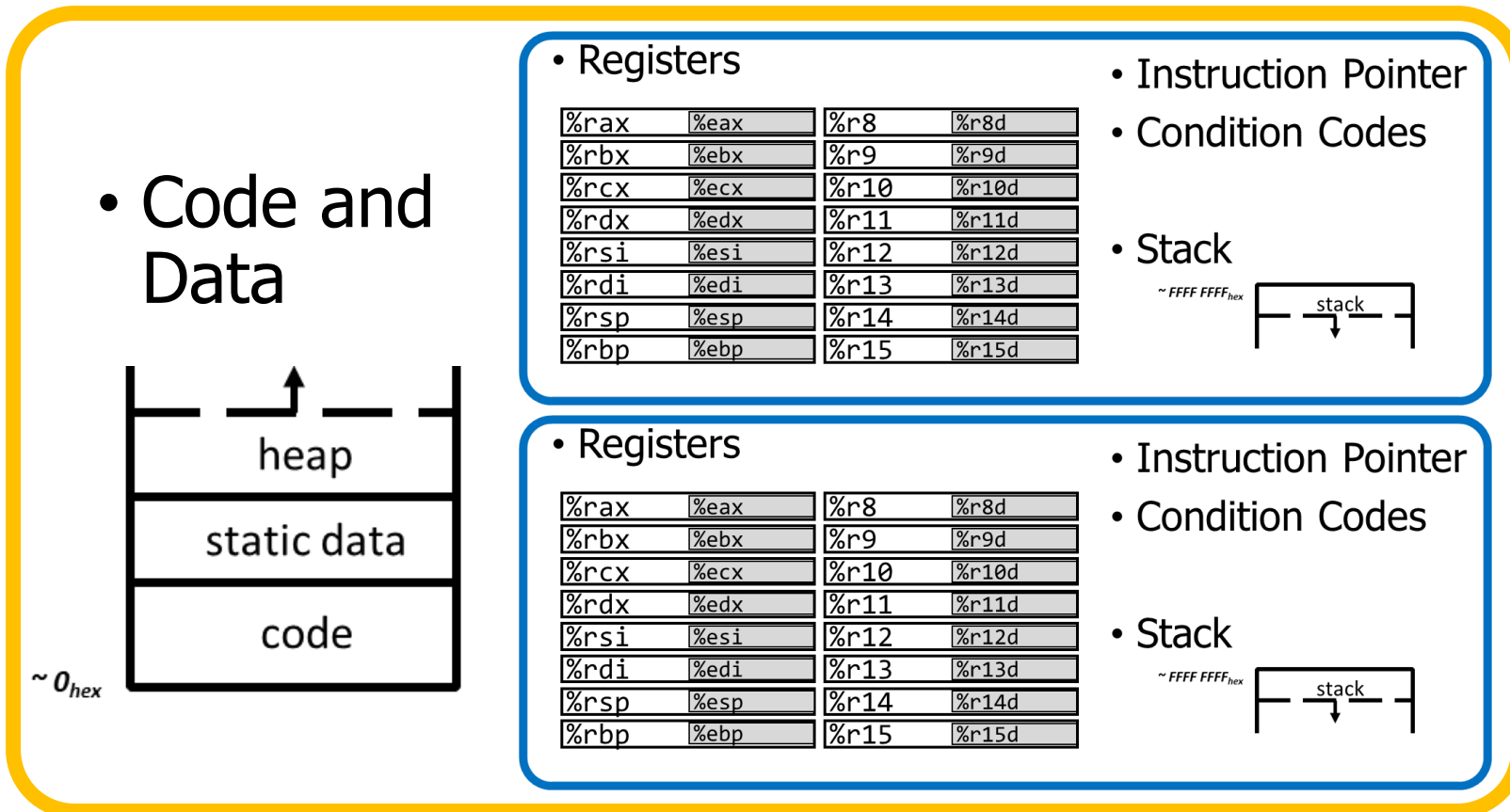
- Instruction Pointer
- Condition Codes

## • Stack



# Alternate view of a process

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



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:

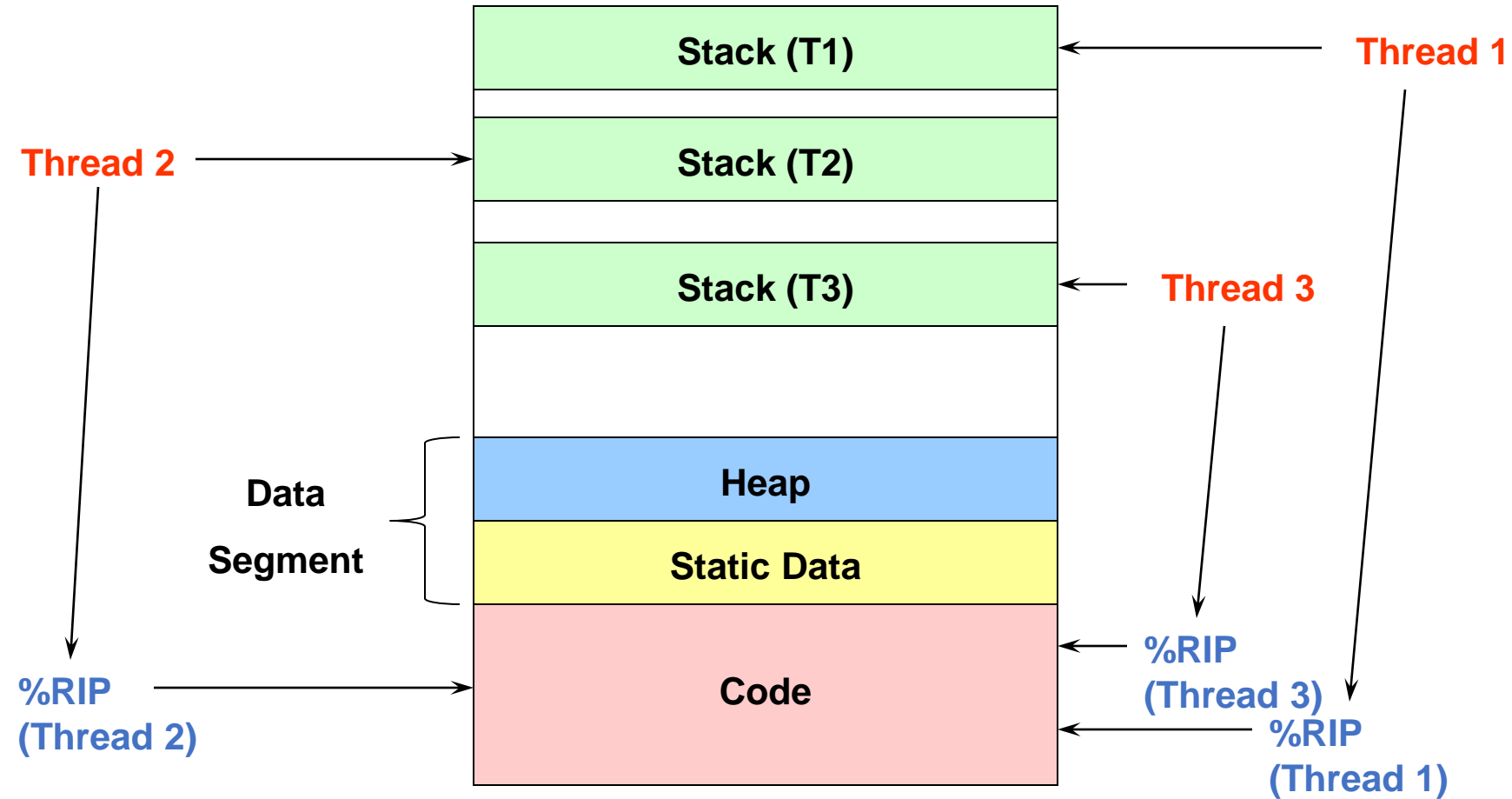
You want a tab for each web page you open:

- 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



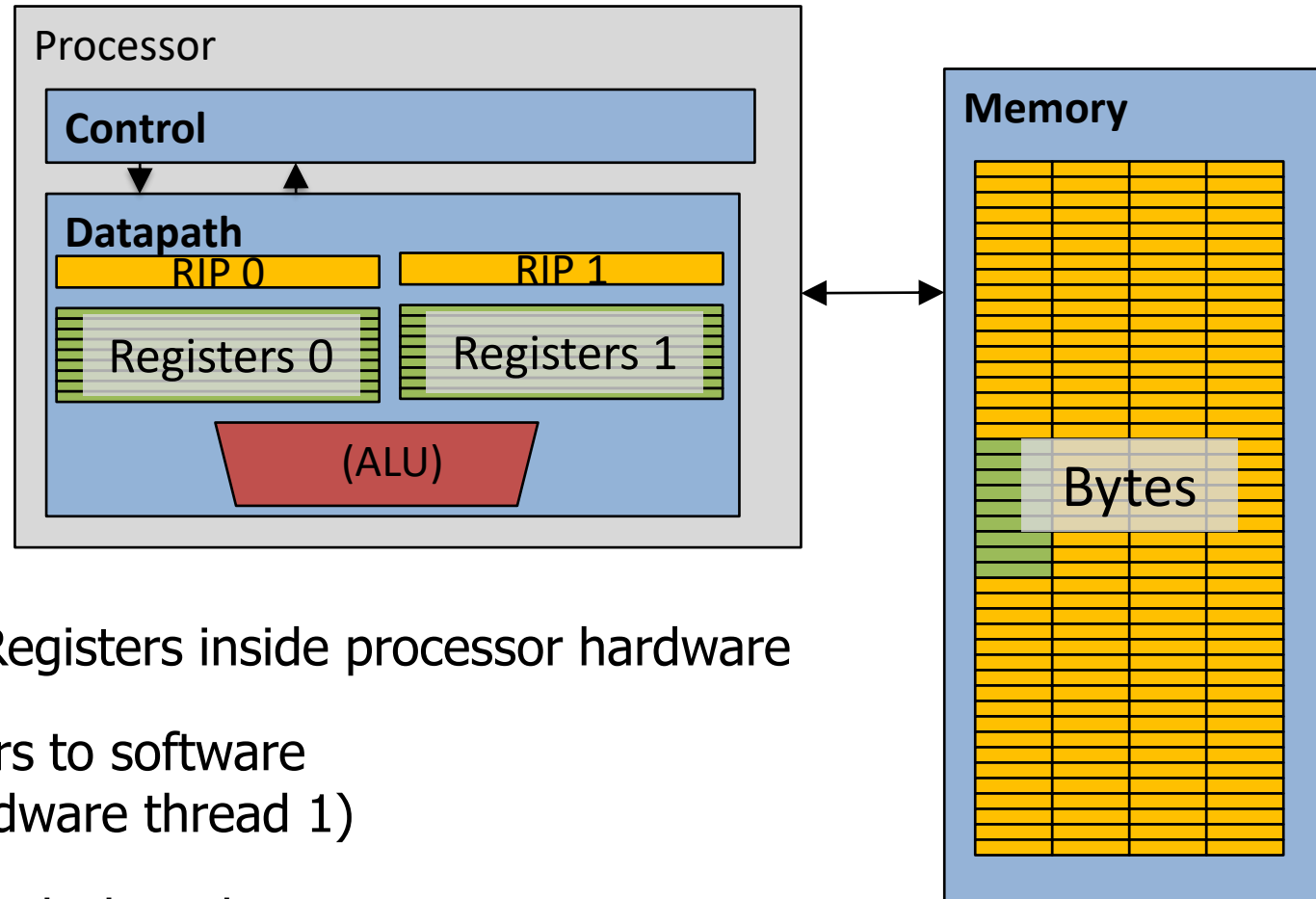
# Multithreading processors

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

Long memory latency to memory on cache miss?

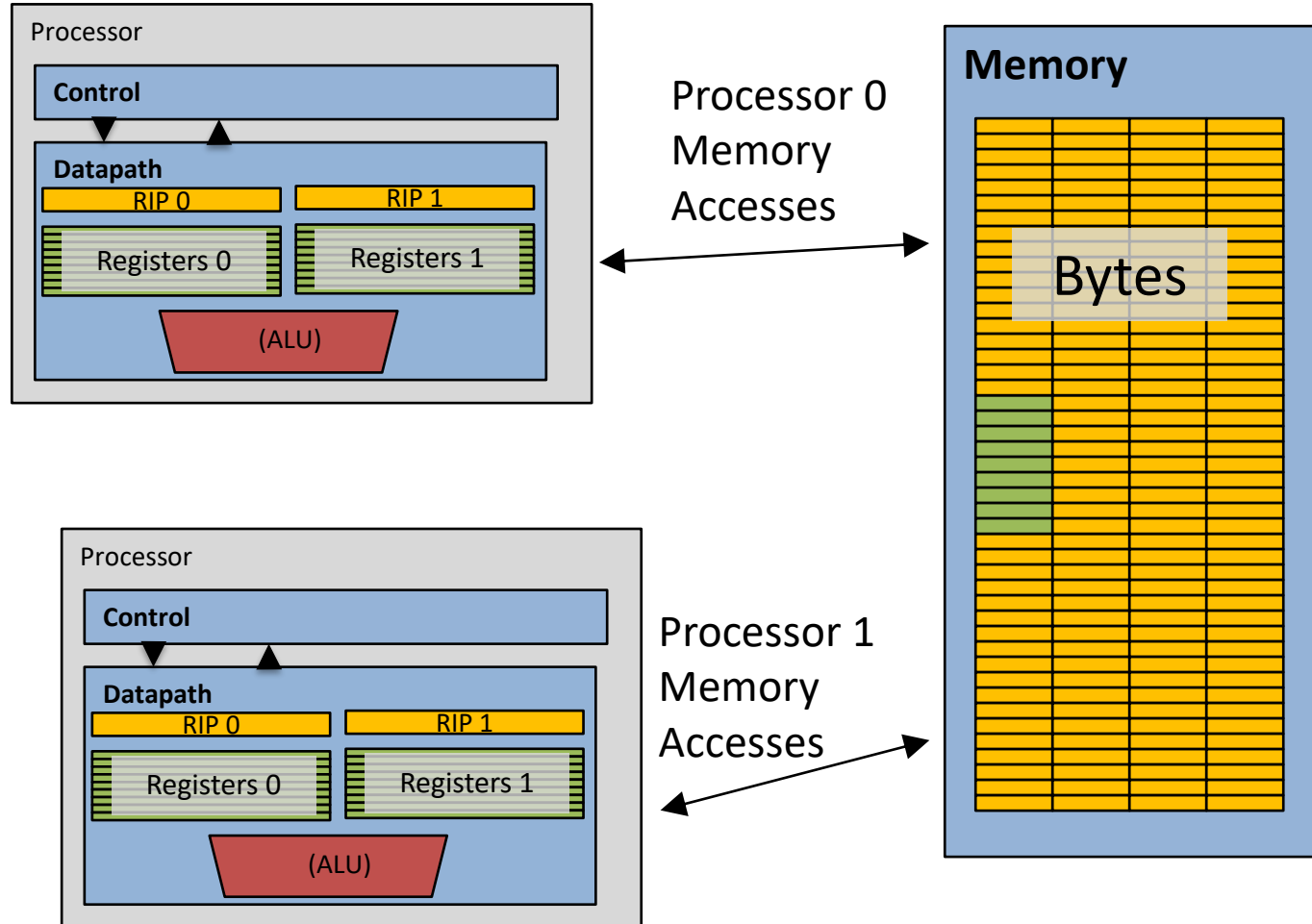
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of swapping between threads must be much less than cache miss latency

# 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

# Multithreading, multicore processors



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

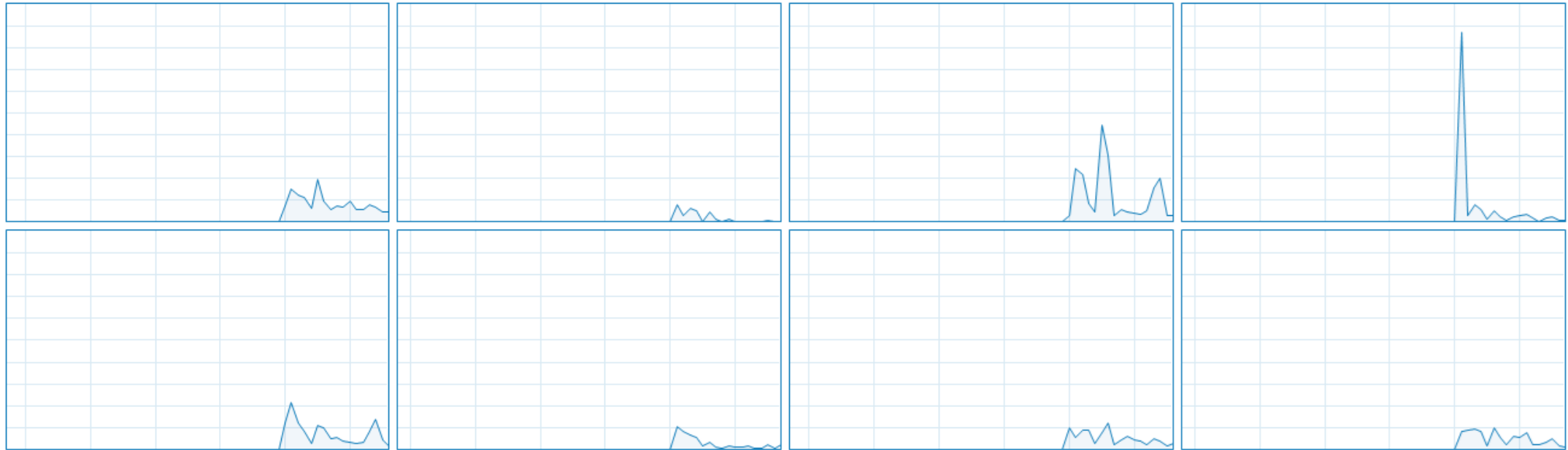
# Example: i7 processor

## CPU

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

% Utilization over 60 seconds

100%



Utilization	Speed	Base speed:	3.60 GHz
2%	4.08 GHz	Sockets:	1
Processes	Threads	Cores:	4
236	2909	Logical processors:	8
Handles	111153	Virtualization:	Enabled
Up time		L1 cache:	256 KB
12:02:28:40		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?

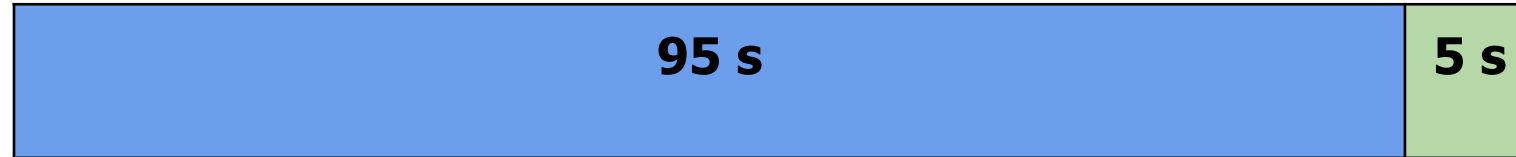
# Challenges to concurrency

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



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

$$5 \text{ s} \rightarrow 2.5 \text{ s: Speedup} = 100/97.5 = 1.026$$

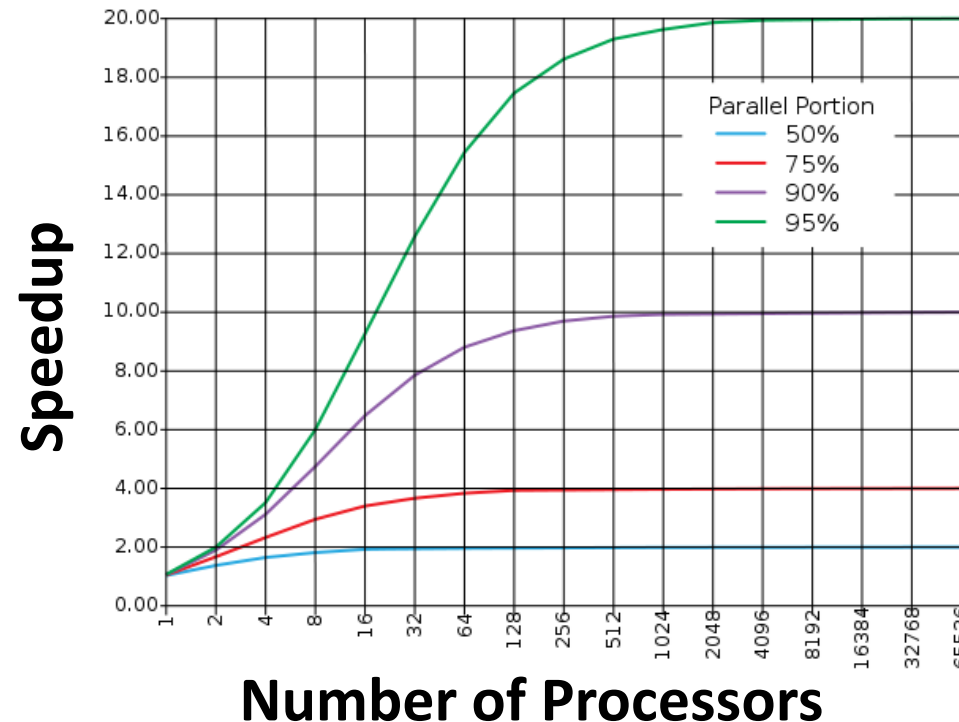
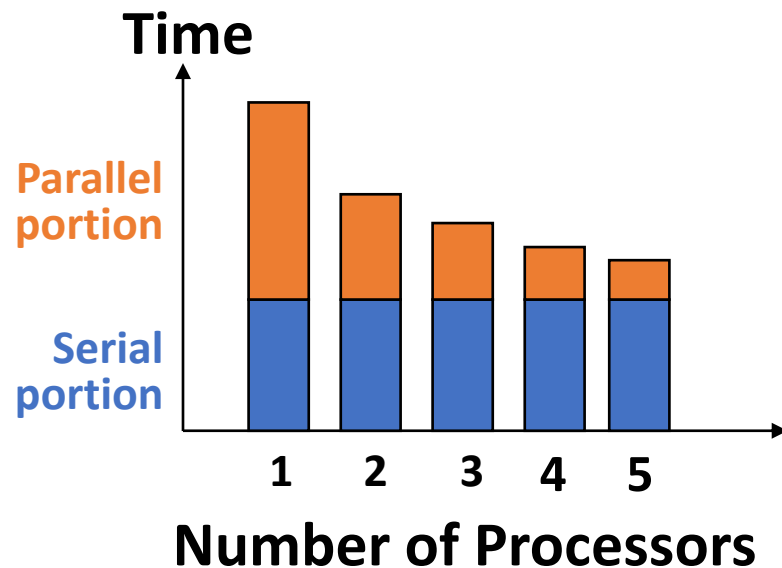
$$5 \text{ s} \rightarrow 1 \text{ s: Speedup} = 100/96 = 1.042$$

$$5 \text{ s} \rightarrow 0.001\text{s: Speedup} = 100/95.001 = 1.053$$

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

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

Time



Thread 1	Thread 2
<code>mov (%edi), %eax</code>	
<code>add \$0x1, %eax</code>	
<code>mov %eax, (\$edi)</code>	
	<code>mov (%edi), %eax</code>
	<code>add \$0x1, %eax</code>
	<code>mov %eax, (%edi)</code>

Final value of count: 2

Thread 1	Thread 2
<code>mov (%edi), %eax</code>	
	<code>mov (%edi), %eax</code>
	<code>add \$0x1, %eax</code>
	<code>mov %eax, (%edi)</code>
<code>add \$0x1, %eax</code>	
<code>mov %eax, (%edi)</code>	

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!! 

We'll pick this one for CS213
  - OR
  2. Synchronizing reading and writing to get deterministic behavior

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.
- Return is implicit call to `pthread_exit`

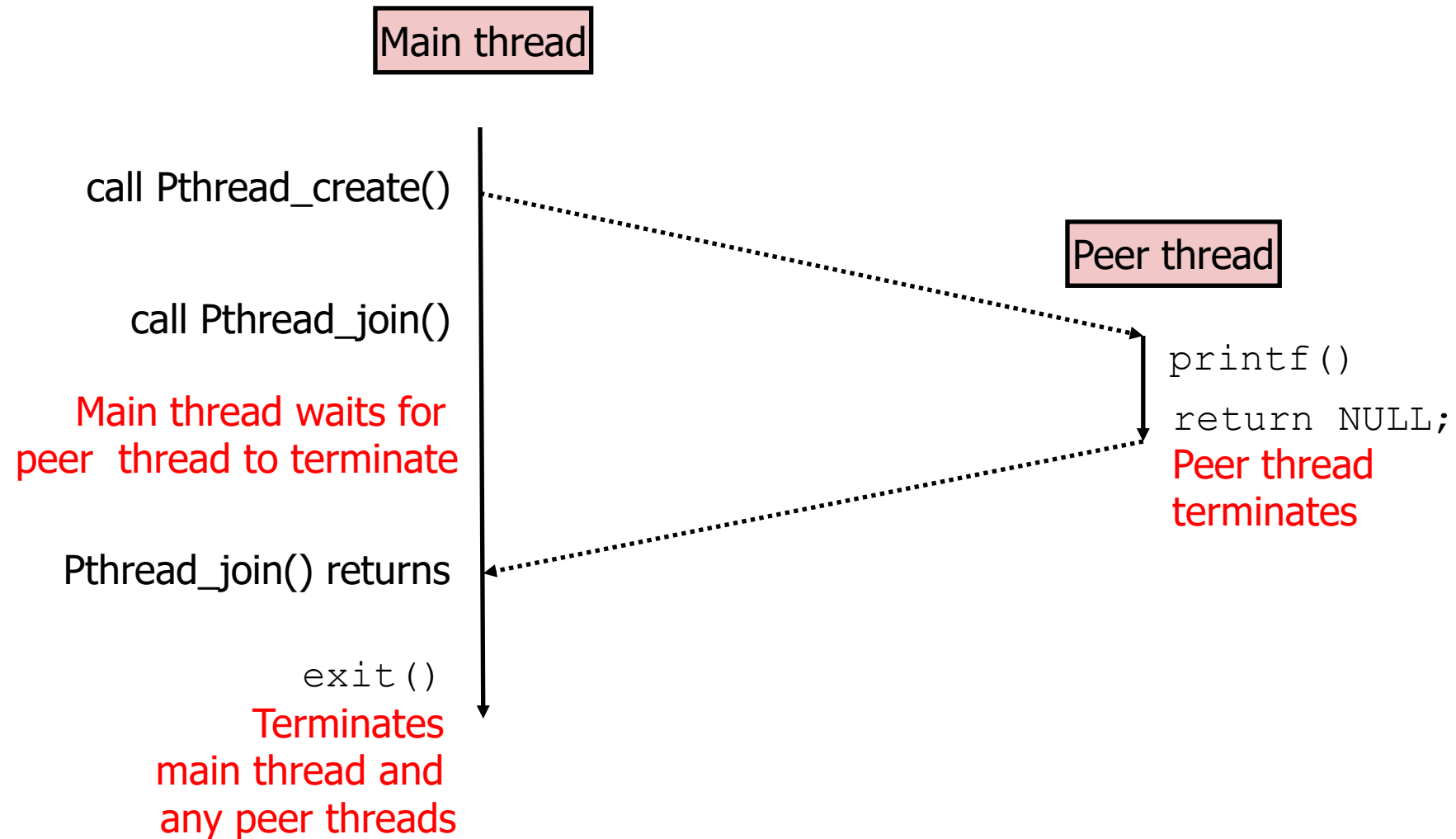
```
void pthread_exit(void* value_ptr);
```

- 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



## Example: parallel sum of vector

```
double vector[vector_len] = {1, 2, 3, ..., vector_len};
```

```
// determine result sequentially  
double sequential_sum = 0;  
for (int i=0; i<vector_len; i++) {  
    sequential_sum += vector[i];  
}
```

# Example: parallel sum of vector

```
double vector[vector_len] = {1, 2, 3, ..., vector_len};
```

## Parallelization Plan

1. Create num\_threads different threads
2. Threads create “partial” sums for their portion of the work
  - Each thread does (vector\_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 vector

## 1. Create num\_threads different threads

```
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 vector

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

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

# Example: parallel sum of vector

## 3. Wait until done, then sum the partial results

```
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 reduced version of `parallel-sum-ex.c`



# Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000  
Sequential sum: 199999999000000000 (878576632 cycles)  
Parallel sum: 0 (44 cycles)
```

Vector of 200 million length

No threads created

Only the sequential version is run

# Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum: 199999999000000000 (878576632 cycles)
Parallel sum: 0 (44 cycles)
```

```
$ ./parallel-sum-ex 1 1 200000000
Sequential sum: 199999999000000000 ( 902438479 cycles)
Parallel sum: 199999999000000000 (1169222739 cycles)
```

```
$ ./parallel-sum-ex 8 1 200000000
Sequential sum: 199999999000000000 ( 888810917 cycles)
Parallel sum: 199999999000000000 (1033659530 cycles)
```

Vector of 200 million length

1 to 8 threads created. No speedup??!

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

8 starts to pay back a little bit. But need more parallelism for a big win.

# Running the parallel sum application

```
$ ./parallel-sum-ex 0 1 200000000
Sequential sum: 199999999000000000 (878576632 cycles)
Parallel sum: 0 (44 cycles)
```

```
$ ./parallel-sum-ex 1 1 200000000
Sequential sum: 199999999000000000 ( 902438479 cycles)
Parallel sum: 199999999000000000 (1169222739 cycles)
```

```
$ ./parallel-sum-ex 8 1 200000000
Sequential sum: 199999999000000000 ( 888810917 cycles)
Parallel sum: 199999999000000000 (1033659530 cycles)
```

```
$ ./parallel-sum-ex 16 1 200000000
Sequential sum: 199999999000000000 (895258209 cycles)
Parallel sum: 199999999000000000 (693511997 cycles)
```

Vector 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: 199999999000000000 (878576632 cycles)
Parallel sum: 0 (44 cycles)
```

```
$ ./parallel-sum-ex 1 1 200000000
Sequential sum: 199999999000000000 ( 902438479 cycles)
Parallel sum: 199999999000000000 (1169222739 cycles)
```

```
$ ./parallel-sum-ex 8 1 200000000
Sequential sum: 199999999000000000 ( 888810917 cycles)
Parallel sum: 199999999000000000 (1033659530 cycles)
```

```
$ ./parallel-sum-ex 16 1 200000000
Sequential sum: 199999999000000000 (895258209 cycles)
Parallel sum: 199999999000000000 (693511997 cycles)
```

```
$ ./parallel-sum-ex 32 1 200000000
Sequential sum: 199999999000000000 (886174224 cycles)
Parallel sum: 199999999000000000 (609774231 cycles)
```

```
$ ./parallel-sum-ex 64 1 200000000
Sequential sum: 199999999000000000 (898098616 cycles)
Parallel sum: 199999999000000000 (426420305 cycles)
```

Vector 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: 199999999000000000 (878576632 cycles)
Parallel sum: 0 (44 cycles)

$ ./parallel-sum-ex 1 1 200000000
Sequential sum: 199999999000000000 ( 902438479 cycles)
Parallel sum: 199999999000000000 (1169222739 cycles)

$ ./parallel-sum-ex 8 1 200000000
Sequential sum: 199999999000000000 ( 888810917 cycles)
Parallel sum: 199999999000000000 (1033659530 cycles)

$ ./parallel-sum-ex 16 1 200000000
Sequential sum: 199999999000000000 (895258209 cycles)
Parallel sum: 199999999000000000 (693511997 cycles)

$ ./parallel-sum-ex 32 1 200000000
Sequential sum: 199999999000000000 (886174224 cycles)
Parallel sum: 199999999000000000 (609774231 cycles)

$ ./parallel-sum-ex 64 1 200000000
Sequential sum: 199999999000000000 (898098616 cycles)
Parallel sum: 199999999000000000 (426420305 cycles)

$ ./parallel-sum-ex 128 1 200000000
Sequential sum: 199999999000000000 (891919128 cycles)
Parallel sum: 199999999000000000 (493951974 cycles)
```

Vector 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

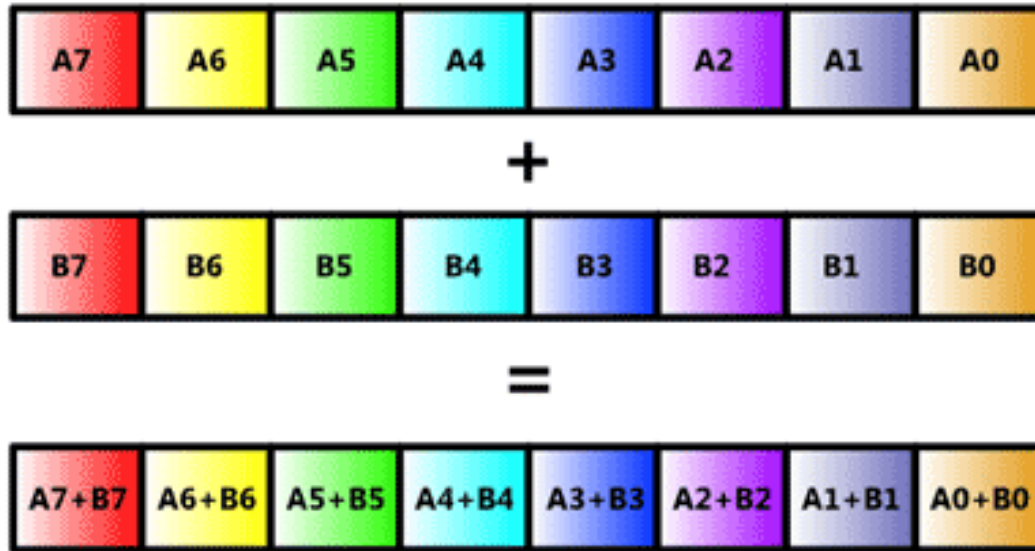
- *Data-Level Parallelism (DLP)*: Executing one operation on multiple data streams
  - SIMD: Single Instruction Multiple Data
- **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

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

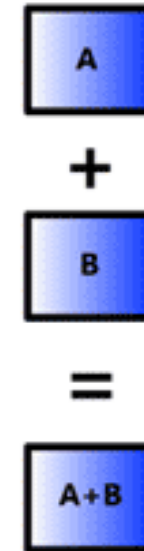
- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well



## SIMD Mode

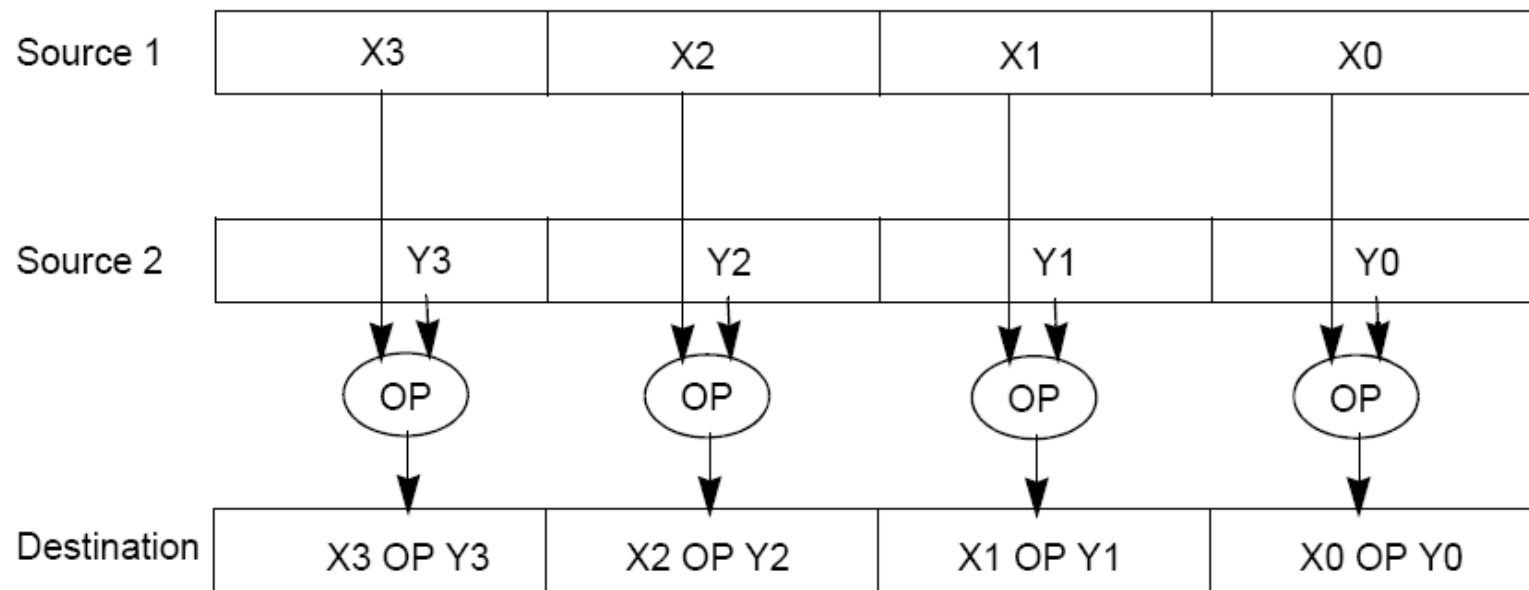


## Scalar Mode



# 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

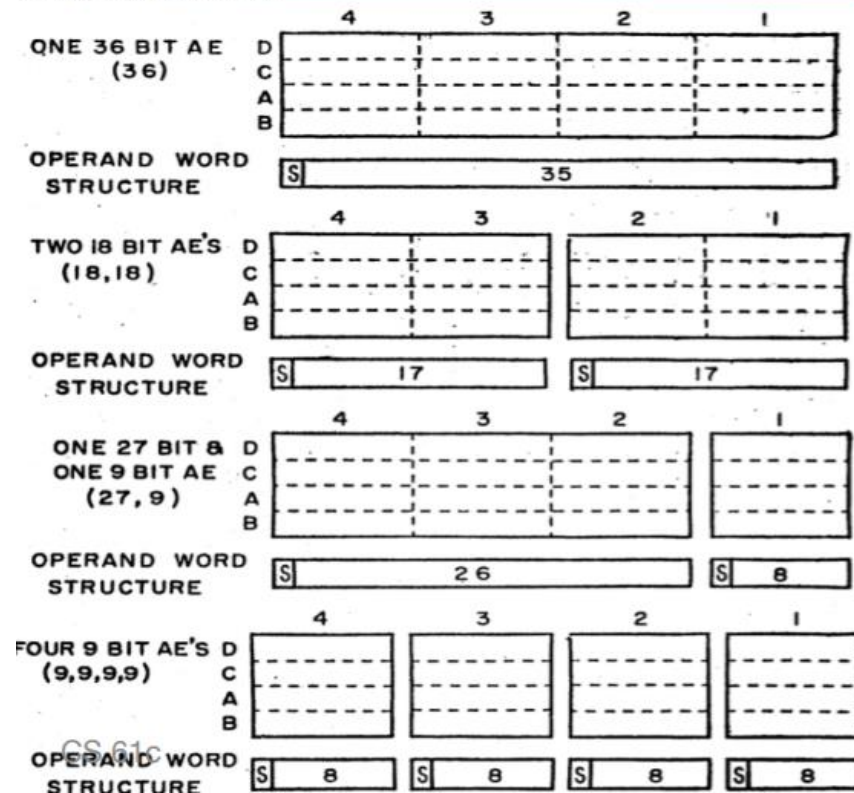
# SIMD in the Real World

- 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

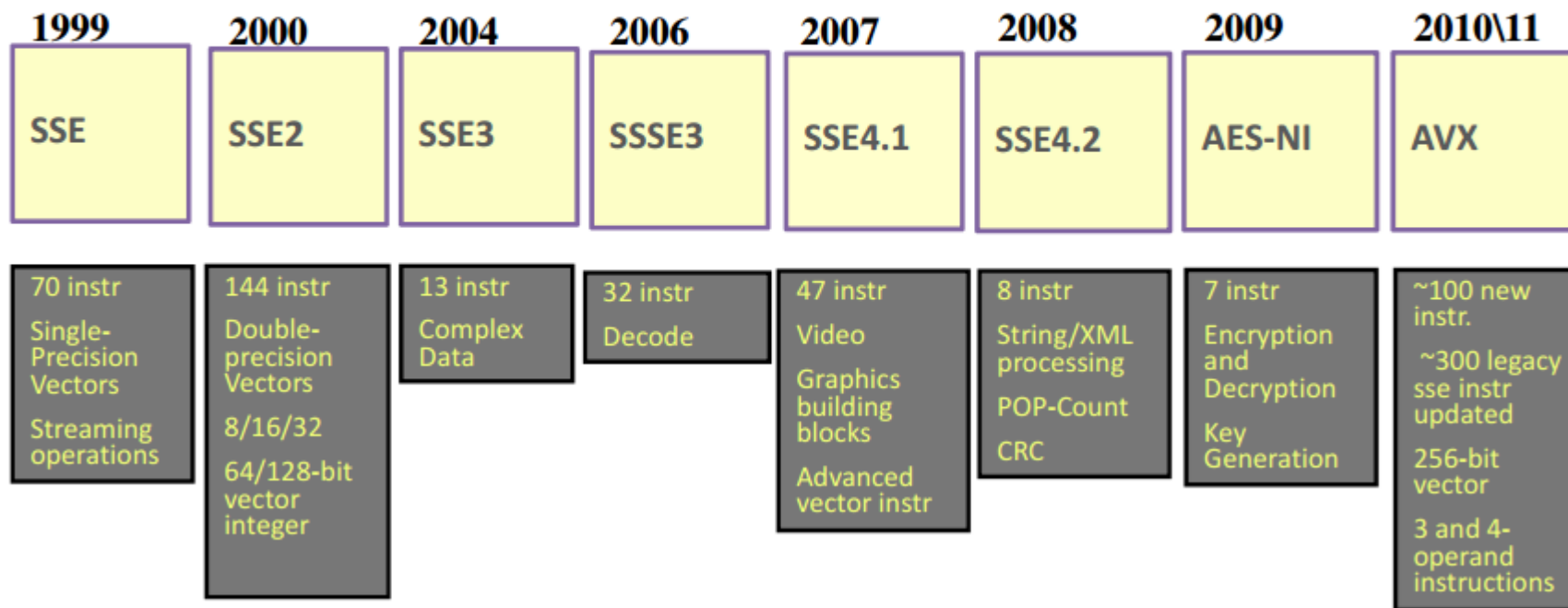
Computer Science 61C Spring 2019

Weaver



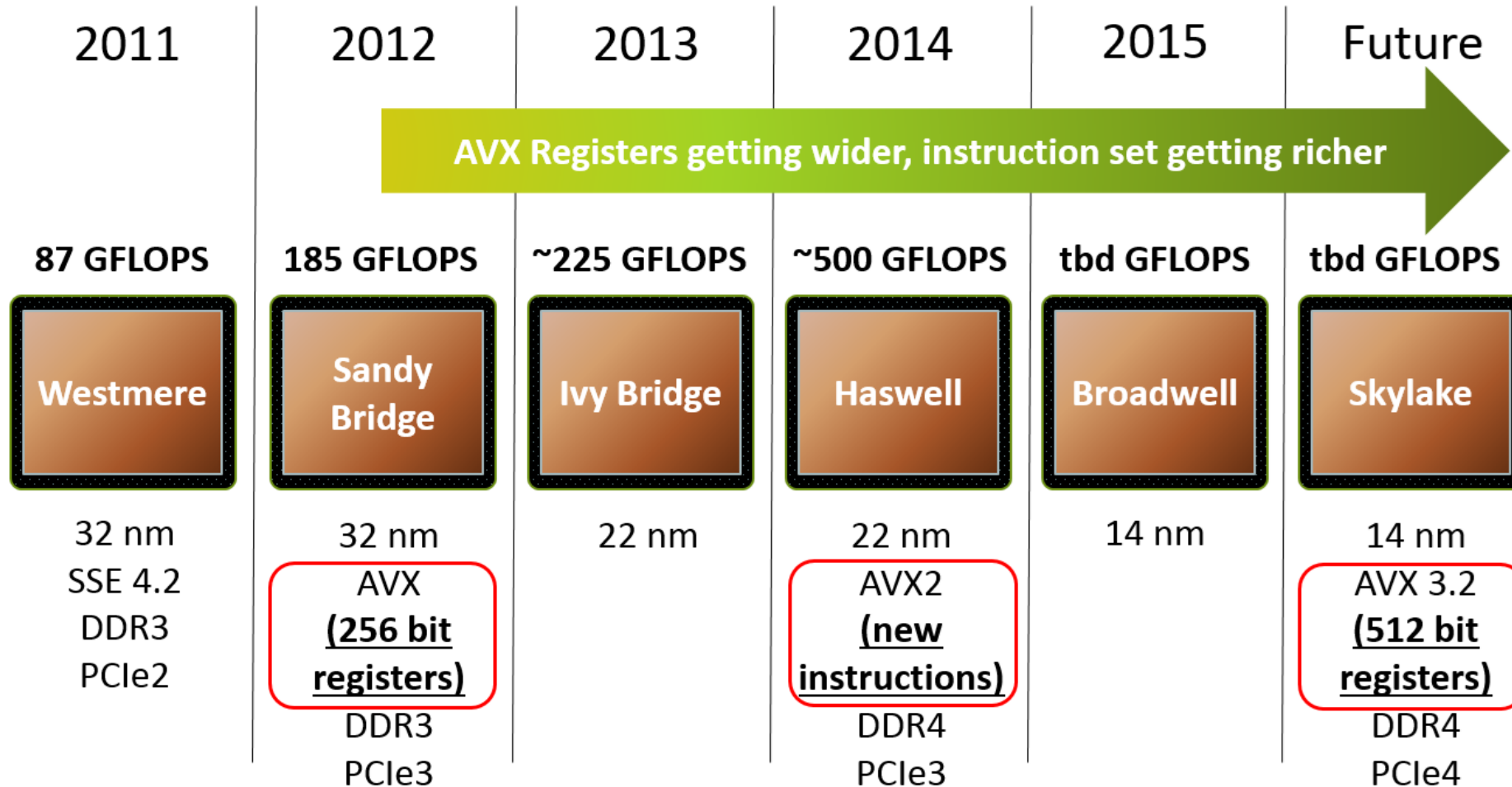
# Intel SIMD has been continuously extended

## SIMD: Continuous Evolution

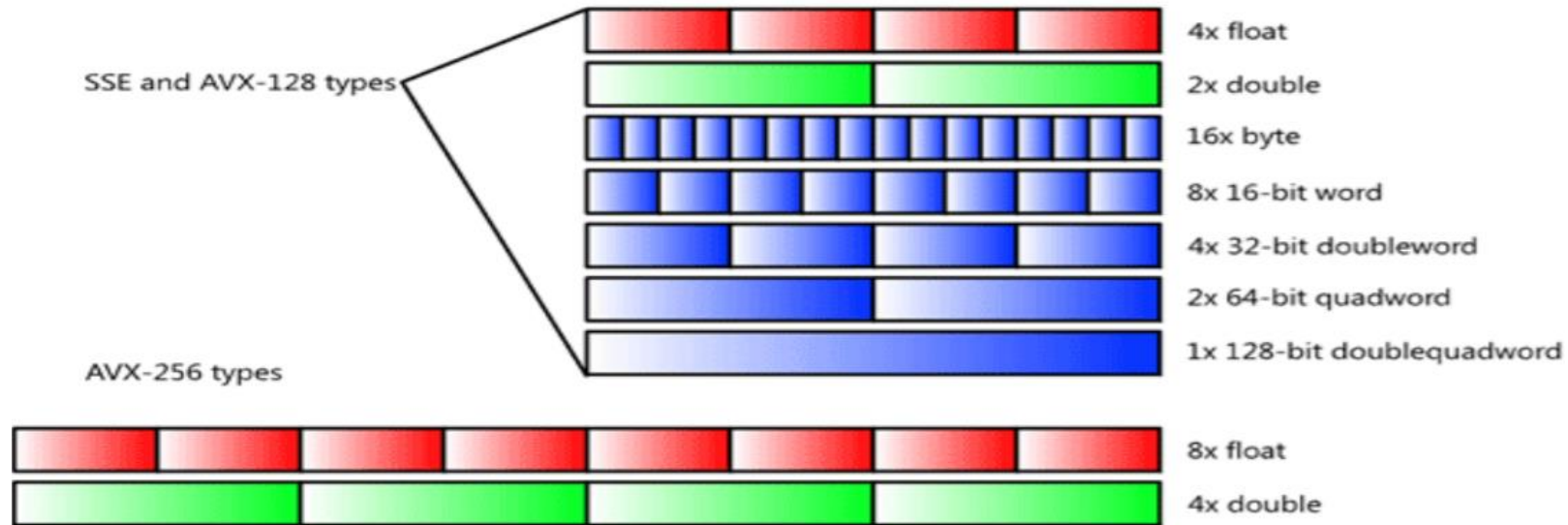


# And it has increased in size a lot

## Intel Advanced Vector eXtensions



# 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

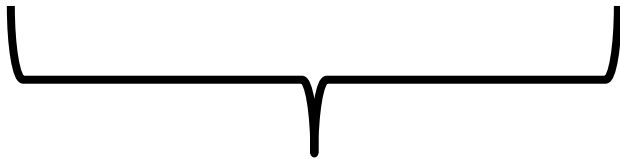
- SSE-2+ supports wider data types to allow  $16 \times 8$ -bit and  $8 \times 16$ -bit operands

# 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);
```

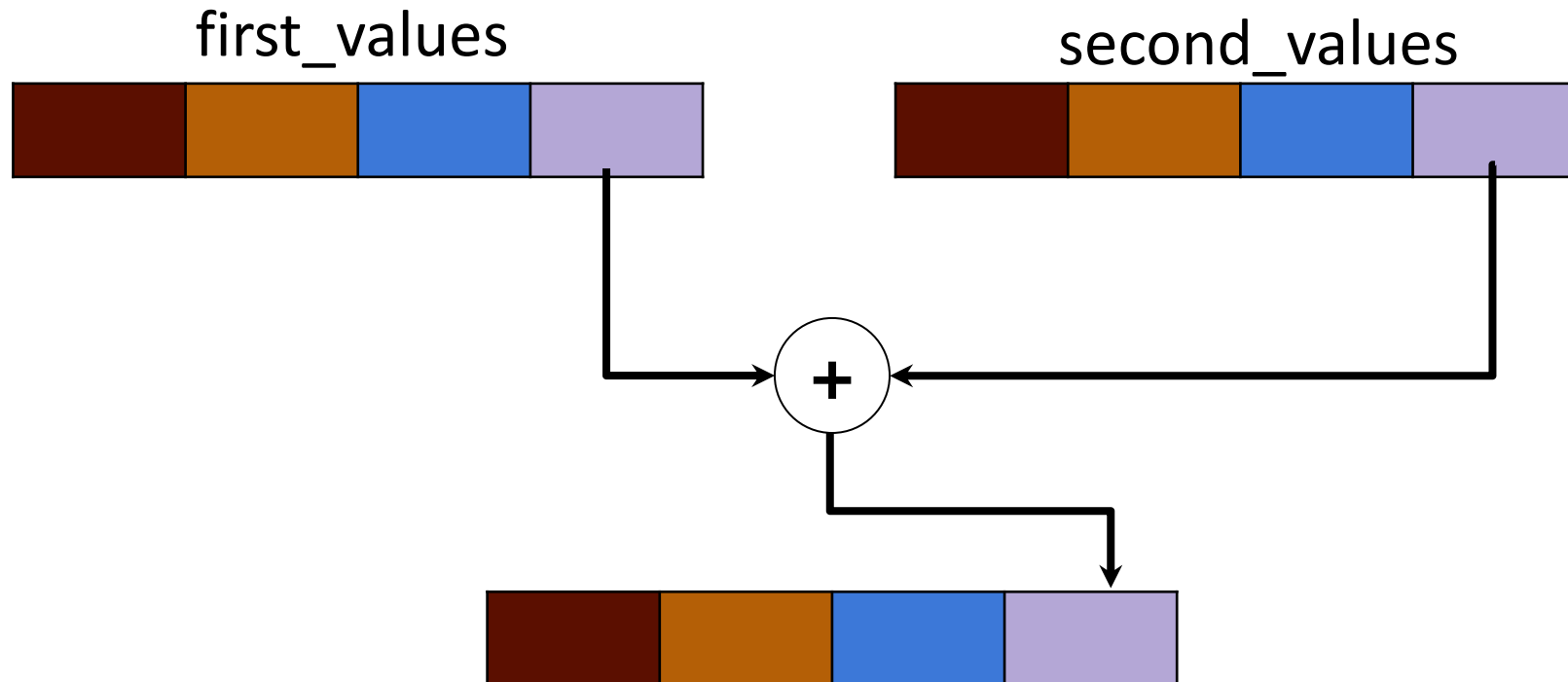


WHAT????

`_mm_add_epi32(first_values, second_values)`

↑  
**MultiMedia** extension  
(They all start with this)

Arguments are  
**Extended Packed Integers**,  
each **32**-bits in size  
(signed)



## 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
- ☐ Convert

## mm\_add\_epi32

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

### Synopsis

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddq xmm, xmm
CUID 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

Sooooooooo  
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]);

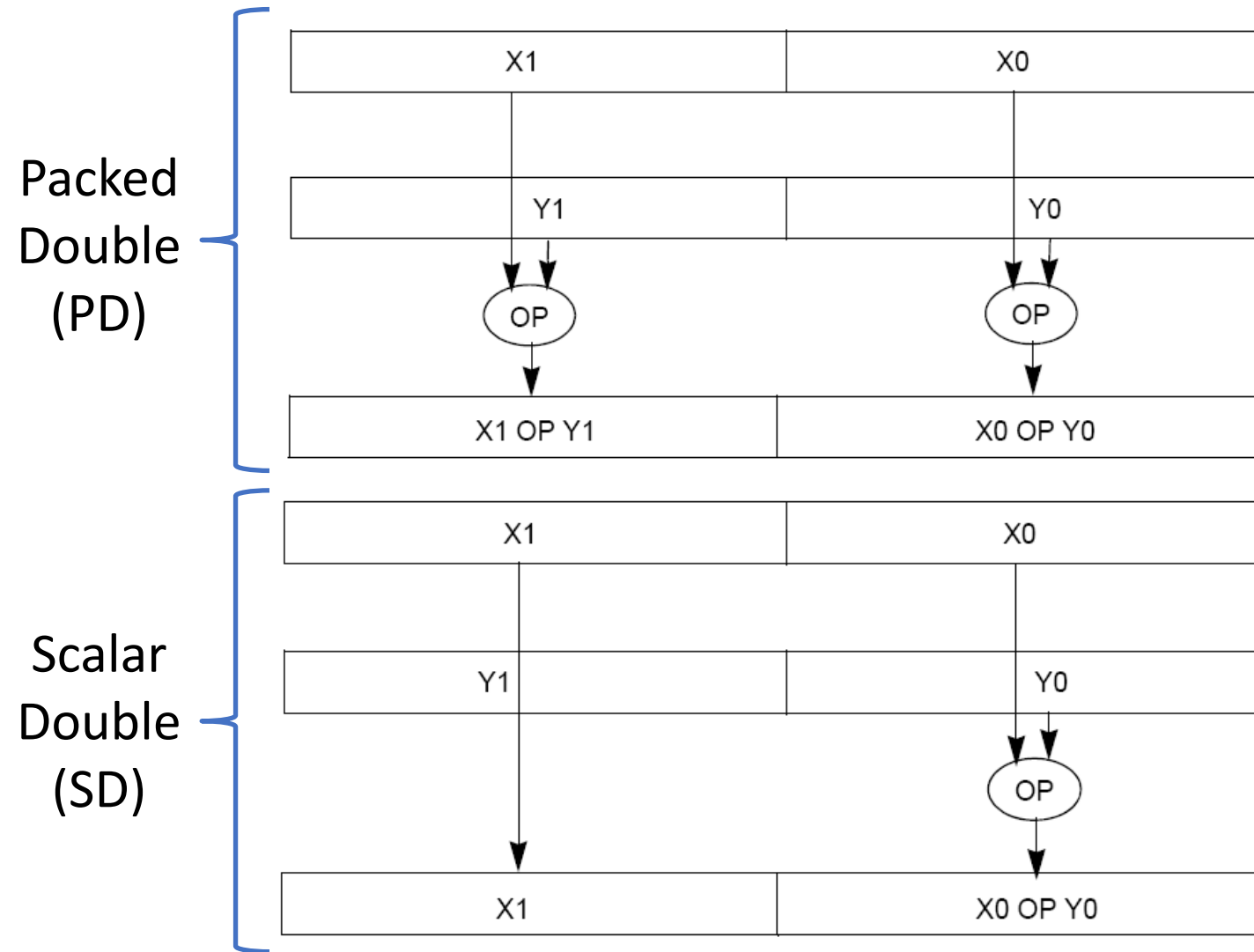
        // add each pair of 32-bit integers in the 128-bit chunks
        first_values = _mm_add_epi32(first_values, second_values);

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

    ...
}

```

# You can do this with floating point numbers too!



# Example: Reversing an array in 7 steps (animated)

