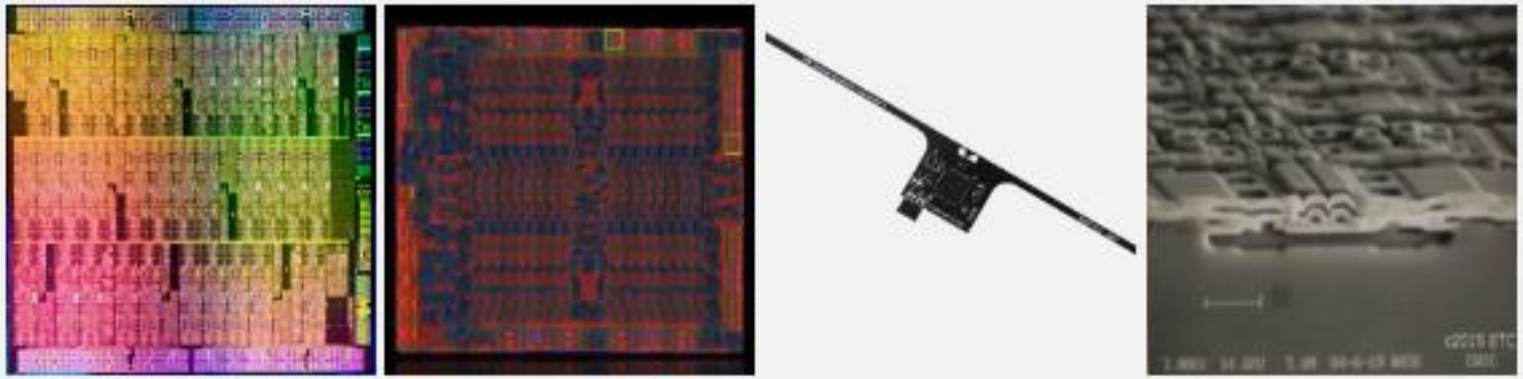
18-344: Computer Systems and the Hardware-Software Interface



Course Description

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Some pesky details…

- No mid-term and final exam for this class
- Grade break-down: 80% labs and 20% written assignments
- Lab 0 is released today, due Sep 4 by 11:59:59 pm ET
  - Please read the lab 0 handout in the course webpage carefully
  - Lab 0 must be completed individually
  - Labs 1-2 must have the same partner
  - You can choose to switch partners before doing labs 3-5
- Please post common questions in the general channel
  - Ensures faster response
  - Use the DM option to discuss personal situations

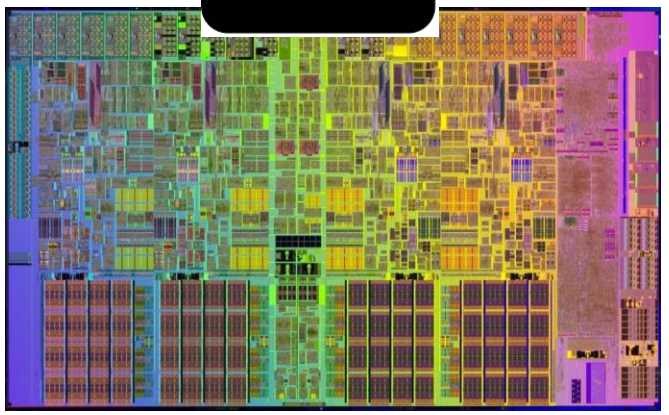
What is the hardware/software boundary?

- An ISA or Computer Architecture?
- A division of labor between Computer Engineers and Programmers?
- A split between what you can change and what you cannot change?
- Python vs. Verilog?

The 213 view of the world

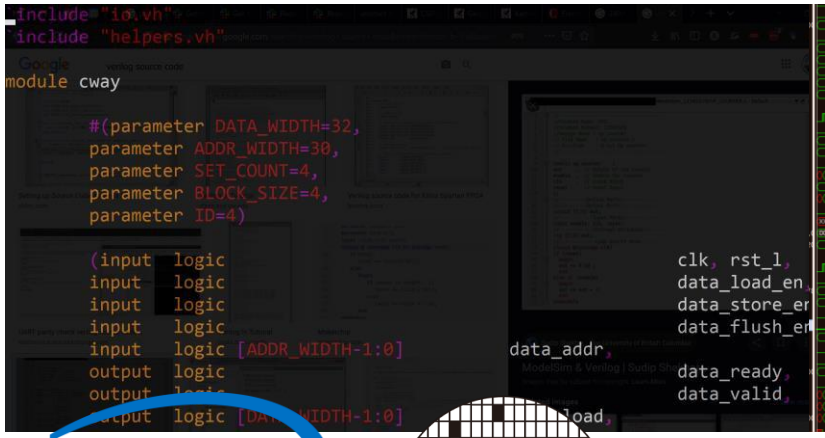
```
int walk_page_range(struct mm_struct *mm, unsigned long start,
                unsigned long end, const struct mm_walk_ops *ops,
                void *private)
{
        int err = 0;
        unsigned long next;
        struct vm_area_struct *vma;
        struct mm_walk walk = {
                .ops            = ops,
                .mm             = mm,
                .private        = private,
        };

        if (start >= end)
                return -EINVAL;

        if (!walk.mm)
                return -EINVAL;

        mmap_assert_locked(walk.mm);

        vma = find_vma(walk.mm, start);
        do {
                if (!vma) { /* after the last vma */
                        walk.vma = NULL;
                        next = end;
```

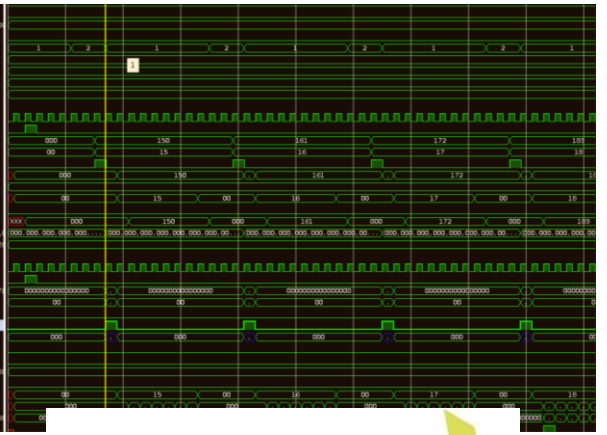
- ISA is the *immutable* foundation of the system
- High-level language compiles to ISA
- Linux (or other) OS provides important low-level services
- Low-level optimization: know HW structure to make smart code changes

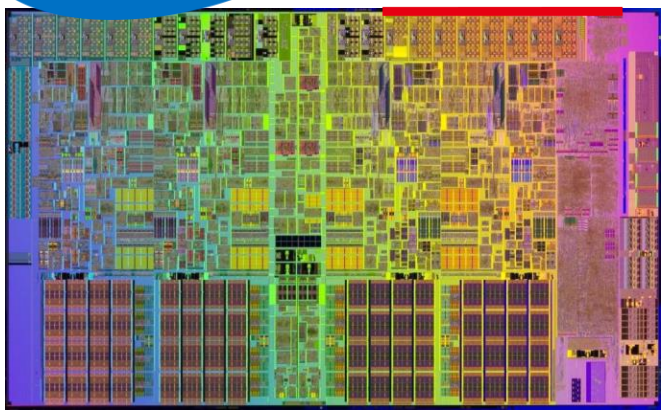


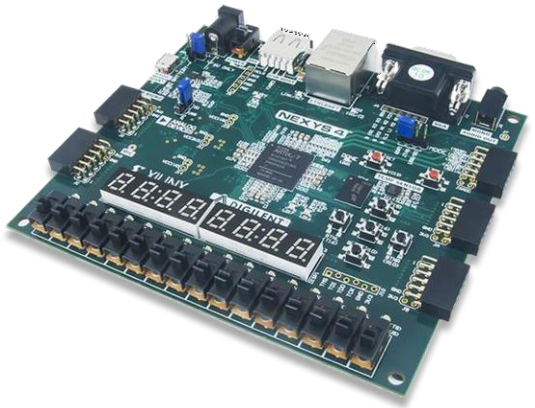
The 240 view of the world

- What's an ISA? (RISCV-240 not withstanding)
- SystemVerilog describes your hardware
- What's an OS? What is *software* even?
- Implement through simulation, ASIC fabrication or FPGA configuration





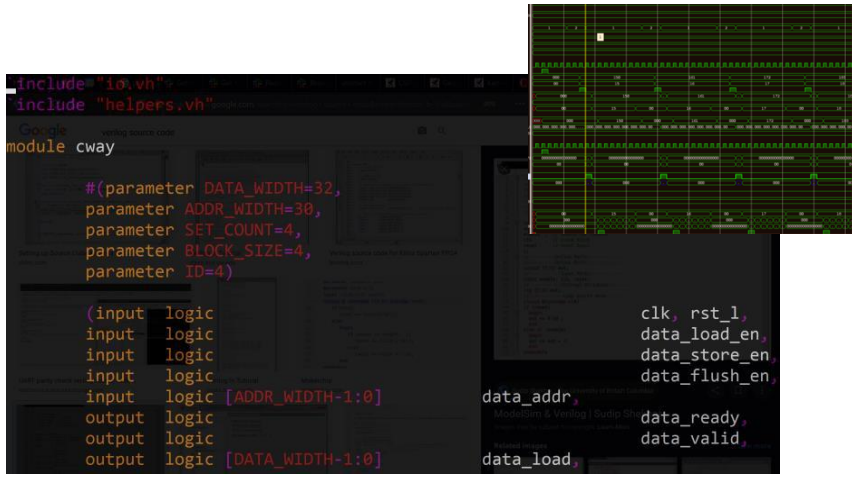




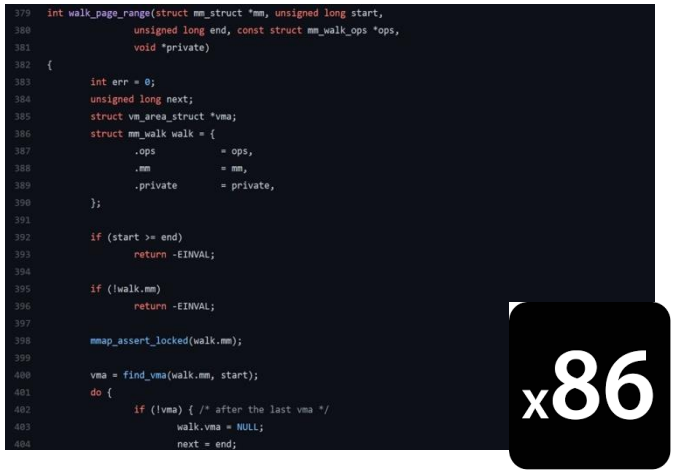
Relative Mutability/Non-Recurring Eng. (NRE) Cost?

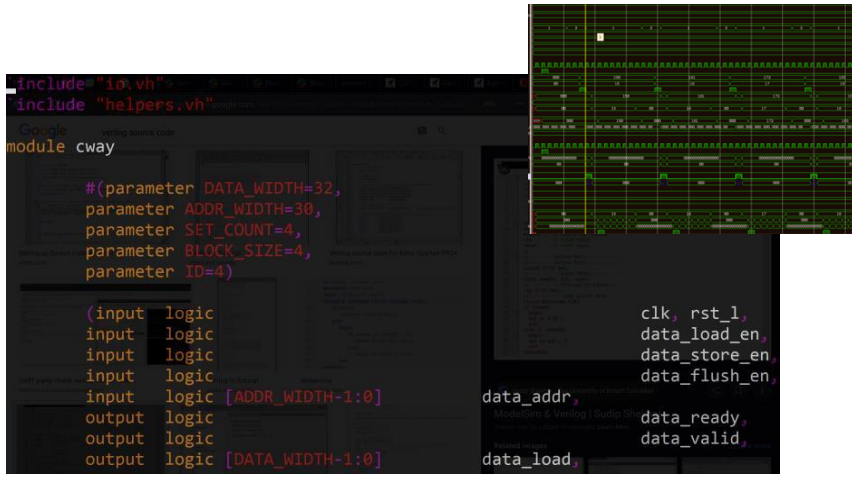
```
int walk_page_range(struct mm_struct *mm, unsigned long start,
                unsigned long end, const struct mm_walk_ops *ops,
                void *private)
{
        int err = 0;
        unsigned long next;
        struct vm_area_struct *vma;
        struct mm_walk walk = {
                .ops            = ops,
                .mm             = mm,
                .private        = private,
        };

        if (start >= end)
                return -EINVAL;

        if (!walk.mm)
                return -EINVAL;

        mmap_assert_locked(walk.mm);

        vma = find_vma(walk.mm, start);
        do {
                if (!vma) { /* after the last vma */
                        walk.vma = NULL;
                        next = end;
```

x86

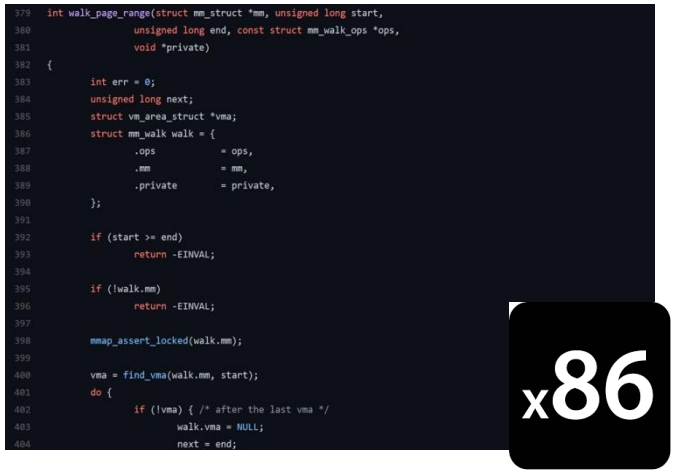


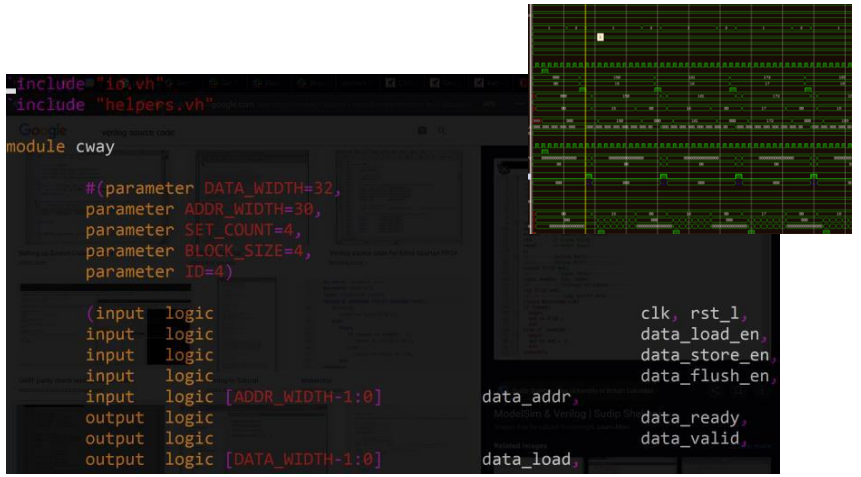
Relative Observability?

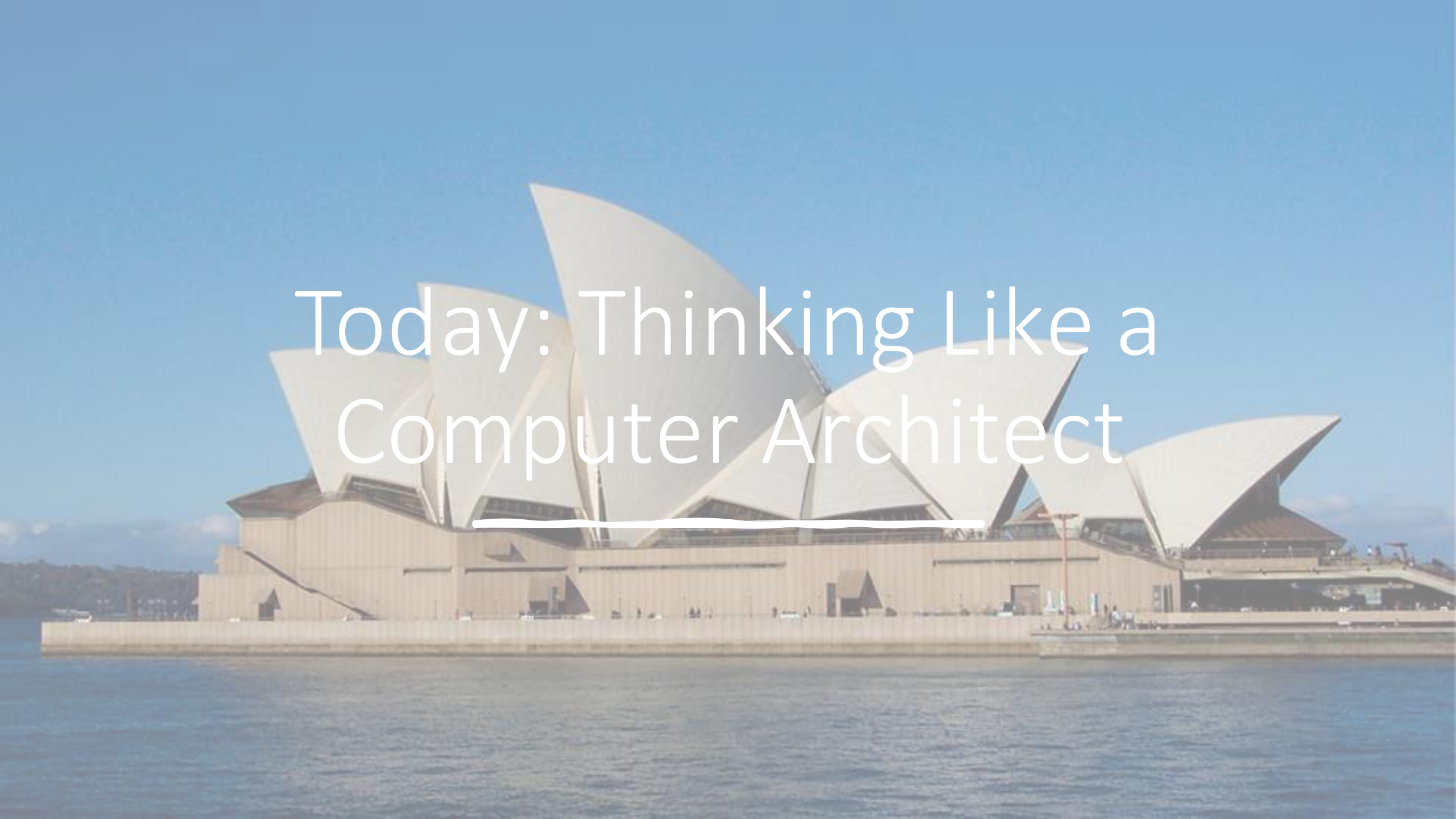




Relative Optimizability?

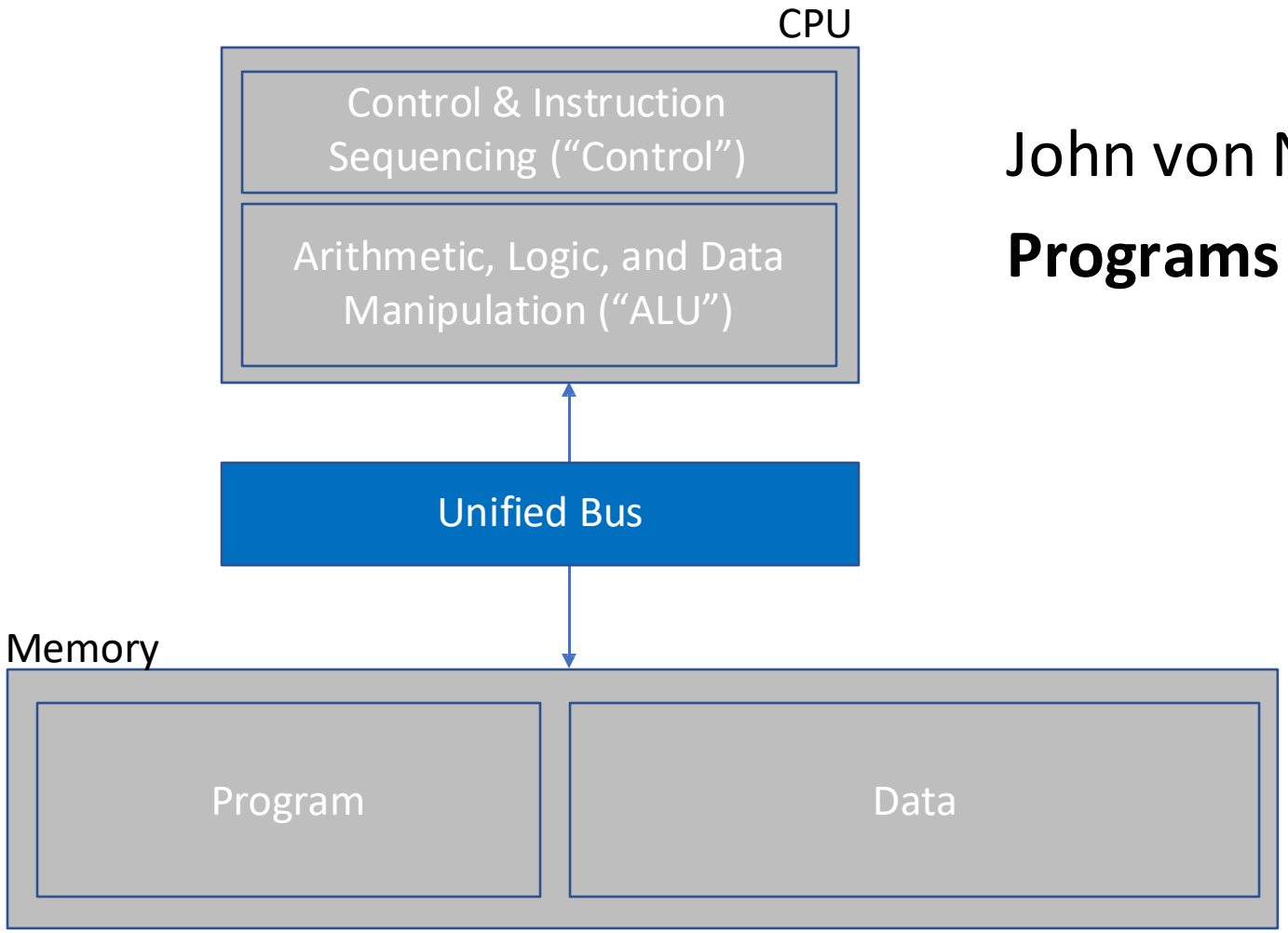






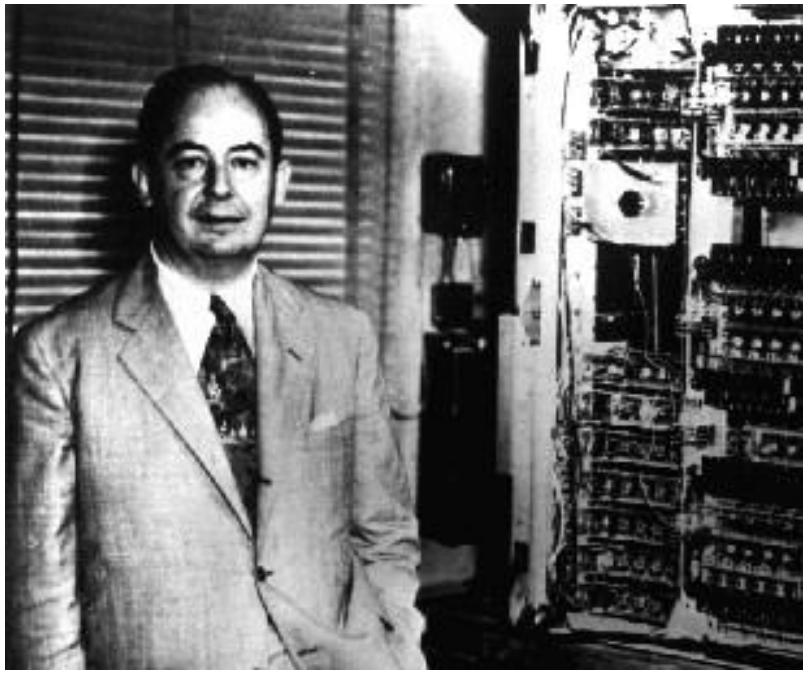
Today: Thinking Like a Computer Architect

Our first hw/sw interface:
The Von Neumann Computing Model

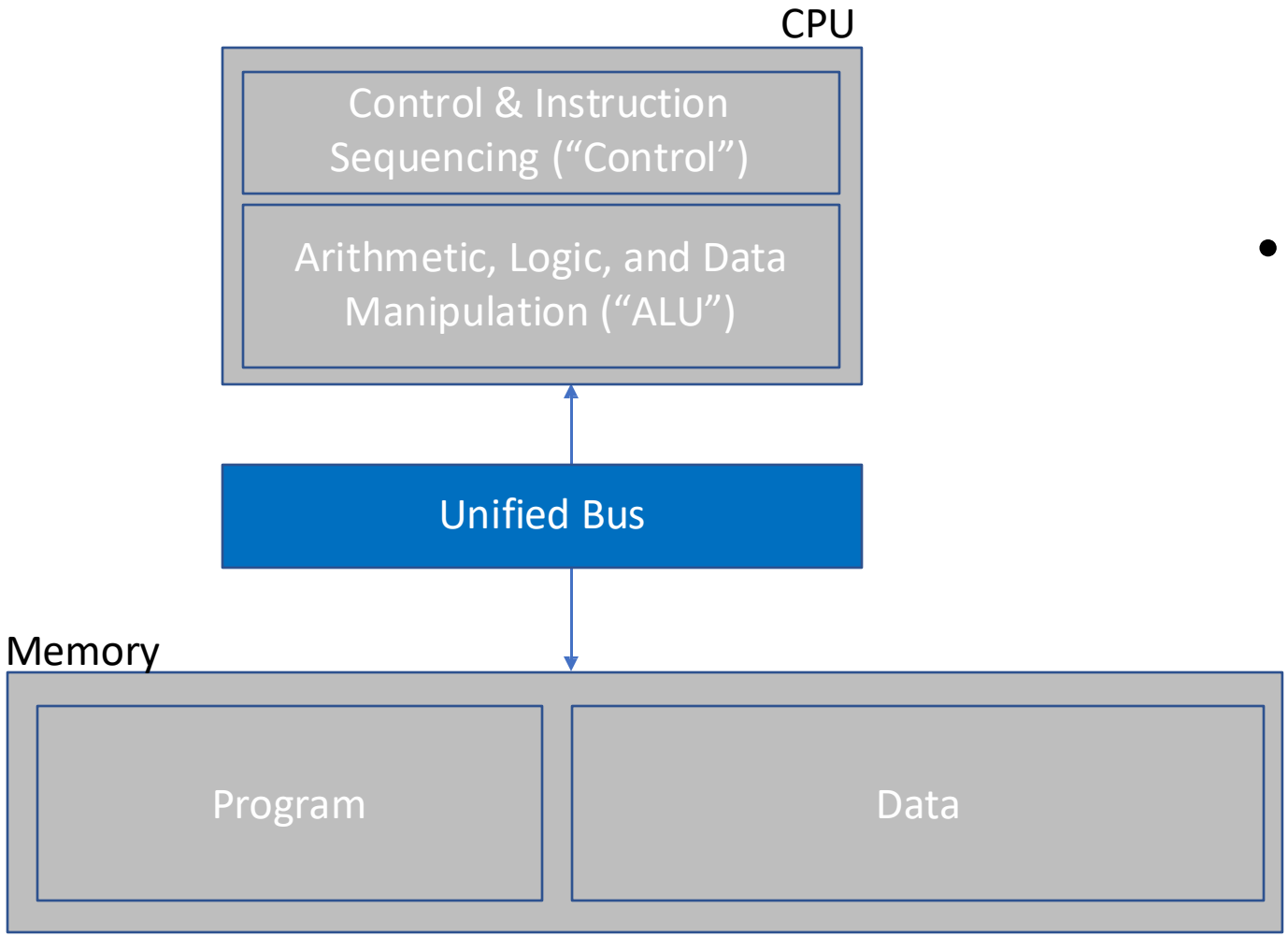


John von Neumann's Big Idea:

Programs are data.

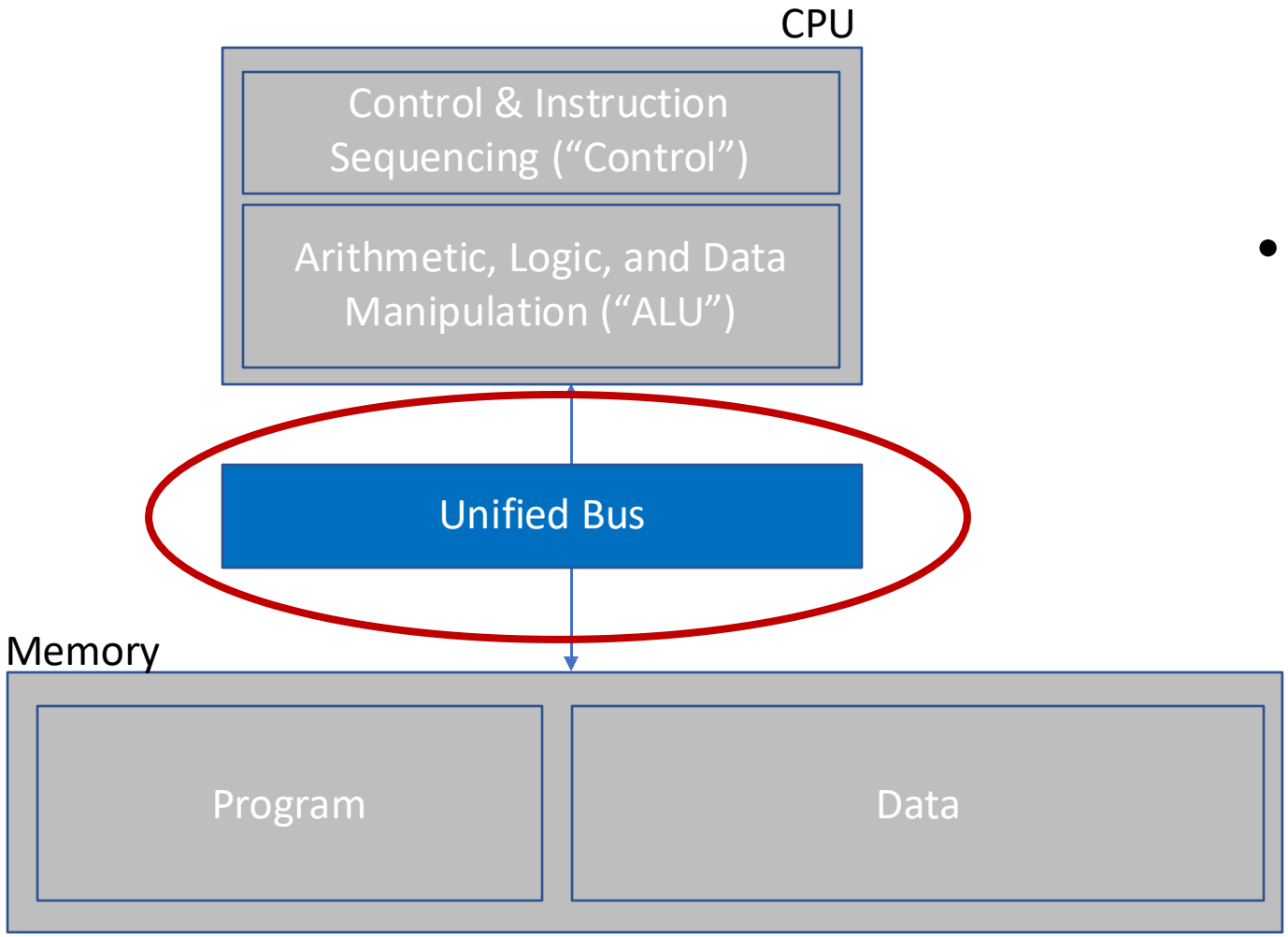


The Von Neumann / Stored-Program Computing Model



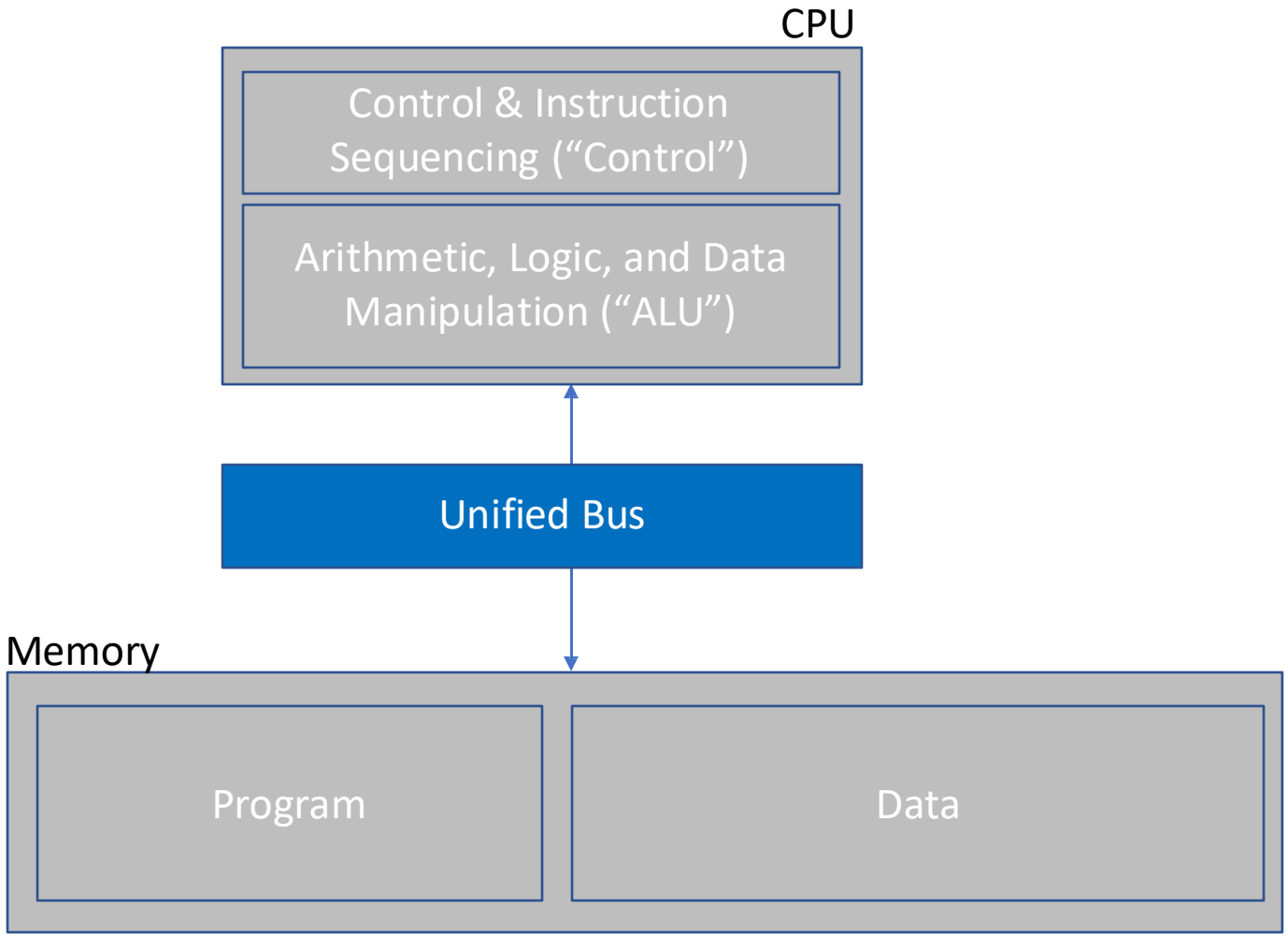
- Let’s optimize! Where is there a bottleneck in the Von Neumann abstract machine?

The Von Neumann / Stored-Program Computing Model



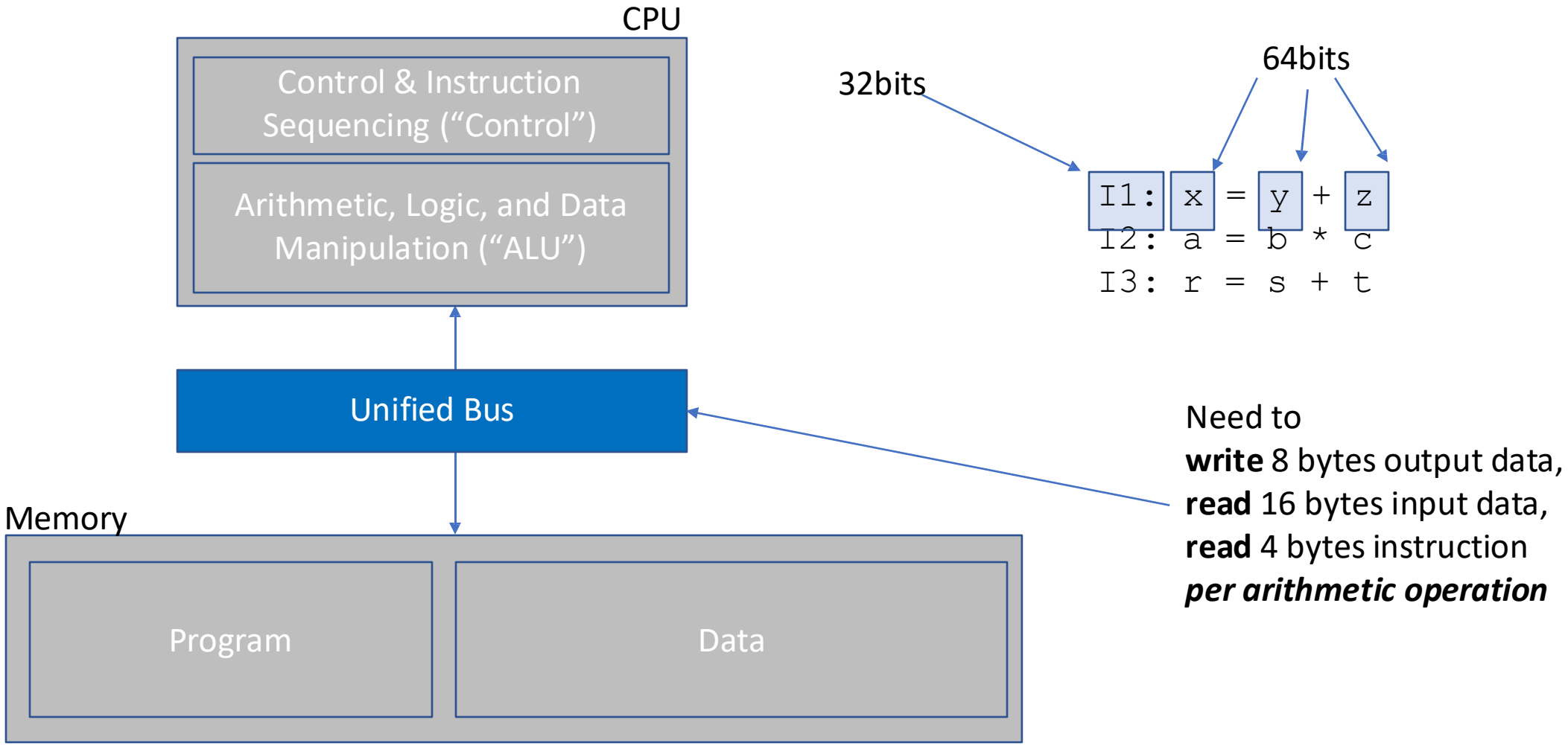
- Data & Program share a bus into the CPU. Need to time multiplex access to the bus.

The Von Neumann / Stored-Program Computing Model

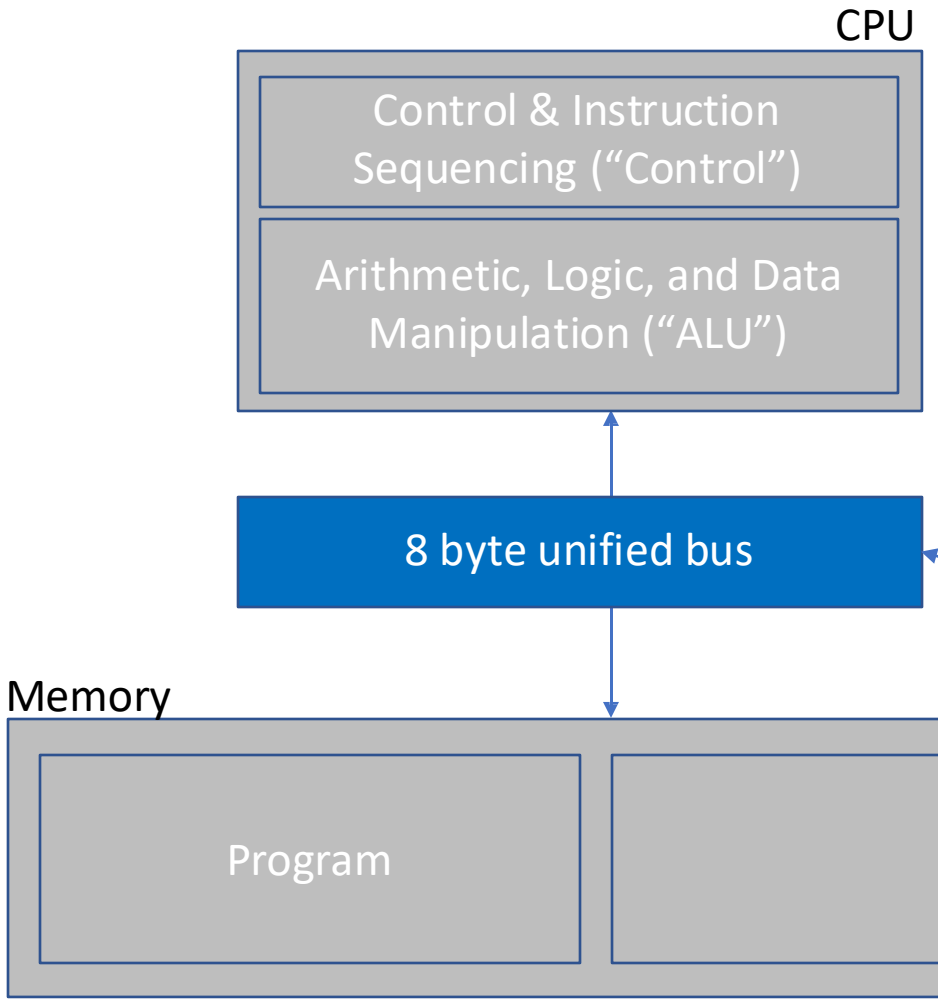


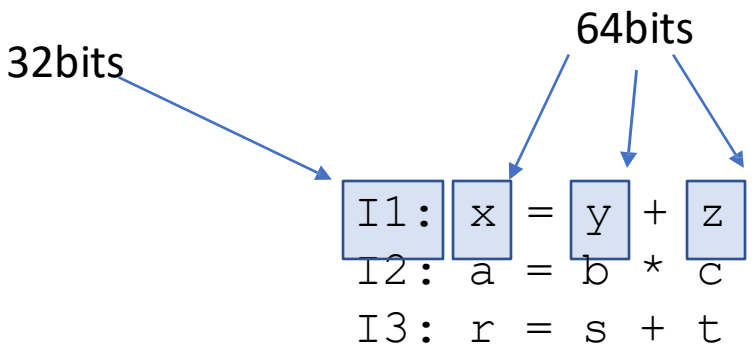

```
I1: x = y + z
I2: a = b * c
I3: r = s + t
```

The Von Neumann / Stored-Program Computing Model



The Von Neumann / Stored-Program Computing Model

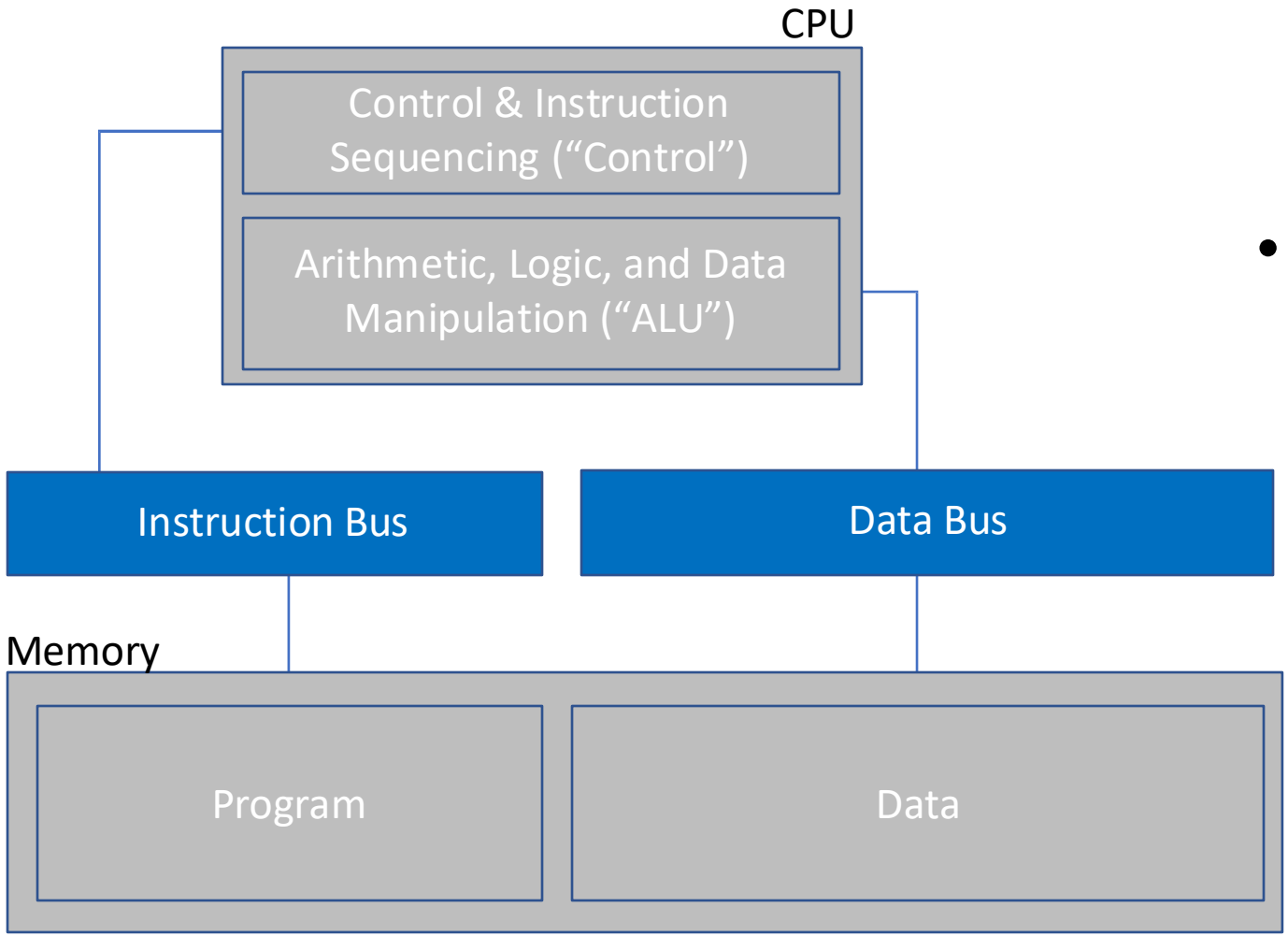




With a one word (8 byte) bus:
1 data write cycle,
2 data read cycles,
1 instruction read cycle
per arithmetic operation

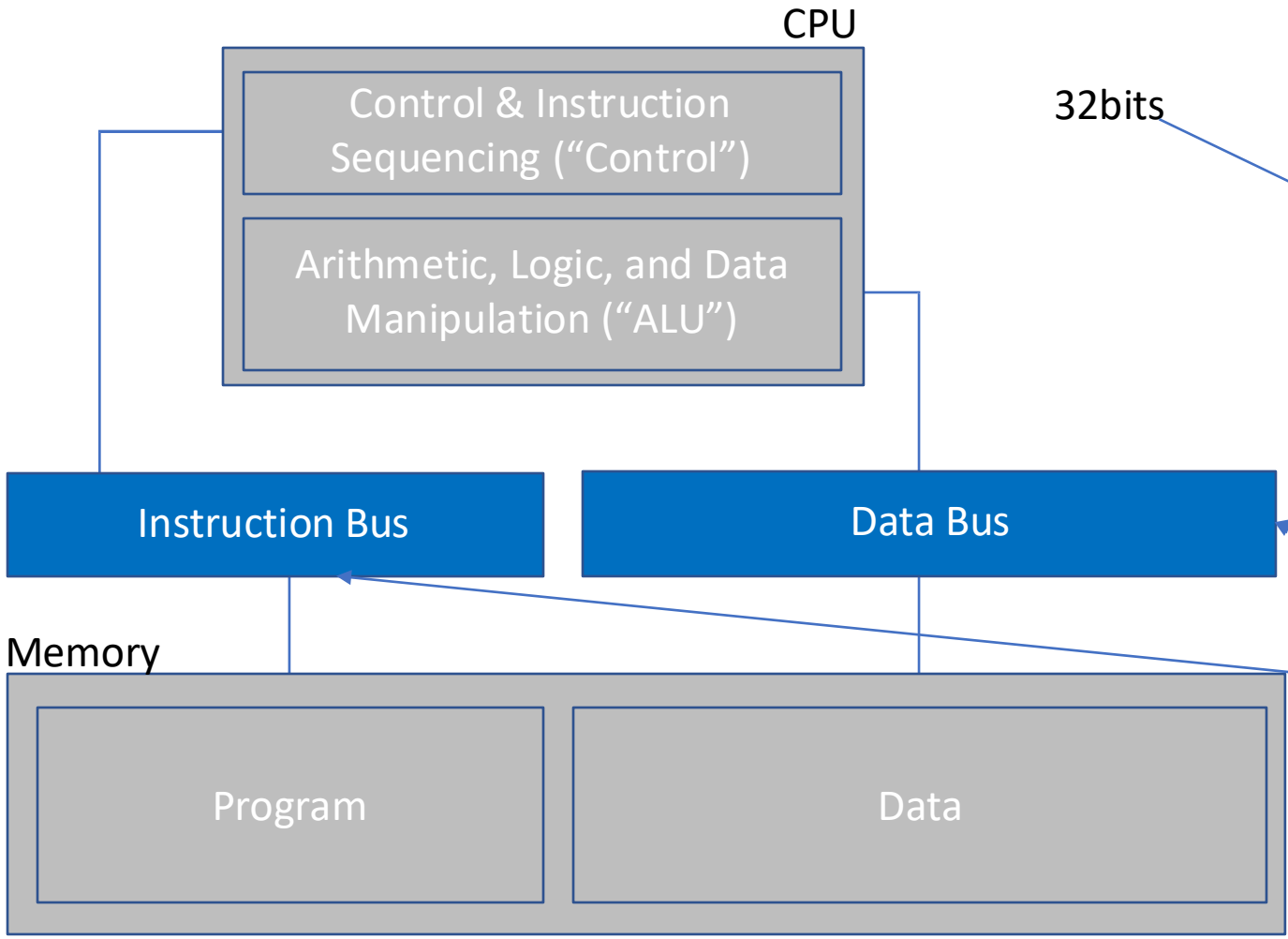
4 cycles per instruction!

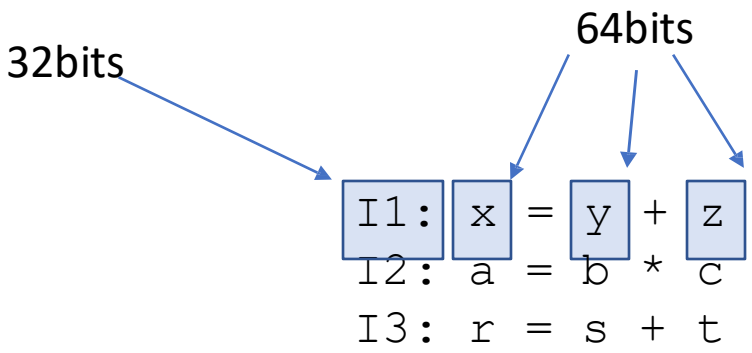
Alternative to von Neumann: the Harvard Architecture



- Split bus architecture provides *simultaneous* access to program and to data

Alternative to von Neumann: the Harvard Architecture

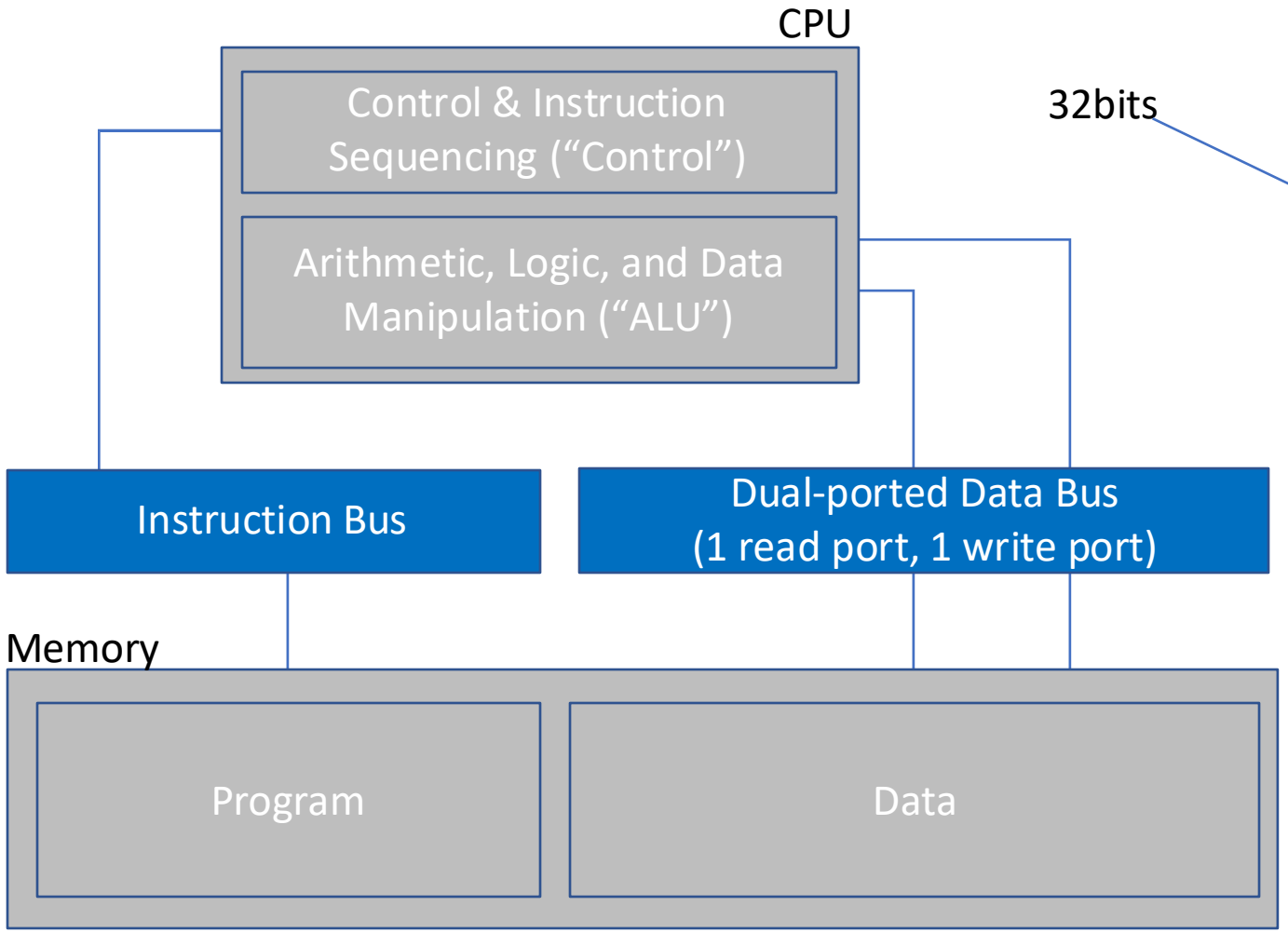


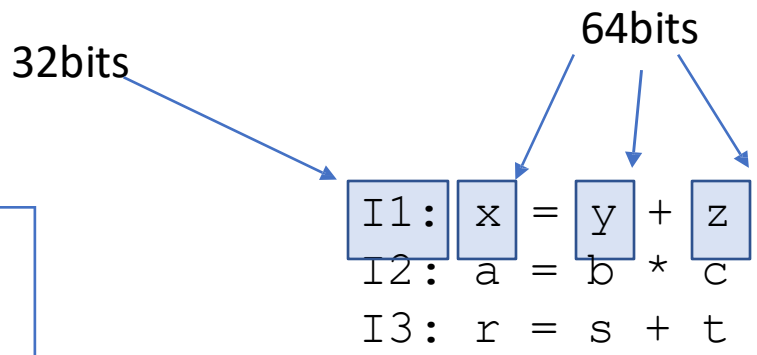


With a one word (8 byte) bus:
1 data write cycle,
2 data read cycles, simultaneous with **1 instruction read** cycle
per arithmetic operation

3 cycles per instruction!

Optimizing our Harvard Architecture

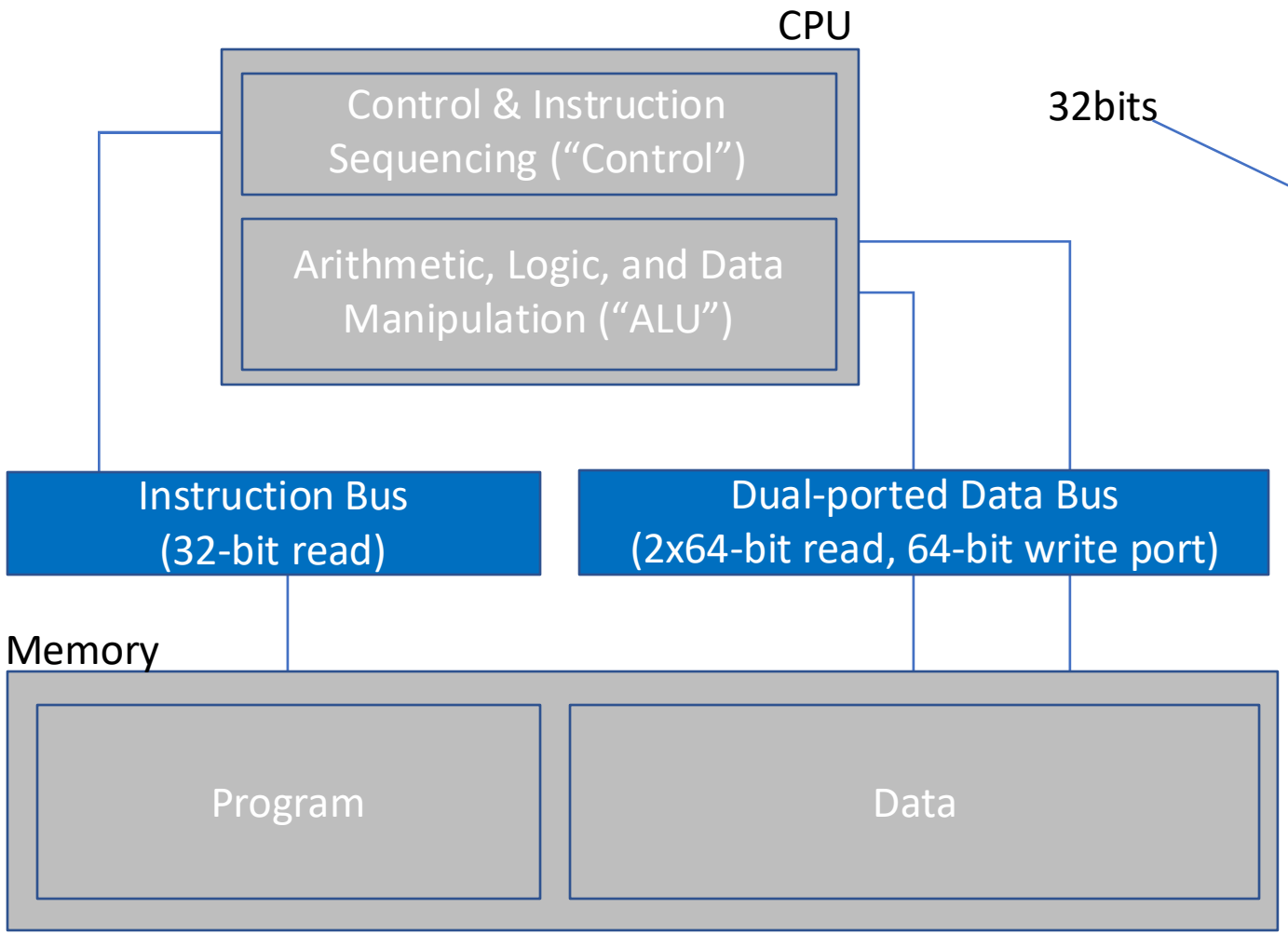


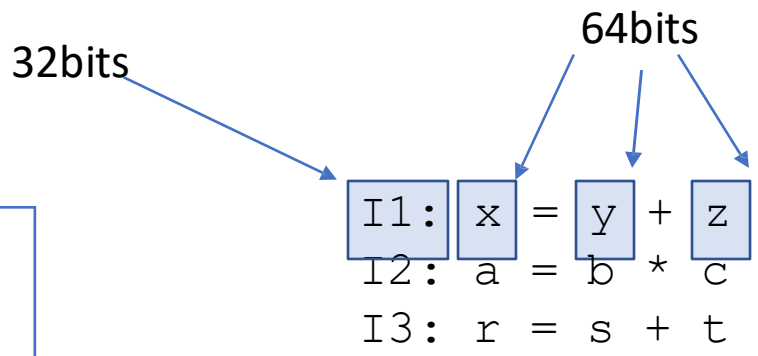


With a one word (8 byte) bus:
1 data write cycle simultaneous with **2 data read** cycles simultaneous with **1 instruction read** cycle
per arithmetic operation

2 cycles per instruction!

Optimizing our Harvard Architecture

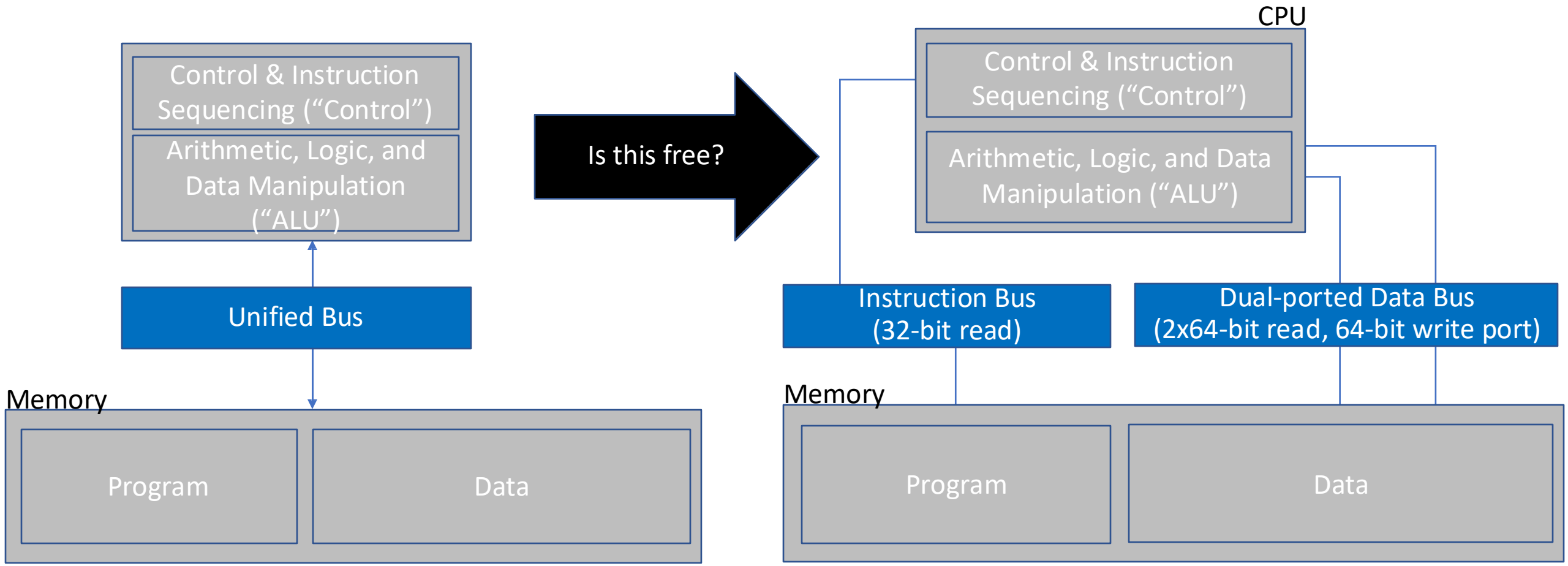




With a one word (8 byte) bus:
1 data write cycle simultaneous with **1 data read** cycles simultaneous with **1 instruction read** cycle
per arithmetic operation

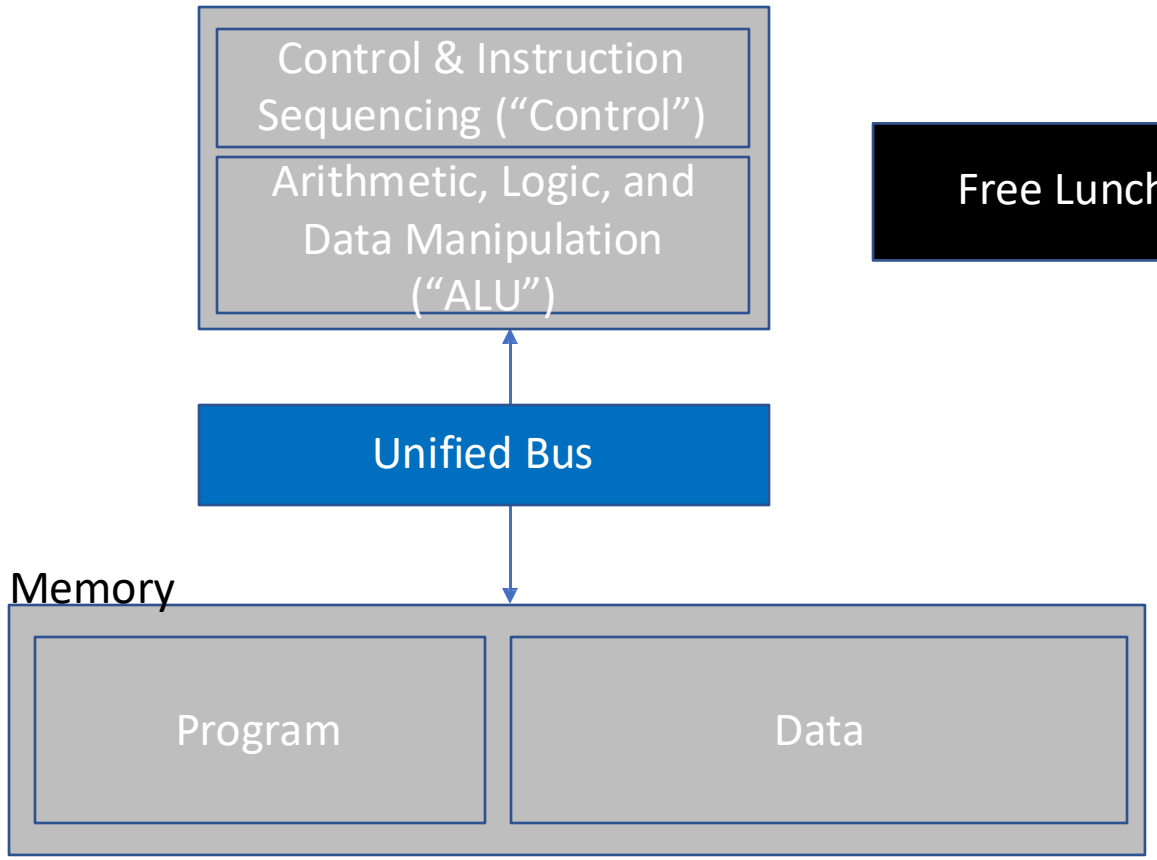
1 cycle per instruction!

Thinking about the costs of HW optimization



```
$ ./destiny config/SRAM_128_1_32.cfg
```

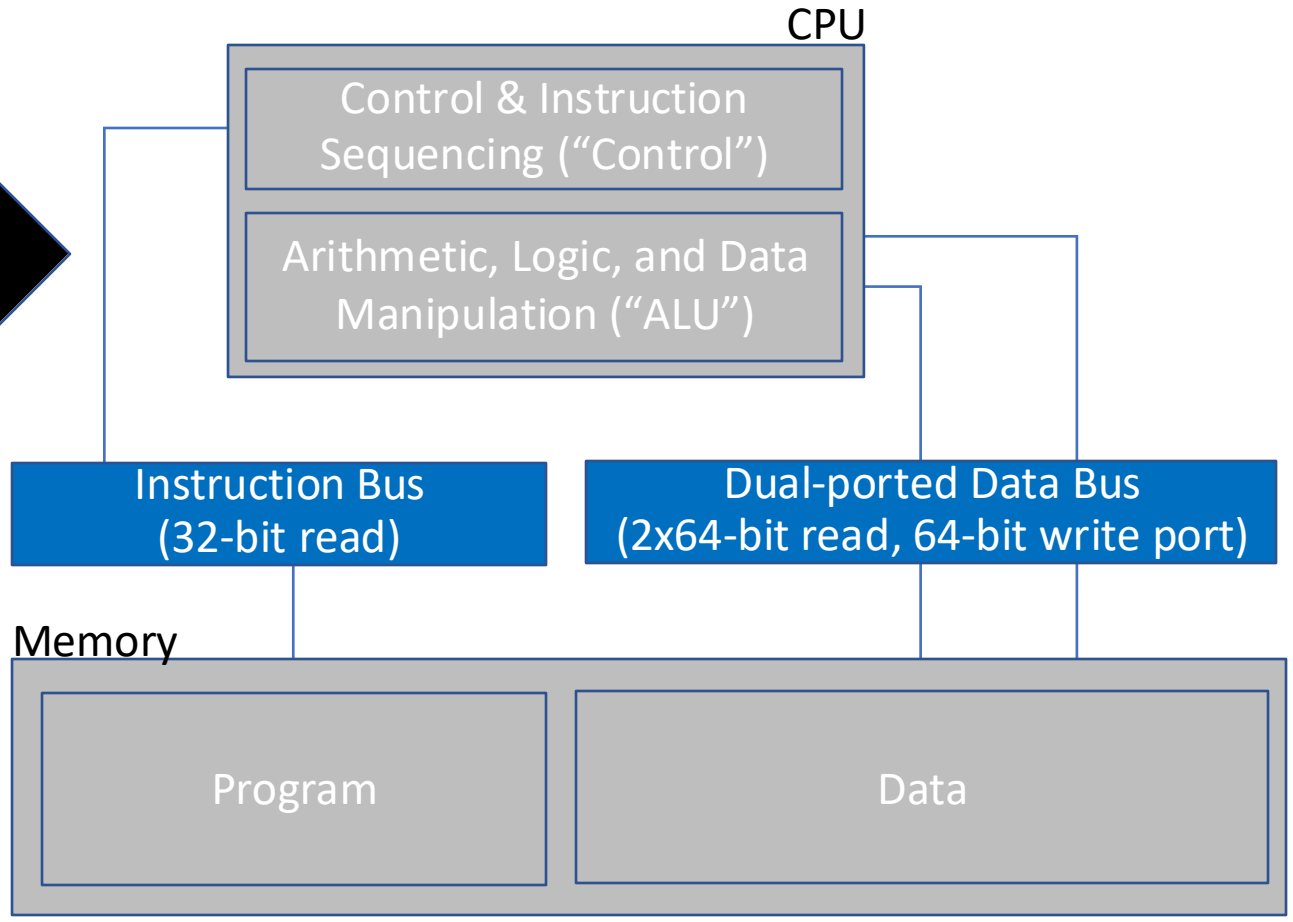
4-byte Memory Interface:
Read Latency = 0.289ns
Write Latency = 0.212ns



Free Lunch!

```
$ ./destiny config/SRAM_128_1_128.cfg
```

16-byte Memory Interface:
Read Latency = 0.289ns
Write Latency = 0.212ns

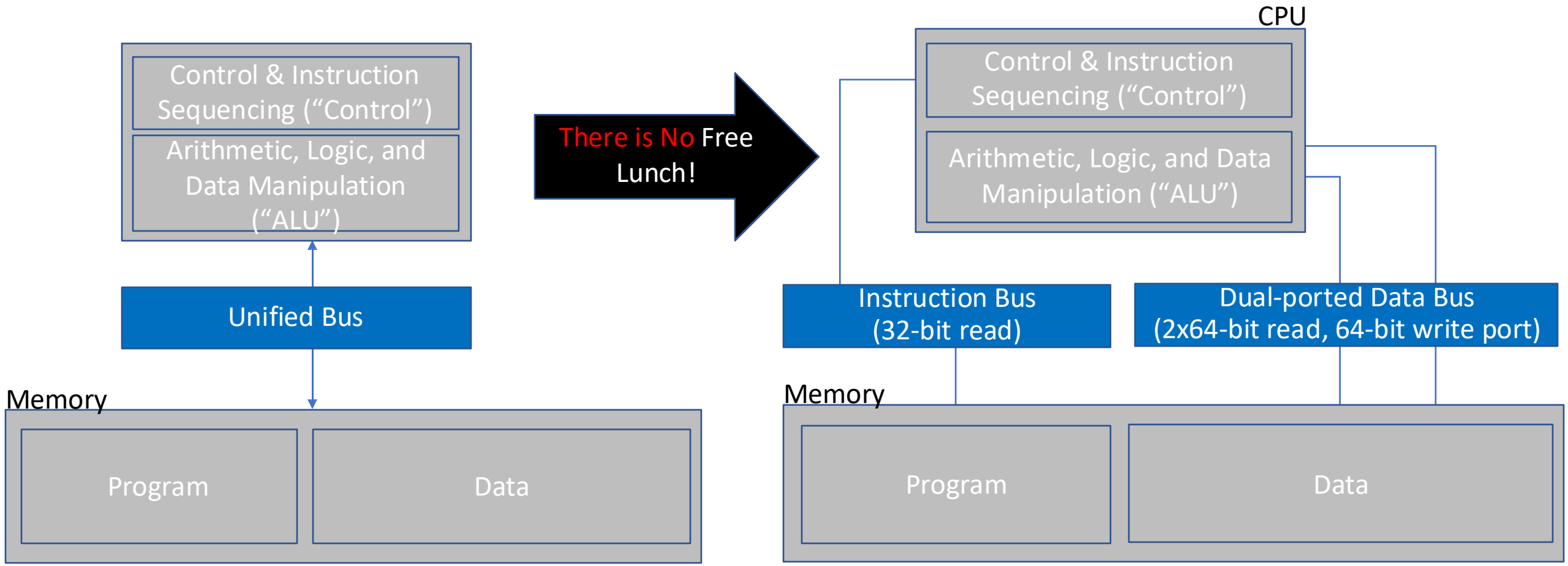


```
$ ./destiny config/SRAM_128_1_32.cfg
```

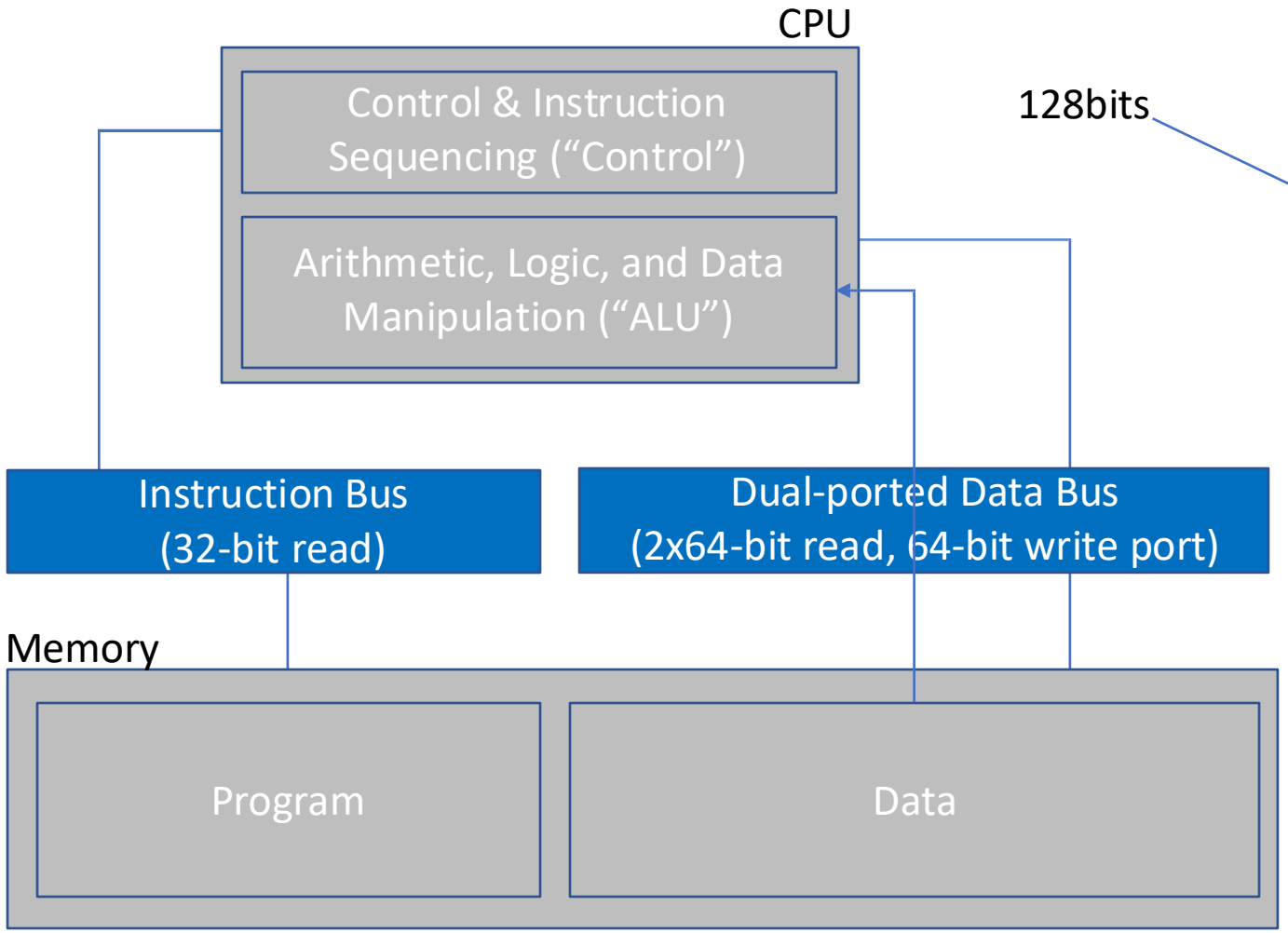
4-byte Memory Interface:
Read **Energy** = **0.836pJ**
Write **Energy** = **0.738pJ**

```
$ ./destiny config/SRAM_128_1_128.cfg
```

16-byte Memory Interface:
Read **Energy** = **1.51pJ**
Write **Energy** = **1.30pJ**



How about optimizing instruction supply?

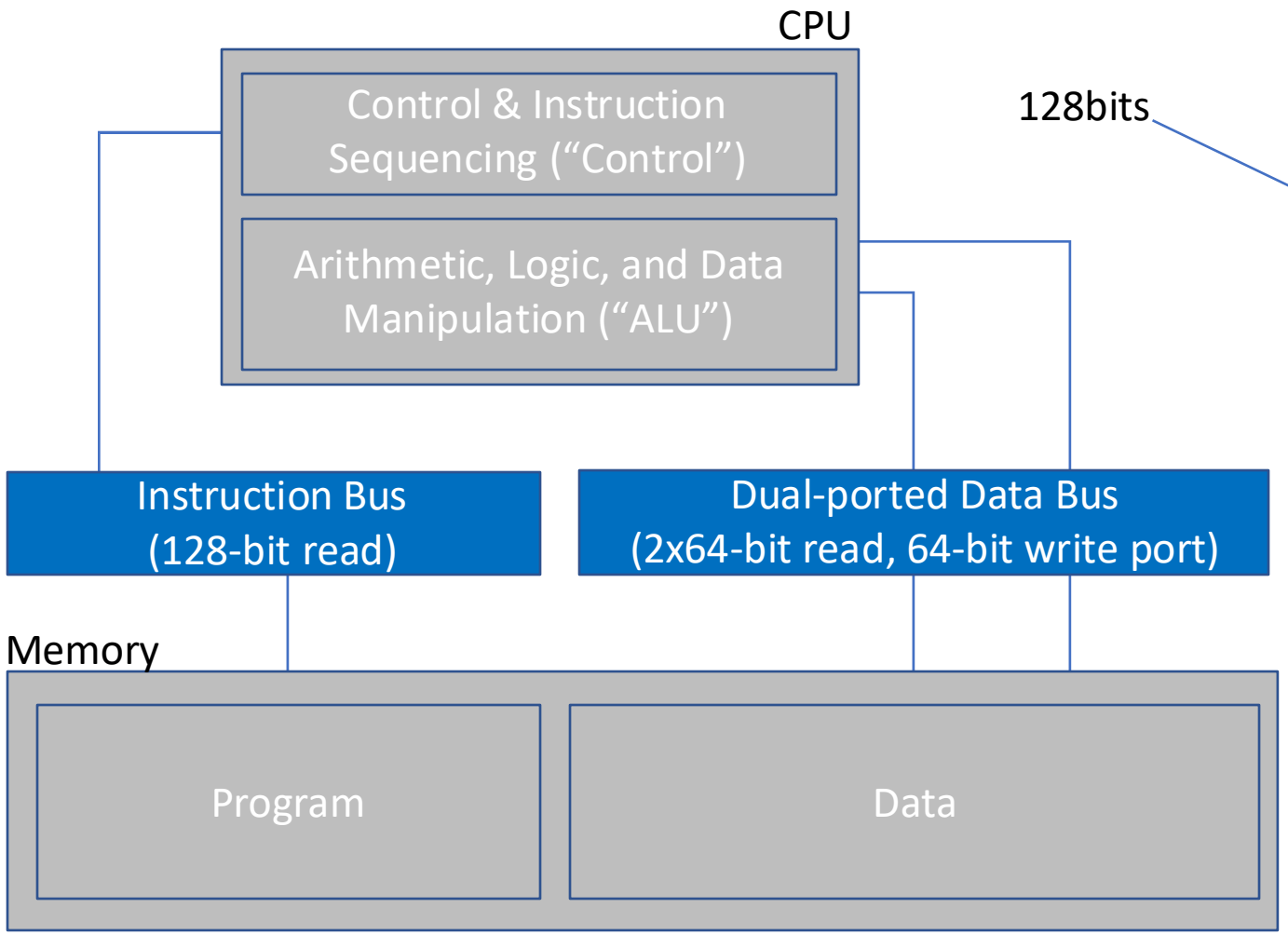


128bits

```
I1: x = y + z
I2: a = b * c
I3: r = s + t
I4: q = n + m
```

These four instructions take **four cycles** to fetch sequentially on our instruction bus

How about optimizing instruction supply?

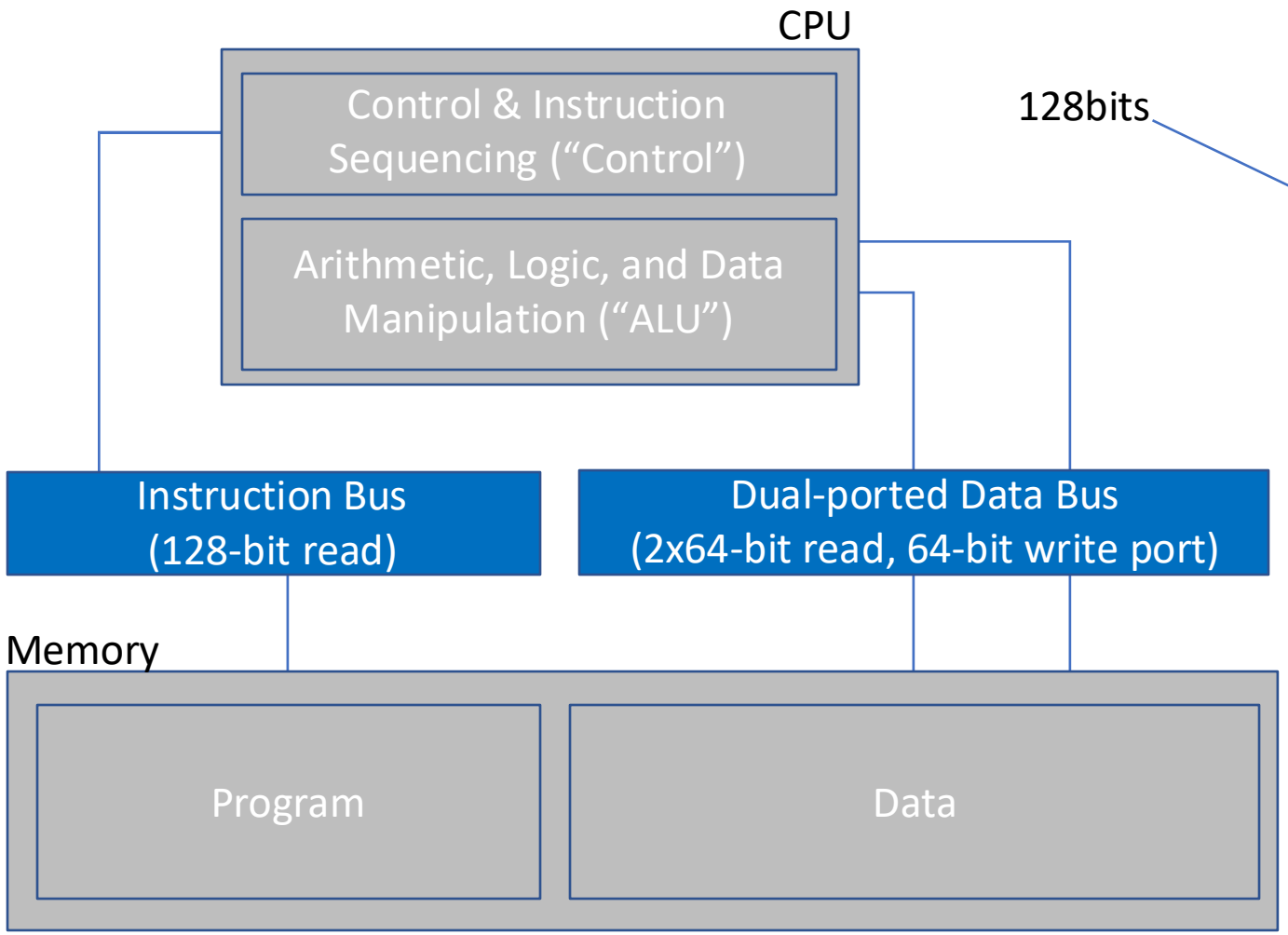


128bits

```
I1: x = y + z
I2: a = b * c
I3: r = s + t
I4: q = n + m
```

16 byte instruction bus:
1 instruction read cycle *every 4 operations*

Is this optimization a good tradeoff?



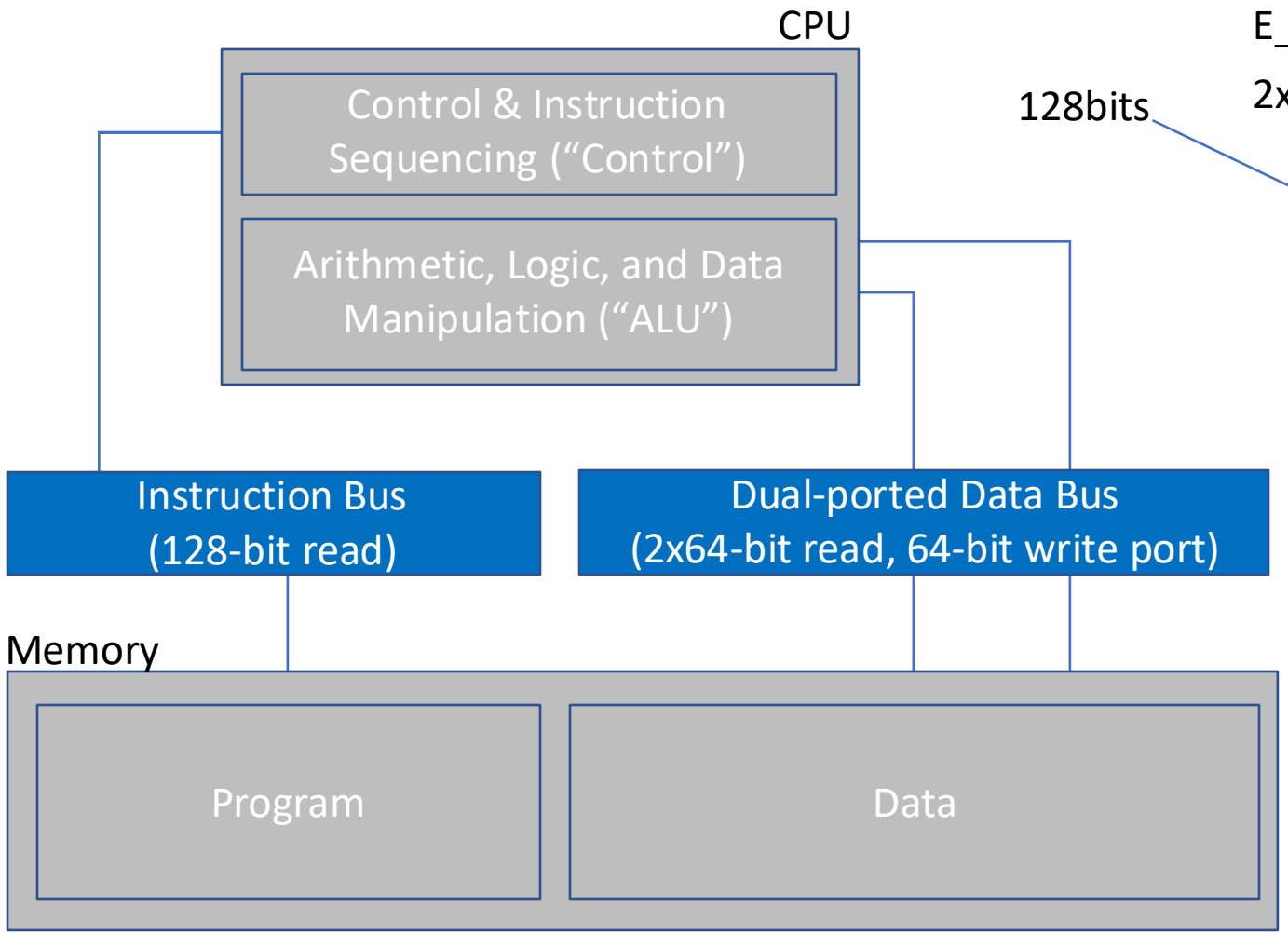
128bits

```
I1: x = y + z
I2: a = b * c
I3: r = s + t
I4: q = n + m
```

16 byte instruction bus:
1 instruction read cycle every 4 operations

4B Read Energy = 0.836pJ
16B Read Energy = 1.51pJ

Is this optimization a good tradeoff?



E_old = n x EpF

E_new = (n/4) x (EpF x 2) = ½(n x EpF)

2x savings in total instruction fetch energy!

128bits

```
I1: x = y + z
I2: a = b * c
I3: r = s + t
I4: q = n + m
```

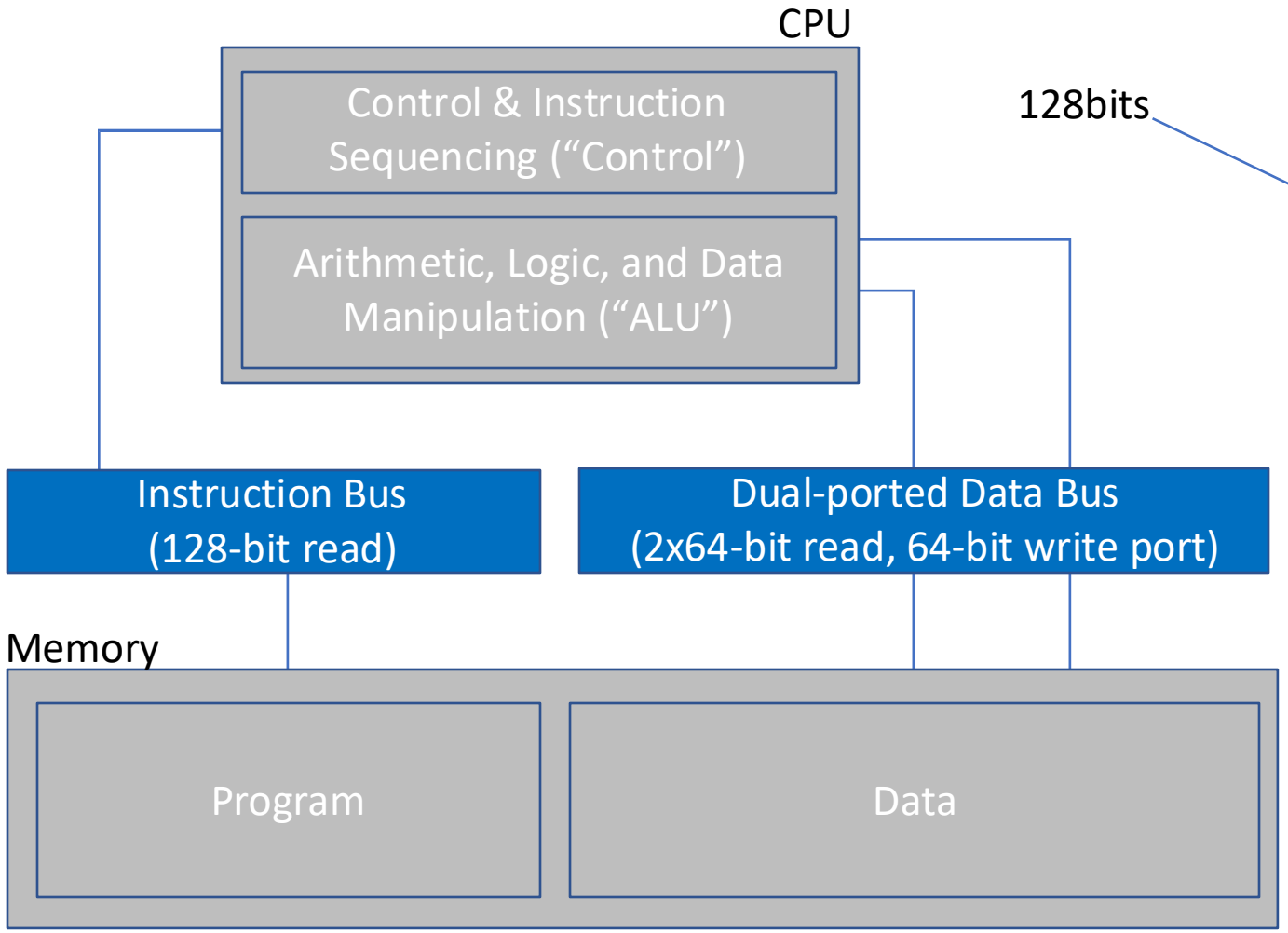
16 byte instruction bus:
1 instruction read cycle every 4 operations
4x fewer instruction fetches

4B Read Energy = 0.836pJ

16B Read Energy = 1.51pJ

~2x more energy / fetch

Is this optimization a good tradeoff?



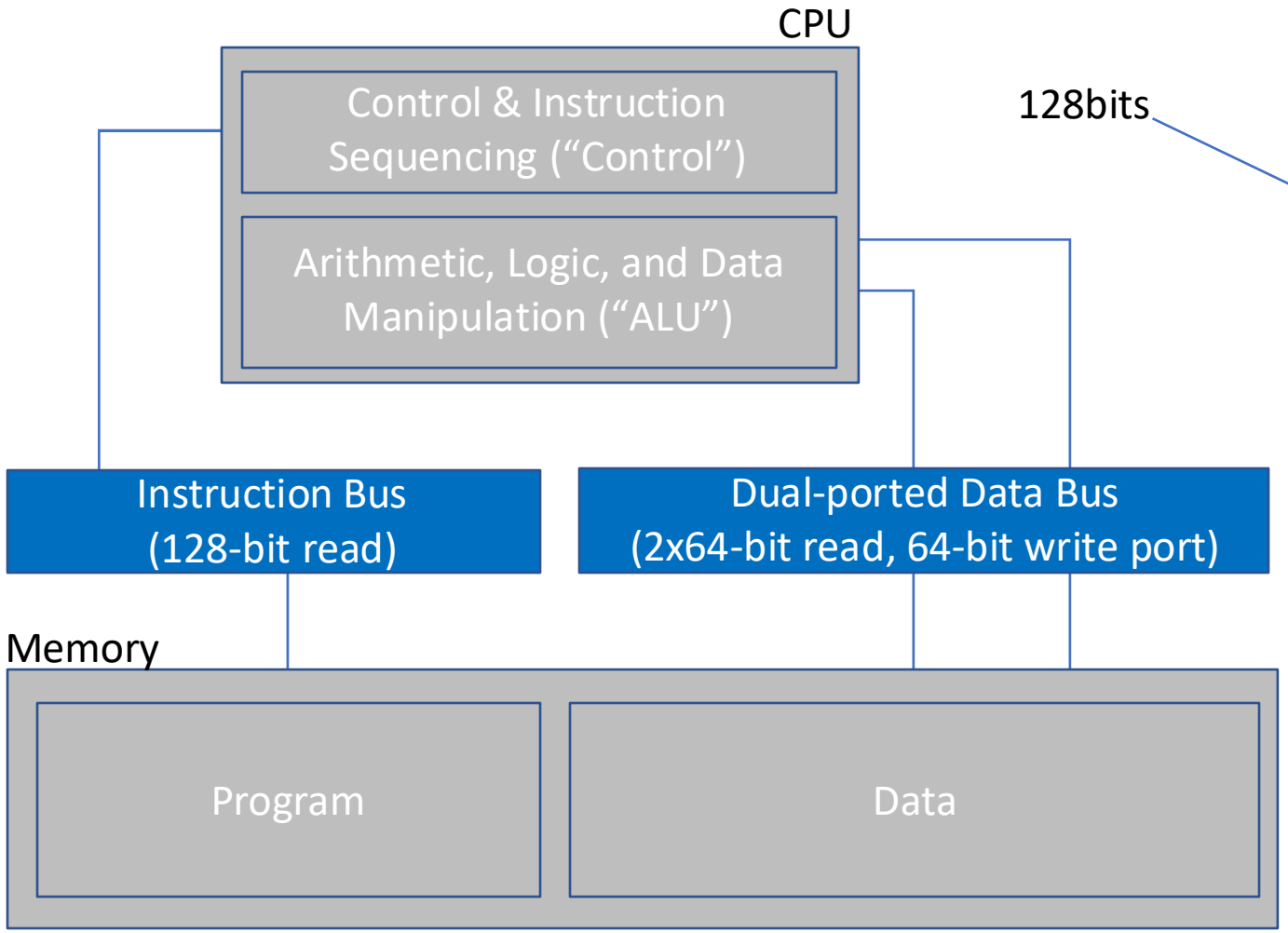
When might this optimization *not* pay off?

```
I1: x = y + z
I2: a = b * c
I3: r = s + t
I4: q = n + m
```

16 byte instruction bus:
1 instruction read cycle every 4 operations
4x fewer instruction fetches

4B Read Energy = 0.836pJ
16B Read Energy = 1.51pJ
~2x more energy / fetch

Is this optimization *always* a good tradeoff?



128bits

```
I1: if (x==0)
I2:   a = b * c
I3:   y = z + w
I4:   q = n + m
```

1 instruction read cycle every 4 operations

4x fewer reads, ***if we execute the operations!***

OR: 4x more useless fetching if x!=0

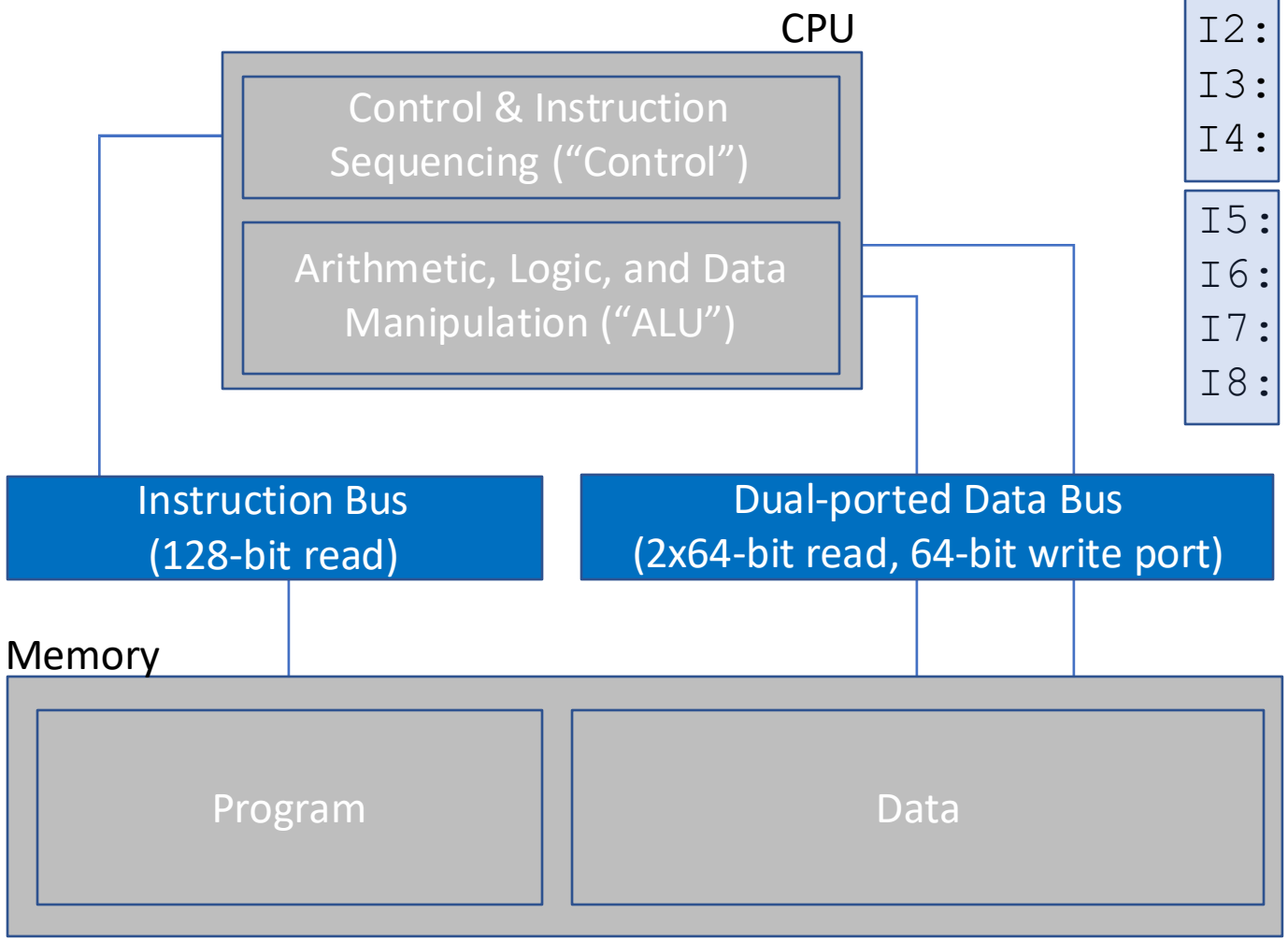
(more on this branching topic in a future lecture)

4B Read Energy = 0.836pJ

16B Read Energy = 1.51pJ

~2x more energy / fetch

How about changing the code?




```
I1: if (x==0)
I2:  a = b * 2
I3:  y = z + w
I4:  q = n + m
I5: //else
I6:  a = b * 4
I7:  y = z + w
I8:  q = n + m
```

Compiler!

```
I1: y = z + w
I2: q = n + m
I3: if (x==0)
I4: a = b * 2
I5: //else
I6:  a = b * 4
I7: //other
I8: //stuff
```

Restructure code, increase likelihood to execute

Avoid needless fetch of non-executing code

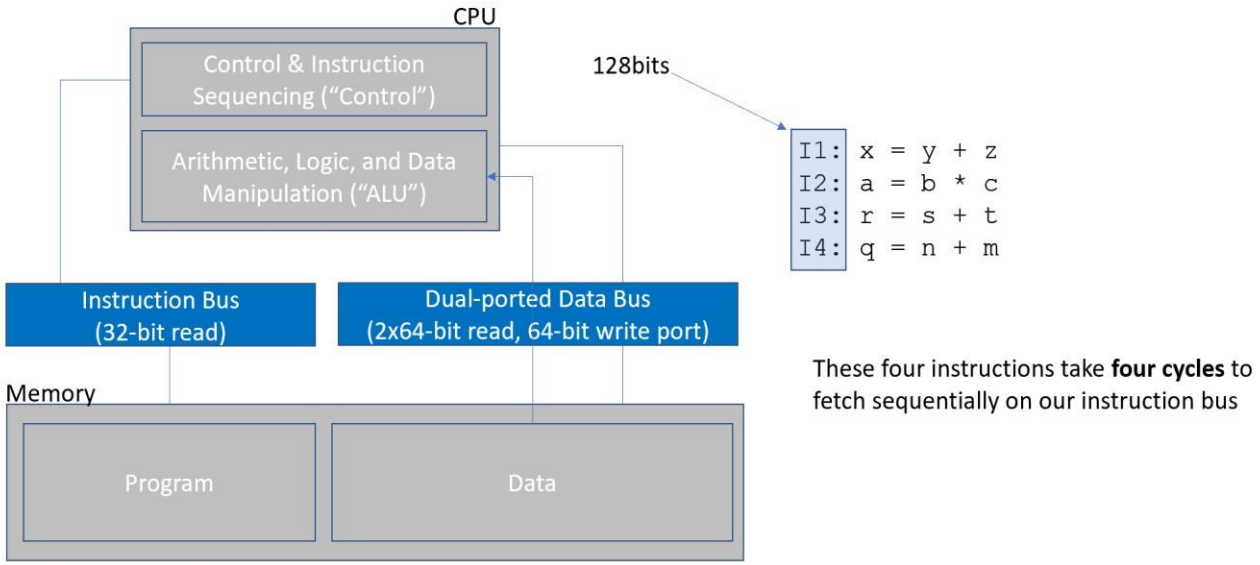
Q: What if x is most often non-zero?

A key Law of the HW/SW Universe

Amdahl's Law

Let's revisit the proposition that we should optimize instruction supply

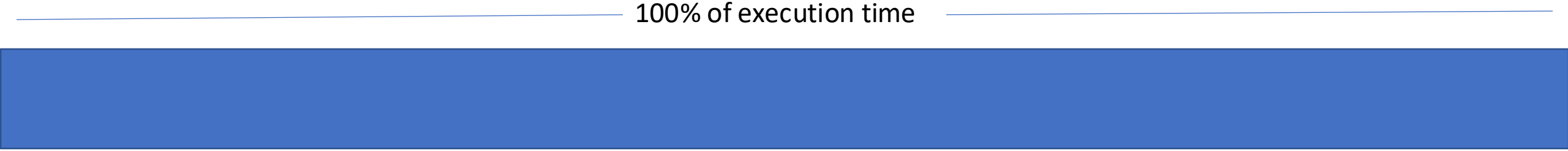
How about optimizing instruction supply?



These four instructions take **four cycles** to fetch sequentially on our instruction bus

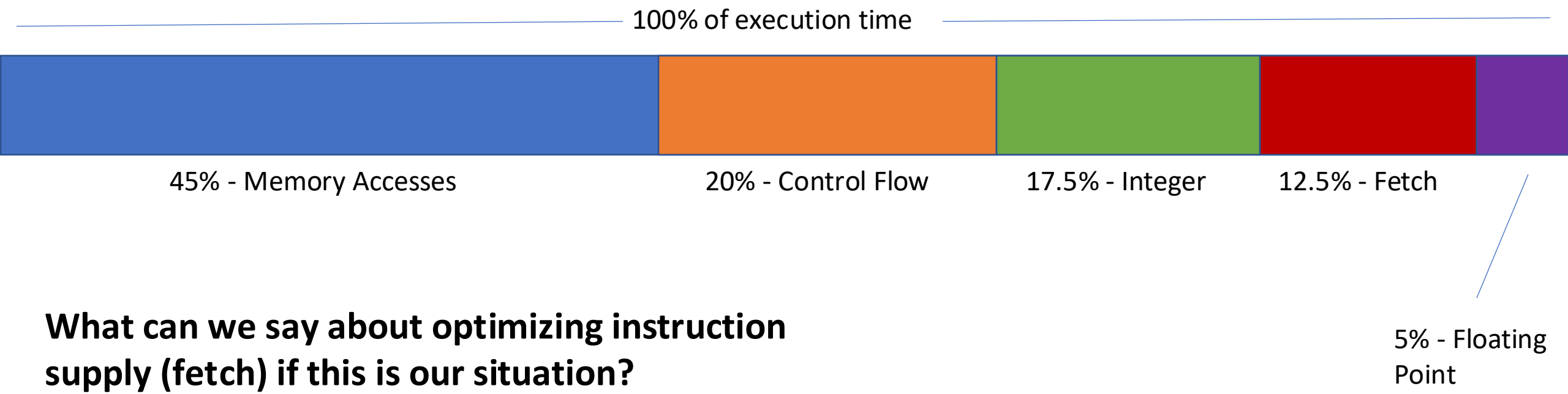
How do we decide if this part of the system is really worth optimizing?

Amdahl’s Law



Amdahl's Law

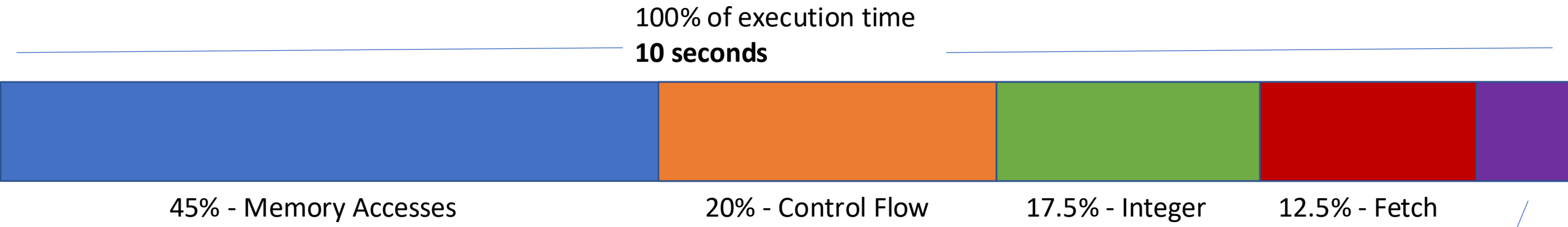
Imagine we have a perfectly precise measurement tool to break down execution time…



What can we say about optimizing instruction supply (fetch) if this is our situation?

Amdahl's Law

What if we make fetch 4x faster?

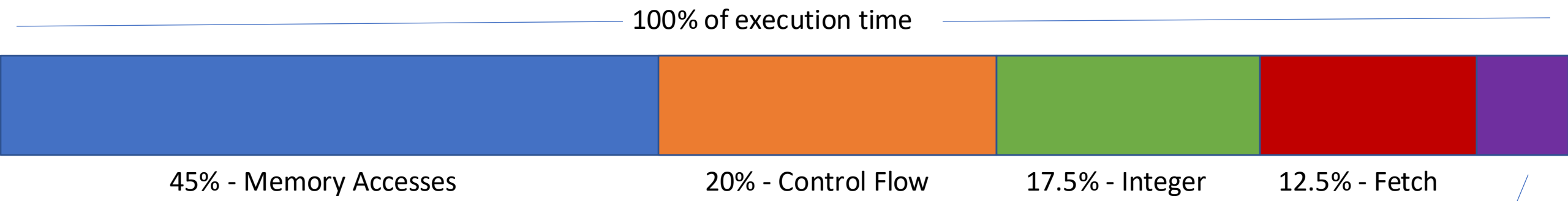


We have 45% + 20% + 17.5% + 5% = 87.5% of the execution running at its original speed (ie 87.5% of 10 seconds = 8.75s)

We have 12.5% of the execution running 4x faster. Originally, we had 12.5% of 10 units of time = 1.25s. If 4x faster, fetch takes 0.31s. Savings of 0.94s

Amdahl's Law

What if we make fetch 4x faster?



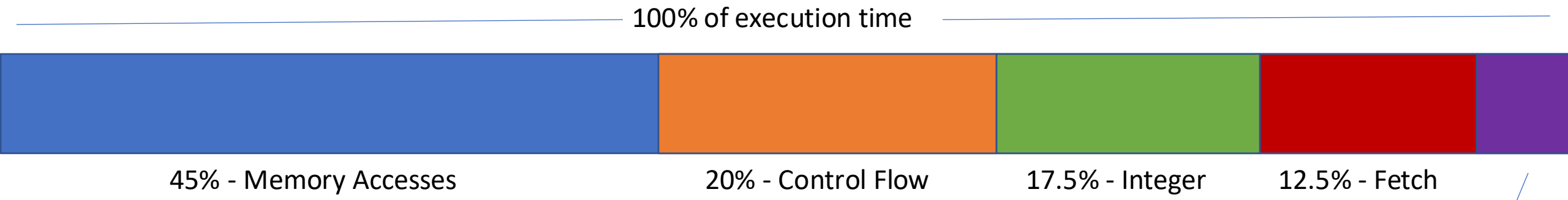
(0.875 x 10)/1 + (0.125 x 10)/4 = 9.06

What does 0.906 mean?

Amdahl's Law

What if we make fetch 4x faster?

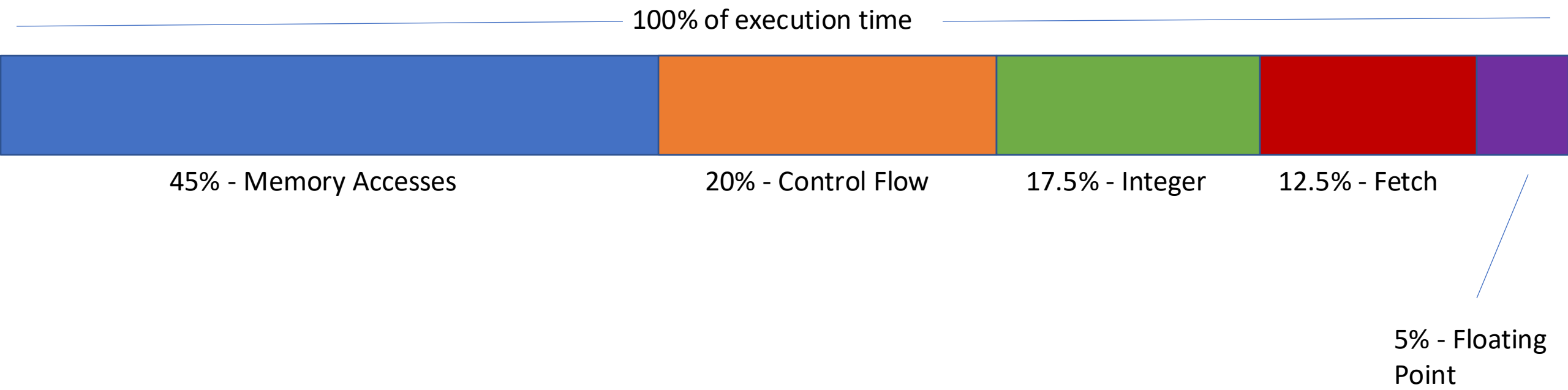
<10% improvement overall, despite 4x improvement on one part of the system!



4x improvement in fetch means execution takes 9.0625s instead of 10s
Saved 0.94s overall. 0.94s / 10s * 100 = 9.4% time savings

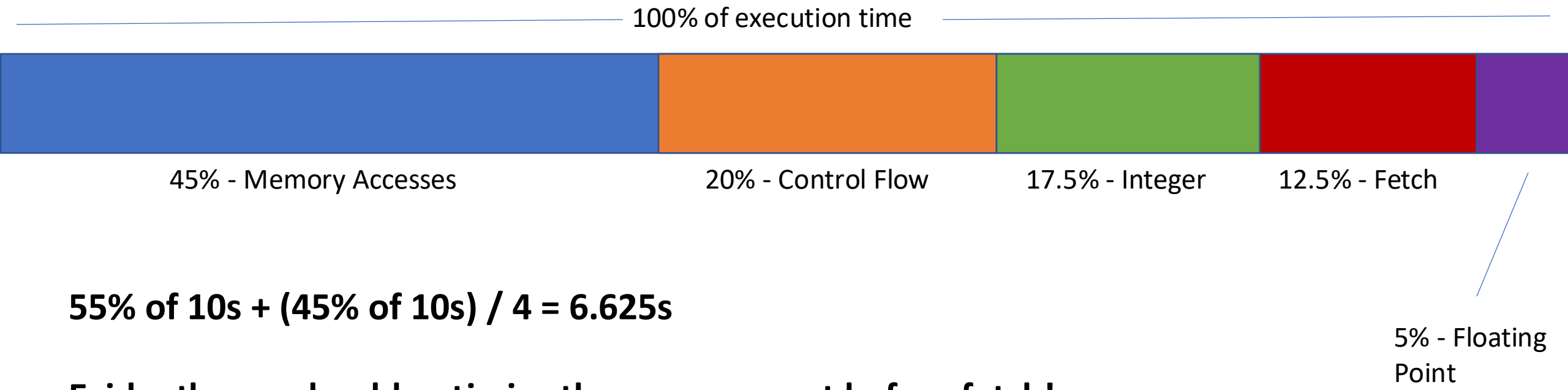
Amdahl's Law

What if we make memory accesses 4x faster?



Amdahl's Law

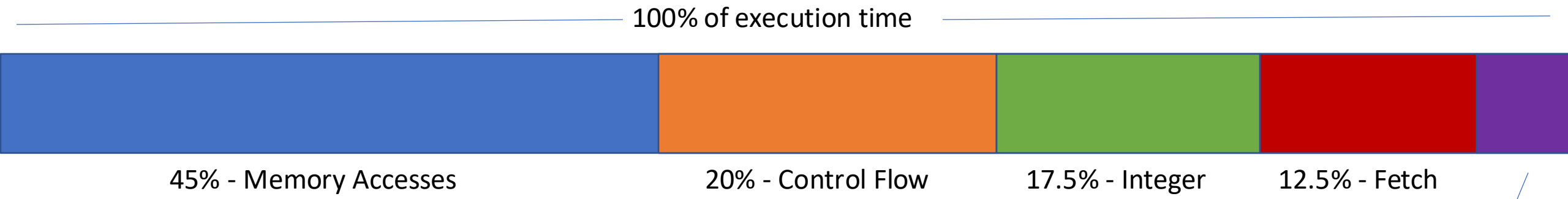
What if we make memory accesses 4x faster?



55% of 10s + (45% of 10s) / 4 = 6.625s

Evidently, we should optimize the memory part before fetch!

Amdahl's Law

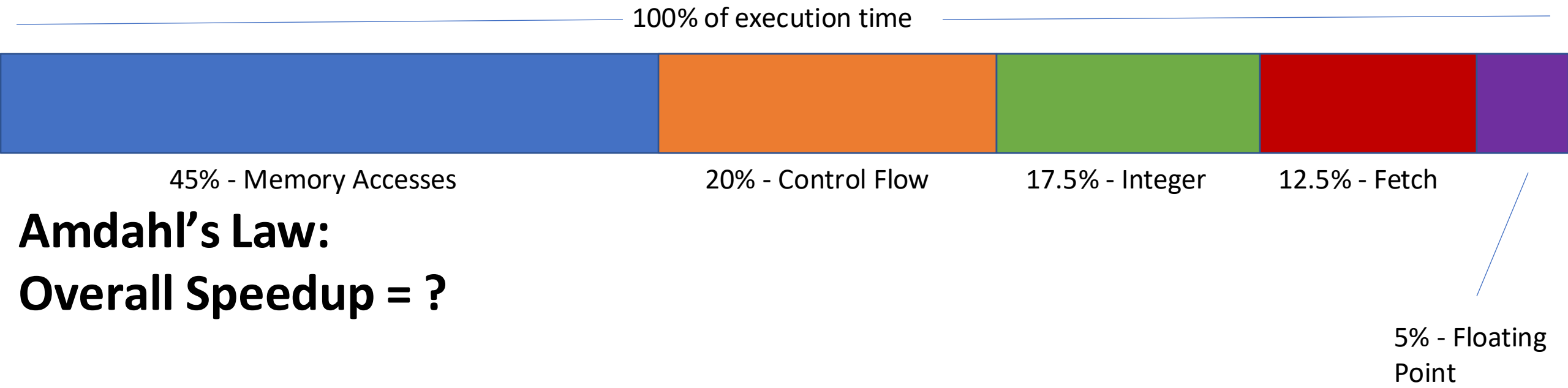


Amdahl's Law:

if fraction *p* can be optimized:

Optimized Time = [(1-p) * t / 1] + [(p * t) / speedup]

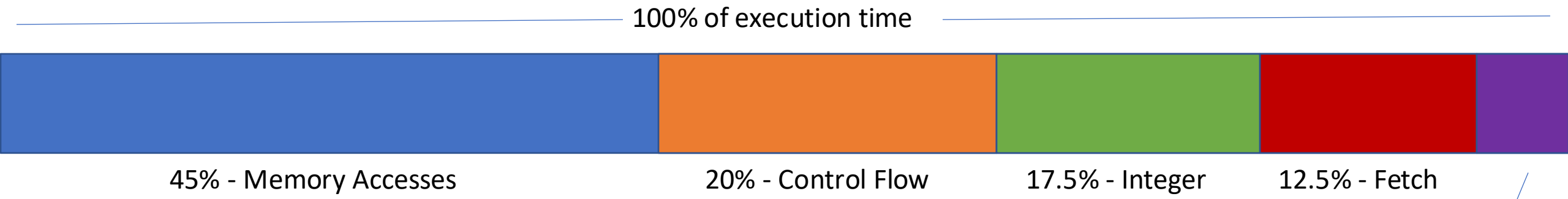
Amdahl's Law



Amdahl's Law:

Overall Speedup = ?

Amdahl's Law



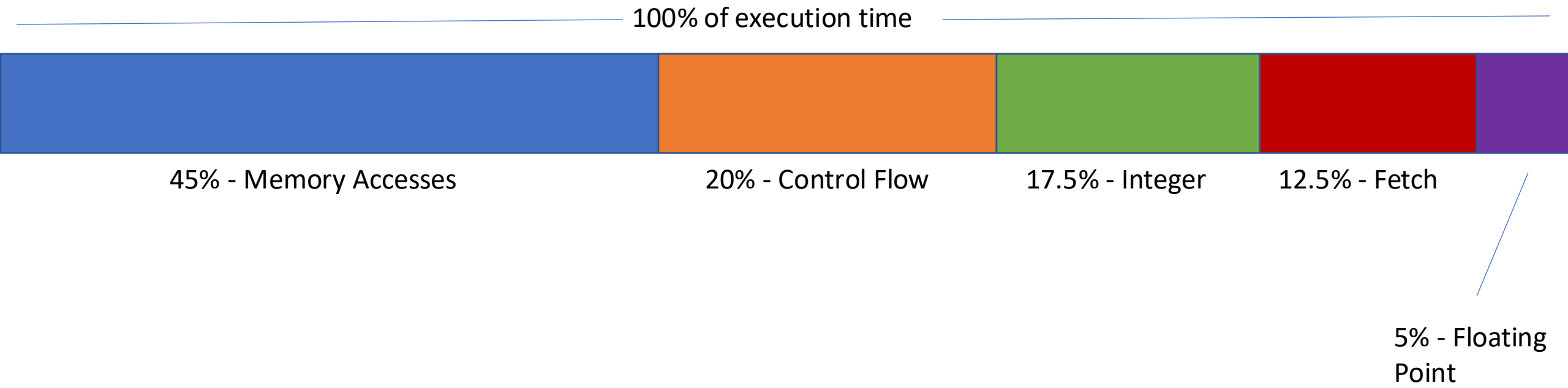
Amdahl's Law:

Overall Speedup = T_orig / T_opt

Overall Speedup = 1 / (1 – p + p / s)

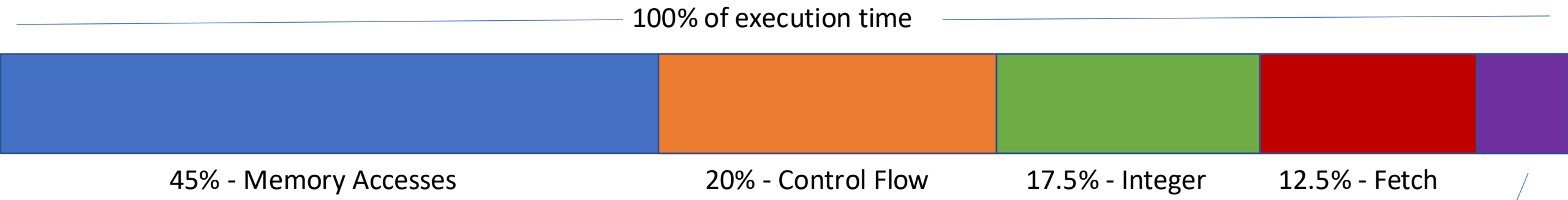
Amdahl's Law

By how much do we have to improve the memory part of the system to get a 2x total speedup?



Amdahl's Law

By how much do we have to improve the memory part of the system to get a 2x total speedup?



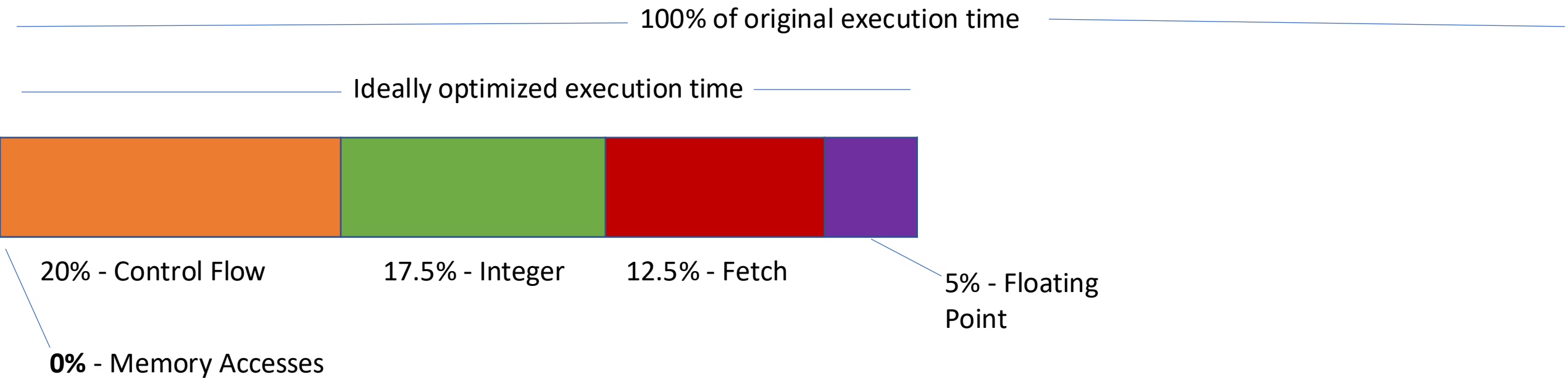
0.55 x 10s + (0.45 x 10s)/*speedup* = 10s / 2 = 5s

4.5s / speedup = 5s – 5.5s

speedup = -9x

Well that's a strange amount by which to speed up a program… conclusion?

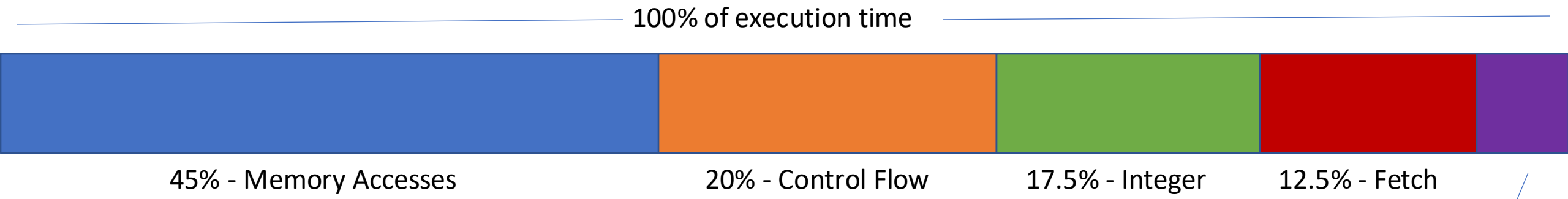
Limit Cases for Amdahl's Law



What if we *completely* optimize away the memory part?

5.5s optimized execution time

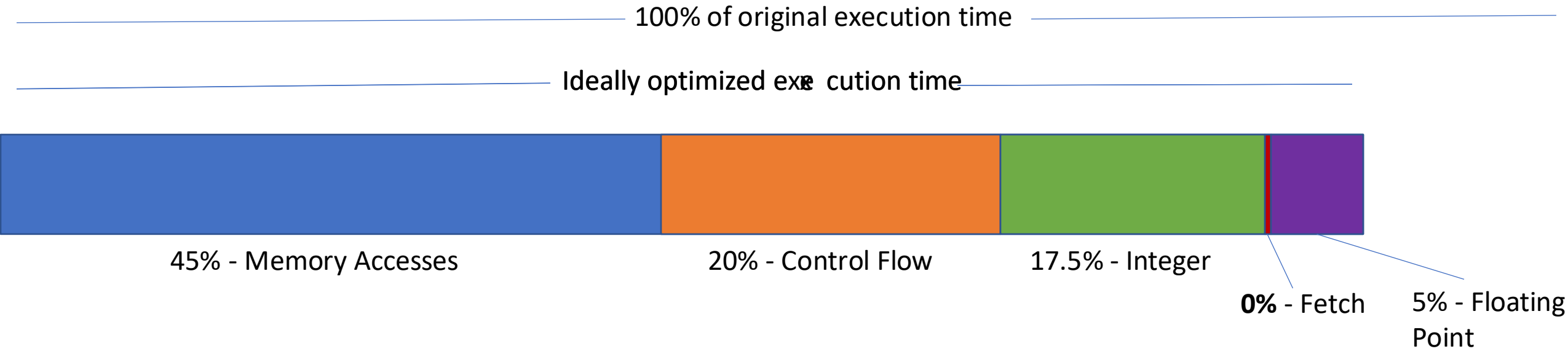
Limit Cases for Amdahl's Law



Let's try again:

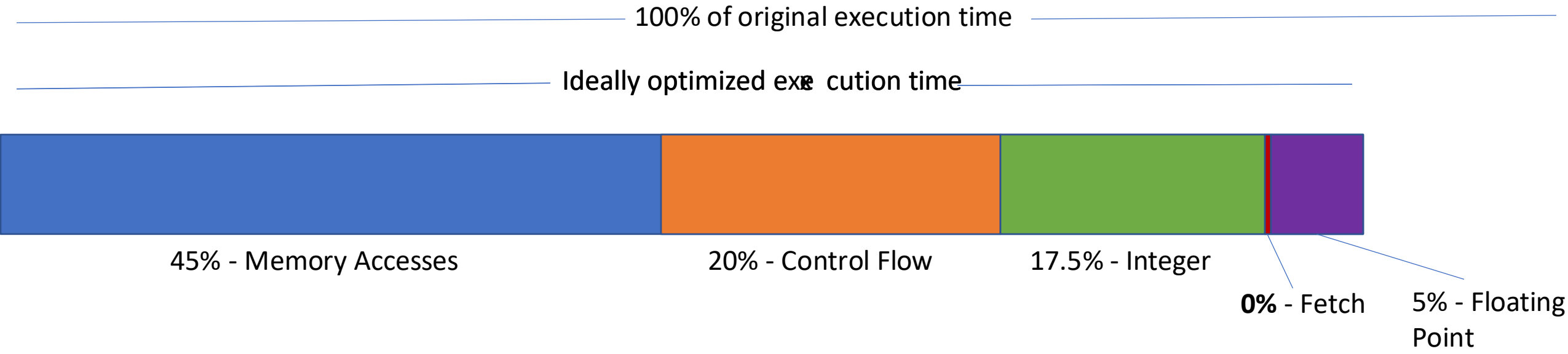
What if we *completely* optimize away the optimizable part?

Limit Cases for Amdahl's Law



What if we *completely* optimize away the optimizable part? (How much is left over here?)

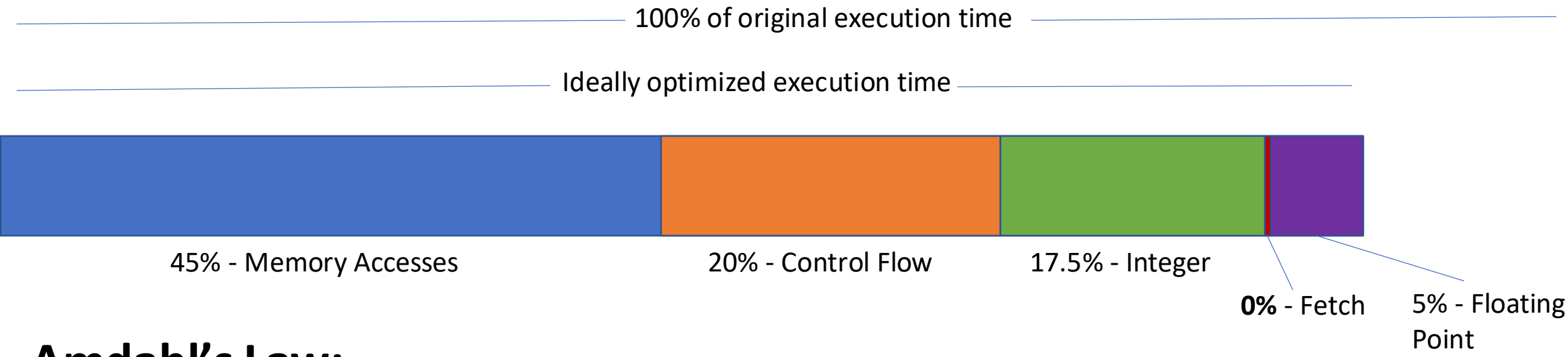
Limit Cases for Amdahl's Law



What if we *completely* optimize away the optimizable part?

8.75s optimized execution time

Limit Cases for Amdahl's Law

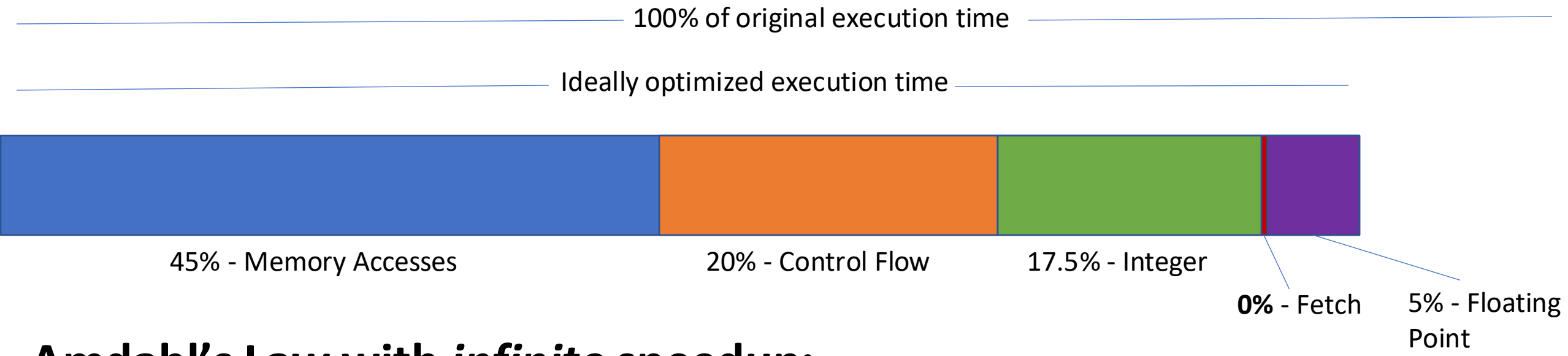


Amdahl's Law:

optimized time = [1-p x time / 1.0] + [p x time / *s*]

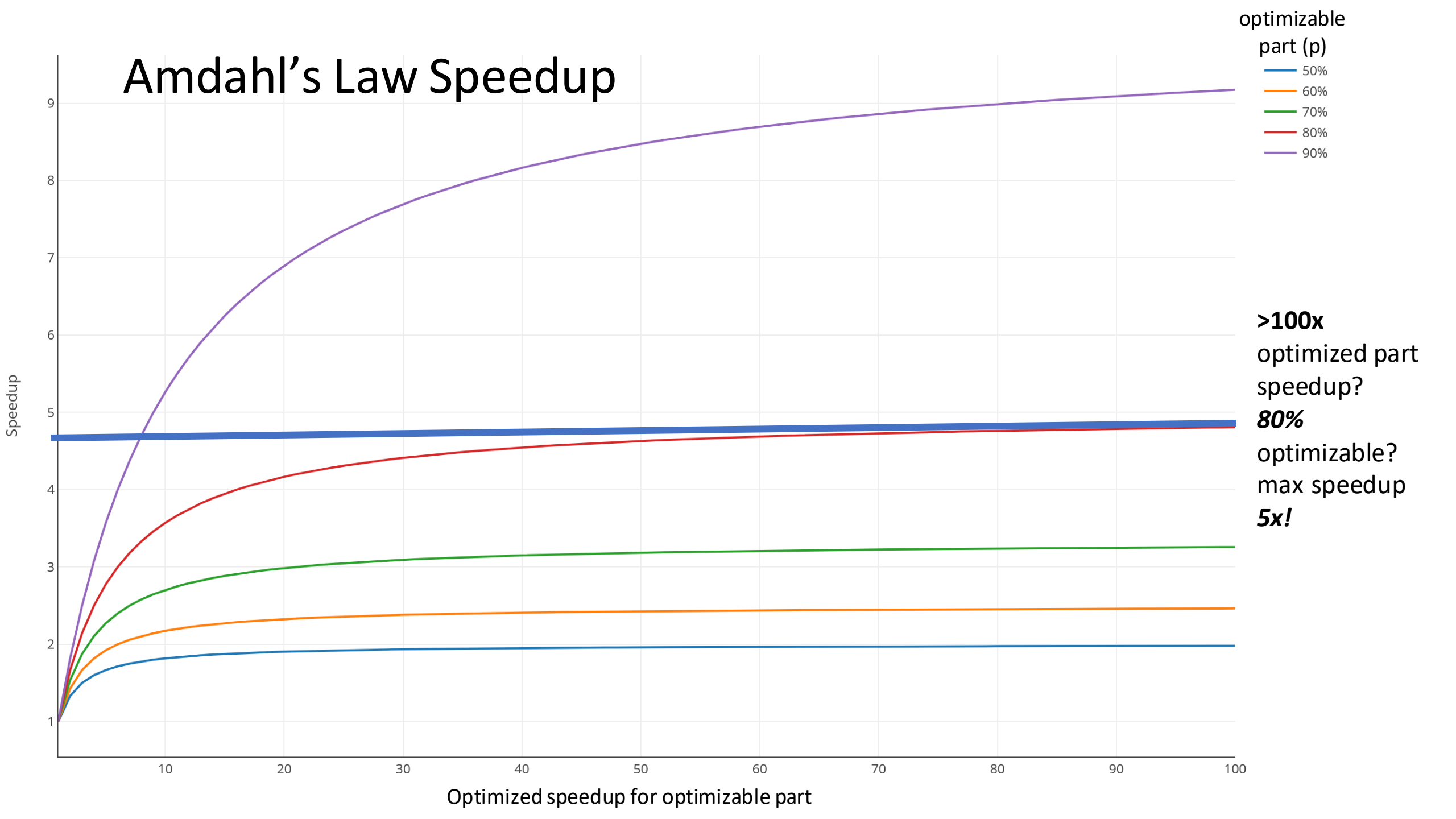
Overall speedup = 1 / (1 - p + p / s)

Limit Cases for Amdahl's Law



Amdahl's Law with *infinite* speedup:
optimized time with infinite speedup of *p* = [1-p x time / 1.0]
Overall speedup with infinite speedup of *p* = 1 / (1 - p)

Amdahl's Law Speedup

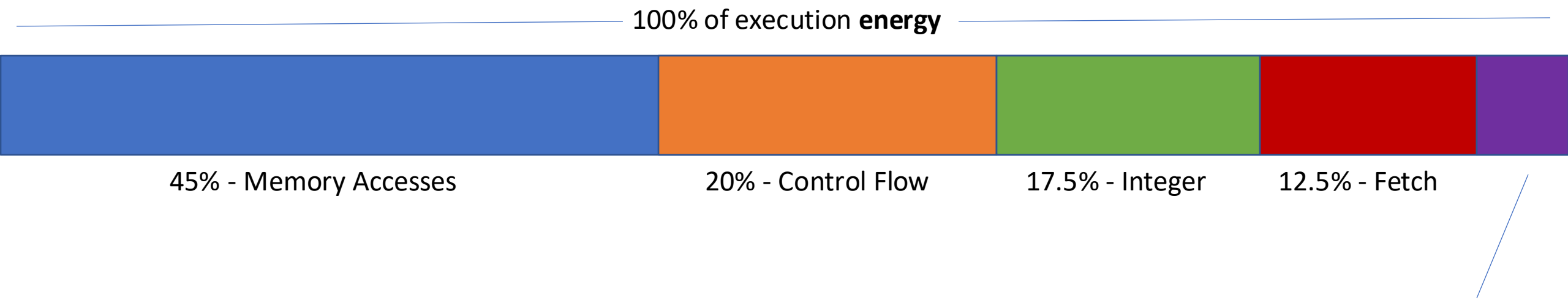


>100x optimized part speedup?

80% optimizable?

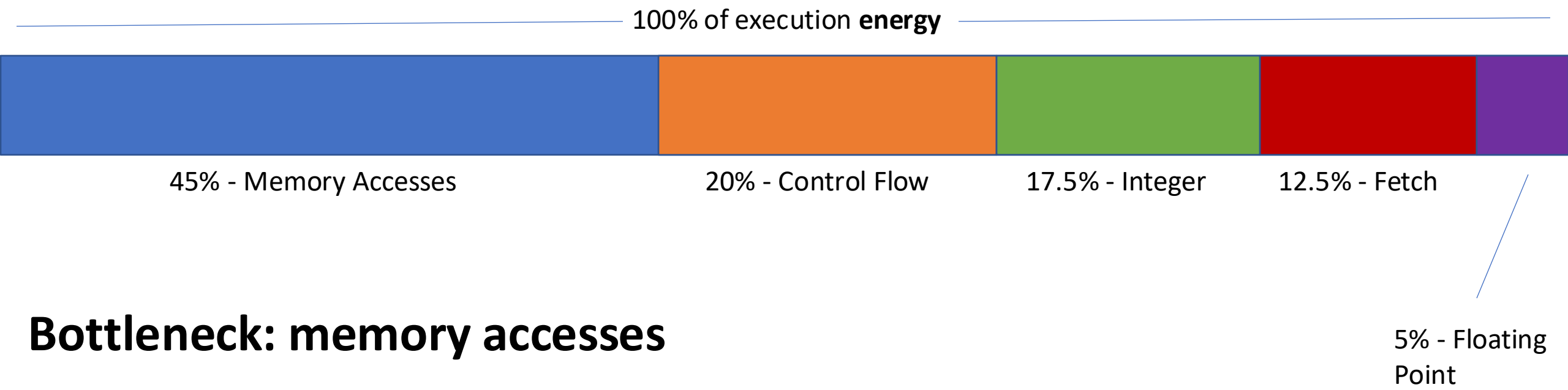
max speedup ***5x!***

Amdahl's Law is Extremely Versatile



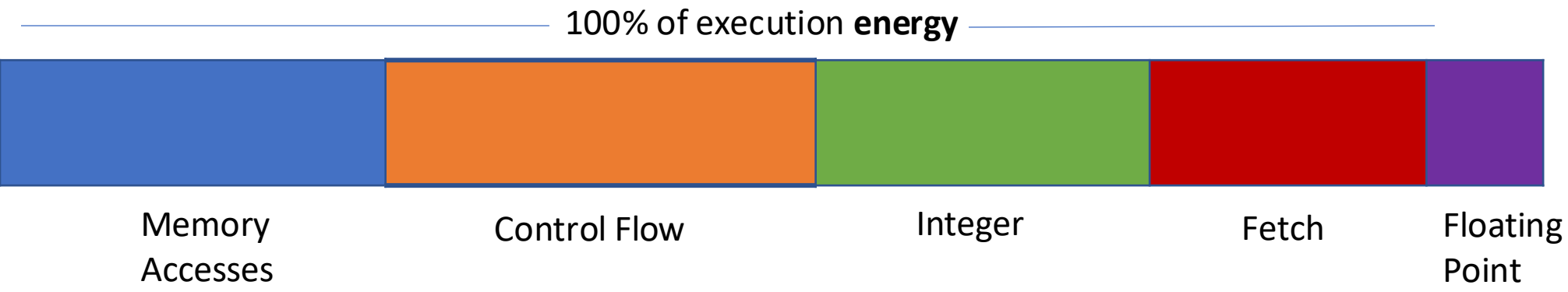
Works for any optimization problem and goal. Always focus on the biggest slice & the rest doesn't matter.

Using Amdahl's Law to push the bottleneck around



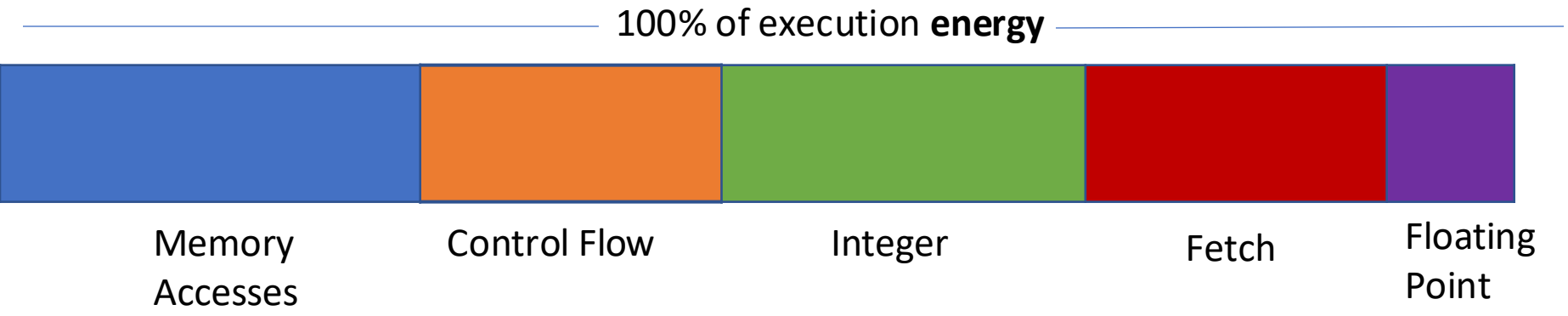
Bottleneck: memory accesses

Using Amdahl's Law to push the bottleneck around



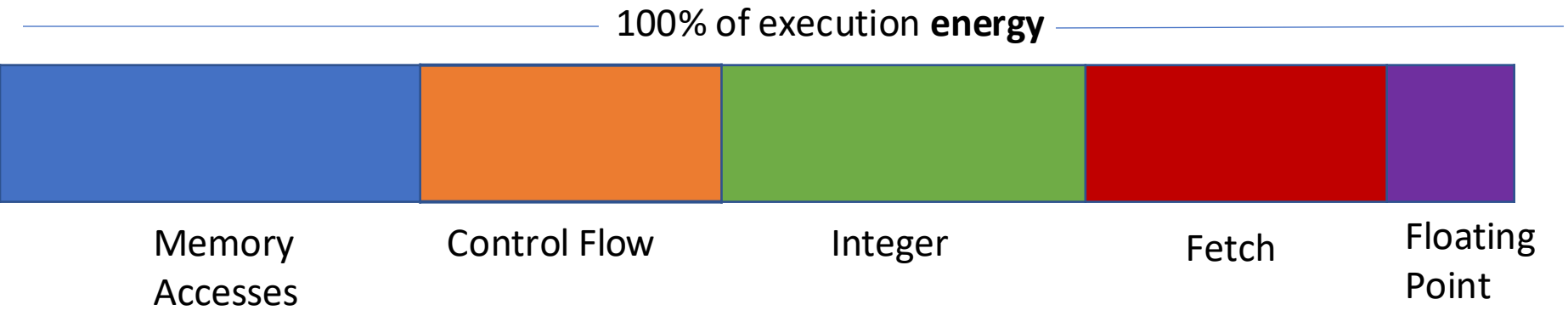
New bottleneck: control flow

Using Amdahl's Law to push the bottleneck around



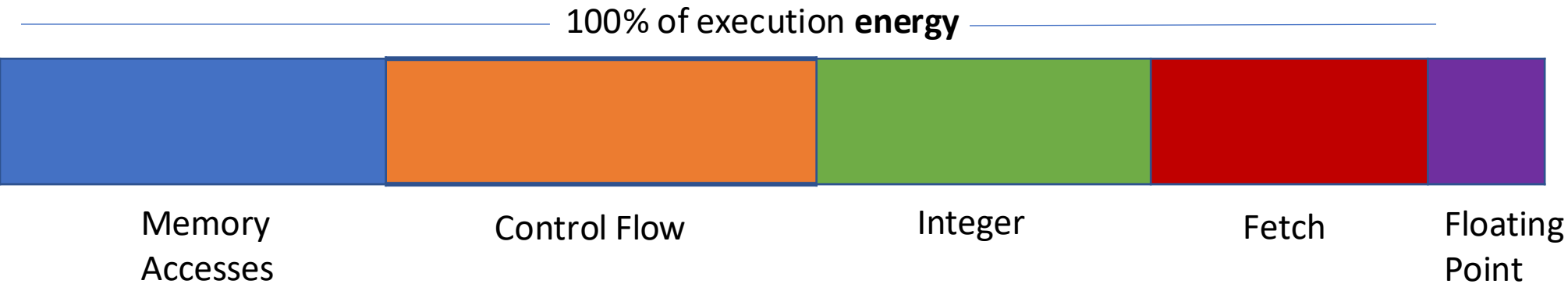
New bottleneck: memory accesses (again!)

Using Amdahl's Law to push the bottleneck around



Remember: Amdahl tells us to optimize the biggest slice

Another view of the world: Gustaffson's Law



Idea: find an *optimizable* part of your system and make it *bigger*

If we know that memory is optimizable, why not optimize more and do more memory accesses?

Another view of the world: Gustaffson's Law

Gustafson's Law: Sequential part does not grow as optimizable part grows. Can always add more optimizable part and make sequential part matter less

Assume that we can scale up # of parallel memory accesses, N

Assume we can scale input up to use all N parallel accesses

```
data_size = 10
data[data_size] = {…}
if(…){ }
…//18 more of these conditionals
if(…){ }

for d in 0..data_size{ d++ }
```


```
data_size = 100000
data[data_size] = {…}
if(…){ }
…//18 more of these conditionals
if(…){ }
#parallel[N=1000]
for d in 0..data_size{ d++ }
```

Another view of the world: Gustaffson's Law

85% - Memory Accesses

Gustafson's Law for overall speedup with speedup factor of N:

(assume) Optimized time = T = 1

Unoptimized time = T' = (1-p)T + pT*N = (1-p) + pN

Scaled Speedup = T' / T = (1-p) + pN

Another view of the world: Gustaffson's Law

Scale parallel memory accesses, N, up to 1000?

***Scaled Speedup* = 1-p + 1000p = 999p + 1**

***Scaled Speedup* = 999 * 0.85 + 1 = 850x**

85% - Memory Accesses

Gustafson's Law for overall speedup with speedup factor of N:

(assume) Optimized time = T = 1

Unoptimized time = T' = (1-p)T + pT*N = (1-p) + pN

Scaled Speedup = T' / T = (1-p) + pN

What did we just learn?

- Two high-level architectural models
- Identify performance bottlenecks
- Develop optimizations to mitigate bottlenecks
- Analyze resulting improvement from mitigating bottlenecks
- Identifying persistent performance limiters (e.g., branches)
- Optimize in software ***or*** hardware
- (Almost) never bet against Gene Amdahl in an optimization contest!

What to think about next?

- What is a computer architecture?
- What matters when defining a HW/SW interface?
- What is above the ISA and what is below the ISA?
- What is hidden from the programmer and what is exposed?