

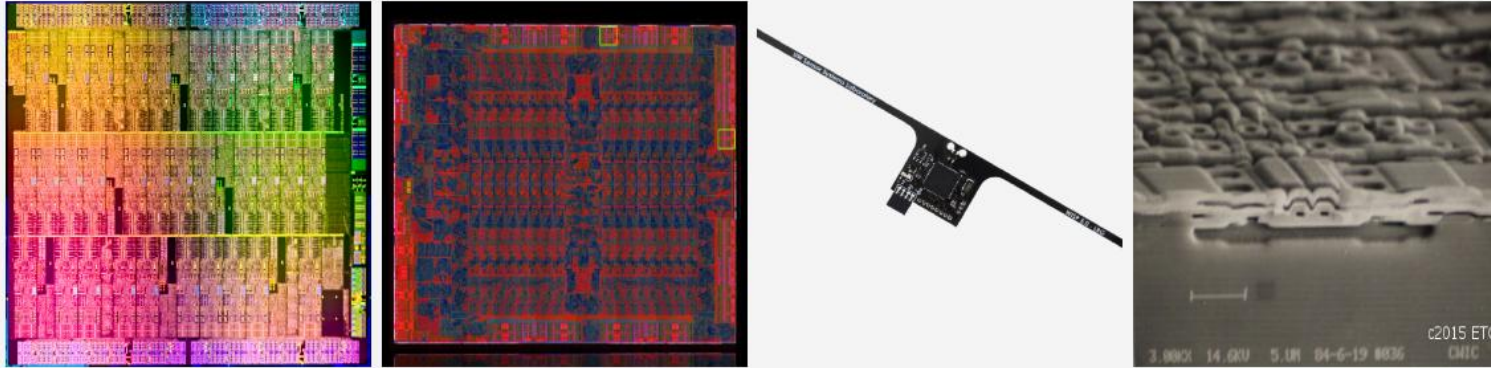


18-344: Computer Systems and the Hardware-Software Interface

[Home](#)
[Syllabus](#)
[Course Schedule](#)
[Lab Details](#)
[Homework Details](#)
[Recitation Slides](#)
[Exam Details](#)
[Piazza](#)

[Staff](#)

18-344: Computer Systems and the Hardware-Software Interface (Fall 2026 Lecture 2)



Course Description

LECTURE 2

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.

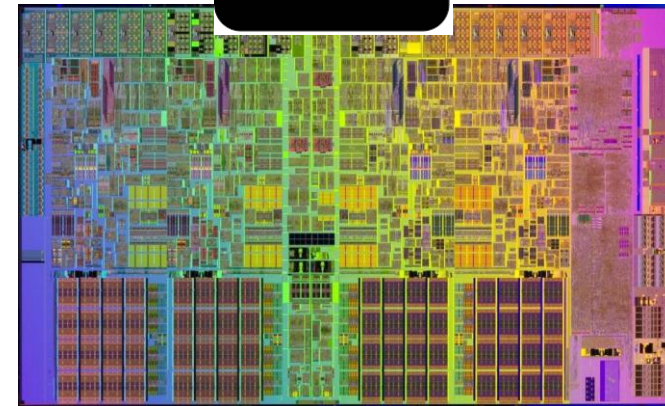
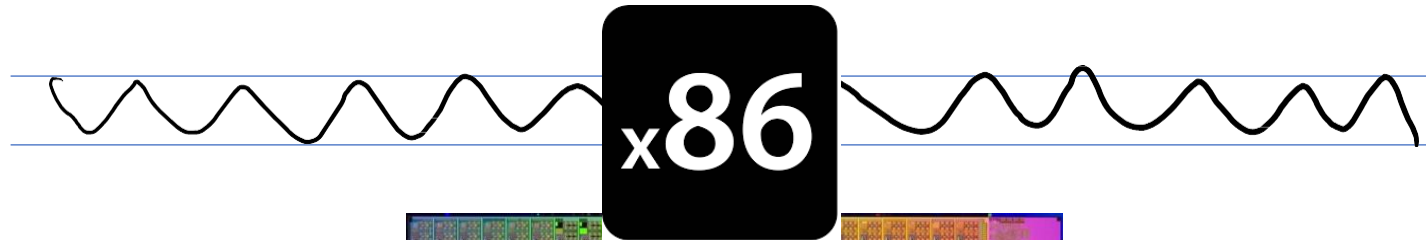
What is the hardware/software boundary?

- 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

- 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

```
379 int walk_page_range(struct mm_struct *mm, unsigned long start,
380                    unsigned long end, const struct mm_walk_ops *ops,
381                    void *private)
382 {
383     int err = 0;
384     unsigned long next;
385     struct vm_area_struct *vma;
386     struct mm_walk walk = {
387         .ops = ops,
388         .mm = mm,
389         .private = private,
390     };
391
392     if (start >= end)
393         return -EINVAL;
394
395     if (!walk.mm)
396         return -EINVAL;
397
398     mmap_assert_locked(walk.mm);
399
400     vma = find_vma(walk.mm, start);
401     do {
402         if (!vma) { /* after the last vma */
403             walk.vma = NULL;
404             next = end;
405         }
```



The 240 view of the world

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

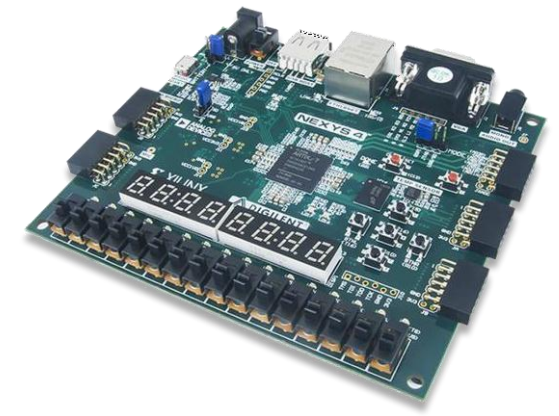
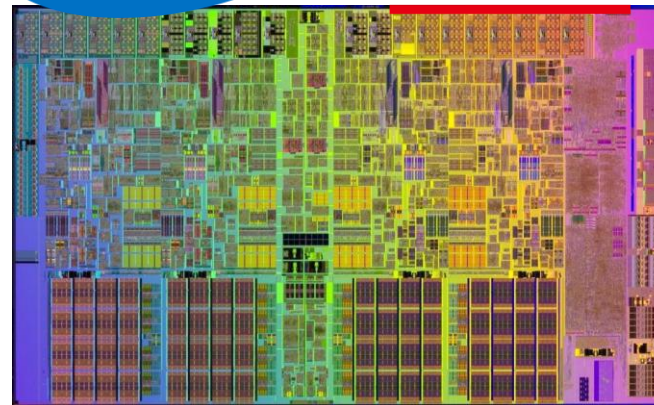
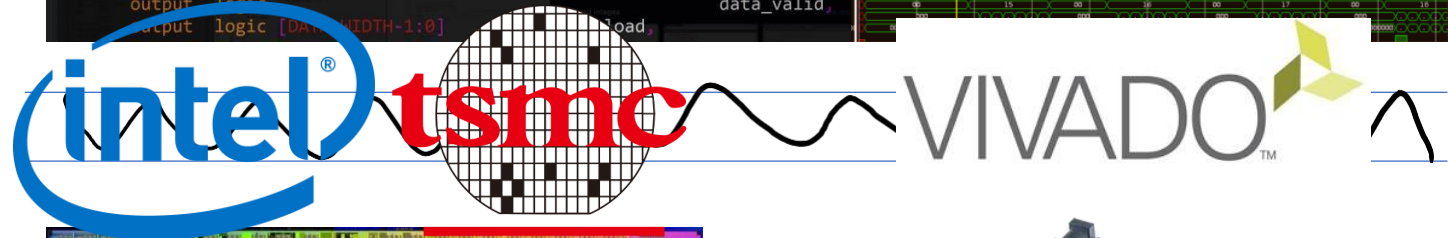
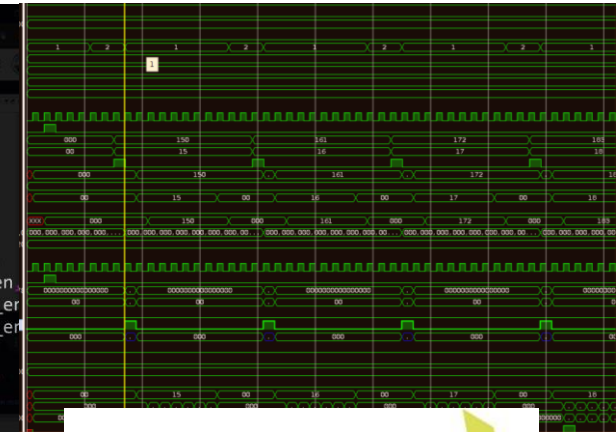
```
include "io.vh"
include "helpers.vh"

module cway

  #(parameter DATA_WIDTH=32,
    parameter ADDR_WIDTH=30,
    parameter SET_COUNT=4,
    parameter BLOCK_SIZE=4,
    parameter ID=4)

  (input logic
    input logic
    input logic
    input logic
    input logic [ADDR_WIDTH-1:0]
    output logic
    output logic [DATA_WIDTH-1:0]

    clk, rst_l,
    data_load_en,
    data_store_en,
    data_flush_en,
    data_addr,
    data_ready,
    data_valid,
    load
```



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

```
179 int walk_page_range(struct mm_struct *mm, unsigned long start,
180                    unsigned long end, const struct mm_walk_ops *ops,
181                    void *private)
182 {
183     int err = 0;
184     unsigned long next;
185     struct vm_area_struct *vma;
186     struct mm_walk_walk = {
187         .ops      = ops,
188         .mm       = mm,
189         .private   = private,
190     };
191
192     if (start >= end)
193         return -EINVAL;
194
195     if (!walk.mm)
196         return -EINVAL;
197
198     mmap_assert_locked(walk.mm);
199
200     vma = find_vma(walk.mm, start);
201     do {
202         if (!vma) { /* after the last vma */
203             walk.vma = NULL;
204             next = end;
205         }
206     } while (1);
207 }
```



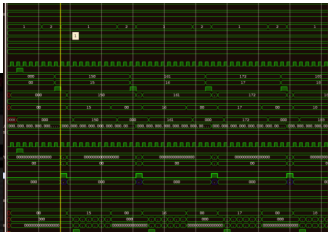
```
include "io.vh"
include "helpers.vh"

module cway

#(parameter DATA_WIDTH=32,
parameter ADDR_WIDTH=30,
parameter SET_COUNT=4,
parameter BLOCK_SIZE=4,
parameter ID=4)

(input logic
input logic
input logic
input logic
input logic [ADDR_WIDTH-1:0]
output logic
output logic
output logic [DATA_WIDTH-1:0]

clk, rst_l,
data_load_en,
data_store_en,
data_flush_en,
data_addr,
data_ready,
data_valid,
data_load,
```



Relative Observability?

```
179 int walk_page_range(struct mm_struct *mm, unsigned long start,
180                    unsigned long end, const struct mm_walk_ops *ops,
181                    void *private)
182 {
183     int err = 0;
184     unsigned long next;
185     struct vm_area_struct *vma;
186     struct mm_walk_walk = {
187         .ops = ops,
188         .mm = mm,
189         .private = private,
190     };
191
192     if (start >= end)
193         return -EINVAL;
194
195     if (!walk_mm)
196         return -EINVAL;
197
198     mmap_assert_locked(walk_mm);
199
200     vma = find_vma(walk_mm, start);
201     do {
202         if (!vma) { /* after the last vma */
203             walk.vma = NULL;
204             next = end;
205         }
206     } while (1);
207 }
```



```
include "io.vh"
include "helpers.vh"

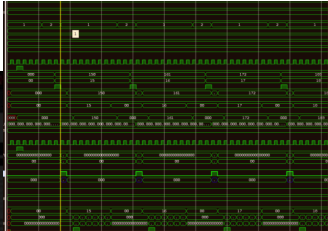
module cway

    #(parameter DATA_WIDTH=32,
      parameter ADDR_WIDTH=30,
      parameter SET_COUNT=4,
      parameter BLOCK_SIZE=4,
      parameter ID=4)

    (input logic
     input logic
     input logic
     input logic
     input logic [ADDR_WIDTH-1:0]
     output logic
     output logic
     output logic [DATA_WIDTH-1:0]

     clk, rst_l,
     data_load_en,
     data_store_en,
     data_flush_en,

     data_addr,
     data_ready,
     data_valid,
     data_load,
```



Relative Optimizability?

```
179 int walk_page_range(struct mm_struct *mm, unsigned long start,
180                    unsigned long end, const struct mm_walk_ops *ops,
181                    void *private)
182 {
183     int err = 0;
184     unsigned long next;
185     struct vm_area_struct *vma;
186     struct mm_walk_walk = {
187         .ops      = ops,
188         .mm       = mm,
189         .private  = private,
190     };
191
192     if (start >= end)
193         return -EINVAL;
194
195     if (!walk_mm)
196         return -EINVAL;
197
198     mmap_assert_locked(walk_mm);
199
200     vma = find_vma(walk_mm, start);
201     do {
202         if (!vma) { /* after the last vma */
203             walk.vma = NULL;
204             next = end;
205         }
206     } while (1);
207 }
```



```
include "io.vh"
include "helpers.vh"

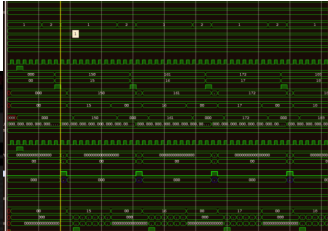
module cway

    #(parameter DATA_WIDTH=32,
      parameter ADDR_WIDTH=30,
      parameter SET_COUNT=4,
      parameter BLOCK_SIZE=4,
      parameter ID=4)

    (input logic
     input logic
     input logic
     input logic
     input logic [ADDR_WIDTH-1:0]
     output logic
     output logic
     output logic [DATA_WIDTH-1:0]

     clk, rst_l,
     data_load_en,
     data_store_en,
     data_flush_en,

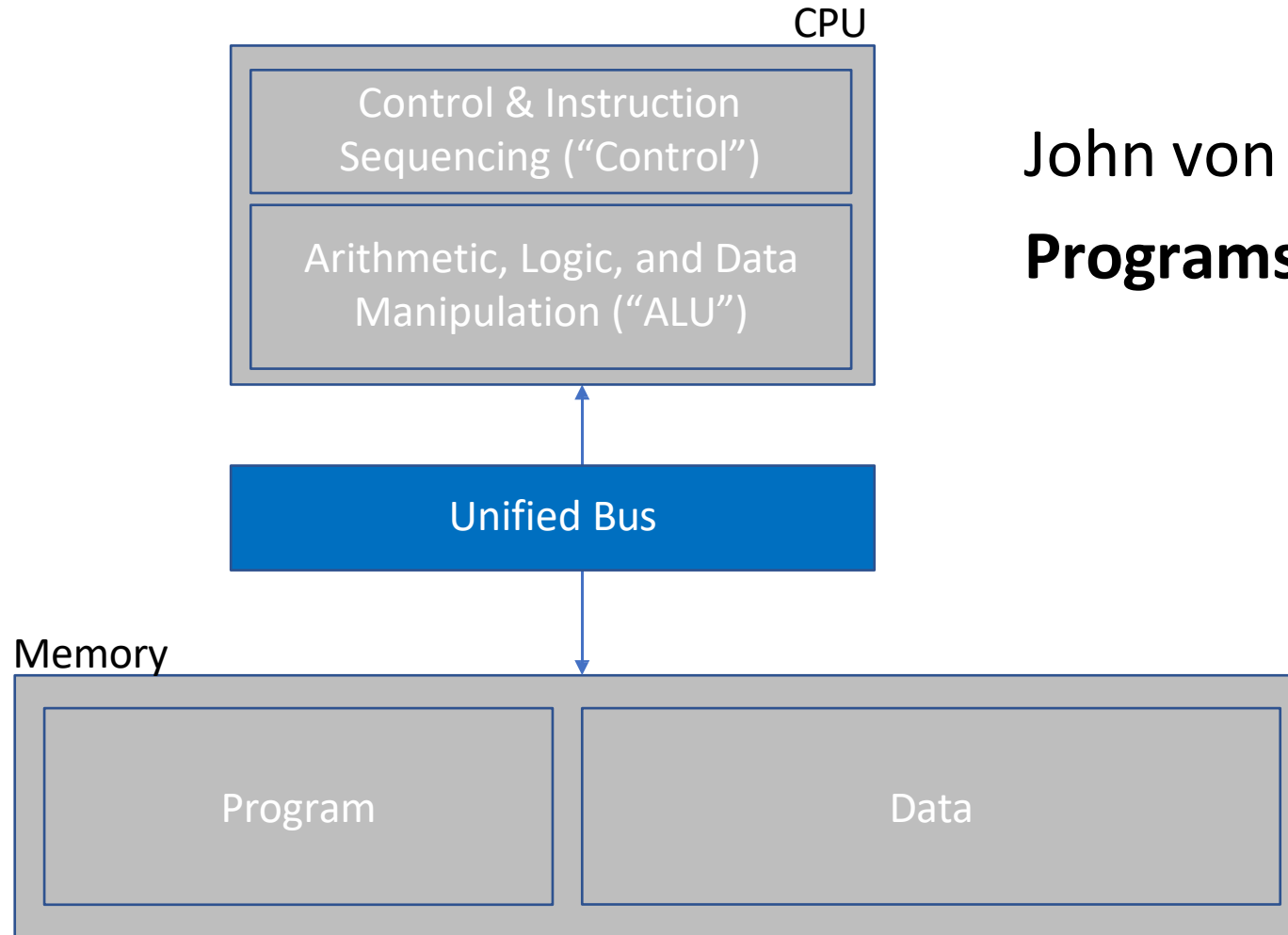
     data_addr,
     data_ready,
     data_valid,
     data_load,
```



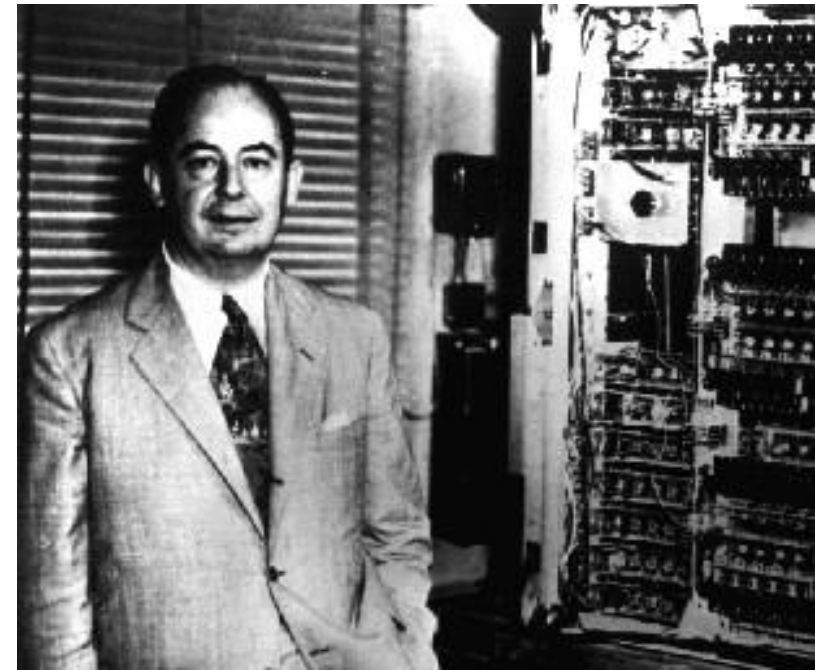
Today: Thinking Like a Computer Architect



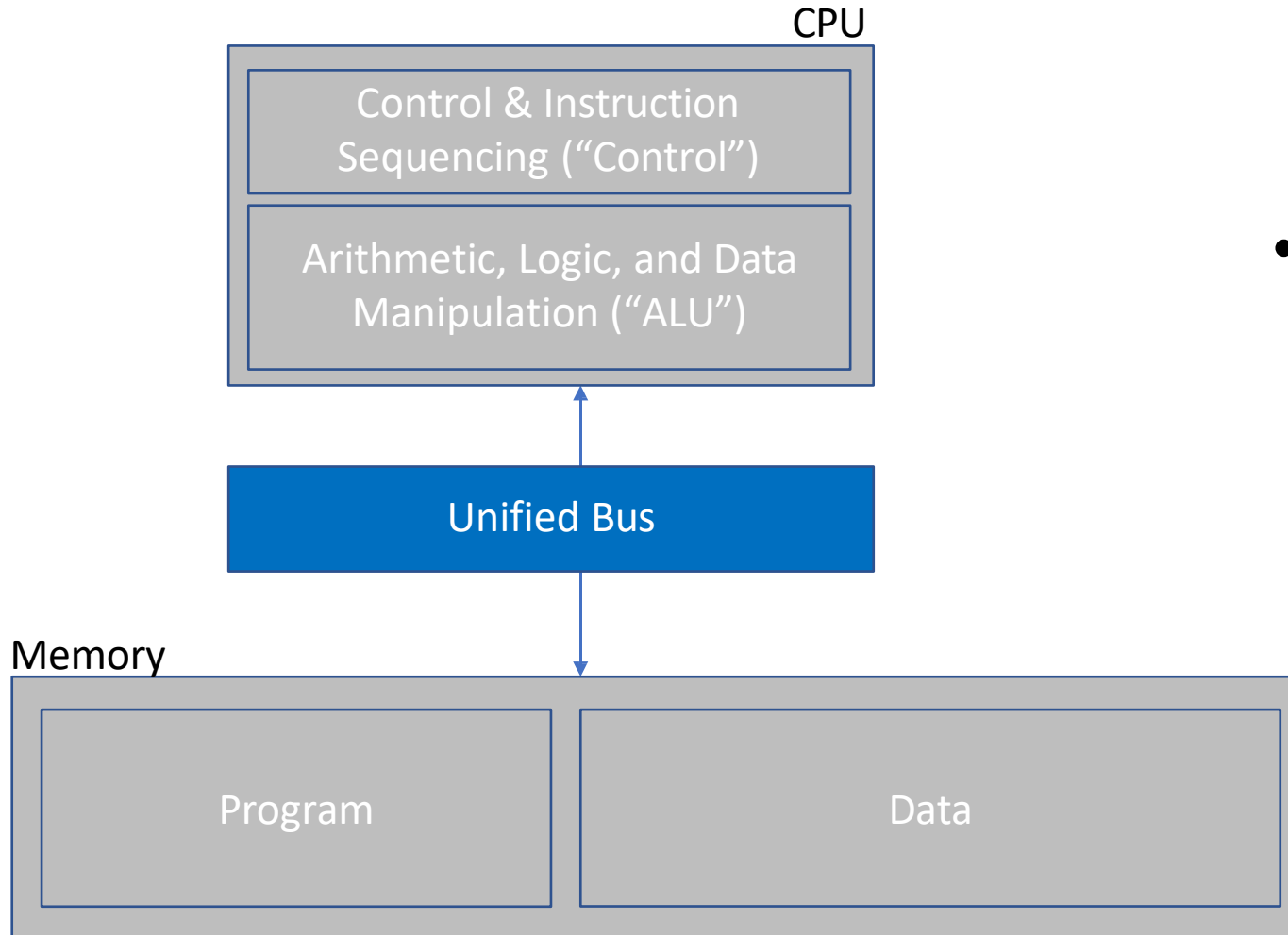
Our first hw/sw interface: The Von Neumann Computing Model



John von Neumann's Big Idea:
Programs are data.

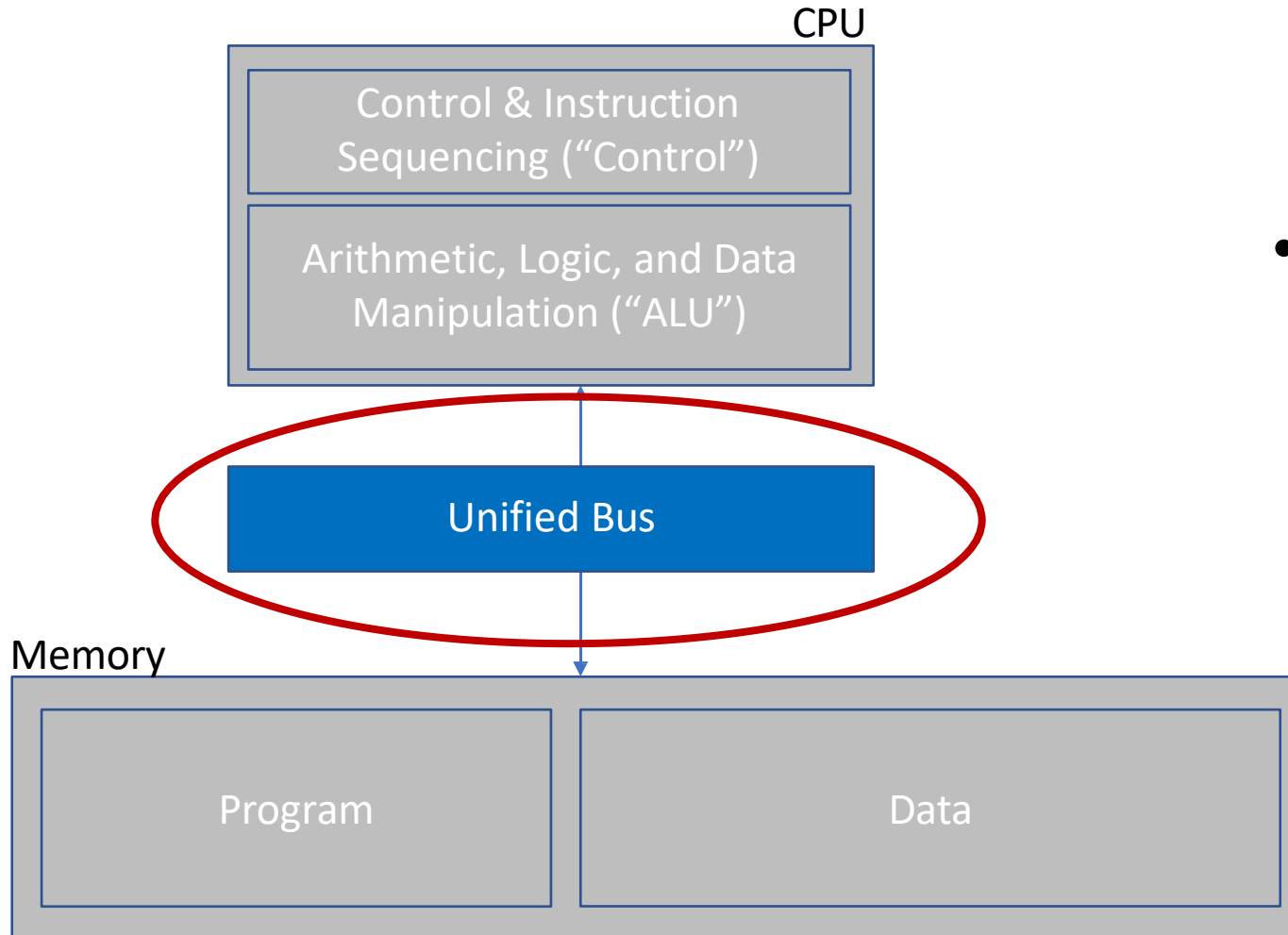


The Von Neumann / Stored-Program Computing Model



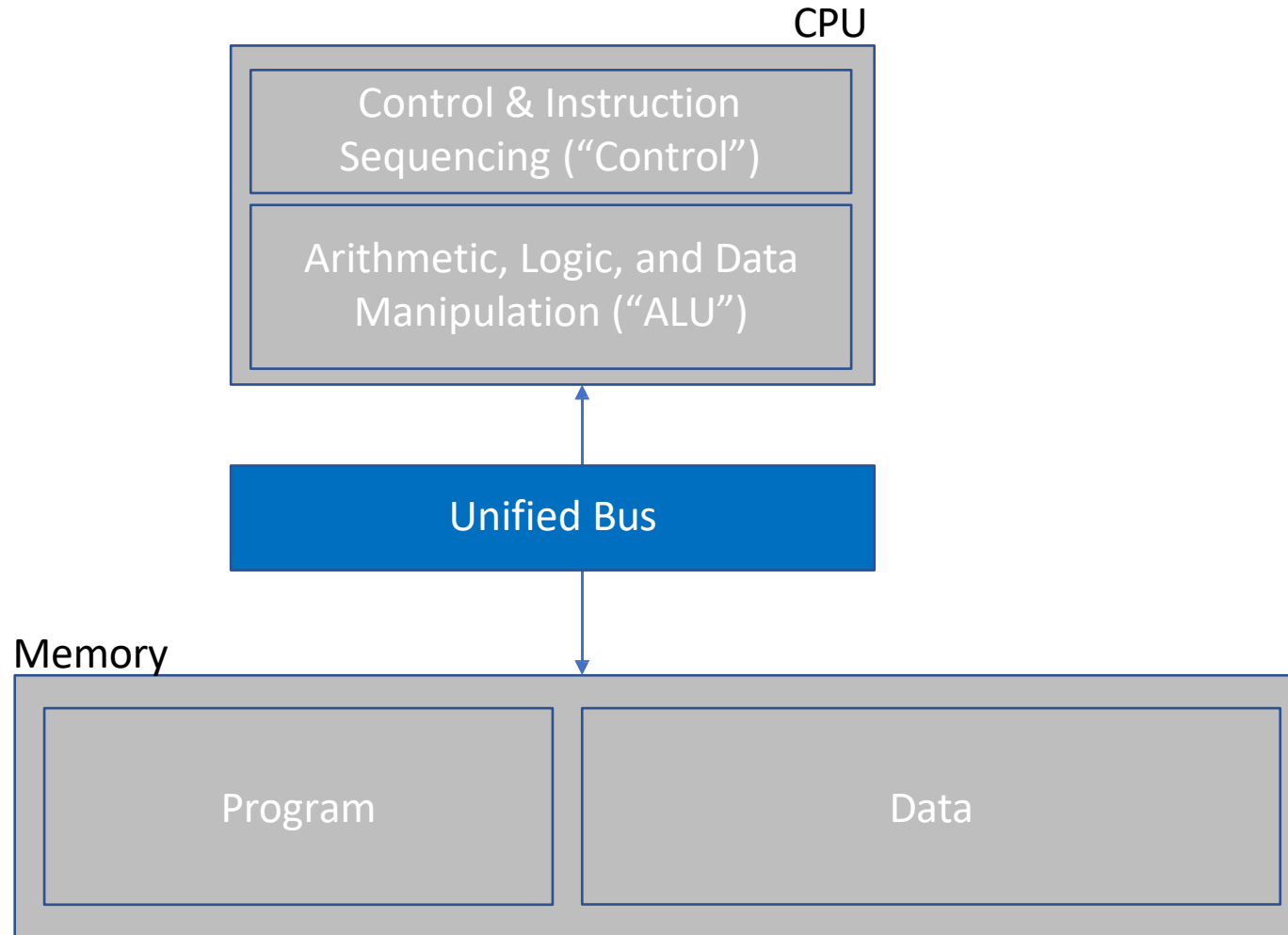
- Let's optimize! Where is there a bottleneck in the Von Neumann abstract machine?

The Von Neumann / Stored-Program Computing Model



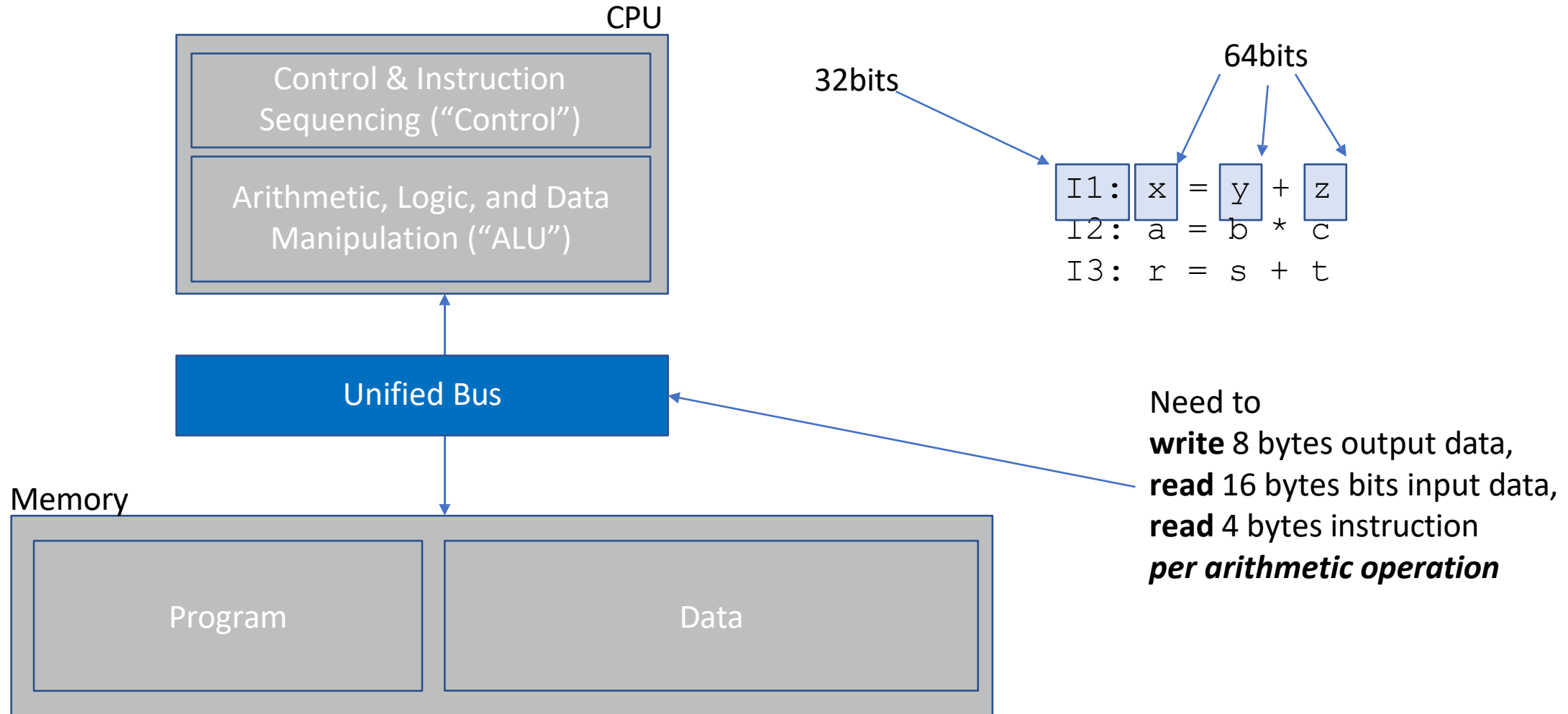
- Data & Program share a bus into the CPU. Need to time multiplex access to the bus.

The Von Neumann / Stored-Program Computing Model

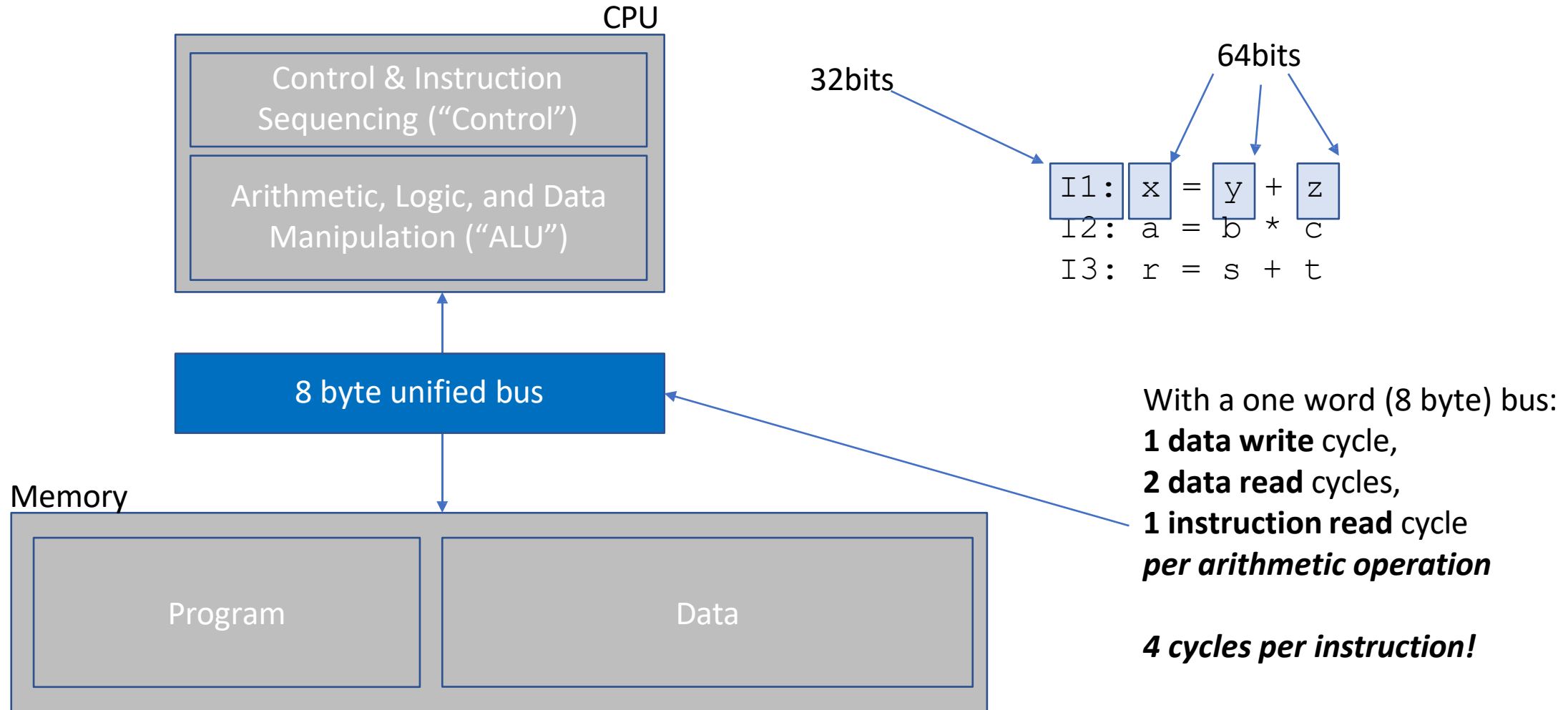


I1: $x = y + z$
I2: $a = b * c$
I3: $r = s + t$

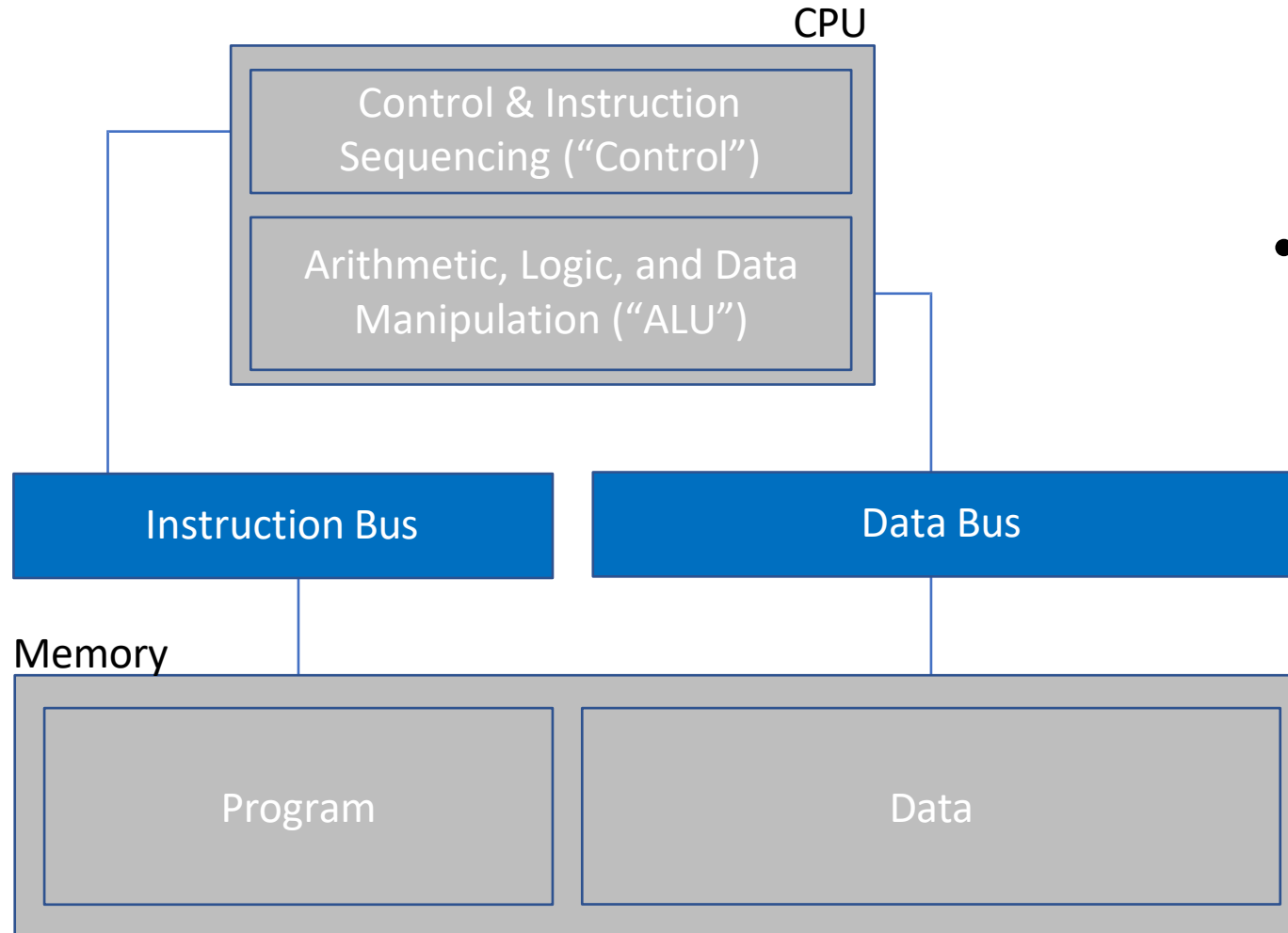
The Von Neumann / Stored-Program Computing Model



The Von Neumann / Stored-Program Computing Model

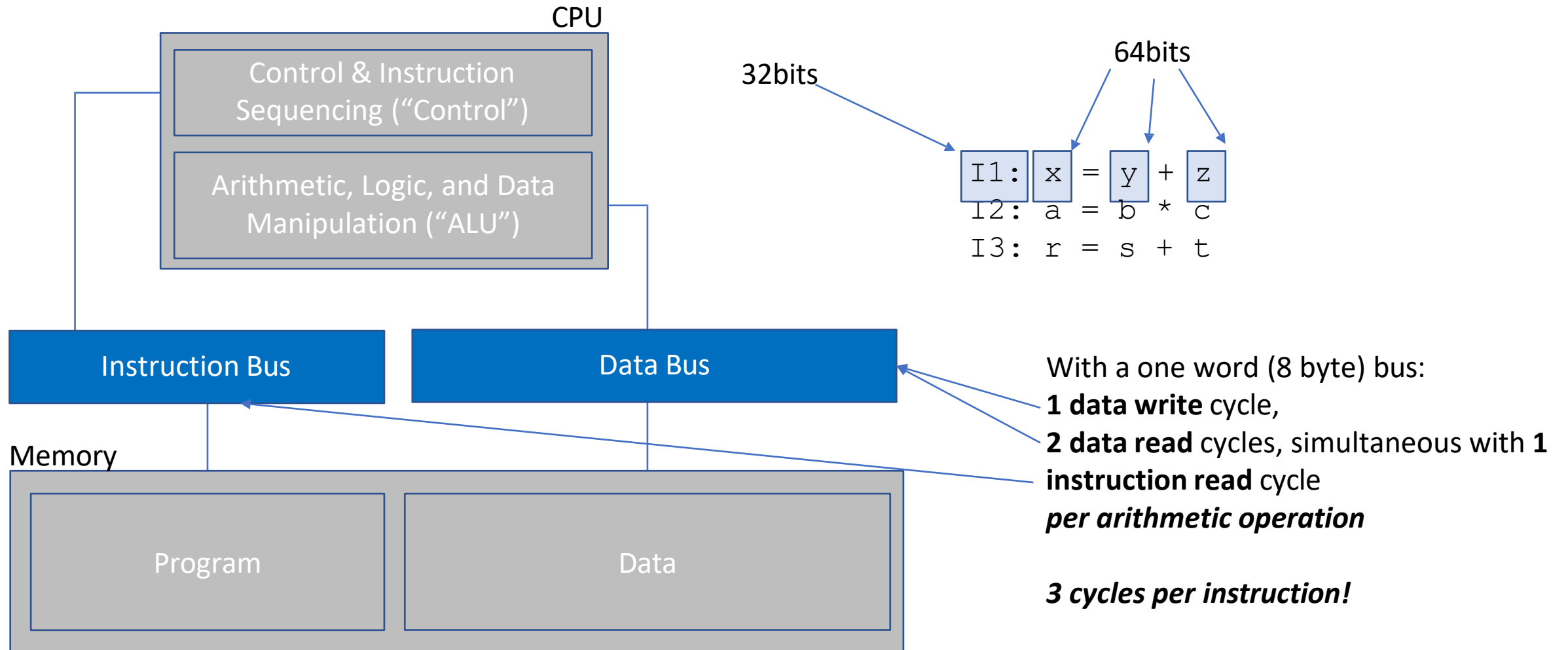


Alternative to von Neumann: the Harvard Architecture

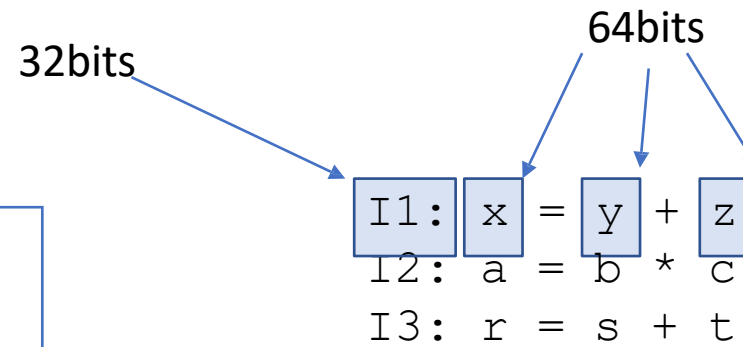
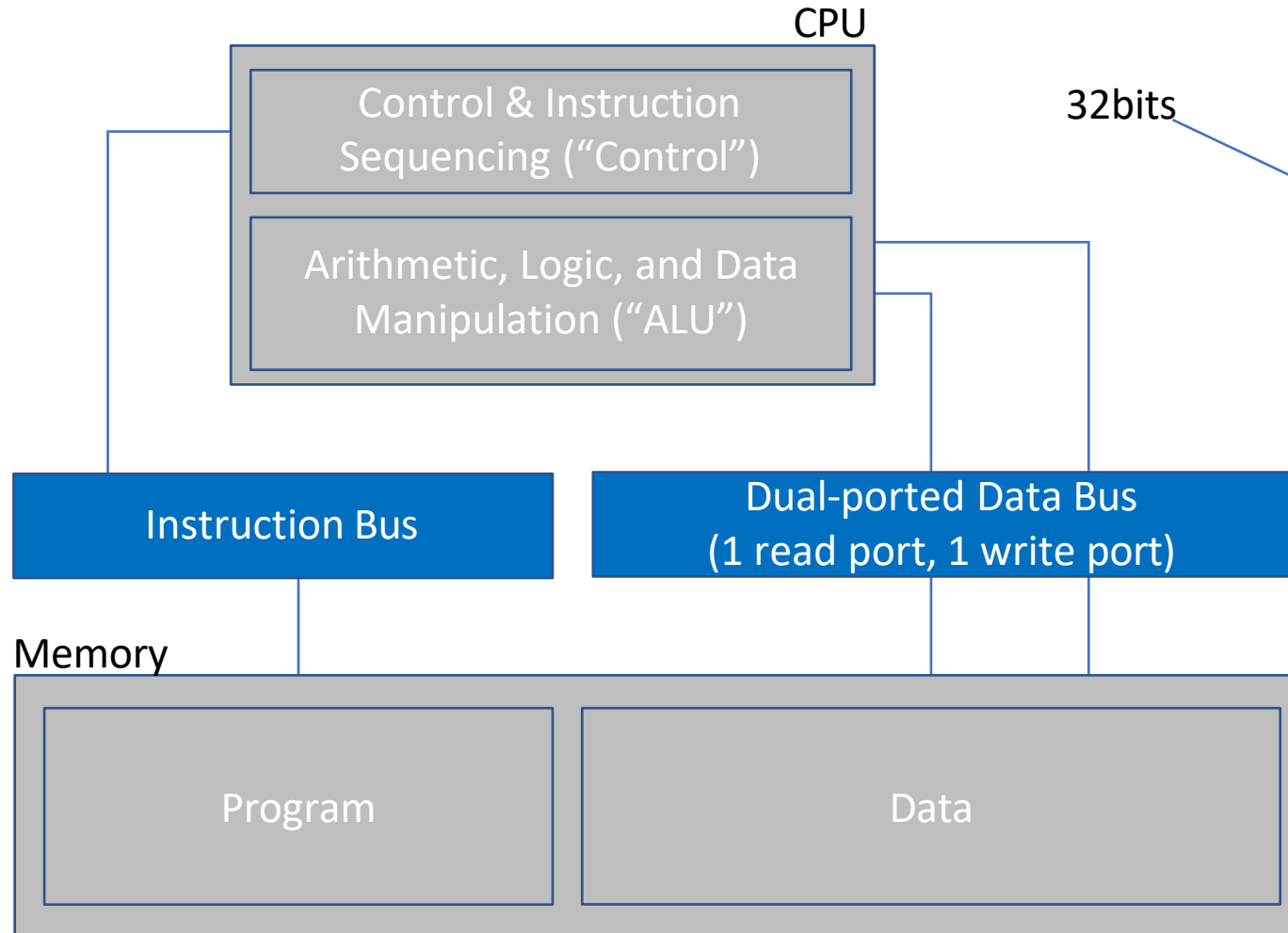


- Split bus architecture provides *simultaneous* access to program and to data

Alternative to von Neumann: the Harvard Architecture



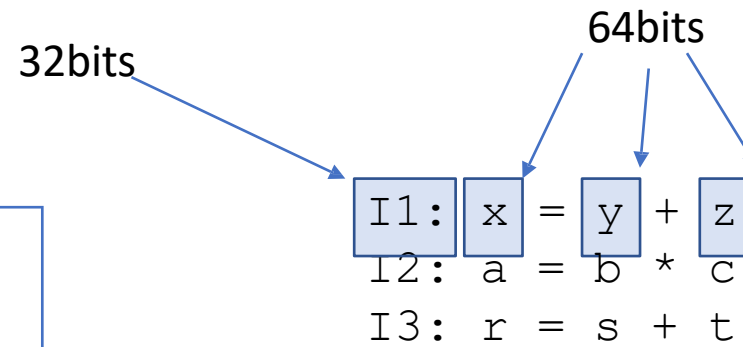
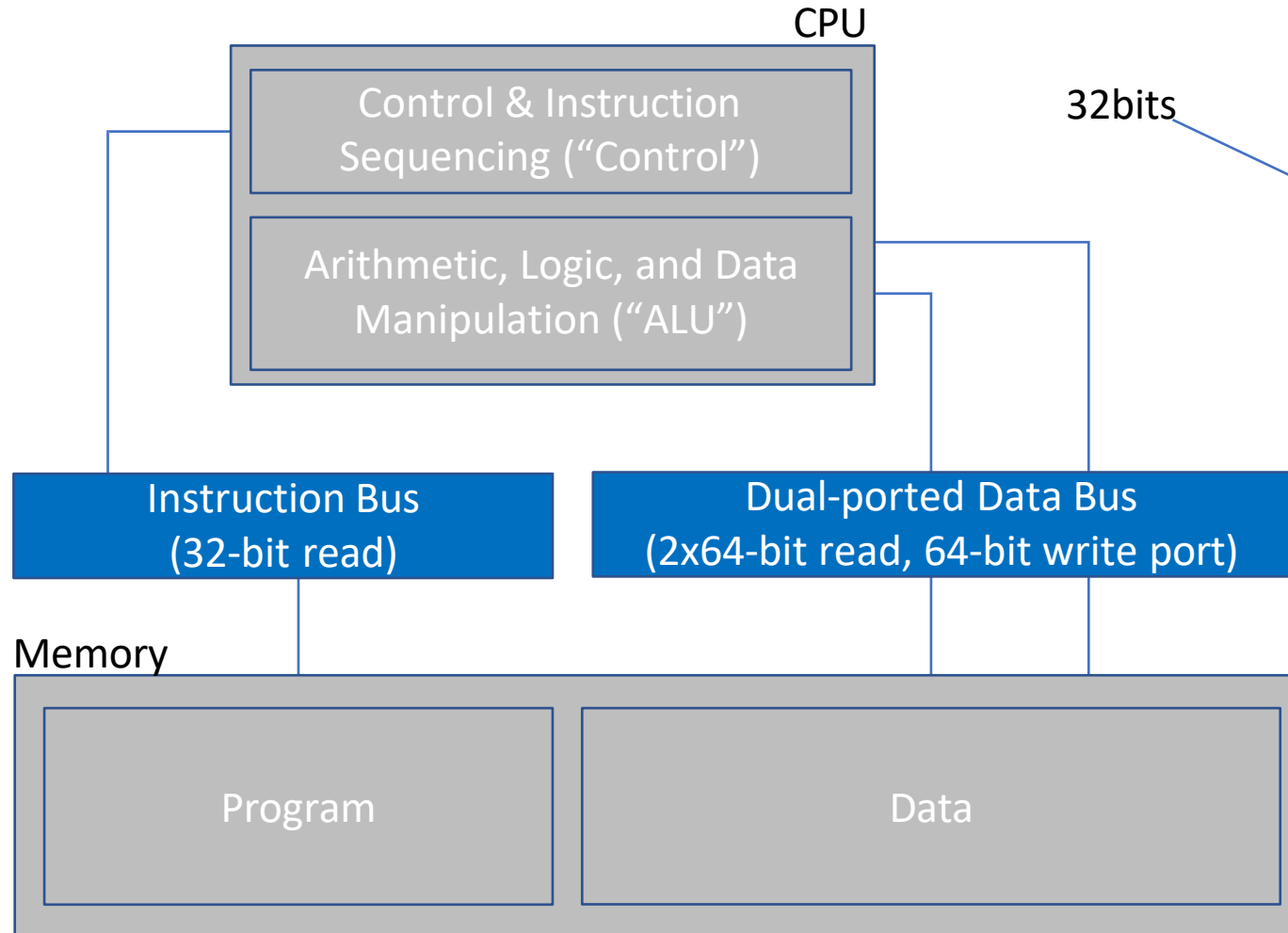
Optimizing our Harvard Architecture



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!

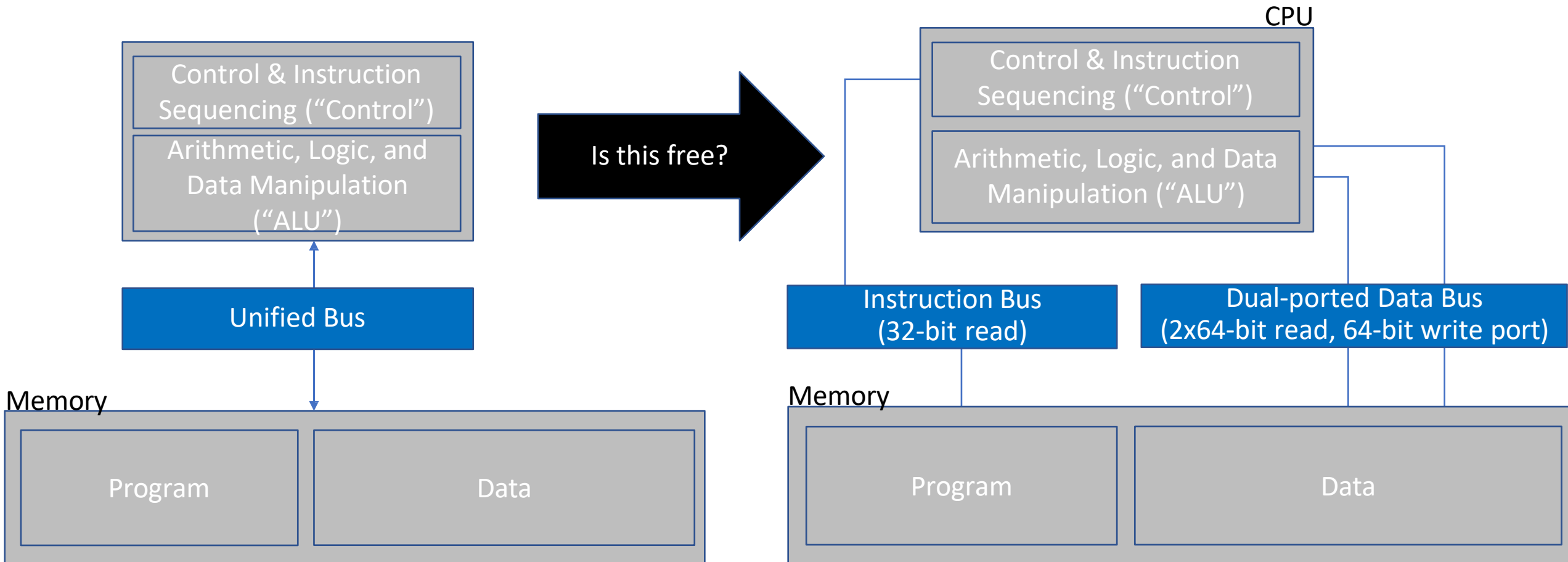
Optimizing our Harvard Architecture



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

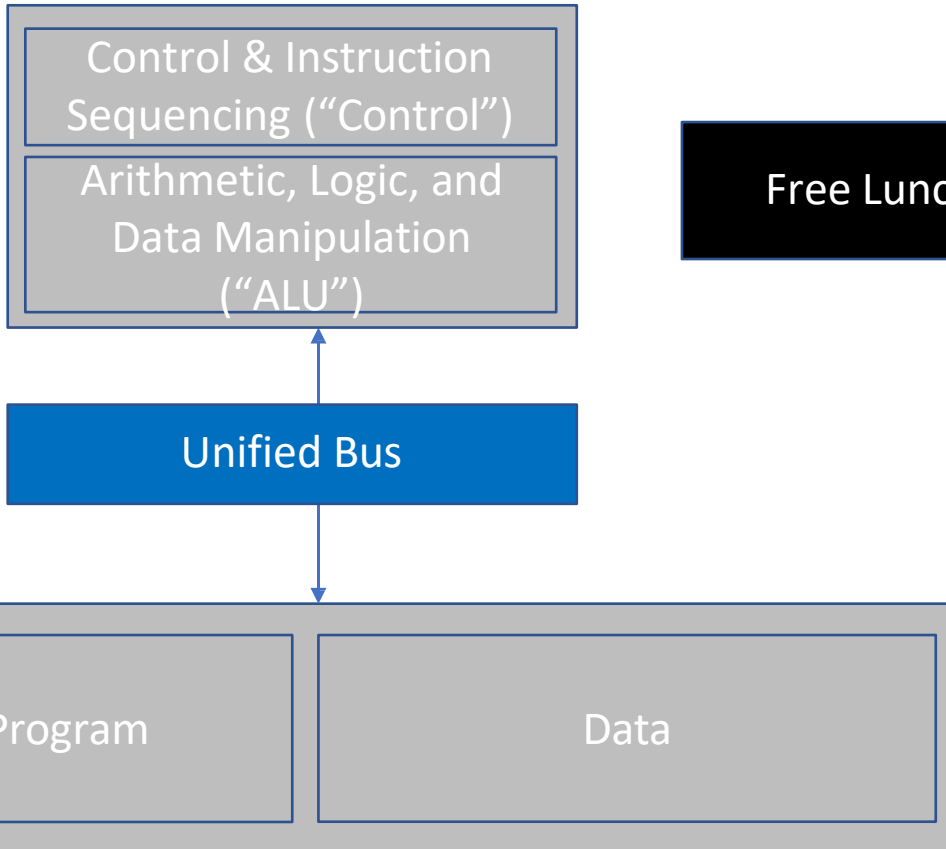


```
$ ./destiny config/SRAM_128_1_32.cfg
```

4-byte Memory Interface:

Read Latency = 0.289ns

Write Latency = 0.212ns



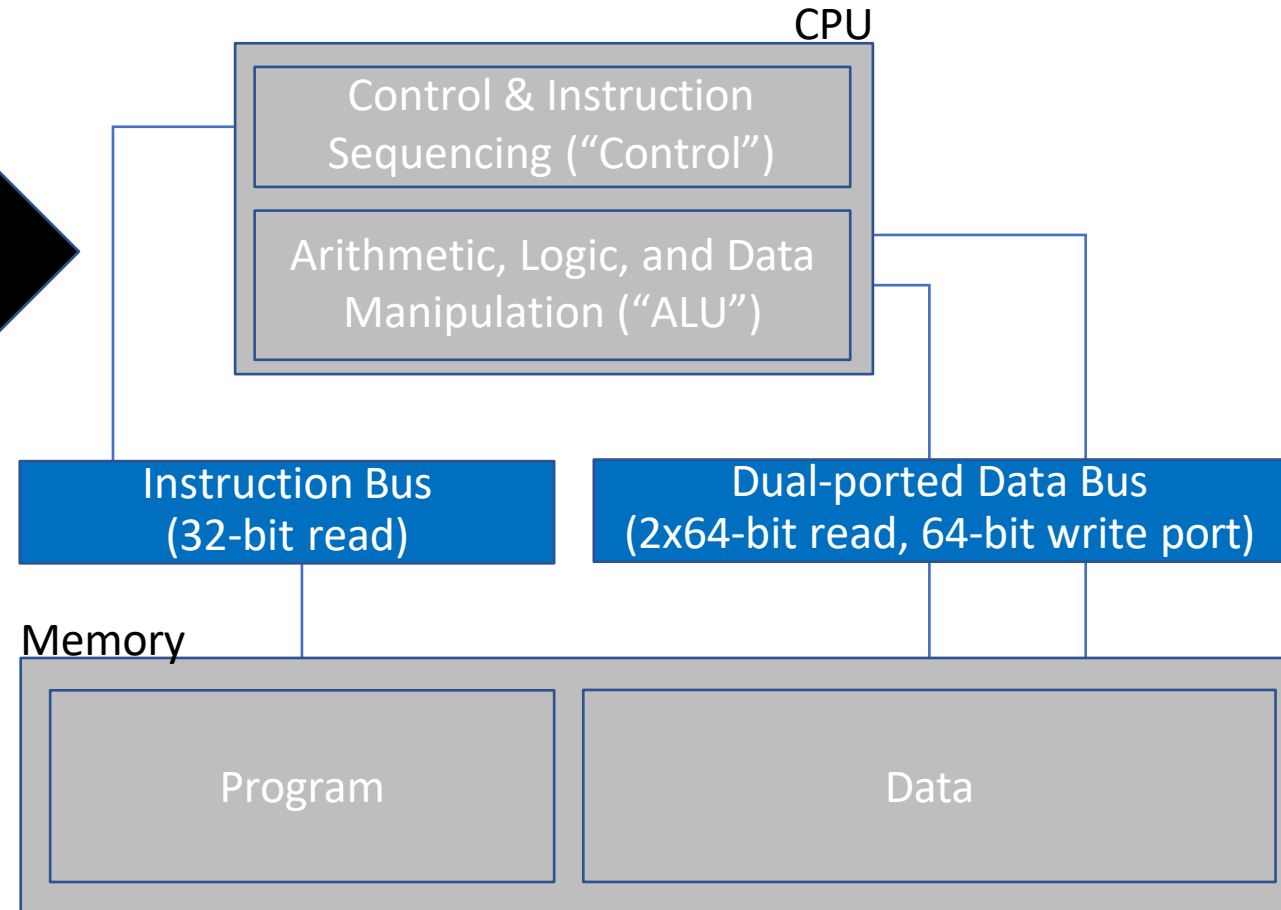
Free Lunch!

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

16-byte Memory Interface:

Read Latency = 0.289ns

Write Latency = 0.212ns

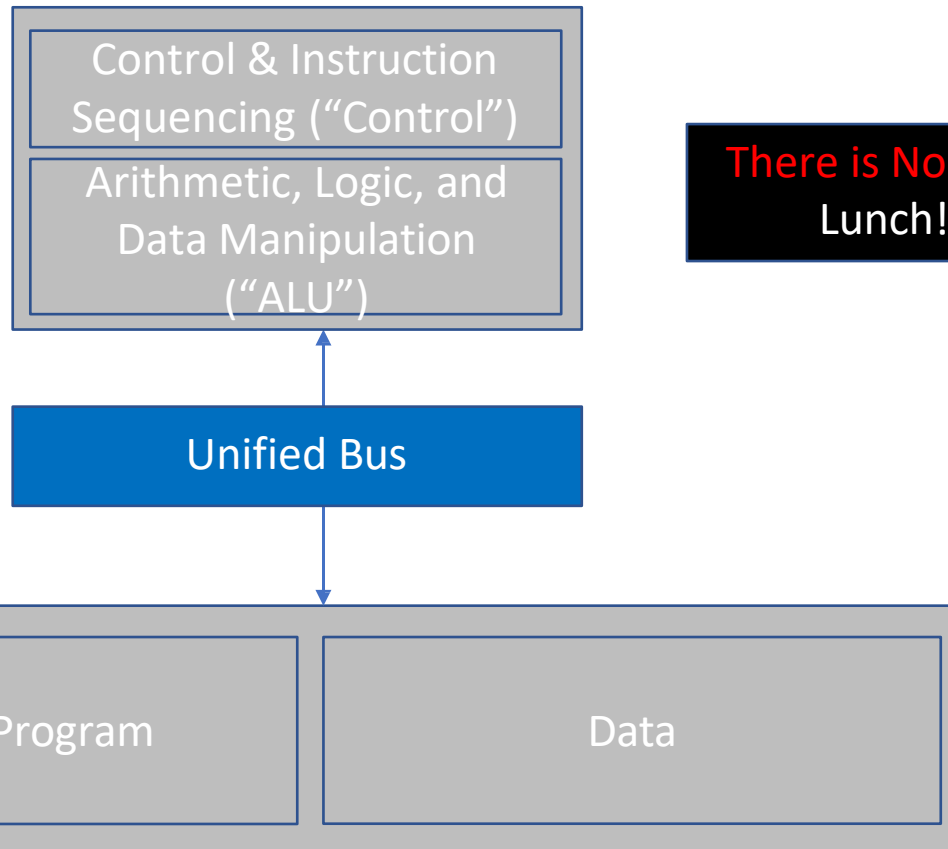


```
$ ./destiny config/SRAM_128_1_32.cfg
```

4-byte Memory Interface:

Read **Energy** = **0.836pJ**

Write **Energy** = **0.738pJ**



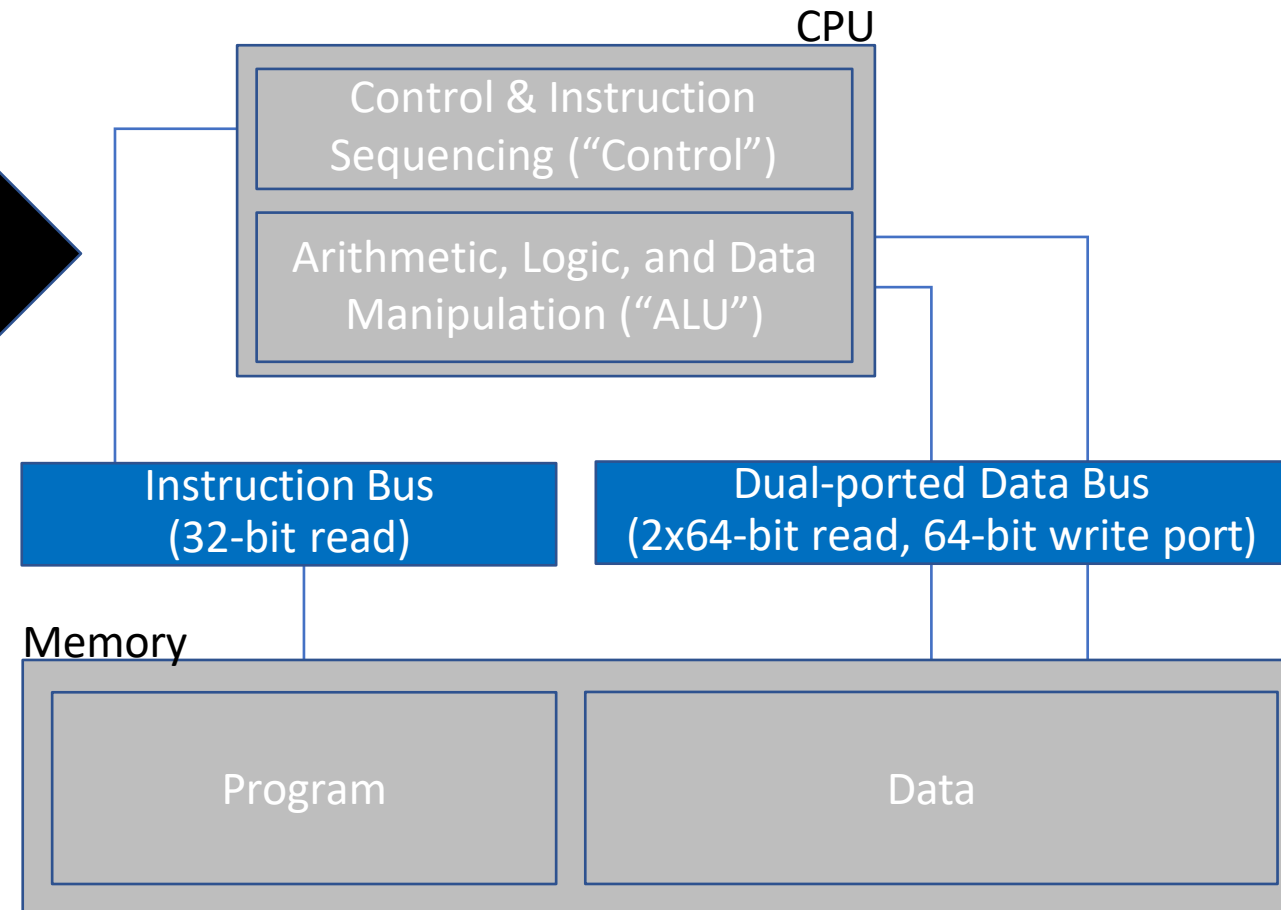
There is No Free Lunch!

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

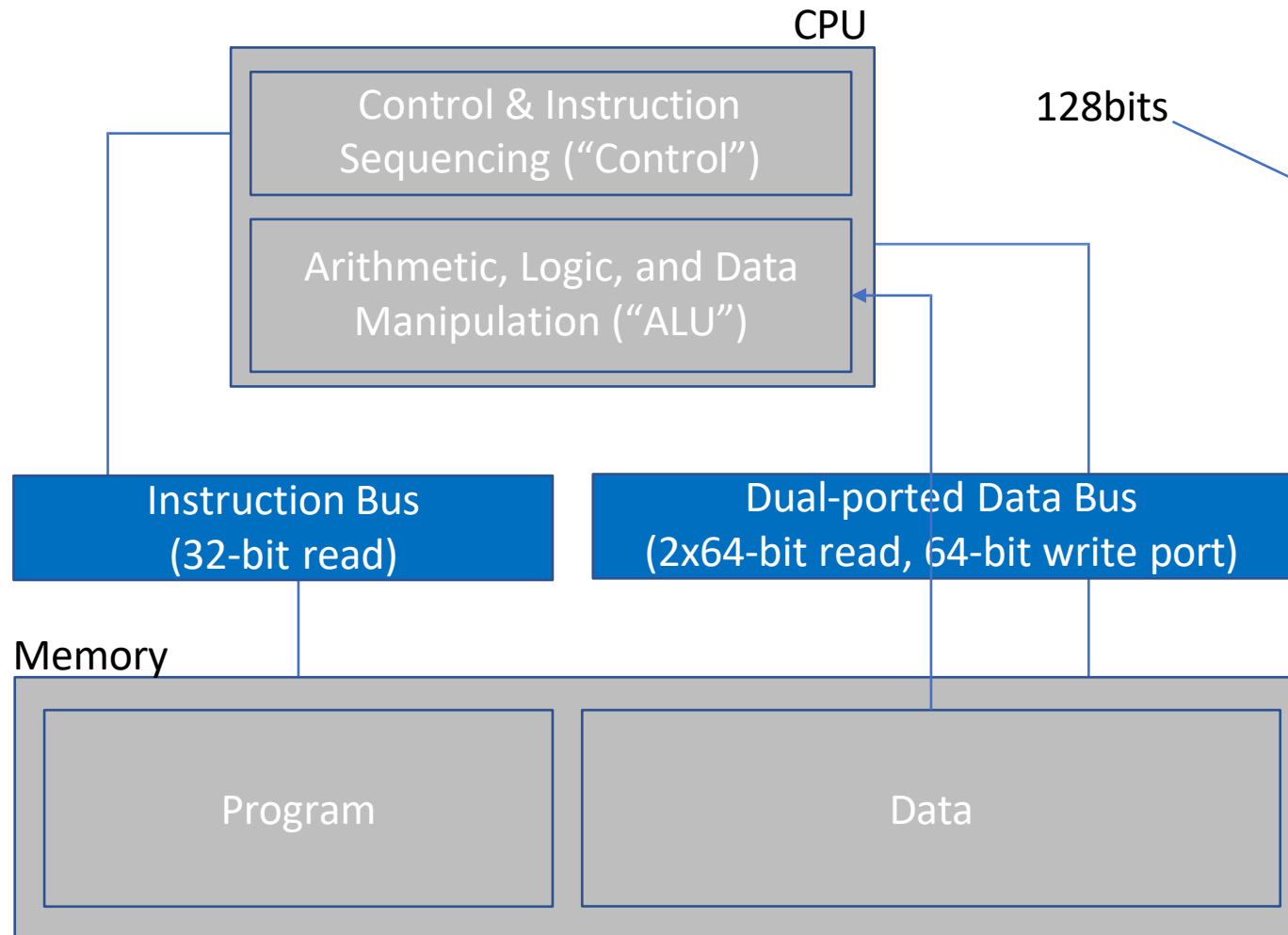
16-byte Memory Interface:

Read **Energy** = **1.51pJ**

Write **Energy** = **1.30pJ**



How about optimizing instruction supply?

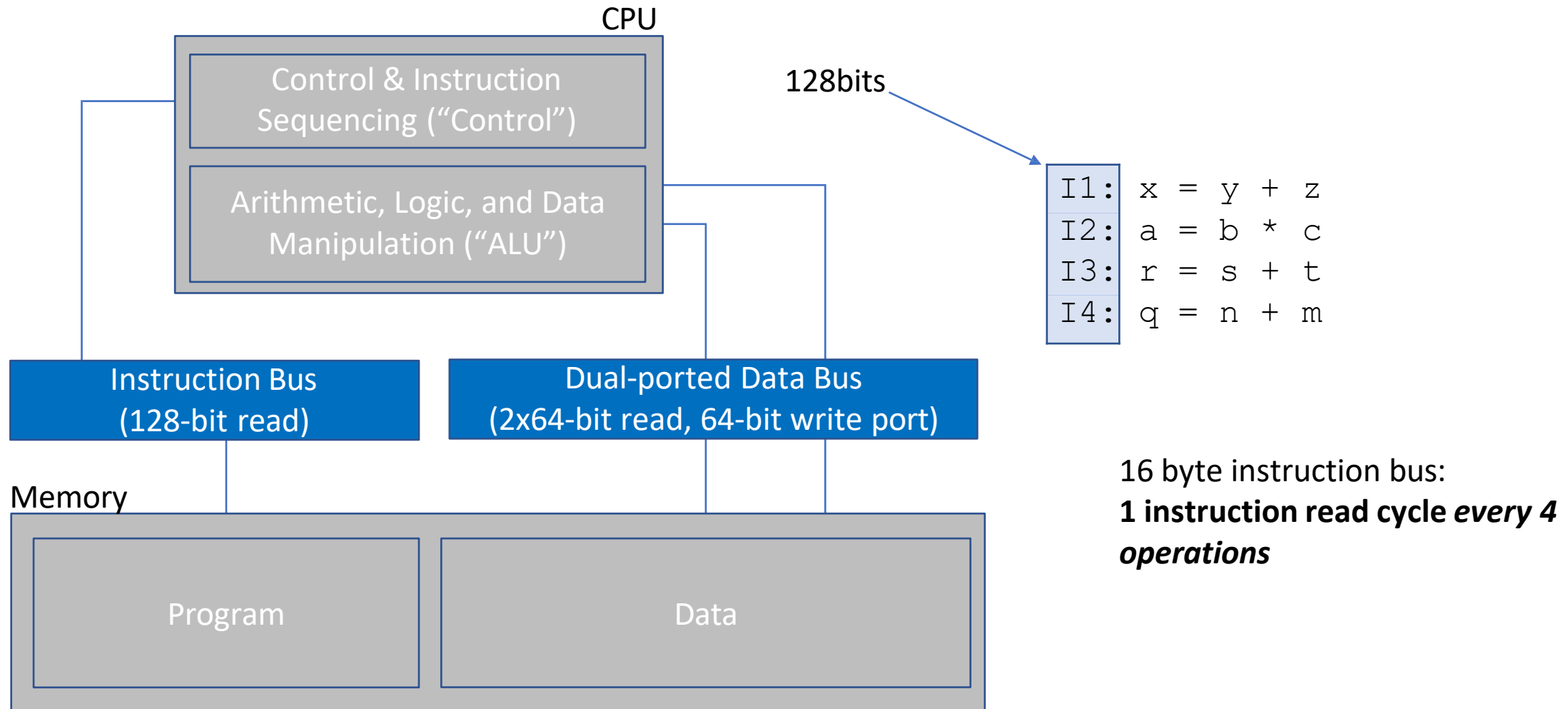


128bits

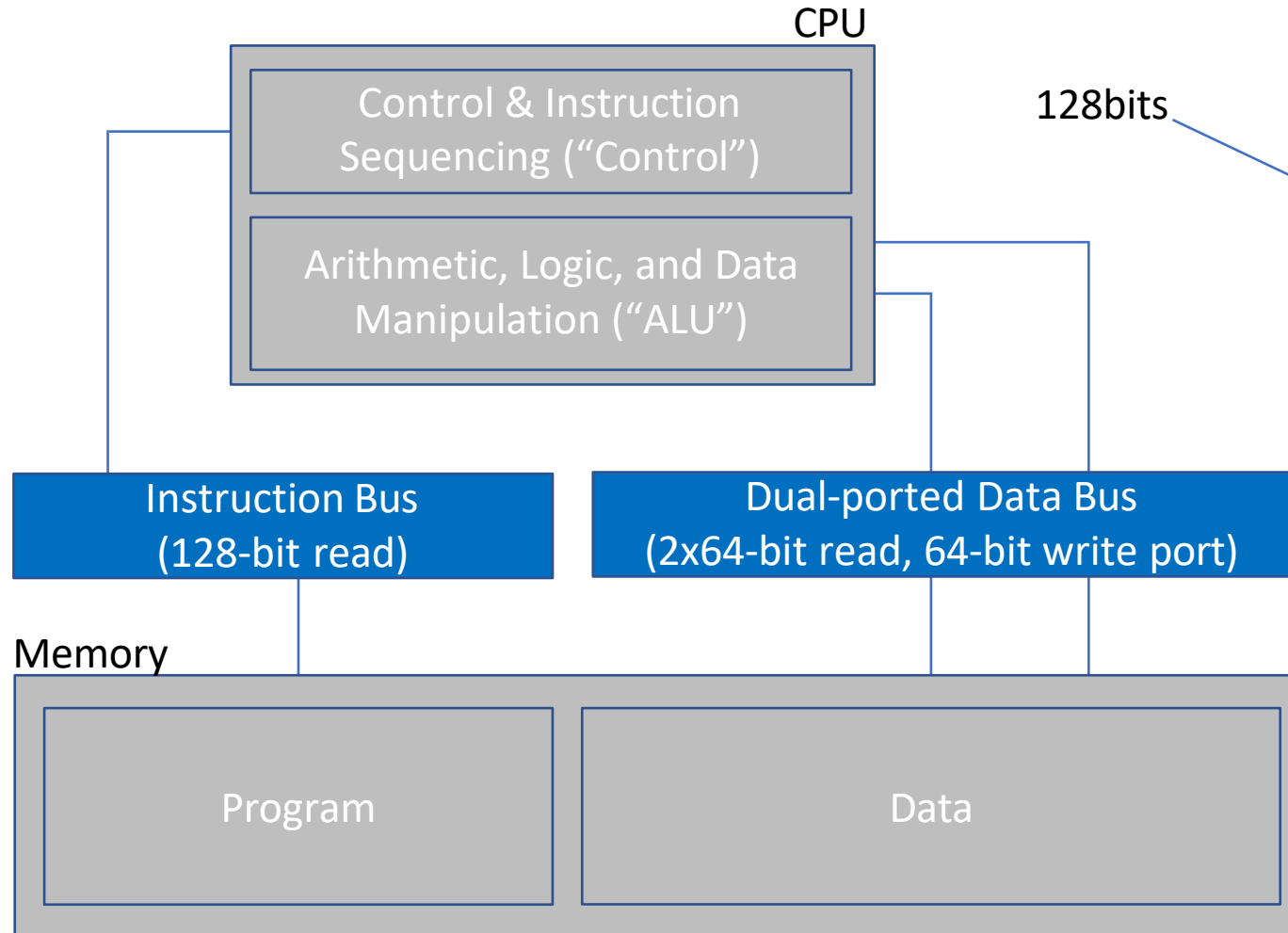
I1:	x	=	y	+	z
I2:	a	=	b	*	c
I3:	r	=	s	+	t
I4:	q	=	n	+	m

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

How about optimizing instruction supply?



Is this optimization a good tradeoff?

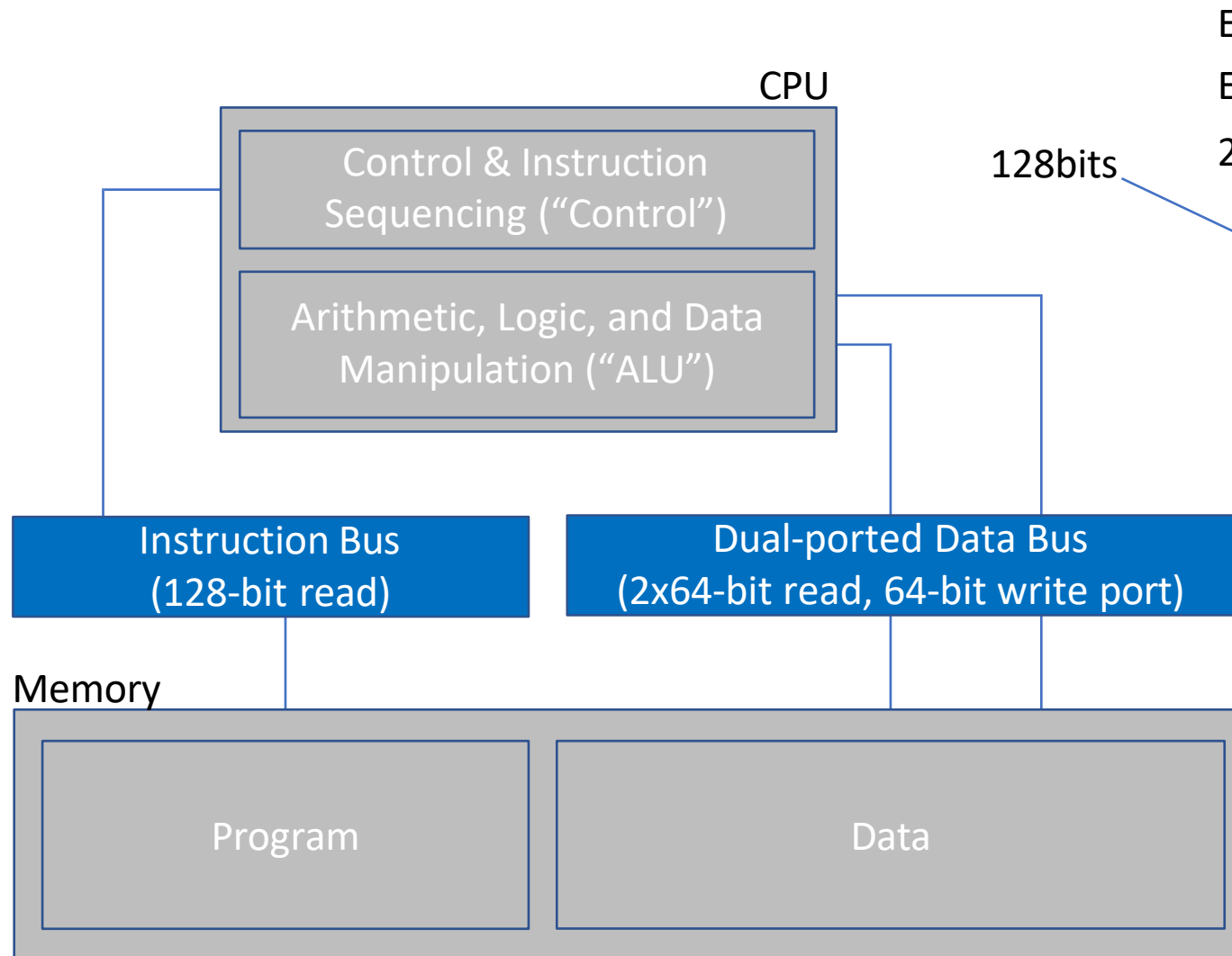


128bits

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?



$$E_{\text{old}} = n \times E_{\text{pF}}$$

$$E_{\text{new}} = (n/4) \times (E_{\text{pF}} \times 2) = \frac{1}{2}(n \times E_{\text{pF}})$$

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

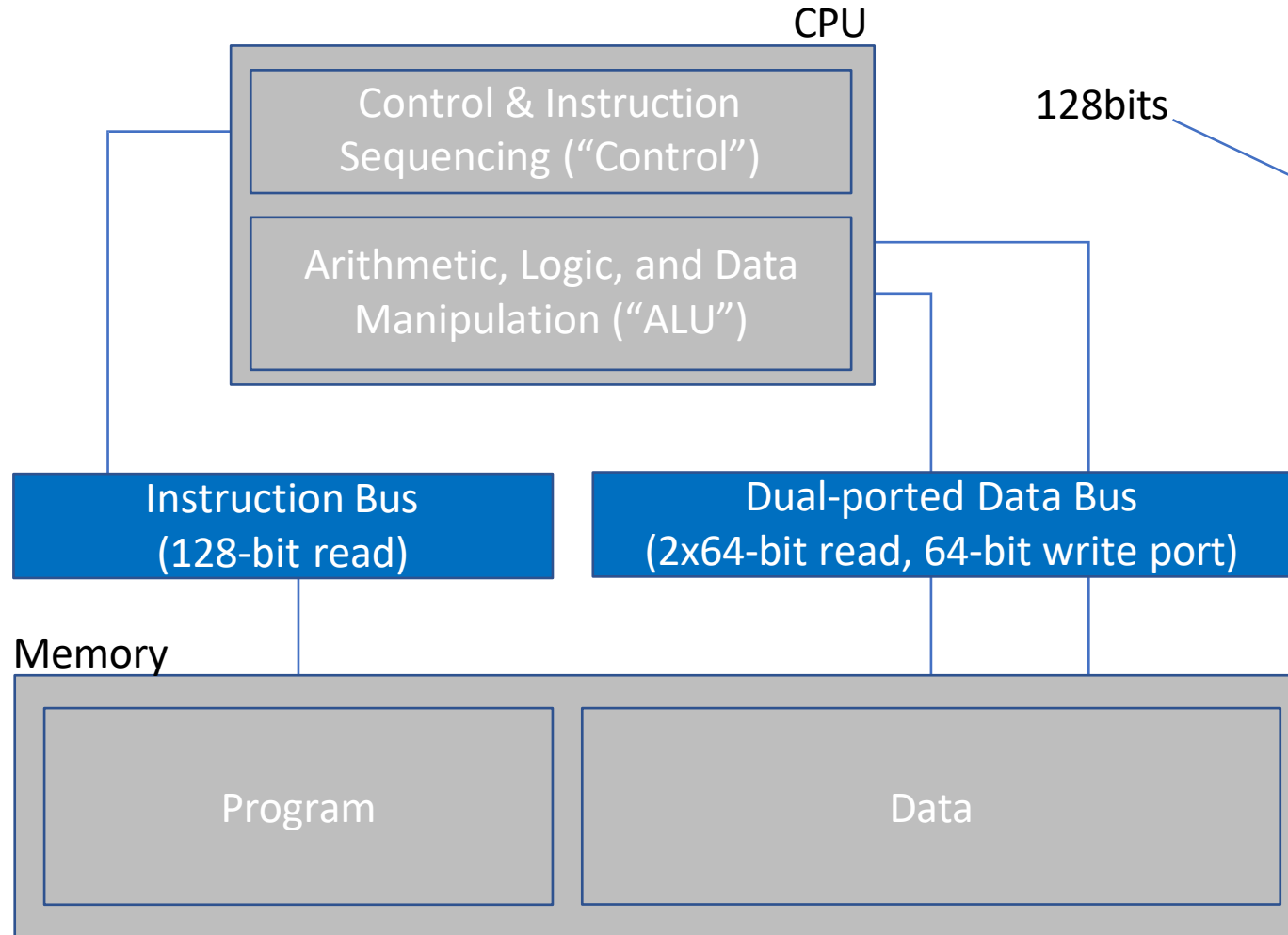
4x fewer instruction fetches

4B Read Energy = 0.836pJ

16B Read Energy = 1.51pJ

~2x more energy / fetch

Is this optimization a good tradeoff?



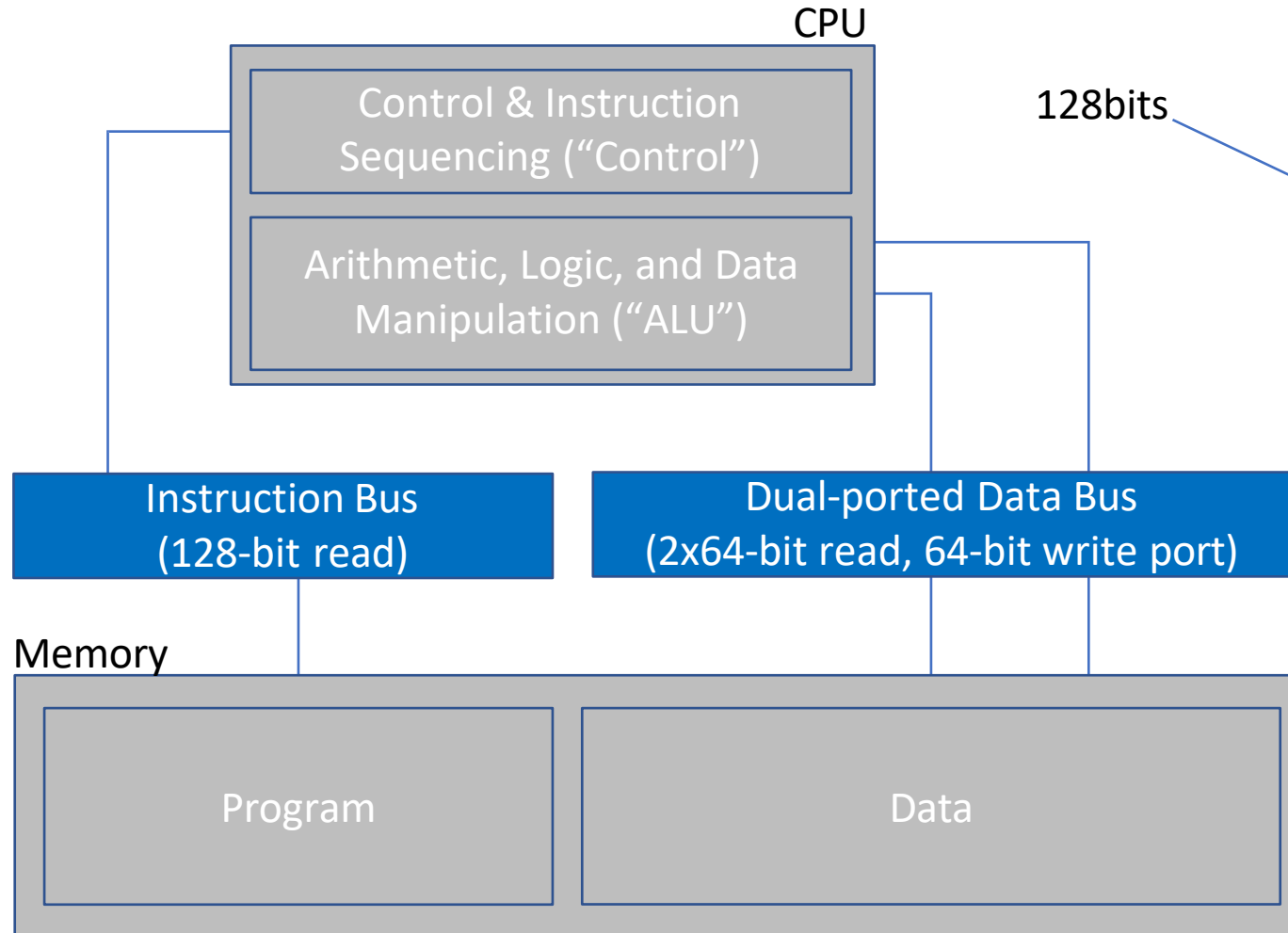
When might this optimization **not** pay off?

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?



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

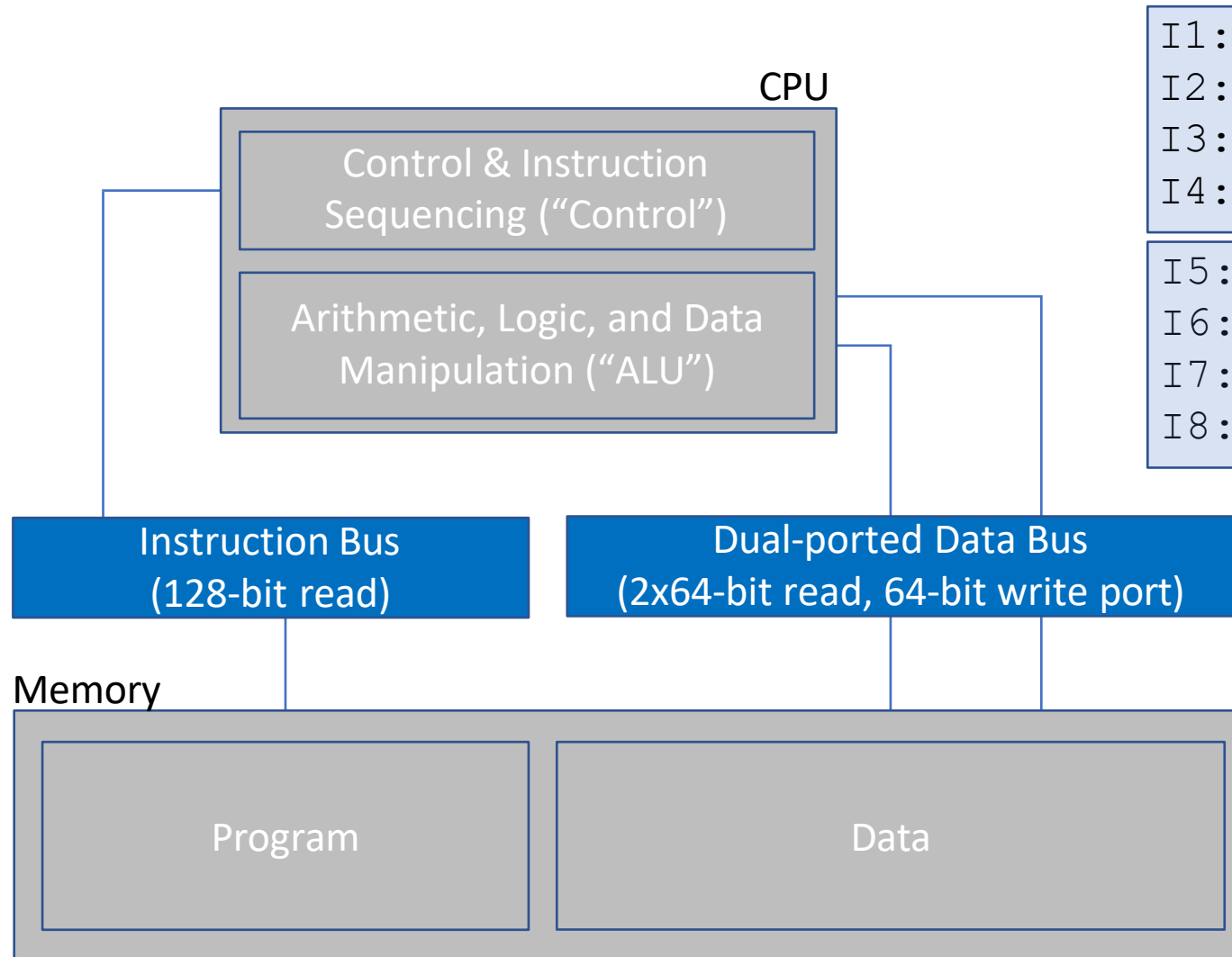
1 instruction read cycle every 4 operations
4x fewer reads, ***if we execute the operations!***
OR: 4x more useless fetching if x!=0
(more on this branching topic in a future lecture)

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

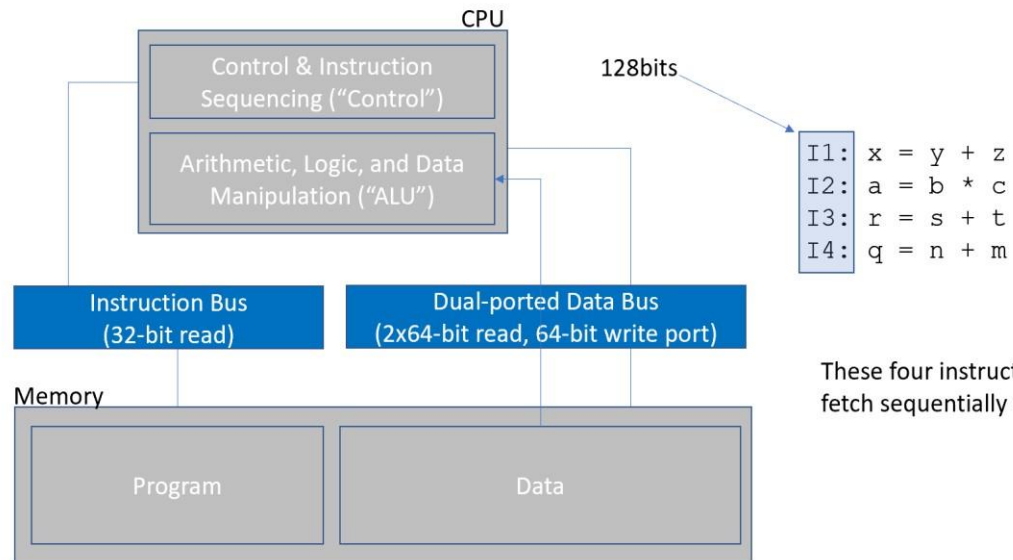
Q: What if x is most often non-zero?

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?



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

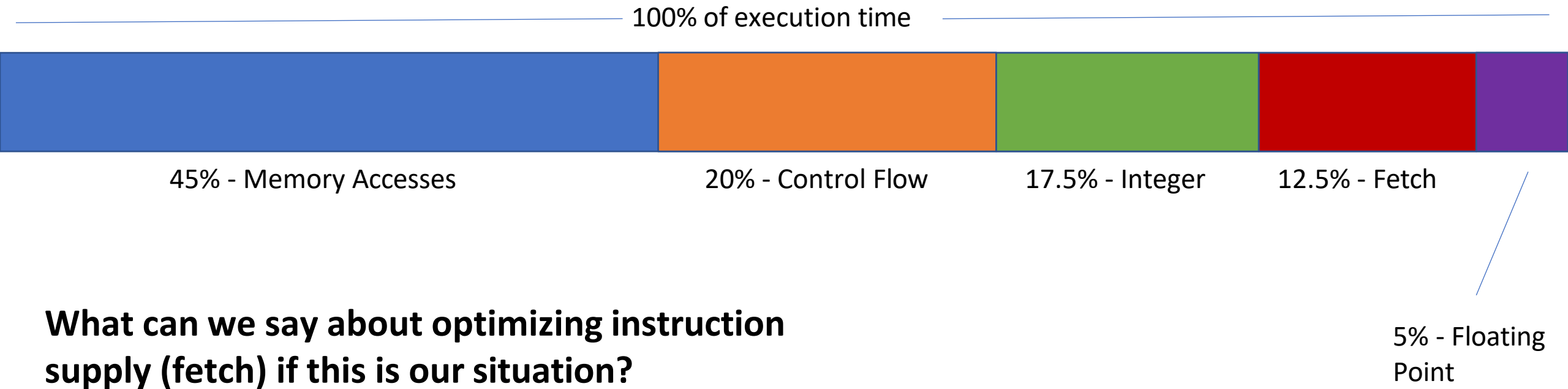
Amdahl's Law

100% of execution time

A horizontal blue bar spanning the width of the slide, positioned below the text '100% of execution time'. It represents the total execution time in a parallel computing context.

Amdahl's Law

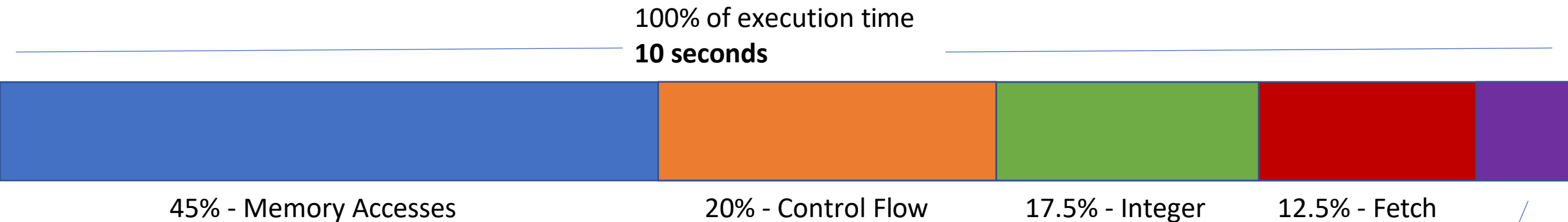
Imagine we have a perfectly precise measurement tool to break down execution time...



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

Amdahl's Law

What if we make fetch 4x faster?



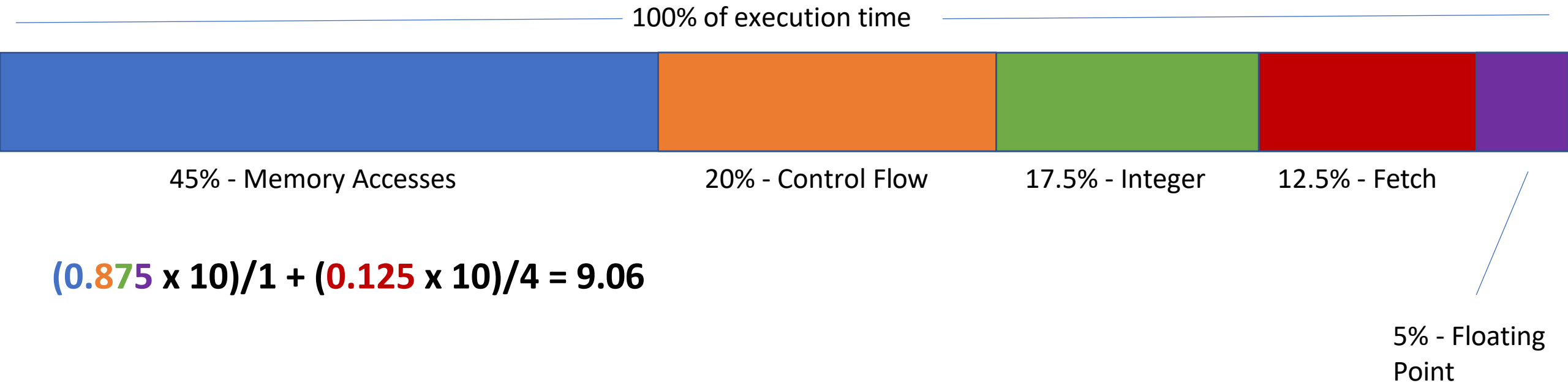
We have $45\% + 20\% + 17.5\% + 5\% = 87.5\%$ of the execution running at its original speed (ie 87.5% of 10 seconds = 8.75s)

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

5% - Floating Point

Amdahl's Law

What if we make fetch 4x faster?

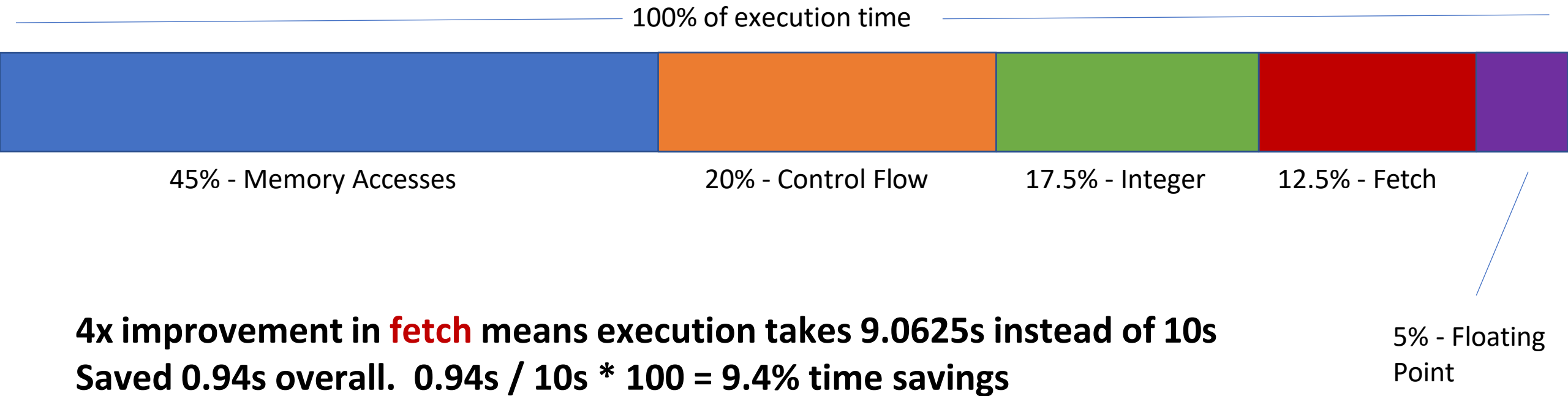


What does 9.06 mean?

Amdahl's Law

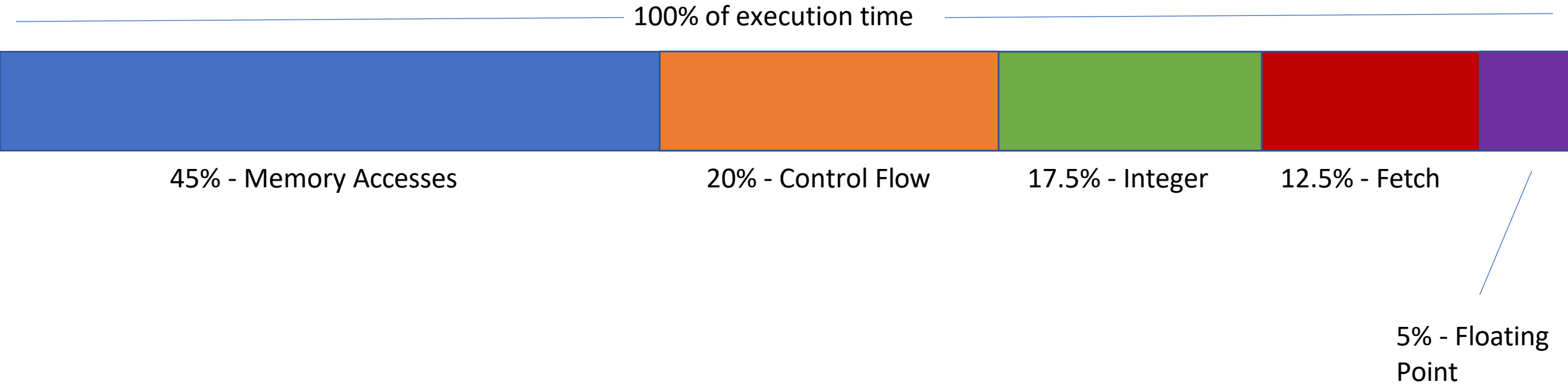
What if we make fetch 4x faster?

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



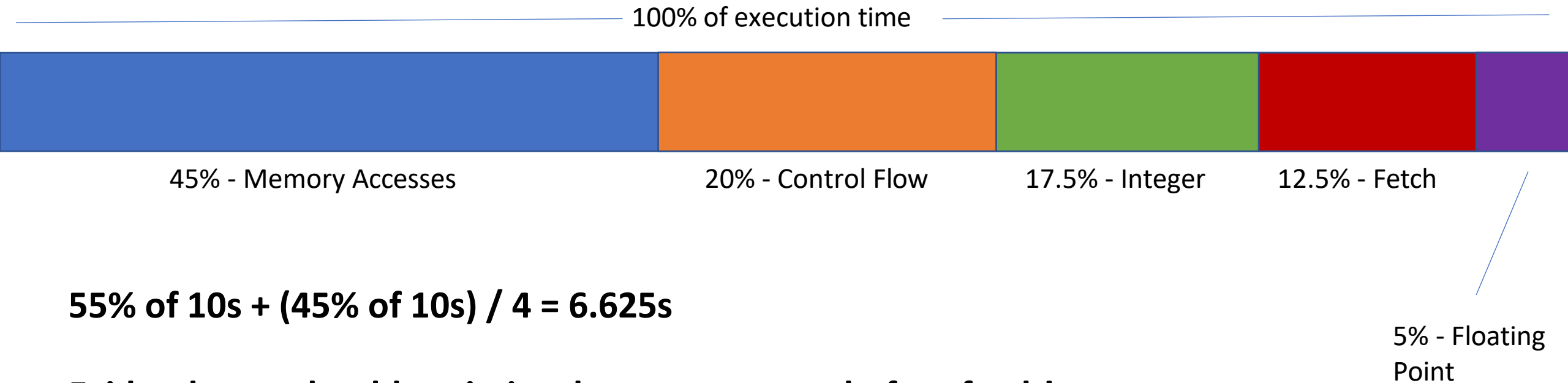
Amdahl's Law

What if we make memory accesses 4x faster?



Amdahl's Law

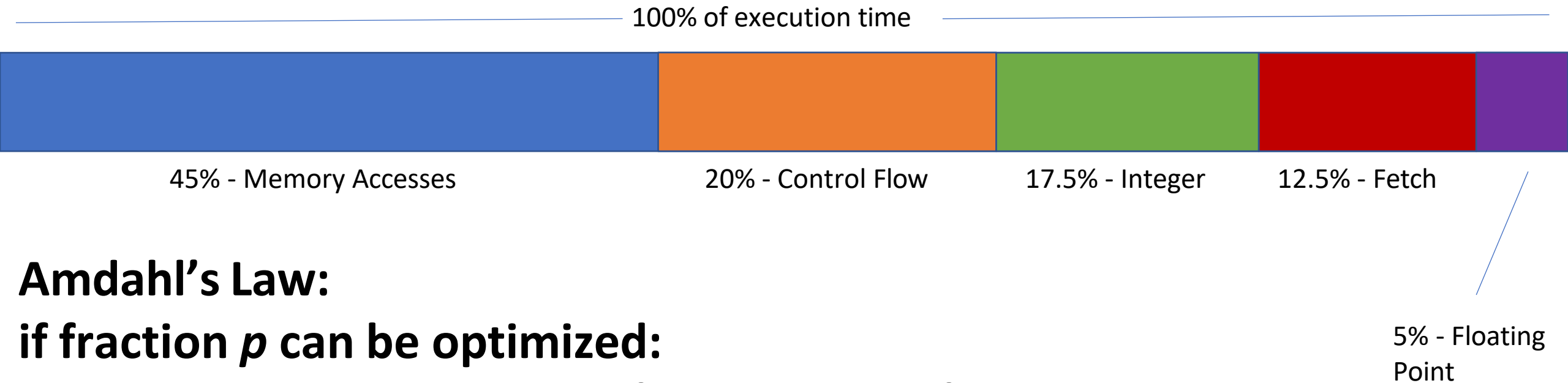
What if we make memory accesses 4x faster?



$$55\% \text{ of } 10\text{s} + (45\% \text{ of } 10\text{s}) / 4 = 6.625\text{s}$$

Evidently, we should optimize the memory part before fetch!

Amdahl's Law

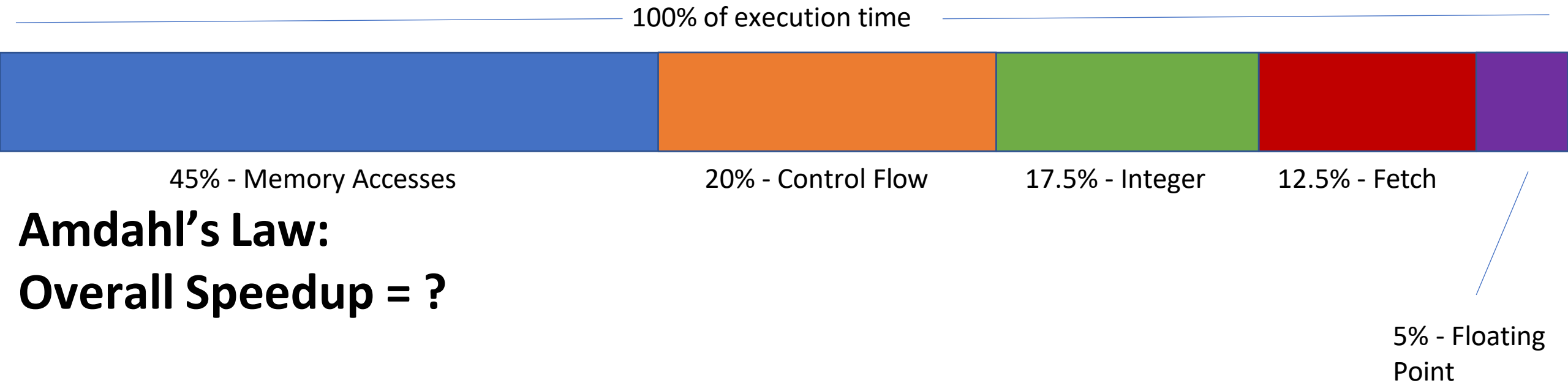


Amdahl's Law:

if fraction p can be optimized:

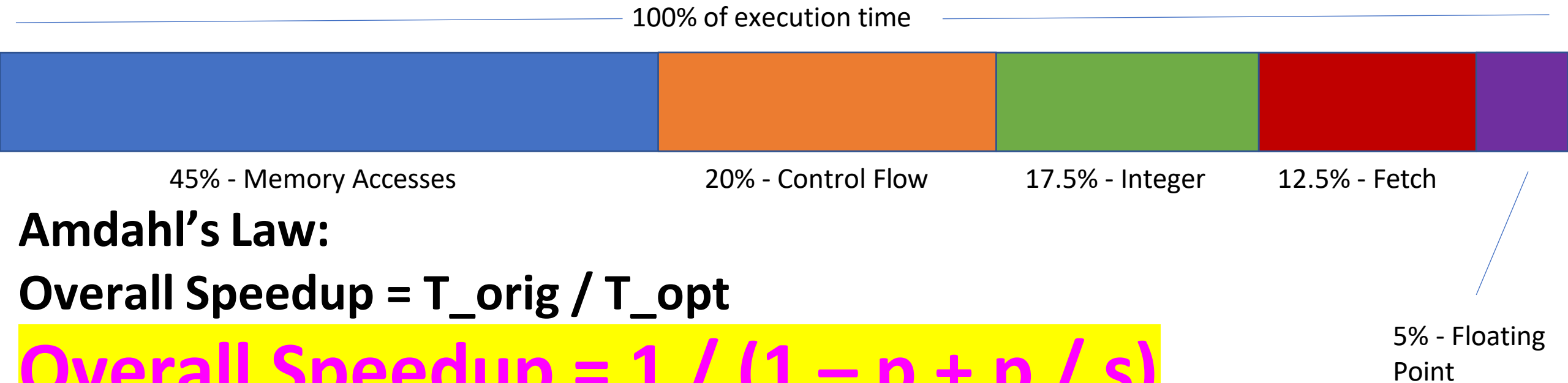
$$\text{Optimized Time} = [(1-p) * t / 1] + [(p * t) / \text{speedup}]$$

Amdahl's Law



Amdahl's Law:
Overall Speedup = ?

Amdahl's Law



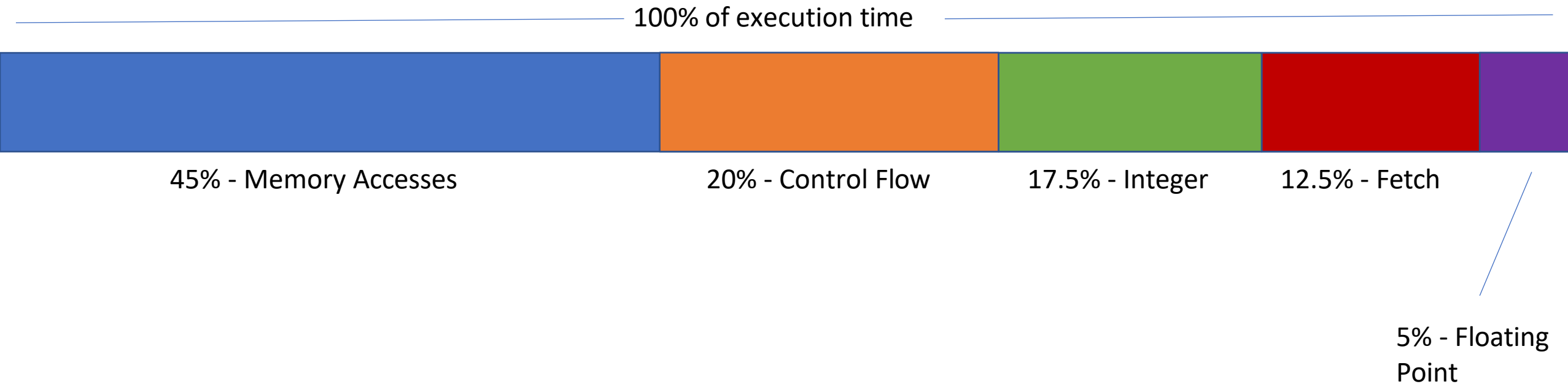
Amdahl's Law:

Overall Speedup = $T_{\text{orig}} / T_{\text{opt}}$

$$\text{Overall Speedup} = 1 / (1 - p + p / s)$$

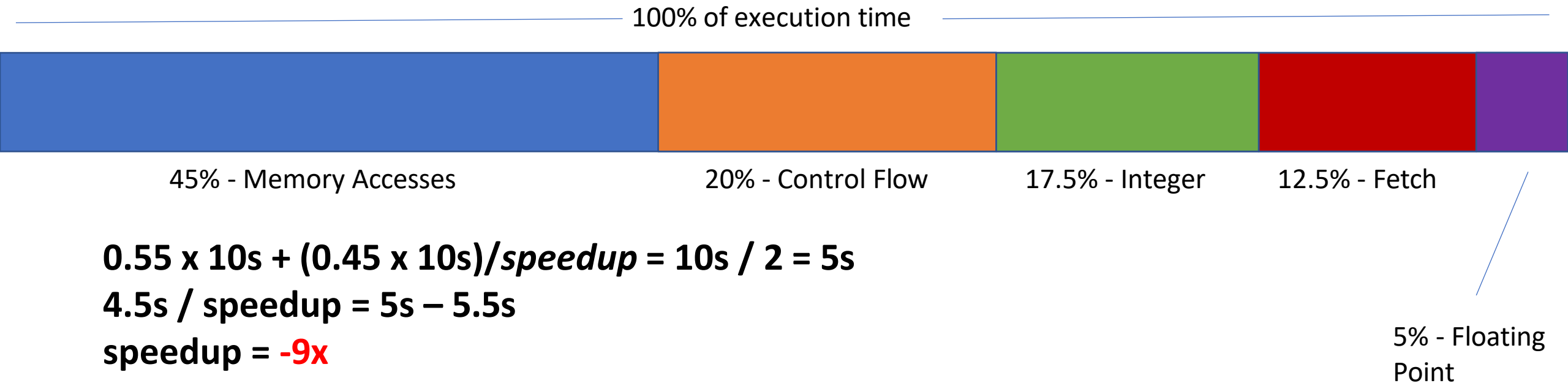
Amdahl's Law

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?



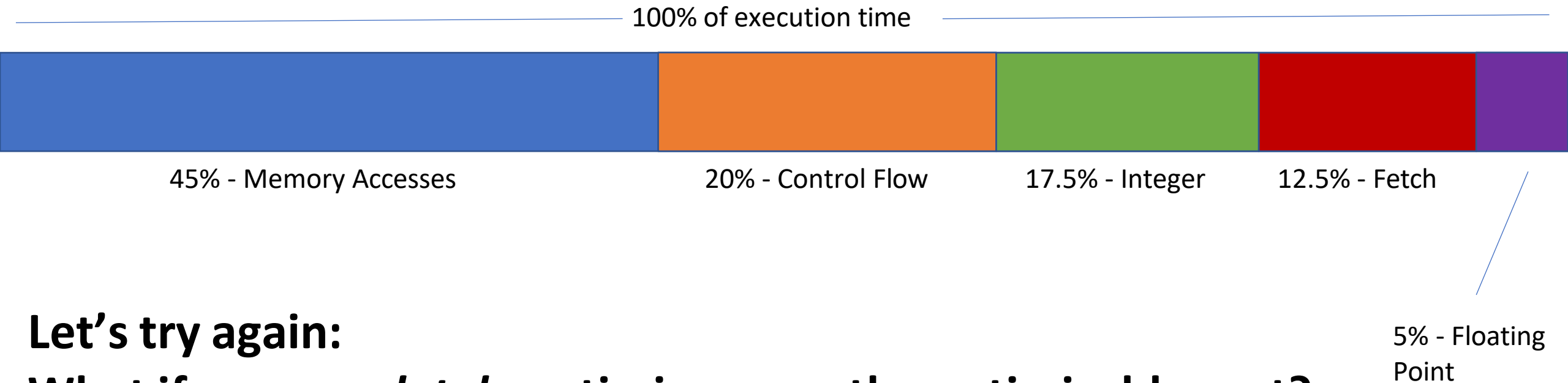
$$0.55 \times 10s + (0.45 \times 10s) / \text{speedup} = 10s / 2 = 5s$$

$$4.5s / \text{speedup} = 5s - 5.5s$$

$$\text{speedup} = -9x$$

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

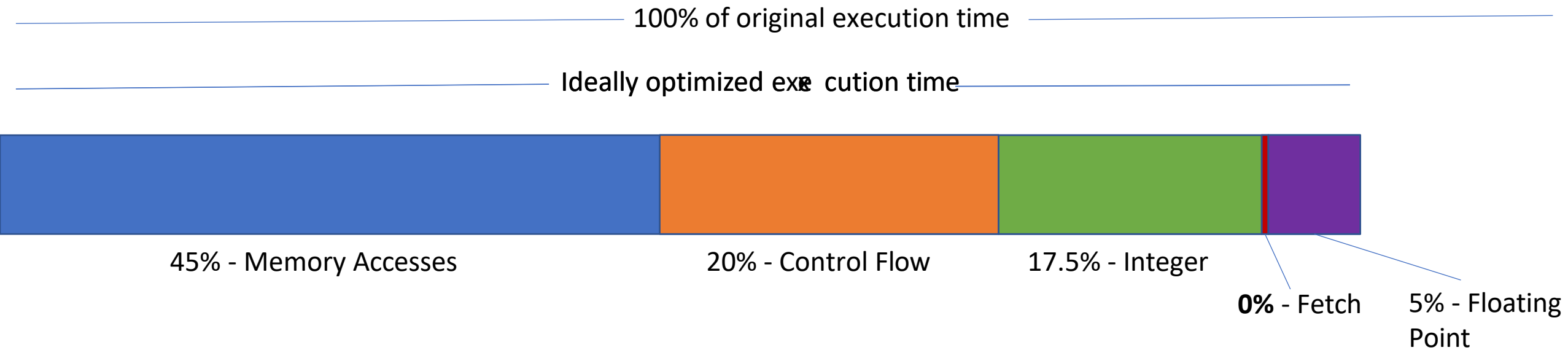
Limit Cases for Amdahl's Law



Let's try again:

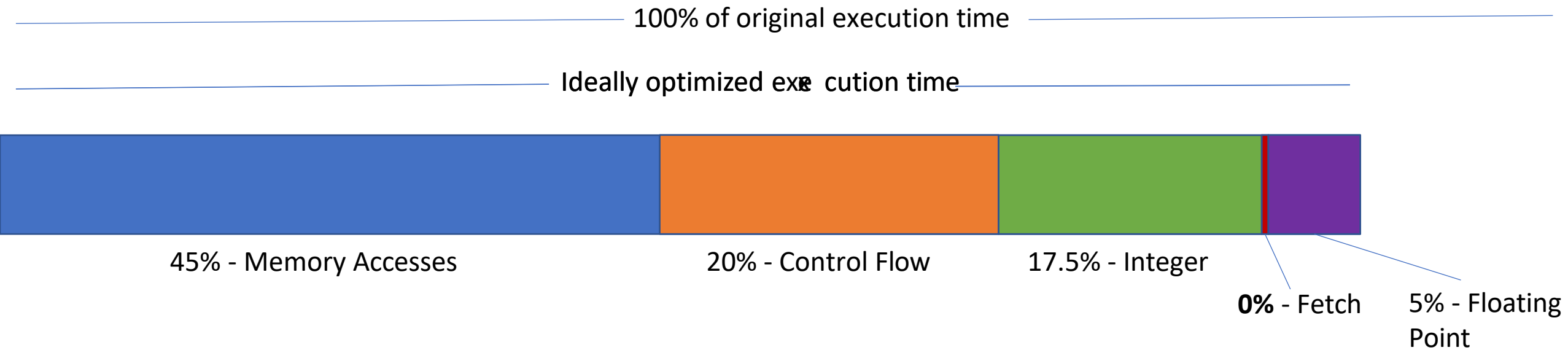
What if we *completely* optimize away the optimizable part?

Limit Cases for Amdahl's Law



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

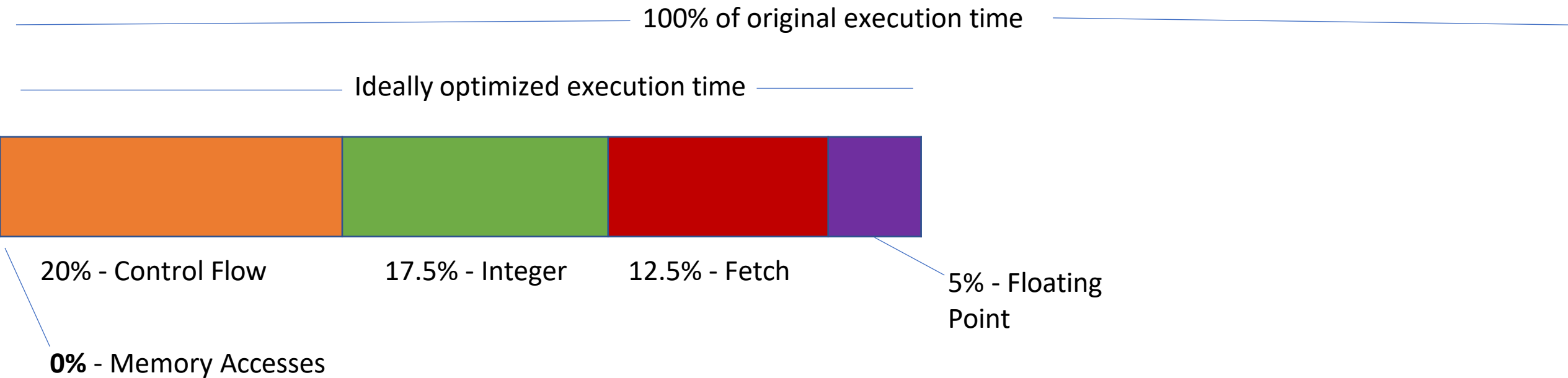
Limit Cases for Amdahl's Law



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

8.75s optimized execution time

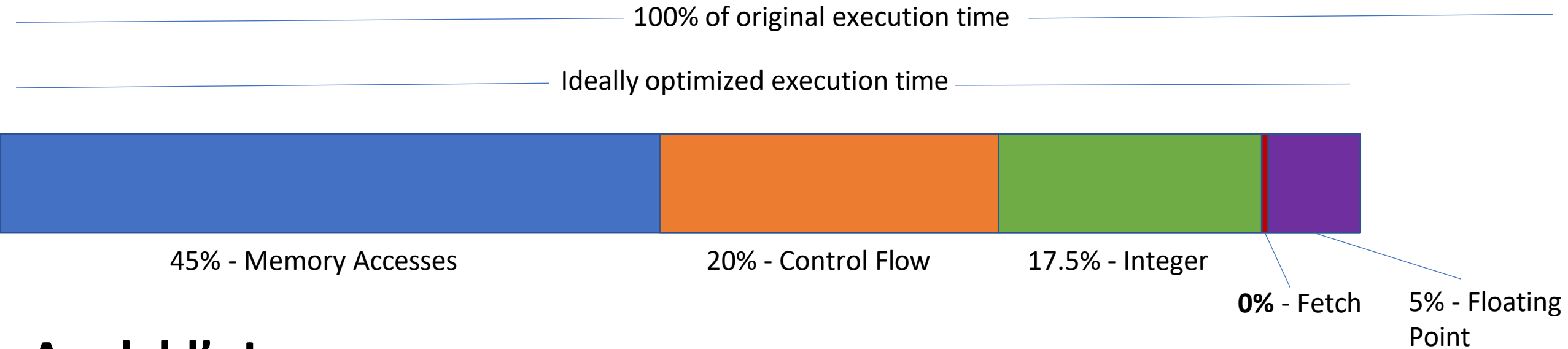
Limit Cases for Amdahl's Law



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

5.5s optimized execution time

Limit Cases for Amdahl's Law

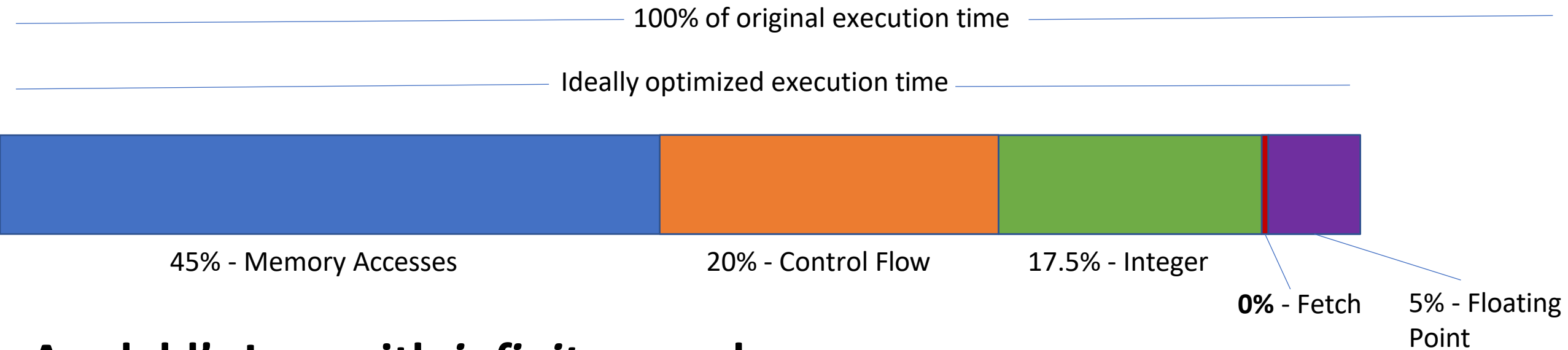


Amdahl's Law:

$$\text{optimized time} = [1-p \times \text{time} / 1.0] + [p \times \text{time} / s]$$

$$\text{Overall speedup} = 1 / (1 - p + p / s)$$

Limit Cases for Amdahl's Law

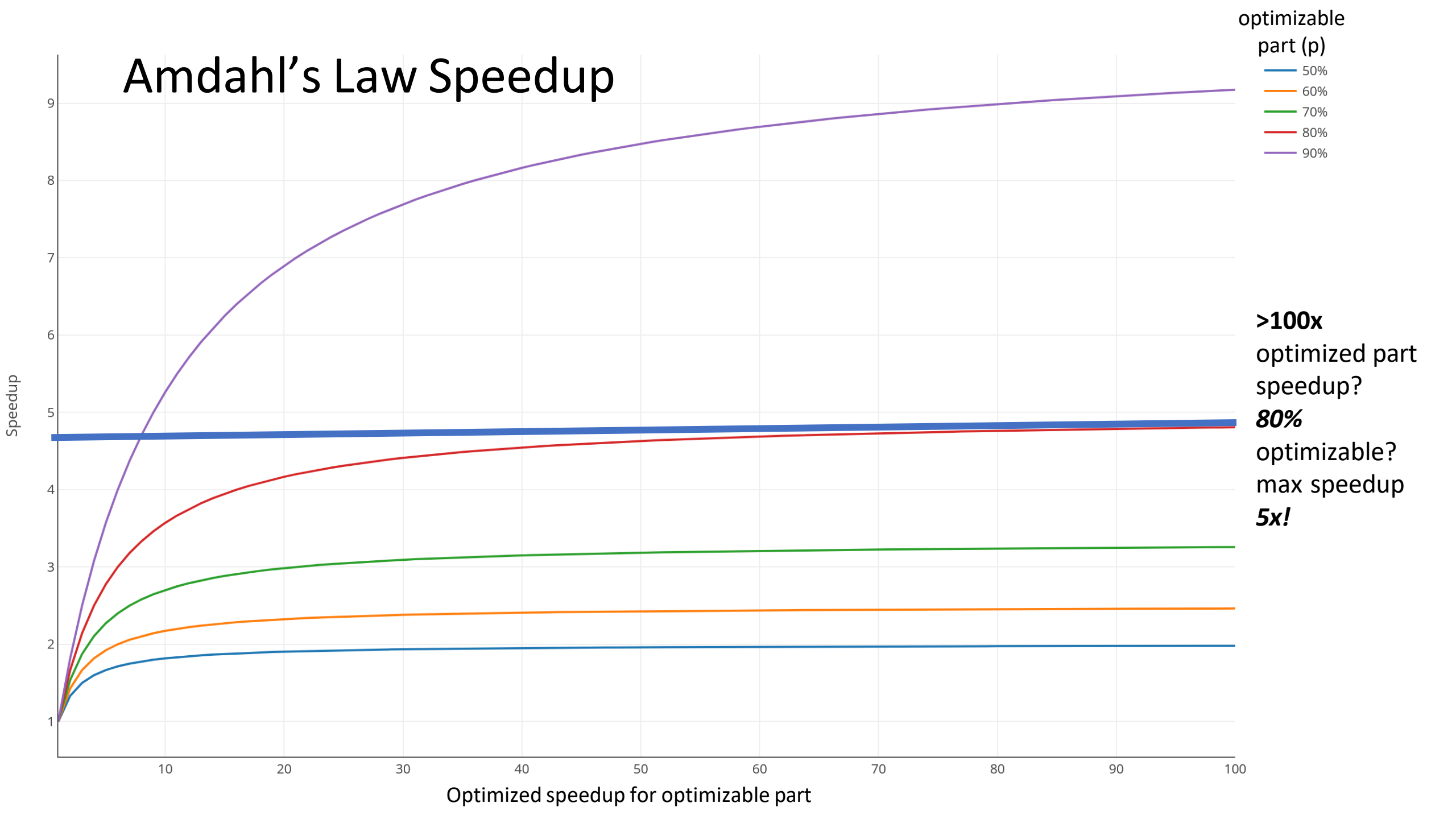


Amdahl's Law with *infinite* speedup:

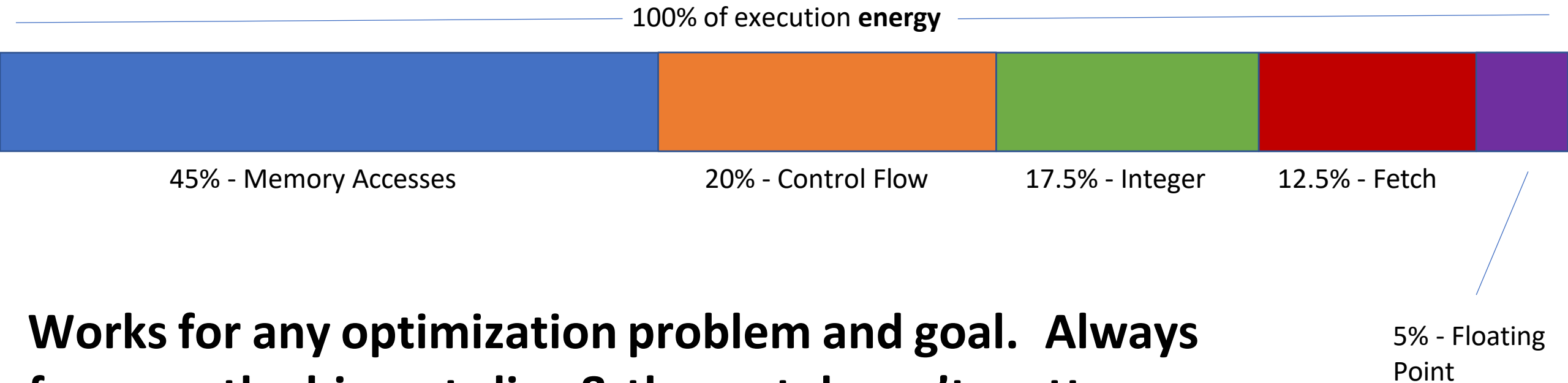
optimized time with infinite speedup of $p = [1 - p \times \text{time} / 1.0]$

Overall speedup with infinite speedup of $p = 1 / (1 - p)$

Amdahl's Law Speedup

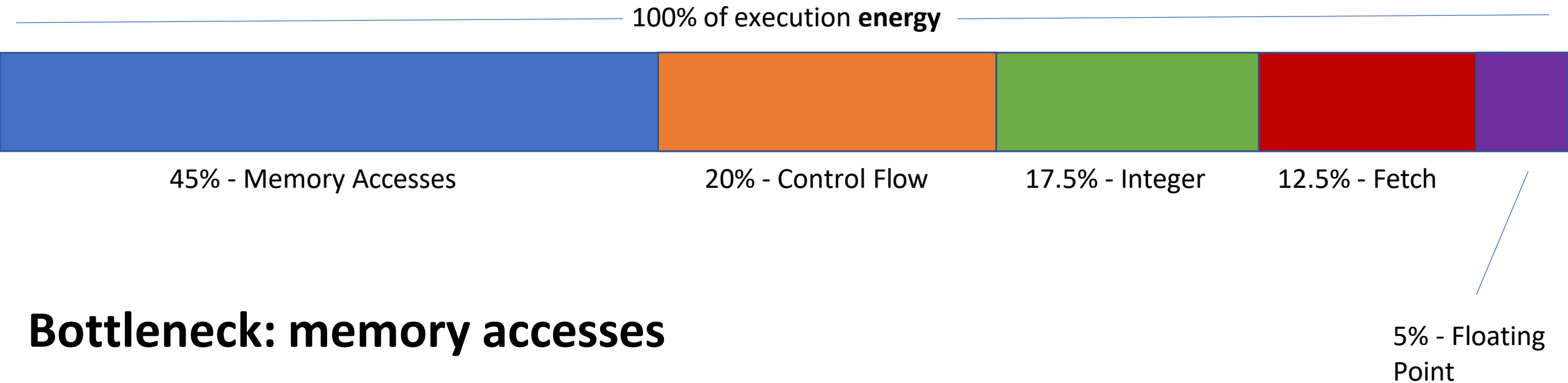


Amdahl's Law is Extremely Versatile

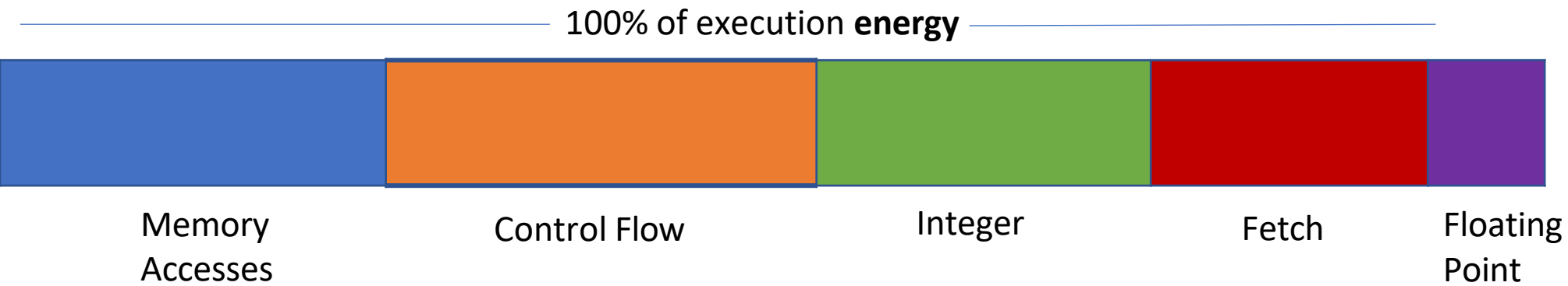


Works for any optimization problem and goal. Always focus on the biggest slice & the rest doesn't matter.

Using Amdahl's Law to push the bottleneck around

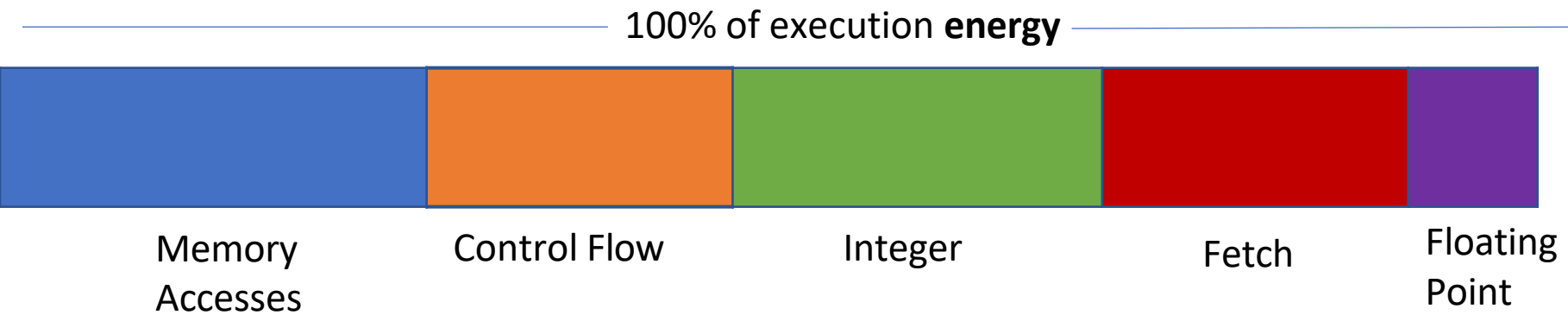


Using Amdahl's Law to push the bottleneck around



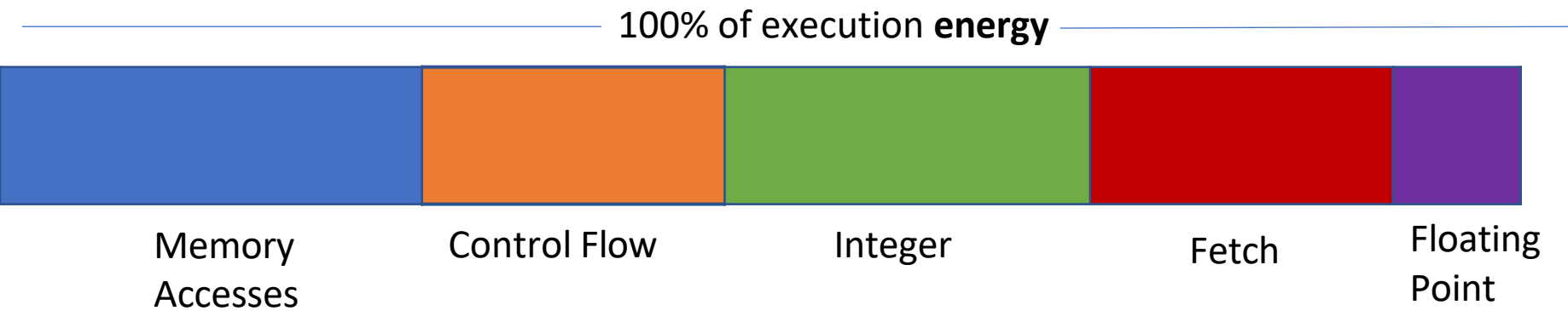
New bottleneck: control flow

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: Gustafson's Law

Intuition:

- Work tends to be time limited, and as we all know, work tends to fill the time.
- So, for many problems, the amount of work we do grows alongside our ability to do it.
- We need to judge the real-world impact by what we will do, not what we once did.



85% - Memory Accesses

“The speedup we experience is equal to the amount of parallelization, except for the percentage of the problem which isn’t parallelizable and needs to be done sequentially”.

Gustafson's Law says the scaled speedup S with P processors is:

- $S(P) = P - \alpha \cdot (P - 1)$

Where:

- P = number of processors
- α = fraction of time spent on the part of the program that is inherently **sequential** (cannot be parallelized)

Another view of the world: Gustafson'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++ }
```



Gustafson!

```
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:

Scale parallel memory accesses, N, up to 1000?

$$\text{Scaled Speedup} = 1-p + 1000p = 999p + 1$$

$$\text{Scaled Speedup} = 999 * 0.85 + 1 = 850x$$



85% - Memory Accesses

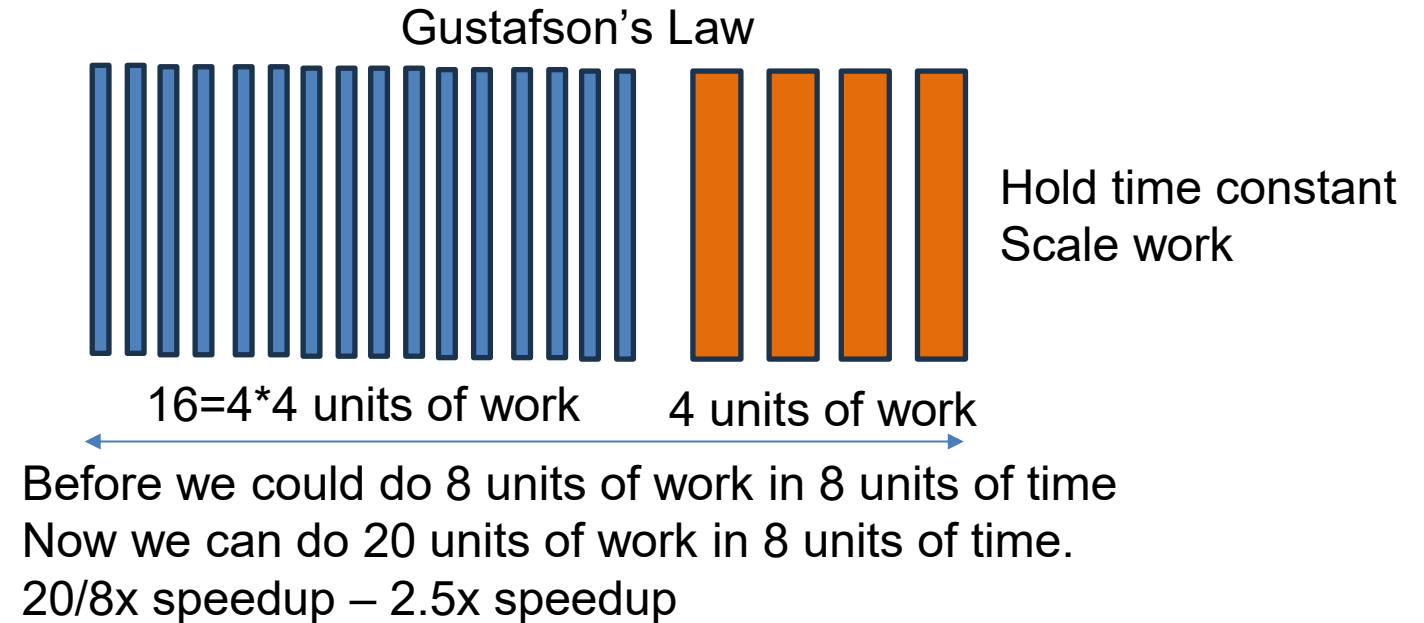
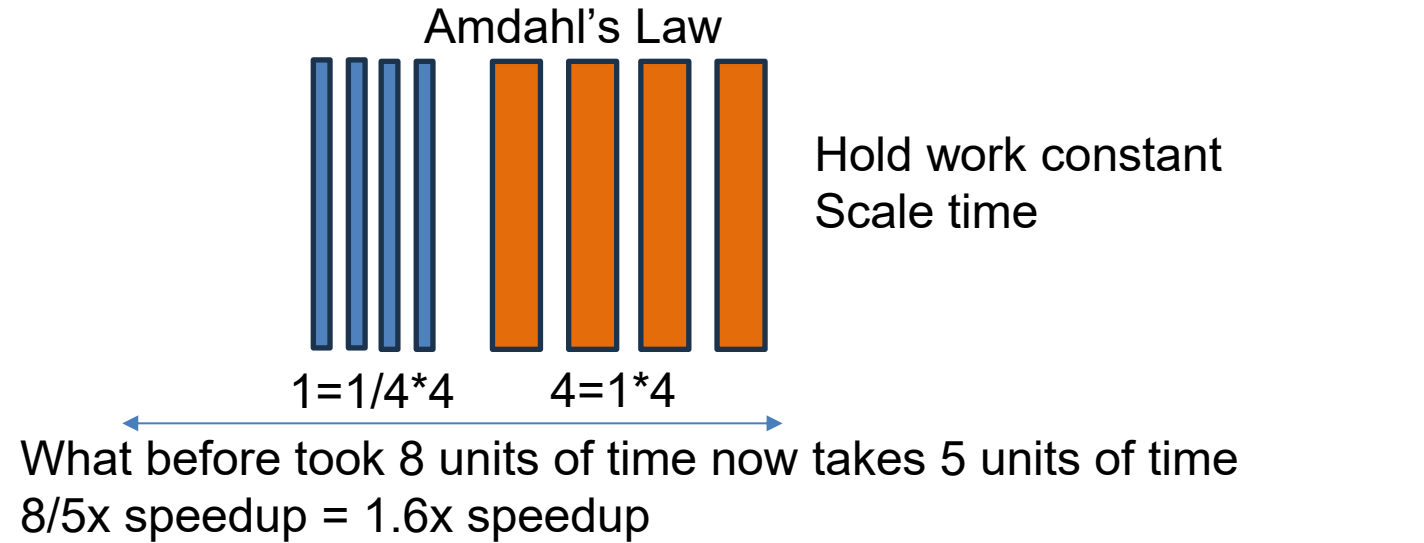
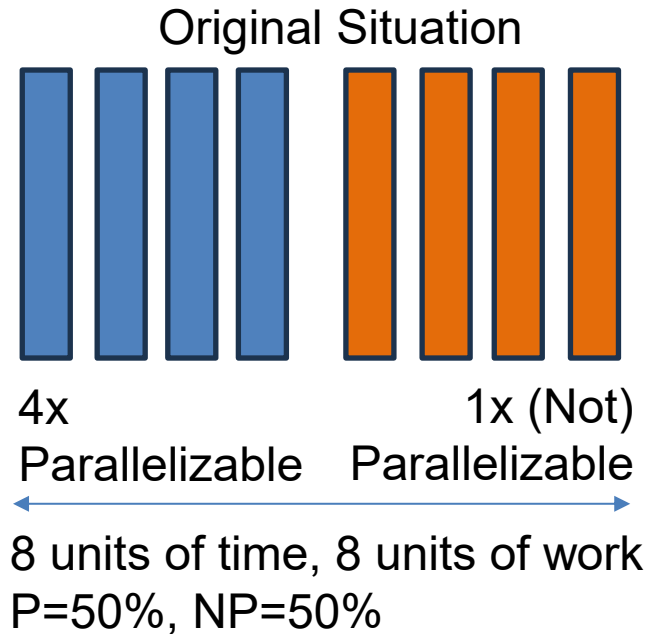
Gustafson's Law for overall speedup with speedup factor of N:

(assume) Optimized time = $T = 1$

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

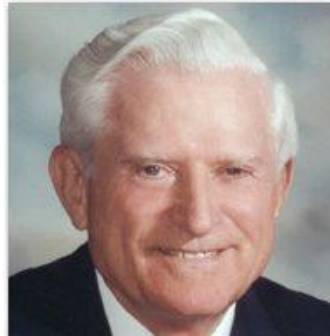
Scaled Speedup = $T' / T = (1-p) + pN$

Amdahl's Law vs Gustafson's Law



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?