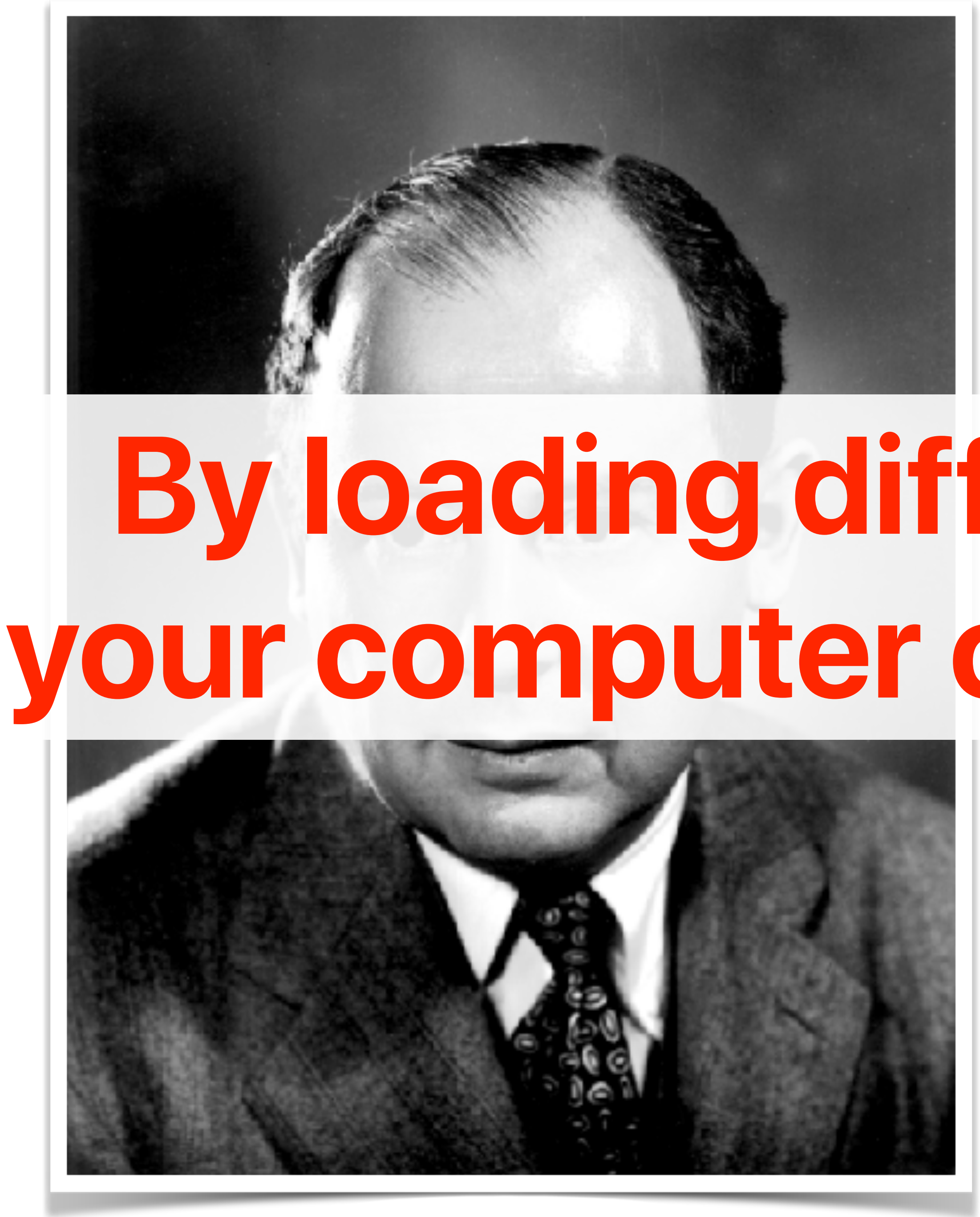
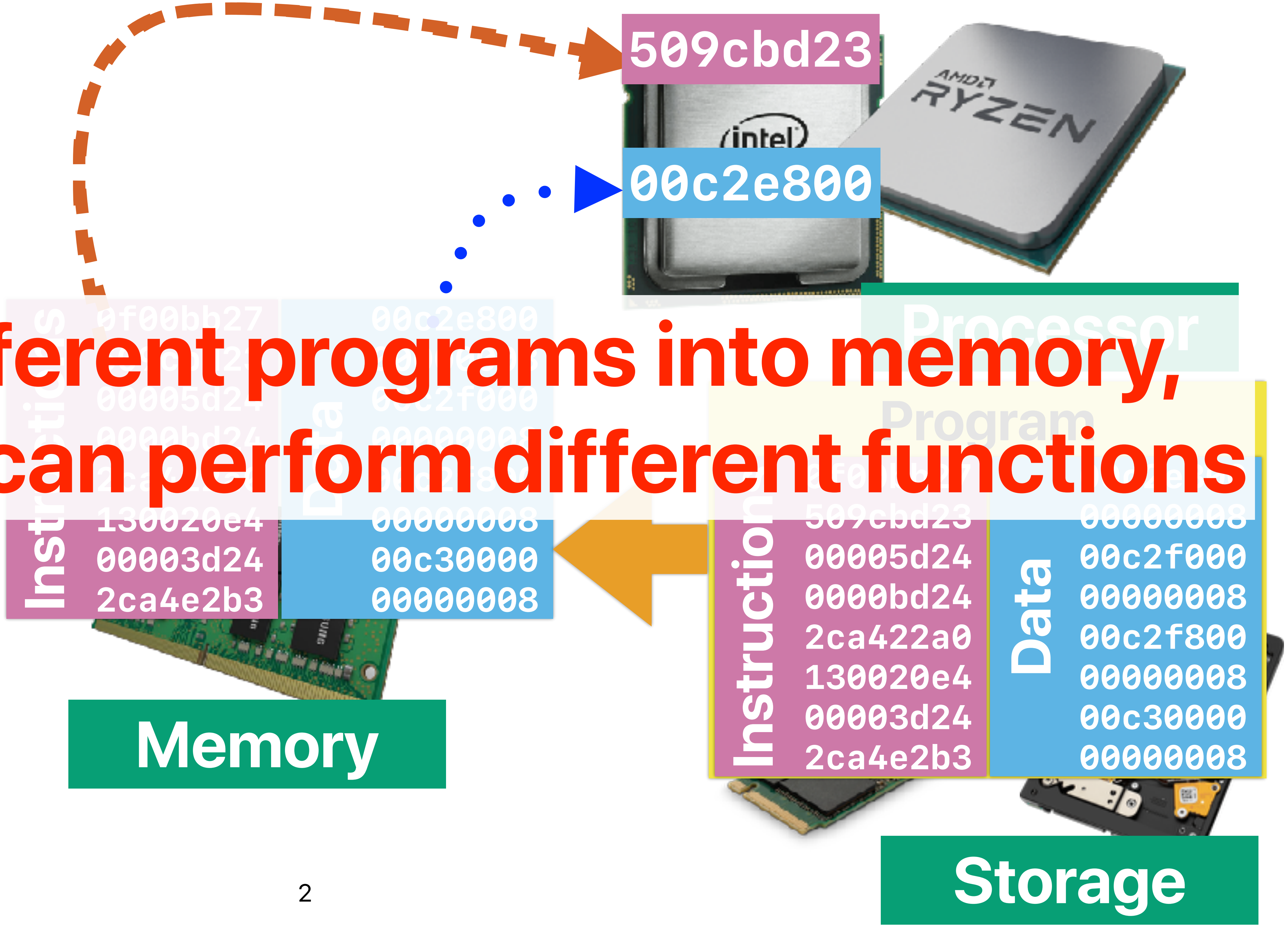
Performance (I): The Basics

Hung-Wei Tseng

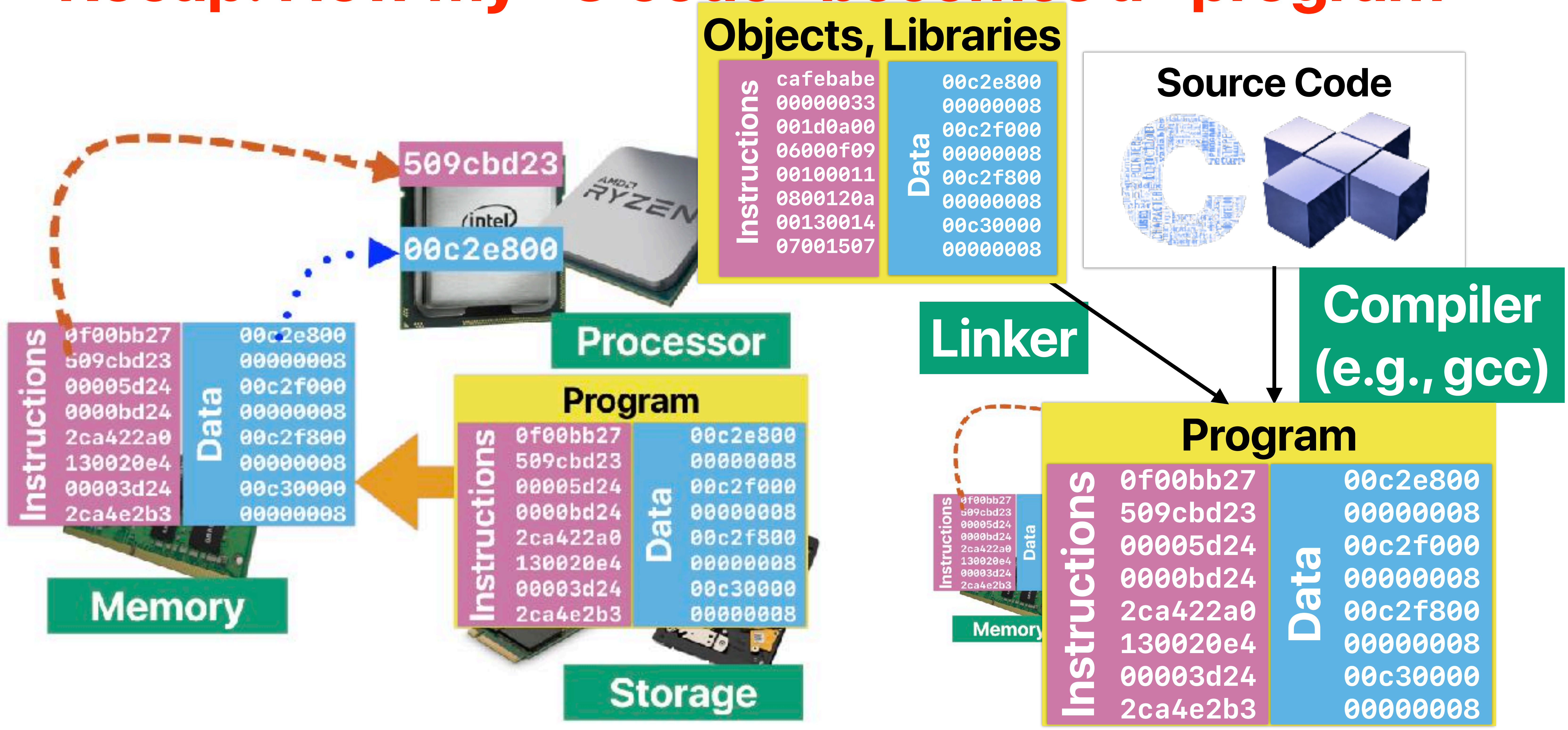
Recap: von Neumman Architecture





By loading different programs into memory, your computer can perform different functions

Recap: How my “C code” becomes a “program”



Objects, Libraries

Instructions
```
cafebabe
00000033
001d0a00
06000f09
00100011
0800120a
00130014
07001507
```

Data
```
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008
```

Source Code

Compiler (e.g., gcc)

Linker

Program

Instructions
```
0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3
```

Data
```
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008
```

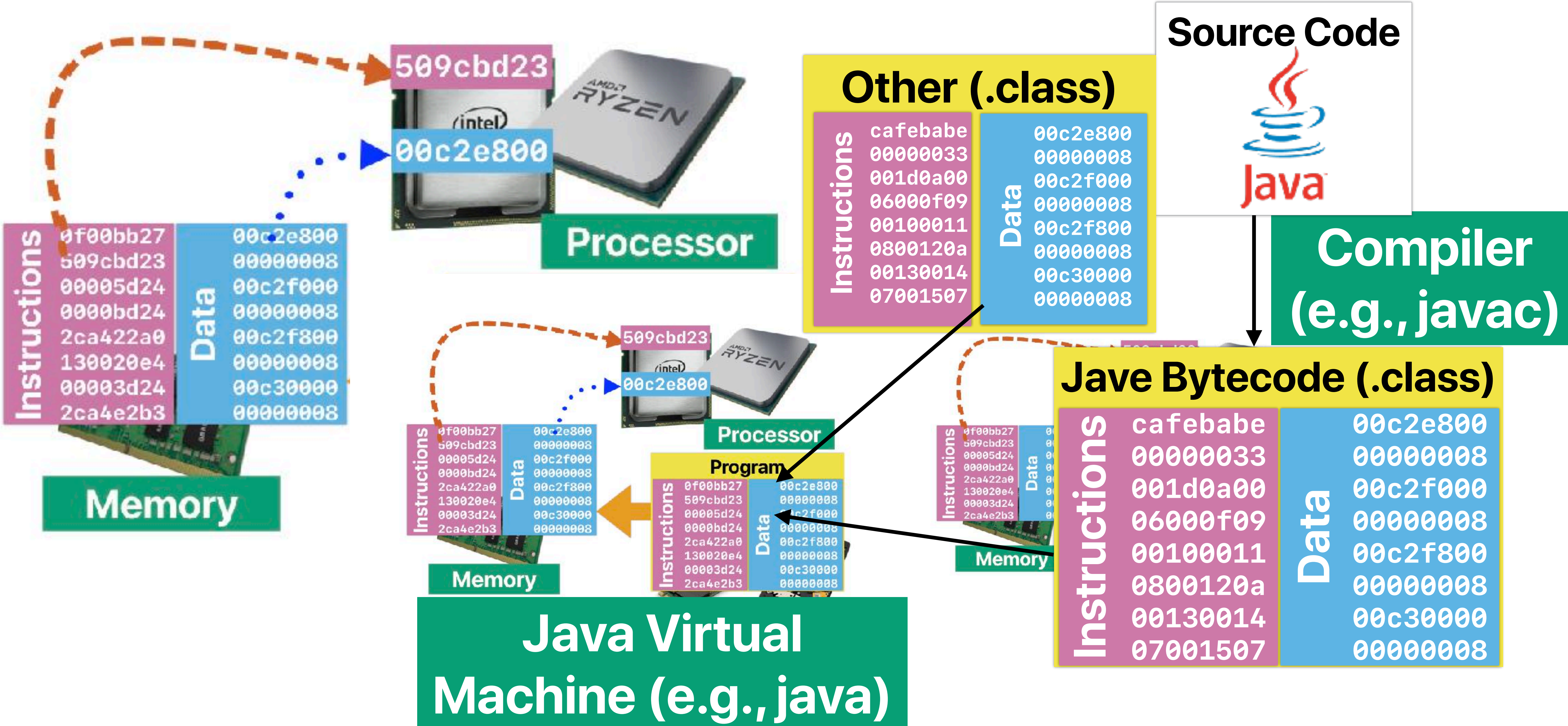
Storage

Memory

Processor

```
509cbd23
00c2e800
```

Recap: How my “Java code” becomes a “program”



Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

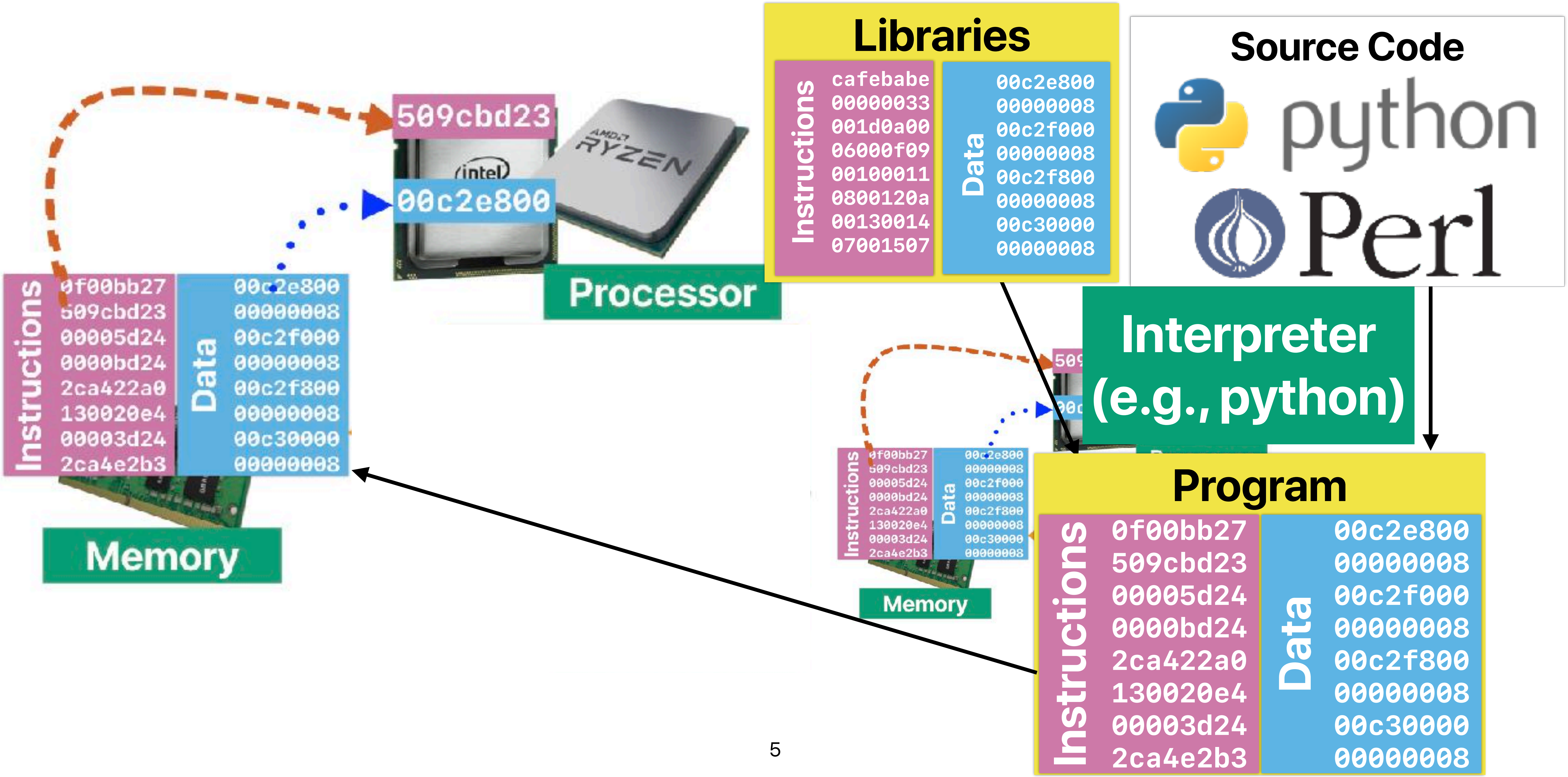
Jave Bytecode (.class)

Instructions	Data
cafebabe	00c2e800
00000033	00000008
001d0a00	00c2f000
06000f09	00000008
00100011	00c2f800
0800120a	00000008
00130014	00c30000
07001507	00000008

Other (.class)

Instructions	Data
cafebabe	00c2e800
00000033	00000008
001d0a00	00c2f000
06000f09	00000008
00100011	00c2f800
0800120a	00000008
00130014	00c30000
07001507	00000008

Recap: How my “Python code” becomes a “program”



Libraries

Instructions	Data
cafebabe	00c2e800
00000033	00000008
001d0a00	00c2f000
06000f09	00000008
00100011	00c2f800
0800120a	00000008
00130014	00c30000
07001507	00000008

Program

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Memory

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Outline

- Definition of “Performance”
- What affects each factor in “Performance Equation”
- Instruction Set Architecture & Performance

Definition of “Performance”

CPU Performance Equation

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

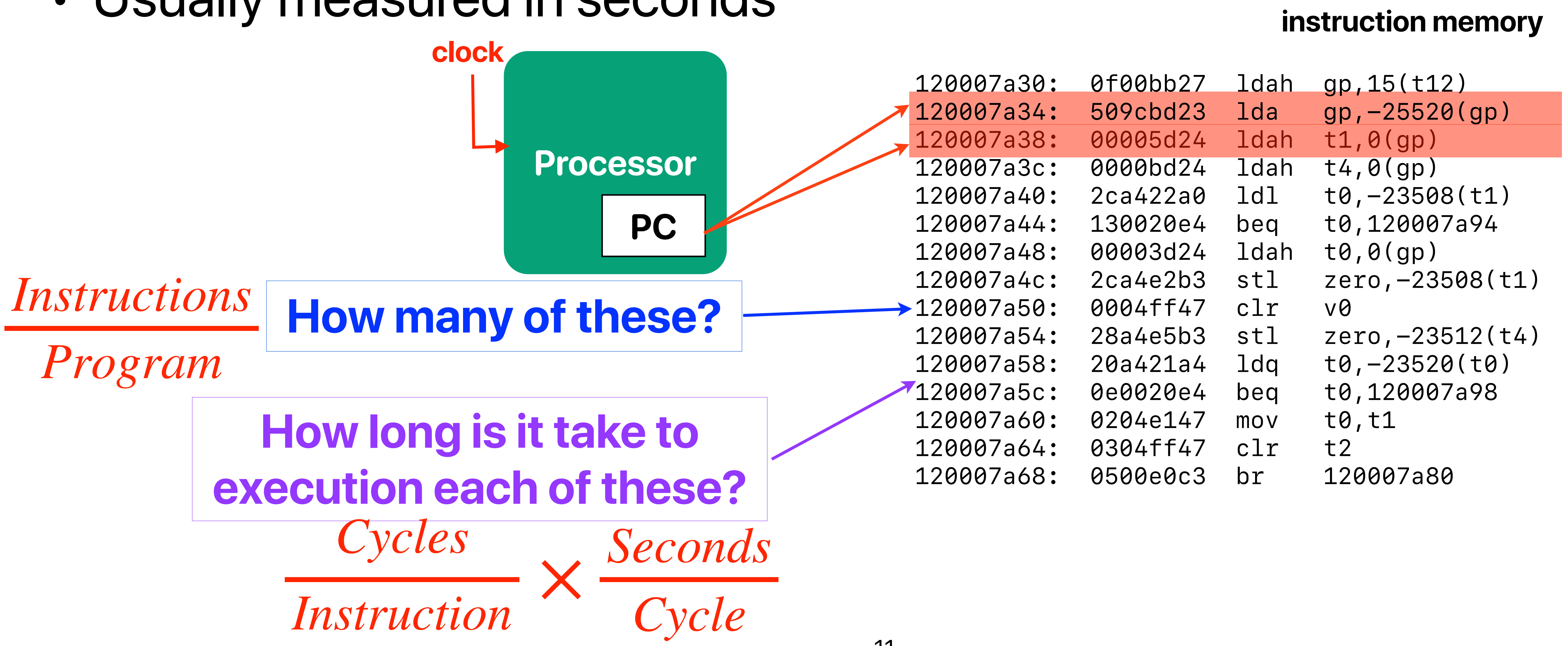
$$ET = IC \times CPI \times CT$$

$1GHz = 10^9 Hz = \frac{1}{10^9} sec\ per\ cycle = 1\ ns\ per\ cycle$

$$\frac{1}{Frequency(i.e., clock\ rate)}$$

Execution Time

- The simplest kind of performance
- Shorter execution time means better performance
- Usually measured in seconds



instruction memory

```
120007a30:  0f00bb27  ldah  gp,15(t12)
120007a34:  509cbd23  lda   gp,-25520(gp)
120007a38:  00005d24  ldah  t1,0(gp)
120007a3c:  0000bd24  ldah  t4,0(gp)
120007a40:  2ca422a0  ldl   t0,-23508(t1)
120007a44:  130020e4  beq   t0,120007a94
120007a48:  00003d24  ldah  t0,0(gp)
120007a4c:  2ca4e2b3  stl   zero,-23508(t1)
120007a50:  0004ff47  clr   v0
120007a54:  28a4e5b3  stl   zero,-23512(t4)
120007a58:  20a421a4  ldq   t0,-23520(t0)
120007a5c:  0e0020e4  beq   t0,120007a98
120007a60:  0204e147  mov   t0,t1
120007a64:  0304ff47  clr   t2
120007a68:  0500e0c3  br    120007a80
```

$$\frac{Instructions}{Program}$$

How many of these?

How long is it take to execution each of these?

$$\frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

Speedup

- The relative performance between two machines, X and Y. Y is n times faster than X

$$n = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

- The speedup of Y over X

$$Speedup = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

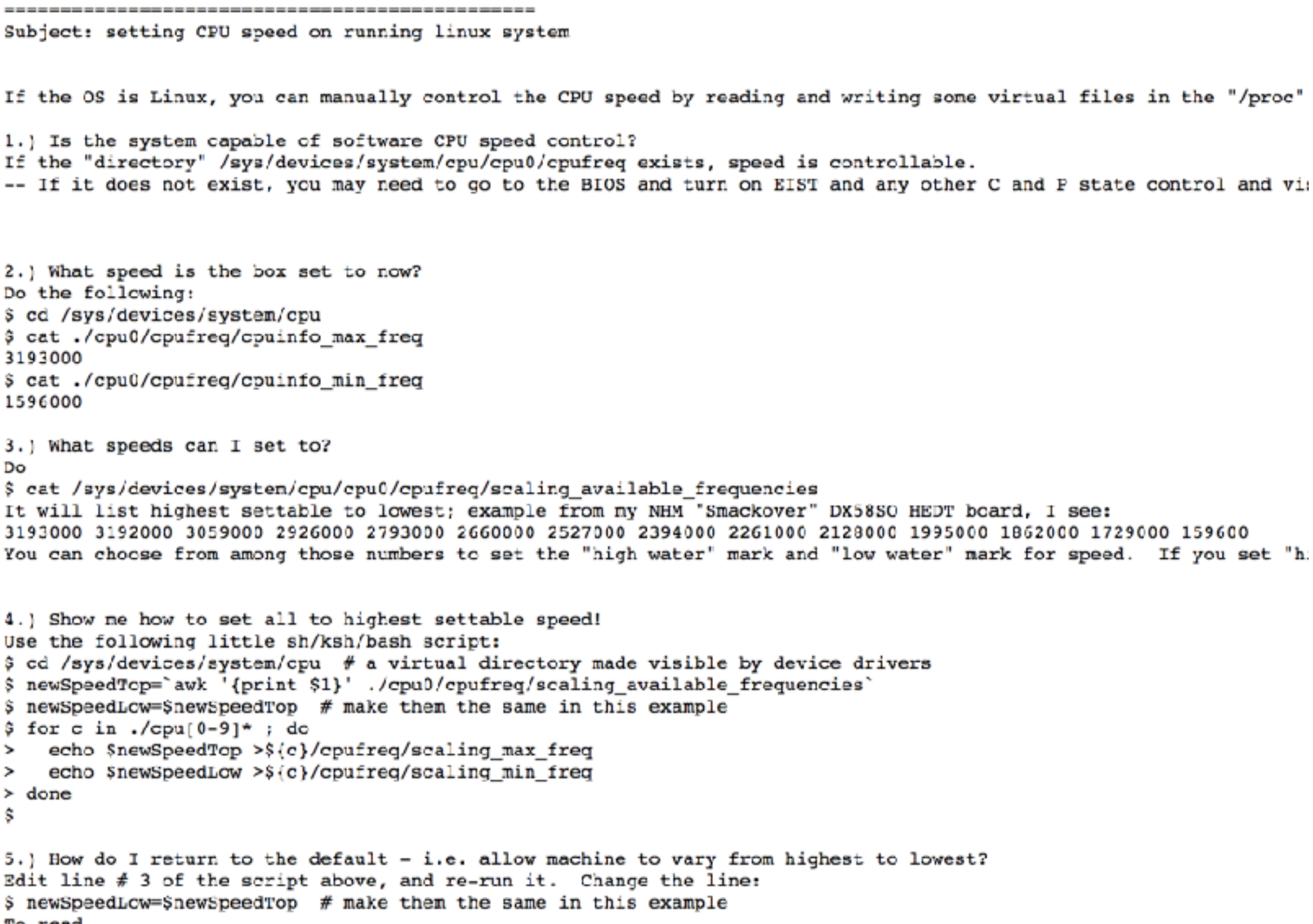
What Affects Each Factor in Performance Equation

Use “performance counters” to figure out!

- Modern processors provides performance counters
  - instruction counts
  - cache accesses/misses
  - branch instructions/mis-predictions
- How to get their values?
  - You may use “perf stat” in linux
  - You may use Instruments —> Time Profiler on a Mac
  - Intel’s vtune — only works on Windows w/ intel processors
  - You can also create your own functions to obtain counter values

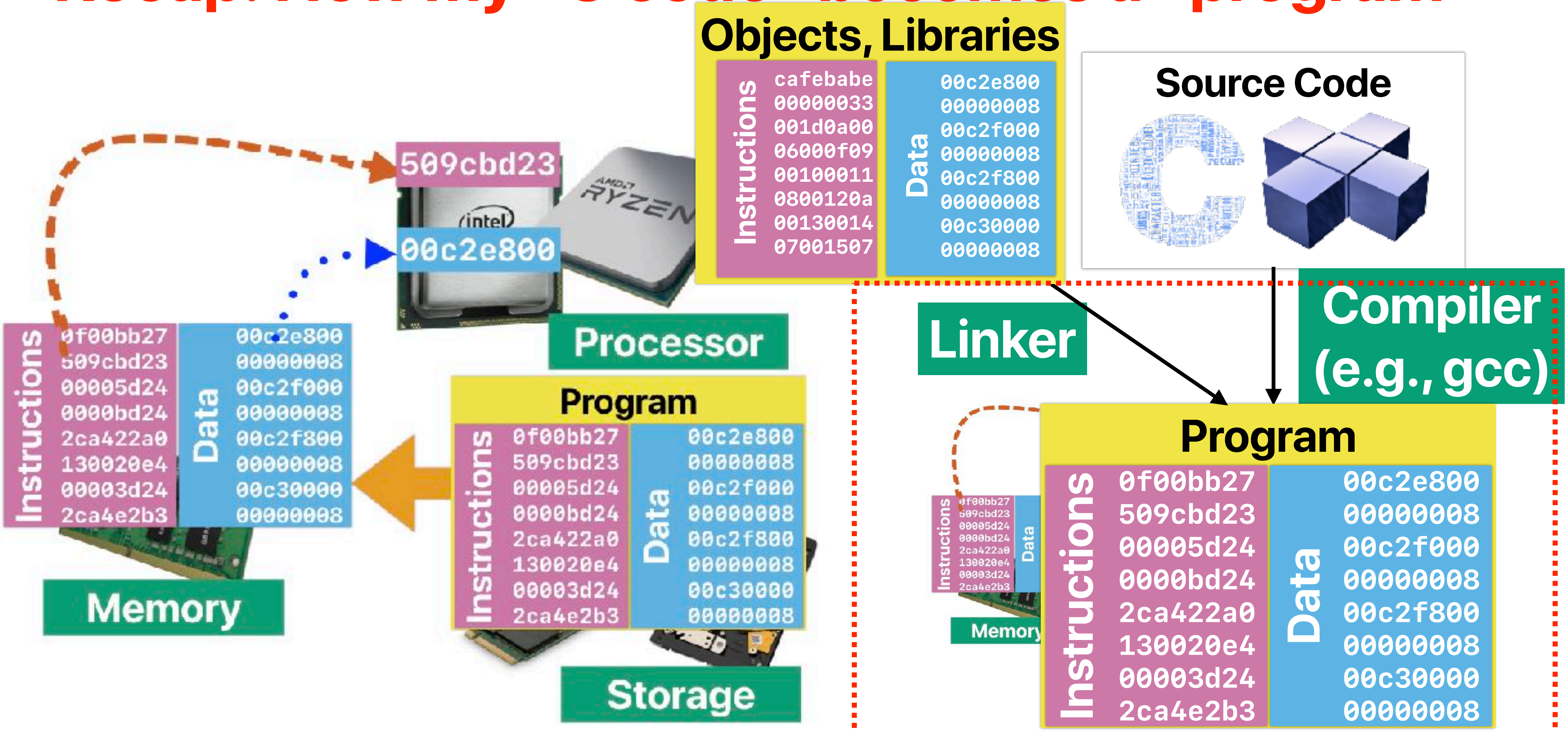
Programmers can also set the cycle time

https://software.intel.com/sites/default/files/comment/1716807/how-to-change-frequency-on-linux-pub.txt



```
===============================================
Subject: setting CPU speed on running linux system


If the OS is Linux, you can manually control the CPU speed by reading and writing some virtual files in the "/proc"

1.) Is the system capable of software CPU speed control?
If the "directory" /sys/devices/system/cpu/cpu0/cpufreq exists, speed is controllable.
-- If it does not exist, you may need to go to the BIOS and turn on EIST and any other C and P state control and vi



2.) What speed is the box set to now?
Do the following:
$ cd /sys/devices/system/cpu
$ cat ./cpu0/cpufreq/cpuinfo_max_freq
3193000
$ cat ./cpu0/cpufreq/cpuinfo_min_freq
1596000

3.) What speeds can I set to?
Do
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
It will list highest settable to lowest; example from my NHM "Smackover" DX58SO HEDT board, I see:
3193000 3192000 3059000 2926000 2793000 2660000 2527000 2394000 2261000 2128000 1995000 1862000 1729000 159600
You can choose from among those numbers to set the "high water" mark and "low water" mark for speed.  If you set "h


4.) Show me how to set all to highest settable speed!
Use the following little sh/ksh/bash script:
$ cd /sys/devices/system/cpu  # a virtual directory made visible by device drivers
$ newSpeedTop=`awk '{print $1}' ./cpu0/cpufreq/scaling_available_frequencies`
$ newSpeedLow=$newSpeedTop  # make them the same in this example
$ for c in ./cpu[0-9]* ; do
>   echo $newSpeedTop >${c}/cpufreq/scaling_max_freq
>   echo $newSpeedLow >${c}/cpufreq/scaling_min_freq
> done
$

5.) How do I return to the default - i.e. allow machine to vary from highest to lowest?
Edit line # 3 of the script above, and re-run it.  Change the line:
$ newSpeedLow=$newSpeedTop  # make them the same in this example
To read
```

Recap: How my "C code" becomes a "program"



Objects, Libraries

Instructions	Data
cafebabe	00c2e800
00000033	00000008
001d0a00	00c2f000
06000f09	00000008
00100011	00c2f800
0800120a	00000008
00130014	00c30000
07001507	00000008

Source Code

Compiler (e.g., gcc)

Linker

Program

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Storage

Memory

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

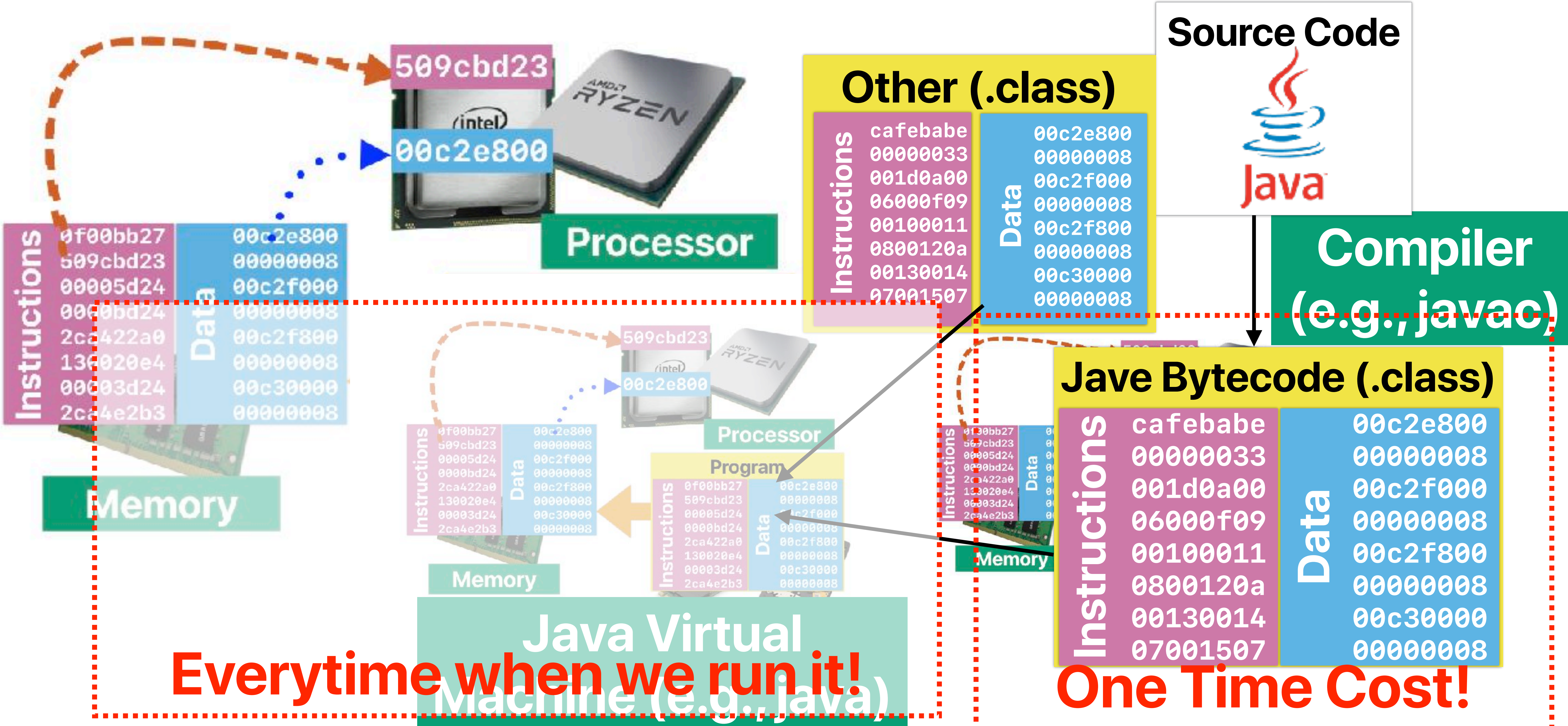
Processor

509cbd23

00c2e800

One Time Cost!

Recap: How my "Java code" becomes a "program"



Source Code

Java

Compiler (e.g., javac)

Other (.class)

Instructions	Data
cafebabe	00c2e800
00000033	00000008
001d0a00	00c2f000
06000f09	00000008
00100011	00c2f800
0800120a	00000008
00130014	00c30000
07001507	00000008

Jave Bytecode (.class)

Instructions	Data
cafebabe	00c2e800
00000033	00000008
001d0a00	00c2f000
06000f09	00000008
00100011	00c2f800
0800120a	00000008
00130014	00c30000
07001507	00000008

Processor

509cbd23

00c2e800

Memory

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

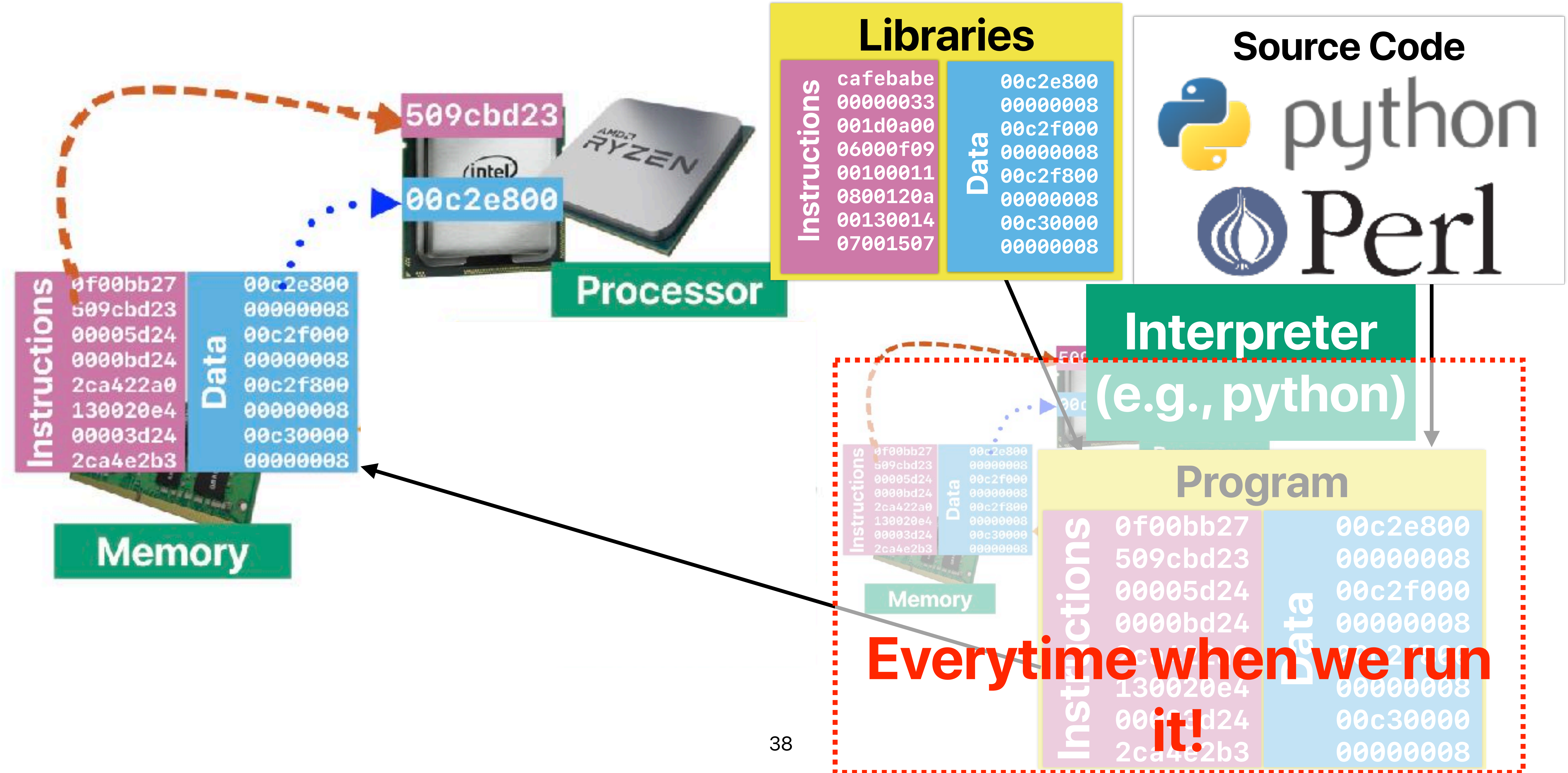
Program

Java Virtual Machine (e.g., java)

Everytime when we run it!

One Time Cost!

Recap: How my “Python code” becomes a “program”



Libraries

Instructions

```
cafebabe
00000033
001d0a00
06000f09
00100011
0800120a
00130014
07001507
```

Data

```
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008
```

Source Code

Processor

Memory

Instructions

```
0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3
```

Data

```
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008
```

Interpreter

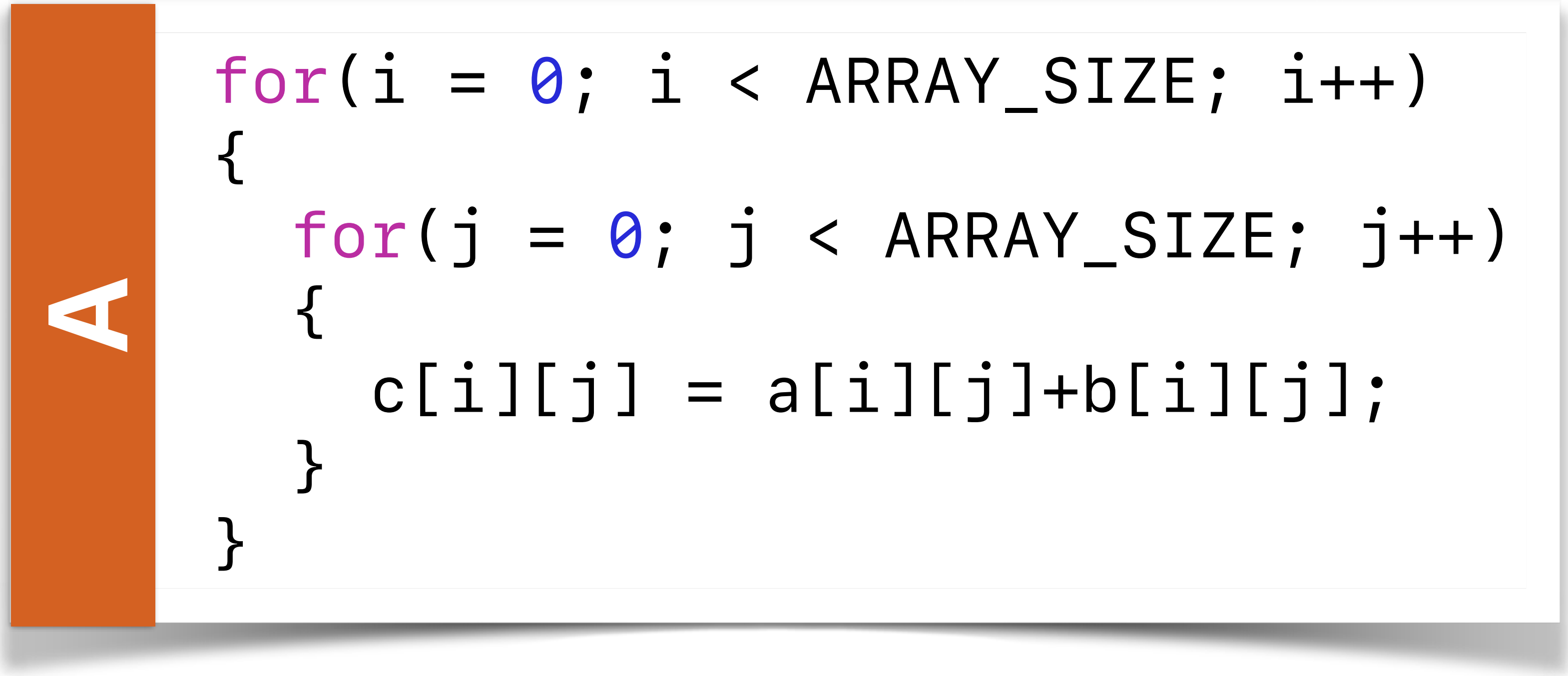
(e.g., python)

Program

Everytime when we run it!

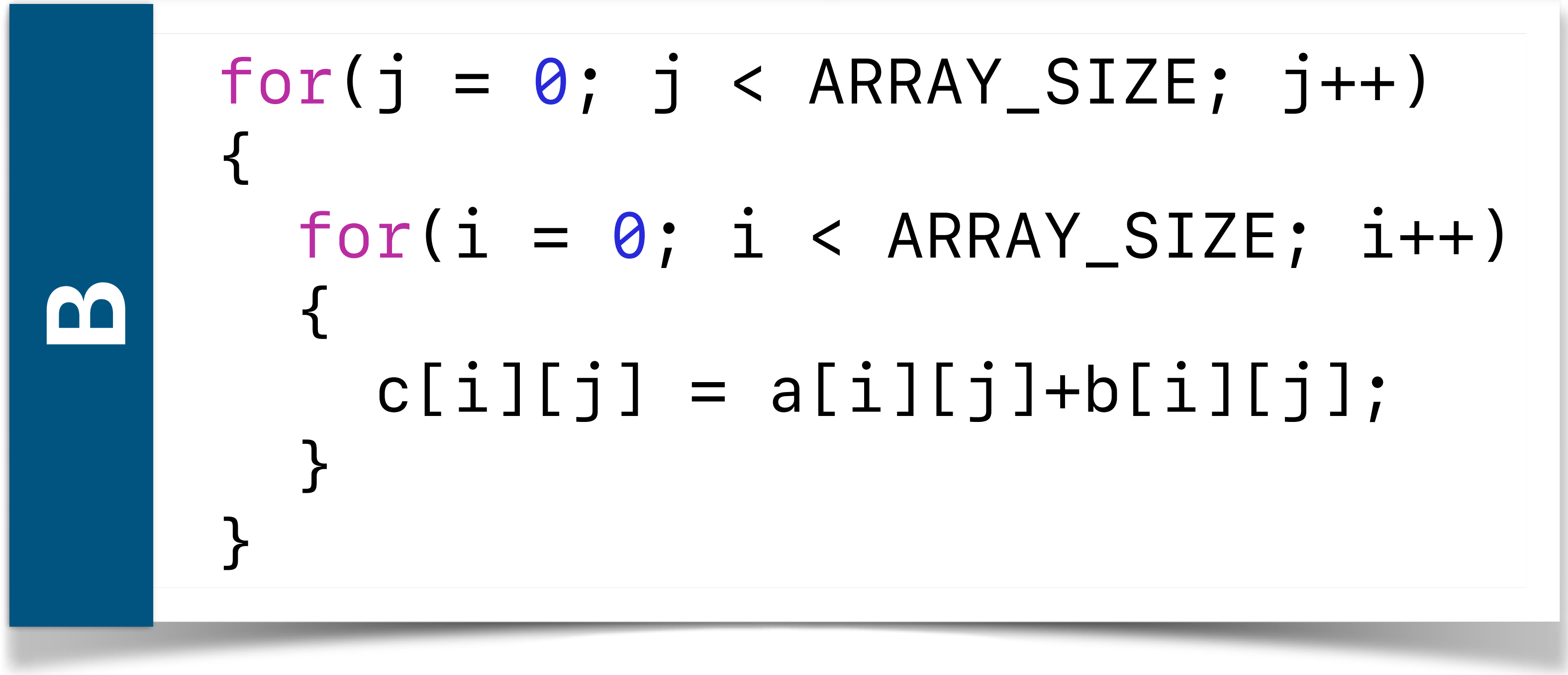
Revisited the demo with compiler optimizations!

- gcc has different optimization levels.
  - -O0 — no optimizations
  - -O3 — typically the best-performing optimization



A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```



B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Demo revisited — compiler optimization

- Compiler can reduce the instruction count, change CPI — with "limited scope"
- Compiler CANNOT help improving "crummy" source code

```
if(option)
    std::sort(data, data + arraySize);
```

Compiler can never add this — only the programmer can!

```
for (unsigned c = 0; c < arraySize*1000; ++c) {
        if (data[c%arraySize] >= INT_MAX/2)
            sum ++;
    }
}
```

How about “computational complexity”

- Algorithm complexity provides a good estimate on the performance if —
  - Every instruction takes exactly the same amount of time
  - Every operation takes exactly the same amount of instructions

These are unlikely to be true

Summary of CPU Performance Equation

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

$$ET = IC \times CPI \times CT$$

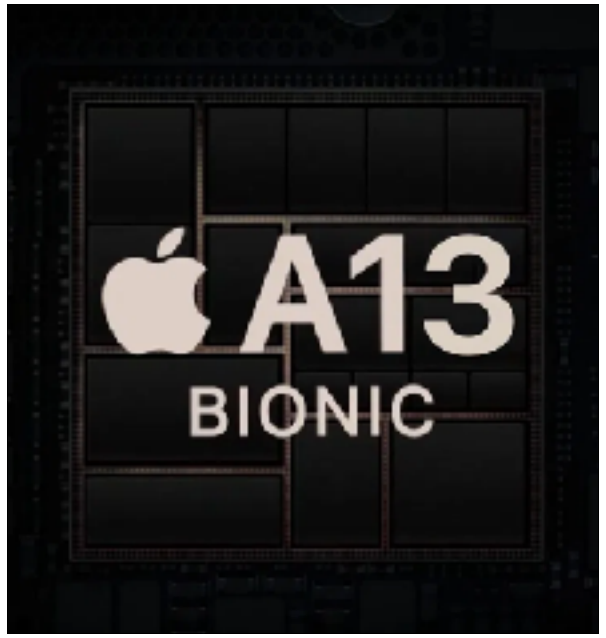
- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, **programmer**
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, **programmer**
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture, **programmer**

Instruction Set Architecture (ISA) & Performance

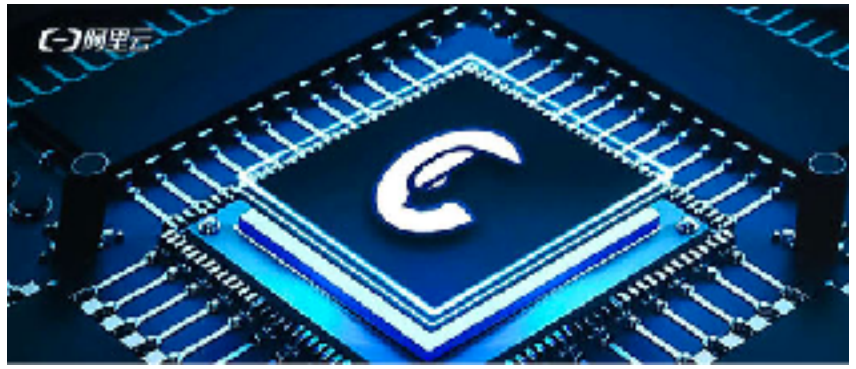
Recap: ISA — the interface b/w processor/software

- Operations
  - Arithmetic/Logical, memory access, control-flow (e.g., branch, function calls)
  - Operands
    - Types of operands — register, constant, memory addresses
    - Sizes of operands — byte, 16-bit, 32-bit, 64-bit
- Memory space
  - The size of memory that programs can use
  - The addressing of each memory locations
  - The modes to represent those addresses

Popular ISAs

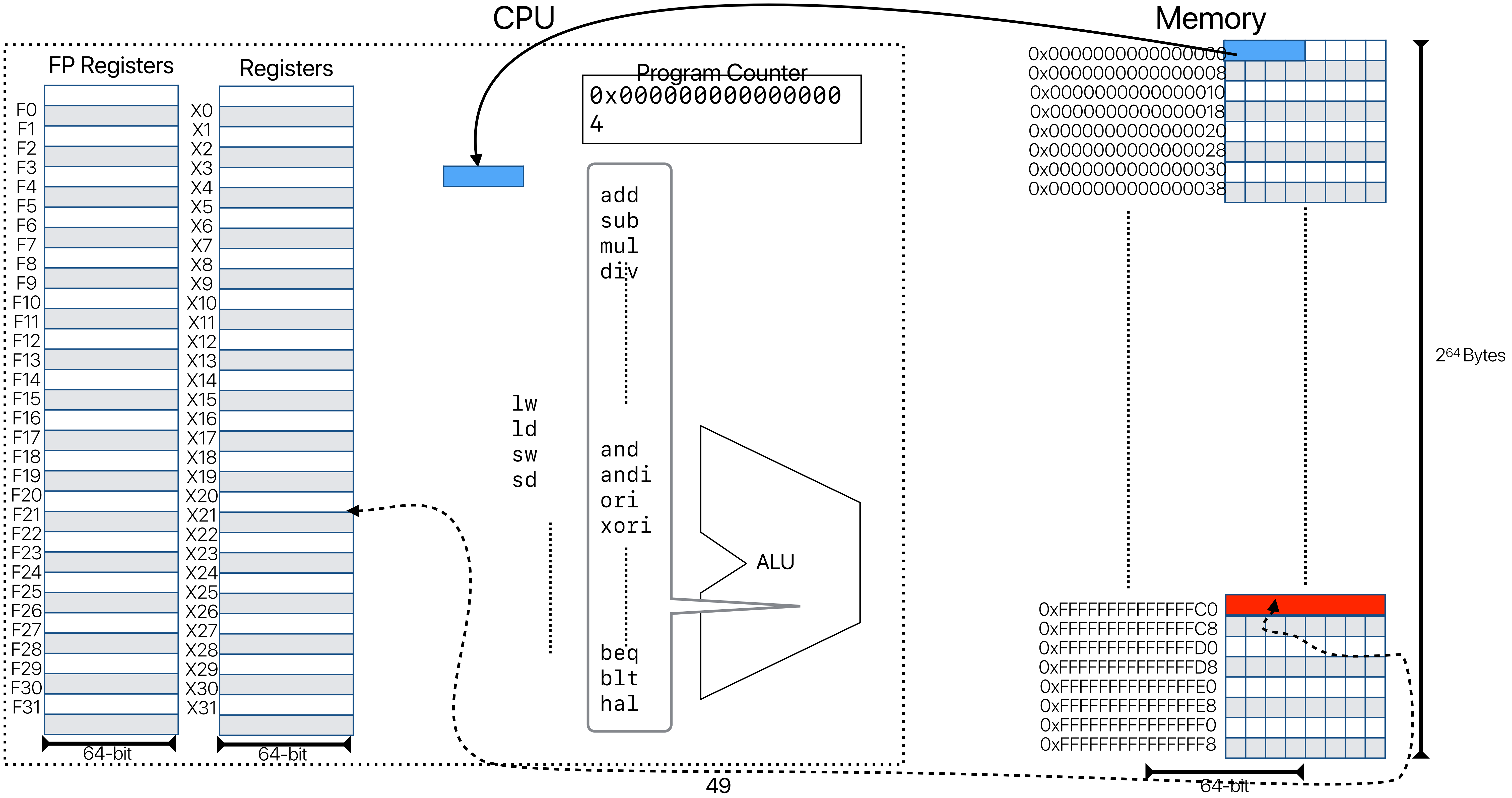


arm



RISC-V

The abstracted “RISC-V” machine



Subset of RISC-V instructions

Category	Instruction	Usage	Meaning
Arithmetic	add	add x1, x2, x3	x1 = x2 + x3
	addi	addi x1,x2, 20	x1 = x2 + 20
	sub	sub x1, x2, x3	x1 = x2 − x3
Logical	and	and x1, x2, x3	x1 = x2 & x3
	or	or x1, x2, x3	x1 = x2 | x3
	andi	andi x1, x2, 20	x1 = x2 & 20
	sll	sll x1, x2, 10	x1 = x2 ∗ 2^10
	srl	srl x1, x2, 10	x1 = x2 / 2^10
Data Transfer	ld	ld x1, 8(x2)	x1 = mem[x2+8]
	sd	sd x1, 8(x2)	mem[x2+8] = x1
Branch	beq	beq x1, x2, **25**	if(x1 == x2), PC = PC + **100**
	bne	bne x1, x2, **25**	if(x1 != x2), PC = PC + **100**
Jump	jal	jal **25**	$ra = PC **+ 4**, PC = 100
	jr	jr $ra	PC = $ra

The only type of instructions can access memory

Popular ISAs

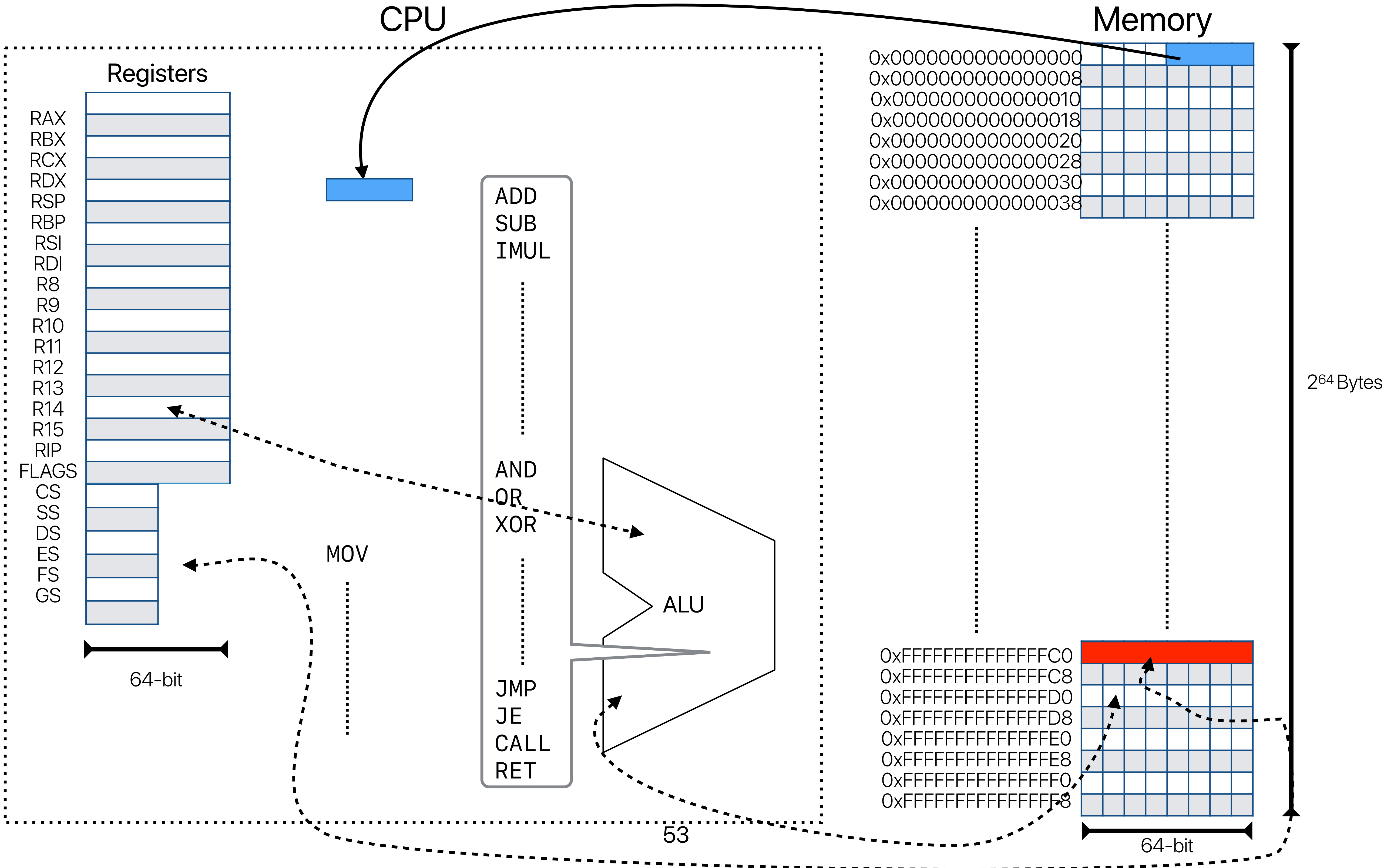
Complex Instruction Set Computers (CISC)

Reduced Instruction Set Computers (RISC)

How many operations: CISC v.s. RISC

- CISC (Complex Instruction Set Computing)
  - Examples: x86, Motorola 68K
  - Provide **many** **powerful/complex** instructions
    - Many: more than 1503 instructions since 2016
    - Powerful/complex: an instruction can perform both ALU and memory operations
    - Each instruction takes more cycles to execute
- RISC (Reduced Instruction Set Computer)
  - Examples: ARMv8, RISC-V, MIPS (the first RISC instruction, invented by the authors of our textbook)
  - Each instruction only performs simple tasks
  - Easy to decode
  - Each instruction takes less cycles to execute

The abstracted x86 machine



RISC-V v.s. x86

	RISC-V	x86
ISA type	Reduced Instruction Set Computers (RISC)	Complex Instruction Set Computers (CISC)
instruction width	32 bits	1 ~ 17 bytes
code size	larger	smaller
registers	32	16
addressing modes	reg+offset	base+offset base+index scaled+index scaled+index+offset
hardware	simple	complex

Amdahl’s Law — and It’s Implication in the Multicore Era

H&P Chapter 1.9

M. D. Hill and M. R. Marty. Amdahl’s Law in the Multicore Era. In Computer, vol. 41, no. 7, pp. 33-38, July 2008.

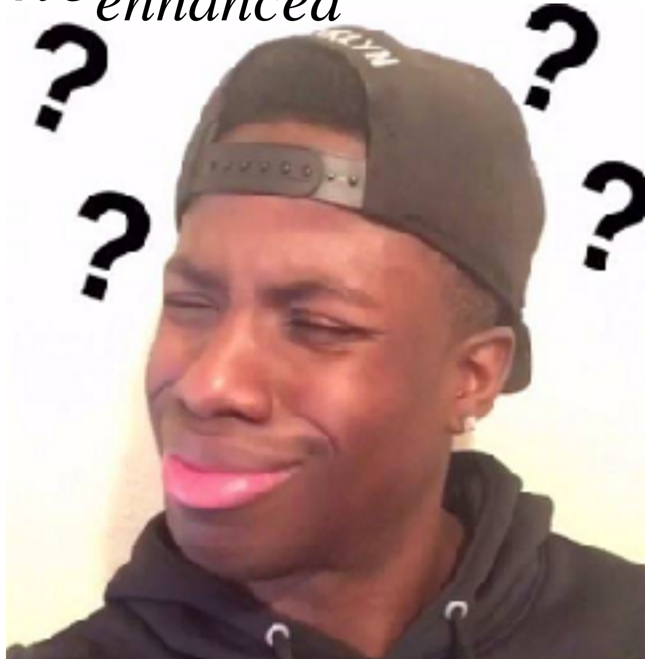
Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$$

f — The fraction of time in the original program

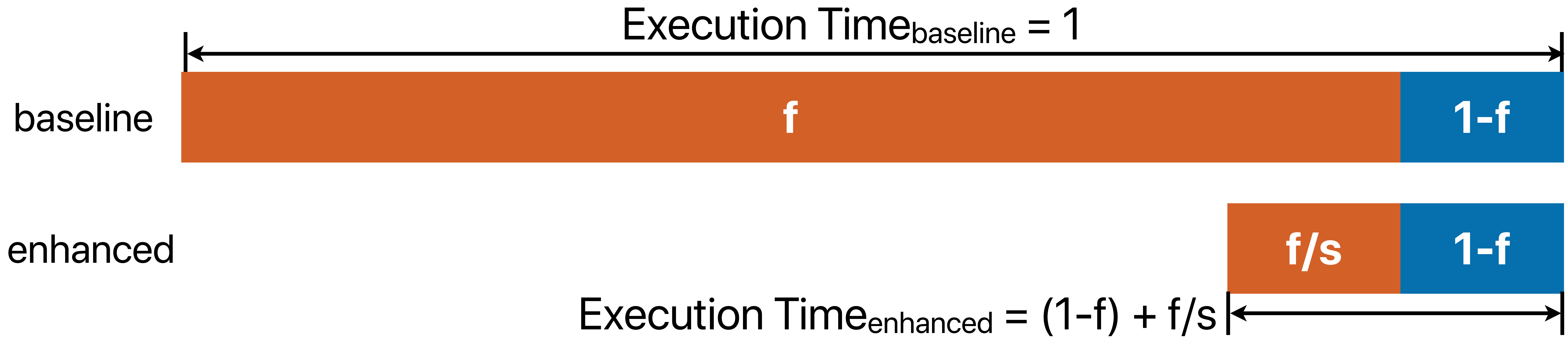
s — The speedup we can achieve on f

$$Speedup_{enhanced} = \frac{Execution\ Time_{baseline}}{Execution\ Time_{enhanced}}$$



Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$$


Execution Time$_{baseline}$ = 1
baseline
f
1-f
enhanced
f/s
1-f
Execution Time$_{enhanced}$ = (1-f) + f/s

$$Speedup_{enhanced} = \frac{Execution\ Time_{baseline}}{Execution\ Time_{enhanced}} = \frac{1}{(1-f) + \frac{f}{s}}$$

Replay using Amdahl's Law

- Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle when using a 2GHz processor.
  - If we double the CPU clock rate to 4GHz that helps to accelerate all instructions by 2x except that load/store instruction cannot be improved — their CPI will become 12 cycles. What's the performance improvement after this change?

How much time in load/store? $500000 \times (0.2 \times 6) \times 0.5\ ns = 300000\ ns \rightarrow 60\,\%$

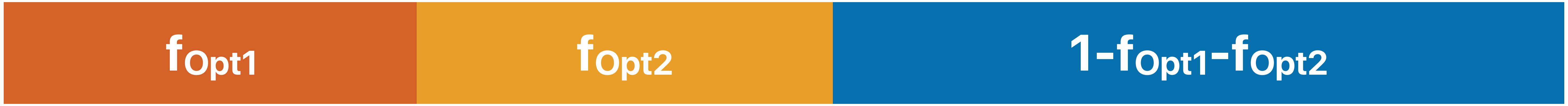
How much time in the rest? $500000 \times (0.8 \times 1) \times 0.5\ ns = 200000\ ns \rightarrow 40\,\%$

$$Speedup_{enhanced}(f, s) = \frac{1}{(1 - f) + \frac{f}{s}}$$

$$Speedup_{enhanced}(40\,\%, 2) = \frac{1}{(1 - 40\%) + \frac{40\,\%}{2}} = 1.25 \times$$

Amdahl's Law on Multiple Optimizations

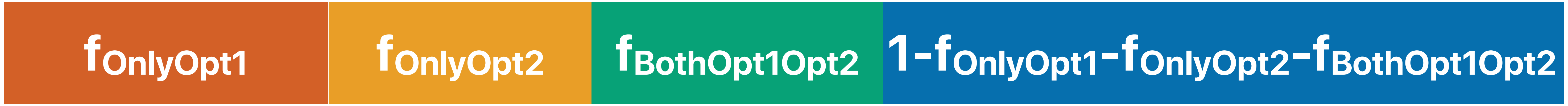
- We can apply Amdahl's law for multiple optimizations
- These optimizations must be dis-joint!
  - If optimization #1 and optimization #2 are dis-joint:



f_{Opt1} | f_{Opt2} | $1-f_{Opt1}-f_{Opt2}$

$$Speedup_{enhanced}(f_{Opt1}, f_{Opt2}, s_{Opt1}, s_{Opt2}) = \frac{1}{(1-f_{Opt1}-f_{Opt2}) + \frac{f\_Opt1}{s\_Opt1} + \frac{f\_Opt2}{s\_Opt2}}$$

  - If optimization #1 and optimization #2 are not dis-joint:



$f_{OnlyOpt1}$ | $f_{OnlyOpt2}$ | $f_{BothOpt1Opt2}$ | $1-f_{OnlyOpt1}-f_{OnlyOpt2}-f_{BothOpt1Opt2}$

$$Speedup_{enhanced}(f_{OnlyOpt1}, f_{OnlyOpt2}, f_{BothOpt1Opt2}, s_{OnlyOpt1}, s_{OnlyOpt2}, s_{BothOpt1Opt2})$$
$$= \frac{1}{(1-f_{OnlyOpt1}-f_{OnlyOpt2}-f_{BothOpt1Opt2}) + + \frac{f\_BothOpt1Opt2}{s\_BothOpt1Opt2} + \frac{f\_OnlyOpt1}{s\_OnlyOpt1} + \frac{f\_OnlyOpt2}{s\_OnlyOpt2}}$$

Amdahl's Law Corollary #1

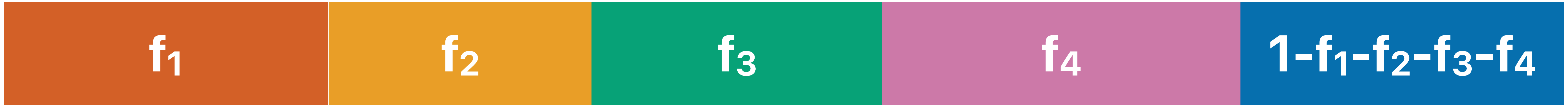
- The maximum speedup is bounded by

$$Speedup_{max}(f, \infty) = \frac{1}{(1-f) + \frac{f}{\infty}}$$

$$Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$$

Corollary #1 on Multiple Optimizations

- If we can pick just one thing to work on/optimize



f_1 f_2 f_3 f_4 $1-f_1-f_2-f_3-f_4$

$$Speedup_{max}(f_1, \infty) = \frac{1}{(1 - f_1)}$$

$$Speedup_{max}(f_2, \infty) = \frac{1}{(1 - f_2)}$$

$$Speedup_{max}(f_3, \infty) = \frac{1}{(1 - f_3)}$$

$$Speedup_{max}(f_4, \infty) = \frac{1}{(1 - f_4)}$$

The biggest f_x would lead to the largest $Speedup_{max}$!

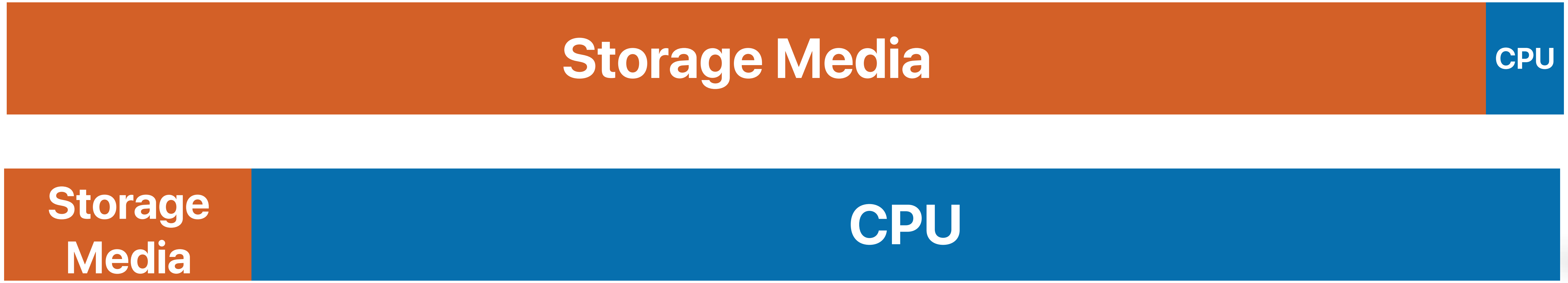
Corollary #2 — make the common case fast!

- When f is small, optimizations will have little effect.
- Common == **most time consuming** not necessarily the most frequent
- The uncommon case doesn’t make much difference
- The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.

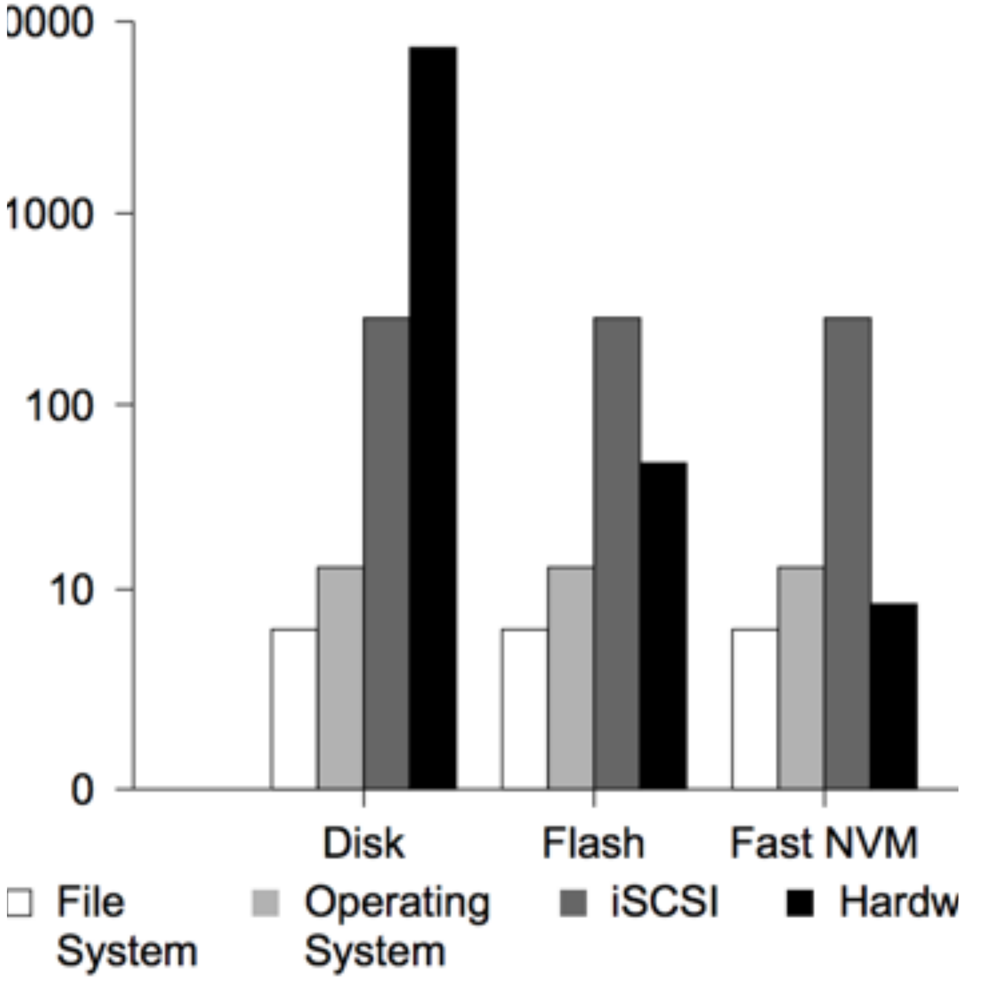
Identify the most time consuming part

- Compile your program with -pg flag
- Run the program
    - It will generate a gmon.out
    - gprof your_program gmon.out > your_program.prof
- It will give you the profiled result in your_program.prof

If we repeatedly optimizing our design based on Amdahl's law…



- With optimization, the common becomes uncommon.
- An uncommon case will (hopefully) become the new common case.
- Now you have a new target for optimization.
- — You have to revisit "Amdahl's Law" every time you applied some optimization



Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 2010.

Don’t hurt non-common part too mach

- If the program spend 90% in A, 10% in B. Assume that an optimization can accelerate A by 9x, by hurts B by 10x...
- Assume the original execution time is T. The new execution time

$$ET_{new} = \frac{ET_{old} \times 90\,\%}{9} + ET_{old} \times 10\% \times 10$$

$$ET_{new} = 1.1 \times ET_{old}$$

$$Speedup = \frac{ET_{old}}{ET_{new}} = \frac{ET_{old}}{1.1 \times ET_{old}} = 0.91 \times$$ ......slowdown!

You may not use Amdahl’s Law for this case as Amdahl’s Law does NOT
(1) consider overhead
(2) bound to slowdown

Amdahl's Law on Multicore Architectures

- Symmetric multicore processor with n cores (if we assume the processor performance scales perfectly)

$$Speedup_{parallel}(f_{parallelizable}, n) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{n}}$$

Corollary #3, Corollary #4 & Corollary #5

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{\infty}}$$

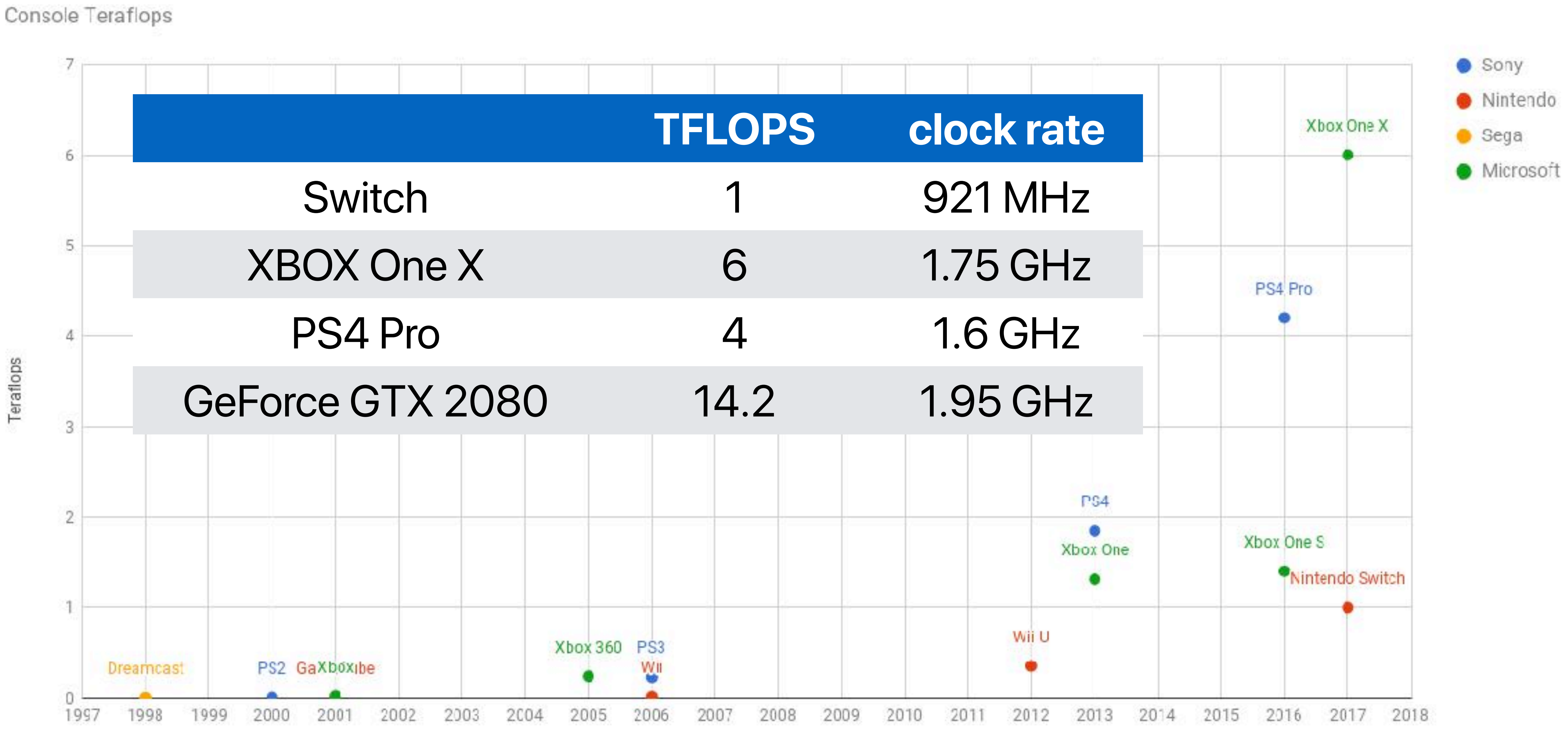
$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

- Single-core performance still matters — it will eventually dominate the performance
- Finding more “parallelizable” parts is also important
- If we can build a processor with unlimited parallelism — the complexity doesn’t matter as long as the algorithm can utilize all parallelism — that’s why bitonic sort works!

“Fair” Comparisons

Andrew Davison. Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. In Humour the Computer, MITP, 1995
V. Sze, Y. -H. Chen, T. -J. Yang and J. S. Emer. How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful. In IEEE Solid-State Circuits Magazine, vol. 12, no. 3, pp. 28-41, Summer 2020.

TFLOPS (Tera FLoating-point Operations Per Second)



	TFLOPS	clock rate
Switch	1	921 MHz
XBOX One X	6	1.75 GHz
PS4 Pro	4	1.6 GHz
GeForce GTX 2080	14.2	1.95 GHz

Is TFLOPS (Tera FLoating-point Operations Per Second) a good metric?

$$TFLOPS = \frac{\#\ of\ floating\ point\ instructions \times 10^{-12}}{Exection\ Time}$$

$$= \frac{IC \times \%\ of\ floating\ point\ instructions \times 10^{-12}}{IC \times CPI \times CT}$$

$$= \frac{\%\ of\ floating\ point\ instructions \times 10^{-12}}{CPI \times CT}$$

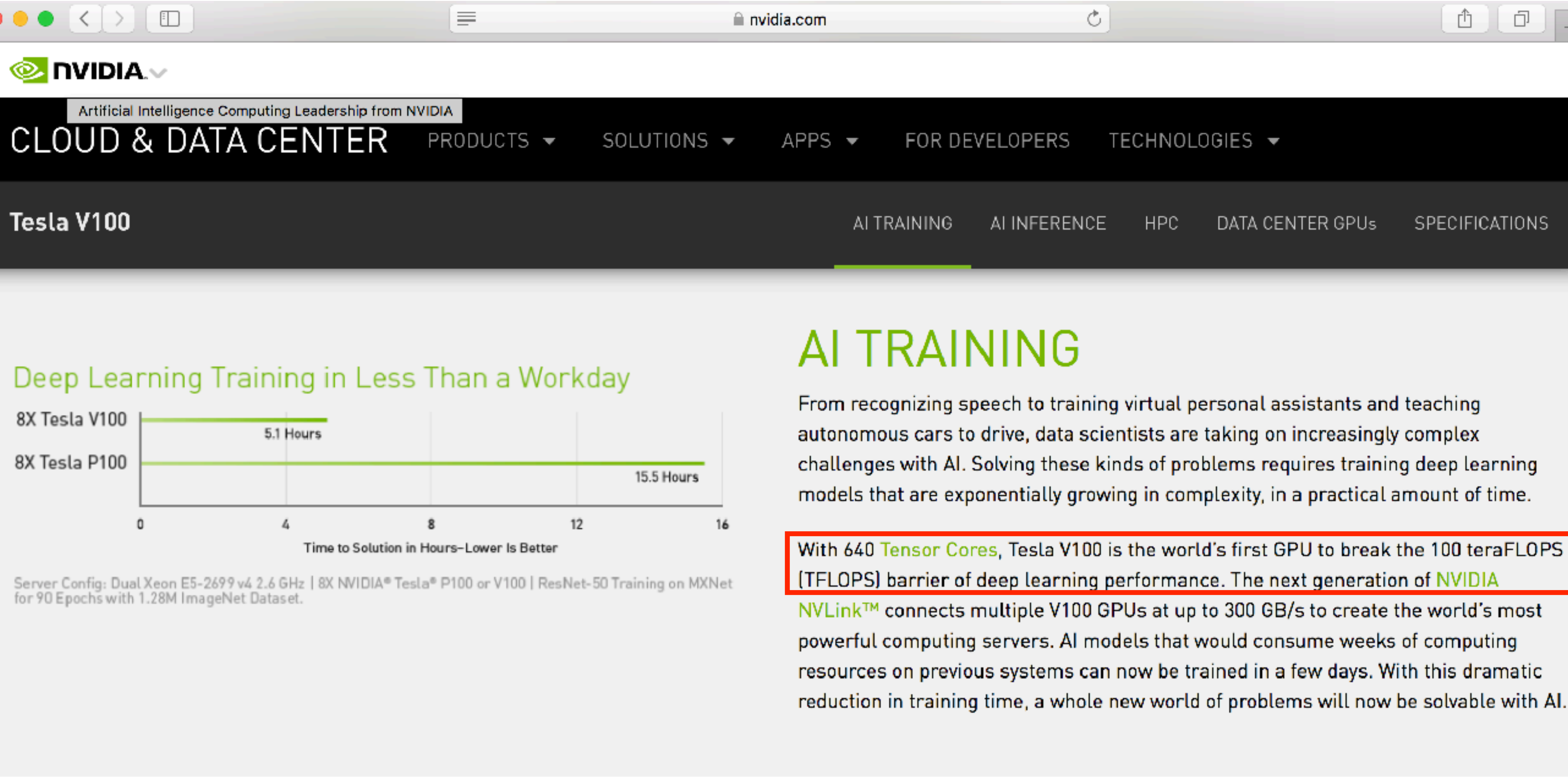
IC is gone!

- Cannot compare different ISA/compiler
  - What if the compiler can generate code with fewer instructions?
  - What if new architecture has more IC but also lower CPI?
- Does not make sense if the application is not floating point intensive

TFLOPS (Tera FLoating-point Operations Per Second)

- Cannot compare different ISA/compiler
  - What if the compiler can generate code with fewer instructions?
  - What if new architecture has more IC but also lower CPI?
- Does not make sense if the application is not floating point intensive

	TFLOPS	clock rate
Switch	1	921 MHz
XBOX One X	6	1.75 GHz
PS4 Pro	4	1.6 GHz
GeForce GTX 2080	14.2	1.95 GHz



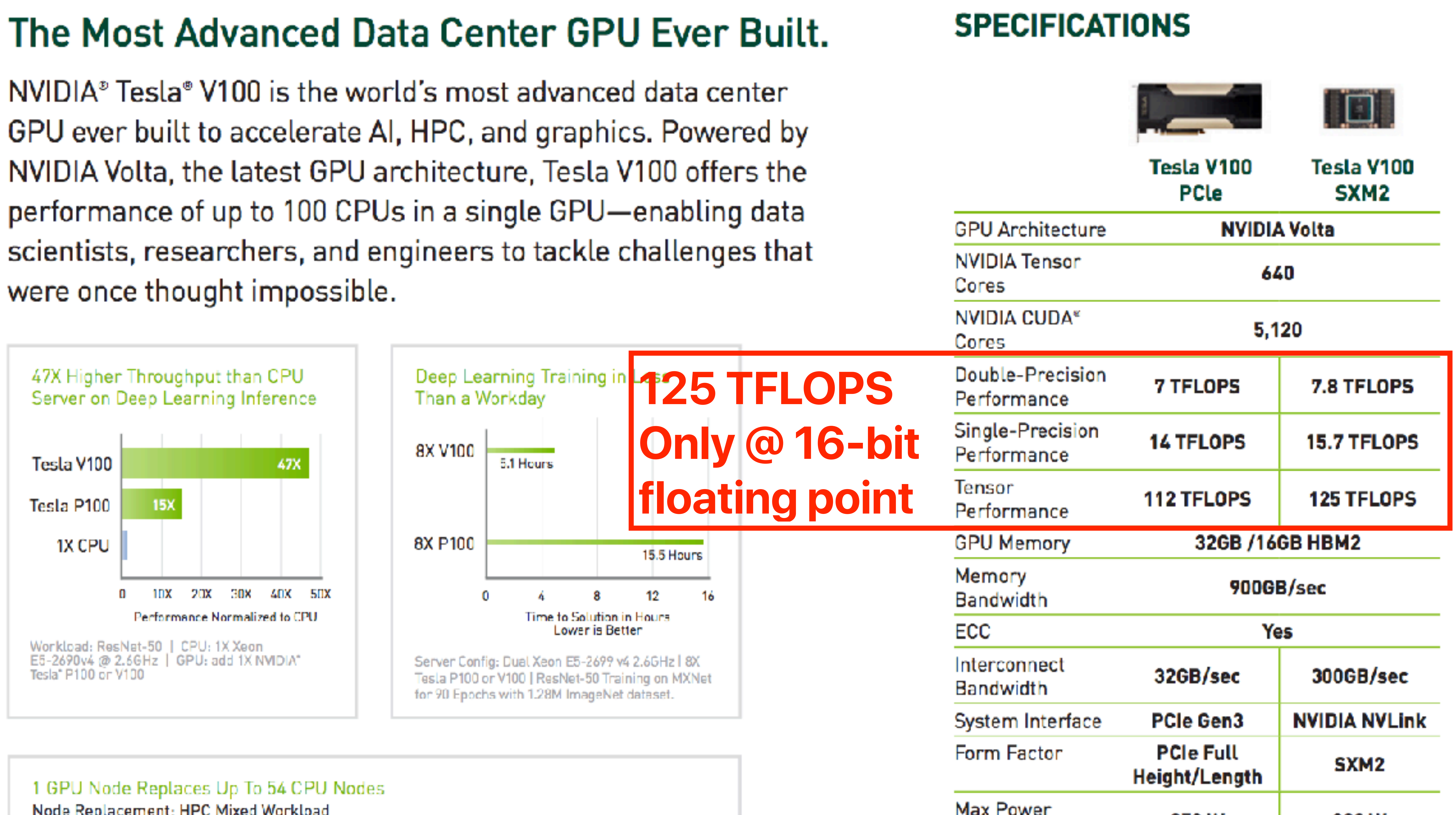
Deep Learning Training in Less Than a Workday

Server Config: Dual Xeon E5-2699 v4 2.6 GHz | 8X NVIDIA® Tesla® P100 or V100 | ResNet-50 Training on MXNet for 90 Epochs with 1.28M ImageNet Dataset.

AI TRAINING

From recognizing speech to training virtual personal assistants and teaching autonomous cars to drive, data scientists are taking on increasingly complex challenges with AI. Solving these kinds of problems requires training deep learning models that are exponentially growing in complexity, in a practical amount of time.

With 640 Tensor Cores, Tesla V100 is the world's first GPU to break the 100 teraFLOPS (TFLOPS) barrier of deep learning performance. The next generation of NVIDIA NVLink™ connects multiple V100 GPUs at up to 300 GB/s to create the world's most powerful computing servers. AI models that would consume weeks of computing resources on previous systems can now be trained in a few days. With this dramatic reduction in training time, a whole new world of problems will now be solvable with AI.



The Most Advanced Data Center GPU Ever Built.

NVIDIA® Tesla® V100 is the world's most advanced data center GPU ever built to accelerate AI, HPC, and graphics. Powered by NVIDIA Volta, the latest GPU architecture, Tesla V100 offers the performance of up to 100 CPUs in a single GPU—enabling data scientists, researchers, and engineers to tackle challenges that were once thought impossible.

47X Higher Throughput than CPU Server on Deep Learning Inference

Workload: ResNet-50 | CPU: 1X Xeon E5-2690v4 @ 2.6GHz | GPU: add 1X NVIDIA® Tesla® P100 or V100

Deep Learning Training in Less Than a Workday

Server Config: Dual Xeon E5-2699 v4 2.6GHz | 8X Tesla P100 or V100 | ResNet-50 Training on MXNet for 90 Epochs with 1.28M ImageNet dataset.

1 GPU Node Replaces Up To 54 CPU Nodes

Node Replacement: HPC Mixed Workload

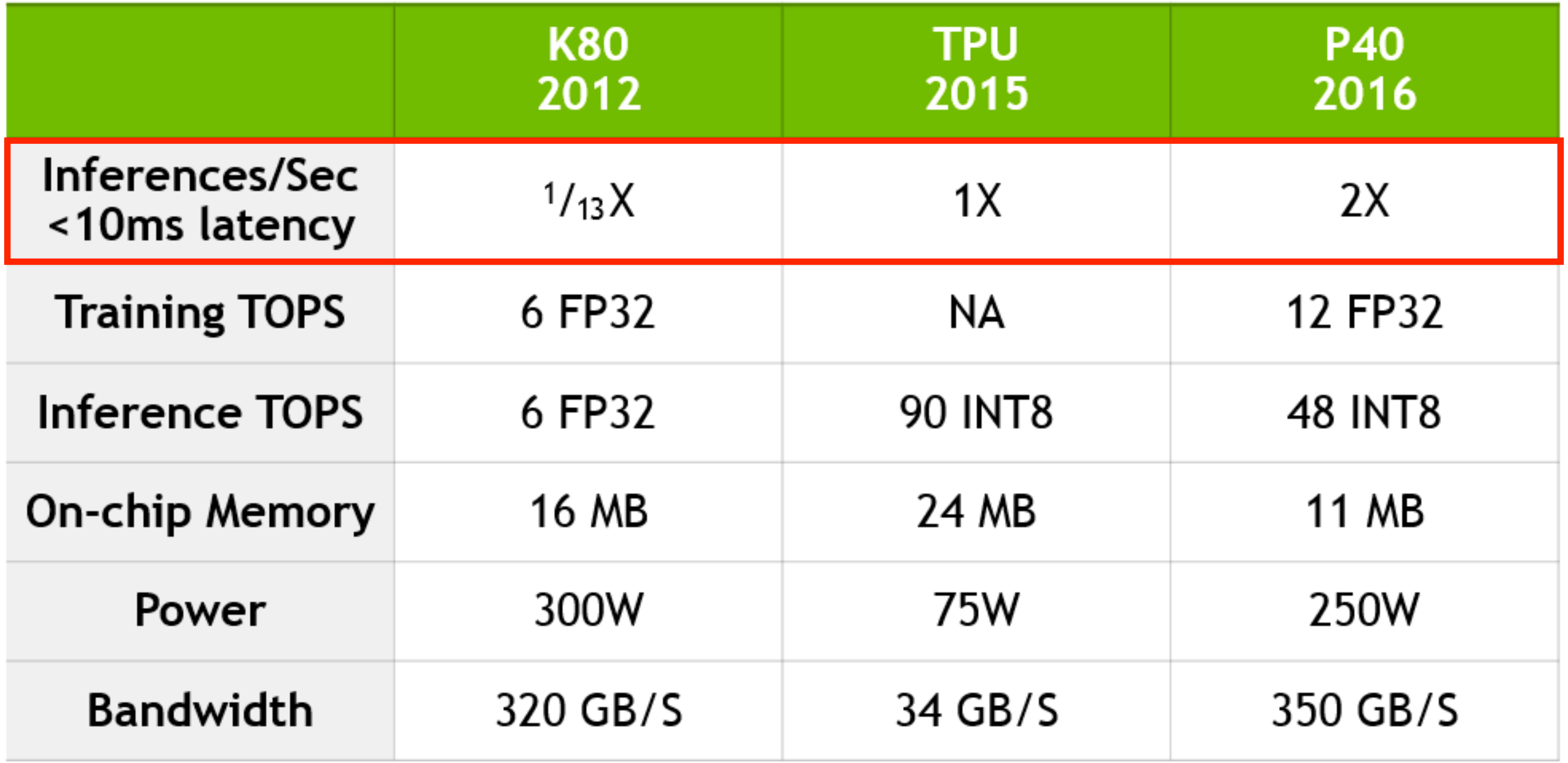
SPECIFICATIONS

125 TFLOPS
Only @ 16-bit
floating point

	Tesla V100 PCle	Tesla V100 SXM2
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
GPU Memory	32GB /16GB HBM2	
Memory Bandwidth	900GB/sec	
ECC	Yes	
Interconnect Bandwidth	32GB/sec	300GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink
Form Factor	PCIe Full Height/Length	SXM2
Max Power		

They try to tell it's the better AI hardware

https://blogs.nvidia.com/blog/2017/04/10/ai-drives-rise-accelerated-computing-datacenter/



	K80 2012	TPU 2015	P40 2016
Inferences/Sec <10ms latency	$^{1}/_{13}$X	1X	2X
Training TOPS	6 FP32	NA	12 FP32
Inference TOPS	6 FP32	90 INT8	48 INT8
On-chip Memory	16 MB	24 MB	11 MB
Power	300W	75W	250W
Bandwidth	320 GB/S	34 GB/S	350 GB/S

Inference per second

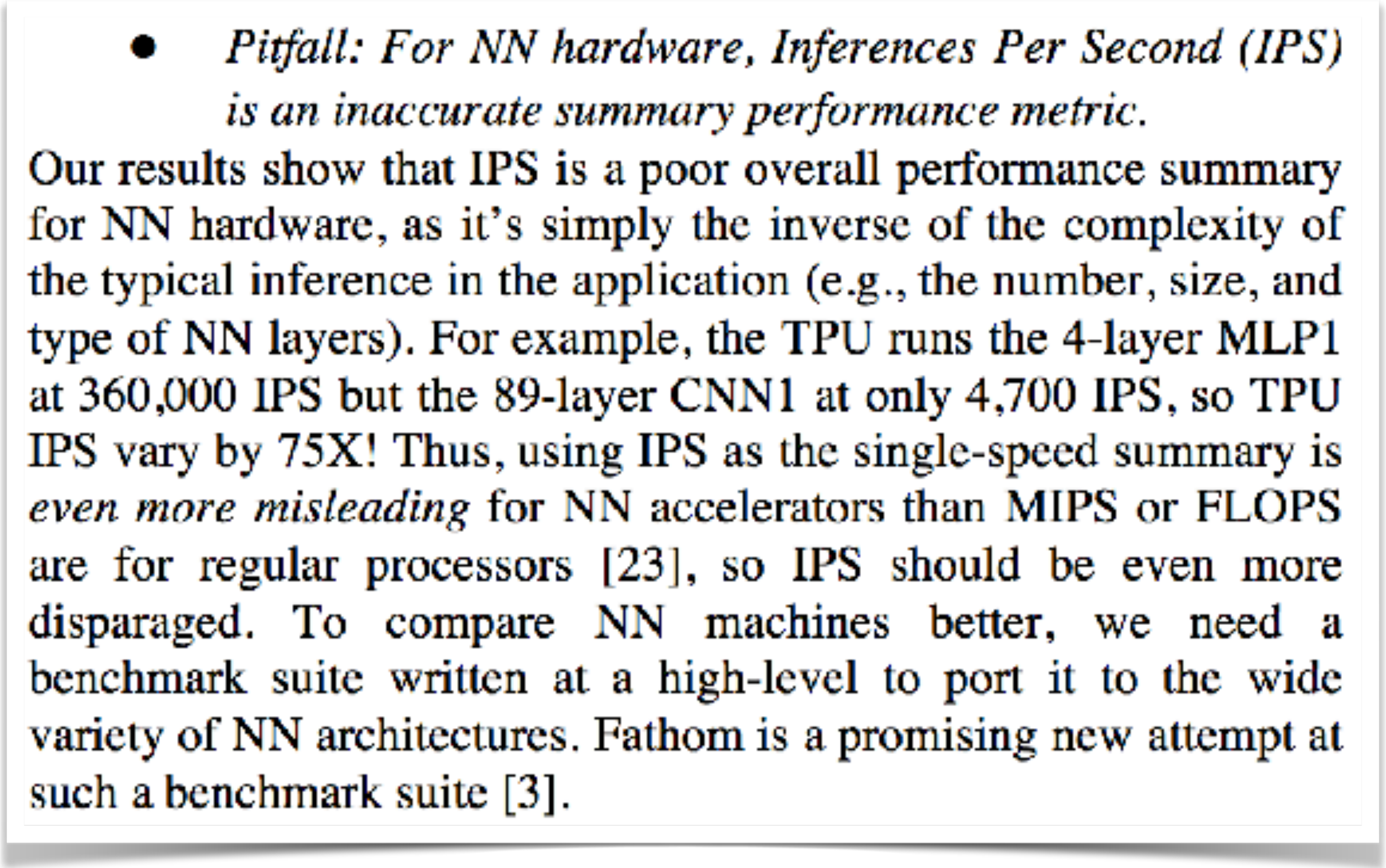
$$\frac{Inferences}{Second} = \frac{Inferences}{Operation} \times \frac{Operations}{Second}$$

$$= \frac{Inferences}{Operation} \times [\frac{operations}{cycle} \times \frac{cycles}{second} \times \#\_of\_PEs \times Utilization\_of\_PEs]$$

	Hardware	Model	Input Data
Operations per inference		V	
Operations per cycle	V		
Cycles per second	V		
Number of PEs	V		
Utilization of PEs	V	V	
Effectual operations out of (total) operations		V	V
Effectual operations plus unexploited ineffectual operations per cycle	V		

What’s wrong with inferences per second?

- There is no standard on how they inference — but these affect!
  - What model?
  - What dataset?
- That’s why Facebook is trying to promote an AI benchmark — MLPerf



- *Pitfall: For NN hardware, Inferences Per Second (IPS) is an inaccurate summary performance metric.*

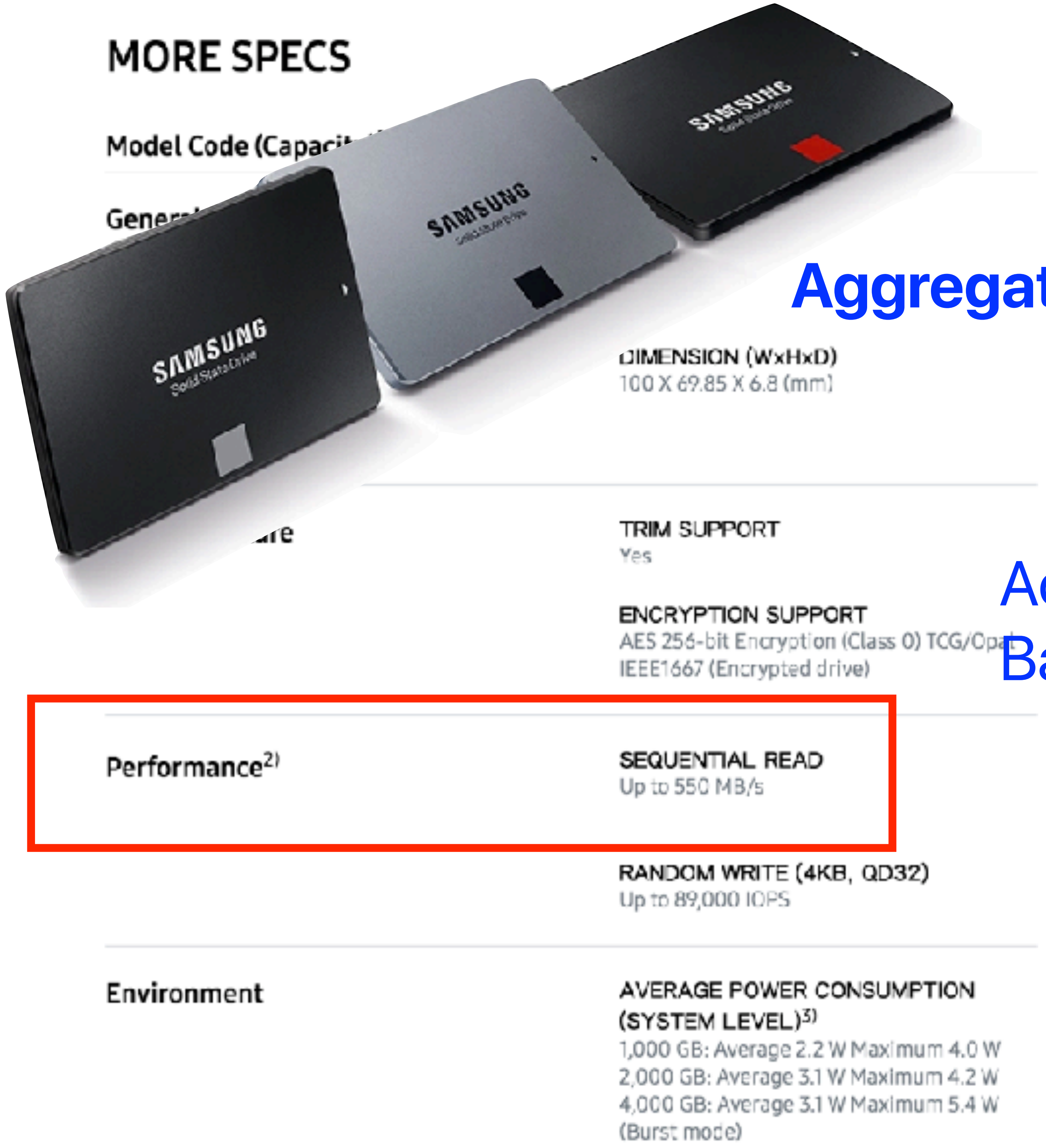
Our results show that IPS is a poor overall performance summary for NN hardware, as it’s simply the inverse of the complexity of the typical inference in the application (e.g., the number, size, and type of NN layers). For example, the TPU runs the 4-layer MLP1 at 360,000 IPS but the 89-layer CNN1 at only 4,700 IPS, so TPU IPS vary by 75X! Thus, using IPS as the single-speed summary is *even more misleading* for NN accelerators than MIPS or FLOPS are for regular processors [23], so IPS should be even more disparaged. To compare NN machines better, we need a benchmark suite written at a high-level to port it to the wide variety of NN architectures. Fathom is a promising new attempt at such a benchmark suite [3].

Choose the right metric — Latency v.s. Throughput/Bandwidth

Latency v.s. Bandwidth/Throughput

- Latency — the amount of time to finish an operation
    - Access time
    - Response time
- Throughput — the amount of work can be done within a given period of time
    - Bandwidth (MB/Sec, GB/Sec, Mbps, Gbps)
    - IOPs (I/O operations per second)
    - FLOPs (Floating-point operations per second)
    - IPS (Inferences per second)

RAID — Improving throughput



MORE SPECS

Model Code (Capacit

Gener

DIMENSION (WxHxD)
100 X 69.85 X 6.8 (mm)

are

TRIM SUPPORT
Yes

ENCRYPTION SUPPORT
AES 256-bit Encryption (Class 0) TCG/Opa
IEEE1667 (Encrypted drive)

Performance[2]

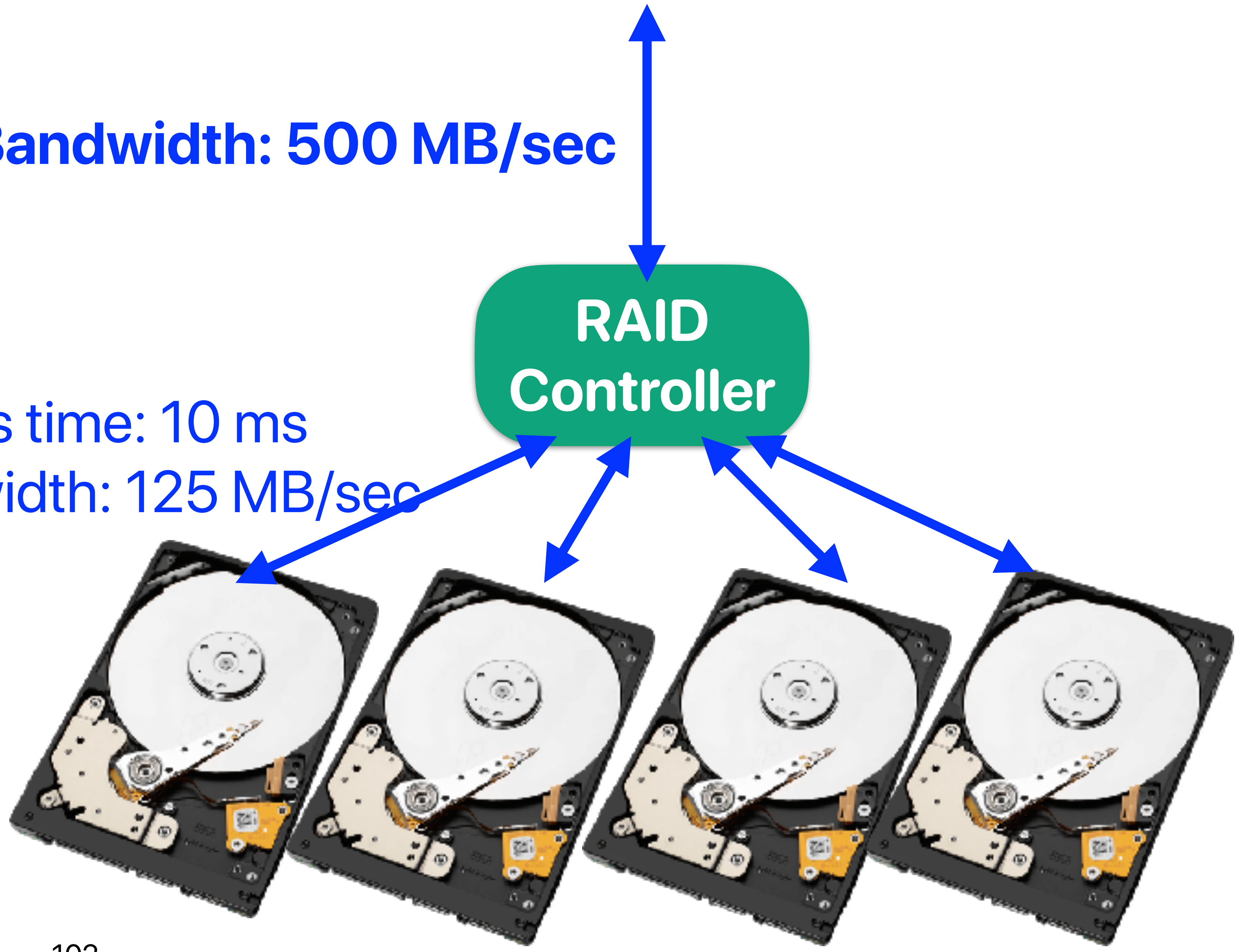
SEQUENTIAL READ
Up to 550 MB/s

RANDOM WRITE (4KB, QD32)
Up to 89,000 IOPS

Environment

AVERAGE POWER CONSUMPTION (SYSTEM LEVEL)[3]
1,000 GB: Average 2.2 W Maximum 4.0 W
2,000 GB: Average 3.1 W Maximum 4.2 W
4,000 GB: Average 3.1 W Maximum 5.4 W
(Burst mode)

Aggregated Bandwidth: 500 MB/sec



Access time: 10 ms
Bandwidth: 125 MB/sec

Latency/Delay v.s. Throughput

	Toyota Prius	100 Gb Network
	•100 miles (161 km) from UCSD •75 MPH on highway! •Max load: 374 kg = 2,770 hard drives (2TB per drive) 	•100 miles (161 km) from UCSD •Lightspeed! — $3*10^8$m/sec •Max load:4 lanes operating at 25GHz
bandwidth	290GB/sec	100 Gb/s or 12.5GB/sec
total latency	3.5 hours	2 Peta-byte over 167772 seconds = 1.94 Days
latency in getting the first moivie	You see nothing in the first 3.5 hours	100GB/100Gb = 8 secs! You can start watching the first movie in 8 secs!

Extreme Multitasking Performance

- Dual 4K external monitors
- 1080p device display
- 7 applications

Qualcomm Snapdragon is a product of Qualcomm Technologies, Inc. and/or its subsidiaries.

What’s missing in this video clip?

- The ISA of the “competitor”
- Clock rate, CPU architecture, cache size, how many cores
- How big the RAM?
- How fast the disk?

12 ways to Fool the Masses When Giving Performance Results on Parallel Computers

- Quote only 32-bit performance results, not 64-bit results.
- Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
- Quietly employ assembly code and other low-level language constructs.
- Scale up the problem size with the number of processors, but omit any mention of this fact.
- Quote performance results projected to a full system.
- Compare your results against scalar, unoptimized code on Crays.
- When direct run time comparisons are required, compare with an old code on an obsolete system.
- If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.
- Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.
- Mutilate the algorithm used in the parallel implementation to match the architecture.
- Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment.
- If all else fails, show pretty pictures and animated videos, and don't talk about performance.