# Performance Evaluation

# CPU Performance Equation

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

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

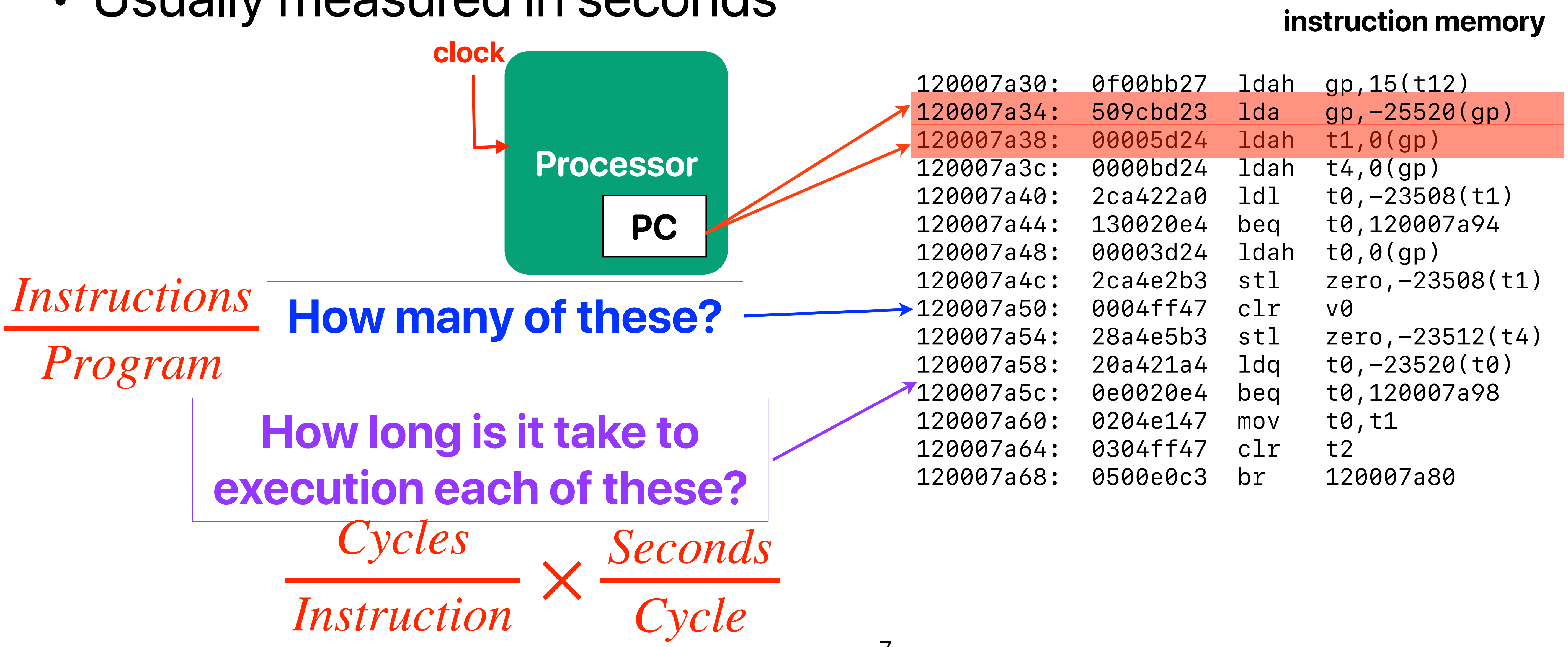
$$ET = IC \times CPI \times CT$$

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

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

# 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
```

How many of these?

How long is it take to execution each of these?

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

# Speedup

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

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

- The speedup of X over Y

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

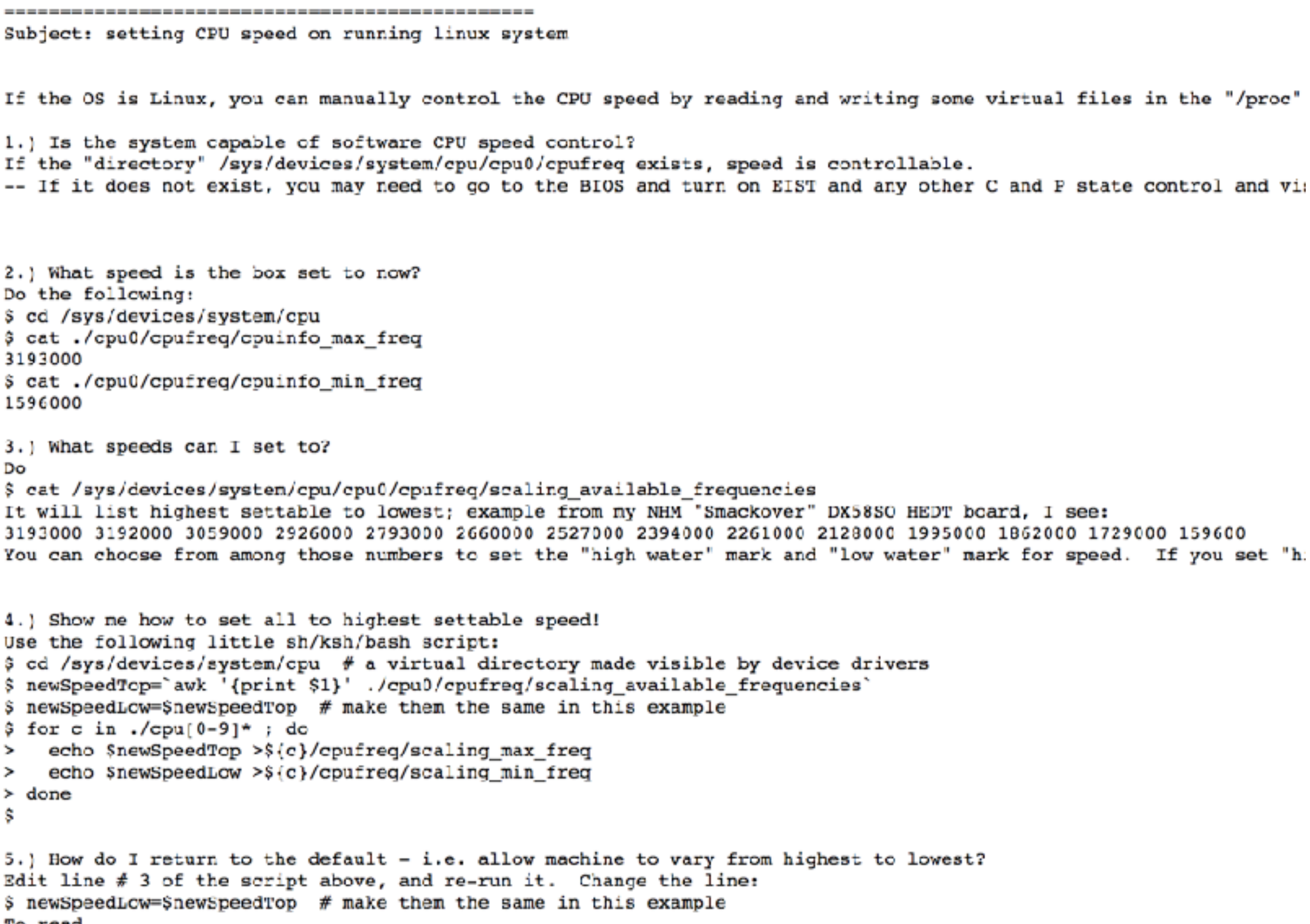
# What Affects Each Factor in Performance Equation

# 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

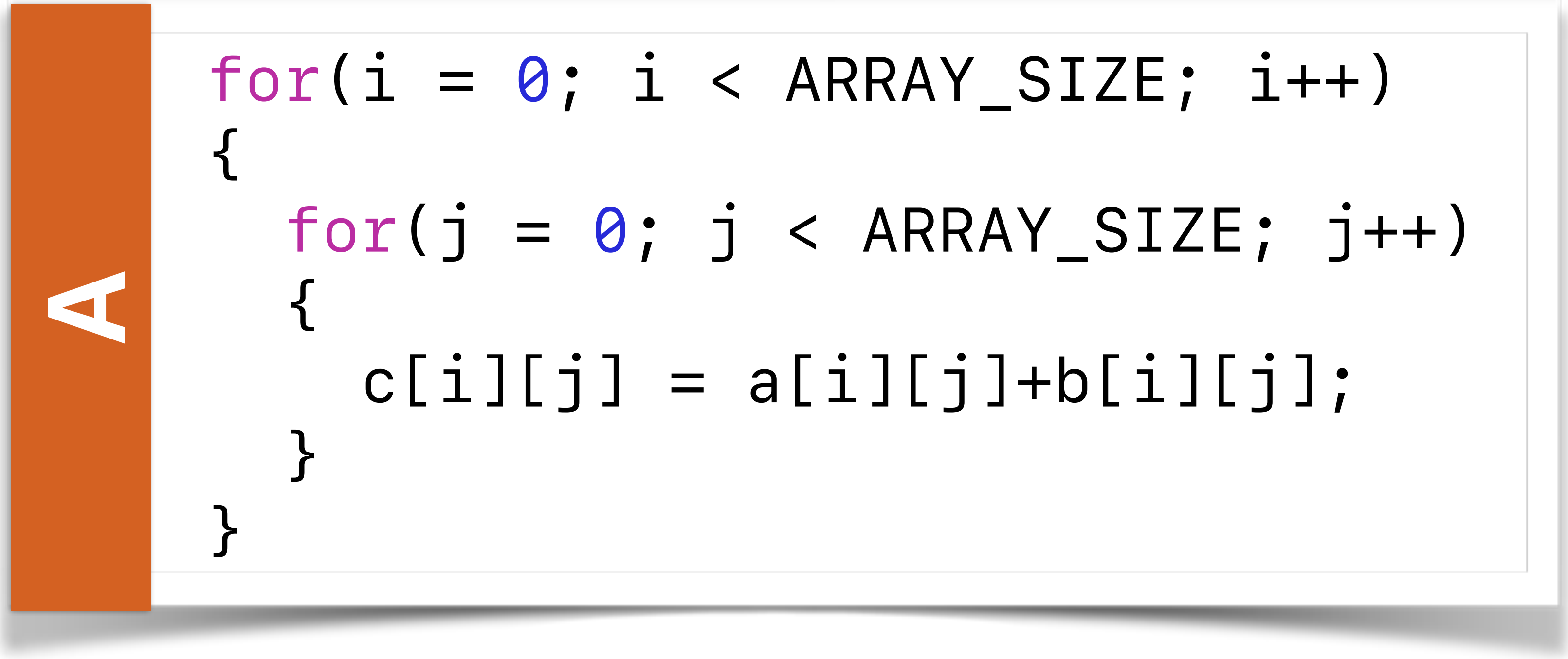
https://software.intel.com/sites/default/files/comment/1716807/how-to-change-frequency-on-linux-pub.txt



```
================================================
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
```

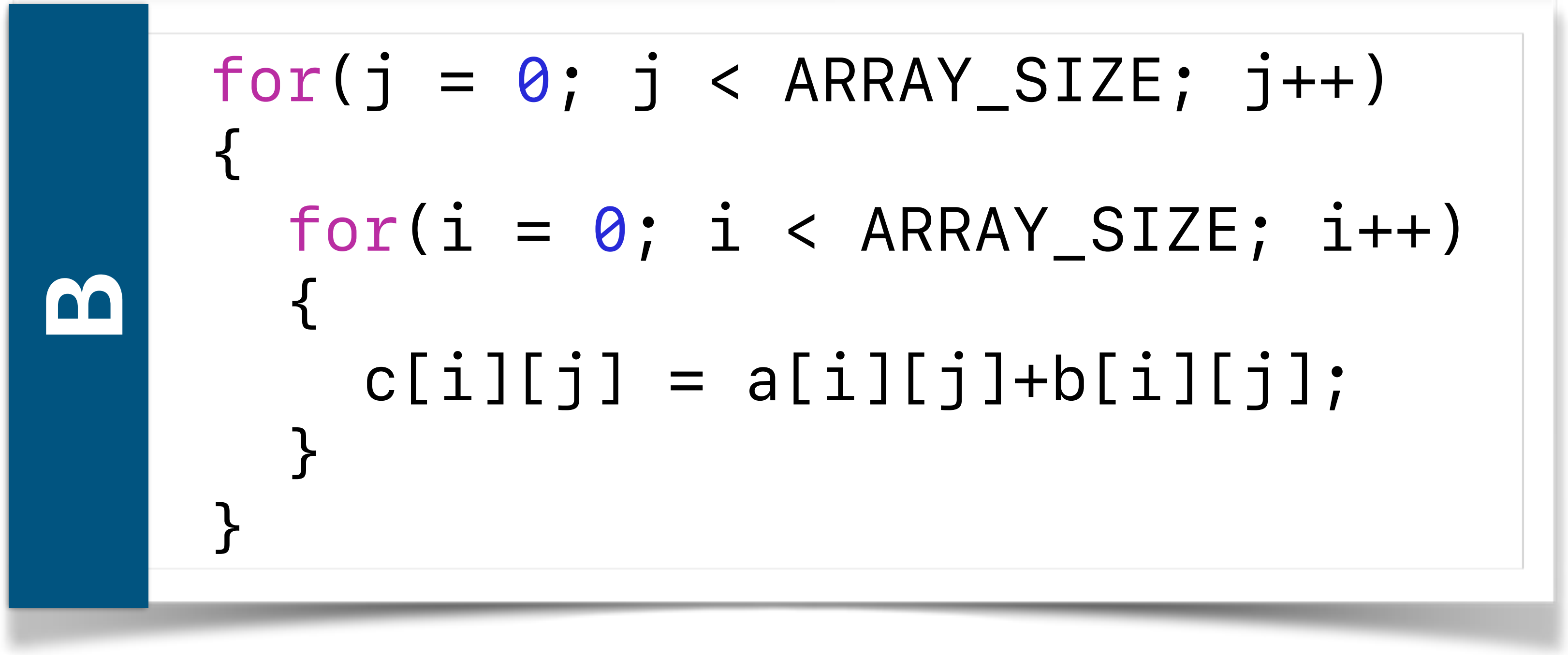
# Revisited the demo with compiler optimizations!

- 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

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

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

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

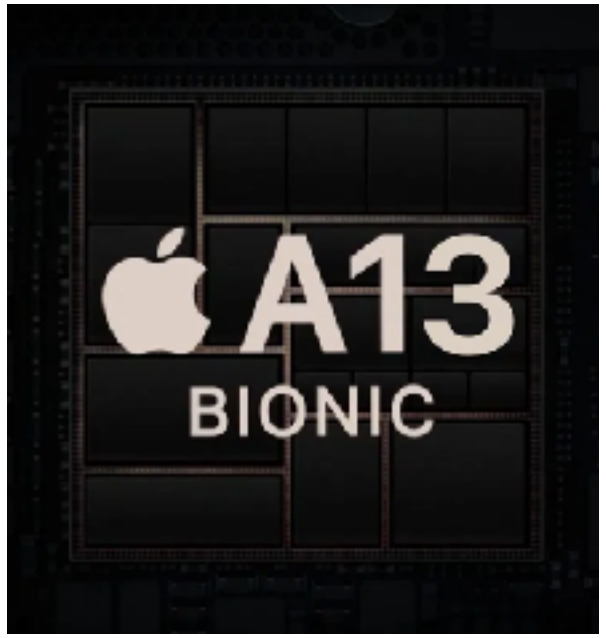
- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, **programmer**
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, **programmer**
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture,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

# Instruction Set Architecture (ISA) & Performance

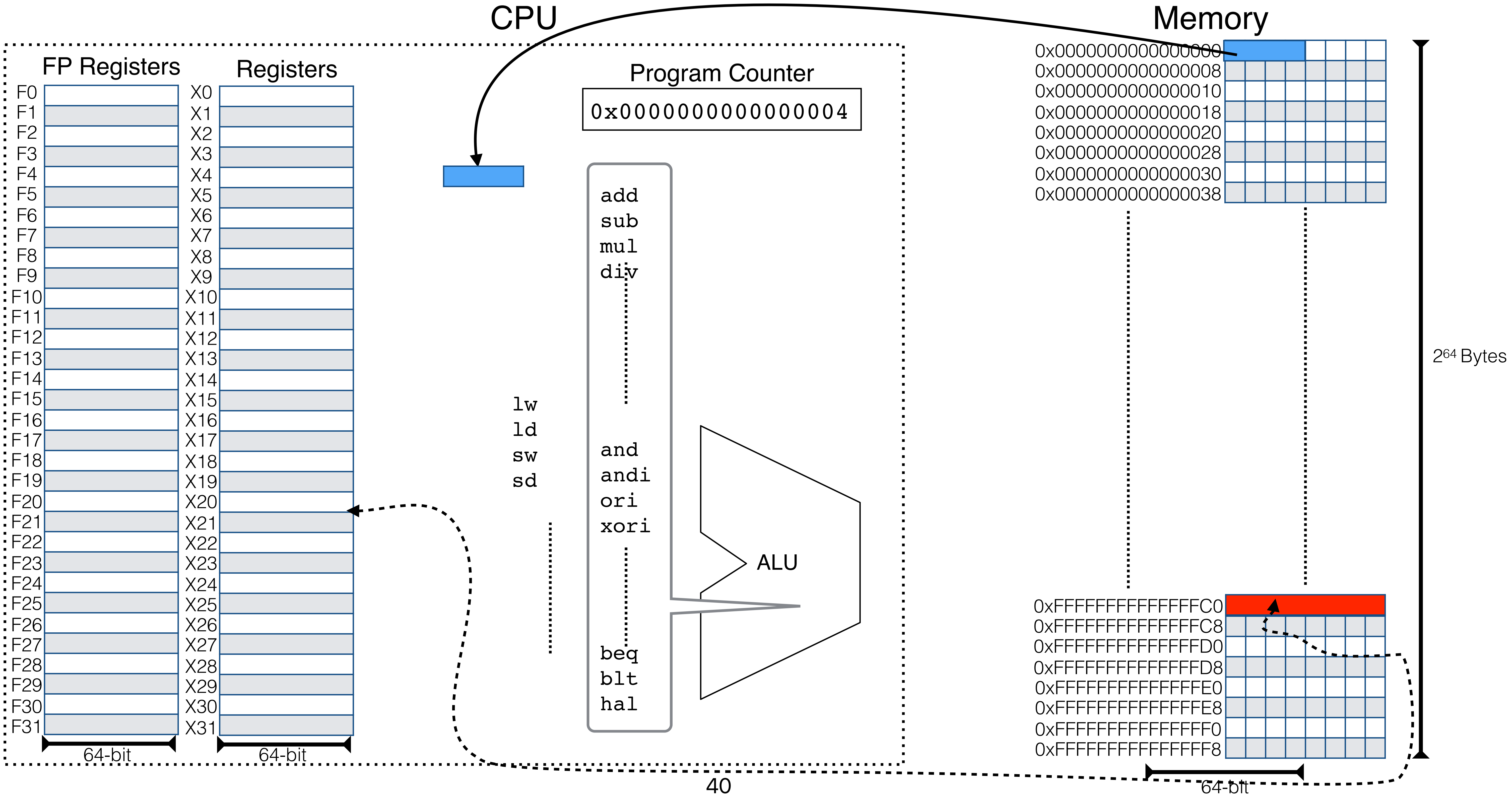
# Recap: ISA — the interface b/w processor/software

- 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



# The abstracted "RISC-V" machine

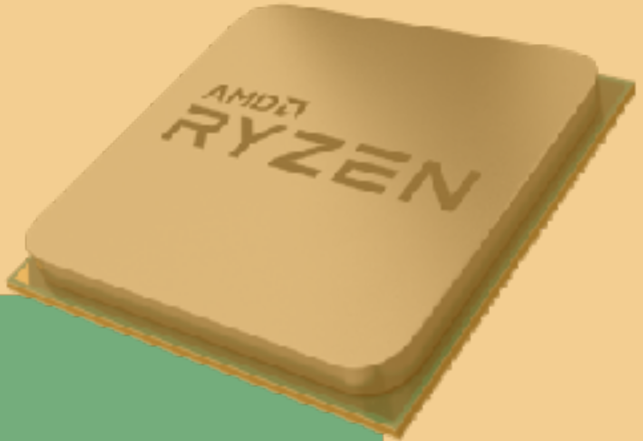


# Subset of RISC-V instructions

| 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

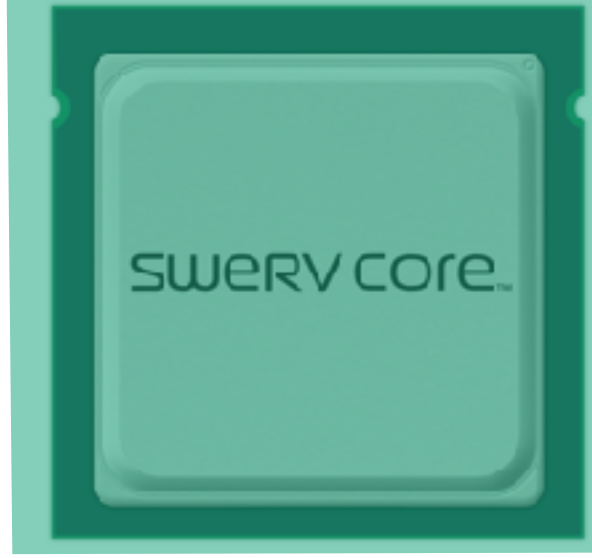


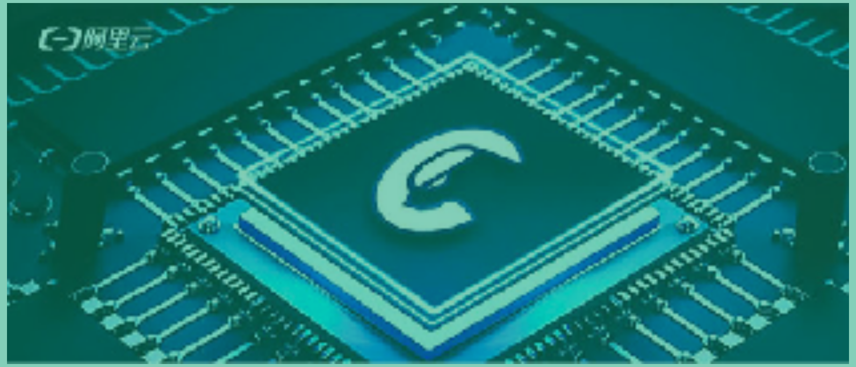
x86

Complex Instruction Set Computers (CISC)

arm

Reduced Instruction Set Computers (RISC)



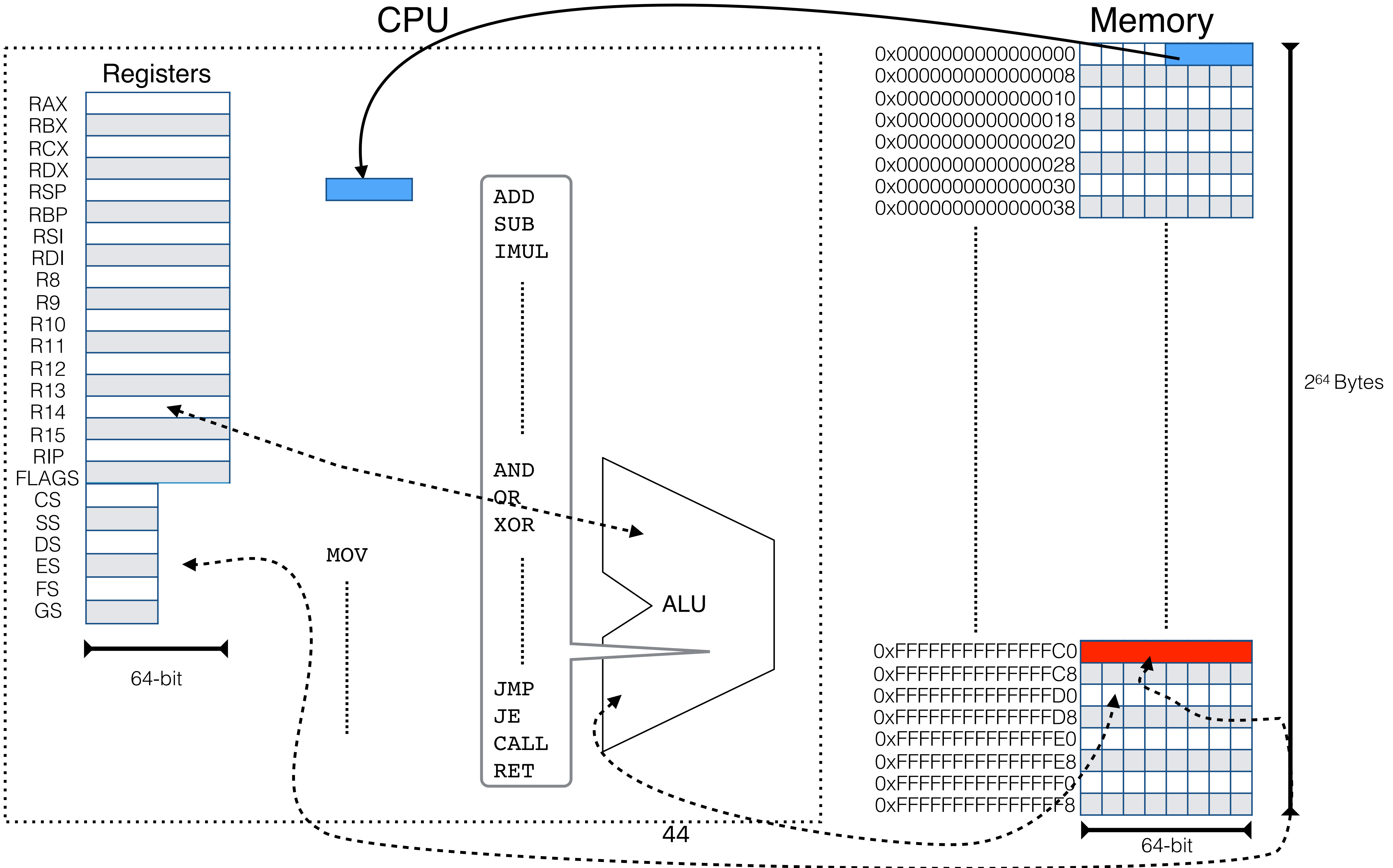


RISC-V

# 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<br>base+index<br>scaled+index<br>scaled+index+offset |
| hardware | simple | complex |

# User-defined data structure

- Programming languages allow user to define their own data types
- In C, programmers can use `struct` to define new data structure

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

**How many bytes each "struct node" will occupy?**

# Memory addressing/alignment

- Almost every popular ISA architecture uses “byte-addressing” to access memory locations
- Instructions generally work faster when the given memory address is aligned
  - Aligned — if an instruction accesses an object of size $n$ at address $X$, the access is **aligned** if $X$ **mod** $n$ **= 0**.
  - Some architecture/processor does not support aligned access at all
  - Therefore, compilers only allocate objects on “aligned” address

# 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.

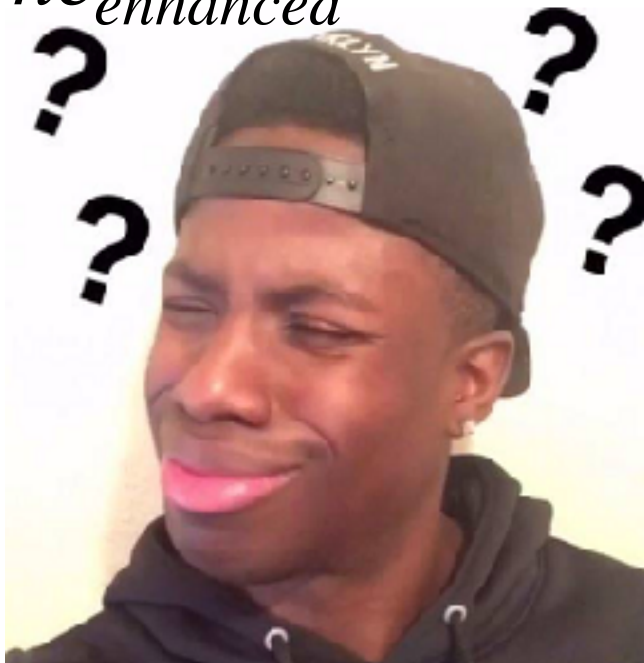
# Amdahl's Law

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

$f$ — The fraction of time in the original program

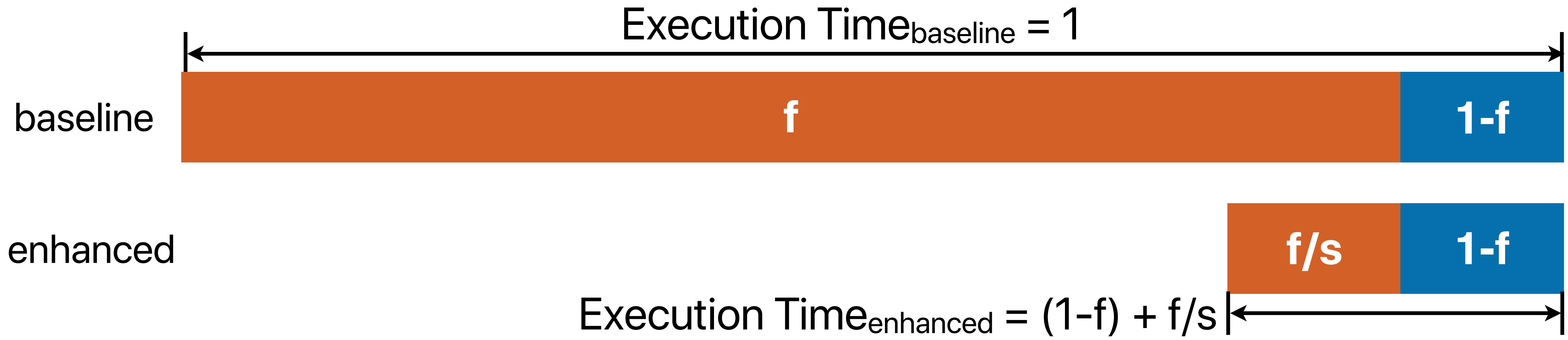
$s$ — The speedup we can achieve on f

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



# Amdahl's Law

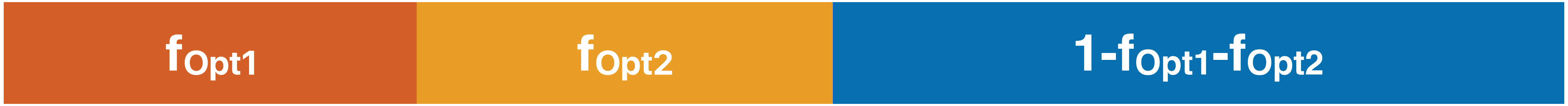
$$Speedup_{enhanced}(f,s) = \frac{1}{(1-f)+\frac{f}{s}}$$


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

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

# Amdahl's Law on Multiple Optimizations

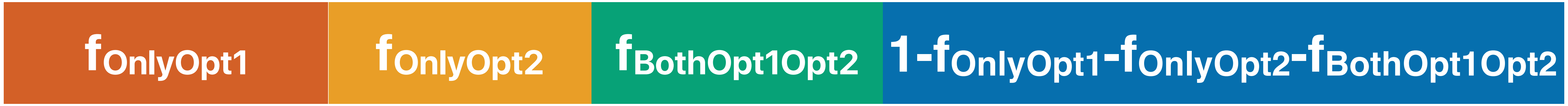
- We can apply Amdahl's law for multiple optimizations
- These optimizations must be dis-joint!
  - If optimization #1 and optimization #2 are dis-joint:



$f_{Opt1}$ $f_{Opt2}$ $1\text{-}f_{Opt1}\text{-}f_{Opt2}$

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

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



$f_{OnlyOpt1}$ $f_{OnlyOpt2}$ $f_{BothOpt1Opt2}$ $1\text{-}f_{OnlyOpt1}\text{-}f_{OnlyOpt2}\text{-}f_{BothOpt1Opt2}$

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

# Amdahl's Law Corollary #1

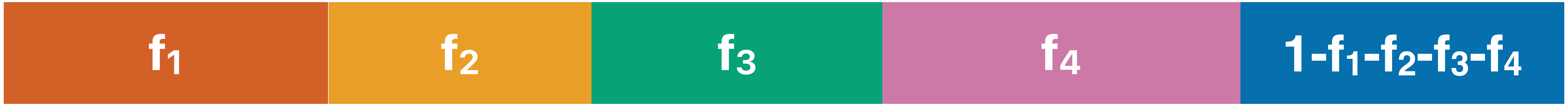
- The maximum speedup is bounded by

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

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

# 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 $Speedup_{max}$!

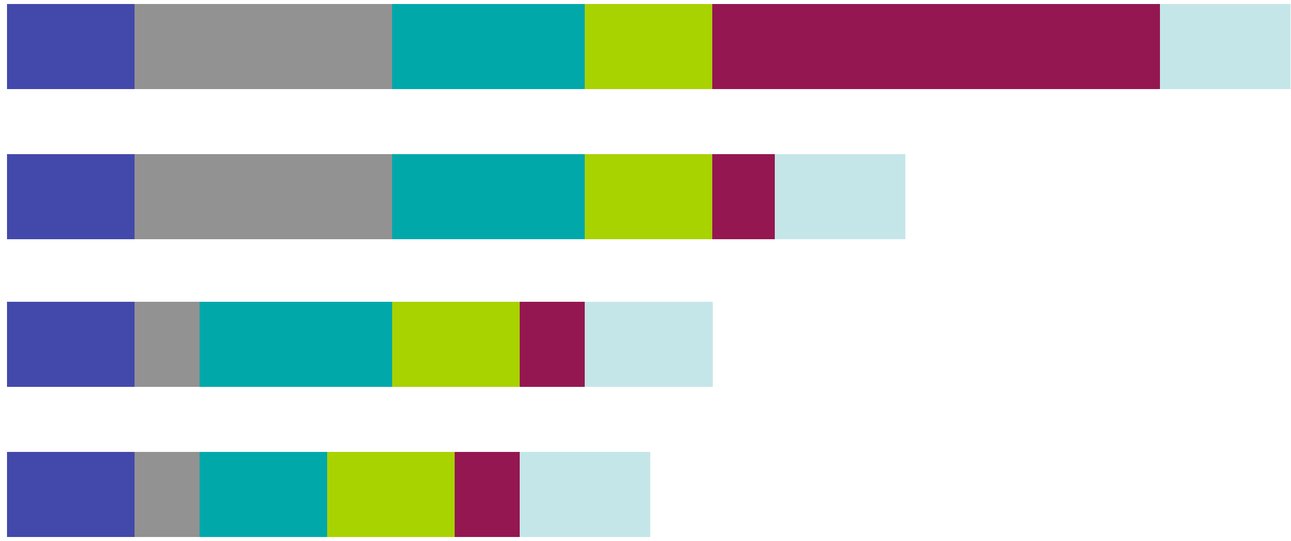
# Corollary #2 — make the common case fast!

- 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...



Common case

🟪 7x => 1.4x

⬜ 4x => 1.3x

🟦 1.3x => 1.1x

Total = 20/10 = 2x

- With optimization, the common becomes uncommon.
- An uncommon case will (hopefully) become the new common case.
- Now you have a new target for optimization.

# 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

$$T_{new} = \frac{T \times 0.9}{9} + T \times 0.1 \times 10$$

$$T_{new} = 1.1T$$

$$\text{Speedup} = \frac{T}{1.1T} = 0.91$$

# 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

8cx

Qualcomm snapdragon

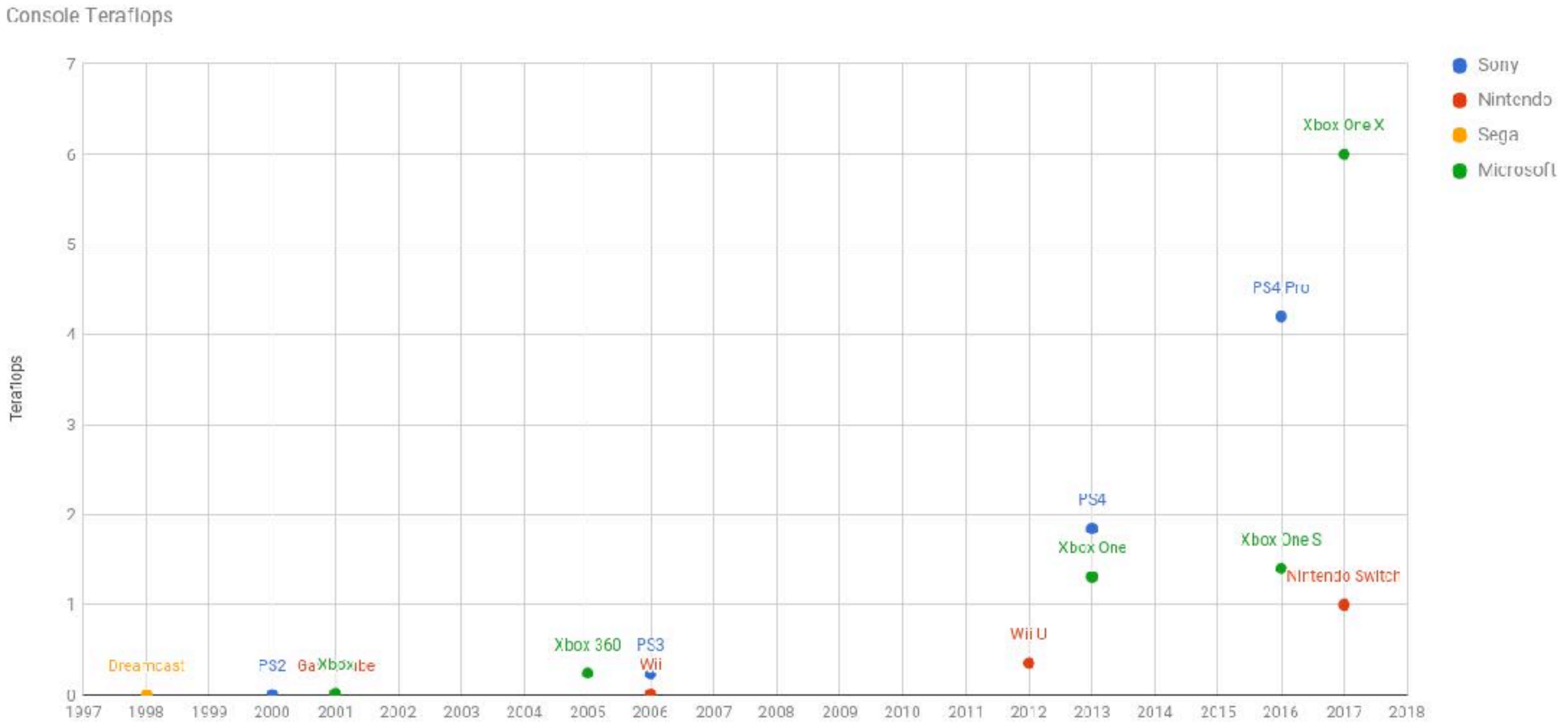
# Extreme Multitasking Performance

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

# 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?

# TFLOPS (Tera FLoating-point Operations Per Second)



# TFLOPS (Tera FLoating-point Operations Per Second)

- TFLOPS does not include instruction count!
  - Cannot compare different ISA/compiler
  - Different CPI of applications, for example, I/O bound or computation bound
  - If new architecture has more IC but also lower CPI?

| | TFLOPS | clock rate |
|---|---|---|
| XBOX One | 6 | 1.75 GHz |
| PS4 Pro | 4 | 1.6 GHz |
| GeForce GTX 1080 | 8.228 | 3.5 GHz |

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

- 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

$$\text{TFLOPS} = \frac{\#\text{ of floating point instructions} / 10^{12}}{\text{Execution Time}}$$

$$= \frac{\text{IC} \times \text{ \% of floating point instructions}}{\text{IC} \times \text{ CPI} \times \text{CycleTime} \times 10^{12}} = \frac{\text{Clock Rate} \times \text{\% FP ins.}}{\text{CPI} \times 10^{12}}$$

# Latency v.s. throughput

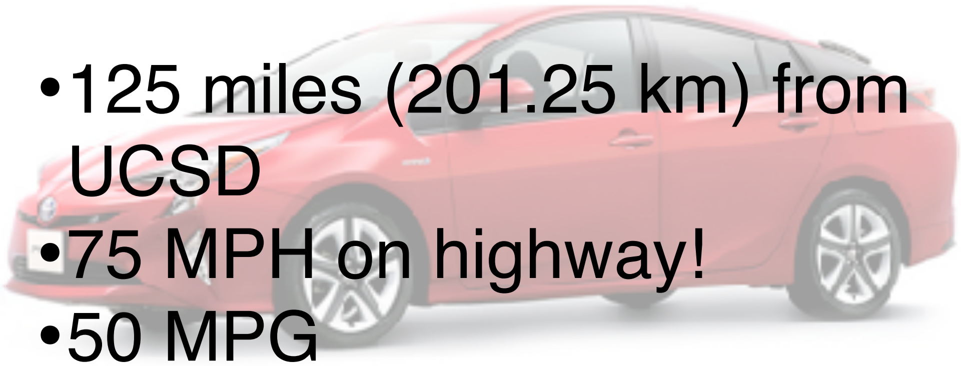
- Consider the following characteristics of flash-based SSDs and Optane-based SSDs.

| | Flash | Optane |
|---|---|---|
| **Latency** | ~ 100 us (read)<br>~ 1 ms (write) | 7 us (read)<br>18 us (write) |
| **Bandwidth** | 3.5 GB/sec (read)<br>2.1 GB/sec (write) | 1.35 GB/sec (read)<br>290 MB/sec (write) |

# Latency and Bandwidth trade-off

- Increase bandwidth can hurt the response time of a single task
- If you want to transfer a 2 Peta-Byte video from UCLA
    - 125 miles (201.25 km) from UCSD
    - Assume that you have a 100Gbps ethernet
        - 2 Peta-byte over 167772 seconds = 1.94 Days
        - 22.5TB in 30 minutes
        - Bandwidth: 100 Gbps

# Or …

| | Toyota Prius<br><br>•125 miles (201.25 km) from UCSD<br>•75 MPH on highway!<br>•50 MPG<br>•Max load: 374 kg = 2,770 hard drives (2TB per drive) | 10Gb Ethernet<br> |
|---|---|---|
| bandwidth | 290GB/sec | 100 Gb/s or<br>12.5GB/sec |
| latency | 4 hours | 2 Peta-byte over 167772 seconds = 1.94 Days |
| response time | You see nothing in the first 4 hours | You can start watching the movie as soon as you get a frame! |