CS 211: Intro to Computer Architecture

13.1: CPU Performance

Minesh Patel

Spring 2025 – Tuesday 22 April

Announcements

- Ongoing
  - **WA9:** due **Friday (April 18) @ 11:59 pm**
  - **PA5:** due **Monday (May 5) @ 11:59 pm**

- Upcoming
  - **WA10:** TBA: planned for Friday
  - **Final Exam:** May 14th

We've Done It: RISC-V CPU!

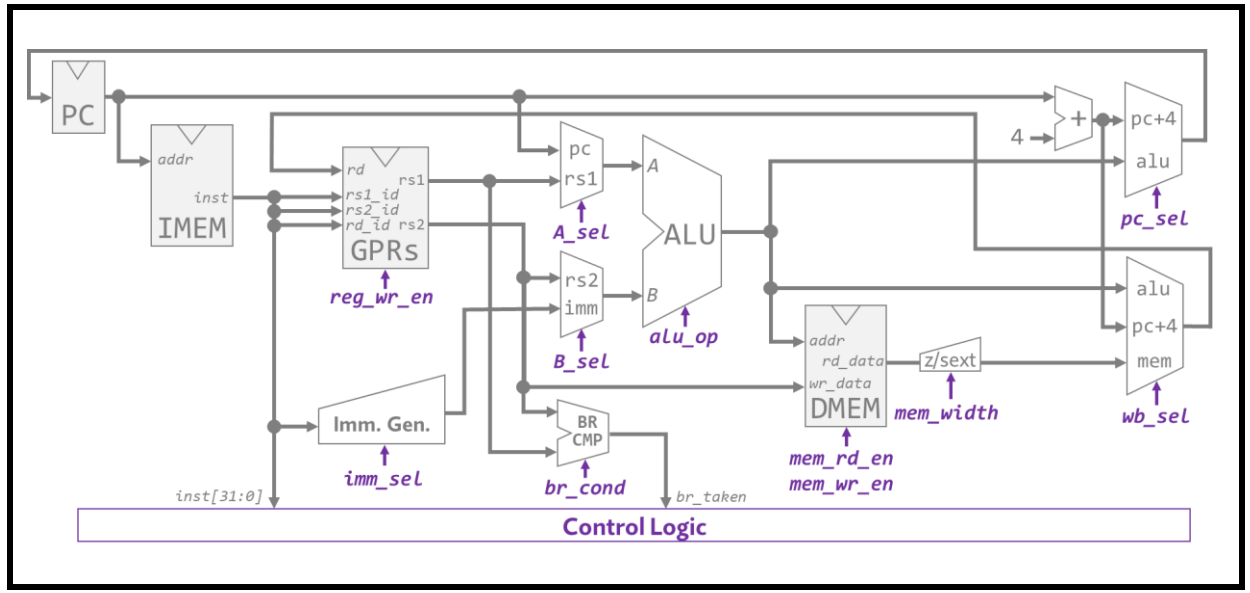
- Everything boils down to **logic operations on numbers**

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        print("Hello, World");
        return EXIT_SUCCESS;
}
```
hello.c

```
main:
        addi sp,sp,-32
        sd ra,24(sp)
        sd s0,16(sp)
        addi s0,sp,32
        mv a5,a0
        sd a1,-32(s0)
        sw a5,-20(s0)
        lla a0,.LC0
        call puts
        li a5,0
        mv a0,a5
        ld ra,24(sp)
        ld s0,16(sp)
        addi sp,sp,32
        jr ra
```
hello.S

```
00001050: 4743 433a 2028 5562 756e 7475 2031 312e  GCC: (Ubuntu 11.
00001060: 342e 302d 3175 6275 6e74 7531 7e32 322e  4.0-1ubuntu1~22.
00001070: 3034 2920 3131 2e34 2e30 0041 3200 0000  04) 11.4.0.A2...
00001080: 7269 7363 7600 0128 0000 0005 7276 3634  riscv..(....rv64
00001090: 6932 7030 5f6d 3270 305f 6132 7030 5f66  i2p0_m2p0_a2p0_f
000010a0: 3270 305f 6432 7030 5f63 3270 3000 002e  2p0_d2p0_c2p0...
000010b0: 7368 7374 7274 6162 002e 696e 7465 7270  shstrtab..interp
000010c0: 002e 6e6f 7465 2e67 6e75 2e62 7569 6c64  ..note.gnu.build
000010d0: 2d69 6400 2e6e 6f74 652e 4142 492d 7461  -id..note.ABI-ta
000010e0: 6700 2e67 6e75 2e68 6173 6800 2e64 796e  g..gnu.hash..dyn
000010f0: 7379 6d00 2e64 796e 7374 7200 2e67 6e75  sym..dynstr..gnu
00001100: 2e76 6572 7369 6f6e 002e 676e 752e 7665  .version..gnu.ve
00001110: 7273 696f 6e5f 7200 2e72 656c 612e 6479  rsion_r..rela.dy
00001120: 6e00 2e72 656c 612e 706c 7400 2e74 6578  n..rela.plt..tex
00001130: 7400 2e72 6f64 6174 6100 2e65 685f 6672  t..rodata..eh_fr
00001140: 616d 655f 6864 7200 2e65 685f 6672 616d  ame_hdr..eh_fram
00001150: 6500 2e70 7265 696e 6974 5f61 7272 6179  e..preinit_array
00001160: 002e 696e 6974 5f61 7272 6179 002e 6669  ..init_array..fi
00001170: 6e69 5f61 7272 6179 002e 6479 6e61 6d69  ni_array..dynami
00001180: 6300 2e64 6174 6100 2e67 6f74 002e 6273  c..data..got..bs
00001190: 7300 2e63 6f6d 6d65 6e74 002e 7269 7363  s..comment..risc
000011a0: 762e 6174 7472 6962 7574 6573 0000 0000  v.attributes....
```
hello



Agenda

- **Performance on a Single-Cycle CPU**
  - Optimizing Instructions per Program
  - Optimizing Time per Instruction
- The Memory Wall

Program Execution Time: Intuition

- How long will this code take to run on our CPU?

```c
#include <stdbool.h>

bool maj(bool a, bool b, bool c)
{
    return a * c + a * b + b * c;
}
```
maj.c

```
maj:
    and  a5, a0, a2
    and  a0, a0, a1
    addw a5, a5, a0
    and  a2, a2, a1
    addw a0, a2, a5
    snez a0, a0
    ret
```
maj.S

7 Instructions

Time = #Instructions x Time/Instruction

7 cycles = 7 instructions x 1 cycle / inst

"single-cycle CPU"

Program Execution Time: Intuition

- How long will this code take to run?

```
void func(uint64_t a[10], uint64_t b[10])
{
    for(int i = 0; i < 10; i++)
        a[i]++;
}
```
inc.c

```
func:
    addi a4,a0,80
.loop:
    ld   a5, 0(a0)
    addi a0, a0,8
    addi a5, a5,1
    sd   a5, -8(a0)
    bne  a0, a4, .loop
    ret
```
inc.S

7 "Static Instructions"

52 "Dynamic Instructions"

$$\text{Time} = \#\text{Instructions} \times \frac{\text{Time}}{\text{Instruction}}$$

52 cycles = 52 instructions x 1 cycle / inst

Improving Performance

- **Static instructions:** the compiled machine code
- **Dynamic instructions:** the sequence of instructions that get executed

$$\texttt{Time} = \underbrace{\texttt{\#Instructions}}_{\textit{Dynamic Instructions}} \times \underbrace{\frac{\texttt{Time}}{\texttt{Instruction}}}_{\textit{Clock Speed}}$$

1. **Optimize the code**
2. **Execute instructions faster**

Seems obvious, but **the challenges** lie in **the details**

Agenda

- Performance on a Single-Cycle CPU
  - **Optimizing Instructions per Program**
  - Optimizing Time per Instruction
- The Memory Wall

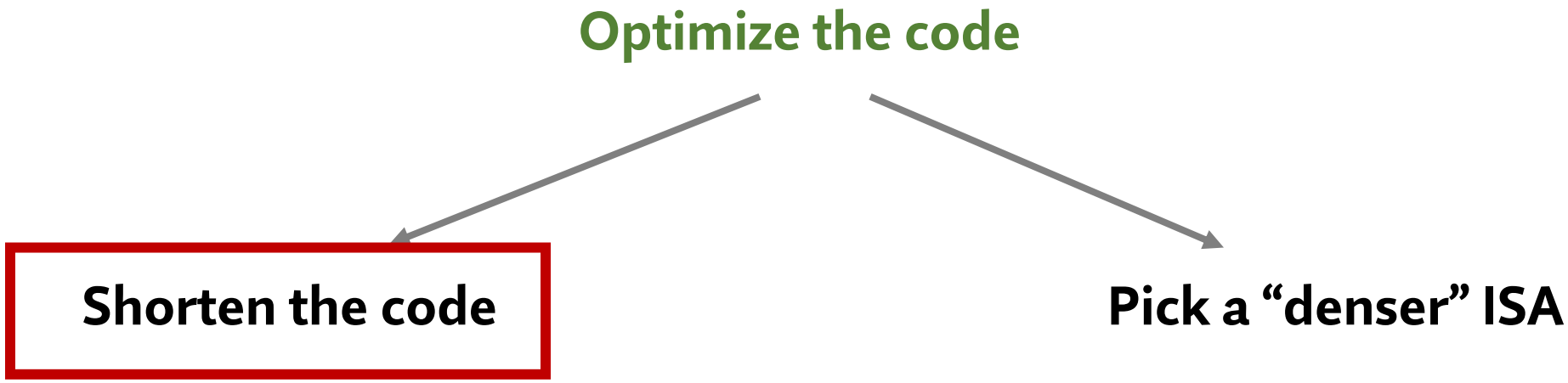
Improving Performance

$$\text{Time} = \underbrace{\#\text{Instructions}}_{\textit{Program Length}} \times \underbrace{\frac{\text{Time}}{\text{Instruction}}}_{\textit{Clock Speed}}$$

1. **Optimize the code**
2. **Execute instructions faster**

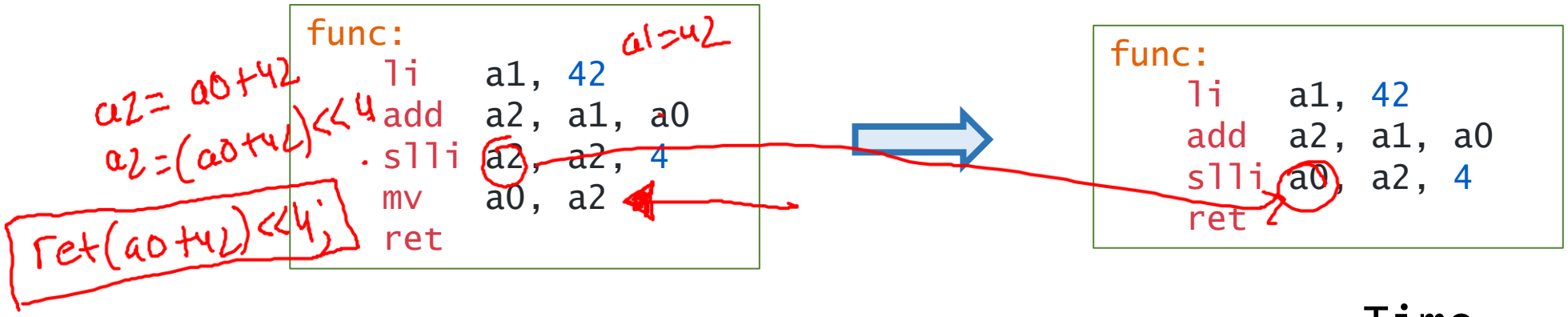
Optimizing Program Code

- Two main ways to go about this on our single-cycle CPU



- Requires rewriting the program
  - **You:** better or more concise algorithm
  - **Compiler:** rely on code optimizations

Compiler Optimization



$$\text{Time} = \#\text{Instructions} \times \frac{\text{Time}}{\text{Instruction}}$$

~~5 cycles = 5 instructions~~ x 1 cycle / instruction

4 cycles = 4 instructions

$$\text{speedup} = \frac{t_{old}}{t_{new}} = \frac{5 \text{ cycles}}{4 \text{ cycles}} = \mathbf{1.25x}$$

Example: Compiler Optimization Levels

```c
uint64_t sum(uint64_t k)
{
    return k == 0 ? 0 : k + sum(k - 1);
}
```

gcc -O0

```
sum:
        addi  sp,sp,-32
        sd    ra,24(sp)
        sd    a0,8(sp)
        ld    a5,8(sp)
        beq   a5,zero,.L2
        ld    a5,8(sp)
        addi  a5,a5,-1
        mv    a0,a5
        call  sum
        mv    a4,a0
        ld    a5,8(sp)
        add   a5,a4,a5
        j     .L4
.L2:
        li    a5,0
.L4:
        mv    a0,a5
        ld    ra,24(sp)
        addi  sp,sp,32
        jr    ra
```

rec → call sum

jr ra = ret

130

gcc -O1

```
sum:
        addi  sp,sp,-16
        sd    ra,8(sp)
        sd    s0,0(sp)
        mv    s0,a0
        bne   a0,zero,.L4
.L2:
        mv    a0,s0
        ld    ra,8(sp)
        ld    s0,0(sp)
        addi  sp,sp,16
        jr    ra
.L4:
        addi  a0,a0,-1
        call  sum
        add   s0,s0,a0
        j     .L2
```

108

gcc -O2/-O3

```
sum:
        mv    a5,a0
        li    a0,0
        beq   a5,zero,.L4
.L3:
        add   a0,a0,a5
        addi  a5,a5,-1
        bne   a5,zero,.L3
        ret
.L4:
        ret
```

3 inst

41

gcc -Os

```
sum:
        mv    a5,a0
        li    a0,0
.L3:
        beq   a5,zero,.L1
        add   a0,a0,a5
        addi  a5,a5,-1
        j     .L3
.L1:
        ret
```

4 inst

52

$$\text{speedup} = \frac{130 \text{ cycles}}{41 \text{ cycles}} = 3.17\text{x}$$

Q: Which code is fastest?

Dynamic instruction count for sum(12)

Example: `-ffast-math`

- Extreme case (**not recommended**): can "cut corners"
  - Approximate math operations
  - Skip/ignore error checking

`-ffast-math`

Sets the options `-fno-math-errno`, `-funsafe-math-optimizations`, `-ffinite-math-only`, `-fno-rounding-math`, `-fno-signaling-nans`, `-fcx-limited-range` and `-fexcess-precision=fast`.

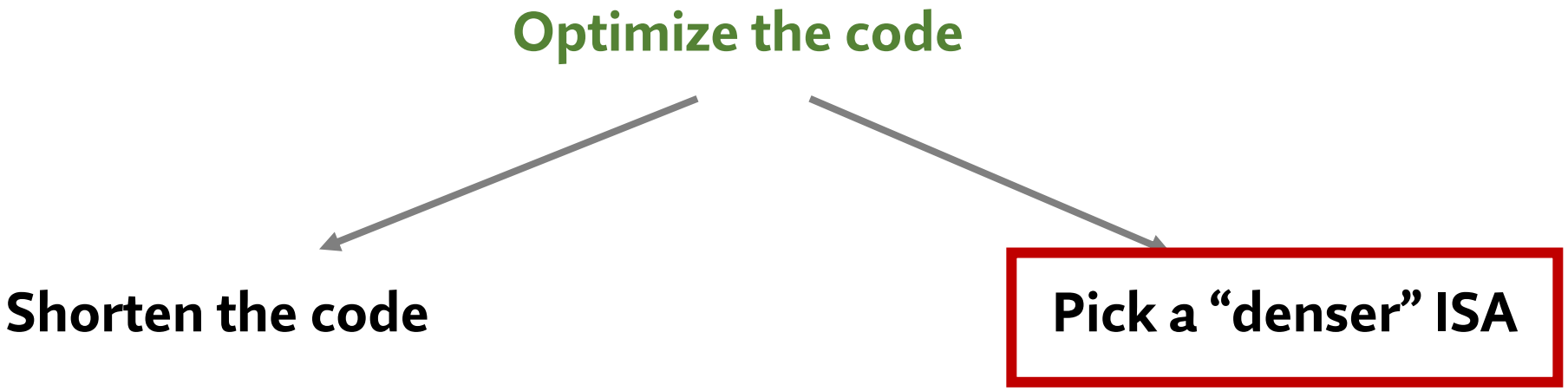
This option causes the preprocessor macro `__FAST_MATH__` to be defined.

This option is not turned on by any `-O` option besides `-Ofast` since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Optimizing Program Code

- Two main ways to go about this on our single-cycle CPU



- Requires rewriting the program
  - **You:** learn a new ASM language ☺
  - **Compiler:** need a different compiler

Choice of ISA

- **Semantic Gap:** how much information does each instruction convey
  - Complex instructions (e.g., full string operations like `strlen`)
  - Simple instructions (e.g., add two numbers)

$X \leftarrow 1.0 + 0.5 \times X + 0.25 \times X^2$

```
; To compute P(x) = C0 + C1*x + C2*x**2
; where C0 = 1.0,  C1 = .5, and C2 = .25

        POLYF   X,#2,PTABLE
        .
        .
        .
PTABLE: .FLOAT  0.25    ; C2
        .FLOAT  0.5     ; C1
        .FLOAT  1.0     ; C0
```

VAX MACRO and Instruction Set Reference Manual

RV32I Base Integer Instructions

Inst	Name	FMT
add	ADD	R
sub	SUB	R
xor	XOR	R
or	OR	R
and	AND	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arith*	R
slt	Set Less Than	R
sltu	Set Less Than (U)	R

https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf

ISA-Level Tradeoffs

Complex Instruction Set
(CISC – "Complex Instruction Set Computer")

- Variable-length instructions (1B – 50B+)
- Hundreds of instruction types

Simple Instruction Set
(RISC – "Reduced Instruction Set Computer")

- Fixed-size instructions (e.g., 32 bits)
- Fewer instruction types

- Tradeoffs:
  - **Complexity:** compiler vs hardware
  - **Backwards compatibility:** rewriting applications is expensive
  - **Performance:**
    - Ease of optimization in compiler vs. hardware
    - Code size (number of instructions)

Godbolt Example: RISC-V vs. x86_64

```
1   void mymemcpy(char *restrict d, const char *restrict s, int len)
2   {
3       while(len--)
4           *d++ = *s++;
5   }
```

RISC-V (64-bits) gcc (trunk)

```
1   mymemcpy:
2           li      a5,0
3   .L2:
4           sext.w  a4,a5
5           bne     a4,a2,.L3
6           ret
7   .L3:
8           add     a3,a1,a5
9           lbu     a3,0(a3)
10          add     a4,a0,a5
11          addi    a5,a5,1
12          sb      a3,0(a4)
13          j       .L2
```

x86-64 gcc 13.2

```
1   mymemcpy:
2           mov     ecx, edx
3           rep movsb
4           ret
```

Pseudocode for REP MOVSB

```
Repeat:
        MEM[dst] = MEM[src]
        SRC++;
        DST++;
        COUNT--;
Until COUNT == 0;
```

x86_64 String Operations: Small Semantic Gap

`rep movs dst src`

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; FI;
    ELSE
        Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg – 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
          IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
          or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
     OD;
```

```
DEST ← SRC;
IF (Byte move)
    THEN IF DF = 0
        THEN
            (R|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

"Intel 64 and IA-32 Architectures Software Developer's Manual" Volume 2B: Instruction Set Reference

Agenda

- Performance on a Single-Cycle CPU
  - Optimizing Instructions per Program
  - **Optimizing Time per Instruction**
- The Memory Wall

Improving Performance

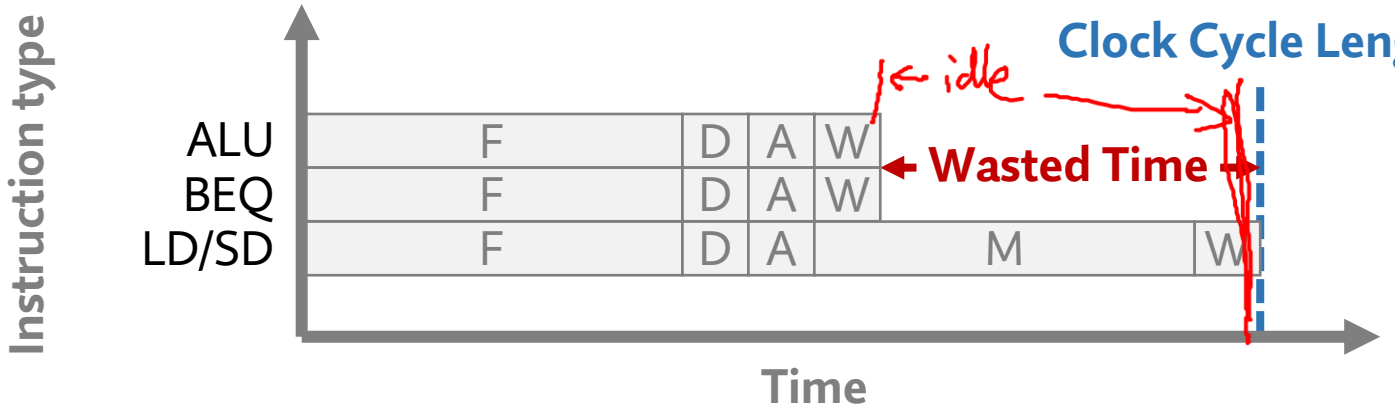
$$\text{Time} = \underbrace{\#\text{Instructions}}_{\textit{Program Length}} \times \underbrace{\frac{\text{Time}}{\text{Instruction}}}_{\textit{Clock Speed}}$$

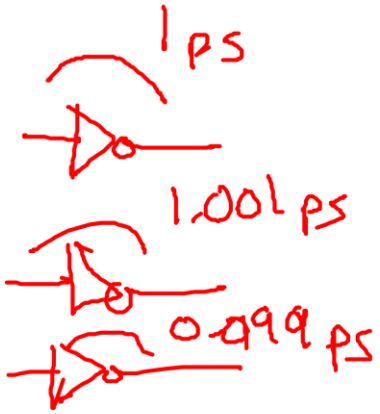
1. Optimize the code

2. Execute instructions faster

Increasing Clock Speeds

- **Observation:** a single-cycle CPU is constrained by the **slowest instruction**





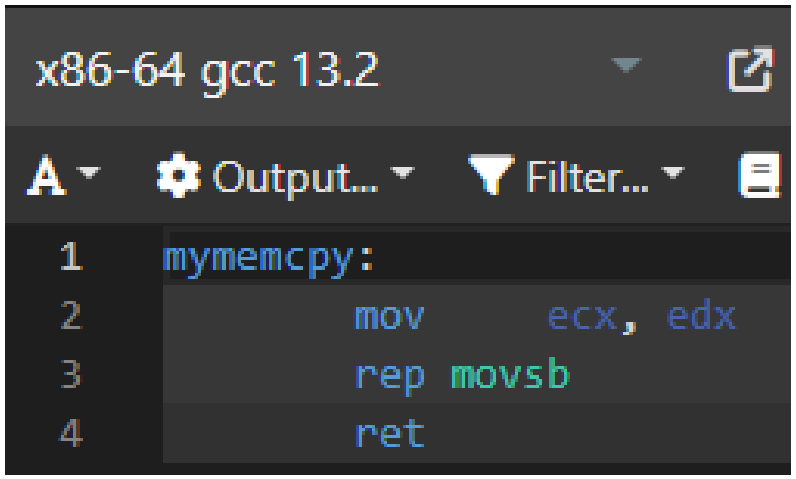
- Faster clock requires optimizing the **slowest instruction**

1. **Simplify instructions**: avoid slow instructions in the ISA
2. **Better circuit technology**: Everything becomes faster
3. **Better microarchitecture**: optimize the **slowest instruction**

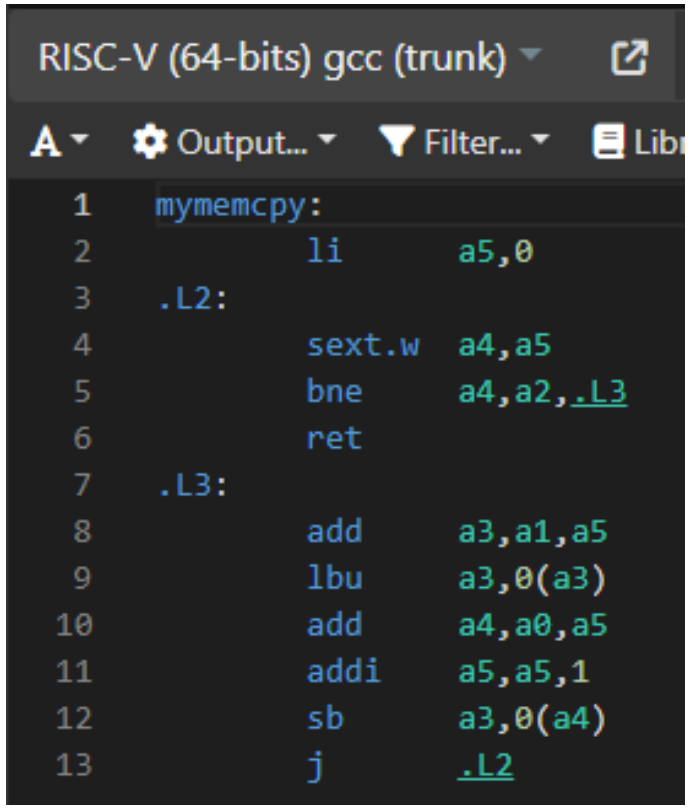
No free lunch!

(1/3) Simplify Instructions

- **Unclear benefits**: more + fast vs. few + slow



Similar performance in many cases

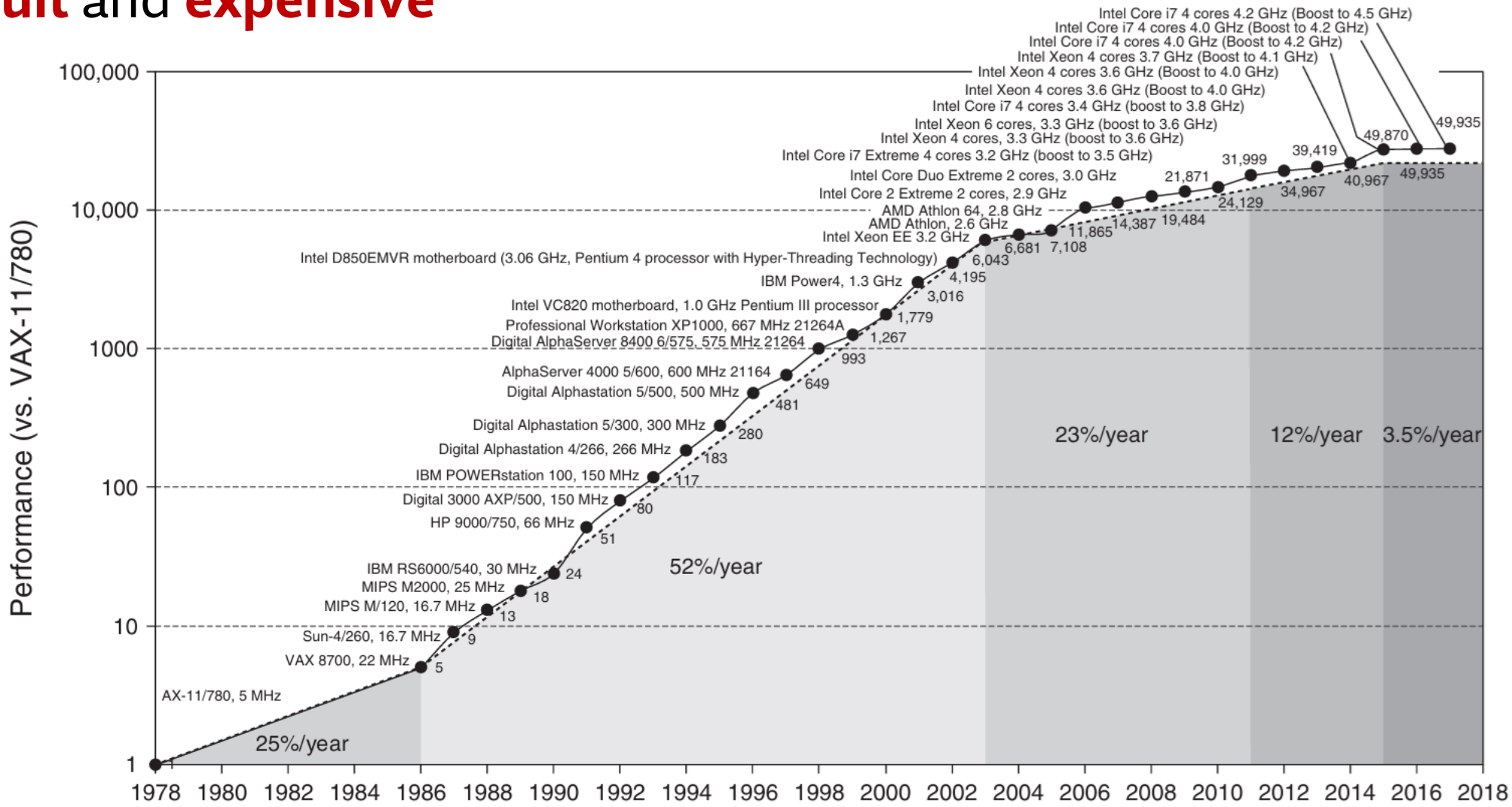


- Very easy to program
- Very slow to execute

- Harder to program
- Instructions are quick to execute

(2/3) Better Circuit Technology

- **Difficult** and **expensive**



Hennessy & Patterson, "A Quantitative Approach to Computer Architecture," 6/E., Figure 2.2

(3/3) Better Microarchitecture: Efficient Logic

- Doable, but **difficult** and **expensive**

```
func:
    add a0, a0, a1
```

$$\text{Time} = \#\text{Instructions} \times \frac{\text{Time}}{\text{Instruction}}$$

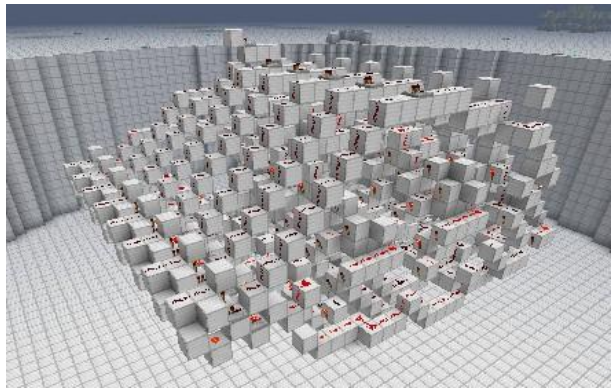
~~5 ns~~ = 1 instructions x ~~5 ns / inst~~

4 ns = 4 ns

Simple Adder
(Carry Ripple)

https://i.ytimg.com/vi/ZaC_E8CjejE/maxresdefault.jpg

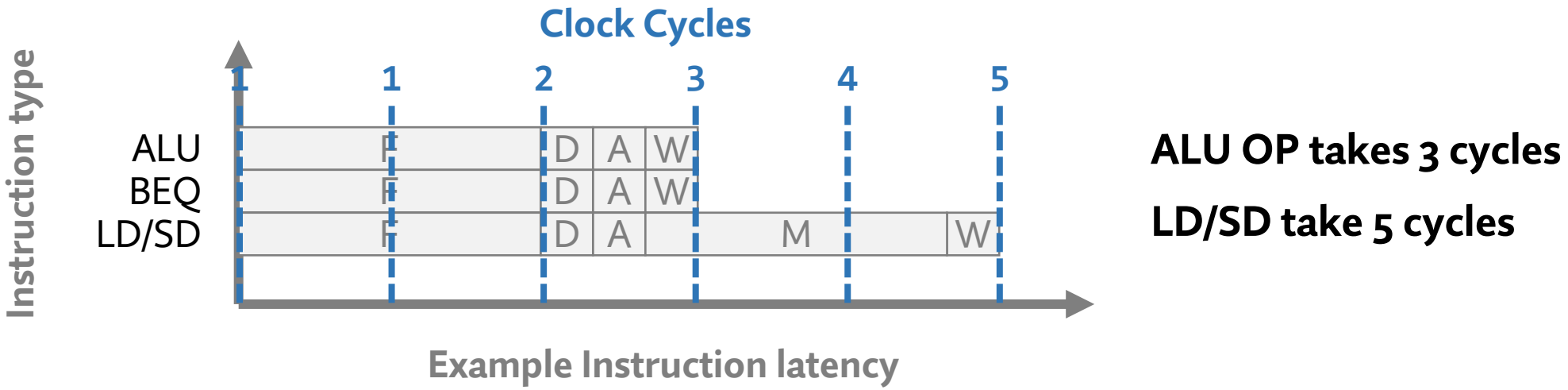
Fancy Adder
(Kogge-Stone)



https://www.planetminecraft.com/project/8-bit-kogge-stone-adder/

(3/3) Better Microarchitecture: Multiple Cycles

- **Key Idea:** set a **fast clock** and allow instruction to take **multiple cycles**



ALU OP takes 3 cycles

LD/SD take 5 cycles

- *Multi-cycle logic design is beyond the scope of this course*

Example: Multi-Cycle CPU Performance

```
func:
    addi a4,a0,80
.loop:
    ld   a5, 0(a0)
    addi a0, a0,8
    addi a5, a5,1
    sd   a5, -8(a0)
    bne  a0, a4, .loop
    ret
```

7 Static inst.

52 Dynamic inst.

inc.S

Instruction Type	Latency (Cycles)
Non-Memory (e.g., ALU, BR)	1 → 32 = 32
Memory (LOAD/STORE)	8 → 20 +160
	192

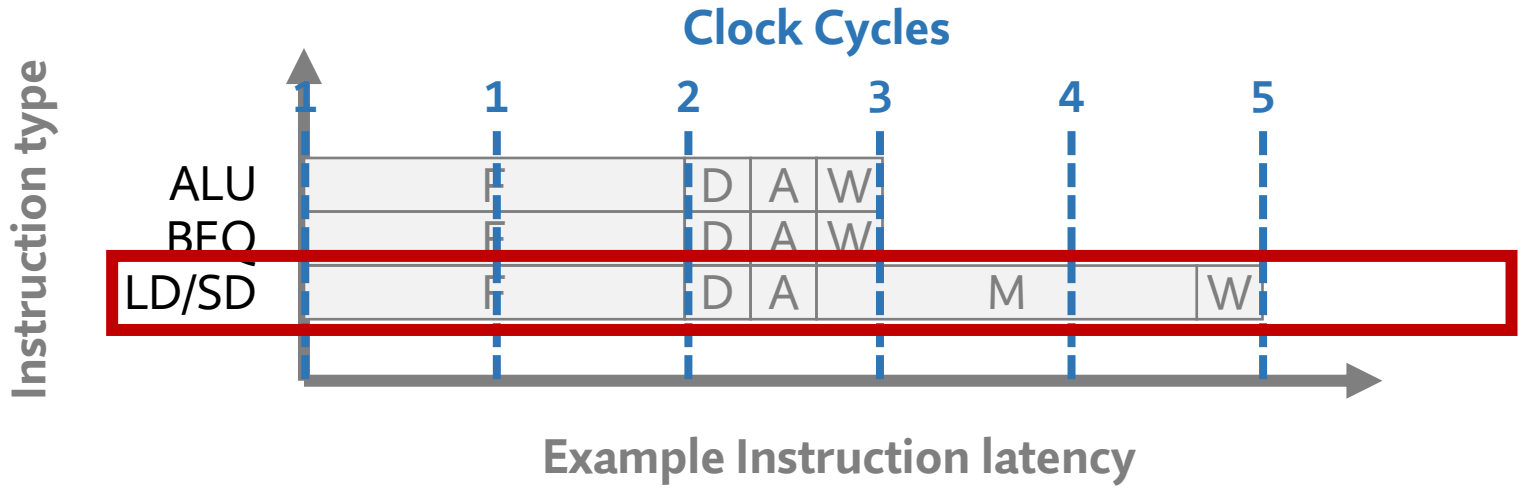
$$\text{Time} = \#\text{Instructions} \times \frac{\text{Time}}{\text{Instruction}}$$

= (20 memory inst **x** 8 cycles/inst)
+ (32 memory inst **x** 1 cycles/inst)
= 160 + 32
= **192**

Must know the breakdown of program instructions

Example: Multi-Cycle CPU Performance

- Performance depends on **which instructions** we execute
  - Prefer faster instructions!

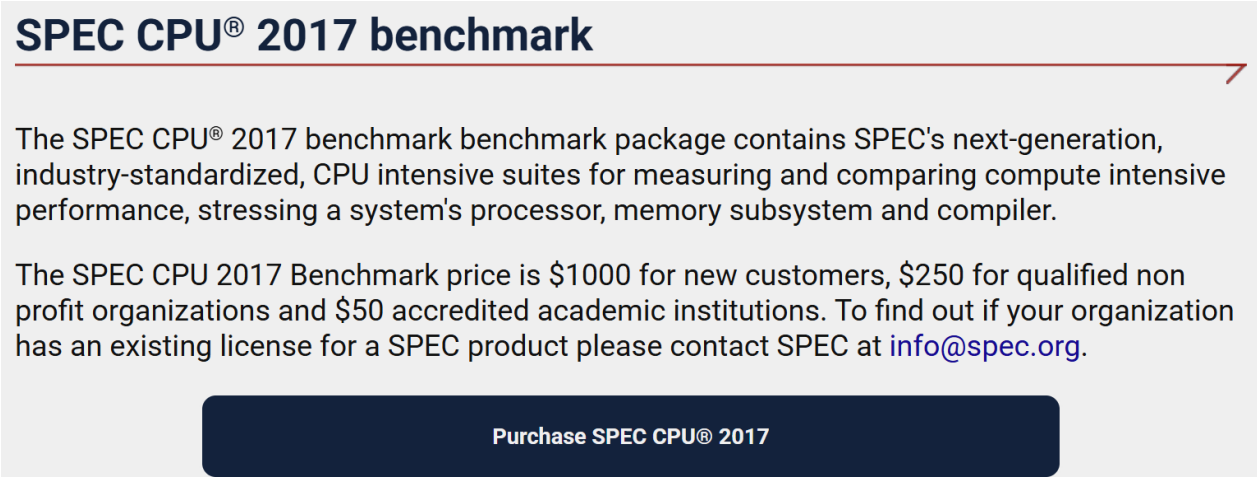


General rule of thumb:
memory is slow

CPU Performance (Generally)

- **Q:** What does CPU performance mean?
- **A: Nothing!** Depends on which instructions you choose to run
  - "Workloads": the programs that we are interested in

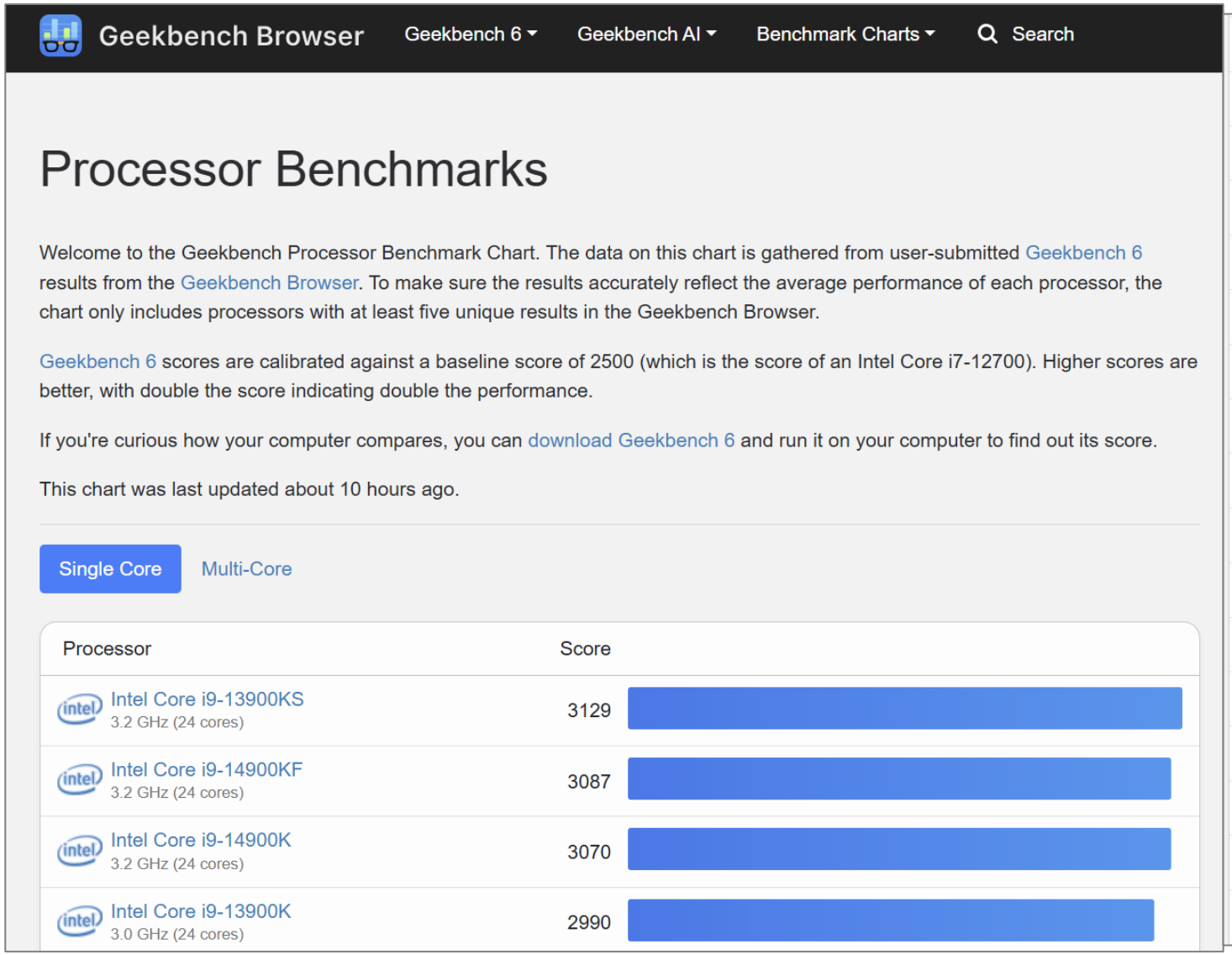
- **Benchmarks:** standard workloads used to compare performance
  - Ideally (!) representative of real-world use cases

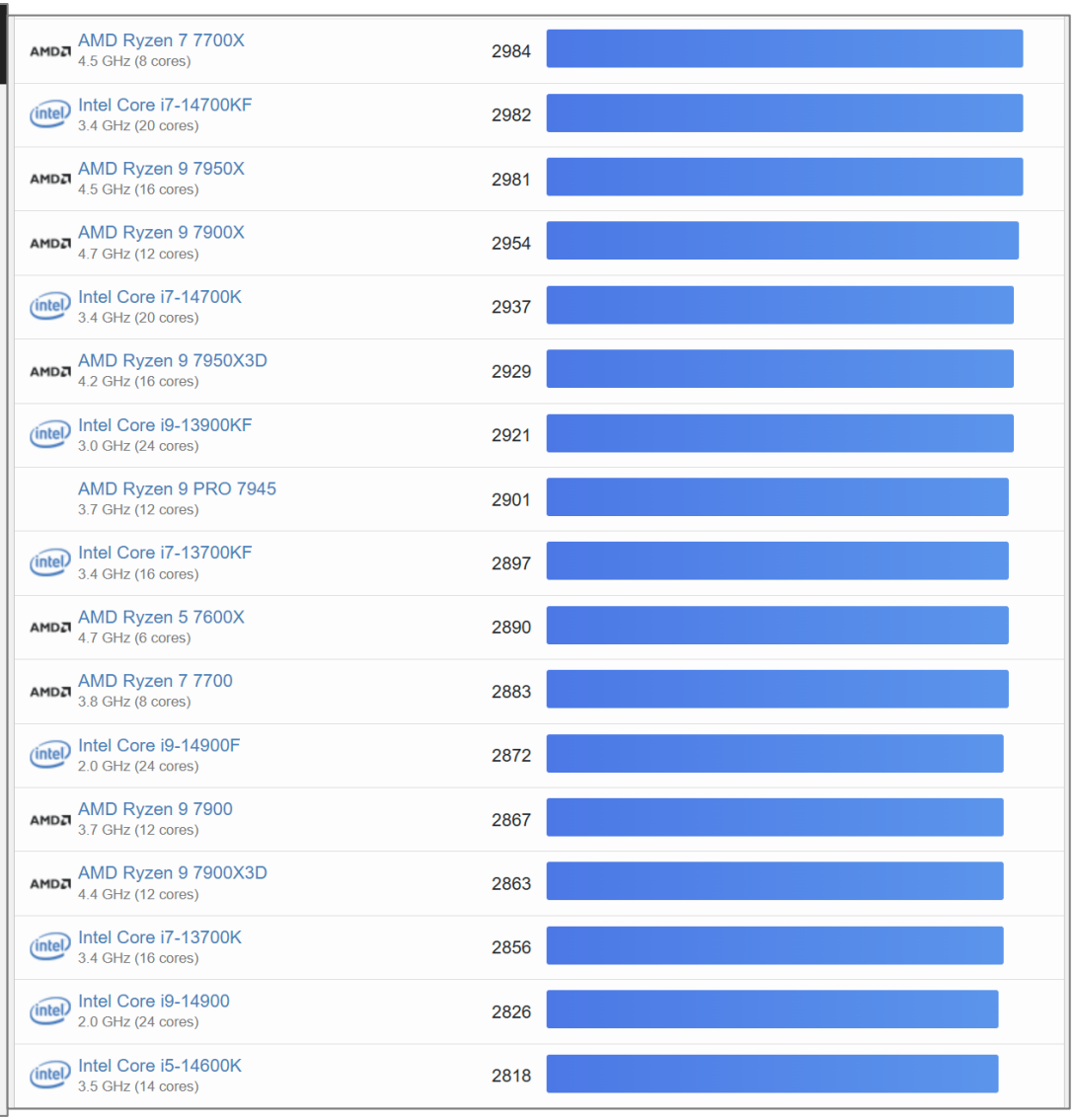


https://www.spec.org/cpu2017/

https://www.geekbench.com/

Example: Geekbench Rankings





https://browser.geekbench.com/processor-benchmarks

CS 211: Intro to Computer Architecture

13.1: CPU Performance

Minesh Patel

Spring 2025 – Tuesday 22 April