

# Assembly vs. machine language

---

- ❑ So far we've been using **assembly language**.
  - We assign names to operations (e.g., **add**) and operands (e.g., **\$t0**).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- ❑ Programs must eventually be translated into **machine language**, a binary format that can be stored in memory and decoded by the CPU.
- ❑ MIPS machine language is designed to be easy to decode.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
- ❑ Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

# Three MIPS formats

- ❑ simple instructions all 32 bits wide
- ❑ very structured, no unnecessary baggage
- ❑ only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Signed value

# Constants

---

- ❑ Small constants are used quite frequently (50% of operands)

e.g.,     A = A + 5;  
           B = B + 1;  
           C = C - 18;

- ❑ MIPS Instructions:

```
addi $29, $29, 4  
slti $8, $18, 10  
andi $29, $29, 6  
ori  $29, $29, 4
```

# Larger constants

- ❑ Larger constants can be loaded into a register 16 bits at a time.
  - The load upper immediate instruction **lui** loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  - An immediate logical OR, **ori**, then sets the lower 16 bits.
- ❑ To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui $s0, 0x003D          # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900     # $s0 = 003D 0900
```

- ❑ This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.
- ❑ Pseudo-instructions may contain large constants. Assemblers will translate such instructions correctly.

# Loads and stores

---

- ❑ The limited 16-bit constant can present problems for accesses to global data.
  - Let's assume the assembler puts a variable at address 0x10010004.
  - 0x10010004 is bigger than 32,767
- ❑ In these situations, the assembler breaks the immediate into two pieces.

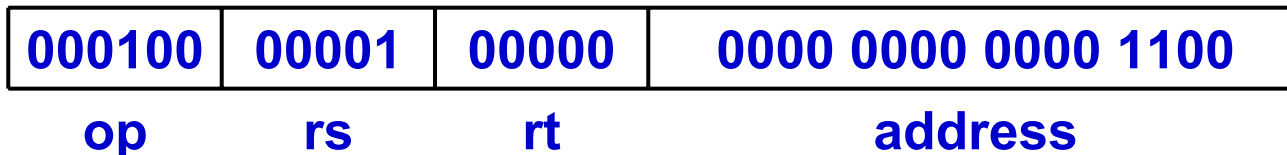
```
lui    $at, 0x1001           # 0x1001 0000
lw     $t1, 0x0004($at)     # Read from Mem[0x1001 0004]
```

# Branches

- ❑ For branch instructions, the constant field is not an address, but an *offset* from the *next* program counter (PC+4) to the target address.

```
        beq    $at, $0, L
        add    $v1, $v0, $0
        add    $v1, $v1, $v1
        j     Somewhere
L:      add    $v1, $v0, $v0
```

- ❑ Since the branch target L is three *instructions* past the first **add**, the address field would contain  $3 \times 4 = 12$ . The whole **beq** instruction would be stored as:



# Larger branch constants

---

- ❑ Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- ❑ If you do need to branch further, you can use a jump with a branch. For example, if “Far” is very far away, then the effect of:

```
    beq  $s0, $s1, Far
    ...
```

can be simulated with the following actual code.

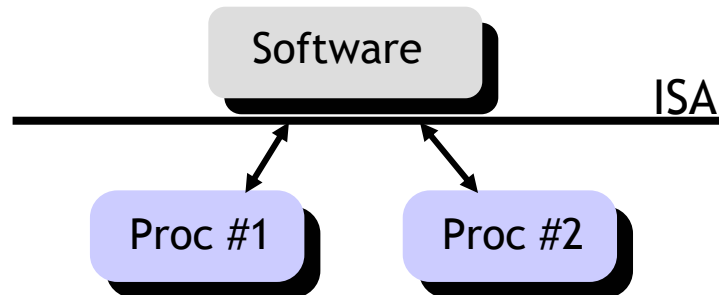
```
        bne  $s0, $s1, Next
        j    Far
Next:   ...
```

- ❑ Again, the MIPS designers have taken care of the common case first.

# Summary Instruction Set Architecture (ISA)

---

- ❑ The ISA is the interface between hardware and software.
- ❑ The ISA serves as an **abstraction layer** between the HW and SW
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent



- ❑ An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars.
- ❑ Before ISAs, software was re-written for each new machine.

# RISC vs. CISC

---

- ❑ MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy, one of the authors of our textbook.
- ❑ The architecture is similar to that of other RISC architectures, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors.
- ❑ Older processors used complex instruction sets, or **CISC** architectures.
  - Many powerful instructions were supported, making the assembly language programmer's job much easier.
  - But this meant that the processor was more complex, which made the hardware designer's life harder.
- ❑ Many new processors use reduced instruction sets, or **RISC** architectures.
  - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
  - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- ❑ Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.

# RISC vs. CISC

---

## □ Characteristics of ISAs

CISC	RISC
Variable length instruction	Single word instruction
Variable format	Fixed-field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

# A little ISA history

---

- ❑ **1964: IBM System/360, the first computer family**
  - IBM wanted to sell a range of machines that ran the same software
- ❑ **1960's, 1970's: Complex Instruction Set Computer (CISC) era**
  - Much assembly programming, compiler technology immature
  - Simple machine implementations
  - Complex instructions simplified programming, little impact on design
- ❑ **1980's: Reduced Instruction Set Computer (RISC) era**
  - Most programming in high-level languages, mature compilers
  - Aggressive machine implementations
  - Simpler, cleaner ISA's facilitated pipelining, high clock frequencies
- ❑ **1990's: Post-RISC era**
  - ISA complexity largely relegated to non-issue
  - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
  - ISA compatibility outweighs any RISC advantage in general purpose
  - Embedded processors prefer RISC for lower power, cost
- ❑ **2000's: ??? EPIC? Dynamic Translation?**

---

# Chapter 4 – Assessing and Understanding Performance

# Why know about performance

---

## ❑ Purchasing Perspective:

- **Given a collection of machines, which has the**
  - Best Performance?
  - Lowest Price?
  - Best Performance/Price?

## ❑ Design Perspective:

- **Faced with design options, which has the**
  - Best Performance Improvement?
  - Lowest Cost?
  - Best Performance/Cost ?

## ❑ Both require

- **Metric for evaluation**
- **Basis for comparison**

# Computer Performance: TIME, TIME, TIME

---

## □ Response Time (latency)

- How long does it take for my job to run?
- How long does it take to execute a job?
- How long must I wait for the database query?

## □ Throughput

- How many jobs can the machine run at once?
- What is the average execution rate?
- How much work is getting done?

□ *If we upgrade a machine with a new processor what do we increase?*

*If we add a new machine to the lab what do we increase?*

# Book's Definition of Performance

---

- ❑ For some program running on machine X,

$$\text{Performance}_x = 1 / \text{Execution time}_x$$

- ❑ "X is n times faster than Y"

$$\text{Performance}_x / \text{Performance}_y = n$$

- ❑ **Problem:**

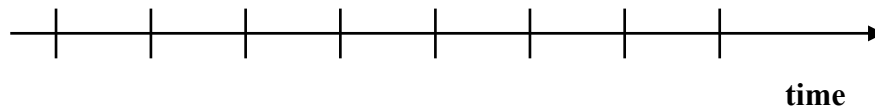
- machine A runs a program in 20 seconds
- machine B runs the same program in 25 seconds

# Clock Cycles

- ❑ Instead of reporting execution time in seconds, we often use cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- ❑ Clock “ticks” indicate when to start activities (one abstraction):



- ❑ cycle time = time between ticks = seconds per cycle
- ❑ clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 200 Mhz. clock has a  $\frac{1}{200 \times 10^6} \times 10^9 = 5$  nanoseconds cycle time

# How to Improve Performance

---

$$\square \frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- So, to improve performance (everything else being equal) you can either

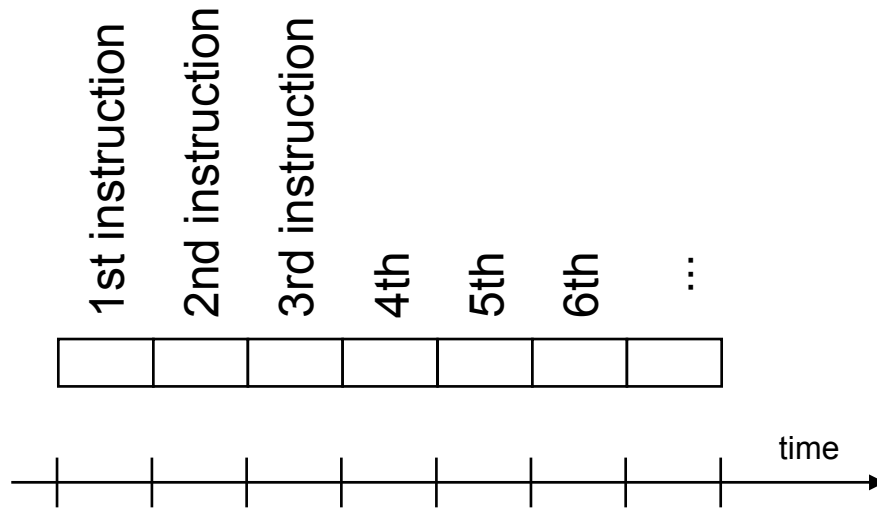
           ↓ the # of required cycles for a program, or

           ↓ the clock cycle time or, said another way,

           ↑ the clock rate.

# How many cycles are for a program?

- ❑ Could assume that # of cycles = # of instructions



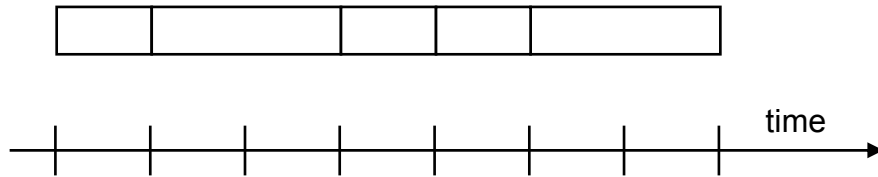
*This assumption is incorrect,*

*different instructions take different amounts of time on different machines.*

*Why? hint: remember that these are machine instructions, not lines of C code*

# Different numbers of cycles for different instructions

---



- ❑ Multiplication takes more time than addition
- ❑ Floating point operations take longer than integer ones
- ❑ Accessing memory takes more time than accessing registers
  
- ❑ Important point: changing the cycle time often changes the number of cycles required for various instructions (more later)

# Example

---

- Our favorite program runs in 10 seconds on computer A, which has a 400 Mhz. clock. We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program. What clock rate should we tell the designer to target?

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

For program A: 10 seconds =  $\text{Cycles}_A \times 1/400\text{MHz}$

For program B: 6 seconds =  $\text{Cycles}_B \times 1/\text{clock rate}_B$

$\text{Cycles}_B = 1.2 \text{Cycles}_A$

$\text{Clock rate}_B = 800\text{MHz}$

# Now that we understand cycles

---

- **A given program will require**
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- **We have a vocabulary that relates these quantities:**
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - **CPI** (cycles per instruction)
    - a floating point intensive application might have a higher CPI
  - **MIPS** (millions of instructions per second)
    - this would be higher for a program using simple instructions

# Another Way to Compute CPU Time

---

$$\text{CPU Time (or, Execution Time)} = \frac{\text{\# of instructions}}{\text{program}} \times \frac{\text{\# of cycles}}{\text{instruction}} \times \frac{\text{\# of seconds}}{\text{cycle}}$$

$$= \text{instruction count} \times \text{CPI} \times \text{cycle time}$$

$$= \text{instruction count} \times \text{CPI} \times \frac{1}{\text{clock rate}}$$

# Performance

---

- ❑ Performance is determined by execution time
- ❑ Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction (CPI)?
  - average # of instructions per second?
  
- ❑ Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

# CPI Example

---

- Suppose we have two implementations of the same instruction set architecture (ISA).

For some program P,

Machine A has a clock cycle time of 10 ns. and a CPI of 2.0

Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

What machine is faster for this program, and by how much?

$$\text{CPU time}_A = \text{IC} \times \text{CPI} \times \text{cycle time} = \text{IC} \times 2.0 \times 10\text{ns} = 20 \times \text{IC ns}$$

$$\text{CPU time}_B = \text{IC} \times 1.2 \times 20\text{ns} = 24 \times \text{IC ns}$$

So, A is 1.2 (=24/20) times faster than B

- If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?

# # of Instructions Example

---

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C  
The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

Which sequence will be faster? How much? (assume CPU starts execute the 2<sup>nd</sup> instruction after the 1<sup>st</sup> one completes)  
What is the CPI for each sequence?

$$\# \text{ of cycles}_1 = 2 \times 1 + 1 \times 2 + 2 \times 3 = 10$$

$$\# \text{ of cycles}_2 = 4 \times 1 + 1 \times 2 + 1 \times 3 = 9 \quad \text{So, sequence 2 is 1.1 times faster}$$

$$\text{CPI}_1 = 10 / 5 = 2$$

$$\text{CPI}_2 = 9 / 6 = 1.5$$

# MIPS Example

---

- ❑ Two different compilers are being tested for a 100 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.

The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

- ❑ Which sequence will be faster according to MIPS?
- ❑ Which sequence will be faster according to execution time?

# of instruction<sub>1</sub> = 5M + 1M + 1M = 7M, # of instruction<sub>2</sub> = 10M + 1M + 1M = 12M

# of cycles<sub>1</sub> = 5M × 1 + 1M × 2 + 1M × 3 = 10M cycles = 0.1 seconds

# of cycles<sub>2</sub> = 10M × 1 + 1M × 2 + 1M × 3 = 15M cycles = 0.15 seconds

So, MIPS<sub>1</sub> = 7M/0.1 = 70MIPS, MIPS<sub>2</sub> = 12M/0.15 = 80MIPS > MIPS<sub>1</sub>

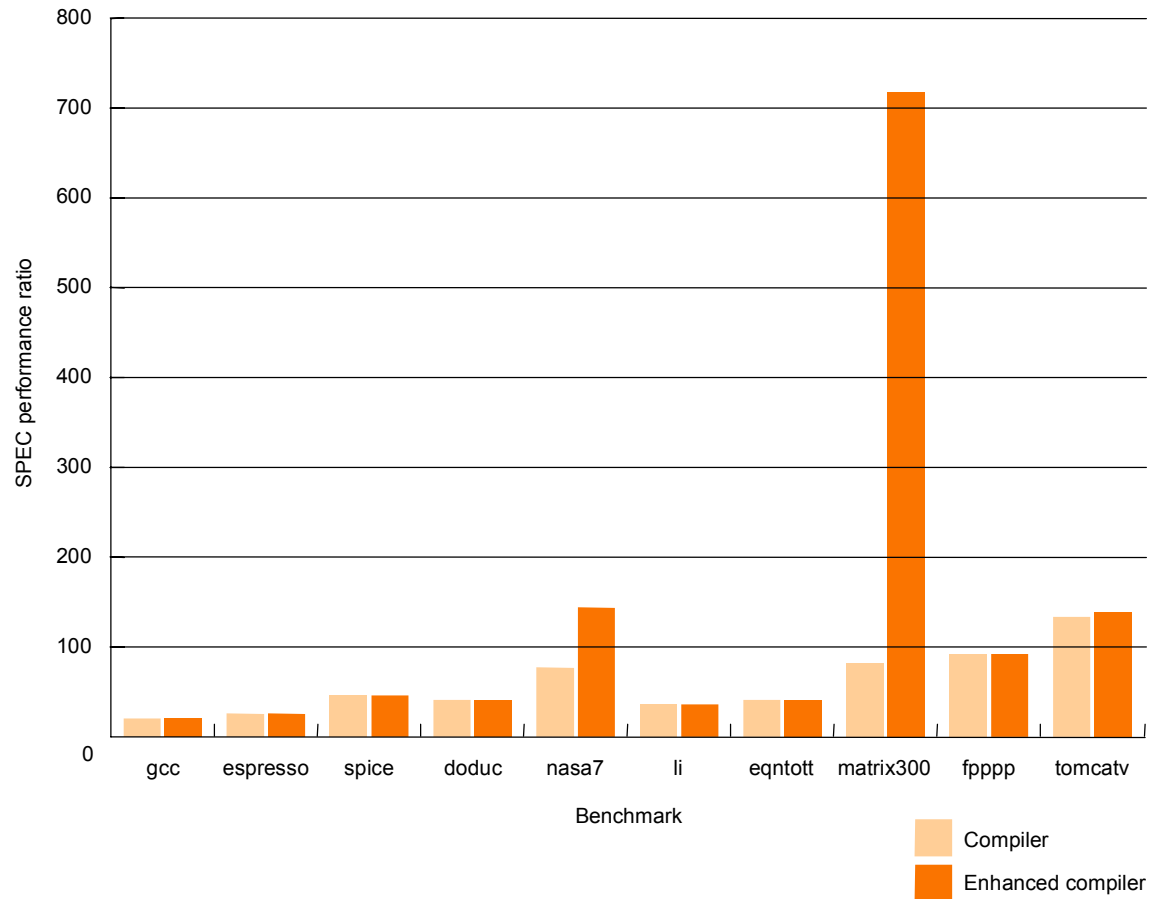
# Benchmarks

---

- ❑ **Performance best determined by running a real application**
  - Use programs typical of expected workload
  - Or, typical of expected class of applications  
e.g., compilers/editors, scientific applications, graphics, etc.
- ❑ **Small benchmarks**
  - nice for architects and designers
  - easy to standardize
  - can be abused
- ❑ **SPEC (System Performance Evaluation Cooperative)**
  - companies have agreed on a set of real program and inputs
  - valuable indicator of performance (and compiler technology)
  - can still be abused

# SPEC '89

## □ Compiler “enhancements” and performance



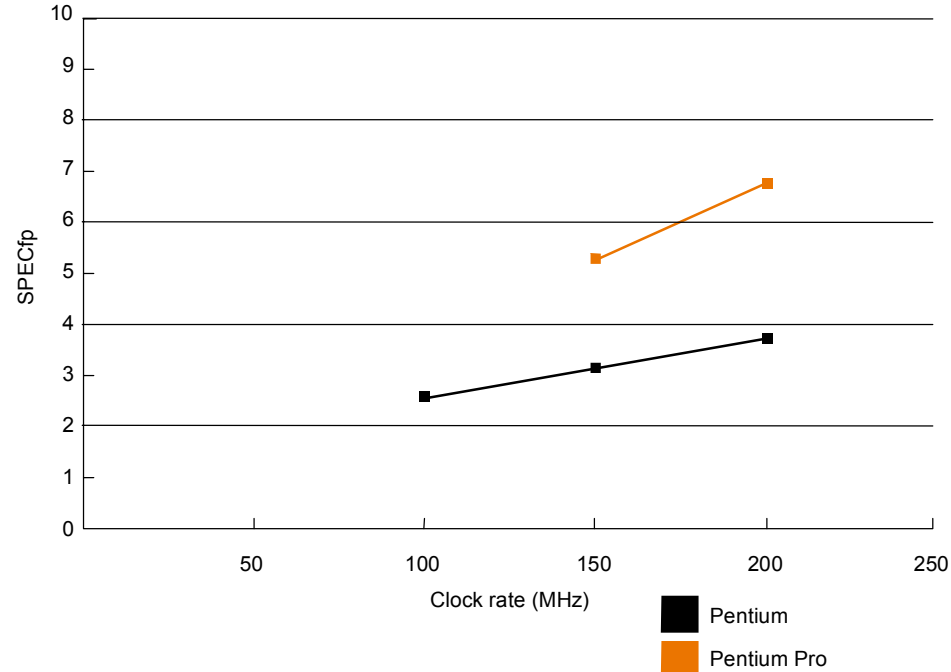
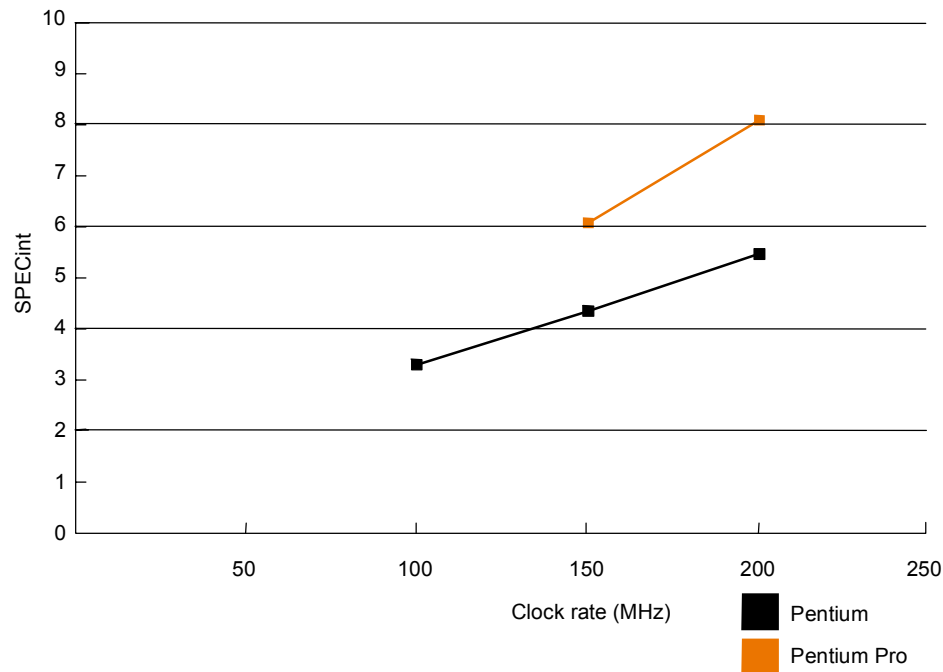
# SPEC '95

Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
jpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Navier Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

# SPEC '95

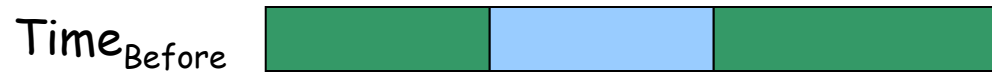
*Does doubling the clock rate double the performance?*

*Can a machine with a slower clock rate have better performance?*



# Amdahl's Law

**Execution Time After Improvement = Execution Time Unaffected + ( Execution Time Affected / Amount of Improvement )**



Execution time w/o E (Before)

Speedup (E) =

Execution time w E (After)

□ **Example:**

"Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

**How about making it 5 times faster?**

□ *Principle: Make the common case fast*

# Example

---

- Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

$$10/6$$

- We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

$$100-x+x/5 = 100/3, \quad x=83.3$$

# Remember

---

- ❑ **Performance is specific to a particular program/s**
  - **Total execution time is a consistent summary of performance**
- ❑ **For a given architecture performance increases come from:**
  - **increases in clock rate (without adverse CPI affects)**
  - **improvements in processor organization that lower CPI**
  - **compiler enhancements that lower CPI and/or instruction count**
- ❑ **Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance**
- ❑ **You should not always believe everything you read! Read carefully!**  
**(see newspaper articles, e.g., Exercise 2.37)**