Fundamentals of Computer Design

One of the exciting aspects of computer architecture and computer design is the rapid pace of development. Consider the IBM 7094 in 1965. This machine was one of the biggest and fastest machines of the 60’s. It featured interrupts, so the CPU could continue processing while I/O tasks executed. It also could add floating point numbers at a blazing 350,000 instructions per second. The standard system also contained 32K of fast core memory in 36 bit words and occupied an entire air conditioned room. This system cost around $3.5 million.

Compare that to the used Pentium II laptop I bought off EBay this summer for $450. At a clock speed of 300Mhz this machine can execute approximately 63 MFLOPS. (We’ll see later that these are not particularly good ways to benchmark systems), and came with 96Mb of RAM. It also weighs about 6 pounds. This is a massive increase in performance over the IBM 7094!

In accordance with Moore’s Law, computer performance has doubled about every 18 months. This has been due to several reasons:

- Architectural advances in the 70’s (e.g. interrupts)
- Integrated Circuits and VLSI in the 80’s and to today

The mainframe era suffered from binary compatibility problems – this was eventually addressed by higher level languages (FORTRAN, C). The virtual death of assembly language programs reduced the need for object-code compatibility. Second, the creation of standard OS’s like UNIX helped lower the cost and risk of a new architecture. Eventually we would develop new architectures like RISC which is popular today.

In the 80’s there was a huge debate over RISC vs. CISC. With the exception of Intel’s market dominance, RISC is probably the winner – the performance of RISC processors has improved dramatically and caters better to today’s ideas of using instruction level parallelism and superscalar processing. However, RISC is quite a bit more complicated than originally proposed and a key part of any modern system is how well your compiler can optimize code. An efficient compiler on a slow system may very well beat out an inefficient compiler on a fast system!

Task of a Computer Designer

A computer designer has many facets to take into account. Let’s look at some of these but first define a few terms:

Instruction Set Architecture – The assembly instructions visible to a programmer
Organization – High-level design aspects such as how much cache, replacement policies, cache mapping algorithm, bus structure, internal CPU design.

Computer Architecture – We’ll refer to this as instruction set design, organization of the hardware design, and the actual hardware itself.

In addition to these technical specs, a computer designer must also consider other factors, such as cost, marketing, and functional specifications. Often these may be driven by the market (witness the birth of the Celeron processor). Everything affects everything else!

Some examples:
- Apps sucked into OS
- Interconnect redesigned for new Apps (e.g. graphics bandwidth)
- Instruction Set & hardware designed for new Apps (e.g. MMX, MMX2)
- Power, CPU, ISA designed for mobile users (Transmeta)

The interactions among all these components form a huge constraint satisfaction problem, which is in the class of NP hard problems!

Usually the functional requirements for today’s machines are set by the company or the market. Today’s requirements are somewhat strange:

- To minimize cost implies a simple design
- To maximize performance implies complex design or sophisticated technology
- Time to market matters! Implies simple design and great secrecy
- Time to productivity! Implies need complete vertical solutions in place
- Don’t mess up – requires simulation, QA, quantification

Throughout this course we’ll focus mostly on the CPU, Memory, and Instruction Set. However, one should not lose sight of the bigger picture.
**Technology and Computer Usage Trends**

Related to the topic of considering all factors, technological and otherwise, into the design of a computer, a company must also consider and predict trends in computer use.

Consider IBM:

In 1943, Thomas Watson predicted “I think there is a world market for maybe five computers.” IBM continued to gamble on mainframes over minicomputers and analysts estimate this cost IBM $5 billion. (The IBM PC was an “undercover” project and saved the company)

In the 70’s and 80’s IBM pursued the high-speed Josephson Junction, spending $2 billion, before scrapping it and using CMOS like everyone else.

So let’s look at some current trends in computer systems today:

- Memory use by programs growing by a factor of 1.5 to 2 per year. Lesson: don’t make your word size or address space too small!

- Replacement of assembly by HLL’s. Lesson: compilers are critical!

- Growing demand on multimedia. Lesson: Design for high-speed graphic interfaces.

- Growing demand for networking. Lesson: Also design for high-speed I/O! Gigabit networking to the desktop?

- Growing demand for simplicity in attaching I/O devices. Rise of USB, perhaps Firewire next?

A few trends in implementation technology:

- IC density is increasing at close to 50% per year, while the die size is also increasing at close to 15% per year. This results in transistor count increasing at 60-80% per year. Factors into a lower future cost for developing the same chip!

- Semiconductor DRAM density increasing around 60% per year, bandwidth and latency also improving.

- Disk technology increasing at an astounding rate, about 50% per year. In 7/2001, Maxtor released a 100Gb hard drive for just $300. IBM announced innovations that will allow them to reach 400Gb drives. Access time has also increased, although slower than drive density (about 33% in 10 years).
Cost Trends

While all of these trends are good for the consumer in the form of lower costs, it places additional burdens on the computer designer. To design a system today with the rapid pace of technology, one actually has to design a system with the idea of implementing technology that may not even exist, at least not commercially! This has propagated all the way to software, where teams will design an operating system for a hardware platform that exists only in specifications.

Nevertheless, cost trends have become one of the major themes in the computer industry today. This is often ignored in computer science, but is essential for designers to make intelligent decisions about whether or not a new feature should be included. To introduce just a few of these factors, let’s look at time issues, wafer size, and cost distribution.

Time issues:

- The **learning curve** drives down costs over time as one learns how to better implement and manufacture technology.

- The learning curve is typically measured by **yield**. The yield is the percentage of manufactured chips that actually work (i.e. they don’t contain some kind of defect). You might be surprised to learn that a significant number of manufactured chips must simply be thrown away – initial estimates on the yield for the AMD K6 chip were as low as 20%. The yield improves over time for a given chip process, and can be used as a metric for where one stands in the learning curve.

- Other factors include volume and commodities. Increased volume can reduce the learning curve and decrease cost due to increased purchasing and manufacturing efficiency. Commodities are products sold by multiple vendors in large volume. The low-end PC business has become essentially a commodity business. Product cost is lowered as competition and volume components become cheaper.

Cost of an Integrated Circuit:

- While the cost of an IC has dropped exponentially, the basic procedure of silicon manufacturing is unchanged. A wafer is tested and chopped into dies that are packaged. The cost of a packaged IC is:

\[
IC\_Cost = \frac{Cost\_Die + Cost\_Test + Cost\_Packaging}{Yield}
\]

We can compute the cost of a die by:
The maximum number of dies per wafer is basically the area of the wafer divided by the area of the die. We must subtract off a correction factor since dies are currently square, while wafers are round:

\[
\text{Dies per Wafer} = \frac{\pi (\text{Wafer Radius})^2}{\text{Die Area}} - \frac{\pi (\text{Wafer Diameter})}{\sqrt{2(\text{Die Area})}} - \text{Num Test Dies}
\]

The maximum number of dies per wafer is unrealistic today due to imperfections in the manufacturing process. A more realistic value is the die yield which can be expressed as:

\[
\text{Die Yield} = \text{Wafer Yield} \times \left(1 + \frac{\text{Defects per unit area} \times \text{Die Area}}{\alpha} \right)^{-\alpha}
\]

Wafer Yield refers to wafers that are entirely bad and need not be tested. Defects per unit area typically ranged from 0.6 to 1.2 per square centimeter. Alpha is a parameter that corresponds to the number of masking levels (a measure of complexity). Today's processes have an alpha from 3 to 6.

FYI, here are the sizes of some dies of typical processors:

- .25u K6- 2 = 68 mm²
- .25u K6- 3 = 135 mm²
- .25u Pentium II = 105 mm²
- .25u PPC 604e = 47.3 mm²
- .18u Pentium 4: 217 mm²
- .18u Transmeta TM5600 88 mm²
- .13u Transmeta TM5800 25? mm²

Implications of a larger die size?

The diameter of wafers has increased from 200mm (8 inch) in 1993 to 300mm today (12 inch).

The book has some examples of computing the die yield for various wafer and die sizes. However, for a designer, the wafer yield, cost, and defects are all determined by the manufacturing process. The designer has control over the die area. If alpha is typically 3, then this means that the die_yield is proportional to \((\text{Die Area})^3\). Plugging this in to the equation for die cost gives us:

\[
\text{Die Cost} = f(\text{Die Area}^4)
\]

; Size matters!
• Testing Costs are another important part of the cost for an IC. Lots of tests are available, ranging from a visual test (for an obvious mistake), to parametric tests of the process itself, to end-product tests using known inputs and desired outputs. Testing costs have been increasing over time as chips become more complex.

• Packaging is another cost which can drive up the final price of the IC. The materials and the number of external pins are the largest factors in the packaging. An example of expensive packaging is the Slot-Based Pentium II chip, although it had the added advantage of desirable thermal properties and expandability options.

Finally, the distribution of cost in an entire system needs to be considered. While the CPU was once the most expensive component, today’s CPU’s are now less than 10% of the overall system cost for typical desktop computers. Power, processor boards, I/O devices, and components such as the memory, monitor and hard drives are typically the most expensive today. The configuration of all these components can have a significant impact on the performance of the complete system.

Measuring Performance

What does it mean to say one computer is faster than another? Advertisers are currently using Mhz, or the clock speed, as the benchmark for what CPU is better. If one processor has a clock rate of 1000 Mhz vs. a different processor with a clock rate of 800 Mhz, it appears on the surface that the first processor is better than the second to the tune of 200 Mhz. However, this logic only holds true within a given type of processor as different types of processors are capable of doing different amounts of work within a cycle. For example, a 733 Mhz G4 processor performs about the same amount of work in the same amount of time as a 1.5Ghz Pentium IV (on certain tasks, anyway). To achieve the high clock speeds, Intel had to increase the pipeline to 20 stages on the Pentium 4 – (interestingly the Itanium has a lower clock speed but higher ILP). Moreover, other components can have an even larger impact on system performance such as the amount of memory, hard drive speed, system buses, etc.

In fact, given the choice of two equivalent processors but with different clock speeds, most designers will choose the device with the lower clock speed:

• Less crosstalk, electromagnetic interference, potential for error at lower clock speeds
• Less heat
• Less power consumption

Unfortunately, typical consumers today tend to make purchases based upon the clock speed of a computer system.

A better metric is to use execution time. This is simply the time required to complete some task. In comparing two machines, X, and Y, we can then compute:
\[ n = \frac{\text{Execution Time}(Y)}{\text{Execution Time}(X)} \]

This gives us a ratio of the performance of Y vs. X on the particular task. Another metric sometimes used is performance, which is just the reciprocal of execution time:

\[ \text{Performance}(X) = \frac{1}{\text{Execution Time}(X)} \]

Unfortunately it is somewhat difficult to precisely measure execution time. We can break time up into four categories:

- **Wall-clock time, response time, or elapsed time** - Total time to complete a task. Note the system might be waiting on I/O, performing multiple tasks, OS activities, etc.
- **CPU time** – Only the time spent computing in the CPU, not waiting for I/O. This includes time spent in the OS and time spent in user processes.
- **System CPU Time** – Portion of CPU time spent performing OS related tasks.
- **User CPU Time** – Portion of CPU time spent performing user related tasks.

The unix “time” function computes user, system, and elapsed time. However, it is notorious for under-measuring actual time.

The next problem that arises is the issue of **what programs should we run** as benchmarks for the timing runs? We have several options:

- **Real Programs** – Run the programs users will use. Examples are compilers, office tools, etc. Unfortunately there is a porting problem among different architectures, so it might not be a fair comparison if the same software is coded differently.

- **Kernels** – These are small, intensive, key pieces from real programs that are run repeatedly to evaluate performance. Examples include Livermore Loops and Linpack. Here is a code fragment from Livermore Loops:

```c
for (l=1; l<=loop; l++) {
    for (k=0; k<n; k++) {
        x[k] = q + y[k] * (r*z[k+10]+t*z[k+11]);
    }
}
```

A big problem of these kernels is they’re not real programs, so no user really uses them and therefore it may be misleading in actual tasks.

- **Toy benchmarks** --. This includes small problems like quicksort, sieve of eratosthenes, etc. They are best saved for CS201 programming assignments!
• Synthetic benchmarks - These are similar to kernels, but try to match the average frequency of operations and operands of a large set or programs. Whetstone and Dhrystone are popular here. The problem is no user really runs these either. Programs typically reside entirely in cache and don’t test the entire system performance!

• Benchmark Suites - These are a collection of benchmarks together in an attempt to measure the performance of processors with a variety of applications. Once again, this suffers from problems of OS support, compiler quality, system components, etc. Nevertheless, suites such as the SPEC benchmark seem to be required for the industry today, even if the results may be somewhat meaningless.

Here are some programs in the SPECint92 benchmark, intended to measure integer performance (there is now a SPEC2000 benchmark).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Lines of Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso</td>
<td>C</td>
<td>13500</td>
<td>Minimize Boolean functions.</td>
</tr>
<tr>
<td>Li</td>
<td>C</td>
<td>7413</td>
<td>Lisp interpreter that solves 8 queens problem</td>
</tr>
<tr>
<td>Compress</td>
<td>C</td>
<td>1503</td>
<td>LZ compression on a 1Mb file</td>
</tr>
<tr>
<td>Gcc</td>
<td>C</td>
<td>83589</td>
<td>GNU C Compiler</td>
</tr>
</tbody>
</table>

Here are some programs in the SPECfp92 benchmark suite, intended to measure floating point performance:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Lines of Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spice2g6</td>
<td>FORTRAN</td>
<td>18476</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>Alvinn</td>
<td>C</td>
<td>272</td>
<td>Neural network training simulation</td>
</tr>
<tr>
<td>Ear</td>
<td>C</td>
<td>4483</td>
<td>Inner ear model that filters and detects various sounds</td>
</tr>
<tr>
<td>Su2cor</td>
<td>FORTRAN</td>
<td>2514</td>
<td>Compute masses of elementary particles from Quark-Gluon theory</td>
</tr>
</tbody>
</table>

How many of these algorithms do you know? Do you actually run these types of things yourself? (maybe, probably not)

Beware of reports on benchmarks! Companies may even hand-code optimizations so that particular programs execute faster. For example, certain flags during compilation can
have a huge effect on final execution time for some of these. The Whetstone loop contains the following expression:

\[
\text{SQRT} (\text{EXP}(X))
\]

A brief analysis yields:

\[
\sqrt{e^x} = e^{x/2} = \text{EXP}(X/2)
\]

It would be surprising to see such an optimization automatically performed by a compiler due to the expected rarity of encountering SQRT(EXP(X)). Nevertheless, several compilers did perform this optimization! A program compiled this way would likely run faster on the Whetstone loop than a program that was not compiled this way.

How might you do benchmarks? What factors should be included?

**Comparing and Summarizing Performance**

Some other problems in comparing performance of systems and programs is shown in the following table. The table lists the time in seconds that each computer requires to run a particular program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>P2</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>1001</td>
<td>110</td>
<td>40</td>
</tr>
</tbody>
</table>

Which computer is better? For P1, A is 10 times faster than B, but for P2 B is 10 times faster than A.

**Total Execution Time: A consistent summary measure**

The simplest approach to summarize relative performance is the total execution time of the two programs. So then B is 9.1 times faster than A (1001/110), while C is 2.75 times faster than B. Of course this is all relative to the programs that are selected.

Some other simple aggregate metrics are:

Arithmetic Mean: \[ \frac{1}{n} \sum_{i=1}^{n} \text{Time}_i \]

Harmonic Mean: \[ \frac{n}{\sum_{i=1}^{n} \frac{1}{\text{Rate}_i}} \] where Rate = 1 / \text{Time}_i, execution time for ith of n progs
We can also apply weights to separate programs, if one program is more important or is more heavily used than another:

**Weighted Arithmetic Mean:** \[ \frac{\sum_{i=1}^{n} Weight_i \times Time_i}{n} \]

**Weighted Harmonic Mean:** \[ \frac{1}{\sum_{i=1}^{n} \frac{Weight_i}{Rate_i}} \]

The weighted means can be better, but one has to be careful about the weights as it can force one program to become dominant.

**Normalized Execution Time**

A second approach is to normalize the execution time with respect to some reference machine. We can then compute the geometric mean:

**Geometric Mean:** \[ \left( \prod_{i=1}^{n} ExecutionTimeRatio_i \right)^{\frac{1}{n}} \]

For example, given the data in the previous table for Computer A, B, and C, we can compute the geometric mean referenced to A as:

For A:

Geometric Mean: \( \left( \frac{1 \times 1000}{1 \times 1000} \right)^{\frac{1}{2}} = 1 \)

For B:

Geometric Mean: \( \left( \frac{10 \times 100}{1 \times 1000} \right)^{\frac{1}{2}} = 1 \)

For C:

Geometric Mean: \( \left( \frac{20 \times 20}{1 \times 1000} \right)^{\frac{1}{2}} = 0.63 \)

This says that the execution time of these programs on C is 0.63 of A and B.

As a side note, the geometric mean has the nice property that the ratio of the means = the means of the ratios.
Drawbacks to using geometric means include:

- Don’t predict actual execution time
- Danger of designers working to do better on benchmarks, not overall, e.g. a 2 to 1 second improvement will reward the system as much as a 500 to 250 second improvement

Ideal is to use real workloads wherever possible!

**Quantitative Principles of Computer Design**

Now that we have looked at ways to evaluate performance, let’s look at ways to increase performance.

The most pervasive principle is simply, **make the common case go fast**. If something is going to be done often, make it run quick! To do this, one must;

- Validate that a case is common or uncommon
- Tasks done in hardware → fast, Tasks done in software → slow
  - But… Hardware that isn’t used still incurs cost (how?)
- The nice thing is that common cases are typically simpler than uncommon cases
  - Overflows, interrupts
  - Truly simple cases are both cheap and fast

To help quantify this principle, we can refer to **Amdahl’s Law**.

**Amdahl’s Law**: The performance improvement to be gained from using some faster mode of execution is limited by the fraction of time the faster mode can be used.

This can be gained by defining **speedup**:

\[
\text{Speedup} = \frac{\text{Performance for entire task using enhancement}}{\text{Performance for entire task without enhancement}}
\]

Another variant is based on the ratio of Execution times, where

\[
\text{Execution time} = \frac{1}{\text{Speedup}}
\]

The overall speedup is the ratio of execution times:

\[
\text{Overall Speedup} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]

Fraction(enhanced) is just the fraction of time the enhancement can be used.
For example, if a new enhancement is 20 times faster but can only be used 5% of the time, then:

\[
\text{Overall Speedup} = \frac{1}{(1 - 0.05) + \frac{0.05}{20}} = 1.049 \quad \text{(not much speedup)}
\]

If this new enhancement is expensive, it is probably not worth the effort unless every last bit of speed is critical, or there is some application that can use this enhancement more often. This illustrates the law of diminishing returns: The incremental improvement in speedup gained by an additional improvement of just a portion of the computation diminishes as improvements are added.

As another example, consider a critical benchmark. Floating Point Square Root is responsible for 20% of the execution time. You could increase this operation by a factor of 10 via hardware. Or at the same cost, you could make all FP instructions run 2 times faster, which accounts for 50% of the execution time. Which is better?

(Plug in the numbers to the equation to see)

**CPU Performance**

All commercial computers are synchronous – they have a clock that “ticks” once per cycle. Usually the cycle time is a constant, but it may vary (e.g. SpeedStep). The duration is usually on the order of nanoseconds, or more commonly the rate is expressed instead, e.g. 100 Mhz. CPU time for a program can then be expressed two ways:

- CPU Time = CPU clock cycles for a program * Clock cycle time
- CPU Time = CPU clock cycles for a program / Clock Rate

Additionally, we can count the number of instructions executed – the instruction path length or Instruction Count (IC). Given IC and the number of clock cycles we can calculate the average number of clock cycles per instruction, or CPI:

\[
\text{CPI (Ave # clock cycles per instr)} = \frac{\text{CPU clock cycles for a program}}{\text{IC}}
\]

We will use CPI throughout this course!

With a little algebraic manipulation we can use CPI to compute CPU Time:

- CPU clock cycles for a program = CPI * IC
  - Substitute this in...

\[
\text{CPU Time} = \frac{\text{CPI} \times \text{IC} \times \text{Clock Cycle Time}}{	ext{Clock Rate}}
\]

Or

\[
\text{CPU Time} = \frac{\text{CPI} \times \text{IC}}{	ext{Clock Rate}}
\]
Unfortunately, CPI, IC, and the Clock Cycle Time are all subtly inter-related. The CPI and cycle time depend on the instruction set, the instruction set depends on the hardware, the hardware impacts the cycle time, etc.

Sometimes CPI is more useful to deal with in terms of all individual instructions. We can denote this by:

\[
\text{CPU Time} = \left( \sum_{i=1}^{n} CPI_i \times IC_i \right) \times \text{ClockCycleTime} \\
\text{And} \\
\text{CPI Total} = \left( \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{\text{InstructionCount}} \right)
\]

Here, \( i \) varies over each instruction, and we must sum the results for each instruction to get the total CPU Time. CPI\(_i\) should be measured since reference manuals won’t include the effects of cache misses, branch predictions, etc.

Example:

CPI is useful because it is measurable, unlike the nebulous “fraction of execution” in Amdahl’s equation. Consider the previous example to compute the speedup of Floating Point Square Root:

Suppose you run your program and as it runs collect the following data:

- 100 total instructions
- 25 of these instructions are floating point instructions
- 2 of these instructions are floating point square root
- 100 clock cycles were spent on floating point instructions
  - 40 of these 100 cycles were spent on floating point square root
- 100 clock cycles were spent on non-floating point instructions

We can calculate:

- Frequency of FP operations = \( \frac{25}{100} = 0.25 \)
- Frequency of FPSQ operations = \( \frac{2}{100} = 0.02 \)
- Ave CPI of FP operations = \( \frac{100}{25} = 4 \)
- Ave CPI of FPSQ operations = \( \frac{40}{2} = 20 \)
- Ave CPI of non-FP operations = \( \frac{100}{75} = 1.33 \)

If we could reduce the CPI of FPSQ by 10 (down to 2), or reduce the CPI of all FP operations by 2, which is better?

Answer:
First calculate the original CPI without any enhancements. Then we’ll use that to 
calculate the CPU Time, and do the same thing for both enhancements. Finally we can 
use Amdahl’s Law to determine the speedup.

\[
\text{CPI}_{\text{Original}} = \frac{\text{CPU Cycles for the program}}{\text{IC}}
\]

\[= \frac{200}{100} = 2.0\]

We can compute the CPI with the new FPSQ operation by subtracting off the cycles 
saved from the original CPI:

\[
\text{CPI}_{\text{with new FPSQ}} = \text{CPI}_{\text{Original}} - 0.02(\text{CPI}_{\text{Old FPSQ}} - \text{CPI}_{\text{new FPSQ only}})
\]

\[= 2.0 - 0.02(20-2)\]

\[= 1.65\]

Similarly, we can compute the CPI with new FP operations all around by subtracting off 
the cycles saved from the original CPI:

\[
\text{CPI}_{\text{with new FP}} = \text{CPI}_{\text{Original}} - 0.25(\text{CPI}_{\text{Old FP}} - \text{CPI}_{\text{new FP only}})
\]

\[= 2.0 - 0.25 (4 - 2)\]

\[= 1.5\]

Since \(\text{CPI}_{\text{with new FP}}\) is lower, it is better than \(\text{CPI}_{\text{with new FPSQ}}\). We can plug this into 
Amdahl’s law to see the speedup:

\[
\text{Speedup (new FP)} = \frac{\text{CPU Time (original)}}{\text{CPU Time (new FP)}}
\]

\[= \frac{\text{IC} \times \text{Clock Cycles} \times \text{CPI}_{\text{Original}}}{\text{IC} \times \text{Clock Cycles} \times \text{CPI}_{\text{with new FP}}}
\]

\[= \frac{\text{CPI}_{\text{Original}}}{\text{CPI}_{\text{with new FP}}}
\]

\[= \frac{2.0}{1.5}
\]

\[= 1.33\]

This should match the same result from the earlier exercise, but we started from empirical 
measurements rather than given a speedup of 10.

**Measuring the Components of CPU Performance**

To perform the analysis we just did, we need to measure cycle time, IC, and CPI:

- Cycle Time is easy to measure for an existing CPU (whatever frequency it is 
  running at).
- Cycle Time is hard to measure for a CPU in design! The logic design, VLSI 
  process, simulations, and timing analysis need to be done.
• IC – This is one thing that is relatively easy to measure, just count up the instructions. This can be done with instruction trace, logging, or simulation tools.

• CPI – This can be difficult to measure exactly because it depends on the processor organization as well as the instruction stream. Pipelining and caching will affect the CPI, but it is possible to simulate the system in design and estimate the CPI.

The chart below illustrates the effect of caching and temporal locality of reference on 10 programs in the SPEC92 benchmark suite. The green instructions account for 90% of the instruction executions, while the red accounts for 80%. So for example, in the ear program about 5% of the instructions account for 80% of the execution, and 9% account for 90%. This means about 91% of the code only accounts for 10% of the instruction executions.

**10-90 Observations on SPEC92 Programs**

![Diagram showing the 10-90 observations on SPEC92 programs.](image)

**Comments on the Memory Hierarchy**

We’ll only briefly touch on caching and the memory hierarchy; you should already be familiar with this from CS221. The basic principle behind the memory hierarchy is that **smaller is faster**. Of course, users want lots of memory available, but large memories have longer signal propagation delays and slower decoding time. Conversely, small memories are fast due to short propagation and more power can be used per memory cell in a small design but with a large cost.
The solution is a hierarchy:

- A tiny amount of very fast memory (registers), say 100 bytes
- A small amount of fast memory below that (cache), say 64K
- Perhaps a second cache, slightly larger but slower, say 256K
- Main memory, say 64MB
- I/O Devices, e.g. disk, network (Gigabytes)

Fortunately locality of reference helps us; this behavior says that programs tend to access similar data repeatedly and we also tend to access data contiguously in memory. This means we should be able to access data quickly most of the time at the top of the hierarchy and rarely need to venture to the slow, bottom part of the hierarchy.

We’ll review caches, cache hits, cache misses, performance evaluation, and the relationship with virtual memory in more detail later when we get to Chapter 5.

**Common Fallacies and Pitfalls**

We’ve already talked about perhaps the biggest fallacy, the Myth of Megahertz, along with some of the problems that come with benchmark suites (e.g., tuning a system to the benchmark, finding good benchmarks). A similar myth is the usefulness of the metric of MIPS.

MIPS = Millions of Instructions Per Second

\[
\text{MIPS} = \frac{\text{Instruction \_Count}}{\text{Execution\_Time} \times 10^6} = \frac{\text{Clock \_Rate}}{\text{CPI} \times 10^6}
\]

While MIPS is easy to understand and sounds impressive, it is not a good metric by itself. First, it has the same problems of all benchmarks - what program is executing and being timed in this fashion. Also consider the following example:

Floating Point instructions are more complex than Integer instructions and generally take longer to execute. Let’s say a multiply FP instruction requires 4 clock cycles. However, instead of executing the FP multiply instruction, we could instead use a software floating point routine that used only Integer instructions. Since the integer instructions are simpler, they will require fewer clock cycles. Let’s say for the purpose of simplicity that each integer instruction requires 2 clock cycles and it takes 20 of them to implement the FP multiply. Then for a 1 Mhz machine:

FP Multiply has a CPI of 4
MIPS = 1 / 4 = 0.25

The software FP Multiply using integer instructions has a CPI of 2
MIPS = 1 / 2 = 0.50
The software version has higher MIPS! This is because we can run two integer instructions in the time it takes to do one floating point instruction. However, lost in this analysis is the fact that we need to execute many more integer instructions to do the same amount of logic that the single floating point instruction can do, so the software version is actually much slower.

A similar fallacy to MIPS is that of MFLOPS. MFLOPS = Millions of Floating Point Operations per Second. This differs from MIPS in that MFLOPS is interested in operations (e.g. multiply, divide) instead of raw instructions. It becomes difficult to apply across architectures when one machine implements an operation in hardware while another does not (e.g., tanh is implemented). Additionally, the MFLOPS rating changes on mixtures of fast and slow floating-point operations – a program with many floating point adds will have a higher rating than one with many floating point divides.

Finally, there is the fallacy that peak performance tracks observed performance. Peak performance is the maximum performance possible, while observed performance is actual performance.

As an example the Alpha processor in 1994 was announced as being capable of executing 1.2 billion instructions per second at its 300 Mhz clock rate.

How can this be? The answer lies in the superscalar ideas – executing multiple instructions at once. To achieve this feat, the processor would have to execute two integer instructions and two floating point instructions on each clock cycle. This is virtually impossible except for very specific cases! Needless to say, the SPEC numbers for the processor were not very close to the peak performance (although very good compared to its competition at the time).

The peak performance terms are primarily seen in reference to the supercomputer domain.