# Table of Contents

1. Computer Architecture, Chapter 1  
2. Growth in Performance  
3. Growth in Clock Rate  
4. Mystery Graph  
5. Energy and Power  
6. Energy and Power Example  
7. Microprocessor Chip Example  
8. Chip Floorplan  
9. Silicon Boule  
10. Wafer Example  
11. Dies Per Wafer Example  
12. Cost  
13. Yield  
14. Yield Models - Poisson  
15. Yield Models - Binomial  
16. Yield Models - Bose-Einstein  
17. Dependability  
18. MTTF Example  
19. MTTR Example  
20. Performance Measurements  
21. Summarizing Performance  
22. Average vs. Relative Throughput  
23. How to Mislead with Performance Reports  
24. Speedup and Amdahl's Law  
25. Reliability Amdahl Example  
26. Processor Performance Equation  
27. CPI  
28. CPI Example  
29. CPI Example, Speedup  
30. Instruction Set Architecture - Registers  
31. Instruction Set Architecture - Formats  
32. Instruction Set Architecture - Operands  
33. Instruction Set Architecture - Addressing Modes
1. Computer Architecture, Chapter 1

Fundamentals of Quantitative Design and Analysis

Computer Architecture, A Quantitative Approach, Fifth Edition,
John L. Hennessy and David A. Patterson, 2011.
2. Growth in Performance

Figure 1.1 Growth in processor performance since the late 1970s. This chart plots performance relative to the VAX.
3. Growth in Clock Rate

Figure 1.11 Growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 25% per year. During the “renaissance period” of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 1% per year, while single processor performance improved at less than 22% per year.
4. Mystery Graph
5. Energy and Power

For CMOS chips, the traditional primary energy consumption has been in switching transistors, also called *dynamic energy*. The energy required per transistor is proportional to the product of the capacitive load driven by the transistor and the square of the voltage:

\[ \text{Energy}_{\text{dynamic}} \propto \text{Capacitive load} \times \text{Voltage}^2 \]

This equation is the energy of pulse of the logic transition of 0→1→0 or 1→0→1. The energy of a single transition (0→1 or 1→0) is then:

\[ \text{Energy}_{\text{dynamic}} \propto \frac{1}{2} \times \text{Capacitive load} \times \text{Voltage}^2 \]

The power required per transistor is just the product of the energy of a transition multiplied by the frequency of transitions:

\[ \text{Power}_{\text{dynamic}} \propto \frac{1}{2} \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched} \]

For a fixed task, slowing clock rate reduces power, but not energy.

Clearly, dynamic power and energy are greatly reduced by lowering the voltage, so voltages have dropped from 5V to just under 1V in 20 years. The capacitive load is a function of the number of transistors connected to an output and the technology, which determines the capacitance of the wires and the transistors.
6. Energy and Power Example

Some microprocessors today are designed to have adjustable voltage, so a 15% reduction in voltage may result in a 15% reduction in frequency. What would be the impact on dynamic energy and on dynamic power?

Since the capacitance is unchanged, the answer for energy is the ratio of the voltages

\[
\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2}{\text{Voltage}^2} = 0.85^2 = 0.72
\]

thereby reducing energy to about 72% of the original. For power, we add the ratio of the frequencies

\[
\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = 0.72 \times \frac{(\text{Frequency switched} \times 0.85)}{\text{Frequency switched}} = 0.61
\]

shrinking power to about 61% of the original.
7. Microprocessor Chip Example

Figure 1.13 Photograph of an Intel Core i7 microprocessor die, which is evaluated in Chapters 2 through 5. The dimensions are 18.9 mm by 13.6 mm (257 mm$^2$) in a 45 nm process. (Courtesy Intel.)
8. Chip Floorplan

Figure 1.14 Floorplan of Core i7 die in Figure 1.13 on left with close-up of floorplan of second core on right.
9. Silicon Boule

Wikipedia: A boule is a single crystal ingot produced by synthetic means which results in a cylindrical rod of material.
10. Wafer Example

Figure 1.15 This 300 mm wafer contains 280 full Sandy Bridge dies, each 20.7 by 10.5 mm in a 32 nm process. (Sandy Bridge is Intel’s successor to Nehalem used in the Core i7.) At 216 mm², the formula for dies per wafer estimates 282. (Courtesy Intel.)

The number of dies per wafer is approximately the area of the wafer divided by the area of the die. It can be more accurately estimated by

\[
\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} = \frac{\pi \times \text{Wafer diameter}}{\sqrt{2} \times \text{Die area}}
\]
11. Dies Per Wafer Example

**Example**  Find the number of dies per 300 mm (30 cm) wafer for a die that is 1.5 cm on a side and for a die that is 1.0 cm on a side.

**Answer**  When die area is 2.25 cm²:

\[
\text{Dies per wafer} = \frac{\pi \times (\frac{30}{2})^2}{2.25} - \frac{\pi \times 30}{\sqrt{2} \times 2.25} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270
\]

Since the area of the larger die is 2.25 times bigger, there are roughly 2.25 as many smaller dies per wafer:

\[
\text{Dies per wafer} = \frac{\pi \times (\frac{30}{2})^2}{1.00} - \frac{\pi \times 30}{\sqrt{2} \times 1.00} = \frac{706.9}{1.00} - \frac{94.2}{1.41} = 640
\]
12. Cost

\[
\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}
\]

\[
\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}
\]
13. Yield

\[ \text{Die yield} = \frac{\text{Wafer yield}}{(1 + \text{Defects per unit area} \times \text{Die area})^N} \]

This Bose–Einstein formula is an empirical model developed by looking at the yield of many manufacturing lines [Sydow 2006]. \textit{Wafer yield} accounts for wafers that are completely bad and so need not be tested. For simplicity, we’ll just assume the wafer yield is 100%. Defects per unit area is a measure of the random manufacturing defects that occur. In 2010, the value was typically 0.1 to 0.3 defects per square inch, or 0.016 to 0.057 defects per square centimeter, for a 40 nm process, as it depends on the maturity of the process (recall the learning curve, mentioned earlier). Finally, \( N \) is a parameter called the process-complexity factor, a measure of manufacturing difficulty. For 40 nm processes in 2010, \( N \) ranged from 11.5 to 15.5.

---

**Example**

Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.031 per cm\(^2\) and \( N \) is 13.5.

**Answer**

The total die areas are 2.25 cm\(^2\) and 1.00 cm\(^2\). For the larger die, the yield is

\[ \text{Die yield} = \frac{1}{(1 + 0.031 \times 2.25)^{13.5}} = 0.40 \]

For the smaller die, the yield is

\[ \text{Die yield} = \frac{1}{(1 + 0.031 \times 1.00)^{13.5}} = 0.66 \]

That is, less than half of all the large dies are good but two-thirds of the small dies are good.
14. Yield Models - Poisson

Probability of \( k \) defects assuming an average of \( \lambda \) defects per die:

\[
P(k) = \frac{e^{-\lambda} \lambda^k}{k!}
\]

\( \lambda = D_0A, \quad D_0 = \text{average defect density, e.g. defects per cm}^2, \quad A = \text{die area} \)

Yield = \( P(0) = e^{-D_0A} \)

If \( D_0 = D_1 + D_2 + ... \), then Yield = \( e^{-D_1A} e^{-D_2A} ... \)

Reference: Yield Modeling and Analysis, Robert C. Leachman, 2014: [source](source), [local](local)
15. Yield Models - Binomial

Probability that \( k \) out of \( n \) total defects are on a particular die:

\[
P(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}
\]

\( n = D_0 A_w \), \( A_w = \) wafer area, \( p = A/A_w \)

Yield = \( P(0) = (1 - A/A_w)^{D_0 A_w} \)

If \( A_w >> A \), then Yield \( \approx e^{-D_0 A} \)
16. Yield Models - Bose-Einstein

Variable defect density, with probability density $f(D)$:

\[
\text{Yield} = \int_0^\infty e^{-DA}f(D)dD
\]

If $f(D) = \frac{1}{D_0^2}e^{-\frac{D}{D_0}}$ (exponential distribution), then

\[
\text{Yield} = \frac{1}{1 + AD_0}
\]

This Seeds model, applied to $N$ critical mask layers, leads to the Bose-Einstein model:

\[
\text{Yield} = \left(\frac{1}{1 + AD_0}\right)^N
\]

Note that the Bose-Einstein defect density $D_0$ is per-layer, so it is not the same value as in the Poisson and Binomial models.
17. Dependability

- *Module reliability* is a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant. Hence, the *mean time to failure* (MTTF) is a reliability measure. The reciprocal of MTTF is a rate of failures, generally reported as failures per billion hours of operation, or *FIT* (for *failures in time*). Thus, an MTTF of 1,000,000 hours equals $10^9/10^6$ or 1000 FIT. Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. If a collection of modules has exponentially distributed lifetimes—meaning that the age of a module is not important in probability of failure—the overall failure rate of the collection is the sum of the failure rates of the modules.

- *Module availability* is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption. For nonredundant systems with repair, module availability is

\[
\text{Module availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}
\]
18. MTTF Example

Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 ATA controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 ATA cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

The sum of the failure rates is

\[
\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1,000,000} + \frac{1}{500,000} + \frac{1}{200,000} + \frac{1}{200,000} + \frac{1}{1,000,000}
\]

\[
= \frac{10 + 2 + 5 + 5 + 1}{1,000,000 \text{ hours}} = \frac{23}{1,000,000} = \frac{23,000}{1,000,000,000 \text{ hours}}
\]

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate:

\[
\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1,000,000,000 \text{ hours}}{23,000} = 43,500 \text{ hours}
\]

or just under 5 years.
19. MTTR Example

Disk subsystems often have redundant power supplies to improve dependability. Using the components and MTTFs from above, calculate the reliability of redundant power supplies. Assume one power supply is sufficient to run the disk subsystem and that we are adding one redundant power supply.

We need a formula to show what to expect when we can tolerate a failure and still provide service. To simplify the calculations, we assume that the lifetimes of the components are exponentially distributed and that there is no dependency between the component failures. MTTF for our redundant power supplies is the mean time until one power supply fails divided by the chance that the other will fail before the first one is replaced. Thus, if the chance of a second failure before repair is small, then the MTTF of the pair is large.

Since we have two power supplies and independent failures, the mean time until one disk fails is $\frac{\text{MTTF}_{\text{power supply}}}{2}$. A good approximation of the probability of a second failure is MTTR over the mean time until the other power supply fails. Hence, a reasonable approximation for a redundant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}_{\text{power supply}}}{2} \cdot \frac{\text{MTTF}_{\text{power supply}}}{2} = \frac{\text{MTTF}_{\text{power supply}}^2}{2 \times \text{MTTR}_{\text{power supply}}}$$

Using the MTTF numbers above, if we assume it takes on average 24 hours for a human operator to notice that a power supply has failed and replace it, the reliability of the fault tolerant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}_{\text{power supply}}^2}{2 \times \text{MTTR}_{\text{power supply}}} = \frac{200,000^2}{2 \times 24} \approx 830,000,000$$

making the pair about 4150 times more reliable than a single power supply.
20. Performance Measurements

Standardized benchmark application suites

SPEC: Standard Performance Evaluation Corporation, spec.org

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Ultra 5 time (sec)</th>
<th>Opteron time (sec)</th>
<th>SPEC Ratio</th>
<th>Itanium 2 time (sec)</th>
<th>SPEC Ratio</th>
<th>Opteron/Itanium times (sec)</th>
<th>Itanium/Opteron SPEC Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>wupwise</td>
<td>1600</td>
<td>51.5</td>
<td>31.06</td>
<td>56.1</td>
<td>28.53</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>swim</td>
<td>3100</td>
<td>125.0</td>
<td>24.73</td>
<td>70.7</td>
<td>43.85</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>mgrid</td>
<td>1800</td>
<td>98.0</td>
<td>18.37</td>
<td>65.8</td>
<td>27.36</td>
<td>1.49</td>
<td>1.49</td>
</tr>
<tr>
<td>applu</td>
<td>2100</td>
<td>94.0</td>
<td>22.34</td>
<td>50.9</td>
<td>41.25</td>
<td>1.85</td>
<td>1.85</td>
</tr>
<tr>
<td>mesa</td>
<td>1400</td>
<td>64.6</td>
<td>21.69</td>
<td>108.0</td>
<td>12.99</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>gangel</td>
<td>2900</td>
<td>86.4</td>
<td>33.57</td>
<td>40.0</td>
<td>72.47</td>
<td>2.16</td>
<td>2.16</td>
</tr>
<tr>
<td>art</td>
<td>2600</td>
<td>92.4</td>
<td>28.13</td>
<td>21.0</td>
<td>123.67</td>
<td>4.40</td>
<td>4.40</td>
</tr>
<tr>
<td>cquake</td>
<td>1300</td>
<td>72.6</td>
<td>17.92</td>
<td>36.3</td>
<td>35.78</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>facerec</td>
<td>1900</td>
<td>73.6</td>
<td>25.80</td>
<td>86.9</td>
<td>21.86</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>ammp</td>
<td>2200</td>
<td>136.0</td>
<td>16.14</td>
<td>132.0</td>
<td>16.63</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>lucas</td>
<td>2000</td>
<td>88.8</td>
<td>22.52</td>
<td>107.0</td>
<td>18.76</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>fma3d</td>
<td>2100</td>
<td>120.0</td>
<td>17.48</td>
<td>131.0</td>
<td>16.09</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>sixtrack</td>
<td>1100</td>
<td>123.0</td>
<td>8.95</td>
<td>68.8</td>
<td>15.99</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>aspi</td>
<td>2600</td>
<td>150.0</td>
<td>17.36</td>
<td>231.0</td>
<td>11.27</td>
<td>0.65</td>
<td>0.65</td>
</tr>
</tbody>
</table>

Geometric mean 20.86 27.12 1.30 1.30

Figure 1.17 SPECfp2000 execution times (in seconds) for the Sun Ultra 5—the reference computer of SPEC2000—and execution times and SPEC R 2000 ratios for the AMD Opteron and Intel Itanium 2. (SPEC2000 multiplies the ratio of execution times by 100 to remove the decimal point from the result, so 20.86 is reported as 208.6.) The final two columns show the ratios of execution times and SPEC R 20000 ratios. This figure demonstrates the irrelevance of the reference computer in relative performance. The ratio of the execution times is identical to the ratio of the SPEC R 20000 ratios, and the ratio of the geometric means (27.12/20.86 = 1.30) is identical to the geometric mean of the ratios (1.30).
## 21. Summarizing Performance

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

*Which system is faster?*

(from D. Chasaki)
### Average vs. Relative Throughput

... depends who's selling

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

**Average throughput**

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.50</td>
<td>2.00</td>
<td>1.25</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Throughput relative to B**

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>2.00</td>
<td>0.50</td>
<td>1.25</td>
</tr>
</tbody>
</table>

**Throughput relative to A**

(from D. Chasaki)
23. How to Mislead with Performance Reports

- Select pieces of workload that work well on your design, ignore others
- Use unrealistic data set sizes for application (too big or too small)
- Report throughput numbers for a latency benchmark
- Report latency numbers for a throughput benchmark
- Report performance on a kernel and claim it represents an entire application
- Use 16-bit fixed-point arithmetic (because it’s fastest on your system) even though application requires 64-bit floating-point arithmetic
- Use a less efficient algorithm on the competing machine
- Report speedup for an inefficient algorithm (bubblesort)
- Compare hand-optimized assembly code with unoptimized C code
- Compare your design using next year’s technology against competitor’s year old design (1% performance improvement per week)
- Ignore the relative cost of the systems being compared
- Report averages and not individual results
- Report speedup over unspecified base system, not absolute times
- Report efficiency not absolute times
- Report MFLOPS not absolute times (use inefficient algorithm)

[David Bailey “Twelve ways to fool the masses when giving performance results for parallel supercomputers”]

(from D. Chasaki)
24. Speedup and Amdahl's Law

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

Alternatively,

\[
\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}
\]

\[
\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left(1 - \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}\right)
\]

The overall speedup is the ratio of the execution times:

\[
\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{\left(1 - \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}\right)}
\]

Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

\[
\text{Fraction}_{\text{enhanced}} = 0.4; \text{Speedup}_{\text{enhanced}} = 10; \text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56
\]
25. Reliability Amdahl Example

Amdahl’s law is applicable beyond performance. Let’s redo the reliability example from page 35 after improving the reliability of the power supply via redundancy from 200,000-hour to 830,000,000-hour MTTF, or 4150X better.

The calculation of the failure rates of the disk subsystem was

\[
\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1,000,000} + \frac{1}{500,000} + \frac{1}{200,000} + \frac{1}{200,000} + \frac{1}{1,000,000}
\]

\[
= \frac{10 + 2 + 5 + 5 + 1}{1,000,000 \text{ hours}} = \frac{23}{1,000,000 \text{ hours}}
\]

Therefore, the fraction of the failure rate that could be improved is 5 per million hours out of 23 for the whole system, or 0.22.

The reliability improvement would be

\[
\text{Improvement}_{\text{power supply pair}} = \frac{1}{(1 - 0.22) + \frac{0.22}{4150}} = \frac{1}{0.78} = 1.28
\]

Despite an impressive 4150X improvement in reliability of one module, from the system’s perspective, the change has a measurable but small benefit.
26. Processor Performance Equation

CPI = average clock cycles per instruction

\[
\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}
\]

CPU time = Instruction count \times \text{Cycles per instruction} \times \text{Clock cycle time}

\[
\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}
\]

As this formula demonstrates, processor performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. Furthermore, CPU time is equally dependent on these three characteristics; for example, a 10% improvement in any one of them leads to a 10% improvement in CPU time.
27. CPI

\[ \text{CPU clock cycles} = \sum_{i=1}^{n} IC_i \times CPI_i \]

\[ \text{CPU time} = \left( \sum_{i=1}^{n} IC_i \times CPI_i \right) \times \text{Clock cycle time} \]

\[ \text{CPI} = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{IC_i}{\text{Instruction count}} \times CPI_i \]
28. CPI Example

Suppose we have made the following measurements:

Frequency of FP operations = 25%
Average CPI of FP operations = 4.0
Average CPI of other instructions = 1.33
Frequency of FPSQR = 2%
CPI of FPSQR = 20

Assume that the two design alternatives are to decrease the CPI of FPSQR to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the processor performance equation.

First, observe that only the CPI changes; the clock rate and instruction count remain identical. We start by finding the original CPI with neither enhancement:

\[
\text{CPI}_{\text{original}} = \sum_{i=1}^{n} \text{CPI}_i \times \left( \frac{\text{IC}_i}{\text{Instruction count}} \right)
\]

\[
= (4 \times 25\%) + (1.33 \times 75\%) = 2.0
\]

We can compute the CPI for the enhanced FPSQR by subtracting the cycles saved from the original CPI:

\[
\text{CPI}_{\text{with new FPSQR}} = \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}})
\]

\[
= 2.0 - 2\% \times (20 - 2) = 1.64
\]

We can compute the CPI for the enhancement of all FP instructions the same way or by summing the FP and non-FP CPIs. Using the latter gives us:

\[
\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625
\]

Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better.
29. CPI Example, Speedup

\[
\text{Speedup}_{\text{new FP}} = \frac{\text{CPU time}_\text{original}}{\text{CPU time}_\text{new FP}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_\text{original}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_\text{new FP}}
\]

\[
= \frac{\text{CPI}_\text{original}}{\text{CPI}_\text{new FP}} = \frac{2.00}{1.625} = 1.23
\]

Alternative formulation and solution using Amdahl's law:

(FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

\[
\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22
\]

\[
\text{Speedup}_{\text{FP}} = \frac{1}{(1 - 0.5) + \frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23
\]

Improving the performance of the FP operations overall is slightly better.
30. Instruction Set Architecture - Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Use</th>
<th>Preserved across a call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
<td>N.A.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assembler temporary</td>
<td>No</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2–3</td>
<td>Values for function results and expression evaluation</td>
<td>No</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4–7</td>
<td>Arguments</td>
<td>No</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>8–15</td>
<td>Temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>16–23</td>
<td>Saved temporaries</td>
<td>Yes</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24–25</td>
<td>Temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>26–27</td>
<td>Reserved for OS kernel</td>
<td>No</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 1.4 MIPS registers and usage conventions. In addition to the 32 general-purpose registers (R0–R31), MIPS has 32 floating-point registers (F0–F31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number.

R0 is hard-wired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. R0 can also be used as a source when a zero value is needed.

R31 is the default destination register used by jump instructions. Otherwise R31 can be used as a normal register.

All other registers are available for general-purpose use.

We will use generic assembly pseudo-code, e.g. DSUBI R3,R2,#100
# 31. Instruction Set Architecture - Formats

## Basic instruction formats

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25 25</td>
<td>21 20</td>
<td>16 15</td>
<td>11 10</td>
<td>6 5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25 25</td>
<td>21 20</td>
<td>16 15</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25 25</td>
<td></td>
</tr>
</tbody>
</table>

## Floating-point instruction formats

<table>
<thead>
<tr>
<th>FR</th>
<th>opcode</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25 25</td>
<td>21 20</td>
<td>16 15</td>
<td>11 10</td>
<td>6 5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FI</th>
<th>opcode</th>
<th>fmt</th>
<th>ft</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25 25</td>
<td>21 20</td>
<td>16 15</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.6 MIPS64 instruction set architecture formats.** All instructions are 32 bits long. The R format is for integer register-to-register operations, such as DADDU, DSUBU, and so on. The I format is for data transfers, branches, and immediate instructions, such as LD, SD, BEQZ, and DADDIs. The J format is for jumps, the FR format for floating-point operations, and the FI format for floating-point branches.

---

shamt = shift amount, used with the shift and rotate instructions, the amount by which the source operand rs is rotated/shifted.
## 32. Instruction Set Architecture - Opcodes

<table>
<thead>
<tr>
<th>Instruction type/opcode</th>
<th>Instruction meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data transfers</strong></td>
<td></td>
</tr>
<tr>
<td>LB, LBU, SB, LH, LBU, SH</td>
<td>Load byte, load byte unsigned, store byte (to/from integer registers)</td>
</tr>
<tr>
<td>LW, LWU, SW</td>
<td>Load word, load word unsigned, store word (to/from integer registers)</td>
</tr>
<tr>
<td>LD, SD</td>
<td>Load double word, store double word (to/from integer registers)</td>
</tr>
<tr>
<td>L.S, L.D, S.S, S.D</td>
<td>Load SP float, load DP float, store SP float, store DP float</td>
</tr>
<tr>
<td><strong>MFCO, MFCI</strong></td>
<td>Copy from/to GPRI to/from a special register</td>
</tr>
<tr>
<td><strong>MOV, S, MOVD</strong></td>
<td>Copy one SP or DP FP register to another FP register</td>
</tr>
<tr>
<td><strong>MFCI, MFC2</strong></td>
<td>Copy 32 bits from/to FP registers from/to integer registers</td>
</tr>
<tr>
<td><strong>Arithmetic/logical</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ADD, ADDI, ADDU, ADDIU</strong></td>
<td>Add, add immediate (all immediate are 16 bits), signed and unsigned</td>
</tr>
<tr>
<td><strong>SUB, SUBU</strong></td>
<td>Subtract, signed and unsigned</td>
</tr>
<tr>
<td><strong>DMUL, DMULL, DMIV, DIV</strong></td>
<td>Multiply and divide, signed and unsigned; multiply-unsigned; all operations take and yield</td>
</tr>
<tr>
<td><strong>DUI, MODD</strong></td>
<td>64-bit values</td>
</tr>
<tr>
<td><strong>AND, ANDI</strong></td>
<td>And, and immediate</td>
</tr>
<tr>
<td><strong>O1, O1I, ORI, XORI</strong></td>
<td>Or, or immediate, exclusive or, exclusive or immediate</td>
</tr>
<tr>
<td><strong>LUI</strong></td>
<td>Load upper immediate; loads bits 32 to 63 of register with immediate, then sign-extends</td>
</tr>
<tr>
<td><strong>DSSL, DSSL, DSSA, DSSLX, DSSLW, DSSLW</strong></td>
<td>Shifts both immediate (DS_2) and variable form (DS_1); shifts are sign left logical, right logical, right arithmetic</td>
</tr>
<tr>
<td><strong>SLT, SLTI, SLTUL, SLTIU</strong></td>
<td>Set less than, set less than immediate, signed and unsigned</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td></td>
</tr>
<tr>
<td><strong>BEQZ, BNEZ</strong></td>
<td>Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td><strong>BEQ, BNE</strong></td>
<td>Branch GPR equal/not equal; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td><strong>BLEZ, BGEZ</strong></td>
<td>Test comparison bit in the FP status register branch; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td><strong>J, JR</strong></td>
<td>Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)</td>
</tr>
<tr>
<td><strong>JAL, JALR</strong></td>
<td>Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)</td>
</tr>
<tr>
<td><strong>TRAP</strong></td>
<td>Transfer to operating system at a vectored address</td>
</tr>
<tr>
<td><strong>ENET</strong></td>
<td>Return to user code from an exception; restore user mode</td>
</tr>
</tbody>
</table>

### Floating point

- **ADD, ADDS, ADDPS, ADDPS**
- **SUB, SUBS, SUBPS, SUBPS**
- **MUL, MULS, MULPS, MULPS**
- **MADD, MADDS, MADDPS, MADDPS**
- **DIV, DIVS, DIVPS, DIVPS**
- **CVT, CVT**
- **Cvt, Cvt**
- **Cvt, Cvt**

*Figure 1.5 Subset of the instructions in MIPS64. SP = single precision; DP = double precision. Appendix A gives much more detail on MIPS64. For data, the most significant bit number is 0; least is 63.*
### 33. Instruction Set Architecture - Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed</td>
<td>Add R3,(R1 + R2)</td>
<td>Regs[R3] ← Regs[R3] + Mem[Reg[R1] + Regs[R2]]</td>
<td>Sometimes useful in array addressing; R1 = base of array; R2 = index amount.</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>Regs[R1] ← Regs[R1] + Mem[1001]</td>
<td>Sometimes useful for accessing static data; address constant may need to be large.</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>Regs[R1] ← Regs[R1] + Mem[Reg[R3]]</td>
<td>If R3 is the address of a pointer $p$, then mode yields $p$.</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1,(R2)+</td>
<td>Regs[R1] ← Regs[R1] + Mem[Reg[R2]]</td>
<td>Useful for stepping through arrays within a loop, R2 points to start of array, each reference increments R2 by size of an element, $d$.</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1, -(R2)</td>
<td>Regs[R2] ← Regs[R2] - d [Reg[R2]]</td>
<td>Same as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] + d]</td>
<td>Used to index arrays. May be applied to any indexed addressing mode in some computers.</td>
</tr>
</tbody>
</table>

**Figure A.6 Selection of addressing modes with examples, meaning, and usage.** In autoincrement/decrement and scaled addressing modes, the variable $d$ designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A-36.

Errata: use displacement addressing to simulate register indirect with 0 for the address displacement.