Amdahl’s Law

- The fundamental theorem of performance optimization
- Made by Amdahl!
  - One of the designers of the IBM 360
- Optimizations do not (generally) uniformly affect the entire program
  - The more widely applicable a technique is, the more valuable it is
  - Conversely, limited applicability can (drastically) reduce the impact of an optimization.

Always heed Amdahl’s Law!!!

It is central to many many optimization problems
Amdahl’s Law

- The fundamental theorem of performance optimization
- Made by Amdahl!
  - One of the designers of the IBM 360

- Optimizations do not (generally) uniformly affect the entire program
  - The more widely applicable a technique is, the more valuable it is
  - Conversely, limited applicability can (drastically) reduce the impact of an optimization.

Always heed Amdahl’s Law!!!
It is central to many many optimization problems
Amdahl’s Law in Action

• SuperJPEG-O-Rama2010 ISA extensions **
  – Speeds up JPEG decode by 10x!!!
  – Act now! While Supplies Last!
Amdahl’s Law in Action

• SuperJPEG-O-Rama2010 ISA extensions **
  – Speeds up JPEG decode by 10x!!!
  – Act now! While Supplies Last!

**
SuperJPEG-O-Rama Inc. makes no claims about the usefulness of this software for any purpose whatsoever. It may not even build. It may cause fatigue, blindness, lethargy, malaise, and irritability. Debugging maybe hazardous. It will almost certainly cause ennui. Do not taunt SuperJPEG-O-Rama. Will not, on grounds of principle, decode images of Justin Beiber. Images of Lady Gaga maybe transposed, and meat dresses may be rendered as tofu. Not covered by US export control laws or the Geneva convention, although it probably should be. Beware of dog. Increases processor cost by 45%. Objects in the rear view mirror may appear closer than they are. Or is it farther? Either way, watch out! If you use SuperJPEG-O-Rama, the cake will not be a lie. All your base are belong to 141L. No whining or complaining. Wingeing is allowed, but only in countries where “wingeing” is a word.
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 ISA extensions
  - Speeds up JPEG decode by 10x!!!
  - Act now! While Supplies Last!

Increases processor cost by 45%. Objects in the rear view mirror may appear closer than they are. Or is it farther? Either way, watch out! If you use SuperJPEG-O-Rama, the cake will not be a lie. All your base are belong to 141L. No whining or complaining. Wingeing is allowed, but only in countries where “wingeing” is a word.
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?

![Diagram showing time spent on JPEG decode with and without JOR2k]

30s
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?

<table>
<thead>
<tr>
<th></th>
<th>w/o JOR2k</th>
<th>w/ JOR2k</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Decode</td>
<td>21s</td>
<td>21s</td>
</tr>
</tbody>
</table>

30s
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?

\[
\text{Performance: } \frac{30}{21} = 1.42x \text{ Speedup } \neq 10x
\]
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of its time doing JPEG decode
- How much does JOR2k help?

Performance: \(\frac{30}{21} = 1.42x\) Speedup !\(\neq 10x\)

Amdahl ate our Speedup!
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?

<table>
<thead>
<tr>
<th></th>
<th>JPEG Decode</th>
<th>Overall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o JOR2k</td>
<td></td>
<td>30s</td>
</tr>
<tr>
<td>w/ JOR2k</td>
<td>21s</td>
<td></td>
</tr>
</tbody>
</table>

Performance: $\frac{30}{21} = 1.42 \times$ Speedup $\neq 10 \times$

Is this worth the 45% increase in cost?
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?

<table>
<thead>
<tr>
<th></th>
<th>JPEG Decode</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o JOR2k</td>
<td>21s</td>
<td>30s</td>
</tr>
<tr>
<td>w/ JOR2k</td>
<td>30s</td>
<td>21s</td>
</tr>
</tbody>
</table>

Performance: \( \frac{30}{21} = 1.42x \text{ Speedup} \neq 10x \\

Is this worth the 45% increase in cost?

Metric = Latency * Cost =>
Amdahl’s Law in Action

• SuperJPEG-O-Rama2010 in the wild
• PictoBench spends 33% of it’s time doing JPEG decode
• How much does JOR2k help?

Performance: 30/21 = 1.42x Speedup != 10x

Is this worth the 45% increase in cost?

Metric = Latency * Cost => No
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of its time doing JPEG decode
- How much does JOR2k help?

<table>
<thead>
<tr>
<th></th>
<th>w/o JOR2k</th>
<th>w/ JOR2k</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Decode</td>
<td>30s</td>
<td>21s</td>
</tr>
</tbody>
</table>

Performance: $\frac{30}{21} = 1.42 \times$ Speedup $\neq 10 \times$

Is this worth the 45% increase in cost? No

Metric = Latency * Cost $\Rightarrow$ No

Metric = Latency^2 * Cost $\Rightarrow$ No
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?

Performance: 30/21 = 1.42x Speedup != 10x

Is this worth the 45% increase in cost?

Metric = Latency * Cost => No
Metric = Latency^2 * Cost => Yes
Explanation

- Latency*Cost and Latency²*Cost are smaller-is-better metrics.
- Old System: No JOR2k
  - Latency = 30s
  - Cost = C (we don’t know exactly, so we assume a constant, C)

- New System: With JOR2k
  - Latency = 21s
  - Cost = 1.45 * C

- Old Latency*Cost = 30*C
- New Latency*Cost = 21*1.45*C
- New/Old Latency*Cost = 21*1.45*C/30*C = 1.015
- New is bigger (worse) than old by 1.015x

- Old Latency²*Cost = 30²*C
- New Latency²*Cost = 21²*1.45*C
- New/Old Latency²*Cost = 21²*1.45*C/30²*C = 0.71
- New is smaller (better) than old by 0.71x

In general, you can make C = 1, and just leave it out.
Explanation

- Latency*Cost and Latency^2*Cost are smaller-is-better metrics.
- **Old System:** No JOR2k
  - Latency = 30s
  - Cost = C (we don’t know exactly, so we assume a constant, C)
- **New System:** With JOR2k
  - Latency = 21s
  - Cost = 1.45 * C

For Latency*Cost:
- Old: 30*C
- New: 21*1.45*C
- New/Old = 21*1.45*C/30*C = 1.015
- New is bigger (worse) than old by 1.015x

For Latency^2*Cost:
- Old: 30^2*C
- New: 21^2*1.45*C
- New/Old = 21^2*1.45*C/30^2*C = 0.71
- New is smaller (better) than old by 0.71x

In general, you can make C = 1, and just leave it out.
Explanation

- Latency*Cost and Latency\(^2\)*Cost are smaller-is-better metrics.
- Old System: No JOR2k
  - Latency = 30s
  - Cost = C (we don’t know exactly, so we assume a constant, C)
- New System: With JOR2k
  - Latency = 21s
  - Cost = 1.45 * C
- Latency*Cost
  - Old: 30*C
  - New: 21*1.45*C
  - New/Old = 21*1.45*C/30*C = 1.015
  - New is bigger (worse) than old by 1.015x
Explanation

- Latency*Cost and Latency^2*Cost are smaller-is-better metrics.

- Old System: No JOR2k
  - Latency = 30s
  - Cost = C (we don’t know exactly, so we assume a constant, C)

- New System: With JOR2k
  - Latency = 21s
  - Cost = 1.45 * C

- Latency*Cost
  - Old: 30*C
  - New: 21*1.45*C
  - New/Old = 21*1.45*C/30*C = 1.015
  - New is bigger (worse) than old by 1.015x

- Latency^2*Cost
  - Old: 30^2 *C
  - New: 21^2 *1.45*C
  - New/Old = 21^2*1.45*C/30^2*C = 0.71
  - New is smaller (better) than old by 0.71x
Explanation

- Latency*Cost and Latency^{2}*Cost are smaller-is-better metrics.
- Old System: No JOR2k
  - Latency = 30s
  - Cost = C (we don’t know exactly, so we assume a constant, C)
- New System: With JOR2k
  - Latency = 21s
  - Cost = 1.45 * C
- Latency*Cost
  - Old: 30*C
  - New: 21*1.45*C
  - New/Old = 21*1.45*C/30*C = 1.015
  - New is bigger (worse) than old by 1.015x
- Latency^{2}*Cost
  - Old: 30^{2} *C
  - New: 21^{2} *1.45*C
  - New/Old = 21^{2} *1.45*C/30^{2} *C = 0.71
  - New is smaller (better) than old by 0.71x
- In general, you can make C = 1, and just leave it out.
Amdahl’s Law

• The second fundamental theorem of computer architecture.
• If we can speed up $x$ of the program by $S$ times.
• Amdahl’s Law gives the total speed up, $S_{tot}$

$$S_{tot} = \frac{1}{\left(\frac{x}{S} + (1-x)\right)}.$$
Amdahl’s Law

- The second fundamental theorem of computer architecture.
- If we can speed up $x$ of the program by $S$ times
- Amdahl’s Law gives the total speed up, $S_{tot}$

$$S_{tot} = \frac{1}{\frac{x}{S} + (1-x)}$$

Sanity check:

$$x = 1 \Rightarrow S_{tot} = \frac{1}{\frac{1}{S} + (1-1)} = \frac{1}{S} = S$$
Amdahl’s Corollary #1

- Maximum possible speedup $S_{\text{max}}$, if we are targeting $x$ of the program.

\[
S = \infty
\]

\[
S_{\text{max}} = \frac{1}{1-x}
\]
Amdahl’s Law Example #1

• **Protein String Matching Code**
  • It runs for 200 hours on the current machine, and spends 20% of time doing integer instructions
  • How much faster must you make the integer unit to make the code run 10 hours faster?
  • How much faster must you make the integer unit to make the code run 50 hours faster?
Amdahl’s Law Example #2

- Protein String Matching Code
  - 4 days execution time on current machine
    - 20% of time doing integer instructions
    - 35% percent of time doing I/O
  - Which is the better tradeoff?
    - Compiler optimization that reduces number of integer instructions by 25% (assume each integer instruction takes the same amount of time)
    - Hardware optimization that reduces the latency of each IO operations from 6us to 5us.
Explanation

- **Speed up integer ops**
  - $x = 0.2$
  - $S = \frac{1}{1-0.25} = 1.33$
  - $S_{\text{int}} = \frac{1}{0.2/1.33 + 0.8} = 1.052$

- **Speed up IO**
  - $x = 0.35$
  - $S = \frac{6\text{us}}{5\text{us}} = 1.2$
  - $S_{\text{io}} = \frac{1}{0.35/1.2 + 0.65} = 1.062$

- **Speeding up IO is better**
Amdahl’s Corollary #2

• Make the common case fast (i.e., x should be large)!
  • Common == “most time consuming” not necessarily “most frequent”
  • The uncommon case doesn’t make much difference
  • Be sure of what the common case is
  • The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.

• Repeat…
  • With optimization, the common becomes uncommon.
  • An uncommon case will (hopefully) become the new common case.
  • Now you have a new target for optimization.
Amdahl’s Corollary #2: Example

- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently (e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.)
  - e.g. Focus on function call overheads (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

Common case
Amdahl’s Corollary #2: Example

• In the end, there is no common case!
  • Options:
    • Global optimizations (faster clock, better compiler)
    • Divide the program up differently
      • e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
      • e.g. Focus on function call over heads (which are everywhere).
    • War of attrition
  • Total redesign (You are probably well-prepared for this)

Common case

7x => 1.4x
Amdahl’s Corollary #2: Example

- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
e.g. Focus on function call over heads (which are everywhere).
- War of attrition
- Total redesign (You are probably well-prepared for this)

Common case
- 7x => 1.4x
- 4x => 1.3x
Amdahl’s Corollary #2: Example

- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
    - e.g. Focus on function call overhead (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

Common case:
- 7x => 1.4x
- 4x => 1.3x
- 1.3x => 1.1x

Total = 20/10 = 2x
Amdahl’s Corollary #2: Example

- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
    - e.g. Focus on function call over heads (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

**Common case**
- 7x => 1.4x
- 4x => 1.3x
- 1.3x => 1.1x

Total = 20/10 = 2x
Amdahl’s Corollary #3

• Benefits of parallel processing
• $p$ processors
• $x$ of the program is $p$-way parallelizable
• Maximum speedup, $S_{par}$

$$S_{par} = \frac{1}{\frac{x}{p} + (1-x)}$$
Amdahl’s Corollary #3

- Benefits of parallel processing
- $p$ processors
- $x$ of the program is $p$-way parallelizable
- Maximum speedup, $S_{par}$

$$S_{par} = \frac{1}{ \frac{x}{p} + (1-x) }.$$ 

- A key challenge in parallel programming is increasing $x$ for large $p$.
  - $x$ is pretty small for desktop applications, even for $p = 2$
  - This is a big part of why multi-processors are of limited usefulness.
Example #3

• Recent advances in process technology have quadruple the number transistors you can fit on your die.

• Currently, your key customer can use up to 4 processors for 40% of their application.

• You have two choices:
  • Increase the number of processors from 1 to 4
  • Use 2 processors but add features that will allow the application to use 2 processors for 80% of execution.

• Which will you choose?
Amdahl’s Corollary #4

- Amdahl’s law for latency (L)
- By definition
  - Speedup = oldLatency/newLatency
  - newLatency = oldLatency \times \frac{1}{\text{Speedup}}
- By Amdahl’s law:
  - newLatency = x \times \text{old Latency} / S + \text{oldLatency} \times (1-x)
- Amdahl’s law for latency
  - newLatency = x \times \text{oldLatency} / S + \text{oldLatency} \times (1-x)
Amdahl’s Non-Corollary

- Amdahl’s law does not bound slowdown
  - newLatency = x*oldLatency/S + oldLatency*(1-x)
  - newLatency is linear in 1/S

- Things can only get so fast, but they can get arbitrarily slow.

- Do not hurt the non-common case too much!
Amdahl’s Non-Corollary

- Amdahl’s law does not bound slowdown
  - newLatency \( = \frac{x \cdot \text{oldLatency}}{S} + \text{oldLatency} \cdot (1-x) \)
  - newLatency is linear in \( \frac{1}{S} \)
- Example: \( x = 0.01 \) of execution, oldLat = 1
  - \( S = 0.001; \)
    - Newlat = \( 1000 \cdot \text{Oldlat} \cdot 0.01 + \text{Oldlat} \cdot (0.99) = \sim 10 \cdot \text{Oldlat} \)
  - \( S = 0.00001; \)
    - Newlat = \( 100000 \cdot \text{Oldlat} \cdot 0.01 + \text{Oldlat} \cdot (0.99) = \sim 1000 \cdot \text{Oldlat} \)

Things can only get so fast, but they can get arbitrarily slow.

Do not hurt the non-common case too much!
Amdahl’s Non-Corollary

- Amdahl’s law does not bound slowdown
  - newLatency = x*oldLatency/S + oldLatency*(1-x)
  - newLatency is linear in 1/S
- Example: \( x = 0.01 \) of execution, oldLat = 1
  - \( S = 0.001; \)
    - newlat = 1000*Oldlat *0.01 + Oldlat *(0.99) = \sim 10*Oldlat
  - \( S = 0.00001; \)
    - newlat = 100000*Oldlat *0.01 + Oldlat *(0.99) = \sim 1000*Oldlat
- Things can only get so fast, but they can get arbitrarily slow.
  - Do not hurt the non-common case too much!
Amdahl’s Example #4
This one is tricky

• Memory operations currently take 30% of execution time.
• A new widget called a “cache” speeds up 80% of memory operations by a factor of 4.
• A second new widget called a “L2 cache” speeds up 1/2 the remaining 20% by a factor of 2.
• What is the total speed up?
**Explanation**

- **Apply the L1 cache first**
  - $S_1 = 4$
  - $x_1 = 0.8 \times 0.3$
  - $S_{totL1} = 1 / (x_1 / S_1 + (1-x_1))$
  - $S_{totL1} = 1 / (0.8 \times 0.3 / 4 + (1-(0.8 \times 0.3))) = 1 / (0.06 + 0.76) = 1.2195$ times

- **Then, apply the L2 cache**
  - $S_{L2} = 2$
  - $x_{L2} = 0.3 \times (1 - 0.8) / 2 = 0.03$
  - $S_{totL2} = 1 / (0.03 / 2 + (1-0.03)) = 1 / (0.015 + 0.97) = 1.015$ times

- **Combine**
  - $S_{totL2} = S_{totL2}' \times S_{totL1} = 1.02 \times 1.21 = 1.237$
Answer in Pictures

OOPS: Speed up = 1.242
Amdahl’s Pitfall: This is wrong!

- You cannot trivially apply optimizations one at a time with Amdahl’s law.
- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8 \times .3$
  - $S_{\text{totL1}} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{\text{totL1}} = 1/(0.8 \times 0.3/4 + (1-(0.8 \times 0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Then, apply the L2 cache
  - $S_{L2} = 2$
  - $x_{L2} = 0.3 \times (1 - 0.8)/2 = 0.03$
  - $S_{\text{totL2}} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$ times
- Combine
  - $S_{\text{totL2}} = S_{\text{totL2'}} \times S_{\text{totL1}} = 1.02 \times 1.21 = 1.237$

- What’s wrong? -- after we do the L1 cache, the execution time changes, so the fraction of execution that the L2 effects actually grows
Amdahl’s Pitfall: This is wrong!

- You cannot trivially apply optimizations one at a time with Amdahl’s law.
  - Apply the L1 cache first
    - $S_1 = 4$
    - $x_1 = .8 \times .3$
    - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
    - $S_{totL1} = 1/(0.8 \times 0.3/4 + (1-(0.8 \times 0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
  - Then, apply the L2 cache
    - $S_{L2} = 2$
    - $x_{L2} = 0.3 \times (1 - 0.8)/2 = 0.03$
    - $S_{totL2} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$ times
  - Combine
    - $S_{totL2} = S_{totL2} \times S_{totL1} = 1.02 \times 1.21 = 1.237$

- What’s wrong? -- after we do the L1 cache, the execution time changes, so the fraction of execution that the L2 effects actually grows
Amdahl’s Pitfall: This is wrong!

- You cannot trivially apply optimizations one at a time with Amdahl’s law.
- Apply the L1 cache first
  - \( S_1 = 4 \)
  - \( x_1 = .8 \times .3 \)
  - \( S_{\text{totL1}} = 1/(x_1/S_1 + (1-x_1)) \)
  - \( S_{\text{totL1}} = 1/(0.8 \times 0.3 / 4 + (1-(0.8 \times 0.3))) = 1/(0.06 + 0.76) = 1.2195 \) times
- Then, apply the L2 cache
  - \( S_{L2} = 2 \)
  - \( x_{L2} = 0.3 \times (1 - 0.8) / 2 = 0.03 \)
  - \( S_{\text{totL2}} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015 \) times
- Combine
  - \( S_{\text{totL2}} = S_{\text{totL2}'} \times S_{\text{totL1}} = 1.02 \times 1.21 = 1.237 \)
  - This is wrong
  - So is this

- What’s wrong? -- after we do the L1 cache, the execution time changes, so the fraction of execution that the L2 effects actually grows
Answer in Pictures

\[
\text{Speed up} = 1.242
\]
Multiple optimizations done right

• We can apply the law for multiple optimizations
• Optimization 1 speeds up $x_1$ of the program by $S_1$
• Optimization 2 speeds up $x_2$ of the program by $S_2$
  • $Stot = 1/(x_1/S_1 + x_2/S_2 + (1-x_1-x_2))$

Note that $x_1$ and $x_2$ must be disjoint!
• i.e., $S_1$ and $S_2$ must not apply to the same portion of execution.
• If not then, treat the overlap as a separate portion of execution and measure it's speed up independently
  • ex: we have $x_1$ only, $x_2$ only, and $x_1$ & $x_2$ and $S_1$ only, $S_2$ only, and $S_1$ & $S_2$$
    • $Stot = 1/(x_1/S_1 + x_2/S_2 + x_1&2/S_1&2 + (1-x_1-x_2))$

• You can estimate $S_1&2$ as $S_1$ only * $S_2$ only, but the real value could be higher or lower.
Multiple optimizations done right

• We can apply the law for multiple optimizations
• Optimization 1 speeds up x1 of the program by S1
• Optimization 2 speeds up x2 of the program by S2
  • \( Stot = 1/(x_1/S_1 + x_2/S_2 + (1-x_1-x_2)) \)
• Note that x1 and x2 must be disjoint!
  • i.e., S1 and S2 must not apply to the same portion of execution.
• If not then, treat the overlap as a separate portion of execution and measure it’s speed up independently
  • ex: we have \( x_{1\text{only}}, x_{2\text{only}}, \) and \( x_{1&2} \) and \( S_{1\text{only}}, S_{2\text{only}}, \) and \( S_{1&2} \)
  • Then \( S_{tot} = 1/(x_{1\text{only}}/S_{1\text{only}} + x_{2\text{only}}/S_{2\text{only}} + x_{1&2}/S_{1&2} + (1 - x_{1\text{only}} - x_{2\text{only}} - x_{1&2})) \)
  • You can estimate \( S_{1&2} \) as \( S_{1\text{only}} \times S_{2\text{only}} \), but the real value could be higher or lower.
Multiple Opt. Practice

- Combine both the L1 and the L2
  - memory operations are 30% of execution time
  - $S_{L1} = 4$
  - $x_{L1} = 0.3 \times 0.8 = 0.24$
  - $S_{L2} = 2$
  - $x_{L2} = 0.3 \times (1 - 0.8)/2 = 0.03$
  - $S_{totL2} = 1/(x_{L1}/S_{L1} + x_{L2}/S_{L2} + (1 - x_{L1} - x_{L2}))$
  - $S_{totL2} = 1/(0.24/4 + 0.03/2 + (1 - 0.24 - 0.03))$
    - $= 1/(0.06 + 0.015 + 0.73)) = 1.24$ times
The Idea of the CPU
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

---

**CPU**

**PC**

- Instruction Memory
  ```
  [80000180] 0001d821 addu $27, $0, $1
  [80000184] 3c019000 lui $1, $28672
  [80000188] ac202000 sw $2, $912($1)
  [8000018c] 3c019000 lui $1, $28672
  [80000190] ac202000 sw $4, $916($1)
  [80000194] 401a6800 mfc0 $26, $13
  [80000198] 001a2082 srl $4, $26, 2
  [8000019c] 3084001f andi $4, $4, 31
  [800001a0] 34020004 ori $2, $0, 4
  [800001a4] 3c049000 lui $4, $28672 __m1_
  [800001a8] 0000000c syscall
  [800001ac] 34020001 ori $2, $0, 1
  [800001b0] 001a2082 srl $4, $26, 2
  [800001b4] 3084001f andi $4, $4, 31
  ```

- Data Memory
  ```
  [7ffffffe60] 74736574 test 2. a
  [7ffffff70] 5f524553 SER_T
  [7ffffff80] 78303d47 g=0 x L
  [7ffffff90] 5f444a61 AND_MODE
  [7fffffff0] 70410033 3 Apple
  [7ffffffe0] 656b6366 socket Re
  [7fffffff0] 616c2270 p l
  ```
The Stored Program Computer

• The program is data
  • It is a series of bits
  • It lives in memory
  • A series of discrete “instructions”

• The program counter (PC) control execution
  • It points to the current instruction
  • Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"

- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The program is *data*
- It is a series of bits
- It lives in memory
- A series of discrete “instructions”

The program counter (PC) control execution
- It points to the current instruction
- Advances through the program
The Stored Program Computer

- The program is data
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

![CPU Diagram](image-url)

**Instruction Memory**

```
[80000010] 001d0101  addu $27, $0, $1
[80000014] 3c019000  lui $1, -28672
[80000018] ac220200  sw $2, $12($1)
[8000001c] 3c019000  lui $1, -28672
[80000020] ac240204  sw $4, $16($1)
[80000024] 401a6000  mfc0 $26, $13
[80000028] 001a2502  srl $4, $26, 2
[8000002c] 30840001f  andi $4, $4, 31
[80000030] 34020004  ori $2, $0, 4
[80000034] 3c049000  lui $4, -28672 [__m1__]
[80000038] 0000000c  syscall
[8000003c] 34020001  ori $2, $0, 1
[80000040] 001a2002  srl $4, $26, 2
[80000044] 3084001f  andi $4, $4, 31
```

**Data Memory**

```
[7ffffffc] 74736574 73612e32 test2.as
[7ffffffe] 5552e53e 54584554 SER_06.txt
[7ffffffe] 78303d47 3a364631 G=0xf61
[7ffffffe] 5f444e41 AND_MODE
[7ffffffe] 70410033 5f656760 3 Apple
[7ffffffe] 656e6366 65525f74 socket_re
[7ffffffe] 616c2270 68836e75 p/launch
```

The Stored Program Computer

- The program is data
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The Stored Program Computer

- The program is \textit{data}
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"

- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

![Diagram of CPU, Instruction Memory, and Data Memory]
The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions
The Instruction Set Architecture (ISA)

• The ISA is the set of instructions a computer can execute
• All programs are combinations of these instructions
• It is an abstraction that programmers (and compilers) use to express computations
  • The ISA defines a set of operations, their semantics, and rules for their use.
  • The software agrees to follow these rules.
The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions
- It is an abstraction that programmers (and compilers) use to express computations
  - The ISA defines a set of operations, their semantics, and rules for their use.
  - The software agrees to follow these rules.
- The hardware can implement those rules IN ANY WAY IT Chooses!
  - Directly in hardware
  - Via a software layer (i.e., a virtual machine)
  - Via a trained monkey with a pen and paper
  - Via a software simulator (like SPIM)
The MIPS ISA
Two ISAs

• MIPS
  • Simple, elegant, easy to implement
  • Designed with the benefit many years ISA design experience
  • Designed for modern programmers, tools, and applications

• x86
  • Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  • Designed for 1970s technology
  • Nearly the last in long series of unfortunate ISA designs.
  • The dominant ISA in modern computer systems.
Two ISAs

- **MIPS**
  - Simple, elegant, easy to implement
  - Designed with the benefit of many years ISA design experience
  - Designed for modern programmers, tools, and applications

- **x86**
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.
MIPS Basics

• Instructions
  • 4 bytes (32 bits)
  • 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  • Instructions operate on memory and registers
MIPS Basics

• Instructions
  • 4 bytes (32 bits)
  • 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  • Instructions operate on memory and registers

• Memory Data types (also aligned)
  • Bytes -- 8 bits
  • Half words -- 16 bits
  • Words -- 32 bits
  • Memory is denote “M” (e.g., M[0x10] is the byte at address 0x10)
MIPS Basics

- **Instructions**
  - 4 bytes (32 bits)
  - 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  - Instructions operate on memory and registers

- **Memory Data types (also aligned)**
  - Bytes -- 8 bits
  - Half words -- 16 bits
  - Words -- 32 bits
  - Memory is denoted “M” (e.g., M[0x10] is the byte at address 0x10)

- **Registers**
  - 32 4-byte registers in the “register file”
  - Denoted “R” (e.g., R[2] is register 2)
Bytes and Words

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0006</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>

- In modern ISAs (including MIPS) memory is “byte addressable”
- In MIPS, half words and words are aligned.
The MIPS Register File

- All registers are the same
- Where a register is needed any register will work
- By convention, we use them for particular tasks
- Argument passing
- Temporaries, etc.
- These rules ("the register discipline") are part of the ISA
- $zero is the "zero register"
- It is always zero.
- Writes to it have no effect.

<table>
<thead>
<tr>
<th>Name</th>
<th>number</th>
<th>use</th>
<th>Callee saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assemble Temp</td>
<td>no</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2 - 3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$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 - $k1</td>
<td>26 - 27</td>
<td>Res. for OS</td>
<td>yes</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS R-Type Arithmetic Instructions

- **R-Type instructions** encode operations of the form “a = b OP c” where ‘OP’ is +, -, <<, &, etc.

- **Bit fields**
  - “opcode” encodes the operation type.
  - “funct” specifies the particular operation.
  - “rs” are “rt” source registers; “rd” is the destination register
    - 5 bits can specify one of 32 registers.

- **“shamt”** is the “shift amount” for shift operations
  - Since registers are 32 bits, 5 bits are sufficient

### Examples

- **add $t0, $t1, $t2**
  - opcode = 0, funct = 0x20

- **nor $a0, $s0, $t4**
  - opcode = 0, funct = 0x27

- **sll $t0, $t1, 4**
  - opcode = 0, funct = 0x0, shamt = 4
R-Type encodes “register-indirect” jumps

Jump register
- \texttt{jr rs}: \texttt{PC} = \texttt{R[rs]}

Jump and link register
- \texttt{jalr rs, rd}: \texttt{R[rd]} = \texttt{PC + 8}; \texttt{PC} = \texttt{R[rs]}
- \texttt{rd} default to \texttt{$ra} (i.e., the assembler will fill it in if you leave it out)

**Examples**

- \texttt{jr $t2}
  - \texttt{PC} = \texttt{r[10]}
  - \texttt{opcode} = 0, \texttt{funct} = 0x8
- \texttt{jalr $t0}
  - \texttt{PC} = \texttt{R[8]}
  - \texttt{R[31]} = \texttt{PC + 8}
  - \texttt{opcode} = 0, \texttt{funct} = 0x9
MIPS I-Type Arithmetic Instructions

- I-Type arithmetic instructions encode operations of the form “a = b OP #”
  - ‘OP’ is +, -, <<, &, etc and # is an integer constant
  - More formally, e.g.: R[rd] = R[rs] + 42
- Components
  - “opcode” encodes the operation type.
  - “rs” is the source register
  - “rd” is the destination register
- “immediate” is a 16 bit constant used as an argument for the operation

Examples
- addi $t0, $t1, -42
  - opcode = 0x8
- ori $t0, $zero, 42
  - R[4] = R[0] | 42
  - opcode = 0xd
  - Loads a constant into $t0
MIPS I-Type Branch Instructions

- I-Type also encode branches
  - if (R[rd] OP R[rs])
    \[ PC = PC + 4 + 4 \times \text{Immediate} \]
  - else
    \[ PC = PC + 4 \]

- Components
  - “rs” and “rt” are the two registers to be compared
  - “rt” is sometimes used to specify branch type.

- “immediate” is a 16 bit branch offset
  - It is the signed offset to the target of the branch
  - Limits branch distance to 32K instructions
  - Usually specified as a label, and the assembler fills it in for you.

Examples

- \text{beq} $t0$, $t1$, $-42$
    \[ PC = PC + 4 + 4 \times -42 \]
  - opcode = 0x4
- \text{bgez} $t0$, $-42$
  - if R[8] >= 0
    \[ PC = PC + 4 + 4 \times -42 \]
  - opcode = 0x1
  - rt = 1
MIPS I-Type Memory Instructions

- I-Type also encode memory access
  - Store: \( M[R[rs] + \text{Immediate}] = R[rt] \)
  - Load: \( R[rt] = M[R[rs] + \text{Immediate}] \)
- MIPS has load/stores for byte, half word, and word
- Sub-word loads can also be signed or unsigned
  - Signed loads sign-extend the value to fill a 32 bit register.
  - Unsigned zero-extend the value.
- “immediate” is a 16 bit offset
  - Useful for accessing structure components
  - It is signed.

Examples

- `lw $t0, 4($t1)`
  - opcode = 0x23
- `sb $t0, -17($t1)`
  - opcode = 0x28
MIPS J-Type Instructions

- J-Type encodes the jump instructions
- Plain Jump
  - JumpAddress = \( \{PC+4[31:28], Address, 2'b0\} \)
  - Address replaces most of the PC
  - \( PC = \text{JumpAddress} \)
- Jump and Link
  - \( R[\text{ra}] = PC + 8; PC = \text{JumpAddress}; \)
- J-Type also encodes misc instructions
  - syscall, interrupt return, and break

Examples

- \( j \ $t0 \)
  - \( PC = R[8] \)
  - opcode = 0x2
- \( jal \ $t0 \)
  - \( R[31] = PC + 8 \)
  - \( PC = R[8] \)