Lecture 1: Cost/Performance, DLX, Pipelining, Caches, Branch Prediction

Prof. Fred Chong
ECS 250A Computer Architecture
Winter 1999
(Slides based upon Patterson UCB CS252 Spring 1998)

Computer Architecture Is …
the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.
Amdahl, Blaaw, and Brooks, 1964

Computer Architecture’s Changing Definition
• 1950s to 1960s: Computer Architecture Course
  Computer Arithmetic
• 1970s to mid 1980s: Computer Architecture Course
  Instruction Set Design, especially ISA appropriate for compilers
• 1990s: Computer Architecture Course
  Design of CPU, memory system, I/O system, Multiprocessors

ECS 250A Course Focus
Understanding the design techniques, machine structures, technology factors, evaluation methods that will determine the form of computers in 21st Century

Computer Architecture Topics
- Instruction Set Architecture
- Pipelining, Hazard Resolution, Superscalar, Reordering, Prediction, Speculation, Vector, DSP
- Addressing, Protection, Exception Handling
- Coherence, Bandwidth, Latency
- Emerging Technologies
- Interleaving
- Interconnection Network
- Bus protocols
- Topologies, Routing, Bandwidth, Latency, Reliability

Computer Architecture’s Changing Definition
- Technology
- Programming Languages
- Operating Systems
- Measurement & Evaluation
- Interface Design (ISA)
- Instruction Set Design
- Organization
- Hardware

Computer Architecture Topics
- Instruction Set Architecture
- Pipelining, Hazard Resolution, Superscalar, Reordering, Prediction, Speculation, Vector, DSP
- Addressing, Protection, Exception Handling
- Coherence, Bandwidth, Latency
- Emerging Technologies
- Interleaving
- Interconnection Network
- Bus protocols
- Topologies, Routing, Bandwidth, Latency, Reliability

Computer Architecture Topics
- Instruction Set Architecture
- Pipelining, Hazard Resolution, Superscalar, Reordering, Prediction, Speculation, Vector, DSP
- Addressing, Protection, Exception Handling
- Coherence, Bandwidth, Latency
- Emerging Technologies
- Interleaving
- Interconnection Network
- Bus protocols
- Topologies, Routing, Bandwidth, Latency, Reliability
Topic Coverage
- Performance/Cost, DLX, Pipelining, Caches, Branch Prediction
- ILP, Loop Unrolling, Scoreboarding, Tomasulo, Dynamic Branch Prediction
- Trace Scheduling, Speculation
- Vector Processors, DSPs
- Memory Hierarchy
- I/O
- Interconnection Networks
- Multiprocessors

ECS250A: Staff

Instruction: Fred Chong
Office: EUR-3031 chong@ics
Office Hours: Mon 4-6pm or by appt.
T. A: Diana Keen
Office: EUR-2239 keen@ics
TA Office Hours: Fri 1-3pm

Class: Mon 6:10-9pm
Web page: http://arch.cs.ucdavis.edu/~chong/250A/
Lectures available online before 1PM day of lecture
Newsgroup: ucd.class.cs250a{.d}

Grading
- Problem Sets 35%
- 1 In-class exam (prelim simulation) 20%
- Project Proposals and Drafts 10%
- Project Final Report 25%
- Project Poster Session (CS colloquium) 10%

VLSI Transistors

CMOS Inverter

CMOS NAND Gate
Integrated Circuits Costs

\[
\text{IC cost} = \text{Die cost} + \text{Testing cost} + \text{Packaging cost}
\]

Die cost = Wafer cost / Dies per Wafer * Die yield

Dies per Wafer = \( \frac{\text{Wafer diam}^2}{2} \) - Die Area - Test dies

Die Area = \( 2 \times \text{Die Area} \)

Die Yield = Wafer yield * \( \frac{1}{\text{Dies per wafer}} \) * \( \frac{1}{\text{Die Area}} \)

Die Cost goes roughly with die area

Real World Examples

<table>
<thead>
<tr>
<th>Chip</th>
<th>Metal Line</th>
<th>Wafer Defect</th>
<th>Area</th>
<th>Dies/ Yield</th>
<th>Die Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>386DX</td>
<td>0.90</td>
<td>$900</td>
<td>1.0</td>
<td>43</td>
<td>350</td>
</tr>
<tr>
<td>486DX2</td>
<td>0.80</td>
<td>$1200</td>
<td>1.0</td>
<td>81</td>
<td>181</td>
</tr>
<tr>
<td>PowerPC 601</td>
<td>4.0</td>
<td>$1700</td>
<td>1.3</td>
<td>121</td>
<td>115</td>
</tr>
<tr>
<td>HP PA 7100</td>
<td>3.0</td>
<td>$1300</td>
<td>1.0</td>
<td>196</td>
<td>66</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>3.70</td>
<td>$1500</td>
<td>1.2</td>
<td>234</td>
<td>53</td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>3.70</td>
<td>$1700</td>
<td>1.6</td>
<td>256</td>
<td>48</td>
</tr>
<tr>
<td>Pentium</td>
<td>3.80</td>
<td>$1500</td>
<td>1.5</td>
<td>296</td>
<td>40</td>
</tr>
</tbody>
</table>


Cost/Performance

What is Relationship of Cost to Price?

- Component Costs
- Direct Costs (add 25% to 40%) recurring costs: labor, purchasing, scrap, warranty
- Gross Margin (add 6% to 18%) nonrecurring costs: R&D, marketing, sales, equipment maintenance, rental, financing cost, pre-tax profit, taxes
- Average Discount to get List Price (add 33% to 66%): volume discounts and/or retailer markup

Component Cost

Direct Cost

Gross Margin

Average Discount

Average Selling Price

List Price

Gross Margin

Average Discount

Cost

Summary: Price vs. Cost

Technology Trends: Microprocessor Capacity

Moore’s Law

CMOS improvements:
- Die size: 2X every 3 yrs
- Line width: half 7 yrs

Alpha 21264: 15 million
Pentium Pro: 5.5 million
PowerPC 630: 0.9 million
Alpha 21164: 9.3 million
Sparc Ultra: 5.2 million

Chip Prices (August 1993)

- Assume purchase 10,000 units

<table>
<thead>
<tr>
<th>Chip</th>
<th>Area</th>
<th>Mfg. cost</th>
<th>Price</th>
<th>Multiplier</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>386DX</td>
<td>43</td>
<td>$9</td>
<td>$31</td>
<td>3.4</td>
<td>Intense Competition</td>
</tr>
<tr>
<td>486DX2</td>
<td>81</td>
<td>$35</td>
<td>$245</td>
<td>7.0</td>
<td>No Competition</td>
</tr>
<tr>
<td>PowerPC 601</td>
<td>121</td>
<td>$77</td>
<td>$280</td>
<td>3.6</td>
<td>Recoup R&amp;D?</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>234</td>
<td>$202</td>
<td>$1231</td>
<td>6.3</td>
<td>Early in shipments</td>
</tr>
<tr>
<td>Pentium</td>
<td>296</td>
<td>$473</td>
<td>$965</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

Summary: Price vs. Cost

Memory Capacity
(Single Chip DRAM)

<table>
<thead>
<tr>
<th>Year</th>
<th>Size (Mb)</th>
<th>Cycle Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>0.0625</td>
<td>250</td>
</tr>
<tr>
<td>1983</td>
<td>0.25</td>
<td>220</td>
</tr>
<tr>
<td>1986</td>
<td>1</td>
<td>190</td>
</tr>
<tr>
<td>1990</td>
<td>4</td>
<td>165</td>
</tr>
<tr>
<td>1993</td>
<td>16</td>
<td>146</td>
</tr>
<tr>
<td>1996</td>
<td>64</td>
<td>120</td>
</tr>
<tr>
<td>2000</td>
<td>256</td>
<td>100</td>
</tr>
</tbody>
</table>

Technology Trends
(Summary)

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Speed (latency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>2x in 3 years</td>
</tr>
<tr>
<td>DRAM</td>
<td>4x in 3 years</td>
</tr>
<tr>
<td>Disk</td>
<td>4x in 3 years</td>
</tr>
</tbody>
</table>

Processor Performance
(1.35X before, 1.55X now)

<table>
<thead>
<tr>
<th>Year</th>
<th>Processor Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1987</td>
<td>DEC Alpha 21264/600</td>
</tr>
<tr>
<td>1988</td>
<td>DEC Alpha 5/500</td>
</tr>
<tr>
<td>1989</td>
<td>DEC Alpha 5/300</td>
</tr>
<tr>
<td>1990</td>
<td>DEC Alpha 4/266</td>
</tr>
<tr>
<td>1991</td>
<td>IBM POWER 100</td>
</tr>
<tr>
<td>1992</td>
<td>DEC AXP/500</td>
</tr>
<tr>
<td>1993</td>
<td>HP 9000/750</td>
</tr>
<tr>
<td>1994</td>
<td>Sun -4/260</td>
</tr>
<tr>
<td>1995</td>
<td>IBM RS/6000</td>
</tr>
<tr>
<td>1996</td>
<td>MIPS M/120</td>
</tr>
<tr>
<td>1997</td>
<td>MIPS M/2000</td>
</tr>
</tbody>
</table>

Performance Trends
(Summary)

- Workstation performance (measured in Spec Marks) improves roughly 50% per year (2X every 18 months)
- Improvement in cost performance estimated at 70% per year

Computer Engineering
Methodology

Technology Trends
**Computer Engineering Methodology**

- Evaluate Existing Systems for Bottlenecks
- Technology Trends
- Technology Benchmarks

**Measurement Tools**
- Benchmarks, Traces, Mixes
- Hardware: Cost, delay, area, power estimation
- Simulation (many levels)
  - ISA, RT, Gate, Circuit
- Queuing Theory
- Rules of Thumb
- Fundamental “Laws”/Principles

**The Bottom Line: Performance (and Cost)**

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (ppm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>BAG-Sud Airways</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

- Time to run the task (ExTime)
  - Execution time, response time, latency
- Tasks per day, hour, week, sec, ns ... (Performance)
  - Throughput, bandwidth

**The Bottom Line: Performance (and Cost)**

"X is n times faster than Y" means

\[
\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}
\]

- Speed of Concorde vs. Boeing 747
- Throughput of Boeing 747 vs. Concorde
Amdahl’s Law

Speedup due to enhancement E:

\[
\text{Speedup}(E) = \frac{\text{ExTime} \text{ w/o } E}{\text{ExTime} \text{ w/ } E} = \frac{\text{Performance} \text{ w/ } E}{\text{Performance} \text{ w/o } E}
\]

Suppose that enhancement E accelerates a fraction \( F \) of the task by a factor \( S \), and the remainder of the task is unaffected.

\[ \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (1 - \text{Fraction enhanced}) + \text{Fraction enhanced} \]

\[
\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction enhanced}) + \text{Fraction enhanced}}
\]

Amdahl’s Law

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

\[ \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + 0.1/2) = 0.95 \times \text{ExTime}_{\text{old}} \]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{0.95} = 1.053
\]

Amdahl’s Law

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

\[ \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (1 - \text{Fraction enhanced}) + \text{Fraction enhanced} \]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction enhanced}) + \text{Fraction enhanced}}
\]

Amdahl’s Law

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

\[ \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + 0.1/2) = 0.95 \times \text{ExTime}_{\text{old}} \]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{0.95} = 1.053
\]

Amdahl’s Law

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

\[ \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (1 - \text{Fraction enhanced}) + \text{Fraction enhanced} \]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction enhanced}) + \text{Fraction enhanced}}
\]

Aspects of CPU Performance

<table>
<thead>
<tr>
<th>CPU time</th>
<th>Seconds</th>
<th>Instructions/Program</th>
<th>Cycles/Second</th>
<th>Operations/MIPS</th>
<th>Megabytes/Second</th>
<th>Cycles/Second</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>( X )</td>
<td>( (X) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>( X )</td>
<td>( X )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. Set.</td>
<td>( X )</td>
<td>( X )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>( X )</td>
<td>( X )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td>( X )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Cycles Per Instruction**

“Average Cycles per Instruction”

\[ CPI = \frac{\text{CPU time}}{\text{Instruction Count}} = \frac{\text{CycleTime} \times \sum_{i=1}^{n} CPI_i \cdot F_i}{\text{Instruction Count}} \]

“Instruction Frequency”

\[ CPI = \sum_{i=1}^{n} CPI_i \cdot F_i \]

Invest Resources where time is Spent!

**Example: Calculating CPI**

**Base Machine (Reg/Reg)**

\[ \text{Op} \quad \text{Free} \quad \text{Cycles} \quad \text{CPI}(i) \quad \text{(% Time)} \]

<table>
<thead>
<tr>
<th>ALU</th>
<th>50%</th>
<th>1</th>
<th>0.5</th>
<th>(33%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>0.4</td>
<td>(27%)</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>0.2</td>
<td>(13%)</td>
</tr>
<tr>
<td>Branch</td>
<td>25%</td>
<td>2</td>
<td>0.4</td>
<td>(27%)</td>
</tr>
</tbody>
</table>

Typical Mix

**SPEC: System Performance Evaluation Cooperative**

- **First Round 1989**
  - 10 programs yielding a single number ("SPECmarks")

- **Second Round 1992**
  - SPECint92 (6 integer programs) and SPECfp92 (14 floating point programs)
    - Compiler Flags unlimited. March 93 of DEC 4000 Model 610:
      - spice: unix://deux/dec/4000/vax/speccomp/has_bcopy_/hoopy/lib/has_bcopy_/has_bcopy_/hoopy/lib/hoopy/has_bcopy_/spice_/has_bcopy_/has_bcopy_/spice_/has_bcopy_/has_bcopy_/
      - wave5: /all/all/document/aigen/u=4/u=200
      - nasa7: /home/nasa/u=4/u=200/lc=blas

- **Third Round 1995**
  - new set of programs: SPECint95 (8 integer programs) and SPECfp95 (10 floating point)
    - “benchmarks useful for 3 years”
      - Single flag setting for all programs: SPECint_base95, SPECfp_base95

**How to Summarize Performance**

- Arithmetic mean (weighted arithmetic mean) tracks execution time: \( \sum(T_i)/n \) or \( \sum(W_i \cdot T_i) \)
- Harmonic mean (weighted harmonic mean) of rates (e.g., MFLOPS) tracks execution time: \( n/\sum(1/R_i) \) or \( n/\sum(W_i/R_i) \)
- Normalized execution time is handy for scaling performance (e.g., \( X \times \) times faster than SPARCstation 10)
- But do not take the arithmetic mean of normalized execution time, use the geometric: \( (\prod x_i)^{1/n} \)

**SPEC First Round**

- One program: 99% of time in single line of code
- New front-end compiler could improve dramatically

**Impact of Means on SPECmark89 for IBM 550**

<table>
<thead>
<tr>
<th>Program</th>
<th>Before</th>
<th>After</th>
<th>Before</th>
<th>After</th>
<th>Before</th>
<th>After</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>30</td>
<td>29</td>
<td>49</td>
<td>51</td>
<td>9.31</td>
<td>9.22</td>
<td>7.64</td>
<td>7.86</td>
</tr>
<tr>
<td>espresso</td>
<td>35</td>
<td>34</td>
<td>65</td>
<td>67</td>
<td>7.64</td>
<td>7.86</td>
<td>6.68</td>
<td>6.68</td>
</tr>
<tr>
<td>spice</td>
<td>47</td>
<td>47</td>
<td>510</td>
<td>510</td>
<td>6.59</td>
<td>5.69</td>
<td>5.81</td>
<td>5.45</td>
</tr>
<tr>
<td>dodux</td>
<td>46</td>
<td>49</td>
<td>41</td>
<td>38</td>
<td>6.58</td>
<td>5.45</td>
<td>7.95</td>
<td>8.66</td>
</tr>
<tr>
<td>li</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>3.54</td>
<td>3.54</td>
<td>3.54</td>
<td>3.54</td>
</tr>
<tr>
<td>eqntott</td>
<td>40</td>
<td>40</td>
<td>22</td>
<td>22</td>
<td>9.01</td>
<td>9.01</td>
<td>8.91</td>
<td>8.91</td>
</tr>
<tr>
<td>matrix200</td>
<td>78</td>
<td>730</td>
<td>56</td>
<td>6</td>
<td>3.43</td>
<td>0.37</td>
<td>6.68</td>
<td>6.68</td>
</tr>
<tr>
<td>fpppp</td>
<td>90</td>
<td>87</td>
<td>34</td>
<td>35</td>
<td>2.97</td>
<td>3.07</td>
<td>2.97</td>
<td>3.07</td>
</tr>
<tr>
<td>tomcatv</td>
<td>33</td>
<td>138</td>
<td>20</td>
<td>19</td>
<td>2.01</td>
<td>1.94</td>
<td>2.01</td>
<td>1.94</td>
</tr>
<tr>
<td>mean</td>
<td>54</td>
<td>72</td>
<td>124</td>
<td>106</td>
<td>54.42</td>
<td>49.26</td>
<td>54.42</td>
<td>49.26</td>
</tr>
</tbody>
</table>

**Symbolic Notation**

- Geometric: \( (\prod x_i)^{1/n} \)
- Arithmetic: \( \sum(T_i)/n \)
- Weighted Arithmetic: \( \sum(W_i \cdot T_i) \)

**SPECmark89, Time: Weighted Time:**

- Ratio to VAX: 1.33
- Ratio to Arithmetic: 1.18
- Ratio to Weighted Arithmetic: 1.09
Performance Evaluation

- “For better or worse, benchmarks shape a field”
- Good products created when have:
  - Good benchmarks
  - Good ways to summarize performance
- Given sales is a function in part of performance relative to competition, investment in improving product as reported by performance summary
- If benchmarks/summary inadequate, then choose between improving product for real programs vs. improving product to get more sales; Sales almost always wins!
- Execution time is the measure of computer performance!

Instruction Set Architecture (ISA)

A good interface:
- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels

Evolution of Instruction Sets

- Single Accumulator (EDSAC 1950)
- Accumulator + Index Registers (Manchester Mark I, IBM 700 series 1953)
- Separation of Programming Model from Implementation (B5000 1963, IBM 360 1964)
- High-level Language Based
- Concept of a Family
- General Purpose Register Machines
- Complex Instruction Sets (Vax, Intel 432 1977-80)
- Load/Store Architecture (CDC 6600, Cray 1 1963-76)
- RISC (Mips, Sparc, HP-PA, IBM RS6000, ... 1987)

Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)
- Design decisions must take into account:
  - Technology
  - Machine organization
  - Programming languages
  - Compiler technology
  - Operating systems
- And they in turn influence these

A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
- No indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-RISC, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: MIPS

Register-Register

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rd</th>
<th>Rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>Op</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary, #1

- Designing to Last through Trends
  - Capacity
    - Logic: 2x in 3 years, 2x in 3 years
    - DRAM: 4x in 3 years, 2x in 10 years
    - Disk: 4x in 3 years, 2x in 10 years
  - Speed
    - 6 yrs to graduate to 16x CPU speed, DRAM/Disk size
- Time to run the task
  - Execution time, response time, latency
  - Tasks per day, hour, week, sec, ns, ...
    - Throughput, bandwidth
- “X is n times faster than Y” means
  \[
  \frac{ExTime(Y)}{Performance(X)} = \frac{ExTime(X)}{Performance(Y)}
  \]

Summary, #2

- Amdahl’s Law:
  \[
  \text{Speedup}_{\text{overall}} = \frac{1}{\frac{ExTime_{\text{old}}}{Performance(Y)} + \frac{\text{Fraction}_{\text{enhanced}}}{ExTime_{\text{new}}}}
  \]
- CPI Law:
  \[
  \text{ExTime}_{\text{new}} = \frac{\text{CPU time}}{\text{Seconds}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
  \]

Pipelining: Its Natural!

- Laundry Example
  - Ann, Brian, Cathy, Dave
  - Each have one load of clothes
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

Sequential Laundry

<table>
<thead>
<tr>
<th>Time</th>
<th>Task Order</th>
<th>Task Order</th>
<th>Task Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Midnight</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry

<table>
<thead>
<tr>
<th>Time</th>
<th>Task Order</th>
<th>Task Order</th>
<th>Task Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Midnight</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Pipelined laundry takes 3.5 hours for 4 loads

[Page 9]
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload.
- Pipeline rate limited by slowest pipeline stage.
- Multiple tasks operating simultaneously.
- Potential speedup = Number pipe stages.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to “fill” pipeline and time to “drain” it reduces speedup.

Computer Pipelines

- Execute billions of instructions, so throughput is what matters.
- DLX desirable features: all instructions same length, registers located in same place in instruction format, memory operands only in loads or stores.

5 Steps of DLX Datapath

- Instruction Fetch
- Instr. Decode Reg. Fetch
- Memory Access
- Write Back

Visualizing Pipelining

- Data stationary control: local decode for each instruction phase / pipeline stage.

Its Not That Easy for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle.
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away).
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock).
  - Control hazards: Pipelining of branches & other instructions stall the pipeline until the hazard “bubbles” in the pipeline.

Page 10
### Example: Dual-port vs. Single-port
- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed
  - SpeedUp = Pipeline depth/\(1 + 0\) x (clock\(_{unpipelined}\)/clock\(_{pipelined}\)) = Pipeline depth
  - SpeedUp = Pipeline depth/\(1 + 0.4 \times 1\) x (clock\(_{unpipelined}\)/clock\(_{unpipelined} / 1.05\)) = 0.75 x Pipeline Depth
  - SpeedUp = Pipeline depth/\(0.75 \times \) Pipeline Depth = 1.33
- Machine A is 1.33 times faster

### Speed Up Equation for Pipelining
- CPI\(_{pipelined}\) = Ideal CPI + Pipeline stall clock cycles per instr
- Speedup = Ideal CPI x Pipeline depth x Clock Cycle\(_{unpipelined}\)/Clock Cycle\(_{pipelined}\)
- Speedup = Pipeline depth x Clock Cycle\(_{unpipelined}\)/Pipeline stall CPI x Clock Cycle\(_{pipelined}\)

### Three Generic Data Hazards
- Instr\(_r\), followed by Instr\(_s\)
- Read After Write (RAW)
  - Instr\(_s\) tries to read operand before Instr\(_r\) writes it
Three Generic Data Hazards
Instr_I followed by Instr_J

- Write After Read (WAR)
  Instr_J tries to write operand before Instr_I reads it
  - Gets wrong operand
- Can’t happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Three Generic Data Hazards
Instr_I followed by Instr_J

- Write After Write (WAW)
  Instr_J tries to write operand before Instr_I writes it
  - Leaves wrong result (Instr_I not Instr_J)
- Can’t happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

Forwarding to Avoid Data Hazard

Data Hazard Even with Forwarding

HW Change for Forwarding

Figure 3.20, Page 161

Data Hazard Even with Forwarding

Figure 3.12, Page 153

Figure 3.10, Page 149

Figure 3.13, Page 154
Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \) in memory.

**Slow code:**
- LW Rb, b
- LW Rc, c
- ADD Ra, Rb, Rc
- LW Rj, a
- SW a, Ra
- LW Re, e
- LW Rf, f
- SUB Rd, Re, Rf
- SW d, Rd

**Fast code:**
- LW Rb, b
- LW Rc, c
- LW Re, e
- ADD Ra, Rb, Rc
- LW Rj, a
- SW a, Ra
- SUB Rd, Re, Rf
- SW d, Rd

Control Hazard on Branches

Three Stage Stall

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- DLX branch tests if register = 0 or ° 0
- DLX Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Branch Stall Impact

- 1 clock cycle penalty for branch versus 3

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear
#2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% DLX branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
#3: Predict Branch Taken
  - 53% DLX branches taken on average
  - But haven’t calculated branch target address in DLX
  - DLX still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome
#4: Delayed Branch
  - Define branch to take place AFTER a following instruction
  - Branch instruction sequential successor
  - Branch target if taken
  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
  - DLX uses this
Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Cancelling branches allow more slots to be filled

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Evaluating Branch Alternatives

<table>
<thead>
<tr>
<th>Scheduling Scheme</th>
<th>Branch Penalty</th>
<th>CPI Stall</th>
<th>Speedup vs. Unpipelined</th>
<th>Speedup vs. Stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>1.42</td>
<td>3.5</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Predict taken</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1.09</td>
<td>4.5</td>
<td>1.29</td>
<td></td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.57</td>
<td>4.6</td>
<td>1.31</td>
<td></td>
</tr>
</tbody>
</table>

Conditional & Unconditional = 14%, 65% change PC

Pipelining Summary

- Just overlap tasks, and easy if tasks are independent
- Speed Up $\propto$ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW, WAR, WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction