The Von Neumann Computer Model

- Partitioning of the computing engine into components:
  - Central Processing Unit (CPU): Control Unit (instruction decode, sequencing of operations), Datapath (registers, arithmetic and logic unit, buses).
  - Memory: Instruction and operand storage.
  - Input/Output (I/O) sub-system: I/O bus, interfaces, devices.
  - The stored program concept: Instructions from an instruction set are fetched from a common memory and executed one at a time.

Major CPU Performance Limitation: The Von Neumann computing model implies sequential execution, one instruction at a time.
Generic CPU Machine Instruction Execution Steps

(Implied by The Von Neumann Computer Model)

- **Instruction Fetch**
  - Obtain instruction from program storage

- **Instruction Decode**
  - Determine required actions and instruction size

- **Operand Fetch**
  - Locate and obtain operand data

- **Execute**
  - Compute result value or status

- **Result Store**
  - Deposit results in storage for later use

- **Next Instruction**
  - Determine successor or next instruction

Major CPU Performance Limitation: The Von Neumann computing model implies sequential execution one instruction at a time
CPU Organization

• **Datapath Design:**
  –Capabilities & performance characteristics of principal Functional Units (FUs):
    • (e.g., Registers, ALU, Shifters, Logic Units, ...)
  –Ways in which these components are interconnected (buses connections, multiplexors, etc.).
  –How information flows between components.

• **Control Unit Design:**
  –Logic and means by which such information flow is controlled.
  –Control and coordination of FUs operation to realize the targeted Instruction Set Architecture to be implemented (can either be implemented using a finite state machine or a microprogram).

• **Description of hardware operations with a suitable language, possibly using Register Transfer Notation (RTN).**
Recent Trends in Computer Design

- The cost/performance ratio of computing systems have seen a steady decline due to advances in:
  - Integrated circuit technology: *decreasing feature size, \( \lambda \)
    - Clock rate improves roughly proportional to improvement in \( \lambda \)
    - Number of transistors improves proportional to \( \lambda^2 \) (or faster).
  - Architectural improvements in CPU design.

- Microprocessor systems directly reflect IC and architectural improvement in terms of a yearly 35 to 55% improvement in performance.
- Assembly language has been mostly eliminated and replaced by other alternatives such as C or C++
- Standard operating Systems (UNIX, Windows) lowered the cost of introducing new architectures.
- Emergence of RISC architectures and RISC-core (x86) architectures.
- Adoption of quantitative approaches to computer design based on empirical performance observations.
- Increased importance of exploiting thread-level parallelism (TLP) in main-stream computing systems.
Mass-produced microprocessors a cost-effective high-performance replacement for custom-designed mainframe/minicomputer CPUs
Microprocessor Performance
1987-97

Integer SPEC92 Performance

100x performance in the last decade
Microprocessor Frequency Trend

Result:
- Deeper Pipelines
- Longer stalls
- Higher CPI (lowers effective performance per cycle)

Frequency doubles each generation
Number of gates/clock reduce by 25%
Leads to deeper pipelines with more stages
(e.g. Intel Pentium 4E has 30+ pipeline stages)

Realty Check:
Clock frequency scaling is slowing down!
(Did silicone finally hit the wall?)

Why?
1- Power leakage
2- Clock distribution delays

Why?
1- Power leakage
2- Clock distribution delays

Processors:
- 386, 486
- Pentium (R)
- Pentium Pro
- Pentium (R) II
- Pentium (R) III
- Pentium (R) IV
- Pentium (R) Extreme Edition
- Intel Core (R) 2 Duo
- Intel Core (R) i7
- AMD Opteron
- AMD Phenom
- AMD Athlon
- IBM Power PC

Gate Delays/ Clock
- Intel
- IBM Power PC
- DEC

Processor freq scales by 2X per generation

Why?
1- Power leakage
2- Clock distribution delays

Result:
Deeper Pipelines
Longer stalls
Higher CPI (lowers effective performance per cycle)
Microprocessor Transistor Count Growth Rate

Moore's Law: 2X transistors/Chip
Every 1.5 years
(circa 1970)

Currently > 1 Billion

Alpha 21264: 15 million
Pentium Pro: 5.5 million
PowerPC 620: 6.9 million
Alpha 21164: 9.3 million
Sparc Ultra: 5.2 million
Computer Technology Trends: Evolutionary but Rapid Change

- **Processor:**
  - 1.5-1.6 performance improvement every year; Over 100X performance in last decade.

- **Memory:**
  - DRAM capacity: > 2x every 1.5 years; 1000X size in last decade.
  - Cost per bit: Improves about 25% per year.

- **Disk:**
  - Capacity: > 2X in size every 1.5 years.
  - Cost per bit: Improves about 60% per year.
  - 200X size in last decade.
  - Only 10% performance improvement per year, due to mechanical limitations.

- **Expected State-of-the-art PC by end of year 2005:**
  - Processor clock speed: > 4000 MegaHertz (4 Giga Hertz)
  - Memory capacity: > 4000 MegaByte (4 Giga Bytes)
  - Disk capacity: > 500 GigaBytes (0.5 Tera Bytes)
Hierarchy of Computer Architecture

High-Level Language Programs

Software

Machine Language Program

Software/Hardware Boundary

Hardware

Logic Diagrams

Circuit Diagrams

Instruction Set Architecture

Operating System

Instr. Set Proc. I/O system

Compiler Firmware

Application

Assembly Language Programs

e.g. BIOS (Basic Input/Output System)

Instruction Set

Architecture

Microprogram

Register Transfer Notation (RTN)

VLSI placement & routing

Circuit Design Layout

Digital Design

Logic Diagrams

Circuit Diagrams

EECC551 - Shaaban

#10  Lec # 1  Fall 2005  9-6-2005
Computer Architecture Vs. Computer Organization

• The term **Computer architecture** is sometimes erroneously restricted to computer **instruction set design**, with other aspects of computer design called **implementation**

• More accurate definitions:
  
  – **Instruction set architecture (ISA):** The actual programmer-visible instruction set and serves as the boundary between the software and hardware.
  
  – Implementation of a machine has two components:
    
    • **Organization:** includes the high-level aspects of a computer’s design such as: The memory system, the bus structure, the internal CPU unit which includes implementations of arithmetic, logic, branching, and data transfer operations.
    
    • **Hardware:** Refers to the specifics of the machine such as detailed logic design and packaging technology.

• In general, **Computer Architecture** refers to the above three aspects: Instruction set architecture, organization, and hardware.
The Task of A Computer Designer

• Determine what attributes that are important to the design of the new machine (CPU).

• Design a machine to maximize performance while staying within cost and other constraints and metrics.

• It involves more than instruction set design.
  – Instruction set architecture.
  – CPU Micro-architecture (CPU design).
  – Implementation.

• Implementation of a machine has two components:
  – Organization.
  – Hardware.
Recent Architectural Improvements

- Increased optimization and utilization of cache systems.
- Memory-latency hiding techniques.
- Optimization of pipelined instruction execution.
- Dynamic hardware-based pipeline scheduling.
- Improved handling of pipeline hazards.
- Improved hardware branch prediction techniques.
- Exploiting Instruction-Level Parallelism (ILP) in terms of multiple-instruction issue and multiple hardware functional units.
- Inclusion of special instructions to handle multimedia applications (limited vector processing).
- High-speed bus designs to improve data transfer rates.
Computer System Components

CPU Core
1 GHz - 3.8 GHz
4-way Superscaler
RISC or RISC-core (x86):
  Deep Instruction Pipelines
  Dynamic scheduling
  Multiple FP, integer FUs
  Dynamic branch prediction
  Hardware speculation

SDRAM
PC100/PC133
100-133MHz
64-128 bits wide
2-way interleaved
~900 MBYTES/SEC

Double Date Rate (DDR) SDRAM
PC3200
200 MHz DDR
64-128 bits wide
4-way interleaved
~3.2 GBYTES/SEC
(one 64bit channel)
~6.4 GBYTES/SEC
(two 64bit channels)

RAMbus DRAM (RDRAM)
400MHz DDR
16 bits wide (32 banks)
~1.6 GBYTES/SEC

Current Standard

All Non-blocking caches
L1 16-128K 1-2 way set associative (on chip), separate or unified
L2 256K-2M 4-32 way set associative (on chip) unified
L3 2-16M 8-32 way set associative (off or on chip) unified

Examples:
  Alpha, AMD K7: EV6, 200-400 MHz
  Intel PII, PIII: GTL+ 133 MHz
  Intel P4 800 MHz

CPU
  Caches
  Front Side Bus (FSB)

Memory Controller
  Memory

I/O Buses
  NICs
  Networks
  I/O Devices:

S/32-64 bits wide
133-528 MBYTES/SEC

Example: PCI-X 133MHz 64 bit
1024 MBYTES/SEC

I/O Bus
  Memory Bus
  Off or On-chip

Memory
  Controllers
  Disks
  Displays
  Keyboards
  NICs
  Networks
  I/O Buses
  Memory Controller
  Front Side Bus (FSB)
  Caches
  CPU

Current Standard

Example: PCI, 33-66MHz
32-64 bits wide
133-528 MBYTES/SEC

North Bridge
  South Bridge
  Chipset

RAMbus DRAM (RDRAM)
400MHz DDR
16 bits wide (32 banks)
~1.6 GBYTES/SEC
CPU Performance Evaluation: Cycles Per Instruction (CPI)

- Most computers run synchronously utilizing a CPU clock running at a constant clock rate:
  
  where: Clock rate = 1 / clock cycle

- The CPU clock rate depends on the specific CPU organization (design) and hardware implementation technology (VLSI) used.

- A computer machine (ISA) instruction is comprised of a number of elementary or micro operations which vary in number and complexity depending on the instruction and the exact CPU organization (Design).
  - A micro operation is an elementary hardware operation that can be performed during one CPU clock cycle.
  - This corresponds to one micro-instruction in microprogrammed CPUs.
  - Examples: register operations: shift, load, clear, increment, ALU operations: add, subtract, etc.

- Thus a single machine instruction may take one or more CPU cycles to complete termed as the Cycles Per Instruction (CPI).

- Average CPI of a program: The average CPI of all instructions executed in the program on a given CPU design.

(From 550)
Computer Performance Measures: Program Execution Time

• For a specific program compiled to run on a specific machine (CPU) “A”, the following parameters are provided:
  – The total instruction count of the program.
  – The average number of cycles per instruction (average CPI).
  – Clock cycle of machine “A”

• How can one measure the performance of this machine running this program?
  – Intuitively the machine is said to be faster or has better performance running this program if the total execution time is shorter.
  – Thus the inverse of the total measured program execution time is a possible performance measure or metric:
    \[ \text{Performance}_A = \frac{1}{\text{Execution Time}_A} \]

How to compare performance of different machines?
What factors affect performance? How to improve performance?
Comparing Computer Performance Using Execution Time

- To compare the performance of two machines (or CPUs) “A”, “B” running a given specific program:
  \[
  \text{Performance}_A = \frac{1}{\text{Execution Time}_A} \\
  \text{Performance}_B = \frac{1}{\text{Execution Time}_B}
  \]

- Machine A is \( n \) times faster than machine B means (or slower? if \( n < 1 \)):

\[
\text{Speedup} = n = \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}
\]

- Example:
  (i.e. Speedup is ratio of performance, no units)

For a given program:

Execution time on machine A: \( \text{Execution}_A = 1 \) second
Execution time on machine B: \( \text{Execution}_B = 10 \) seconds

\[
\text{Speedup} = \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}
\]

\[
\text{Performance}_B = 10 / 1 = 10
\]

The performance of machine A is 10 times the performance of machine B when running this program, or: Machine A is said to be 10 times faster than machine B when running this program.

The two CPUs may target different ISAs provided the program is written in a high level language (HLL)
CPU Execution Time: The CPU Equation

- A program is comprised of a number of instructions executed, \( I \)
  - Measured in: instructions/program

- The average instruction executed takes a number of cycles per instruction (CPI) to be completed.
  - Measured in: cycles/instruction, CPI

- CPU has a fixed clock cycle time \( C = 1/\text{clock rate} \)
  - Measured in: seconds/cycle

- CPU execution time is the product of the above three parameters as follows:

\[
T = I \times \text{CPI} \times C
\]

\( T \) is the CPU execution time per program in seconds

\( I \) is the number of instructions executed

\( \text{CPI} \) is the average CPI for the program

\( C \) is the CPU clock cycle

(This equation is commonly known as the CPU performance equation)

(From 550)
CPU Execution Time: Example

- A Program is running on a specific machine with the following parameters:
  - Total executed instruction count: 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz. (clock cycle = 5x10^-9 seconds)
- What is the execution time for this program:

  \[
  \text{CPU time} = \frac{\text{Instruction count} \times \text{CPI} \times \text{Clock cycle}}{	ext{Clock rate}}
  \]

  \[
  = \frac{10,000,000 \times 2.5 \times 1}{5 \times 10^{-9}}
  \]

  \[
  = 10,000,000 \times 2.5 \times 5 \times 10^{-9}
  \]

  \[
  = 0.125 \text{ seconds}
  \]

(From 550)
Aspects of CPU Execution Time

CPU Time = Instruction count \times CPI \times Clock cycle

\[ T = I \times CPI \times C \]

Instruction Count (executed)

CPI (Average CPI)

Clock Cycle C

Depends on:
- Program Used
- Compiler
- ISA
- CPU Organization

Depends on:
- Program Used
- Compiler
- ISA
- CPU Organization Technology (VLSI)

(From 550)
### Factors Affecting CPU Performance

**CPU time** = Seconds

= Instructions x Cycles x Seconds

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count I</th>
<th>CPI</th>
<th>Clock Cycle C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program</strong></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Instruction Set</strong></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Architecture (ISA)</strong></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Organization</strong></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(CPU Design)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>(VLSI)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ T = I \times \text{CPI} \times C \]

*From 550*
### Performance Comparison: Example

- From the previous example: A Program is running on a specific machine with the following parameters:
  - Total executed instruction count, \( I \): 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz.

- Using the same program with these changes:
  - A new compiler used: New instruction count 9,500,000
    New CPI: 3.0
  - Faster CPU implementation: New clock rate = 300 MHZ

- What is the speedup with the changes?

\[
\text{Speedup} = \frac{I_{\text{old}} \times \text{CPI}_{\text{old}} \times \text{Clock cycle}_{\text{old}}}{I_{\text{new}} \times \text{CPI}_{\text{new}} \times \text{Clock Cycle}_{\text{new}}}
\]

\[
\text{Speedup} = \frac{(10,000,000 \times 2.5 \times 5 \times 10^{-9})}{(9,500,000 \times 3 \times 3.33 \times 10^{-9})} = \frac{.125}{.095} = 1.32
\]

or 32% faster after changes.

(From 550)
Instruction Types & CPI

• Given a program with \( n \) types or classes of instructions executed on a given CPU with the following characteristics:

\[
C_i = \text{Count of instructions of type}_i \\
CPI_i = \text{Cycles per instruction for type}_i \\
i = 1, 2, \ldots, n
\]

Then:

\[
\text{CPI} = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count I}}
\]

Where:

\[
\text{CPU clock cycles} = \sum_{i=1}^{n} \left( CPI_i \times C_i \right)
\]

Instruction Count \( I = \sum C_i \)
**Instruction Types & CPI: An Example**

- An instruction set has three instruction classes:

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

- Two code sequences have the following instruction counts:

<table>
<thead>
<tr>
<th>Code Sequence</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- CPU cycles for sequence 1 = \(2 \times 1 + 1 \times 2 + 2 \times 3 = 10\) cycles

  CPI for sequence 1 = \(\frac{\text{clock cycles}}{\text{instruction count}}\)  
  \(= 10 / 5 = 2\)

- CPU cycles for sequence 2 = \(4 \times 1 + 1 \times 2 + 1 \times 3 = 9\) cycles

  CPI for sequence 2 = \(9 / 6 = 1.5\)
Instruction Frequency & CPI

• Given a program with \( n \) types or classes of instructions with the following characteristics:

\[
C_i = \text{Count of instructions of type}_i \quad i = 1, 2, \ldots, n
\]

\[
CPI_i = \text{Average cycles per instruction of type}_i
\]

\[
F_i = \text{Frequency or fraction of instruction type}_i \text{ executed}
\]

\[
= C_i/ \text{total executed instruction count} = C_i/I
\]

Then:

\[
CPI = \sum_{i=1}^{n} \left( CPI_i \times F_i \right)
\]

Fraction of total execution time for instructions of type \( i \) = \[
\frac{CPI_i \times F_i}{CPI}
\]

(From 550)
### Instruction Type Frequency & CPI: A RISC Example

#### Program Profile or Executed Instructions Mix

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq, F&lt;sub&gt;i&lt;/sub&gt;</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt;</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt; x F&lt;sub&gt;i&lt;/sub&gt;</th>
<th>% Time</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt; x F&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
<td>23% x 5/2.2</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
<td>45% x 1/2.2</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
<td>14% x .3/2.2</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
<td>18% x .4/2.2</td>
</tr>
</tbody>
</table>

**Typical Mix**

\[
CPI = \sum_{i=1}^{n} \left( CPI_i \times F_i \right)
\]

\[
CPI = 0.5 \times 1 + 0.2 \times 5 + 0.1 \times 3 + 0.2 \times 2 = 2.2
\]

\[
= 0.5 + 1 + 0.3 + 0.4
\]

\[
\text{Sum} = 2.2
\]
Each metric has a purpose, and each can be misused.
Choosing Programs To Evaluate Performance

Levels of programs or benchmarks that could be used to evaluate performance:

- **Actual Target Workload:** Full applications that run on the target machine.

- **Real Full Program-based Benchmarks:**
  - Select a specific mix or suite of programs that are typical of targeted applications or workload (e.g. SPEC95, SPEC CPU2000).

- **Small “Kernel” Benchmarks:**
  - Key computationally-intensive pieces extracted from real programs.
    - Examples: Matrix factorization, FFT, tree search, etc.
  - Best used to test specific aspects of the machine.

- **Microbenchmarks:**
  - Small, specially written programs to isolate a specific aspect of performance characteristics: Processing: integer, floating point, local memory, input/output, etc.
Types of Benchmarks

Pros

- Representative

- Portable.
- Widely used.
- Measurements useful in reality.

- Easy to run, early in the design cycle.

- Identify peak performance and potential bottlenecks.

Cons

- Very specific.
- Non-portable.
- Complex: Difficult to run, or measure.

- Less representative than actual workload.

- Easy to “fool” by designing hardware to run them well.

- Peak performance results may be a long way from real application performance.
SPEC: System Performance Evaluation Cooperative

The most popular and industry-standard set of CPU benchmarks.

- **SPECmarks, 1989:**
  - 10 programs yielding a single number (“SPECmarks”).

- **SPEC92, 1992:**
  - SPECInt92 (6 integer programs) and SPECfp92 (14 floating point programs).

- **SPEC95, 1995:**
  - SPECInt95 (8 integer programs):
    - go, m88ksim, gcc, compress, li, ijpeg, perl, vortex
  - SPECfp95 (10 floating-point intensive programs):
    - tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5
  - Performance relative to a Sun SuperSpark I (50 MHz) which is given a score of SPECInt95 = SPECfp95 = 1

- **SPEC CPU2000, 1999:**
  - CINT2000 (11 integer programs). CFP2000 (14 floating-point intensive programs)
  - Performance relative to a Sun Ultra5_10 (300 MHz) which is given a score of SPECInt2000 = SPECfp2000 = 100

All based on execution time and give speedup over a reference CPU
## SPEC CPU2000 Programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Language</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>C</td>
<td>Compression</td>
</tr>
<tr>
<td>175.vpr</td>
<td>C</td>
<td>FPGA Circuit Placement and Routing</td>
</tr>
<tr>
<td>176.gcc</td>
<td>C</td>
<td>C Programming Language Compiler</td>
</tr>
<tr>
<td>181.mcf</td>
<td>C</td>
<td>Combinatorial Optimization</td>
</tr>
<tr>
<td>186.crafty</td>
<td>C</td>
<td>Game Playing: Chess</td>
</tr>
<tr>
<td>197.parser</td>
<td>C</td>
<td>Word Processing</td>
</tr>
<tr>
<td>252.eon</td>
<td>C++</td>
<td>Computer Visualization</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>C</td>
<td>PERL Programming Language</td>
</tr>
<tr>
<td>254.gap</td>
<td>C</td>
<td>Group Theory, Interpreter</td>
</tr>
<tr>
<td>255.vortex</td>
<td>C</td>
<td>Object-oriented Database</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>C</td>
<td>Compression</td>
</tr>
<tr>
<td>300.twolf</td>
<td>C</td>
<td>Place and Route Simulator</td>
</tr>
<tr>
<td>168.wupwise</td>
<td>Fortran 77</td>
<td>Physics / Quantum Chromodynamics</td>
</tr>
<tr>
<td>171.swim</td>
<td>Fortran 77</td>
<td>Shallow Water Modeling</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>Fortran 77</td>
<td>Multi-grid Solver: 3D Potential Field</td>
</tr>
<tr>
<td>173.applu</td>
<td>Fortran 77</td>
<td>Parabolic / Elliptic Partial Differential Equations</td>
</tr>
<tr>
<td>177.mesa</td>
<td>C</td>
<td>3-D Graphics Library</td>
</tr>
<tr>
<td>178.galgel</td>
<td>Fortran 90</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>179.art</td>
<td>C</td>
<td>Image Recognition / Neural Networks</td>
</tr>
<tr>
<td>183.equake</td>
<td>C</td>
<td>Seismic Wave Propagation Simulation</td>
</tr>
<tr>
<td>187facetec</td>
<td>Fortran 90</td>
<td>Image Processing: Face Recognition</td>
</tr>
<tr>
<td>188.ammp</td>
<td>C</td>
<td>Computational Chemistry</td>
</tr>
<tr>
<td>189.lucas</td>
<td>Fortran 90</td>
<td>Number Theory / Primality Testing</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>Fortran 90</td>
<td>Finite-element Crash Simulation</td>
</tr>
<tr>
<td>200.sixtrack</td>
<td>Fortran 77</td>
<td>High Energy Nuclear Physics Accelerator Design</td>
</tr>
<tr>
<td>301.apsi</td>
<td>Fortran 77</td>
<td>Meteorology: Pollutant Distribution</td>
</tr>
</tbody>
</table>

**CINT2000** (Integer)

**CFP2000** (Floating Point)

### Source:
http://www.spec.org/osg/cpu2000/
## Top 20 SPEC CPU2000 Results (As of March 2002)

### Top 20 SPECint2000

<table>
<thead>
<tr>
<th>#</th>
<th>MHz</th>
<th>Processor</th>
<th>int peak</th>
<th>int base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1300</td>
<td>POWER4</td>
<td>814</td>
<td>790</td>
</tr>
<tr>
<td>2</td>
<td>2200</td>
<td>Pentium 4</td>
<td>811</td>
<td>790</td>
</tr>
<tr>
<td>3</td>
<td>2200</td>
<td>Pentium 4 Xeon</td>
<td>810</td>
<td>788</td>
</tr>
<tr>
<td>4</td>
<td>1667</td>
<td>Athlon XP</td>
<td>724</td>
<td>697</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>Alpha 21264C</td>
<td>679</td>
<td>621</td>
</tr>
<tr>
<td>6</td>
<td>1400</td>
<td>Pentium III</td>
<td>664</td>
<td>648</td>
</tr>
<tr>
<td>7</td>
<td>1050</td>
<td>UltraSPARC-III Cu</td>
<td>610</td>
<td>537</td>
</tr>
<tr>
<td>8</td>
<td>1533</td>
<td>Athlon MP</td>
<td>609</td>
<td>587</td>
</tr>
<tr>
<td>9</td>
<td>750</td>
<td>PA-RISC 8700</td>
<td>604</td>
<td>568</td>
</tr>
<tr>
<td>10</td>
<td>833</td>
<td>Alpha 21264B</td>
<td>571</td>
<td>497</td>
</tr>
<tr>
<td>11</td>
<td>1400</td>
<td>Athlon</td>
<td>554</td>
<td>495</td>
</tr>
<tr>
<td>12</td>
<td>833</td>
<td>Alpha 21264A</td>
<td>533</td>
<td>511</td>
</tr>
<tr>
<td>13</td>
<td>600</td>
<td>MIPS R14000</td>
<td>500</td>
<td>483</td>
</tr>
<tr>
<td>14</td>
<td>675</td>
<td>SPARC64 GP</td>
<td>478</td>
<td>449</td>
</tr>
<tr>
<td>15</td>
<td>900</td>
<td>UltraSPARC-III</td>
<td>467</td>
<td>438</td>
</tr>
<tr>
<td>16</td>
<td>552</td>
<td>PA-RISC 8600</td>
<td>441</td>
<td>417</td>
</tr>
<tr>
<td>17</td>
<td>750</td>
<td>POWER RS64-IV</td>
<td>439</td>
<td>409</td>
</tr>
<tr>
<td>18</td>
<td>700</td>
<td>Pentium III4-IV</td>
<td>438</td>
<td>431</td>
</tr>
<tr>
<td>19</td>
<td>800</td>
<td>Itanium</td>
<td>365</td>
<td>358</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>MIPS R12000</td>
<td>353</td>
<td>328</td>
</tr>
</tbody>
</table>

### Top 20 SPECfp2000

<table>
<thead>
<tr>
<th>MHz</th>
<th>Processor</th>
<th>fp peak</th>
<th>fp base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1300</td>
<td>POWER4</td>
<td>1169</td>
<td>1098</td>
</tr>
<tr>
<td>1000</td>
<td>Alpha 21264C</td>
<td>960</td>
<td>776</td>
</tr>
<tr>
<td>1050</td>
<td>UltraSPARC-III Cu</td>
<td>827</td>
<td>701</td>
</tr>
<tr>
<td>2200</td>
<td>Pentium 4 Xeon</td>
<td>802</td>
<td>779</td>
</tr>
<tr>
<td>2200</td>
<td>Pentium 4</td>
<td>801</td>
<td>779</td>
</tr>
<tr>
<td>833</td>
<td>Alpha 21264B</td>
<td>784</td>
<td>643</td>
</tr>
<tr>
<td>800</td>
<td>Itanium</td>
<td>701</td>
<td>701</td>
</tr>
<tr>
<td>833</td>
<td>Alpha 21264A</td>
<td>644</td>
<td>571</td>
</tr>
<tr>
<td>1667</td>
<td>Athlon XP</td>
<td>642</td>
<td>596</td>
</tr>
<tr>
<td>750</td>
<td>PA-RISC 8700</td>
<td>581</td>
<td>526</td>
</tr>
<tr>
<td>750</td>
<td>PA-RISC 8700</td>
<td>547</td>
<td>504</td>
</tr>
<tr>
<td>600</td>
<td>MIPS R14000</td>
<td>529</td>
<td>499</td>
</tr>
<tr>
<td>675</td>
<td>SPARC64 GP</td>
<td>509</td>
<td>371</td>
</tr>
<tr>
<td>900</td>
<td>UltraSPARC-III</td>
<td>482</td>
<td>427</td>
</tr>
<tr>
<td>1400</td>
<td>Athlon</td>
<td>458</td>
<td>426</td>
</tr>
<tr>
<td>1400</td>
<td>Pentium III</td>
<td>456</td>
<td>437</td>
</tr>
<tr>
<td>1400</td>
<td>Pentium III</td>
<td>440</td>
<td>397</td>
</tr>
<tr>
<td>450</td>
<td>POWER3-II</td>
<td>433</td>
<td>426</td>
</tr>
<tr>
<td>500</td>
<td>Alpha 21264</td>
<td>422</td>
<td>383</td>
</tr>
<tr>
<td>400</td>
<td>MIPS R12000</td>
<td>407</td>
<td>382</td>
</tr>
</tbody>
</table>

Source: [http://www.aceshardware.com/SPECmine/top.jsp](http://www.aceshardware.com/SPECmine/top.jsp)
Computer Performance Measures: 
MIPS (Million Instructions Per Second) Rating

- For a specific program running on a specific CPU the MIPS rating is a measure of how many millions of instructions are executed per second:

\[
\text{MIPS Rating} = \frac{\text{Instruction count}}{(\text{Execution Time} \times 10^6)}
= \frac{\text{Instruction count}}{(\text{CPU clocks} \times \text{Cycle time} \times 10^6)}
= \frac{(\text{Instruction count} \times \text{Clock rate})}{(\text{Instruction count} \times \text{CPI} \times 10^6)}
= \frac{\text{Clock rate}}{(\text{CPI} \times 10^6)}
\]

- Major problem with MIPS rating: As shown above the MIPS rating does not account for the count of instructions executed (I).
  - A higher MIPS rating in many cases may not mean higher performance or better execution time. i.e. due to compiler design variations.

- In addition the MIPS rating:
  - Does not account for the instruction set architecture (ISA) used.
    - Thus it cannot be used to compare computers/CPUs with different instruction sets.
  - **Easy to abuse:** Program used to get the MIPS rating is often omitted.
    - Often the Peak MIPS rating is provided for a given CPU which is obtained using a program comprised entirely of instructions with the lowest CPI for the given CPU design which does not represent real programs.

(From 550)
Computer Performance Measures: MIPS (Million Instructions Per Second) Rating

- Under what conditions can the MIPS rating be used to compare performance of different CPUs?

- The MIPS rating is only valid to compare the performance of different CPUs provided that the following conditions are satisfied:
  1. **The same program is used**
     (actually this applies to all performance metrics)
  2. **The same ISA is used**
  3. **The same compiler is used**

    ⇒ (Thus the resulting programs used to run on the CPUs and obtain the MIPS rating are identical at the machine code level including the same instruction count)

(From 550)
Compiler Variations, MIPS, Performance: An Example

- For the machine with instruction classes:

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

- For a given program two compilers produced the following instruction counts:

<table>
<thead>
<tr>
<th>Instruction counts (in millions) for each instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code from:</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Compiler 1</td>
</tr>
<tr>
<td>Compiler 2</td>
</tr>
</tbody>
</table>

- The machine is assumed to run at a clock rate of 100 MHz

(From 550)
Compiler Variations, MIPS, Performance: An Example (Continued)

\[
\text{MIPS} = \frac{\text{Clock rate}}{(\text{CPI} \times 10^6)} = \frac{100 \text{ MHz}}{(\text{CPI} \times 10^6)}
\]

\[
\text{CPI} = \frac{\text{CPU execution cycles}}{\text{Instructions count}}
\]

\[
\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)
\]

\[
\text{CPU time} = \text{Instruction count} \times \text{CPI} / \text{Clock rate}
\]

• For compiler 1:
  – \(\text{CPI}_1 = \frac{(5 \times 1 + 1 \times 2 + 1 \times 3)}{(5 + 1 + 1)} = \frac{10}{7} = 1.43\)
  – \(\text{MIPS Rating}_1 = \frac{100}{(1.428 \times 10^6)} = 70.0\)
  – \(\text{CPU time}_1 = \frac{((5 + 1 + 1) \times 10^6 \times 1.43)}{(100 \times 10^6)} = 0.10 \text{ seconds}\)

• For compiler 2:
  – \(\text{CPI}_2 = \frac{(10 \times 1 + 1 \times 2 + 1 \times 3)}{(10 + 1 + 1)} = \frac{15}{12} = 1.25\)
  – \(\text{MIPS Rating}_2 = \frac{100}{(1.25 \times 10^6)} = 80.0\)
  – \(\text{CPU time}_2 = \frac{((10 + 1 + 1) \times 10^6 \times 1.25)}{(100 \times 10^6)} = 0.15 \text{ seconds}\)

MIPS rating indicates that compiler 2 is better while in reality the code produced by compiler 1 is faster.
MIPS32 (The ISA not the metric) **Loop Performance Example**

For the loop:

```plaintext
for (i=0; i<1000; i=i+1) {
    x[i] = x[i] + s;
}
```

MIPS32 assembly code is given by:

```plaintext
lw     $3,  8($1)       ; load s in $3
addi   $6,  $2,  4000   ; $6 = address of last element + 4

loop: lw     $4,  0($2)       ; load x[i] in $4
      add    $5,  $4, $3      ; $5 has x[i] + s
      sw     $5,  0($2)       ; store computed x[i]
      addi   $2,  $2,  4      ; increment $2 to point to next x[ ] element
      bne    $6,  $2,  loop   ; last loop iteration reached?
```

The MIPS code is executed on a specific CPU that runs at 500 MHz (clock cycle = 2ns = 2x10^-9 seconds) with following instruction type CPIs:

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>4</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
</tr>
<tr>
<td>Store</td>
<td>7</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
</tr>
</tbody>
</table>

For this MIPS code running on this CPU find:

1- Fraction of total instructions executed for each instruction type
2- Total number of CPU cycles
3- Average CPI
4- Fraction of total execution time for each instructions type
5- Execution time
6- MIPS rating, peak MIPS rating for this CPU

X[ ] array of words in memory, base address in $2, s a constant word value in memory, address in $1
MIPS32 (The ISA) Loop Performance Example (continued)

- The code has 2 instructions before the loop and 5 instructions in the body of the loop which iterates 1000 times,
- Thus: Total instructions executed, $I = 5 \times 1000 + 2 = 5002$ instructions

1. Number of instructions executed/fraction $F_i$ for each instruction type:
   - ALU instructions = $1 + 2 \times 1000 = 2001$  
     CPI$_{ALU} = 4$  
     Fraction$_{ALU} = F_{ALU} = 2001/5002 = 0.4 = 40\%$
   - Load instructions = $1 + 1 \times 1000 = 1001$  
     CPI$_{Load} = 5$  
     Fraction$_{Load} = F_{Load} = 1001/5002 = 0.2 = 20\%$
   - Store instructions = 1000  
     CPI$_{Store} = 7$  
     Fraction$_{Store} = F_{Store} = 1000/5002 = 0.2 = 20\%$
   - Branch instructions = 1000  
     CPI$_{Branch} = 3$  
     Fraction$_{Branch} = F_{Branch} = 1000/5002 = 0.2 = 20\%$

2. $CPU$ clock cycles
   \[
   CPU\, clock\, cycles = \sum_{i=1}^{n} \left( CPI_i \times C_i \right) 
   \]
   \[
   = 2001 \times 4 + 1001 \times 5 + 1000 \times 7 + 1000 \times 3 = 23009 \text{ cycles}
   \]

3. Average CPI = $CPU$ clock cycles / $I = 23009/5002 = 4.6$

4. Fraction of execution time for each instruction type:
   - Fraction of time for ALU instructions = $CPI_{ALU} \times F_{ALU} / CPI = 4 \times 0.4/4.6 = 0.348 = 34.8\%$
   - Fraction of time for load instructions = $CPI_{Load} \times F_{Load} / CPI = 5 \times 0.2/4.6 = 0.217 = 21.7\%$
   - Fraction of time for store instructions = $CPI_{Store} \times F_{Store} / CPI = 7 \times 0.2/4.6 = 0.304 = 30.4\%$
   - Fraction of time for branch instructions = $CPI_{Branch} \times F_{Branch} / CPI = 3 \times 0.2/4.6 = 0.13 = 13\%$

5. Execution time = $I \times CPI \times C = CPU$ cycles $\times C = 23009 \times 2 \times 10^{-9} =$
   \[
   = 4.6 \times 10^{-5} \text{ seconds} = 0.046 \text{ msec} = 46 \text{ usec}
   \]

6. MIPS rating = Clock rate / ($CPI \times 10^6$) = $500 / 4.6 = 108.7$ MIPS
   - The CPU achieves its peak MIPS rating when executing a program that only has instructions of the type with the lowest CPI. In this case branches with $CPI_{Branch} = 3$
   - Peak MIPS rating = Clock rate / ($CPI_{Branch} \times 10^6$) = $500/3 = 166.67$ MIPS

(From 550)
Computer Performance Measures:

**MFLOPS** (Million FLOating-Point Operations Per Second)

- A floating-point operation is an addition, subtraction, multiplication, or division operation applied to numbers represented by a single or a double precision floating-point representation.
- MFLOPS, for a specific program running on a specific computer, is a measure of millions of floating-point operations (megaflops) per second:

\[
MFLOPS = \frac{\text{Number of floating-point operations}}{(\text{Execution time} \times 10^6)}
\]

- MFLOPS rating is a better comparison measure between different machines (applies even if ISAs are different) than the MIPS rating.
  - Applicable even if ISAs are different
- Program-dependent: Different programs have different percentages of floating-point operations present. i.e compilers have no floating-point operations and yield a MFLOPS rating of zero.
- Dependent on the type of floating-point operations present in the program.
  - **Peak MFLOPS rating for a CPU**: Obtained using a program comprised entirely of the simplest floating point instructions (with the lowest CPI) for the given CPU design which does not represent real floating point programs.

(From 550)
Quantitative Principles of Computer Design

- Amdahl’s Law:

  The performance gain from improving some portion of a computer is calculated by:

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

  or

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

(From 550)
Performance Enhancement Calculations: Amdahl's Law

- The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used.

- Amdahl’s Law:

  Performance improvement or speedup due to enhancement E:

  \[
  \text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E} = \frac{\text{Performance without } E}{\text{Performance with } E}
  \]

  Suppose that enhancement E accelerates a fraction F of the execution time by a factor S and the remainder of the time is unaffected then:

  Execution Time with E = \((1 - F) + \frac{F}{S}\) \times \text{Execution Time without } E

  Hence speedup is given by:

  \[
  \text{Speedup}(E) = \frac{\text{Execution Time without } E}{((1 - F) + \frac{F}{S}) \times \text{Execution Time without } E} = \frac{1}{(1 - F) + \frac{F}{S}}
  \]

F (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied.
Pictorial Depiction of Amdahl’s Law

Enhancement E accelerates fraction F of original execution time by a factor of S

Before:
Execution Time without enhancement E: (Before enhancement is applied)

- shown normalized to $1 = (1-F) + \frac{F}{S} = 1$

<table>
<thead>
<tr>
<th>Unaffected fraction: $(1-F)$</th>
<th>Affected fraction: $F$</th>
</tr>
</thead>
</table>

Unchanged

| Unaffected fraction: $(1-F)$ | $\frac{F}{S}$ |

After:
Execution Time with enhancement E:

$$\text{Speedup}(E) = \frac{\text{Execution Time without enhancement } E}{\text{Execution Time with enhancement } E} = \frac{1}{(1-F) + \frac{F}{S}}$$

(From 550)
Performance Enhancement Example

- For the RISC machine with the following instruction mix given earlier:

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

CPI = 2.2

- If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:

Fraction enhanced = F = 45% or .45
Unaffected fraction = 100% - 45% = 55% or .55
Factor of enhancement = 5/2 = 2.5

Using Amdahl’s Law:

\[
\text{Speedup}(E) = \frac{1}{(1 - F) + F/S} = \frac{1}{.55 + .45/2.5} = 1.37
\]

(From 550)
An Alternative Solution Using CPU Equation

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

CPI = 2.2

If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:

Old CPI = 2.2
New CPI = \(0.5 \times 1 + 0.2 \times 2 + 0.1 \times 3 + 0.2 \times 2 = 1.6\)

\[
\text{Speedup(E)} = \frac{\text{Original Execution Time}}{\text{New Execution Time}} = \frac{\text{Instruction count} \times \text{old CPI} \times \text{clock cycle}}{\text{Instruction count} \times \text{new CPI} \times \text{clock cycle}}
\]

\[
= \frac{\text{old CPI}}{\text{new CPI}} = \frac{2.2}{1.6} = 1.37
\]

Which is the same speedup obtained from Amdahl’s Law in the first solution.
Performance Enhancement Example

- A program runs in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program four times faster?

\[
\text{Desired speedup} = 4 = \frac{100}{\text{Execution Time with enhancement}}
\]

→ Execution time with enhancement = 25 seconds

\[
25 \text{ seconds} = (100 - 80 \text{ seconds}) + \frac{80 \text{ seconds}}{n}
\]

→ \[ 25 \text{ seconds} = 20 \text{ seconds} + \frac{80 \text{ seconds}}{n} \]

→ \[ 5 = \frac{80 \text{ seconds}}{n} \]

→ \[ n = \frac{80}{5} = 16 \]

Hence multiplication should be 16 times faster to get a speedup of 4.
Performance Enhancement Example

• For the previous example with a program running in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program five times faster?

\[
\text{Desired speedup} = 5 = \frac{100}{\text{Execution Time with enhancement}}
\]

→ Execution time with enhancement = 20 seconds

\[
20 \text{ seconds} = (100 - 80 \text{ seconds}) + \frac{80 \text{ seconds}}{n}
\]

\[
20 \text{ seconds} = \frac{20 \text{ seconds}}{n} + \frac{80 \text{ seconds}}{n}
\]

→ 0 = \frac{80 \text{ seconds}}{n}

No amount of multiplication speed improvement can achieve this.

(From 550)
Extending Amdahl's Law To Multiple Enhancements

• Suppose that enhancement $E_i$ accelerates a fraction $F_i$ of the execution time by a factor $S_i$ and the remainder of the time is unaffected then:

\[
\text{Speedup} = \frac{\text{Original Execution Time}}{\left( (1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i} \right) \times \text{Original Execution Time}}
\]

\[
\text{Speedup} = \frac{1}{\left( (1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i} \right)}
\]

Note: All fractions $F_i$ refer to original execution time before the enhancements are applied.
Amdahl's Law With Multiple Enhancements: Example

- Three CPU performance enhancements are proposed with the following speedups and percentage of the code execution time affected:
  - Speedup_1 = S_1 = 10, Percentage_1 = F_1 = 20%
  - Speedup_2 = S_2 = 15, Percentage_2 = F_2 = 15%
  - Speedup_3 = S_3 = 30, Percentage_3 = F_3 = 10%

- While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.

- What is the resulting overall speedup?

\[
\text{Speedup} = \frac{1}{\left(1 - \sum_i F_i + \sum_i \frac{F_i}{S_i}\right)}
\]

- Speedup = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30]
  \[= 1 / [ .55 + .0333 ]\]
  \[= 1 / .5833 = 1.71\]
Pictorial Depiction of Example

Before:
Execution Time with no enhancements: 1

After:
Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

Note: All fractions \(F_i, i = 1, 2, 3\) refer to original execution time.
Instruction Set Architecture (ISA)

“... 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.

The instruction set architecture is concerned with:

- Organization of programmable storage (memory & registers): Includes the amount of addressable memory and number of available registers.
- Data Types & Data Structures: Encodings & representations.
- Instruction Set: What operations are specified.
- Instruction formats and encoding.
- Modes of addressing and accessing data items and instructions
- Exceptional conditions.
Evolution of Instruction Sets

- Single Accumulator (EDSAC 1950)
  - Accumulator + Index Registers
    (Manchester Mark I, IBM 700 series 1953)

- Separation of Programming Model from Implementation
- High-level Language Based
  (B5000 1963)
- Concept of a Family
  (IBM 360 1964)
- 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)
Types of Instruction Set Architectures According To Operand Addressing Fields

Memory-To-Memory Machines:
- Operands obtained from memory and results stored back in memory by any instruction that requires operands.
- No local CPU registers are used in the CPU datapath.
- Include:
  - The 4 Address Machine.
  - The 3-address Machine.
  - The 2-address Machine.

The 1-address (Accumulator) Machine:
- A single local CPU special-purpose register (accumulator) is used as the source of one operand and as the result destination.

The 0-address or Stack Machine:
- A push-down stack is used in the CPU.

General Purpose Register (GPR) Machines:
- The CPU datapath contains several local general-purpose registers which can be used as operand sources and as result destinations.
- A large number of possible addressing modes.
- Load-Store or Register-To-Register Machines: GPR machines where only data movement instructions (loads, stores) can obtain operands from memory and store results to memory.

CISC to RISC observation (load-store simplifies CPU design)
General-Purpose Register (GPR) Machines

- Every ISA designed after 1980 uses a load-store GPR architecture (i.e. RISC, to simplify CPU design).
- Registers, like any other storage form internal to the CPU, are faster than memory.
- Registers are easier for a compiler to use.
- GPR architectures are divided into several types depending on two factors:
  - Whether an ALU instruction has two or three operands.
  - How many of the operands in ALU instructions may be memory addresses.
### ISA Examples

<table>
<thead>
<tr>
<th>Machine</th>
<th>Number of General Purpose Registers</th>
<th>Architecture</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDSAC</td>
<td>1</td>
<td>accumulator</td>
<td>1949</td>
</tr>
<tr>
<td>IBM 701</td>
<td>1</td>
<td>accumulator</td>
<td>1953</td>
</tr>
<tr>
<td>CDC 6600</td>
<td>8</td>
<td>load-store</td>
<td>1963</td>
</tr>
<tr>
<td>IBM 360</td>
<td>16</td>
<td>register-memory</td>
<td>1964</td>
</tr>
<tr>
<td>DEC PDP-8</td>
<td>1</td>
<td>accumulator</td>
<td>1965</td>
</tr>
<tr>
<td>DEC PDP-11</td>
<td>8</td>
<td>register-memory</td>
<td>1970</td>
</tr>
<tr>
<td>Intel 8008</td>
<td>1</td>
<td>accumulator</td>
<td>1972</td>
</tr>
<tr>
<td>Motorola 6800</td>
<td>1</td>
<td>accumulator</td>
<td>1974</td>
</tr>
<tr>
<td>DEC VAX</td>
<td>16</td>
<td>register-memory</td>
<td>1977</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memory-memory</td>
<td></td>
</tr>
<tr>
<td>Intel 8086</td>
<td>1</td>
<td>extended accumulator</td>
<td>1978</td>
</tr>
<tr>
<td>Motorola 68000</td>
<td>16</td>
<td>register-memory</td>
<td>1980</td>
</tr>
<tr>
<td>Intel 80386</td>
<td>8</td>
<td>register-memory</td>
<td>1985</td>
</tr>
<tr>
<td>MIPS</td>
<td>32</td>
<td>load-store</td>
<td>1985</td>
</tr>
<tr>
<td>HP PA-RISC</td>
<td>32</td>
<td>load-store</td>
<td>1986</td>
</tr>
<tr>
<td>SPARC</td>
<td>32</td>
<td>load-store</td>
<td>1987</td>
</tr>
<tr>
<td>PowerPC</td>
<td>32</td>
<td>load-store</td>
<td>1992</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>32</td>
<td>load-store</td>
<td>1992</td>
</tr>
<tr>
<td>HP/Intel IA-64</td>
<td>128</td>
<td>load-store</td>
<td>2001</td>
</tr>
<tr>
<td>AMD64 (EMT64)</td>
<td>16</td>
<td>register-memory</td>
<td>2003</td>
</tr>
</tbody>
</table>
## Typical Memory Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Sample Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Add R4, #3</td>
<td>Regs[R4] ← Regs[R4] + 3</td>
</tr>
<tr>
<td>Absolute</td>
<td>Add R1, (1001)</td>
<td>Regs[R1] ← Regs[R1] + Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1, @ (R3)</td>
<td>Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1, (R2) +</td>
<td>Regs[R1] ← Regs[R1] + Mem[Regs[R2]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Regs[R2] ← Regs[R2] + d</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1, - (R2)</td>
<td>Regs[R2] ← Regs[R2] - d</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Regs[R1] ← Regs[Regs[R1] + Mem[Regs[R2]]]</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>
</tr>
</tbody>
</table>

**For GPR ISAs**
# Addressing Modes Usage Example

For 3 programs running on VAX ignoring direct register mode:

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Avg Usage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displacement</td>
<td>42%</td>
<td>32% to 55%</td>
</tr>
<tr>
<td>Immediate</td>
<td>33%</td>
<td>17% to 43%</td>
</tr>
<tr>
<td>Register deferred (indirect)</td>
<td>13%</td>
<td>3% to 24%</td>
</tr>
<tr>
<td>Scaled</td>
<td>7%</td>
<td>0% to 16%</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>3%</td>
<td>1% to 6%</td>
</tr>
<tr>
<td>Misc</td>
<td>2%</td>
<td>0% to 3%</td>
</tr>
</tbody>
</table>

75% displacement & immediate
88% displacement, immediate & register indirect.

Observation: In addition Register direct, Displacement, Immediate, Register Indirect addressing modes are important.

CISC to RISC observation
(fewer addressing modes simplify CPU design)
Displacement Address Size Example

Avg. of 5 SPECint92 programs v. avg. 5 SPECfp92 programs

1% of addresses > 16-bits
12 - 16 bits of displacement needed

CISC to RISC observation
### Operation Types in The Instruction Set

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>Integer arithmetic and logical operations: add, or</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Loads-stores (move on machines with memory addressing)</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call, and return, traps.</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating point operations: add, multiply.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, decimal multiply, decimal to character conversion</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
<tr>
<td>Media</td>
<td>The same operation performed on multiple data (e.g Intel MMX, SSE)</td>
</tr>
</tbody>
</table>
Instruction Usage Example: Top 10 Intel X86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent</th>
<th>Total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>96%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observation: Simple instructions dominate instruction usage frequency.
Instruction Set Encoding

Considerations affecting instruction set encoding:

– To have as many registers and addressing modes as possible.

– The Impact of the size of the register and addressing mode fields on the average instruction size and on the average program.

– To encode instructions into lengths that will be easy to handle in the implementation. On a minimum to be a multiple of bytes.

  • Fixed length encoding: Faster and easiest to implement in hardware.
  • Variable length encoding: Produces smaller instructions.
  • Hybrid encoding.

CISC to RISC observation
### Three Examples of Instruction Set Encoding

<table>
<thead>
<tr>
<th>Operations &amp; no of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier n</th>
<th>Address field n</th>
</tr>
</thead>
</table>

#### Variable: VAX (1-53 bytes)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

#### Fixed: MIPS, PowerPC, SPARC (Each instruction is 4 bytes)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address Specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address Specifier 1</th>
<th>Address Specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address Specifier</th>
<th>Address field 1</th>
<th>Address field 2</th>
</tr>
</thead>
</table>

#### Hybrid: IBM 360/370, Intel 80x86
Complex Instruction Set Computer (CISC)

- Emphasizes doing more with each instruction:
  - Thus fewer instructions per program (more compact code).

- Motivated by the high cost of memory and hard disk capacity when original CISC architectures were proposed
  - When M6800 was introduced: 16K RAM = $500, 40M hard disk = $55,000
  - When MC68000 was introduced: 64K RAM = $200, 10M HD = $5,000

- Original CISC architectures evolved with faster more complex CPU designs but backward instruction set compatibility had to be maintained.

- Wide variety of addressing modes:
  - 14 in MC68000, 25 in MC68020

- A number instruction modes for the location and number of operands:
  - The VAX has 0- through 3-address instructions.

- Variable-length instruction encoding.
Example CISC ISA:  
Motorola 680X0

18 addressing modes:

- Data register direct.
- Address register direct.
- Immediate.
- Absolute short.
- Absolute long.
- Address register indirect.
- Address register indirect with postincrement.
- Address register indirect with predecrement.
- Address register indirect with displacement.
- Address register indirect with index (8-bit).
- Address register indirect with index (base).
- Memory indirect postindexed.
- Memory indirect preindexed.
- Program counter indirect with index (8-bit).
- Program counter indirect with index (base).
- Program counter indirect with displacement.
- Program counter memory indirect postindexed.
- Program counter memory indirect preindexed.

GPR ISA (Register-Memory)

Operand size:

- Range from 1 to 32 bits, 1, 2, 4, 8, 10, or 16 bytes.

Instruction Encoding:

- Instructions are stored in 16-bit words.
- the smallest instruction is 2- bytes (one word).
- The longest instruction is 5 words (10 bytes) in length.
Example CISC ISA:

**Intel IA-32, X86 (80386)**

GPR ISA (Register-Memory)

Operand sizes:
• Can be 8, 16, 32, 48, 64, or 80 bits long.
• Also supports string operations.

Instruction Encoding:
• The smallest instruction is one byte.
• The longest instruction is 12 bytes long.
• The first bytes generally contain the opcode, mode specifiers, and register fields.
• The remainder bytes are for address displacement and immediate data.

12 addressing modes:
• Register.
• Immediate.
• Direct.
• Base.
• Base + Displacement.
• Index + Displacement.
• Scaled Index + Displacement.
• Based Index.
• Based Scaled Index.
• Based Index + Displacement.
• Based Scaled Index + Displacement.
• Relative.
Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the machine.
- Reduced CPI. Goal: At least one instruction per clock cycle. (CPI = 1 or less)
- Designed with pipelining in mind.
- Fixed-length instruction encoding.
- Only load and store instructions access memory. (Thus more instructions executed than CISC)
- Simplified addressing modes.
  - Usually limited to immediate, register indirect, register displacement, indexed.
- Delayed loads and branches.
- Instruction pre-fetch and speculative execution.
- Examples: MIPS, SPARC, PowerPC, Alpha

~1983
Example RISC ISA:

HP Precision Architecture,
HP PA-RISC

7 addressing modes:

• Register
• Immediate
• Base with displacement
• Base with scaled index and displacement
• Predecrement
• Postincrement
• PC-relative

Operand sizes:

• Five operand sizes ranging in powers of two from 1 to 16 bytes.

Instruction Encoding:

• Instruction set has 12 different formats.
• All are 32 bits in length.
RISC ISA Example:

**MIPS R3000 (32-bits)**

**Instruction Categories:**
- Load/Store.
- Computational.
- Jump and Branch.
- Floating Point (using coprocessor).
- Memory Management.
- Special.

**Instruction Encoding:** 3 Instruction Formats, all 32 bits wide.

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump target</td>
</tr>
</tbody>
</table>

**Register:**
- R0 - R31
- PC
- HI
- LO

**5 Addressing Modes:**
- Register direct (arithmetic).
- Immediate (arithmetic).
- Base register + immediate offset (loads and stores).
- PC relative (branches).
- Pseudodirect (jumps)

**Operand Sizes:**
- Memory accesses in any multiple between 1 and 4 bytes.

(Used as target ISA for CPU design in 550)
# A RISC ISA Example: MIPS

## Register-Register

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td>sa</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

## Register-Immediate

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

## Branch

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>displacement</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

## Jump / Call

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>
```
An Instruction Set Example: MIPS64

• A RISC-type 64-bit instruction set architecture based on instruction set design considerations of chapter 2:
  
  – Use general-purpose registers with a load/store architecture to access memory.
  
  – Reduced number of addressing modes: displacement (offset size of 16 bits), immediate (16 bits).
  
  – Data sizes: 8 (byte), 16 (half word), 32 (word), 64 (double word) bit integers and 32-bit or 64-bit IEEE 754 floating-point numbers.
  
  – Use fixed instruction encoding (32 bits) for performance.
  
  – 32, 64-bit general-purpose integer registers GPRs, R0, ..., R31. R0 always has a value of zero.
  
  – Separate 32, 64-bit floating point registers FPRs: F0, F1 ... F31. When holding a 32-bit single-precision number the upper half of the FPR is not used.

(64-bit version of 32-bit MIPS ISA used in 550)
MIPS64 Instruction Format

I - type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immediate</th>
</tr>
</thead>
</table>

Encodes: Loads and stores of bytes, words, half words. All immediates (rt ← rs op immediate)
Conditional branch instructions
Jump register, jump and link register (rs = destination, immediate = 0)

R - type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

Register-register ALU operations: rd ← rs func rt Function encodes the data path operation:
Add, Sub .. Read/write special registers and moves.

J - Type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
</table>

Jump and jump and link. Trap and return from exception
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

**Register (direct)**

- First Operand
- Second Operand
- Destination

**Immediate ALU**

**Displacement: Base+index**
(loads/stores)

**PC-relative**

Branches

Pseudodirect Addressing for jumps (J-Type) not shown here
# MIPS64 Instructions: Load and Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1,30(R2)</td>
<td>Load double word</td>
<td>Regs[R1] ←_64 Mem[30+Regs[R2]]</td>
</tr>
<tr>
<td>LW R1, 60(R2)</td>
<td>Load word</td>
<td>Regs[R1] ←_64 Mem[60+Regs[R2]]₀_{32} # # Mem[60+Regs[R2]]</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
<td>Regs[R1] ←_64 Mem[40+Regs[R3]]₀_{56} # # Mem[40+Regs[R3]]</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
<td>Regs[R1] ←_64 Mem[40+Regs[R3]]₀_{56} # # Mem[40+Regs[R3]]</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load half word</td>
<td>Regs[R1] ←_64 Mem[40+Regs[R3]]₀_{48} # # Mem[40+Regs[R3]]</td>
</tr>
<tr>
<td>L.S F0, 50(R3)</td>
<td>Load FP single</td>
<td>Regs[F0] ←_64 Mem[50+Regs[R3]] # # 0_{32}</td>
</tr>
<tr>
<td>L.D F0, 50(R2)</td>
<td>Load FP double</td>
<td>Regs[F0] ←_64 Mem[50+Regs[R2]]</td>
</tr>
<tr>
<td>SD R3,500(R4)</td>
<td>Store double word</td>
<td>Mem[500+Regs[R4]] ←_64 Reg[R3]</td>
</tr>
<tr>
<td>SW R3,500(R4)</td>
<td>Store word</td>
<td>Mem[500+Regs[R4]] ←_32 Reg[R3]</td>
</tr>
<tr>
<td>S.S F0, 40(R3)</td>
<td>Store FP single</td>
<td>Mem[40, Regs[R3]] ←_32 Reg[F0]₀...31</td>
</tr>
<tr>
<td>S.D F0,40(R3)</td>
<td>Store FP double</td>
<td>Mem[40+Regs[R3]] ←_64 Reg[F0]</td>
</tr>
<tr>
<td>SH R3, 502(R2)</td>
<td>Store half</td>
<td>Mem[502+Regs[R2]] ←_16 Regs[R3]₄₈...₆₃</td>
</tr>
<tr>
<td>SB  R2, 41(R3)</td>
<td>Store byte</td>
<td>Mem[41 + Regs[R3]] ←_₈ Regs[R2]₅₆...₆₃</td>
</tr>
</tbody>
</table>
MIPS64 Instructions: Arithmetic/Logical

DADDU R1, R2, R3   Add unsigned   Regs[R1] ← Regs[R2] + Regs[R3]

DADDI R1, R2, #3   Add immediate  Regs[R1] ← Regs[R2] + 3

LUI R1, #42        Load upper immediate Regs[R1] ← 0^{32} ##42 ## 0^{16}

DSLL R1, R2, #5    Shift left logical Regs[R1] ← Regs[R2] << 5

DSLTI R1, R2, R3   Set less than if (regs[R2] < Regs[R3] )
                        Regs[R1] ← 1 else Regs[R1] ← 0
## MIPS64 Instructions: Control-Flow

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>J name</td>
<td>Jump</td>
<td>$PC_{36..63} \leftarrow name$</td>
</tr>
<tr>
<td>JAL name</td>
<td>Jump and link</td>
<td>$Regs[31] \leftarrow PC+4; \quad PC_{36..63} \leftarrow name;\quad ((PC+4)-2^{27}) \leq name &lt; ((PC + 4) + 2^{27})$</td>
</tr>
<tr>
<td>JALR R2</td>
<td>Jump and link register</td>
<td>$Regs[R31] \leftarrow PC+4; \quad PC \leftarrow Regs[R2]$</td>
</tr>
<tr>
<td>JR R3</td>
<td>Jump register</td>
<td>$PC \leftarrow Regs[R3]$</td>
</tr>
<tr>
<td>BEQZ R4, name</td>
<td>Branch equal zero</td>
<td>if (Reg[R4] ==0) $PC \leftarrow name;\quad ((PC+4) - 2^{17}) \leq name &lt; ((PC+4) + 2^{17}$</td>
</tr>
<tr>
<td>BNEZ R4, Name</td>
<td>Branch not equal zero</td>
<td>if (Reg[R4] != 0) $PC \leftarrow name \quad ((PC+4) - 2^{17}) \leq name &lt; ((PC+4) + 2^{17}$</td>
</tr>
<tr>
<td>MOVZ R1,R2,R3</td>
<td>Conditional move if zero</td>
<td>if (Reg[R3] ==0) $Regs[R1] \leftarrow Regs[R2]$</td>
</tr>
</tbody>
</table>
The Role of Compilers

The Structure of Recent Compilers:

Dependencies

Language dependent
machine dependent

Somewhat Language
dependent largely machine
independent

Small language dependencies
machine dependencies slight
(e.g. register counts/types)

Highly machine dependent
language independent

Function:

Front-end per Language

High-level
Optimizations

Global Optimizer

Code generator

Transform Language to Common
intermediate form

For example procedure inlining
and loop transformations

Include global and local
optimizations + register allocation

Detailed instruction selection
and machine-dependent
optimizations; may include or
be followed by assembler

EECC551 - Shaaban
<table>
<thead>
<tr>
<th>Optimization name</th>
<th>Explanation</th>
<th>Percentage of the total number of optimizing transforms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-level</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure integration</td>
<td>Replace procedure call by procedure body</td>
<td>N.M.</td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common subexpression elimination</td>
<td>Replace two instances of the same computation by single copy</td>
<td>18%</td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Replace all instances of a variable that is assigned a constant with the constant</td>
<td>22%</td>
</tr>
<tr>
<td>Stack height reduction</td>
<td>Rearrange expression tree to minimize resources needed for expression evaluation</td>
<td>N.M.</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global common subexpression elimination</td>
<td>Same as local, but this version crosses branches</td>
<td>13%</td>
</tr>
<tr>
<td>Copy propagation</td>
<td>Replace all instances of a variable A that has been assigned X (i.e., A = X) with X</td>
<td>11%</td>
</tr>
<tr>
<td>Code motion</td>
<td>Remove code from a loop that computes same value each iteration of the loop</td>
<td>16%</td>
</tr>
<tr>
<td>Induction variable elimination</td>
<td>Simplify/eliminate array-addressing calculations within loops</td>
<td>2%</td>
</tr>
<tr>
<td><strong>Machine-dependent</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strength reduction</td>
<td>Many examples, such as replace multiply by a constant with adds and shifts</td>
<td>N.M.</td>
</tr>
<tr>
<td>Pipeline scheduling</td>
<td>Reorder instructions to improve pipeline performance</td>
<td>N.M.</td>
</tr>
<tr>
<td>Branch offset optimization</td>
<td>Choose the shortest branch displacement that reaches target</td>
<td>N.M.</td>
</tr>
</tbody>
</table>

*Major Types of Compiler Optimization*
Change in instruction count for the programs lucas and mcf from SPEC2000 as compiler optimizations vary.