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.

(Chapters 1, 2)
Generic CPU Machine Instruction Execution Steps

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

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

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

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

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

6. **Next Instruction**
   - Determine successor or next instruction
Hardware Components of Any Computer

Five classic components of all computers:
1. Control Unit; 2. Datapath; 3. Memory; 4. Input; 5. Output

Processor

Computer

Processor (active)
Control Unit
Datapath

Memory (passive)
(where programs, data live when running)

Devices
Input
Output

Keyboard, Mouse, etc.
Disk
Display, Printer, etc.
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 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, NT) 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.
1988 Computer Food Chain

- Mainframe
- Supercomputer
- Mini-supercomputer
- Mini-computer
- Work-station
- PC
- Massively Parallel Processors
1997 Computer Food Chain

- Mainframe
- Supercomputer
- Server
- Workstation
- PC
- PDA
**Processor Performance Trends**

Mass-produced microprocessors a cost-effective high-performance replacement for custom-designed mainframe/minicomputer CPUs
Microprocessor Performance 1987-97

Integer SPEC92 Performance

DEC Alpha 21264/600
DEC Alpha 5/500
DEC Alpha 5/300
DEC Alpha 4/266
IBM POWER 100
IBM RS/6000
DEC AXP/500
HP 9000/750
IBM MIPS M/120
MIPS M/200
MIPS M/260
Sun -4/260

0 200 400 600 800 1000 1200

87 88 89 90 91 92 93 94 95 96 97
Microprocessor Clock Frequency Trend

- 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)
Microprocessor Transistor Count Growth Rate

Year

Count
100000000
10000000
1000000
100000
10000
1000

Moore’s Law

Currently up to 500 million
Alpha 21264: 15 million
Pentium Pro: 5.5 million
PowerPC 620: 6.9 million
Alpha 21164: 9.3 million
Sparc Ultra: 5.2 million

Moore’s Law:
2X transistors/Chip
Every 1.5 years
(circa 1970)
Increase of Capacity of VLSI Dynamic RAM (DRAM) Chips

<table>
<thead>
<tr>
<th>Year</th>
<th>size (Megabit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>0.0625</td>
</tr>
<tr>
<td>1983</td>
<td>0.25</td>
</tr>
<tr>
<td>1986</td>
<td>1</td>
</tr>
<tr>
<td>1989</td>
<td>4</td>
</tr>
<tr>
<td>1992</td>
<td>16</td>
</tr>
<tr>
<td>1996</td>
<td>64</td>
</tr>
<tr>
<td>1999</td>
<td>256</td>
</tr>
<tr>
<td>2000</td>
<td>1024</td>
</tr>
</tbody>
</table>

1.55X/yr, or doubling every 1.6 years
Microprocessor Cost Drop Over Time
Example: Intel PIII

Intel list price (1000 units)

$1000
$900
$800
$700
$600
$500
$400
$300
$200
$100
$0

May 1999
June 1999
July 1999
Aug. 1999
Sept. 1999
Oct. 1999
Nov. 1999
Dec. 1999
Jan. 2000
Feb. 2000
Mar. 2000
Apr. 2000
May 2000
June 2000
July 2000
Aug. 2000

Date

1000 MHz
867 MHz
733 MHz
600 MHz
500 MHz
450 MHz

EECC551 - Shaaban
#13   Lec # 1 Spring 2004   3-8-2004
DRAM Cost Over Time

Current First Quarter 2004 cost:
~ $0.15 per MB
## Recent Technology Trends (Summary)

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Speed (latency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>2x in 3 years</td>
<td>2x in 3 years</td>
</tr>
<tr>
<td>DRAM</td>
<td>4x in 3 years</td>
<td>2x in 10 years</td>
</tr>
<tr>
<td>Disk</td>
<td>4x in 3 years</td>
<td>2x in 10 years</td>
</tr>
</tbody>
</table>
Computer Technology Trends: Evolutionary but Rapid Change

• **Processor:**
  - 2X in speed every 1.5 years; 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 2004:**
  - Processor clock speed: > 3600 MegaHertz (3.6 Giga Hertz)
  - Memory capacity: > 4000 MegaByte (4 Giga Bytes)
  - Disk capacity: > 300 GigaBytes (0.3 Tera Bytes)
A Simplified View of The Software/Hardware Hierarchical Layers
# A Hierarchy of Computer Design

<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Modules</th>
<th>Primitives</th>
<th>Descriptive Media</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Electronics</td>
<td>Gates, FF’s</td>
<td>Transistors, Resistors, etc.</td>
<td>Circuit Diagrams</td>
</tr>
<tr>
<td>2</td>
<td>Logic</td>
<td>Registers, ALU’s …</td>
<td>Gates, FF’s ….</td>
<td>Logic Diagrams</td>
</tr>
<tr>
<td>3</td>
<td>Organization</td>
<td>Processors, Memories</td>
<td>Registers, ALU’s …</td>
<td>Register Transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Notation (RTN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Microprogramming</td>
<td>Assembly Language</td>
<td>Microinstructions</td>
<td>Microprogram</td>
</tr>
<tr>
<td>5</td>
<td>Assembly language</td>
<td>OS Routines</td>
<td>Assembly language</td>
<td>Assembly Language</td>
</tr>
<tr>
<td></td>
<td>programming</td>
<td></td>
<td>Instructions</td>
<td>Programs</td>
</tr>
<tr>
<td>6</td>
<td>Procedural</td>
<td>Applications</td>
<td>OS Routines</td>
<td>High-level Language</td>
</tr>
<tr>
<td></td>
<td>Programming</td>
<td></td>
<td>High-level Languages</td>
<td>Programs</td>
</tr>
<tr>
<td>7</td>
<td>Application</td>
<td>Systems</td>
<td>Procedural Constructs</td>
<td>Problem-Oriented</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Programs</td>
</tr>
</tbody>
</table>

**Low Level - Hardware**

**High Level - Software**
Hierarchy of Computer Architecture

High-Level Language Programs

Software

Machine Language Program

Software/Hardware Boundary

Hardware

Logic Diagrams

Circuit Diagrams

Assembly Language Programs

Instruction Set Architecture

Microprogram

Register Transfer Notation (RTN)

EECC551 - Shaaban
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.
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.
The Task of A Computer Designer

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

• 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.
  – 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.
- High-speed bus designs to improve data transfer rates.
Computer System Components

CPU Core
1 GHz - 3.2 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 (64 bit)

Double Data 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

CPU
Caches

Memory Bus
Memory

Controllers

Disks
Displays
Keyboards

I/O Buses
Example: PCI, 32-64MHz
32-64 bits wide
133-528 MBYTES/SEC

NICs

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

I/O Devices:

Chipset
North Bridge
South Bridge

I/O Subsystem

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

Input/Output and Storage
- Disks, WORM, Tape

Emerging Technologies
- Interleaving
- Bus protocols

Memory Hierarchy
- DRAM
- L2 Cache
- L1 Cache

VLSI
- Instruction Set Architecture

Pipelining, Hazard Resolution, Superscalar, Reordering, Branch Prediction, Speculation, VLIW, Vector, DSP, ...

Multiprocessing, Simultaneous CPU Multi-threading

Thread Level Parallelism (TLB)

Pipelining and Instruction Level Parallelism (ILP)

Addressing, Protection, Exception Handling
- Coherence, Bandwidth, Latency

RAID

Coherence, Bandwidth, Latency

Exception Handling
Computer Performance Evaluation: Cycles Per Instruction (CPI)

• Most computers run synchronously utilizing a CPU clock running at a constant clock rate:

  where:  \[ \text{Clock rate} = \frac{1}{\text{clock cycle}} \]

• A computer machine 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 and implementation.
  – 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 cycles to complete termed as the Cycles Per Instruction (CPI).

(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?

(From 550)
Comparing Computer Performance Using Execution Time

• To compare the performance of two machines (or CPUs) “A”, “B” running a given specific program:

\[
\begin{align*}
\text{Performance}_A &= \frac{1}{\text{Execution Time}_A} \\
\text{Performance}_B &= \frac{1}{\text{Execution Time}_B}
\end{align*}
\]

• Machine A is \( n \) times faster than machine B means:

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

• Example:

For a given program:

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

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

\[
\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A} = \frac{10}{1} = 10
\]

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.

(From 550)
CPU Execution Time: The CPU Equation

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

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

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

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

$$T = I \times \text{CPI} \times C$$

(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.

- What is the execution time for this program:

\[
\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \frac{1}{\text{Clock cycle}}
\]

\[
= 10,000,000 \times 2.5 \times \frac{1}{200 \times 10^6}
\]

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

\[
= .125 \text{ seconds}
\]

(From 550)
Aspects of CPU Execution Time

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

- Depends on:
  - Program Used
  - Compiler
  - ISA

- Depends on:
  - CPU Organization
  - Technology

Instruction Count \( I \)

CPI

Clock Cycle \( C \)
## Factors Affecting CPU Performance

The CPU time can be calculated as the product of the number of instructions, cycles, and seconds. The formula is:

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \text{Instructions} \times \frac{\text{Cycles}}{\text{Instruction}} \times \text{Seconds}$$

<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>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td><strong>Instruction Set Architecture (ISA)</strong></td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td><strong>Organization</strong></td>
<td>$\times$</td>
<td></td>
<td>$\times$</td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td></td>
<td></td>
<td>$\times$</td>
</tr>
</tbody>
</table>

(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?

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

or 32% faster after changes.

(From 550)
Instruction Types & CPI

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

\[
C_i = \text{Count of instructions of type}_i \\
CPI_i = \text{Cycles per instruction for type}_i
\]

Then:

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

Where:

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

\[
\text{Instruction Count} = \sum C_i
\]
Instruction Types And 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

\[\text{CPI for sequence 1} = \frac{\text{clock cycles}}{\text{instruction count}} = \frac{10}{5} = 2\]

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

\[\text{CPI for sequence 2} = \frac{9}{6} = 1.5\]

(From 550)
Instruction Frequency & CPI

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

\[
C_i = \text{Count of instructions of type } i \\
CPI_i = \text{Average cycles per instruction of type } i \\
F_i = \text{Frequency of instruction type } i \\
= \frac{C_i}{\text{total instruction count}}
\]

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

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq, $F_i$</th>
<th>CPI, $CPI_i$</th>
<th>$CPI_i \times F_i$</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23% = .5/2.2</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45% = 1/2.2</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14% = .3/2.2</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18% = .4/2.2</td>
</tr>
</tbody>
</table>

Typical Mix

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

$$\text{CPI} = .5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 = 2.2$$

(From 550)
Metrics of Computer Performance

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.

**Actual Target Workload**

**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.

**Full Application Benchmarks**

**Small “Kernel” Benchmarks**

**Microbenchmarks**
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
## 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>
</tbody>
</table>

### CINT2000 (Integer)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Language</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<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>187.facerec</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>

### CFP2000 (Floating Point)

### 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 III Xeon</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>750</td>
<td>PA-RISC 8700</td>
<td>581</td>
<td>526</td>
</tr>
<tr>
<td>750</td>
<td>PA-RISC 8700</td>
<td>581</td>
<td>526</td>
</tr>
<tr>
<td>1533</td>
<td>Athlon MP</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>500</td>
<td>PA-RISC 8600</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)

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

  \[
  \text{MIPS} = \frac{\text{Instruction count}}{\text{(Execution Time} \times 10^6)}
  \]

  \[
  = \frac{\text{Instruction count}}{\text{(CPU clocks} \times \text{Cycle time} \times 10^6)}
  \]

  \[
  = \frac{(\text{Instruction count} \times \text{Clock rate})}{(\text{Instruction count} \times \text{CPI} \times 10^6)}
  \]

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

- Problems with MIPS rating:
  - No account for the instruction set used.
  - Program-dependent: A single machine does not have a single MIPS rating since the MIPS rating may depend on the program used.
  - Easy to abuse: Program used to get the MIPS rating is often omitted.
  - Cannot be used to compare computers with different instruction sets.
  - A higher MIPS rating in many cases may not mean higher performance or better execution time. i.e. due to compiler design variations.
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>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)

MIPS = Clock rate / (CPI x 10^6) = 100 MHz / (CPI x 10^6)

CPI = CPU execution cycles / Instructions count

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

CPU time = Instruction count x CPI / Clock rate

• For compiler 1:
  – CPI\textsubscript{1} = (5 x 1 + 1 x 2 + 1 x 3) / (5 + 1 + 1) = 10 / 7 = 1.43
  – MIP\textsubscript{1} = 100 / (1.428 x 10^6) = 70.0
  – CPU time\textsubscript{1} = ((5 + 1 + 1) x 10^6 x 1.43) / (100 x 10^6) = 0.10 seconds

• For compiler 2:
  – CPI\textsubscript{2} = (10 x 1 + 1 x 2 + 1 x 3) / (10 + 1 + 1) = 15 / 12 = 1.25
  – MIP\textsubscript{2} = 100 / (1.25 x 10^6) = 80.0
  – CPU time\textsubscript{2} = ((10 + 1 + 1) x 10^6 x 1.25) / (100 x 10^6) = 0.15 seconds

(From 550)
Computer Performance Measures:

MFOLPS (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 double precision floating-point representation.
- MFLOPS, for a specific program running on a specific computer, is a measure of millions of floating point-operation (megaflops) per second:

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

- A better comparison measure between different machines than MIPS rating.
- Program-dependent: Different programs have different percentages of floating-point operations present. i.e compilers have no such operations and yield a MFLOPS rating of zero.
- Dependent on the type of floating-point operations present in the program.

(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:

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

  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:

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

  Hence speedup is given by:

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

(From 550)
Pictorial Depiction of Amdahl’s Law

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

Before:
Execution Time without enhancement E:

| Unaffected, fraction: (1- F) | Affected fraction: F |

Unaffected

After:
Execution Time with enhancement E:

Unaffected, fraction: (1- F)  F/S

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

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

(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 = \( .5 \times 1 + .2 \times 2 + .1 \times 3 + .2 \times 2 = 1.6 \)

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.

(From 550)
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?

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

\[
\text{Execution time with enhancement} = 25 \text{ seconds}
\]

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

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

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

\[
n = \frac{80}{5} = 16
\]

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

(From 550)
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?

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

\[
\text{Execution time with enhancement} = 20 \text{ seconds}
\]

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

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

\[
0 = \frac{80 \text{ seconds}}{n}
\]

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 refer to original execution time.
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\(_1 = F_2 = 15\%\)
  - Speedup\(_3 = S_3 = 30\) Percentage\(_1 = F_3 = 10\%\)

• 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?

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

• 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

<table>
<thead>
<tr>
<th>Unaffected, fraction:</th>
<th>.55</th>
<th>F₁ = .2</th>
<th>F₂ = .15</th>
<th>F₃ = .1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>/ 10</td>
<td>/ 15</td>
<td>/ 30</td>
</tr>
</tbody>
</table>

Unaffected, fraction: .55

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

Speedup = 1 / .5833 = 1.71

Note: All fractions 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.
Operand Locations in Four ISA Classes

(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store

Processor

TCS

ALU

Memory

EECC551 - Shaaban
# Code Sequence  $C = A + B$

for Four Instruction Sets

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1, A</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>
General-Purpose Register (GPR) Machines

• Every machine designed after 1980 uses a load-store GPR architecture.

• 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.
## General-Purpose Register Machines

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register</td>
<td>Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute (see Ch 3).</td>
<td>Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.</td>
</tr>
<tr>
<td>(0,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-memory</td>
<td>Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.</td>
<td>Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.</td>
</tr>
<tr>
<td>(1,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Most compact. Doesn’t waste registers for temporaries.</td>
<td>Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.</td>
</tr>
<tr>
<td>(3,3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Advantages and disadvantages of the three most common types of general-purpose register machines.
# 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-11</td>
<td>8</td>
<td>register-memory</td>
<td>1970</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>Motorola 68000</td>
<td>16</td>
<td>register-memory</td>
<td>1980</td>
</tr>
<tr>
<td>MIPS</td>
<td>32</td>
<td>load-store</td>
<td>1985</td>
</tr>
<tr>
<td>SPARC</td>
<td>32</td>
<td>load-store</td>
<td>1987</td>
</tr>
</tbody>
</table>
## Examples of GPR Machines

<table>
<thead>
<tr>
<th>Number of memory addresses</th>
<th>Maximum number of operands allowed</th>
<th>SPARK, MIPS</th>
<th>PowerPC, ALPHA</th>
<th>Intel 80x86, Motorola 68000</th>
<th>VAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></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[Reg[R3]]]</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1, (R2) +</td>
<td>Regs[R1] ← Regs[R1] + Mem[Reg[R2]]</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1, - (R2)</td>
<td>Regs[R2] ← Regs[R2] - d</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>
Addressing Modes Usage Example

For 3 programs running on VAX ignoring direct register mode:

Displacement: 42% avg, 32% to 55%
Immediate: 33% avg, 17% to 43%
Register deferred (indirect): 13% avg, 3% to 24%
Scaled: 7% avg, 0% to 16%
Memory indirect: 3% avg, 1% to 6%
Misc: 2% avg, 0% to 3%

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

Observation: In addition Register direct, Displacement, Immediate, Register Indirect addressing modes are important.
Utilization of Memory Addressing Modes

Summary of use of memory addressing modes (including immediates).
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
## 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>Graphics</td>
<td>Pixel operations, compression/ decompression operations</td>
</tr>
</tbody>
</table>
### Instruction Usage Example: Top 10 Intel X86 Instructions

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

**Total**

|                     |                 | **96%**            |

**Observation:** Simple instructions dominate instruction usage frequency.
Breakdown of control flow instructions into three classes: calls or returns, jumps and conditional branches for SPEC CPU2000 programs.
Type and Size of Operands

• Common operand types include (assuming a 64 bit CPU):
  
  Character (1 byte)
  Half word (16 bits)
  Word (32 bits)
  Double word (64 bits)

• IEEE standard 754: single-precision floating point (1 word), double-precision floating point (2 words).

• For business applications, some architectures support a decimal format (packed decimal, or binary coded decimal, BCD).
Type and Size of Operands

Distribution of data accesses by size for SPEC CPU2000 benchmark programs

- Double word (64 bits): 70% floating-point average, 59% integer average
- Word (32 bits): 29% floating-point average, 26% integer average
- Half word (16 bits): 0% floating-point average, 5% integer average
- Byte (8 bits): 1% floating-point average, 10% integer average
Instruction Set Encoding

Considerations affecting instruction set encoding:

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

- The Impact of 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.
### 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: DLX, MIPS, PowerPC, SPARC

<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>

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

- Emphasizes doing more with each instruction
- 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.

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 X86, 386/486/Pentium

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.

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.
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.
- Designed with pipelining in mind.
- Fixed-length instruction encoding.
- Only load and store instructions access memory.
- 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
Example RISC ISA:

PowerPC

8 addressing modes:

- Register direct.
- Immediate.
- Register indirect.
- Register indirect with immediate index (loads and stores).
- Register indirect with register index (loads and stores).
- Absolute (jumps).
- Link register indirect (calls).
- Count register indirect (branches).

Operand sizes:

- Four operand sizes: 1, 2, 4 or 8 bytes.

Instruction Encoding:

- Instruction set has 15 different formats with many minor variations.
- All are 32 bits in length.
Example RISC ISA:

HP Precision Architecture, HP-PA

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.
Example RISC ISA:
Compaq Alpha AXP

4 addressing modes:
- Register direct.
- Immediate.
- Register indirect with displacement.
- PC-relative.

Operand sizes:
- Four operand sizes: 1, 2, 4 or 8 bytes.

Instruction Encoding:
- Instruction set has 7 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.

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

**Operand Sizes:**
- Memory accesses in any multiple between 1 and 8 bytes.

**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>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td>jump target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Registers**
- R0 - R31
- PC
- HI
- LO
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>sa</td>
<td>funct</td>
<td></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>immediate</td>
<td></td>
<td></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>displacement</td>
<td></td>
<td></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.
# MIPS64 Instruction Format

**I - type instruction**

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

Encode: Loads and stores of bytes, words, half words. All immediates (rt ← rs op immediate)
Conditional branch instructions (rs1 is register, rd unused)
Jump register, jump and link register (rd = 0, rs = destination, immediate = 0)

**R - type instruction**

| Opcode | rs | rt | rd | shamt | func |

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**

| Opcode | Offset added to PC |

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

- All instructions 32 bits wide

**Register (direct)**
- Format: `op rs rt rd`
- First Operand
- Second Operand
- Destination

**Immediate**
- Format: `op rs rt immed`

**Displacement: Base+index**
- Format: `op rs rt immed`
- Memory Access

**PC-relative**
- Format: `op rs rt immed`
- Memory Access

**Branches**
- Format: `op rs rt immed`
- Memory Access
## MIPS64 Instructions: Load and Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1,30(R2)</td>
<td>Load double word</td>
<td>Regs[R1] ←&lt;sub&gt;64&lt;/sub&gt; Mem[30+Regs[R2]]</td>
</tr>
<tr>
<td>LW R1, 60(R2)</td>
<td>Load word</td>
<td>Regs[R1] ←&lt;sub&gt;64&lt;/sub&gt; (Mem[60+Regs[R2]])&lt;sup&gt;32&lt;/sup&gt; #&lt;br&gt;Mem[60+Regs[R2]]</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
<td>Regs[R1] ←&lt;sub&gt;64&lt;/sub&gt; (Mem[40+Regs[R3]])&lt;sup&gt;56&lt;/sup&gt; #&lt;br&gt;Mem[40+Regs[R3]]</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
<td>Regs[R1] ←&lt;sub&gt;64&lt;/sub&gt; 0&lt;sup&gt;56&lt;/sup&gt; #&lt;br&gt;Mem[40+Regs[R3]]</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load half word</td>
<td>Regs[R1] ←&lt;sub&gt;64&lt;/sub&gt; (Mem[40+Regs[R3]])&lt;sup&gt;48&lt;/sup&gt; #&lt;br&gt;Mem[40 + Regs[R3]] # # Mem [41+Regs[R3]]</td>
</tr>
<tr>
<td>L.S F0, 50(R3)</td>
<td>Load FP single</td>
<td>Regs[F0] ←&lt;sub&gt;64&lt;/sub&gt; Mem[50+Regs[R3]] #&lt;br&gt;0&lt;sup&gt;32&lt;/sup&gt;</td>
</tr>
<tr>
<td>L.D F0, 50(R2)</td>
<td>Load FP double</td>
<td>Regs[F0] ←&lt;sub&gt;64&lt;/sub&gt; Mem[50+Regs[R2]]</td>
</tr>
<tr>
<td>SD R3,500(R4)</td>
<td>Store double word</td>
<td>Mem [500+Regs[R4]] ←&lt;sub&gt;64&lt;/sub&gt; Reg[R3]</td>
</tr>
<tr>
<td>SW R3,500(R4)</td>
<td>Store word</td>
<td>Mem [500+Regs[R4]] ←&lt;sub&gt;32&lt;/sub&gt; Reg[R3]</td>
</tr>
<tr>
<td>S.S F0, 40(R3)</td>
<td>Store FP single</td>
<td>Mem [40, Regs[R3]] ←&lt;sub&gt;32&lt;/sub&gt; Reg[F0] 0...31</td>
</tr>
<tr>
<td>S.D F0,40(R3)</td>
<td>Store FP double</td>
<td>Mem[40+Regs[R3]] ←&lt;sub&gt;64&lt;/sub&gt; Reg[F0]</td>
</tr>
<tr>
<td>SH R3, 502(R2)</td>
<td>Store half</td>
<td>Mem[502+Regs[R2]] ←&lt;sub&gt;16&lt;/sub&gt; Regs[R3]&lt;sup&gt;48...63&lt;/sup&gt;</td>
</tr>
<tr>
<td>SB R2, 41(R3)</td>
<td>Store byte</td>
<td>Mem[41 + Regs[R3]] ←&lt;sub&gt;8&lt;/sub&gt; 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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</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>$Reg_{31} \leftarrow PC+4$; $PC_{36..63} \leftarrow name$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$((PC+4)-2^{27}) \leq name &lt; ((PC+4)+2^{27})$</td>
</tr>
<tr>
<td>JALR R2</td>
<td>Jump and link register</td>
<td>$Reg_{31} \leftarrow PC+4$; $PC \leftarrow Reg_{R2}$</td>
</tr>
<tr>
<td>JR R3</td>
<td>Jump register</td>
<td>$PC \leftarrow Reg_{R3}$</td>
</tr>
<tr>
<td>BEQZ R4, name</td>
<td>Branch equal zero</td>
<td>if $(Reg_{R4} == 0)$ $PC \leftarrow name$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$((PC+4)-2^{17}) \leq name &lt; ((PC+4)+2^{17}$</td>
</tr>
<tr>
<td>BNEZ R4, Name</td>
<td>Branch not equal zero</td>
<td>if $(Reg_{R4} != 0)$ $PC \leftarrow name$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$((PC+4)-2^{17}) \leq name &lt; ((PC+4)+2^{17}$</td>
</tr>
<tr>
<td>MOVZ R1,R2,R3</td>
<td>Conditional move if zero</td>
<td>if $(Reg_{R3} == 0)$ $Reg_{R1} \leftarrow Reg_{R2}$</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:**
- 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
## Major Types of Compiler Optimization

<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>High-level</td>
<td>At or near the source level; machine-independent</td>
<td>N.M.</td>
</tr>
<tr>
<td>Procedure integration</td>
<td>Replace procedure call by procedure body</td>
<td>N.M.</td>
</tr>
<tr>
<td>Local</td>
<td>Within straight-line code</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>Global</td>
<td>Across a branch</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>Machine-dependent</td>
<td>Depends on machine knowledge</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>
Change in instruction count for the programs lucas and mcf from SPEC2000 as compiler optimizations vary.