ELE 594 Modern Processor Design
Spring 2005

• Today:
  – Previous lecture’s review
  – First part
    • Overview, Advanced Topics and Project Info.
  – Second Part
    • Performance, Instruction Set Architecture

ELE 594 Modern Processor Design
Spring 2005

• Instructor: Resit Sendag
• Office: Kelley 211
• Office Hours: 2:00-3:00pm Tuesdays and Wednesdays
• Email: sendag@ele.uri.edu
• Phone: 401-874-9423

Modern Processor Design

• Time
  4:00pm – 6:30pm on Tuesdays
• Location
  Kelley 203
• Course Webpage
  http://www.ele.uri.edu/faculty/sendag/ele594
• Credits: 3
• Prerequisite
  ELE 305 or ELE 405 or equivalent or permission of instructor

Textbooks


  The required text is available at the bookstore in Memorial Union, and both books are available through amazon.com.
Grading

Homework: 20% (2 homework assignments)
Midterm Exam (?): 30% (in class? or take-home)
Term Project: 40% (surveys or research ideas)
Participation, Presentation: 10%

Important Dates

Term Project proposal due: February 15, 2005
Term Project progress report due: March 22, 2005
Midterm exam: March 29, 2005
Term Project preliminary report due: April 19, 2005
Term Project final report due: May 10, 2005

Class email list

- Add your name to the class e-mail list by sending a message to "majordomo@ele.uri.edu".
- In the body of the message say "subscribe ele594" (without the quotes).
- Then respond to the email that it sends you for confirmation.
- It is very important that you add yourself to this list since some of the information for this class will be distributed via e-mail.

Computer Accounts

- If you don’t already have one, you must get an ELE account to use the machines in Graduate Student Lab.
- You must contact the ELE System Manager for the account in Kelley 117.
- A storage area is granted for each ELE account designated as the folder /grads/account. So if the login being used is yourname, the storage location for this user is /grads/yourname. You will automatically get an email account: yourname@ele.uri.edu.
High performance Computing Seminar

Bi-weekly High Performance Computing Seminar (HPCS):
- You are encouraged to attend the HPC seminars of the ECE department.
- The seminar covers the recent work of the computer engineering faculty in the areas of computer architecture, I/O and storage systems, and computer networks.
- Add your name to the HPCS e-mail list by sending a message to "majordomo@ele.uri.edu". In the body of the message say "subscribe hpcs" (without the quotes). Then respond to the email that it sends you for confirmation.
- The information for HPC seminar will be distributed via e-mail.

Term Project

- Survey Projects
  - List of topics will be provided
  - Read at least 10 research papers

- Research Projects
  - one of the areas discussed during class; however, you are free to propose any topic in microprocessor architecture.
  - Propose some new architectural idea and quantitatively evaluate it using a simulation study.
  - Verify the results of a previously published research paper.

What is this course about?

- Computers are designed at three levels.
  - The lowest level involves intricate knowledge of circuits.
  - The middle level involves the logical specification of a hardware system using a hardware definition language like Verilog or VHDL.
  - The highest, most abstract, level involves simulation of the computer, but not at the level of individual gates.
- In this class, we investigate computer architecture with a particular focus on microprocessor design.
- We will concentrate at the highest level of abstraction.
- The class will explore the current trends and future directions of processor microarchitecture.

Topics to be covered

- Processor Design
- Pipelined Processors
- Memory and I/O Systems
- Superscalar Processors
- Superscalar Techniques
- IBM PowerPC 620 Architecture
- Intel's P6 Architecture
- Survey of Superscalar Processors
  - Compaq/DEC Alpha, HP PA-PISC, IBM Power, Intel IA-32, x86-64, MIPS, Motorola, Sun SPARC v8.9
- Advanced Instruction Flow Techniques
- Advanced Register Data Flow Techniques
- Executing Multiple Threads
Modern Processor Design
Spring 2005

Lecture 1: Outlines

• First part:
  – A glance at Advanced Topics and Project Information
• 10 minutes break
• Second part:
  – Review of Performance (Ch1), Instruction Set Architecture (Ch2)

Processor Performance

• FAST…FASTER…NOT ENOUGH?
• Processor Performance Equation

\[
\frac{1}{Performance} = \frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instructions} \times \frac{Time}{Cycle}
\]

• Compiler – eliminate redundant and useless inst. In the object code (reduces instruction count)
• Faster Circuit technology – faster propagation (reduces cycle time)
• The above two always desirable (do not affect the other two terms in the equation.)

Improving the Performance

• Reduce instruction count
  – Include more complex instructions ?
    • Can decrease the # of inst. Executed significantly
    • CISC ISA vs. RISC ISA (twice as many inst. in RISC)
    • The complexity of execution unit increases
      – May increase the cycle time !
  – Unrolling (unrolling loops may result in fewer # of inst.)
    • => static code size ↑ =>ICache hit rate ↑ => CPI ↑

Improving the Performance

• Reduce instruction count
  – Dynamically Eliminating Redundant Computations
    • A microarchitectural technique
    • Frequently executed instructions with the same data set (store the results and reuse)
      • => Complexity of HW ↑ => cycle time ↑
  – Can potentially lead to increasing CPI and/or cycle time.
Improving the Performance

• Reduce CPI
  – RISC
    • =>inst. Count ↑
  – Pipelining
    • Overlap the processing of multiple instructions

\[
\text{CPI}_{\text{pipeline}} = \frac{\text{CPI}_{\text{nonpipelined}}}{\text{Number of pipeline stages}}
\]

• Control and data hazards limits the overlap
  – Branch Prediction
    • Reduces the stall cycles if correct
    • Increases the hardware complexity => may increase the cycle time
    • Can potentially increase the CPI in deeper pipelines (due to large misprediction penalty)
  – Cache memory – reduce average memory latency
  – Superscalar
    • Multiple simultaneous pipelines
    • Process multiple instructions in each pipe stage
    • Can significantly reduce CPI
    • The complexity of each pipe stage can increase => cycle time & pipeline depth may increase, which can in turn increase CPI

Improving the Performance

• Reduce cycle time
  – Key technique: Pipelining
    • The latency of each pipe stage determines machine cycle time.
    • Deeper pipeline => lower cycle times
  – Major technique to increase the performance in the recent decade.
  – Increasing the clock freq. through deeper pipelining may increase the CPI
    • Increases the ranch misprediction penalty
    • ALU op. requires multiple cycles, the latency between two dependent inst. Increases
    • The average latency of mem. Ops. Increase
    • H/W and latency overhead by pipelining

Improving the Performance

• Conclusion:
  – Improving Processor performance is NOT straightforward task.
    • Many tradeoffs…
**Instruction-level Parallelism (ILP)**

- ILP = concurrent processing of multiple instructions
- Machines for ILP
  - Superpipelined machines
    - Baseline cycle is divided into \( m \) shorter minor cycles (degree \( m \))
    - Each instruction still requires one baseline cycle, but a new instruction can issue in every minor cycle.
  - From Scalar to Superscalar machines
    - Scalar = Pipelined processors that are designed to fetch and issue at most one instruction in every machine cycle.
    - Superscalar can fetch and issue multiple instructions every machine cycle.
    - Superscalar-Superpipelined \( m \times k \) stages with superscalar degree of \( n \).
  - Very-Long-Instruction-Word machines
    - Exploit ILP in compiler

**Limits of ILP**

- Deeper pipelines = more ILP?
  - Needed to achieve very high clk frequencies, has physical limitations
- wider issue = more ILP?
  - Large instruction window
  - increase HW complexity
- Pipeline stalls
  - Control hazards -- Branch
  - Data hazards – RAW, WAR, WAW
- Processor-Memory gap is increasing
  - In 2010s, 1000s of cycles to retrieve data from memory

**Increasing the ILP—Superscalar Techniques**

- Instruction Flow techniques – Branch processing
  - Branch Prediction
    - Branch target speculation => BTB
    - Branch condition speculation => branch prediction
      - Taken, not-taken, history-based (2-bit, two-level adaptive, correlated)
    - When hard to predict, Predication
- Register data Flow Techniques -- ALU inst. processing
  - Register renaming at dispatch (Eliminates false dependences WAR, WAW)
  - Reorder buffer and reservation stations
  - Out-of-order execution core between an in-order front-end.

**Increasing the ILP—Superscalar Techniques**

- Other Register Data Flow Techniques
  - Value Locality
  - Value Prediction
  - Value Reuse
- Memory data Flow Techniques – Load/store inst. Processing
  - Memory address generation, address translation, data memory accessing
  - Load Store Queue and Memory Hierarchy
  - Load Bypassing and Load Forwarding
Increasing the ILP—Superscalar Techniques

- Other Memory data Flow Techniques
  - Wider issue => Need more bandwidth!
    - Multi-ported data cache
  - Higher clk frequency => Need methods to hide mem. Latency
    - Multiple level of caches
      - Currently at least two-levels
    - Non-blocking cache
    - Victim cache
    - Cache size, associativity, block size, etc.

Memory Hierarchy

- Processor-DRAM Performance gap
- 1/3 to 2/3 die area for caches, TLB
- Alpha 21264: 108 clock to memory \( \Rightarrow \) 648 instruction issues during miss
- 3 Cs: Compulsory, Capacity, Conflict
- 4 Questions: where, how, which, write
- Applied recursively to create multilevel caches
- Performance = \( f(\text{hit time}, \text{miss rate}, \text{miss penalty}) \)
  - danger of concentrating on just one when evaluating performance

Increasing the ILP—Superscalar Techniques

- Increasing MEMORY Performance --continued
  - Prefetching
    - Software/Hardware initiated,
    - Sequential prefetching: Prefetch on miss, Tagged prefetching
    - Prefetching with arbitrary strides
    - Stream buffers
    - Wrong-path prefetching
    - Timing and accuracy is important: prefetch scheduling and prefetch distance

Cache Optimization Summary

\[
\text{CPUtime} = IC \times \left( \text{CPI}_{\text{Inst.}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \right) \times \text{Clock cycle time}
\]

<table>
<thead>
<tr>
<th>Technique</th>
<th>MR</th>
<th>MP</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td>–</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>–</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Subblock Placement</td>
<td>+</td>
<td>+</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td>+</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td>+</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Small &amp; Simple Caches</td>
<td>–</td>
<td>+</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Avoiding Address Translation</td>
<td>+</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Pipelining Writes</td>
<td>+</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

memory hierarchy art: taste in selecting between alternatives to find combination that fits well together
Increasing the ILP—Superscalar Techniques

- Increasing MEMORY Performance --continued
  - Breaking the Limits: When everything else fails – Speculate!
    - Load address prediction
    - Load value prediction
    - Memory dependence prediction

Advanced register data flow techniques

- Value Locality and Redundant execution
  - Likelihood of a computed result to recur later
- Exploiting value locality
  - Without speculation
  - With Speculation
- Frequent Value Locality
- Address Correlation

Executing Multiple Threads

- Discussion up to this point was processing single-thread of execution (serial program)
- ILP reaches to a limit
- A new type of parallelism: TLP
- OS developers invented the concept of time-sharing
  - Slow I/O
- Multiprocessors and cache coherence

Executing Multiple Threads

- Explicit Multithreaded Architectures
  - Fine-grained multithreading: switching between multiple thread contexts every cycle
  - Coarse-grained: switches contexts only on long-latency events
  - Simultaneous Multithreading (SMT): no context switching, allow instrs from multiple threads to be processed simultaneously (out-of-order)
- Implicit Multithreading
  - Thread-level speculation
Research Paper Reading

- As graduate students, you are now researchers.
- Most information of importance to you will be in research papers.
- Ability to rapidly scan and understand research papers is key to your success.
- So: 1-2 paper / week in this course
- Papers “Readings in Computer Architecture” or online
- Think about methodology and approach

Review of Fundamental Concepts

- Instruction Set Architecture
- Machine Organization
- Instruction Execution Cycle
- Pipelining
- Memory
- Bus (Peripheral Hierarchy)
- Performance Iron Triangle

Review of Performance

- ACM Digital Library
  - International Symposium on Computer Architecture (ISCA)
  - International Symposium on Microarchitecture (MICRO)
  - Architectural Support for Programming Languages and Operating Systems (ASPLOS)
  - International Conference on Supercomputing (ICS)
  - Symposium on Parallel Algorithms and Architectures (SPAA)
- IEEE Computer Society Digital Library
  - High Performance Computer Architecture (HPCA)
  - IEEE Computer
  - IEEE Micro
  - IEEE Transactions on Computers
  - IEEE Transactions on Parallel and Distributed Systems
- On-line Journals
  - Computer Architecture Letters
  - The Journal of Instruction-Level Parallelism
  - ACM TACO - ACM Transactions on Architecture and Code Optimization
Performance

- Measure, Report, and Summarize
- Make intelligent choices
- See through the marketing hype
- Key to understanding underlying organizational motivation

Why is some hardware better than others for different programs?

What factors of system performance are hardware related? (e.g., Do we need a new machine, or a new operating system?)

How does the machine's instruction set affect performance?

Performance

- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)
    - a floating point intensive application might have a higher CPI
  - MIPS (millions of instructions per second)
    - this would be higher for a program using simple instructions

Performance

- Performance is determined by execution time
- Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction?
  - average # of instructions per second?
- Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

Execution Time

- Elapsed Time
  - counts everything (disk and memory accesses, I/O, etc.)
  - a useful number, but often not good for comparison purposes
- CPU time
  - doesn't count I/O or time spent running other programs
  - can be broken up into system time, and user time
- Our focus: user CPU time
  - time spent executing the lines of code that are "in" our program
Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
    e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - Nice for architects and designers
  - Easy to standardize
  - Can be abused
- SPEC (System Performance Evaluation Cooperative)
  - Companies have agreed on a set of real program and inputs
  - Valuable indicator of performance (and compiler technology)

SPEC 2000 Benchmarks

- Integer
  - Vpr (FPGA Circuit placement and routing)
  - Gcc (C programming language compiler)
  - Crafty (Game playing: chess)
  - Parser (Word processing)
  - Eon (Computer visualization)
  - Perlbmk (PERL programming language)
- Floating-Point
  - Equake (Seismic Wave Propagation Simulation)
  - Facerec (Image Processing - face recognition)
  - Fima3d (Finite-element crash simulation)
  - Sixtrack (High Energy Nuclear Physics Accelerator Design)
  - Apsi (Meteorology: Pollutant distribution)

Measuring performance

- Kernel
  - Representative program fragments
  - Focusing on individual feature not the overall picture
- Mixes
  - A program that has a certain number of instruction occurrence
  - Calculation
- Toy Benchmarks
  - Toy programs, such as fibonacci, prime number...
- Synthetic benchmarks
  - Programs intend to give specific mix

Which is faster?

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

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

"X is n times faster than Y" means

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

- Speed of Concorde vs. Boeing 747
- Throughput of Boeing 747 vs. Concorde

Definitions

- Performance is in units of things per sec
  - bigger is better
- If we are primarily concerned with response time
  \[
  \text{performance}(x) = \frac{1}{\text{execution_time}(x)}
  \]

"X is n times faster than Y" means

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

Computer Performance

<table>
<thead>
<tr>
<th>Inst Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>Inst. Set.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

CPU time = \(\text{Seconds} = \text{Instructions} \times \text{Cycles} \times \text{Seconds}\)

Cycles Per Instruction (Throughput)

"Average Cycles per Instruction"

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

\[
\text{CPU time} = \text{Cycle Time} \times \sum_{j=1}^{n} \text{CPI}_j \times I_j
\]

\[
\text{CPI} = \sum_{j=1}^{n} \text{CPI}_j \times F_j \quad \text{where} \quad F_j = \frac{I_j}{\text{Instruction Count}}
\]

"Instruction Frequency"
Example: Calculating CPI bottom up

<table>
<thead>
<tr>
<th>Base Machine (Reg / Reg)</th>
<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>(33%)</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>.2</td>
<td>(13%)</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
<td></td>
</tr>
</tbody>
</table>

Typical Mix of instruction types in program

Example: Branch Stall Impact

- Assume CPI = 1.0 ignoring branches (ideal)
- Assume solution was stalling for 3 cycles
- If 30% branch, Stall 3 cycles on 30%

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>70%</td>
<td>1</td>
<td>.7</td>
<td>(37%)</td>
</tr>
<tr>
<td>Branch</td>
<td>30%</td>
<td>4</td>
<td>1.2</td>
<td>(63%)</td>
</tr>
</tbody>
</table>

=> new CPI = 1.9
New machine is 1/1.9 = 0.52 times faster (i.e. slow!)

Speed Up Equation for Pipelining

For simple RISC pipeline, CPI = 1:

\[
\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

Amdahl's Law

Execution Time After Improvement =

\[
\frac{\text{Execution Time Unaffected} + \left( \text{Execution Time Affected} / \text{Amount of Improvement} \right)}{\text{Execution Time Affected}}
\]

Speedup due to enhancement E:

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

Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected.
Amdahl’s Law

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

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

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

\[ \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + \frac{1}{2}) = 0.95 \times \text{ExTime}_{\text{old}} \]

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

How to Summarize Performance

• Arithmetic mean

• Arithmetic mean of times (for N programs)

\[ \text{Meaningful only when all N programs run equally often, otherwise Weighted arithmetic mean is better} \]

Weighted Arithmetic mean:

\[ \text{Summation of (Weight x Time of each program)} / N \]
How to Summarize Performance

- **MIPS (Million Instructions Per Second)**
  
  \[
  \text{MIPS} = \frac{\text{Instruction count}}{(\text{Exec. Time} \times 10^6)} = \frac{\text{Clock Rate}}{(\text{CPI} \times 10^6)}
  \]

  Should be used only when comparing different architectures with the same instruction set, compiler, OS, clock speed.

- **MFLOPS (Million Floating-Point Operations per Second)**
  
  \[
  \text{MFLOPS} = \frac{\text{FP operations}}{(\text{Exec. Time} \times 10^6)}
  \]

  Not all the architectures implement the same floating point operations.

  Not all FP ops do equal amount of work.

Remember

- **Performance is specific to a particular program/s**
  - Total execution time is a consistent summary of performance

- **For a given architecture performance increases come from:**
  - Increases in clock rate (without adverse CPI affects)
  - Improvements in processor organization that lower CPI
  - Compiler enhancements that lower CPI and/or instruction count

- **Pitfall:** Expecting improvement in one aspect of a machine’s performance to affect the total performance

Summary, #1

- **Time to run the task**
  - Execution time, response time, latency

- **Tasks per day, hour, week, sec, ns, …**
  - Throughput, bandwidth

- “X is n times faster than Y” means
  
  \[
  \frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}
  \]

Summary, #2

- **Amdahl’s Law:**
  
  \[
  \text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}}} \times \text{Speedup}_{\text{enhanced}}
  \]

- **CPI Law:**

  \[
  \begin{align*}
  \text{CPU time} & = \text{Seconds} = \frac{\text{Instructions} \times \text{Cycles} \times \text{Seconds}}{	ext{Program} \times \text{Instruction} \times \text{Cycle}} \\
  \end{align*}
  \]

  - Execution time is the REAL measure of computer performance!

  - Good products created when have:
    - Good benchmarks, good ways to summarize performance
A good interface:

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

What is an ISA?

- The parts of a processor's design that needs to be understood in order to write assembly language, such as the machine language instructions and registers.
- Parts of the architecture that are left to the implementation, such as number of superscalar functional units, cache size and cycle speed, are not part of the ISA.

(Taken from http://dictionary.reference.com)
What are the components of an ISA?

- Sometimes known as The Programmers Model of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The Machine Instruction Set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i.e. Register Transfers
- The Instruction Format
  - Size and meaning of fields within the instruction
- The nature of the Fetch/Execute cycle
  - Things that are done before the operation code is known

What Must an Instruction Specify?

Data Flow

- Which operation to perform: \( \text{add r}0, r1, r3 \)
  - Ans: Op code: add, load, branch, etc.
- Where to find the operand or operands \( \text{add r}0, r1, r3 \)
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result \( \text{add r}0, r1, r3 \)
  - Again CPU register or memory cell
- Location of next instruction \( \text{add r}0, r1, r3 \)
  - Almost always memory cell pointed to by program counter—PC
- Sometimes there is no operand, or no result, or no next instruction. Can you think of examples?

Instructions Can Be Divided into 3 Classes

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - Load—source is memory and destination is register
  - Store—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Changes the form of one or more operands to produce a result stored in another location
  - Add, Sub, Shift, etc.
- Branch instructions (control flow instructions)
  - Any instruction that alters the normal flow of control from executing the next instruction in sequence
  - Br Loc, Brz Loc2, — unconditional or conditional branches
### Examples of Data Movement Instructions

<table>
<thead>
<tr>
<th>Instruct.</th>
<th>Meaning</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A, B</td>
<td>Move 16 bits from mem. loc. A to loc. B</td>
<td>VAX11</td>
</tr>
<tr>
<td>lwz R3, A</td>
<td>Move 32 bits from mem. loc. A to reg. R3</td>
<td>PPC601</td>
</tr>
<tr>
<td>li $3, 455</td>
<td>Load the 32 bit integer 455 into reg. 3</td>
<td>MIPS R3000</td>
</tr>
<tr>
<td>mov R4, dout</td>
<td>Move 16 bits from R4 to out port dout</td>
<td>DEC PDP11</td>
</tr>
<tr>
<td>IN, AL, KBD</td>
<td>Load a byte from in port KBD to accum.</td>
<td>Intel Pentium</td>
</tr>
<tr>
<td>LEA.L (A0), A2</td>
<td>Load address pointed to by A0 into A2</td>
<td>M68000</td>
</tr>
</tbody>
</table>

- Lots of variation, even with one instruction type

### Examples of ALU (Arithmetic and Logic Unit) Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULF A, B, C</td>
<td>multiply the 32-bit floating point values at mem loc’ns. A and B, store at C</td>
<td>VAX11</td>
</tr>
<tr>
<td>abs r3, r1</td>
<td>Store abs value of r1 in r3</td>
<td>PPC601</td>
</tr>
<tr>
<td>ori $2, $1, 255</td>
<td>Store logical OR of reg $1 with 255 into reg $2</td>
<td>MIPS R3000</td>
</tr>
<tr>
<td>DEC R2</td>
<td>Decrement the 16-bit value stored in reg R2</td>
<td>DEC PDP11</td>
</tr>
<tr>
<td>SHL AX, 4</td>
<td>Shift the 16-bit value in reg AX left by 4 bit pos’ns.</td>
<td>Intel 8086</td>
</tr>
</tbody>
</table>

- Notice again the complete dissimilarity of both syntax and semantics.

### CPU Registers Associated with Flow of Control—Branch Insts.

- Program counter usually locates next inst.
- Condition codes may control branch
- Branch targets may be separate registers

Processor State

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C N V Z</td>
</tr>
</tbody>
</table>

Branch Targets
HLL Conditionals Implemented by Control Flow Change

- Conditions are computed by arithmetic instructions
- Program counter is changed to execute only instructions associated with true conditions

C language

```c
if NUM==5 then SET=7
```

Assembly language

```assembly
CMP.W  #5, NUM ;the comparison
BNE    L1 ;conditional branch
MOV.W  #7, SET ;action if true
L1 ... ;action if false
```

CPU Registers may have a “personality”

- Architecture classes are often based on how where the operands and result are located and how they are specified by the instruction.
- They can be in CPU registers or main memory

<table>
<thead>
<tr>
<th>Stack</th>
<th>Arithmetic Registers</th>
<th>Address Registers</th>
<th>General Purpose Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack Machine</td>
<td>Arithmetic Machine</td>
<td>Address Machine</td>
<td>General Register Machine</td>
</tr>
</tbody>
</table>

3, 2, 1, & 0 Address ISAs

- The classification is based on arithmetic instructions that have two operands and one result
- The key issue is “how many of these are specified by memory addresses, as opposed to being specified implicitly”
- A 3 address instruction specifies memory addresses for both operands and the result \(R \leftarrow Op1 \text{ op } Op2\)
- A 2 address instruction overwrites one operand in memory with the result \(Op2 \leftarrow Op1 \text{ op } Op2\)
- A 1 address instruction has a processor, called the accumulator register to hold one operand & the result (no addr. needed) \(Acc \leftarrow Acc \text{ op } Op1\)
- A 0 address + uses a CPU register stack to hold both operands and the result \(TOS \leftarrow TOS \text{ op } SOS\) where TOS is Top Of Stack, SOS is Second On Stack)
- The 4-address instruction, hardly ever seen, also allows the address of the next instruction to specified explicitly.

The 4 Address Instruction

- Explicit addresses for operands, result & next instruction
- Example assumes 24-bit addresses
  - Discuss: size of instruction in bytes
The 3 Address Instruction

- Address of next instruction kept in processor state register—the PC (Except for explicit Branches/Jumps)
- Rest of addresses in instruction
  - Discuss: savings in instruction word size

Memory

CPU

Op1Addr: Op1
Op2Addr: Op2
ResAddr: Res

NextiAddr: Nexti

Program counter

Where to find next instruction

<table>
<thead>
<tr>
<th>Bits:</th>
<th>8</th>
<th>24</th>
<th>24</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>ResAddr</td>
<td>Op1Addr</td>
<td>Op2Addr</td>
<td></td>
</tr>
</tbody>
</table>

Which operation

Where to put result

Where to find operands

The 2 Address Instruction

- Result overwrites Operand 2
- Needs only 2 addresses in instruction but less choice in placing data

Memory

CPU

Op1Addr: Op1
Op2Addr: Op2, Res

NextiAddr: Nexti

Program counter

Where to find next instruction

<table>
<thead>
<tr>
<th>Bits:</th>
<th>8</th>
<th>24</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Op2Addr</td>
<td>Op1Addr</td>
<td></td>
</tr>
</tbody>
</table>

Which operation

Where to find operands

Instruction formats

The 1 Address Instruction

- Special CPU register, the accumulator, supplies 1 operand and stores result
- One memory address used for other operand

Memory

CPU

Op1Addr: Op1

Accumulator

Where to find operand2, and where to put result

Instruction format

<table>
<thead>
<tr>
<th>Bits:</th>
<th>8</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Op1Addr</td>
<td></td>
</tr>
</tbody>
</table>

Which operation

Where to find operand1

The 0 Address Instruction

- Uses a push down stack in CPU
- Arithmetic uses stack for both operands and the result
- Computer must have a 1 address instruction to push and pop operands to and from the stack

Memory

CPU

Op1Addr: Op1

TOS
SOS

etc.

Stack

Program counter

Where to find next instruction

Instruction format

<table>
<thead>
<tr>
<th>Bits:</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Op1Addr</td>
</tr>
</tbody>
</table>

Which operation

Where to find operands, and where to put result (on the stack)
Example: Expression evaluation for 3-0 address instructions.

Evaluate \( a = (b+c) \times d - e \)

<table>
<thead>
<tr>
<th>3-address</th>
<th>2-address</th>
<th>1-address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>add a, b, c</td>
<td>load a, b</td>
<td>load b</td>
<td>push b</td>
</tr>
<tr>
<td>mpy a, a, d</td>
<td>add a, c</td>
<td>add c</td>
<td>push c</td>
</tr>
<tr>
<td>sub a, a, e</td>
<td>mpy a, d</td>
<td>mpy d</td>
<td>add</td>
</tr>
<tr>
<td>sub b, e</td>
<td>push d</td>
<td>push e</td>
<td>store a</td>
</tr>
</tbody>
</table>

- # of instructions & # of addresses both vary
- Discuss as examples: size of code in each case

Real Machines are Not So Simple

- Most real machines have a mixture of 3, 2, 1, 0, 1 1/2 address instructions
- A distinction can be made on whether arithmetic instructions use data from memory
- If ALU instructions only use registers for operands and result, machine type is load-store (or called register-register)
  - Only load and store instructions reference memory
  - Alpha, ARM, MIPS, IBM PowerPC, Sun SPARC (reg-reg)
  - Intel 80x86, Motorola 68000 (reg-mem)
- Other machines have a mix of register-memory and memory-memory instructions

Memory Addressing

- Assumption: Bytes(8 bit), Half words(16 bits), Words(32 bits), Double words(64 bits)
- Bytes ordering within a Word:
  - Little Endian – Intel x86
  3 2 1 0
  00000000 00000000 00000000 00000100
  - Big Endian – Sun SPARC
  0 1 2 3
  00000100 00000000 00000000 00000000
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

- Bytes are nice, but most data items use larger "words".
- For MIPS R3000, a word is 32 bits or 4 bytes.
- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$

Words are aligned

- i.e., what are the least 2 significant bits of a word address?

Memory Addressing

- Alignment: Accessing data larger than a byte must be aligned. The definition is that an access to data of size $S$ bytes at byte address $A$ is aligned if $A \mod S = 0$.
- Example: half word is 2 bytes long, therefore, the address is aligned at byte offset 0, 2, 4, 6, and misaligned at byte offset 1, 3, 5, 7.

- (See Figure 2.5 in the book for complete list)

Addressing Modes

- An addressing mode is hardware support for a useful way of determining a memory address.
- Different addressing modes solve different HLL problems.
  - Some addresses may be known at compile time, e.g. global vars.
  - Others may not be known until run time, e.g. pointers.
  - Addresses may have to be computed. Examples include:
    - Record (struct) components:
      - variable base(full address) + const.(small)
    - Array components:
      - const. base(full address) + index var.(small)
  - Possible to store constant values w/o using another memory cell by storing them with or adjacent to the instruction itself.
HLL Examples of Structured Addresses

- **C language**: `rec -> count`
  - `rec` is a pointer to a record: full address variable
  - `count` is a field name: fixed byte offset, say 24

- **C language**: `v[i]`
  - `v` is fixed base address of array: full address constant
  - `i` is name of variable index: no larger than array size

- Variables must be contained in registers or memory cells
- Small constants can be contained in the instruction
- Result: need for “address arithmetic.”
  - E.g. Address of `Rec -> Count` is address of `Rec + offset of count`.

Common Addressing Modes

- **Immediate Addressing**
  (Instruction contains the operand.)
- **Direct Addressing**
  (Instruction contains address of operand)
- **Indirect Addressing**
  (Instruction contains address of address of operand)
- **Register Indirect Addressing**
  (Register contains address of operand)
- **Displacement (Based) (Indexed) Addressing**
  (Address of operand = register + constant)
- **Relative Addressing**
  (Address of operand = PC + constant)

Big ideas of ISAs

- How many arguments for an ALU operation are explicit?
- How is memory addressed?
- How is I/O spoken to?
- Fixed instruction size?
  - If so, what size?

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)
MIPS arithmetic

Of course this complicates some things...

C code: \[ A = B + C + D; \]
\[ E = F - A; \]

MIPS code:
\[
\begin{align*}
&\text{add } \$t0, \$s1, \$s2 \\
&\text{add } \$s0, \$t0, \$s3 \\
&\text{sub } \$s4, \$s5, \$s0
\end{align*}
\]

• Operands must be registers, only 32 registers provided

Instructions

• Load and store instructions
• Example:
  
  C code:
  
  \[ A[8] = h + A[8]; \]
  
  MIPS code:
  
  \[
  \begin{align*}
  &\text{lw } \$t0, 32(\$s3) \\
  &\text{add } \$t0, \$s2, \$t0 \\
  &\text{sw } \$t0, 32(\$s3)
  \end{align*}
  \]

• Store word has destination last
• Remember arithmetic operands are registers, not memory!

So far we’ve learned:

• MIPS
  – loading words but addressing bytes
  – arithmetic on registers only

• Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>

Control

• Decision making instructions
  – alter the control flow,
  – i.e., change the "next" instruction to be executed

• MIPS conditional branch instructions:
  
  \[
  \begin{align*}
  &\text{bne } \$t0, \$t1, Label \\
  &\text{beq } \$t0, \$t1, Label
  \end{align*}
  \]

• Example: \text{if (i==j) } h = i + j; 
  
  \[
  \begin{align*}
  &\text{bne } \$s0, \$s1, Label \\
  &\text{add } \$s3, \$s2, \$t0 \\
  &\text{add } \$s1, \$s0, \$s1 \\
  &\text{Label: } ....
  \end{align*}
  \]
Control

- MIPS unconditional branch instructions:
  - `j label`
  - Example:
    ```
    if (i!=j) beq $s4, $s5, Lab1
    h=i+j; add $s3, $s4, $s5
    else
      h=i-j; Lab1:sub $s3, $s4, $s5
      Lab2:...
    ```

- Can you build a simple for loop?

So far:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1,$s2,$s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1,100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1,100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
<tr>
<td>bne $s4,$s5,L</td>
<td>Next instr. is at Label if $s4 ≠ $s5</td>
</tr>
<tr>
<td>beq $s4,$s5,L</td>
<td>Next instr. is at Label if $s4 = $s5</td>
</tr>
<tr>
<td>j Label</td>
<td>Next instr. is at Label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formats:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>J</td>
</tr>
</tbody>
</table>

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  ```
  if $s1 < $s2 then
    $t0 = 1
  else
    $t0 = 0
  ```
- Can use this instruction to build "blt $s1, $s2, Label"
  - can now build general control structures
- Note that the assembler needs a register to do this,
  - there are policy of use conventions for registers

Addresses in Branches and Jumps

<table>
<thead>
<tr>
<th>Instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bne $t4,$t5,Label</td>
</tr>
<tr>
<td>beq $t4,$t5,Label</td>
</tr>
<tr>
<td>j Label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formats:</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
</tr>
<tr>
<td>J</td>
</tr>
</tbody>
</table>

- Addresses are not 32 bits
  - How do we handle this with load and store
    instructions?
Addresses in Branches

- Instructions:
  - `bne $t4,$t5,Label` Next instruction is at Label if $t4$≠$t5$
  - `beq $t4,$t5,Label` Next instruction is at Label if $t4$=$t5$

- Formats:
  - Could specify a register (like lw and sw) and add it to address
    - use Instruction Address Register (PC = program counter)
    - most branches are local (principle of locality)
  - Jump instructions just use high order bits of PC
    - address boundaries of 256 MB

To summarize (see Fig.2.31):

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r31</th>
<th>PC</th>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
</table>

**Programmable storage**
- $2^{32}$ x bytes
- 31 x 32-bit GPRs (R0=0)
- 32 x 32-bit FP regs (paired DP)
- HI, LO, PC

**Arithmetic logical**
- Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU,
- AddI, AddIU, SLTI, SLTIU, AndI, OrI, Xori, LUI,
- SLL, SRL, SRA, SLLV, SRLV, SRAV

**Memory Access**
- LB, LBU, LH, LHU, LW, LWL, LW
- SB, SH, SWL, SWR

**Control**
- J, JAL, JR, JALR
- BEq, BNE, BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL

32-bit instructions on word boundary