Computer Structure
234267

Lecturers:
Lihu Rappoport
Adi Yoaz
General Course Information

◆ Grade
  ◆ 20% : 4 exercises (all mandatory)
    • Submission in pairs
  ◆ 80% Final exam

◆ Course web site
  ◆ Foils will be on the web several days before the class
Class Focus

◆ CPU
  ❖ Introduction: performance, instruction set (RISC vs. CISC)
  ❖ Pipeline, hazards
  ❖ Branch prediction
  ❖ Out-of-order execution

◆ Memory Hierarchy
  ❖ Cache and cache coherency
  ❖ Main memory
  ❖ Virtual Memory

◆ More topics
  ❖ Multi-threading
  ❖ System
  ❖ Power considerations
Architecture & Microarchitecture

- **Architecture**
  The processor features seen by the “user”
  - Instruction set, addressing modes, data width, ...

- **Micro-architecture**
  The internal implementation of a processor
  - Caches size and structure, number of execution units, ...

- **Processors with different \( \mu \text{Arch} \) can support the same architecture**

- **Compatibility**
  - A new processor can run existing software
  - The processor \( \mu \text{Arch} \) is new, but it supports the **Architecture** of past generations, possibly adding to it
Moore’s Law

The number of transistors doubles every ~2 years

Transistor count

CPI – Cycles Per Instruction

- **CPUs work according to a clock signal**
  - Clock cycle is measured in nsec (10^-9 of a second)
  - Clock frequency (= 1/clock cycle) measured in GHz (10^9 cyc/sec)

- **Instruction Count (IC)**
  - Total number of instructions executed in the program

- **CPI – Cycles Per Instruction**
  - Average #cycles per Instruction (in a given program)
  - \[ \text{CPI} = \frac{\text{#cycles required to execute the program}}{\text{IC}} \]
  - IPC (= 1/CPI) : Instructions per cycles
Calculating the CPI of a Program

- **IC_i**: #times instruction of type i is executed in the program
- **IC**: #instruction executed in the program: \( IC = \sum_{i=1}^{n} IC_i \)
- **Fi**: relative frequency of instruction of type i: \( Fi = IC_i/IC \)
- **CPI_i**: #cycles to execute instruction of type i
  - e.g.: \( CPI_{\text{add}} = 1 \), \( CPI_{\text{mul}} = 3 \)
- #cycles required to execute the entire program:
  \[
  \#cyc = \sum_{i=1}^{n} CPI_i \times IC_i = CPI \times IC
  \]
- **CPI**: 
  \[
  CPI = \frac{\#cyc}{IC} = \frac{\sum_{i=1}^{n} CPI_i \times IC_i}{IC} = \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC} = \sum_{i=1}^{n} CPI_i \times F_i
  \]
CPU Time

- CPU Time - time required to execute a program

CPU Time = IC \times CPI \times clock cycle

Our goal: minimize CPU Time

- Minimize clock cycle: more GHz (process, circuit, uArch)
- Minimize CPI: uArch (e.g.: more execution units)
- Minimize IC: architecture (e.g.: AVX™)
Suppose enhancement E accelerates a fraction $F$ of the task by a factor $S$, and the remainder of the task is unaffected, then:

$$t'_{exe} = t_{exe} \times \left( (1 - F) + \frac{F}{S} \right)$$

Speedup\textsubscript{overall} $= \frac{t_{exe}}{t'_{exe}} = \frac{1}{(1 - F) + \frac{F}{S}}$
Amdahl’s Law: Example

• Floating point instructions improved to run at $2\times$, but only 10% of executed instructions are FP

$$t'_{\text{exe}} = t_{\text{exe}} \times (0.9 + 0.1 / 2) = 0.95 \times t_{\text{exe}}$$

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

Corollary:
Make The Common Case Fast
Comparing Performance

- **Peak Performance**
  - MIPS, GFLOPS
  - Often not useful: unachievable / unsustainable in practice

- **Benchmarks**
  - Real applications, or representative parts of real apps
  - Synthetic benchmarks representative parts of real apps
  - Targeted at the specific system usages
  - SPEC INT – integer applications
    - Data compression, C compiler, Perl interpreter, database system, chess-playing, Text-processing, ...
  - SPEC FP – floating point applications
    - Mostly important scientific applications
  - TPC Benchmarks
    - Measure transaction-processing throughput
Evaluating Performance of future CPUs

- Use a performance simulator to evaluate the performance of a new feature / algorithm
  - Models the uarch to a great detail
  - Run 100’s of representative applications

- Produce the performance s-curve
  - Sort the applications according to the IPC increase
  - Baseline (0) is the processor without the new feature

![Graph showing Bad S-curve and Good S-curve with positive and negative outliers.](image)
Instruction Set Design

The ISA is what the user / compiler see

The HW implements the ISA

software

hardware
ISA Considerations

- Reduce the IC to reduce execution time
  - E.g., a single vector instruction performs the work of multiple scalar instructions

- Simple instructions $\Rightarrow$ simpler HW implementation
  - Higher frequency, lower power, lower cost

- Code size
  - Long instructions take more time to fetch
  - Longer instructions require a larger memory
    - Important in small devices, e.g., cell phones
1% of data values > 16-bits
12 – 16 bits of needed
Software Specific Extensions

- Extend arch to accelerate exec of specific apps
  - Reduce IC

- **Example: SSE™ – Streaming SIMD Extensions**
  - 128-bit packed (vector) / scalar single precision FP (4x32)
  - Introduced on Pentium® III on ’99
  - 8 new 128 bit registers (XMM0 – XMM7)
  - Accelerates graphics, video, scientific calculations, ...

- **Packed:**
  - 128-bits
  - \[x_3, x_2, x_1, x_0\] + \[y_3, y_2, y_1, y_0\] = \[x_3+y_3, x_2+y_2, x_1+y_1, x_0+y_0\]

- **Scalar:**
  - 128-bits
  - \[x_3, x_2, x_1, x_0\] + \[y_3, y_2, y_1, y_0\] = \[x_3+y_3, x_2+y_2, x_1+y_1, x_0+y_0\]
CISC Processors

- CISC – Complex Instruction Set Computer
  - The idea: a high level machine language
  - Example: x86

- Characteristic
  - Many instruction types, with many addressing modes
  - Some of the instructions are complex
    - Execute complex tasks, and require many cycles
  - Small number of registers, in many cases not orthogonal
    - E.g., some operations supported on specific registers
  - ALU operations directly on memory
    \[
    \text{add} \quad \text{eax}, \ 7680[\text{ecx}] \quad ; \text{eax} \leftarrow \text{MEM}[7680+\text{ecx}]+\text{eax}
    \]
  - Variable length instructions
    - common instructions get short codes \(\Rightarrow\) save code length
CISC Drawbacks

- **Complex instructions and complex addressing modes**
  - Complicates the processor
  - Slows down the simple, common instructions
  - Contradicts Make The Common Case Fast

- **Not compiler friendly**
  - Non orthogonal registers
  - Unused complex addressing modes

- **Variable length instructions are a pain**
  - Difficult to decode few instructions in parallel
    - As long as instruction is not decoded, its length is unknown
      - Unknown where the inst. ends, and where the next inst. starts
  - An instruction may cross a cache line or a page
## Top 10 x86 Instructions

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

Simple instructions dominate instruction frequency
**RISC Processors**

- **RISC – Reduced Instruction Set Computer**
  - The idea: simple instructions enable fast hardware

- **Characteristics**
  - A small instruction set, with few instruction formats
  - Simple instructions that execute simple tasks
    - Most of them require a single cycle (with pipeline)
  - A few indexing methods
  - ALU operations on registers only
    - Memory is accessed using Load and Store instructions only
  - Many orthogonal registers – all instructions and all addressing modes available with any register
  - Three address machine: \( \text{Add dst, src1, src2} \)
  - Fixed length instructions

- **Compiler friendly – easier to utilize the ISA**
  - Orthogonal, simple
RISC Processors (Cont.)

- **Simple architecture ➔ Simple micro-architecture**
  - Simple, small and fast control logic
  - Simpler to design and validate
  - Leave space for large on die caches
  - Shorten time-to-market

- **Existing RISC processor are not “pure” RISC**
  - E.g., support division which takes many cycles
  - Examples: MIPSTM, SparcTM, AlphaTM, PowerTM

- **Modern CISC processors use RISC μarch ideas**
  - Internally translate the CISC instructions into RISC-like operations
    - the inside core looks much like that of a RISC processor