Components of a Computer

- CPU
  - Control
  - Datapath

- Memory

- Devices
  - Input
  - Output
Levels of Program Code

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability

- **Assembly language**
  - Textual representation of instructions

- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

```
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
muli $2, $5, 4
add $2, $4,$2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```
Old School Machine Structures (Layers of Abstraction)

Software

Application (ex: browser)
  Compiler
  Assembler

Operating System (Mac OS X)

Hardware

Processor
  Memory
  I/O system

Datapath & Control
  Digital Design
  Circuit Design
  Transistors

Instruction Set Architecture
New-School Machine Structures

Software

- Parallel Requests
  Assigned to computer
e.g., Search “Katz”

- Parallel Threads
  Assigned to core
e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- Hardware descriptions
  All gates working in parallel at the same time

Hardware

Harness Parallelism & Achieve High Performance

Warehouse Scale Computer

Smart Phone

Computer
Core ...
Memory (Cache)
Input/Output
Core
Instruction Unit(s)
Functional Unit(s)
A + B, A + B, A + B, A + B
Main Memory
Logic Gates

Ning Xie (http://www.csd.uwo.ca/courses/CS3350B Computer Architecture CPU Performance analysis Winter, 2016)
Why do computers become so complicated?

- Eight Great Ideas in Pursuing performance
  - Design for Moore’s Law
  - Use abstraction to simplify design
  - Make the common case fast
  - Performance via parallelism
  - Performance via pipelining
  - Performance via prediction
  - Hierarchy of memories
  - Dependability via redundancy
Understanding Performance

- **Algorithm**
  - Determines number of operations executed

- **Programming language, compiler, architecture**
  - Determine number of machine instructions executed per operation

- **Processor and memory system**
  - Determine how fast instructions are executed

- **I/O system (including OS)**
  - Determines how fast I/O operations are executed
Performance Metrics

- Purchasing perspective
  given a collection of machines, which has the
  - best performance?
  - least cost?
  - best cost/performance?

- Design perspective
  faced with design options, which has the
  - best performance improvement?
  - least cost?
  - best cost/performance?

- Both require
  - basis for comparison
  - metric for evaluation

- Our goal is to understand what factors in the architecture contribute to overall system performance and the relative importance (and cost) of these factors
CPU Performance

- Normally interested in reducing
  - Response time (aka execution time) – the time between the start and the completion of a task
    - Important to individual users
  - Thus, to maximize performance, need to minimize execution time
    \[
    \text{performance}_X = \frac{1}{\text{execution\_time}_X}
    \]
    If \( X \) is \( n \) times faster than \( Y \), then
    \[
    \frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n
    \]
- And increasing
  - Throughput - the total amount of work done in a given time
    - Important to data center managers
  - Decreasing response time almost always improves throughput
Almost all computers are constructed using a clock that determines when events take place in the hardware.

- **Clock period (cycle):** duration of a clock cycle
  - determines the speed of a computer processor
  - e.g., $250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$

- **Clock frequency (rate):** cycles per second
  - the inverse of the clock period
  - e.g., $3.0\text{GHz} = 3000\text{MHz} = 3.0 \times 10^9\text{Hz}$

- $CR = 1 / CC$
Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) - time the CPU spends working on a task
  - Does not include time waiting for I/O or running other programs
    
    \[
    \text{CPU execution time} = \#\text{CPU clock cycles} \times \text{clock cycle for a program}
    \]
    
    or
    
    \[
    \text{CPU execution time} = \#\text{CPU clock cycles} / \text{clock rate for a program}
    \]

- Can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program
Instruction Performance

\[
\text{#CPU clock cycles} = \text{#Instructions} \times \text{Average # of clock cycles per instruction for a program}
\]

- **Clock cycles per instruction (CPI)** - the average number of clock cycles each instruction takes to execute
  - Different instructions may take different amounts of time depending on what they do
  - A way to compare two different implementations of the same instruction set architecture (ISA)
The Classic Performance Equation

\[ \text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle} \]

or

\[ \text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}} \]

- Always bear in mind that the only complete and reliable measure of computer performance is time.

- For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time or higher CPI that offsets the improvement in instruction count.

- Similarly, because CPI depends on type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.
Overall effective CPI = \[ \sum_{i=1}^{n} (CPI_i \times IC_i) \]

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>CPI(_i)</th>
<th>Freq \times CPI(_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\sum = 2.2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.6</td>
</tr>
</tbody>
</table>

(1) How much faster would the machine be if a better data cache reduced the average load time to 2 cycles? 

CPU time new = 1.6 \times IC \times CC so 2.2/1.6 means 37.5% faster
Overall effective CPI = \( \sum_{i=1}^{n} (CPI_i \times IC_i) \)

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>CPI(_i)</th>
<th>Freq \times CPI(_i)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>.5</td>
<td>.25</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>.3</td>
<td>.3</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>.2</td>
<td>.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\sum = 2.2)</td>
<td></td>
<td>2.0</td>
<td>1.95</td>
</tr>
</tbody>
</table>

(2) How does this compare with using branch prediction to shave a cycle off the branch time?

CPU time new = 2.0 \times IC \times CC so 2.2/2.0 means 10% faster

(3) What if two ALU instructions could be executed at once?

CPU time new = 1.95 \times IC \times CC so 2.2/1.95 means 12.8% faster
Understanding Program Performance

CPU time = Instruction_count × CPI × clock_cycle

- The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware.

<table>
<thead>
<tr>
<th></th>
<th>Instruction_count</th>
<th>CPI</th>
<th>clock_cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Programming language</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Processor organization</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Performance Summary

CPU Time = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, $T_c$
A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\frac{15 \times 0.6}{1.1} = 8.2$ sec
b. $15 \times 0.6 \times 1.1 = 9.9$ sec
c. $\frac{15 \times 1.1}{0.6} = 27.5$ sec
In complementary metal oxide semiconductor (CMOS) integrated circuit technology

\[
\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched} \times 30 \times 1000
\]

(5V → 1V)
Suppose a new CPU has

- 85% of capacitive load of old CPU
- 15% voltage and 15% frequency reduction

\[
\frac{P_{\text{new}}}{P_{\text{old}}} = \frac{C_{\text{old}} \times 0.85 \times (V_{\text{old}} \times 0.85)^2 \times F_{\text{old}} \times 0.85}{C_{\text{old}} \times V_{\text{old}}^2 \times F_{\text{old}}} = 0.85^4 = 0.52
\]

The power wall

- We can't reduce voltage further
- We can't remove more heat

How else can we improve performance?
Uniprocessor Performance

- Constrained by power, instruction-level parallelism, memory latency
Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization
SPEC CPU Benchmark

- Programs used to measure performance
  - Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - Develops benchmarks for CPU, I/O, Web, ...
- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

\[
\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}
\]
<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Instruction Count x 10^9</th>
<th>CPI</th>
<th>Clock cycle time (seconds x 10^-9)</th>
<th>Execution Time (seconds)</th>
<th>Reference Time (seconds)</th>
<th>SPEC ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>perl</td>
<td>2252</td>
<td>0.60</td>
<td>0.376</td>
<td>508</td>
<td>9770</td>
<td>19.2</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>bzip2</td>
<td>2390</td>
<td>0.70</td>
<td>0.376</td>
<td>629</td>
<td>9650</td>
<td>15.4</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>gcc</td>
<td>794</td>
<td>1.20</td>
<td>0.376</td>
<td>358</td>
<td>8050</td>
<td>22.5</td>
</tr>
<tr>
<td>Combinatorial optimization</td>
<td>mcf</td>
<td>221</td>
<td>2.66</td>
<td>0.376</td>
<td>221</td>
<td>9120</td>
<td>41.2</td>
</tr>
<tr>
<td>Go game (AI)</td>
<td>go</td>
<td>1274</td>
<td>1.10</td>
<td>0.376</td>
<td>527</td>
<td>10490</td>
<td>19.9</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>hmmer</td>
<td>2616</td>
<td>0.60</td>
<td>0.376</td>
<td>590</td>
<td>9330</td>
<td>15.8</td>
</tr>
<tr>
<td>Chess game (AI)</td>
<td>sjeng</td>
<td>1948</td>
<td>0.80</td>
<td>0.376</td>
<td>586</td>
<td>12100</td>
<td>20.7</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>libquantum</td>
<td>659</td>
<td>0.44</td>
<td>0.376</td>
<td>109</td>
<td>20720</td>
<td>190.0</td>
</tr>
<tr>
<td>Video compression</td>
<td>h264avc</td>
<td>3793</td>
<td>0.50</td>
<td>0.376</td>
<td>713</td>
<td>22130</td>
<td>31.0</td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>omnetpp</td>
<td>367</td>
<td>2.10</td>
<td>0.376</td>
<td>290</td>
<td>6250</td>
<td>21.5</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>aistar</td>
<td>1250</td>
<td>1.00</td>
<td>0.376</td>
<td>470</td>
<td>7020</td>
<td>14.9</td>
</tr>
<tr>
<td>XML parsing</td>
<td>xalancbmk</td>
<td>1045</td>
<td>0.70</td>
<td>0.376</td>
<td>275</td>
<td>6900</td>
<td>25.1</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>25.7</td>
</tr>
</tbody>
</table>
Many profiling tools

- gprof (static instrumentation)
- cachegrind, Dtrace (dynamic instrumentation)
- perf (performance counters)

perf in linux-tools, based on event sampling

- Keep a list of where “interesting events” (cycle, cache miss, etc) happen
- CPU Feature: Counters for hundreds of events
  - Performance: Cache misses, branch misses, instructions per cycle, ...

Intel®64 and IA-32 Architectures Software Developer’s Manual: Appendix A lists all counters

perf user guide: https://perf.wiki.kernel.org/index.php/Tutorial
Exercise 1

```c
void copymatrix1(int (*src)[n],
                 int (*dst)[n], int n) {
    int i,j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            dst[i][j] = src[i][j];
}

void copymatrix2(int (*src)[n],
                 int (*dst)[n], int n) {
    int i,j;
    for (j = 0; j < n; j++)
        for (i = 0; i < n; i++)
            dst[i][j] = src[i][j];
}
```

- copymatrix1 vs copymatrix2
  - What do they do?
  - What is the difference?
  - Which one performs better? Why?

- `perf stat -e cycles -e cache-misses ./copymatrix1`
  - What does the output like?
  - How to interpret it?
  - Which program performs better?
void lower1 (char* s) {
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= ’A’ && s[i] <= ’Z’)
            s[i] -= ’A’-’a’;
}

void lower2 (char* s) {
    int i;
    int n = strlen(s);
    for (i = 0; i < n; i++)
        if (s[i] >= ’A’ && s[i] <= ’Z’)
            s[i] -= ’A’-’a’;
}

- lower1 vs lower2
  - What do they do?
  - What is the difference?
  - Which one performs better? Why?

- perf stat -e cycles -e cache-misses ./lower1
  perf stat -e cycles -e cache-misses ./lower2
  - What does the output like?
  - How to interpret it?
  - Which program performs better?