Technology Trends

- CPU speed doubles every 18 months (Moore's Law)
- Memory speed doubles every 10 years ➔ CPU-Memory Speed Gap
- Memory density quadruples every 2 years

Storage Hierarchy Properties

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Access Time</th>
<th>Cost/MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Integer Registers</td>
<td>256 B</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Primary Cache (SRAM)</td>
<td>36 KB</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Secondary Cache</td>
<td>1 MB</td>
<td>4-30 ns</td>
<td>?</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
<td>512 MB</td>
<td>30-60 ns</td>
<td>$0.30</td>
</tr>
<tr>
<td>Disk Drive</td>
<td>160 GB</td>
<td>8-30 ms</td>
<td>$0.0008</td>
</tr>
</tbody>
</table>
Evolution of Intel Processor Features

<table>
<thead>
<tr>
<th>Processor</th>
<th>Date</th>
<th>Frequency</th>
<th>Transistors</th>
<th>Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>60 MHz</td>
<td>3.1 M</td>
<td>L1: 16 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L2: 256 KB</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>200 MHz</td>
<td>5.5 M</td>
<td>L1: 16 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L2: 256 KB</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>266 MHz</td>
<td>7 M</td>
<td>L1: 32 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L2: 256 KB</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>500 MHz</td>
<td>8.2 M</td>
<td>L1: 32 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L2: 512 KB</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>1.5 GHz</td>
<td>42 M</td>
<td>*L1: 8 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*L2: 256 KB</td>
</tr>
<tr>
<td>Xeon</td>
<td>2002</td>
<td>1.70 GHz</td>
<td>42 M</td>
<td>*L1: 8 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*L2: 512 KB</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2004</td>
<td>2.00 GHz</td>
<td>140 M</td>
<td>*L1: 64 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*L2: 2 MB</td>
</tr>
</tbody>
</table>

* On-die caches

Clock Speeds

<table>
<thead>
<tr>
<th>500 MHz Pentium</th>
<th>CPU</th>
<th>250 MHz SKAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td></td>
<td>L2 Cache</td>
</tr>
<tr>
<td>Video</td>
<td></td>
<td>Main Memory</td>
</tr>
<tr>
<td>Bridge</td>
<td></td>
<td>100 MHz DDR</td>
</tr>
<tr>
<td>PCI Bus (32 or 64 bits)</td>
<td></td>
<td>33 MHz</td>
</tr>
</tbody>
</table>

Example

**Pseudo-Code:**
```
i = 0; j = 0
for (i < N) {
    for (j < N) { A[i][j] = B[j] * A[i][j]; }
}
```

**Machine Pseudo-Code:**
```
// R1 contains address of A[i][j]
L1: if (outer loop done) Exit outer loop;
    Initialize pointer R2;
L2: if (inner loop done) Exit inner loop;
    R3 <-- *R1; // Load register 3 from memory
    R4 <-- *R2; // Load register 4 from memory
    R5 <-- R3 * R4; // Multiply
    *R1 <-- R5; // Store result back to memory
    R1 += 1; // Move pointer to next element
    R2 += 1; // Move pointer to next element
goto L2; // End of inner loop
goto L1; // End of outer loop
```

General Comments

- The previous slide is an over simplification!!!
- Arithmetic (and logic) operations are very simple
  - Requires bringing data from memory to the ALU (Arithmetic and Logic Unit)
  - Operations are actually done in general-purpose registers
- There are $3N^2$ arithmetic operations and at least $2N^2$ memory reads, $N^2$ memory writes
- Large N Execution Time (Naive): $3N^2 t_1 + 3N^2 t_2$
- Large N Execution Time (Likely): $3N^2 t_3$
- CPU-Memory access concurrency
- Bottleneck: Main memory accesses →
- Execution time will be dominated by memory access pattern and memory speed !!!
Basic Instruction Execution

- Start
- Fetch Next Instruction
- Register Inst.
- Fetch Data
- Execute
- PC
- IR
- Memory Inst.
- Done

Sources Of CPU Speed
- RISC (Reduced Instruction Set Computer)
  » Simpler instructions allow shorter clock cycle times
- Shorter Wires
  » Signal gets to destination faster: Smaller cycle time
- Instruction Pipelining
  » Stages: Fetch, Decode, Get Source, Execute, Complete
- Superscalar (Multiple instruction stream exec)
  » Out-of-Order Instruction Execution/Completion
- Branch Prediction
  » Avoid instruction stall preceding comparison result by executing code based on predicted comparison

Processor Registers
- What types of registers can be found in a simple integer RISC CPU?
  » Program Counter (PC): Address of next instruction
  » Instruction Register (IR): The most recently fetched instruction
  » Status Register (SR): Results of comparisons, errors, etc.
  » Stack Pointer (SP): Address of the top stack element
  » General Registers (R[0], R[31]): Operands
- How are these registers used during program execution?

Pipelining
- Max speed of 1 MHz pipeline: 1 MIPS (1,000,000 Instructions Per Second)
Cache Memory

- L1 cache typically dedicates a part only to instructions.
- Arithmetic operations are only done in general registers. Must read x from main memory into cache(s) and into a general register.

Cache Memory Operation

- Copy data to storage that is closer (and faster) to its place of use.
- Decreases access time to the cached data on subsequent accesses.

Read Operation

```c
if (X is in cache) { // Hit: T1
    Read from cache;
} else { // Miss: T2 + T1
    Read from memory into cache;
    Read from cache;
}
```

Cache Design

- EMAT: Effective Memory Access Time
- \( EMAT = \left( \frac{H \times T1 + M \times (T1 + T2)}{H + M} \right) \)
  - H: Number of cache hits
  - M: Number of cache misses
  - T1: Cache access time
  - T2: Memory access time
  - This form is closer to the operational steps of caching.
- \( EMAT = h \times T1 + (1-h) \times (T1 + T2) \)
  - h: Hit ratio (H / (H + M))
  - This form is easier to use.
- \( EMAT = T1 + m \times T2 \) where \( m = (1-h) \)
  - m: Miss ratio
  - This form indicates that you must always pay the cost of a cache access.

EMAT Example

- Consider:
  - Memory access time \( T2 = 50 \text{ ns} \)
  - Cache access time \( T1 = 4 \text{ ns} \)
- What is the EMAT if \( h = 0.90 \)?
  - \( EMAT = 0.90 \times 4 + 0.10 \times 54 = 9.0 \text{ ns} \)
  - \( EMAT = 4 + 0.10 \times 50 = 9.0 \text{ ns} \)
- What is the EMAT if \( h = 0.95 \)?
  - \( EMAT = 0.95 \times 4 + 0.05 \times 54 = 6.5 \text{ ns} \)
  - \( EMAT = 4 + 0.05 \times 50 = 6.5 \text{ ns} \)
- What size data cache would be required in our example to get an EMAT of 8 ns?
  - Depends on organization of cache, memory layout, ...
**Cache Memory**

Consider the following program:

```c
S = 0;
for (i=0; i < N; i++) { S = S + x[i]; }
T = 0;
for (i=0; i < N; i++) { T = T + x[i] * x[(i%4)*100]; }
```

Would cache memory speed up execution?

- The inner loops are small pieces of code that can be fetched repeatedly from cache memory (Spatial Locality) as it is executed.
- For small enough N, the data array x[] can be stored entirely in cache and can be accessed from cache in the second loop (Spatial Locality).
- The inner loops are small pieces of code that can be fetched repeatedly from cache memory (Spatial Locality) as it is executed.
- The variables x[0], x[100], x[200], x[300] are repeatedly accessed in the second loop (Temporal Locality), i.e., closely accessed in time.

**Machine Pseudo-code**

```c
// R1 initially contains address of A[i][j]
L1: if (outer loop done) Exit outer loop;
    Initialize pointer R2; // R2 contains address of B[j]
L2: if (inner loop done) Exit inner loop;
    R3 <-- *R1; // Load register 3 from memory
    R4 <-- *R2; // Load register 4 from memory
    R5 <-- R3 * R4; // Multiply
    *R1 <-- R5; // Store result back to memory
    R1 = R1+4; // Move pointer to next element
    R2 = R2+4; // Move pointer to next element
    goto L2; // End of inner loop
    goto L1; // End of outer loop
```

**CPU And Memory Time**

**CPU Time (If no memory waits)**
- Execute inner loop 1M (= 1K x 1K) times
- 8 instructions per iteration → 8 million instructions → 16 ms (8/500 sec)

**Memory Time**
- Memory Accesses: 2M integer reads; 1M integer writes;
- 50 ns per integer → 3M x 50 ns = 150 ms

**Significantly Reduce Run-Time??**
- Significantly increase memory speed (hard to do)
- Use a memory hierarchy (i.e., cache memory)
- Layout data to increase hit ratio; restructure algorithm
  - e.g. Try to keep B[][] array in L1 cache
Memory Technology

- **DRAM (Dynamic RAM)**
  - Main memory technology
  - Very dense (1 bit = 1 transistor and 1 capacitor) and inexpensive
  - Requires periodic refresh (by reading) ➔ Slows down overall speed
  - Cycle time = 2 × Access time (Precharge Time)

- **SRAM (Static RAM)**
  - Cache memory technology
  - 4-6 transistors and 0 capacitors per bit ➔
    - No refresh period (no charge leakage)
    - No precharge time
    - Faster access time and Lower density

DRAM Burst Mode

- **Operation**: Load address; Read/Write data
- **5-5-5-5 Memory**
  - 5 cycles to read 8 bytes (two long integers); 20 cycles for 32 bytes
  - 66 MHz Memory Bus ➔ 75 ns = 5 × 15 ns to read 8 bytes
  - If 100 MHz Memory Bus ➔ 50 ns to read 8 bytes
  - The 5 cycles includes cycle time and latency time

- **5-1-1-1 Memory (Burst Mode)**
  - Applies to consecutive memory accesses
  - Best case: 8 cycles to get 32 bytes ➔ 2 cycles for every 8 bytes
  - Worst case: 5 cycles to get 8 bytes and rest

Life Of A System Call

- **User Process**
  - `read(fd, buf, nb)`
  - `syscall(proc, arg, ...)`
  - `syscall (proc, arg, ...)` which traps into kernel
  - Enter kernel mode (privileged instructions and access to all of memory)
  - Send read request to controller of I/O device
  - Put process on queue while waiting for I/O to complete
  - Save process state and switch control to another process

- **I/O Device**
  - Transmit data to the controller’s buffer

- **Device Controller**
  - Initiate operation on I/O device
  - Request use of bus and transfer data to main memory after bus grant
  - Interrupt CPU when read request has finished
  - Request CPU when read request has finished

- **CPU**
  - Interrupt transfers CPU control to the interrupt service routine
  - Save state of interrupted process
  - Disable all other interrupts while handling interrupt
  - Give control of CPU to a process selected by the scheduler (restore CPU state and copy data from kernel buffer to user buffer)
  - Reenable interrupts, restore user-mode and return from syscall

Life Of A Device Read Request

- **CPU**
  - `read()` ultimately executes `syscall()` which traps into kernel
  - Enter kernel mode (privileged instructions and access to all of memory)
  - Send read request to controller of I/O device
  - Put process on queue while waiting for I/O to complete
  - Save process state and switch control to another process

- **Device Controller**: Initiate operation on I/O device

- **I/O Device**: Transmit data to the controller’s buffer

- **Device Controller**
  - Request use of bus and transfer data to main memory after bus grant
  - Interrupt CPU when read request has finished

- **CPU**
  - Interrupt transfers CPU control to the interrupt service routine
  - Save state of interrupted process
  - Disable all other interrupts while handling interrupt
  - Give control of CPU to a process selected by the scheduler (restore CPU state and copy data from kernel buffer to user buffer)
  - Reenable interrupts, restore user-mode and return from syscall
Privileged Instructions

- **Examples**
  - Change virtual memory protection bits
  - Disable/enable interrupt
  - Load timer registers

- **Two (2) Modes of Operation**
  - **User Mode**: Execute instructions in user program
  - **Supervisor (Kernel) Mode**: Execute instructions in operating system
  - Mode bit in status register indicates the execution mode

- **Causes of mode switching**
  - Interrupts
  - System calls

- Privileged instructions can only be used in supervisor mode

CPU Protection

- **Timer**
  - Interrupts computer at periodic time interval (time slice or quantum)
  - Used to implement time-sharing
  - Timer is a counter
    - That is decremented every clock tick
    - Causes a timer interrupt when it reaches 0

- **Time of day computed from timer value**

- **Load timer is a privileged instruction**

Memory Protection

- **Needed To**
  - Prevent one process from accessing another process' memory
  - Prevent user from directly accessing system resources or executing privileged instructions
    - Interrupt vectors and Resource tables
    - Virtual memory maps (MMU, TLB, main memory)
    - Flushing cache memory

- **Approaches**
  - Base and limit registers determine range of legal addresses
  - **Virtual Memory**
    - Each virtual page is mapped to a physical (actual) memory page
    - Each page has access flags and a "dirty bit"
    - Some pages may actually reside on disk (swap area)

Direct Memory Access (DMA)

- **For high-speed I/O devices that can transmit at memory speeds**

- **Device controller transfers blocks of data to/from main memory buffer without CPU intervention**

- **I/O Controller registers are mapped to main memory addresses**
  - Block location descriptor (defines location of block on device)
  - Transmission size (number of bytes to transfer)
  - Address of read/write buffer in main memory
  - Request status (e.g., GO bit, DONE bit, ERROR bit, error code)

- **Setting up a DMA read**
  - Set block descriptor, block count, and buffer address
  - Set GO bit (I/O device asynchronously handles request)
**Polled vs Interrupt-Driven I/O**

- **Polled I/O**
  - CPU periodically checks DONE bit in device status register
  - Disadvantage: Waste CPU cycles since CPU is so much faster than I/O
    - Especially if there are a large number of I/O devices

- **Interrupt-Driven I/O**
  - Interrupt CPU when I/O is done
    - OS saves state of the CPU (registers and program counter)
    - Interrupt vector contains location of interrupt handling code
  - Key to good response time in multitasking OS
  - Drawback: the OS overhead associated with each interrupt
    - This is a problem in real-time systems (100s of I/Os per sec)
    - An Approach: Use a clock to periodically interrupt CPU at which time CPU polls devices for real-time applications.

**Themes**

- **Parallel Execution ➔ Higher Speed**
- **Cache Hierarchy ➔ Hide Slower Memory ➔**
  - Increase Effective Memory Speed
- **Protection ➔ Higher Reliability**
  - Price might be higher overhead ➔ Slower Speed