Matrix Multiply Example

C-Level Code:
```
int A[N][N], B[N][N], C[N][N];
for (int i=0; i < N; i++)
    for (j=0; j < N; j++)
        C[i][j] = 0;
for (k=0; k < N; k++)
    { C[i][j] = C[i][j] + (A[i][k] * B[k][j]); }
```

Machine-Level:
```
L: if (inner loop done) Exit inner loop;
R4 <-- *R1; // Load register 4 from memory
R5 <-- *R2; // Load register 5 from memory
R4 <-- R5 * R4; // Multiply
R6 <-- R6 + R4; // Value of C[i][j]
*R3 <-- R6; // Store result back to memory
R1 = R1 + 4; // Move pointer to next A[[][]]
R2 = R2 + ... ; // Move pointer to next B[[][]]
goto L; // End of inner loop
```

Hardware Context

Processor Registers
- **Program Counter (PC)**
  » Contains address of next instruction
- **Instruction Register (IR)**
  » Contains the most recently fetched instruction
- **Status Register (SR)**
  » Results of comparisons, errors, etc.
  » Sometimes called **Processor Status Word (PSW)**
- **Stack Pointer (SP)**
  » Address of the top stack element
- **General Registers (R[0]..R[7])**
  » Operands
**Pentium Clock Speeds**

- 500 MHz Pentium
- 250 MHz SRAM
- L1 Cache
- L2 Cache
- 100 MHz SDRAM
- 100 MHz PCI Bus (32 or 64 bits)
- 33 MHz PCI Bus
- 100 MHz, 32-Bit Processor Bus
- 10 nsec
- Video
- Main Memory
- Bridge
- Controller
- ... 30.3 nsec
- Controller
- Network
- Need for asynchrony between components
  - Buffers smooth out traffic
  - May still get “stalls”

**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>Pentium Pro</td>
<td>1995</td>
<td>200 MHz</td>
<td>5.5 M</td>
<td>L1: 16 KB, 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, 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, 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, *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, *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, *L2: 2 MB</td>
</tr>
</tbody>
</table>

- * On-die caches
- 1 MHz = 10^6 cycles per sec, 1 GHz = 10^9

**Pipelining Concept**

- Sequential Process Example (4 operations)
  - \( \text{op}(0); \text{op}(1); \text{op}(2); \text{op}(3); \)
  - Time: \( t(0) + t(1) + t(2) + t(3) \)

- Pipelining Opportunity:
  - Apply process to stream of data: \( x[0], x[1], \ldots \)

- Ideal Pipeline of Example ( \( t(i) = 1, \text{all } i \) )

```
0    |  |   |   |   |   |    \\
1    | op(0) |   |   |   |   |    \\
5    |  |   |   |   |   |   |    \\
6    |  |   |   |   |   |   |    \\
7    |  |   |   |   |   |   |    \\
```

**Basic Instruction Execution**

- Start
  - Fetch Next Instruction
  - Execute
  - Fetch Data
  - Execute
  - Memory Inst.
  - Done

- Real processors have many more stages

- Register Inst.
- Execute
- Fetch Data
Pipelining Instruction Execution

- A Gantt Chart is a space-time diagram

<table>
<thead>
<tr>
<th>CPU Clock Tick</th>
<th>Fetch I</th>
<th>Fetch D</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**Ideal Case**
Starting at tick 3, 1 instruction finishes every tick.

CPU can "stall" for various reasons (memory wait).

### 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 (L1) Cache (SRAM)</td>
<td>36 KB</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Secondary (L2) 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>

### Technology Trends

- CPU speed doubles every 18 months (Moore’s Law)
- Memory speed doubles every 10 years
- But memory density quadruples every 2 years!
- Cache memories attempt to bridge CPU-memory gap

### 2-Level Cache Memory

- L1 cache
  - Typically split between instructions and data
- Arithmetic operations
  - Are done in General Registers (GRs)
  - Must read X from main memory into cache(s) and then into a GR
Basic Cache Memory Operation

- **Cache Memory**
  - Copy data to storage that is closer (and faster) to its place of use
  - Decrease access time on repeated accesses
- **Read Memory Operation**
  ```
  if (X is in cache) { // Hit
    Read X from cache; // T1
  } else { // Miss
    Read X from memory into cache; // T2
    Read X from cache; // + T1
  }
  ```

- Cache design is complicated

---

EMAT

**EMAT: Effective Memory Access Time**

**EMAT** = \[ \frac{H \times T_1 + M \times (T_1 + T_2)}{H + M} \]

- H: Number of cache hits
- M: Number of cache misses
- T1: Cache access time
- T2: Memory access time
- Form closer to the operational steps of caching

**EMAT** = \( h \times T_1 + (1-h) \times (T_1 + T_2) \)

- h: Hit ratio \( \frac{H}{H + M} \)
- This form is easier to use

**EMAT** = \( T_1 + m \times T_2 \) where \( m = (1-h) \)

- m = 1-h: Miss ratio
- Indicates you always pay the cost of a cache access

---

Cache Memory Example

```plaintext
for (i=0; i < N; i=i+M) { x[i] = x[i] + 1; }
```

**L:**
- if (loop done) Exit inner loop;
- \( R2 \leftarrow \ast R1; \) // Load R2 with \( x[i] \) from memory
- \( R2 \leftarrow R2 + 1; \) // Increment
- \( R2 \leftarrow \ast R1; \) // Store \( x[i] \)
- \( R1 \leftarrow R1 + M; \) // Move pointer to next \( x[] \)
- goto L; // End of inner loop

**Inner Loop**

- For \( M=8 \): \( EMAT(data) = ? \)

- **EMAT (instr) = ?**

---

Temporal Locality

**Consider the following program:**

```plaintext
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]; }
```

- High temporal locality
  - "A recently referenced memory location will be referenced again in the near future"
  - Equivalently: The distance between references in the memory reference string is small

**Examples**

- Small inner loops of instructions that fit into cache memory
- Small enough \( N \)
- The variables \( x[0], x[100], x[200], x[300] \) are repeatedly accessed in the second loop; i.e., closely accessed in time
Memory System Complexities

- “Chunk” of cache (cache line)
  - Size may be different than size of main memory “chunk”; e.g.,
    - Memory accessed in 8-byte chunks
    - But cache line is 32 bytes → Need 4 memory accesses
  - Multiple memory accesses to fill 1 cache line
- Main memory access latency
  - Must wait multiple memory cycles to access first chunk
  - May be able to get subsequent chunk in one memory cycle
  - e.g., 5-1-1-1 memory (8 memory cycles to get 32 bytes)
- Cache organization effects performance

Virtual Memory

- Map memory addresses at run-time
  - Physical (actual) address = f(logical address)
  - Typically, f( ) is implemented as a page table
- Basic Ideas
  - Hide details of real physical memory from user
  - Each user has n contiguous (linear) address spaces
    - Each begins at address 0
    - Paging (n = 1) versus Segmentation (n ≥ 1)

Virtual Memory (Paging)

- Virtual Memory and Addresses
- Physical Memory and Addresses
- Program Paging Hardware Memory
- Valid Bit (In memory?)
- Modified Bit (Dirty?)
- Page Table (PT)
- Address Cache
- PT Address
- Address
- Offset
- Frame
- Offset
- Physical Address
- Virtual Address

Simple Interrupts (1)

- Interrupt (Trap, Exception)
  - A vectored transfer of control to the supervisor
  - Through a trap table
    - One entry for each trap type
    - Branch Table: 256 addresses stored in first 1KB of memory
- Examples
  - User requests OS service
    - system call via software trap
  - I/O device request completion
  - Arithmetic overflow or underflow
  - Page fault (virtual address not in main memory)
  - Memory-protection violation (segmentation fault)
  - Undefined instruction
**Simple Interrupts (2)**

- **Interrupt Processing**
  - Disable other interrupts while processing interrupt
  - Ignore new interrupt until after reenabling interrupts
  - Interrupts usually have an **interrupt priority**

- **Process A**
  - System Call
  - Interrupt
  - Initiate I/O

- **Process B**
  - System Call
  - Interrupt Handler

- **OS Kernel**
  - Interrupt
  - Scheduler Resumes A

- **Device**
  - Initiate I/O

---

**Life Of A System Call**

- **User Process**
  - `read(fd,buf,nb)`
  - File Descriptor, Buffer Address, Number of Bytes
  - `syscall(proc,arg,...)`

- **Top Half of Kernel**
  - (may block)
  - C library functions

- **Bottom Half of Kernel**
  - (never blocks)
  - Hardware Interrupt

---

**Blocking Read System Call**

- **User Program**
  1. Push nb onto stack
  2. Push buf onto stack
  3. Push fd onto stack
  4. Call read

- **C Library**
  5. Reg ← read op-code
  6. Trap to OS kernel

- **OS Kernel**
  7. Check syscall args
  8. Jump thru syscall table

- **Contexts/Modes**: User, Kernel, Instr, Heap, Stack

---

**Life Of A Device Read Request (1)**

- **CPU**
  - `read()` results in `syscall()` which traps into kernel
  - Machine code for `read()` code comes from a library
  - Enter **kernel mode** from **user mode** (Perform **mode switch**)
  - Allows privileged instructions and access to all of memory
  - Queue request if I/O device is not ready
  - Send read request to controller of I/O device
  - Put process on queue while waiting for I/O to finish
  - Perform **context switch**

- **Device Controller**
  - Save process state and switch control (give CPU) to another process

- **Initiate operation on I/O device**
Life Of A Device Read Request (2)

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

- **Device Controller**
  » Request use of bus
  » Transfer data to main memory after bus grant
    * Controller "shares" memory bus with CPU and other devices
  » **Interrupt CPU** when read request has finished

- **CPU**
  » Interrupt transfers CPU control to the interrupt service routine
  » Save state of interrupted process
  » Disable lower priority 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**
  » Perform I/O
  » Change virtual memory protection bits
  » Disable/enable interrupt

- **Two (2) Modes of Operation**
  » **User Mode**: Execute instructions in user program
  » **Supervisor (Kernel) Mode**
    * Can execute privileged instructions
    * Can access VM pages in operating system kernel
  » **Mode bit** in **status register** indicates execution mode
  » Causes of mode switching
    * Interrupt, System call (software trap)

  **Execute privileged instr. only in kernel mode!!**

Reading

- Tanenbaum, Section 2.1
- Tanenbaum, Sections 10.1-10.3
  » Focus: fork(), waitpid(), execv() and other exec system calls

Reflection

- **Important Concepts?**
  » What?
  » Why?

- **Difficulties?**
  » What?
  » Why?