Matrix Multiply Example

**Pseudo-Code:**
```pseudocode
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 Pseudo-Code:**
```machine
L: if (inner loop done) Exit inner loop;
R4 <-- *R1; // Load register 3 from memory
R5 <-- *R2; // Load register 4 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[i][k]
R2 = R2+...; // Move pointer to next B[k][j]
goto L; // End of inner loop
```

**Basic Instruction Execution**

Start

```
<table>
<thead>
<tr>
<th>Fetch Next Instruction</th>
<th>Register Inst.</th>
<th>Fetch Data</th>
<th>Execute</th>
<th>Memory Inst.</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Pipelining Instruction Execution**

A Gantt Chart is a space-time diagram

<table>
<thead>
<tr>
<th>CPU Clock</th>
<th>Fetch I</th>
<th>Fetch D</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tick 1</td>
<td>I1</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
</tr>
<tr>
<td></td>
<td>I3</td>
<td>I2</td>
<td>I2</td>
</tr>
</tbody>
</table>

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

**CPU can "stall" for various reasons (memory wait)**
Hardware Context

Processor Status Word
- Contains address of next 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

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.

Stack Pointer (SP)
- Address of the top stack element

General Registers (R[0..R[7])
- Operands

Clock Speeds

- 500 MHz Pentium
- 250 MHz SRAM
- 100 MHz, 32-Bit Processor Bus
- Video
- Main Memory 100 MHz SDRAM
- PCI Bus (32 or 64 bits), 33 MHz

Cache Memory

- L1 cache typically split between instructions and data
- Arithmetic operations are only done in GRs
  - Must read X from main memory into cache(s) and then into a GR

- Need for asynchrony between components
  - Buffers smooth out traffic
  - May still get "stalls"

General Registers

ALU

Inst Data

L1 Cache

L2 Cache

Main Memory
Cache Memory Operation

- **Cache Memory**
  » Copy data to storage that is closer (and faster) to its place of use.
  » Decreases access time to cached data on subsequent 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.

---

Temporal Locality

- **Consider the following program:**
  ```
  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 memory location that has been recently referenced will be referenced again in the near future.
  » Equivalently: Small distance between references in memory reference string.

- **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.

---

EMAT

- **EMAT: Effective Memory Access Time**
  
  ```
  EMAT = [ H x T1 + M x ( T1 + T2 ) ] / ( H + M )
  ```

  » 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.

- **Examples**
  ```
  EMAT = h x T1 + (1-h) x ( T1 + T2 )
  ```

  » h: Hit ratio (H / (H + M)).

  » This form is easier to use.

- **Examples**
  ```
  EMAT = T1 + m x T2 where m = (1-h)
  ```

  » m: Miss ratio.

  » This form indicates that you must always pay the cost of a cache access.

---

Memory System Complexities

- **“Chunk” of cache (cache line)**
  » Size may be different than size of main memory “chunk”
  » e.g., 32 bytes versus 8 bytes.
  » Requires 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.

- **Cache organization effects performance**
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 are an attempt to bridge CPU-memory gap

![Technology Trends](image)

Storage Hierarchy Properties

<table>
<thead>
<tr>
<th>* 1 KB = 2¹⁰ = 1024 Bytes, 1 MB = 2²⁰, 1 GB = 2³⁰</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>

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 = 10⁶ cycles per sec, 1 GHz = 10⁹

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](image)
Virtual Memory (Paging)

Virtual Address

<table>
<thead>
<tr>
<th>page#</th>
<th>offset</th>
</tr>
</thead>
</table>

Physical Address

<table>
<thead>
<tr>
<th>frame#</th>
<th>offset</th>
</tr>
</thead>
</table>

Valid Bit

(In memory?)

Modified Bit

(Dirty?)

Program

Paging Hardware

Memory

Simple Interrupts (1)

- **Interrupt (Trap, Exception)**
  - A vectored transfer of control to the supervisor
  - Through a trap table
    - One entry for each trap type
    - PC: A branch table with 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)

- **Disable** all interrupts while processing an interrupt
  - Ignore new interrupt until after reenabling interrupts
    - i.e., New interrupt is pending
  - Interrupts usually have an interrupt priority

Life Of A System Call

User Process

```
read (fd, buf, nb)  
systemcall (proc, arg, ...)  
```

File Descriptor, Buffer Address, Number of Bytes

C library functions

Process State

CPU State

Hardware Interrupt

I/O controller buffer

I/O Buffer

I/O Wait Queue

Top Half of Kernel (may block)

Bottom Half of Kernel (never blocks)

Interrupt Handler

I/O Buffer

Interrupt Handler

Schedulier Resumes A

Initiate I/O

System Call

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

- **CPU**
  - read() results in `syscall()` which traps into the 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 complete
  - Perform context switch
    - Save process state and switch control (give CPU) to another process
- **Device Controller**
  - 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 usage 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

**Blocking Read System Call**

1. Push nbytes
2. Push buf
3. Push fd
4. Call read
5. Reg ← read op-code
6. Trap to OS kernel
7. Check syscall args
8. Jump thru syscall table
9. Exec syscall handler
10. Scheduler
11. Interrupt handler
12. Scheduler

**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 the execution mode
  - Causes of mode switching
    - Interrupt, System call (software trap)
- **Privileged instructions execute only in kernel mode**