11_cpuarch.md

Charudatta Korde

Memory Hierarchy

The memory hierarchy in computer architecture is designed to provide a balance between speed and size, optimizing cost and performance. It consists of different types of memory, each with distinct characteristics.

Types of Memory

1. **Cache Memory**:
   - **Speed**: Extremely fast
   - **Size**: Small (kilobytes to megabytes)
   - **Purpose**: Stores frequently accessed data to speed up processing.
2. **Main Memory (RAM)**:
   - **Speed**: Fast
   - **Size**: Moderate (gigabytes)
   - **Purpose**: Holds data and programs currently in use.
3. **Secondary Storage**:
   - **Speed**: Slower
   - **Size**: Large (gigabytes to terabytes)
   - **Purpose**: Stores data and programs not currently in use (e.g., HDD, SSD).

Speed and Size Comparison

Memory Type	Speed	Size
Cache Memory	Extremely Fast	Small (KB to MB)
Main Memory	Fast	Moderate (GB)
Secondary Storage	Slower	Large (GB to TB)

Cache Memory

Cache memory plays a crucial role in improving the performance of the CPU by storing frequently accessed data and instructions.

Importance and Function

- **Importance**: Reduces the time it takes to access data from the main memory.
- **Function**: Acts as a buffer between the CPU and main memory, storing copies of frequently accessed data to reduce latency.

Levels of Cache (L1, L2, L3)

1. **L1 Cache**:
   - Closest to the CPU core
   - Smallest size, but fastest
   - Separate instruction and data caches (Harvard architecture)
2. **L2 Cache**:
   - Larger than L1
   - Slower than L1, but still fast
   - Shared between cores in some CPUs
3. **L3 Cache**:
   - Largest cache level
   - Slowest of the three, but still faster than main memory
   - Typically shared across all cores in a multi-core processor

Virtual Memory

Virtual memory allows a computer to compensate for physical memory shortages by temporarily transferring data from RAM to disk storage.

Concept and Benefits

- **Concept**: Creates an illusion of a large, continuous memory space by using both physical memory and disk space.
- **Benefits**: Increases the apparent amount of RAM, isolates processes, and provides memory protection.

Implementation (Paging, Segmentation)

1. **Paging**: Divides memory into fixed-size pages, managing them with a page table.
2. **Segmentation**: Divides memory into variable-sized segments, each representing a logical unit such as a function or data structure.

Pipelining and Its Stages

Pipelining is a technique used in CPU design to increase instruction throughput by overlapping the execution of multiple instructions.

Stages of Pipelining

1. **Fetch**: Retrieve the instruction from memory.
2. **Decode**: Interpret the instruction and prepare the necessary data.
3. **Execute**: Perform the operation specified by the instruction.
4. **Memory Access**: Read/write data from/to memory if needed.
5. **Write Back**: Store the result back into a register.

Instruction Level Parallelism (ILP)

ILP is the parallel execution of multiple instructions to improve CPU performance.

Concepts of ILP

- **Parallelism**: Executing multiple instructions simultaneously to increase performance.
- **Dependencies**: Handling data and control dependencies between instructions.

Techniques to Achieve ILP

1. **Pipelining**: Overlapping instruction execution stages.
2. **Superscalar Execution**: Issuing multiple instructions per cycle.
3. **Out-of-Order Execution**: Reordering instructions to avoid stalls and improve efficiency.

Advanced CPU Architectures

Superscalar Architecture

- **Description**: Executes more than one instruction per clock cycle by employing multiple execution units.
- **Benefits**: Increases instruction throughput and overall performance.

VLIW (Very Long Instruction Word) Architecture

- **Description**: Uses long instruction words that encode multiple operations to be executed in parallel.
- **Benefits**: Simplifies hardware by relying on the compiler to identify parallelism, but requires sophisticated compiler support.

Performance Metrics

Measuring CPU Performance

1. **CPI (Cycles Per Instruction)**: Average number of cycles needed to execute an instruction.

2. **MIPS (Million Instructions Per Second)**: Number of millions of instructions executed per second.
3. **FLOPS (Floating Point Operations Per Second)**: Number of floating-point operations executed per second.

Factors Affecting Performance

- **Clock Speed**: Higher clock speeds generally mean faster execution.
- **Instruction Set Efficiency**: Optimized instructions can execute more efficiently.
- **Parallelism**: Increased parallelism through techniques like ILP and superscalar execution improves performance.
- **Cache Performance**: Efficient caching reduces memory access time and improves speed.

Euclidean Algorithm for GCD

Assembly Language (x86) Here's a simple assembly language program to find the GCD of two numbers using the Euclidean algorithm:

```
section .data
    num1 dw 56      ; First number
    num2 dw 98      ; Second number

section .bss
    result resw 1   ; Result

section .text
    global _start

_start:
    mov ax, [num1]  ; Load num1 into AX
    mov bx, [num2]  ; Load num2 into BX

gcd_loop:
    cmp bx, 0       ; Compare BX with 0
    je done         ; If BX is 0, jump to done
    xor dx, dx      ; Clear DX for division
    div bx          ; Divide AX by BX, quotient in AX, remainder in DX
    mov ax, bx      ; Move BX into AX
    mov bx, dx      ; Move remainder into BX
    jmp gcd_loop    ; Repeat the loop

done:
    mov [result], ax ; Store result in memory

    ; Exit the program
```

```
    mov eax, 1      ; System call number (sys_exit)
    xor ebx, ebx    ; Exit code 0
    int 0x80        ; Call kernel
```

Explanation of Assembly Code

1. **Data Section**: Initializes two numbers (`num1` and `num2`).
2. **BSS Section**: Declares a variable (`result`) to store the result.
3. **Text Section**: Contains the code to execute the Euclidean algorithm.
    - **Loading Values**: `mov ax, [num1]` and `mov bx, [num2]` load the numbers into registers `AX` and `BX`.
    - **GCD Loop**: Repeats the division and remainder calculation until `BX` becomes 0.
    - **Store Result**: Stores the result in memory.

C Language

Here's the same algorithm implemented in C:

```c
#include <stdio.h>

int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    int num1 = 56;
    int num2 = 98;
    int result = gcd(num1, num2);

    printf("GCD of %d and %d is %d\n", num1, num2, result);
    return 0;
}
```

Explanation of C Code

1. **gcd Function**: Implements the Euclidean algorithm.
    - **While Loop**: Repeats the calculation until `b` becomes 0.
    - **Modulus Operation**: `a % b` calculates the remainder.
    - **Return Statement**: Returns the greatest common divisor.
2. **Main Function**:
    - **Initialize Numbers**: `num1` and `num2` are initialized.

- **Call gcd Function**: Calls the `gcd` function and stores the result.
- **Print Result**: Prints the result to the console.

Comparison

- **Low-Level vs. High-Level**: Assembly language provides low-level control over the hardware, allowing precise manipulation of registers and memory. In contrast, C is a high-level language that abstracts these details, making it easier to read and write.
- **Instruction Set**: Assembly language requires explicit instructions for each operation, while C uses high-level constructs like functions and loops.
- **Portability**: Assembly language is specific to a particular CPU architecture (x86 in this case), while C is more portable across different platforms with minimal changes.
- **Readability**: C code is generally more readable and maintainable, making it easier for developers to understand and modify.

Example C Code

Here's a simple C function that adds two integers:

```c
int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5;
    int y = 10;
    int result = add(x, y);
    return 0;
}
```

Corresponding Assembly Code

Let's see how this C code might be translated into x86 assembly language. The exact assembly code can vary depending on the compiler and optimization settings, but here's a typical translation:

```
section .data
    x dd 5          ; Declare variable x with value 5
    y dd 10         ; Declare variable y with value 10
    result dd 0     ; Declare variable result

section .text
    global _start

_start:
    ; Call add(x, y)
```

```
    mov eax, [x]    ; Move value of x into EAX register
    mov ebx, [y]    ; Move value of y into EBX register
    call add        ; Call add function
    mov [result], eax ; Move return value into result

    ; Exit program
    mov eax, 1      ; System call number (sys_exit)
    xor ebx, ebx    ; Exit code 0
    int 0x80        ; Call kernel

add:
    push ebp        ; Save base pointer
    mov ebp, esp    ; Set base pointer to current stack pointer
    mov eax, [ebp+8] ; Move first argument (a) into EAX
    mov ebx, [ebp+12] ; Move second argument (b) into EBX
    add eax, ebx    ; Add EBX to EAX (a + b)
    pop ebp         ; Restore base pointer
    ret             ; Return from function
```

Translation Process

1. **Data Section**:
   - The data section declares variables `x`, `y`, and `result`.
2. **Text Section**:
   - The text section contains the executable code.
   - The `global _start` directive marks the entry point of the program.
3. **_start Label**:
   - The main function is represented by the `_start` label.
   - The values of `x` and `y` are loaded into registers `EAX` and `EBX`, respectively.
   - The `call add` instruction calls the `add` function.
4. **add Function**:
   - The `add` function is defined in assembly.
   - The `push ebp` and `mov ebp, esp` instructions set up the stack frame.
   - The arguments `a` and `b` are accessed from the stack using the base pointer (`[ebp+8]` and `[ebp+12]`).
   - The `add eax, ebx` instruction adds the values.
   - The `pop ebp` and `ret` instructions clean up the stack and return to the caller.

Key Points

- **Registers**: General-purpose registers (`EAX`, `EBX`) are used to hold values and perform operations.
- **Stack Frame**: The stack is used to pass arguments to functions and manage local variables.

- **Function Calls**: The `call` and `ret` instructions manage function calls and returns.
- **Instruction Mapping**: Each C statement typically maps to one or more assembly instructions.

Summary

- **C Code**: High-level, human-readable code that defines the logic of the program.
- **Assembly Code**: Low-level, machine-readable code that directly controls the CPU.
- **Translation Process**: The compiler translates C code into assembly code, optimizing it for the target architecture.

GCC Compiler

GNU Compiler Collection (GCC) is a versatile compiler that supports multiple programming languages, including C, C++, Objective-C, Fortran, Ada, Go, and Rust. It is widely used for compiling code for various hardware architectures and operating systems.

Key Features - I

- **Language Support**: Supports a wide range of programming languages.
- **Cross-Compilation**: Can compile code for different target platforms.
- **Optimization**: Includes various optimization options to improve performance.
- **Open Source**: Distributed under the GNU General Public License (GPL).
- **Extensibility**: Can be extended with plugins and additional front ends.

NASM Assembler

Netwide Assembler (NASM) is an assembler for the Intel x86 architecture, capable of producing 16-bit, 32-bit, and 64-bit programs. It is known for its simplicity and ease of use.

Key Features - II

- **Architecture Support**: Supports x86 and x86-64 architectures.
- **Output Formats**: Can generate various binary formats, including ELF, COFF, Mach-O, and binary files.
- **Syntax**: Uses a variant of Intel assembly syntax, which is simpler than AT&T syntax.
- **Open Source**: Distributed under the simplified BSD license.
- **Portability**: Can run on Unix-like systems, Windows, and DOS.

Comparison - II

Feature	GCC Compiler	NASM Assembler
Language Support	Multiple languages (C, C++, etc.)	Assembly language
Architecture	Multiple architectures	x86 and x86-64
Output Formats	Various formats (ELF, COFF, etc.)	Various formats (ELF, COFF, etc.)
License	GPL	BSD
Optimization	Yes	No
Cross-Compilation	Yes	No
Syntax	Depends on language	Intel assembly syntax

Summary

- **GCC** is a comprehensive compiler for high-level languages, offering extensive optimization and cross-compilation capabilities.
- **NASM** is a specialized assembler for x86 architectures, known for its simplicity and ease of use.