# Computer Architecture Study Guide

\*\*1. Introduction to Computer Architecture and Assembly Language\*\*
Computer architecture defines the instruction set, data types, registers, memory management, and I/O model of a computer system. It dictates how software interacts with hardware. Assembly language is a human-readable, low-level programming language that is machine-dependent, meaning it's specific to a particular CPU architecture like MIPS, ARM, or x86. It's crucial for accessing bare-metal hardware, optimizing performance, and is commonly used in device drivers and embedded systems. Machine code, composed of 0s and 1s, is the lowest-level format that the computer directly executes.

\*\*2. Data Representation\*\*
Numbers can be represented in various formats: binary (base 2), decimal (base 10), and hexadecimal (base 16). Conversion rules exist to switch between these bases. For example, to convert decimal to binary, repeatedly divide by 2 and record the remainders. For signed numbers, two primary methods are used: Signed Magnitude and Two's Complement. In Signed Magnitude, the Most Significant Bit (MSB) indicates the sign (0 for positive, 1 for negative), while the remaining bits represent the magnitude. This method has issues with addition and two representations for zero (±0).

Two's Complement is the preferred method for representing signed numbers in computers. In this system, the MSB carries a negative weight (e.g., in an N-bit number, the MSB represents -2^(N-1)). This format allows for consistent arithmetic operations and has only one representation for zero. To convert a positive number to its negative Two's Complement equivalent (or vice versa), you invert all the bits (0s become 1s, 1s become 0s) and then add 1 to the result, ignoring any overflow. Sign extension is an important concept where the sign bit is copied to higher-order bits when expanding a number to a larger bit representation, preserving its value.

\*\*3. Arithmetic Operations\*\*
MIPS architecture follows design principles like "simplicity favors regularity" and "smaller is faster." Arithmetic operations typically involve three operands: two source registers and one destination register (e.g., `add $dst, $src0, $src1`). Common instructions include `add` (addition), `sub` (subtraction), and `addi` (add immediate, for adding a constant). Unsigned addition (`addu`) and subtraction (`subu`) are also available and do not cause an overflow exception. Overflow occurs when the result of an arithmetic operation exceeds the maximum representable value for the given number of bits, leading to an incorrect sign or magnitude.

\*\*4. Memory Operations\*\*
Computers use both registers and main memory to store data. Registers are a small, fast set of storage locations (MIPS has 32, each 32-bit "word") used for frequently accessed data. Main memory is much larger and slower, used for composite data like arrays and structures. MIPS CPUs can only perform arithmetic or logical operations on data in registers. Therefore, data from memory must first be loaded into registers using `lw` (load word) or `lbu` (load byte unsigned), and results are stored back to memory using `sw` (store word) or `sb` (store byte). Memory can be word-addressable (address increments by word size, typically 4 bytes for 32-bit words) or byte-addressable (each byte has a unique address). MIPS is byte-addressed, meaning word addresses increment by 4. Memory alignment is critical; word operations (`lw`, `sw`) require addresses to be multiples of 4 to avoid exceptions. Big-endian and Little-endian refer to how bytes within a word are ordered in memory.

\*\*5. Logical Operations and Shifters\*\*
Logical operations act on binary numbers according to Boolean logic, including AND, OR, NOT, and XOR. They are essential for bit manipulation, control flow, and hardware interaction. For example, the `and` operation can "mask" bits, selecting some and clearing others to 0. Shifters manipulate data at the bit level, optimizing mathematical operations and memory address calculations. There are two main types: Logical and Arithmetic. Logical shifters (`sll` for shift left logical, `srl` for shift right logical) fill empty spaces with 0s. Arithmetic shifters (`sra` for shift right arithmetic) preserve the sign of the number by filling empty spaces with the old most significant bit (MSB) during right shifts. Left shifts (`sll`) are equivalent to multiplying by powers of 2 (A << N = A \* 2^N), and right arithmetic shifts (`sra`) are equivalent to dividing by powers of 2 (A >>> N = A / 2^N).

\*\*6. Machine Coding (MIPS Instruction Formats)\*\*
MIPS instructions are encoded in binary as 32-bit "machine code" words. The operation code (opcode) specifies the instruction, and operands (like registers) are also binary numbers. MIPS uses three instruction formats: R-type, I-type, and J-type, each 32 bits wide. R-type instructions are for operations with only register operands (e.g., `add`, `sub`, `and`, `or`, `srl`, `sll`). Their fields include opcode, `rs` (first source register), `rt` (second source register), `rd` (destination register), `shamt` (shift amount), and `funct` (function code). I-type instructions include an immediate operand, and J-type instructions are used for jumping. Understanding these formats is crucial for converting assembly language into machine code.

\*\*7. Floating-Point Arithmetic\*\*
Floating-point numbers provide greater precision and range compared to fixed-point numbers. The IEEE 754 standard defines representations for single-precision (32-bit, "float") and double-precision (64-bit, "double") floating-point numbers. A floating-point number is generally represented as ±M × B^E, where M is the mantissa, B is the base (2 for binary), and E is the exponent. The IEEE 754 standard uses an implicit leading 1 for the mantissa (saving a bit of precision) and a biased exponent to represent both positive and negative exponents. Floating-point addition is more complex than integer addition, requiring exponents to be aligned before mantissas can be added, and then the result normalized.

\*\*8. SYSCALL Functions\*\*
SYSCALL functions provide an interface for assembly programs to interact with the operating system and the outside world. These are system calls that perform tasks like printing integers, strings, or characters to the screen, reading input from the keyboard, allocating dynamic memory, and file I/O. In MIPS, specific registers are used for SYSCALL functions: `$v0` holds the service number (indicating the desired action), `$a0-$a2` or `$f12` hold arguments for the action, and `$v0-$v1` or `$f0` return results from the action. Programs typically load the service number into `$v0`, load any arguments, issue the `syscall` instruction, and then retrieve results.