Addressing Modes x121321313466464645dddd232132424543564656477

Gjjhgj

# The Term *Addressing Modes* Refers To The Way In Which The Operand Of An Instruction Is Specified. Information Contained In The Instruction Code Is The Value Of The Operand Or The Address Of The Result/Operand. Following Are The Main Addressing Modes That Are Used On Various Platforms And Architectures.

**1) Immediate Mode**

The operand is an immediate value is stored explicitly in the instruction:

**2) Index Mode**

The address of the operand is obtained by adding to the contents of the general register (called index register) a constant value. The number of the index register and the constant value are included in the instruction code. Index Mode is used to access an array whose elements are in successive memory locations. The content of the instruction code, represents the starting address of the array and the value of the index register, and the index value of the current element. By incrementing or decrementing index register different element of the array can be accessed.

**3) Indirect Mode**

The effective address of the operand is the contents of a register or main memory location, location whose address appears in the instruction. Indirection is noted by placing the name of the register or the memory address given in the instruction in parentheses. The register or memory location that contains the address of the operand is a pointer. When an execution takes place in such mode, instruction may be told to go to a specific address. Once it's there, instead of finding an operand, it finds an address where the operand is located.

**4) Absolute (Direct) Mode**

The address of the operand is embedded in the instruction code.

**5) Register Mode**

The name (the number) of the CPU register is embedded in the instruction. The register contains the value of the operand. The number of bits used to specify the register depends on the total number of registers from the processor set.

**) Displacement Mode**

Similar to index mode, except instead of a index register a base register will be used. Base register contains a pointer to a memory location. An integer (constant) is also referred to as a displacement. The address of the operand is obtained by adding the contents of the base register plus the constant. The difference between index mode and displacement mode is in the number of bits used to represent the constant. When the constant is represented a number of bits to access the memory, then we have index mode. Index mode is more appropriate for array accessing; displacement mode is more appropriate for structure (records) accessing.

**7) Autoincrement /Autodecrement Mode**

A special case of indirect register mode. The register whose number is included in the instruction code, contains the address of the operand. Autoincrement Mode = after operand addressing , the contents of the register is incremented. Decrement Mode = before operand addressing, the contents of the register is decrement.

**Multiplying Two Numbers in Memory**
On the right is a diagram representing the storage scheme for a generic computer. The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.

**The CISC Approach**
The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

MULT 2:3, 5:2

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher debdutta language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a \* b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-debdutta language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

instructions directly into the hardware.

instructions directly into the hardware.

A

D

E

F

n

Y

Z

**The RISC Approach**
RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly debdutta instructions. The compiler must also perform more work to convert a high-debdutta language statement into code of this form.

|  |  |
| --- | --- |
| **CISC**  | **RISC**  |
| Emphasis on hardware  | Emphasis on software  |
| Includes multi-clockcomplex instructions  | Single-clock,reduced instruction only  |
| Memory-to-memory:"LOAD" and "STORE"incorporated in instructions  | Register to register:"LOAD" and "STORE"are independent instructions  |
| Small code sizes,high cycles per second  | Low cycles per second,large code sizes  |
| Transistors used for storingcomplex instructions  | Spends more transistorson memory registers  |

However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

**The Performance Equation**
The following equation is commonly used for expressing a computer's performance ability:



The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

**RISC Roadblocks**
Despite the advantages of RISC based processing, RISC chips took over a decade to gain a foothold in the commercial world. This was largely due to a lack of software support.

Although Apple's Power Macintosh line featured RISC-based chips and Windows NT was RISC compatible, Windows 3.1 and Windows 95 were designed with CISC processors in mind. Many companies were unwilling to take a chance with the emerging RISC technology. Without commercial interest, processor developers were unable to manufacture RISC chips in large enough volumes to make their price competitive.

Another major setback was the presence of Intel. Although their CISC chips were becoming increasingly unwieldy and difficult to develop, Intel had the resources to plow through development and produce powerful processors. Although RISC chips might surpass Intel's efforts in specific areas, the differences were not great enough to persuade buyers to change technologies.

**The Overall RISC Advantage**
Today, the Intel x86 is arguable the only chip which retains CISC architecture. This is primarily due to advancements in other areas of computer technology. The price of RAM has decreased dramatically. In 1977, 1MB of DRAM cost about $5,000. By 1994, the same amount of memory cost only $6 (when adjusted for inflation). Compiler technology has also become more sophisticated, so that the RISC use of RAM and emphasis on software has become ideal.

Instruction execution cycle

Once a program is in memory it has to be executed. To do this, each instruction must be looked at, decoded and acted upon in turn until the program is completed. This is achieved by the use of what is termed the 'instruction execution cycle', which is the cycle by which each instruction in turn is processed. However, to ensure that the execution proceeds smoothly, it is is also necessary to synchronise the activites of the processor.

To keep the events synchronised, the clock located within the CPU [control unit](http://www.eastaughs.fsnet.co.uk/cpu/structure-cu.htm) is used. This produces regular pulses on the system bus at a specific frequency, so that each pulse is an equal time following the last. This clock pulse frequency is linked to the clock speed of the processor - the higher the clock speed, the shorter the time between pulses. Actions only occur when a pulse is detected, so that commands can be kept in time with each other across the whole computer unit.

The instruction execution cycle can be clearly divided into three different parts, which will now be looked at in more detail. For more on each part of the cycle click the relevant heading, or use the next arrow as before to proceed though each stage in order.

[**Fetch Cycle**](http://www.eastaughs.fsnet.co.uk/cpu/execution-fetch.htm)
The fetch cycle takes the address required from memory, stores it in the [instruction register](http://www.eastaughs.fsnet.co.uk/cpu/structure-reg.htm), and moves the program counter on one so that it points to the next instruction.

[**Decode Cycle**](http://www.eastaughs.fsnet.co.uk/cpu/execution-decode.htm)
Here, the control unit checks the instruction that is now stored within the [instruction register](http://www.eastaughs.fsnet.co.uk/cpu/structure-reg.htm). It determines which opcode and addressing mode have been used, and as such what actions need to be carried out in order to execute the instruction in question.

[**Execute Cycle**](http://www.eastaughs.fsnet.co.uk/cpu/execution-decode.htm)
The actual actions which occur during the execute cycle of an instruction depend on both the instruction itself, and the addressing mode specified to be used to access the data that may be required. However, four main groups of actions do exist, which are discussed in full later on.

Clicking the next arrow below will take you to further information relating to the fetch cycle.