3 Assembly Languages

3.1 Assembly Language Instructions

Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'. It is the processor's 'instruction set'.

3.1.1 Advantages of Assembly Language

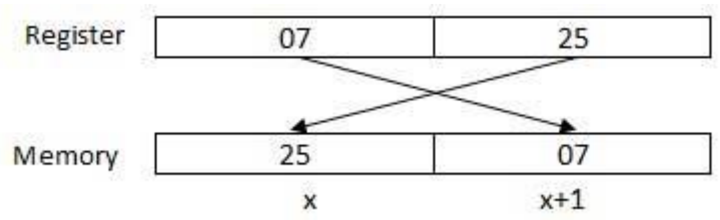
Some advantages are:

- It requires less memory and execution time;
- It is suitable for hardware-specific complex jobs;
- It is suitable for time-critical jobs;
- It is most suitable for writing interrupt service routines and other memory resident programs.

3.1.2 Addressing Data in Memory

The processor may access one or more bytes of memory at a time. Let us consider a hexadecimal number 0725H. This number will require two bytes of memory. The high-order byte or most significant byte is 07 and the low-order byte is 25.

The processor stores data in reverse-byte sequence, i.e., the low-order byte is stored in low memory address and high-order byte in high memory address. So, if processor brings the value 0725H from register to memory, it will transfer 25 first to the lower memory address and 07 to the next memory address.



A two-byte number in the processor and in RAM

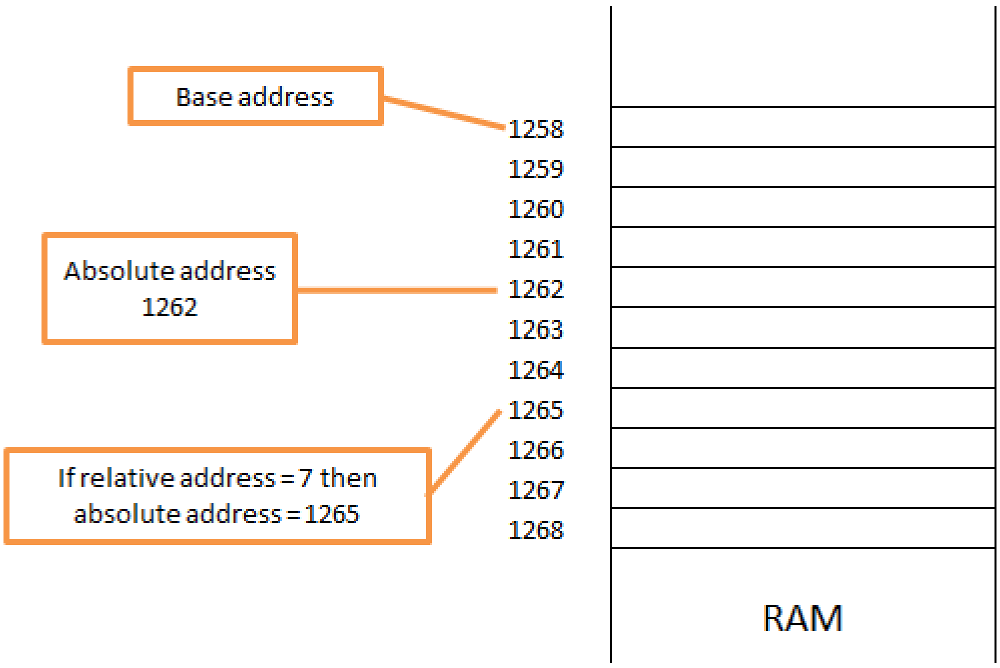
When the processor gets the numeric data from memory to register, it again reverses the bytes.

Absolute address

An absolute address is a direct reference of specific location.

Relative address

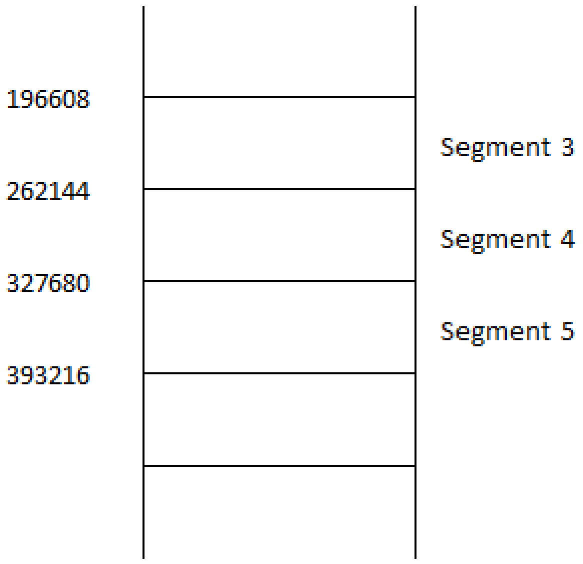
A relative address is a memory address that represents some distance from a starting point (base address), such as the first byte of a program or table. The absolute address is derived by adding it to the base address.



Absolute and relative addresses

Segment and offset values

Some memories are divided into segments (e.g. of 64KB each). So the physical address is obtained by combining a Segment address and an Offset address.

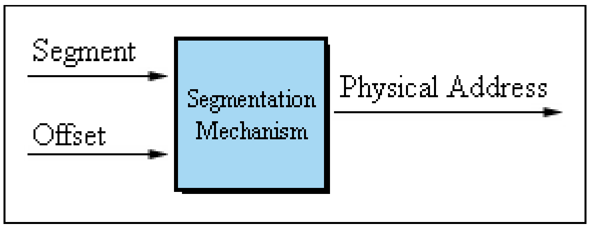


Segment 4: Offset 83

gives a physical address of 262227

(262144+83)

A simple example of a Segment address and an Offset address



The physical address is obtained by combining the Segment and Offset values

Instruction Format

An instruction consists of:

- operations
- operands

A few examples

Instruction	Operation	Operand/s	Meaning of Instruction
LDA #0011 0011	LDA	#0011 0011	Load the accumulator with 0011 0011
SHL	SHL		Logical shift contents of the accumulator to the left by 1
JZE rep	JZE	rep	Jump to 'rep' if last operation result was 0

Examples of assembly language instructions

The part of a machine instruction that tells the computer what to do is called Operation. Its code in binary or hex is called Opcode. Often the operation itself is referred to as opcode. The opcode is given a mnemonic name so that it would be easy to use, for example the operation 'load into the accumulator' is shortened to LDA and 'subtract' is shortened to SUB.

3.2 Instruction Groups

Data Transfer instructions

- MOV Moves byte or word to register or memory
- PUSH Push a word on stack
- POP Pop a word from stack

Logical Instructions

- NOT Logical not (1's complement)
- AND Logical and
- OR Logical or
- XOR Logical exclusive-or

Arithmetic Instruction

- ADD , ADC Add and Add with carry
- SUB, SBB Subtract and Subtract with borrow

- INC Increment
- DEC Decrement
- CMP Compare

Transfer Instructions

- JMP Unconditional Jump
- JE Jump on Equal
- JNE Jump on Not Equal
- JL Jump if Less
- JLE Jump if less or equal
- JG Jump if Greater
- JGE Jump if Greater or Equal
- JC, JNC Jump on carry or Jump on No Carry
- CALL Call Subroutine
- RET Return from subroutine

Flag Manipulation

- CLC Clear Carry
- STC Set Carry

Shift and Rotate

- SHL, SHR Logical Shift Left or Right
- RCL, RCR Rotate through Carry Left or Right

3.3 Registers

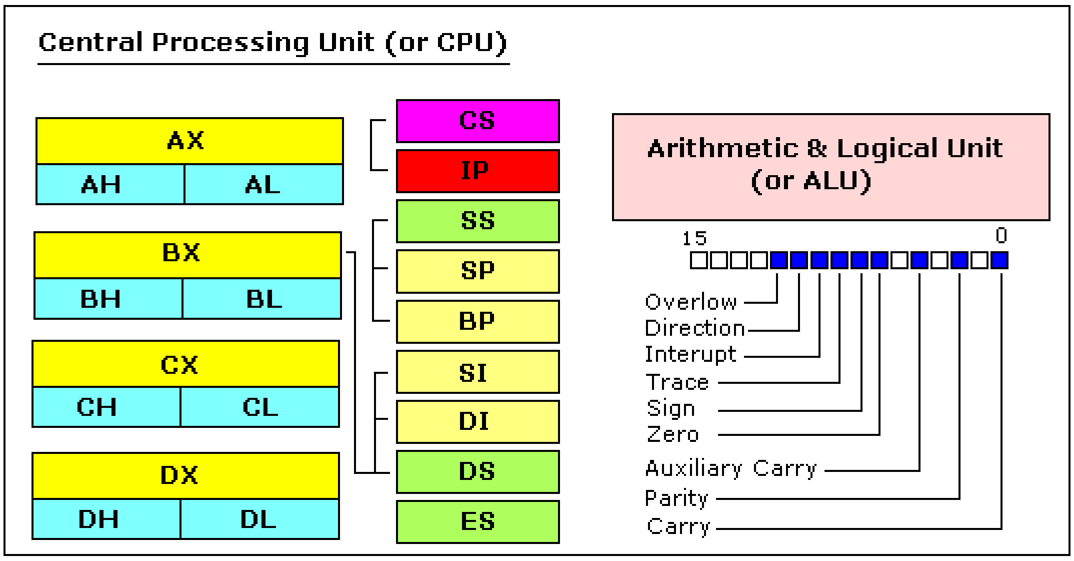
In computer architecture, a processor register is a small amount of storage available as part of a CPU or other digital processor.

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register".

In 16-bit mode (such as provided by the Pentium processor when operating as a Virtual 8086 - this is the mode used when Windows 95 displays a DOS prompt), the processor provides the programmer with 14 internal registers, each 16 bits wide. They are grouped into several categories as follows:

- Four general-purpose registers, AX, BX, CX, and DX. Each of these is a combination of two 8-bit registers which are separately accessible as AL, BL, CL, DL (the "low'' bytes) and AH, BH, CH, and DH (the "high'' bytes). For example, if AX contains the 16-bit number 1234h, then AL contains 34h and AH contains 12h.
- Four special-purpose registers, SP, BP, SI, and DI.
- Four segment registers, CS, DS, ES, and SS.
- The instruction pointer, IP (sometimes referred to as the program counter).
- The status flag register, FLAGS.

Although first four registers are referred to as "general-purpose", each of them is designed to play a particular role in common use:



CPU registers

- AX is the "accumulator"; some of the operations, such as MUL and DIV, require that one of the operands be in the accumulator. Some other operations, such as ADD and SUB, may be applied to any of the registers (that is, any of the eight general- and special-purpose registers) but are more efficient when working with the accumulator.
- BX is the "base" register; it is the only general-purpose register which may be used for indirect addressing. For example, the instruction MOV [BX], AX causes the contents of AX to be stored in the memory location whose address is given in BX.
- CX is the "count" register. The looping instructions (LOOP, LOOPE, and LOOPNE), the shift and rotate instructions (RCL, RCR, ROL, ROR, SHL, SHR, and SAR), and the string instructions (with the prefixes REP, REPE, and REPNE) all use the count register to determine how many times they will repeat.
- DX is the "data" register; it is used together with AX for the word-size MUL and DIV operations, and it can also hold the port number for the IN and OUT instructions, but it is mostly available as a convenient place to store data, as are all of the other general-purpose registers.

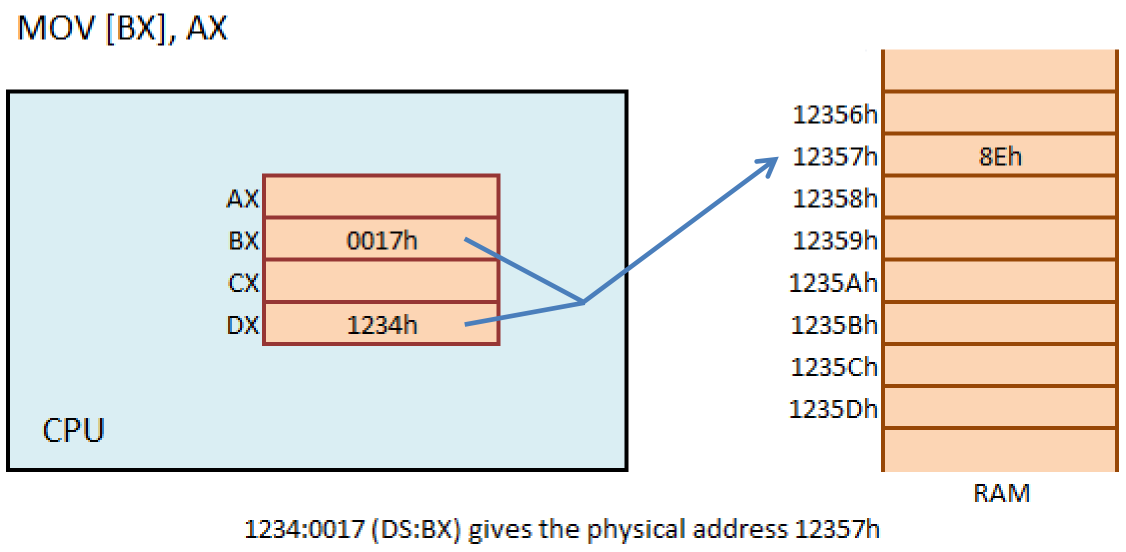
Here are brief descriptions of the four special-purpose registers:

- SP is the stack pointer, indicating the current position of the top of the stack. You should generally never modify this directly, since the subroutine and interrupt call-and-return mechanisms depend on the contents of the stack.
- BP is the base pointer, which can be used for indirect addressing similar to BX.
- SI is the source index, used as a pointer to the current character being read in a string instruction (LODS, MOVS, or CMPS). It is also available as an offset to add to BX or BP when doing indirect addressing; for example, the instruction MOV [BX+SI], AX copies the contents of AX into the memory location whose address is the sum of the contents of BX and SI.

- DI is the destination index, used as a pointer to the current character being written or compared in a string instruction (MOVS, STOS, CMPS, or SCAS). It is also available as an offset, just like SI.

Since all of these registers are 16 bits wide, they can only contain addresses for memory within a range of 64K (=2^16) bytes. To support machines with more than 64K of physical memory, Intel implemented the concept of segmented memory. At any given time, a 16-bit address will be interpreted as an offset within a 64K segment determined by one of the four segment registers (CS, DS, ES, and SS).

As an example, in the instruction MOV [BX], AX mentioned above, the BX register really provides the offset of a location in the current data segment; to find the true physical address into which the contents of the accumulator will be stored, you have to add the value in BX to the address of the start of the data segment. This segment start address is determined by taking the 16-bit number in DS and multiplying by 16. Therefore, if DS contains 1234h and BX contains 0017h, then the physical address will be 1234h * 16 + 0017h. This gives 12340h + 0017h = 12357h. (This computation illustrates one reason why hexadecimal is so useful; multiplication by 16 corresponds to shifting the hex digits left one place and appending a zero.) We refer to this combined address as 1234:0017 or, more generally, as DS:BX. All this is shown in the diagram below.



Calculating the physical address

Each segment register has its own special uses:

- CS points at the segment containing the current program. It permits several instances of a single program to run at once (in a multitasking operating system), all sharing the same code segment in memory; each instance has its own data and stack segments where the information specific to the instance is kept. Picture multiple windows, each running the same word processor on a different document; each one needs its own data segment to store its document, but they can all execute the same loaded copy of the word processor.
- DS generally points at segment where variables are defined.
- ES extra segment register, it's up to a coder to define its usage e.g. it can be used when data from two segments need to be accessed at once.
- SS points at the segment containing the stack.

The instruction pointer, IP, gives the address of the next instruction to be executed, relative to the code segment. The only way to modify this is with a branch instruction.

The status register, FLAGS, is a collection of 1-bit values which reflect the current state of the processor and the results of recent operations. Some flags of the 8086 are listed below:

- Carry (bit 0): set if the last arithmetic operation ended with a leftover carry bit.
- Parity (bit 2): set if the low-order byte of the last data operation contained an even number of 1 bits (that is, it signals an even parity condition).
- Auxiliary Carry (bit 4): used when working with binary coded decimal (BCD) numbers.
- Zero (bit 6): set if the last computation had a zero result. After a comparison (CMP, CMPS, or SCAS), this indicates that the values compared were equal (since their difference was zero).
- Sign (bit 7): set if the last computation had a negative result (a 1 in the leftmost bit).
- Trace (bit 8): when set, this puts the CPU into single-step mode, as used by debuggers.
- Interrupt (bit 9): when set, interrupts are enabled.
- Overflow (bit 11): set if the last arithmetic operation caused a signed overflow.

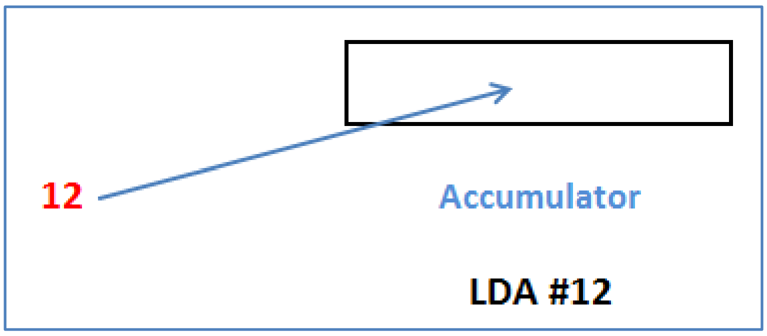
3.4 Addressing Modes

An addressing mode indicates how operators should be treated. Sometimes an operator has to be treated as a value. Sometimes it indicates an address. Sometimes it indicates an address which contents indicate an address where the final value is found.

Immediate addressing

Immediate addressing means that the data is found inside the instruction itself.

Example: ‘LDA #12’ means: load 12 into A (the accumulator).



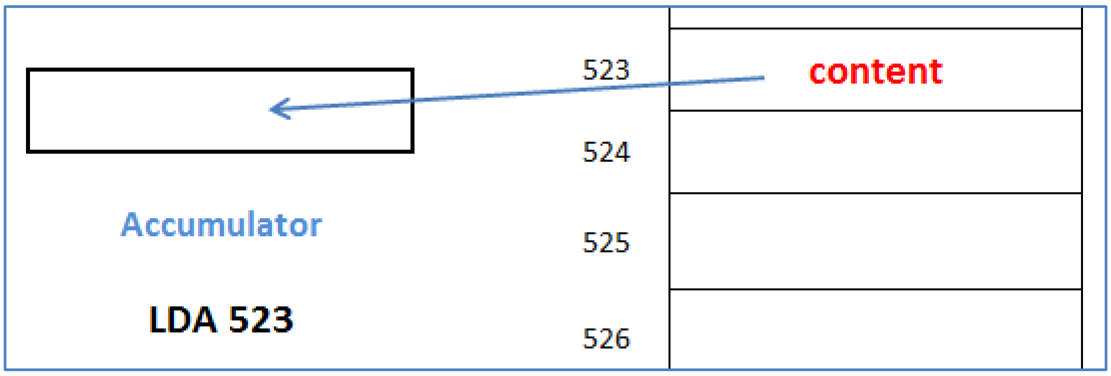
Immediate Addressing

Advantage: very fast.
Disadvantage: the value in the instruction never changes.

Direct addressing

Direct addressing means the code refers directly to a location in memory.

Example: 'LDA 523' means: load the contents of address 523 into A.



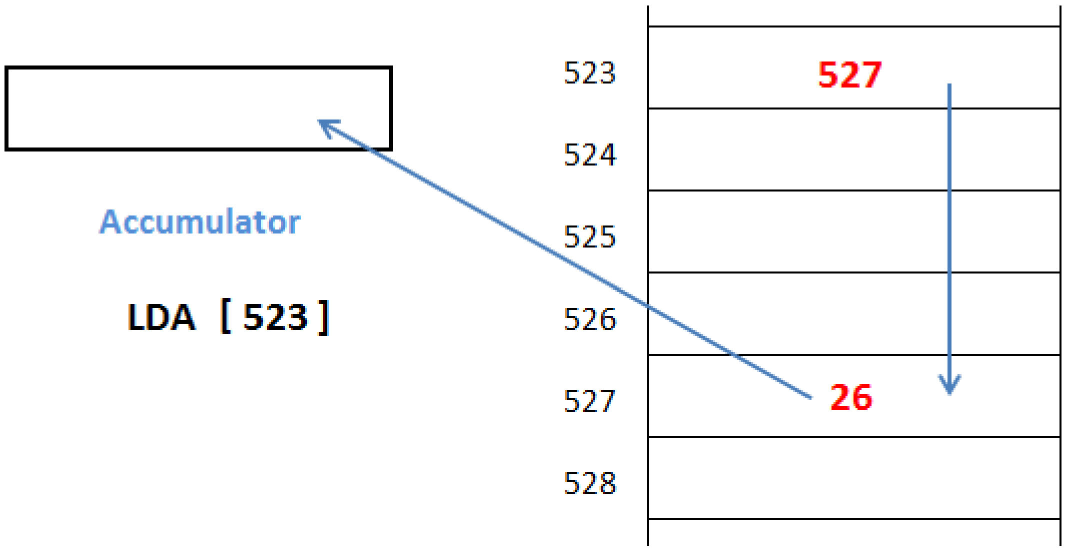
Direct Addressing

Advantage: fast (but not as fast as immediate addressing).
Disadvantage: the program cannot be relocated.

Re-locatable code refers to code that can be put anywhere in RAM.

Indirect addressing

In indirect addressing the operand in the instruction is an address and at this address is found another address which indicates the value required e.g. LDA [523]



Indirect addressing

Register addressing

Register addressing is a term used to indicate an instruction where the operands are registers for example: Add R4, R3 (which means get the value in register R3 and add it to the value in register R4 placing the value in R4.

Indexed addressing

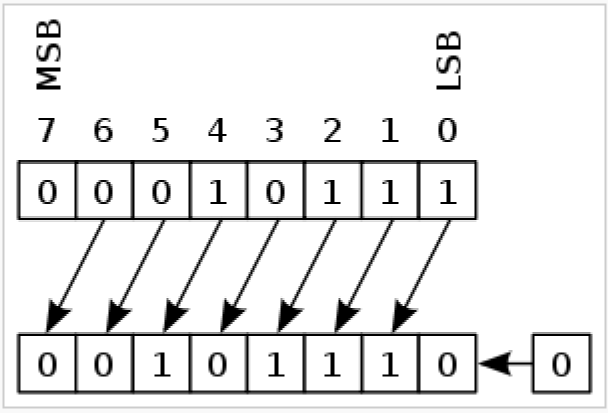
Indexed addressing means that the final address for the data is determined by adding an offset to a base address e.g. MOV CX, [BX + DI] (the value in the base index register BX is combined with the number in the destination index register DI to provide address of the number to be loaded into the CX register).

3.5 Examples of instructions

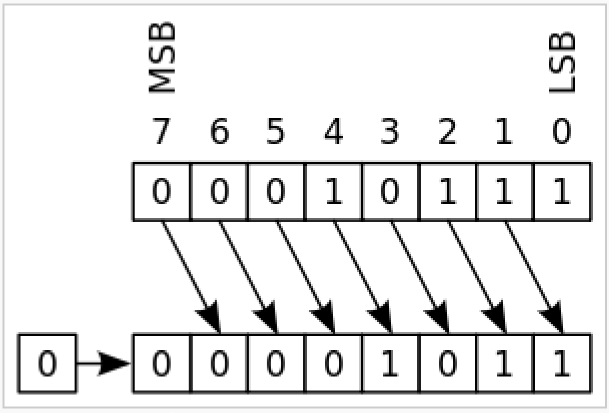
OPCODE	NOTES AND EXAMPLES
MOV	MOV is the mnemonic for Move MOV TOTAL, 48 ; Transfer the value 48 to the memory variable ; TOTAL MOV AL, 10 ; Transfer the value 10 to the AL register MOV CL, TABLE[2] ; Copies the 3rd element of array TABLE in CL MOV [EBX], 72 ; Copies 72 at the address indicated in ECB
ADD	ADD AH, BH ; Add the number in the BH register to the number ; in the AH register and store in the AH register ADD MARKS, 10 ; Add 10 to the variable MARKS ADD EBX, [ECX] ; Add the number found in the address indicated in ; ECX with the number in EBX and store in EBX
ADC	Same as ADD but it adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand.
SUB	SUB is the mnemonic for Subtract SUB EBX, EAX ; Subtract the number found in register EAX from ; the number found in register EBX and store the ; result in EBX SUB [ad], 4Bh ; Subtract the hex number 4B from the number ; found in address ad and place the result in ; address ad
SBB	SBB destination, source Subtract with borrow. Subtracts 'source' + 'carry flag' from 'destination', storing result in destination.
DEC	DEC is the mnemonic for Decrease by 1 DEC EAX ; subtract 1 from the number found in EAX
INC	INC is the mnemonic for Increase by 1 INC EAX ; add 1 to the number found in register EAX
CMP	COMP is the mnemonic for Compare CMP DX, 00 ; Compare the DX value with zero JE L7 ; If DX is equal to 0 then jump to label L7 CMP EDX, 10 ; Compares whether the counter has reached 10 JLE LP1 ; If it is less than or equal to 10, then jump to LP1
JMP	JMP is the mnemonic for Jump. It is an unconditional jump. All the jump statements that follow are all conditional jumps. JMP lab ; Jump to the label 'lab'
JG	JG is the mnemonic for Jump if Greater CMP ECX, [ad] ; Compare the number in ECX with the number in ; address ad JG here ; If in the previous operation the value of the first

	; operand (ECX) was greater than the value of the ; second operand ([ad]) then jump to the label 'here'.
JGE	JGE is the mnemonic for Jump if Greater or equal
JL	JL is the mnemonic for Jump if Less
JLE	JLE is the mnemonic for Jump if Less or Equal
JE	JE is the mnemonic for Jump if Equal
JNE	JNE is the mnemonic for Jump of Not Equal
JC	JC is the mnemonic for Jump on Carry
JNC	JNC is the mnemonic for Jump on No Carry
PUSH	PUSH AX ; Put the value found in register AX on top of the ; stack.
POP	POP CX ; Put the value found at the top of the stack in ; register CX. Remove the top value from the stack.
AND	For example, say the BL register contains 0011 1010. If you need to clear the high-order bits to zero, you AND it with 0FH. AND BL, 0FH ; This sets BL to 0000 1010
OR	For example, let us assume the AL register contains 0011 1010, you need to set the four low-order bits, you can OR it with a value 0000 1111. OR BL, 0FH ; This sets BL to 0011 1111
NOT	Suppose AX contains 0101 0011 NOT AX ; AX will now contain 1010 1100
XOR	Suppose AX contains 10110001 and BX contains 10011000 XOR AX, BX ; AX will now contain 00101001
CALL, RET	CALL stands for Subroutine Call RET stands for Return (from subroutine call) The CALL instruction first pushes the current code location onto the hardware supported stack in memory, and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes. The RET instruction implements a subroutine return mechanism. This instruction first pops a code location from the stack and then performs an unconditional jump to the retrieved code location. Syntax CALL <label> RET
CLC	CLC is the mnemonic for Clear Carry. It is a flag manipulation instruction that sets the carry flag to 0.
STC	STC is the mnemonic for Set Carry. It sets the carry flag to 1.
SHL,	SHL is the mnemonic for Shift to the Left. Likewise SHR stands for Shift to the

SHR	Right. SHL and SHR are called logical shifts. Examples: SHL EAX 1 ; Multiply the value of EAX by 2 (if the most significant bit is 0) SHR EBX CL ; Store in EBX the result of dividing the value of EBX by 2^n ; where n is the value in CL (when conditions apply) Unlike an arithmetic shift, a logical shift does not preserve the bits that are shifted out of the register.
SAL, SAR	SAL is the mnemonic for Shift Arithmetic Left. Likewise SAR stands for Shift Arithmetic Right. The arithmetic shift is sometimes known as a signed shift because it preserves the sign of the number.
RCL, RCR	RCL is the mnemonic for Rotate through Carry Left. Likewise RCR stands for Rotate through Carry Right. These operations perform a perfect rotation but not through the register only but through the register plus the carry bit. See diagram further down.

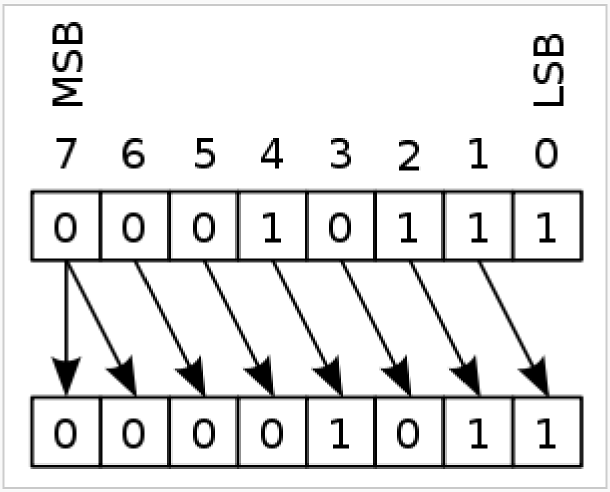


Logical left shift one bit



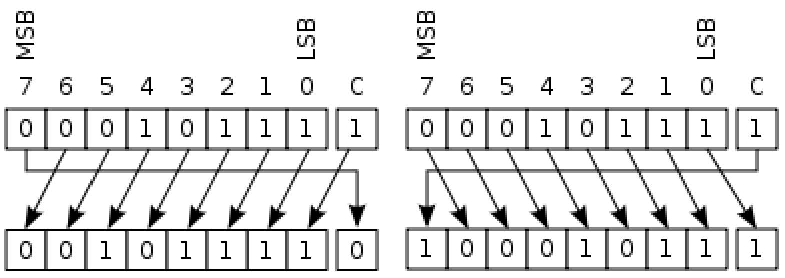
Logical right shift one bit

Logical shift by one bit



A right arithmetic shift of a binary number by 1. The empty position in the most significant bit is filled with a copy of the original MSB.

A right arithmetic shift



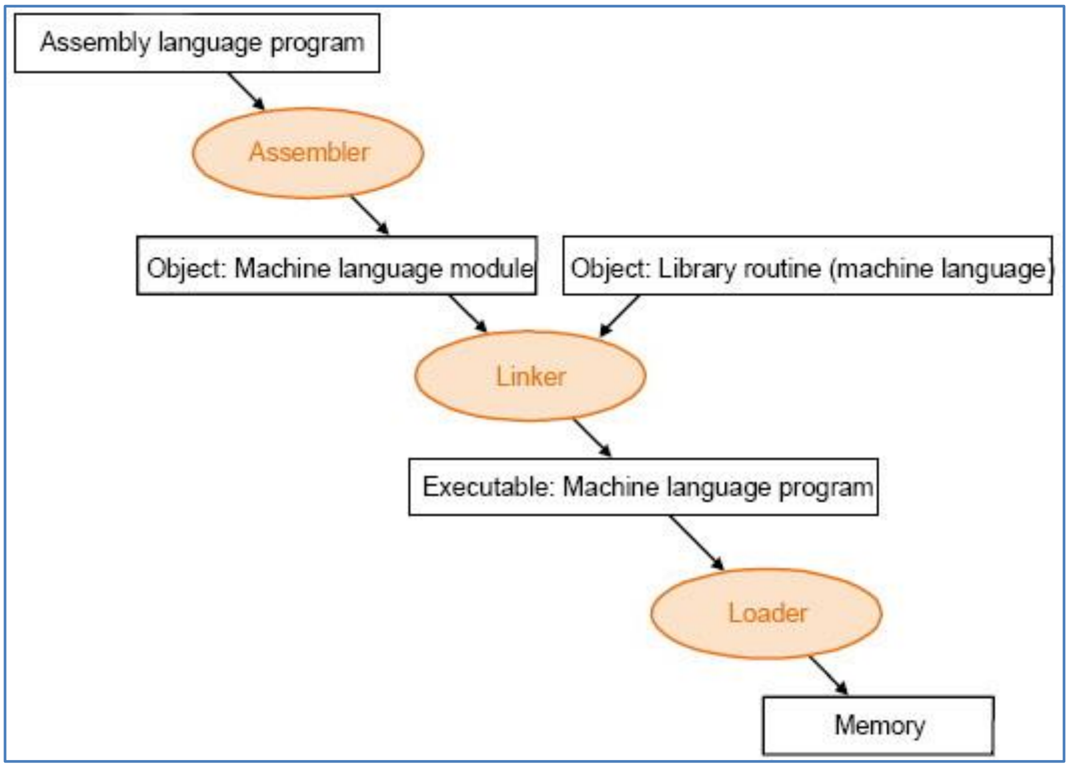
Left rotate through carry and right rotate through carry

3.6 Assemblers

An assembler translates a program from assembly language to machine code. An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.

Assemblers are classified on the number of times it takes them to read the source code before translating it; there are both single-pass and multi-pass assemblers. Moreover, some high-end assemblers provide enhanced functionality by enabling the use of control statements, data abstraction services and providing support for object-oriented programming structures.

3.6.1 The Assembly Process


Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

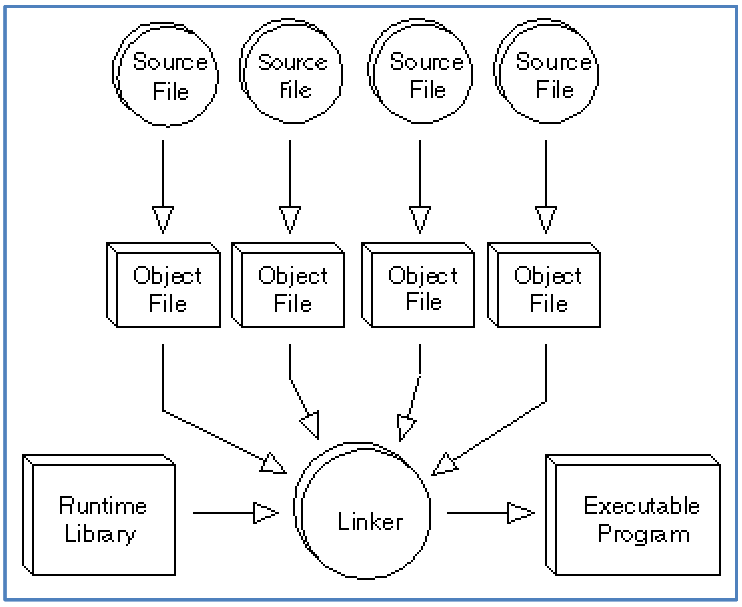
The assembly process

The assembly process consists of the following steps:

1) Assembling the source code into an object file
2) Linking the object file with other modules or libraries into an executable program

3) Loading the program into memory
4) Running the program

Linker



Linker

A linker, also called link editor and binder, is a program that combines object modules to form an executable program. Many programming languages allow you to write different pieces of code, called modules, separately. This simplifies the programming task because you can break a large program into small, more manageable pieces. Eventually, though, you need to put all the modules together. This is the job of the linker.

In addition to combining modules, a linker also replaces symbolic addresses with real addresses. Therefore, you may need to link a program even if it contains only one module.

Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is both faster and more portable, since it does not require the presence of the library on the system where it is run.

Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both the executable and the library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.

Loader

A loader is an operating system utility that copies programs from a storage device to main memory, where they can be executed. In addition to copying a program into main memory, the loader can also replace virtual addresses with physical addresses.
Most loaders are transparent, i.e., you cannot directly execute them, but the operating system uses them when necessary.

Relocation

Relocation is the process of assigning addresses to various parts of a program and adjusting the code and data in the program to reflect the assigned addresses. A linker usually performs relocation in conjunction with symbol resolution, the process of searching files and libraries to replace symbolic references or names of libraries with actual usable addresses in memory before running a program. Although relocation is typically done by the linker at link time, it can also be done at execution time by a relocating loader, or by the running program itself.

3.6.2 Cross Assembler

A cross assembler is an assembler that generates machine language for a different type of computer than the one the assembler is running in. It is used to develop programs for computers on a chip or microprocessors used in specialized applications that are either too small or are otherwise incapable of handling the development software.

3.6.3 Macro Assembler

Macro assemblers include a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging.

3.6.4 Meta Assembler

Meta-assembler is a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language.