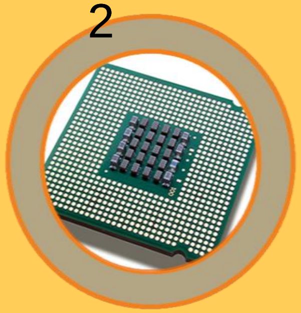
# 8.2: Features of Low-Level Languages



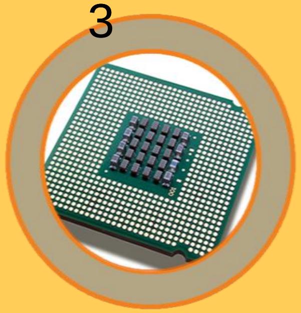
## What this module is about

- In this module we discuss:

a. explain the concepts and, using examples, demonstrate an understanding of the use of the accumulator, registers, and program counter;

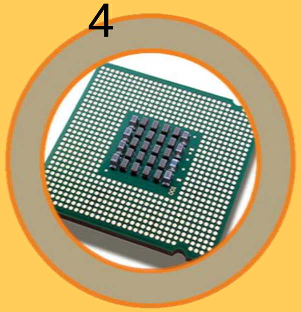
b. describe immediate, direct, indirect, relative and indexed addressing of memory when referring to low-level languages;

**c. discuss the concepts and, using examples, show an understanding of mnemonics, opcode, operand and symbolic addressing in assembly language to include simple arithmetic operations, data transfer and flow-control.**



## Low-Level Languages

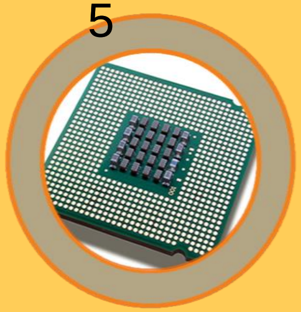
- A low level language is one whose programming statements are geared towards a particular CPU family, such as the x86 family of processors. Low level languages are almost (but not quite) machine code.
- 'Assembly language' is an example of a low level programming language.
- Chip makers such as Intel and ARM provide programmers with an Assembly Language with which to code their particular CPU.



# Low-Level Languages

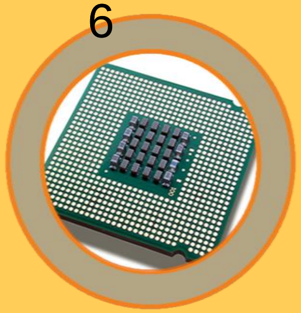
- Some features of Low Level languages include
  - They are CPU specific, making direct use of internal registers
  - 'Mnemonics' are used as programming code such as MOV or ADD
  - Many different memory modes can be used (previous presentation)
  - Labels are used as reference points to allow the code to jump from one part to another.
- Advantages
  - Low level languages allow for close control of the CPU, for example many device drivers are coded in assembly language.
  - They can be very efficient. Well-optimised code written in a low level language can be made to run very quickly compared to other programming paradigms.
- Disadvantages
  - They are difficult to use as the programming commands can be quite obscure
  - A good assembly language programmer needs to know a lot of detail about the internal structure of the CPU - e.g.its registers and memory management methods
  - Low level languages produce the least portable source code.
- Assembly language looks like this:

```
.MODEL SMALL;
.STACK;
.CODE;
mov ah,1h; moves the value 1h to register ah
mov cx,07h;moves the value 07h to register cx
int 10h;
```



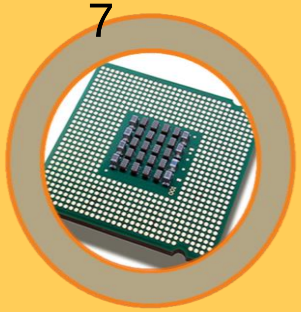
## Mnemonics

- The registers within a CPU store and process binary data. In theory the programmer could load data and instructions directly into the registers in pure binary, like this:
  - 10110000 00110010
- This instructs a Intel 8086 processor to load 32 into its accumulator. Difficult to remember!
- A slightly easier way of coding this would be to code the same instruction in hexadecimal. Like this:
  - B0 32
- This is '**machine code**' and it is difficult to program at this level and yet retain an understanding of what the software is doing. So the CPU chip makers supply a set of **Mnemonics** for the programmer to use with their processors.
- Mnemonics are a set of programming instructions that are later translated into pure machine code by a piece of software called an '**assembler**'.
- For example the machine code above in mnemonic form looks like:
  - MOV AL, 32h
- This makes more sense to the programmer. You can see that the command is instructing the CPU to load 32 hex immediately into a register called AL.



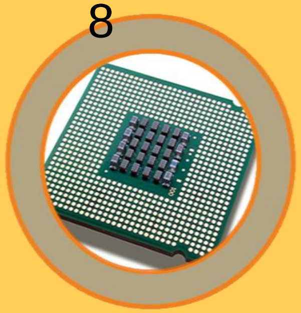
# Instruction Set

- Assembly language consists of a set of mnemonics that can be used to program a CPU. There are a number of mnemonics that together make up the complete **instruction set** of the CPU. They can be grouped according to the kind of processing they cover.
- For example, in the Intel 80186 instruction set some of the mnemonics are
  - **Arithmetic mnemonics** : ADD, SUB, DIV, MUL
  - **Data transfer mnemonics** : MOV, POP, IN, OUT
  - **Logic mnemonics** : AND, OR, XOR, NOT
  - **Jump mnemonics** : JMP, JZ (jump if zero)
  - **Miscellaneous mnemonics** : NOP (Do nothing for a bit)
- One of the skills in writing assembly code is to become familiar with the instruction set of the target CPU.



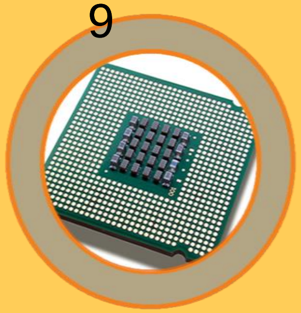
## Opcodes and Operands

- An opcode is short for '**Operation Code**'.
- An opcode is a single instruction that can be executed by the CPU. In machine language it is a binary or hexadecimal value such as 'B6' loaded into the instruction register.
- In assembly language mnemonic form an opcode is a command such as MOV or ADD or JMP.
- For example
  - MOV, AL, 34h
- The opcode is the MOV instruction. The other parts are called the '**operands**'.
- Operands are manipulated by the opcode. In this example, the operands are the register named AL and the value 34 hex.



## Symbolic addressing

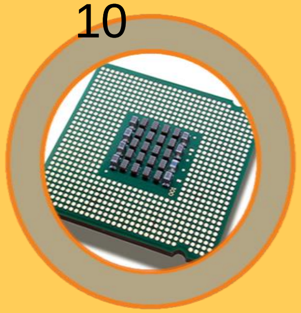
- Another key aspect of programming is to fetch or store data and instructions from memory.
- The simplest way to do this is to refer directly to a memory location such as #3001. But this brings a number of problems (remember last presentation)
  - it is difficult to see the meaning of the data in location #3001
  - the data may not be able to be located at #3001 because another program is already using that location.
- To overcome this issue, the idea of 'symbolic addressing' is used. Instead of referring to an absolute location, the assembly language allows you to define a 'symbol' for the data item or location.



# Symbolic addressing

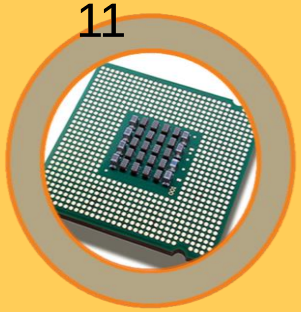
```
        .MODEL SMALL      Use small memory model
        .STACK 2048       define the stack
DIAM    EQU 2             define a constant called diam
VarA    DB                define a variable called VarA as a byte
VarB    DW                define a variable called VarB as a word
main:   MOV AL,[VarA]     Move data in VarB into register AL
```

- The symbols being defined by this bit of code are DIAM, VarA, VarB. In the case of DIAM it is referring to a constant value of 2. The size of the variables VarA and VarB are defined, but notice that their location is not defined.
- It is the job of the assembler to resolve the variables into locations in memory



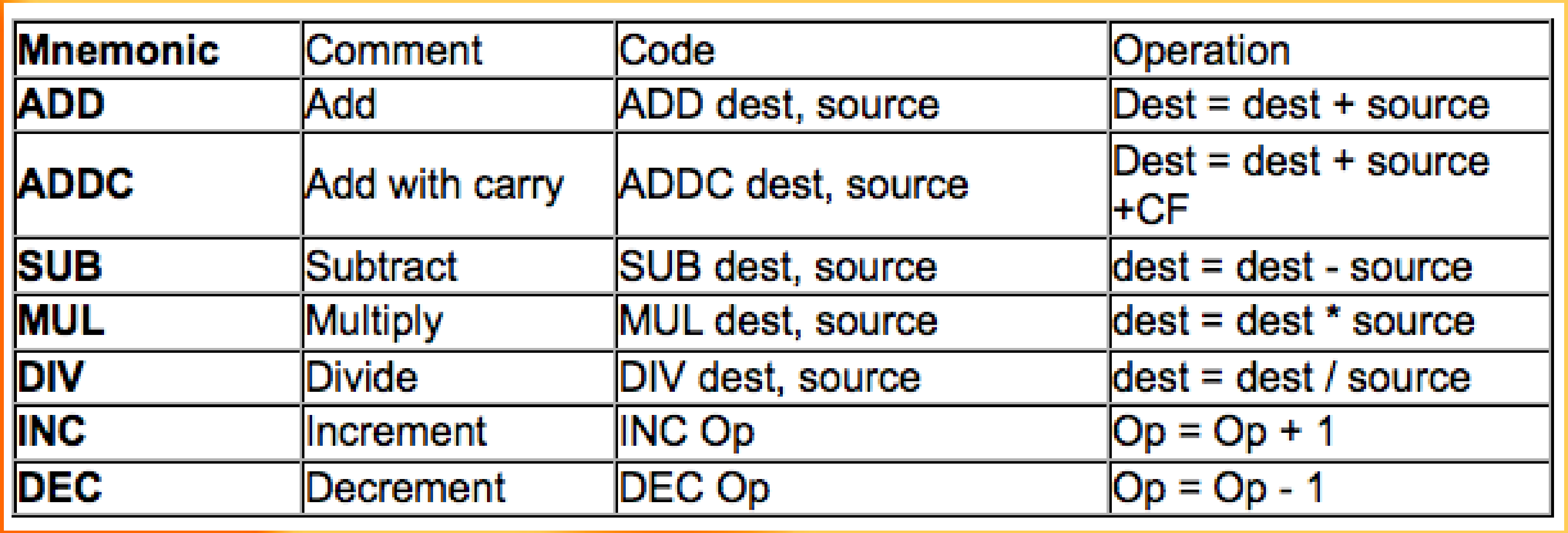
## Symbolic addressing

- The advantages of using symbolic addressing over direct memory references are:
  - The program is re-locatable in memory. It does not particularly care about its absolute location, it will still work
  - Using symbols makes the software much more understandable
- When the code is ready to be loaded and run, a '**symbol table**' is created by the assembler for the linker and loader to use to place the software into memory.

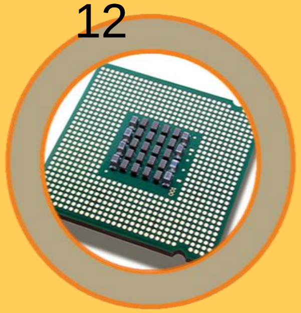


# Arithmetic operations

- *Any software language needs to support the basic functions of a running program and a low level language is no different.*
- *It must be able to support the basic arithmetic operations of adding, subtracting, multiplying and dividing.*
- *These operations are determined by a set of fairly self-evident mnemonics.*

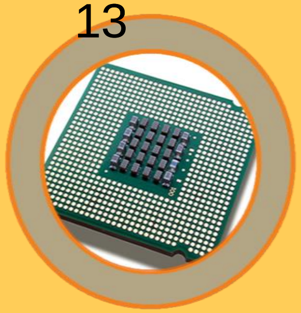


| Mnemonic | Comment | Code | Operation |
|---|---|---|---|
| ADD | Add | ADD dest, source | Dest = dest + source |
| ADDC | Add with carry | ADDC dest, source | Dest = dest + source +CF |
| SUB | Subtract | SUB dest, source | dest = dest - source |
| MUL | Multiply | MUL dest, source | dest = dest * source |
| DIV | Divide | DIV dest, source | dest = dest / source |
| INC | Increment | INC Op | Op = Op + 1 |
| DEC | Decrement | DEC Op | Op = Op - 1 |



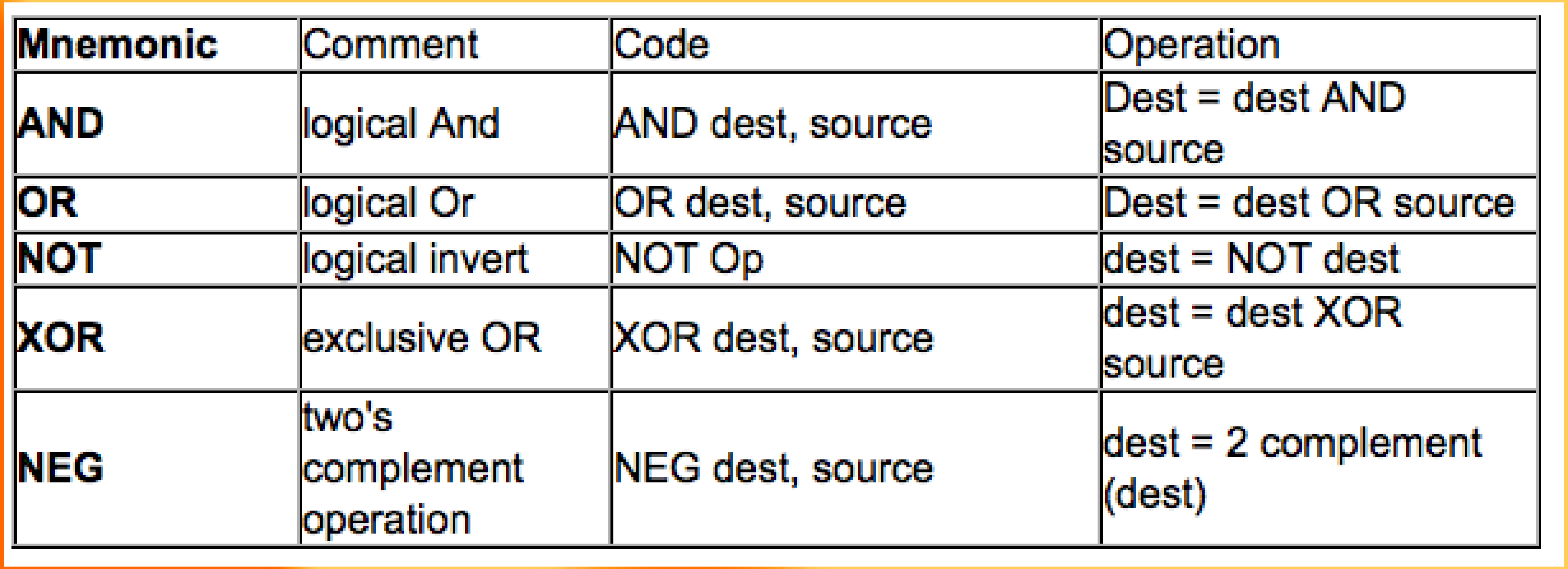
# Arithmetic operations

- The full instruction set of an actual CPU also includes variations to the basic arithmetic operations such as signed arithmetic, bit shifts and bit rotations.
- Operations such as additions have a single opcode followed by two operands. On the other hand operations such as incrementing and decrementing only need one operand.
- There are **flags** also available in the Program Status Word (PSW) register to indicate that the operation may have caused an overflow or underflow. There is also a **carry flag** available to indicate if the operation has resulted in an arithmetic carry (or borrow) has occurred

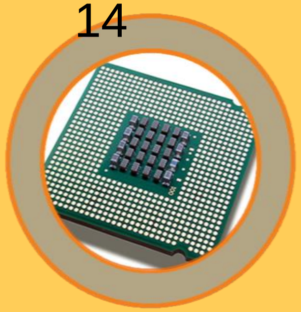


# Logic operations

- As well as the usual arithmetic operations, a CPU will also be performing logic operations.
- These logical operations will also set or reset a number of flags in the Program Status Word (PSW) depending on the outcome. These flags can then be tested to cause a branch or a jump in the code to occur.

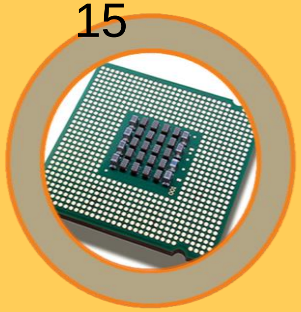


| **Mnemonic** | Comment | Code | Operation |
|---|---|---|---|
| **AND** | logical And | AND dest, source | Dest = dest AND source |
| **OR** | logical Or | OR dest, source | Dest = dest OR source |
| **NOT** | logical invert | NOT Op | dest = NOT dest |
| **XOR** | exclusive OR | XOR dest, source | dest = dest XOR source |
| **NEG** | two's complement operation | NEG dest, source | dest = 2 complement (dest) |



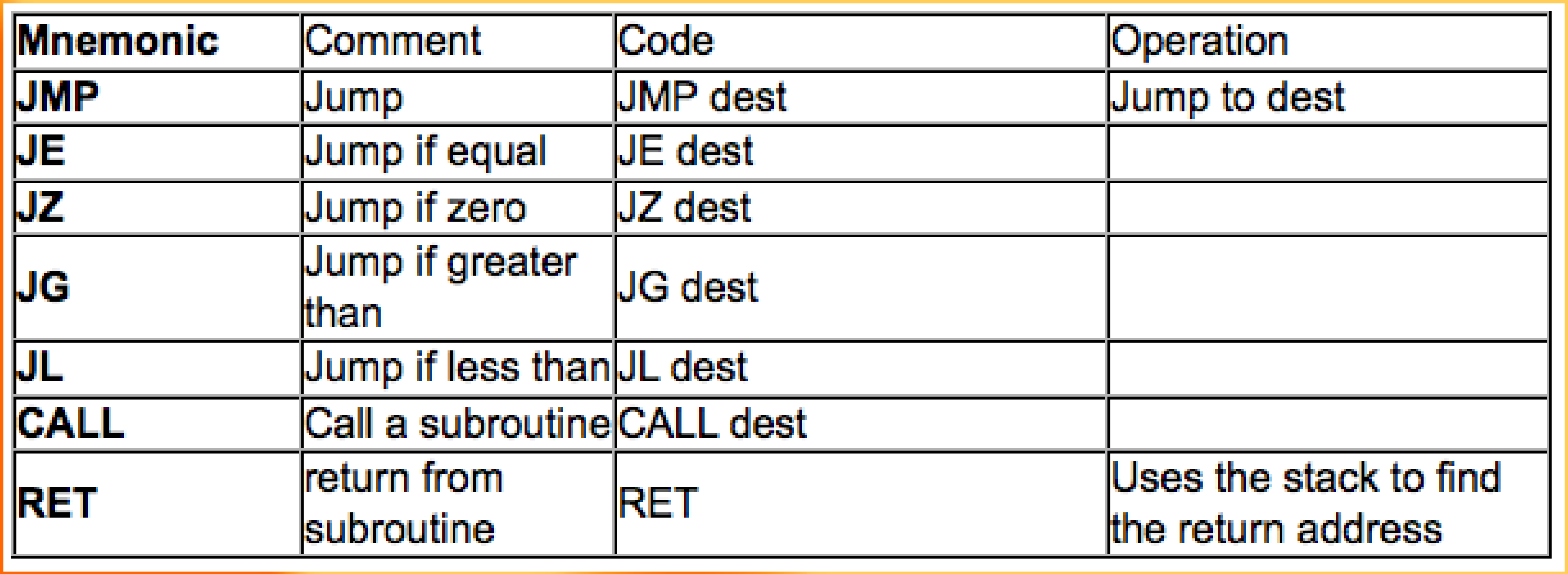
# Jumps and branching

- Most programs need to run code that is dependent on the outcome of some test within the software, this is called 'branching'.
- Common programming forms include loops, iteration and subroutine calls, and these need branching and jumping to work.
- Sometimes the code needs to jump to another location regardless. This is called an **unconditional jump**. For example to avoid the next line of code being executed at the end of a routine.
- On the other hand, a jump may be dependent on some condition being met such as a flag being 1 or 0. This is called a **conditional jump**.
- A **relative jump** will cause the instruction an offset number away from the current one to be executed. An **absolute jump** will go to the instruction at a specific address.

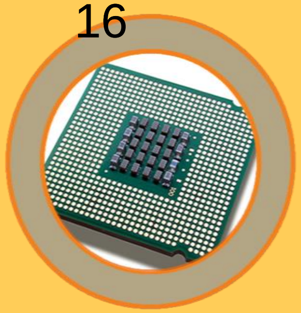


# Jumps and branching

- The conditional jumps shown usually look to the Program Status Word (PSW) register to see if the zero or carry flag has been set, if the condition is true, then the program counter is loaded with the address of the instruction given in the jump command.
- The CALL and RET commands make use of the stack to locate subroutine locations and return addresses.

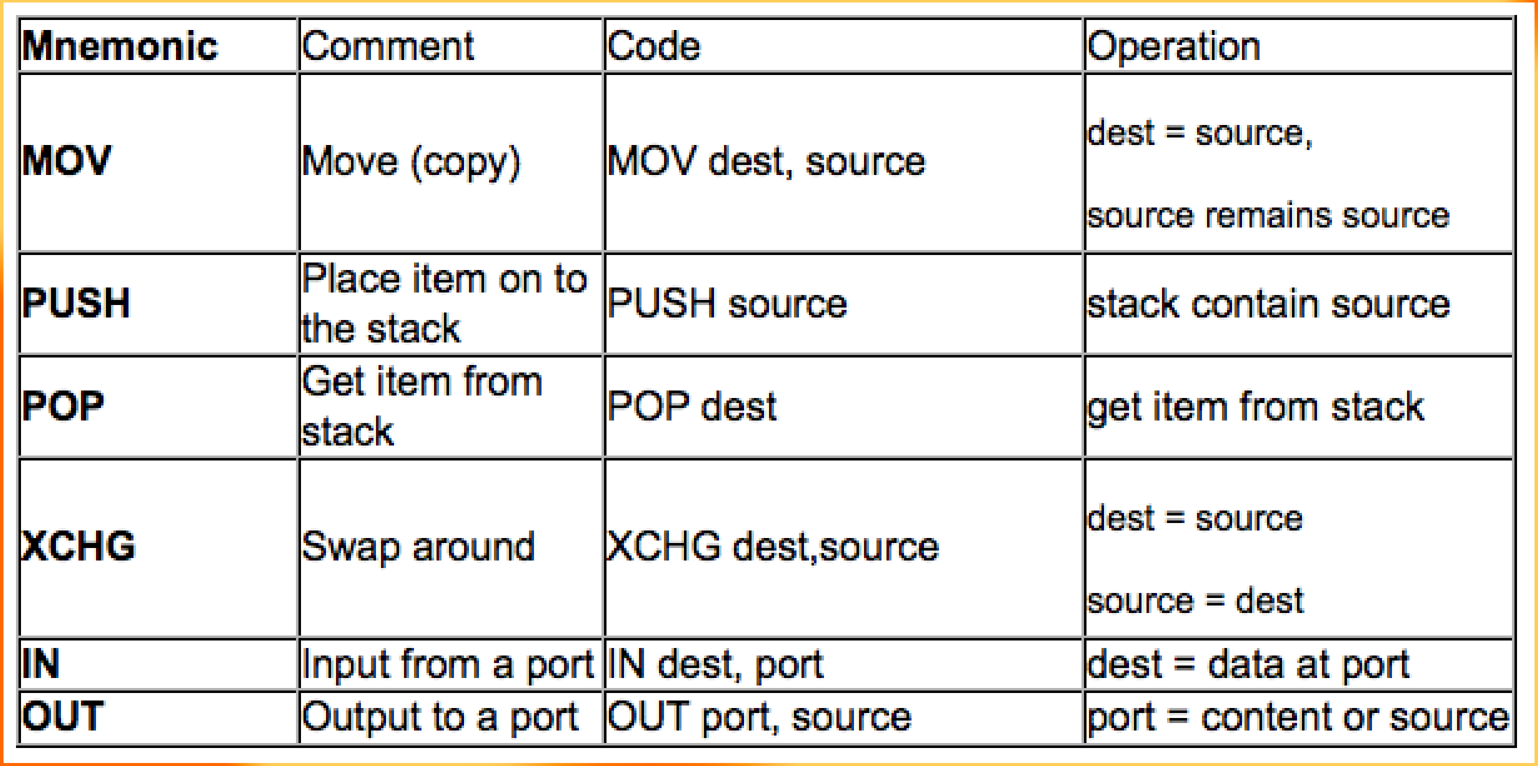


| Mnemonic | Comment | Code | Operation |
|---|---|---|---|
| JMP | Jump | JMP dest | Jump to dest |
| JE | Jump if equal | JE dest | |
| JZ | Jump if zero | JZ dest | |
| JG | Jump if greater than | JG dest | |
| JL | Jump if less than | JL dest | |
| CALL | Call a subroutine | CALL dest | |
| RET | return from subroutine | RET | Uses the stack to find the return address |

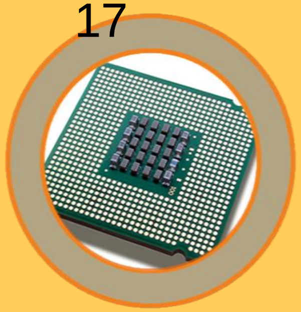


## Data transfer

- A CPU, in order to process data, needs to move that data around. This includes moving data between its internal registers and it includes moving data in an out of external RAM.



| Mnemonic | Comment | Code | Operation |
|---|---|---|---|
| **MOV** | Move (copy) | MOV dest, source | dest = source,<br>source remains source |
| **PUSH** | Place item on to the stack | PUSH source | stack contain source |
| **POP** | Get item from stack | POP dest | get item from stack |
| **XCHG** | Swap around | XCHG dest,source | dest = source<br>source = dest |
| **IN** | Input from a port | IN dest, port | dest = data at port |
| **OUT** | Output to a port | OUT port, source | port = content or source |



## Data transfer

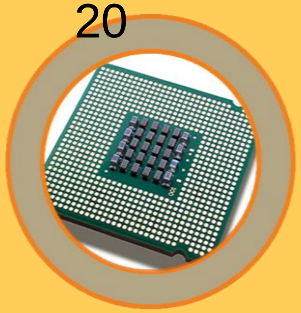
- The MOV command may include a number of variants within the full instruction set, but it basically copies data from one location to another.
- The PUSH and POP operations are the commands that control the contents of the stack. The stack is used to control subroutine calls and returning from subroutines.
- A CPU may have a number of 'ports'. A port is one or more physical pins on the chip assigned to handling data moving into and out of the chip.
- For example a serial port or an 8 bit data port may be available on the chip. CPUs especially developed for control purposes have many complicated ports, for example an automotive engine management CPU.

## Example

| | | |
|---|---|---|
| | .MODEL SMALL | *Define the memory model to be used* |
| | .STACK 2048 | *Set up the stack* |
| **Two** | EQU 2 | *Define a constant* |
| **VarB** | DB | *Define a symbol as a byte* |
| **VarW** | DW | *Define a symbol as a word* |
| **S** | DB "Hello World", 0 | *Define a string* |
| | .CODE | *Start of code block* |
| **main:** | MOV AX, DGROUP | *Move an address into a word register* |
| | MOV DS, AX | |
| | MOV [VarB], 35 | *Initialise variable called VarB* |
| | SUB VarB, 35 | *Subtract 35 from VarB, this will cause a zero flag in the PSW to be set* |
| | JZ :mysub | *Jump if zero to label mysub* |
| | OUT #1,3 | *If not zero then output data to port 1* |
| | JMP :leave | *Unconditionally jump to label leave* |
| **mysub:** | IN AL, #1 | *Read port 1 and load into register AL* |
| **leave:** | MOV AL, 34 | |

# Example Notes

- The first statement .MODEL SMALL is a telling the assembler what kind of memory model to use, in this case 'small' will probably mean only a 16 bit word is needed for addressing purposes. This is faster than having to fully resolve a full 32 bit word address every time.
- Then the stack is declared along with some symbolic variables.
- Another declaration .CODE defines the start of the code itself.
- Labels such as main, mysub and leave are used to identify specific points within the code
- There are a number of MOV commands to shuffle data around
- There is a SUB arithmetic command that alters the value of the variable varB
- There is a conditional jump JZ (JZ :mysub) that is testing the outcome of the prior operation. This code is artificial as VarB was initialised with 35 and then 35 was subtracted, so the result will always be zero. Or this is may be an example of a typical software bug where the programmer did not intend to do that.
- You can also see an unconditional JMP command (JMP :leave) to by-pass the instruction at mysub.



## Further Task

- When you get home see if you can find any Intel x86 assembly code simulators that illustrate how all this really works.
- There may be lots of animations on the web to show this visually.