Microprocessor realizes each program instruction as the sequence of the following simple steps:

1. fetch next instruction or its part from memory and placing it in the instruction register,

2. increase instruction pointer register so that it would indicate the placement of the next instruction to be executed,

3. decode previously fetched program instruction,

4. whenever program instruction requires some data that is stored in computer memory, determine its localization is and read data into internal processor registers,

5. execute the program instruction and place its results at the proper location (registers/memory). Return to point 1 of the cycle.
Instruction coding

**Prefix** - repeat prefixes (REP, REPE, REPNE), segment override (CS, SS, DS, ES), operand-size or address-size override.

**Opcode** - in this field the type of instruction and its operands are coded.

**ModR/M, Reg** - required as specification of instruction operands.

**SIB** - Scale, Index, Base: used in 32-bit mode with instruction requiring addressing with scaled registers, e.g. [EBX*4].
Displacement - 1, 2 or 4 bytes displacement used with memory addressing,

Immediate value - immediate scalar 1, 2 or 4 byte long value used as an operand for instruction.
Example.

```
mov ax, es:[BX+1000h]
```

in machine code: 268B870010h

26h – ES,
8Bh – instruction opcode
   mov r16, r/m16
   mov r32, r/m32

87h – Mod R/M field:

```
r16=ax
m16=bx+16-bit displacement
```

0010h – displacement*

*it should be noted that this value is stored in memory in little endian ordering.
Instruction pipelines
Previously described instruction cycle is executed in strictly sequential order which can be illustrated with the following figure (to make problem simpler we assume only two stages in the cycle):

Instruction realization cycle in strictly sequential case

In case of sequential execution of program instructions the fetching stage of “i+1” instruction begins after the execution stage of “i” is finished. Some μP circuits (if they can be separated) are inactive for significant periods of time. Sequential execution of program instructions is characteristic of older microprocessors such as Z80, Intel 8080 etc.
There exist the great potential to improve the effectiveness of instructions execution cycle by allowing to overlap the execution of multiply instructions. This is possible when instruction cycle is divided into stages which can be executed by separated circuits in parallel fashion. It means that execution and fetching microprocessor circuits must be separated. Such scheme is called the instruction pipeline.

Instruction realization cycle in pipelined fashion

Instruction pipeline was introduced in Intel family starting from i8088 and i8086 microprocessors. This approach can significantly accelerate the process of program execution. However some gaps in pipeline may occur as result of jump/call instructions and inter-instruction dependences.
Pipeline in Pentium processor
Pentium processor is equipped with two parallel pipeline: V-pipeline and U-pipeline. Each pipeline has its own ALU, instruction decoder, address generator and cache memory buffer. Pipelines execute integer operand instructions in 5 consecutive stages.
Selected groups of instructions of Intel x86 microprocessor:
- *arithmetical operations*: perform arithmetical operations of addition (ADD), subtraction (SUB), division (DIV) and multiplication (MUL);
- *bit instructions*: instructions of bits shifting and rotating (SHL, SHR, ROL, ROR, RCL, RCR), bit alternative (OR), conjunction (AND, TEST), exclusive disjunction (XOR), etc.
- *comparison instruction* (CMP);
- *conditional jumps*: jumps taken under specific conditions indicated by processor flags (JE, JZ, JA, JAE, etc.):

<table>
<thead>
<tr>
<th>Instruction mnemonic</th>
<th>Description of jump conditions</th>
<th>Processor flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OF  CF ZF PF SF</td>
</tr>
<tr>
<td>Simple flag tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JE/JZ</td>
<td>Jump when zero</td>
<td>0</td>
</tr>
<tr>
<td>JP/JPE</td>
<td>Jump when even number of 1's</td>
<td>1</td>
</tr>
<tr>
<td>JO</td>
<td>Jump when overflow in U2 operations</td>
<td>1</td>
</tr>
<tr>
<td>JS</td>
<td>Jump when result is negative (U2)</td>
<td>1</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>Jump when not zero</td>
<td>0</td>
</tr>
<tr>
<td>JNP/JPO</td>
<td>Jump when odd number of 1's</td>
<td>0</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump when there is no overflow</td>
<td>0</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump when result is not negative</td>
<td>0</td>
</tr>
</tbody>
</table>
### Architecture and components of Computer System

#### Selected groups of instructions

<table>
<thead>
<tr>
<th>Instruction mnemonic</th>
<th>Description of jump conditions</th>
<th>Processor flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OF</td>
</tr>
<tr>
<td><strong>U2 code operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JL/JNGE</td>
<td>Jump when less (OF!=SF)</td>
<td>X</td>
</tr>
<tr>
<td>JLE/JNG</td>
<td>Jump when less or equal (ZF=1 or OF!=SF)</td>
<td>X</td>
</tr>
<tr>
<td>JNL/JGE</td>
<td>Jump when greater or equal (OF=SF)</td>
<td>X</td>
</tr>
<tr>
<td>JNLE/JG</td>
<td>Jump when greater (ZF=0 and OF=SF)</td>
<td>X</td>
</tr>
<tr>
<td><strong>Natural code</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JB/JNBE/JC</td>
<td>Jump when below</td>
<td>1</td>
</tr>
<tr>
<td>JBE/JNA</td>
<td>Jump when below or equal (CF=1 or ZF=1)</td>
<td>X</td>
</tr>
<tr>
<td>JNB/JAE/JNC</td>
<td>Jump when above or equal</td>
<td>0</td>
</tr>
<tr>
<td>JNBE/JA</td>
<td>Jump when above</td>
<td>0</td>
</tr>
<tr>
<td><strong>Test of CX register</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump when CX=0</td>
<td></td>
</tr>
</tbody>
</table>

- **unconditional jump** (JUMP), **procedure calling instruction** (CALL), **return from procedure** (RET);
- **stack operations** (PUSH, POP, PUSHF, POPF, PUSHA, POPA);
The address of stack top is stored in SS:SP registers.

**PUSH** xxx
SP <- SP-2
SS:[SP] <- xxx

**POP** xxx
xxx <- SS:[SP]
SP <- SP + 2

Example:
- **loop instructions:**

**LOOP, LOOPE (LOOPZ), LOOPNE, (LOOPNZ) label**

a) decrement CX by 1;
b) if CX is not equal to zero jump to label. In case of LOOPE and LOOPNE instructions there is also proper condition checked: ZF=1 for LOOPE and ZF=0 for LOOPNE. If that additional condition is not fulfilled then proceed to next program instruction regardless to CX value.

- **string operations:**

**REP, REPE (REPZ), REPNE (REPNZ)** (iterative execution of string instruction)

a) check whether CX is equal to zero, if so stop iterations and proceed to next program instruction;
b) execute string instruction;
c) decrement CX by 1;
d) in case of REPE and REPNE prefixes test ZF flag. If proper condition is fulfilled: ZF=1 for REPE and ZF=0 for REPNE than jump to a). For REP prefix jump to a) unconditionally.
**MOVSB, MOVSW, MOVSQ** (moving of block of data)

a) send byte (B), word (W, 16-bits) or double word (D, 32-bits) from localization ES:SI to address DS:DI;
b) If DF=0 then: SI=SI+1{2, 4}, DI=DI+1{2, 4}. Otherwise: SI=SI-1{2, 4}, DI=DI-1{2, 4}.

Sample code:

```assembly
.model small
.stack 100h
.data
tab1 db 256 dup(10)
tab2 db 256 dup(0)
N dw 256
.code
start:   mov ax, @data
          mov ds, ax
          mov es, ax
          mov si, offset tab1
          mov di, offset tab2
          mov cx, N
          cld
          rep movsb
          mov ax, 4C00h
          int 21h
.end start
```
**CMPS{B, W, D}** (compare blocks of data)

a) compare and set processor flags two memory cells of byte (B), word (W, 16-bits) or double word (D, 32-bits) length taken from locations indicated by: ES:SI and DS:DI;

b) If DF=0 then: SI=SI+1{2, 4}, DI=DI+1{2, 4}. Otherwise: SI=SI-1{2, 4}, DI=DI-1{2, 4}.

Sample code:

```assembly
.model small
.stack 100h
.data
txt1 db "Przykładowy tekst"
txt2 db "Przykładowe tekst"
N equ $ - txt2
txt3 db "identyczne", 10, 13, '¥'
txt4 db "rozne", 10, 13, '¥'
.code
.start:    mov ax, @data
           mov ds, ax
           mov es, ax
           mov si, offset txt1
           mov di, offset txt2
           mov cx, N
           cld
           repe cmpsb
           jz l1
           mov dx, offset txt4
           mov ah, 09h
           int 21h
           jmp l2
.l1:       mov dx, offset txt3
           mov ah, 09h
           int 21h
.l2:       mov ax, 4C00h
           int 21h
.end start
```
SCAS\{B, W, D\} (search for data in memory block)

a) subtract from AL\{AX, EAX\} one byte (B), word (W, 16-bits) or double word (D, 32-bits) taken from location indicated by: ES:DI. Result is volatile except from processor flags.
b) If DF=0 then: SI=SI+1\{2, 4\}, DI=DI+1\{2, 4\}. Otherwise: SI=SI-1\{2, 4\}, DI=DI-1\{2, 4\}.

Sample code:

```
.model small
.stack 100h
.data
txt1 db "Przykładowy tekst"
N equ $ - txt1
znak db 'u'
txt2 db "znaleziono", 10, 13, '
'txt3 db "brak", 10, 13, '
.code
start:  mov ax, @data
        mov ds, ax
        mov es, ax
        mov di, offset txt1
        mov cx, N
        mov al, znak
        cld
        repne scasb
        jz 11
        mov dx, offset txt3
        mov ah, 09h
        int 21h
        jmp l2
11:   mov dx, offset txt2
20:   mov ah, 09h
21:   int 21h
22:   jmp l2
23:   l2:  mov ax, 4C00h
24:   int 21h
25:   jmp l2
26:   end start
```
STOS{B, W, D} (filling block of memory)

a) send AL{AX, EAX} to address DS:DI;
b) If DF=0 then: SI=SI+1{2, 4}, DI=DI+1{2, 4}. Otherwise: SI=SI-1{2, 4}, DI=DI-1{2, 4}.

Sample code:

```assembly
.model small
.stack 100h
.data
.tab db 256 dup(?)
N equ $ - tab
znak db 100
.code
cdecl start: mov ax, @data
  mov ds, ax
  mov es, ax
  mov di, offset tab
  mov cx, N
  mov al, znak
cld
  rep stosb
  mov ax, 4C00h
  int 21h
end start
```
Selected groups of instructions

- *data transmission instructions*: MOV, XCHG, IN, OUT, etc.
- *processor state changing instructions*: CLD, STD, CLI, STI, etc.