# Basic Computer Operation Cycle



# Instruction Cycle

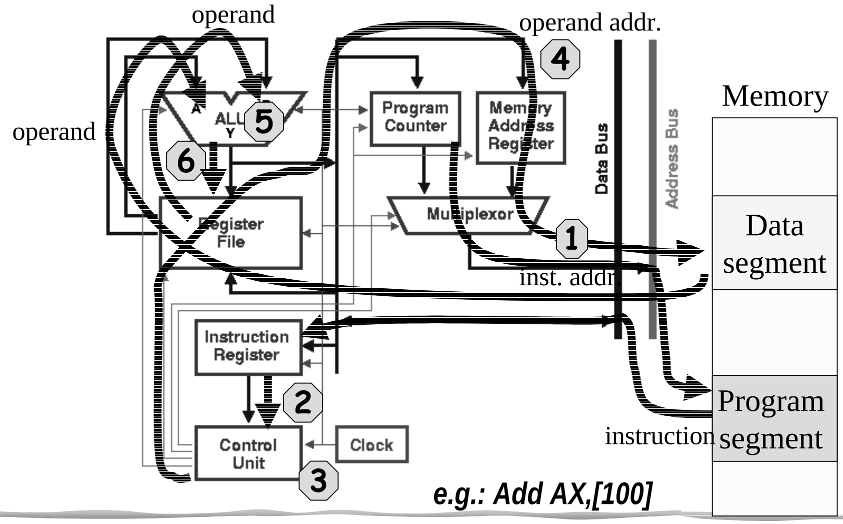
1. Fetch the instruction from memory into a control register
2. Decode the instruction
3. Locate the operands used by the instruction
4. Fetch operands from memory (if necessary)



# Instruction Cycle (Cont.)

5. Execute the operation in processor register
6. Store the results in the proper place
7. Go back to step 1 to fetch the next instruction






*e.g.: Add AX,[100]*

# Addressing Modes



# Instruction basics

- An instruction in computer contains
  - Opcode and
  - Operand(s) (Optional)
- Opcode specifies the operation to be performed
- Operand specifies the data to be processed



# Addressing modes

- Operands are required for some operations (e.g. ADD)
- *Addressing mode* tells how we can determine the exact location of the data (operand) we want to manipulate
- The more powerful a CPU, the more modes it supports



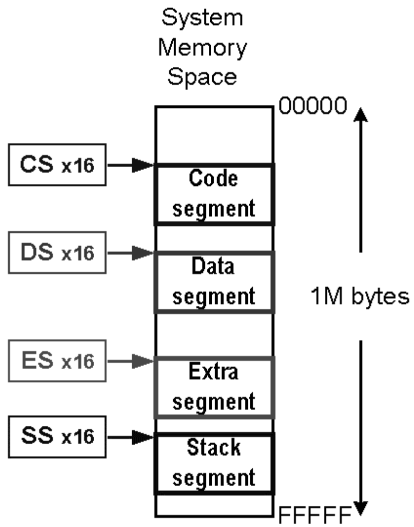
# Common addressing modes

- Implied Mode
- Immediate Mode
- Register and Register-Indirect Modes
- Direct Addressing Mode
- Indirect Addressing Mode
- Relative Addressing Mode
- Indexed Addressing Mode

- We use Intel's 8086 (real mode) as examples to study addressing modes



## Intel 8086's addressing scheme




**Segment address**

- starting address of a segment (after x16)
- specified by CS, DS, ES & SS
- 16-bit long

**Offset address**

- address relative to a segment address
- specified by IP, SI, DI & SP etc
- 16-bit long



- How to get the real address?

Effective address = segment address x 16 + offset address

Example:

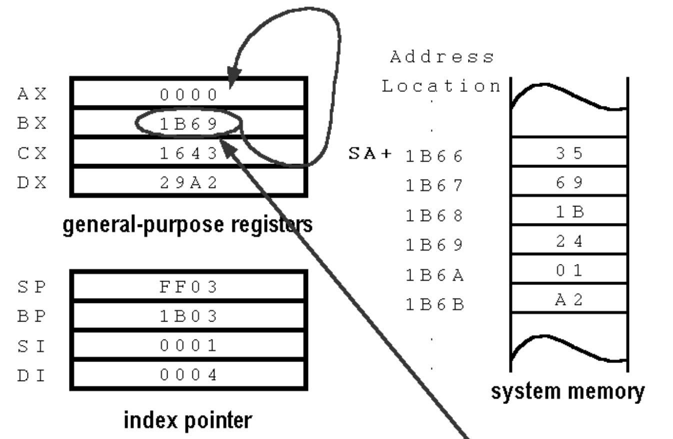
Segment addr. = 1031H

Offset addr. = 0023H

Effective addr. = 1031H x 10H + 0023H

= 10333H

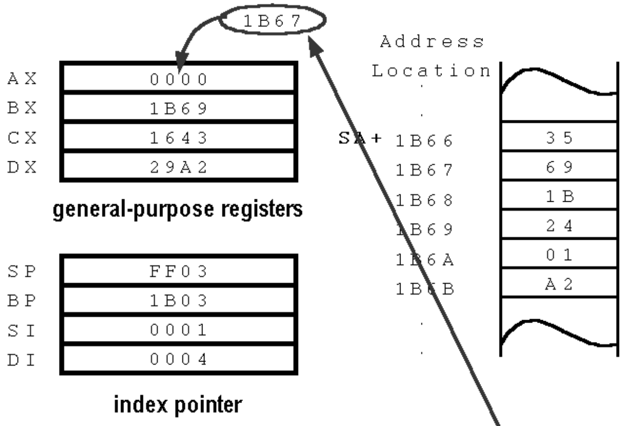





**Mode:** register

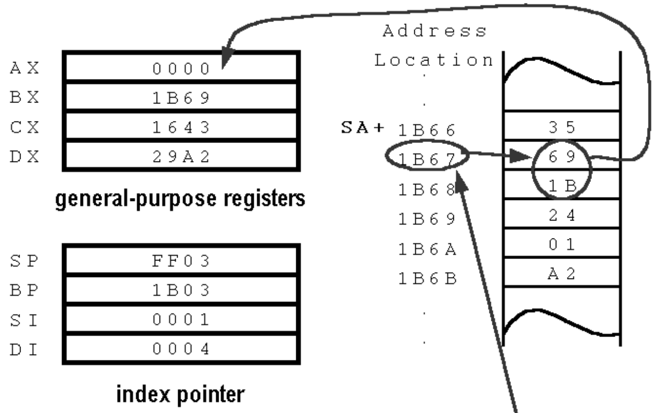
**Example:** MOV AX, BX

**Result:** 1B69H -> AX



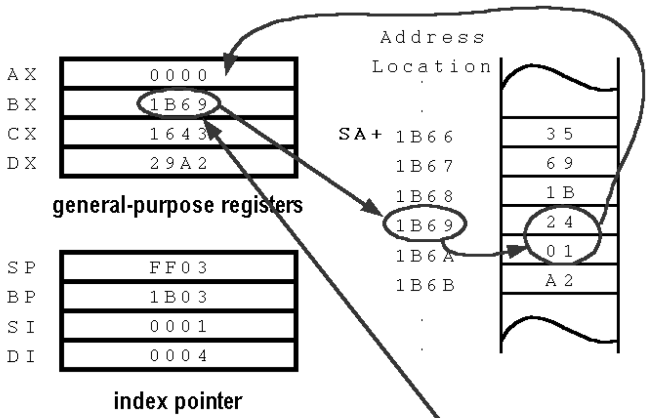

**Mode:** immediate
**Example:** MOV AX,1B67H
**Result:** 1B67H -> AX





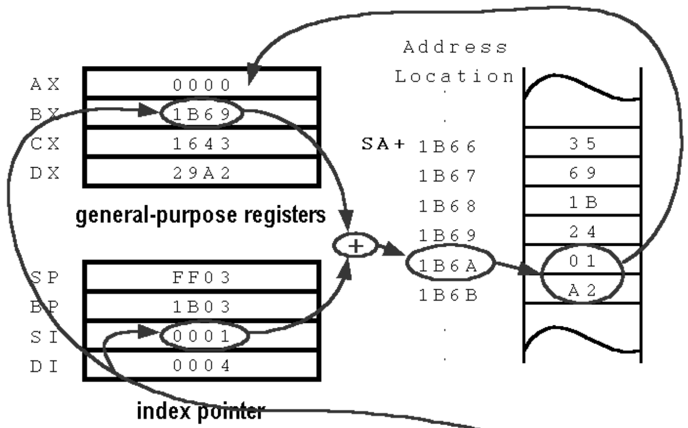

**Mode:** direct
**Example:** MOV AX,[1B67H]
**Result:** 1B69H -> AX



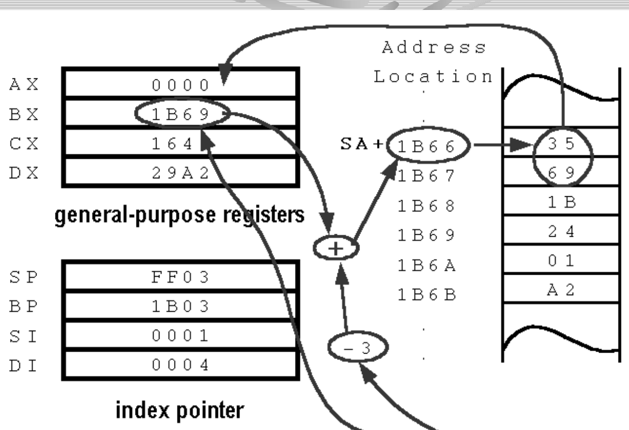



**Mode:** register indirect
**Example:** MOV AX,[BX]
**Result:** 0124H -> AX



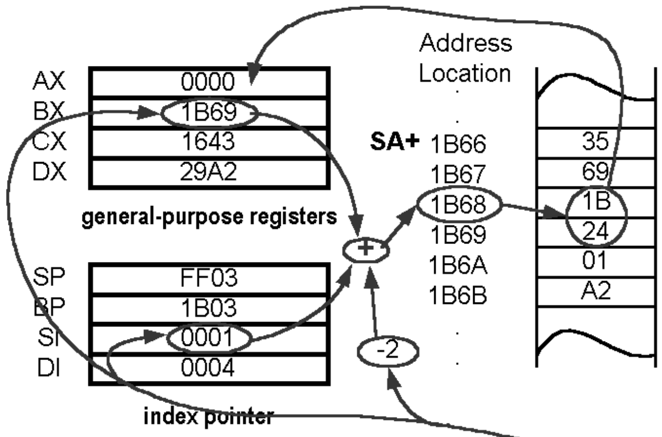



**Mode:** base-plus-index
**Example:** MOV AX,[BX+SI]
**Result:** A201H -> AX




**Mode:** register relative
**Example:** MOV AX,[BX-3H]
**Result:** 6935H -> AX



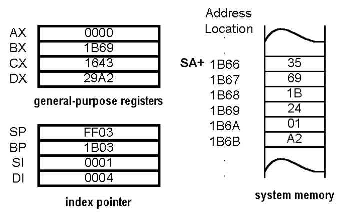



**Mode:** base relative-plus-index
**Example:** MOV AX,[BX+SI-2H]
**Result:** 241BH -> AX



# Summary of 8086 addressing modes

| | | | | |
|---|---|---|---|---|
| register | : | MOV AX,BX | : | 1B69H -> AX |
| immediate | : | MOV AX,1B67H | : | 1B67H -> AX |
| direct | : | MOV AX,[1B67H] | : | 1B69H -> AX |
| register indirect | : | MOV AX,[BX] | : | 0124H -> AX |
| base-plus-index | : | MOV AX,[BX+SI] | : | A201H -> AX |
| register relative | : | MOV AX,[BX-3H] | : | 6935H -> AX |
| base relative-plus-index | : | MOV AX,[BX+SI-2H] | : | 241BH -> AX |





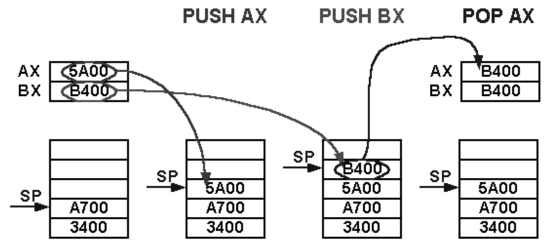

# stack memory

- The stack memory
  - holds data temporarily and
  - stores return addresses for procedures.
- The stack memory is maintained by the stack pointer (SP) and the stack segment register (SS) in Intel’s 8086.

# stack memory

- The stack memory is a LIFO (last-in first-out) memory, which describes the way that data are stored and removed from the stack.






**STACK operations**



# How to program a CPU?



# Machine language

- *Machine language* (machine code) is a sequence of bit patterns that a CPU can recognize and operate accordingly
- A bit pattern will activate pre-defined action of digital circuit inside the CPU chip and eventually perform a pre-defined operation

- Machine language is machine dependent
- It is difficult for humans to remember bit patterns but meaningful names, so we use names to denote patterns. (Each corresponds to an opcode.)



- Different combinations of opcode & operands form different instructions
- Each model of CPU has its own instruction set
- *Instruction set* defines the operation, clock required, addressing modes supported, max. no. of operands associated with an opcode



# Assembly language

- An *assembly language program* is actually a sequence of instructions
- Assembly language is a low level language, which means it is machine dependent
- High level language such as C/C++ and java is machine independent



- An assembly language program typically has 4 fields: label, opcode, operand and comment of an instruction
- *Addressing modes* tells how we can determine the exact location of the data (operand) we want to manipulate

- Humans use assembly language to write programs and use assembler to translate assembly programs into machine codes
- "*Debug*" is an environment for one to learn and debug 8086 assembly language programs



## Commands supported by *DEBUG*

```
assemble     A [address]
compare      C range address
dump         D [range]
enter        E address [list]
fill         F range list
go           G [=address] [addresses]
hex          H value1 value2
input        I port
load         L [address] [drive] [firstsector] [number]
move         M range address
name         N [pathname] [arglist]
output       O port byte
proceed      P [=address] [number]
quit         Q
register     R [register]
search       S range list
trace        T [=address] [value]
unassemble   U [range]
write        W [address] [drive] [firstsector] [number]
allocate expanded memory        XA [#pages]
deallocate expanded memory      XD [handle]
map expanded memory pages       XM [Lpage] [Ppage] [handle]
display expanded memory status  XS
```



**Example 1**

Compute 40H+50H and store the result in register CX.

```
C:\WINDOWS>debug
-a
1AF1:0100 mov ax,50
1AF1:0103 mov bx,40
1AF1:0106 add ax,bx
1AF1:0108 mov cx,ax
1AF1:010A
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=0100   NV UP EI PL NZ NA PO NC
1AF1:0100 B85000        MOV     AX,0050
-t

AX=0050  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=0103   NV UP EI PL NZ NA PO NC
1AF1:0103 BB4000        MOV     BX,0040
-t

AX=0050  BX=0040  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=0106   NV UP EI PL NZ NA PO NC
1AF1:0106 01D8          ADD     AX,BX
-t

AX=0090  BX=0040  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=0108   NV UP EI PL NZ NA PE NC
1AF1:0108 89C1          MOV     CX,AX
-t

AX=0090  BX=0040  CX=0090  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=010A   NV UP EI PL NZ NA PE NC
1AF1:010A 2CCD          SUB     AL,CD
-
```



**Example 2**

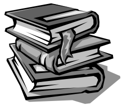
Compute 1+2..+10 and store the result in register AX

```
C:\WINDOWS>debug
-a
1AF1:0100 mov ax,0   ; clear reg. ax
1AF1:0103 mov cx,0a  ; init ctr=10 (ten)
1AF1:0106 add ax,cx  ; accumulate values stored in reg. cx
1AF1:0108 dec cx     ; decrease ctr by 1
1AF1:0109 jnz 106    ; repeat until cx=0
1AF1:010B nop
1AF1:010C

-u 100
1AF1:0100 B80000        MOV     AX,0000
1AF1:0103 B90A00        MOV     CX,000A
1AF1:0106 01C8          ADD     AX,CX
1AF1:0108 49            DEC     CX
1AF1:0109 75FB          JNZ     0106
1AF1:010B 90            NOP
1AF1:010C 7514          JNZ     0122

-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=0100   NV UP EI PL NZ NA PO NC
1AF1:0100 B80000        MOV     AX,0000
-g 10c

AX=0037  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1AF1  ES=1AF1  SS=1AF1  CS=1AF1  IP=010C   NV UP EI PL ZR NA PE NC
1AF1:010C 7514          JNZ     0122
-
```



# Learn more

- CPU Design and Architecture
  http://www.linux.se/doc/HOWTO/CPU-Design-HOWTO-3.html
- http://www.eie.polyu.edu.hk/~enyhchan/HowItWorks.htm
  - How CPU Works
- Microprocessor instruction set cards
  http://vmoc.museophile.com/cards/



# What you’re expected to know

- Basic instruction cycle
  - procedures of executing an instruction
- Different addressing modes
  - How to get the data to do the operation when an instruction is given?
- How stack memory operates?



# What you’re expected to know

- Use simple 8086 operations such as MOV, ADD, SUB, DEC, INC, JC and JNC to write a simple assembly language program to implement a task.
- Trace a simple sequence of 8086 instructions and derive its execution result.