System Level Programming

Software College of SCU

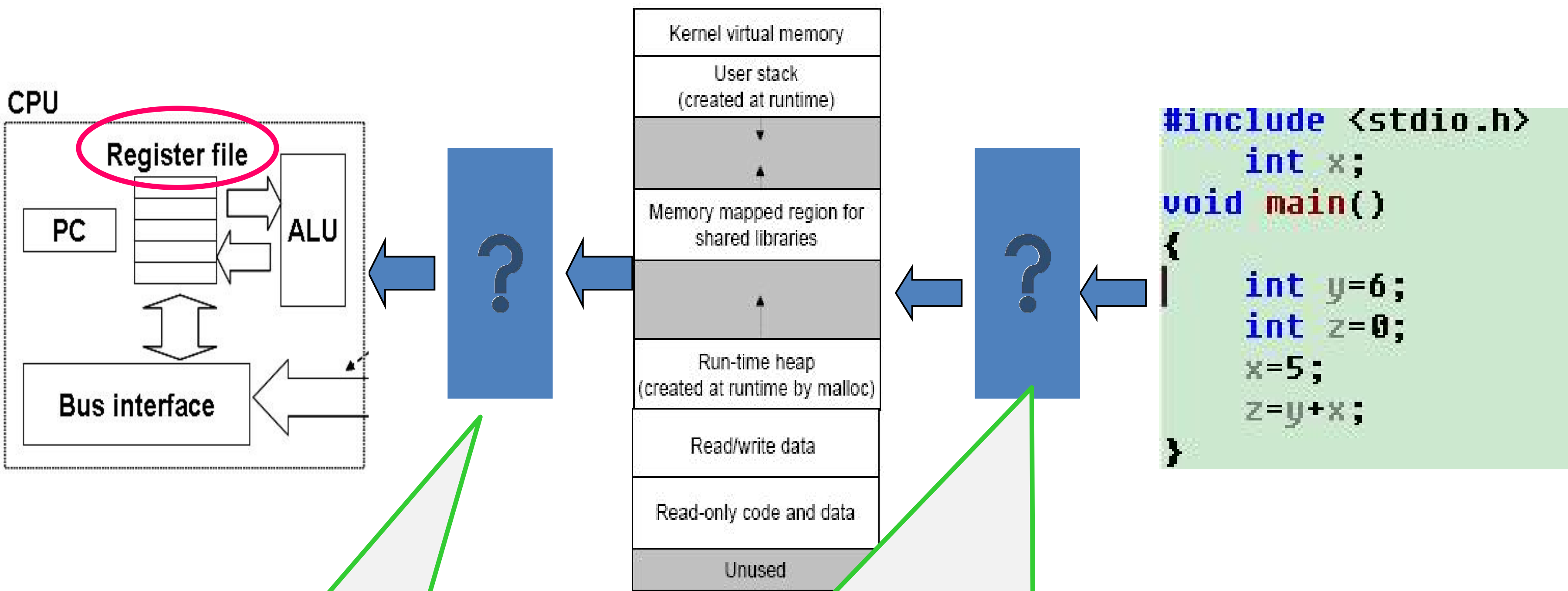
Week03

Unit 3. Representation of Code

- 3.0 Brief Introduction : SLP 1.5
- 3.1 X86 Instructions : CSAPP Chapter 3(GCC) & intel microprocess
- 3.2 x86 Assembly Language Programming
- 3.3 Disassembly in IDE

3.0 Brief Introduction (1/15)

- A micro view



3.How does CPU execute machine instruction?

1.How to translate assembly code into machine code?
2.How to allocate memory?

3.0 Brief Introduction (2/15)

- Questions
  - What a CPU can and can not understand
    - Can not: "c = (a + b) * d"
    - Can: "Add the contents of these two registers together and put the result in the first one."
  - How to encode high level language instructions to machine instructions?

3.0 Brief Introduction (3/15)

- **You will learn:**
  - Assembly language Review
  - C statements *VS* Assembly language instructions

- **Why learn it?**

3.0 Brief Introduction (4/15)

- Since compilers exist, why learn how to write assembly code?
  - Have complete control over hardware
  - Understand hardware-level program execution
    - Important for understanding security vulnerabilities, and how to avoid introducing them
  - Optimize performance-critical code
  - Implement code generators (compilers, JIT compilers)

3.0 Brief Introduction (5/15)

- Q1: One way to inspect the machine code

```
#include <stdio.h>
int x;
void main()
{
    int y=6; int z=0;  unsigned char *cptr;            int i;
    x=5;
    z=y+x;
    cptr = ((unsigned char *) 0x0040D746);   // address of instruction ”x=5”
    for (i = 0; i < 10; i++)    {      // 0x0040D750-0x0040D746 = 10(decimal)
         printf("%02X ", *cptr);
         cptr++;
    }  //one way to inspect machine code
    printf("\n");
}
```

3.0 Brief Introduction (6/15)

- The code and data are stored in memory separately.
  - Code(cs+ip): mov eax,dword ptr[ebp-4]
  - Data(stack+ebp): in stack, accessed via [ebp-4]

```
5:          int y=6;
0040D738    mov         dword ptr [ebp-4],6
6:          int z=0;
0040D73F    mov         dword ptr [ebp-8],0
7:          unsigned char *cptr;
8:          int i;
9:          x=5;
0040D746    mov         dword ptr [_x (00427e34)],5
10:         z=y+x;
0040D750    mov         eax,dword ptr [ebp-4]
0040D753    add         eax,dword ptr [_x (00427e34)]
0040D759    mov         dword ptr [ebp-8],eax
11:
12:         cptr = ((unsigned char *) 0x0040D746);   // address of instruction ”a=5”
0040D75C    mov         dword ptr [ebp-0Ch],offset main+26h (0040d746)
13:         for (i = 0; i < 10; i++) // 0x0040D750-0x0040D746 = 10(decimal)
0040D763    mov         dword ptr [ebp-10h],0
0040D76A    jmp         main+55h (0040d775)
0040D76C    mov         ecx,dword ptr [ebp-10h]
0040D76F    add         ecx,1
```

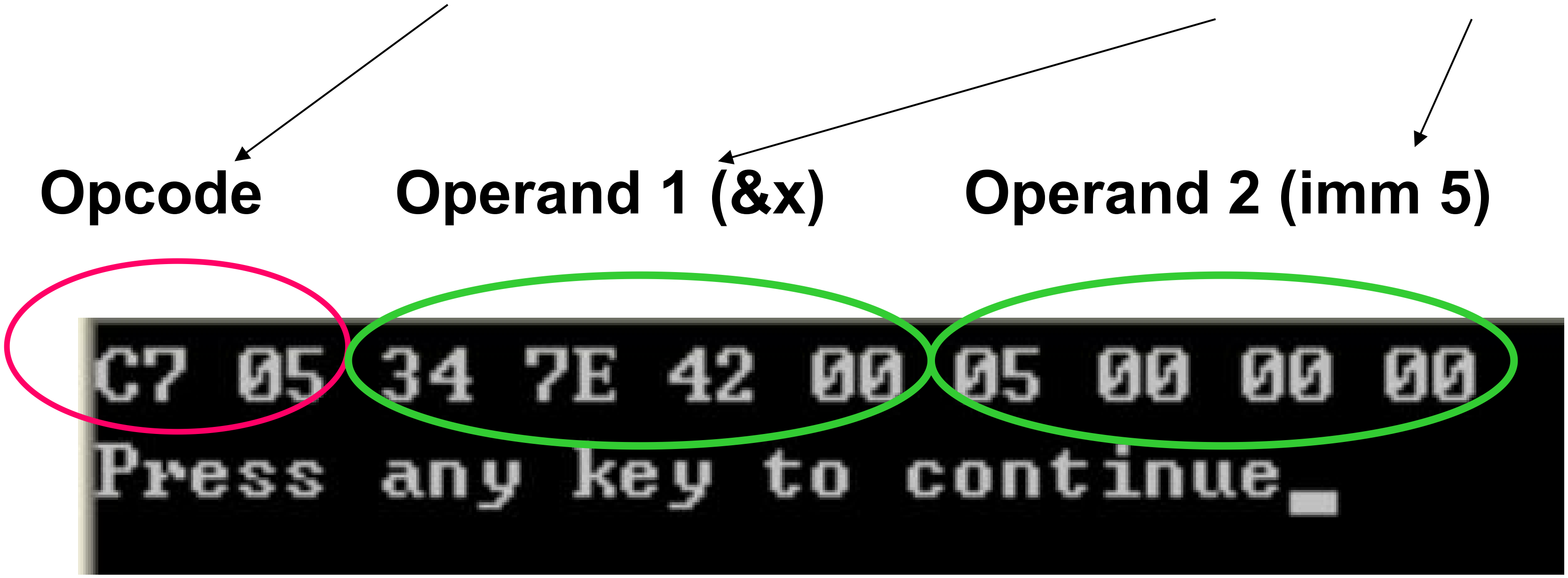
3.0 Brief Introduction (7/15)

- Instruction
  - CPU related
  - Opcode：操作码
    - The operations/Opcode/操作码 that the CPU can support
  - Operand：操作数/操作数的地址
    - Also define the method in which Operand are accessed(访问操作数的模式,register /mem/imm)

Opcode	Operand

3.0 Brief Introduction (8/15)

- **0040D746 mov dword ptr [_x (00427e34)],5**



- The opcode instructs the CPU to move the four-byte literal in the last 4 bytes to the address followed the opcode.
  - Immediate Data “5” stored in instruction code directly.
  - Global data “x” stored in a certain section of memory.

3.0 Brief Introduction (9/15)

- Two stages of each Instruction Execution
  - CPU executes instructions one by one

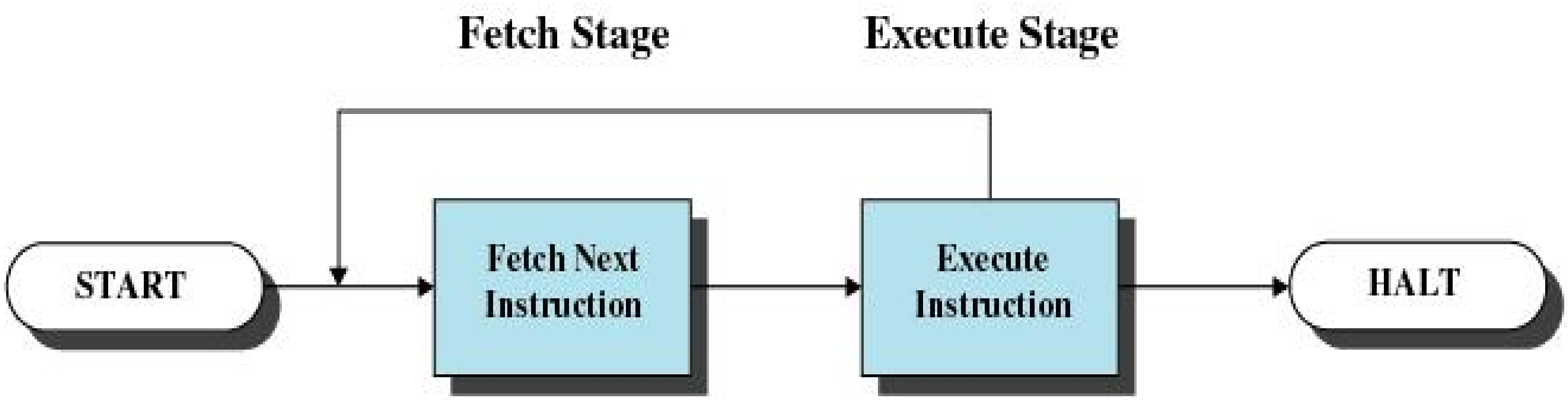
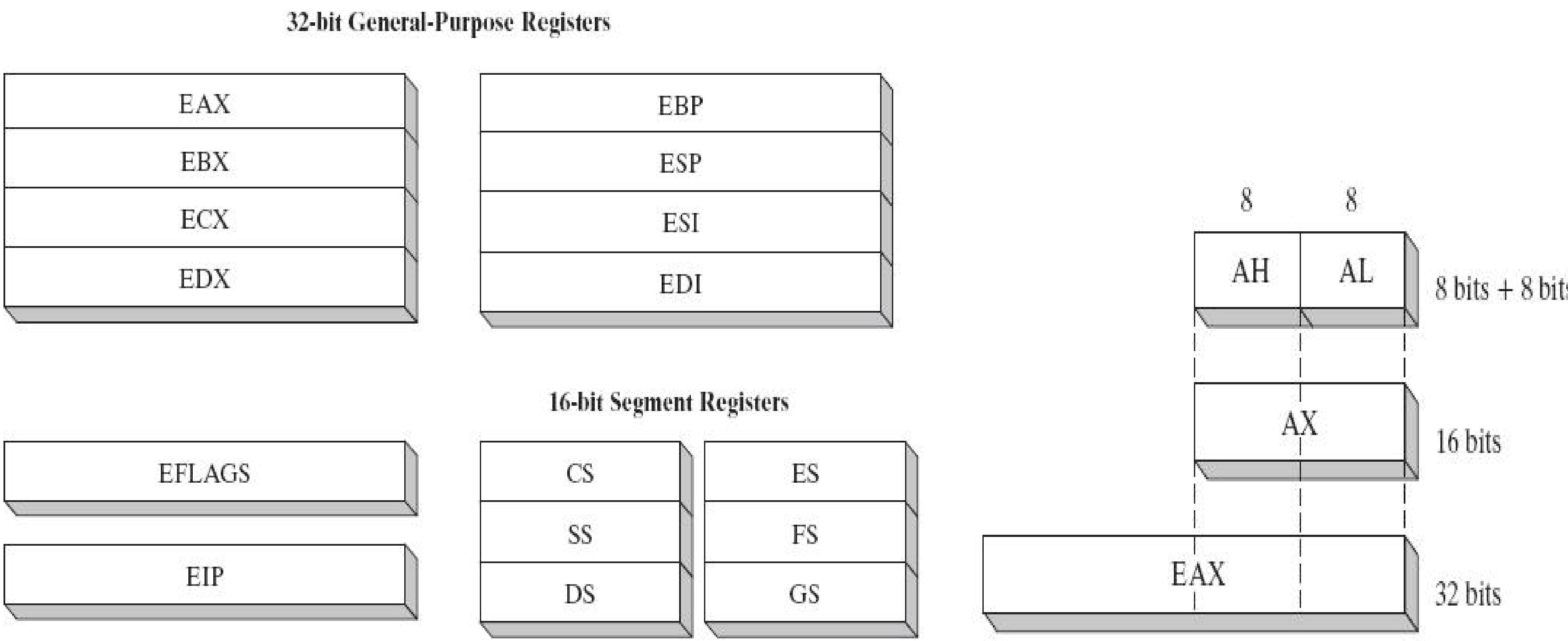


Figure 1.2 Basic Instruction Cycle

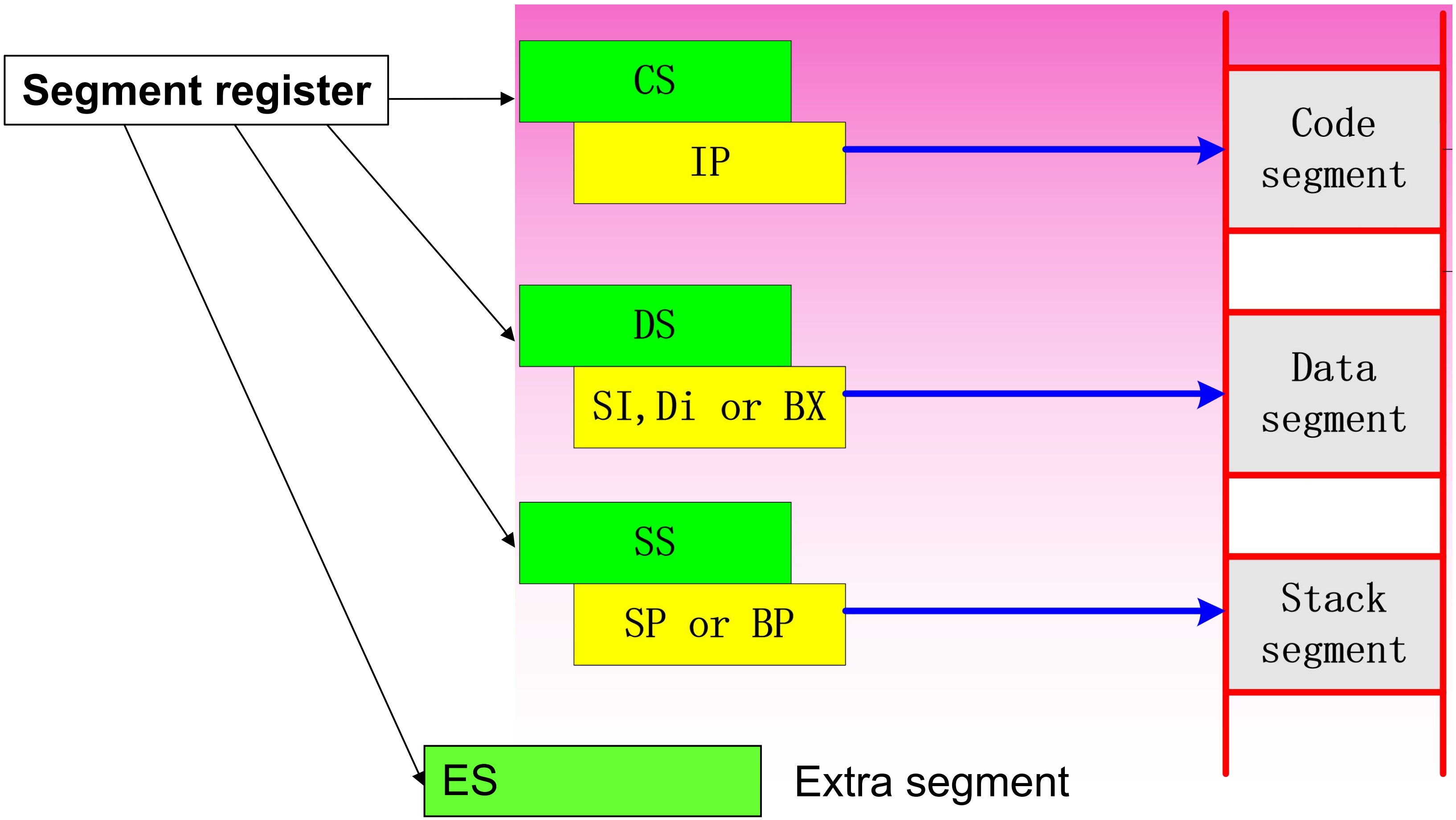
3.0 Brief Introduction (10/15)

- 8086 Register/ Real Mode



In real mode

3.0 Brief Introduction (11/15)



3.0 Brief Introduction (12/15)

- Selected "x86" processors

CPU	Vendor	Year	Bits	Note
8086	Intel	1978	16	
80386	Intel	1985	32	32-bit, virtual memory
Pentium	Intel	1993	32	
Pentium Pro	Intel	1995	32	
Pentium III	Intel	1999	32	
Pentium 4	Intel	2004	32	
Opteron	AMD	2003	64	First 64-bit x86 ("AMD64")

- Subsequent Intel CPUs adopted the AMD64 architecture (calling it "EM64T")
  - Often called "x86-64" or just "x64"

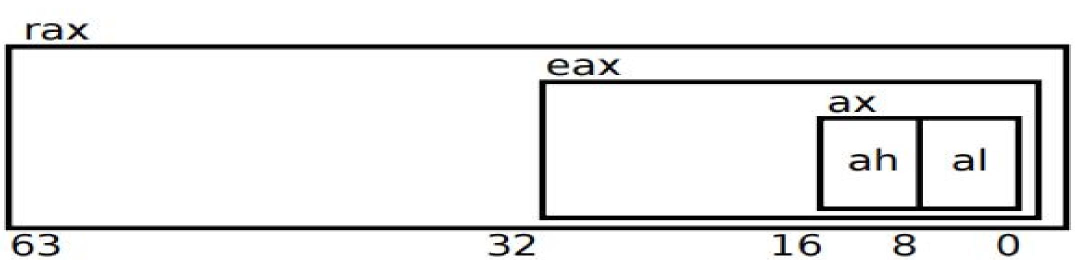
3.0 Brief Introduction (13/15)

- x86-64 registers

Register(s)	Note
rip	Instruction pointer
rax	Function return value
rdi, rsi	
rbx, rcx, rdx	
rsp, rbp	Stack pointer, frame pointer
r8, r9, ..., r15	

3.0 Brief Introduction (14/15)

- “Sub”-registers
  - For historical reasons (evolution of x86 architecture from 16 to 64 bits), each data register is divided into
    - Low byte
    - Second lowest byte
    - Lowest 2 bytes (16 bits)
    - Lowest 4 bytes (32 bits)
  - E.g., rax register has al, ah, ax, eax:



3.0 Brief Introduction (15/15)

- Summary of brief introduction
  - C to ASM
    - Code in ASM
  - ASM to Machine code
    - Machine code executed by CPU

Unit 3. Representation of Code

- 3.0 Brief Introduction
- <u>3.1 x86 Instructions</u>
- 3.2 x86 Assembly Language Programming
- 3.3 Disassembly in IDE

3.1.2 8086 Instructions

- <u>3.1.1 Operand-addressing Mode</u>
- 3.1.2 Instruction Set

3.1.1 Operand-addressing Mode (1/8)

- Operand Addressing：
  - What determines which data item will be fetched at each step of a instruction's execution?
    - In c : use a variable
    - In machine code : operand-addressing mode

- Operand-addressing Mode Classified
  - Register
  - Immediate
  - Memory

3.1.1 Operand-addressing Mode (2/8)

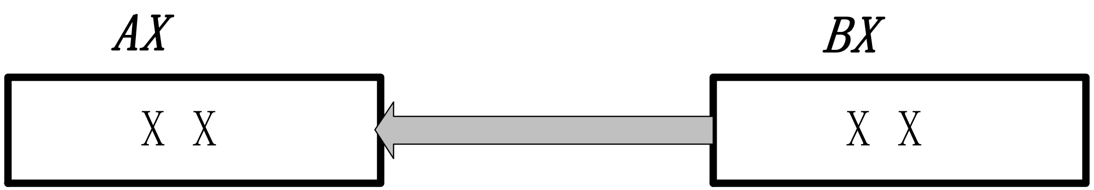
- Data-addressing Mode Classified in X86
  - Register (寄存器寻址)
  - Immediate(立即寻址)
  - Memory(存储器寻址)
    - direct addressing (直接寻址)
    - register indirect addressing(寄存器间接寻址)
    - base-plus-index addressing(基址加变址寻址)
    - Register relative Addressing(寄存器相对寻址方式)
    - base relative-plus-index addressing(相对基址变址寻址)
    - scaled-index addressing(比例变址寻址)
  - I/O addressing
    - direct i/o port addressing   in ax,80h；
    - ...

3.1.1 Operand-addressing Mode (3/8)

Type	Instruction	Source	Address Generation	Destination
Register	MOV AX,BX	Register BX		Register AX
Immediate	MOV CH,3AH	Data 3AH		Register CH
Direct	MOV [1234H],AX	Register AX	DS × 10H + DISP 10000H + 1234H	Memory address 11234H
Register indirect	MOV [BX],CL	Register CL	DS × 10H + BX 10000H + 0300H	Memory address 10300H
Base-plus-index	MOV [BX+SI],BP	Register SP	DS × 10H + BX + SI 10000H + 0300H + 0200H	Memory address 10500H
Register relative	MOV CL,[BX+4]	Memory address 10304H	DS × 10H + BX + 4 10000H + 0300H + 4	Register CL
Base relative-plus-index	MOV ARRAY[BX+SI],DX	Register DX	DS × 10H + ARRAY + BX + SI 10000H + 1000H + 0300H + 0200H	Memory address 11500H
Scaled index	MOV [EBX+2 × ESI],AX	Register AX	DS × 10H + EBX + 2 × ESI 10000H + 00000300H + 00000400H	Memory address 10700H

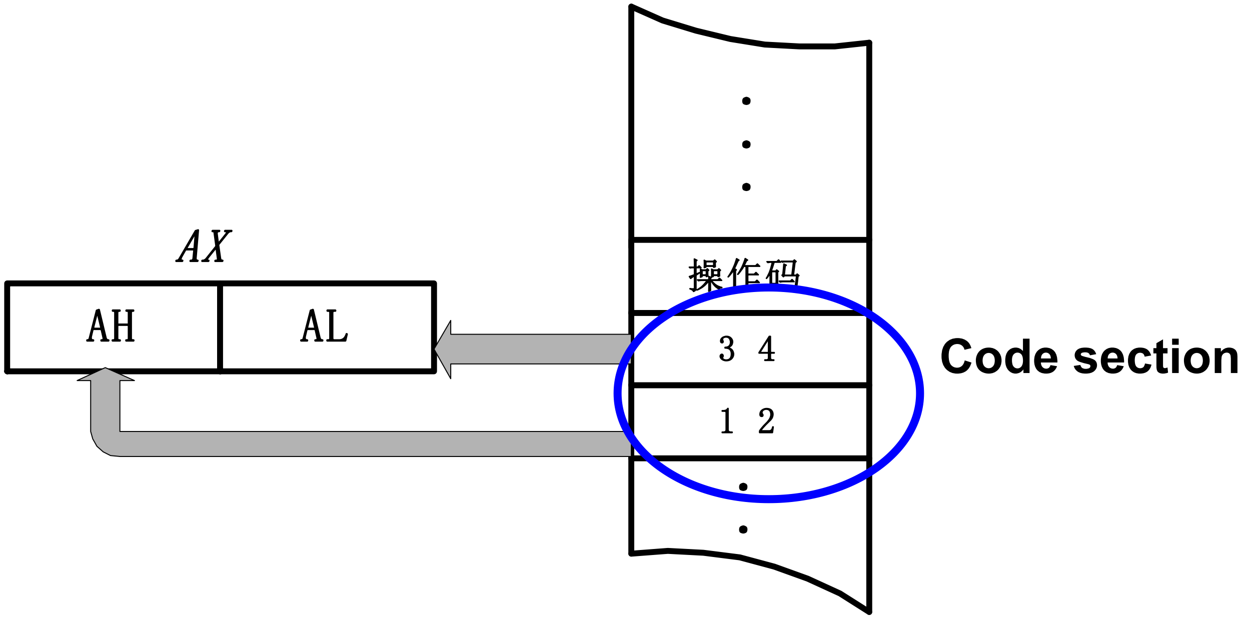
3.1.1 Operand-addressing Mode (4/8)

- Register Addressing
  - All operands are stored in general purpose registers.
    - 16 bits :AX， BX， CX， DX， SI， DL， SP or BP；
    - 8 bits: AH， AL， BH， BL， CH， CL， DH or DL。
  - MOV AX， BX   ； copy data from BX to AX



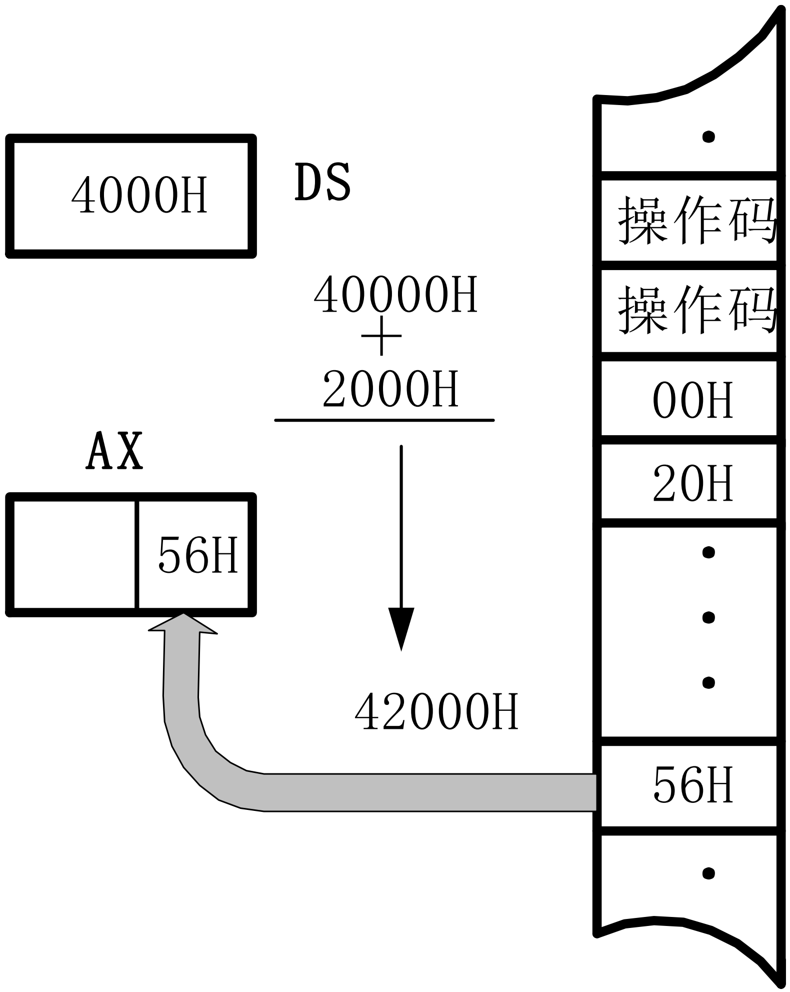
3.1.1 Operand-addressing Mode (5/8)

- Immediate addressing
  - MOV AX，1234H



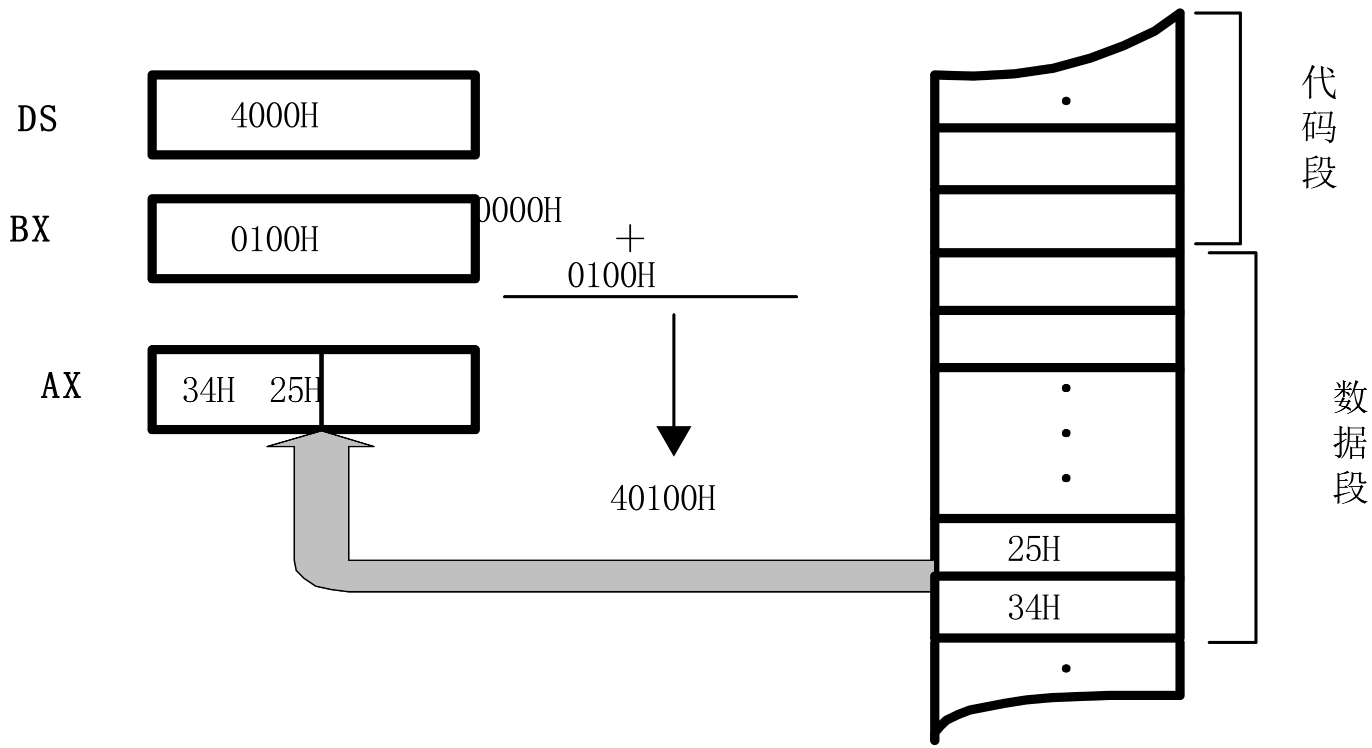
3.1.1 Operand-addressing Mode (6/8)

- Direct Memory Addressing
  - MOV AL， DS:[2000H] ;
  - 1. linear address = DS shifts towards left for 4 bits + offset
  - 2. One byte data in that address will be copy to AL



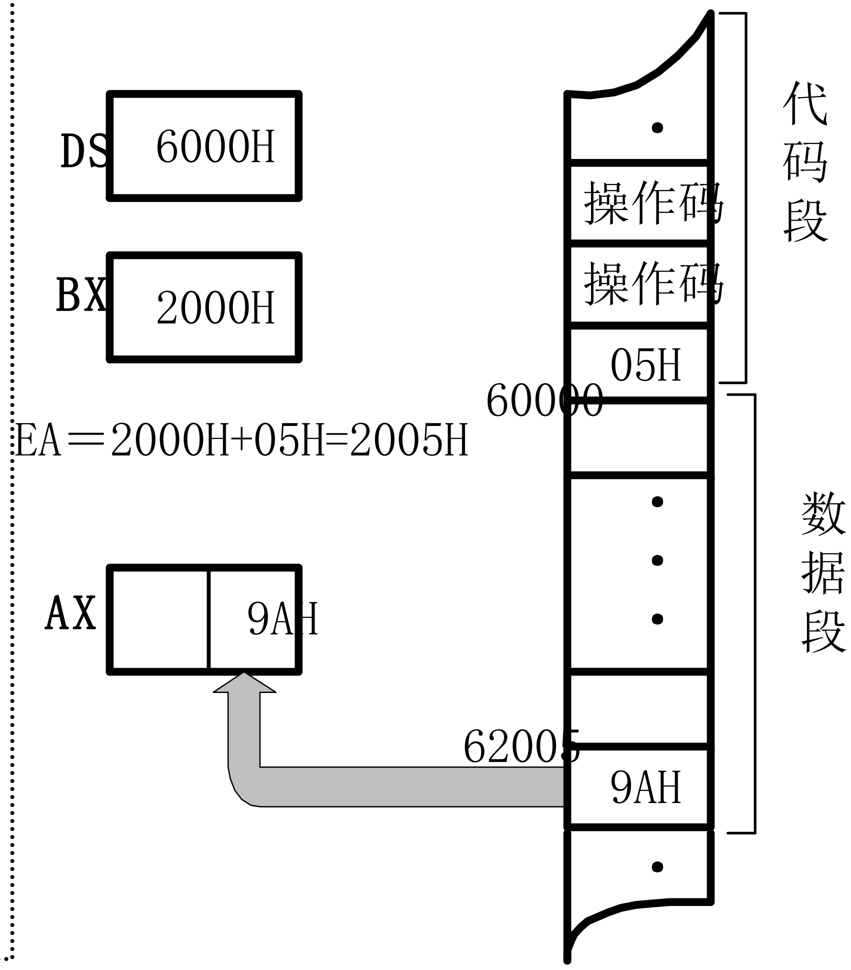
3.1.1 Operand-addressing Mode (7/8)

- Register Indirect addressing
  - MOV AX，[BX]；
  - linear address = DS shifts towards left for 4 bits + effective address in BX



3.1.1 Operand-addressing Mode (8/8)

- register relative addressing
  - MOV AL，[BX+5]
  - linear address = DS shifts towards left for 4 bits + BX+5



3.1 X86 Instructions

- 3.1 8086 Instructions
  - 3.1.1 Operand-addressing Mode
  - <u>3.1.2 Instruction Set</u>

3.1.2 Instruction Set(1/12)

- 8086 Instruction Set: classified according to their functions
  - 1) Data transforming/数据传送类
  - 2) Arithmetic operation/算术运算类
  - 3) Bits and logic operation/位与逻辑运算类
  - 4) String operation/字符串处理类
  - 5) Branch and loop/控制转移类
  - 6) CPU control/处理器控制类

3.1.2 Instruction Set(2/12)

- 8086 Instruction Set: Description about an instruction
  - A. Function:指令的基本功能
  - B. Affection: how do the instruction affect the Flag Register/ PSW
  - C. Data-addressing mode:
  - D. Limitation: which register can or can’t be used

3.1.2 Instruction Set(3/12)

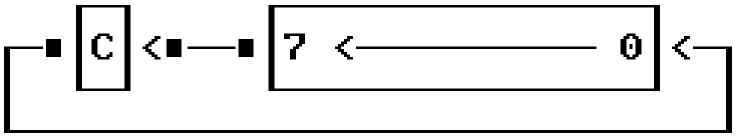
- 1) Data transforming/数据传送类
  - MOV (**Mov**e)
  - PUSH (**Push** data onto stack), POP (**Pop** data from stack)
    - PUSH AX POP BX
  - XCHG (E**xch**an**g**e data)
  - XLAT (Table look-up translation)
  - LEA (**L**oad **E**ffective **A**ddress)
  - LDS (**L**oad pointer using **DS**),LES (**L**oad **ES** with pointer)
  - LAHF (**L**oad **f**lags into **AH** register), SAHF (**S**tore **AH** into **f**lags)
  - PUSHF (**Push** **f**lags onto stack), POPF (**Pop** data into **f**lags register)
  - IN (**In**put from port), OUT (**Out**put to port)

3.1.2 Instruction Set(4/12)

- 2) Arithmetic operation/算术运算类
  - ADD (Add), ADC (**Ad**d with **c**arry), INC (**Inc**rement by 1)
  - SUB (**Sub**traction), SBB (**S**u**b**traction with **b**orrow), DEC (**Dec**rement by 1), NEG (Two's complement **neg**ation)
  - CMP (**Comp**are operands)
  - MUL (Unsigned **mul**tiply), IMUL (S**i**gned **mul**tiply)
  - DIV (Unsigned **div**ide), IDIV (S**i**gned **div**ide)
  - CBW (**C**onvert **b**yte to **w**ord), CWD (**C**onvert **w**ord to **d**oubleword)
  - DAA (**D**ecimal **a**djust AL after **a**ddition), DAS (**D**ecimal **a**djust AL after **s**ubtraction), AAA (**A**SCII **a**djust AL after **a**ddition), AAS (**A**SCII **a**djust AL after **s**ubtraction), AAM (**A**SCII **a**djust AX after **m**ultiplication), AAD (**A**SCII **a**djust AX before **d**ivision)
    - used with unpacked binary-coded decimal

3.1.2 Instruction Set(5/12)

- 3) Bits and logic operation/位与逻辑运算类
  - NOT, AND, OR, XOR,
  - TEST (Logical compare (AND))
  - SHL, SHR
  - SAL, SAR
  - ROL (Rotate left), ROR (Rotate right)
  - RCL (Rotate through carray left), RCR (Rotate through carry right)



3.1.2 Instruction Set(6/12)

- 4) String operation/字符串处理类
  - MOVSB (**Mov**e **b**yte from **s**tring to string)
  - STOSB (**Sto**re **byte** in **string**), LODSB (**Lo**a**d** **b**yte)
  - CMPSB (**Comp**are bytes in memory)
  - SCASB (**C**ompa**r**e **byte** **s**tring)

    - Prefix: REPxx (**Rep**eat CMPS/MOVS/SCAS/STOS)/REPZ

3.1.2 Instruction Set(7/12)

```
0040D729   lea        edi,[ebp-50h];edi = ebp-50h
0040D72C   mov        ecx,14h;loop counter
0040D731   mov        eax,0CCCCCCCCh;tag
0040D736   rep stos   dword ptr [edi]
```

; write the value of eax to memory location addressed by edi for ecx times, edi automatically increases 4 in each iteration.

3.1.2 Instruction Set(8/12)

- 5) Branch and loop/控制转移类
  - JMP (**Jump**)
  - Jxx (**J**ump if condition)/JE,JNE; JS,JNS; JO,JNO; JC,JNC; JP,JNP;JB,JNB; JA,JNA (unsigned); JG,JNG,JL,JNL (signed)
  - LOOP, LOOPx (**Loop** control)/LOOPZ,LOOPNZ,JCXZ
  - CALL (**Call** procedure),RET (**Ret**urn from procedure)
  - INT (Call DOS **int**errupt), IRET (**Ret**urn from **i**nterrupt)

3.1.2 Instruction Set(9/12)

- Branch Instructions Example

```
if (a == 1) {
        a = 5;
}
```

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed.

```
5:         if(a==1)
004010BF 83 7D FC 01             cmp      dword ptr [ebp-4],1
004010C3 75 07                   jne      main+2Ch (004010cc)
6:             a=5;
004010C5 C7 45 FC 05 00 00 00    mov      dword ptr [ebp-4],5
7:
8:         return 0;
004010CC 33 C0                   xor      eax,eax
```

JNE Jump if not equal. Test ZF=0.

3.1.2 Instruction Set(10/12)

```
if (a == 4) {  a = 5;  }
else {  a = 4;  }
```

Jump Instruction

```
5:          if (a == 4) {
004010BF 83 7D FC 04             cmp          dword ptr [ebp-4],4
004010C3 75 09                   jne          main+2Eh (004010ce)
6:              a = 5;
004010C5 C7 45 FC 05 00 00 00 mov          dword ptr [ebp-4],5
7:          }
8:          else {
004010CC EB 07                   jmp          main+35h (004010d5)
9:              a = 4;
004010CE C7 45 FC 04 00 00 00 mov          dword ptr [ebp-4],4
10:         }
11:
12:         return 0;
004010D5 33 C0                   xor          eax,eax
```

EIP 004010CC

EIP 004010D5

3.1.2 Instruction Set(11/12)

- Loops are constructed by compilers in terms of jumps and branches

while (a > 4)
{ a = a - 1; }

```
5:          while (a > 4) {
004010BF 83 7D FC 04         cmp         dword ptr [ebp-4],4
004010C3 7E 0B               jle         main+30h (004010d0)
6:          a = a - 1;
004010C5 8B 45 FC            mov         eax,dword ptr [ebp-4]
004010C8 83 E8 01            sub         eax,1
004010CB 89 45 FC            mov         dword ptr [ebp-4],eax
7:          }
004010CE EB EF               jmp         main+1Fh (004010bf)
8:
9:          return 0;
004010D0 33 C0               xor         eax,eax
```

3.1.2 Instruction Set(12/12)

- 6) CPU control/处理器控制类
  - CLC (**Cl**ear **c**arry flag), STC (**S**e**t** **c**arry flag), CMC (**Com**plement **c**arry flag)
  - CLD(**Cl**ear **d**irection flag),STD (**S**e**t** **d**irection flag)
  - CLI (**Cl**ear **i**nterrupt flag),STI (**S**e**t** **i**nterrupt flag)
  - NOP (**N**o **op**eration)
  - HLT (Enter **halt** state)
    - halts the CPU until the next external interrupt is fired.
  - WAIT (**Wait** until not busy)
  - LOCK (Assert BUS **LOCK**# signal) (for multiprocessing)
  - ESC (Escape)

Unit 3. Representation of Code

- 3.0 Intro
- 3.1 X86 Instructions
- <u>3.2 Assembly Language Programming</u>
- 3.3 Disassembly in IDE

3.2 Assembly Language Programming(1/10)

- Two styles of assembly language
  - GAS(GNU Assembly)/AT&T
    - The one on the CSAPP book
  - Intel/MASM
    - Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax.

3.2 Assembly Language Programming(2/10)

- Types of Statements in Assembly Language
  - 1. Executable Instructions (可执行指令)
    - Generate machine code for the processor to execute at runtime
    - Instructions tell the processor what to do
  - 2. Macros (宏指令)
    - Translated by the assembler into real instructions
    - Simplify the programmer task
  - 3. Assembler Directives (伪指令)
    - Provide information to the assembler while translating a program
    - Used to define segments, allocate memory variables, etc.
    - Non-executable: directives are not part of the instruction set

3.2 Assembly Language Programming(3/10)

- Example: Adding and Subtracting Integers

```
TITLE Add and Subtract              (AddSub.asm)
; This program adds and subtracts 32-bit integers.
.686
.MODEL FLAT, STDCALL//stdcall为API调用时右边的参数先入栈
.STACK
INCLUDE Irvine32.inc
.DATA
.CODE
main PROC
    mov eax,10000h                  ; EAX = 10000h
    add eax,40000h                  ; EAX = 50000h
    sub eax,20000h                  ; EAX = 30000h
    call DumpRegs                   ; display registers
    exit                            ; macro defined in Irvine32.inc
main ENDP
END main
```

3.2 Assembly Language Programming(4/10)

Procedure **DumpRegs** is defined in **Irvine32.lib** library

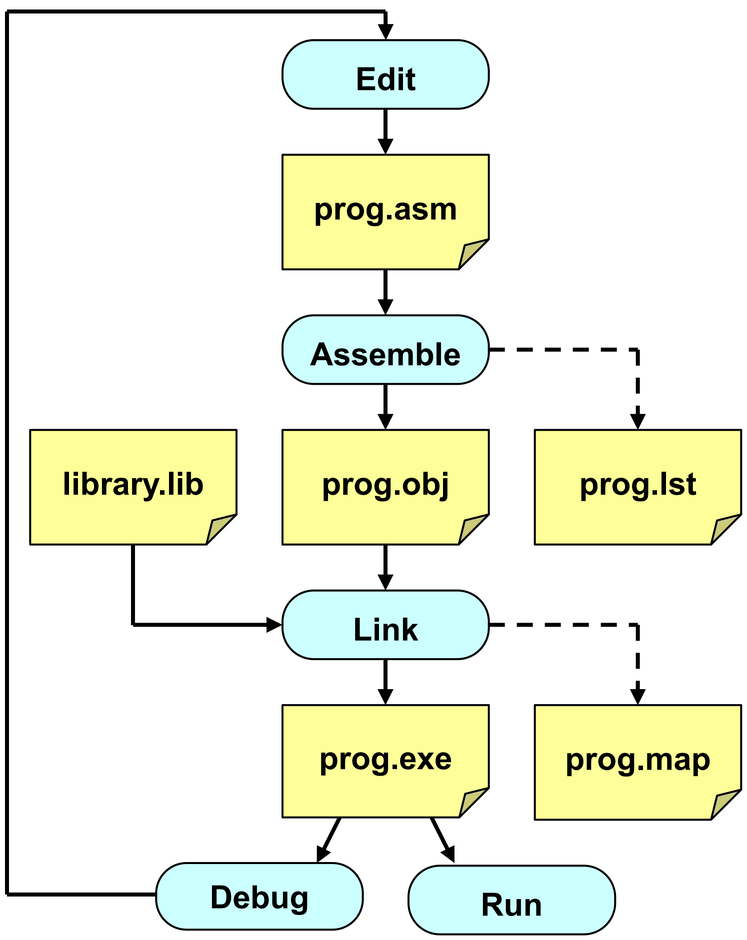
It produces the following console output,

showing registers and flags:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

3.2 Assembly Language Programming(5/10)

- Editor
  - Write new (**.asm**) programs
  - Make changes to existing ones
- Assembler: **ML.exe** program
  - Translate (**.asm**) file into object (**.obj**) file in machine language
  - Can produce a listing (**.lst**) file that shows the work of assembler
- Linker: **LINK32.exe** program
  - Combine object (**.obj**) files with link library (**.lib**) files
  - Produce executable (**.exe**) file
  - Can produce optional (**.map**) file



3.2 Assembly Language Programming(6/10)

- Listing File :**Use it to see how your program is assembled**
- Contains
  - Source code
  - Object code
  - Relative addresses
  - Segment names
  - Symbols
    - Variables
    - Procedures
    - Constants

Object & source code in a listing file

```
00000000                        .code
00000000                        main PROC
00000000  B8 00060000               mov eax, 60000h
00000005  05 00080000               add eax, 80000h
0000000A  2D 00020000               sub eax, 20000h

0000000F  6A 00                     push 0
00000011  E8 00000000  E            call ExitProcess
00000016                        main ENDP
                                END main
```

Relative Addresses | **object code (hexadecimal)** | **source code**

3.2 Assembly Language Programming(7/10)

- Map File

Start	Stop	Length	Name	Class
00000H	00010H	00011H	DATA	DATA
00020H	0011FH	00100H	SS_SEG	STACK
00120H	0012FH	00010H	CODE	CODE

Program entry point at 0012:0000

3.2 Assembly Language Programming(8/10)

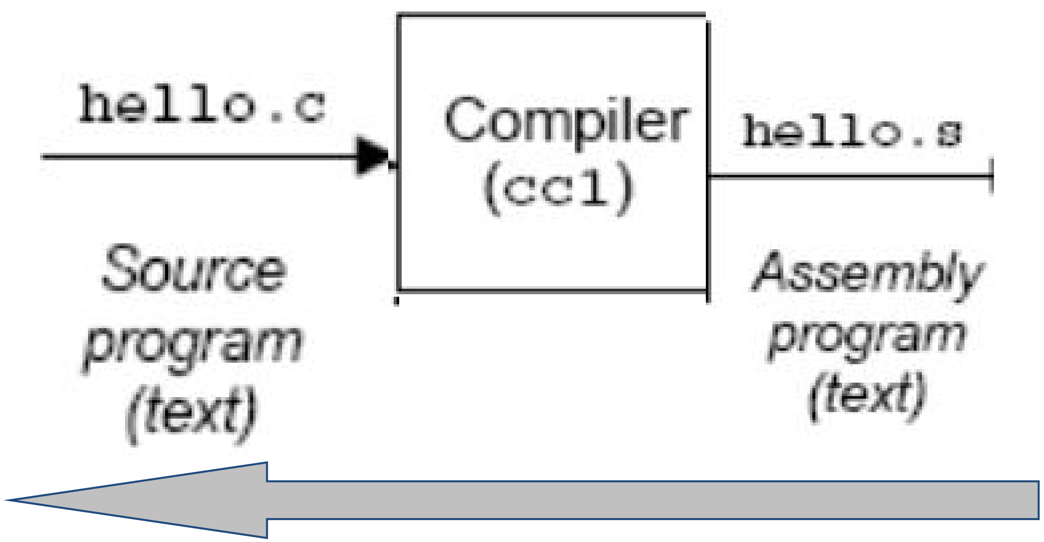
- You should learn, if you want to master Assembly Language
  - Instruction Set, as well as data-addressing mode
  - Pseudo-operation/directive
  - Others, such as macro
  - The development tools

3.2 Assembly Language Programming(9/10)

- Lab 3 Preview
  - Practice 1: observe C program by IDE disassembly
  - Practice 2: program a simple x86 BootLoader

3.2 Assembly Language Programming(10/10)

- In this course, we will not focus on Assembly Language programming
  - We will pay more attention to Disassembly/反汇编



Unit 3. Representation of Code

- 3.0 Brief Introduction
- 3.1 X86 Instructions
- 3.2 Assembly Language Programming
- <u>3.3 Disassembly in IDE</u>

3.3 Disassembly in IDE(1/3)

- Disassembly
  - Useful tool for examining object code
  - Produces approximate rendition of assembly code
  - Analyzes bit pattern of series of instructions
  - Can be run on either executable or .o file
    - Debug and Release version

3.3 Disassembly in IDE(2/3)

- In Code::Blocks, we can Disassembly
  - By using the Disassembly Windows when debugging

3.3 Disassembly in IDE(3/3)

- Register Allocation
  - The compiler would like to be able to allocate a register for every variable in the source program, because every variable that lives in a register speeds up the program's execution by some amount
  - CPU has a few registers， when and how to use each register for which variable.
  - Who do it
    - Assembly level： programmer him/herself
    - C level： Compiler