Intel Assembly

Intel Assembly

Goal: to gain a knowledge of Intel 32-bit assembly instructions

References:

- M. Pietrek, "Under the Hood: Just Enough Assembly Language to Get By"
  - MSJ Article, February 1998 www.microsoft.com/msj
  - Part II", MSJ Article, June 1998 www.microsoft.com/msj

- IA-32 Intel® Architecture Software Developer's Manual,
  - Volume 1: Basic Architecture
    www.intel.com/design/Pentium4/documentation.htm#manuals
  - Volume 2A: Instruction Set Reference A-M
    www.intel.com/design/pentium4/documentation.htm#manuals
  - Volume 2B: Instruction Set Reference N-Z
    www.intel.com/design/pentium4/documentation.htm#manuals

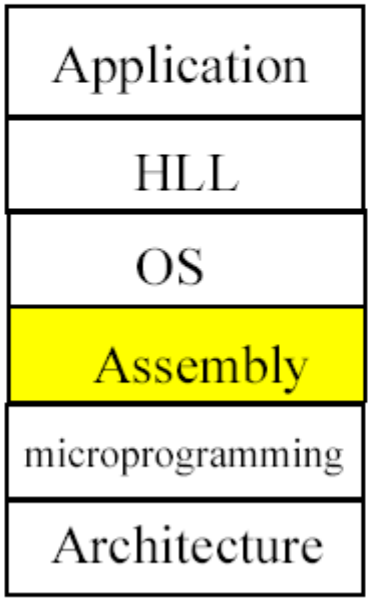
Assembly Programming

- Machine Language
  - binary
  - hexadecimal
  - machine code or object code
- Assembly Language
  - mnemonics
  - assembler
- High-Level Language
  - Pascal, Basic, C
  - compiler

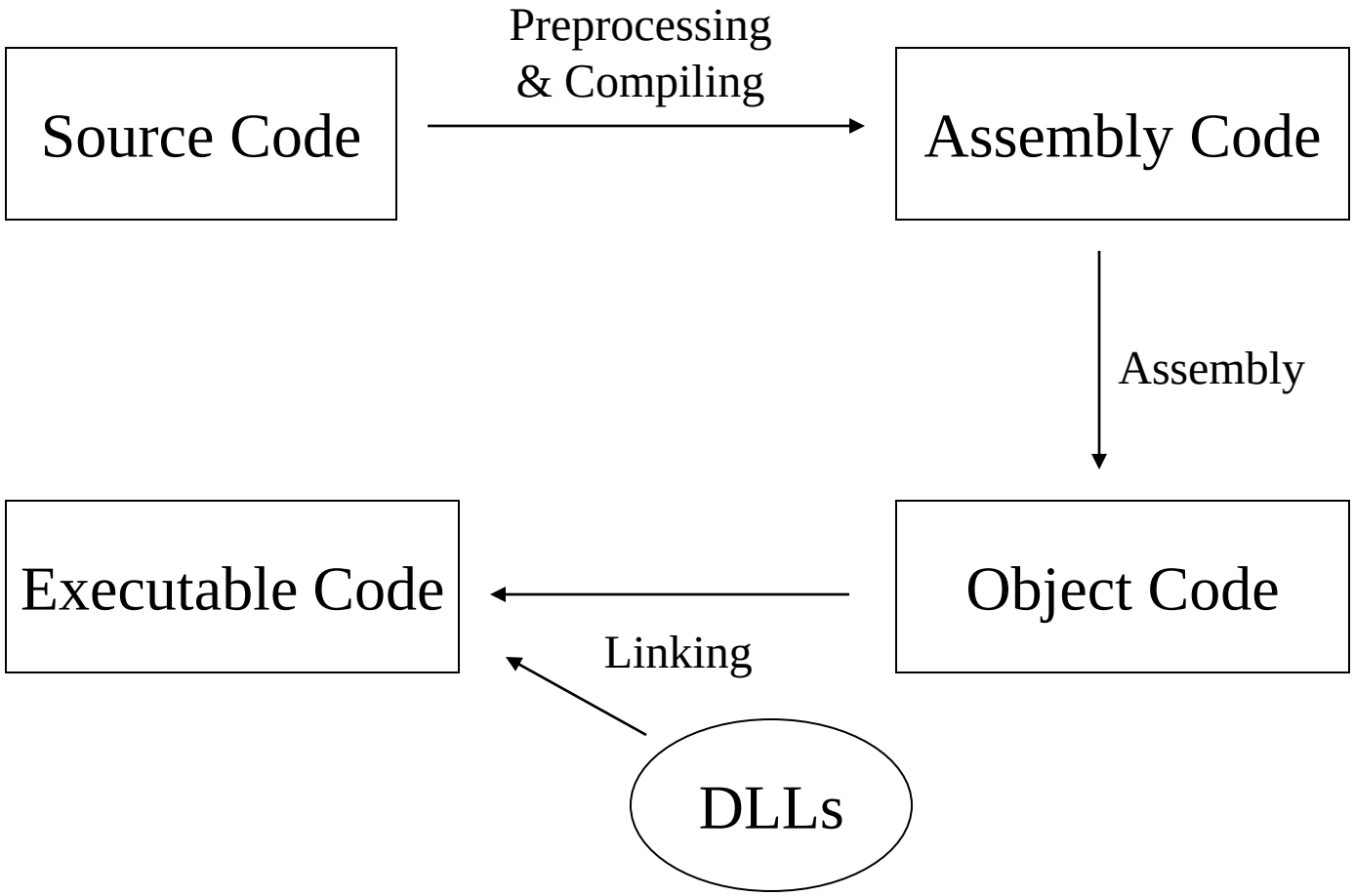
Assembly Language Programming

Motivations

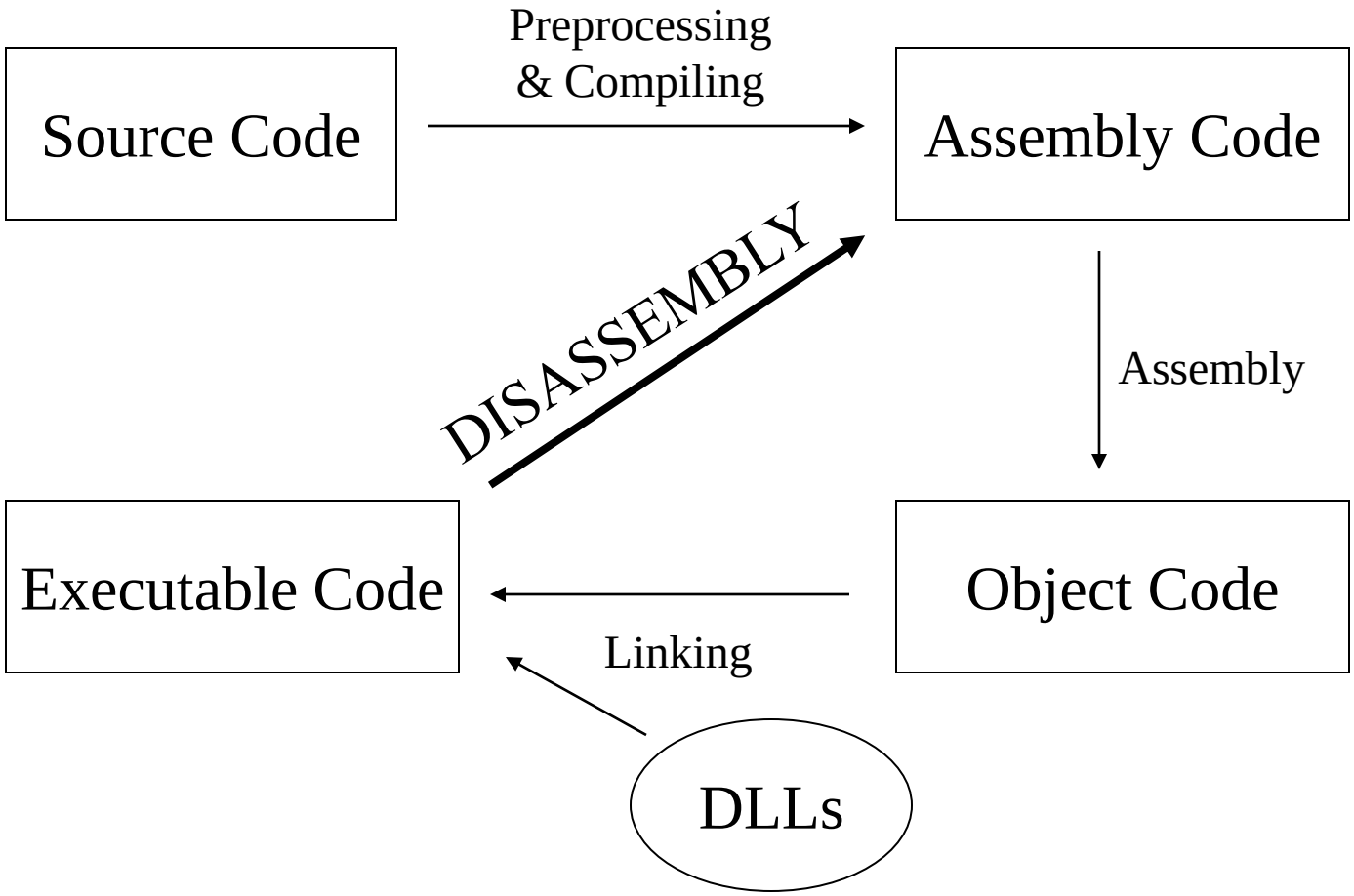
- Why do you learn assembly language?



What Does It Mean to Disassemble Code?



What Does It Mean to Disassemble Code?



Why is Disassembly Useful in Malware Analysis?

- It is not always desirable to execute malware: disassembly provides a static analysis.
- Disassembly enables an analyst to investigate all parts of the code, something that is not always possible in dynamic analysis.
- Using a disassembler and a debugger in combination creates synergy.

32-bit Instructions

- Instructions are represented in memory by a series of "opcode bytes."
- A variance in instruction size means that disassembly is position specific.
- Most instructions take zero, one, or two arguments:

```
instruction destination, source
```

For example: add eax, ebx

is equivalent to the expression eax = eax + ebx

General format

mnemonic	operand(s)	;comments

MOV	destination,source	;copy source operand to destination

Example:

```
MOV    DX,CX
```

Example 2:

```
MOV    CL,55H
MOV    DL,CL
MOV    AH,DL
MOV    AL,AH
MOV    BH,CL
MOV    CH,BH
```

AH	AL
BH	BL
CH	CL
DH	DL

Section B.1: The 8086 Instruction Set

Page 865

MOV Move

Flags: Unchanged

Format: MOV dest, source ; copy source to dest

Function: Copy a word or byte from a register, memory location, or immediate number to a register or memory location. Source and destination must be of the same size and cannot both be memory locations.

MOV instruction (16 bits)

```
MOV    CX,468H
MOV    AX,CX
MOV    DX,AX
MOV    BX,DX
MOV    DI,BX
MOV    SI,DI
MOV    DS,SI
MOV    BP,DI
```

What if ...

MOV AL,DX

Rule #1:

moving a value that is too large into a register will cause an error

```
MOV   BL,7F2H       ;Illegal: 7F2H is larger than 8 bits
MOV   AX,2FE456H    ;Illegal
```

Rule #2:

Data can be moved directly into nonsegment registers only

(Values cannot be loaded directly into any segment register. To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register.)

```
MOV   AX,2345H          MOV   DI,1400H
MOV   DS,AX             MOV   ES,DI
```

Rule #3:

If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

```
MOV BX, 5        BX =0005
                 BH = 00, BL = 05
```

ADD instruction

```
ADD     destination,source     ;ADD the source operand to the destination
```

```
MOV    AL,25H
MOV    BL,34H
ADD    AL,BL
```

You can do this,

```
MOV    DH,25H
ADD    DH,34H
```

- What is the corresponding C++ code ?
- The way C++ compiler implements '+' operation is fixed
- Assembly language has more flexibility
- Assembly language can tailor the code closer to the hardware

(That is where the efficiency comes from)

ADD Signed or Unsigned ADD

Flags: Affected: OF,SF,ZF,AF,PF,CF

Format: ADD dest, source ; dest=dest+source

Function: Adds source operand to destination operand and places the result in destination. Both source and destination operands must match (e.g.,both byte size or word size) and only one of them can be in memory.

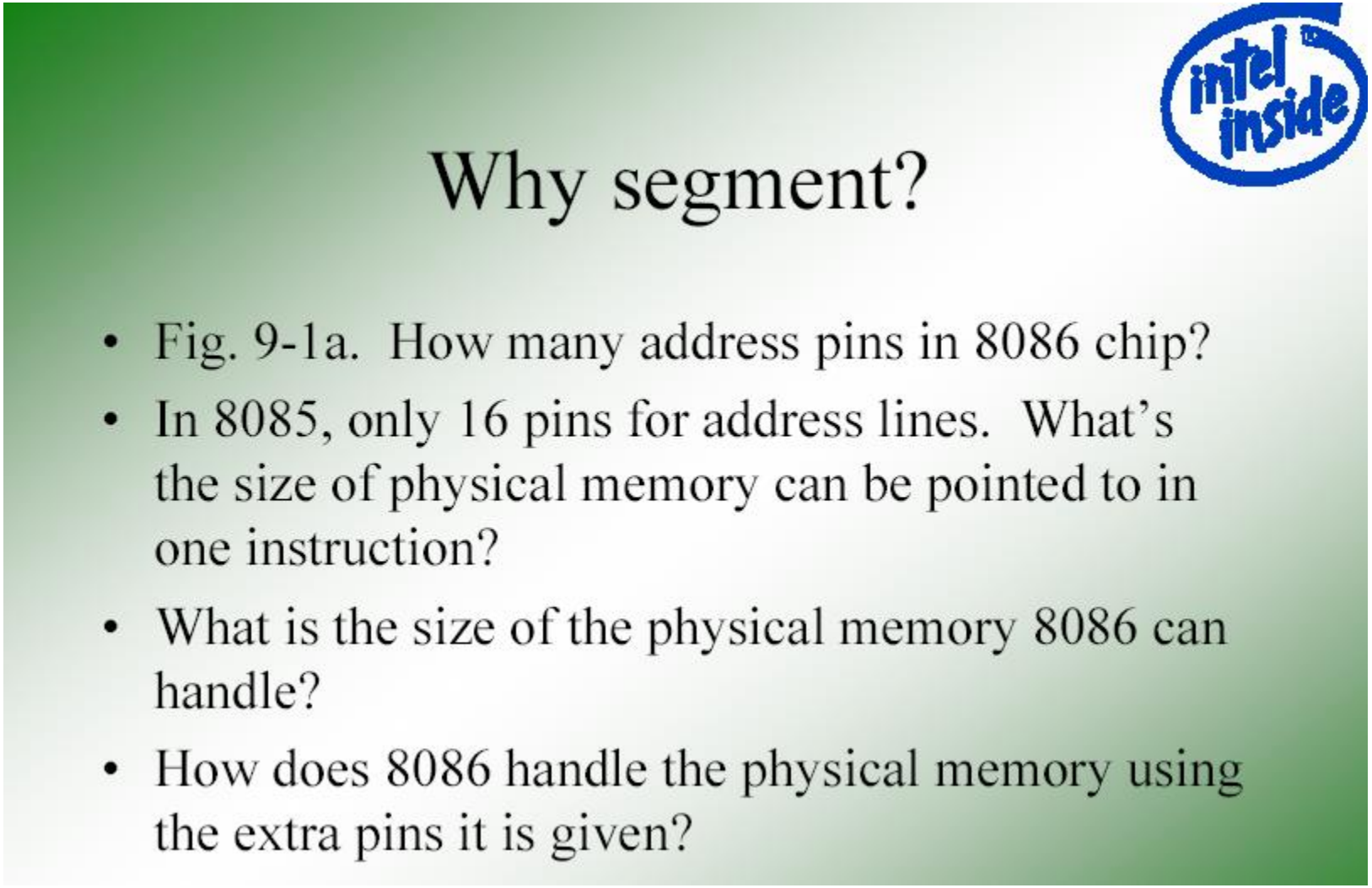
Program Segments

- A segment is an area of memory that includes up to 64K bytes
- Begins on an address evenly divisible by 16
- 8085 could address a max. of 64K bytes of physical memory
  - it has only 16 pins for the address lines (2^{16} = 64K)
- 8088/86 stayed compatible with 8085
  - Range of 1MB of memory, it has 20 address pins (2^{20} = 1 MB)
  - Can handle 64KB of code, 64KB of data, 64KB of stack
- A typical Assembly language program consist of three segments:
  - Code segments
  - Data segment
  - Stack segment

Program Segments…*a sample*

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
        .MODEL SMALL
        .STACK 64
        .DATA
DATA1           DB      52H
DATA2           DB      29H
SUM             DB      ?
        .CODE
MAIN            PROC    FAR         ;this is the program entry point
        MOV     AX,@DATA            ;load the data segment address
        MOV     DS,AX               ;assign value to DS
        MOV     AL,DATA1            ;get the first operand
        MOV     BL,DATA2            ;get the second operand
        ADD     AL,BL               ;add the operands
        MOV     SUM,AL  ;store the result in location SUM
        MOV     AH,4CH  ;set up to
        INT     21H                 ;return to DOS
MAIN            ENDP
        END     MAIN                ;this is the program exit point
```

Program Segments



Program Segments

Code segment

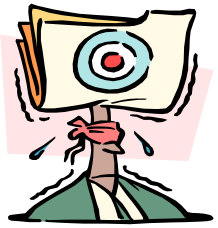
The 8086 fetches the instructions (opcodes and operands) from the code segments.

The 8086 address types:

- Physical address
- Offset address
- Logical address

- Physical address
  - 20-bit address that is actually put on the address pins of 8086
  - Decoded by the memory interfacing circuitry
  - A range of 00000H to FFFFFH
  - It is the actual physical location in RAM or ROM within 1 MB mem. range
- Offset address
  - A location within a 64KB segment range
  - A range of 0000H to FFFFH
- Logical address
  - consist of a segment value and an offset address

Program Segments…*example*



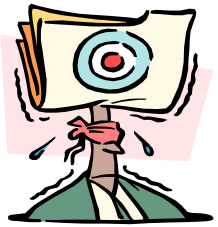
Define the addresses for the 8086 when it fetches the instructions (opcodes and operands) from the code segments.

- Logical address:
  - Consist of a **CS** (code segment) and an **IP** (instruction pointer)
    format is **CS:IP**

- Offset address
  - **IP** contains the offset address

- Physical address
  - generated by shifting the **CS** left one hex digit and then adding it to the **IP**
  - the resulting 20-bit address is called the physical address

Program Segments…*example*



Suppose we have:

CS	2500
IP	95F3

- Logical address:
  - Consist of a **CS** (code segment) and an **IP** (instruction pointer)
    format is **CS:IP** **2500:95F3H**

- Offset address
  - **IP** contains the offset address which is **95F3H**

- Physical address
  - generated by shifting the **CS** left one hex digit and then adding it to the **IP**
    25000 + 95F3 = 2E5F3H

Program Segments

Data segment

Data segment refers to an area of memory set aside for data

- Format DS:BX or DI or SI
- example:

  DS:0200 = 25

  DS:0201 = 12

  DS:0202 = 15

  DS:0203 = 1F

  DS:0204 = 2B

Program Segments

Data segment

Example:

Add 5 bytes of data: 25H, 12H, 15H, 1FH, 2BH

Not using data segment

```
MOV     AL,00H      ;clear AL
ADD     AL,25H      ;add 25H to AL
ADD     AL,12H
ADD     AL,15H
ADD     AL,1FH
ADD     AL,2BH
```

Program Segments

Data segment

Example:

Add 5 bytes of data: 25H, 12H, 15H, 1FH, 2BH

using data segment with a constant offset

Data location in memory:

DS:0200 = 25

DS:0201 = 12

DS:0202 = 15

DS:0203 = 1F

DS:0204 = 2B

Program:

```
MOV     AL,0
ADD     AL,[0200]
ADD     AL,[0201]
ADD     AL,[0202]
ADD     AL,[0203]
ADD     AL,[0204]
```

Program Segments

Data segment

Example:

Add 5 bytes of data: 25H, 12H, 15H, 1FH, 2BH

using data segment with an offset register

Program:

```
MOV     AL,0
MOV     BX,0200H
ADD     AL,[BX]
INC     BX              ;same as "ADD BX,1"
ADD     AL,[BX]
INC     BX
ADD     AL,[BX]
INC     BX
ADD     AL,[BX]
```

Endian conversion

- Little endian conversion:
  In the case of 16-bit data, the low byte goes to the low memory location and the high byte goes to the high memory address. (Intel, Digital VAX)

- Big endian conversion:
  The high byte goes to low address. (Motorola)

Example:

Suppose DS:6826 = 48, DS:6827 = 22,

Show the contents of register BX in the instruction **MOV BX,[6826]**

Little endian conversion: BL = 48H, and BH = 22H

Program Segments

Stack segment

Stack

A section of RAM memory used by the CPU to store information temporarily.

- Registers: SS (Stack Segment) and SP (stack Pointer)
- Operations: PUSH and POP
  - PUSH – the storing of a CPU register in the stack
  - POP – loading the contents of the stack back into the CPU
- Logical and offset address format: SS:SP

Flag Register

- Flag Register (status register)
  - 16-bit register
  - Conditional flags: CF, PF, AF, ZF, SF, OF
  - Control flags: TF, IF, DF

R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag

SF = sign flag
ZF = zero flag
AF = auxiliary carry flag
PF = parity flag
CF = carry flag

Flag Register and ADD instruction

- Flag Register that may be affected
  - Conditional flags: CF, PF, AF, ZF, SF, OF

Flow Control I

- **`JMP` *`location`***

  Transfers program control to a different point in the instruction stream without recording return information.

```
jmp eax
jmp 0x00934EE4
```

Flow Control II

- **CMP *value, value* / Jcc *location***

  The *compare* instruction compares two values, setting or clearing a variety of flags (e.g., ZF, SF, OF). Various *conditional jump* instructions use flags to branch accordingly.

```
cmp  eax, 4
je   40320020
```

```
cmp  [ebp+10h], eax
jne  40DC0020
```

Flow Control III

- **TEST *value, value* / Jcc *location***

  The *test* instruction does a logical AND of the two values. This sets the SF, ZF, and PF flags. Various *conditional jump* instructions use these flags to branch.

```
test  eax, eax
jnz   40DA0020
```

```
test  edx, 0056FCE2
jz    56DC0F20
```

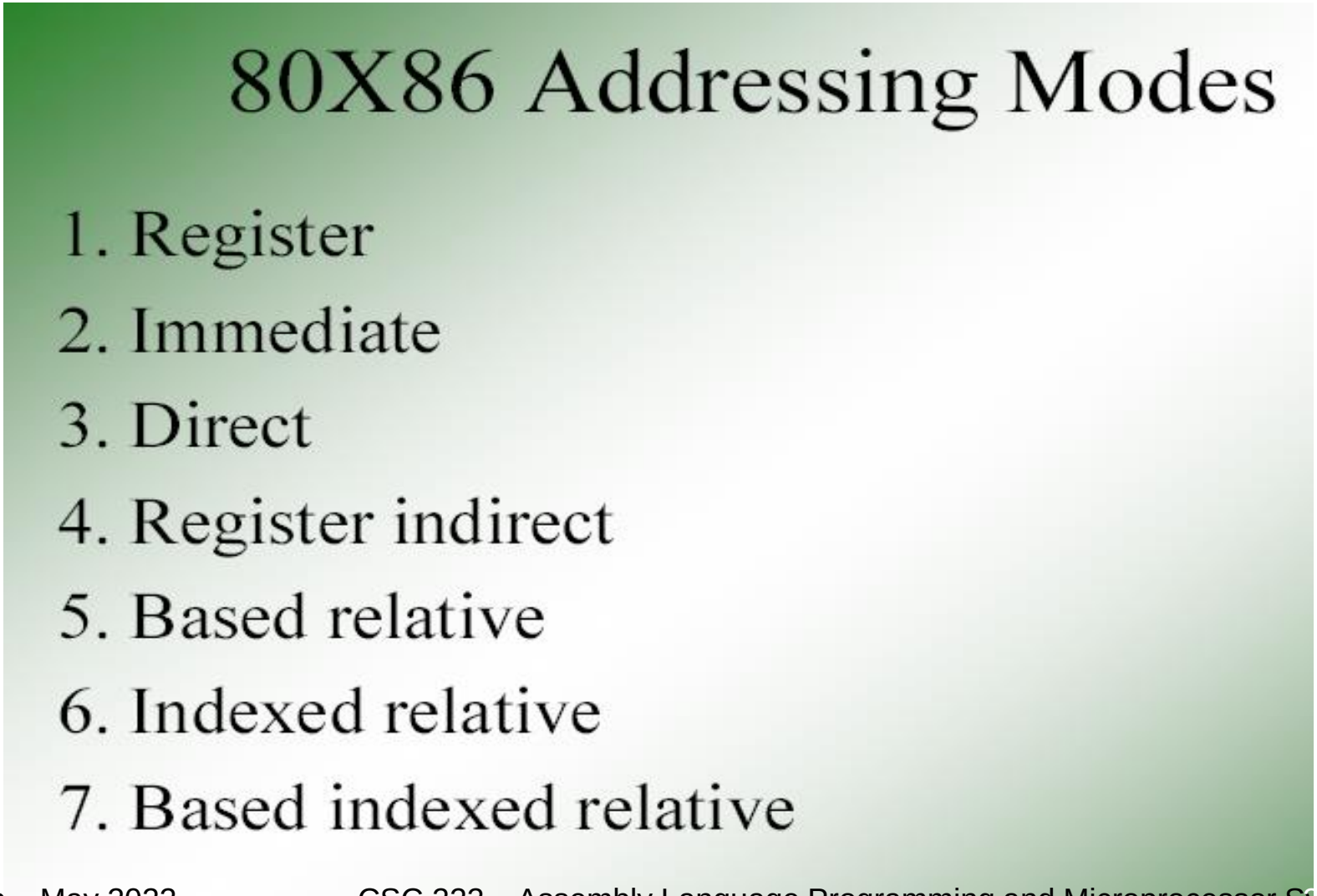
Looping using zero flag

- The zero flag is set (ZF=1), when the counter becomes zero (CX=0)

- *Example:* add 5 bytes of data

```
        MOV   CX,05        ;CX holds the loop count
        MOV   BX,0200H     ;BX holds the offset data address
        MOV   AL,00        ;initialize AL
ADD_LP: ADD   AL,[BX]      ;add the next byte to AL
        INC   BX           ;increment the data pointer
        DEC   CX           ;decrement the loop counter
        JNZ   ADD_LP       ;jump to next iteration if counter
                           ;not zero
```

Addressing Modes – Accessing operands (data) in various ways



Register Addressing Mode

- Registers are used to hold the data
- Memory is not accessed (hence is fast)
- Source and destination registers must match in size
- Exp:

```
MOV BX,DX
MOV ES,AX
ADD AL,BH
MOV CL,AX  (error)
```

Immediate Addressing Mode

- The source operand is a constant
- can be used to load information to any registers except the segment registers and flag registers (can be done indirectly)
- operands come immediately after the opcode
- EXP: MOV AX,2550H
  MOV CX,625
  MOV BL,40H
- How about MOV DS, 0123H ? (page 42)

Direct Addressing Mode

- The data is in memory
- The address of the operand is provided in the instruction directly
- The address is the offset address
- The physical address can be calculated using the content in the DS register
- Exp1: MOV DL,[2400] ;move contents of DS:2400H into DL
- Exp2: MOV AL,99H
  MOV [3518],AL

Register Indirect Addressing Mode

- The address of the memory location is in a register (SI, DI, or BX only)
- The physical address is calculated using the content of DS
- EXP:

```
MOV CL,[SI]   ;move contents of DS:SI into CL
MOV [DI],AH   ;move contents of AH into DS:DI
MOV [SI],AX   ; little endian is applied
              ;moves contents of AX into memory
              ;locations DS:SI and DS:SI +1
```

Based Relative Addressing Mode

- BX or BP can be used
- *effective address*: [BX]+disp, or [BP]+disp
- The physical address is calculated using DS for [BX]+disp, and SS for [BP]+disp
- Exp:

```
MOV CX,[BX]+10   ;move DS:BX+10 & DS:BX+10+1
                 ;into CX.  PA= DS(sl) +BX+10
MOV AL,[BP]+5    ;PA = SS (sl) + BP + 5
```

Index Relative Addressing Mode

- Similar to the based relative addressing mode
- DI and SI are used to hold the offset address
- DS is used for calculating physical address

EXP: MOV DX,[SI]+5 ;PA = DS (sl) + SI + 5

MOV CL,[DI]+20 ;PA = DS (sl) + DI + 20

Based Indexed Addressing Mode

- Combining the previous two modes
- One base register and one index register are used
- for physical address calculation, DS is used for BX; SS is used for BP
- EXP: MOV CL,[BX][DI]+8 ;PA=DS(sl)+BX+DI +8

  MOV AH,[BP][SI]+29 ;PA=SS(sl)+BP+SI +29

Assembly Language Programming

Assembly Programming

- Assembly Language instruction consist of four fields

[label:] mnemonic [operands] [;comment]

  - Labels
    - See rules
  - mnemonic, operands
    - MOV AX, 6764
  - comment
    - ; this is a sample program

Model Definition

MODEL directive –selects the size of the memory model

- **MODEL MEDIUM**
  - Data must fit into 64KB
  - Code can exceed 64KB
- **MODEL COMPACT**
  - Data can exceed 64KB
  - Code cannot exceed 64KB
- **MODEL LARGE**
  - Data can exceed 64KB (but no single set of data should exceed 64KB)
  - Code can exceed 64KB
- **MODEL HUGE**
  - Data can exceed 64KB (data items i.e. arrays can exceed 64KB)
  - Code can exceed 64KB
- **MODEL TINY**
  - Data must fit into 64KB
  - Code must fit into 64KB
  - Used with COM files

Segments

Segment definition:

The 80x86 CPU has four segment registers: CS, DS, SS, ES

Segments of a program:

.STACK ; marks the beginning of the stack segment

example:

```
.STACK    64      ;reserves 64B of memory for the stack
```

.DATA ; marks the beginning of the data segment

example:

```
.DATA1   DB      52H
                 ;DB directive allocates memory in byte-size chunks
```

Segments

.CODE ; marks the beginning of the code segment

- starts with PROC (procedures) directive
- the PROC directive may have the option FAR or NEAR
- ends by ENDP directives

Assemble, Link, and Run Program

STEP	INPUT	PROGRAM	OUTPUT
1. Edit the program	keyboard	editor	myfile.asm
2. Assemble the program	myfile.asm	MASM or TASM	myfile.obj myfile.lst myfile.crf
3. Link the program	myfile.obj	LINK or TLINK	myfile.exe myfile.map

Assemble, Link, Run Files

.asm – source file

.obj – machine language file

.lst – list file

- it lists all the Opcodes, Offset addresses, and errors that MASM detected

.crf – cross-reference file

- an alphabetical list of all symbols and labels used in the program as well as the program line numbers in which they are referenced

.map – map file

- to see the location and number of bytes used when there are many segments for code or data

PAGE and TITLE directives

PAGE [lines],[columns]

- To tell the printer how the list should be printed
- Default mode is 66 lines per page with 80 characters per line
- The range for number of lines is 10 to 255 and for columns is 60 to 132

TITLE

- Print the title of the program
- The text after the TITLE pseudo-instruction cannot be more than 60 ASCII characters

Control Transfer Instructions

- NEAR – When control transferred to a memory location within the current code segment
- FAR – When control is transferred outside the current code segment
- CS:IS – This register always points to the address of the next instruction to be executed.
  - In a NEAR jump, IP is updated, CS remains the same
  - In a FAR jump, both CS and IP are updated

Control Transfer Instructions

- Conditional Jumps – See Table 2-1
- Short Jump
  - – All conditional jumps are short jump
  - – The address of the target must be within –128 to +127 bytes of the IP
  - – The conditional jump is a two-byte instruction:
    - One byte is the opcode of the J condition
    - The 2nd byte is between 00 and FF
      - 256 possible addresses:
        - forward jump to +127
        - backward jump to –128

Control Transfer Instructions

- forward jump to +127:
  - calculation of the target address:
    - by adding the IP of the following instruction to the operand (see page 65)

- backward jump to –128
  - the 2nd byte is the 2's complement of the displacement value
  - Calculation of the target address:
    - the 2nd byte is added to the IP of the instruction after the jump (see Program 2-1, and page 65)

Control Transfer Instructions

- Unconditional Jumps – "JMP label" - When control is transferred unconditionally to the target location label
  - SHORT JUMP – "JMP SHORT label"
  - NEAR JUMP – "JMP label"
  - FAR JUMP – "JMP FAR PTR label"

- CALL statements – A control transfer instruction used to call a procedure
  - In NEAR call IP is saved on the stack (see figure 2-5, page 67)
  - In FAR call both CS and IP are saved on the stack
  - RET – the last instruction of the called subroutine

Control Transfer Instructions

- Assembly Language Subroutine
  - one main program and many subroutines
  - main program – is the entry point from DOS and is FAR
  - subroutines – called within the main program
    - can be FAR or NEAR
  - if after PROC nothing is mentioned, it defaults to NEAR

Data Types and Data Definition

- 80x86 data types
  - 8-bit or 16-bit
  - Positive or negative

  - *example1:*
    number $5_{10}(101_2)$ will be 0000 01010

  - *example2:*
    number $514_{10}(10\ 0000\ 0010_2)$ will be 0000 0010 0000 0010

Data Types and Data Definition

- Assembler data directives
  - **ORG** (origin) – to indicate the beginning of the offset address
    - *example:*

```
ORG     0010H
```

  - **DB** (define byte) – allocation of memory in byte-sized chunks
    - *example:*

```
DATA1   DB      25              ;decimal
DATA2   DB      10001001B       ;binary
DATA3   DB      12H             ;hex
DATA4   DB      '2591'          ;ASCII numbers
DATA5   DB      ?               ;set aside a byte
DATA6   DB      'Hello'         ;ASCII characters
DATA7   DB      "O' Hi"         ;ASCII characters
```

Data Types and Data Definition

- Assembler data directives
  - **DUP** (duplicate) – to duplicate a given number of characters
    - *example:*

```
DATA1    DB     0FFH, 0FFH, 0FFH, 0FFH        ;fill 4 bytes with FF
```

Can be replaced with:

```
DATA2    DB     4 DUP(0FFH)                   ;fill 4 bytes with FF

DATA3    DB     30 DUP(?)                     ;set aside 30 bytes

DATA4    DB     5 DUP (2 DUP (99))            ;fill 10 bytes with 99
```

Data Types and Data Definition

- Assembler data directives
  - **DW** (define word) – allocate memory 2 bytes (one word) at a time
    - *example:*

```
DATA1    DW    342                ;decimal
DATA2    DW    01010001001B       ;binary
DATA3    DW    123FH              ;hex
DATA4    DW    9,6,0CH, 0111B,'Hi'  ;Data numbers
DATA5    DW    8 DUP (?)          ;set aside 8 words
```

  - **EQU** (equate) – define a constant without occupying a memory location
    - *example:*

```
COUNT    EQU    25

;COUNT can be used in many places in the program
```

Data Types and Data Definition

- Assembler data directives
  - **DD** (define doubleword) – allocate memory 4 bytes (2 words) at a time
    - *example:*

```
DATA1    DD    1023                    ;decimal
DATA2    DD    01010001001001110110B   ;binary
DATA3    DD    7A3D43F1H               ;hex
DATA4    DD    54H, 65432H,65533       ;Data numbers
```

  - **DQ** (define quadwordequate) – allocate memory 8 bytes (4 words) at a time
    - *example:*

```
DATA1    DQ    6723F9H     ;hex
DATA2    DQ    'Hi'        ;ASCII characters
DATA3    DQ    ?           ;nothing
```

Data Types and Data Definition

- Assembler data directives
  - **DT** (define ten bytes) – allocates packed BCD numbers (used in multibyte addition of BCD numbers)
    - *example:*

```
DATA1   DT   123456789123        ;BCD
DATA2   DT   ?                   ;nothing
DATA3   DT   76543d              ;assembler will convert decimal
                                  number to hex and store it
```

Full Segment Definition

- Simple segment definition – refers to newer definition
  - Microsoft MASM 5.0 or higher
  - Borland's TASM ver. 1
- Full segment definition – refers to older definition
- SEGMENT directive – indicate to the assembler the beginning of a segment
- END directive – indicate to the assembler the beginning of a segment

Example:

```
label    SEGMENT    [options]
         ; statements
label    ENDS
```

EXE vs. COM

- COM files
  - Smaller in size (max of 64KB)
  - Does not have header block
- EXE files
  - Unlimited size
  - Do have header block (512 bytes of memory, contains information such as size, address location in memory, stack address)

Converting from EXE to COM

Procedure for conversion to COM from EXE

1. Change the source file to the COM format
2. Assemble
3. Link
4. Use utility program EXE2BIN that comes with DOS

```
C:>EXE2BIN program1, program1.com
```

Arithmetic and Logic Instructions and Programs

Arithmetic and Logic Instructions

- Integer Number Formats
- Signed (chap.6) and unsigned (chap.3) arithmetic Instructions
- Unsigned addition
- Unsigned Subtraction
- Unsigned Multiplication
- Unsigned Division
- Logic Instructions
- Rotate Instructions
- BCD and ASCII Operations & Conversion

Unsigned Addition

- Addition Instructions: ADD, ADC, INC
- Flags Changes (shown in Debug)
- The Carry Flag CF
- What's different between ADD and ADC?

Multiword numbers

- What happen if you have to add numbers that are larger than 2^{16}?

Unsigned Subtraction

- Subtraction Instructions: SUB,SBB,DEC
- What are the steps? (page 87)
- How are the flags got changed?
- Example 3-2, why is AF=0 afterward?
- What will be the value of AF after the execution of the program in Exp 3-3?
- What's different between SUB and SBB?

Subtraction of unsigned numbers

SUB dest,source ;dest=dest-source

Steps:

1. Take the 2's complement of the source operand
2. Add it to the destination operand
3. Invert the carry

MUL

	Operand 1	Operand 2	Result
byte * byte	AL	Register or memory	AX
word * word	AX		DX AX
word * byte	AL=byte,AH=0		DX AX

DIV

	Numerator	Denominator	Quotient	Remainder
byte / byte	AL=byte, AH=0	Register or memory	AL	AH
word / word	AX=word, DX=0		AX	DX
word / byte	AX=word		AL	AH
doubleword/word	DXAX=doubleword		AX	DX

Exception for division

1. Denominator is 0
2. Quotient is too large for the assigned register

Logic Instructions

- AND — Use to mask certain bits, test for zero
- OR — Use to test for zero
- XOR — Use to clear the contents of a register
  also to see if two register have the same value
- SHR — 0 -> MSB --->LSB ->CF
- SHL — CF<- MSB <--- LSB <-0
- CMP — CF=1 if dest<source, else CF=0
  ZF=1 if dest=source, else ZF=0

Table 3-3 (page 96)

- ROR, RCR *What's the difference?*
- ROL, RCL *What's the difference?*

Integer Number Formats

- Binary
- BCD (Binary Coded Decimal)
  - packed BCD (one byte has 2 BCD numbers)
    - 59_{10}= 01011001,
  - unpacked BCD (only the lower 4 bits are used)
    - 9_{10} = 00001001, 5_{10} = 00000101
- ASCII (page 102)
- Exercise: 29_{10}

binary? Unpacked BCD? Packed BCD? ASCII?

ASCII to Unpacked BCD

- ASCII for '0' is 30H = $0011\ 0000_2$
- Unpacked BCD for 0 is $0000\ 0000_2$
- How do you convert from ASCII to BCD in general?

ASCII to Packed BCD

- First convert to unpacked BCD
- Then combine the two unpacked BCD to make packed BCD
- Exp: ‘47’ (page 104)
- How about from packed BCD to ASCII?

BCD Addition

- Problem

```
MOV AL,17H
ADD AL,28H
```

- What is the result?
- What is the problem?
- How can you correct the problem?
- CISC microprocessors provide an instruction to deal with this problem (not RISC)

DAA

- DAA (Decimal Adjust for Addition)
  - Problem: After adding packed BCD numbers, the results is no longer BCD
  - EXP: 17 (0001 0111)

    + 28 (0010 1000)

    45 (0011 1111 = 3F)
  - Solution:
  - add 0110 to the lower 4 bits when the lower 4 bits is greater than 9 or if AF=1
  - add 011 to the upper 4 bits when the upper 4 bits is greater than 9 or if CF=1

Interrupt Programming with C

- Using C "high-level assembly"
- C programmers do need need to have detailed knowledge of 80x86 assembly
- C programmers write programs using:
  - DOS function calls INT 21H
  - BIOS interrupts
- Compilers provide functions:
  - int86 (calling any of the PC's interrupts)
  - intdos (only for INT 21H DOS function calls)

Interrupt Programming with C

- Programming BIOS interrupts with C/C++
  - Set registers to desired values
  - Call int86
    - Upon return from int86, the 80x86 registers can be accesses
  - To access the 80x86 registers use union of the REGS structure already defined by C compiler
    - union REGS regin,regout;
    - Registers for access are 16-bit (x) or 8-bit (h) format

Interrupt Programming with C

Example:

```
/*  C language                                  Assembly language  */
union REGS region,regout;
regin.h.ah=0x25;                                /* mov ah,25h      ;AH=25H  */
regin.x.dx=0x4567;                              /* mov dx,4567h  ;DH=4567H  */
regin.x.si=0x1290;                              /* mov si, 1290h  ;SI=1290H */
int86(interrupt#,&regin,&regout);               /* int #                    */
```

Interrupt Programming with C

- Programming INT 21H DOS function calls with C/C++
  - intdos used for DOS function calls
    - intdos(&regin,&regout); /* to be used for INT 21H only */

Signed Numbers, Strings, Tables

Signed Byte Operands

- D7 (MSB) is the sign, 0 means positive
- D0 to D6 are used for the magnitude of the number
- The range of positive number is [0,127]
- If 2's complement is used for negative numbers, the range is ?
- How do you represent –127 in binary format? How about –128?

Word-Sized Signed Numbers

- D15 is for the sign
- D0-D14 is for the magnitude
- What will be the range?
- Can Pentium 4 specify double word sized signed numbers for operands? What will be the range?

Overflow

- What is an overflow?
- If the result of an operation on signed numbers is too large for the register, the overflow flag (OF) will be raised to notify the programmers
- It is up to the programmer to take care of it
- EXP: 96 +70 0110 0000

  0100 0110

  1010 0110

Overflow in 8-bit operations

- OF is set to 1 if:

(1) There is a carry from D6 to D7 but no carry out of D7 (CF=0)

(2) There is a carry from D7 out (CF=1) but no carry from D6 to D7

- If there is a carry both from D6 to D7 and from D7 out, OF=0
- Why?

Overflow in 16-bit operations

- OF is set to 1 if:

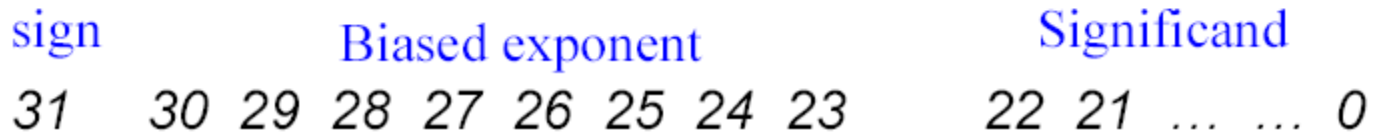
(1) There is a carry from D14 to D15 but no carry out of D15 (CF=0)

(2) There is a carry from D15 out (CF=1) but no carry from D14 to D15

- Examples (page 177, 178)
- The idea is to make use of the OF to handle the overflow problem

IEEE Single-Precision Floating-point Numbers

- 32 bits of data

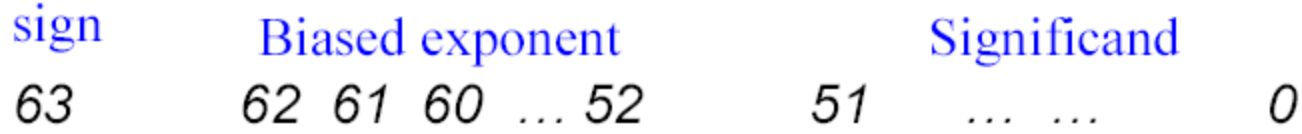


- **Convert from real to single-precision floating-point**

1. From real to binary form
2. Represent the binary number in scientific form **1**.xxxx E yyyy
3. Assign the sign bit
4. The exponent portion yyyy is added to 7F, to obtain the biased exponent, place in bits 23 to 30
5. The significand xxxx is placed in bits 22 to 0

IEEE Double-Precision Floating-point Numbers

- 64 bits of data



- Convert from real to floating-point

1. From real to binary form
2. Represent the binary number in scientific form **1**.xxxx E yyyy
3. Assign the sign bit
4. The exponent portion yyyy is added to 3FF, to obtain the biased exponent, place in bits 52 to 63
5. The significand xxxx is placed in bits 51 to 0

Using MASM

- Developed by Microsoft
- Used to translate 8086 assembly language into machine language
- 3 steps:
  - Prepare .ASM file using a text editor
  - Create .OBJ file using MASM
  - Create .EXE file using LINK
  - Once you have the .EXE file, debug can be used to test and run the program

Disassembly Using IDA Pro

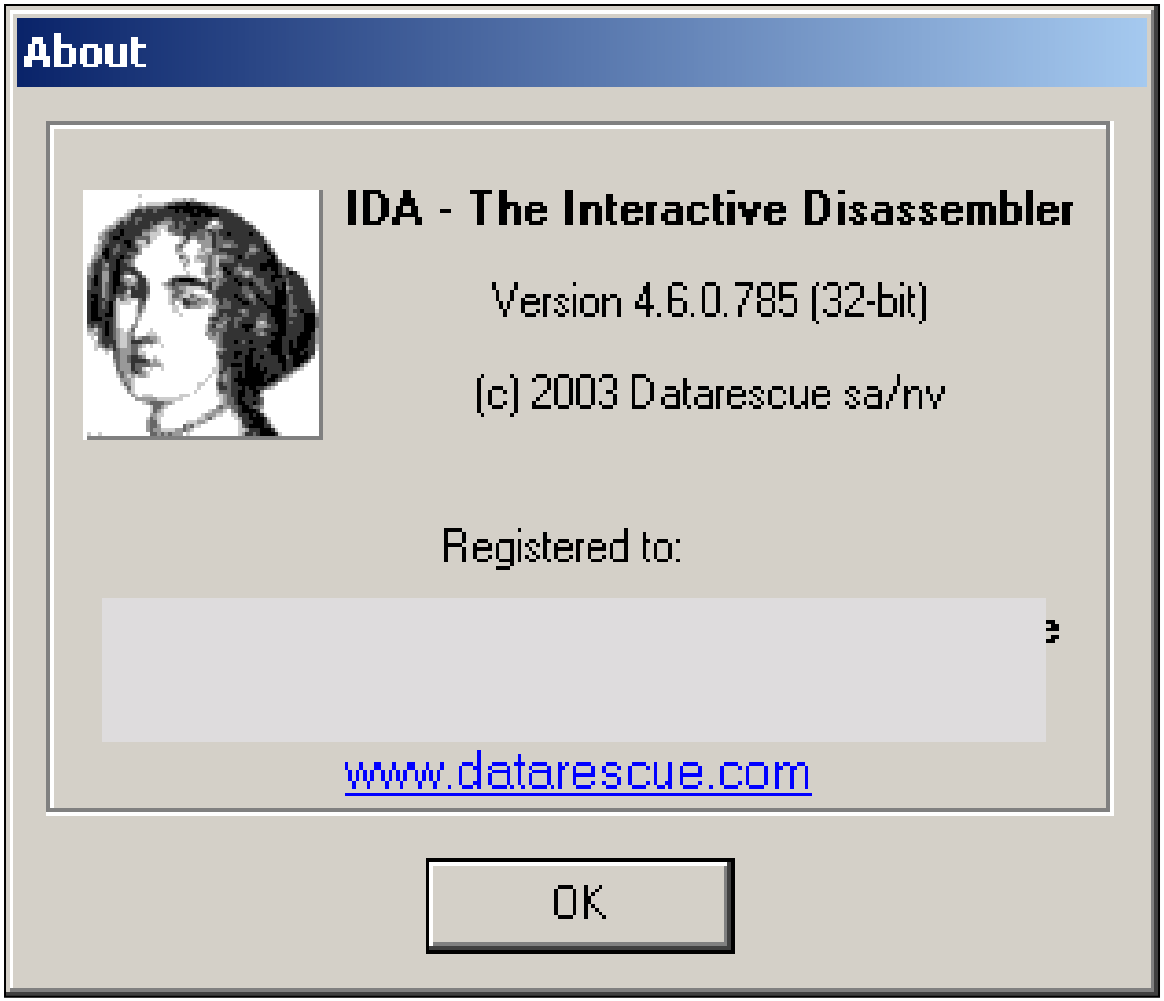
IDA Pro Disassembler

- Interactive disassembler commercially developed by Datarescue
- Supports over 30 families of processors (Intel x86, SPARC)
- Supports many file formats (PE, ELF)
- Provides powerful SDK

Using IDA Pro

- Loading a file
- General settings
- Views
- Navigating through the code
- Adding analysis content
- Searches (binary, text)
- Patching & scripting
- Exiting and saving

Tools – IDA Pro Demonstration



Reference Web Sites

1. http://www.intel.com
3. http://www.computerhistory.org
4. http://cedar.intel.com/media/training/proc_apps_3/tutorial/index.htm
5. http://www.intel.com/home/desktop/pentium4