中国科学技术大学
University of Science and Technology of China

Data movement Instructions of x86

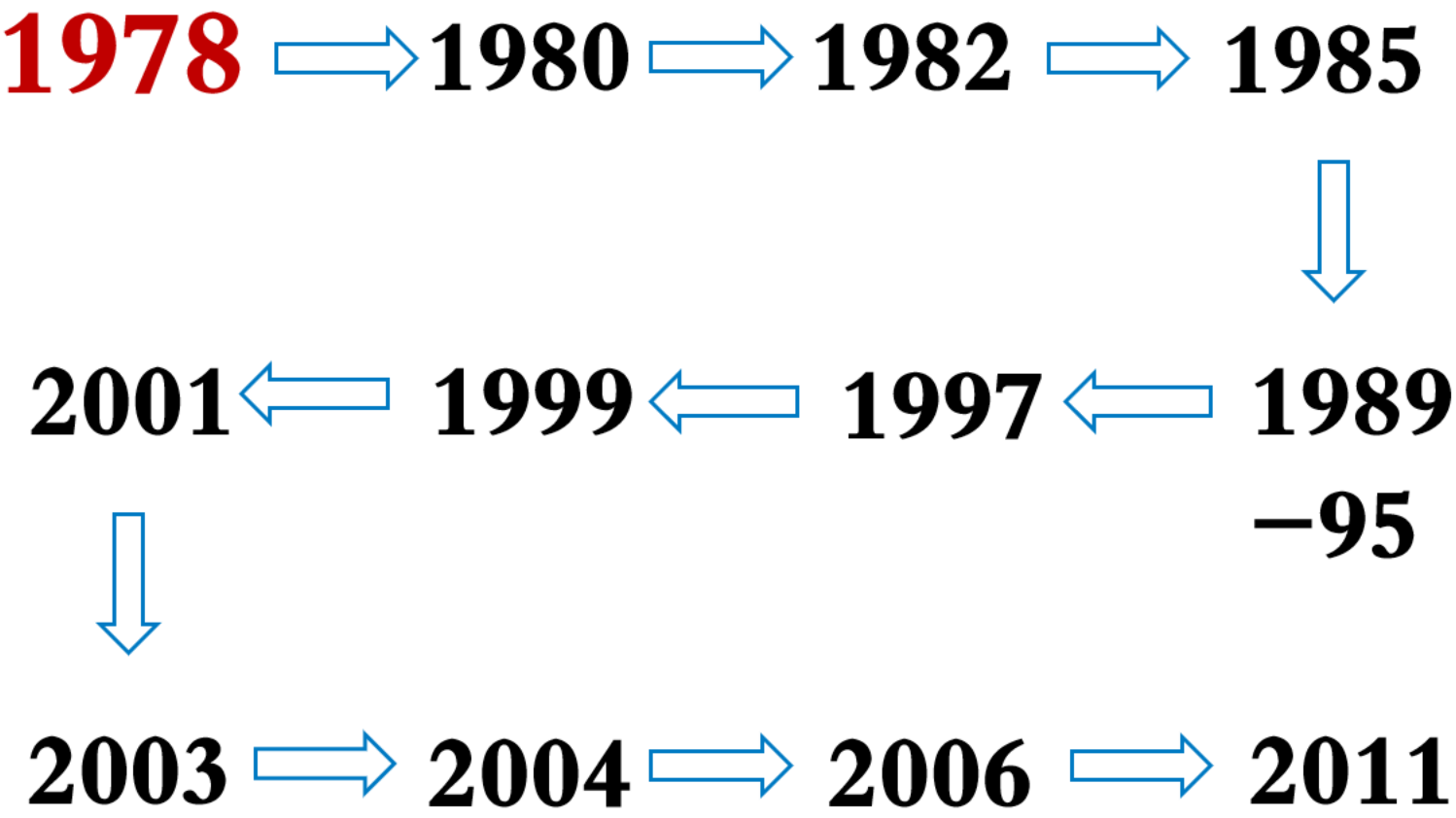
董宇轩 和泳毅 李莘

CONTENTS

中国科学技术大学
University of Science and Technology of China

- Evolution of the Intel x86
- Instructions
- Data Movement Instructions
- Summary

Evolution of the Intel x86



Instructions

An instruction is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

- **Label** (optional)
- **Instruction mnemonic** (required)
- **Operand(s)** (usually required)
- **Comment** (optional)

This is how the different parts are arranged:

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

Instructions

- **Label**

Data labels and ***Code labels***.

- ***Data labels***
  - *count DWORD 100*
  - *array DWORD 1024, 2048*
    DWORD 4096, 8192
  - *target:*
    mov ax,bx
    ...
    jmp target
- ***Code labels***
  - *L1: movax,bx*
  - *L2: …*

Instructions

- **Instruction Mnemonic**

Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

Instructions

- **Operands**

Example	Operand Type
96	Integer literal
2 + 4	Integer expression
eax	Register
count	Memory

- *stc* *; set Carry flag*
- *inc eax* *; add 1 to EAX*
- *mov count,ebx* *; move EBX to count*
- *imul eax,ebx,5*

Instructions

- **Comments**

The following information is typically included at the top of a program listing:

- Description of the program's purpose
- Names of persons who created and/or revised the program
- Program creation and revision dates
- Technical notes about the program's implementation

Comments can be specified in two ways:

- Single-line comments.
- Block comments

```
➢ COMMENT   !
        This line is a comment.
        This line is also a comment.
   !
```

Data Movement Instructions

- **Operand Types**

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

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]

Data Movement Instructions

中国科学技术大学
University of Science and Technology of China

- **Operand Types**

There are three basic types of operands:

- **Immediate**—uses a numeric literal expression
- **Register**—uses a named register in the CPU
- **Memory**—references a memory location

Data Movement Instructions

- **Operand Types**

Instruction Operand Notation, 32-Bit Mode.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

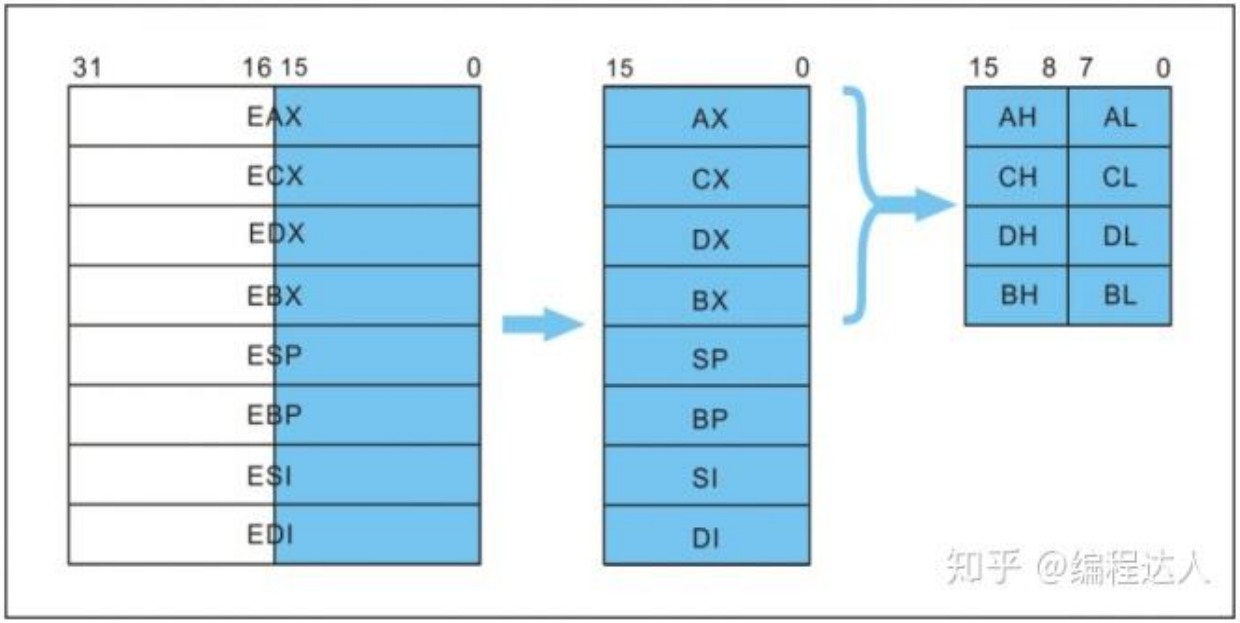
Data Movement Instructions

- **Some basic concepts**

General register :AX、BX、CX、DX

EAX、ECX、EDX、EBX：The extension of ax,bx,cx,dx

eax, ebx, ecx, edx, esi, edi, ebp, esp: The name of the general registers on the CPU in the X86 assembly language, which are 32-bit registers. When interpreted in C, these registers can be treated as variables.



Data Movement Instructions

中国科学技术大学
University of Science and Technology of China

- **Universal data movement instructions**

MOV	Transferring words or bytes.
MOVSX	Expand symbols, then transfer.
MOVZX	Zero expansion, then transfer.
PUSH	Push words into the stack.
POP	Pop words out of the stack.
PUSHA	Push AX,CX, DX,BX,SP,BP,SI,DI into the stack.
POPA	Pop DI,SI,BP,SP,BX,DX,CX,AX into the stack.
PUSHAD	Push EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI into the stack.
POPAD	Pop EDI,ESI,EBP,ESP,EBX,EDX,ECX,EAX onto the stack.
BSWAP	Exchange the order of bytes in 32-bit registers.
XCHG	Exchange words or bytes.
CMPXCHG	Compare and exchange operands.
XADD	Exchange first and then accumulate.
XLAT	Byte lookup table conversion.

Data Movement Instructions

中国科学技术大学
University of Science and Technology of China

- **Inputs and outputs movement instructions**

IN	I/O port input.
OUT	I/O port output.

Data Movement Instructions

- **Destination address movement instructions**

LEA	Load valid address.
LDS	Send target pointer, and load contents of pointer to DS.
LES	Send target pointer, and load contents of pointer to ES.
LFS	Send target pointer, and load its contents into FS.
LGS	Send target pointer, and load its contents into GS.
LSS	Send target pointer, and load its contents into SS.

Data Movement Instructions

中国科学技术大学
University of Science and Technology of China

- **Flags movement instructions**

LAHF	Register transfer, loading of the flags into the AH.
SAHF	Register transfer, loading the AH contents into the flag register.
PUSHF	Pushing.
POPF	Poping.
PUSHD	32-bit pushing.
POPD	32-bit poping.

Data Movement Instructions

- **Floating-point movement instructions**

FLDZ	0.0->ST(0)	Machine code D9 EE
FLD1	1.0->ST(0)	Machine code D9 E8
FLDPI	π->ST(0)	Machine code D9 EB
FLDL2T	ln10/ln2->ST(0)	Machine code D9 E9
FLDL2E	1/ln2->ST(0)	Machine code D9 EA
FLDLG2	ln2/ln10->ST(0)	Machine code D9 EC
FLDLN2	ln2->ST(0)	Machine code D9 ED

FLD real4 ptr mem	Single precision floating point	
		Machine code D9 mm000mmm
FLD real8 ptr mem	Double Precision Floating Point	
		Machine code DD mm000mmm
FLD real10 ptr mem	Crossover Floating Point	
		Machine code DB mm101mmm

Data Movement Instructions

中国科学技术大学
University of Science and Technology of China

- **Floating-point movement instructions**

FILD	**word ptr mem**	2-byte integers	Machine code DF mm000mmm
FILD	**dword ptr mem**	4-byte integers	Machine code DB mm000mmm
FILD	**qword ptr mem**	8-byte integers	Machine code DF mm101mmm

FST	**real4 ptr mem**	Single precision floating point Machine code D9 mm010mmm
FST	**real8 ptr mem**	Double Precision Floating Point Machine code DD mm010mmm

FIST	**word ptr mem**	2-byte integers	Machine code DF mm010mmm
FIST	**dword ptr mem**	4-byte integers	Machine code DB mm010mmm

FSTP	**real4 ptr mem**	Single precision floating point Machine code D9 mm011mmm
FSTP	**real8 ptr mem**	Double Precision Floating Point Machine code DD mm011mmm
FSTP	**real10 ptr mem**	Crossover Floating Point Machine code DB mm111mmm

Data Movement Instructions

中国科学技术大学
University of Science and Technology of China

- **Floating-point movement instructions**

FISTP word ptr mem	2-byte integers	Machine code DF mm011mmm
FISTP dword ptr mem	4-byte integers	Machine code DB mm011mmm
FISTP qword ptr mem	8-byte integers	Machine code DF mm111mmm
FCMOVB	ST(0),ST(i) <	Machine code DA C0iii
FCMOVBE	ST(0),ST(i) <=	Machine code DA D0iii
FCMOVE	ST(0),ST(i) =	Machine code DA C1iii
FCMOVNB	ST(0),ST(i) >=	Machine code DB C0iii
FCMOVNBE	ST(0),ST(i) >	Machine code DB D0iii
FCMOVNE	ST(0),ST(i) !=	Machine code DB C1iii
FCMOVNU	ST(0),ST(i) order	Machine code DB D1iii
FCMOVU	ST(0),ST(i) disorder	Machine code DA D1iii

Data Movement Instructions

- **Some basic concepts**

```
.data                    //Data Area
 sum DWORD 0
 .code      //Code Area
       mov eax,sum
       mov var2,ax
```

- **BYTE 8 bits**
- **WORD 16 bits**
- **DWORD 32 bits**

Data Movement Instructions

The purpose of the data movement instruction is to copy a piece of data from one location to another. In that case, the data movement instruction will contain a source and a destination operand, and the instruction will copy the value of the original operand to the destination operand and overwrite it.

There are five types of data transfer instructions, namely, **mov, moves, movz, push,** and **pop**.

- **MOV Instruction**

The MOV instruction copies data from a source operand to a destination operand. Known as a data transfer instruction, it is used in virtually every program. Its basic format shows that the first operand is the destination and the second operand is the source:

MOV destination,source

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer register (IP, EIP, or RIP) cannot be a destination operand.

Data Movement Instructions

- **MOV Instruction**

Here is a list of the standard MOV instruction formats:

(imm-Immediate reg- Register mem- Memory)

```
MOV     reg,reg
MOV     mem,reg
MOV     reg,mem
MOV     mem,imm
MOV     reg,imm
```

Data Movement Instructions

- **MOV Instruction**

A single MOV instruction cannot be used to move data directly from one memory location to another. Instead, you must move the source operand's value to a register before assigning its value to a memory operand:

```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax,var1
mov var2,ax
```

Data Movement Instructions

- **MOV Instruction**

The following code example shows how the same 32-bit register can be modified using differently sized data. When oneWord is moved to AX, it overwrites the existing value of AL. When oneDword is moved to EAX, it overwrites AX. Finally, when 0 is moved to AX, it overwrites the lower half of EAX.

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
.code
 mov eax,0              ; EAX = 00000000h
 mov al,oneByte         ; EAX = 00000078h
mov ax,oneWord          ; EAX = 00001234h
 mov eax,oneDword       ; EAX = 12345678h
mov ax,0                ; EAX = 12340000h
```

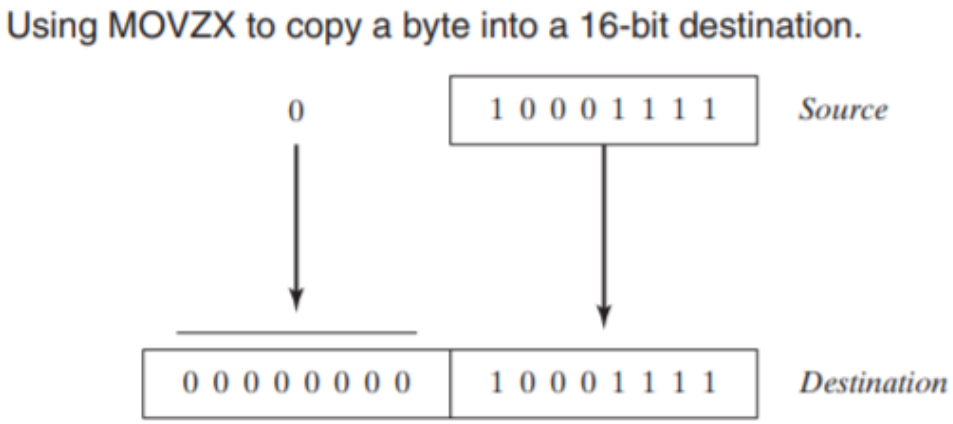
- **MOVZX Instruction**

The MOVZX instruction (move with zero-extend) copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits. This instruction is only used with unsigned integers. There are three variants :

```
MOVZX reg32,reg/mem8
MOVZX reg32,reg/mem16
MOVZX reg16,reg/mem8
```

```
.data
byteVal BYTE 10001111b
.code
movzx ax,byteVal     ; AX = 0000000010001111b
```

- **MOVZX Instruction**




```
mov bx,0A69Bh
movzx eax,bx          ; EAX = 0000A69Bh
movzx edx,bl          ; EDX = 0000009Bh
movzx cx,bl           ; CX = 009Bh
```

- **MOVZX Instruction**

The following examples use memory operands for the source and produce the same results:

```
.data
byte1 BYTE 9Bh
word1 WORD 0A69Bh
.code
movzx eax,word1          ; EAX = 0000A69Bh
movzx edx,byte1          ; EDX = 0000009Bh
movzx cx,byte1           ; CX = 009Bh
```

- **MOVSX Instruction**

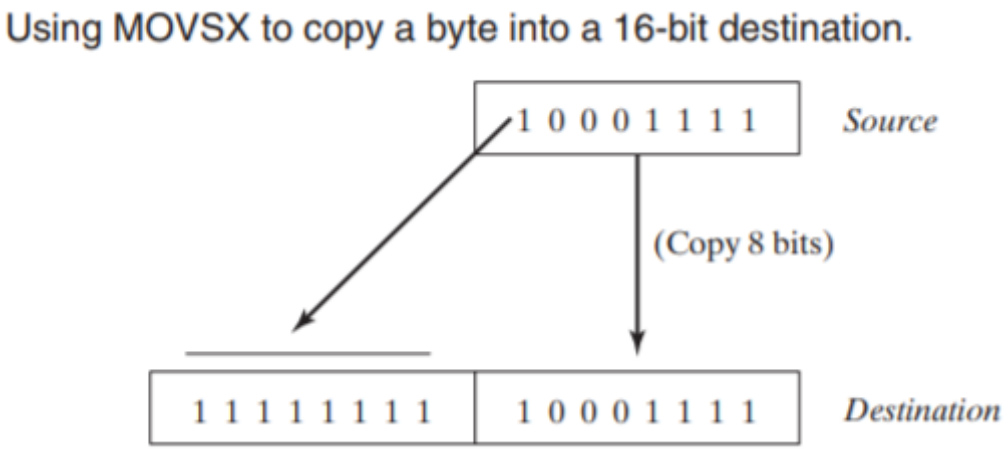
The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction is only used with signed integers. There are three variants:

```
MOVSX reg32,reg/mem8
MOVSX reg32,reg/mem16
MOVSX reg16,reg/mem8
```

```
.data
byteVal BYTE 10001111b
.code
movsx ax,byteVal              ; AX = 1111111110001111b
```

Data Movement Instructions

- **MOVSX Instruction**




```
mov bx,0A69Bh
movsx eax,bx        ; EAX = FFFFA69Bh
movsx edx,bl        ; EDX = FFFFFF9Bh
movsx cx,bl         ; CX = FF9Bh
```

- **MOVSX Instruction**

The following examples use memory operands for the source and produce the same results:

```
.data
byte1 BYTE 9Bh
word1 WORD 0A69Bh
.code
movzx eax,word1          ; EAX = 0000A69Bh
movzx edx,byte1          ; EDX = 0000009Bh
movzx cx,byte1           ; CX = 009Bh
```

Summary

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the x86 is more difficult to build than computers like ARMv7 and MIPS, but the large market meant in the PC era that AMD and Intel could afford more resources to help overcome the added complexity. What the x86 lacks in style, it rectifies with market size, making it beautiful from the right perspective.

Summary

Its saving grace is that the most frequently used x86 architectural components are not too difficult to implement, as AMD and Intel have demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

In the post-PC era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device.

中国科学技术大学
University of Science and Technology of China

Thank you