Chapter 4 – Basic Instructions

This chapter covers

- ✓ **instructions used to copy data from one location to another and**
- ✓ **instructions used for integer arithmetic.**
- ✓ **what types of operands are allowed for the various instructions.**
- ✓ **The concepts of time and space efficiency are introduced.**
- ✓ **Finally, some methods are given for accomplishing equivalent operations even when the desired operand types are not allowed.**

After studying this chapter you will know how to copy data between memory and CPU registers, and between two registers. You will also know how to use 80×86 addition, subtraction, multiplication, and division instructions, and how execution of these instructions affects flags.

Chapter 4 – Basic Instructions

4.1 Copying Data

mov (move) instructions copy data from one location to another. These instructions has the form

```
mov destination, source
```

and copies a byte, word, or doubleword value from the source operand location to the destination operand location. The destination location must be the same size as the source. A mov instruction is similar to a simple assignment statement in a high-level language.

```
mov Count, ecx          ;Count := Number
```

Chapter 4 – Basic Instructions

4.1 Copying Data

One limitation of the 80×86 architecture is that not all "logical" combinations of source and destination operands are allowed. In particular, ***you cannot have both source and destination in memory***.

```
mov Count, Number      ; illegal for two memory operands
```

All 80×86 mov instructions are coded with the same mnemonic. The assembler selects the correct opcode and other bytes of the machine code by looking at the operands as well as the mnemonic.

Chapter 4 – Basic Instructions

4.1 Copying Data

mov instructions that have an immediate source operand and a register destination operand.

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	Opcode
		386	486	Pentium		
Register 8	immediate byte	2	1	1	2	
AL						**B0**
CL						**B1**
DL						**B2**
BL						**B3**
AH						**B4**
CH						**B5**
DH						**B6**
AH						**B7**

Chapter 4 – Basic Instructions

4.1 Copying Data

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	Opcode
		386	486	Pentium		
Register 16	immediate word	2	1	1	3 *(plus prefix byte)*	
AX						**B8**
CX						**B9**
DX						**BA**
BX						**BB**
SP						**BC**
BP						**BD**
SI						**BE**
DI						**BF**

Chapter 4 – Basic Instructions
4.1 Copying Data

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	Opcode
		386	486	Pentium		
Register 32	immediate doubleword	2	1	1	5	
EAX						**B8**
ECX						**B9**
EDX						**BA**
EBX						**BB**
ESP						**BC**
EBP						**BD**
ESI						**BE**
EDI						**BF**

Chapter 4 – Basic Instructions
4.1 Copying Data

clock cycles: The length of time an instruction takes to execute is measured in.

determining the actual time: you must know the clock speed of the processor.

The Intel 8088 in the original IBM PC had a clock speed of 4.77 MHz (210ns).

Now many 80×86 personal computers operate at speeds higher than 200 MHz. (5 ns).

Chapter 4 – Basic Instructions

4.1 Copying Data

The number of bytes for each instruction is the same for the Intel 80386, 80486, and Pentium processors, which is because the object code is identical.

It would also be the same for 8086, 8088, 80186, and 80286 processors except that no 32-bit registers were available.

It may be surprising that the op codes for word and doubleword immediate-to-register mov instructions are identical.

One bit of this descriptor determines whether operands are 16-bit or 32-bit length by default.

With the assembler directives and link options, this bit is set to 1 to indicate 32-bit operands.

Chapter 4 – Basic Instructions

4.1 Copying Data

Therefore, the B8 opcode means, for instance, to copy the immediate doubleword following the opcode to EAX, not an immediate word to AX. If you code the 16-bit instruction

```
mov ax,0
```

then the assembler inserts the prefix byte 66 in front of the object code B8 0000, so that the code generated is actually 66 B8 0000.

In general, the prefix byte 66 tells the assembler to switch from the default operand size (32-bit or 16-bit) to the alternative size (16-bit or 32-bit) for the single instruction that follows the prefix byte.

Chapter 4 – Basic Instructions
4.1 Copying Data

Instructions sometimes affect various flag bits in the EFLAGS register. In general, an instruction may have one of three effects:

✓ no flags are altered
✓ specific flags are given values depending on the results of the instruction
✓ some flags may be altered, but their settings cannot be predicted

All mov instructions fall in the first category:

No mov instruction changes any flag.

Chapter 4 – Basic Instructions

4.1 Copying Data

The following figure lists the **mov instructions** that have *an immediate source* and *a memory destination*.

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	opcode
		386	486	Pentium		
Memory byte direct register indirect other	Immediate byte	2	1	1	 7 3 3-8	C6
Memory word direct register indirect other	Immediate word	2	1	1	 8 4 4-9	C7
Memory doubleword direct register indirect other	Immediate doubleword	2	1	1	 10 6 6-11	C7

Chapter 4 – Basic Instructions
4.1 Copying Data

lists most of the remaining 80×86 mov instructions.

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
register 8	register 8	2	1	1	2	8A
register 16	register 16	2	1	1	2	8B
register 32	register 32	2	1	1	2	8B
register 8	memory byte	4	1	1	2-7	8A
register 16	memory word	4	1	1	2-7	8B
register 32	memory doubleword	4	1	1	2-7	8B
AL	direct memory byte	4	1	1	5	A0
AX	direct word	4	1	1	5	A1
EAX	direct doubleword	4	1	1	5	A1

Chapter 4 – Basic Instructions

4.1 Copying Data

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
memory byte	register 8	2	1	1	2-7	88
memory word	register 16	2	1	1	2-7	89
memory doubleword	register 32	2	1	1	2-7	89
direct memory byte	AL	2	1	1	5	A2
direct word	AX	2	1	1	5	A3
direct doubleword	EAX	2	1	1	5	A3
segment register	register 16	2	3	1	2	8E
register 16	segment register	2	3	1	2	8C
segment register	memory word	2	3+	2+	2-7	8E
memory word	segment register	2	3	1	2-7	8C

Chapter 4 – Basic Instructions

4.1 Copying Data

Note that sometimes the same opcode is used for what appear to be distinct instructions, for example for a register 8 to register 8 move and for a memory byte to register 8 move. *In these cases the second byte of the instruction determines not only the destination register, it also encodes the source register or indicates the mode of a memory source byte*.

Two distinct instructions copy a memory operand to the accumulator. For example, either of opcodes A1 and 8B could be used to encode the instruction mov eax, Number. *The difference is that the 8B instruction opcode can also be used to copy doublewords to other destination registers, while the A1 opcode is specific to the accumulator*. An assembler normally uses the A1 version since it is one byte shorter.

It should also be noted that instructions that access memory may require more than the number of clock cycles listed. *One reason this can occur is memory that does not respond rapidly enough; in this case wait states, wasted clock cycles, are inserted until the memory responds*. Even with fast memory, extra cycles can be required to access a word or doubleword that is not aligned in memory -that is, stored on an address that is a multiple of two or four, respectively. A programmer should plan to keep frequently-used data in registers when possible.

Chapter 4 – Basic Instructions

4.1 Copying Data

When you first look at all the mov instructions summarized in previous chapters, you may think that you can use them to copy any source value to any destination location. However, many seemingly logical combinations are not available. These include

- ✓**a move with both source and destination in memory**
- ✓**immediate source to segment register destination**
- ✓**any move to or from the flag register**
- ✓**any move to the instruction pointer register**
- ✓**a move from one segment register to another segment register**
- ✓**any move where the operands are not the same size**
- ✓**a move of several objects**

You may need to do some of these operations. We describe below how to accomplish some of them.

Chapter 4 – Basic Instructions

4.1 Copying Data

Although there is no mov instruction to copy from a memory source to a memory destination, two moves using an intermediate register can accomplish the same thing.

```
mov Count, Number        ;illegal for two memory operands


mov eax, Number          ;Count := Number mov Count, eax
mov Count, eax
```

each using the accumulator EAX and one direct memory operand. Some register other than EAX could be used, but each of these instructions using the accumulator requires five bytes, while each of the corresponding instructions using some other register takes six bytes-EAX is chosen in the interest of space efficiency.

Chapter 4 – Basic Instructions

4.1 Copying Data

To load an immediate value into a segment register, one can use an immediate to register 16 move, followed by a register 16 to segment register move. This sequence is needed to initialize the data segment register DS when coding with segmented memory models.

```
        mov     ax,label
        mov     ds,ax
```

Chapter 4 – Basic Instructions

4.1 Copying Data

To change the size of data from a word to a byte, it is legal, for example, to transfer a word to a register 16, and then move out just the high-order or low-order byte to a destination.

Going the other way, one can piece together two bytes in the high and low bytes of a 16-bit register and then copy the resulting word to some destination. These techniques are occasionally useful, and others will be discussed in Chapter 8. It is sometimes necessary to extend a byte-length number to word or doubleword length, or a word length number to four bytes.

Suppose that you have source and destination locations declared as

```
source  DWORD 4 DUP(?)
dest    DWORD 4 DUP(?)
```

and that you want to copy all four doublewords from the source to the destination. One way to do this is with four instructions

Chapter 4 – Basic Instructions
4.1 Copying Data

```
mov eax,source      ;copy first doubleword
mov dest,eax
mov eax, source+4   ;copy second doubleword
mov dest+4,eax
mov eax, source+8   ;copy third doubleword
mov dest+8,eax
mov eax, source+12  ;copy fourth doubleword
mov dest+12,eax
```

An address like source+4 refers to the location four bytes (one doubleword) after the address of source. Since the four doublewords reserved at source are contiguous in memory, source+4 refers to the second doubleword. This code clearly would not be space efficient if you needed to copy 40 or 400 doublewords.

Chapter 4 – Basic Instructions

4.1 Copying Data

The 80×86 has a very useful xchg instruction that exchanges data in one location with data in another location.

```
        Temp := Value1;          {swap Value1 and Value2}
        Value1 := Value2;
        Value2 := Temp;
```

Assuming that Value1 is stored in the EAX register and Value2 is stored in the EBX register, the above swap can be coded as

```
        xchg eax, ebx            ;swap Value1 and Value2
```

Instead of using the xchg instruction, one could code

```
        mov ecx, eax             ;swap Value1 and Value2
        mov eax, ebx
        mov ebx, ecx
```

However, each of these mov instructions takes one clock cycle and two bytes for a total of three clock cycles and six bytes of code; the xchg instruction requires only two bytes and two clock cycles (on a Pentium). In addition, it is much easier to write one instruction than three, and the resulting code is easier to understand.

Chapter 4 – Basic Instructions

4.1 Copying Data

Figure 4-4 lists the various forms of the xchg instruction. Since 16-bit and 32-bit instructions are the same, distinguished by a prefix byte, they are shown together in the table.

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	opcode
		386	486	Pentium		
register 8	register 8	3	3	3	2	86
register 8	memory byte	5	5	3	2-7	86
EAX/AX	register 32/16	3	3	2	1	
	ECX/CX					91
	EDX/DX					92
	EBX/BX					93
	ESP/SP					94
	EBP/BP					95
	ESI/SI					96
	EDI/DI					97
register 32/16	register 32/16	3	3	3	2	87
register 32/16	memory 32/16	5	5	3	2-7	87

Chapter 4 – Basic Instructions

4.1 Copying Data

The xchg instructions illustrate again that the accumulator sometimes plays a special role in a computer's architecture. There are special instructions for swapping another register with the accumulator that are both faster than and require fewer bytes than the corresponding general-use register-to-register exchanges. These instructions can be also be used with the accumulator as the second operand.

Note that you cannot use an xchg instruction to swap two memory operands. In general, 80×86 instructions do not allow two memory operands.

Like mov instructions, xchg instructions do not alter any status flag; that is, after execution of an xchg instruction, the contents of the EFLAGS register remains the same as it was before execution of the instruction.

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

The Intel 80×86 microprocessor has add and sub instructions to perform addition and subtraction using byte, word, or doubleword length operands. The operands can be interpreted as unsigned numbers or 2's complement signed numbers. The 80×86 architecture also has inc and dec instructions to increment (add 1 to) and decrement (subtract 1 from) a single operand, and a neg instruction that negates (takes the 2's complement of) a single operand.

add, sub, inc, dec, and neg instructions all update flags in the EFLAGS register. The SF, ZF, OF, PF, and AF flags are set according to the value of the result of the operation.

Each add instruction has the form

add *destination, source*

The sub instructions all have the form

sub *destination, source*

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Here are some examples showing how these instructions function at execution time.

Before	Instruction executed	After
AX: 00 75 CX: 01 A2	add ax,cx	AX: 02 17 CX: 01 A2 SF 0 ZF 0 CF 0 OF 0

Addition and subtraction instructions set the sign flag SF to be the same as the high-order bit of the result. Thus,

when these instructions are used to add or subtract 2’s complement integers, SF=1 indicates a negative result.

The zero flag ZF is 1 if the result is zero, and 0 if the result is nonzero.

The carry flag CF records a carry out of the high order bit with addition or a borrow with subtraction. The overflow flag OF records overflow.

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Before	Instruction executed	After
EAX: 00 00 00 75 ECX: 00 00 01 A2	sub eax,ecx	EAX: FF FF FE D3 ECX: 00 00 01 A2 SF 1 ZF 0 CF 0 OF 0
AX: 77 AC CX: 4B 35	add ax,cx	AX: C2 E1 CX: 4B 35 SF 1 ZF 0 CF 0 OF 1
EAX: 00 00 00 75 ECX: 00 00 01 A2	sub ecx,eax	EAX: 00 00 00 75 ECX: 00 00 01 2D SF 0 ZF 0 CF 0 OF 0
BL: 4B	add bl,4	BL: 4F SF 0 ZF 0 CF 0 OF 0

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Before	Instruction executed	After
DX: FF 20 Word at value: FF 20	sub dx,value	DX: 00 00 value: FF 20 SF 0 ZF 1 CF 0 OF 0
EAX: 00 00 00 09	add eax,1	EAX: 00 00 00 0A SF 0 ZF 0 CF 0 OF 0
Doubleword at Dbl: 00 00 01 00	sub Dbl,1	Dbl: 00 00 00 FF SF 0 ZF 0 CF 0 OF 0

One reason that 2's complement form is used to represent signed numbers is that it does not require special hardware for addition or subtraction; the same circuits can be used to add unsigned numbers and 2's complement numbers. The flag values have different interpretations, though, depending on the operand type. For instance, if you add two large unsigned numbers and the high order bit of the result is 1, then SF will be set to 1, but this does not indicate a negative result, only a relatively large sum. For an add with unsigned operands, CF=1 would indicate that the result was too large to store in the destination, but with signed operands, OF=1 would indicate a size error.

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Figure 4.5 gives information for both addition and subtraction instructions. For each add there is a corresponding sub instruction with exactly the same operand types, number of clock cycles, and number of bytes of object code.

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode add	sub
register 8	immediate 8	2	1	1	3	80	80
register 16	immediate 8	2	1	1	3	83	83
register 32	immediate 8	2	1	1	3	83	83
register 16	immediate 16	2	1	1	4	81	81
register 32	immediate 32	2	1	1	6	81	81
AL	immediate 8	2	1	1	2	04	2C
AX	immediate 16	2	1	1	3	05	2D
EAX	immediate 32	2	1	1	5	05	2D

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode add	sub
memory byte	immediate 8	7	3	3	3+	80	80
Memory word	immediate 8	7	3	3	3+	83	83
Memory doubleword	immediate 8	7	3	3	3+	83	83
Memory word	immediate 16	7	3	3	4+	81	81
Memory doubleword	immediate 32	7	3	3	6+	81	81
register 8	register 8	2	1	1	2	02	2A
register 16	register 16	2	1	1	2	03	2B
register 32	register 32	2	1	1	2	03	2B

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode add	sub
register 8	memory byte	6	2	2	2+	02	2A
register 16	Memory word	6	2	2	2+	03	2B
register 32	Memory doubleword	6	2	2	2+	03	2B
Memory word	immediate 16	7	3	3	2+	00	28
Memory doubleword	immediate 32	7	3	3	2+	01	29

Chapter 4 – Basic Instructions
4.2 Integer Addition and Subtraction Instruction

In addition or subtraction operands are the fastest when both operands are in registers and the slowest when the destination operand is in memory.

It is interesting to note that it is faster to add an operand in memory to the contents of a register than to add the value in a register to a memory operand; this is true since memory must be accessed twice in the latter case, once to get the first addend and once to store the sum.

Notice that an immediate source can be a single byte even when the destination is a word or doubleword. Since immediate operands are often small, this makes the object code more compact. Byte-size operands are sign-extended to word or doubleword size at run time before the addition or subtraction operation.

It may be surprising that some add and sub instructions have the same opcode. In such cases, one of the fields in the second instruction byte distinguishes between addition and subtraction.

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

The **inc** (increment) and **dec** (decrement) instructions are special-purpose addition and subtraction instructions, always using 1 as an implied source. They have the forms

```
        inc destination
```

and

```
        dec destination
```

The inc and dec instructions treat the value of the destination operand as an unsigned integer.

They affect the OF, SF, and ZF flags just like addition or subtraction of one, but they do not change the carry flag CF.

Chapter 4 – Basic Instructions
4.2 Integer Addition and Subtraction Instruction

Example

Before	Instruction executed	After
ECX: 00 00 01 A2	inc ecx	EAX: 00 00 01 A3 SF 0 ZF 0 OF 0
AL: F5	dec al	AL: F4 SF 1 ZF 0 OF 0
Word at Count: 00 09	inc Count	Count: 00 0A SF 0 ZF 0 OF 0
BX: 00 01	dec bx	BX: 00 00 SF 0 ZF 1 OF 0
EDX: 7F FF FF FF	inc edx	EDX 80 00 00 00 SF 1 ZF 0 OF 1

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Destination Operand	Clock Cycles			Number of Bytes	opcode	
	386	486	Pentium		inc	dec
register 8	2	1	1	2	FE	FE
register 16	2	1	1	1		
AX					40	48
CX					41	49
DC					42	4A
BX					43	4B
SP					44	4C
BP					45	4D
SI					46	4E
DI					47	4F

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Destination Operand	Clock Cycles			Number of Bytes	opcode	
	386	486	Pentium		inc	dec
register 32	2	1	1	1		
EAX					40	48
ECX					41	49
EDC					42	4A
EBX					43	4B
ESP					44	4C
EBP					45	4D
ESI					46	4E
EDI					47	4F
memory byte	6	3	3	2+	FE	FE
Memory word	6	3	3	2+	FF	FF
memory doubleword	6	3	3	2+	FF	FF

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

The inc and dec instructions are especially useful for incrementing and decrementing counters. For example, the instructions

```
add cx,1  ;incrementing loop counter
```

and

```
inc cx    ;incrementing loop counter
```

are functionally equivalent. The add instruction requires three bytes (three bytes instead of four since the immediate operand will fit in one byte), while the inc instruction requires one byte. Either executes in one clock cycle on a Pentium, so execution times are identical.

4.2 Integer Addition and Subtraction Instruction

A neg instruction negates, or finds the 2's complement of, its single operand. When a positive value is negated the result is negative; a negative value will become positive. Zero remains zero. Each neg instruction has the form

```
neg destination
```

Before	Instruction executed	After
BX: 01 A2	neg bx	BX: FE 5E SF 1 ZF 0
DH: F5	neg dl	DH: 0B SF 0 ZF 0
Word at Flag: 00 01	neg Flag	Flag: FF FF SF 1 ZF 0
EAX: 00 00 00 00	neg eax	EDX 00 00 00 00 SF 0 ZF 1

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

Figure 4.7 shows allowable operands for neg instructions .

	Clock Cycles				
Destination Operand	386	486	Pentium	Number of Bytes	opcode
register 8	2	1	1	2	F6
register 16	2	1	1	2	F7
register 32	2	1	1	2	F7
Memory byte	6	3	3	2+	F6
Memory word	6	3	3	2+	F7
memory doubleword	6	3	3	2+	F7

Chapter 4 – Basic Instructions

4.2 Integer Addition and Subtraction Instruction

An example of a complete program that uses the new instructions.

The program inputs values for three numbers, *x*, *y* and *z*, evaluates the expression $-(x + y - 2z + 1)$ and displays the result.

The design implemented is

```
prompt for and input value for x;
convert × from ASCII to 2's complement form;
expression := x;
prompt for and input value for y;
convert y from ASCII to 2's complement form;
add y to expression, giving × + y;
prompt for and input value for z;
convert z from ASCII to 2's complement form;
calculate 2*z as (z + z);
subtract 2*z from expression, giving × + y −2*z;
add 1 to expression, giving × + y −2*z + 1;
negate expression, giving −(x + y −2*z + 1);
convert the result from 2's complement to ASCII;
display the result;
```

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

The 80×86 architecture has two multiplication instruction mnemonics.

Any **`imul`** instruction treats its operands as signed numbers.

A **`mul`** instruction treats its operands as unsigned binary numbers.

If only non-negative numbers are to be multiplied, `mul` should usually be chosen instead of imul since it is a little faster.

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

The mul instruction has a single operand; its format is

```
mul source
```

The source operand may be byte, word, or doubleword-length, and it may be in a register or in memory.

The location of the other number to be multiplied is always the accumulator-AL for a byte source, AX for a word source, and EAX for a doubleword source.

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

If *source* has byte length, then it is multiplied by the byte in AL; the product is 16 bits long, with a destination of the AX register. So the original value in AX is replaced

If *source* has word length, then it is multiplied by the word in AX; the product is 32 bits long, with its low order 16 bits going to the AX register and its high order 16 bits going to the DX register. So the original values in AX and DX are both wiped out.

If *source* is a doubleword, then it is multiplied by the doubleword in EAX; the product is 64 bits long, with its low order 32 bits in the EAX register and its high order 32 bits in the EDX register. So the values in EAX and EDX are replaced by the product.

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

At first glance, it may seem strange that the product is twice the length of its two factors. However, this also occurs in ordinary decimal multiplication; if, for example, two four-digit numbers are multiplied, the product will be seven or eight digits long.

Even when provision is made for double-length products, **it is useful to be able to tell whether the product is the same size as the source**; that is, if the high-order half is zero.

4.3 Multiplication Instructions

With mul instructions, the carry flag CF and overflow flag OF are set to 1 if the high order half of the product is not zero, but are cleared to 0 if the high order half of the product is zero.

These are the only meaningful flag values following multiplication operations; previously set values of AF, PF, SF, and ZF flags may be destroyed.

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

Figure 4.10 summarizes the allowable operand types for mul instructions. No immediate operand is allowed in a mul.

The actual number of clock cycles for the 80386 and 80486 depends on the numbers being multiplied.

	Clock Cycles				
Source Operand	**386**	**486**	**Pentium**	**Number of Bytes**	**opcode**
register 8	9-14	13-18	11	2	F6
register 16	9-22	13-26	11	2	F7
register 32	9-38	13-42	10	2	F7
memory byte	12-17	13-18	11	2+	F6
memory word	12-25	13-26	11	2+	F7
memory doubleword	12-41	13-42	10	2+	F7

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

Here are some examples to illustrate how the `mul` instructions work.

Before	Instruction executed	After
AX: 00 05 BX: 00 02 DX: ?? ??	mul bx	DX: 00 00 AX: 00 0A CF,OF 0
EAX: 00 00 00 0A EDX: ?? ?? ?? ??	mul eax	EDX: 00 00 00 00 EAX: 00 00 00 64 CF,OF 0
AX: ?? 05 Byte at Factor: FF	mul Factor	AX: 04 FB CF,OF 1

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

The signed multiplication instructions use mnemonic imul.

There are three formats, each with a different number of operands.

The first format is

```
imul source
```

the same as for mul, with *source* containing one factor and the accumulator the other. Again, the source operand cannot be immediate.

The destination is AX, DX:AX, or EDX:EAX, depending on the size of the source operand. The carry and overflow flags are set to 1 if the bits in the high-order half are significant, and cleared to 0 otherwise.

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

Single-operand imul instructions are summarized in Fig. 4.11. Notice that this table is identical to Fig. 4.10. Even the opcodes are the same for mul and single-operand imul instructions, with a field in the second byte of the instruction distinguishing the two.

	Clock Cycles				
Source Operand	**386**	**486**	**Pentium**	**Number of Bytes**	**opcode**
register 8	9-14	13-18	11	2	F6
register 16	9-22	13-26	11	2	F7
register 32	9-38	13-42	10	2	F7
memory byte	12-17	13-18	11	2+	F6
memory word	12-25	13-26	11	2+	F7
memory doubleword	12-41	13-42	10	2+	F7

4.3 Multiplication Instructions

The second imul format is

```
imul    register,source
```

Here the source operand can be in a register, in memory, or immediate. The other factor is in the register, which also serves as the destination. Operands must be words or doublewords, not bytes. The product must "fit" in same size as the factors; if it does, CF and OF are cleared to 0, if not they are set to 1.

Chapter 4 – Basic Instructions
4.3 Multiplication Instructions

Figure 4.12 summarizes two-operand imul instructions. Note that some of these instructions have two byte long opcodes.

Immediate operands can be either the size of the destination register or a single byte.

Single-byte operands are sign extended before multiplication-that is, the sign bit is copied to leading bit positions, giving a 16 or 32-bit value that represents the same signed integer as the original 8-bit operand.

Chapter 4 – Basic Instructions
4.3 Multiplication Instructions

Operand1	Operand2	Clock Cycles			Number of Bytes	opcode
		386	486	Pentium		
register 16	register 16	9-22	13-26	11	3	0F AF
register 32	register 32	9-38	13-42	10	3	0F AF
register 16	Memory word	12-25	13-26	11	3+	0F AF
register 32	Memory doubleword	12-41	13-42	10	3+	0F AF
register 16	immediate byte	9-14	13-18	11	3	6B
register 16	immediate word	9-22	13-26	11	4	69
register 32	immediate byte	9-14	13-18	11	3	6B
register 32	immediate doubleword	9-38	13-42	10	6	69

4.3 Multiplication Instructions

The third `imul` format is

```
imul register,source,immediate
```

With this version, the first operand, a register, is only the destination for the product; the two factors are the contents of the register or memory location given by *source* and the immediate value. Operands *register* and *source* are the same size, both 16-bit or both 32-bit. If the product will fit in the destination register, then CF and OF are cleared to 0; if not, they are set to 1.

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

Register Destination	Source	Immediate Operand	Clock Cycles			Number of Bytes	opcode
			386	486	Pentium		
register 16	register 16	byte	9-14	13-18	10	3	6B
register 16	register 16	word	9-22	13-26	10	4	69
register 16	memory 16	byte	12-17	13-18	10	3+	6B
register 16	memory 16	Word	12-25	13-26	10	4+	69
register 32	register 32	byte	9-14	13-18	10	3	6B
register 32	register 32	doubleword	9-38	13-42	10	6	69
register 32	memory 32	Byte	9-17	13-18	10	3+	6B
register 32	memory 32	doubleword	9-41	13-42	10	6+	69

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

Some examples will help show how the imul instructions work.

Before	Instruction executed	After
AX: 00 05 BX: 00 02 DX: ?? ??	imul bx	DX: 00 00 AX: 00 0A CF,OF 0
AX: ?? 05 Byte at Factor: FF	imul Factor	AX: FF FB CF,OF 0
EBX: 00 00 00 0A Byte at Factor: FF	imul ebx,10	EBX: 00 00 00 64 CF,OF 0
ECX: FF FF FF F4 Doubleword at Double: FF FF FF B2	imul ecx,Double	ECX: 00 00 03 A8 CF,OF 0
Word at Value: 08 F2 BX: ?? ??	imul bx,value,1000	BX: F1 50 CF,OF 1

Chapter 4 – Basic Instructions

4.3 Multiplication Instructions

It is faster to calculate 2*z* by adding *z* to itself than by using a multiplication instruction. In that situation, *z* was in the AX register, so

```
add ax, ax ; compute 2z
```

did the job. This instruction is two bytes long, and on a Pentium system takes one clock cycle. To do the same task using multiplication, you can code

```
imul ax, 2 ; compute 2z
```

This instruction is three bytes long since the immediate operand 2 is short enough to fit in a single byte; it takes 13-18 clock cycles on an 80486 or 10 clock cycles on a Pentium, much longer than the addition instruction

Chapter 4 – Basic Instructions
4.4 Division Instructions

The Intel 80×86 instructions for division parallel those of the single-operand multiplication instructions; idiv is for division of signed 2's complement integers and div is for division of unsigned integers. The division instructions have formats

```
        idiv source
```

and

```
        div source
```

The source operand identifies the divisor. The divisor can be in a register or memory, but not immediate.
Both div and idiv use an implicit dividend (the operand you are dividing into). If *source* is byte length, then the double-length dividend is word size and is assumed to be in the AX register. If *source* is word length, then the dividend is a doubleword and is assumed to have its low order 16 bits in the AX register and its high order 16 bits in the DX register. If *source* is doubleword length, then the dividend is a quadword (64 bits) and is assumed to have its low order 32 bits in the EAX register and its high order 32 bits in the EDX register.

Chapter 4 – Basic Instructions
4.4 Division Instructions

The table in Fig. 4.15 summarizes the locations of the dividend, divisor, quotient, and remainder for 80×86 division instructions.

Source(divisor) size	Other operand (dividend)	quotient	remainder
Byte	AX	AL	AH
Word	DX:AX	AX	DX
doubleword	EDX:EAX	EAX	EDX

For all division operations, the dividend, divisor, quotient, and remainder must satisfy the equation

dividend = quotient*divisor + remainder

For unsigned div operations, the dividend, divisor, quotient, and remainder are all treated as non-negative numbers.

For signed idiv operations, the sign of the quotient is determined by the signs of the dividend and divisor using the ordinary rules of signs; the sign of the remainder is always the same as the sign of the dividend.

Chapter 4 – Basic Instructions

4.4 Division Instructions

The division instructions do not set flags to any significant values. They may destroy previously set values of AF, CF, OF, PF, SF, and ZF flags.

Before	Instruction executed	After
EDX: 00 00 00 00 EAX: 00 00 00 64 EBX: 00 00 00 0D	div ebx	EDX: 00 00 00 09 EAX: 00 00 00 07
DX: 00 00 AX: 00 64 CX: 00 0D	idiv cx	DX: 00 09 AX: 00 07
AX: 00 64 Byte at Divisor: 0D	div Divisor	AX: 09 07

In each of these examples, the decimal number 100 is divided by 13. Since

100 = 7 * 13 + 9

Chapter 4 – Basic Instructions
4.4 Division Instructions

For operations where the dividend or divisor is negative, equations analogous to the one above are

$$100 = (-7) * (-13) + 9$$

$$-100 = (-7) * 13 + (-9)$$

$$-100 = 7 * (-13) + (-9)$$

The following examples reflect these equations for word size divisors of 13 or −13.

Before	Instruction executed	After
DX: 00 00 AX: 00 64 CX: FF F3	Idiv cx	DX: 00 09 AX: FF F9
DX: FF FF AX: FF 9C CX: 00 0D	idiv cx	DX: FF F7 AX: FF F9

Chapter 4 – Basic Instructions

4.4 Division Instructions

Before	Instruction executed	After
DX: FF FF AX: FF 9C CX: FF F3	idiv cx	DX: FF F7 AX: 0007

In the second and third examples, the dividend −100 is represented as the 32 bit number FF FF FF 9C in the DX and AX registers.

4.4 Division Instructions

Finally, here are two examples to help illustrate the difference between signed and unsigned division.

Before	Instruction executed	After
AX: FE 01 BL: E0	idiv bl	AX: E1 0F
AX: FE 01 BL: FF	div bl	AX: 00 FF

With the signed division, −511 is divided by −32, giving a quotient of 15 and a remainder of −31. With the unsigned division, 65025 is divided by 255, giving a quotient of 255 and a remainder of 0.

Chapter 4 – Basic Instructions

4.4 Division Instructions

With multiplication, the double length destination in each single-operand format guarantees that the product will fit in the destination location-nothing can go wrong during a single-operand multiplication operation.

There can be errors during division. One obvious cause is an attempt to divide by zero. A less obvious reason is a quotient that is too large to fit in the single-length destination; if, say, 00 02 46 8A is divided by 2, the quotient 1 23 45 is too large to fit in the AX register. If an error occurs during the division operation, the 80×86 generates an exception. The routine, or interrupt handler, that services this exception may vary from system to system.

The 80×86 leaves the destination registers undefined following a division error.

Chapter 4 – Basic Instructions

4.4 Division Instructions

Figure 4.16 lists the allowable operand types for idiv instructions and Fig. 4.17 lists the allowable operand types for div instructions. The only differences in the two tables are in the number of clock cycles columns; div operations are slightly faster than idiv operations.

	Clock Cycles				
Operand	386	486	Pentium	Number of Bytes	opcode
register 8	19	19	22	2	F6
register 16	27	27	30	2	F7
register 32	43	43	48	2	F7
memory byte	22	22	22	2+	F6
memory word	30	28	30	2+	F7
memory doubleword	46	44	48	2+	F7

Chapter 4 – Basic Instructions

4.4 Division Instructions

Figure 4-17 div instructions

	Clock Cycles				
Operand	386	486	Pentium	Number of Bytes	opcode
register 8	14	16	17	2	F6
register 16	22	24	25	2	F7
register 32	38	40	41	2	F7
memory byte	17	16	17	2+	F6
memory word	25	24	25	2+	F7
memory doubleword	41	40	41	2+	F7

Chapter 4 – Basic Instructions

4.4 Division Instructions

When arithmetic is being done with operands of a given length, the dividend must be converted to double length before a division operation is executed.

For unsigned division, a doubleword-size dividend must be converted to quadword size with leading zero bits in the EDX register. This can be accomplished many ways, two of which are

```
mov     edx,0
```

and

```
sub     edx,edx
```

Similar instructions can be used to put a zero in DX prior to unsigned division by a word operand or to put a zero in AH prior to unsigned division by a byte operand.

Chapter 4 – Basic Instructions

4.4 Division Instructions

The situation is more complicated for signed division.

A positive dividend must be extended with leading 0 bits, but a negative dividend must be extended with leading 1 bits.

The 80×86 has three instructions for this task. The cbw, cwd, and cdq instructions are different from the instructions covered before in that these instructions have no operands.

The cbw instruction always has AL as its source and AX as its destination,

cwd always has AX as its source and DX and AX as its destination, and

cdq always has EAX as its source and EDX and EAX as its destination.

The source register is not changed, but is extended as a signed number into AH, DX, or EDX. These instructions are summarized together in Fig. 4.18, which also includes the cwde instruction that extends the word in AX to its signed equivalent in EAX, paralleling the job that cbw does.

Chapter 4 – Basic Instructions

4.4 Division Instructions

These instructions are summarized together in Fig. 4.18, which also includes the cwde instruction that extends the word in AX to its signed equivalent in EAX, paralleling the job that cbw does.'

	Clock Cycles				
Instruction	**386**	**486**	**Pentium**	**Number of Bytes**	**opcode**
Cbw	3	3	3	1	98
cwd	2	3	2	1	99
cdq	2	3	2	1	99
cwde	3	3	3	1	98

Chapter 4 – Basic Instructions

4.4 Division Instructions

The cbw (convert byte to word) instruction extends the 2's complement number in the AL register half to word length in AX.

The cwd (convert word to double) instruction extends the word in AX to a doubleword in DX and AX.

The cdq (convert double to quadword) instruction extends the word in EAX to a quadword in EDX and EAX.

The cwde (convert word to double extended) instruction extends the word in AX to a doubleword in EAX; this is not an instruction that would normally be used to prepare for division.

Each instruction copies the sign bit of the original number to each bit of the high order half of the result.

None of these instructions affect flags.

Chapter 4 – Basic Instructions

4.4 Division Instructions

Some examples are

Before	Instruction executed	After
AX: 07 0D DX: ?? ??	cwd	DX: 00 00 AX: 07 0D
EAX: FF FF FA 13 EDX: ?? ?? ?? ??	cdq	EDX: FF FF FF FF EAX: FF FF FF 13
AL: 53	cbw	AX: 00 53
AL: C6	cbw	AX: FF C6
AX: FF 2A	cwde	EAX: FF FF FF 2A

Chapter 4 – Basic Instructions

4.4 Division Instructions

Two "move" instructions are somewhat similar to the above "convert" instructions. These instructions copy an 8-bit or 16-bit source operand to a 16-bit or 32-bit destination, extending the source value.

The movzx instruction always extends the source value with zero bits. It has the format

```
        movzx  register,source
```

The movsx instruction extends the source value with copies of the sign bit. It has a similar format

```
        movsx  register,source
```

Data about these instructions is in Fig. 4.19. With either instruction the source operand can be in a register or in memory. Neither instruction changes any flag value.

Chapter 4 – Basic Instructions

4.4 Division Instructions

FIG 4.19 MOVSX AND MOVZX INSTRUCTIONS

		Clock Cycles				opcode	
Destination	Source	386	486	Pentium	Number of Bytes	movsx	movzx
register 16	register 8	3	3	3	3	0F BE	0F B6
register 32	register 8	3	3	3	3	0F BE	0F B6
register 32	register 16	3	3	3	3	0F BE	0F B7
register 16	memory byte	6	3	3	3+	0F BE	0F B6
register 32	memory byte	6	3	3	3+	0F BE	0F B6
register 32	memory word	6	3	3	3+	0F BE	0F B7

Chapter 4 – Basic Instructions

4.4 Division Instructions

Here are a few examples showing how these instructions work.

Before	**Instruction executed**	**After**
`Word at value: 07 0D`	`movsx ecx,value`	`ECX: 00 00 07 0D`
`Word at value: F7 0D`	`movsx ecx,value`	`ECX: FF FF F7 0D`
`Word at value: 07 0D`	`movzx ecx,value`	`ECX: 00 00 07 0D`
`Word at value: F7 0D`	`movzx ecx,value`	`ECX: 00 00 F7 0D`

Chapter 4 – Basic Instructions

4.4 Addition and Subtraction of Larger Numbers

The add and sub instructions covered in Section 4.2 work with byte-length, word-length, or doubleword-length operands. Although the range of values that can be stored in a doubleword is large, −2,147,483,648 (80000000_{16}) to 2,147,483,647 ($7FFFFFFF_{16}$), it is sometimes necessary to do arithmetic with even larger numbers. Very large numbers can be added or subtracted a group of bits at a time.

Assume that the two numbers to be added are in four doublewords in the data segment.

```
Nbr1Hi   DWORD   ?    ;High order 32 bits of Nbr1
Nbr1Lo   DWORD   ?    ;Low order 32 bits of Nbr1
Nbr2Hi   DWORD   ?    ;High order 32 bits of Nbr2
Nbr2Lo   DWORD   ?    ;Low order 32 bits of Nbr2
```

Chapter 4 – Basic Instructions

4.4 Addition and Subtraction of Larger Numbers

The following code fragment adds Nbr2 to Nbr1, storing the sum at the doublewords reserved for Nbr1.

```
mov     eax,Nbr1Lo        ;Low order 32 bits of Nbr1
add     eax,Nbr2Lo        ;add Low order 32 bits of Nbr2
mov     Nbr1Lo,eax        ;sum to destination
mov     eax,Nbr1Hi        ;High order 32 bits of Nbr1
adc     eax,Nbr2Hi      ;add High order 32 bits of Nbr2 & carry
mov     Nbr1Hi,eax       ;sum to destination
```

The **adc** instructions are identical to corresponding **add** instructions except that the extra 1 is added if CF is set to 1.

For subtraction, **sbb** (subtract with borrow) instructions function like **sub** instructions except that if CF is set to 1, an extra 1 is subtracted from the difference.

Chapter 4 – Basic Instructions

4.4 Addition and Subtraction of Larger Numbers

Large numbers can be subtracted in groups of bits, working right to left.

Figure 4.21 lists the allowable operand types for adc and sbb instructions. This table is identical to Fig. 4.5 except for a few opcodes.

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode adc	sbb
register 8	immediate 8	2	1	1	3	80	80
register 16	immediate 8	2	1	1	3	83	83
register 32	immediate 8	2	1	1	3	83	83
register 16	immediate 16	2	1	1	4	81	81
register 32	immediate 32	2	1	1	6	81	81
AL	immediate 8	2	1	1	2	14	1C
AX	immediate 16	2	1	1	3	15	1D
EAX	immediate 32	2	1	1	5	15	1D

Chapter 4 – Basic Instructions

4.4 Addition and Subtraction of Larger Numbers

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode add	sub
memory byte	immediate 8	7	3	3	3+	80	80
Memory word	immediate 8	7	3	3	3+	83	83
Memory doubleword	immediate 8	7	3	3	3+	83	83
Memory word	immediate 16	7	3	3	4+	81	81
Memory doubleword	immediate 32	7	3	3	6+	81	81
register 8	register 8	2	1	1	2	12	1A
register 16	register 16	2	1	1	2	13	1B
register 32	register 32	2	1	1	2	13	1B

Chapter 4 – Basic Instructions

4.4 Addition and Subtraction of Larger Numbers

Destination Operand	Source Operand	Clock Cycles 386	Clock Cycles 486	Clock Cycles Pentium	Number of Bytes	opcode add	opcode sub
register 8	memory byte	6	2	2	2+	12	1A
register 16	Memory word	6	2	2	2+	13	1B
register 32	Memory doubleword	6	2	2	2+	13	1B
Memory byte	Register 8	7	3	3	2+	10	18
Memory word	register 16	7	3	3	2+	11	19
Memory doubleword	register 32	7	3	3	2+	11	19

Chapter 4 – Basic Instructions

4.4 Addition and Subtraction of Larger Numbers

Multiplication and division operations with longer numbers are even more involved than addition and subtraction. Often techniques for adding and subtracting longer numbers are used to implement algorithms that are similar to grade school multiplication and division procedures for decimal numbers.

If one really needs to use longer numbers, it takes more than a set of arithmetic procedures. One may also need procedures like itoa and atoi in order to convert long numbers to and from ASCII character format.

Chapter 4 – Basic Instructions

chapter summary

The Intel 80×86 mov instruction is used to copy data from one location to another. All but a few combinations of source and destination locations are allowed. The xchg instruction swaps the data stored at two locations.

The 80×86 architecture has a full set of instructions for arithmetic with byte-length, word-length, and doubleword-length integers. The add and sub instructions perform addition and subtraction; inc and dec add and subtract 1, respectively. The neg instruction negates its operand.

There are two multiplication and two division mnemonics. The imul and idiv instructions assume that their operands are signed 2's complement numbers; mul and div assume that their operands are unsigned. Many multiplication instructions start with single-length operands and produce double-length products; other formats form a product the same length as the factors. Division instructions always start with a double-length dividend and single-length divisor; the outcome is a single-length quotient and a single-length remainder. The cbw, cwd, and cdq instructions aid in producing a double-length dividend before signed division. Flag settings indicate possible errors during multiplication; an error during division produces a hardware exception that invokes a procedure to handle the error.

Chapter 4 – Basic Instructions

chapter summary

Instructions that have operands in registers are generally faster than those that reference memory locations. Multiplication and division instructions are slower than addition and subtraction instructions.

The adc and sbb instructions make it possible to add numbers longer than doublewords a group of bits at a time, incorporating a carry or borrow from one group into the addition or subtraction of the next group to the left. The carry or borrow is recorded in the carry flag CF. The 80×86 clc, stc, and cmc instructions enable the programmer to clear, set, and complement the carry flag when necessary.
The machine language level is just one level of abstraction at which a computer can be viewed. Above this level are the high-level language level and the application level. Below the machine language level are the microcode level and the hardware level.