6

i386 Addressing Modes and Assembler Instructions

This chapter contains information specific to the Intel i386 processor architecture, which includes the i386, i486, and Pentium processors. The first section, ªi386 Registers and Addressing Modes,º lists the registers available and describes the addressing modes used by assembler instructions. The second section, ªi386 Assembler Instructions,º lists each assembler instruction with NeXT assembler syntax.

Note: Don't confuse the i386 *architecture* with the i386 *processor*. NEXTSTEP makes use of instructions specific to the i486 processor, and will not run on an i386 processor.

i386 Registers and Addressing Modes

This section describes the conventions used to specify addressing modes and instruction mnemonics for the Intel i386 processor architecture. The instructions themselves are detailed in the next section, ªi386 Assembler

Instructions.º

Instruction Mnemonics

The instruction mnemonics that the assembler uses are based on the mnemonics described in the relevant Intel processor manuals.

Note: Branch instructions are always long (32 bits) for non-local labels on the NeXT i386 architecture machines. This allows the link editor to do procedure ordering (see the description of the **-sectorder** option in the **ld**(1) man page, and the ªLink Optimizationº paper in the directory **/NextLibrary/Documentation/NextDev/Concepts/Performance**).

Registers

Many instructions accept registers as operands. The available registers are listed in this section. The NeXT assembler for Intel i386 processors always uses names beginning with a percent sign (`%') for registers, so naming conflicts with identifiers aren't possible; further, all register names are in lowercase letters.

General Registers

Each of the 32-bit general registers of the i386 architecture are accessible by different names, which specify parts of that register to be used. For example, the AX register can be accessed as a single byte (**%ah** or **%al**), a 16-bit value (**%ax**), or a 32-bit value (**%eax**). Figure 6-1 shows the names of these registers and their relation to the full 32-bit storage for each register:

F0.eps ,

Figure 6-1

Floating-Point Registers

%st
%st(0)-%st(7)

Segment Registers

%cs	code segment register
%ss	stack segment register
%ds	data segment register
%es	data segment register (string operation destination segment)
%fs	data segment register
%gs	data segment register

Other Registers

%cr0-%cr3	control registers
%db0-%db7	debug registers
%tr3-%tr7	test registers

Operands and Addressing Modes

The i386 architecture uses four kinds of instruction operands:

- Register
- Immediate

- Direct Memory
- Indirect Memory

Each type of operand corresponds to an addressing mode. Register operands specify that the value stored in the named register is to be used by the operator. Immediate operands are constant values specified in assembler code. Direct memory operands are the memory location of labels, or the value of a named register treated as an address. Indirect memory operands are calculated at run time from the contents of registers and optional constant values.

Register Operands

A register operand is given simply as the name of a register. It can be any of the identifiers beginning with `%' listed above; for example, **%eax**. When an operator calls for a register operand of a particular size, the operand is listed as *r8*, *r16*, or *r32*.

Immediate Operands

Immediate operands are specified as numeric values preceded by a dollar sign (`$'). They are decimal by default, but can be marked as hexadecimal by beginning the number itself with `0x'. Simple calculations are allowed if grouped in parentheses. Finally, an immediate operand can be given as a label, in which case its value is the address of that label. Here are some examples:

```
$100
$0x5fec4
$(10*6)     # calculated by the assembler
$begloop
```

A reference to an undefined label is allowed, but that reference must be resolved at link time.

Direct Memory Operands

Direct memory operands are references to labels in assembler source. They act as static references to a single

location in memory relative to a specific segment, and are resolved at link time. Here's an example:

```
        .data
var:    .byte 0         # declare a byte-size variable labelled "var"
        .text
        .
        .
        .
        movb %al,var    # move the low byte of the AX register into the
                        #  memory location specified by "var"
```

By default, direct memory operands use the **%ds** segment register. This can be overridden by prefixing the operands with the segment register desired and a colon:

```
        movb %es:%al,var  # move the low byte of the AX register into the
                          #  memory location in the segment given by %es
                          # and "var"
```

Note that the segment override applies only to the memory operands in an instruction; ªvarº is affected, but not **%al**. The string instructions, which take two memory operands, use the segment override for both. A less common way of indicating a segment is to prefix the operator itself:

```
       es/movb %al,%var   # same as above
```

Indirect Memory Operands

Indirect memory operands are calculated from the contents of registers at run time. An indirect memory operand can contain a base register, and index register, a scale, and a displacement. The most general form is:

*displacement***(***base_register***,***index_register***,***scale***)**

displacement is an immediate value. The base and index registers may be any 32-bit general register names, except that **%esp** can't be used as an index register. *scale* must be 1, 2, 4, or 8; no other values are allowed. The displacement and scale can be omitted, but at least one register must be specified. Also, if items from the end are omitted, the preceding commas can also be omitted, but the comma following an omitted item must remain:

```
10(%eax,%edx)
(%eax)
12(,%ecx,2)
12(,%ecx)
```

The value of an indirect memory operand is the memory location given by the contents of the register, relative to a segment's base address. The segment register used is **%ss** when the base register is **%ebp** or **%esp**, and **%ds** for all other base registers. For example:

```
movl (%eax),%edx    # default segment register here is %ds
```

The above assembler instruction moves 32 bits from the address given by **%eax** into the **%edx** register. The address **%eax** is relative to the **%ds** segment register. A different segment register from the default can be specified by prefixing the operand with the segment register name and a colon (`:'):

```
movl %es:(%eax),%edx
```

A segment override can also be specified as an operator prefix:

```
es/movl (%eax),%edx
```

i386 Assembler Instructions

Note the following points about the information contained in this section:

- **Name** is the name that appears in the upper left corner of a page in the Intel manuals.
- **Operation Name** is the name that appears after the operator name in the Intel manuals. Processor-specific instructions are marked as they occur.
- The form of operands is that used in Intel's *i486 Microprocessor Programmer's Reference Manual*.
- The order of operands is **source** → **destination**, the opposite of the order in Intel's manuals.

Instructions

Name	Operator	Operand	Operation Name
aaa	aaa		ASCII Adjust after Addition
aad	aad		ASCII Adjust AX before Division
aam	aam		ASCII Adjust AX after Division
aas	aas		ASCII Adjust AL after Subtraction
adc	adc	$*imm8,r/m8*	Add with Carry
	adc	$*imm16,r/m*16	
	adc	$*imm32,r/m32*	
	adc	$*imm8,r/m16*	
	adc	$*imm8,r/m32*	
	adc	*r8,r/m8*	
	adc	*r16,r/m16*	
	adc	*r32,r/m32*	
	adc	*r/m8,r8*	
	adc	*r/m16,r16*	
	adc	*r/m32,r32*	

add	add	$*imm8,r/m8*	Add
	add	$*imm16,r/m*16	
	add	$*imm32,r/m32*	
	add	$*imm8,r/m16*	
	add	$*imm8,r/m32*	
	add	*r8,r/m8*	
	add	*r16,r/m16*	
	add	*r32,r/m32*	
	add	*r/m8,r8*	
	add	*r/m16,r16*	
	add	*r/m32,r32*	
and	and	$*imm8,r/m8*	Logical AND
	and	$*imm16,r/m*16	
	and	$*imm32,r/m32*	
	and	$*imm8,r/m16*	
	and	$*imm8,r/m32*	
	and	*r8,r/m8*	
	and	*r16,r/m16*	
	and	*r32,r/m32*	
	and	*r/m8,r8*	
	and	*r/m16,r16*	
	and	*r/m32,r32*	
arpl	arpl	*r16,r/m16*	Adjust RPL Field of Selector
bound	bound	*m16&16,r16*	Check Array Index Against Bounds
	bound	*m32&32,r32*	

bsf	bsf	*r/m16,r16*	Bit Scan Forward
	bsf	*r/m32,r16*	
bsr	bsr	*r/m16,r16*	Bit Scan Reverse
	bsr	*r/m32,r16*	
bswap	bswap	*r32*	Byte Swap (i486-specific)
bt	bt	*r16,r/m16*	Bit Test
	bt	*r32,r/m32*	
	bt	$*imm8,r/m16*	
	bt	$*imm8,r/m32*	
btc	btc	*r16,r/m16*	Bit Test and Complement
	btc	*r32,r/m32*	
	btc	$*imm8,r/m16*	
	btc	$*imm8,r/m32*	
btr	btr	*r16,r/m16*	Bit Test and Reset
	btr	*r32,r/m32*	
	btr	$*imm8,r/m16*	
	btr	$*imm8,r/m32*	
bts	bts	*r16,r/m16*	Bit Test and Set
	bts	*r32,r/m32*	
	bts	$*imm8,r/m16*	
	bts	$*imm8,r/m32*	

call	call	*rel16*	Call Procedure
	call	*r/m16*	
	call	*ptr16:16*	
	call	*m16:16*	
	call	*rel32*	
	call	*r/m32*	
	lcall	$*imm16,*$*imm32*	
	lcall	*m16*	
	lcall	*m32*	
cbw/cwde	cbw		Convert Byte to Word/ Convert Word to Doubleword
	cwde		
clc	clc		Clear Carry Flag
cld	cld		Clear Direction Flag
cli	cli		Clear Interrupt Flag
clts	clts		Clear Task-Switched Flag in CR0
cmc	cmc		Complement Carry Flag
cmp	cmp	$*imm8,r/m8*	Compare Two Operands
	cmp	$*imm16,r/m*16	
	cmp	$*imm32,r/m32*	
	cmp	$*imm8,r/m16*	
	cmp	$*imm8,r/m32*	

	cmp	*r8,r/m8*	
	cmp	*r16,r/m16*	
	cmp	*r32,r/m32*	
	cmp	*r/m8,r8*	
	cmp	*r/m16,r16*	
	cmp	*r/m32,r32*	
cmps/cmpsb/cmpsw/cmpsd			Compare String Operands
	cmps	*m8,m8*	
	cmps	*m16,m16*	
	cmps	*m32,m32*	
	cmpsb		
	cmpsw		
	cmpsd		
	(optional forms with segment override)		
	cmpsb	*%seg***:0(%esi),%es:0(%edi)**	
	cmpsw	*%seg***:0(%esi),%es:0(%edi)**	
	cmpsd	*%seg***:0(%esi),%es:0(%edi)**	
cmpxchg	cmpxchg	*r8,r/m8*	Compare and Exchange (i486-specific)
	cmpxchg	*r16,r/m16*	
	cmpxchg	*r32,r/m32*	
cmpxchg8b	cmpxchg8b	*m32*	Compare and Exchange 8 Bytes (Pentium-specific)
cpuid	cpuid		CPU Identification

			(Pentium-specific)
cwd/cdq	cwd cdq		Convert Word to Doubleword/ Convert Doubleword to Quadword
daa	daa		Decimal Adjust AL after Addition
das	das		Decimal Adjust AL after Subtraction
dec	dec dec dec dec dec	*r/m8* *r/m16* *r/m32* *r16* *r32*	Decrement by 1
div	div div div	*r/m8,***%al** *r/m16,***%ax** *r/m32,***%eax**	Unsigned Divide
enter	enter	$*imm16,*$*imm8*	Make Stack Frame for Procedure Parameters
f2xm1	f2xm1		Computer 2^X-1
fabs	fabs		Absolute Value

fadd/faddp/fiadd			Add
	fadd	*m32real*	
	fadd	*m64real*	
	fadd	ST(i),ST	
	fadd	ST,ST(i)	
	faddp	ST,ST(i)	
	fadd		
	fiadd	*m32int*	
	fiadd	*m16int*	
fbld	fbld	*m80dec*	Load Binary Coded Decimal
fbstp	fbstp	*m80dec*	Store Binary Coded Decimal and Pop
fchs	fchs		Change Sign
fclex/fnclex			Clear Exceptions
	fclex		
	fnclex		
fcom/fcomp/fcompp			Compare Real
	fcom	*m32real*	
	fcom	*m64real*	
	fcom	ST(i)	
	fcom		
	fcomp	*m32real*	
	fcomp	*m64real*	

	fcomp	ST(i)	
	fcomp		
	fcompp		
fcos	fcos		Cosine
fdecstp	fdecstp		Decrement Stack-Top Pointer
fdiv/fdivp/fidiv			Divide
	fdiv	*m32real*	
	fdiv	*m64real*	
	fdiv	ST(i),ST	
	fdiv	ST,ST(i)	
	fdivp	ST,ST(i)	
	fdiv		
	fidiv	*m32int*	
	fidiv	*m16int*	
fdivr/fdivpr/fidivr			Reverse Divide
	fdivr	*m32real*	
	fdivr	*m64real*	
	fdivr	ST(i),ST	
	fdivr	ST,ST(i)	
	fdivrp	ST,ST(i)	
	fdivr		
	fidivr	*m32int*	
	fidivr	*m16int*	

ffree	ffree	ST(i)	Free Floating-Point Register
ficom/ficomp			Compare Integer
	ficom	*m16real*	
	ficom	*m32real*	
	ficomp	*m16int*	
	ficomp	*m32int*	
fild	filds	*m16int*	Load Integer
	fildl	*m32int*	
	fildq	*m64int*	
fincstp	fincstp		Increment Stack-Top Pointer
finit/fninit	finit		Initialize Floating-Point Unit
	fninit		
fist/fistp	fists	*m16int*	Store Integer
	fistl	*m32int*	
	fistps	*m16int*	
	fistpl	*m32int*	
	fistpq	*m64int*	
fld	flds	*m32real*	Load Real
	fldl	*m64real*	
	fldt	*m80real*	
	fld	ST(i)	
fld1/fldl2t/fldl2e/fldpi/fldlg2/gldln2/fldz			Load Constant

	fld1		
	fld2t		
	fld2e		
	fldpi		
	fldlg2		
	fldln2		
	fldz		
fldcw	fldcw	*m2byte*	Load Control Word
fldenv	fldenv	*m14/28byte*	Load FPU Environment
fmul/fmulp/fimul			Multiply
	fmul	*m32real*	
	fmul	*m64real*	
	fmul	ST(i),ST	
	fmul	ST(i),ST	
	fmulp	ST,ST(i)	
	fmul		
	fimul	*m32int*	
	fimul	*m16int*	
fnop	fnop		No Operation
fpatan	fpatan		Partial Arctangent
fprem	fprem		Partial Remainder

fprem1	fprem1		Partial Remainder
fptan	fptan		Partial Tangent
frndint	frndint		Round to Integer
frstor	frstor	*m94/108byte*	Restore FPU State
fsave/fnsave			Store FPU State
	fsave	*m94/108byte*	
	fnsave	*m94/108byte*	
fscale	fscale		Scale
fsin	fsin		Sine
fsincos	fsincos		Sine and Cosine
fsqrt	fsqrt		Square Root
fst/fstp	fst	*m32real*	Store Real
	fst	*m64real*	
	fst	ST(i)	
	fstp	*m32real*	
	fstp	*m64real*	
	fstp	*m80real*	
	fstp	ST(i)	
fstcw/fnstcw			Store Control Word

	fstcw	*m2byte*
	fnstcw	*m2byte*
fstenv/fnstenv		Store FPU Environment
	fstenv	*m14/28byte*
	fnstenv	*m14/28byte*
fstsw/fnstsw		Store Status Word
	fstsw	*m2byte*
	fstsw	**%ax**
	fnstsw	*m2byte*
	fnstsw	**%ax**
fsub/fsubp/fisub		Subtract
	fsub	*m32real*
	fsub	*m64real*
	fsub	ST(i),ST
	fsub	ST,ST(i)
	fsubp	ST,ST(i)
	fsub	
	fisub	*m32int*
	fisub	*m16int*
fsubr/fsubpr/fisubr		Reverse Subtract
	fsubr	*m32real*
	fsubr	*m64real*
	fsubr	ST(i),ST
	fsubr	ST,ST(i)

	fsubpr	ST,ST(i)	
	fsubr		
	fisubr	*m32int*	
	fisubr	*m16int*	
ftst	ftst		Test
fucom/fucomp/fucompp			Unordered Compare Real
	fucom	ST(i)	
	fucom		
	fucomp	ST(i)	
	fucomp		
	fucompp		
fwait	fwait		Wait
fxam	fxam		Examine
fxch	fxch	ST(i)	Exchange Register Contents
	fxch		
fxtract	fxtract		Extract Exponent and Significand
fyl2x	fyl2x		Compute y€×€$\log_2 x$
fyl2xp1	fyl2xp1		Compute y€×€$\log_2(x+1)$

hlt	hlt		Halt
idiv	idiv	*r/m8*	Signed Divide
	idiv	*r/m16*,**%ax**	
	idiv	*r/m32*,**%eax**	
imul	imul	*r/m8*	Signed Multiply
	imul	*r/m16*	
	imul	*r/m32*	
	imul	*r/m16,r16*	
	imul	*r/m32,r32*	
	imul	$*imm8,r/m16,r16*	
	imul	$*imm8,r/m32,r32*	
	imul	$*imm8,r16*	
	imul	$*imm8,r32*	
	imul	$*imm16,r/m16,r16*	
	imul	$*imm32,r/m32,r32*	
	imul	$*imm16,r16*	
	imul	$*imm32,r32*	
in	in	$*imm8,***%al**	Input from Port
	in	$*imm8,***%ax**	
	in	$*imm8,***%eax**	
	in	**%dx**,**%al**	
	in	**%dx**,**%ax**	
	in	**%dx**,**%eax**	
inc	inc	*r/m8*	Increment by 1
	inc	*r/m16*	

	inc	*r/m32*	
	inc	*r16*	
	inc	*r32*	
ins/insb/insw/insd			Input from Port to String
	ins		
	insb		
	insw		
	insd		
int/into	int	3	Call to Interrupt Procedure
	int	$*imm8*	
	into		
invd	invd		Invalidate Cache (i486-specific)
invlpg	invlpg	m	Invalidate TLB Entry (i486-specific)
iret/iretd	iret		Interrupt Return
	iretd		
j*cc*			Jump if Condition is Met
	ja	*rel8*	short if above
	jae	*rel8*	short if above or equal
	jb	*rel8*	short if below
	jbe	*rel8*	short if below or equal
	jc	*rel8*	short if carry

jcxz	*rel8*	short if **%cx** register is 0
jecxz	*rel8*	short if **%ecx** register is 0
je	*rel8*	short if equal
jz	*rel8*	short if 0
jg	*rel8*	short if greater
jge	*rel8*	short if greater or equal
jl	*rel8*	short if less
jle	*rel8*	short if less or equal
jna	*rel8*	short if not above
jnae	*rel8*	short if not above or equal
jnb	*rel8*	short if not below
jnbe	*rel8*	short if not below or equal
jnc	*rel8*	short if not carry
jne	*rel8*	short if not equal
jng	*rel8*	short if not greater
jnge	*rel8*	short if not greater or equal
jnl	*rel8*	short if not less
jnle	*rel8*	short if not less or equal
jno	*rel8*	short if not overflow
jnp	*rel8*	short if not parity
jns	*rel8*	short if not sign
jnz	*rel8*	short if not 0
jo	*rel8*	short if overflow
jp	*rel8*	short if parity
jpe	*rel8*	short if parity even
jpo	*rel8*	short if parity odd
js	*rel8*	short if sign
jz	*rel8*	short if zero
ja	*rel16/32*	near if above

jae	*rel16/32*	near if above or equal
jb	*rel16/32*	near if below
jbe	*rel16/32*	near if below or equal
jc	*rel16/32*	near if carry
je	*rel16/32*	near if equal
jz	*rel16/32*	near if 0
jg	*rel16/32*	near if greater
jge	*rel16/32*	near if greater or equal
jl	*rel16/32*	near if less
jle	*rel16/32*	near if less or equal
jna	*rel16/32*	near if not above
jnae	*rel16/32*	near if not above or equal
jnb	*rel16/32*	near if not below
jnbe	*rel16/32*	near if not below or equal
jnc	*rel16/32*	near if not carry
jne	*rel16/32*	near if not equal
jng	*rel16/32*	near if not greater
jnge	*rel16/32*	near if not greater or less
jnl	*rel16/32*	near if not less
jnle	*rel16/32*	near if not less or equal
jno	*rel16/32*	near if not overflow
jnp	*rel16/32*	near if not parity
jns	*rel16/32*	near if not sign
jnz	*rel16/32*	near if not 0
jo	*rel16/32*	near if overflow
jp	*rel16/32*	near if parity
jpe	*rel16/32*	near if parity even
jpo	*rel16/32*	near if parity odd

	js	*rel16/32*	near if sign
	jz	*rel16/32*	near if 0
jmp	jmp	*rel8*	Jump
	jmp	*rel16*	
	jmp	*r/m16*	
	jmp	*rel32*	
	jmp	*r/m32*	
	ljmp	$*imm16*,$*imm32*	
	ljmp	*m16*	
	ljmp	*m32*	
lahf	lahf		Load Flags into AH Register
lar	lar	*r/m16,r16*	Load Access Rights Byte
	lar	*r/m32,r32*	
lea	lea	*m,r16*	Load Effective Address
	lea	*m,r32*	
leave	leave		High Level Procedure Exit
lgdt/lidt	lgdt	*m16&32*	Load Global/Interrupt
	lidt	*m16&32*	Descriptor Table Register
lgs/lss/lds/les/lfs			Load Full Pointer
	lgs	*m16:16,r16*	
	lgs	*m16:32,r32*	
	lss	*m16:16,r16*	

	lss	*m16:32,r32*	
	lds	*m16:16,r16*	
	lds	*m16:32,r32*	
	les	*m16:16,r16*	
	les	*m16:32,r32*	
	lfs	*m16:16,r16*	
	lfs	*m16:32,r32*	
lldt	lldt	*r/m16*	Load Local Descriptor Table Register
lmsw	lmsw	*r/m16*	Load Machine Status Word
lock	lock		Assert LOCK# Signal Prefix
lods/lodsb/lodsw/lodsd			Load String Operand
	lods	*m8*	
	lods	*m16*	
	lods	*m32*	
	lodsb		
	lodsw		
	lodsd		
	(optional forms with segment override)		
	lodsb	*%seg***:0(%esi),%al**	
	lodsw	*%seg***:0(%esi),%al**	
	lodsd	*%seg***:0(%esi),%al**	

loop/loop*cond*			Loop Control with CX Counter
	loop	*rel8*	
	loope	*rel8*	
	loopz	*rel8*	
	loopne	*rel8*	
	loopnz	*rel8*	
lsl	lsl	*r/m16,r16*	Load Segment Limit
	lsl	*r/m32,r32*	
ltr	ltr	*r/m16*	Load Task Register
mov	mov	*r8,r/m8*	Move Data
	mov	*r16,r/m16*	
	mov	*r32,r/m32*	
	mov	*r/m8,r8*	
	mov	*r/m16,r16*	
	mov	*r/m16,r16*	
	mov	*Sreg,r/m16*	
	mov	*r/m16,Sreg*	
	mov	*moffs8,***%al**	
	mov	*moffs8,***%ax**	
	mov	*moffs8,***%eax**	
	mov	**%al**,*moffs8*	
	mov	**%ax**,*moffs16*	
	mov	**%eax**,*moffs32*	
	mov	$*imm8,reg8*	
	mov	$*imm16,reg*16	
	mov	$*imm32,reg32*	

	mov	$*imm8,r/m8*	
	mov	$*imm16,r/m*16	
	mov	$*imm32,r/m32*	
mov	mov	*r32,***%cr0**	Move to/from Special Registers
	mov	**%cr0**/**%cr2**/**%cr3**,*r32*	
	mov	**%cr2**/**%cr3**,*r32*	
	mov	**%dr0-3**,*r32*	
	mov	**%dr6**/**%dr7**,*r32*	
	mov	*r32,***%dr0-3**	
	mov	*r32,***%dr6**/**%dr7**	
	mov	**%tr4**/**%tr5**/**%tr6**/**%tr7**,*r32*	
	mov	*r32***,%tr4**/**%tr5**/**%tr6**/**%tr7**	
	mov	**%tr3**,*r32*	
	mov	*r32***,%tr3**	
movs/movsb/movsw/movsd			Move Data from String to String
	movs	*m8,m8*	
	movs	*m16,m16*	
	movs	*m32,m32*	
	movsb		
	movsw		
	movsd		
	(optional forms with segment override)		
	movsb	*%seg***:0(%esi),%es:0(%edi)**	
	movsw	*%seg***:0(%esi),%es:0(%edi)**	
	movsd	*%seg***:0(%esi),%es:0(%edi)**	

movsx	movsx	*r/m8,r16*	Move with Sign-Extend
	movsx	*r/m8,r32*	
	movsx	*r/m16,r32*	
movzx	movzx	*r/m8,r16*	Move with Zero-Extend
	movzx	*r/m8,r32*	
	movzx	*r/m16,r32*	
mul	mul	*r/m8,***%al**	Unsigned Multiplication of AL
	mul	*r/m16,***%ax**	or AX
	mul	*r/m32,***%eax**	
neg	neg	*r/m8*	Two's Complement Negation
	neg	*r/m16*	
	neg	*r/m32*	
nop	nop		No Operation
not	not	*r/m8*	One's Complement Negation
	not	*r/m16*	
	not	*r/m32*	
or	or	$*imm8,r/m8*	Logical Inclusive OR
	or	$*imm16,r/m*16	
	or	$*imm32,r/m32*	
	or	$*imm8,r/m16*	
	or	$*imm8,r/m32*	
	or	*r8,r/m8*	

	or	*r16,r/m16*	
	or	*r32,r/m32*	
	or	*r/m8,r8*	
	or	*r/m16,r16*	
	or	*r/m32,r32*	
out	out	**%al**,$*imm8*	Output to Port
	out	**%ax**,$*imm8*	
	out	**%eax**,$*imm8*	
	out	**%al**,**%dx**	
	out	**%ax**,**%dx**	
	out	**%eax**,**%dx**	
outs/outsb/outsw/outsd			Output String to Port
	outs	*r/m8,***%dx**	
	outs	*r/m16,***%dx**	
	outs	*r/m32,***%dx**	
	outsb		
	outsw		
	outsd		
pop	pop	*m16*	Pop a Word from the Stack
	pop	*m32*	
	pop	*r16*	
	pop	*r32*	
	pop	**%ds**	
	pop	**%es**	
	pop	**%ss**	

	pop	**%fs**	
	pop	**%gs**	
popa/popad			Pop all General Registers
	popa		
	popad		
popf/popfd	popf		Pop Stack into FLAGS or EFLAGS Register
	popfd		
push	push	*m16*	Push Operand onto the Stack
	push	*m32*	
	push	*r16*	
	push	*r32*	
	push	$*imm8*	
	push	$*imm16*	
	push	$*imm32*	
	push	*Sreg*	
pusha/pushad			Push all General Registers
	pusha		
	pushad		
pushf/pushfd			Push Flags Register onto the Stack
	pushf		
	pushfd		
rcl/rcr/rol/ror			Rotate
	rcl	1,*r/m8*	

rcl	**%cl**,*r/m8*
rcl	$*imm8,r/m8*
rcl	1,*r/m16*
rcl	**%cl**,*r/m16*
rcl	$*imm8,r/m16*
rcl	1,*r/m32*
rcl	**%cl**,*r/m32*
rcl	$*imm8,r/m32*
rcr	1,*r/m8*
rcr	**%cl**,*r/m8*
rcr	$*imm8,r/m8*
rcr	1,*r/m16*
rcr	**%cl**,*r/m16*
rcr	$*imm8,r/m16*
rcr	1,*r/m32*
rcr	**%cl**,*r/m32*
rcr	$*imm8,r/m32*
rol	1,*r/m8*
rol	**%cl**,*r/m8*
rol	$*imm8,r/m8*
rol	1,*r/m16*
rol	**%cl**,*r/m16*
rol	$*imm8,r/m16*
rol	1,*r/m32*
rol	**%cl**,*r/m32*
rol	$*imm8,r/m32*
ror	1,*r/m8*
ror	**%cl**,*r/m8*

	ror	$*imm8,r/m8*	
	ror	1,*r/m16*	
	ror	**%cl**,*r/m16*	
	ror	$*imm8,r/m16*	
	ror	1,*r/m32*	
	ror	**%cl**,*r/m32*	
	ror	$*imm8,r/m32*	
rdmsr	rdmsr		Read from Model-Specific Register (Pentium-specific)
rdstc	rdstc		Read from Time Stamp Counter (Pentium-specific)
rep/repe/repz/repne/repnz			Repeat Following String Operation
	rep ins	**%dx**,*rm8*	
	rep ins	**%dx**,*rm16*	
	rep ins	**%dx**,*rm32*	
	rep movs	*m8,m8*	
	rep movs	*m16,m16*	
	rep movs	*m32,m32*	
	rep outs	*rm8,***%dx**	
	rep outs	*rm16,***%dx**	
	rep outs	*rm32,***%dx**	
	rep lods	*m8*	
	rep lods	*m16*	
	rep lods	*m32*	
	rep stos	*m8*	
	rep stos	*m16*	

	rep stos	*m32*	
	repe cmps	*m8,m8*	
	repe cmps	*m16,m16*	
	repe cmps	*m32,m32*	
	repe scas	*m8*	
	repe scas	*m16*	
	repe scas	*m32*	
	repne cmps	*m8,m8*	
	repne cmps	*m16,m16*	
	repne cmps	*m32,m32*	
	repne scas	*m8*	
	repne scas	*m16*	
	repne scas	*m32*	
ret	ret		Return from Procedure
	ret	$*imm16*	
rsm	rsm		Resume from System-Management Mode (Pentium-specific)
sahf	sahf		Store AH into Flags
sal/sar/shl/shr			Shift Instructions
	sal	1,*r/m8*	
	sal	**%cl**,*r/m8*	
	sal	$*imm8,r/m8*	
	sal	1,*r/m16*	

sal	**%cl**,*r/m16*
sal	$*imm8,r/m16*
sal	1,*r/m32*
sal	**%cl**,*r/m32*
sal	$*imm8,r/m32*
sar	1,*r/m8*
sar	**%cl**,*r/m8*
sar	$*imm8,r/m8*
sar	1,*r/m16*
sar	**%cl**,*r/m16*
sar	$*imm8,r/m16*
sar	1,*r/m32*
sar	**%cl**,*r/m32*
sar	$*imm8,r/m32*
shl	1,*r/m8*
shl	**%cl**,*r/m8*
shl	$*imm8,r/m8*
shl	1,*r/m16*
shl	**%cl**,*r/m16*
shl	$*imm8,r/m16*
shl	1,*r/m32*
shl	**%cl**,*r/m32*
shl	$*imm8,r/m32*
shr	1,*r/m8*
shr	**%cl**,*r/m8*
shr	$*imm8,r/m8*
shr	1,*r/m16*
shr	**%cl**,*r/m16*
shr	$*imm8,r/m16*

	shr	1,*r/m32*	
	shr	**%cl**,*r/m32*	
	shr	$*imm8,r/m32*	
sbb	sbb	$*imm8,r/m8*	Integer Subtraction with Borrow
	sbb	$*imm16,r/m*16	
	sbb	$*imm32,r/m32*	
	sbb	$*imm8,r/m16*	
	sbb	$*imm8,r/m32*	
	sbb	*r8,r/m8*	
	sbb	*r16,r/m16*	
	sbb	*r32,r/m32*	
	sbb	*r/m8,r8*	
	sbb	*r/m16,r16*	
	sbb	*r/m32,r32*	
scas/scasb/scasw/scasd			Compare String Data
	scas	*m8*	
	scas	*m16*	
	scas	*m32*	
	scasb		
	scasw		
	scasd		
	(optional forms with segment override)		
	scasb	**%al,***%seg***:0(%edi)**	
	scasw	**%ax,***%seg***:0(%edi)**	
	scasd	**%eax,***%seg***:0(%edi)**	

set*cc*			Byte Set on Condition
	seta	*r/m8*	above
	setae	*r/m8*	above or equal
	setb	*r/m8*	below
	setbe	*r/m8*	below or equal
	setc	*r/m8*	carry
	sete	*r/m8*	equal
	setg	*r/m8*	greater
	setge	*r/m8*	greater or equal
	setl	*r/m8*	less
	setle	*r/m8*	less or equal
	setna	*r/m8*	not above
	setnae	*r/m8*	not abover or equal
	setnb	*r/m8*	not below
	setnbe	*r/m8*	not below or equal
	setnc	*r/m8*	not carry
	setne	*r/m8*	not equal
	setng	*r/m8*	not greater
	setnge	*r/m8*	not greater or equal
	setnl	*r/m8*	not less
	setnle	*r/m8*	not less or equal
	setno	*r/m8*	not overflow
	setnp	*r/m8*	not parity
	setns	*r/m8*	not sign
	setnz	*r/m8*	not zero
	seto	*r/m8*	overflow
	setp	*r/m8*	parity
	setpe	*r/m8*	parity even

	setpo	*r/m8*	parity odd
	sets	*r/m8*	sign
	setz	*r/m8*	zero
sgdt/sidt	sgdt	*m*	Store Global/Interrupt
	sidt	*m*	Descriptor Table Register
shld	shld	$*imm8,r16,r/m16*	Double Precision Shift Left
	shld	$*imm8,r32,r/m32*	
	shld	**%cl**,*r16,r/m16*	
	shld	**%cl**,*r32,r/m32*	
shrd	shrd	$*imm8,r16,r/m16*	Double Precision Shift Right
	shrd	$*imm8,r32,r/m32*	
	shrd	**%cl**,*r16,r/m16*	
	shrd	**%cl**,*r32,r/m32*	
sldt	sldt	*r/m16*	Store Local Descriptor Table Register
smsw	smsw	*r/m16*	Store Machine Status Word
stc	stc		Set Carry Flag
std	std		Set Direction Flag
sti	sti		Set Interrupt Flag

stos/stosb/stosw/stosd			Store String Data
	stos	*m8*	
	stos	*m16*	
	stos	*m32*	
	stosb		
	stosw		
	stosd		
	(optional forms with segment override)		
	stosb	**%al,***%seg***:0(%edi)**	
	stosw	**%ax,***%seg***:0(%edi)**	
	stosd	**%eax,***%seg***:0(%edi)**	
str	str	*r/m16*	Store Task Register
sub	sub	$*imm8,r/m8*	Integer Subtraction
	sub	$*imm16,r/m*16	
	sub	$*imm32,r/m32*	
	sub	$*imm8,r/m16*	
	sub	$*imm8,r/m32*	
	sub	*r8,r/m8*	
	sub	*r16,r/m16*	
	sub	*r32,r/m32*	
	sub	*r/m8,r8*	
	sub	*r/m16,r16*	
	sub	*r/m32,r32*	
test	test	$*imm8,r/m8*	Logical Compare
	test	$*imm16,r/m*16	

	test	$*imm32,r/m32*	
	test	*r8,r/m8*	
	test	*r16,r/m16*	
	test	*r32,r/m32*	
verr, verw	verr	*r/m16*	Verify a Segment for Reading or Writing
	verw	*r/m16*	
wait	wait		Wait
wbinvd	wbinvd		Write-Back and Invalidate Cache (i486-specific)
wrmsr	wrmsr		Write to Model-Specific Register (Pentium-specific)
xadd	xadd	*r8,r/m8*	Exchange and Add (i486-specific)
	xadd	*r16,r/m16*	
	xadd	*r32,r/m32*	
xchg	xchg	*r16,***%ax**	Exchange Register/Memory with Register
	xchg	**%ax**,*r16*	
	xchg	**%eax**,*r32*	
	xchg	*r32,***%eax**	
	xchg	*r8,r/m8*	
	xchg	*r/m8,r8*	
	xchg	*r16,r/m16*	
	xchg	*r/m16,r16*	

	xchg	*r32,r/m32*	
	xchg	*r/m32,r32*	
xlat/xlatb	xlat	*m8*	Table Look-up Translation
	xlatb		
xor	xor	$*imm8,r/m8*	Logical Exclusive OR
	xor	$*imm16,r/m*16	
	xor	$*imm32,r/m32*	
	xor	$*imm8,r/m16*	
	xor	$*imm8,r/m32*	
	xor	*r8,r/m8*	
	xor	*r16,r/m16*	
	xor	*r32,r/m32*	
	xor	*r/m8,r8*	
	xor	*r/m16,r16*	
	xor	*r/m32,r32*	