Tec de Monterrey
CAMPUS ESTADO DE MEXICO

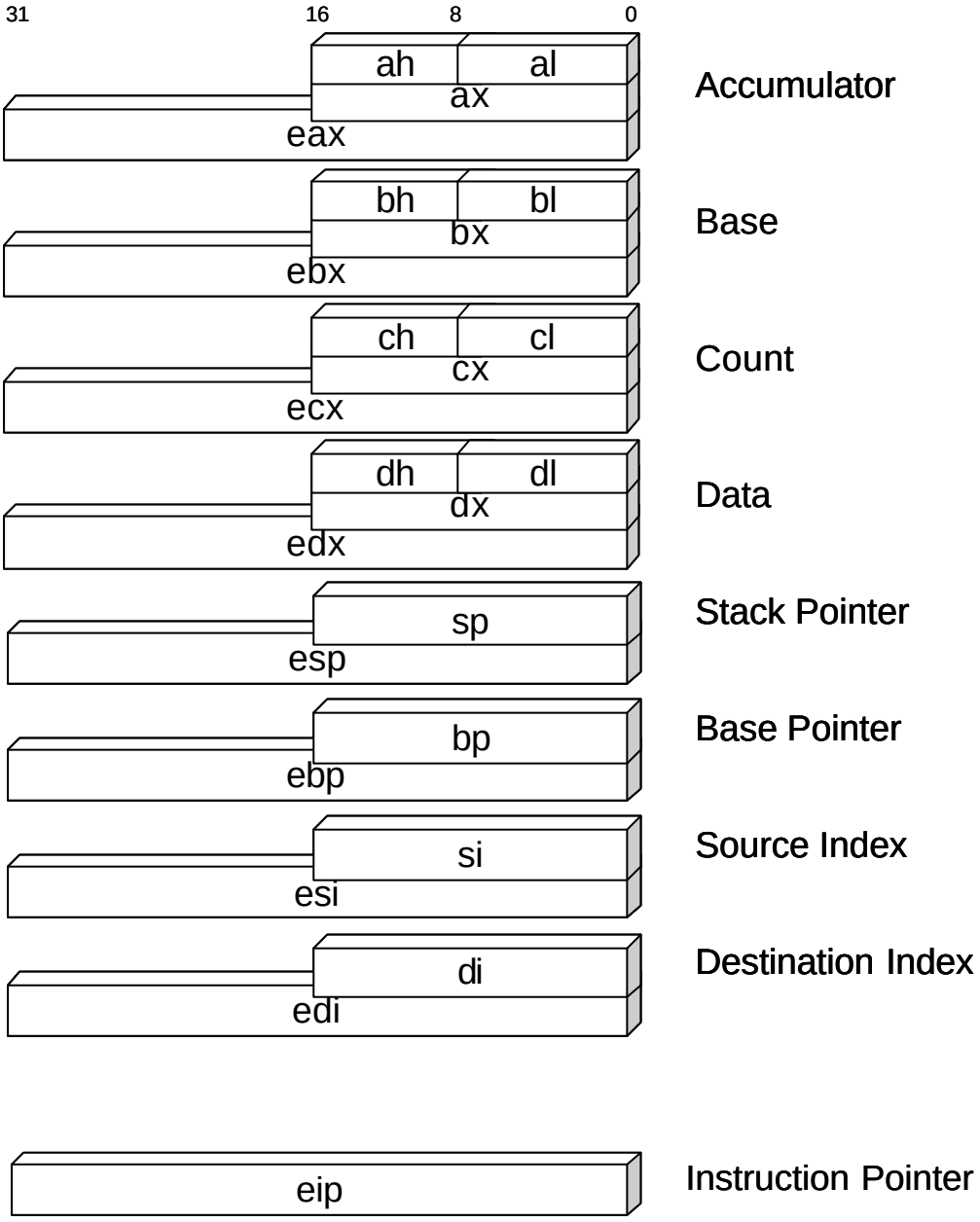
Assembly Language

Course Notes

Cortesía: MSc. Ariel Ortiz

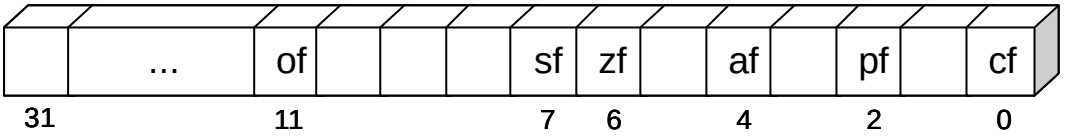
January 2003

Registers



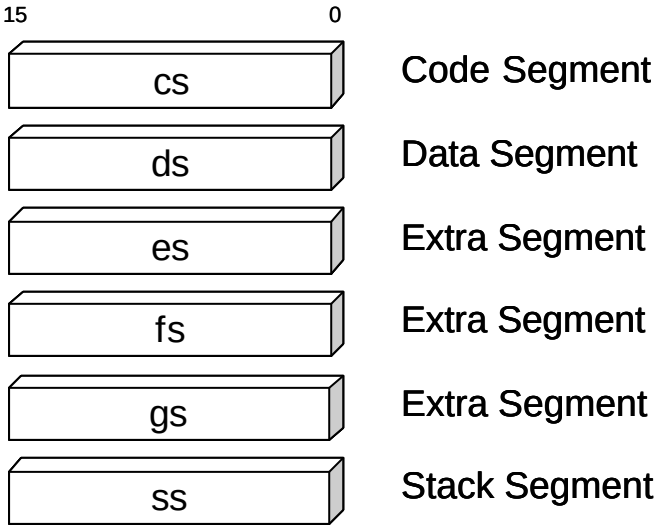
Flags Register

eflags



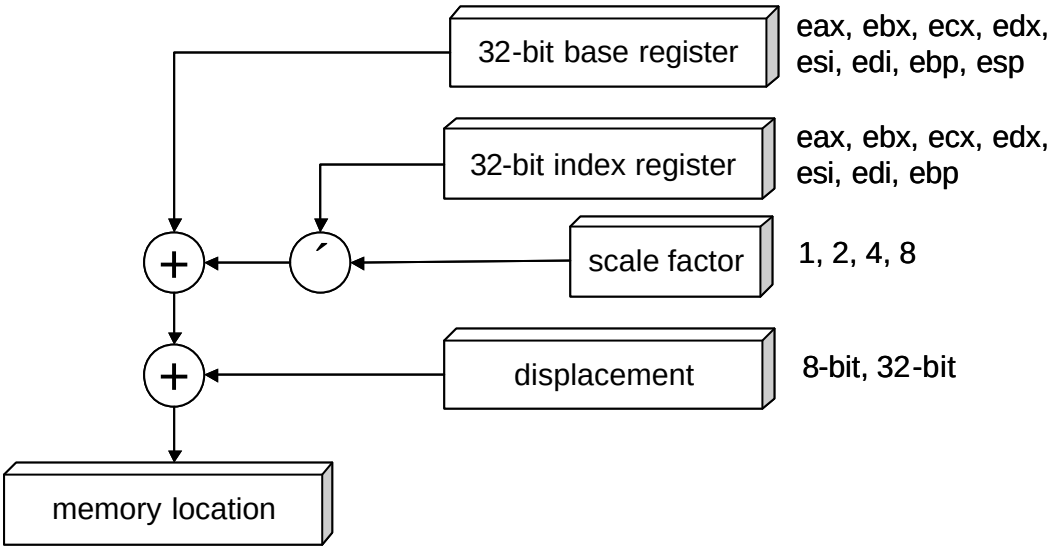
Flag	*Name*	*Purpose*
cf	Carry Flag	Is set if the result of an arithmetic operation involving unsigned numbers overflows.
of	Overflow Flag	Is set if the result of an arithmetic operation involving signed numbers overflows.
sf	Sign Flag	Is set if the result of an arithmetic or logical operation is negative.
zf	Zero Flag	Is set if the result of an arithmetic or logical operation is zero.
pf	Parity Flag	Is set if the result of an arithmetic or logical operation has an even number of 1 bits in its 8 least significant bits.
af	Auxiliary Flag	Is set if the result of an arithmetic operation has a carry out from the low-order nibble. Used in binary-coded decimal (BCD) operations.

Segment Registers



Memory Locations

[*base + index * scale + displacement*]



C Linkage Convention

- When combining C++ and assembly language modules, it is important to follow the *C Linkage Convention*.
- The C Linkage Convention establishes the machine registers usage, the layout of arguments put on the stack, the layout of built-in types such as integers and floats, the form of names passed by the compiler to the linker, and the amount of type checking required from the linker.
- There also is another linkage convention called the *C++ linkage convention*, which is the default used by C++ compilers and linkers. This convention depends on a technique called *name mangling*, which consists on applying a special algorithm to modify public names, thus permitting overloading. Because different C++ compilers apply different mangling algorithms, in general it's better to stick with the C linkage convention when mixing C++ and other languages.
- In C++, the C Linkage Convention is achieved by using the `extern "C"` directive. Example:

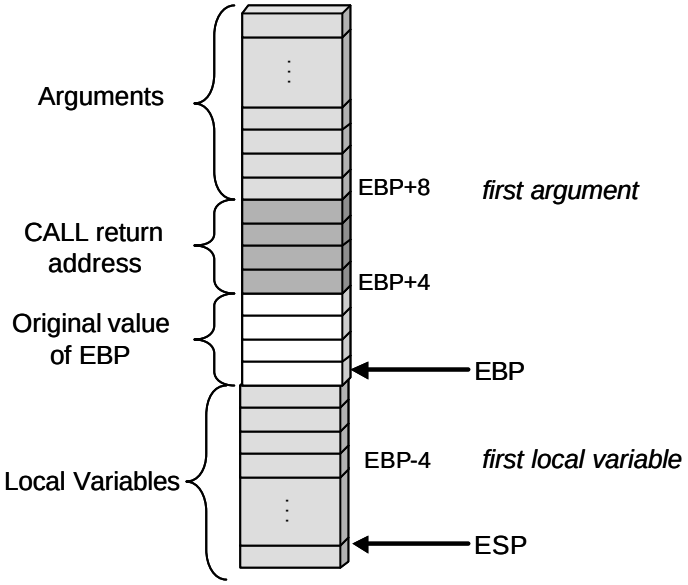
  ```
  extern "C" int foo(int x);
  ```

- NASM labels are exported to other modules using the GLOBAL directive. Symbols defined elsewhere may be imported to a NASM module using the EXTERN directive. All other labels are local to the NASM source file.
- In the Windows platform, all names in an assembly language source file must have an underscore (_) prefix if they refer to names declared in a C++ source file. This is not so in the Linux platform, where no prefix is required.
- When calling a function, arguments are pushed to the stack in reverse order, that is, from right to left as they syntactically appear in the C++ source. Once the arguments are in the stack, a CALL instruction to the desired function is executed. When the function returns, the arguments are still in the stack and must be removed. Adjusting the ESP register through an ADD instruction does the trick.
- A called function must preserve the original values of following registers: EBX, EBP, ESI and EDI. All other registers (including EFLAGS, FPU, MMX, and SSE registers) may be freely modified by the called function.
- The **function prologue** should be as follows:

  ```
  push    ebp
  mov     ebp, esp
  ```

  This preserves the value of the EBP register, so that it can now point to the current top of stack.
- Space for local variables may be allocated by subtracting to ESP the number of bytes required.

- At the beginning of the function execution, the stack has the following layout:



- Once the called function ends, the following **function epilogue** should be executed to undo the actions of the function prologue:

```
pop     ebp
ret
```

- The C++/NASM data types counterparts are summarized in the following table:

C++ Data Type	*Size (bits)*	*NASM Data Type*
`char`	8	BYTE
`short`	16	WORD
`int`	32	DWORD
`long`	32	DWORD
`void*` (pointer to any type)	32	DWORD
`float`	32	DWORD
`double`	64	QWORD
`long double`	80	TWORD

- Function return values are placed in the following registers:

C++ Data Type	*Register*
`char`, `short`, `int`, `long`, and `void*`	EAX
`float`, `double`, and `long double`	ST0

Integer Operations

Data Transfers

Instruction	*Operation*	*Notes*
MOV *dest*, *orig*	*dest* ← *orig*	Move. Operands must be the same size (BYTE, WORD or DWORD). *dest* may be a register or memory location. *orig* may be a register, memory location or immediate value. Both *orig* and *dest* can't be memory locations at the same time.
XCHG *op1*, *op2*	temp ← *op1* *op1* ← *op2* *op2* ← temp	Exchange. Operands must be the same size (BYTE, WORD or DWORD). At least one of the operands must be a register. The other one may be a memory location or another register.

Stack Manipulation

Instruction	*Operation*	*Notes*
PUSH *op*	ESP ← ESP – 4 [ESP] ← *op*	Operand must be a DWORD register, memory location or immediate value.
POP *dest*	*dest* ← [ESP] ESP ← ESP + 4	Operand must be a DWORD register or memory location.
PUSHF	ESP ← ESP – 4 [ESP] ← EFLAGS	Push flags.
POPF	EFLAGS ← [ESP] ESP ← ESP + 4	Pop flags.

Condition Codes

Suffix	*Meaning*	*Flag Interpretation*	*Notes*
O	*Overflow*	`OF==1`	
NO	*No Overflow*	`OF==0`	
S	*Sign*	`SF==1`	
NS	*Not Sign*	`SF==0`	
P PE	*Parity* *Parity Even*	`PF==1`	
NP PO	*Not Parity* *Parity Odd*	`PF==0`	
Z E	*Zero* *Equal*	`ZF==1`	
NZ NE	*Not Zero* *Not Equal*	`ZF==0`	
C B NAE	*Carry* *Below* *Not Above nor Equal*	`CF==1`	Used for UNSIGNED comparisons.
NC NB AE	*No Carry* *Not Below* *Above or Equal*	`CF==0`	
BE NA	*Below or Equal* *Not Above*	`CF==1 || ZF==1`	
A NBE	*Above* *Not Below nor Equal*	`CF==0 && ZF==0`	
L NGE	*Less* *Not Greater nor Equal*	`SF!=OF`	Used for SIGNED comparisons.
GE NL	*Greater or Equal* *Not Less*	`SF==OF`	
LE NG	*Less or Equal* *Not Greater*	`ZF==1 || SF!=OF`	
G NLE	*Greater* *Not Less nor Equal*	`ZF==0 && SF==OF`	

Conditional Data Transfers

Instruction	*Operation*	*Notes*

CMOV*cc dest*, *orig*	if (*cc*) {
	dest ← *orig*
	}
		cc is any of the condition codes. Operands must be the same size (WORD or DWORD). *dest* must be a register. *orig* may be a register or memory location.

SET*cc dest*	if (*cc*) {
	dest ← 0x01
	} else {
	dest ← 0x00
	}
		cc is any of the condition codes. *dest* must be a BYTE register or memory location.

Flow Control

Instruction	*Operation*	*Notes*
JMP *dest*	EIP ← *dest*	Unconditional jump. *dest* may be a DWORD register, memory location or immediate value (typically a label).

J*cc dest*	if (*cc*) {
	EIP ← EIP + *dest*
	}
		Conditional short jump. *cc* is any of the condition codes. *dest* must be an immediate value (typically a label) within a signed 8-bit range (-128 to 127).

J*cc* NEAR *dest*	if (*cc*) {
	EIP ← EIP + *dest*
	}
		Conditional near jump. *cc* is any of the condition codes. *dest* must be an immediate value (typically a label) within a signed 32-bit range.

CALL *dest*	ESP ← ESP – 4 [ESP] ← EIP EIP ← *dest*	Call subroutine. *dest* may be a DWORD register, memory location or immediate value (typically a label).
RET	EIP ← [ESP] ESP ← ESP + 4	Return from subroutine.

Carry Flag

Instruction	*Operation*	*Notes*
CLC	CF ← 0	Clear carry.
STC	CF ← 1	Set carry.
CMC	CF ← ~CF	Complement carry.

Addition

Instruction	*Operation*	*Notes*
ADD *dest*, *orig*	*dest* ← *dest* + *orig*	Same restrictions as MOV instruction. Modified flags: OF SF ZF AF PF CF
ADC *dest*, *orig*	*dest* ← *dest* + *orig* + CF	Add with carry. Same restrictions as MOV instruction. Modified flags: OF SF ZF AF PF CF
INC *dest*	*dest* ← *dest* + 1	Increment. *dest* may be a BYTE, WORD or DWORD register or memory location. Modified flags: OF SF ZF AF PF

Subtraction

Instruction	*Operation*	*Notes*
SUB *dest*, *orig*	*dest* ← *dest* – *orig*	Subtract. Same restrictions as MOV instruction. Modified flags: OF SF ZF AF PF CF
SBB *dest*, *orig*	*dest* ← *dest* – *orig* – CF	Subtract with borrow. Same restrictions as MOV instruction. Modified flags: OF SF ZF AF PF CF
DEC *dest*	*dest* ← *dest* – 1	Decrement. Same restrictions as INC instruction. Modified flags: OF SF ZF AF PF
NEG *dest*	*dest* ← – *dest*	Two's complement. Same restrictions as INC instruction. Sets CF, unless *dest* is zero, in which case CF is cleared. Modified flags: OF SF ZF AF PF CF
CMP *op1*, *op2*	IGNORE ← *op1* – *op2*	Compare. Same restrictions as MOV instruction. Modified flags: OF SF ZF AF PF CF

Multiplication

Instruction	*Operation*	*Notes*

MUL *orig*	if (size(*orig*)== 8) {
	AX ← AL × *orig*
	} else if (size(*orig*)==16) {
	DX:AX ← AX × *orig*
	} else if (size(*orig*)==32) {
	EDX:EAX ← EAX × *orig*
	}
		Used for UNSIGNED multiplications.
		orig may be a BYTE, WORD or DWORD register or memory location.

IMUL *orig*	if (size(*orig*)== 8) {
	AX ← AL × *orig*
	} else if (size(*orig*)==16) {
	DX:AX ← AX × *orig*
	} else if (size(*orig*)==32) {
	EDX:EAX ← EAX × *orig*
	}
		Used for SIGNED multiplications.
		orig may be a BYTE, WORD or DWORD register or memory location.

IMUL *dest*, *orig*	*dest* ← *dest* × *orig*	Operands must be the same size (WORD or DWORD). *dest* must be a register. *orig* may be a register, memory location or immediate value.
IMUL *dest*, *orig*, *const*	*dest* ← *orig* × *const*	Operands must be the same size (WORD or DWORD). *dest* must be a register. *orig* may be a register or memory location. *const* must be an immediate value.

Division

Instruction	*Operation*	*Notes*

DIV *orig*	if (size(*orig*)== 8) {
	AL ← AX / *orig*
	AH ← AX % *orig*
	} else if (size(*orig*)==16) {
	AX ← DX:AX / *orig*
	DX ← DX:AX % *orig*
	} else if (size(*orig*)==32) {
	EAX ← EDX:EAX / *orig*
	EDX ← EDX:EAX % *orig*
	}
		Used for UNSIGNED divisions.
		Produces an exception (INT 0) if divide by zero or if quotient doesn't fit.

IDIV *orig*	if (size(*orig*)== 8) {
	AL ← AX / *orig*
	AH ← AX % *orig*
	} else if (size(*orig*)==16) {
	AX ← DX:AX / *orig*
	DX ← DX:AX % *orig*
	} else if (size(*orig*)==32) {
	EAX ← EDX:EAX / *orig*
	EDX ← EDX:EAX % *orig*
	}
		Used for SIGNED divisions.
		Produces an exception (INT 0) if divide by zero or if quotient doesn't fit.

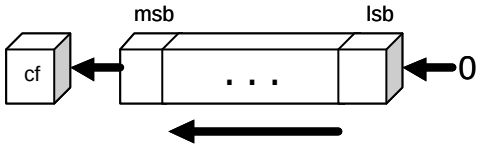
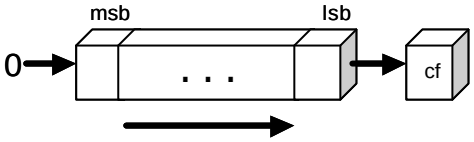
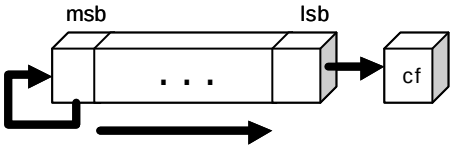
Data Extensions

Instruction	*Operation*	*Notes*
CBW	AX ← SignExtend(AL)	Convert byte to word.
CWD	DX:AX ← SignExtend(AX)	Convert word to dword.
CDQ	EDX:EAX ← SignExtend(EAX)	Convert dword to qword.
MOVSX *dest*, *orig*	*dest* ← SignExtend(*orig*)	Move with sign extend. *dest* must be a WORD or DWORD register. *orig* may be a BYTE or WORD register or memory location. *dest* must be larger than *orig*.
MOVZX *dest*, *orig*	*dest* ← ZeroExtend(*orig*)	Move with zero extend. Same restrictions as MOVSX instruction.

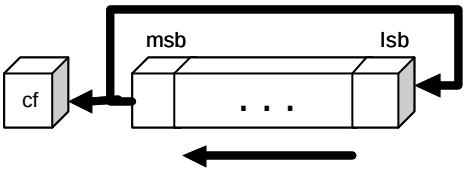
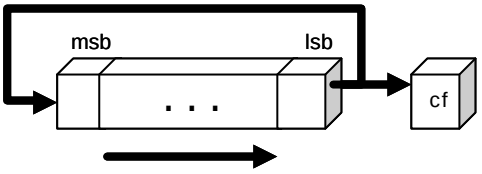
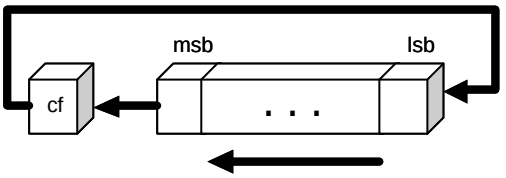
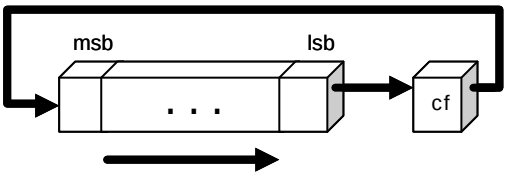
Logical

<table>
<tr><th>Instruction</th><th>Operation</th><th>Notes</th></tr>
<tr><td>AND dest, orig</td><td>dest ← dest & orig
<table>
<tr><th>X</th><th>Y</th><th>X & Y</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table></td><td>Same restrictions as MOV instruction.
Modified flags:
SF ZF PF
CF←0 OF←0</td></tr>
<tr><td>OR dest, orig</td><td>dest ← dest | orig
<table>
<tr><th>X</th><th>Y</th><th>X | Y</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table></td><td>Same restrictions as MOV instruction.
Modified flags:
SF ZF PF
CF←0 OF←0</td></tr>
<tr><td>XOR dest, orig</td><td>dest ← dest ^ orig
<table>
<tr><th>X</th><th>Y</th><th>X ^ Y</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table></td><td>Exclusive OR. Same restrictions as MOV instruction.
Modified flags:
SF ZF PF
CF←0 OF←0</td></tr>
<tr><td>NOT dest</td><td>dest ← ~dest
<table>
<tr><th>X</th><th>~X</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
</table></td><td>One’s complement.
Same restrictions as INC instruction.
Modified flags:
SF ZF PF
CF←0 OF←0</td></tr>
<tr><td>TEST dest, orig</td><td>IGNORE ← dest & orig</td><td>Same restrictions as MOV instruction.
Modified flags:
SF ZF PF
CF←0 OF←0</td></tr>
</table>

Shift

Instruction	*Operation*	*Notes*
SHL *dest*, *count*		Shift left. *dest* may be a BYTE, WORD or DWORD register or memory location. *count* may be CL or an immediate value. Modified flags: SF ZF PF CF
SHR *dest*, *count*		Shift right. Same restrictions as SHL instruction. Modified flags: SF ZF PF CF
SAR *dest*, *count*		Shift arithmetic right. Same restrictions as SHL instruction. Modified flags: SF ZF PF CF

Rotate

Instruction	*Operation*	*Notes*
ROL *dest*, *count*		Rotate left. Same restrictions as SHL instruction. Modified flags: SF ZF PF CF
ROR *dest*, *count*		Rotate right. Same restrictions as SHL instruction. Modified flags: SF ZF PF CF
RCL *dest*, *count*		Rotate through carry left. Same restrictions as SHL instruction. Modified flags: SF ZF PF CF
RCR *dest*, *count*		Rotate through carry right. Same restrictions as SHL instruction. Modified flags: SF ZF PF CF

Floating Point Operations

Real Transfers

Instruction	*Operation*	*Notes*
FLD *mem*	push(*mem*)	*mem* must be a DWORD, QWORD or TWORD memory location.
FLD ST*n*	push(ST*n*)	
FST *mem*	mem ← ST0	*mem* must be a DWORD or QWORD memory location.
FST ST*n*	ST*n* ← ST0	
FSTP *mem*	*mem* ← pop()	*mem* must be a DWORD, QWORD or TWORD memory location.
FSTP ST*n*	ST*n* ← pop()	
FXCH	temp ← ST0 ST0 ← ST1 ST1 ← temp	
FXCH ST*n*	temp ← ST0 ST0 ← ST*n* ST*n* ← temp	

Integer Transfers

Instruction	*Operation*	*Notes*
FILD *mem*	push(*mem*)	*mem* must be a WORD, DWORD or QWORD memory location.
FIST *mem*	*mem* ← ST0	*mem* must be a WORD or DWORD memory location.
FISTP *mem*	*mem* ← pop()	*mem* must be a WORD, DWORD or QWORD memory location.

Packed BCD Transfers

Instruction	*Operation*	*Notes*
FBLD mem	push(mem)	*mem* must be a TWORD memory location.
FBSTP mem	mem ← pop()	*mem* must be a TWORD memory location.

Loading Constants

Instruction	*Operation*
FLDZ	push(+0.0)
FLD1	push(1.0)
FLDPI	push(π)
FLDL2E	push($\log_2 e$)
FLDL2T	push($\log_2 10$)
FLDLG2	push($\log_{10} 2$)
FLDLN2	push($\log_e 2$)

Addition

Instruction	*Operation*	*Notes*
FADD ST*n*, ST0	ST*n* ← ST*n* + ST0	
FADD ST0, ST*n*	ST0 ← ST0 + ST*n*	
FADD *mem*	ST0 ← ST0 + *mem*	*mem* must be a real DWORD or QWORD memory location.
FADDP ST*n*, ST0	ST*n* ← ST*n* + ST0 pop()	
FIADD *mem*	ST0 ← ST0 + *mem*	*mem* must be an integer WORD or DWORD memory location.

Normal Subtraction

Instruction	*Operation*	*Notes*
FSUB ST*n*, ST0	ST*n* ← ST*n* – ST0	
FSUB ST0, ST*n*	ST0 ← ST0 – ST*n*	
FSUB *mem*	ST0 ← ST0 – *mem*	*mem* must be a real DWORD or QWORD memory location.
FSUBP ST*n*, ST0	ST*n* ← ST*n* – ST0 pop()	
FISUB *mem*	ST0 ← ST0 – *mem*	*mem* must be an integer WORD o DWORD memory location.

Reversed Subtraction

Instruction	*Operation*	*Notes*
FSUBR ST*n*, ST0	ST*n* ← ST0 – ST*n*	
FSUBR ST0, ST*n*	ST0 ← ST*n* – ST0	
FSUBR *mem*	ST0 ← *mem* – ST0	*mem* must be a real DWORD or QWORD memory location.
FSUBRP ST*n*, ST0	ST*n* ← ST0 – ST*n* pop()	
FISUBR *mem*	ST0 ← *mem* – ST0	*mem* must be an integer WORD or DWORD memory location.

Multiplication

Instruction	*Operation*	*Notes*
FMUL ST*n*, ST0	ST*n* ← ST*n* × ST0	
FMUL ST0, ST*n*	ST0 ← ST0 × ST*n*	
FMUL *mem*	ST0 ← ST0 × *mem*	*mem* must be a real DWORD or QWORD memory location.
FMULP ST*n*, ST0	ST*n* ← ST*n* × ST0 pop()	
FIMUL *mem*	ST0 ← ST0 × *mem*	*mem* must be an integer WORD or DWORD memory location.

Normal Division

Instruction	*Operation*	*Notes*
FDIV ST*n*, ST0	ST*n* ← ST*n* ÷ ST0	
FDIV ST0, ST*n*	ST0 ← ST0 ÷ ST*n*	
FDIV *mem*	ST0 ← ST0 ÷ *mem*	*mem* must be a real DWORD or QWORD memory location.
FDIVP ST*n*, ST0	ST*n* ← ST*n* ÷ ST0 pop()	
FIDIV *mem*	ST0 ← ST0 ÷ *mem*	*mem* must be an integer WORD or DWORD memory location.

Reversed Division

Instruction	*Operation*	*Notes*
FDIVR ST*n*, ST0	ST*n* ← ST0 ÷ ST*n*	
FDIVR ST0, ST*n*	ST0 ← ST*n* ÷ ST0	
FDIVR *mem*	ST0 ← *mem* ÷ ST0	*mem* must be a real DWORD or QWORD memory location.
FDIVRP ST*n*, ST0	ST*n* ← ST0 ÷ ST*n* pop()	
FIDIVR *mem*	ST0 ← *mem* ÷ ST0	*mem* must be an integer WORD or DWORD memory location.

Transcendental

All trigonometric operations work with radians.

Instruction	*Operation*	*Notes*
F2XM1 FYL2X	x ← pop() push(2^x - 1) x ← pop() y ← pop() push(y × $\log_2$(x))	It must be true that: -0.5 ≤ x ≤ +0.5
FYL2XP1	x ← pop() y ← pop() push(y × $\log_2$(x + 1))	It must be true that: $-1+\frac{\sqrt{2}}{2} \le x \le 1-\frac{\sqrt{2}}{2}$
FPTAN	x ← pop() push(tan(x)) push(1.0)	Computes partial tangent. It must be true that: $0 \le x < \pi \times 2^{62}$
FPATAN	x ← pop() y ← pop() push(arctan(y / x))	Computes the partial arctangent.
FSIN	ST0 ← sin(ST0)	Computes sine.
FCOS	ST0 ← cos(ST0)	Computes cosine.
FSINCOS	x ← pop() push(sin(x)) push(cos(x))	Computes sine and cosine.

Comparisons

Instruction	*Operation*	*Notes*
FCOMI ST0, ST*n*	compare(ST0, ST*n*)	Modifies ZF, PF and CF (see table).
FCOMIP ST0, ST*n*	compare(ST0, ST*n*) pop()	Modifies ZF, PF and CF (see table).

FCMOVB ST0, ST*n*	if(below)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVBE ST0, ST*n*	if(below or equal)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVE ST0, ST*n*	if(equal)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVNB ST0, ST*n*	if(not below)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVNBE ST0, ST*n*	if(not below nor equal)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVNE ST0, ST*n*	if(not equal)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVNU ST0, ST*n*	if(not unordered)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

FCMOVU ST0, ST*n*	if(unordered)
	ST0 ← ST*n*
	endif
		Must be executed after FCOMI or FCOMIP.

Compare Table

compare(x, y)	ZF	PF	CF
x > y	0	0	0
x < y	0	0	1
x = y	1	0	0
Not Comparable	1	1	1

Miscellaneous

Instruction	*Operation*	*Notes*
FINIT	Resets the FPU to its default state.	Empties the FPU register stack.

FABS	if(ST0 < 0)
	ST0 ← –ST0
	endif
		Computes the absolute value.

FCHS	ST0 ← –ST0	Change sign.
FRNDINT	ST0 ← round(ST0)	Rounds ST0 to an integer.
FSQRT	ST0 ← $\sqrt{\text{ST0}}$	Computes square root.
FPREM	ST0 ← remainder(ST0 ÷ ST1)	Computes partial remainder of ST0 divided by ST1 using repeated subtractions.
FSCALE	ST0 ← ST0 × $2^{\text{int(ST1)}}$	Scales by powers of two.
FXTRACT	temp ← pop() push(exponent(temp)) push(mantisa(temp))	Breaks a number down into exponent and mantissa.

MMX Operations

Empty MMX State

Instruction	*Operation*	*Notes*
EMMS	Empty MMX state.	Should be used at the end of a sequence of MMX instructions in order to allow subsequent FPU instructions.

Data Transfers

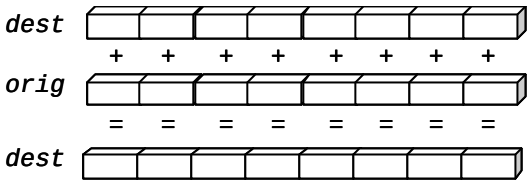
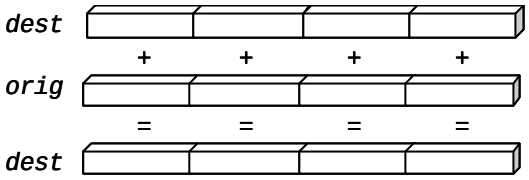
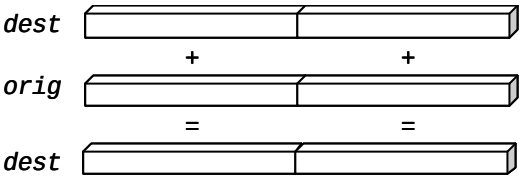
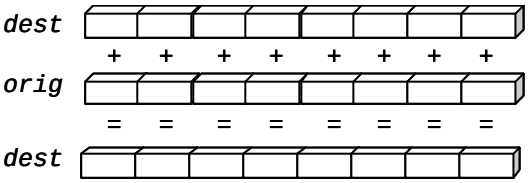
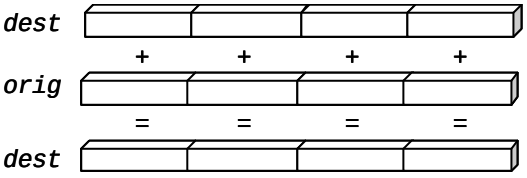
Instruction	*Operation*	*Notes*
MOVD *dest*, *orig*	*dest* ← *orig*	Copies the low 32 bits of *orig* into *dest*. One of *dest* or *orig* must be a QWORD register. The other one may be a DWORD register or memory location. If *dest* is a QWORD register, its top 32 bits are set to zero.
MOVQ *dest*, *orig*	*dest* ← *orig*	At least one of *dest* or *orig* must be a QWORD register. The other one may be another QWORD register or memory location.

Data Range Limits for Saturation

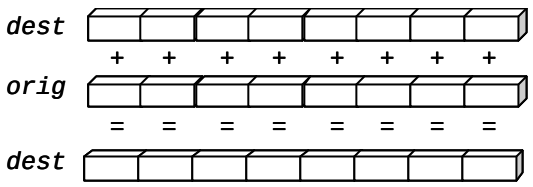
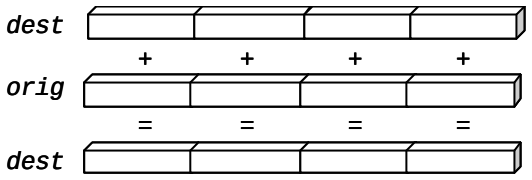
Data Type	*Lower Limit*		*Upper Limit*	
	decimal	*hexadecimal*	*decimal*	*hexadecimal*
SIGNED BYTE	–128	0x80	127	0x7F
SIGNED WORD	–32,768	0x8000	32,767	0x7FFF
UNSIGNED BYTE	0	0x00	255	0xFF
UNSIGNED WORD	0	0x0000	65,535	0xFFFF

General MMX instruction restrictions: *dest* must be a QWORD register. *orig* may be a QWORD register or memory location.

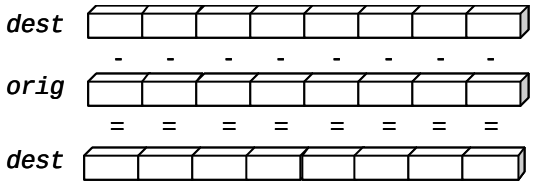
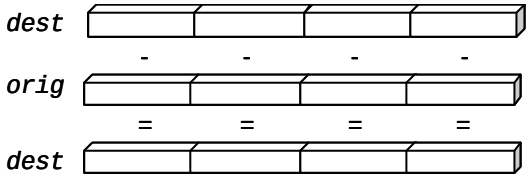
Addition

Instruction	*Operation*	*Notes*
PADDB *dest*, *orig*		Packed truncated byte addition.
PADDW *dest*, *orig*		Packed truncated word addition.
PADDD *dest*, *orig*		Packed truncated dword addition.
PADDSB *dest*, *orig*		Packed signed saturated byte addition.
PADDSW *dest*, *orig*		Packed signed saturated word addition.

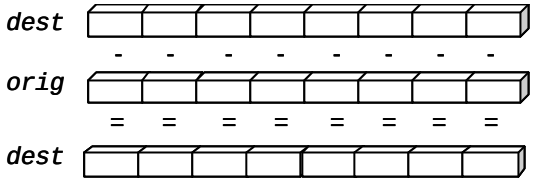
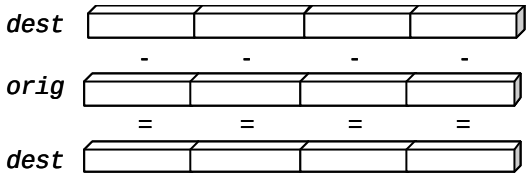
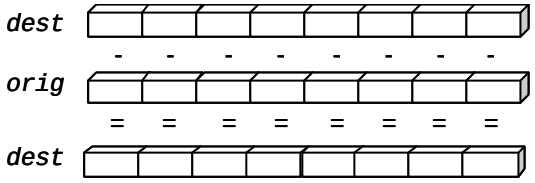
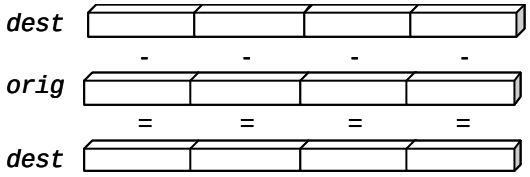
Addition (continued)

Instruction	*Operation*	*Notes*
PADDUSB *dest*, *orig*		Packed unsigned saturated byte addition.
PADDUSW *dest*, *orig*		Packed unsigned saturated word addition.

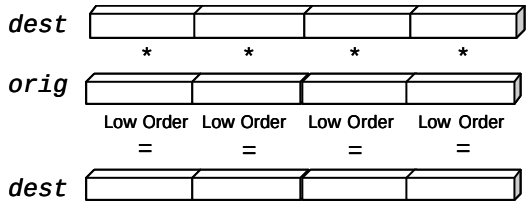
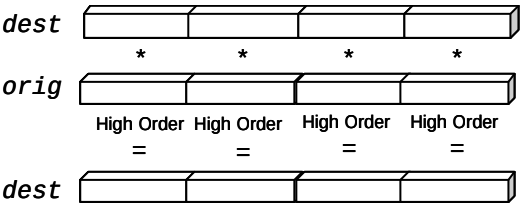
Subtraction

Instruction	*Operation*	*Notes*
PSUBB *dest*, *orig*		Packed truncated byte subtraction.
PSUBW *dest*, *orig*		Packed truncated word subtraction.
PSUBD *dest*, *orig*		Packed truncated dword subtraction.

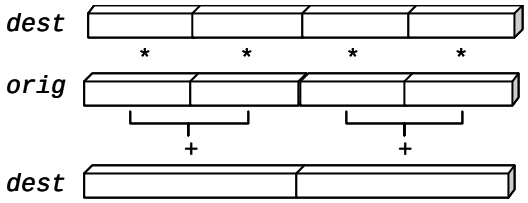
Subtraction (continued)

Instruction	*Operation*	*Notes*
PSUBSB *dest*, *orig*		Packed signed saturated byte subtraction.
PSUBSW *dest*, *orig*		Packed signed saturated word subtraction.
PSUBUSB *dest*, *orig*		Packed unsigned saturated byte subtraction.
PSUBUSW *dest*, *orig*		Packed unsigned saturated word subtraction.

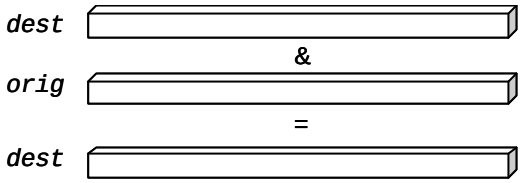
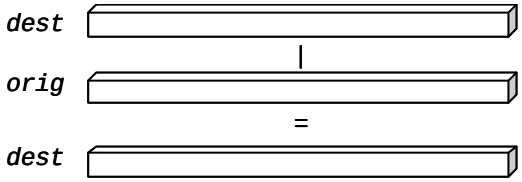
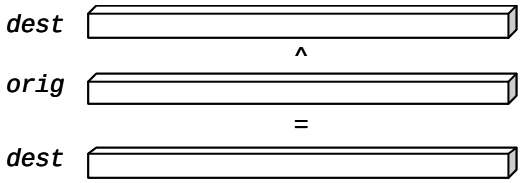
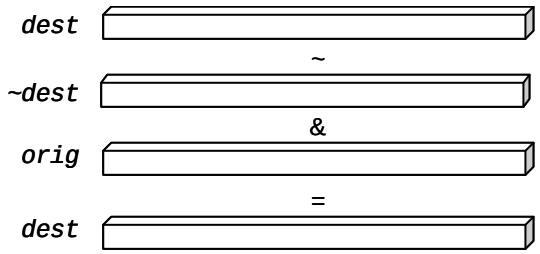
Multiplication

Instruction	*Operation*	*Notes*
PMULLW *dest*, *orig*		Packed signed multiply low word.
PMULHW *dest*, *orig*		Packed signed multiply high word.

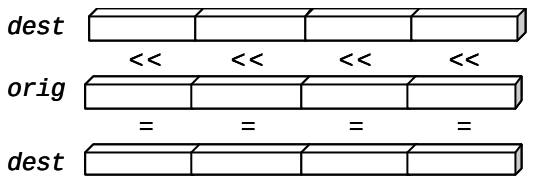
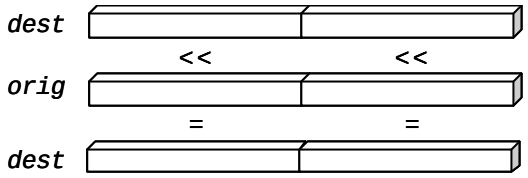
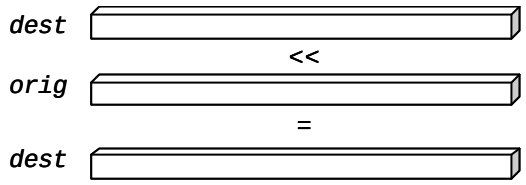
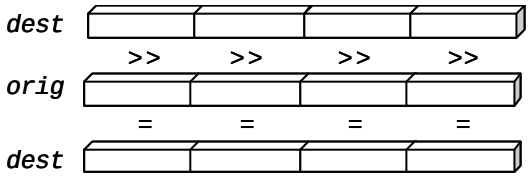
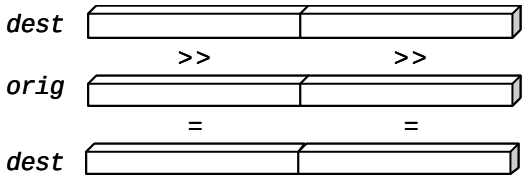
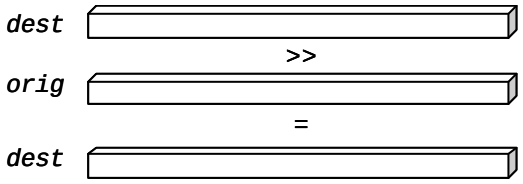
Multiplication and Addition

Instruction	*Operation*	*Notes*
PMADDWD *dest*, *orig*		Packed signed multiply and add.

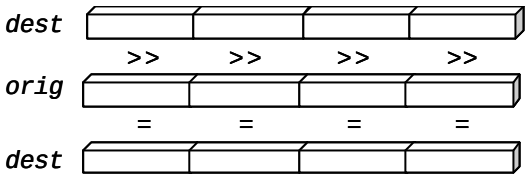
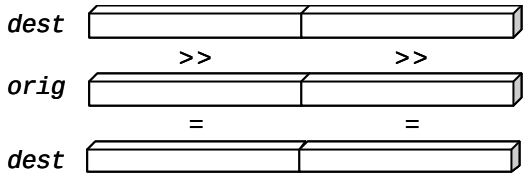
Logical

Instruction	*Operation*	*Notes*
PAND *dest*, *orig*		Bitwise qword AND.
POR *dest*, *orig*		Bitwise qword OR.
PXOR *dest*, *orig*		Bitwise qword XOR.
PANDN *dest*, *orig*		Bitwise qword AND/NOT.

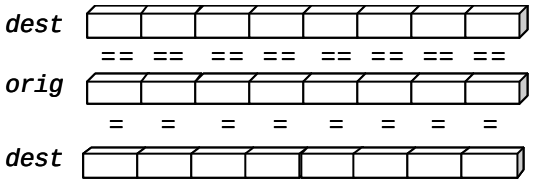
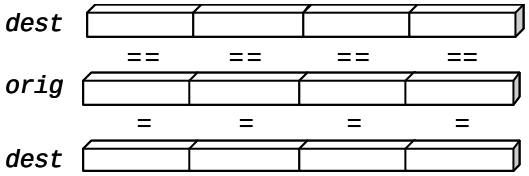
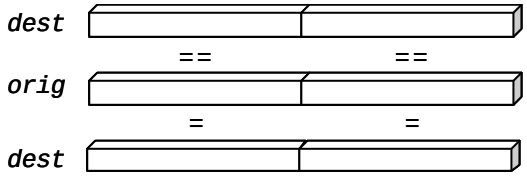
Shift Logical

Instruction	*Operation*	*Notes*
PSLLW *dest*, *orig*		Packed word logical shift left. *orig* may also be an immediate value.
PSLLD *dest*, *orig*		Packed dword logical shift left. *orig* may also be an immediate value.
PSLLQ *dest*, *orig*		Packed qword logical shift left. *orig* may also be an immediate value.
PSRLW *dest*, *orig*		Packed word logical (unsigned) shift right. *orig* may also be an immediate value.
PSRLD *dest*, *orig*		Packed dword logical (unsigned) shift right. *orig* may also be an immediate value.
PSRLQ *dest*, *orig*		Packed qword logical (unsigned) shift right. *orig* may also be an immediate value.

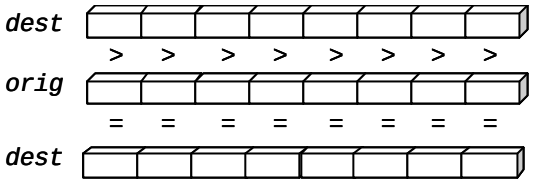
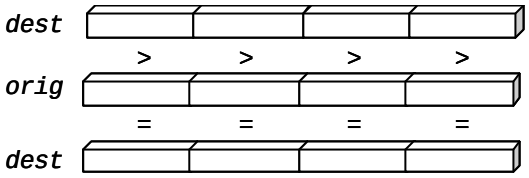
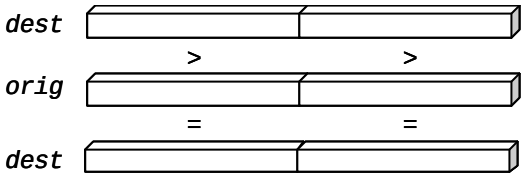
Shift Arithmetic

Instruction	*Operation*	*Notes*
PSRAW *dest*, *orig*		Packed word arithmetic (signed) shift right. *orig* may also be an immediate value.
PSRAD *dest*, *orig*		Packed dword arithmetic (signed) shift right. *orig* may also be an immediate value.

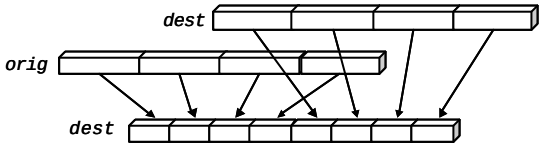
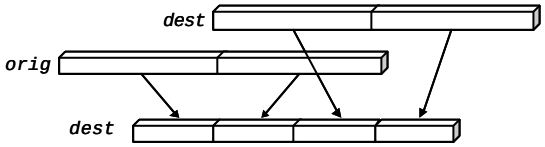
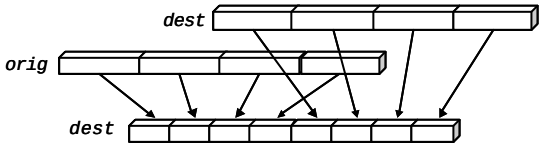
Compare Equal

Instruction	*Operation*	*Notes*
PCMPEQB *dest*, *orig*		Packed compare for equal bytes. For each resulting byte: All ones if true, all zeros if false.
PCMPEQW *dest*, *orig*		Packed compare for equal words. For each resulting word: All ones if true, all zeros if false.
PCMPEQD *dest*, *orig*		Packed compare for equal dwords. For each resulting dword: All ones if true, all zeros if false.

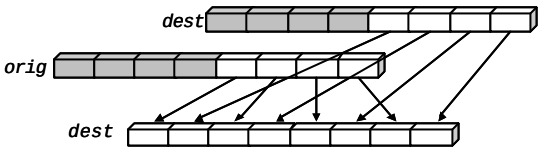
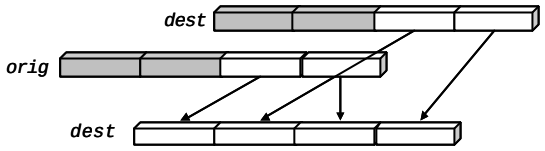
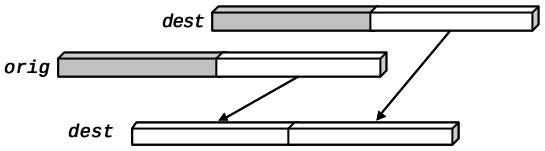
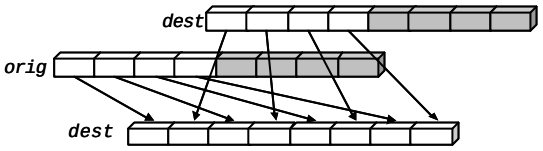
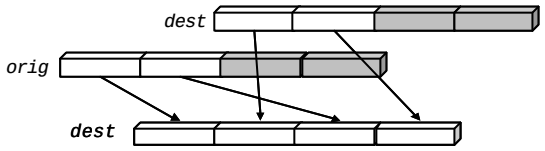
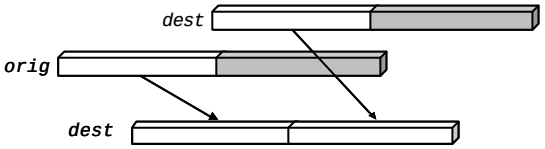
Compare Greater Than

Instruction	*Operation*	*Notes*
PCMPGTB *dest*, *orig*		Packed compare for greater than bytes. For each resulting byte: All ones if true, all zeros if false.
PCMPGTW *dest*, *orig*		Packed compare for greater than words. For each resulting word: All ones if true, all zeros if false.
PCMPGTD *dest*, *orig*		Packed compare for greater than dwords. For each resulting dword: All ones if true, all zeros if false.

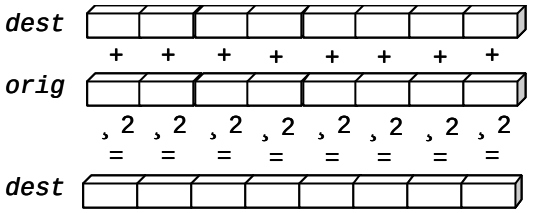
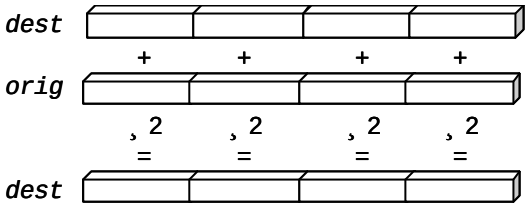
Pack

Instruction	*Operation*	*Notes*
PACKSSWB *dest*, *orig*		Pack words into bytes with signed saturation.
PACKSSDW *dest*, *orig*		Pack dwords into words with signed saturation.
PACKUSWB *dest*, *orig*		Pack words into bytes with unsigned saturation.

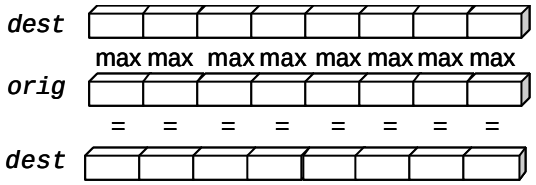
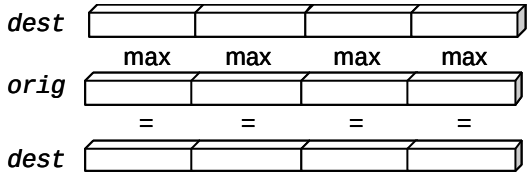
Unpack

Instruction	*Operation*	*Notes*
PUNPCKLBW *dest*, *orig*		Unpack low packed bytes.
PUNPCKLWD *dest*, *orig*		Unpack low packed words.
PUNPCKLDQ *dest*, *orig*		Unpack low packed dwords.
PUNPCKHBW *dest*, *orig*		Unpack high packed bytes.
PUNPCKHWD *dest*, *orig*		Unpack high packed words.
PUNPCKHDQ *dest*, *orig*		Unpack high packed dwords.

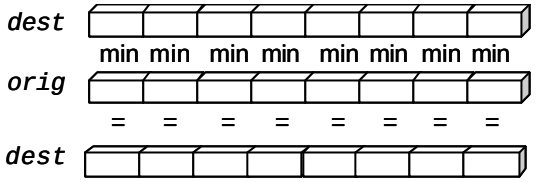
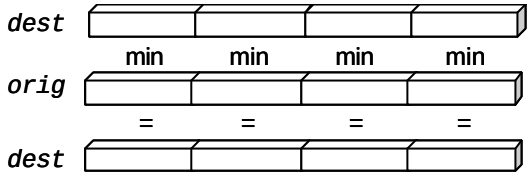
Average

Instruction	*Operation*	*Notes*
PAVGB *dest*, *orig*		Packed unsigned byte average with rounding (0.5 ↑).
PAVGW *dest*, *orig*		Packed unsigned word average with rounding (0.5 ↑).

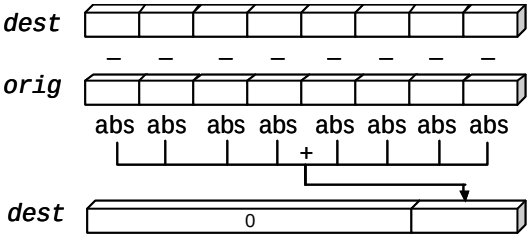
Maximum

Instruction	*Operation*	*Notes*
PMAXUB *dest*, *orig*		Maximum of packed unsigned bytes.
PMAXSW *dest*, *orig*		Maximum of packed signed words.

Minimum

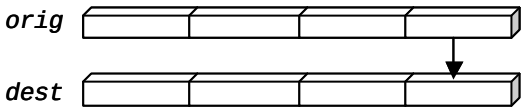
Instruction	*Operation*	*Notes*
PMINUB *dest*, *orig*		Minimum of packed unsigned bytes.
PMINSW *dest*, *orig*		Minimum of packed signed words.

Absolute Difference Addition

Instruction	*Operation*	*Notes*
PSADBW *dest*, *orig*		Computes the absolute differences of the packed unsigned bytes. Differences are then summed to produce an unsigned word result.

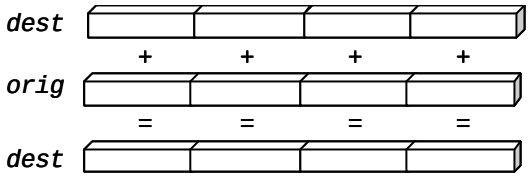
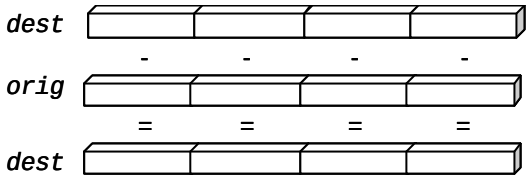
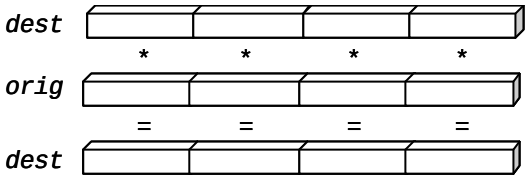
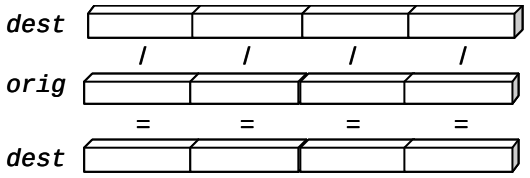
SSE Operations

Data Transfers

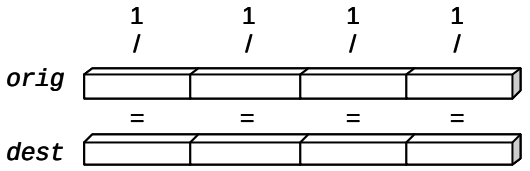
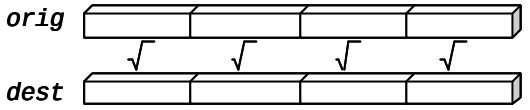
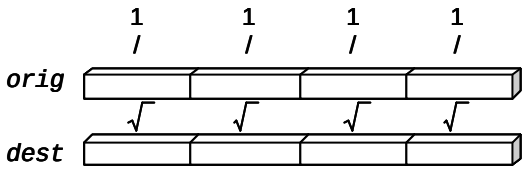
Instruction	*Operation*	*Notes*
MOVUPS *dest*, *orig*	*dest* ← *orig*	Copies the packed single precision floating-point values of *orig* into *dest*. One of *dest* or *orig* must be a XMM register. The other one may be a XMM register or a 128-bit memory location.
MOVSS *dest*, *orig*		Copies the least significant single precision floating-point value of *orig* into *dest*. One of *dest* or *orig* must be a XMM register. The other one may be a XMM register or a 32-bit memory location.

General SSE instruction restrictions: *dest* must be a XMM register. *orig* may be a XMM register or a 128-bit memory location.

Basic Arithmetic Instructions

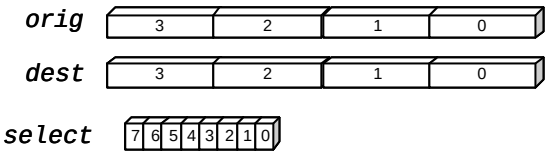
Instruction	*Operation*	*Notes*
ADDPS *dest*, *orig*		Add packed single precision floating-point values.
SUBPS *dest*, *orig*		Subtract packed single precision floating-point values.
MULPS *dest*, *orig*		Multiply packed single precision floating-point values.
DIVPS *dest*, *orig*		Divide packed single precision floating-point values.

Reciprocal and Square Root Instructions

RCPPS *dest*, *orig*		Compute the approximate reciprocals of packed single precision floating-point values.
SQRTPS *dest*, *orig*		Compute the square root of packed single precision floating-point values.
RSQRTPS *dest*, *orig*		Compute the approximate reciprocals of the square root of packed single precision floating-point values.

Shuffle Instruction

SHUFPS *dest*, *orig*, *select*



select bit 1	*select* bit 0	action
0	0	`dest[0] = dest[0]`
0	1	`dest[0] = dest[1]`
1	0	`dest[0] = dest[2]`
1	1	`dest[0] = dest[3]`

select bit 3	*select* bit 2	action
0	0	`dest[1] = dest[0]`
0	1	`dest[1] = dest[1]`
1	0	`dest[1] = dest[2]`
1	1	`dest[1] = dest[3]`

select bit 5	*select* bit 4	action
0	0	`dest[2] = orig[0]`
0	1	`dest[2] = orig[1]`
1	0	`dest[2] = orig[2]`
1	1	`dest[2] = orig[3]`

select bit 7	*select* bit 6	action
0	0	`dest[3] = orig[0]`
0	1	`dest[3] = orig[1]`
1	0	`dest[3] = orig[2]`
1	1	`dest[3] = orig[3]`

Shuffle packed single precision floating-point values. *select* must be an 8-bit immediate value.

SSE2 Operations

All 64-bit MMX instructions have a counterpart with exactly the same name in the 128-bit SSE2 instruction set. The only difference at the assembly language level is that the MMX instructions use 64-bit registers (MM0 ... MM7) while SSE2 instructions use 128-bit registers (XMM0 ... XMM7). For example, the following MMX code:

```
paddusb mm0, mm1
```

adds 8 unsigned packed bytes using saturated arithmetic. Its SSE2 counterpart would be:

```
paddusb xmm0, xmm1
```

which adds 16 unsigned packed bytes, also using saturated arithmetic.

The following tables describe some integer instructions that are part of the SSE2 instruction but do not exist in the MMX instruction set or that have an extended functionality.

Data Transfers

Instruction	*Notes*
MOVD *dest*, *orig*	**Move Doubleword.** Copies a doubleword from the *orig* operand to the *dest* operand. The *orig* and *dest* operands can be general-purpose registers (EAX, EBX, etc.), MMX registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword an MMX register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX registers, between XMM registers, between general-purpose registers, or between memory locations. When the *dest* operand is an MMX register, the *orig* operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the *dest* operand is an XMM register, the *orig* operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

MOVQ *dest*, *orig*	**Move Quadword.** Copies a quadword from the *orig* operand (second operand) to the *dest* operand. The source and destination operands can be MMX registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX registers or between an MMX register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations. When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.
MOVDQU *dest*, *orig*	**Move Unaligned Double Quadword.** Moves a double quadword from the *orig* operand to the *dest* operand. This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.
MOVDQ2Q *dest*, *orig*	**Move Quadword from XMM to MMX Register.** Moves the low quadword from the *orig* operand to the *dest* operand. The *orig* operand is an XMM register and the *dest* operand is an MMX register.
MOVQ2DQ *dest*, *orig*	**Move Quadword from MMX to XMM Register.** Moves the quadword from the *orig* operand to the low quadword of the *dest* operand. The *orig* operand is an MMX register and the *dest* operand is an XMM register. The high quadword of *dest* is cleared to all 0s.

Shift Logical

Instruction	*Notes*
PSLLDQ *dest*, *count*	**Shift Double Quadword Left Logical.** Shifts the *dest* operand to the left by the number of bytes (not bits) specified in the *count* operand. The empty low-order bytes are cleared (set to all 0s). If the value specified by the *count* operand is greater than 15, the *dest* operand is set to all 0s. The *dest* operand is an XMM register. The *count* operand is an 8-bit immediate.
PSRLDQ *dest*, *count*	**Shift Double Quadword Right Logical.** Shifts the *dest* operand to the right by the number of bytes (not bits) specified in the *count* operand. The empty high-order bytes are cleared (set to all 0s). If the value specified by the *count* operand is greater than 15, the *dest* operand is set to all 0s. The *dest* operand is an XMM register. The *count* operand is an 8-bit immediate.

NASM Specifics

Segments

Directive	*Notes*
SECTION .data	States the beginning of the initialized data segment.
SECTION .bss	States the beginning of the uninitialized data segment.
SECTION .text	States the beginning of the segment that contains the program's executable instructions.

Symbol Exporting and Importing

Directive	*Notes*
GLOBAL *symbol*	Export *symbol* to linker and external modules.
EXTERN *symbol*	Import *symbol* defined in an external module.

Declaring Initialized Data

Pseudo-Instruction	*Notes*	*Size (bits)*
symbol DB *value*	Define byte	8
symbol DW *value*	Define word	16
symbol DD *value*	Define dword	32
symbol DQ *value*	Define qword	64
symbol DT *value*	Define tword	80

Declaring Uninitialized Data

Pseudo-Instruction	*Notes*	*Size (bits)*
symbol RESB *num*	Reserve *num* bytes	8
symbol RESW *num*	Reserve *num* words	16
symbol RESD *num*	Reserve *num* dwords	32
symbol RESQ *num*	Reserve *num* qwords	64
symbol REST *num*	Reserve *num* twords	80

Defining Constants

Pseudo-Instruction	*Notes*
symbol EQU *value*	Defines symbol to a given constant value.

Expressions

Listed in increasing order of precedence.

Operator	*Notes*
|	OR
^	XOR
&	AND
<< >>	Shift left and shift right
+ –	Binary addition and subtraction
* / // % %%	Multiplication, unsigned division, signed division, unsigned modulo, signed modulo
+ – ~	Unary plus, minus and negate
$	Evaluates to the assembly position at the beginning of the line containing the expression

ASCII Codes

Character	ASCII	
	Dec	Hex
NUL	0	0x00
SOH	1	0x01
STX	2	0x02
ETX	3	0x03
EOT	4	0x04
ENQ	5	0x05
ACK	6	0x06
BEL	7	0x07
BS	8	0x08
HT	9	0x09
LF	10	0x0A
VT	11	0x0B
FF	12	0x0C
CR	13	0x0D
SO	14	0x0E
SI	15	0x0F
DLE	16	0x10
DC1	17	0x11
DC2	18	0x12
DC3	19	0x13
DC4	20	0x14
NAK	21	0x15
SYN	22	0x16
ETB	23	0x17
CAN	24	0x18
EM	25	0x19
SUB	26	0x1A
ESC	27	0x1B
FS	28	0x1C
GS	29	0x1D
RS	30	0x1E
US	31	0x1F
SP	32	0x20
!	33	0x21
"	34	0x22
#	35	0x23
$	36	0x24
%	37	0x25
&	38	0x26
'	39	0x27

Character	ASCII	
	Dec	Hex
(	40	0x28
)	41	0x29
*	42	0x2A
+	43	0x2B
,	44	0x2C
-	45	0x2D
.	46	0x2E
/	47	0x2F
0	48	0x30
1	49	0x31
2	50	0x32
3	51	0x33
4	52	0x34
5	53	0x35
6	54	0x36
7	55	0x37
8	56	0x38
9	57	0x39
:	58	0x3A
;	59	0x3B
<	60	0x3C
=	61	0x3D
>	62	0x3E
?	63	0x3F
@	64	0x40
A	65	0x41
B	66	0x42
C	67	0x43
D	68	0x44
E	69	0x45
F	70	0x46
G	71	0x47
H	72	0x48
I	73	0x49
J	74	0x4A
K	75	0x4B
L	76	0x4C
M	77	0x4D
N	78	0x4E
O	79	0x4F

Character	ASCII	
	Dec	Hex
P	80	0x50
Q	81	0x51
R	82	0x52
S	83	0x53
T	84	0x54
U	85	0x55
V	86	0x56
W	87	0x57
X	88	0x58
Y	89	0x59
Z	90	0x5A
[	91	0x5B
\	92	0x5C
]	93	0x5D
^	94	0x5E
_	95	0x5F
`	96	0x60
a	97	0x61
b	98	0x62
c	99	0x63
d	100	0x64
e	101	0x65
f	102	0x66
g	103	0x67
h	104	0x68

Character	ASCII	
	Dec	Hex
i	105	0x69
j	106	0x6A
k	107	0x6B
l	108	0x6C
m	109	0x6D
n	110	0x6E
o	111	0x6F
p	112	0x70
q	113	0x71
r	114	0x72
s	115	0x73
t	116	0x74
u	117	0x75
v	118	0x76
w	119	0x77
x	120	0x78
y	121	0x79
z	122	0x7A
{	123	0x7B
|	124	0x7C
}	125	0x7D
~	126	0x7E
DEL	127	0x7F

WWW Resources

Course Page	http://aortiz.cem.itesm.mx/cb00852.html
Linux Assembly	http://linuxassembly.org/
Webopedia	http://webopedia.internet.com/
Dr. Dobb's	http://www.x86.org/
Linux Online	http://www.linux.org/
Linux Journal	http://www.linuxjournal.com/
Linux Gazette	http://www.linuxgazette.com/
Linux en México	http://www.linux.org.mx/
Intel	http://www.intel.com/
AMD	http://www.amd.com/

References

[ANTONAKOS] James Antonakos. *The Pentium Microprocesor.* Prentice Hall. 1997.

[DUNTEMANN] Jeff Duntemann. *Assembly Language Step-by-Step: Programming with DOS and Linux.* 2nd Edition. John Wiley & Sons, 2000.

[INTEL] *Pentium 4: Intel Architecture Software Developer's Manual.* Volumes 1 and 2. Intel Corporation, 2000.

[NEVELN] Bob Neveln. *Linux Assembly Language Programming.* Prentice Hall, 2000.

[RAYMOND96] Eric Raymond. *The New Hacker's Dictionary.* Third Edition. MIT Press, 1999.

[RAYMOND99] Eric Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly, 1999.

[TATHAM] Simon Tatham. *NASM: The Netwide Assembler Documentation.* 1997.