The 80x86 Instruction Set Appendix D

The following three tables discuss the integer/control, floating point, and MMX instruction sets. This document uses the following abbreviations:

imm- A constant value, must be appropriate for the operand size.

imm8- An eight-bit immediate constant. Some instructions limit the range of this value to less than 0..255.

immL- A 16- or 32-bit immediate constant.

immH- A 16- or 32-bit immediate constant.

reg- A general purpose integer register.

reg8- A general purpose eight-bit register

reg16- A general purpose 16-bit register.

reg32- A general purpose 32-bit register.

mem- An arbitrary memory location using any of the available addressing modes.

mem16- A word variable using any legal addressing mode.

mem32- A dword variable using any legal addressing mode.

mem64- A qword variable using any legal addressing mode.

label- A statement label in the program.

ProcedureName-The name of a procedure in the program.

Instructions that have two source operands typically use the first operand as a source operand and the second operand as a destination operand. For exceptions and other formats, please see the description for the individual instruction.

Note that this appendix only lists those instructions that are generally useful for application programming. HLA actually supports some additional instructions that are useful for OS kernel developers; please see the HLA documentation for more details on those instructions.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
aaa()	ASCII Adjust after Addition. Adjusts value in AL after a decimal addition operation.
aad()	ASCII Adjust before Division. Adjusts two unpacked values in AX prior to a decimal division.
aam()	ASCII Adjust AX after Multiplication. Adjusts the result in AX for a decimal mulitply.
aas()	ASCII Adjust AL after Subtraction. Adjusts the result in AL for a decimal subtraction.

adc(imm, reg);
adc(imm, mem);
adc(reg, reg);
adc(reg, mem);
adc(mem, reg);
	Add with carry. Adds the source operand plus the carry flag to the destination operand.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

add(imm, reg);
add(imm, mem);
add(reg, reg);
add(reg, mem);
add(mem, reg);
	Add. Adds the source operand to the destination operand.

and(imm, reg);
and(imm, mem);
and(reg, reg);
and(reg, mem);
and(mem, reg);
	Bitwise AND. Logically ANDs the source operand into the destination operand. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

bound(reg, mem); bound(reg, immL, immH);	Bounds check. Reg and memory operands must be the same size and they must be 16 or 32-bit values. This instruction compares the register operand against the value at the specified memory location and raises an exception if the register's value is less than the value in the memory location. If greater or equal, then this instruction compares the register to the next word or dword in memory and raises an exception if the register's value is greater. The second form of this instruction is an HLA extended syntax instruction. HLA encodes the constants as two memory locations and then emits the first form of this instruction using these newly created memory locations. For the second form, the constant values must not exceed the 16-bit or 32-bit register size.
bsf(reg, reg); bsr(mem, reg);	Bit Scan Forward. The two operands must be the same size and they must be 16-bit or 32-bit operands. This instruction locates the first set bit in the source operand and stores the bit number into the destination operand and clears the zero flag. If the source operand does not have any set bits, then this instruction sets the zero flag and the dest register value is undefined.
bsr(reg, reg); bsr(mem, reg);	Bit Scan Reverse. The two operands must be the same size and they must be 16-bit or 32-bit operands. This instruction locates the last set bit in the source operand and stores the bit number into the destination operand and clears the zero flag. If the source operand does not have any set bits, then this instruction sets the zero flag and the dest register value is undefined.
bswap(reg32);	Byte Swap. This instruction reverses the order of the bytes in a 32-bit register. It swaps bytes zero and three and it swaps bytes one and two. This effectively converts data between the little endian (used by Intel) and big endian (used by some other CPUs) formats.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

bt(reg, mem);
bt(reg, reg);
bt(imm8, reg);
bt(imm8, mem);
	Register and memory operands must be 16- or 32-bit values. Eight bit immediate values must be in the range 0..15 for 16-bit registers, 0..31 for 32-bit registers, and 0..255 for memory operands. Source register must be in the range 0..15 or 0..31 for registers. Any value is legal for the source register if the destination operand is a memory location. This instruction copies the bit in the second operand, whose bit position the first operand specifies, into the carry flag.

btc(reg, mem);
btc(reg, reg);
btc(imm8, reg);
btc(imm8, mem);
	Bit test and complement. As above, except this instruction also complements the value of the specified bit in the second operand. Note that this instruction first copies the bit to the carry flag, then complements it. To support atomic operations, the memory-based forms of this instruction are always “memory locked” and they always directly access main memory; the CPU does not use the cache for this result. Hence, this instruction always operates at memory speeds (i.e., slow).

btr(reg, mem);
btr(reg, reg);
btr(imm8, reg);
btr(imm8, mem);
	Bit test and reset. Same as BTC except this instruction tests and resets (clears) the bit.

bts(reg, mem);
bts(reg, reg);
bts(imm8, reg);
bts(imm8, mem);
	Bit test and set. Same as BTC except this instructions tests and sets the bit.

call label;
call(label);
call(reg32);
call(mem32);
	Pushes a return address onto the stack and calls the subroutine at the address specified. Note that the first two forms are the same instruction. The other two forms provide indirect calls via a register or a pointer in memory.

cbw();	Convert Byte to Word. Sign extends AL into AX.
cdq();	Convert double word to quadword. Sign extends EAX into EDX:EAX.
clc();	Clear Carry.
cld();	Clear direction flag. When the direction flag is clear the string instructions increment ESI and/or EDI after each operation.
cli();	Clear the interrupt enable flag.
cmc();	Complement (invert) Carry.

cmova(mem, reg);
cmova(reg, reg);
cmova(reg, mem);
	Conditional Move (if above). Copies the source operand to the destination operand if the previous comparison found the left operand to be greater than (unsigned) the right operand (c=0, z=0). Register and memory operands must be 16-bit or 32-bit values, eight-bit operands are illegal. Does not affect the destination operand if the condition is false.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

cmovae(mem, reg);
cmovae(reg, reg);
cmovae(reg, mem);
	Conditional move if above or equal (see cmova for details).

cmovb(mem, reg);
cmovb(reg, reg);
cmovb(reg, mem);
	Conditional move if below (see cmova for details).

cmovbe(mem, reg);
cmovbe(reg, reg);
cmovbe(reg, mem);
	Conditional move if below or equal (see cmova for details).

cmovc(mem, reg);
cmovc(reg, reg);
cmovc(reg, mem);
	Conditional move if carry set (see cmova for details).

cmove(mem, reg);
cmove(reg, reg);
cmove(reg, mem);
	Conditional move if equal (see cmova for details).

cmovg(mem, reg);
cmovg(reg, reg);
cmovg(reg, mem);
	Conditional move if (signed) greater (see cmova for details).

cmovge(mem, reg);
cmovge(reg, reg);
cmovge(reg, mem);
	Conditional move if (signed) greater or equal (see cmova for details).

cmovl(mem, reg);
cmovl(reg, reg);
cmovl(reg, mem);
	Conditional move if (signed) less than (see cmova for details).

cmovle(mem, reg);
cmovle(reg, reg);
cmovle(reg, mem);
	Conditional move if (signed) less than or equal (see cmova for details).

cmovna(mem, reg);
cmovna(reg, reg);
cmovna(reg, mem);
	Conditional move if (unsigned) not greater (see cmova for details).

cmovnae(mem, reg);
cmovnae(reg, reg);
cmovnae(reg, mem);
	Conditional move if (unsigned) not greater or equal (see cmova for details).

cmovnb(mem, reg);
cmovnb(reg, reg);
cmovnb(reg, mem);
	Conditional move if (unsigned) not less than (see cmova for details).

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

cmovnbe(mem, reg);
cmovnbe(reg, reg);
cmovnbe(reg, mem);
	Conditional move if (unsigned) not less than or equal (see cmova for details).

cmovnc(mem, reg);
cmovnc(reg, reg);
cmovnc(reg, mem);
	Conditional move if no carry/carry clear (see cmova for details).

cmovne(mem, reg);
cmovne(reg, reg);
cmovne(reg, mem);
	Conditional move if not equal (see cmova for details).

cmovng(mem, reg);
cmovng(reg, reg);
cmovng(reg, mem);
	Conditional move if (signed) not greater (see cmova for details).

cmovnge(mem, reg);
cmovnge(reg, reg);
cmovnge(reg, mem);
	Conditional move if (signed) not greater or equal (see cmova for details).

cmovnl(mem, reg);
cmovnl(reg, reg);
cmovnl(reg, mem);
	Conditional move if (signed) not less than (see cmova for details).

cmovnle(mem, reg);
cmovnle(reg, reg);
cmovnle(reg, mem);
	Conditional move if (signed) not less than or equal (see cmova for details).

cmovno(mem, reg);
cmovno(reg, reg);
cmovno(reg, mem);
	Conditional move if no overflow / overflow flag = 0 (see cmova for details).

cmovnp(mem, reg);
cmovnp(reg, reg);
cmovnp(reg, mem);
	Conditional move if no parity / parity flag = 0 / odd parity (see cmova for details).

cmovns(mem, reg);
cmovns(reg, reg);
cmovns(reg, mem);
	Conditional move if no sign / sign flag = 0 (see cmova for details).

cmovnz(mem, reg);
cmovnz(reg, reg);
cmovnz(reg, mem);
	Conditional move if not zero (see cmova for details).

cmovo(mem, reg);
cmovo(reg, reg);
cmovo(reg, mem);
	Conditional move if overflow / overflow flag = 1 (see cmova for details).

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

cmovp(mem, reg);
cmovp(reg, reg);
cmovp(reg, mem);
	Conditional move if parity flag = 1 (see cmova for details).

cmovpe(mem, reg);
cmovpe(reg, reg);
cmovpe(reg, mem);
	Conditional move if even parity / parity flag = 1(see cmova for details).

cmovpo(mem, reg);
cmovpo(reg, reg);
cmovpo(reg, mem);
	Conditional move if odd parity / parity flag = 0 (see cmova for details).

cmovs(mem, reg);
cmovs(reg, reg);
cmovs(reg, mem);
	Conditional move if sign flag = 1 (see cmova for details).

cmovz(mem, reg);
cmovz(reg, reg);
cmovz(reg, mem);
	Conditional move if zero flag = 1 (see cmova for details).

cmp(imm, reg);
cmp(imm, mem);
cmp(reg, reg);
cmp(reg, mem);
cmp(mem, reg);
	Compare. Compares the first operand against the second operand. The two operands must be the same size. This instruction sets the condition code flags as appropriate for the condition jump and set instructions. This instruction does not change the value of either operand.

cmpsb();
repe.cmpsb();
repne.cmpsb();
	Compare string of bytes. Compares the byte pointed at byte ESI with the byte pointed at by EDI and then adjusts ESI and EDI by ±1 depending on the value of the direction flag. Sets the flags according to the result. With the REPNE (repeat while not equal) flag, this instruction compares up to ECX bytes until all the first byte it finds in the two string that are equal. With the REPE (repeat while equal) prefix, this instruction compares two strings up to the first byte that is different. See the chapter on the String Instructions for more details.

cmpsw()
repe.cmpsw();
repne.cmpsw();
	Compare a string of words. Like cmpsb except this instruction compares words rather than bytes and adjusts ESI/EDI by ±2.

cmpsd()
repe.cmpsd();
repne.cmpsd();
	Compare a string of double words. Like cmpsb except this instruction compares double words rather than bytes and adjusts ESI/EDI by ±4.

cmpxchg(reg, mem); cmpxchg(reg, reg);	Reg and mem must be the same size. They can be eight, 16, or 32 bit objects. This instruction compares the value in the accumulator (al, ax, or eax) against the second operand. If the two values are equal, this instruction copies the source (first) operand to the destination (second) operand. Otherwise, this instruction copies the second operand into the accumulator.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
cmpxchg8b(mem64);	Compares the 64-bit value in EDX:EAX with the memory operand. If the values are equal, then this instruction stores the 64-bit value in ECX:EBX into the memory operand and sets the zero flag. Otherwise, this instruction copies the 64-bit memory operand into the EDX:EAX registers and clears the zero flag.
cpuid();	CPU Identification. This instruction identifies various features found on the different Pentium processors. See the Intel documentation on this instruction for more details.
cwd();	Convert Word to Double. Sign extends AX to DX:AX.
cwde();	Convert Word to Double Word Extended. Sign extends AX to EAX.
daa();	Decimal Adjust after Addition. Adjusts value in AL after a decimal addition.
das();	Decimal Adjust after Subtraction. Adjusts value in AL after a decimal subtraction.
dec(reg); dec(mem);	Decrement. Subtracts one from the destination memory location or register.

div(reg);
div(reg8, ax);
div(reg16, dx:ax);
div(reg32, edx:eax);
div(mem);
div(mem8, ax);
div(mem16, dx:ax);
div(mem32, edx:eax);
div(imm8, ax);
div(imm16, dx:ax);
div(imm32, edx:eax);
	Divides accumulator or extended accumulator (dx:ax or edx:eax) by the source operand. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then divides the accumulator or extended accumulator by the contents of this memory location. Note that the accumulator operand is twice the size of the source (divisor) operand. This instruction computes the quotient and places it in AL, AX, or EAX and it computes the remainder and places it in AH, DX, or EDX (depending on the divisor's size). This instruction raises an exception if you attempt to divide by zero or if the quotient doesn't fit in the destination register (AL, AX, or EAX).
	
	This instruction performs an unsigned division.

enter(imm16, imm8);	Enter a procedure. Creates an activation record for a procedure. The first constant specifies the number of bytes of local variables. The second parameter (in the range 0..31) specifies the static nesting level (lex level) of the procedure.

idiv(reg);
idiv(reg8, ax);
idiv(reg16, dx:ax);
idiv(reg32, edx:eax);
idiv(mem);
idiv(mem8, ax);
idiv(mem16, dx:ax);
idiv(mem32, edx:eax);
idiv(imm8, ax);
idiv(imm16, dx:ax);
idiv(imm32, edx:eax);
	Divides accumulator or extended accumulator (dx:ax or edx:eax) by the source operand. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then divides the accumulator or extended accumulator by the contents of this memory location. Note that the accumulator operand is twice the size of the source (divisor) operand. This instruction computes the quotient and places it in AL, AX, or EAX and it computes the remainder and places it in AH, DX, or EDX (depending on the divisor's size). This instruction raises an exception if you attempt to divide by zero or if the quotient doesn't fit in the destination register (AL, AX, or EAX).
	
	This instruction performs a signed division. The condition code bits are undefined after executing this instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

imul(reg);
imul(reg8, al);
imul(reg16, ax);
imul(reg32, eax);
imul(mem);
imul(mem8, al);
imul(mem16, ax);
imul(mem32, eax);
imul(imm8, al);
imul(imm16, ax);
imul(imm32, eax);
	Multiplies the accumulator (AL, AX, or EAX) by the source operand. The source operand will be the same size as the accumulator. The product produces an operand that is twice the size of the two operands with the product winding up in AX, DX:AX, or EDX:EAX. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then multiplies the accumulator by the contents of this memory location.
	
	This instruction performs a signed multiplication. Also see INTMUL.
	This instruction sets the carry and overflow flag if the H.O. portion of the result (AH, DX, EDX) is not a sign extension of the L.O. portion of the product. The sign and zero flags are undefined after the execution of this instruction.

in(imm8, al);
in(imm8, ax);
in(imm8, eax);
in(dx, al);
in(dx, ax);
in(dx, eax);
	Input data from a port. These instructions read a byte, word, or double word from an input port and place the input data into the accumulator register. Immediate port constants must be in the range 0..255. For all other port addresses you must use the DX register to hold the 16-bit port number. Note that this is a privileged instruction that will raise an exception in many Win32 Operating Systems.

inc(reg); inc(mem);	Increment. Adds one to the specified memory or register operand. Does not affect the carry flag. Sets the overflow flag if there was signed overflow. Sets the zero and sign flags according to the result. Note that Z=1 indicates an unsigned overflow.
int(imm8);	Call an interrupt service routine specified by the immediate operand. Note that Windows does not use this instruction for system calls, so you will probably never use this instruction under Windows. Note that INT(3); is the user breakpoint instruction (that raises an appropriate exception). INT(0) is the divide error exception. INT(4) is the overflow exception. However, it's better to use the HLA RAISE statement than to use this instruction for these exceptions.

intmul(imm, reg);
intmul(imm, reg, reg);
intmul(imm, mem, reg);
intmul(reg, reg);
intmul(mem, reg);
	Integer mutiply. Multiplies the destination (last) operand by the source operand (if there are only two operands; or it multiplies the two source operands together and stores the result in the destination operand (if there are three operands). The operands must all be 16 or 32-bit operands and they must all be the same size.
	
	This instruction computes a signed product. This instruction sets the overflow and carry flags if there was a signed arithmetic overflow; the zero and sign flags are undefined after the execution of this instruction.

into();	Raises an exception if the overflow flag is set. Note: the HLA pseudo-variable "@into" controls the code generation for this instruction. If @into is false, HLA ignores this instruction; if @into is true (default), then HLA emits the object code for this instruction. Note that if the overflow flag is set, this instruction behaves like the "INT(4);" instruction.
iret();	Return from an interrupt. This instruction is not generally usable from an application program. It is for use in interrupt service routines only.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
ja *label*;	Conditional jump if (unsigned) above. You would generally use this instruction immediately after a CMP instruction to test to see if one operand is greater than another using an unsigned comparison. Control transfers to the specified label if this condition is true, control falls through to the next instruction if the condition is false.
jae *label*;	Conditional jump if (unsigned) above or equal. See JA above for details.
jb *label*;	Conditional jump if (unsigned) below. See JA above for details.
jbe *label*;	Conditional jump if (unsigned) below or equal. See JA above for details.
jc *label*;	Conditional jump if carry is one. See JA above for details.
je *label*;	Conditional jump if equal. See JA above for details.
jg *label*;	Conditional jump if (signed) greater. See JA above for details.
jge *label*;	Conditional jump if (signed) greater or equal. See JA above for details.
jl *label*;	Conditional jump if (signed) less than. See JA above for details.
jle *label*;	Conditional jump if (signed) less than or equal. See JA above for details.
jna *label*;	Conditional jump if (unsigned) not above. See JA above for details.
jnae *label*;	Conditional jump if (unsigned) not above or equal. See JA above for details.
jnb *label*;	Conditional jump if (unsigned) below. See JA above for details.
jnbe *label*;	Conditional jump if (unsigned) below or equal. See JA above for details.
jnc *label*;	Conditional jump if carry flag is clear (no carry). See JA above for details.
jne *label*;	Conditional jump if not equal. See JA above for details.
jng *label*;	Conditional jump if (signed) not greater. See JA above for details.
jnge *label*;	Conditional jump if (signed) not greater or equal. See JA above for details.
jnl *label*;	Conditional jump if (signed) not less than. See JA above for details.
jnle *label*;	Conditional jump if (signed) not less than or equal. See JA above for details.
jno *label*;	Conditional jump if no overflow (overflow flag = 0). See JA above for details.
jnp *label*;	Conditional jump if no parity/parity odd (parity flag = 0). See JA above for details.
jns *label*;	Conditional jump if no sign (sign flag = 0). See JA above for details.
jnz *label*;	Conditional jump if not zero (zero flag = 0). See JA above for details.
jo *label*;	Conditional jump if overflow (overflow flag = 1). See JA above for details.
jp *label*;	Conditional jump if parity (parity flag = 1). See JA above for details.
jpe *label*;	Conditional jump if parity even (parity flag = 1). See JA above for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
jpo *label*;	Conditional jump if parity odd (parity flag = 0). See JA above for details.
js *label*;	Conditional jump if sign (sign flag = 0). See JA above for details.
jz *label*;	Conditional jump if zero (zero flag = 0). See JA above for details.
jcxz *label*;	Conditional jump if CX is zero. See JA above for details. Note: the range of this branch is limited to ±128 bytes around the instruction. HLA does not check for this (MASM reports the error when it assembles HLA's output). Since this instruction is slower than comparing CX to zero and using JZ, you probably shouldn't even use this instruction. If you do, be sure that the target label is nearby in your code.
jecxz *label*;	Conditional jump if ECX is zero. See JA above for details. Note: the range of this branch is limited to ±128 bytes around the instruction. HLA does not check for this (MASM reports the error when it assembles HLA's output). Since this instruction is slower than comparing ECX to zero and using JZ, you probably shouldn't even use this instruction. If you do, be sure that the target label is nearby in your code.

jmp *label*;
jmp(*label*);
jmp *ProcedureName*;
jmp(mem32);
jmp(reg32);
	Jump Instruction. This instruction unconditionally transfers control to the specified destination operand. If the operand is a 32-bit register or memory location, the JMP instruction transfers control to the instruction whose address appears in the register or the memory location.
	
	Note: you should execise great care when jumping to a procedure label. The JMP instruction does not push a return address or any other data associated with a procedure's activation record. Hence, when the procedure attempts to return it will use data on the stack that was pushed prior to the execution of the JMP instruction; it is your responsibility to ensure such data is present on the stack when using JMP to transfer control to a procedure.

lahf();	Load AH from FLAGs. This instruction loads the AH register with the L.O. eight bits of the FLAGs register. See SAHF for the flag layout.
lea(reg32, mem); lea(mem, reg32);	Load Effective Address. These instructions, which are both semantically identical, load the 32-bit register with the address of the specified memory location. The memory location does not need to be a double word object. Note that there is never any ambiguity in this instruction since the register is always the destination operand and the memory location is always the source.
leave();	Leave procedure. This instruction cleans up the activation record for a procedure prior to returning from the procedure. You would normally use this instruction to clean up the activation record created by the ENTER instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

lock prefix	The lock prefix assets a special pin on the processor during the execution of the following instruction. In a multiprocessor environment, this ensures that the processor has exclusive use of a shared memory object while the instruction executes. The lock prefix may only precede one of the folowing instructions:
	ADD, ADC, AND BTC, BTR, BTS, CMPXCHG, DEC, INC NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. Furthermore, this prefix is only valid for the forms that have a memory operand as their destination operand. Any other instruction or addressing mode will raise an undefined opcode exception.
	
	HLA does not directly support the LOCK prefix on these instructions (if it did, you would normally write instructions like "lock.add();" and "lock.bts();" However, you can easily add this instruction to HLA's instruction set through the use of the following macro:
	`#macro lock;`
	`byte $F0; // $F0 is the opcode for the lock prefix.`
	`#endmacro;`
	
	To use this macro, simply precede the instruction you wish to lock with an invocation of the macro, e.g.,
	
	`lock add( al, mem );`
	
	Note that a LOCK prefix will dramatically slow an instruction down since it must access main memory (i.e., no cache) and it must negotiate for the use of that memory location with other processors in a multiprocessor system. The LOCK prefix has very little value in single processor systems.

lodsb();	Load String Byte. Loads AL with the byte whose address appears in ESI. Then it increments or decrements ESI by one depending on the value of the direction flag. See the chapter on string instructions for more details. Note: HLA does not allow the use of any repeat prefix with this instruction.
lodsw();	Load String Word. Loads AX from [ESI] and adds ±2 to ESI. See LODSB for more details.
loadsd();	Load String Double Word. Loads EAX from [ESI] and adds ±4 to ESI. See LODSB for more details.
loop *label*;	Decrements ECX and jumps to the target label if ECX is not zero. See JA for more details. Like JECX, this instruction is limited to a range of ±128 bytes around the instruction and only MASM will catch the range error. Since this instruction is actually slower than a DEC/JNZ pair, you should probably avoid using this instruction.
loope *label*;	Check the zero flag, decrement ECX, and branch if the zero flag was set and ECX did not become zero. Same limitations as LOOP. See LOOP above for details.
loopne *label*;	Check the zero flag, decrement ECX, and branch if the zero flag was clear and ECX did not become zero. Same limitations as LOOP. See LOOP above for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
loopnz *label*;	Same instruction at LOOPNE.
loopz *label*;	Same instruction as LOOPZ.

mov(imm, reg);
mov(imm, mem);
mov(reg, reg);
mov(reg, mem);
mov(mem, reg);
 	
mov(mem16, mem16);
mov(mem32, mem32);
	Move. Copies the data from the source (first) operand to the destination (second) operand. The operands must be the same size. Note that the memory to memory moves are an HLA extension. HLA compiles these statements into a push(source)/pop(dest) instruction pair.

movsb(); rep.movsb();	Move string of bytes. Copies the byte pointed at byte ESI to the byte pointed at by EDI and then adjusts ESI and EDI by ±1 depending on the value of the direction flag. With the REP (repeat) prefix, this instruction moves ECX bytes. See the chapter on the String Instructions for more details.
movsw(); rep.movsw();	Move string of words. Like MOVSB above except it copies words and adjusts ESI/EDI by ±2.
movsd(); rep.movsd();	Move string of words. Like MOVSB above except it copies double words and adjusts ESI/EDI by ±4.
movsx(reg, reg); movsx(mem, reg);	Move with sign extension. Copies the (smaller) source operand to the (larger) destination operand and sign extends the value to the size of the larger operand. The source operand must be smaller than the destination operand.
movzx(reg, reg); movzx(mem, reg);	Move with zero extension. Copies the (smaller) source operand to the (larger) destination operand and zero extends the value to the size of the larger operand. The source operand must be smaller than the destination operand.

mul(reg);
mul(reg8, al);
mul(reg16, ax);
mul(reg32, eax);
mul(mem);
mul(mem8, al);
mul(mem16, ax);
mul(mem32, eax);
mul(imm8, al);
mul(imm16, ax);
mul(imm32, eax);
	Multiplies the accumulator (AL, AX, or EAX) by the source operand. The source operand will be the same size as the accumulator. The product produces an operand that is twice the size of the two operands with the product winding up in AX, DX:AX, or EDX:EAX. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then multiplies the accumulator by the contents of this memory location.
	
	This instruction performs a signed multiplication. Also see INTMUL. The carry and overflow flags are cleared if the H.O. portion of the result is zero, they are set otherwise. The sign and zero flags are undefined after this instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
neg(reg); neg(mem);	Negate. Computes the two's complement of the operand and leaves the result in the operand. This instruction clears the carry flag if the result is zero, it sets the carry flag otherwise. It sets the overflow flag if the original value was the smallest possible negative value (which has no positive counterpart). It sets the sign and zero flags according to the result obtained.
nop();	No Operation. Consumes space and time but does nothing else. Same instruction as "xchg(eax, eax);"
not(reg); not(mem);	Bitwise NOT. Inverts all the bits in its operand. **Note**: this instruction does not affect any flags.

or(imm, reg);
or(imm, mem);
or(reg, reg);
or(reg, mem);
or(mem, reg);
	Bitwise OR. Logically ORs the source operand with the destination operand and leaves the result in the destination. The two operands must be the same size. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

out(al, imm8);
out(ax, imm8);
out(eax, imm8);
out(al, dx);
out(ax, dx);
out(eax, dx);
	Outputs the accumulator to the specified port. See the IN instruction for limitations under Win32.

pop(reg); pop(mem);	Pop a value off the stack. Operands must be 16 or 32 bits.
popa();	Pops all the 16-bit registers off the stack. The popping order is DI, SI, BP, SP, BX, DX, CX, AX.
popad();	Pops all the 32-bit registers off the stack. The popping order is EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.
popf();	Pops the 16-bit FLAGS register off the stack. Note that in user (application) mode, this instruction ignores the interrupt disable flag value it pops off the stack.
popfd();	Pops the 32-bit EFLAGS register off the stack. Note that in user (application) mode, this instruction ignores many of the bits it pops off the stack.
push(reg); push(mem);	Pushes the specified 16-bit or 32-bit register or memory location onto the stack. Note that you cannot push eight-bit objects.
pusha();	Pushes all the 16-bit general purpose registers onto the stack in the roder AX, CX, DX, BX, SP, BP, SI, DI.
pushad();	Pushes all the 32-bit general purpose registers onto the stack in the roder EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

pushd(imm);
pushd(reg);
pushd(mem);
	Pushes the 32-bit operand on to the stack. Generally used to push constants or anonymous variables. Note that this is a synonym for PUSH if you specify a register or typed memory operand.

pushf();	Pushes the value of the 16-bit FLAGS register onto the stack.
pushfd();	Pushes the value of the 32-bit FLAGS register onto the stack.

pushw(imm);
pushw(reg);
pushw(mem);
	Pushes the 16-bit operand on to the stack. Generally used to push constants or anonymous variables. Note that this is a synonym for PUSH if you specify a register or typed memory operand.

rcl(imm, reg);
rcl(imm, mem);
rcl(cl, reg);
rcl(cl, mem);
	Rotate through carry, left. Rotates the destination (second) operand through the carry the number of bits specified by the first operand, shifting the bits from the L.O. to the H.O. position (i.e., rotate left). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

rcr(imm, reg);
rcr(imm, mem);
rcr(cl, reg);
rcr(cl, mem);
	Rotate through carry, right. Rotates the destination (second) operand through the carry the number of bits specified by the first operand, shifting the bits from the H.O. to the L.O. position (i.e., rotate right).
	The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

rdtsc();	Read Time Stamp Counter. Returns in EDX:EAX the number of clock cycles that have transpired since the last reset of the processor. You can use this instruction to time events in your code (i.e., to determine whether one instruction sequence takes more time than another).
ret(); ret(imm16);	Return from subroutine. Pops a return address off the stack and transfers control to that location. The second form of the instruction adds the immediate constant to the ESP register to remove the procedure's parameters from the stack.

rol(imm, reg);
rol(imm, mem);
rol(cl, reg);
rol(cl, mem);
	Rotate left. Rotates the destination (second) operand the number of bits specified by the first operand, shifting the bits from the L.O. to the H.O. position (i.e., rotate left). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

ror(imm, reg);
ror(imm, mem);
ror(cl, reg);
ror(cl, mem);
	Rotate right. Rotates the destination (second) operand the number of bits specified by the first operand, shifting the bits from the H.O. to the L.O. position (i.e., rotate right). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
sahf();	Store AH into FLAGs. Copies the value in AH into the L.O. eight bits of the FLAGs register. Note that this instruction will not affect the interrupt disable flag when operating in user (application) mode. Bit #7 of AH goes into the Sign flag, bit #6 goes into the zero flag, bit #4 goes into the auxilary carry (BCD carry) flag, bit #2 goes into the parity flag, and bit #0 goes into the carry flag. This instruction also clears bits one, three, and five of the FLAGs register. It does not affect any other bits in FLAGs or EFLAGs.

sal(imm, reg);
sal(imm, mem);
sal(cl, reg);
sal(cl, mem);
	Shift Arithmetic Left. Same instruction as SHL. See SHL for details.

sar(imm, reg);
sar(imm, mem);
sar(cl, reg);
sar(cl, mem);
	Shift Arithmetic Right. Shifts the destination (second) operand to the right the specified number of bits using an arithmetic shift right algorithm. The carry flag contains the value of the last bit shifted out of the second operand. The overflow flag is only defined when the bit shift count is one, this instruction always clears the overflow flag. The sign and zero flags are set according to the result.

sbb(imm, reg);
sbb(imm, mem);
sbb(reg, reg);
sbb(reg, mem);
sbb(mem, reg);
	Subtract with borrow. Subtracts the source (first) operand and the carry from the destination (second) operand. Sets the condition code bits according to the result it computes. This instruction sets the flags the same way as the SUB instruction. See SUB for details.

scasb();
repe.scasb();
repne.scasb();
	Scan string byte. Compares the value in AL against the byte that EDI points at and sets the flags accordingly (same as the CMP instruction). Adds ±1 to EDI after the comparison (based on the setting of the direction flag). With the REPE (repeat while equal) prefix, this instruction will scan through as many as ECX bytes in memory as long as each byte that EDI points at is equal to the value in AL (i.e., it scans for the first value not equal to the value in AL). With the REPNE prefix, this instruction scans through as many as ECX bytes as long as the value that EDI points at is not equal to AL (i.e., it scans for the first byte matching AL's value). See the chapter on string instructions for more details.

scasw();
repe.scasw();
repne.scasw();
	Scan String Word. Compares the value in AX against the word that EDI points at and set the flags. Adds ±2 to EDI after the operation. Also supports the REPE and REPNE prefixes (see SCASB above).

scasd();
repe.scasd();
repne.scasd();
	Scan String Double word. Compares the value in EAX against the double word that EDI points at and set the flags. Adds ±4 to EDI after the operation. Also supports the REPE and REPNE prefixes (see SCASB above).

seta(reg); seta(mem);	Conditional set if (unsigned) above (Carry=0 and Zero=0). Stores a one in the destination operand if the result of the previous comparison found the first operand to be greater than the second using an unsigned comparison. Stores a zero into the destination operand otherwise.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
setae(reg); setae(mem);	Conditional set if (unsigned) above or equal (Carry=0). See SETA for details.
setb(reg); setb(mem);	Conditional set if (unsigned) below (Carry=1). See SETA for details.
setbe(reg); setbe(mem);	Conditional set if (unsigned) below or equal (Carry=1 or Zero=1). See SETA for details.
setc(reg); setc(mem);	Conditional set if carry set (Carry=1). See SETA for details.
sete(reg); sete(mem);	Conditional set if equal (Zero=1). See SETA for details.
setg(reg); setg(mem);	Conditional set if (signed) greater (Sign=Overflow and Zero=0). See SETA for details.
setge(reg); setge(mem);	Conditional set if (signed) greater or equal (Sign=Overflow or Zero=1). See SETA for details.
setl(reg); setl(mem);	Conditional set if (signed) less than (Sign<>Overflow). See SETA for details.
setle(reg); setle(mem);	Conditional set if (signed) less than or equal (Sign<>Overflow or Zero = 1). See SETA for details.
setna(reg); setna(mem);	Conditional set if (unsigned) not above (Carry=1 or Zero=1). See SETA for details.
setnae(reg); setnae(mem);	Conditional set if (unsigned) not above or equal (Carry=1). See SETA for details.
setnb(reg); setnb(mem);	Conditional set if (unsigned) not below (Carry=0). See SETA for details.
setnbe(reg); setnbe(mem);	Conditional set if (unsigned) not below or equal (Carry=0 and Zero=0). See SETA for details.
setnc(reg); setnc(mem);	Conditional set if carry clear (Carry=0). See SETA for details.
setne(reg); setne(mem);	Conditional set if not equal (Zero=0). See SETA for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
setng(reg); setng(mem);	Conditional set if (signed) not greater (Sign<>Overflow or Zero = 1). See SETA for details.
setnge(reg); setnge(mem);	Conditional set if (signed) not greater than (Sign<>Overflow). See SETA for details.
setnl(reg); setnl(mem);	Conditional set if (signed) not less than (Sign=Overflow or Zero=1). See SETA for details.
setnle(reg); setnle(mem);	Conditional set if (signed) not less than or equal (Sign=Overflow and Zero=0). See SETA for details.
setno(reg); setno(mem);	Conditional set if no overflow (Overflow=0). See SETA for details.
setnp(reg); setnp(mem);	Conditional set if no parity (Parity=0). See SETA for details.
setns(reg); setns(mem);	Conditional set if no sign (Sign=0). See SETA for details.
setnz(reg); setnz(mem);	Conditional set if not zero (Zero=0). See SETA for details.
seto(reg); seto(mem);	Conditional set if Overflow (Overflow=1). See SETA for details.
setp(reg); setp(mem);	Conditional set if Parity (Parity=1). See SETA for details.
setpe(reg); setpe(mem);	Conditional set if Parity even (Parity=1). See SETA for details.
setpo(reg); setpo(mem);	Conditional set if Parity odd (Parity=0). See SETA for details.
sets(reg); sets(mem);	Conditional set if sign set(Sign=1). See SETA for details.
setz(reg); setz(mem);	Conditional set if zero (Zero=1). See SETA for details.

shl(imm, reg);
shl(imm, mem);
shl(cl, reg);
shl(cl, mem);
	Shift left. Shifts the destination (second) operand to the left the number of bit positions specified by the first operand. The carry flag contains the value of the last bit shifted out of the second operand. The overflow flag is only defined when the bit shift count is one, this instruction sets overflow flag if the sign changes as a result of this instruction's execution. The sign and zero flags are set according to the result.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

shld(imm8, reg, reg);
shld(imm8, reg, mem);
shld(cl, reg, reg);
shld(cl, reg, mem);
	Shift Left Double precision. The first operand is a bit count. The second operand is a source and the third operand is a destination. These operands must be the same size and they must be 16- or 32-bit values (no eight bit operands). This instruction treats the second and third operands as a double precision value with the second operand being the L.O. word or double word and the third operand being the H.O. word or double word. The instruction shifts this double precision value the specified number of bits and sets the flags in a manner identical to SHL. Note that this instruction does not affect the source (second) operand's value.

shr(imm, reg);
shr(imm, mem);
shr(cl, reg);
shr(cl, mem);
	Shift right. Shifts the destination (second) operand to the right the number of bit positions specified by the first operand. The last bit shifted out goes into the carry flag. The overflow flag is set if the H.O. bit originally contained one. The sign flag is cleared and the zero flag is set if the result is zero.

shrd(imm8, reg, reg);
shrd(imm8, reg, mem);
shrd(cl, reg, reg);
shrd(cl, reg, mem);
	Shift Right Double precision. The first operand is a bit count. The second operand is a source and the third operand is a destination. These operands must be the same size and they must be 16- or 32-bit values (no eight bit operands). This instruction treats the second and third operands as a double precision value with the second operand being the H.O. word or double word and the third operand being the L.O. word or double word. The instruction shifts this double precision value the specified number of bits and sets the flags in a manner identical to SHR. Note that this instruction does not affect the source (second) operand's value.

stc();	Set Carry. Sets the carry flag to one.
std();	Set Direction. Sets the direction flag to one. If the direction flag is one, the string instructions decrement ESI and/or EDI after each operation.
sti();	Set interrupt enable flag. Generally this instruction is not usable in user (application) mode. In kernel mode it allows the CPU to begin processing interrupts.
stosb(); rep.stosb();	Store String Byte. Stores the value in AL at the location whose address EDI contains. Then in adds ±1 to EDI. If the REP prefix is present, this instruction repeats the number of times specified in the ECX register. This instruction is useful for quickly clearing out byte arrays.
stosw(); rep.stosw();	Store String Word. Stores the value in AX at location [EDI] and then adds ±2 to EDI. See STOSB for details.
stosd(); rep.stosd();	Store String Double word. Stores the value in EAX at location [EDI] and then adds ±4 to EDI. See STOSB for details.

sub(imm, reg);
sub(imm, mem);
sub(reg, reg);
sub(reg, mem);
sub(mem, reg);
	Subtract. Subtracts the first operand from the second operand and leaves the difference in the destination (second) operand. Sets the zero flag if the two values were equal (which produces a zero result), sets the carry flag if there was unsigned overflow or underflow; sets the overflow if there was signed overflow or underflow; sets the sign flag if the result is negative (H.O. bit is one). Note that SUB sets the flags identically to the CMP instruction, so you can use conditional jump or set instructions after SUB the same way you would use them after a CMP instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

test(imm, reg);
test(imm, mem);
test(reg, reg);
test(reg, mem);
test(mem, reg);
	Test operands. Logically ANDs the two operands together and sets the flags but does not store the computed result (i.e., it does not disturb the value in either operand). Always clears the carry and overflow flags. Sets the sign flag if the H.O. bit of the computed result is one. Sets the zero flag if the computed result is zero.

xadd(mem, reg);
xadd(reg, reg);
	Adds the first operand to the second operand and then stores the original value of the second operand into the first operand:
	`xadd( source, dest );`
	`temp := dest`
	`dest := dest + source`
	`source := temp`
	This instruction sets the flags in a manner identical to the ADD instruction.

xchg(reg, reg);
xchg(reg, mem);
xchg(mem, reg);
	Swaps the values in the two operands which must be the same size. Does not affect any flags.

xlat();	Translate. Computes AL := [EBX + AL]; That is, it uses the value in AL as an index into a lookup table whose base address is in EBX. It copies the specified byte from this table into AL.

xor(imm, reg);
xor(imm, mem);
xor(reg, reg);
xor(reg, mem);
xor(mem, reg);
	Exclusive-OR. Logically XORs the source operand with the destination operand and leaves the result in the destination. The two operands must be the same size. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

Table 2: Floating Point Instruction Set

Instruction	Description
f2xm1();	Compute 2^x-1 in ST0, leaving the result in ST0.
fabs();	Computes the absolute value of ST0.

fadd(mem);
fadd(st*i*, st0);
fadd(st0, st*i*);
	Add operand to st0 or add st0 to destination register (st*i*, i=0..7). If the operand is a memory operand, it must be a *real32* or *real64* object.

faddp(); faddp(st0, st*i*);	With no operands, this instruction adds st0 to st1 and then pops st0 off the FPU stack.

Table 2: Floating Point Instruction Set

Instruction	Description
fbld(mem80);	This instruction loads a ten-byte (80-bit) packed BCD value from memory and converts it to a real80 object. This instruction does not check for an invalid BCD value. If the BCD number contains illegal digits, the result is undefined.
fbstp(mem80);	This instruction pops the real80 object off the top of the FPU stack, converts it to an 80-bit BCD value, and stores the result in the specified memory location (tbyte).
fchs();	This instruction negates the floating point value on the top of the stack (st0).
fclex();	This instruction clears the floating point exception flags.
fcmova(st*i*, st0);[a]	Floating point conditional move if above. Copies st*i* to st0 if c=0 and z=0 (unsigned greater than after a CMP).
fcmovae(st*i*, st0);	Floating point conditional move if above or equal. Copies st*i* to st0 if c=0 (unsigned greater or equal after a CMP).
fcmovb(st*i*, st0);	Floating point conditional move if below. Copies st*i* to st0 if c=1 (unsigned less than after a CMP).
fcmovbe(st*i*, st0);	Floating point conditional move if below or equal. Copies st*i* to st0 if c=1 or z=1 (unsigned less than or equal after a CMP).
fcmove(st*i*, st0);	Floating point conditional move if equal. Copies st*i* to st0 if z=1 (equal after a CMP).
fcmovna(st*i*, st0);	Floating point conditional move if not above. Copies st*i* to st0 if c=1 or z=1 (unsigned not above after a CMP).
fcmovnae(st*i*, st0);	Floating point conditional move if not above or equal. Copies st*i* to st0 if c=1 (unsigned not above or equal after a CMP).
fcmovnb(st*i*, st0);	Floating point conditional move if not below. Copies st*i* to st0 if c=0 (unsigned not below after a CMP).
fcmovnbe(st*i*, st0);	Floating point conditional move if not below or equal. Copies st*i* to st0 if c=0 and z=0 (unsigned not below or equal after a CMP).
fcmovne(st*i*, st0);	Floating point conditional move if not equal. Copies st*i* to st0 if z=0 (not equal after a CMP).
fcmovnu(st*i*, st0);	Floating point conditional move if not unordered. Copies st*i* to st0 if the last floating point comparison did not produce an unordered result (parity flag = 0).
fcmovu(st*i*, st0);	Floating point conditional move if not unordered. Copies st*i* to st0 if the last floating point comparison produced an unordered result (parity flag = 1).

fcom();
fcom(mem);
fcom(st0, st*i*);
	Compares the value of ST0 with the operand and sets the floating point condition bits based on the comparison. If the operand is a memory operand, it must be a *real32* or *real64* value. Note that to test the condition codes you will have to copy the floating point status word to the FLAGs register; see the chapter on floating point arithmetic for details.

Table 2: Floating Point Instruction Set

Instruction	Description
fcomi(st0, st*i*);[b]	Compares the value of ST0 with the second operand and sets the appropriate bits in the FLAGs register.
fcomip(st0, st*i*);	Compares the value of ST0 with the second operand, sets the appropriate bits in the FLAGs register, and then pops ST0 off the FPU stack.

fcomp();
fcomp(mem);
fcomp(st*i*);
	Compares the value of ST0 with the operand, sets the floating point status bits, and then pops ST0 off the floating point stack. With no operands, this instruction compares ST0 to ST1. Memory operands must be *real32* or *real64* objects.

fcompp();	Compares ST0 to ST1 and then pops both values off the stack. Leaves the result of the comparison in the floating point status register.
fcos();	Computes ST0 = cos(ST0).
fdecstp();	Rotates the items on the FPU stack.

fdiv(mem);
fdiv(st*i*, st0);
fdiv(st0, st*i*);
	Floating point division. If a memory operand is present, it must be a *real32* or *real64* object; FDIV will divide ST0 by the memory operand and leave the quotient in ST0. If the FDIV operands are registers, FDIV diviides the destination (second) operand by the source (first) operand and leaves the result in the destination operand.

fdivp(); fdivp(st*i*);	With no operands, this instruction divides ST1 by ST0, pops ST0, and replaces the new top of stack with the quotient (replacing the previous ST1 value).

fdivr(mem);
fdivr(st*i*, st0);
fdivr(st0, st*i*);
	Floating point divide with reversed operands. Like FDIV, but computes operand/ST0 rather than ST0/operand.

fdivrp(); fdivrp(st*i*);	Floating point divide and pop, reversed. Like FDIVP except it computes operand/ST0 rather than ST0/operand.
ffree(st*i*);	Frees the specified floating point register.
fiadd(mem);	Memory operand must be a 16-bit or 32-bit signed integer. This instruction converts the integer to a real, pushes the value, and then executes FADDP();
ficom(mem);	Floating point compare to integer. Memory operand must be an *int16* or *int32* object. This instruction converts the memory operand to a *real80* value and compares ST0 to this value and sets the status bits in the floating point status register.
ficomp(mem);	Floating point compare to integer and pop. Memory operand must be an *int16* or *int32* object. This instruction converts the memory operand to a *real80* value and compares ST0 to this value and sets the status bits in the floating point status register. After the comparison, this instructions pop ST0 from the FPU stack.
fidiv(mem);	Floating point divide by integer. Memory operand must be an *int16* or *int32* object. These instructions convert their integer operands to a *real80* value and then divide ST0 by this value, leaving the result in ST0.

Table 2: Floating Point Instruction Set

Instruction	Description
fidivr(mem);	Floating point divide by integer, reversed. Like FIDIV above, except this instruction computes mem/ST0 rather than ST0/mem.
fild(mem);	Floating point load integer. Mem operand must be an int16 or int32 object. This instructions converts the integer to a real80 object and pushes it onto the FPU stack.
fimul(mem);	Floating point multiply by integer. Converts *int16* or *int32* operand to a *real80* value and multiplies ST0 by this result. Leaves product in ST0.
fincstp();	Rotates the registers on the FPU stack.
finit();	Initializes the FPU for use.
fist(mem);	Converts ST0 to an integer and stores the result in the specified memory operand. Memory operand must be an *int16* or *int32* object.
fistp(mem);	Floating point integer store and pop. Pops ST0 value off the stack, converts it to an integer, and stores the integer in the specified location. Memory operand must be a word, double word, or quad word (64-bit integer) object.
fisub(mem);	Floating point subtract integer. Converts *int16* or *int32* operand to a *real80* value and subtracts it from ST0. Leaves the result in ST0.
fisubr(mem);	Floating point subtract integer, reversed. Like FISUB except this instruction compute mem-ST0 rather than ST0-mem. Still leaves the result in ST0.
fld(mem); fld(st*i*);	Floating point load. Loads (pushes) the specified operand onto the FPU stack. Memory operands must be *real32*, *real64*, or *real80* objects. Note that FLD(ST0) duplicates the value on the top of the floating point stack.
fld1();	Floating point load 1.0. This instruction pushes 1.0 onto the FPU stack.
fldcw(mem16);	Load floating point control word. This instruction copies the word operand into the floating point control register.
fldenv(mem28);	This instruction loads the FPU status from the block of 28 bytes specified by the operand. Generally, only an operating system would use this instruction.
fldl2e();	Floating point load constant. Loads $\log_2(e)$ onto the stack.
fldl2t();	Floating point load constant. Loads $\log_2(10)$ onto the stack.
fldlg2();	Floating point load constant. Loads $\log_{10}(2)$ onto the stack.
fldln2();	Floating point load constant. Loads $\log_e(2)$ onto the stack.
fldpi();	Floating point load constant. Loads the value of pi (π) onto the stack.
fldz();	Floating point load constant. Pushes the value 0.0 onto the stack.

Table 2: Floating Point Instruction Set

Instruction	Description

fmul(mem);
fmul(st*i*, st0);
fmul(st0, st*i*);
	Floating point multiply. If the operand is a memory operand, it must be a real32 or real64 value; in this case, FMUL multiplies the memory operand and ST0, leaving the product in ST0. For the other two forms, the FMUL instruction multiplies the first operand by the second and leaves the result in the second operand.

fmulp(); fmulp(st0, st*i*);	Floating point multiply and pop. With no operands this instruction computes ST1:=ST0*ST1 and then pops ST0. With two register operands, this instruction computes ST0 times the destination register and then pops ST0.
fnop();	Floating point no-operation.
fpatan();	Floating point partial arctangent. Computes ATAN(ST1/ST0), pops ST0, and then stores the result in the new TOS value (previous ST1 value).
fprem();	Floating point remainder. This instruction is retained for compatibility with older programs. Use the FPREM1 instruction instead.
fprem1();	Floating point partial remainder. This instruction computes the remainder obtained by dviding ST0 by ST1, leaving the result in ST0 (it does not pop either operand). If the C2 flag in the FPU status register is set after this instruction, then the computation is not complete; you must repeatedly execute this instruction until C2 is cleared.
fptan();	Floating point partial tangent. This instruction computes TAN(ST0) and replaces the value in ST0 with this result. Then it pushes 1.0 onto the stack. This instruction sets the C2 flag if the input value is outside the acceptable range of $\pm 2^{63}$.
frndint();	Floating point round to integer. This instruction rounds the value in ST0 to an integer using the rounding control bits in the floating point control register. Note that the result left on TOS is still a real value. It simply doesn't have a fractional component. You may use this instruction to round or truncate a floating point value by setting the rounding control bits appropriately. See the chapter on floating point arithmetic for details.
frstor(mem108);	Restores the FPU status from a 108-byte memory block.
fsave(mem108);	Writes the FPU status to a 108-bye memory block.
fscale();	Floating point scale by power of two. ST1 contains a scaling value. This instruction multiplies ST0 by 2^{st1}.
fsin();	Floating point sine. Replaces ST0 with sin(ST0).
fsincos();	Simultaneously computes the sin and cosine values of ST0. Replaces ST0 with the sine of ST0 and then it pushes the cosine of (the original value of) ST0 onto the stack. Original ST0 value must be in the range $\pm 2^{63}$.
fsqrt();	Floating point square root. Replaces ST0 with the square root of ST0.

Table 2: Floating Point Instruction Set

Instruction	Description
fst(mem); fst(st*i*);	Floating point store. Stores a copy of ST0 in the destination operand. Memory operands must be *real32* or *real64* objects. When storing the value to memory, FST converts the value to the smaller format using the rounding control bits in the floating point control register to determine how to convert the *real80* value in ST0 to a *real32* or *real64* value.
fstcw(mem16);	Floating point store control word. Stores a copy of the floating point control word in the specified *word* memory location.
fstenv(mem28);	Floating point store FPU environment. Stores a copy of the 28-byte floating point environment in the specified memory location. Normally, an OS would use this when switch contexts.
fstp(mem); fstp(st*i*);	Floating point store and pop. Stores ST0 into the destination operand and then pops ST0 off the stack. If the operand is a memory object, it must be a *real32*, *real64*, or *real80* object.
fstsw(ax); fstsw(mem16);	Stores a copy of the 16-bit floating point status register into the specified word operand. Note that this instruction automatically places the C1, C2, C3, and C4 condition bits in appropriate places in AH so that a following SAHF instruction will set the processor flags to allow the use of a conditional jump or conditional set instruction after a floating point comparison. See the chapter on floating point arithmetic for more details.

fsub(mem);
fsub(st0, st*i*);
fsub(st*i*, st0);
	Floating point subtract. With a single memory operand (which must be a real32 or real64 object), this instruction subtracts the memory operand from ST0. With two register operands, this instruction computes *dest := dest - src* (where *src* is the first operand and *dest* is the second operand).

fsubp(); fsubp(st0, st*i*);	Floating point subtract and pop. With no operands, this instruction computes ST1 := ST0 - ST1 and then pops ST0 off the stack. With two operands, this instruction computes STi := ST*i* - ST0 and then pops ST0 off the stack.

fsubr(mem);
fsubr(st0, st*i*);
fsubr(st*i*, st0);
	Floating point subtract, reversed, With a real32 or real64 memory operand, this instruction computes ST0 := mem - ST0. For the other two forms, this instruction computes *dest := src - dest* where *src* is the first operand and *dest* is the second operand.

fsubrp(); fsubrp(st0, st*i*);	Floating point subtract and pop, reversed. With no operands, this instruction computes ST1 := ST0 - ST1 and then pops ST0. With two operands, this instruction computes STi := ST0 - STi and then pops ST0 from the stack.
ftst();	Floating point test against zero. Compares ST0 with 0.0 and sets the floating point condition code bits accordingly.

Table 2: Floating Point Instruction Set

Instruction	Description
fucom(st*i*); fucom();	Floating point unordered comparison. With no operand, this instruction compares ST0 to ST1. With an operand, this instruction compares ST0 to STi and sets floating point status bits accordingly. Unlike FCOM, this instruction will not generate an exception if either of the operands is an illegal floating point value; instead, this sets a special status value in the FPU status register.
fucomi(st*i*, st0);	Floating point unordered comparison.
fucomp(); fucomp(st*i*);	Floating point unorder comparison and pop. With no operands, compares ST0 to ST1 using an unordered comparsion (see FUCOM) and then pops ST0 off the stack. With an FPU operand, this instruction compares ST0 to the specified register and then pops ST0 off the stack. See FUCOM for more details.
fucompp(); fucompp(st*i*);	Floating point unordered compare and double pop. Compares ST0 to ST1, sets the condition code bits (without raising an exception for illegal values, see FUCOM), and then pops both ST0 and ST1.
fwait();	Floating point wait. Waits for current FPU operation to complete. Generally an obsolete instruction. Used back in the days when the FPU was on a different chip than the CPU.
fxam();	Floating point Examine ST0. Checks the value in ST0 and sets the condition code bits according to the type of the value in ST0. See the chapter on floating point arithmetic for details.
fxch(); fxch(st*i*);	Floating point exchange. With no operands this instruction exchanges ST0 and ST1 on the FPU stack. With a single register operand, this instruction swaps ST0 and STi.
fxtract();	Floating point exponent/mantissa extraction. This instruction breaks the value in ST0 into two pieces. It replaces ST0 with the real representation of the binary exponent (e.g. 2^5 becomes 5.0) and then it pushes the mantissa of the value with an exponent of zero.
fyl2x();	Floating point partial logarithm computation. Computes ST1 := ST1 * $\log_2$(ST0); and then pops ST0.
fyl2xp1();	Floating point partial logarithm computation. Computes ST1 := ST1 * $\log_2$(ST0 + 1.0) and then pops ST0 off the stack. Original ST0 value must be in the range $(-\left(1-\frac{\sqrt{2}}{2}\right),\left(1-\frac{\sqrt{2}}{2}\right))$

a. Floating point conditional move instructions are only available on Pentium Pro and later processors.
b. FCOMIx instructions are only available on Pentium Pro and later processors.

The following table uses these abbreviations:

Reg32- A 32-bit general purpose (integer) register.

mm*i*- One of the eight MMX registers, MM0..MM7.

imm8- An eight-bit constant value; some instructions have smaller ranges that 0..255. See the particular instruction for details.

mem64- A memory location (using an arbitrary addressing mode) that references a qword value.

Note: Most instructions have two operands. Typically the first operand is a source operand and the second operand is a destination operand. For exceptions, see the description of the instruction.

Table 3: MMX Instruction Set

Instruction	Description
emms();	Empty MMX State. You must execute this instruction when you are finished using MMX instructions and before any following floating point instructions.

movd(reg32, mm*i*);
movd(mem32, mm*i*);
movd(mm*i*, reg32);
movd(mm*i*, mem32);
	Moves data between a 32-bit integer register or dword memory location and an MMX register (mm0..mm7). If the destination operand is an MMX register, then the source operand is zero-extended to 64 bits during the transfer. If the destination operand is a dword memory location or 32-bit register, this instruction copies only the L.O. 32 bits of the MMX register to the destination.

movq(mem64, mm*i*);
movq(mm*i*, mem64);
movq(mm*i*, mm*i*);
	This instruction moves 64 bits between an MMX register and a *qword* variable in memory or between two MMX registers.

packssdw(mem64, mm*i*); packssdw(mm*i*, mm*i*);	Pack and saturate two signed double words from source and two signed double words from destination and store result into destination MMX register. This process involves taking these four double words and "saturating" them. This means that if the value is in the range -32768..32768 the value is left unchanged, but if it's greater than 32767 the value is set to 32767 or if it's less than -32768 the value is clipped to -32768. The four double words are packed into a single 64-bit MMX register. The source operand supplies the upper two words and the destination operand supplies the lower two words of the packed 64-bit result. See the chapter on the MMX instructions for more details.
packsswb(mem64, mm*i*); packsswb(mm*i*, mm*i*);	Pack and saturate four signed words from source and four signed words from destination and store the result as eight signed bytes into the destination MMX register. See the chapter on the MMX instructions for more details. The bytes obtained from the destination register wind up in the L.O. four bytes of the destination; the bytes computed from the signed saturation of the source register wind up in the H.O. four bytes of the destination register.

Table 3: MMX Instruction Set

Instruction	Description
packusdw(mem64, mm*i*); packusdw(mm*i*, mm*i*);	Pack and saturate two unsigned double words from source and two unsigned double words from destination and store result into destination MMX register. This process involves taking these four double words and "saturating" them. This means that if the value is in the range 0..65535 the value is left unchanged, but if it's greater than 65535 the value is clipped to 65535. The four double words are packed into a single 64-bit MMX register. The source operand supplies the upper two words and the destination operand supplies the lower two words of the packed 64-bit result. See the chapter on the MMX instructions for more details.
packuswb(mem64, mm*i*); packuswb(mm*i*, mm*i*);	Pack and saturate four unsigned words from source and four unsigned words from destination and store the result as eight unsigned bytes into the destination MMX register. Word values greater than 255 are clipped to 255 during the saturation operation. See the chapter on the MMX instructions for more details. The bytes obtained from the destination register wind up in the L.O. four bytes of the destination; the bytes computed from the signed saturation of the source register wind up in the H.O. four bytes of the destination register.
paddb(mem64, mm*i*); paddb(mm*i*, mm*i*);	Packed Add of Bytes. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow occurs in any byte, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
paddd(mem64, mm*i*); paddd(mm*i*, mm*i*);	Packed Add of Double Words. This instruction adds together the individual dwords of the two operands. The addition of each dword is independent of the other two dwords; there is no carry from dword to dword. If an overflow occurs in any dword, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
paddsb(mem64, mm*i*); paddsb(mm*i*, mm*i*);	Packed Add of Bytes, signed saturated. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow or underflow occurs in any byte, then the value saturates at -128 or +127. This instruction does not affect any flags.
paddsw(mem64, mm*i*); paddsw(mm*i*, mm*i*);	Packed Add of Words, signed saturated. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow or underflow occurs in any word, the value saturates at either -32768 or +32767. This instruction does not affect any flags.
paddusb(mem64, mm*i*); paddusb(mm*i*, mm*i*);	Packed Add of Bytes, unsigned saturated. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow or underflow occurs in any byte, then the value saturates at 0 or 255. This instruction does not affect any flags.

Table 3: MMX Instruction Set

Instruction	Description
paddusw(mem64, mm*i*); paddusw(mm*i*, mm*i*);	Packed Add of Words, unsigned saturated. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow or underflow occurs in any word, the value saturates at either 0 or 65535. This instruction does not affect any flags.
paddw(mem64, mm*i*); paddw(mm*i*, mm*i*);	Packed Add of Words. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow occurs in any word, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
pand(mem64, mm*i*); pand(mm*i*, mm*i*);	Packed AND. This instruction computes the bitwise AND of the source and the destination values, leaving the result in the destination. This instruction does not affect any flags.
pandn(mem64, mm*i*); pandn(mm*i*, mm*i*);	Packed AND NOT. This instruction makes a temporary copy of the first operand and inverts all of the bits in this copy; then it ANDs this value with the destination MMX register. This instruction does not affect any flags.
pavgb(mem64, mm*i*); pavgb(mm*i*, mm*i*);	Packed Average of Bytes. This instruction computes the average of the eight pairs of bytes in the two operands. It leaves the result in the destination (second) operand.
pavgw(mem64, mm*i*); pavgw(mm*i*, mm*i*);	Packed Average of Words. This instruction computes the average of the four pairs of words in the two operands. It leaves the result in the destination (second) operand.
pcmpeqb(mem64, mm*i*); pcmpeqb(mm*i*, mm*i*);	Packed Compare for Equal Bytes. This instruction compares the individual bytes in the two operands. If they are equal this instruction sets the corresponding byte in the destination (second) register to $FF (all ones); if they are not equal, this instruction sets the corresponding byte to zero.
pcmpeqd(mem64, mm*i*); pcmpeqd(mm*i*, mm*i*);	Packed Compare for Equal Double Words. This instruction compares the individual double wordsin the two operands. If they are equal this instruction sets the corresponding double wordin the destination (second) register to $FFFF_FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero.
pcmpeqw(mem64, mm*i*); pcmpeqw(mm*i*, mm*i*);	Packed Compare for Equal Words. This instruction compares the individual words in the two operands. If they are equal this instruction sets the corresponding word in the destination (second) register to $FFFF (all ones); if they are not equal, this instruction sets the corresponding word to zero.

Table 3: MMX Instruction Set

Instruction	Description
pcmpgtb(mem64, mm*i*); pcmpgtb(mm*i*, mm*i*);	Packed Compare for Greater Than, Bytes. This instruction compares the individual bytes in the two operands. If the destination (second) operand byte is greater than the source (first) operand byte, then this instruction sets the corresponding byte in the destination (second) register to $FF (all ones); if they are not equal, this instruction sets the corresponding byte to zero. Note that there is no PCMPLEB instruction. You can simulate this instruction by swapping the operands in the PCMPGTB instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.
pcmpgtd(mem64, mm*i*); pcmpgtd(mm*i*, mm*i*);	Packed Compare for Greater Than, Double Words. This instruction compares the individual dwords in the two operands. If the destination (second) operand dword is greater than the source (first) operand dword, then this instruction sets the corresponding dword in the destination (second) register to $FFFF_FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero. Note that there is no PCMPLED instruction. You can simulate this instruction by swapping the operands in the PCMPGTD instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.
pcmpgtw(mem64, mm*i*); pcmpgtw(mm*i*, mm*i*);	Packed Compare for Greater Than,Words. This instruction compares the individual words in the two operands. If the destination (second) operand word is greater than the source (first) operand word, then this instruction sets the corresponding word in the destination (second) register to $FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero. Note that there is no PCMPLEW instruction. You can simulate this instruction by swapping the operands in the PCMPGTW instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.

Table 3: MMX Instruction Set

Instruction	Description
pextrw(imm8, mm*i*, reg32);	Packed Extraction of a word. The imm8 value must be a constant in the range 0..3. This instruction copies the specified word from the MMX register into the L.O. word of the destination 32-bit integer register. This instruction zero extends the 16-bit value to 32 bits in the integer register. Note that there are no extraction instructions for bytes or dwords. However, you can easily extract a byte using PEXTRW and an AND or XCHG instruction (depending on whether the byte number is even or odd). You can use MOVD to extract the L.O. dword. to extract the H.O. dword of an MMX register requires a bit more work; either extract the two words and merge them or move the data to memory and grab the dword you're interested in.
pinsw(imm8, reg32, mm*i*);	Packed Insertion of a word. The imm8 value must be a constant in the range 0..3. This instruction copies the L.O. word from the 32-bit integer register into the specified word of the destination MMX register. This instruction ignores the H.O. word of the integer register.
pmaddwd(mem64, mm*i*); pmaddwd(mm*i*, mm*i*);	Packed Multiple and Accumulate (Add). This instruction multiplies together the corresponding words in the source and destination operands. Then it adds the two double word products from the multiplication of the two L.O. words and stores this double word sum in the L.O. dword of the destination MMX register. Finally, it adds the two double word products from the multiplication of the H.O. words and stores this double word sum in the H.O. dword of the destination MMX register.
pmaxw(mem64, mm*i*); pmaxw(mm*i*, mm*i*);	Packed Signed Integer Word Maximum. This instruction compares the four words between the two operands and stores the signed maximum of each corresponding word in the destination MMX register.
pmaxub(mem64, mm*i*); pmaxub(mm*i*, mm*i*);	Packed Unsigned Byte Maximum. This instruction compares the eight bytes between the two operands and stores the unsigned maximum of each corresponding byte in the destination MMX register.
pminw(mem64, mm*i*); pminw(mm*i*, mm*i*);	Packed Signed Integer Word Minimum. This instruction compares the four words between the two operands and stores the signed minimum of each corresponding word in the destination MMX register.
pminub(mem64, mm*i*); pminub(mm*i*, mm*i*);	Packed Unsigned Byte Minimum. This instruction compares the eight bytes between the two operands and stores the unsigned minimum of each corresponding byte in the destination MMX register.
pmovmskb(mm*i*, reg32);	Move Byte Mask to Integer. This instruction creates a byte by extracting the H.O. bit of the eight bytes from the MMX source register. It zero extends this value to 32 bits and stores the result in the 32-bit integer register.
pmulhuw(mem64, mm*i*); pmulhuw(mm*i*, mm*i*);	Packed Multiply High, Unsigned Words. This instruction multiplies the four unsigned words of the two operands together and stores the H.O. word of the resulting products into the corresponding word of the destination MMX register.

Table 3: MMX Instruction Set

Instruction	Description
pmulhw(mem64, mm*i*); pmulhw(mm*i*, mm*i*);	Packed Multiply High, Signed Words. This instruction multiplies the four signed words of the two operands together and stores the H.O. word of the resulting products into the corresponding word of the destination MMX register.
pmullw(mem64, mm*i*); pmullw(mm*i*, mm*i*);	Packed Multiply Low, Signed Words. This instruction multiplies the four signed words of the two operands together and stores the L.O. word of the resulting products into the corresponding word of the destination MMX register.
por(mem64, mm*i*); por(mm*i*, mm*i*);	Packed OR. Computes the bitwise OR of the two operands and stores the result in the destination (second) MMX register.
psadbw(mem64, mm*i*); psadbw(mm*i*, mm*i*);	Packed Sum of Absolute Differences. This instruction computes the absolute value of the difference of each of the unsigned bytes between the two operands. Then it adds these eight results together to form a word sum. Finally, the instruction zero extends this word to 64 bits and stores the result in the destination (second) operand.
pshufw(imm8,mem64,mm*i*);	Packed Shuffle Word. This instruction treats the imm8 value as an array of four two-bit values. These bits specify where the destination (third) operand's words obtain their values. Bits zero and one tell this instruction where to obtain the L.O. word, bits two and three specify where word #1 comes from, bits four and five specify the source for word #2, and bits six and seven specify the source of the H.O. word in the destination operand. Each pair of bytes specifies a word number in the source (second) operand. For example, an immediate value of %00011011 tells this instruction to grab word #3 from the source and place it in the L.O. word of the destination; grab word #2 from the source and place it in word #1 of the destination; grab word #1 from the source and place it in word #2 of the destination; and grab the L.O. word of the source and place it in the H.O. word of the destination (i.e., swap all the words in a manner similar to the BSWAP instruction).

pslld(mem, mm*i*);
pslld(mm*i*, mm*i*);
pslld(imm8, mm*i*);
	Packed Shift Left Logical, Double Words. This instruction shifts the destination (second) operand to the left the number of bits specified by the first operand. Each double word in the destination is treated as an independent entity. Bits are not carried over from the L.O. dword to the H.O. dword. Bits shifted out are lost and this instruction always shifts in zeros.

psllq(mem, mm*i*);
psllq(mm*i*, mm*i*);
psllq(imm8, mm*i*);
	Packed Shift Left Logical, Quad Word. This instruction shifts the destination operand to the left the number of bits specified by the first operand.

psllw(mem, mm*i*);
psllw(mm*i*, mm*i*);
psllw(imm8, mm*i*);
	Packed Shift Left Logical,Words. This instruction shifts the destination (second) operand to the left the number of bits specified by the first operand. Bits shifted out are lost and this instruction always shifts in zeros. Each word in the destination is treated as an independent entity. Bits are not carried over from the L.O. words into the next higher word. Bits shifted out are lost and this instruction always shifts in zeros.

Table 3: MMX Instruction Set

Instruction	Description

psard(mem, mm*i*);
psard(mm*i*, mm*i*);
psard(imm8, mm*i*);
	Packed Shift Right Arithmetic, Double Word. This instruction treats the two halves of the 64-bit register as two double words and performs separate arithmetic shift rights on them. The bit shifted out of the bottom of the two double words is lost.

psarw(mem, mm*i*);
psarw(mm*i*, mm*i*);
psarw(imm8, mm*i*);
	Packed Shift Right Arithmetic, Word. This instruction operates independently on the four words of the 64-bit destination register and performs separate arithmetic shift rights on them. The bit shifted out of the bottom of the four words is lost.

psrld(mem, mm*i*);
psrld(mm*i*, mm*i*);
psrld(imm8, mm*i*);
	Packed Shift Right Logical, Double Words. This instruction shifts the destination (second) operand to the right the number of bits specified by the first operand. Each double word in the destination is treated as an independent entity. Bits are not carried over from the H.O. dword to the L.O. dword. Bits shifted out are lost and this instruction always shifts in zeros.

pslrq(mem, mm*i*);
pslrq(mm*i*, mm*i*);
pslrq(imm8, mm*i*);
	Packed Shift Right Logical, Quad Word. This instruction shifts the destination operand to the right the number of bits specified by the first operand.

pslrw(mem, mm*i*);
pslrw(mm*i*, mm*i*);
pslrw(imm8, mm*i*);
	Packed Shift Right Logical,Words. This instruction shifts the destination (second) operand to the right the number of bits specified by the first operand. Bits shifted out are lost and this instruction always shifts in zeros. Each word in the destination is treated as an independent entity. Bits are not carried over from the H.O. words into the next lower word. Bits shifted out are lost and this instruction always shifts in zeros.

psubb(mem64, mm*i*); psubb(mm*i*, mm*i*);	Packed Subtract of Bytes. This instruction subtracts the individual bytes of the source (first) operand from the corresponding bytes of the destination (second) operand. The subtraction of each byte is independent of the other eight bytes; there is no borrow from byte to byte. If an overflow or underflow occurs in any byte, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
psubd(mem64, mm*i*); psubd(mm*i*, mm*i*);	Packed Subtract of Double Words. This instruction subtracts the individual dwords of the source (first) operand from the corresponding dwords of the destination (second) operand. The subtraction of each dword is independent of the other; there is no borrow from dword to dword. If an overflow or underflow occurs in any dword, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
psubsb(mem64, mm*i*); psubsb(mm*i*, mm*i*);	Packed Subtract of Bytes, signed saturated. This instruction subracts the individual bytes of the source operand from the corresponding bytes of the destination operand, saturating to -128 or +127 if overflow or underflow occurs. The subtraction of each byte is independent of the other seven bytes; there is no carry from byte to byte. This instruction does not affect any flags.

Table 3: MMX Instruction Set

Instruction	Description
psubsw(mem64, mm*i*); psubsw(mm*i*, mm*i*);	Packed Subtract of Words, signed saturated. This instruction subracts the individual words of the source operand from the corresponding words of the destination operand, saturating to -32768 or +32767if overflow or underflow occurs. The subtraction of each word is independent of the other three words; there is no carry from word to word. This instruction does not affect any flags.
psubusb(mem64, mm*i*); psubusb(mm*i*, mm*i*);	Packed Subtract of Bytes, unsigned saturated. This instruction subracts the individual bytes of the source operand from the corresponding bytes of the destination operand, saturating to 0 if underflow occurs. The subtraction of each byte is independent of the other seven bytes; there is no carry from byte to byte. This instruction does not affect any flags.
psubusw(mem64, mm*i*); psubusw(mm*i*, mm*i*);	Packed Subtract of Words, unsigned saturated. This instruction subracts the individual words of the source operand from the corresponding words of the destination operand, saturating to 0 if underflow occurs. The subtraction of each word is independent of the other three words; there is no carry from word to word. This instruction does not affect any flags.
psubw(mem64, mm*i*); psubw(mm*i*, mm*i*);	Packed Subtract of Words. This instruction subtracts the individual words of the source (first) operand from the corresponding words of the destination (second) operand. The subtraction of each word is independent of the others; there is no borrow from word to word. If an overflow or underflow occurs in any word, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
punpckhbw(mem64, mm*i*); punpckhbw(mm*i*, mm*i*);	Unpack high packed data, bytes to words. This instruction unpacks and interleaves the high-order four bytes of the source (first) and destination (second) operands. It places the H.O. four bytes of the destination operand at the even byte positions in the destination and it places the H.O. four bytes of the source operand in the odd byte positions of the destination operand.
punpckhdq(mem64, mm*i*); punpckhdq(mm*i*, mm*i*);	Unpack high packed data, dwords to qword. This instruction copies the H.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) H.O. dword of the destination operand to the L.O. dword of the destination.
punpckhwd(mem64, mm*i*); punpckhwd(mm*i*, mm*i*);	Unpack high packed data, words to dwords. This instruction unpacks and interleaves the high-order two words of the source (first) and destination (second) operands. It places the H.O. two words of the destination operand at the even word positions in the destination and it places the H.O. words of the source operand in the odd word positions of the destination operand.
punpcklbw(mem64, mm*i*); punpcklbw(mm*i*, mm*i*);	Unpack low packed data, bytes to words. This instruction unpacks and interleaves the low-order four bytes of the source (first) and destination (second) operands. It places the L.O. four bytes of the destination operand at the even byte positions in the destination and it places the L.O. four bytes of the source operand in the odd byte positions of the destination operand.

Table 3: MMX Instruction Set

Instruction	Description
punpckldq(mem64, mm*i*); punpckldq(mm*i*, mm*i*);	Unpack low packed data, dwords to qword. This instruction copies the L.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) L.O. dword of the destination operand to the L.O. dword of the destination (i.e., it doesn't change the L.O. dword of the destination).
punpcklwd(mem64, mm*i*); punpcklwd(mm*i*, mm*i*);	Unpack low packed data, words to dwords. This instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand.
pxor(mem64, mm*i*); pxor(mm*i*, mm*i*);	Packed Exclusive-OR. This instruction exclusive-ORs the source operand with the destination operand leaving the result in the destination operand.

