

GAS IA-32 Assembly Language Programming

Tony Richardson

April 12, 2006

The following table contains a partial list of IA-32 assembly mnemonics. An *S* in the mnemonic represents a size specification letter of either b (8 bits), w (16 bits) or l (32 bits). If one of the operands is a register, then the register name implies an operand size and the specification letter may be omitted. For example, either “movl %eax, count” or “mov %eax, count” may be used and the 32-bit contents of register EAX will be copied to the memory location labeled count. However, when using “mov*S* \$30, count” then *S* must be replaced with b, w, or l to indicate if the 30 is being copied to a byte, a word or an int location in memory.

In the operand columns, i represents an immediate (numeric) operand, r a register and m a memory location. If the operand is enclosed in brackets, then it is optional. A destination operand appears on the same line as a corresponding source operand. The adc/add entry below indicates that if the first operand is either immediate or a register then the second operand can be either a register or memory location. If the first operand is a memory location, then the second operand must be a register.

Memory references are of the form *disp(base, index, scale)* where *base* and *index* are optional 32-bit base and index registers, *disp* is an optional displacement, and *scale* may have the values 1 (default), 2, 4, or 8. The actual address is equal to *disp + base + index * scale*. Here are a few examples (assume that xarray labels the first element of an array of ints):

```
# Direct addressing
movl  xarray, %eax           # move first element of array into EAX

# Indexed addressing
movl  $0, %edi
movl  xarray(%edi,4), %eax   # move first element of array into EAX
incl  %edi
movl  xarray(%edi,4), %eax   # move second element of array into EAX
movl  $10, %edi
movl  xarray(%edi,4), %eax   # move eleventh element of array into EAX

# Indirect addressing
movl  $xarray, %ebx         # move base address into EBX
movl  (%ebx), %eax          # move first element of array into EAX
movl  $0, %edi
movl  (%ebx, %edi, 4), %eax  # move first element of array into EAX
incl  %edi
movl  (%ebx, %edi, 4), %eax  # move second element of array into EAX
movl  $10, %edi
movl  (%ebx, %edi, 4), %eax  # move second element of array into EAX

# The displacement may be negative.
# This is also an example of indirect addressing.
movl  -4(%ebp), %eax        # move data from stack frame into EAX
```

This is only a partial list of mnemonics. Floating-point processor, undocumented, MMX, SSE, and SSE2 instructions have been omitted. 16-bit mode and privileged mode specific instructions have also been left out. The table was extracted from the NASM documentation and converted to AT&T syntax. I recommend the NASM documentation (as well as the Intel documentation) as a complete source of documentation for IA-32 assembly language programming.

Mnemonic	Op 1 (SRC)	Op 2 (DEST)	Description
aaa aas			<p>ASCII Adjustments</p> <p>AAA (ASCII Adjust After Addition) should be used after a one-byte ADD instruction whose destination was the AL register: by means of examining the value in the low nibble of AL and also the auxiliary carry flag AF, it determines whether the addition has overflowed, and adjusts it (and sets the carry flag) if so. You can add long BCD strings together by doing ADD/AAA on the low digits, then doing ADC/AAA on each subsequent digit.</p> <p>AAS (ASCII Adjust AL After Subtraction) works similarly to AAA, but is for use after SUB instructions rather than ADD.</p>
aam aad	[i]		<p>ASCII Adjustments</p> <p>AAM (ASCII Adjust AX After Multiply) is for use after you have multiplied two decimal digits together and left the result in AL: it divides AL by ten and stores the quotient in AH, leaving the remainder in AL. The divisor 10 can be changed by specifying an operand to the instruction: a particularly handy use of this is AAM 16, causing the two nibbles in AL to be separated into AH and AL.</p> <p>AAD (ASCII Adjust AX Before Division) performs the inverse operation to AAM: it multiplies AH by ten, adds it to AL, and sets AH to zero. The multiplier 10 can be changed.</p>
adcS addS	i/r m	r/m r	<p>Add with Carry, Add</p> <p>ADC performs integer addition: it adds its two operands together, plus the value of the carry flag, and leaves the result in its destination operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.</p> <p>ADD performs integer addition: it adds its two operands together, and leaves the result in its destination operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.</p> <p>The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent ADC instruction.</p>
andS	i/r m	r/m r	<p>Bitwise AND</p> <p>AND performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.</p>
boundS	r	m	<p>Check Array Index Against Bounds</p> <p>BOUND expects its second operand to point to an area of memory containing two signed values of the same size as its first operand (i.e. two words for the 16-bit form; two doublewords for the 32-bit form). It performs two signed comparisons: if the value in the register passed as its first operand is less than the first of the in-memory values, or is greater than or equal to the second, it throws a BR exception. Otherwise, it does nothing.</p>
bsfS bsrS	r/m	r	<p>Bit Scan</p> <p>BSF searches for the least significant set bit in its source operand, and if it finds one, stores the index in its destination operand. If no set bit is found, the contents of the destination operand are undefined. If the source operand is zero, the zero flag is set.</p>

			<p>BSR performs the same function as BSF, but searches from the top instead, so it finds the most significant set bit.</p> <p>Bit indices are from 0 (least significant) to 15 or 31 (most significant). The destination operand can only be a register. The source operand can be a register or a memory location.</p>
bswapl	r		<p>Byte Swap BSWAP swaps the order of the four bytes of a 32-bit register: bits 0–7 exchange places with bits 24–31, and bits 8–15 swap with bits 16–23. There is no explicit 16-bit equivalent: to byte-swap AX, BX, CX or DX, XCHG can be used.</p>
btS btcS btrS btsS	i r	r/m r/m	<p>Bit Test These instructions all test one bit of their second operand, whose index is given by the first operand, and store the value of that bit into the carry flag. Bit indices are from 0 (least significant) to 15 or 31 (most significant).</p> <p>In addition to storing the original value of the bit into the carry flag, BTR also resets (clears) the bit in the operand itself. BTS sets the bit, and BTC complements the bit. BT does not modify its operands.</p> <p>The destination can be a register or a memory location. The source can be a register or an immediate value.</p> <p>If the destination operand is a register, the bit offset should be in the range 0–15 (for 16-bit operands) or 0–31 (for 32-bit operands). An immediate value outside these ranges will be taken modulo 16/32 by the processor.</p> <p>If the destination operand is a memory location, then an immediate bit offset follows the same rules as for a register. If the bit offset is in a register, then it can be anything within the signed range of the register used (ie, for a 32-bit operand, it can be (-2^{31}) to $(2^{31} - 1)$)</p>
call	i		<p>Call Subroutine CALL calls a subroutine, by means of pushing the current instruction pointer (IP) on the stack, and then jumping to a given address.</p>
cbw/cbtw cwde/cwtl cwd/cwtd cdq/cltd			<p>Sign Extensions All these instructions sign-extend a short value into a longer one, by replicating the top bit of the original value to fill the extended one.</p> <p>CBW extends AL into AX by repeating the top bit of AL in every bit of AH. CWDE extends AX into EAX. CWD extends AX into DX:AX by repeating the top bit of AX throughout DX, and CDQ extends EAX into EDX:EAX.</p> <p>CBTW is a synonym for CBW. Similarly CWTL, CWTD, CLTD are synonyms for CWDE, CWD, and CDQ respectively.</p>
clc cld cli			<p>Clear Flags These instructions clear various flags. CLC clears the carry flag; CLD clears the direction flag; and CLI clears the interrupt flag (thus disabling interrupts).</p> <p>To set the carry, direction, or interrupt flags, use the STC, STD and STI instructions. To invert the carry flag, use CMC.</p>
cmc			<p>Complement Carry Flag CMC changes the value of the carry flag: if it was 0, it sets it to 1, and vice versa.</p>
cmovCC	r/m	r	<p>Conditional Move</p>

			<p>CMOV moves its source (first) operand into its destination (second) operand if the given condition code is satisfied; otherwise it does nothing.</p> <p>For a list of condition codes see the <i>jCC</i> (conditional branch) instruction.</p> <p>These instructions are only available in the P6 family of Pentium processors (Pentium Pro, Pentium II, and newer).</p>
cmpS	i/r r/m	r/m r	<p>Compare Integers CMP performs a ‘mental’ subtraction of its first operand from its second operand, and affects the flags as if the subtraction had taken place, but does not store the result of the subtraction anywhere</p>
cmpsS			<p>Compare Strings CMPSB compares the byte at %DS:(%ESI) with the byte at %ES:(%EDI), and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) ESI and EDI.</p> <p>The segment register used to load from (%ESI) can be overridden by using a segment register name as a prefix (for example, ES CMPSB). The use of ES for the load from (%EDI) cannot be overridden.</p> <p>CMPSW and CMPSL work in the same way, but they compare a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.</p> <p>The REPE and REPNE prefixes (equivalently, REPZ and REPNZ) may be used to repeat the instruction up to %ECX times until the first unequal or equal byte is found.</p>
cmpxchgS	r	r/m	<p>Compare and Exchange CMPXCHG compares its destination (second) operand to the value in AL, AX or EAX (depending on the operand size of the instruction). If they are equal, it copies its source (first) operand into the destination and sets the zero flag. Otherwise, it clears the zero flag and copies the destination register to AL, AX or EAX.</p> <p>CMPXCHG is intended to be used for atomic operations in multitasking or multiprocessor environments. To safely update a value in shared memory, for example, you might load the value into EAX, load the updated value into EBX, and then execute the instruction LOCK CMPXCHG %EBX, VALUE. If value has not changed since being loaded, it is updated with your desired new value, and the zero flag is set to let you know it has worked. (The LOCK prefix prevents another processor doing anything in the middle of this operation: it guarantees atomicity.) However, if another processor has modified the value in between your load and your attempted store, the store does not happen, and you are notified of the failure by a cleared zero flag, so you can go round and try again.</p>
cmpxchg8b	m		<p>Compare and Exchange Eight Bytes This is a larger and more unwieldy version of CMPXCHG: it compares the 64-bit (eight-byte) value stored at m with the value in EDX:EAX. If they are equal, it sets the zero flag and stores ECX:EBX into the memory area. If they are unequal, it clears the zero flag and stores the memory contents into EDX:EAX.</p> <p>CMPXCHG8B can be used with the LOCK prefix, to allow atomic execution. This is useful in multi-processor and multi-tasking environments.</p>
cpuid			<p>Get CPU Identification Code CPUID returns various information about the processor it is being executed on. It fills the four registers EAX, EBX, ECX and EDX with information, which varies depending</p>

			<p>on the input contents of EAX.</p> <p>CPUID also acts as a barrier to serialise instruction execution: executing the CPUID instruction guarantees that all the effects (memory modification, flag modification, register modification) of previous instructions have been completed before the next instruction gets fetched.</p> <p>The information returned is as follows:</p> <ul style="list-style-type: none"> • If EAX is zero on input, EAX on output holds the maximum acceptable input value of EAX, and EBX:EDX:ECX contain the string "GenuineIntel" (or not, if you have a clone processor). That is to say, EBX contains "Genu", EDX contains "ineI" and ECX contains "ntel". • If EAX is one on input, EAX on output contains version information about the processor, and EDX contains a set of feature flags, showing the presence and absence of various features. For example, bit 8 is set if the CMPXCHG8B instruction is supported, bit 15 is set if the conditional move instructions are supported, and bit 23 is set if MMX instructions are supported. • If EAX is two on input, EAX, EBX, ECX and EDX all contain information about caches and TLBs (Translation Lookahead Buffers).
daa das			<p>Decimal Adjustments</p> <p>These instructions are used in conjunction with the add and subtract instructions to perform binary-coded decimal arithmetic in packed (one BCD digit per nibble) form.</p> <p>DAA should be used after a one-byte ADD instruction whose destination was the AL register: by means of examining the value in the AL and also the auxiliary carry flag AF, it determines whether either digit of the addition has overflowed, and adjusts it (and sets the carry and auxiliary-carry flags) if so. You can add long BCD strings together by doing ADD/DAA on the low two digits, then doing ADC/DAA on each subsequent pair of digits.</p> <p>DAS works similarly to DAA, but is for use after SUB instructions rather than ADD.</p>
decS	r/m		<p>Decrement Integer</p> <p>DEC subtracts 1 from its operand. It does not affect the carry flag: to affect the carry flag, use SUB 1, something. DEC affects all the other flags according to the result.</p> <p>This instruction can be used with a LOCK prefix to allow atomic execution.</p>
divS	r/m		<p>Unsigned Integer Divide</p> <p>DIV performs unsigned integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:</p> <ul style="list-style-type: none"> • For DIVB, AX is divided by the given operand; the quotient is stored in AL and the remainder in AH. • For DIVW, DX:AX is divided by the given operand; the quotient is stored in AX and the remainder in DX. • For DIVL, EDX:EAX is divided by the given operand; the quotient is stored in EAX and the remainder in EDX. <p>Signed integer division is performed by the IDIV instruction</p>
enter	i	i	<p>Create Stack Frame</p> <p>ENTER construct a stack frame for a high-level language procedure call. The first operand gives the amount of stack space to allocate for local variables; the second gives the nesting level of the procedure (for languages like Pascal, with nested procedures).</p> <p>The function of ENTER, with a nesting level of zero, is equivalent to</p>

			<p>PUSHL %EBP MOVL %ESP, %EBP SUBL operand1, %ESP</p> <p>This creates a stack frame with the procedure parameters accessible upwards from EBP, and local variables accessible downwards from EBP.</p> <p>With a nesting level of one, the stack frame created is 4 (or 2) bytes bigger, and the value of the final frame pointer EBP is accessible in memory at $-4(\%EBP)$.</p> <p>This allows ENTER, when called with a nesting level of two, to look at the stack frame described by the previous value of EBP, find the frame pointer at offset -4 from that, and push it along with its new frame pointer, so that when a level-two procedure is called from within a level-one procedure, $-4(\%EBP)$ holds the frame pointer of the most recent level-one procedure call and $-8(\%EBP)$ holds that of the most recent level-two call. And so on, for nesting levels up to 31.</p> <p>Stack frames created by ENTER can be destroyed by the LEAVE instruction</p>
idivS	r/m		<p>Signed Integer Divide IDIV performs signed integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:</p> <ul style="list-style-type: none"> • For IDIVB, AX is divided by the given operand; the quotient is stored in AL and the remainder in AH. • For IDIVW, DX:AX is divided by the given operand; the quotient is stored in AX and the remainder in DX. • For IDIVL, EDX:EAX is divided by the given operand; the quotient is stored in EAX and the remainder in EDX. <p>Unsigned integer division is performed by the DIV instruction.</p>
imulS	r/m i/r/m i, r/m	r r	<p>Signed Integer Multiply IMUL performs signed integer multiplication. For the single-operand form, the other operand and destination are implicit, in the following way:</p> <ul style="list-style-type: none"> • For IMULB, AL is multiplied by the given operand; the product is stored in AX. • For IMULW, AX is multiplied by the given operand; the product is stored in DX:AX. • For IMULL, EAX is multiplied by the given operand; the product is stored in EDX:EAX. <p>The two-operand form multiplies its two operands and stores the result in the destination (second) operand. The three-operand form multiplies its first two operands and stores the result in the last operand.</p> <p>In the forms with an 8-bit immediate operand and another longer source operand, the immediate operand is considered to be signed, and is sign-extended to the length of the other source operand.</p> <p>Unsigned integer multiplication is performed by the MUL instruction.</p>
inS	i/DX i/DX i/DX	AL AX EAX	<p>Input From I/O Port IN reads a byte, word or doubleword from the specified I/O port, and stores it in the given destination register. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in DX. See also OUT.</p>

incS	r/m	<p>Increment Integer INC adds 1 to its operand. It does not affect the carry flag: to affect the carry flag, use ADD \$1,something. INC affects all the other flags according to the result.</p> <p>This instruction can be used with a LOCK prefix to allow atomic execution.</p> <p>See also DEC.</p>
insS		<p>Input String From I/O Port INSB inputs a byte from the I/O port specified in DX and stores it at %ES:(%EDI). It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) EDI.</p> <p>Segment override prefixes have no effect for this instruction: the use of ES for the load from (%EDI) cannot be overridden.</p> <p>INSW and INSL work in the same way, but they input a word or a doubleword instead of a byte, and increment or decrement the addressing register by 2 or 4 instead of 1.</p> <p>The REP prefix may be used to repeat the instruction ECX times.</p> <p>See also OUTSB, OUTSW and OUTSL.</p>
int int3 into	i	<p>Software Interrupt INT causes a software interrupt through a specified vector number from 0 to 255.</p> <p>INT3 is not precisely equivalent to INT 3: the short form, since it is designed to be used as a breakpoint, bypasses the normal IOPL checks in virtual-8086 mode, and also does not go through interrupt redirection.</p> <p>INTO performs an INT 4 software interrupt if and only if the overflow flag is set.</p>
invd		<p>Invalidate Internal Caches INVD invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It does not write the contents of the caches back to memory first: any modified data held in the caches will be lost. To write the data back first, use WBINVD.</p>
invlpg	m	<p>Invalidate TLB Entry INVLPG invalidates the translation lookahead buffer (TLB) entry associated with the supplied memory address.</p>
iret		<p>Return From Interrupt IRET returns from an interrupt (hardware or software) by means of popping EIP, CS and the flags off the stack and then continuing execution from the new CS:EIP.</p> <p>IRET pops EIP as 4 bytes, pops a further 4 bytes of which the top two are discarded and the bottom two go into CS, and pops the flags as 4 bytes as well, taking 12 bytes off the stack.</p>
jCC	i	<p>Conditional Branch The conditional jump instructions execute a near (same segment) jump if and only if their conditions are satisfied. For example, JNZ jumps only if the zero flag is not set.</p> <p>There are 6 status flags that are associated with the conditional jump instructions:</p> <ul style="list-style-type: none"> • CF – Carry flag. Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an

			<p>overflow condition for unsigned–integer arithmetic. It is also used in multiple–precision arithmetic.</p> <ul style="list-style-type: none"> • PF – Parity flag. Set if the least–significant byte of the result contains an even number of 1 bits; cleared otherwise. • AF – Adjust flag. Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary–coded decimal (BCD) arithmetic. • ZF – Zero flag. Set if the result is zero; cleared otherwise. • SF – Sign flag. Set equal to the most–significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.) • OF – Overflow flag. Set if the integer result is too large a positive number or too small a negative number (excluding the sign–bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed–integer (two’s complement) arithmetic. <p>The condition codes to be used instead of <i>CC</i> are listed below. Many of these condition codes have synonyms, so several will be listed at a time.</p> <ul style="list-style-type: none"> • O indicates the overflow flag is set; NO is the complement. • B, C and NAE indicate that the carry flag is set; NB, NC and AE are complements. • E and Z indicate the zero flag is set; NE and NZ are complements. • BE and NA indicate that the carry or zero flags is set; NBE and A are complements. • S indicates the sign flag is set; NS is complement. • P and PE indicate that the parity flag is set; NP and PO are complements. • L and NGE indicate that exactly one of the sign and overflow flags is set; NL and GE are complements. • LE and NG indicate that the zero flag is set, or exactly one of the sign and overflow flags is set; NLE and G are complements.
jcxz jecxz	i		<p>Jump if CX/ECX is Zero JCXZ performs a short jump (with maximum range 128 bytes) if and only if the contents of the CX register is 0. JECXZ does the same thing, but with ECX.</p>
jmp	i		<p>Jump JMP jumps to a given address.</p>
lahf			<p>Load AH From Flags LAHF sets the AH register according to the contents of the low byte of the flags word.</p> <p>The operation of LAHF is:</p> $AH \leftarrow SF:ZF:0:AF:0:PF:1:CF$ <p>See also SAHF.</p>
lea	m	r	<p>Load Effective Address LEA, despite its syntax, does not access memory. It calculates the effective address specified by its first operand as if it were going to load or store data from it, but instead it stores the calculated address into the register specified by its second operand. This can be used to perform quite complex calculations (e.g. LEA 100(%EBX,%ECX,4),</p>

			<p>%EAX) in one instruction.</p> <p>LEA, despite being a purely arithmetic instruction which accesses no memory, still requires that the first operand be written as if it were a memory reference.</p>
leave			<p>Destroy Stack Frame LEAVE destroys a stack frame of the form created by the ENTER instruction. It is functionally equivalent to MOVL %EBP, %ESP followed by POPL %EBP.</p>
lodsS			<p>Load From String LODSB loads a byte from %DS:(%ESI) into AL. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) ESI.</p> <p>The segment register used to load from (%ESI) can be overridden by using a segment register name as a prefix (for example, ES LODSB).</p> <p>LODSW and LODSL work in the same way, but they load a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.</p>
loop loope loopz	i	[CX/ECX]	<p>LOOP decrements its counter register (either CX or ECX – if one is not specified explicitly, the BITS setting dictates which is used) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label. The jump has a range of 128 bytes.</p> <p>LOOPE (or its synonym LOOPZ) adds the additional condition that it only jumps if the counter is nonzero and the zero flag is set. Similarly, LOOPNE (and LOOPNZ) jumps only if the counter is nonzero and the zero flag is clear.</p>
movS	i/r r/m	r/m r	<p>Move Data MOV copies the contents of its source (second) operand into its destination (first) operand.</p>
movsS			<p>Move String MOVSB copies the byte at (%ESI) to (%EDI). It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) ESI and EDI.</p> <p>MOVSW and MOVSD work in the same way, but they copy a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.</p> <p>The REP prefix may be used to repeat the instruction ECX times.</p>
movsSS movzSS	r/m	r	<p>Move Data with Sign or Zero Extend MOVSB sign–extends its source operand to the length of its destination operand, and copies the result into the destination operand. MOVZ does the same, but zero–extends rather than sign–extending.</p> <p>Because the source (first) and destination (second) operands are of different sizes two size specification letters are used in the mnemonic. The first S indicates the size of the first operand. Possible suffixes are bl (from byte to long), bw (from byte to word), wl (from word to long), bq (from byte to quadruple word), wq (from word to quadruple word), and lq (from long to quadruple word).</p>
mulS	r/m		<p>Unsigned Integer Multiply MUL performs unsigned integer multiplication. The other operand to the multiplication, and the destination operand, are implicit, in the following way:</p>

			<ul style="list-style-type: none"> • For MULB AL is multiplied by the given operand; the product is stored in AX. • For MULW, AX is multiplied by the given operand; the product is stored in DX:AX. • For MULL, EAX is multiplied by the given operand; the product is stored in EDX:EAX. <p>Signed integer multiplication is performed by the IMUL instruction</p>
negS notS	r/m		<p>Two's and One's Complement NEG replaces the contents of its operand by the two's complement negation (invert all the bits and then add one) of the original value. NOT, similarly, performs one's complement (inverts all the bits).</p>
nop			<p>No Operation NOP performs no operation.</p>
orS	i/r r/m	r/m r	<p>Bitwise OR OR performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination operand.</p>
out	i/DX i/DX i/DX	AL AX EAX	<p>Output Data to I/O Port OUT writes the contents of the given source register to the specified I/O port. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in DX. See also IN.</p>
outsS			<p>Output String to I/O Port OUTSB loads a byte from (%ESI) and writes it to the I/O port specified in DX. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) ESI.</p> <p>OUTSW and OUTSD work in the same way, but they output a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.</p> <p>The REP prefix may be used to repeat the instruction ECX times.</p>
popS	r/m		<p>Pop Data From the Stack POP loads a value from the stack (from (%ESP)) and then increments the stack pointer.</p> <p>The operand-size attribute of the instruction determines whether the stack pointer is incremented by 2 or 4.</p>
popa popaw popal			<p>Pop All General Purpose Registers</p> <ul style="list-style-type: none"> • POPAW pops a word from the stack into each of, successively, DI, SI, BP, nothing (it discards a word from the stack which was a placeholder for SP), BX, DX, CX and AX. It is intended to reverse the operation of PUSHAW, but it ignores the value for SP that was pushed on the stack by PUSHAW. • POPAL pops twice as much data, and places the results in EDI, ESI, EBP, nothing (placeholder for ESP), EBX, EDX, ECX and EAX. It reverses the operation of PUSHAL. <p>POPA is an alias mnemonic for either POPAW or POPAL, depending on the current BITS setting.</p>
popf popfw			<p>Pop Flags Register</p> <ul style="list-style-type: none"> • POPFW pops a word from the stack and stores it in the bottom 16 bits of the

popfl			<ul style="list-style-type: none"> flags register (or the whole flags register, on processors below a 386). POPFL pops a doubleword and stores it in the entire flags register. <p>POPF is an alias mnemonic for either POPFW or POPFL, depending on the current BITS setting. See also PUSHF</p>
pushS	i/r/m		<p>Push Data on Stack PUSH decrements the stack pointer, ESP by 2 or 4, and then stores the given value at (%ESP).</p> <p>The operand–size attribute of the instruction determines whether the stack pointer is decremented by 2 or 4.</p>
pusha pushaw pushal			<p>Push All General-Purpose Registers PUSHAW pushes, in succession, AX, CX, DX, BX, SP, BP, SI and DI on the stack, decrementing the stack pointer by a total of 16.</p> <p>PUSHAL pushes, in succession, EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI on the stack, decrementing the stack pointer by a total of 32.</p> <p>In both cases, the value of SP or ESP pushed is its original value, as it had before the instruction was executed.</p> <p>PUSHA is an alias mnemonic for either PUSHAW or PUSHAL, depending on the current BITS setting.</p>
pushf pushfw pushfl			<p>Push Flags Register</p> <ul style="list-style-type: none"> PUSHFW pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386). PUSHFD pops a doubleword and stores it in the entire flags register. <p>PUSHF is an alias mnemonic for either PUSHFW or PUSHFL, depending on the current BITS setting. See also POPF</p>
rclS rcrS	1/CL/i	r/m	<p>Bitwise Rotate Through Carry Bit RCL and RCR perform a 9–bit, 17–bit or 33–bit bitwise rotation operation, involving the given source/destination (second) operand and the carry bit. Thus, for example, in the operation RCLB \$1,%AL, a 9–bit rotation is performed in which AL is shifted left by 1, the top bit of AL moves into the carry flag, and the original value of the carry flag is placed in the low bit of AL.</p> <p>The number of bits to rotate by is given by the first operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.</p>
ret retn	[i]		<p>Return From Procedure Call RET, and its exact synonym RETN, pop EIP from the stack and transfer control to the new address. Optionally, if a numeric second operand is provided, they increment the stack pointer by a further i bytes after popping the return address.</p>
rolS rorS	1/CL/i	r/m	<p>Bitwise Rotate ROL and ROR perform a bitwise rotation operation on the given source/destination (second) operand. Thus, for example, in the operation ROLB 1, AL, an 8–bit rotation is performed in which AL is shifted left by 1 and the original top bit of AL moves round into the low bit.</p> <p>The number of bits to rotate by is given by the first operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.</p>
sahf			<p>Store AH to Flags</p>

			<p>SAHF sets the low byte of the flags word according to the contents of the AH register. The operation of SAHF is:</p> <p style="text-align: center;">AH → SF:ZF:0:AF:0:PF:1:CF</p> <p>See also LAHF</p>
salS sarS	1/CL/i	r/m	<p>Bitwise Arithmetic Shifts</p> <p>SAL and SAR perform an arithmetic shift operation on the given source/destination (second) operand. The vacated bits are filled with zero for SAL, and with copies of the original high bit of the source operand for SAR. SAL is a synonym for SHL.</p> <p>The number of bits to shift by is given by the first operand. Only the bottom five bits of the shift count are considered by processors above the 8086.</p>
sbbS	i/r r/m	r/m r	<p>Subtract With Borrow</p> <p>SBB performs integer subtraction: it subtracts its first operand, plus the value of the carry flag, from its second, and leaves the result in its destination (second) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent SBB instruction.</p> <p>To subtract one number from another without also subtracting the contents of the carry flag, use SUB.</p>
scasS			<p>Scan String</p> <p>SCASB compares the byte in AL with the byte at (%EDI), and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) EDI.</p> <p>SCASW and SCASD work in the same way, but they compare a word to AX or a doubleword to EAX instead of a byte to AL, and increment or decrement the addressing registers by 2 or 4 instead of 1.</p> <p>The REPE and REPNE prefixes (equivalently, REPZ and REPNZ) may be used to repeat the instruction up to ECX times until the first unequal or equal byte is found.</p>
setCC	r/m		<p>Set Register From Condition</p> <p>SETcc sets the given 8-bit operand to zero if its condition is not satisfied, and to 1 if it is.</p> <p>See Jcc for list of condition codes.</p>
shlS shrS	1/CL/i	r/m	<p>Bitwise Logical Shifts</p> <p>SHL and SHR perform a logical shift operation on the given source/destination (second) operand. The vacated bits are filled with zero. A synonym for SHL is SAL.</p> <p>The number of bits to shift by is given by the second operand. Only the bottom five bits of the shift count are considered by processors above the 8086.</p>
stc std sti			<p>Set Flags</p> <p>These instructions set various flags. STC sets the carry flag; STD sets the direction flag; and STI sets the interrupt flag (thus enabling interrupts).</p> <p>To clear the carry, direction, or interrupt flags, use the CLC, CLD and CLI instructions. To invert the carry flag, use CMC.</p>
stosS			<p>Store Byte to String</p> <p>STOSB stores the byte in AL at (%EDI), and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is</p>

			<p>clear, decrements if it is set) EDI.</p> <p>STOSW and STOSD work in the same way, but they store the word in AX or the doubleword in EAX instead of the byte in AL, and increment or decrement the addressing registers by 2 or 4 instead of 1.</p> <p>The REP prefix may be used to repeat the instruction ECX times.</p>
sub $\$$	i/r r/m	r/m r	<p>Subtract Integers SUB performs integer subtraction: it subtracts its first operand from its second, and leaves the result in its destination (second) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent SBB instruction.</p>
test $\$$	i/r	r/m	<p>Test Bits (notional bitwise AND) TEST performs a “mental” bitwise AND of its two operands, and affects the flags as if the operation had taken place, but does not store the result of the operation anywhere.</p>
wait fwait			<p>Wait for Floating-Point Processor WAIT, on 8086 systems with a separate 8087 FPU, waits for the FPU to have finished any operation it is engaged in before continuing main processor operations, so that (for example) an FPU store to main memory can be guaranteed to have completed before the CPU tries to read the result back out.</p> <p>On higher processors, WAIT is unnecessary for this purpose, and it has the alternative purpose of ensuring that any pending unmasked FPU exceptions have happened before execution continues.</p>
xadd $\$$	r	r/m	<p>Exchange and Add XADD exchanges the values in its two operands, and then adds them together and writes the result into the destination (second) operand. This instruction can be used with a LOCK prefix for multi-processor synchronisation purposes.</p>
xchg $\$$	r/m r	r r/m	<p>Exchange XCHG exchanges the values in its two operands. It can be used with a LOCK prefix for purposes of multi-processor synchronisation.</p>
xlat xlatb			<p>Translate Byte in Lookup Table XLATB adds the value in AL, treated as an unsigned byte, to EBX, and loads the byte from the resulting address back into AL.</p>
xor $\$$	i/r r/m	r/m r	<p>Bitwise Exclusive OR XOR performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (second) operand.</p>