## Data Movement Instructions

**mov** — Move

The mov instruction copies the data item referred to by its first operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its second operand (i.e. a register or memory).

*Syntax*

mov <reg>, <reg> | mov <reg>, <mem> | mov <mem>, <reg> | mov <con>, <reg> | mov <con>, <mem>

*Examples*

mov %ebx, %eax — copy the value in EBX into EAX

movb $5, var(,1) — store the value 5 into the byte at location var

**push** — Push on stack

The push instruction places its operand onto the top of the hardware supported stack in memory.

*Syntax*

push <reg32> | push <mem> | push <con32>

*Examples*

push %eax — push eax on the stack

push var(,1) — push the 4 bytes at address *var* onto the stack

**pop** — Pop from stack

The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location).

*Syntax*

pop <reg32> | pop <mem>

*Examples*

pop %edi — pop the top element of the stack into EDI.

pop (%ebx) — pop the top element of the stack into memory at the four bytes starting at location EBX.

**lea** — Load effective address

The lea instruction places the *address* specified by its first operand into the register specified by its second operand.

*Syntax*

lea <mem>, <reg32>

*Examples*

lea (%ebx,%esi,8), %edi — the quantity EBX+8*ESI is placed in EDI.

lea val(,1), %eax — the value *val* is placed in EAX.

## Arithmetic and Logic Instructions

**add** — Integer addition

The add instruction adds together its two operands, storing the result in its second operand.

*Syntax*

add <reg>, <reg> | add <mem>, <reg> | add <reg>, <mem> | add <con>, <reg> | add <con>, <mem>

*Examples*

add $10, %eax — EAX is set to EAX + 10

addb $10, (%eax) — add 10 to the single byte stored at memory address stored in EAX

**sub** — Integer subtraction

The sub instruction stores in the value of its second operand the result of subtracting the value of its first operand from the value of its second operand.

*Syntax*

sub <reg>, <reg> | sub <mem>, <reg> | sub <reg>, <mem> | sub <con>, <reg> | sub <con>, <mem>

*Examples*

sub %ah, %al — AL is set to AL - AH

sub $216, %eax — subtract 216 from the value stored in EAX

**inc, dec** — Increment, Decrement
The inc instruction increments the contents of its operand by one. The dec instruction decrements the contents of its operand by one.
*Syntax*
inc <reg> | inc <mem> | dec <reg> | dec <mem>
*Examples*
dec %eax — subtract one from the contents of EAX
incl var(,1) — add one to the 32-bit integer stored at location *var*
**imul** — Integer multiplication
The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).
The two-operand form multiplies its two operands together and stores the result in the second operand. *Syntax*
imul <reg32>, <reg32>  | imul <mem>, <reg32> | imul <con>, <reg32>, <reg32> | imul <con>, <mem>, <reg32>
*Examples*
imul (%ebx), %eax — multiply the contents of EAX by the 32-bit contents of the memory at location EBX. Store the result in EAX.
imul $25, %edi, %esi — ESI is set to EDI * 25
**idiv** — Integer division
The idiv instruction divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.
*Syntax*
idiv <reg32>
idiv <mem>
*Examples*
idiv %ebx — divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX and the remainder in EDX.
idivw (%ebx) — divide the contents of EDX:EAS by the 32-bit value stored at the memory location in EBX. Place the quotient in EAX and the remainder in EDX.
**and, or, xor** — Bitwise logical and, or, and exclusive or
These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.
*Syntax*
and <reg>, <reg>
and <mem>, <reg>
and <reg>, <mem>
and <con>, <reg>
and <con>, <mem>
or con stesse combinazioni di operandi delll'and
xor con stesse combinazione  di operandi  dell'and
*Examples*
and $0x0f, %eax — clear all but the last 4 bits of EAX.
xor %edx, %edx — set the contents of EDX to zero.
**not** — Bitwise logical not
Logically negates the operand contents (that is, flips all bit values in the operand).
*Syntax*
not <reg>
not <mem>
*Example*
not %eax — flip all the bits of EAX
**neg** — Negate

Performs the two's complement negation of the operand contents.
*Syntax*
neg <reg>
neg <mem>
*Example*
neg %eax — EAX is set to (- EAX)
**shl, shr** — Shift left and right
These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater then 31 are performed modulo 32.
*Syntax*
shl <con8>, <reg>
shl <con8>, <mem>
shl %cl, <reg>
shl %cl, <mem>
shr <con8>, <reg>
shr <con8>, <mem>
shr %cl, <reg>
shr %cl, <mem>
*Examples*
shl $1, eax — Multiply the value of EAX by 2 (if the most significant bit is 0)
shr %cl, %ebx — Store in EBX the floor of result of dividing the value of EBX by $2^n$ where *n* is the value in CL. Caution: for negative integers, it is *different* from the C semantics of division!

## Control Flow Instructions

The x86 processor maintains an instruction pointer (EIP) register that is a 32-bit value indicating the location in memory where the current instruction starts. Normally, it increments to point to the next instruction in memory begins after execution an instruction. The EIP register cannot be manipulated directly, but is updated implicitly by provided control flow instructions.
We use the notation <label> to refer to labeled locations in the program text. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example,

```
      mov 8(%ebp), %esi
begin:
       xor %ecx, %ecx
       mov (%esi), %eax
```

The second instruction in this code fragment is labeled begin. Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name begin. This label is just a convenient way of expressing the location instead of its 32-bit value.
**jmp** — Jump
Transfers program control flow to the instruction at the memory location indicated by the operand.
*Syntax*
jmp <label>
*Example*
jmp begin — Jump to the instruction labeled begin.
**j*condition*** — Conditional jump
These instructions are conditional jumps that are based on the status of a set of condition codes that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this

word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the jz instruction performs a jump to the specified operand label if the result of the last arithmetic operation was zero. Otherwise, control proceeds to the next instruction in sequence.
A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, cmp (see below). For example, conditional branches such as jle and jne are based on first performing a cmp operation on the desired operands.

*Syntax*

**je** <label> (jump when equal)
**jne** <label> (jump when not equal)
**jz** <label> (jump when last result was zero)
**jg** <label> (jump when greater than)
**jge** <label> (jump when greater than or equal to)
**jl** <label> (jump when less than)
**jle** <label> (jump when less than or equal to)

*Example*

```
cmp %ebx, %eax
jle done
```

If the contents of EAX are less than or equal to the contents of EBX, jump to the label *done*. Otherwise, continue to the next instruction.

**cmp** — Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.

*Syntax*

```
cmp <reg>, <reg>
cmp <mem>, <reg>
cmp <reg>, <mem>
cmp <con>, <reg>
```

*Example*

```
cmpb $10, (%ebx)
jeq loop
```

If the byte stored at the memory location in EBX is equal to the integer constant 10, jump to the location labeled *loop*.

**call**, **ret** — Subroutine call and return

These instructions implement a subroutine call and return. The call instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.
The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.

*Syntax*

```
call <label>
ret
```