Intel Instruction Set (gas)

• These slides provide the gas format for a subset of the Intel processor instruction set, including:
  – Operation Mnemonic
  – Name of Operation
  – Syntax
  – Operation
  – Legal Operands
  – Examples
  – Description
  – Effect on Flag Bits
Gas Addressing Mode Syntax

- Register  
  reg  
  %eax
- Immediate data  
  idata  
  $0x1234
- Direct (memory)  
  mem  
  label (in source)
- Register Indirect  
  mem  
  (%eax)
- Register Indirect  
  mem  
  disp(%eax)
  with fixed displacement
- Offset (for jcc/jmp)  
  mem  
  label (in source)
- Port (for in/out)  
  idata  
  $0x12
  (%dx)  
  (%dx)
ADD
Integer Addition

Syntax:
addb src, dest
addw src, dest
addl src, dest

Operation:
dest ← dest + src

Description
This instruction adds the contents of the dest and src operands and stores the result in the location specified by dest. The operands must be of the same size. If the operands are signed integers, the OF flag indicates an invalid result. If the operands are unsigned, the CF flag indicates a carry out of the destination. If the operands are unpacked BCD digits, the AF flag indicates a decimal carry.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>
AND

Boolean AND

Syntax:

- `andb src, dest`
- `andw src, dest`
- `andl src, dest`

Operation:

- `dest ← dest & src`

Legal Operands

- `src`  `dest`  
- `idata, reg`
- `idata, mem`
- `reg, reg`
- `mem, reg`
- `reg, mem`

Examples:

- `andl $10, %eax`
- `andb $10, label`
- `andw %bx, %ax`
- `andl label, %eax`
- `andl %eax, label`

Description

This instruction performs a bit by bit AND operation on the dest and src operands and stores the result in the dest operand. The AND operation is defined as:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
CALL
Near Procedure Call

Syntax:
call dest

Operation:
push %eip
%eip ← dest

Legal Operands

dest
offset %eip ← %eip + offset
call label

mem %eip ← contents of mem
call (%eax)

reg %eip ← contents of reg
call %eax

Description
This instruction pushes the address of the next instruction (EIP) onto the stack. The instruction pointer is then set to the value of the operand.

If the operand is an offset, the operand value is a memory address relative to the current value of %eip. If the operand is a memory address or a register, the subroutine address is taken indirectly from the operand.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
CLI
Clear Interrupt Enable Flag

Syntax:
cli

Legal Operands
none

Examples:
cli

Operation:
IF = 0

Description
This instruction clears the interrupt enable flag (IF) and disables the processing of interrupts. This instruction is used to prevent interrupts during short sequences of code that could fail if an interrupt were allowed to occur in the middle of the code sequence. The IF should not be turned off for “long” periods of time as this could prevent the processing of critical I/O operations such as causing incoming data to be overrun before the processor can execute the ISR code required to process it.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
CMP
Compare Integers

Syntax:
- cmpb op1, op2
- cmpw op1, op2
- cmpl op1, op2

Operation:
NULL \( \leftarrow \) op2 – op1

Description
This instruction subtracts the contents of the src operands from the dest operand and discards the result. Only the eflags register is affected as follows:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Signed Compare</th>
<th>Unsigned Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 &lt; op2</td>
<td>ZF == 0 &amp; &amp; SF == OF</td>
<td>CF == 0 &amp; &amp; ZF == 0</td>
</tr>
<tr>
<td>op1 &lt;= op2</td>
<td>SF == OF</td>
<td>CF == 0</td>
</tr>
<tr>
<td>op1 == op2</td>
<td>ZF == 1</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>op1 &gt;= op2</td>
<td>ZF == 1</td>
<td></td>
</tr>
<tr>
<td>op1 &gt; op2</td>
<td>SF != OF</td>
<td>CF == 1</td>
</tr>
</tbody>
</table>

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>
DEC
Decrement

Syntax:
- decb op1
- decw op1
- decl op1

Operation:
- \( \text{op1} \leftarrow \text{op1} - 1 \)

Description
This instruction subtracts the value 1 from \( \text{op1} \). This instruction is often used to decrement indexes and therefore does not affect the carry flag (CF). In all other respects, it is equivalent to the instruction:

\[
\text{subb} \quad $1, \text{op1}
\]

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>
IN
Input from I/O Port

Syntax:  Legal Operands  Examples:
inb port, %al  port  inw $0x72, %ax
inw port, %ax  idata (one byte)  inw (%dx), %al
inl port, %eax (%dx)  inb (%dx), %al

Operation:
reg ← src (port)

Description
This instruction reads a byte, word, or long word into the specified accumulator from the designated I/O port. If you use an immediate data value in the instruction, you can address only the first 256 ports. If the port is specified in the %dx register, you can access any of the 65536 ports.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>


INC
Increment

Syntax:

<table>
<thead>
<tr>
<th></th>
<th>Legal Operands</th>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>incb op1</td>
<td>op1</td>
<td></td>
</tr>
<tr>
<td>incw op1</td>
<td>reg</td>
<td>incl %eax</td>
</tr>
<tr>
<td>incl op1</td>
<td>mem</td>
<td>incl label</td>
</tr>
</tbody>
</table>

Operation:
op1 ← op1 + 1

Description

This instruction adds the value 1 to op1. This instruction is often used to increment indexes and therefore does not affect the carry flag (CF). In other respects, it is equivalent to the instruction:

    addb $1, op1

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>
### INT

**Software Interrupt**

### Syntax:

```
int vector
```

### Operation:

```
push %eflags
push %cs
push %eip
TF ← 0
if (IDT(vector).type = INTERRUPT_GATE)  IF ← 0
%eip ← destination (IDT(vector))
```

### Description

This instruction is used as a system call. The int 3 instruction is usually encoded as a single byte 0xcc and used as a breakpoint instruction for debuggers.

### Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>x</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
IRET
Interrupt Return

Syntax:
iret

Operation:
(pop %eip
pop %cs
pop %eflags

Description
This instruction signals a return from an interrupt. NOTE: All of the pops shown are executed before the processor starts execution at the restored value of %eip. The three pops are handled as an “atomic” operation, i.e. executed as a single unit.

Flags

<table>
<thead>
<tr>
<th></th>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
Jcc
Jump if Condition

Syntax:
\[
jcc \text{ offset}
\]

Operation:
if (cc) %eip \leftarrow %eip + \text{sign\_extend (offset)}

Description
This instruction executes a conditional jump. It does not change the state of the flags. It executes the jump based on the value(s) of the flag bits as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
<th>After \text{cmp} _ x, y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ja</td>
<td>jump above</td>
<td>CF == 0 &amp;&amp; ZF == 0</td>
<td>unsigned y &gt; x</td>
</tr>
<tr>
<td>jae</td>
<td>jump above or equal</td>
<td>CF == 0</td>
<td>unsigned y &gt;= x</td>
</tr>
<tr>
<td>jb</td>
<td>jump below</td>
<td>CF == 1</td>
<td>unsigned y &lt; x</td>
</tr>
<tr>
<td>jbe</td>
<td>jump below or equal</td>
<td>CF == 1 | ZF == 1</td>
<td>unsigned y &lt;= x</td>
</tr>
<tr>
<td>jc</td>
<td>jump if carry</td>
<td>CF == 1</td>
<td></td>
</tr>
<tr>
<td>jcxz</td>
<td>jump if %cx == 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jecxz</td>
<td>jump if %ecx == 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Jump if Condition

Description (Continued)

<table>
<thead>
<tr>
<th>Jcc</th>
<th>Description</th>
<th>Condition</th>
<th>After <code>cmp_ x, y</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td>jump equal</td>
<td>ZF == 1</td>
<td>y == x</td>
</tr>
<tr>
<td><code>jg</code></td>
<td>jump greater</td>
<td>SF == OF &amp;&amp; ZF = 0</td>
<td>signed y &gt; x</td>
</tr>
<tr>
<td><code>jge</code></td>
<td>jump greater or equal</td>
<td>SF == OF</td>
<td>signed y &gt;= x</td>
</tr>
<tr>
<td><code>jl</code></td>
<td>jump less</td>
<td>SF != OF</td>
<td>signed y &lt; x</td>
</tr>
<tr>
<td><code>jle</code></td>
<td>jump less or equal</td>
<td>SF != OF</td>
<td></td>
</tr>
<tr>
<td><code>jna</code></td>
<td>jump not above</td>
<td>(same as jbe)</td>
<td></td>
</tr>
<tr>
<td><code>jnae</code></td>
<td>jump not above or equal</td>
<td>(same as jb)</td>
<td></td>
</tr>
<tr>
<td><code>jnb</code></td>
<td>jump not below</td>
<td>(same as jae)</td>
<td></td>
</tr>
<tr>
<td><code>jnbe</code></td>
<td>jump not below or equal</td>
<td>(same as ja)</td>
<td></td>
</tr>
<tr>
<td><code>jnc</code></td>
<td>jump no carry</td>
<td>CF == 0</td>
<td></td>
</tr>
<tr>
<td><code>jne</code></td>
<td>jump not equal</td>
<td>ZF == 0</td>
<td>y != x</td>
</tr>
<tr>
<td><code>jng</code></td>
<td>jump not greater</td>
<td>(same as jle)</td>
<td></td>
</tr>
<tr>
<td><code>jnge</code></td>
<td>jump not greater or equal</td>
<td>(same as jl)</td>
<td></td>
</tr>
<tr>
<td><code>jnl</code></td>
<td>jump not less</td>
<td>(same as jge)</td>
<td></td>
</tr>
<tr>
<td><code>jnle</code></td>
<td>jump not less or equal</td>
<td>(same as jg)</td>
<td></td>
</tr>
</tbody>
</table>
Jcc (Continued)
Jump if Condition

Description (Continued)

- jno  jump no overflow  OF == 0
- jnp  jump no parity    PF == 0
- jns  jump no sign      SF == 0
- jnz  jump not zero     ZF == 0
- jo   jump if overflow  OF == 1
- jp   jump if parity    PF == 1
- jpe  jump if parity even PF == 1
- jpo  jump if parity odd PF == 0
- js   jump if sign      SF == 1
- jz   jump if zero      ZF == 1

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
JMP

Jump

Syntax:
jmp dest

Legal Operands

dest
offset  %eip ← %eip + offset
mem     %eip ← contents of mem
reg     %eip ← contents of reg

Examples:
jmp label
jmp (%eax)
jmp %eax

Operation:
%eip ← dest

Description
This instruction executes an unconditional jump. It doesn’t change the state of the flags.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
LEA
Load Effective Address

Syntax:
lea src, dest

Operation:
dest ← address (src)

Description
This instruction loads the address specified by the memory operand into the destination register. No memory access cycle takes place. It doesn’t change the state of the flags.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
LOOPcc
Decrement %ecx and Branch

Syntax:
loop offset
loopz offset
loopnz offset
loope offset
loopne offset

Operation:
%ecx ← %ecx - 1
if (cc & (%ecx != 0)) %eip ← %eip + offset

Description
These instructions support a decrement and branch operation. For all variants other than LOOP, the decrement and branch is combined with a test on the ZF bit. A loop counter is assumed in the register %ecx.

Flags
<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
MOV
Move Data

**Syntax:**
- movb src, dest
- movw src, dest
- movl src, dest

**Operation:**
dest ← src

**Legal Operands**
- src, dest
- idata, reg
- idata, mem
- reg, reg
- mem, reg
- reg, mem

**Examples:**
- movl $10, %eax
- movb $10, label
- movw %bx, %ax
- movl label, %eax
- movl %eax, label

**Description**
This instruction copies the contents of the src operand into dest.

**Flags**

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
NEG
Not

**Syntax:**
- negb op1
- negw op1
- negl op1

**Legal Operands**
- op1
- reg
- mem

**Examples:**
- negl %eax
- negl label

**Operation:**
- op1 $\leftarrow - \text{op1}$

**Description**
This instruction performs a two’s complement on the operand.

**Flags**

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>

NOP
No Operation

Syntax:

| nop |

Operation:

(nothing)

Description
This instruction does nothing except take time to be executed. Hence, it is used in timing loops or where the execution of the next instruction needs to be delayed for some reason, e.g. giving enough time for a hardware register to be ready.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
**Syntax:**
notb op1
notw op1
notl op1

**Legal Operands**  
op1
reg
mem

**Examples:**
notl %eax
notl label

**Operation:**
op1 ← ~ op1

**Description**
This instruction performs a logical NOT or one’s complement on the operand. The flags are unaffected.

**Flags**

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
OR

Boolean OR

**Syntax:**
orb src, dest
orw src, dest
orl src, dest

**Operation:**
dest ← dest | src

**Description**
This instruction performs a bit by bit OR operation on the dest and src operands and stores the result in the dest operand. The OR operation is defined as:

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Flags**

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
OUT
Output to I/O Port

Syntax:
outb %al, port
outw %ax, port
outl %eax, port

Legal Operands
port
idata (one byte)
(%dx)

Examples:
outb %al, $0x72
outw %ax, (%dx)

Operation:
dest (port) ← reg

Description
This instruction writes a byte, word, or long word from the specified accumulator to the designated I/O port. If you use an immediate data value in the instruction, you can address only the first 256 ports. If the port is specified in the %dx register, you can access any of the 65536 ports.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
POP
Pop Value off Stack

Syntax:
- popw dest
- popl dest

Operation:
dest ← contents of mem at %esp
%esp ← %esp + (w)? 2 : 4

Description
This instruction pops the current value at the top of the stack (lowest memory address), stores it in the dest operand, and increments the stack pointer by the size of the value.

popl is always preferred to keep the stack pointer aligned on long word boundaries, i.e. addresses with the two LSBs = 0.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
PUSH
Push Value onto Stack

**Syntax:**
- `pushw src`
- `pushl src`

**Operation:**
- `%esp ← %esp - (w)? 2 : 4`
- contents of mem at `%esp ← src`

**Description**
This instruction decrements the stack pointer by the size of the value, and stores the value of the `src` operand onto the top of the stack (lowest memory address).

`pushl` is always preferred to keep the stack pointer aligned on long word boundaries, i.e. addresses with the two LSBs = 0.

**Flags**

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
RET
Near Return from Subroutine

Syntax:
ret count

Operation:
%eip ← pop (%esp)
%esp ← %esp + count

Description
This instruction restores the instruction pointer to the value it held before the previous call instruction. The value of the EIP that had been saved on the stack is popped. If the count operand is present, the count value is added to %esp, removing arguments that were pushed onto the stack for the subroutine call.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
SAL / SHL
Shift Arithmetic Left / Shift Logical Left

Syntax:
- salb count, dest
- salw count, dest
- sall count, dest

Operation:
dest \leftarrow dest \ll count

Description
This instruction shifts the dest operand count bits to the left and fills the LSBs with zeros. It updates the flag bits appropriately. (Arithmetic and logical are the same.)

Legal Operands
- count
- dest
- idata
- reg
- mem
- %cl
- reg
- mem

Examples:
- salw $4, %ax
- salb $4, label
- shll %cl, %eax
- shlw %cl, label

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
SAR
Shift Arithmetic Right

Syntax:

<table>
<thead>
<tr>
<th>dest</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>sarb</td>
<td>count, dest</td>
</tr>
<tr>
<td>sarw</td>
<td>count, dest</td>
</tr>
<tr>
<td>sarl</td>
<td>count, dest</td>
</tr>
</tbody>
</table>

Operation:

dest ← dest >> count
(with sign bit extension)

Description

This instruction shifts the dest operand count bits to the right and fills the MSBs with copies of the sign bit. It updates the flag bits appropriately. (Preserves sign.)

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
**SHR**

Shift Logical Right

**Syntax:**
- `shrb count, dest`
- `shrw count, dest`
- `shrl count, dest`

**Operation:**
- `dest ← dest >> count`
- (without sign bit extension)

**Description**
This instruction shifts the dest operand count bits to the right and fills the MSBs with zeros. It updates the flag bits appropriately. (Does not preserve sign.)

**Legal Operands**
- `count`
- `dest`
- `idata`
- `reg`
- `idata`
- `mem`
- `%cl`
- `reg`
- `%cl`
- `mem`

**Examples:**
- `shrw $4, %ax`
- `shrb $4, label`
- `shrl %cl, %eax`
- `shrw %cl, label`

**Flags**
- `OE`
- `DE`
- `IF`
- `TF`
- `SF`
- `ZF`
- `AF`
- `PF`
- `CF`

<table>
<thead>
<tr>
<th></th>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
STI
Set Interrupt Enable Flag

Syntax:            Legal Operands            Examples:
sti              none                  sti

Operation:
IF = 1

Description
This instruction sets the interrupt enable flag (IF) and enables the processing of interrupts. This instruction is used when the code is ready to process interrupts.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
SUB
Integer Subtraction

Syntax:
subb src, dest
subw src, dest
subl src, dest

Operation:
dest ← dest - src

Description
This instruction subtracts the contents of the src operand from the dest operand and stores the result in the location specified by dest. The operands must be of the same size. If the operands are signed integers, the OF flag indicates an invalid result. If the operands are unsigned, the CF flag indicates a borrow into the destination. If the operands are unpacked BCD digits, the AF flag indicates a decimal borrow.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
TEST
Logical Compare

Syntax:
- testb src, dest
- testw src, dest
- testl src, dest

Operation:
NULL ← dest & src

Description
This instruction ANDs the contents of the src operand with the dest operand and discards the result. It sets the flags.

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>?</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
XOR
Boolean XOR

Syntax:
- xorb src, dest
- xorw src, dest
- xorl src, dest

Operation:
dest ← dest ^ src

Description
This instruction performs a bit by bit XOR operation on the dest and src operands and stores the result in the dest operand. The XOR operation is defined as:

<table>
<thead>
<tr>
<th>XOR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Flags

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>