LDS/LES/LSS

Load far pointer ~~ outside of current segment

General forms:

lds reg,mem  {E.g., load reg w/value @ mem, & seg w/mem+2
lseg reg,mem

XCHG

Exchange values

General Forms:

xchg  reg,reg
xchg  reg,mem
xchg  mem,reg

Examples:

xchg  ah,al  ;[AH] < - > [AL]

DUP

Duplicate data for initialization

\[ \text{dw } 128 \text{ dup (?) } \quad \text{-- “Uninitialized” memory} \]

Repeat Prefixes

These are used to cause auto repeat of a string instruction. In general they use CX as a repeat counter and will decrement CX and repeat the instruction until CX = 0.

REP  \quad \text{repeat until } CX = 0
REPZ \quad \text{repeat while } ZF=1 \text{ and } CX!=0
REPE \quad \text{same as REPZ}
REPNZ \quad \text{repeat while } ZF=0 \text{ and } CX!=0
REPNE \quad \text{same as REPNZ}
Programmer’s Model of the Machine

Processor Registers

Flags

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>GP register</td>
</tr>
<tr>
<td>BX</td>
<td>GP register or BASE register</td>
</tr>
<tr>
<td>CX</td>
<td>GP register or count register</td>
</tr>
<tr>
<td>DX</td>
<td>GP register</td>
</tr>
<tr>
<td>SI</td>
<td>INDEX register</td>
</tr>
<tr>
<td>DI</td>
<td>INDEX register</td>
</tr>
<tr>
<td>BP</td>
<td>BASE register, stack frame pointer</td>
</tr>
<tr>
<td>SP</td>
<td>stack pointer</td>
</tr>
<tr>
<td>IP</td>
<td>instruction pointer</td>
</tr>
<tr>
<td>CS</td>
<td>code segment</td>
</tr>
<tr>
<td>DS</td>
<td>data segment</td>
</tr>
<tr>
<td>ES</td>
<td>extra data segment</td>
</tr>
<tr>
<td>SS</td>
<td>stack segment</td>
</tr>
</tbody>
</table>

Segment Selection for Memory Addressing:

- Code Segment: CS
- Data Segment: DS
- Stack Segment: SS

Implied segment usage by index, pointer and base registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>SS</td>
</tr>
<tr>
<td>BP</td>
<td>SS</td>
</tr>
<tr>
<td>BX</td>
<td>DS</td>
</tr>
<tr>
<td>SI</td>
<td>DS</td>
</tr>
<tr>
<td>DI</td>
<td>DS (or ES for some string instructions)</td>
</tr>
</tbody>
</table>

Based addressing using BX uses DS. ([BX]+offset)
Based addressing using BP uses SS ([BP]+offset)
Based Indexed addressing uses DS ([BP]+[SI]+offset)

Default segment usage may be overridden with an explicit segment override prefix byte.
FLAGS – Condition Codes
These are status indicators or condition codes that provide information about the most recently executed instruction that sets flags (not all instructions modify the flags).

SF (sign): Indicates that the last instruction produced a NEGATIVE result. SF=1 -> result is negative; SF=0 -> result is positive

ZF (zero): Indicates that the last instruction produced a ZERO result. ZF=1 -> result is 0; ZF=0 -> result is not 0

CF (carry): Indicates that a carry out of the most significant bit occurred. CF=1 -> carry occurred; CF=0 -> no carry occurred

AF (aux carry): Indicates that a carry occurred out of bit3 to bit 4 (used for BCD). AF=1 -> aux carry occurred; AF=0 -> no aux carry occurred

PF (parity): Indicates the parity of the last result. PF=1 -> even parity; PF=0 -> odd parity

Arithmetic and Logical Instructions

Two operand Arithmetic and Logical Instructions: The two operand arithmetic and logical operations all have the same general format. These instructions set all of the condition codes.

General forms:
- opc dest,src
- opc reg,idata
- opc mem,idata
- opc reg,reg
- opc reg,mem
- opc mem,reg

The specific instructions are:
- ADD - addition: add dest,src
- SUB - subtraction: sub dest,src
- CMP - compare: cmp dest,src
- AND - bitwise and: and dest,src
- OR - bitwise or: or dest,src
- XOR - bitwise exclusive or: xor dest,src
**Increment and Decrement Instructions**: These instructions increment or decrement, i.e. add 1 or subtract 1 from the operand. They set all condition codes except CF (carry).

General form:

```
opc  dest
```

```
opc  reg
opc  mem
```

The specific instructions are:

- **INC** increment  `inc dest`
- **DEC** decrement  `dec dest`

**Multiplication and Division**: These instructions use specific registers for one of the operands and the result of the operation. The result will be in AX for 8 bit operations (AL as multiplicand), DX:AX for 16 bit operations.

For division, the quotient will be in the low half of the result and the remainder will be in the high half.

General Form:

```
opc  src
```

```
opc  reg
opc  mem
```

Specific Instructions:

- **MUL** - unsigned multiply  `mul src`
- **IMUL** signed multiply  `imul src`
- **DIV** unsigned divide  `div src`
- **IDIV** signed divide  `idiv src`
Input/Output Instructions

IN      Input from port
  General Forms
  in     acc,port
  in     AL, 20H
  in     acc,dx

INS     Input string from port

OUT     Output to port
  General Forms
  out    port,acc
  out    27h,AX
  out    dx,acc

OUTS    Output string to port

Control Transfer Instructions

Unconditional Jump: There are two variations of unconditional jump, NEAR and FAR. A near jump is a jump within the same segment, and the CS segment register is not changed. A far jump is a jump to another segment, and the CS segment register is changed.

General Forms: Near JMP
  jmp  offset   ; IP <- IP+offset
  jmp  reg      ; IP <- reg
  jmp  mem      ; IP <- [mem]

General Forms: Far JMP
  jmp  idata    ; CS:IP <- idata
  jmp  mem      ; CS:IP <- [mem]

Conditional Jump: There are only NEAR conditional jumps. The jump is taken if the condition code is met, if not, the jump is not taken and the next instruction following the jump instruction will be executed. On 8086-80286 processors, the offset to the jump target is only 1 byte in size, and therefore on these processors, a conditional jump can only be taken to a destination within +/- 128 bytes of the jump (approximately). On 80386 and later processors, this restriction is removed, and a full 32 bit offset can be used to the jump target.

General Form:
  jcc  offset   ; IP <- IP+offset if cc is true
Examples:

- **jc off**  jump if carry true       CF=1
- **jnc off** jump if carry not true  CF=0
- **jz off**  jump if zero true       ZF=1
- **jnz off** jump if zero not true   ZF=0
- **jle off** jump if less than or equal to                     SF!=0
- **jnle off** jump if not less than or equal to CF=0 &        ZF=0
- **jcxz off** jump if CX = 0 This is a special case

**NOTE:** Look at the table on page 134 of the text for a complete list of conditional jump instructions.

**Subroutine Call:** As with the unconditional jump, there are NEAR and FAR versions of this instruction. The CALL instruction pushes the return address (current value of IP for CS:IP) onto the stack, and then loads IP or CS:IP with a new value. This transfers control to the new location, with the previous location saved at the top of the stack.

General Forms: Near CALL

- **call offset** ;IP pushed, IP<- IP+offset
- **call reg**   ;IP pushed; IP <- [reg]
- **call mem**   ;IP pushed; IP <- [mem]

General Forms: Far CALL

- **call idata** ;CS:IP pushed; CS:IP <- idata
- **call mem**   ;CS:IP pushed; CS:IP <- [mem]

**Return from Subroutine:** The return from subroutine is accomplished by popping the value from the top of the stack into either IP for a NEAR return, or CS:IP for a FAR return. The size of the call and the return must match. (i.e. a near call requires a near return, and a far call requires a far return).

General Form:

- **ret**       ;near return
- **retf**      ;far return

**Note:** If PROC declarations are used properly, the assembler will understand whether a subroutine is NEAR or FAR, and generate the proper sized return when a RET instruction is used. In special cases, a far return can be forced by using the RETF instruction.