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:
- \texttt{opc dest,src}
- \texttt{opc reg,ida}
- \texttt{opc mem,ida}
- \texttt{opc reg,reg}
- \texttt{opc reg,mem}
- \texttt{opc mem,reg}

The specific instructions are:
- \texttt{ADD} - addition: \texttt{add dest,src}
- * \texttt{ADC} add with carry: \texttt{adc dest,src}
  - Add src to dest & increment by 1 if CF=1
- \texttt{SUB} subtraction: \texttt{sub dest,src}
- * \texttt{SBB} subtract with borrow: \texttt{sbb dest,src}
  - Subtract src from dest and decrement by 1 if CF=1
- \texttt{CMP} compare: \texttt{cmp dest,src}
- \texttt{AND} bitwise and: \texttt{and dest,src}
- \texttt{OR} bitwise or: \texttt{or dest,src}
- \texttt{XOR} bitwise exclusive or: \texttt{xor dest,src}
- * \texttt{TEST} bitwise test: \texttt{test dest,src}
  - Performs a bitwise and, but doesn’t store result
  - \texttt{SETS FLAGS; TEST:AND as CMP:SUB}

One operand Arithmetic and Logical Instructions: The instructions all have the same general format.

General form:
- \texttt{opc reg}
- \texttt{opc mem}
**NEG**  two’s complement negation:  \textit{neg dst}

Special case instruction. Bitwise complement, \textit{doesn’t set flags}

* NOT  bitwise complement:  \textit{not dest}

And saves result.

**Binary Coded Decimal (BCD):**  Decimal coding in binary…

Decimal digits: 0,1,2,…9

Requires 1 nibble (4 bits) to represent 0000,… 1001

Unpacked BCD: 1 digit/byte  Packed BCD: 2 digits/byte

12345

Packed: 00000001 00100011 01000101

**Two’s Complement:**

\textbf{Noun:}  Representation system

\textbf{Verb:}  Operation of conversion

**Representation:**

\[ v = \text{sign bit} \]

\[
\begin{array}{c|c}
7 = & 0111 \\
\ldots & \ldots \\
0 = & 0000 \\
-1 = & 1111 \\
\ldots & \ldots \\
-7 = & 1001 \\
\end{array}
\]

\textbf{Must know number of data digits} to preserve the sign bit!

**Operation:**

\textit{Flip the bits and add 1.}

\[
\begin{array}{c}
1 \\
34 \\
+ 98 \\
\hline
2 \\
\end{array}
\]

Segmented Memory Model and Assembly Language.
The Intel X86 processors use a segmented memory model. All code and data must reside in some segment. In order for the processor to access a given element of data, the segment address of the segment in which it resides must be loaded in some segment register.

When the assembler is translating the source program, it must know the segment where each element being translated should reside. Additionally, when an instruction specifies a reference to a memory location, it needs to know which segment values are currently in each segment register so that it can determine whether the specified data is reachable using the values currently in the segment registers.

Assembler directives (pseudo-ops) are used to specify to the assembler which segments code and data are being generated in and what segment values are currently in the segment registers.

An Intel X86 assembly language program is made up of segments. All code and data for a program is declared within the context of some segment. The programmer declares segments via the `SEGMENT` directive. The first time that the segment directive is used, the segment parameters can be specified (alignment and combine class). Each subsequent time, they are not necessary, but if given, must match the original declaration. If a project has multiple source files, it is generally a good idea to put all of the segment declarations in a common include file and include that file in each source module. This ensures that the segment declarations agree between modules.

When the linker combines object modules to produce the final executable, it combines the data from each segment for each object module, in the order in which they are seen. For example: Object modules A, B, and C each declare segment TEXT, and the linker sees them in the order A, B, C. The linker will produce one TEXT segment in the output file. The A data will appear first in the output file, followed by the B data and finally the C data. In 16 bit real mode, no segment can be larger than 64K bytes. If the result of combining object modules results in more than 64K of data in a segment, the linker will issue an error.

**Some Assembler Directives**

**SEGMENT**: The `SEGMENT` directive is used to begin a segment in memory. The syntax is as follows:

```
name SEGMENT style, alignment, combine-class
```

Example: 

```
TEXT SEGMENT PUBLIC, WORD, ‘CODE’
```

The `name` is any valid symbolic name for the segment. Each use of the segment directive for a given `name`, refers to the same segment.
Style can be either PUBLIC or STACK. It informs the linker of visibility and type of the segment. If a segment is declared PUBLIC, segments with the same name in different modules will be treated as the same segment, and all occurrences of the given segment in each module will be concatenated to form the complete segment. If they are not PUBLIC, each segment with the same name will be treated as a different segment. STACK segments are used to reserve stack space. The linker doesn’t concatenate them but rather reserves as much space as the largest instance of the segment.

Alignment specifies the boundary on which the beginning of the segment is to be aligned. The choices are BYTE, WORD, DWORD, PARA (for paragraph, 16 bytes), and PAGE (for page or 4k bytes). If no alignment is specified, BYTE will be used.

Combine-Class is used by the linker for grouping similar segments in memory. All segments with the same combine-class will be placed together in memory. Normally, all code segments are given the combine class ‘CODE’ and all data segments are given the combine class ‘DATA’.

ENDS: The ENDS directive is used to end a segment. The syntax is:

```
name ENDS
```

Where `name` matches the name of the preceding SEGMENT directive.

Example: `_TEXT ENDS`

ASSUME: The ASSUME directive is used to inform the assembler of what values are loaded into the segment registers. ASSUME has the following syntax:

```
ASSUME seg-reg:seg-name
```

Example: `ASSUME CS:CODE, DS:DATA, ES: DATA`

This tells the assembler that the segment address of the segment named CODE has been loaded into CS and that the segment address of the segment named DATA has been loaded into DS and ES. The assembler can then use this information to determine whether memory variables are reachable using the addresses currently in the segment registers, and issue error messages if not. NOTE: the ASSUME directive doesn’t cause any value to be loaded into a segment register; it is merely a statement to the assembler that it may assume that that has happened. It is still the programmer’s responsibility to ensure that the specified value has, in fact, been loaded into the segment register.

The special syntax: ASSUME sreg:NOTHING can be used to tell the assembler to assume nothing about a given segment register and to not use it to generate an address.

Example: `ASSUME ES:NOTHING`