Review conversion from one base to another in text as well as two's complement.

**Table 1: ASCII (American Standard Code for Information Interchange)**

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NUL</td>
<td>32</td>
<td>20</td>
<td></td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SOH</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>STX</td>
<td>34</td>
<td>22</td>
<td>`</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ETX</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>EOT</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ENQ</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ACK</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BEL</td>
<td>39</td>
<td>27</td>
<td>’</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>BS</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>TAB</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>LF</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>VT</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>FF</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>CR</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>SO</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>SI</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
</tbody>
</table>
### Table 2: ASCII (American Standard Code for Information Interchange)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
<th>Dec</th>
<th>Hex</th>
<th>Sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10</td>
<td>DLE</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DC1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>DC2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>DC3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>DC4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>NAK</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>SYN</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>ETB</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>CAN</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>EM</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>SUB</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>ESC</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>FS</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>GS</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>RS</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>US</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
</tbody>
</table>
Assembly Directives

**ASCII:** Stored using an assembler directive `db`:

```
floatstr db 'Float number -> %f ', 10, 0
main1_str: db ' Rectangular Areas', 10, 0
temp_buf: times 200 db 0
temp_buf_size: equ $-temp_buf
```

Word-sized (`dw`) and doubleword-sized data (`dd`):

```
neg_exponent: dd -100
```

**Little endian:** Least significant byte is always stored in the lowest memory location.

<table>
<thead>
<tr>
<th>303H</th>
<th>302H</th>
<th>301H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12H</td>
<td>34H</td>
</tr>
</tbody>
</table>

Storage of the number 1234

High-order byte  
Low-order byte
Floating Point Formats

For single precision, the sign bit + 8-bit exponent + 24-bit mantissa = 33 bits!

The mantissa has a hidden 1 bit in the leftmost position that allows it to be stored as a 23-bit value.

The mantissa is first normalized to be $\geq 1$ and $< 2$, e.g., 12 in binary is 1100, normalized is $1.1 \times 2^3$.

The exponent is also biased by adding 127 (single) or 1023 (double), e.g. the 3 in the previous example is stored as $127 + 3 = 130$ (82H).
Floating Point Formats and Directives

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Normal</th>
<th>Sign</th>
<th>Expon</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>+12</td>
<td>1100</td>
<td>1.1 x 2³</td>
<td>0</td>
<td>10000010</td>
<td>1000000 00000000 00000000</td>
</tr>
</tbody>
</table>

There are two exceptions:
The number 0.0 is stored as all zeros.
The number infinity is stored as all ones in the exponent and all zeros in the mantissa.
(The sign bit is used to indicate + or - infinity.)

Directive is \textit{dd} for single, \textit{dq} for double and \textit{dt} for 10 bytes:

\begin{itemize}
  \item \textit{dd} 1.2
  \item \textit{dq} 1.e+10
  \item \textit{dt} 3.141592653589793238462
\end{itemize}
**Intel Assembly**

Format of an assembly instruction:

<table>
<thead>
<tr>
<th>LABEL</th>
<th>OPCODE</th>
<th>OPERANDS</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA1</td>
<td>db</td>
<td>00001000b</td>
<td>;Define DATA1 as decimal 8</td>
</tr>
<tr>
<td>START:</td>
<td>mov</td>
<td>eax, ebx</td>
<td>;Copy ebx to eax</td>
</tr>
</tbody>
</table>

**LABEL:**

Stores a symbolic name for the memory location that it represents.

**OPCODE:**

The instruction itself.

**OPERANDS:**

A register, an immediate or a memory address holding the values on which the operation is performed.

There can be from 0 to 3 operands.
Data Addressing Modes

Data registers:

16-bit registers
ah  ax  al
8-bit names

32-bit extensions

Let's cover the data addressing modes using the *mov* instruction.

Data movement instructions move data (bytes, words and doublewords) between registers and between registers and memory.

Only the *movs* (strings) instruction can have both operands in memory.

Most data transfer instructions do not change the EFLAGS register.
**Data Addressing Modes**

**Register**

```
mov eax, ebx
```

**Immediate**

```
mov ch, 0x4b
```

**Direct** (eax), **Displacement** (other regs)

```
mov [0x4321], eax
```

Source

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>4b</td>
</tr>
</tbody>
</table>

Dest

<table>
<thead>
<tr>
<th>Register</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>[0x4321]</td>
</tr>
</tbody>
</table>
Data Addressing Modes

- **Register Indirect**
  
  \[ \text{mov} \ [ebx], \ cl \]
  
  Any of \(eax, ebx, ecx, edx, ebp, edi\) or \(esi\) may be used.

- **Base-plus-index**
  
  \[ \text{mov} \ [ebx+esi], \ ebp \]
  
  Any combination of \(eax, ebx, ecx, edx, ebp, edi\) or \(esi\).

- **Register relative**
  
  \[ \text{mov} \ cl, \ [ebx+4] \]
  
  A second variation includes: \(\text{mov} \ eax, \ [ARR+ebx]\)
Data Addressing Modes

- **Base relative-plus-index**

  A second variation includes: `mov eax, [ebx+edi+4]`

- **Scaled-index**

  A second variation includes: `mov eax, ebx*2+ecx+offset`
  Scaling factors can be 2X, 4X or 8X.
Data Addressing Modes

Register addressing

Note: `mov` really COPIES data from the source to destination register.

- Never mix an 16-bit register with a 32-bit, etc.
  
  For example

  ```
  mov eax, bx ;ERROR: NOT permitted.
  ```

- None of the `mov` instruction effect the EFLAGS register.

Immediate addressing:

The value of the operand is given as a constant in the instruction stream.

```
mov eax, 0x12345
```

- Use `b` for binary, `q` for octal and nothing for decimal.

- ASCII data requires a set of apostrophes:

```
mov eax, ‘A’ ;Moves ASCII value 0x41 into eax.
```
Data Addressing Modes

Register and immediate addressing example:

```assembly
global main
section .text ; start of the code segment.
main:
    mov  eax, 0  ; Immediate addressing.
    mov  ebx, 0x0000
    mov  ecx, 0
    mov  esi, eax  ; Register addressing.
    ...
```

Direct addressing:

Transfers between memory and `al`, `ax` and `eax`.

Usually encoded in 3 bytes, sometime 4:

```assembly
    mov  al, DATA1  ; Copies a byte from DATA1.
    mov  al, [0x4321]  ; Some assemblers don’t allow this.
    mov  al, ds:[0x1234]
    mov  DATA2, ax  ; Copies a word to DATA2.
```
Data Addressing Modes

Displacement:

\[
\text{mov cl, DATA1} \quad ;\text{Copies a byte from DATA1.}
\]

\[
\text{mov edi, SUM} \quad ;\text{Copies a doubleword from SUM.}
\]

Displacement instructions are encoded with up to 7 bytes (32 bit register and a 32 bit displacement).

Direct and displacement addressing example:

\[
\begin{align*}
\text{global main} \\
0000 & \quad \text{section .data} \\
0000\ 10 & \quad \text{DATA1} \hspace{0.5cm} \text{db} \hspace{0.5cm} 0x10 \\
0001\ 00 & \quad \text{DATA2} \hspace{0.5cm} \text{db} \hspace{0.5cm} 0 \\
0000 & \quad \text{section .text} \\
\text{main:} & \\
0017\ A0\ 0000\ R & \quad \text{mov al, DATA1} \\
001A\ 8B\ 1E\ 0001\ R & \quad \text{mov bx, DATA2}
\end{align*}
\]

Note: Direct addressing (using \textbf{al}) requires 3 bytes to encode while Displacement (using \textbf{bx}) requires 4.
Data Addressing Modes

Register Indirect addressing:
Offset stored in a register is added to the segment register.

\[
\text{mov } ecx, [ebx] \\
\text{mov } [edi], [ebx]
\]

The memory to memory mov is allowed with string instructions.
Any register EXCEPT esp for the 80386 and up.
For eax, ebx, ecx, edx, edi and esi: The data segment is the default.
For ebp: The stack segment is the default.

Some versions of register indirect require special assembler directives byte, word, or dword

\[
\text{mov } al, [edi] \quad ;\text{Clearly a byte-sized move.} \\
\text{mov } [edi], 0x10 \quad ;\text{Ambiguous, assembler can’t size.}
\]

Does [edi] address a byte, a word or a double-word?
Use:

\[
\text{mov } byte [edi], 0x10 \quad ;A \text{ byte transfer.}
\]
Data Addressing Modes

**Base-Plus-Index addressing:**
Effective address computed as:
seg_base + base + index.

*Base registers:* Holds starting location of an array.
- ebp (stack)
- ebx (data)
- Any 32-bit register except esp.

*Index registers:* Holds offset location.
- edi
- esi
- Any 32-bit register except esp.

```
mov ecx, [ebx+edi] ; Data segment copy.
mov ch,  [ebp+esi] ; Stack segment copy.
mov dl,  [eax+ebx] ; EAX as base, EBX as index.
```
**Data Addressing Modes**

*Base-Plus-Index addressing:*

\[
\text{mov } edx, [ebx+edi]
\]

### Memory

- **eax**
- **ebx** 0 0 0 0 1 0 0 0
- **ecx**
- **edx** F 0 1 2  A B 0 3
- **esp**
- **ebp**
- **edi** 0 0 1 0
- **esi**

### Physical Address Calculations

1. **cs** 0 1 0 0
2. **ds** 0 1 0 0
3. **es**
4. **ss**

**Seg Base Trans.**

+ 1010H

**Paging**

**Physical Address**

F012AB03
Data Addressing Modes

Register Relative addressing:

Effective address computed as:
\[ \text{seg\_base} + \text{base} + \text{constant}. \]

- `mov eax, [ebx+1000H] ; Data segment copy.
- `mov [ARRAY+esi], BL ; Constant is ARRAY.
- `mov edx, [LIST+esi+2] ; Both LIST and 2 are constants.
- `mov edx, [LIST+esi-2] ; Subtraction.

Same default segment rules apply with respect to `ebp, `ebx, `edi and `esi.

Displacement constant is any 32-bit signed value.

Base Relative-Plus-Index addressing:

Effective address computed as:
\[ \text{seg\_base} + \text{base} + \text{index} + \text{constant}. \]

- `mov dh, [ebx+edi+20H] ; Data segment copy.
- `mov ax, [FILE+ebx+edi] ; Constant is FILE.
- `mov [LIST+ebp+esi+4], dh ; Stack segment copy.
- `mov eax, [FILE+ebx+ecx+2] ; 32-bit transfer.

Designed to be used as a mechanism to address a two-dimensional array.
**Data Addressing Modes**

**Base Relative-Plus-Index addressing:**

\[
\text{MOV ax, [ebx+esi+100H]}
\]

<table>
<thead>
<tr>
<th>eax</th>
<th>A 3 1 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>ecx</td>
<td>0 0 2 0</td>
</tr>
<tr>
<td>edx</td>
<td></td>
</tr>
<tr>
<td>esp</td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td></td>
</tr>
<tr>
<td>edi</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>esi</td>
<td></td>
</tr>
<tr>
<td>cs</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>ds</td>
<td></td>
</tr>
<tr>
<td>es</td>
<td></td>
</tr>
<tr>
<td>ss</td>
<td></td>
</tr>
</tbody>
</table>

Memory

\[
A316
\]

100H

1030H

Seg
Base
Trans.

Paging
**Data/Code Addressing Modes**

*Scaled-Index addressing:*

Effective address computed as:

\[
\text{seg}_\text{base} + \text{base} + \text{constant} \times \text{index}.
\]

- **mov** `eax, [ebx+4*ecx]` ; Data segment DWORD copy.
- **mov** `[eax+2*edi-100H], cx` ; Whow !
- **mov** `eax, [ARRAY+4*ecx]` ; Std array addressing.

**Code Memory-Addressing Modes:**

Used in `jmp` and `call` instructions.

Three forms:

- Direct
- PC-Relative
- Indirect

**Direct**

Absolute jump address is stored in the instruction following the opcode.
Code Addressing Modes

An intersegment jump:

- Opcode: EA
- Offset (low): 0000
- Offset (high): 0000
- Segment(low): 00
- Segment(high): 10

This far jmp instruction loads cs with 1000H and eip with 00000000H.
A far call instruction is similar.

PC-Relative

- A displacement is added to the EIP register.
- This constant is encoded into the instruction itself, as above.

Intrasegment jumps:
- Short jumps use a 1-byte signed displacement.
- Near jumps use a 4-byte signed displacement.
  - The assembler usually computes the displacement and selects the appropriate form.
**Code Addressing Modes**

**Indirect**

Jump location is specified by a register.

There are three forms:

- **Register:**
  
  Any register can be used: eax, ebx, ecx, edx, esp, ebp, edi or esi.

  ```
  jmp  eax ; Jump within the code segment.
  ```

- **Register Indirect:**

  Intra-segment jumps can also be stored in the data segment.

  ```
  jmp  [ebx] ; Jump address in data segment.
  ```

- **Register Relative:**

  ```
  jmp  [TABLE+ebx] ; Jump table.
  jmp  [edi+2]
  ```
Stack Addressing Modes

The stack is used to hold temporary variables and stores return addresses for procedures. *push* and *pop* instructions are used to manipulate it. *call* and *ret* also refer to the stack implicitly.

Two registers maintain the stack, **esp** and **ss**. A **LIFO** (Last-in, First-out) policy is used.

The stack grows toward lower address.

Data may be pushed from any of the registers or segment registers.

Data may be popped into any register except **cs**.

```plaintext
popfd ;Pop doubleword for stack to EFLAG.
pushfd ;Pushes EFLAG register.
push 1234H ;Pushes 1234H.
push dword [ebx] ;Pushes double word in data seg.
pushad ;eax, ecx, edx, ebx, esp, ebp, esi, edi
pop eax ;Pops 4 bytes.
```