Chapter 12 -- The Assembly Process

```
THE ASSEMBLY PROCESS
--------------------

 -- a computer understands machine code
 -- people (and compilers) write assembly language

  assembly     -----------------       machine
  source  -->  |  assembler    | -->   code
  code         -----------------

an assembler is a program -- a very deterministic program --
  it translates each instruction to its machine code.



  in the past, there was a one-to-one correspondence between
  assembly language instructions and  machine language instructions.

  this is no longer the case.  Assemblers are now-a-days made more
  powerful, and can "rework" code.

  The Pentium (being based on the 8086) has a one-to-one
  correspondence between assembly language instructions and
  machine language instructions.




ASSEMBLY
---------
 the assembler's job is to
   1. assign addresses
   2. generate machine code


 an assembler will

  -- assign addresses

  -- generate machine code

  -- generate an image of what memory must look like for the
     program to be executed.


 a simple assembler will make 2 complete passes over the data
 (source code) to complete this task.
    pass 1:  create complete SYMBOL TABLE
             generate machine code for instructions other than
               branches, jumps, call, lea, etc. (those instructions
               that rely on an address for their machine code).
    pass 2:  complete machine code for instructions that didn't get
             finished in pass 1.
```

```
assembler starts at the top of the source code program,
and SCANS.   It looks for
  -- directives   (.data  .code  .stack .486, etc. )
  -- instructions

  IMPORTANT:
  there are separate memory spaces for data and instructions.
  the assembler allocates them IN SEQENTIAL ORDER as it scans
  through the source code program.

  the starting addresses are fixed -- ANY program will be assembled
  to have data and instructions that start at the same address.


Generating Machine Code for an Instruction
------------------------------------------

This is complex due to the large variety of addressing
modes combined with the large number of instructions.

Most often, the machine code for an instruction consists of
  1) an 8-bit opcode
     (the choice of opcode will depend somewhat on the addressing
      modes used for the operands)
  followed by
  2) one or more bytes describing the addressing modes for
     the operands.


EXAMPLE INSTRUCTION:
    add eax, 24

    Find Appendix C.  That is where all this machine code
    stuff is specified.

    For the add instruction, the table lists:

    add   reg, r/m        03 /r
          r/m, reg        01 /r
          r/m, immed      81 /0 id

    The only one that would match the operand types is the
    third one in the list:
    add   r/m, immed      81 /0 id

    So, this is the one we choose.

    The 81 is the 8-bit opcode.

    What follows the opcode is information about the addressing
    mode of the 2 operands (add always has exactly 2 operands
    and the addressing mode of each must be explicitly specified)

    Commonly, both operands are described (exactly) by the
    encoding of a single byte that Intel calls the ModR/M byte.

    Within the machine code description (81 /0 id), the
    /0 symbol describes part of this ModR/M byte.
    /0  is found in the explanations table on page 352.
        It says the reg field of the ModR/M byte is 000.
```

```
The ModR/M byte:

  This byte describes the addressing mode of operands.

  It is divided up into 3 fields as follow


  BITS   7  6        5  4  3         2  1  0
         mod         reg/opcode       r/m

  For this example instruction, bits 5, 4, 3 are set to be 000,
  giving
  BITS   7  6        5  4  3         2  1  0
         mod         reg/opcode       r/m
                     0  0  0

  This tells that the second operand is an immediate.
  The description of the first operand will be done with
  the mode and r/m fields of the ModR/M byte.

  Look in the table (page 353) to find register mode,
  using register EAX (since that is what the example instruction has).
    Table says that Mod is  11, R/M is 000  giving

  BITS   7  6        5  4  3         2  1  0
         mod         reg/opcode       r/m
         1  1        0  0  0         0  0  0


  The last step is to get the id part.  From the explanation
  table (page 352), id is described as 32-bit immediate.
  Therefore id corresponds to a 32-bit two's complement representation
  of the value 24.

  This is 0000 0000 0000 0000 0000 0000 0001 1000
  In hex, this is 0x00000018.



Putting all this stuff together, we get machine code for
the example instruction    ( add  eax, 24 )

Note that everything is in hexadecimal.
AND, immediate values are listed least significant byte first!

   opcode
    81
   ModR/M byte
    c0
   immediate
    18
    00
    00
    00

Written out left to right:
  81 c0 18 00 00 00
```

```
One more example of generating machine code.

Machine code for the Pentium instruction

  dec  dword ptr [EDX]


  From page 349, we want the form of the decrement instruction

     dec  r/m               ff /1


  The opcode is ff, and it describes that there will be one operand,
  and it is of the general form.  The /1 says that the
  Reg field of the ModR/M byte will be 001.

  BITS   7  6        5  4  3          2  1  0
         Mod         Reg/Opcode        R/M
                     0  0  1

  The table on page 353 describes the Mod and R/M fields.
  Find the register direct addressing mode, using register EDX
  in the table.  It gives Mod 00 and R/M 010.

  BITS   7  6        5  4  3          2  1  0
         Mod         Reg/Opcode        R/M
         0  0        0  0  1          0  1  0

  In hex, this is 0a.

  The machine code is now complete:  ff 0a.



A BIG EXAMPLE:

 .data
a1  dd  4
a2  dd  ?
a3  dd  5 dup(0)

 .code
main:    mov  ecx, 20
         mov  eax, 15
         mov  edx, 0
         jz   target_label
loop1:   add  edx, eax
         imul [ebp + 8]
         dec  ecx
         jg   loop1
target_label:
         done



First step:  putting the data section together.

Upon scanning the source code, the token .data is read.
```

```
This is the directive that tells the assembler that what follows
gets allocated within the data section of the program.

  Remember that a directive is a "command" to the assembler
  about how to assemble the source code.

The next token encountered is  the label a1.

This symbol (label) is not yet in its symbol table, so the
assembler assigns an address, and places it in the symbol
table.

  Remember, the assembler assigns the first available address
  within the data section.


    Symbol table
    symbol      address
    ---------------------
    a1         0040 0000  (I made up this address, 'cuz we need
                           a starting address for the data section.)

The next token is dd.  It lets the assembler know to allocate
one doubleword of space at the current address.

The next token is 4.  It tells the assembler that the value of
the allocated space is to be the value 4.

The following line does much the same,
  placing a2 in the symbol table at the next available address
     (0x0040 0004)
  allocating 1 doubleword
  not putting something specific in the allocated space



When finished with the data section, we will have the

  symbol table
    symbol      address
    ---------------------
    a1         0040 0000
    a2         0040 0004
    a3         0040 0008
               0040 000c
               0040 0010
               0040 0014
               0040 0018

               0040 001c  (the next available address within the
                           data section.  NOT PART OF THE TABLE.)

and,

     memory map of data section
address      contents     notes
             hex
0040 0000    0000 0004    for a1
0040 0004    0000 0000    for a2 (defaults to 0)
0040 0008    0000 0000    5 double words for a3
0040 000c    0000 0000
```

```
0040 0010   0000 0000
0040 0014   0000 0000
0040 0018   0000 0000
```

Upon encounting the .code directive, the assembler knows that the next addresses it assigns will be within the code section of the program (separate from the data).

Assume that the code will be assembled such that the first instruction is placed at address 0x0000 0000.

The code (repeated):

```
 .code
main:    mov  ecx, 20
         mov  eax, 15
         mov  eax, 0
         jz   target_label
loop1:   add  edx, eax
         imul [ebp + 8]
         dec  ecx
         jg   loop1
target_label:
         done
```

The first token picked up after the .code directive is the label main. (As with ALL symbols,) the assembler looks to see if this symbol is already in the symbol table. It is not, so the assembler assigns the first available address, and places it in the symbol table.

```
  symbol table
    symbol      address
    ---------------------
    a1         0040 0000
    a2         0040 0004
    a3         0040 0008
               0040 000c
               0040 0010
               0040 0014
               0040 0018

               0040 001c  (the next available address within the
                           data section.  NOT PART OF THE TABLE.)
    main       0000 0000
```

Next, the assembler picks up the token mov. It knows that this is an instruction, and reads the rest of the instruction in order to generate the machine code for this instruction.

```
    mov  ecx, 20

    mov  reg, immed       b8 + rd
```

No ModR/M byte needed, since the register is incorporated into the opcode byte, and the immediate must follow.

```
    rd (from table on page 352) is 1,  b8+1=b9

    The immediate is 0x00000014.

    So, the machine code will be
     b9 14 00 00 00

    These 5 bytes are placed at address 0x0000 0000, and
    the next available address for an instruction becomes
    0x0000 0005.
```

The assembler is ready for the next token. It will be the second mov instruction in the program. It knows that this is an instruction, and reads the rest of the instruction in order to generate the machine code for this instruction.

```
    mov  eax, 15

    mov  reg, immed       b8 + rd

    No ModR/M byte needed, since the register is incorporated
    into the opcode byte, and the immediate must follow.

    rd (from table on page 352) is 0,  b8+0=b8

    The immediate is 0x0000000f.

    So, the machine code will be
     b8 10 00 00 00

    These 5 bytes are placed at address 0x0000 0005, and
    the next available address for an instruction becomes
    0x0000 000a.
```

The assembler is ready for the next token. It will be the third mov instruction in the program. It knows that this is an instruction, and reads the rest of the instruction in order to generate the machine code for this instruction.

```
    mov  edx, 0

    mov  reg, immed       b8 + rd

    No ModR/M byte needed, since the register is incorporated
    into the opcode byte, and the immediate must follow.

    rd (from table on page 352) is 2,  b8+2=ba

    The immediate is 0x00000000.

    So, the machine code will be
     ba 00 00 00 00

    These 5 bytes are placed at address 0x0000 000a, and
    the next available address for an instruction becomes
    0x0000 000f.
```

```
The assembler is ready for the next token.  It will be the
jz instruction in the program.  It knows that
this is an instruction, and reads the rest of the instruction
in order to generate the machine code for this instruction.

    jz   target_label

    jz   rel32                0f 84 "cd"

    The "cd" is a 32-bit code offset.  It needs to be the
    difference between what the PC will be when executing this
    code and the address assigned for label target_label.

    The problem with this is that target_label has not yet
    been assigned an address.  So, the assembler will need to
    wait on figuring out the 32-bit code offset portion of
    this instruction until the second pass of the assembler.

    The assembler does know that this instruction will be
    exactly 6 bytes long, so it can continue with assembly
    at location 0x0000 0015.

A memory map of text section so far is:

     memory map of text section
address      contents

0000 0000   b9 14 00 00 00
0000 0005   b8 0f 00 00 00
0000 000a   ba 00 00 00 00
0000 000f   0f 84 ?? ?? ?? ??
0000 0015



The assembler is ready for the next token.  It will be the
label loop1.  The assembler checks if this symbol is in the
symbol table.  It is not, so the assembler assigns an address
and places the symbol in the table.

  symbol table
    symbol      address
    ---------------------
    a1         0040 0000
    a2         0040 0004
    a3         0040 0008
               0040 000c
               0040 0010
               0040 0014
               0040 0018

               0040 001c  (the next available address within the
                           data section.  NOT PART OF THE TABLE.)
    main       0000 0000
    loop1      0000 0015



The assembler is ready for the next token.  It will be the
```

```
add instruction in the program.  It knows that
this is an instruction, and reads the rest of the instruction
in order to generate the machine code for this instruction.


         add  edx, eax

         add  reg, r/m   03 /r
         or
         add  r/m, reg   01 /r

         I doesn't matter which one is chosen.  They are the
         same length.  Chose the first one.

         /r means that the ModR/M byte has both a register
         operand and a R/M operand.

  BITS   7  6        5  4  3          2  1  0
         Mod         Reg/Opcode        R/M
         1  1        0  1  0          0  0  0

  In hex, this is d0.

  So, the machine code for the instruction is 03 d0.
  These 2 bytes are placed at address 0x0000 0015.
  The next available address for code will be 0x0000 0017.


A memory map of text section so far is:

     memory map of text section
address      contents

0000 0000   b9 14 00 00 00
0000 0005   b8 0f 00 00 00
0000 000a   ba 00 00 00 00
0000 000f   0f 84 ?? ?? ?? ??
0000 0015   03 d0
0000 0017




On to the next instruction.

         imul [ebp + 8]

         imul r/m      f7 /5

         /5 means that the ModR/M byte has a register
         field of 101

         The addressing mode for [ebp + 8] is under disp32[EBP]
         in the table on page 353.

  BITS   7  6        5  4  3          2  1  0
         Mod         Reg/Opcode        R/M
         1  0        1  0  1          1  0  1

  In hex, this is ad.
  The 32-bit displacement follows the ModR/M byte.  It contains
  a 32-bit 2's complement encoding of the value 8.
    0x 00 00 00 08

  The machine code for this instruction is f7 ad 08 00 00 00.
  These 6 bytes are placed at address 0x0000 0017.
  The next available address for code will be 0x0000 0019.

A memory map of text section so far is:

     memory map of code section
address      contents

0000 0000   b9 14 00 00 00
0000 0005   b8 10 00 00 00
0000 000a   ba 00 00 00 00
0000 000f   0f 84 ?? ?? ?? ??
0000 0015   03 d0
0000 0017   f7 ad 08 00 00 00
0000 001d



The next instruction is easy.

         dec  ecx

         dec  reg          48 + rd

    rd is 1 for ecx.  So the machine code is the single byte 49.

     memory map of code section
address      contents

0000 0000   b9 14 00 00 00
0000 0005   b8 10 00 00 00
0000 000a   ba 00 00 00 00
0000 000f   0f 84 ?? ?? ?? ??
0000 0015   03 d0
0000 0017   f7 ad 08 00 00 00
0000 001d   49
0000 001e




The decrement instruction is followed by

         jg   loop1

         jg   rel32            0f 8f "cd"


    Like the other control instruction:
    the "cd" is a 32-bit code offset.  It needs to be the
    difference between what the PC will be when executing this
    code and the address assigned for label target_label.

    The assembler does know that this instruction will be
    exactly 6 bytes long.
```

```
     To calculate "cd",

     at execution time (for taken control instruction):
      contents of PC + offset field  --> PC

      PC points to instruction after the control instruction
      when offset is added.


     at assembly time:

     byte offset = target addr - ( addr of instruction after conditional
                                   control instr addr )

                 = addr loop1  -  (6 + 0x0000 001e)
                  (taken from symbol table)

                 =  0x0000 0015 - 0x0000 0024

        Notice that this would be a negative number.  That is
        OK -- generate a 32-bit 2's complement value.

        0000 0000 0000 0000 0000 0000 0001 0101
      - 0000 0000 0000 0000 0000 0000 0010 0100
      ------------------------------------------
      becomes

        0000 0000 0000 0000 0000 0000 0001 0101
      + 1111 1111 1111 1111 1111 1111 1101 1100
      ------------------------------------------
        1111 1111 1111 1111 1111 1111 1111 0001

        in hex 0x ff ff ff f1



        this value is "cd",
        giving the machine code 0f 8f f1 ff ff ff
        (Remember that the least significant byte comes first.)


      Again,
      memory map of code section (so far)
 address      contents

 0000 0000   b9 14 00 00 00
 0000 0005   b8 10 00 00 00
 0000 000a   ba 00 00 00 00
 0000 000f   0f 84 ?? ?? ?? ??
 0000 0015   03 d0
 0000 0017   f7 ad 80 00 00 00
 0000 001d   49
 0000 001e   0f 8f f1 ff ff ff
 0000 0024



 The last thing we'll worry about in the table is
 the next label:  target_addr

 It gets placed in the symbol table at the next available
```

address, 0x0000 0024.

We now have a completed symbol table:

```
    symbol      address
    ---------------------
    a1         0040 0000
    a2         0040 0004
    a3         0040 0008
               0040 000c
               0040 0010
               0040 0014
               0040 0018

               0040 001c  (the next available address within the
                           data section.  NOT PART OF THE TABLE.)
    main       0000 0000
    loop1      0000 0015
    target_addr0000 0024
```

After this first pass of the assembler is done, ALL the labels have been given addresses.

During this second pass of the assembler, any remaining code left to be completed is completed. For this example code fragment, that is the jz instruction at address 0x0000 000f.

All that remains is the offset calculation. It works just like the calculation for the other control instruction.

```
    byte offset = target addr - ( addr of instruction after conditional
                                   control instr addr )

                = addr target_addr  -  (6 + 0x0000 000f)
                 (taken from symbol table)

                =  0x0000 0024 - 0x0000 0015

        0000 0000 0000 0000 0000 0000 0010 0100
      - 0000 0000 0000 0000 0000 0000 0001 0101
      -----------------------------------------
        0000 0000 0000 0000 0000 0000 0000 1111
```

Notice that this offset is a positive number. This is ok.
It corresponds to a branch/jump forward in the code.
A negative offset would correspond to a branch/jump backward within the code.

It is the offset of 0x 0000000f that gets placed into the machine code.

The completed machine code is

```
     memory map of text section
address      contents

0000 0000   b9 14 00 00 00
0000 0005   b8 10 00 00 00
```

```
0000 000a   ba 00 00 00 00
0000 000f   0f 84 0f 00 00 00
0000 0015   03 d0
0000 0017   f7 ad 80 00 00 00
0000 001d   49
0000 001e   0f 8f f1 ff ff ff
0000 0024
```