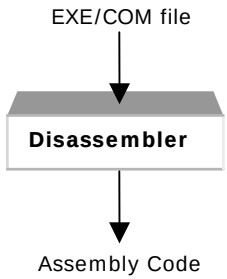
52 Writing Disassembler

"Blessed are the peacemakers."

Disassembler is the one which produces Assembly code for a given binary (EXE / COM)file. In this chapter let's see how to write a disassembler.

52.1 Prelude

We have already seen about assembler, linker and compiler. While we were discussing about decompilation (converting EXE file to C), we used disassembler to convert a binary file to assembly file. Thus disassembler provides a way to view the binary file with certain readability. In otherwords, disassembler can be used to read or edit a binary file in a better way.



Debugger is a tool to edit binary files. DOS's DEBUG is one such readily available Debugger. We also have other efficient Debuggers like TD (Turbo Debugger) etc. All debuggers use disassembler to provide assembly listing.

52.2 Secrets

In binary files the machine instructions are stored. Each binary code represents certain assembly instruction. So for writing disassembler, you need to know machine codes and corresponding assembly instructions. Disassembling is simply the reverse of assembling.

52.3 2asm

2asm is a disassembler utility that converts binary files to 80x86 assembler. The code was originally from the GNU C++ debugger, as ported to DOS by **DJ Delorie** and **Kent Williams**. Later **Robin Hilliard** modified it. This code was licensed under GNU's GPL. This disassembler is entirely table driven so one can easily change the instructions. When I checked this code it worked better than DOS's DEBUG. According to me it is really good as it uses tough logic.

The emulated coprocessor instructions on interrupts 34--3E are disassembled if the "-e" command line option is specified.

Command line switches (case sensitive):

-e : Disassemble (unoverridden) emulated 80*87 instructions (not default)
-3 : Assume code is 32 bit (default==16)
-x : Output all numbers in pure hex (no leading zeros or trailing "h"s.)

-s : Don't specify operand size (ie omit "byte ptr", "word ptr" and "dword ptr" from instruction output
-d : Don't specify distance of calls and jumps (near/far/short) (not default)

52.3.1 Table.c

Following is the table implementation for the disassembler. By the term table we mean array. It is wise to place the corresponding instructions in the array, so that we can fetch it for the given opcode.

```
/* Percent tokens in strings:
   First char after '%':
      A - direct address
      C - reg of r/m picks control register
      D - reg of r/m picks debug register
      E - r/m picks operand
      F - flags register
      G - reg of r/m picks general register
      I - immediate data
      J - relative IP offset
+       K - call/jmp distance
      M - r/m picks memory
      O - no r/m, offset only
      R - mod of r/m picks register only
      S - reg of r/m picks segment register
      T - reg of r/m picks test register
      X - DS:ESI
      Y - ES:EDI
      2 - prefix of two-byte opcode
+       e - put in 'e' if use32 (second char is part of reg name)
+           put in 'w' for use16 or 'd' for use32 (second char is 'w')
+       j - put in 'e' in jcxz if prefix==0x66
      f - floating point (second char is esc value)
      g - do r/m group 'n', n==0..7
      p - prefix
      s - size override (second char is a,o)
+       d - put d if double arg, nothing otherwise (pushfd, popfd &c)
+       w - put w if word, d if double arg, nothing otherwise
(lodsw/lodsd)
+       P - simple prefix

   Second char after '%':
      a - two words in memory (BOUND)
      b - byte
      c - byte or word
      d - dword
+       f - far call/jmp
```

```
+       n - near call/jmp
        p - 32 or 48 bit pointer
+       q - byte/word thingy
      s - six byte pseudo-descriptor
      v - word or dword
        w - word
+       x - sign extended byte
      F - use floating regs in mod/rm
      1-8 - group number, esc value, etc
*/

/* watch out for aad && aam with odd operands */


char *opmap1[256] = {
/* 0 */
  "add %Eb,%Gb",      "add %Ev,%Gv",     "add %Gb,%Eb",   "add %Gv,%Ev",
  "add al,%Ib",       "add %eax,%Iv",    "push es",        "pop es",
  "or %Eb,%Gb",       "or %Ev,%Gv",      "or %Gb,%Eb",     "or %Gv,%Ev",
  "or al,%Ib",        "or %eax,%Iv",     "push cs",        "%2 ",
/* 1 */
  "adc %Eb,%Gb",      "adc %Ev,%Gv",     "adc %Gb,%Eb",   "adc %Gv,%Ev",
  "adc al,%Ib",       "adc %eax,%Iv",    "push ss",        "pop ss",
  "sbb %Eb,%Gb",      "sbb %Ev,%Gv",     "sbb %Gb,%Eb",   "sbb %Gv,%Ev",
  "sbb al,%Ib",       "sbb %eax,%Iv",    "push ds",        "pop ds",
/* 2 */
  "and %Eb,%Gb",      "and %Ev,%Gv",     "and %Gb,%Eb",   "and %Gv,%Ev",
  "and al,%Ib",       "and %eax,%Iv",    "%pe",            "daa",
  "sub %Eb,%Gb",      "sub %Ev,%Gv",     "sub %Gb,%Eb",   "sub %Gv,%Ev",
  "sub al,%Ib",       "sub %eax,%Iv",    "%pc",            "das",
/* 3 */
  "xor %Eb,%Gb",      "xor %Ev,%Gv",     "xor %Gb,%Eb",   "xor %Gv,%Ev",
  "xor al,%Ib",       "xor %eax,%Iv",    "%ps",            "aaa",
  "cmp %Eb,%Gb",      "cmp %Ev,%Gv",     "cmp %Gb,%Eb",   "cmp %Gv,%Ev",
  "cmp al,%Ib",       "cmp %eax,%Iv",    "%pd",            "aas",
/* 4 */
  "inc %eax",         "inc %ecx",        "inc %edx",       "inc %ebx",
  "inc %esp",         "inc %ebp",        "inc %esi",       "inc %edi",
  "dec %eax",         "dec %ecx",        "dec %edx",       "dec %ebx",
  "dec %esp",         "dec %ebp",        "dec %esi",       "dec %edi",
/* 5 */
  "push %eax",        "push %ecx",       "push %edx",      "push %ebx",
  "push %esp",        "push %ebp",       "push %esi",      "push %edi",
  "pop %eax",         "pop %ecx",        "pop %edx",       "pop %ebx",
  "pop %esp",         "pop %ebp",        "pop %esi",       "pop %edi",
/* 6 */
  "pusha%d ",         "popa%d ",         "bound %Gv,%Ma","arpl %Ew,%Rw",
```

```
  "%pf",              "%pg",              "%so",              "%sa",
  "push %Iv",         "imul %Gv,%Ev,%Iv","push %Ix", "imul %Gv,%Ev,%Ib",
  "insb",             "ins%ew",           "outsb",            "outs%ew",
/* 7 */
  "jo %Jb",           "jno %Jb",          "jc %Jb",           "jnc %Jb",
  "je %Jb",           "jne %Jb",          "jbe %Jb",          "ja %Jb",
  "js %Jb",           "jns %Jb",          "jpe %Jb",          "jpo %Jb",
  "jl %Jb",           "jge %Jb",          "jle %Jb",          "jg %Jb",
/* 8 */
/*  "%g0 %Eb,%Ib",      "%g0 %Ev,%Iv",     "%g0 %Ev,%Ib", "%g0 %Ev,%Ib",
*/
  "%g0 %Eb,%Ib",      "%g0 %Ev,%Iv",      "%g0 %Ev,%Ix",    "%g0 %Ev,%Ix",
  "test %Eb,%Gb",     "test %Ev,%Gv",     "xchg %Eb,%Gb", "xchg %Ev,%Gv",
  "mov %Eb,%Gb",      "mov %Ev,%Gv",      "mov %Gb,%Eb",    "mov %Gv,%Ev",
  "mov %Ew,%Sw",      "lea %Gv,%M ",      "mov %Sw,%Ew",     "pop %Ev",
/* 9 */
  "nop",             "xchg %ecx,%eax","xchg %edx,%eax","xchg %ebx,%eax",
  "xchg %esp,%eax", "xchg %ebp,%eax","xchg %esi,%eax","xchg %edi,%eax",
  "cbw",              "cwd",              "call %Ap",         "fwait",
  "pushf%d ",         "popf%d ",          "sahf",             "lahf",
/* a */
  "mov al,%Oc",       "mov %eax,%Ov",     "mov %Oc,al",    "mov %Ov,%eax",
  "%P movsb",         "%P movs%w",        "%P cmpsb",         "%P cmps%w ",
  "test al,%Ib",      "test %eax,%Iv",    "%P stosb",         "%P stos%w ",
  "%P lodsb",         "%P lods%w ",       "%P scasb",         "%P scas%w ",
/* b */
  "mov al,%Ib",       "mov cl,%Ib",       "mov dl,%Ib",       "mov bl,%Ib",
  "mov ah,%Ib",       "mov ch,%Ib",       "mov dh,%Ib",       "mov bh,%Ib",
  "mov %eax,%Iv",     "mov %ecx,%Iv",     "mov %edx,%Iv", "mov %ebx,%Iv",
  "mov %esp,%Iv",     "mov %ebp,%Iv",     "mov %esi,%Iv", "mov %edi,%Iv",
/* c */
  "%g1 %Eb,%Ib",      "%g1 %Ev,%Ib",      "ret %Iw",          "ret",
  "les %Gv,%Mp",      "lds %Gv,%Mp",      "mov %Eb,%Ib",    "mov %Ev,%Iv",
  "enter %Iw,%Ib",    "leave",            "retf %Iw",         "retf",
  "int 03",           "int %Ib",          "into",             "iret",
/* d */
  "%g1 %Eb,1",        "%g1 %Ev,1",        "%g1 %Eb,cl",       "%g1 %Ev,cl",
  "aam ; %Ib",        "aad ; %Ib",        "setalc",           "xlat",
#if 0
  "esc 0,%Ib",        "esc 1,%Ib",        "esc 2,%Ib",        "esc 3,%Ib",
  "esc 4,%Ib",        "esc 5,%Ib",        "esc 6,%Ib",        "esc 7,%Ib",
#else
  "%f0",              "%f1",              "%f2",              "%f3",
  "%f4",              "%f5",              "%f6",              "%f7",
#endif
/* e */
  "loopne %Jb",       "loope %Jb",        "loop %Jb",       "j%j cxz %Jb",
```

```
  "in al,%Ib",          "in %eax,%Ib",      "out %Ib,al",     "out %Ib,%eax",
  "call %Jv",           "jmp %Jv",          "jmp %Ap",         "jmp %Ks%Jb",
  "in al,dx",           "in %eax,dx",       "out dx,al",      "out dx,%eax",
/* f */
  "lock %p ",           0,                  "repne %p ",       "repe %p ",
  "hlt",                "cmc",              "%g2",             "%g2",
  "clc",                "stc",              "cli",             "sti",
  "cld",                "std",              "%g3",             "%g4"
};


char *second[] = {
/* 0 */
  "%g5",                "%g6",              "lar %Gv,%Ew",    "lsl %Gv,%Ew",
  0,                    "loadall",          "clts",            "loadall",
  "invd",               "wbinvd",           0,                 0,
  0,                    0,                  0,                 0,
/* 1 */
  "mov %Eb,%Gb",        "mov %Ev,%Gv",      "mov %Gb,%Eb",    "mov %Gv,%Ev",
  0,                    0,                  0,                 0,
  0,                    0,                  0,                 0,
  0,                    0,                  0,                 0,
/* 2 */
  "mov %Rd,%Cd",        "mov %Rd,%Dd",      "mov %Cd,%Rd",    "mov %Dd,%Rd",
  "mov %Rd,%Td",        0,                  "mov %Td,%Rd",     0,
  0,                    0,                  0,                 0,
  0,                    0,                  0,                 0,
/* 3 */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* 4 */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* 5 */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* 6 */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* 7 */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* 8 */
  "jo %Jv",             "jno %Jv",          "jb %Jv",          "jnb %Jv",
  "jz %Jv",             "jnz %Jv",          "jbe %Jv",         "ja %Jv",
  "js %Jv",             "jns %Jv",          "jp %Jv",          "jnp %Jv",
  "jl %Jv",             "jge %Jv",          "jle %Jv",         "jg %Jv",
```

```
/* 9 */
  "seto %Eb",          "setno %Eb",        "setc %Eb",        "setnc %Eb",
  "setz %Eb",          "setnz %Eb",        "setbe %Eb",       "setnbe %Eb",
  "sets %Eb",          "setns %Eb",        "setp %Eb",        "setnp %Eb",
  "setl %Eb",          "setge %Eb",        "setle %Eb",       "setg %Eb",
/* a */
  "push fs",           "pop fs",           0,                 "bt %Ev,%Gv",
  "shld %Ev,%Gv,%Ib", "shld %Ev,%Gv,cl", 0,                 0,
  "push gs",           "pop gs",           0,                "bts %Ev,%Gv",
  "shrd %Ev,%Gv,%Ib", "shrd %Ev,%Gv,cl", 0,               "imul %Gv,%Ev",
/* b */
  "cmpxchg %Eb,%Gb",  "cmpxchg %Ev,%Gv", "lss %Mp",      "btr %Ev,%Gv",
  "lfs %Mp",           "lgs %Mp",         "movzx %Gv,%Eb","movzx %Gv,%Ew",
  0,                   0,                 "%g7 %Ev,%Ib",  "btc %Ev,%Gv",
  "bsf %Gv,%Ev",       "bsr %Gv,%Ev",     "movsx %Gv,%Eb","movsx %Gv,%Ew",
/* c */
  "xadd %Eb,%Gb",      "xadd %Ev,%Gv",    0,                 0,
  0,                   0,                 0,                 0,
  "bswap eax",         "bswap ecx",       "bswap edx",       "bswap ebx",
  "bswap esp",         "bswap ebp",       "bswap esi",       "bswap edi",
/* d */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* e */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
/* f */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
};

char *groups[][8] = {   /* group 0 is group 3 for %Ev set */
/* 0 */
  { "add",             "or",              "adc",             "sbb",
    "and",             "sub",             "xor",             "cmp"
},
/* 1 */
  { "rol",             "ror",             "rcl",             "rcr",
    "shl",             "shr",             "shl",             "sar"
},
/* 2 */  /* v   v*/
  { "test %Eq,%Iq",    "test %Eq,%Iq",    "not %Ev",         "neg %Ev",
    "mul %Ec",         "imul %Ec",        "div %Ec",         "idiv %Ec" },
/* 3 */
  { "inc %Eb",         "dec %Eb",         0,                 0,
    0,                 0,                 0,                 0
},
```

```
/* 4 */
  { "inc %Ev",          "dec %Ev",          "call %Kn%Ev",   "call %Kf%Ep",
    "jmp %Kn%Ev",       "jmp %Kf%Ep",       "push %Ev",        0
},
/* 5 */
  { "sldt %Ew",         "str %Ew",          "lldt %Ew",        "ltr %Ew",
    "verr %Ew",         "verw %Ew",         0,                 0
},
/* 6 */
  { "sgdt %Ms",         "sidt %Ms",         "lgdt %Ms",        "lidt %Ms",
    "smsw %Ew",         0,                  "lmsw %Ew",        0
},
/* 7 */
  { 0,                  0,                  0,                 0,
    "bt",               "bts",              "btr",             "btc"
}
};

/* zero here means invalid.  If first entry starts with '*', use st(i)
*/
/* no assumed %EFs here.  Indexed by RM(modrm())
*/
char *f0[]     = { 0, 0, 0, 0, 0, 0, 0, 0};
char *fop_9[]  = { "*fxch st,%GF" };
char *fop_10[] = { "fnop", 0, 0, 0, 0, 0, 0, 0 };
char *fop_12[] = { "fchs", "fabs", 0, 0, "ftst", "fxam", 0, 0 };
char *fop_13[] = { "fld1", "fldl2t", "fldl2e", "fldpi",
                   "fldlg2", "fldln2", "fldz", 0 };
char *fop_14[] = { "f2xm1", "fyl2x", "fptan", "fpatan",
                   "fxtract", "fprem1", "fdecstp", "fincstp" };
char *fop_15[] = { "fprem", "fyl2xp1", "fsqrt", "fsincos",
                   "frndint", "fscale", "fsin", "fcos" };
char *fop_21[] = { 0, "fucompp", 0, 0, 0, 0, 0, 0 };
char *fop_28[] = { 0, 0, "fclex", "finit", 0, 0, 0, 0 };
char *fop_32[] = { "*fadd %GF,st" };
char *fop_33[] = { "*fmul %GF,st" };
char *fop_36[] = { "*fsubr %GF,st" };
char *fop_37[] = { "*fsub %GF,st" };
char *fop_38[] = { "*fdivr %GF,st" };
char *fop_39[] = { "*fdiv %GF,st" };
char *fop_40[] = { "*ffree %GF" };
char *fop_42[] = { "*fst %GF" };
char *fop_43[] = { "*fstp %GF" };
char *fop_44[] = { "*fucom %GF" };
char *fop_45[] = { "*fucomp %GF" };
char *fop_48[] = { "*faddp %GF,st" };
char *fop_49[] = { "*fmulp %GF,st" };
```

```
char *fop_51[] = { 0, "fcompp", 0, 0, 0, 0, 0, 0 };
char *fop_52[] = { "*fsubrp %GF,st" };
char *fop_53[] = { "*fsubp %GF,st" };
char *fop_54[] = { "*fdivrp %GF,st" };
char *fop_55[] = { "*fdivp %GF,st" };
char *fop_60[] = { "fstsw ax", 0, 0, 0, 0, 0, 0, 0 };

char **fspecial[] = { /* 0=use st(i), 1=undefined 0 in fop_* means
undefined */
  0, 0, 0, 0, 0, 0, 0, 0,
  0, fop_9, fop_10, 0, fop_12, fop_13, fop_14, fop_15,
  f0, f0, f0, f0, f0, fop_21, f0, f0,
  f0, f0, f0, f0, fop_28, f0, f0, f0,
  fop_32, fop_33, f0, f0, fop_36, fop_37, fop_38, fop_39,
  fop_40, f0, fop_42, fop_43, fop_44, fop_45, f0, f0,
  fop_48, fop_49, f0, fop_51, fop_52, fop_53, fop_54, fop_55,
  f0, f0, f0, f0, fop_60, f0, f0, f0,
};

char *floatops[] = { /* assumed " %EF" at end of each.  mod != 3 only */
/*00*/ "fadd", "fmul", "fcom", "fcomp",
       "fsub", "fsubr", "fdiv", "fdivr",
/*08*/ "fld", 0, "fst", "fstp",
       "fldenv", "fldcw", "fstenv", "fstcw",
/*16*/ "fiadd", "fimul", "ficomw", "ficompw",
       "fisub", "fisubr", "fidiv", "fidivr",
/*24*/ "fild", 0, "fist", "fistp",
       "frstor", "fldt", 0, "fstpt",
/*32*/ "faddq", "fmulq", "fcomq", "fcompq",
       "fsubq", "fsubrq", "fdivq", "fdivrq",
/*40*/ "fldq", 0, "fstq", "fstpq",
       0, 0, "fsave", "fstsw",
/*48*/ "fiaddw", "fimulw", "ficomw", "ficompw",
       "fisubw", "fisubrw", "fidivw", "fidivr",
/*56*/ "fildw", 0, "fistw", "fistpw",
       "fbldt", "fildq", "fbstpt", "fistpq"
};
```

52.3.2 Disasm.c

Following is the main routine for the disassembler.

```
/* Code starts here...         */

#include <stdio.h>
#include <string.h>
#include <setjmp.h>
```

```
#include <stdlib.h>

typedef unsigned long word32;
typedef unsigned short word16;
typedef unsigned char word8;
typedef signed long int32;
typedef signed short int16;
typedef signed char int8;

typedef union {
  struct {
    word16 ofs;
    word16 seg;
  } w;
  word32 dword;
} WORD32;

/* variables controlled by command line flags */
static int8  seg_size=16;   /* default size is 16 */
static int8  do_hex = 0;    /* default is to use reassemblable
instructions */
static int8  do_distance = 1; /* default is to use reassemblable
instructions */
static word8 do_emul87 = 0; /* don't try to disassemble emulated
instrcutions */
static word8 do_size = 1;   /* default to outputting explicit operand
size */
static word8 must_do_size;  /* used with do_size */

static int wordop;           /* dealing with word or byte operand */
static FILE *infile;         /* input stream */
static word8 instruction_length;
static instruction_offset;
static word16 done_space; /* for opcodes with > one space */
static word8 patch87;  /*fudge variable used in 8087 emu patching code*/

static char ubuf[100], *ubufp;
static col;               /* output column */
static prefix;            /* segment override prefix byte */
static modrmv;            /* flag for getting modrm byte */
static sibv;              /* flag for getting sib byte   */
static opsize;            /* just like it says ...       */
static addrsize;
static jmp_buf reached_eof; /* jump back when reached eof */

/* some defines for extracting instruction bit fields from bytes */
```

```
#define MOD(a)      (((a)>>6)&7)
#define REG(a)      (((a)>>3)&7)
#define RM(a)       ((a)&7)
#define SCALE(a)  (((a)>>6)&7)
#define INDEX(a)  (((a)>>3)&7)
#define BASE(a)   ((a)&7)


extern char *opmap1[];      /* stuff from text.c */
extern char *second[];
extern char *groups[][8];
extern char *f0[];
extern char *fop_9[];
extern char *fop_10[];
extern char *fop_12[];
extern char *fop_13[];
extern char *fop_14[];
extern char *fop_15[];
extern char *fop_21[];
extern char *fop_28[];
extern char *fop_32[];
extern char *fop_33[];
extern char *fop_36[];
extern char *fop_37[];
extern char *fop_38[];
extern char *fop_39[];
extern char *fop_40[];
extern char *fop_42[];
extern char *fop_43[];
extern char *fop_44[];
extern char *fop_45[];
extern char *fop_48[];
extern char *fop_49[];
extern char *fop_51[];
extern char *fop_52[];
extern char *fop_53[];
extern char *fop_54[];
extern char *fop_55[];
extern char *fop_60[];
extern char **fspecial[];
extern char *floatops[];

/* prototypes */

static void ua_str(char *);
static word8 unassemble(word16);
static word8 getbyte(void);
```

```
static word8 silent_getbyte(void);
static word8 silent_returnbyte(word8 );
static modrm(void);
static sib(void);
static void uprintf(char *, ...);
static void uputchar(char );
static int bytes(char );
static void outhex(char , int , int , int , int );
static void reg_name(int , char );
static void do_sib(int );
static void do_modrm(char );
static void floating_point(int );
static void percent(char , char );

static char *addr_to_hex(int32 addr, char splitup)
{
  static char buffer[11];
  WORD32 adr;
  char hexstr[2];

  strcpy(hexstr, do_hex?"h":"");
  adr.dword = addr;
  if (splitup) {
    if (adr.w.seg==0 || adr.w.seg==0xffff) /* 'coz of wraparound */
      sprintf(buffer, "%04X%s", adr.w.ofs, hexstr);
    else
      sprintf(buffer, "%04X%s:%04X%s", adr.w.seg, hexstr, adr.w.ofs,
hexstr);
  } else {
    if (adr.w.seg==0 || adr.w.seg==0xffff) /* 'coz of wraparound */
      sprintf(buffer, "%04X%s", adr.w.ofs, hexstr);
    else
      sprintf(buffer, "%08lX%s", addr, hexstr);
  }
  return buffer;
}

static word8 getbyte(void)
{
  int16 c;

  c = fgetc(infile);
  if (c==EOF)
    longjmp(reached_eof, 1);
  printf("%02X", c);   /* print out byte */
  col+=2;
  if (patch87) {
```

```
    c -= 0x5C;     /* fixup second byte in emulated '87 instruction */
    patch87 = 0;
  }
  instruction_length++;
  instruction_offset++;
  return c;
}

/* used for lookahead */
static word8 silent_getbyte(void)
{
  return fgetc(infile);
}
/* return byte to input stream */
static word8 silent_returnbyte(word8 c)
{
  return ungetc(c, infile);
}

/*
   only one modrm or sib byte per instruction, tho' they need to be
   returned a few times...
*/

static modrm(void)
{
  if (modrmv == -1)
    modrmv = getbyte();
  return modrmv;
}

static sib(void)
{
  if (sibv == -1)
    sibv = getbyte();
  return sibv;
}

/*-------------------------------------------------------------------*/
static void uprintf(char *s, ...)
{
  vsprintf(ubufp, s, ...);
  while (*ubufp)
    ubufp++;
}
```

```
static void uputchar(char c)
{
  if (c == '\t') {
    if (done_space) {      /* don't tab out if already done so */
      uputchar(' ');
    } else {
      done_space = 1;
      do {
        *ubufp++ = ' ';
      } while ((ubufp-ubuf) % 8);
    }
  } else
    *ubufp++ = c;
  *ubufp = 0;
}

/*--------------------------------------------------------------------*/
static int bytes(char c)
{
  switch (c) {
  case 'b':
       return 1;
  case 'w':
       return 2;
  case 'd':
       return 4;
  case 'v':
       if (opsize == 32)
         return 4;
       else
         return 2;
  }
  return 0;
}

/*--------------------------------------------------------------------*/
static void outhex(char subtype, int extend, int optional, int defsize,
int sign)
{
  int n=0, s=0, i;
  int32 delta;
  unsigned char buff[6];
  char *name;
  char  signchar;

  switch (subtype) {
  case 'q':
```

```
        if (wordop) {
          if (opsize==16) {
            n = 2;
          } else {
            n = 4;
          }
        } else {
          n = 1;
        }
        break;

    case 'a':
        break;
    case 'x':
        extend = 2;
        n = 1;
        break;
    case 'b':
        n = 1;
        break;
    case 'w':
        n = 2;
        break;
    case 'd':
        n = 4;
        break;
    case 's':
        n = 6;
        break;
    case 'c':
    case 'v':
        if (defsize == 32)
          n = 4;
        else
          n = 2;
        break;
    case 'p':
        if (defsize == 32)
          n = 6;
        else
          n = 4;
        s = 1;
        break;
    }
    for (i=0; i<n; i++)
      buff[i] = getbyte();
    for (; i<extend; i++)
```

```
    buff[i] = (buff[i-1] & 0x80) ? 0xff : 0;
  if (s) {
    uprintf("%02X%02X:", buff[n-1], buff[n-2]);
    n -= 2;
  }
  switch (n) {
  case 1:
       delta = *(signed char *)buff;
       break;
  case 2:
       delta = *(signed int *)buff;
       break;
  case 4:
       delta = *(signed long *)buff;
       break;
  }
  if (extend > n) {
    if (subtype!='x') {
      if ((long)delta<0) {
        delta = -delta;
        signchar = '-';
      } else
        signchar = '+';
      if (delta || !optional)
        uprintf(do_hex?"%c%0*lX":"%c%0*lXh", signchar,
do_hex?extend:extend+1, delta);
    } else {
      if (extend==2)
        delta = (word16) delta;
      uprintf(do_hex?"%0.*lX":"%0.*lXh", 2*extend+1, delta);
/*      uprintf(do_hex?"%0.*lX":"%0.*lXh", 2*(do_hex?extend:extend+1),
delta); */
    }
    return;
  }
  if ((n == 4) && !sign) {
    name = addr_to_hex(delta, 0);
    uprintf("%s", name);
    return;
  }
  switch (n) {
  case 1:
       if (sign && (char)delta<0) {
         delta = -delta;
         signchar = '-';
       } else
         signchar = '+';
```

```
        if (sign)
          uprintf(do_hex?"%c%02X":"%c%03Xh",signchar,(unsigned
    char)delta);
        else
          uprintf(do_hex?"%02X":"%03Xh", (unsigned char)delta);
        break;

      case 2:
        if (sign && (int)delta<0) {
          signchar = '-';
          delta = -delta;
        } else
          signchar = '+';
        if (sign)
          uprintf(do_hex?"%c%04X":"%c%05Xh", signchar,(int)delta);
        else
          uprintf(do_hex?"%04X":"%05Xh", (unsigned int)delta);
        break;

      case 4:
        if (sign && (long)delta<0) {
          delta = -delta;
          signchar = '-';
        } else
          signchar = '+';
        if (sign)
          uprintf(do_hex?"%c%08X":"%c%09lXh", signchar, (unsigned
    long)delta);
        else
          uprintf(do_hex?"%08X":"%09lXh", (unsigned long)delta);
        break;
      }
    }

    /*-------------------------------------------------------------------*/
    static void reg_name(int regnum, char size)
    {
      if (size == 'F') { /* floating point register? */
        uprintf("st(%d)", regnum);
        return;
      }
      if (((size == 'v') && (opsize == 32)) || (size == 'd'))
        uputchar('e');
      if ((size=='q' || size == 'b' || size=='c') && !wordop) {
        uputchar("acdbacdb"[regnum]);
        uputchar("llllhhhh"[regnum]);
      } else {
```

```
    uputchar("acdbsbsd"[regnum]);
    uputchar("xxxxppii"[regnum]);
  }
}

/*---------------------------------------------------------------------*/
static void do_sib(int m)
{
  int s, i, b;

  s = SCALE(sib());
  i = INDEX(sib());
  b = BASE(sib());
  switch (b) {     /* pick base */
  case 0: ua_str("%p:[eax"); break;
  case 1: ua_str("%p:[ecx"); break;
  case 2: ua_str("%p:[edx"); break;
  case 3: ua_str("%p:[ebx"); break;
  case 4: ua_str("%p:[esp"); break;
  case 5:
       if (m == 0) {
         ua_str("%p:[");
         outhex('d', 4, 0, addrsize, 0);
       } else {
         ua_str("%p:[ebp");
       }
       break;
  case 6: ua_str("%p:[esi"); break;
  case 7: ua_str("%p:[edi"); break;
  }
  switch (i) {     /* and index */
  case 0: uprintf("+eax"); break;
  case 1: uprintf("+ecx"); break;
  case 2: uprintf("+edx"); break;
  case 3: uprintf("+ebx"); break;
  case 4: break;
  case 5: uprintf("+ebp"); break;
  case 6: uprintf("+esi"); break;
  case 7: uprintf("+edi"); break;
  }
  if (i != 4) {
    switch (s) {    /* and scale */
      case 0: uprintf(""); break;
      case 1: uprintf("*2"); break;
      case 2: uprintf("*4"); break;
```

```
      case 3: uprintf("*8"); break;
    }
  }
}

/*---------------------------------------------------------------------*/
static void do_modrm(char subtype)
{
  int mod = MOD(modrm());
  int rm = RM(modrm());
  int extend = (addrsize == 32) ? 4 : 2;

  if (mod == 3) { /* specifies two registers */
    reg_name(rm, subtype);
    return;
  }
  if (must_do_size) {
    if (wordop) {
      if (addrsize==32 || opsize==32) {   /* then must specify size */
        ua_str("dword ptr ");
      } else {
        ua_str("word ptr ");
      }
    } else {
      ua_str("byte ptr ");
    }
  }
  if ((mod == 0) && (rm == 5) && (addrsize == 32)) {/* mem operand with
32 bit ofs */
    ua_str("%p:[");
    outhex('d', extend, 0, addrsize, 0);
    uputchar(']');
    return;
  }
  if ((mod == 0) && (rm == 6) && (addrsize == 16)) { /*16 bit dsplcmnt*/
    ua_str("%p:[");
    outhex('w', extend, 0, addrsize, 0);
    uputchar(']');
    return;
  }
  if ((addrsize != 32) || (rm != 4))
    ua_str("%p:[");
  if (addrsize == 16) {
    switch (rm) {
    case 0: uprintf("bx+si"); break;
    case 1: uprintf("bx+di"); break;
    case 2: uprintf("bp+si"); break;
```

```
    case 3: uprintf("bp+di"); break;
    case 4: uprintf("si"); break;
    case 5: uprintf("di"); break;
    case 6: uprintf("bp"); break;
    case 7: uprintf("bx"); break;
    }
  } else {
    switch (rm) {
    case 0: uprintf("eax"); break;
    case 1: uprintf("ecx"); break;
    case 2: uprintf("edx"); break;
    case 3: uprintf("ebx"); break;
    case 4: do_sib(mod); break;
    case 5: uprintf("ebp"); break;
    case 6: uprintf("esi"); break;
    case 7: uprintf("edi"); break;
    }
  }
  switch (mod) {
  case 1:
       outhex('b', extend, 1, addrsize, 0);
       break;
  case 2:
       outhex('v', extend, 1, addrsize, 1);
       break;
  }
  uputchar(']');
}

/*---------------------------------------------------------------------*/
static void floating_point(int e1)
{
  int esc = e1*8 + REG(modrm());

  if (MOD(modrm()) == 3) {
    if (fspecial[esc]) {
      if (fspecial[esc][0][0] == '*') {
        ua_str(fspecial[esc][0]+1);
      } else {
        ua_str(fspecial[esc][RM(modrm())]);
      }
    } else {
      ua_str(floatops[esc]);
      ua_str(" %EF");
    }
  } else {
    ua_str(floatops[esc]);
```

```
    ua_str(" %EF");
  }
}

/*-------------------------------------------------------------------*/
/* Main table driver
*/
static void percent(char type, char subtype)
{
  int32 vofs;
  char *name;
  int extend = (addrsize == 32) ? 4 : 2;
  char c;

start:
  switch (type) {
  case 'A':                         /* direct address */
       outhex(subtype, extend, 0, addrsize, 0);
       break;

  case 'C':                         /* reg(r/m) picks control reg */
       uprintf("C%d", REG(modrm()));
       must_do_size = 0;
       break;

  case 'D':                         /* reg(r/m) picks debug reg */
       uprintf("D%d", REG(modrm()));
       must_do_size = 0;
       break;

  case 'E':                         /* r/m picks operand */
       do_modrm(subtype);
       break;

  case 'G':                         /* reg(r/m) picks register */
       if (subtype == 'F')               /* 80*87 operand?   */
         reg_name(RM(modrm()), subtype);
       else
         reg_name(REG(modrm()), subtype);
       must_do_size = 0;
       break;

  case 'I':                           /* immed data */
       outhex(subtype, 0, 0, opsize, 0);
       break;

  case 'J':                           /* relative IP offset */
```

```
     switch(bytes(subtype)) {              /* sizeof offset value */
     case 1:
          vofs = (int8)getbyte();
          break;
     case 2:
          vofs = getbyte();
          vofs += getbyte()<<8;
          vofs = (int16)vofs;
          break;
     case 4:
          vofs = (word32)getbyte();           /* yuk! */
          vofs |= (word32)getbyte() << 8;
          vofs |= (word32)getbyte() << 16;
          vofs |= (word32)getbyte() << 24;
          break;
     }
     name = addr_to_hex(vofs+instruction_offset,1);
     uprintf("%s", name);
     break;

case 'K':
     if (do_distance==0)
       break;
     switch (subtype) {
     case 'f':
          ua_str("far ");
          break;
     case 'n':
          ua_str("near ");
          break;
     case 's':
          ua_str("short ");
          break;
     }
     break;

case 'M':                              /* r/m picks memory */
     do_modrm(subtype);
     break;

case 'O':                              /* offset only */
     ua_str("%p:[");
     outhex(subtype, extend, 0, addrsize, 0);
     uputchar(']');
     break;

case 'P':                              /* prefix byte (rh) */
```

```
        ua_str("%p:");
        break;

    case 'R':                           /* mod(r/m) picks register */
        reg_name(REG(modrm()), subtype);      /* rh */
        must_do_size = 0;
        break;

    case 'S':                           /* reg(r/m) picks segment reg */
        uputchar("ecsdfg"[REG(modrm())]);
        uputchar('s');
        must_do_size = 0;
        break;

    case 'T':                           /* reg(r/m) picks T reg */
        uprintf("tr%d", REG(modrm()));
        must_do_size = 0;
        break;

    case 'X':                           /* ds:si type operator */
        uprintf("ds:[");
        if (addrsize == 32)
          uputchar('e');
        uprintf("si]");
        break;

    case 'Y':                           /* es:di type operator */
        uprintf("es:[");
        if (addrsize == 32)
          uputchar('e');
        uprintf("di]");
        break;

    case '2':                           /* old [pop cs]! now indexes */
        ua_str(second[getbyte()]);      /* instructions in 386/486   */
        break;

    case 'g':                          /* modrm group `subtype' (0--7) */
        ua_str(groups[subtype-'0'][REG(modrm())]);
        break;

    case 'd':                           /* sizeof operand==dword? */
        if (opsize == 32)
          uputchar('d');
        uputchar(subtype);
        break;
```

```
case 'w':                          /* insert explicit size specifier */
     if (opsize == 32)
       uputchar('d');
     else
       uputchar('w');
     uputchar(subtype);
     break;

case 'e':                           /* extended reg name */
     if (opsize == 32) {
       if (subtype == 'w')
         uputchar('d');
       else {
         uputchar('e');
         uputchar(subtype);
       }
     } else
       uputchar(subtype);
     break;

case 'f':                      /* '87 opcode */
     floating_point(subtype-'0');
     break;

case 'j':
     if (addrsize==32 || opsize==32) /* both of them?! */
       uputchar('e');
     break;

case 'p':                      /* prefix byte */
     switch (subtype)  {
     case 'c':
     case 'd':
     case 'e':
     case 'f':
     case 'g':
     case 's':
          prefix = subtype;
          c = getbyte();
          wordop = c & 1;
          ua_str(opmap1[c]);
          break;
     case ':':
          if (prefix)
            uprintf("%cs:", prefix);
          break;
     case ' ':
```

```
            c = getbyte();
            wordop = c & 1;
            ua_str(opmap1[c]);
            break;
        }
        break;

  case 's':                             /* size override */
        switch (subtype) {
        case 'a':
            addrsize = 48 - addrsize;
            c = getbyte();
            wordop = c & 1;
            ua_str(opmap1[c]);
/*            ua_str(opmap1[getbyte()]); */
            break;
        case 'o':
            opsize = 48 - opsize;
            c = getbyte();
            wordop = c & 1;
            ua_str(opmap1[c]);
/*            ua_str(opmap1[getbyte()]); */
            break;
        }
        break;
   }
}

static void ua_str(char *str)
{
  int c;

  if (str == 0) {
    uprintf("<invalid>");
    return;
  }
  if (strpbrk(str, "CDFGRST")) /* specifiers for registers=>no size 2b
specified */
    must_do_size = 0;
  while ((c = *str++) != 0) {
    if (c == '%') {
      c = *str++;
      percent(c, *str++);
    } else {
      if (c == ' ') {
        uputchar('\t');
      } else {
```

```
        uputchar(c);
      }
    }
  }
}

static word8 unassemble(word16 ofs)
{
  char *str;
  int   c, c2;

  printf("%04X ", ofs);
  prefix = 0;
  modrmv = sibv = -1;     /* set modrm and sib flags */
  opsize = addrsize = seg_size;
  col = 0;
  ubufp = ubuf;
  done_space = 0;
  instruction_length = 0;
  c = getbyte();
  wordop = c & 1;
  patch87 = 0;
  must_do_size = do_size;
  if (do_emul87) {
    if (c==0xcd) { /* wanna do emu '87 and ->ing to int? */
      c2 = silent_getbyte();
      if (c2 >= 0x34 && c2 <= 0x3E)
        patch87 = 1;       /* emulated instruction!  => must repatch two
bytes */
      silent_returnbyte(c2);
      c -= 0x32;
    }
  }
  if ((str = opmap1[c])==NULL) {    /* invalid instruction? */
    uputchar('d');                  /* then output byte defines */
    uputchar('b');
    uputchar('\t');
    uprintf(do_hex?"%02X":"%02Xh",c);
  } else {
    ua_str(str);                      /* valid instruction */
  }
  printf("%*s", 15-col, " ");
  col = 15 + strlen(ubuf);
  do {
    uputchar(' ');
    col++;
  } while (col < 43);
```

```
  printf("%s\n", ubuf);
  return instruction_length;
}

static word8 isoption(char c)
{
  return (c=='/' || c=='-');
}

void main(int argc, char *argv[])
{
  word16 instr_len;
  word16 offset;
  char infilename[80];
  char c;

#if defined(DEBUG)
  clrscr();
#endif

  *infilename = 0;
  while (--argc) {
    argv++;
    if (**argv=='?') {
hlp:  fprintf(stderr,
      "2ASM  Version 1.01  (C) Copyright 1992, Robin Hilliard\n"
      "Converts binary files to 80*86 assembly\n"
      "Usage:\n"
      "\t2asm <file> [-e] [-3] [-x] [-s] [-d]\n"
      "Switches:\n"
      "\t-e :\tDisassemble (unoverridden) emulated 80*87 instructions\n"
      "\t-3 :\tAssume code is 32 bit (default==16)\n"
      "\t-x :\tOutput numbers in pure hex (default is reassemblable)\n"
      "\t-s :\tDon't specify operand size, even where necessary\n"
      "\t-d :\tDon't specify distance short/near/far jmps and calls"
      );
      exit(1);
    }
    if (isoption(**argv)) {
      while (isoption(**argv)) {
nextflag:
        switch (c = *(++*argv)) {
        case 'e':
             do_emul87 = 1;
             break;
        case '3':
             seg_size = 32;
```

```
            break;
          case 'x':
            do_hex = 1;
            break;
          case 's':
            do_size = 0;
            break;
          case 'd':
            do_distance = 0;
            break;
          case '?':
          case 'H':
            goto hlp;
          case '#': /* hidden flag in the Loft's programs! */
            fprintf(stderr,"Last compiled on " __TIME__ ", " __DATE__);
            exit(1);
          default:
            fprintf(stderr, "Unknown option: `-%c'", c);
            exit(1);
          }
          ++*argv;
        }
      } else { /* assume that its a file name */
        if (*infilename) {
          fprintf(stderr,"Unknown file argument: \"%s\"", *argv);
          exit(1);
        }
        strcpy(infilename,*argv);
      }
    }
    if ((infile=fopen(infilename,"rb"))==NULL) {
      printf("Unable to open %s",infilename);
      exit(2);
    }
    offset = 0;
    strlwr(infilename);
    if (strstr(infilename, ".com")) /* not perfect, fix later */
      instruction_offset = offset = 0x100;
    if (!setjmp(reached_eof)) {
      do {
        instr_len = unassemble(offset);
        offset += instr_len;
      } while (instr_len); /* whoops, no files > 64k */
    }
}
```

52.3.3 2asm.prj

Add the above two programs: `Table.c` and `Disasm.c` in project file `2asm.prj` and compile. You will get an EXE file `2asm.exe` that you can use as disassembler.