Compilers and Compiler Generators © P.D. Terry, 2000

Appendix D

Source code for a macro assembler

This appendix gives the complete source code for the macro assembler for the single-accumulator machine discussed in Chapter 7.

assemble.cpp | misc.h | set.h | sh.s | sh.cpp | la.h | la.cpp | sa.h | sa.cpp | st.h | st.cpp | st.h | st.cpp | mh.h | mh.cpp | asmbase.h | as.h | as.cpp | mc.h | mc.cpp

```
----- assemble.cpp ------------------------------------------------------------

// Macro assembler/interpreter for the single-accumulator machine
// P.D. Terry,  Rhodes University, 1996

#include "mc.h"
#include "as.h"

#define version          "Macro Assembler 1.0"
#define usage            "Usage: ASSEMBLE source [listing]\n"

void main(int argc, char *argv[])
{ bool errors;
  char reply;
  char sourcename[256], listname[256];

  // check on correct parameter usage
  if (argc == 1) { printf(usage); exit(1); }
  strcpy(sourcename, argv[1]);
  if (argc > 2) strcpy(listname, argv[2]);
  else appendextension(sourcename, ".lst", listname);

  MC *Machine   = new(MC);
  AS *Assembler = new AS(sourcename, listname, version, Machine);
  Assembler->assemble(errors);
  if (errors)
  { printf("\nAssembly failed\n"); }
  else
  { printf("\nAssembly successful\n");
    while (true)
    { printf("\nInterpret? (y/n) ");
      do
      { scanf("%c", &reply);
      } while (toupper(reply) != ’N’ && toupper(reply) != ’Y’);
      if (toupper(reply) == ’N’) break;
      scanf("%*[^\n]"); getchar();
      Machine->interpret();
    }
  }
  delete Machine;
  delete Assembler;
}


----- misc.h ------------------------------------------------------------------

// Various common items for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef MISC_H
#define MISC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define  boolean  int
#define  bool     int
#define  true     1
```

```
#define  false    0
#define  TRUE     1
#define  FALSE    0
#define  maxint   INT_MAX

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  define  pathsep ’\\’
#else
#  define  pathsep ’/’
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{ int i;
  char old[256];
  strcpy(old, oldstr);
  i = strlen(old);
  while ((i > 0) && (old[i-1] != ’.’) && (old[i-1] != pathsep)) i--;
  if ((i > 0) && (old[i-1] == ’.’)) old[i-1] = 0;
  if (ext[0] == ’.’) sprintf(newstr,"%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

#define ASM_alength  8   // maximum length of mnemonics, labels
#define ASM_slength  35  // maximum length of comment and other strings

typedef char ASM_alfa[ASM_alength + 1];
typedef char ASM_strings[ASM_slength + 1];

#include "set.h"

enum ASM_errors {
  ASM_invalidcode, ASM_undefinedlabel, ASM_invalidaddress,
  ASM_unlabelled, ASM_hasaddress, ASM_noaddress,
  ASM_excessfields, ASM_mismatched, ASM_nonalpha,
  ASM_badlabel, ASM_invalidchar, ASM_invalidquote,
  ASM_overflow
};

typedef Set<ASM_overflow> ASM_errorset;

#endif /* MISC_H */


----- set.h -------------------------------------------------------------------

// Simple set operations

#ifndef SET_H
#define SET_H

template <int maxElem>
class Set {                         // { 0 .. maxElem }
  public:
    Set()                           // Construct { }
    { clear(); }

    Set(int e1)                     // Construct { e1 }
    { clear(); incl(e1); }

    Set(int e1, int e2)             // Construct { e1, e2 }
    { clear(); incl(e1); incl(e2); }

    Set(int e1, int e2, int e3)     // Construct { e1, e2, e3 }
    { clear(); incl(e1); incl(e2); incl(e3); }

    Set(int n, int e1[])            // Construct { e[0] .. e[n-1] }
    { clear(); for (int i = 0; i < n; i++) incl(e1[i]); }

    void incl(int e)                // Include e
    { if (e >= 0 && e <= maxElem) bits[wrd(e)] |= bitmask(e); }

    void excl(int e)                // Exclude e
    { if (e >= 0 && e <= maxElem) bits[wrd(e)] &= ~bitmask(e); }

    int memb(int e)                 // Test membership for e
    { if (e >= 0 && e <= maxElem) return((bits[wrd(e)] & bitmask(e)) != 0);
      else return 0;
    }

    int isempty(void)               // Test for empty set
    { for (int i = 0; i < length; i++) if (bits[i]) return 0;
      return 1;
    }

    Set operator + (const Set &s)   // Union with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] | s.bits[i];
      return r;
    }

    Set operator * (const Set &s)   // Intersection with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] & s.bits[i];
      return r;
    }

    Set operator - (const Set &s)   // Difference with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] & ~s.bits[i];
      return r;
    }

    Set operator / (const Set &s)   // Symmetric difference with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] ^ s.bits[i];
      return r;
    }

  private:
    unsigned char       bits[(maxElem + 8) / 8];
    int                 length;
    int wrd(int i)      { return(i / 8); }
    int bitmask(int i)  { return(1 << (i % 8)); }
    void clear()        { length = (maxElem + 8) / 8;
                          for (int i = 0; i < length; i++) bits[i] = 0;
                        }
};

#endif /* SET_H */


----- sh.s --------------------------------------------------------------------

// Source handler for assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef SH_H
#define SH_H

#include "misc.h"

const int linemax = 129;        // limit on source line length

class SH {
  public:
    FILE *lst; // listing file
    char ch;   // latest character read

    void nextch(void);
    // Returns ch as the next character on current source line, reading a new
    // line where necessary.  ch is returned as NUL if src is exhausted

    bool endline(void)             { return (charpos == linelength); }
    // Returns true when end of current line has been reached

    bool startline(void)           { return (charpos == 1); }
    // Returns true if current ch is the first on a line

    void writehex(int i, int n)    { fprintf(lst, "%02X%*c", i, n-2, ’ ’); }
    // Writes (byte valued) i to lst file as hex pair, left-justified in n spaces

    void writetext(char *s, int n) { fprintf(lst, "%-*s", n, s); }
    // Writes s to lst file left-justified in n spaces

    SH();
    // Default constructor

    SH(char *sourcename, char *listname, char *version);
    // Initializes source handler, and displays version information on lst file.
    // Opens src and lst files using given names

    ~SH();
    // Closes src and lst files

  private:
    FILE *src;              // source file
    int charpos;            // character pointer
    int linelength;         // line length
    char line[linemax + 1]; // last line read
};

#endif /*SH_H*/


----- sh.cpp -----------------------------------------------------------------

// Source handler for assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "sh.h"

void SH::nextch(void)
{ if (ch == ’\0’) return;        // input exhausted
  if (charpos == linelength)     // new line needed
  { linelength = 0; charpos = 0; ch = getc(src);
    while (ch != ’\n’ && !feof(src))
    { if (linelength < linemax) { line[linelength] = ch; linelength++; }
      ch = getc(src);
    }
    if (feof(src))
      line[linelength] = ’\0’;   // mark end with an explicit nul
    else
      line[linelength] = ’ ’;    // mark end with an explicit space
    linelength++;
  }
  ch = line[charpos]; charpos++; // pass back unique character
}

SH::SH(char *sourcename, char *listname, char *version)
{ src = fopen(sourcename, "r");
  if (src == NULL)
    { printf("Could not open input file\n"); exit(1); }
  lst = fopen(listname, "w");
  if (lst == NULL)
    { printf("Could not open listing file\n"); lst = stdout; }
  fprintf(lst, "%s\n\n", version);
  ch = ’ ’; charpos = 0; linelength = 0;
}

SH::SH()
{ src = NULL; lst = NULL; ch = ’ ’; charpos = 0; linelength = 0; }

SH::~SH()
{ if (src) fclose(src); src = NULL;
  if (lst) fclose(lst); lst = NULL;
}


----- la.h -------------------------------------------------------------------

// Lexical analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef LA_H
#define LA_H

#include "misc.h"
#include "sh.h"

enum LA_symtypes {
  LA_unknown, LA_eofsym, LA_eolsym, LA_idsym, LA_numsym, LA_comsym,
  LA_commasym, LA_plussym, LA_minussym, LA_starsym
};

struct LA_symbols {
  bool islabel;      // if in first column
  LA_symtypes sym;   // class
  ASM_strings str;   // lexeme
  int num;           // value if numeric
};

class LA {
  public:
    void getsym(LA_symbols &SYM, ASM_errorset &errors);
    // Returns the next symbol on current source line.
    // Adds to set of errors if necessary and returns SYM.sym = unknown
    // if no valid symbol can be recognized
    LA(SH *S);
    // Associates scanner with source handler S and initializes scanning

  private:
    SH *Srce;
    void getword(LA_symbols &SYM);
    void getnumber(LA_symbols &SYM, ASM_errorset &errors);
    void getcomment(LA_symbols &SYM);
    void getquotedchar(LA_symbols &SYM, char quote, ASM_errorset &errors);
};

#endif /*LA_H*/


----- la.cpp -----------------------------------------------------------------

// Lexical analyzer for assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "la.h"

void LA::getword(LA_symbols &SYM)
// Assemble identifier or opcode, in UPPERCASE for consistency
{ int length = 0;
  while (isalnum(Srce->ch))
  { if (length < ASM_slength)
    { SYM.str[length] = toupper(Srce->ch); length++; }
    Srce->nextch();
  }
  SYM.str[length] = ’\0’;
}

void LA::getnumber(LA_symbols &SYM, ASM_errorset &errors)
// Assemble number and store its identifier in UPPERCASE for consistency
{ int length = 0;
  while (isdigit(Srce->ch))
  { SYM.num = SYM.num * 10 + Srce->ch - ’0’;
    if (SYM.num > 255) errors.incl(ASM_overflow);
    SYM.num %= 256;
    if (length < ASM_slength) { SYM.str[length] = toupper(Srce->ch); length++; }
    Srce->nextch();
  }
  SYM.str[length] = ’\0’;
}

void LA::getcomment(LA_symbols &SYM)
// Assemble comment
{ int length = 0;
  while (!Srce->endline())
  { if (length < ASM_slength) { SYM.str[length] = Srce->ch; length++; }
    Srce->nextch();
  }
  SYM.str[length] = ’\0’;
}

void LA::getquotedchar(LA_symbols &SYM, char quote, ASM_errorset &errors)
// Assemble single character address token
{ SYM.str[0] = quote;
  Srce->nextch(); SYM.num = Srce->ch; SYM.str[1] = Srce->ch;
  if (!Srce->endline()) Srce->nextch();
  SYM.str[2] = Srce->ch; SYM.str[3] = ’\0’;
  if (Srce->ch != quote) errors.incl(ASM_invalidquote);
  if (!Srce->endline()) Srce->nextch();
}

void LA::getsym(LA_symbols &SYM, ASM_errorset &errors)
{ SYM.num = 0; SYM.str[0] = ’\0’;   // empty string
  while (Srce->ch == ’ ’ && !Srce->endline()) Srce->nextch();
  SYM.islabel = (Srce->startline() && Srce->ch != ’ ’
                 && Srce->ch != ’;’ && Srce->ch != ’\0’);
  if (SYM.islabel && !isalpha(Srce->ch)) errors.incl(ASM_badlabel);
  if (Srce->ch == ’\0’) { SYM.sym = LA_eofsym; return; }
  if (Srce->endline()) { SYM.sym = LA_eolsym; Srce->nextch(); return; }
  if (isalpha(Srce->ch))
    { SYM.sym = LA_idsym; getword(SYM); }
  else if (isdigit(Srce->ch))
    { SYM.sym = LA_numsym; getnumber(SYM, errors); }
  else switch (Srce->ch)
    { case ’;’:
        SYM.sym = LA_comsym; getcomment(SYM); break;
      case ’,’:
        SYM.sym = LA_commasym; strcpy(SYM.str, ","); Srce->nextch(); break;
      case ’+’:
        SYM.sym = LA_plussym; strcpy(SYM.str, "+"); Srce->nextch(); break;
      case ’-’:
        SYM.sym = LA_minussym; strcpy(SYM.str, "-"); Srce->nextch(); break;
      case ’*’:
        SYM.sym = LA_starsym; strcpy(SYM.str, "*"); Srce->nextch(); break;
      case ’\’’:
      case ’"’:
        SYM.sym = LA_numsym; getquotedchar(SYM, Srce->ch, errors); break;
      default:
        SYM.sym = LA_unknown; getcomment(SYM); errors.incl(ASM_invalidchar);
        break;
    }
}

LA::LA(SH* S)
{ Srce = S; Srce->nextch(); }


----- sa.h -------------------------------------------------------------------

// Syntax analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef SA_H
#define SA_H

#include "misc.h"
#include "la.h"

const int SA_maxterms = 16;

enum SA_termkinds {
  SA_absent, SA_numeric, SA_alphameric, SA_comma, SA_plus, SA_minus, SA_star
};

struct SA_terms {
  SA_termkinds kind;
  int number;      // value if known
  ASM_alfa name;   // character representation
};

struct SA_addresses {
  char length;     // number of fields
  SA_terms term[SA_maxterms - 1];
};

struct SA_unpackedlines {
  // source text, unpacked into fields
  bool labelled;
  ASM_alfa labfield, mnemonic;
  SA_addresses address;
  ASM_strings comment;
  ASM_errorset errors;
};

class SA {
  public:
    void parse(SA_unpackedlines &srcline);
    // Analyzes the next source line into constituent fields

    SA(LA *L);
    // Associates syntax analyzer with its lexical analyzer L

  private:
    LA *Lex;
    LA_symbols SYM;
    void GetSym(ASM_errorset &errors);
    void getaddress(SA_unpackedlines &srcline);
};

#endif /*SA_H*/


----- sa.cpp -----------------------------------------------------------------

// Syntax analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "sa.h"
#include "set.h"

typedef Set<LA_starsym> symset;
```

```
void SA::GetSym(ASM_errorset &errors)
{ Lex->getsym(SYM, errors); }

void SA::getaddress(SA_unpackedlines &srcline)
// Unpack the addressfield of line into srcline
{ symset allowed(LA_idsym, LA_numsym, LA_starsym);
  symset possible = allowed + symset(LA_commasym, LA_plussym, LA_minussym);
  srcline.address.length = 0;
  while (possible.memb(SYM.sym))
  { if (!allowed.memb(SYM.sym))
      srcline.errors.incl(ASM_invalidaddress);
    if (srcline.address.length < SA_maxterms - 1)
      srcline.address.length++;
    else
      srcline.errors.incl(ASM_excessfields);
    sprintf(srcline.address.term[srcline.address.length - 1].name, "%.*s",
            ASM_alength, SYM.str);
    srcline.address.term[srcline.address.length - 1].number = SYM.num;
    switch (SYM.sym)
    { case LA_numsym:
        srcline.address.term[srcline.address.length - 1].kind = SA_numeric;
        break;
      case LA_idsym:
        srcline.address.term[srcline.address.length - 1].kind = SA_alphameric;
        break;
      case LA_plussym:
        srcline.address.term[srcline.address.length - 1].kind = SA_plus;
        break;
      case LA_minussym:
        srcline.address.term[srcline.address.length - 1].kind = SA_minus;
        break;
      case LA_starsym:
        srcline.address.term[srcline.address.length - 1].kind = SA_star;
        break;
      case LA_commasym:
        srcline.address.term[srcline.address.length - 1].kind = SA_comma;
        break;
    }
    allowed = possible - allowed;
    GetSym(srcline.errors); // check trailing comment, parameters
  }
  if (!(srcline.address.length & 1)) srcline.errors.incl(ASM_invalidaddress);
}

void SA::parse(SA_unpackedlines &srcline)
{ symset startaddress(LA_idsym, LA_numsym, LA_starsym);
  srcline.labfield[0] = ’\0’;
  strcpy(srcline.mnemonic, "   ");
  srcline.comment[0] = ’\0’;
  srcline.errors = ASM_errorset();
  srcline.address.term[0].kind = SA_absent;
  srcline.address.term[0].number = 0;
  srcline.address.term[0].name[0] = ’\0’;
  srcline.address.length = 0;
  GetSym(srcline.errors);       // first on line - opcode or label ?
  if (SYM.sym == LA_eofsym) { strcpy(srcline.mnemonic, "END"); return; }
  srcline.labelled = SYM.islabel;
  if (srcline.labelled)         // must look for the opcode
  { srcline.labelled = srcline.errors.isempty();
    sprintf(srcline.labfield, "%.*s", ASM_alength, SYM.str);
    GetSym(srcline.errors);     // probably an opcode
  }
  if (SYM.sym == LA_idsym)      // has a mnemonic
  { sprintf(srcline.mnemonic, "%.*s", ASM_alength, SYM.str);
    GetSym(srcline.errors);     // possibly an address
    if (startaddress.memb(SYM.sym)) getaddress(srcline);
  }
  if (SYM.sym == LA_comsym || SYM.sym == LA_unknown)
  { strcpy(srcline.comment, SYM.str); GetSym(srcline.errors); }
  if (SYM.sym != LA_eolsym)     // spurious symbol
  { strcpy(srcline.comment, SYM.str); srcline.errors.incl(ASM_excessfields); }
  while (SYM.sym != LA_eolsym && SYM.sym != LA_eofsym)
    GetSym(srcline.errors);     // consume garbage
}

SA::SA(LA * L)
{ Lex = L; }


----- st.h ------------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using simple linked list
// P.D. Terry, Rhodes University, 1996

#ifndef ST_H
#define ST_H

#include "misc.h"
#include "mc.h"
#include "sh.h"

enum ST_actions { ST_add, ST_subtract };

typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v, ST_actions a);

struct ST_forwardrefs {   // forward references for undefined labels
  MC_bytes byte;          // to be patched
  ST_actions action;      // taken when patching
  ST_forwardrefs *nlink;  // to next reference
};

struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool defined;           // true after defining occurrence encountered
  ST_entries *slink;      // to next entry
  ST_forwardrefs *flink;  // to forward references
};

class ST {
  public:
    void printsymboltable(bool &errors);
    // Summarizes symbol table at end of assembly, and alters errors to true if
    // any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // Records action to be applied later in fixing up forward references.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes *mem, ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)

  private:
    SH *Srce;
    ST_entries *lastsym;
    void findentry(ST_entries *&symentry, char *name, bool &found);
};

#endif /*ST_H*/


----- st.cpp ---------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using simply linked list
// P.D. Terry, Rhodes University, 1996

#include "st.h"

void ST::printsymboltable(bool &errors)
{ fprintf(Srce->lst, "\nSymbol Table\n");
  fprintf(Srce->lst, "------------\n");
  ST_entries *symentry = lastsym;
  while (symentry)
  { Srce->writetext(symentry->name, 10);
    if (!symentry->defined)
    { fprintf(Srce->lst, " --- undefined"); errors = true; }
    else
    { Srce->writehex(symentry->value, 3);
      fprintf(Srce->lst, "%5d", symentry->value);
    }
    putc('\n', Srce->lst);
    symentry = symentry->slink;
  }
  putc('\n', Srce->lst);
}
```

```
void ST::findentry(ST_entries *&symentry, char *name, bool &found)
{ symentry = lastsym;
  found = false;
  while (!found && symentry)
  { if (!strcmp(name, symentry->name))
      found = true;
    else
      symentry = symentry->slink;
  }
  if (found) return;
  symentry = new ST_entries;  // make new forward reference entry
  sprintf(symentry->name, "%.*s", ASM_alength, name);
  symentry->value = 0;
  symentry->defined = false;
  symentry->flink = NULL;
  symentry->slink = lastsym;
  lastsym = symentry;
}

void ST::enter(char *name, MC_bytes value)
{ ST_entries *symentry;
  bool found;
  findentry(symentry, name, found);
  symentry->value = value;
  symentry->defined = true;
}

void ST::valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined)
{ ST_entries *symentry;
  ST_forwardrefs *forwardentry;
  bool found;
  findentry(symentry, name, found);
  value = symentry->value;
  undefined = !symentry->defined;
  if (!undefined) return;
  forwardentry = new ST_forwardrefs; // new node in reference chain
  forwardentry->byte = location; forwardentry->action = action;
  if (found)                         // it was already in the table
    forwardentry->nlink = symentry->flink;
  else                               // new entry in the table
    forwardentry->nlink = NULL;
  symentry->flink = forwardentry;
}

void ST::outstandingreferences(MC_bytes mem[], ST_patch fix)
{ ST_forwardrefs *link;
  ST_entries *symentry = lastsym;
  while (symentry)
  { link = symentry->flink;
    while (link)
    { fix(mem, link->byte, symentry->value, link->action);
      link = link->nlink;
    }
    symentry = symentry->slink;
  }
}

ST::ST(SH *S)
{ Srce = S; lastsym = NULL; }


----- st.h --------------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using hashing technique with collision stepping
// P.D. Terry, Rhodes University, 1996

#ifndef ST_H
#define ST_H

#include "misc.h"
#include "mc.h"
#include "sh.h"

const int tablemax = 239;   // symbol table size
const int tablestep = 7;    // a prime number

enum ST_actions { ST_add, ST_subtract };

typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v, ST_actions a);
typedef short tableindex;
```

```
struct ST_forwardrefs {  // forward references for undefined labels
  MC_bytes byte;         // to be patched
  ST_actions action;     // taken when patching
  ST_forwardrefs *nlink; // to next reference
};

struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool used;              // true when in use already
  bool defined;           // true after defining occurrence encountered
  ST_forwardrefs *flink;  // to forward references
};

class ST {
  public:
    void printsymboltable(bool &errors);
    // Summarizes symbol table at end of assembly, and alters errors
    // to true if any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // Records action to be applied later in fixing up forward references.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes mem[], ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)

  private:
    SH *Srce;
    ST_entries hashtable[tablemax + 1];
    void findentry(tableindex &symentry, char *name, bool &found);
};

#endif /*ST_H*/


----- st.cpp -----------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using hashing technique with collision stepping
// P.D. Terry, Rhodes University, 1996

#include "st.h"

void ST::printsymboltable(bool &errors)
{ fprintf(Srce->lst, "\nSymbol Table\n");
  fprintf(Srce->lst, "------------\n");
  for (tableindex i = 0; i < tablemax; i++)
  { if (hashtable[i].used)
    { Srce->writetext(hashtable[i].name, 10);
      if (!hashtable[i].defined)
      { fprintf(Srce->lst, " --- undefined"); errors = true; }
      else
      { Srce->writehex(hashtable[i].value, 3);
        fprintf(Srce->lst, "%5d", hashtable[i].value);
      }
      putc('\n', Srce->lst);
    }
  }
  putc('\n', Srce->lst);
}

tableindex hashkey(char *ident)
{ const int large = (maxint - 256); // large number in hashing function
  int sum = 0, l = strlen(ident);
  for (int i = 0; i < l; i++) sum = (sum + ident[i]) % large;
  return (sum % tablemax);
}

void ST::findentry(tableindex &symentry, char *name, bool &found)
{ enum { looking, entered, caninsert, overflow } state;
  symentry = hashkey(name);
  state = looking;
  tableindex start = symentry;
  while (state == looking)
  { if (!hashtable[symentry].used)
      { state = caninsert; break; }
    if (!strcmp(name, hashtable[symentry].name))
      { state = entered; break; }
    symentry = (symentry + tablestep) % tablemax;
    if (symentry == start) state = overflow;
  }
  switch (state)
  { case caninsert:
      sprintf(hashtable[symentry].name, "%.*s", ASM_alength, name);
      hashtable[symentry].value = 0;
      hashtable[symentry].used = true;
      hashtable[symentry].flink = NULL;
      hashtable[symentry].defined = false;
      break;
    case overflow:
      printf("Symbol table overflow\n");
      exit(1);
      break;
    case entered:   // no further action
      break;
  }
  found = (state == entered);
}

void ST::enter(char *name, MC_bytes value)
{ tableindex symentry;
  bool found;
  findentry(symentry, name, found);
  hashtable[symentry].value = value;
  hashtable[symentry].defined = true;
}

void ST::valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined)
{ tableindex symentry;
  ST_forwardrefs *forwardentry;
  bool found;
  findentry(symentry, name, found);
  value = hashtable[symentry].value;
  undefined = !hashtable[symentry].defined;
  if (!undefined) return;
  forwardentry = new ST_forwardrefs; // new node in reference chain
  forwardentry->byte = location;
  forwardentry->action = action;
  if (found)                         // it was already in the table
    forwardentry->nlink = hashtable[symentry].flink;
  else                               // new entry in the table
    forwardentry->nlink = NULL;
  hashtable[symentry].flink = forwardentry;
}

void ST::outstandingreferences(MC_bytes mem[], ST_patch fix)
{ ST_forwardrefs *link;
  for (tableindex i = 0; i < tablemax; i++)
  { if (hashtable[i].used)
    { link = hashtable[i].flink;
      while (link)
      { fix(mem, link->byte, hashtable[i].value, link->action);
        link = link->nlink;
      }
    }
  }
}

ST::ST(SH *S)
{ Srce = S;
  for (tableindex i = 0; i < tablemax; i++) hashtable[i].used = false;
}


----- mh.h --------------------------------------------------------------------

// Macro analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef MH_H
#define MH_H

#include "asmbase.h"

typedef struct MH_macentries *MH_macro;
```

```
class MH {
  public:
    void newmacro(MH_macro &m, SA_unpackedlines header);
    // Creates m as a new macro, with given header line that includes the
    // formal parameters

    void storeline(MH_macro m, SA_unpackedlines line);
    // Adds line to the definition of macro m

    void checkmacro(char *name, MH_macro &m, bool &ismacro, int &params);
    // Checks to see whether name is that of a predefined macro.  Returns
    // ismacro as the result of the search.  If successful, returns m as
    // the macro, and params as the number of formal parameters

    void expand(MH_macro m, SA_addresses actualparams,
                ASMBASE *assembler, bool &errors);
    // Expands macro m by invoking assembler for each line of the macro
    // definition, and using the actualparams supplied in place of the
    // formal parameters appearing in the macro header.
    // errors is altered to true if the assembly fails for any reason

    MH();
    // Initializes macro handler

  private:
    MH_macro lastmac;
    int position(MH_macro m, char *str);
    void substituteactualparameters(MH_macro m,
                                    SA_addresses actualparams,
                                    SA_unpackedlines &nextline);
};

#endif /*MH_H*/


----- mh.cpp ------------------------------------------------------------------

// Macro analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "mh.h"

struct MH_lines {
  SA_unpackedlines text;          // a single line of macro text
  MH_lines *link;                 // link to the next line in the macro
};

struct MH_macentries {
  SA_unpackedlines definition;    // header line
  MH_macro mlink;                 // link to next macro in list
  MH_lines *firstline, *lastline; // links to the text of this macro
};

void MH::newmacro(MH_macro &m, SA_unpackedlines header)
{ m = new MH_macentries;
  m->definition = header;         // store formal parameters
  m->firstline = NULL;            // no text yet
  m->mlink = lastmac;             // link to rest of macro definitions
  lastmac = m;                    // and this becomes the last macro added
}

void MH::storeline(MH_macro m, SA_unpackedlines line)
{ MH_lines *newline = new MH_lines;
  newline->text = line;           // store source line
  newline->link = NULL;           // at the end of the queue
  if (m->firstline == NULL)       // first line of macro?
    m->firstline = newline;       // form head of new queue
  else
    m->lastline->link = newline;  // add to tail of existing queue
  m->lastline = newline;
}

void MH::checkmacro(char *name, MH_macro &m, bool &ismacro, int &params)
{ m = lastmac; ismacro = false; params = 0;
  while (m && !ismacro)
  { if (!strcmp(name, m->definition.labfield))
      { ismacro = true; params = m->definition.address.length; }
    else
      m = m->mlink;
  }
}
```

```
int MH::position(MH_macro m, char *str)
// Search formals for match to str; returns 0 if no match
{ bool found = false;
  int i = m->definition.address.length - 1;
  while (i >= 0 && !found)
  { if (!strcmp(str, m->definition.address.term[i].name))
      found = true;
    else
      i--;
  }
  return i;
}

void MH::substituteactualparameters(MH_macro m,
         SA_addresses actualparams, SA_unpackedlines &nextline)
// Substitute label, mnemonic or address components into
// nextline where necessary
{ int j = 0, i = position(m, nextline.labfield); // check label
  if (i >= 0) strcpy(nextline.labfield, actualparams.term[i].name);
  i = position(m, nextline.mnemonic);            // check mnemonic
  if (i >= 0) strcpy(nextline.mnemonic, actualparams.term[i].name);
  j = 0;                                         // check address fields
  while (j < nextline.address.length)
  { i = position(m, nextline.address.term[j].name);
    if (i >= 0) nextline.address.term[j] = actualparams.term[i];
    j += 2;                                      // bypass commas
  }
}

void MH::expand(MH_macro m, SA_addresses actualparams,
                ASMBASE *assembler, bool &errors)
{ SA_unpackedlines nextline;
  if (!m) return;                                // nothing to do
  MH_lines *current = m->firstline;
  while (current)
  { nextline = current->text;                    // retrieve line of macro text
    substituteactualparameters(m, actualparams, nextline);
    assembler->assembleline(nextline, errors);   // and asssemble it
    current = current->link;
  }
}

MH::MH()
{ lastmac = NULL; }


----- asmbase.h ---------------------------------------------------------------

// Base assembler class for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef ASMBASE_H
#define ASMBASE_H

#include "misc.h"
#include "sa.h"

class ASMBASE {
  public:
    virtual void assembleline(SA_unpackedlines &srcline, bool &failure) = 0;
    // Assemble srcline, reporting failure if it occurs
};

#endif /*A_H*/


----- as.h ---------------------------------------------------------------------

// One-pass macro assembler for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef AS_H
#define AS_H

#include "asmbase.h"
#include "mc.h"
#include "st.h"
#include "sh.h"
#include "mh.h"

class AS : ASMBASE {
  public:
    void assemble(bool &errors);
    // Assembles and lists program.
    // Assembled code is dumped to file for later interpretation, and left
    // in pseudo-machine memory for immediate interpretation if desired.
    // Returns errors = true if assembly fails

    virtual void assembleline(SA_unpackedlines &srcline, bool &failure);
    // Assemble srcline, reporting failure if it occurs

    AS(char *sourcename, char *listname, char *version, MC *M);
    // Instantiates version of the assembler to process sourcename, creating
    // listings in listname, and generating code for associated machine M

  private:
    SH *Srce;
    LA *Lex;
    SA *Parser;
    ST *Table;
    MC *Machine;
    MH *Macro;

    struct { ASM_alfa spelling; MC_bytes byte; } optable[256];
    int opcodes;         // number of opcodes actually defined
    struct objlines { MC_bytes location, opcode, address; };
    objlines objline;    // current line as assembled
    MC_bytes location;   // location counter
    bool assembling;     // monitor progress of assembly
    bool include;        // handle conditional assembly

    MC_bytes bytevalue(char *mnemonic);
    void enter(char *mnemonic, MC_bytes thiscode);
    void termvalue(SA_terms term, MC_bytes &value, ST_actions action,
                   bool &undefined, bool &badaddress);
    void evaluate(SA_addresses address, MC_bytes &value,
                  bool &undefined, bool &malformed);

    void listerrors(ASM_errorset allerrors, bool &failure);
    void listcode(void);
    void listsourceline(SA_unpackedlines &srcline, bool coderequired,
                        bool &failure);
    void definemacro(SA_unpackedlines &srcline, bool &failure);
    void firstpass(bool &errors);
};

#endif /*AS_H*/


----- as.cpp ------------------------------------------------------------------

// One-pass macro assembler for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "as.h"

const bool nocodelisted = false;
const bool codelisted = true;

enum directives {
  AS_err = 61,  // erroneous opcode
  AS_nul = 62,  // blank opcode
  AS_beg = 63,  // introduce program
  AS_end = 64,  // end of source
  AS_mac = 65,  // introduce macro
  AS_ds  = 66,  // define storage
  AS_equ = 67,  // equate
  AS_org = 68,  // set location counter
  AS_if  = 69,  // conditional
  AS_dc  = 70   // define constant byte
};

MC_bytes AS::bytevalue(char *mnemonic)
{ int look, l = 1, r = opcodes;
  do                                // binary search
  { look = (l + r) / 2;
    if (strcmp(mnemonic, optable[look].spelling) <= 0) r = look - 1;
    if (strcmp(mnemonic, optable[look].spelling) >= 0) l = look + 1;
  } while (l <= r);
  if (l > r + 1)
    return (optable[look].byte);  // found it
  else
    return (optable[0].byte);     // err entry
}

void AS::enter(char *mnemonic, MC_bytes thiscode)
```

```
// Add (mnemonic, thiscode) to optable for future look up
{ strcpy(optable[opcodes].spelling, mnemonic);
  optable[opcodes].byte = thiscode;
  opcodes++;
}

void backpatch(MC_bytes mem[], MC_bytes location, MC_bytes value, ST_actions how)
{ switch (how)
  { case ST_add:
      mem[location] = (mem[location] + value) % 256; break;
    case ST_subtract:
      mem[location] = (mem[location] - value + 256) % 256; break;
  }
}

void AS::termvalue(SA_terms term, MC_bytes &value, ST_actions action,
                   bool &undefined, bool &badaddress)
// Determine value of a single term, recording outstanding action
// if undefined so far, and recording badaddress if malformed
{ undefined = false;
  switch (term.kind)
  { case SA_absent:
    case SA_numeric:
      value = term.number % 256; break;
    case SA_star:
      value = location; break;
    case SA_alphameric:
      Table->valueofsymbol(term.name, location, value, action, undefined); break;
    default:
      badaddress = true; value = 0; break;
  }
}

void AS::evaluate(SA_addresses address, MC_bytes &value, bool &undefined,
                  bool &malformed)
// Determine value of address, recording whether undefined or malformed
{ ST_actions nextaction;
  MC_bytes nextvalue;
  bool unknown;
  malformed = false;
  termvalue(address.term[0], value, ST_add, undefined, malformed);
  int i = 1;
  while (i < address.length)
  { switch (address.term[i].kind)
    { case SA_plus:  nextaction = ST_add; break;
      case SA_minus: nextaction = ST_subtract; break;
      default:       nextaction = ST_add; malformed = true; break;
    }
    i++;
    termvalue(address.term[i], nextvalue, nextaction, unknown, malformed);
    switch (nextaction)
    { case ST_add:      value = (value + nextvalue) % 256; break;
      case ST_subtract: value = (value - nextvalue + 256) % 256; break;
    }
    undefined = (undefined || unknown);
    i++;
  }
}

static char *ErrorMsg[] = {
  " - unknown opcode",
  " - address field not resolved",
  " - invalid address field",
  " - label missing",
  " - spurious address field",
  " - address field missing",
  " - address field too long",
  " - wrong number of parameters",
  " - invalid formal parameters",
  " - invalid label",
  " - unknown character",
  " - mismatched quotes",
  " - number too large",
};

void AS::listerrors(ASM_errorset allerrors, bool &failure)
{ if (allerrors.isempty()) return;
  failure = true;
  fprintf(Srce->lst, "Next line has errors");
  for (int error = ASM_invalidcode; error <= ASM_overflow; error++)
    if (allerrors.memb(error)) fprintf(Srce->lst, "%s\n", ErrorMsg[error]);
}
```

```
void AS::listcode(void)
// List generated code bytes on source listing
{ Srce->writehex(objline.location, 4);
  if (objline.opcode >= AS_err && objline.opcode <= AS_if)
    fprintf(Srce->lst, "       ");
  else if (objline.opcode <= MC_hlt)              // OneByteOps
    Srce->writehex(objline.opcode, 7);
  else if (objline.opcode == AS_dc)               // DC special case
    Srce->writehex(objline.address, 7);
  else                                            // TwoByteOps
  { Srce->writehex(objline.opcode, 3);
    Srce->writehex(objline.address, 4);
  }
}

void AS::listsourceline(SA_unpackedlines &srcline, bool coderequired,
                        bool &failure)
// List srcline, with option of listing generated code
{ listerrors(srcline.errors, failure);
  if (coderequired) listcode(); else fprintf(Srce->lst, "           ");
  Srce->writetext(srcline.labfield, 9);
  Srce->writetext(srcline.mnemonic, 9);
  int width = strlen(srcline.address.term[0].name);
  fputs(srcline.address.term[0].name, Srce->lst);
  for (int i = 1; i < srcline.address.length; i++)
  { width += strlen(srcline.address.term[i].name) + 1;
    putc(’ ’, Srce->lst);
    fputs(srcline.address.term[i].name, Srce->lst);
  }
  if (width < 30) Srce->writetext(" ", 30 - width);
  fprintf(Srce->lst, "%s\n", srcline.comment);
}

void AS::definemacro(SA_unpackedlines &srcline, bool &failure)
// Handle introduction of a macro (possibly nested)
{ MC_bytes opcode;
  MH_macro macro;
  bool declared = false;
  int i = 0;
  if (srcline.labelled)                           // name must be present
    declared = true;
  else
    srcline.errors.incl(ASM_unlabelled);
  if (!(srcline.address.length & 1))              // must be an odd number of terms
    srcline.errors.incl(ASM_invalidaddress);
  while (i < srcline.address.length)              // check that formals are names
  { if (srcline.address.term[i].kind != SA_alphameric)
      srcline.errors.incl(ASM_nonalpha);
    i += 2;                                       // bypass commas
  }
  listsourceline(srcline, nocodelisted, failure);
  if (declared) Macro->newmacro(macro, srcline);  // store header
  do
  { Parser->parse(srcline);                       // next line of macro text
    opcode = bytevalue(srcline.mnemonic);
    if (opcode == AS_mac)                         // nested macro?
      definemacro(srcline, failure);              // recursion handles it
    else
    { listsourceline(srcline, nocodelisted, failure);
      if (declared && opcode != AS_end && srcline.errors.isempty())
        Macro->storeline(macro, srcline);         // add to macro text
    }
  } while (opcode != AS_end);
}

void AS::assembleline(SA_unpackedlines &srcline, bool &failure)
// Assemble single srcline
{ if (!include) { include = true; return; }       // conditional assembly
  bool badaddress, found, undefined;
  MH_macro macro;
  int formal;
  Macro->checkmacro(srcline.mnemonic, macro, found, formal);
  if (found)                                      // expand macro and exit
  { if (srcline.labelled) Table->enter(srcline.labfield, location);
    if (formal != srcline.address.length)         // number of params okay?
      srcline.errors.incl(ASM_mismatched);
    listsourceline(srcline, nocodelisted, failure);
    if (srcline.errors.isempty())                 // okay to expand?
      Macro->expand(macro, srcline.address, this, failure);
    return;
  }
  badaddress = false;
  objline.location = location; objline.address = 0;
```

```
    objline.opcode = bytevalue(srcline.mnemonic);
    if (objline.opcode == AS_err)                   // check various constraints
      srcline.errors.incl(ASM_invalidcode);
    else if (objline.opcode > AS_mac ||
             objline.opcode > MC_hlt && objline.opcode < AS_err)
      { if (srcline.address.length == 0) srcline.errors.incl(ASM_noaddress); }
    else if (objline.opcode != AS_mac && srcline.address.length != 0)
      srcline.errors.incl(ASM_hasaddress);
    if (objline.opcode >= AS_err && objline.opcode <= AS_dc)
    { switch (objline.opcode)                       // directives
      { case AS_beg:
          location = 0;
          break;
        case AS_org:
          evaluate(srcline.address, location, undefined, badaddress);
          if (undefined) srcline.errors.incl(ASM_undefinedlabel);
          objline.location = location;
          break;
        case AS_ds:
          if (srcline.labelled) Table->enter(srcline.labfield, location);
          evaluate(srcline.address, objline.address, undefined, badaddress);
          if (undefined) srcline.errors.incl(ASM_undefinedlabel);
          location = (location + objline.address) % 256;
          break;
        case AS_nul:
        case AS_err:
          if (srcline.labelled) Table->enter(srcline.labfield, location);
          break;
        case AS_equ:
          evaluate(srcline.address, objline.address, undefined, badaddress);
          if (srcline.labelled)
            Table->enter(srcline.labfield, objline.address);
          else
            srcline.errors.incl(ASM_unlabelled);
          if (undefined) srcline.errors.incl(ASM_undefinedlabel);
          break;
        case AS_dc:
          if (srcline.labelled) Table->enter(srcline.labfield, location);
          evaluate(srcline.address, objline.address, undefined, badaddress);
          Machine->mem[location] = objline.address;
          location = (location + 1) % 256;
          break;
        case AS_if:
          evaluate(srcline.address, objline.address, undefined, badaddress);
          if (undefined) srcline.errors.incl(ASM_undefinedlabel);
          include = (objline.address != 0);
          break;
        case AS_mac:
          definemacro(srcline, failure);
          break;
        case AS_end:
          assembling = false;
          break;
      }
    }
    else                                            // machine ops
    { if (srcline.labelled) Table->enter(srcline.labfield, location);
      Machine->mem[location] = objline.opcode;
      if (objline.opcode > MC_hlt)                  // TwoByteOps
      { location = (location + 1) % 256;
        evaluate(srcline.address, objline.address, undefined, badaddress);
        Machine->mem[location] = objline.address;
      }
      location = (location + 1) % 256;              // bump location counter
    }
    if (badaddress) srcline.errors.incl(ASM_invalidaddress);
    if (objline.opcode != AS_mac) listsourceline(srcline, codelisted, failure);
  }

  void AS::firstpass(bool &errors)
  // Make first and only pass over source code
  { SA_unpackedlines srcline;
    location = 0; assembling = true; include = true; errors = false;
    while (assembling)
      { Parser->parse(srcline); assembleline(srcline, errors); }
    Table->printsymboltable(errors);
    if (!errors) Table->outstandingreferences(Machine->mem, backpatch);
  }

  void AS::assemble(bool &errors)
  { printf("Assembling ...\n");
    fprintf(Srce->lst, "(One Pass Macro Assembler)\n\n");
    firstpass(errors);
    Machine->listcode();
  }

  AS::AS(char *sourcename, char *listname, char *version, MC *M)
  { Machine = M;
    Srce    = new SH(sourcename, listname, version);
    Lex     = new LA(Srce);
    Parser  = new SA(Lex);
    Table   = new ST(Srce);
    Macro   = new MH();
    // enter opcodes and mnemonics in ALPHABETIC order
    // done this way for ease of modification later
    opcodes = 0;   // bogus one for erroneous data
    enter("Error   ", AS_err);   // for lines with no opcode
    enter("   ", AS_nul); enter("ACI", MC_aci); enter("ACX", MC_acx);
    enter("ADC", MC_adc); enter("ADD", MC_add); enter("ADI", MC_adi);
    enter("ADX", MC_adx); enter("ANA", MC_ana); enter("ANI", MC_ani);
    enter("ANX", MC_anx); enter("BCC", MC_bcc); enter("BCS", MC_bcs);
    enter("BEG", AS_beg); enter("BNG", MC_bng); enter("BNZ", MC_bnz);
    enter("BPZ", MC_bpz); enter("BRN", MC_brn); enter("BZE", MC_bze);
    enter("CLA", MC_cla); enter("CLC", MC_clc); enter("CLX", MC_clx);
    enter("CMC", MC_cmc); enter("CMP", MC_cmp); enter("CPI", MC_cpi);
    enter("CPX", MC_cpx); enter("DC",  AS_dc);  enter("DEC", MC_dec);
    enter("DEX", MC_dex); enter("DS",  AS_ds);  enter("END", AS_end);
    enter("EQU", AS_equ); enter("HLT", MC_hlt); enter("IF",  AS_if);
    enter("INA", MC_ina); enter("INB", MC_inb); enter("INC", MC_inc);
    enter("INH", MC_inh); enter("INI", MC_ini); enter("INX", MC_inx);
    enter("JSR", MC_jsr); enter("LDA", MC_lda); enter("LDI", MC_ldi);
    enter("LDX", MC_ldx); enter("LSI", MC_lsi); enter("LSP", MC_lsp);
    enter("MAC", AS_mac); enter("NOP", MC_nop); enter("ORA", MC_ora);
    enter("ORG", AS_org); enter("ORI", MC_ori); enter("ORX", MC_orx);
    enter("OTA", MC_ota); enter("OTB", MC_otb); enter("OTC", MC_otc);
    enter("OTH", MC_oth); enter("OTI", MC_oti); enter("POP", MC_pop);
    enter("PSH", MC_psh); enter("RET", MC_ret); enter("SBC", MC_sbc);
    enter("SBI", MC_sbi); enter("SBX", MC_sbx); enter("SCI", MC_sci);
    enter("SCX", MC_scx); enter("SHL", MC_shl); enter("SHR", MC_shr);
    enter("STA", MC_sta); enter("STX", MC_stx); enter("SUB", MC_sub);
    enter("TAX", MC_tax);
  }


  ----- mc.h --------------------------------------------------------------------

  // Definition of simple single-accumulator machine and simple emulator
  // P.D. Terry, Rhodes University, 1996

  #ifndef MC_H
  #define MC_H

  #include "misc.h"

  // machine instructions - order important
  enum MC_opcodes {
    MC_nop, MC_cla, MC_clc, MC_clx, MC_cmc, MC_inc, MC_dec, MC_inx, MC_dex,
    MC_tax, MC_ini, MC_inh, MC_inb, MC_ina, MC_oti, MC_otc, MC_oth, MC_otb,
    MC_ota, MC_psh, MC_pop, MC_shl, MC_shr, MC_ret, MC_hlt, MC_lda, MC_ldx,
    MC_ldi, MC_lsp, MC_lsi, MC_sta, MC_stx, MC_add, MC_adx, MC_adi, MC_adc,
    MC_acx, MC_aci, MC_sub, MC_sbx, MC_sbi, MC_sbc, MC_scx, MC_sci, MC_cmp,
    MC_cpx, MC_cpi, MC_ana, MC_anx, MC_ani, MC_ora, MC_orx, MC_ori, MC_brn,
    MC_bze, MC_bnz, MC_bpz, MC_bng, MC_bcc, MC_bcs, MC_jsr, MC_bad = 255 };

  typedef enum { running, finished, nodata, baddata, badop } status;
  typedef unsigned char MC_bytes;

  class MC {
    public:
      MC_bytes mem[256];    // virtual machine memory

      void listcode(void);
      // Lists the 256 bytes stored in mem on requested output file

      void emulator(MC_bytes initpc, FILE *data, FILE *results, bool tracing);
      // Emulates action of the instructions stored in mem, with program counter
      // initialized to initpc.  data and results are used for I/O.
      // Tracing at the code level may be requested

      void interpret(void);
      // Interactively opens data and results files, and requests entry point.
      // Then interprets instructions stored in MC_mem

      MC_bytes opcode(char *str);
      // Maps str to opcode, or to MC_bad (0FFH) if no match can be found
      MC();
      // Initializes accumulator machine

    private:
      struct processor {
        MC_bytes a;      // Accumulator
        MC_bytes sp;     // Stack pointer
        MC_bytes x;      // Index register
        MC_bytes ir;     // Instruction register
        MC_bytes pc;     // Program count
        bool z, p, c;    // Condition flags
      };
      processor cpu;
      status ps;

      char *mnemonics[256];
      void trace(FILE *results, MC_bytes pcnow);
      void postmortem(FILE *results, MC_bytes pcnow);
      void setflags(MC_bytes MC_register);
      MC_bytes index(void);
  };

  #endif /*MC_H*/


  ----- mc.cpp ------------------------------------------------------------------

  // Definition of simple single-accumulator machine and simple emulator
  // P.D. Terry, Rhodes University, 1996

  #include "misc.h"
  #include "mc.h"

  // set break-in character as CTRL-A (cannot easily use \033 on MS-DOS)
  const int ESC = 1;

  inline void increment(MC_bytes &x)
  // Increment with folding at 256
  { x = (x + 257) % 256; }

  inline void decrement(MC_bytes &x)
  // Decrement with folding at 256
  { x = (x + 255) % 256; }

  MC_bytes MC::opcode(char *str)
  // Simple linear search suffices for illustration
  { for (int i = 0; str[i]; i++) str[i] = toupper(str[i]);
    MC_bytes l = MC_nop;
    while (l <= MC_jsr && strcmp(str, mnemonics[l])) l++;
    if (l <= MC_jsr) return l; else return MC_bad;
  }

  void MC::listcode(void)
  // Simply print all 256 bytes in 16 rows
  { MC_bytes nextbyte = 0;
    char filename[256];
    printf("Listing code ... \n");
    printf("Listing file [NUL] ? ");
    gets(filename);
    if (*filename == ’\0’) return;
    FILE *listfile = fopen(filename, "w");
    if (listfile == NULL) listfile = stdout;
    putc(’\n’, listfile);
    for (int i = 1; i <= 16; i++)
    { for (int j = 1; j <= 16; j++)
      { fprintf(listfile, "%4d", mem[nextbyte]); increment(nextbyte); }
      putc(’\n’, listfile);
    }
    if (listfile != stdout) fclose(listfile);
  }

  void MC::trace(FILE *results, MC_bytes pcnow)
  // Simple trace facility for run time debugging
  { fprintf(results, " PC = %02X  A = %02X  ", pcnow, cpu.a);
    fprintf(results, " X = %02X  SP = %02X  ", cpu.x, cpu.sp);
    fprintf(results, " Z = %d P = %d C = %d", cpu.z, cpu.p, cpu.c);
    fprintf(results, " OPCODE = %02X  (%s)\n", cpu.ir, mnemonics[cpu.ir]);
  }

  void MC::postmortem(FILE *results, MC_bytes pcnow)
  // Report run time error and position
  { switch (ps)
    { case badop:   fprintf(results, "Illegal opcode"); break;
```

```
    case nodata:  fprintf(results, "No more data"); break;
    case baddata: fprintf(results, "Invalid data"); break;
  }
  fprintf(results, " at %d\n", pcnow);
  trace(results, pcnow);
  printf("\nPress RETURN to continue\n");
  scanf("%*[^\n]"); getchar();
  listcode();
}

inline void MC::setflags(MC_bytes MC_register)
// Set P and Z flags according to contents of register
{ cpu.z = (MC_register == 0); cpu.p = (MC_register <= 127); }

inline MC_bytes MC::index(void)
// Get indexed address with folding at 256
{ return ((mem[cpu.pc] + cpu.x) % 256); }

void readchar(FILE *data, char &ch, status &ps)
// Read ch and check for break-in and other awkward values
{ if (feof(data)) { ps = nodata; ch = ’ ’; return; }
  ch = getc(data);
  if (ch == ESC) ps = finished;
  if (ch < ’ ’ || feof(data)) ch = ’ ’;
}

int hexdigit(char ch)
// Convert CH to equivalent value
{ if (ch >= ’a’ && ch <= ’e’) return(ch + 10 - ’a’);
  if (ch >= ’A’ && ch <= ’E’) return(ch + 10 - ’A’);
  if (isdigit(ch)) return(ch - ’0’);
  else return(0);
}

int getnumber(FILE *data, int base, status &ps)
// Read number in required base
{ bool negative = false;
  char ch;
  int num = 0;
  do
  { readchar(data, ch, ps);
  } while (!(ch > ’ ’ || feof(data) || ps != running));
  if (ps == running)
  { if (feof(data))
      ps = nodata;
    else
    { if (ch == ’-’) { negative = true; readchar(data, ch, ps); }
      else if (ch == ’+’) readchar(data, ch, ps);
      if (!isxdigit(ch))
        ps = baddata;
      else
      { while (isxdigit(ch) && ps == running)
        { if (hexdigit(ch) < base && num <= (maxint - hexdigit(ch)) / base)
            num = base * num + hexdigit(ch);
          else
            ps = baddata;
          readchar(data, ch, ps);
        }
      }
    }
    if (negative) num = -num;
    if (num > 0)
      return num % 256;
    else
      return (256 - abs(num) % 256) % 256;
  }
  return 0;
}

void MC::emulator(MC_bytes initpc, FILE *data, FILE *results, bool tracing)
{ MC_bytes pcnow;           // Old program count
  MC_bytes carry;           // Value of carry bit

  cpu.z = false; cpu.p = false; cpu.c = false; // initialize flags
  cpu.a = 0;     cpu.x = 0;     cpu.sp = 0;    // initialize registers
  cpu.pc = initpc;                             // initialize program counter
  ps = running;
  do
  { cpu.ir = mem[cpu.pc];   // fetch
    pcnow = cpu.pc;         // record for use in tracing/postmortem
    increment(cpu.pc);      // and bump in anticipation
    if (tracing) trace(results, pcnow);
    switch (cpu.ir)         // execute
```

```
{ case MC_nop:
    break;
  case MC_cla:
    cpu.a = 0; break;
  case MC_clc:
    cpu.c = false; break;
  case MC_clx:
    cpu.x = 0; break;
  case MC_cmc:
    cpu.c = !cpu.c; break;
  case MC_inc:
    increment(cpu.a); setflags(cpu.a); break;
  case MC_dec:
    decrement(cpu.a); setflags(cpu.a); break;
  case MC_inx:
    increment(cpu.x); setflags(cpu.x); break;
  case MC_dex:
    decrement(cpu.x); setflags(cpu.x); break;
  case MC_tax:
    cpu.x = cpu.a; break;
  case MC_ini:
    cpu.a = getnumber(data, 10, ps); setflags(cpu.a); break;
  case MC_inb:
    cpu.a = getnumber(data, 2, ps); setflags(cpu.a); break;
  case MC_inh:
    cpu.a = getnumber(data, 16, ps); setflags(cpu.a); break;
  case MC_ina:
    char ascii;
    readchar(data, ascii, ps);
    if (feof(data)) ps = nodata;
    else { cpu.a = ascii; setflags(cpu.a); }
    break;
  case MC_oti:
    if (cpu.a < 128)
      fprintf(results, "%d ", cpu.a);
    else
      fprintf(results, "%d ", cpu.a - 256);
    if (tracing) putc('\n', results);
    break;
  case MC_oth:
    fprintf(results, "%02X ", cpu.a);
    if (tracing) putc('\n', results);
    break;
  case MC_otc:
    fprintf(results, "%d ", cpu.a);
    if (tracing) putc('\n', results);
    break;
  case MC_ota:
    putc(cpu.a, results);
    if (tracing) putc('\n', results);
    break;
  case MC_otb:
    int bits[8];
    MC_bytes number = cpu.a;
    for (int loop = 0; loop <= 7; loop++)
      { bits[loop] = number % 2; number /= 2; }
    for (loop = 7; loop >= 0; loop--)
      fprintf(results, "%d", bits[loop]);
    putc(' ', results);
    if (tracing) putc('\n', results);
    break;
  case MC_psh:
    decrement(cpu.sp); mem[cpu.sp] = cpu.a; break;
  case MC_pop:
    cpu.a = mem[cpu.sp]; increment(cpu.sp); setflags(cpu.a); break;
  case MC_shl:
    cpu.c = (cpu.a * 2 > 255); cpu.a = cpu.a * 2 % 256;
    setflags(cpu.a); break;
  case MC_shr:
    cpu.c = cpu.a & 1; cpu.a /= 2; setflags(cpu.a); break;
  case MC_ret:
    cpu.pc = mem[cpu.sp]; increment(cpu.sp); break;
  case MC_hlt:
    ps = finished; break;
  case MC_lda:
    cpu.a = mem[mem[cpu.pc]]; increment(cpu.pc); setflags(cpu.a); break;
  case MC_ldx:
    cpu.a = mem[index()]; increment(cpu.pc); setflags(cpu.a); break;
  case MC_ldi:
    cpu.a = mem[cpu.pc]; increment(cpu.pc); setflags(cpu.a); break;
  case MC_lsp:
    cpu.sp = mem[mem[cpu.pc]]; increment(cpu.pc); break;
  case MC_lsi:
```

```
  cpu.sp = mem[cpu.pc]; increment(cpu.pc); break;
case MC_sta:
  mem[mem[cpu.pc]] = cpu.a; increment(cpu.pc); break;
case MC_stx:
  mem[index()] = cpu.a; increment(cpu.pc); break;
case MC_add:
  cpu.c = (cpu.a + mem[mem[cpu.pc]] > 255);
  cpu.a = (cpu.a + mem[mem[cpu.pc]]) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_adx:
  cpu.c = (cpu.a + mem[index()] > 255);
  cpu.a = (cpu.a + mem[index()]) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_adi:
  cpu.c = (cpu.a + mem[cpu.pc] > 255);
  cpu.a = (cpu.a + mem[cpu.pc]) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_adc:
  carry = cpu.c;
  cpu.c = (cpu.a + mem[mem[cpu.pc]] + carry > 255);
  cpu.a = (cpu.a + mem[mem[cpu.pc]] + carry) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_acx:
  carry = cpu.c;
  cpu.c = (cpu.a + mem[index()] + carry > 255);
  cpu.a = (cpu.a + mem[index()] + carry) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_aci:
  carry = cpu.c;
  cpu.c = (cpu.a + mem[cpu.pc] + carry > 255);
  cpu.a = (cpu.a + mem[cpu.pc] + carry) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sub:
  cpu.c = (cpu.a < mem[mem[cpu.pc]]);
  cpu.a = (cpu.a - mem[mem[cpu.pc]] + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sbx:
  cpu.c = (cpu.a < mem[index()]);
  cpu.a = (cpu.a - mem[index()] + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sbi:
  cpu.c = (cpu.a < mem[cpu.pc]);
  cpu.a = (cpu.a - mem[cpu.pc] + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sbc:
  carry = cpu.c;
  cpu.c = (cpu.a < mem[mem[cpu.pc]] + carry);
  cpu.a = (cpu.a - mem[mem[cpu.pc]] - carry + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_scx:
  carry = cpu.c;
  cpu.c = (cpu.a < mem[index()] + carry);
  cpu.a = (cpu.a - mem[index()] - carry + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sci:
  carry = cpu.c;
  cpu.c = (cpu.a < mem[cpu.pc] + carry);
  cpu.a = (cpu.a - mem[cpu.pc] - carry + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_cmp:
  cpu.c = (cpu.a < mem[mem[cpu.pc]]);
  setflags((cpu.a - mem[mem[cpu.pc]] + 256) % 256);
  increment(cpu.pc); break;
case MC_cpx:
  cpu.c = (cpu.a < mem[index()]);
  setflags((cpu.a - mem[index()] + 256) % 256);
  increment(cpu.pc); break;
case MC_cpi:
  cpu.c = (cpu.a < mem[cpu.pc]);
  setflags((cpu.a - mem[cpu.pc] + 256) % 256);
  increment(cpu.pc); break;
case MC_ana:
  cpu.a &= mem[mem[cpu.pc]];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
case MC_anx:
  cpu.a &= mem[index()];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
case MC_ani:
  cpu.a &= mem[cpu.pc];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
case MC_ora:
  cpu.a |= mem[mem[cpu.pc]];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
```

```
      case MC_orx:
        cpu.a |= mem[index()];
        increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
      case MC_ori:
        cpu.a |= mem[cpu.pc];
        increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
      case MC_brn:
        cpu.pc = mem[cpu.pc]; break;
      case MC_bze:
        if (cpu.z) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bnz:
        if (!cpu.z) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bpz:
        if (cpu.p) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bng:
        if (!cpu.p) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bcs:
        if (cpu.c) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bcc:
        if (!cpu.c) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_jsr:
        decrement(cpu.sp);
        mem[cpu.sp] = (cpu.pc + 1) % 256; // push return address
        cpu.pc = mem[cpu.pc]; break;
      default:
        ps = badop; break;
    }
  } while (ps == running);
  if (ps != finished) postmortem(results, pcnow);
}

void MC::interpret(void)
{ char filename[256];
  FILE *data, *results;
  bool tracing;
  int entry;
  printf("\nTrace execution (y/N/q)? ");
  char reply = getchar(); scanf("%*[^\n]"); getchar();
  if (toupper(reply) != ’Q’)
  { tracing = toupper(reply) == ’Y’;
    printf("\nData file [STDIN] ? "); gets(filename);
    if (filename[0] == ’\0’) data = NULL;
    else data = fopen(filename, "r");
    if (data == NULL)
      { printf("taking data from stdin\n"); data = stdin; }
    printf("\nResults file [STDOUT] ? "); gets(filename);
    if (filename[0] == ’\0’) results = NULL;
    else results = fopen(filename, "w");
    if (results == NULL)
      { printf("sending results to stdout\n"); results = stdout; }
    printf("Entry point? ");
    if (scanf("%d%*[^\n]", &entry) != 1) entry = 0; getchar();
    emulator(entry % 256, data, results, tracing);
    if (results != stdout) fclose(results);
    if (data != stdin) fclose(data);
  }
}

MC::MC()
{ for (int i = 0; i <= 255; i++) mem[i] = MC_bad;
  // Initialize mnemonic table
  for (i = 0; i <= 255; i++) mnemonics[i] = "???";
  mnemonics[MC_aci] = "ACI";  mnemonics[MC_acx] = "ACX";
  mnemonics[MC_adc] = "ADC";  mnemonics[MC_add] = "ADD";
  mnemonics[MC_adi] = "ADI";  mnemonics[MC_adx] = "ADX";
  mnemonics[MC_ana] = "ANA";  mnemonics[MC_ani] = "ANI";
  mnemonics[MC_anx] = "ANX";  mnemonics[MC_bcc] = "BCC";
  mnemonics[MC_bcs] = "BCS";  mnemonics[MC_bng] = "BNG";
  mnemonics[MC_bnz] = "BNZ";  mnemonics[MC_bpz] = "BPZ";
  mnemonics[MC_brn] = "BRN";  mnemonics[MC_bze] = "BZE";
  mnemonics[MC_cla] = "CLA";  mnemonics[MC_clc] = "CLC";
  mnemonics[MC_clx] = "CLX";  mnemonics[MC_cmc] = "CMC";
  mnemonics[MC_cmp] = "CMP";  mnemonics[MC_cpi] = "CPI";
  mnemonics[MC_cpx] = "CPX";  mnemonics[MC_dec] = "DEC";
  mnemonics[MC_dex] = "DEX";  mnemonics[MC_hlt] = "HLT";
  mnemonics[MC_ina] = "INA";  mnemonics[MC_inb] = "INB";
  mnemonics[MC_inc] = "INC";  mnemonics[MC_inh] = "INH";
  mnemonics[MC_ini] = "INI";  mnemonics[MC_inx] = "INX";
  mnemonics[MC_jsr] = "JSR";  mnemonics[MC_lda] = "LDA";
  mnemonics[MC_ldi] = "LDI";  mnemonics[MC_ldx] = "LDX";
  mnemonics[MC_lsi] = "LSI";  mnemonics[MC_lsp] = "LSP";
  mnemonics[MC_nop] = "NOP";  mnemonics[MC_ora] = "ORA";
  mnemonics[MC_ori] = "ORI";  mnemonics[MC_orx] = "ORX";
```

```
   mnemonics[MC_ota] = "OTA";  mnemonics[MC_otb] = "OTB";
   mnemonics[MC_otc] = "OTC";  mnemonics[MC_oth] = "OTH";
   mnemonics[MC_oti] = "OTI";  mnemonics[MC_pop] = "POP";
   mnemonics[MC_psh] = "PSH";  mnemonics[MC_ret] = "RET";
   mnemonics[MC_sbc] = "SBC";  mnemonics[MC_sbi] = "SBI";
   mnemonics[MC_sbx] = "SBX";  mnemonics[MC_sci] = "SCI";
   mnemonics[MC_scx] = "SCX";  mnemonics[MC_shl] = "SHL";
   mnemonics[MC_shr] = "SHR";  mnemonics[MC_sta] = "STA";
   mnemonics[MC_stx] = "STX";  mnemonics[MC_sub] = "SUB";
   mnemonics[MC_tax] = "TAX";
 }
```