7 ADVANCED ASSEMBLER FEATURES

It cannot be claimed that the assemblers of the last chapter are anything other than toys - but by now the reader will be familiar with the drawbacks of academic courses. In this chapter we discuss some extensions to the ideas put forward previously, and then leave the reader with a number of suggestions for exercises that will help turn the assembler into something more closely resembling the real thing.

Complete source code for the assembler discussed in this chapter can be found in Appendix D. This source code and equivalent implementations in Modula-2 and Pascal are also to be found on the accompanying source diskette.

7.1 Error detection

Our simple assemblers are deficient in a very important area - little attempt is made to report errors in the source code in a helpful manner. As has often been remarked, it is very easy to write a translator if one can assume that it will only be given correctly formed programs. And, as the reader will soon come to appreciate, error handling adds considerably to the complexity of any translation process.

Errors can be classified on the basis of the stage at which they can be detected. Among others, some important potential errors are as follows:

Errors that can be detected by the source handler

- Premature end of source file - this might be a rather fatal error, or its detection might be used to supply an effective `END` line, as is done by some assemblers, including our own.

Errors that can be detected by the lexical analyser

- Use of totally unrecognizable characters.
- Use of symbols whose names are too long.
- Comment fields that are too wide.
- Overflow in forming numeric constants.
- Use of non-digit characters in numeric literals.
- Use of symbols in the label field that do not start with a letter.

Errors that can be detected by the syntax analyser

- Use of totally unrecognizable symbols, or misplaced symbols, such as numbers where the comment field should appear.

- Failure to form address fields correctly, by misplacing operators, omitting commas in parameter lists, and so on.

Errors that can be detected by the semantic analyser

These are rather more subtle, for the semantics of ASSEMBLER programming are often deliberately vague. Some possible errors are:

- Use of undefined mnemonics.
- Failure to define all labels.
- Supplying address fields for one-byte instructions, or for directives like `BEG, END`.
- Omitting the address for a two-byte instruction, or for directives like `DS` or `DC`.
- Labelling any of the `BEG, ORG, IF` or `END` directives.
- Supplying a non-numeric address field to `ORG` or `EQU`. (This might be allowed in some circumstances).
- Attempting to reference an address outside the available memory. A simple recovery action here is to treat all addresses *modulo* the available memory size, but this, almost certainly, needs reporting.
- Use of the address of "data" as the address in a "branch" instruction. This is sometimes used in clever programming, and so is not usually regarded as an error.
- Duplicate definition, either of macro names, of formal parameter names, or of label names. This may allow trick effects, but should probably be discouraged.
- Failure to supply the correct number of actual parameters in a macro expansion.
- Attempting to use address fields for directives like `ORG, DS, IF` and `EQU` that cannot be fully evaluated at the time these directives take effect. This is a particularly nasty problem in a one-pass system, for forward references will be set up to object bytes that have no real existence.

The above list is not complete, and the reader is urged to reflect on what other errors might be made by the user of the assembler.

A moment’s thought will reveal that many errors can be detected during the first pass of a two-pass assembler, and it might be thought reasonable not to attempt the second pass if errors are detected on the first one. However, if a complete listing is to be produced, showing object code alongside source code, then this will have to wait for the second pass if forward references are to be filled in.

How best to report errors is a matter of taste. Many assemblers are fairly cryptic in this regard, reporting each error only by giving a code number or letter alongside the line in the listing where the error was detected. A better approach, exemplified in our code, makes use of the idea of constructing a *set* of errors. We then associate with each parsed line, not a Boolean error field, but one of some suitable set type. As errors are discovered this set can be augmented, and at an appropriate time error reporting can take place using a routine like `listerrors` that can be found in the enhanced assembler class in Appendix D.

This is very easily handled with implementation languages like Modula-2 or Pascal, which directly support the idea of a set type. In C++ we can make use of a simple template set class, with operators overloaded so as to support virtually the same syntax as is found in the other languages. Code for such a class appears in the appendix.

7.2 Simple expressions as addresses

Many assemblers allow the programmer the facility of including expressions in the address field of instructions. For example, we might have the following (shown fully assembled, and with some deliberate quirks of coding):

```
Macro Assembler 1.0 on 30/05/96 at 21:47:53

(One Pass Assembler)

00                 BEG              ; Count chars and lowercase letters
00         LOOP                     ; LOOP
00  0D             INA              ;   Read(CH)
01  2E 2E          CPI  PERIOD      ;   IF CH = "." THEN EXIT
03  36 19          BZE  EXIT
05  2E 61          CPI  SMALLZ - 25 ;   IF (CH >= "a")
07  39 12          BNG  * + 10
09  2E 7B          CPI  SMALLZ + 1  ;      AND (CH <= "z")
0B  38 12          BPZ  * + 6
0D  19 20          LDA  LETTERS     ;     THEN INC(Letters)
0F  05             INC
10  1E 20          STA  LETTERS     ;   END
12  19 21          LDA  LETTERS + 1 ;   INC(Total)
14  05             INC
15  1E 21          STA  LETTERS + 1
17  35 00          BRN  LOOP        ; END
19  19 20  EXIT    LDA  LETTERS
1B  0F             OTC              ; Write(Letters)
1C  19 21          LDA  TOTAL
1E  0F             OTC              ; Write(Total)
1F  18             HLT
20  00     LETTERS DC   0           ; RECORD Letters, Total : BYTE END
21         TOTAL   EQU  *
21  00             DC   0
22         SMALLZ  EQU  122         ; ascii 'z'
22         PERIOD  EQU  46          ; ascii '.'
22                 END
```

Here we have used addresses like `LETTERS + 1` (meaning one location after that assigned to `LETTERS`), `SMALLZ-25` (meaning, in this case, an obvious 97), and `* + 6` and `* + 10` (a rather dangerous notation, meaning "6 bytes after the present one" and "10 bytes after the present one", respectively). These are typical of what is allowed in many commercial assemblers. Quite how complicated the expressions can become in a real assembler is not a matter for discussion here, but it is of interest to see how to extend our one-pass assembler if we restrict ourselves to addresses of a form described by

```
   Address  =   Term { "+" Term  |  "-" Term } .
   Term     =   Label  |  number |  "*" .
```

where `*` stands for "address of this byte". Note that we can, in principle, have as many terms as we like, although the example above used only one or two.

In a one-pass assembler, address fields of this kind can be handled fairly easily, even allowing for the problem of forward references. As we assemble each line we compute the value of each address field as fully as we can. In some cases (as in `* + 6`) this will be completely; in other cases forward references will be needed. In the forward reference table entries we record not only the address of the bytes to be altered when the labels are finally defined, but also whether these values are later to be added to or subtracted from the value already residing in that byte. There is a slight complication in that all expressions must be computed *modulo* 256 (corresponding to a two's complement representation).

Perhaps this will be made clearer by considering how a one-pass assembler would handle the above code, where we have deliberately delayed the definition of `LETTERS, TOTAL, SMALLZ` and `PERIOD` till the end. For the `LETTERS + 1` address in instructions like `STA   LETTERS + 1` we assemble as though the instruction were `STA   1`, and for the `SMALLZ - 25` address in the instruction `CPI   SMALLZ - 25` we assemble as though the instruction were `CPI   -25`, or, since addresses are computed *modulo* 256, as though the instruction were `CPI 231`. At the point just before `LETTERS` is defined, the assembled code would look as follows:

```
Macro Assembler 1.0 on 30/05/96 at 21:47:53

(One Pass Assembler)

00                 BEG               ; Count chars and lowercase letters
00         LOOP                      ; LOOP
00  0D             INA               ;   Read(CH)
01  2E 00          CPI  PERIOD       ;   IF CH = "." THEN EXIT
03  36 00          BZE  EXIT
05  2E E7          CPI  SMALLZ - 25 ;   IF (CH >= "a")
07  39 12          BNG  * + 10
09  2E 01          CPI  SMALLZ + 1  ;      AND (CH <= "z")
0B  38 12          BPZ  * + 6
0D  19 00          LDA  LETTERS     ;     THEN INC(Letters)
0F  05             INC
10  1E 00          STA  LETTERS     ;   END
12  19 01          LDA  LETTERS + 1 ;   INC(Total)
14  05             INC
15  1E 01          STA  LETTERS + 1
17  35 00          BRN  LOOP        ; END
19  19 00  EXIT    LDA  LETTERS
1B  0F             OTC              ; Write(Letters)
1C  19 00          LDA  TOTAL
1E  0F             OTC              ; Write(Total)
1F  18             HLT
20  00     LETTERS DC   0           ; RECORD Letters, Total : BYTE END
21         TOTAL   EQU  *
21  00             DC   0
22         SMALLZ  EQU  122         ; ascii 'z'
22         PERIOD  EQU  46          ; ascii '.'
22                 END
```

with the entries in the symbol and forward reference tables as depicted in Figure 7.1.

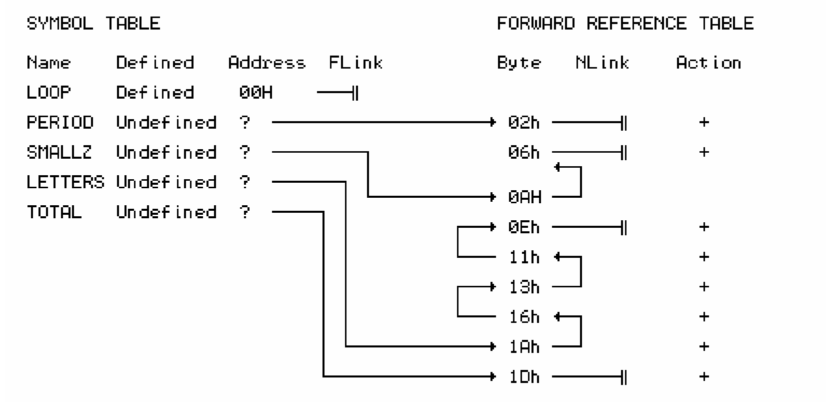

Figure 7.1 Symbol table and forward reference table when simple expressions are allowed to form composite address fields

To incorporate these changes requires modifications to the lexical analyser, (which now has to be able to recognize the characters `+`, `-` and `*` as corresponding to lexical tokens or symbols), to the syntax analyser (which now has more work to do in decoding the address field of an instruction - what was previously the complete address is now possibly just one term of a complex address), and to the semantic analyser (which now has to keep track of how far each address has been computed, as well as maintaining the symbol table).

Some of these changes are almost trivial: in the lexical analyser we simply extend the `LA_symtypes` enumeration, and modify the `getsym` routine to recognize the comma, plus, minus and asterisk as new tokens.

The changes to the syntax analyser are more profound. We change the definition of an unpacked line:

```
  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;
  };
```

and provide a rather grander routine for doing the syntax analysis, which also takes more care to detect errors than before. Much of the spirit of this analysis is similar to the code used in the previous assemblers; the main changes occur in the `getaddress` routine. However, we should comment on the choice of an array to store the entries in an address field. Since each line will have a varying number of terms it might be thought better (especially with all the practice we have been having!) to use a dynamic structure. This has not been done here because we do not really need to create a new structure for each line - once we have assembled a line the address field is of no further interest, and the structure used to record it is thus reusable. However, we need to check that the capacity of the array is never exceeded.

The semantic actions needed result in a considerable extension to the algorithm used to evaluate an address field. The algorithm used previously is delegated to a `termvalue` routine, one that is called repeatedly from the main `evaluate` routine. The forward reference handling is also marginally more complex, since the forward reference entries have to record the outstanding action to be performed when the back-patching is finally attempted. The revised table handler interface needed to accommodate this is as follows:

```
  enum ST_actions { ST_add, ST_subtract };

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

  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)
  };
```

Exercises

7.1 Is it possible to allow a one-pass assembler to handle address fields that contain more general forms of expression, including multiplication and division? Attempt to do so, restricting your effort to the case where the expression is evaluated strictly from left to right.

7.2 One drawback of using dynamic structures for storing the elements of a composite address field is that it may be difficult to recover the storage when the structures are destroyed or are no longer needed. Would this drawback detract from their use in constructing the symbol table or forward reference table?

7.3 Improved symbol table handling - hash tables

In assembly, a great deal of time can be spent looking up identifiers and mnemonics in tables, and it is worthwhile considering how one might improve on the very simple linear search used in the symbol table handler of the previous chapter. A popular way of implementing very efficient table look-up is through the use of **hashing functions**. These are discussed at great length in most texts on data structures, and we shall provide only a very superficial discussion here, based on the idea of maintaining a symbol table in an array of fixed maximum length. For an assembler for a machine as simple as the one we are considering, a fairly small array would surely suffice. Although the possibilities for choosing identifiers are almost unlimited, the choice for any one program will be severely limited - after all, with only 256 bytes in the machine, we are scarcely likely to want to define even as many as 256 labels!

With this in mind we might set up a symbol table structure based on the following declarations:

```
struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool used;              // true after entry made in a table slot
  bool defined;           // true after defining occurrence encountered
  ST_forwardrefs *flink;  // to forward references
};


const int tablemax = 239; // symbol table size (prime number)
ST_entries hashtable[tablemax + 1];
```

The table is initialized by setting the `used` field for each entry to `false` before assembly commences; every time a new entry is made in the table this field is set to `true`.

The fundamental idea behind hashing is to define a simple function based on the characters in an identifier, and to use the returned value as an initial index or key into the table, at which position we hope to be able to store or find the identifier and its associated value. If we are lucky, all identifiers will map to rather scattered and different keys, making or finding an entry in the table will never take more than one comparison, and by the end of assembly there will still be unused slots in the table, and possibly large gaps between the slots that are used.

Of course, we shall never be totally lucky, except, perhaps, in trivial programs. Hash functions are kept very simple so that they can be computed quickly. The simplest of such functions will have the undesirable property that many different identifiers may map onto the same key, but a little reflection will show that, no matter how complicated one makes the function, one always runs the risk that this will happen. Some hash functions are clearly very risky - for example, simply using the value of the first letter in the identifier as a key. It would be much better to use something like

```
          hash = (ident[first] * ident[last]) % tablemax;
```

(which would still fail to discriminate between identifiers like ABC and CBA), or

```
          hash = (ident[first] * 256 + ident[last]) % tablemax;
```

(which would still fail to discriminate between identifiers like AC and ABC).

The subtle part of using a hash table concerns the action to take when we find that some other identifier is occupying the slot identified by the key (when we want to add to the table) or that a different identifier is occupying the slot (when we want to look up the value of an identifier in the table).

If this happens - an event known as a **collision** - we must be prepared to probe elsewhere in the table looking for the correct entry, a process known as **rehashing**. This can be done in a variety of ways. The easiest is simply to make a simple linear search in unit steps from the position identified by the key. This suffers from the disadvantage that the entries in the table tend to get very clustered - for example, if the key is simply the first letter, the first identifier starting with *A* will grab the obvious slot, the second identifier starting with *A* will collide with the first starting with *B*, and so on. A better technique is to use bigger steps in looking for the next slot. A fairly effective way is to use steps defined by a moderately small prime number - and, as we have already suggested, to use a symbol table that is itself able to contain a prime number of items. Then in the worst case we shall easily be able to detect that the table is full, while still being able to utilize every available slot before this happens.

The implementation in Appendix D shows how these ideas can be implemented in a table handler compatible with the rest of the assembler. The suggested hashing function is relatively complicated, but is intended to produce a relatively large range of keys. The search itself is programmed using the so-called *state variable* approach: while searching we can be in one of four states - still looking, found the identifier we are looking for, found a free slot, or found that the table is full.

The above discussion may have given the impression that the use of hashing functions is so beset with problems as to be almost useless, but in fact they turn out to be the method of choice for serious applications. If a little care is taken over the choice of hashing function, the collision rate can be kept very low, and the speed of access very high.

Exercises

7.3 How could one make use of a hash table to speed up the process of matching mnemonics to opcodes?

7.4 Could one use a single hash table to store opcode mnemonics, directive mnemonics, macro labels, and user defined labels?

7.5 In the implementation in Appendix D the hash function is computed within the symbol table handler itself. It might be more efficient to compute it as the identifier is recognized within the scanner. What modifications would be needed to the scanner interface to achieve this?

Further reading

Our treatment of hash functions has been very superficial. Excellent treatments of this subject are to be found in the books by Gough (1988), Fischer and LeBlanc (1988, 1991) and Elder (1994).

7.4 Macro processing facilities

Programming in ASSEMBLER is a tedious business at the best of times, because assembler languages are essentially very simple, and lack the powerful constructs possible in high level languages. One way in which life can be made easier for programmers is to permit them to use macros. A **macro** is a device that effectively allows the assembler language to be extended, by the programmer defining new mnemonics that can then be used repeatedly within the program thereafter. As usual, it is useful to have a clear syntactic description of what we have in mind. Consider the following modification to the `PRODUCTIONS` section of the second Cocol grammar of section 6.1, which allows for various of the extensions now being proposed:

```
  PRODUCTIONS
    ASM               = StatementSequence "END" EOF .
    StatementSequence = { Statement [ comment ] EOL } .
    Statement         = Executable | MacroExpansion | Directive .
    Executable        = [ Label ] [ OneByteOp | TwoByteOp Address ] .
    OneByteOp         = "HLT" | "PSH" | "POP"  (* | . . . . etc *) .
    TwoByteOp         = "LDA" | "LDX" | "LDI"  (* | . . . . etc *) .
    Address           = Term { "+" Term | "-" Term } .
    Term              = Label | number | "*" .
    MacroExpansion    = [ Label ] MacroLabel ActualParameters .
    ActualParameters  = [ OneActual { "," OneActual } ] .
    OneActual         = Term | OneByteOp | TwoByteOp .
    Directive         =   Label "EQU" KnownAddress
                        | [ Label ] ( "DC" Address | "DS" KnownAddress )
                        | "ORG" KnownAddress | "BEG"
                        | "IF" KnownAddress | MacroDefinition .
    Label             = identifier .
    KnownAddress      = Address .
    MacroDefinition   = MacroLabel "MAC" FormalParameters [ comment ] EOL
                        StatementSequence
                        "END" .
    MacroLabel        = identifier .
    FormalParameters  = [ identifier { "," identifier } ] .
```

Put less formally, we are adopting the convention that a macro is defined by code like

```
     LABEL   MAC     P1, P2, P3 ...  ; P1, P2, P3 ... are formal parameters
                                     ; lines of code as usual,
                                     ; using P1, P2, P3 ... in various fields
             END                     ; end of definition
```

where `LABEL` is the name of the new instruction, and where `MAC` is a new directive. For example, we might have

```
     SUM     MAC     A,B,C   ; Macro to add A to B and store in C
             LDA     A
             ADD     B
             STA     C
             END             ; of macro SUM
```

It must be emphasized that a macro definition gives a template or model, and does not of itself immediately generate executable code. The program will, in all probability, not have labels or variables with the same names as those given to the formal parameters.

If a program contains one or more macro definitions, we may then use them to generate executable code by a **macro expansion**, which takes a form exemplified by

```
             SUM     X,Y,Z
```

where `SUM`, the name of the macro, appears in the opcode field, and where `X,Y,Z` are known as **actual parameters**. With `SUM` defined as in this example, code of the apparent form

```
             SUM     X,Y,Z
         L1  SUM     P,Q,R
```

would be expanded by the assembly process to generate actual code equivalent to

```
             LDA     X
             ADD     Y
             STA     Z
         L1  LDA     P
             ADD     Q
             STA     R
```

In the example above the formal parameters appeared only in the address fields of the lines constituting the macro definition, but they are not restricted to such use. For example, the macro

```
     CHK     MAC     A,B,OPCODE,LAB
     LAB     LDA     A
             CPI     B
             OPCODE  LAB
             END                ; of macro CHK
```

if invoked by code of the form

```
             CHK     X,Y,BNZ,L1
```

would produce code equivalent to

```
      L1     LDA     X
             CPI     Y
             BNZ     L1
```

A *macro* facility should not be confused with a *subroutine* facility. The definition of a macro causes no code to be assembled, nor is there any obligation on the programmer ever to expand any particular macro. On the other hand, defining a subroutine *does* cause code to be generated immediately. Whenever a macro is expanded the assembler generates code equivalent to the macro body, but with the actual parameters textually substituted for the formal parameters. For the call of a subroutine the assembler simply generates code for a special form of jump to the subroutine.

We may add a macro facility to a one-pass assembler quite easily, if we stipulate that each macro must be fully defined before it is ever invoked (this is no real limitation if one thinks about it).

The first problem to be solved is that of macro definition. This is easily recognized as imminent by the `assembleline` routine, which handles the `MAC` directive by calling a `definemacro` routine from within the switching construct responsible for handling directives. The `definemacro` routine provides (recursively) for the definition of one macro within the definition of another one, and for fairly sophisticated error handling.

The definition of a macro is handled in two phases. Firstly, an entry must be made into a macro table, recording the name of the macro, the number of parameters, and their formal names. Secondly, provision must be made to store the source text of the macro so that it may be rescanned every time a macro expansion is called for. As usual, in a C++ implementation we can make effective use of yet another class, which we introduce with the following public interface:

```
  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
  };
```

The algorithm for assembling an individual line is, essentially, the same as before. The difference is that, before assembly, the `mnemonic` field is checked to see whether it is a user-defined macro name rather than a standard machine opcode. If it is, the macro is expanded, effectively by assembling lines from the text stored in the macro body, rather than from the incoming source.

The implementation of the macro handler class is quite interesting, and calls for some further commentary:

- A variable of `MC_macro` type is simply a pointer to a node from which depends a queue of unpacked source lines. This header node records the unpacked line that forms the macro header itself, and the address field in this header line contains the formal parameters of the macro.

- Macro expansion is accomplished by passing the lines stored in the queue to the same `assembleline` routine that is responsible for assembling "normal" lines. The mutual recursion which this introduces into the system (the assembler has to be able to invoke the macro expansion, which has to be able to invoke the assembler) is handled in a C++ implementation by declaring a small base class

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

The assembler class is then derived from this one, and the base class is also used as a formal parameter type in the `MH::expand` function. The same sort of functionality is achieved in Pascal and Modula-2 implementations by passing the `assembleline` routine as an actual parameter directly to the `expand` routine.

- The macro expansion has to substitute the actual parameters from the address field of the macro invocation line in the place of any formal parameter references that may appear in each of the lines stored in the macro "body" before those lines can be assembled. These formal parameters may of course appear as labels, mnemonics, or as elements of addresses.

- A macro expansion may instigate another macro expansion - indeed any use of macro processing other than the most trivial probably takes advantage of this feature. Fortunately this is easily handled by the various routines calling one another in a (possibly) mutually recursive manner.

Exercises

7.6 The following represents an attempt to solve a very simple problem:

```
        BEG
CR      EQU     13        ; ASCII carriage return
LF      EQU     10        ; ASCII line feed
WRITE   MAC     A, B, C   ; write integer A and characters B,C
        LDA     A
        OTI               ; write integer
        LDI     B
        OTA               ; write character
        LDI     C
        OTA               ; write character
        END               ; of WRITE macro
READ    MAC     A
        INI
        STA     A
        WRITE   A, CR, LF ; reflect on output
        END               ; of READ macro
LARGE   MAC     A, B, C   ; store larger of A,B in C
        LDA     A
        CMP     B
        BPZ     * + 3
        LDA     B
        STA     C
        END               ; of LARGE macro

        READ    X
        READ    Y
        READ    Z
        LARGE   X, Y, LARGE
        LARGE   LARGE, Z, LARGE
EXIT    WRITE   LARGE, CR, LF
        HLT
LARGE   DS      1
X       DS      1
Y       DS      1
Z       DS      1
        END               ; of program
```

If this were assembled by our macro assembler, what would the symbol, forward reference and macro tables look like just before the line labelled `EXIT` was assembled? Is it permissible to use the identifier `LARGE` as both the name of a macro and of a label?

7.7 The `LARGE` macro of the last example is a little dangerous, perhaps. Addresses like `* + 3` are apt to cause trouble when modifications are made, because programmers forget to change absolute addresses or offsets. Discuss the implications of coding the body of this macro as

```
              LDA     A
              CMP     B
              BPZ     LAB
              LDA     B
       LAB    STA     C
              END             ; of LARGE macro
```

7.8 Develop macros using the language suggested here that will allow you to simulate the **if ... then ... else**, **while ... do**, **repeat ... until**, and **for** loop constructions allowed in high level languages.

7.9 In our system, a macro may be defined *within* another macro. Is there any advantage in allowing this, especially as macros are all entered in a globally accessible macro table? Would it be possible to make nested macros obey scope rules similar to those found in Pascal or Modula-2?

7.10 Suppose two macros use the same formal parameter names. Does this cause problems when attempting macro expansion? Pay particular attention to the problems that might arise in the various ways in which nesting of macro expansions might be required.

7.11 Should one be able to redefine macro names? What does our system do if this is attempted, and how should it be changed to support any ideas you may have for improving it?

7.12 Should the number of formal and actual parameters be allowed to disagree?

7.13 To what extent can a macro assembler be used to accept code in one assembly language and translate it into opcodes for another one?

7.5 Conditional assembly

To realize the full power of an assembler (even one with no macro facilities), it may be desirable to add the facility for what is called **conditional assembly**, whereby the assembler can determine at assembly-time whether to include certain sections of source code, or simply ignore them. A simple form of this is obtained by introducing an extra directive `IF`, used in code of the form

```
            IF      Expression
```

which signals to the assembler that the *following* line is to be assembled only if the *assembly-time* value of *Expression* is non-zero. Frequently this line might be a macro invocation, but it does not have to be. Thus, for example, we might have

```
    SUM     MAC     A,B,C
            LDA     A
            ADD     B
            STA     C
            END             ; macro
            . . .
    FLAG    EQU     1
            . . .
            IF      FLAG
            SUM     X,Y,RESULT
```

which (in this case) would generate code equivalent to

```
            LDA     X
            ADD     Y
            STA     RESULT
```

but if we had set `FLAG EQU 0` the macro expansion for `SUM` would not have taken place.

This may seem a little silly, and another example may be more convincing. Suppose we have defined the macro

```
SUM   MAC     A,B,C,FLAG
      LDA     A
      IF      FLAG
      ADI     B
      IF      FLAG-1
      ADX     B
      STA     C
      END             ; macro
```

Then if we ask for the expansion

```
      SUM     X,45,RESULT,1
```

we get assembled code equivalent to

```
      LDA     X
      ADI     45
      STA     RESULT
```

but if we ask for the expansion

```
      SUM     X,45,RESULT,0
```

we get assembled code equivalent to

```
      LDA     X
      ADX     45
      STA     RESULT
```

This facility is almost trivially easily added to our one-pass assembler, as can be seen by studying the code for the first few lines of the `AS::assembleline` function in Appendix D (which handles the inclusion or rejection of a line), and the `case AS_if` clause that handles the recognition of the `IF` directive. Note that the value of *Expression* must be completely defined by the time the `IF` directive is encountered, which may be a little more restrictive than we could allow with a two-pass assembler.

Exercises

7.14 Should a macro be allowed to contain a reference to itself? This will allow recursion, in a sense, in assembly language programming, but how does one prevent the system from getting into an indefinite expansion? Can it be done with the facilities so far developed? If not, what must be added to the language to allow the full power of recursive macro calls?

7.15 *N!* can be defined recursively as

if *N* = 1 **then** *N!* = 1 **else** *N! = N(N-1)!*

In the light of your answer to Exercise 7.14, can you make use of this idea to let the macro assembler developed so far generate code for computing 4! by using recursive macro calls?

7.16 Conditional assembly may be enhanced by allowing constructions of the form

```
IF      EXPRESSION
   line 1
   line 2
   . . .
```

```
                    ENDIF
```

with the implication that the code up to the directive `ENDIF` is only assembled if `EXPRESSION` evaluates to a non-zero result at assembly-time. Is this really a necessary, or a desirable variation? How could it be implemented? Other extensions might allow code like that below (with fairly obvious meaning):

```
                    IF      EXPRESSION
                         line 1
                         line 2
                         . . .
                    ELSE
                         line m
                         line n
                         . . .
                    ENDIF
```

7.17 Conditional assembly might be made easier if one could use Boolean expressions rather than numerical ones. Discuss the implications of allowing, for example

```
                    IF      A > 0
```

or

```
                    IF      A <> 0 AND B = 1
```

7.6 Relocatable code

The assemblers that we have considered so far have been load-and-go type assemblers, producing the machine instructions for the absolute locations where they will reside when the code is finally executed. However, when developing a large program it is convenient to be able to assemble it in sections, storing each separately, and finally linking the sections together before execution. To some extent this can be done with our present system, by placing an extra load on programmers to ensure that all the sections of code and data are assembled for different areas in memory, and letting them keep track of where they all start and stop.

This is so trivial that it need be discussed no further here. However, such a scheme, while in keeping with the highly simplified view of actual code generation used in this text, is highly unsatisfactory. More sophisticated systems provide the facility for generating relocatable code, where the decision as to where it will finally reside is delayed until loading time.

At first sight even this seems easy to implement. With each byte that is generated we associate a flag, indicating whether the byte will finally be loaded unchanged, or whether it must be modified at load time by adding an offset to it. For example, the section of code

```
       00                  BEG
       00  19 06           LDA  A
       02  22 37           ADI  55
       04  1E 07           STA  B
       06  0C     A        DC   12
       07  00     B        DC   0
       08                  END
```

contains two bytes (assembled as at 01h and 05h) that refer to addresses which would alter if the code was relocated. The assembler could easily produce output for the loader on the lines of the following (where, as usual, values are given in hexadecimal):

```
    19 0    06 1    22 0    37 0    1E 0    07 1    0C 0    00 0
```

Here the first of each pair denotes a loadable byte, and the second is a flag denoting whether the byte needs to be offset at load time. A *relocatable code* file of this sort of information could, again, be preceded by a count of the number of bytes to be loaded. The loader could read a set of such files, effectively concatenating the code into memory from some specified overall starting address, and keeping track as it did so of the offset to be used.

Unfortunately, the full ramifications of this soon reach far beyond the scope of a naïve discussion. The main point of concern is how to decide which bytes must be regarded as relocatable. Those defined by "constants", such as the opcodes themselves, or entries in the symbol table generated by `EQU` directives are clearly "absolute". Entries in the symbol table defined by "labels" in the label field of other instructions may be thought of as relocatable, but bytes defined by expressions that involve the use of such labels are harder to analyse. This may be illustrated by a simple example.

Suppose we had the instruction

```
            LDA   A - B
```

If `A` and `B` are absolute, or are both relocatable, and both defined in the section of code being assembled, then the difference is absolute. If `B` is absolute and `A` is relocatable, then the difference is still relocatable. If `A` is absolute and `B` is relocatable, then the difference should probably be ruled inadmissible. Similarly, if we have an instruction like

```
            LDA   A + B
```

the sum is absolute if both `A` and `B` are absolute, is relocatable if `A` is relocatable and `B` is absolute, and probably inadmissible otherwise. Similar arguments may be extended to handle an expression with more than two operands (but notice that expressions with multiplication and division become still harder to analyse).

The problem is exacerbated still further if - as will inevitably happen when such facilities are properly exploited - the programmer wishes to make reference to labels which are *not* defined in the code itself, but which may, perhaps, be defined in a separately assembled routine. It is not unreasonable to expect the programmer explicitly to declare the names of all labels to be used in this way, perhaps along the lines of

```
            BEG
            DEF     A,B,C     ; these are available for external use
            USE     X,Y,Z     ; these are not defined, but required
```

In this case it is not hard to see that the information presented to the loader will have to be quite considerably more complex, effectively including those parts of the symbol table relating to the elements of the `DEF` list, and those parts of the forward reference tables that relate to the `USE` list. Rather cowardly, we shall refrain from attempting to discuss these issues in further detail here, but leave them as interesting topics for the more adventurous reader to pursue on his or her own.

7.7 Further projects

The following exercises range from being almost trivial to rather long and involved, but the reader who successfully completes them will have learned a lot about the assembly translation process, and possibly even something about assembly language programming.

7.18 We have discussed extensions to the one-pass assembler, rather than the two-pass assembler.

Attempt to extend the two-pass assembler in the same way.

7.19 What features could you add to, and what restrictions could you remove from the assembly process if you used a two-pass rather than a one-pass assembler? Try to include these extra features in your two-pass assembler.

7.20 Modify your assembler to provide for the generation of relocatable code, and possibly for code that might be handled by a linkage editor, and modify the loader developed in Chapter 4, so as to include a more sophisticated linkage editor.

7.21 How could you prevent programmers from branching to "data", or from treating "instruction" locations as data - assuming that you thought it desirable to do so? (As we have mentioned, assembler languages usually allow the programmer complete freedom in respect of the treatment of identifiers, something which is expressly forbidden in strictly typed languages like Pascal, but which some programmers regard as a most desirable feature of a language.)

7.22 We have carefully chosen our opcode mnemonics for the language so that they are lexically unique. However, some assemblers do not do this. For example, the 6502 assembler as used on the pioneering Apple II microcomputer had instructions like

```
LDA  2      equivalent to our     LDA  2
```

and

```
LDA  #2     equivalent to our     LDI  2
```

that is, an extra character in the address field denoted whether the addressing mode was "direct" or "immediate". In fact it was even more complex than that: the `LDA` mnemonic in 6502 assembler could be converted into one of 8 machine instructions, depending on the exact form of the address field. What differences would it make to the assembly process if you had to cater for such conventions? To make it realistic, study the 6502 assembler mnemonics in detail.

7.23 Another variation on address field notation was provided by the Intel 8080 assembler, which used mnemonics like

```
MOV  A, B         and     MOV  B, A
```

to generate different single byte instructions. How would this affect the assembly process?

7.24 Some assemblers allow the programmer the facility to use "local" labels, which are not really part of a global symbol list. For example, that provided with the UCSD p-System allowed code like

```
LAB    MVI   A, 4
       JMP   $2
       MVI   B, C
$2     NOP
       LHLD  1234
LAB2   XCHG
       POP   H
$2     POP   B
       POP   D
       JMP   $2
LAB3   NOP
```

Here the `$2` label between the `LAB1` and `LAB2` labels and the `$2` label between the `LAB2` and `LAB3` labels are local to those demarcated sections of code. How difficult is it to add this sort of facility to an assembler, and what would be the advantages in having it?

7.25 Develop a one-pass or two-pass macro assembler for the stack-oriented machine discussed in Chapter 4.

7.26 As a more ambitious project, examine the assembler language for a real microprocessor, and write a good macro assembler for it.