Systems Programming

Chapter 2
Assemblers

1

Outline

- Design and implementation of assemblers
  - fundamental functions:
    - translating mnemonic operation codes to their machine language equivalents
    - assigning machine addresses to symbolic labels
  - machine dependence
    - different machine instruction formats and codes
- Design of basic assembler for SIC
  - a starting point for building more advanced assemblers
- Machine-independent features
- Features of an assembler not reflected in the assembler language, such as number of passes.

3

Homework #1 (Due: Feb 10th)

- Download the SIC simulator (via anonymous ftp) (faculty/beck/SystemSoftware.tar) from rohan.sdsu.edu and install it on a system of your choice.
- Do Chapter 1's Exercises: 1, 3, 6, 8, 10, 12.
- Do Chapter 2's Exercises: S1: 1, 2; S2:3, S3:2
- Test your codes using the SIC simulator.
- Get a screen dump of your runs and submit them along with your codes printed on paper (Be wise in using the paper and toner: use small fonts, draft/toner saving/multi-page output/full duplex modes if possible!)

2

A Simple SIC Assembler

- Figure 2.2
- "Loc" column gives the machine address for each part of the assembled program
- Assemblers
  - do translation of source code to machine code
  - process assembler directives
  - write the generated object code onto some output device

6

- Figure 2.1
- Main routine reads records from an input device (identified with device code F1) and copies them to an output device (code 05)
  - Main routine calls subroutine RDREC to read a record into a buffer
  - and subroutine WRREC to write a record from the buffer to the output device

Why the buffer is necessary?

4

- **STCH BUFFER, X** *STORE CHARACTER IN BUFFER*

  indicates indirect addressing

- Comment lines start with "."
- **Figure 2.2:**
  - End of each record is marked with a null character ("00")
  - The end of the file to be copied is indicated by a zero-length record

5

Translation Process

Source program → Object code

1. Convert mnemonic operation codes to their machine language equivalents
2. Convert symbolic operands to their equivalent machine addresses
3. Build the machine instructions in the proper format
4. Convert the data constants specified in the source program into their internal machine representations
5. Write the object program and the assembly listing

7

Translation of addresses…

Consider the statement

```
10  1000  FIRST  STL  RETADR  141033
                        ↑
                  forward reference
```

If we attempt to translate the program line by line, . . . ?

8

Processing Assembler Directives

- Also called as pseudo-instructions
- They provide instructions to the assembler itself.

Are they translated to machine code?

9

Some Assembler Directives

- START: specify name and starting address for the program
- END : indicate the end of the source program and (optionally) specify the first executable instruction in the program.
- BYTE : Generate character or hexadecimal constant
- WORD : Generate one-word integer constant
- RESB : Reserve the indicated number of bytes for a data area
- RESW : Reserve the indicated number of words for a data area

10

Writing Object Code

- Object program will later be loaded into memory for execution.
- Simple object program format contains three types of records:
  - Header record: program name, starting address, and length
  - Text records: translated instructions and data
  - End record: marks the end of the program and specifies the address where the execution is to begin

11

- Header record:
  - Col. 1 H
  - Col. 2-7 Program name
  - Col. 8-13 Starting address of object program
  - Col. 14-19 Length of object program in bytes

The term column is used rather than byte to refer to positions within object program records

12

- Text record:
  - Col. 1 T
  - Col. 2-7 Starting address for object code in this record
  - Col. 8-9 Length of object code in this record in bytes
  - Col. 10-69 Object code

13

- End record:
  - Col. 1 E
  - Col. 2-7 Address of first executable instruction in object program

14

- Figure 2.3
- Note that there is no object code corresponding to addresses 1033-2038.

15

Pass 1 (define symbols)

1. Assign addresses to all statements in the program
2. Save the values (addresses) assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives (such as determining the length of data areas defined by BYTE, RESW, etc.)

16

Pass 2 (assemble instructions and generate object program):

1. Assemble instructions (translating operation codes and looking up addresses.)
2. Generate data values defined by BYTE, WORD, etc.
3. Perform processing of assembler directives not done during Pass 1.
4. Write the object program and the assembly listing.

17

Assembler Algorithm and Data Structures

- Our assembler uses two internal tables:
  - The Operation Code Table (OPTAB)
    - used to look up mnemonic operation codes and translate them to their machine language equivalents
  - The Symbol Table (SYMTAB)
    - used to store values (addresses) assigned to labels

18

- LOCCTR – Location Counter
  - used to help in the assignment of addresses
    - initialized to the beginning address
    - Incremented after processing each statement (How?)
    - Whenever we reach a label, the current value of LOCCTR gives the address to be associated with that label.

19

- Usage of OPTAB
  - Pass 1: to look up and validate operation codes in the source program
  - Pass 2: to translate the operation codes to machine language

  How does the usage change for a machine that has instructions of different lengths?

20

- Usage of SYMTAB
  - includes the name and value (address) for each label in the source program, together with flags to indicate error conditions
  - may also contain other info about the data area or instruction labeled
  - **Pass 1:** labels are entered into table as they are encountered along with their addresses
  - **Pass 2:** symbols used as operands are looked up in table to obtain the addresses to be inserted in the assembled instructions.

21

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key.
  - in most cases, OPTAB is a static table
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.
  - selection of a suitable hashing function ?

22

Pass 1 – Pass 2 Communication

- Two alternatives:
  - Both passes can read the original source program as input
  - Pass 1 can generate an intermediate file that contains
    - each source statement together with its assigned address, error indicators, etc.
- Logic flow of two passes
  - Figure 2.4 a/b

23

Program Relocation

- Multiprogramming: we may load and run several programs at the same time.
  - more productive use of hardware
- Relocation: we must be able to load programs into memory wherever there is room, rather than a fixed address at assembly time.
  - if we knew in advance exactly which programs were to be executed concurrently in this way, … ?

24

- Actual starting address of the program is not known until load time.
- Program in Figure 1 is an example of an absolute program (or absolute assembly) – it must be loaded at address 10000 in order to execute properly.

55 101B LDA THREE 00102D

load register A from memory address 102D

What happens if we load our program at address 2000?

25

- Some parts should remain the same regardless of where the program is loaded:

85 102D THREE WORD 3 000003

- Looking at the object code alone, it is in general not possible to tell which values represent addresses and which represent constant data items.
  - Assembler cannot make the necessary changes in the addresses used by the program
  - But assembler can identify for the loader those parts of the object program that need modification

An object program that contains the information necessary to perform this kind of modification is called a relocatable program.

26

- Figs 2.5 – 2.6 – 2.7
- Note that no matter where the program is loaded, RDREC is always 1036 bytes past the starting address of the program.

27

Solving the relocation problem

1. When the assembler generates the object code for the JSUB instruction we are considering, it will insert the address of RDREC relative to the start of the program (this is why LOCCTR is initialized to 0)
2. The assembler will also produce a command for the loader, instructing it to add the beginning address of the program to the address field in the JSUB instruction at load time.

28

Modification Record

Col 1 M

Col. 2-7 Starting location of the address field to be modified, relative to the beginning of the program

Col. 8-9 Length of the address field to be modified, in half bytes.

why?

For the JSUB instruction we are using, modification record would be:

M00000705

29

- The only parts of the program that require modification at load time are those that specify direct (as opposed to relative) addresses.
- Fig. 2.8 – The load addresses in the Text records are interpreted as relative, rather than absolute, locations.
  - there is one Modification record for each address field that needs to be changed when the program is relocated

30

Machine-Independent Assembler Features

- Implementation of Literals
- Implementation of Expressions
- Program blocks and control sections

31

Literals

- It is often convenient for the programmer to be able to write the value of a constant operand as a part of the instruction that uses it.
  - no definition and label are required
- Fig 2.9

```
45  001A  ENDFIL  LDA  =C'EOF'    032010

215 1062  WLOOP   TD   =X'05'     E32011
```

32

Immediate operand vs. Literal vs.

- The operand value is assembled as part of the machine instruction (Imm.Op)
- The assembler generates the specified value as a constant as the target address for the machine instruction. (The effect of using a literal is exactly the same as if the programmer had defined the constant explicitly and used the label assigned to the constant as the instruction operand)
  - Most assemblers recognize duplicate literals and store only one copy of the specified data value.

33

Literal Pools

- All of the literal operands used in a program are gathered together into one or more literal pools.
  - Normally literals are placed into a pool at the end of the program.
  - In some cases, however, it is desirable to place literals into a pool at some other location in the object program
    - LTORG assembler directive: creates a literal pool that contains all of the literal operands used since the previous LTORG (or beginning of the program.)

34

Handling Literal Operands

- Literal Table LITTAB: Basic data structure
  - contains literal name, operand value and length, address assigned to the operand when it is placed in a literal pool
  - often organized as a hash table, using the literal name or value as the key

PASS 1: search LITTAB for the literal, if not in, add into the table

If a LTORG directive or end of program is encountered, each literal currently in the table is assisned an address (unless done before)

PASS 2: For each literal operand encountered, search for the operand address from the table, insert the data values at appropriate places in the object program

Generate a modification record if required.

35

Symbol Defining Statements

- Up to now, user-defined symbols were used to mark labels.
  - Their value is their address!
- EQU assembler directive is used to define symbols and specify their values:

```
symbol    EQU    value
```

36

Uses of Symbolic Names

1-to establish symbolic names that can be used for improved readability in place of numerica values

```
        +LDT    #4096

MAXLEN  EQU     4096  //into SYMTAB
        +LDT    #MAXLEN
```

How does this affect resulting object code? source code?

37

Uses of Symbolic Names

2-in defining mnemonic names for registers in a machine with a number of general-purpose registers; e.g., R0, R1, R2,...

```
BASE    EQU     R1
COUNT   EQU     R2
INDEX   EQU     R3
```

Names reflect the logical function of registers in the program.

We skip ORG (ORiGin) directive that assigns values to symbols indirectly for the time being.

38

Restrictions in symbol defining assembler directives

All symbols used on the right-hand side of the statement must have been defined previously in the program.

Consider:

```
ALPHA   RESW    1
BETA    EQU     ALPHA
```

vs. WHY ?

```
BETA    EQU     ALPHA
ALPHA   RESW    1
```

39

Expressions

- Assemblers allow arithmetic expressions formed using the operators +, -, *, and /.
- Individual terms of the expressions:
  - constants
  - user-defined symbols
  - special terms
    - such as "*" - the current value of the location counter: represents the value of the next unassigned memory location.

```
106  BUFEND  EQU  *  //address of next byte
                     //after the buffer area.
```

40

- The values of terms:
  - absolute term: constants
  - relative terms: labels on instructions and data areas, references to the location counter value
- The values of expressions:
  - absolute expression: an expression that contains
    - only absolute terms
    - relative terms provided that they occur in pairs and the terms in each such pair have opposite signs. (None of the relative terms may enter into a * or / operation)
  - relative expression: an expression in which all of the relative terms except one can be paired as described above and the remaining unpaired relative term must have a positive sign.

41

Why so many rules?

- Hint: A relative term or expression represents some value that may be written as (S+r), where
  - S is the starting address of the program
  - and r is the value of the term or expression relative to the starting address.

```
107MAXLEN  EQU  BUFEND-BUFFER
```

What about BUFEND+BUFFER, 100-BUFFER, 3*BUFFER ?

42

Symbol Table revisited

To determine the type of an expression, we must keep track of the types of all symbols defined in the program:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

43

Gaining flexibility

- So far all example programs were treated as a unit.
  - How come? There were subroutines and data areas within the code !
- We will see two features that allow more flexible handling of the source and object programs:
  - program blocks: refer to segments of code that are rearranged within a single object program unit
  - control sections: refer to segments that are translated into independent object program units.

44

Program Blocks

- allow the generated machine instructions and data to appear in the object program in a different order from the corresponding source statements
  - USE assembler directive indicates which portions of the source program belong to the various blocks
  - may also indicate a continuation of a previously begun block.
  - Figure 2.11: unnamed default block, CDATA block, CBLKS block.

45

- The assembler will (logically) rearrange these segments to gather together the pieces of each block.
- Pass 1: maintains a separate location counter for each program block
  - initialized to 0 at the beginning
  - saved when switching to another block
  - restored when resuming a previous block

Block name	**Block#**	**Address**	**Length**
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

46

- Pass 2: The assembler needs the address for each symbol relative to the start of the object program.
  - simply adds the location of the symbol, relative to the start of its block, to the assigned block starting address.

Figure 2.12 – MAXLEN does not have a block number !!

47

```
20   0006    0      LDA          LENGTH      032060
```

LENGTH's relative location is 0003 within program block 1 (CDATA) whose starting address is 0066
Desired target address: 0003 + 0066 = 0069

Advantages of the separation of the program into blocks:

- improved program readability : definitions of data areas are placed in the source program close to the statements that reference them.
  - human factors and machine configurations conflict!

48

Fig.2.13 – Corresponding Object Program

- It does not matter that the Text records of the object program are not in sequence by address
  - the loader will simply load the object code from each record at the indicated address.

49

Control Sections and Program Linking

- A control section is a part of the program that maintains its identity after assembly
  - each such control section can be loaded and relocated independently of the others
  - different control sections are most often used for subroutines of a program
    - programmer can assemble, load, and manipulate each of these control sections separately (flexibility!)

50

Linking them together

- External references: Instructions in one control section might need to refer to instructions or data located in another section
  - Because control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way.
  - The assembler generates information for each external reference that will allow the loader to perform the required linking.

51

Fig.2.15

- 5 START COPY – first control section
- 109 RDREC CSECT – second section
- 193 WRREC CSECT – third section

Symbols defined in one control section may not be used directly by another section: they must be identified as external references for the loader to handle:

- EXTDEF (external definition) – names external symbols
- EXTREF (external reference) – names symbols used here but defined elsewhere

52

How to handle external references?

```
15  0003  CLOOP   +JSUB  RDREC  4B100000
```

External reference

The assembler inserts an address of zero and passes information to the loader – to be inserted at load time!

```
160 0017          +STCH   BUFFER,X  57900000
```

External reference

```
190 0028    MAXLEN WORD   BUFEND-BUFFER  000000
```

53

- The assembler
- must remember in which control section a symbol is defined.
- must allow the same symbol to be used in different control sections.

- Any attempt to refer to a symbol in another control section must be flagged as an error unless the symbol is identified as an external reference.

54

- So far we have seen how the assembler leaves room in the object code for the values of external symbols.
- Now we will see how the assembler informs the loader about the empty locations to be filled out.
  - Define record: gives info about external symbols that are defined in the section
  - Refer record: lists symbols that are used as external references by the control section.

55

Define record:

- Col 1 D
- Col 2-7 Name of external symbol defined
- Col 8-13 Relative address of symbol within this control section
- Col 14-73 Repeat above info for others

Refer record:

- Col 1 R
- Col 2-7 Name of external symbol referred to
- Col 8-73 Names of other external ref symbols

56

Modification record (revised):

Col 1 M

Col. 2-7 Starting address of the field to be modified, relative to the beginning of the control section

Col. 8-9 Length of the field to be modified, in half bytes.

Col 10 Modification flag (+ or -)

Col 11-16 External symbol whose value is to be added to or subtracted from the indicated field

57