CSI 402 – Lecture 5
(Assemblers – Continued)
**Example 1:** Consider the SIC instruction

\[ \text{LDA THREE} \]

Assume the following:

- The \texttt{START} directive specifies the value 100 (decimal).
- The LC value of the above instruction is 103 (decimal).
- The symbol \texttt{THREE} has LC value = 115 (decimal).

The assembled form of the instruction (in hex) is 000073.

**Remarks:**

- The above instruction uses \underline{absolute} addresses.
- Works correctly as long as the program starts from location 100.
- This is too rigid a requirement in a \underline{multiprogramming} environment.
**Relocatable program:**

- A program that works correctly regardless of starting address.

**Remarks:**

- Assembler should produce relocatable object code.

- Assembler assumes the starting address to be zero; all addresses specified are relative to the starting address of the program.

- Assembler identifies parts of the object program that need to be modified when the program is relocated. (The modification will be done by the **loader**.)
How can an assembler produce relocatable code?

- Each time the assembler produces an instruction with an address, a modification record (or M-record) is produced.

- Each M-record contains
  - Starting location of the address field to be modified.
  - Length of the address field (in say, bytes, half-bytes or bits).

- M-records are appended to the object code.

**Example 2:** Relocatable translation for the instruction in Example 1.

- The object code produced is 00000F.

- M-record: Starting location = 4, Length = 15 bits (or 2 bytes).
Relocation for SIC and SIC/XE:

- **SIC:** All instructions except RSUB and I/O instructions cause a modifier record to be written.

  **Reason:** All instructions except RSUB and I/O instructions use a (15 bit) memory address.

- **SIC/XE:** Only 4-byte instructions may cause a modifier record to be written.

**Exercise:** For SIC/XE, the 1, 2 or 3-byte instructions don’t need an M-record. Why?
Errors Detected by Assemblers

- Undefined symbols.
- Multiply defined symbols.
- Illegal opcode.
- Missing or extra operands.
- Relative addressing infeasible (SIC/XE).

**Exercise:** For each of the errors above, indicate in which pass of a 2-pass assembler the error can be detected.
**Literal:**

- A constant operand written as part of the instruction.

**Examples:**

- LDA =C’PQR’
- TD =X’05’

**Note:** Literals are different from immediate operands. (An example to illustrate the difference appears on the next page.)
Example: The SIC/XE statement

    LDA #112

will be assembled into the 3-byte instruction 010070 (hex).

Exercise: Verify the above translation.

However, the statement

    LDA =C’PQR’

is equivalent to

    LDA LIT1
    .
    .
    LIT1 BYTE =C’PQR’

Note: LIT1 is a new label created by the assembler.
Remarks:

- Assembler may create new (special) labels for literals.

- Normally, a “literal pool” is created at the end of a program.

- Assembler may choose to have the literal pool at a different point (instead of at the end) to allow PC-relative addressing.

- A special directive LTORG used for this purpose. (See example on the next page.)
Remarks on Literals (continued):

Example:

LDA =C’PQR’
.
.
.
J    NEXT
LTORG
.
.
.
NEXT  LDB    #80

Some assemblers use a Literal Table to save space for duplicate literals.
Symbol Defining Directives

- EQU directive is useful in defining constants.

**Example:**

```
MAXADR   EQU   65535
```

- Allows instructions such as

```
LDA      #MAXADR
```

- The Symbol Table can be used to handle such constants.
Expressions may involve constants.

**Example:**

```
NREC   EQU   200
RSIZE  EQU   15

LOC   RESW   NREC*SIZE
```

Expressions may also involve addresses.

**Example:**

```
STA   START+2
```
Expression evaluation needed in both passes.

- Values of expressions used with RESW and RESB directives must be computed in Pass 1.
- Values of expressions such as START+2 may need to be computed in Pass 2.

Typical expression evaluation algorithm:

1. Convert expression to postfix form.
2. Evaluate postfix expression using a stack.
One-Pass Assemblers

- **One-pass:** Assembler makes only one physical pass over the source file.

- **Main problem:** Forward references.

- **Two types:** Load-and-go and Object File Assemblers.

(a) **Load-and-Go Assemblers:**

- Intermediate version and final object code are kept in main memory.

- **Advantage:** Program can begin execution right after assembly.

- To handle forward references, modify the Symbol Table (ST).
In a two-pass assembler, each symbol table (ST) entry contains a symbol and its LC value.

For a one-pass assembler, each ST entry has:

- Symbol
- Defined? (Boolean flag)
- LC Value
- Pointer to the list of locations where the LC value for the symbol is needed. (The list becomes empty once we have the LC value for the symbol.)

**Outline of Algorithm:** See Handout 5.1.

**Example:** To be discussed in class using the program segment in Handout 5.2.
Object File Assemblers

- Not commonly used.
- Object code bytes written out to the file are “unavailable” for patching.
- Patching is done at run time (by the loader).
- As in load-and-go assemblers, use the modified symbol table and store forward references as linked lists.
- When a symbol gets defined, output a text record for each forward reference of the symbol using the list.

Example: To be discussed in class using the program segment in Handout 5.2.
Under some circumstances, an assembler may not be able to produce object code in two passes.

**Example:**

```
ALPHA EQU BETA
DELTA EQU ALPHA
.
.
ARRAY RESW ALPHA
.
.
DELTA EQU 24
```
Multi-Pass Assemblers (continued)

- Multi-pass assemblers are not common.
  - Assembly takes more time.
  - Handling ST requires additional overhead. (A symbol may appear in the label field many times.)
  - Such programs are more difficult to understand.

Suggested Exercises:

1. Do the exercises mentioned in slides 5-5, 5-6 and 5-8.

2. Study the algorithm for 1-pass assembly discussed in Handout 5.1. (Make sure that you understand the algorithm well enough to apply it to program segments such as the one shown in Handout 5.2.)