Chapter-3 Assembler

Assembly language to Machine language

Agenda

- Elements of Assembly language Programming.
- Design of the Assembler.
- Advanced Assembly process.
- Variants of Assembler design.

Why Assembly?

- It gives direct control over the hardware.
- It’s all about Speed, Precision and Hardware-level control.

Modern Uses of Assembly

•Bootloaders (e.g., BIOS, UEFI)

•OS kernels (Linux, Windows)

•Device drivers

•Embedded systems (microcontrollers)

•Cryptography and signal processing

•Compiler backend and optimization tools

Assembly Language programming

- Low Level
- Specific to certain computer system(family of computer system)

Processor Family	Used In	Example Assembly Mnemonics	Notes
Intel x86 / x86-64	PCs, laptops, servers (Intel, AMD CPUs)	MOV AX, BX, ADD AX, 1	Most common for desktops and Windows/Linux systems.
ARM / ARM64 (AArch64)	Mobile phones, tablets, Raspberry Pi, IoT devices	MOV R0, #1, ADD R1, R2, R3	Power-efficient; dominant in smartphones (Android, iPhone).
MIPS	Embedded systems, routers, older consoles	LW $t0, 0($t1), ADD $s1, $s2, $s3	Used in education and embedded systems.
Motorola 68000 (68K)	Early Apple Macs, Amigas, embedded systems	MOVE.L D0, D1, ADDQ.W #1, D2	Historically important; elegant design.
PowerPC	Older Macs, game consoles (PS3, Wii), embedded	addi r3, r4, 1	Now less common but used in some industrial devices.
RISC-V	Open-source, modern embedded & research CPUs	addi x1, x0, 10, add x2, x1, x3	Gaining rapid popularity; open and free ISA.
SPARC	Sun/Oracle servers, UNIX systems	ADD %o0, %o1, %o2	Used mainly in enterprise-grade systems.
IBM System/360 – z/Architecture	Mainframes (IBM zSeries)	L 1, DATA, A 1, SUM	Used in large-scale enterprise computing.

Assembly Language Format

LABEL	MNEMONIC	OPERANDS	COMMENTS

- Syntax : [label [:]…] [mnemonic [operands]][; comments]
- LABEL (symbols) are optional and are used to identify a particular statement so that it can be referred to from other parts of the program.
- MNEMONIC defines the function of a particular line and it can either be one of the 68000instruction mnemonics(incase of Motorola 68K) or it may be an assembler directive or pseudo-instruction

Assembly Language Format

- OPERAND maybe a register name, an address or data or a label referring to either an address or data.
  - When two operands are required, the first operand specifies the destination for the result of the operation and the second operand specifies the source of the data for the operand.
- COMMENTS describes /explains the function.
  - Any text preceded by Semi-colon(;) or asterisk(*) considered as a comment.

A simple assembly language

- Each Assembly language has two operands, the first operand is always a register which can be any of AREG, BREG, CREG and DREG.
- The second operand refers to a memory word(or constant) using first or second operand specification.

Register	Meaning	Typical Use
AREG	Accumulator Register	Used for arithmetic/logic results
BREG	Base Register	Used for intermediate values
CREG	Counter Register	Used for loop counters
DREG	Data Register	Used for data storage

Assembly Language Statements

- An Assembly program contains three kinds of statements-
- 1. Imperative statements (IC),
- 2. Declarative statements (DS), and
- 3. Assembler Directives (AD).

Imperative Statements

- Indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.
- Opcode reg operand memory operand

```
E.g  MOV AREG, VALUE1   ; Move VALUE1 into AREG
     ADD AREG, VALUE2   ; Add VALUE2 to AREG
     SUB BREG, TEMP     ; Subtract TEMP from BREG
     MUL AREG, BREG     ; Multiply AREG and BREG
     JMP LOOP           ; Jump to label LOOP
```

Declaration statements

- Define data and storage locations.
- [Label] DS <constant>
- [Label] DC <value>
- DS- declare storage: reserves area of memory and associated names with them.
- E.g A DS 1 ;reserves memory areas of 1 word & associate A
- G DS 200 ;reserves a block of 200 memory words.

```
COUNT   DC    5         ; Declare a constant with value 5
  MSG   DC    'HELLO'   ; Declare a string constant
```

- DC-declare constant: constructs memory words containing constants.

Assembler Directives

- Instruct the assembler itself to perform certain actions during the assembly of a program.
- E.g START <constant>
- ; the first word of the object program generated by the assembler should be placed in the memory word with address <constant>.
- END [<operand spec>]
- ; indicates the end of the source program to be assembled.

A simple assembly scheme

Design specification of an assembler

1. Identify the information
2. Select data structure
3. Determine the processing necessary to maintain the information in the data structure
4. Determine the processing necessary to perform the task.

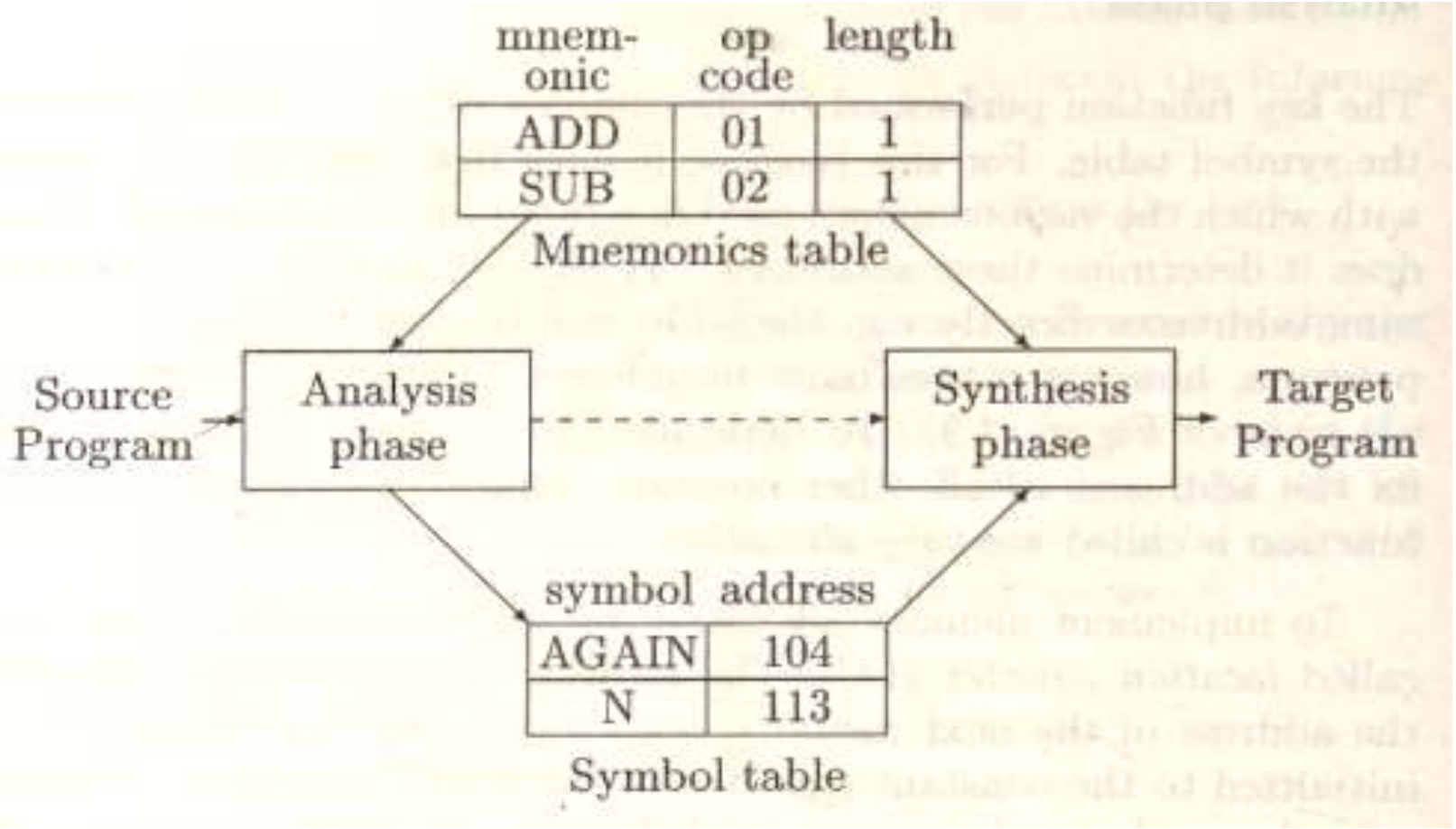
Synthesis phase

- Consider the assembly statement MOVER AREG, N
- How can we synthesize the machine instruction corresponding to the statement?
- Soln: We need to have information about-
- A. address of the memory word with which name A is associated
- B. Machine operation code corresponding to the mnemonic MOVER
- The use of Symbol table(name and address field) and Mnemonics table(mnemonic and opcode fields) is considered during synthesis phase.

Analysis Phase

- Building Symbol table is a key activity of the phase.
- How does it determine the addresses?
- Soln: memory allocation implemented with Location counter(LC).
- Location counter always contains the address of the next memory word. LC is initialized to the constant specified in the START statement.
- LC processing maintains the location counter despite different statements reserve different amount of memory.

Data structures of the assembler



Pass Structure of Assemblers

- Pass is an abstract noun describing the processing performed by the language processor.
- Passes of a language processor is one complete scan of the source program or it's equivalent representation.
- Two pass translation:
  - pass I : analysis of source program (IR)
  - pass II : synthesis of target program (IC)
- Single pass translation:
  - If contain forward reference, left incomplete until the address referenced symbol becomes known. Table of Incomplete Instructions (TII).

Design of a Two pass Assembler

- Advanced assembler Directives
- ORIGIN: indicate that LC should be set to the address given by the address specification. (useful in non-consecutive memory words)
  - Syntax ORIGIN <address spec>
    - Where <address spec> is an <operand spec> or <constant>
- EQU: defines(associates) the symbol to represent the <constant> or the <operand spec>. No LC processing is implied.
  - Syntax <symbol> EQU <operand spec>
- LTORG: to specify where literals should be placed. By default assembler places the literals after the END statement.

Exercise

1. An assembly program contains the statement

```
X       EQU     Y+25
```

Indicated how the EQU statement can be processed if

A) Y is back reference

B) Y is forward reference

2. can the operand expression in an ORIGIN statement contain forward references? If so, outline how the statement can be processed in a two pass assembly scheme.

Solution

1. A. Back Reference

```
Y    EQU    1000
X    EQU    Y + 25   ; look up Y and compute X = 1025 and store it
```

B. Forward Reference

```
X   EQU   Y + 25   ; line 1 (Y undefined)
 ...               ; other code
Y   EQU   1000     ; line 50
```

- Pass-1: store X → expr(Y+25).
- Pass-2: Y=1000 → compute X=1025 → substitute.

Solution

2. Yes. But we must delay evaluating it until pass-2 because it affects the LC(Location Counter).

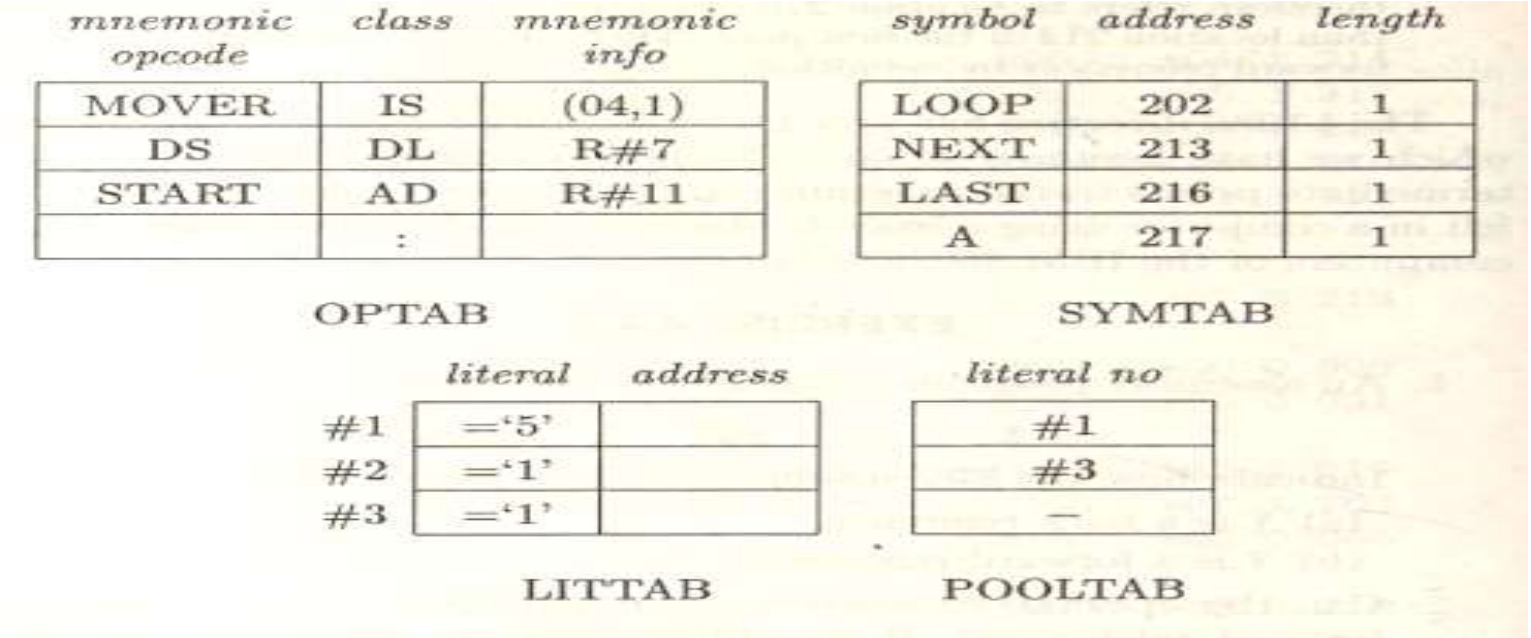
   Example:

```
ORG START + 50     ; line 1 (START undefined in Pass-1)
...
START EQU 2000     ; later
```

- Pass-1: record ORG with expr START+50, continue, write intermediate lines with placeholders.
- Pass-2: evaluate START+50 = 2050, set LC=2050, then generate object code with correct addresses.

Pass I of the Assembler

- Pass I uses the following data structures
- OPTAB: a table of mnemonics opcodes and related information
- SYMTAB: symbol table
- LITTAB: A table of literals used in the program.



mnemonic opcode	*class*	*mnemonic info*
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

OPTAB

symbol	*address*	*length*
LOOP	202	1
NEXT	213	1
LAST	216	1
A	217	1

SYMTAB

	literal	*address*
#1	='5'	
#2	='1'	
#3	='1'	

LITTAB

literal no
#1
#3
–

POOLTAB

Assembler first pass (pass I) Algorithm

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB [1] := 1;
   littab_ptr := 1;
2. While next statement not an END statement
   (a) If label is present then
   this_label := symbol in label field;
   Enter (*this_label*, *locn_cntr*) in SYMTAB.
   (b) If an LTORG statement then
   (i) Process literals LITTAB [POOLTAB [pooltab_ptr]] ...
   LITTAB [littab_ptr−1] to allocate memory. Update
   loc_cntr accordingly.
   (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (iii) POOLTAB [*pooltab_ptr*] := *littab_ptr*;
   (c) If a START or ORIGIN statement then
   loc_cntr := value of operand;

Assembler first pass (pass I) Algorithm

(d) If an EQU statement then
  (i) *this_addr* := value of operand expression;
  (ii) Correct the symtab entry for *this_label* to (*this_label*, *this_addr*).

(e) If a declarative statement then
  (i) *code* := code of the declarative statement;
  (ii) *size* := size of memory area required by DC/DS.
  (iii) *loc_cntr* := *loc_cntr* + *size*;
  (iv) Generate IC '(DL, *code*) ...'.

(f) If an imperative statement then
  (i) *code* := machine opcode from OPTAB;
  (ii) *loc_cntr* := *loc_cntr* + instruction length;

Assembler first pass (pass I) Algorithm

(iii) If operand is a literal then
this_literal := literal in operand field;
LITTAB [*littab_ptr*] := *this_literal*;
littab_ptr := *littab_ptr* + 1;
else (i.e. operand is a symbol)
this_entry := SYMTAB entry number of symbol
in SYMTAB;
Generate IC '(IS, *code*)(S, *this_entry*)';

3. (Processing of END statement)
   (a) Perform step 2(b).
   (b) Generate IC '(AD,02)'.
   (c) Goto pass II.

Intermediate code forms

- IC consists of a set of units with 3 fields.
  1. address
  2. Representation of the mnemonic opcode
  3. Representation of operands
- NB: Variants of intermediate codes, specifically the operand and address fields arise in practice due to the tradeoff between processing efficiency and memory economy.
- We assume that the information in the mnemonic field have the same representation in all variants. (statement class, code)

IC for Imperative Statements

- Two variants arise based on the information in the operand field.

  For simplicity, we assume the address field to contain identical information in both variants.

- Variant I and Variant II

Variant I

- The first operand is represented by a single digit number that is code for a register (1-4 for AREG-DREG) or the condition code itself (1-6 for LT-ANY).
- The second operand or memory operand is represented by a pair of the form (operand class, code)
- Where operand class is one of C(constant), S(symbol) or L(literal).
- For a constant, the code field contains the internal representation of the constant itself. E.g. START 200 is (C,200)
- For a symbol or literal, the code field contains the ordinal number of the operand's entry in SYMTAB or LITTAB.

Variant II

- This variant differs from variant I of the intermediate code in that the operand fields of the source statements are selectively replaced by their processed forms.
- For declarative statements and assembler directives, processing of the operand fields is essential to support LC processing.
- For imperative statements, the operand field is processed only to identify literal references.
- Literals are entered in LITTAB, and are represented as(L, m) in IC.
- Symbolic references in the source statement are not processed at all during pass I.

Example Intermediate code for variant I and II

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   (S,01)
LOOP    MOVER   AREG, A         (IS,05)   (1)(S,01)
          :                       :
        SUB     AREG, ='1'      (IS,02)   (1)(L,01)
        BC      GT, LOOP        (IS,07)   (1)(S,02)
        STOP                    (IS,00)
A       DS      1               (DL, 02)  (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

A) Variant I

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   A
LOOP    MOVER   AREG, A         (IS,05)   AREG, A
          :                       :
        SUB     AREG, ='1'      (IS,02)   AREG, (L,01)
        BC      GT, LOOP        (IS,07)   GT, LOOP
        STOP                    (IS,00)
A       DS      1               (DL,02)   (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

B) Variant II

Pass II of the Assembler

1. *code area address* := address of *code area*;
   pooltab_ptr := 1;
   loc_cntr := 0;
2. While next statement not an END statement
   (a) Clear *machine code buffer*;
   (b) If an LTORG statement
      (i) Process literals in LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [POOLTAB [*pooltab_ptr* +1]] −1 similar to processing of constants in a DC statement, i.e. assemble the literals in the *machine code buffer*.
      (ii) *size* := size of memory area required for literals;

Pass II of the Assembler

(c) If a START or ORIGIN statement then
  (i) *loc_cntr* := value of operand;
  (ii) *size* := 0;
(d) If a declarative statement
  (i) If a DC statement then
    Assemble the constant in the *machine code buffer.*
  (ii) *size* := size of memory area required by DC/DS;

Pass II of the Assembler

(e) If an imperative statement

(i) Get operand address from SYMTAB or LITTAB.

(ii) Assemble instruction in *machine code buffer*.

(iii) *size* := size of instruction;

(f) If *size* $\neq$ 0 then

(i) Move contents of *machine code buffer* to the address *code area address* + *loc_cntr*;

(ii) *loc_cntr* := *loc_cntr* + *size*;

3. (Processing of END statement)

(a) Perform steps 2(b) and 2(f)

(b) Write *code area* into output file.

Listing and Error Reporting

- The basic decision is whether to produce program listing and error reports in pass I or delay these actions until pass II.
- If in pass I: has the advantage that the source program need not be preserved till pass II. Conserves memory and avoids duplicate processing.
- But it can only report certain errors(e.g. syntax and semantic errors like duplicate definitions of symbols)
- Other errors like references to undefined variables can only be reported at the end of the source program.

A Single Pass Assembler

- It is an Assembler that reads the source code only once to translate assembly language to machine code.
- Faster than two-pass assembler because it scans the program only once.
- Useful for small systems or when memory is limited.

Problems of Single Pass Assembly

- Forward Reference (Labels used before being defined) are harder to handle.
- Solved by using symbol table with placeholders and Back patching (filling the correct answer later)
- Segment Registers: segment base addresses may be referenced before they are defined, making it impossible to resolve segment-relative addresses in a single pass without back-patching or multiple passes.

How Single pass assembler works

- 1. Initialize symbol table and location counter (LOCCTR)
- It keeps track of memory addresses.
- 2. read each statement once
- If it is a label, add it to the symbol table with current LC
- If it is an instruction, generate the machine code using the opcode table.
- If it is a directive(like DS, DC), allocate or initialize memory.

Cont…

- 3. handle operands
- If operand is already defined -> use it’s address.
- If operand is forward referenced -> mark for back-patching.
- 4.Increment LOCCTR according to instruction size.
- 5. output machine code
- Write object code directly to output.

Example

```
START 100
LOOP   MOVER AREG, NUM
       ADD   AREG, ONE
       SUB   AREG, TWO
       BC    LT, LOOP
NUM    DC    5
ONE    DC    1
TWO    DC    2
END
```

Example

*assume a **simple hypothetical machine** with one-byte opcodes and 2-byte addresses for simplicity

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

- Start address = 100
(hex or decimal, let's assume decimal).
- Instruction sizes (for simplicity):
  - MOVER = 3 bytes
  - ADD = 3 bytes
  - SUB = 3 bytes
  - BC = 3 bytes
  - DC = 2 bytes

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 1: Initialize

•**LOCCTR** = 100

•**Symbol Table (ST)** = empty

Step 2: Process START directive

START 100

•Set **LOCCTR = 100**

•Symbol table unchanged.

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 3: Process LOOP label + MOVER

•Add **LOOP** to **ST** with address = LOCCTR = 100

Symbol	Address
LOOP	100

•NUM not defined yet → mark as **forward reference**.

•Generate **machine code**:

•Suppose opcode for MOVER AREG = 10

•Address of NUM unknown → placeholder 00

Address 100: 10 00 00 ; MOVER AREG, NUM (placeho

•Increment **LOCCTR** by 3 → **LOCCTR = 103**

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 4: ADD AREG, ONE

•ONE not defined → placeholder

•Opcode for ADD AREG = 20

•Generate machine code:

Address 103: 20 00 00 ; ADD AREG, ONE (placehol

•Increment **LOCCTR** by 3 → **LOCCTR = 106**

•Update **forward reference table** to backpatch ONE

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 5: SUB AREG, TWO

•TWO not defined → placeholder

•Opcode for SUB AREG = 21

Address 106: 21 00 00 ; SUB AREG, TWO (placeholder)

•Increment **LOCCTR** by 3 → **LOCCTR = 109**

•Update forward reference table for TWO

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 6: BC LT, LOOP

•LOOP is already in ST = 100

•Opcode for BC LT = 30

•Generate machine code:

Address 109: 30 00 64

; BC LT, LOOP (jump to 100 decimal = 64 hex)

•Increment **LOCCTR** by 3 → **LOCCTR = 112**

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 7: NUM DC 5

•Add NUM to ST = 112

•Machine code = value 5 (2 bytes):

Address 112: 00 05

•Increment LOCCTR by 2 → **LOCCTR = 114**

•**Backpatch MOVER**

placeholder (NUM = 112 → 70 hex):

Address 100: 10 00 70 ; MOVER AREG, NUM

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 8: ONE DC 1

•Add ONE to ST = 114

•Machine code = 1 (2 bytes):

Address 114: 00 01

•Increment LOCCTR → 116

•Backpatch ADD

placeholder (ONE = 114 → 72 hex):

Address 103: 20 00 72 ; ADD AREG, ONE

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 9: TWO DC 2

•Add TWO to ST = 116

•Machine code = 2 (2 bytes):

Address 116: 00 02

•Increment LOCCTR → 118

•Backpatch SUB placeholder (TWO = 116 → 74 hex):

Address 106: 21 00 74 ; SUB AREG, TWO

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Step 10: END Directive

•Stop assembly

Final Symbol Table

Symbol	Address
LOOP	100
NUM	112
ONE	114
TWO	116

Example

- START 100
- LOOP MOVER AREG, NUM
- ADD AREG, ONE
- SUB AREG, TWO
- BC LT, LOOP
- NUM DC 5
- ONE DC 1
- TWO DC 2
- END

Final Machine code in memory

Address	Machine Code	Instruction
100	10 00 70	MOVER AREG, NUM
103	20 00 72	ADD AREG, ONE
106	21 00 74	SUB AREG, TWO
109	30 00 64	BC LT, LOOP
112	00 05	NUM DC 5
114	00 01	ONE DC 1
116	00 02	TWO DC 2