University Institute of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Bachelor of Engineering

Subject Name: System Programming

Subject Code: CST-315

Chapter-1.2
Assemblers

- Single pass Assembler for Intel x86
- Algorithm of Single Pass Assembler

Chapter-1.2 Assemblers

The CPU consists of an instruction interpreter, a location counter, an instruction register and various working registers and general registers.

- Example:
- To illustrate how these components of the machine structure interact, let us consider a simple computer **(SC-6251). The SC-6251 has four general registers, designated 00, 01, 10 and 11 in the binary notation.**
- The instruction format is as follows:

Operation Code (op)	Register Number (reg)	Memory Location (addr)

- For example, the instruction ADDR 2, 176 Would cause the data stored in memory location 176 to be added to the current contents of the general register 2. **The resulting sum would be left as the new contents of register 2.**

Chapter-1.2 Assemblers

- MEMORY
- The basic unit of memory is a byte- 8 bits of information, i.e, each addressable position in memory can contain eight bits of information.
- BYTE 1 BYTE 8 BITS
- HALFWORD 2 BYTES 16 BITS
- WORD 4 BYTES 32 BITS
- DOUBLEWORD 8 BYTES 64 BITS

The size of memory is up to 2^{24} bytes

Chapter-1.2 Assemblers

Advantages of assembly language

- Since mnemonics replace machine instruction it is easy to write, debug and understand in comparison to machine codes.
- Useful to write lightweight application (in embedded system like traffic light) because it needs fewer codes than high level language.

Disadvantages of assembly language

- Mnemonics in assembly language are in abbreviated form and in large number, so they are hard to remember.
- Program written in assembly language are machine dependent, so are incompatible for different type of machines.
- A program written in assembly language is less efficient to same program in machine language.
- Mnemonics can be different for different machines according to manufacturer's so assembly language suffers from the defect of non-standardization

Chapter-1.2 Assemblers

ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

An assembly language provides the following five basic facilities that simplifies programming:

1. **Mnemonic operation code**: A small word that acts as an identifier for the instruction. The mnemonics are written in code segment. In following examples mov, sub, add, jmp, call, and mul are the mnemonics

a. MOV Move/assign one value to another (label)

b. SUB Subtract one value from another

c. ADD Adds two values

d. JMP Jump to a specific location

e. CALL Call a procedure/module

f. MUL Multiply two values

g. **Example: ADD R1, R2**

Chapter-1.2 Assemblers

2. **Symbols / Labels:** Symbols or labels are just **like variables** in any other language.

Example int a=5; Symbol table is used to handle such things

3. **Data declarations:** Data can be declared in a variety of notations, including **the decimal notation.** It avoids the need to manually specify constants in representations that a computer can understand, for example, specify -5 as (11111011)
4. **Location Counter (LC):** Indicate next instruction to be executed
5. **Literals: Constant value.**

Example: R1=a+8, 8 is literal. Literals are stored in literal table data structure.

Chapter-1.2 Assemblers

ASSEMBLY LANGUAGE STATEMENTS

1. **Imperative Statements:** indicates an action to be performed during the execution of the assembled program.

Each imperative statement typically translates into one machine instruction.

2. **Declaration Statements:** the syntax of declaration statements is:

[Label] DS

[Label] DC

The DS (short) for declare storage statement reserves areas of memory and associates names with them .e.g.

A DS 1 The above statement reserves a memory area of 1 word and associates the name A with it.

The DC (short for declare constant) statement declare memory words containing constants.

3. **Assembler Directives:** Assembler directives are Pseudo-Instructions. They provide instructions to the assembler itself. They are not translated into machine operation codes

Chapter-1.2 Assemblers

Basic assembler directives are:

ASSUME (The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment.)

DB - Defined Byte (DB directive is used to declare a byte type variable or to store a byte in memory location)

DD - Defined Double Word

DQ - Defined Quad Word

DT - Define Ten Byte

Chapter-1.2 Assemblers

DATA STRUCTURE/TABLES USED BY ASSEMBLER

1. Machine-Opcode Table (MOT) or (Operation Code Table) or Mnemonics table :

A mnemonic is an abbreviation for an operation.

This table consists of the fields: **Name of mnemonic, binary value, instruction length, format of instruction.**

- MOT table is used to look up mnemonic operation codes and translate them to their machine language equivalents

Chapter-1.2 Assemblers

Example:

Name of Mnemonic	Binary Values	Instruction length	Instruction Format
MUL	010111010101	4	CISC
ADD	11010101010	4	CISC
SUB	01010010010	4	CISC
Load	011001010101	4	CISC

Machine-Opcode Table (MOT) or (Operation Code Table) or Mnemonics table

Chapter-1.2 Assemblers

DATA STRUCTURE/TABLES USED BY ASSEMBLER

2. **Pseudo Opcode Table (POT):**

This table consists of the fields:

Name of Pseudo code

Action associated with Pseudo code.

POT is the fixed length table. This direct assembler what action should be taken corresponding to any pseudo code given in the program.

Chapter-1.2 Assemblers

Pseudo	Actions
START	It is used to specify the starting execution of a program.
USING	It specifies the base table register that is been used.
DROP	It is used to remove the register used in the base table.

Pseudo Opcode Table (POT)

Chapter-1.2 Assemblers

DATA STRUCTURE/TABLES USED BY ASSEMBLER

3. Symbol Table (ST)

Symbol table is used for keeping the track of symbol that are defined in the program.

It is used to give a location for a symbol specified.

The assembler creates the symbol table section for the object file. It makes an entry in the symbol table for each symbol that is defined or referenced in the input file and is needed during linking

- In pass 1, whenever a symbol is defined corresponding entry is made in symbol table.
- In pass2, symbol table is used for generating machine code of a symbol.

Chapter-1.2 Assemblers

Symbol	Description	Address assigned in RAM
A	Variable A defined in Assembly language.	0040 0000
B	Variable B defined in Assembly language	0040 0004

Symbol Table

Chapter-1.2 Assemblers

DATA STRUCTURE/TABLES USED BY ASSEMBLER

4. Literal Table (LT)

Literal table is used for keeping track of literals that are encountered in the programs.

- We directly specify the value, literal is used to give a location for the value.
- In pass 1, whenever a Literal is defined and for entry is made in Literal table.
- In pass2, Literal table is used for generating binary code of a Literal.

Literals are always encountered in the operand field of an instruction.

Chapter-1.2 Assemblers

Value of literal	Length of literal
28	4 byte
15	4 byte

Literal Table

Chapter-1.2 Assemblers

DATA STRUCTURE/TABLES USED BY ASSEMBLER

5. Base Table (BT)

This store the information of available register in hardware of the system.
Example, Register1 is free but R2 and R3 are not free.

Register	Available
R1	Free
R2	Not Free
R3	Not Free

Simple assembly language code for addition of 2 numbers and storing the result in location:-

```
PGM     START   0
BEGIN   BALR    15
        USING   BEGIN+2, 15
        L       1, FOUR
        A       1, FIVE
        ST      1, TEMP
FOUR    DL      F'4'
FIVE    DC      F'3'
TEMP    DS      1F
END
```

identifiers to access registers and parts thereof

Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH, AL	CH, CL	DH, DL	BH, BL	SPL	BPL	SIL	DIL

Single pass Assembler for Intel x86

- **General-Purpose Registers (GPR) - 16-bit naming conventions**
- The 8 GPRs are as follows:
- Accumulator register (AX). Used in arithmetic operations
- Counter register (CX). Used in shift/rotate instructions and loops.
- Data register (DX). Used in arithmetic operations and I/O operations.
- Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).
- Stack Pointer register (SP). Pointer to the top of the stack.
- Stack Base Pointer register (BP). Used to point to the base of the stack.
- Source Index register (SI). Used as a pointer to a source in stream operations.
- Destination Index register (DI). Used as a pointer to a destination in stream operations.

Single pass Assembler for Intel x86

- The 6 Segment Registers are:
- Stack Segment (SS). Pointer to the stack ('S' stands for 'Stack').
- Code Segment (CS). Pointer to the code ('C' stands for 'Code').
- Data Segment (DS). Pointer to the data ('D' comes after 'C').
- Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').
- F Segment (FS). Pointer to more extra data ('F' comes after 'E').
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

Single pass Assembler for Intel x86

Labels

- Act as place markers
- marks the address (offset) of code and data
- Easier to memorize and more flexible

  eg. mov ax, [0020]→mov ax, val
- Follow identifier rules

Data label

- must be unique

  example: myArray BYTE 10
- Code label (ends with a colon)
- target of jump and loop instructions

  example: L1: mov ax, bx ... jmp L1

Single pass Assembler for Intel x86

Reserved words and identifiers

- Reserved words cannot be used as identifiers
- Instruction mnemonics, directives, type attributes, operators, predefined symbols
- Identifiers
  - 1-247 characters, including digits
- case insensitive (by default)
- first character must be a letter, _, @, or $

examples: var1 Count $first _main MAX open_file @@myfile xVal _12345

Single pass Assembler for Intel x86

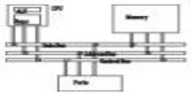
- Comments • Comments are good! • explain the program's purpose • tricky coding techniques • application-specific explanations • Single-line comments • begin with semicolon (;) • block comments • begin with COMMENT directive and a programmer-chosen character and end with the same programmer-chosen character COMMENT ! This is a comment and this line is also a comment !
- directive marking a comment comment copy definitions from Irvine32.inc code segment. 3 segments: code, data, stack beginning of a procedure destination source defined in Irvine32.inc to end a program marks the last line and define the startup procedure Example: adding/subtracting integers TITLE Add and Subtract (AddSub.asm) ; This program adds and subtracts 32-bit integers.
- Defining data
- Intrinsic data types (1 of 2) • BYTE, SBYTE • 8-bit unsigned integer; 8-bit signed integer • WORD, SWORD • 16-bit unsigned & signed integer • DWORD, SDWORD • 32-bit unsigned & signed integer • QWORD • 64-bit integer • TBYTE • 80-bit integer
- Intrinsic data types (2 of 2) • REAL4 • 4-byte IEEE short real • REAL8 • 8-byte IEEE long real • REAL10 • 10-byte IEEE extended real
- Data definition statement • A data definition statement sets aside storage in memory for a variable. • May optionally assign a name (label) to the data. • Only size matters, other attributes such as signed are just reminders for programmers. • Syntax: [name] directiveinitializer [,initializer] . . . At least one initializer is required, can be ? • All initializers become binary data in memory
- Integer constants • [{+|-}] digits [radix] • Optional leading + or – sign • binary, decimal, hexadecimal, or octal digits • Common radix characters: • h– hexadecimal • d– decimal (default) • b– binary • r– encoded real • o– octal Examples: 30d, 6Ah, 42, 42o, 1101b Hexadecimal beginning with letter: 0A5h

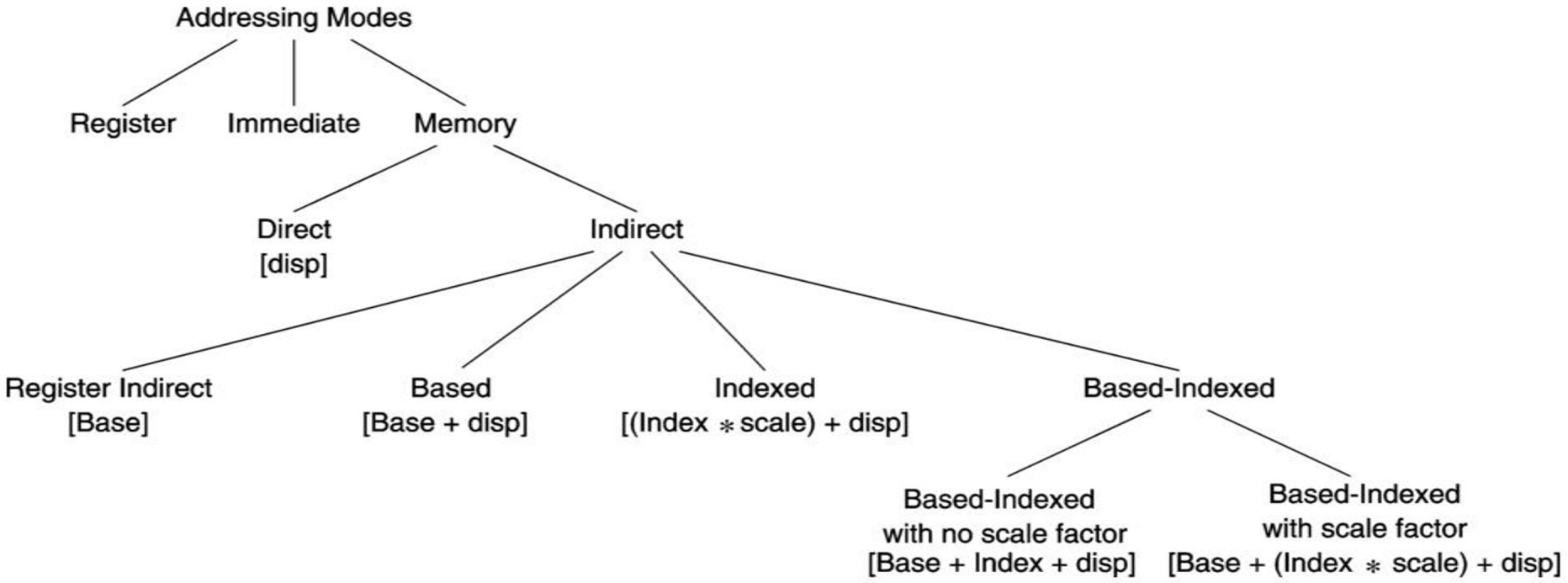
Single pass Assembler for Intel x86

- Integer expressions • Operators and precedence levels: • Examples:
- Real number constants (encoded reals) • Fixed point v.s. floating point • Example 3F800000r=+1.0,37.75=42170000r • double 1 8 23 S E M ±1.bbbb×2 (E-127) 1 11 52 S E M
- Real number constants (decimal reals) • [sign]integer.[integer][exponent] sign → {+|-} exponent → E[{+|-}]integer • Examples: 2. +3.0 -44.2E+05 26.E5
- Character and string constants • Enclose character in single or double quotes • 'A', "x" • ASCII character = 1 byte • Enclose strings in single or double quotes • "ABC" • 'xyz' • Each character occupies a single byte • Embedded quotes: • ‘Say "Goodnight," Gracie’ • "This isn't a test"
- Defining BYTE and SBYTE Data Each of the following defines a single byte of storage: value1 BYTE 'A‘ ; character constant value2 BYTE 0 ; smallest unsigned byte value3 BYTE 255 ; largest unsigned byte value4 SBYTE -128 ; smallest signed byte value5 SBYTE +127 ; largest signed byte value6 BYTE ? ; uninitialized byte A variable name is a data label that implies an offset (an address).
- Defining multiple bytes Examples that use multiple initializers: list1 BYTE 10,20,30,40 list2 BYTE 10,20,30,40 BYTE 50,60,70,80 BYTE 81,82,83,84 list3 BYTE ?,32,41h,00100010b list4 BYTE 0Ah,20h,‘A’,22h
- Defining strings (1 of 2) • A string is implemented as an array of characters • For convenience, it is usually enclosed in quotation marks • It usually has a null byte at the end • Examples: str1 BYTE "Enter your name",0 str2 BYTE 'Error: halting program',0 str3 BYTE 'A','E','I','O','U' greeting1 BYTE "Welcome to the Encryption Demo program " BYTE "created by Kip Irvine.",0 greeting2 \ BYTE "Welcome to the Encryption Demo program " BYTE "created by Kip Irvine.",0
- Defining strings (2 of 2) • End-of-line character sequence: • 0Dh = carriage return • 0Ah = line feed str1 BYTE "Enter your name: ",0Dh,0Ah BYTE "Enter your address: ",0 newLine BYTE 0Dh,0Ah,0 Idea: Define all strings used by your program in the same area of the data segment.

Single pass Assembler for Intel x86

Addressing Modes





Chapter-1.2 Assemblers

Assembly - Addressing Modes

Most assembly language instructions require operands to be processed. An operand address provides the location, where the data to be processed is stored. Some instructions do not require an operand, whereas some other instructions may require one, two, or three operands.

When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source. Source contains either the data to be delivered (immediate addressing) or the address (in register or memory) of the data. Generally, the source data remains unaltered after the operation.

The three basic modes of addressing are –

1. Register addressing
2. Immediate addressing
3. Memory addressing

Chapter-1.2 Assemblers

1. Register Addressing

In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

For example,

```
MOV DX, TAX_RATE        ; Register in first operand
MOV COUNT, CX           ; Register in second operand
MOV EAX, EBX            ; Both the operands are in registers
```

As processing data between registers does not involve memory, it provides fastest processing of data.

Chapter-1.2 Assemblers

2. Immediate Addressing

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

For example,

```
BYTE_VALUE      DB     150     ; A byte value is defined
WORD_VALUE      DW     300     ; A word value is defined
ADD             BYTE_VALUE,    65          ; An immediate operand 65 is added
MOV             AX,            45H     ; Immediate constant 45H is transferred to AX
```

Chapter-1.2 Assemblers

3. Direct Memory Addressing

- When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the **DS register** and an offset value. This offset value is also called **effective address**.
- In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

Chapter-1.2 Assemblers

3. Direct Memory Addressing

- In direct memory addressing, one of the operands refers to a memory location and the

For example,

```
ADD     BYTE_VALUE, DL  ; Adds the register in the memory location
MOV     BX, WORD_VALUE  ; Operand from the memory is added to register
```

Chapter-1.2 Assemblers

Direct-Offset Addressing

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data

```
BYTE_TABLE DB  14, 15, 22, 45      ; Tables of bytes
WORD_TABLE DW  134, 345, 564, 123  ; Tables of words
```

The following operations access data from the tables in the memory into registers −

```
MOV CL, BYTE_TABLE[2]   ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2  ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3]   ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3  ; Gets the 4th element of the WORD_TABLE
```

Chapter-1.2 Assemblers

4. Indirect Memory Addressing

This addressing mode utilizes the computer's ability of *Segment: Offset* addressing.

Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

- **Indirect addressing is generally used for variables containing several elements like, arrays.**
- Starting address of the array is stored in, say, the **EBX register.**
- The following code snippet shows how to access different elements of the variable.

Chapter-1.2 Assemblers

```
MY_TABLE TIMES 10 DW 0  ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE]     ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110          ; MY_TABLE[0] = 110
ADD EBX, 2              ; EBX = EBX +2
MOV [EBX], 123          ; MY_TABLE[1] = 123
```

Chapter-1.2 Assemblers

Difference between Pass 1 and Pass 2 Assembler

Pass 1 Assembler	Pass 2 Assembler
Performs single pass	Perform two passes
In first pass itself complete assembly program is converted into machine code i.e., it collects the symbols or labels, pseudo keyword, operators, literals in their respective table data structure. After this **i.e within same pass immediately** fetch the stored information from table and converts into machine code. **Both of these processes could be done together.**	For same operation it performs two passes. First pass it collects the collects the symbols or labels, pseudo keyword, operators, literals in their respective table data structure. Now run Second pass and converts everything into binary format.

Chapter-1.2 Assemblers

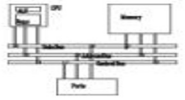
4. Difference between X85 and X86 Architecture

The Intel 8085 is an 8 bit microprocessor created in 1976	The Intel 8086 is a 16 bit microprocessor created in 1978
8085 is a 8 bit processor, number of flags are 5 and memory capacity is 64KB	8086 is a 16 bit processor, number of flags are 9 and memory capacity is 1 MB
8085 doesn't have an instruction queue	8086 has an instruction queue.
8085 doesn't support a pipelined architecture	8086 supports a pipelined architecture.

Table1: Assembly Language Instructions for X85 and X86

Single pass Assembler for Intel x86

Operand types

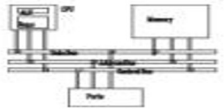


- Three basic types of operands:
  - Immediate – a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - Register – the name of a register
    - register name is converted to a number and encoded within the instruction
  - Memory – reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location

45

Single pass Assembler for Intel x86

Instruction operand notation



Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

Single Pass Storage algorithm

One-pass assemblers are used when

it is necessary or desirable to avoid a second pass over the source program the external storage for the intermediate file between two passes is slow or is inconvenient to use Main problem: forward references to both data and instructions.

Data Structures Required are:

- **Op code table**
- **Symbol table**

pass 1: loop until the end of the program

1. Read in a line of assembly code
2. Assign an address to this line increment N (word addressing or byte addressing)
3. Save address values assigned to labels in symbol tables
4. Process assembler directives constant declaration space reservation

Single Pass Storage algorithm

Algorithm for Pass 1 assembler:

```
 begin
 if starting address is given
LOCCTR = starting address;
else
LOCCTR = 0;
while OPCODE != END do ;; or EOF
begin
read a line from the code
if there is a label
 if this label is in SYMTAB, then error
else insert (label, LOCCTR) into SYMTAB
search OPTAB for the op code
if found
LOCCTR += N ;; N is the length of this instruction (4 for MIPS)
else if
this is an assembly directive
update LOCCTR as directed
else error
write line to intermediate file
end
program size = LOCCTR - starting address;
end
```

References

- PPT - Intel x86 Assembly Fundamentals PowerPoint Presentation, free download - ID:4403217 (slideserve.com)
- Introduction to software | computer Software (w3htmlschool.com)
- Types of Software – GeeksforGeeks

THANK YOU