SYSTEM PROGRAMMING MID SEM

COMPLETELY REFERENCED FROM THE DONOVAN BOOK INCLUDING DIAGRAMS & EXAMPLES & STRICTLY ADHERING TO THE SYLLABUS

---

1. Foundations of System Programming

1.1 Introduction and Definition

- **Distinction Between System and Application Programming**
  System programming involves developing software that interacts closely with hardware and manages system resources. Unlike application programming—which creates software for end-user tasks—system programming is about constructing the very infrastructure (e.g., operating systems, compilers, device drivers) that supports all other applications.
    - **Examples:** Operating system kernels, assemblers, and runtime libraries.
    - **Emphasis:** A deep understanding of both hardware mechanisms and software abstractions is essential.`

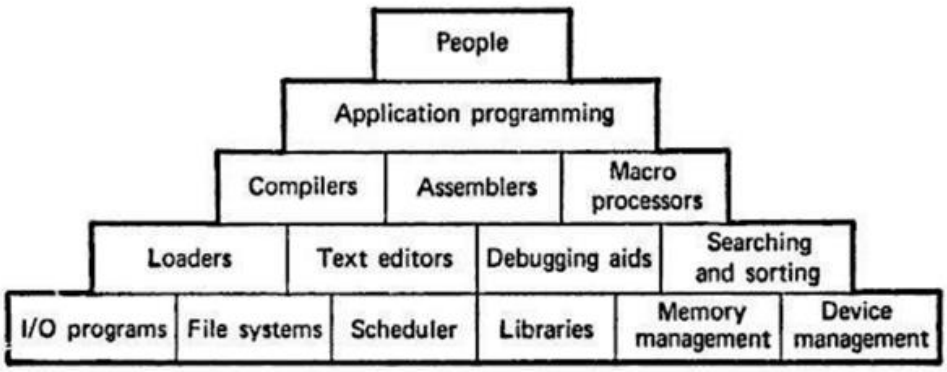


FIGURE 1.1 Foundations of systems programming

- **Core Goals and Challenges**
    - **Efficiency and Performance:** Code must be optimized for speed and minimal resource consumption.
    - **Direct Hardware Control:** Provides precise management of resources like memory, I/O devices, and processor cycles.
    - **Robustness and Stability:** The software must handle errors gracefully and operate reliably even under stress.
    - **Layered Abstraction:** Although system programming interacts with low-level hardware, it employs layers of abstraction to simplify complexity while maintaining control.

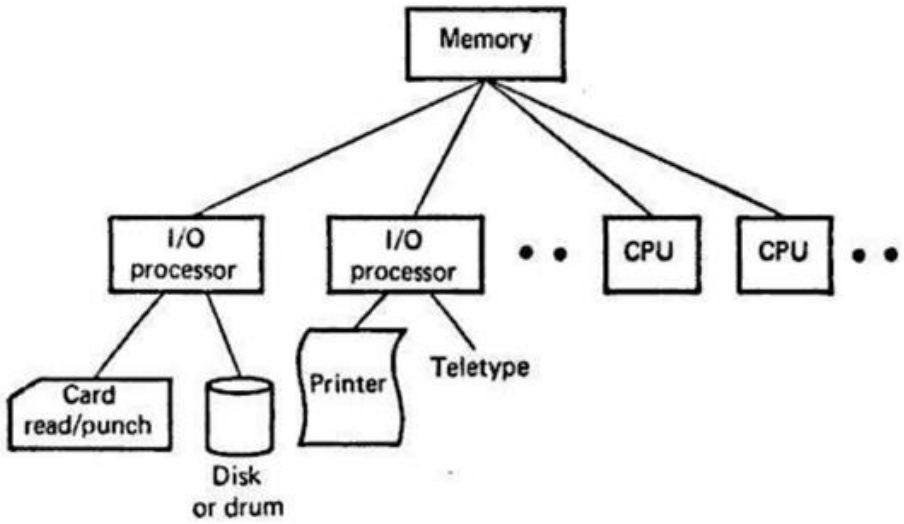
1.2 Historical and Conceptual Background

- **Evolution of System Programming**

- **Early Days:** Initially, programmers wrote in machine or assembly language, manually controlling every hardware detail.
- **Transition to Higher-Level Languages:** As systems grew in complexity, languages like C emerged to balance low-level control with higher-level abstraction.
- **Impact on Modern Computing:** The evolution has led to more sophisticated operating systems and compilers, setting the groundwork for modern multitasking, security, and performance optimization techniques.

- **Key Milestones in the Discipline**
  - **Birth of Operating Systems:** The necessity to manage multiple programs and devices led to the development of early operating systems.
  - **Development of Compilers and Linkers:** These tools automated the translation of high-level code into machine instructions, reducing the manual effort required in programming.
  - **Concept of Layered Architecture:** This idea, heavily emphasized in the book, underpins how modern systems are organized—from hardware up through various software layers.

1.3 Fundamental Concepts and Techniques



- **Machine Model Abstraction**
  Every computer system, regardless of its underlying hardware, can be modeled as having a central processor, a memory system, and I/O facilities. This abstraction simplifies reasoning about:
  - **Instruction Execution:** How the CPU processes operations.
  - **Data Movement:** How data is transferred between various levels of memory and devices.

    - **Resource Management:** How resources are allocated and controlled across the system

- **Core Techniques**
    - **Memory Management:**
        - Allocation and deallocation strategies.
        - Memory protection and segmentation to avoid conflicts.
        - Use of virtual memory concepts for efficient resource utilization.
    - **Process Control and Scheduling:**
        - Techniques for managing concurrent processes, context switching, and CPU scheduling.
        - Handling process states and transitions between running, waiting, and blocked.
    - **I/O Management:**
        - Strategies for handling input/output operations including interrupts, polling, and Direct Memory Access (DMA).
        - Design of device drivers that act as the interface between hardware and software.
    - **Concurrency and Synchronization:**
        - Mechanisms such as locks, semaphores, and monitors to ensure safe concurrent execution.
        - Avoiding race conditions and ensuring deadlock-free operation.

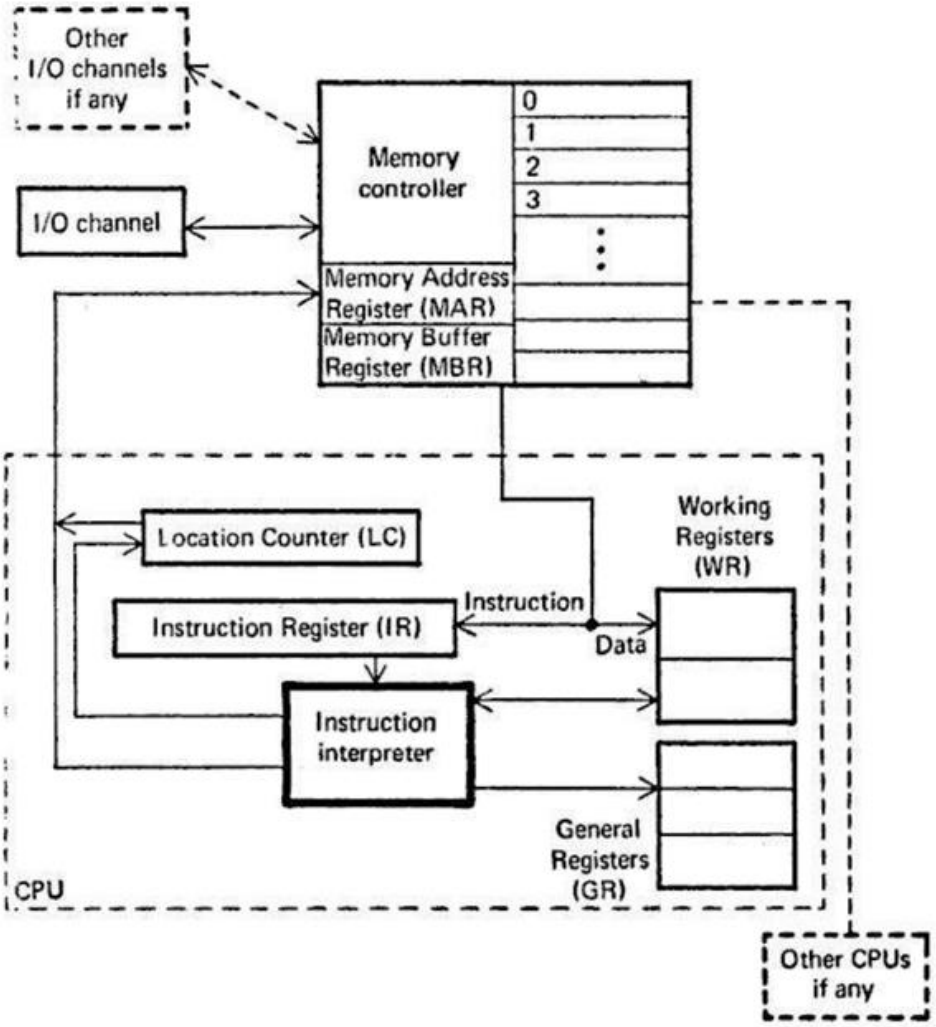
---

2. General Machine Structure

2.1 Overview of Machine Architecture

- **Basic Components of a Computer System**
  A computer can be divided into several interrelated parts:
    - **Central Processing Unit (CPU):** Executes instructions and performs calculations.
    - **Memory:** Ranges from high-speed registers and cache to larger, slower main memory and non-volatile secondary storage.
    - **Input/Output (I/O) Devices:** Facilitate communication between the computer and the external environment (keyboards, monitors, disks, networks).
    - **Bus Systems:** Serve as communication channels connecting these components.

- **Simplified Machine Model**
  Donovan's text uses a simplified machine model to help explain:
    - **Instruction Cycle:** How instructions are fetched, decoded, and executed.
    - **Data Paths:** How information flows between CPU, memory, and I/O.
    - **Abstraction Layers:** How higher-level operations are built upon basic hardware functions.



2.2 The CPU: Heart of the Machine

- **Internal Components and Their Roles**
    - **Arithmetic Logic Unit (ALU):**
        - Performs all arithmetic and logical operations.
        - Handles tasks ranging from simple calculations to complex decision-making.
    - **Control Unit:**

        - Directs the sequence of operations by fetching, decoding, and executing instructions.
        - Coordinates between the CPU, memory, and I/O devices.
    - **Registers:**
        - Serve as the CPU's internal storage, holding data, addresses, and instructions temporarily.
        - Include special-purpose registers (e.g., program counter, status register) that control program flow.
- **Detailed Look at the Fetch-Decode-Execute Cycle**
    - **Fetch:**
        - The CPU retrieves an instruction from main memory using the program counter.
        - This step is critical as it determines what operation will be performed next.
    - **Decode:**
        - The instruction is interpreted to understand which operation is needed and what operands to use.
        - Involves breaking down the instruction into opcodes and operands.
    - **Execute:**
        - The decoded instruction is performed.
        - This could involve arithmetic calculations, data movement, or interaction with I/O.
    - **Analogy:** Consider the process like a chef in a kitchen: fetching ingredients (instructions), reading the recipe (decoding), and preparing the dish (execution).

2.3 Memory Hierarchy and Its Importance

- **Levels of Memory in a System**
    - **Registers:**
        - Extremely fast but very limited in number.
    - **Cache Memory:**
        - Provides a bridge between the high-speed registers and slower main memory.
        - Often split into multiple levels (L1, L2, and sometimes L3) for efficiency.
    - **Main Memory (RAM):**
        - Used for storing active programs and data.

        - Its speed and capacity are balanced against cost.
    - **Secondary Storage:**
        - Non-volatile memory used for long-term storage, such as hard drives and SSDs.
    - **Virtual Memory Concepts:**
        - Techniques such as paging and segmentation allow systems to use disk space to extend RAM, managing larger programs than physically possible with available RAM.
- **Trade-Offs and Design Considerations**
    - **Speed vs. Capacity:** Faster memory is generally more expensive and has less capacity.
    - **Cost vs. Performance:** Designers must balance the cost of memory with the need for speed in critical operations.
    - **Impact on System Performance:** The overall system speed is often determined by how efficiently data is moved through these levels.

2.4 Input/Output Systems and Bus Architecture

- **I/O Systems Detailed**
    - **Device Interfaces:**
        - The methods by which peripherals connect and communicate with the CPU.
        - These interfaces standardize how data is exchanged and commands are issued.
    - **Interrupts:**
        - Allow devices to signal the CPU when immediate attention is needed.
        - This mechanism minimizes the need for the CPU to constantly poll devices.
    - **Direct Memory Access (DMA):**
        - Permits peripherals to transfer data directly to/from memory without burdening the CPU with routine data movement.
        - Enhances overall system efficiency, especially for high-speed data transfer tasks.
- **Bus Architecture and Its Role**
    - **System Bus:**
        - The primary communication pathway that connects the CPU, memory, and I/O devices.

        - Critical for synchronizing data transfers and maintaining system coherence.
    - **I/O Bus:**
        - Often designed for peripherals, handling lower data transfer rates.
        - Sometimes separate from the system bus to optimize specialized functions.
    - **Integration and Communication:**
        - The efficient design of bus systems ensures that all components work in unison.
        - Emphasizes timing, bandwidth, and error handling to reduce data bottlenecks.

---

3. Simplified Machine Architecture & Its Components

3.1 Overview and Concept (Some part of this similar and repetitive of the earlier section but since this a separate topic in the syllabus so covered it again)

- **Definition & Purpose**
  A simplified machine architecture is a conceptual model that strips away extraneous details to focus on the fundamental components of a computer. This model helps system programmers and computer engineers understand how instructions are processed and how data flows through the system.
    - *Key Insight:* Even though real machines are complex, the simplified model highlights the essential elements common to nearly every computer system.

3.2 Core Components of the Simplified Architecture

3.2.1 Central Processing Unit (CPU)

- **Functionality:**
    - The CPU is the "brain" of the computer. It is responsible for executing instructions and performing arithmetic and logical operations.
- **Key Sub-components:**
    - **Arithmetic Logic Unit (ALU):** Performs mathematical and logical operations.
    - **Control Unit:** Fetches, decodes, and directs the execution of instructions.
    - **Registers:**
        - *Working Registers:* Serve as temporary storage for data during computation.
        - *General-purpose Registers:* Used by programmers to store operands and intermediate results.
    - **Instruction Cycle:**

        - **Fetch:** Retrieve an instruction from memory.
        - **Decode:** Interpret the instruction.
        - **Execute:** Carry out the instruction.

3.2.2 Memory System

- **Primary Role:**
    - Stores both data and instructions in binary form.
- **Organization:**
    - **Bit, Byte, and Word:**
        - Information is stored as bits (0s and 1s) grouped into bytes (8 bits) and words (typically 16 to 32 bits, depending on the machine).
    - **Addressing:**
        - Every memory location has an address, which can be thought of as a mailbox containing data.
        - Instructions and data share the same memory medium, and their interpretation depends on the context (data vs. instruction).

3.2.3 Input/Output (I/O) Subsystem

- **Function:**
    - Facilitates communication between the computer's internal components and the external environment (keyboards, printers, disks, etc.).
- **Distinct Elements:**
    - **I/O Processors/Channels:**
        - These specialized units manage data transfer between memory and peripheral devices.
        - They often work asynchronously with the CPU, meaning I/O operations can occur in parallel with other processes.
    - **Interrupts:**
        - Devices can signal the CPU (via interrupts) when they require attention, reducing the need for constant polling.

3.2.4 Bus Systems

- **Definition & Role:**
    - Buses are the physical communication pathways that connect the CPU, memory, and I/O devices.
- **Types:**
    - **System Bus:**

        - Connects the CPU and main memory.
    - **I/O Bus:**
        - Connects peripheral devices to the system, often with different speed and bandwidth considerations.
- **Importance:**
    - Ensures that data is transmitted efficiently and synchronously between components.

3.3 Simplification Benefits

- **Easier Reasoning:**
    - The simplified architecture allows one to focus on how basic operations—such as instruction execution and data movement—occur without getting overwhelmed by hardware-specific complexities.
- **Foundation for Advanced Topics:**
    - Understanding the simplified model is crucial before delving into more complex subjects like addressing modes, pipelining, or advanced I/O strategies.

---

4. System Software & Its Components

4.1 Introduction and Importance

- **Definition:**
    - System software consists of programs that manage and control the hardware components of a computer, thereby providing a platform for application software.
- **Key Roles:**
    - It bridges the gap between the hardware and the user's applications by managing resources, ensuring efficiency, and maintaining system stability.
- **Historical Perspective:**
    - Early computers were programmed directly in machine language. As systems evolved, programs like assemblers, loaders, and operating systems were developed to streamline and automate these tasks.

4.2 Major Components of System Software

4.2.1 Assemblers

- **Function:**
    - Translate mnemonic assembly language (human-readable code) into machine language (binary code).
- **Key Aspects:**

    - **Source Program:** The assembly language code written by the programmer.
    - **Object Program:** The machine language output generated by the assembler.
    - **Advantages:**
        - Makes programming more accessible and less error-prone compared to coding directly in binary.
- **Historical Note:**
    - Initially, programmers had to write in machine language using ones and zeros; the introduction of assembly language was a significant advancement in system programming.

4.2.2 Loaders

- **Purpose:**
    - Load object programs from secondary storage into main memory and prepare them for execution.
- **Functions:**
    - **Relocation:**
        - Adjust addresses so that a program can be loaded into an arbitrary memory location.
    - **Linking:**
        - Resolve symbolic references between different object modules or subroutines.
    - **Efficiency:**
        - The loader minimizes wasted memory by replacing the translator (such as the assembler) in core during execution.

4.2.3 Macro Processors

- **Role:**
    - Provide a way to define shorthand notations (macros) for repeated code segments.
- **How They Work:**
    - The macro processor scans the source program for macro calls and replaces them with the corresponding macro definitions.
- **Benefits:**
    - Reduces the need to write repetitive code.

    - Facilitates system specialization (for example, in configuring operating systems for particular installations).

- **Historical Evolution:**
    - Macro facilities have grown in importance not only for code abbreviation but also for the dynamic configuration of system components.

4.2.4 Compilers and Interpreters

- **Compilers:**
    - Translate high-level language programs (e.g., FORTRAN, COBOL, PL/I) into object code.
    - Must handle complex language features such as pointer management and structured data.
- **Interpreters:**
    - Execute source code directly, translating it on the fly into actions that mimic machine code execution.
- **Modern Requirements:**
    - Modern compilers are required to closely interact with operating systems (e.g., managing hardware interrupts) and to provide robust error checking and optimization features.

4.2.5 Operating Systems

- **Definition:**
    - The operating system is the core software that manages computer resources (processors, memory, I/O devices) and provides services to application programs.
- **Primary Functions:**
    - **Resource Allocation:**
        - Schedules processor time, manages memory, and controls peripheral devices.
    - **User Interface:**
        - Provides the framework (command-line or graphical) for users to interact with the system.
    - **Protection and Error Handling:**
        - Ensures that users and programs do not interfere with each other and that errors are managed gracefully.
- **Components within the OS:**
    - **Traffic Controller/Scheduler:**

        - Determines which tasks or jobs are executed when.
    - **Memory Management Module:**
        - Controls allocation and protection of memory.
    - **File System:**
        - Manages data storage, retrieval, and security.
- **User Viewpoint:**
    - From the perspective of the user, the operating system abstracts the complex details of hardware management and provides a set of useful facilities.

4.2.6 Other Supporting Utilities

- **Subroutine Libraries and Linkage Editors:**
    - These provide pre-written routines (for tasks like mathematical calculations or string processing) that can be linked to user programs.
- **Utility Programs:**
    - Programs like sort/merge utilities, tape copy routines, and debugging tools support overall system functionality and ease of programming.
- **Impact on Productivity:**
    - The evolution of these tools has allowed computer systems to become vastly more efficient, enabling the use of "ready-made" packages and facilitating multiprogramming and multitasking.

4.3 Evolution and Interrelation of System Software Components

- **Integration and Relocation:**
    - The move from standalone assemblers to integrated systems (with relocating loaders and dynamic linking) highlights the evolution toward more flexible and efficient system software.
- **User and Developer Perspectives:**
    - While the operating system shields the user from hardware complexities, system software like assemblers, loaders, and macro processors provide the tools that allow programmers to develop complex, reliable applications without reinventing basic mechanisms.
- **Overall Impact:**
    - The effectiveness and sophistication of system software directly influence the productivity of the computer system as a whole. In modern computing, these components are critical for ensuring optimal performance, security, and maintainability.

---

5. Low-Level Languages

5.1 Overview and Definitions

- **Machine Language:**
    - **Definition:** The most basic form of programming language consisting of binary (or hexadecimal) codes that a processor executes directly.
    - **Characteristics:**
        - **Hardware-Dependent:** Each machine's architecture dictates the binary instructions.
        - **Efficient but Unreadable:** Offers maximum speed and control, yet difficult for humans to write and debug.
        - **No Abstraction:** Operates at the level of bits and bytes, handling raw data.
    - **Example:**
        - A machine language instruction might appear as a series of 0s and 1s, such as 01011000 00100000 00010011 10011100, which directly corresponds to a load operation.
- **Assembly Language:**
    - **Definition:** A symbolic representation of machine language that uses mnemonics instead of raw binary codes.
    - **Features:**
        - **Mnemonic Codes:** Replace numeric opcodes with symbols like L (load), A (add), and ST (store).
        - **Symbolic Addressing:** Labels are used to represent memory addresses, making code easier to understand.
        - **Assembler Dependency:** An assembler converts the assembly code into machine code.
    - **Benefits:**
        - **Improved Readability:** Easier to comprehend and maintain compared to pure binary.
        - **Control & Efficiency:** Retains low-level hardware control, allowing for precise optimization.
    - **Real-World Example:**
        - An instruction such as L 1,FIVE tells the processor to load the data located at the address marked by the label FIVE into register 1.
- **Summary of Low-Level Languages:**
    - Both machine language and assembly language provide a foundation for controlling computer hardware. While machine language is extremely efficient,

assembly language introduces symbolic notation to make programming more manageable without sacrificing control.

---

6. Basics of an Assembly Language

6.1 Core Components and Structure

- **Mnemonics:**
    - **Definition:** Short, symbolic codes representing processor instructions.
    - **Examples:**
        - L (Load), A (Add), ST (Store), BCT (Branch on Count).
    - **Usage:**
        - They abstract away the underlying binary opcodes, allowing the programmer to focus on logic rather than binary details.
- **Operands:**
    - **Definition:** Entities that the instruction acts upon.
    - **Types:**
        - **Registers:** Numbers or names representing processor registers.
        - **Immediate Values:** Constants embedded within the instruction (e.g., =F'10').
        - **Memory Addresses:** Typically referenced via labels.
    - **Role:**
        - Define the source and destination for the data that the instruction will manipulate.
- **Labels:**
    - **Definition:** User-defined symbolic names representing memory addresses.
    - **Purpose:**
        - Facilitate branching, looping, and data referencing without needing to use numeric addresses.
    - **Example:**
        - In a program, a label such as LOOP might mark the beginning of a loop structure.
- **Directives (Pseudo-Ops):**
    - **Definition:** Special instructions that guide the assembler rather than execute on the CPU.

    - **Common Directives:**
        - **START/END:** Delimit the beginning and end of a program.
        - **DC (Define Constant):** Allocates storage with an initial value.
        - **DS (Define Storage):** Reserves memory without setting a value.
        - **EQU:** Equates a symbol to a constant or an expression.
        - **USING/DROP:** Manage the base register assignments for address calculations.
    - **Impact:**
        - Direct the assembly process by affecting how code is translated and how data is organized.
- **Comments:**
    - **Definition:** Annotations within the code that are ignored by the assembler.
    - **Purpose:**
        - Provide clarity, documentation, or notes to human readers.
    - **Notation:**
        - Typically preceded by a special character (such as a semicolon or asterisk), depending on the assembler's syntax.

6.2 The Assembly Process

- **Translation Role of the Assembler:**
    - Converts human-readable assembly code into machine code.
    - Generates supporting data structures such as Symbol Tables and Literal Tables to keep track of labels, literals, and addresses.
- **Pass Structure:**
    - **Two-Pass Assembly:**
        - **Pass 1:** Scans through the source code to assign addresses and build symbol and literal tables.
            - Handles directives that define data (e.g., DC, DS) and processes labels.
        - **Pass 2:** Uses the tables from Pass 1 to generate final machine code.
            - Resolves symbolic references and produces object code formatted for the loader.
- **Importance of Structured Format:**

- The consistency in format (fields for labels, mnemonics, operands, and comments) allows the assembler to accurately parse each instruction and process it accordingly.

- **Expanded Example:**
  - Consider the statement:
  - **LOOP L 1,FIVE** ; Load the value at label FIVE into register 1
    - **Label Field:** LOOP defines the entry point for a loop.
    - **Mnemonic Field:** L stands for the load instruction.
    - **Operand Field:** 1,FIVE specifies that register 1 is the destination and the address defined by FIVE is the source.
    - **Comment Field (after the ;):** Provides a human-readable explanation and is ignored by the assembler.

---

7. Instructions & Basic Elements

7.1 Instruction Structure and Addressing Modes

- **Opcode/Mnemonic Field:**
  - Represents the operation to be performed, typically one or two letters long.
  - **Example:** A for addition or L for load.
- **Operands:**
  - **Definition:** Specify the sources of data or the destinations for results.
  - **Examples of Operand Types:**
    - **Register Operands:** Represent processor registers (e.g., register 1, register 15).
    - **Immediate Operands:** Direct constant values, such as =F'10'.
    - **Memory Operands:** Use labels to represent addresses where data is stored.
- **Addressing Modes:**
  - **Immediate Addressing:** The operand is part of the instruction itself.
  - **Direct/Absolute Addressing:** The instruction specifies the exact memory address.
  - **Indexed/Relative Addressing:** Combines a base register value with an offset.
    - *Example:* In IBM 360 assembly, an effective address is computed as: **Effective Address = Base Register Content + Displacement.**

7.2 Instruction Formats and Fields

- **Fixed Length:**
    - Many systems use fixed-length instructions (commonly 2, 4, or 6 bytes), which simplifies decoding.
- **Common Formats:**
    - **RR Format (Register-to-Register):**
        - Both operands are registers.
        - Used for operations like AR 1,2 (add contents of register 2 to register 1).
    - **RX Format (Register-to-Memory):**
        - Involves one register operand and one memory operand, calculated using a base register and an offset.
        - *Example:* L 1,FIVE might be encoded with fields for the opcode, the destination register (1), an index field (if used), and a displacement value derived from FIVE.
    - **RS Format (Register-to-Storage):**
        - Used when an instruction involves storage operands, often with multiple fields for registers and displacements.
- **Fields in an Instruction:**
    - **Opcode Field:** Holds the binary code for the operation.
    - **Register Field(s):** Specify which registers are involved.
    - **Index/Base Fields:** Assist in calculating effective addresses for memory operands.
    - **Displacement Field:** Provides an offset that is added to a base register's value.
- **Detailed Example:**
    - For an RX instruction such as L 1,=F'10', the assembler must:
        - Look up the binary opcode for L.
        - Determine the register (1) and that no index is used (index set to zero).
        - Calculate the displacement for the literal =F'10' (which may involve placing the literal in a literal pool and assigning an address during the assembly process).

---

8. Types of Statements & Their Format

8.1 Classification of Assembly Statements (similar and repetitive to the previous part)

- **Executable Instructions:**
    - **Definition:** Instructions that are directly converted into machine code.

    - **Characteristics:**
        - They carry out operations such as data movement, arithmetic, and branching.
        - They are the “active” parts of the program.
    - **Example:**
        - A 1,FOUR adds the contents stored at label FOUR to register 1.
- **Assembler Directives (Pseudo-Ops):**
    - **Definition:** Instructions that guide the assembly process rather than generate machine code.
    - **Common Directives:**
        - **START:** Sets the starting address of the program.
        - **END:** Indicates the termination of the source program.
        - **DC (Define Constant):** Allocates storage and initializes it with a constant.
        - **DS (Define Storage):** Reserves memory space without initialization.
        - **EQU:** Associates a label with a constant or the result of an expression.
        - **USING/DROP:** Manage the base register assignments for addressing.
    - **Purpose:**
        - They help set up the environment and data structures required for the program and are processed by the assembler to produce tables for use in later passes.
- **Comments:**
    - **Definition:** Non-executable text intended for human readers.
    - **Notation:**
        - Often prefixed by a specific character (such as a semicolon ; or an asterisk *), comments are ignored during the assembly.
    - **Purpose:**
        - Aid in code readability and maintenance, offering explanations, revision notes, or reminders.

8.2 Format and Layout of a Statement

- **Field Layout:**
    - Assembly language statements are typically organized into columns or fixed-width fields, ensuring that each part of the statement is parsed correctly.
    - **Typical Fields Include:**

1. **Label Field:**
    - Usually occupies a fixed number of characters (e.g., columns 1–8).
    - Used for defining symbols for addresses.
2. **Mnemonic Field:**
    - Contains the opcode or directive (often in columns 10–15).
3. **Operand Field:**
    - May include one or more operands (starting at column 17).
    - Can list registers, constants, and symbols separated by commas.
4. **Comment Field:**
    - May begin at a specified column (e.g., column 40 or later) and continues to the end of the line.
    - These are ignored by the assembler.

- **Example Statement:**

```
LOOP    L   1,FIVE    ; Load register 1 with the value at label FIVE
        A   1,F'25'   ; Add constant 25 to register 1
        ST  1,FOUR    ; Store the result in memory at label FOUR
        BCT 3,LOOP    ; Branch to LOOP if register 3 is non-zero
```

    - Each field is carefully aligned so the assembler can determine which part of the line represents a label, an instruction, its operands, or a comment.

- **Spacing and Alignment:**
    - The strict format minimizes ambiguity and helps in error detection.
    - Assembly language editors or tools often enforce this format, and deviations may cause errors during assembly.
- **Handling Literals and Pseudo-Ops:**
    - Literals (e.g., =F'10') are recognized and stored in a literal pool.
    - Pseudo-ops like LTORG force the placement of the literal pool into memory, updating the location counter accordingly.
    - Directives (like EQU, DS, and DC) impact the location counter and the definition of symbols without generating machine code.
- **Error Detection:**

- The well-defined format allows the assembler to detect errors such as missing fields, improper spacing, or undefined symbols early in the process.

---

9. Overview: What Is an Assembler?

9.1 Definition and Purpose

- **Assembler Definition:**
  An assembler is a software tool that converts assembly language—a symbolic, mnemonic-based programming language—into machine language (object code) that the CPU can execute.
- **Primary Objectives:**
  - **Translation:** Transform human-readable assembly code into binary machine instructions.
  - **Auxiliary Data Generation:** Produce essential tables (Symbol Table, Literal Table, Base Table) that facilitate address resolution and later support the loader.
  - **Pseudo-Op Processing:** Interpret assembler directives (such as START, DC, DS, EQU, USING, LTORG) that manage program structure, memory allocation, and symbol definitions.
  - **Relocation Support:** Embed information that permits the program to be loaded at an arbitrary memory location, often via base-relative addressing.

9.2 Key Roles in System Software

- **Bridge to Hardware:**
  - Assemblers provide a human-friendly way to write low-level programs that directly control hardware.
- **Efficiency and Control:**
  - Assembly language allows highly optimized code, which is critical in system-level programming where performance is paramount.
- **Historical Context:**
  - Before high-level languages and sophisticated compilers existed, programmers wrote directly in machine language. The introduction of assembly language marked a crucial step toward abstraction while still maintaining hardware control.

---

10. Assembler Design: A General Approach

10.1 Design Methodology

- **Problem Specification:**

    - Clearly define the task: to convert assembly source into machine code and generate loader information.

- **Data Structures Determination:**
    - **Machine-Operation Table (MOT):** Contains each mnemonic's binary opcode, instruction length (commonly 2, 4, or 6 bytes), and format (RR, RX, RS, etc.).
    - **Pseudo-Operation Table (POT):** Lists directives (pseudo-ops) and specifies their processing routines. Often maintained separately for Pass 1 and Pass 2 since their roles differ.
    - **Symbol Table (ST):** Records every label encountered with its assigned address (the current location counter value), length, and a relocation attribute (relative or absolute).
        - *Error Handling:* Duplicate definitions are flagged as errors.
    - **Literal Table (LT):** Collects literal values (e.g., =F'10') as they occur and assigns them addresses—usually when a LTORG pseudo-op is encountered or at the END directive.
    - **Base Table (BT):** Maintains assignments for base registers (set via USING and managed with DROP) to support base-relative addressing.
- **Algorithm Specification:**
    - The design is modular—dividing the overall task into manageable functions (e.g., reading source cards, evaluating expressions, updating the location counter).
    - The overall algorithm is iterative and refined as you adjust data formats and error handling strategies.

10.2 Modularity in Assembler Design

- **Function Decomposition:**
    - The assembler is not a monolithic block but a collection of routines such as:
        - **READ1/READ2:** For sequentially reading each source statement.
        - **MOTGET1/MOTGET2:** For looking up instruction properties in the MOT.
        - **POTGET1/POTGET2:** For processing pseudo-ops.
        - **STSTO/STGET:** For inserting and retrieving symbols from the Symbol Table.
        - **LTSTO/LTGEN:** For handling literals.
        - **EVAL:** For evaluating arithmetic expressions and resolving addresses.
        - **BTSTO/BTGET:** For managing and utilizing base register information.
        - **PUNCH/PRINT:** For formatting output listings and generating object code.
- **Advantages:**

    - Each module can be individually designed, tested, and optimized.
    - Critical modules (like symbol table searching) are optimized for performance since they may be executed thousands of times in large programs.

10.3 Performance Considerations and Error Handling

- **Forward References:**
    - Symbols may be used before they are defined; the assembler must collect all definitions in Pass 1 so that Pass 2 can resolve them.
- **Error Detection:**
    - Errors like undefined symbols, misformatted statements, or redefinitions are detected by checking the symbol table during Pass 1.
- **Efficiency Gains:**
    - For large programs, searching the symbol table (using binary search when the table is sorted) can significantly reduce assembly time.
    - Optimizing table lookups and expression evaluations is crucial because they are executed repeatedly.
- **Relocation and Alignment:**
    - The assembler must adjust the location counter to ensure proper alignment (e.g., fullwords starting on addresses that are multiples of four).

---

11. Pass Structure of Assemblers

11.1 Rationale for a Two-Pass Approach

- **Handling Forward References:**
    - In many assembly programs, labels are referenced before they are defined. A single pass would struggle with this, but a two-pass system collects all definitions first.
- **Division of Responsibilities:**
    - **Pass 1:**
        - **Primary Tasks:**
            - Scan the entire source code to assign addresses using a **Location Counter (LC)**.
            - Build the Symbol Table (recording labels with their addresses, lengths, and relocation status).
            - Process pseudo-ops that affect program layout (e.g., DS, DC, EQU) and record literals in the Literal Table.

        - Save a copy of the processed source, annotated with LC values, for use in Pass 2.
    - **Error Checks:**
        - Identify duplicate labels and incorrect pseudo-op usage.
- **Pass 2:**
    - **Primary Tasks:**
        - Re-read the stored source from Pass 1.
        - Translate each instruction into its final machine code by looking up opcodes (via MOTGET2) and resolving operands using the Symbol Table and Literal Table.
        - Compute effective addresses for memory references using base registers (via the Base Table) and calculate displacements.
        - Process any pseudo-ops that generate object code (e.g., DC) and format the output into a final object deck suitable for the loader.
    - **Error Checks:**
        - Resolve any undefined symbols and check that calculated displacements fit within the allowed field widths.

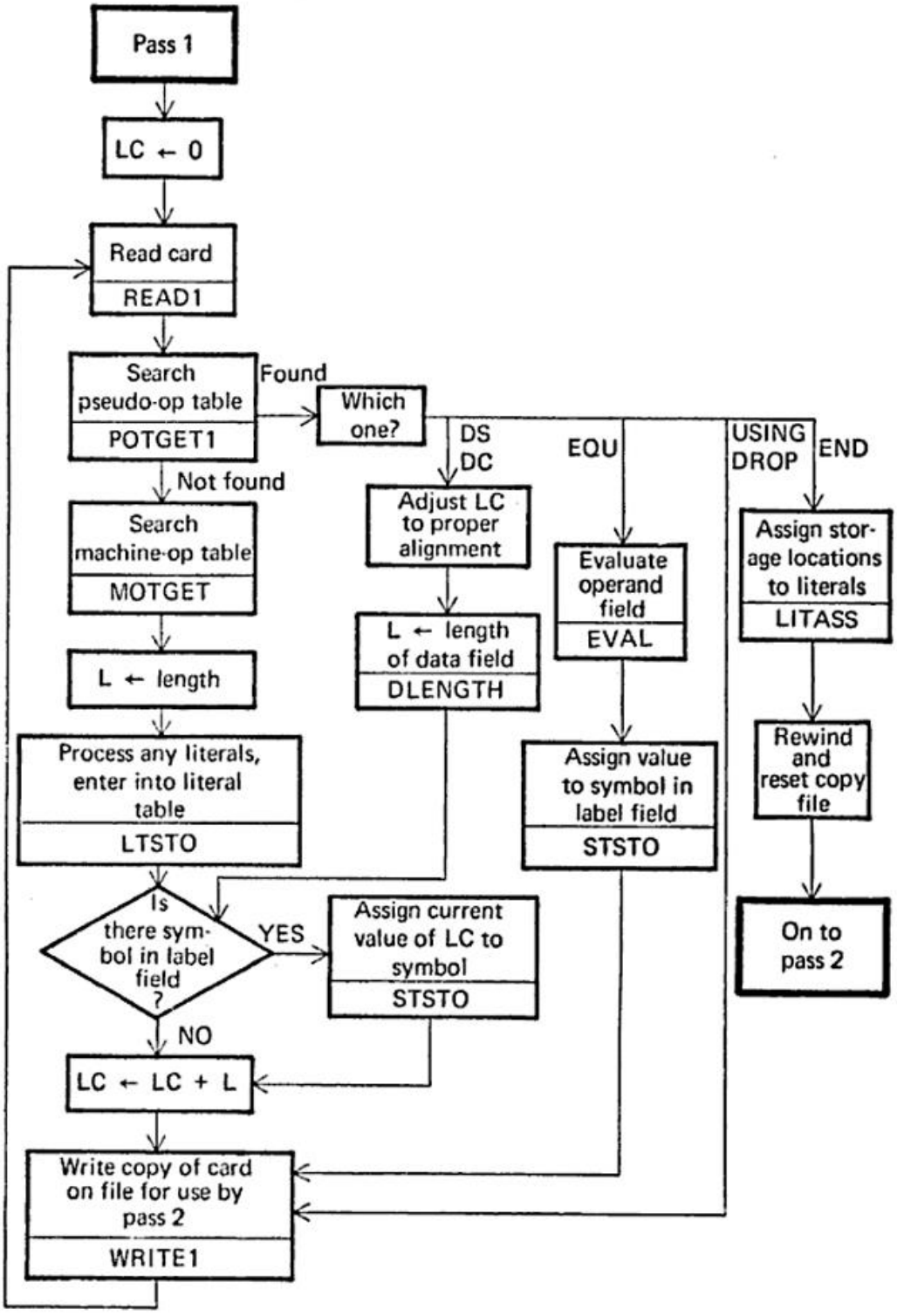
- **Advantages of Two-Pass Assembly:**
    - Resolves forward references without complex backpatching.
    - Segregates symbol definition from code generation, which simplifies the design and debugging.

11.2 Detailed Process Flow

Pass 1: Define Symbols and Collect Literals

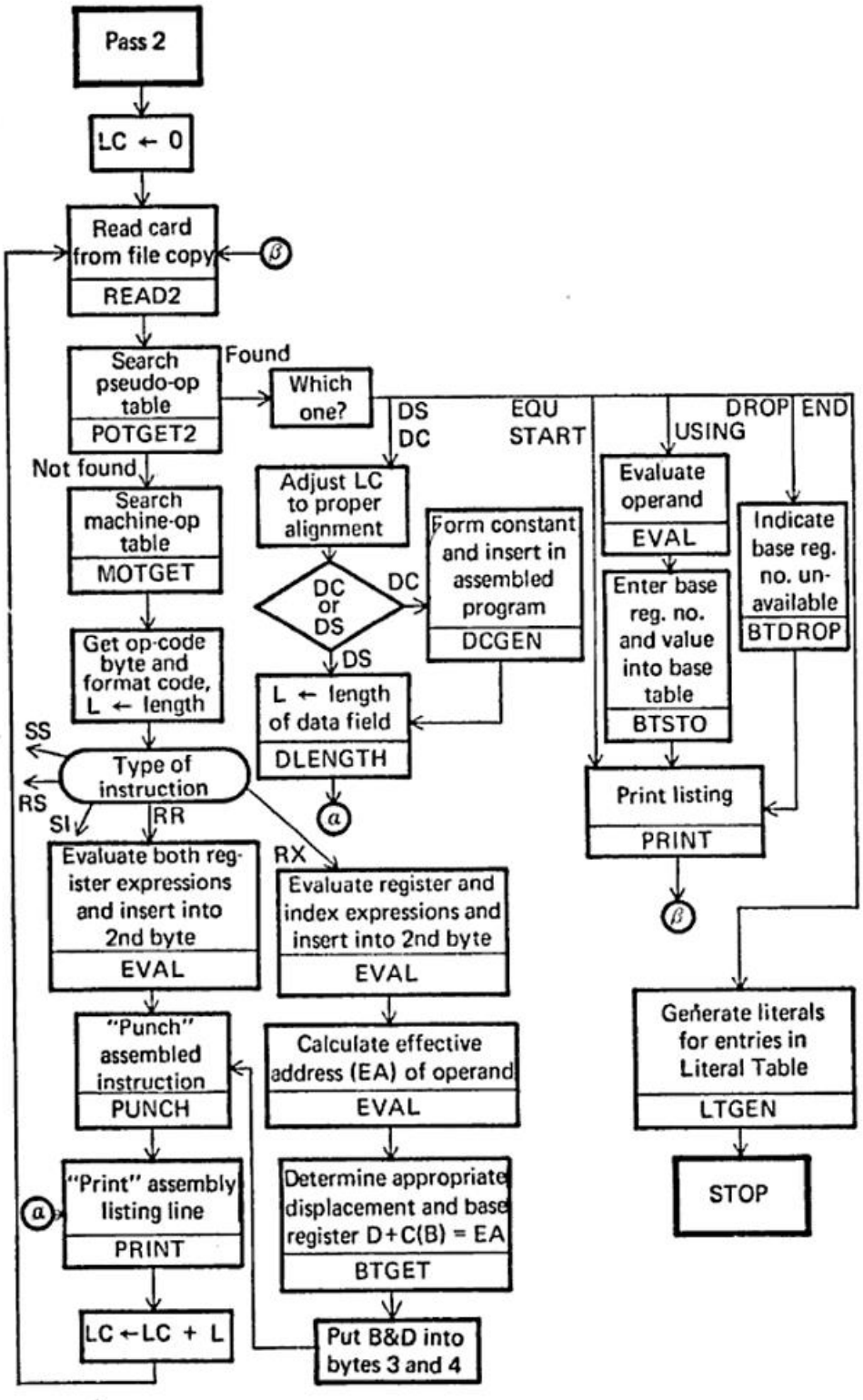
- **Initialization:**
    - Set LC to the starting address (provided by the START directive).
- **Line-by-Line Processing:**
    - **Determine Instruction Length:**
        - For each statement, use MOTGET1 to find the corresponding instruction length.
        - Update LC by the instruction's length; adjust for alignment if necessary.
    - **Symbol Handling:**
        - If a label is present, store it in the Symbol Table with its current LC value.

        - For pseudo-ops like EQU, evaluate the expression immediately and record the value.
    - **Literal Collection:**
        - Detect literals (e.g., =F'10') in operand fields and insert them into the Literal Table if they aren't already present.
        - When a literal pool is to be emitted (via LTORG or at END), assign addresses to all literals.
    - **Saving a Source Copy:**
        - Save the processed card with its LC value for use in Pass 2.
- **Flow Diagram Insight:**
    - This Flowchart illustrates subroutines such as POTGET1 (for pseudo-op detection), MOTGET1, STSTO, and LTSTO.


Pass 1
LC ← 0
Read card
READ1
Search pseudo-op table
POTGET1
Found
Which one?
DS
DC
EQU
USING
DROP
END
Not found
Search machine-op table
MOTGET
L ← length
Adjust LC to proper alignment
L ← length of data field
DLENGTH
Evaluate operand field
EVAL
Assign value to symbol in label field
STSTO
Assign storage locations to literals
LITASS
Rewind and reset copy file
On to pass 2
Process any literals, enter into literal table
LTSTO
Is there symbol in label field ?
YES
Assign current value of LC to symbol
STSTO
NO
LC ← LC + L
Write copy of card on file for use by pass 2
WRITE1

Pass 2: Generate Object Code and Resolve Addresses

- **Reinitialization:**
    - Reset LC to the starting address.
    - Retrieve the saved source copy from Pass 1.
- **Translation of Each Statement:**
    - **Opcode and Operand Resolution:**
        - For each statement, use MOTGET2 to fetch the opcode, instruction format, and expected length.
        - Evaluate operand fields using the EVAL routine, and look up symbol values in the Symbol Table (STGET).
    - **Effective Address Calculation:**
        - For memory operands (especially in RX formats), determine a suitable base register from the Base Table.
        - Calculate the displacement as:
          Displacement = (Symbol's LC value) – (Base Register Content)
        - Ensure that the displacement fits within the allowed number of bits (e.g., 12 bits on IBM 360).
    - **Object Code Generation:**
        - Assemble the binary fields (opcode, register numbers, base, displacement) into the final machine instruction.
        - Format the instruction as hexadecimal (or in the required format) for the object deck.
    - **Output:**
        - Write each assembled instruction to the output file (object deck) and produce a listing line that includes the original source, its LC value, and the generated code.
- **Flow Diagram Insight:**
    - This Flowchart illustrates subroutines such as READ2, POTGET2, MOTGET2, EVAL, BTGET, and PUNCH, showing how each field is processed.



11.3 Additional Considerations in Pass 2

- **Pseudo-Ops Revisited:**
    - Although most pseudo-ops affecting layout are handled in Pass 1, those that generate code (e.g., DC) require full processing in Pass 2.

- The USING and DROP pseudo-ops are processed to update the Base Table so that effective address calculations are correct.

- **Error Reporting:**
  - Pass 2 checks for unresolved symbols and improper operand formats.
  - If a symbol is not found in the Symbol Table or if a displacement exceeds field limits, errors are reported for correction.

---

12. Design of a Two-Pass Assembler

12.1 Detailed Architecture and Algorithm

- **Modular Routines:**
  - The assembler's design is broken into many smaller routines, each responsible for a specific function. For example:
    - **READ1/READ2:** Handle input reading.
    - **MOTGET1/MOTGET2:** Search the Machine-Operation Table for instruction details.
    - **STSTO/STGET:** Manage entries in the Symbol Table, ensuring that duplicate definitions are flagged.
    - **LTSTO/LTGEN:** Handle literal collection and generation.
    - **EVAL:** A crucial module that evaluates arithmetic expressions and resolves symbolic operands.
    - **BTSTO/BTGET:** Manage the Base Table for effective address computation.
    - **PUNCH/PRINT:** Convert binary instruction data to a human-readable listing and proper output format.

- **Algorithm Flow:**
  - The algorithm iterates over each source line, updating the Location Counter, processing symbols and literals, and finally generating the machine code.
  - In Pass 1, the assembler focuses solely on gathering information without producing final output, which simplifies the handling of forward references.
  - In Pass 2, the assembler combines the collected data with the source instructions to output fully resolved machine code.

12.2 Example Illustration and Table Formats

- **Sample Assembly Program:**
  - Consider a sample program with directives and instructions:

```
PRGAM2  START   0
        USING   *,15
SETUP   EQU     -
        LA      15,SETUP
        SR      TOTAL,TOTAL
        L       1,FIVE
        A       1,FOUR
        ST      1,TEMP
FIVE    DC      F'5'
FOUR    DC      F'4'
TEMP    DS      1F
        END
```

- **Pass 1 Processing:**
  - The assembler sets LC = 0, then reads the START directive.
  - The USING directive sets up the base register (15), and a temporary base table entry is made.
  - Each label (PRGAM2, SETUP, FIVE, FOUR, TEMP) is stored in the Symbol Table with its respective LC value.
  - Literal references (if any) are noted in the Literal Table.
- **Tables:**
  - **Symbol Table (ST):**

Symbol	Value (LC)	Length	Relocation
PRGAM2	0	1	R
SETUP	(evaluated)	1	R
FIVE	(e.g., 16)	4	R
FOUR	(e.g., 12)	4	R
TEMP	(e.g., 20)	4	R

  - **Literal Table (LT):**
    - Records each literal along with its assigned address when LTORG or END is encountered.
  - **Base Table (BT):**
    - Records the contents of base registers as specified by USING. For example, register 15 might be set to the starting address (or updated by subsequent USING directives).

- **Pass 2 Outcome:**
    - Using the symbol, literal, and base tables, each instruction is converted into its machine code equivalent (e.g., opcode, register fields, and computed displacements), then output in hexadecimal format for the loader.

12.3 Error Handling and Optimization

- **Error Detection:**
    - The assembler checks for duplicate symbol definitions in Pass 1.
    - In Pass 2, if a symbol is not found or if an operand cannot be resolved, an error is flagged.
- **Optimization in Table Searches:**
    - Sorting and using binary search for symbol table lookups can significantly reduce processing time, especially in large programs.
    - Efficient expression evaluation (EVAL) is critical since it may be invoked multiple times per statement.
- **Modularity Benefits:**
    - By decomposing the assembler into small, specialized routines, improvements in one module (such as faster search algorithms in STGET) benefit the entire assembly process without needing to redesign the whole system.

---

13. Macro Definition & Expansion

Macro processors are specialized tools used in assembly language programming to simplify repetitive coding tasks. They allow you to define a block of instructions once (using directives like MACRO and MEND) and then reuse that code by simply calling the macro by name. This not only reduces the amount of typing required but also minimizes errors, as the same block of code is maintained in a single definition rather than being duplicated throughout the program. Moreover, macro processors support parameters, which let you customize each macro call with different arguments, making your code more flexible and easier to manage.

In addition to code reuse, macro processors provide powerful features such as concatenation of macro parameters to dynamically create symbols, and the automatic generation of unique labels to prevent conflicts when macros are expanded multiple times. They can also support conditional expansion and even nested macro definitions, enabling sophisticated text substitution before the actual assembly process begins. This preprocessing step generates an expanded source code, free of macro definitions, which is then passed to the assembler for conversion into machine code.

Macro Definition

- **What It Is:**
  A macro is a named block of code defined between the directives **MACRO** and **MEND**. In

this block, you write a set of assembly instructions once and later reuse them via a macro call.

- **Purpose:**
    - **Code Reuse:** Instead of writing repetitive instructions, you define a macro and call it wherever needed.
    - **Error Reduction:** By writing the code once, you reduce the chance of mistakes from manual repetition.
    - **Abstraction:** It allows you to work at a higher level by hiding complex sequences behind a single mnemonic.
- **Structure & Example:**
  A typical macro definition includes a header (with the macro name and formal parameters), a body of instructions, and an ending directive.

```
SUM MACRO &A, &B, &RESULT
    MOVR AX, &A       ; Move the value from &A into AX
    ADD  AX, &B       ; Add the value from &B to AX
    MOVM AX, &RESULT  ; Move the result into &RESULT
MEND
```

In this example, **SUM** is the macro name and **&A**, **&B**, and **&RESULT** are the formal parameters. The body contains three instructions that perform an addition.

Macro Expansion

- **What It Is:**
  When the assembler encounters a macro call, it expands the macro—replacing the call with the body of the macro, where every formal parameter is substituted by the actual argument provided.
- **Process Details:**
    1. **Lookup:** The macro processor searches for the macro name in the **Macro Name Table (MNT)**.
    2. **Argument Mapping:** It sets up the **Argument List Array (ALA)**, which maps each formal parameter to the actual argument from the macro call.
    3. **Substitution:** The processor reads the macro body from the **Macro Definition Table (MDT)** and replaces occurrences of formal parameters with the corresponding actual values.
    4. **Insertion:** The resulting expanded instructions are then inserted into the source code in place of the macro call.
- **Example:**
  With the above definition, if you call:

```
SUM X1, X2, X3
```

the ALA is prepared as:

- **&A** → X1
- **&B** → X2
- **&RESULT** → X3
  The macro is expanded to:

```
MOVR AX, X1
ADD  AX, X2
MOVM AX, X3
```

This expanded code then goes through the normal assembly process.

---

14. Arguments in Macros

Positional Parameters

- **Definition:**
  Formal parameters in a macro definition are matched with the actual arguments by their position.
- **How It Works:**
  The first actual argument replaces the first formal parameter, the second replaces the second, and so on.
- **Example:**
  In the macro **SUM** defined above, a call:

```
SUM X1, X2, X3
```

maps as follows:

- &A becomes X1
- &B becomes X2
- &RESULT becomes X3

- **Key Point:**
  Positional parameters are simple and intuitive when the number of parameters is small. The order in the call must exactly match the order in the macro header.

Keyword Parameters (Optional)

- **Definition:**
  Some macro processors allow you to specify parameters by name instead of position. This is especially useful for macros with many parameters or when you want to override default values.
- **Example:**
  A macro might be defined with defaults:

```
ADDVAL MACRO &SRC=DEFAULTSRC, &DST=DEFAULTDST, &CONST=5
    L  2, &SRC
    A  2, &CONST
    ST 2, &DST
MEND
```

A call could be:

```
ADDVAL &DST=DATAOUT, &SRC=DATAIN
```

Here, **&CONST** uses its default value because it isn't explicitly provided.

- **Advantage:**
  Keyword parameters enhance clarity and flexibility, especially in complex macros.

---

15. Concatenation of Macro Parameters

What It Is:

- **Definition:**
  Concatenation in macro processing means combining a macro parameter with other literal text to form a new symbol or string during expansion.
- **Purpose:**
  - **Dynamic Symbol Creation:** Enables the creation of new identifiers based on macro parameters without manually specifying them.
  - **Compact Code:** Reduces repetition by programmatically building symbol names.

How It Works:

- When a macro parameter appears adjacent to other characters (without any space), the macro processor concatenates the parameter's value with the literal text.

- Some assemblers require a special concatenation operator (e.g., an arrow -> or another symbol) to indicate that concatenation should occur.

Example:

```
MACRO
MAKE_LABEL &ID
    L R1, DATA&ID
MEND
```

- **Call:**

```
MAKE_LABEL 1
```

- **Expansion:**
  The processor replaces **&ID** with 1 and concatenates it with DATA, producing:

```
L R1, DATA1
```

- **Note:**
  This allows you to create symbols dynamically, especially useful when dealing with arrays or repetitive structures.

---

16. Generation of Unique Labels

Why Unique Labels Are Needed:

- **Problem:**
  When a macro includes internal labels (used for jumps or branches), multiple expansions of the macro could result in duplicate labels, causing conflicts in the final source code.
- **Solution:**
  The macro processor automatically generates unique labels for each macro expansion. It does this by appending a unique identifier—such as a counter or a timestamp—to the label.

How It Works:

- **Automatic Label Generation:**
  For each macro expansion, any internal label (e.g., .LBL) is replaced with a unique label such as .LBL1, .LBL2, etc.
- **Example Macro with Internal Label:**

```
MACRO
MAC_GREATER &X, &Y, &RESULT
    CMP AX, &Y
    JG  .LBL      ; Jump if X is greater than Y
    MOVR AX, &Y
.LBL MOVM AX, &RESULT
MEND
```

Without unique label generation, if **MAC_GREATER** is expanded twice, both copies would have the same label .LBL.

- **After Unique Label Generation:**
    - First call might expand to:

```
CMP AX, X
JG  .LBL1
MOVR AX, Y
.LBL1 MOVM AX, RESULT
```

    - Second call might expand to:

```
CMP AX, X
JG  .LBL2
MOVR AX, Y
.LBL2 MOVM AX, RESULT
```

- **Result:**
  Each macro call now has its own unique internal label, avoiding any potential conflicts.

---

17. Conditional Macro Expansion

Overview:

- **Definition:**
  Conditional macro expansion lets the macro processor decide whether or not to include

specific parts of the macro body based on conditions. This is similar to “if” statements in high-level languages, but it happens at assembly time (before the actual assembly).

- **Purpose:**
    - To allow flexible code generation based on the values of macro parameters.
    - To include or exclude blocks of code within a macro depending on conditions.

Key Directives:

- **AIF (Assembly IF):**
    - Evaluates a logical expression.
    - If the expression is true, it causes the macro processor to jump to a specific internal label (a marker in the macro body) to skip or include code.
    - **Format:**

```
AIF (expression) .LABEL
```

Here, if the expression is true, control is transferred to the line labeled .LABEL within the macro body.

- **AGO (Assembly GO):**
    - Provides an unconditional branch within a macro.
    - **Format:**

```
AGO .LABEL
```

This causes an unconditional jump to the macro label .LABEL.

Example:

Consider a macro that conditionally performs an addition only if two parameters are not equal:

```
COND_ADD MACRO &X, &Y, &RESULT
    CMP  AX, &X         ; Compare AX with &X
    AIF  (&X EQ &Y) .SKIP  ; If &X equals &Y, jump to .SKIP
    ADD  AX, &Y         ; Otherwise, add &Y to AX
.LABEL .SKIP
    MOVM AX, &RESULT    ; Store the result
MEND
```

- **Explanation:**
    - The AIF instruction checks if **&X equals &Y**.
    - If true, it jumps to the label .SKIP, thus **skipping the addition**.
    - If false, it executes the ADD instruction before continuing.

Usage Consideration:

Conditional expansion is very powerful in tailoring macro-generated code to specific circumstances, reducing redundant instructions and enhancing flexibility.

---

18. Nested Macros

Overview:

- **Definition:**
  A nested macro is one that contains a call to another macro (or even itself, which leads to recursion). This means that within the body of one macro, you can invoke another macro.
- **Purpose:**
    - **Modularity:** Breaks down complex tasks into smaller, reusable macro pieces.
    - **Reusability:** You can combine simple macros to build more complex behavior.

How It Works:

- When a macro call is encountered within a macro's body, the macro processor temporarily suspends the expansion of the outer macro and processes the inner macro call.
- After the inner macro is expanded, control returns to the outer macro's expansion process.

Example:

Inner Macro Definition:

```yaml
PRINT_MSG MACRO &MSG
    WTO &MSG
MEND
```

Outer Macro Definition:

```yaml
PROCESS_DATA MACRO &SRC, &DST
    L  2, &SRC
    A  2, =F'10'
    ST 2, &DST
    PRINT_MSG 'DATA PROCESSED'
MEND
```

Macro Call:

```nginx
PROCESS_DATA DATAIN, DATAOUT
```

- **Expansion Flow:**
    1. The outer macro PROCESS_DATA is expanded.
    2. When the line PRINT_MSG 'DATA PROCESSED' is reached, the processor recognizes it as a macro call.
    3. It then expands PRINT_MSG separately, replacing it with WTO 'DATA PROCESSED'.
    4. After finishing the inner macro expansion, the outer macro expansion resumes.

- **Final Expanded Code:**

```
L  2, DATAIN
A  2, =F'10'
ST 2, DATAOUT
WTO 'DATA PROCESSED'
```

Usage Consideration:

Nested macros allow you to build hierarchical, modular code. However, they add complexity, so careful design and debugging are necessary.

---

19. Macros Defining Macros

Overview:

- **Definition:**
  In some advanced macro processors, you can have a macro that, when expanded, generates another macro definition. Essentially, one macro can output macro definitions as part of its expansion.
- **Purpose:**
  - To create highly flexible and reusable code.
  - Useful when you want to generate boilerplate macro code dynamically based on certain parameters.

How It Works:

- When a macro that defines another macro is expanded, its output includes a complete macro definition (i.e., it includes a new macro header, body, and MEND).
- The generated macro definition is then processed by the assembler (or macro processor) as if it were written in the source file.

Example:

Imagine you have a macro that generates a simple macro to load a constant into a register.

Defining Macro:

```pgsql
DEFINE_LOAD_MACRO MACRO &NAME, &CONST
    &NAME MACRO &R
          L  &R, =F'&CONST'
    MEND
MEND
```

Macro Call:

```nginx
DEFINE_LOAD_MACRO LOAD_CONST, 100
```

- **Expansion:**
    - The macro DEFINE_LOAD_MACRO is expanded, and it generates a new macro called LOAD_CONST.
- The generated macro might look like:

```
LOAD_CONST MACRO &R
      L  &R, =F'100'
MEND
```

- **Usage:**
  Later in your program, you can call:

```
LOAD_CONST R1
```

which expands to:

```
L R1, =F'100'
```

Usage Consideration:

Macros that define macros are less common but are powerful in scenarios where code generation must be dynamic and adaptable. They enable a form of meta-programming, where you write code that writes code.

---

20. Macro Processor Design

20.1 Overview and Objectives

- **Purpose of a Macro Processor:**
  A macro processor is a preprocessor that scans the source code for macro definitions and calls. It "expands" these macros—replacing each macro call with the corresponding block of code (with parameter substitution) before the assembler translates the expanded code into machine language.
- **Key Goals:**
  - **Reduce Repetitive Coding:** Instead of writing the same sequence of instructions multiple times, you define them once.
  - **Enhance Code Readability & Maintainability:** Changes to repetitive code are made in one place.
  - **Enable Parameterization:** Macros can accept parameters (either positional or keyword) to make them flexible.
  - **Support Advanced Features:** Such as conditional expansion, nested macros, and even macros that define other macros (meta-macros).
- **Implementation Context:**
  Macro processing can be integrated as a separate phase (a standalone macro processor) or built into the assembler itself. In many systems, macro processing is a separate module that produces an expanded source code, which is then fed into the assembler.

---

21. Macro Processor Design: Architectural Components & Data Structures

21.1 Key Data Structures

1. **Macro Name Table (MNT):**
   - **What It Holds:**
     - Each macro's name.
     - A pointer (or index) to the start of the macro's definition in the Macro Definition Table (MDT).

- **Example:**
  If you define a macro SUM at MDT index 5, the MNT contains an entry like:
  - **Name:** SUM
  - **MDT Index:** 5

21.2. Macro Definition Table (MDT):

- **What It Holds:**
  - The complete set of macro definition lines (the macro's body).
  - Each line may include formal parameters that appear as placeholders (e.g., &A, &B).
- **Example:**
  For the macro:

```
SUM MACRO &A, &B, &RESULT
    MOVR AX, &A
    ADD  AX, &B
    MOVM AX, &RESULT
MEND
```

The MDT will store:

- Line 1: MOVR AX, &A
- Line 2: ADD AX, &B
- Line 3: MOVM AX, &RESULT
  (The MEND line simply marks the end.)

21.2.1 Argument List Array (ALA):

- **What It Holds:**
  - A temporary mapping between the macro's formal parameters and the actual arguments passed during a macro call.
- **Example:**
  For the call SUM X1, X2, X3, the ALA maps:
  - **&A** → X1
  - **&B** → X2
  - **&RESULT** → X3

21.2.1 Other Supporting Structures:

- **Counters:** For MDT and MNT indices (e.g., MDTC, MNTC) that keep track of the next available entry.
- **Temporary Workspaces:** To hold intermediate expanded code lines before they are written to the new source file.

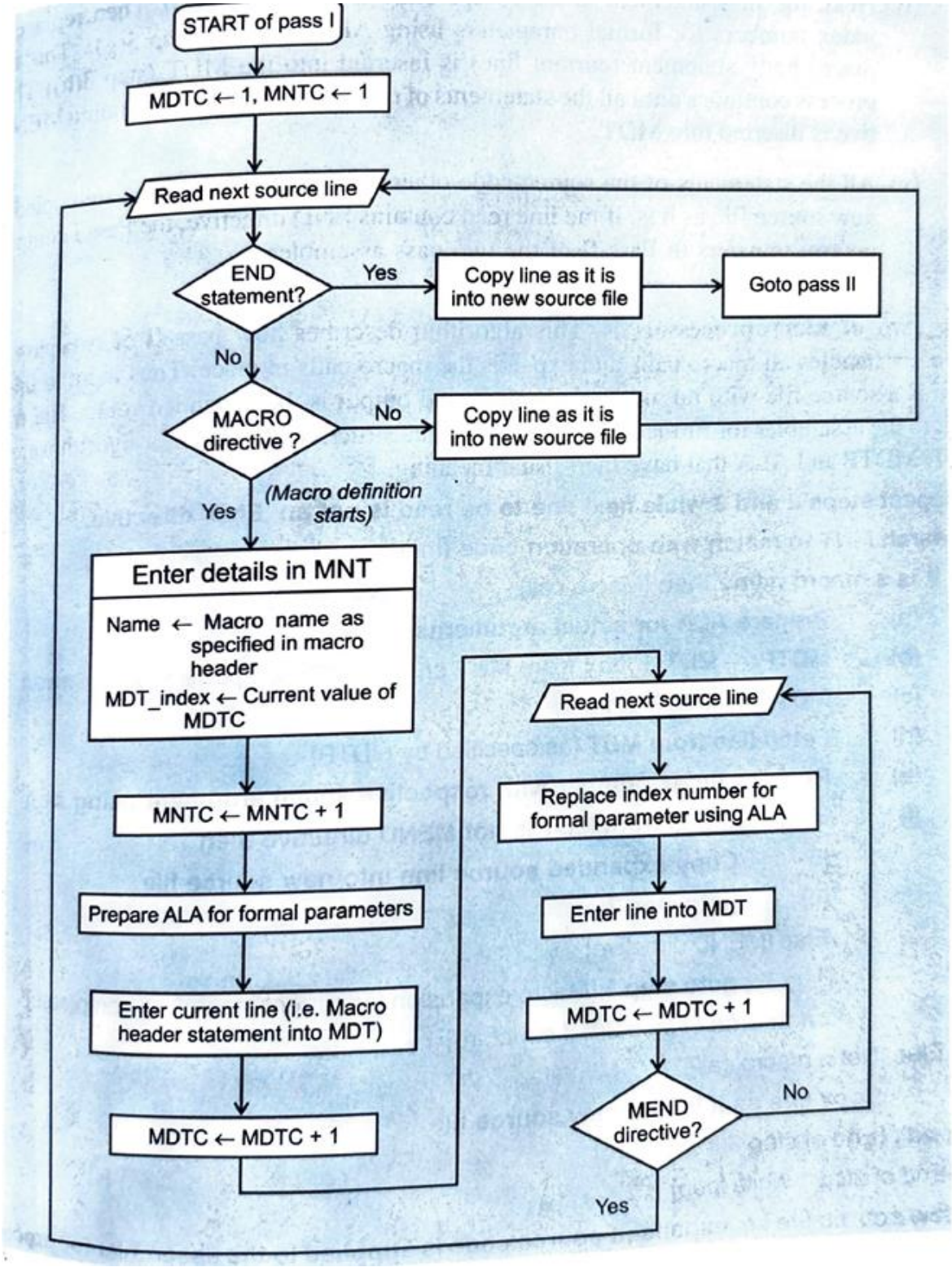
21.3 Flow of Macro Processing

Macro processing is usually implemented as a two-pass process (though a single-pass approach is also possible):

1. **Pass I – Definition Phase:**
    - **Objective:**
        - Scan the source code to **identify and store all macro definitions**.
        - Remove the macro definitions from the source, leaving behind the macro calls and regular code.
    - **Steps:**

1. **Initialize Counters:** Set MDTC and MNTC to 1.
2. **Read Source Line (READ1):**
    - If a line begins with the MACRO directive, start macro definition.
3. **Store Macro Header:**
    - Extract the macro name and formal parameters.
    - Insert an entry in the MNT with the macro name and the current MDTC (pointer to MDT).
4. **Store Macro Body:**
    - Read each line until the MEND directive.
    - Process each line, possibly replacing formal parameters with index numbers (for easier substitution later).
    - Update the MDT counter (MDTC) after each line.
5. **Copy Non-Macro Lines:**
    - Lines that are not macro definitions are copied verbatim to a new source file.

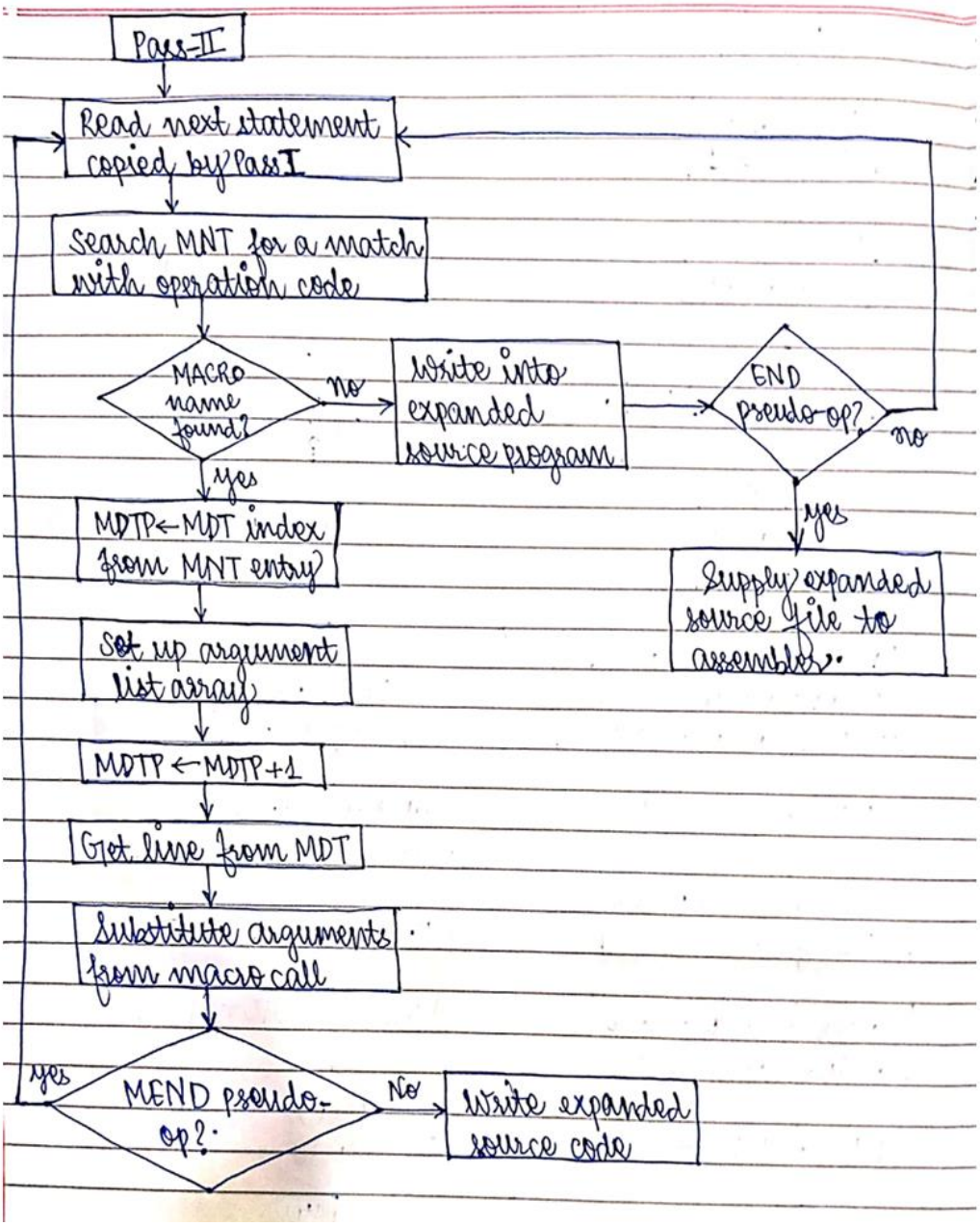
- **Output of Pass I:**
    - A new source file without any macro definitions but with macro calls intact.
    - Fully built MNT and MDT (and an ALA prototype for each macro's formal parameters).



2. **Pass II – Expansion Phase:**
    - **Objective:**
        - Process the new source file, identify macro calls, and expand them by substituting actual parameters.
    - **Steps:**

1. **Read Expanded Source (READ2):**
    - For each line, check if the operation mnemonic matches a macro name from the MNT.
2. **If Macro Call:**
    - Set up the ALA by mapping formal parameters to the actual arguments provided.
    - Using the MNT entry, retrieve the starting point in the MDT.
    - Read the macro's body from the MDT line by line.
    - For each line, substitute the formal parameter placeholders (or index numbers) with the actual arguments from the ALA.
    - Insert the expanded lines into the new output source file.
3. **If Not a Macro Call:**
    - Simply copy the line to the output.

- **Final Output:**
    - A fully expanded source file, free of macro definitions, ready for assembly.



---

22. Two-Pass vs. Single-Pass Macro Processors

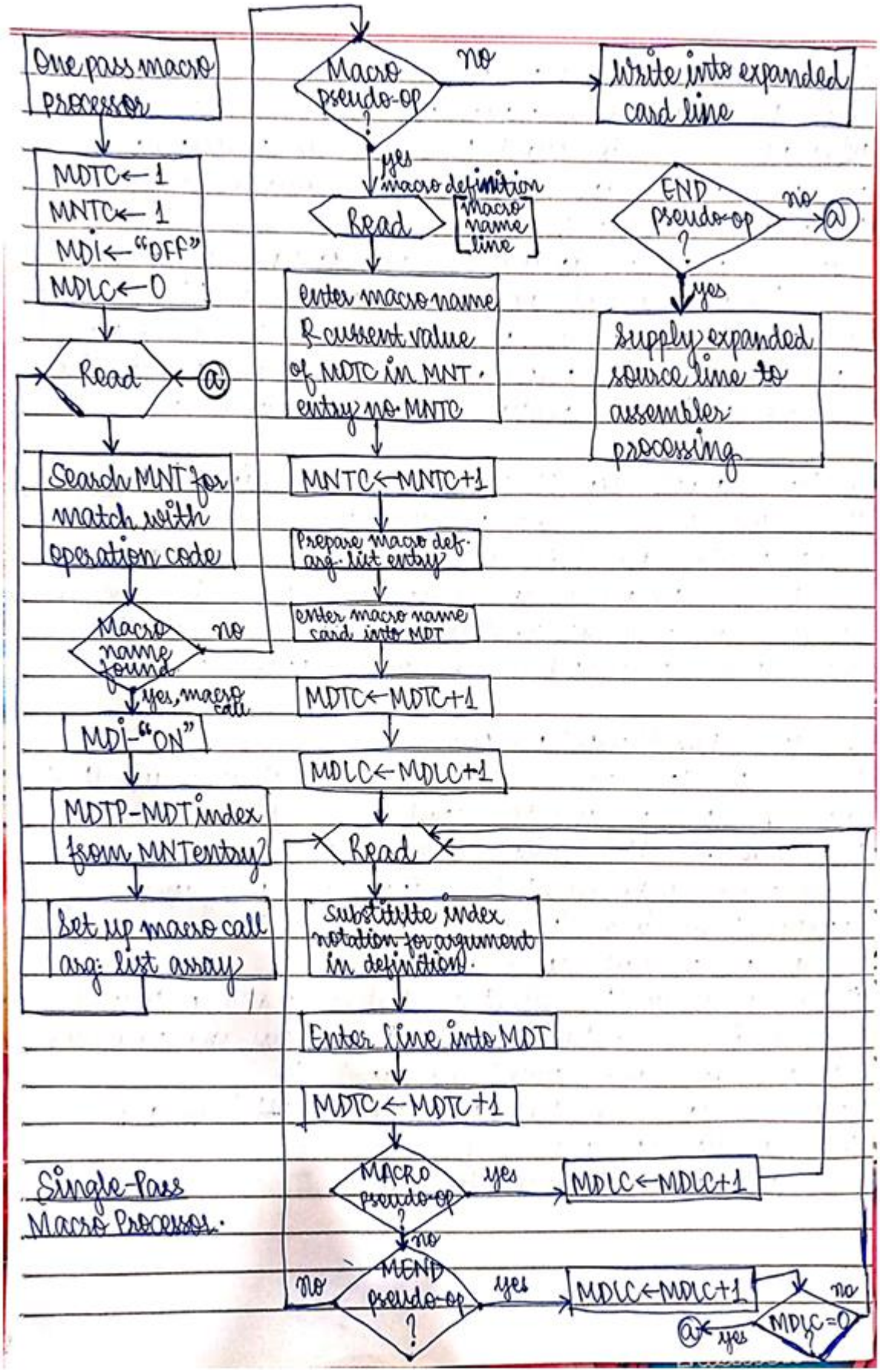
22.1 Two-Pass Macro Processor

- **Advantages:**
    - **Handles Forward References:**
        - Since Pass I collects all macro definitions before any expansion, you can call macros even if their definitions appear later in the source file.
    - **Structured Approach:**
        - Separates the concerns of definition and expansion, making the design modular and easier to debug.

- **Disadvantages:**
    - **Two Scans Required:**
        - The source file is read twice, which might be slower on very large programs (though in practice, the overhead is minimal).
- **When to Use:**
    - When macro definitions can appear anywhere in the source, and you want flexibility in the order of definitions and calls.

22.2. Single-Pass Macro Processor

- **Advantages:**
    - **Faster Processing:**
        - The source code is processed only once.
    - **Memory Efficiency:**
        - Avoids the need to store a separate expanded source file.
- **Disadvantages:**
    - **Limited Handling of Forward References:**
        - Typically, you must define a macro before it is called, which can limit flexibility.
    - **Increased Complexity:**
        - It may require on-the-fly expansion and backpatching, which makes the design more complex.
- **When to Use:**
    - In systems where speed is critical and source code is well-ordered (macro definitions appear before their calls).


One pass macro processor
MDTC ← 1
MNTC ← 1
MDI ← "OFF"
MDLC ← 0
Read
Search MNT for match with operation code
Macro name found?
no
yes, macro call
MDI ← "ON"
MDTP ← MDT index from MNT entry
Set up macro call arg: list array
Macro pseudo-op ?
no
Write into expanded card line
yes
Read
macro definition
macro name line
Enter macro name & current value of MDTC in MNT entry no. MNTC
MNTC ← MNTC+1
Prepare macro def. arg. list entry
Enter macro name card into MDT
MDTC ← MDTC+1
MDLC ← MDLC+1
Read
Substitute index notation for argument in definition
Enter line into MDT
MDTC ← MDTC+1
MACRO pseudo-op ?
yes
MDLC ← MDLC+1
no
MEND pseudo-op ?
no
yes
MDLC ← MDLC+1
MDLC = 0 ?
no
yes
END pseudo-op ?
no
yes
Supply expanded source line to assembler processing
Single-Pass Macro Processor.

22.3. Implementation Within an Assembler

- **Integration Choices:**
    - **Standalone Macro Processor:**
        - Processes the source code, expands all macros, and then feeds the resulting expanded source code to the assembler.
    - **Built-In Macro Processor:**
        - Integrated into the assembler, where macro expansion is one of the early steps in the assembly process.
- **Trade-Offs:**
    - A **built-in macro processor** can optimize integration, sharing data structures like the symbol table between macro processing and assembly.
    - A **separate macro processor** may be easier to develop and maintain as a distinct module.

---

24. Detailed Example Walkthrough

Example Program: Two-Pass Macro Processing

Consider a simple program that defines a macro to add two numbers:

Macro Definition (in the source):

Macro Definition (in the source):

```css
SUM MACRO &A, &B, &RESULT
    MOVR AX, &A
    ADD  AX, &B
    MOVM AX, &RESULT
MEND
```

Rest of the Program:

```pgsql
START 0
USING *,15
SUM X1, X2, X3
WRITE X3
END
```

Pass I: Definition Phase

- **Step 1:**
  - **Read "SUM MACRO &A, &B, &RESULT"**
  - **Store in MNT:**
    - Entry: Name = SUM, MDT Index = (say) 1
  - **Record formal parameters in the ALA prototype:** &A, &B, &RESULT.
- **Step 2:**
  - **Store Macro Body in MDT:**
    - MDT[1]: MOVR AX, &A
    - MDT[2]: ADD AX, &B
    - MDT[3]: MOVM AX, &RESULT
    - MDT[4]: MEND (marks the end)

    - **Increment MDT Counter (MDTC) appropriately.**
- **Step 3:**
    - **Copy other source lines unchanged** to the new source file.
    - Macro definitions are removed from the output.

Pass II: Expansion Phase

- **Step 1:**
    - **Read through the new source file line by line.**
    - When encountering SUM X1, X2, X3, the processor:
        - Looks up SUM in the MNT to get the MDT index.
        - Sets up the ALA:
            - &A = X1, &B = X2, &RESULT = X3.
- **Step 2:**
    - **Expand the macro:**
        - Read MDT[1]: MOVR AX, &A → Replace &A with X1 → MOVR AX, X1
        - Read MDT[2]: ADD AX, &B → Replace &B with X2 → ADD AX, X2
        - Read MDT[3]: MOVM AX, &RESULT → Replace &RESULT with X3 → MOVM AX, X3
        - MDT[4] is MEND, marking the end.
- **Step 3:**
    - **Insert expanded lines** in place of the macro call in the output source file.
- **Final Expanded Source:**

```
START 0
USING *,15
MOVR AX, X1
ADD  AX, X2
MOVM AX, X3
WRITE X3
END
```

- This expanded code is then passed to the assembler proper

[Thank You – Ash & ChatGPT]