A Comprehensive Guide to Designing and Implementing an Assembler for a Custom Toy Instruction Set

I. Understanding the Assembler's Role

An assembler is a pivotal piece of system software that forms the critical link between human-understandable assembly language and the raw binary machine code that a computer's central processing unit (CPU) executes. This section elucidates the fundamental nature of assemblers, the pedagogical value of constructing one for a custom "toy" Instruction Set Architecture (ISA), and the intricate relationships between an ISA, assembly language, and machine code.

A. What is an Assembler? Bridging Assembly Language and Machine Code.

At its core, an assembler is a translator program. It takes source code written in assembly language—a low-level programming language that uses symbolic names called mnemonics to represent processor operations—and converts it into an equivalent sequence of machine code instructions.[1] Machine code is the native language of the processor, consisting of binary digits (bits) that directly control the CPU's operations. Assembly language provides a layer of abstraction above machine code, making it more feasible for humans to write programs that interact closely with the hardware, compared to the extraordinarily error-prone task of writing binary code directly.[2] For instance, instead of writing a binary sequence like 1011001000010000 (which might represent an operation to load a value into a register), a programmer can write a mnemonic like MOV R2, #16.

The assembler is specific to a particular Instruction Set Architecture (ISA). An ISA defines the set of instructions, registers, memory addressing capabilities, and other architectural features of a processor.[1] Consequently, an assembler designed for an Intel x86 ISA cannot be used to assemble code for an ARM ISA, and vice versa. This specificity arises because the assembler must know the exact binary representation for each mnemonic and how to encode operands according to the target ISA's rules. It effectively bridges the gap between the symbolic representation preferred by programmers and the binary format required by the hardware.[1]

The translation process performed by an assembler, while simpler than that of a compiler for a high-level language, shares some fundamental characteristics. Compilers translate languages like C++ or Python into lower-level code, often assembly language. The assembler then performs the subsequent step of converting

this assembly code into executable machine code. This involves phases analogous to those in compilation, such as lexical analysis (breaking down the source code into tokens), parsing (checking the syntax of instructions), and code generation (producing the binary output).[3] Thus, an assembler can be viewed as a specialized type of compiler, one that typically has a more direct, often one-to-one, mapping from source statements (assembly instructions) to target statements (machine instructions).[2]

B. Why Build an Assembler for a Toy ISA? Learning and Control.

Constructing an assembler, particularly for a custom-designed "toy" ISA, is a profoundly educational endeavor. A toy ISA is a simplified instruction set, designed not for commercial production but for learning purposes. It typically features a small number of instructions, limited addressing modes, and a straightforward memory model.[5] Building an assembler for such an ISA offers several benefits:

1. **Deep Understanding of Computer Architecture:** The process necessitates a thorough understanding of how instructions are structured, how operands are encoded, how memory is addressed, and the overall functioning of a CPU.
2. **Insight into Software-Hardware Interface:** It demystifies the critical interface between software programs and the underlying hardware, showing concretely how symbolic code is transformed into electrical signals that control the processor.
3. **Practical Experience with Language Translation:** It provides hands-on experience with fundamental language processing techniques, including lexical analysis, symbol table management, and code generation logic.
4. **Control and Customization:** Designing both the ISA and the assembler allows for complete control over the architecture and its symbolic representation, facilitating experimentation and a tailored learning experience.

The "toy" aspect ensures the project remains manageable while still covering the core principles of assembler design and operation. Projects like "ToyASM" for a simplified Toy ISA illustrate this approach, aiming to make programming the ISA easier by providing a more human-readable alternative to writing raw binary.[5] This act of building even a simple assembler for a toy ISA powerfully demonstrates the concept of abstraction in computing. By creating mnemonics and symbolic addresses, the assembler hides the complex and error-prone details of binary machine code, presenting the programmer with a more abstract and manageable interface to the hardware.[5] This layering of abstractions is a cornerstone of modern computer science,

enabling the development of highly complex systems.

C. The Relationship: ISA, Assembly Language, and Machine Code.

Understanding the distinct roles and interdependencies of the ISA, assembly language, and machine code is crucial for comprehending the assembler's function:

- **Instruction Set Architecture (ISA):** The ISA is the abstract model of a computer that defines how the CPU is controlled by software; it acts as the definitive contract between the hardware and the software.[1] It specifies the set of operations the processor can perform, the available registers, supported data types, how hardware manages main memory, key features like virtual memory (if any), and the input/output model.[8] Essentially, the ISA is the "language the processor speaks".[2] Different ISAs (e.g., x86, ARM, MIPS, RISC-V) have different instruction sets and architectural features, meaning programs written for one ISA must be recompiled or translated to run on another.[1]
- **Assembly Language:** Assembly language is a low-level programming language that provides a symbolic, human-readable representation of the machine code instructions defined by a specific ISA.[2] Each assembly instruction mnemonic typically corresponds directly to a single machine instruction, although assemblers can also support pseudo-instructions or macros that expand into multiple machine instructions for programmer convenience.[2] Assembly language is ISA-specific; for example, x86 assembly language is distinct from ARM assembly language.
- **Machine Code:** Machine code is the sequence of binary digits (0s and 1s) that a processor can directly understand and execute.[1] It is the ultimate output of an assembler (or a compiler that targets machine code directly). Each instruction in machine code contains an opcode (operation code) that specifies the operation to be performed, and operands that specify the data or locations to be used by that operation. The format of these machine code instructions is strictly defined by the ISA.

In summary, the ISA dictates the capabilities of the processor and the structure of its machine code. Assembly language offers a more convenient, symbolic way for programmers to write programs that utilize these ISA-defined instructions. The assembler then translates this assembly language into the corresponding machine code that the processor can execute.[2] This hierarchical relationship forms the foundation of how software commands are ultimately realized by computer hardware.

II. Designing Your Custom Toy Instruction Set Architecture (ISA)

Before embarking on the construction of an assembler, the target Instruction Set Architecture (ISA) must be meticulously defined. For a "toy" ISA, the primary goal is simplicity to facilitate learning and keep the assembler project manageable. Every design choice for the ISA will directly influence the complexity of the assembler. A well-considered, minimalistic ISA will lead to a more straightforward assembler implementation.

A. Core Components of an ISA (Registers, Memory Model, Data Types).

The foundational elements of any ISA include its registers, memory model, and supported data types.

- **Registers:** These are small, high-speed storage locations within the CPU used to hold data and intermediate results during program execution.[9] For a toy ISA, key decisions include:
  - **Number and Type:** How many general-purpose registers (GPRs) will there be? A small number, such as 4, 8, or 16, is typical for a toy ISA.[10] Will there be any special-purpose registers, like a Program Counter (PC) to hold the address of the next instruction, or a Stack Pointer (SP)? In many simple ISAs, the PC is an implicit register managed by the control unit.
  - **Size:** What is the bit-width of each register (e.g., 8-bit, 16-bit)? This often aligns with the ISA's primary data processing width.[10]
- **Memory Model (Organization):** This defines how the CPU accesses and interacts with the main memory.[9]
  - **Address Space Size:** How many unique memory locations can be addressed? This is determined by the width of memory addresses. For example, an 8-bit address can access $2^8=256$ locations, while a 16-bit address can access $2^{16}=65536$ (64K) locations.[10] The address space size chosen for a toy ISA should be modest.
  - **Character Size (Memory Unit Size):** What is the size of the smallest addressable unit of memory? This is almost universally 8 bits (one byte).[10]
  - **Architecture:** Most general-purpose computers use a Von Neumann architecture, where instructions and data share the same memory space and are accessed via the same bus.[9] This is generally simpler to implement for a toy ISA than a Harvard architecture (separate memory for instructions and data).
  - **Alignment:** Will data larger than a byte (e.g., 16-bit words) need to be aligned

on specific memory boundaries (e.g., even addresses)? Enforcing alignment can simplify hardware but adds constraints for the programmer and assembler. For a toy ISA, disallowing unaligned access or simply not supporting multi-byte accesses that span unaligned boundaries might be easier.

- **Data Types:** This specifies the kinds of data that the ISA's instructions can operate on.[8] For a toy ISA, simplicity is paramount. Often, a single integer data type (e.g., 8-bit or 16-bit, possibly signed or unsigned) is sufficient.[10] More complex types like floating-point numbers are usually omitted.

The careful definition of these components is crucial. For instance, the "Easy ISA" example, while aiming for simplicity, chose 64-bit integers and a 64-bit address space, which are quite large for a typical "toy" learning project and would significantly complicate the assembler if all features were fully utilized.[10] A more constrained design, such as 8 or 16-bit registers and addresses, would be more appropriate for a first toy ISA project.

B. Defining Instruction Formats: Fixed vs. Variable Length.

Instruction formats dictate the binary layout of machine instructions—how opcodes, operand specifiers, and immediate values are arranged within an instruction word.[9] There are two primary approaches:

- **Fixed-Length Instructions:** All instructions have the same bit length (e.g., 16 bits, 32 bits). This approach significantly simplifies the process of fetching instructions from memory and decoding them, as the boundaries of each instruction are predictable.[9] Many RISC (Reduced Instruction Set Computer) architectures, like ARM in its traditional A32/T32 instruction sets, often feature fixed-length instructions.[9]
- **Variable-Length Instructions:** Instructions can have different lengths. This allows for more compact code, as simple instructions can be shorter, but it complicates the fetching and decoding logic because the assembler and CPU must determine the length of the current instruction to find the beginning of the next one.[9] The x86 architecture is a well-known example of an ISA with variable-length instructions.[9] The "Easy ISA" example also considered variable-sized arithmetic instructions, a choice that was critiqued for making field determination and hardware pre-fetching difficult.[10]

For a toy ISA and its corresponding assembler, a **fixed-length instruction format is**

strongly recommended. This simplicity in instruction fetching and decoding translates directly into simpler logic for the assembler, particularly during Pass 1 for Location Counter management and Pass 2 for code generation.

C. Crafting Opcodes and Mnemonics.

Once the types of operations are decided, they need binary representations (opcodes) and human-readable symbols (mnemonics).

- **Opcodes (Operation Codes):** These are the unique binary patterns within an instruction that tell the CPU which operation to perform (e.g., add, load, store, branch).[14] The number of bits allocated for the opcode field in the instruction format determines the maximum number of unique instructions the ISA can support. For example, a 4-bit opcode field allows for 24=16 distinct instructions. Opcodes should be assigned systematically.[10]
- **Mnemonics:** These are short, symbolic names that are easier for programmers to remember and use than raw binary opcodes.[2] Examples include ADD for addition, LDA for "Load Accumulator," or JMP for "Jump." The assembler will be responsible for mapping these mnemonics to their corresponding binary opcodes using an Operation Code Table (OPTAB). The "Easy ISA" example assigned numerical codes like 000 for add, 001 for addf, etc., which would then be part of the binary instruction.[10]

A small, well-chosen set of instructions is vital for a toy ISA. Each instruction should have a clear purpose and contribute to the ISA's ability to perform basic computations and control flow.

D. Specifying Operand Types and Addressing Modes for Simplicity.

Instructions operate on data, and the way they specify this data is determined by operand types and addressing modes.

- **Operand Types:** These define the nature of the data an instruction can work with. Common types include:
  - **Registers:** The instruction operates on data held in one or more CPU registers.
  - **Immediate Values:** The data itself is embedded directly within the instruction.
  - **Memory Locations:** The instruction operates on data stored in main memory.
- **Addressing Modes:** These define how the instruction calculates or specifies the

effective address of an operand.[9] For a toy ISA, limiting the number and complexity of addressing modes is crucial for keeping the assembler design manageable. Some simple and common modes include:

- **Immediate Addressing:** The operand is a constant value encoded directly in the instruction (e.g., ADD R1, #5, where #5 is the immediate value).[9] The size of the immediate field in the instruction format needs to be defined.
- **Register Direct Addressing:** The operand is contained in a specified CPU register (e.g., ADD R1, R2, where R1 and R2 refer to registers).[9]
- **Memory Direct Addressing (Absolute Addressing):** The instruction contains the full memory address of the operand (e.g., LOAD R1, [0x100], where 0x100 is the absolute memory address).[9] This requires an address field in the instruction large enough to hold a complete memory address.
- **Register Indirect Addressing:** The instruction specifies a register that holds the memory address of the operand (e.g., LOAD R1,, where R2 contains the address of the data to be loaded).[9]

More complex addressing modes like indexed addressing (base register + index register + offset) or relative addressing (offset from the current Program Counter) significantly increase the complexity of both the CPU hardware and the assembler's operand parsing and encoding logic.[9] The "Easy ISA" example chose displacement and register indirect modes for memory addressing.[10] However, its critique of a 64-bit displacement field as being too large and wasteful for many common cases serves as an important lesson: constants and offsets in a toy ISA should be sized appropriately to avoid unnecessary complexity and instruction bloat.[10]

The careful selection of a few simple, orthogonal addressing modes will make the ISA easier to learn and the assembler easier to implement. Orthogonality, in this context, means that instructions, registers, and addressing modes can be combined in a consistent and predictable manner. While full orthogonality can lead to an explosion of instruction variants, a degree of regularity—where similar instructions support similar addressing modes—can simplify the assembler's logic by reducing the number of special cases it needs to handle. For instance, if an ADD instruction supports register-to-register and register-to-immediate operations, it would be logical for a SUB instruction to support the same. This thoughtful, limited application of regularity is beneficial even in toy ISA design.

Furthermore, during the ISA design phase, particularly when defining data types and the memory model, it is prudent to anticipate the types of assembler directives that

will be needed. Assembler directives, such as those for defining data (DB for Define Byte, DW for Define Word) or reserving memory space (RESB for Reserve Byte), are not machine instructions themselves but commands to the assembler.[4] If the ISA is designed to support 8-bit and 16-bit data, then corresponding directives like DB and DW become natural and necessary tools for the programmer, and the assembler must be designed to handle them. This foresight links the ISA's capabilities with the assembler's feature set.

E. Table 1: Key Design Choices for Your Toy ISA

To consolidate these decisions, a table outlining the specific characteristics of the toy ISA is invaluable. This table serves as the definitive specification for the hardware architecture that the assembler will target.

ISA Component	Your Toy ISA Specification (Example)	Rationale/Notes
Registers	4 general-purpose 8-bit registers (R0, R1, R2, R3), 8-bit PC, 8-bit SP	Small number for simplicity; 8-bit matches data size. PC/SP for basic functions.
Word Size	8 bits	Simplifies data handling.
Address Space	256 bytes (28)	Manageable for a toy system; addresses fit in 8 bits.
Data Types	8-bit signed integers	Single data type to reduce complexity.
Instruction Length	Fixed, 16 bits (2 bytes)	Simplifies assembler parsing and CPU fetching/decoding.
Endianness	Little-Endian	For multi-byte data (e.g., 16-bit addresses in instructions).

Key Instructions & Mnemonics	LOAD Rd, [addr] (Load R_d from memory)	Core data movement.
	STORE Rs, [addr] (Store R_s to memory)	Core data movement.
	MOV Rd, Rs (Move R_s to R_d)	Register transfer.
	ADDI Rd, imm (Add immediate to R_d)	Basic arithmetic.
	ADDR Rd, Rs (Add R_s to R_d)	Basic arithmetic.
	SUBI Rd, imm (Subtract immediate from R_d)	Basic arithmetic.
	SUBR Rd, Rs (Subtract R_s from R_d)	Basic arithmetic.
	JMP addr (Jump to address)	Unconditional control flow.
	JZ Rs, addr (Jump to address if R_s is zero)	Conditional control flow.
	HALT (Stop execution)	Program termination.
Opcodes (Example 4-bit)	LOAD: 0001, STORE: 0010, MOV: 0011, ADDI: 0100, etc.	Unique binary code for each operation.
Addressing Modes Supported	Immediate (for ADDI, SUBI), Register Direct (for MOV, ADDR, SUBR, JZ), Memory Direct (for LOAD, STORE, JMP, JZ address part)	Limited set for simplicity.

Instruction Formats (Example)	**Type 1 (Mem Op):** `OP (4b)	Rd (2b)
	Type 2 (Reg Op): `OP (4b)	Rd (2b)
	Type 3 (Imm Op): `OP (4b)	Rd (2b)
	Type 4 (JMP Op): `OP (4b)	Unused(4b)
	Type 5 (JZ Op): `OP (4b)	Rs (2b)
	Type 6 (HALT): `OP (4b)	Unused (12b)` (HALT)

This detailed specification is paramount. The "toy" nature of the ISA is a critical design constraint that directly impacts the feasibility and complexity of the assembler project. Each feature, such as a new addressing mode or the decision for variable-length instructions, can exponentially increase the assembler's parsing logic, symbol resolution challenges, and code generation rules.[10] Thus, a conscious effort to simplify the ISA is essential for keeping the assembler project itself manageable and educational.

III. The Architecture of a Two-Pass Assembler

Most assemblers, especially those needing to handle forward references efficiently, employ a two-pass architecture. This approach involves scanning the assembly source code twice, with each pass performing distinct tasks.

A. Why Two Passes? The Forward Reference Problem.

A significant challenge in assembling code is the "forward reference." This occurs when an instruction refers to a symbol (typically a label) that is defined later in the source program.[16] For example, a jump instruction like JMP LOOP_END might appear before the label LOOP_END: is defined.

If an assembler were to process the code in a single pass, it would encounter the JMP LOOP_END instruction without knowing the memory address of LOOP_END. This

poses a problem because the assembler needs this target address to generate the correct machine code for the jump. In some ISAs, the size of the jump instruction itself can vary depending on how far away the target label is (e.g., a short jump for nearby labels versus a long jump for distant ones).[16] A one-pass assembler would either have to guess the instruction size (and risk errors if the guess is wrong, requiring code regeneration) or always use the largest possible instruction size, leading to inefficient code.[16]

The two-pass assembler elegantly solves this issue [1]:

- **Pass 1:** The primary purpose of the first pass is to scan the entire source program to identify all symbols (labels) and determine their memory addresses. This information is stored in a data structure called the Symbol Table (SYMTAB). During this pass, the assembler calculates the size of each instruction and data area to correctly assign addresses, but it does not generate the final machine code.[16]
- **Pass 2:** The second pass re-scans the source program. With the completed SYMTAB from Pass 1, the assembler now has the addresses of all symbols. It can then translate each assembly instruction into its binary machine code equivalent, correctly resolving all forward references by looking up the symbol addresses in the SYMTAB.[16]

This two-pass strategy ensures that all symbol addresses are known before machine code generation begins, allowing for accurate and efficient translation.[16]

B. Pass 1: Symbol Definition and Address Assignment.

The overarching goal of Pass 1 is to construct the Symbol Table (SYMTAB) and assign a memory address to every instruction and data definition in the assembly program. No executable machine code is produced during this pass; the focus is solely on address calculation and symbol recording.

- 1. Lexical Analysis: Tokenizing the Assembly Code.
  The initial step in Pass 1 involves lexical analysis, where the assembler reads the source program character by character and groups these characters into a sequence of meaningful units called tokens.3 These tokens represent the basic building blocks of the assembly language, such as labels (e.g., START_LOOP), mnemonics (e.g., LOAD, ADD), operands (e.g., register names like R1, immediate values like #100, symbols like DATA_VAR), assembler directives (e.g., ORG, DB), and punctuation (e.g., colons, commas).4 For instance, a line of assembly code like LOOP: MOV R1, #10 would be tokenized into: LOOP (label), : (punctuation),

MOV (mnemonic), R1 (register operand), , (punctuation), #10 (immediate operand).

- 2. Building the Symbol Table (SYMTAB).
  As Pass 1 processes the tokenized input, it identifies symbols, primarily labels appearing in the label field of statements. When a label is encountered, the assembler records the symbol name and its corresponding memory address—the current value of the Location Counter (LOCCTR)—into the SYMTAB.18 The SYMTAB serves as a dictionary for all user-defined symbols.
  An essential function during this phase is error checking, specifically for duplicate symbols. If the assembler encounters a label that already exists in the SYMTAB, it flags a "redefinition of symbol" error.18
- 3. Managing the Location Counter (LOCCTR).
  The Location Counter (LOCCTR) is a variable maintained by the assembler that keeps track of the memory address for the current instruction or data item being processed.4 It is initialized at the beginning of the assembly process, typically to zero or to a value specified by an assembler directive like ORG.
  After each source statement is analyzed, the LOCCTR is incremented by the size of the machine instruction or data area that the statement will generate.18 For an ISA with fixed-length instructions, this increment is constant for all instructions. For ISAs with variable-length instructions, or when processing data definition directives, the assembler must determine the specific size from an Operation Code Table (OPTAB) or the directive itself.21 The accurate management of the LOCCTR is fundamental, as it directly determines the addresses assigned to labels in the SYMTAB and ensures the correct memory layout of the program. Any inaccuracies in LOCCTR updates will lead to a misaligned and non-functional program.
- 4. Handling Assembler Directives (Initial Processing).
  Certain assembler directives directly influence address assignment and symbol definition and therefore must be processed, at least partially, during Pass 1.4
  - ORG (Origin): This directive explicitly sets the LOCCTR to a specified value, changing the starting address for subsequent code or data.[23]
  - EQU (Equate) or SET: These directives assign a specific value (which could be a constant or an address) to a symbol. This symbol and its value are entered into the SYMTAB.[23]
  - Data Definition Directives (DB, DW, etc.) and Reservation Directives (RESB, RESW, etc.): When these directives are encountered, any associated label is entered into the SYMTAB with the current LOCCTR value. The LOCCTR is then

incremented by the amount of memory space defined or reserved by the directive.[21] The actual binary data for DB or DW is typically generated in Pass 2. The END directive usually signals the termination of Pass 1 processing.

At the conclusion of Pass 1, the SYMTAB contains all defined symbols and their addresses, and the total length of the program has been determined. Some assemblers may generate an intermediate representation of the program at this stage—a processed version of the source code, perhaps as a list of tokenized instructions with their assigned addresses and types.[25] This intermediate file can then be used as the input for Pass 2, potentially streamlining its operations by avoiding the need to re-tokenize and re-parse the original source code.[21]

C. Pass 2: Machine Code Generation.

The primary objective of Pass 2 is to take the information gathered in Pass 1 (especially the completed SYMTAB) and generate the final, executable machine code.

- 1. Parsing Instructions.
  The assembler reads the source program again, either from the original file or from the intermediate representation created in Pass 1. Each line is parsed to identify its components: label (usually ignored in Pass 2 for code generation purposes, but used for the listing file), mnemonic, and operands. This step is analogous to the syntax analysis phase in compilers, ensuring the instruction structure is valid.3
- 2. Resolving Symbols using SYMTAB.
  A crucial step in Pass 2 is the resolution of symbolic operands. When an instruction's operand is a symbol (e.g., DATA_VAR in LOAD R1, DATA_VAR, or TARGET_LABEL in JMP TARGET_LABEL), the assembler looks up this symbol in the SYMTAB to retrieve its memory address (or value, in the case of symbols defined with EQU).16 This retrieved address is then used in the formation of the machine instruction. If a symbol used as an operand is not found in the SYMTAB, an "undefined symbol" error is reported.19
- 3. Translating Mnemonics to Opcodes (using OPTAB).
  For each assembly instruction, the mnemonic is looked up in the Operation Code Table (OPTAB). The OPTAB provides the corresponding binary opcode for that instruction.4 Depending on the ISA and OPTAB design, it might also provide information about the instruction's expected format, length, and the types of operands it takes.
- 4. Encoding Operands into Machine Instructions.

Once the opcode is known and symbolic operands are resolved to addresses or values, these components must be encoded into their binary representations and placed into the correct fields within the machine instruction word. This encoding is strictly dictated by the instruction formats defined for the ISA.11
    - **Register Operands:** Register names (e.g., R0, R1) are converted into their numerical binary representations (e.g., 00, 01).
    - **Immediate Operands:** Immediate values are converted into their binary form, fitting the size of the immediate field in the instruction format.
    - **Memory Operands (Addresses):** Resolved addresses (from SYMTAB or direct numerical values) are converted to binary and placed in the address field(s) of the instruction. The precise bit patterns and their placement are specific to each instruction format within the toy ISA.
- 5. Generating the Final Machine Code.
  The assembled binary instructions, along with any data generated by directives like DB or DW, are written to an output file. This file can be in various formats, such as raw binary (.bin), Intel HEX (.hex), or a more complex object file format if linking is intended.16 Additionally, a listing file is often generated. This human-readable file typically shows each original assembly language statement alongside its assigned memory address and the generated machine code (usually in hexadecimal), as well as any error messages.16

Error handling is an ongoing process across both passes. Pass 1 typically detects syntax errors, illegal mnemonics, and redefined symbols.[18] Pass 2 is where errors like undefined symbols or operand mismatches (that depend on symbol resolution) are usually caught.[19] A common strategy is to terminate the assembly if critical errors are found in Pass 1, as these would prevent the correct construction of the SYMTAB, rendering Pass 2 ineffective.[18] If Pass 1 completes with only warnings or no errors, Pass 2 proceeds.

D. Table 2: Overview of Two-Pass Assembler Operations

The distinct roles and interactions of the two passes can be summarized as follows:

Pass	Primary Goals	Key Activities	Data Structures Utilized/Updated
Pass 1	Define all symbol	Read source,	SYMTAB (Write),

	addresses, calculate total program length.	tokenize, update Location Counter (LOCCTR), populate Symbol Table (SYMTAB), process address-affecting directives (ORG, EQU, RESB/W), generate intermediate file/data.	OPTAB (Read), LOCCTR (Update), LITTAB (Write, if used).
Pass 2	Generate executable machine code, resolve forward references, produce listing.	Read source/intermediate file, parse instructions, look up symbols in SYMTAB, look up mnemonics in OPTAB, encode operands, assemble instruction bytes, generate data from DB/DW, format output files (.bin,.hex,.lst).	SYMTAB (Read), OPTAB (Read), LITTAB (Read, if used), Output Buffers (Write).

This two-pass architecture provides a robust framework for translating assembly language into machine code, effectively handling the complexities introduced by symbolic addressing and forward references.

IV. Essential Data Structures for Your Assembler

A well-designed assembler relies on several key data structures to manage information during the translation process. These structures store details about operation codes, symbols, literals, and the current assembly address. For a toy assembler, these can be implemented simply, yet understanding their roles is crucial.

A. Operation Code Table (OPTAB): Mapping Mnemonics to Machine Operations.

- **Purpose:** The OPTAB is a fundamental data structure that stores the mapping between the symbolic mnemonics of assembly language instructions and their corresponding binary machine operation codes (opcodes).[4] It serves as the assembler's dictionary for valid instructions.

- **Content:** Each entry in the OPTAB typically contains:
  - The instruction mnemonic (e.g., LOAD, ADD, JMP). This often serves as the key for lookups.
  - The numerical opcode (in binary or hexadecimal).
  - Information about the instruction's format (e.g., number of operands, types of operands expected).
  - The length of the instruction in bytes (especially important if the ISA has variable-length instructions, though for a toy ISA with fixed-length instructions, this might be implicit or uniform).
- **Implementation:** For efficiency, especially in assemblers handling large instruction sets, the OPTAB is often implemented as a hash table, using the mnemonic as the key.[4] However, for a toy assembler with a small, fixed set of instructions (e.g., 10-20 mnemonics), a simpler implementation like an array of structures or a list that is searched linearly or via binary search (if sorted) might be perfectly adequate and easier to implement. The OPTAB is generally a static table, meaning its contents are defined when the assembler is created and do not change during the assembly of a program.
- **Usage:**
  - **Pass 1:** The OPTAB is consulted to validate if a mnemonic encountered in the source code is a legitimate instruction. It also provides the instruction's length, which is necessary for correctly incrementing the Location Counter (LOCCTR).[4]
  - **Pass 2:** The OPTAB is used to retrieve the actual binary opcode and instruction format details needed to construct the machine code instruction.[4]

B. Symbol Table (SYMTAB): Tracking Labels and Variables.

- **Purpose:** The SYMTAB is a dynamic data structure built by the assembler to keep track of all user-defined symbols and their associated values (typically memory addresses).[4] Symbols include labels that mark specific memory locations (e.g., for jump targets or data) and symbols defined via directives like EQU.
- **Content:** Each entry in the SYMTAB typically contains:
  - The symbol name (e.g., LOOP_START, COUNTER). This serves as the key.
  - The value associated with the symbol (usually its memory address as determined by the LOCCTR, or a constant value if defined by EQU).
  - Possibly other attributes or flags, such as the symbol's type (e.g., label, constant), its length if it represents a data area, or error flags (e.g., to indicate if a symbol has been defined multiple times).[4] Storing such type information

allows the assembler to treat symbols appropriately; for example, an EQU-defined constant would be substituted directly, whereas a label's address would be used in a jump instruction.

- **Implementation:** Like the OPTAB, the SYMTAB is often implemented as a hash table to allow for efficient insertion of new symbols (during Pass 1) and fast lookups (during Pass 1 to check for duplicates, and during Pass 2 to resolve symbolic references in operands).[4] For a toy assembler handling a small number of symbols, simpler structures could also be considered, trading some performance for ease of implementation.
- **Usage:**
  - **Pass 1:** The SYMTAB is populated as the assembler encounters labels in the source code. The current value of the LOCCTR is typically assigned as the address for these labels.[4] It is also checked for duplicate symbol definitions.
  - **Pass 2:** The SYMTAB is read to find the addresses (or values) of symbols that appear in the operand fields of instructions or directives. This resolved address/value is then used to generate the machine code.[4]

C. Literal Table (LITTAB) and Pool Table (POOLTAB) (Brief Overview).

- **Purpose:**
  - **LITTAB (Literal Table):** Manages literals, which are constants specified directly within an instruction, often prefixed by an equals sign (e.g., =C'EOF' or =X'01'). Unlike immediate operands, literals are grouped together in memory in a "literal pool," and the instruction contains the address of the literal in this pool.[4]
  - **POOLTAB (Pool Table):** Used in conjunction with LITTAB, especially when assembler directives like LTORG are used. LTORG instructs the assembler to place the collected literal pool at the current location. POOLTAB helps manage multiple literal pools if they exist.[25]
- **Context:** For a very simple toy assembler, explicit LITTAB and POOLTAB management might be an advanced feature. If the toy ISA only supports immediate operands (where the constant is part of the instruction itself) and does not have a distinct concept of pooled literals, these tables may not be necessary. However, if the ISA design allows for defining data constants that are referenced by instructions but stored separately, a LITTAB would be required.
- **Usage (if implemented):**
  - **Pass 1:** When a literal is encountered, it is added to the LITTAB (if not already present). The LITTAB stores the literal itself and later, its assigned address

when the literal pool is generated (typically at an LTORG directive or at the end of the program).[4]
    - **Pass 2:** When an instruction referencing a literal is processed, the assembler looks up the literal's address in the LITTAB and uses this address in the generated machine code. The actual byte values for the literals are also generated and placed in the object code at the literal pool's location.

D. Location Counter (LOCCTR): Keeping Track of Addresses.

- **Purpose:** The LOCCTR is not a table but a crucial variable maintained by the assembler, primarily during Pass 1. Its value represents the memory address of the instruction or data item currently being processed or the next available address for allocation.[4]
- **Functionality:**
    - **Initialization:** The LOCCTR is initialized at the beginning of the assembly process. This initial value is typically 0, unless a directive like ORG or START specifies a different starting address.
    - **Incrementation:** As each assembly statement is processed in Pass 1, the LOCCTR is incremented by the number of bytes that the corresponding machine instruction or data area will occupy in memory. For example, if instructions are 2 bytes long, the LOCCTR increments by 2 after each instruction. A directive like RESB 10 would cause it to increment by 10.
    - **Address Assignment:** The value of the LOCCTR at the time a label is encountered is the address assigned to that label in the SYMTAB.[18]
- **Usage:** Primarily used in Pass 1 to assign addresses to symbols and to calculate the total length of the program. It is also referenced in Pass 2 to generate the address information in the listing file.

E. Table 3: Core Data Structures in an Assembler

The roles and interactions of these data structures are summarized below:

Data Structure	Purpose	Typical Content	Primary Pass(es) Utilized/Modified
OPTAB	Store mnemonic-to-opcod e mappings and	{Mnemonic (key), Opcode, Instruction Format, Instruction	Created before assembly; Read in Pass 1 (validation,

	instruction characteristics.	Length}	length); Read in Pass 2 (opcode, format).
SYMTAB	Store user-defined symbol-to-address/value mappings.	{SymbolName (key), Address/Value, Type, Flags (e.g., defined, error)}	Written in Pass 1 (as symbols are defined); Read in Pass 1 (check duplicates); Read in Pass 2 (resolve symbols).
LITTAB (if used)	Manage literals and their assigned addresses.	{LiteralString (key), Value, Length, Address}	Written in Pass 1 (as literals are encountered); Read in Pass 2 (resolve literal references).
LOCCTR (Variable)	Keep track of the current memory address during assembly.	Current memory address (integer value).	Initialized and Updated extensively in Pass 1; Read in Pass 2 (for listing).

These data structures work in concert, enabling the assembler to systematically analyze the source program, resolve symbolic references, and generate the correct machine code. The choice of their specific implementation (e.g., hash table vs. array) can affect the assembler's performance and development complexity, with simpler structures often being suitable for the limited scope of a toy assembler project.

V. The Translation Process: From Assembly to Machine Code

The transformation of assembly language statements into binary machine code is a methodical process involving several distinct stages within the assembler. This section details these stages, focusing on how mnemonics and operands are converted and assembled according to the defined Instruction Set Architecture (ISA).

A. Lexical Analysis: Breaking Down the Source.

As previously outlined for Pass 1, the very first step in processing any assembly language source file is lexical analysis, or tokenization.[3] The assembler reads the input

program as a stream of characters and divides each line into a sequence of tokens. Tokens are the smallest meaningful units of the language. For a typical assembly language, these include:

- **Labels:** Identifiers that mark a specific line or memory location (e.g., LOOP_START).
- **Mnemonics:** Symbolic representations of machine operations (e.g., LDA, ADD, STO).
- **Operands:** Data or addresses that the instruction operates on. These can be registers (e.g., R1, %ACC), immediate values (e.g., #$10, 25), symbols (e.g., MY_VARIABLE, JUMP_TARGET), or memory references (e.g., [0x100], ``).
- **Directives:** Commands to the assembler itself (e.g., ORG, DB, END).
- **Punctuation:** Characters like colons (after labels), commas (separating operands), and comment indicators.

For example, the assembly line START: LDA R1, #$10 ; Load immediate value would be tokenized into something like: START (Label), : (Punctuation), LDA (Mnemonic), R1 (Operand-Register), , (Punctuation), #$10 (Operand-Immediate), and the comment would typically be noted and then ignored for subsequent processing. This token stream is then passed to the next phase.

B. Syntax Analysis: Validating Instruction Structure.

Following lexical analysis, the assembler performs syntax analysis (often called parsing) on the stream of tokens.[3] This phase checks whether the sequence of tokens for each statement conforms to the grammatical rules of the assembly language and the specific requirements of the target ISA. Key checks include:

- **Correct Mnemonic:** Is the operation mnemonic a valid one defined in the OPTAB?
- **Operand Count:** Does the instruction have the correct number of operands for the given mnemonic? (e.g., ADD might require two operands, HALT might require zero).
- **Operand Types:** Are the types of the provided operands compatible with what the instruction expects? (e.g., if an instruction expects a register, an immediate value would be an error).
- **Addressing Mode Validity:** Is the way operands are specified (addressing mode) valid for this particular instruction?
- **Directive Syntax:** Are assembler directives used with the correct syntax and

arguments?

For example, if the toy ISA defines an ADD instruction that only operates on two registers (ADDR Rd, Rs), then an assembly statement like ADDR R1, #5 would be flagged as a syntax error because the second operand is an immediate value, not a register. Similarly, if an instruction format expects a memory address, providing only a register might be incorrect. For a toy assembler, syntax analysis might not involve building a complex parse tree or Abstract Syntax Tree (AST) as in full compilers [20]; instead, it might be implemented as a set of validation rules applied directly to the token list for each instruction. This phase ensures that instructions are well-formed before the assembler attempts to generate machine code.

C. Mapping Mnemonics to Opcodes.

During Pass 2, a core part of the translation is converting the instruction mnemonic into its numerical machine opcode. This is achieved by looking up the mnemonic token (e.g., LDA) in the Operation Code Table (OPTAB).[4] The OPTAB entry for the mnemonic provides the corresponding binary or hexadecimal opcode.[14] For instance, if the toy ISA specifies that the LDA instruction has an opcode of 0110b, the OPTAB will store this association. This numerical opcode forms a key part of the final machine instruction word. The OPTAB might also yield other information relevant to encoding, such as the instruction's specific format type if the ISA has multiple formats.

D. Encoding Operands.

Once the mnemonic is validated and its opcode retrieved, the assembler must process and encode the instruction's operands. This involves converting each operand from its symbolic or literal representation in the assembly source into the binary format required by the instruction and placing it into the appropriate field(s) within the machine instruction word.[26] The encoding process depends heavily on the operand type and the addressing modes defined by the ISA.

- 1. Register Operands:
  If an operand is a register (e.g., R1, %R2), the assembler converts the register identifier into its numerical representation as defined by the ISA (e.g., R0 -> 00, R1 -> 01, R2 -> 10, R3 -> 11 for an ISA with 4 registers requiring 2 bits for encoding). This binary number is then placed into the designated register field(s) of the machine instruction word.26
- 2. Immediate Values:

If an operand is an immediate value (e.g., #$10, 25), the assembler converts this numerical value into its binary representation.26 This binary value is then placed into the immediate data field of the instruction. The size of this field (number of bits) is predetermined by the ISA's instruction format. The assembler must also handle potential size mismatches (e.g., an 8-bit immediate value provided for a 4-bit field would be an error).

- 3. Memory Addresses (Labels, Variables, Direct Addresses):
  Encoding memory address operands is often the most complex part:
  - **Symbolic Addresses (Labels/Variables):** If an operand is a symbol (e.g., JMP LOOP_START or LOAD R1, COUNT_VAR), its memory address is retrieved from the Symbol Table (SYMTAB), which was populated during Pass 1.[17] This numeric address is then converted to binary and encoded into the address field of the instruction.
  - **Direct Numerical Addresses:** If the assembly code specifies a direct numerical address (e.g., LOAD R1, [0x100]), this number is converted to its binary form and placed in the instruction's address field. The specific encoding depends on the addressing mode. For direct addressing, the full (or partial, if base-displacement is used) address is encoded. For register indirect addressing (e.g., LOAD R1,), the operand itself is a register (R2), and its encoding follows the register operand rules; the CPU then uses the content of R2 as the address at runtime.

A crucial consideration for multi-byte immediate values or addresses is **endianness**. If the toy ISA operates on, for example, 16-bit addresses or immediates but has a byte-addressable memory (or instructions are encoded as a sequence of bytes), the assembler must adhere to a defined byte order (big-endian or little-endian) when placing these multi-byte quantities into the machine code.[6] This should be a part of the ISA specification. For instance, if the ISA is little-endian and a 16-bit address 0x1234 needs to be encoded, the byte 0x34 would be stored before the byte 0x12.

E. Constructing the Binary Instruction Word based on ISA Formats.

After the opcode has been determined and all operands have been converted to their binary representations, these binary components are assembled into a single binary string—the machine instruction word. This assembly process strictly follows the predefined instruction format(s) of the ISA.11 Each component (opcode, register specifiers, immediate data, address fields) is placed into its designated bit positions within the instruction word.
For example, if a toy ISA has a 16-bit fixed instruction length and a particular instruction type has the format:
| Opcode (4 bits) | DestReg (2 bits) | SrcReg (2 bits) | Immediate (8 bits) |

An assembly instruction like ADDI R1, R2, #10 (assuming ADDI opcode is 0100b, R1 is 01b, R2 is 10b, and #10 is 00001010b) would be constructed by concatenating these binary pieces: 0100 01 10 00001010.
The choices made during ISA design—particularly regarding instruction formats, the number of different formats, and the complexity of addressing modes—have a profound and direct impact on the logic required within the assembler for parsing operands and constructing these binary instruction words.[12] A simple, regular ISA with few fixed-length formats significantly simplifies this stage of the assembler. Conversely, an ISA with many variable-length formats or complex addressing schemes (like the ModR/M byte in x86 [26]) would necessitate far more intricate logic in the assembler.

It is also important to recognize that errors can propagate through the assembly process. An error made in an earlier phase, such as an incorrect address calculation for a label in Pass 1 resulting in a faulty SYMTAB entry, will lead to the generation of incorrect machine code in Pass 2 when that label is referenced. This underscores the need for robust error checking at each stage of assembly to help pinpoint the original source of any problems.

F. Table 4: Example Toy ISA Instruction Encoding

Based on the example Toy ISA defined in Table 1, the following table illustrates the encoding process for a few sample instructions. Assume the opcodes are: LOAD=0001, ADDR=0101, STORE=0010, JMP=1000. Registers R0-R3 are encoded as 00-11.

Assembly Instruction Example	Instruction Format Name (from Table 1)	Breakdown of Fields (Bits)	Example Binary Encoding (16 bits)	Hex Representation
LOAD R1, 0x30	Type 1 (Mem Op)	OP(4):0001, Rd(2):01, Addr(8):00110000	0001 01 00110000 (Unused 2b)	1530
ADDR R2, R3	Type 2 (Reg Op)	OP(4):0101, Rd(2):10, Rs(2):11	0101 10 11 000000	5B00

STORE R0, (Assume R1 contains 0x45)	Type 1 (Mem Op)	OP(4):0010, Rs(2):00, Addr(8):01000101	0010 00 01000101 (Rs used as source for STORE, Addr from)	2045 (Conceptual, actual STORE might differ based on ISA table)
JMP LABEL_X (Assume LABEL_X is at 0xA2)	Type 4 (JMP Op)	OP(4):1000, Addr(8):10100010	1000 0000 10100010	80A2

(Note: The STORE R0, example above simplifies how the address from `` would be encoded. A true "Store Register to Address-in-Register" would likely have a different format or opcode. The table illustrates the general principle of filling fields based on a defined format. For the example ISA in Table 1, STORE Rs, [addr] implies addr is a direct 8-bit address. If STORE Rs, (store content of Rs to address in Rd) was an instruction, it might use a Type 2 format with a specific opcode.)

This table provides concrete examples of how symbolic assembly instructions are systematically translated into their binary machine code equivalents, bridging the gap between human-readable code and hardware-executable instructions.

VI. Handling Assembler Directives

Assembler directives, also known as pseudo-operations or pseudo-ops, are commands embedded in the assembly source code that instruct the assembler on how to perform the assembly process. Unlike assembly language instructions, directives are not translated into executable machine instructions for the CPU.[4] Instead, they control aspects like data definition, memory allocation, symbol definition, program organization, and output formatting.

A. Purpose of Directives (vs. Instructions).

The fundamental distinction is that instructions tell the CPU what to do at *runtime*, while directives tell the assembler what to do at *assembly time*.[4] Directives manage the environment and context in which instructions are assembled. They allow the programmer to:

- Define constants and variables.
- Allocate and initialize memory for data.
- Reserve blocks of memory.
- Control the location counter (i.e., where code and data are placed in memory).
- Define program segments (e.g., code segment, data segment).
- Mark the beginning and end of the program.
- Include other source files.
- Define macros (though macros are an advanced feature often beyond a basic toy assembler).

By providing these capabilities, directives form a sort of "meta-language" that the assembler itself understands and acts upon, enabling more structured and manageable assembly language programming. The assembler, therefore, plays a dual role: it is a translator for CPU instructions and an interpreter for these directive commands.

B. Common Data Definition Directives.

These directives are used to allocate memory space and optionally initialize it with specific values.

- **DB (Define Byte):** This directive allocates one or more consecutive bytes of memory and initializes them with specified values.[15] Values can be numerical (e.g., 0x12, 255) or character strings (e.g., 'Hello').
  - Example: MY_BYTE DB 0x41 allocates one byte and stores the value 4116 (ASCII 'A').
  - Example: MESSAGE DB 'Hi', 0x0D, 0x0A allocates three bytes for 'H', 'i', carriage return, and line feed.
- **DW (Define Word):** This directive allocates memory for one or more words and initializes them.[15] The size of a "word" is ISA-dependent (e.g., 16 bits, 32 bits).
  - Example: DATA_WORD DW 0x1234 (for a 16-bit word ISA) allocates two bytes and stores 123416. The assembler must handle endianness if the word size is greater than the character size (byte).
- **RESB (Reserve Byte) / DS (Define Storage - byte):** This directive reserves a specified number of bytes of memory without initializing them.[21]
  - Example: BUFFER RESB 100 reserves 100 bytes of memory, labeled BUFFER.
- **RESW (Reserve Word) / DSW (Define Storage - word):** This directive reserves a specified number of words of memory without initializing them.[21]
  - Example: ARRAY RESW 50 reserves 50 words of memory for an array.

C. Program Control and Symbol Definition Directives.

These directives influence the assembly process, control program structure, and define symbols.

- **ORG (Origin):** Sets the Location Counter (LOCCTR) to a specified address.[23] Subsequent instructions or data will be assembled starting from this new address.
  - Example: ORG 0x0100 tells the assembler to start assembling the following code at memory location 010016.
- **EQU (Equate):** Assigns a constant value to a symbol.[15] The symbol then represents this value, not a memory location. This is useful for defining named constants.
  - Example: MAX_SIZE EQU 100. Wherever MAX_SIZE is used later, the assembler substitutes the value 100.
- **SET:** Similar to EQU, but often allows the symbol to be redefined later in the program.[23]
- **END:** Marks the end of the assembly language source program.[4] It signals to the assembler that there is no more code to process. Optionally, it can specify the starting execution address for the program (e.g., END START_LABEL), which is then included in the object file for the loader.
- **SEGMENT and PROC (Procedure):** These directives are used in more advanced assemblers to define logical program segments (like code, data, stack segments) and procedures or functions.[15] For a simple toy assembler, these might be omitted or greatly simplified.

D. Processing Directives in Each Pass.

The two-pass nature of the assembler means directives are processed at different stages, depending on their function:

- **Pass 1:**
  - ORG: Processed immediately to set the LOCCTR to the specified value. This affects the addresses assigned to all subsequent labels and instructions.
  - EQU / SET: The symbol and its defined value are entered into the Symbol Table (SYMTAB). This is crucial because these symbols might be used in subsequent instructions or directives within Pass 1 itself (e.g., BUFFER_SIZE EQU 10, MY_BUFFER RESB BUFFER_SIZE).
  - DB, DW, RESB, RESW: If these directives have a label, the label and the current LOCCTR value are entered into the SYMTAB. The LOCCTR is then incremented

by the number of bytes defined or reserved by the directive. The actual binary data for DB or DW is not typically generated in Pass 1, only space allocation is handled.
    - END: Signals the end of source input for Pass 1. The assembler can then calculate the total program length.
- **Pass 2:**
    - ORG, EQU, SET: These directives usually do not require further action in Pass 2 regarding code generation, but they are included in the assembly listing.
    - DB, DW: The actual binary values specified in these directives are generated and written to the object code output stream at the addresses determined in Pass 1.
    - RESB, RESW: These directives typically do not result in any data being written to the object file by the assembler itself (as they reserve uninitialized space). The loader would be responsible for allocating this space in memory when the program is loaded. The assembler might simply advance its current position in the conceptual object file image.
    - END: May provide the execution start address, which is included in the object file (e.g., in an End record for Intel HEX or object module formats).

The need to process certain directives in specific passes underscores the logic of the multi-pass design. For instance, EQU must be handled in Pass 1 so that the equated symbol's value is available if it's used later in Pass 1 to define the size of a reserved memory block. Similarly, ORG must modify the LOCCTR in Pass 1 to ensure all subsequent labels receive correct addresses. This careful staging of directive processing is essential for the correct functioning of the assembler.

VII. Generating Output

Once the assembler has processed the assembly language source code through its passes, its final task is to produce output files. The primary output is the machine code itself, but other useful files, like a listing file, are also commonly generated.

A. Common Output Formats.

Assemblers can produce machine code in several formats, each suited for different purposes:

- **Raw Binary (.bin):** This is a direct, byte-for-byte image of the executable machine code and any defined data, exactly as it would appear in the computer's

memory.[27] It contains no metadata, no addresses, and no checksums—just the pure binary content. Raw binary files are simple and are often used for loading directly into memory for execution by a simulator, for programming simple ROMs, or in bootloader scenarios where the loading mechanism is very basic.

- **Intel HEX (.hex):** This format represents binary information as ASCII text.[33] Each line (record) in an Intel HEX file contains hexadecimal characters encoding the byte count, starting address for the data in that record, a record type (indicating if it's data, end-of-file, an extended address, etc.), the data itself, and a checksum for error detection.[33] This format is widely used for programming EPROMs, microcontrollers, and other programmable logic devices because it's human-readable (to some extent) and can be easily transmitted over text-based communication channels. Common record types include 00 for data, 01 for end-of-file, 02 for extended segment address, and 04 for extended linear address.[33]
- **Object Files (.obj, .o):** These are more structured and complex formats designed to be processed by a linker.[27] Object files typically contain not only the machine code (often in segments or sections) but also:
    - **Symbol Table Information:** Names and addresses of defined symbols (labels, variables) that can be referenced by other modules (public symbols) or symbols that are defined in other modules but used in the current one (external symbols).
    - **Relocation Information:** Instructions or data that refer to addresses whose final values are not known until link time (e.g., references to external symbols or position-independent code). The linker uses this information to patch the addresses once all modules are combined.
    - **Debugging Information:** Data that allows a debugger to map machine code back to source lines, inspect variables, etc. Common object file formats include ELF (Executable and Linkable Format, used on Linux and many Unix-like systems), COFF (Common Object File Format), and Microsoft's PE (Portable Executable) format (which uses .obj for its object files).
- **Listing File (.lst):** This is a human-readable text file generated by the assembler that serves as a valuable diagnostic and debugging tool.[16] It typically shows:
    - The original assembly language source statements.
    - The memory address assigned to each statement by the assembler (from the LOCCTR).
    - The machine code (usually in hexadecimal) generated for each instruction or data directive.

    - The contents of the Symbol Table.
    - Any error or warning messages generated during assembly. The listing file is indispensable for understanding how the assembler translated the source code and for debugging both the assembly program and the assembler itself. By comparing the assembler's output (addresses, generated hex) against manual calculations or expected results for test programs, the assembler developer can verify the correctness of their SYMTAB, OPTAB, LOCCTR logic, and instruction encoding.

B. Choosing an Output Format for a Toy Assembler.

For a toy assembler project, especially a first attempt, simplicity in output generation is often preferred.

- **Raw Binary (.bin):** This is generally the easiest format to implement. The assembler simply writes out the generated machine code bytes sequentially. This format is ideal if the toy ISA will be run on a simple simulator that can load a memory image directly.
- **Intel HEX (.hex):** Implementing a subset of the Intel HEX format (e.g., only Data records (type 00) and an End-Of-File record (type 01)) can be a good learning exercise and is relatively straightforward. It makes the output somewhat more portable and inspectable than raw binary.
- **Listing File (.lst):** Regardless of the machine code format chosen, generating a comprehensive listing file is highly recommended. Its utility in debugging the assembler and the assembly programs written for the toy ISA cannot be overstated.

Full-fledged object file formats like ELF or COFF are usually too complex for an initial toy assembler project due to the need to handle relocation, extensive symbol table information, and various section types. The choice of output format should align with the intended use of the assembled code; for a purely educational assembler and an accompanying simple simulator, a combination of raw binary and a detailed listing file often provides the best balance of simplicity and utility.

VIII. Conceptual Steps to Build Your Toy Assembler

Building an assembler, even for a toy ISA, is a significant software engineering task. A structured, step-by-step approach is essential for managing complexity and ensuring a successful outcome. The following conceptual steps outline a practical path to

creating your toy assembler. An incremental development strategy—building and testing components one by one—is highly recommended over attempting to implement all features simultaneously.

A. Step 1: Finalize Your Toy ISA Definition.

Before any assembler code is written, the target Instruction Set Architecture (ISA) must be completely and unambiguously defined. This is the blueprint that the assembler will implement.

- **Action:** Solidify all aspects of your toy ISA as detailed in Section II and summarized in Table 1. This includes:
  - The set of registers (number, size, names).
  - The memory model (address space size, word size, endianness).
  - Supported data types.
  - Instruction formats (fixed vs. variable length, field layouts for opcode and operands).
  - A complete list of all instructions with their mnemonics and corresponding binary opcodes.
  - The addressing modes supported by each instruction.
- This definition document will be referenced constantly throughout the assembler development process.

B. Step 2: Implement Lexical Analyzer (Tokenizer).

The first functional component of the assembler is the lexical analyzer.

- **Action:** Develop a module (a function or class) that takes a line of assembly source code as input and outputs a list of tokens.[3] Each token should represent a meaningful unit, such as a label, mnemonic, register, immediate value, symbol, directive, or punctuation.
  - Consider how to handle comments (usually stripped out or marked as ignorable).
  - Define rules for recognizing different token types (e.g., labels end with a colon, immediates might start with '#' or '$', registers with '%').
  - For a toy assembler, this can often be achieved using string manipulation functions and regular expressions, if available and appropriate for the chosen implementation language. More formal tools like Lex/Flex are powerful but might be overkill for a simple project.
- **Testing:** Test the tokenizer thoroughly with various valid and invalid assembly

lines to ensure it correctly identifies and separates tokens.

C. Step 3: Implement Pass 1.

Pass 1 focuses on building the Symbol Table (SYMTAB) and assigning addresses.

- **Action:**
  1. **Initialize:** Set the Location Counter (LOCCTR) to its starting value (e.g., 0 or as specified by an ORG directive if encountered first). Initialize an empty SYMTAB and OPTAB (the OPTAB can be pre-populated with your ISA's instruction definitions).
  2. **Process Lines:** Read the input assembly file line by line. For each line:
     - Use the lexical analyzer to get the tokens.
     - **Label Handling:** If the first token is a label, add the label and the current value of the LOCCTR to the SYMTAB. Perform error checking for duplicate labels.[18]
     - **Mnemonic/Directive Identification:** Identify the instruction mnemonic or assembler directive.
     - **LOCCTR Update:**
       - If it's an instruction, determine its length (this is simple if all instructions are fixed-length). Increment LOCCTR by this length.
       - If it's a data definition directive (DB, DW), calculate the number of bytes allocated and increment LOCCTR accordingly.
       - If it's a reservation directive (RESB, RESW), increment LOCCTR by the number of bytes reserved.
       - If it's an ORG directive, set LOCCTR to the specified value.
       - If it's an EQU directive, add the symbol and its equated value to the SYMTAB. EQU does not advance LOCCTR.
     - **Error Checking:** Validate mnemonics against OPTAB and check for syntax errors in directives.
  3. **Intermediate Storage (Optional but Recommended):** Store the processed information for each line (e.g., token list, assigned LOCCTR value) in an intermediate data structure or file. This will be the input for Pass 2 and avoids re-lexing.[21]
  4. **Termination:** Continue until an END directive is encountered or the end of the file is reached. Record the total program length (final LOCCTR - starting address).
- **Testing:** After implementing Pass 1, test it with sample programs. Verify that the

SYMTAB is correctly populated with all labels and their addresses, and that the LOCCTR values assigned to each line are accurate. Check the handling of various directives.

D. Step 4: Implement Pass 2.

Pass 2 performs the actual translation into machine code.

- **Action:**
  1. **Initialize:** Reset any necessary counters or prepare output file(s).
  2. **Process Lines:** Read the input, line by line, from the original source or the intermediate representation generated by Pass 1. For each statement:
     - **Instruction Processing:**
       - Parse the instruction to identify the mnemonic and operands.
       - Look up the mnemonic in OPTAB to get its binary opcode and instruction format information.
       - For each operand:
         - If it's a symbol (label or EQUated symbol), look up its value (address or constant) in the SYMTAB. Report an error if the symbol is not found (undefined symbol).[19]
         - Convert the operand (register number, immediate value, resolved address/constant) into its required binary representation according to the ISA specification.
       - Assemble the binary opcode and the encoded binary operand(s) into the final machine instruction word, following the specific instruction format for that instruction type.[12]
     - **Data Directive Processing:** For directives like DB or DW, convert the specified data values into their binary representations.
     - **Output Generation:** Write the generated machine instruction or data bytes to the chosen output file (e.g., .bin or .hex). Also, generate a line for the listing file, showing the original assembly, the assigned address (from Pass 1 or current Pass 2 context), and the generated machine code in hexadecimal.
  3. **Termination:** Continue until the END directive or end of input. Finalize output files (e.g., write an End-of-File record for Intel HEX).
- **Testing:** Test Pass 2 with programs that utilize all defined instructions, addressing modes, and directives. Carefully examine the generated machine code (in the binary/HEX file and the listing file) to ensure it matches manually calculated

expected values.

E. Step 5: Testing and Debugging.

Thorough testing is crucial for a functional assembler.

- **Action:**
  1. **Create Test Cases:** Develop a comprehensive suite of small assembly language programs. Each program should be designed to test specific features:
     - Each instruction in the ISA.
     - Each addressing mode.
     - Each assembler directive.
     - Combinations of features.
     - Edge cases (e.g., zero values, maximum values, forward references, backward references).
     - Error conditions (e.g., invalid mnemonics, undefined symbols, duplicate labels, incorrect operand types).
  2. **Manual Assembly:** For each test case, manually determine the expected machine code output and the expected contents of the listing file.
  3. **Compare Results:** Run your assembler on the test cases. Compare the assembler's output (binary files, HEX files, listing files) against your manually derived expected results. Discrepancies indicate bugs in the assembler.
  4. **Use a Simulator (if available):** If a simulator exists for your toy ISA (as mentioned for ToyASM [5]), running the assembler's output on the simulator provides functional testing. This "closes the loop" by verifying that the generated code actually performs the intended operations. Even a very simple, custom-built simulator for the toy ISA can be invaluable. Such a simulator would typically involve an array to represent memory, variables for registers, and a main loop to fetch, decode, and execute the toy machine instructions from the loaded binary file.
- **Debugging:** Use print statements, a debugger, and the listing file itself to trace the assembler's logic and pinpoint the source of errors.

By following these steps incrementally and testing rigorously at each stage, the complex task of building an assembler can be made manageable and ultimately successful.

IX. Conclusion: Your Journey into Assembler Design

The development of an assembler, even for a simplified "toy" Instruction Set Architecture, is a rewarding project that offers substantial insights into the fundamental workings of computer systems. This endeavor bridges the conceptual gap between human-readable symbolic programming and the binary instructions that directly command a processor.

A. Recap of Key Learnings.

Throughout this guide, several core concepts have been emphasized:

- **The Primacy of ISA Definition:** The design of the Instruction Set Architecture is the bedrock upon which the assembler is built. Choices regarding instruction formats, opcodes, addressing modes, and the memory model directly dictate the complexity and logic of the assembler. A simpler, well-defined ISA leads to a more manageable assembler project.
- **The Two-Pass Architecture:** The canonical two-pass approach is crucial for effectively handling forward references—symbols used before they are defined. Pass 1 focuses on identifying all symbols and their memory addresses, populating a Symbol Table. Pass 2 then uses this table to generate the final machine code, resolving all symbolic references.
- **Essential Data Structures:** Key data structures like the Operation Code Table (OPTAB) for mapping mnemonics to opcodes, and the Symbol Table (SYMTAB) for tracking user-defined symbols and their addresses, are central to the assembler's operation. The Location Counter (LOCCTR) plays a vital role in address assignment during Pass 1.
- **Systematic Translation:** The conversion from assembly language to machine code is a methodical process involving lexical analysis (tokenization), syntax analysis (validation), opcode lookup, operand encoding, and the final construction of binary instruction words according to the ISA's formats.
- **The Role of Directives:** Assembler directives, distinct from machine instructions, provide control over the assembly process itself, enabling data definition, memory allocation, and program organization.

Successfully completing such a project provides a profound, practical understanding of how software instructs hardware. This knowledge is not merely academic; it is foundational for anyone seeking a deeper comprehension of operating systems, compiler design, embedded systems programming, or computer architecture itself. The process demystifies what happens "under the hood" when a program runs.

B. Potential Next Steps and Further Exploration.

Once a basic toy assembler is functional, several avenues for further exploration and enhancement can be pursued:

- **Advanced Assembler Features:**
  - **Macro Processing:** Implement a macro facility, allowing programmers to define reusable blocks of assembly code that can be invoked by a single macro name.[6] This typically involves adding a macro definition phase and an expansion phase, often before or integrated with Pass 1.
  - **Conditional Assembly:** Allow parts of the assembly code to be included or excluded based on conditions evaluated at assembly time. This is useful for creating different versions of a program from the same source.
  - **More Complex Expression Evaluation:** Allow operands to be more complex arithmetic or logical expressions involving symbols and constants, which the assembler would evaluate during Pass 2.
- **ISA and Assembler Evolution:**
  - **Expand the Toy ISA:** Introduce new instructions, more sophisticated addressing modes, or additional data types to the toy ISA, and then extend the assembler to support these new features. This mirrors the real-world evolution of ISAs and their corresponding toolchains.
  - **Develop a Linker:** If the assembler is enhanced to produce relocatable object files (a more advanced output format than raw binary or simple HEX), the next logical step would be to design and build a simple linker that can combine multiple object files into a single executable program.
- **Supporting Tools:**
  - **Simulator/Virtual Machine:** Develop or enhance a simulator for the toy ISA. This allows for robust testing of the assembler's output and provides a complete environment for running and debugging programs written for the toy architecture.[5]
  - **Debugger:** Create a simple symbolic debugger that can work with the assembler's output and listing files to allow step-by-step execution, inspection of registers and memory, and setting breakpoints.
- **Study of Real-World Systems:**
  - Analyze the design of real-world assemblers like NASM (Netwide Assembler) for the x86 architecture or GAS (GNU Assembler) which supports multiple ISAs.
  - Delve deeper into the intricacies of complex ISAs such as x86-64,

ARMv8/ARMv9, or RISC-V.

The journey of creating an assembler is an iterative one. The first version will likely be a starting point, and as understanding and ambition grow, the assembler and its surrounding ecosystem of tools can evolve. This iterative development process itself is a valuable lesson, reflecting how complex software systems are built and maintained in practice. Ultimately, constructing an assembler is more than just a programming exercise; it is an exploration into the heart of computation.

Works cited

1. What is instruction set architecture (ISA)? - Lenovo, accessed June 4, 2025, https://www.lenovo.com/us/en/glossary/instruction-set-architecture/
2. What defines a CPU and relation with its ISA and Assembly ... - Reddit, accessed June 4, 2025, https://www.reddit.com/r/hardware/comments/11cm8gk/what_defines_a_cpu_and_relation_with_its_isa_and/
3. Phases Of A Compiler | A Detailed Explanation (+Flowcharts) // Unstop, accessed June 4, 2025, https://unstop.com/blog/phases-of-a-compiler
4. elearningatria.files.wordpress.com, accessed June 4, 2025, https://elearningatria.files.wordpress.com/2013/10/assembler-design.pdf
5. Lab N/A - git and ToyASM | CS 2130 - GitHub Pages, accessed June 4, 2025, https://researcher111.github.io/uva-cso1-F23-DG/labs/lab6-git-toyasm.html
6. Assemblers and Compilers | 50.002 CS - natalieagus.github.io, accessed June 4, 2025, https://natalieagus.github.io/50002/notes/assemblersandcompilers
7. Lab 6 - Toy Assembly - CS 2130, accessed June 4, 2025, https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/labs/lab6-toy-assembly.html
8. What is Instruction Set Architecture (ISA)? – Arm®, accessed June 4, 2025, https://www.arm.com/glossary/isa
9. Key Components Of Instruction Set Architecture - FasterCapital, accessed June 4, 2025, https://fastercapital.com/topics/key-components-of-instruction-set-architecture.html
10. www.ece.lsu.edu, accessed June 4, 2025, https://www.ece.lsu.edu/ee4720/2000/lsli03.pdf
11. Computer Organization | Instruction Formats (Zero, One, Two and ..., accessed June 4, 2025, https://www.geeksforgeeks.org/computer-organization-instruction-formats-zero-one-two-three-address-instruction/
12. punctiliousprogrammer.com, accessed June 4, 2025, https://punctiliousprogrammer.com/wp-content/uploads/2017/01/instorg.pdf

13. Designing an Instruction Set Architecture - natalieagus.github.io, accessed June 4, 2025, https://natalieagus.github.io/50002/notes/instructionset
14. About the supported instructions table - IBM, accessed June 4, 2025, https://www.ibm.com/docs/en/hla-and-tf/1.6.0?topic=instructions-about-supported-table
15. Assembler Directives of 8086 Microprocessor | UKEssays.com, accessed June 4, 2025, https://www.ukessays.com/essays/engineering/assembler-directive-of-8086-microprocessor.php
16. Two-pass assemblers, accessed June 4, 2025, http://users.cis.fiu.edu/~downeyt/cop3402/two-pass.htm
17. Understanding Two-Pass Assemblers | Cratecode, accessed June 4, 2025, https://cratecode.com/info/two-pass-assemblers
18. First pass - IBM, accessed June 4, 2025, https://www.ibm.com/docs/ro/ssw_aix_72/assembler/idalangref_firstpass.html
19. Understanding Assembler Passes, accessed June 4, 2025, https://www.infania.net/misc/rs6000-tl1/manuals/adoclib/aixassem/alangref/assemblg.html
20. Phases of a Compiler | GeeksforGeeks, accessed June 4, 2025, https://www.geeksforgeeks.org/phases-of-a-compiler/
21. 2 Pass Assembler Algorithm | PDF | Assembly Language | Computer ..., accessed June 4, 2025, https://www.scribd.com/document/443686945/2-Pass-Assembler-Algorithm
22. A Brief Study on How a Simple Program Get Executed on Computer Hardware by Using Different System Program - IJIRT, accessed June 4, 2025, https://ijirt.org/publishedpaper/IJIRT148652_PAPER.pdf
23. Assembler Directives - Collegeek, accessed June 4, 2025, https://collegeek.com/8085_microprocessor/assemblerdirectives.html
24. Data Defining Directives - TheJat.in, accessed June 4, 2025, https://www.thejat.in/learn/data-defining-directives
25. Design of 2-Pass Assemblers | PDF - Scribd, accessed June 4, 2025, https://www.scribd.com/doc/281188219/Assembler
26. 80386 Programmer's Reference Manual -- Section 2.5, accessed June 4, 2025, https://www.scs.stanford.edu/05au-cs240c/lab/i386/s02_05.htm
27. NASM Manual, accessed June 4, 2025, https://www.ele.uva.es/~jesman/BigSeti/seti1/nasm/nasmdoc2.html
28. Linux X86 Assembly - How To Make Payload Extraction Easier - Secure Ideas, accessed June 4, 2025, https://www.secureideas.com/blog/2021/07/linux-x86-assembly-how-to-make-payload-extraction-easier.html
29. About Assemblers - Assembler Algorithm and Data Structures PDF - Scribd, accessed June 4, 2025, https://www.scribd.com/document/309342172/ABOUT-ASSEMBLERS-ASSEMBLER

-ALGORITHM-AND-DATA-STRUCTURES-pdf
30. Machine Code: Encoding Operand Arguments - UAF CS, accessed June 4, 2025, https://www.cs.uaf.edu/2009/spring/cs641/lecture/01_29_arguments.html
31. ORG Assembler Statement - User's Guides for Keil C51 Development Tools, accessed June 4, 2025, https://developer.arm.com/documentation/101655/latest/Ax51-User-s-Guide/Control-Statements/Reference/ORG-Assembler-Statement
32. PROC Assembler Statement - User's Guides for Keil C51 Development Tools, accessed June 4, 2025, https://developer.arm.com/documentation/101655/latest/Ax51-User-s-Guide/Control-Statements/Reference/PROC-Assembler-Statement
33. Intel HEX - Wikipedia, accessed June 4, 2025, https://en.wikipedia.org/wiki/Intel_HEX