Specification Document – Assembler, Linker, and Loader (RISC-V)

Project: Pipeline Processor Design
Module: Assembler, Linker, and Loader
Team Members: Shankhesh, Preet, Lavkush

---

1. Role in the Project

The *Assembler, Linker, and Loader* module is responsible for translating **RISC-V assembly programs** into executable machine code, resolving symbol references, and loading the program into memory for execution on the **pipeline processor**.
It interacts with:

- **Programming Language Design & Compiler Design modules →** Receives generated RISC-V assembly code.
- **Virtual Machine** → Used for simulation and verification.
- **Operating System** → Provides loader support.
- **System Integration** → Combines all modules for a functional pipeline processor.

---

2. Assembler Specification (RISC-V)

2.1 Input & Output

- **Input:**
    - File format: .s or .asm (RISC-V assembly).
    - Contains comments (#), labels, mnemonics, directives, and constants.

- **Output:**

- Object file: .o (ELF format for RISC-V).

2.2 Assembly Language Program File Format

```None
# Example: Add two numbers
.data
num1: .word 5
num2: .word 10

.text
.globl main
main:
    lw x5, num1
    lw x6, num2
    add x7, x5, x6
    sw x7, result
    ecall
```

2.3 Instruction Mnemonics (RISC-V Subset Supported)

- **Arithmetic:** add, sub, mul, div, addi
- **Logical:** and, or, xor, sll, srl
- **Data Movement:** lw, sw, li, mv
- **Branching:** beq, bne, blt, bge, jal, jalr
- **System:** ecall, ebreak

2.4 Directives Supported

- **.**data, .text, .globl, .word, .byte, .asciz, .align, .space, .end

---

2.5 Label Specification

- Must start with a letter or _.
- Case-sensitive (main ≠ Main).
- Terminated with a colon (:).
- Cannot use reserved keywords.

2.6 Machine Language Specification

- Target: **RISC-V 32-bit (RV32I)** base ISA.
- Instruction size: **32 bits fixed**.
- Encoding formats: R-type, I-type, S-type, B-type, U-type, J-type.

2.7 Assembler Stages

1. **Lexical Analysis** – Tokenize mnemonics, labels, operands.
2. **Syntax Analysis** – Validate against RISC-V grammar.
3. **Semantic Checks** – Validate register usage, immediate ranges.
4. **Symbol Table Generation** – Map labels to addresses.
5. **Relocation Information** – For linker to patch addresses.
6. **Machine Code Generation** – Encode instructions into binary.

---

3. Linker Specification (RISC-V)

3.1 Input & Output

- **Input:** Multiple .o files (ELF).
- **Output:** Single executable (a.out or custom ELF executable).

3.2 Object File Format

- ELF (Executable and Linkable Format) for RISC-V.
- Sections: .text, .data, .bss, .symtab, .rel.text, .rel.data.

3.3 Symbol Management

- External symbols resolved across object files.
- Global and local symbol handling.
- Support for **weak symbols**.

3.4 Relocation

- RISC-V relocation types (e.g., R_RISCV_32, R_RISCV_JAL, R_RISCV_BRANCH).
- Adjust addresses based on final memory layout.

3.5 Libraries

- Static libraries (.a) supported.
- Search order: local path → standard RISC-V lib path.

---

4. Loader Specification (RISC-V)

4.1 Loader Responsibilities

- Read ELF executable from storage.
- Allocate memory for code, data, and stack.
- Apply relocations if required.
- Set **Program Counter** to _start or main.
- Transfer control to program.

4.2 Loading Methodology

1. Open executable.
2. Parse ELF headers.
3. Load segments into memory.
4. Initialize stack and heap.
5. Jump to the entry point.

---

5. Development & Execution Flow

1. **Compiler (Other module)** → Generates RISC-V .s file.
2. **Assembler (Our module)** → Converts .s → .o.
3. **Linker (Our module)** → Links .o files → ELF executable.
4. **Loader (Our module)** → Loads ELF into memory.
5. **Pipeline Processor** → Executes instructions.

---

6. Alternative Plan

If not executable on the physical pipeline processor:

- Use **RISC-V Spike simulator** or **QEMU RISC-V** for functional testing.
- Integrate with the Virtual Machine module for debugging and verification.

Plan Week Wise:

- **Implementation Language:**C++ (with support tools like **Lex & Yacc / Flex & Bison** for parsing)

Module	Title	Start	End	Deliverables
1	Instruction Set & Specification Design	Week 1	Week 2	ISA design doc, instruction formats, opcodes
2	Assembler (Phase 1 – Basic)	Week 2	Week 4	C++ assembler supporting labels, literals, bytecode output
3	Assembler (Phase 2 – Advanced)	Week 4	Week 5	Macro processing, conditional assembly, .vmo object file output
4	Linker	Week 5	Week 6	Merging multiple .vmo files, symbol resolution, relocation handling
5	Loader	Week 6	Week 7	Loading .vmc executable into VM memory
6	Virtual Machine Execution Core	Week 7	Week 8	Stack, registers, memory, instruction execution
7	Toolchain Integration	Week 8	Week 9	Cross-compiler setup, testing pipeline, automation
8	Testing, Debugging & Documentation	Week 9	Week 10	Test cases, performance evaluation, final report

Technical Details per Module

🔹 Module 1 – Instruction Set & Specification:

- Define **instruction formats** (e.g., R-type, I-type, J-type)
- Assign **opcodes** and register encodings

- Deliverable: **ISA Specification Document**
- Example instruction format:

```None
ADD R1, R2, R3
Binary → [opcode=0001][dst=01][src1=10][src2=11]
```

🔹 **Module 2 – Assembler (Phase 1 – Basic)**

- Implement **Assembler in C++**
- **Functions:**
  - Lexical analysis (tokenize mnemonics, labels, registers)
  - Syntax parsing (using Flex/Bison or manual parser)
  - Symbol table creation (first pass)
  - Machine code generation (second pass)
- Output: .vmc (binary) or .vmo (object file)

🔹 **Module 3 – Assembler (Phase 2 – Advanced)**

- Add **directives**: .data, .text, .global
- Add **macro processor**
- Add **relocation table generation**
- Output object file .vmo with:
  - Code section
  - Symbol table
  - Relocation table

🔹 **Module 4 – Linker**

- Merge multiple .vmo object files

- Functions:
    - Create **global symbol table**
    - Apply relocations
    - Detect **duplicate / undefined symbols**
- Output: single .vmc executable

🔹 **Module 5 – Loader**

- Read .vmc format
- Load into **virtual memory**
- Set **program counter (PC)** at entry point
- Provide **debugging hooks** (dump memory, trace execution)

🔹 **Module 6 – Virtual Machine Execution Core**

- Stack-based execution or Register-based (based on ISA choice)
- Components:
    - Registers (R0..Rn, PC, SP, FLAGS)
    - Instruction Decoder
    - Execution Unit
- Support for:
    - Arithmetic
    - Memory access
    - Control flow
    - I/O (basic)

🔹 **Module 7 – Toolchain Integration**

- Cross-compiler setup:
    - C++ compiler (g++, clang++)
    - Lex/Yacc (or Flex/Bison) for assembler parsing
- Build system setup (Makefile / CMake)

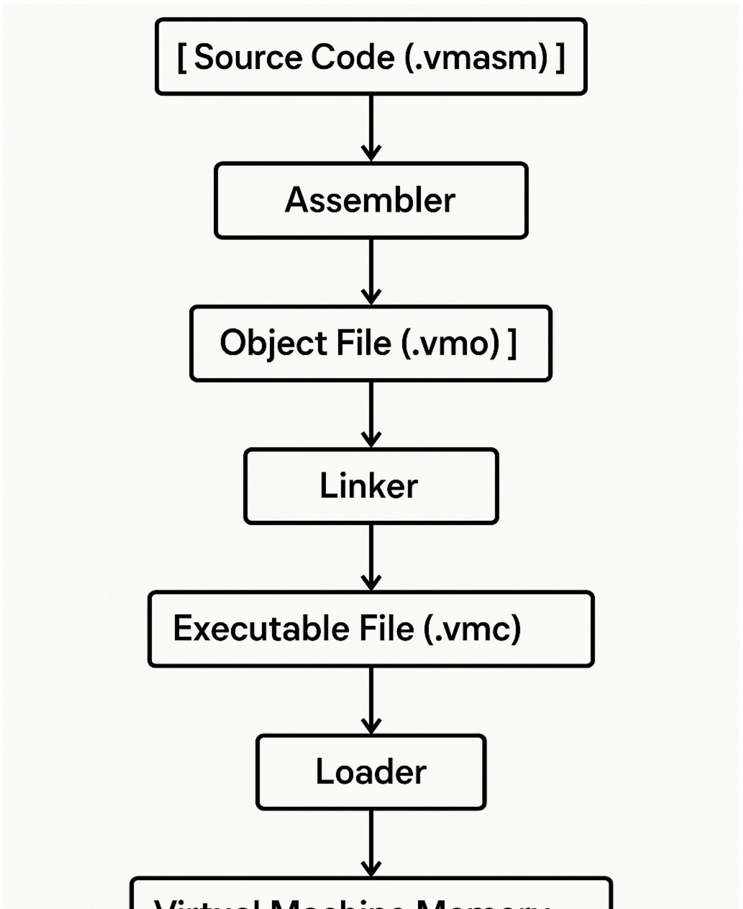
- Optional: integrate with **LLVM IR** for testing

🔹 **Module 8 – Testing, Debugging, Documentation**

- Unit tests for assembler, linker, loader
- Test programs:
    - Arithmetic test
    - Recursive factorial
    - Loop with branching
- Debugging tools: **symbol dump, hex dump**
- Deliverables: Test suite + final project report

Tools Setup

- **Compiler:** GCC / Clang (C++)
- **Parser Tools:** Flex (Lex) + Bison (Yacc)
- **Build Tool:** CMake / Makefile
- **Debugger:** GDB
- **Version Control:** GitHub/GitLab


[Source Code (.vmasm)]
Assembler
Object File (.vmo)]
Linker
Executable File (.vmc)
Loader