
CSD Project: Module 1 Design & Implementation Document

- **Team:** Assembler, Linker, and Loader (MIPS)
- **Module:** 1 - Assembler Design and Prototyping
- **Date:** August 23, 2025

Team Members & Contributions

Roll Number	Team Role	Key Contributions for Module 1
CS22B050	**Person A:** Language Architecture & Validation	• Designed and documented the human readable Stack Assembly Language (.stkasm). • Designed and documented the binary Bytecode (.vm) file format. • Created the test files (valid & invalid) to check the parser's correctness. • Developed the C++ validator program to automate testing.
CS22B015	**Person B:** Bytecode generation & Back-end	• Implemented the core C++ data structures (structures.h) to represent all instructions in memory. • Wrote the "Bytecode Emitter" (emitter.cpp), the core logic that converts instruction objects into their final binary form. • Created script to test and validate the binary file generation.
CS22B021	**Person C:** Parser & Front-end	• Implemented the C++ "Tokenizer" and "Parser" (parser.cpp) to read and understand the .stkasm language. • Wrote the main assembler.cpp program, integrating the parser and emitter to create the complete workflow. • Set up the Makefile and managed the overall C++ project structure.

---

1. Introduction: Our Goal for Module 1

The goal for our team in Module 1 was to design and build a working prototype of a custom **Assembler**. Based on the project's architecture, this assembler's job is to translate a new, human-readable **Stack-Based Assembly Language** into a binary **Bytecode** format that our project's Virtual Machine (VM) can run.

This document details the two file formats we designed, the implementation of our C++ prototype, and a complete example of the assembly process.

---

2. The Human-Readable Assembly Format (.stkasm)

This is the text-based language that the **Compiler Team** will generate and our Assembler will read. We designed it to be simple and clear.

- **Syntax Rules:**

    - One instruction per line.
    - Comments begin with `#`.
    - Labels (for functions or jumps) are a name followed by a colon (e.g., `main:`).
- **Module 1 Instruction Set:**
    - `iconst <value>`: Pushes an integer number onto the stack.
    - `iadd`: Takes the top two integers from the stack, adds them, and pushes the result back.
    - `invoke <label> <num_args>`: Calls a function.
    - `ret`: Returns from the current function.

---

3. The Binary Bytecode Format (`.vm`)

This is the binary output file from our assembler. It's designed to be compact and easy for the **Virtual Machine Team**'s VM to parse and execute.

- **File Structure:** A simple header followed by a list of binary instructions.
- **Data Order:** All numbers are stored in **little-endian** format.

Header (First 8 bytes of the file):

Size	Description	Example Value
4 bytes	Magic Number (to identify the file type)	`0x5354414B` ("STAK")
4 bytes	Number of instructions in the file	`5`

Instruction Opcodes (1 byte each):

Opcode	Instruction
`0x01`	`iconst`
`0x02`	`iadd`
`0x03`	`invoke`
`0x04`	`ret`

Instruction Binary Format:

- **`iconst`**: [opcode (1 byte)] [value (4 bytes)]
- **`iadd`**: [opcode (1 byte)]
- **`invoke`**: [opcode (1 byte)] [function_id (4 bytes)] [num_args (1 byte)]
- **`ret`**: [opcode (1 byte)]

---

4. A Complete Example: `valid_program.stkasm` to `program.vm`

Here is a simple example showing the complete flow of our assembler.

Step 1: The Input File

The assembler starts by reading this simple text file.

valid_program.stkasm (Input)

```
# This program calculates (10 + 20)
invoke main 0

main:
    iconst 10
    iconst 20
    iadd
    ret
```

Step 2: Parsing

The **Parser (Person C)** reads the file. It ignores comments and identifies the main label. It then converts each line of text into a list of instruction "objects" in memory. This list is the internal representation of the program.

Step 3: Bytecode Emission

The **Bytecode Emitter (Person B)** takes the list of instruction objects. It goes through the list one by one and translates each object into its binary form according to the .vm specification. For the invoke main instruction, it looks up "main" in its symbol table to find its address.

Step 4: The Output File

The final stream of binary bytes is written to the output file program.vm. This file is no longer human-readable and must be inspected with a hex editor.

program.vm (Final Binary Output)

```
// This is what the inside of the file looks like, byte-by-byte.
// Address | Hexadecimal Bytes                              | Description
//-----------------------------------------------------------------------------
-------
// --- Header (8 bytes) ---
00000000 | 4B 41 54 53 05 00 00 00                        | "STAK", 5 instructions
// --- Code Section (16 bytes) ---
00000008 | 03 01 00 00 00 00                              | invoke main (address
1), 0 args
0000000E | 01 0A 00 00 00                                 | iconst 10
00000013 | 01 14 00 00 00                                 | iconst 20
00000018 | 02                                             | iadd
00000019 | 04                                             | ret
```

---

5. Conclusion & Next Steps

For Module 1, our team successfully designed the necessary file formats and developed a working C++ prototype of our stack-based assembler. The prototype can correctly parse our new assembly language and generate a valid binary bytecode file.

For future modules, our next steps will be:

- To expand the instruction set in collaboration with the VM and Compiler teams.
- To implement a linker to combine multiple `.vm` files.
- To fully integrate with the Compiler team's output and the VM team's input.