Lab 3: Build your own Assembler
CSC 250 SP 2025

Introduction:

An **assembler** is a program that translates **assembly language** into **machine code** (binary instructions) that a CPU can execute.

Assembly language uses human-readable mnemonics, while **machine code** is made of numbers (binary or hexadecimal).

Assembly Instruction	Machine Code (Hex)	Binary Equivalent
LOAD A, 5	1005	0001 0000 0000 0101
ADD A, B	3000	0011 0000 0000 0000
JUMP LOOP	5003	0101 0000 0000 0011

Steps:

1. Down load Java Assembler/CPU from Moodle
    a. There are two files
        i. IntelliJ project – Windows or Mac
        ii. Java Source - PI
    b. Download your preference

    Note: The program runs in an infinite loop

Program overview:

1. static {} (Static Block)

- **Purpose:** Initializes the OPCODES map with machine opcodes corresponding to assembly instructions. This map helps in translating assembly instructions (like LOAD, STORE, etc.) into their machine code equivalents.
- **Details**: It adds opcode mappings for commands such as LOAD, STORE, ADD, SUB, JUMP, each mapped to a unique string (e.g., "1" for LOAD)**.**

---

2. passOne(String[] lines)

- **Purpose:** Parses the assembly code and builds the symbol table and sets up the initial memory.
- **Details:**

- It processes each line in the given assembly program. If the line defines a label (e.g., LOOP:), it records the memory address of the label in the symbolTable.
- If a variable (VAR) is declared, it stores the variable name and its memory address in the symbolTable and initializes the memory with its value.
- For general instructions, the code adds the line to the assemblyCode list and increments the memoryAddress to keep track of where the instruction should be placed in memory.

---

3. passTwo()

- **Purpose:** Converts the assembly code into machine code (hexadecimal format) and stores it in the machineCode list.
- **Details:**
  - For each instruction in the assemblyCode list, it retrieves the corresponding opcode from the OPCODES map.
  - The operand (usually a label or a number) is checked:
    - If it’s a label, the symbolTable is consulted to get the address (converted to hex).
    - If it's a number, it's directly converted to hex.
    - If the operand is missing or invalid, it defaults to "00".
  - Each instruction is then combined into a machine code format (opcode + operand) and added to the machineCode list.

---

4. hexToBinary(String hex)

- **Purpose:** Converts a hexadecimal string into its binary representation.
- **Details:**
  - The method takes a hex string (e.g., "1A"), converts it to a decimal integer, and then formats it as a 12-bit binary string (padded with zeros if needed).
  - This helps in visualizing the machine code in binary form, useful for low-level debugging.

---

5. executeMachineCode()

- **Purpose:** Executes the machine code instructions stored in the machineCode list.
- **Details:**
    - It uses the programCounter to iterate through the machineCode list.
    - For each instruction, the method extracts the opcode and operand, and then performs the corresponding operation:
        - LOAD (opcode "1") loads data from memory to register A.
        - STORE (opcode "2") stores data from register A to memory.
        - ADD (opcode "3") adds the value of register B to register A.
        - SUB (opcode "4") subtracts the value of register B from register A.
        - JUMP (opcode "5") modifies the programCounter to jump to a new address, effectively creating loops.
        - HALT (opcode "6") stops execution.
    - The program continues execution until the HALT instruction is encountered, or if it reaches the end of the machineCode list.

---

6. main(String[] args)

- **Purpose:** The entry point of the program, where the assembler is run.
- **Details:**
    - It defines a sample assembly program that includes variable declarations, instructions, and a loop.
    - It calls passOne() to parse the program and fill the symbolTable and memory.
    - Then, it calls passTwo() to convert the assembly code into machine code and store it in the machineCode list.
    - The program prints the symbolTable (showing labels and their memory addresses) and the generated machine code (in both hex and binary format).
    - Finally, it executes the machine code using executeMachineCode() and prints the execution steps, including memory and register updates.

Lab 3: Build your own Assembler
CSC 250 SP 2025

Summary of Key Components:

- **OPCODES Map:** Stores opcodes for assembly instructions and their corresponding machine code values**.**
- **symbolTable Map:** Stores labels and variable names with their memory addresses.
- **assemblyCode List:** Holds the raw assembly instructions after parsing.
- **machineCode List:** Holds the converted machine code in hexadecimal format.
- **memory Array**: A simulated memory storage for variables and program data.
- **Registers (regA, regB):** Used to store intermediate results during execution**.**

Lab Requirements

1. Implement HALT instruction to stop infinite loops

2. Add support more registers (C, D)

3. Add additional instruction set (MUL, DIV, AND, OR)

4. Handle Immediate Values (LOAD A, #5)

Answer the following questions

1. Explain the role of passOne and passTwo methods in the assembler simulation. How do they contribute to the assembly and execution process?

2. What is the purpose of the symbolTable in this program, and how is it used during the execution of machine code?

3. What happens when the HALT instruction is encountered in the executeMachineCode method? Describe the control flow at this point.

4. In the executeMachineCode method, how does the program handle the JUMP instruction? How does the program counter change?

5. How does the program convert an operand (like a variable or label) into machine code?

6. What would happen if an unknown instruction or opcode is encountered in the passTwo method? How does the program handle this?

7. What happens when the STORE opcode is executed? Describe what happens to the value in regA and how it affects memory.

8. What would happen if the memoryAddress was not properly incremented in the passOne method? How would that affect the simulation of memory and the final machine code output?

9. In a real CPU, what would be the role of the control unit in executing the machine code generated by this assembler simulation?