Project6 for Intro to Computer Systems 2025 spring

Yiming Cheng

12450588

Before All

Note1 The function of `proj0` is totally integrated into `Proj6` which means you dont need to use `proj0` to delete comments anymore.This project will first delete comments and blanks and then do the assembling job.Also,after discussing with Prof Billingsley,although efficiency is not required,this project has used `deque` to boost search efficiency,making it even faster than the online IDE

Note2 Tests of following asm files has completely passed,

`MaxL.asm` `RectL.asm` `PongL.asm`

`Add.asm` `Max.asm` `Rect.asm` `Pong.asm`

`Mult.asm` `Fill.asm`

You can use

```
python \path1\to\Assmebler.py \path2\to\Input.asm
```

to generate `\path2\to\Input.hack`

Or just use the pre-assembled `.hack` file of the above asm files in `\data` file

Note3 Thank you for reading this file

Jump to `Quick Setup` for details of environment

Jump to `Class Structure` for the structure of `Assembler` class and its subclass

Jump to `XXX.py` for sub-modules' instruction of `XXX.py` and their seperate tests

Jump to `Assembler.py`\ `tests` for tests details of all test asm files

Again appreciate for reading.

File Structure

```
.
├── README.pdf
├── data
│   ├── Add.asm
│   ├── Add.hack
│   ├── Fill.asm
│   ├── Fill.hack
│   ├── Max.asm
```

```
│   ├── Max.hack
│   ├── MaxL.asm
│   ├── MaxL.hack
│   ├── Mult.asm
│   ├── Mult.hack
│   ├── Pong.asm
│   ├── Pong.hack
│   ├── PongL.asm
│   ├── PongL.hack
│   ├── Rect.asm
│   ├── Rect.hack
│   ├── RectL.asm
│   └── RectL.hack
└── src
    ├── Assembler.py
    ├── Coding.py
    ├── Lexer.py
    ├── Parser.py
    └── SymbolTable.py

3 directories, 24 files
```

Quick Setup

Prerequisites

`Python 3.x`: The assembler is written in Python and requires `Python 3` or higher to run.

Required Libraries

The following Python libraries are used in this project:

`re`(regular expressions)

`collections` (for deque data structure)

`sys` (for command-line arguments)

`os` (for file and directory operations)

These libraries are built-in to Python, so no additional installation is required.

OS

Strongly recommend `Linux` or `MacOS`

`HarmonyOS` version is on its way

`Windows` will never appear

Usage

1. **Navigate to the Project Folder**:

```
cd /path/to/Folder
```

2. **Prepare Your Input File**:

   e.g `\path\to\input.asm`

3. **Run the Assembler**:

```
python Assembler.py input.asm
```

4. **output**

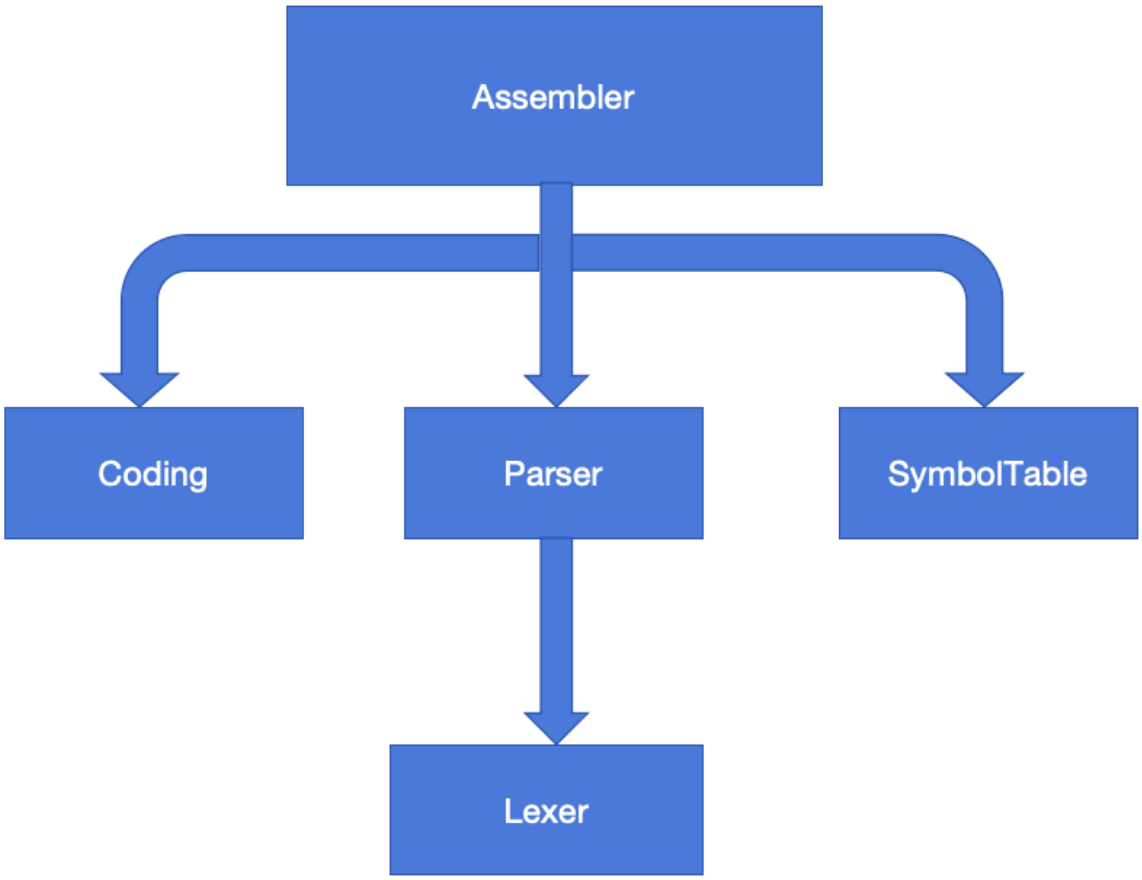
You will get `\path\to\input.hack`

protocal

This project's author is Eamin,current a predoc student in UChicago,Copyright preserved

This project is under `WTFPL` license

Class Structure



`SymbolTable.py`

Module Introduction

The `SymbolTable` class is a core module designed to manage a symbol table that maps symbols (variable names to memory locations). This class provides methods to add, delete, retrieve, and check for the existence of symbols within the table.

The `SymbolTable` class is essential for managing symbols during the compilation process. It ensures that the compiler can track variable names, labels, and their respective memory locations. This allows the compiler to resolve addresses and handle memory management effectively.

Features

- **`__init__()`**: The table is initialized with predefined symbols such as `SP`, `LCL`, `R0`, `SCREEN`, and `KBD` along with their corresponding memory addresses.

- **`add_symbol(symbol_name,symbol_position)`**: Adds a new symbol with a given memory address to the table.

- **`del_symbol(symbol_name,symbol_position)`**: Deletes a symbol either by its name or its memory address.

- **`get_symbol(symbol_name)`**: Retrieves the memory address of a symbol.

- **`contain_symbol(symbol_name)`**: Checks if a symbol exists in the table.

- `print_symboltable(self)` Prints the entire symbol table.

Running Environment

No extra out package needed

to run tests for SymbolTable.py

```
python SymbolTable.py
```

Tests

The script provides following tests:

```
if __name__ == "__main__":

    print("Running tests for SymbolTable...\n")

    st = SymbolTable()

    # Test printing the symbol table
    st.print_symboltable()

    # Test adding a new symbol
    st.add_symbol("TEST", 999)
    assert st.get_symbol("TEST") == 999
    print("✅ Add symbol test passed.")

    # Test deleting a symbol by name
    st.del_symbol(symbol_name="TEST")
    assert not st.contain_symbol("TEST")
    print("✅ Delete by name test passed.")
```

```
    # Test deleting a symbol by position
    st.del_symbol(symbol_position=4)
    assert not st.contain_symbol("THAT")
    print("✅ Delete by position test passed.")
    print("✅ Contain symbol test passed.")
    print("✅ Print test passed.")

    print("\nAll tests passed!")
```

test result:

```
Running tests for SymbolTable...

{'SP': 0, 'LCL': 1, 'ARG': 2, 'THIS': 3, 'THAT': 4, 'R0': 0, 'R1': 1, 'R2': 2, 'R3': 3,
'R4': 4, 'R5': 5, 'R6': 6, 'R7': 7, 'R8': 8, 'R9': 9, 'R10': 10, 'R11': 11, 'R12': 12,
R13': 13, 'R14': 14, 'R15': 15, 'SCREEN': 16384, 'KBD': 24576}

✅ Add symbol test passed.
✅ Delete by name test passed.
✅ Delete by position test passed.
✅ Contain symbol test passed.
✅ Print test passed.

All tests passed!
```

`Lexer.py`

Module Introduction

The `Lexer` class is a core module designed to perform lexical analysis on assembly-like language code. It reads input source code, tokenizes it into meaningful symbols, and categorizes these symbols into different token types. This module is essential for breaking down source code into tokens, which will be used in later stages of the compilation process.

The `Lexer` ensures that the compiler can correctly identify and process variables, numbers, operators, and comments while maintaining an organized stream of tokens for parsing.

Features

- **`__init__(file_name)`**: Initializes the lexer by reading a file and tokenizing its contents.
- **`has_more_commands()`**: Checks if there are more commands (non-empty token queue) to process.
- **`next_command()`**: Retrieves the next command, which consists of a sequence of tokens.
- **`has_more_tokens()`**: Checks if there are more tokens in the current command.
- **`next_token()`**: Fetches the next token from the current command.
- **`peek_token()`**: Returns the next token without consuming it.
- **`tokenize(lines)`**: Processes a list of code lines into categorized token sequences.
- **`remove_comment(line)`**: Removes comments from a line.
- **`classify_token(word)`**: Determines the type of a token (number, identifier, or operator).
- **`is_NUM(word)`**: Checks if a word is a valid number.
- **`is_OP(word)`**: Checks if a word is a valid operator.
- **`is_ID(word)`**: Checks if a word is a valid identifier.

Running Environment

No external dependencies are required.

To run tests for `Lexer.py`:

```
python Lexer.py
```

Tests

The script provides the following tests:

```
if __name__ == "__main__":
    # Create a temporary test file for testing
    test_code = """
    // This is a comment
    LOOP = 10;
    ADD R1, R2, R3
    """
    with open('test.asm', 'w') as file:
        file.write(test_code)

    # Test Lexer Initialization and Tokenization
    lexer = Lexer('test.asm')
    print("Test 1: Lexer Initialization")
    assert lexer.has_more_commands(), "Lexer should have
commands to process."
    command = lexer.next_command()
    assert len(command) > 0, "Command should contain
tokens."
    print("✅ Lexer Initialization Test Passed")

    # Test Tokenize Function
    print("Test 2: Tokenize Function")
    lexer = Lexer('test.asm')
    lexer.next_command()
```

```
    tokens = lexer.current_tokens
    assert (ID, "LOOP") in tokens, "LOOP should be
classified as an ID."
    assert (OP, "=") in tokens, "Equals sign should be
classified as an OP."
    assert (NUM, "10") in tokens, "10 should be classified
as a NUM."
    assert (OP, ";") in tokens, "Semicolon should be
classified as an OP."
    print("✅ Tokenize Function Test Passed")

    # Test Remove Comment Function
    print("Test 3: Remove Comment Function")
    lexer = Lexer('test.asm')
    line = "LOOP = 10; // This is a comment"
    cleaned_line = lexer.remove_comment(line)
    assert cleaned_line == "LOOP = 10;", "Comment removal
failed."
    print("✅ Remove Comment Function Test Passed")

    # Test Classify Token Function
    print("Test 4: Classify Token Function")
    lexer = Lexer('test.asm')
    assert lexer.classify_token("LOOP") == (ID, "LOOP"),
"LOOP should be classified as ID."
    assert lexer.classify_token("=") == (OP, "="), "Equals
sign should be classified as OP."
    assert lexer.classify_token("10") == (NUM, "10"), "10
should be classified as NUM."
    assert lexer.classify_token(";") == (OP, ";"),
"Semicolon should be classified as OP."
    print("✅ Classify Token Function Test Passed")
```

```
    # Clean up the temporary test file
    import os
    os.remove('test.asm')

    print("\nAll tests passed!")
```

test result：

```
(base) eamin@EamindeMacBook-Pro src % python Lexer.py
Test 1: Lexer Initialization
✅ Lexer Initialization Test Passed
Test 2: Tokenize Function
✅ Tokenize Function Test Passed
Test 3: Remove Comment Function
✅ Remove Comment Function Test Passed
Test 4: Classify Token Function
✅ Classify Token Function Test Passed

All tests passed!
```

`Parser.py`

Module Introduction

The `Parser` module is responsible for processing assembly-like commands that have been tokenized by the `Lexer`. It classifies each command into three types (`A_COMMAND`, `C_COMMAND`, `L_COMMAND`), extracts relevant symbols, and parses the components of C-commands (`dest`, `comp`, and `jump`). This module plays a crucial role in translating human-readable assembly code into machine-understandable binary instructions.

Features

- **`__init__(file)`**: Initializes the parser by linking it with a `Lexer` instance and setting up internal command information.
- **`has_more_commands()`**: Checks if there are more commands left in the input file.
- **`advance()`**: Moves to the next command and determines its type.
- **`command_type()`**: Identifies the type of the current command (`A_COMMAND`, `C_COMMAND`, or `L_COMMAND`).
- **`symbol()`**: Extracts the symbol or address from `A_COMMAND` and `L_COMMAND`.
- **`dest()`**: Retrieves the destination field from a `C_COMMAND`.
- **`comp()`**: Retrieves the computation field from a `C_COMMAND`.
- **`jump()`**: Retrieves the jump field from a `C_COMMAND`.

Running Environment

No external dependencies are required beyond `Lexer.py`.

To run tests for `Parser.py`:

```
python Parser.py
```

Tests

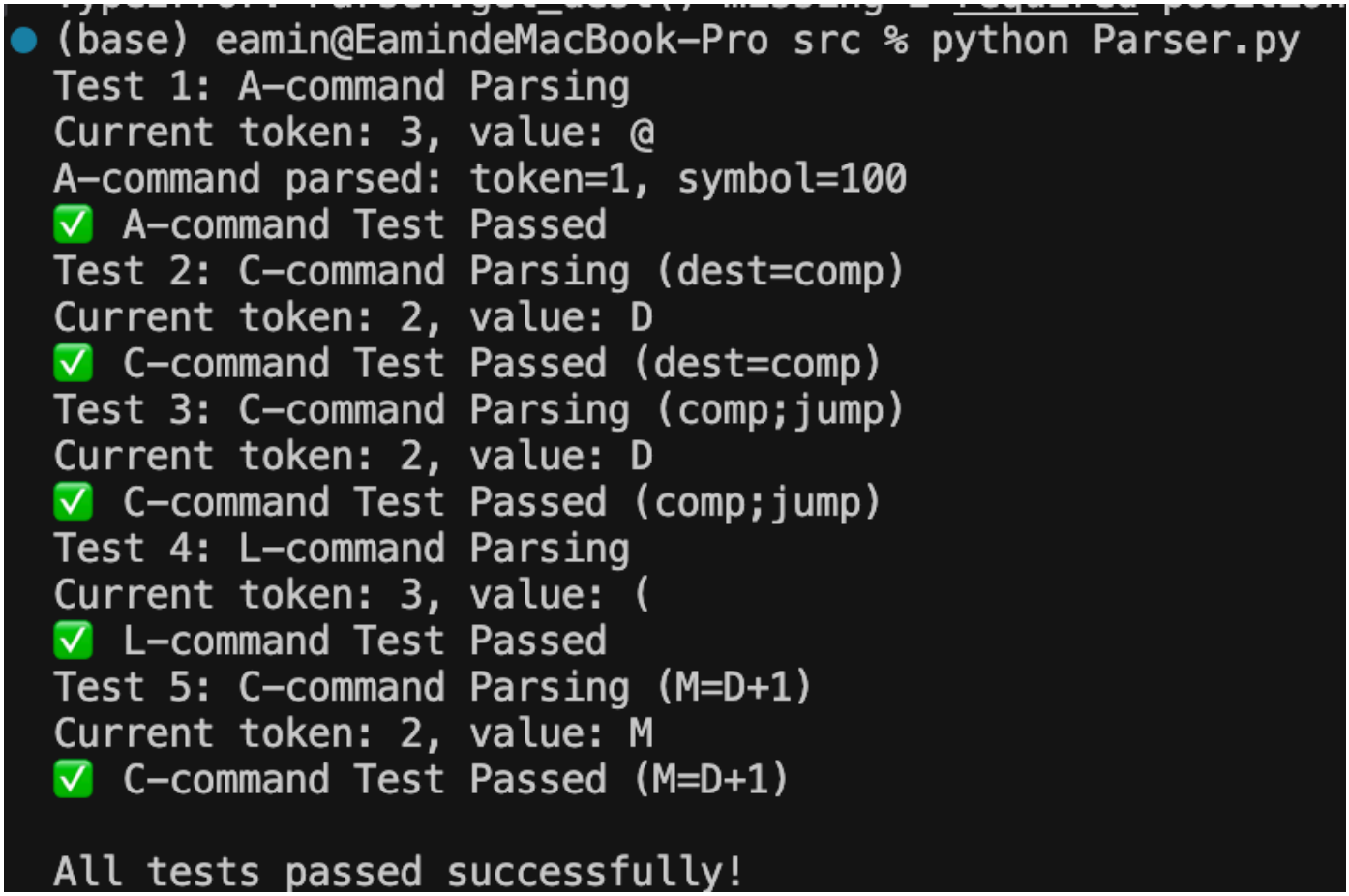
The script includes the following tests:

```
if __name__ == "__main__":

 # Create a temporary test file
```

```
 test_code = """
@100
D=A
0;JMP
(LOOP)
"""
with open('test.asm', 'w') as file:
    file.write(test_code)
# Initialize parser
parser = Parser('test.asm')

# Test A-command Parsing
print("Test 1: A-command Parsing")
parser.advance()
assert parser.command_type() == Parser.A_COMMAND,
"Expected A_COMMAND"
assert parser.symbol() == "100", "Expected symbol 100"
print("✅ A-command Test Passed")

# Test C-command Parsing
print("Test 2: C-command Parsing")
parser.advance()
assert parser.command_type() == Parser.C_COMMAND,
"Expected C_COMMAND"
assert parser.dest() == "D", "Expected dest D"
assert parser.comp() == "A", "Expected comp A"
print("✅ C-command Test Passed")

# Test Jump Parsing
print("Test 3: Jump Parsing")
parser.advance()
```

```
assert parser.command_type() == Parser.C_COMMAND,
"Expected C_COMMAND"
assert parser.comp() == "0", "Expected comp 0"
assert parser.jump() == "JMP", "Expected jump JMP"
print("✅ Jump Test Passed")

# Test L-command Parsing
print("Test 4: L-command Parsing")
parser.advance()
assert parser.command_type() == Parser.L_COMMAND,
"Expected L_COMMAND"
assert parser.symbol() == "LOOP", "Expected symbol LOOP"
print("✅ L-command Test Passed")

# Cleanup
import os
os.remove('test.asm')

print("\nAll tests passed!")
```

Test Results



`Coding.py`

Module Introduction

The `Coding` module is responsible for generating binary instructions for the Hack assembly language. It includes methods for translating A-commands and C-commands into binary format. The A-command includes a 15-bit address, while the C-command involves the computation (`comp`), destination (`dest`), and jump (`jump`) fields, which are encoded in binary. The `Coding` module allows for easy conversion of assembly language instructions into machine-readable binary code.

Features

- **`__init__()`**: Initializes the `Coding` instance. This method is currently empty but sets up the necessary structure for further command generation.
- **`generateA(addr)`**: Generates the binary representation of an A-instruction (used for variables or memory addresses). It prepends a '0' to the 15-bit address.
- **`generateC(dest, comp, jump)`**: Generates the binary representation of a C-instruction (used for computations, memory accesses, and jumps). It combines binary encodings of the `comp`, `dest`, and `jump` fields.
- **`getDest(d)`**: Converts the destination part (`dest`) of a C-instruction into its binary representation based on predefined encoding.
- **`getComp(c)`**: Converts the computation part (`comp`) of a C-instruction into its binary representation based on predefined encoding.
- **`getJump(j)`**: Converts the jump part (`jump`) of a C-instruction into its binary representation based on predefined encoding.
- **`convertToBits(n, length=0)`**: Converts a number to a binary string of a specified length. It ensures zero-padding if a length is provided.

Running Environment

This module does not have any external dependencies and can be executed as a standalone Python script.

To run tests for `Coding.py`:

```
python Coding.py
```

Tests

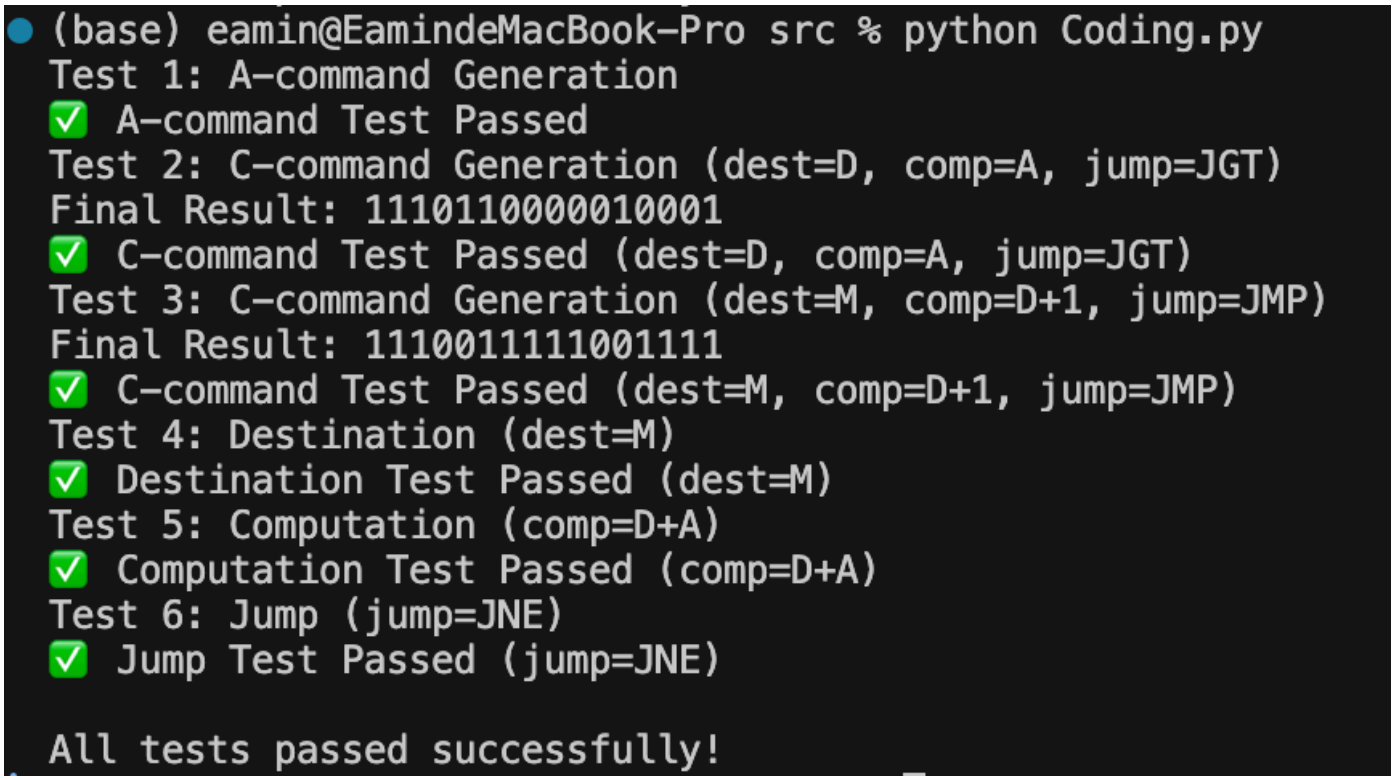
The script includes the following tests for generating A-commands and C-commands:

```
if __name__ == "__main__":

# Initialize the Coding class for testing

code = Coding()
# Test 1: A-command generation (Address: 100)
print("Test 1: A-command Generation")
result = code.generateA(100)
expected = '0000000001100100'  # 100 in 15-bit binary with
leading 0
assert result == expected, f"Expected {expected}, but got
{result}"
print("✅ A-command Test Passed")

# Test 2: C-command generation (dest=D, comp=A, jump=JGT)
print("Test 2: C-command Generation (dest=D, comp=A,
jump=JGT)")
result = code.generateC("D", "A", "JGT")
expected = '1110110000010001'  # Corrected expected value
assert result == expected, f"Expected {expected}, but got
{result}"
```

```
print("✅ C-command Test Passed (dest=D, comp=A,
jump=JGT)")

# Test 3: C-command generation (dest=M, comp=D+1,
jump=JMP)
print("Test 3: C-command Generation (dest=M, comp=D+1,
jump=JMP)")
result = code.generateC("M", "D+1", "JMP")
expected = '1110011111001111'  # Corrected expected C-
instruction (16 bits)
assert result == expected, f"Expected {expected}, but got
{result}"
print("✅ C-command Test Passed (dest=M, comp=D+1,
jump=JMP)")

# Test 4: Destination part of C-command (dest=M)
print("Test 4: Destination (dest=M)")
result = code.getDest("M")
expected = '001'  # 'M' maps to '001' in binary
assert result == expected, f"Expected {expected}, but got
{result}"
print("✅ Destination Test Passed (dest=M)")

# Test 5: Computation part of C-command (comp=D+A)
print("Test 5: Computation (comp=D+A)")
result = code.getComp("D+A")
expected = '0000010'  # 'D+A' maps to '0000010' in binary
assert result == expected, f"Expected {expected}, but got
{result}"
print("✅ Computation Test Passed (comp=D+A)")

# Test 6: Jump part of C-command (jump=JNE)
print("Test 6: Jump (jump=JNE)")
```

```
result = code.getJump("JNE")
expected = '101'  # 'JNE' maps to '011' in binary
assert result == expected, f"Expected {expected}, but got
{result}"
print("✅ Jump Test Passed (jump=JNE)")

print("\nAll tests passed successfully!")
```

test result



`Assembler.py`

Module Introduction

The `Assembler` module is responsible for converting Hack assembly language code into Hack machine language (binary code). It processes a `.asm` file, performs two passes over the instructions, and outputs a `.hack` file. The module includes logic for handling A-commands, C-commands, and L-commands, as well as managing a symbol table to handle labels and variables.

Features

- **`__init__()`**: Initializes the assembler with an empty symbol table and sets the starting address for variables to 16.
- **`pass0(file)`**: First pass of the assembler. It scans the assembly file and identifies labels (L-commands). It adds these labels to the symbol table with a unique address.
- **`pass1(infile, outfile)`**: Second pass of the assembler. It generates machine code for each command, using the symbol table for label resolution and variable addresses, then writes the results to the output `.hack` file.
- **`_get_address(symbol)`**: Looks up the address for a symbol, either from the symbol table or by assigning it a new address if it's a new variable.
- **`_outfile(infile)`**: Determines the correct output file path by replacing the `.asm` extension with `.hack`. If the output file already exists, it will be overwritten.
- **`assemble(file)`**: Main method that runs both passes of the assembler and generates the final `.hack` output file.

Running Environment

This module does not have any external dependencies and can be executed as a standalone Python script.

To run the assembler on a `.asm` file, use the following command:

```
python Assembler.py path/to/input.asm
```

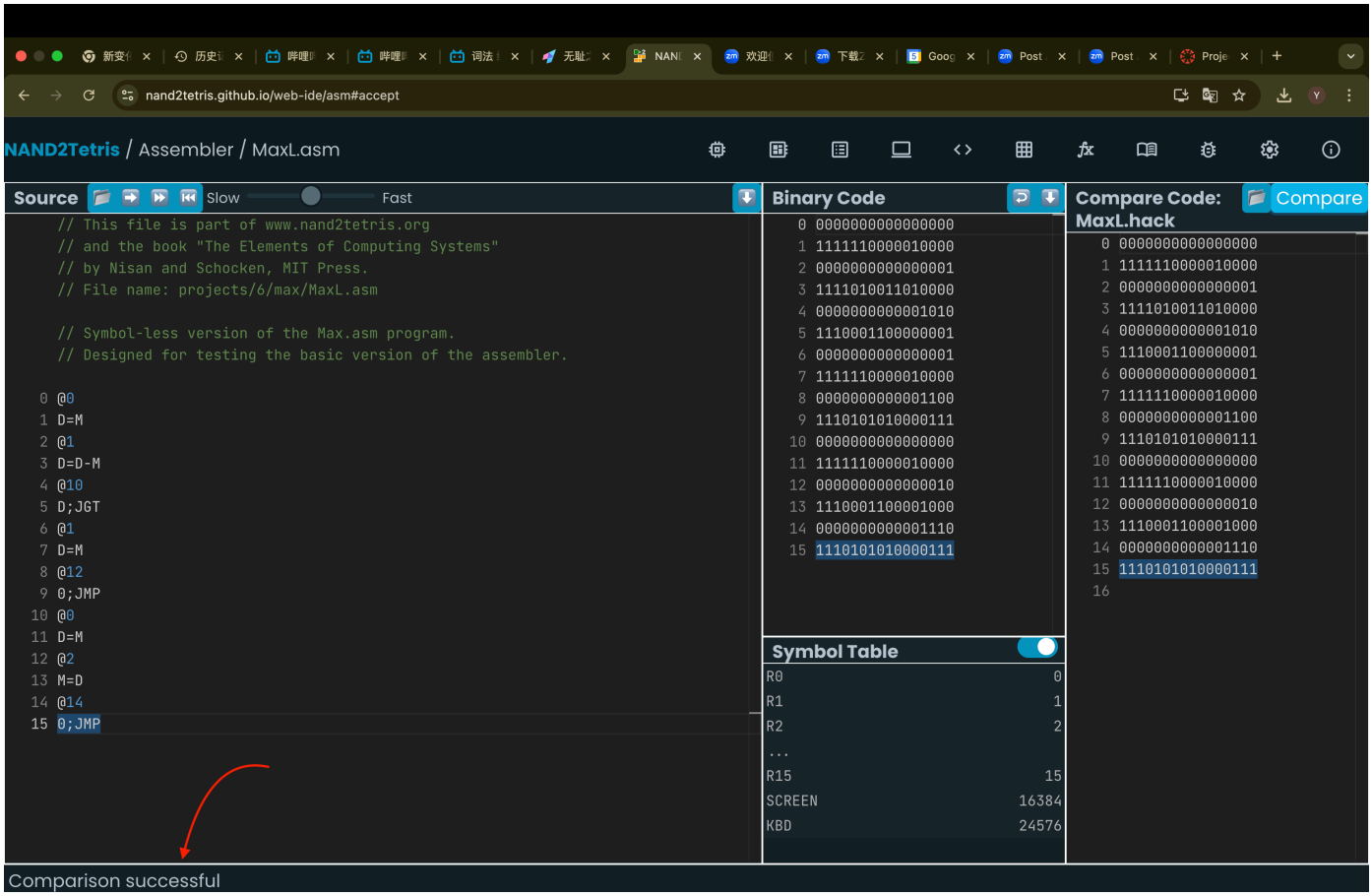
Tests：

Running all the files in the `data\input` file with

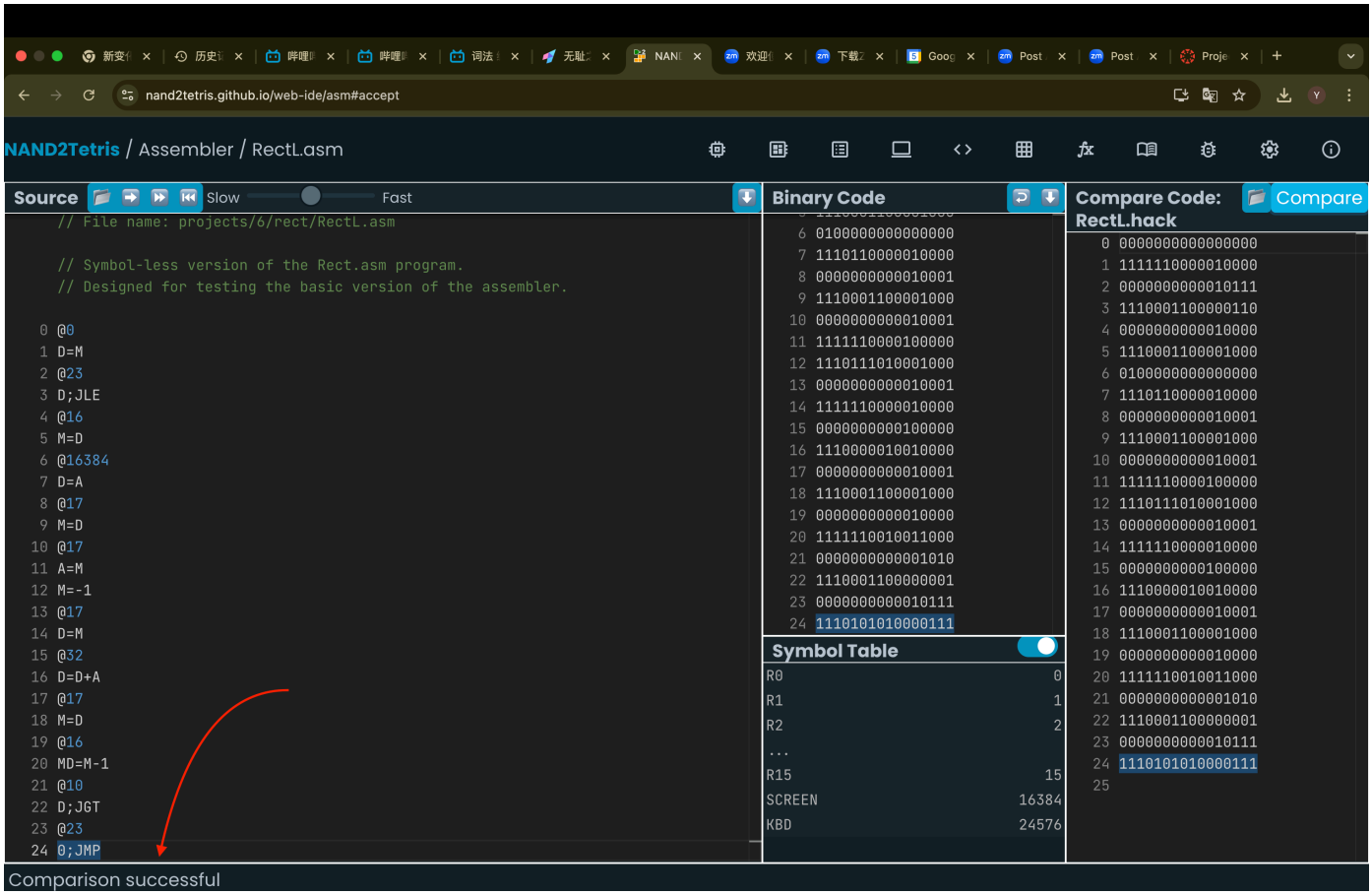
```
python Assembler.py XXX.asm
```

Got the output

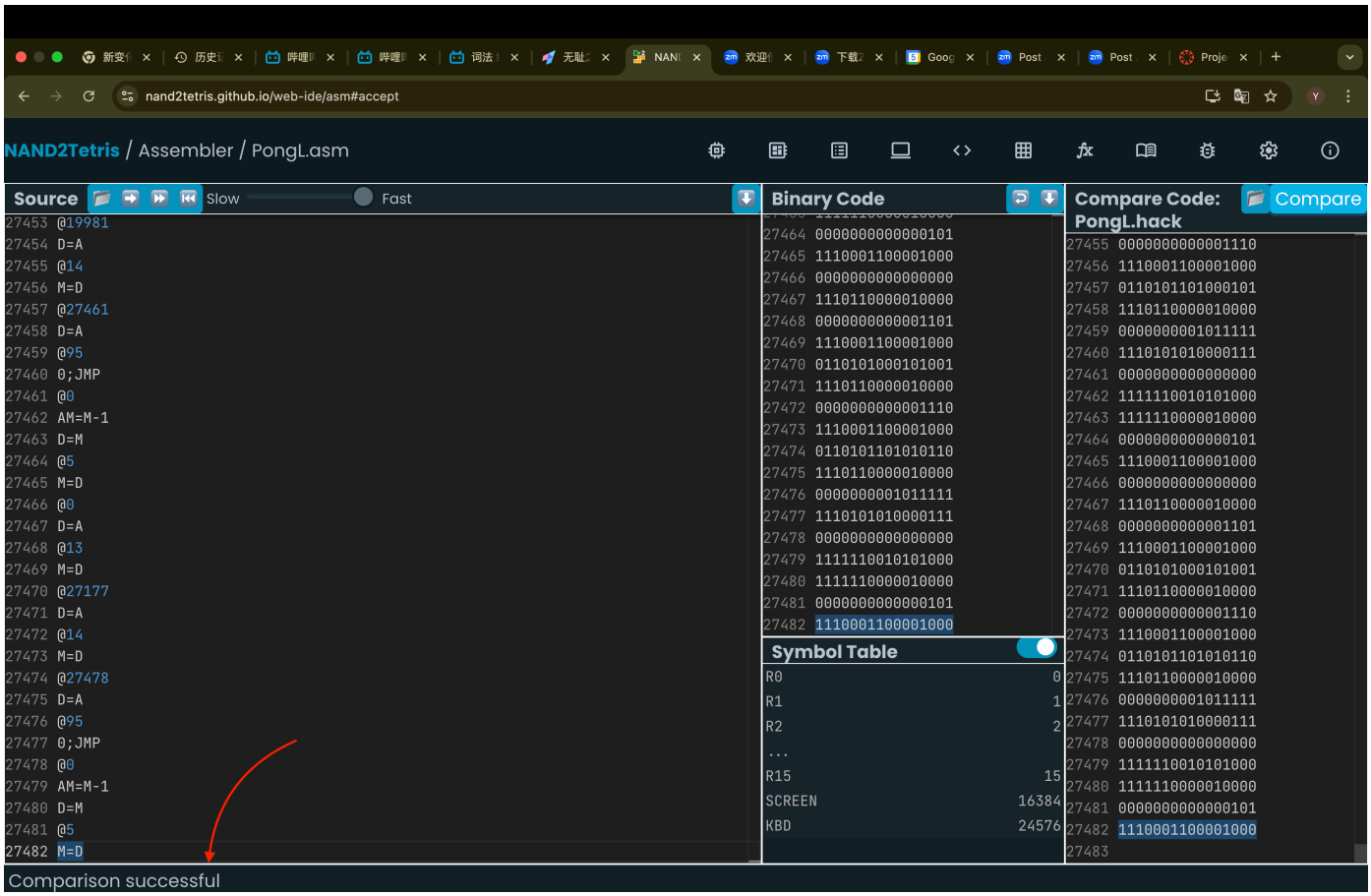
MaxL.asm $\boxed{100\%match}$



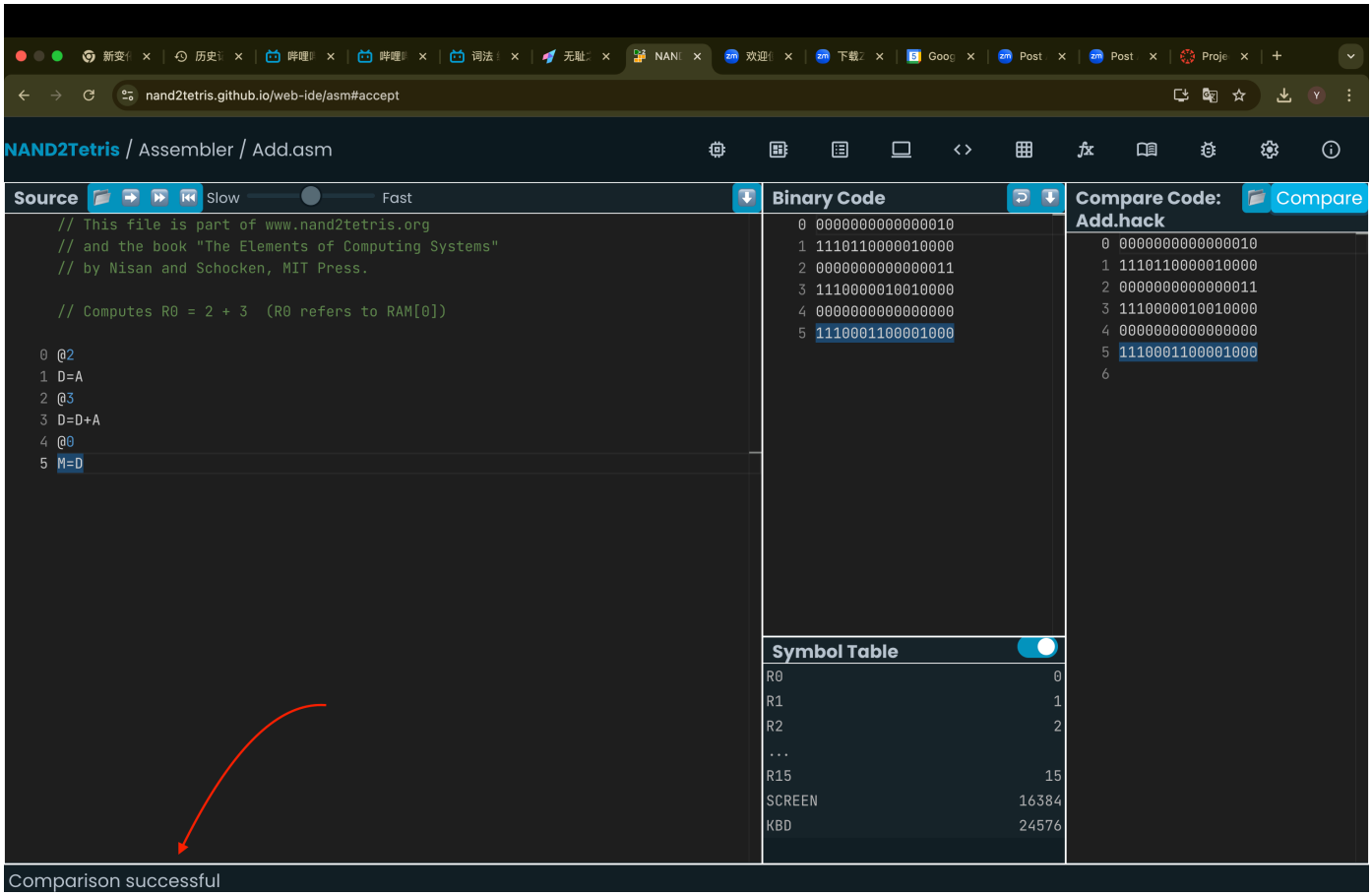
RectL.asm $\boxed{100\%match}$



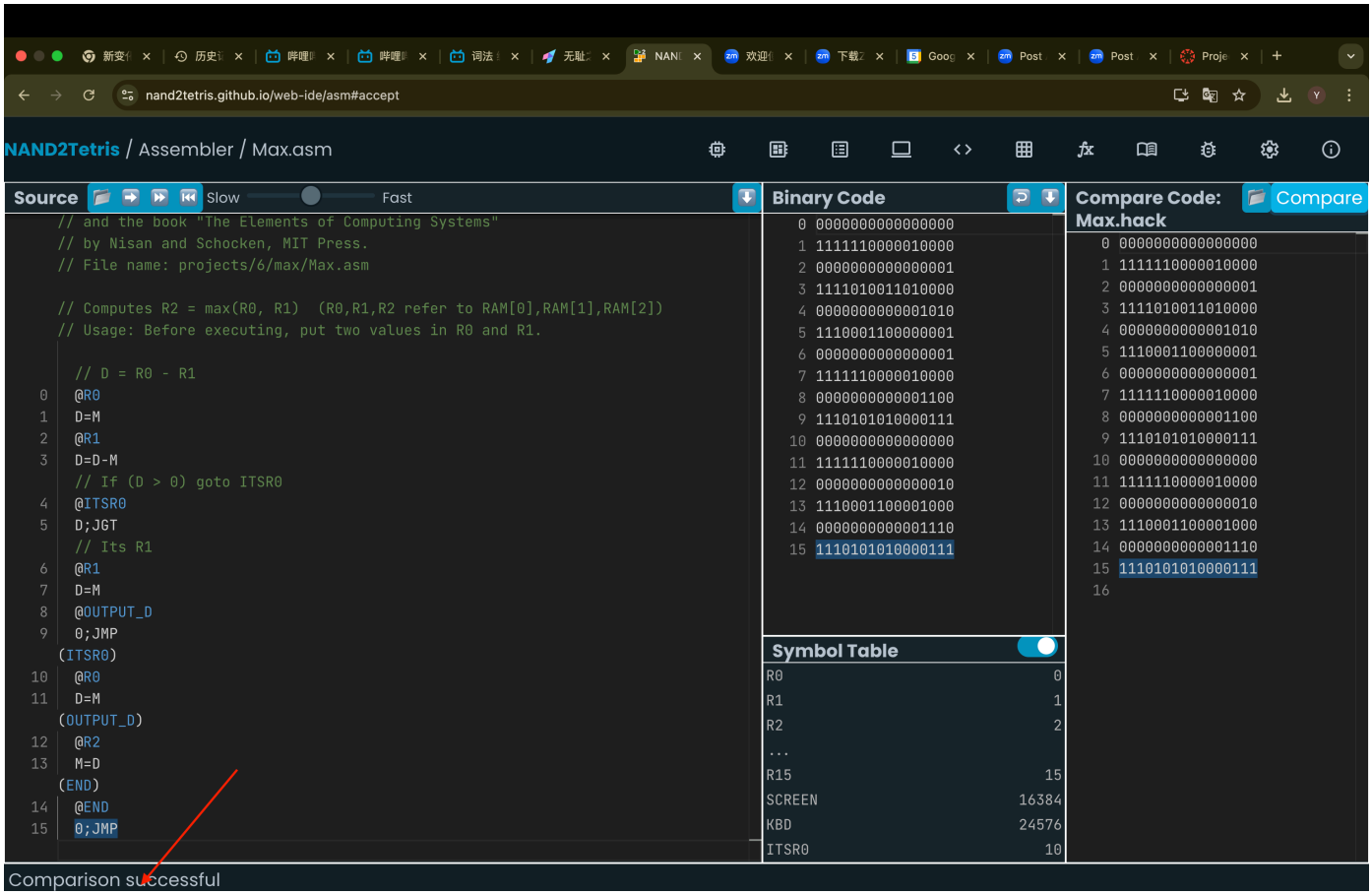
PongL.asm $\boxed{100\%match}$



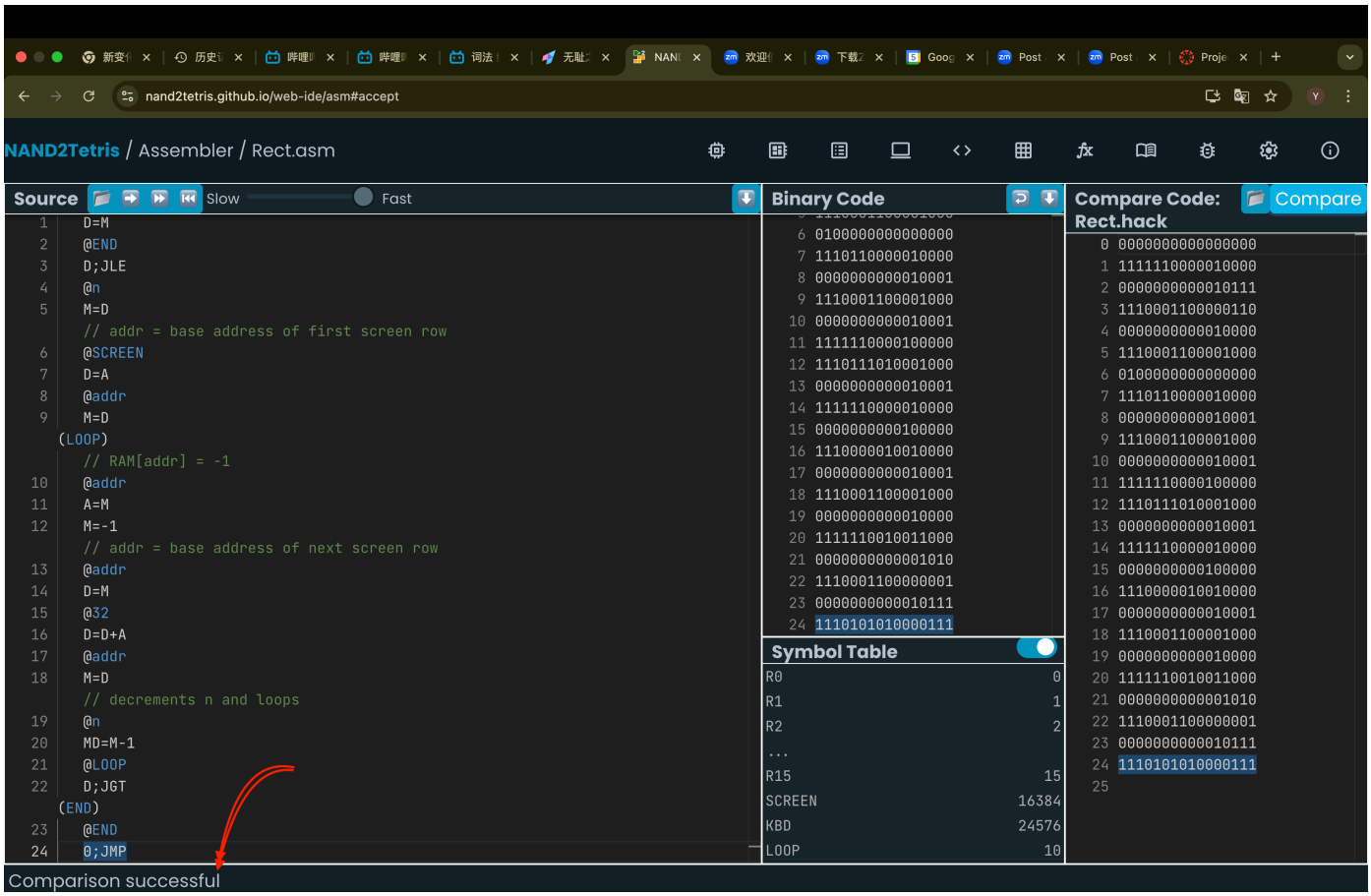
Add.asm $\boxed{100\%match}$



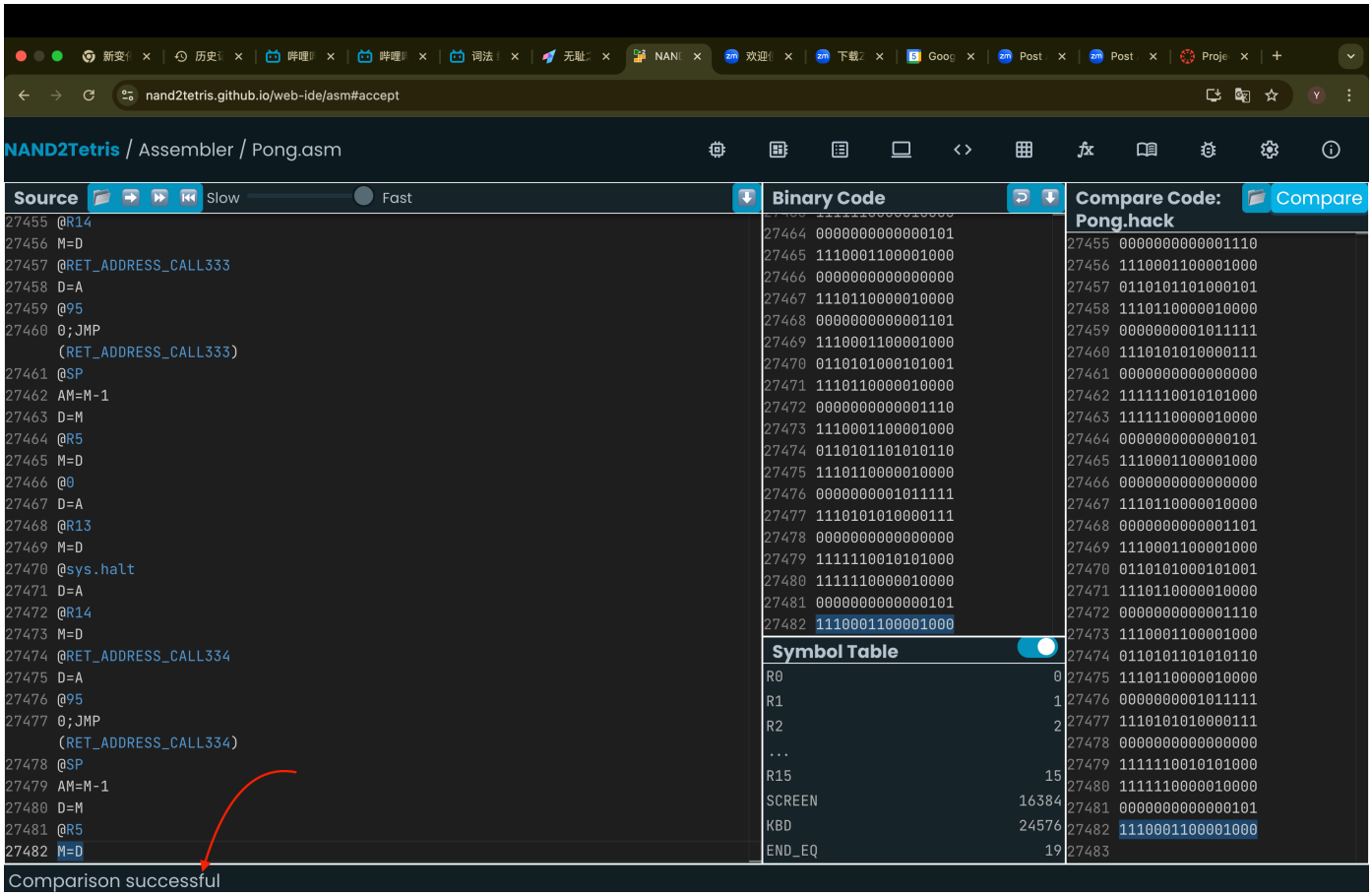
Max.asm 100%match



Rect.asm 100%match

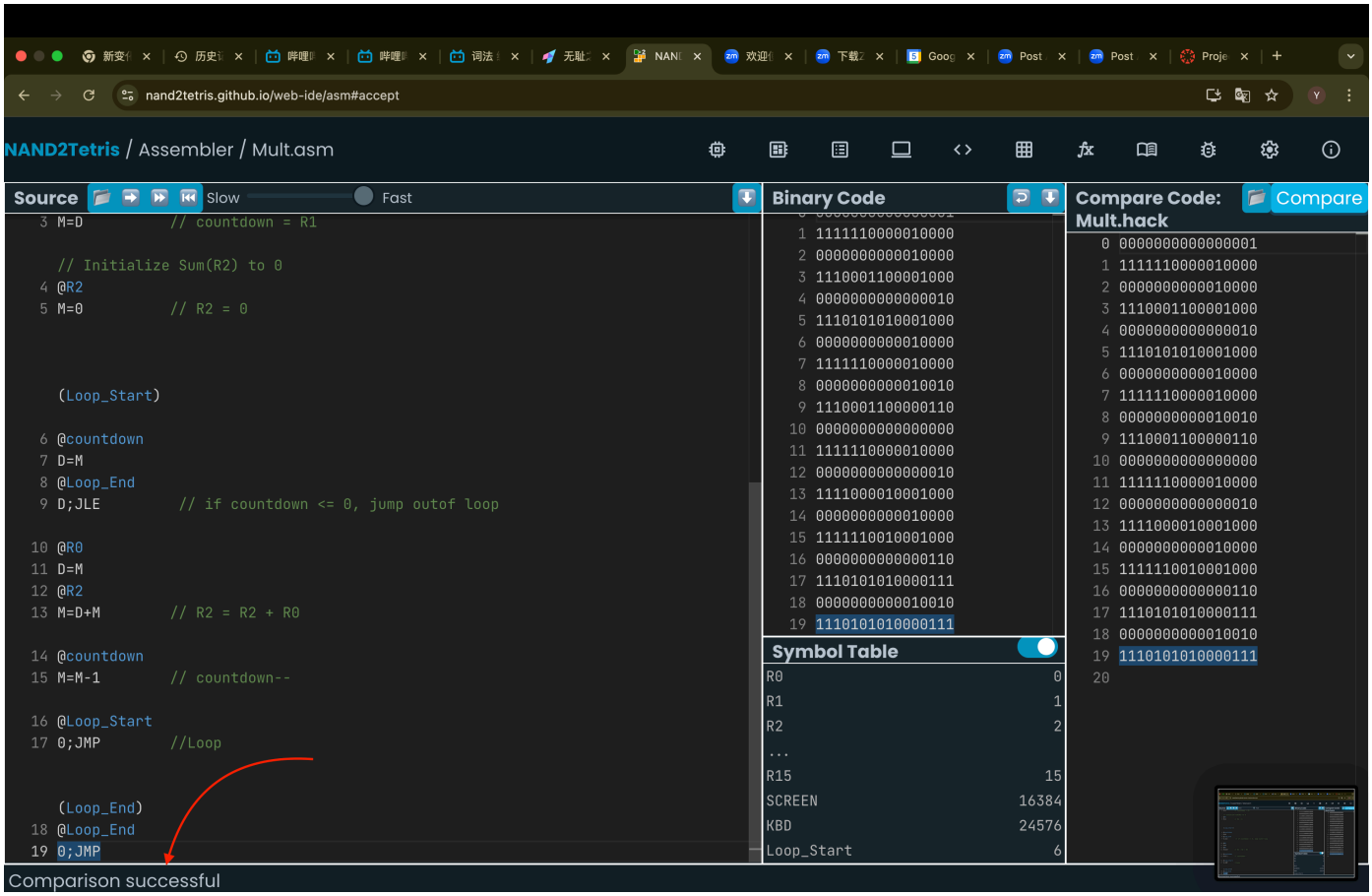


Pong.asm 100%match



####

Mult.asm 100%match



Fill.asm $100\%match$