

Project6 for Intro to Computer Systems 2025 spring

Yiming Cheng

12450588

Project6 for Intro to Computer Systems 2025 spring

[Before All](#)

[File Structure](#)

[Quick Setup](#)

[Class Structure](#)

[SymbolTable.py](#)

[Module Introduction](#)

[Features](#)

[Running Environment](#)

[Tests](#)

[Lexer.py](#)

[Module Introduction](#)

[Features](#)

[Running Environment](#)

[Tests](#)

[Parser.py](#)

[Module Introduction](#)

[Features](#)

[Running Environment](#)

[Tests](#)

[Coding.py](#)

[Module Introduction](#)

Features

Running Environment

Tests

`Assembler.py`

Module Introduction

Features

Running Environment

Tests:

MaxL.asm	100%match
RectL.asm	100%match
PongL.asm	100%match
Add.asm	100%match
Max.asm	100%match
Rect.asm	100%match
Pong.asm	100%match
Mult.asm	100%match
Fill.asm	100%match

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\Assembler.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

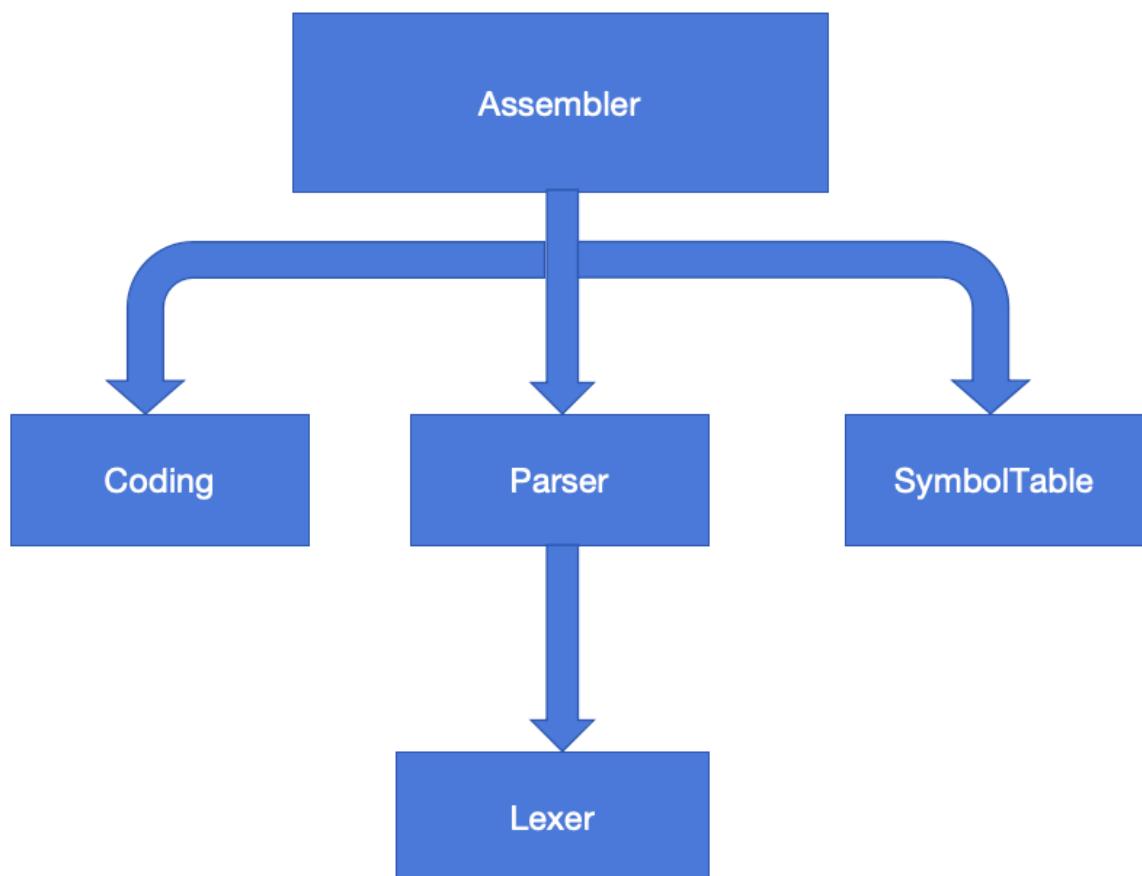
You will get `\path\to\input.hack`

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.contains_symbol("TEST")
    print("✓ Delete by name test passed.")
```

```

# Test deleting a symbol by position
st.del_symbol(symbol_position=4)
assert not st.contains_symbol("THAT")
print("✓ Delete by position test passed.")
print("✓ Contains 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.
✓ Contains 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

- (base) eamin@EamindeMacBook-Pro src % python Parser.py
Test 1: A-command Parsing
Current token: 3, value: @
A-command parsed: token=1, symbol=100
✓ A-command Test Passed
Test 2: C-command Parsing (dest=comp)
Current token: 2, value: D
✓ C-command Test Passed (dest=comp)
Test 3: C-command Parsing (comp;jump)
Current token: 2, value: D
✓ C-command Test Passed (comp;jump)
Test 4: L-command Parsing
Current token: 3, value: (
✓ L-command Test Passed
Test 5: C-command Parsing (M=D+1)
Current token: 2, value: M
✓ C-command Test Passed (M=D+1)

All tests passed successfully!

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

- (base) eamin@EamindeMacBook-Pro src % python Coding.py

Test 1: A-command Generation

✅ A-command Test Passed

Test 2: C-command Generation (dest=D, comp=A, jump=JGT)

Final Result: 1110110000010001

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

Test 3: C-command Generation (dest=M, comp=D+1, jump=JMP)

Final Result: 1110011111001111

✅ C-command Test Passed (dest=M, comp=D+1, jump=JMP)

Test 4: Destination (dest=M)

✅ Destination Test Passed (dest=M)

Test 5: Computation (comp=D+A)

✅ Computation Test Passed (comp=D+A)

Test 6: Jump (jump=JNE)

✅ Jump Test Passed (jump=JNE)

All tests passed successfully!

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 100%match

NAND2Tetris / Assembler / MaxL.asm

Source Slow Fast

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/6/max/MaxL.asm

// Symbol-less version of the Max.asm program.
// Designed for testing the basic version of the assembler.

0 @0
1 D=M
2 @1
3 D=D-M
4 @10
5 D;JGT
6 @1
7 D=M
8 @12
9 0;JMP
10 @0
11 D=M
12 @2
13 M=D
14 @14
15 0;JMP
```

Binary Code

0	0000000000000000
1	1111110000010000
2	0000000000000001
3	1111010011010000
4	00000000000001010
5	1110001100000001
6	0000000000000001
7	1111110000010000
8	0000000000001100
9	1110101010000111
10	0000000000000000
11	1111110000010000
12	00000000000000010
13	1110001100001000
14	00000000000001110
15	1110101010000111

Compare Code: Compare MaxL.hack

0	0000000000000000
1	1111110000010000
2	0000000000000001
3	1111010011010000
4	00000000000001010
5	1110001100000001
6	0000000000000001
7	1111110000010000
8	0000000000001100
9	1110101010000111
10	0000000000000000
11	1111110000010000
12	00000000000000010
13	1110001100001000
14	00000000000001110
15	1110101010000111
16	

Symbol Table

R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576

Comparison successful

RectL.asm 100%match

NAND2Tetris / Assembler / RectL.asm

Source Slow Fast

```
// File name: projects/6/rect/RectL.asm

// Symbol-less version of the Rect.asm program.
// Designed for testing the basic version of the assembler.

0 @0
1 D=M
2 @23
3 D;JLE
4 @16
5 M=D
6 @16384
7 D=A
8 @17
9 M=D
10 @17
11 A=M
12 M=-1
13 @17
14 D=M
15 @32
16 D=D+A
17 @17
18 M=D
19 @16
20 MD=M-1
21 @10
22 D;JGT
23 @23
24 0;JMP
```

Binary Code

0	1100011000010000
1	0100000000000000
2	1101011000010000
3	00000000000010001
4	1110001100001000
5	00000000000010001
6	11100011000010001
7	00000000000010000
8	1111110000010000
9	00000000000010000
10	11100011000010000
11	1111110000010000
12	11101110100010000
13	00000000000010001
14	1111110000010000
15	00000000000010000
16	111000010010000
17	00000000000010001
18	11100011000010000
19	00000000000010000
20	11111100010010000
21	00000000000010100
22	1110001100000001
23	00000000000010111
24	1110101010000111

Compare Code: Compare RectL.hack

0	0000000000000000
1	1111110000010000
2	0000000000000001
3	1111001100000001
4	0000000000000000
5	1110001100000001
6	0100000000000000
7	1110110000010000
8	0000000000000001
9	1110001100000001
10	0000000000000001
11	1111110000010000
12	11101110100010000
13	0000000000000001
14	1111110000010000
15	0000000000000000
16	1110000100100000
17	0000000000000001
18	11100011000010000
19	00000000000010000
20	11111100010010000
21	00000000000010100
22	1110001100000001
23	00000000000010111
24	1110101010000111
25	

Symbol Table

R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576

Comparison successful

PongL.asm

100%match

NAND2Tetris / Assembler / PongLasm

Source Binary Code

Compare Code:

Symbol Table	Value
R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576

Comparison successful

```
27453 @19981
27454 D=A
27455 @14
27456 M=D
27457 @27461
27458 D=A
27459 @95
27460 @;JMP
27461 @0
27462 AM=M-1
27463 D=M
27464 @5
27465 M=D
27466 @0
27467 D=A
27468 @13
27469 M=D
27470 @27177
27471 D=A
27472 @14
27473 M=D
27474 @27478
27475 D=A
27476 @95
27477 @;JMP
27478 @0
27479 AM=M-1
27480 D=M
27481 @5
27482 M=D
```

```
27464 0000000000000101
27465 1110001100001000
27466 0000000000000000
27467 1110110000010000
27468 0000000000000101
27469 1110001100001000
27470 0110101000101001
27471 1110110000010000
27472 0000000000000110
27473 1110001100001000
27474 0110101101010110
27475 1110110000010000
27476 0000000001011111
27477 1110101010000111
27478 0000000000000000
27479 1111110010101000
27480 1111110000010000
27481 0000000000000101
27482 1110001100001000
```

```
27455 0000000000001110
27456 1110001100001000
27457 0110101101000101
27458 1110110000010000
27459 0000000001011111
27460 1110101010000111
27461 0000000000000000
27462 1111110010101000
27463 1111110000010000
27464 0000000000000101
27465 1110001100001000
27466 0000000000000000
27467 1110110000010000
27468 0000000000000101
27469 1110001100001000
27470 0110101000101001
27471 1110110000010000
27472 0000000000000110
27473 1110001100001000
27474 0110101101010110
27475 1110110000010000
27476 0000000001011111
27477 1110101010000111
27478 0000000000000000
27479 1111110010101000
27480 1111110000010000
27481 0000000000000101
27482 1110001100001000
27483
```

Add.asm

100%match

NAND2Tetris / Assembler / Add.asm

Source Slow Fast

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.

// Computes R0 = 2 + 3 (R0 refers to RAM[0])

0 @2
1 D=A
2 @3
3 D=D+A
4 @0
5 M=D
```

Binary Code

0	00000000000000000000000000000010
1	11101100000010000
2	00000000000000000000000000000011
3	11100000010010000
4	00000000000000000000000000000000
5	11100001100001000

Compare Code: Compare Add.hack

0	00000000000000000000000000000010
1	11101100000010000
2	00000000000000000000000000000011
3	11100000010010000
4	00000000000000000000000000000000
5	11100001100001000
6	

Symbol Table

R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576

Comparison successful

Max.asm 100%match

NAND2Tetris / Assembler / Max.asm

Source Slow Fast

```
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/6/max/Max.asm

// Computes R2 = max(R0, R1) (R0,R1,R2 refer to RAM[0],RAM[1],RAM[2])
// Usage: Before executing, put two values in R0 and R1.

// D = R0 - R1
0 @R0
1 D=M
2 @R1
3 D=D-M
// If (D > 0) goto ITSR0
4 @ITSR0
5 D;JGT
// Its R1
6 @R1
7 D=M
8 @OUTPUT_D
9 0;JMP
(ITSR0)
10 @R0
11 D=M
(OUTPUT_D)
12 @R2
13 M=D
(END)
14 @END
15 0;JMP
```

Binary Code

0	00000000000000000000000000000000
1	11111100000010000
2	00000000000000000000000000000001
3	1111010011010000
4	00000000000001010
5	1110001100000001
6	0000000000000001
7	1111110000010000
8	0000000000001100
9	1110101010000111
10	0000000000000000
11	1111110000010000
12	0000000000000010
13	1110001100001000
14	0000000000001110
15	1110101010000111

Compare Code: Compare Max.hack

0	00000000000000000000000000000000
1	11111100000010000
2	00000000000000000000000000000001
3	1111010011010000
4	00000000000001010
5	1110001100000001
6	0000000000000001
7	1111110000010000
8	0000000000001100
9	1110101010000111
10	0000000000000000
11	1111110000010000
12	0000000000000010
13	1110001100001000
14	0000000000001110
15	1110101010000111
16	

Symbol Table

R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576
ITSR0	10

Comparison successful

Rect.asm

100%match

NAND2Tetris / Assembler / Rect.asm

Source Slow Fast

Binary Code

6	0100000000000000
7	1110110000010000
8	0000000000010001
9	1110001100001000
10	0000000000010001
11	1111110000010000
12	1110111010001000
13	0000000000010001
14	1111110000010000
15	0000000000010000
16	1110000010001000
17	0000000000010001
18	1110001100001000
19	0000000000010000
20	1111110010001000
21	0000000000001010
22	1110001100000001
23	0000000000010111
24	1110101010000111

Compare Code: Compare Rect.hack

Symbol Table

R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576
LOOP	10

Comparison successful

Pong.asm

100%match

NAND2Tetris / Assembler / Pong.asm

Source	Binary Code	Compare Code:
27455 @R14	27464 0000000000000101	27455 0000000000001110
27456 M=D	27465 110001100001000	27456 111001100001000
27457 @RET_ADDRESS_CALL333	27466 0000000000000000	27457 0110101101000101
27458 D=A	27467 1100110000010000	27458 110110000010000
27459 @95	27468 00000000000001101	27459 0000000001011111
27460 0;JMP	27469 1100011000010000	27460 1101010100001111
(RET_ADDRESS_CALL333)	27470 01101010000101001	27461 0000000000000000
27461 @SP	27471 1100110000010000	27462 11111100010101000
27462 AM=M-1	27472 00000000000001110	27463 1111110000010000
27463 D=M	27473 1100011000010000	27464 0000000000000101
27464 @R5	27474 0110101101010110	27465 1100011000010000
27465 M=D	27475 1100110000010000	27466 0000000000000000
27466 @0	27476 0000000001011111	27467 110110000010000
27467 D=A	27477 110010100000111	27468 00000000000001101
27468 @R13	27478 0000000000000000	27469 1100011000010000
27469 M=D	27479 11111100010101000	27470 01101010000101001
27470 @sys.halt	27480 1111110000010000	27471 1100110000010000
27471 D=A	27481 0000000000000101	27472 0000000000000110
27472 @R14	27482 1100011000010000	27473 1100011000010000
27473 M=D		
27474 @RET_ADDRESS_CALL334		
27475 D=A		
27476 @95		
27477 0;JMP		
(RET_ADDRESS_CALL334)		
27478 @SP		
27479 AM=M-1		
27480 D=M		
27481 @R5		
27482 M=D		

Comparison successful

####

Mult.asm 100%match

NAND2Tetris / Assembler / Mult.asm

Source	Binary Code	Compare Code:
Slow	Fast	Compare
3 M=D // countdown = R1	1 1111110000010000	Mult.hack
// Initialize Sum(R2) to 0	2 0000000000010000	0 0000000000000001
4 @R2	3 1110001100001000	1 1111110000010000
5 M=0 // R2 = 0	4 0000000000000010	2 0000000000000001
(Loop_Start)	5 1110101010001000	3 1110001100001000
6 @countdown	6 0000000000001000	4 0000000000000010
7 D=M	7 1111110000010000	5 1110101010001000
8 @Loop_End	8 0000000000001000	6 0000000000001000
9 D;JLE // if countdown <= 0, jump out of loop	9 1110001100000110	7 1111110000010000
10 @R0	10 0000000000000000	8 0000000000000001
11 D=M	11 1111110000010000	9 1110001100000110
12 @R2	12 0000000000000010	10 0000000000000000
13 M=D+M // R2 = R2 + R0	13 1111000010001000	11 1111110000010000
14 @countdown	14 0000000000000000	12 0000000000000000
15 M=M-1 // countdown--	15 1111110000010000	13 1111000010001000
16 @Loop_Start	16 0000000000000010	14 0000000000000000
17 0;JMP //Loop	17 1110101010000111	15 1111110000010000
(Loop_End)	18 0000000000001000	16 0000000000000110
18 @Loop_End	19 1110101010000111	17 1110101010000111
19 0;JMP		18 0000000000000100

Comparison successful

Fill.asm 100%match

NAND2Tetris / Assembler / Fill.asm

Source	Binary Code	Compare Code:
Slow	Fast	Compare
18 M=-1 // Set pixel to black	19 0000000000010000	Fill.hack
19 @CURSOR	20 1111110111010000	10 0110000000000000
20 D=M+1 // Increment position	21 1110001100001000	11 1110110000010000
21 M=D	22 0000000000001000	12 0000000000001000
22 @LOOP // Return to main loop	23 1110101010000111	13 1111010011010000
23 0;JMP	24 0100000000000000	14 0000000000000010
(CLEAR_WHITE)	25 1110110001000000	15 1110001100000010
// Detect if arrives the start	26 0000000000000000	16 0000000000000000
24 @SCREEN // Start of screen memory	27 1111010011010000	17 1111110000100000
25 D=A-1 // Offset to avoid clearing top-left artifact	28 0000000000000000	18 1110111010001000
26 @CURSOR	29 1110001100000010	19 0000000000001000
27 D=D-M // Compare current position with start	30 0000000000001000	20 1111110111010000
28 @LOOP // If at start, return to loop	31 1111110000100000	21 1110001100001000
29 D;JEQ	32 1110101010000000	22 0000000000000010
(CLEAR)	33 0000000000000000	23 1110101010000111
30 @CURSOR	34 1111110001000000	24 0100000000000000
31 A=M // Access current position	35 1110001100001000	25 1110110001000000
32 M=0 // Set pixel to white	36 0000000000000000	26 0000000000000000
33 @CURSOR	37 1110101010000111	27 1110101010000000
34 D=M-1 // Decrement position		28 0000000000000000
35 M=D		29 1110001100000010
36 @LOOP // Return to main loop		30 0000000000000000
37 0;JMP		31 1111110000100000

Comparison successful