C PROGRAMS:

1. Decimal To Binary Conversion

CODE:

#include <stdio.h>

void decToBinary(int n) {

// array to store binary number

int binaryNum[32];

// counter for binary array

int i = 0;

while (n > 0) {

// storing remainder in binary array

binaryNum[i] = n % 2;

n = n / 2;

i++;

}

// printing binary array in reverse order

for (int j = i - 1; j >= 0; j--)

printf("%d", binaryNum[j]);

}

int main() {

int decimalNum;

printf("Enter a decimal number: ");

scanf("%d", &decimalNum);

printf("Binary equivalent: ");

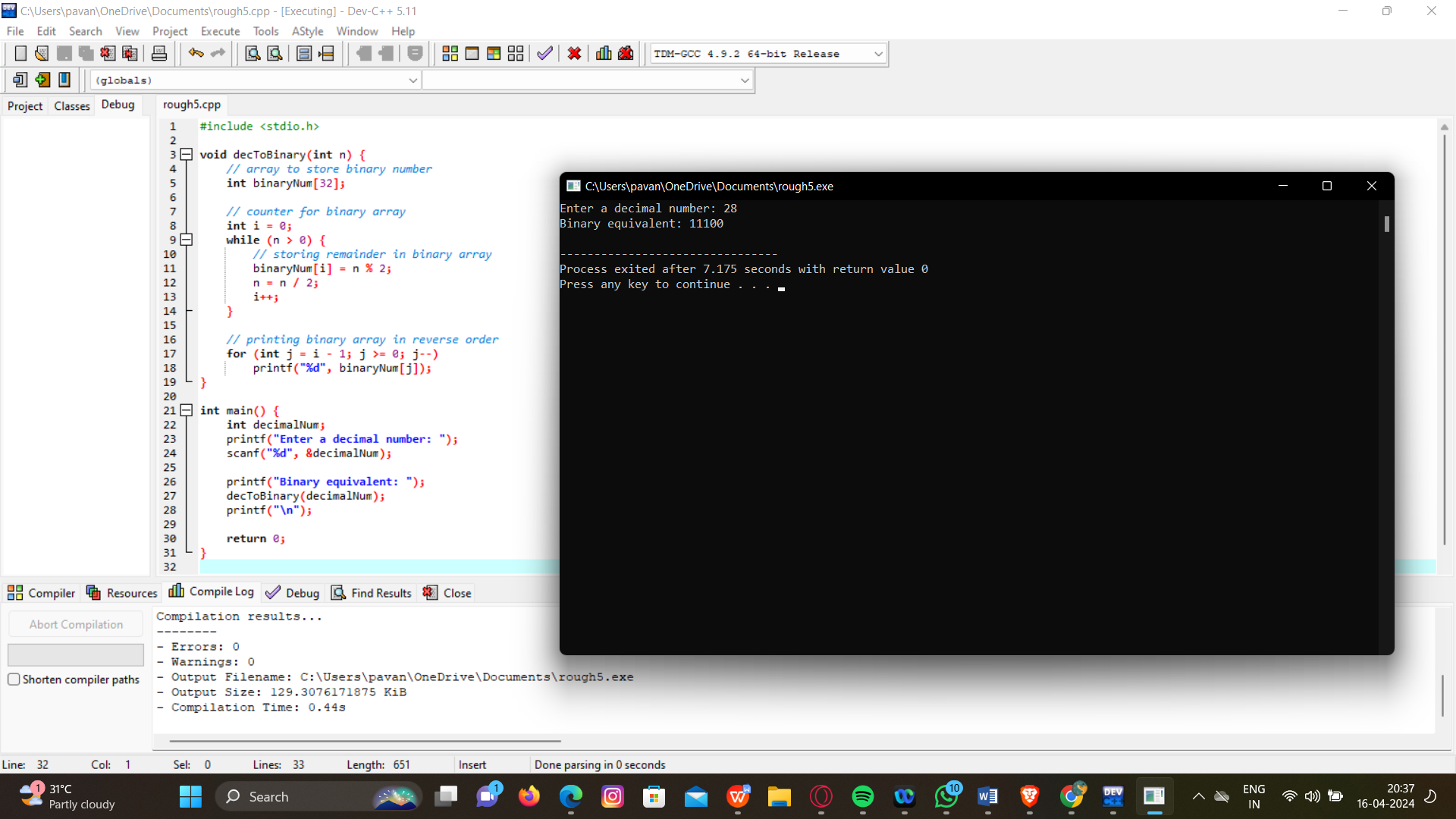
decToBinary(decimalNum);

printf("\n");

return 0;

}

OUTPUT:



1. Hexadecimal To Decimal Conversion

CODE:

#include <stdio.h>

#include <math.h>

int main() {

char hex[10];

int len = 0, dec = 0, base = 1;

printf("Enter a hexadecimal number: ");

scanf("%s", hex);

// Calculating the length of the hexadecimal number

while (hex[len] != '\0') {

len++;

}

len--;

// Converting hexadecimal to decimal

for (int i = len; i >= 0; i--) {

if (hex[i] >= '0' && hex[i] <= '9') {

dec += (hex[i] - '0') \* base;

} else if (hex[i] >= 'A' && hex[i] <= 'F') {

dec += (hex[i] - 'A' + 10) \* base;

} else if (hex[i] >= 'a' && hex[i] <= 'f') {

dec += (hex[i] - 'a' + 10) \* base;

}

base \*= 16;

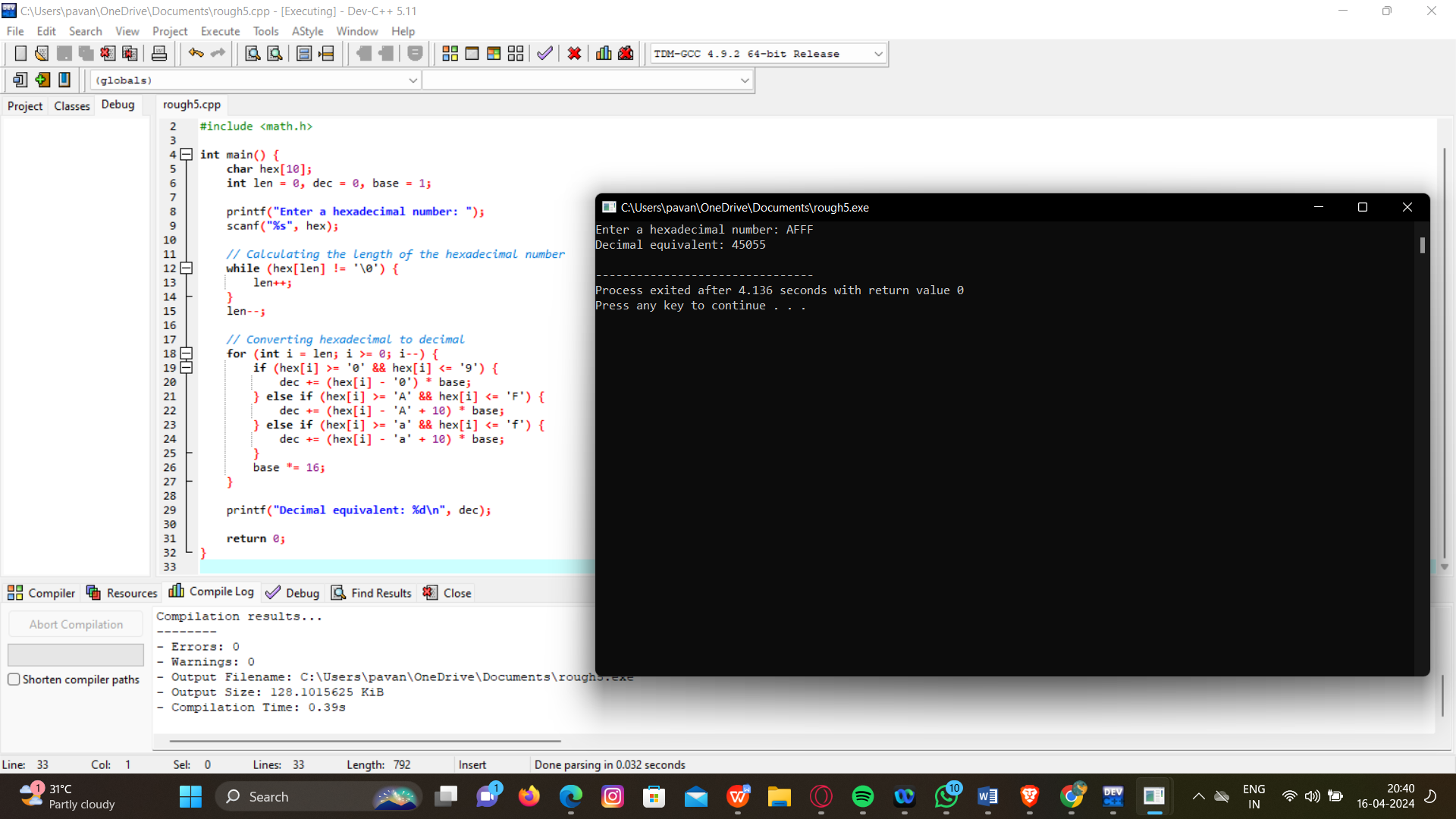
}

printf("Decimal equivalent: %d\n", dec);

return 0;

}

OUTPUT:



1. Decimal To Octal Conversion

CODE:

#include <stdio.h>

int main() {

int decimalNum, octalNum[100], i = 0;

printf("Enter a decimal number: ");

scanf("%d", &decimalNum);

// Convert decimal to octal

while (decimalNum != 0) {

octalNum[i] = decimalNum % 8;

decimalNum /= 8;

i++;

}

printf("Octal equivalent: ");

// Print octal number in reverse order

for (int j = i - 1; j >= 0; j--) {

printf("%d", octalNum[j]);

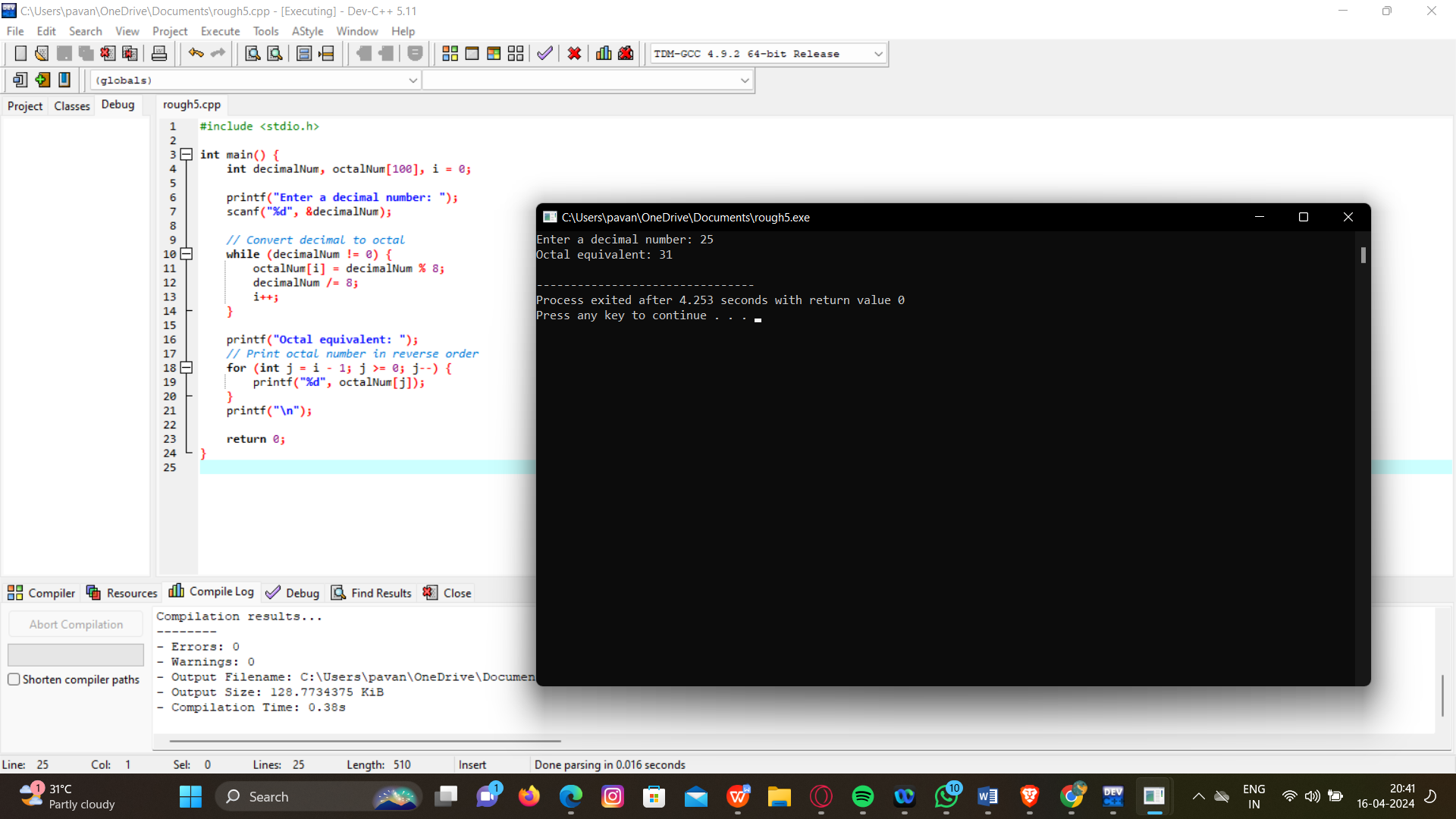
}

printf("\n");

return 0;

}

OUTPUT:



1. Binary To Decimal Conversion

CODE:

#include <stdio.h>

#include <math.h>

int main() {

long long binaryNum;

int decimalNum = 0, i = 0, remainder;

printf("Enter a binary number: ");

scanf("%lld", &binaryNum);

// Convert binary to decimal

while (binaryNum != 0) {

remainder = binaryNum % 10;

binaryNum /= 10;

decimalNum += remainder \* pow(2, i);

i++;

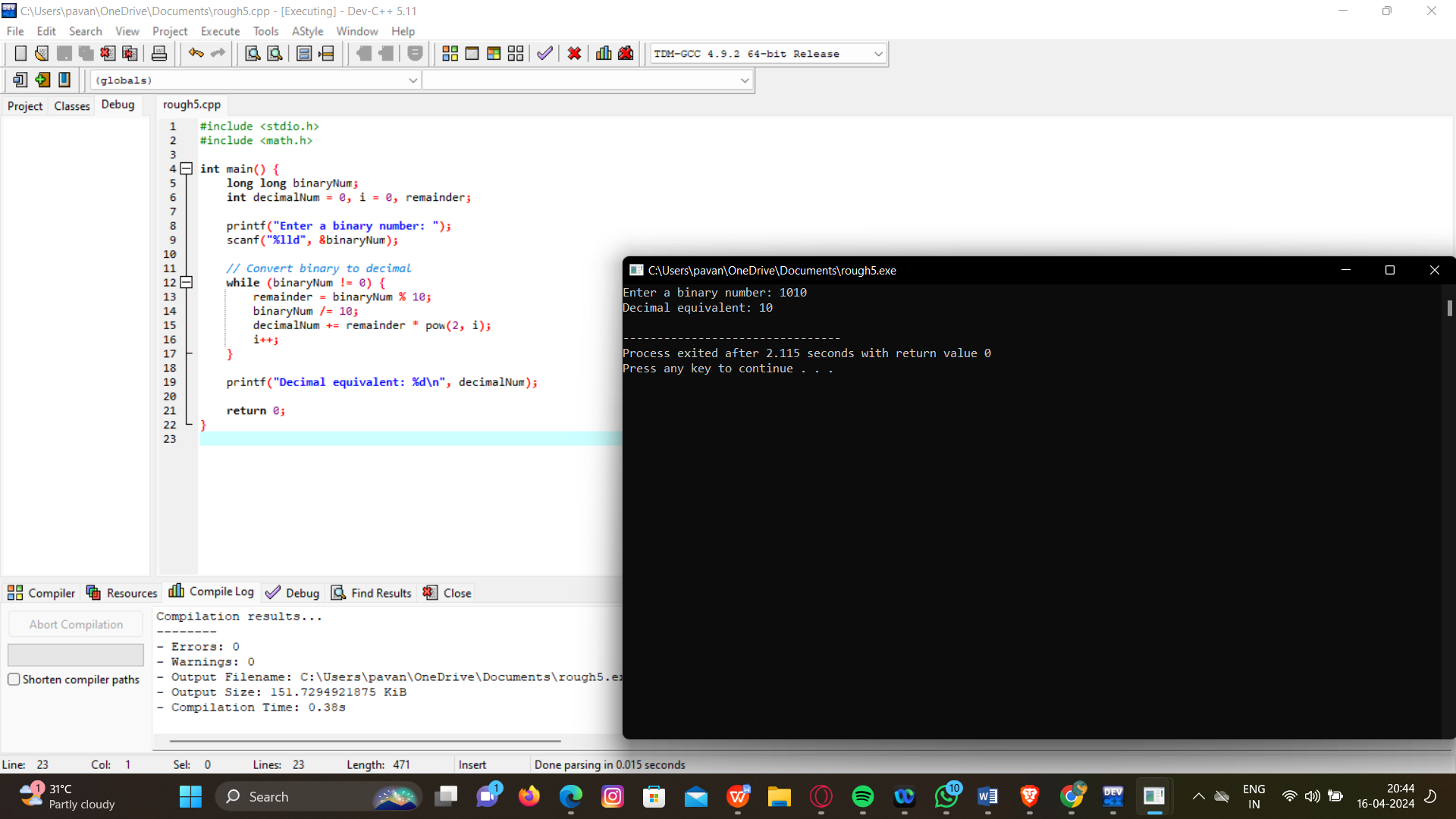
}

printf("Decimal equivalent: %d\n", decimalNum);

return 0;

}

OUTPUT:



1. Two Stage Pipeline

CODE:

#include <stdio.h>

int main() {

int instructions[] = {1, 2, 3, 4, 0}; // Instructions: 1=ADD, 2=SUB, 3=MUL, 4=DIV, 0=END

int data[2] = {10, 5}; // Initial data values

int pc = 0, ir, op1, op2, result;

while (1) {

ir = instructions[pc++]; // Instruction Fetch

switch (ir) {

case 1: // ADD

op1 = data[0];

op2 = data[1];

result = op1 + op2;

printf("ADD: %d + %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 2: // SUB

op1 = data[0];

op2 = data[1];

result = op1 - op2;

printf("SUB: %d - %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 3: // MUL

op1 = data[0];

op2 = data[1];

result = op1 \* op2;

printf("MUL: %d \* %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 4: // DIV

op1 = data[0];

op2 = data[1];

if (op2 != 0) {

result = op1 / op2;

printf("DIV: %d / %d = %d\n", op1, op2, result);

data[0] = result;

} else {

printf("DIVISION BY ZERO ERROR!\n");

}

break;

case 0: // END

printf("Program execution completed.\n");

return 0;

default:

printf("Unknown instruction!\n");

return 1;

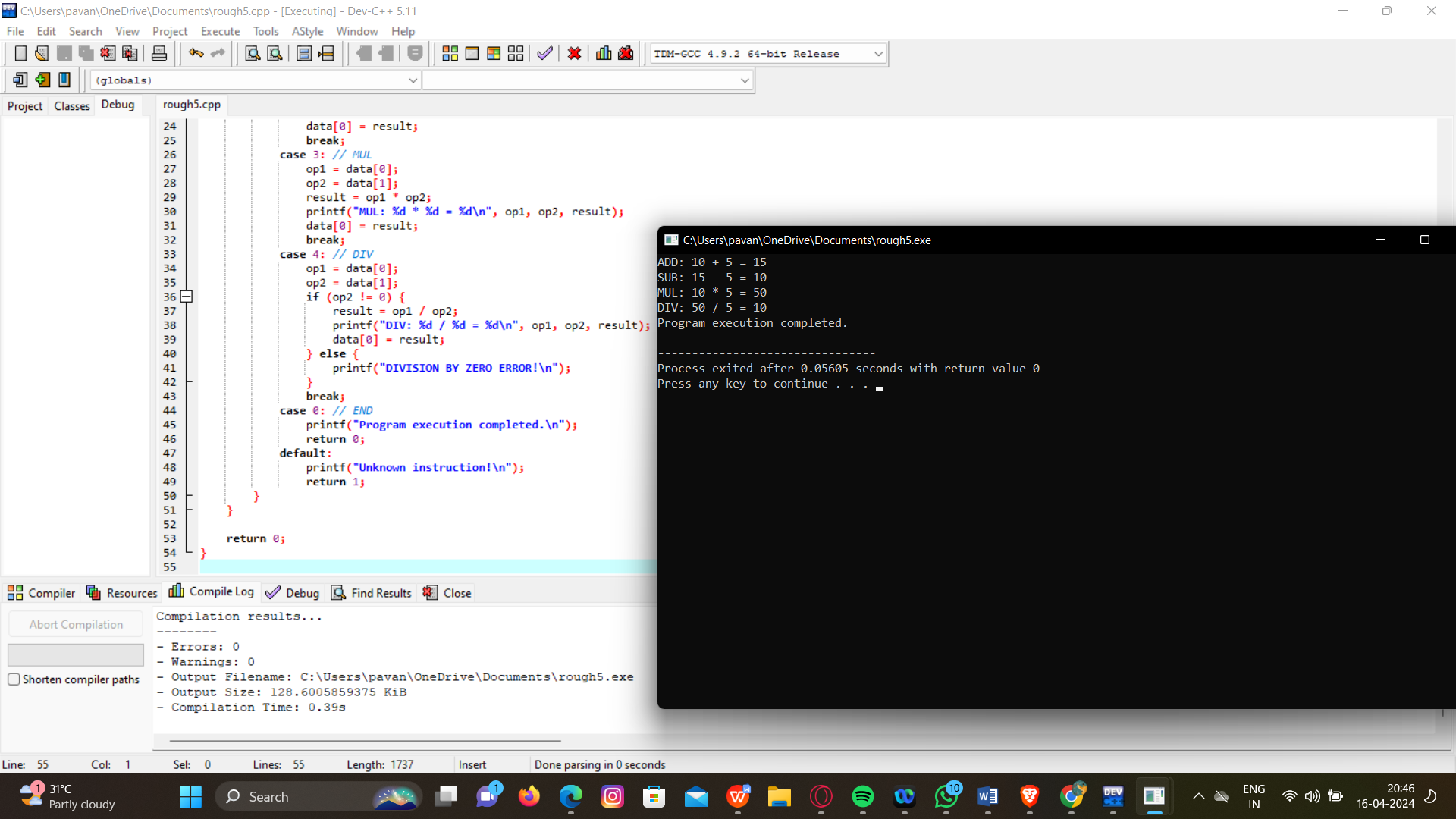
}

}

return 0;

}

OUTPUT:



1. Four Stage Pipeline

CODE:

#include <stdio.h>

int main() {

int instructions[] = {1, 2, 3, 4, 0}; // Instructions: 1=ADD, 2=SUB, 3=MUL, 4=DIV, 0=END

int data[2] = {10, 5}; // Initial data values

int pc = 0, ir, op1, op2, result;

while (1) {

ir = instructions[pc++]; // Instruction Fetch

if (ir == 0) {

printf("Program execution completed.\n");

break;

}

switch (ir) {

case 1: // ADD

op1 = data[0];

op2 = data[1];

result = op1 + op2;

printf("ADD: %d + %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 2: // SUB

op1 = data[0];

op2 = data[1];

result = op1 - op2;

printf("SUB: %d - %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 3: // MUL

op1 = data[0];

op2 = data[1];

result = op1 \* op2;

printf("MUL: %d \* %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 4: // DIV

op1 = data[0];

op2 = data[1];

if (op2 != 0) {

result = op1 / op2;

printf("DIV: %d / %d = %d\n", op1, op2, result);

data[0] = result;

} else {

printf("DIVISION BY ZERO ERROR!\n");

}

break;

default:

printf("Unknown instruction!\n");

return 1;

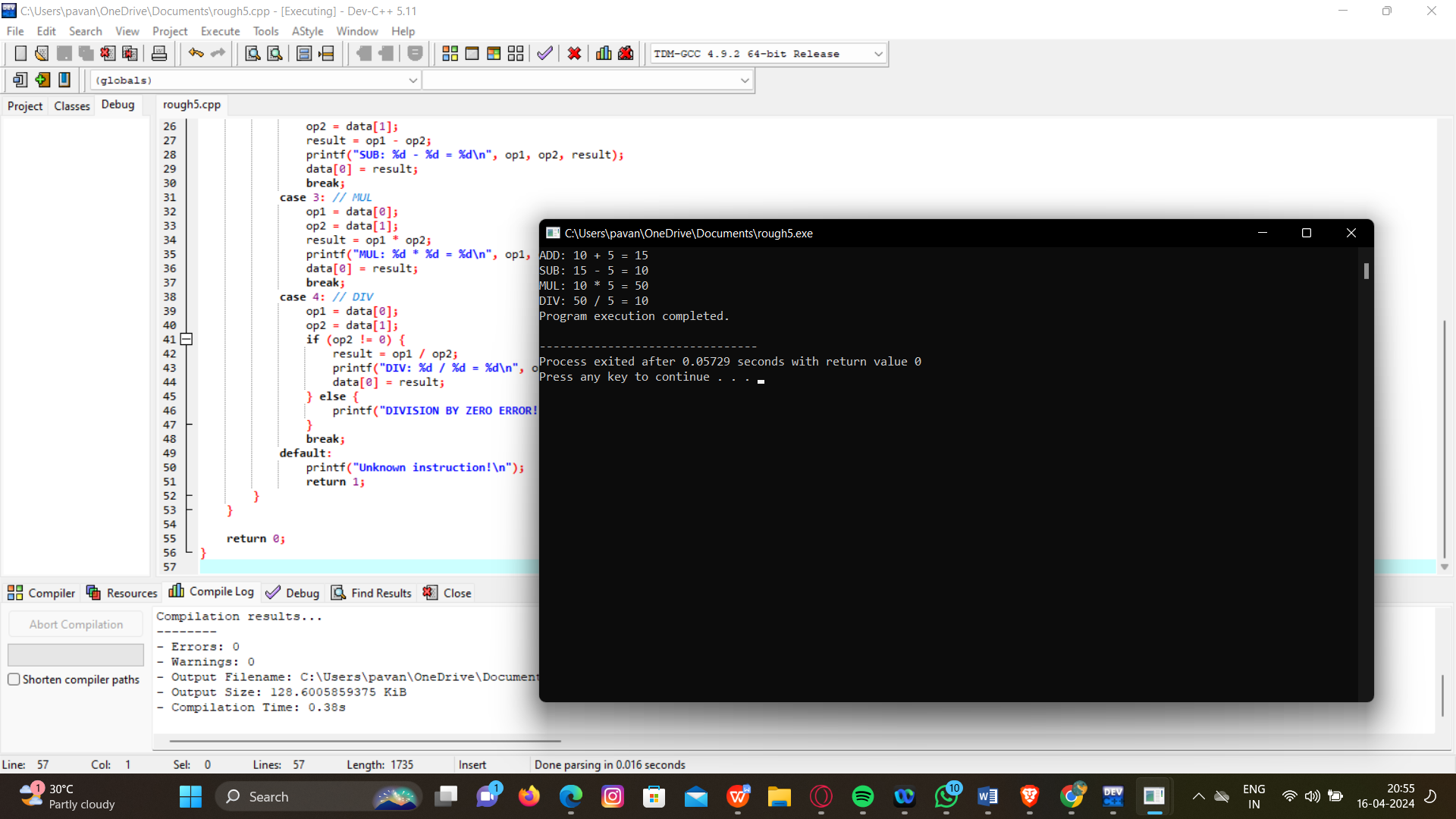
}

}

return 0;

}

OUTPUT:



1. Cpu Performance

CODE:

#include <stdio.h>

int main() {

long long iterations = 1000000000; // Number of iterations for computation

double result = 0.0;

for (long long i = 0; i < iterations; i++) {

result += i \* 2.5; // Perform some computation

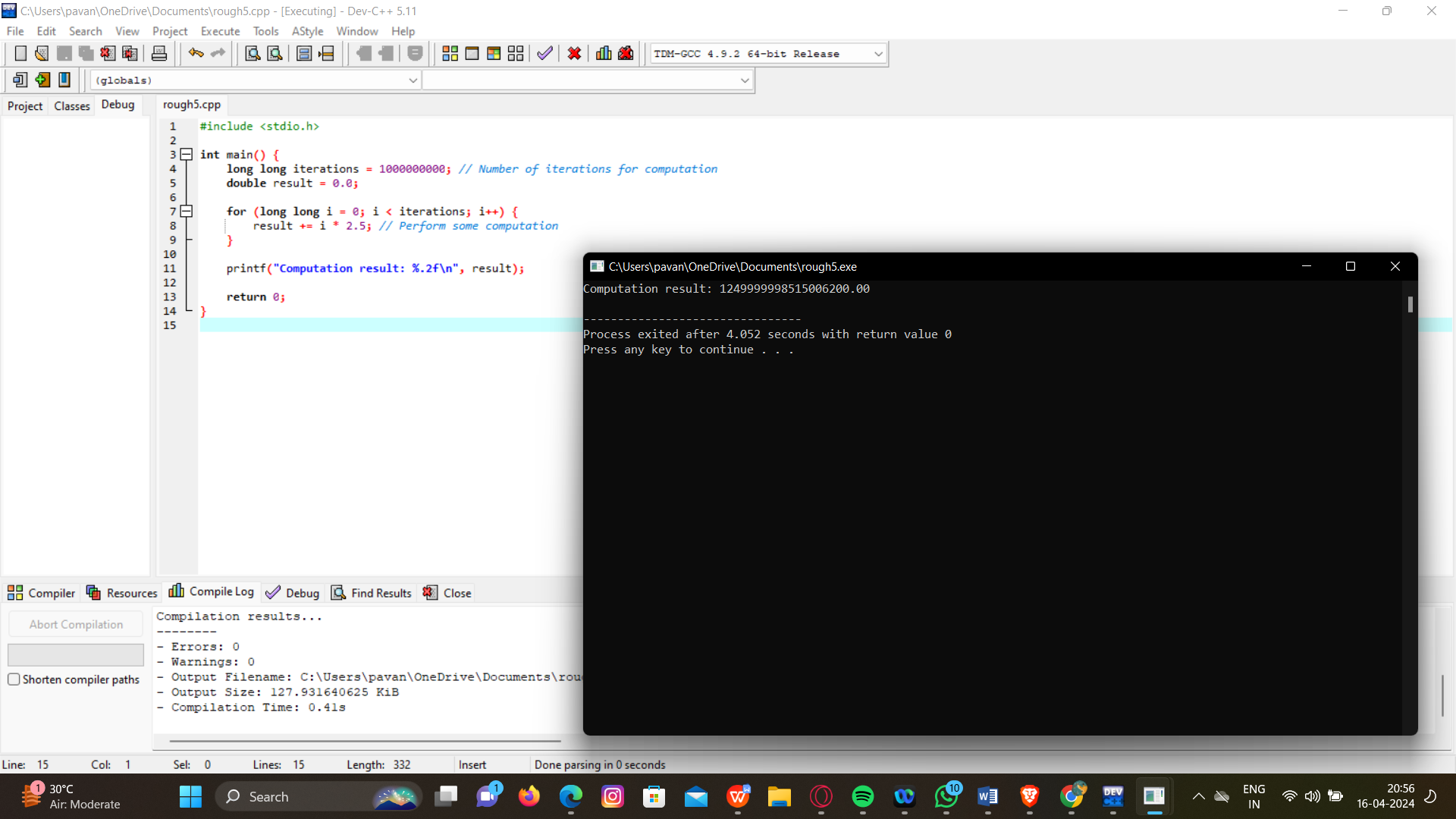
}

printf("Computation result: %.2f\n", result);

return 0;

}

OUTPUT:



1. Four Stage And Operation

CODE:

#include <stdio.h>

int main() {

int instructions[] = {1, 2, 0}; // Instructions: 1=ADD, 2=SUB, 0=END

int data[2] = {10, 5}; // Initial data values

int pc = 0, ir, op1, op2, result;

while (1) {

ir = instructions[pc++]; // Instruction Fetch

if (ir == 0) {

printf("Program execution completed.\n");

break;

}

switch (ir) {

case 1: // ADD

op1 = data[0];

op2 = data[1];

result = op1 + op2;

printf("ADD: %d + %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 2: // SUB

op1 = data[0];

op2 = data[1];

result = op1 - op2;

printf("SUB: %d - %d = %d\n", op1, op2, result);

data[0] = result;

break;

default:

printf("Unknown instruction!\n");

return 1;

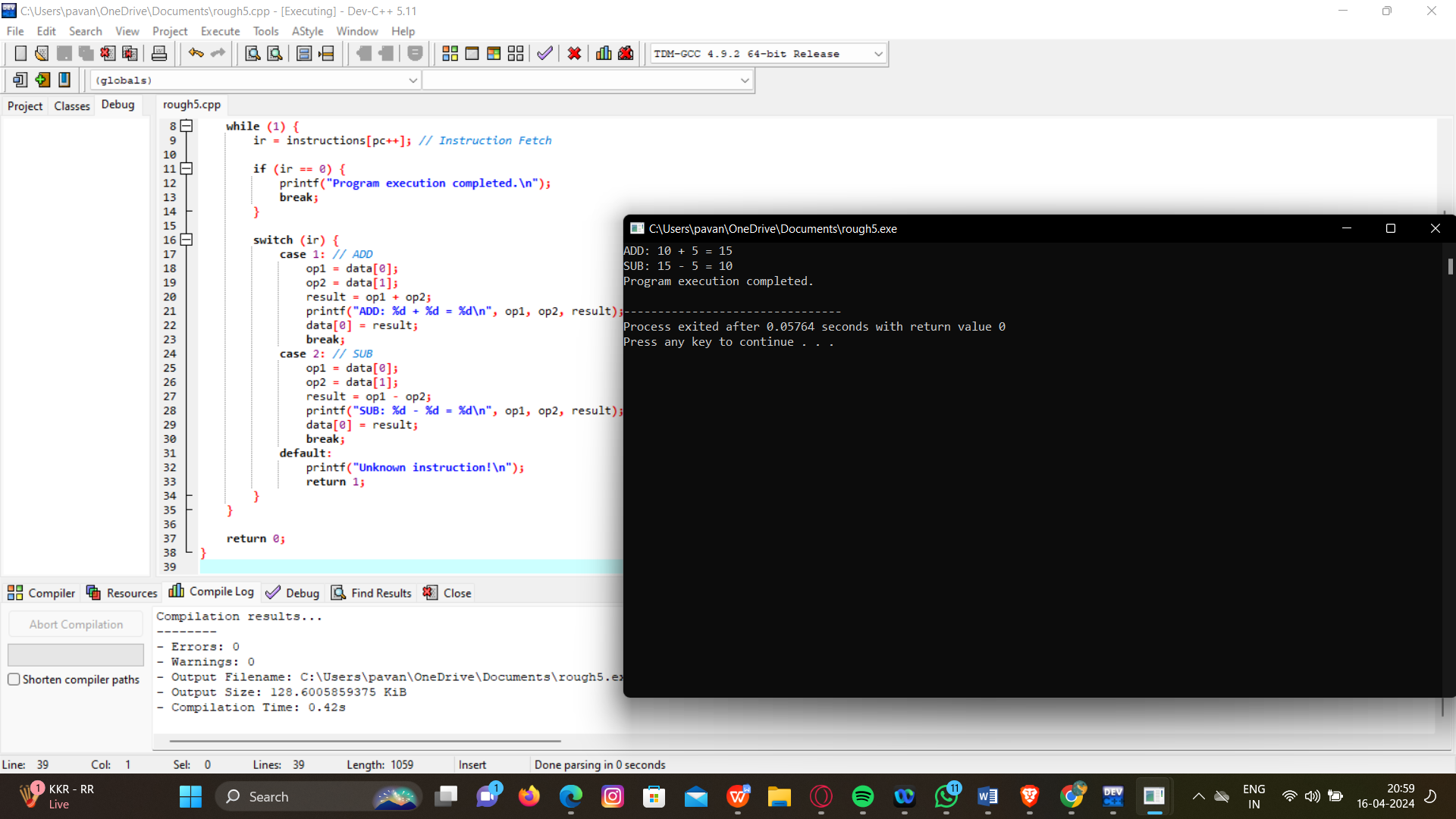
}

}

return 0;

}

OUTPUT:



1. Three Stage And Operation

CODE:

#include <stdio.h>

int main() {

int instructions[] = {1, 2, 0}; // Instructions: 1=ADD, 2=SUB, 0=END

int data[2] = {10, 5}; // Initial data values

int pc = 0, ir, op1, op2, result;

while (1) {

ir = instructions[pc++]; // Instruction Fetch

if (ir == 0) {

printf("Program execution completed.\n");

break;

}

switch (ir) {

case 1: // ADD

op1 = data[0];

op2 = data[1];

result = op1 + op2;

printf("ADD: %d + %d = %d\n", op1, op2, result);

data[0] = result;

break;

case 2: // SUB

op1 = data[0];

op2 = data[1];

result = op1 - op2;

printf("SUB: %d - %d = %d\n", op1, op2, result);

data[0] = result;

break;

default:

printf("Unknown instruction!\n");

return 1;

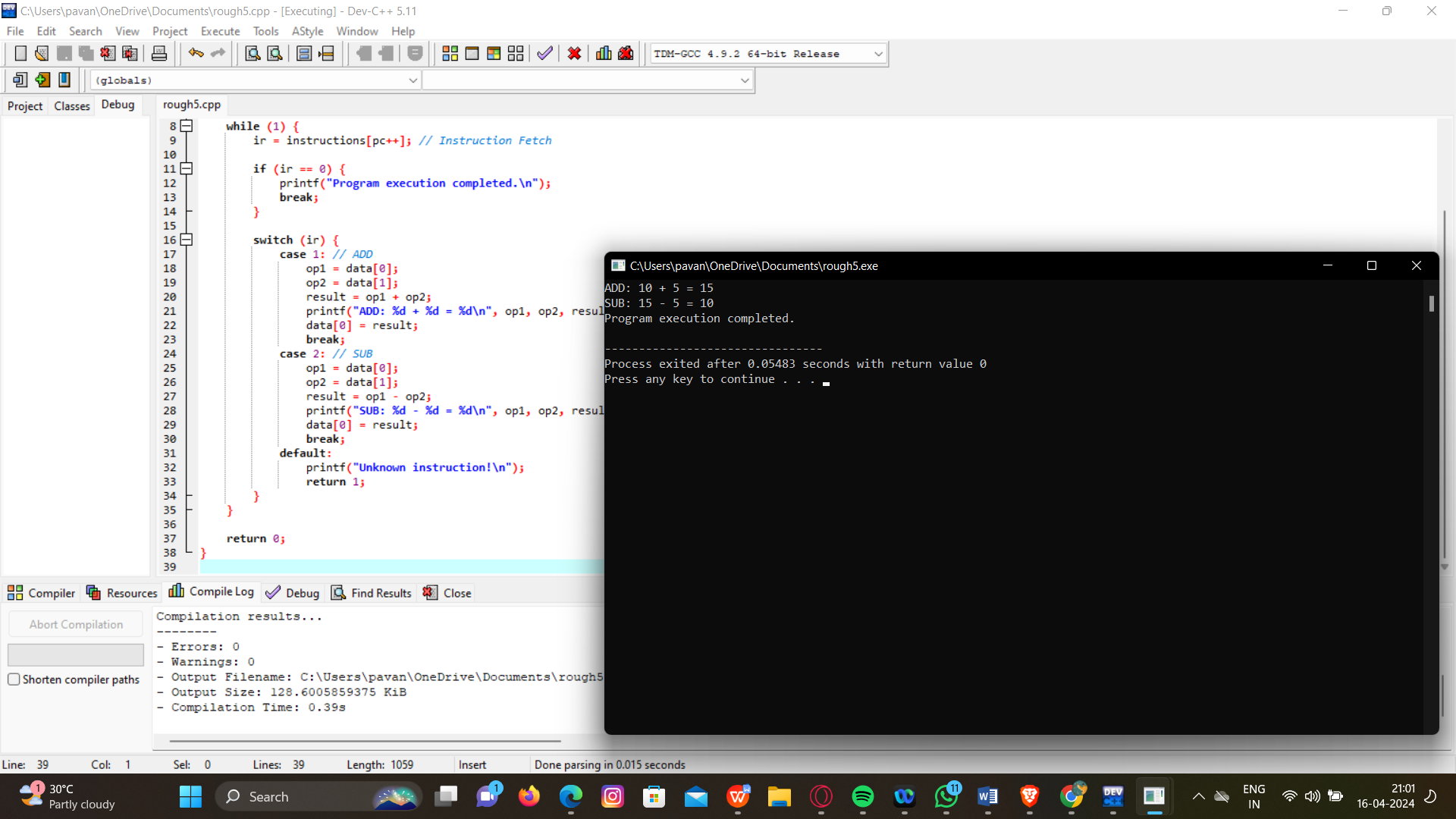
}

}

return 0;

}

OUTPUT:



1. Two Stage And Operation

CODE:

#include <stdio.h>

int main() {

int instructions[] = {1, 2, 0}; // Instructions: 1=ADD, 2=SUB, 0=END

int data[2] = {10, 5}; // Initial data values

int pc = 0, ir, op1, op2, result;

while (1) {

ir = instructions[pc++]; // Instruction Fetch

if (ir == 0) {

printf("Program execution completed.\n");

break;

}

switch (ir) {

case 1: // ADD

op1 = data[0];

op2 = data[1];

result = op1 + op2;

printf("ADD: %d + %d = %d\n", op1, op2, result);

break;

case 2: // SUB

op1 = data[0];

op2 = data[1];

result = op1 - op2;

printf("SUB: %d - %d = %d\n", op1, op2, result);

break;

default:

printf("Unknown instruction!\n");

return 1;

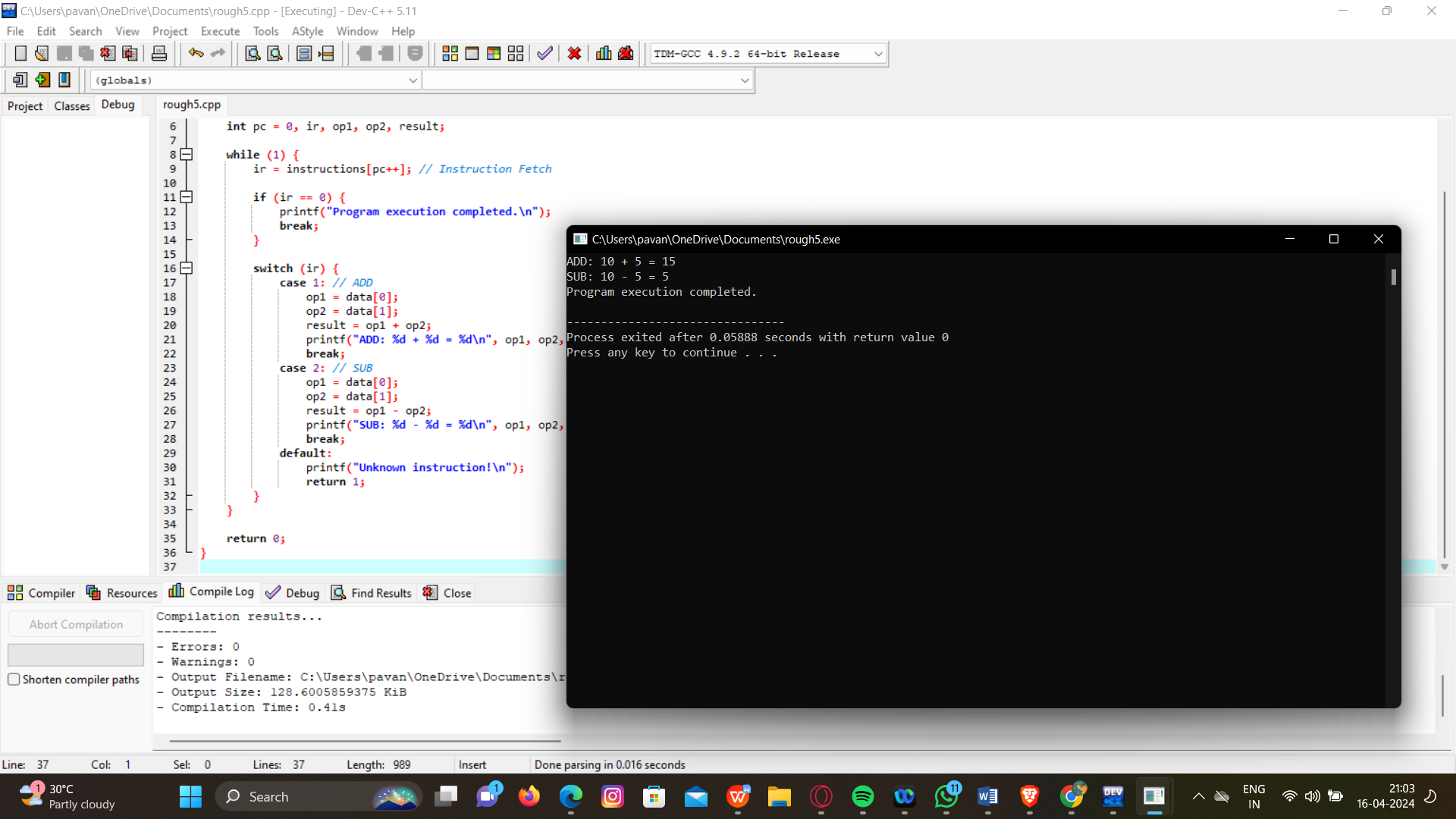
}

}

return 0;

}

OUTPUT:



1. Booth Algorithm

CODE:

#include <stdio.h>

void booth(int \*A, int \*B, int \*result, int size) {

int i, Q[100], BR, AC = 0, QR, temp;

int SC = size \* 2;

QR = B[size - 1];

for (i = size - 2; i >= 0; i--) {

Q[i + 1] = Q[i];

}

Q[0] = 0;

Q[size] = 0;

for (i = 0; i < SC; i++) {

BR = (Q[i] == Q[i + 1]) ? 0 : A[(i + 1) / 2];

if (Q[i] == 1 && Q[i + 1] == 0) {

AC = AC - BR;

} else if (Q[i] == 0 && Q[i + 1] == 1) {

AC = AC + BR;

}

temp = AC;

AC = AC >> 1;

if ((i + 1) % 2 == 0) {

AC = AC | (temp & 1) << (size - 1);

}

QR = QR >> 1;

}

\*result = AC;

}

int main() {

int A[5] = {0, 1, 1, 0};

int B[5] = {0, 0, 1, 0};

int result;

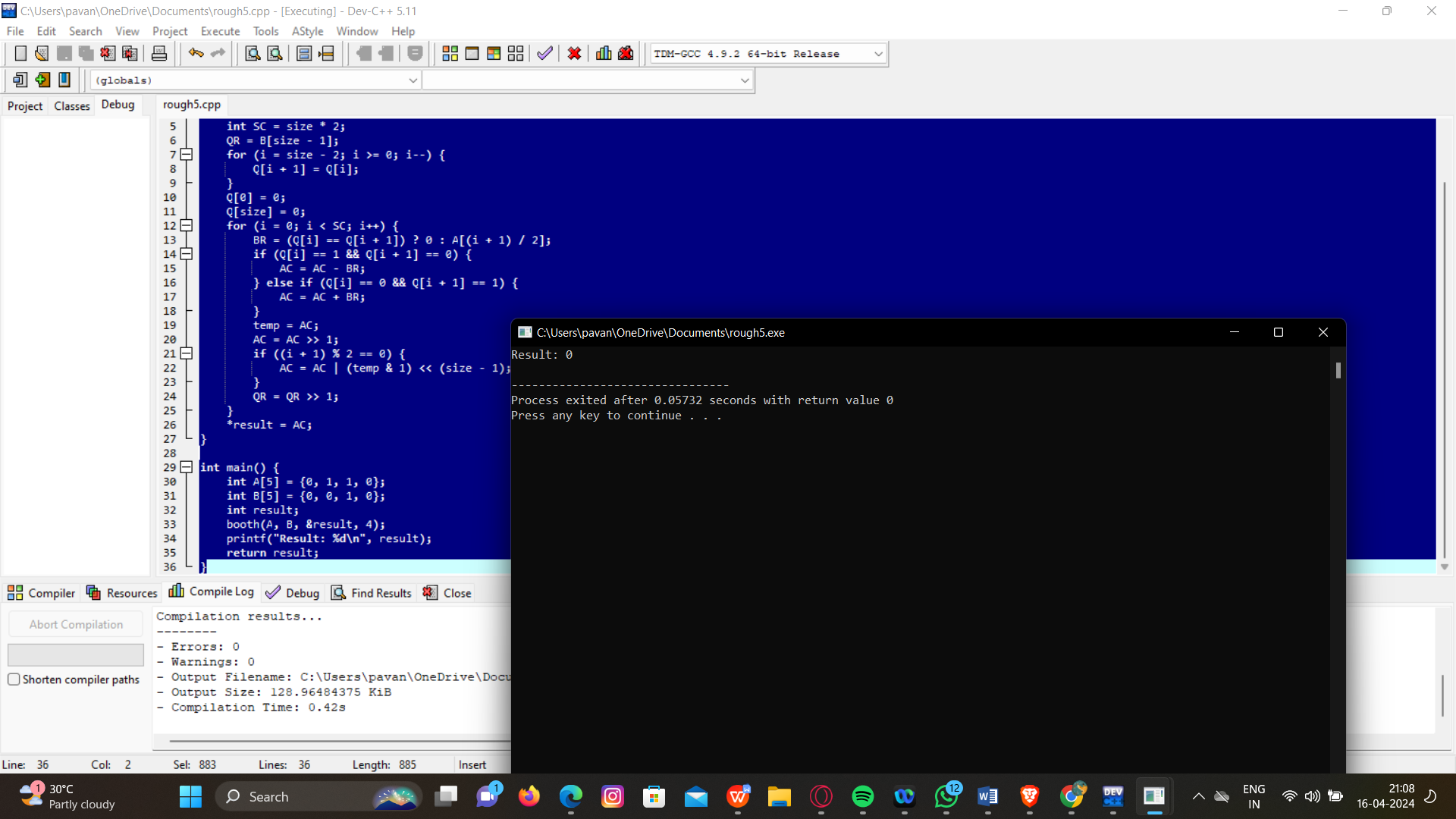
booth(A, B, &result, 4);

printf("Result: %d\n", result);

return result;

}

OUTPUT:



1. Integer Restoring Division

CODE:

#include <stdio.h>

int main() {

int dividend, divisor;

printf("Enter dividend and divisor: ");

scanf("%d %d", &dividend, &divisor);

if (divisor == 0) {

printf("Error: Division by zero\n");

return 1; // Exit program with error status

}

int n = sizeof(int) \* 8; // Number of bits

unsigned int dividend\_u = dividend < 0 ? -dividend : dividend; // Absolute value of dividend

unsigned int divisor\_u = divisor < 0 ? -divisor : divisor; // Absolute value of divisor

// Initialize registers and quotient

unsigned int A = dividend\_u;

unsigned int M = divisor\_u;

unsigned int Q = 0;

int Qn1 = 0; // Q[-1] bit (MSB of quotient)

for (int i = 0; i < n; i++) {

// Left shift A and Q

A <<= 1;

Q <<= 1;

// Set LSB of Q to bit shifted out of A

if (A & (1 << (n + 1))) {

Q |= 1;

}

// Restore if necessary

if (Qn1 == 1 || A >= M) {

A -= M;

Q |= 1;

}

// Update Qn1 for next iteration

Qn1 = Q & 1;

}

// Adjust quotient and remainder signs based on input signs

int quotient = (dividend < 0) ^ (divisor < 0) ? -Q : Q;

int remainder = (dividend < 0) ? -A : A;

printf("Quotient: %d\n", quotient);

printf("Remainder: %d\n", remainder);

return 0;

}

OUTPUT: 