9/28/2016

Mohamad Mahdi Ziaee

109561770

Booth's multiplication

CSCI-246: HW3

Contents

[Description 1](#_Toc462827302)

[Source code 2](#_Toc462827303)

[com.booth.main.MainApplication 2](#_Toc462827304)

[com.booth.utilities.BoothUtils 2](#_Toc462827305)

[com.booth.utilities.Result 7](#_Toc462827306)

[Sample output 8](#_Toc462827307)

[Output 1 8](#_Toc462827308)

[Output 2 9](#_Toc462827309)

# Description

The program is a simulator for the Booth’s 2’s complement number multiplier. In order run the application, execute com.booth.main.MainApplication class. The application will ask for two inputs in binary format and runs the algorithm to generate the product of the two numbers passed in binary format.

Here is the list of frameworks used:

**Language**: JAVA

**Compiler version**: 1.8

**IDE**: Eclipse (Mars 4.5.2)

**Repository**:

<https://github.com/momazia/ComputerArchitecture/tree/Booth_multiplier/BoothNumberMultiplier>

# Source code

## com.booth.main.MainApplication

package com.booth.main;

import java.io.IOException;

import com.booth.utilities.BoothUtils;

/\*\*

\* The main application to be executed in order to run the simulator.

\*

\* @author Max

\*

\*/

public class MainApplication {

public static void main(String[] args) throws IOException {

// Calling the main method driver to start the simulation

BoothUtils.getInstance().driver();

}

}

## com.booth.utilities.BoothUtils

package com.booth.utilities;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.math.BigInteger;

/\*\*

\* Main utility class for Booth algorithm, using Singleton design pattern.

\*

\* @author Max

\*

\*/

public class BoothUtils {

/\*\*

\* Constants

\*/

private static final short MAX\_BIT = 16;

private static final short ASCII\_shortEGER\_INDEX = 48;

private static final short ONE\_BIT\_MASK = 1;

private static BoothUtils instance;

/\*\*

\* Private constructor to avoid initializing the utility class

\*/

private BoothUtils() {

}

/\*\*

\* Returns an instance of this utility class

\*

\* @return

\*/

public static BoothUtils getInstance() {

if (instance == null) {

instance = new BoothUtils();

}

return instance;

}

/\*\*

\* The main method which starts the simulation by getting two numbers from

\* user and runs booth multiplier.

\*

\* @throws IOException

\*/

public void driver() throws IOException {

// Getting user's inputs in binary format (string)

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Enter number 1 (binary): ");

String firstNum = br.readLine();

System.out.println("Enter number 2 (binary): ");

String secondNum = br.readLine();

// Converting the binary strings into short type

int product = boothMultiplier((short) Integer.parseInt(firstNum, 2), (short) Integer.parseInt(secondNum, 2));

System.out.println("Result: " + Integer.toBinaryString(product));

}

/\*\*

\* 1 bit ALU sums/subtracts the two given bits based on the operation flag

\* and carry in bit.

\*

\* @param firstBit

\* @param secondBit

\* @param operation

\* 0 for add, 1 for subtraction

\* @param carryIn

\* @return

\*/

public Result ALU1Bit(final short firstBit, final short secondBit, final int operation, final int carryIn) {

short newSecondBit = secondBit;

if (operation == 1) {

newSecondBit = negate(secondBit);

}

Result result = new Result();

result.setValue((short) (firstBit ^ newSecondBit ^ carryIn));

result.setCarryOut((short) ((firstBit & newSecondBit) + (carryIn & (firstBit ^ newSecondBit))));

return result;

}

/\*\*

\* 16 bit ALU sums/subtracts the two 16 bit values based on the operation

\* flag passed. Throws IllegalArgumentException if an overflow happens.

\*

\* @param value1

\* @param value2

\* @param operation

\* 0 for add, 1 for subtraction

\* @return

\*/

public short ALU16Bit(final short value1, final short value2, final int operation) {

int carryIn = operation;

char[] finalResult = new char[MAX\_BIT];

// Going through the bits in both values, starting for 0 bit and adding

// them together.

for (short i = 0; i < MAX\_BIT; i++) {

short value1Bit = getBitAt(value1, i);

short value2Bit = getBitAt(value2, i);

Result result = ALU1Bit(value1Bit, value2Bit, operation, carryIn);

carryIn = result.getCarryOut();

finalResult[MAX\_BIT - i - 1] = (char) (result.getValue() + ASCII\_shortEGER\_INDEX);

}

short finalResultshort = (short) Integer.parseInt(new String(finalResult), 2);

// Checking the end result to see if an overflow happened

if (overflow(value1, value2, finalResultshort, operation)) {

throw new IllegalArgumentException("Bad input entry, overflow happened!");

}

return finalResultshort;

}

/\*\*

\* Runs Booth's multiplier algorithm for the two values passed. It also

\* prints every step into the system console.

\*

\* @param multiplier

\* @param multiplicand

\* @return

\*/

public int boothMultiplier(short multiplier, short multiplicand) {

short ac = 0, cycleCounter = MAX\_BIT;

short md = multiplicand;

short mq = multiplier;

short dj = 0;

System.out.println("Cycle-counter\tMD\t\t\t\tAC\t\t\t\tMQ\t\t\tMQ\_1");

while (cycleCounter > 0) {

short dj1 = getBitAt(mq, 0);

printLine(ac, cycleCounter, md, mq, dj);

if (dj == 1 && dj1 == 0) {

// Addition

ac = ALU16Bit(ac, md, (short) 0); // 0 means add

printLine(ac, cycleCounter, md, mq, dj);

} else if (dj == 0 && dj1 == 1) {

// Subtraction

ac = ALU16Bit(ac, md, (short) 1); // 1 means sub

printLine(ac, cycleCounter, md, mq, dj);

}

short ac\_0 = getBitAt(ac, 0);

ac = (short) (ac >> 1);

mq = (short) (mq >> 1);

if (ac\_0 == 1) {

mq = (short) (mq | (short) 0x8000); // Changing the sign bit to

// one.

} else {

mq = (short) (mq & (short) 0x7FFF); // Changing the sign bit to

// zero.

}

dj = dj1;

printLine(ac, cycleCounter, md, mq, dj);

cycleCounter--;

}

// Combining the two values AC and MQ.

return new BigInteger(Integer.toBinaryString(0xFFFF & ac) + Integer.toBinaryString(0xFFFF & mq), 2).intValue();

}

/\*\*

\* Prints the parameters passed in a tabular format.

\*

\* @param ac

\* @param cycleCounter

\* @param md

\* @param mq

\* @param dj1

\*/

private void printLine(short ac, short cycleCounter, short md, short mq, short dj1) {

String output = format(cycleCounter, 5);

output += "\t\t" + format(md, 16);

output += "\t\t" + format(ac, 16);

output += "\t\t" + format(mq, 16);

output += "\t" + dj1;

System.out.println(output);

}

/\*\*

\* Formats a given short value in binary for the number of bits passed.

\*

\* @param value

\* @param bits

\* @return

\*/

private String format(short value, int bits) {

return String.format("%" + bits + "s", Integer.toBinaryString(0xFFFF & value)).replace(' ', '0');

}

/\*\*

\* Determines if the value passed is positive, negative or zero.

\*

\* @param value

\* 1 for +ve, -1 for -ve and zero for zero.

\* @return

\*/

public short sign(final short value) {

if (value == 0)

return 0;

if (value >> 15 != 0)

return -1;

return +1;

}

/\*\*

\* Checks the values and the final result based on the operation flag passed

\* to see if an overflow happened.

\*

\* @param value1

\* @param value2

\* @param result

\* @param operation

\* @return

\*/

public boolean overflow(final short value1, final short value2, final short result, final int operation) {

short value1Sign = sign(value1);

short value2Sign = sign(value2);

short resultSign = sign(result);

// If operation was addition

if (operation == 0) {

return (value1Sign > 0 && value2Sign > 0 && resultSign < 0)

|| (value1Sign < 0 && value2Sign < 0 && resultSign > 0);

}

// If operation was subtraction

return (value1Sign > 0 && value2Sign < 0 && resultSign < 0)

|| (value1Sign < 0 && value2Sign > 0 && resultSign > 0);

}

/\*\*

\* Returns the bit at the index passed for the value.

\*

\* @param value

\* @param index

\* @return

\*/

public short getBitAt(final short value, final int index) {

return (short) (value >> index & ONE\_BIT\_MASK);

}

/\*\*

\* Negates the value passed.

\*

\* @param value

\* @return

\*/

public short negate(final short value) {

return (short) (~value & ONE\_BIT\_MASK);

}

}

## com.booth.utilities.Result

package com.booth.utilities;

/\*\*

\* The main POJO which holds the result of the multiplication.

\*

\* @author Max

\*

\*/

public class Result {

private short value;

private short carryOut;

/\*\*

\* Constructor which sets the parameters.

\*

\* @param value

\* @param carryOut

\*/

public Result(short value, short carryOut) {

super();

this.value = value;

this.carryOut = carryOut;

}

/\*\*

\* Default constructor.

\*/

public Result() {

}

public short getValue() {

return value;

}

public void setValue(short value) {

this.value = value;

}

public short getCarryOut() {

return carryOut;

}

public void setCarryOut(short carryOut) {

this.carryOut = carryOut;

}

}

# Sample output

## Output 1

Enter number 1 (binary):

0000000000001001

Enter number 2 (binary):

1111111111110101

Cycle-counter MD AC MQ MQ\_1

10000 1111111111110101 0000000000000000 0000000000001001 0

10000 1111111111110101 0000000000001011 0000000000001001 0

10000 1111111111110101 0000000000000101 1000000000000100 1

01111 1111111111110101 0000000000000101 1000000000000100 1

01111 1111111111110101 1111111111111010 1000000000000100 1

01111 1111111111110101 1111111111111101 0100000000000010 0

01110 1111111111110101 1111111111111101 0100000000000010 0

01110 1111111111110101 1111111111111110 1010000000000001 0

01101 1111111111110101 1111111111111110 1010000000000001 0

01101 1111111111110101 0000000000001001 1010000000000001 0

01101 1111111111110101 0000000000000100 1101000000000000 1

01100 1111111111110101 0000000000000100 1101000000000000 1

01100 1111111111110101 1111111111111001 1101000000000000 1

01100 1111111111110101 1111111111111100 1110100000000000 0

01011 1111111111110101 1111111111111100 1110100000000000 0

01011 1111111111110101 1111111111111110 0111010000000000 0

01010 1111111111110101 1111111111111110 0111010000000000 0

01010 1111111111110101 1111111111111111 0011101000000000 0

01001 1111111111110101 1111111111111111 0011101000000000 0

01001 1111111111110101 1111111111111111 1001110100000000 0

01000 1111111111110101 1111111111111111 1001110100000000 0

01000 1111111111110101 1111111111111111 1100111010000000 0

00111 1111111111110101 1111111111111111 1100111010000000 0

00111 1111111111110101 1111111111111111 1110011101000000 0

00110 1111111111110101 1111111111111111 1110011101000000 0

00110 1111111111110101 1111111111111111 1111001110100000 0

00101 1111111111110101 1111111111111111 1111001110100000 0

00101 1111111111110101 1111111111111111 1111100111010000 0

00100 1111111111110101 1111111111111111 1111100111010000 0

00100 1111111111110101 1111111111111111 1111110011101000 0

00011 1111111111110101 1111111111111111 1111110011101000 0

00011 1111111111110101 1111111111111111 1111111001110100 0

00010 1111111111110101 1111111111111111 1111111001110100 0

00010 1111111111110101 1111111111111111 1111111100111010 0

00001 1111111111110101 1111111111111111 1111111100111010 0

00001 1111111111110101 1111111111111111 1111111110011101 0

Result: 11111111111111111111111110011101

## Output 2

Enter number 1 (binary):

1111111111110101

Enter number 2 (binary):

0000000000001001

Cycle-counter MD AC MQ MQ\_1

10000 0000000000001001 0000000000000000 1111111111110101 0

10000 0000000000001001 1111111111110111 1111111111110101 0

10000 0000000000001001 1111111111111011 1111111111111010 1

01111 0000000000001001 1111111111111011 1111111111111010 1

01111 0000000000001001 0000000000000100 1111111111111010 1

01111 0000000000001001 0000000000000010 0111111111111101 0

01110 0000000000001001 0000000000000010 0111111111111101 0

01110 0000000000001001 1111111111111001 0111111111111101 0

01110 0000000000001001 1111111111111100 1011111111111110 1

01101 0000000000001001 1111111111111100 1011111111111110 1

01101 0000000000001001 0000000000000101 1011111111111110 1

01101 0000000000001001 0000000000000010 1101111111111111 0

01100 0000000000001001 0000000000000010 1101111111111111 0

01100 0000000000001001 1111111111111001 1101111111111111 0

01100 0000000000001001 1111111111111100 1110111111111111 1

01011 0000000000001001 1111111111111100 1110111111111111 1

01011 0000000000001001 1111111111111110 0111011111111111 1

01010 0000000000001001 1111111111111110 0111011111111111 1

01010 0000000000001001 1111111111111111 0011101111111111 1

01001 0000000000001001 1111111111111111 0011101111111111 1

01001 0000000000001001 1111111111111111 1001110111111111 1

01000 0000000000001001 1111111111111111 1001110111111111 1

01000 0000000000001001 1111111111111111 1100111011111111 1

00111 0000000000001001 1111111111111111 1100111011111111 1

00111 0000000000001001 1111111111111111 1110011101111111 1

00110 0000000000001001 1111111111111111 1110011101111111 1

00110 0000000000001001 1111111111111111 1111001110111111 1

00101 0000000000001001 1111111111111111 1111001110111111 1

00101 0000000000001001 1111111111111111 1111100111011111 1

00100 0000000000001001 1111111111111111 1111100111011111 1

00100 0000000000001001 1111111111111111 1111110011101111 1

00011 0000000000001001 1111111111111111 1111110011101111 1

00011 0000000000001001 1111111111111111 1111111001110111 1

00010 0000000000001001 1111111111111111 1111111001110111 1

00010 0000000000001001 1111111111111111 1111111100111011 1

00001 0000000000001001 1111111111111111 1111111100111011 1

00001 0000000000001001 1111111111111111 1111111110011101 1

Result: 11111111111111111111111110011101