VIET NAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER ARCHITECTURE

Assignment Report

Multiplication of two 32-bit integers

Supervisor: Huỳnh Phúc Nghị
Group: 10 - L10
Members: Nguyễn Lâm - 2311822
Phạm Duy Quý - 2312900
Nguyễn Võ Đức Sơn - 2312974

Ho Chi Minh City, 11/2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

Topic: Implement the program which performs the multiplication algorithm between two integers in the textbook (pic.3.4 or pic.3.5), applied for signed numbers. The input data is read from binary file INT2.BIN.

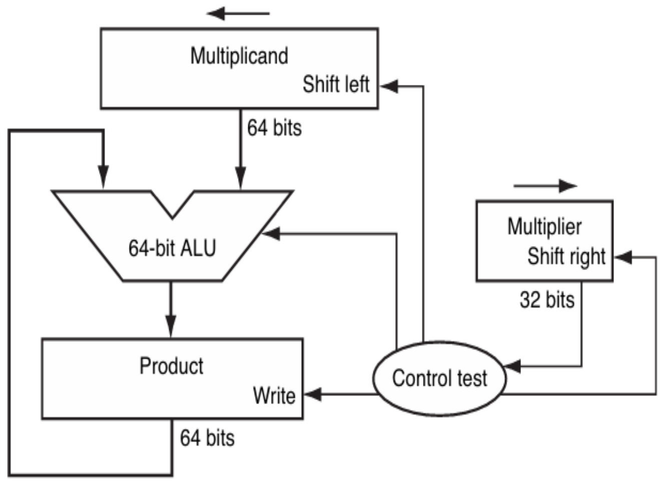

Figure 1: Multiplication hardware (Figure 3.4 from Textbook).

In multiplication, the first operand called *multiplicand* while the second one is called *multiplier*, and the result of this calculation is also known as *product*. Obviously, multiplicating of binary numbers is absolutely similar to what we do with decimal numbers. Recalling what we have learned in school, we need to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier, and shifting the intermediate product one digit to the left of the earlier intermediate products.

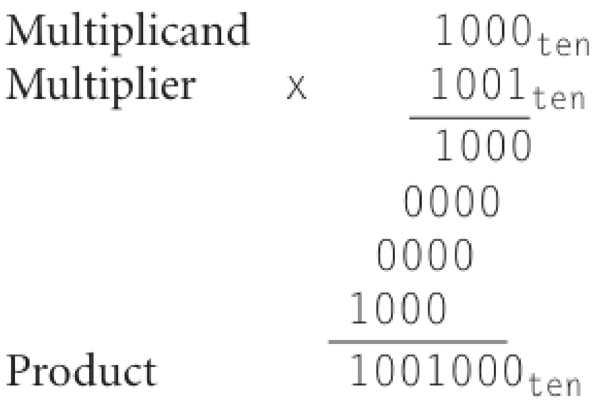

Figure 2: Multiplication example (from Textbook).

2 Ideas and Implementation

2.1 Ideas for implementation

Whenever the result of the calculation if a 32-bit integer, it is can be easy for us to apply the algorithm mentioned above. However, in reality, there are lots of situation which has a 64-bit integer result. Unfortunately, MIPS just support us with 32-bit register, so we have to use two 32-bit register to illustrate the result, with the first register save 32 upper bits and the second keep the remain bits. The entire process is conducted as below:

- We just simply do like what have been introduced in the algorithm, choose the rightmost bit of the multiplier, conduct the *and* operation with the multiplicand, save the result and then shift the chosen to its next left one. The above process is done continuously until there is no chosen bit left from the multiplier.
- We got a stair-shape sum which needs to be computed to get the final result, this is the stage where 64-bit number may be produced. And here is the solution:
  - The temporary *product* is increased whenever there is a new LSB bit of the *multiplier* perform the *and* operation with the *multiplicand*.
  - In case of producing 64-bit number, the result of the above sum considered in 32 low bit is smaller than both of its operand. Simultaneously, a *carry bit* is also created in this calculation.
  - If there is a signal which tells us that the *carry bit* exists, we add it to higher 32-bit register.

– The process goes on with a new LSB from the *multiplier*.

2.2 Implementation

Implementation is the source code sent attached with this report.

3 Statistical indexes about the source code

3.1 Test case

There are 15 test cases that we have already created, and the expected results are shown in the table below:

Table 1: Test cases and expected results.

TC	Multiplicand	Multiplier	Result (Decimal)	Result (Hexadecimal)
1	-2147483648	1	-2147483648	FFFFFFFF 80000000
2	2147483647	1	2147483647	00000000 7FFFFFFF
3	-2147483648	-1	2147483648	00000000 80000000
4	-2147483648	0	0	00000000 00000000
5	-1	-1	1	00000000 00000001
6	1	-1	-1	11111111 11111111
7	-32	-12	384	00000000 00000180
8	-56789	34567	-1963025363	FFFFFFFF 8AFE9C2D
9	100000	10000	1000000000	00000000 3B9ACA00
10	0	-12345	0	00000000 00000000
11	12125215	283953	3442991174895	00000321 A24414EF
12	985725	61767217	60885489977325	00003760 020087ED
13	-98754321	-98754321	-1.219190134472169 e+16	FFD4AF87 C33614E6
14	-25582352	123856102	-3168530398711904	FFF4BE3D 071FEFA0
15	0	0	0	00000000 00000000

3.2 Execution Result

The program output for each test case is shown at the pictures below:

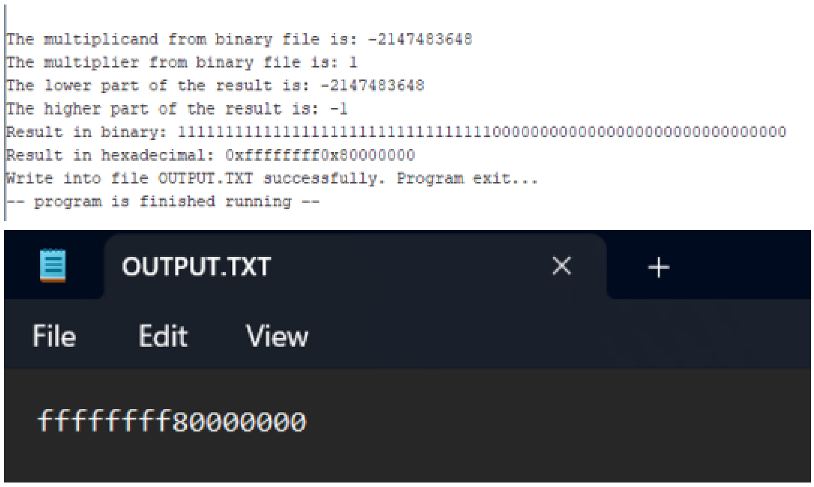

Figure 3: Testcase 1

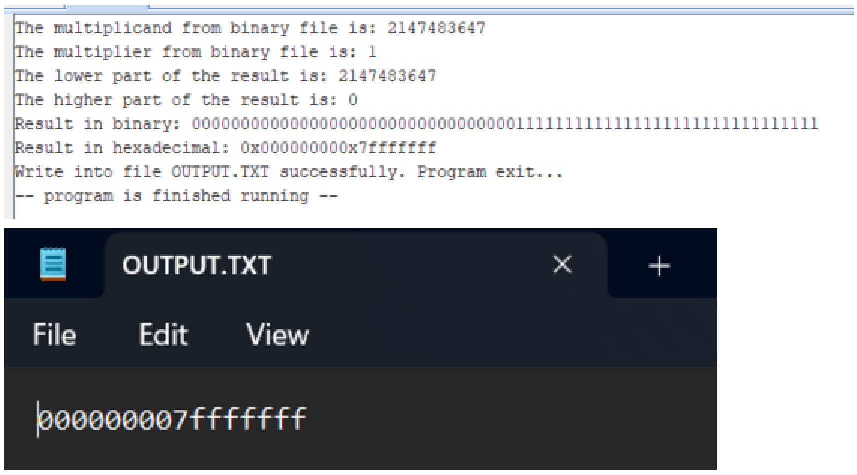

Figure 4: Testcase 2

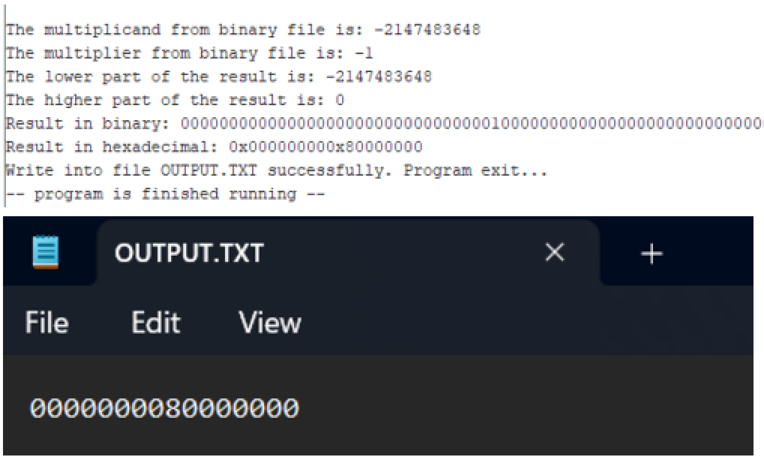

Figure 5: Testcase 3

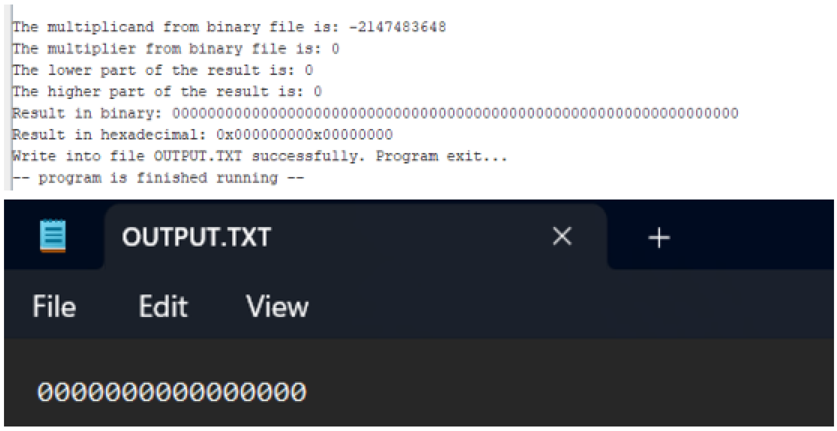

Figure 6: Testcase 4

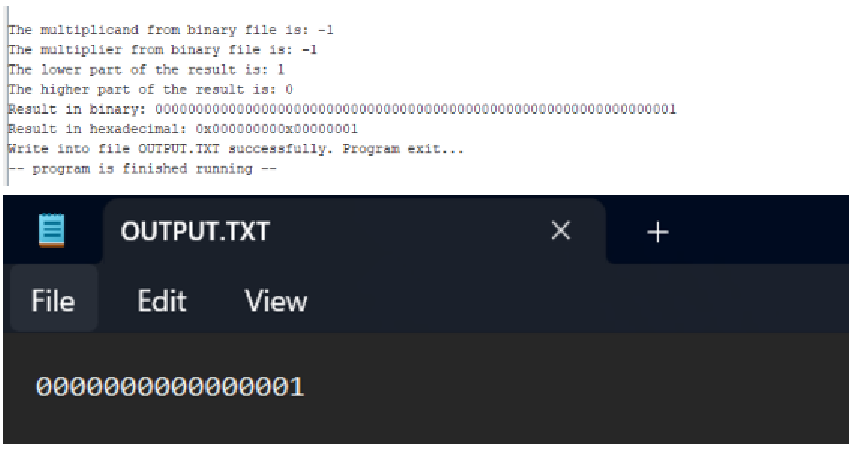

Figure 7: Testcase 5

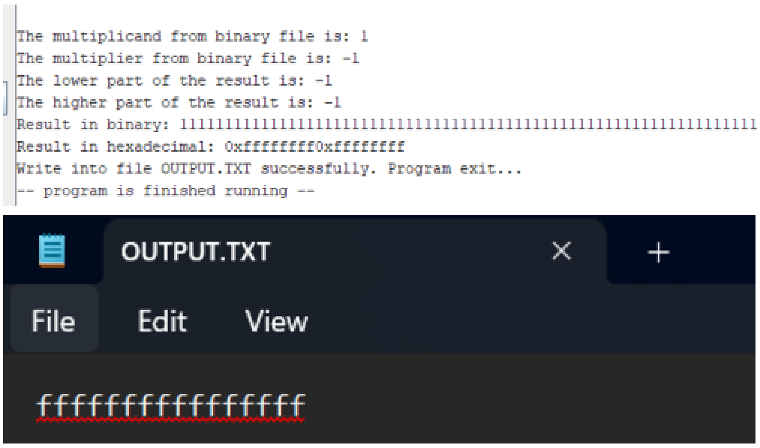

Figure 8: Testcase 6

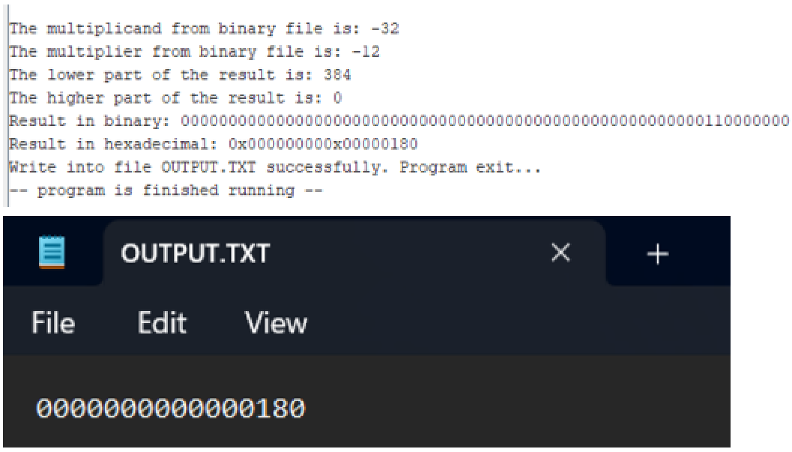

Figure 9: Testcase 7

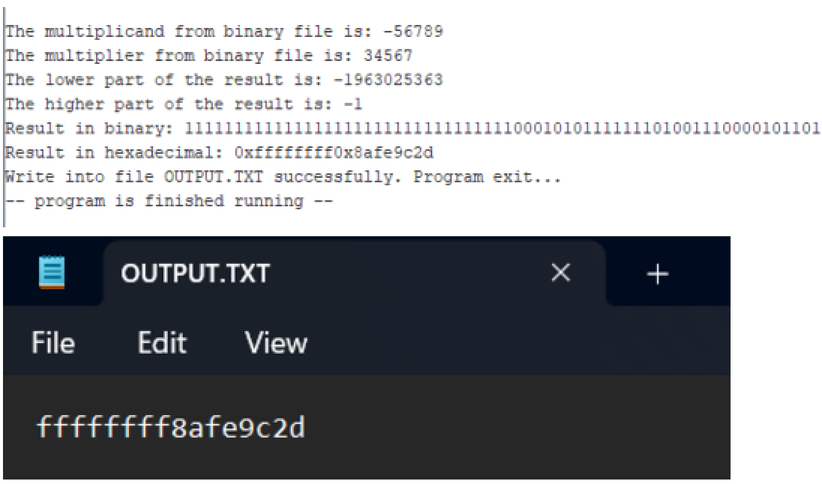

Figure 10: Testcase 8

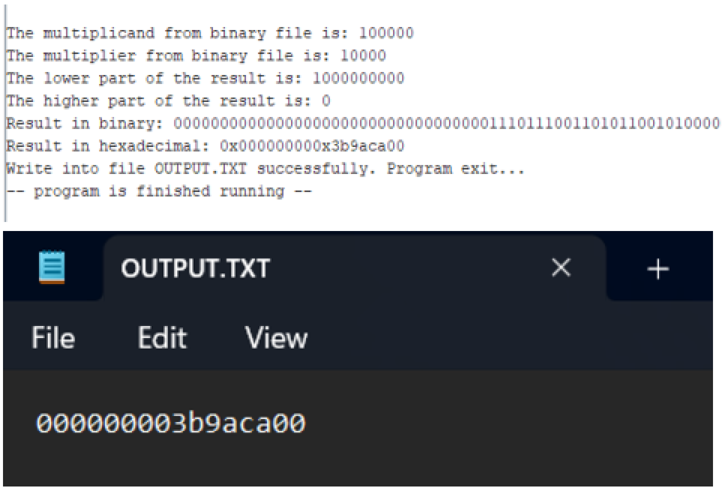

Figure 11: Testcase 9

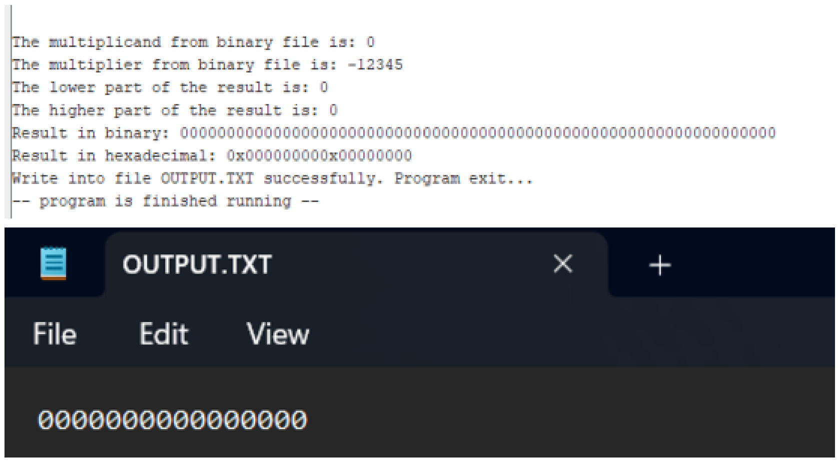

Figure 12: Testcase 10

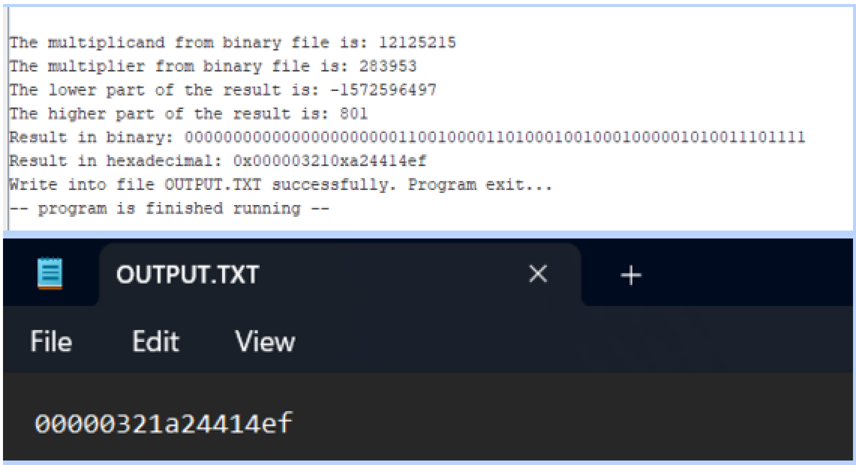

Figure 13: Testcase 11

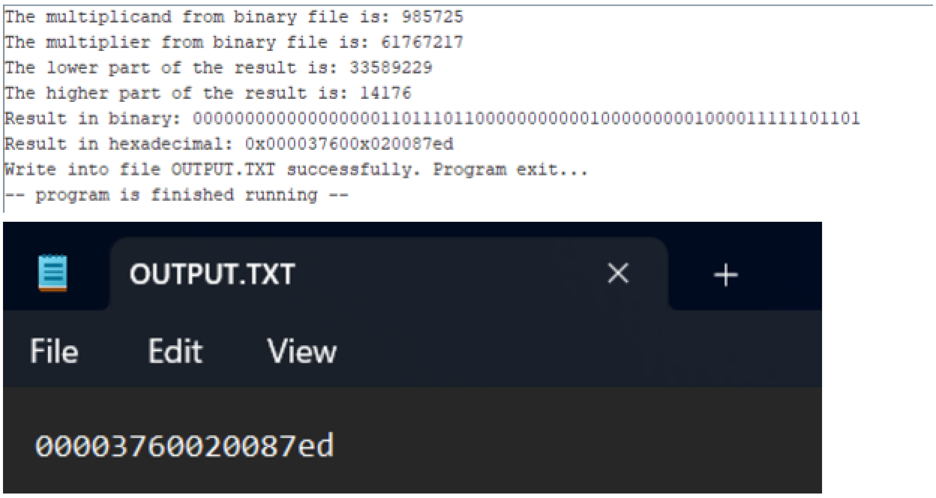

Figure 14: Testcase 12

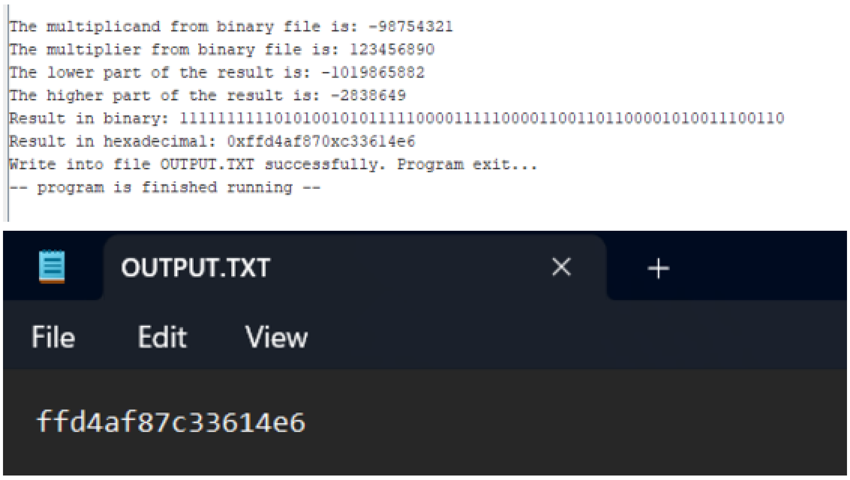

Figure 15: Testcase 13

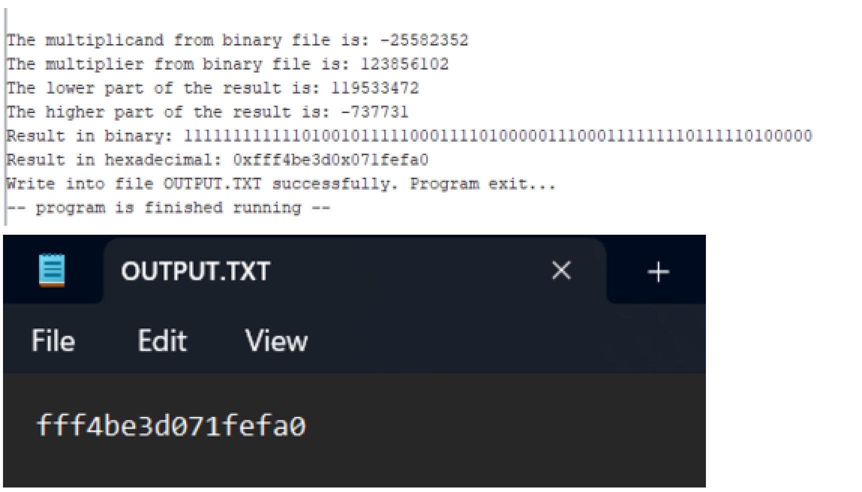

Figure 16: Testcase 14

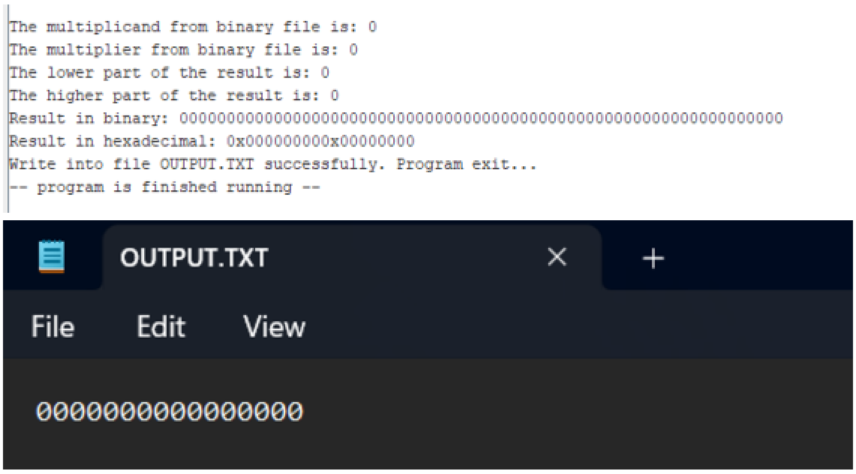

Figure 17: Testcase 15

3.3 Instruction type count

For each test case, the total number of executed instructions based on each instruction type is presented in the table below:

Table 2: Instruction type count

Test Case	R-type	I-type	J-type	Total
1	252	487	35	774
2	247	485	33	765
3	249	487	26	762
4	81	259	22	362
5	249	487	26	762
6	252	487	43	782
7	253	487	27	767
8	276	487	46	809
9	263	485	34	782
10	81	257	22	360
11	275	485	36	796
12	275	485	36	796
13	320	487	51	858
14	308	487	51	846
15	81	357	22	360

4 Execution Time

4.1 Clock Cycle Time for each Clock Cycle System

For the Single Clock Cycle system, all steps of an instruction are executed in one single clock cycle time, so the clock cycle time of this system must be the instruction with slowest execution time, in order to make sure all instructions can be done before the clock edge triggers. This is also called the *critical path delay*. With the given clock rate of 1GHz, we assume that this is also the clock rate for the system, and the clock cycle time is T = 1ns. So, the execution time is calculated as below:

$$\text{Execution Time} = \text{Instruction Count} \cdot \text{T}.$$

For the Multiple Clock Cycle system, each instruction is divided into five steps, with each step completing in one clock cycle. So the clock cycle time is determined by the slowest step. As the assignment requirement does not include any information about delay time for each step, we assume that all steps have an equal delay time, which is also the clock cycle time, since a cycle in multiple clock cycle just perform a step only, it is much faster than what we have in Single clock cycle, assume that :

$$T = \frac{1}{5} = 0.2 \text{ (ns)}.$$

And the execution time:

$$\text{Execution Time} = (\text{Load} \cdot 5 + \text{Add} \cdot 4 \\ + \text{Jump} \cdot 2 + \text{Store} \cdot 4 + \text{Branch} \cdot 3.)$$

For the Pipeline system, the execution of multiple instructions is overlapped across five stages (Instruction fetch, Instruction decode/register file read, Execution/address calculation, Memory access, Write-back), and the clock cycle time is determined by the slowest stage (as all stages are completed in one single clock cycle). As the assignment requirement does not include any information about delay time for each step, we assume that all stages have an equal delay time, which is also the clock cycle time:

$$T = \frac{1}{5} = 0.2 \text{ (ns)}.$$

And the execution time:

$$\text{Execution Time} = (4 + \text{Instruction Count}) \cdot \text{T}.$$

4.2 Instruction distribution for each test case

For each test case, the total number of executed instructions based on five types of basic instruction is presented in the table below:

Table 3: Instruction Distribution

TC	Load	Add	Jump	Store	Branch	Other	Total
1	2	550	37	17	116	52	774
2	2	545	35	17	116	50	765
3	2	549	28	17	116	50	762
4	2	226	24	17	49	44	362
5	2	549	28	17	116	50	762
6	2	550	45	17	116	52	782
7	2	549	29	17	116	54	767
8	2	550	48	17	116	76	809
9	2	545	36	17	116	66	782
10	2	225	24	17	48	44	360
11	2	545	38	17	116	44	762
12	2	545	45	17	116	110	835
13	2	550	53	17	116	120	858
14	2	550	53	17	116	108	846
15	2	225	24	17	48	44	360

4.3 Execution time and Speedup

The execution time in case of Single Clock Cycle (SCC), Multi Clock Cycle (MCC) and Pipeline systems, and speedup of Pipeline comparing to Single Clock Cycle and Multi Clock Cycle systems are shown at the table below:

Table 4: Execution time and Speedup

TC	SCC (ns)	MCC (ns)	Pipeline (ns)	Speedup w/MCC	Speedup w/SCC
1	774	540	155.6	3.470437	4.974293
2	765	535.2	153.8	3.479844	4.973992
3	762	535.6	153.2	3.496084	4.97389
4	362	235.4	73.2	3.215847	4.945355
5	762	535.6	153.2	3.496084	4.97389
6	782	543.2	157.2	3.455471	4.974555
7	767	536	154.2	3.476005	4.97406
8	809	544.4	162.6	3.348093	4.9754
9	782	535.6	157.2	3.407125	4.974555
10	360	234	72.8	3.214286	4.945055
11	762	536.4	153.2	3.501305	4.97389
12	835	539.2	167.8	3.213349	4.976162
13	858	546.4	172.4	3.169374	4.976798
14	846	546.4	170	3.214118	4.976471
15	360	234	72.8	3.214286	4.945055

5 Summary

We have already completely finished our Computer Architecture assignment with the topic *"Multiplication of two 32-bit integers."*
This report examines the multiplication algorithm of two 32-bit integers in textbook in MIPS assembly language, using MARS MIPS 4.5. The algorithm involves iterative addition and shifting operations. As the product of a multiplication operation is 64-bit, while MIPS only have 32-bit registers for integers, the two registers are connected to store the 32-bit higher and 32-bit lower part of the product.
Additionally, the report not only evaluates the correctness of the program implementing the algorithm through test cases but also examines its performance by measuring the execution time with a given clock frequency. Insights gained from this implementation underscore the significance of understanding foundational al-

gorithms in computer architecture, and how basic operations are implemented in hardware level.

6 References

[1] Patterson, D. A., & Hennessy, J. L. (2011). *Computer Organization and Design: The Hardware/Software Interface (4th edition).*

[2] *Tutorial on MIPS Programming using MARS. (n.d.).* `https://profile.iiita.ac.in/bibhas.ghoshal/COA_2021/tutorials/Tutorial_MIPS_Using_MARS.pdf`