Homework #1: Unsigned & Signed Integer Overflow and Binary Representation using C++ and MIPS Assembly language.

Zi Xuan Li

The Grove School of Engineering, The City College of New York

CSC 34200 E[19155]: Computer Organization

Professor Gertner

February 19th, 2024

Table of Contents

Objective: This assignment aims to deepen our understanding of integer representation, overflow, and arithmetic in C++ and MIPS assembly. Through multiple tasks, we explore signed and unsigned integer behavior involving overflow. For my report specifically, I will be documenting my results using a 64-bit computer and a 64-bit C++ compiler.

64-bit Signed Integers:

Largest Positive 64-bit Signed Integer

```cpp
#include <iostream>
#include <bitset>
#include <limits>

using namespace std;

int main() {
    long long largest_positive = numeric_limits<long long>::max();  // Largest positive 64-bit signed integer
    cout << "Largest positive 64-bit signed integer: " << largest_positive << endl;
    cout << "Binary representation: " << bitset<64>(largest_positive) << endl;

    long long result = largest_positive + 1;    // Adding 1 to observe overflow
    cout << "Result after adding 1: " << result << endl;
    cout << "Binary representation of result: " << bitset<64>(result) << endl;

    return 0;
}
```

Figure 1: C++ code snippet that calculates the largest positive 64-bit signed integer, adds 1 to it, resulting in overflow, and prints the result.

```
Largest positive 64-bit signed integer: 9223372036854775807
Binary representation:
0111111111111111111111111111111111111111111111111111111111111111
Result after adding 1: -9223372036854775808
Binary representation of result:
1000000000000000000000000000000000000000000000000000000000000000
```

Figure 2: Output of the code snippet in figure 1.

Most Negative 64-bit Signed Integer

```cpp
#include <iostream>
#include <bitset>
#include <limits>

using namespace std;

int main() {
    long long most_negative = numeric_limits<long long>::min();  // Most negative
64-bit signed integer
    cout << "Most negative 64-bit signed integer: " << most_negative << endl;
    cout << "Binary representation: " << bitset<64>(most_negative) << endl;

    long long result = most_negative - 1;    // Subtracting 1 to observe overflow
    cout << "Result after subtracting 1: " << result << endl;
    cout << "Binary representation of result: " << bitset<64>(result) << endl;

    return 0;
}
```

Figure 3: C++ code snippet that calculates the most negative 64-bit signed integer, subtracts 1 to it, resulting in overflow, and prints the result.

```
Most negative 64-bit signed integer: -9223372036854775808
Binary representation:
1000000000000000000000000000000000000000000000000000000000000000
Result after subtracting 1: 9223372036854775807
Binary representation of result:
0111111111111111111111111111111111111111111111111111111111111111
```

Figure 4: Output of the code snippet in figure 3.

64-bit Unsigned Integers

Largest Positive 64-bit Unsigned Integer

```cpp
#include <iostream>
#include <bitset>
#include <limits>

using namespace std;

int main() {
    unsigned long long largest_positive = numeric_limits<unsigned long long>::max();    // Largest positive 64-bit unsigned integer
    cout << "Largest positive 64-bit unsigned integer: " << largest_positive << endl;
    cout << "Binary representation: " << bitset<64>(largest_positive) << endl;

    unsigned long long result = largest_positive + 1;    // Adding 1 to observe overflow
    cout << "Result after adding 1: " << result << endl;
    cout << "Binary representation of result: " << bitset<64>(result) << endl;

    return 0;
}
```

Figure 5: C++ code snippet that calculates the largest positive 64-bit unsigned integer, adds 1 to it, resulting in overflow, and prints the result.

```
Largest positive 64-bit unsigned integer: 18446744073709551615
Binary representation:
1111111111111111111111111111111111111111111111111111111111111111
Result after adding 1: 0
Binary representation of result:
0000000000000000000000000000000000000000000000000000000000000000
```

Figure 6: Output of the code snippet in figure 5.

Most Negative 64-bit Unsigned Integer

```cpp
#include <iostream>
#include <bitset>
#include <limits>

using namespace std;

int main() {
    unsigned long long most_negative = numeric_limits<unsigned long long>::min();
// Most negative 64-bit unsigned integer
    cout << "Most negative 64-bit unsigned integer: " << most_negative << endl;
    cout << "Binary representation: " << bitset<64>(most_negative) << endl;

    unsigned long long result = most_negative - 1;    // Subtracting 1 to observe
overflow
    cout << "Result after subtracting 1: " << result << endl;
    cout << "Binary representation of result: " << bitset<64>(result) << endl;

    return 0;
}
```

Figure 7: C++ code snippet that calculates the most negative 64-bit unsigned integer, subtracts 1 to it, resulting in overflow, and prints the result.

```
Most negative 64-bit unsigned integer: 0
Binary representation:
0000000000000000000000000000000000000000000000000000000000000000
Result after subtracting 1: 18446744073709551615
Binary representation of result:
1111111111111111111111111111111111111111111111111111111111111111
```

Figure 8: Output of the code snippet in figure 7.

Integer Overflow and its Consequences

Integer overflow is the result of an arithmetic operation that exceeds the maximum value that can be represented by a data type. In other words when the result of a calculation is too large to be stored in available memory, an overflow occurs. The consequences of overflow include

data corruption, security vulnerabilities, and undefined behavior. For example, if an integer overflow happens during a financial calculation, it may result in a negative account balance to become positive.

In binary representation, integer overflow occurs when an operation exceeds the maximum representable value for the given bits. The sign bit plays a crucial role in determining whether the overflow is positive or negative. For example, when an overflow occurs with signed integers using two's complement representation, the sign bit can change unexpectedly leading to incorrect results.

Situations Where Overflow Can Occur

Overflow in arithmetic operations like addition and subtraction happens when the result exceeds the allocated number of bits. For instance, in a 64-bit signed integer, the range is -9223372036854775808 to 9223372036854775807. If the result surpasses these limits, overflow occurs. Addition overflow occurs when the result of adding two positive numbers is greater than the maximum value representable, or when the result of adding two negative numbers is less than the minimum value representable. Similarly, subtraction overflow occurs when the result of subtracting a large number from a smaller one is less than the minimum value representable by the data type. An example of addition overflow is in figure 2. An example of subtraction overflow is in figure 4. Regarding unsigned integers, overflow occurs only when adding 1 to the largest integer which results in wrapping around to the smallest integer as shown in figure 6.

MIPS

Addition for Signed Integers

Figures 9 & 10 display a runtime exception indicating an arithmetic overflow has occurred. In MIPS assembly, integer overflow does not result in wrap-around behavior by default because the processor does not automatically handle overflow conditions. Unlike in C++, where integer overflow causes the largest positive signed integer to wrap around to the most negative signed integer.

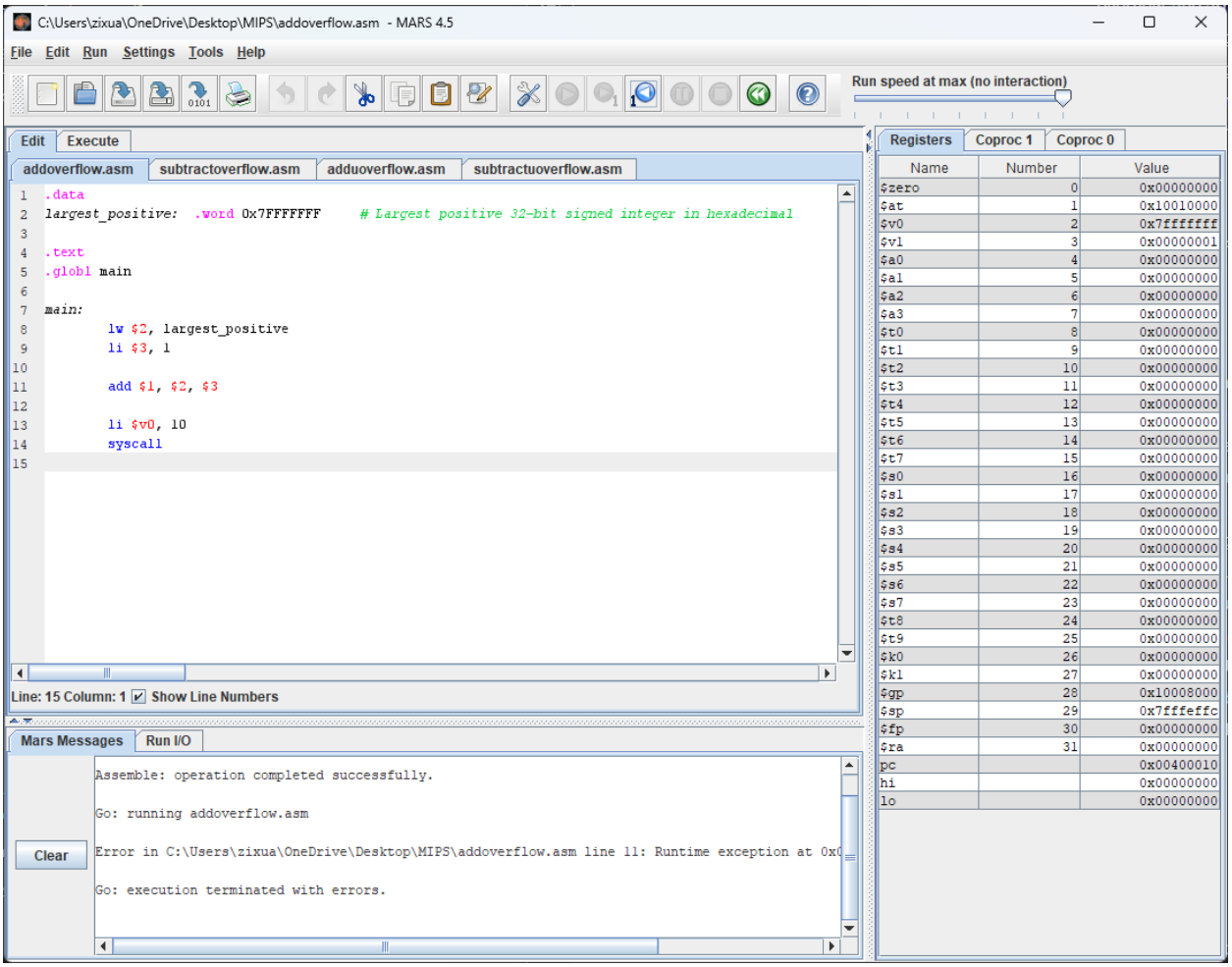

Figure 9: MIPS code snippet to demonstrate overflow by using add on the largest positive 32-bit signed integer.

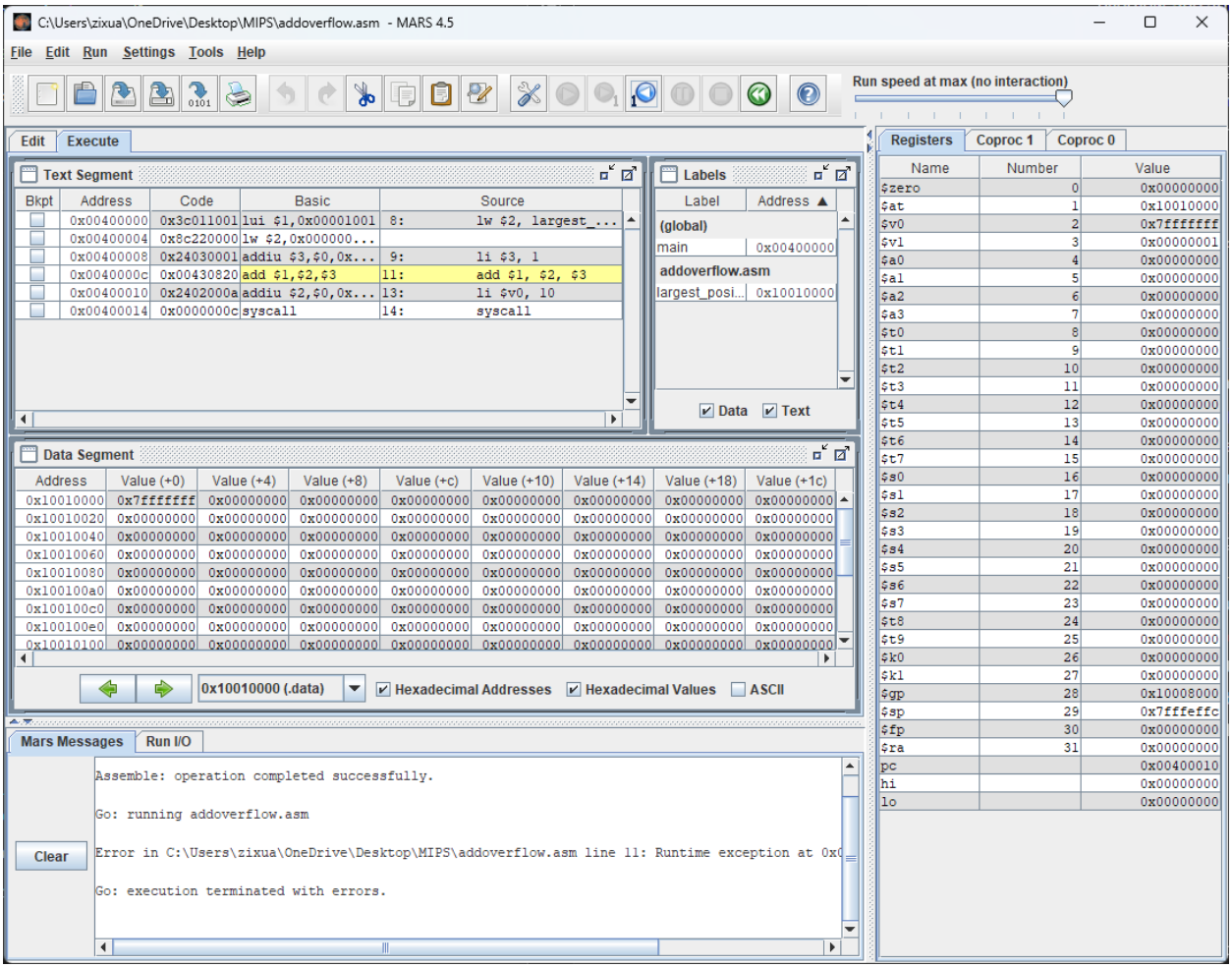

Figure 10: Output of code snippet in figure 9.

Subtraction for Signed Integers

Figures 11 & 12 display a runtime exception indicating an arithmetic overflow has occurred. Again, in MIPS assembly, integer overflow does not result in wrap-around behavior by default because the processor does not automatically handle overflow conditions. In this case, integer overflow would cause the most negative signed integer to wrap around to the most positive signed integer in C++.

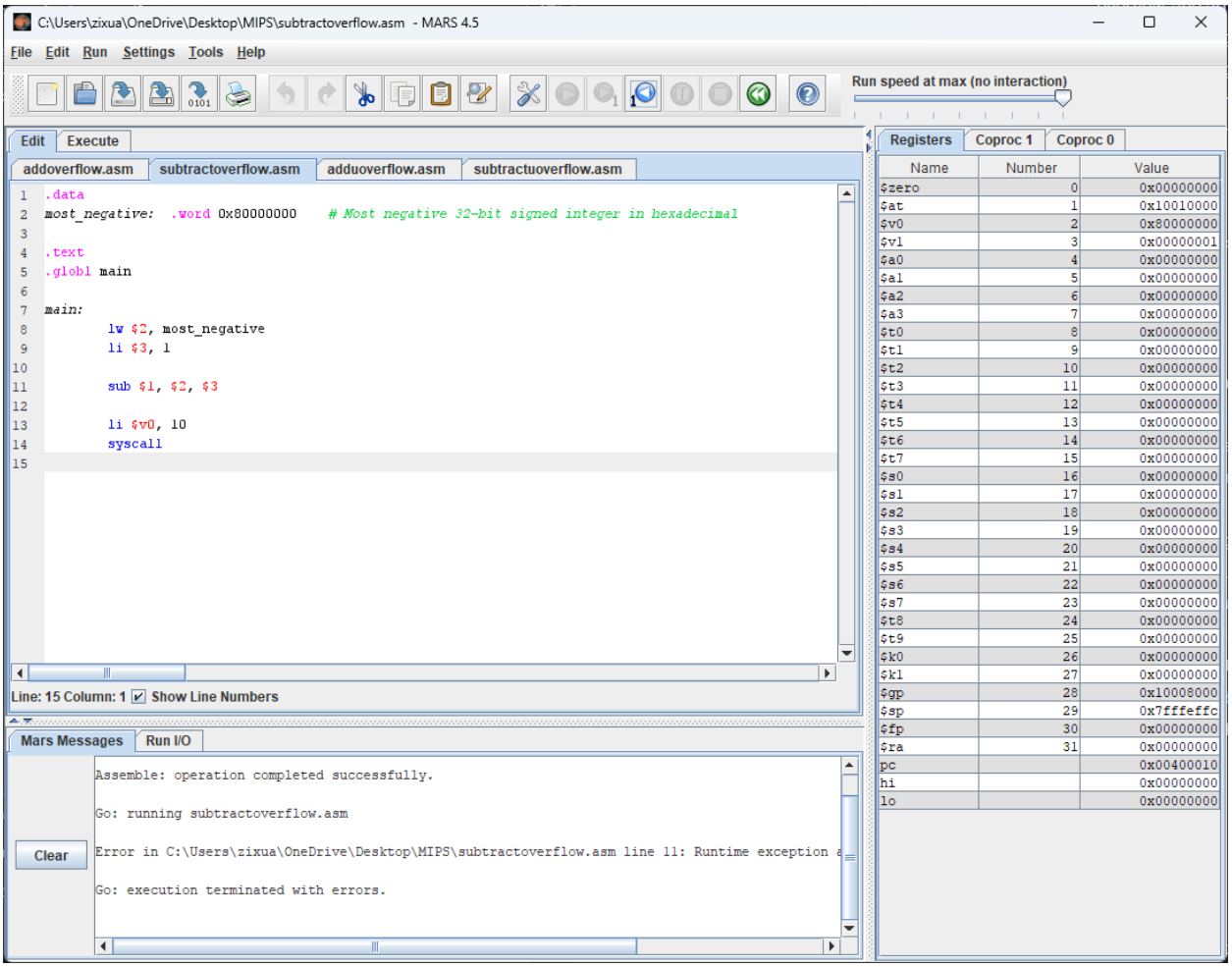

Figure 11: MIPS code snippet to demonstrate overflow by using sub on the most negative 32-bit signed integer.

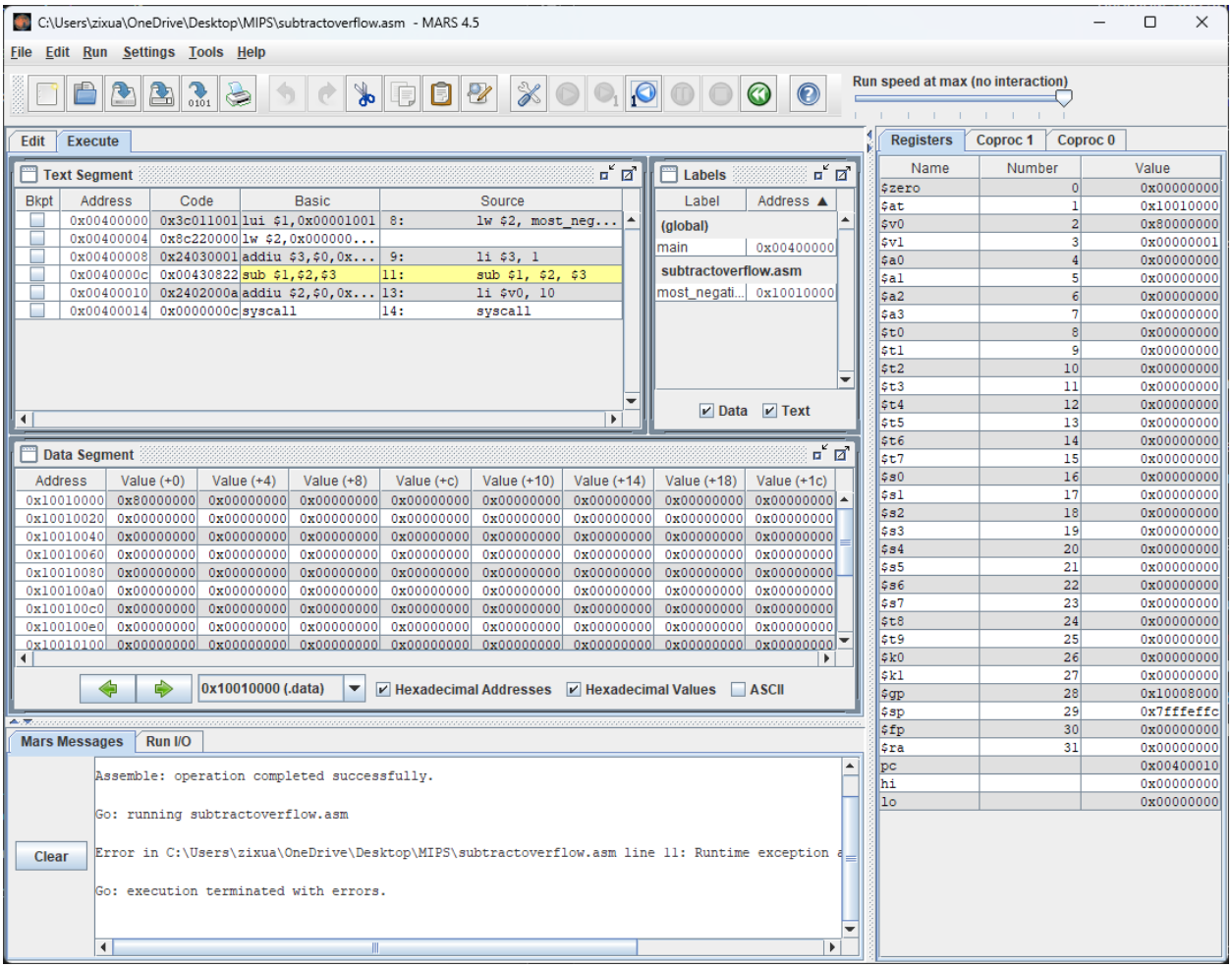

Figure 12: Output of code snippet in figure 11.

Addition for Unsigned Integers

Figures 13 & 14 do not display a run time exception, instead it displays the program is finished running. This is because ADDU treats the operands as unsigned numbers. This can be seen in register $at which is 0x10010000 because of the wraparound that occurred when adding 0x00000001 to 0xffffffff.

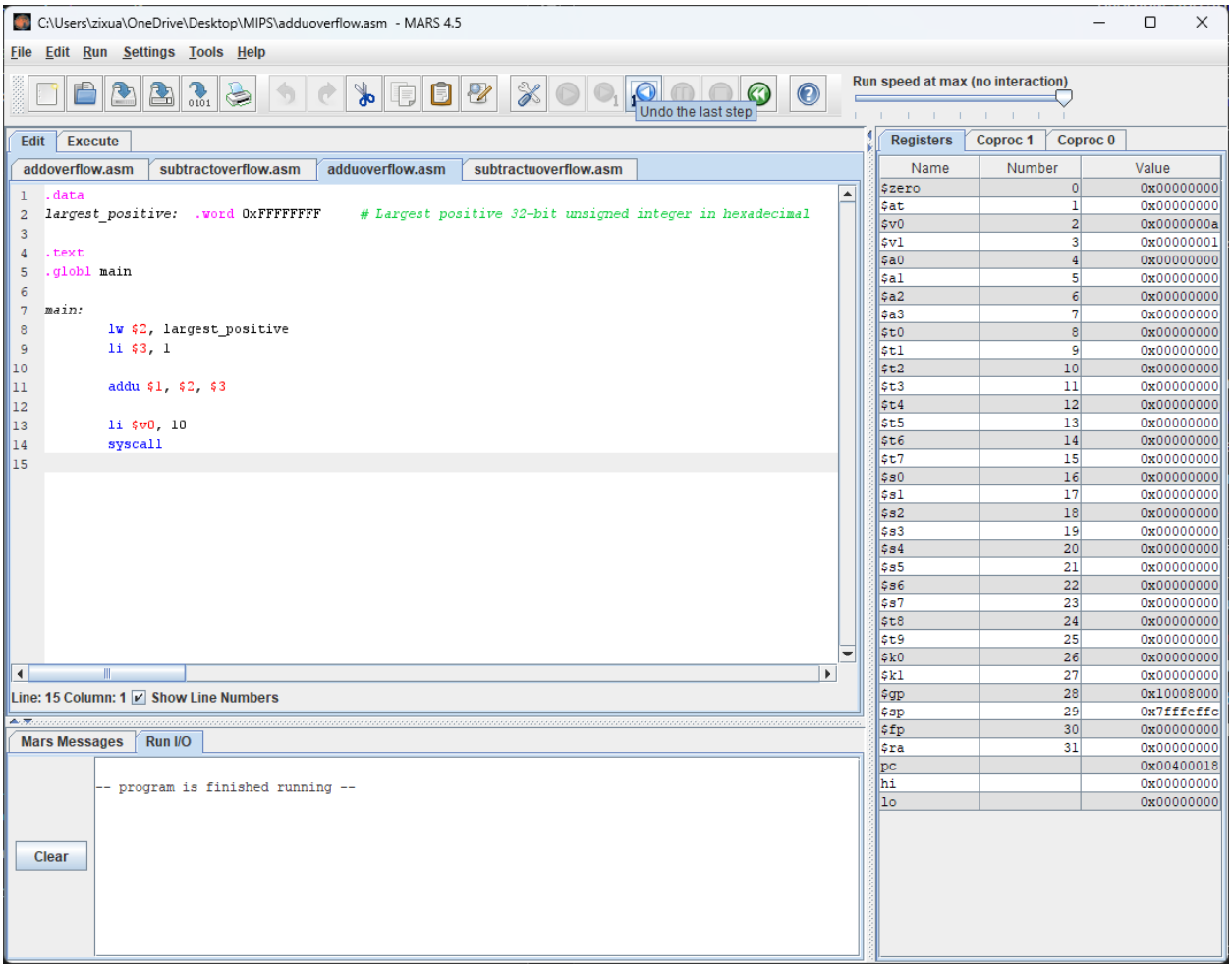

Figure 13: MIPS code snippet to demonstrate overflow by using addu on the largest positive 32-bit signed integer.

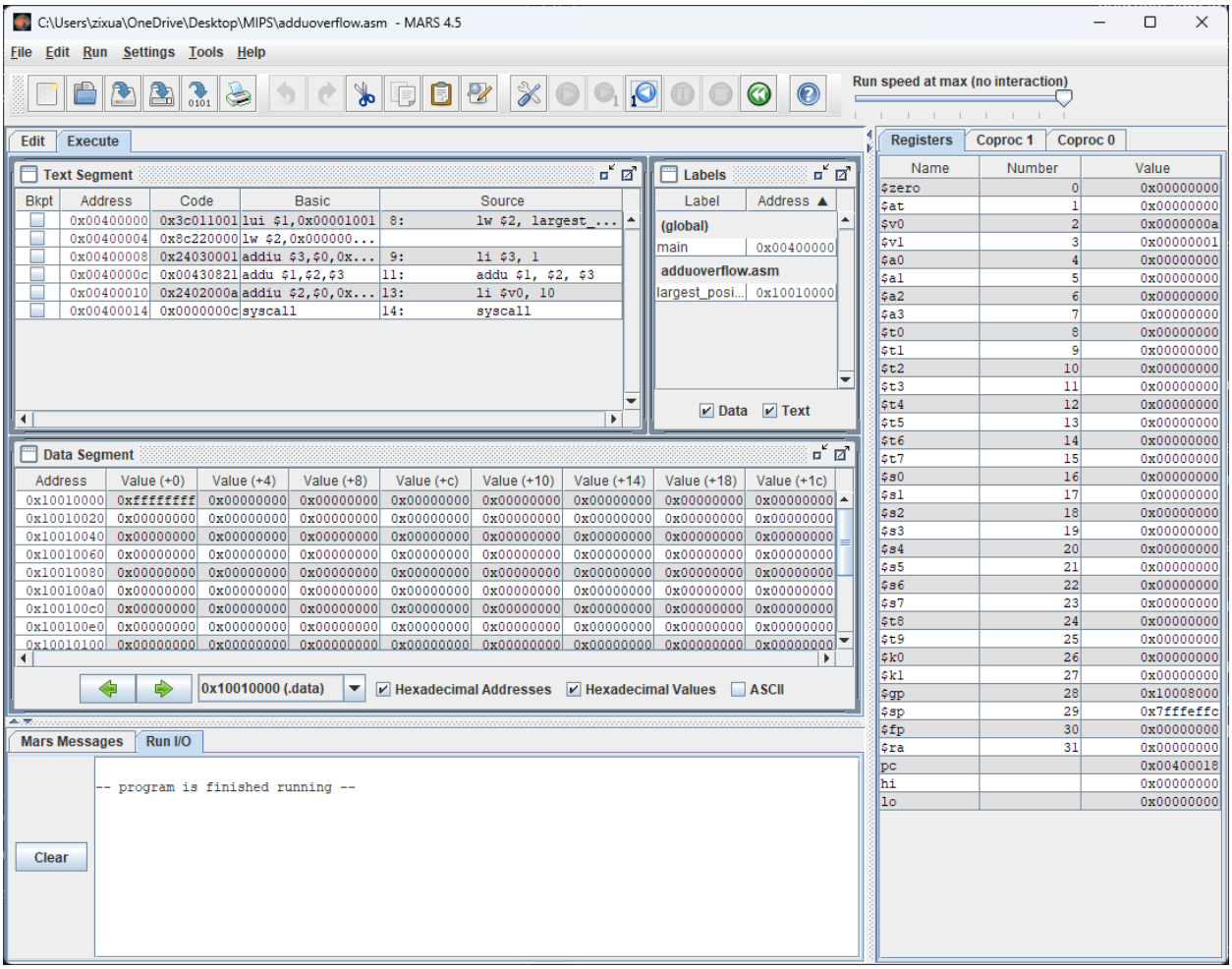

Figure 14: Output of code snippet in figure 13.

Subtraction for Unsigned Integers

Figures 15 & 16 do not display a run time exception, instead it displays the program is finished running. This is because SUBU treats the operands as unsigned numbers. This can be seen in register $at which is 0xffffffff because of the wraparound that occurred when subtracting 0x00000001 from 0x00000000.

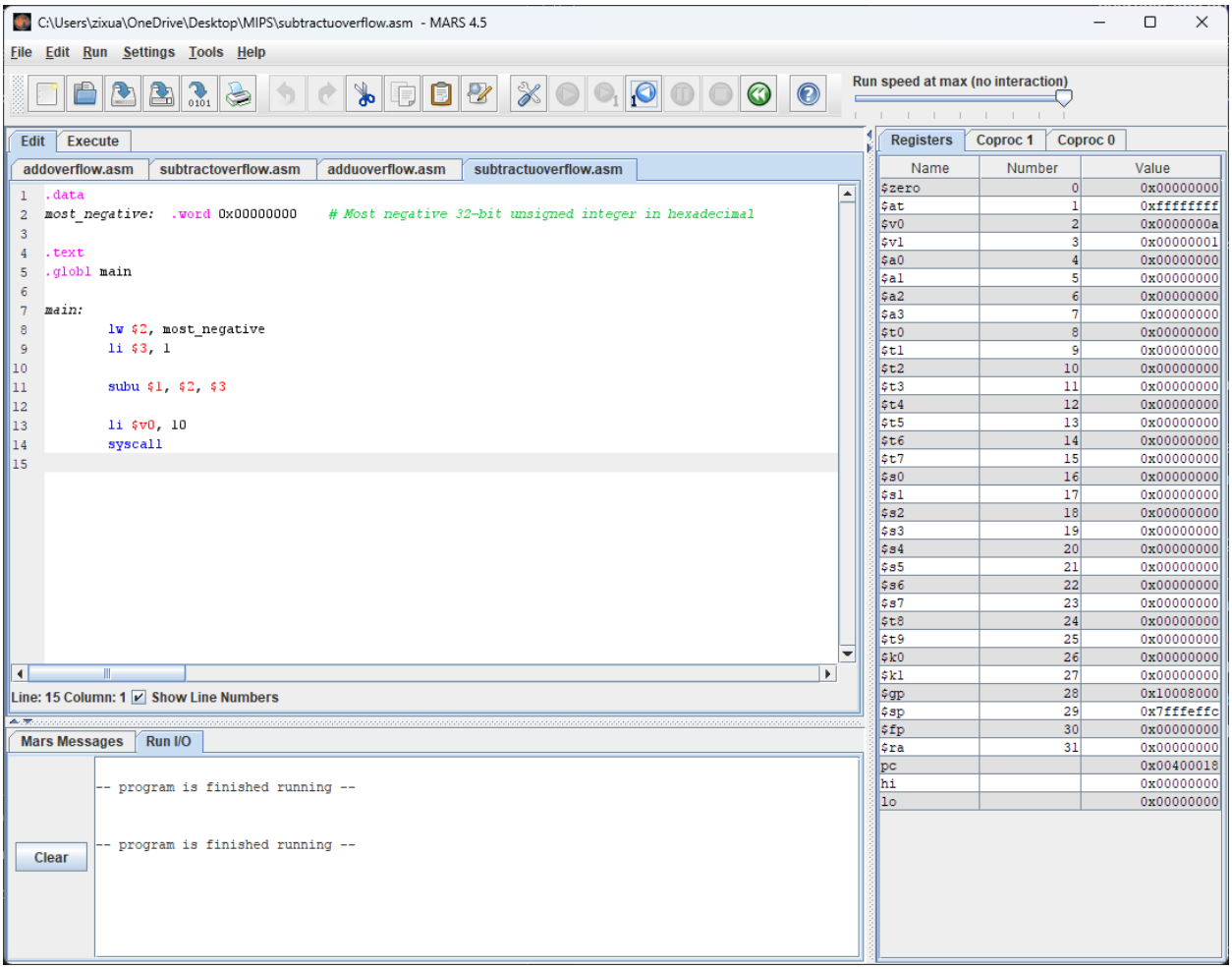

Figure 15: MIPS code snippet to demonstrate overflow by using subu on the most negative 32-bit signed integer.

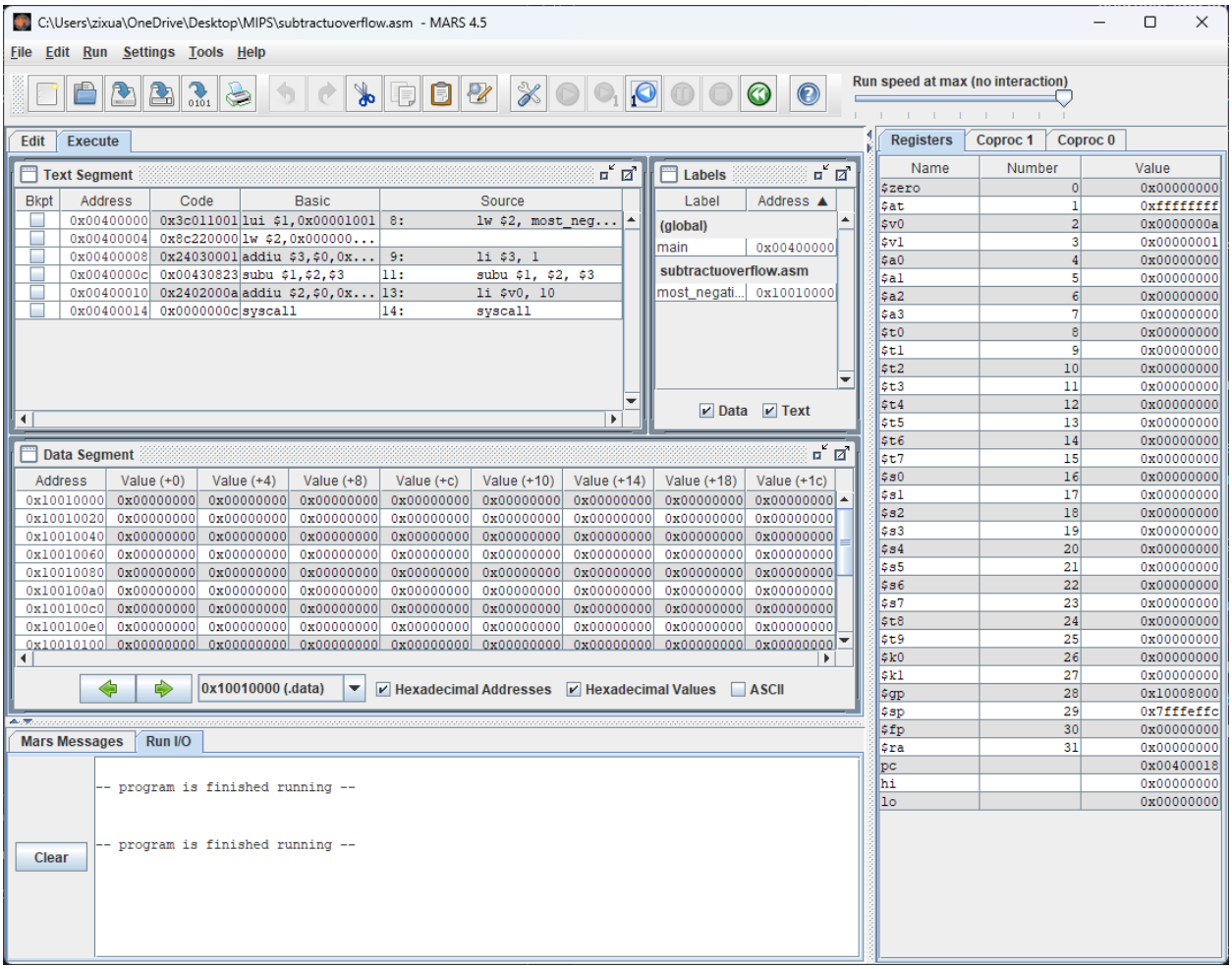

Figure 16: Output of code snippet in figure 15.

Conclusion: In this report, we explored how integer overflow manifests differently in signed and unsigned integers. We also talk about the possible consequences of overflow on binary representation and the importance of the signed bit. We also observed how integer overflow behaves differently in signed and unsigned integers. Adding 1 to the largest positive signed integer results in overflow, flipping the sign bit and wrapping around to the most negative integer, showcasing the limits of signed integer representation. Similarly, subtracting 1 from the most negative signed integer leads to overflow, wrapping around to the largest positive integer.

In contrast, adding 1 to the largest positive unsigned integer simply wraps around to 0, highlighting the absence of overflow checks for unsigned arithmetic.