COE 301/ICS 233, Term 161

Computer Architecture & Assembly Language

HW# 3 Solution

Q.1. Write a MIPS assembly program to perform **signed division** of 32-bit numbers using the algorithm studied in class. The program should ask the user to inter two integers and then display the result of division displaying both the quotient and remainder. Test your program using the following numbers:

1. +17 ÷ +3
2. +17 ÷ -3
3. -17 ÷ +3
4. -17 ÷ -3

A sample execution of the program is shown below:

Enter the dividend: 17
Enter the divisor: -3
Result of division: Quotient = -5 Remainder = 2

```
# Description: A program to implement 32-bit signed multiplication
################## Data segment ######################
.data
msg1: .asciiz "Enter the dividend:"
msg2: .asciiz "Enter the divisor:"
msg3: .asciiz "Result of division: "
msg4: .asciiz "Quotient = "
msg5: .asciiz "  Remainder = "
################## Code segment ######################
.text
.globl main
main:   # main program entry

# Getting the dividend
        li $v0, 4
        la $a0, msg1
        syscall
        li $v0, 5
        syscall
        move $s1, $v0
        xor $s5, $s5, $s5
        bgez $s1, Skip1
        li $s5, 0x80000000              # Storing the sign of the dividend
        neg $s1, $s1
Skip1:

# Getting the divisor
        li $v0, 4
        la $a0, msg2
```

```
        syscall
        li $v0, 5
        syscall
        move $s2, $v0
        xor $s6, $s6, $s6
        bgez $s2, Skip2
        li $s6, 0x80000000          # Storing the sign of the divisor
        neg $s2, $s2
Skip2:

# Performing signed division

        xor $s3, $s3, $s3           # Rem=0
        li $s4, 32                  # Loop counter
Loop:
        rol $t1, $s1, 1             # Shift (Remainder, Quotient) Left
        andi $t1, $t1, 1
        sll $s1, $s1, 1
        sll $s3, $s3, 1
        or $s3, $s3, $t1

        subu $t1, $s3,  $s2         # Difference = Remainder – Divisor

        sgtu $t2, $t1, $s3          # Check if Difference < 0
        bnez $t2, Negative
        move $s3, $t1               # Remainder = Difference
        ori $s1, $s1, 0x0001        # Set least significant bit of Quotient
Negative:
        addi $s4, $s4, -1
        bnez $s4, Loop

# Setting the sign of the result
        bgez $s5, Skip3             # Setting remainder sign
        neg $s3, $s3
Skip3:
        xor $s6, $s6, $s5           # Setting quotient sign
        bgez $s6, Skip4
        neg      $s1, $s1
Skip4:

# Displaying Result
        li $v0, 4
        la $a0, msg3
        syscall

        li $v0, 4                   # Display quotient
        la $a0, msg4
        syscall
        li $v0, 1
        move $a0, $s1
```

```
        syscall

        li $v0, 4                   # Display remainder
        la $a0, msg5
        syscall
        li $v0, 1
        move $a0, $s3
        syscall

        li $v0, 10                  # Exit program
        syscall
```

Results of running the program on the given test cases:

```
Enter the dividend:**** user input : 17
Enter the divisor:**** user input : 3
Result of division: Quotient = 5  Remainder = 2

Enter the dividend:**** user input : 17
Enter the divisor:**** user input : -3
Result of division: Quotient = -5  Remainder = 2

Enter the dividend:**** user input : -17
Enter the divisor:**** user input : 3
Result of division: Quotient = -5  Remainder = -2

Enter the dividend:**** user input : -17
Enter the divisor:**** user input : -3
Result of division: Quotient = 5  Remainder = -2
```

Q.2. Write a procedure, **GCD**, that receives two positive numbers in $a0 and $a1 and returns their greatest common divisor in register $v0. It is required that the procedure **preserves the content of all used registers** according to the MIPS programming convention by saving them and restoring them on the stack. The pseudo code of the GCD procedure is given below:

```
int gcd(int m, int n) {
   if ((m % n) == 0)
      return n;
   else
      return gcd(n, m % n);
}
```

```
GCD:
        divu $a0, $a1
        mfhi $t0
        bne $t0, $0, Else
        move $v0, $a1
        jr $ra
        Else:
        addi $sp, $sp, -4
```

```
sw $ra, ($sp)
move $a0, $a1
move $a1, $t0
jal GCD
lw $ra, ($sp)
addi $sp, $sp, 4
jr $ra
```

Q2.

(i) Given that **Multiplicand=1010** and **Multiplier=1011,** using **signed multiplication**, show the **signed** multiplication of **Multiplicand** by **Multiplier**. The result of the multiplication should be an 8 bit **signed** number in HI and LO registers. Show the steps of your work.

Iteration		**Multiplicand**	**Sign**	**Product = HI,LO**
0	Initialize (LO = Multiplier)	1010		0000 101**1**
1	LO[0] = 1 => ADD		1	1010 1011
	Shift Product = (HI, LO) right 1 bit	1010		1101 010**1**
2	LO[0] = 1 => ADD		1	0111 0101
	Shift Product = (HI, LO) right 1 bit	1010		1011 101**0**
3	LO[0] = 0 => Do nothing		1	1011 1010
	Shift Product = (HI, LO) right 1 bit	1010		1101 110**1**
4	LO[0] = 1 => SUB		0	0011 1101
	Shift Product = (HI, LO) right 1 bit			**0001 1110**

(ii) Given that **Dividend=1011** and **Divisor=0010**, Using **unsigned division**, show the **unsigned** division of **Dividend** by **Divisor**. The result of division should be stored in the Remainder and Quotient registers. Show the steps of your work.

Iteration		**Remainder (HI)**	**Quotient (LO)**	**Divisor**	**Difference**
0	Initialize	0000	1011	0010	
1	1: SLL, Difference	0001	0110	0010	1111
	2: Diff < 0 => Do Nothing	0001	0110	0010	
2	1: SLL, Difference	0010	1100	0010	0000
	2: Rem = Diff, set lsb Quotient	0000	110**1**	0010	
3	1: SLL, Difference	0001	1010	0010	1111
	2: Diff < 0 => Do Nothing	0001	1010	0010	
4	1: SLL, Difference	0011	0100	0010	0001
	2: Rem = Diff, set lsb Quotient	**0001**	**0101**	0010	

Q.3. What is the decimal value of the following single-precision floating-point numbers?

(i) `0010 0000 0001 1100 0000 0000 0000 0000`

$= + (1.0011100\ldots.0)_2 * 2^{(64-127)}$
$= + 1.21875 * 2^{-63}$

(ii) `1100 1111 1110 1000 0000 0000 0000 0000`

$= - (1.110100\ldots.0)_2 * 2^{(159-127)}$
$= - 1.8125 * 2^{32}$

Q.4. Show the IEEE 754 binary representation for: `-24.0625` in …

(i) Single Precision

1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(ii) Double precision

1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Q.5. Perform the following floating-point operations rounding the result to the nearest even. Perform the operation assuming both infinite precision and using only guard, round and sticky bits. Compare your solution in both cases.

(i) `0100 0011 1000 0000 0000 0000 0000 0000`
`-0100 0001 1000 0000 0000 0000 0000 1100`

With Infinite Precision:

```
      1.000 0000 0000 0000 0000 0000          x 2^8
-     1.000 0000 0000 0000 0000 1100          x 2^4
-------------------------------------------------------
=     1.000 0000 0000 0000 0000 0000          x 2^8
-     0.000 1000 0000 0000 0000 0000 1100     x 2^8
-------------------------------------------------------

=    01.000 0000 0000 0000 0000 0000          x 2^8
+    11.111 0111 1111 1111 1111 1111 0100     x 2^8
-------------------------------------------------------
=    00.111 0111 1111 1111 1111 1111 0100     x 2^8
=   +0.111 0111 1111 1111 1111 1111 0100      x 2^8
=   +1.110 1111 1111 1111 1111 1110 1000      x 2^7
```

Then, we round to the nearest even and we do not add a 1 since the least significant bit is 0. Thus, the result will be:

```
      +1.110 1111 1111 1111 1111 1110          x 2^7
```

With Guard, Round and Sticky Bits:

We add three bits for each operand representing G, R, S bits as follows.

```
     1.000 0000 0000 0000 0000 0000 000      x 2^8
-    1.000 0000 0000 0000 0000 1100 000      x 2^4
-------------------------------------------------
=    1.000 0000 0000 0000 0000 0000 000      x 2^8
-    0.000 1000 0000 0000 0000 0000 110      x 2^8
-------------------------------------------------

=   01.000 0000 0000 0000 0000 0000 000      x 2^8
+   11.111 0111 1111 1111 1111 1111 010      x 2^8
-------------------------------------------------
=   00.111 0111 1111 1111 1111 1111 010      x 2^8

=   +0.111 0111 1111 1111 1111 1111 010      x 2^8

=   +1.110 1111 1111 1111 1111 1110 100      x 2^7
```

Then, we round to the nearest even and we do not add a 1 since the least significant bit is 0. Thus, the result will be:

```
     +1.110 1111 1111 1111 1111 1110          x 2^7
```

(ii) `0011 1111 1000 0000 0000 0000 0000 0000`

`-0011 1111 1000 0100 0000 0000 0000 0000`

With Infinite Precision:

```
     1.000 0000 0000 0000 0000 0000          x 2^0
-    1.000 0100 0000 0000 0000 0000          x 2^0
-------------------------------------------------

=    01.000 0000 0000 0000 0000 0000         x 2^0
+    10.111 1100 0000 0000 0000 0000         x 2^0
-------------------------------------------------
=    11.111 1100 0000 0000 0000 0000         x 2^0

=   -0.000 0100 0000 0000 0000 0000          x 2^0

=   -1.000 0000 0000 0000 0000 0000          x 2^-5
```

No rounding is necessary in this case.

The case using Guard, Round and Sticky bits will produce identical result since there is no right shifting.

(iii) `0011 1111 1111 1111 1111 1111 1111 1110`

`+0011 0100 0100 0000 0000 0000 0000 0000`

With Infinite Precision:

```
     1.111 1111 1111 1111 1111 1110          x 2^0
+    1.100 0000 0000 0000 0000 0000          x 2^-23
-------------------------------------------------
=    1.111 1111 1111 1111 1111 1110          x 2^0
+    0.000 0000 0000 0000 0000 0001 1000 0000…000 x 2^0
-------------------------------------------------

=    1.111 1111 1111 1111 1111 1111 1000 0000…000 x 2^0
-------------------------------------------------
```

Then, we round to the nearest even and we add a 1 since the least significant bit is 1. Thus, the result will be:

$$= \quad 10.000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad \times 2^{0}$$

$$= \quad 1.000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad \times 2^{1}$$

With Guard, Round and Sticky Bits:

$$\begin{array}{lll} & 1.111\ 1111\ 1111\ 1111\ 1111\ 1110\ 000 & \times 2^{0} \\ + & 1.100\ 0000\ 0000\ 0000\ 0000\ 0000\ 000 & \times 2^{-23} \\ \hline = & 1.111\ 1111\ 1111\ 1111\ 1111\ 1110 & \times 2^{0} \\ + & 0.000\ 0000\ 0000\ 0000\ 0000\ 0001\ 100 & \times 2^{0} \\ \hline = & 1.111\ 1111\ 1111\ 1111\ 1111\ 1111\ 100 & \times 2^{0} \end{array}$$

Then, we round to the nearest even and we add a 1 since the least significant bit is 1. Thus, the result will be:

$$= \quad 10.000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad \times 2^{0}$$

$$= \quad 1.000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad \times 2^{1}$$

Q.6. Given that $x =$ **0101 1111 1011 1110 0100 0000 0000 0000**
$y =$ **0011 1111 1111 1000 0000 0000 0000 0000**
$z =$ **1101 1111 1011 1110 0100 0000 0000 0000**
represent single precision IEEE 754 floating-point numbers. Perform the following operations showing all work:

(i) $x + y$

We are going to solve the question using guard, round and sticky bits.

$$\begin{array}{lll} & 1.011\ 1110\ 0100\ 0000\ 0000\ 0000\ 000 & \times 2^{64} \\ + & 1.111\ 1000\ 0000\ 0000\ 0000\ 0000\ 000 & \times 2^{0} \\ \hline = & 1.011\ 1110\ 0100\ 0000\ 0000\ 0000\ 000 & \times 2^{64} \\ + & 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 001 & \times 2^{64} \\ \hline = & 1.011\ 1110\ 0100\ 0000\ 0000\ 0000\ 001 & \times 2^{64} \end{array}$$

Rounding the result to the nearest even gives the result:

$$= \quad +1.011\ 1110\ 0100\ 0000\ 0000\ 0000 \quad \times 2^{64}$$

(ii) Result of (**i**) + *z*

```
     1.011 1110 0100 0000 0000 0000 000      x 2^64
-    1.011 1110 0100 0000 0000 0000 000      x 2^64
=   01.011 1110 0100 0000 0000 0000 000      x 2^64
+   10.100 0001 1100 0000 0000 0000 000      x 2^64

=   00.000 0000 0000 0000 0000 0000 000      x 2^64
=   +0.000 0000 0000 0000 0000 0000 000      x 2^64
```

Rounding the result to the nearest even gives the result:

```
=   +0.000 0000 0000 0000 0000 0000          x 2^64
```

Note that this number is equivalent to 0.

(iii) Why is the result of (**ii**) counterintuitive?

The result is counterintuitive because z=-x and one expects that the result we will obtain will be equal to y. However, we got the result as 0. The main reason is that when we added x and y the result was x due to rounding. Thus, when we subtracted z=-x, we got 0.