King
CS 331
17 October 2020

Homework Assignment 3

Objectives

The purpose of this assignment is to:

- Understand different number representations including signed, unsigned integers, hexadecimal and floating point numbers,
- Perform different binary arithmetic operations including addition, subtraction, multiplication, and division,
- Identify when arithmetic operations result in overflow or underflow conditions.

Guidelines

All question numbers refer to the exercises at the end of Chapter 3 of the textbook. Solutions for the following problems are to be completed independently. Type each answer and clearly show the work/steps. Make sure to write any formula you use, do not plug in numbers right away.

Questions

Exercise 2.12.1 to 2.12.4

Assume that registers $s0 and $s1 hold the values 0x8000 0000 and 0xD000 0000, respectively.

1. What is the value of $t0 for the following assembly code?
   ```
   add $t0, $s0, $s1
   ```
2. Is the result in $t0 the desired result, or has there been overflow?
3. For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?
   ```
   sub $t0, $s0, $s1
   ```
4. Is the result in $t0 the desired result, or has there been overflow?

Given: s0=(8000 0000)_16, s1=(D000 0000)_16

1. t0 = (8000 0000)_16 + (D000 0000)_16
   t0 = (1000...0000)_2 + (1101...0000)_2 = (1 0101...0000)_2
   t0 = (0001 0101...0000)_2 = (0001 5000...0000)_16

2. Since the sum results in an overflow, t0 has the value (5000...0000)_16.

3. t0 = (8000 0000)_16 - (D000 0000)_16
   t0 = (1000...0000)_2 - (1101...0000)_2
   t0 = (1000...0000)_2 + (0011...0000)_2 = (1011...0000)_2
   t0 = (B000...0000)_16

4. Since (8000 0000)_16 - (D000 0000)_16 should evaluate to a negative number and the computed difference in hexadecimal is a positive number (B000...0000)_16, an overflow has occurred.

Exercise 3.1

What is 5ED4 - 07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

Given: 5ED4 (E=14, D=13), 07A4 (A=10)

5ED4 - 07A4 = (5-0), (14-7), (13-10), (4-4) = 5730

Exercise 3.12 (Notice the numbers are octal (base 8) and pay attention to how many iterations are involved.)

Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.

Given: (62)_8 × (12)_8

First convert operands from octal to binary, where each digit is represented by
(0)_8 = (000)_2
(1)_8 = (001)_2
...
(6)_8 = (110)_2
(7)_8 = (111)_2:

(62)_8 = (110 010)_2
(12)_8 = (001 010)_2

Evaluate (110 010)_2 × (001 010)_2:

Iter.	Step	Multiplier	Multiplicand	Product
0	Initial Values	**001 010**	**000 000 110 010**	**000 000 000 000**
1	1. 0->No operation	001 010	000 000 110 010	000 000 000 000

	2. Shift left Mcand	001 010	**000 001 100100**	000000 000 000
	3. Shift right Mplier	**000 101**	000 001 100 100	000000 000 000
2	1. 1->Prod=Prod+Mcand	000 101	000 001 100 100	**000001 100 100**
	2. Shift left Mcand	000 101	**000 011 001 000**	000001 100 100
	3. Shift right Mplier	**000 010**	000 011 001 000	000001 100 100
3	1. 0->No operation	000 010	000 011 001 000	000001 100 100
	2. Shift left Mcand	000 010	**000 110 010 000**	000001 100 100
	3. Shift right Mplier	**000 001**	000 110 010 000	000001 100 100
4	1. 1->Prod=Prod+Mcand	000 001	000 110 010 000	**000111 110 100**
	2. Shift left Mcand	000 001	**001 100 100 000**	000111 110 100
	3. Shift right Mplier	**000 000**	001 100 100 000	000111 110 100
5	1. 0->No operation	000 000	001 100 100 000	000111 110 100
	2. Shift left Mcand	000 000	**011 001 000 000**	000111 110 100
	3. Shift right Mplier	**000 000**	011 001 000 000	000111 110 100
6	1. 0->No operation	000 000	011 001 000 000	000111 110 100
	2. Shift left Mcand	000 000	**110 010 000 000**	000111 110 100
	3. Shift right Mplier	**000 000**	110 010 000 000	000111 110 100

Thus, $(110\ 010)_2 \times (001\ 010)_2 = (000\ 111\ 110\ 100)_2 = (764)_8 = 500$

Exercise 3.21

If the bit pattern 0×0C000000 is placed into the Instruction Register, what MIPS instruction will be executed?

Given: bit pattern $(0C00\ 0000)_{16}$

$(0C00\ 0000)_{16} = (0000\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2$

op = $(0000\ 11)_2$ = 3 = jal
imm = 0

The I-format instruction jal 0 will be executed.

Exercise 3.22

What decimal number does the bit pattern 0×0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

Given: bit pattern (0C00 0000)_16
Use: X = (-1)^(sign) × (1 + Fraction) × (2)^(Exponent - bias)

(0C00 0000)_16 = (0000 1100 0000 0000 0000 0000 0000 0000)_2

(0 0001 1000 00000000000000000000000)_2
S = 0
Fraction = 0
Exponent = (0001 1000)_2 = 24

X = (-1)^(sign) × (1 + Fraction) × (2)^(Exponent - bias)
X = (-1)^(0) × (1 + 0) × (2)^(24 - 127) = (1) × (1) × (2)^(-103)
X = (2)^(-103)

Last Exercise

Given the following 32-bit binary sequences representing single precision IEEE 754 floating point numbers:

a = 0100 0000 1101 1000 0000 0000 0000 0000
b = 1011 1110 1110 0000 0000 0000 0000 0000

Perform the following arithmetic and show the final addition and multiplication results in both normalized binary format and IEEE 754 single-precision format. Show your steps (Do not convert a and b to decimal base, compute the addition and multiplication, then convert the results back to normalized binary and single precision).

a. a + b
b. a × b

Given:
a = 0 10000001 1011000 0000 0000 0000 0000 = +1.1011 × 2^+2
b = 1 01111101 1100000 0000 0000 0000 0000 = -1.1100 × 2^-2 = +1.0100 × 2^-2

a. Floating-point addition, a + b
1. Align binary points (shift number with smaller exponent)
+1.101 100 × 2^+2
-0.000 111 × 2^+2 = +1.111 001 × 2^+2

2. Add significands

```
         +1.101 100
         +1.111 001
         ---------
         +1.100 101

      3. Normalize result and check for under/overflow
         1.100 101 × 2^+2

      4. Round and normalize if necessary
         1.1001 0100 × 2^+2

b. Floating-point multiplication, a × b
      1. Add exponents
         (2^+2) + (2^-2) = 2^0

      2. Multiply significands


      3. Normalize result and check for under/overflow


      4. Round and normalize if necessary


      5. Determine sign of result from signs of operands
```