# Exercise 1

## Binary to MIPS Instruction

1100 0101 0111 0000 0000 0000 0000 0000

110001 01 0111 0000 0000 0000 0000 0000

Opcode 0d110001 -> 0x31 -> load FP single [lwc1]

Lwc1 is type I

Unclear whether you only want the type or the assembly code, so here is the assembly code:

I type:

Opcode rs rt immediate

110001 01011 10000 0000000000000000

0d11 0d16

Lwc1 $t3 $s0 0d0

lwcl1 $s0, 0($t3)

## Binary Single precision IEEE 754 to decimal

1100 0101 0111 0000 0000 0000 0000 0000

IEEE 754 floating point looks like this (MIPS Greensheet)

Sign exponent fraction

1 bit 8 bits 23 bits

Sign Exponent Fraction

1 10001010 11100000000000000000000

0d1 0d138 ½ + ¼ + 1/8 = 7/8 = 0,875

Bias = 127 for single precision (MIPS Greensheet)

So with hidden bit:

(-1)^S x (1+Fraction) x 2^(Exponent-Bias) = (-1)^1 x (1 + 0,875) x 2^(138 – 127) = -3840

Without hidden bit:

(-1)^S x Fraction x 2^(Exponent-Bias) = (-1)^1 x 0,875 x 2^(138 – 127) = -1792

## Binary Signed integer to hex sign magnitude

1100 0101 0111 0000 0000 0000 0000 0000

MSB is 1, so the value is negative. In order to turn it positive: flip the bits and add 1.

1100 0101 0111 0000 0000 0000 0000 0000

0011 1010 1000 1111 1111 1111 1111 1111 flip bits

0011 1010 1001 0000 0000 0000 0000 0000 add 1

3 A 9 0 0 0 0 0 convert each 4 bits to hex

So it’s magnitude in hex is 3A90 0000

## Binary unsigned integer to hex

1100 0101 0111 0000 0000 0000 0000 0000

Converting each 4 bits to hex gives:

C 5 7 0 0 0 0 0

So it becomes C570 0000 in hex

## Binary to ASCII

1100 0101 0111 0000 0000 0000 0000 0000

Each ASCII character is each 8 bits.

11000101 01110000 00000000 00000000

0xC5 0x70 0x0 0x0

Using the asciitable.com for conversion

The ASCII is: ┼ p NUL NUL

# Exercise 2

Okay, the question states that the trick works for floating points, but in the slides the trick is given for integers. So I assume that the terms are swapped in the questions.

Trying out the division test for floating point numbers:

1) 1,7 div -0,5 gives Q = -3, R = 0,2

2) -1,7 div -0,5 gives Q = 3, R = -0,2

When testing the integer division trick:

Sign(Remainder) = Sign(Dividend)

Sign(Quotient) = Sign(Dividend) XOR Sign(Divisor)

Where positive is 0, negative is 1.

For 1):

Sign(Remainder) = Sign(1,7) = 0 (pos)

Sign(Quotient) = Sign(1,7) XOR Sign(-0,5) = 1 (neg)

For 2):

Sign(Remainder) = Sign(-1,7) = 1 (neg)

Sign(Quotient) = Sign(-1,7) XOR Sign(-0,5) = 0 (pos)

So the integer trick works for floating points as well.

“Can you do similarly for multiplications, both integers and floating-point numbers?”

Where positive is 0, negative is 1.

Result = Multiplicand \* Multiplier

Sign(Result) = Sign(Multiplicand) XOR Sign(Multiplier)

-1 x -1 = 1

-1 x 1 = -1

1 x -1 = -1

1 x 1 = 1

This works for both floating points and integers.

# Exercise 3

The overflow occurs when the sign of the result is opposite of those of the operands. The signs of the operands have to be the same. Found in the slides of lecture 6. So when two negative numbers added together result in a positive number. Or when two positive numbers added together result in a negative number.

Let a be input one, b input two, r the result of addition, and o whether there is overflow or not.

Then

O = (a(msb) XNOR b(msb)) AND (r(msb) XOR a(msb))

First check if operands have same sign using XNOR, then check whether the first operand and the result have different signs. If both are true, then overflow must occur.

# Exercise 4

For single precision, the bias is 127 (MIPS Greensheet). 127 is 2^7 – 1. This is the maximum value of 7 bits, one bit less than the amount of bits in the exponent. This would mean for the 16-bit variant the bias is 2^4 – 1 = 15, because the exponent is 5 bits.

The 16-bit floating point should be able to represent +- 0, +- inf and NaN, just like the floating point types.

For the numbers it can represent, the range of the exponent is from 1 to MAX – 1 (MIPS Greensheet). So from 1 to 30. Adjusting with the bias gives the range of -14 to 15.

There are 16 – 1 (sign bit) – 5 (exponent) = 10 bits for fraction. Answer number 3 has 10 bits in the fraction (so after the decimal point). This is of course because the fraction part of the binary representation does not include the hidden bit.

This means the answer is **3**

# Exercise 5

Since the question does not define what kind of adder you are asking for: half-adder, full-adder, one-bit, 64-bit. I will assume you are asking for 64-bit adders.

## Slide 15: refined version

Diagram

Description automatically generated

In the 64 bit case, there is still one ALU. So also one adder.

## Slide 16: Faster multiplier

## Diagram, engineering drawing Description automatically generated

For this multiplier in the 64-bit case, there are 64 adders.

## Slide 17: Tree multiplier

Diagram

Description automatically generated

In the 64-bit case, there will be 63 adders placed in the tree structure.

# Exercise 6:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | Description of the step | Quotient | Divisor | Remainder |
| 0 | Initial values | 0000 | 0100 0000 | 0000 1001 |
| 1 | Rem = Rem – Div | 0000 | 0100 0000 | 1100 1001 |
| Rem < 0 -> +Div, sll Q, Q0 = 0 | 0000 | 0100 0000 | 0000 1001 |
| Shift Div right | 0000 | 0010 0000 | 0000 1001 |
| 2 | Rem = Rem – Div | 0000 | 0010 0000 | 1110 1001 |
| Rem < 0 -> +Div, sll Q, Q0 = 0 | 0000 | 0010 0000 | 0000 1001 |
| Shift Div right | 0000 | 0001 0000 | 0000 1001 |
| 3 | Rem = Rem – Div | 0000 | 0001 0000 | 1111 1001 |
| Rem < 0 -> +Div, sll Q, Q0 = 0 | 0000 | 0001 0000 | 0000 1001 |
| Shift Div right | 0000 | 0000 1000 | 0000 1001 |
| 4 | Rem = Rem – Div | 0000 | 0000 1000 | 0000 0001 |
| Rem > 0 : sll Q, Q0 = 1 | 0001 | 0000 1000 | 0000 0001 |
| Shift Div right | 0001 | 0000 0100 | 0000 0001 |
| 5 | Rem = Rem – Div | 0001 | 0000 0100 | 1111 1101 |
| Rem < 0 -> +Div, sll Q, Q0 = 0 | 0010 | 0000 0100 | 0000 0001 |
| Shift Div right | 0010 | 0000 0010 | 0000 0001 |

Blue is left shift, Red means negative test result, Green means positive test result, Orange is right shift, Purple is updated to old version (for remainder)

So Quotient = 0b0010 = 0d2, Divisor = 0b10 = 0b2, Remainder = 0b1 = 0d1. This is correct because 9 / 4 = 2 rest 1.

|  |  |  |
| --- | --- | --- |
| Signal | Figure 3.8 (slide 21) | Figure 3.11 (improved version slide 22) |
| Write | Enables storing the result from the subtraction in the ALU into the registers | Same |
| Remainder | Is used to test whether the current value stored in the remainder is negative | Same |
| Shift left (Quotient) | Shifts the quotient to the left after the test. Stores either a 0 or 1 inside the LSB of the quotient | Is now shift left for the Remainder/Quotient register |
| ALU (Sub) | Flag for deciding on addition/subtraction in the ALU | Same |
| (Divisor) shift right | Shifts the divisor to the right | Is now the shift right to compensate the shifts of the remainder. It only shifts the remainder to the right, not the entire remainder/quotient register |
| ALU (flag) | Non-existent | It tells the control test whether the result of the subtraction is negative. |