FLOATING POINT MULTIPLICATION

Problem 6: Multiply the numbers $1.110_{ten} \times 10^{10}$ and $9.200_{ten} \times 10^{-5}$

Assume that we can store only four digits of the significand and two digits of the exponent

Step 1: Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result:

$$10 + 127 = 137, \text{ and } -5 + 127 = 122,$$

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field.

FLOATING POINT MULTIPLICATION

The problem is with the bias because we are adding the biases as well as the exponents:

New exponent = (10 + 127) + (-5 + 127) = (5 + 2 x 127) = 259

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

New exponent = 137 + 122 - 127 = 132 = (5 + 127)

Step 2: Multiplication of the significants

$$
\begin{array}{r}
1.110_{ten} \\
\times \quad 9.200_{ten} \\
\hline
0000 \\
0000 \\
2220 \\
9990 \\
\hline
10212000_{ten}
\end{array}
$$

Product significand: 10.212000_{ten}

Assuming, that we can keep only three digits to the right of the decimal point, the product is 10.212×10^5.

FLOATING POINT MULTIPLICATION

Step 3: This product is unnormalized, so we need to normalize it:

$$10.212_{ten} \text{ x } 10^5 = 1.0212_{ten} \text{ x } 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent.

At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4: We assumed that the significand is only four digits long (excluding the sign), so we must round the number.

The number 1.0212_{ten} x 10^6 is rounded to four digits in the significand to 1.021_{ten} x 10^6

FLOATING POINT MULTIPLICATION

Step 5: The sign of the product depends on the signs of the original operands.

- If they are both the same, the sign is positive;
- otherwise, it's negative.

Hence, the product is $\mathbf{+1.021_{ten} \times 10^6}$

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication, the sign of the product is determined by the signs of the operands.

FLOATING POINT MULTIPLICATION

Multiplication of binary floating-point numbers is similar to the steps just seen.

Step 1: We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result.

Step 2: Multiplication of significands.

Step 3: Optional normalization step. The size of the exponent is checked for overflow or underflow.

Step 4: Then the product is rounded. If rounding leads to further normalization, we once again check for exponent size.

Step 5: set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

FLOATING POINT MULTIPLICATION

Problem 7: Multiply the numbers 0.5_{ten} and -0.4375_{ten} .

Convert the two numbers to binary and represent it in the normalized scientific notation, assume 4 bits of precision.

$$0.5_{ten} = 0.1_{two} = 0.1_{two} \text{ x } 2^{0} = 1.000_{two} \text{ x } 2^{-1}$$

$$-0.4375_{ten} = -0.0111_{two} = -0.0111_{two} \text{ x } 2^{0} = -1.110_{two} \text{ x } 2^{-2}$$

Now the algorithm,

Step 1: Adding the exponents without bias: -1 + (-2) = -3

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

FLOATING POINT MULTIPLICATION

Step 2: Multiplying the significants

$$
\begin{array}{r}
1.000_{two} \\
\times \quad 1.110_{two} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{two}
\end{array}
$$

This product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$

Step 3: Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow.

Step 4: Rounding the product makes no change: $1.110_{two} \times 2^{-3}$

FLOATING POINT MULTIPLICATION

Step 5: Since the signs of the original operands differ, make the sign of the product negative.

Hence, the product is

$$\mathbf{-1.110_{two} \times 2^{-3}}$$

Converting to decimal to check our results:

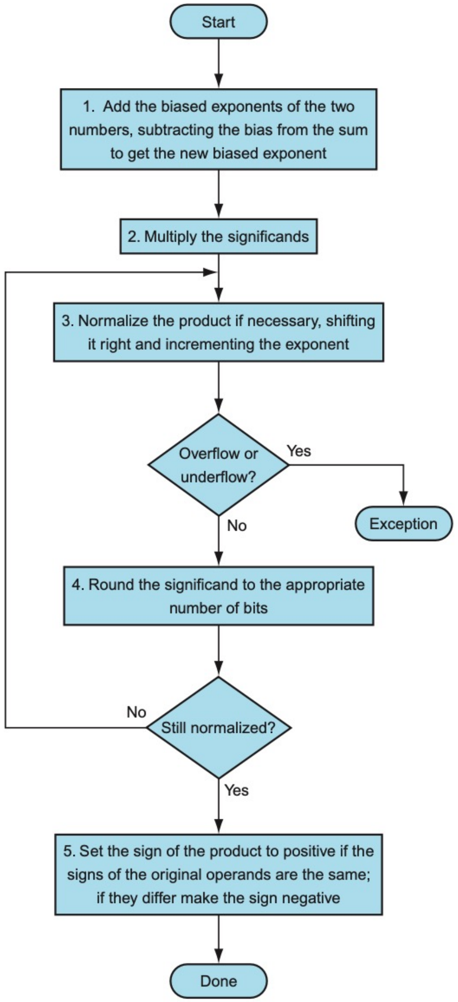
$$-1.110_{two} \text{ x } 2^{-3} = -0.001110_{two} = -0.00111_{two}$$

$$(0 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) + (1 \times 2^{-5})$$

$$0 + 0 + 0.125 + 0.0625 + 0.03125 = \mathbf{-\ 0.21875_{ten}}$$

The product of 0.5_{ten} and $-\ 0.4375_{ten}$ is indeed -0.21875_{ten}

FLOATING POINT MULTIPLICATION



FLOATING POINT INSTRUCTIONS IN MIPS

MIPS supports the IEEE 754 single precision and double precision formats with these

- Floating-point *addition, single* (`add.s`) and *addition, double* (`add.d`)
- Floating-point *subtraction, single* (`sub.s`) and *subtraction, double* (`sub.d`)
- Floating-point *multiplication, single* (`mul.s`) and *multiplication, double* (`mul.d`)
- Floating-point *division, single* (`div.s`) and *division, double* (`div.d`)
- Floating-point *comparison, single* (`c.x.s`) and *comparison, double* (`c.x.d`), where `x` may be *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), or *greater than or equal* (`ge`)
- Floating-point *branch, true* (`bc1t`) and *branch, false* (`bc1f`)

Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.

FLOATING POINT INSTRUCTIONS IN MIPS

The MIPS designers decided to add separate floating-point registers—called `$f0, $f1, $f2`, ...—used either for single precision or double precision.

Hence, they included separate loads and stores for floating-point registers: `lwc1` and `swc1`.

The base registers for floating-point data transfers which are used for addresses remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
lwc1     $f4,c($sp)  # Load 32-bit F.P. number into F4
lwc1     $f6,a($sp)  # Load 32-bit F.P. number into F6
add.s    $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1     $f2,b($sp)  # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name. Thus, the pair of single precision registers `$f2` and `$f3` also form the double precision register named `$f2`.

FLOATING POINT INSTRUCTIONS IN MIPS

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2, . . . , $f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 – $f6	FP sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 – $f6	FP sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision