

# LAB C: Integer Multiplication and Division. Floating Point (FP) Calculations

## C1 Integer multiplication and division (`mul`, `mulh`, `mulhu`, `mulhsu`, `div`, `rem`)

When 64 bit integers are multiplied, depending on their values, the result can easily take more than 64 bits.

### (`mul`)

For multiplication of small values, e.g.  $11 \times 12 = 132$  that produce a result with up to 64 bits a single `mul` (multiply) instruction is sufficient for calculating the lower 64 bits:

```
addi x5,x0,11
addi x6,x0,12
mul x7,x5,x6
```

Save the above example as a file named `c1a.asm` for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5 t0 0x000000000000000B 11
x6 t1 0x000000000000000C 12
x7 t2 0x0000000000000084 132
```

### (`mulh`, `mulhu`, `mulhsu`)

For multiplication of larger values that produce a result with more than 64 bits additional `mulh` (multiply upper half), `mulhu` (multiply upper half unsigned), and `mulhsu` (multiply upper half signed unsigned), instructions are provided for calculating the upper 64 bits. Note that the `mul` instruction is still needed for calculating the lower 64 bits:

```
v1: DD -2
v2: DD 16
ld x5, v1(x0)
ld x6, v2(x0)
mul x7,x5,x6 ;lower 64 bits
mulh x8,x5,x6 ;upper 64 bits (signed, signed)
mulhu x9, x5, x6 ;upper 64 bits (unsigned, unsigned)
mulhsu x10, x5, x6 ;upper 64 bits (signed, unsigned)
mulhsu x11, x6, x5 ;upper 64 bits (signed, unsigned)
```

To illustrate this, in the above example we have chosen as a multiplicand  $-2$  for its simple hexadecimal representation of `0xfffffffffffffffe`. We have then multiplied it by `16` which effectively shifted the hexadecimal digits of the multiplicand by 1 position (4 bits) to the left. Save the above example as a file named `c1b.asm` for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5 t0 0xfffffffffffffffe -2
x6 t1 0x0000000000000010 16
x7 t2 0xffffffffffffffe0 -32
x8 s0 0xfffffffffffffffe -1
x9 s1 0x000000000000000f 15
x10 a0 0xfffffffffffffffe -1
x11 a1 0x000000000000000f 15
```

Note the different values obtained by the `mulh`, `mulhu`, and `mulhsu` instructions.

In the following example we ask the user to i) input the width  $w$  and the height  $h$  of a rectangle, ii) use `mul` to calculate the area  $a=w*h$  of the rectangle, and iii) output the calculated result:

```
s0:  DC    "Rectangle area calculation.\0"
s1:  DC    "Enter a:\0"
s2:  DC    "Enter b:\0"
s3:  DC    "Rectangle area="
      addi  x5, x0, s0
      ecall x0, x5, 4 ;out info
      addi  x5, x0, s1
      ecall x1, x5, 4 ;prompt a
      ecall x6, x0, 5 ;inp a
      addi  x5, x0, s2
      ecall x1, x5, 4 ;prompt b
      ecall x7, x0, 5 ;inp b
      mul   x6, x6, x7 ;area=a*b
      addi  x5, x0, s3
      ecall x1, x5, 4 ;out area
      ecall x1, x6, 0 ;out result
```

Save the above example as a file named **c1c.asm** for possible future use.

Compile and run the above example. Here is a sample dialog obtained when testing the above program:

```
Rectangle area calculation.
Enter a:121
Enter b:200
Rectangle area=24200
```

### **(div)**

Integer division is carried out by the `div` (divide) instruction as shown in the following example where we calculate the value of  $1000/20$ :

```
addi  x5, x0, 1000
addi  x6, x0, 20
div   x7, x5, x6
```

Save the above example as a file named **c1d.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0  0x000000000000003E8 1000
x6  t1  0x00000000000000014 20
x7  t2  0x00000000000000032 50
```

### **(rem)**

The remainder can be obtained by using the `rem` (remainder) instruction as shown in the following example where we calculate  $67\%16$ :

```
addi  x5, x0, 67
addi  x6, x0, 16
rem   x7, x5, x6
```

Save the above example as a file named **c1e.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0  0x00000000000000043 67
x6  t1  0x00000000000000010 16
x7  t2  0x00000000000000003 3
```

In the following example we ask the user to i) input two positive integers  $a$  and  $b$ , ii) use the Euclid's algorithm to find their greatest common divisor  $\text{gcd}(a,b)$ , and iii) output the calculated result:

```
s0:  DC    "Greatest common divisor.\0"
s1:  DC    "Enter a:\0"
s2:  DC    "Enter b:\0"
s3:  DC    "gcd(a,b)="
      addi x5, x0, s0
      ecall x0, x5, 4 ;out info
      addi x5, x0, s1
      ecall x1, x5, 4 ;prompt a
      ecall x6, x0, 5 ;inp a
      addi x5, x0, s2
      ecall x1, x5, 4 ;prompt b
      ecall x7, x0, 5 ;inp b
loop: rem  x8, x6, x7
      add  x6, x0, x7
      add  x7, x0, x8
      bne  x8, x0, loop
      addi x5, x0, s3
      ecall x1, x5, 4 ;out gcd
      ecall x0, x6, 0 ;out result
```

Save the above example as a file named **c1f.asm** for possible future use.

Compile and run the above example. Here is a sample dialog obtained when testing the above program:

```
Greatest common divisor.
Enter a:12
Enter b:15
gcd(a,b)=3
```

**Exercise cex1a:** Write an assembly program that i) inputs a small positive integer, ii) calculates the factorial of that integer, and iii) outputs the calculated result. Here is a possible sample dialog (the user entered 5 so  $5!$  was calculated):

```
n! Enter n:5
5!=120
```

Compile and test your solution with different input values.

Save your solution as a file named **cex1a.asm** for possible future use.

**Exercise cex1b:** Write an assembly program that i) inputs a positive integer  $n$ , ii) sums all positive integers smaller than  $n$  in a loop, and iii) outputs the obtained result. For verification calculate the sum directly using the expression  $(n*(n-1))/2$  and output the resulting value. Here is a possible sample dialog (the user entered 20):

```
sum(1..n-1) Enter n:20
sum(1..19)=190
(n*(n-1))/2=190
```

Compile and test your solution with different input values.

Save your solution as a file named **cex1b.asm** for possible future use.

**Exercise cex1c:** Write an assembly program that i) inputs a positive integer *i*, ii) finds all its divisors, and iii) outputs the calculated result. Here is a possible sample dialog (the user entered 10):

Find all divisors.

Enter i:10

Divisors:

1

2

5

10

Compile and test your solution with different input values.

Save your solution as a file named **cex1c.asm** for possible future use.

**Exercise cex1d:** Modify the program **cex1c.asm** so that the found divisors are not output but stored in the memory after the compiled code. Then, depending on the found divisors, output as a result either “prime” (the only divisors were 1 and the number itself) or “not prime” (other divisors but 1 and the number itself were found.)

Save your solution as a file named **cex1d.asm** for possible future use.

## C2 Floating Point (FP) calculations (DF, fld, fadd.d, fsub.d, fmul.d, fdiv.d, fsd, fmin.d, fmax.d, fsqrt.d, fsgnj.d, fsgnjn.d, fsgnjx.d, fmv.x.d, fmv.d.x, fcvt.l.d, fcvt.d.l, flt.d, fle.d)

Floating Point (FP) values and calculations in this section are in double precision (64-bit.)

### (DF)

Double precision FP values can be created and stored in memory at compile time by the DF (Define Float) assembler command as follows:

```
DF      1.0
```

Compile the above example and check the result in the **Mem** window:

```
0x0000000000000000 0x3ff0000000000000 4607182418800017408
```

```
1.0000000000000000E+000      DF      1.0
```

The normalized double precision FP representation of 1.0 begins with the three hexadecimal digits of 0x3FF which written in binary format will be 0011 1111 1111 and in decimal format  $2^{10}-1=1023$ . This represents a nonnegative FP number as its first bit (the sign bit) is 0. The exponent of the FP number is  $1023-1023=0$ . The mantissa of the FP number is represented by the remaining 13 hexadecimal digits on the right of the first three and contains only 0s. The value of the FP number will therefore be  $1.0_2 * 2^0 = 1.0$

Compare the above representation with the representation of -1.0 below:

```
DF      -1.0
```

```
0x0000000000000000 0xbff0000000000000 -4616189618054758400
```

```
-1.0000000000000000E+000      DF      -1.0
```

The only difference is in the first bit changing to 1 (0xbff vs. 0x3ff) and, as this is the sign bit, the number becomes negative.

```
DF      2.0
```

```
0x0000000000000000 0x4000000000000000 4611686018427387904
```

```
2.0000000000000000E+000      DF      2.0
```

Now note that for the FP number 2.0 shown above the exponent field becomes  $0x400=1024$  so the exponent will be  $1024-1023=1$  and thus we multiply the mantissa by  $2^1$ . As 2.0 is a power of 2, all bits in its mantissa are 0s.

```

DF      0.5
0x0000000000000000 0x3fe0000000000000 4602678819172646912
5.0000000000000000E-001      DF      0.5

```

For the FP number 0.5 shown above the exponent field becomes  $0x3FE=1022$  so the exponent will be  $1022-1023=-1$  and thus we multiply the mantissa by  $2^{-1}$ . Again, as 0.5 is a power of 2, all bits in its mantissa are 0s.

We will use now the FP number 3.0 which is not a power of 2 to illustrate how the mantissa bits work.

```

DF      3.0
0x0000000000000000 0x4008000000000000 4613937818241073152
3.0000000000000000E+000      DF      3.0

```

The mantissa of the FP number 3.0 above begins with the hexadecimal digit of  $0x8$  followed by all 0s. This means the mantissa has only its leftmost bit set to 1 with all other bits set to 0. Recall that we had the same exponent bits when analyzing the FP number 2.0 above and we obtained an exponent value of 1. We can, therefore, calculate the value of this FP number as  $1.1 \times 2^1 = 1 \times 2^1 + 1 \times 2^0 = 2 + 1 = 3.0$ .

### (fld)

Through the fld (FP load double) instruction FP values in the memory can be loaded into FP registers:

```

d1:  DF      2.0
d2:  DF      3.0
      fld     f1, d1(x0)
      fld     f2, d2(x0)

```

Save the above example as a file named **c2a.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```

f1  0x4000000000000000  2.0000000000000000E+000
f2  0x4008000000000000  3.0000000000000000E+000

```

### (fadd.d, fsub.d, fmul.d, fdiv.d)

Here is an example of FP addition, subtraction, multiplication, and division:

```

d1:  DF      2.0
d2:  DF      3.0
      fld     f1, d1(x0)
      fld     f2, d2(x0)
      fadd.d   f3, f1, f2          ; 2.0+3.0=  5.0
      fsub.d   f4, f1, f2          ; 2.0-3.0= -1.0
      fmul.d   f5, f1, f2          ; 2.0*3.0=  6.0
      fdiv.d   f6, f1, f2          ; 2.0/3.0=  0.(6)

```

Save the above example as a file named **c2b.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```

f1  0x4000000000000000  2.0000000000000000E+000
f2  0x4008000000000000  3.0000000000000000E+000
f3  0x4014000000000000  5.0000000000000000E+000
f4  0xbff0000000000000 -1.0000000000000000E+000
f5  0x4018000000000000  6.0000000000000000E+000
f6  0x3fe5555555555555  6.6666666666666663E-001

```

Note that since FP numbers are approximations of real numbers and the FP arithmetic has a limited precision, in some cases obtained results may differ, depending on the order of the calculations. We will use the following sample program to illustrate this effect:

```
v1:  DF      -1.1e17
v2:  DF      1.1e17
v3:  DF      3.0
      fld     f1, v1(x0)
      fld     f2, v2(x0)
      fld     f3, v3(x0)
      fadd.d  f4, f1, f2
      fadd.d  f4, f4, f3
      fadd.d  f5, f2, f3
      fadd.d  f5, f1, f5
```

Save the above example as a file named **c2c.asm** for possible future use.

The above code calculates the following two sums:

$f4 = (v1+v2) + v3$

$f5 = v1 + (v2+v3)$

Since addition is associative we would expect to obtain the same resulting values the FP register  $f4$  and  $f5$ .

Now compile and run the above example. Check the resulting values in the **Regs** window:

```
f1  0xc3786cc6acd4b000 -1.1000000000000000E+017
f2  0x43786cc6acd4b000  1.1000000000000000E+017
f3  0x4008000000000000  3.0000000000000000E+000
f4  0x4008000000000000  3.0000000000000000E+000
f5  0x0000000000000000  0.0000000000000000E+000
```

The resulting values in  $f4$  and  $f5$  shown above are clearly different which illustrates that, depending on the employed values, FP addition may not be associative.

Note that we will obtain a different result if we change the first two lines in the above program as follows:

```
v1:  DF      -1.1e15
v2:  DF      1.1e15
v3:  DF      3.0
      fld     f1, v1(x0)
      fld     f2, v2(x0)
      fld     f3, v3(x0)
      fadd.d  f4, f1, f2
      fadd.d  f4, f4, f3
      fadd.d  f5, f2, f3
      fadd.d  f5, f1, f5
```

Save the modified example as a file named **c2d.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
f1  0xc30f438daa060000 -1.1000000000000000E+015
f2  0x430f438daa060000  1.1000000000000000E+015
f3  0x4008000000000000  3.0000000000000000E+000
f4  0x4008000000000000  3.0000000000000000E+000
f5  0x4008000000000000  3.0000000000000000E+000
```

The resulting values in  $f4$  and  $f5$  shown above are now the same. The demonstrated effect is due to loss of precision. Indeed, since the mantissa of the double precision FP values contains 52 bits, it can represent up to 15 decimal digits without rounding. This is just enough to obtain

the correct result in the second case where the operands differ by about  $10^{15}$  but is clearly not sufficient in the first case where the operands differ by about  $10^{17}$ .

### (fsd)

The calculated FP results are always placed in FP registers. From there they can be stored in the memory by using the `fsd` (FP store double) instruction as shown in the following example which calculates  $(3.0+3.0)/(3.0*3.0)$  and stores the result in a memory location right after the program:

```
d:    DF        3.0
      fld      f1, d(x0)
      fadd.d   f2, f1, f1
      fmul.d   f3, f1, f1
      fdiv.d   f4, f2, f3
      fsd      f4, m(x9)
m:    DM        1
```

Save the above example as a file named `c2e.asm` for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
f1  0x4008000000000000    3.000000000000000E+000
f2  0x4018000000000000    6.000000000000000E+000
f3  0x4022000000000000    9.000000000000000E+000
f4  0x3fe5555555555555    6.666666666666663E-001
```

Also check the stored values in the **Mem** window:

```
0x0000000000000000    0x4008000000000000    4613937818241073152
3.000000000000000E+000 d:    DF        3.0
0x00000000000000020    0x3fe5555555555555    4604180019048437077
6.666666666666663E-001 m:    DM        1
```

### (fmin.d, fmax.d)

While getting the minimum or the maximum of two integers values requires several instructions, the minimum or the maximum of two FP values, could be obtained with a single instruction. Indeed, the `fmin.d` (FP minimum) instruction finds the minimum of the values in its source registers `fs1` and `fs2` and stores the result in its destination register `fd`. Respectively, the `fmax.d` (FP maximum) instruction can be used for the maximum FP value. Here is a usage example:

```
fc1:  DF        1.0
fc2:  DF        2.0
      fld      f1, fc1(x0)
      fld      f2, fc2(x0)
      fmin.d   f3, f1, f2
      fmax.d   f4, f1, f2
```

Save the above example as a file named `c2f.asm` for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
f1  0x3ff0000000000000    1.000000000000000E+000
f2  0x4000000000000000    2.000000000000000E+000
f3  0x3ff0000000000000    1.000000000000000E+000
f4  0x4000000000000000    2.000000000000000E+000
```

### (fsqrt.d)

The `fsqrt.d` (FP square root) instruction calculates the square root of the value in the single source register `fs` and places the result in the destination register `fd`:

```
d:    DF        2.0
      fld      f1, d(x0)
```

```
fsqrt.d    f2, f1
```

Save the above example as a file named **c2g.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
f1  0x4000000000000000  2.000000000000000E+000
```

```
f2  0x3ff6a09e667f3bcd  1.4142135623730951E+000
```

### **(fsgnj.d, fsgnjn.d, fsgnjx.d)**

The sign injection FP instructions `fsgnj.d` (FP sign inject), `fsgnjn.d` (FP sign inject NOT), and `fsgnjx.d` (FP sign inject XOR) transfer the value of `fs1` to `fd` and adjust the sign of the result (produced result takes all bits except the sign bit from `fs1`.) More precisely, `fsgnj.d` sets the sign of `fd` to the sign of the value in `fs2`, `fsgnjn.d` sets the sign of `fd` to the opposite sign of the value in `fs2`, and `fsgnjx.d` sets the sign of `fd` to the XOR of the sign bits of `fs1` and `fs2`. As shown below, sign injection instructions can be used to transfer a value, to transfer the negate of a value, and to transfer the absolute of a value:

```
fc1:  DF      -2.0
fc2:  DF      3.0
      fld      f1, fc1(x0)
      fld      f2, fc2(x0)
      fsgnj.d   f3, f1, f1 ;f3=f1
      fsgnjn.d  f4, f1, f1 ;f3=-f1
      fsgnjx.d  f5, f1, f1 ;f3=abs(f1)
```

Save the above example as a file named **c2h.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
f1  0xc000000000000000 -2.000000000000000E+000
```

```
f2  0x4008000000000000  3.000000000000000E+000
```

```
f3  0xc000000000000000 -2.000000000000000E+000
```

```
f4  0x4000000000000000  2.000000000000000E+000
```

```
f5  0x4000000000000000  2.000000000000000E+000
```

### **(fmv.x.d, fmv.d.x)**

The `fcvt.x.d` (FP move double to long) and `fcvt.d.x` (FP move long to double) instructions move data between the FP and the integer registers without conversion:

```
d:    DF      2.0
      fld      f0, d(x0)
      fmv.x.d   x30, f0
      addi     x31, x0, -1
      fmv.d.x   f1, x31
```

Save the above example as a file named **c2i.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```
x30 t5  0x4000000000000000 4611686018427387904
```

```
x31 t6  0xFFFFFFFFFFFFFFF -1
```

```
f0  0x4000000000000000  2.000000000000000E+000
```

```
f1  0xfffffffffffffffff -NaN(7fffffffffffffff)
```

### **(fcvt.l.d, fcvt.d.l)**

The `fcvt.l.d` (FP convert double to long) and `fcvt.d.l` (FP convert long to double) instructions conduct FP to integer and vise versa conversions. In the following example we first calculate the approximate value of  $\pi$  by an FP division ( $22.0/7.0$ ) and then convert the obtained result to an integer:

```
addi     x5, x0, 22
addi     x6, x0, 7
fcvt.d.l  f5, x5      ;from 64bit int
```



```

fcvt.d.l    f6, x6
fdiv.d      f7, f5, f6 ;22.0/7.0= 3.1428...
fcvt.l.d    x7, f7      ;to 64bit int

```

Save the above example as a file named **c2j.asm** for possible future use.

Compile and run the above example. Check the resulting values in the **Regs** window:

```

x5 t0  0x0000000000000016 22
x6 t1  0x0000000000000007 7
x7 t2  0x0000000000000003 3
f5 0x4036000000000000 2.200000000000000E+001
f6 0x401c000000000000 7.000000000000000E+000
f7 0x4009249249249249 3.1428571428571428E+000

```

### **(flt.d, fle.d)**

Comparing double precision FP values is carried out by the `flt.d` (FP less than) and `fle.d` (FP less or equal) instructions. To illustrate their use we will create an assembly program corresponding to the following pseudo-code:

```
if (x<a) x=b elsif (b<=x) x=a
```

The values of `a` and `b` are stored in the memory at compile time. The value of `x` is provided by the user at run time. The following assembly program will prompt the user to enter a value for `x`, then will calculate the new value of `x` as per the above pseudo-code, and finally will output the result. These three steps will be repeated in a loop until the program execution is cancelled.

```

s1:  DC      "Enter x:\0"
s2:  DC      "Result:\0"
c0:  DF      0.0
a:   DF      3.0
b:   DF      5.0
     fld     f0, c0(x0)
     fld     f1, a(x0)
     fld     f2, b(x0)
loop: addi   x5, x0, s1
     ecalls  x1, x5, 4 ;out info
     ecalls  f3, x0, 6 ;inp x
     flt.d   x1, f3, f1
     beq     x1, x0, cont
     fadd.d  f3, f0, f2
     beq     x0, x0, done
cont: fle.d  x1, f2, f3
     beq     x1, x0, done
     fadd.d  f3, f0, f1
done: addi   x5, x0, s2
     ecalls  x1, x5, 4 ;out x
     ecalls  x0, f3, 1 ;out res
     beq     x0, x0, loop

```

Save the above example as a file named **c2k.asm** for possible future use.

Compile and run the above example. Here is an example of a possible dialog:

```

Enter x:0
Result:5.0
Enter x:2
Result:5.0
Enter x:3
Result:3.0

```

```
Enter x:4
Result:4.0
Enter x:3.5
Result:3.5
Enter x:4.99
Result:4.99
Enter x:5
Result:3.0
Enter x:6
Result:3.0
Enter x:Cancelled
```

**Exercise cex2a:** Write an assembly program for calculating the dot product of two vectors. Use DD assembler commands to store in the beginning of the memory two sample vectors, e.g. 1.21, 5.85, -7.3, 23.1, -5.55 and 3.14, -2.1, 44.2, 11.0, -7.77. Calculate the corresponding dot product and store the resulting value (-3.392210E+001) in the memory right after the code. Test your program with different sample vectors. Save your solution as a file named **cex2a.asm** for possible future use.

**Exercise cex2b:** Create an assembly program that approximates the value of  $e$  by summing  $1.0/0! + 1.0/1! + 1.0/2! + 1.0/3! + \dots + 1.0/n!$ . Run the program with different values of  $n$  and compare the results. Determine the properly calculated digits in each case using the following reference value:  $e = 2.71828182845904523536$ . Save your solution as a file named **cex2b.asm** for possible future use.

**Exercise cex2c:** Create an assembly program for calculating the square root of a given FP number  $a$  using the bisection method for root finding. Essentially you will have to iterate to find an approximation of the root of  $x*x-a=0$  with a given precision. Note that the root can be found in the interval  $(a, 1)$  for  $a < 1$  and in the interval  $(1, a)$  for  $a > 1$ . Compile and test your solution with different precisions. Save your solution as a file named **cex2c.asm** for possible future use.