

Assignment Project Exam Help
Floating Point

<https://powcoder.com>

Add WeChat powcoder

Outline

- Special “numbers” revisited
 - Rounding
 - FP add/sub
 - FP on MIPS
 - Integer multiplication & division
- Assignment Project Exam Help
- <https://powcoder.com>
- Add WeChat powcoder

IEEE 754 Floating Point Review



Precision	Sign (S)	Exponent (E)	Fraction (F)	Bias
Float	1 bit	8 bits	23 bits	127
Double	1 bit	11 bits	52 bits	1023

https://powcoder.com
Add WeChat powcoder

$$(-1)^S \times (1+F) \times 2^{(E-\text{bias})}$$

- Numbers in *normalized* form, i.e., 1.xxxx...
- The standard also defines special symbols

Special Numbers Reviewed

- Special symbols (single precision)

Exponent	Fraction	Object represented
0	0	0
0	Nonzero	\pm denormalized number
1-254	Anything	\pm floating point number
255	0	\pm infinity
255	Nonzero	NaN (Not a Number)

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
 - If infinity is not an error, these shouldn't be either.
 - Called Not a Number (NaN)
 - Exponent = 255, Significand = 0
- Why is this useful?
 - Hope NaNs help with debugging?
 - They contaminate: `op(NaN, X) = NaN`

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0

Assignment Project Exam Help

– Smallest representable positive number:

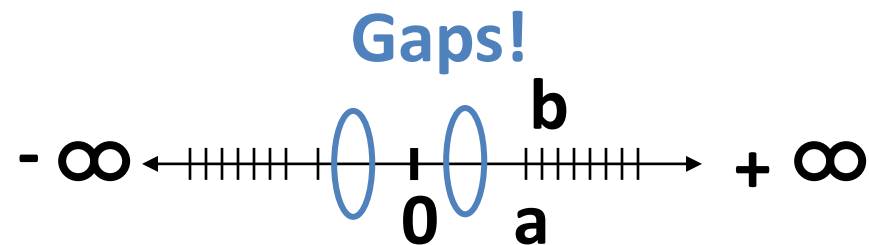
$$a = 1.00000000000000000000000000000000_2 * 2^{-126} = 2^{-126}$$

– Second smallest representable positive number:

[illegible]

$$a - 0 = 2^{-126}$$

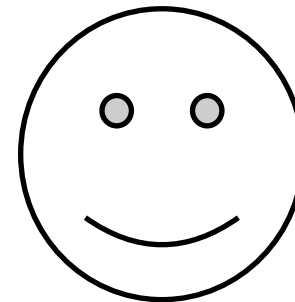
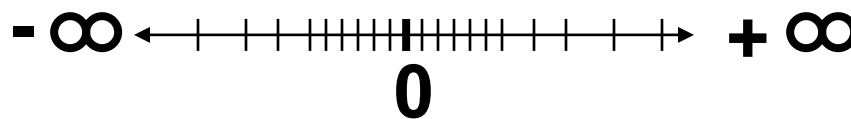
$$b - a = 2^{-149}$$



Normalization and implicit 1 is to blame!

Representation for Denorms (2/2)

- Solution: special symbol in exponent field
 - Use 0 in exponent field, nonzero for fraction
 - Denormalized number
 - Has no leading 1
 - **Has implicit exponent = -126 (i.e. don't subtract bias)**
 - Smallest positive *float*: $2e-149$
 - 2nd smallest positive *float*: $2e-148$



Small numbers and Denormalized

$1.0000000000000000000000001_2 \times 2^{-126}$

$1.0000000000000000000000001_2 \times 2^{-126}$

$1.0000000000000000000000000_2 \times 2^{-126}$

$0.1111111111111111111111111_2 \times 2^{-126}$ Denormalized!

$0.1111111111111111111111110_2 \times 2^{-126}$

$0.1111111111111111111111101_2 \times 2^{-126}$

...

$0.0000000000000000000000011_2 \times 2^{-126}$

$0.0000000000000000000000010_2 \times 2^{-126}$

$0.0000000000000000000000001_2 \times 2^{-126}$

Next smaller number is zero

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Rounding

- When we perform math on real numbers, we must worry about rounding to fit the result in the significant field.
 - The FP hardware carries **two extra bits** of precision, and then rounds to get the proper value
 - Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

IEEE Has Four Rounding Modes

1. Round towards +infinity

- ALWAYS round “up”: 2.001 \rightarrow 3
 - -2.001 \rightarrow -2
- Assignment Project Exam Help $\lceil x \rceil$ or $\lceil x \rceil$

2. Round towards -infinity

- ALWAYS round “down”: 1.999 \rightarrow 1,
 - -1.999 \rightarrow -2
- <https://powcoder.com>
Add WeChat powcoder $\lfloor x \rfloor$ or $\lfloor x \rfloor$

3. Truncate

- Just drop the last bits (round towards 0)

4. Round to (nearest) even

- Normal rounding, almost

Round to Even

- Round like you learned in grade school
- **Except** if the value is right on the borderline, in which case we round to the nearest EVEN number
 - 2.5 -> 2
 - 3.5 -> 4
- Insures *fairness*
 - This way, half the time we round up on tie, the other half time we round down
- This is the default rounding mode

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

FP Addition and Subtraction 1/2

- ***Much*** more difficult than with integers
- Cannot just add significands
- Recall how we do it:
 1. De-normalize to match larger exponent
 2. Add significands to get resulting one
 3. Normalize and check for under/overflow
 4. Round if needed (may need to goto 3)
- Note: If signs differ, perform a subtract instead
 - Subtract is similar except for step 2

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

FP Addition and Subtraction 2/2

- Problems in implementing FP add/sub:
 - If signs differ for add (or same for sub), what is the sign of the result?
- Question:
 - How do we integrate this into the integer arithmetic unit?
 - Answer: We don't! Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

MIPS Floating Point Architecture (1/4)

- Separate floating point instructions:
 - Single Precision:
`add.s, sub.s, mul.s, div.s`
 - Double Precision:
`add.d, sub.d, mul.d, div.d`
- These instructions are ***far more complicated*** than their integer counterparts, so they can take much longer to execute.

MIPS Floating Point Architecture (2/4)

- Observations

- It's inefficient to have different instructions take vastly differing amounts of time.
- Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.
- Some programs do no floating point calculations
- It takes lots of hardware relative to integers to make Floating Point fast

MIPS Floating Point Architecture (3/4)

- Pre 1990 Solution:
 - separate chip to do floating point (FP)
- Coprocessor 1: FP chip
 - Contains 32 32-bit registers: $\$f0, \$f1, \dots$
 - Usually registers specified in FP instructions refer to this set
 - Separate load and store: **lwc1** and **swc1** (“load word coprocessor 1”, “store ...”)
 - Double Precision: by convention, **even**/odd pair contain one DP FP number: $\$f0/\$f1, \$f2/\$f3, \dots, \$f30/\$f31$ where the **even register** is the name



MIPS Floating Point Architecture (4/4)

- Pre 1990 Computers contains multiple separate chips:
 - Processor: handles all the normal stuff
 - Coprocessor 1: handles FP and only FP;
 - more coprocessors? (yes, more on this later)
- Today, FP coprocessor integrated with CPU, or specialized or inexpensive chips may leave out FP HW
- Instructions to move data between main processor and coprocessors, e.g., mfc0, mtc0, mfc1, mtc1

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Some More Example FP Instructions

```
abs.s $f0, $f2 # f0 = abs( f2 );  
neg.s $f0, $f2 # f0 = - f2;  
sqrt.s $f0, $f2 # f0 = sqrt( f2 );
```

```
c.lt.s $f0, $f2 # is $f0 < $f2 ?  
bc1t    label    # branch on condition true
```

See 4th edition text 3.5 and App. B for a complete list of floating point instructions

Copying, Conversion, Rounding

```
mfc1 $t0, $f0    # copy $f0 to $t0
mtc1 $t0, $f0    # copy $t0 to $f0
```

```
cvt.d.s $f0 $f2  # f0f1 gets float f2 converted to double
cvt.d.w $f0 $f2  # f0f1 gets int f2 converted to double
```

```
cvt.s.d $f0 $f2  # f0 gets double f2f3 converted to float
cvt.s.w $f0 $f2  # f0 gets int f2 converted to float
```

```
ceil.w.s $f0 $f2 # round to next higher integer
floor.w.s $f0 $f2 # round down to next lower integer
trunc.w.s $f0 $f2 # round towards zero
round.w.s $f0 $f2 # round to closest integer
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Dealing with Constants

`float a = 3.14;`

- Option 1

- Declare constant 3.14 in data segment of memory
- Load the address label
- Load to coprocessor

```
.data
PI: .float 3.14
.text
la    $t0 PI
lwc1  $f0 ($t0)
l.s   $f0 PI      # easiest
```

- Option 2

- Compute hexadecimal IEEE representation for 3.14 (it is 0x4048F5C3)
- Load immediate
- Move to coprocessor

```
lui   $t0 0x4048
ori   $t0 $t0 0xF5C3
mtc1  $t0 $f0
```

Option 3, pseudoinstruction
not available in MARS:
`li.s $f0, 3.14`

Floating Point Register Conventions

(\$f0, \$f1), and (\$f2, \$f3)	Function return registers used to return float and double values from function calls.
(\$f12, \$f13) and (\$f14, \$f15)	Two pairs of registers used to pass float and double valued arguments to functions. Pairs of registers are parenthesized because they have to pass double values. To pass float values, only \$f12 and \$f14 are used.
\$f4, \$f6, \$f8, \$f10, \$f16, \$f18	Temporary registers
\$f20, \$f22, \$f24, \$f26, \$f28, \$f30	Save registers whose values are preserved across function calls

Unfortunately no nice names (e.g., \$t#, \$s#) like with the main registers)

With double precision instructions, the high-order 32-bits are in the implied odd register.

Fahrenheit to Celsius



```
float f2c(float f) { return 5.0/9.0*(f-32.0); }
```

.text

f2c:

```
    la    $t0, const5
    lwc1   $f16, ($t0)
    la    $t0, const9
    lwc1   $f18, ($t0)
    div.s  $f16, $f16, $f18    # f16 = 5.0/9.0
    la    $t0, const32
    lwc1   $f18, ($t0)
    sub.s  $f18, $f12, $f18    # f18 = fahr-32.0
    mul.s  $f0, $f16, $f18     # return f16*f18
    jr     $ra
```

.data

```
const5: .float 5.0
const9: .float 9.0
const32: .float 32.0
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Debugging FP Code in MARS

- MARS displays floating point registers in hexadecimal
- This makes debugging floating point code tricky...
 - Can use MARS “Floating Point Representation” tool to examine single precision
 - Alternatively **syscall** can be used to print to console

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Service	Code in \$v0	Arguments
Print float	2	\$f12 = float to print
Print double	3	\$f12 = double to print
Print string	4	\$a0 = address of null-terminated string to print

```
.data
spaceString:    .asciiz " "
newlineString:  .asciiz "\n"
```

```
printSpace:
    li $v0, 4
    la $a0, spaceString
    syscall
    jr $ra

printNewLine:
    li $v0, 4
    la $a0, newlineString
    syscall
    jr $ra

printFloat: # in $f12
    li $v0, 2
    syscall
    jr $ra
```

```
# print( float vec[4] )
printFloatVector:
    addi $sp, $sp, -8
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    move $s0, $a0
    lwc1 $f12, 0($s0)
    jal printFloat
    jal printSpace
    lwc1 $f12, 4($s0)
    jal printFloat
    jal printSpace
    lwc1 $f12, 8($s0)
    jal printFloat
    jal printSpace
    lwc1 $f12, 12($s0)
    jal printFloat
    jal printNewLine
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    addi $sp, $sp, 4
    jr $ra
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

REMEMBER: Floating Point Fallacy

- FP add, subtract associative? **FALSE!**

$$x = -1.5 \times 10^{38} \quad y = 1.5 \times 10^{38} \quad z = 1.0$$

$$\begin{aligned} x + (y + z) &= -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) \\ &= -1.5 \times 10^{38} + (1.5 \times 10^{38}) \\ &= \mathbf{0.0} \end{aligned}$$

$$\begin{aligned} (x + y) + z &= (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 \\ &= (0.0) + 1.0 \\ &= \mathbf{1.0} \end{aligned}$$

- Floating Point add, subtract are not associative!
 - Floating point result *approximates* real result!

Casting floats \leftrightarrow ints

- `(int) floating point expression`

- Coerces and converts it to the nearest integer
(C uses truncation)

```
i = (int) (3.14159 * E);
```

- `(float) expression`

- converts integer to nearest floating point

```
f = f + (float) i;
```

int → float → int

```
if ( i == (int) ((float) i) ) {  
    printf("true");  
}
```

Assignment Project Exam Help

<https://powcoder.com>

- Does this always print true?
 - No, it will **not** always print “true”
 - Large values of integers don’t have exact floating point representations
- What about double?

Add WeChat powcoder



float → int → float

```
if ( f == (float) ((int) f) ) {  
    printf("true");  
}
```

Assignment Project Exam Help

<https://powcoder.com>

- Does this always print true?
 - No, it will **not** always print "true"
 - Small floating point numbers (<1) don't have integer representations
 - Same is true for large numbers
 - For other numbers, rounding errors



MIPS Integer Multiplication

- Syntax of Multiplication (signed): **MULT** reg1 reg2
- Result of multiplying 32 bit registers has 64 bits
- MIPS splits 64-bit result into 2 special registers
 - upper half in **hi**, lower half in **lo**
 - Registers **hi** and **lo** are separate from the 32 general purpose registers
 - Use **MFHI reg** to move from **hi** to register
 - Use **MFLO reg** to move from **lo** to another register
- Unusual syntax compared to other instructions!

MIPS Integer Multiplication Example

$$a = b * c;$$

Let b be \$s2; let c be \$s3;

And let a be \$s0 and \$s1 (it may be up to 64 bits)

Assignment Project Exam Help

<https://powcoder.com>

```
mult  $s2  $s3    # b*c
mfhi  $s0         # get upper half of product
mflo  $s1         # get lower half of product
```

- We often only care about the low half of the product!

MIPS Integer Division

- Syntax of Division (signed): **DIV reg1 reg2**
 - Divides register 1 by register 2
 - Puts remainder of division in hi
 - Puts quotient of division in lo
- Notice that this can be used to implement both the division operator (/) and modulo operator (%) in a high level language

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

MIPS Integer Division Example

a = c / d;

b = c % d;

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Variable	Register
a	\$s0
b	\$s1
c	\$s2
d	\$s3

```
div    $s2 $s3    # lo=c/d, hi=c%d
mflo   $s0        # get quotient
mfhi   $s1        # get remainder
```


Unsigned Instructions and Overflow

- MIPS has versions of **mult** and **div** for **unsigned operands**:

multu, divu

- Determines whether the product and quotient are changed if the operands are signed or unsigned.

- Typically unsigned instructions check for overflow (e.g., add vs addu)
- MIPS does not check overflow or division by zero on ANY signed/unsigned multiply, divide instruction
 - Up to the software to check “hi”, “divisor”



Things to Remember

- Integer multiplication and division:
 - `mult`, `div`, `mfhi`, `mflo`
- New MIPS registers (`$f0-$f31`) and instructions in two flavours
 - Single Precision `.s`
 - Double Precision `.d`
- FP add and subtract are *not associative*...
- IEEE 754 NaN & Denorms (precision) review
- IEEE 754's Four different rounding modes

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Review and More Information

- Textbook

- Section 3.5 Floating Point

- We saw the representation and addition and multiplication algorithm material earlier in the term
 - And now we have seen the Floating-Point instructions

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder