Computer Architecture

Exercise 9

(floating point)

Peter Stallinga

Universidade do Algarve 2024-2025

(Decimal) floating point numbers are of the form

$$n = f \cdot 10^{e}$$

with *f* the fraction and *e* the exponent. **Both *f* and *e* are essentially integers**: The system is called floating point, because we can ‘float’ the point in the fraction while adjusting the exponent so that there are no non-zero digits after the point:

$$3.478 \cdot 10^{5} = 3478.0 \cdot 10^{2}.$$

Both 3478 and 2 are integers. We can also ‘normalize’ the number so that the first non-zero digit is right after the floating point to get a fraction between 0 and 1:

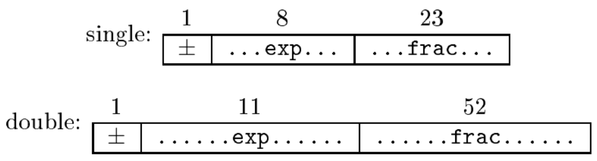
$$3.478 \cdot 10^{5} = 0.3478 \cdot 10^{6}.$$

But don’t forget that the numbers continue to be intrinsically integers. And if the number of digits in both *f* and *e* are limited we can represent them in a computer bit pattern. In that case the floating-point computer calculations are not always exact and we have to tolerate calculation errors caused by rounding. The larger the bit pattern, the smaller the rounding errors. That is why we use double instead of float in C when we want higher quality calculations.

The IEEE 754 convention introduced for binary floating point is the following. A 32-bit single (float) is essentially of the form

$$\pm 1.\text{frac} \cdot 2^{\text{exp}-127}.$$

One bit for the sign, 23 bits for the fraction, where the significand is 1.frac, and 8 bits for the exp, where the exponent is then ‘excess-127’, namely exp−127.



Other sizes also exist (half, double, etc.). See the table below. Only single and double are implemented in MIPS.

floating point:

±	...exp...	...frac...

name	total size		±	exp		frac
	bits	bytes	bits	bits	(excess)	bits
half	16	2	1	5	(15)	10
single	32	4	1	8	(127)	23
double	64	8	1	11	(1023)	52
extended	80	10	1	15	(16383)	64
quadruple	128	16	1	15	(16383)	112
octuple	256	32	1	19	(262143)	236

As an example, the hexadecimal bit pattern `0x3f000000` is:

3	f	0	0	0	0	0	0
0011	1111	0000	0000	0000	0000	0000	0000

grouping the bits:

sign	exp	frac
0	01111110	00000000000000000000000

which translates into
sign: $+$
exponent: $\texttt{exp} - 127 = 126 - 127 = -1$
significand: $1.\texttt{frac} = 1.0$
So the number is: $+1.0 \times 2^{-1} = 0.5$ (base 10).

For these calculations, MIPS has floating point registers and floating point operations. The 32 FP registers (`$f`) again consist of `$t` registers, `$s` registers, and `$a` registers that have the usual meaning we have seen before for integer registers: temporary, saved, and argument registers, resp. Note there are no return-value registers (`$v`)

Floating-point registers:

$f0	$ft0	$f8	$fs0	$f16	$fa6	$f24	$fs8
$f1	$ft1	$f9	$fs1	$f17	$fa7	$f25	$fs9
$f2	$ft2	$f10	$fa0	$f18	$fs2	$f26	$fs10
$f3	$ft3	$f11	$fa1	$f19	$fs3	$f27	$fs11
$f4	$ft4	$f12	$fa2	$f20	$fs4	$f28	$ft8
$f5	$ft5	$f13	$fa3	$f21	$fs5	$f29	$ft9
$f6	$ft6	$f14	$fa4	$f22	$fs6	$f30	$ft10
$f7	$ft7	$f15	$fa5	$f23	$fs7	$f31	$ft11

Arithmetic instructions are of the type, for example

```
add.s $fd, $ft, $fs
```

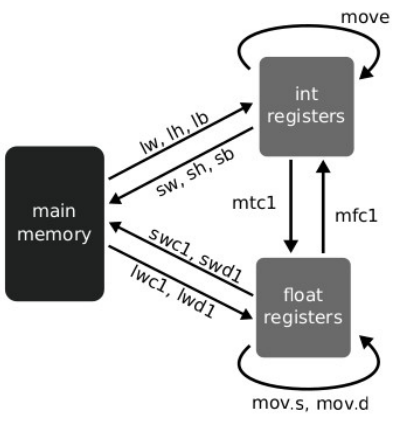
(Adding the single-precision float in the source register `$fs` to the single in the target register `$ft` and place the result in the destination single register `$fd`).

In case we want to do 64-bit (double-precision) calculations we use the same format, but replace the `.s` by `.d`:

```
add.d $fd, $ft, $fs
```

It will add the double in the two *consecutive* registers {`$fs`, `$fs+1`} to the two consecutive registers {`$ft`, `$ft+1`} and place the result in the two consecutive registers {`$fd`, `$fd+1`}.

Summarizing moving bit patters around between memory, integer registers and FP registers:



Write a MIPS program that checks if indeed the above bit pattern is equal to 0.5

Convert (pen and paper) to a hexadecimal bit pattern the following decimal numbers:

- 9.0
- 6.125
- -5/32

What decimal numbers are represented here in IEEE 754:

- 0x42e48000
- 0x00800000
- 0xff8000000
- 0xff80000001

Check your answers in MIPS

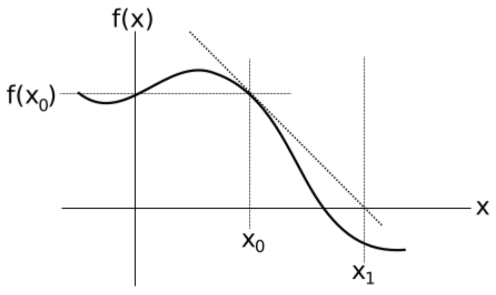


Figure 61: Method of Newton-Raphson for finding zeros in functions. Starting at a point x_0, a new estimation x_1 is found based on the function value and its derivative at x_0, namely $x_1 = x_0 - f(x_0)/f'(x_0)$.

Write a MIPS program with a function that calculates the square-root of the argument (*A*) using the iterative Newton-Raphson method, as described in the book (Section 6.7).

$$x = \mathrm{sqrt}(A)$$

or finding the zero of the function

$$f(x) = x^2 - A = 0$$

Then, iteratively,

$$x_{n+1} = x_n - f(x)/f'(x)$$

$$= x_n - (x_n^2 - A)/2x_n$$

The precision of the calculation given as the second argument of the function.

New instructions for today:

`swc1, swd1, lwc1, lwd1`	Store/load single/double FP to/from memory
`mtc1, mfc1`	Copy from integer register to FP register and back
`mov.s`	Copy single/double FP registers
`add.s, mul.s, div.s`	Airthmetic
`c.eq.s`	Compare if equal singles
`bc1t, bc1f`	Branch if comparing condition is true/false