11 | Floating Point Data & Instructions

Some examples of floating point data include those with decimals or in scientific notation (like 5.2e23). Floating point numbers are used to represent non-integers or any number, including integers, typically numbers large or close to 0.

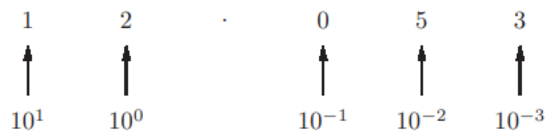

Figure 4.16: Decimal representation of the number 12.053

It works the same in binary (called **fixed point** representation for rational numbers). And below the following is a diagram of fractions.

power of 2	in base 10	in base 2
2^2	4	100_2
2^1	2	10_2
2^0	1	1_2
2^{-1}	$1/2 = 0.5$	0.1_2
2^{-2}	$1/4 = 0.25$	0.01_2
2^{-3}	$1/8 = 0.125$	0.001_2

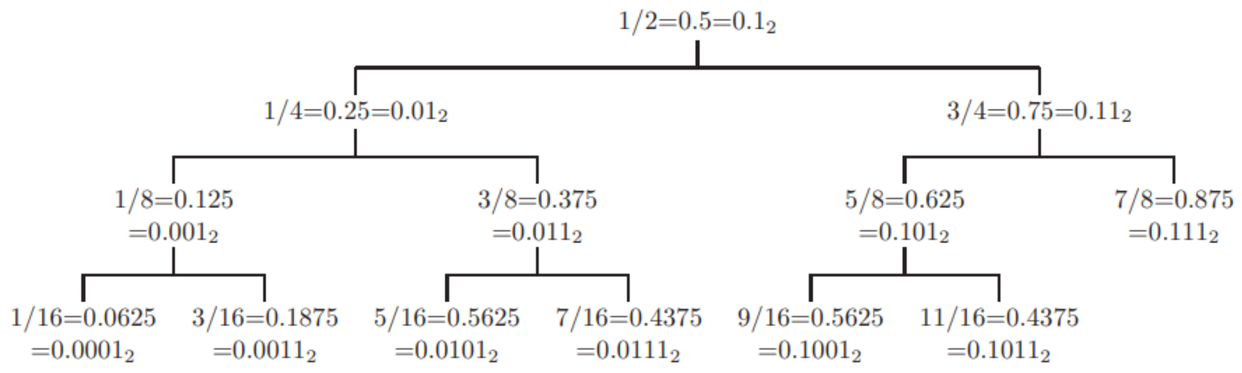

Figure 4.17: Diagram of fractions using binary fixed point representation

mantissa Part before the e in scientific notation (excluding .)

Ex. In 103.07, it's 103; in 5.02e23, it's 5.02; in -45.0, it's 45; in 0.0013e-5, it's 13 This is because you're supposed to exclude leading and trailing 0's and ignore all dots, thus showing everything.

exponent Part after the e

Ex. For 6.02e23, the exponent is 23 with the base as 10.
You can write 103.07 in different ways: 10.307e1 = 1.0307e2 = 1030.7e-1
If you want to increase the number, decrease the exponent (slide decimal →)

normal form 1 digit before the dot

Ex. To write 103.07 in normal form, move the decimal to the left twice to get only one number before the decimal: 1.0307e2; the *mantissa* of this would be 10307

6.02e23 is already in its normal form; its *mantissa* is 602.
-45.0 in normal form is -4.5e1; its *mantissa* is 45 (notice: doesn't include the sign)
0.0013e-95 in normal form is -1.3e-98; its *mantissa* is 13.

Let's do it in binary (take note of above table and diagram shown). For example, 23.375 in decimal can be represented by 10111.011_2 in binary. However, it's important to note some numbers have no exact representation in base 10 (like irrational numbers), and some numbers that have an exact representation in base 10 have none in base 2.

Example: 1/10 = $0.0001100110011001100\ldots_2$ (notice how it keeps repeating)

This can be dangerous in high-level programming (for example, a for loop with a 0.1 increment). A lesson here is not to lose floating point unless you have to (no float or doubles).

IEEE 754 Floating point standard: Single precision (32 bits)

Sign (1 bit) sign of the number; 0 = positive, 1 = negative

Exponent (8 bits) exponent of 2, excess-127 notation, where exponent is the amount by which this 8-bit field exceeds 127 = 0111 1111

Ex: 1000 0011 (this is 131), exponent = 4 (131 - 127)
0111 1110 (this is 126), exponent = -1 (126 - 127)

Fraction (23 bits) drops leading 1 from normalized mantissa (take the mantissa and normalize it, and thus drop the 1 since it's always going to be 1)

Special case: 0.0 = $0000\ 0000_{16}$

Example: 5.25

Sign =	0 (it's a positive number)			
5.25 =	5 ¼	=	101.01_2	(remember, $2^{-2} = ¼$)
Normalize:	101.01 =		$101.01 = 1.0101 * 2^2$	
Fraction:	(drop the 1 after the decimal, so you're left with 0101, and the rest are 0's)			
	0101 0000 0000 0000 0000 000 (remember, a fraction is 23 bits)			
Exponent:	2; in excess-127 notation, exponent = 1000 0001 (129, thus 129 - 127 = 2)			
Result =	**0**	**1000 0001**	**0101 0000 0000 0000 0000 000**	
Group every 4:		**0100 0000 1010 1000 0000 0000 0000 0000**		
Hexadecimal:		**40a8 0000**$_{16}$ (you can check in MARS too using .data instructions)		

Floating Point Instructions (machine language)

Floating point instruction formats include **FR** (floating point register format, used for most instructions) and **FI** (floating point immediate format, used for conditional branch).

Register Format - FR   Example: add.s $fd, $fs, $ft   # $fd = $fs + $ft

Opcode	6 bits		
fmt	5 bits	(op)	
ft	5 bits	(source register)	right operand
fs	5 bits	(source register)	left operand
fd	5 bits	(dest reg)	result
funct	6 bits	(op)	

- Opcode, fmt, and funct form an expanding opcode

opcode	**fmt**	**ft**	**fs**	**fd**	**funct**
31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0

Example: sub.s $f2, $f18, $f6 # $f2 = $f18 - $f6

How did we get the opcode, fmt, and funct? These are included in the Appendix (the numbers in that order according to it are 11, 10, and 1; convert them from hexadecimal to binary to get this). The two registers are 18 (for fs) and 6 (for ft), while 2 is for fd.

opcode	**fmt**	**ft**	**fs**	**fd**	**funct**
01 0001	1 0000	0 0110	1 0010	0 0010	00 0001

32-bit instruction:	0100 0110 0000 0110 1001 0000 1000 0001
Hexadecimal instruction:	**4606 9081**$_{16}$

Memory reference instructions include:

- lwc1 load word coprocessor 1 (floating point)
- swc1 store word coprocessor 1 (floating point)

The word "coprocessor" comes from floating point processor is on a different chip or area of a chip called that name.

I (immediate format)

Example: lwc1 $f2, 8($a0) # load float reg

opcode	**fmt**	**ft**	**immediate**
11 0001	0 0100	0 0010	0000 0000 0000 1000

32-bit instruction:	1100 0100 1000 0010 0000 0000 0000 1000
Hexadecimal instruction:	**c482 0008**$_{16}$

Conditional branch instructions: Two steps

Compare floating point registers

Branch if comparison is true or false

Compare floating point registers, using the FR format Example: c.le.s $fs, $ft # fs <= ft

●	Opcode	6 bits	(always going to be 11_{16})
●	fmt	5 bits	(10_{16} = single vs. 11_{16} = double)
●	ft	5 bits	(2nd source register)
●	fs	5 bits	(1st source register)
●	fd	5 bits	(unused)
●	funct	6 bits	(select comparison: == 32, < 3c, <= 3e)

Example: c.le.s $f8, $f4 # branch if $f8 <= $f4

opcode	**fmt**	**ft**	**fs**	**fd**	**funct**
01 0001	1 0000	0 0100	0 1000	? ????	11 1110

32-bit instruction: 0100 0110 0000 0100 0100 0000 0011 1110
Hexadecimal instruction: **$4604\ 403e_{16}$**

Branch if comparison true or false, using FI format Example: bc1t label

- Opcode 6 bits (11_{16})
- fmt 5 bits (08_{16})
- ft 5 bits (branch on 1 = true, 0 = false)
- immediate 16 bits (relative branch (word) address)

Example:

```
target:
        c.le.s    $f8, $f4
        bc1t      target        # branch to target if comparison is true
```

opcode	**fmt**	**ft**	**immediate**
01 0001	0 1000	0 0001	1111 1111 1111 1110

32-bit instruction: 0100 0101 0000 0001 1111 1111 1111 1110
Hexadecimal instruction: **$4501\ fffe_{16}$**

l