

COMP0068 Computer Architecture & Operating Systems

Lecture 10: High-level Data Types

Background Reading:

Irv Englander (Chapter 3 & 5)

Harris & Harris Section 5.3

Lecture Overview

- We have looked at how MIPS assembly can process a number of key data types:
 - Signed and unsigned 32-bit integer words (integers).
 - Bytes and half-words (shorts).
 - Arrays of integers
- This lecture will look at how high-level data types are handled within the computer:
 - Characters and Strings
 - Real numbers in fixed, and floating point notation.
- We will check that the data layout directives in MARS actually lay out our data in the data segment as we would expect.
- We will create a simple MIPS floating point calculator.

CHARACTER DATA

ASCII - American Standard Code for Information Interchange

Represents Characters

7-bit code for teletype machines
form basis of many existing sets
Basic Unix character set.

NB: 0x30 = '0', 0x31 = '1'
0x41 = 'A', 0x61 = 'a'

Control Characters (00→1F) still used

HT (09 ₁₆)	(horizontal) tab character
LF (0A ₁₆)	line feed character
CR (0D ₁₆)	(carriage) return character
ESC (1B ₁₆)	escape character
BS (08 ₁₆)	backspace character

		000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P	'	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	\$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(8	H	X	h	x
1001	9	HT	EM)	9	I	Y	i	y
1010	10	LF	SUB	*	:	J	Z	j	z
1011	11	VT	ESC	+	;	K	[k	{
1100	12	FF	FS	,	<	L	\	l	
1101	13	CR	GS	-	=	M]	m	}
1110	14	SO	RS	.	>	N	^	n	~
1111	15	SI	US	/	?	O	_	o	DEL

Strings of Characters

- Python class “str” is internally complex and employs Unicode characters.
- But C-language representation of a string is much simpler:

- Assuming ASCII encoding we have ...

Hello !

- Terminated with a null character (0x00).



0x48	0x65	0x6c	0x6c	0x6f	0x20	0x21	0x00
------	------	------	------	------	------	------	------

- Test the MARS ‘.asciiz’ data directive ...
does MARS correctly lay out the memory as expected ?

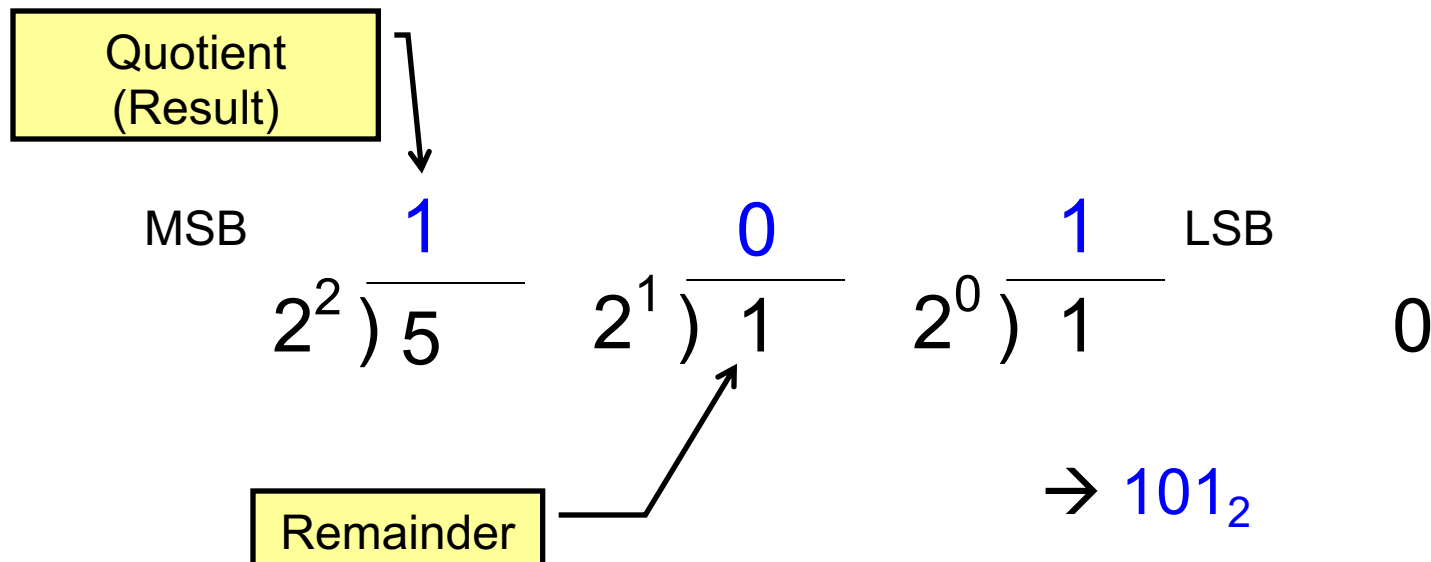
Recall converting binary to decimal ...

2 symbols in binary – 0,1

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$$

And decimal to binary ...

One way of converting from decimal to binary is by first dividing by the largest power of two ... and then successively dividing by smaller ones ...



Representation of Real Numbers

The power series representation of integers can be extended to provide for real numbers:-

$$R = \dots + s_n 2^n + \dots s_3 2^3 + s_2 2^2 + s_1 2^1 + s_0 2^0 + s_{-1} 2^{-1} + s_{-2} 2^{-2} + \dots + s_{-m} 2^{-m} + \dots$$

The series

$$s_{-1} 2^{-1} + s_{-2} 2^{-2} + \dots + s_{-m} 2^{-m} + \dots$$

provides the fractional part of the number

where $s_{-1} s_{-2} \dots s_{-m}$ have to be binary digits equal to 0 or 1.

$$2^{-1} = 1/2 \text{ or } 0.5 \quad 2^{-2} = 1/4 \text{ or } 0.25 \quad 2^{-3} = 1/8 \text{ or } 0.125$$

$$3.625_{10} = 11.101_2$$

$$11.101_2 = 1 \cdot 2 + 1 \cdot 1 + 1 \cdot 0.5 + 0 \cdot 0.25 + 1 \cdot 0.125 = 3.625_{10}$$

Decimal Real Numbers → Binary

Again fairly easy from binary to decimal:

$$11.101_2 = 1*2 + 1*1 + 1*0.5 + 0*0.25 + 1*0.125 = 3.625_{10}$$

Decimal to binary – do the same but continue when we get to $\frac{1}{2}$, $\frac{1}{4}$, etc.

$$5.75_{10} = (?)_2$$

- a) Conversion of the integer part: same as before – repeated division by 2

$$5_{10} = 101_2$$

- b) Conversion of the fractional part: perform a repeated multiplication by 2 and extract the integer part of the result

$$\begin{array}{llll} 0.75 \times 2 & = & 1.5 & \Rightarrow \text{extract } 1 \\ 0.5 \times 2 & = & 1.0 & \Rightarrow \text{extract } 1 \\ 0 & & & \Rightarrow \text{stop} \end{array} \quad \downarrow$$

$$0.75_{10} = 0.11_2$$

Combine the results from integer and fractional part, $5.75_{10} = 101.11_2$

Fixed Point Representation of Real Numbers

This holds the n bit symbols surrounding the binary point
 in an n -bit integer and uses integer ops to do maths

i.e. $S_{n-m-1} \dots S_1 S_0 \cdot S_{-1} S_{-2} \dots S_{-m}$

i.e. The real number	0 0 0 0 0 1 . 1 0 (decimal 1.5)
is treated like	0 0 0 0 0 1 1 0 (decimal 6 !)

The position of the fixed point is not recorded anywhere –
 the software has to remember where it is.

Fixed Point Examples

Let register A hold 1.1_2 and register B hold 1.0_2

Assume 12-bit fixed point with 4 fractional bits:


point between bits 3 & 4

A = 000000011000 & B = 000000010000

 
position of fractional point

A+B =

000000011000
000000010000
000000101000


= 2.5_{10}


A * B =

000000011000
000000010000
000000000000000110000000

Result of integer multiply

Need to shift right by 4 to lose last 4 zeroes

→ 000000011000


2's Complement of Real Numbers

Complement all the bits in the usual way and
 then add 1 to the **rightmost fractional bit**

Example: $1.5_{10} = 1.1_2$

$$\begin{aligned}
 -1.1_2 &= -00001.1000_2 \\
 &= 11110.0111_2 + 00000.0001_2 \quad \text{complement the bits} \\
 &= 11110.1000_2
 \end{aligned}$$

$$1.1 + (-1.1) = 0$$

Check the result:-

$$\begin{array}{r}
 \dots 000001.1_2 \\
 \underline{\dots 111110.1_2} \\
 \dots 000000.0
 \end{array}$$

ADDITION OF 1 GOES INTO LEAST SIGNIFICANT BIT

Going from fixed point real numbers to floating point format numbers ... reviewing Exponential (or Scientific) Notation.

We know that a number like 12345 can be represented as:

$$12345 = 1.2345 * 10^4$$

Any number can be represented in exponential or scientific notation:

$$(\text{sign}) (\text{mantissa}) * (\text{base})^{\text{exponent}}$$

We use a base equal to 10 for decimal numbers.

We can always make the mantissa lie between 1.0 (inclusive) and 10.0 (exclusive) by adjusting the exponent appropriately:

$$-0.0012345 = -1.2345 * 10^{-3}$$

Exponential Notation for Binary Numbers

Analogous results apply for the binary ... in this case the base for the exponential notation is 2 :

$$10010 = 1.0010 * 2^4 \quad \text{and} \quad -0.0011 = -1.1 * 2^{-3}$$

Format needs to consider the sign of the number:

$$(\text{sign}) (\text{mantissa}) * 2^{\text{exponent}}$$

Again we can always make the mantissa lie between 1.0 and 2.0 (exclusive) by adjusting the exponent appropriately ... in this case (because it's binary) ... the mantissa would always start with a 1 ... ***unless?***

Floating Point Representation of Real Numbers

IEEE 754 floating point standard represents a real number, X , by putting X into the normalized form:

$$X = (-1)^S * 1.F * 2^{E - \text{Bias}} \quad \text{What ?}$$

S is the sign bit:

S = 0 for a positive number and **s = 1** for a negative number, since $(-1)^0 = 1$ while $(-1)^1 = -1$

F is all the bits **to the right of the leftmost 1** in the mantissa (get greater accuracy from our bits if we assume a leading 1)

The **Bias** is fixed integer (**127, 1023, ..**)

And **(E – Bias)** is the exponent.

Why this form?

S, E and F are used to represent X

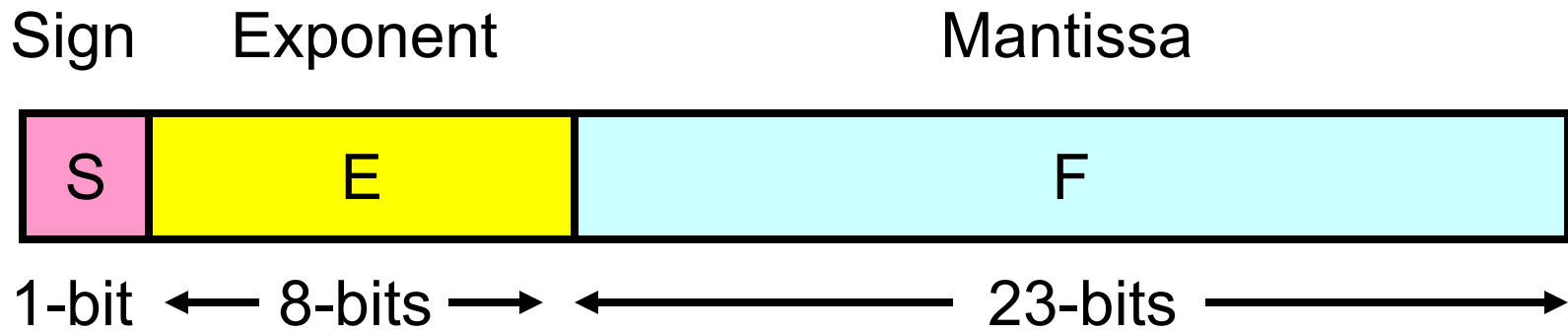
Floating Point – IEEE single precision format

IEEE single precision is a 32-bit format that is equivalent to the Java or C-language ‘float’ data type.

Sign is represented by a single bit (0=positive, 1=negative)

The exponent is a 8- bit number biased by 127_{10}

The mantissa consists of 23- bits (with an implicit leading 1)



IEEE Single Precision Format Example

Turn -12.75_{10} into single precision form.

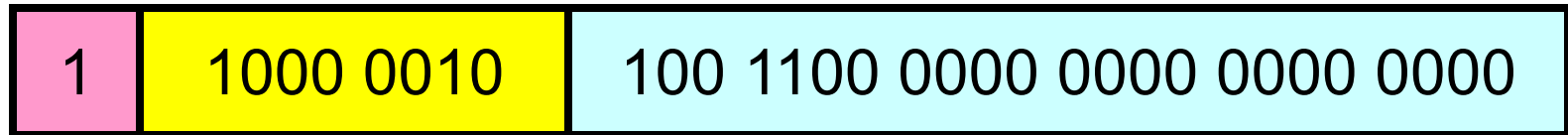
$$-12.75_{10} = -1100.11_2 = -1.10011_2 * 2^3 = (-1)^1 * 1.10011_2 * 2^{130-127}$$

not 2's complement

$$S = 1 \quad E = 130_{10} = 10000010_2 \quad F = 1001100...00..0$$



1-bit ← 8-bits → ← 23-bits →



→ 1100 0001 0100 1100 0000 0000 0000 0000

→ **C14C0000**₁₆

The IEEE Single Precision Floating Point Format is Complex – 5 Minute Activity

- Someone come up with a floating point number.
- Spend 5 minutes trying to come up with the IEEE single precision 32-bit format of the floating point number in hexadecimal.
- Work through the calculation on the whiteboard.
- We will then test your manual result using MARS

IEEE Single Precision range

- Smallest (absolute value) normalized number is:

$$1.000000000000000000000000000000 \times 2^{-126}$$

$$\text{Min } E = 0000\ 0001_2$$

- Largest normalized number is:

$$1.111111111111111111111111111111 \times 2^{127} \sim 2 * 2^{127} = 2^{128}$$

$$\text{Max } E = 1111\ 1110_2$$

Not all numbers can be handled in Normalised Form ...

Denormalised Numbers: very small numbers and zero

YES –
that's zero

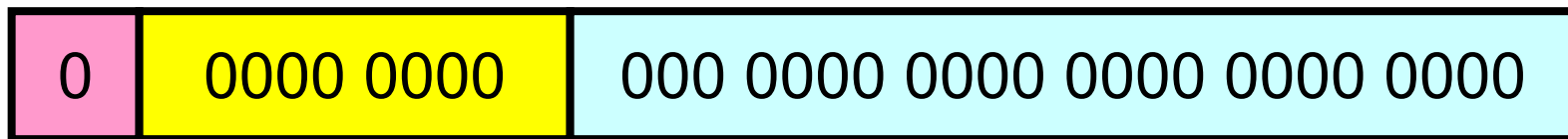
YES –
plus one

Representation: $X = (-1)^S * 0.F * 2^{E+1-Bias} : E=0$

$E = 0$ identifies a denormalised number : F is 23-bits

Zero is: $(-1)^0 * 2^{0+1-Bias} * 0.0000000000$

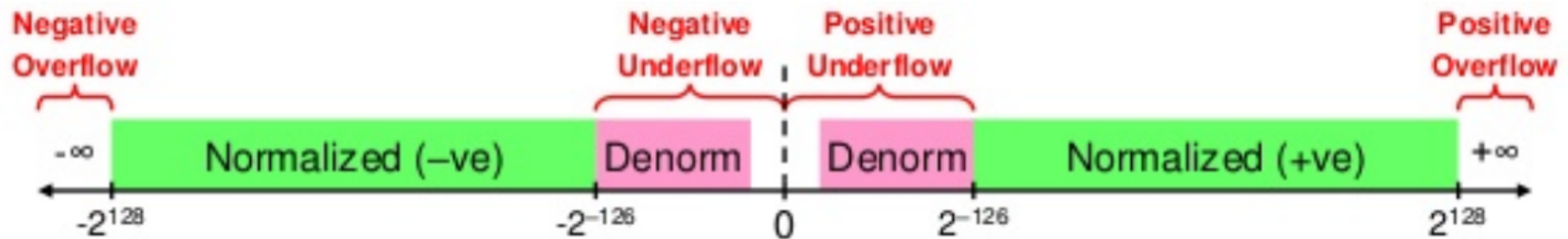
$S = 0$ $E = 0 = 00000000_2$ $F = 0000000...00..0$



Floating point zero is same as integer 0!

Normalised and denormalised numbers

- Denormalised numbers are used to :
 1. Fill the gap between 0 and the smallest normalised float
 2. Provide gradual underflow to zero



Special cases: Zero, Infinity, and NaN

Zero

- Exponent field $E = 0$ and $F = 0$

Infinity

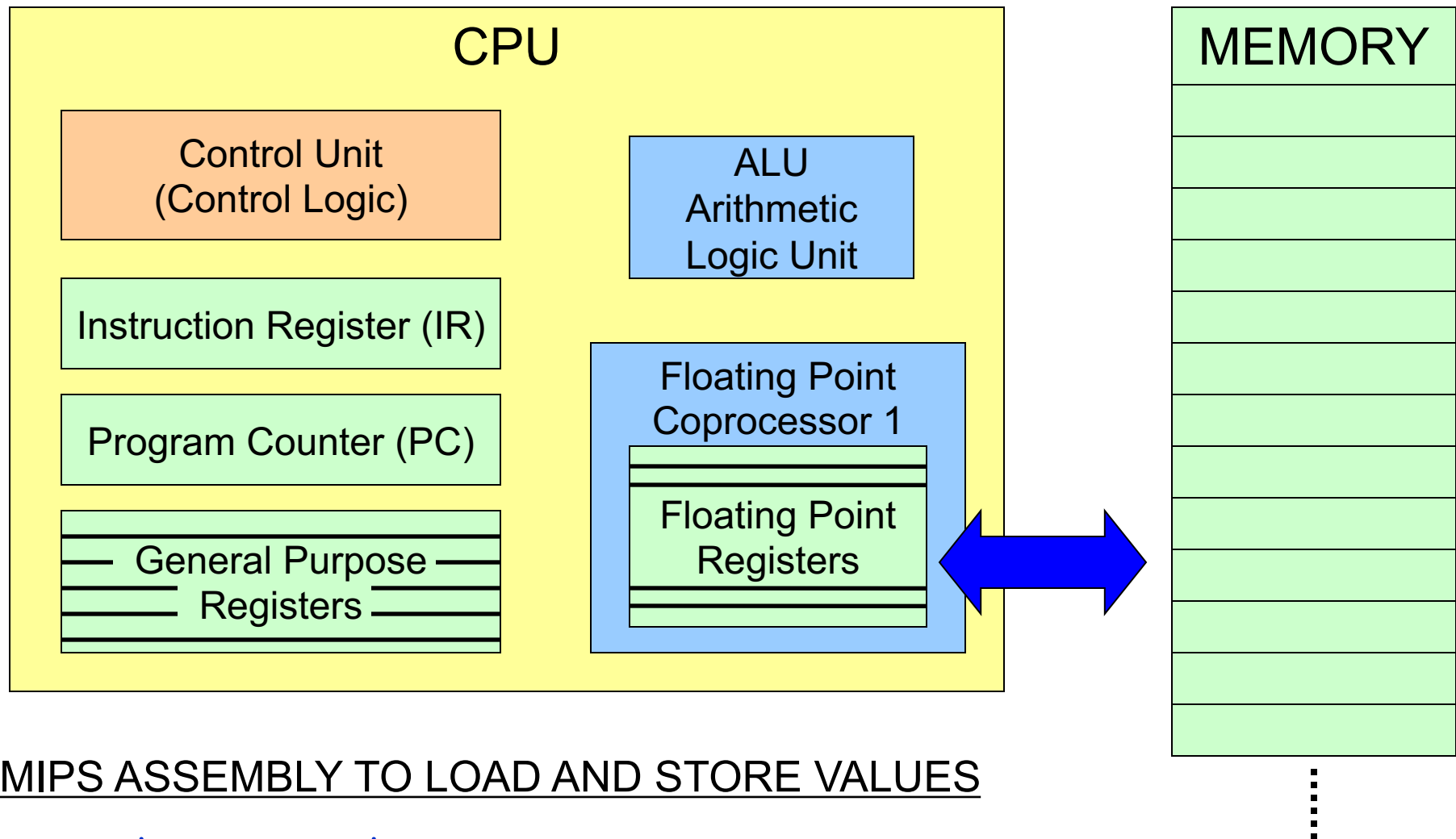
- Infinity is a special value represented with **maximum E** and **$F = 0$**
- For single precision with 8 bit exponent: maximum $E = 255$
- Infinity can result from overflow
- $+\infty$ and $-\infty$ are possible according to sign bit S

NaN (Not a Number)

- NaN is a special value represented with **maximum E** and **$F \neq 0$ (since $00..$ represent infinity)**
- Result from exceptional situations, such as $0/0$ or $\text{sqrt}(-ve)$
- Operation on a NaN results is NaN: $\text{Op}(X, \text{NaN}) = \text{NaN}$

MIPS has a floating point co-processor^{UCL}

built in with 32 floating point registers



MIPS ASSEMBLY TO LOAD AND STORE VALUES

lwc1 \$f1, 100(\$s2) Load word to coprocessor 1.
swc1 \$f1, 100(\$s2) Store word from coprocessor 1.

MIPS Floating Point Instructions

Instruction	Description
<code>add.s \$f2, \$f4, \$f6</code>	FP add (single precision)
<code>sub.s \$f2, \$f4, \$f6</code>	FP subtract (single precision)
<code>mul.s \$f2, \$f4, \$f6</code>	FP multiple (single precision)
<code>div.s \$f2, \$f4, \$f6</code>	FP divide (single precision)
<code>add.d \$f2, \$f4, \$f6</code>	FP add (double precision)
<code>sub.d \$f2, \$f4, \$f6</code>	FP subtract (double precision)
<code>mul.d \$f2, \$f4, \$f6</code>	FP multiple (double precision)
<code>div.d \$f2, \$f4, \$f6</code>	FP divide (double precision)

Floating point registers go from \$f0 to \$f31.

Double precision is 64-bit and uses 2 registers in a row (hence the values \$f0, \$f2, \$f4, etc. are only allowed)

Floating Point Calculator

```
# Floating-Point Calculator Version 2.0

        .data
vals:    .float 3.14, 10.0, 0.0

        .text
        .globl main

main:

        la    $4, vals      # Load $4 with data address.
        lwc1  $f0, 0($4)    # Load first 32-bit float.
        lwc1  $f1, 4($4)    # Load second 32-bit float.
        mul.s $f2, $f0, $f1 # Multiple floats.
        swc1  $f2, 8($4)    # Save result to memory.

        li    $v0 10        # Exit program using
        syscall             # syscall 10 (exit)
```

See how it works in MARS

Lecture Summary

- This lecture has been about storing different types of data in memory including characters, strings and real numbers (in both fixed point format and floating point format).
- We looked in detail at the IEEE single-precision format but the double precision format is similar with more bits available for mantissa and exponent.
- We have looked at normalized format ($0 < E < 255$) and denormalized format ($E = 0$).
- We have looked at the representation of special cases: zero, +/-infinity and NaN
- We introduced the FPU (Floating Point Unit) of the MIPS CPU and creating a floating point calculator.