MIPS floating-point arithmetic

ƒ Floating-point computations are vital for many applications, but correct implementation of floating-point hardware and software is very tricky.

ƒ Today we'll study the IEEE 754 standard for floating-point arithmetic.

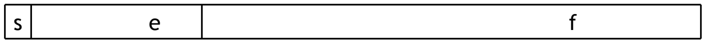
- Floating-point number representations are complex, but limited.
- Addition and multiplication operations require several steps.
- The MIPS architecture includes support for floating-point arithmetic.

Floating-point representation

ƒ IEEE numbers are stored using a kind of scientific notation.

$$\pm \text{mantissa} \times 2^{\text{exponent}}$$

ƒ We can represent floating-point numbers with three binary fields: a sign bit s, an exponent field e, and a fraction field f.



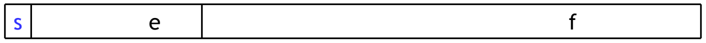
ƒ The IEEE 754 standard defines several different precisions.

— Single precision numbers include an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.

- Double precision numbers have an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.

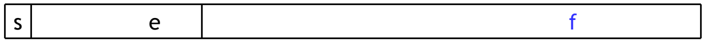
ƒ There are also various extended precision formats. For example, Intel processors use an 80-bit format internally.

Sign



- The sign bit is 0 for positive numbers and 1 for negative numbers.
- But unlike integers, IEEE values are stored in signed magnitude format.

Mantissa



- The field f contains a binary fraction.
- The actual mantissa of the floating-point value is (1 + f).
  - In other words, there is an implicit 1 to the left of the binary point.
  - For example, if f is 01101…, the mantissa would be 1.01101…
- There are many ways to write a number in scientific notation, but there is always a *unique* normalized representation, with exactly one non-zero digit to the left of the point.

$$0.232 \times 10^3 = 23.2 \times 10^1 = 2.32 \times 10^2 = \ldots$$

- A side effect is that we get a little more precision: there are 24 bits in the mantissa, but we only need to store 23 of them.

Exponent

s	e	f

- The e field represents the exponent as a biased number.
  - It contains the actual exponent *plus* 127 for single precision, or the actual exponent *plus* 1023 in double precision.
  - This converts all single-precision exponents from −127 to +127 into unsigned numbers from 0 to 255, and all double-precision exponents from −1024 to +1023 into unsigned numbers from 0 to 2047.
- Two examples with single-precision numbers are shown below.
  - If the exponent is 4, the e field will be 4 + 127 = 131 (10000011_2).
  - If e contains 01011101 (93_{10}), the actual exponent is 93 − 127 = −34.

Converting an IEEE 754 number to decimal

s	e	f

- The decimal value of an IEEE number is given by the formula:

$$(1 - 2s) \times (1 + f) \times 2^{e\text{-bias}}$$

- Here, the s, f and e fields are assumed to be in decimal.
  - (1 − 2s) is 1 or −1, depending on whether the sign bit is 0 or 1.
  - We add an implicit 1 to the fraction field f, as mentioned earlier.
  - Again, the bias is either 127 or 1023, for single or double precision.

Example IEEE-decimal conversion

- Let’s find the decimal value of the following IEEE number.

1 01111100 11000000000000000000000

- First convert each individual field to decimal.
  - The sign bit s is 1.
  - The e field contains 01111100 = 124_{10}.
  - The mantissa is 0.11000… = 0.75_{10}.
- Then just plug these decimal values of s, e and f into our formula.

$$2^{e-bias} \quad (1 - 2s) \times (1 + f) \times$$

- This gives us $(1 - 2) \times (1 + 0.75) \times 2^{124-127} = (-1.75 \times 2^{-3}) = -0.21875$.

Converting a decimal number to IEEE 754

- What is the single-precision representation of 347.625?

1. First convert the number to binary: 347.625 = 101011011.101_2.
2. Normalize the number by shifting the binary point until there is a single 1 to the left:

$$101011011.101 \times 2^0 = 1.01011011101 \times 2^8$$

3. The bits to the right of the binary point, 01011011101_2, comprise the fractional field f.
4. The number of times you shifted gives the exponent. In this case, the field e should contain 8 + 127 = 135 = 10000111_2.
5. The number is positive, so the sign bit is 0.

- The final result is:

0 10000111 01011011101000000000000

Special values

ƒ The smallest and largest possible exponents e=00000000 and e=11111111 (and their double precision counterparts) are reserved for special values.

ƒ If the mantissa is always (1 + f), then how is 0 represented?

  – The fraction field f should be 0000...0000.

  – The exponent field e contains the value 00000000.

  – With signed magnitude, there are *two* zeroes: +0.0 and –0.0.

ƒ There are representations of positive and negative infinity, which might sometimes help with instances of overflow.

– The fraction f is 0000...0000.

– The exponent field e is set to 11111111.

ƒ Finally, there is a special “not a number” value, which can handle some cases of errors or invalid operations such as 0.0/0.0.

– The fraction field f is set to any non-zero value.

– The exponent e will contain 11111111.

Limits of the IEEE representation

ƒEven some integers cannot be represented in the IEEE format.

```
int x   = 33554431;
float y = 33554431;
printf( "%d\n", x );
printf( "%f\n", y );
```

ƒSome simple decimal numbers cannot be represented exactly in binary to begin with.

$$0.10_{10} = 0.0001100110011\ldots_2$$

0.10

ƒ During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.

ƒ A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.

– The Patriot incremented a counter once every 0.10 seconds.

– It multiplied the counter value by 0.10 to compute the actual time.

ƒ However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.

ƒ This doesn't seem like much, but after 100 hours the time ends up being off by 0.34 seconds—enough time for a Scud to travel 500 meters!

ƒ Professor Skeel wrote a short article about this.

Roundoff Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.

Floating-point addition example

ƒ To get a feel for floating-point operations, we'll do an addition example.

– To keep it simple, we'll use base 10 scientific notation.

– Assume the mantissa has four digits, and the exponent has one digit.

ƒ The text shows an example for the addition:

99.99 + 0.161 = 100.151

ƒAs normalized numbers, the operands would be written as:

9.999×10^{1} 1.610×10^{-1}

Steps 1-2: the actual addition

1. Equalize the exponents.

   The operand with the smaller exponent should be rewritten by increasing its exponent and shifting the point leftwards.

   $$1.610 \times 10^{-1} = 0.0161 \times 10^{1}$$

   With four significant digits, this gets rounded to 0.016×10^{1}.

   This can result in a loss of least significant digits—the rightmost 1 in this case. But rewriting the number with the larger exponent could result in loss of the *most* significant digits, which is much worse.

2. Add the mantissas.

$$\begin{array}{rrcl} & 9.999 & \times & 10^{1} \\ + & 0.016 & \times & 10^{1} \\ \hline & 10.015 & \times & 10^{1} \end{array}$$

Steps 3-5: representing the result

3. Normalize the result if necessary.

   $$10.015 \times 10^{1} = 1.0015 \times 10^{2}$$

   This step may cause the point to shift either left or right, and the exponent to either increase or decrease.

4. Round the number if needed.

   1.0015×10^{2} gets rounded to 1.002×10^{2}.

5. Repeat Step 3 if the result is no longer normalized.

   We don't need this in our example, but it's possible for rounding to add digits—for example, rounding 9.9995 yields 10.000.

Our result is 1.002×10^{2}, or 100.2. The correct answer is 100.151, so we have the right answer to four significant digits, but there's a small error already.

Extreme errors

ƒ As we saw, rounding errors in addition can occur if one argument is much smaller than the other, since we need to match the exponents.

ƒ An extreme example with 32-bit IEEE values is the following.

$$(1.5 \times 10^{38}) + (1.0 \times 10^{0}) = 1.5 \times 10^{38}$$

The number 1.0×10^{0} is much smaller than 1.5×10^{38}, and it basically gets rounded out of existence.

ƒ This has some nasty implications. The order in which you do additions can affect the result, so (x + y) + z is not always the same as x + (y + z)!

```
float x = -1.5e38;
float y =  1.5e38;
printf( “%f\n”, (x + y) + 1.0 );
printf( “%f\n”, x + (y + 1.0) );
```

The history of floating-point computation

ƒ In the past, each machine had its own implementation of floating-point arithmetic hardware and/or software.

— It was impossible to write portable programs that would produce the same results on different systems.

— Many strange tricks were needed to get correct answers out of some machines, such as Crays or the IBM System 370.

ƒ It wasn’t until 1985 that the IEEE 754 standard was adopted.

- The standard is very complex and difficult to implement efficiently.
- But having a standard at least ensures that all compliant machines will produce the same outputs for the same program.

Floating-point hardware

- Intel introduced the 8087 coprocessor around 1981.
  - The main CPU would call the 8087 for floating-point operations.
  - The 8087 had eight separate 80-bit floating-point registers that could be accessed in a stack-like fashion.
  - Some of the IEEE standard is based on the 8087.
- Intel’s 80486, introduced in 1989, included floating-point support in the main processor itself.
- The MIPS floating-point architecture and instruction set still reflect the old coprocessor days, with separate floating-point registers and special instructions for accessing those registers.

MIPS floating-point architecture

- MIPS includes a separate set of 32 floating-point registers, `$f0-$f31`.
  - Each register is 32 bits long and can hold a single-precision value.
  - Two registers can be combined to store a double-precision number. You can have up to 16 double-precision values in registers $f0-$f1, $f2-$f3, ..., $f30-$f31.
  - $f0 is *not* hardwired to the value 0.0!
- There are also separate instructions for floating-point arithmetic. The operands *must* be floating-point registers, and not immediate values.

```
add.s  $f1, $f2, $f3   # Single-precision $f1 = $f2 + $f3
```

- There are other basic operations as you would expect.
  - `sub.s` for subtraction
  - `mul.s` for multiplication
  - `div.s` for division

Floating-point register transfers

- mov.s and mov.d copy data between floating-point registers.
- Use mtc1 and mfc1 to transfer data between the integer registers $0-$31 and the floating-point registers $f0-$f31.
  - Be careful with the order of the operands in these instructions.

```
mtc1  $t0, $f0       # $f0 = $t0
mfc1  $t0, $f0       # $t0 = $f0
```

- There are also special loads and stores for transferring data between the floating-point registers and memory. (The base address is still given in an integer register.)

```
lwc1  $f2, 0($a0)    # $f2 = M[$a0]
swc1  $f4, 4($sp)    # M[$sp+4] = $f4
```

- The “c1” in the instruction names stands for “coprocessor 1.”

Floating-point comparisons

- We also need special instructions for comparing floating-point values, since slt and sltu only apply to signed and unsigned integers.

```
c.le.s  $f2, $f4
c.eq.s  $f2, $f4
c.lt.s  $f2, $f4
```

- The comparison result is stored in a *special* coprocessor register.
- You can then branch based on whether this register contains 1 or 0.

```
bc1t  Label     # branch if true
bc1f  Label     # branch if false
```

- Here is how you can branch to the label Exit if $f2 = $f4.

```
c.eq.s  $f2, $f4
bc1t    Exit
```

Floating-point functions

ƒ There are conventions for passing data to and from functions.
– Floating-point arguments are placed in $f12-$f15.
– Floating-point return values go into $f0-$f1.
ƒ We also split the register-saving chores, just like earlier.
– $f0-$f19 are caller-saved.
– $f20-$f31 are callee-saved.
ƒThese are the same basic ideas as before because we still have the same problems to solve—now it's just with different registers.

Floating-point constants

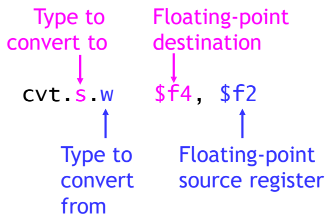
ƒ MIPS does not support immediate floating-point arithmetic instructions, so you must load constant values into a floating-point register first.

ƒ ƒ Newer versions of MIPS simulators support the li.s and li.d pseudo-instructions,

```
li.s $f6, 0.55555  # $f6 = 0.55555
```

Type conversions

- You can cast integers to floating-point values using the MIPS type conversion instructions.


Type to convert to

Floating-point destination

`cvt.s.w $f4, $f2`

Type to convert from

Floating-point source register

- Possible types for conversions are integers (w), single-precision (s) and double-precision (d) floating-point.

```
li      $t0, 32      # $t0 = 32
mtc1    $t0, $f2     # $f2 = 32
cvt.s.w $f4, $f2     # $f4 = 32.0
```

A complete example

- Here is a slightly different version of the textbook example of converting single-precision temperatures from Fahrenheit to Celsius.

celsius = (fahrenheit – 32.0) × 5.0 / 9.0

```
celsius:
li      $t0, 32
mtc1    $t0, $f4
cvt.s.w $f4, $f4          # $f4 = 32.0
li.s    $f6, 0.55555      # $f6 = 5.0 / 9.0
sub.s   $f0, $f12, $f4    # $f0 = $f12 - 32.0
mul.s   $f0, $f0, $f6     # $f0 = $f0 * 5.0/9.0
jr      $ra
```

- This example demonstrates a couple of things.
  - The argument is passed in $f12, and the return value is placed in $f0.
  - We use two different ways of loading floating-point constants.
  - We used only caller-saved floating-point registers.

Summary

ƒ The IEEE 754 standard defines number representations and operations for floating-point arithmetic.

ƒ Having a finite number of bits means we can't represent all possible real numbers, and errors will occur from approximations.

ƒ MIPS processors implement the IEEE 754 standard.

- There is a separate set of floating-point registers, $f0-$f31.
- New instructions handle basic floating-point operations, comparisons and branches. There is also support for transferring data between the floating-point registers, main memory and the integer registers.
- We still have to deal with issues of argument and result passing, and register saving and restoring in function calls.