Floating Point

- What can be represented in N bits?
  - Unsigned: 0 to $2^N - 1$
  - 2’s Complement: - $2^{N-1}$ to $2^{N-1} - 1$
- But, what about--
  - Very large numbers?
    - 9,349,398,989,762,448,859,087,678
    - $1.23 \times 10^{37}$
  - Very small numbers?
    - $0.0000000000000000000000000000045691$
    - $2.98 \times 10^{-32}$
  - Fractional values? 0.35
  - Mixed numbers? 10.28
  - Irrationals? $\pi$
Why Floating Point?

- Floating point representation makes numerical computation much easier. You could write all your programs using integers or fixed-point representations, but this is tedious and error-prone. For example, you could write a program with the understanding that all integers in the program are 100 times bigger than the number they represent. The integer 2345, for example, would represent the number 23.45. As long as you are consistent, everything works.
- This is actually the same as using fixed point notation. In fixed point binary notation, the binary point is assumed to lie between two of the bits. This is the same as an understanding that a particular power of two multiplies the integer represented by the bits.
- But it is very hard to stay consistent. A programmer must remember where the decimal (or binary) point "really is" in each number. Sometimes one program needs to deal with several different ranges of numbers. Consider a program that must deal with both the measurements that describe the dimensions on a silicon chip (say 0.000000010 to 0.0000010000 meters) and also the speed of electrical signals, 10000000.0 to 300000000.0 meters/second. It is hard to find a place to fix the decimal point so that all these values can be represented.
- Notice that in writing those numbers (0.000000010, 0.0000010000, 10000000.0, 300000000.0), the decimal point was put where it was needed.

Floating Point

- The essential idea of floating point representation is that a fixed number of bits are used (usually 32 or 64) and that the binary point "floats" to where it is needed in those bits. Of course, the computer only holds bit patterns. Some of the bits of a floating point representation must be used to say where the binary point lies.
- Floating point expressions can represent numbers that are very small and numbers that are very large. When a floating point calculation is performed, the binary point floats to the correct position in the result. The programmer does not need to explicitly keep track of it.
Early Floating Point

- The first digital computer (according to some), the Z1, built by Konrad Zuse in 1938 was also the first computer to use floating point representation. Early computers were built to do engineering and scientific calculation so it is no surprise that the invention of floating point happened at the same time. In later years it was realized that computers are very useful things, and not just for calculation. Often floating point was not supported.

- Many early minicomputers and microprocessors did not directly support floating point in hardware. For example, Intel processor chips before the 80486 did not directly support floating point. A floating point operation in a program (written in C, say) was compiled into a sequence of bit-manipulation instructions that did the required operations. Computers used for graphics or engineering calculation often had an additional chip, called a co-processor, that performed floating point operations in hardware.

- MIPS processors are very strong on floating point, and have supported the IEEE standard from their (and its) beginning. MIPS chips are often used in high-end engineering and graphics workstations and are famous for their fast floating point operations.

Scientific Notation

- Up until 1985, each hardware manufacturer had their own type of floating point. Worse, different machines from the same manufacturer might have different types of floating point! And when floating point was not supported in the hardware, the different compilers emulated different floating point types.

- To address this problem the IEEE (Institute of Electrical and Electronics Engineers) created a standard for floating point. This is the IEEE 754 standard, released in 1985 after many years of development. All hardware and software since then follow this standard.

- The idea of floating point comes from scientific notation for numbers. You have probably seen this before. Here is a number in scientific notation:

\[ 1.38502 \times 10^3 \]

- The **mantissa** is 1.38502 and always has a decimal point after the first digit. The **exponent** is 03. The above expression means:

\[ 1.38502 \times 1000 = 1385.02 \]

- The decimal point "floats" to where it belongs. In effect, the mantissa gives the digits and the exponent says where the decimal point should go.
Scientific Notation

- Writing the multiplication symbol (\times) and the superscript exponent is tedious, and not suitable for programs. A slightly altered notation is used in computer programs and in input and output text:

\[
1.38502 \times 10^{+03} = 1.38502e+03 \\
1.38502 \times 10^{-03} = 1.38502e-03 \\
-1.38502 \times 10^{+03} = -1.38502e+03 \\
-1.38502 \times 10^{-03} = -1.38502e-03
\]

- Caution: For QTSpim, only a small "e" is allowed in floating point constants. Some software systems, such as Java, allow both upper and lower case e's.

Fixed Number of Digits

- Each expression on the previous page has eight digits. Two of the digits are used to "float" the decimal point. The remaining six show the digits of the number.

- The number of digits in the mantissa is called the precision of the floating point number. When we start talking about binary floating point, the precision will be the number of bits in the mantissa. With a 32-bit single-precision floating point value the mantissa is 24 bits, so single precision floats have 24 bits of precision.

- Caution: sometimes in casual speech people say that single precision values have 32 bits of precision, but this is not actually correct.

- Another caution: the mantissa uses 23 bits to represent its 24 bits of precision. This trick will be discussed shortly.
IEEE 754 Floating Point

- An IEEE 754 single-precision floating point value is 32 bits long. The bits are divided into fixed-sized fields as shown below.
- Bits 0 through 22 are for the mantissa; bits 23 through 30 are for the exponent; and bit 31 is the sign bit. The mantissa and exponent fields work like the similar parts in scientific notation. The sign bit gives the sign of the entire expression: a 0 bit means positive and a 1 bit means negative.

![IEEE 754 Single Precision Format Diagram]

More on the Mantissa

- The mantissa represents a number in 24-bit base two positional notation that looks like this:
  \[1.xxxxxxxxxxxxxxxxxxxxxx\] (x = 0 or 1; there are 23 x's)
- **Very tricky:** The mantissa represents the 23-bit binary fraction part of a 24-bit number that starts with a "1". The 2\(^{nd}\) place (the one's place) is presumed to contain a 1 and is not present in the mantissa. This trick gives us 24 bits of precision with only 23 bits.
- For example, the binary number 1.111100001011011010001 is represented as 111100001011011010001
How the Mantissa Works

- Since leading zeros of a binary number can be dropped, it is safe to not actually include the first 1. There will always be one, so there is no need to show it. The exponent is adjusted so that the most significant 1 is in the one's place of the mantissa. For example: say that you wish to represent 00011.00101101 as a float.
  - Drop the leading zeros:  11.00101101
  - Now there is a leading one.
  - Move the binary point so that the leading one is in the one's place (the exponent shows the move):
    \[1.100101101 \times 2^1\]
  - Drop the leading one: 100101101
  - Add zero bits on the right so that there are 23 bits in total:
    \[10010110100000000000000 \times 2^1\]
  - The mantissa is: 10010110100000000000000

The Exponent

- The eight bits 23 through 30 contain the exponent. The exponent is an integer, but may be negative, zero, or positive. You might guess that an 8-bit two's complement integer would work perfectly for this, but a different type of notation is used.
- The exponent is expressed using a biased integer. The true exponent is added to +127 \(_{10}\) resulting in an unsigned binary integer. The biased integer is encoded using 8-bit unsigned binary.
  - A biased exponent of +127 represents the actual exponent 0.
  - A biased exponent of +128 represents the actual exponent 1.
  - A biased exponent of +126 represents the actual exponent -1.
- Exceptions: (1) A floating point value of 0.0 is represented with a mantissa of 23 zero bits and an exponent of 8 zero bits. It is not legal to have an exponent of zero with a non-zero mantissa. (2) The exponent 255 \(_{10}\) (1111 1111) is used to signal various problems, such as division by zero.
**Float Formula**

- Here is a formula that summarizes the past several pages. In it, s is the sign bit (0 or 1), M is the mantissa (000...000 to 1111...111) and E is the biased exponent.

\[ \text{value} = (-1)^s \times 1.M \times 2^{E-127} \]

- To convert a paper-and-pencil number into IEEE floating point, fill in each piece of the picture. Let us represent 1.0 as a 32-bit IEEE 754 float.

**Questions**

- What is the sign bit of 1.0?
- What is the mantissa of 1.0? (Hint: remember the assumed "1." preceding the mantissa)
- What is the actual exponent of 1.0?
- What is the biased exponent of 1.0? (Hint: add the actual exponent to 127)

- What is the sign bit of 1.0? 0 (for positive)
- What is the mantissa of 1.0? 000 0000 0000 0000 0000 0000 0000
- What is the actual exponent of 1.0? 0
- What is the biased exponent of 1.0? \(127_{10} = 0111 1111_2\)
1.0

- If the bit pattern is written in hexadecimal, it is 0x3F800000. To check that this is correct, use QTSpim. Write a program that declares a floating point 1.0, then run QTSpim.

```plaintext
## Program to represent 1.0
.text
.globl main
main:
.data
val: .float 1.0
## End of file
```

QTSpim Data

- Of course, to the memory system a 32-bit floating point value is just another 32 bit pattern. Here is (part) of QTSpim's display showing this:

```
DATA
[0x10000000]...[0x10000000] 0x00000000
[0x10000000] 0x00000000
[0x10000000] 0x3F800000 0x00000000 0x00000000 0x00000000
[0x10000000] 0x10000000 0x00000000
```

- Can -0.1 be represented precisely as a 32-bit float?
-0.1

- No. The number 0.1 can't be represented precisely in binary. Using floating point representation does not change anything. It does not matter that the number is negative.
- But let us proceed to see how -0.1 is (imprecisely) represented.
  - The sign bit of -0.1 is 1 (for negative).
  - The binary fraction for 0.1 is 0.000110011001100110011001100...
  - The mantissa of 0.1 is:
    • Shift the leading bit into the one's place: 1.100110011001100110011001100...
    • The shift was 4 places left, for an exponent of -4
    • Drop the one bit in the one's place: .100110011001100110011001100...
    • Retain 23 bits: 100 1100 1100 1100 1100 1100
  - The actual exponent is -4
  - The biased exponent is 127 - 4 = 123 = 0111 1011
- The final result is: 1011 1101 1100 1100 1100 1100 1100 which is stored as 0xdeadbeef because QTSpin rounds up the last bit.

MIPS Floating Point

- Floating point on MIPS was originally done in a separate chip called coprocessor 1 (also called the FPA for Floating Point Accelerator). Modern MIPS chips include floating point operations on the main processor chip. But the instructions sometimes act as if there were still a separate chip.
- MIPS has 32 single precision (32 bit) floating point registers.
- The registers are named $f0 – f31
- $f0 is not special (it can hold any bit pattern, not just zero).
- Single precision floating point load, store, arithmetic, and other instructions work with these registers.
Double Precision

- MIPS also has hardware for double precision (64 bit) floating point operations. For this, it uses pairs of single precision registers to hold operands. There are 16 pairs, named \$f0, \$f2, \ldots, \$f30. Only the even numbered register is specified in a double precision instruction; the odd numbered register of the pair is included automatically.
- Some MIPS processors allow only even-numbered registers (\$f0, \$f2, \ldots) for single precision instructions. However QTSpim allows all 32 registers in single precision instructions. These notes follow that usage.

Single Precision Load

- Actual hardware has a delay between a load instruction and the time when the data reaches the register. In QTSpim, there is an option that disables the load delay. For the time being, disable the load delay. (Floating point is tricky enough already).
- Loading a single precision value is done with a pseudoinstruction:

  \[
  \text{l.s fd,addr} \quad \# \text{load register fd from addr} \\
  \quad \# \text{(pseudoinstruction)}
  \]

  This instruction loads 32 bits of data from address addr into floating point register \$fd (where d is 0, 1, 2, \ldots, 31). Whatever 32 bits are located at addr are copied into \$fd. If the data makes no sense as a floating point value, that is OK for this instruction. Later on the mistake will be caught when floating point operations are attempted.

  - Every one of the \(2^{32}\) patterns that 32 bits can form is a legal two's complement integer. Are some of those patterns not legal as a floating point value? Hint: think about how zero is represented.
Single Precision Store

It is not legal to have an exponent field of zero with a non-zero mantissa. There are other combinations that are also illegal.

- Sometimes the floating point registers are used as temporary registers for integer data. For example, rather than storing a temporary value to memory, you can copy it to an unused floating point register. This is OK, as long as you don't try to do math with them.
- There is a single precision store pseudoinstruction:

  s.s fd,addr # store register fd to addr
  # (pseudoinstruction)

- Whatever 32 bits are in fd are copied to addr. In both of these pseudoinstructions, the address, addr, can be an ordinary symbolic address, or an indexed address.

Floating Point Load Immediate

- There is a floating point load immediate pseudoinstruction. This loads a floating point register with a constant value that is specified in the instruction.

  li.s fd,val # load register $fd with val
  # (pseudoinstruction)

- Here is a code snippet showing this:

  li.s $f1,1.0 # $f1 = constant 1.0
  li.s $f2,2.0 # $f2 = constant 2.0
  li.s $f10,1.0e-5 # $f10 = 0.00001

- (Take a guess:) Could the first instruction be written: li.s $f1,1 ?
Example Program

No. The decimal point is needed in the constant, at least for QTSpim.

• Here is a program that exchanges (swaps) the floating point values at valA and valB. Notice how the two floating point values are written. The first in the ordinary style; the second in scientific notation.

```assembly
## swap.asm
## Exchange the values in valA and valB.
.text
.globl main
main:
    l.s $f0, valA     # $f0 <-- valA
    l.s $f1, valB    # $f1 <-- valB
    s.s $f0, valB    # $f0 --> valB
    s.s $f1, valA    # $f1 --> valA
    li $v0, 10       # code 10 --- exit
    syscall          # Return to OS
.data
valA: .float 8.32  # 32 bit floating point value
valB: .float -0.6234e4 # 32 bit floating point value
                       # small 'e' only
## End of file
```

Floating Point Services

• To print a floating point value to the QTSpim monitor, use service 2 (for single precision) or service 3 (for double precision). Register $f12 contains the single-precision floating point number to be printed. Registers $f12 and $f13 contain the double-precision number to be printed. To read a floating point value from the user, use service 6 (for single precision) or service 7 (for double precision).

• The following slide shows a mistake. It first does this correctly (using system call 2). Then does it incorrectly using the integer print service (system call 1). Of course, the 32 bits of the floating point value can be interpreted as an integer, so system call 2 innocently does what we asked it to do.
Mistake

```asm
## print.asm
##
## Print out a 32 bit pattern, first as a float,
## then as an integer.
.text
.globl main
main:
    l.s $f12,val       # load the float argument
    li $v0,2           # code 2 == print float
    syscall           # (correct)
    li $v0,4
    syscall           # print
    la $a0,lfeed       # line separator
    syscall           
    lw $a0,val        # load the same bit pattern
    li $v0,1           # code 1 == print int
    syscall           # (mistake)
    li $v0,10          # code 10 == exit
    syscall           # Return to OS
.data
val:       .float -8.32 # negative value
lfeed:     .asciiz "n"
## End of file
```

No Type Checking

- This type of mistake often happens when programming in "C" where type checking is weak. Sometimes the wrong type can be passed to a function (such as printf) and odd things happen. Here is the output of the program:

![Output of the program](image)

- Compilers that keep track of the data types of values, and make sure that the correct types are used as arguments do strong type checking. Java is strongly typed. In assembly language, type checking is largely absent.

- The first line of output shows the 32 bits interpreted as a single precision floating point value. Is this output correct?
Precision of Single Precision Floats

- There are two things wrong: (1) the value -8.32 cannot be represented exactly in binary, and (2) QTSpim prints out way too many digits.
- Single precision floats have (recall) only 24 bits of precision. This is the equivalent of 7 to 8 decimal digits. QTSpim prints out many more digits than are actually represented. This is incorrect. QTSpim should have printed -8.319999 to the window.
- The 7 or 8 decimal digits of precision is much worse than most electronic calculators. It is usually unwise to use single precision floating point in programs. (But this lecture uses it since the goal is to explain concepts, not to write production grade programs). Double precision has 15 or 16 decimal places of precision.

Double Precision Floats

- The exponent is expanded from 8 to 11 bits;
- The mantissa is expanded from 23 to 52 bits, giving 15-16 decimal digits of precision;
- The exponent bias is increased from 127 to 1023.
Other Floating Point Issues

- **Zero**
  Zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

- **Denormalized numbers**
  If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number \((-1)^s \times 0.f \times 2^{-126}\), where \(s\) is the sign bit and \(f\) is the fraction. For double precision, denormalized numbers are of the form \((-1)^s \times 0.f \times 2^{-1022}\). From this you can interpret zero as a special type of denormalized number.

Other Floating Point Issues

- **Infinity**
  The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

- **NaN (not a number)**
  The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN).
  A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined. An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.
  Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.
Floating Point References


Summary

- Floating-point history was presented;
- Examples of floating-point representation in QTSpim were presented;
- MIPS Floating-point instructions were explained and illustrated with example programs;
- The IEEE standard 754 was explained.