EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 13: Floating-Point Arithmetic

Announcements

- Midterm

Notes on Integer Arithmetic in MIPS

- Computers are made to deal with numbers
- What can we represent in *N* bits?
  - Unsigned integers: 0 to 2^{N-1}
  - Signed Integers (Two's Complement): $-2^{(N-1)}$ to $2^{(N-1)} - 1$.
- In case of overflow (result doesn't fit in 32 bits), what should be done?
- MIPS designers provide instructions that cause overflow to be detected (add), and instructions that do not cause overflow to be detected (addu)
- It is up to the programmer to deal with overflow:
  - C ignores overflow, hence MIPS C compilers always generate the unsigned version of the arithmetic instructions addu, addiu, subu no matter what the type of the variable is.
  - In Fortran, overflow is not ignored, and MIPS Fortran compilers pick the appropriate instruction depending on the type of the operands.
- MIPS detects overflow with an exception (also called interrupt)
- Exceptions: An exception is simply an unscheduled procedure (function) call
  - The address of the instruction that overflowed is saved in a register and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
  - The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

Notes on Integer Arithmetic in MIPS (cont'd)

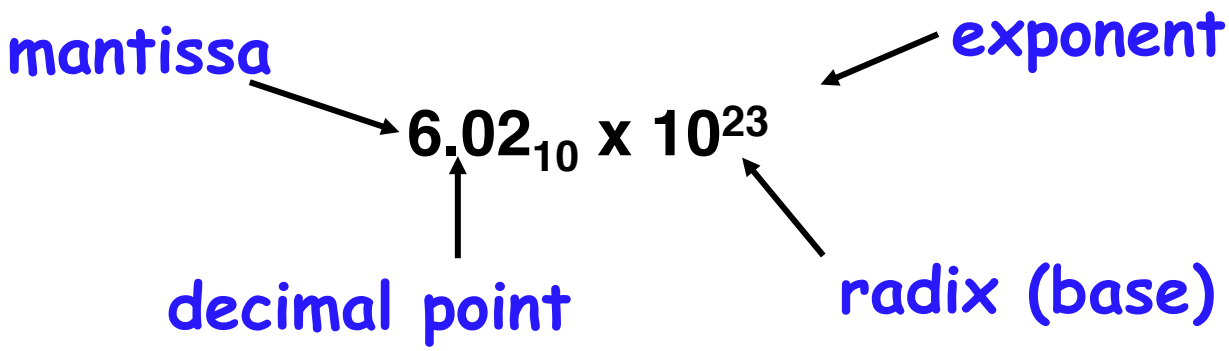
- MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception.
- The instruction *move from system control* (mfc0) is used to copy EPC (and other special registers) into a general-purpose register so that MIPS software has the option of returning to the offending instruction. mfc0 $s1, $epc
- Although MIPS can trap overflow, there is no conditional branch to test overflow. Is it possible to write a sequence of instructions that discovers overflow for signed/ unsigned numbers and branch accordingly to some procedure to handle the overflow?
- Multiplication and Division in MIPS: (R-format)
  - Mult(Multu) $s0, $s1
  - Div(Divu) $s0, $s1
- Same hardware unit used for both (review EECE 320)
  - Result is produced in a 64-bit register part of hardware unit

Notes on Integer Arithmetic in MIPS (cont'd)

- Multiply: HI:LO represent product
  - MIPS provides 2 instructions mfhi (mflo) move from HI (move from LO) to move HI (LO) into a general purpose register. EX: mfhi $s0
- Multiply pseudo-instructions: mul (mulo,mulou) $rd, $rs1, $rs2
- Divide: LO = $s2/$s3 (quotient), HI = Ss2 mod $s3 (remainder); use mfhi (mflo)
- Divide pseudo-instructions: Div (Divu) $rd,$rs1,$rs2

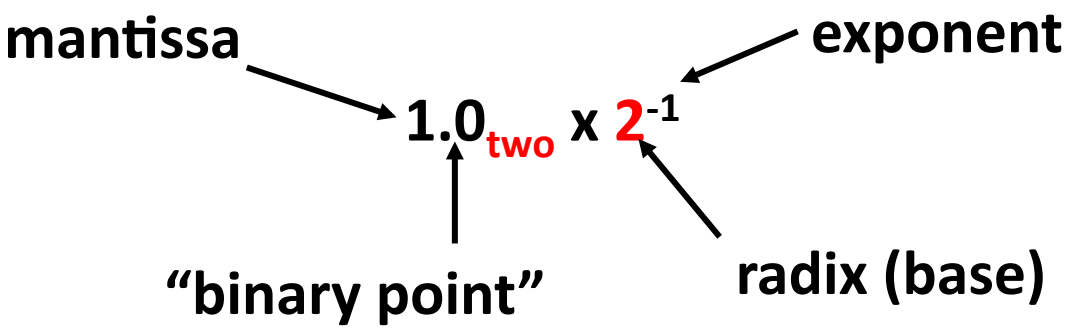
Real Numbers

- Decimal or real numbers:
  - Very large numbers? (seconds/century)
    $3{,}155{,}760{,}000_{10}$ ($3.15576_{10} \times 10^{9}$)
  - Very small numbers? (atomic diameter)
    0.00000001_{10} ($1.0_{10} \times 10^{-8}$)
  - Rationals (repeating pattern)
    2/3 (0.666666666. . .)
  - Irrationals
    $2^{1/2}$ (1.414213562373. . .)
  - Transcendentals
    e (2.718...), π (3.141...)
- Represent real numbers in scientific notation:



Scientific Notation

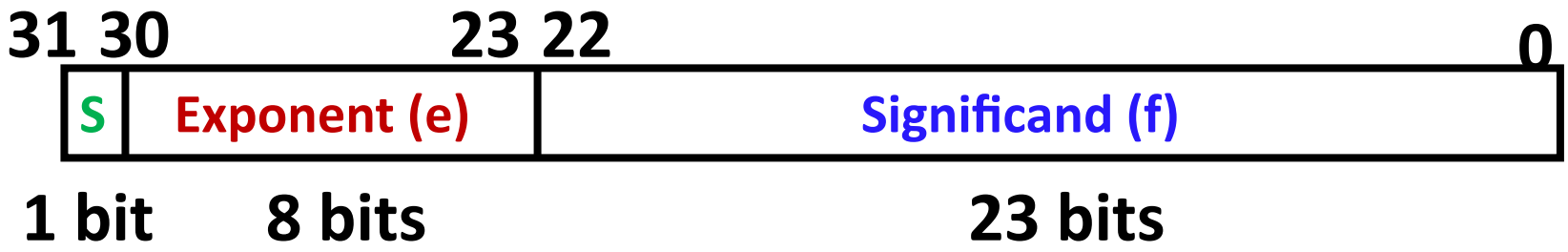
- Normalized scientific notation: no leadings 0s (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}.
- Binary scientific notation:



- Computer arithmetic that supports it called *floating point*, because it represents numbers where binary point is not fixed, as it is for integers
  - Declare such variable in C as float

Floating Point Representation

- Normal format: $+1.xxxxxxxxxx_{two} * 2^{yyyy_{two}}$
- Multiple of Word Size (32 bits)



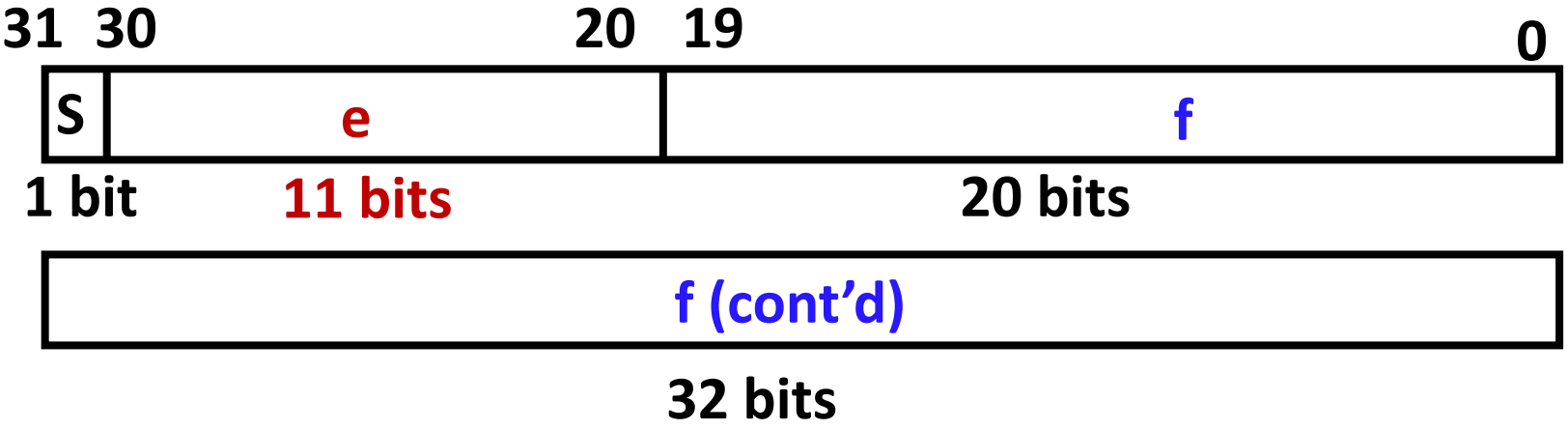
- S represents Sign
- Exponent (e) represents y's (in 2's complement)
- Significand (f) represents x's
- Represent numbers as small as 2^{-128} to as large as $1.1111\ldots_2 \times 2^{127}$.
- Representation of number 0:
  - Has exponent all 0's so that hardware doesn't attach 1 in all 0's significand.
  - S is disregarded
  - More about this in IEEE FP standard

Floating Point Representation (cont'd)

- What if result too large? ($> 1.1111\ldots_2 \times 2^{127}$)
  - Overflow!
  - Overflow ➔ Exponent larger than represented in 8-bit Exponent field
- What if result too small? ($>0, < 2^{-128}$)
  - Underflow!
  - Underflow ➔ Negative exponent larger than represented in 8-bit Exponent field
- How to reduce chances of overflow or underflow?

Double Precision Floating Point Representation

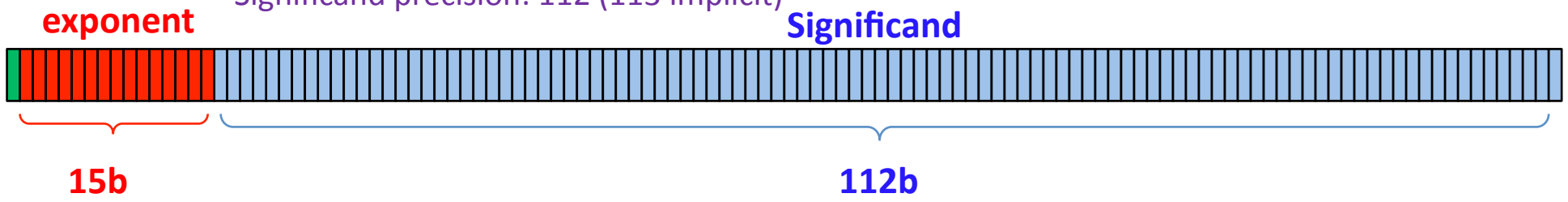
- Next Multiple of Word Size (64 bits)



- Double Precision (vs. Single Precision)
  - C variable declared as double
  - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
  - But primary advantage is greater accuracy due to larger significand
- Quad Precision Floating Point Representation (IEEE 754-2008 standard)
  - Next Multiple of Word Size (128 bits)
  - Unbelievable range of numbers
  - Unbelievable precision (accuracy)

Quad Precision Floating Point Representation

- Officially referred to as binary128.
- Format:
  - Sign bit: 1
  - Exponent width: 15
  - Significand precision: 112 (113 implicit)



IEEE 754 Floating Point Standard

- Kahan: "Father" of the Floating point standard

- Single Precision (Double Precision similar)
- Sign bit: 1 means negative, 0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: Significand < 1 (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0
- Kahan wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using *integer* compares.
- Could break FP number into 3 parts: compare signs, then compare exponents, then compare significands
- Wanted it to be faster, single compare if possible, especially if positive numbers
- Then want order:
  - Highest order bit is sign (negative < positive)
  - Exponent next, so big exponent => bigger #
  - Significand last: exponents same => bigger #

IEEE 754 Floating Point Standard (cont'd)

- Negative Exponent?
  - 2's comp? 1.0×2^{-1} v. $1.0 \times 2^{+1}$ (1/2 v. 2)

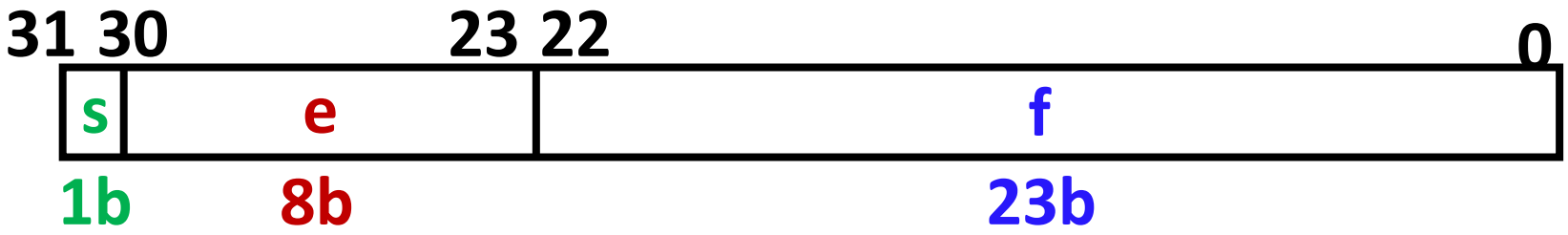
1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

- This notation using unsigned integer compare of 1/2 v. 2 makes 1/2 > 2!
- Instead, pick notation 0000 0001 is most negative, and 1111 1111 is most positive
  - 1.0×2^{-1} v. $1.0 \times 2^{+1}$ (1/2 v. 2)

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

- Called Biased Notation, where bias is a number subtracted to get real number
  - IEEE 754 uses bias of 127 for single precision
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision
  - Bias converts all single-precision exponents from -128 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.

Summary of IEEE 754 Single Precision FP Standard



- $(-1)^{S} \times (1 + f) \times 2^{(e-127)}$
- Double precision identical, except with exponent bias of 1023
- Exponent is treated as an unsigned number
- Bias will produce actual number
- Example
  - If the actual exponent is 4, the e field will be 4 + 127 = 131 (10000011_2).
  - If e contains 01011101 (93), the actual exponent is 93 - 127 = -34.
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.

Computing the Significand

- Method 1 (Fractions):
  - In decimal: $0.340_{(10)}$ => $340_{(10)}/1000_{(10)}$ => $34_{(10)}/100$
  - In binary: $0.110_{(2)}$ => $110_{(2)}/1000_{(2)} = 6_{(10)}/8_{(10)}$
    => $11_{(2)}/100_{(2)} = 3_{(10)}/4_{(10)}$
  - Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

- Method 2 (Place Values):
  - Convert from scientific notation
  - In decimal: $1.6732 = (1x10^{0}) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$
  - In binary: $1.1001 = (1x2^{0}) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$
  - Interpretation of value in each position extends beyond the decimal/binary point
  - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

Example: Converting Binary IEEE 754 FP Number to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 – 127 = - 23
- Significand:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
    $= 1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$
    $= 1.0_{ten} + 0.666115_{ten}$
- Represents: $1.666115_{ten} * 2^{-23} \sim 1.986 * 10^{-7}$ (about 2/10,000,000)
- Another example: 1 01111100 11000000000000000000000
- Decimal equivalent: – 0.21875

Converting Decimal to IEEE 754 FP

- Simple Case: If denominator is an exponent of 2 (2, 4, 8, 16, etc.), then it's easy.
- Show IEEE 754 FP representation of -0.75
  - $-0.75 = -3/4 = -11_{two}/100_{two} = -0.11_{two}$
  - Normalized to $-1.1_{two} \times 2^{-1}$.
  - $(-1)^S \times (1 + f) \times 2^{(e-127)}$
  - $(-1)1 \times (1 + .100\ 0000 \ldots 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000

- Not So Simple Case: If denominator is not an exponent of 2.
  - Then we can't represent number precisely, but that's why we have so many bits in significand: for precision
  - Once we have significand, normalizing a number to get the exponent is easy.
  - So how do we get the significand of a never-ending number?
- Fact: All rational numbers have a repeating pattern when written out in decimal.
  - Fact: This still applies in binary.
- To finish conversion:
  - Write out binary number with repeating pattern.
  - Cut it off after correct number of bits (different for single v. double precision).
  - Derive Sign, Exponent and Significand fields.