Floating Point

Outline

- Review from last time
- Integer multiplication & division
- FP add/sub
- FP on MIPS
- Special “numbers”
- Rounding

IEEE 754 Floating Point Review

- **Summary (single precision):**

31	30 ... 23	22 ... 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- $(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

  - Double precision identical, except with exponent bias of 1023

- **Interpretation of value in each position extends beyond the decimal/binary point**

$$1.1001 = (1 \times 2^{0}) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

Special Numbers Reviewed

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	???

Multiplication (1/3)

- **Paper and pencil example (unsigned):**

```
Multiplicand   1000      8
 Multiplier   x1001          9
              -----
                  1000
                 0000
                0000
              +1000
              --------
              01001000
```


- m bits x n bits = m + n bit product

Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use `mfhi` register & `mflo` register to move from hi, lo to another register

Multiplication (3/3)

- **Example:**
  - **in C:** `a = b * c;`
  - **in MIPS:**
    - let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)


```
mult $s2,$s3    # b*c
mfhi $s0        # upper half of
                # product into $s0
mflo $s1        # lower half of
                # product into $s1
```

- **Note: Often, we only care about the lower half of the product.**

Division (1/3)

- **Paper and pencil example (unsigned):**

```
                      1001     Quotient
Divisor 1000|1001010  Dividend
           -1000
              10
              101
              1010
             -1000
                10     Remainder
            (or Modulo result)
```


- **Dividend = Quotient x Divisor + Remainder**

Division (2/3)

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit values in register 1 by 32-bit value in register 2:
    - puts remainder of division in hi
    - puts quotient of division in lo
- Notice that this can be used to implement both the C division operator (/) and the C modulo operator (%)

Division (3/3)

- **Example:**
  - **in C:**
```
a = c / d;
b = c % d;
```
  - **in MIPS:**

    - let a be $s0; let b be $s1; let c be $s2; and let d be $s3

```
div  $s2,$s3  # lo=c/d, hi=c%d
mflo $s0      # get quotient
mfhi $s1      # get remainder
```

Unsigned Instructions & Overflow

- MIPS also has versions of these two arithmetic instructions for **unsigned operands**:

```
multu
divu
```


  - Determines whether or not the product and quotient are changed if the operands are signed or unsigned.

- MIPS <u>does not</u> check overflow on ANY signed/unsigned multiply, divide instr
  - Up to the software to check hi

FP Addition & Subtraction 1/2

- <u>Much</u> more difficult than with integers
- Can't just add significands
- How do we do it?
  1. De-normalize to match larger exponent
  2. Add significands to get resulting one
  3. Normalize (& check for under/overflow)
  4. Round if needed (may need to goto 3)
- Note: If signs differ, just perform a subtract instead.
- Subtract is similar

FP Addition & Subtraction 2/2

- Problems in implementing FP add/sub:
  - If signs differ for add (or same for sub), what will be the sign of the result?

- Question: How do we integrate this into the integer arithmetic unit?

  - Answer: We don't!

MIPS Floating Point Architecture (1/4)

- Separate floating point instructions:
  - Single Precision:

    `add.s, sub.s, mul.s, div.s`
  - Double Precision:

    `add.d, sub.d, mul.d, div.d`

- These instructions are <u>far more complicated</u> than their integer counterparts, so they can take much longer to execute.

MIPS Floating Point Architecture (2/4)

- Problems:
  - It's inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.
  - Some programs do no floating point calculations
  - It takes lots of hardware relative to integers to do Floating Point fast

MIPS Floating Point Architecture (3/4)

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most of the registers specified in `.s` and `.d` instruction refer to this set
  - separate load and store: `lwc1` and `swc1` (“load word coprocessor 1”, “store …”)
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`
    - Even register is the name

MIPS Floating Point Architecture (4/4)

- 1990 Computer actually contains multiple separate chips:
  - Processor: handles all the normal stuff
  - Coprocessor 1: handles FP and only FP;
  - more coprocessors?… Yes, later
  - Today, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
- Appendix pages A-70 to A-74 contain many, many more FP operations.

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	???

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
  - If infinity is not an error, these shouldn't be either.
  - Called **N**ot **a** **N**umber (**NaN**)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN,X) = NaN

Special Numbers (cont'd)

- What have we defined so far? (Single Precision)?

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Representation for Denorms (1/2)

- **Problem: There's a gap among representable FP numbers around 0**
  - **Smallest representable pos num:**

    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$

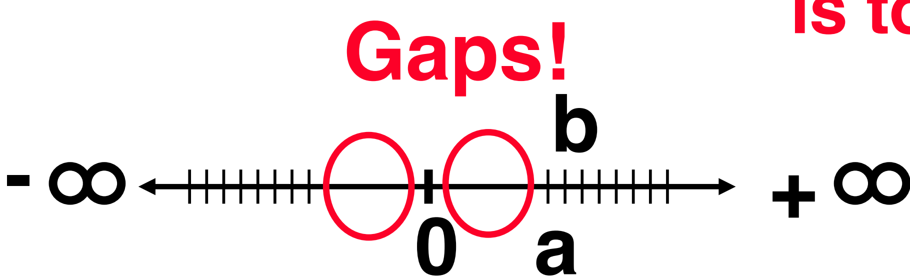
  - **Second smallest representable pos num:**

    $b = 1.000\ldots\ldots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

    $a - 0 = 2^{-126}$

    $b - a = 2^{-149}$

Normalization and implicit 1 is to blame!

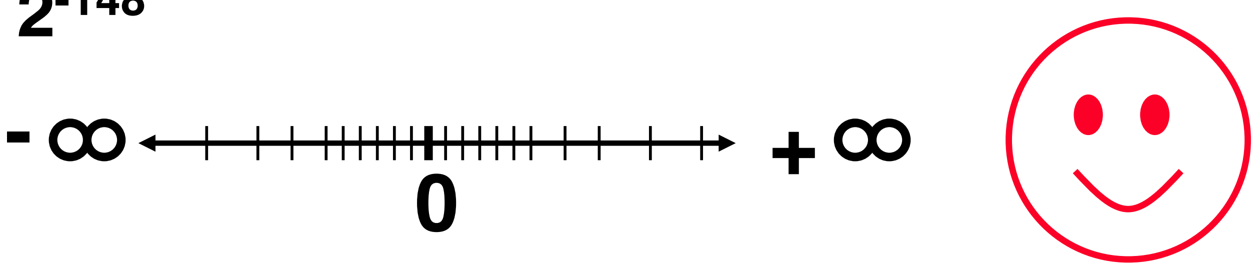


Representation for Denorms (2/2)

- **Solution:**
  - We still haven't used Exponent = 0, Significand nonzero
  - Denormalized number: no leading 1, implicit exponent = -126.
  - Smallest representable pos num:

    $a = 2^{-149}$
  - Second smallest representable pos num:

    $b = 2^{-148}$



Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.

- The FP hardware carries two extra bits of precision, and then round to get the proper value

- Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer

IEEE Four Rounding Modes

1. Round towards +infinity
   - ALWAYS round "up": 2.001 -> 3
   - -2.001 -> -2

2. Round towards -infinity
   - ALWAYS round "down": 1.999 -> 1,
   - -1.999 -> -2

3. Truncate
   - Just drop the last bits (round towards 0)

4. Round to (nearest) even
   - Normal rounding, almost

Round to Even

- Round like you learned in grade school
- Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - 2.5 -> 2
  - 3.5 -> 4

- Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down
  - Ask statistics majors
- This is the default rounding mode

Things to Remember

- Integer mul & div: `mult,div,mfhi,mflo`
- New MIPS registers (`$f0-$f31`), instruct.:
  - Single Precision (32 bits, 2×10^{-38}… 2×10^{38}):
    `add.s, sub.s, mul.s, div.s`
  - Double Precision (64 bits , 2×10^{-308}…2×10^{308}):
    `add.d, sub.d, mul.d, div.d`
- FP add & subtract are **not associative**.
- IEEE 754
  - NaN & Denorms (precision)
  - Four different rounding modes

BSAS#27: Floating Point Fallacy

- FP add, subtract associative: FALSE!
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
  - $x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$
    $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = \underline{0.0}$
  - $(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$
    $= (0.0) + 1.0 = \underline{1.0}$

- **Therefore, Floating Point add, subtract are not associative!**
  - Why? FP result approximates real result!
  - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

BSAS#26: Casting floats <-> ints

- `(int)` *`floating point exp`*
  - Coerces and converts it to the nearest integer (C uses truncation)
  - `i = (int) (3.14159 * f);`

- `(float)` *`exp`*
  - converts integer to nearest floating point
  - `f = f + (float) i;`

BSAS#26: int -> float -> int

```
if (i == (int)((float) i)) {
  printf("true");
}
```


- Will **not** always print "true"

- Large values of integers don't have exact floating point representations
- What about `double`?

BSAS#26: float -> int -> float

```
if (f == (float)((int) f)) {
  printf("true");
}
```


- Will **not** always print "true"

- Small floating point numbers (<1) don't have integer representations
- For other numbers, rounding errors