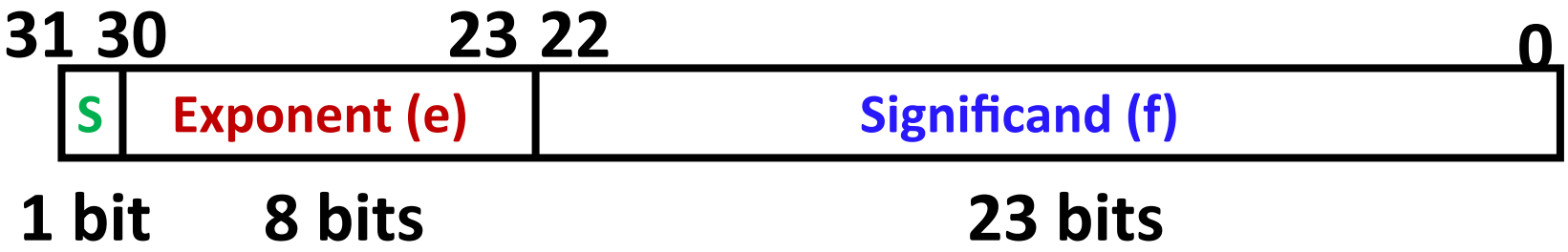
EECE 321: Computer Organization

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

Lecture 14: Floating-Point Arithmetic

Floating Point Representation

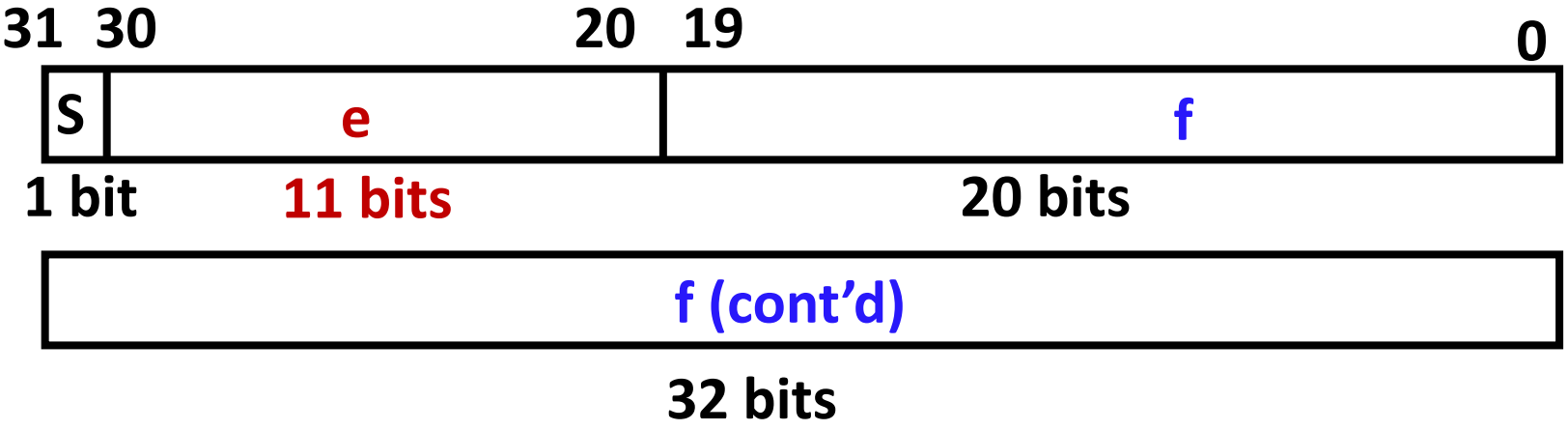
- Normal format: $+1.xxxxxxxxxx_{two} * 2^{yyyy_{two}}$
- Multiple of Word Size (32 bits)



- 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

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)

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)

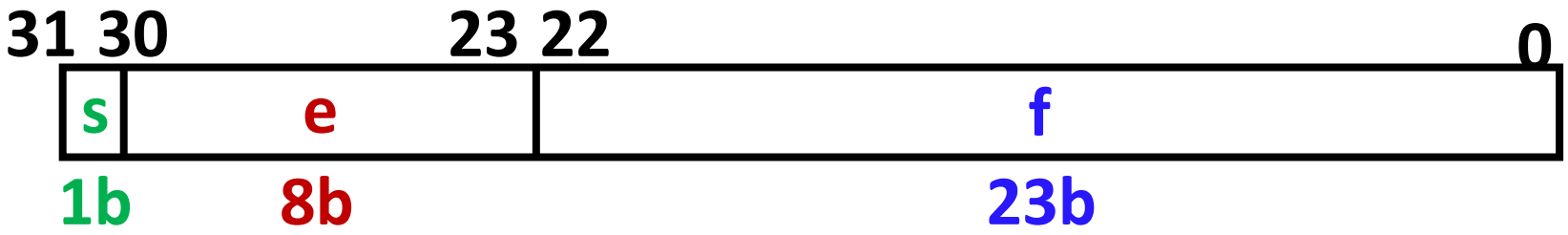
1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

- 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: $0.340_{(10)}$ => $340_{(10)}/1000_{(10)}$ => $34_{(10)}/100_{(10)}$
  - 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.

More Examples

- What is the single-precision representation of 347.625?
  - $347.625 = 101011011.101_{(2)}$.
  - Normalize the number: $101011011.101 = 1.01011011101 \times 2^8$.
  - The e field should contain: 8 + 127 = 1000 0111.
  - Result: 0 10000111 01011011101000000000000
- What is the decimal equivalent of the following floating point number?

1	1000 0001	111 0000 0000 0000 0000 0000

Special Values

- The smallest and largest possible exponents e=00000000 and e=11111111 (and their double precision counterparts) are reserved for special values.
- 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.
- 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.
- Finally, there is a special "not a number" or NAN 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.

Special Value	s	e	f
0	X	All-zeros	All-zeros
+∞	0	All-ones	All-zeros
-∞	1	All-ones	All-zeros
NAN	X	All-ones	Non-zero

Range of IEEE 754 FP Single-Precision Numbers

- What is the smallest positive single-precision value that can be represented?
  - The smallest positive non-zero number is $1 * 2^{-126} = 2^{-126}$.
  - The smallest e is 00000001 (1).
  - The smallest f is 00000000000000000000000 (0).
- The largest possible "normal" number is $(2 - 2^{-23}) * 2^{127} = 2^{128} - 2^{104}$.
  - The largest possible e is 11111110 (254).
  - The largest possible f is 11111111111111111111111 ($=1 - 2^{-23}$).
- In comparison, the smallest and largest possible 32-bit integers in two's complement are only -2^{31} and $2^{31} - 1$.
- How can we represent so many more values in the IEEE 754 format, even though we use the same number of bits as regular integers?

Finiteness

- There aren't more IEEE numbers.
- With 32 bits, there are $2^{32} - 1$, or about 4 billion, different bit patterns.
  - These can represent 4 billion integers or 4 billion reals.
  - But there are an infinite number of reals, and the IEEE format can only represent some of the ones from about -2^{128} to $+2^{128}$.
  - Represent same number of values between 2^n and 2^{n+1} as 2^{n+1} and 2^{n+2}.

- Thus, floating-point arithmetic has "issues"
  - Small round-off errors can accumulate with multiplications or exponentiations, resulting in big errors.
  - **Not Associative**: Rounding errors can invalidate many basic arithmetic principles such as the associative law, (x + y) + z = x + (y + z).

- The IEEE 754 standard guarantees that all machines will produce the same results — but those results may not be mathematically correct!

Limits of the IEEE representation

- Even some integers cannot be represented in the IEEE format.

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

Answer: 33554431

33554432.000000

- Some simple decimal numbers cannot be represented exactly in binary to begin with.
  - 0.10 = 0.0001100110011...

The '0.1' Dilemma

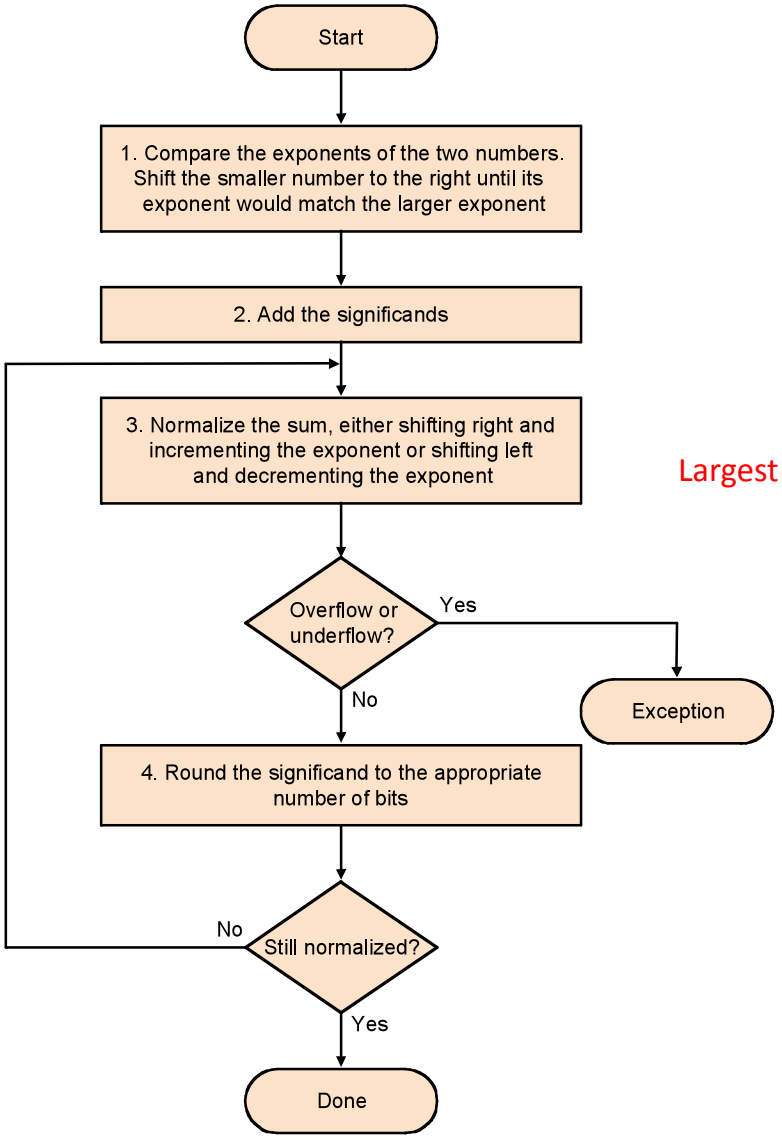
- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- 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!
- Professor Skeel (from UIUC) wrote a short article about this.
  - Round-off Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.

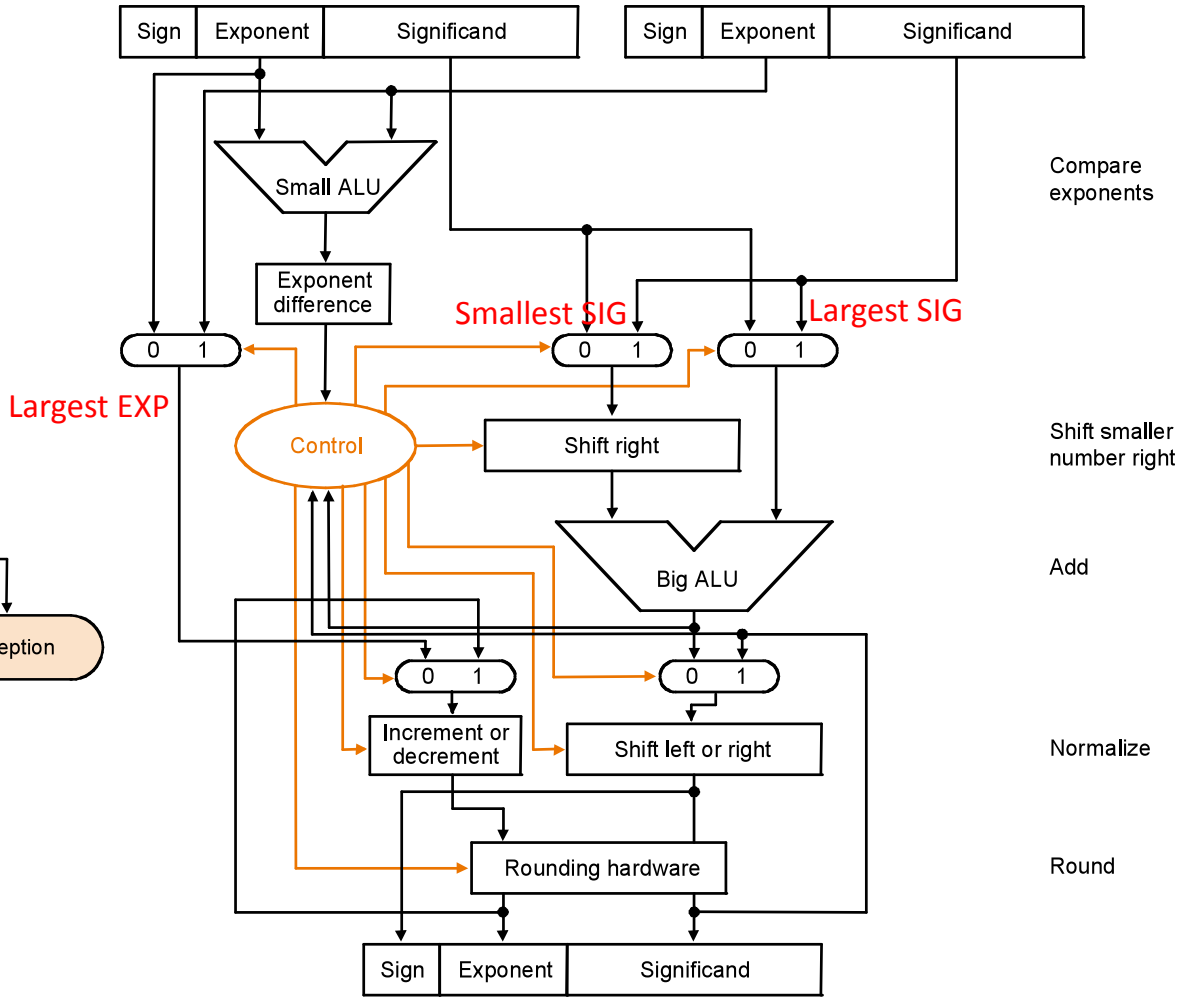
Multiplication and Floating Point

Floating-Point Addition and Multiplication

- Much more difficult than with integers; can't just add/multiply significands!
- How do we do addition?
  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 renormalize)
- EX: Assume we keep four bits of precision: $0.5 - 0.4375 = 1.000\text{x}2^{-1} - 1.110\text{x}2^{-2}$.
  - De-normalize and add significands: $1.000\text{x}2^{-1} - 0.111\text{x}2^{-1} = 0.001 \text{ x } 2^{-1}$.
  - Normalize the sum checking for overflow: $1.000 \text{ x } 2^{-4}$.
  - No need for rounding in this case.
- If signs ≠, do a subtract. (Subtract similar)
- Multiplication: To multiply two floating-point values, first multiply their magnitudes and add their exponents.
  - Then round and normalize the result
  - The sign of the product is the exclusive-or of the signs of the factors.
- Question: How do we integrate these into the integer arithmetic unit?
- Answer: We don't!

FP Addition Algorithm and Datapath





Rounding

- Math on real numbers ⇒ we worry about rounding to fit result in the significant field.
- Rounding occurs when converting…
  - double to single precision
  - floating point # to an integer
- FP hardware carries 3 extra bits of precision, and rounds for proper value
  - Guard, Round, Sticky bits.
  - The goal is to obtain final results as if the intermediate results were calculated using infinite precision and then rounded.

```
mantissa format plus extra bits:

      1.XXXXXXXXXXXXXXXXXXXXXXX   0   0   0

      ^          ^                ^   ^   ^
      |          |                |   |   |
      |          |                |   |   -  sticky bit (s)
      |          |                |   -  round bit (r)
      |          |                -  guard bit (g)
      |          -  23 bit mantissa from a representation
      -  hidden bit
```

Rounding (cont'd)

- When a mantissa is to be shifted in order to align radix points, the bits that fall off the least significant end of the mantissa go into these extra bits.
- The guard & round bits are just 2 extra bits of precision used in calculations.
- The sticky bit is an indication of what is/could be in lesser significant bits that are not kept. If a value of 1 ever is shifted into the sticky bit position, that sticky bit remains a 1 ("sticks" at 1), despite further shifts.

```
Example:
   mantissa from representation,  11000000000000000000100
   must be shifted by 8 places (to align radix points)

                                                    g r s
   Before first shift:  1.11000000000000000000100 0 0 0
   After 1 shift:       0.11100000000000000000010 0 0 0
   After 2 shifts:      0.01110000000000000000001 0 0 0
   After 3 shifts:      0.00111000000000000000000 1 0 0
   After 4 shifts:      0.00011100000000000000000 0 1 0
   After 5 shifts:      0.00001110000000000000000 0 0 1
   After 6 shifts:      0.00000111000000000000000 0 0 1
   After 7 shifts:      0.00000011100000000000000 0 0 1
   After 8 shifts:      0.00000001110000000000000 0 0 1
```

- IEEE four modes of rounding:
  - Round towards + ∞: ALWAYS round "up": 2.1 ⇒ 3, -2.1 ⇒ -2
  - Round towards - ∞: ALWAYS round "down": 1.9 ⇒ 1, -1.9 ⇒ -2
  - Truncate: Just drop the last bits (round towards 0)
  - Round to (nearest) even (default): Normal rounding, almost: 2.5 ⇒ 2, 3.5 ⇒ 4

MIPS Floating Point Architecture

- MIPS supports the IEEE 754 SP & DP formats.
- Separate floating point instructions:
  - Single Precision:

    add.s, sub.s, mul.s, div.s, c.eq.s (also neq, lt, le, gt, ge)
  - Double Precision:

    add.d, sub.d, mul.d, div.d, c.eq.d (also neq, lt, le, gt, ge)
- FP branch instructions: bc1t, bc1f
- FP comparisons set a bit to true/false, and a FP branch decides to branch based on that bit.
- These are far more complicated than their integer counterparts
  - Can take much longer to execute
- Problems:
  - Inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a particular piece of data will not change FP to int within a program.
    - Only 1 type of instruction will be used on it.
  - Some programs do no FP calculations
  - It takes lots of hardware relative to integers to do FP fast

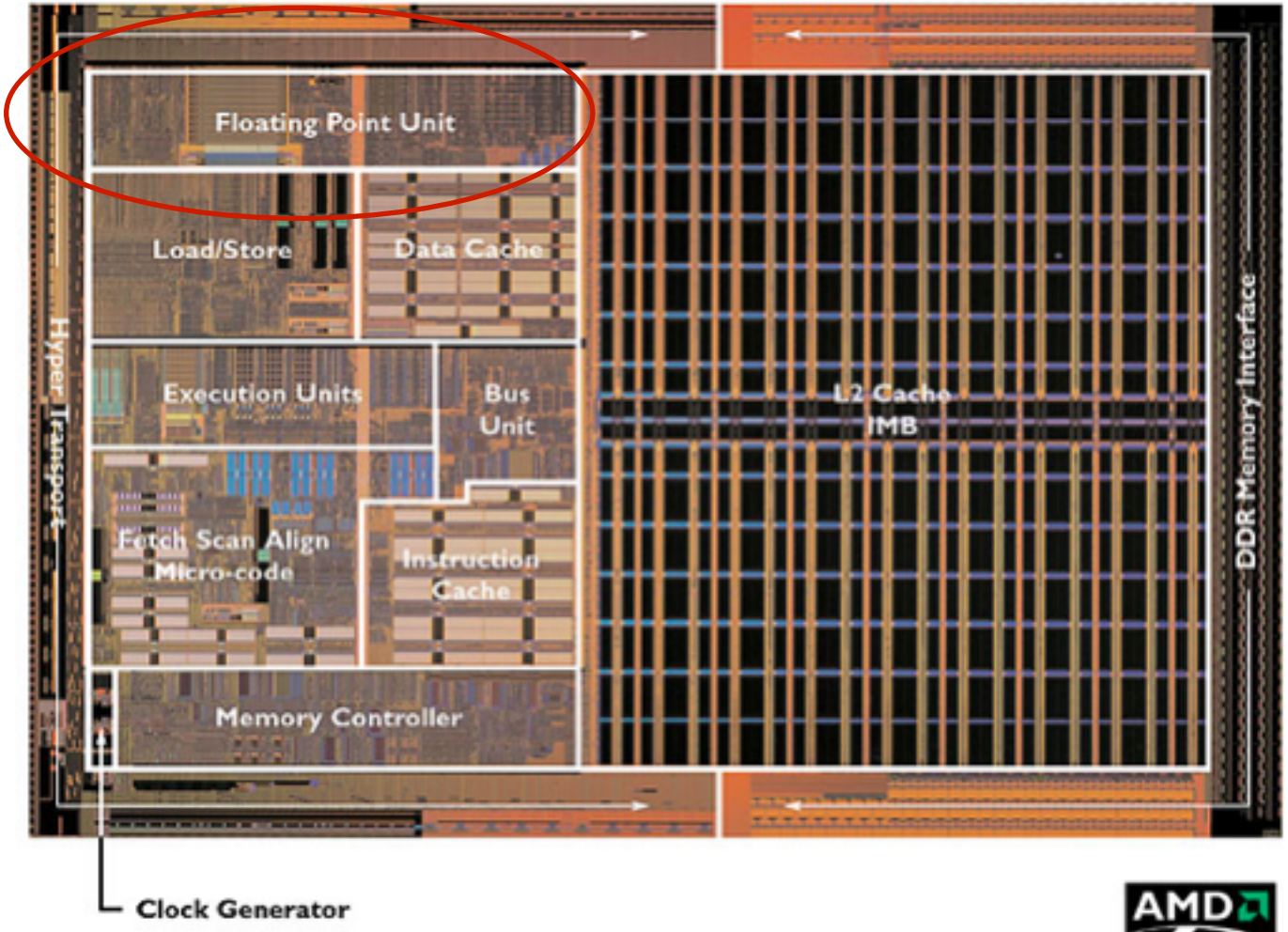
MIPS Floating Point Architecture

- 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 instructions refer to this set
  - separate load and store: lwc1 and swc1
    ("load word coprocessor 1", "store …")
  - The base registers for FP data transfers remain integers
- Double Precision: by convention, even/odd pair contain one DP FP number: $f0/$f1, $f2/$f3, … , $f30/$f31
  - Even register is the name
- Ex: load 2 SP numbers from memory, add them and store result back:
  - lwc1 $f4, 4($sp)
  - lwc1 $f6, 8($sp)
  - add.s $f2, $f4, $f6
  - swc1 $f2, 12($sp)
- Ex: load 2 DP numbers from memory, add them and store result back:
  - lwc1 $f4, 4($sp) # loads f4, f5
  - lwc1 $f6, 8($sp) # loads f6, f7
  - add.d $f2, $f4, $f6 # sum in f2, f3
  - swc1 $f2, 12($sp) # stores f2, f3

FP Hardware

- When floating point was introduced in microprocessors, there wasn't enough transistors on chip to implement it.
  - You had to buy a floating point co-processor (e.g., the Intel 8087)
- As a result, many ISA's use separate registers for floating point.
- Modern transistor budgets enable floating point to be on chip.
  - Intel's 486 was the first x86 with built-in floating point (1989)
- Even the newest ISA's have separate register files for floating point.
  - Makes sense from a chip floor-planning perspective.

FPU Like Co-Processor on Chip



Example: FP Matrix Multiplication

```
void mm(double x[][],double y[][], double z[][]){
    int i,j,k;
    for(i=0;i!=32;i++){
          for(j=0;j!=32;j++){
                for(k=0;k!=32;k++){
                      x[i][j] += y[i][k] * z[k][j];
                }
          }
    }
}
```

- x, y, z are 32x32 2-Dimensional arrays
- They are stored like 32 1-D arrays, except each element is a 32 element array.
- So indices skip 32-element arrays corresponding to rows (row-major).

```
mm:  ...
     li  $t1,32          # row size/loop end
     li  $s0,0           # init i=0
Li:  li  $s1,0           # init j=0
Lj:  li  $s2,0           # init k=0

     sll  $t2,$s0,5      # row-size of x
     addu $t2,$t2,$s1    # $t2=i*32+j
     sll  $t2,$t2,3      # byte offset of [i][j]
     addu $t2,$a0,$t2    # add base address to offset
     l.d  $f4,0($t2)     # $f4 = 8 bytes of x[i][j]
```

Example (cont'd)

```
Lk:   sll  $t0,$s2,5      # row-size of z
      addu $t0,$t0,$s1    # $t0=i*32+j
      sll  $t0,$t0,3      # byte offset of [k][j]
      addu $t0,$a2,$t0    # add base address to offset
      l.d  $f16,0($t0)    # $f4 = 8 bytes of z[k][j]

      sll  $t2,$s0,5      # row-size of y
      addu $t0,$t0,$s2    # $t0=i*32+k
      sll  $t0,$t0,3      # byte offset of [i][k]
      addu $t0,$a1,$t0    # add base address to offset
      l.d  $f18,0($t0)    # $f18 = 8 bytes of y[i][k]

      mul.d  $f16,$f18,$f16  # $f16=y[i][k]*z[k][j]
      add.d  $f4,$f4,$f16    # $f4=x[i][j]   +y[i][k]*z[k][j]

      addiu  $s2,$s2,1    # k++
      bne    $s2,$t1,Lk   # k-loop
      s.d    $f4,0($t2)   # x[i][j]=$f4

      addiu  $s1,$s1,1    # j++
      bne    $s1,$t1,Lj   # j-loop

      addiu  $s0,$s0,1    # i++
      bne    $s0,$t1,Li   # i-loop
      . . .
```