EECE 321: Computer Organization

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

Lecture 15: Floating-Point Arithmetic

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

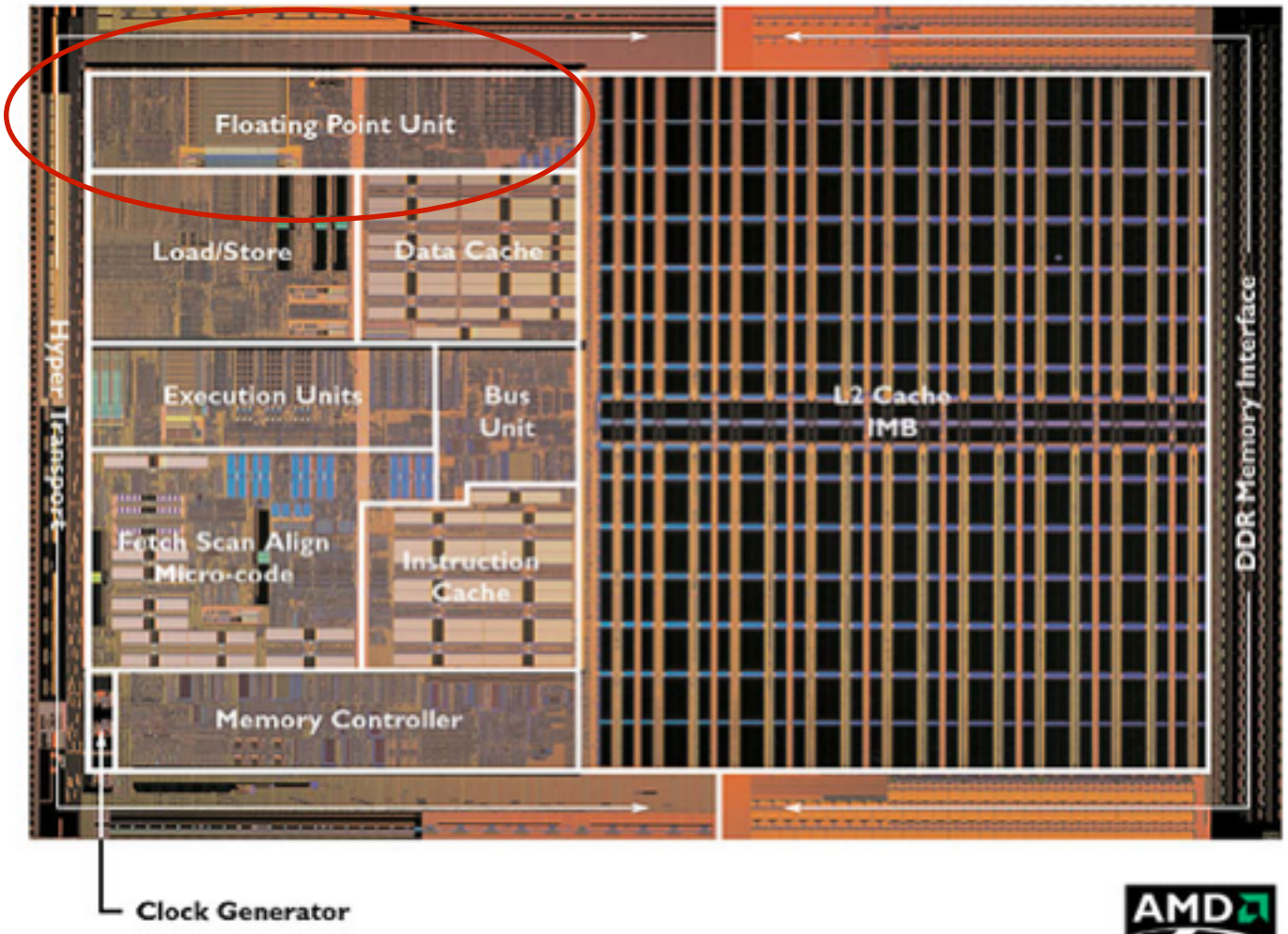
MIPS Floating Point Architecture

- 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
      . . .
```

Performance

- Reading assignment:
  - Sections 1.4, 1.5, 1.8

Defining Performance

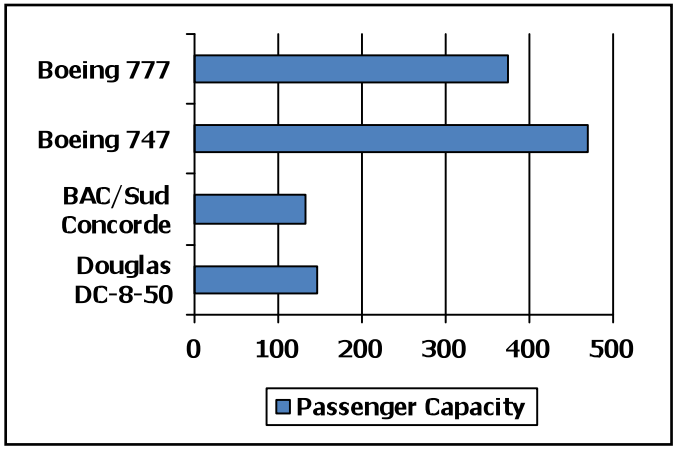
- In this part of the course we are concerned with assessing the performance of a computer.
- Why is performance important?
  - It enables making intelligent choices
  - See through the marketing hype: Does it really work as fast as they claim?
  - It is key to understanding underlying organizational motivation.
- How do we compare different computers? Why is some hardware better than others for different programs?
- What factors of system performance are hardware related?
  - For example, do we need a new machine, or a new operating system?
- How does the machine's instruction set affect performance?
  - Do we need more simple instructions, or few complex instructions?
  - What type of instructions do we need to include?
    - Ex: For multimedia applications, Intel added specific MMX instructions
- To answer these questions, we need to understand what determines the performance of a machine.

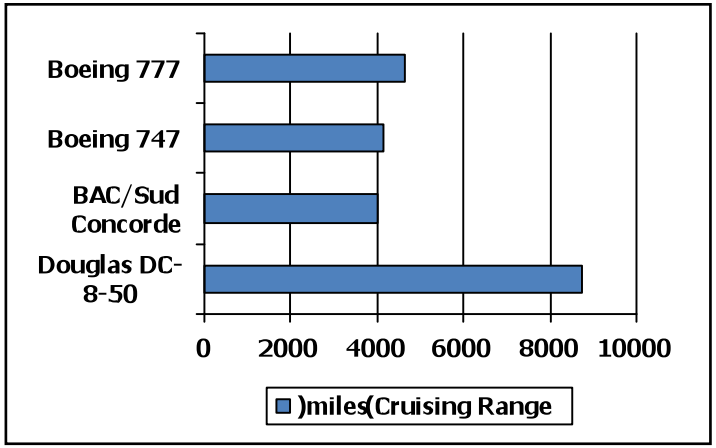
Which of these airplanes has the best performance?

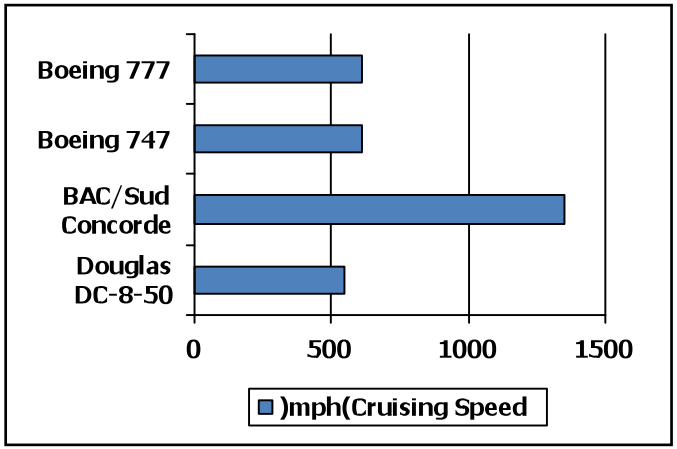
Airplane	Passengers	Range (mi)	Speed (mph)
Boeing 777	375	4630	610
Boeing 747	470	4150	610
BAC/Sud Concorde	132	4000	1350
Douglas DC-8-50	146	8720	544

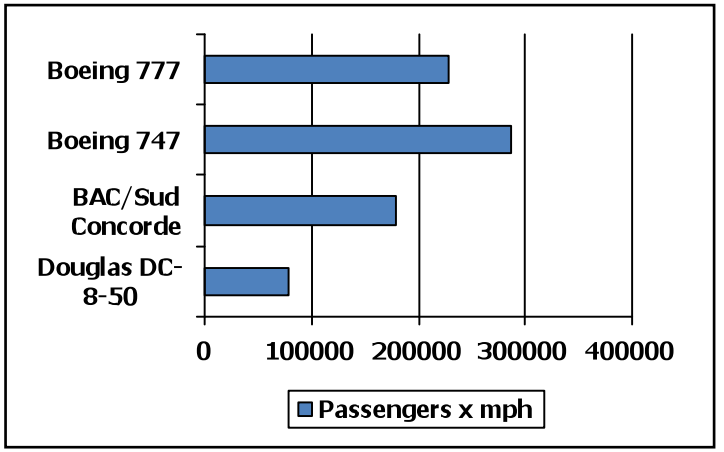
- Performance: Fastest, largest, longest range?
- Which plane transfers a single passenger 4000 miles in the shortest time?
- Which plane transfers 470 passengers 4000 miles in the shortest time?
- Performance can refer to completing a job as quickly as possible, or completing the most jobs in a given time.
- Example:
  - A program is running on 2 different workstations, the faster workstation is the one that gets the job done first.
  - Here one is interested in reducing *response time* or *execution time*.
  - If a computer center maintains two time-shared computers that run jobs submitted by many users, the faster computer is the one that completes the most jobs per day.
  - Here one is interested in maximizing *throughput*.

Which airplane has the best performance?









Computer Performance: TIME

- Metric: Response Time (latency)
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?
- Metric: Throughput
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?
- If we upgrade a machine with a new processor what do we increase?
- If we add a new machine to the lab what do we increase?
- In discussing the performance of machines, we will be primarily concerned with CPU execution time.
  - This is the time spent executing the lines of code that are "in" our program.
  - Time spent on I/O, or running other programs, or OS time is not included.

Performance Metrics

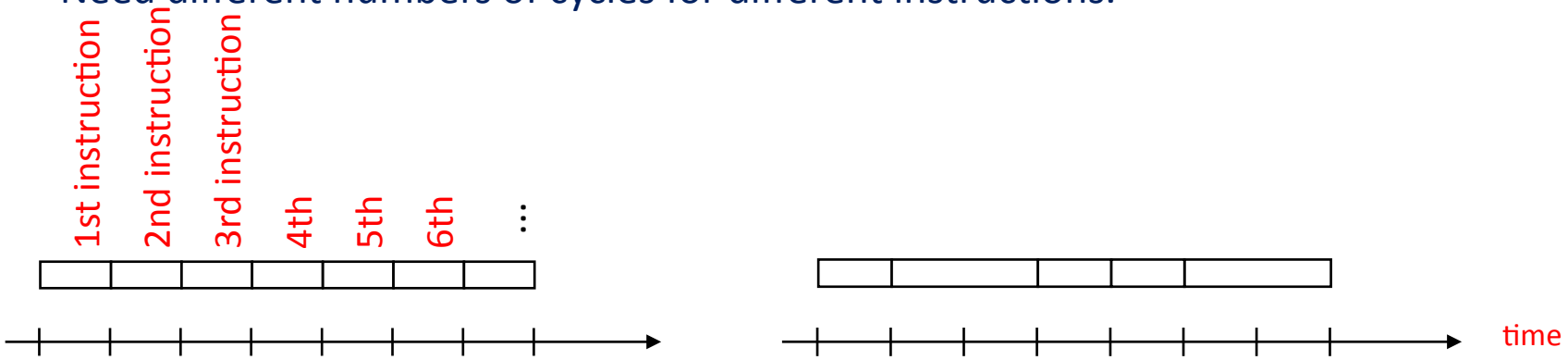
- For some program running on machine X,
  $\text{Performance}_X = 1 / \text{Execution time}_X$
- "X is *n* times faster than Y"
  $\text{Performance}_X / \text{Performance}_Y = n$
- Problem:
  - machine A runs a program in 20 seconds
  - machine B runs the same program in 25 seconds
  - Which is faster and by how much?
- Instead of reporting execution time in seconds, we often use clock cycles:

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- CPU execution time = # CPU clock cycles x Clock cycle time.
  = CPU clock cycles / Clock frequency.
- So, to improve performance (everything else being equal) you can either
  - Decrease the # of required cycles for a program,
  - Decrease the clock cycle time or, said another way, increase the clock rate.

How many cycles are required for a program?

- Could assume that # of cycles = # of instructions
- Incorrect assumption: Different instructions take different amounts of time on different machines.
- Need different numbers of cycles for different instructions.



- Multiplication takes more time than addition; Floating point operations take longer than integer ones; Accessing memory takes more time than accessing registers.
- Can compute average clock cycles per instruction (CPI), so
  - CPU clock cycles = Instructions per program x Avg. CC per instruction.
- Note: Changing the cycle time often changes the number of cycles required for various instructions.

Putting Things Together

- So, CPU time = Instruction count x CPI x Clock cycle time.

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instruction} \times \frac{Seconds}{Clock\ cycle}$$

- A given program will require
  - some number of instructions
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction): a floating point intensive application might have a higher CPI

- Another performance metric: MIPS (millions of instructions per second)
  this would be higher for a program using simple instructions

$$\text{MIPS} = \frac{\#\text{Instructions}}{\text{Exec. Time} \times 10^6}$$

Performance

- Performance is determined by execution time. Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction (CPI)?
  - average # of instructions per second?
- Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

Example 1: Cycles Per Instruction

- Suppose we have two implementations (machine A and machine B) of the same instruction set architecture (ISA).

  For some program,

  Machine A has a clock cycle time of 1 ns and a CPI of 2.0
  Machine B has a clock cycle time of 2 ns and a CPI of 1.2

  Which machine is faster for this program, and by how much?

- Answer: $$\frac{CPU\ performance_A}{CPU\ performance_B} = \frac{Execution\ time_B}{Execution\ time_A} = \frac{I \times 1.2 \times 2}{I \times 2 \times 1} = 1.2$$

- Hence machine A is 1.2 times faster than machine B for this program.

- If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?