

EECE 417 Computer Systems Architecture

**Department of Electrical and Computer Engineering
Howard University**

Charles Kim

Spring 2007

Computer Organization and Design (3rd Ed)

-The Hardware/Software Interface

by

**David A. Patterson
John L. Hennessy**

Chapter Three

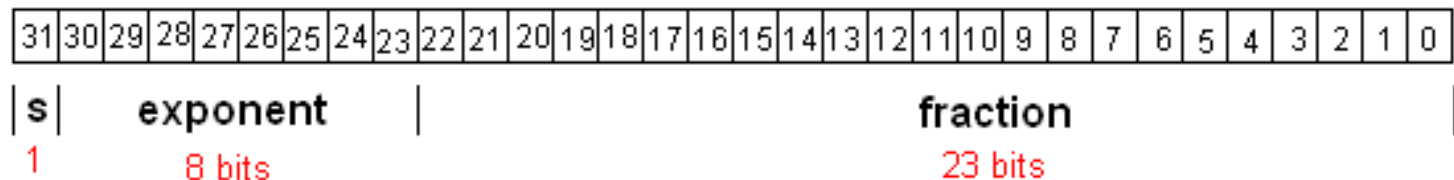
Arithmetic for Computers - Part B

Floating Point

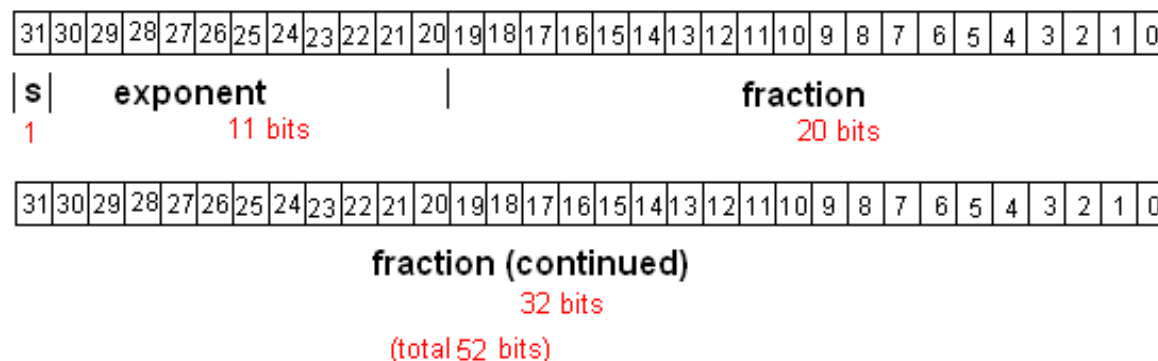
- *Reals*
- We need a way to represent *reals*
 - numbers with fractions, e.g., 3.1416 (**decimal** point)
 - very small numbers, e.g., .000000001 or 1.0×10^{-9}
 - very large numbers, e.g., 3.15576×10^9
- Scientific Notation & Normalized Scientific Notation (no leading 0)
 - 1.0×10^{-9} (normalized)
 - **0.1** $\times 10^{-8}$ (**Not** normalized)
 - **10.0** $\times 10^{-10}$ (**Not** normalized)
- Scientific Notation for Binary Numbers
 - $1.0 \times 2^{-1} = 0.1$ (**binary** point)
 - 0.01 ---> 1.0×2^{-2}
 - 100000 ---> 1.0×2^5
- Why the term “floating point”?
 - computer arithmetic that supports numbers in which **binary point is not fixed**

Floating-Point Representation

- Representation:
 - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
 - more bits for significand gives more accuracy
 - more bits for exponent increases range
 - But we have fixed word size
- IEEE 754 floating point standard:
 - **single precision**: 8 bit exponent, 23 bit significand



- **double precision**: 11 bit exponent, 52 bit significand



Floating-Point Representation

- “Fraction” vs. “significand”
 - Significand: 24 bit number (including the leading 1)
 - fraction: 23 bit number (without the leading 1)
- Leading “1” bit of significand is implicit
 - What if just 0 ---> then exponent 0
- IEEE 754 Encoding of floating-point numbers

Single Precision		Double Precision		Object
Exponent	Fraction	Exponent	Fraction	Represented
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormaized
1- 254	anything	1 - 2046	anything	± floating point
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (not a number)

Floating-Point

- Exponent is “biased” to make sorting easier
 - exponent comes first, then fraction later
 - all 0s is smallest exponent all 1s is largest
 - **bias of 127 for single precision** and 1023 for double precision
 - for positive and negative exponents
 - 00000000 for most negative number
 - 11111111 for most positive number
 - exponent of 0 ---->127-->0111 1111
 - exponent of 1 ---->127+1 --> 1000 0000
 - exponent of -1 ---->127-1 --->0111 1110
- **summary 1: value represented by a floating number is:**
$$(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent} - \text{bias}}$$
- **summary 2: representation of a value by the floating point notation:**
$$(\text{exponent} + \text{bias}) \text{----> "exponent" (8-bit)}$$

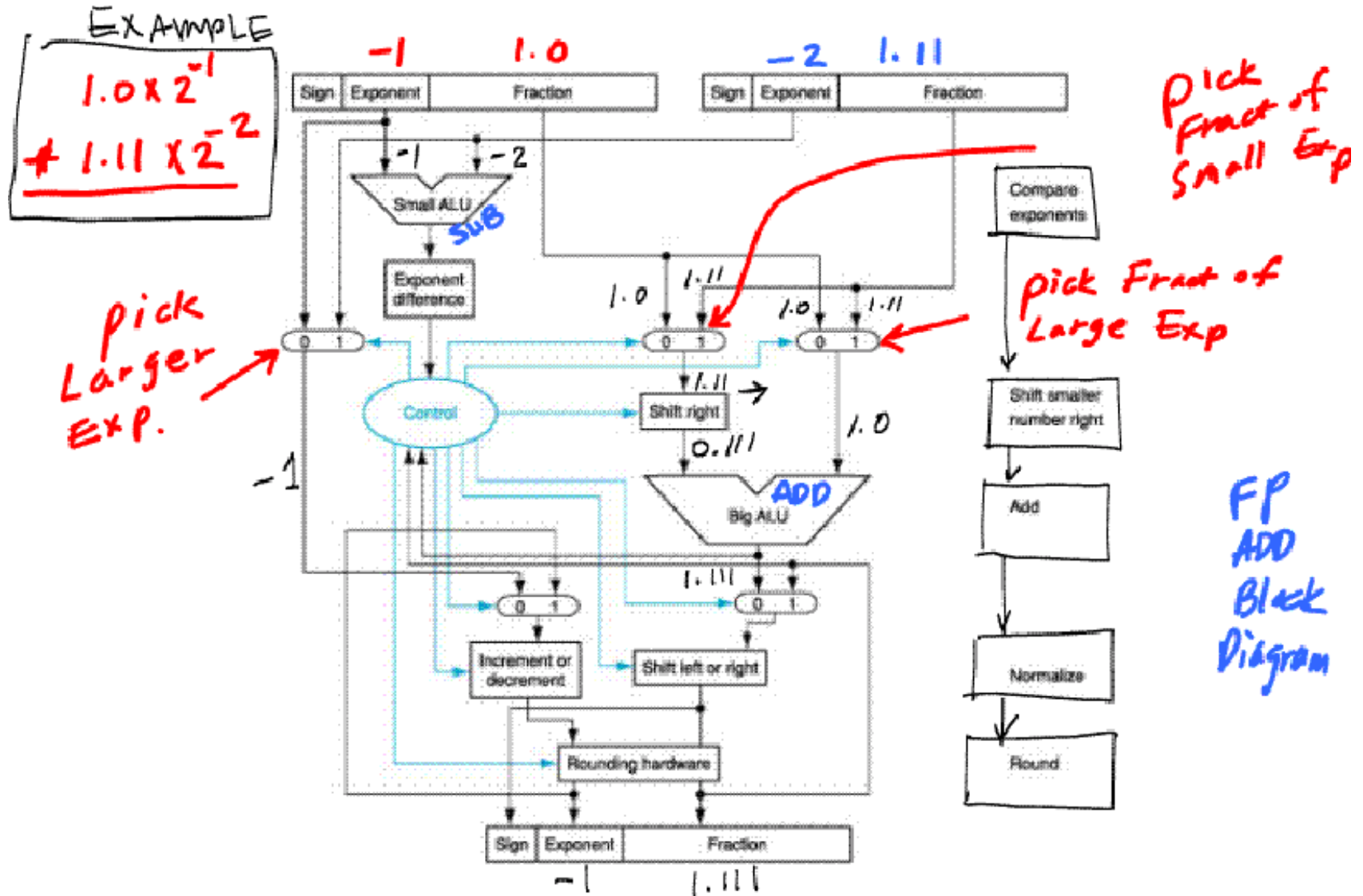
Floating Point Examples

- Reading a floating point number
 - 0xC0A00000 = ?
 - 1100 0000 1010 0000 0000 0000 0000 0000
 - s < exp > < fraction >
 - negative number with exp=129 and fraction=.010000000000000000
 - $-(1.01) \times 2^{(129-127)} = -(1.01) \times 2^2 = -101 \text{ ----} \rightarrow -5$
 - EXAMPLE
 - Convert the floating point numbers into a decimal numbers
 - 0xAD100000 ---->
 - 0x24924000 ---->
- Conversion to a floating point number
 - decimal: $-.75 = -(\frac{1}{2} + \frac{1}{4})$
 - binary: $-.11 = -1.1 \times 2^{-1}$
 - floating point: exponent = $-1+127=126 = 01111110$
 - IEEE single precision: 1 01111110 100000000000000000000000
 - Example
 - 0.625 ----> floating Point

Floating Point Addition

- Decimal Number Case Illustration (up to 4 decimal digits)
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- Step 1
 - Align the number that has smaller exponent so that its exponent matches the exponent of the larger number
 - $1.610 \times 10^{-1} \rightarrow 0.01610 \times 10^1 \rightarrow 0.016 \times 10^1$ (only 4 digits)
- Step 2
 - Addition of the significands $(9.999 + 0.016 = 10.015) \times 10^1$
- Step 3
 - Normalization: 1.0015×10^2
- Step 4
 - Rounding the number to 4 digits
 - 1.002×10^2

Floating point addition block diagram



Floating Point Addition Example

- Floating point addition of $0.5 + (-0.4375)$ in **binary** version
- Step 0 - Floating Point Notation
 - $0.5 \rightarrow 0.1 = 1.0 \times 2^{-1}$
 - $-0.4375 \rightarrow -0.0111 = -1.11 \times 2^{-2}$
- Step 1 - Alignment with larger exponent
 - 1.0×2^{-1}
 - $-1.11 \times 2^{-2} = -0.111 \times 2^{-1}$
- Step 2 - Addition of significands
 - $(1.0 + (-0.111)) \times 2^{-1} = 0.001 \times 2^{-1}$
- Step 3 - Normalization
 - $0.001 \times 2^{-1} = 1.000 \times 2^{-4}$
- Step 4 - Rounding
 - $1.000 \times 2^{-4} \rightarrow 0.0625$

Floating Point Multiplication

- Example First
 - $(1.11 \times 10^{10}) \times (9.200 \times 10^{-5})$
 - Limitation: 4 digits of significand and 2 digits for exponent
- Step 1 - Addition of two exponents
 - $10 + (-5) = 5$
- Step 2 - Multiplication of significands
 - $1.11 \times 9.200 \rightarrow 1100 \times 9200$ (with decimal point six digits from the right of the product)
 - $10212000 \rightarrow 10.2120000 \rightarrow 10.212 \times 10^5$.
- Step 3 - Normalization
 - $10.212 \times 10^5 = 1.0212 \times 10^6$.
- Step 4 - Rounding
 - $1.0212 \times 10^6 \rightarrow 1.021 \times 10^6$.
- Step 5 - Sign
 - $+ 1.021 \times 10^6$ (both have the same sign)

Floating Point Multiplication Example

- Floating point multiplication of 0.5 and -0.4375 in binary version
- Step 0 - floating point notation
 - $0.5 \rightarrow 1.000 \times 2^{-1}$
 - $-0.4375 \rightarrow -1.110 \times 2^{-2}$
- Step 1 - Adding the exponents
 - $-1 + (-2) = -3$
- Step 2 - Multiplying the significands
 - 1000×1110 (with binary points sixth digit from right) = 1.110000
 - With exponent: 1.110000×2^{-3} .
- Step 3 - Normalization
- Step 4- Rounding
 - $1.110000 \times 2^{-3} \rightarrow 1.110 \times 2^{-3}$.
- Step 5 - Sign
 - $- 1.110 \times 2^{-3}$.

Floating Point in MIPS

- **MIPS Floating Point**
 - 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 has hardware for double precision (64 bit) floating point operations.
 - Uses pairs of single precision registers to hold operands.
 - There are 16 pairs, named **\$f0, \$f2, — \$f30**. (even numbered register)
- Some MIPS processors allow only even-numbered registers (\$f0, \$f2,...) for single precision instructions. However **SPIM** allows all 32 registers in single precision instructions.

Floating Point Instructions

- **Arithmetic**

- `add.s $f2, $f4, $f6` `# $f2=$f4+$f6`
- `sub.s $f2, $f4, $f6` `#s --single precision`
- `mul.s $f2, $f4, $f6`
- `div.s $f2, $f4, $f6`
- `add.d $f2, $f4, $f6` `#d -- double precision`
- `sub.d $f2, $f4, $f6`
- `mul.d $f2, $f4, $f6`
- `div.d $f2, $f4, $f6`

- **Data Transfer**

- `lwc1 $f1, 100($s2)` `#load word from coprocessor 1`
- `swc1 $f1, 100($s2)` `#store word to coprocessor 1`

- **Conditional Branch**

- `c.lt.s $f2, $f4` `#cond=1 if $f2<$f4`
- `c.lt.d $f2, $f4`
- `bclt 25` `#if cond==1 (true), PC-rel branch`
- `bclf 25` `#if cond==0 (false), PC-rel branch`

Floating Point Example (p.209)

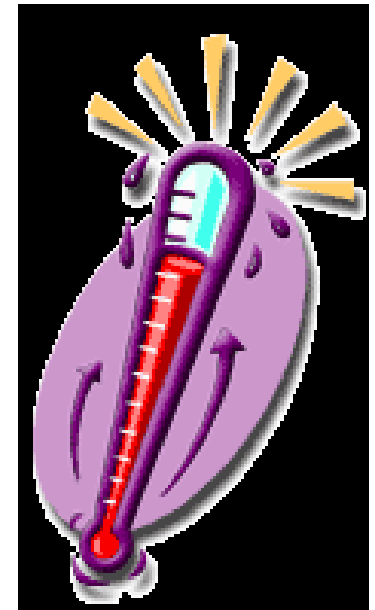
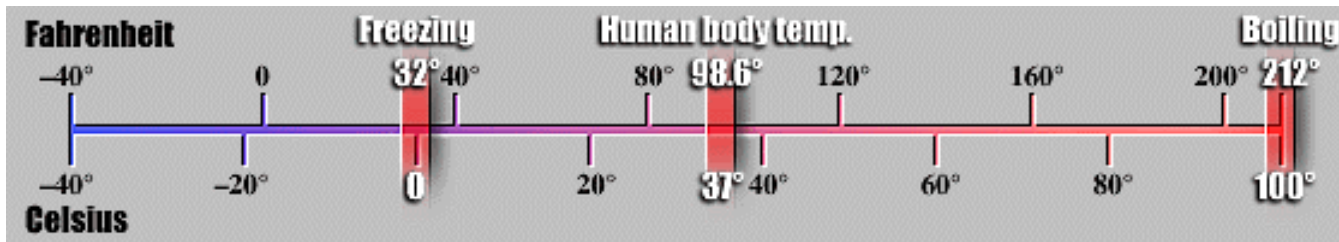
- Conversion of temperature from Fahrenheit to Celsius

```
float f2c (float fahr)
```

```
{
```

```
    return ((5.0/9.0)*(fahr-32.0));
```

```
}
```



Floating Point Example - Conversion f2c (p.209) 1/2

```
p209.asm - Notepad
File Edit Format View Help

#p209.asm
#Floating Point
#Conversion from Fahrenheit to Celsius
# float f2c (float fahr)
#
# {
#     return ((5.0/9.0)*(fahr-32.0));
# }
# We will read fahr from keyboard
# fahr--->$f0
# cel --->$f12
# Base mem addre for fp --->$s2
main:
    .data    0x10010000                #starting address
    .asciiz  "\nTemperature Conversion from Fahrenheit to Celsius"
    .data    0x10010100                #starting address
    .asciiz  "\nType Temperature in Fahrenheit: "
    .data    0x10010200                #starting address
    .asciiz  "\nThe Temperature in Celsius is: "
    .data    0x10010300
    .float   5.0, 9.0, 32.0            #single
    .text                                #code part
    ori      $v0, $zero, 4             #msg1
    lui      $a0, 0x1001
    ori      $a0, $a0, 0
    syscall
    ori      $v0, $zero, 4             #msg2
    lui      $a0, 0x1001
    ori      $a0, $a0, 0x0100
    syscall
```

Floating Point Example - Conversion f2c (p.209) 2/2

```
        ori    $v0, $zero, 6    #read fp input
        syscall                               #now type-in fp is in $f0
#5.0 ---> $f16
        lui    $s2, 0x1001
        ori    $s2, $s2, 0x0300    #base addr of fp
        lwcl   $f16, 0($s2)
        lwcl   $f18, 4($s2)
        lwcl   $f20, 8($s2)
#9.0 --->$f18
#32.0 --->$f20
        div.s   $f16, $f16, $f18    #f16=5.0/9.0
        sub.s   $f20, $f0, $f20    #f20=fahr-32.0
        mul.s   $f12, $f16, $f20    #f12=(5.0/9.0)*(fahr-32.0)
#result
        ori    $v0, $zero, 4    #msg3
        lui    $a0, 0x1001
        ori    $a0, $a0, 0x0200
        syscall
#Print the result
        ori    $v0, $zero, 2    #request for fp print ($f12)
        syscall
        j      main
```

Check with SPIM

```
.data    0x10010300
.float   5.0, 9.0, 32.0
```

```
[0x10010300]                0x40a00000  0x41100000  0x42000000  0x00000000
[0x10010310] ... [0x10040000] 0x00000000
```

PCSpim

File Simulator Window Help

Single Floating Point Registers

FP0 = 44.0000	FP8 = 0.000000	FP16 = 5.00000	FP24 = 0.000000
FP1 = 0.000000	FP9 = 0.000000	FP17 = 0.000000	FP25 = 0.000000
FP2 = 0.000000	FP10 = 0.000000	FP18 = 9.00000	FP26 = 0.000000
FP3 = 0.000000	FP11 = 0.000000	FP19 = 0.000000	FP27 = 0.000000
FP4 = 0.000000	FP12 = 0.000000	FP20 = 32.0000	FP28 = 0.000000

[0x0040004c] 0x3c121001 lui \$t8, 4097 ; 33:
[0x00400050] 0x36520300 ori \$t8, \$t8, 768 ; 34:
[0x00400054] 0xc6500000 lwc1 \$f16, 0(\$t8) ; 35:
[0x00400058] 0xc6520004 lwc1 \$f18, 4(\$t8) ; 36:
[0x0040005c] 0xc6540008 lwc1 \$f20, 8(\$t8) ; 37:
[0x00400060] 0x46128403 div.s \$f16, \$f16, \$f18 ; 40:

Console

Temperature Conversion from Fahrenheit to Celsius

Type Temperature in Fahrenheit: 44

[0x10010200] 0x6568540a 0x6d655420 0x61726570
[0x10010210] 0x206e6920 0x736c6543 0x20737569
[0x10010220] ... [0x10010300] 0x00000000
[0x10010300] 0x40a00000 0x41100000 0x42000000
[0x10010310] ... [0x10040000] 0x00000000

Two-Dimensional Matrices (p.210)

- $X = X + Y * Z$
- X, Y, Z: Square matrices of 4x4
- Double Precision Calculation

```
void mm (double x[][], double y[][], double z[][])  
{  
    int i, j, k;  
    for (i=0; i!=4; i=i+1)  
        for (j=0; j!=4; j=j+1)  
            for (k=0; k!=4; k=k+1)  
                x[i][j]=x[i][j]+y[i][k]*z[k][j];  
}
```

- \$a0, \$a1, and \$a2 : Base addrs of X, Y, and Z, respectively
- \$s0, \$s1, and \$s2: integer variables of i, j, and k, respectively

Array Layout

- Row Major Order
 - First row elements, then second row elements, etc
- No pseudoinstruction
 - `li`, `l.d`, `s.d` (not here!)
- Core Instructions only (with directives)
 - `double d1, d2, etc` # declaring double
 - # precision fp
 - `lwc1` #load single precision fp
 - `swc1` #store single precision fp
- Loop structure
 - Do $Y * Z$ first
 - Then Do $X + Y * Z$
 - Keep for k, j, i

s: base addr

A(0,0)	S+0
A(0,1)	S+8
A(0,2)	S+16
A(0,3)	S+24
A(1,0)	S+32
A(1,1)	S+40
A(1,2)	

**The addr of A(i,j)
= $S + [i * 4 + j] * 8$**

Double Precision Floating Point Multiplication (p.210) 1/4

```
p210.asm - Notepad
File Edit Format View Help

#p210.asm
|
main:
    .data    0x10010000                #starting address of first string
    .asciiz  "\nDouble Precision Matrix Multiplication\n"  #msg1
    .data    0x10010040
    .asciiz  "Finished\n"
    .data    0x10010100
    .double  1.1, 1.2, 1.3, 1.4        #X
    .double  2.1, 2.2, 2.3, 2.4
    .double  3.1, 3.2, 3.3, 3.4
    .double  4.1, 4.2, 4.3, 4.4
    .data    0x10010200
    .double  10.1, 10.2, 10.3, 10.4    #Y
    .double  20.1, 20.2, 20.3, 20.4
    .double  30.1, 30.2, 30.3, 30.4
    .double  40.1, 40.2, 40.3, 40.4
    .data    0x10010300
    .double  0.11, 0.12, 0.13, 0.14    #Z
    .double  0.21, 0.22, 0.23, 0.24
    .double  0.31, 0.32, 0.33, 0.34
    .double  0.41, 0.42, 0.43, 0.44
```

Double Precision Floating Point Multiplication (p.210) 2/4

```
.text                                     #code part
ori    $v0, $zero, 4    #msg1
lui    $a0, 0x1001
ori    $a0, $a0, 0
syscall
lui    $a0, 0x1001
ori    $a0, $a0, 0x0100                #Base addr of X
lui    $a1, 0x1001
ori    $a1, $a1, 0x0200                #Base addr of Y
lui    $a2, 0x1001
ori    $a2, $a2, 0x0300                #Base addr of Z
ori    $t0, $zero, 4                  # $t0=4=size of matrix
ori    $s0, $zero, 0                  # i=0
L1:    ori    $s1, $zero, 0            # j=0
L2:    ori    $s2, $zero, 0            # k=0
#loading X[i][j] into $f4
# Addr of X(i,j)=Base_Addr + (i*4+j)*8
sll    $t2, $s0, 2                    #i*4
addu   $t2, $t2, $s1                  #i*4+j
sll    $t2, $t2, 3                    #(i*4+j)*8
addu   $t2, $a0, $t2                  #(i*4+j)*8 + Base_addr
lwc1   $f4, 0($t2)                    # First 4 bytes
lwc1   $f5, 4($t2)                    # Second 4 bytes
```

Double Precision Floating Point Multiplication (p.210) 3/4

```
L3:
#Loading of Y[i][k] into $f8
# Addr of Y(i,k)=Base_Addr + (i*4+k)*8
    sll    $t3, $s0, 2           #i*4
    addu   $t3, $t3, $s2        #i*4+k
    sll    $t3, $t3, 3          #(i*4+k)*8
    addu   $t3, $a1, $t3        #(i*4+k)*8 + Base_addr
    lwc1   $f8, 0($t3)          # First 4 bytes
    lwc1   $f9, 4($t3)         # Second 4 bytes

# Loading of Z[k][j] into $f10
# Addr of Z(k,j)=Base_Addr + (k*4+j)*8
    sll    $t4, $s2, 2           #k*4
    addu   $t4, $t4, $s1        #k*4+j
    sll    $t4, $t4, 3          #(k*4+j)*8
    addu   $t4, $a2, $t4        #(k*4+j)*8 + Base_addr
    lwc1   $f10, 0($t4)         # First 4 bytes
    lwc1   $f11, 4($t4)        # Second 4 bytes

#Multiplication
    mul.d  $f8, $f8, $f10       #Y=Y*Z
    add.d  $f4, $f4, $f8        #X=X+Y*Z
```

Double Precision Floating Point Multiplication (p.210) 4/4

```
#increment of k
    addiu    $s2, $s2, 1          #k=k+1
    bne      $s2, $t0, L3
    swc1     $f4, 0($t2)          #first
    swc1     $f5, 4($t2)          #second

#increment of j
    addiu    $s1, $s1, 1          #j=j+1
    bne      $s1, $t0, L2
    addiu    $s0, $s0, 1          #i=i+1
    bne      $s0, $t0, L1

    ori      $v0, $zero, 4        #msg1
    lui      $a0, 0x1001
    or       $a0, $a0, 0x0040
    syscall
```

- Care for printing out the result?

Rounding

- Floating Point Number are normally approximations
 - Why?
 - Infinite variety of real numbers, but
 - 2^{53} ways of expression in double precision fp
 - IEEE 754 Rounding Modes of Approximation
 - 2 extra bits on the right during intermediate
 - guard bit and round bit

- Guard and Round bits - Illustration

- Addition Example (3 significant decimal digits)

$$2.56 \times 10^0 + 2.34 \times 10^2$$

- Without Guard and Round

$$\begin{array}{r} 0.02 \\ 2.34 \\ \hline 2.36 \end{array} \rightarrow 2.36 \times 10^2$$

- With Guard and Round

guard
round

$$\begin{array}{r} 0.02\text{56} \\ 2.34\text{00} \\ \hline 2.36\text{56} \end{array} \rightarrow 2.37 \times 10^2$$

00-49 : round down
51-99 : round up

Accuracy in floating points

- Measure of accuracy
 - the number of errors in the LSBs of the significands
 - “units in the last place” ---> ulp
 - Problem when a number is half-way in-between (i.e., 0.5--->0? 1?)
 - Norm --round to nearest even number
 - a third bit - “sticky” bit (next to the *guard* and *round*)
 - the sticky bit is set whenever there are nonzero bits to the right of the round bit
 - Sticky bit example
- Without sticky bit
 - $5.01 \times 10^{-1} + 2.34 \times 10^2$ (3 significant decimal digits)
 - 0.00501 ---> 0.0050
 - 2.3400 2.3400
 - 2.3450 ---> 2.34 (nearest even)
- With sticky bit
 - 0.00501 ---> 0.00501
 - 2.34000 ---> 2.3400
 - 2.34501 ----> 2.35 (rounded up from .501)

Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
 - two's complement
 - IEEE 754 floating point
- Computer instructions determine “meaning” of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines