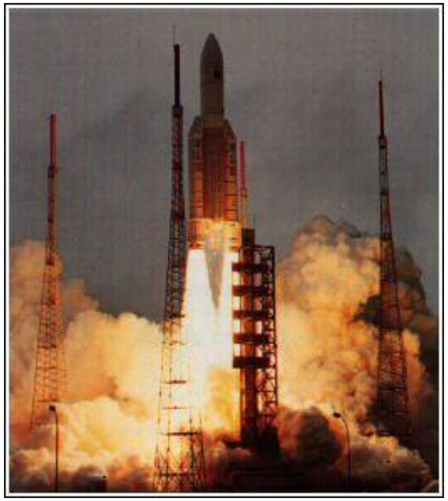
Computer Organization

Lecture 12 - Floating Point

Reading: 3.5-3.9

Homework: in UCLASS



Why did the Ariane 5 Explode?
(image source: `java.sun.com`)

Outline - Floating Point

- **Motivation and Key Ideas**
- **IEEE 754 Floating Point Format**
- **Range and precision**
- **Floating Point Arithmetic**
- **MIPS Floating Point Instructions**
- **Rounding & Errors**
- **Summary**

Floating Point - Motivation

- **Review: n-bit integer representations**
  - **Unsigned:** **0 to 2^n-1**
  - **Signed Two's Complement:** **- 2^{n-1} to 2^{n-1}-1**
  - **Biased (excess-b):** **-b to 2^n-b**

- **Problem: how do we represent:**
  - **Very large numbers** **9,345,524,282,135,672, 2^{354}**
  - **Very small numbers** **0.00000000000000005216, 2^{-100}**
  - **Rational numbers (유리수)** **2/3**
  - **Irrational numbers (무리수)** **sqrt(2), e, π**
  - **Transcendental numbers (초월수)** **e, π**

Fixed Point Representation

- Idea: fixed-point numbers with fractions
- Decimal point (binary point) marks start of fraction
  - Decimal: $3.2503 = 3 \times 10^{0} + 2 \times 10^{-1} + 5 \times 10^{-2} + 3 \times 10^{-4}$
  - Binary: $1.0100001 = 1 \times 2^{0} + 1 \times 2^{-2} + 1 \times 2^{-7}$
- Problems
  - Limited locations for “decimal point” (binary point”)
  - Won’t work for very small or very larger numbers

Another Approach: Scientific Notation

- **Represent a number as a combination of**
  - **Significand (mantissa): Normalized number**
    AND
  - **Exponent (base 10)**
- **Example: 6.02×10^{23}** ← **Exponent**

Significand (mantissa)

Radix (base)

Floating Point

- **Key idea: adapt scientific notation to binary**
  - **Fixed-width binary number for significand**
  - **Fixed-width binary number for exponent (base 2)**
- **Idea: represent a number as**

$$1.xxxxxxx_{two} \times 2^{yyyy}$$

Exponent

Leading '1' (Implicit)

Significand (mantissa)

Radix (2)

Important Points:

This is a <u>tradeoff</u> between precision and range 정밀도와 범위

Arithmetic is <u>approximate</u> - error is inevitable!

Outline - Floating Point

- **Motivation and Key Ideas**
- **IEEE 754 Floating Point Format**
- **Range and precision**
- **Floating Point Arithmetic**
- **MIPS Floating Point Instructions**
- **Rounding & Errors**
- **Summary**

IEEE 754 Floating Point

(Institute of Electrical and Electronics Engineers)

- **Single precision (C/C++/Java `float` type)**

S	E Exponent	F Significand
1 bit	8 bits	23 bits

$$\text{Value N} = (-1)^S \times 1.F \times 2^{E-127}$$

← **Bias**

- **Double precision (C/C++/Java `double` type)**

S	E Exponent	F Significand
1 bit	11 bits	20 bits

F Significand (continued - 52 bits total)
32 bits

$$\text{Value N} = (-1)^S \times 1.F \times 2^{E-1023}$$

← **Bias**

Floating Point Examples

$8.75_{10} = 1000.11_2$

- **8.75_{ten} =**
  - **Single Precision:**
    - **Significand:**
    - **Exponent:**

0	1	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

S **E Exponent** **F Significand**

  - **Double Precision:**
    - **Significand:**
    - **Exponent:**

0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

S **E Exponent** **F Significand**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

F Significand (continued - 52 bits total)

Floating Point Examples

$-0.375_{10} = -0.011_2$

- **-0.375_{ten} =**
  - **Single Precision:**
    - **Significand:**
    - **Exponent:**

1	0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

S **E Exponent** **F Significand**

  - **Double Precision:**
    - **Significand:**
    - **Exponent:**

1	0	1	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

S **E Exponent** **F Significand**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

F Significand (continued - 52 bits total)

Floating Point Examples

- **Q: What is the value of the following single-precision word?**

0	0 0 0 0 1 0 0 0	1 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
S	**E Exponent**	**F Significand**

- **Significand =**
- **Exponent =**
- **Final Result =**

Special Values in IEEE Floating Point

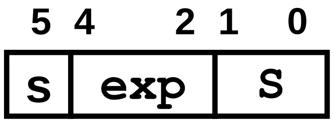
- **0000000 exponent - reserved for**
  - **zero value (all bits zero)**
  - **"Denormalized numbers" - drop the "1."**
    - **Used for "very small" numbers … "gradual underflow"**
    - **Smallest denormalized number (single precision):**
      $0.00000000000000000000001 \times 2^{-126} = 2^{-149}$
- **1111111 exponent**
  - **Infinity - 111111 exponent, zero significand**
  - **NaN (Not a Number) - 1111111 exponent, nonzero significand**

Outline - Floating Point

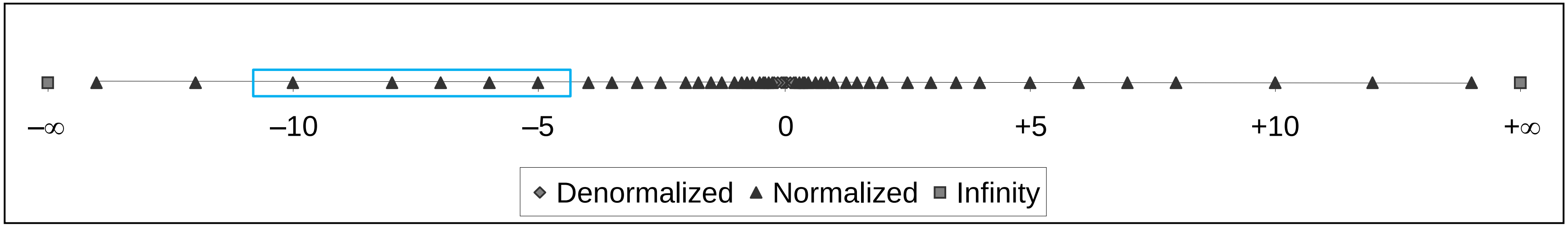
- Motivation and Key Ideas
- IEEE 754 Floating Point Format
- Range and precision
- Floating Point Arithmetic
- MIPS Floating Point Instructions
- Rounding & Errors
- Summary

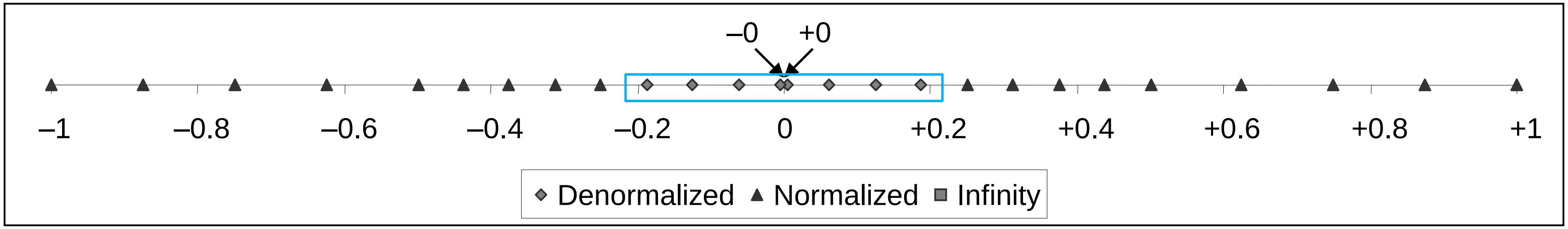
Floating Point Range and Precision

- The tradeoff: range in exchange for uniformity
- “Tiny” example: floating point with:
  - 3 exponent bits
  - 2 signficand bits



Graphic and Example Source:
R. Bryant and D. O’Halloran,
Computer Systems: A Programmer’s Perspective,
© Prentice Hall, 2002





Visualizing Floating Point - “Small” FP Representation

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of 7.
  - the last three bits are the `frac`
- Same General Form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

7	6 3	2 0
`s`	`exp`	`significand`

Example Source:
R. Bryant and D. O’Halloran,
Computer Systems: A Programmer’s Perspective,

Small FP - Values Related to Exponent

Exp	exp	E	2^E	
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf, Nan).

Lecture: L06

Biased

biased representation

real value

- Add a bias (offset) of $2^{n-1}-1$ to represent all numbers
  - Most negative number: 000...0 = $-(2^{n-1}-1)$
  - Zero: 011...1 = 0
  - Most positive number: 111...1 = $+2^{n-1}$

소숫점 이하값이 normalized 값은 1.xxx이고 denormalized 값은 0.xxx

Bias 값 = 7

Small FP Example - Dynamic Range

	s	exp	frac	*E*	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

Learning from Tiny & Small FP

- Non-uniform spacing of numbers
  - very small spacing for large negative exponents
  - very large spacing for large positive exponents
- Exact representation: sums of powers of 2

$$2^2+2^{-1}+2^{-3}$$

- Approximate representation: everything else

Summary: IEEE Floating Point Values

Single precision		Double precision		Object Represented
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	nonzero	0	nonzero	+/- denormalized number
1-254	anything	1-2046	anything	+/- floating-point number
255	0	2047	0	+/0 infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

Fig. 3.14 / p. 246

IEEE Floating Point - Interesting Numbers

Description	`exp`	`frac`	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}} = 2^{-\{149,1074\}}$
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$

Smallest Pos. Denorm.:
- Single $\approx 1.4 \times 10^{-45}$
- Double $\approx 4.9 \times 10^{-324}$

Largest Denormalized:
- Single $\approx 1.18 \times 10^{-38}$
- Double $\approx 2.2 \times 10^{-308}$

$\varepsilon = 2^{-\{23,52\}}$

Smallest Pos. Normalized:
- Just larger than largest denormalized

Largest Normalized:
- Single $\approx 3.4 \times 10^{38}$
- Double $\approx 1.8 \times 10^{308}$

Outline - Floating Point

- Motivation and Key Ideas
- IEEE 754 Floating Point Format
- Range and precision
- Floating Point Arithmetic
- MIPS Floating Point Instructions
- Rounding & Errors
- Summary

Floating Point Addition
(4th Edition Fig. 3.15, 5th Edition Fig. 3.14 : Flowchart)

1. **Align** binary point to number with larger exponent
2. **Add** significands
3. **Normalize** result and adjust exponent;
   if overflow/underflow, throw exception
5. Round result (go to 3 if normalization needed again)

A	1.11×2^{0}	1.11×2^{0}	1.75
+ B	$+\ 1.00 \times 2^{-2}$	$+\ \underline{0.01 \times 2^{0}}$	$\underline{0.25}$
		10.00×2^{0}	
	(Normalize)	1.00×2^{1}	2.00

Floating Point Adder - Hardware

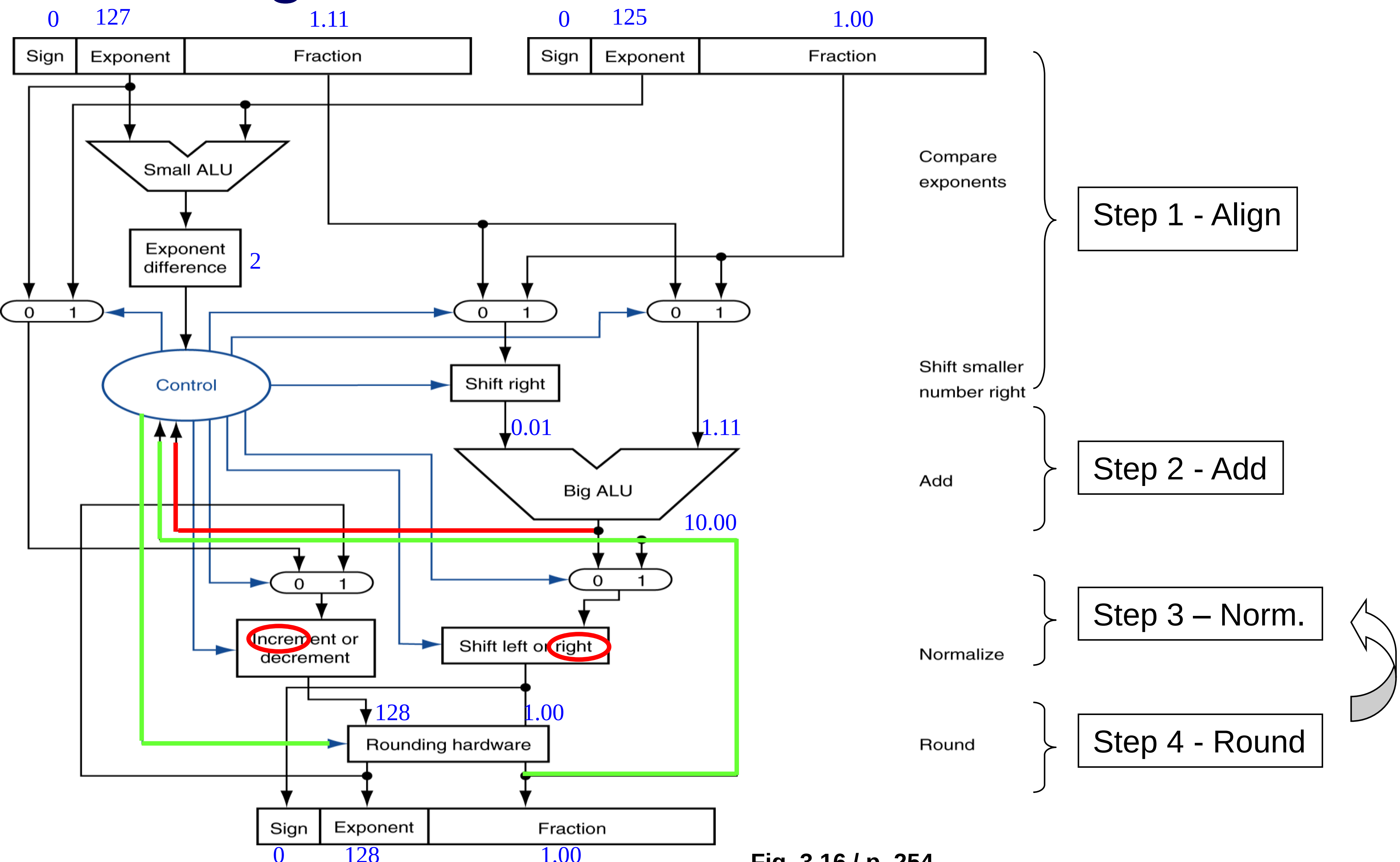


Fig. 3.16 / p. 254

Floating Point Adder - Hardware

- **Much more complex than integer adder**
- **Doing it in one clock cycle would take too long**
  - **Much longer than integer operations**
  - **Slower clock would penalize all instructions**
- **FP adder usually takes several cycles**
  - **Can be pipelined**

Floating Point Multiplication (Fig. 3.17)

1. Add 2 exponents together to get new exponent (subtract 127 to get proper biased value)
2. Multiply significands
3. Normalize result if necessary (shift right) & adjust exponent
4. If overflow/underflow throw exception
5. Round result (go to 3 if normalization needed again)
6. Set sign of result using sign of X, Y

Floating Point Multiplier - Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

Outline - Floating Point

- **Motivation and Key Ideas**
- **IEEE 754 Floating Point Format**
- **Range and precision**
- **Floating Point Arithmetic**
- **MIPS Floating Point Instructions**
- **Rounding & Errors**
- **Summary**

MIPS Floating Point Instructions

- Organized as <u>coprocessor</u> `c1` `(read as c one)`
  - Separate FP registers `$f0-$f31`
    - Double precision uses even/odd reg. pairs (e.g. `$f2-$f3`)
  - Separate operations
  - Separate data transfer (to same memory)
- Basic operations
  - `add.s` - single `add.d` - double
  - `sub.s` - single `sub.d` - double
  - `mul.s` - single `mul.d` - double
  - `div.s` - single `div.d` - double
- Examples
  - `add.s $f2, $f5, $f7` `$f2 = $f5 + f7`
  - `mul.d $f8, $f10, $f2` `$f8 = $f10 X $f2`

`$f8-$f9` `$f10-$f11` `$f2-$f3`

항상 짝수 register number
regs in pair : 64 bits

MIPS Floating Point Instructions (cont'd)

- Data transfer from FP to/from memory:
  - `lwc1`, `swc1` (`l.s`, `s.s`) - load/store float to/from fp reg
    - `lwc1 $f1, 100($s2)   # $f1 = Memory[$s2+100]`
  - `ldc1`, `sdc1` - load/store double to/from fp reg pair
    - `ldc1 $f2, 100($s2)   #f2-f3 = Memory[$s2+100]`
- Testing / branching
  - `c.lt.s`, `c.lt.d`, `c.eq.s`, `c.eq.d`, …
    compare and set condition bit if true
    - c.lt.s $f2, $f4 # if ($f2 < $f4), cond = 1; else cond = 0; single precision
    - c.lt.d $f2, $f4 # if (($f2-$f3) < ($f4-$f5)), cond = 1; else cond = 0; double precision (use a pair for comparison)
  - `bc1t`, `bc1f` - branch if condition true / false
    - bclt 25 # if (cond ==1) go to PC+4+100

FP Example: ° F to ° C

- **C code:**

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in $f12, result in $f0, literals in global memory space

- **Compiled MIPS code:**

```
f2c: lwc1   $f16, const5($gp)
     lwc1   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

```
main()
{
   ...
   f2c(75);
   ...
}
```

FP Example: Matrix Arithmetic

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements

$$\begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix}$$

- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of `x`, `y`, `z` in `$a0`, `$a1`, `$a2`, and `i`, `j`, `k` in `$s0`, `$s1`, `$s2`

FP Example: Matrix Arithmetic

```
    li   $t1, 32        # $t1 = 32 (row size/loop end)
    li   $s0, 0         # i = 0; initialize 1st for loop
L1: li   $s1, 0         # j = 0; restart 2nd for loop
L2: li   $s2, 0         # k = 0; restart 3rd for loop
```

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

```
L3: sll  $t0, $s2, 5    # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1  # $t0 = k * size(row) + j
    sll  $t0, $t0, 3    # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0  # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
    …
```

FP Example: Matrix Arithmetic

```
…
    sll   $t0, $s0, 5        # $t0 = i*32 (size of row of y)
    addu  $t0, $t0, $s2      # $t0 = i*size(row) + k
    sll   $t0, $t0, 3        # $t0 = byte offset of [i][k]
    addu  $t0, $a1, $t0      # $t0 = byte address of y[i][k]
    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 k + 1
    bne   $s2, $t1, L3       # if (k != 32) go to L3
    s.d   $f4, 0($t2)        # x[i][j] = $f4

    addiu $s1, $s1, 1        # $j = j + 1
    bne   $s1, $t1, L2       # if (j != 32) go to L2

    addiu $s0, $s0, 1        # $i = i + 1
    bne   $s0, $t1, L1       # if (i != 32) go to L1
```

Outline - Floating Point

- Motivation and Key Ideas
- IEEE 754 Floating Point Format
- Range and precision
- Floating Point Arithmetic
- MIPS Floating Point Instructions
- Rounding & Errors
- Summary

Rounding

0.01000001 (in 3-bit mentissa)

- **Extra bits allow rounding after computation**
  - **Guard digit (may shift into number during normalization)**
  - **Round digit - used to round when guard bit is shifted during normalization**
  - **Sticky bit - set when there are 1's to the right of the round digit.**

Example: $2.56 * 10^0 + 2.34 \times 10^2 = ?$

0-49: round down
51-99: round up
50: round to nearest even

Ex)
2.3650 => 2.36
2.3651 => 2.37

guard
round

2.3400
0.0256

2.3656 = 2.37

since 56 > 50
round up

Example with round-to-even (extra)

$2.56 * 10^0 + 2.34 \times 10^2 = ?$

Round-to-even

0-49: round down

51-99: round up

50: round to nearest even

Ex)

2.3650 => 2.36

2.3651 => 2.37

$$
\begin{array}{r}
2.3400 \\
0.0256 \\
\hline
2.3656 = 2.37 * 10^2
\end{array}
$$

guard

round

Since 56 > 5, round up

Sticky bit

Example:

$5.03 * 10^{-1} + 2.34 \times 10^{2} = ?$

0.0050 (sticky bit is set to 1 due to 3)

2.3400

2.34**<u>50</u>** => $2.34 * 10^{2}$

* if there is no sticky bit, 50 is rounded to the nearest even 2.34

With sticky bit = 1, the result is $2.35 * 10^{2}$ since **<u>50</u>** > 50 (see page 39).

IEEE 754 Rounding Modes

- IEEE 754 supports four rounding modes
  - Round to nearest even (default)
  - Round-Toward-Zero (Truncate)
  - Round up (toward +∞) Round down (toward -∞)

US ISR always rounds 0.5 dollars up

가까운 값으로 round하고 **중간 값은 가까운 짝수로 round**

- Example: "Dollar" rounding

Mode	$1.40	$1.51	$1.50	$2.50	-$1.50
Round-to-even	$1	$2	$2	$2	$-2
Round-toward-zero	$1	$1	$1	$2	$-1
Round-down	$1	$1	$1	$2	$-2
Round-up	$2	$2	$2	$3	$-1

Table Source: R. Bryant and D. O'Halloran, *Computer Systems: A Programmer's Perspective*, Prentice-Hall, 2003 (Fig. 2.25, p. 90)

Limitations on Floating-Point Math

- Most numbers are approximate
- Roundoff error is inevitable
- Range (and accuracy) vary depending on exponent
- "Normal" math properties not guaranteed:
  - Inverse `(1/r)*r ≠ 1`
  - Associative `(A+B) + C ≠ A + (B+C)`
    `(A*B) * C ≠ A * (B*C)`
  - Distributive `(A+B) * C ≠ A*B + B*C`
- Scientific calculations require error management take a numerical analysis for more info

Associativity not Guaranteed!

- **An example where associativity fails:**

x	-1.50E+38
y	1.50E+38
z	1.0

(x+y)+z	x+(y+z)
0.00E+00	-1.50E+38
1.0	1.50E+38
1.00E+00	0.00E+00

- **Parallelism exacerbates the problem:**
  - **Order of operations may vary due in unexpected ways**
  - **Parallel programs must be validated at varying degrees of parallelism**

IEEE Floating Point - Special Properties

- Floating Point 0 same as Integer 0
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - A > B if:
    - A.EXP > B.EXP <u>or</u>
    - A.EXP=B.EXP <u>and</u> A.SIG > B.SIG

This is equivalent to unsigned comparision!

  - But, must first compare sign bits
  - Must consider -0 == 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?

Summary - Chapter 3

- **Important Topics**
  - **Signed & Unsigned Numbers (3.2)**
  - **Addition and Subtraction (3.1)**
  - **Constructing an ALU (C.5)**
  - **Multiplication and Division (3.3, 3.4)**
  - **Floating Point (3.5-3.8)**
- **Coming Up:**
  - **Processor Design! (Chapter 4)**

1. 프로그램 해석 문제

```
func:
        li      $v0, 0                      #V0 = 0
        li      $t0, 0                      #v1 = 0
L1:     add     $t1, $a0, $t0
        lb      $t2, 0($t1)                 #load one byte
        beq     $t2, $zero, L3
        bne     $t2, $a1, L2
        add     $v0, $v0, 1
L2:     add     $t0, $t0, 1
        j       L1
L3:     jr      $ra
```

```
입력:
$a0 = address of string s[];
$a1 = some character c;
```