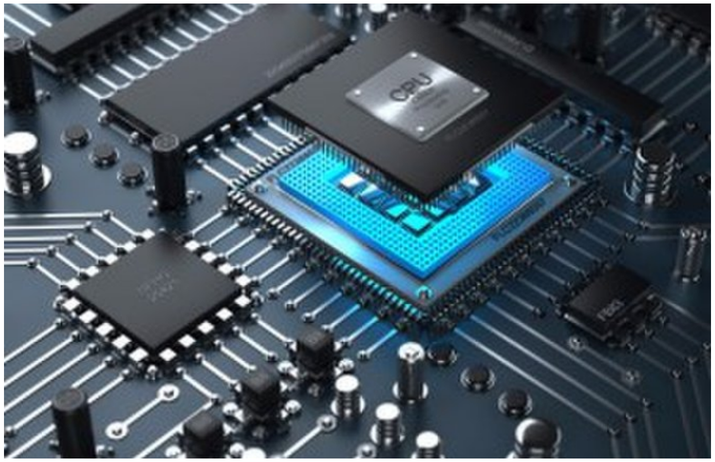


Arithmetic for Computers 2: Floating Point Numbers

CS 154: Computer Architecture

Lecture #9

Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- Lab 4 due today!

- Lab 5 out soon

- Syllabus (Schedule Section) has been updated

Midterm Exam (**Wed. 2/12**)

What's on It?

- Everything we've done so far from start to Monday, 2/10

What Should I Bring?

- Your pencil(s), eraser, MIPS Reference Card (on **1** page)
- You can bring **1** sheet of hand-written notes (turn it in with exam). 2 sides ok.

What Else Should I Do?

- ***IMPORTANT***: Come to the classroom 5-10 minutes EARLY
- **If you are late, I may not let you take the exam**
- ***IMPORTANT***: Use the bathroom before the exam – once inside, you cannot leave
- Random seat assignments
- Bring your UCSB ID

Lecture Outline

- Floating Point Numbers Representations
- IEEE 754 F-P Standard
- Arithmetic in F-P
- Instructions for F-P
- Hardware implementations

Floating Point

- Representation for non-integral numbers
- Including very small and very large numbers

- Usually follows some "normalized" form of scientific notation

Floating Point Numbers in CPUs

We need 3 pieces of information to produce a binary floating point number:

$$\pm N \times 2^{E}$$

The **sign** of the number (positive or negative)	The **mantissa** (aka **significand**) of the number	The **exponent** of the number

Representation in MIPS (**Single Precision**)

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>s</td><td colspan="8">exponent</td><td colspan="23">fraction</td></tr>
<tr><td>1 bit</td><td colspan="8">8 bits</td><td colspan="23">23 bits</td></tr>
</table>

- The actual form is: $(-1)^S \times (1 + \text{Fraction}) \times 2^{\text{Exponent - Bias}}$
  - Called the IEEE 754 F-P Standard (more on this coming up)

- MIPS design for "single-precision" has:

8 bits for exponent and 23 bits for fraction

- Gives a range from 2.0 x 10^{-38} to 2.0 x 10^{38} – quite large!

- **Overflow** can occur: here it means that the exponent is too large to be represented in the exponent field.
- If a *negative* exponent is too large, then we get **underflow**.

Double Precision Floating Points

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent												fraction																		
1 bit	11 bits												20 bits																		

fraction (continued)
32 bits

- Single Precision is **`float`** in C/C++

- Double Precision is **`double`** in C/C++

- 64 bits (2 words) instead of 32 bits
- 11 bits for exponent (instead of 8)
- 52 bits for fraction (instead of 23)

Gives a wider range and greater precision than single-precision

Range is: 2.0×10^{-308} to 2.0×10^{308}

IEEE 754 Floating-Point Standard

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

- Includes single and double-precision definitions (since 1980s)
  - Very widespread in almost all CPUs today

- S = 0 ➔ positive *S = 1* ➔ *negative*

- The "1" in "1 + Fraction" is ***implicit***

$$(1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots)$$

- The "Bias" is **127** for single-precision and **1023** for double-precision

Examples with single-precision:

S = 0, E = 0x82, F = 0 is:

(+1) x (1 + 0) x 2 $^{(130\text{-}127)}$

= 1 x 2^3 = **8**

S = 0, E = 0x83, F = 0x600000 is:

(+1) x (1 + 0.11) x 2 $^{(131\text{-}127)}$

= 1.11 x 2^4 = 11100 = **28**

Useful website: https://www.h-schmidt.net/FloatConverter/IEEE754.html

More Examples!

- Hex word for single-precision F-P is: **0x3FA00000**
- So:

0011 1111 1010 0000 … 0000

S = 0 **E** = 0x7F = 127 **F** = 010…0

- So:

Number $= (+1) \times (1 + 0.01) \times 2^{(127-127)} = 1.01$ (bin)

$= 1 + 1 \times 2^{-2} =$ **1.25**

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$
$2^{-5} = 0.03125$

Yet More Examples!!

- Hex word for single-precision F-P is: **0xBF300000**
- So:

1011 1111 0011 0000 … 0000

S = 1 **E** = 0x7E = 126 **F** = 011…0

- So:

Number $= (-1) \times (1 + 0.011) \times 2^{(126-127)} = 1.011$ (bin)

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

$= -(1 + 0.25 + 0.125) \times 0.5$

= -0.6875

Even More Examples!!!

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$
$2^{-5} = 0.03125$

- What is the single-precision word (in hex) of the F-P number **29.125**?
- Ok, here we go:

  I am reminded that **0.125** = 2^{-3}

  And, I know that **29** in binary is: **11101**

  So $\mathbf{29.125}_{(10)} = \mathbf{11101.001}_{(2)} = \mathbf{1.1101001 \times 2^{4}}$

  This is a positive number, so **S = 0**

  F = 1101001000…0 (23 bits in all)

  E = 4 + 127 = 131 = 10000011

- So:

  Number in bin = 0 10000011 1101001000…0

  or 0100 0001 1110 1001 0…0

  = **0x41E90000**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

Special Exponent Values

Consider Single-Precision Numbers:

- Exponents **0x00** and **0xFF** are reserved

- Smallest exponent is 1 ➔ Actual exponent = 1 - 127 = -126
- Smallest fraction is 0
- So, I get $\pm 1.0 \times 2^{-126} \cong \pm 1.2 \times 10^{-38}$

- Largest exponent is 0xFE = 254 ➔ Actual exp. = 127
- Largest fraction is 111…11 , which approaches 1
- So, I get $\pm 2.0 \times 2^{+127} \cong \pm 3.4 \times 10^{+38}$

Special IEEE 754 Values

- IEEE 754 allows for special symbols to represent "unusual events"

- When **S = 0**, **E = 0xFF**, **F = 0**,

  IEEE calls the number "***inf***" (i.e. infinity)

- "***-inf***" is when **S = 1**, **E = 0xFF**, **F = 0**

- These are to optionally allow programmers to divide by 0.

- Allows for the result of invalid operations

  These are called "Not a Number" or "***NaN***"

  - Example: 0/0 , inf – inf, etc…

Floating-Point Addition

Consider a 4-digit decimal example: **$9.999 \times 10^{1} + 1.610 \times 10^{-1}$**

1. Align decimal points
   - Shift number with smaller exponent
   - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
2. Add significands
   - 10.015×10^{1}
3. Normalize result & check for over/underflow
   - 1.0015×10^{2}
4. Round and renormalize *if necessary* *(what? why? Be patient…)*
   - 1.002×10^{2}

Floating-Point Addition

Consider a 4-digit ***binary*** example: **$1.000 \times 2^{-1} + -1.110 \times 2^{-2}$**

1. Align decimal points
   - Shift number with smaller exponent
   - $1.000 \times 2^{-1} + -0.111 \times 2^{-1}$
2. Add significands
   - 0.001×2^{-1}
3. Normalize result & check for over/underflow
   - 1.000×2^{-4}
4. Round and renormalize *if necessary*
   - $1.000 \times 2^{-4} = 0.0625$

Re: Rounding in Binary F-P

- Can we create ANY floating point number in binary?

- What about 0.3333... (i.e. ***1/3***)?
- In binary, ***1/10*** is the infinitely repeating fraction **0.0001100110011001100110011001100110011001100**...

- Since we cannot create ALL F-P numbers in binary, rounding (i.e. approximating) is necessary

- Many users are not aware of the approximation because of the way values are displayed
  - The actual stored value is the nearest representable binary fraction

C++ Program to Illustrate Rounding in Binary F-P

```
#include <iostream>
#include <iomanip>
int main()
{
    // Try running the program without the next 2 lines
    // as a comparison. Or change the precision number around.
    std::cout << std::setprecision(30);
    std::cout << std::fixed;

    float a = 1.0/3;
    double b = 1.0/3;
    std::cout << a << "\n" << b << "\n";

    float x = 1.0/10;
    double y = 1.0/10;
    std::cout << x << "\n" << y;
}
```

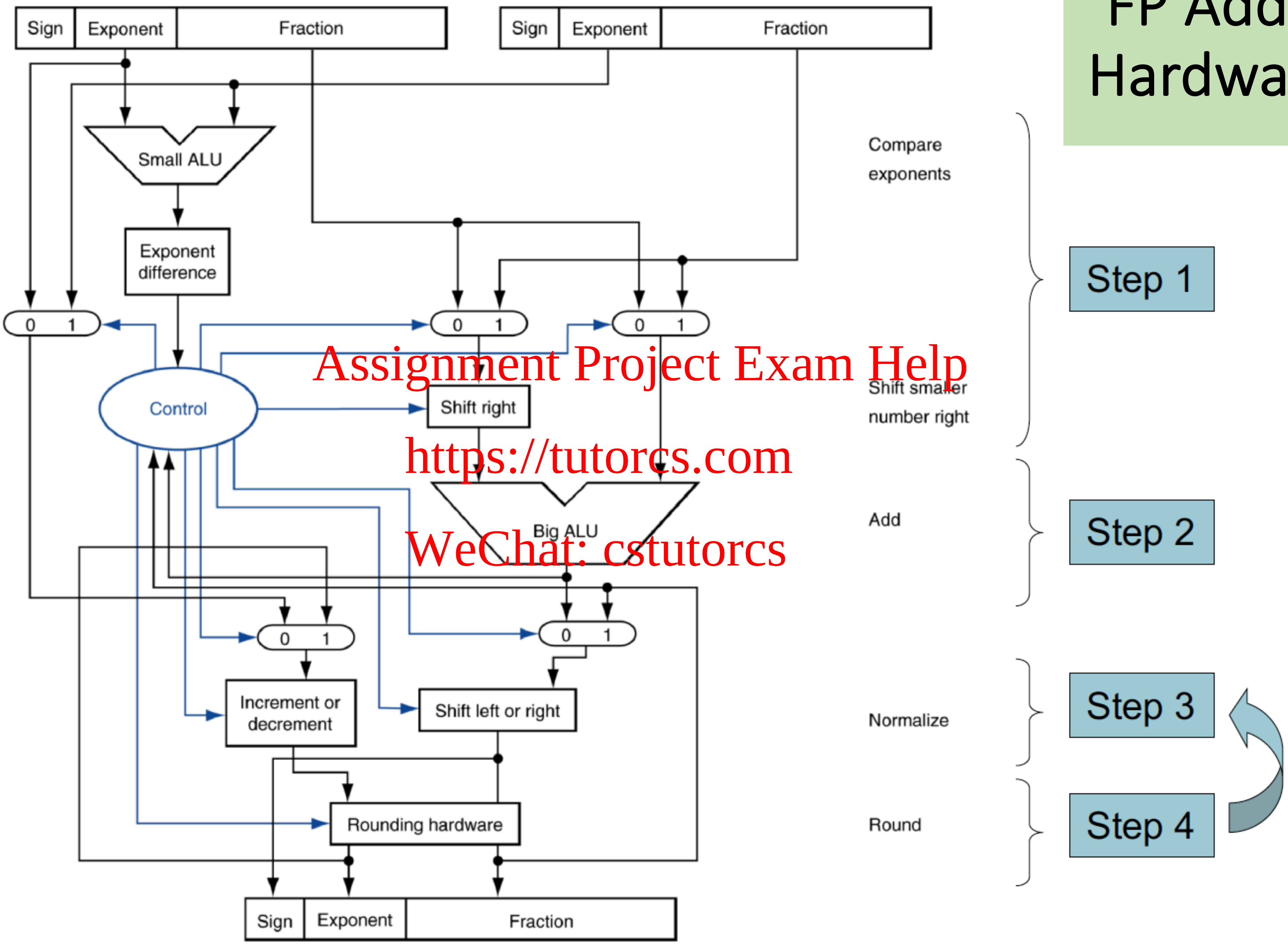
Floating-Point Adder Hardware

- Much more complex than integer adder
  - Remember the 4 steps from a couple of slides ago?...

- Doing it in one clock cycle would take too long
  - Would force a slower clock on the system
  - How much we can do in 1 clock cycle is a matter for later discussion

- FP adder usually takes several cycles
  - Can be pipelined for more efficient operation

FP Adder Hardware



FP Other Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder

- FP arithmetic hardware (incl. addition) is usually in a ***co-processor*** & does:
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ←→ integer conversion

- Operations usually takes several cycles
  - Can be pipelined

MIPS FP Instructions

	Single-Precision	*Double-Precision*
Addition	`add.s`	`add.d`
Subtraction	`sub.s`	`sub.d`
Multiplication	`mul.s`	`mul.d`
Division	`div.s`	`div.d`
Comparisons	`c.`*`xx`*`.s`	`c.`*`xx`*`.d`
Where *xx* can be	eq, neq, lt, gt,	le, ge
Example: **`c.eq.s`**		
Load	`lwc1`	`lwd1`
Store	`swc1`	`swd1`

Also, F-P branch, true (**`bc1t`**) and branch, false (**`bc1f`**)

MIPS FP Instructions

- FP instructions operate only on FP registers
- Programs generally don't do integer ops on FP data, or vice versa
- More registers with minimal code-size impact

The Floating Point Registers

- MIPS has 32 ***separate*** registers for floating point:
  - **`$f0`**, **`$f1`**, etc…

- Paired for double-precision
  - **`$f0/$f1`**, **`$f2/$f3`**, etc…

- Example MIPS assembly code:

```
lwc1 $f4, 0($sp)     # Load 32b F.P. number into F4
lwc1 $f6, 4($sp)     # Load 32b F.P. number into F6
add.s $f2, $f4, $f6  # F2 = F4 + F6 single precision
swc1 $f2, 8($sp)     # Store 32b F.P. number from F2
```

Example Code

C++ code:

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


Assume:

fahr in **$f12**, **result** in **$f0**, constants in global memory space (i.e. defined in **.data**)

Compiled MIPS code:


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

YOUR TO-DOs for the Week

- Readings!

- Work on Lab 5!

- Start studying for the midterm!

</LECTURE>