

CS61C : Machine Structures

Lecture #12 – Floating Point II, MIPS Instruction Format III

2008-7-10

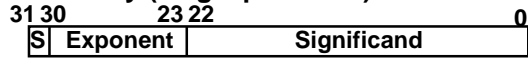


Albert Chae, Instructor

Review

- Floating Point numbers approximate values that we want to use.

- Summary (single precision):



1 bit 8 bits 23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Why do we use bias?

- Double precision same idea with more bits

- 11 bit exponent, bias of 1023
- 52 bit significand



Peer Instruction from yesterday

- There is no interval of numbers where the gap between a floating point representation and the next biggest is 1. **False**

???? ???? 000 0000 0000 0000 0000 0000
--

???? ???? 000 0000 0000 0000 0000 0001
--



Converting Decimal to FP (1/4)

- Simple Case: If denominator is an exponent of 2 (2, 4, 8, 16, etc.), then it's easy.

- Show MIPS representation of -0.75

• $-0.75 = -3/4$

• $-11_{\text{two}}/100_{\text{two}} = -0.11_{\text{two}}$

• Normalized to $-1.1_{\text{two}} \times 2^{-1}$

• $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

• $(-1)^1 \times (1 + .100\ 0000 \dots 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000
---	-----------	------------------------------



Converting Decimal to FP (2/4)

- 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 neverending number?



Converting Decimal to FP (3/4)

- 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.



Example: Representing 1/3 in MIPS

- $1/3$
 - = $0.33333..._{10}$
 - = $0.25 + 0.0625 + 0.015625 + 0.00390625 + \dots$
 - = $1/4 + 1/16 + 1/64 + 1/256 + \dots$
 - = $2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \dots$
 - = $0.0101010101..._2 \times 2^0$
 - = $1.0101010101..._2 \times 2^{-2}$
- Sign: 0
- Exponent = $-2 + 127 = 125 = 01111101$
- Significand = $0101010101...$

Cal

0	0111 1101	0101 0101 0101 0101 0101 0101
---	-----------	-------------------------------

 CS61C L12 Floating Point II, MIPS Instruction Format III (7) Chae, Summer 2008 © UCB

Converting Decimal to FP (4/4)

- -2.340625×10^1
1. Denormalize: -23.40625
 2. Convert integer part:
 $23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$
 3. Convert fractional part:
 $.40625 = .25 + (.15625 = .125 + (.03125)) = .01101_2$
 4. Put parts together and normalize:
 $10111.01101 = 1.011101101 \times 2^4$
 5. Convert exponent: $127 + 4 = 1000011_2$

Cal

1	1000 0011	011 1011 0100 0000 0000 0000
---	-----------	------------------------------

 CS61C L12 Floating Point II, MIPS Instruction Format III (8) Chae, Summer 2008 © UCB

Understanding the Significand (1/2)

- Method 1 (Fractions):
 - In decimal: $0.340_{10} \Rightarrow 340_{10}/1000_{10} \Rightarrow 34_{10}/100_{10}$
 - In binary: $0.110_2 \Rightarrow 110_2/1000_2 = 6_{10}/8_{10} \Rightarrow 11_2/100_2 = 3_{10}/4_{10}$
 - Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

Cal CS61C L12 Floating Point II, MIPS Instruction Format III (9) Chae, Summer 2008 © UCB

Understanding the Significand (2/2)

- Method 2 (Place Values):
 - Convert from scientific notation
 - In decimal: $1.6732 = (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
 - In binary: $1.1001 = (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
 - 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

Cal CS61C L12 Floating Point II, MIPS Instruction Format III (10) Chae, Summer 2008 © UCB

Peer Instruction

1	1000 0001	111 0000 0000 0000 0000 0000
---	-----------	------------------------------

What is the decimal equivalent of the floating pt # above?

- | | |
|----|----------------|
| 1: | -3.5 |
| 2: | -7 |
| 3: | -7.5 |
| 4: | $-7 * 2^{129}$ |
| 5: | $-129 * 2^7$ |

Cal CS61C L12 Floating Point II, MIPS Instruction Format III (11) Chae, Summer 2008 © UCB

Peer Instruction Answer

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
---	-----------	------------------------------

S Exponent Significand

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

$(-1)^1 \times (1 + .111) \times 2^{(129-127)}$

$-1 \times (1.111) \times 2^{(2)}$

-111.1

-7.5_{ten}

- | | |
|----|----------------|
| 1: | -3.5 |
| 2: | -7 |
| 3: | -7.5 |
| 4: | $-7 * 2^{129}$ |
| 5: | $-129 * 2^7$ |

Cal CS61C L12 Floating Point II, MIPS Instruction Format III (12) Chae, Summer 2008 © UCB

Quote of yesterday

“95% of the folks out there are completely clueless about floating-point.”

James Gosling
Sun Fellow
Java Inventor
1998-02-28



Precision and Accuracy

Don't confuse these two terms!

Precision is a count of the number bits in a computer word used to represent a value.

Accuracy is a measure of the difference between the actual value of a number and its computer representation.

High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.

Example: `float pi = 3.14;`

pi will be represented using all 24 bits of the significant (highly precise), but is only an approximation (not accurate).

pi is actually some infinitely non-repeating number (transcendental), not 3.14



Representation for 0

• Represent 0?

- exponent all zeroes
- significand all zeroes too
- What about sign?

• +0: 0 00000000 000000000000000000000000
• -0: 1 00000000 000000000000000000000000

• Why two zeroes?

- Helps in some limit comparisons
- Ask math majors



Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.

• Why?

- OK to do further computations with ∞
E.g., $X/0 > Y$ may be a valid comparison

- Ask math majors

• IEEE 754 represents $\pm \infty$

- Most positive exponent reserved for ∞
- Significands all zeroes



Special Numbers

• What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	<u>nonzero</u>	???

• Professor Kahan had clever ideas; “Waste not, want not”

- Exp=0,255 & Sig!=0 ...



Representation for Not a Number

• What is `sqrt(-4.0)` or `0/0`?

- If ∞ not an error, these shouldn't be either.
- Called **Not a Number (NaN)**
- Exponent = 255, Significand nonzero

• Why is this useful?

- Hope NaNs help with debugging?
- They contaminate: `op(NaN, X) = NaN`



Representation for Denorms (1/2)

- **Problem:** There's a gap among representable FP numbers around 0

- **Smallest representable pos num:**

$$a = 1.0 \dots_2 * 2^{-126} = 2^{-126}$$

- **Second smallest representable pos num:**

$$b = 1.000 \dots_2 * 2^{-126} = 2^{-126} + 2^{-149}$$

$$a - 0 = 2^{-126}$$

$$b - a = 2^{-149}$$

Normalization
and implicit 1
is to blame!



Representation for Denorms (2/2)

- **Solution:**

- We still haven't used Exponent = 0, Significand nonzero

- Denormalized number: no leading 1, **implicit exponent = -126.**

- **Smallest representable pos num:**

$$a = 2^{-149}$$

- **Second smallest representable pos num:**

$$b = 2^{-148}$$



Overview

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>Denorm</u>
1-254	anything	+/- fl. pt. #
255	<u>0</u>	<u>+/- ∞</u>
255	<u>nonzero</u>	<u>NaN</u>



Peer Instruction

- Let $f(1, 2) = \#$ of floats between 1 and 2
- Let $f(2, 3) = \#$ of floats between 2 and 3

1: $f(1, 2) < f(2, 3)$
 2: $f(1, 2) = f(2, 3)$
 3: $f(1, 2) > f(2, 3)$



Peer Instruction Answer

- Let $f(1, 2) = \#$ of floats between 1 and 2
- Let $f(2, 3) = \#$ of floats between 2 and 3

1: $f(1, 2) < f(2, 3)$
 2: $f(1, 2) = f(2, 3)$
 3: $f(1, 2) > f(2, 3)$



Administrivia

- **Assignments**

- **Proj1** due 7/11 @ 11:59pm (preliminary ag)
 - Extra OH today: 1:30-4pm... maybe longer?
- **Quiz 5/6** due 7/14 @ 11:59pm
- **HW3** due 7/15 @ 11:59pm
- **Proj2** due 7/18 @ 11:59pm



Administrivia...Midterm

- **Midterm Mon 2008-07-21 @7-10pm, 155 Dwinelle**
 - Conflicts/DSP? Email me
 - You can bring green sheet and one handwritten double sided note sheet
- **How should we study for the midterm?**
 - Form study groups...don't prepare in isolation!
 - Attend the
 - faux midterm: 7/16 @ 6-9pm 10 Evans
 - review session: 7/17 in lecture (will go over faux exam)
 - Write up your handwritten 1-page study sheet
 - Go over old exams (except the one we use for faux)- HKN office has put them online (link on website)
- Attend TA office hours and work out hard probs



CS61C L12 Floating Point II, MIPS Instruction Format III (25)

Chae, Summer 2008 ©UCB

Rounding

- Math on real numbers \Rightarrow we worry about rounding to fit result in the significant field.
- FP hardware carries 2 extra bits of precision, and rounds for proper value
- Rounding occurs when...
 - converting double to single precision
 - converting floating point # to an integer
 - Intermediate step if necessary



CS61C L12 Floating Point II, MIPS Instruction Format III (26)

Chae, Summer 2008 ©UCB

IEEE Four Rounding Modes

- Round towards $+\infty$
 - ALWAYS round "up": $2.1 \Rightarrow 3$, $-2.1 \Rightarrow -2$
- Round towards $-\infty$
 - ALWAYS round "down": $1.9 \Rightarrow 1$, $-1.9 \Rightarrow -2$
- Round towards 0 (i.e., truncate)
 - Just drop the last bits
- Round to (nearest) even (default)
 - Normal rounding, almost: $2.5 \Rightarrow 2$, $3.5 \Rightarrow 4$
 - Like you learned in grade school (almost)
 - Insures fairness on calculation
 - Half the time we round up, other half down
 - Also called Unbiased



CS61C L12 Floating Point II, MIPS Instruction Format III (27)

Chae, Summer 2008 ©UCB

Rounding in binary

- Steps to Use it (in binary)
 - Determine place to be rounded to
 - Figure out the two possible outcomes (its binary so 1 or 0 in last place)
 - If one outcome is closer to current number than other, pick that outcome
 - If both outcomes are equidistant pick the outcome that ends in 0



CS61C L12 Floating Point II, MIPS Instruction Format III (28)

Chae, Summer 2008 ©UCB

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand 1000 8
Multiplier   x1001 9
-----
           1000
            0000
             0000
+1000
-----
          01001000
```

- m bits \times n bits = $m + n$ bit product



CS61C L12 Floating Point II, MIPS Instruction Format III (29)

Chae, Summer 2008 ©UCB

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
 - 32-bit value \times 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
 - `mult register1, register2`
 - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
 - puts product **upper half in hi**, **lower half in lo**
 - **hi** and **lo** are 2 registers separate from the 32 general purpose registers
 - Use **mfhi** register & **mflo** register to **move from hi, lo** to another register



CS61C L12 Floating Point II, MIPS Instruction Format III (30)

Chae, Summer 2008 ©UCB

Integer Multiplication (3/3)

- Example:

- in C: $a = b * c$;

- in MIPS:

- let b be \$s2; let c be \$s3; and let a be \$s0 and \$s1 (since it may be up to 64 bits)

```
mult $s2,$s3 # b*c
mfhi $s0     # upper half
of          # product
into $s0
mflo $s1     # lower half of
            # product into $s1
```

- Note: Often, we only care about the lower half of the product.



Integer Division (1/2)

- Paper and pencil example (unsigned):

Divisor	1000		1001	Quotient
			1010	Dividend
			-1000	
			10	
			101	
			1010	
			-1000	
			10	Remainder
			(or Modulo result)	

- Dividend = Quotient x Divisor + Remainder



Integer Division (2/2)

- Syntax of Division (signed):

- div register1, register2

- Divides 32-bit register 1 by 32-bit register 2:

- puts remainder of division in hi, quotient in lo

- Implements C division (/) and modulo (%)

- Example in C: $a = c / d$;
 $b = c \% d$;

- in MIPS: $a \leftrightarrow \$s0$; $b \leftrightarrow \$s1$; $c \leftrightarrow \$s2$; $d \leftrightarrow \$s3$

```
div $s2,$s3 # lo=c/d, hi=c%d
mflo $s0    # get quotient
mfhi $s1    # get remainder
```



Unsigned Instructions & Overflow

- MIPS also has versions of mult, div for unsigned operands:

- multu

- divu

- Determines whether or not the product and quotient are changed if the operands are signed or unsigned.

- MIPS does not check overflow on ANY signed/unsigned multiply, divide instr

- Up to the software to check hi



FP Addition & Subtraction

- Much more difficult than with integers (can't just add significands)

- How do we do it?

- De-normalize to match larger exponent
 - Add significands to get resulting one
 - Normalize (& check for under/overflow)
 - Round if needed (may need to renormalize)

- If signs \neq , do a subtract. (Subtract similar)

- If signs \neq for add (or = for sub), what's ans sign?

- Question: How do we integrate this into the integer arithmetic unit? [Answer: We don't!]



MIPS Floating Point Architecture (1/4)

- Separate floating point instructions:

- Single

- Precision:

- add.s, sub.s, mul.s, div.s

- Double

- Precision:

- add.d, sub.d, mul.d, div.d

- These are far more complicated than their integer counterparts

- Can take much longer to execute



MIPS Floating Point Architecture (2/4)

- **Problems:**
 - Inefficient to have different instructions take vastly differing amounts of time.
 - Generally, a particular piece of data will not change FP \Leftrightarrow 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 (3/4)

- **1990 Solution: Make a completely separate chip that handles only FP.**
- **Coprocessor 1: FP chip**
 - contains 32 32-bit registers: $\$f0, \$f1, \dots$
 - most of the registers specified in `.s` and `.d` instruction refer to this set
 - separate load and store: `lwc1` and `swc1` (“load word coprocessor 1”, “store ...”)
 - **Double Precision: by convention, even/odd pair contain one DP FP number:**
 $\$f0/\$f1, \$f2/\$f3, \dots, \$f30/\$f31$
 - **Even register** is the name



MIPS Floating Point Architecture (4/4)

- **1990 Computer actually contains multiple separate chips:**
 - Processor: handles all the normal stuff
 - Coprocessor 1: handles FP and only FP;
 - more coprocessors?... Yes, later
 - Today, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW
- **Instructions to move data between main processor and coprocessors:**
 - `mfc0, mtc0, mfc1, mtc1`, etc.
- **Appendix contains many more FP ops**



Peer Instruction

True or False

1. Converting float \rightarrow int \rightarrow float produces same float number
2. Converting int \rightarrow float \rightarrow int produces same int number
3. FP add is associative:
 $(x+y)+z = x+(y+z)$



Peer Instruction Answer



To conclude on floating point

- **Reserve exponents, significands:**

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>Denorm</u>
1-254	anything	+/- fl. pt. #
255	<u>0</u>	<u>+/- ∞</u>
255	<u>nonzero</u>	<u>NaN</u>
- Integer `mult, div` uses `hi, lo` regs
 - `mghi` and `mflo` copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive



Decoding Example (6/7)

• MIPS Assembly (Part 2):

```
Loop:    or    $v0,$0,$0
        slt   $t0,$0,$a1
        beq   $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```

• Next step: translate to C code (be creative!)



Decoding Example (7/7)

Before Hex: • After C code (Mapping below)

```
00001025hex    $v0: product
0005402Ahex    $a0: multiplicand
11000003hex    $a1: multiplier
00441020hex    product = 0;
20A5FFFFhex    while (multiplier > 0) {
08100001hex    product += multiplicand;
                multiplier -= 1;
                }
```

```
Loop:    or    $v0,$0,$0
        slt   $t0,$0,$a1
        beq   $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```

Demonstrated Big 61C Idea: Instructions are just numbers, code is treated like data

Exit:

Review from before: lui

• So how does lui help us?

• Example:

```
addi    $t0,$t0, 0xABABCDCD
```

becomes:

```
lui     $at, 0xABAB
ori     $at, $at, 0xCDCD
add     $t0,$t0,$at
```

• Now each I-format instruction has only a 16-bit immediate.

• Wouldn't it be nice if the assembler would this for us automatically?

- If number too big, then just automatically replace addi with lui, ori, add



True Assembly Language (1/3)

• Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions

• What happens with pseudoinstructions?

- They're broken up by the assembler into several "real" MIPS instructions.
- But what is a "real" MIPS instruction? Answer in a few slides

• First some examples



Example Pseudoinstructions

• Register Move

```
move    reg2,reg1
```

Expands to:

```
add     reg2,$zero,reg1
```

• Load Immediate

```
li      reg,value
```

If value fits in 16 bits:

```
addi    reg,$zero,value
```

else:

```
lui     reg,upper 16 bits of value
```

```
ori     reg,$zero,lower 16 bits
```



True Assembly Language (2/3)

• Problem:

- When breaking up a pseudoinstruction, the assembler may need to use an extra reg.
- If it uses any regular register, it'll overwrite whatever the program has put into it.

• Solution:

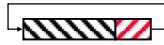
- Reserve a register (\$1, called \$at for "assembler temporary") that assembler will use to break up pseudo-instructions.
- Since the assembler may use this at any time, it's not safe to code with it.



Example Pseudoinstructions

- Rotate Right Instruction

```
ror reg, value
```

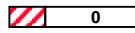


Expands to:

```
srl $at, reg, value
```



```
sll reg, reg, 32-value
```



```
or reg, reg, $at
```



- “No Operation” instruction

```
nop
```

Expands to instruction = 0_{ten},

```
sll $0, $0, 0
```



Example Pseudoinstructions

- Wrong operation for operand

```
addu reg, reg, value # should be addiu
```

If value fits in 16 bits, addu is changed to:

```
addiu reg, reg, value
```

else:

```
lui $at, upper 16 bits of value
```

```
ori $at, $at, lower 16 bits
```

```
addu reg, reg, $at
```

- How do we avoid confusion about whether we are talking about MIPS assembler with or without pseudoinstructions?



True Assembly Language (3/3)

- **MAL** (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions

- **TAL** (True Assembly Language): set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)

- A program must be converted from MAL into TAL before translation into 1s & 0s.



Questions on Pseudoinstructions

- Question:

- How does MIPS recognize pseudo-instructions?

- Answer:

- It looks for officially defined pseudo-instructions, such as **ror** and **move**

- It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully



Rewrite TAL as MAL

- TAL:

```

Loop:   or    $v0,$0,$0
        slt  $t0,$0,$a1
        beq  $t0,$0,Exit
        add  $v0,$v0,$a0
        addi $a1,$a1,-1
        j    Loop
Exit:

```

- This time convert to MAL

- It's OK for this exercise to make up MAL instructions



Rewrite TAL as MAL (Answer)

- TAL:

```

Loop:   or    $v0,$0,$0
        slt  $t0,$0,$a1
        beq  $t0,$0,Exit
        add  $v0,$v0,$a0
        addi $a1,$a1,-1
        j    Loop
Exit:

```

- MAL:

```

Loop:   li    $v0,0
        bge  $zero,$a1,Exit
        add  $v0,$v0,$a0
        decr $a1, 1
        j    Loop
Exit:

```



Peer Instruction

True or False

Which of the instructions below are **MAL** and which are **TAL**?

- A. `addi $t0, $t1, 40000`
- B. `beq $s0, 10, Exit`
- C. `sub $t0, $t1, 1`



In conclusion

- **Disassembly is simple and starts by decoding `opcode` field.**
 - Be creative, efficient when authoring C
- **Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)**
 - Only TAL can be converted to raw binary
 - Assembler's job to do conversion
 - Assembler uses reserved register `$at`
 - MAL makes it much easier to write MIPS

