inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

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

2008-7-10

Albert Chae, Instructor

Review

- Floating Point numbers <u>approximate</u> values that we want to use.
- Summary (single precision):

31	30 … 23	22 … 0
S	Exponent	Significand
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
  - $-11_{two}/100_{two} = -0.11_{two}$
  - Normalized to $-1.1_{two} \times 2^{-1}$
  - $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - $(-1)^1 \times (1 + .100\ 0000 \ldots 0000) \times 2^{(126-127)}$

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

  $= 0.33333\ldots_{10}$

  $= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \ldots$

  $= 1/4 + 1/16 + 1/64 + 1/256 + \ldots$

  $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots$

  $= 0.0101010101\ldots_2 * 2^0$

  $= 1.0101010101\ldots_2 * 2^{-2}$

  - Sign: 0
  - Exponent = -2 + 127 = 125 = 01111101
  - Significand = 0101010101…

0	0111 1101	0101 0101 0101 0101 0101 010

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: $127 + 4 = 10000011_2$

1	1000 0011	011 1011 0100 0000 0000 0000

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**

Understanding the Significand (2/2)

- **Method 2 (Place Values):**
  - **Convert from scientific notation**
  - **In decimal: $1.6732 = (1x10^0) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$**
  - **In binary: $1.1001 = (1x2^0) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$**
  - **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**

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
```


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
```

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 00000000000000000000000`
  - `-0: 1 00000000 00000000000000000000000`
- Why two zeroes?
  - Helps in some limit comparisons
  - Ask math majors

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, 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 ± ∞
  - 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	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	???

- 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:

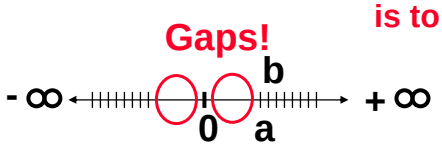
    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:

    $b = 1.000\ldots\ldots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

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

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

Normalization and implicit 1 is to blame!

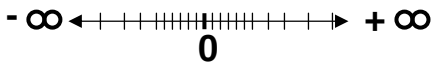


Representation for Denorms (2/2)

- 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	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

Peer Instruction

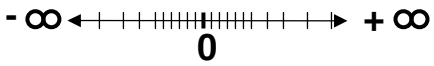
- Let `f(1,2)` = # of floats between 1 and 2
- Let `f(2,3)` = # of floats between 2 and 3

```
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

Rounding

- Math on real numbers ⇒ 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

IEEE Four Rounding Modes

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

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

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

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

  - m bits x n bits = m + n bit product

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 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

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):

```
                 1001    Quotient
Divisor 1000|1001010    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↔$s0;b↔$s1;c↔$s2;d↔$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 ≠, do a subtract. (Subtract similar)
  - If signs ≠ 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 <u>far more complicated</u> 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 <u>particular piece of data will not change FP ⇔ `int`</u> within a program.
    - Only 1 type of instruction will be used on it.
  - Some programs <u>do no FP calculations</u>
  - 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`, …
  - 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`, … , `$f30`/`$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` -> `int` -> `float` produces same `float` number
2. Converting `int` -> `float` -> `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	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

- Integer `mult`, `div` uses `hi`, `lo` regs
  - `mfhi` and `mflo` copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive

Decoding Machine Language

- How do we convert 1s and 0s to C code?

  Machine language ⇒ C?

- For each 32 bits:
  - Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.
  - Use instruction type to determine which fields exist.
  - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  - Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding Example (1/7)

- Here are six machine language instructions in hexadecimal:

  00001025_{hex}
  $0005402A_{hex}$
  11000003_{hex}
  00441020_{hex}
  $20A5FFFF_{hex}$
  08100001_{hex}

- Let the first instruction be at address $4{,}194{,}304_{ten}$ ($0x00400000_{hex}$).
- Next step: convert hex to binary

Decoding Example (2/7)

- The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

- Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4-31	rs	rt	immediate		
J	2 or 3	target address				

Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:

```
R  000000 00000000000001000000100101
R  000000 00000001010100000000101010
I  000100 01000000000000000000000011
R  000000 00010001000001000000100000
I  001000 00101001011111111111111111
J  000010 00000100000000000000000001
```

- Look at opcode:
  0 means R-Format,
  2 or 3 mean J-Format,
  otherwise I-Format.

- Next step: separation of fields

Decoding Example (4/7)

- Fields separated based on format/opcode:

Format:						
R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

- Next step: translate ("disassemble") to MIPS assembly instructions

Decoding Example (5/7)

- MIPS Assembly (Part 1):

Address:	Assembly instructions:	
0x00400000	or	$2,$0,$0
0x00400004	slt	$8,$0,$5
0x00400008	beq	$8,$0,3
0x0040000c	add	$2,$2,$4
0x00400010	addi	$5,$5,-1
0x00400014	j	0x100001

- Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

Decoding Example (6/7)

- MIPS Assembly (Part 2):

```
        or    $v0,$0,$0
Loop:   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:

00001025_{hex}
$0005402A_{hex}$
11000003_{hex}
00441020_{hex}
$20A5FFFF_{hex}$
08100001_{hex}

- After C code (Mapping below)

  $v0: product
  $a0: multiplicand
  $a1: multiplier

```
product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}
```

```
      or    $v0,$0,$0
Loop: 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

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
```

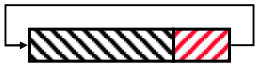

True Assembly Language (2/3)

- **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:

```
            or    $v0,$0,$0
Loop:       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:

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

- MAL:

```
            li    $v0,0
Loop:       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 <u>much</u> easier to write MIPS