

`inst.eecs.berkeley.edu/~cs61c/su05`

CS61C : Machine Structures

Lecture #11: FP II & Pseudo Instructions



2005-07-07

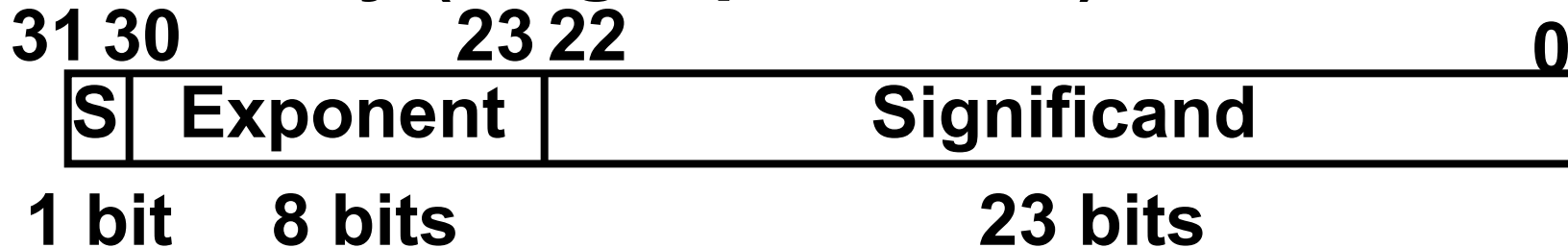
Andy Carle



FP Review

- Floating Point numbers approximate values that we want to use.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers
 - Every desktop or server computer sold since ~1997 follows these conventions

- Summary (single precision):



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

- Double precision identical, bias of 1023



Representation for Denorms (1/3)

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

- **Smallest representable pos num:**

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

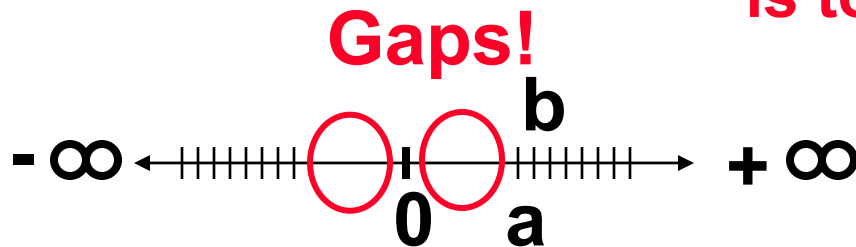
- **Second smallest representable pos num:**

$$b = 1.000\dots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$$

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

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

**Normalization
and implicit 1
is to blame!**



Representation for Denorms (2/3)

- **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:**

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



Representation for Denorms (3/3)

- Normal FP equation:
 - $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
- If (fp.exp == 0 and fp.significant != 0)
 - Denorm
 - $(-1)^S \times (0 + \text{Significand}) \times 2^{(-126)}$



IEEE Four Rounding Modes

- **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**
 - **floating point # to an integer**



IEEE Four Rounding Modes

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



Clarification - IEEE Four Rounding Modes

- This is just an example in base 10 to show you the 4 modes.
- What really happens is...
 - 1) in **binary**, not decimal!
 - 2) at the lowest bit of the mantissa with the **guard bit(s)** as our extra bit(s), and you need to decide how these extra bit(s) affect the result if the guard bits are “**100...**”
 - 3) If so, you're half-way between the representable numbers.
E.g., 0.1010 is $5/8$, halfway between our representable $4/8$ [$1/2$] and $6/8$ [$3/4$]. Which number do we round to? 4 modes!



Integer Multiplication (1/3)

- Paper and pencil example (**unsigned**):

Multiplicand	1000	8
Multiplier	<u>x1001</u>	9
	1000	
	0000	
	0000	
	+1000	
	<u>01001000</u>	

- m bits \times 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**):

$$\begin{array}{r} \text{Divisor } 1000 \overline{) 1001010} \\ \underline{-1000} \\ 10 \\ \underline{101} \\ \underline{1010} \\ \underline{-1000} \\ 10 \end{array} \begin{array}{l} \text{Quotient} \\ \text{Dividend} \\ \\ \\ \\ \\ \text{Remainder} \\ \text{(or Modulo result)} \end{array}$$

- $\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$



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 pages A-70 to A-74 contain many, many more FP operations.**



FP/Math Summary

- 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
 - `mfhi` and `mflo` copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive



Administrivia

- **Midterm TOMORROW!!!11!one!**
 - **11:00 – 2:00**
 - **277 Cory**
 - **You may bring with you:**
 - **The green sheet from COD or a photocopy thereof**
 - **One 8 1/2" x 11" note sheet with *handwritten* notes on *one side***
 - **No books, calculators, other shenanigans**
 - **Conflicts, DSP, other issues:**
 - **let me know ASAP**



Project 1 is due Sunday night

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 do 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:**

```
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:**

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

Exit:
```



Peer Instruction

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

- **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**

