CSE 431
Computer Architecture
Fall 2015

Chapter 3: Arithmetic for Computers

Mary Jane Irwin (www.cse.psu.edu/~mji)

[Adapted from *Computer Organization and Design, 5th Edition*, Patterson & Hennessy, © 2014, MK]

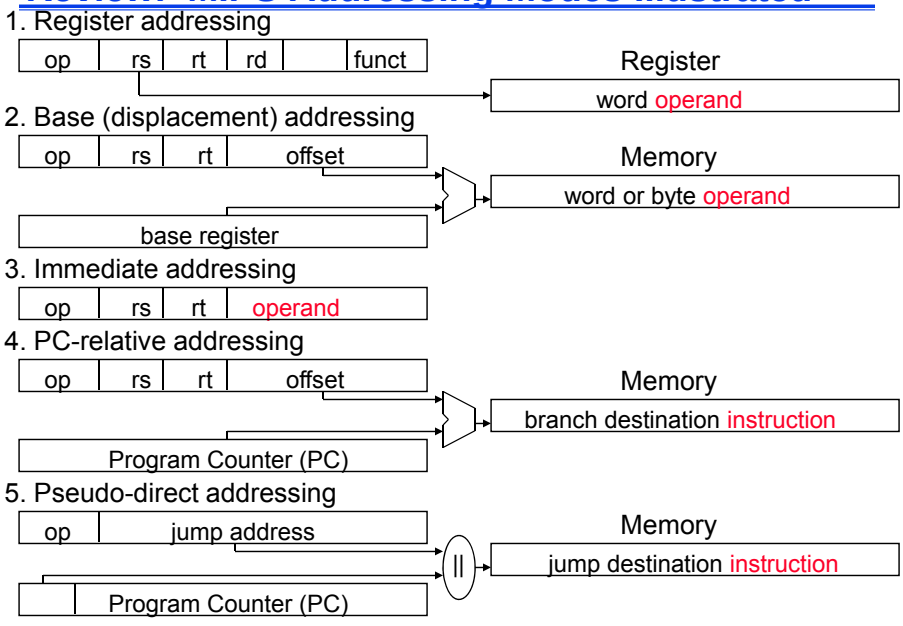
Reminders

- This lecture
  - MIPS arithmetic, PH Chapter 3, Appendix B
- Next week
  - MIPS single-cycle datapath and control, PH 4.1-4.4
  - MIPS pipelined datapaths, data hazards, PH 4.5-4.7
- Reminders
  - HW1 due midnight Sept 3rd in Angel DropBox
  - Quiz 1 will close midnight Sept 7th
  - Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9th and 10th from 7:30 to 9pm
  - HW2 will come out Sept 4th (our first using SimpleScalar)
  - First evening midterm exam scheduled
    - Tuesday, October 6th, 20:15 to 22:15, Location 22 Deike
    - Please let me know ASAP (via email) if you have a conflict

Review: MIPS (RISC) Design Principles

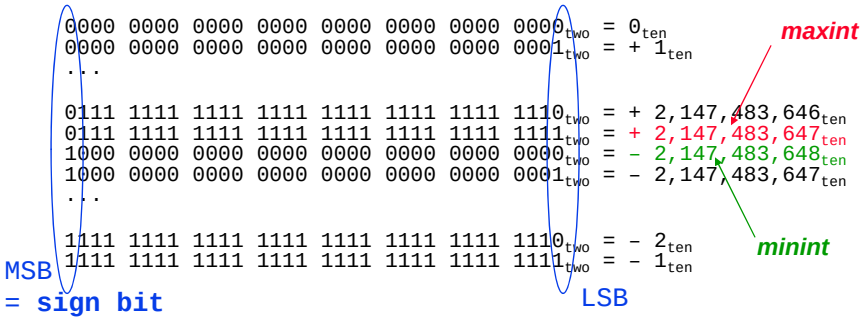
- Simplicity favors regularity
  - Makes implementation simpler, enables higher performance at lower design cost
    - fixed size instructions, small number of instruction formats, opcode always the first 6 bits, etc.
- Smaller is faster/better
  - Reduces design and implementation costs, improves chip yield, enables higher performance (and lower power?)
    - limited instruction set, limited number of registers in RF, limited number of addressing modes, etc.
- Make the common case fast
  - Find the biggest impact on performance and optimize that
    - arithmetic operands from the RF (load-store machine), immediate operands in instructions, etc.
- Good design demands good compromises
    - three instruction formats

Review: MIPS Addressing Modes Illustrated



Number Representations

- 32-bit signed integers (2's complement):



- Converting <32-bit values into 32-bit values
  - copy the most significant bit (the sign bit) into the "empty" bits

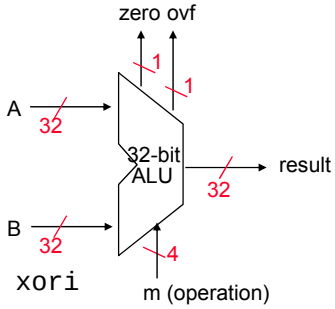
```
0010   -> 0000 0010
1010   -> 1111 1010
```

  - sign extend versus zero extend (`lb` vs. `lbu`)

MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

```
add, addi, addiu, addu
sub, subu
mult, multu, div, divu
sqrt
and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```



- With special handling for
  - sign extend – `addi, addiu, slti, sltiu`
  - zero extend – `andi, ori, xori`
  - overflow detection – `add, addi, sub`
  - overflow detection in software – `mult, div`

Detecting Overflow

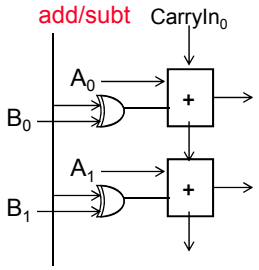
- Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a value bit of the result and not the proper sign bit
  - When adding operands with different signs or when subtracting operands with the same sign, overflow can *never* occur

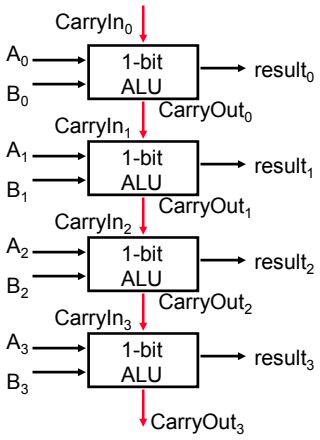
Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

- MIPS signals overflow with an exception (aka interrupt) – an unscheduled procedure call to the OS where the EPC contains the address of the instruction that caused the exception

Addition and Subtraction on the MIPS ALU

- For addition, build a 32-bit adder from 32 1-bit adders. This is called a ripple-carry adder (RCA). Its slow, but easy to understand. How about subtraction ?
  - Remember the "rules" for converting a 2s complement number to its negative equivalent?

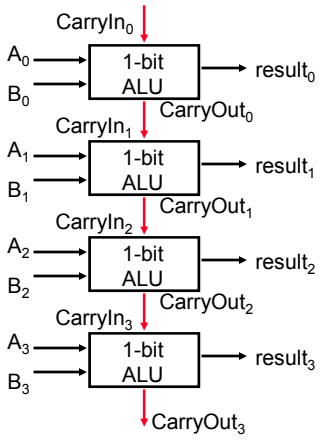




- In reality, just negating and then adding !

But What about Performance?

- The critical path of n-bit RCA is n*CP = O(n)



n	RCA	CLA
8		
16		
32		
64		
128		

- Design trick – throw hardware at it (Carry Lookahead Adder) for an O(log n) adder

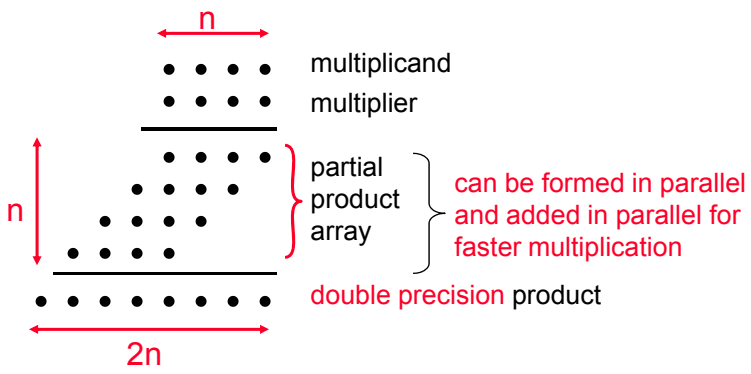
Arithmetic for Multimedia Operations

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use our 32-bit adder, with a "partitioned " carry chain
    - Operate on 4×8-bit, 2×16-bit, or 1×32-bit vectors
  - SIMD (single-instruction, multiple-data), data-level parallelism

- Saturating operations
  - On overflow, result is largest representable signed value
    - E.g., clipping in audio, saturation in video

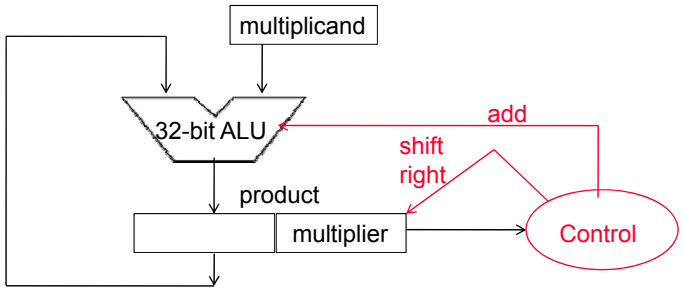
Multiply

- Binary multiplication is just a *bunch* of right shifts and adds



"If war were arithmetic, the mathematicians would rule the world."

Add and Right Shift Multiplier Hardware



MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product

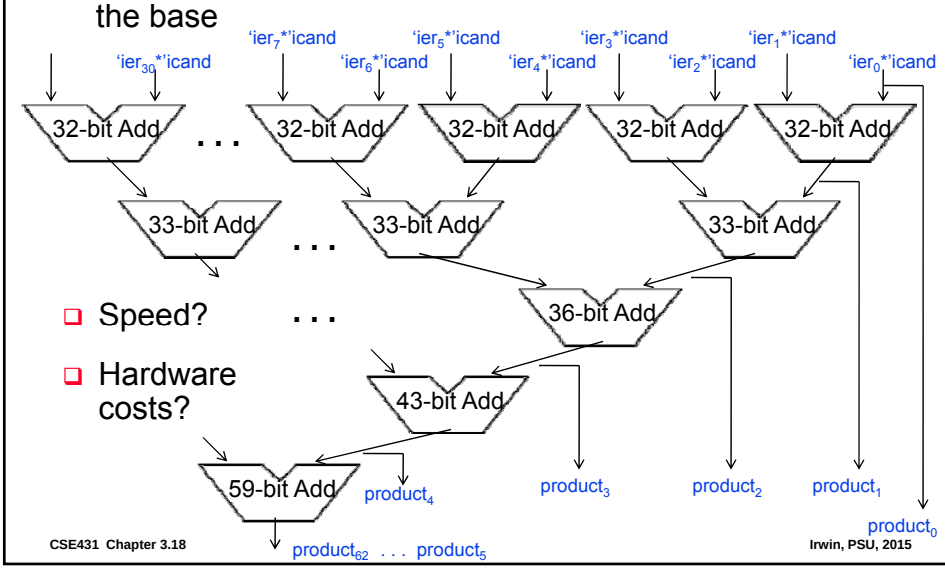
```
mult   $s0, $s1      # hi||lo = $s0 * $s1
```

0	16	17	0	0	0x18
alu	$s0	$s1	unused	unused	mult

  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file
- What is the speed of our add and right shift (serial) 4-bit multiplier assuming a RCA is used?
- Thus, multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Fast Multiplication Hardware

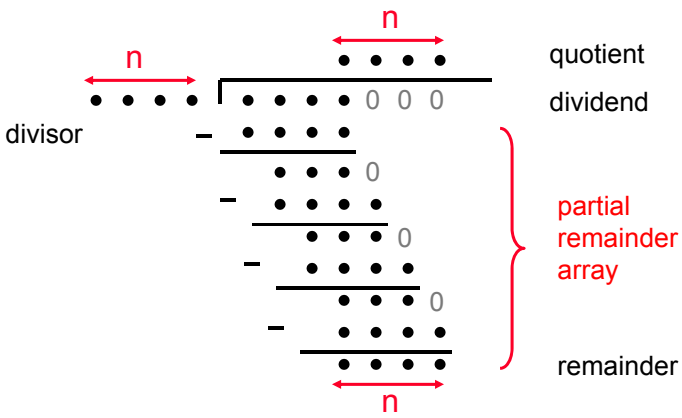
- Can build a faster multiplier by using a parallel tree of adders with one 32-bit adder for each bit of the multiplier at the base
- Speed?
- Hardware costs?



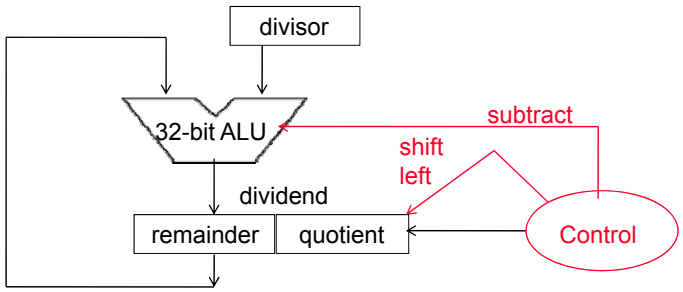
Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

dividend = quotient x divisor + remainder



Left Shift and Subtract Division Hardware



Aside: Previous Example, Step by Step

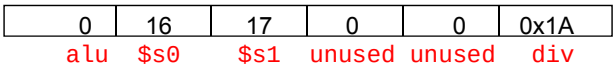
State of rem | quot	How we got to that state
0000 0110	initialize remainder | quotient with 0 | dividend
0000 1100	shift left 1
1110 1100	trial subtract (remainder – divisor, 0000 – 0010 is 1110) set the lsb of quotient to 0
0000 1100	trial subtract result is negative, restore the previous remainder
0001 1000	shift left
1111 1000	trial subtract (0001 – 0010 is 1111) set the lsb of quotient to 0
0001 1000	trial subtract result is negative, restore the previous remainder
0011 0000	shift left
0001 0001	trial subtract (0011 – 0010 is 0001) set the lsb of quotient to 1
0010 0010	shift left
0000 0011	trial subtract (0010 – 0010 is 0000) set the lsb of quotient to 1

lsb of quotient is set to complement of msb of trial subtract result

MIPS Divide Instruction

- Divide (`div` and `divu`) generates the reminder in `hi` and the quotient in `lo`

```
div   $s0, $s1      # lo = $s0 / $s1
                    # hi = $s0 mod $s1
```



  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- Speed? Hardware costs?
- As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

Representing Big (and Small) Numbers

- What if we want to encode the approx. age of the earth?

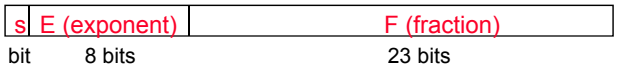
  4,600,000,000 or 4.6×10^9

  or the weight in kg of one a.m.u. (atomic mass unit)

  0.00000000000000000000000000166 or 1.6×10^{-27}

  There is no way we can encode either of the above in a 32-bit integer.

- Floating point representation $(-1)^{sign} \times F \times 2^E$
  - **Still** have to fit everything in 32 bits (single precision)



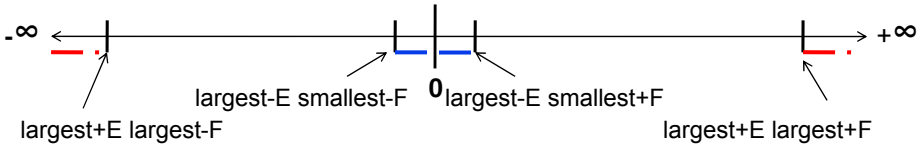
  - The base (2, *not* 10) is **hardwired** in the design of the FPALU
  - More bits in the fraction (F) or the exponent (E) is a trade-off between precision (roughly, the accuracy of the number) and range (roughly, the magnitude of the number)

Aside: Precision vs. Accuracy

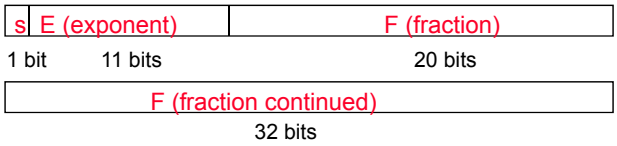
- Precision and accuracy are not the same concept.
- Precision is a quality of one number on its own – how many bits or digits do we retain?
  - How many do you trust? is a question for data collection, statistical analysis, numerical analysis, and hardware design.
- Accuracy is a quality of two numbers – how close are they?
  - When discussing accuracy, one of the two numbers is assumed to be "right". For example, 4.5678 is precise to 5 digits, but as an approximation to π it is only accurate to one-half digit, since (4.5678 – 3.1416) / 3.1416 = 0.45397
  - When discussing precision, you could say that a number is accurate to within ½ unit in the last place. Why?

Exception Events in Floating Point

- Overflow (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- Underflow (floating point) happens when a negative exponent becomes too large to fit in the exponent field



- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
  - Double precision – takes two MIPS words



More Number Representations

- 24-bit signed fractions (sign magnitude):

$X = x_{-1}2^{-1} + x_{-2}2^{-2} + \ldots + x_{-22}2^{-22} + x_{-23}2^{-23}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = +\ 0$
$0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 2^{-23}$
...

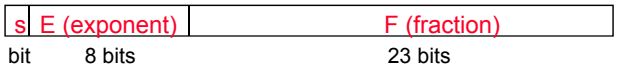
$0111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 1 - 2^{-22}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 1 - 2^{-23}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 0$
$1000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2^{-23}$
...

$1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 1 - 2^{-22}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1 - 2^{-23}$

MSB = **sign bit**

- The most significant bit, s, is the sign bit of the fraction, F
  - To perform negation, just flip the sign bit (easy!)
    - The magnitude of the most positive F and most negative F are the same (not as in two's complement)
  - There are now two representations for an F of 0, one positive, one negative

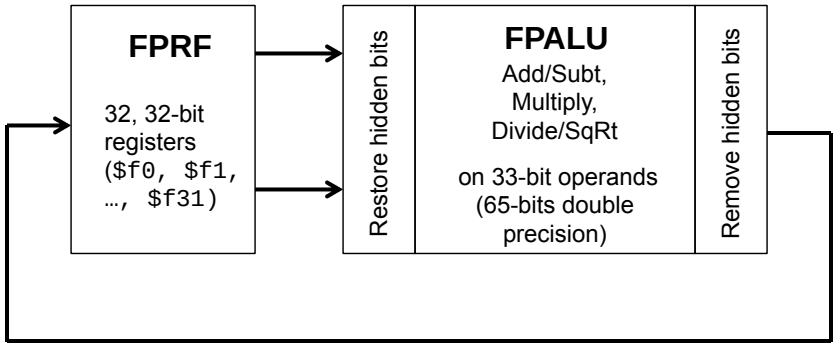
IEEE 754 FP Standard



- Most (all?) computers these days conform to the IEEE 754 floating point standard $(-1)^{sign} \times (1+F) \times 2^{E-bias}$
  - With formats for both single and double precision
  - F is stored in normalized sign magnitude format
    - Normalized means that before the result is put into the FPRF, F is shifted (left) and E decremented (once for every left bit shift) until the msb of F is a 1 (so there is no need to store it!) – called the hidden bit
    - So when a value is read from the FPRF, the hidden bit of 1 has to be restored before performing computation
  - To simplify sorting FP numbers, E comes before F in the word and E is represented in excess (biased) notation where the bias is 127 (1023 for double precision) so the *most negative* exponent is **0**0000001 = $2^{1-127} = 2^{-126}$ and the *most positive* exponent is **1**1111110 = $2^{254-127} = 2^{+127}$ (exponents 00000000 and 11111111 are reserved for special uses)

Floating Point Arithmetic Hardware

- Floating Point Register File (FPRF) and floating point ALU (FPALU)



IEEE 754 FP Examples

$(1+F) \times 2^{E-bias}$

- Examples (in normalized format)
  - Smallest+ = $1 \times 2^{1-127} = 1 \times 2^{-126} = \approx \pm 1.2 \times 10^{-38}$
    0 00000001 1.00000000000000000000000
  - Zero (true 0): 0 00000000 0.00000000000000000000000
    - Note that hardware detects true 0 (F and E all zeros) and does not restored the hidden bit of 1
  - Largest+ = $1 + 1\text{-}2^{-23} \times 2^{254-127} = 2 - 2^{-23} \times 2^{127} = \approx \pm 3.4 \times 10^{+38}$
    0 11111110 1.11111111111111111111111
  - $1.0_2 \times 2^{-1} =$
  - $0.75_{10} \times 2^4 =$

IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
  - ± infinity for division by zero
  - NAN (not a number) for the results of invalid operations such as 0/0
  - True zero is the bit string all zero

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0000	0	0000 … 0000	0	true zero (0)
0000 0000	nonzero	0000 … 0000	nonzero	± denormalized number
0000 0001 to 1111 1110	anything	0000 … 0001 to 1111 … 1110	anything	± floating point number
1111 1111	± 0	1111 … 1111	± 0	± infinity
1111 1111	nonzero	1111 … 1111	nonzero	not a number (NaN)

Support for Accurate Arithmetic

- IEEE 754 FP rounding modes
  - Always round up (toward +∞)
  - Always round down (toward -∞)
  - Truncate
  - Round to nearest even (when the Guard || Round || Sticky are 100) – always creates a 0 in the least significant (kept) bit of F
- Rounding (except for truncation) requires the FPALU to include three extra F bits during calculations
  - Guard bit G – used to provide one F bit when shifting left to normalize a result (e.g., when normalizing F after division or subt)
  - Round bit R– used to improve rounding accuracy
  - Sticky bit S – used to support Round to nearest even; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F during addition/subtraction)

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

Floating Point Addition

- Addition (and subtraction)

$$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - Step 0: Restore the hidden bit in F1 and in F2
  - Step 1: Align fractions by right shifting F2 by E1 - E2 positions (assuming E1 ≥ E2) keeping track of (three of) the bits shifted out in G R and S
  - Step 2: Add the resulting F2 to F1 to form F3
  - Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
    - If F1 and F2 have the same sign → F3 ∈[1,4) → 1 bit right shift F3 and increment E3 (check for overflow)
    - If F1 and F2 have different signs → F3 may require *many* left shifts each time decrementing E3 (check for underflow)
  - Step 4: Round F3 and possibly normalize F3 again
  - Step 5: Rehide the most significant (hidden) bit of F3 before storing the result

Floating Point Addition Example

- Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

  - Step 0: Hidden bits restored in the representation above
  - Step 1:
  - Step 2:
  - Step 3:
  - Step 4:
  - Step 5: Rehide the hidden bit before storing

Floating Point Multiplication

- Multiplication

$$(\pm F1 \times 2^{E1}) \times (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - Step 0: Restore the hidden bit in F1 and in F2
  - Step 1: Add the two (biased) exponents and subtract the bias from the sum, so E1 + E2 – 127 = E3

    also determine the sign of the product (which depends on the sign of the operands (their most significant bits))
  - Step 2: Multiply F1 by F2 to form a double precision F3
  - Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
    - Since F1 and F2 come in normalized $\rightarrow$ F3 $\in$[1,4) $\rightarrow$ 1 bit right shift F3 and increment E3
    - Check for overflow/underflow
  - Step 4: Round F3 and possibly normalize F3 again
  - Step 5: Rehide the most significant (hidden) bit of F3 before storing the result

Floating Point Multiplication Example

- Multiply

  $(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$

  - Step 0: Hidden bits restored in the representation above
  - Step 1:
  - Step 2:
  - Step 3:
  - Step 4:
  - Step 5: Rehide the hidden bit before storing

MIPS Floating Point Instructions

- MIPS has a separate Floating Point RF (`$f0, $f1, …, $f31`) (whose registers are used in *pairs* for double precision values) with special instructions to load to and store from them

```
lwcl  $f1,54($s2)   #$f1 = Memory[$s2+54]
swcl  $f1,58($s4)   #Memory[$s4+58] = $f1
```

- And supports IEEE 754 single

```
add.s $f2,$f4,$f6   #$f2 = $f4 + $f6
```

and double precision operations

```
add.d $f2,$f4,$f6   #$f2||$f3 =
                        $f4||$f5 + $f6||$f7
```

similarly for `sub.s, sub.d, mul.s, mul.d, div.s, div.d`

MIPS Floating Point Instructions, Con't

- And floating point single precision comparison operations

```
c.x.s $f2,$f4        #if($f2 < $f4) cond=1;
                         else cond=0
```

where x may be `eq, neq, lt, le, gt, ge`

and double precision comparison operations

```
c.x.d $f2,$f4        #$f2||$f3 < $f4||$f5
                         cond=1; else cond=0
```

- And floating point branch operations

```
bclt  25             #if(cond==1)
                         go to PC+4+25

bclf  25             #if(cond==0)
                         go to PC+4+25
```


Frequency of Common MIPS Instructions

- Only included those with >3% and >1%

	SPECint	SPECfp
addu	5.2%	3.5%
addiu	9.0%	7.2%
or	4.0%	1.2%
sll	4.4%	1.9%
lui	3.3%	0.5%
lw	18.6%	5.8%
sw	7.6%	2.0%
lbu	3.7%	0.1%
beq	8.6%	2.2%
bne	8.4%	1.4%
slt	9.9%	2.3%
slti	3.1%	0.3%
sltu	3.4%	0.8%

	SPECint	SPECfp
add.d	0.0%	10.6%
sub.d	0.0%	4.9%
mul.d	0.0%	15.0%
add.s	0.0%	1.5%
sub.s	0.0%	1.8%
mul.s	0.0%	2.4%
l.d	0.0%	17.5%
s.d	0.0%	4.9%
l.s	0.0%	4.2%
s.s	0.0%	1.1%
lhu	1.3%	0.0%

Pitfalls and Fallacies

- *Fallacy*: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as integer division by a power of 2.
  - True for unsigned integers, not true for 2's complement integers
    - E.g., -5 ÷ 4 → 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
- *Pitfall*: Floating-point addition is not associative.
  - Does c + (a + b) = (c + a) + b ?
    - If c = $-1.5_{10} \times 10^{38}$, a = $1.5_{10} \times 10^{38}$, and b = 1, then NO - left way gives 0.0 and right way gives 1.0 due to limited precision limitations
- *Fallacy*: Parallel execution strategies that work for integer data types also work for FP data types
  - *Order* of arithmetic operations is important in FP (see Pitfall above)
    - For more take CmpSc 451, 454, 455

What's Next and Reminders

- Next week
  - MIPS datapath and control, PH 4.1-4.4
  - MIPS pipelined datapaths, data hazards, PH 4.5-4.7
- Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9th and 10th from 7:30 to 9pm (HW2 will contain a SimpleScalar simulation question)
- Reminders
  - HW1 due midnight Sept 3rd in Angel DropBox
  - Quiz 1 will close midnight Sept 7th
  - HW2 will come out Sept 4th (our first with a SimpleScalar question)
  - First evening midterm exam scheduled
    - Tuesday, October 6th, 20:15 to 22:15, Location 22 Deike
    - Please let me know ASAP (via email) if you have a conflict