

---

# CmpEn 431 Computer Architecture Fall 2015

## Chapter 3: Arithmetic for Computers

Mary Jane Irwin ( [www.cse.psu.edu/~mji](http://www.cse.psu.edu/~mji) )

[Adapted from *Computer Organization and Design, 5<sup>th</sup> 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 3<sup>rd</sup> in Angel DropBox
- Quiz 1 will close midnight Sept 7<sup>th</sup>
- Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9<sup>th</sup> and 10<sup>th</sup> from 7:30 to 9pm
- HW2 will come out Sept 4<sup>th</sup> (our first using SimpleScalar)
- First evening midterm exam scheduled
  - Tuesday, **October 6<sup>th</sup>**, 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.

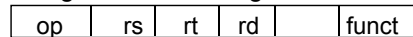


### □ Good design demands good compromises

- three instruction formats

## Review: MIPS Addressing Modes Illustrated

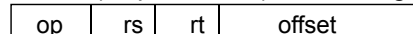
### 1. Register addressing



Register

word **operand**

### 2. Base (displacement) addressing

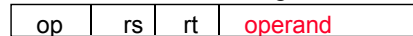


Memory

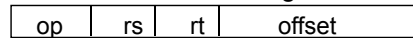
word or byte **operand**

base register

### 3. Immediate addressing



### 4. PC-relative addressing

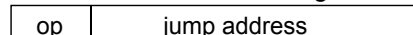


Memory

branch destination **instruction**

Program Counter (PC)

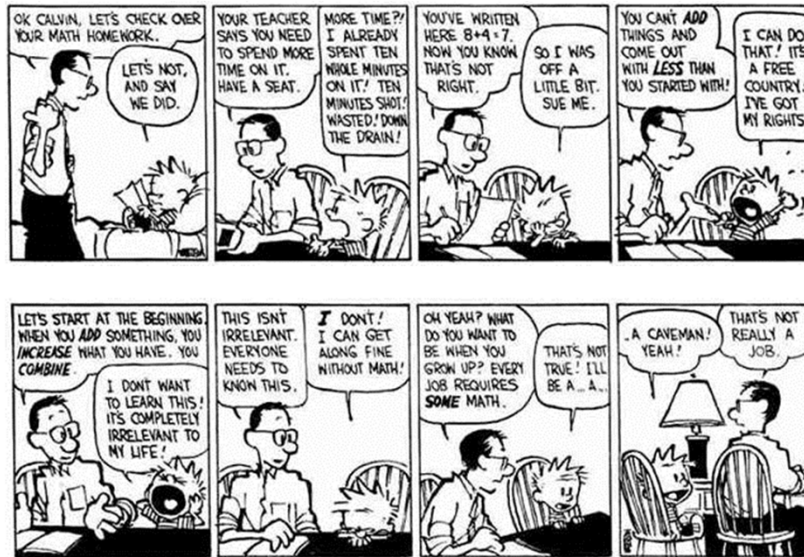
### 5. Pseudo-direct addressing



Memory

jump destination **instruction**

Program Counter (PC)



CSE431 Chapter 3.5

Irwin, PSU, 2015

## Number Representations

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

0000 0000 0000 0000 0000 0000 0000 0000	$_{two} = 0_{ten}$	
0000 0000 0000 0000 0000 0000 0000 0001	$_{two} = +1_{ten}$	
...		
0111 1111 1111 1111 1111 1111 1111 1110	$_{two} = +2,147,483,646_{ten}$	
0111 1111 1111 1111 1111 1111 1111 1111	$_{two} = +2,147,483,647_{ten}$	<i>maxint</i>
1000 0000 0000 0000 0000 0000 0000 0000	$_{two} = -2,147,483,648_{ten}$	
1000 0000 0000 0000 0000 0000 0000 0001	$_{two} = -2,147,483,647_{ten}$	
...		
1111 1111 1111 1111 1111 1111 1111 1110	$_{two} = -2_{ten}$	
1111 1111 1111 1111 1111 1111 1111 1111	$_{two} = -1_{ten}$	<i>minint</i>

MSB = sign bit      LSB

### 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)

CSE431 Chapter 3.6

Irwin, PSU, 2015

## MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

add, addi, addiu, addu

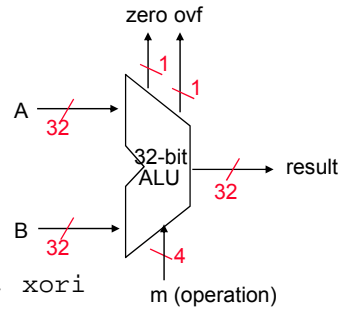
sub, subu

mult, multu, div, divu

sqr

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

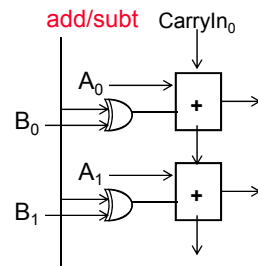
Operation	Operand A	Operand B	Result indicating overflow
A + B	$\geq 0$	$\geq 0$	$< 0$
A + B	$< 0$	$< 0$	$\geq 0$
A - B	$\geq 0$	$< 0$	$< 0$
A - B	$< 0$	$\geq 0$	$\geq 0$

- 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

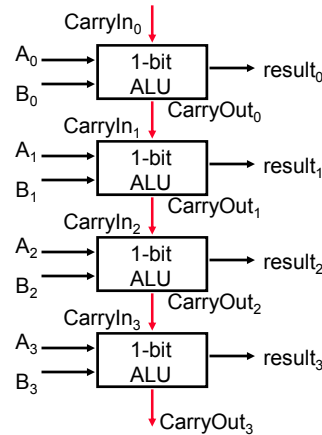
## Addition and Subtraction on the MIPS ALU

- 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?

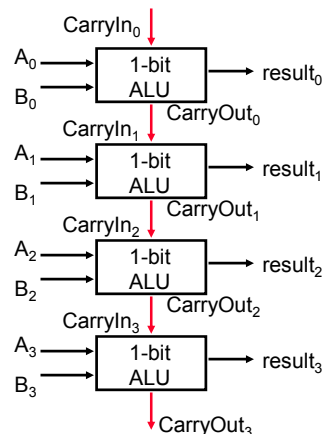


- In reality, just negating and then adding !



## But What about Performance?

- Critical path of n-bit RCA is  $n \cdot CP = O(n)$



n	RCA	CLA
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7

- Design trick – throw hardware at it (Carry Lookahead Adder, **prefix 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

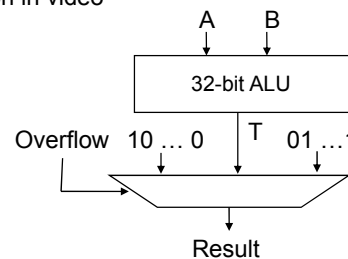


### □ Saturating operations

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

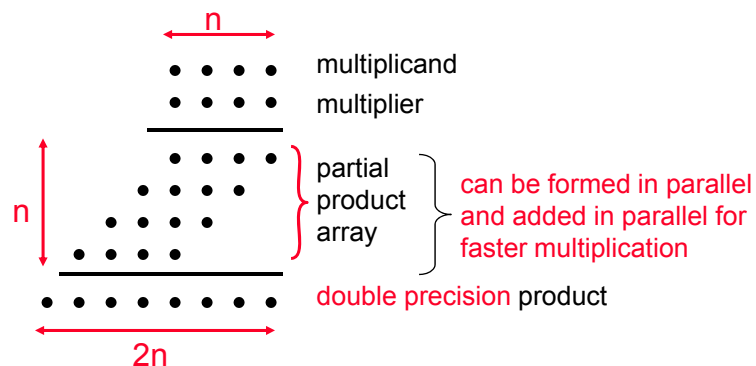
$$\text{Overflow}_{\text{pos}} = T_{31} \ \& \ !A_{31} \ \& \ !B_{31}$$

$$\text{Overflow}_{\text{neg}} = !T_{31} \ \& \ A_{31} \ \& \ B_{31}$$



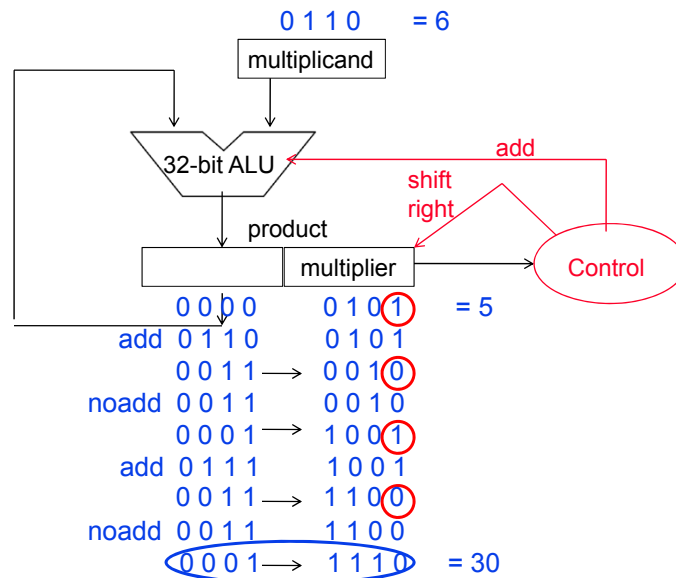
## Multiply

- 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



CSE431 Chapter 3.16

Irwin, PSU, 2015

## 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

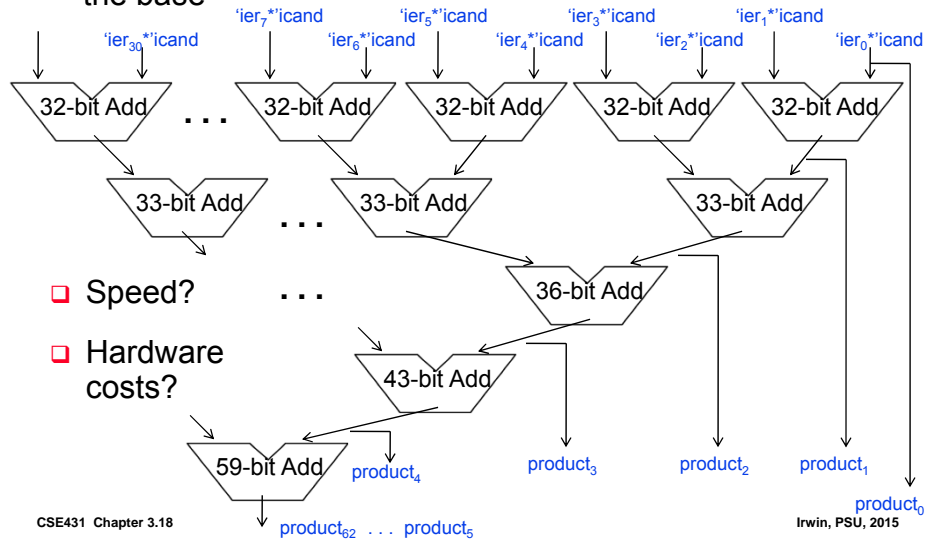
- Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
- Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file
- ❑ What is the speed of our add and right shift (serial) 4-bit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

CSE431 Chapter 3.17

Irwin, PSU, 2015

## Fast Multiplication Hardware

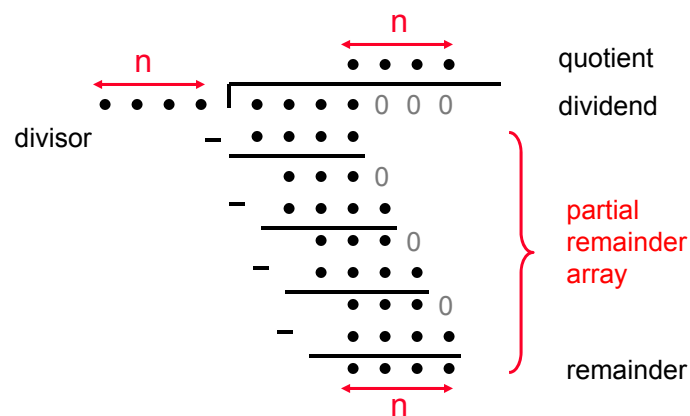
- Can build a faster multiplier by using a parallel tree of adders with one 32-bit adder for each bit of the multiplier at the base



## Division

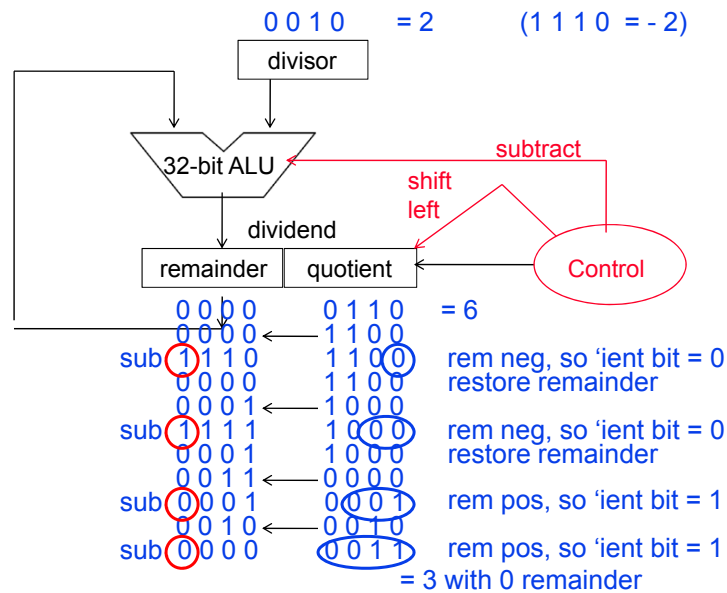
- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

$$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$





## Left Shift and Subtract Division Hardware



CSE431 Chapter 3.21

Irwin, PSU, 2015

## 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

CSE431 Chapter 3.22

Irwin, PSU, 2015

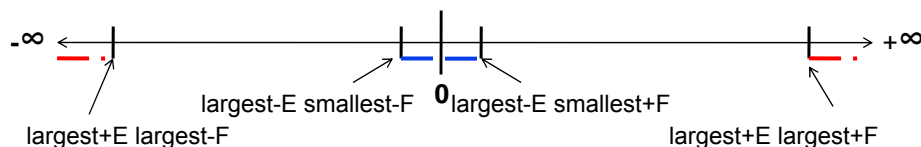


## Aside: Precision vs. Accuracy

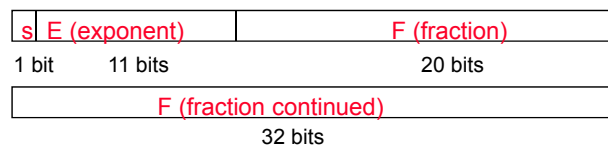
- ❑ Precision and accuracy are not the same concept.
- ❑ Precision is a quality of one number on its own – how many bits or digits do we retain?
  - How many do you trust? is a question for data collection, statistical analysis, numerical analysis, and hardware design.
- ❑ Accuracy is a quality of two 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  $\pi$  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  $\frac{1}{2}$  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):

$$X = x_1 2^{-1} + x_2 2^{-2} + \dots + x_{22} 2^{-22} + x_{23} 2^{-23}$$

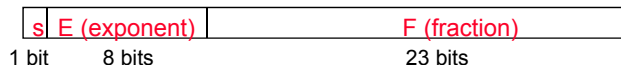
0000 0000 0000 0000 0000 0000<sub>two</sub> = + 0  
 0000 0000 0000 0000 0000 0001<sub>two</sub> = + 2<sup>-23</sup>  
 ...  
 0111 1111 1111 1111 1111 1110<sub>two</sub> = + (1 - 2<sup>-22</sup>)  
 0111 1111 1111 1111 1111 1111<sub>two</sub> = + (1 - 2<sup>-23</sup>)  
 1000 0000 0000 0000 0000 0000<sub>two</sub> = - 0  
 1000 0000 0000 0000 0000 0001<sub>two</sub> = - 2<sup>-23</sup>  
 ...  
 1111 1111 1111 1111 1111 1110<sub>two</sub> = - (1 - 2<sup>-22</sup>)  
 1111 1111 1111 1111 1111 1111<sub>two</sub> = - (1 - 2<sup>-23</sup>)

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

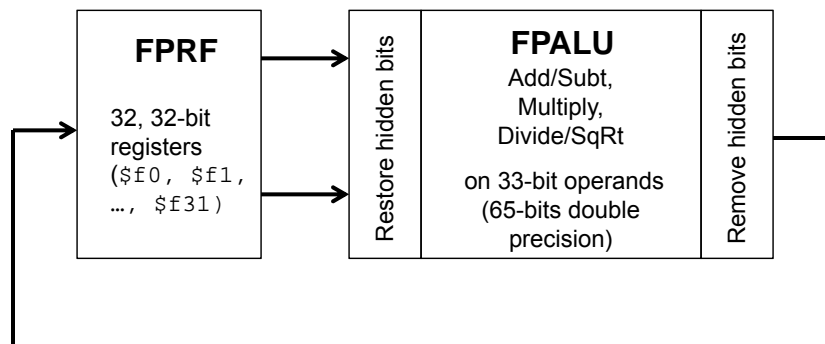


### Most (all?) computers these days conform to the IEEE 754 floating point standard $(-1)^{\text{sign}} \times (1+F) \times 2^{E-\text{bias}}$

- 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  $00000001 = 2^{1-127} = 2^{-126}$  and the *most positive* exponent is  $11111110 = 2^{254-127} = 2^{+127}$  (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

- Examples (in **normalized** format)  $(1+F) \times 2^{E-bias}$

- Smallest+ =  $1 \times 2^{1-127} = 1 \times 2^{-126} \approx 1.2 \times 10^{-38}$   
 $0\ 00000001\ 1.000000000000000000000000$
- Zero (true 0):  $0\ 00000000\ 0.000000000000000000000000$   
 - Note that hardware detects true 0 (F and E all zeros) and does not restore the hidden bit of 1
- Largest+ =  $(1 + 1-2^{-23}) \times 2^{254-127} = (2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{+38}$   
 $0\ 11111110\ 1.111111111111111111111111$
- $1.0_2 \times 2^{-1} = (1+0) \times 2^{126-127}$   
 $0\ 01111110\ 1.000000000000000000000000$
- $0.75_{10} \times 2^4 = 0.11 \times 2^{131-127}$   
 $0\ 10000011\ 0.110000000000000000000000$   
 $\quad \quad \quad \downarrow$   
 $0\ 10000010\ 1.100000000000000000000000$

## IEEE 754 FP Standard Encoding

- ❑ Special encodings are used to represent unusual events
  - $\pm$  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	$\pm$ denormalized number
0000 0001 to 1111 1110	anything	0000 ... 0001 to 1111 ... 1110	anything	$\pm$ floating point number
1111 1111	$\pm 0$	1111 ... 1111	$\pm 0$	$\pm$ infinity
1111 1111	nonzero	1111 ... 1111	nonzero	not a number (NaN)

## Support for Accurate Arithmetic

- ❑ IEEE 754 FP rounding modes
  - Always round up (toward  $+\infty$ )
  - Always round down (toward  $-\infty$ )
  - 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)

$F = 1 . \text{xxxxxxxxxxxxxxxxxxxxxxxxxx} \text{ G R S}$

## 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  $E1 \geq E2$ ) 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  $\rightarrow F3 \in [1,4) \rightarrow 1$  bit **right** shift F3 and **increment** E3 (check for overflow)
  - If F1 and F2 have different signs  $\rightarrow F3$  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: **Shift fraction with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)**
- Step 2: **Add fractions**  
 $1.0000 + (-0.111) = 1.0000 - 0.111 = 0.001$
- Step 3: **Normalize the sum, checking for exponent over/underflow**  
 $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = \dots = 1.000 \times 2^{-4}$
- Step 4: **The sum is already rounded, so we're done**
- 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: Add the exponents (not in bias would be  $-1 + (-2) = -3$  and in bias would be  $(-1+127) + (-2+127) - 127 = (-1 + -2) + (127+127-127) = -3 + 127 = 124$ )
- Step 2: Multiply the fractions  
 $1.0000 \times 1.1100 = 1.110000$
- Step 3: Normalized the product, checking for exp over/underflow  
 $1.110000 \times 2^{-3}$  is already normalized
- Step 4: The product is already rounded, so we're done
- 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%

CSE431 Chapter 3.42

Irwin, PSU, 2015

## 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.,  $-5 \div 4 \rightarrow 11111011_2 \gg 2 = 11111110_2 = -2$
    - Rounds toward  $-\infty$
- 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

CSE431 Chapter 3.43

Irwin, PSU, 2015

## 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 9<sup>th</sup> and 10<sup>th</sup> from 7:30 to 9pm (HW2 will contain a SimpleScalar simulation question)

### ❑ Reminders

- HW1 due midnight Sept 3<sup>rd</sup> in Angel DropBox
- Quiz 1 will close midnight Sept 7<sup>th</sup>
- HW2 will come out Sept 4<sup>th</sup> (our first with a SimpleScalar question)
- First evening midterm exam scheduled
  - Tuesday, **October 6<sup>th</sup>**, 20:15 to 22:15, Location 22 Deike
  - Please let me know ASAP (via email) if you have a conflict