

---

# COSC 3327

## Computer Architecture

### Chapter 3: Arithmetic for Computers

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

Grateful acknowledgement to Dr. Mary Jane Irwin of Penn State University for additional materials.

COSC 3327 Chapter 3.1

---

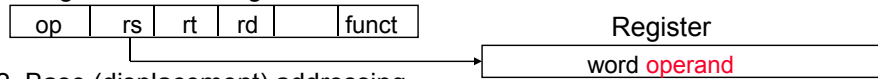
### Review: MIPS (RISC) Design Principles

- ❑ **Simplicity favors regularity**
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- ❑ **Smaller is faster**
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- ❑ **Make the common case fast**
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- ❑ **Good design demands good compromises**
  - three instruction formats

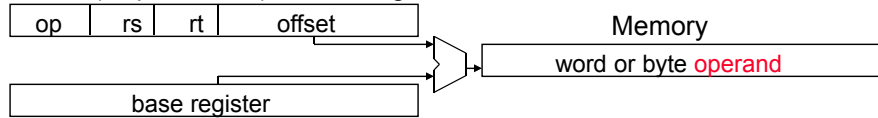
COSC 3327 Chapter 3.2

## Review: MIPS Addressing Modes Illustrated

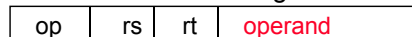
### 1. Register addressing



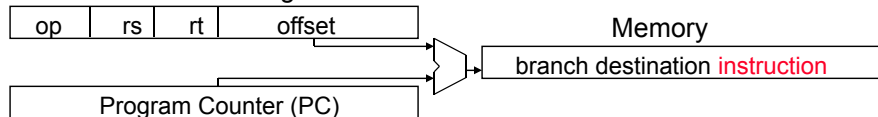
### 2. Base (displacement) addressing



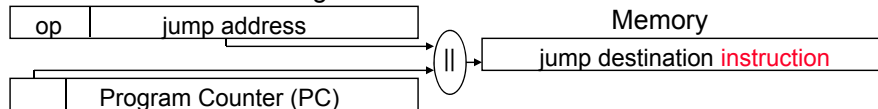
### 3. Immediate addressing



### 4. PC-relative addressing



### 5. Pseudo-direct addressing



COSC 3327 Chapter 3.3

## Number Representations

### 32-bit signed numbers (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 (Most Significant Bit) is circled in blue on the left. LSB (Least Significant Bit) is circled in blue on the right.

### Converting <32-bit values into 32-bit values

- copy the most significant bit (the sign bit) into the "empty" bits
  - 0010  $\rightarrow$  0000 0010
  - 1010  $\rightarrow$  1111 1010
- sign extend versus zero extend (l<sub>b</sub> vs. l<sub>bu</sub>)

COSC 3327 Chapter 3.4

## MIPS Arithmetic Logic Unit (ALU)

- ❑ Must support the Arithmetic/Logic operations of the ISA

add, addi, addiu, addu

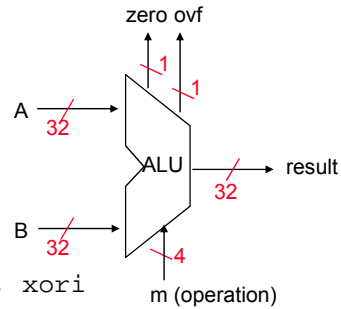
sub, subu

mult, multu, div, divu

sqr

and, andi, nor, or, ori, xor, xori

beq, bne, slt, slti, sltiu, sltu

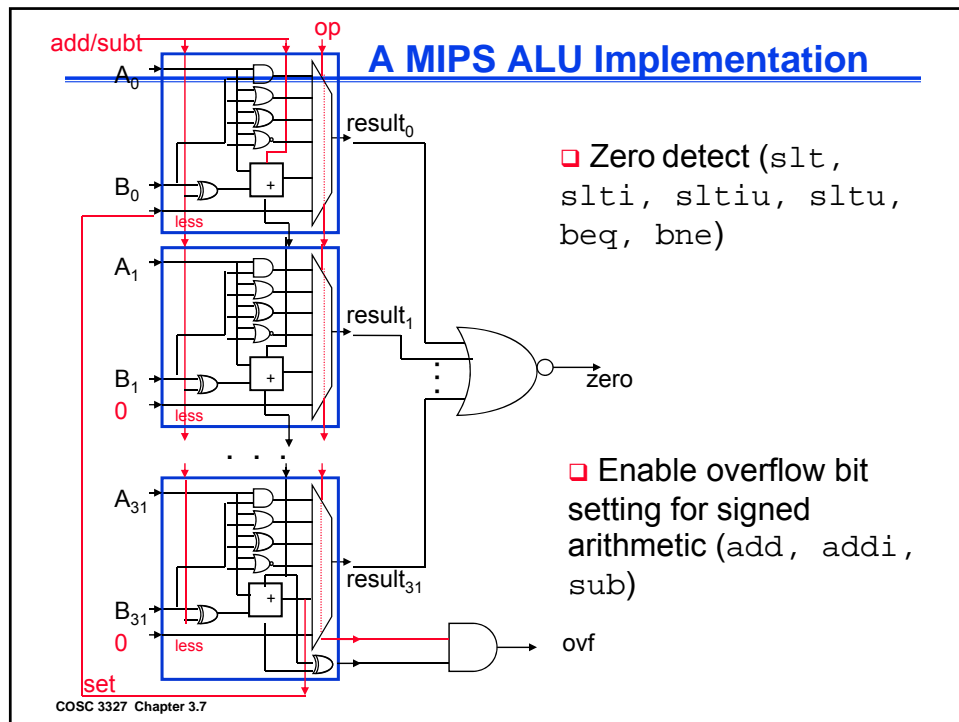


- ❑ With special handling for

- sign extend – addi, addiu, slti, sltiu
- zero extend – andi, ori, xori
- overflow detection – add, addi, sub

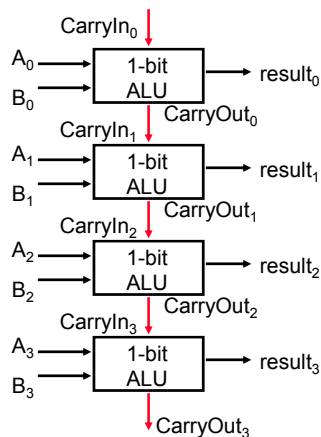
## Dealing with Overflow

- ❑ Some languages (e.g., C) ignore overflow
  - Use MIPS addu, addui, subu instructions
- ❑ Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS add, addi, sub instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action



### But What about Performance?

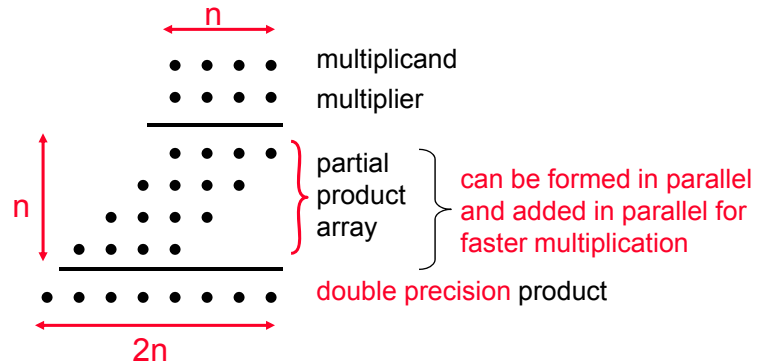
- ☐ Critical path of n-bit ripple-carry adder is  $n \cdot CP$



- ☐ Design trick – throw hardware at it (Carry Lookahead)

## Multiply

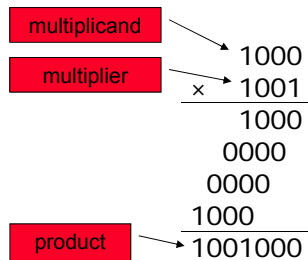
- Binary multiplication is just a *bunch* of right shifts and adds



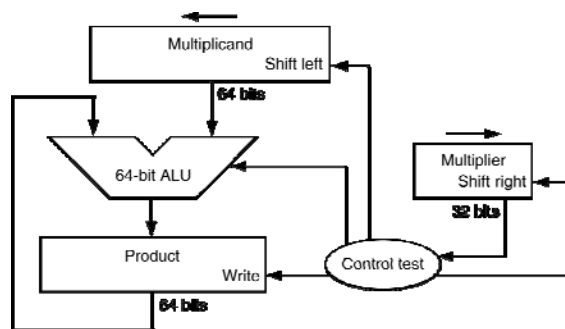
COSC 3327 Chapter 3.9

## Multiplication

- Start with long-multiplication approach

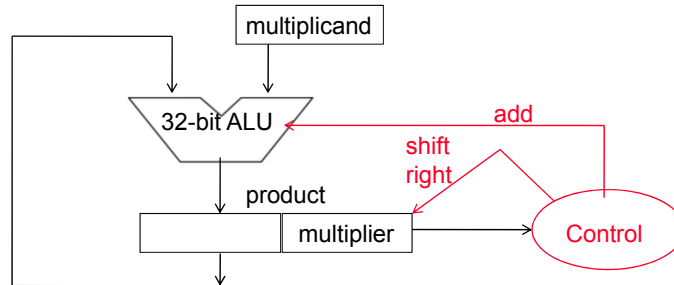


Length of product is the sum of operand lengths



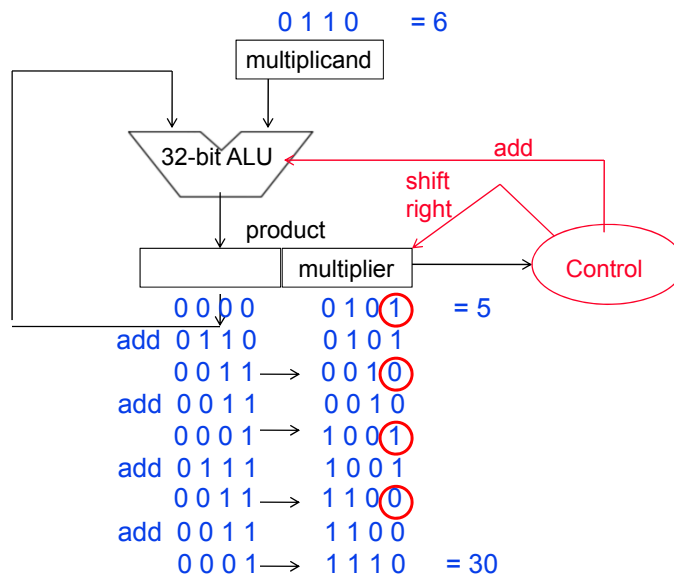
S3.3 Multiplication

## Add and Right Shift Multiplier Hardware



COSC 3327 Chapter 3.11

## Add and Right Shift Multiplier Hardware



COSC 3327 Chapter 3.12

## MIPS Multiply Instruction

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

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

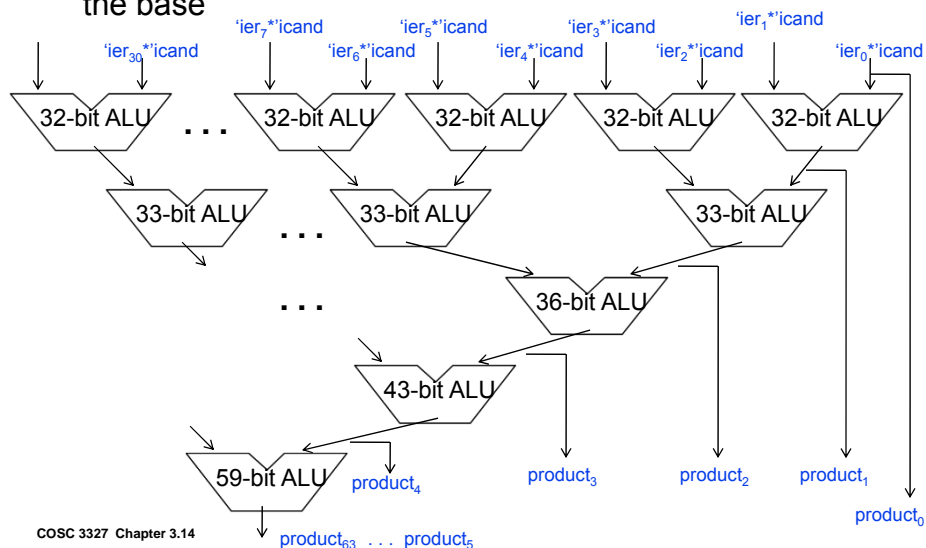
0	16	17	0	0	0x18
---	----	----	---	---	------

- 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
- 
- ❑ Multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

COSC 3327 Chapter 3.13

## 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

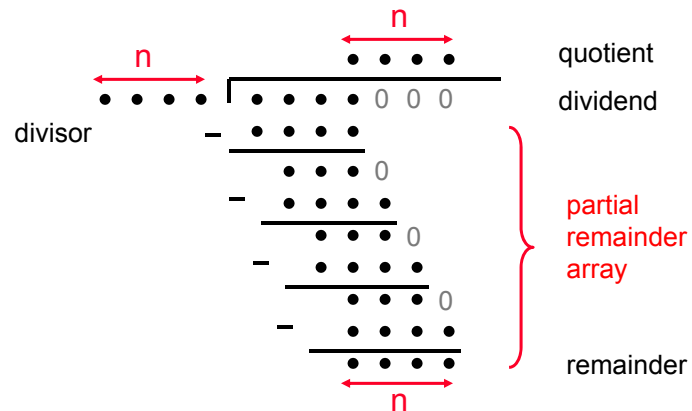


COSC 3327 Chapter 3.14

## Division

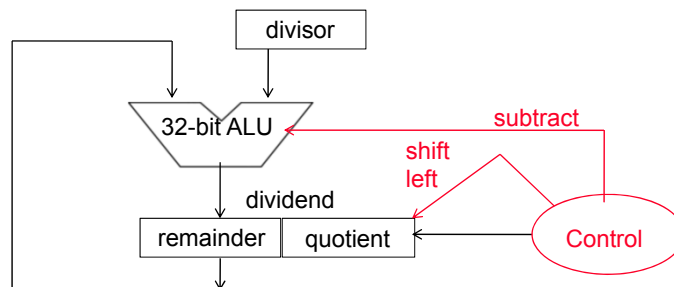
- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

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



COSC 3327 Chapter 3.15

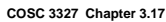
## Left Shift and Subtract Division Hardware



COSC 3327 Chapter 3.16



## Left Shift and Subtract Division Hardware



## MIPS Divide Instruction

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

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

0	16	17	0	0	0x1A
---	----	----	---	---	------

- Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and remainder to (user accessible) registers in the register file
- ❑ 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       $4.6 \times 10^9$

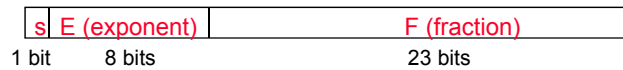
or the weight in kg of one a.m.u. (atomic mass unit)

[illegible]

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

- ❑ Floating point representation  $(-1)^{\text{sign}} \times F \times 2^E$

- Still have to fit everything in 32 bits (single precision)



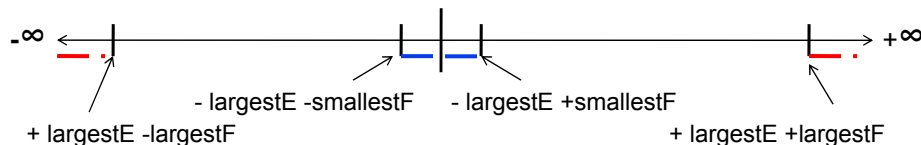
- The base (2, *not* 10) is hardwired in the design of the FPALU
- More bits in the fraction (F) or the exponent (E) is a trade-off between **precision** (accuracy of the number) and **range** (size of the number)

COSC 3327 Chapter 3.19

## Exception Events in Floating Point

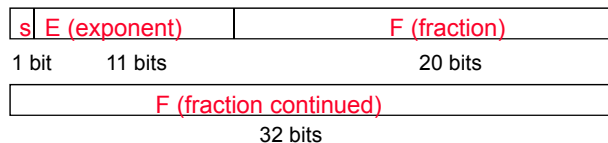
- ❑ **Overflow** (floating point) happens when a positive exponent becomes too large to fit in the exponent field

- ❑ **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



COSC 3327 Chapter 3.20

## IEEE 754 FP Standard

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

- Formats for both single and double precision
- F is stored in **normalized** format where the msb in F is 1 (so there is no need to store it!) – called the **hidden** bit
- 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 is 00000001 =  $2^{1-127} = 2^{-126}$  and the most positive is 11111110 =  $2^{254-127} = 2^{+127}$

□ Examples (in normalized format)

- Smallest+: 0 00000001 1.000000000000000000000000 =  $1 \times 2^{1-127}$
- Zero: 0 00000000 000000000000000000000000 = true 0
- Largest+: 0 11111110 1.111111111111111111111111 =  $2^{-2^{-23}} \times 2^{254-127}$
- $1.0_2 \times 2^{-1} =$
- $0.75_{10} \times 2^4 =$

COSC 3327 Chapter 3.21

## IEEE 754 FP Standard

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

- Formats for both single and double precision
- F is stored in **normalized** format where the msb in F is 1 (so there is no need to store it!) – called the **hidden** bit
- 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 is 00000001 =  $2^{1-127} = 2^{-126}$  and the most positive is 11111110 =  $2^{254-127} = 2^{+127}$

□ Examples (in normalized format)

- Smallest+: 0 00000001 1.000000000000000000000000 =  $1 \times 2^{1-127}$
- Zero: 0 00000000 000000000000000000000000 = true 0
- Largest+: 0 11111110 1.111111111111111111111111 =  $2^{-2^{-23}} \times 2^{254-127}$
- $1.0_2 \times 2^{-1} =$  0 01111110 1.000000000000000000000000
- $0.75_{10} \times 2^4 =$  0 10000010 1.100000000000000000000000

COSC 3327 Chapter 3.22

## 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
0111 1111 to +127,-126	anything	0111 ... 1111 to +1023,-1022	anything	$\pm$ floating point number
1111 1111	+ 0	1111 ... 1111	- 0	$\pm$ infinity
1111 1111	nonzero	1111 ... 1111	nonzero	not a number (NaN)

COSC 3327 Chapter 3.23

## 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 hardware to include extra F bits during calculations
  - Guard bit – used to provide one F bit when shifting left to normalize a result (e.g., when normalizing F after division or subtraction)
  - Round bit – used to improve rounding accuracy
  - Sticky bit – 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)

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

COSC 3327 Chapter 3.24

## 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 bit of F3 before storing the result

COSC 3327 Chapter 3.25

## Floating Point Addition Example

### □ Add

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

- Step 0:
- Step 1:
- Step 2:
- Step 3:
- Step 4:
- Step 5:

COSC 3327 Chapter 3.26

## 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 significand with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)
- Step 2: Add significands  
 $1.0000 + (-0.111) = 1.0000 - 0.111 = 0.001$
- 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

COSC 3327 Chapter 3.27

## 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 (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 bit of F3 before storing the result

COSC 3327 Chapter 3.28

## Floating Point Multiplication Example

### □ Multiply

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

- Step 0:
- Step 1:
  
- Step 2:
  
- Step 3:
  
- Step 4:
  
- Step 5:

## 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 significands  
 $1.0000 \times 1.110 = 1.110000$
- 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 Register File (\$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

COSC 3327 Chapter 3.31

## 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
```

COSC 3327 Chapter 3.32



## 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%