

# Arithmetic for Computers

Dr. Arjan Durrresi  
Louisiana State University  
Baton Rouge, LA 70803  
durrresi@csc.lsu.edu

These slides are available at:

[http://www.csc.lsu.edu/~durrresi/CSC3501\\_07/](http://www.csc.lsu.edu/~durrresi/CSC3501_07/)



- ❑ Signed and Unsigned Numbers
- ❑ Addition and Subtraction
- ❑ Multiplication and Division
- ❑ Floating Point

# Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2)  
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...  
decimal:  $0 \dots 2^n - 1$
- Of course it gets more complicated:  
numbers are finite (overflow) fractions and real numbers  
negative numbers e.g., no MIPS subi instruction; addi can  
add a negative number
- How do we represent negative numbers?  
i.e., which bit patterns will represent which numbers?

# Possible Representations

- | Sign Magnitude: | One's Complement | Two's Complement |
|-----------------|------------------|------------------|
| 000 = +0        | 000 = +0         | 000 = +0         |
| 001 = +1        | 001 = +1         | 001 = +1         |
| 010 = +2        | 010 = +2         | 010 = +2         |
| 011 = +3        | 011 = +3         | 011 = +3         |
| 100 = -0        | 100 = -3         | 100 = -4         |
| 101 = -1        | 101 = -2         | 101 = -3         |
| 110 = -2        | 110 = -1         | 110 = -2         |
| 111 = -3        | 111 = -0         | 111 = -1         |
- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

# MIPS

## □ 32 bit signed numbers:

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = + 2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = +
2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + / maxint
2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two = - \ minint
2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = -
2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two = -
2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = - 3ten
1111 1111 1111 1111 1111 1111 1111 1110two = - 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = - 1ten
```

# Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: "negate" and "invert" are quite different!
- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

```
0010 -> 0000 0010
1010 -> 1111 1010
```
  - "sign extension" (lbu vs. lb)

## MIPS Number Representations

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

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

MSB 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)

## Signed vs. Unsigned

### □ \$s0 has:

1111 1111 1111 1111 1111 1111 1111 1111<sub>two</sub>

### □ \$s1 has:

0000 0000 0000 0000 0000 0000 0000 0001<sub>two</sub>

```
slt $t0, $s0, $s1 #signed comparison
sltu $t1, $s0, $s1 #unsigned comparison
```

## Negation Shortcut

- $x + \bar{x} = -1$
- $x + \bar{x} + 1 = 0$
- $\bar{x} + 1 = -x$

## Sign Extension Shortcut

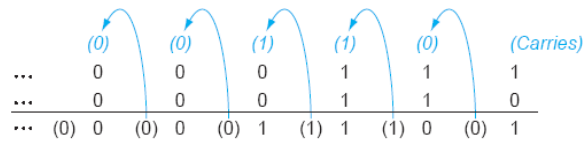
- Convert 16-bit binary version  $2_{\text{ten}}$  and  $-2_{\text{ten}}$  to 32-bit binary numbers

```
0000 0000 0000 0010two
0000 0000 0000 0000 0000 0000 0000 0010two

0000 0000 0000 0010two
negate
1111 1111 1111 1101two
+
                      1
=
1111 1111 1111 1110two (-2ten 16-bit)

1111 1111 1111 1111 1111 1111 1111 1110two (-2ten 32-
bit)
```

# Addition



## MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

add, addi, addiu, addu

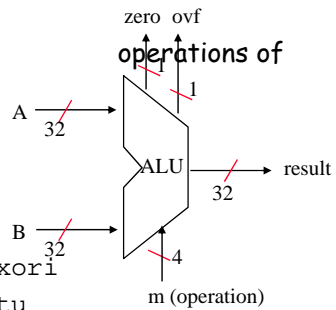
sub, subu, neg

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 andi, ori, xori, slti, sltiu

- zero extend – lbu, addiu, sltiu

- no overflow detected – addu, addiu, subu, multu, divu, sltiu, sltu

## Review: 2's Complement Binary Representation

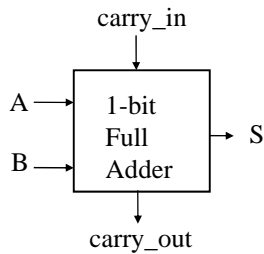
❑ Negate

1011  
and add a 1  
1010  
complement all the bits

❑ Note: negate and invert are different!

	2'sc binary	decimal
$-2^3 =$	1000	-8
$-(2^3 - 1) =$	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1 =$	0111	7

## Review: A Full Adder



A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus \text{carry\_in}$  (odd parity function)

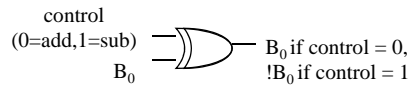
$\text{carry\_out} = A \& B \mid A \& \text{carry\_in} \mid B \& \text{carry\_in}$   
(majority function)

- ❑ How can we use it to build a 32-bit adder?
- ❑ How can we modify it easily to build an adder/subtractor?

## A 32-bit Ripple Carry Adder/Subtractor

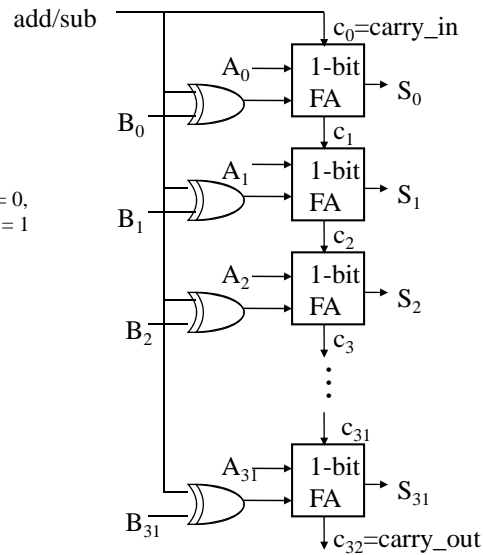
□ Remember 2's complement is just

- complement all the bits



- add a 1 in the least significant bit

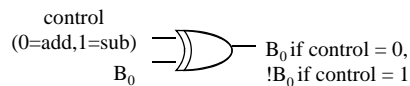
A    0111    →    0111  
B - 0110    →    +



## A 32-bit Ripple Carry Adder/Subtractor

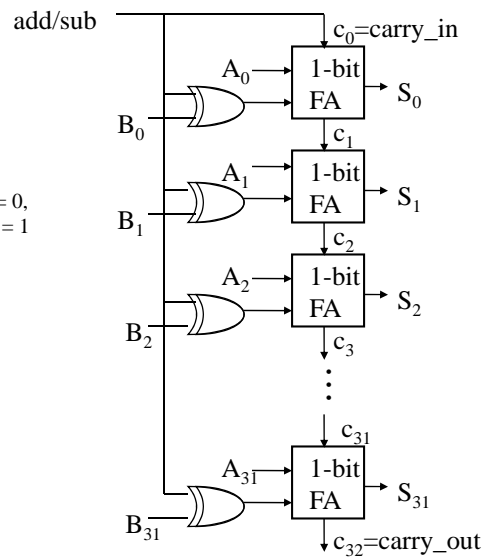
□ Remember 2's complement is just

- complement all the bits



- add a 1 in the least significant bit

A    0111    →    0111  
B - 0110    →    + 1001  
    0001            1  
                    1 0001



## Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

$$\begin{array}{r}
 0111 \quad 0111 \quad 0110 \\
 + 0110 \quad - 0110 \quad - 0101 \\
 \hline
 \end{array}$$

- Two's complement operations easy
  - subtraction using addition of negative numbers

$$\begin{array}{r}
 0111 \\
 + 1010 \\
 \hline
 \end{array}$$

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

$$\begin{array}{r}
 0111 \\
 + 0001 \\
 \hline
 \text{— } 1000
 \end{array}$$

*note that overflow term is somewhat misleading, it does not mean a carry "overflowed"*

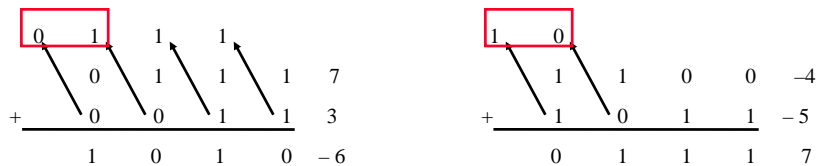
## Overflow Detection

- Overflow: the result is too large to represent in 32 bits
- Overflow occurs when
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On your own: **Prove** you can detect overflow by:
  - Carry *into* MSB xor Carry *out of* MSB, ex for 4 bit signed numbers

$$\begin{array}{r}
 \phantom{+} 0 \phantom{+} 1 \phantom{+} 1 \phantom{+} 1 \phantom{+} 7 \\
 + \phantom{+} 0 \phantom{+} 0 \phantom{+} 1 \phantom{+} 1 \phantom{+} 3 \\
 \hline
 \phantom{+} 0 \phantom{+} 1 \phantom{+} 1 \phantom{+} 0 \phantom{+} 0 \phantom{+} -4 \\
 + \phantom{+} 1 \phantom{+} 0 \phantom{+} 1 \phantom{+} 1 \phantom{+} -5 \\
 \hline
 \end{array}$$

## Overflow Detection

- ❑ Overflow: the result is too large to represent in 32 bits
- ❑ Overflow occurs when
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- ❑ On your own: **Prove** you can detect overflow by:
  - Carry into MSB xor Carry out of MSB, ex for 4 bit signed numbers



## Detecting Overflow

- ❑ No overflow when adding a positive and a negative number
- ❑ No overflow when signs are the same for subtraction
- ❑ Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- ❑ Consider the operations  $A + B$ , and  $A - B$ 
  - Can overflow occur if  $B$  is 0 ?
  - Can overflow occur if  $A$  is 0 ?

## Detecting Overflow

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

## Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  - new MIPS instructions: `addu`, `addiu`, `subu`

*note: addiu still sign-extends!*

*note: sltu, sltiu for unsigned comparisons*

## MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

add, addi, addiu, addu

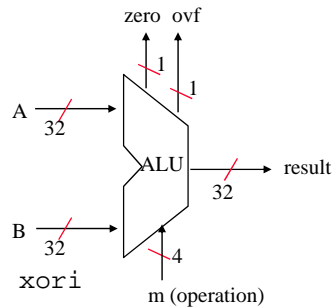
sub, subu, neg

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, andi, ori, xori, slti, sltiu
- zero extend – lbu, addiu, sltiu
- no overflow detected – addu, addiu, subu, multu, divu, sltiu, sltu

## Tailoring the ALU to the MIPS ISA

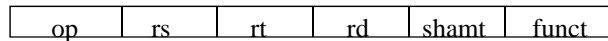
- Need to support the logic operation (and, nor, or, xor)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function, mux to choose the output
- Need to support the set-on-less-than instruction (slt)
  - Use subtraction to determine if  $(a - b) < 0$  (implies  $a < b$ )
  - Copy the sign bit into the low order bit of the result, set remaining result bits to 0
- Need to support test for equality (bne, beq)
  - Again use subtraction:  $(a - b) = 0$  implies  $a = b$
  - Additional logic to "nor" all result bits together
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

## Shift Operations

- Also need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll    $t2, $s0, 8    # $t2 = $s0 << 8 bits
```

```
srl    $t2, $s0, 8    # $t2 = $s0 >> 8 bits
```



- Notice that a 5-bit shamt field is enough to shift a 32-bit value  $2^5 - 1$  or 31 bit positions
- Such shifts are logical because they fill with zeros

## Shift Operations, con't

- An arithmetic shift (`sra`) maintain the arithmetic correctness of the shifted value (i.e., a number shifted right one bit should be  $\frac{1}{2}$  of its original value; a number shifted left should be 2 times its original value)
  - so `sra` uses the most significant bit (sign bit) as the bit shifted in
  - note that there is no need for a `sla` when using two's complement number representation

```
sra    $t2, $s0, 8    # $t2 = $s0 >> 8 bits
```

- The shift operation is implemented by hardware separate from the ALU
  - using a barrel shifter (which would takes lots of gates in discrete logic, but is pretty easy to implement in VLSI)

# Multiplication

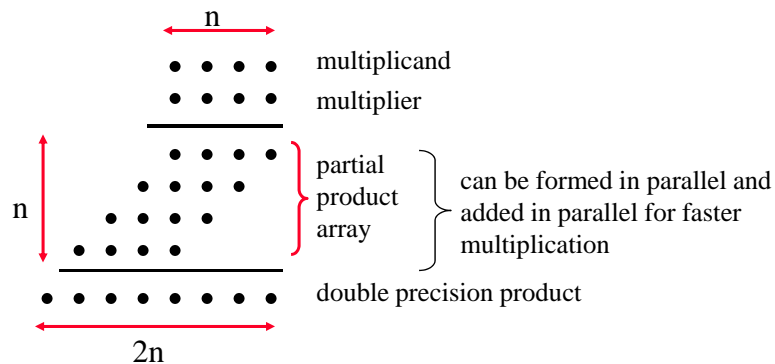
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on a gradeschool algorithm

$$\begin{array}{r} 0010 \text{ (multiplicand)} \\ \underline{\times 1011} \text{ (multiplier)} \end{array}$$

- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

# Multiply

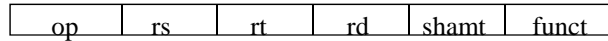
- Binary multiplication is just a *bunch* of right shifts and adds



# MIPS Multiply Instruction

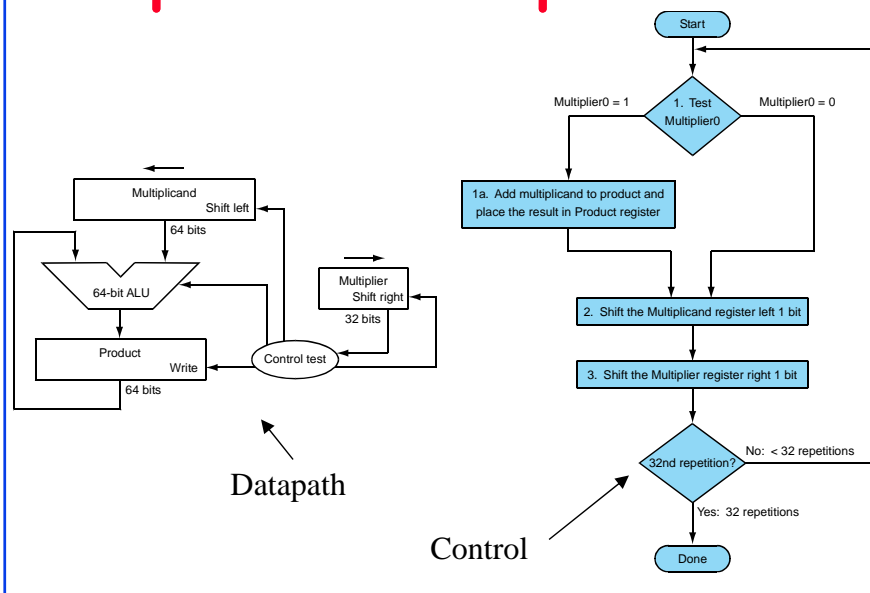
- Multiply produces a double precision product

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



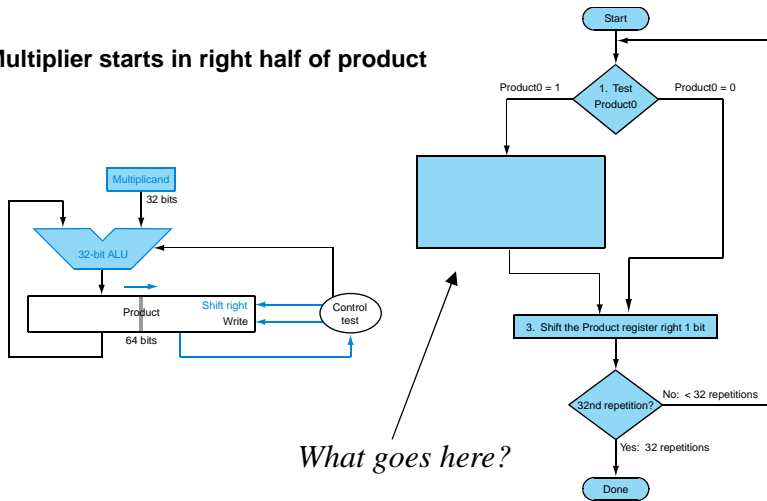
- Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file
- Multiplies are done by fast, dedicated hardware and are much more complex (and slower) than adders
  - Hardware dividers are even *more* complex and even slower; ditto for hardware square root

# Multiplication: Implementation



# Final Version

•Multiplier starts in right half of product

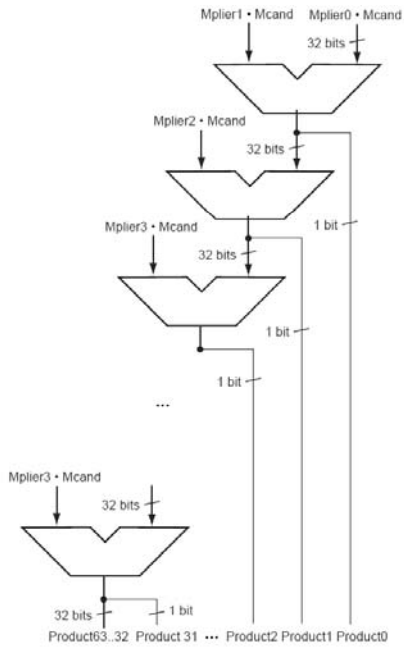


# Example

2 x 3 -> 0010x0011

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a. 1⇒Prod=Prod+Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a. 1⇒Prod=Prod+Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1. 0⇒no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1. 0⇒no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

# Faster Multiplication



# Division

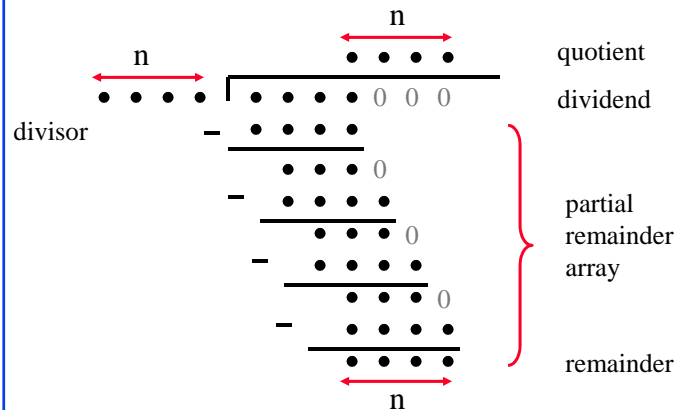
```

      1001      Quotient
      -----
Divisor 1000 |1001010  Dividend
      -1000
      -----
           10
           101
           1010
           -1000
           -----
              10  Remainder
  
```

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

## Division

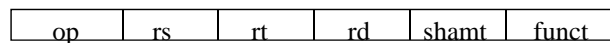
- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts



## MIPS Divide Instruction

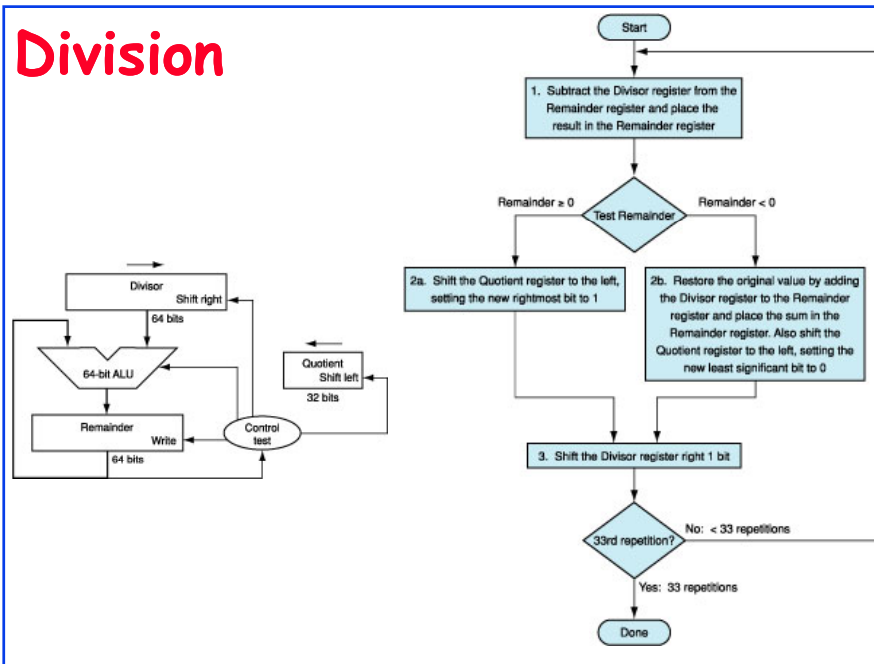
- Divide generates the remainder in `hi` and the quotient in `lo`

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



- Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and 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.

# Division

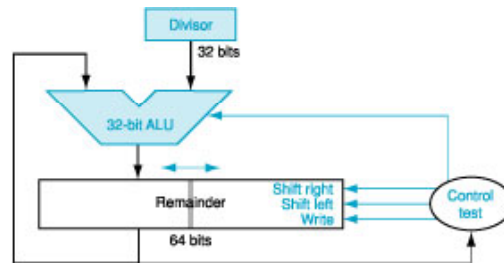


# Division: Example

Using 4-bit, divide 7 by 2, 0111 by 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem=Rem-Div	0000	0010 0000	1110 0111
	2b: Rem<0 ⇒ +Div, sll Q, Q0=0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem=Rem-Div	0000	0001 0000	1111 0111
	2b: Rem<0 ⇒ +Div, sll Q, Q0=0	0001	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1. Rem=Rem-Div	0000	0000 1000	1111 1111
	2b: Rem<0 ⇒ +Div, sll Q, Q0=0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1. Rem=Rem-Div	0000	0000 0100	0000 0011
	2a: Rem=>0 ⇒ sll Q, Q0=1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1. Rem=Rem-Div	0001	0000 0010	0000 0001
	2a: Rem=>0 ⇒ sll Q, Q0=1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

## Final Version



Can we implement faster division using parallel hardware?

## Signed Division

- We must set the sign for both quotient and the remainder
- $\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$
- Example all combinations of  $\pm 7 \div \pm 2$ 
  - $+7 \div +2$ : Quotient +3, Remainder +1:  $7 = 3 \times 2 + (+1)$
  - $-7 \div +2$ : Quotient -3,  
Remainder = Dividend - Quotient  $\times$  Divisor =  $-7 - (-3 \times 2) = -1$
- Rule: The Dividend and the Remainder must have the same signs
  - $+7 \div -2$ : Quotient -3, Remainder +1
  - $-7 \div -2$ : Quotient +3, Remainder -1

## Division in MIPS

- ❑ MIPS has two instructions for both signed and unsigned instructions: `div` and `divu`
- ❑ MIPS divide instructions ignore overflow
- ❑ MIPS software must check the divisor it is zero as well as overflow.

## Floating Point (a brief look)

- ❑ We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g.,  $3.15576 \times 10^9$
- ❑ Representation:
  - sign, exponent, significand:  $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- ❑ Scientific notation,
  - Normalized  $1.0 \times 10^{-9}$ , not normalized  $0.1 \times 10^{-9}$
  - Binary numbers in scientific notation  $1.0 \times 2^{-1}$
- ❑ Floating point: represent numbers in which the binary points is not fixed



# IEEE 754

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand
- Representation:
  - sign, exponent, significand:  $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- Overflow: the exponent is too large to be represented in the exponent field
- Underflow: negative exponent is too large to be represented in the exponent field

# IEEE 754

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit fraction
  - double ~~precision~~: 11 bit exponent, 52 bit fraction
  - $(-1)^s \times (1 + \text{Fraction}) \times 2^E$
  - Make the leading bit implicit → Significant 24 or 54 bits
  - $(-1)^s \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + \dots) \times 2^E$

## IEEE 754 floating-point standard

- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary:  $(-1)^{\text{sign}} \times (1 + \text{significand}) \times 2^{\text{exponent} - \text{bias}}$
- Example:
  - decimal:  $-.75 = -(\frac{1}{2} + \frac{1}{4})$
  - binary:  $-.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  
  - IEEE single precision:  
1 01111110 100000000000000000000000

## Floating point addition

- Add  $9.999 \times 10^1 + 1.610 \times 10^{-1}$  using only four decimal digits of the significant and two decimal digits of the exponent
- Step 1: Align the decimal point of the number that has the smaller exponent:  $1.610 \times 10^{-1} = 0.01610 \times 10^1$
- Step 2: Add the significands:  $9.999 + 0.016 = 10.015$ 
  - The sum is  $10.015 \times 10^1$
- Step 3: Normalize:  $1.0015 \times 10^2$
- Step 4: Round  $1.002 \times 10^2$

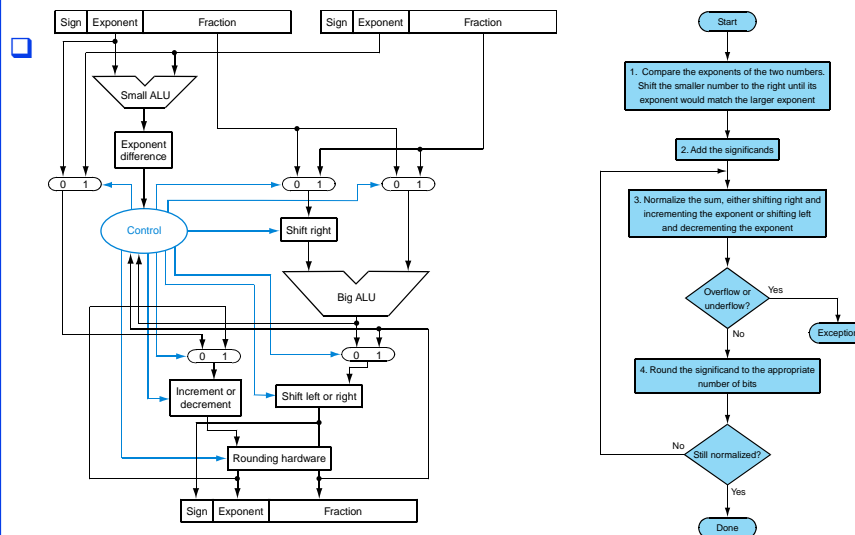
# Floating Point Addition

## □ Addition (and subtraction)

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

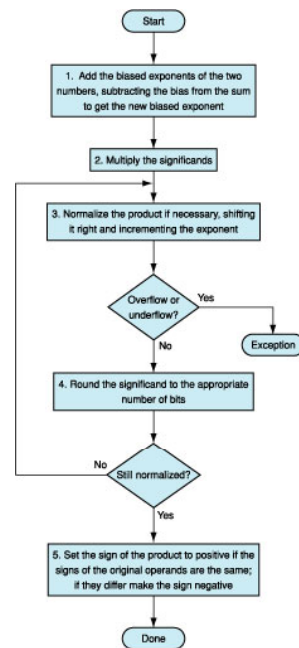
- Step 1: Restore the hidden bit in F1 and in F2
- Step 1: **Align** fractions by right shifting F2 by  $E1 - E2$  positions (assuming  $E1 \geq E2$ ) keeping track of (three of) the bits shifted out in a round bit, a guard bit, and a sticky bit
- Step 2: **Add** the resulting F2 to F1 to form F3
- Step 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  $E3$
  - If F1 and F2 have different signs  $\rightarrow F3$  may require *many* left shifts each time decrementing  $E3$
- Step 4: **Round** F3 and possibly **normalize** F3 again
- Step 5: Rehide the most significant bit of F3 before storing the result

# Floating point addition



# Multiplication

- $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- Step 1. Calculate the exponent  
 $10 + (-5) = 5$ ,  
 with biased exponent:  
 $10 + 127 + (-5) + 127 = 259$  too large  
 $10 + 127 + (-5) + 127 - 127 = 5 + 127$
- Step 2. Multiplication of significands:  
 $1.110 \times 9.200 = 10.21200 = 10.212 \times 10^5$
- Step 3. Normalize:  $1.0212 \times 10^6$   
 Check for overflow and underflow
- Step 4. Round:  $1.021 \times 10^6$
- Step 5. The sign of the product depends on the signs of operands



# MIPS Floating Point Instructions

- MIPS has a separate Floating Point Register File ( $\$f0, \$f1, \dots, \$f31$ ) (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 branch operations

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

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

- And double precision comparison operations

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

## Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

## Round and Guard Digits

- In IEEE 754 there are two extra bits on the right during intermediate additions: guard and round
- Add  $2.56 \times 10^0$  to  $2.34 \times 10^2$ , using only three significant decimal digits
  - Shift  $0.0256 \times 10^2$ , using guard digit for 5 and round digit for 6
  - The sum is 2.365
  - Rounded to 2.37
  - Without guard and round the sum = 2.36

## Summary



- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication)
  
- You may want to look back (Section 3.10 is great reading!)