CMSC 411 | Computer Architecture

Lecture 12: **Performing Division and Handling Floating Point Numbers**

Ergun Simsek

Overview

Previous Lecture

- Algorithms for multiplying unsigned numbers
- Booth’s algorithm for signed number multiplication

This Lecture

- Algorithms for dividing unsigned numbers
- Handling of sign while performing a division
- Hardware design for integer division
- Floating point number arithmetic
- Hardware design for floating point numbers

Dividing Unsigned Numbers: Humans

Let's divide a 7-bit Dividend with a 4-bit Divisor

```
Quotient              0001001
                     _________
Divisor 1000 | 1001010    Dividend
              -1000
              -----
                  1010
                 -1000
                 -----
                    10    Remainder
                          (or Modulo)
```

$(1)_2 < (1000)_2$
We place a zero (0)

$(10)_2 < (1000)_2$
We place a zero (0)

$(100)_2 < (1000)_2$
We place a zero (0)

$(1001)_2 >= (1000)_2$
We place a one (1)

.....

Dividing Unsigned Numbers: Computers

WARNING: This is not exactly happening in our MIPS
Here we are just trying to understand the main concept

Initialization

Remainder	Divisor
0000000	0001000

Paper and pencil example (unsigned):

```
Quotient          0001001
                 ___________
Divisor 1000    |1001010   Dividend
                –1000
                 -----
                    1010
                   –1000
                    -----
                      10        Remainder
                                (or Modulo)
```

$0000001 - 0001000 < 0$
We place a zero (0)

$0000010 - 0001000 < 0$
We place a zero (0)

$0000100 - 00010000 < 0$
We place a zero (0)

$0001001 - 0001000 > 0$
We place a one (1)

…..

3 versions of divide, successive refinement

Note that each time we place a 0 or 1, in fact, we also do a left shift on the quotient

Divide Hardware (version 1)

The 32-bit value of the Divisor starts in the left half of the 64-bit register

Divisor Register: 64-bit

Divisor	00000000000000000000000000000000

The Remainder register is initialized with the value of the Dividend

Remainder Register: 64-bit

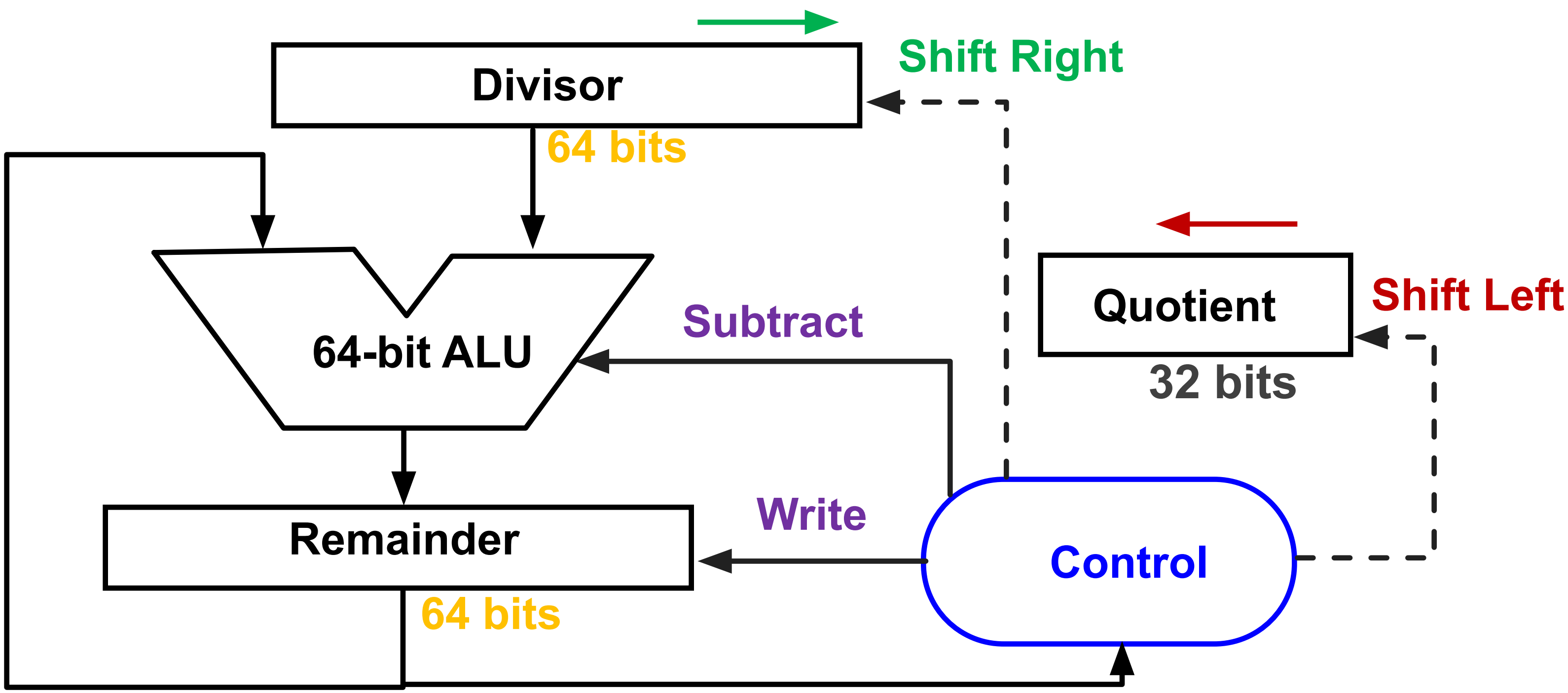
00000000000000000000000000000000	Dividend

Quotient Register: 32-bit

Quoziente

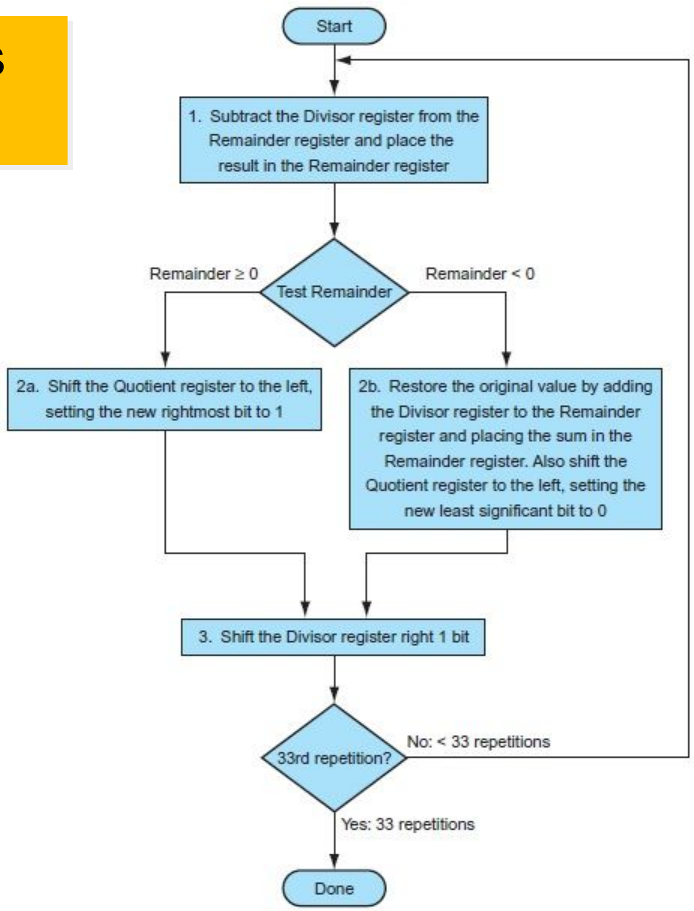
Divide Hardware (version 1)

- The Quotient is shifted left
- The Divisor is shifted to the right every step to align with the Dividend
- Control decides when to shift the Divisor and the Quotient registers and when to write new value into the Remainder register



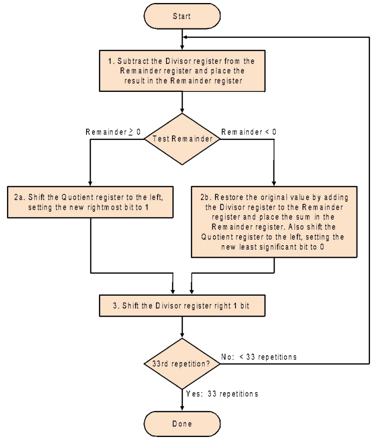
Divide Algorithm Version 1

Dividing two n-bit numbers needs n+1 steps to generate n-bit Quotient and Remainder



- If the Remainder is positive, a 1 is generated in the Quotient
- A negative Remainder indicates that the divisor did not go into the Dividend
- Shifting the Divisor in step 3 aligns the Divisor with the Dividend for next iteration
- We repeat for 33 times (You will understand why in the next slide)

An Example



Follow the division algorithm (version 1) to divide 7 by 2 using only 4-bit binary representation

We always need n+1 iterations to do n-bit division

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1				
2				
3				
4				
5				

What's wrong with the 1^{st} version?

In the 1^{st} version of divide hardware,

Half of the bits in Divisor is always 0

=> Half of 64-bit adder is wasted

=> Half of 64-bit divisor is wasted

Can we decrease the number of iterations to *n* from *n+1*?

Divide Hardware (version 2)

This time let's

- use a 32-bit register for the Divisor, and
- place the dividend in the right half of the Remainder

Divisor Register: 32-bit

Divisor

Remainder Register: 64-bit

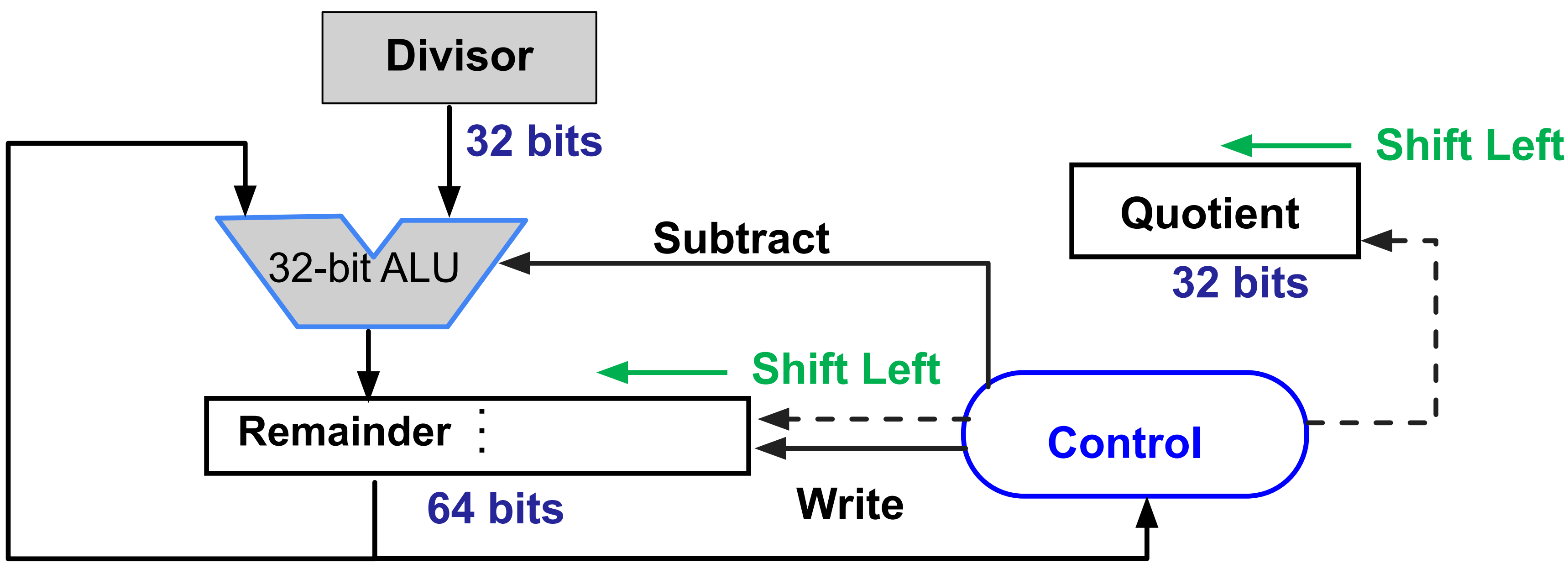
00000000000000000000000000000000 | Dividend

Quotient Register: 32-bit

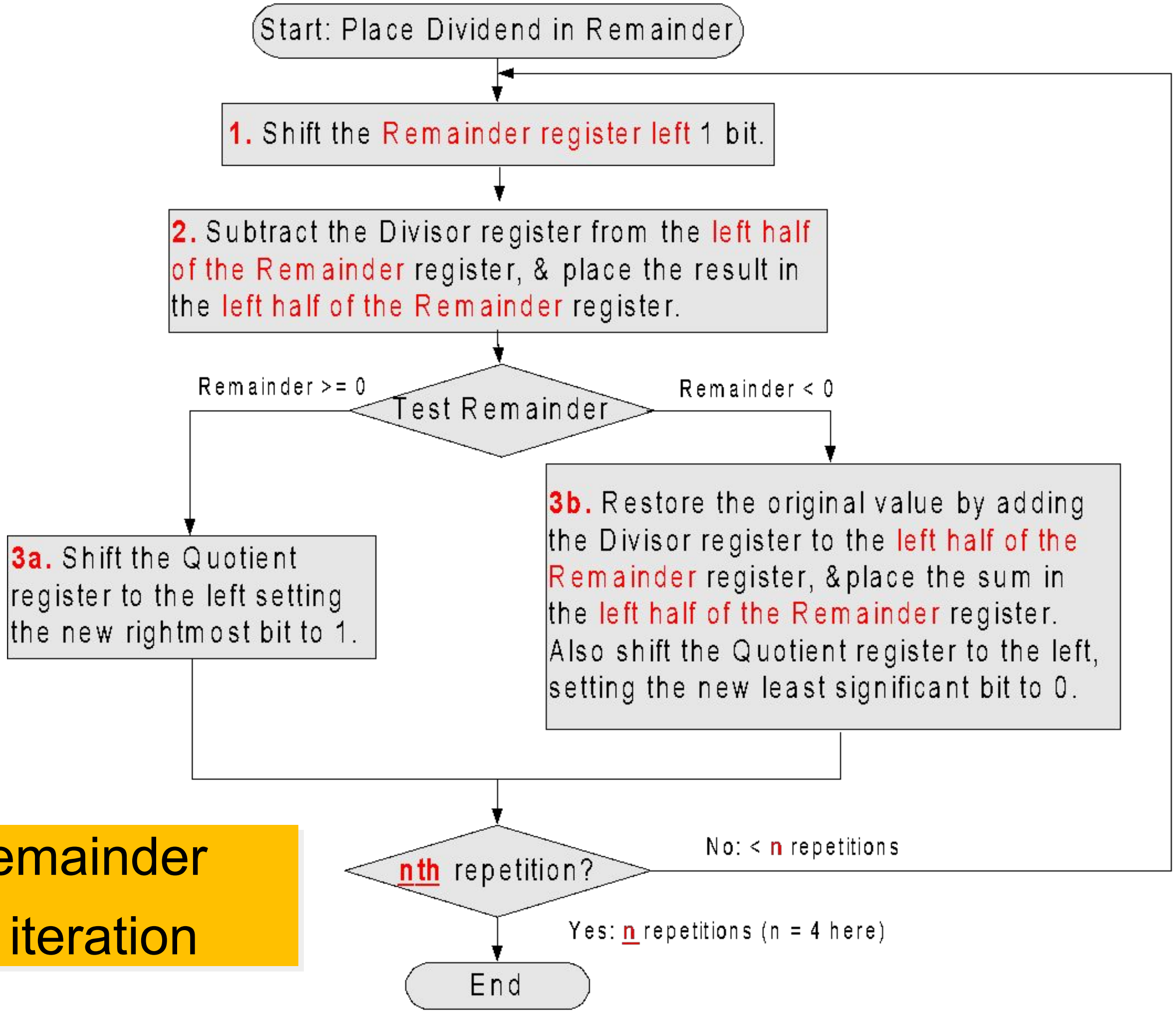
Quoziente

Divide Hardware (version 2)

- Divisor is not shifted
- Both the Remainder and Quotient are shifted to the left
- 1st step cannot produce a 1 in quotient bit (divide by zero) => switch order shift first and then subtract (to save 1 iteration)
- The most significant 32-bits would be used by the ALU as a result register



Divide Algorithm Version 2



Early shifting the remainder register saves one iteration

An Example

Follow the division algorithm (version 2) to divide 7 by 2 using only 4-bit binary representation

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010	0000 0111
1				
2				
3				
4				

What's wrong with the 2nd version?

- Remainder register wastes space
- And this space exactly matches size of Quotient
- Then why don't we combine Quotient register and Remainder register

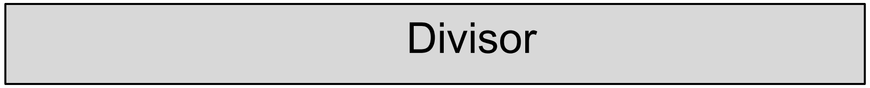
Divide Hardware (version 3)

This time let's

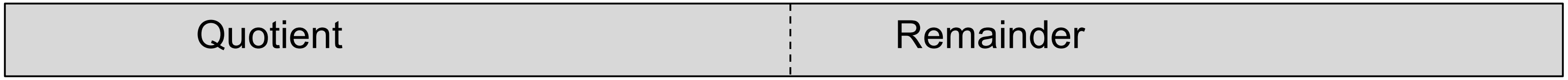
- place the Dividend in the **right** half of the Remainder
- place the Quotient in the **left** half of the Remainder

No Quotient Register

Divisor Register: 32-bit



Remainder Register: 64-bit

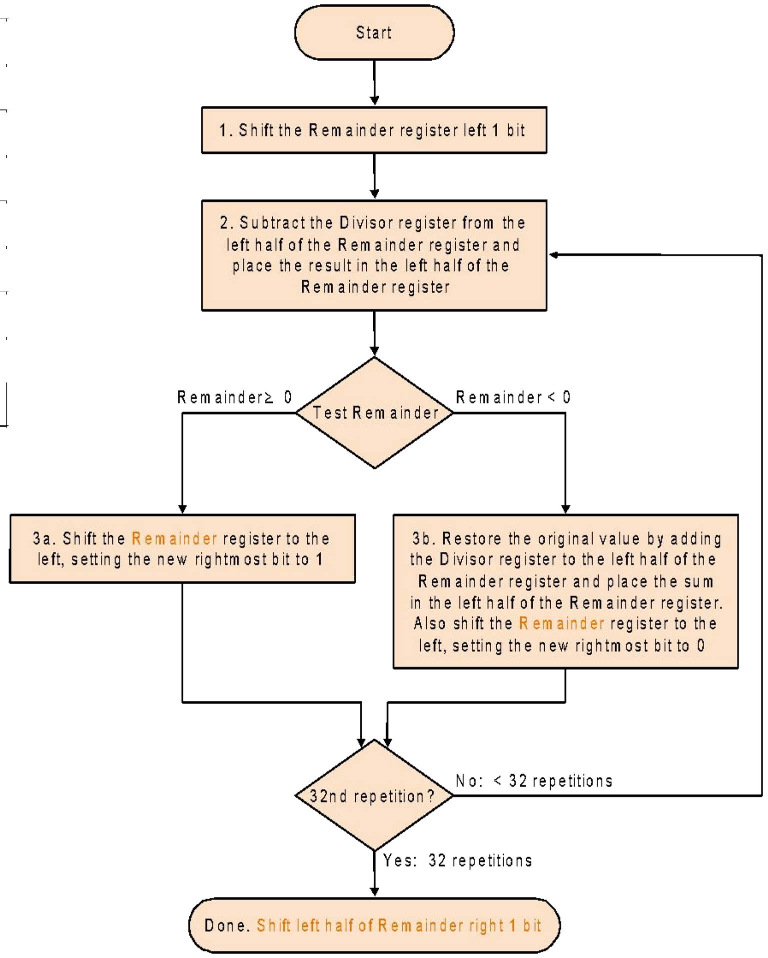


- ❑ The same number of shift operations would apply to the Remainder and the Quotient
- ❑ The Remainder needs to be corrected at the end
- ❑ The most significant 32-bits are still being used by ALU as a result register

Divide Algorithm Version 3

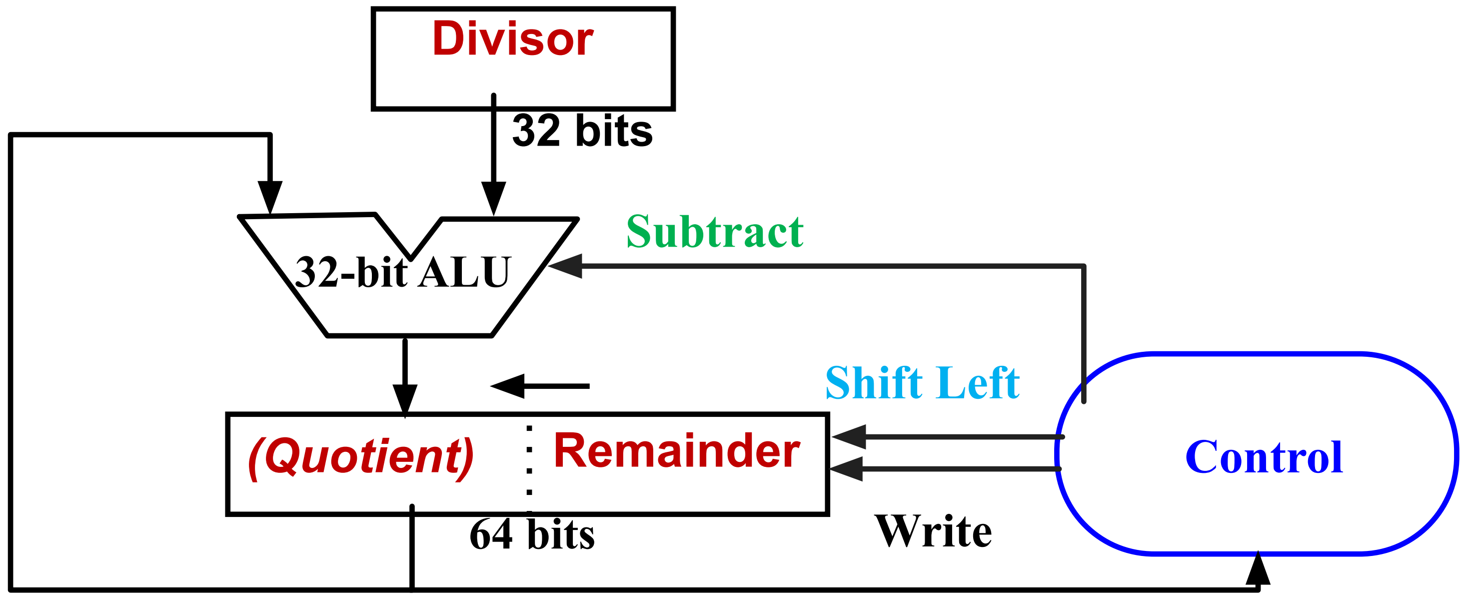
Dividing 7 by 2

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
1			
2			
3			
4			



Remainder would be shifted an extra time and need to be corrected at the end

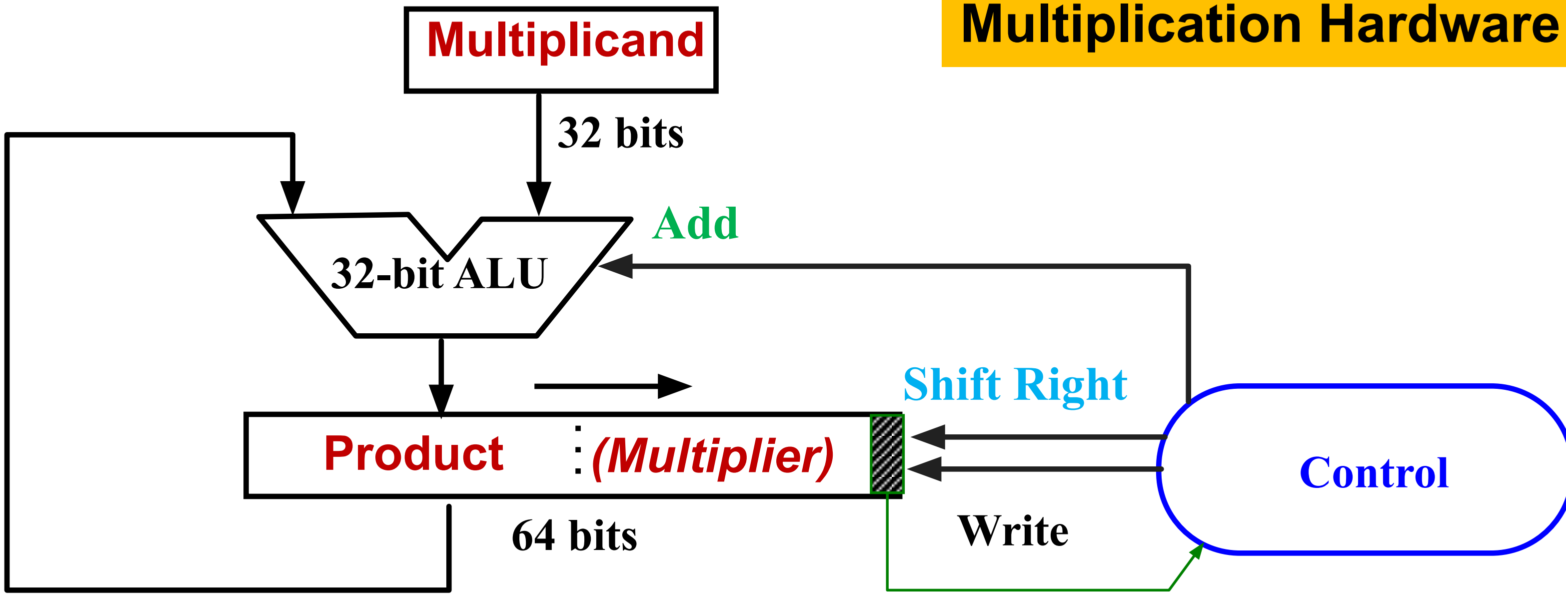
Division Hardware (Spot the Problem!)



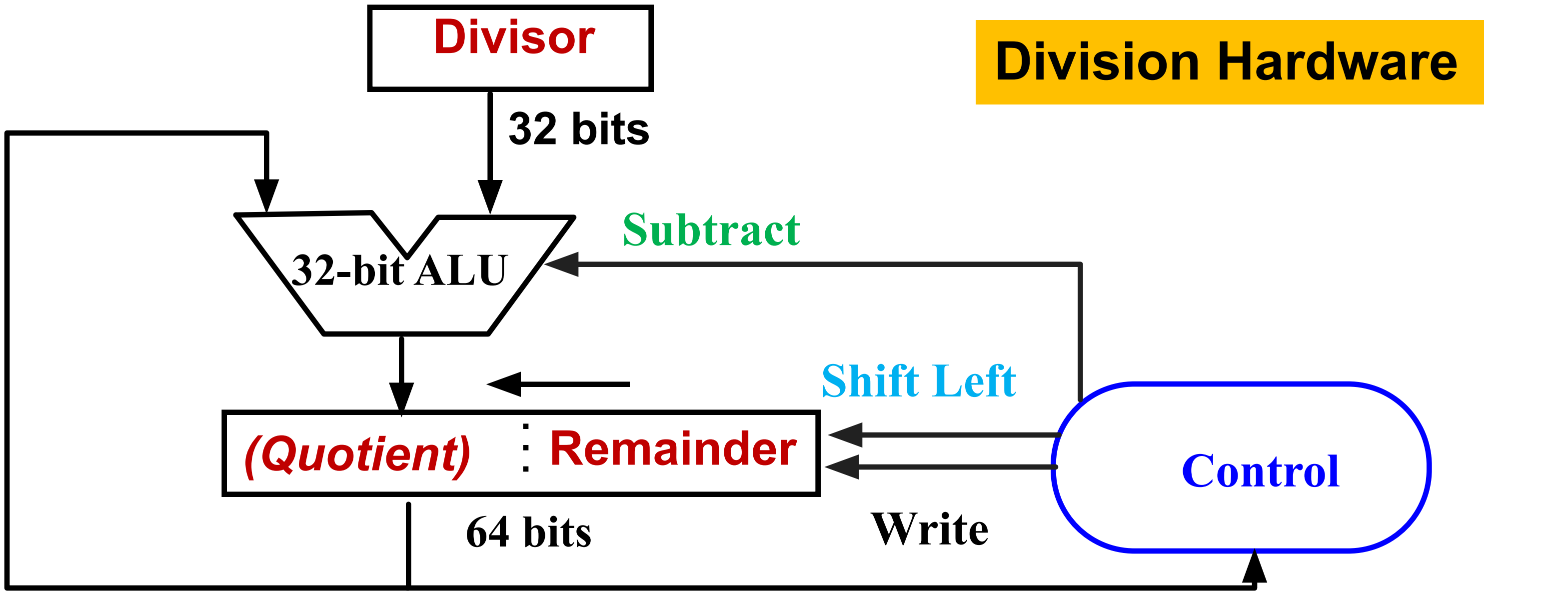
According to the drawing, Quotient should be kept in $HI and Remainder should be kept in $LO, but in MIPS it is reverse!

MIPS: $LO keeps the quotient
$HI keeps the remainder

Multiplication Hardware



Division Hardware



Dividing Signed Numbers

Simplest approach is to remember signs, make positive, and complement quotient and remainder if necessary

- Rule 1: Dividend and Remainder must have same sign
- Rule 2: Quotient negated if Divisor sign & Dividend sign are different

Examples:

Dividend = Quotient × Divisor + Remainder

7 ÷ 2 = 3, remainder = 1

–7 ÷ 2 = –3, remainder = –1

7 ÷ – 2 = –3, remainder = 1

–7 ÷ – 2 = 3, remainder = –1

According to the mathematicians, the remainder has to be a positive number!

Working with
Floating Point Numbers

Remember IEEE 754 Floating-Point Formats

Single-precision format

S	Exponent	Significand (a.k.a. Mantissa)
1 bit	8 bits	23 bits

Bias

$$v = -1^{S} \times 1.Significand \times 2^{Exponent-127}$$

These are 1s and 0s

This is a decimal!

Double-precision format

S	Exponent	Significand (a.k.a. Mantissa)
1 bit	11 bits	52 bits

$$v = -1^{S} \times 1.Significand \times 2^{Exponent-1023}$$

Floating Point Arithmetic

Floating point arithmetic differs from integer arithmetic

- both exponents and magnitudes of the operands must be handled

Subtraction/Addition (in 3 steps)

1. The exponents of the operands must be made equal
2. The fractions are then added or subtracted as appropriate, and
3. The result is normalized.

Floating Point Arithmetic

Example:

Perform the following addition: $(.101 \times 2^3 + .111 \times 2^4)_2$

Start by adjusting the smaller exponent to be equal to the larger exponent, and adjust the fraction accordingly.

$101 \times 2^3 = .010 \times 2^4$

note we lose $.001 \times 2^3$ *of precision in the process*

$(.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5$

Rounding to three significant digits,

$= .100 \times 2^5$

and we have lost another 0.001×2^4

in the rounding process.

Floating Point Addition

For addition (or subtraction) this translates into the following steps:

(1) Compute Ye - Xe *(getting ready to align)*

(2) Right shift Xm to form Xm 2 $^{(Xe - Ye)}$

(3) Compute Xm $2^{(Xe - Ye)}$ + Ym

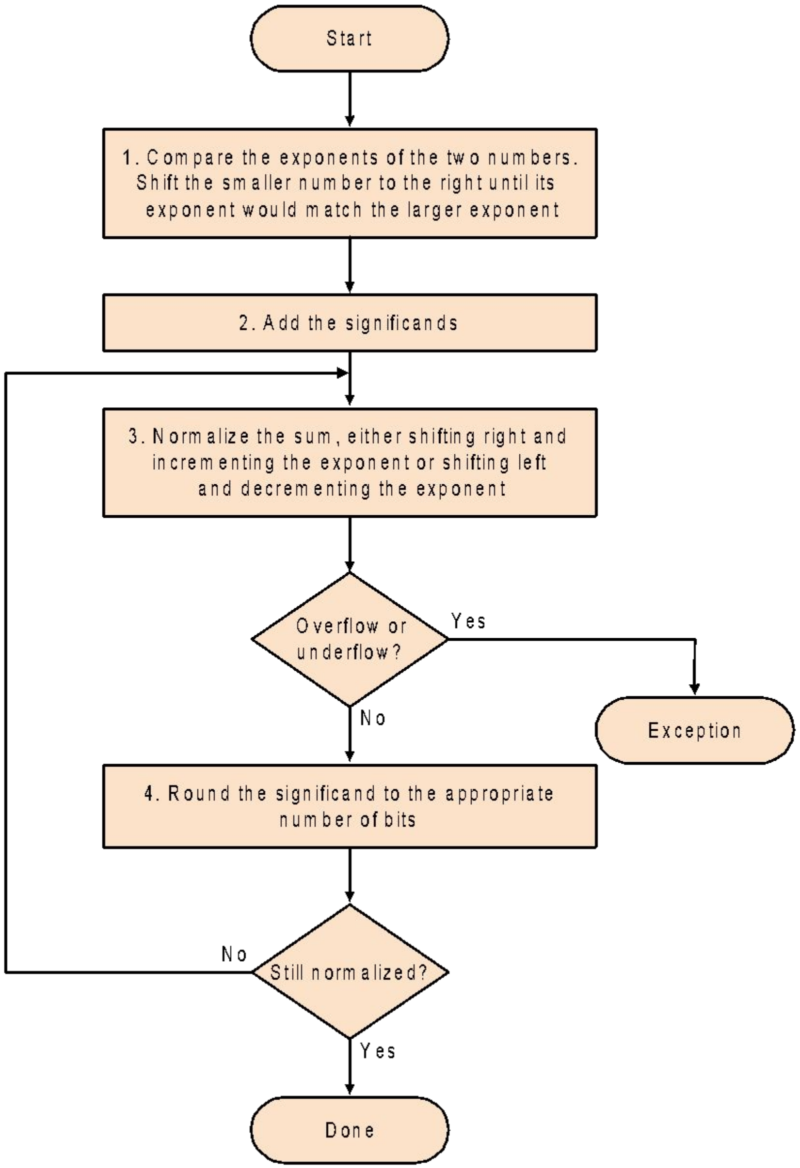
If representation demands normalization, then the following step:

(4) Left shift result, decrement result exponent

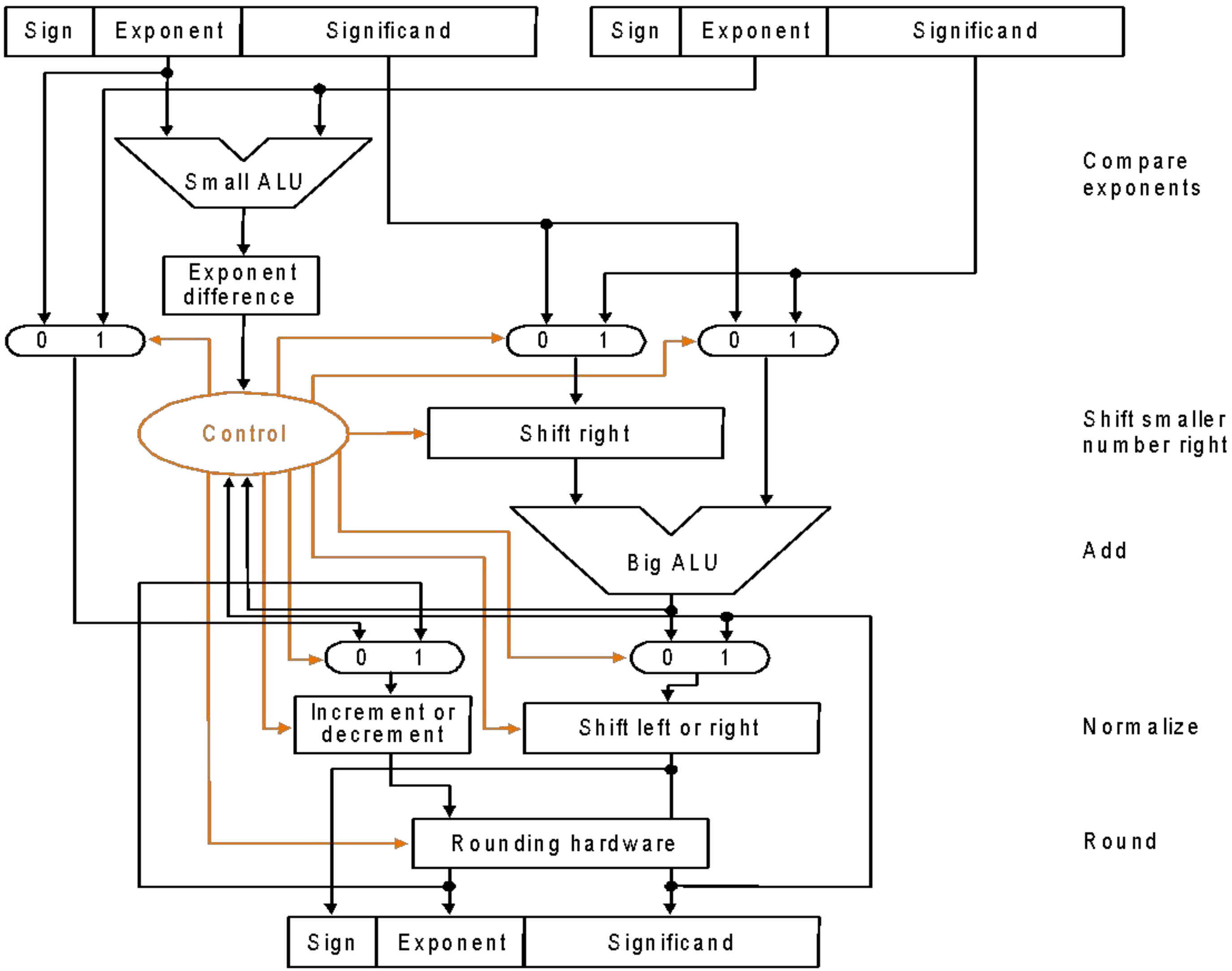
Right shift result, increment result

Continue until MSB of data is (Hidden bit)

(5) If result is 0 mantissa, may need to set exponent to zero by special step



Floating Addition Hardware



Floating Point Multiplication/Division

Performed in a manner similar to floating point add/subtraction

Main Difference: The sign, exponent, and fraction of the result can be computed separately.

- Same/opposite signs produce positive/negative results, respectively
- Exponent of result is obtained by adding/subtracting exponents for multiplication/division.
- Fractions are multiplied or divided according to the operation, and then normalized.

Floating Point Multiplication/Division

<u>Example:</u> Perform : $(+.110 \times 2^5) / (+.100 \times 2^4)_2$

The source operand signs are the same, which means that the result will have a positive sign.

We subtract exponents for division, and so the exponent of the result is 5 – 4 = 1.

We divide fractions, producing the result: 110/100 = 1.10.

Putting it all together,

$$(+.110 \times 2^5) / (+.100 \times 2^4) = (+1.10 \times 2^1).$$

After normalization, the final result is $(+.110 \times 2^2)$.

Computers do the almost same thing to do mult/division
The only difference is that they use **BIAS!**

$2^7 \times 2^{-3} = 2^4$

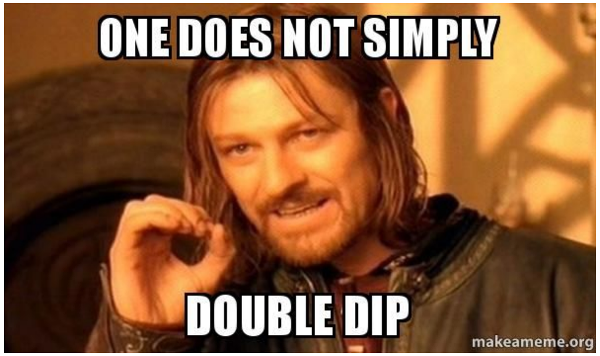
An Example

Let's say we want to use 8 as are our bias
So 2^{15} will represent 2^7
and 2^5 will represent 2^{-3}

If we multiply two representations, we will get

$$2^{15} \times 2^5 = 2^{20}$$

But according to our representation we were supposed to have 2^{12} as the representor of 2^4. What went wrong?



We double dipped the base!
We need to adjust it

Floating Point Multiplication

For addition (or subtraction) this translates into the following steps:

(1) Compute Ye + Xe *(adding exponents)*

(2) Correct the doubly biased exponent

(3) Multiply the significands

(4) Perform normalization

(4) Round the number to the specified size

(5) Calculate the sign of the product