Chapter 3
Arithmetic for Computers

3.1 Introduction

- **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: $0...2^n-1$
- **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?

3.2 Signed and Unsigned Numbers

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:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

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)**

    (lhu vs. lh)

3.3 Addition & Subtraction

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

```
  0111        0111        0110
+ 0110      - 0110      - 0101
------      ------      ------
```

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

```
  0111
+ 1010
------
```

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

```
  0111
+ 0001
------
  1000
```

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

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 ?

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`

- `add, addi, sub` : cause exception on overflow
- `addu, addiu, subu` : do not cause exception on overflow

note: `addiu` *still sign-extends!*
note: `sltu, sltiu` *for unsigned comparisons*

```
addu $t0, $t1, $t2
xor  $t3, $t1, $t2
slt  $t3, $t3, $zero
bne  $t3, $zero, No_overflow
xor  $t3, $t0, $t1
slt  $t3, $t3, $zero
bne $t3, $zero, Overflow
```

Different sign ➔ No Overflow

Same sign ➔ Possible

+ , + result –

- , - result +

For unsigned addition (`$t0= $t1+ $t2`), the test is

```
addu $t0, $t1, $t2
nor  $t3, $t1, $zero
sltu $t3, $t3, $t2
bne  $t3, $zero, Overflow
```

if (t1+t2 > 2^{32}-1) overflow

NOT $t1 = 2^{32}-$t1-1

2^{32}-$t1-1 < $t2 ➔ overflow

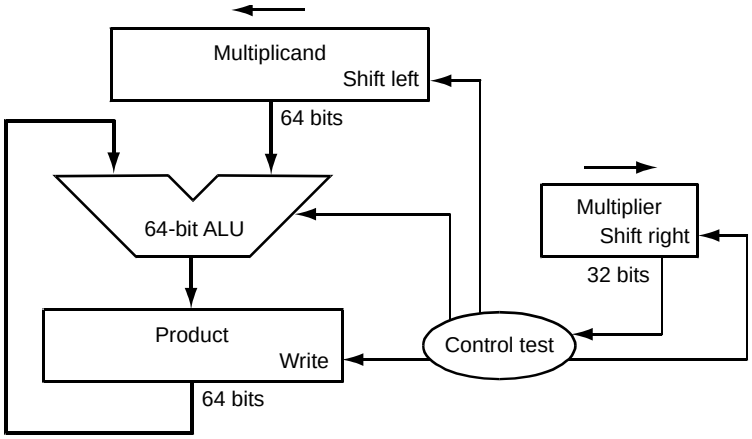
3.4 Multiplication

- 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

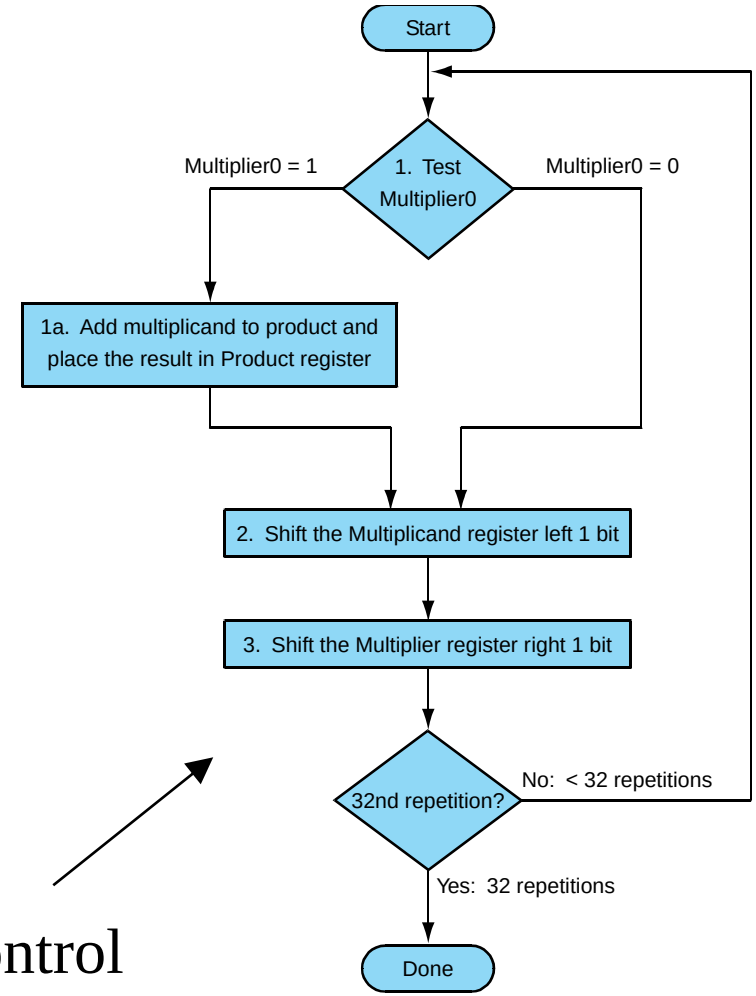
```
    0010   (multiplicand)
  x 1011   (multiplier)
```

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

Multiplication: Implementation



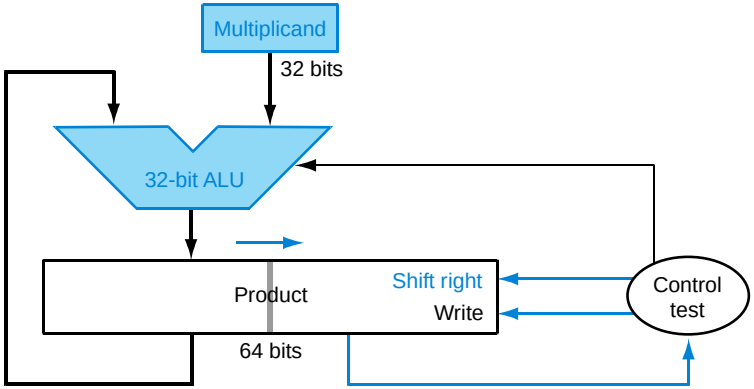
Datapath

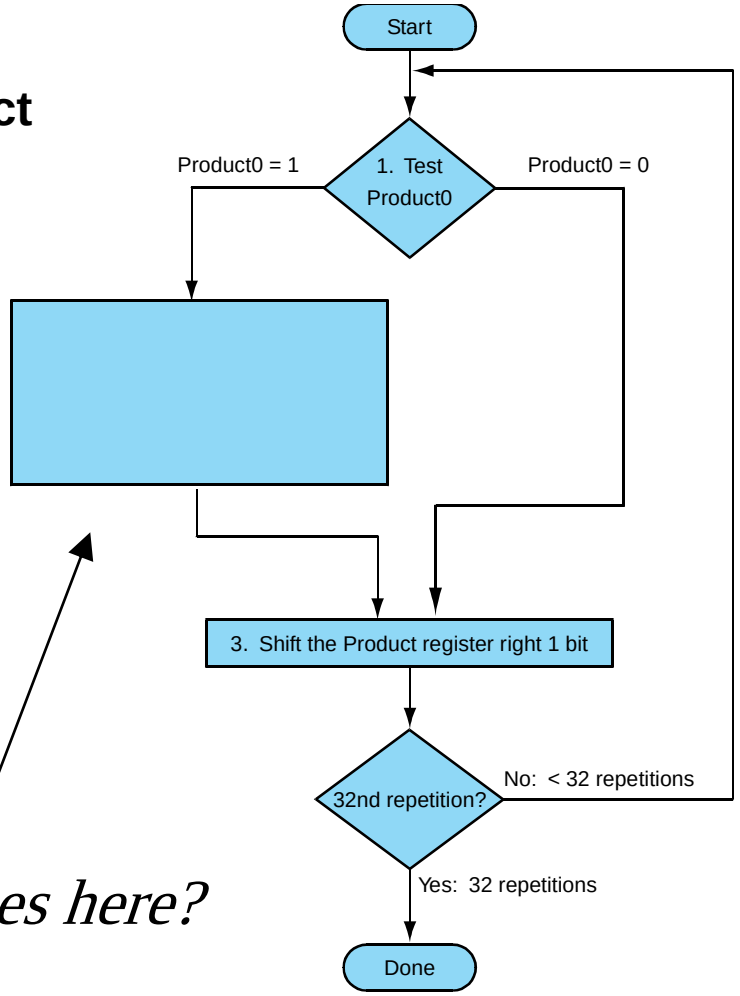


Control

Final Version

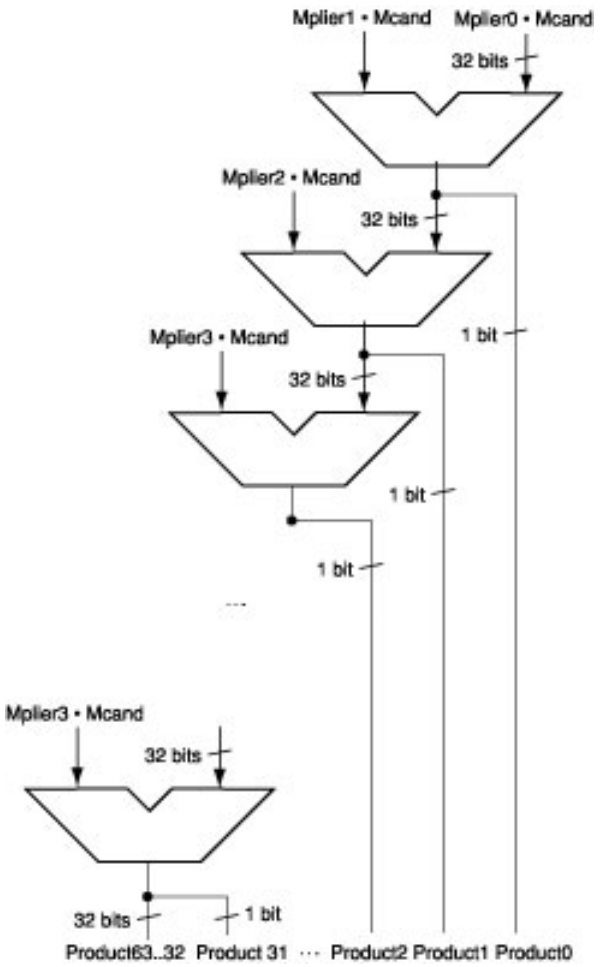
- **Multiplier starts in right half of product**





What goes here?

Faster Multiplication



3.6 Floating Point

- **Real Numbers:**

$3.14159265..._{ten}\,(\pi)$

$2.71828..._{ten}\,(e)$

0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

- **Scientific notation: single digit to the left of the decimal point (last two numbers).**

- **Normalized: scientific notation that has no leading 0s. for example:**

  normalized: $1.0_{ten} \times 10^{-9}$

  not normalized: $0.1_{ten} \times 10^{-8}$ **and** $10.0_{ten} \times 10^{-10}$

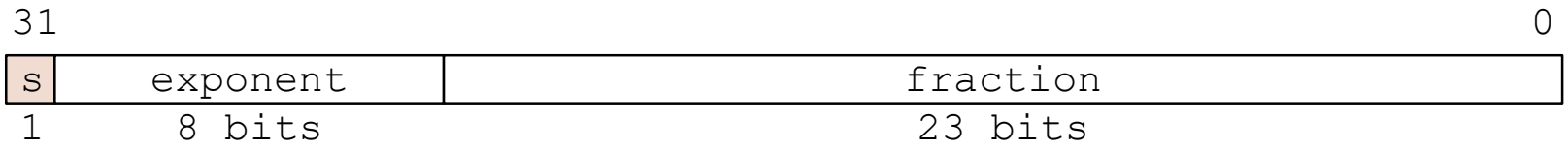
- **Binary numbers:**

  $1.0_{two} \times 2^{-1}$

  $1.xxxxxxxxx_{two} \times 2^{yyyy}$

Floating-Point Representation

- sign, fraction, exponent
- size of fraction and exponent
- size of fraction ➔ enhances the precision of the fraction.
  size of exponent ➔ increases the range
- Floating-point numbers are usually multiple of the size of a word.
- MIPS representation (called *sign and magnitude*):



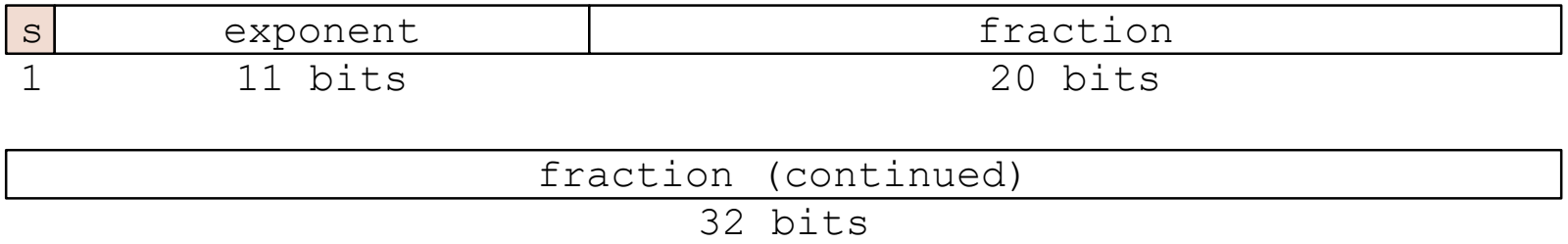
- In general, floating point numbers are generally of the form:

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

- Range: numbers almost as small as $2.0_{ten} \times 10^{-38}$ and as large as $2.0_{ten} \times 10^{38}$
- Overflow and Underflow
- To reduce chances of overflow and underflow : Double precision format

Continue

- **The representation of a double:**



- **Numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$**
- **IEEE 754 makes the leading 1 bit of normalized binary numbers implicit.**
- **Thus, 24 bits long in single precision(1+23) and 53 in double.**
- **Zero has no leading 1, it is given reserved exponent value 0.**
- **Thus $00 \ldots 00_{two}$ represents 0; the rest of the numbers uses:**

$$(-1)^{s} \times (1+\text{Fraction}) \times 2^{E}$$

Where, `Fraction` between 0 and 1

Continue

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1-254	anything	1-2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

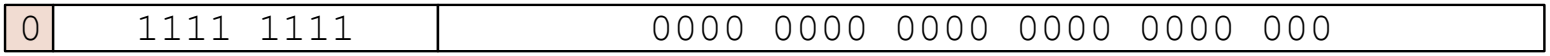
- **The above Table, shows the IEEE 754 encoding of floating-point numbers**
- **Special symbols for unusual events:**

  divide by zero ➔ ±∞ ➔ largest exponent

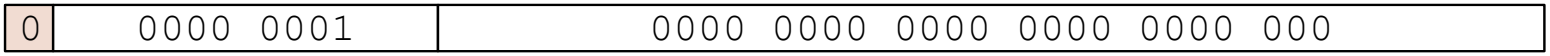
  0/0 or ∞-∞ ➔ NaN
- **For integer comparisons ➔ sign is the most significant bit**
- **Placing exponent before the significand also simplifies sorting of floating-point numbers using integer comparison.**

Continue

- **Negative exponent pose a challenge to simplified sorting. If we use 2's complement for negative exponents, then, $1.0_{two} \times 2^{-1}$ would be represented as:**



and the value $1.0_{two} \times 2^{+1}$ would look the smaller number:



- **Desirable notation: most negative exponent ➔ $00\ldots00_{two}$**
  most positive exponent ➔ $11\ldots11_{two}$
- **This is called biased notation. with the bias being subtracted from normal.**
- **IEEE 754 uses a bias of 127 for single precision, and 1023 for double.**

$$(-1)^{s} \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

Example: Floating-Point Representation

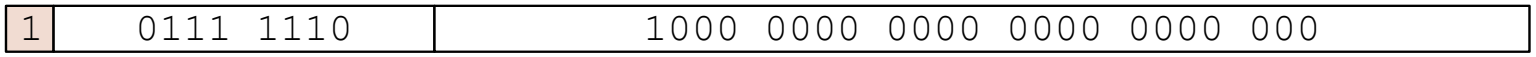
Example: Show the IEEE 754 binary representation of -0.75 in single and double.

-0.75_{ten} in scientific notation: $-0.11_{two} \times 2^{0}$

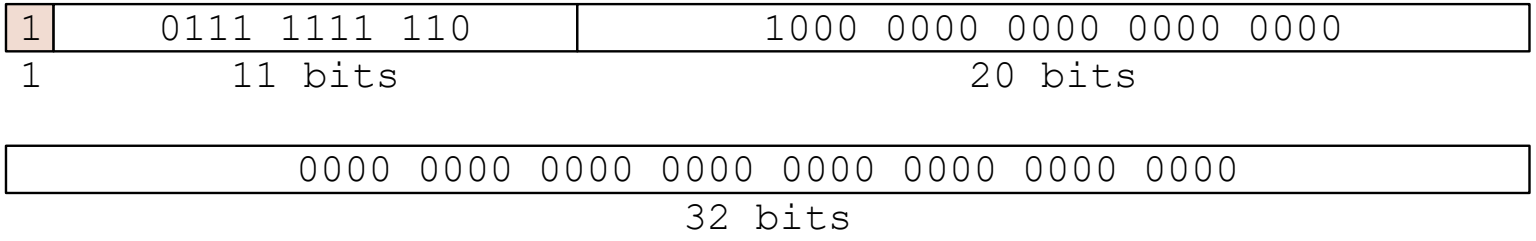
and in normalized scientific: $-1.1_{two} \times 2^{-1}$

$$(-1)^{S} \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

$$(-1)^{1} \times (1+.1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$



- **For double: $(-1)^{1} \times (1+.1000\ 0000\ 0000\ 0000\ 0000 \ldots 0000_{two}) \times 2^{(1022-1023)}$**



Example: Converting Binary to Decimal Floating Point

Example: What decimal number is represented by this single precision float?

1	1000 0001	0100 0000 0000 0000 0000 000

Sign bit=1 **exponent=129** **fraction=1 $\times 2^{-2}$=1/4, or 0.25**

using the basic equation:

$$(-1)^s \times (1 \times \text{Fraction}) \times 2^{(\text{Exponent-Bias})} = (-1) \times (1+0.25) \times 2^{(129-127)}$$

$$= -1 \times 1.25 \times 2^2$$

$$= -1.25 \times 4$$

$$= -5.0$$

Floating-Point Addition

Algorithm (assume 4 decimal digits of the significand and two for the exponent)

$$9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$$

Step1: align the number that has the smaller exponent

$$1.610_{ten} \times 10^{-1} = 0.016 \times 10^{1}$$

Step2: add the significands:

$$\begin{array}{r} 9.999_{ten} \\ 0.016_{ten} \\ \hline 10.015_{ten} \end{array}$$

Step3: normalize the sum: $10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}$

Step4: Round: the number $1.0015_{ten} \times 10^{2}$ is rounded to $1.002_{ten} \times 10^{2}$

Continue

- **In Step3 we normalize the results, forcing a check for overflow and underflow.**

- **All zero bits in the exponent is reserved for representing zero. Also, the pattern of all one bits in the exponent is reserved.**

- **Thus, for single precision, the maximum exponent is 127, and the minimum is -126. The limits for double are 1023 and -1022.**

Example: Decimal Floating-Point Addition

Add the numbers: 0.5_{ten} and -0.4375_{ten}

$0.5_{ten} = 1.000_{two} \times 2^{-1}$

$-0.4375_{ten} = -1.110_{two} \times 2^{-2}$

Step1: $-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$

Step2: $1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$

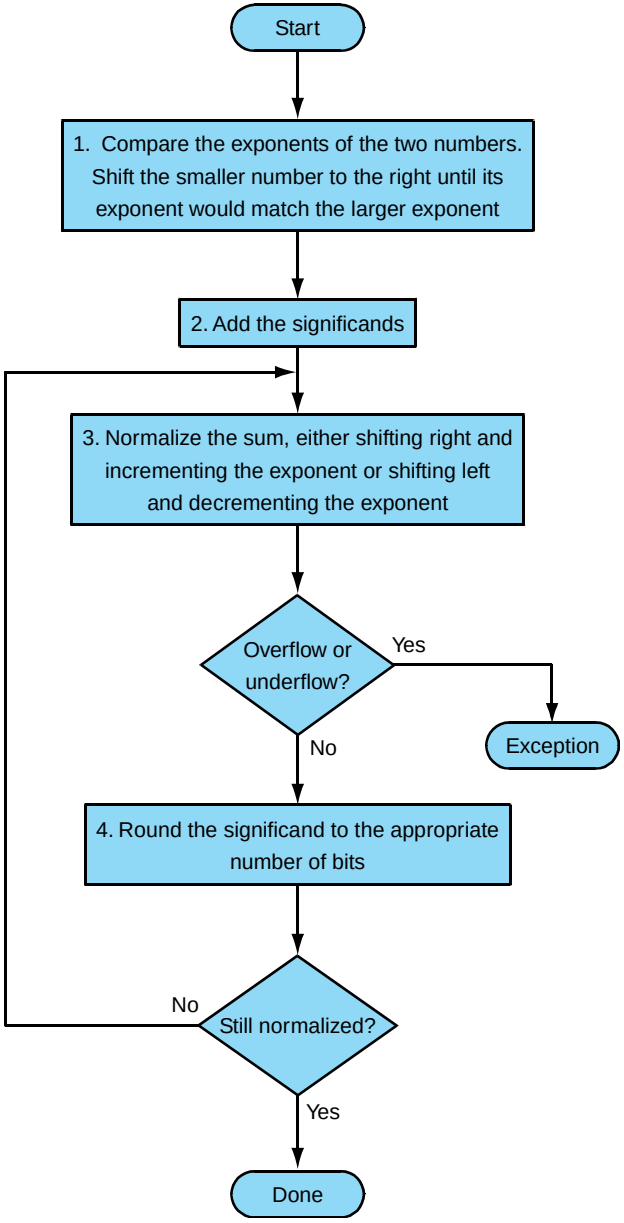
Step3: Normalize the sum $0.001_{two} \times 2^{-1} = 1.000_{two} \times 2^{-4}$

Since $127 \geq -4 \geq -126$ ➔ no overflow or underflow

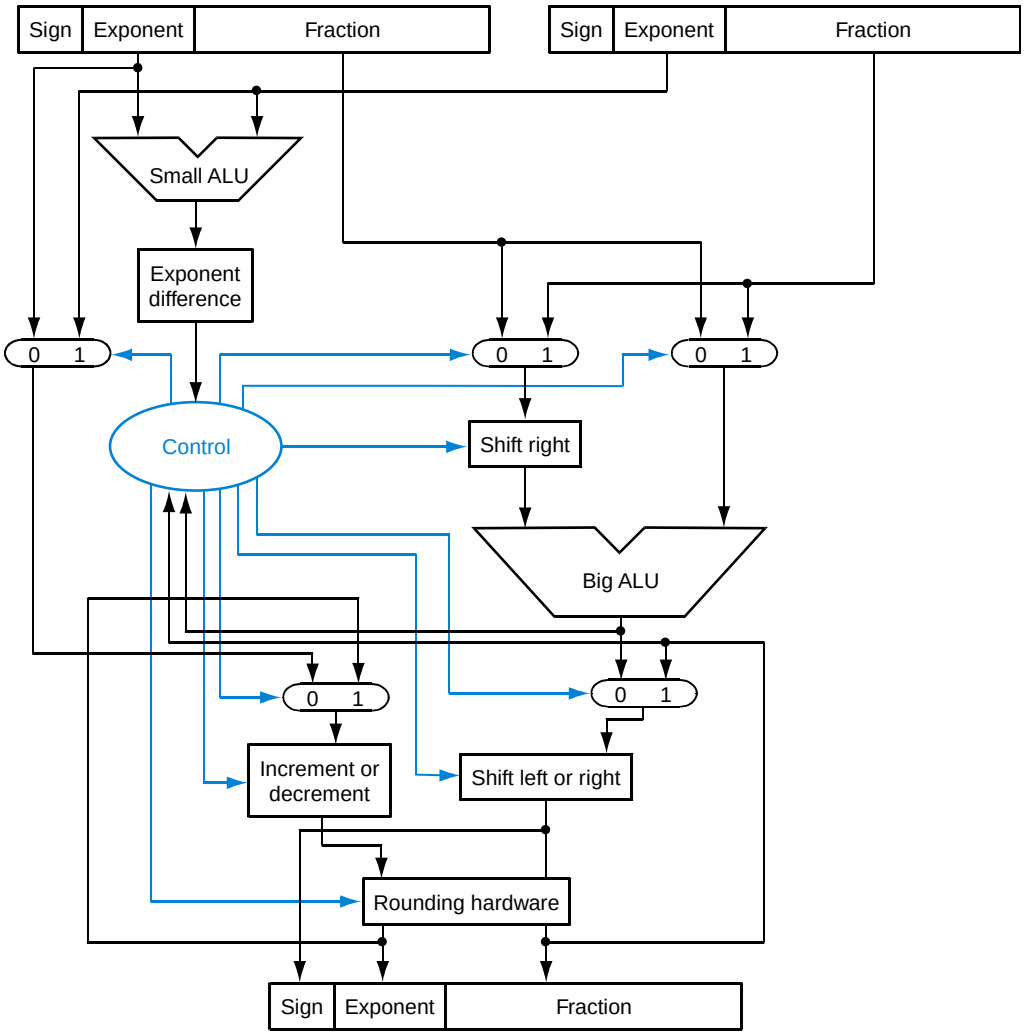
Step4: Round the sum: already fits $1.000_{two} \times 2^{-4}$

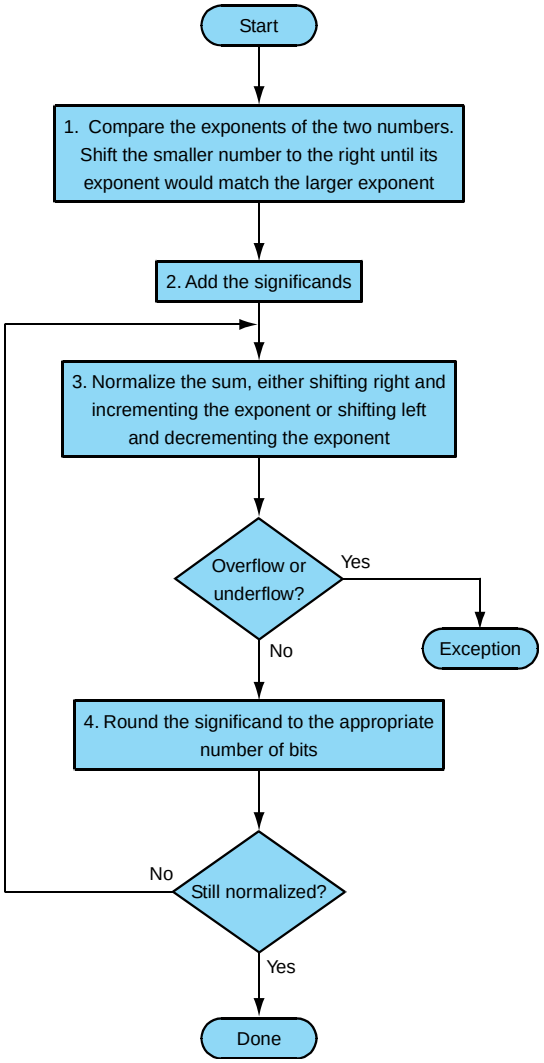
$1.000_{two} \times 2^{-4} = 1/2^{4}_{ten} = 1/16_{ten} = 0.0625 = 0.5 + (-0.4375)$ √

Continue


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Floating point addition





Floating-Point Multiplication

Example: $1.11_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$

Assume 4 digits for significand and 2 digits for exponent.

Step 1. Add biased exponents: (10+127)+(-5+127)-127 = 132

Step 2. Multiply the significands

$$
\begin{array}{r}
1.110_{ten} \\
\times\ 9.200_{ten} \\
\hline
0000 \\
0000 \\
2220 \\
9990 \\
\hline
10212000_{ten}
\end{array}
$$

The product = 10.212000_{ten} ➔ 10.212×10^{5}

Step 3. Normalize the product:

$$10.212_{ten} \times 10^{5} = 1.0212_{ten} \times 10^{6}$$

Continue

Step 4: Round the number $1.0212_{ten} \times 10^6$

$1.021_{ten} \times 10^6$

Step 5: Set the sign of the product:

$+1.021_{ten} \times 10^6$

Example: Decimal Floating-Point Multiplication

Lets multiplying 0.5 by -0.4375

In binary: $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$

Step 1: (-1+127) + (-2+127) - 127 = 124

Step 2:

$$
\begin{array}{r}
1.000_{two} \\
\times\ 1.110_{two} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{two}
\end{array}
$$

The product: $1.110000_{two} \times 2^{-3}$ ➔ $1.110_{two} \times 2^{-3}$

Continue

Step 3: It is already normalized, and since $127 \geq -3 \geq -126$ no Overflow or underflow.

Step 4: Rounding the product:

$1.110_{two} \times 2^{-3}$

Step 5: $-1.110_{two} \times 2^{-3}$

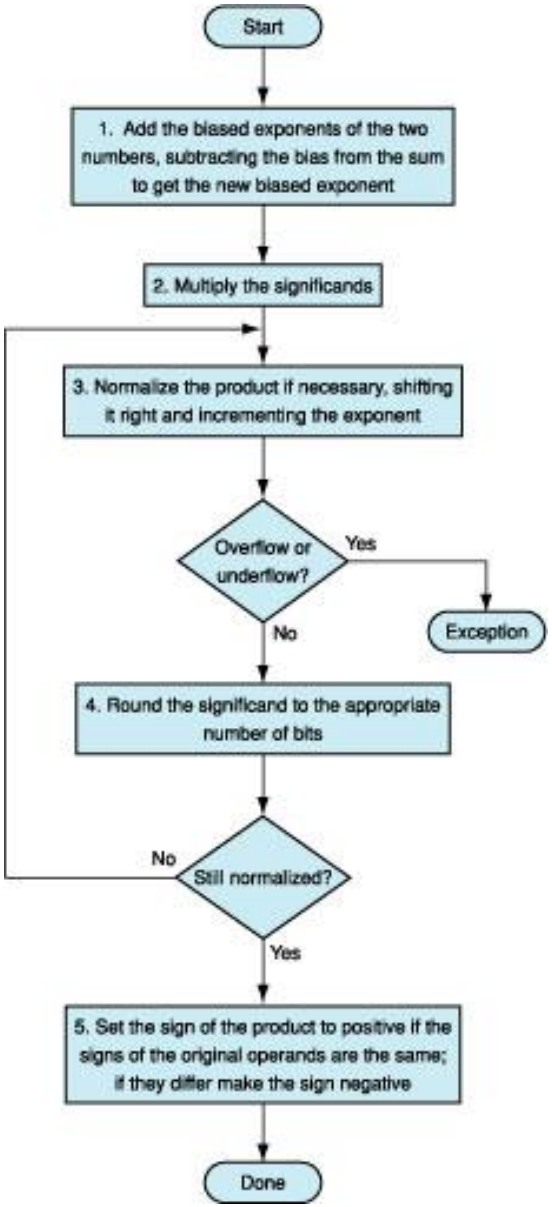
Converting to decimal to check our results:

$-1.110_{two} \times 2^{-3} = -0.00111_{two}$

$= -7/2^{5}_{ten} = -7/32_{ten} = -0.21875_{ten}$

Indeed: $0.5 \times -0.4375_{ten} = -0.21875_{ten}$

Continue


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Floating-Point Instructions in MIPS

- **MIPS Supports the IEEE 754 single precision and double with these instructions:**
  - **Floating-point addition (`add.s`) (`add.d`)**
  - **Floating-point subtractions (`sub.s`) (`sub.d`)**
  - **Floating-point multiplication (`mul.s`) (`mul.d`)**
  - **Floating-point division (`div.s`) (`div.d`)**
  - **Floating-point comparison (`c.x.s`) (`c.x.d`) where x may be (`eq`) (`neq`) (`lt`) (`le`) (`gt`) (`ge`)**
  - **Floating-point branch, true (`bclt`) and false (`bclf`)**
- **Separate floating-point registers $f0, $f1, … , $f31**
- **Example: load two number from memory, add them and then store the sum:**

```
lwc1        $f4, x($sp)
lwc1        $f6, y($sp)
add.s       $f2, $f4, $f6
swc1        $f2, z($sp)
```

Example: Compiling a Floating-Point C program:

Lets converts a temperature in Fahrenheit to Celsius:

```
    float f2c(float fahr)
    {
      return((5.0//9.0)*(fahr – 32.0));
    }
```

```
f2c:
        lwc1    $f16, const5($gp)
        lwc1    $f18, const9($gp)
        div.s   $f16, $f16, $f18        # $f16 = 5.0/9.0
        lwc1    $f18, const32($gp)      # $f18 = 32.0
        sub.s   $f18, $f12, $f18        # $f18 = fahr – 32.0
        mul.s   $f0, $f16, $f18         # $f0 =(5/9)*(fahr - 32.0)
        jr      $ra
```

Example: Compiling Floating-Point C Procedure with 2-Dimensional Matrices into MIPS

Let's perform matrix multiply of: X = X + Y×Z

```
void mm (double x[][], double y[][], double z[][])
{
   int i, j, k;

   for(i=0; i!=32; i=i+1)
   for(j=0; j!=32; j=j+1)
   for(k=0; k!=32; k=k+1)
       x[i][j]=x[i][j] + y[i][k]*z[k][j];
}
```

Continue

```
mm:
        li      $t1, 32
        li      $s0, 0
L1:     li      $s1, 0
L2:     li      $s2, 0

        sll     $t2, $s0, 5
        addu    $t2, $t2, $s1
        sll     $t2, $t2, 3
        addu    $t2, $a0, $t2
        l.d     $f4, 0($t2)

L3:     sll     $t0, $s2, 5
        addu    $t0, $t0, $s1
        sll     $t0, $t0, 3
        addu    $t0, $a2, $t0
        l.d     $f16, 0($t0)

        sll     $t0, $s0, 5
        addu    $t0, $t0, $s2
        sll     $t0, $t0, 3
        addu    $t0, $a1, $t0
        l.d     $f18, 0($t0)

        mul.d   $f16, $f18, $f16
        add.d   $f4, $f4, $f16
        addiu   $s2, $s2, 1
        bne     $s2, $t1, L3
        s.d     $f4, 0($t2)

        addiu   $s1, $s1, 1
        bne     $s1, $t1, L2
        addiu   $s0, $s0, 1
        bne     $s0, $t1, L1
        ...
```