FLOATING POINT

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

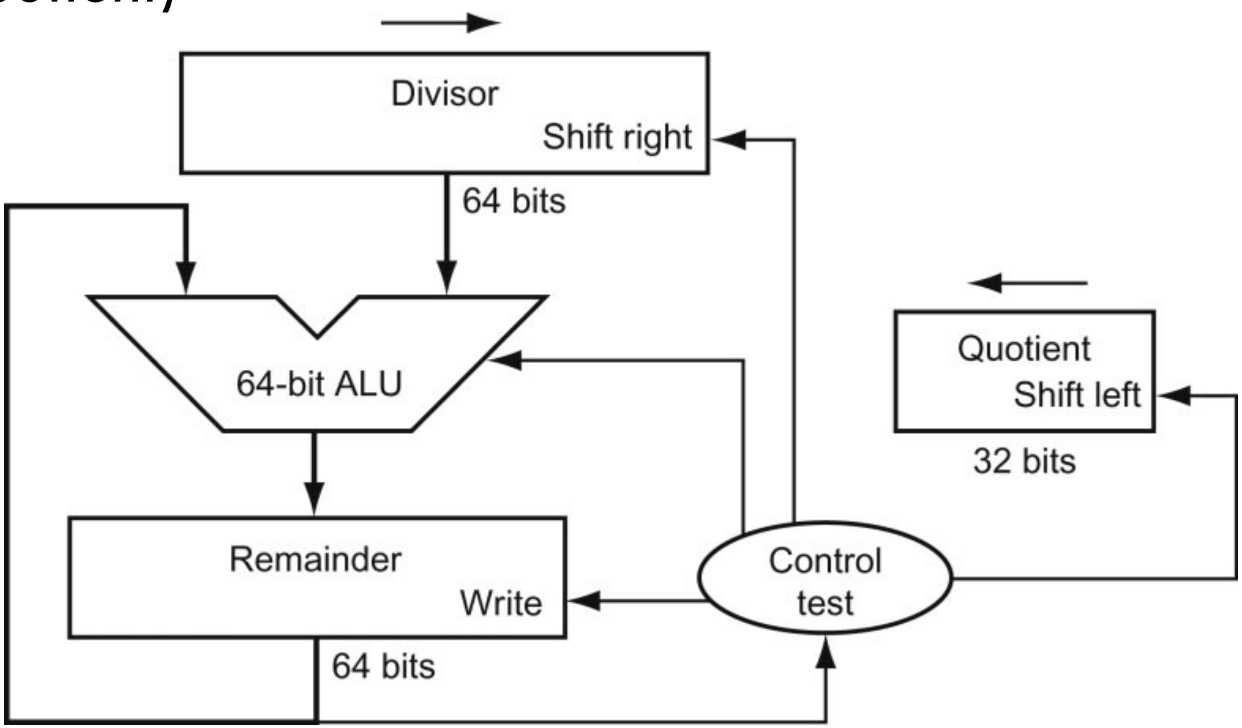
University of Utah

Overview

- Notes
  - Homework 5 is due tonight
    - Do not try to modify the given pseudocodes
  - No class on Tuesday Feb 19th
  - Midterm on Thursday Feb 21st
    - In class, including today's lecture
- This lecture
  - Floating point

Hardware for Division (pp. 189-193)

- A comparison requires a subtract; the sign of the result is examined; if the result is negative, the divisor must be added back
- Similar to multiply, results are placed in Hi (remainder) and Lo (quotient)



MIPS Instructions for Division

- MIPS ignores overflow in division
- Signed division (div)

```
div     $s2, $s3
```

computes the division and stores it in two “internal” registers that can be referred to as hi and lo

```
mfhi    $s0
mflo    $s1
```

mfhi $s0 moves the remainder into $s0

mflo $s1 moves the quotient into $s1

- Unsigned division (divu)

```
divu    $s2, $s3

mfhi    $s0
mflo    $s1
```

Signed Division

- Simplest solution: convert to positive and adjust sign later
- Note that multiple solutions exist for the equation
  - Dividend = Quotient x Divisor + Remainder

+7	div	+2	Quo =	Rem =
-7	div	+2	Quo =	Rem =
+7	div	-2	Quo =	Rem =
-7	div	-2	Quo =	Rem =

Signed Division

- Simplest solution: convert to positive and adjust sign later
- Note that multiple solutions exist for the equation
  - Dividend = Quotient x Divisor + Remainder

+7 div +2	Quo = +3	Rem = +1
-7 div +2	Quo = -3	Rem = -1
+7 div -2	Quo = -3	Rem = +1
-7 div -2	Quo = +3	Rem = -1

Signed Division

- Simplest solution: convert to positive and adjust sign later
- Note that multiple solutions exist for the equation
  - Dividend = Quotient x Divisor + Remainder

+7	div	+2	Quo = +3	Rem = +1
-7	div	+2	Quo = -3	Rem = -1
+7	div	-2	Quo = -3	Rem = +1
-7	div	-2	Quo = +3	Rem = -1

- Convention
  - Dividend and remainder have the same sign
  - Quotient is negative if signs disagree
  - These rules fulfil the equation above

Floating Point Numbers

- Example: how to represent 32.5
  - weights smaller than 1

100000.100_{two}

- Normalized scientific notation: leave a single non-zero digit to the left of the point
  - 3.25×10^1

$$1.00000100_{two} * 2^5 = (1*2^0 + 1*2^{-6}) * 2^5$$

- The IEEE 754 standard

Sign-Magnitude Representation

- Since we are only representing normalized numbers, we are guaranteed that the number is of the form 1.xxxx.

- Every 32-bit number has three fields: sign (S), exponent (E), and fraction (F)
  - value = $(-1)^{S} \times (1+F) \times 2^{E-127}$

1 bit	8 bits	23 bits
S	E	F

- Example: 32.5
  - $32.5 = 100000.100 = 1.00000100_{two} \times 2^{5}$

Sign-Magnitude Representation

- Since we are only representing normalized numbers, we are guaranteed that the number is of the form 1.xxxx.

- Every 32-bit number has three fields: sign (S), exponent (E), and fraction (F)
  - value = $(-1)^{S} \times (1+F) \times 2^{E-127}$

1 bit	8 bits	23 bits
S	E	F

- Example: 32.5
  - $32.5 = 100000.100 = 1.00000100_{two} \times 2^{5}$

0	10000100	00000100000000000000000

Sign-Magnitude Representation

- Since we are only representing normalized numbers, we are guaranteed that the number is of the form 1.xxxx.

- Every 32-bit number has three fields: sign (S), exponent (E), and fraction (F)
  - value = $(-1)^{S} \times (1+F) \times 2^{E-127}$

1 bit	8 bits	23 bits
S	E	F

- Using more bits
  - Increase E bits to represent a wider range of numbers
  - Increase F bits to represent more precision
- Double precision format with 64 bits

1 bit	11 bits	52 bits
S	E	F

Single Precision Floating Point

1 bit	8 bits	23 bits
S	E	F

- Largest number that can be represented

- Smallest number that can be represented

Single Precision Floating Point

1 bit	8 bits	23 bits
S	E	F

- Largest number that can be represented
  - $2.0 \times 2^{128} = 2.0 \times 10^{38}$
- Smallest number that can be represented
  - $1.0 \times 2^{-127} = 2.0 \times 10^{-38}$
- Overflow
  - representing a number larger than the one above
- Underflow
  - representing a number smaller than the one above

Special Notes

- The number "0" has a special code so that the implicit 1 does not get added
  - see discussion of denorms (pg. 222) in the textbook

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0

$1 \times 2^{-126} \times 0$

- The largest exponent value (with zero fraction) represents +/- infinity

- The largest exponent value (with non-zero fraction) represents NaN (not a number) – for the result of 0/0 or (infinity minus infinity)

Special Notes

- The number "0" has a special code so that the implicit 1 does not get added
  - see discussion of denorms (pg. 222) in the textbook

0 | 00000000 | 00000000000000000000000 = 0

$1 \times 2^{-126} \times 0$

- The largest exponent value (with zero fraction) represents +/- infinity

1 | 11111111 | 00000000000000000000000 = -Infinity

$-1 \times 2^{128} \times 1$

- The largest exponent value (with non-zero fraction) represents NaN (not a number) – for the result of 0/0 or (infinity minus infinity)

Special Notes

- The number “0” has a special code so that the implicit 1 does not get added
  - see discussion of denorms (pg. 222) in the textbook

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0

$1 \times 2^{-126} \times 0$

- The largest exponent value (with zero fraction) represents +/- infinity

1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= -Infinity

$-1 \times 2^{128} \times 1$

- The largest exponent value (with non-zero fraction) represents NaN (not a number) – for the result of 0/0 or (infinity minus infinity)

1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	= NaN

$-1 \times 2^{128} \times 1.0000001$

Special Notes

- The number "0" has a special code so that the implicit 1 does not get added
  - see discussion of denorms (pg. 222) in the textbook

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0

$1 \times 2^{-126} \times 0$

- The largest exponent value (with zero fraction) represents +/- infinity

1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= -Infinity

$-1 \times 2^{128} \times 1$

- The largest exponent value (with non-zero fraction) represents NaN (not a number) – for the result of 0/0 or (infinity minus infinity)

1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	= NaN

$-1 \times 2^{128} \times 1.0000001$

These choices impact the smallest and largest numbers that can be represented

Example 1

- Represent -0.75_{ten} in single- and double-precision formats

$$\text{value} = (-1)^{S} \times (1 + F) \times 2^{(E - 127)}$$

- Single

1 bit	8 bits	23 bits
S	E	F

- Double

1 bit	11 bits	52 bits
S	E	F

Example 1

- Represent -0.75_{ten} in single- and double-precision formats

$$-0.75 = -1 \times 0.11_{two} = -1 \times 1.1_{two} \times 2^{-1} = (-1)^{1} \times (1 + F) \times 2^{(E - 127)}$$

- Single

1 bit	8 bits	23 bits
1	01111110	10000000000000000000000
	126	0.5

- Double $-1 \times 1.1_{two} \times 2^{-1} = (-1)^{1} \times (1 + F) \times 2^{(E - 1023)}$

1 bit	11 bits	52 bits
1	01111111110	10 000000000000000000000000000000…000
	1022	0.5

Example 2

- Represent 3.40625_{ten} in single- and double-precision formats

$$\text{value} = (-1)^{S} \times (1 + F) \times 2^{(E - 127)}$$

- Single

1 bit	8 bits	23 bits
S	E	F

- Double

1 bit	11 bits	52 bits
S	E	F

Example 2

- Represent 3.40625_{ten} in single- and double-precision formats

$$3.40625 = 11.01101_{two} = 1.101101_{two} \times 2^{1} = (-1)^{0} \times (1 + F) \times 2^{(E - 127)}$$

- Single

1 bit	8 bits	23 bits
0	10000000	10110100000000000000000
	128	0.703125

- Double $1.101101_{two} \times 2^{1} = (-1)^{0} \times (1 + F) \times 2^{(E - 1023)}$

1 bit	11 bits	52 bits
0	10000000000	101101000000000000000000000000000000...000
	1024	0.703125

Floating Point Addition

- Numbers maintain only 4 decimal digits and 2 exponent digits
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$

  - Convert to the larger exponent
    - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
  - Add
    - 10.015×10^{1}
  - Normalize
    - 1.0015×10^{2}
  - Check for overflow/underflow
  - Round
    - 1.002×10^{2}
  - Re-normalize

Floating Point Addition

- Numbers maintain only 4 decimal digits and 2 exponent digits
  - $9.999 \times 10^1 \quad + \quad 1.610 \times 10^{-1}$

- Convert to the larger exponent
  - $9.999 \times 10^1 \quad + \quad 0.016 \times 10^1$
- Add
  - 10.015×10^1
- Normalize
  - 1.0015×10^2
- Check for overflow/underflow
- Round
  - 1.002×10^2
- Re-normalize

If we had more fraction bits, these errors would be minimized

Floating Point Addition

- Numbers maintain only 4 binary digits and 2 exponent digits
  - $1.010 \times 2^1 + 1.100 \times 2^3$

- Convert to the larger exponent
  - $0.0101 \times 2^3 + 1.100 \times 2^3$
- Add
  - 1.1101×2^3
- Normalize
  - 1.1101×2^3
- Check for overflow/underflow
- IEEE 754 format

0 10000010 11010000000000000000000