Lecture 10: Division, Floating Point

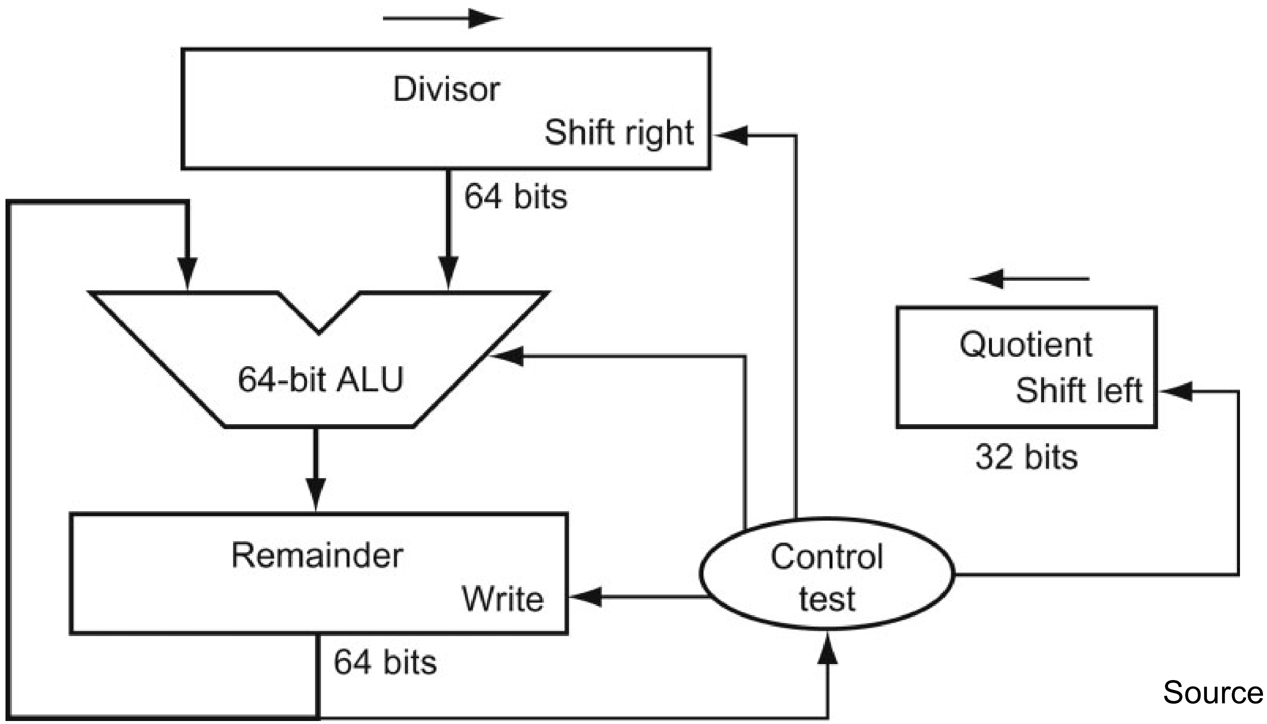
- Today's topics:
  - Division
  - IEEE 754 representations

Divide Example

- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 ➔ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 ➔ shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

Hardware for Division

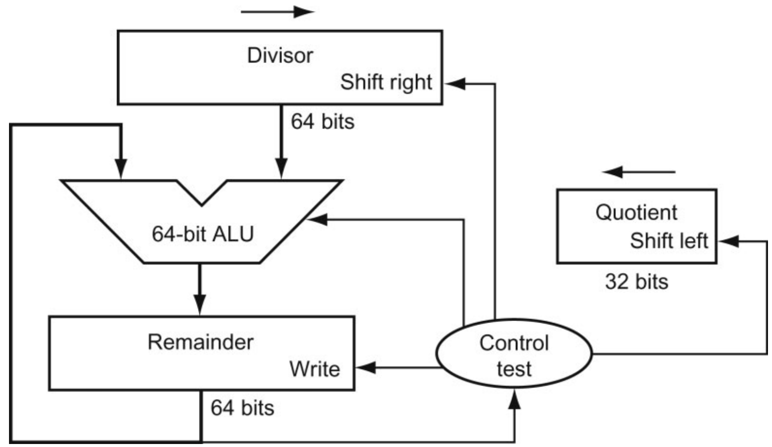


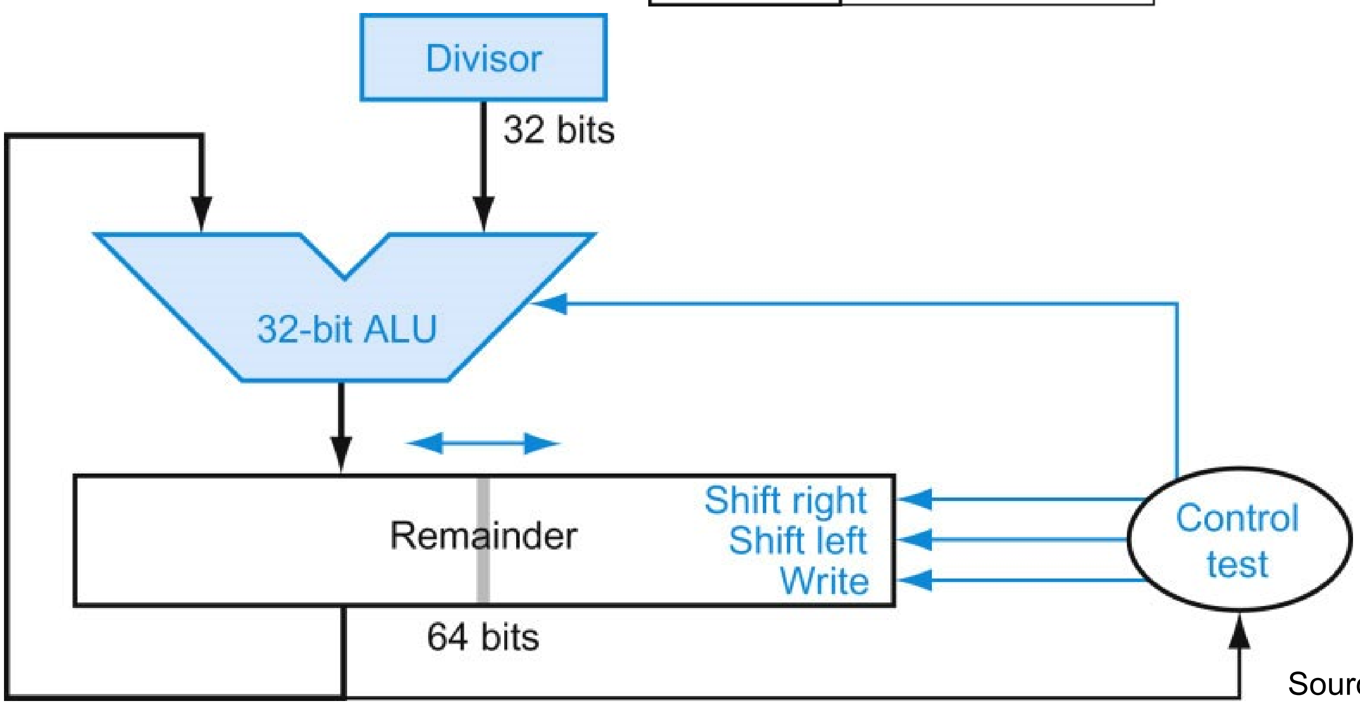
Source: H&P textbook

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)

Efficient Division





Source: H&P textbook

Divisions involving Negatives

- 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 =

Divisions involving Negatives

- 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

Take Homes

- Grade school algorithms are commonly used – the algorithms are even easier in binary (mult by 1 and 0)

- They can be implemented in hardware with shifts, add, sub, checks

- To improve efficiency, look for ineffectuals – are only some bits changing in every step – allows us to use narrow adders and registers – allows us to pack more operands in one register

- Can also improve speed by throwing more transistors and parallel computations at the problem

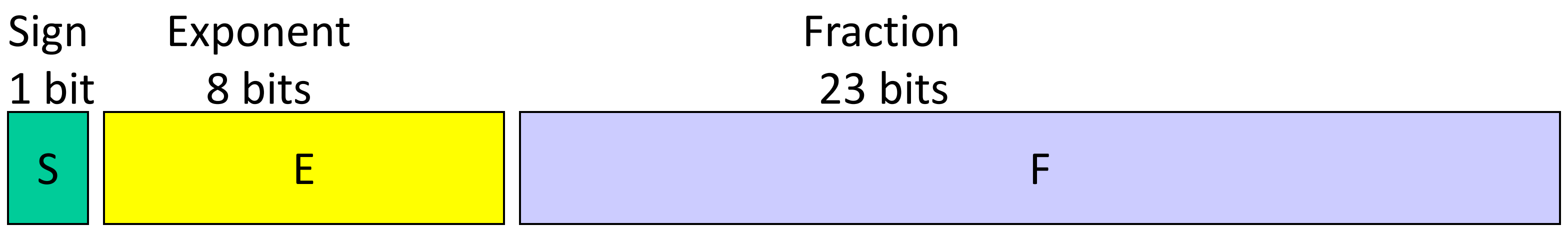
Floating Point

- Normalized scientific notation: single non-zero digit to the left of the decimal (binary) point – example: 3.5×10^9

- $1.010001 \times 2^{-5}{}_{two} = (1 + 0 \times 2^{-1} + 1 \times 2^{-2} + \ldots + 1 \times 2^{-6}) \times 2^{-5}{}_{ten}$

- A standard notation enables easy exchange of data between machines and simplifies hardware algorithms – the IEEE 754 standard defines how floating point numbers are represented

Sign and Magnitude Representation



- More exponent bits ➔ wider range of numbers (not necessarily more numbers – recall there are infinite real numbers)

- More fraction bits ➔ higher precision

- Register value = $(-1)^S$ x F x 2^E

- Since we are only representing normalized numbers, we are guaranteed that the number is of the form 1.xxxx..
  Hence, in IEEE 754 standard, the 1 is implicit
  Register value = $(-1)^S$ x (1 + F) x 2^E

Exponent Representation

- To simplify sort, sign was placed as the first bit
- For a similar reason, the representation of the exponent is also modified: in order to use integer compares, it would be preferable to have the smallest exponent as 00…0 and the largest exponent as 11…1
- This is the biased notation, where a bias is subtracted from the exponent field to yield the true exponent
- IEEE 754 single-precision uses a bias of 127 (since the exponent must have values between -127 and 128)…double precision uses a bias of 1023

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

Sign and Magnitude Representation

Sign 1 bit	Exponent 8 bits	Fraction 23 bits
S	E	F

- Largest number that can be represented: $2.0 \times 2^{128} = 2.0 \times 10^{38}$ (not really – see upcoming details)
- Smallest number that can be represented: $1.0 \times 2^{-127} = 2.0 \times 10^{-38}$ (not really – see upcoming details)
- Overflow: when representing a number larger than the max; Underflow: when representing a number smaller than the min

- Double precision format: occupies two 32-bit registers:
  Largest: Smallest:

Sign 1 bit	Exponent 11 bits	Fraction 52 bits
S	E	F

Details

- The number "0" has a special code so that the implicit 1 does not get added: the code is all 0s
  (it may seem that this takes up the representation for 1.0, but given how the exponent is represented, that's not the case)
  (see discussion of denorms in the textbook)

- 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)

- Note that these choices impact the smallest and largest numbers that can be represented

Value inf — 2 special cases up top that use the reserved exponent field of 255 — 0 255 00…0

Value NAN — 0 255 xx….x

Highest value ~2 x 2^{127} — 0 254 11….1

Value 1 — 0 127 00…0

Exponent field < 127, i.e., after subtracting bias, they are negative exponents, representing numbers < 1

Smallest Norm 1 x 2^{-126} — 0 0..01 00…0

Largest Denorm ~1 x 2^{-126} — 0 0..00 11…1

Smallest Denorm ~2^{-149} — 0 0..00 00…1

Value 0 — 0 00..0 00…0

Special case with exponent field 0, used to represent denorms, that help us gradually approach 0. Denorms don't have implicit 1. They have exp 2^{-126}

Same rules as above, but the sign bit is 1

Same magnitudes as above, but negative numbers

Examples

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

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

  Single: (1 + 8 + 23)

  Double: (1 + 11 + 52)

> Remember:
>
> True exponent $\underset{-127}{\overset{+127}{\rightleftarrows}}$ Exponent in register

- What decimal number is represented by the following single-precision number?

  1 1000 0001 01000…0000

Examples

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

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

  Single: (1 + 8 + 23)
  1 0111 1110 1000…000

  Double: (1 + 11 + 52)
  1 0111 1111 110 1000…000

- What decimal number is represented by the following single-precision number?
  1 1000 0001 01000…0000
  -5.0

Example 2

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent 36.90625_{ten} in single-precision format

36 / 2 = 18 rem 0	0.90625 x 2 = 1.81250
18 / 2 = 9 rem 0	0.8125 x 2 = 1.6250
9 / 2 = 4 rem 1	0.625 x 2 = 1.250
4 / 2 = 2 rem 0	0.25 x 2 = 0.50
2 / 2 = 1 rem 0	0.5 x 2 = 1.00
1 / 2 = 0 rem 1	0.0 x 2 = 0.0

36 is 100100

0.90625 is 0.1110100…0

Example 2

Final representation: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

We've calculated that 36.90625_{ten} = 100100.1110100…0 in binary
Normalized form = $1.001001110100\ldots0 \times 2^5$
(had to shift 5 places to get only one bit left of the point)

The sign bit is 0 (positive number)
The fraction field is 001001110100…0 (the 23 bits after the point)
The exponent field is 5 + 127 (have to add the bias) = 132,
which in binary is 10000100

The IEEE 754 format is 0 10000100 001001110100…..0

	sign	exponent	23 fraction bits