Computer arithmetic

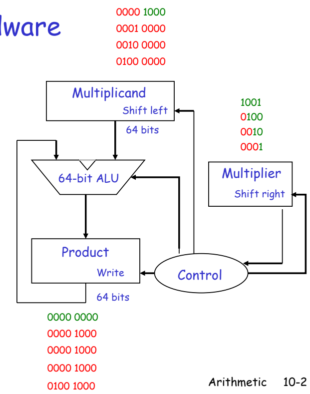
Integers and floats

CS240 Computer Organization
Department of Computer Science
Wellesley College

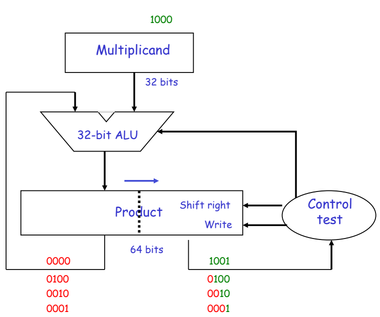
Multiplication hardware

- The multiplication algorithm we learned in grammar school leads to a simple if somewhat inefficient design:

```
          1000    Multiplicand
      x   1001    Multiplier
          1000
         0000
        0000
       1000
       1001000    Product
```



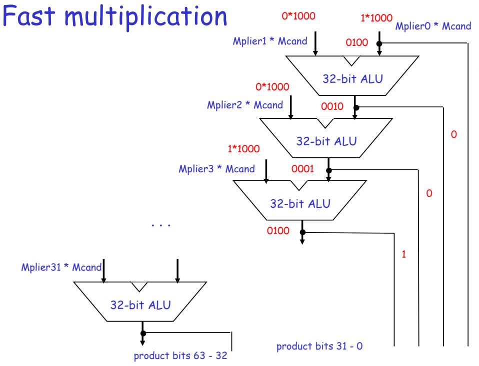
Refined version of mult hardware



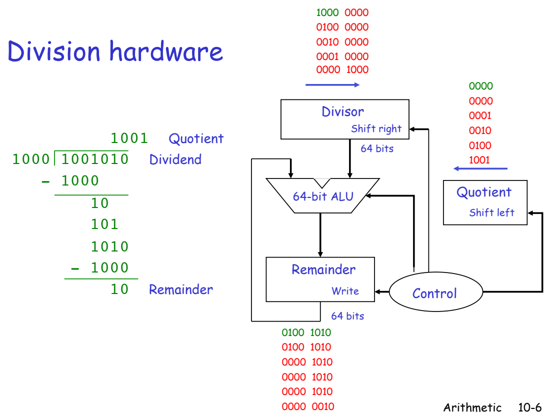
Twos-complement multiplication

- Multiplication of signed integers could be done by multiplying numbers and then figuring out the sign.
- However, the above algorithm works for twos-complement provided we extend the sign bit appropriately.

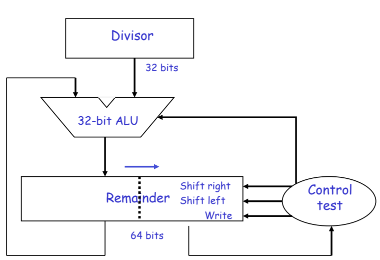
Fast multiplication



Division hardware



Refined version of division hardware



Representing reals

- Numbers with fractions, e.g., 3.14159265… (π).
- Very small numbers, e.g., .000000001 = 1.0×10^{-9}
- Very large numbers, e.g., 3,155,760,000 = 3.15576×10^{9}

Binary fractions

- We can also show binary numbers in normalized scientific notation using the notation:

  $1.xxxx_2 \times 2^{yyyy}$

- We now call the "." symbol the binary point. We also call this format "floating-point" because the binary point is not fixed.

Life is not also so easy

- What if the fractional part is not an exact power of two? We will need to approximate!

  $2^{-1} = .1_2 = .5_{10}$
  $2^{-2} = .01_2 = .25_{10}$
  $2^{-3} = .001_2 = .125_{10}$
  $2^{-4} = .0001_2 = .0625_{10}$
  $2^{-5} = .00001_2 = .03125_{10}$
  $2^{-6} = .000001_2 = .015625_{10}$
  $2^{-7} = .0000001_2 = .0078125_{10}$
  $2^{-8} = .00000001_2 = .0039063_{10}$
  $2^{-9} = .000000001_2 = .0019531_{10}$
  $2^{-10} = .0000000001_2 = .0009766_{10}$ etc…

- Then, sum to the degree of precision necessary. For example, what does $.3_{10} =$?

Floating-point numbers: $(-1)^{sign} \times fraction \times 2^{exponent}$

- A fixed word size means that there is a tradeoff between accuracy and range:
  - more bits for fraction gives more precision of fraction
  - more bits for exponent increases range of numbers that can be represented.

IEEE 754 floating-point standard

- Single precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

s	exponent	fraction
1 bit	8 bits	23 bits

- Double precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

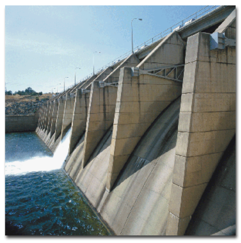
s	exponent	fraction
1 bit	11 bits	20 bits

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

fraction continued
32 bits

Water over (and under) the dam

- Overflow is now the case where the positive exponent is too large to be represented in the exponent field.
- But now there is also underflow, i.e., the case where the negative exponent is too large to fit in the exponent field. In this case, the fractional part has become so small that it cannot be represented.



Some unpleasant details

- Placing the exponent before the fraction simplifies sorting of floating point numbers using integer comparison instructions.
- But, what about negative exponents? They "look" big because the leading digit is a 1!
- Exponent is "biased" to make sorting easier. The bias is subtracted from the normal, unsigned value, to determine the real value. Bias = 127 for single precision and 1023 for double precision:

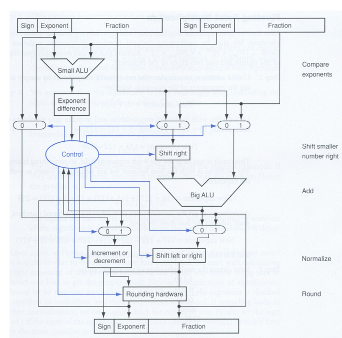
$$(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent - bias}}$$

So an IEEE engineer walks into a bar ...

... and orders 1.0000000000010000000082740370999090373516082763671875 root beers. The bartender says, "I'll have to charge extra; that's a root beer float". And the engineer says, "In that case, make it a double."

We order a $-.75_{10}$ float and make it a double.

Aaaahhhh!!!



Floating point multiplication: 1.000×2^{-1} times -1.110×2^{-2}

1. Add exponent without bias:
   `-1 + (-2) = -3`
2. Multiple significants:

```
      1.000
    x 1.110
    -------
       0000
      1000
     1000
    1000
    -------
   1.110000
```

3. Normalize and check for overflow or underflow:
   1.110×2^{-3}
4. Round the product:
   1.110×2^{-3}
5. Calculate sign of result:
   `1 x (-1) = -1`

The infamous Pentium bug (1994)