Computer Arithmetic

Multiplication

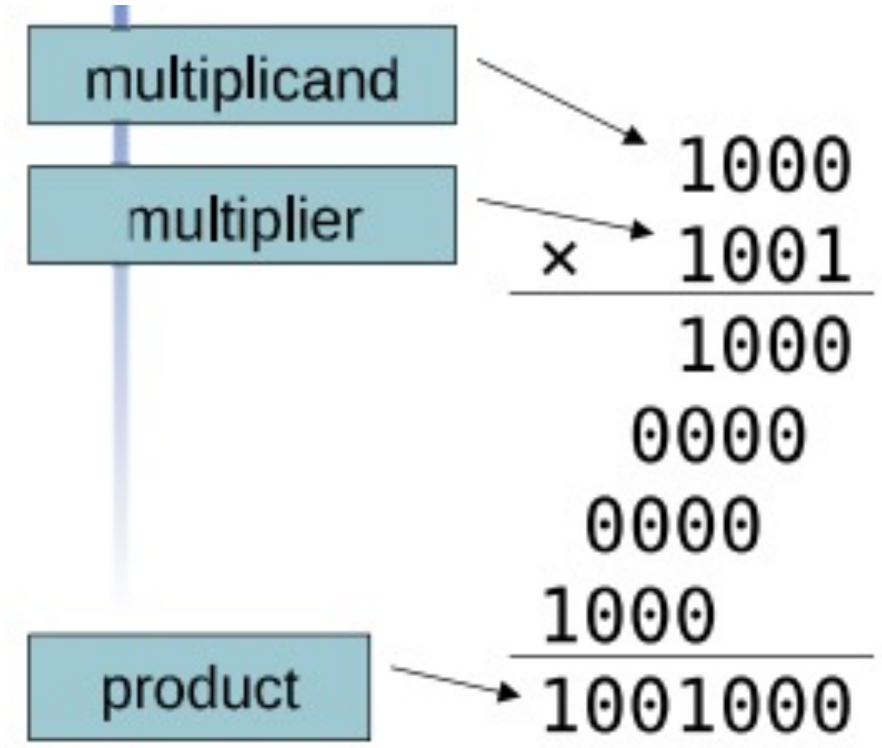
- Binary multiplication is similar to the multiplication of decimal numbers. We have a multiplier and a multiplicand.

  $0 \times 0 = 0$

  $0 \times 1 = 0$

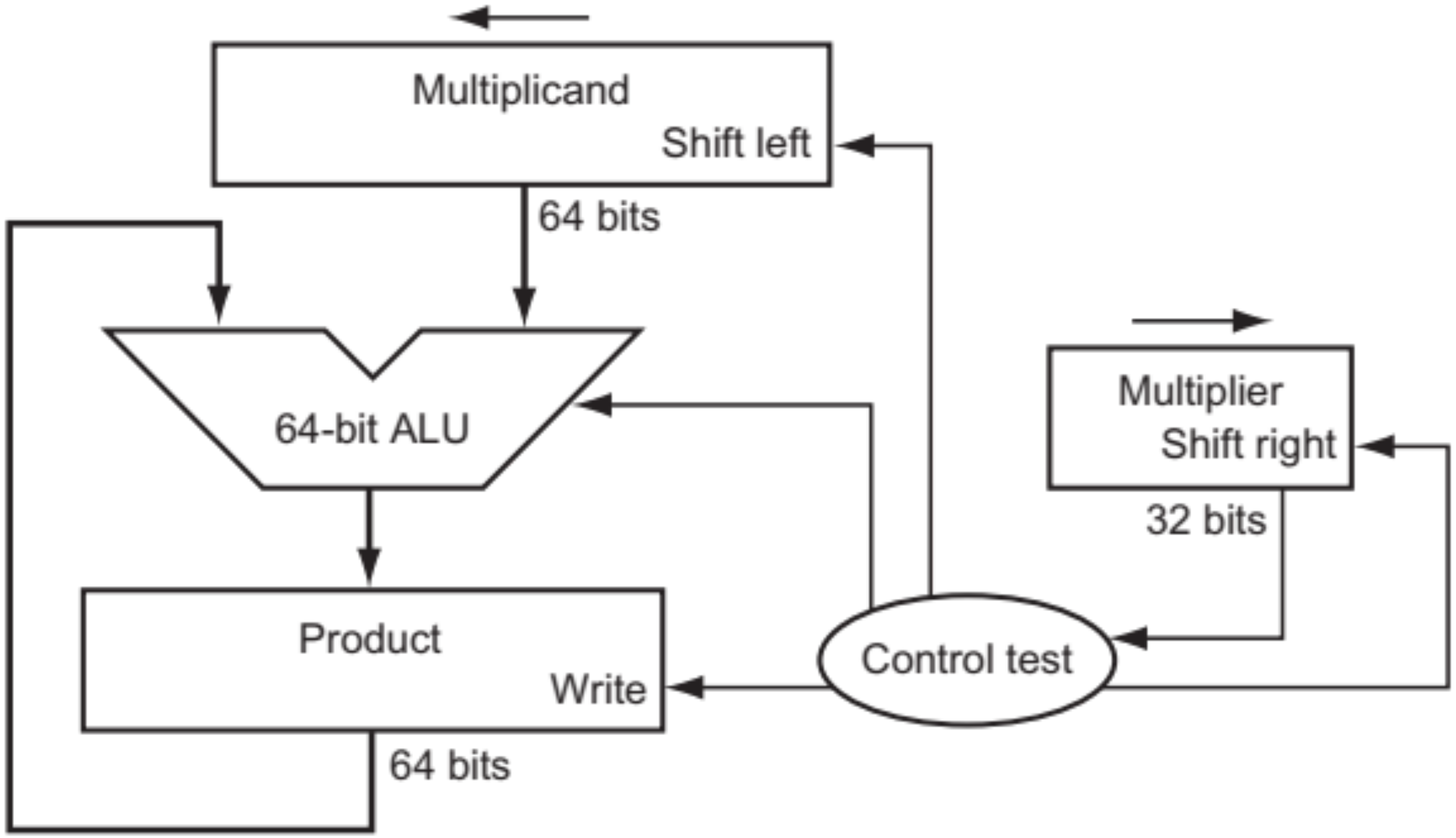
  $1 \times 0 = 0$

  $1 \times 1 = 1$



- The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier.
- In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long. That is, n + m bits are required to represent all possible products.
- Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

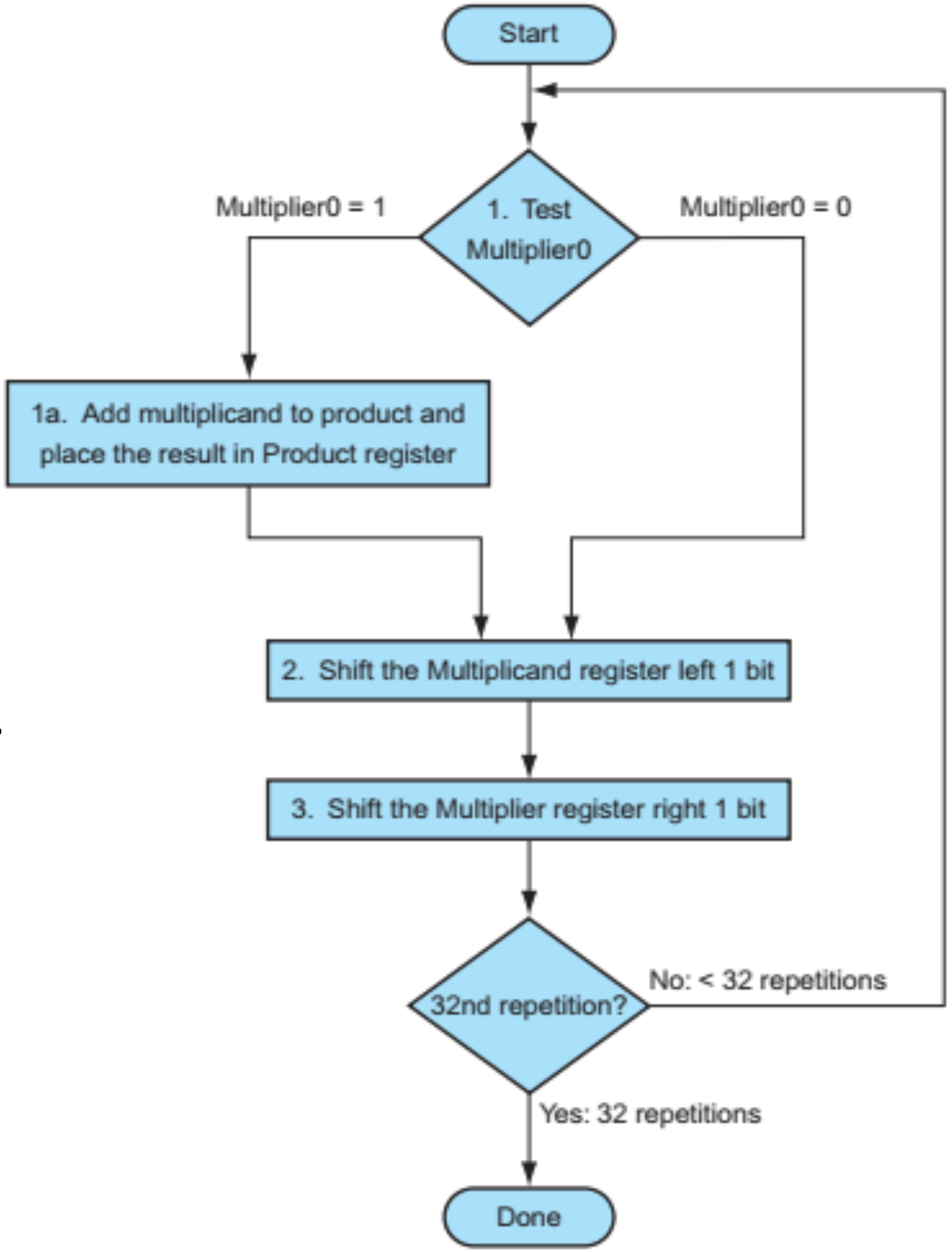
Multiplication hardware



- The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits.
- The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step.
- The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0.
- Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Multiplication algorithm

1. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register.
2. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil.
3. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration.
4. These three steps are repeated 32 times to obtain the product.



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

MIPS Multiplication

- MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called HI and LO.
- To produce a properly signed or unsigned product, MIPS has two instructions: multiply (**mult**) and multiply unsigned (**multu**).
- To fetch the integer 32-bit product, the programmer uses ***move from lo*** (**mflo**).
- The MIPS assembler generates a pseudo instruction for multiply that specifies three general-purpose registers, generating mflo and mfhi instructions to place the product into reg

Division

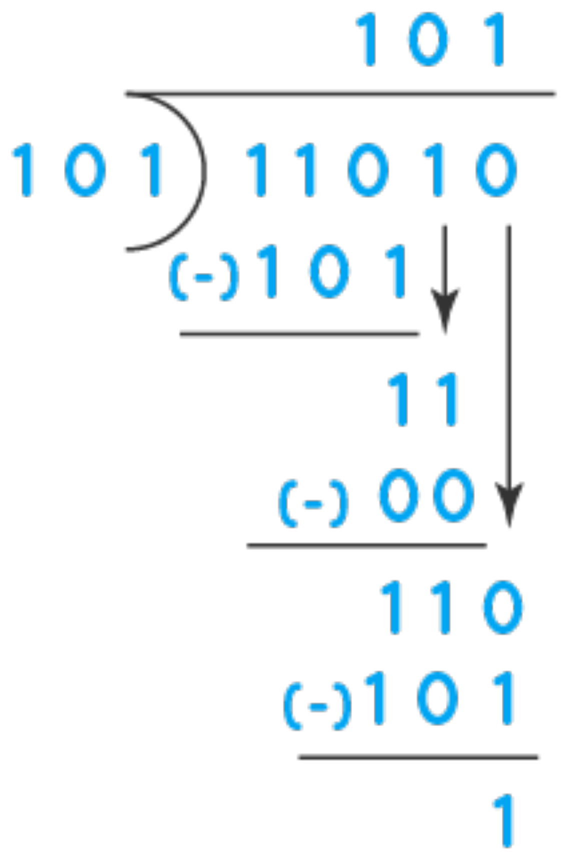
- 1÷1 = 1

  1÷0 = Meaningless

  0÷1 = 0

  0÷0 = Meaningless

```
     94
  5 |473
     45
     --
     23
     20
     --
      3
```



```
          101
     ________
101 ) 110010
   (-)101
      ---
       11
    (-) 00
      ---
       110
    (-)101
       ---
         1
```

mul op1, op2

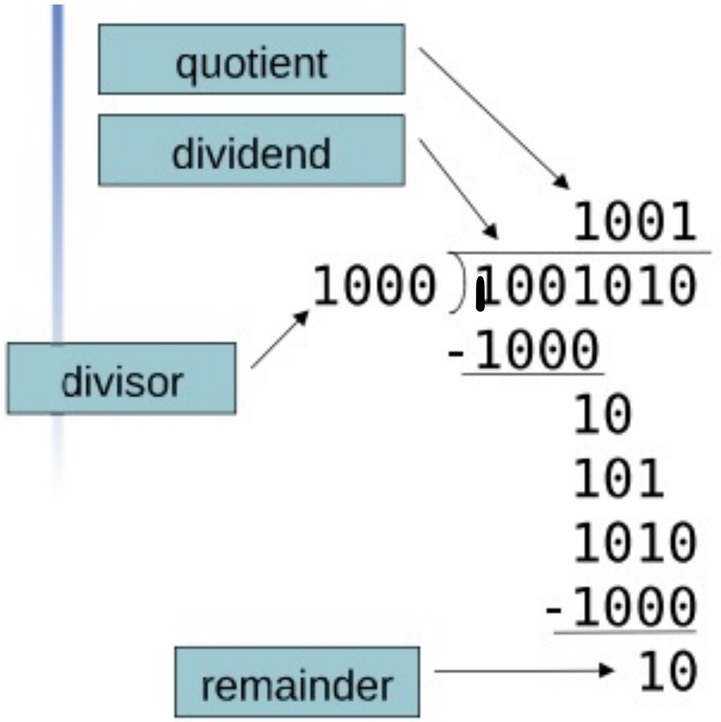
div op1, op2

- Divide's two operands, called the dividend and divisor, and the result, called the quotient, are accompanied by a second result, called the remainder. Here is another way to express the relationship between the components:

$$\text{Dividend} = \text{Quotient} * \text{Divisor} + \text{Remainder}$$

where the remainder is smaller than the divisor.

Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient



$n \times m \rightarrow n$ & m

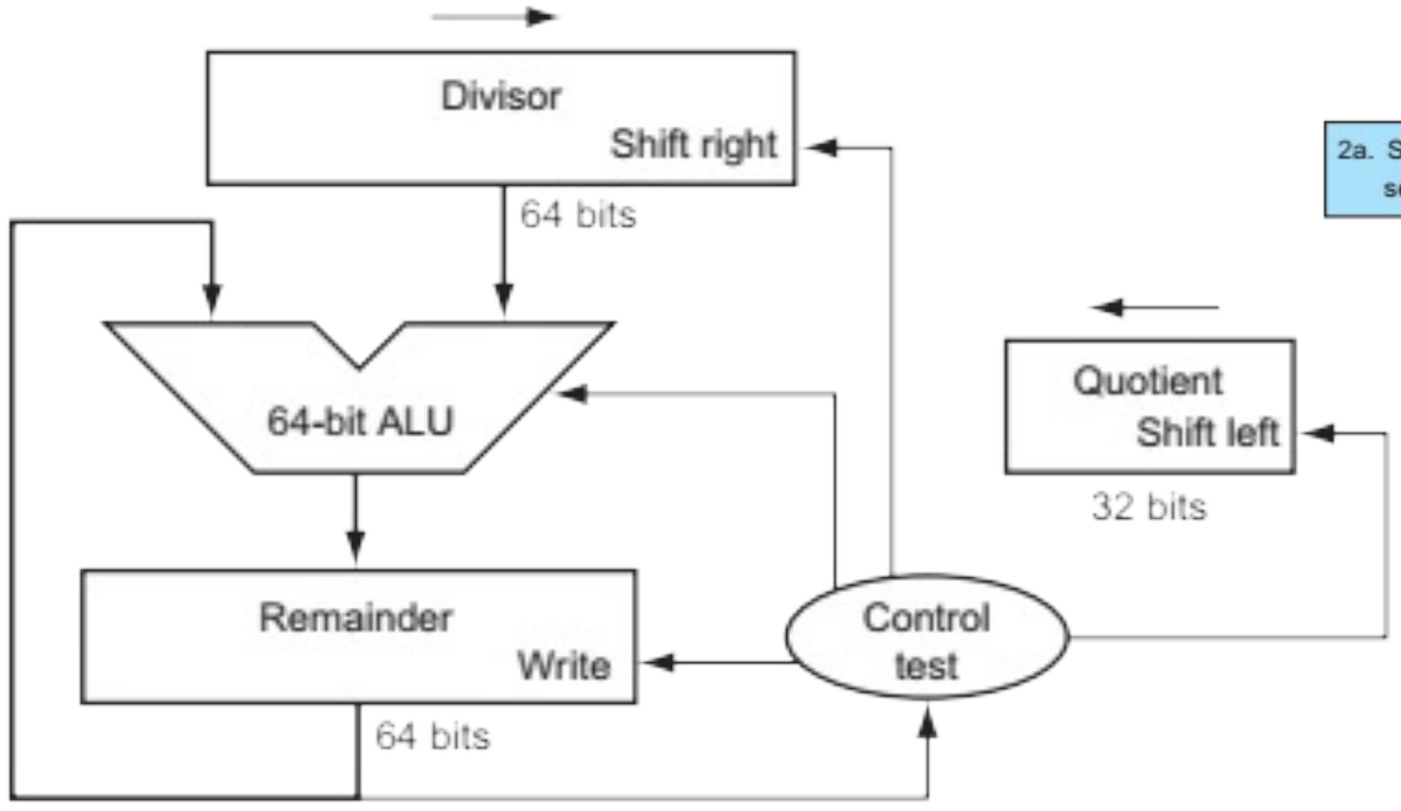
32

32

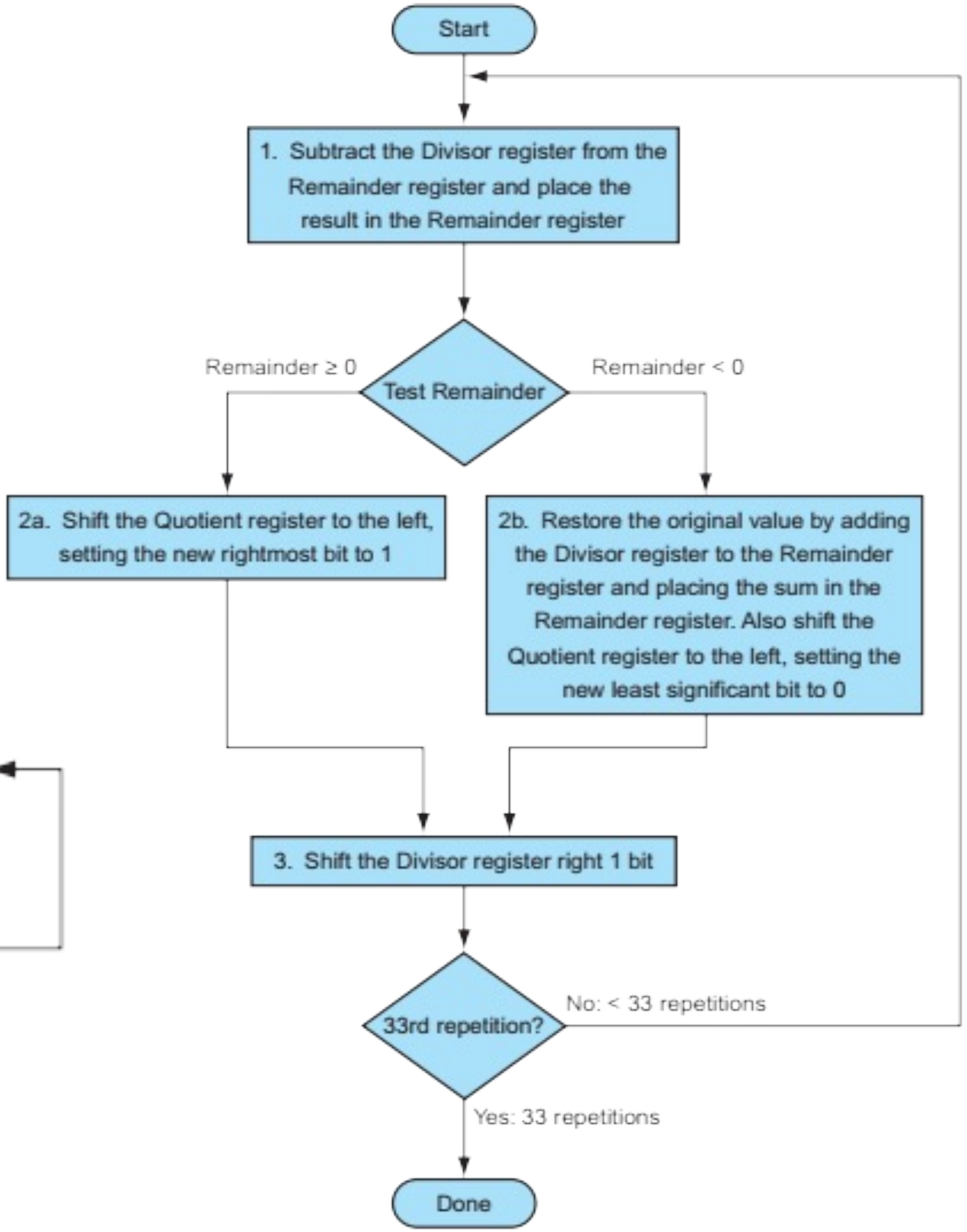
32

n-bit operands yield n-bit quotient and remainder

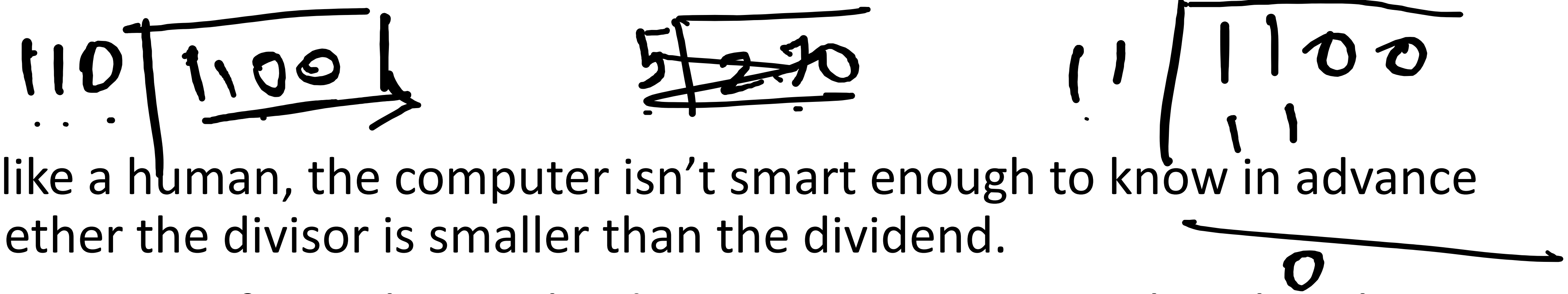
Division algorithm



First version of the division hardware



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



- Unlike a human, the computer isn’t smart enough to know in advance whether the divisor is smaller than the dividend.

Step 1: It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction.

Step 2a: If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient .

Step 2b: If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient . (restoring division)

Step 3: The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

Signed Division

$-7 \div +2$

- *The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.*

Eloboration :

The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

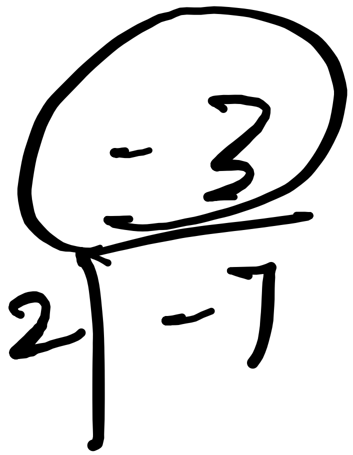
Dividend = Quotient * Divisor + Remainder

$-7 \div 2 = -3$

$-4 \div 2 = 3$

$\rightarrow -3$

Example :



To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{\text{ten}}$ by $\pm 2_{\text{ten}}$. The first case is easy:

$$+7 \div +2\text{: Quotient} = +3,\ +\ \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2\text{: Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2)\\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2\text{: Quotient} = -3,\ \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

1. long division
2. Restoring
3. signed

- We can calculate the other combinations also –

div −7, 3

$$+7 \div -2: \text{Quotient} = -3, \text{Remainder} = +1$$

$$-7 \div -2: \text{Quotient} = +3, \text{Remainder} = -1$$

- Thus the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

signed division

1. perform unsigned division
2. negate quotient
3. signs of nonzero remainder should match dividend

MIPS Division

- MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide.
- As we might expect from the algorithm above, **Hi** contains the remainder, and **Lo** contains the quotient after the divide instruction completes.
- To handle both signed integers and unsigned integers, MIPS has two instructions:

  divide (div) and divide unsigned (divu)

- The MIPS assembler allows divide instructions to specify three registers, generating the mflo or mfhi instructions to place the desired result into a general-purpose register.

$110.10_2 \rightarrow \_\_\_\__{10}$?