CSE261: Computer Architecture

6. Arithmetic for Computers (2)

Seongil Wi

Department of Computer Science and Engineering

Notification: HW1

- Due date: **Today, 11:59PM**

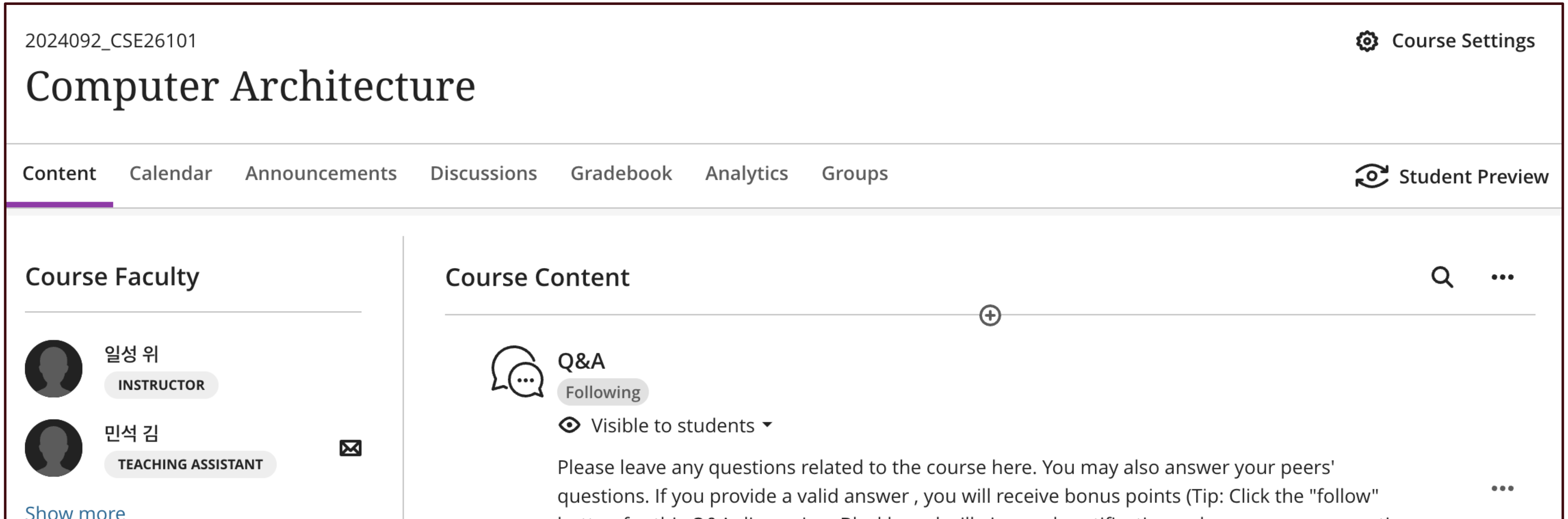
Notification: Midterm Exam

- Oct. 24 (Thursday)
- Class Time (1h 15m)

- T/F problems + Computation problems + Descriptive problems
- Closed book

Notification: Q&A

- Avoid asking questions about issues that have “clear” answers
- Avoid asking duplicated questions
- Your instructors are not here to debug!)
- **Participation point**: If you answer other students’ questions (before my answer), and if the answer is valid one, you will receive bonus point!



Integer Division

Division

- Same as multiplication: just need ***add/sub*** and ***shift operation***!

Division

- Let’s think about the division in grade school level

```
    9
8 | 74
  - 72
  ----
     2
```

```
Quotient
              01001
Divisor 1000 | 01001010   Dividend
               0100
               01001
             -  1000
             --------
                  10
                  101
                  1010
                - 1000
Remainder       ------
                    10
```

Division Algorithm in HW Manner

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

Quotient

```
          01001
     ┌─────────
1000 │ 01001010
       0100
       01001
     -  1000
     ──────────
            10
            101
            1010
          - 1000
          ──────
              10
```

Divisor

Dividend

Remainder

Division Algorithm in HW Manner

Need 5 loops for completing the operation

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

Quotient

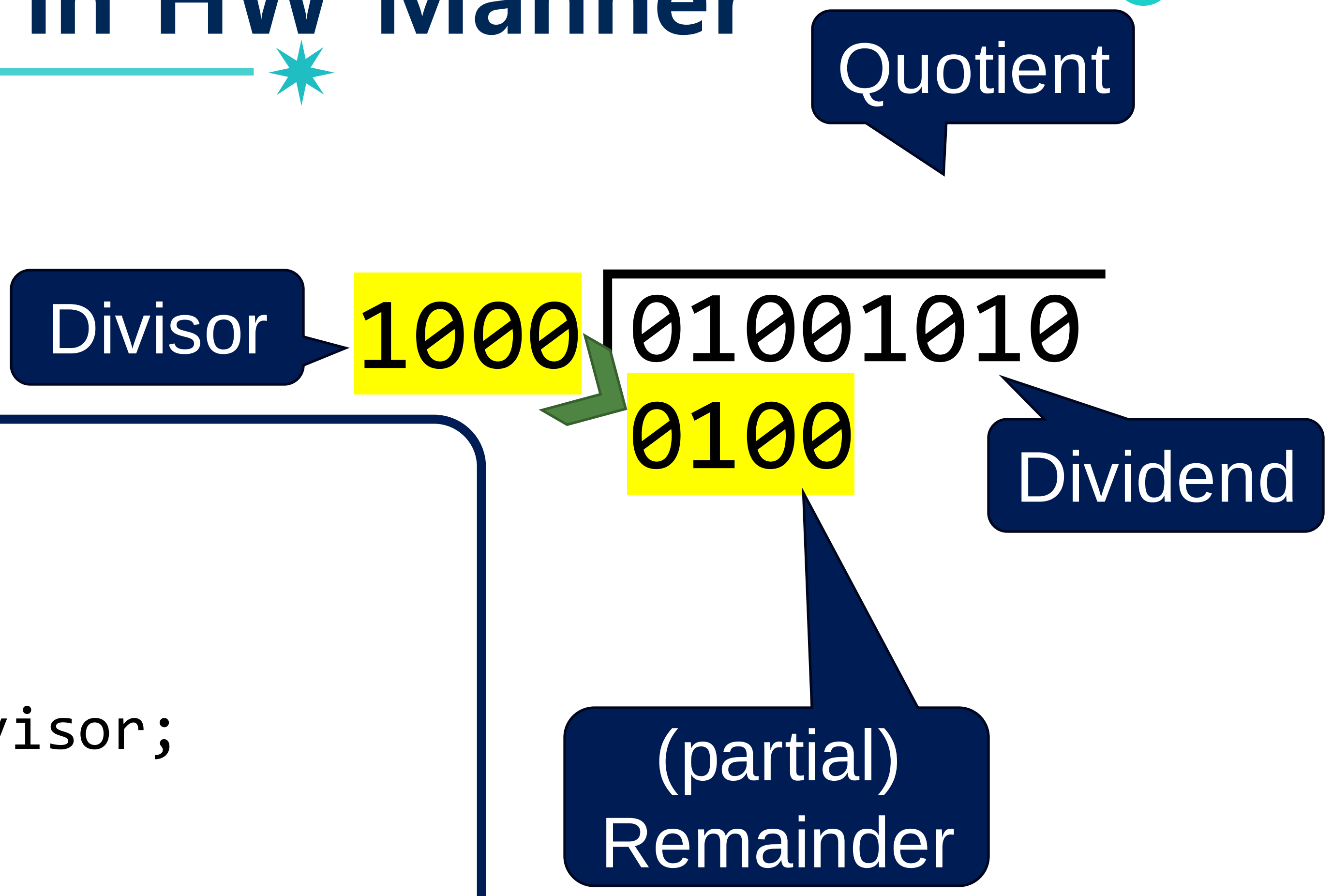
```
          01001
     ___________
1000 | 01001010
       0100
       01001
     -  1000
     ___________
           10
           101
           1010
         - 1000
     ___________
             10
```

Divisor

Dividend

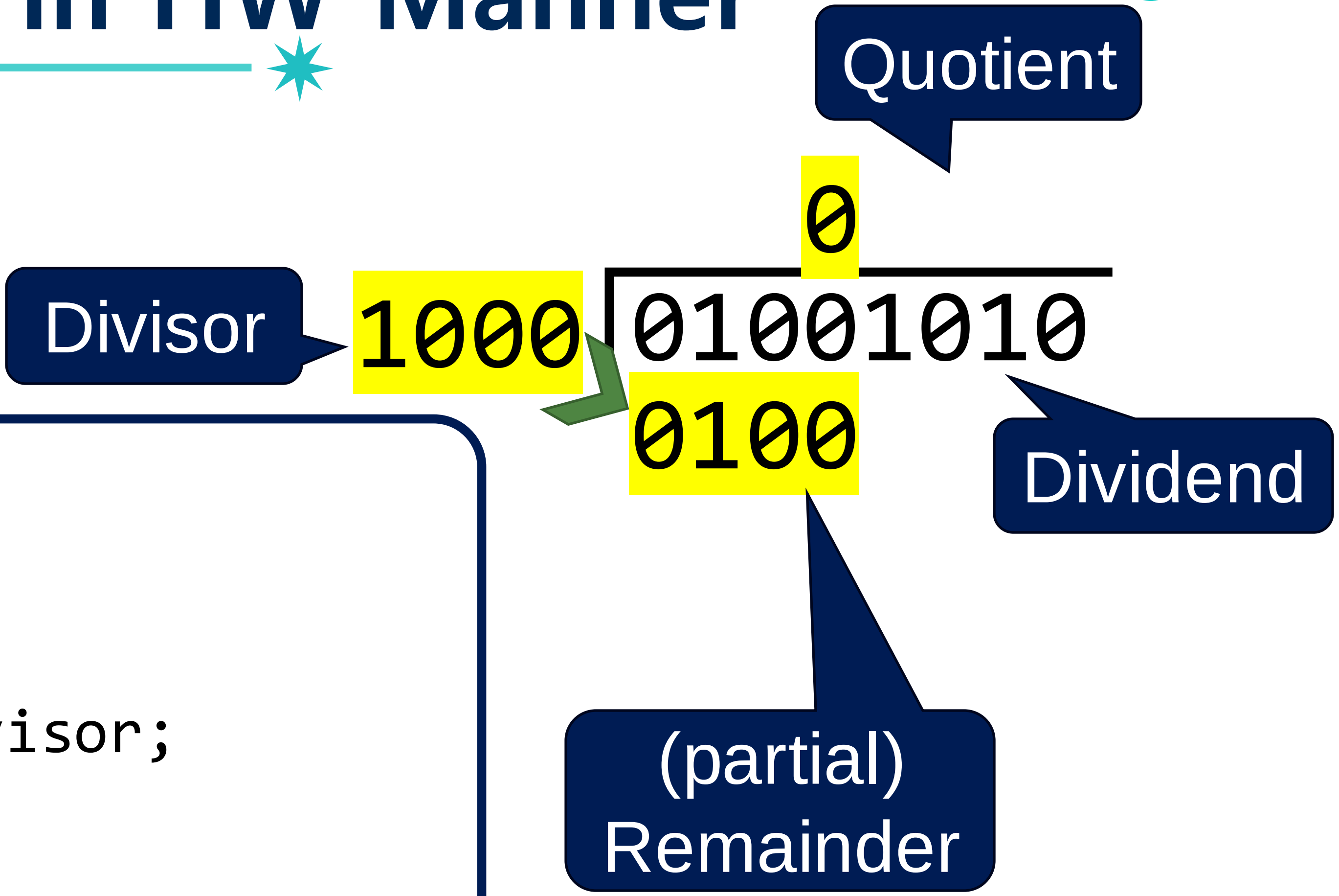
Remainder

Division Algorithm in HW Manner




```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

Division Algorithm in HW Manner




```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

Division Algorithm in HW Manner

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

Quotient

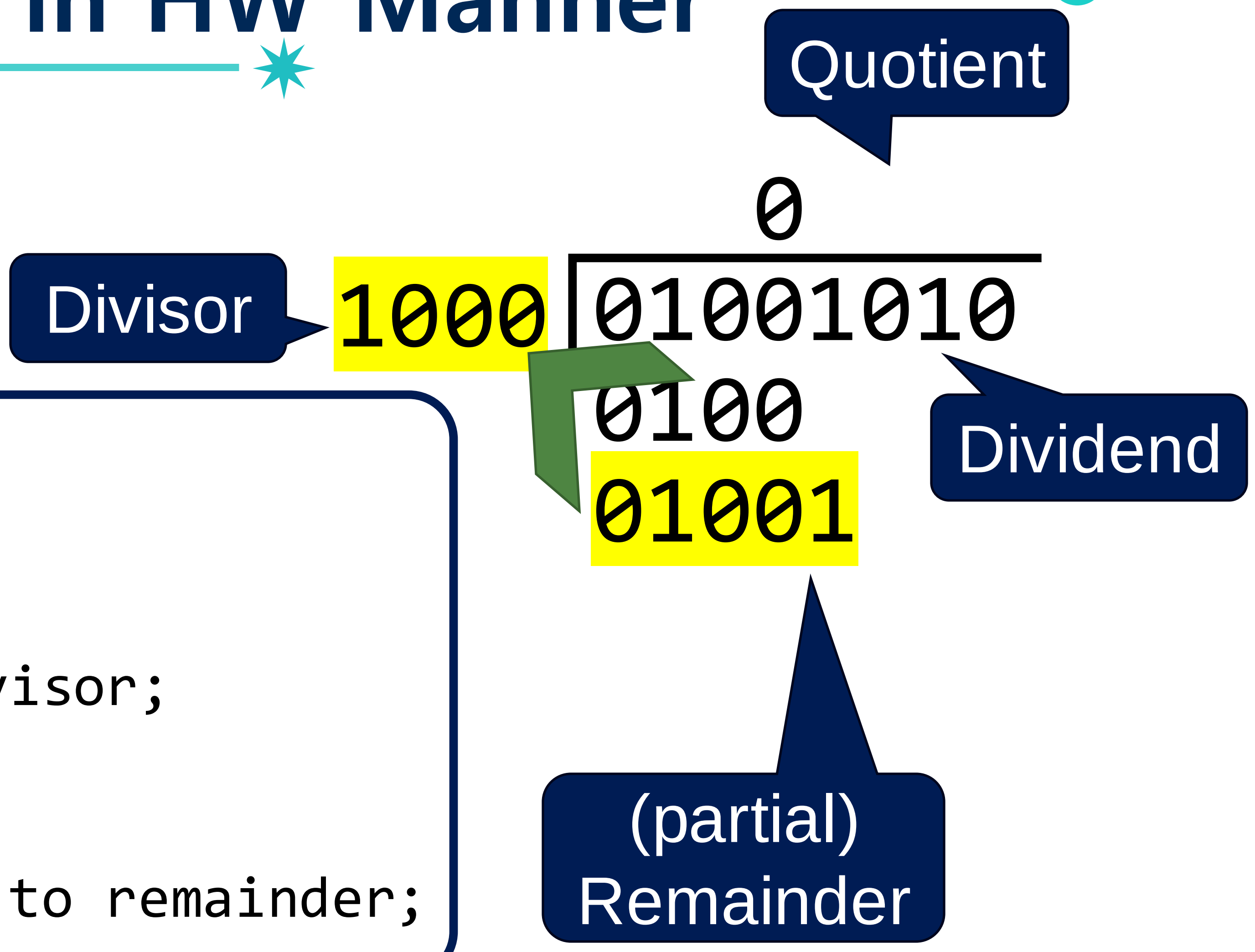
```
          0
1000 | 01001010
       0100
       01001
```

Divisor

Dividend

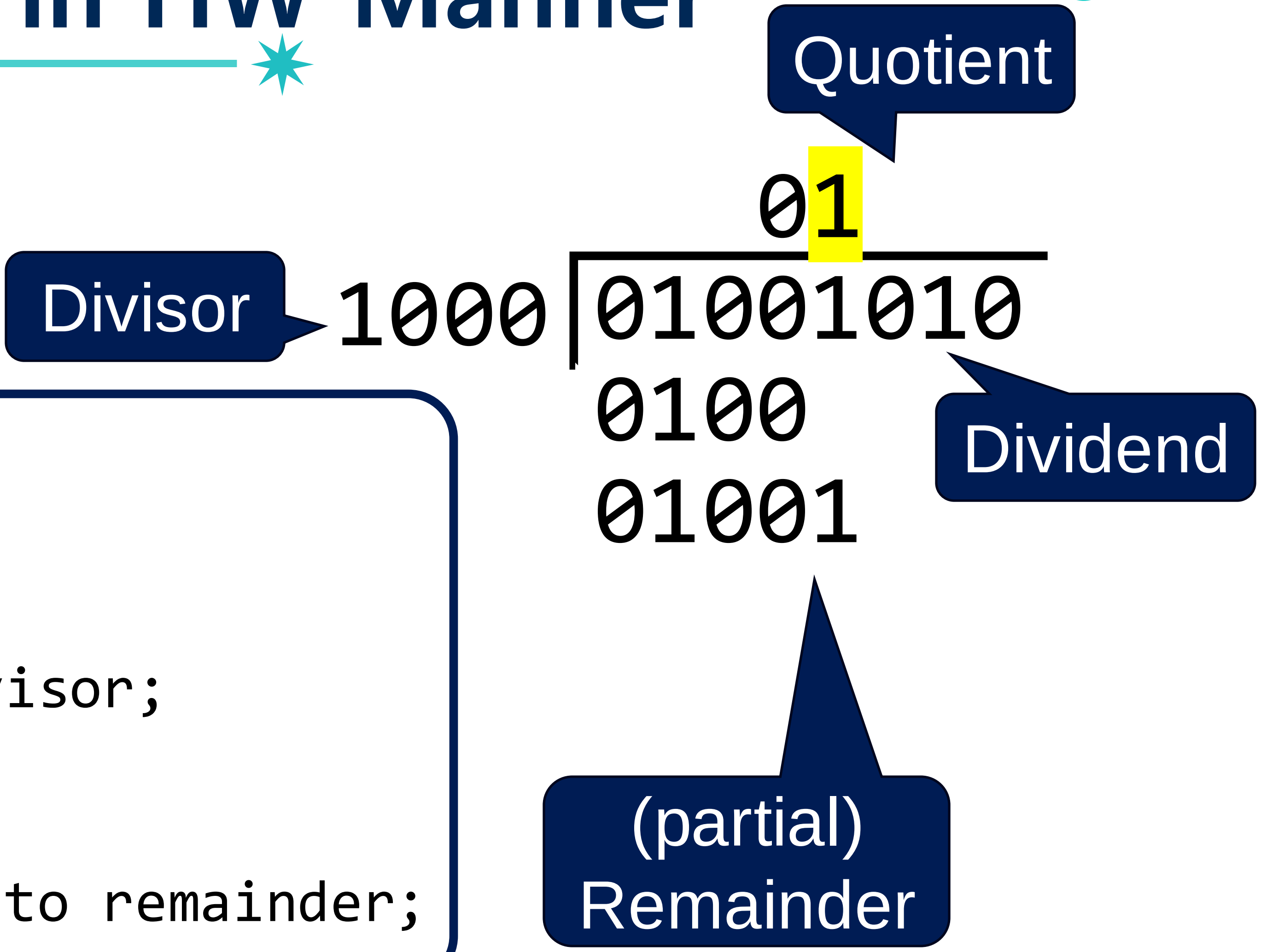
(partial) Remainder

Division Algorithm in HW Manner




```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

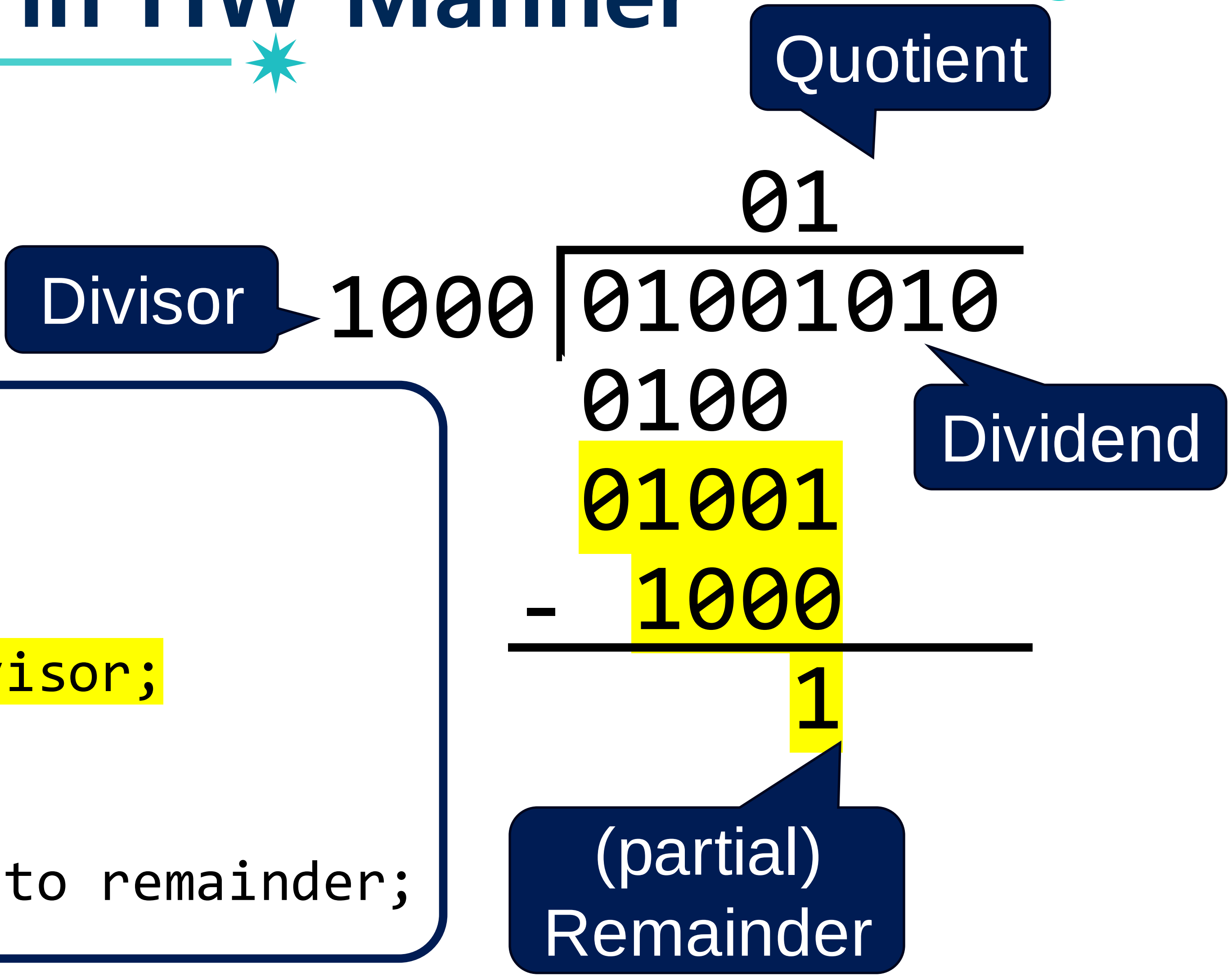
Division Algorithm in HW Manner




```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder - divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

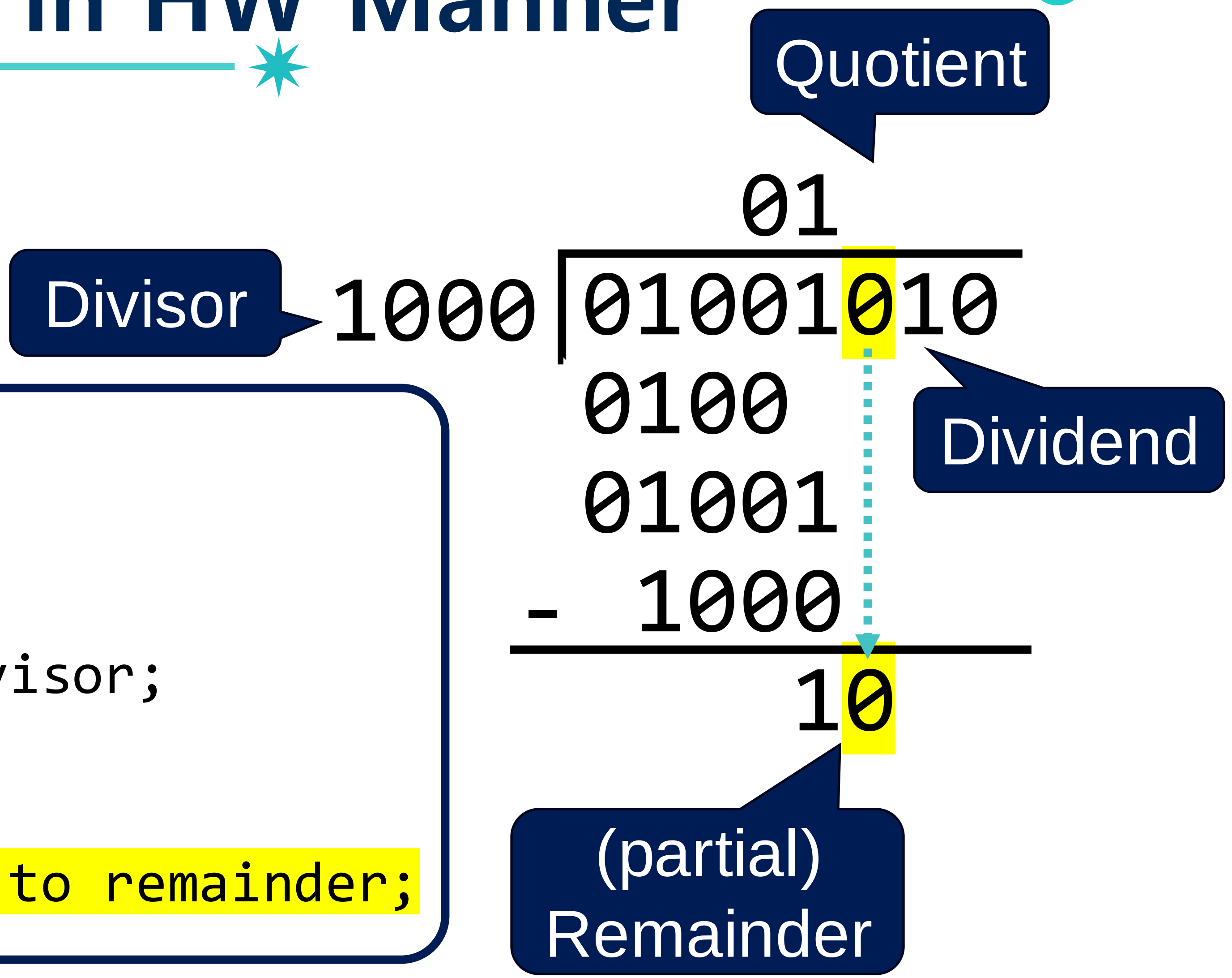
Division Algorithm in HW Manner

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```



Division Algorithm in HW Manner

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder – divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```



Division Algorithm in HW Manner

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder - divisor;
  else:
    quotient bit = 0;
  bring down next dividend bit to remainder;
```

Quotient

```
          01001
     ----------
1000 | 01001010
       0100
       01001
     -  1000
     ----------
           10
           101
           1010
         - 1000
     ----------
             10
```

Divisor

Dividend

Remainder

Division Hardware

In HW, two values are **subtracted** to compare their values
i.e., remainder = remainder – divisor

```
loop len(divisor)+1 times:
  if divisor ≤ remainder:
    quotient bit = 1;
    remainder = remainder - divisor;
  else:
    quotient bit = 0;
  bri
```

If remainder ≥ 0, the result of the subtraction is used **directly as the remainder**

If remainder < 0,
restore the remainder by adding the divisor

Quotient

01001

Divisor 1000 | 01001010 Dividend

0100

10

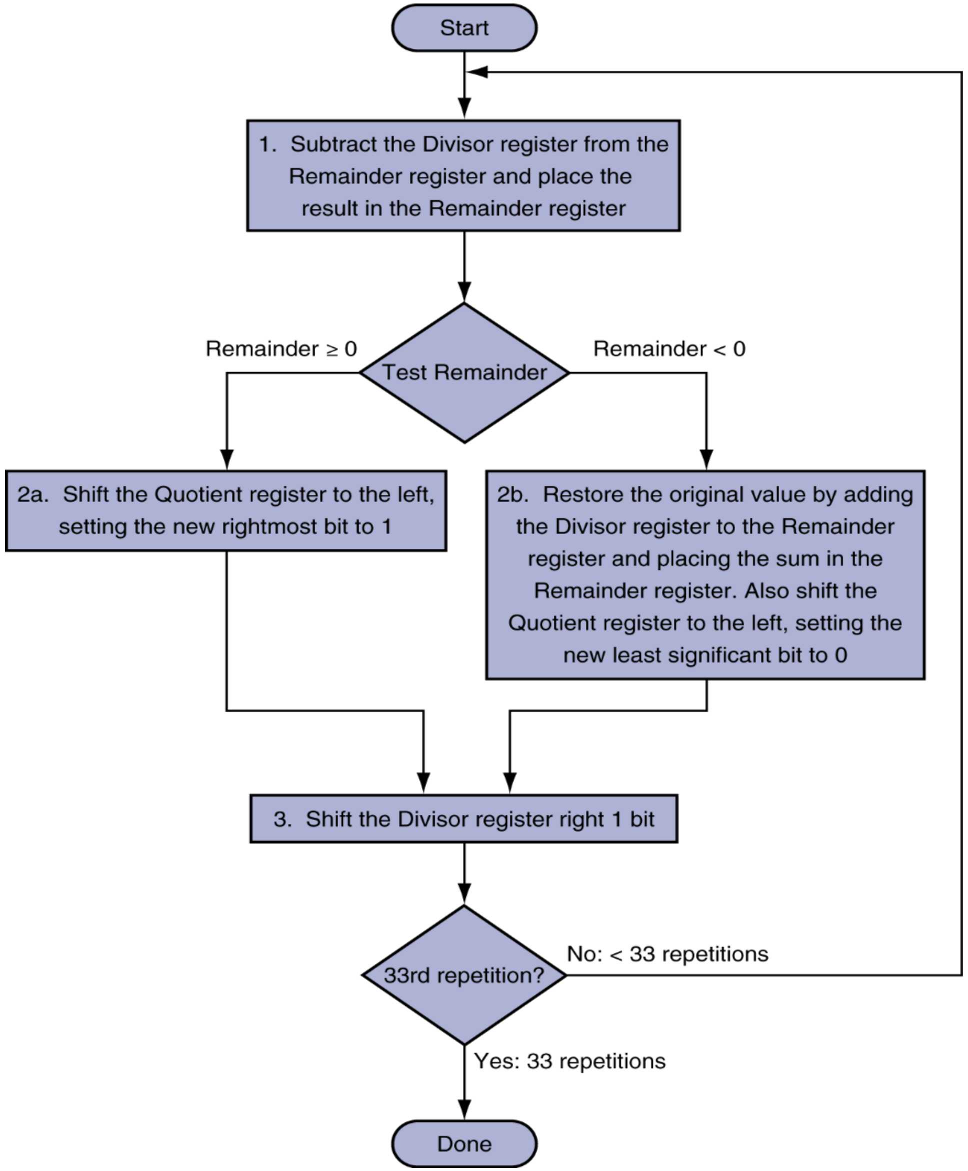
101

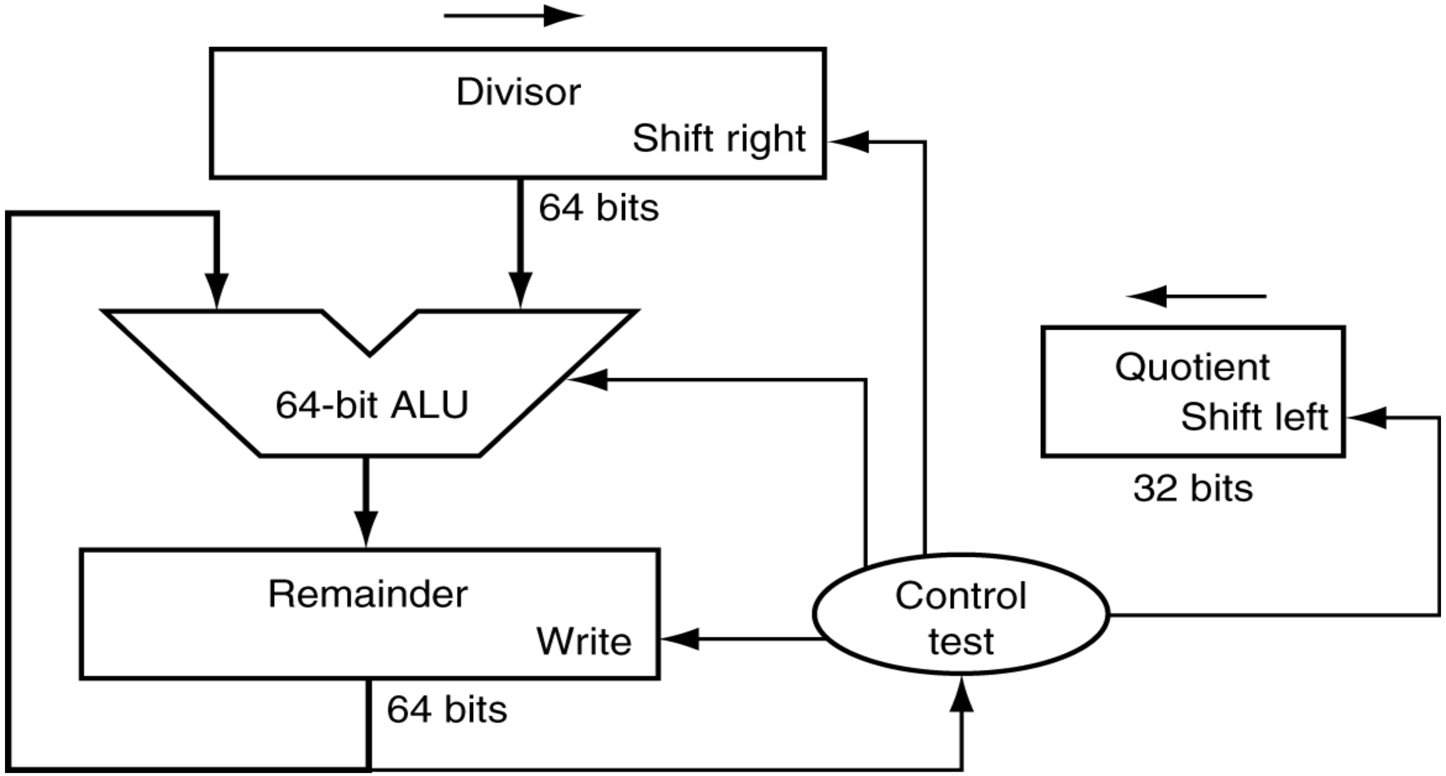
1010

- 1000

Remainder 10

Division Hardware





Division Hardware: Example

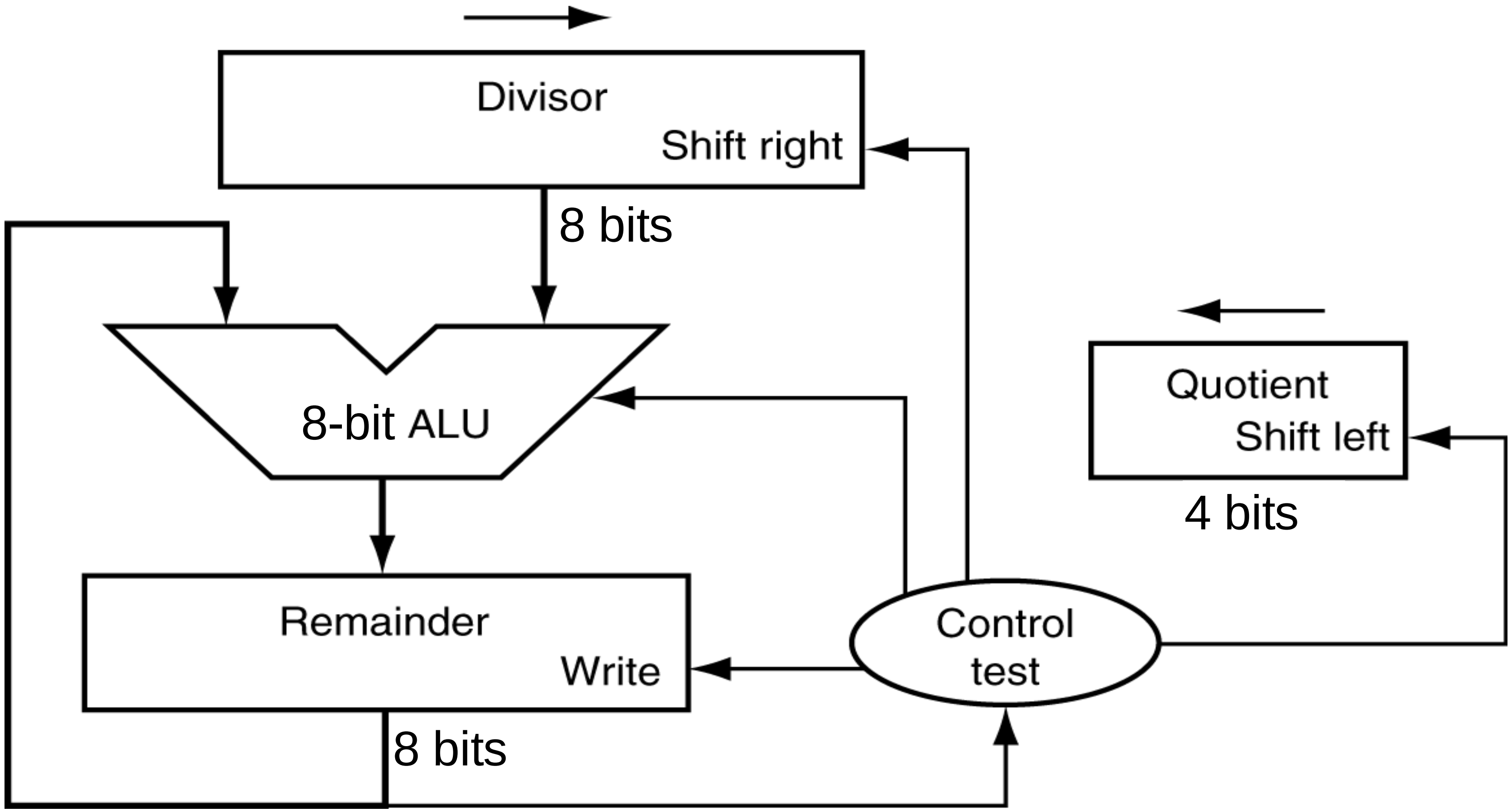
Quotient

Dividend

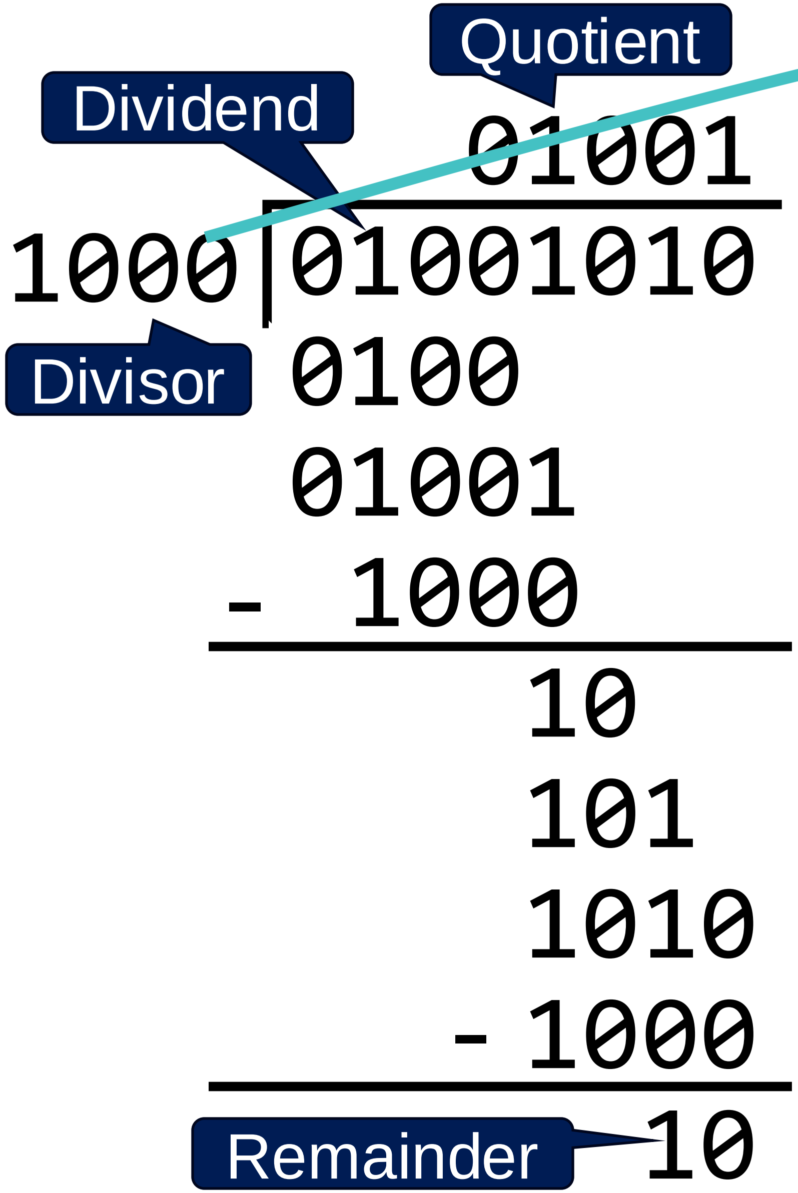
```
          01001
1000 |01001010
      0100
      01001
  -    1000
      -------
          10
          101
          1010
      - 1000
      -------
            10
```

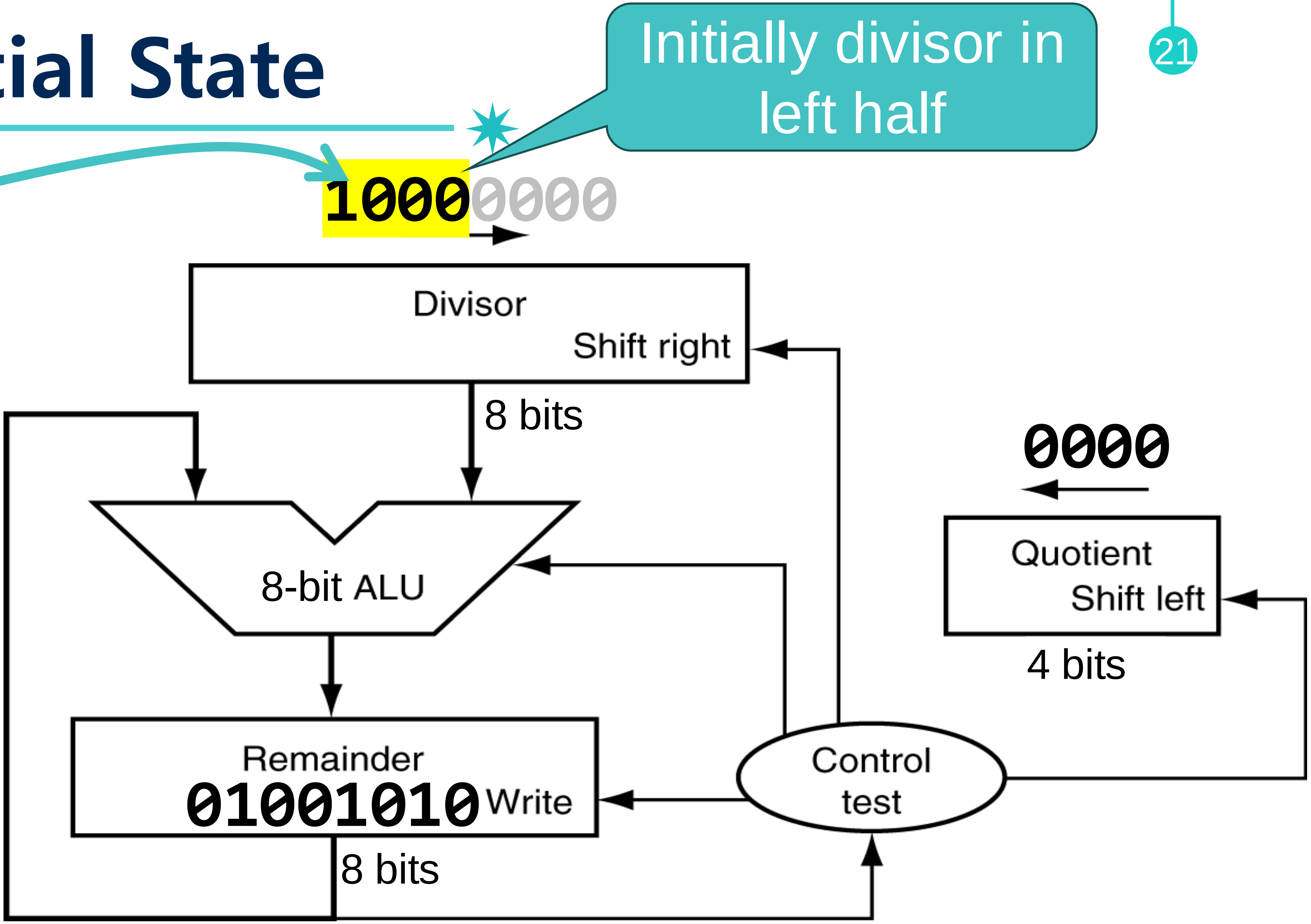
Divisor

Remainder



Example: Initial State





Example: Initial State

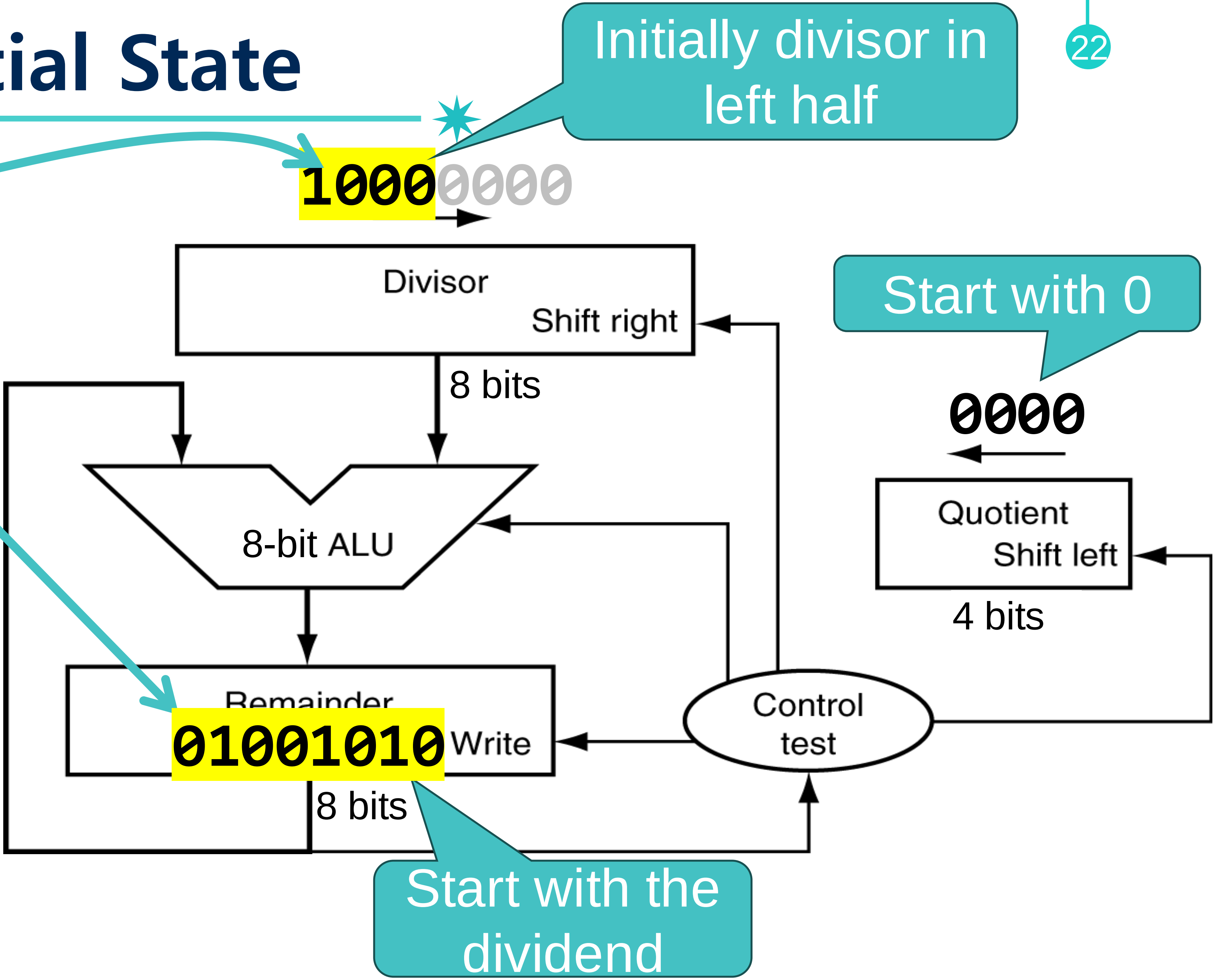
Quotient

Dividend

```
        01001
1000 |01001010
      0100
      01001
     -  1000
      ---------
           10
           101
           1010
         - 1000
      ---------
            10
```

Divisor

Remainder



Example: 1st Iteration - 1

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

```
remainder = remainder - divisor;
```

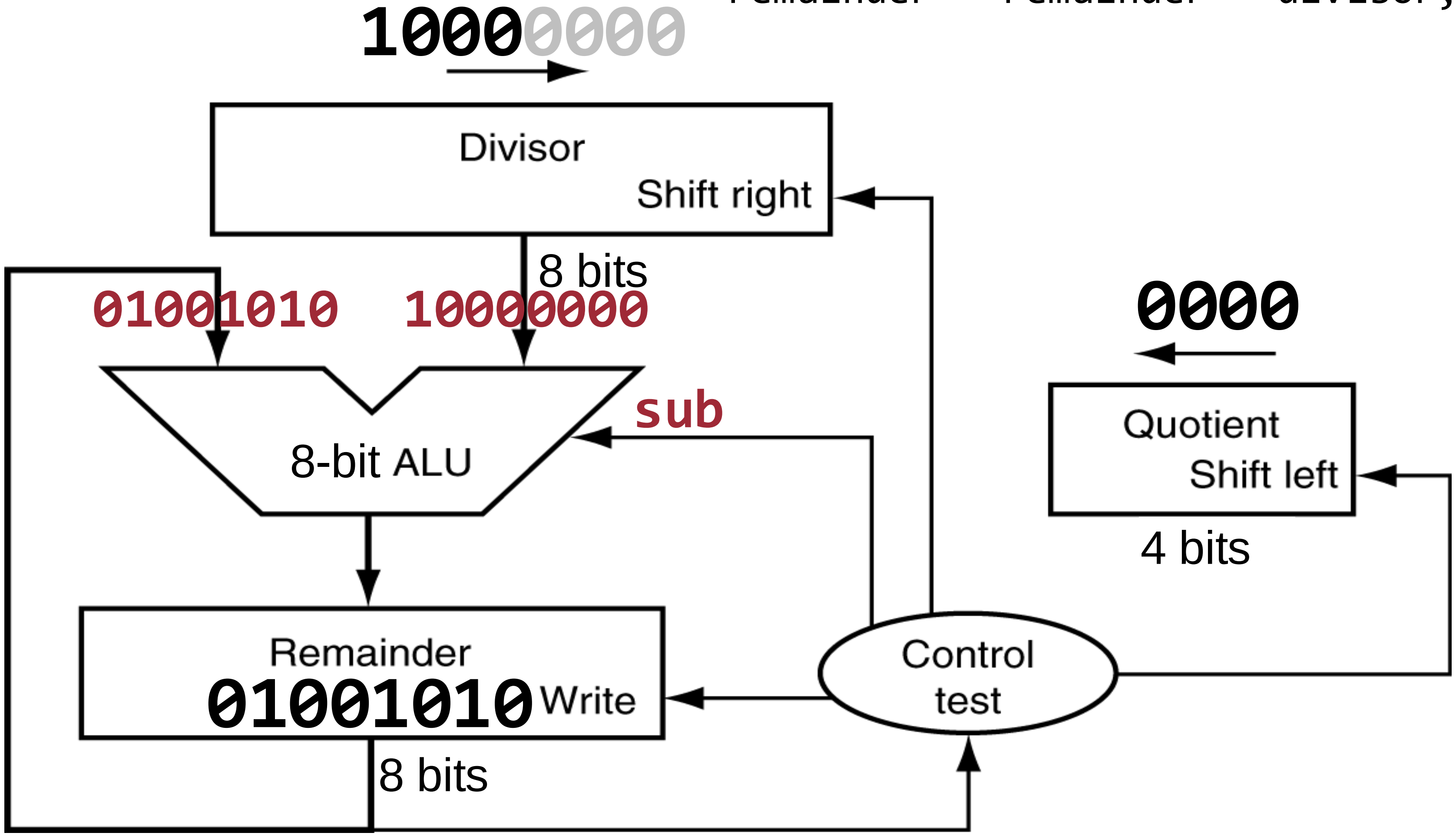
Quotient

Dividend

```
          01001
1000 | 01001010
       0100
       01001
     -  1000
     -------
          10
          101
          1010
        - 1000
     -------
            10
```

Divisor

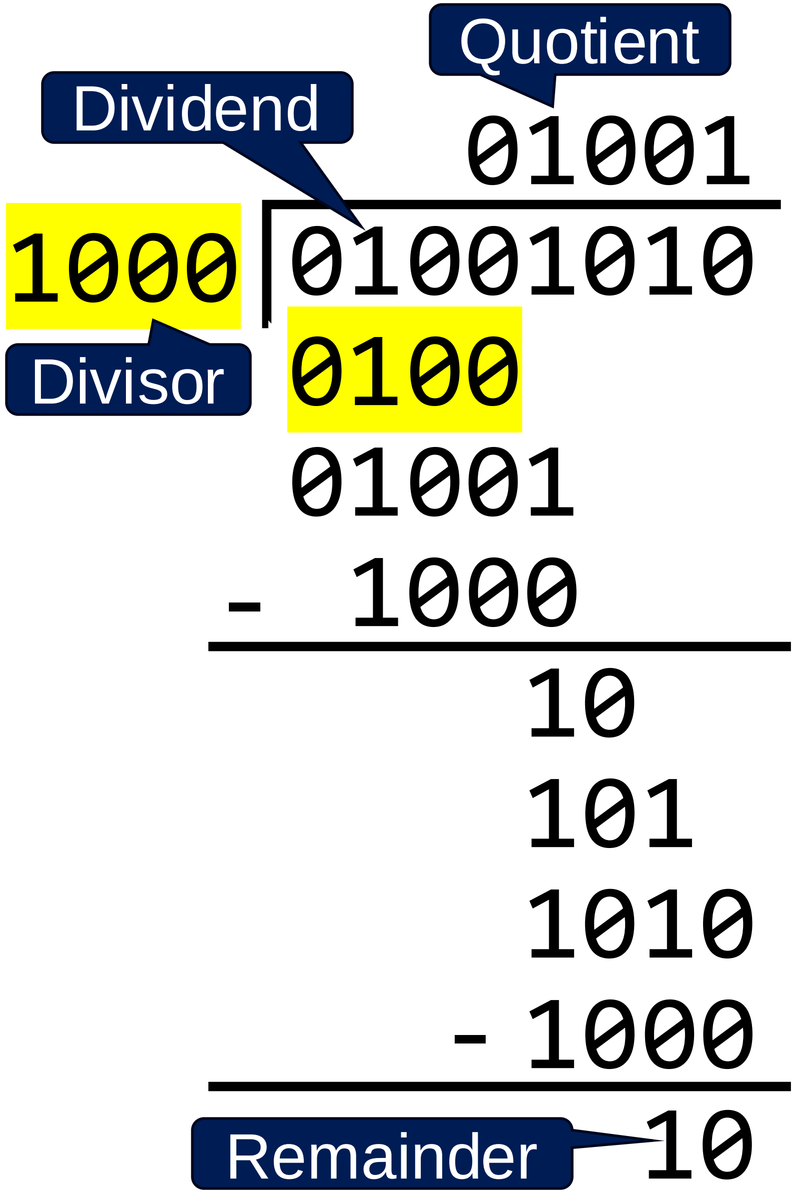
Remainder

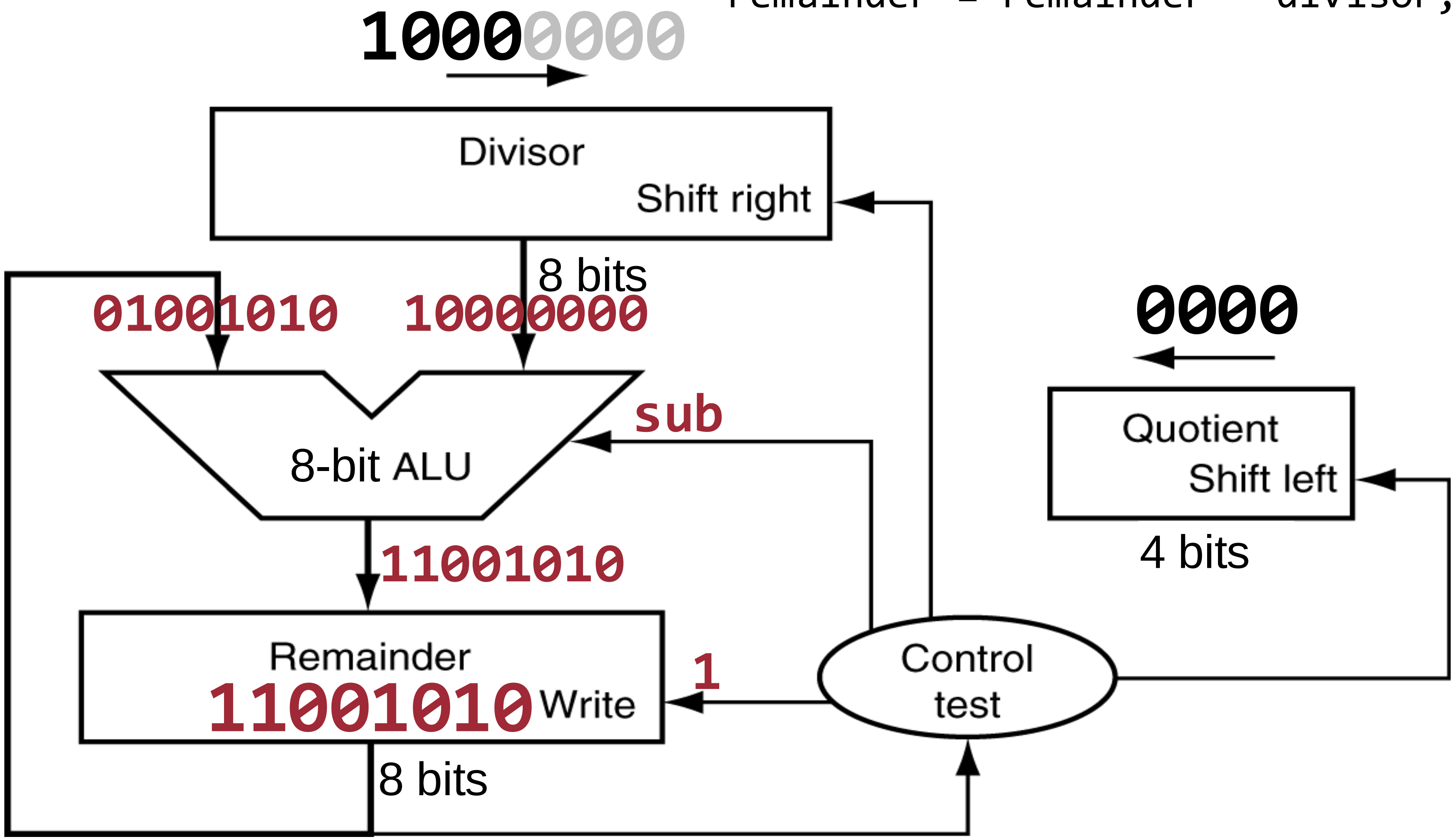


Example: 1st Iteration - 1

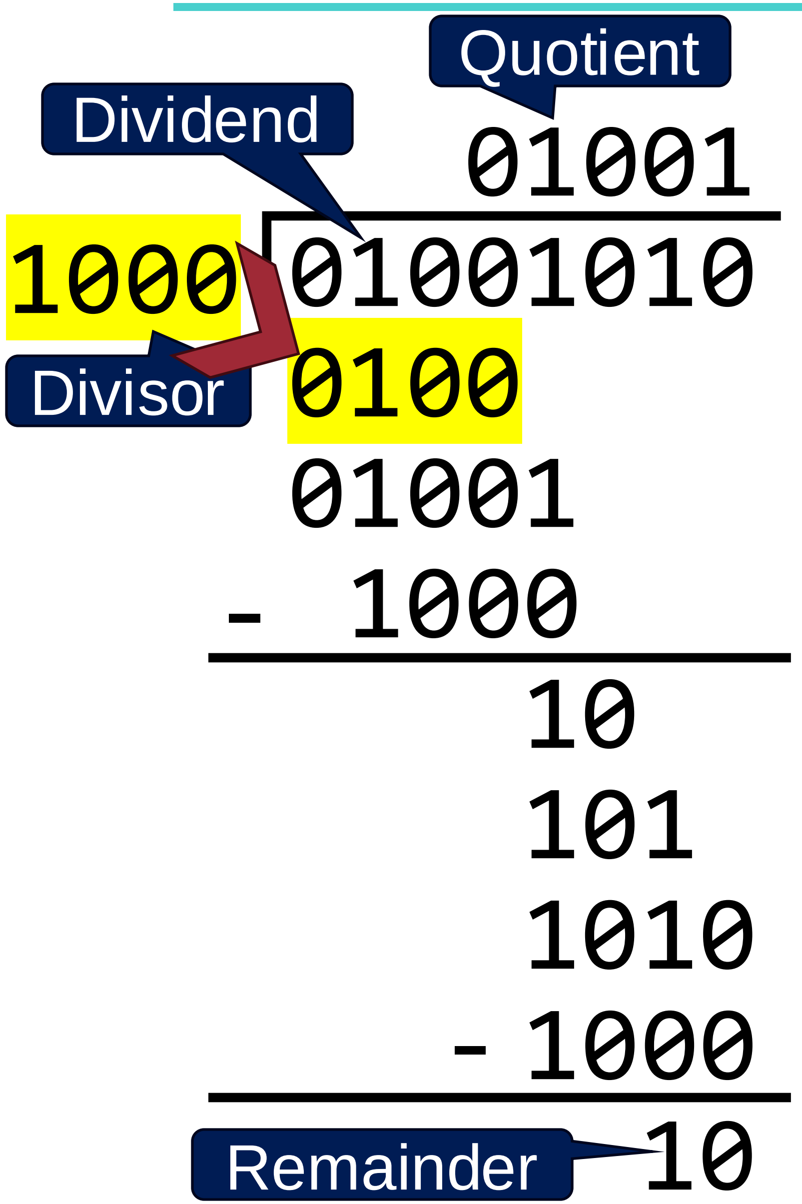
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

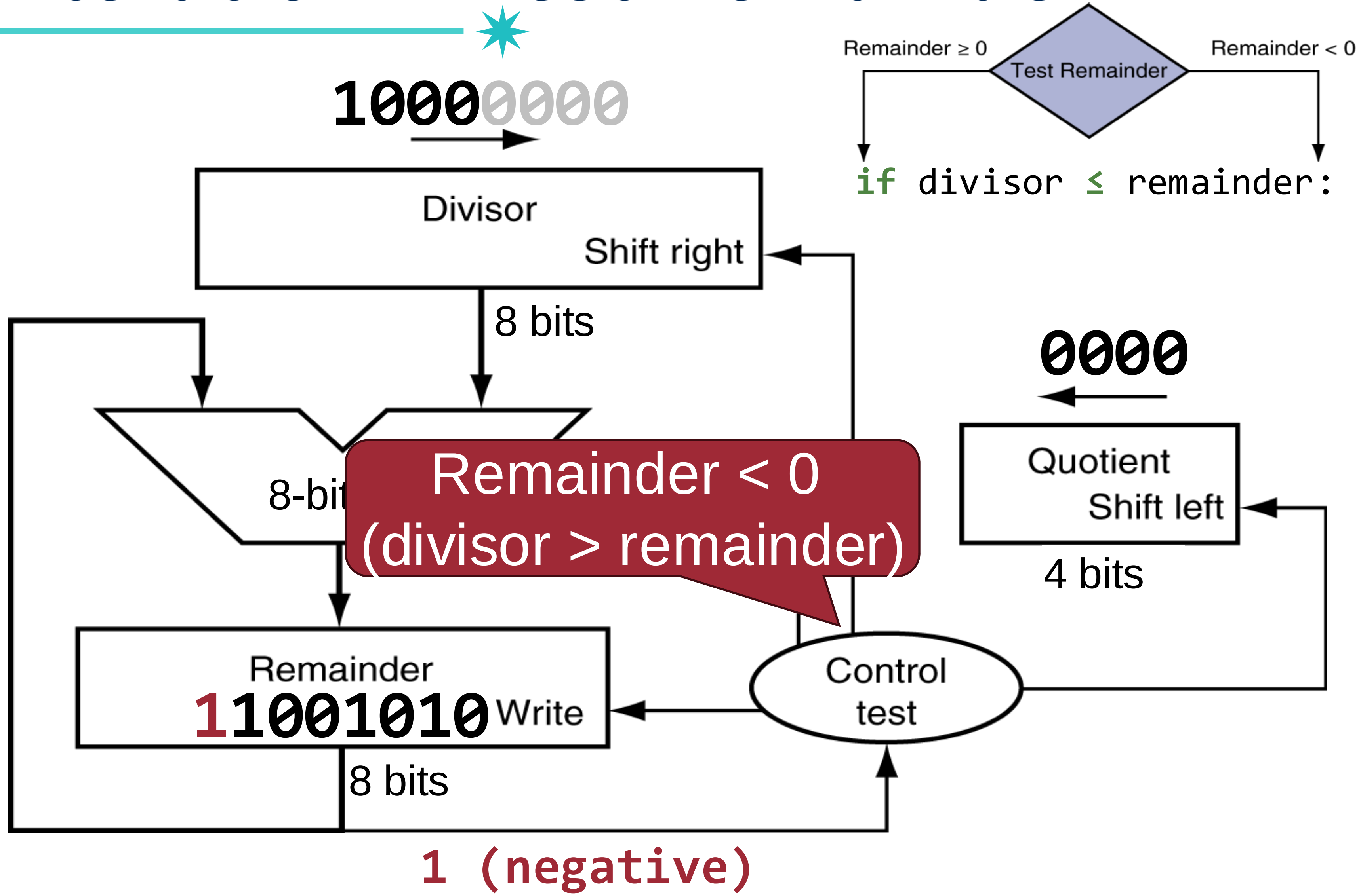
```
remainder = remainder - divisor;
```





Example: 1st Iteration – Test Remainder

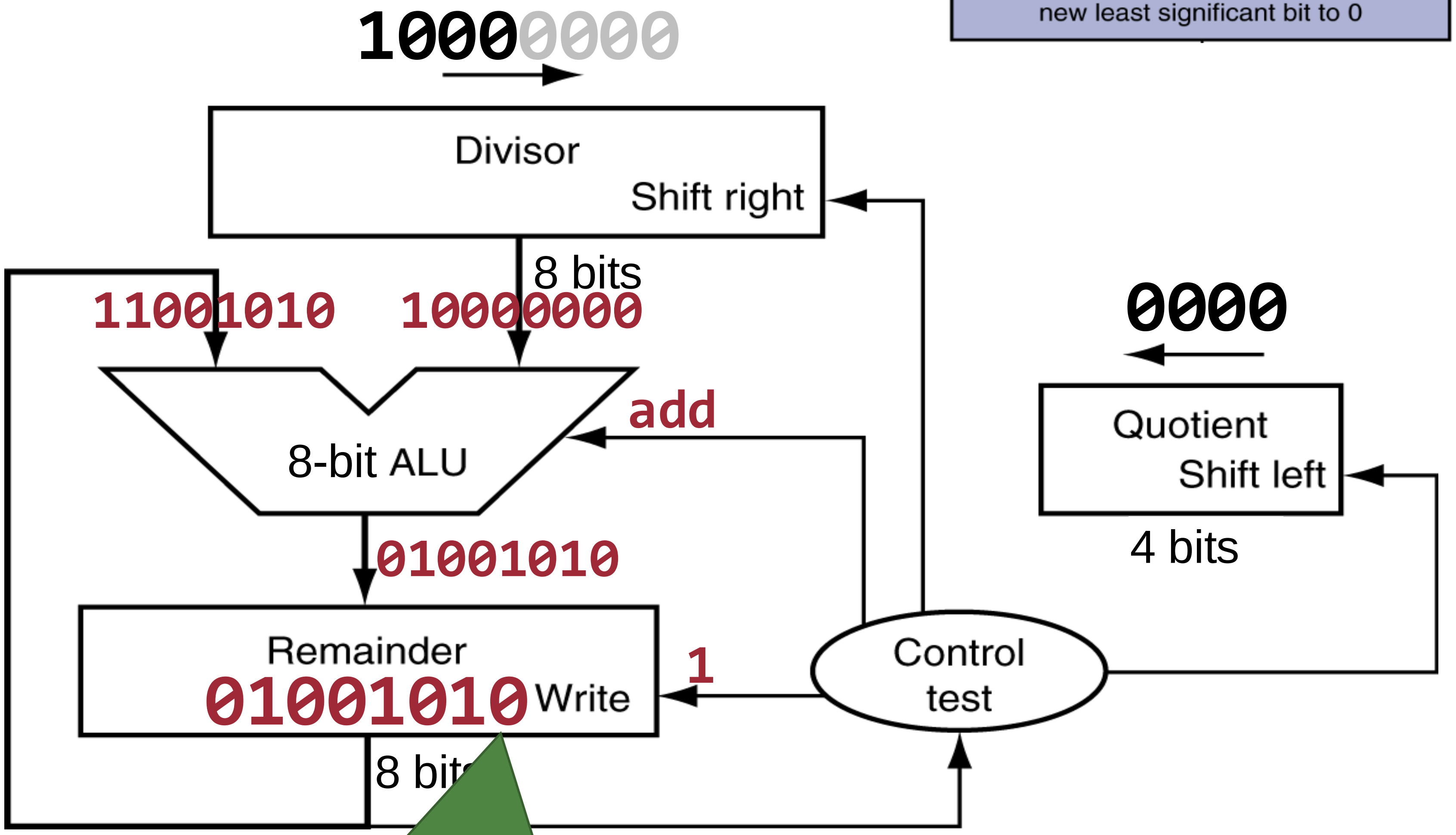




Example: 1st Iteration – 2b

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

```
Quotient
           01001
Dividend
1000 | 01001010
Divisor
       0100
       01001
     -  1000
           10
           101
           1010
         - 1000
Remainder  10
```



Restore the original remainder

Example: 1st Iteration – 2b

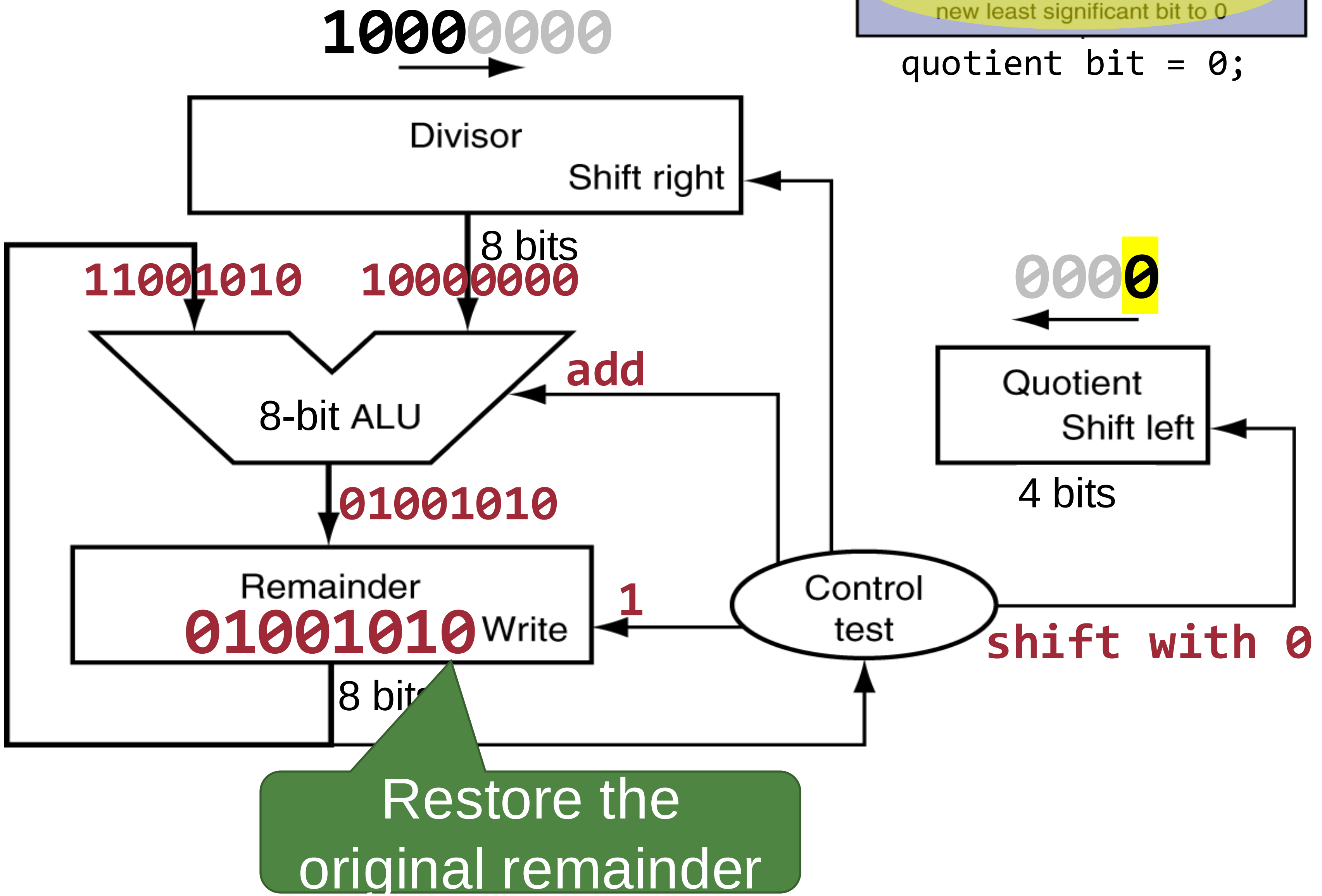
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

```
quotient bit = 0;
```

Quotient
Dividend

```
        01001
1000 | 01001010
       0100
       01001
     -  1000
           10
           101
           1010
         - 1000
             10
```

Divisor
Remainder

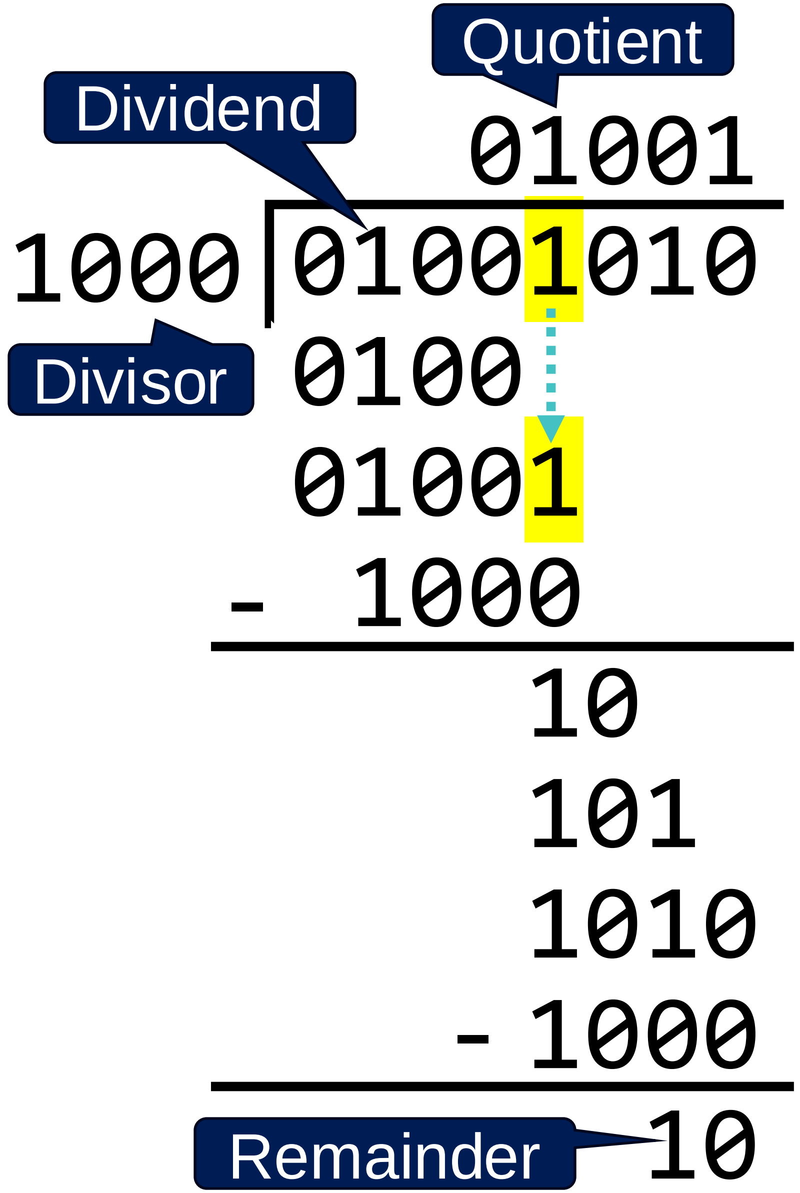


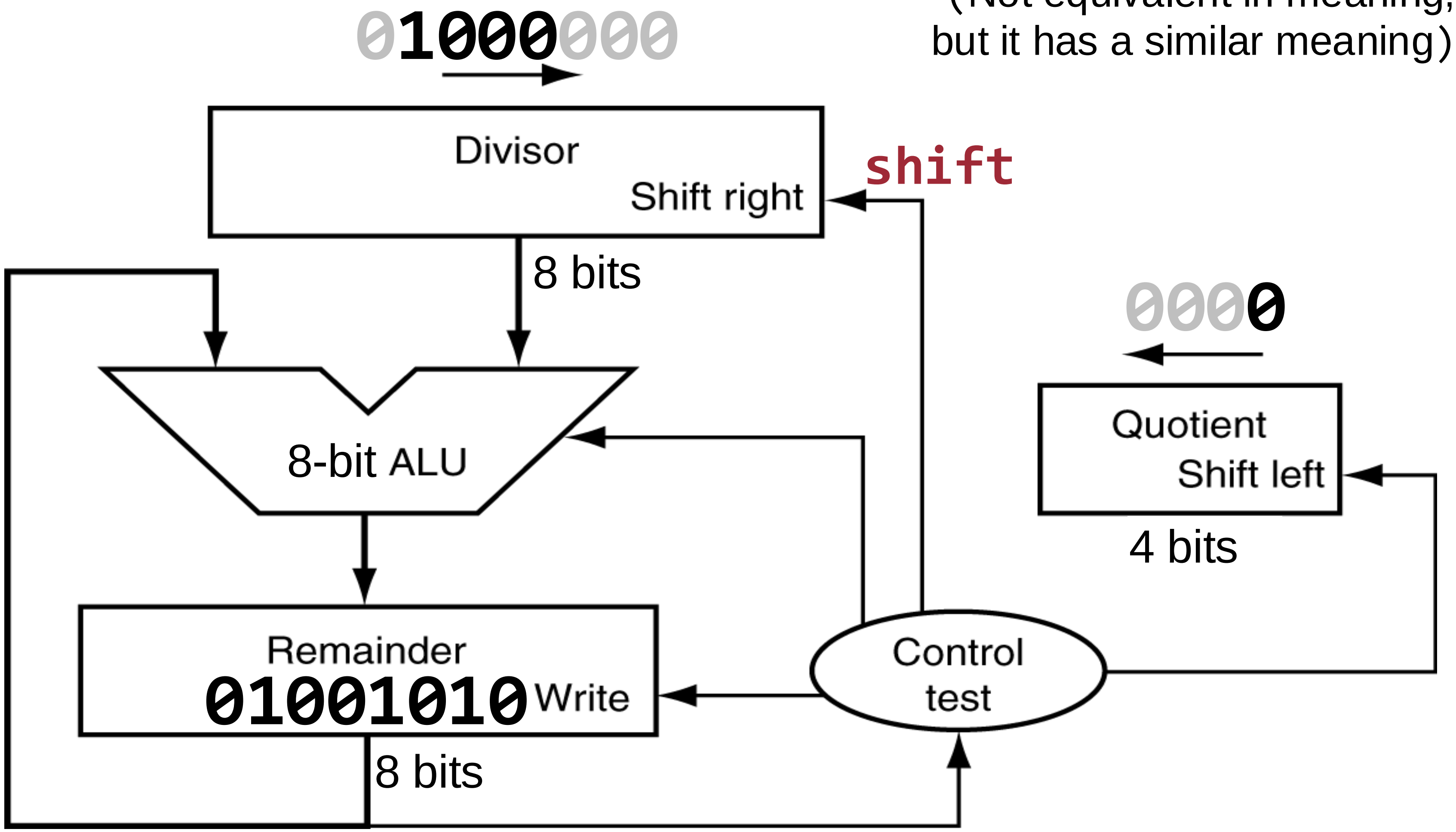
Example: 1st Iteration – 3

3. Shift the Divisor register right 1 bit

bring down next dividend bit to remainder

(Not equivalent in meaning, but it has a similar meaning)

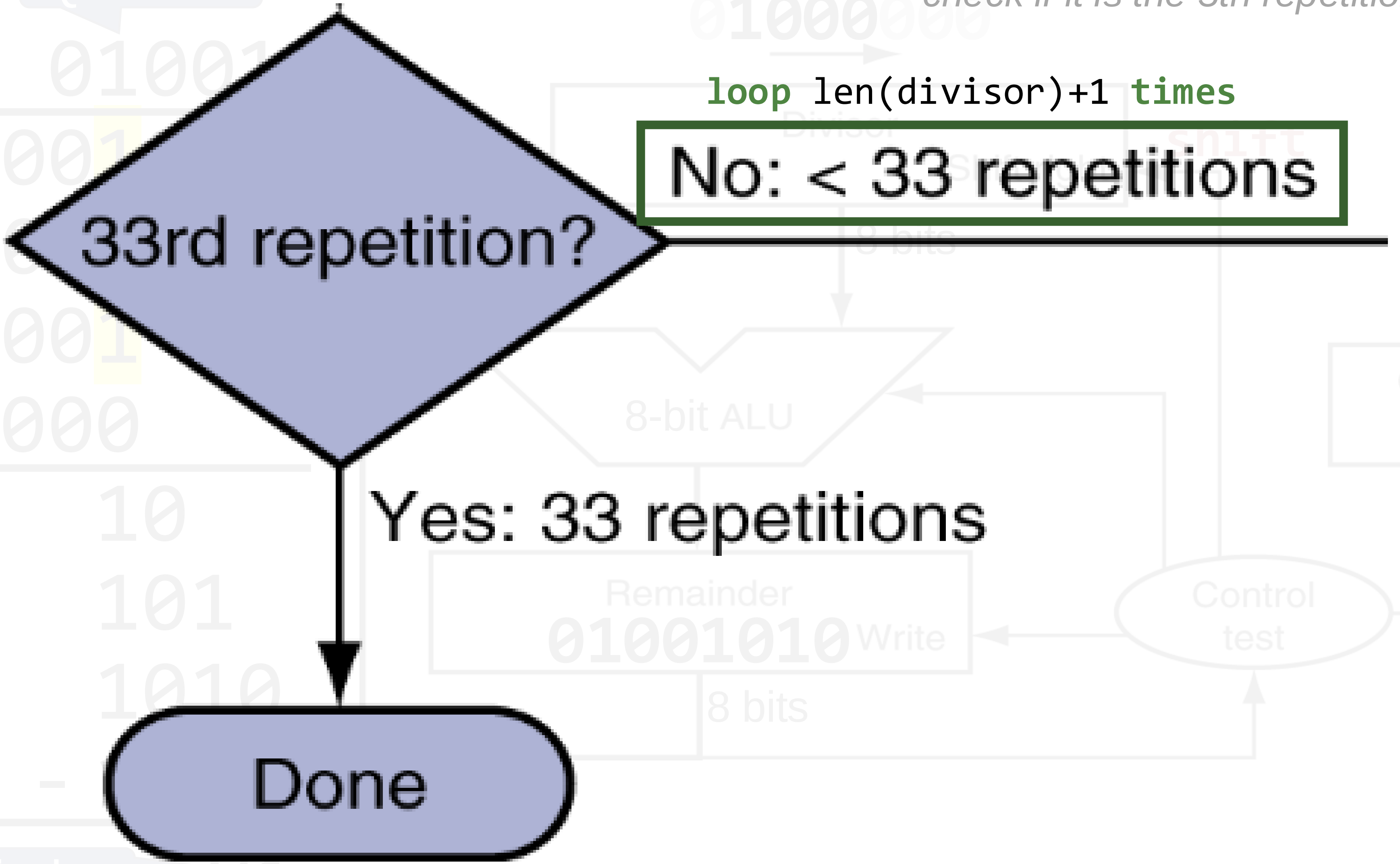




Example: 1st Iteration – 33rd Repetition?

In this example, we should check if it is the 5th repetition

`loop len(divisor)+1 times`



Example: 2nd Iteration – 1

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

```
remainder = remainder - divisor;
```

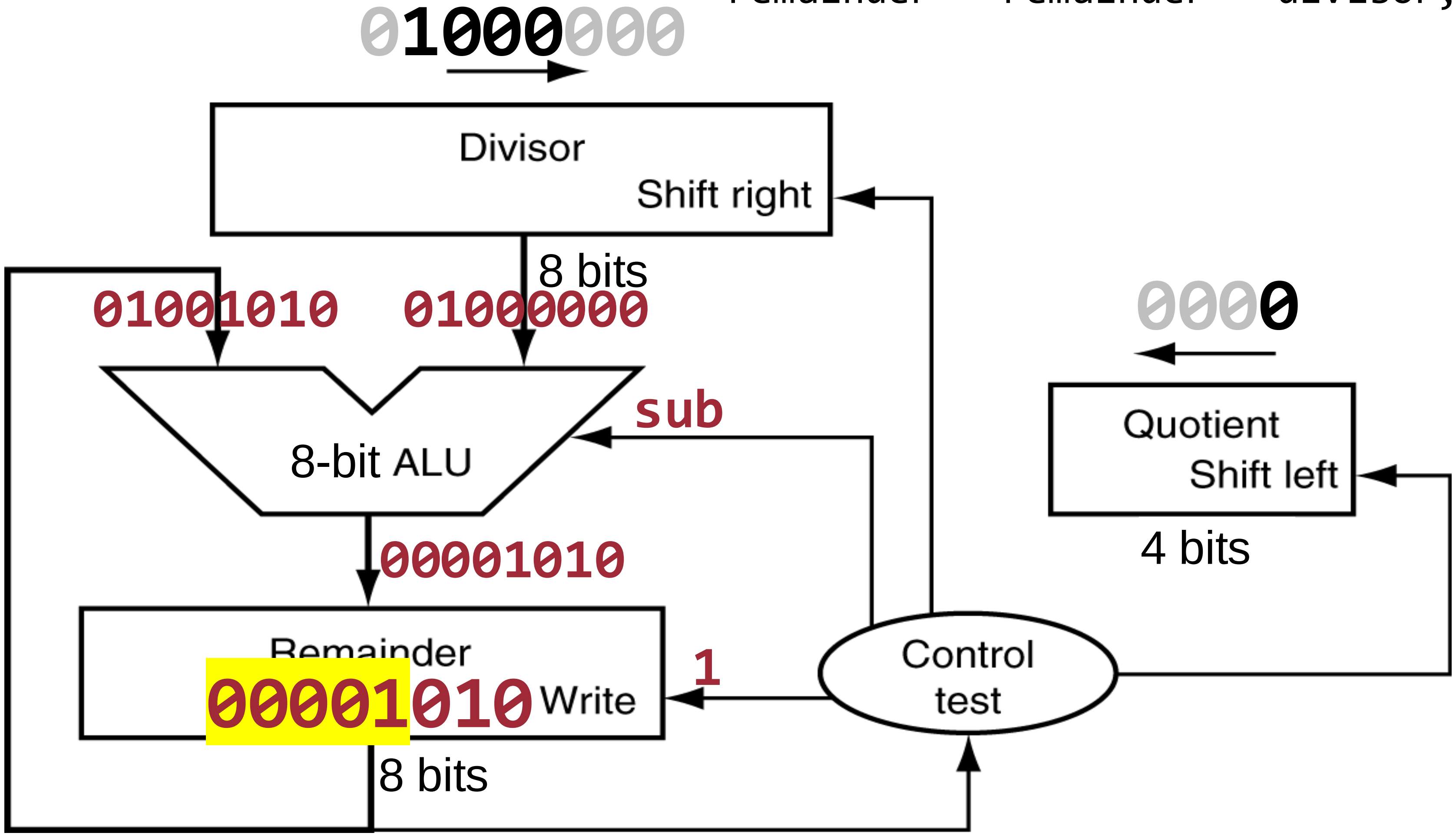
Quotient

Dividend

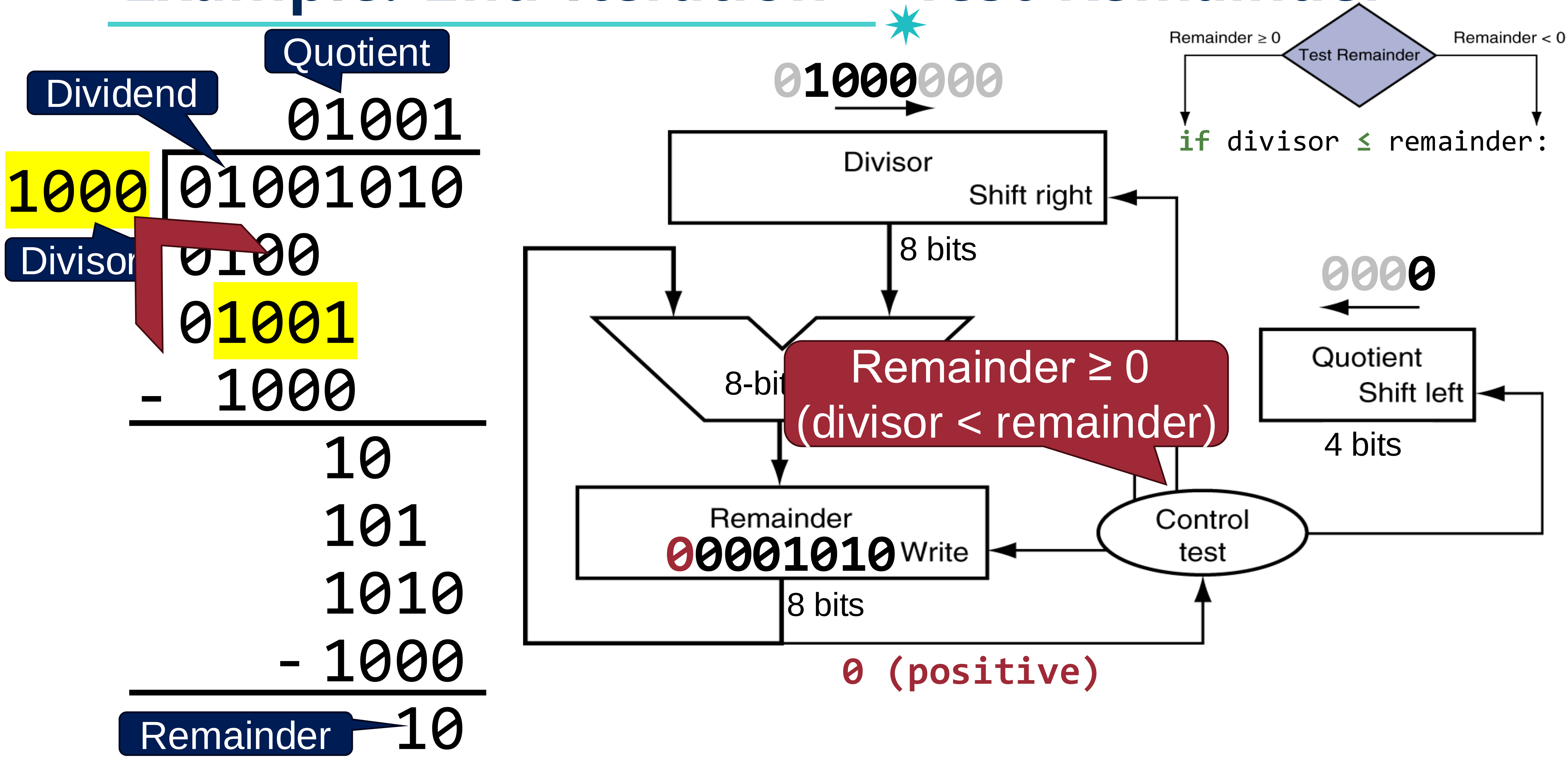
```
        01001
1000 | 01001010
       0100
       01001
    -   1000
           10
           101
           1010
         - 1000
             10
```

Divisor

Remainder



Example: 2nd Iteration – Test Remainder



Example: 2nd Iteration – 2a

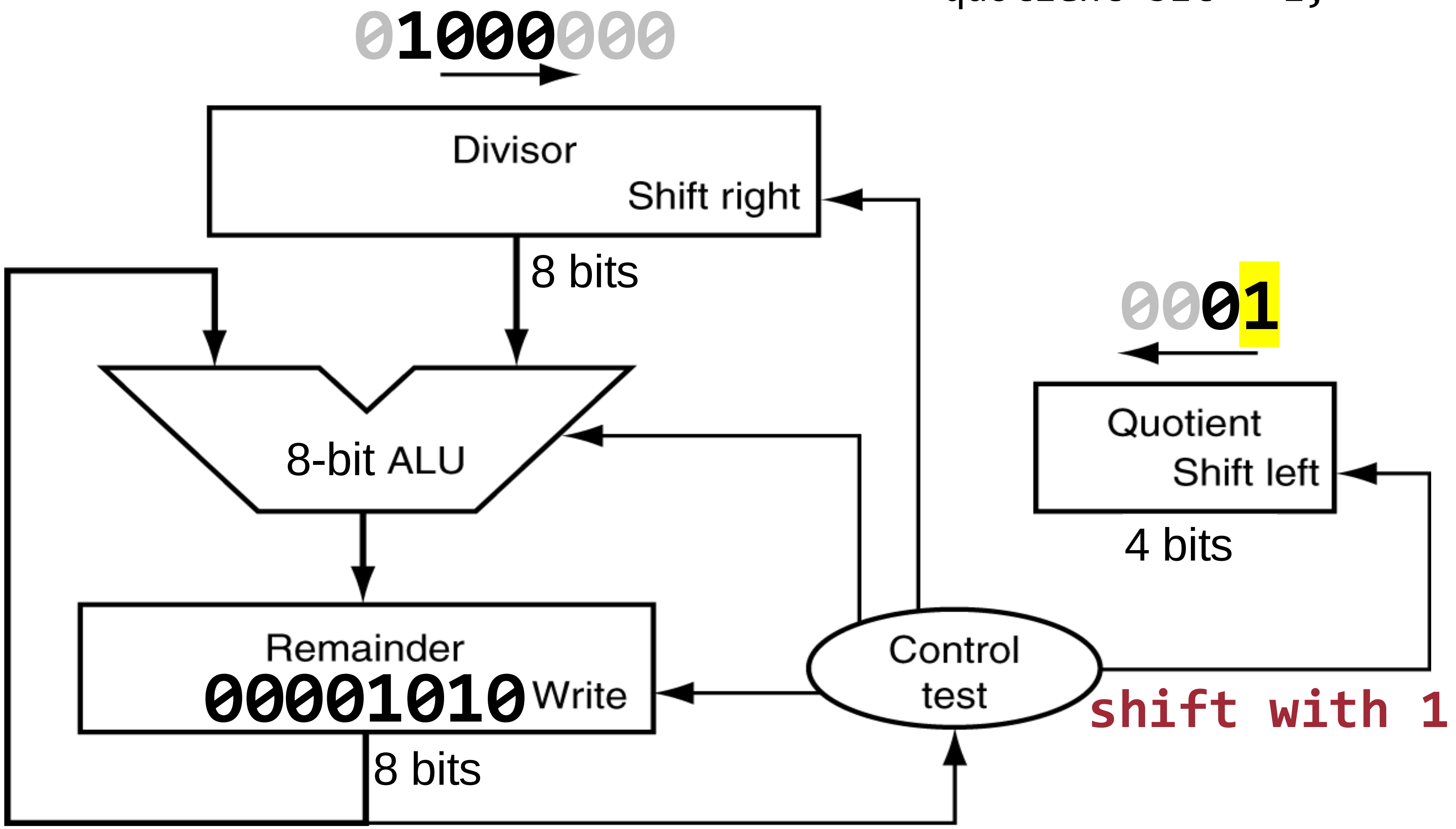
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

```
quotient bit = 1;
```

Quotient
Dividend

```
          01001
1000 | 01001010
       0100
       01001
     -  1000
       -----
          10
          101
          1010
        - 1000
        ------
            10
```

Divisor
Remainder

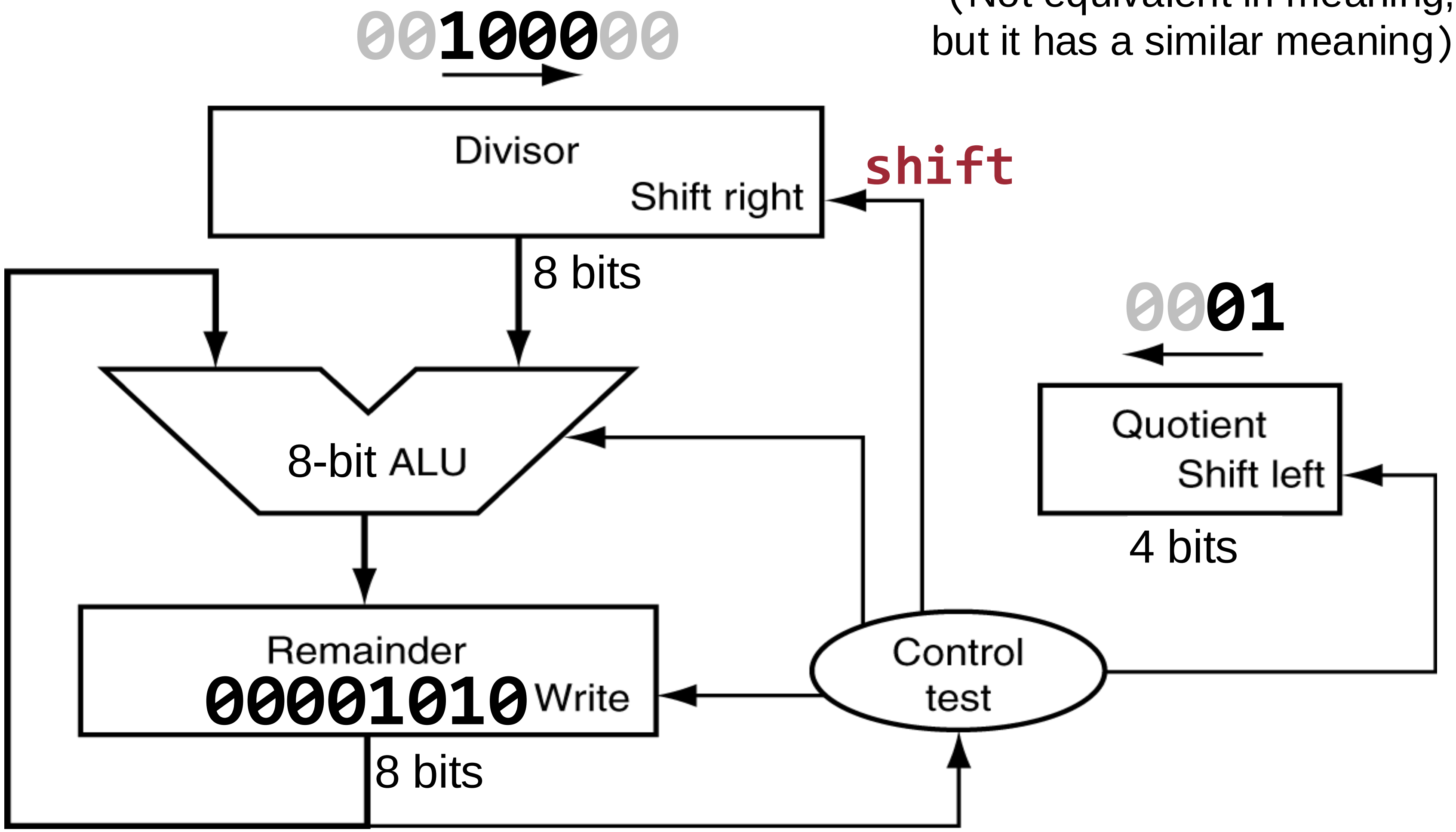


Example: 2nd Iteration – 3

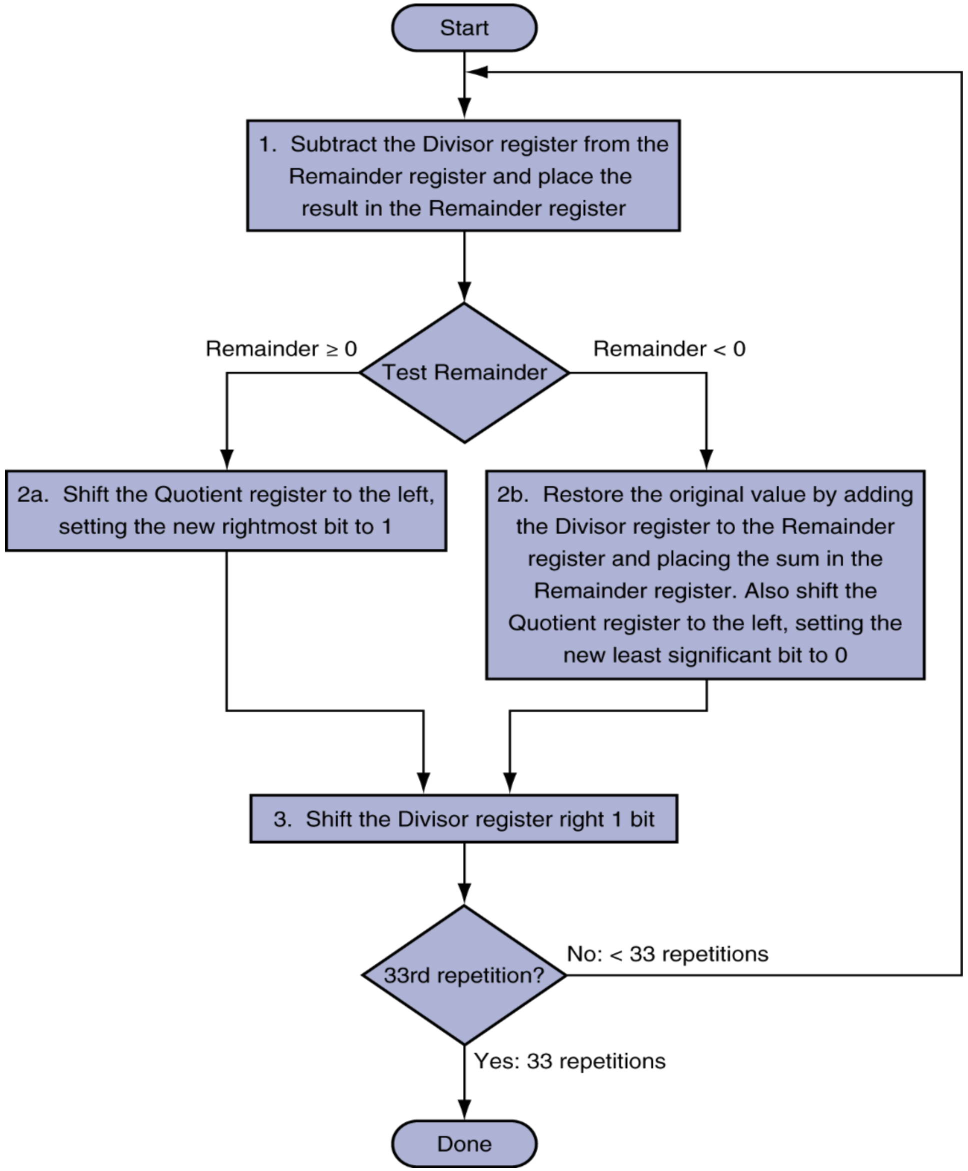
3. Shift the Divisor register right 1 bit

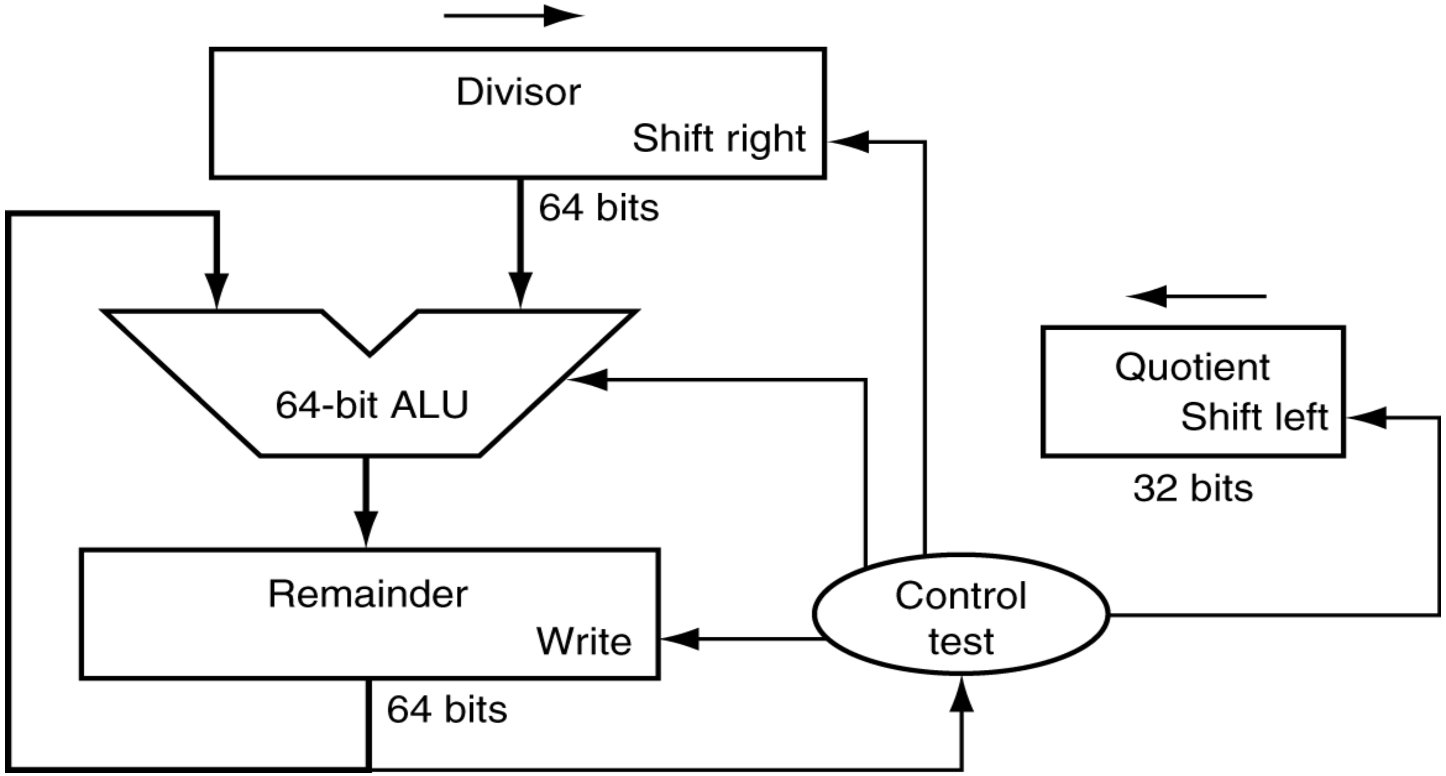
bring down next dividend bit to remainder
(Not equivalent in meaning, but it has a similar meaning)

```
Quotient
Dividend          01001
1000 | 01001010
Divisor 0100
        01001
     -   1000
     ---------
            10
            101
            1010
          - 1000
     ---------
Remainder   10
```



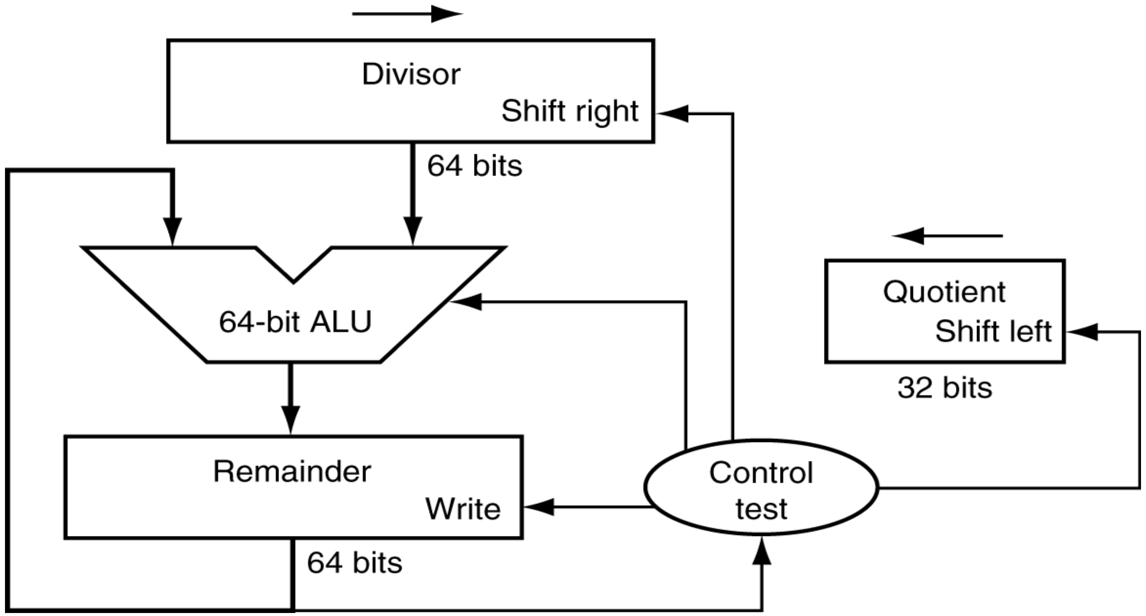
Division Hardware: Algorithm Summary

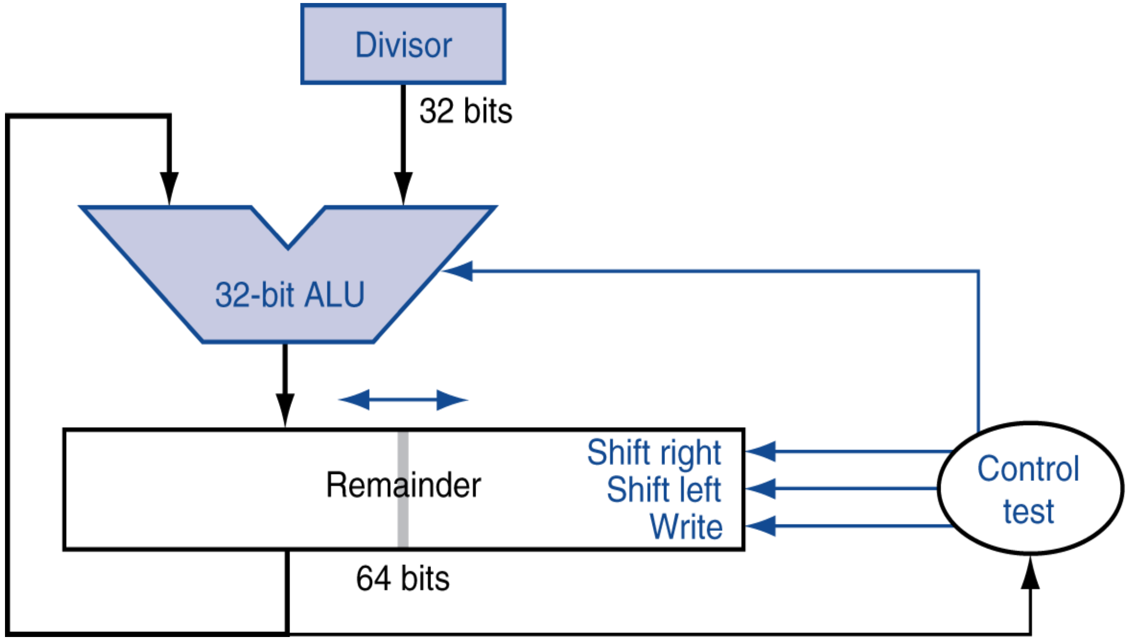




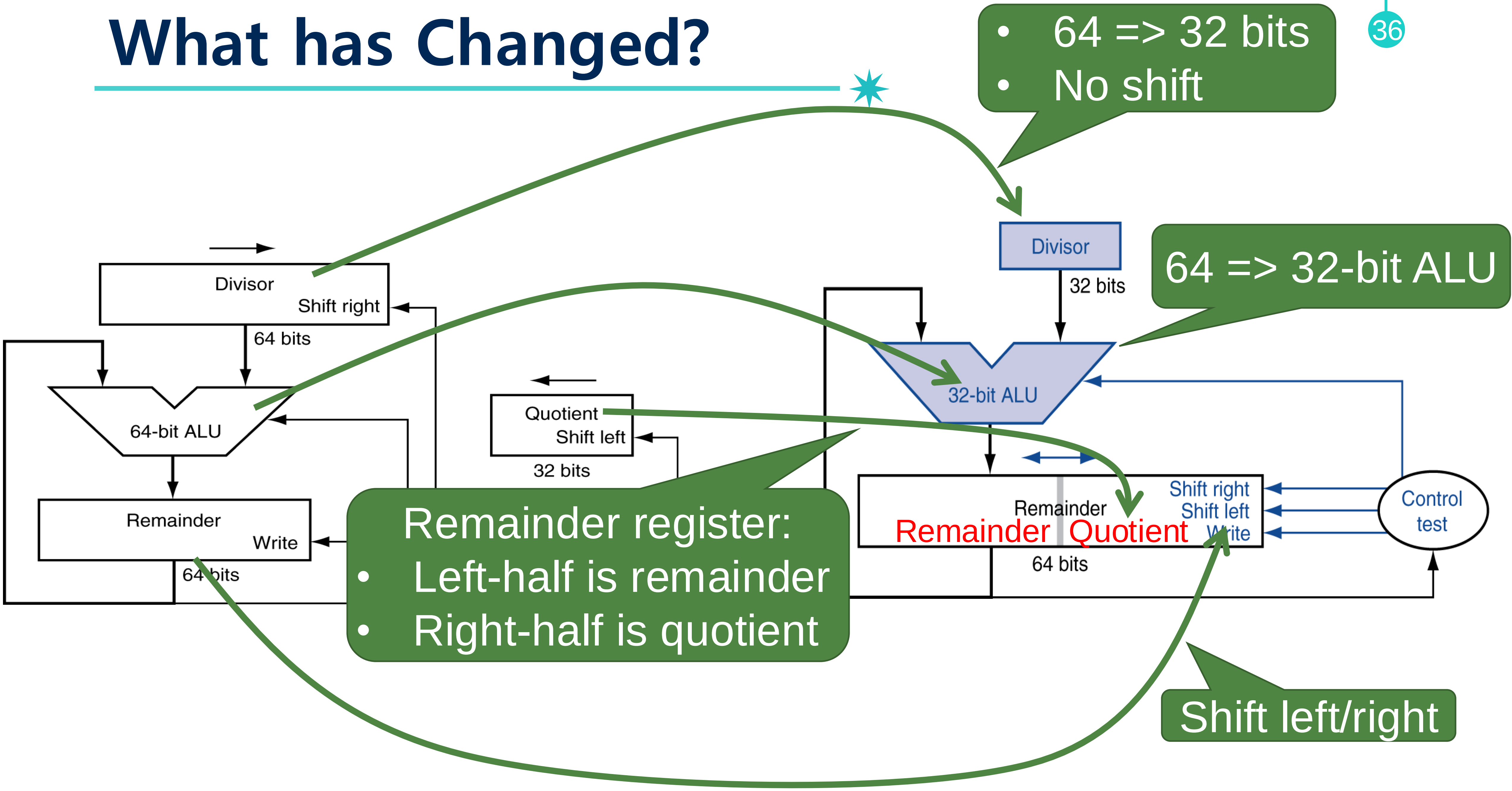
Can we optimize this division hardware?

Optimized Divider



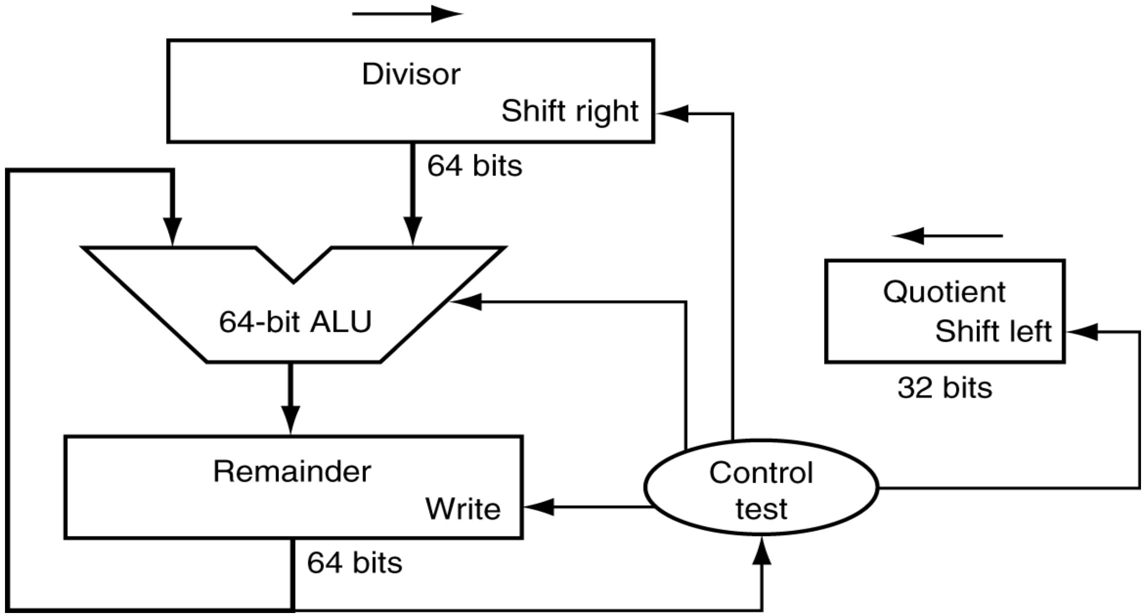


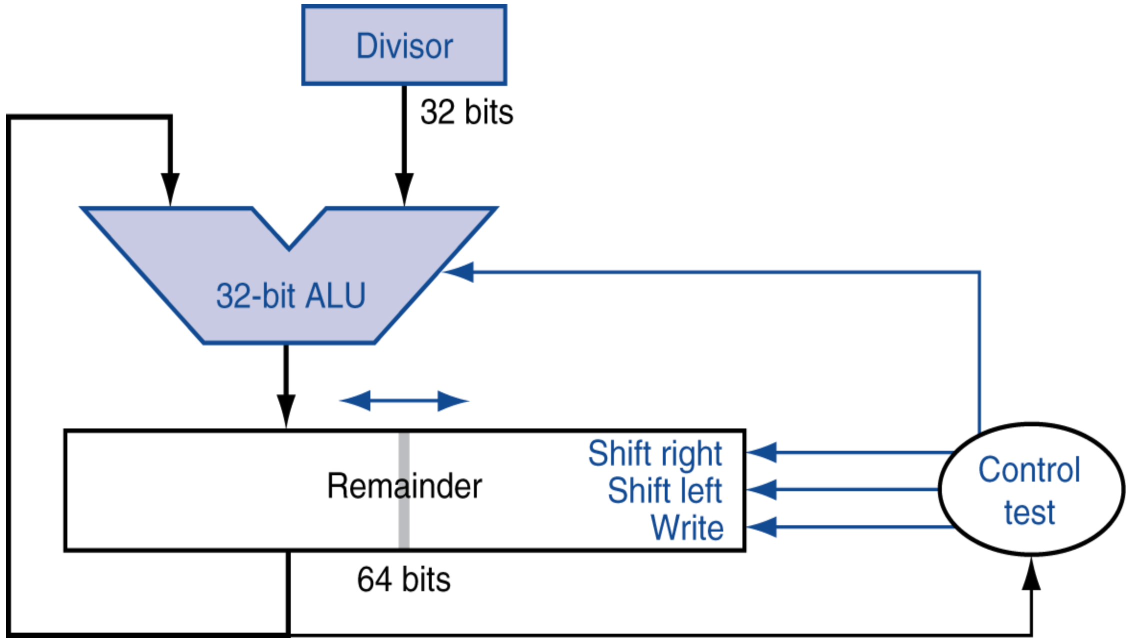
What has Changed?



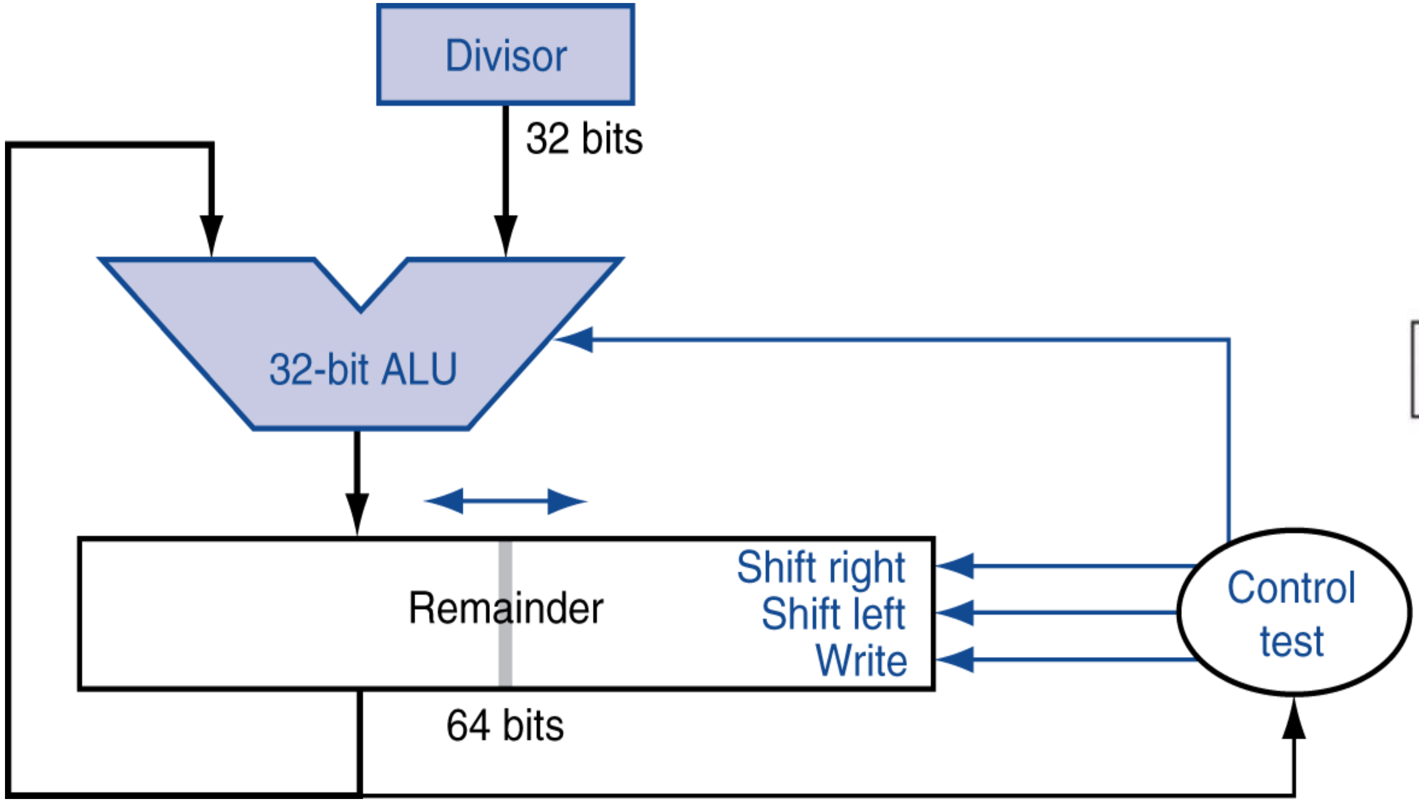
Optimized Divider – Performance Gained

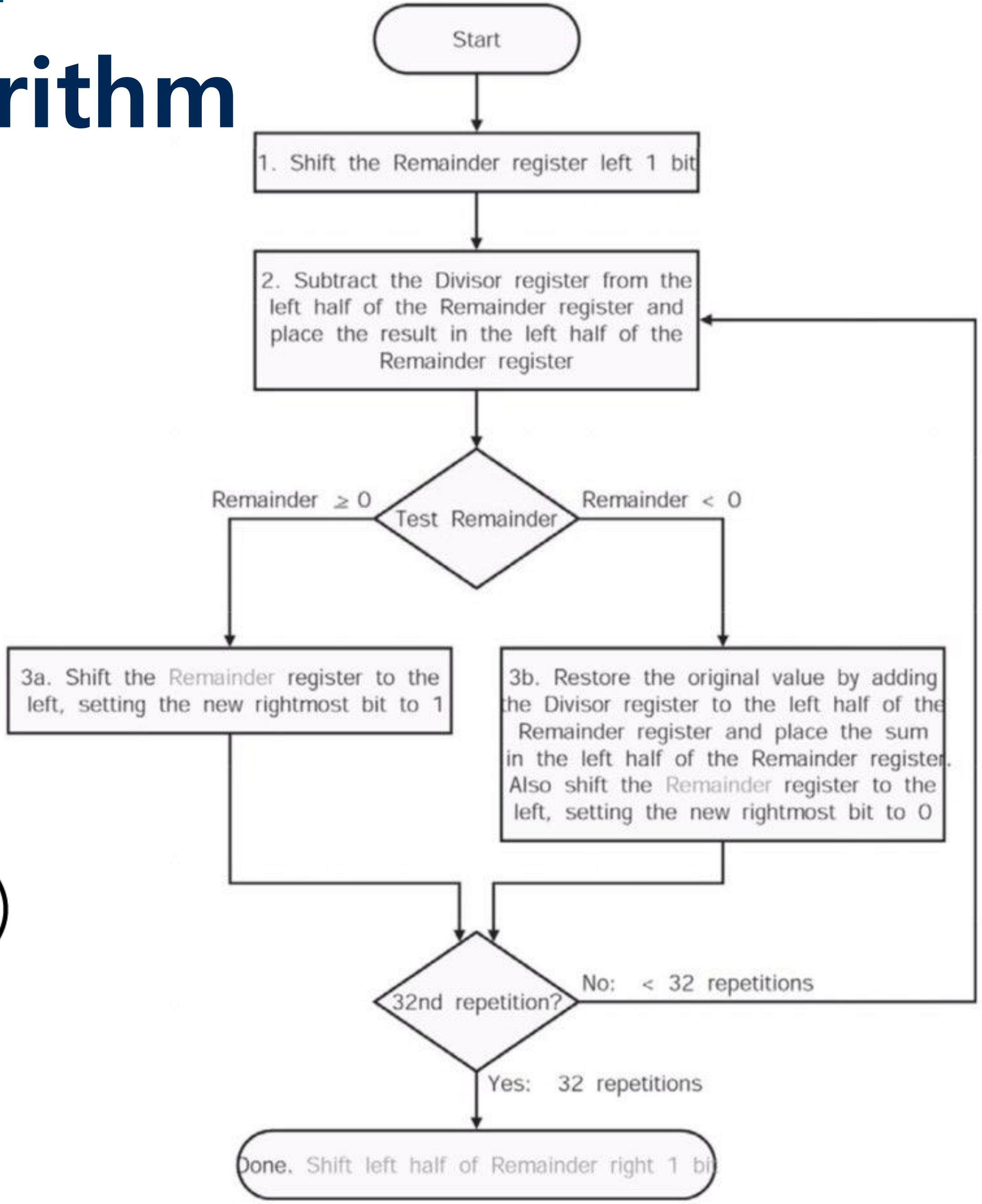
- Looks a lot like an optimized multiplier!
- The hardware is optimized to halve the width of the ALU and registers (64 bits ⇒ 32 bits, Clock cycle time ↓)
- Perform steps in parallel: add/shift (# of clock cycle ↓)





Optimized Divider: Algorithm





Optimized Divider: Example

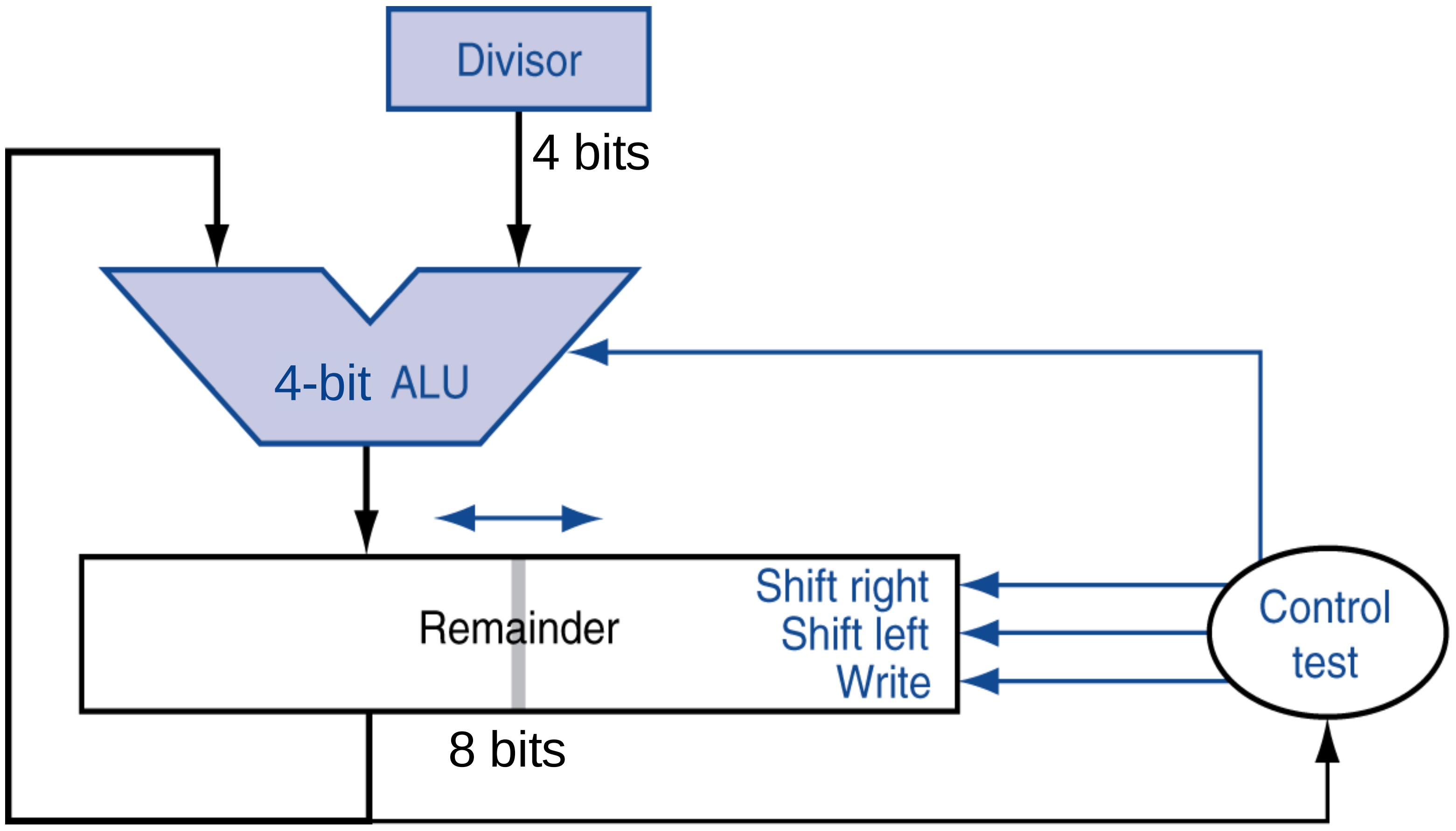
Quotient

Dividend

```
       0011
     ______
0010 |0111
      0
      01
      011
    - 010
    ------
        11
      - 10
    ------
        01
```

Divisor

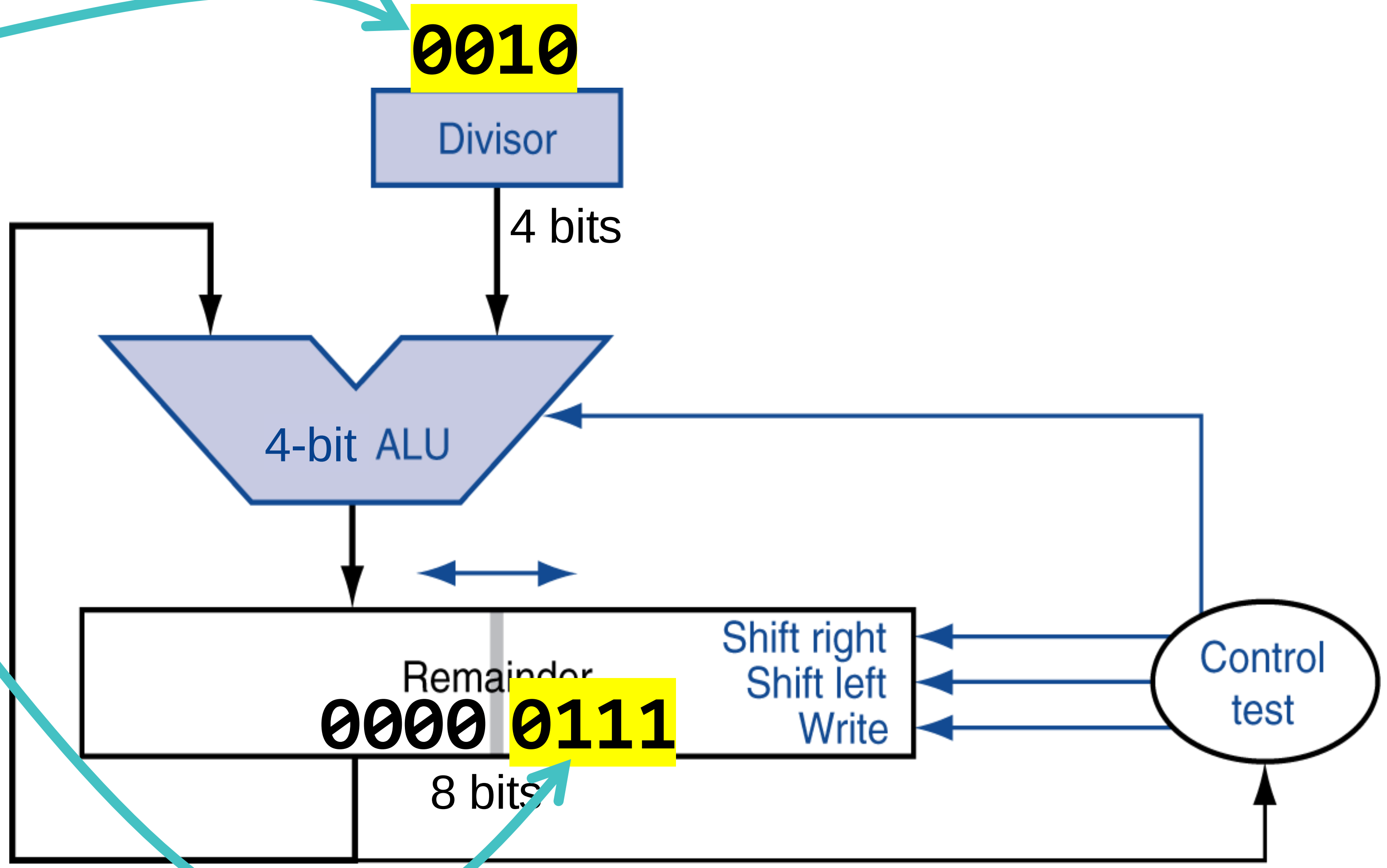
Remainder



Example: Initial State

Quotient

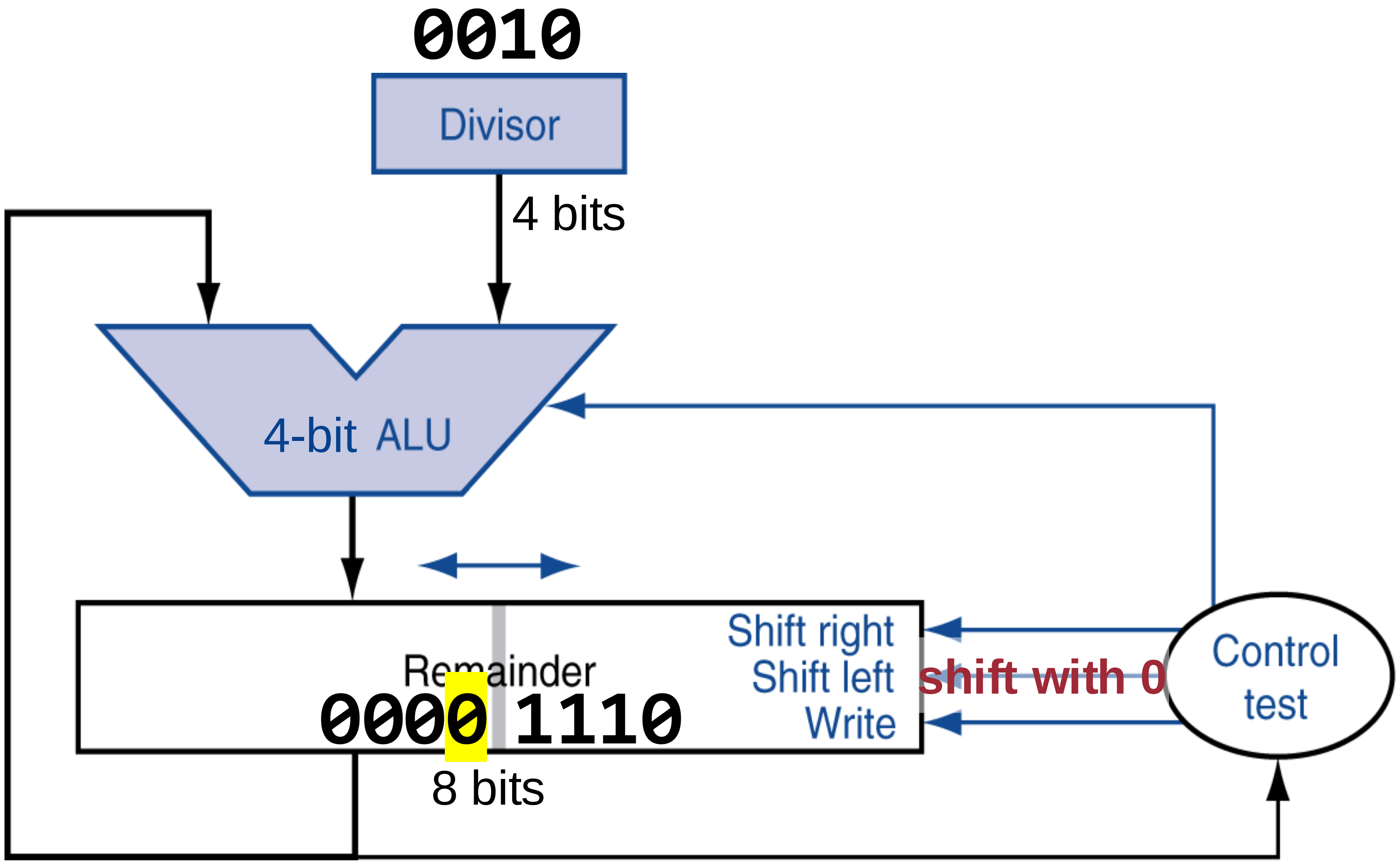
```
Dividend       0011
          0010|0111
Divisor        0
               01
               011
             - 010
             -----
                 11
               - 10
             -----
Remainder        01
```



Example: 1

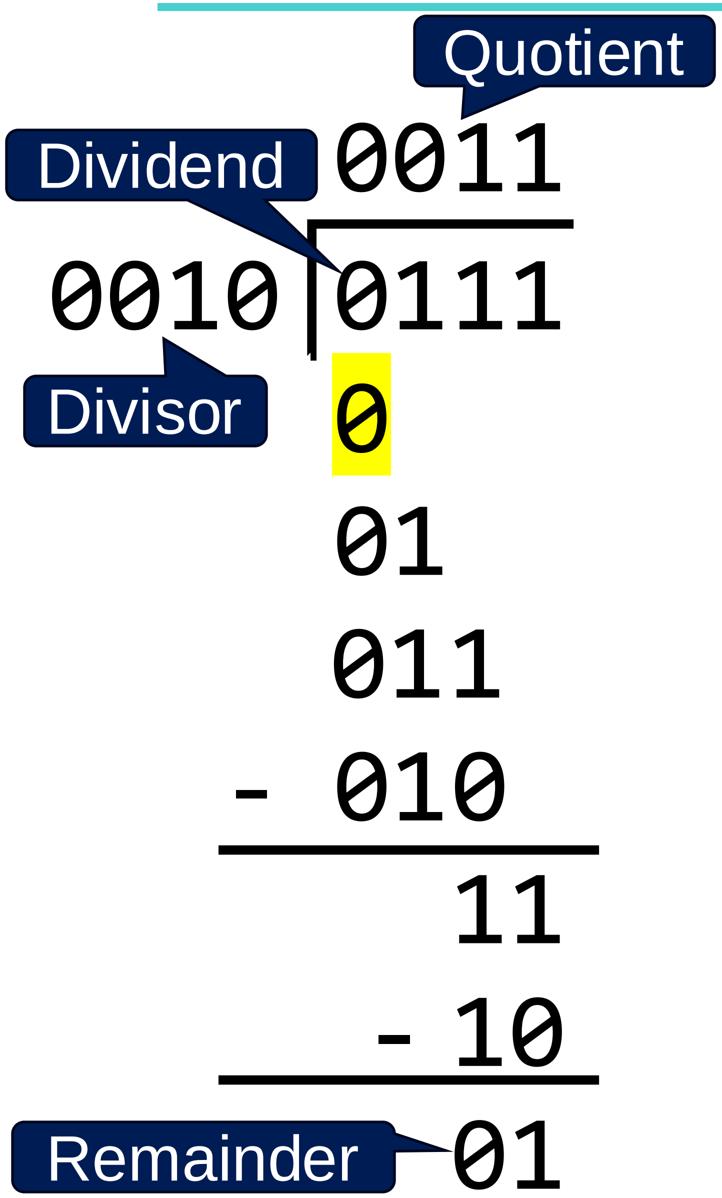
1. Shift the Remainder register left 1 bit

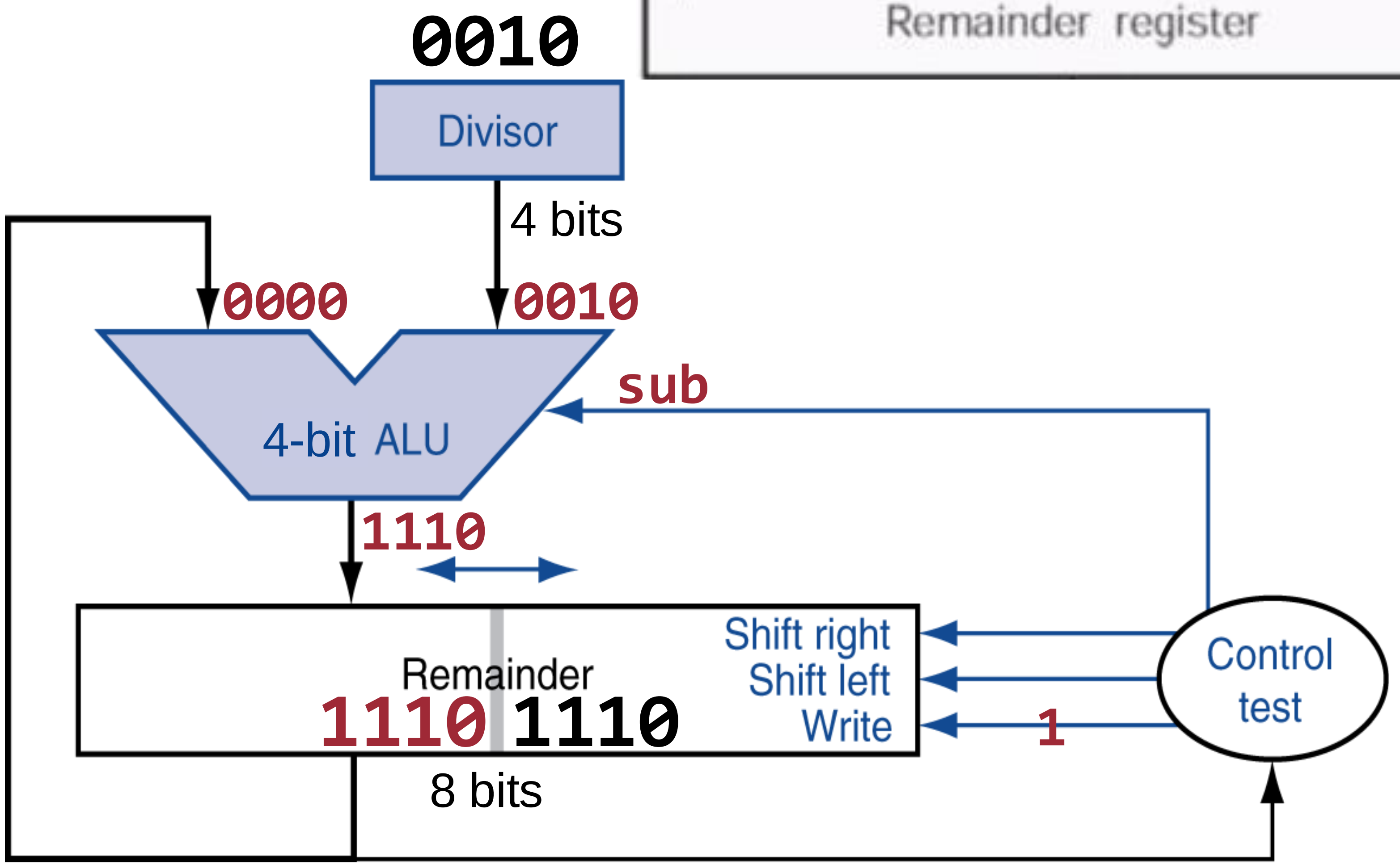
```
                Quotient
                  0011
Dividend        ______
         0010 | 0111
Divisor         0
                01
                011
              - 010
              ______
                  11
                - 10
              ______
Remainder         01
```



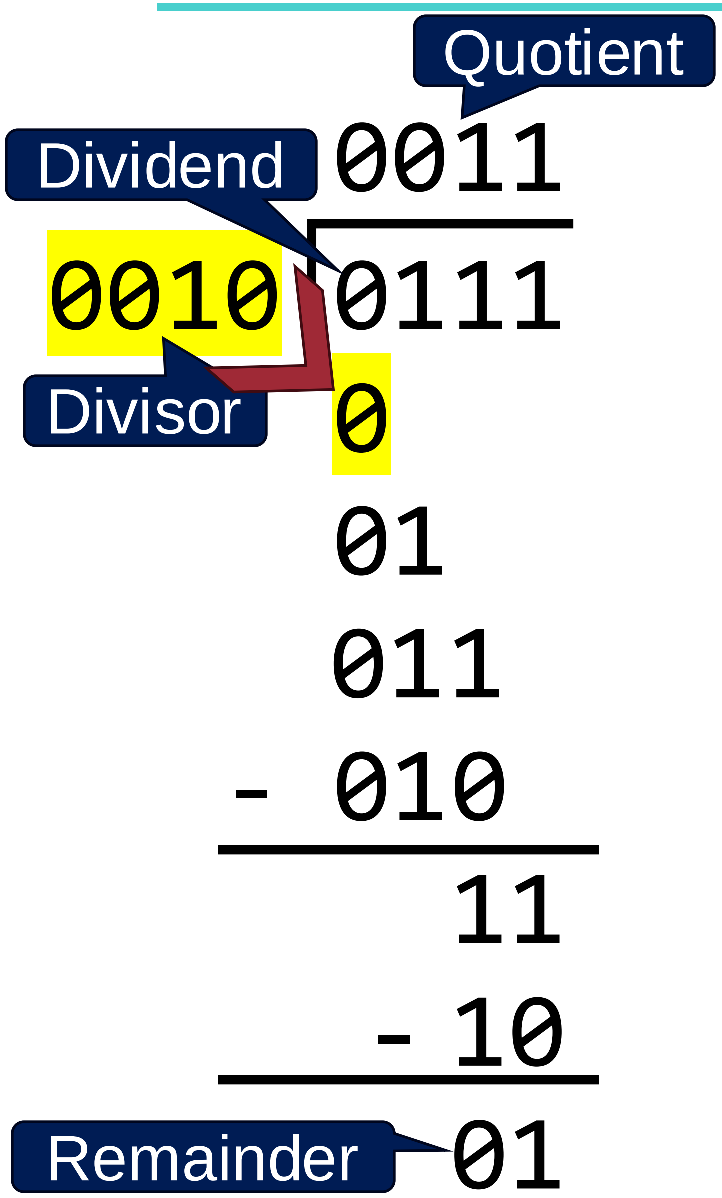
Example: 1st Iteration – 2

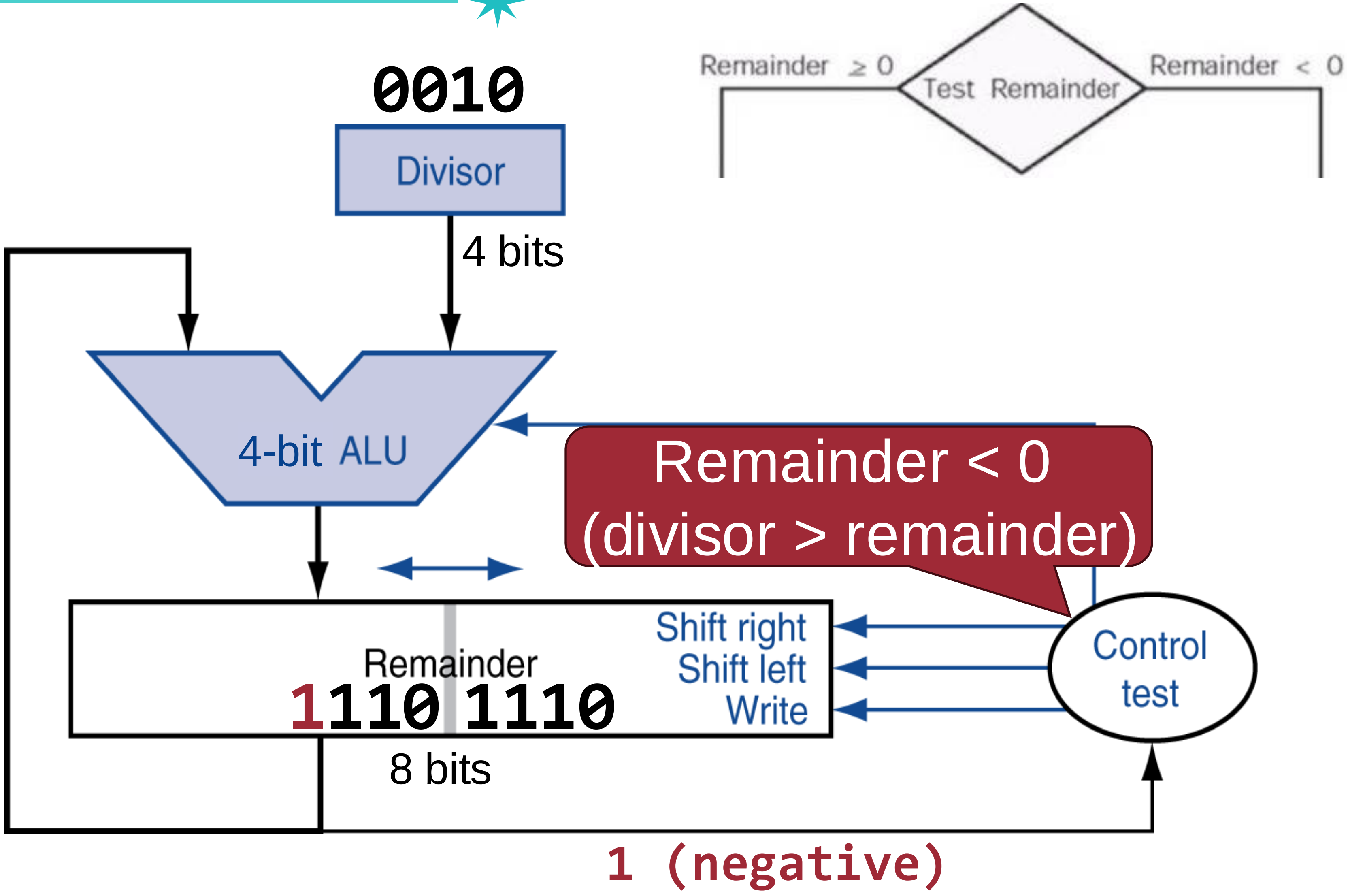
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register



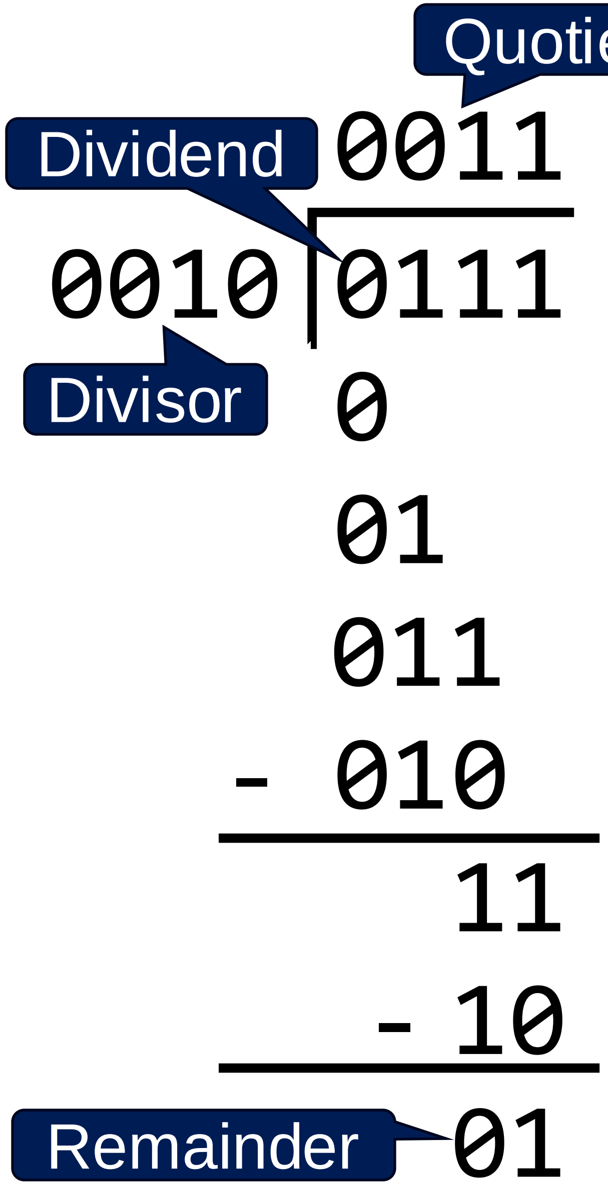


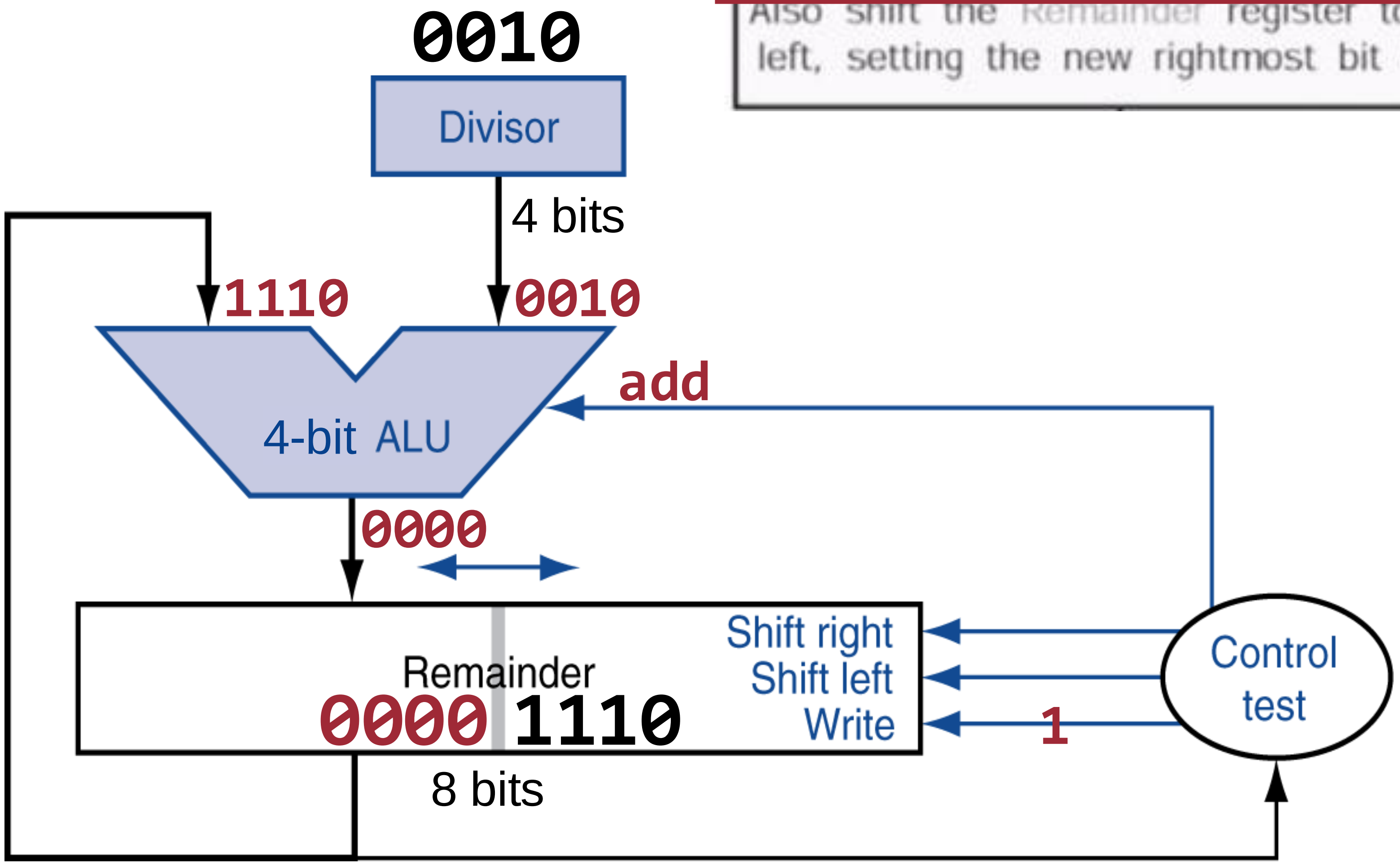
Example: 1st Iteration – Test Remainder





Example: 1st Iteration – 3b





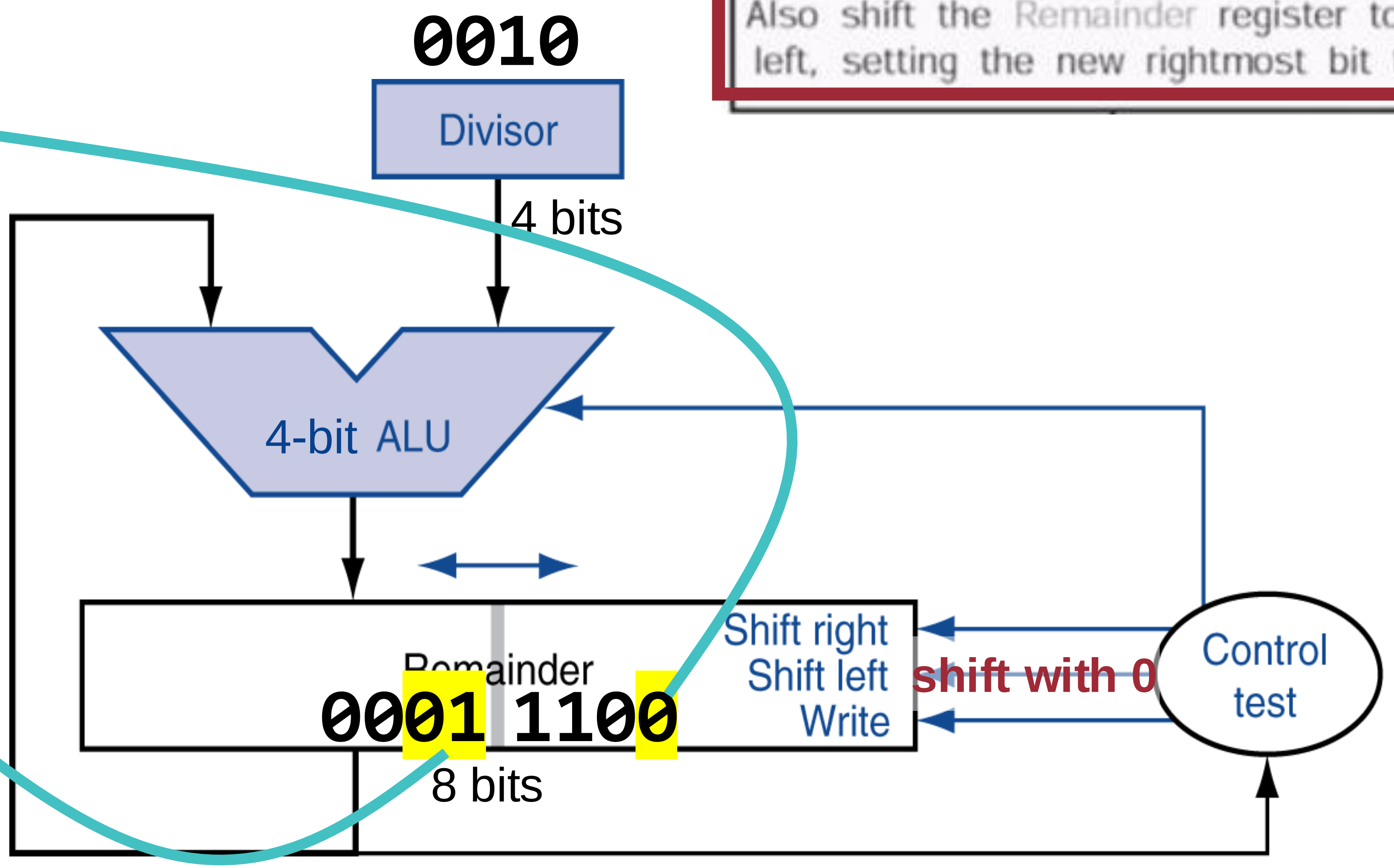
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

Example: 1st Iteration – 3b

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

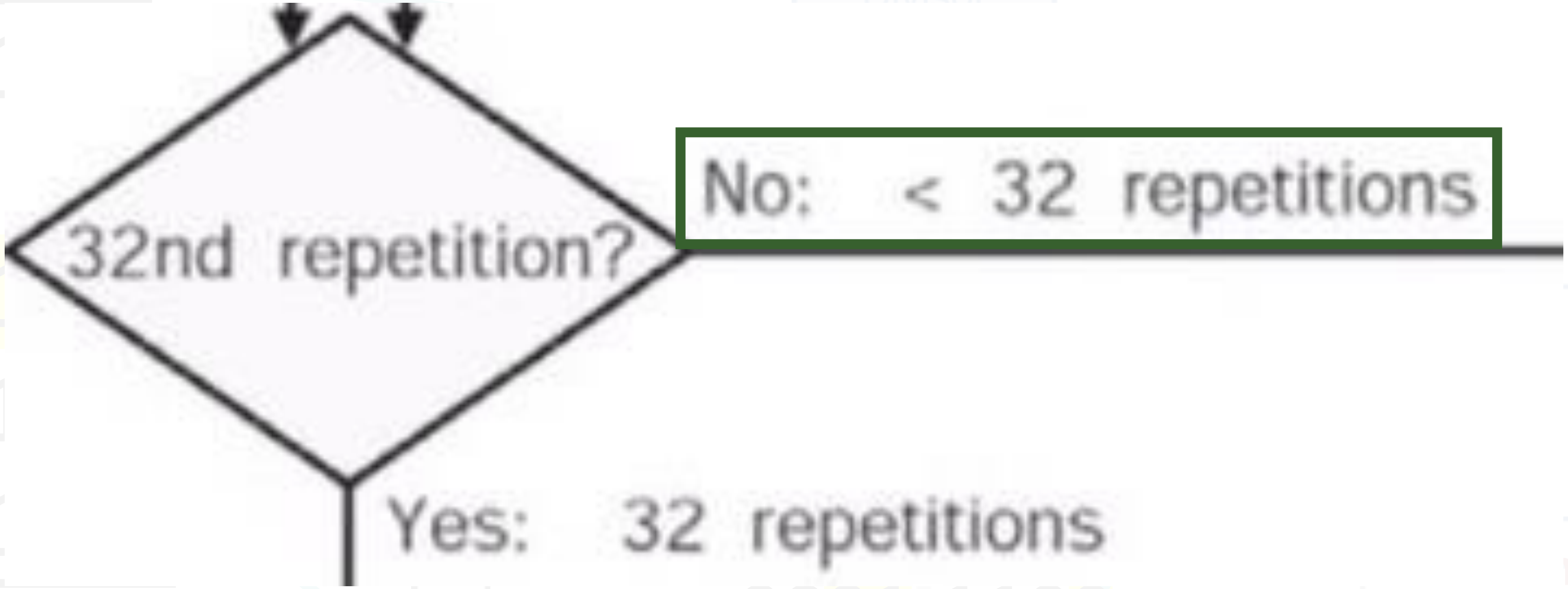
Quotient

```
       0011
Dividend
0010 | 0111
Divisor
       0
       01
       011
     - 010
       ----
         11
       - 10
       ----
Remainder 01
```



Example: 1st Iteration – 32nd Repetition?

In this example, we should check if it is the 4th repetition



Example: 2nd Iteration – 2

> 2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

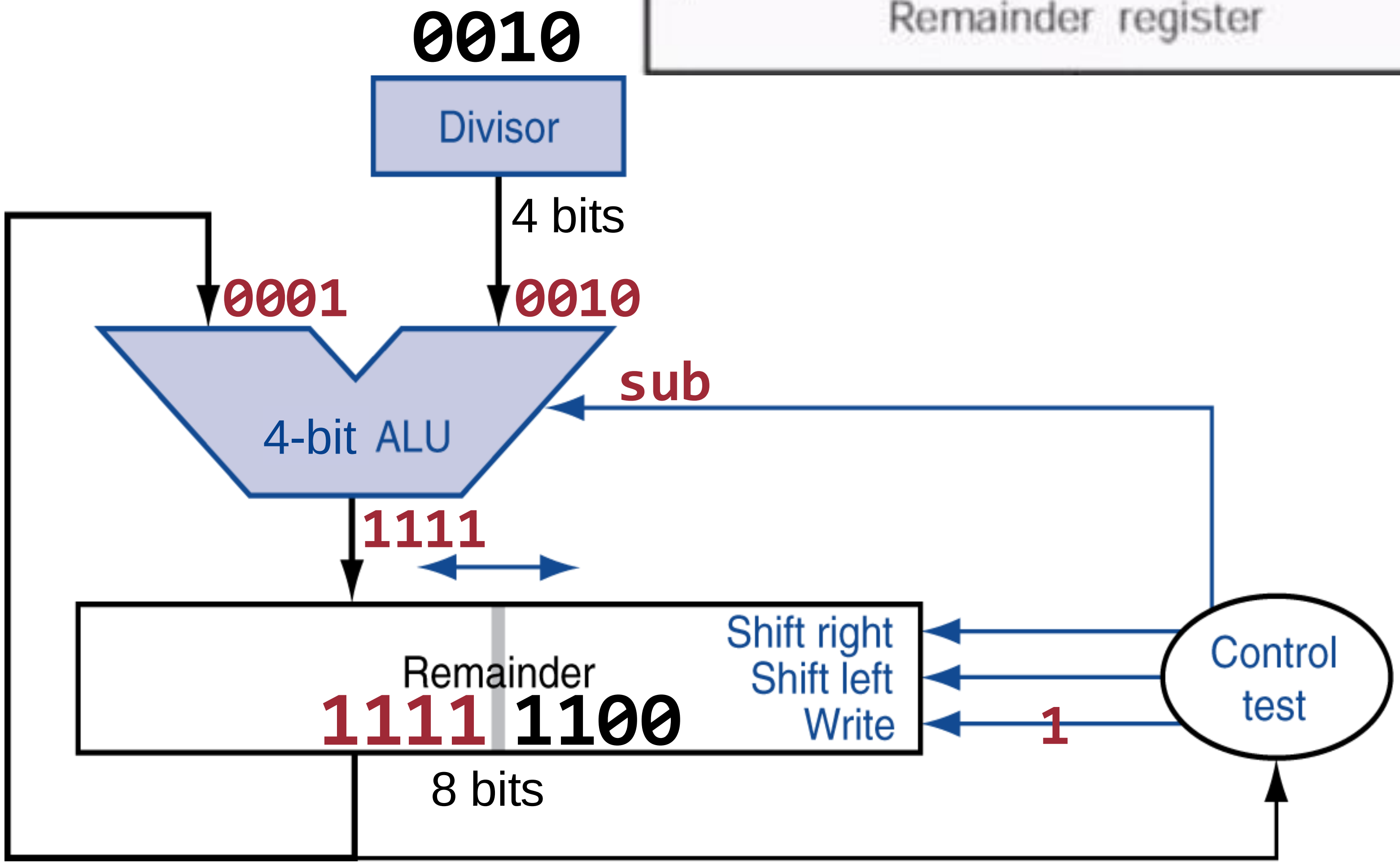
Quotient: 0011

Dividend: 0111

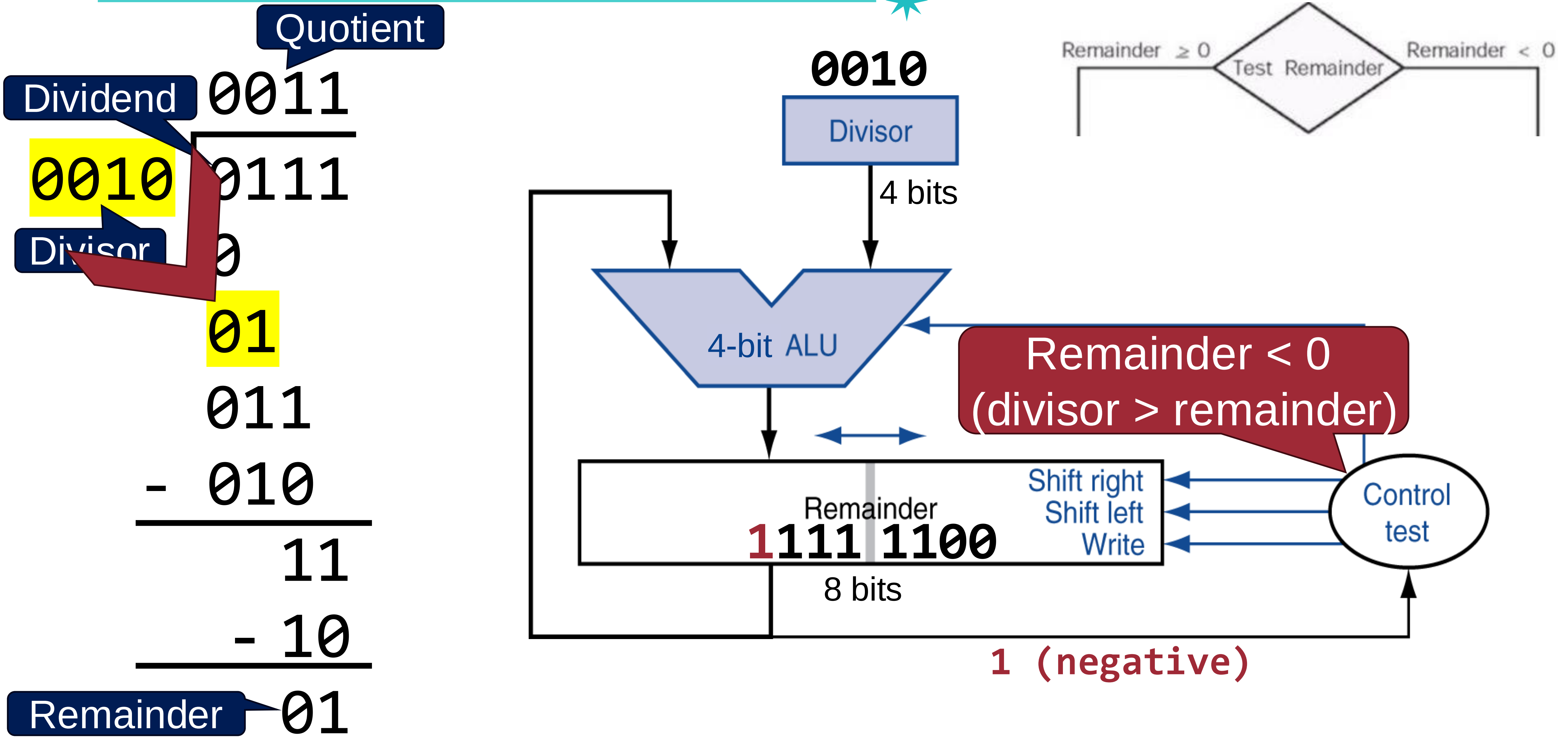
Divisor: 0010

```
         0011
0010 |0111
      0
      01
      011
    - 010
    ------
        11
      - 10
    ------
        01
```

Remainder: 01

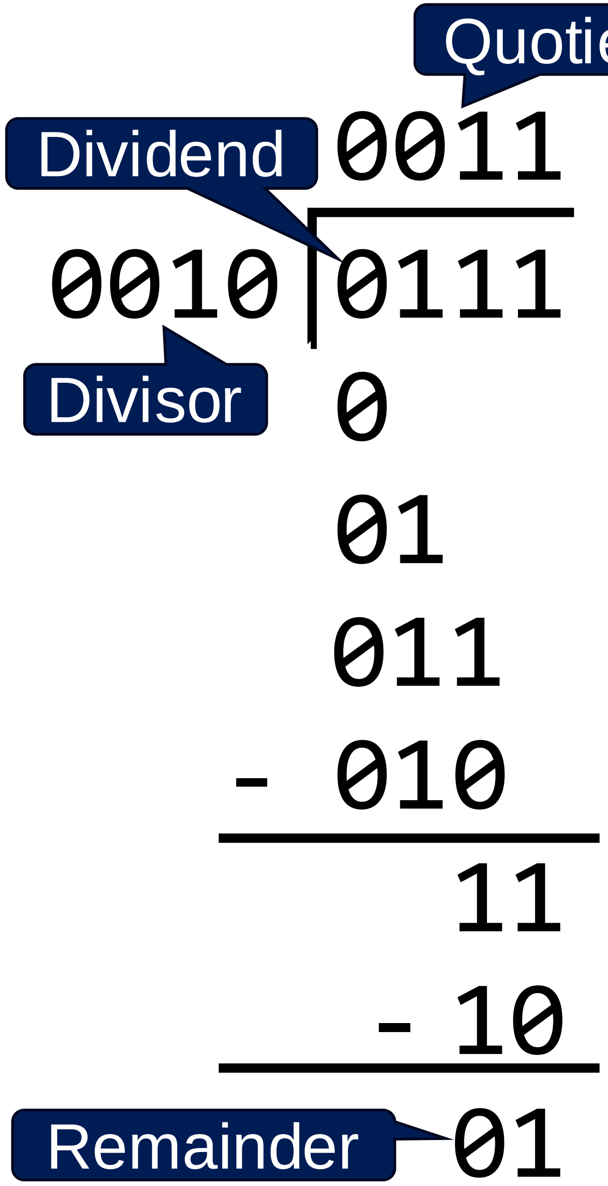


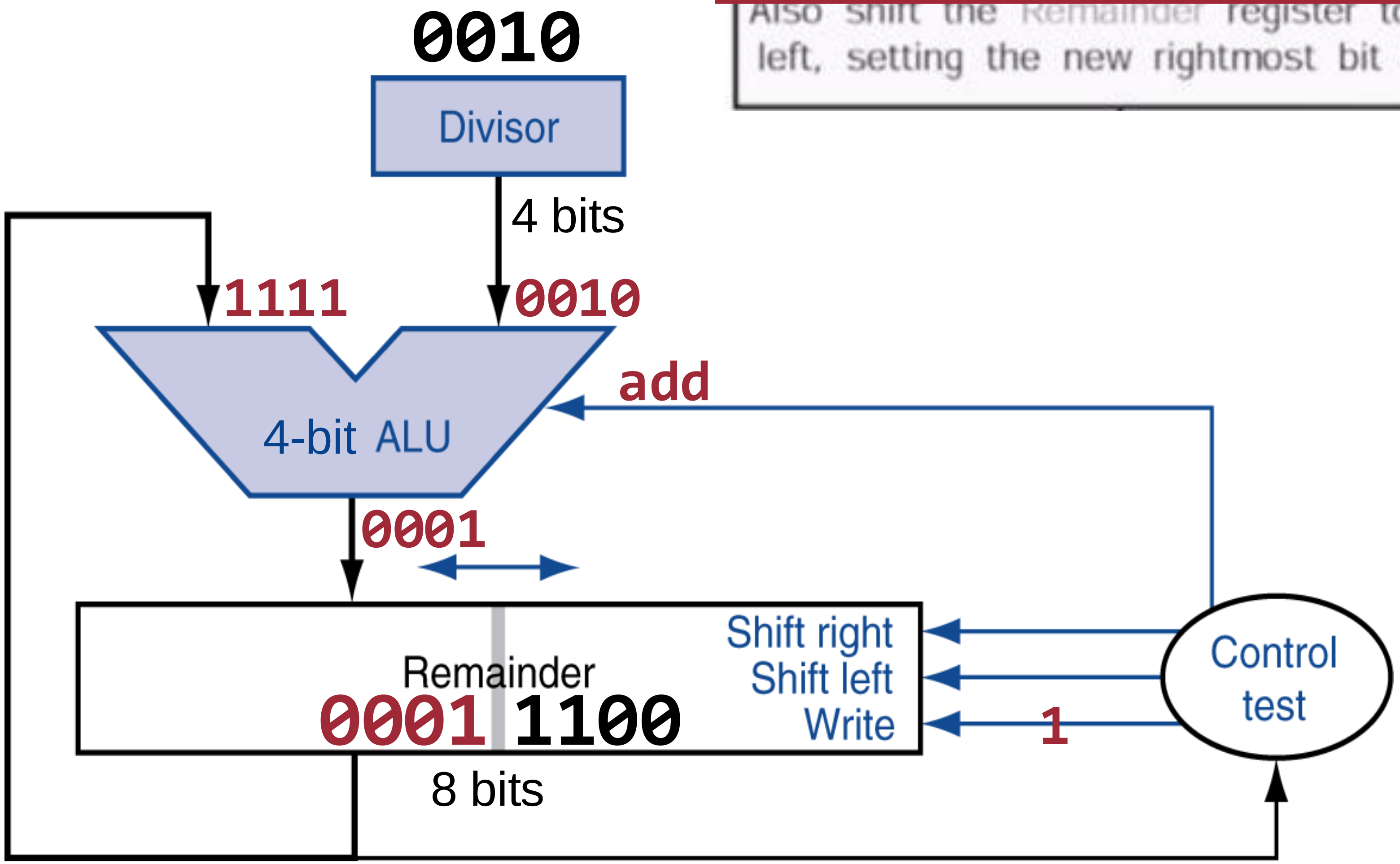
Example: 2nd Iteration – Test Remainder



Example: 2nd Iteration – 3b

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0





Example: 2nd Iteration – 3b

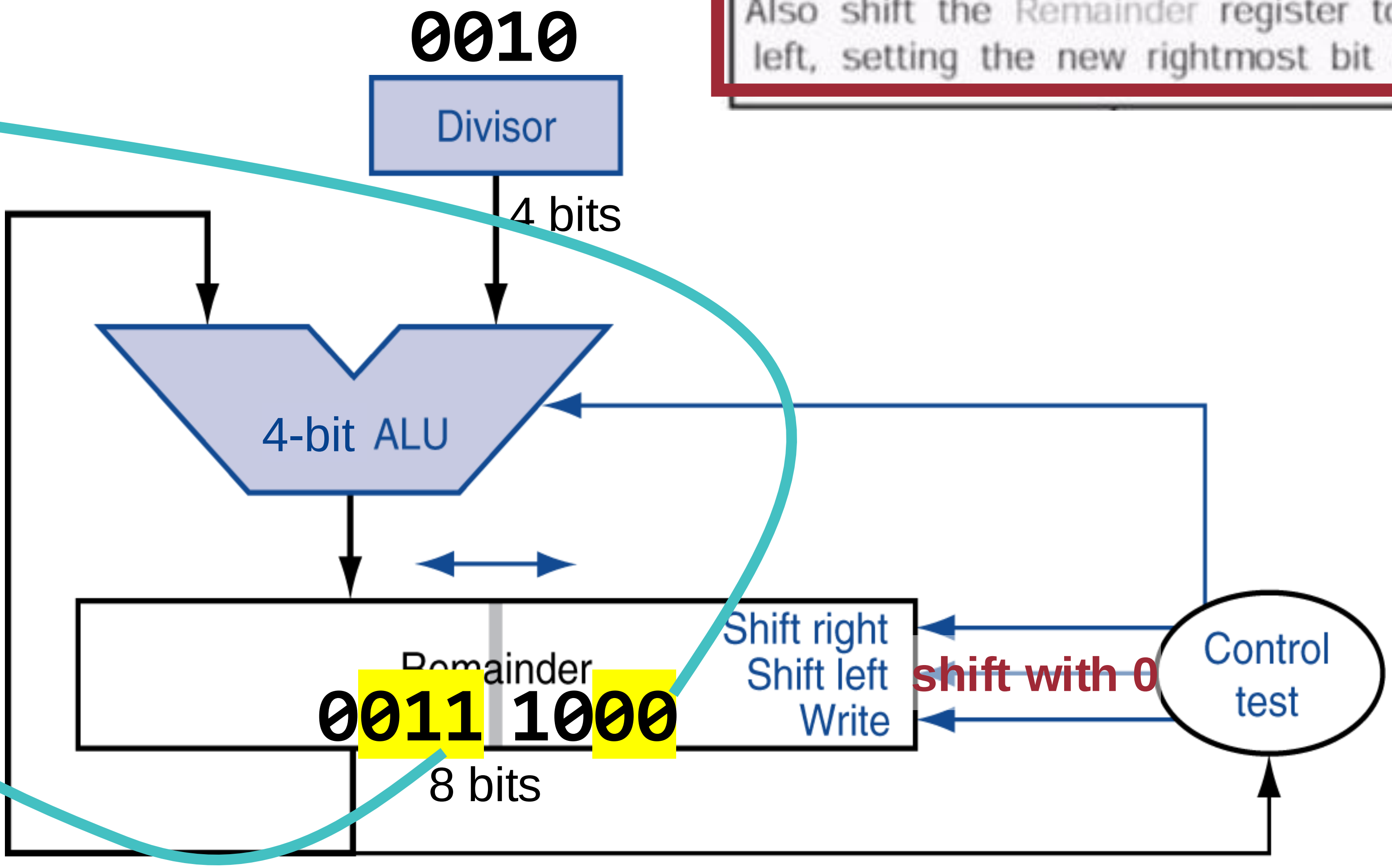
Quotient

Dividend

```
       0011
      _____
0010 | 0111
       0
       01
       011
     - 010
     -----
        11
      - 10
     -----
        01
```

Divisor

Remainder

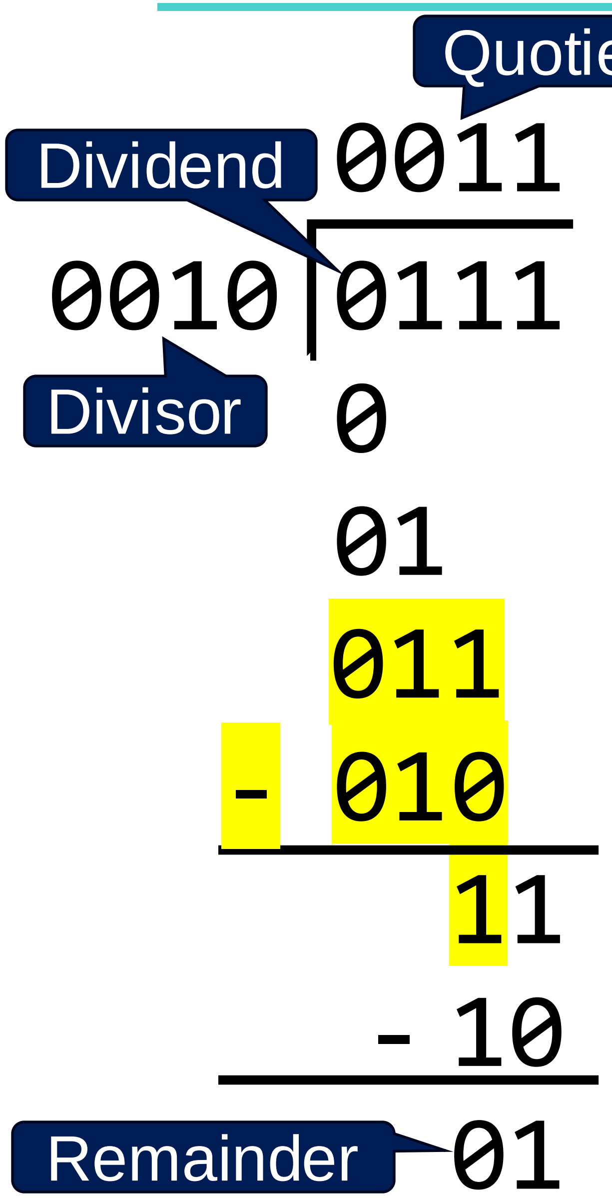


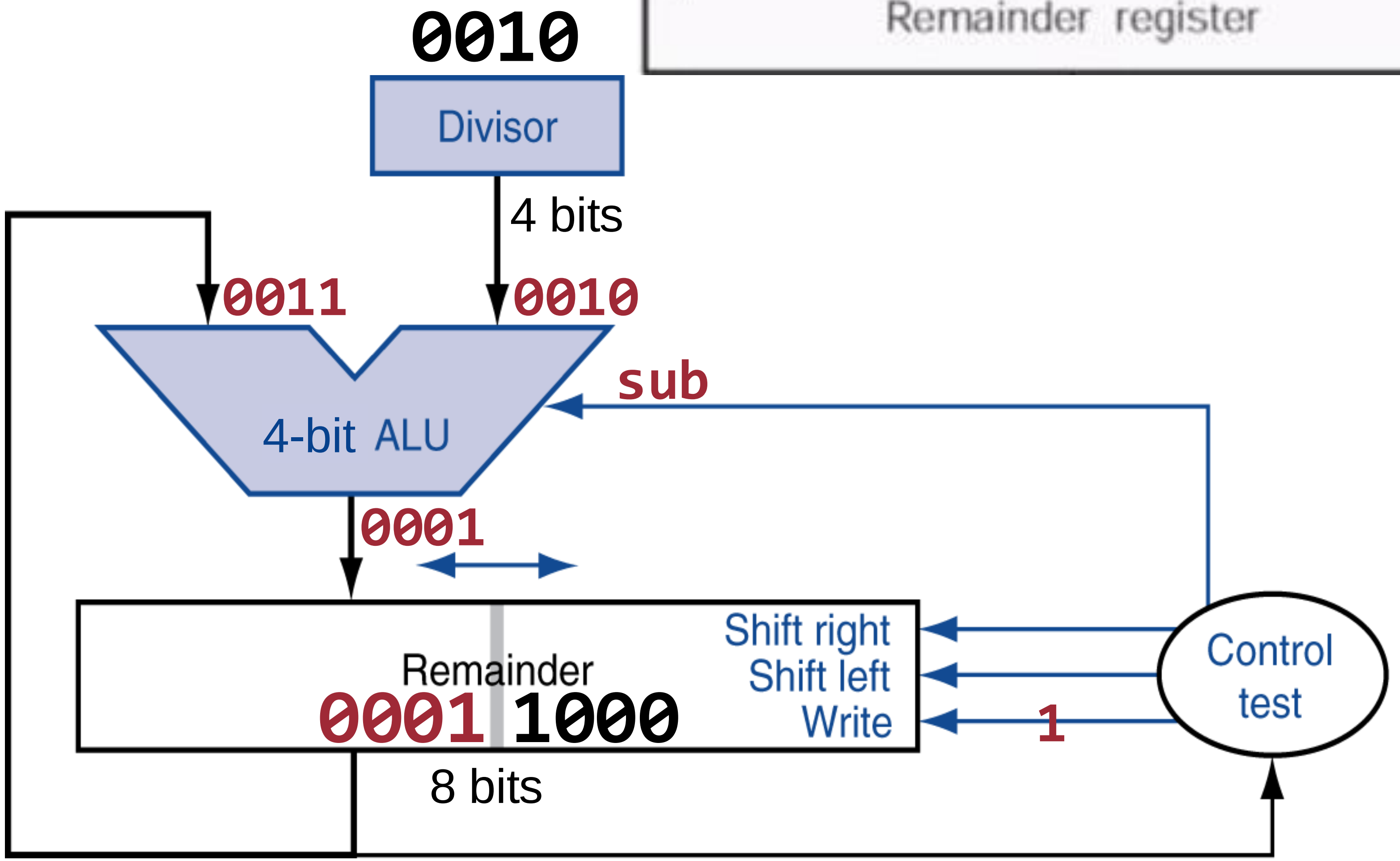
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register.

Also shift the Remainder register to the left, setting the new rightmost bit to 0

Example: 3rd Iteration – 2

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register





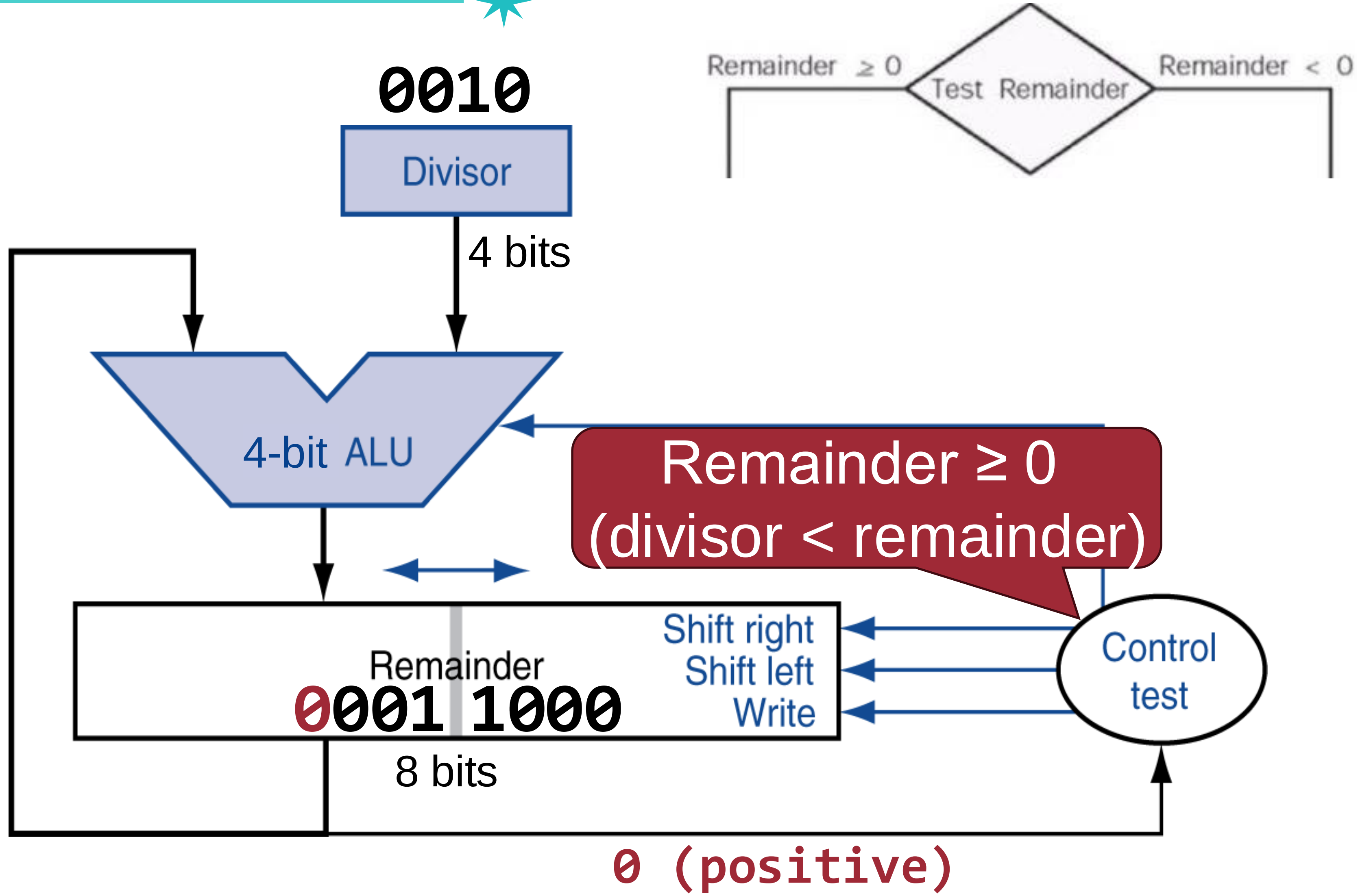
Example: 3rd Iteration – Test Remainder

Quotient

Dividend 0011

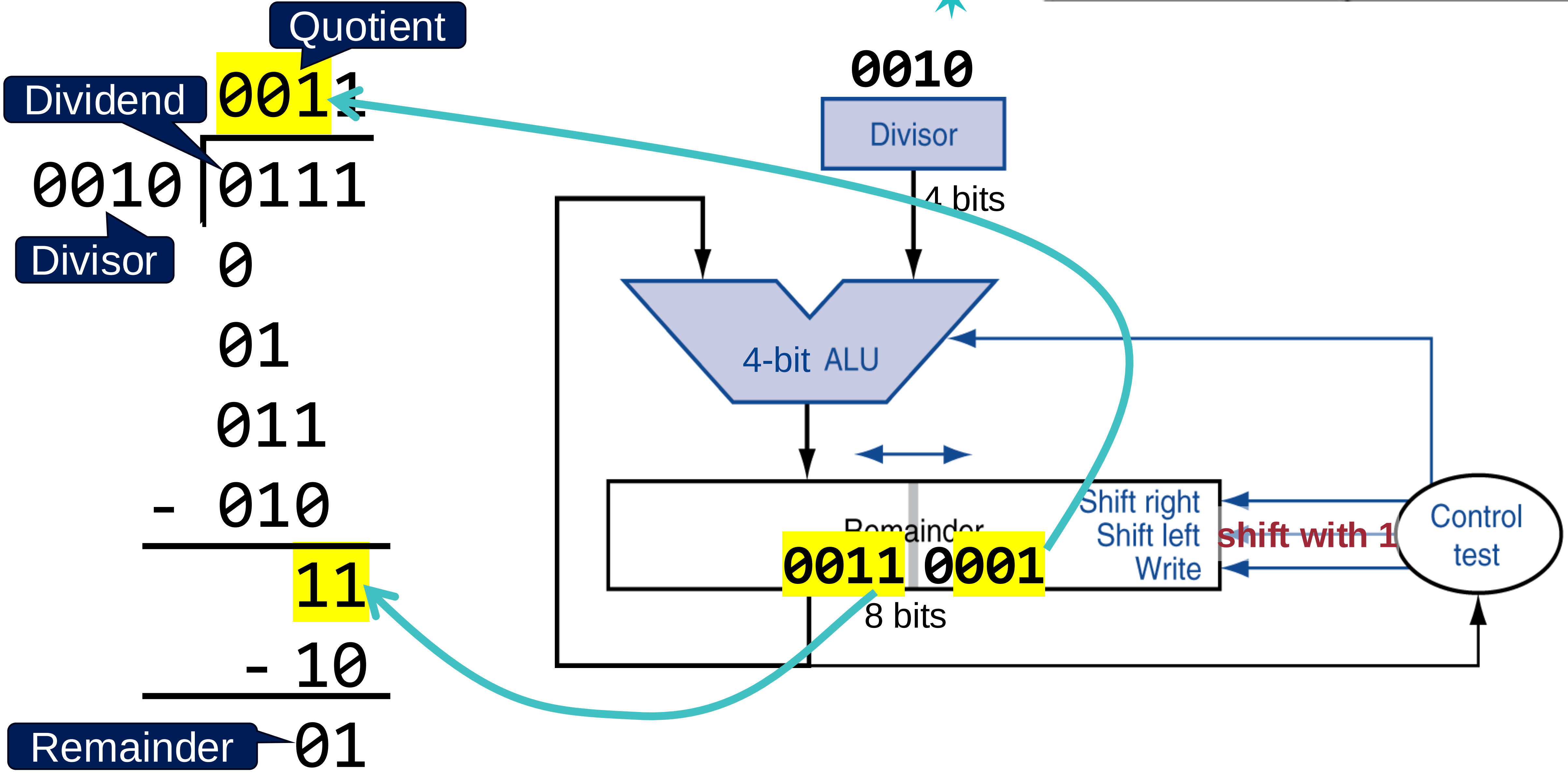
Divisor 0010 | 0111

```
       0011
0010 | 0111
       0
       01
       011
     - 010
        11
      - 10
Remainder 01
```



Example: 3rd Iteration – 3a

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1



Example: 4th Iteration – 2

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

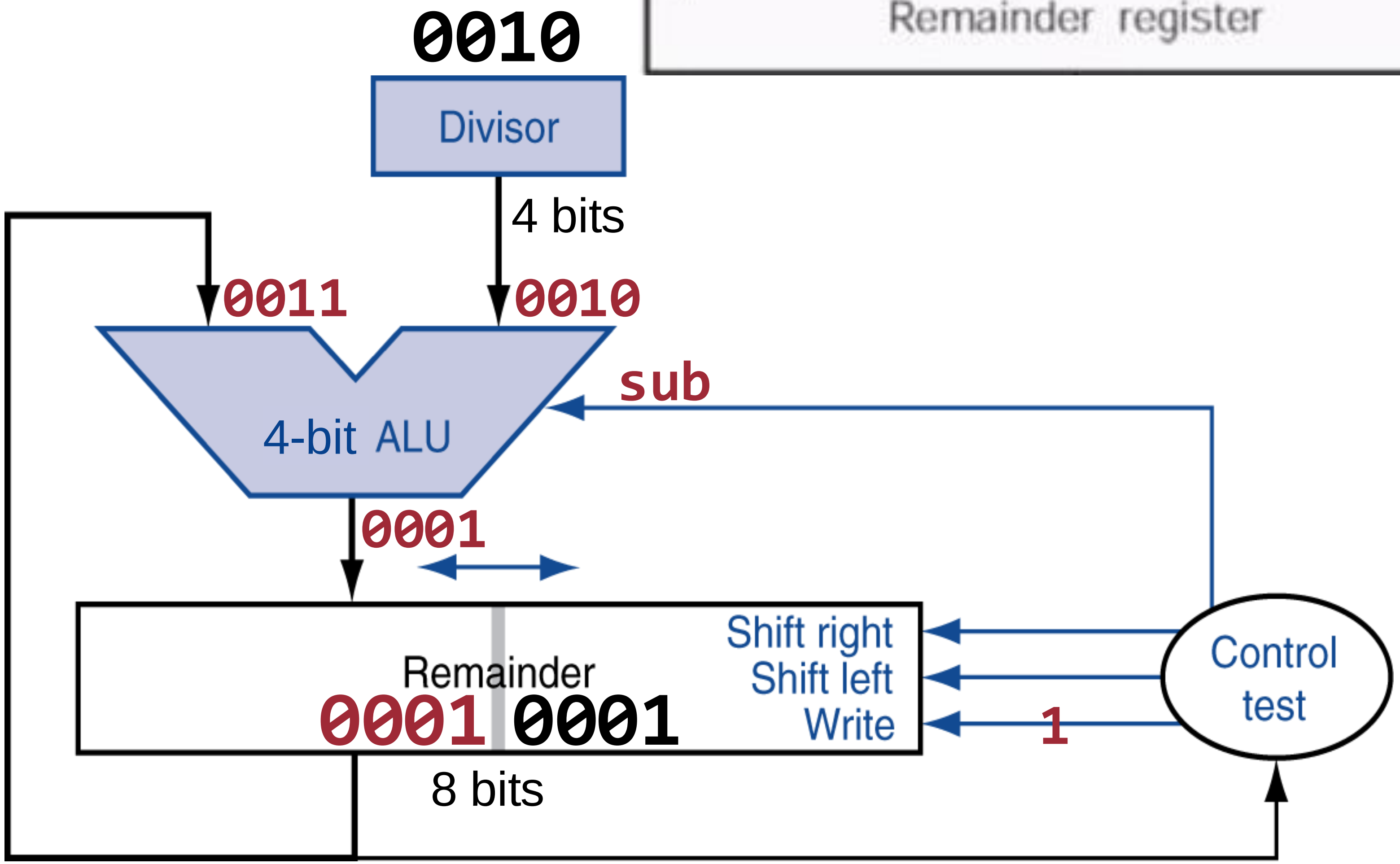
Quotient: 0011

Dividend: 0111

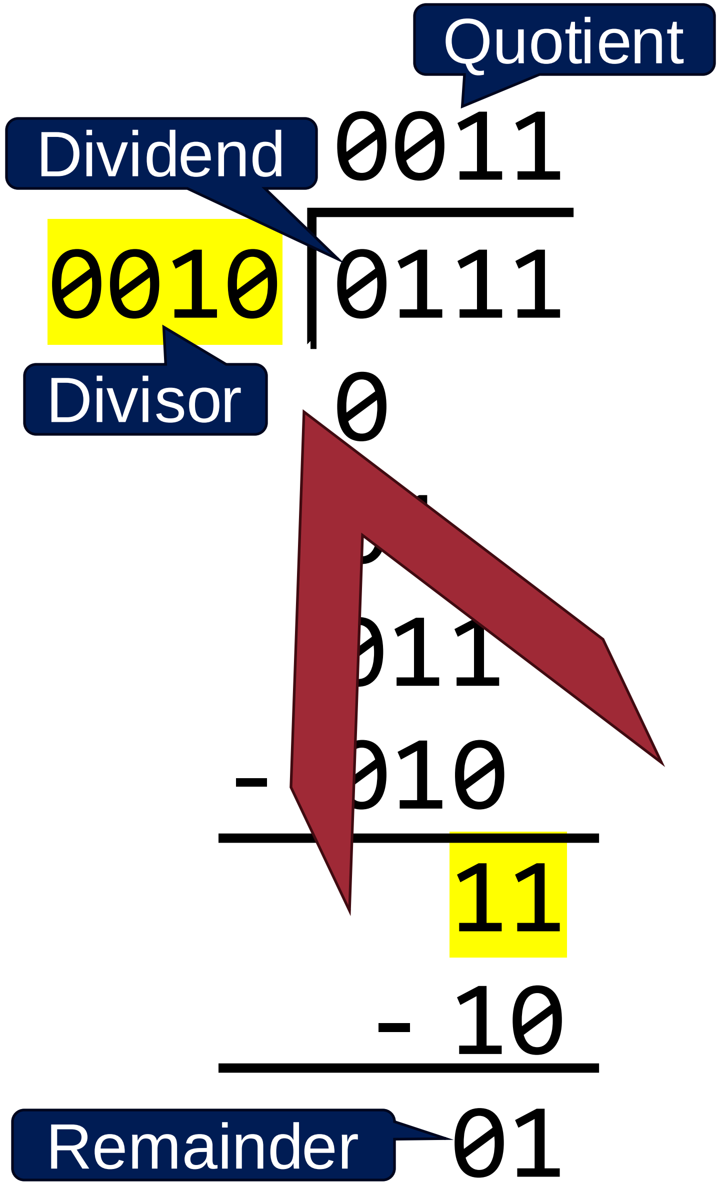
Divisor: 0010

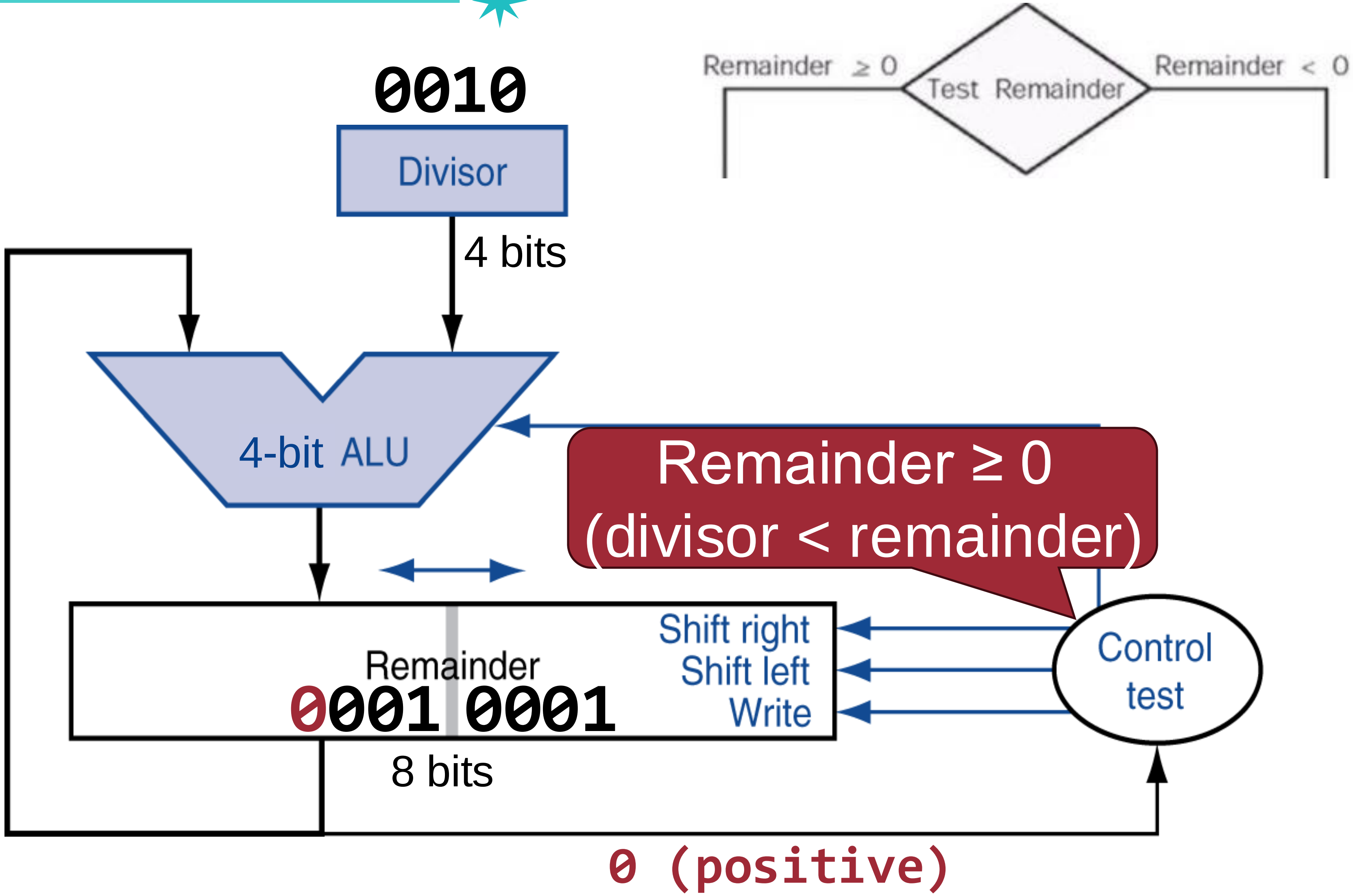
```
        0011
0010 | 0111
       0
       01
       011
     - 010
     -----
         11
       - 10
     -----
         01
```

Remainder: 01



Example: 4th Iteration – Test Remainder





Example: 4th Iteration – 3a

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

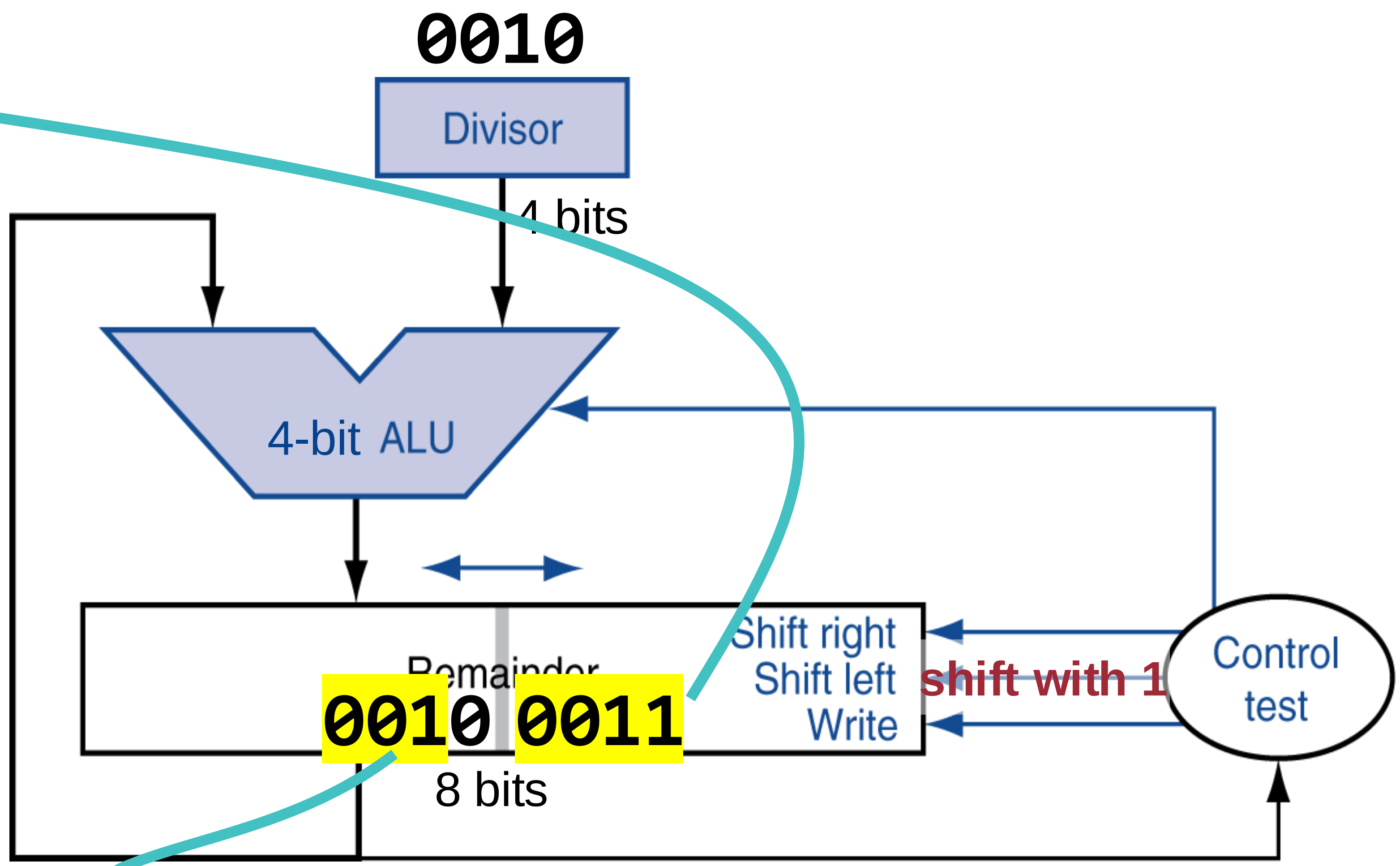
Quotient

Dividend

```
      0011
0010 |0111
      0
      01
      011
    - 010
        11
      - 10
        01
```

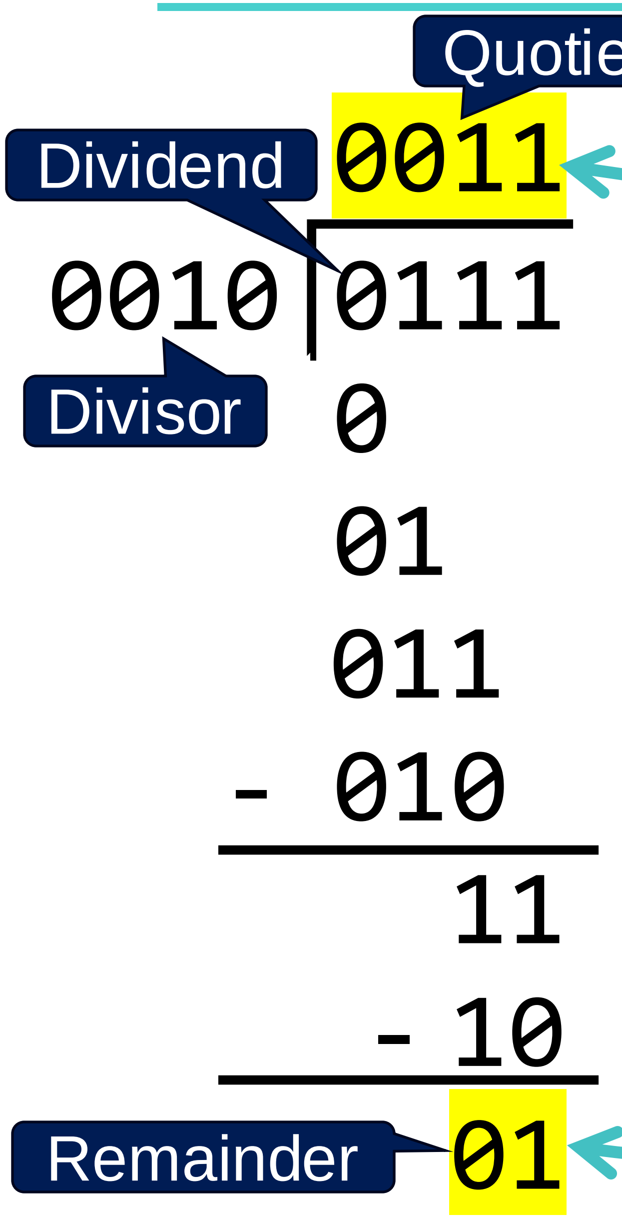
Divisor

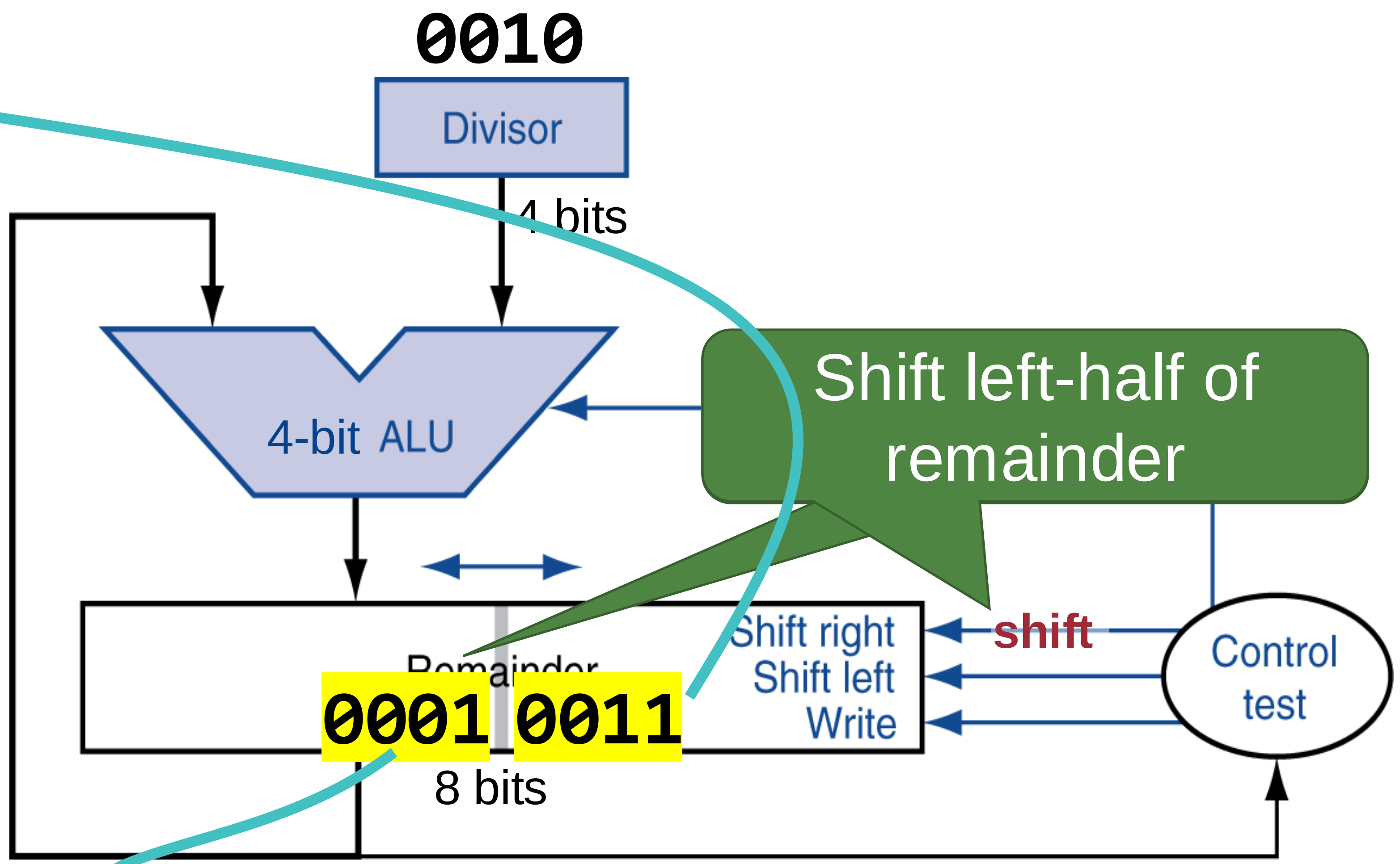
Remainder



Example: Final Step

Done. Shift left half of Remainder right 1 bi





Division of Signed Number

- The ***magnitude*** of quotient/remainder depends on the ***magnitude*** of dividend/divisor
- The ***sign*** of quotient/remainder depends on the ***sign*** of dividend/divisor

Example	Quotient	Remainder
+7 / +2	+3	+1
-7 / +2	-3	-1
+7 / -2	-3	+1
-7 / -2	+3	-1

Division of Signed Number

- The ***magnitude*** of quotient/remainder depends on the ***magnitude*** of dividend/divisor

After calculating with positive values, convert based on the sign at the end

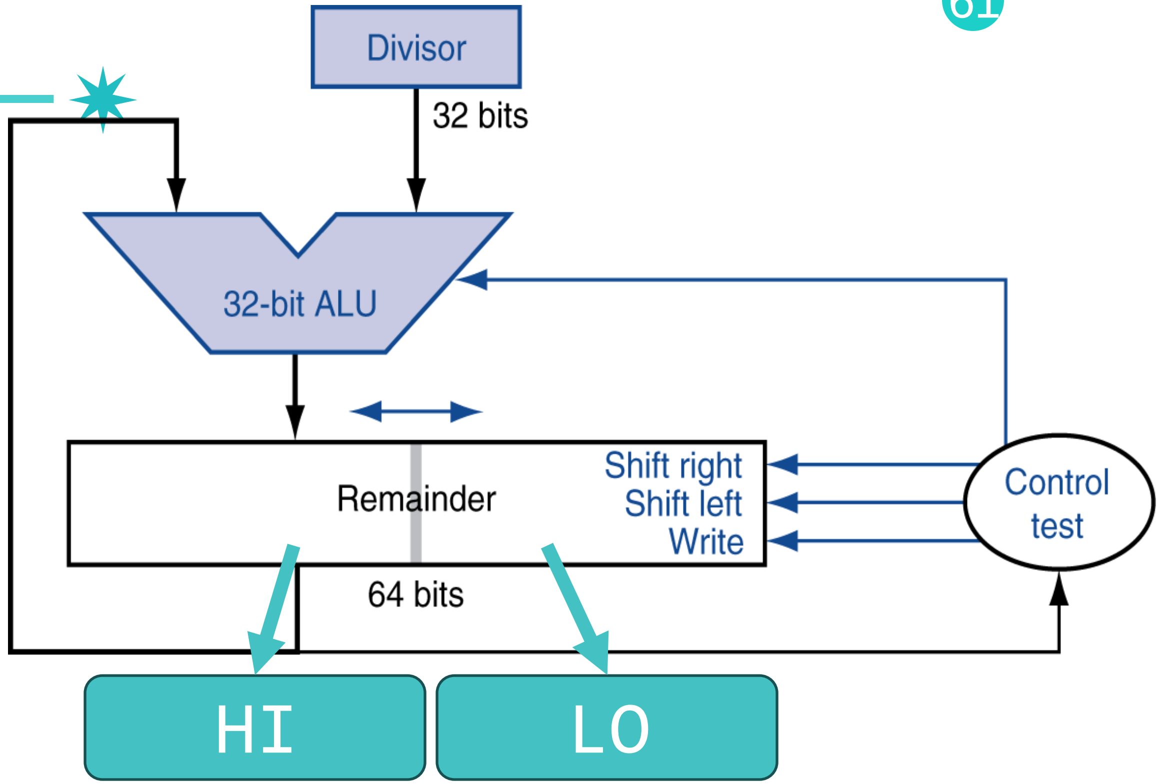
+7 / +2	+3	+1
-7 / +2	-3	-1
+7 / -2	-3	+1
-7 / -2	+3	-1

Faster Division?

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder

MIPS Division

- **Use `HI/LO` registers for result**
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- **Instructions**
  - `div rs, rt / divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result



Floating-point Number Arithmetic

Background: Number System

- Decimal number system

$$\begin{tikzcd} \cdots & 10^{2} & 10^{1} \arrow[l, bend left, "\mathrm{x}10"] & 10^{0} \arrow[l, bend left, "\mathrm{x}10"] \arrow[r, bend right, "/10"'] & 10^{-1} \arrow[r, bend right, "/10"'] & 10^{-2} & \cdots \end{tikzcd}$$

- Binary number system

$$\begin{tikzcd} \cdots & 2^{2} & 2^{1} \arrow[l, bend left, "\mathrm{x}2"] & 2^{0} \arrow[l, bend left, "\mathrm{x}2"] \arrow[r, bend right, "/2"'] & 2^{-1} \arrow[r, bend right, "/2"'] & 2^{-2} & \cdots \end{tikzcd}$$

Floating-point Number: Motivation

We need a way to represent ...

- *Infinite decimal (e.g., 3.1415926535…)*
- *Very small numbers*
- *Very large numbers*

from computer!

Solution: Floating-point Number Representation

Floating-point Number: Motivation

We need a way to represent ...

- *Infinite decimal (e.g., 3.1415926535…)* → *Approximate value* → 3.1415
- *Very small numbers* → *Floating decimal point* → 0.001×10^{-20}
- *Very large numbers* → *Floating decimal point* → 3.15576×10^{19}

Can be represented with a limited number of bits!

Solution: Floating-point Number Representation

Floating-point Number: Notations

- **Scientific notation**: renders numbers with a single digit to the *left* of the point
  - Example: 7.15576×10^4, 0.314×10^1

Normalized

- **Normalized scientific notation**: scientific notation that has *no leading 0s*
  - Example: 7.15576×10^4, 3.14×10^0

Floating-point Number: Notations

- **Scientific notation**: renders numbers with a <u>single digit</u> to the *left* of the point

$$1 \leq \textit{Significand} < 10$$

- **Normalized scientific notation**: <u>scientific notation</u> that has *no leading 0s*
  - Example: 7.15576×10^{4}, 3.14×10^{0}

Significand

Base

Exponent

$$(\text{sign}) \times \text{significand} \times \text{base}^{\text{exponent}}$$

Floating-point Number: Binary

- **Scientific notation**: renders numbers with a single digit to the *left* of the point

$$1 \le \textit{Significand} < 2$$

- **Normalized scientific notation**: scientific notation that has *no leading 0s* ***(= Always 1 to the left of the point)***
  - Example: **1.1×2^{-1}** $(0.75_{ten} \rightarrow 0.5+0.25 \rightarrow 0.11_{two} \rightarrow 1.1 \times 2^{-1})$

2

$$\text{(sign)} \times \text{significand} \times \text{base}^{\text{exponent}}$$

Floating-point Number: Binary

- **Scientific notation**: renders numbers with a single digit to the *left* of the point

1 ≤ Significand < 2

How are floating-point numbers represented in computers?

$\rightarrow 0.11_{two} \rightarrow 1.1 \times 2^{-1}$)

2

(sign) x significand x base^{exponent}

IEEE 754 Floating-point Standard

- Developed in response to *divergence of representations*

Divergence of representations

$$0.11_{two} = 1.1 \times 2^{-1}{}_{two} = 11 \times 2^{-2}{}_{two}$$

↓

Normalized representation

$$1.1 \times 2^{-1}{}_{two}$$

↓

IEEE 756 representation

S	Exponent	Fraction

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Sign
- Single: 1 bit
- Double: 1 bit

Exponent
- Single: 8 bits
- Double: 11 bits

Fractional part of significand
- Single: 23 bits
- Double: 52 bits

S	Exponent	Fraction

$$+1.1 \times 2^{-1}{}_{two}$$

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Fractional part of significand

- Single: 23 bits
- Double: 52 bits

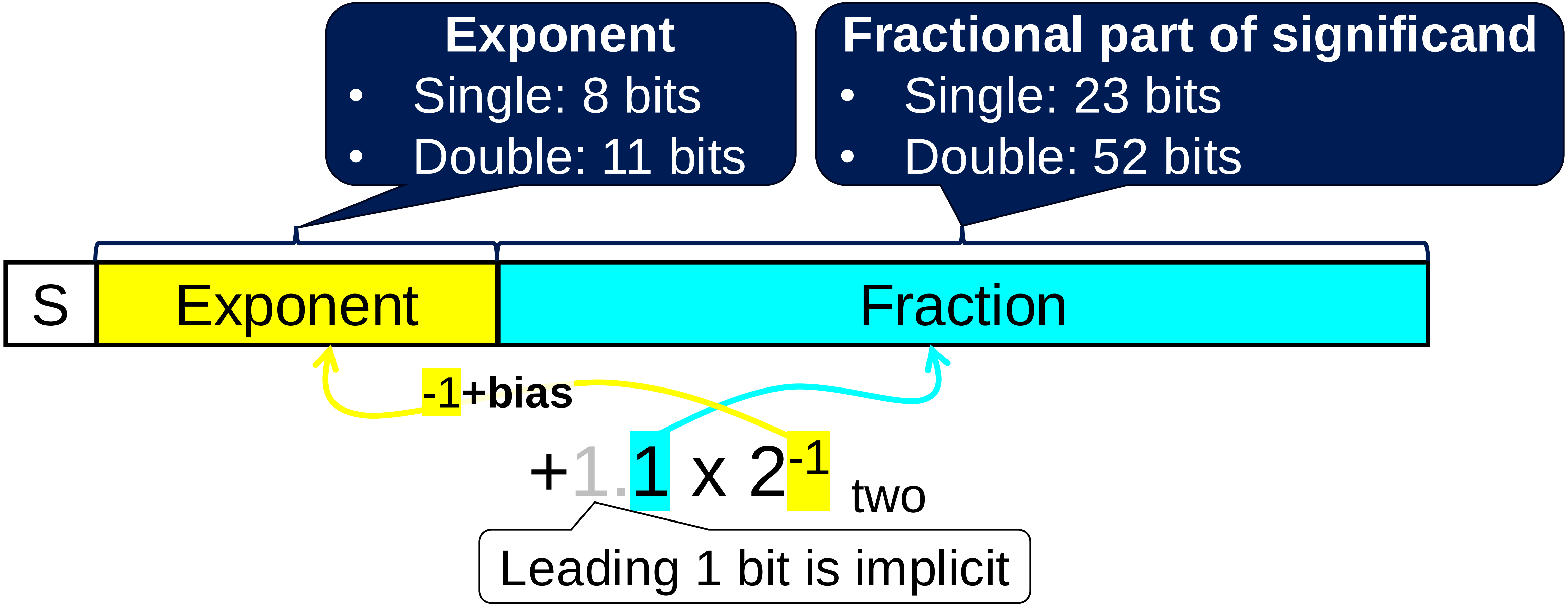
S	Exponent	Fraction

$$+1.1 \times 2^{-1}{}_{\text{two}}$$

Leading 1 bit is implicit

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C



Exponent: Why Biased?

- To make sorting easier
- **Bias**: *127 for single precision* and *1023 for double precision*

S	Exponent	Fraction

-1+bias

$$+1.1 \times 2^{-1}{}_{two}$$

Exponent: Why Biased?

- To make sorting easier
- **Bias**: *127 for single precision* and *1023 for double precision*

Exponent (8 bits)	Decimal (Unsigned)	Biased by 127
00000000	0	0-127 = -127
00000001	1	1-127 = -126
00000010	2	2-127 = -125
…	…	…
11111111	255	255-127 = 128

S	Exponent	

-1+bias

$+1.1 \times 2^{-1}{}_{\text{two}}$

Exponent: Why Biased?

- To make sorting easier
- **Bias**: *127 for single precision* and *1023 for double precision*

Exponent (8 bits)	Decimal (Unsigned)	Biased by 127
00000000	0	0-127 = -127
00000001	1	1-127 = -126
00000010	2	2-127 = -125
…	…	…
11111111	255	255-127 = 128

Easy sorting

S	Exponent	

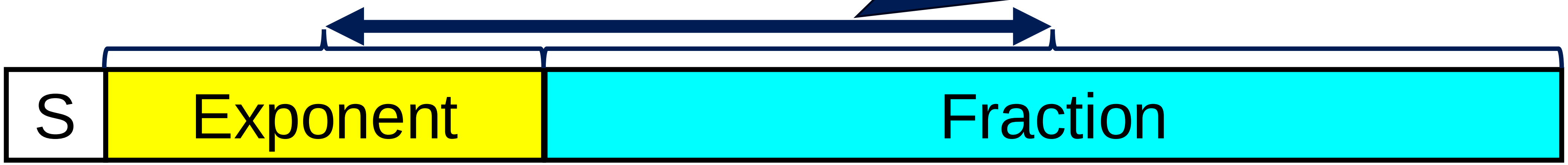
-1+**bias**

$+1.1 \times 2^{-1}$ two

Tradeoff between Precision and Range

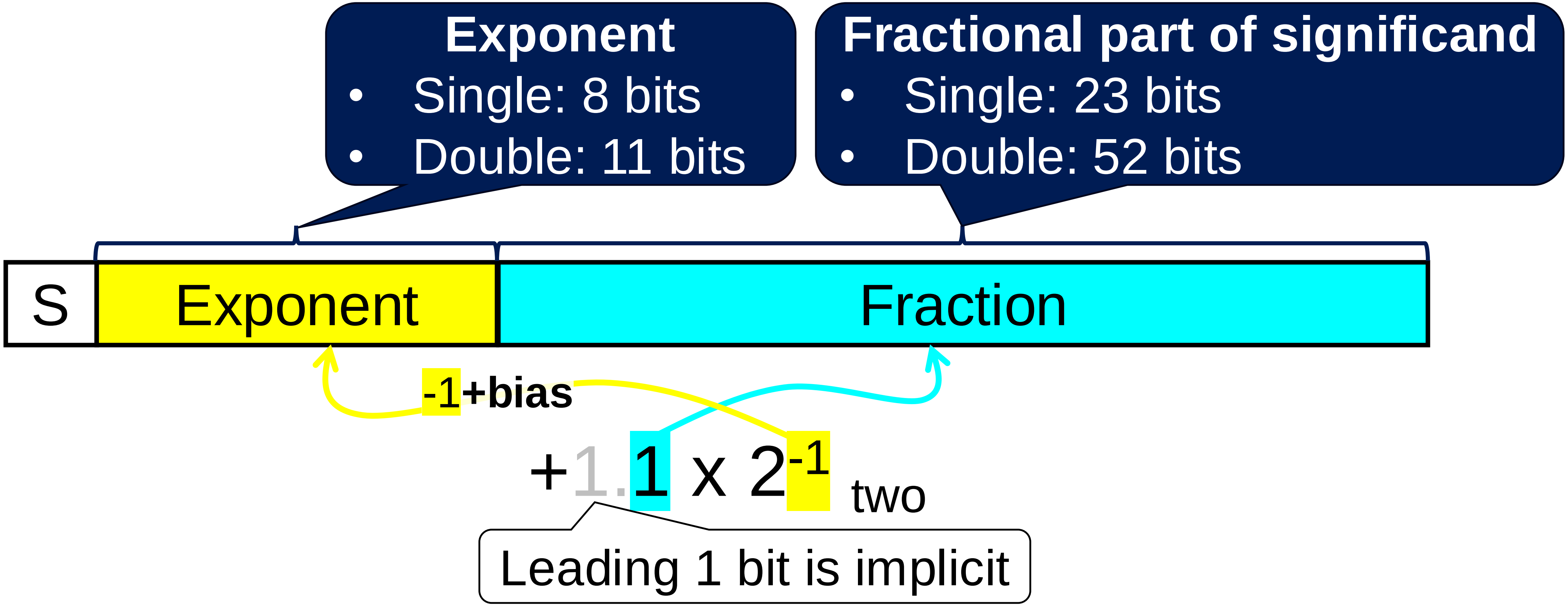
- Increasing *the size of the fraction* enhances ***the precision***
  - Shorter length: 1.1×2^{-1}
  - Longer length: 1.10110×2^{-1}
- Increasing *the size of the exponent* increases ***the range***
  - Shorter length: 1.1×2^{2}
  - Longer length : 1.1×2^{23}

Good design demands good compromise!



IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C



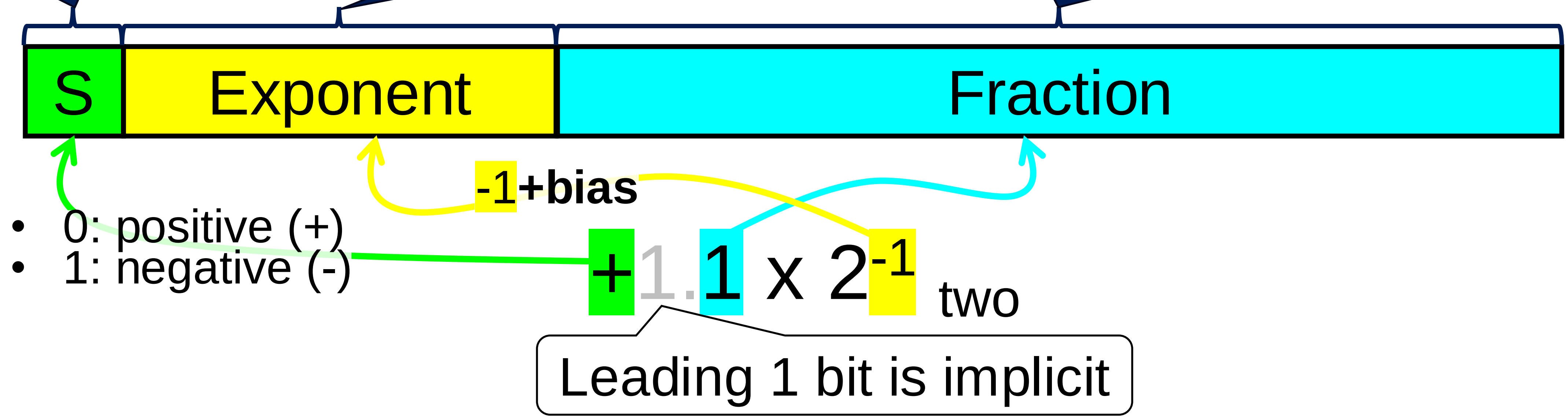
IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Sign
- Single: 1 bit
- Double: 1 bit

Exponent
- Single: 8 bits
- Double: 11 bits

Fractional part of significand
- Single: 23 bits
- Double: 52 bits



IEEE 754 Floating-point Standard: Summary

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

S	Exponent	Fraction

Single-precision Range

- Exponents 00000000 and 11111111 reserved

Used for special cases

- **Smallest value**
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - Exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

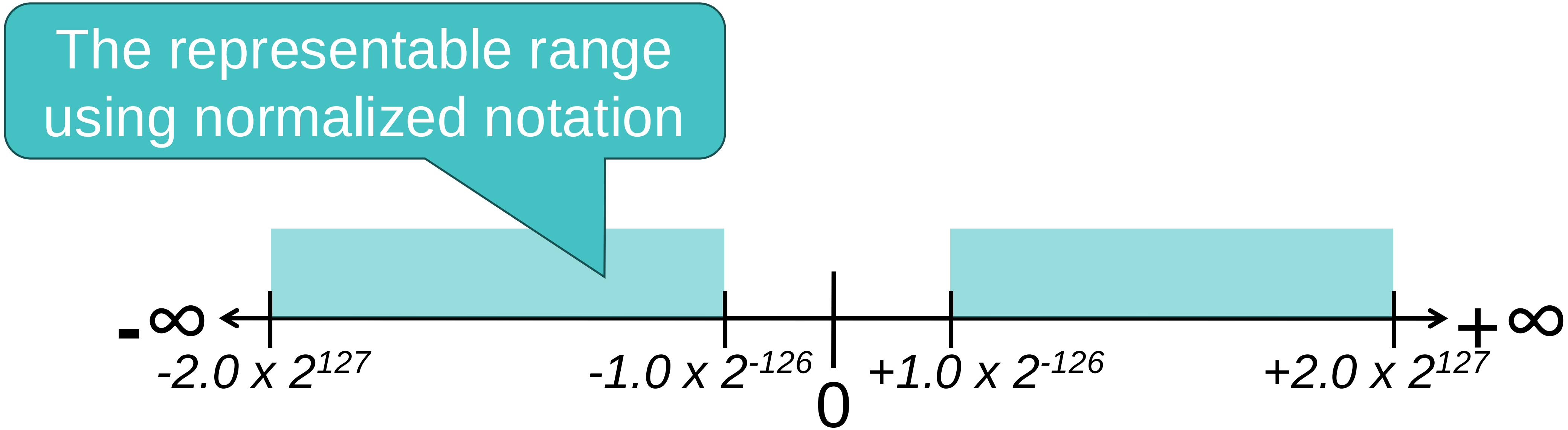
Double-precision Range

- Exponents 0000…00 and 1111…11 reserved

Used for special cases

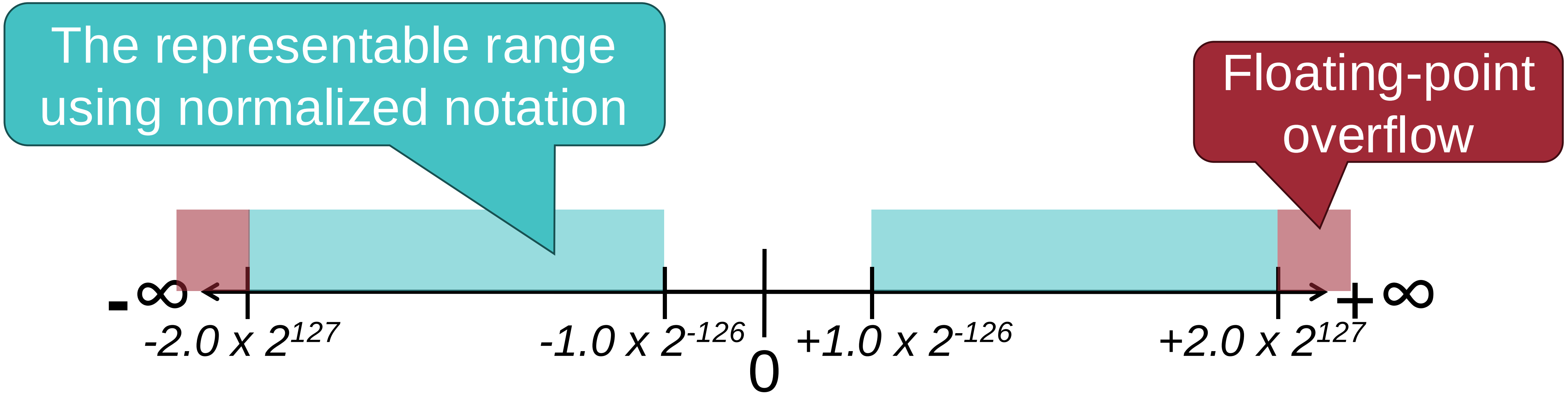
- **Smallest value**
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Overflow and Underflow



Overflow and Underflow

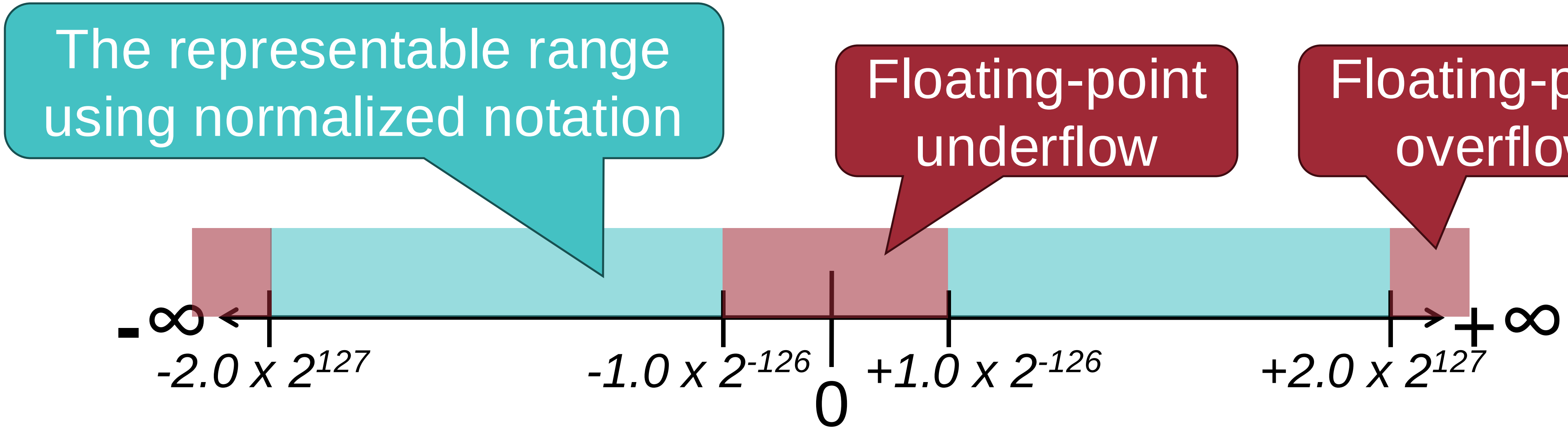
- **Overflow**: occurs when a result has a magnitude ***too big*** to be represented



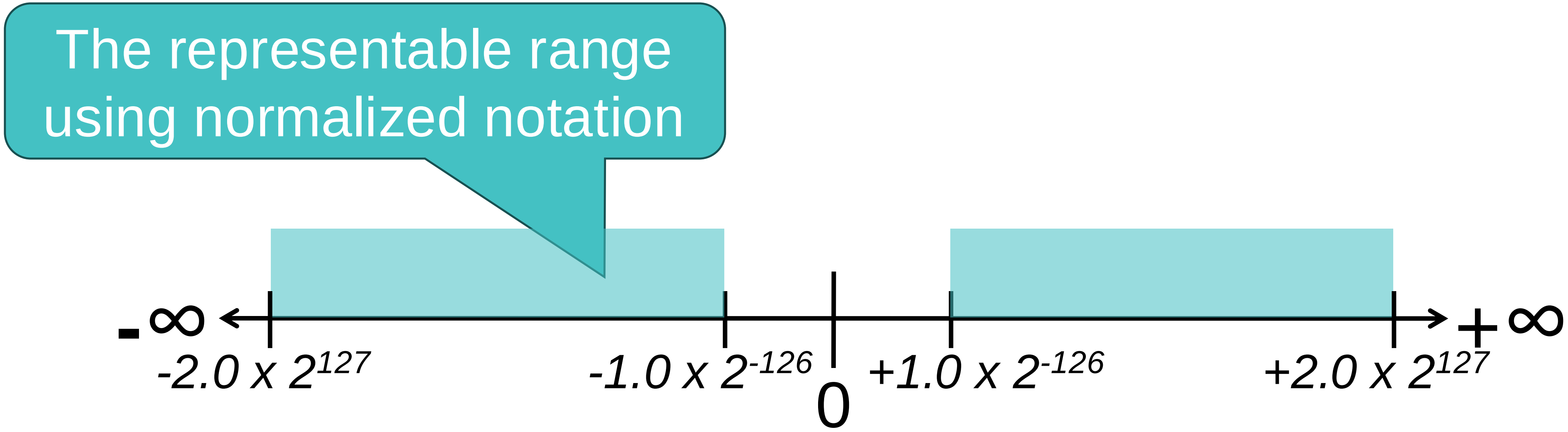
Overflow and Underflow

- **Overflow**: occurs when a result has a magnitude ***too big*** to be represented

- **Underflow**: occurs when a result has a magnitude ***too small*** to be represented

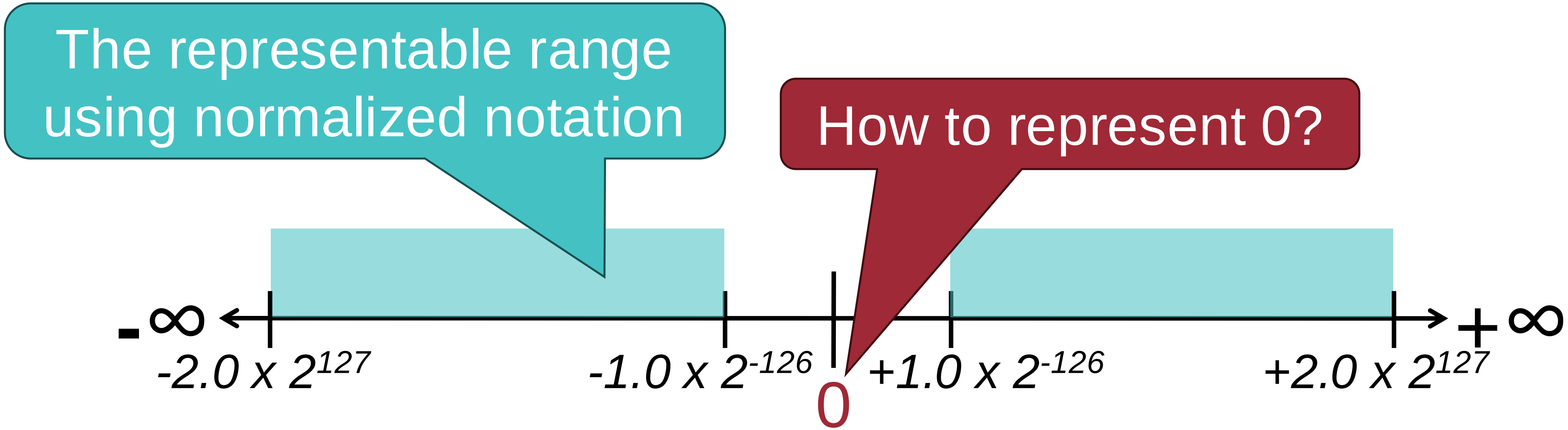


IEEE 754: Special Cases



IEEE 754: Special Cases

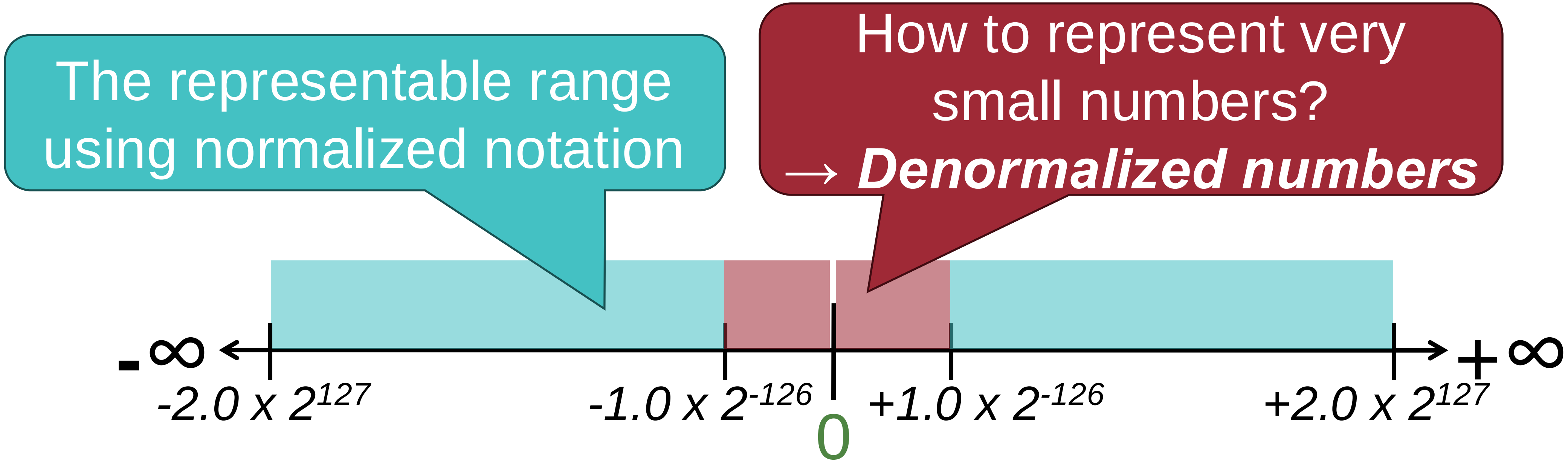
- Exponent = 00…0, Fraction = 00…0
  → Not 1.0 x 2^{-127} but **0**



IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  - → Not 1.0 x 2^{-127} but **0**

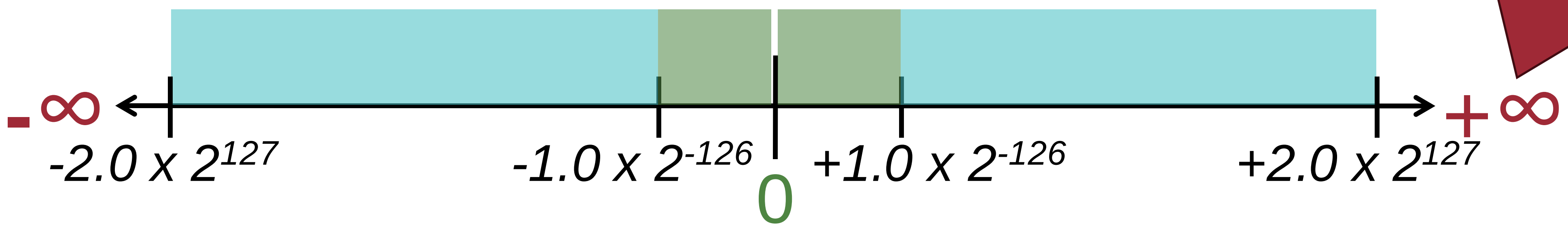
- Exponent = 00…0, Fraction ≠ 00…0
  - → Not (1 + fraction) x 2^{-127} but **(0 + fraction) x 2^{-126}**
  - → Denormalized real numbers (to represent very small numbers)



IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  - → Not 1.0×2^{-127} but **0**
- Exponent = 00…0, Fraction ≠ 00…0
  - → Not $(1 + \text{fraction}) \times 2^{-127}$ but **$(0 + \text{fraction}) \times 2^{-126}$**
  - → Denormalized real numbers (to represent very small numbers)
- Exponent = 11…1, Fraction = 00…0
  - → **±infinity**

How to represent infinity?



IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  - → Not 1.0×2^{-127} but **0**
- Exponent = 00…0, Fraction ≠ 00…0
  - → Not $(1 + \text{fraction}) \times 2^{-127}$ but $\mathbf{(0 + fraction) \times 2^{-126}}$
  - → Denormalized real numbers (to represent very small numbers)
- Exponent = 11…1, Fraction = 00…0
  - → **±infinity**
- Exponent = 11…1, Fraction ≠ 00…0
  - → **Not-a-Number (*NaN*)**
  - → Indicates illegal or undefined result

How to represent the result of invalid operations (e.g., 0/0)?

IEEE 754 Encoding of Floating-point Numbers

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)

Floating-point Addition: Decimal

- Consider a 4-digit decimal example

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

1. Align decimal points

- Shift number with *smaller exponent*
- $9.999 \times 10^{1} + 0.016 \times 10^{1}$

Floating-point Addition: Decimal

- Consider a 4-digit decimal example

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

1. Align decimal points

- Shift number with *smaller exponent*
- $9.999 \times 10^{1} + 0.016 \times 10^{1}$

2. Add significands

- 10.015×10^{1}

Floating-point Addition: Decimal

- Consider a 4-digit decimal example

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

1. Align decimal points

- Shift number with *smaller exponent*
- $9.999 \times 10^1 + 0.016 \times 10^1$

2. Add significands

- 10.015×10^1

3. Normalize result & check for over/underflow

- 1.0015×10^2

4. Round

- 1.002×10^2

Floating-point Addition: Binary

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

Floating-point Addition: Binary

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points

  - Shift number with *smaller exponent*
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

Floating-point Addition: Binary

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points

- Shift number with *smaller exponent*
- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. Add significands

- $0.001_2 \times 2^{-1}$

Floating-point Addition: Binary

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points

- Shift number with *smaller exponent*
- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. Add significands

- $0.001_2 \times 2^{-1}$

3. Normalize result & check for over/underflow

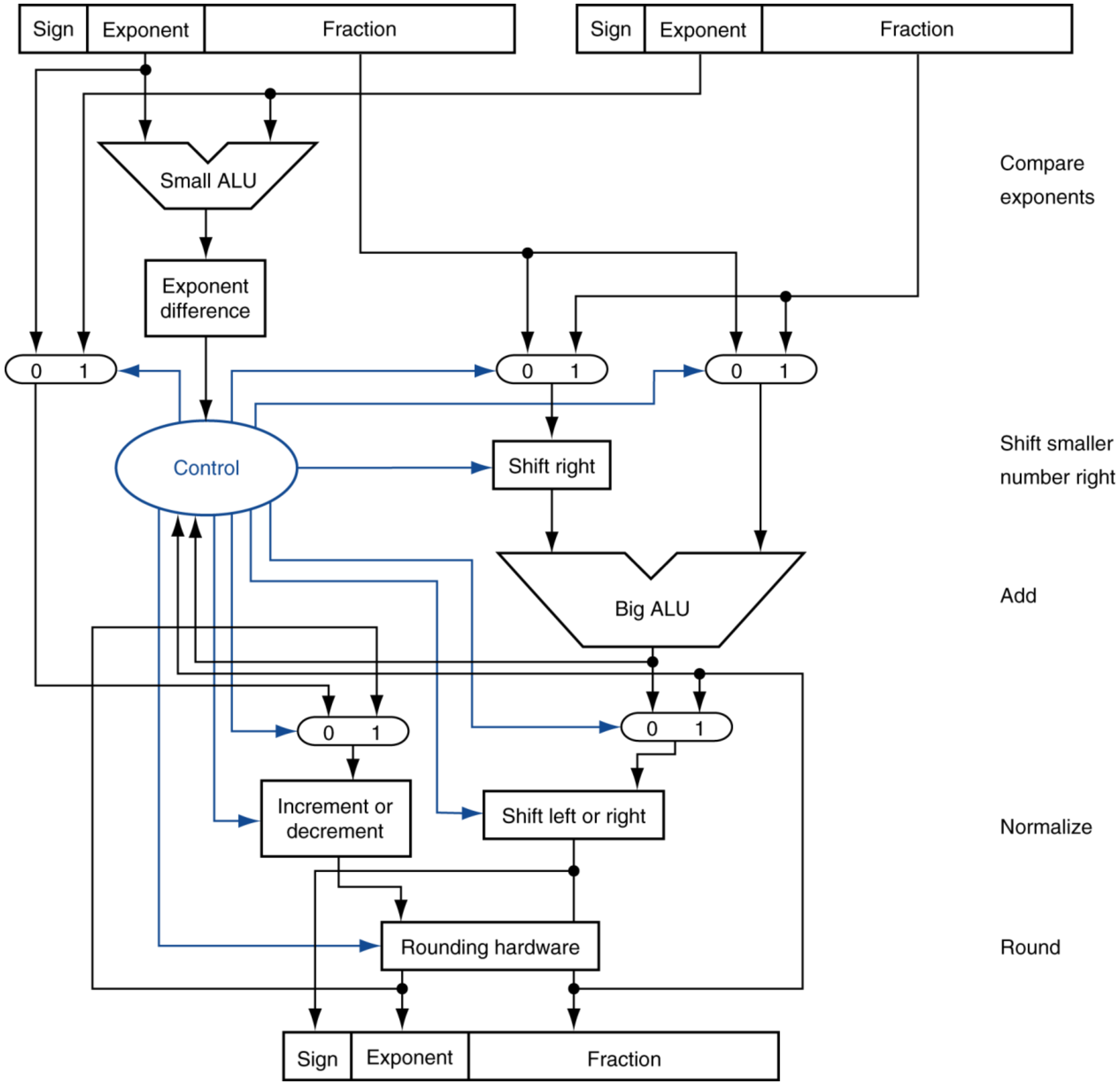
- $1.000_2 \times 2^{-4}$, with no over/underflow

Check $-126 \le -4 \le +127$
in case of a single precision

4. Round

- $1.000_2 \times 2^{-4} = 0.0625$

Floating-point Adder Hardware



Skip the details

Exercise

$$1.0110_2 \text{ x } 2^3 + 1.1000_2 \text{ x } 2^2$$

1. Align binary points

- $1.0110_2 \text{ x } 2^3 + 0.1100_2 \text{ x } 2^3$

2. Add significands

- $10.0010_2 \times 2^3$

3. Normalize result & check for over/underflow

- $1.0001_2 \times 2^4$, with no over/underflow

4. Round

- $1.0001_2 \times 2^4$

Floating-point Multiplication

- Consider a 4-digit decimal example

  $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

1. Add exponents

- New exponent = 10 + –5 = 5

2. Multiply significands

- $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$

3. Normalize result & check for over/underflow

- 1.0212×10^{6}

4. Round

- 1.021×10^{6}

5. Determine sign of result from signs of operands

- $+1.021 \times 10^{6}$

Floating-point Multiplication

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)

1. Add exponents

- Unbiased: –1 + –2 = –3
- Biased: (–1 + 127) + (–2 + 127) – 127 = –3 + 127

For biased exponents, subtract bias from sum

2. Multiply significands

- $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$

3. Normalize result & check for over/underflow

- $1.110_2 \times 2^{-3}$ (no change) with no over/underflow

4. Round

- $1.110_2 \times 2^{-3}$ (no change)

5. Determine sign: + sign × – sign ⇒ – sign

- $-1.110_2 \times 2^{-3} = -0.21875$

Floating-point Instructions in MIPS

F1	F0
…	…
F31	F30

- Separate floating-point registers
  - 32 single-precision: `$f0, $f1, … $f31`
  - Paired for double-precision: `$f0/$f1, $f2/$f3, …`
- FP instructions operate only on FP registers

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than double precision

In MIPS Reference Data...

OPCODES, BASE CONVERSION, ASCII SYMBOLS

3

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	’	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0
(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

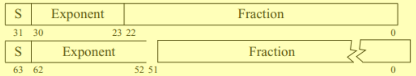
IEEE 754 FLOATING-POINT STANDARD

4

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

where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:

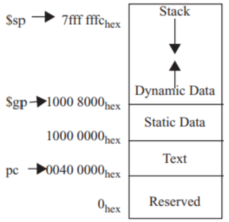


IEEE 754 Symbols

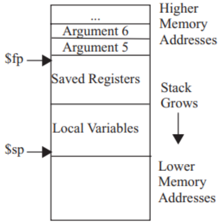
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

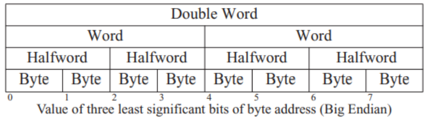
MEMORY ALLOCATION



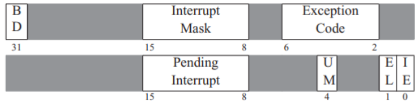
STACK FRAME



DATA ALIGNMENT



EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

MIPS Reference Data Card (“Green Card”) 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

IEEE 754 FLOATING-POINT STANDARD

4

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

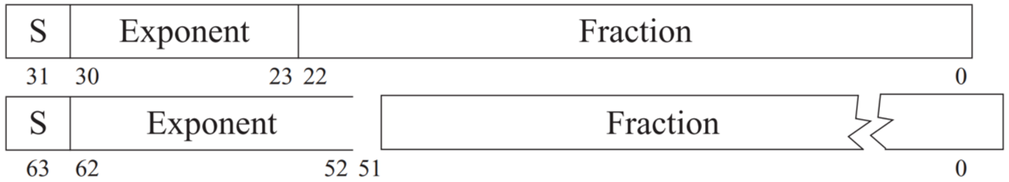
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047



Question?