COMPUTER SYSTEMS ARCHITECTURE SEMINAR 0x03

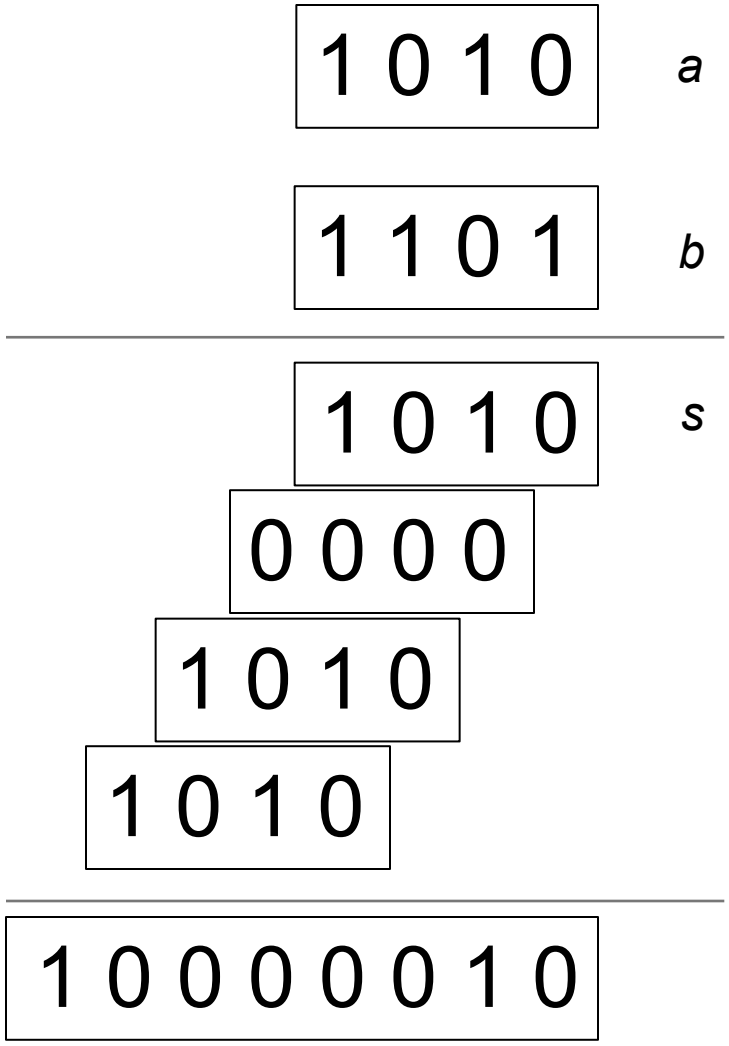
SEMINAR SUPPORT NOTES

Cristian Rusu, Bogdan Macovei

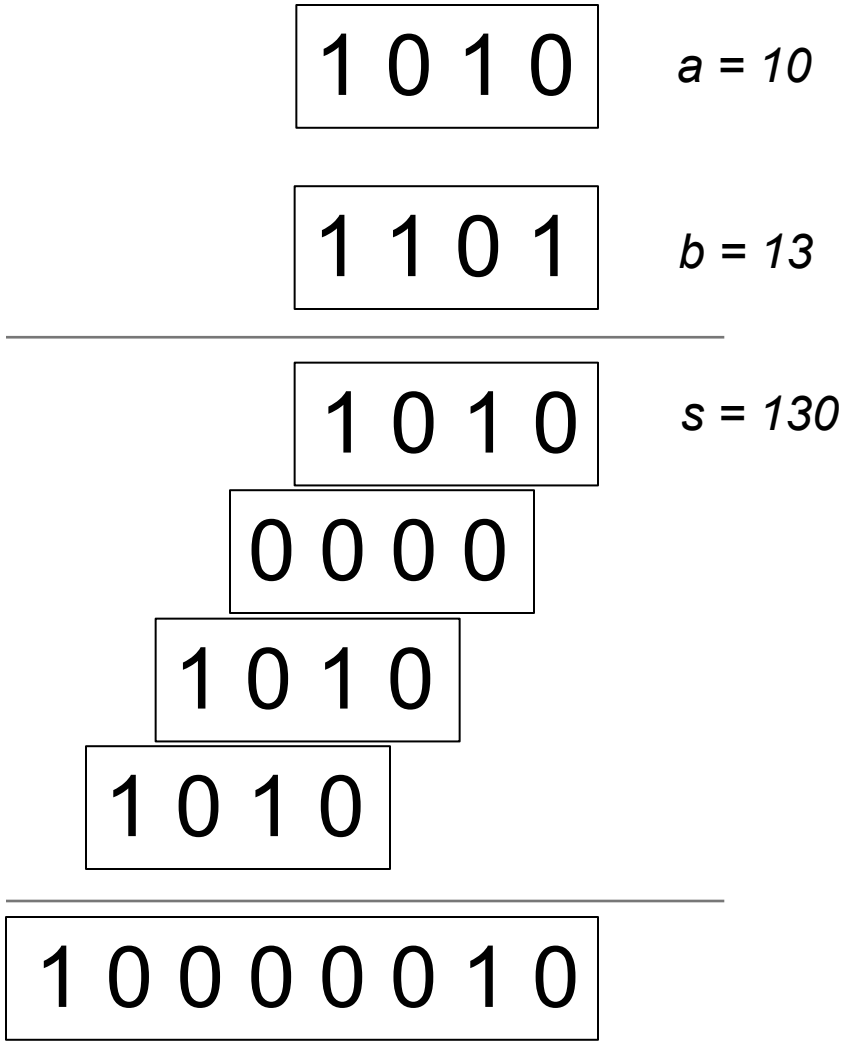
NUMBER MULTIPLICATION, EX. 1

1 0 1 0	*a*
1 1 0 1	*b*
	s

NUMBER MULTIPLICATION, EX. 1



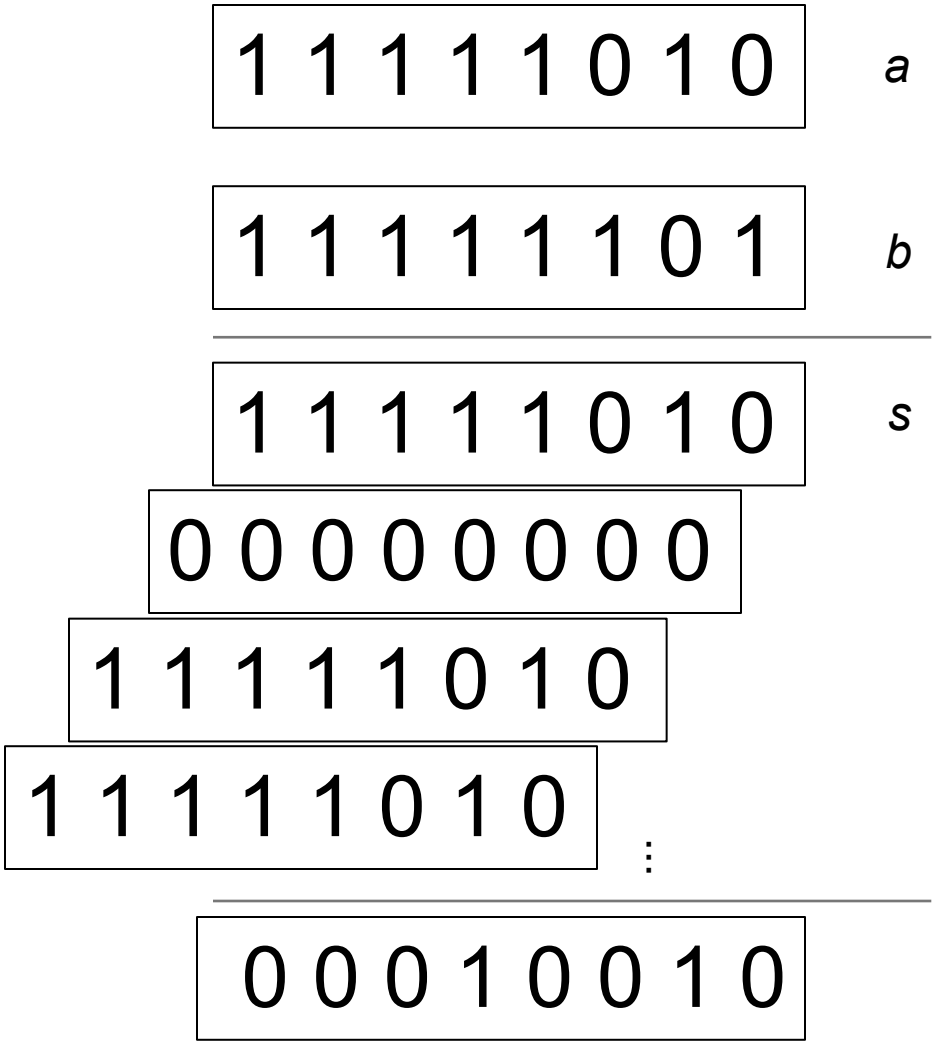
NUMBER MULTIPLICATION, EX. 1



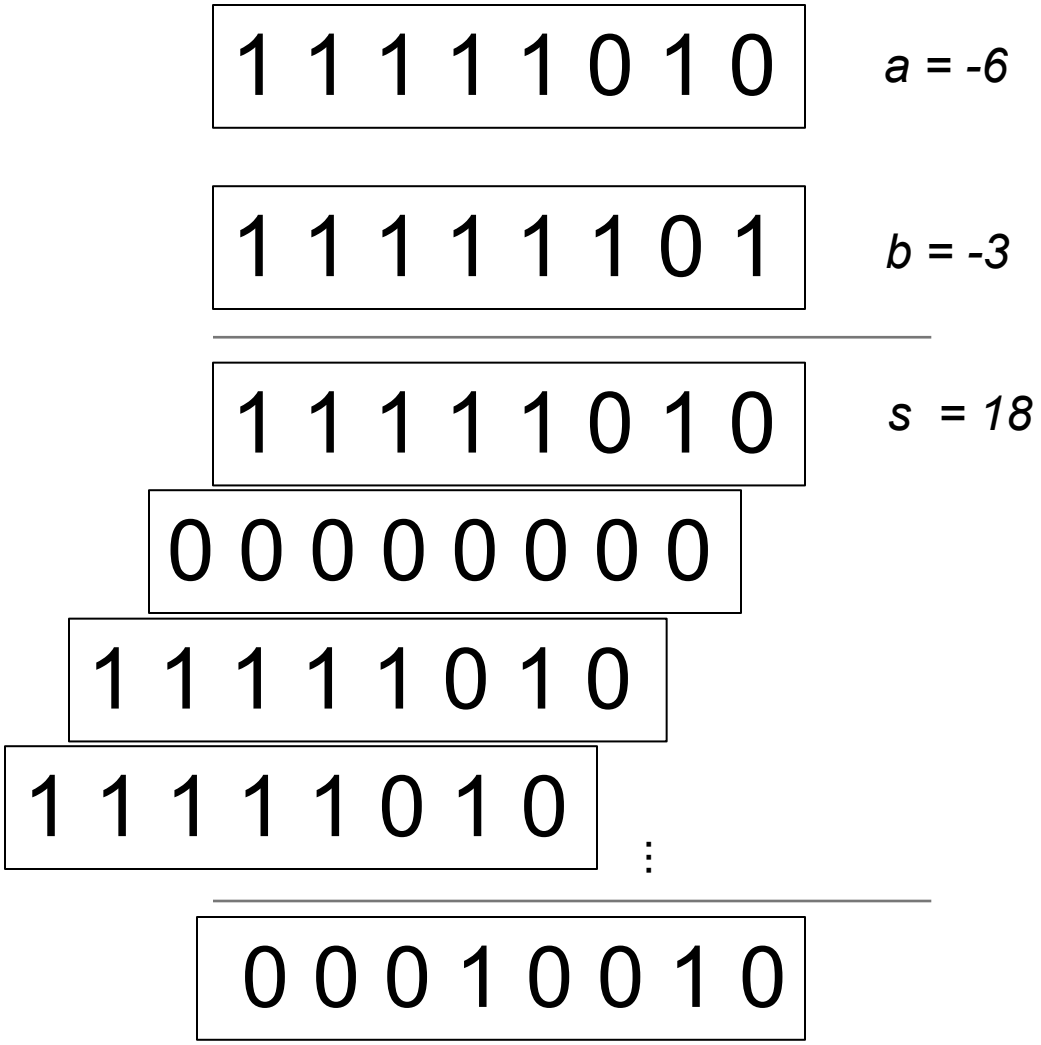
NUMBER MULTIPLICATION, EX. 2

1 0 1 0	*a*
1 1 0 1	*b*
0 0 0 0	*s*

NUMBER MULTIPLICATION, EX. 2



NUMBER MULTIPLICATION, EX. 2



FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution:
- **a x 16**
  - solution:
- **a x 3**
  - solution:
- **a x 7**
  - solution:
- **a / 8**
  - solution:
- **a mod 16**
  - solution:
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution:
- **a x 3**
  - solution:
- **a x 7**
  - solution:
- **a / 8**
  - solution:
- **a mod 16**
  - solution:
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution: a << 4
- **a x 3**
  - solution:
- **a x 7**
  - solution:
- **a / 8**
  - solution:
- **a mod 16**
  - solution:
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution: a << 4
- **a x 3**
  - solution: a << 1 + a
- **a x 7**
  - solution:
- **a / 8**
  - solution:
- **a mod 16**
  - solution:
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution: a << 4
- **a x 3**
  - solution: a << 1 + a
- **a x 7**
  - solution: a << 3 – a
- **a / 8**
  - solution:
- **a mod 16**
  - solution:
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution: a << 4
- **a x 3**
  - solution: a << 1 + a
- **a x 7**
  - solution: a << 3 – a
- **a / 8**
  - solution: a >> 3
- **a mod 16**
  - solution:
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution: a << 4
- **a x 3**
  - solution: a << 1 + a
- **a x 7**
  - solution: a << 3 – a
- **a / 8**
  - solution: a >> 3
- **a mod 16**
  - solution: a & 0x000F
- **a x 72**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a x 2**
  - solution: a << 1, or a + a
- **a x 16**
  - solution: a << 4
- **a x 3**
  - solution: a << 1 + a
- **a x 7**
  - solution: a << 3 – a
- **a / 8**
  - solution: a >> 3
- **a mod 16**
  - solution: a & 0x000F
- **a x 72**
  - solution: a << 6 + a << 3

FAST MULTIPLICATION, EX. 3

- **a mod 16**
  - solution: a & 0x000F
- **a div 16**
  - solution:

FAST MULTIPLICATION, EX. 3

- **a mod 16**
  - solution: a & 0x000F
- **a div 16**
  - solution: (a & FFF0) >> 4, or simply a >> 4
  - also: a = a & FFF0 + a & 000F = div by 16 + mod by 16

FAST MULTIPLICATION, EX. 4

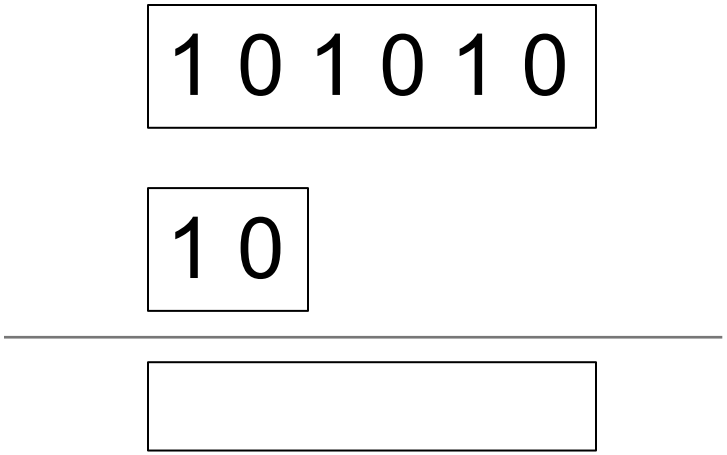
- **a x 2**
  - solution:

FAST MULTIPLICATION, EX. 4

- **a x 2**
  - solution: a = (a < 0) ? -((-a) << 1)) : a << 1

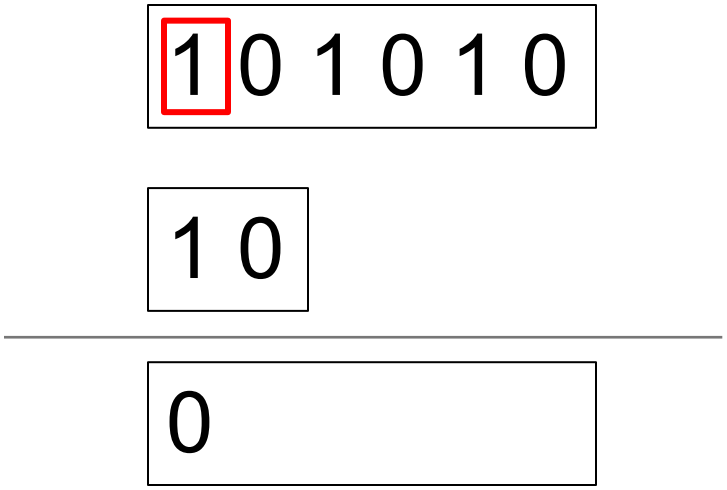
DIVISION, EX. 5

- **101010 / 10**



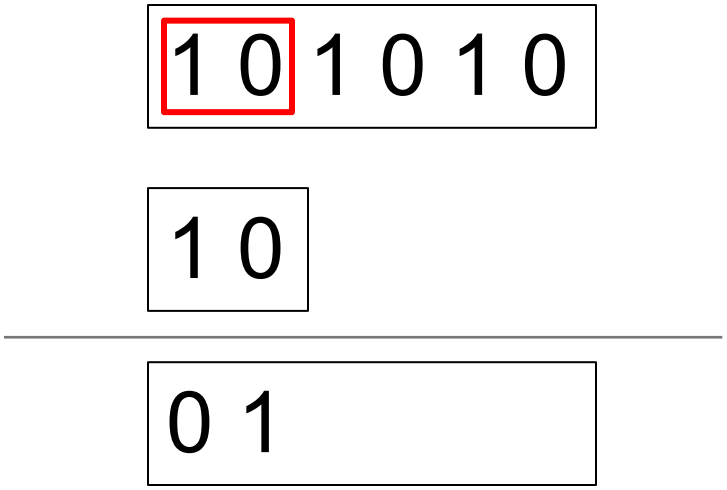
DIVISION, EX. 5

- **101010 / 10**



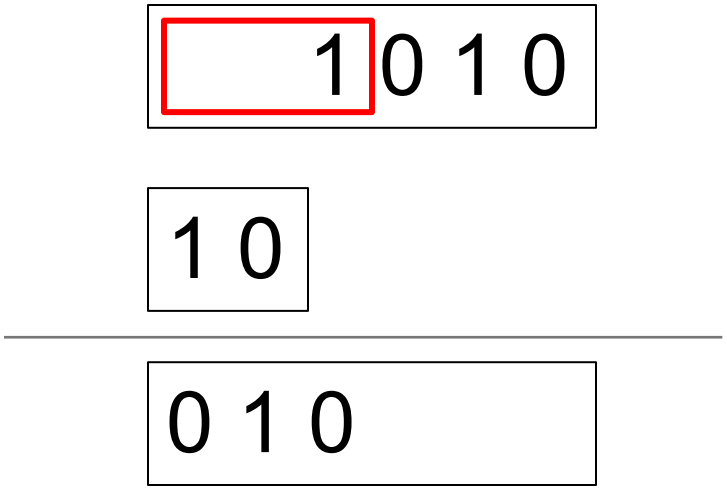
DIVISION, EX. 5

- **101010 / 10**



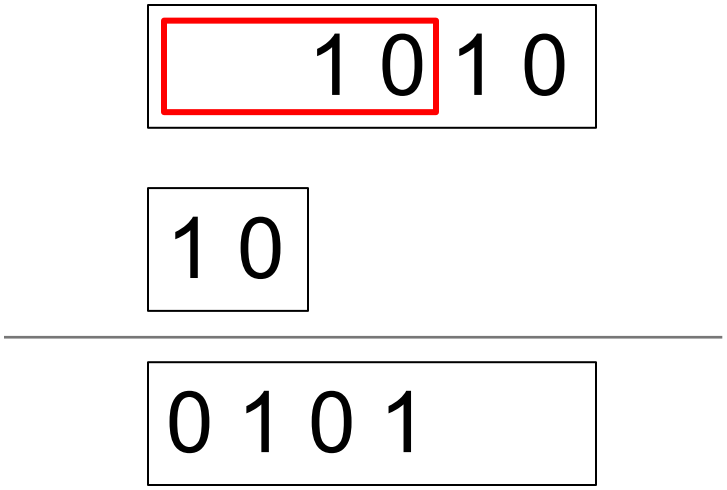
DIVISION, EX. 5

- **101010 / 10**



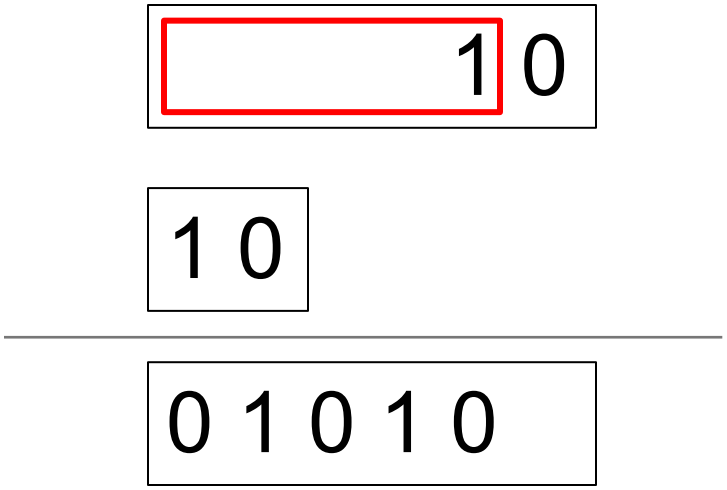
DIVISION, EX. 5

- **101010 / 10**



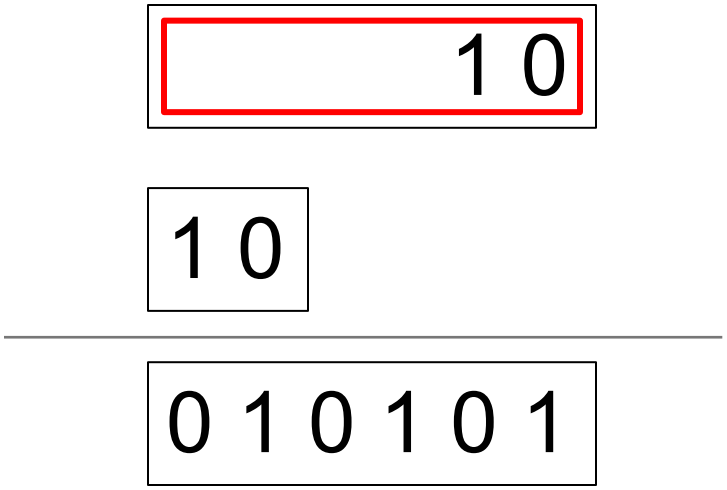
DIVISION, EX. 5

- **101010 / 10**



DIVISION, EX. 5

- **101010 / 10**



DIVISION, EX. 5

- **101010 / 10**

1 0 1 0 1 0	*a*
1 0	*b*
0 1 0 1 0 1	*s*

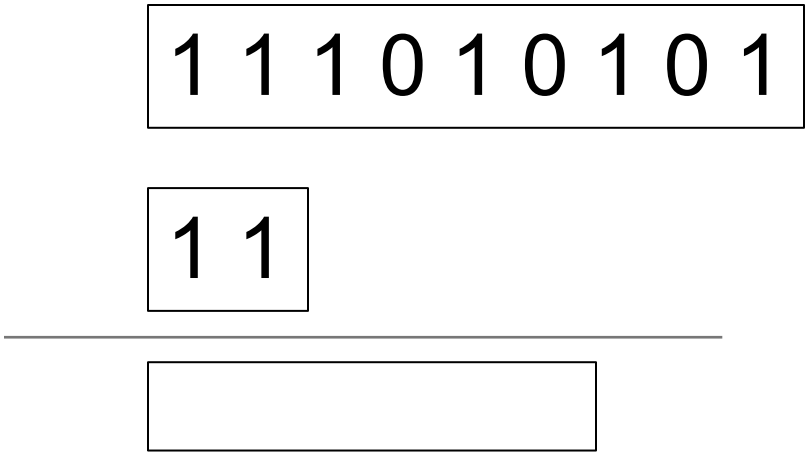
DIVISION, EX. 5

- **101010 / 10**

1 0 1 0 1 0	*a = 42*
1 0	*b = 2*
0 1 0 1 0 1	*s = 21*

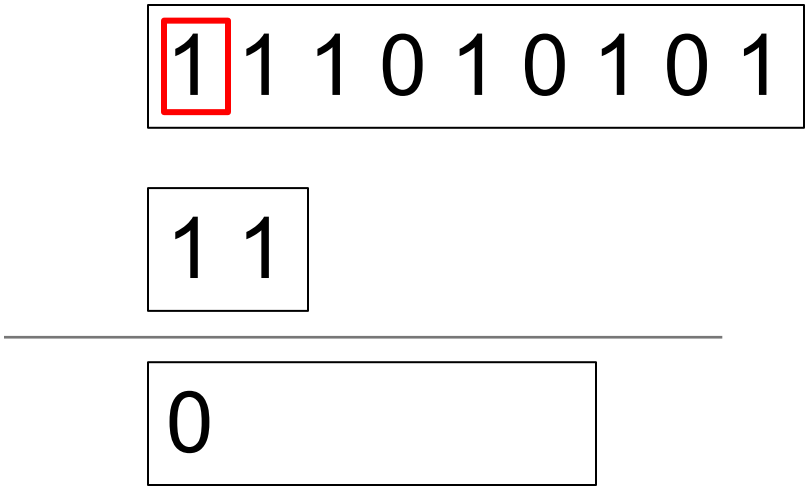
DIVISION, EX. 5

- **111010101 / 11**



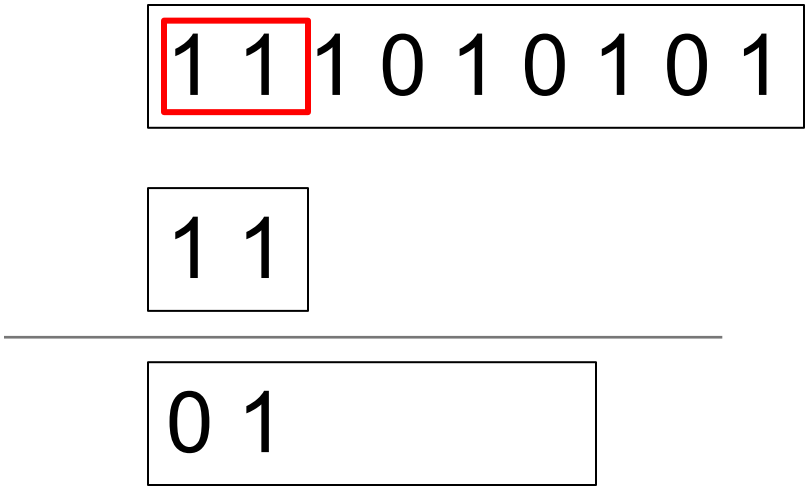
DIVISION, EX. 5

- **111010101 / 11**



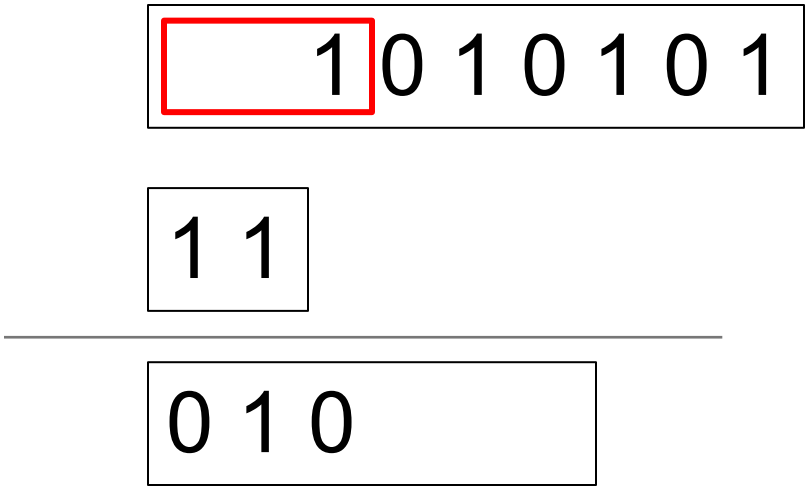
DIVISION, EX. 5

- **111010101 / 11**



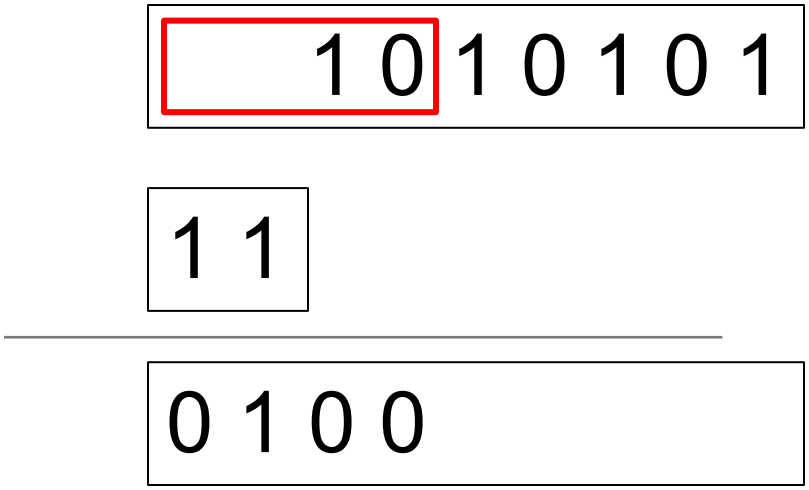
DIVISION, EX. 5

- **111010101 / 11**



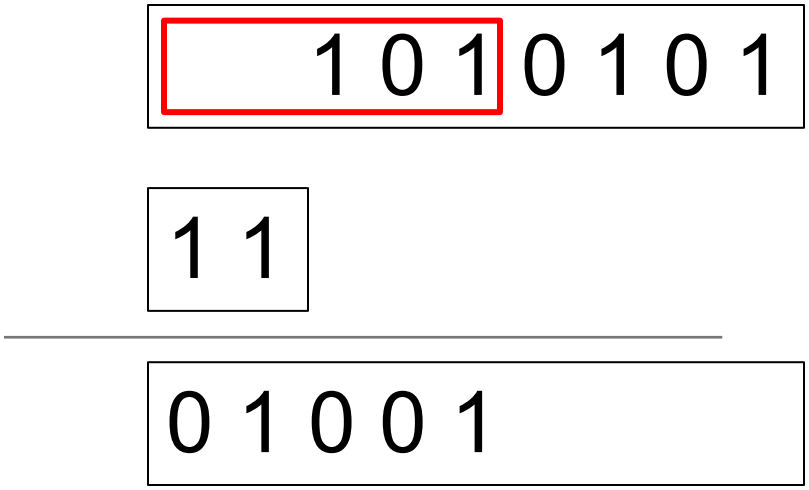
DIVISION, EX. 5

- **111010101 / 11**



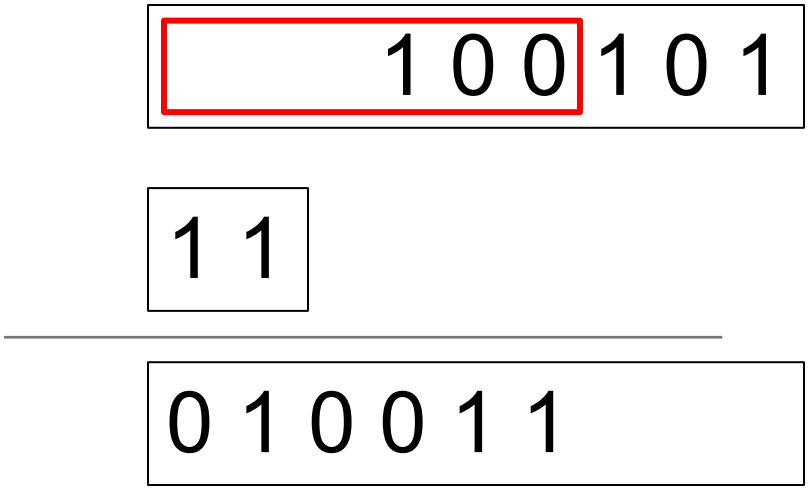
DIVISION, EX. 5

- **111010101 / 11**



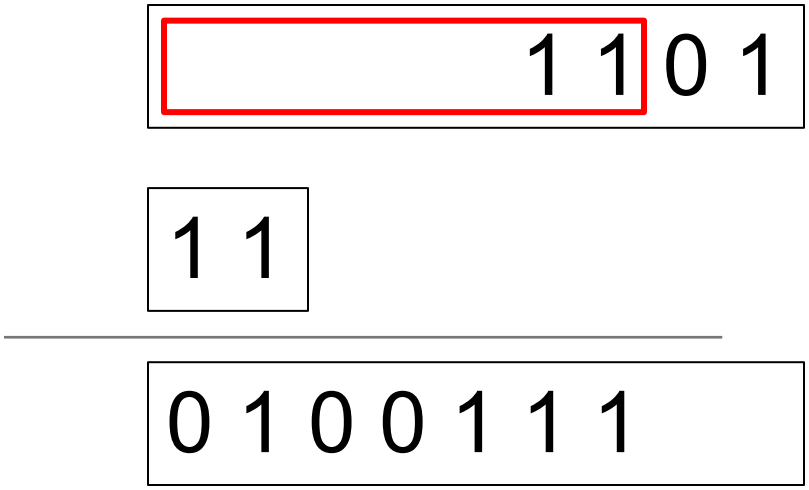
DIVISION, EX. 5

- **111010101 / 11**



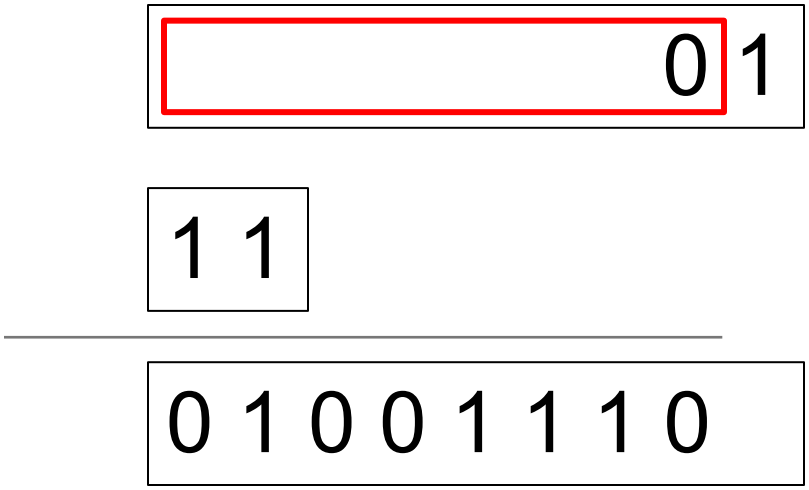
DIVISION, EX. 5

- **111010101 / 11**



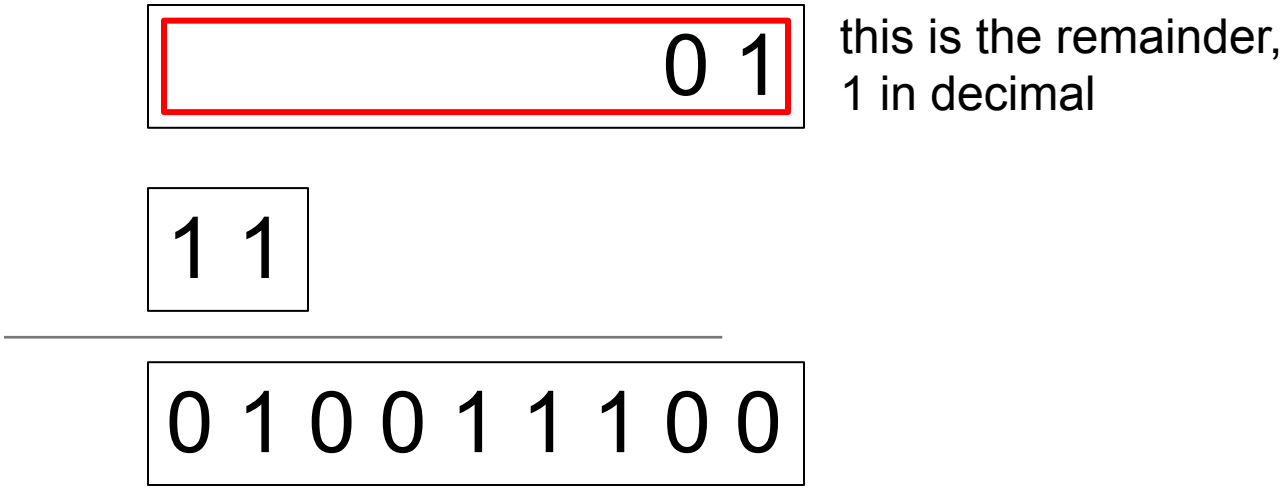
DIVISION, EX. 5

- **111010101 / 11**



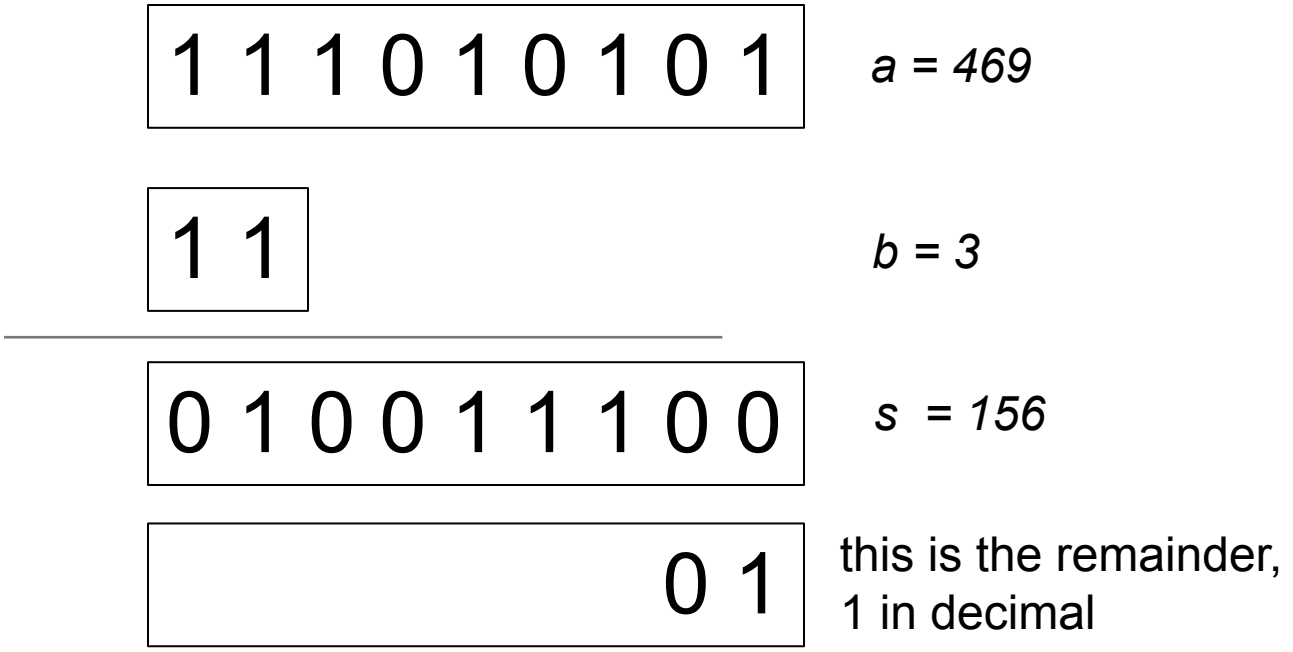
DIVISION, EX. 5

- **111010101 / 11**

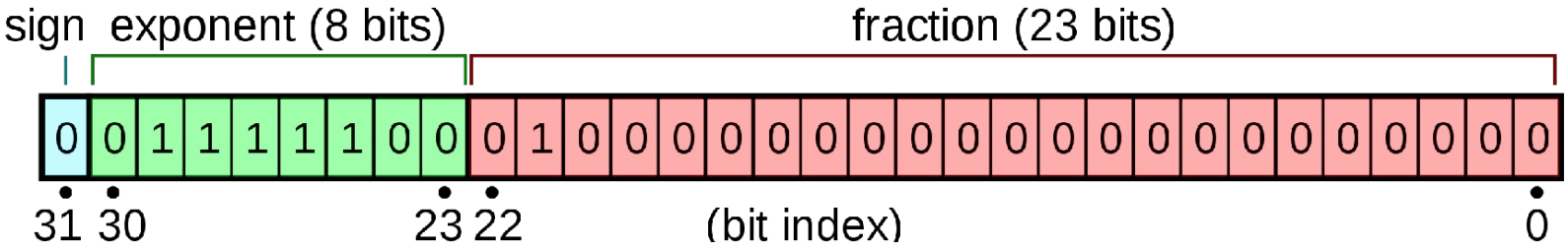


DIVISION, EX. 5

- **111010101 / 11**

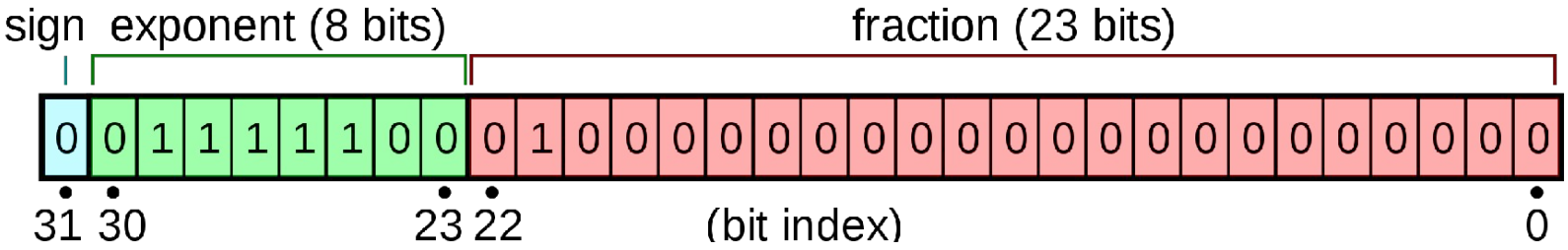


CONVERSION TO IEEE FP, EX. 7



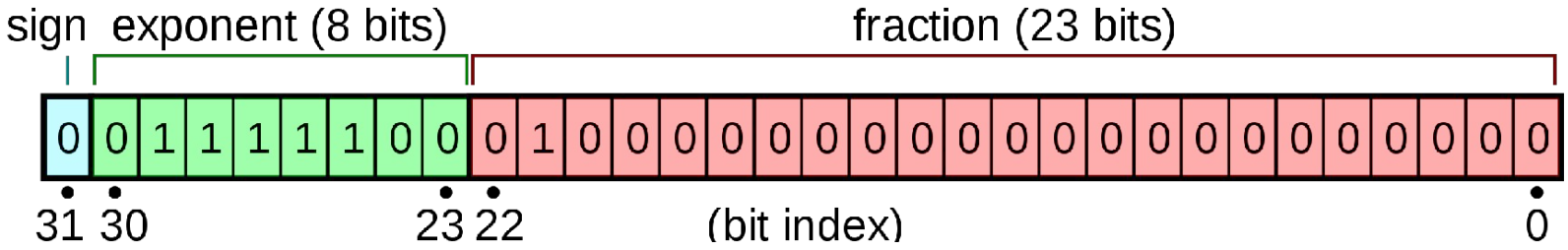
- -1313.3125
  - the integer part is: ?
  - the fractional part: ?

CONVERSION TO IEEE FP, EX. 7



- -1313.3125
  - the integer part is: 1313
  - the fractional part: 0.3125
    - 0.3125 × 2 = 0.625 => 0
    - 0.625 × 2 = 1.25 => 1
    - 0.25 × 2 = 0.5 => 0
    - 0.5 × 2 = 1.0 => 1
  - therefore, $1313.3125_{10} = ?_2$

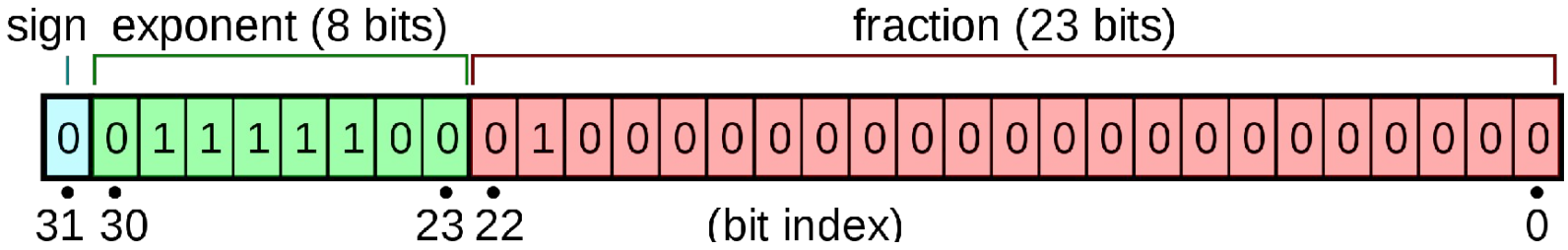
CONVERSION TO IEEE FP, EX. 7



- -1313.3125
  - the integer part is: 1313
  - the fractional part: 0.3125
    - 0.3125 × 2 = 0.625 => 0
    - 0.625 × 2 = 1.25 => 1
    - 0.25 × 2 = 0.5 => 0
    - 0.5 × 2 = 1.0 => 1
  - therefore, $1313.3125_{10} = 10100100001.0101_2$
  - normalization: ?

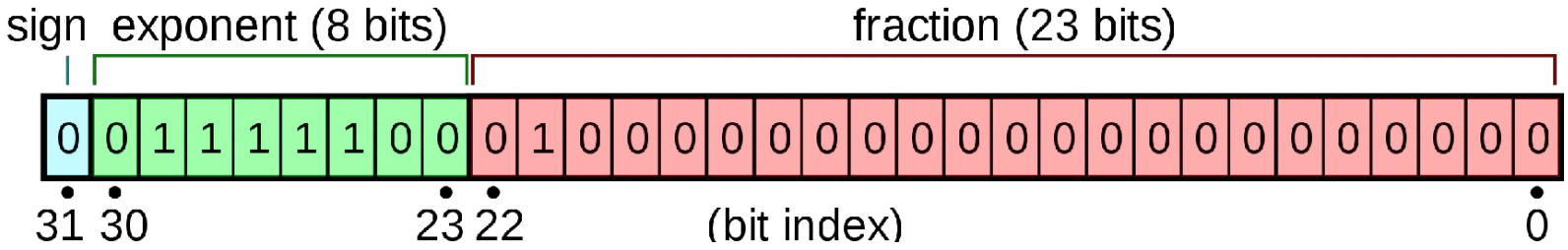
  - the mantissa is ?
  - the exponent is ?
  - the sign is ?

CONVERSION TO IEEE FP, EX. 7



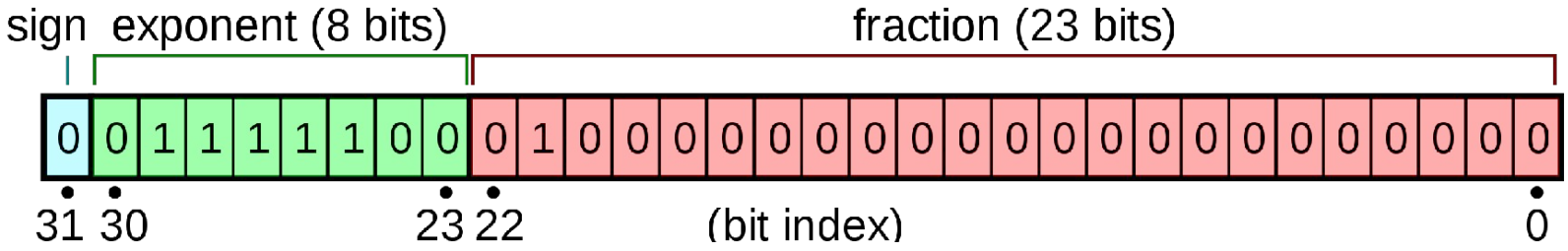
- -1313.3125
  - the integer part is: 1313
  - the fractional part: 0.3125
    - 0.3125 × 2 = 0.625 => 0
    - 0.625 × 2 = 1.25 => 1
    - 0.25 × 2 = 0.5 => 0
    - 0.5 × 2 = 1.0 => 1
  - therefore, $1313.3125_{10} = 10100100001.0101_2$
  - normalization: $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$
  - the mantissa is ?
  - the exponent is ?
  - the sign is ?

CONVERSION TO IEEE FP, EX. 7



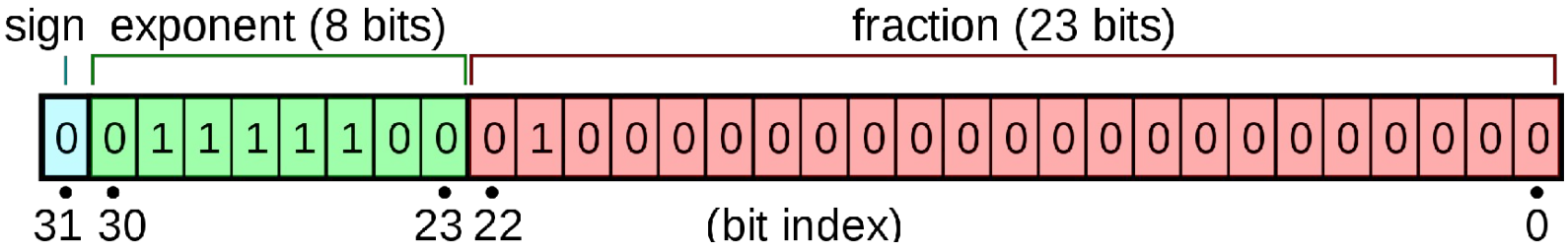
- -1313.3125
  - the integer part is: 1313
  - the fractional part: 0.3125
    - 0.3125 × 2 = 0.625 => 0
    - 0.625 × 2 = 1.25 => 1
    - 0.25 × 2 = 0.5 => 0
    - 0.5 × 2 = 1.0 => 1
  - therefore, $1313.3125_{10} = 10100100001.0101_2$
  - normalization: $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$
  - the mantissa is 01001000010101000000000
  - the exponent is $10 + 127 = 137 = 10001001_2$
  - the sign is 1

CONVERSION TO IEEE FP, EX. 8



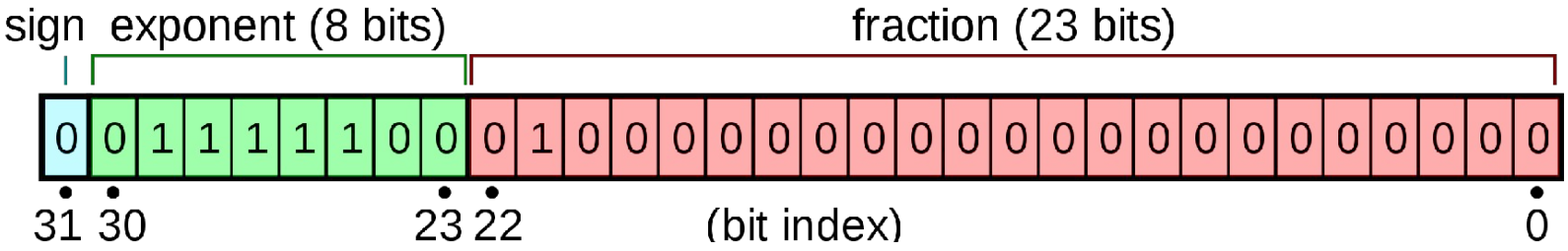
- **we compute abs(a)**

CONVERSION TO IEEE FP, EX. 8



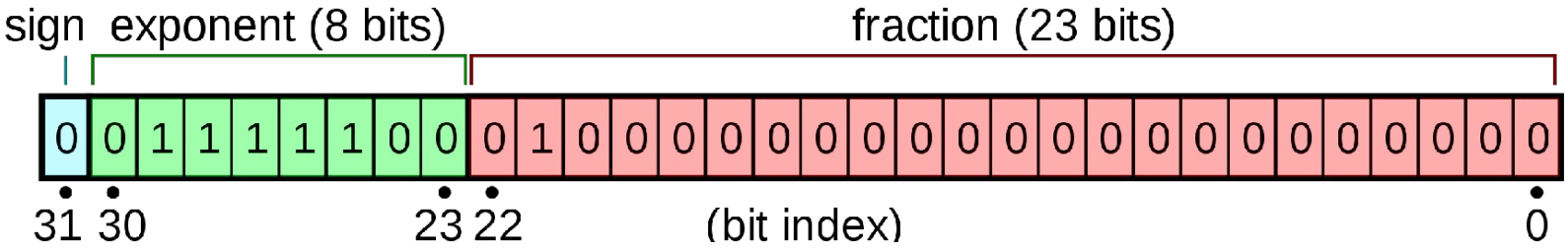
- **we compute abs(a)**
  - **solution: a = a & ~(1 << 31)**

CONVERSION TO IEEE FP, EX. 8



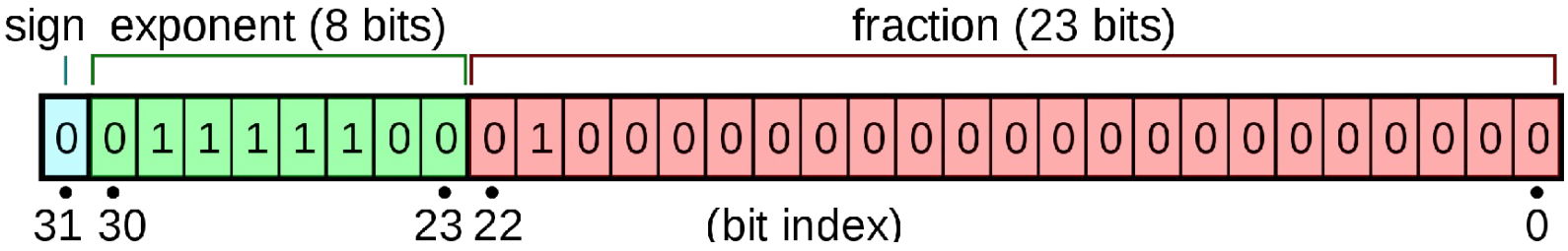
- **change the sign of a**

CONVERSION TO IEEE FP, EX. 8



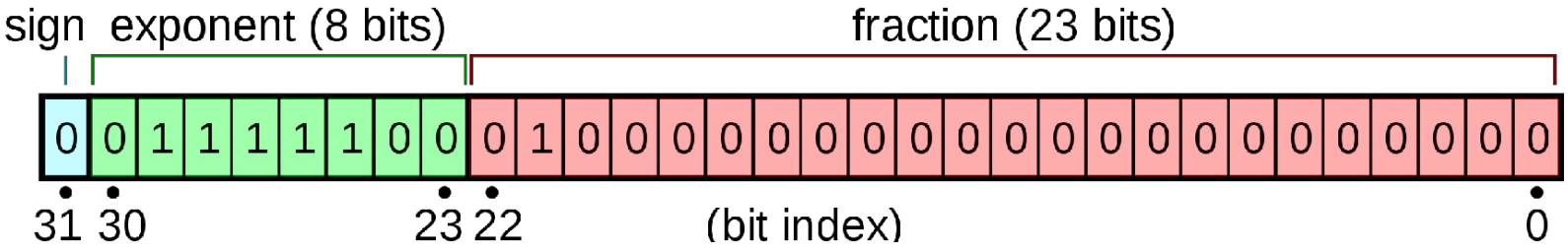
- **change the sign of a**
  - **solution: a = a ^ (1 << 31)**

CONVERSION TO IEEE FP, EX. 8



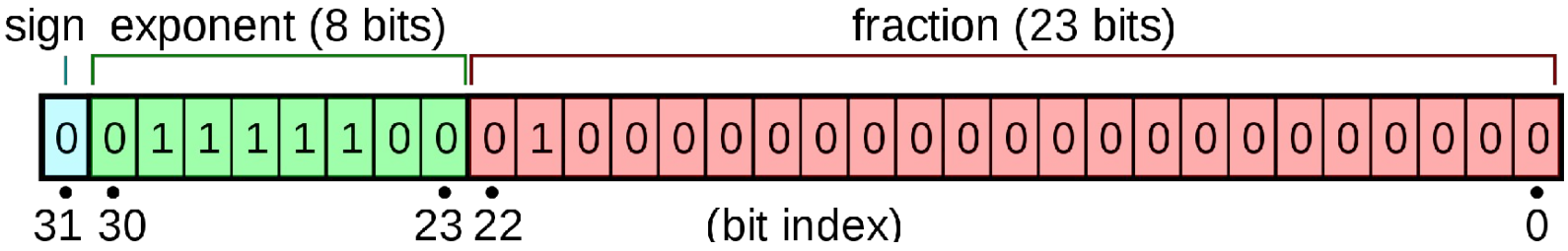
- **divide a by 4**

CONVERSION TO IEEE FP, EX. 8



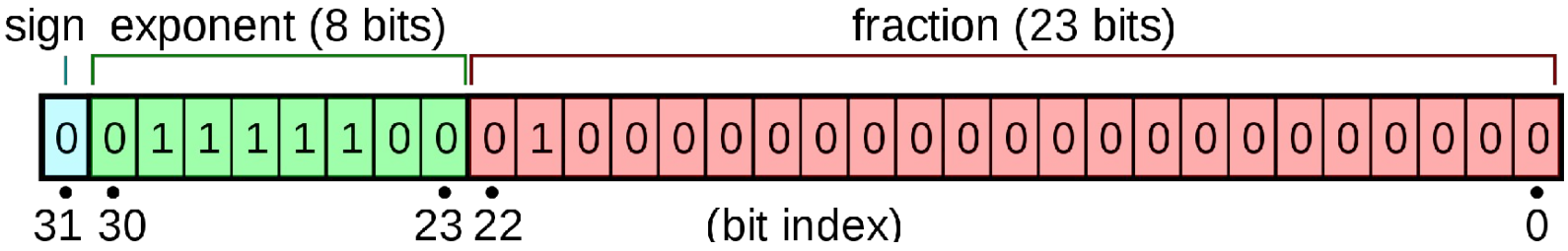
- **divide a by 4**
  - **solution:**
    - **we need the exponent; where is it located?**
      - **MASK =**
    - **extract exponent =**
    - **if exponent > 1 then ...**
    - **we need to update a**
      - **a =**

CONVERSION TO IEEE FP, EX. 8



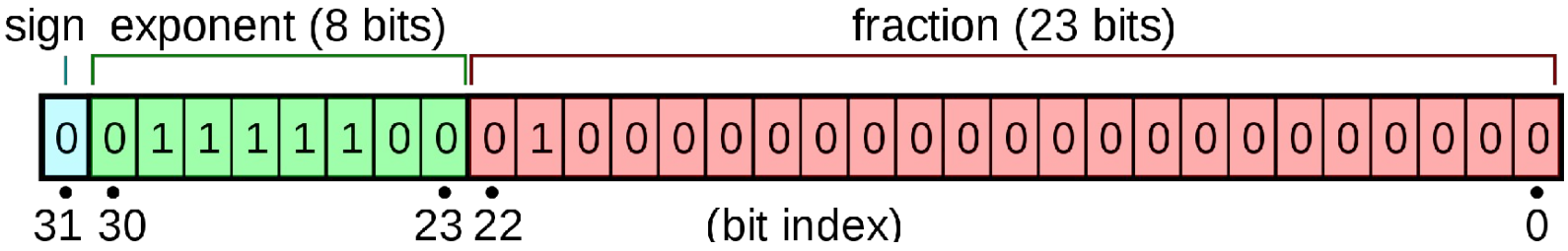
- **divide a by 4**
  - **solution:**
    - **we need the exponent; where is it located?**
      - **MASK = 0x7F800000**
    - **extract exponent =**
    - **if exponent > 1 then ...**
    - **we need to update a**
      - **a =**

CONVERSION TO IEEE FP, EX. 8



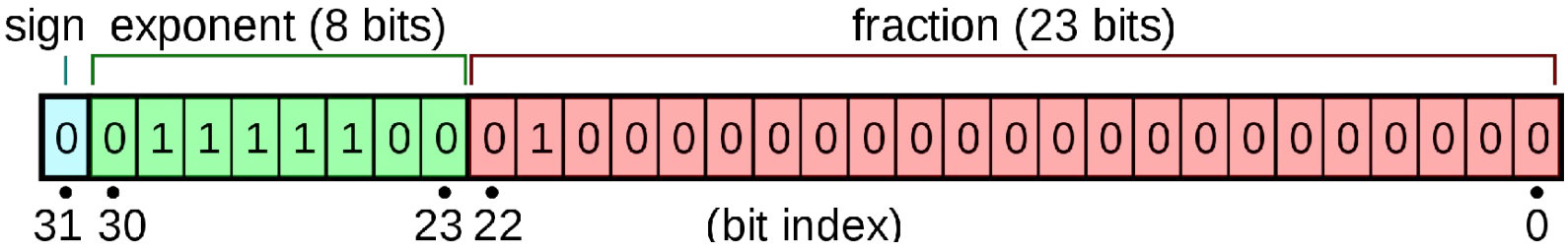
- **divide a by 4**
  - **solution:**
    - **we need the exponent; where is it located?**
      - **MASK = 0x7F800000**
    - **extract exponent = (a & MASK) >> 23**
    - **if exponent > 1 then ...**
    - **we need to update a**
      - **a =**

CONVERSION TO IEEE FP, EX. 8



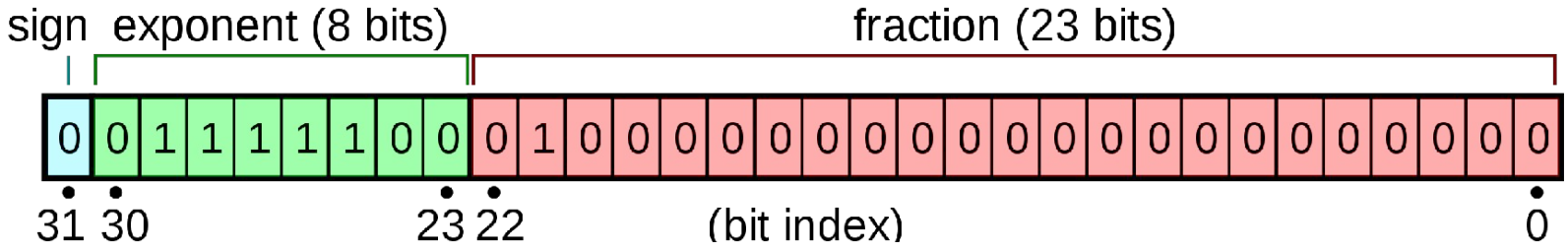
- **divide a by 4**
  - **solution:**
    - **we need the exponent; where is it located?**
      - **MASK = 0x7F800000**
    - **extract exponent = (a & MASK) >> 23**
    - **if exponent > 1 then exponent = exponent – 2, otherwise a = 0**
    - **we need to update a**
      - **a =**

CONVERSION TO IEEE FP, EX. 8



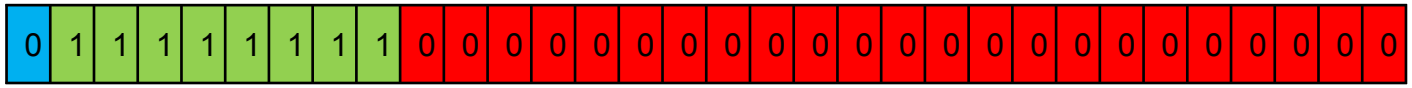
- **divide a by 4**
  - **solution:**
    - **we need the exponent; where is it located?**
      - **MASK = 0x7F800000**
    - **extract exponent = (a & MASK) >> 23**
    - **if exponent > 1 then exponent = exponent – 2, otherwise a = 0**
    - **we need to update a**
      - **a = (a & ~MASK) | (exponent << 23)**

CONVERSION TO IEEE FP, EX. 8



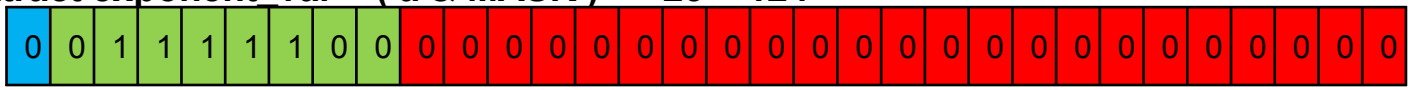
- **divide a by 4**
  - **solution:**
    - **we need the exponent; where is it located?**
      - **MASK = 0x7F800000**

MASK

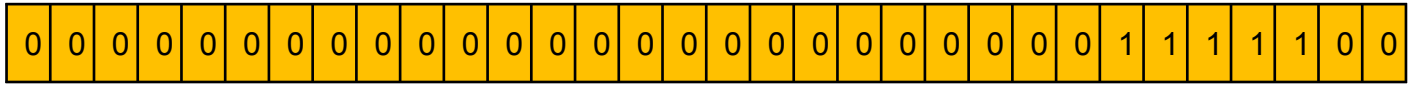


- **extract exponent_var = (a & MASK) >> 23 = 124**

a & MASK

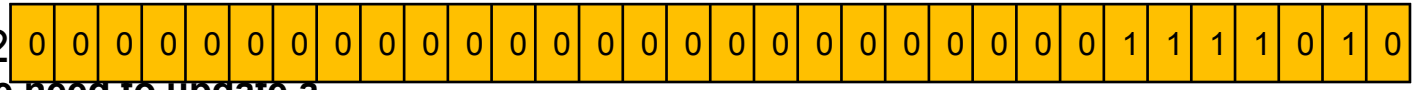


(a & MASK) >> 23



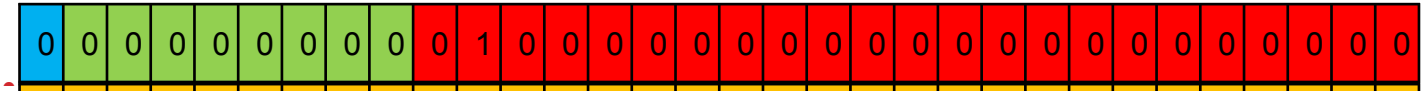
- **if exponent_var > 1 then exponent_var = exponent_var – 2, otherwise a = 0**

exponent_var – 2 = 122

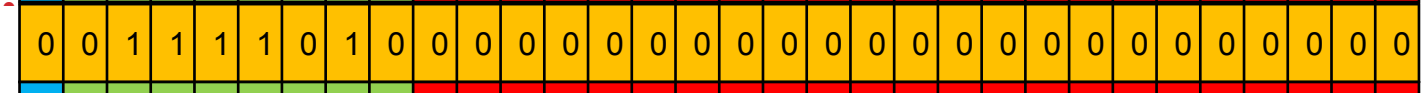


- **we need to update a**
  - **a = (a & ~MASK) | (exponent_var << 23)**

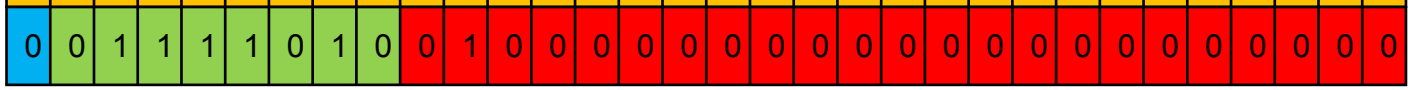
a & ~MASK



exponent_var << 23



(a & ~MASK) | (exponent_var << 23)



FP IN HEX, EX. 9

d) 0xDEADBEEF = 0b11011110101011011011111011101111

- S = 1
- E = 10111101
- M = 01011011011111011101111
- (-1)^S 1.M * 2^{E-127} = (-1) 1.01011011011111011101111 $2^{189-127}$
  = - 1.01011011011111011101111 2^{62}
  = -6259853398707798000

e) 0x44361000 = 0b01000100001101100001000000000000

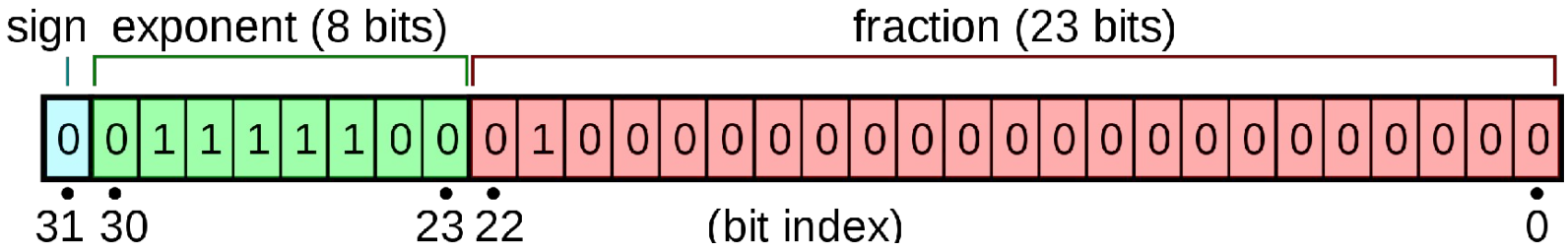
- S = 0
- E = 10001000
- M = 01101100001000000000000
- (-1)^S 1.M * 2^{E-127} = (1) 1. 01101100001000000000000 $2^{136-127}$
  = 1. 01101100001000000000000 2^{9}
  = 728.25

j) 0xC00010FF = 0b11000000000000000010000111111111

- S = 1
- E = 10000000
- M = 00000000010000111111111
- (-1)^S 1.M * 2^{E-127} = (-1) 1.00000000010000111111111 $2^{128-127}$
  = -1.00000000010000111111111 2^{1}
  = -2.001037359237671

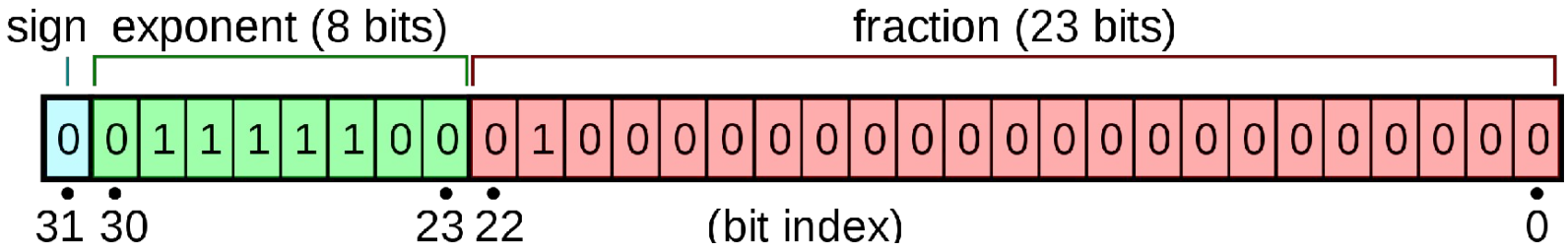
0xFFFFFFFF and 0xFFFF0000 are NaN (by convention),

ZERO IN IEEE FP, EX. 10



- **set s = 0, e = 0, f = 0**

ZERO IN IEEE FP, EX. 10



- **set s = 0, e = 0, f = 0**
- **a = $(-1)^0 \times 1.00\ldots00 \times 2^{-127} = 2^{-127} \neq 0$**

IEEE 754 FP ADDITION, EX. 11

- 0.2 + 0.3

IEEE 754 FP ADDITION, EX. 11

- **0.2 + 0.3**

- **first step, convert each number to the format**
  - 0.2 = + 1.10011001100110011001100 x 2^{-3} = 0.19999998807907104
  - 0.3 = + 1.00110011001100110011001 x 2^{-2} = 0.29999998211860657

IEEE 754 FP ADDITION, EX. 11

- **0.2 + 0.3**

- **first step, convert each number to the format**
  - $0.2 = + 1.10011001100110011001100 \times 2^{-3} = 0.19999998807907104$
  - $0.3 = + 1.00110011001100110011001 \times 2^{-2} = 0.29999998211860657$

- **second step, alignment**
  - $0.2 = + 0.11001100110011001100110|000 \times 2^{-2}$
  - $0.3 = + 1.00110011001100110011001|000 \times 2^{-2}$

IEEE 754 FP ADDITION, EX. 11

- **0.2 + 0.3**

- **first step, convert each number to the format**
  - $0.2 = + 1.10011001100110011001100 \times 2^{-3} = 0.19999998807907104$
  - $0.3 = + 1.00110011001100110011001 \times 2^{-2} = 0.29999998211860657$

- **second step, alignment**
  - $0.2 = + 0.11001100110011001100110|000 \times 2^{-2}$
  - $0.3 = + 1.00110011001100110011001|000 \times 2^{-2}$

- **third step, add**
  - $0.2 + 0.3 = 1.11111111111111111111111|000 \times 2^{-2}$

IEEE 754 FP ADDITION, EX. 11

- **0.2 + 0.3**

- **first step, convert each number to the format**
  - 0.2 = + 1.10011001100110011001100 x 2^{-3} = 0.19999998807907104
  - 0.3 = + 1.00110011001100110011001 x 2^{-2} = 0.29999998211860657

- **second step, alignment**
  - 0.2 = + 0.11001100110011001100110|000 x 2^{-2}
  - 0.3 = + 1.00110011001100110011001|000 x 2^{-2}

- **third step, add**
  - 0.2 + 0.3 = 1.11111111111111111111111|000 x 2^{-2}

- **fourth step, normalization (if necessary)**
  - 0.2 + 0.3 = 1.11111111111111111111111|00 x 2^{-2}

IEEE 754 FP ADDITION, EX. 11

- **0.2 + 0.3**

- **first step, convert each number to the format**
  - 0.2 = + 1.10011001100110011001100 x 2^{-3} = 0.19999998807907104
  - 0.3 = + 1.00110011001100110011001 x 2^{-2} = 0.2999999821860657

- **second step, alignment**
  - 0.2 = + 0.1100110011001100110011|000 x 2^{-2}
  - 0.3 = + 1.00110011001100110011001|000 x 2^{-2}

- **third step, add**
  - 0.2 + 0.3 = 1.11111111111111111111111|000 x 2^{-2}

- **fourth step, normalization (if necessary)**
  - 0.2 + 0.3 = 1.11111111111111111111111|00 x 2^{-2}

- **final step: + 1.11111111111111111111111 × 2^{-2} = 0.4999999701976776**

FAST DIVISION, EX. 12

- **a / 19**

FAST DIVISION, EX. 12

- a / 19

$$a \times \frac{1}{19} \approx \frac{a \times \frac{2938661835}{2^{32}} + \frac{a - a \times \frac{2938661835}{2^{32}}}{2^1}}{2^4}$$

$$a \times \frac{1}{19} \approx (a \times 29386618352^{-32} + (a - a \times 2938661835 \times 2^{-32}) \times 2^{-1}) \times 2^{-4}$$

$$a \times \frac{1}{19} \approx a \times \frac{7233629131}{137438953472}$$

- 1/19 = 0.05263157894
- 7233629131 / 137438953472 = 0.05263157895

FAST DIVISION, EX. 12

- **a / 19**

$$a \times \frac{1}{19} \approx \frac{a \times \frac{2938661835}{2^{32}} + \frac{a - a \times \frac{2938661835}{2^{32}}}{2^1}}{2^4}$$

$$a \times \frac{1}{19} \approx (a \times 29386618352^{-32} + (a - a \times 2938661835 \times 2^{-32}) \times 2^{-1}) \times 2^{-4}$$

$$a \times \frac{1}{19} \approx a \times \frac{7233629131}{137438953472}$$

- **general solution**

$$\frac{a}{D} \approx \frac{\frac{aC}{2^X} + \frac{a - \frac{aC}{2^X}}{2^Y}}{2^Z}$$

$$\approx \frac{2^{X+Y+Z}}{C \times (2^Y - 1) + 2^Z}$$

x

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution:

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.09999990463256835937 5

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.09999990463256835937 5
- **what is the difference between the computed value and 0.1**
  - solution:

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.099999904632568359375

- **what is the difference between the computed value and 0.1**
  - solution: 0.1 - 0.099999904632568359375

- **what is the timing error after 100 hours of operation**
  - solution:

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.09999990463256835937 5
- **what is the difference between the computed value and 0.1**
  - solution: 0.1 - 0.09999990463256835937 5
- **what is the timing error after 100 hours of operation**
  - solution: 100x60x60x10x(0.1 - 0.09999990463256835937 5) ≈ 0.34

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.099999904632568359375
- **what is the difference between the computed value and 0.1**
  - solution: 0.1 - 0.099999904632568359375
- **what is the timing error after 100 hours of operation**
  - solution: 100x60x60x10x(0.1 - 0.099999904632568359375) ≈ 0.34
- **what is the error if we represent 0.1 in IEEE 754 FP format?**
  - solution:

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.09999990463256835937 5
- **what is the difference between the computed value and 0.1**
  - solution: 0.1 - 0.09999990463256835937 5
- **what is the timing error after 100 hours of operation**
  - solution: 100x60x60x10x(0.1 - 0.09999990463256835937 5) ≈ 0.34
- **what is the error if we represent 0.1 in IEEE 754 FP format?**
  - solution: 100x60x60x10x(0.1 - 0.09999999403953552) ≈ 0.021

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.09999990463256835937 5
- **what is the difference between the computed value and 0.1**
  - solution: 0.1 - 0.09999990463256835937 5
- **what is the timing error after 100 hours of operation**
  - solution: 100x60x60x10x(0.1 - 0.09999990463256835937 5) ≈ 0.34
- **what is the error if we represent 0.1 in IEEE 754 FP format?**
  - solution: 100x60x60x10x(0.1 - 0.0999999940395355 2) ≈ 0.021
- **if SCUD missiles can reach MACH 5, what distance can the missile travel during the computed error interval?**
  - solution:

FAST MISSILES, EX. 19

- **compute the binary approximation**
  - solution: 0.09999990463256835937 5
- **what is the difference between the computed value and 0.1**
  - solution: 0.1 - 0.09999990463256835937 5
- **what is the timing error after 100 hours of operation**
  - solution: 100x60x60x10x(0.1 - 0.09999990463256835937 5) ≈ 0.34
- **what is the error if we represent 0.1 in IEEE 754 FP format?**
  - solution: 100x60x60x10x(0.1 - 0.0999999940395355 2) ≈ 0.021
- **if SCUD missiles can reach MACH 5, what distance can the missile travel during the computed error interval?**
  - solution: 1715 m/s * 0.34 s ≈ 583 m, 1715 m/s * 0.021 s ≈ 36 m

FAST INVERSE SQUARE ROOT, Q III

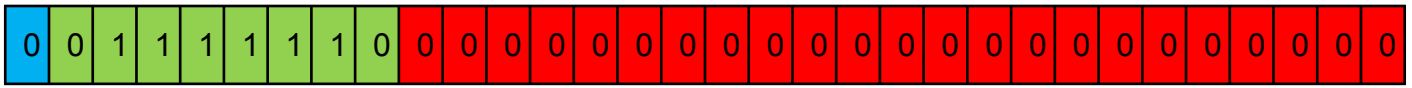
```
552  float Q_rsqrt( float number )
553  {
554          long i;
555          float x2, y;
556          const float threehalfs = 1.5F;
557
558          x2 = number * 0.5F;
559          y  = number;
560          i  = * ( long * ) &y;                                   // evil floating point bit level hacking
561          i  = 0x5f3759df - ( i >> 1 );               // what the duck?
562          y  = * ( float * ) &i;
563          y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564  //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c

FAST INVERSE SQUARE ROOT, Q III

```
552  float Q_rsqrt( float number )
553  {
554          long i;
555          float x2, y;
556          const float threehalfs = 1.5F;
557
558          x2 = number * 0.5F;
559          y  = number;
560          i  = * ( long * ) &y;                       // evil floating point bit level hacking
561          i  = 0x5f3759df - ( i >> 1 );               // what the duck?
562          y  = * ( float * ) &i;
563          y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564  //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

- **the first interesting instruction is on line 560**
- **for example, 0.5 is:**



- **the same bits, but interpreted this time as an integer:**
  - $2^{24} + 2^{25} + 2^{26} + 2^{27} + 2^{28} + 2^{29} = 1056964608$ (this is i)
  - why do we need i?

https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c

FAST INVERSE SQUARE ROOT, Q III

```
552  float Q_rsqrt( float number )
553  {
554          long i;
555          float x2, y;
556          const float threehalfs = 1.5F;
557
558          x2 = number * 0.5F;
559          y  = number;
560          i  = * ( long * ) &y;                       // evil floating point bit level hacking
561          i  = 0x5f3759df - ( i >> 1 );               // what the duck?
562          y  = * ( float * ) &i;
563          y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564  //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

- **the first interesting instruction is on line 560**

- **if M contains the mantissa and E contains the exponent, then**
  - our FP number is $2^{23} \times E + M$
  - and the value of the number is $(1 + M / 2^{23}) \times 2^{(E - 127)}$
  - we observe that:

$$\begin{aligned}
\log_2\left(\left(1+\frac{M}{2^{23}}\right)\times 2^{E-127}\right) &= \log_2\left(1+\frac{M}{2^{23}}\right)+\log_2\left(2^{E-127}\right)\\
&= \log_2\left(1+\frac{M}{2^{23}}\right)+E-127\\
&\approx \frac{M}{2^{23}}+E-127+\mu \quad (\text{we used} \quad \log_2(1+x)\approx x, \mu \text{ is a correction})\\
&= \frac{1}{2^{23}}(2^{23}\times E+M)+\mu-127\\
&= \frac{1}{2^{23}}(\text{the bit representation})+\mu-127
\end{aligned}$$

https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c

FAST INVERSE SQUARE ROOT, Q III

```
552   float Q_rsqrt( float number )
553   {
554           long i;
555           float x2, y;
556           const float threehalfs = 1.5F;
557
558           x2 = number * 0.5F;
559           y  = number;
560           i  = * ( long * ) &y;                                   // evil floating point bit level hacking
561           i  = 0x5f3759df - ( i >> 1 );               // what the duck?
562           y  = * ( float * ) &i;
563           y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564   //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

- **the first interesting instruction is on line 560**
- **why do we compute log(y)? in fact, we want 1/sqrt(y). but:**

$$\log_2\left(\frac{1}{\sqrt{y}}\right) = \log_2\left(y^{-1/2}\right) = -\frac{1}{2}\log_2(y) = -(i \gg 1)$$

- **we are on line 561 now. where does that hexadecimal number come from?**

https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c

FAST INVERSE SQUARE ROOT, Q III

```
552   float Q_rsqrt( float number )
553   {
554           long i;
555           float x2, y;
556           const float threehalfs = 1.5F;
557
558           x2 = number * 0.5F;
559           y  = number;
560           i  = * ( long * ) &y;                       // evil floating point bit level hacking
561           i  = 0x5f3759df - ( i >> 1 );               // what the duck?
562           y  = * ( float * ) &i;
563           y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564   //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

- **we are on line 561 now. where does that hexadecimal number come from?**

$$Y = \frac{1}{\sqrt{y}} \text{ then } \quad \log_2(Y) = \log_2(\frac{1}{\sqrt{y}}) \text{ hence}$$

$$\frac{1}{2^{23}}(M_Y + 2^{23} \times E_Y) + \mu - 127 = -\frac{1}{2}\left(\frac{1}{2^{23}}(M_y + 2^{23} \times E_y) + \mu - 127\right)$$

$$M_Y + 2^{23} \times E_Y = \frac{3}{2} \times 2^{23}(127 - \mu) - \frac{1}{2}(M_y + 2^{23} \times E_y)$$

$$= 0\text{x5f3759df} - (i \gg 1) \quad (\mu \text{ is chosen to minimize the error})$$

- **then, on line 562, y is converted back to FP**

https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c

FAST INVERSE SQUARE ROOT, Q III

```
552  float Q_rsqrt( float number )
553  {
554          long i;
555          float x2, y;
556          const float threehalfs = 1.5F;
557
558          x2 = number * 0.5F;
559          y  = number;
560          i  = * ( long * ) &y;                                   // evil floating point bit level hacking
561          i  = 0x5f3759df - ( i >> 1 );               // what the duck?
562          y  = * ( float * ) &i;
563          y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564  //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

- **line 563 performs one iteration of Newton's method**
  - it improves the result (it is an optimization method)

https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c