Unit 1 - Computer Arithmetic

Fixed-Point (FX) Arithmetic

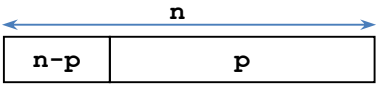
Integer Representation

- $n-bit$ number: $b_{n-1}b_{n-2}\dots b_0$. Special case of FX numbers with p=0.
- For a review on integer number representations (SM, 1C, 2C), refer to ECE4710 – Computer Arithmetic.

	UNSIGNED	**SIGNED**
Decimal Value	$D=\sum_{i=0}^{n-1} b_i 2^i$	$D=-2^{n-1}b_{n-1}+\sum_{i=0}^{n-2} b_i 2^i$
Range of values	$[0, 2^n-1]$	$[-2^{n-1}, 2^{n-1}-1]$

Fixed Point Representation

- Typical representation $[n\ p]$: $n-bit$ number with p fractional bits: $b_{n-p-1}b_{n-p-2}\dots b_0.b_{-1}b_{-2}\dots b_{-p}$
- MATLAB/Octave scripts for Fixed-Point to Decimal conversion, and for Decimal to Fixed-Point conversion: script_fx2dec_converter.zip: my_fxdec.m, my_dec2fx, my_bitcmp.m.



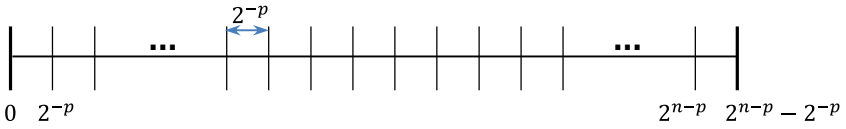
	UNSIGNED	**SIGNED**
Decimal Value	$D=\sum_{i=-p}^{n-p-1} b_i 2^i$	$D=-2^{n-p-1}b_{n-p-1}+\sum_{i=-p}^{n-p-2} b_i 2^i$
Range of values	$\left[\frac{0}{2^p}, \frac{2^n-1}{2^p}\right]=[0, 2^{n-p}-2^{-p}]$	$\left[\frac{-2^{n-1}}{2^p}, \frac{2^{n-1}-1}{2^p}\right]=[-2^{n-p-1}, 2^{n-p-1}-2^{-p}]$
Dynamic Range	$\frac{\lvert 2^{n-p}-2^{-p}\rvert}{\lvert 2^{-p}\rvert}=2^n-1$ $(dB)=20\times\log_{10}(2^n-1)$	$\frac{\lvert -2^{n-p-1}\rvert}{\lvert 2^{-p}\rvert}=2^{n-1}$ $(dB)=20\times\log_{10}(2^{n-1})$
Resolution (1 LSB)	2^{-p}	2^{-p}

- Dynamic Range:

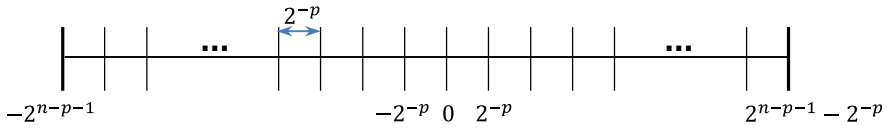
$$Dynamic\ Range=\frac{largest\ abs.value}{smallest\ nonzero\ abs.value}$$

$$Dynamic\ Range(dB)=20\times\log_{10}(Dynamic\ Range)$$

- Unsigned numbers: Range of Values



- Signed numbers: Range of Values

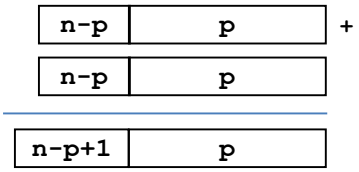


- Examples:

	FX Format	**Range**	**Dynamic Range (dB)**	**Resolution**
UNSIGNED	[8 7]	[0, 1.9922]	48.13	0.0078
	[12 8]	[0, 15.9961]	72.24	0.0039
	[16 10]	[0, 63.9990]	96.33	0.0010
SIGNED	[8 7]	[-1, 0.9921875]	42.14	0.0078
	[12 8]	[-8, 7.99609375]	66.23	0.0039
	[16 10]	[-64, 63.9990234375]	90.31	0.0010

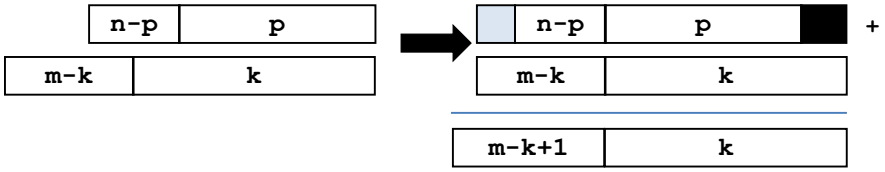
FIXED-POINT ADDITION/SUBTRACTION

- Addition of two numbers represented in the format $[n\ \ p]$:

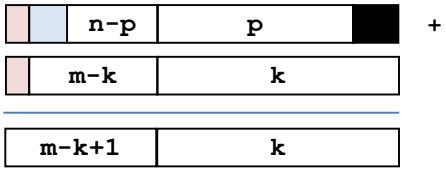


$A \times 2^{-p} \pm B \times 2^{-p} = (A \pm B) \times 2^{-p}$
We perform integer addition/subtraction of A and B. We just need to interpret the result differently by placing the fractional point where it belongs. Notice that the hardware is the same as that of integer addition/subtraction.

When adding/subtracting numbers with different formats $[n\ \ p]$ and $[m\ \ k]$, we first need to align the fractional point so that we use a format for both numbers: it could be $[n\ \ p]$, $[m\ \ k]$, $[n-p+k\ \ k]$, $[m-k+p\ \ p]$. This is done by zero-padding and sign-extending where necessary. In the figure below, the format selected for both numbers is $[m\ \ k]$, while the result is in the format $[m+1\ \ k]$.



Important: The result of the addition/subtraction requires an extra bit in the worst-case scenario. In order to correctly compute it in fixed-point arithmetic, we need to sign-extend (by one bit) the operators prior to addition/subtraction.

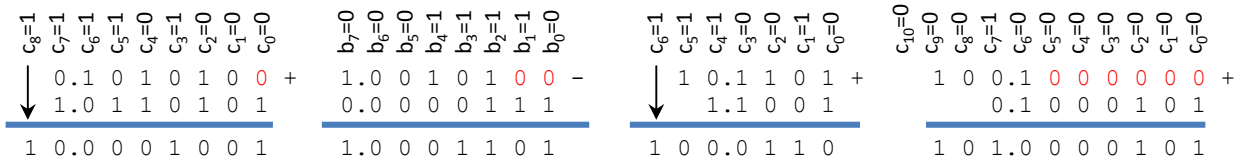


Multi-operand Addition: N numbers of format $[n\ \ p]$: The total number of bits is given by : $n + \lceil \log_2 N \rceil$ (this can be demonstrated by an adder tree). Notice that the number of fractional bits does not change (it remains p), only the integer bits increase by $\lceil \log_2 N \rceil$, i.e., the number of integer bits become $n - p + \lceil \log_2 N \rceil$.

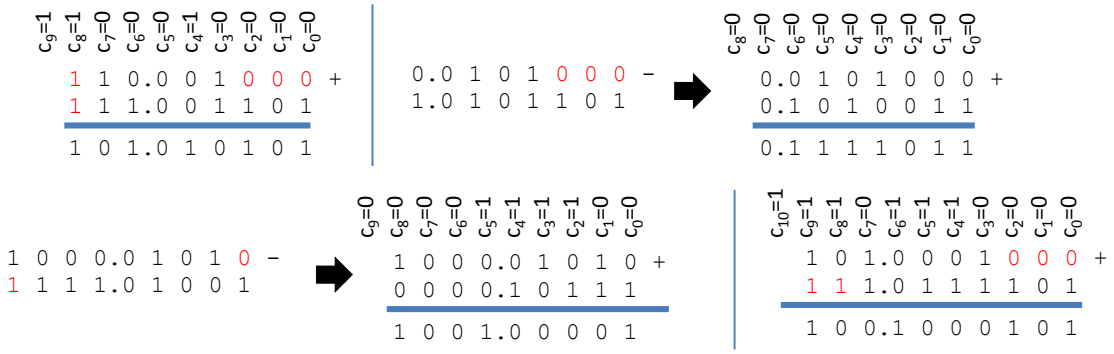
- **Examples**: Calculate the result of the additions and subtractions for the following fixed-point numbers.

UNSIGNED		SIGNED	
0.101010 + 1.0110101	1.00101 − 0.0000111	10.001 + 1.001101	0.0101 − 1.0101101
10.1101 + 1.1001	100.1 + 0.1000101	1000.0101 − 111.01001	101.0001 + 1.0111101

Unsigned:

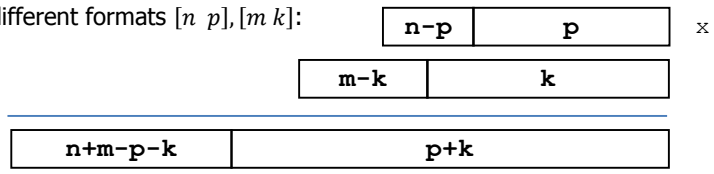


Signed:



FIXED-POINT MULTIPLICATION

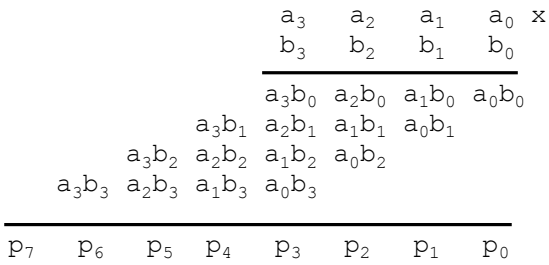
- **Unsigned multiplication**
  Multiplication of two signed numbers represented with different formats $[n\ \ p], [m\ k]$:



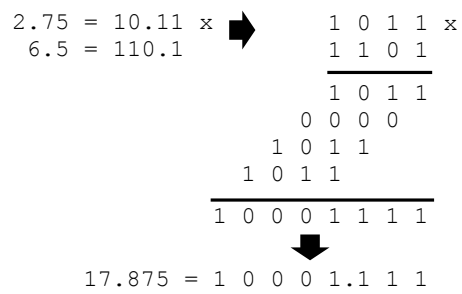
$(A \times 2^{-p}) \times (B \times 2^{-k}) = (A \times B) \times 2^{-p-k}$. We can perform integer multiplication of A and B and then place the fractional point where it belongs. The format of the multiplication result is given by $[n+m\ \ p+k]$. There is no need to align the fractional point of the input quantities.

Special case: $m = n, k = p$
$(A \times 2^{-p}) \times (B \times 2^{-p}) = (A \times B) \times 2^{-2p}$. Here, the format of the multiplication result is given by $[2n\ \ 2p]$.

✓ Multiplication procedure for unsigned integer numbers:

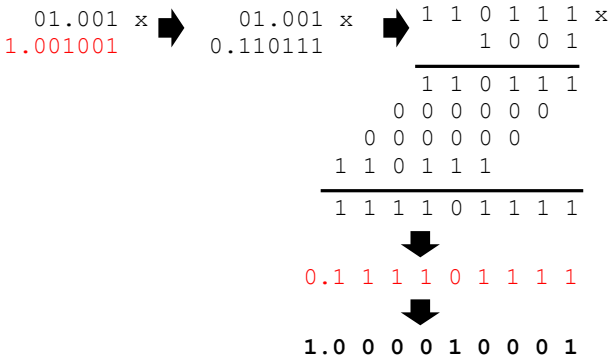


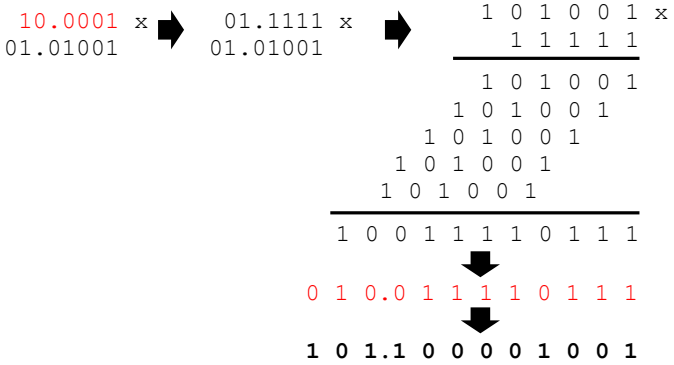
Example: when multiplying, we treat the numbers as integers. Only when we get the result, we place the fractional point where it belongs.

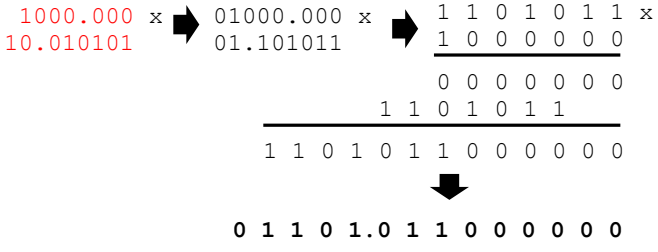


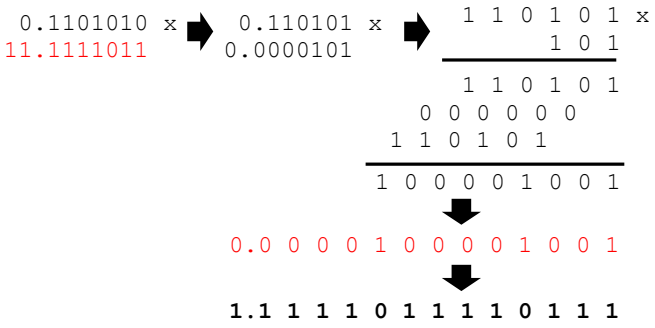
- **Signed Multiplication:** We first take the absolute value of the operands. Then, if at least one of the operands was negative, we need to change the sign of the result. We then place the fractional point where it belongs.

Examples:









FIXED-POINT DIVISION

- **Unsigned integer division:**
  The division of two unsigned integer numbers $^{A}/_{B}$ (where A is the dividend and B the divisor), results in a quotient Q and a remainder R, where $A = B \times Q + R$. Most divider architectures output Q and R.

```
        15  ← Q                    00001111  ← Q
B → 9  ) 140 ← A          B → 1001 ) 10001100 ← A
         90                          1001
        ---                         -----
         50                          10001
         45                          1001
        ---                         -----
          5 ← R                     10000
                                     1001
                                    -----
                                     1110
                                     1001
                                    -----
                                      101 ← R
```

```
ALGORITHM
R = 0
for i = n-1 downto 0
    left shift R (input = a_i)
    if R ≥ B
        q_i = 1, R ← R-B
    else
        q_i = 0
    end
end
```

- For n-bits dividend (A) and m-bits divisor (B):
  - ✓ The largest value for Q is $2^n - 1$ (by using $B = 1$). The smallest value for Q is 0. So, we use n bits for Q.
  - ✓ The remainder R is a value between 0 and $B - 1$. Thus, at most we use m bits for R.
  - ✓ If $A = 0, B \neq 0$, then $Q = R = 0$.
  - ✓ If $B = 0$, we have a division by zero. The result is undetermined.
- In computer arithmetic, integer division usually means getting $Q = \lfloor A/B \rfloor$.

- **Unsigned FX Division:** A_f/B_f
  We first need to align the numbers so that they have the same number of fractional bits, then divide them treating them as integers. The quotient will be integer, while the remainder will have the same number of fractional bits as A_f.
  - ✓ A_f is in the format $[na\ a]$. B_f is in the format $[nb\ b]$. We work with $a \geq b$. If $a < b$, just append 0's to A_f so that $a = b$.

  Step 1: For $a \geq b$, we align the fractional points and then get the integer numbers A and B, which result from:
  $$A = A_f \times 2^a \quad B = B_f \times 2^a$$
  Step 2: Integer division: $\frac{A}{B} = \frac{A_f}{B_f}$
  The numbers A and B are related by the formula: $A = B \times Q + R$, where Q and R are the quotient and remainder of the integer division of A and B. Note that Q is also the quotient of $\frac{A_f}{B_f}$.
  Step 3: To get the correct remainder of $\frac{A_f}{B_f}$, we re-write the previous equation:
  $$A_f \times 2^a = (B_f \times 2^a) \times Q + R \rightarrow A_f = B_f \times Q + (R \times 2^{-a})$$
  Then: $Q_f = Q,\ R_f = R \times 2^{-a}$

  Example: $\frac{1010{,}011}{11{,}1}$

Step 1: Alignment, $a = 3$	$\frac{1010{,}011}{11{,}1} = \frac{1010{,}011}{11{,}100} = \frac{1010011}{11100}$
Step 2: Integer Division	$\frac{1010011}{11100} \Rightarrow 1010011 = 11100(10) + 11011 \rightarrow\ Q = 10, R = 11011$
Step 3: Get actual remainder: $R \times 2^{-a}$	$R_f = 11{,}011$
Verification:	$1010{,}011 = 11{,}1(10) + 11{,}011,\ Q_f = 10, R_f = 11{,}011$

  - ✓ **Adding precision to Qf (quotient of A_f/B_f):**
    The previous procedure only gets Q as an integer. What if we want to get the division result with x number of fractional bits? To do so, after alignment, we append x zeros to $A_f \times 2^a$ and perform integer division.
    $$A = A_f \times 2^a \times 2^x \quad B = B_f \times 2^a$$
    $$A_f \times 2^{a+x} = (B_f \times 2^a) \times Q + R \rightarrow A_f = B_f \times (Q \times 2^{-x}) + (R \times 2^{-a-x})$$
    Then: $Q_f = Q \times 2^{-x},\ R_f = R \times 2^{-a-x}$

    Example: $\frac{1010{,}011}{11{,}1}$ with $x = 2$ bits of precision

Step 1: Alignment, $a = 3$	$\frac{1010{,}011}{11{,}1} = \frac{1010{,}011}{11{,}100} = \frac{1010011}{11100}$
Step 2: Append $x = 2$ zeros	$\frac{1010011}{11100} = \frac{101001\mathbf{100}}{11100}$

Step 3: Integer Division	$\frac{101001\mathbf{00}}{11100} \Rightarrow 101001\mathbf{00} = 11100(1011) + 11000, Q = 1011, R = 11000$
Step 4: Get actual remainder and quotient: $Q_f = Q \times 2^{-x}, R_f = R \times 2^{-a-x}$	$Q_f = 10{,}11, R_f = 0{,}11000$
Verification:	$1010{,}01100 = 11{,}1(10{,}11) + 0{,}11000$

- **Signed FX Division**: In this case (as in the multiplication), we first take the absolute value of the operators A and B. If only one of the operators is negative, the result of $|A|/|B|$ requires a sign change.
  Note that once the correct quotient Q_f with fractional bits is available, getting R_f with the correct sign is not very useful.

  Examples: Get the division result (with $x = 4$ fractional bits) for the following signed fixed-point numbers:

  ✓ $\frac{101.1001}{1.011}$: To positive (numerator and denominator), alignment, and then to unsigned: $a = 4$: $\frac{101.1001}{1.011} = \frac{010.0111}{0.1010} \equiv \frac{100111}{1010}$

```
          0000111110
      ┌──────────
1010  │ 1001110000
        1010
        ────
         10011
          1010
          ────
          10010
           1010
           ────
           10000
            1010
            ────
             1100
             1010
             ────
              100
```

  Append $x = 4$ zeros: $\frac{100111\mathbf{0000}}{1010}$
  Unsigned integer Division:

  $Q = 111110, R = 100$
  $\rightarrow Qf = 11.1110\ (x = 4)$

  Final result (2C): $\frac{101.1001}{1.011} = 011.111$ (this is represented as a signed number)

  ✓ $\frac{11.011}{1.01011}$: To positive (numerator and denominator), alignment, and then to unsigned, $a = 5$: $\frac{00.101}{0.10101} = \frac{0.10100}{0.10101} \equiv \frac{10100}{10101}$

```
          000001111
      ┌──────────
10101 │ 1010000000
        10101
        ─────
         100110
          10101
          ─────
          100010
           10101
           ─────
            11010
            10101
            ─────
              101
```

  Append $x = 4$ zeros: $\frac{10100\mathbf{0000}}{10101}$
  Unsigned integer Division:

  $Q = 1111, R = 101$
  $\rightarrow Qf = 0.1111(x = 4)$

  Final result (2C): $\frac{11.011}{1.01011} = 0.1111$ (this is represented as a signed number)

  ✓ $\frac{10.0110}{01.01}$: To positive (numerator), alignment, and then to unsigned, $a = 4$: $\frac{01.1010}{01.01} = \frac{01.1010}{01.0100} \equiv \frac{11010}{10100}$

```
          000010100
      ┌──────────
10100 │ 1101000000
        10100
        ─────
         11000
         10100
         ─────
         10000
```

  Append $x = 4$ zeros: $\frac{11010\mathbf{0000}}{10100}$
  Unsigned integer Division:

  $Q = 10100, R = 10000$
  $\rightarrow Qf = 1.0100(x = 4)$ ✶ Qf here is represented as an unsigned number

  Final result (2C): $\frac{10.0110}{01.01} = 2C(01.01) = 10.11$

  ✓ $\frac{0.101010}{110.1001}$: To positive (denominator), alignment, and then to unsigned, $a = 5$: $\frac{0.10101}{001.0111} = \frac{0.10101}{001.01110} \equiv \frac{10101}{101110}$

```
          000000111
      ┌──────────
101110│ 1010100000
        101110
        ──────
         1001100
          101110
          ──────
          111100
          101110
          ──────
            1110
```

  Append $x = 4$ zeros: $\frac{10101\mathbf{0000}}{101110}$
  Unsigned integer Division:

  $Q = 111, R = 1110$
  $\rightarrow Qf = 0.0111(x = 4)$

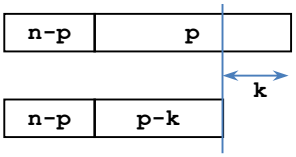
  Final result (2C): $\frac{0.101010}{110.1001} = 2C(0.0111) = 1.1001$

ARITHMETIC FX UNITS. TRUNCATION/ROUNDING/SATURATION

ARITHMETIC FX UNITS

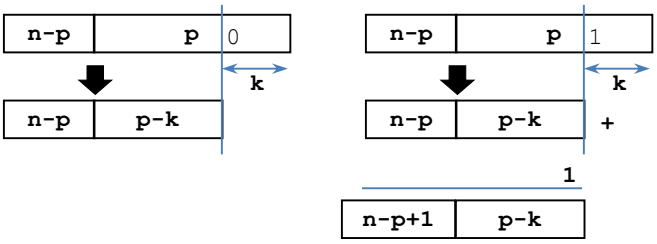
- They are essentially the same as those integer arithmetic units. The main difference is that we need to know where to place the fractional point. The design must keep track of the FX format at every point of the architecture. For example, in FX division, we need first to perform alignment and append x zeros for a desired precision; this incurs in slightly extra hardware.
- One benefit of FX representation is that we can perform truncation, rounding and saturation on the input, intermediate, and output values. These operations usually require extra hardware resources.

TRUNCATION



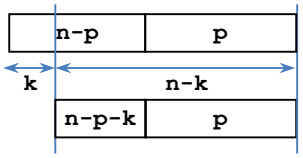
- This is a useful operation when less hardware is required in subsequent operations. However this comes at the expense of less accuracy.
- To assess the effect of truncation, use PSNR (dB) or MSE with respect to a double floating point result or with respect to the original $[n\ \ p]$ format.
- Truncation is usually meant to be truncation of the fractional part. However, we can also truncate the integer part (chop off k MSBs). This is not recommended as it might render the number unusable.

ROUNDING



- This operation allows for hardware savings in subsequent operations at the expense of reduced accuracy. But it is more accurate than simple truncation. However, it requires extra hardware to deal with the rounding operation.
- For the number $b_{n-p-1}b_{n-p-2}\dots b_0.b_{-1}b_{-2}\dots b_{-p}$, if we want to chop k bits (LSB portion), we use the b_{k-p-1} bit to determine whether to round. If the $b_{k-p-1}=0$, we just truncate. If $b_{k-p-1}=1$, we need to add '1' to the LSB of the truncated result.

SATURATION



- This is helpful when we need to restrict the number of integer bits. Here, we are asked to reduce the number of integer bits by k. Simple truncation chops off the integer part by k bits; this might completely modify the number and render it totally unusable. Instead, in saturation, we apply the following rules:
  - ✓ If all the $k+1$ MSBs of the initial number are identical, that means that chopping by k bits does not change the number at all, so we just discard the k MSBs.
  - ✓ If the $k+1$ MSBs are not identical, chopping by k bits does change the number. Thus, here, if the MSB of the initial number is 1, the resulting $(n-k)$-bit number will be $-2^{n-k-p-1}=10\dots0$ (largest negative number). If the MSB is 0, the resulting $(n-k)$-bit number will be $2^{n-k-p-1}-2^{-p}=011\dots1$ (largest positive number).

Examples: Represent the following signed FX numbers in the signed fixed-point format: $[8\ 7]$. You can use rounding or truncation for the fractional part. For the integer part, use saturation.

- $1{,}01101111$:
  To represent this number in the format $[8\ 7]$, we keep the integer bit, and we can only truncate or round the last LSB:
  After truncation: $1{,}0110111$
  After rounding: $1{,}0110111+1=1{,}0111000$

- $11{,}111010011$:
  Here, we need to get rid of on MSB and two LSBs. Let's use rounding (to the next bit).
  Saturation in this case amounts to truncation of the MSB, as the number won't change if we remove the MSB.
  After rounding: $11{,}1110100+1=11{,}1110101$
  After saturation: $1{,}1110101$

- $101{,}111010011$:
  Here, we need to get rid of two MSB and two LSBs.
  Saturation: Since the three MSBs are not the same and the MSB=1 we need to replace the number by the largest negative number (in absolute terms) in the format $[8\ 7]$: $1{,}0000000$

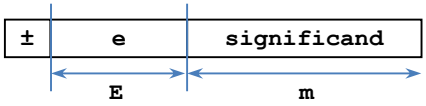
- $011{,}1111011011$:
  Here, we need to get rid of two MSB and three LSBs.
  Saturation: Since the three MSBs are not identical and the MSB=0, we need to replace the number by the largest positive number in the format $[8\ 7]$: $0{,}1111111$

FLOATING-POINT (FP) ARITHMETIC

FLOATING POINT REPRESENTATION

- There are many ways to represent floating numbers. A common way is:

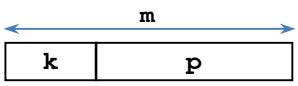
$X = \pm significand \times 2^{e}$



- Exponent e: Signed integer. It is common to encode this field using a bias: $e + bias$. This facilitates zero detection ($e + bias = 0$). Note that the exponent of the actual number is always e regardless of the bias (the bias is just for encoding).

$$e \in [-2^{E-1}, 2^{E-1} - 1\,]$$

- Significand: Unsigned fixed-point number. Usually normalized to a particular format, e.g.: $[0,1), [1,2)$.



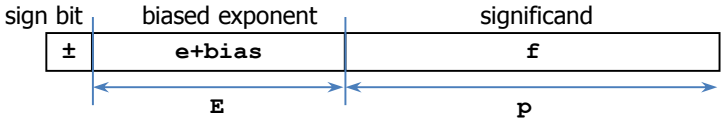
Format (unsigned): $[m\ \ p]$. Range: $\left[0, \frac{2^m-1}{2^p}\right] = [0, 2^{m-p} - 2^{-p}]$, $k = m - p$
If $k = 0 \rightarrow$ Significand $\in [0,1 - 2^{-p}] = [0,1)$
If $k = m \rightarrow$ Significand $\in [0, 2^m - 1]$. Integer significand.

Another common representation of the significand is using $k = 1$ and setting that bit (the MSB) to 1. Here, the range of the significand would be $[0, 2^1 - 2^{-p}]$, but since the integer bit is 1, the values start from 1, which result in the following significand range: $[1, 2^1 - 2^{-p}]$. This is a popular normalization, as it allows us to drop the MSB in the encoding.

IEEE-754 STANDARD

- The representation is as follows:

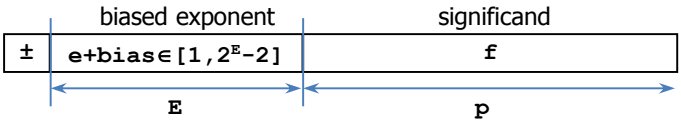
$$X = \pm 1.f \times 2^{e}$$



- **Significand**: Unsigned FX number. The representation is normalized to $s = 1.f$, where f is the mantissa. There is always an integer bit 1 (called hidden 1) in the representation of the significand, so we do not need to indicate in the encoding. Thus, we only use f the mantissa in the significant field.
  Significand range: $[1,2 - 2^{-p}] = [1,2)$ Significand format (unsigned FX): $[p + 1\ p]$

- **Biased exponent**: Unsigned integer with E bits. $bias = 2^{E-1} - 1$. Thus, $exp = e + bias \rightarrow e = exp - bias$. We just subtract the $bias$ from the exponent field in order to get the exponent value e.
  - ✓ $exp = e + bias \in [0, 2^E - 1\,]$. exp is represented as an unsigned integer number with E bits. The bias makes sure that $exp \geq 0$. Also note that $e \in [-2^{E-1} + 1, 2^{E-1}\,]$.
  - ✓ The IEEE-754 standard reserves the following cases: i) $exp = 2^E - 1$ ($e = 2^{E-1}$) to represent special numbers (NaN and $\pm\infty$), and ii) $exp = 0$ to represent the zero and the denormalized numbers. The remaining cases are called ordinary numbers.

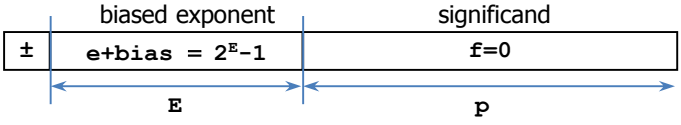
- **Ordinary numbers**:



Range of e: $[-2^{E-1} + 2, 2^{E-1} - 1]$.
Max number: $largest\ significand \times 2^{largest\ exponent}$
$$max = 1.11\ldots1 \times 2^{2^{E-1}-1} = (2 - 2^{-p}) \times 2^{2^{E-1}-1}$$
Min. number: $smallest\ significand \times 2^{smallest\ exponent}$
$$min = 1.00\ldots0 \times 2^{-2^{E-1}+2} = 2^{-2^{E-1}+2}$$

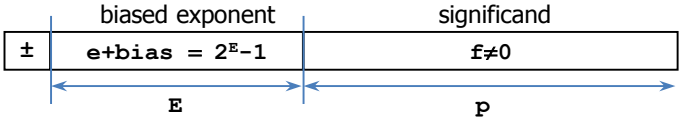
$$Dynamic\ Range = \frac{max}{min} = \frac{(2 - 2^{-p}) \times 2^{2^{E-1}-1}}{2^{-2^{E-1}+2}} = (2 - 2^{-p}) \times 2^{2^E-3}$$
$$Dynamic\ Range\ (dB) = 20 \times \log_{10}\{(2 - 2^{-p}) \times 2^{2^E-3}\}$$

- **Plus/minus Infinite:** $\pm\infty$



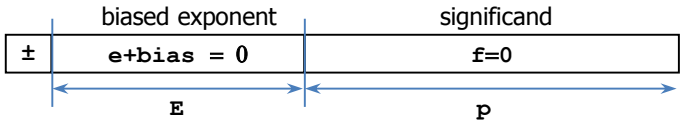
The exp field is a string of 1's. This is a special case where $exp = 2^E - 1$. ($e = 2^{E-1}$)
$$\pm\infty = \pm 2^{2^{E-1}}$$

- **Not a Number:** NaN



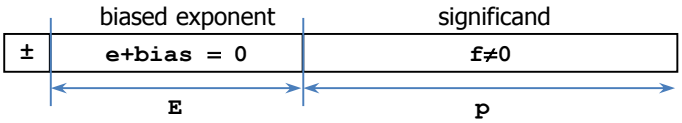
The exp field is a strings of 1's. $exp = 2^E - 1$. This is a special case where $exp = 2^E - 1$ ($e = 2^{E-1}$). The only difference with $\pm\infty$ is that f is a nonzero number.

- **Zero**:



Zero cannot be represented with a normalized significand $s = 1.00\ldots0$ since $X = \pm 1.f \times 2^e$ cannot be zero. Thus, a special code must be assigned to it, where $s = 0.00\ldots0$ and $exp = 0$. Every single bit (except for the sign) is zero. There are two representations for zero. The number zero is a special case of the denormalized numbers, where $s = 0.f$ (see below).

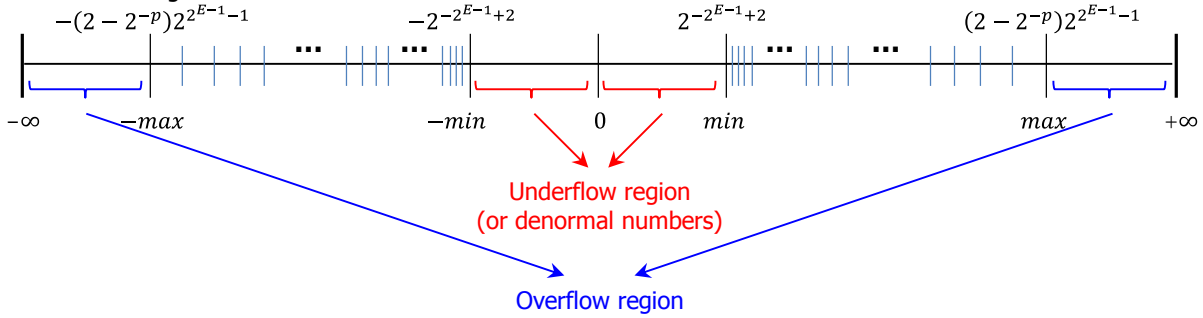
- **Denormalized numbers**: The implementation of these numbers is optional in the standard (except for the zero). Certain small values that are not representable as normalized numbers (and are rounded to zero), can be represented more precisely with denormals. This is a "graceful underflow" provision, which leads to hardware overhead.



These numbers have the exp field equal to zero. The tricky part is that e is set to $-2^{E-1}+2$ (not $-2^{E-1}+1$, as the $e + bias$ formula states). The significand is represented as $s = 0.f$. Thus, the floating point number is $X = \pm 0.f \times 2^{-2^{E-1}+2}$. These numbers can represent numbers lower (in absolute value) than min (the number zero is a special case).

Why is e not $-2^{E-1}+1$? Note that the smallest ordinary number is $2^{-2^{E-1}+2}$.
The largest denormalized number with $e = -2^{E-1}+1$ is: $0.11\ldots1 \times 2^{2^{E-1}-1} = (1-2^{-p}) \times 2^{-2^{E-1}+1}$.
The largest denormalized number with $e = -2^{E-1}+2$ is: $0.11\ldots1 \times 2^{2^{E-1}-2} = (1-2^{-p}) \times 2^{-2^{E-1}+2}$.
By picking $e = -2^{E-1}+2$, the gap between the largest denormalized number and the smallest ordinary number is smaller. Though this specification makes the formula $e + bias = 0$ inconsistent, it helps to improve accuracy.

- Depiction of the range of values:



- The IEEE-754-2008 (revision of IEEE-754-1985) standard defines several representations: half (16 bits, E=5, p=10), single (32 bits, E = 8, p = 23) and double (64 bits, E = 11, p = 52). There is also quadruple precision (128 bits) and octuple precision (256 bits). You can define your own representation by selecting a particular number of bits for the exponent and significand. The table lists various parameters for half, single and double FP arithmetic (ordinary numbers):

	Ordinary numbers		Exponent bits (E)	Range of e	Bias	Dynamic Range (dB)	Significand range	Significand bits (p)
	Min	Max						
Half	2^{-14}	$(2-2^{-10})2^{+15}$	5	$[-14,15]$	15	180.61 dB	$[1,2-2^{-10}]$	10
Single	2^{-126}	$(2-2^{-23})2^{+127}$	8	$[-126,127]$	127	1529 dB	$[1,2-2^{-23}]$	23
Double	2^{-1022}	$(2-2^{-52})2^{+1023}$	11	$[-1022,1023]$	1023	12318 dB	$[1,2-2^{-52}]$	52

- Rules for arithmetic operations:
  - ✓ $Ordinary\ number \div (+\infty) = \pm 0$
  - ✓ $Ordinary\ number \div (0) = \pm\infty$
  - ✓ $(+\infty) \times Ordinary\ number = \pm\infty$
  - ✓ $NaN + Ordinary\ number = NaN$
  - ✓ $(0) \div (0) = NaN$
  - ✓ $(0) \times (\pm\infty) = NaN$
  - $(\pm\infty) \div (\pm\infty) = NaN$
  - $(\infty) + (-\infty) = NaN$

Examples:

- `F43DE962` (single): `1111 0100 0011 1101 1110 1001 0110 0010`
  $e + bias = 1110\ 1000 = 232 \rightarrow e = 232 - 127 = 105$
  $Significand = 1.011\ 1101\ 1110\ 1001\ 0110\ 0010 = 1.4837$
  $X = -1.4837 \times 2^{105} = -6.1085 \times 10^{31}$

- `007FADE5` (single): `0000 0000 0111 1111 1010 1101 1110 0101`
  $e + bias = 0000\ 0000 = 0 \rightarrow Denormal\ number \rightarrow e = -126$
  $Significand = 0.111\ 1111\ 1010\ 1101\ 1110\ 0101 = 0.9975$
  $X = 0.9975 \times 2^{-126} = 1.1725 \times 10^{-38}$

ADDITION/SUBTRACTION

$b_1 = \pm s_1 2^{e_1}$, $s_1 = 1.f_1$ $\quad\quad$ $b_2 = \pm s_2 2^{e_2}$, $s_1 = 1.f_2$

$\rightarrow b_1 + b_2 = \pm s_1 2^{e_1} \pm s_2 2^{e_2}$

If $e_1 \geq e_2$, we simply shift s_2 to the right by $e_1 - e_2$ bits. This step is referred to as alignment shift.

$s_2 2^{e_2} = \frac{s_2}{2^{e_1 - e_2}} 2^{e_1}$

$\rightarrow b_1 + b_2 = \pm s_1 2^{e_1} \pm \frac{s_2}{2^{e_1 - e_2}} 2^{e_1} = \left(\pm s_1 \pm \frac{s_2}{2^{e_1 - e_2}}\right) \times 2^{e_1} = s \times 2^e$

$\rightarrow b_1 - b_2 = \pm s_1 2^{e_1} \mp \frac{s_2}{2^{e_1 - e_2}} 2^{e_1} = \left(\pm s_1 \mp \frac{s_2}{2^{e_1 - e_2}}\right) \times 2^{e_1} = s \times 2^e$

- **Normalization**: Once the operators are aligned, we can add. The result might not be in the format $1.f$, so we need to discard the leading 0's of the result and stop when a leading 1 is found. Then, we must adjust e_1 properly, this results in e.
  - ✓ For example, for addition, when the two operands have similar signs, the resulting significand is in the range $[1,4)$, thus a single bit right shift is needed on the significant to compensate. Then, we adjust e_1 by adding 1 to it (or by left shifting everything by 1 bit). When the two operands have different signs, the resulting significand might be lower than 1 (e.g.: 0.000001) and we need to first discard the leading zeros and then right shift until we get $1.f$. We then adjust e_1 by adding the same number as the number of shifts to the right on the significand.

Note that overflow/underflow can occur during the addition step as well as due to normalization.

Example: $s_3 = \left(\pm s_1 \pm \frac{s_2}{2^{e_1 - e_2}}\right) = 00011.1010$

First, discard the leading zeros: $s_3 = 11.1010$
Normalization: right shift 1 bit: $s = s_3 \times 2^{-1} = 1.11010$
Now that we have the normalized significand s, we need to adjust the exponent e_1 by adding 1 to it: $e = e_1 + 1$:

$$(s_3 \times 2^{-1}) \times 2^{e_1 + 1} = s \times 2^e = 1.1101 \times 2^{e_1 + 1}$$

Example: $b_1 = 1.0101 \times 2^5$, $b_2 = -1.1110 \times 2^3$

$$b = b_1 + b_2 = 1.0101 \times 2^5 - \frac{1.1110}{2^2} \times 2^5 = (1.0101 - 0.011110) \times 2^5$$

$1.0101 - 0.011110 = 0.11011$. To get this result, we convert the operands to the 2C representation (you can also do unsigned subtraction if the result is positive). Here, the result is positive. Finally, we perform normalization:

$$\rightarrow b = b_1 + b_2 = (0.11011) \times 2^5 = (0.11011 \times 2^1) \times 2^5 \times 2^{-1} = 1.1011 \times 2^4$$

Example (Subtraction): $b_1 = 1.0101 \times 2^5$, $b_2 = 1.111 \times 2^5$

$$b = b_1 - b_2 = 1.0101 \times 2^5 - 1.111 \times 2^5 = (1.0101 - 1.111) \times 2^5$$

To subtract, we convert to 2C representation: $R = 01.0101 - 01.1110 = 01.0101 + 10.0010 = 11.0111$. Here, the result is negative. So, we get the absolute value ($|R| = 2C(1.0111) = 0.1001$) and place the negative sign on the final result:

$$\rightarrow b = b_1 - b_2 = -(0.1001) \times 2^5$$

Examples:

✓ X = 50DAD000 - D0FAD000:

50DAD000: 0101 0000 1101 1010 1101 0000 0000 0000

$e + bias = 10100001 = 161 \rightarrow e = 161 - 127 = 34$ $\quad\quad$ $Significand = 1.10110101101$

50DAD000 = $1.10110101101 \times 2^{34}$

D0FAD000: 1101 0000 1111 1010 1101 0000 0000 0000

$e + bias = 10100001 = 161 \rightarrow e = 161 - 127 = 34$ $\quad\quad$ $Significand = 1.11110101101$

D0FAD000 = $-1.11110101101 \times 2^{34}$

$X = 1.10110101101 \times 2^{34} + 1.11110101101 \times 2^{34}$ (unsigned addition)

$X = 11.1010101101 \times 2^{34} = 1.11010101101 \times 2^{35}$
$e + bias = 35 + 127 = 162 = 10100010$
X = 0101 0001 0110 1010 1101 0000 0000 0000 = 516AD000

c_{12}=1 c_{11}=1 c_{10}=1 c_9=1 c_8=1 c_7=0 c_6=1 c_5=0 c_4=1 c_3=1 c_2=0 c_1=1 c_0=0

1.1 0 1 1 0 1 0 1 1 0 1 +
1.1 1 1 1 0 1 0 1 1 0 1

1 1.1 0 1 0 1 0 1 1 0 1 0

✓ $X =$ 60A10000 + C2F97000:
60A10000: 0110 0000 1010 0001 0000 0000 0000 0000
$e + bias = 11000001 = 193 \rightarrow e = 193 - 127 = 66$ $Significand = 1.0100001$
60A10000 = 1.0100001×2^{66}

C2F97000: 1100 0010 1111 1001 0111 0000 0000 0000
$e + bias = 10000101 = 133 \rightarrow e = 133 - 127 = 6$ $Significand = 1.11110010111$
C2F97000 = $-1.11110010111 \times 2^{6}$

$X = 1.0100001 \times 2^{66} - 1.11110010111 \times 2^{6}$
$X = 1.0100001 \times 2^{66} - \frac{1.11110010111}{2^{60}} \times 2^{66}$
Representing the division by 2^{60} requires more than $p + 1 = 24$ bits. Thus, we can approximate the 2nd operand with 0.

$X = 1.0100001 \times 2^{66}$
X = 0110 0000 1010 0001 0000 0000 0000 0000 = 60A10000

MULTIPLICATION

$b_1 = \pm s_1 2^{e_1}$, $b_2 = \pm s_2 2^{e_2}$

$\rightarrow b_1 \times b_2 = (\pm s_1 2^{e_1}) \times (\pm s_2 2^{e_2}) = \pm(s_1 \times s_2) 2^{e_1 + e_2}$

Note that $s = (s_1 \times s_2) \in [1,4)$. This means that the result might require normalization.

Example: $b_1 = 1.100 \times 2^{2}$, $b_2 = -1.011 \times 2^{4}$,
$\rightarrow b = b_1 \times b_2 = -(1.100 \times 1.011) \times 2^{6} = -(10{,}0001) \times 2^{6}$,

Normalization of the result: $b = -(10{,}0001 \times 2^{-1}) \times 2^{7} = -(1{,}00001) \times 2^{7}$.

Note: If the multiplication requires more bits than allowed by the representation (32, 64 bits), we have to do truncation or rounding. It is also possible that overflow/underflow might occur due to large/small exponents and/or multiplication of large/small numbers.

Examples:

✓ $X =$ 7A09D300 × 0BEEF000:
7A09D300: 0111 1010 0000 1001 1101 0011 0000 0000
$e + bias = 11110100 = 244 \rightarrow e = 244 - 127 = 117$ $Significand = 1.0001001110100110000$
7A09D300 = $1.000100111010011 \times 2^{117}$

0BEEF000: 0000 1011 1110 1110 1111 0000 0000 0000
$e + bias = 00010111 = 23 \rightarrow e = 23 - 127 = -104$ $Significand = 1.1101110111100000000$
0BEEF000 = $1.11011101111 \times 2^{-104}$

$X = 1.000100111010011 \times 2^{117} \times 1.11011101111 \times 2^{-104}$
$X = 10.000000101000110101111111101 \times 2^{13} = 1.0000000101000110101111111101 \times 2^{14} = 1.6466 \times 10^{4}$
$e + bias = 14 + 127 = 141 = 10001101$

X = 0100 0110 1000 0000 1010 0011 0101 1111 = 4680A35F (four bits were truncated)

✓ $X =$ 0B09A000 × 8FACC000:
0B092000: 0000 1011 0000 1001 1010 0000 0000 0000
$e + bias = 00010110 = 22 \rightarrow e = 22 - 127 = -105$ $Significand = 1.0001001101$
0B092000 = $1.0001001101 \times 2^{-105}$

8FACC000: 1000 1111 1010 1100 1100 0000 0000 0000
$e + bias = 00011111 = 31 \rightarrow e = 31 - 127 = -96$ $Significand = 1.010110011$
0FACE000 = $1.010110011 \times 2^{-96}$

$X = 1.0001001101 \times 2^{-105} \times -1.010110011 \times 2^{-96} = -1.01110011011111010111 \times 2^{-201} = -0 \times 2^{-126}$
$e + bias = -201 + 127 = -74 < 0$

Here, there is underflow (not even denormalized numbers different than zero can represent it). Then $X \leftarrow -0$.
X = 1000 0000 0000 0000 0000 0000 0000 0000 = 80000000

DIVISION

$b_1 = \pm s_1 2^{e_1}$, $b_2 = \pm s_2 2^{e_2}$

$$\rightarrow \frac{b_1}{b_2} = \frac{\pm s_1 2^{e_1}}{\pm s_2 2^{e_2}} = \pm \frac{s_1}{s_2} 2^{e_1 - e_2}$$

Note that $s = \left(\frac{s_1}{s_2}\right) \in (1/2, 2)$. This means that the result might require normalization.

Example:

$b_1 = 1.100 \times 2^2$, $b_2 = -1.011 \times 2^4$

$$\rightarrow \frac{b_1}{b_2} = \frac{1.100 \times 2^2}{-1.011 \times 2^4} = -\frac{1.100}{1.011} 2^{-2}$$

$\frac{1.100}{1.011}$: unsigned division, here we can include as many fractional bits as we want.

With $x = 4$ (and $a = 0$) we have:

$$\frac{1100\mathbf{0000}}{1011} \Rightarrow 1100\mathbf{0000} = 10101(1011) + 11$$
$$Q_f = 1{,}0101, R_f = 00{,}0011$$

If the result is not normalized, we need to normalized it. In this example, we do not need to do this.

$$\rightarrow \frac{b_1}{b_2} = \frac{1.100 \times 2^2}{-1.011 \times 2^4} = -1.0101 \times 2^{-2}$$

Example:

✓ X = 49742000 ÷ 40490000:

49742000: 0100 1001 0111 0100 0010 0000 0000 0000

$e + bias = 10010010 = 146 \rightarrow e = 146 - 127 = 19$ $\quad Significand = 1.11101000010000000000$

497420000 = $1.1110100001 \times 2^{19}$

40490000: 0100 0000 0100 1001 0000 0000 0000 0000

$e + bias = 10000000 = 128 \rightarrow e = 128 - 127 = 1$ $\quad Significand = 1.10010010000000000000$

0BEEF000 = 1.1001001×2^1

$$X = \frac{1.1110100001 \times 2^{19}}{1.1001001 \times 2^1}$$

```
                  00000000001001101 10
11001001000 ) 11110100001000000000
              11001001000
               1010110010 00
                110010010 00
                 100100000 000
                  11001001000
                   1010111000 00
                    11001001000
                     1001001100 00
                      11001001000
                       101110100 00
```

Alignment:

$$\frac{1.1110100001}{1.1001001} = \frac{1.1110100001}{1.1001001000} = \frac{11110100001}{11001001000}$$

Append $x = 8$ zeros: $\frac{111101000010\mathbf{0000000}}{11001001000}$

Integer division

$Q = 100110110, R = 1011101000 \rightarrow Qf = 1.00110110$

Thus: $X = \frac{1.1110100001 \times 2^{19}}{1.1001001 \times 2^1} = 1.0011011 \times 2^{18} = 1.2109375 \times 2^{18} = 317440$

$e + bias = 18 + 127 = 145 = 10010001$

X = 0100 1000 1001 1011 0000 0000 0000 0000 = 489B0000

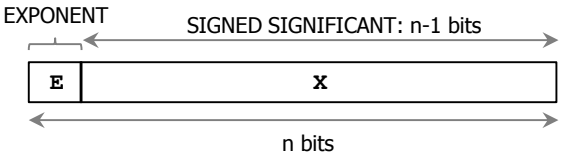
DUAL FIXED-POINT (DFX) ARITHMETIC

- Floating point (FP) and fixed point (FX) arithmetic are the standard numerical representations. Floating point features a large dynamic range at the expense of a large resource usage. On the other hand, fixed point requires fewer resources, but it delivers a low dynamic range. Dual fixed point (DFX) arithmetic is an alternative representation that overcomes the limitations of FX (it greatly improves dynamic range) without the high resource complexity of floating point.

- $n-bit$ Dual Fixed Point (DFX) number: exponent (E), signed significand (X) with $n-1$ bits.
- Exponent 'E': It selects between two scalings for the significand X. Thus, there are two possible cases for a DFX number D:

$$D = \begin{cases} num0{:}\ X.2^{-p_0}, & if\ E = 0 \\ num1{:}\ X.2^{-p_1}, & if\ E = 1' \end{cases} \qquad p_0 > p_1$$

  ✓ $num0$: It has p_0 fractional bits. $num1$: It has p_1 fractional bits.
- Notation of a DFX number: n_p0_p1 or $[n\ p_0\ p_1]$.



BOUNDARY VALUE

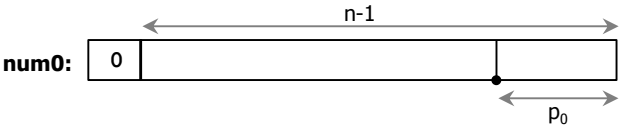
- The scaling (value of E) is determined by the boundary value B:

$$E = \begin{cases} 0\ (num0), & -B \le D < B \\ 1\ (num1), & D < -B\ and\ D \ge B \end{cases}$$

- Boundary value B: Defined as the next incremental value after the maximum positive number of $num0$:

$$Range\ of\ num0 = \frac{-2^{(n-1)-1}}{2^{p_0}}\ to\ \frac{2^{(n-1)-1}-1}{2^{p_0}}$$

$$B = \frac{2^{(n-1)-1}-1}{2^{p_0}} + 1\ LSB = \frac{2^{(n-1)-1}-1}{2^{p_0}} + \frac{1}{2^{p_0}} \rightarrow B = 2^{n-p_0-2}$$



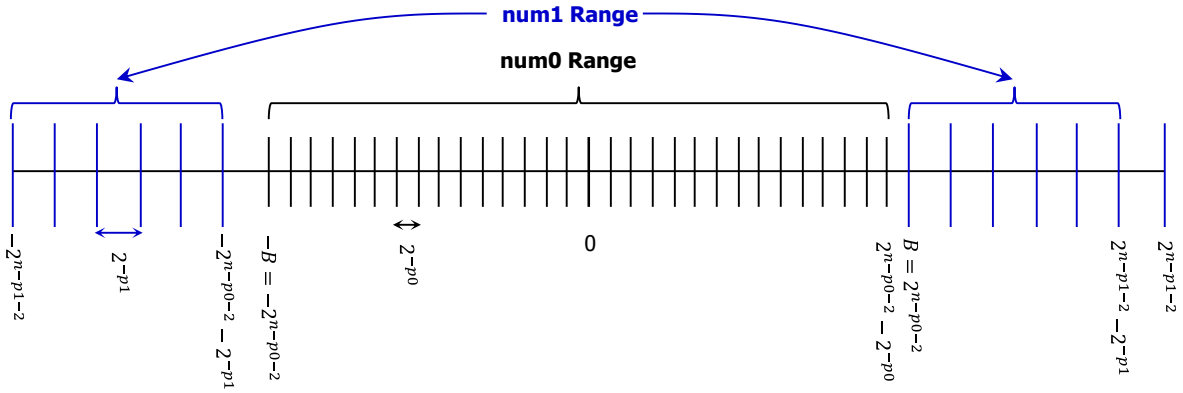
- If we have a $num0$ number with n bits, it seems that we could convert it into a $num1$ number with n bits (by truncating $p_0 - p_1$ LSBs and by sign-extending the MSB of the significand). However, we cannot do this, as the number will not be in the $num1$ range ($D < -B\ and\ D \ge B$).
- To convert a $num0$ number that does not fit with n bits (but with $n+x$ bits, format $(n+x)\_p_0\_p_1$) into a $num1$ number that might fit with n bits (format $n\_p_0\_p_1$), we need to discard the $p_0 - p_1$ LSBs and then sign-extend the significand by $p_0 - p_1 - x$ bits.

  Example: Format 16_7_3 to 14_7_3: $x = 2, p0 - p1 = 4$

  $num0$ number in format 16_7_3: 0 1011 0111.0101101
  Convert to 14_7_3: it has to be $num1$: 1 111011 0111.010 ($num0$ would not work as we will lose MSBs)

RANGES FOR $num0$ and $num1$:

- $num0$ range: There is smaller spacing (2^{-p_0}) between consecutive numbers; thus, they are more accurate.
- $num1$ range: There is larger spacing (2^{-p_1}) between consecutive numbers; thus, they are less accurate.



- Range for $num0$: $[-2^{n-p_0-2}, 2^{n-p_0-2} - 2^{-p_0}]$
- Range for $num1$: $[-2^{n-p_1-2}, -2^{n-p_0-2} - 2^{-p_1}] \cup [\ 2^{n-p_0-2}, 2^{n-p_1-2} - 2^{-p_1}]$

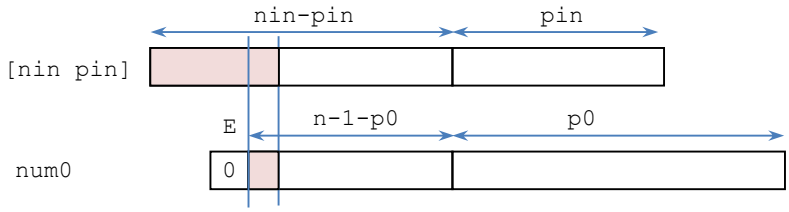
DYNAMIC RANGE

- It is defined as the ratio between the largest absolute value and the smallest nonzero absolute value.

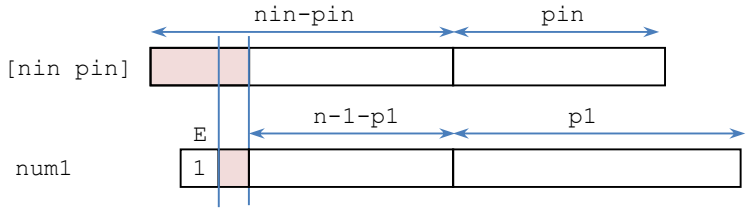
✓ Unsigned numbers with n bits:	$\frac{\lvert 2^n-1\rvert}{\lvert 1\rvert} = 2^n-1 \rightarrow Dynamic\ Range = 20\log_{10}(2^n-1)\ dB$
✓ Signed numbers (2's complement): $[n\ p]$	$\frac{\lvert -2^{n-1-p}\rvert}{\lvert 2^{-p}\rvert} = 2^{n-1} \rightarrow Dynamic\ Range = 20\log_{10}(2^{n-1})\ dB$
✓ Dual fixed point (DFX) numbers: n_p0_p1	$\frac{\lvert -2^{n-2-p_1}\rvert}{\lvert 2^{-p_0}\rvert} = 2^{n-2-p_1+p_0} \rightarrow Dynamic\ Range = 20\log_{10}(2^{n-2-p_1+p_0})\ dB$

FX TO DFX CONVERSION

- What if we have $[nin\ pin]$ (signed) and we want to convert to n_p0_p1?
  - ✓ We first try with $num0$ (since it is more accurate).
  - ✓ For $num0$ we need: $-B \le D < B, B = 2^{n-p_0-2}$. Note that $B - 2^{-p_0} = \mathbf{0}0111\ldots11$. $-B = \mathbf{1}0000\ldots000$. A quick way to check this is by aligning the two formats and then comparing the $nin - pin - (n-1-p0) + 1$ MSBs of the FX format with the MSB (of the significand) of the DFX number. If they are all the same, that means that the number is a $num0$.



  - ✓ If the number is not a $num0$, we try with $num1$:

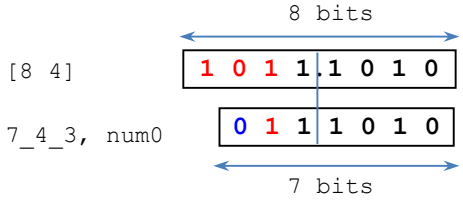


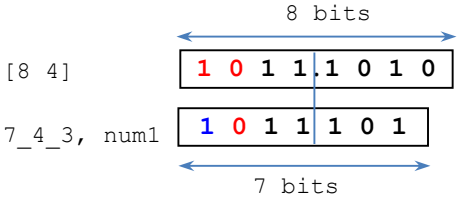
  - ✓ If $num1$ does not work either, it means there is overflow and we need more than n bits.

- Note: This procedure always starts by checking if the number is $num0$. A common mistake is to start by checking if the number is $num1$. The procedure only works for continuous ranges, which is not the case of $num1$. To directly verify that a DFX number is a $num1$, see if we can drop off enough MSBs so as to make it a $num0$. If the integer part is affected, then the number is a $num1$; otherwise it is a $num0$.

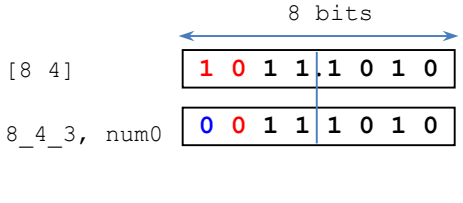
Examples:

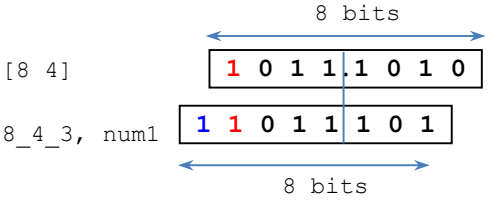
- Signed $[8\ 4]$ to 7_4_3. We first check for $num0$, then for $num1$. None of these work, so there is overflow (need more bits).





- Signed $[8\ 4]$ to 8_4_3. We first check for $num0$, which does not work. Then, we check for $num1$, which works.





Examples

- Convert the following signed fixed-point numbers in format [12 8] to Dual Fixed Point Format 12_8_4:

  ✓ E.EE:
  1110.1110 1110 ⇒ To DFX 12_8_4 ($num0$): 011011101110 = 6EE

  ✓ A.CD:
  1010.1100 1101 ⇒ To DFX 12_8_4 ($num0$): 001011001101 = not a $num0$!
  ⇒ To DFX 12_8_4 ($num1$): 111110101100 = FAC

  ✓ C.1B:
  1100.0001 1011 ⇒ To DFX 12_8_4 ($num0$): 010000011011 = 41B

  ✓ 8.B9:
  1000.1011 1001 ⇒ To DFX 12_8_4 ($num0$): 000010111001 = not a $num0$!
  ⇒ To DFX 12_8_4 ($num1$): 111110001011 = F8B

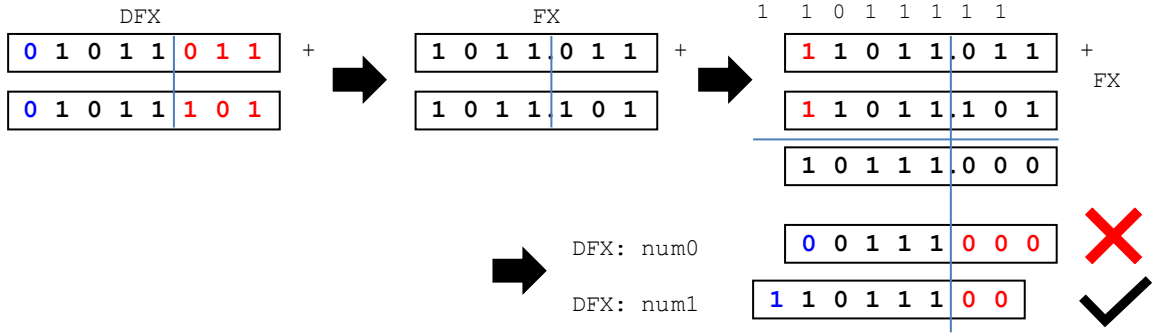
  ✓ 3.0A:
  0011.0000 1010 ⇒ To DFX 12_8_4 ($num0$): 011000001010 = 30A

DUAL FIXED-POINT ADDITION

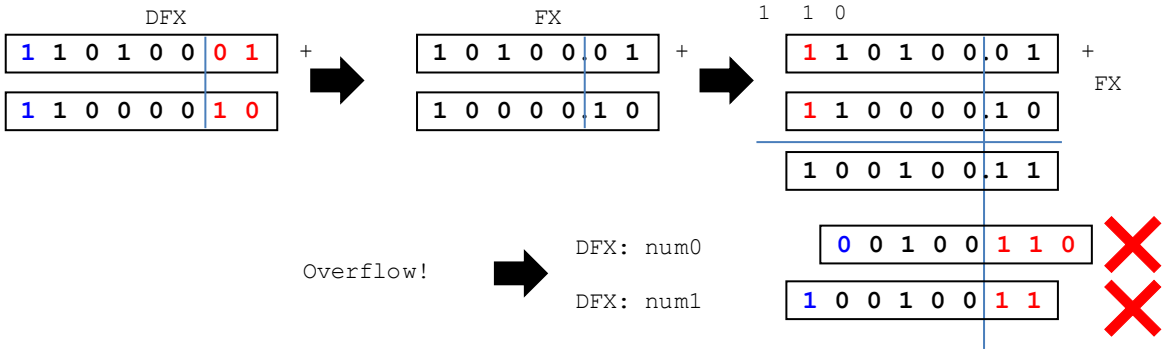
- Here, we add two DFX numbers A and B with n bits. To do this, we need to follow this procedure:
  - ✓ **Pre-scaling**: We get rid of the exponent (E) bit. Then, we align the two $(n-1)$-bit significands (signed FX numbers). Here, alignment might require discarding $p_0 - p_1$ fractional bits (truncation) and sign-extending MSBs by $p_0 - p_1$ bits. *Improving DFX Adder accuracy*: We can save the $p_0 - p_1$ truncated LSBs. In the post-scaler, we might be able to shift in the truncated LSBs. This only works when A and B have different exponents.
  - ✓ **Fixed-point addition**: This is a simple addition of two $(n-1)$-bit significands. Note that in the addition result can have up to n bits. The format would be either $[n\ p_0]$ or $[n\ p_1]$.
  - ✓ **Post-scaling**: The n-bit FX result must be converted to n-bit DFX. In some cases, there can be overflow when the result cannot fit as a $num1$ in DFX.

Examples: Addition of numbers in format 8_3_2. The result should be in the same format.

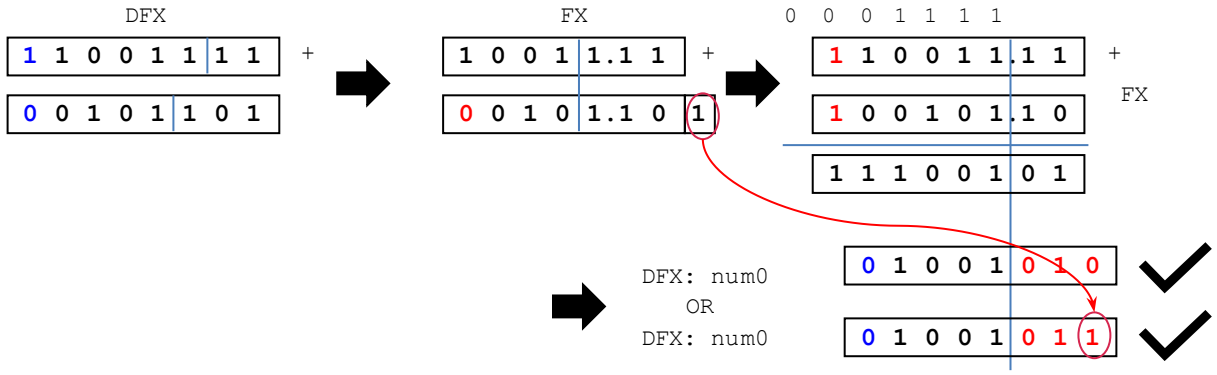
- Addition of two $num0$ numbers: FX addition requires 9 bits. When this FX number is converted to DFX, we notice that it cannot be represented as a $num0$ (unless we change the DFX format to 9_3_2); it will have to be a $num1$.



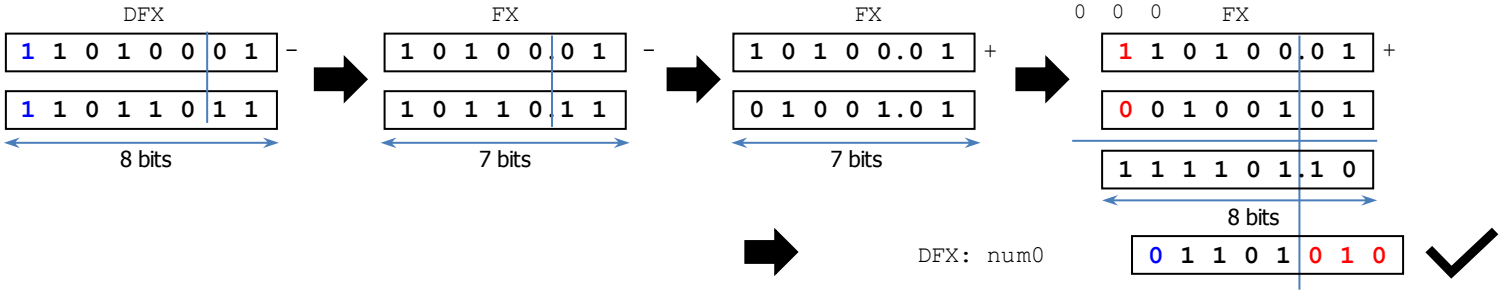
- Addition of two $num1$ numbers. Addition results in overflow (we need 9 bits to represent the result). Here, the result in format 8_3_2 has to include an overflow flag. We can recover from this if we include a carry out (c_n) bit in the circuitry.



- Addition of a $num0$ and a $num1$ number. Addition does not result in overflow, $num0$ suffices to represent the addition in the 8_3_2 format. Here, there are two ways: add zero the LSB or retrieve the saved bit.



- Subtraction of two $num1$ numbers. The operation does not result in overflow, $num0$ suffices to represent the final result in the 8_3_2 format.



DUAL FIXED-POINT MULTIPLICATION

- Here, we add two DFX numbers with n bits. To do this, we get rid of the exponent (E) bit, and then we only need to multiply two $(n-1)$-bit significands in fixed point arithmetic. Then, we convert the FX result into the DFX number.
- Unlike DFX addition, there is no pre-scaling. We just multiply the two signed FX numbers. The result will have 2n-2 bits. We need a Post-Scaling stage that converts this FX result into the DFX format.

- **Examples**: DFX multiplication of numbers in [8 3 2] format.