Notes - Unit 1

COMPUTER ARITHMETIC

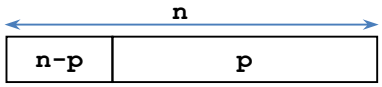
INTEGER NUMBERS

- $n-bit$ number: $b_{n-1}b_{n-2}\dots b_0$

	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: $b_{n-p-1}b_{n-p-2}\dots b_0.b_{-1}b_{-2}\dots b_{-p}$



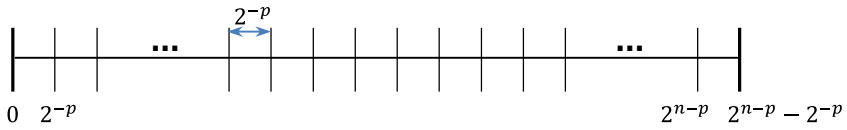
	UNSIGNED	SIGNED
Decimal Value	$D=\sum_{i=-p}^{n-p-1} b_i 2^i$	$D=-2^{n-p-1}b_{n-p-1}+\sum_{i=-p}^{n-p-2} b_i 2^i$
Range of values	$\left[\frac{0}{2^p}, \frac{2^n-1}{2^p}\right]=[0, 2^{n-p}-2^{-p}]$	$\left[\frac{-2^{n-1}}{2^p}, \frac{2^{n-1}-1}{2^p}\right]=[-2^{n-p-1}, 2^{n-p-1}-2^{-p}]$
Dynamic Range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

- Dynamic Range:

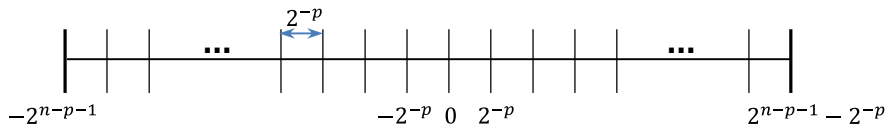
$$Dynamic\ Range=\frac{largest\ abs.value}{smallest\ nonzero\ abs.value}$$

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

- Unsigned numbers: Range of Values



- Signed numbers: Range of Values

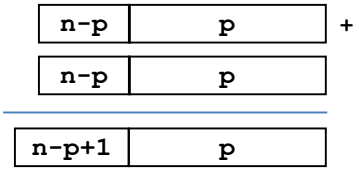


- Examples:

	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

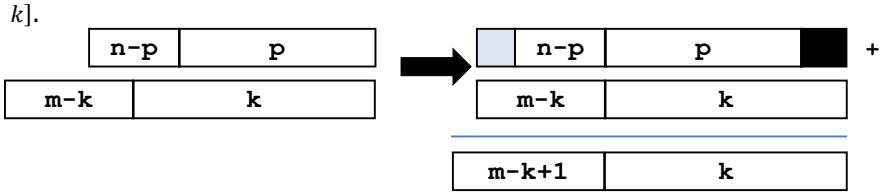
BASIC OPERATIONS

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

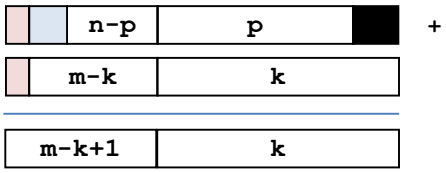


$A \times 2^{-p} \pm B \times 2^{-p} = (A \pm B) \times 2^{-p}$
We perform integer addition/subtraction of A and B. We just need to interpret the result differently by placing the fractional point where it belongs. Notice that the hardware is the same as that of integer addition/subtraction.

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 $[m+1\ k]$.

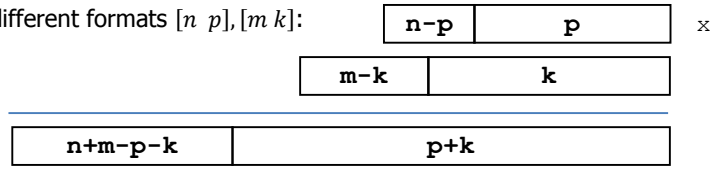


Important: The result of the addition/subtraction requires an extra bit in the worst-case scenario. In order to correctly compute it in fixed-point arithmetic, we need to sign-extend (by one bit) the operators prior to addition/subtraction.



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 $n - p + \lceil \log_2 N \rceil$.

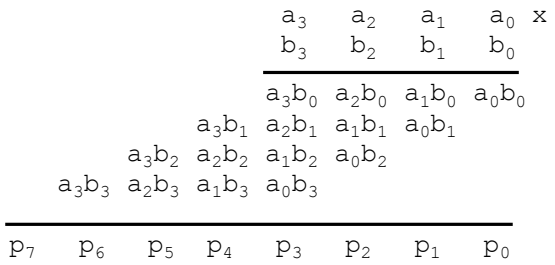
- **Multiplication:**
Multiplication of two signed numbers represented with different formats $[n\ p], [m\ k]$:



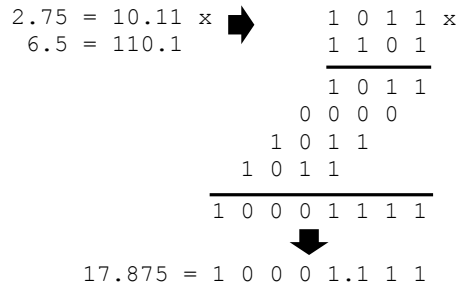
$(A \times 2^{-p}) \times (B \times 2^{-k}) = (A \times B) \times 2^{-p-k}$. We can perform integer multiplication of A and B and then place the fractional point where it belongs. The format of the multiplication result is given by $[n+m\ p+k]$. There is no need to align the fractional point of the input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

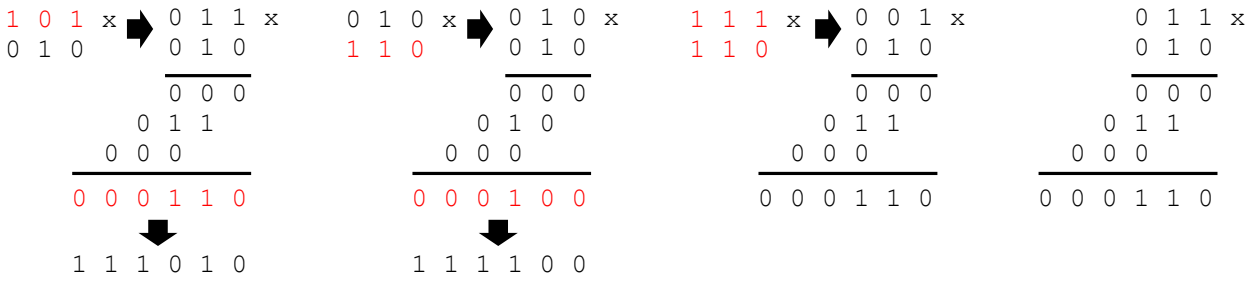
✓ Multiplication procedure for unsigned integer numbers:



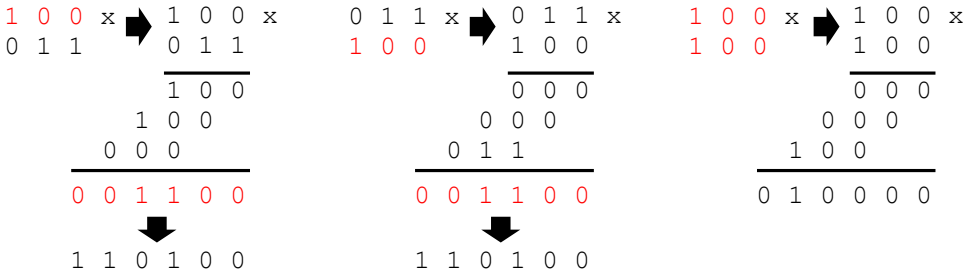
Example: when multiplying, we treat the numbers as integers. Only when we get the result, we place the fractional point where it belongs.



✓ 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.

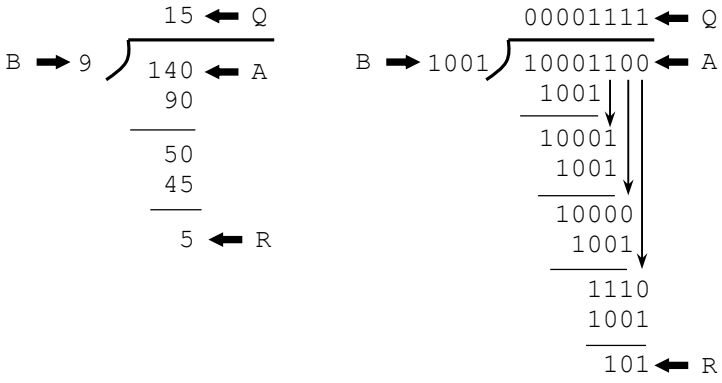


Note: If one of the inputs is -2^{n-1}, then the 2's complement of it is 2^{n-1}, which requires $n+1$ bits. Here, we are allowed to use only n bits, i.e., we do not need to change its sign. This will not affect the final result since if we were to use $n+1$ bits for 2^{n-1}, the MSB=0, which implies that the last row is full of zeros.
Final output: It requires $n+m$ bits. Note that it is only because of the multiplication of -2^{n-1} by -2^{m-1} that we require those $n+m$ bits (in 2's complement representation).



- **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.




```
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
```

✓ Division of unsigned fixed-point numbers: A_f/B_f
To do this, we first need to align the numbers, 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 $[nb\ b]$

Step 1: For $a \geq b$, we align the fractional points and then get the integer numbers A and B, which result from:

$$A = A_f \times 2^a \qquad B = B_f \times 2^a$$

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:

$$A_f \times 2^a = \left(B_f \times 2^a\right) \times Q + R \rightarrow A_f = B_f \times Q + (R \times 2^{-a})$$

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: $1010{,}011 = 11{,}1(10) + 11{,}011,\ Q_f = 10, R_f = 11011$

- ✓ Including precision to the division of unsigned fixed-point numbers: 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.

$$A = A_f \times 2^a \times 2^x \qquad B = B_f \times 2^a$$

$$A_f \times 2^{a+x} = \left(B_f \times 2^a\right) \times Q + R \rightarrow A_f = B_f \times (Q \times 2^{-x}) + (R \times 2^{-a-x})$$

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

$$\frac{1010011}{11100} = \frac{101001100}{11100}$$

Step 3: Integer Division

$$\frac{101001100}{11100} \Rightarrow 101001100 = 11100(1011) + 11000$$

$$Q = 1011, R = 11000$$

Step 4: Get actual remainder and quotient (or result): $Q_f = Q \times 2^{-x}, R_f = R \times 2^{-a-x}$

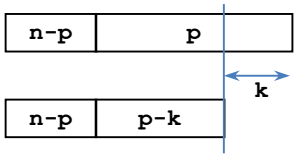
$$Q_f = 10{,}11, R_f = 0{,}11000$$

Verification: $1010{,}01100 = 11{,}1(10{,}11) + 0{,}11000.$

- ✓ Signed division: In this case (just 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 abs(A)/abs(B) requires a sign change.

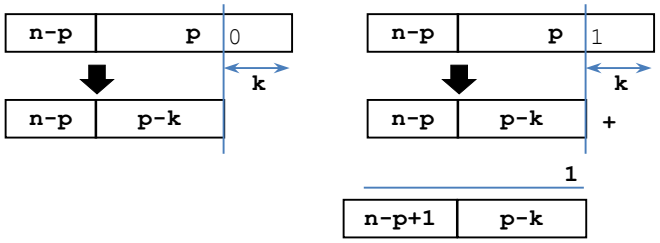
TRUNCATION/ROUNDING

Truncation:



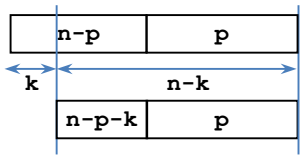
Truncation is helpful when less hardware is required. However this comes at the expense of less accuracy.
To assess the effect of truncation, use PSNR (dB) with respect to a double floating point result or with respect to the original $[n\ p]$.

Rounding:



This operation is helpful when less hardware is required in subsequent operations. However, it requires extra hardware to deal with the rounding.
Usually, if we want to chop 'p' bits, we use the last bit to determine whether to round. If the bit is zero, we just truncate. If the bits is 1, we have to add one to the truncated result.

Saturation:



This is helpful when we need to restrict the number of integer bits. Here, we chop off the integer part by k bits. The result might completely modify the number.
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. If the MSB of the initial number is 1, the resulting $(n-k)$-bit number will be $-2^{n-k-p-1} = 10\ldots0$ (largest negative number). If the MSB is 0, the resulting $(n-k)$-bit number will be $2^{n-k-p-1} - 2^{-p} = 011\ldots1$ (largest positive number).

Example: We can only use 8 bits in the following signed fixed-point format: $[8\ 7]$

- $1{,}01101111$:
  We need to represent this number in the format $[8\ 7]$. We are only required to 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) and saturation:
  Saturation in this case amounts to truncation of the MSB, as the number won't change.
  After rounding: $11{,}1110100 + 1 = 11{,}1110101$
  After saturation: $1{,}1110101$

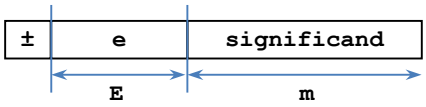
- $101{,}111010011$:
  Here, we need to get rid of two MSB and two LSBs.
  Saturation: Since the three MSBs are not the same and the MSB=1 we need to replace the number by the largest negative number (in absolute terms) in the format $[8\ 7]$: $1{,}0000000$

- $011{,}1111011011$:
  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 REPRESENTATION

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

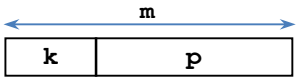
$$X = \pm significand \times 2^{e}$$



- 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.: $[0,1), [1,2)$.



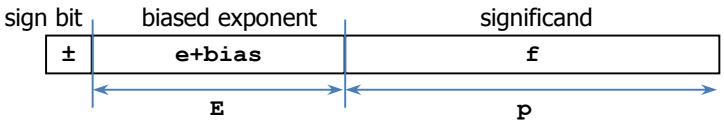
Format (unsigned): $[m\ \ p]$. Range: $\left[0, \frac{2^m-1}{2^p}\right] = [0, 2^{m-p} - 2^{-p}]$, $k = m - p$
If $k = 0 \rightarrow$ Significand $\in [0,1 - 2^{-p}] = [0,1)$
If $k = m \rightarrow$ Significand $\in [0, 2^m - 1]$. Integer significand.

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:

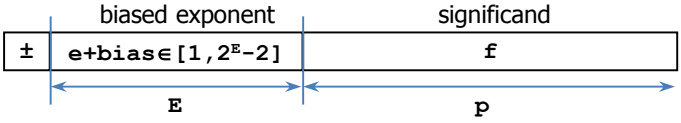
$$X = \pm 1.f \times 2^{e}$$



- **Significand**: 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: $[1,2 - 2^{-p}] = [1,2)$

- **Biased exponent**: 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 un 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: $largest\ significand \times 2^{largest\ exponent}$

$$max = 1.11\ldots1 \times 2^{2^{E-1}-1} = (2 - 2^{-p}) \times 2^{2^{E-1}-1}$$

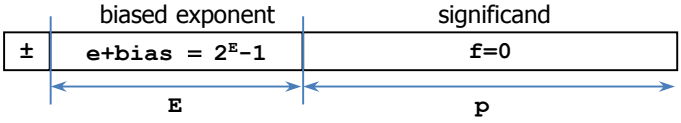
Min. number: $smallest\ significand \times 2^{smallest\ exponent}$

$$min = 1.00\ldots0 \times 2^{-2^{E-1}+2} = 2^{-2^{E-1}+2}$$

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

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

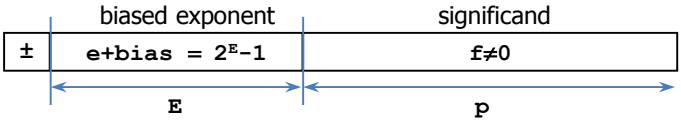
- **Plus/minus Infinite:** $\pm\infty$



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

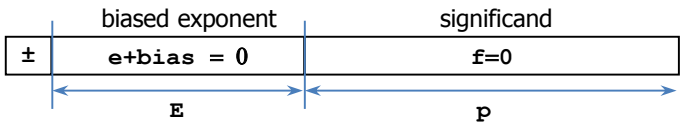
$$\pm\infty = \pm 2^{2^{E-1}}$$

- **Not a Number:** NaN



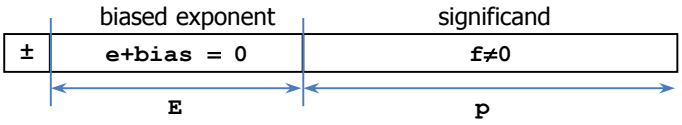
The exp field is a strings of 1's. $exp = 2^E - 1$. This is a special case where $exp = 2^E - 1$ ($e = 2^{E-1}$). The only difference with $\pm\infty$ is that f is a nonzero number.

- **Zero**:



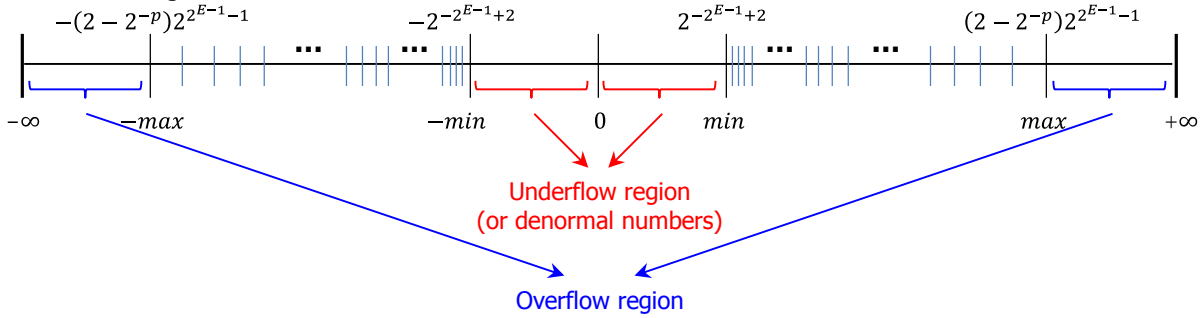
Zero cannot be represented with a normalized significand $s = 1.00\ldots0$ since $X = \pm 1.f \times 2^e$ cannot be zero. Thus, a special code must be assigned to it, where $s = 0.00\ldots0$ and $exp = 0$. Every single bit (except for the sign) is zero. There are two representations for zero.
The number 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 normalized is smaller. Though this specification makes the formula $e + bias = 0$ inconsistent, it helps in accuracy.

- Depiction of the range of values:



- The IEEE-754 standard defines two representations: single (32 bits, E = 8, p = 23) and double (64 bits, E = 11, p = 52). However, you can define your own representation by selecting a particular number of bits for the exponent and significand. The following table specifies various parameters for single and double floating point arithmetic (ordinary numbers):

	Ordinary numbers		Exponent bits (E)	Range of e	Bias	Dynamic Range (dB)	Significand range	Significand bits (p)
	Min	Max						
Single	2^{-126}	$(2-2^{-23})2^{+127}$	8	$[-126,127]$	127	759 dB	$[1,2-2^{-23}]$	23
Double	2^{-1022}	$(2-2^{-52})2^{+1023}$	11	$[-1022,1023]$	1023	6153 dB	$[1,2-2^{-52}]$	52

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

Examples:

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

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

BASIC OPERATIONS:

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

  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 zeros of the result and stop when a leading one is found. Then, we need to adjust e_1 properly, this results in e.
  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 very close to 0 and we might need to first discard the leading zeros and then right shift until we get $1.f$. Then, we 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: $s_3 = \left(\pm s_1 \pm \frac{s_2}{2^{e_1 - e_2}}\right) = 00011.1010$
  We first need to discard the leading zeros: $s_3 = 11.1010$
  Then, we right shift 3 bits: $s = s_3 \times 2^{-1} = 1.0011010$.
  Now that we have the normalized significand s, we need to adjust the exponent e_1 by adding 1 to it: $e = e_1 + 1$:
  $$(s_3 \times 2^{-1}) \times 2^{e_1 + 1} = s \times 2^e = 1.001101 \times 2^{e_1 + 1}$$

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

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

  $1.0101 - 0.011110 = 0.11011$. This might require i) converting the numbers to 2C representation prior to subtraction, and ii) sign extension. Also, if the result is negative, we have to convert it to positive, and then include the negative sign.

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

- **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 $s = (s_1 \times s_2) \in [1,4)$.

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

  $b = b_1 \times b_2 = -(1.100 \times 1.011) \times 2^6 = -(10,0001) \times 2^6$,

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

  Note that 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.

- **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 $s = \left(\frac{s_1}{s_2}\right) \in (1/2, 2)$
  Here, 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.

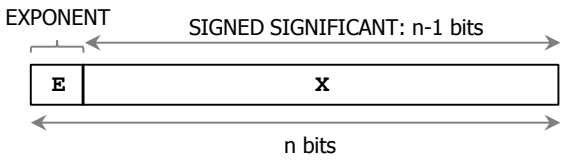
  $$\rightarrow \frac{b_1}{b_2} = \frac{1.100 \times 2^2}{-1.011 \times 2^4} = -1.0101 \times 2^{-2}$$

DUAL FIXED-POINT ARITHMETIC

- $n - bit$ 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:

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

  ✓ $num0$: It has p_0 fractional bits. $num1$: It has p_1 fractional bits.
- Notation of a DFX number: n_p0_p1



BOUNDARY VALUE

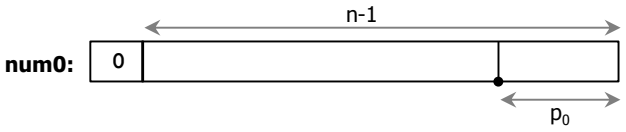
- The scaling (value of E) is decided by the boundary value B:

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

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

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

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



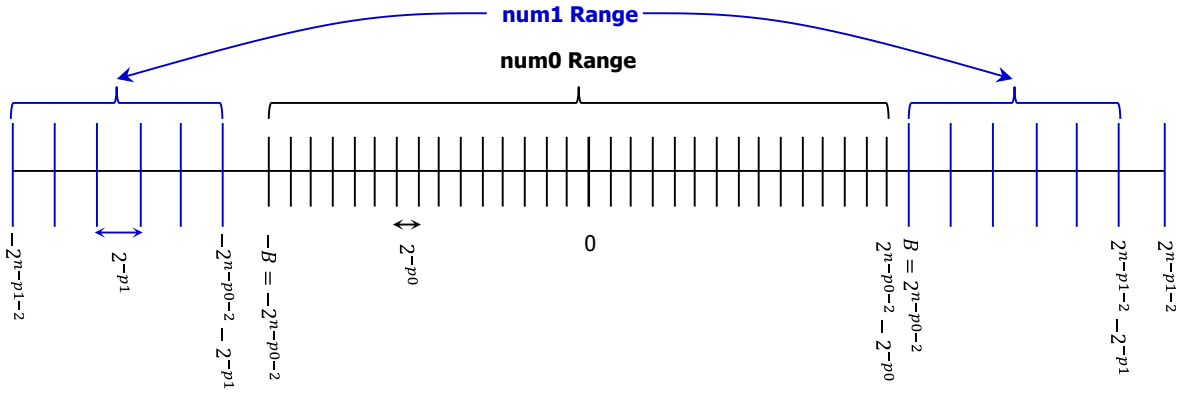
- If we have a $num0$ number with n bits, it seems that we could convert it into a $num1$ number with n bits (by truncating $p_0 - p_1$ LSBs and by sign-extending the MSB of the significand). However, we cannot do this, as the number will not be in the $num1$ range ($D < -B\ and\ D \ge B$).
- To convert 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 $num1$: $[-2^{n-p_1-2}, -2^{n-p_0-2} - 2^{-p_1}] \cup [\, 2^{n-p_0-2}, 2^{n-p_1-2} - 2^{-p_1}]$

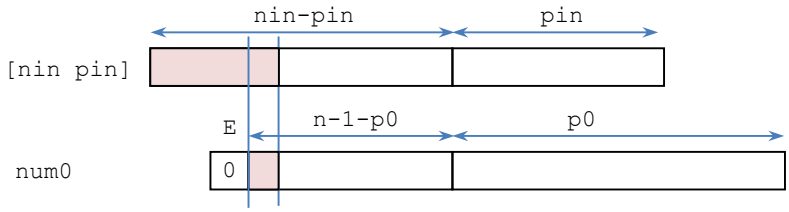
DYNAMIC RANGE

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

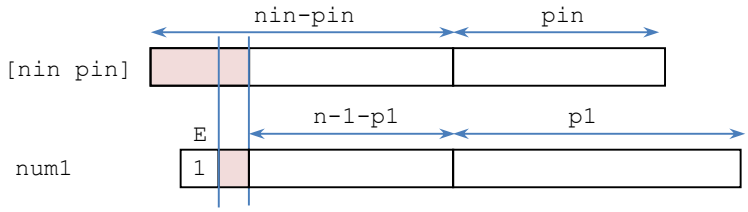
✓ Unsigned numbers with n bits:	$\frac{\lvert 2^n - 1\rvert}{\lvert 1\rvert} = 2^n - 1 \rightarrow Dynamic\ Range = 20\log_{10}(2^n - 1)\ dB$
✓ Signed numbers (2's complement): $[n\ p]$	$\frac{\lvert -2^{n-1-p}\rvert}{\lvert 2^{-p}\rvert} = 2^{n-1} \rightarrow Dynamic\ Range = 20\log_{10}(2^{n-1})\ dB$
✓ 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 exact).
  For $num0$ we need: $-B \leq D < B, B = 2^{n-p_0-2}$. Note that $B - 2^{-p_0} = \mathbf{0}0111\ldots11$. $-B = \mathbf{0}10000\ldots000$. 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 not, we try with $num1$:

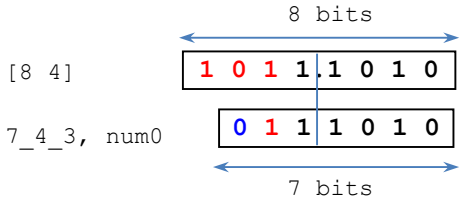


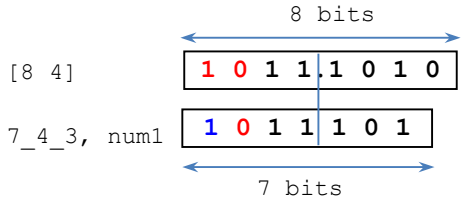
If not, it means we need more than n bits.

- Note that the procedure always start by checking if the number is $num0$. Sometimes it is common to just immediately assign the format of $num1$, without realizing that the number might not be in the $num1$ range. If we are given a $num1$ number and we want to verify it, a quick way is to align them and see if we can chop off enough MSBs so as to make it a $num0$ without affecting the integer part.

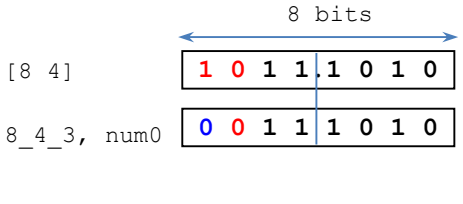
Example:

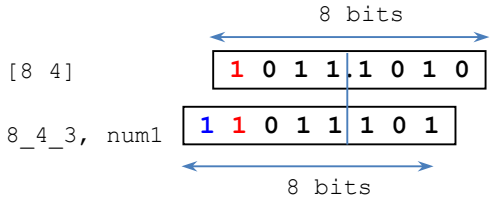
- [8 4] to 7_4_3. We first check for $num0$, then for $num1$. None of these work, so we need more bits.





- [8 4] to 8_4_3. We first check for $num0$, then for $num1$ (which is the one that works).



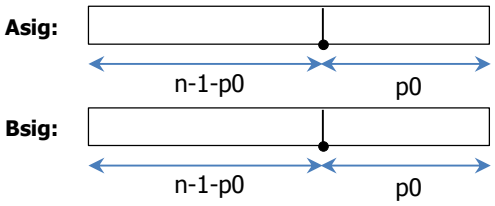


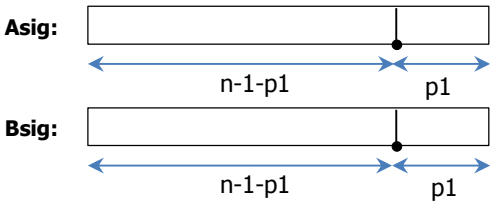
DUAL FIXED POINT ADDITION

- 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 add two $(n-1)$-bit significands in fixed point arithmetic.

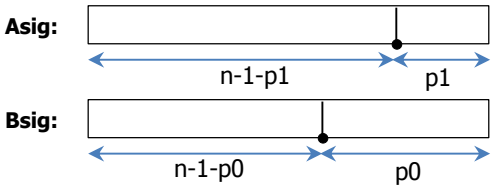
PRE-SCALER

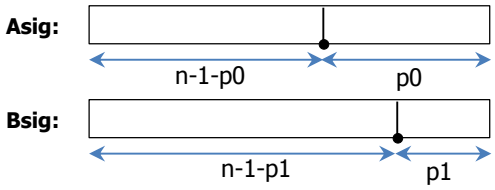
- If they are both either $num0$ or $num1$, addition is straightforward.

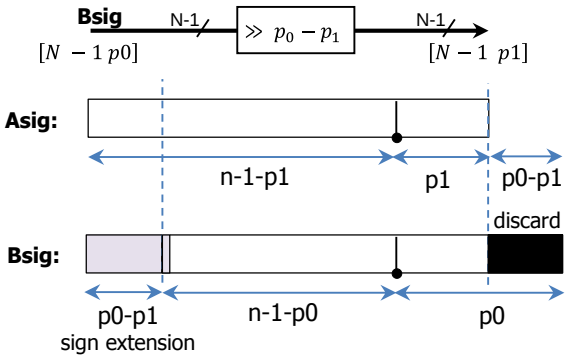


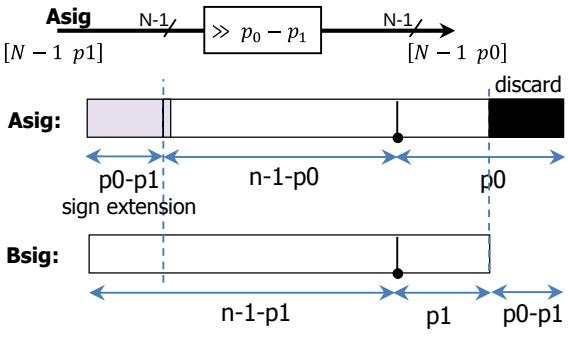


- If one is $num0$ and the other is $num1$, we have to align the fractional points to p_1. This means that we convert $[n-1\ p_0]$ to $[n-1\ p_1]$ by discarding $p_0 - p_1$ fractional bits and by sign-extending the extra $p_0 - p_1$ MSBs. This is not exactly the same as converting $num0$ to $num1$, because the $num0$ number fits with n bits, though the operation is very similar.









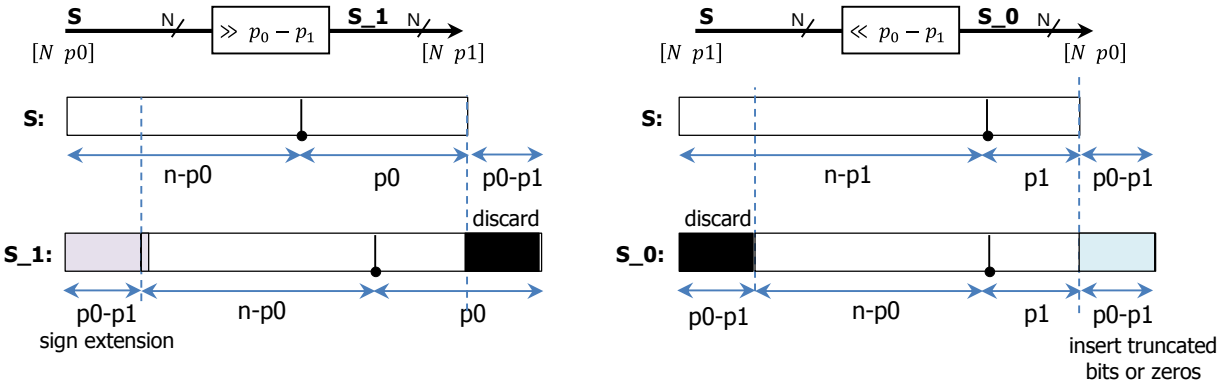
- Converting from $[n-1\ p_0]$ to $[n-1\ p_1]$: This operation consists of: arithmetic shift of $p_0 - p_1$ bits to the right, truncation of $p_0 - p_1$ LSBs, while keeping the fractional point where it is. This operation is not exactly $\gg p_0 - p_1$, but it is usually represented as such.
- **Improving DFX Adder accuracy**: We can save the $p_0 - p_1$ truncated LSBs. In the post-scaler, we might need to convert $[n\ p_1]$ to $[n\ p_0]$. This operation requires shifting to the left, and we can shift in the truncated LSBs. This only works when A and B have different exponents. If A and B are both $num0$, the sum S is $[n\ p_0]$: we cannot shift in any other bit. If A and B are both $num1$, the sum S is $[n\ p_1]$, and there were never truncated LSBs.

FIXED-POINT ADDITION

- Once the numbers are aligned, we perform the fixed point addition of two $(n-1)$-bit FX numbers. This is done by sign-extending the operands to n bits; the result has n bits with either p_0 or p_1 fractional bits.
- DFX addition: We want the result to have the same number of bits as the inputs. We can always sign-extend the MSB of the significand to avoid overflow, but this defeats the purpose of DFX (we better just use FX).
- Overflow of FX addition: Here, we consider the overflow as if the addition were of two $(n-1)$ –bit numbers (with no sign-extension), i.e., $c_{n-1} \oplus c_{n-2}$. We need this overflow since it tells us whether $n-1$ bits suffice for the addition result. Note that the n-bit addition overflow is always zero (due to sign-extension). The FX adder performs n-bit addition (by sign-extending), but at the end, we get rid of the MSB, so that the DFX result has one exponent bit and $n-1$ significand bits.

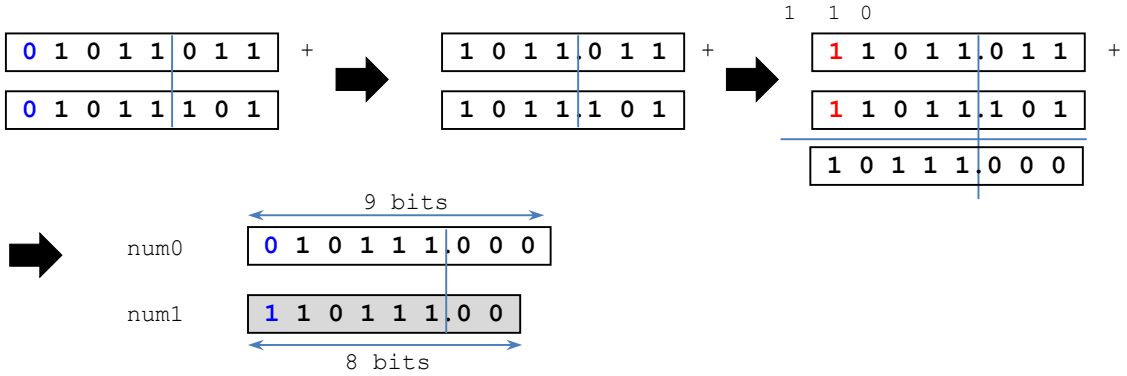
POST-SCALER

- If at least one input is $num1$, then the sum S will be in $[n\ p1]$. If A and B are $num0$, then the sum S will be $[n\ p0]$. The DFX n-bit number could be a $num0$ or $num1$. If the resulting $num0$ number has an overflow, we convert it to $num1$. Note that this resulting $num1$ number might require an overflow.
- From $[n\ p0]$ to $[n\ p1]$: This is the same circuit $\gg p0-p1$ as in the pre-scaler, but here we use n bits as input.
- From $[n\ p1]$ to $[n\ p0]$: Left shift with zero pad (or we shift in the truncated bits that we saved).

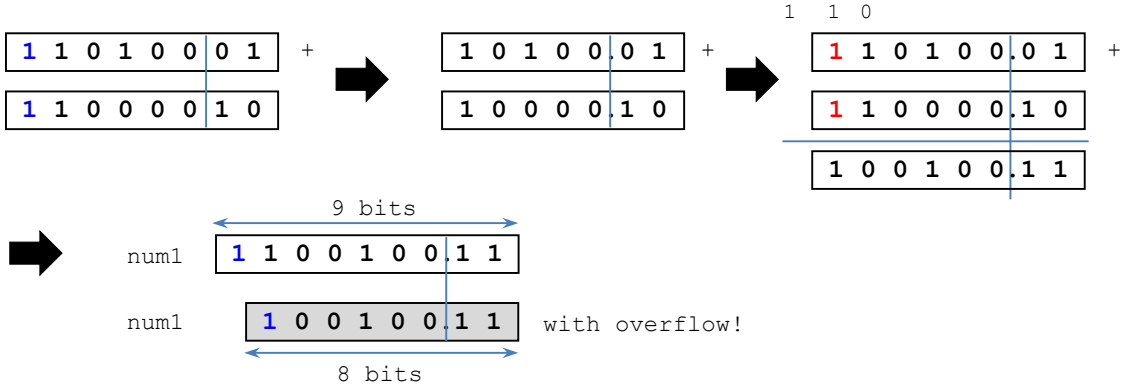


Examples:

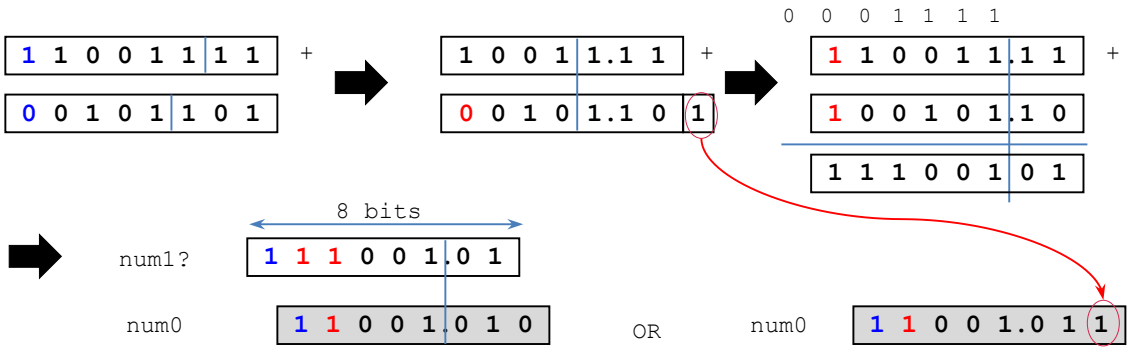
- Here, we have two $num0$ numbers in format 8_3_2. Addition results in overflow, requiring 9 bits to represent the number as a $num0$. A solution is to convert the number to $num1$ (in format 8_3_2) by discarding one LSB.



- Addition of two $num1$ numbers in format 8_3_2. Addition results in overflow, requiring 9 bits to represent the result as a $num1$. 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 in format 8_3_2. Addition does not result in overflow, only 8 bits are required to represent the number as $num1$ (tentatively). We realize that the number does not actually belong in the range of num1, thus we convert the number to $num0$: Here, there are two ways: add zero the LSB or retrieve the saved bit.



- Subtraction of two $num1$ numbers in format 8_3_2. The operation does not result in overflow, only 8 bits are required to represent the number as $num1$ (tentatively). We realize that the number does not actually belong in the range of $num1$, thus we convert the number to $num0$.