Formatting bits to better implement signal processing algorithms

Benoit Lopez - Thibault Hilaire - Laurent-Stéphane Didier

PECCS 2014

January 9^{th} 2014

Outline

In PECCS'14

Smartphone applications, measurement applications, embedded devices, ...

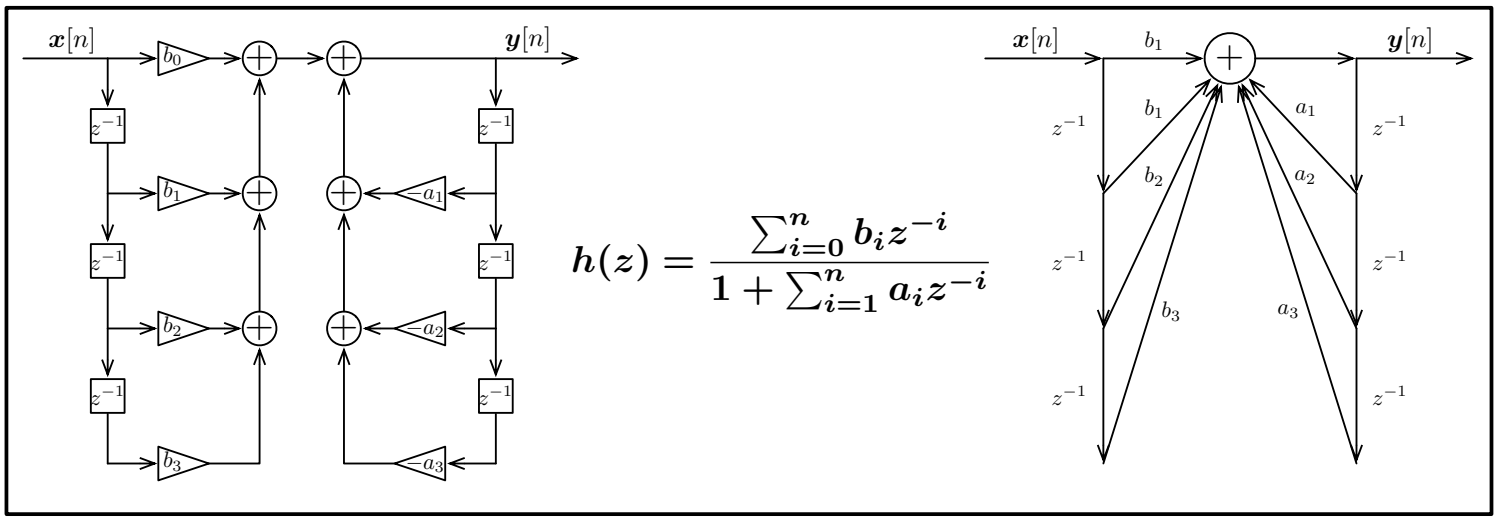
In PECCS'14

Smartphone applications, measurement applications, embedded devices, ...

Implementation problem

We need methodology and tools for the implementation of embedded filter algorithms with only integer arithmetic.

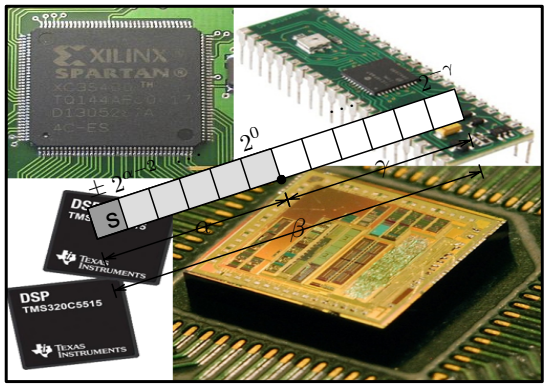
On the first hand... A filter



- Signal Processing
- LTI filters: FIR or IIR
- Its transfer function
- Algorithmic relationship used to compute output(s) from input(s), for example:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i)$$

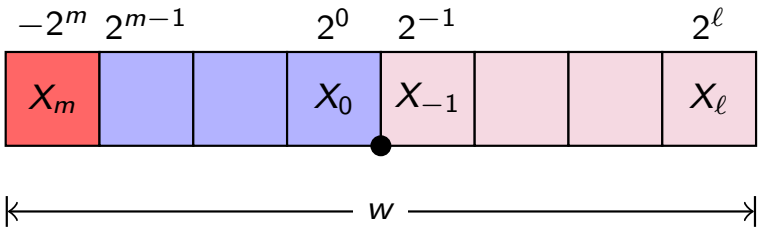
On the other hand... A target



- Hardware target (FPGA, ASIC) or software target (DSP,μC)
- Due to resources constraints (cost, size, power consumption, ...) we have no FPU, so we can only use fixed-point arithmetic

Fixed-Point number

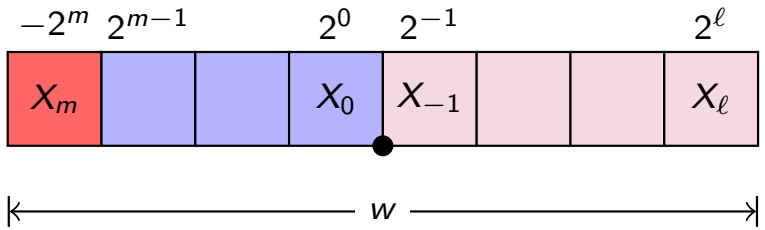
Fixed-point numbers are integers used to approximate real numbers.



- Representation : $X.2^\ell$ with $X = X_m X_{m-1} ... X_0 ... X_\ell$.
- Format : determined by wordlength and fixed-point position, and noted for example (m, ℓ).

Fixed-Point number

Fixed-point numbers are integers used to approximate real numbers.



- Representation : $X.2^\ell$ with $X = X_m X_{m-1} ... X_0 ... X_\ell$.
- Format : determined by wordlength and fixed-point position, and noted for example (m, ℓ).

Computation in finite precision and choice of formats imply errors.

Numerical degradations

- quantization of the coefficients
- round-off errors in computations

IIR Filter

Let H be the transfer function of a $n-th$ order IIR filter :

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}, \quad \forall z \in \mathbb{C}. \tag{1}$$

IIR Filter

Let H be the transfer function of a $n-th$ order IIR filter :

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}, \quad \forall z \in \mathbb{C}. \tag{1}$$

This filter is usually realized with the following algorithm

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \tag{2}$$

where $u(k)$ is the input at step k and $y(k)$ the output at step k.

IIR Filter

Let H be the transfer function of a $n-th$ order IIR filter :

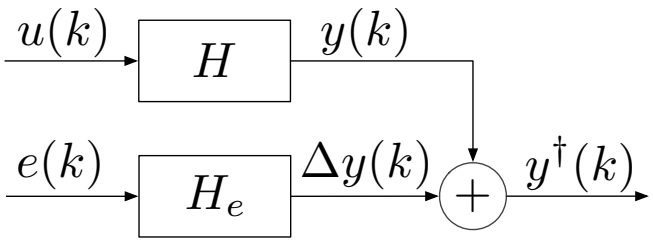
$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}, \quad \forall z \in \mathbb{C}. \tag{1}$$

This filter is usually realized with the following algorithm

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \tag{2}$$

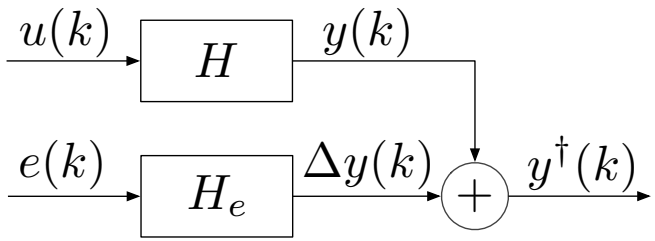
where $u(k)$ is the input at step k and $y(k)$ the output at step k. We can see round-off errors as the add of an error $e(k)$ on the output and only $y^\dagger(k)$ can be computed.

$$y^\dagger(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y^\dagger(k-i) + e(k). \tag{3}$$



$\Delta y(k) \triangleq y^\dagger(k) - y(k)$ can be seen as the result of the error through the filter H_e :

$$H_e(z) = \frac{1}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}, \quad \forall z \in \mathbb{C}.$$



$\Delta y(k) \triangleq y^\dagger(k) - y(k)$ can be seen as the result of the error through the filter H_e :

$$H_e(z) = \frac{1}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}, \quad \forall z \in \mathbb{C}.$$

If the error $e(k)$ is in $[\underline{e}; \overline{e}]$, then we are able to compute $\underline{\Delta y}$ and $\overline{\Delta y}$ such that $\Delta y(k)$ is in $[\underline{\Delta y}; \overline{\Delta y}]$:

$$\begin{aligned} \underline{\Delta y} &= \frac{\overline{e} + \underline{e}}{2} |H_e|_{DC} - \frac{\overline{e} - \underline{e}}{2} \|H_e\|_{\ell\infty} \\ \overline{\Delta y} &= \frac{\overline{e} + \underline{e}}{2} |H_e|_{DC} + \frac{\overline{e} - \underline{e}}{2} \|H_e\|_{\ell\infty} \end{aligned}$$

Fixed-Point implementation and especially the choice of formats, imply errors.
In the context of digital signal processing, we are able to control these errors.

Fixed-Point implementation and especially the choice of formats, imply errors.
In the context of digital signal processing, we are able to control these errors.

Objective:

Given an algorithm and a bound on the final error, find an implementation which reduces the number of bits of the computation while controlling the error on the output result.

Outline

Sum-of-Products

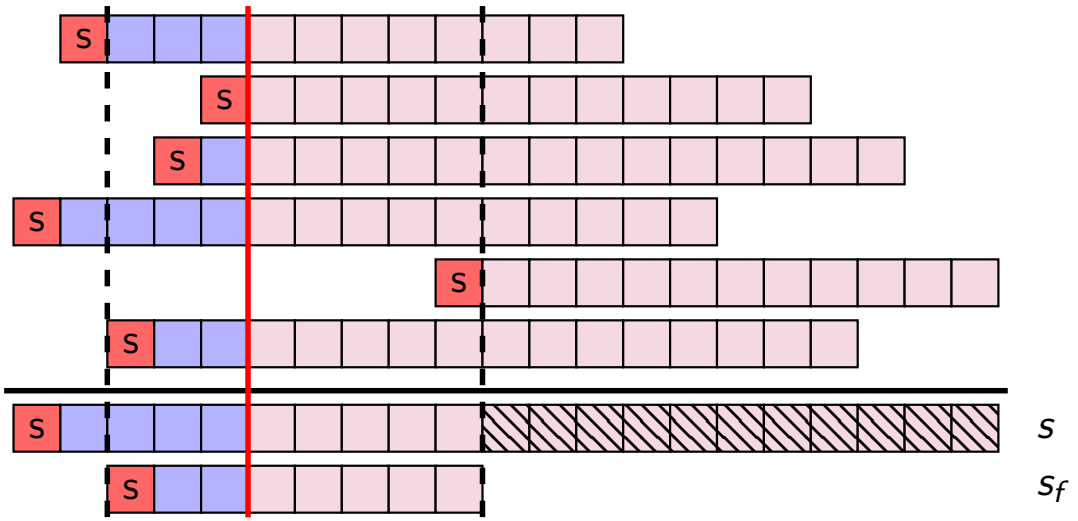
The only operations needed in filter algorithm computation are sum-of-products:

$$s = \sum_{i=1}^{n} c_i \cdot x_i = \sum_{i=1}^{n} p_i$$

where c_i are known constants and x_i variables (inputs, state or intermediate variables).

In the context of filter design, **we know the fixed-point format of the final result.**

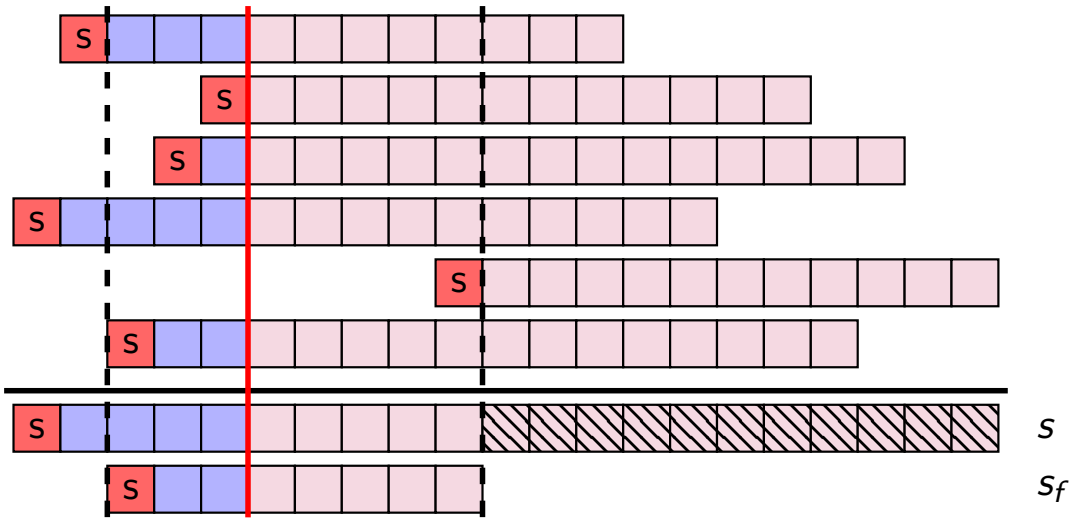
Formatting



Context

A sum of N terms $(p_i)_{1 \leq i \leq N}$ with different formats, and the known FPF of final result (s_f), less than total wordlength (s).

Formatting



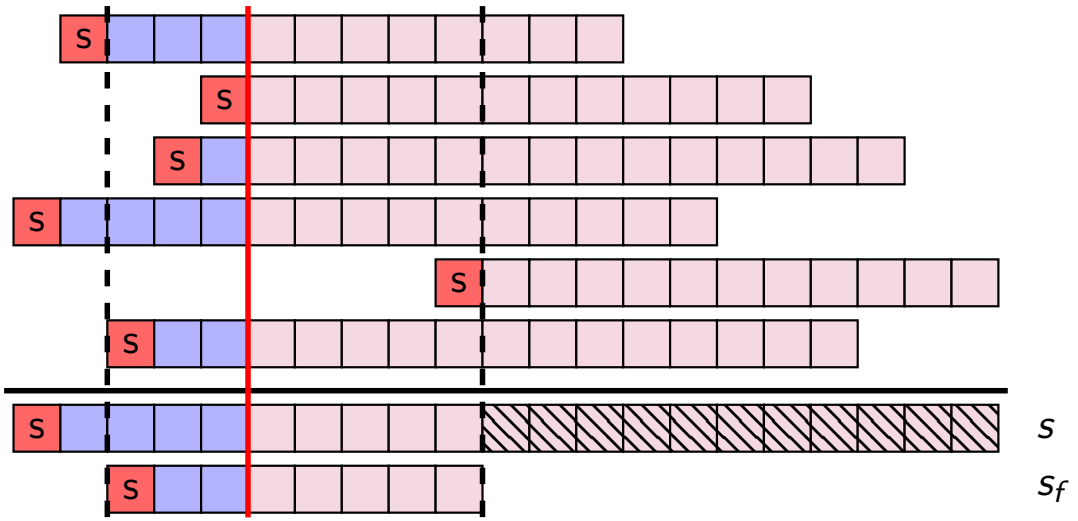
Context

A sum of N terms $(p_i)_{1 \leq i \leq N}$ with different formats, and the known FPF of final result (s_f), less than total wordlength (s).

Questions:

Can we remove bits that don't impact the final result ? If we want a faithful round-off of the final result, can we remove some bits ?

Formatting



Two-step formatting

1. most significant bits
2. least significant bits

Jacskon's Rule (1979)

This Rule states that in consecutive additions and/or subtractions in **two's complement arithmetic**, some intermediate results and operands may overflow. As long as the final result representation can handle the final result without overflow, then the result is valid.

Jacskon's Rule (1979)

This Rule states that in consecutive additions and/or subtractions in **two's complement arithmetic**, some intermediate results and operands may overflow. As long as the final result representation can handle the final result without overflow, then the result is valid.

Example :

We want to compute $104 + 82 - 94$ with 8 bits :

Jacskon's Rule (1979)

This Rule states that in consecutive additions and/or subtractions in **two's complement arithmetic**, some intermediate results and operands may overflow. As long as the final result representation can handle the final result without overflow, then the result is valid.

Example :

We want to compute $104 + 82 - 94$ with 8 bits :
$104 + 82 = -70$ overflow !

Jacskon's Rule (1979)

This Rule states that in consecutive additions and/or subtractions in **two's complement arithmetic**, some intermediate results and operands may overflow. As long as the final result representation can handle the final result without overflow, then the result is valid.

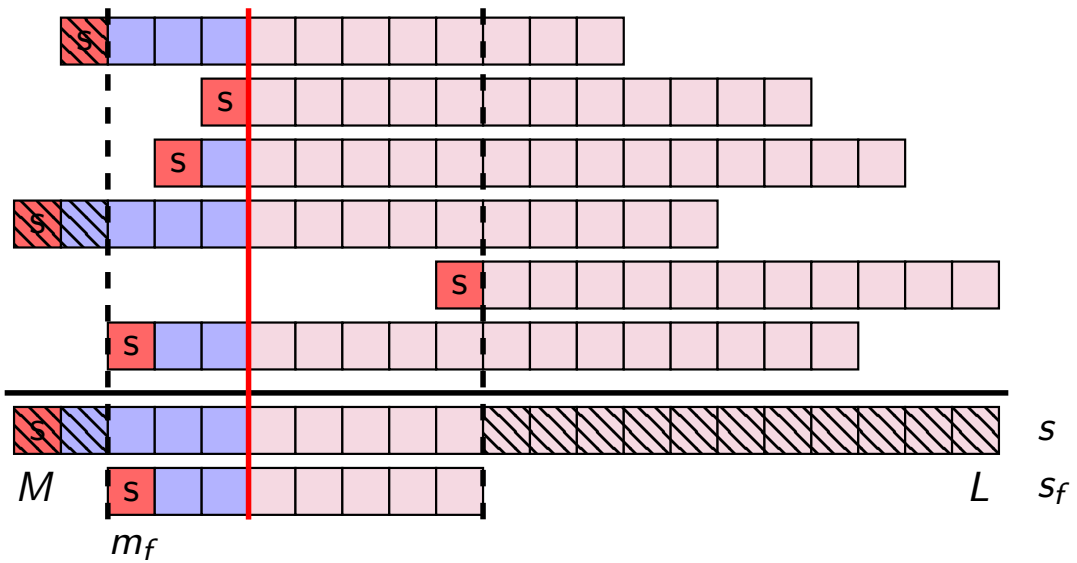
Example :

We want to compute $104 + 82 - 94$ with 8 bits :
$104 + 82 = -70$ overflow !
but $-70 - 94 = 92$ overflow !
This second overflow cancels the first one and we obtain the expected result.

Fixed-Point Jacskon's Rule

Let s be a sum of n fixed-point number p_is, in format (M, L). If s is known to have a final MSB equals to m_f with $m_f < M$, then:

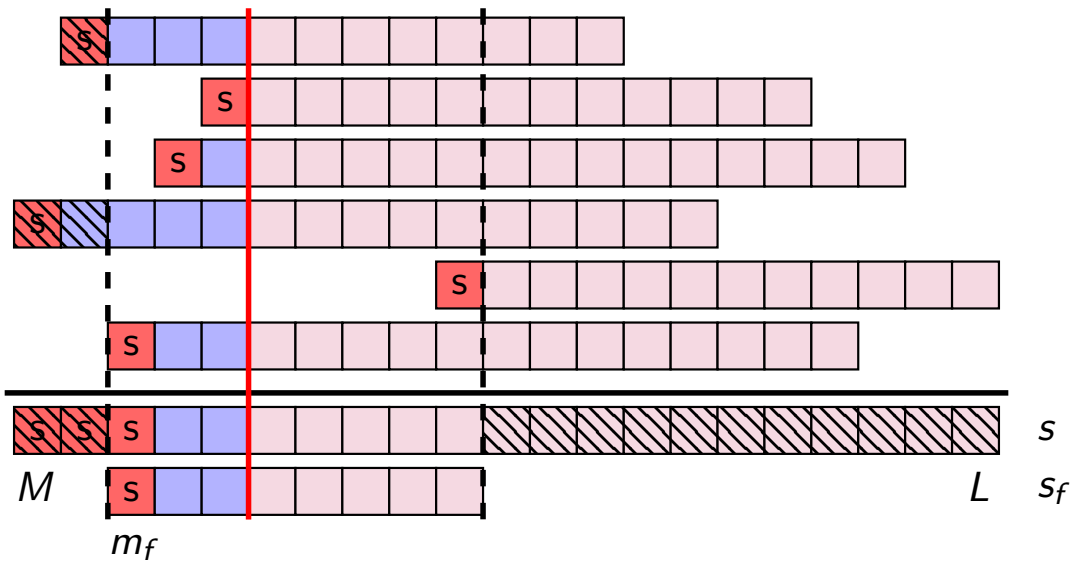
$$s = \bigoplus_{1 \leq i \leq n}^{m_f+1} \left(\sum_{j=L}^{m_f} 2^j p_{i,j} \right)$$



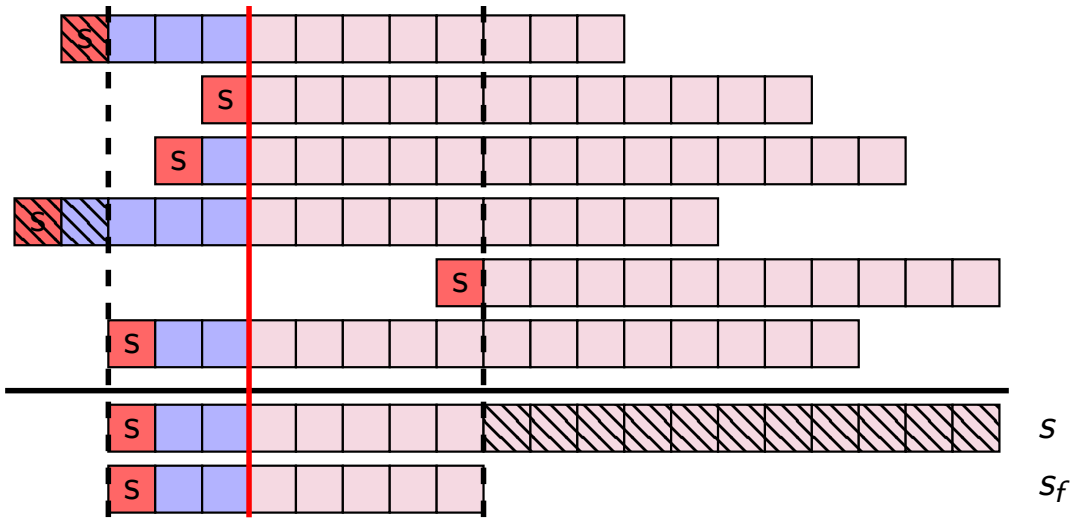
Fixed-Point Jacskon's Rule

Let s be a sum of n fixed-point number p_is, in format (M, L). If s is known to have a final MSB equals to m_f with $m_f < M$, then:

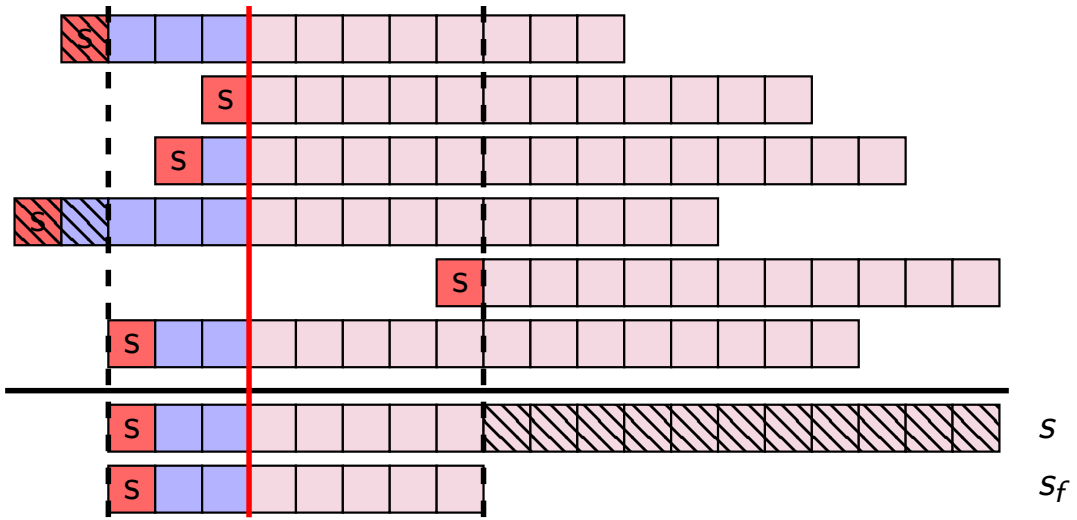
$$s = \bigoplus_{1 \leq i \leq n}^{m_f+1} \left(\sum_{j=L}^{m_f} 2^j p_{i,j} \right)$$



LSB Formatting main idea

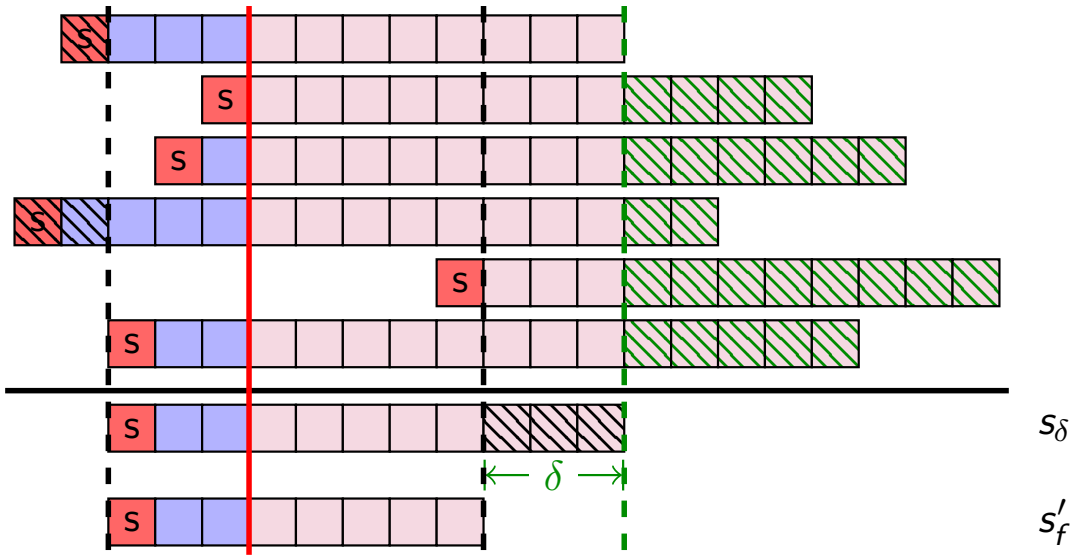


LSB Formatting main idea



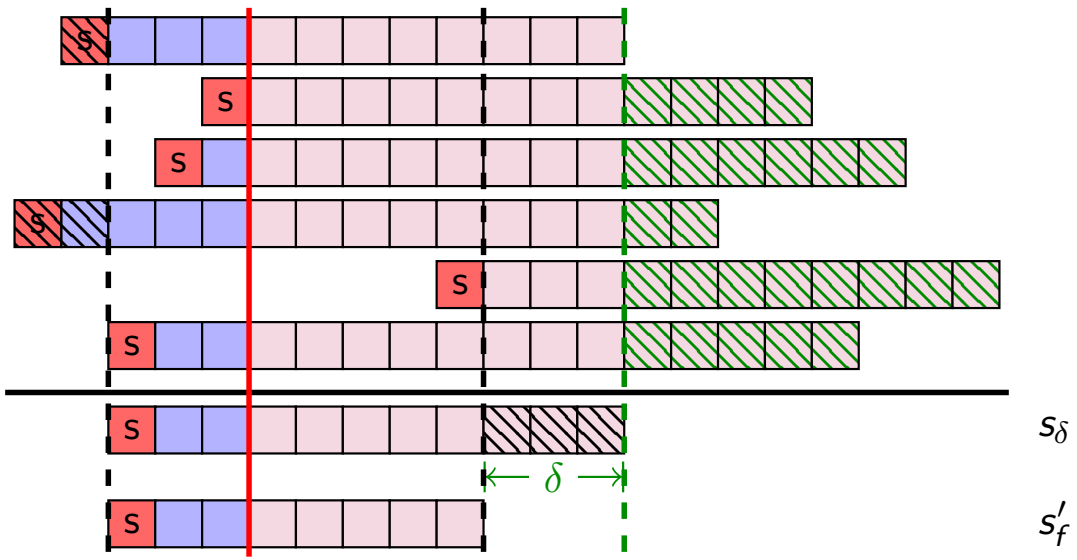
If we remove some bits, we will not compute s_f anymore but a faithful round-off of s_f can be acceptable.

LSB Formatting main idea



Can we determine a minimal δ such that s'_f is always a faithful round-off of s_f ?

LSB Formatting main idea

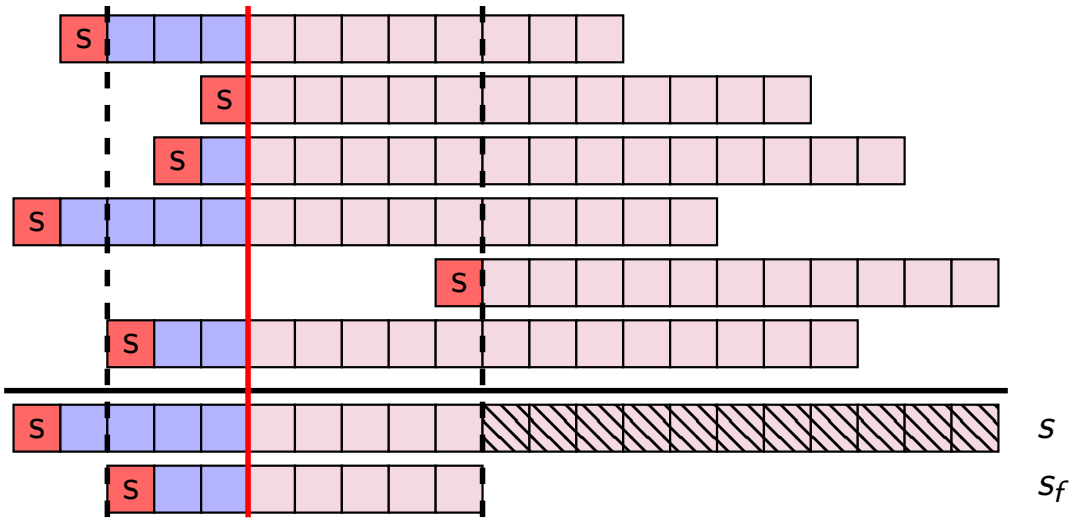


δ evaluation

For both rounding mode (round-to-nearest or truncation), the smallest integer δ that provides $s'_f = \star_{l_f}(s_f)$ is given by:

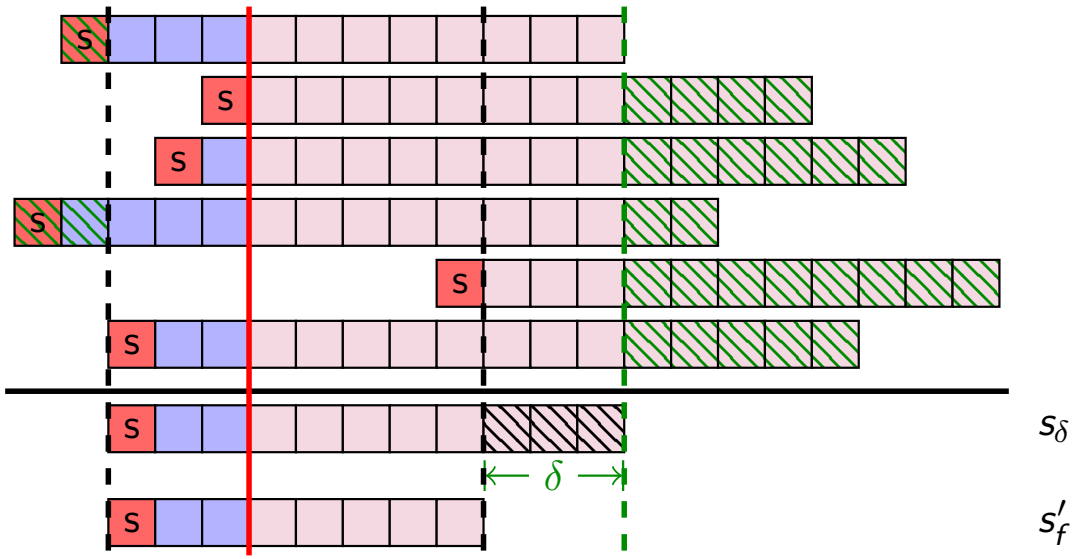
$$\delta = \lceil \log_2(n) \rceil$$

Formatting method



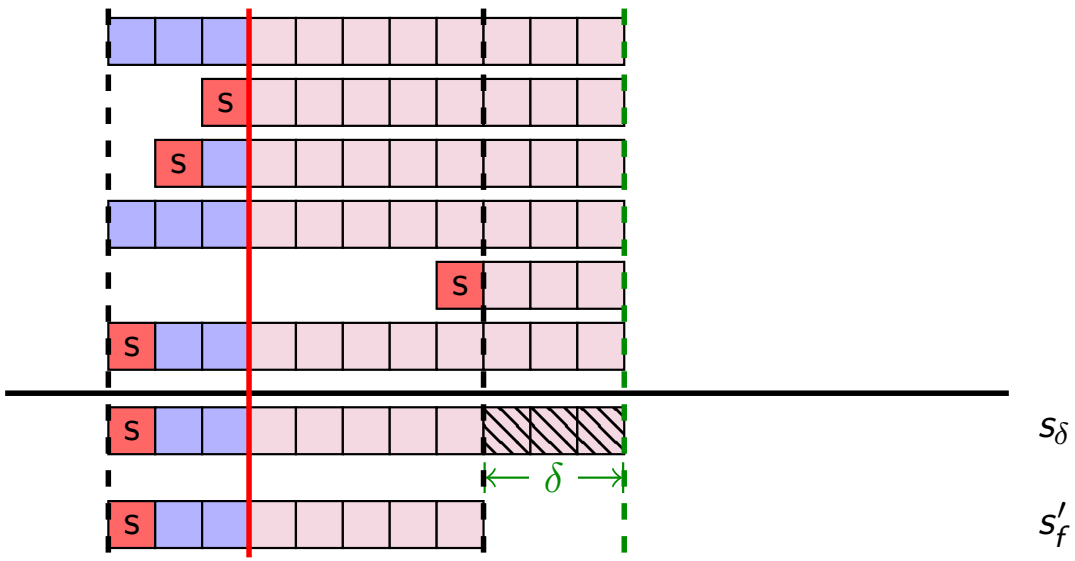
1. we compute δ

Formatting method



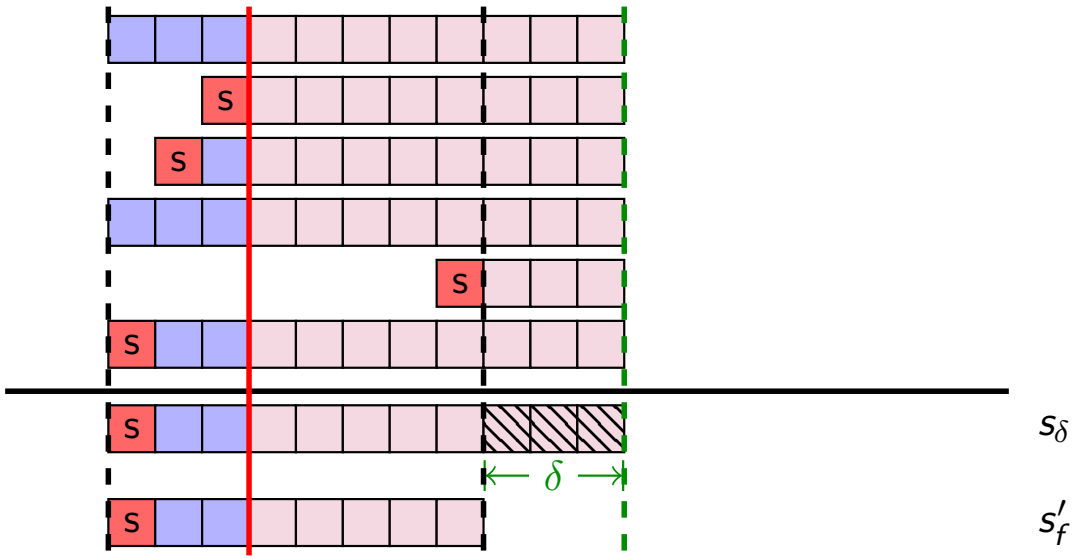
1. we compute δ
2. we format all p_is into FPF $(m_f, l_f - \delta)$

Formatting method



1. we compute δ
2. we format all p_is into FPF $(m_f, l_f - \delta)$
3. we compute s_δ

Formatting method



1. we compute δ
2. we format all p_is into FPF $(m_f, l_f - \delta)$
3. we compute s_δ
4. we obtain s'_f from s_δ

Example

The following algorithm is the fixed-point algorithm of a $4^{th}-$order butterworth filter:

$$\begin{aligned} y(k) \;=\; & 0.0013279914856\ u(k) + 0.00531196594238\ u(k-1) \\ & +0.00796794891357\ u(k-2) + 0.00531196594238\ u(k-3) \\ & +0.0013279914856\ u(k-4) + 2.87109375\ y(k-1) \\ & -3.20825195312\ y(k-2) + 1.63458251953\ y(k-3) \\ & -0.318710327148\ y(k-4) \end{aligned}$$

Inputs datas :

- wordlength of constants, $u(k)$ and $y(k)$: 16 bits
- $u(k) \in [-13, 13]$

Example

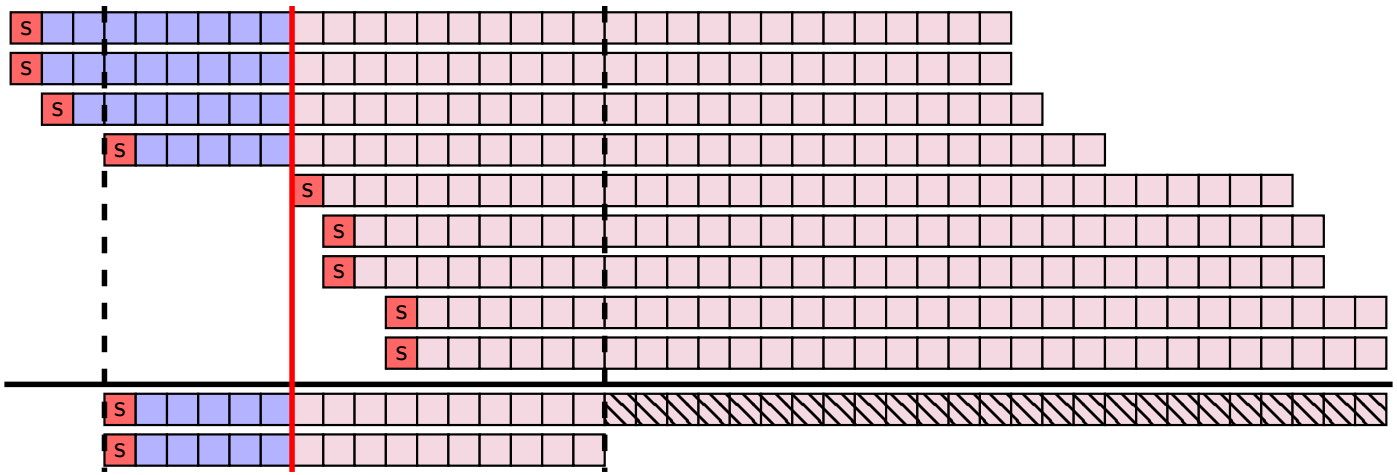
The following algorithm is the fixed-point algorithm of a $4^{th}-$order butterworth filter:

$$\begin{aligned} y(k) \quad = \quad & 0.0013279914856\ u(k) + 0.00531196594238\ u(k-1) \\ & +0.00796794891357\ u(k-2) + 0.00531196594238\ u(k-3) \\ & +0.0013279914856\ u(k-4) + 2.87109375\ y(k-1) \\ & -3.20825195312\ y(k-2) + 1.63458251953\ y(k-3) \\ & -0.318710327148\ y(k-4) \end{aligned}$$

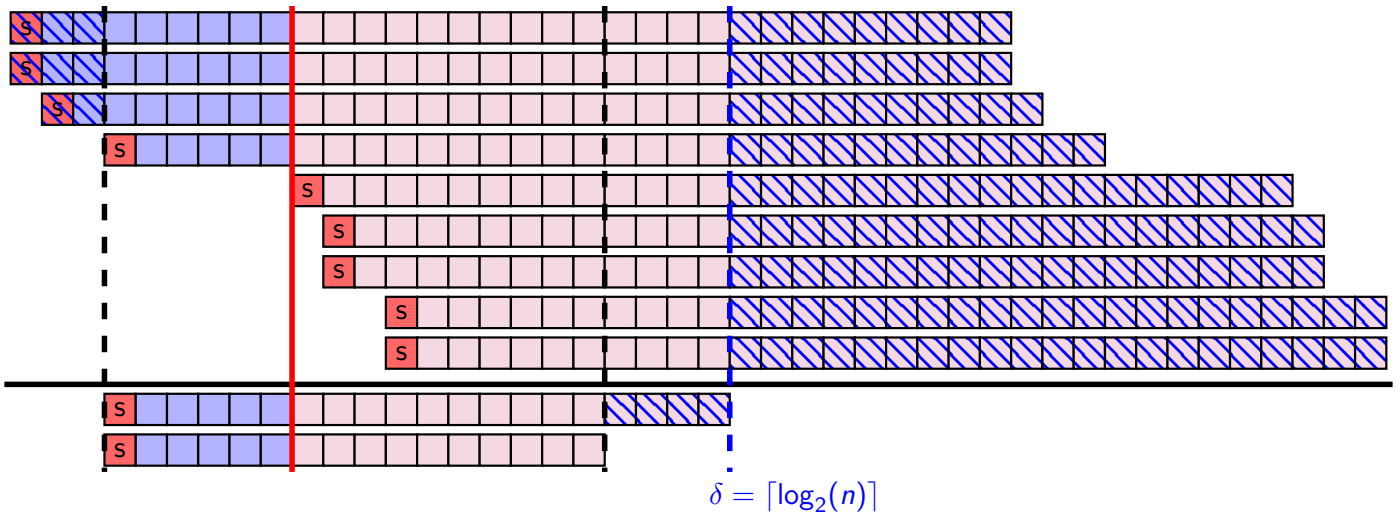
Inputs datas :

- wordlength of constants, $u(k)$ and $y(k)$: 16 bits
- $u(k) \in [-13, 13]$ and $y(k) \in [-17.123541; 17.123541]$

Example



Example



Conclusion

We try to answer the following question :

For a given sum-of-products, how to reduce the number of bits of the operands while controlling the error ?

For this, some works have been realized:

- FiPoGen : a tool generating fixed-point code for a given sum-of-products
- Bits formatting : a first step towards word-length optimization

THANK YOU

Any questions?