DSP platforms: fixed-point vs floating-point
Integer MAC unit
Neural network quantization & MAC

Signal Processing Systems Fall 2025

Lecture 3 (Monday 3.11.)

Outline

- Comparison of fixed-point and floating-point DSP
- Integer MAC unit & FIR filtering
  - Coefficient quantization
  - Word lengths in MAC
- Quantized neural networks
  - Approaches for efficient NN implementation
  - Design Task 3 Problem 3 introduction
    - MAC-based NN implementation
    - Asymmetric quantization of coefficients
    - Energy-efficiency: fixed-point vs floating-point

Saturation & rounding

- See slides 42-54 of Lecture 2
- Main points:
  - Both have reducing impact on word lengths
  - Saturation protects from wrap-up errors
    - Non-associative: final result may depend on the order of arithmetic operations
  - Rounding
    - 3 directive techniques (e.g. truncation), 3 round to nearest (slight bias differences)
    - In algorithm analysis, modelling as a uniformly distributed random variable
    - Correlations of rounding errors can cause problems in filtering, for example
    - Floating-point constants written to code are typically not represented exactly

I. DSP platforms: fixed-point vs floating-point

Mapping applications to DSP/ASP platforms

- Taking a look at the papers
  - J. Janhunen, T. Pitkänen, O. Silvén, M. Juntti (2011) *Fixed- and floating-point processor comparison for MIMO-OFDM detector.* IEEE Journal of Selected Topics in Signal Processing 5: 1588-1598. PAPER-1
  - D. Guenther, R. Leupers, G.Ascheid (2013) *Mapping of MIMO receiver algorithms onto application-specific multi-core platforms*. In: The Tenth International Symposium on Wireless Communication Systems. PAPER-2A
  - D. Guenther, A. Bytyn, R. Leupers, G.Ascheid (2014) *Energy-efficiency of floating-point and fixed-point SIMD cores for MIMO processing systems*. In: 2014 International Symposium on System-on-Chip (SoC). PAPER-2B
- Papers consider implementation of communications baseband processing, MIMO-OFDM detection
  - Enabling Software-Defined Radio (SDR) platforms, where implementation of multiple standards is done in SW on fixed HW
  - Comparisons of fixed-point and floating-point format efficiency
  - Provide examples on how algorithms are mapped to implementation

multiple-input and multiple-output (**MIMO**) is a method for multiplying the capacity of a radio link using multiple transmission and receiving antennas to exploit multipath propagation

orthogonal frequency-division multiplexing (**OFDM**) is a method of encoding digital data on multiple carrier frequencies

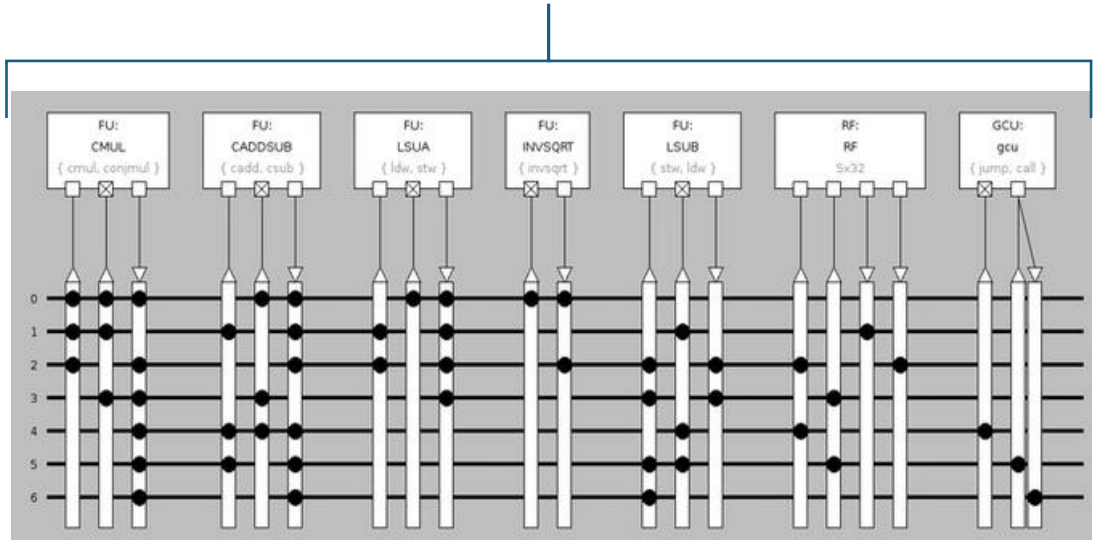
PAPER-1 (Janhunen et al. 2011)

- Considers implementation of MIMO-OFDM on the transport triggered architecture (TTA)

- Complexity and efficiency of **16-bit fixed-point and 12-bit floating-point solutions are compared** in the paper

- What is TTA?
  - TTA is a processor design, which is composed of a set of functional units (FU) and their interconnections
  - FUs implement needed arithmetics and HW accelerations
  - Programs (SW) control the internal data transport between FUs
  - Transport action triggers computation implemented in FU
  - Several parallel transports every cycle
  - HW/SW codesign

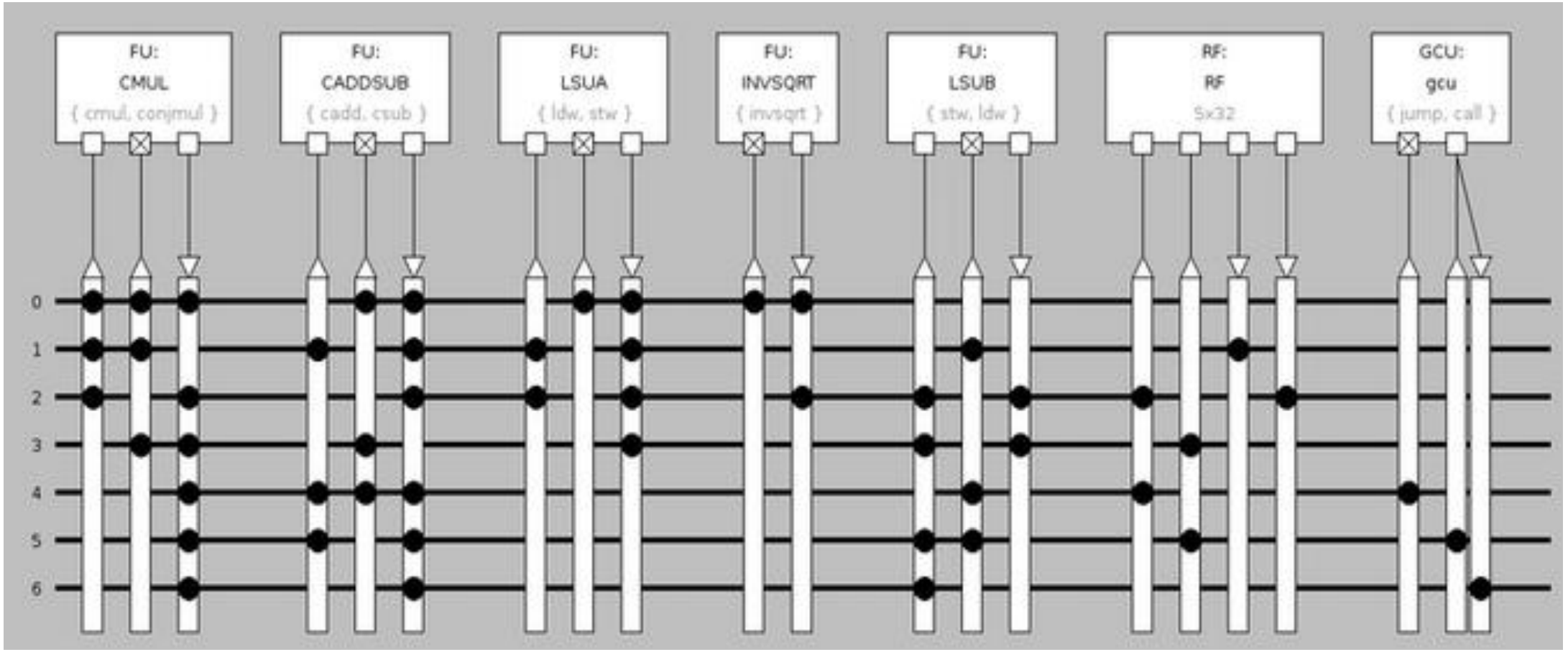
Application-specific processor (ASP)
Functional units, application-based instantiation from a library



Unit interconnections : instructions of the SW code activate

TTA based ASSP example

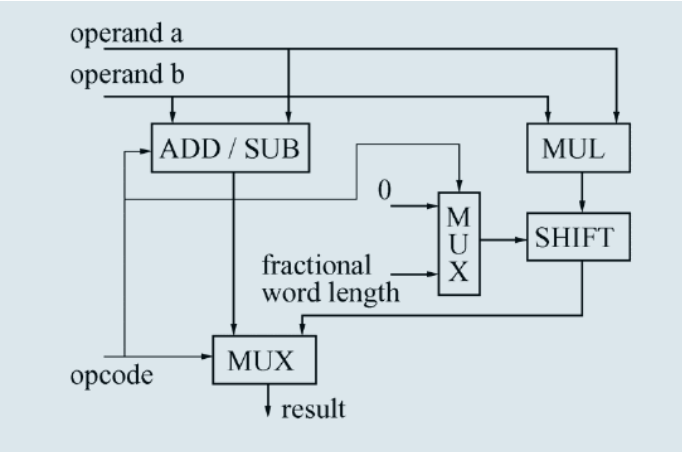
Functional units, application-based instantiation from a library



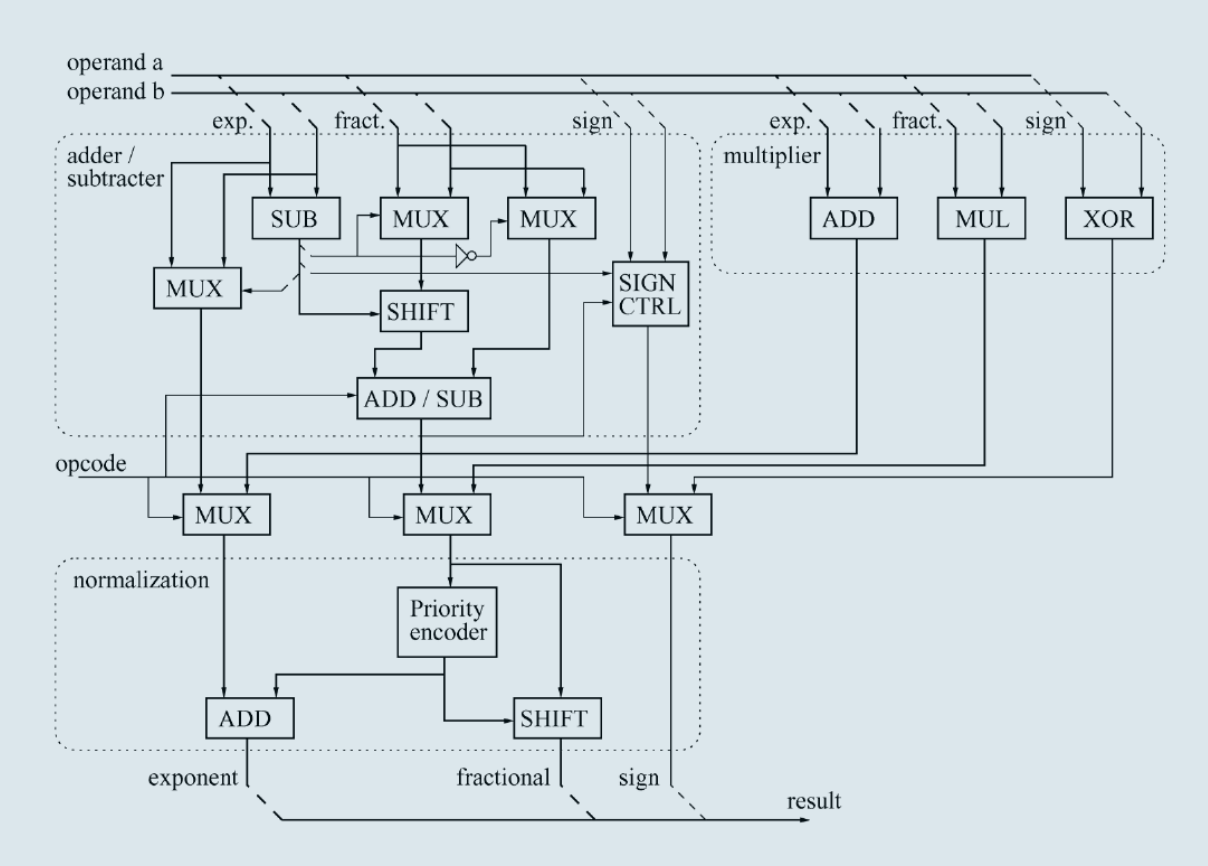
Unit interconnections : instructions of the SW code activate transfer.
Based on specified transfers, FUs are activated

Complexity and efficiency of **16-bit fixed-point and 12-bit floating-point solutions are compared** in the paper

Difference in the complexity of these ALUs is large. However, simplicity of basic fixed-point computation must be compensated by other structures in the HW or extra operations in the SW, when we have a high-dynamic-range application



A fixed-point ALU



A floating-point ALU

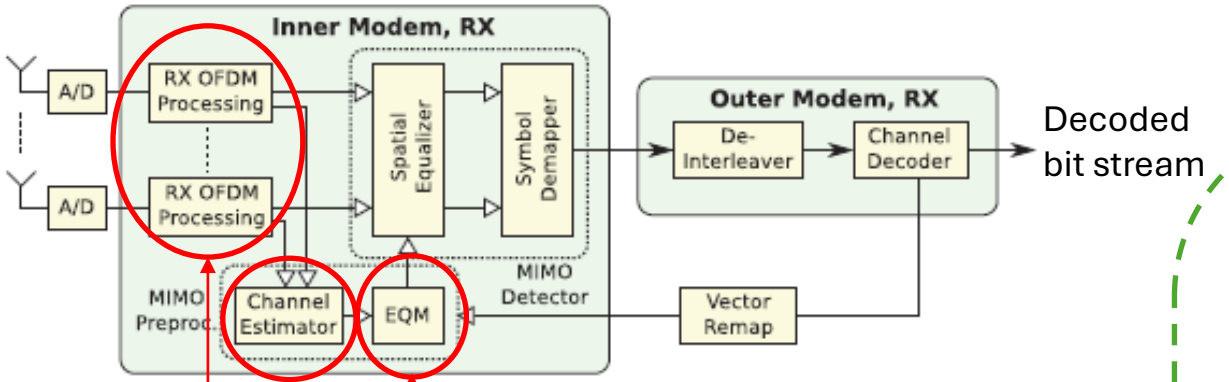
RESULT: Performances found to be quite similar.
This favors taking a custom floating-point solution. Provides tool chain benefits: easier mapping of the algorithm to the implementation

Diagrams from Janhunen et al (2011)

PAPER-2A (Guenther et al 2013)

- Paper contents:
  - MIMO receiver algorithms
  - Identification of processing primitives from algorithms
  - Implementing those primitives on 3 target platforms
    - Two fixed-point integer arithmetic chips: P2012 by STMicroelectronics, TMS320C6474 by Texas Instruments
    - Authors' own floating-point processor, supports complex-valued arithmetic
  - Comparison of these: cycle counts (throughput)
  - Energy aware mapping on research core
    - **Adaptation of bit precision** (number of significand bits) => reduction of switching activity => reduction in energy consumption

MIMO RECEIVER BLOCK DIAGRAM



ALGORITHMS

FFT

equalizer matrix

$$\mathbf{G} = \left(\hat{\mathbf{H}}^H \hat{\mathbf{H}} + N_0 \mathbf{I}_{N_t}\right)^{-1} \hat{\mathbf{H}}^H$$

$$\hat{\mathbf{H}} = \frac{1}{N_t + N_0} \mathbf{S}\mathbf{P}^H$$

Estimate of transmitted symbols

$$\hat{\mathbf{x}} = \mathbf{G}\mathbf{y}$$

etc.

IDENTIFIED COMPUTATIONAL PRIMITIVES

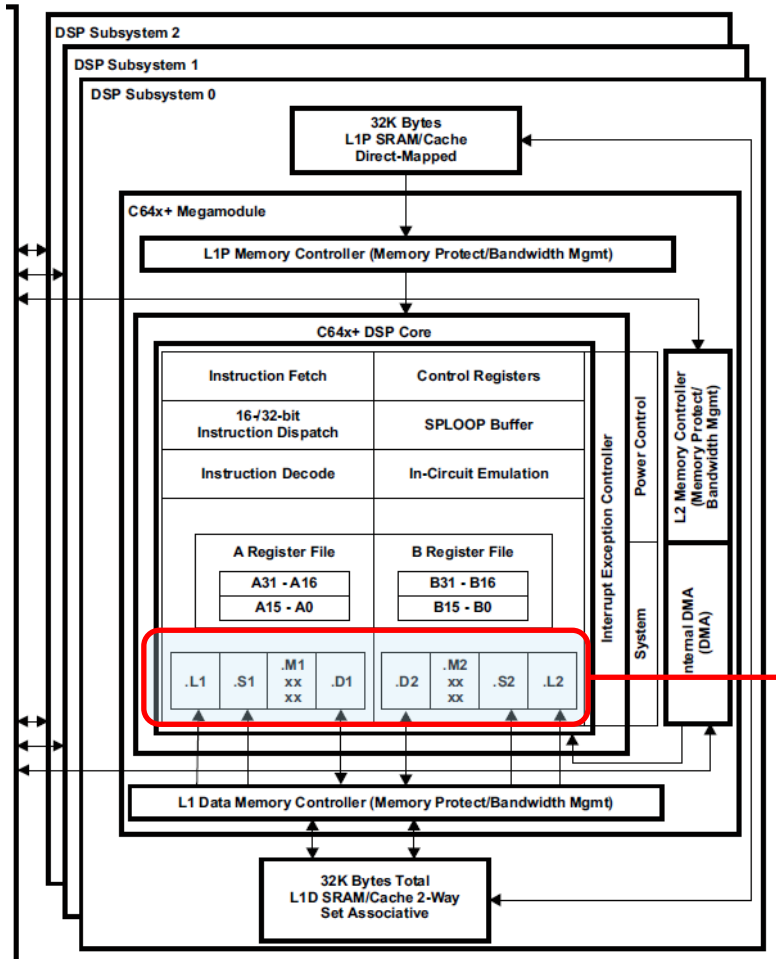
Nucleus	Parameters	Used in
Matrix-matrix mul.	dim, conj	ls-chest, mmse-chest
Matrix-vector mul.	dim, conj	lmmse-eq, lmmse-eq-it
Matrix-scalar mul.	dim	mmse-chest
Vector-scalar mul.	dim	lmmse-eq-it, sinr-it
Vector-vector add.	dim	lmmse-eq-it
Inner product	dim, conj	lmmse-eq-it, sinr-it
Matrix inversion	dim	lmmse-eq, lmmse-eq-it
p/w linear function	regions	demap, svec-remap, var
FFT/iFFT	dim	mod
Scalar inversion	none	mmse-chest, lmmse-eq-it, sinr, sinr-it
Channel coding	codeT, poly	fec

Note. Primitives are independent of the number format at this abstraction level. Some platform independent parameters defined.

The techniques used to implement primitives depend on the platform.

Implementation of primitives

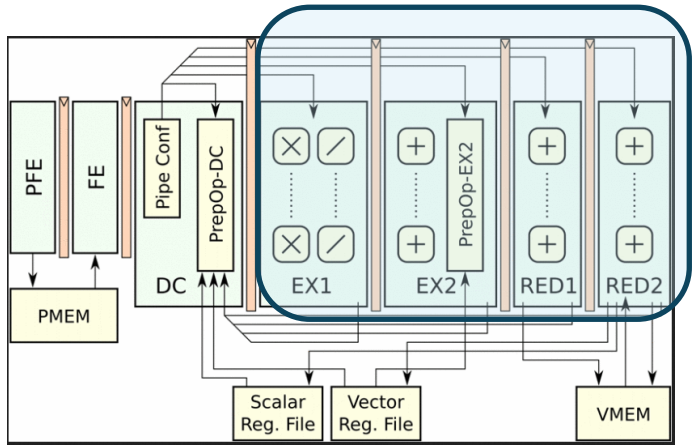
- Fixed-point case
  - High throughput demands => 16-bit data types
  - Matrix inversions must be based on numerically stable algorithms
    - Using Gram-Schmidt QR decomposition
- Floating-point case
  - High dynamic range of numbers
  - Matrix inversion can be computed without decomposition
    - Using divide-and-conquer (DnQ) algorithm (matrix inversion lemma)



functional units
for arithmetics

Functional blocks of three
C64x+ cores in TMS320C6474.

arithmetic operations



Guenther et al. (2013)
Floating-point SIMD processor
core pipeline (7 stages).

SIMD = single instruction multiple data

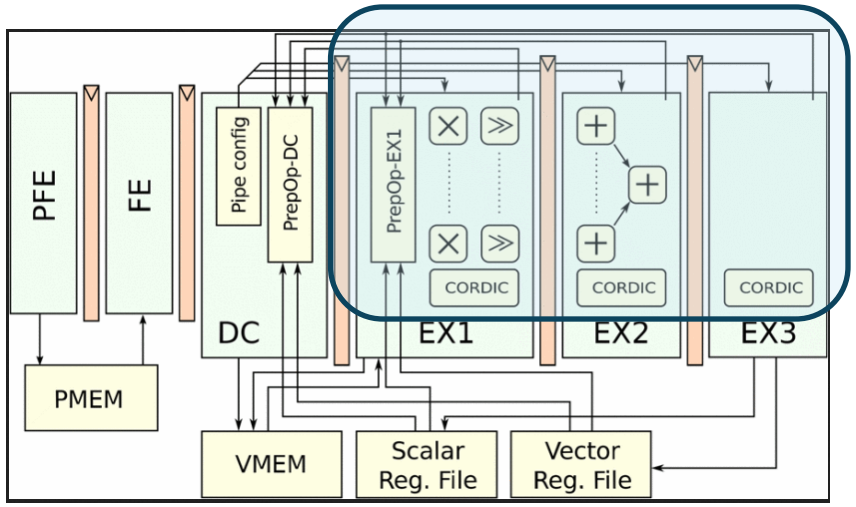
Same operation done for several data items

Items K and K+N at different stages of pipeline
at some time instant

PAPER-2B (Guenther et al 2014)

- Implementation of **fixed-point SIMD core** and comparison to previous floating-point version
  - More fair comparison than in PAPER-2A
  - Experiments with various matrix factorization methods for fixed-point matrix inversion
  - For both cores bit precision (word length, size of significand) is adjusted to find sufficient options
- Results indicate that floating-point is better for algorithms with high dynamic range
  - Fixed-point core delivers better energy efficiency for certain subtasks with little dynamic range

arithmetic operations



Basic arithmetic operations in the pipeline: addition, multiplication, arithmetic shift, CORDIC

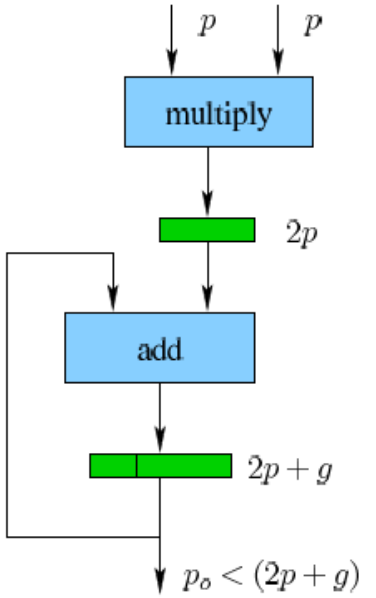
Pipeline has 5-6 stages depending on inclusion of the CORDIC

CORDIC implements Givens rotation, that can be utilized in some matrix factorization techniques.

II. MAC unit

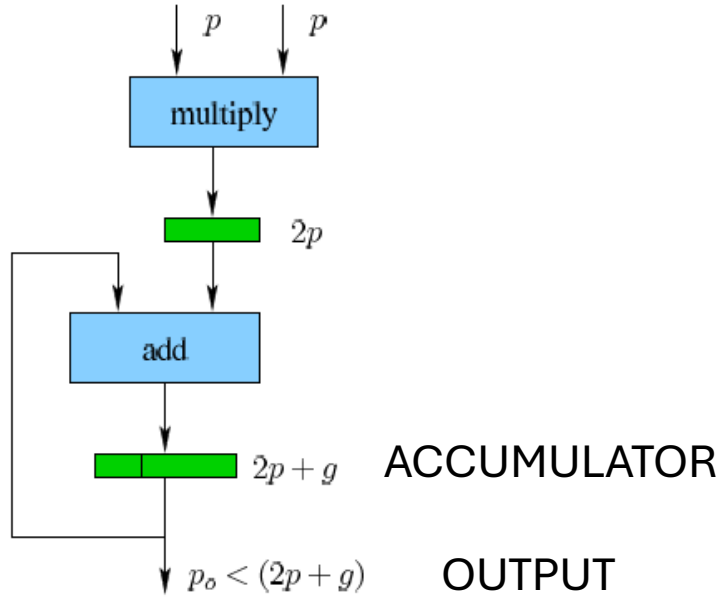
II. Integer multiply-accumulate (MAC) unit

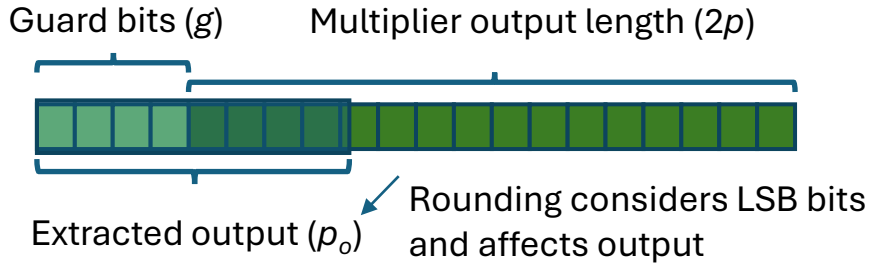
- In DSP, a common operation is to perform multiplication, which is immediately followed by an accumulate operation
- Implemented as a single-cycle operation
- Possible data path of a MAC unit on the right
  - Integer p-bit numbers as multiplier input
  - $2p$-bit multiplier output
  - Accumulator (adder) output with some guard bits (g)
  - Output word length p_o



II. Integer multiply-accumulate (MAC) unit

- In DSP, a common operation is to perform multiplication, which is immediately followed by an accumulate operation
- Implemented as a single-cycle operation
- Possible data path of a MAC unit on the right
  - Integer p-bit numbers as multiplier input
  - $2p$-bit multiplier output
  - Accumulator (adder) output with some guard bits (g)
  - Output word length p_o
- Output word can be based on most significant bits of the accumulator and some rounding operation
  - But, is the range provided by output bits exploited well?
  - Perhaps it should be possible to extract some intermediate portion from the accumulator if all guard bits are not used to hold the final result



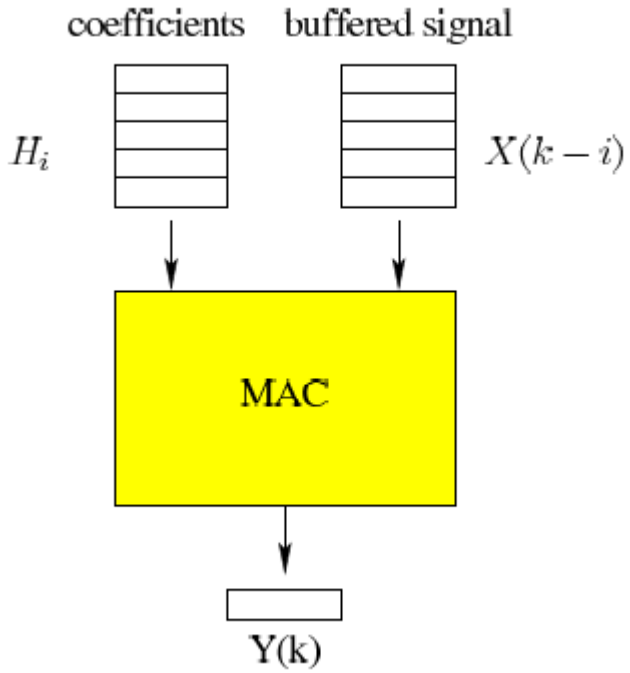


Integer MAC applied to FIR filtering

- FIR (Finite Impulse Response) filter equation:

$$y(k) = \sum_{i=0}^{N-1} h_i x(k-i)$$

- Coefficients mapped to integer values H_i
- Inputs $x(j)$ mapped to integers $X(j)$ (ADC) and buffered
- Computation of $Y(k)$
  1. Reset accumulator to zero
  2. Feed MAC with *N* pairs $(H_i,\ X(k-i))$
  3. Make output $Y(k)$ from accumulator bits



Coefficient quantization

- Integer coefficients are calculated using

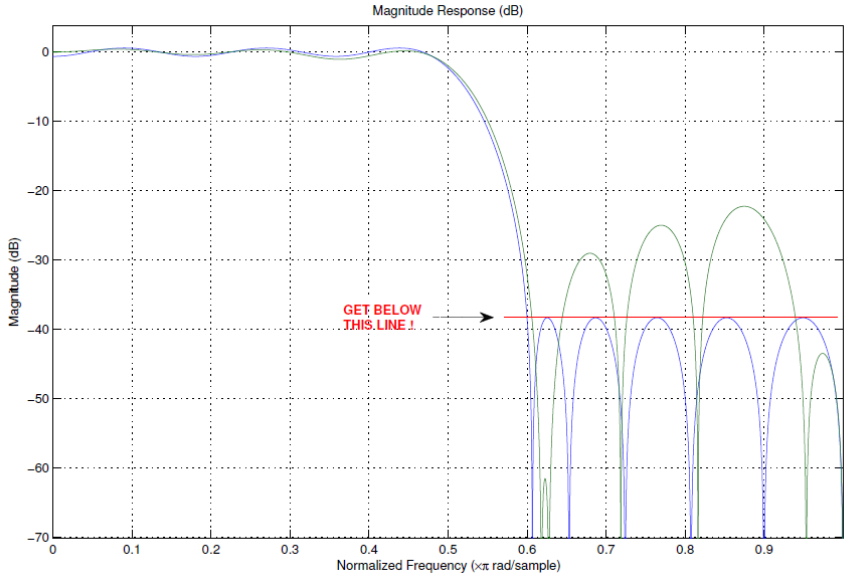
$$H_i = \text{clamp}(\text{round}\left(\frac{h_i}{s}\right)),$$

  where
  - s denotes the slope in slope-bias scaling (bias = 0)
  - round is a rounding-to-nearest operation
  - clamp maps the rounding result to the range of integer format, if necessary
- How the slope s should be chosen?
- Goal: the impact on quality of outputs $Y(k)$ is minimal

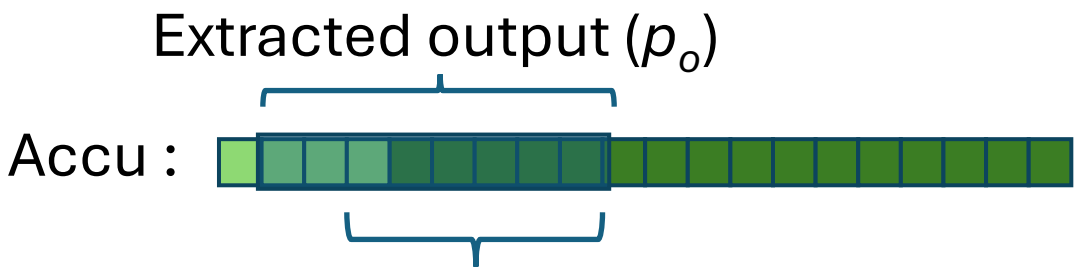
Coefficient quantization

Some ideas to consider:

1. Study frequency response of the quantized filter
2. Keep ratios of original coefficients
3. Utilize range of *p*-bit integers used for coefficients
4. If accumulator is a saturating one, perhaps possibility of clipping can be allowed
5. Maximize utilization of the range of that portion of the accumulator, which will be used to form the output



Preserve $\frac{h_i}{h_j}$ in $\frac{H_i}{H_j}$



Not good if only these bits are affected as higher precision is available!
Having larger H_i values helps

Accumulator word length in MAC

1. NEGLECTING COEFFICIENT VALUES

Accumulator word length p_a must be at least $2p+g_N$, where g_N is computed from the number of coefficients N using

$$g_N = \text{ceil}\ (\ \log_2(N)\ ).$$

2. ANALYSIS BASED ON COEFFICIENT VALUES

The number of bits needed at the accumulator, which is sufficient for avoiding overflows is

$$p_a = \lceil \log_2(\text{RANGE}) \rceil + 1$$

where

$$\text{RANGE} = 2^{p-1} \sum_{i=0}^{N-1} |H_i|.$$

RANGE provided by this formula is approximately the largest value that the accumulator can have.

Exact calculation of RANGE:

Considering capability of the accumulator, what is the worst input signal X(k)?

It is either

1. PMAX for each positive coefficient and NMAX for each negative coefficient
2. PMAX for each negative coefficient and NMAX for each positive coefficient

where

PMAX = positive maximum signal
NMAX = negative maximum (magnitude)

Quick test

$$p_a = \lceil \log_2(\text{RANGE}) \rceil + 1$$

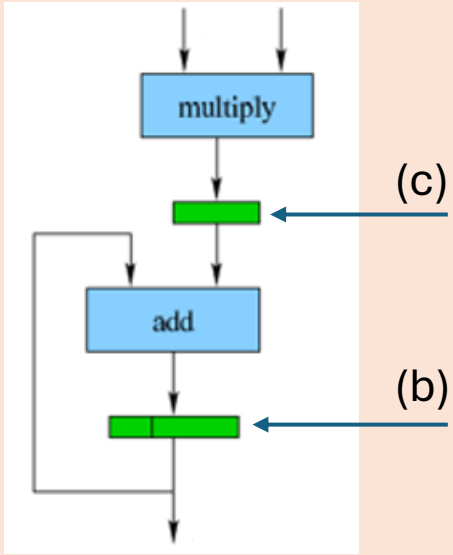
$$\text{RANGE} = 2^{p-1} \sum_{i=0}^{N-1} |H_i|.$$

Assume that a filter has five coefficients +2, -4, +11, -4, +2 and the signal input has 6 bits.

(a) What is the range of filter accumulator outputs?

(b) What is the minimum word length for an accumulator so that there cannot be any overflows?

(c) What about multiplier output, how many bits are sufficient for it?



Goto **presemo.oulu.fi/spslec3** and provide your solutions

III. Neural network quantization

Introduction

Neural networks (NN), especially deep neural networks (DNN), may contain huge number of parameters.

For memory-constrained platforms with limited computational capabilities (e.g., only fixed-point integer arithmetic available), one must carefully consider how DNN is implemented.

Approaches:

- **Post-training quantization (PTQ)**. Trained NN, which works with floating-point numbers, is given. The problem is to convert it to a solution, which uses integer arithmetic.

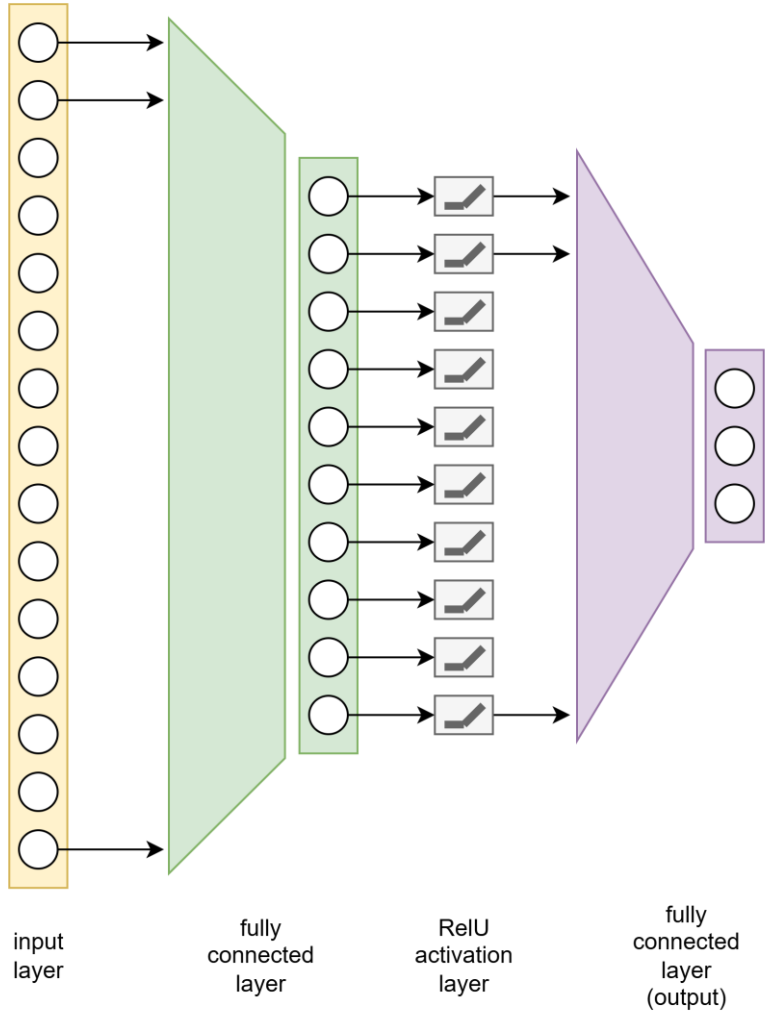
- **Quantization-aware training (QAT)**. Training of NN takes into account use of integer arithmetic in final application. Performance might be higher than with PTQ solution.

- **Knowledge distillation**. Trained NN (so-called teacher model) exists. It is used for training another simpler NN (so-called student model). The goal is to make student model mimic the behavior of teacher model.

NN implementation using integer arithmetic

- **Task**. NN with floating point parameters has been provided to you and you are asked to implement calculations on a platform, which supports just integer arithmetic. How to do it?
  - This is a **PTQ** (post-training quantization) problem

- **Design Task 1 Problem 3**. Let us consider a simple network shown on the right
  - Three layers of computation
  - Fully connected layers use MAC based computation and have parameters (weights & biases) that have been adjusted during training



How NN works?

- Fully connected layers perform an operation of the form

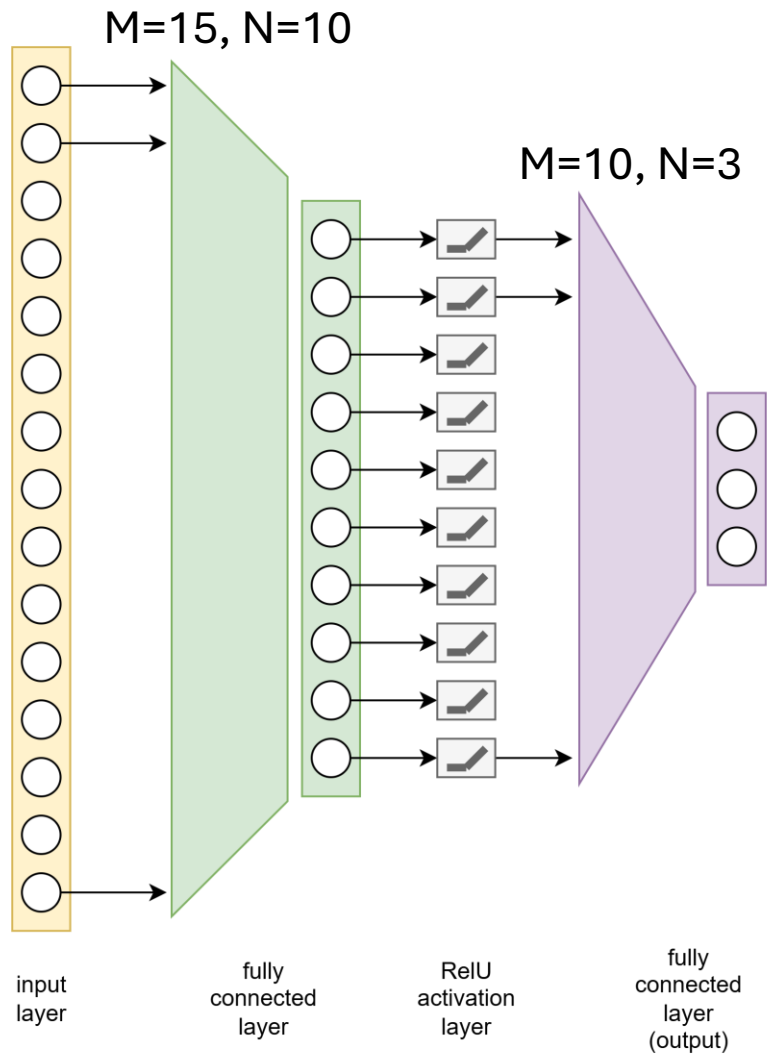
$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where

$\mathbf{x}$ is M-element input vector
$\mathbf{y}$ is N-element output vector
$\mathbf{W}$ is N x M weight matrix
$\mathbf{b}$ is N-element bias* vector

- RelU activation layer performs for each input x operation

$$y = \max(0, x)$$



To implement FC layers, we need some MAC hardware.
Parallel computation possible depending on the number of MAC units.
RelU needs just a check of sign bit and forwarding zero, if the bit is on.

*NOTE: Not bias of slope-bias scaling!

Fully-connected layer

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1M} \\ w_{21} & w_{22} & \cdots & w_{2M} \\ \vdots & \vdots & & \vdots \\ w_{N1} & w_{N2} & \cdots & w_{NM} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_M \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

NM multiplications
NM additions

Computation of single output item

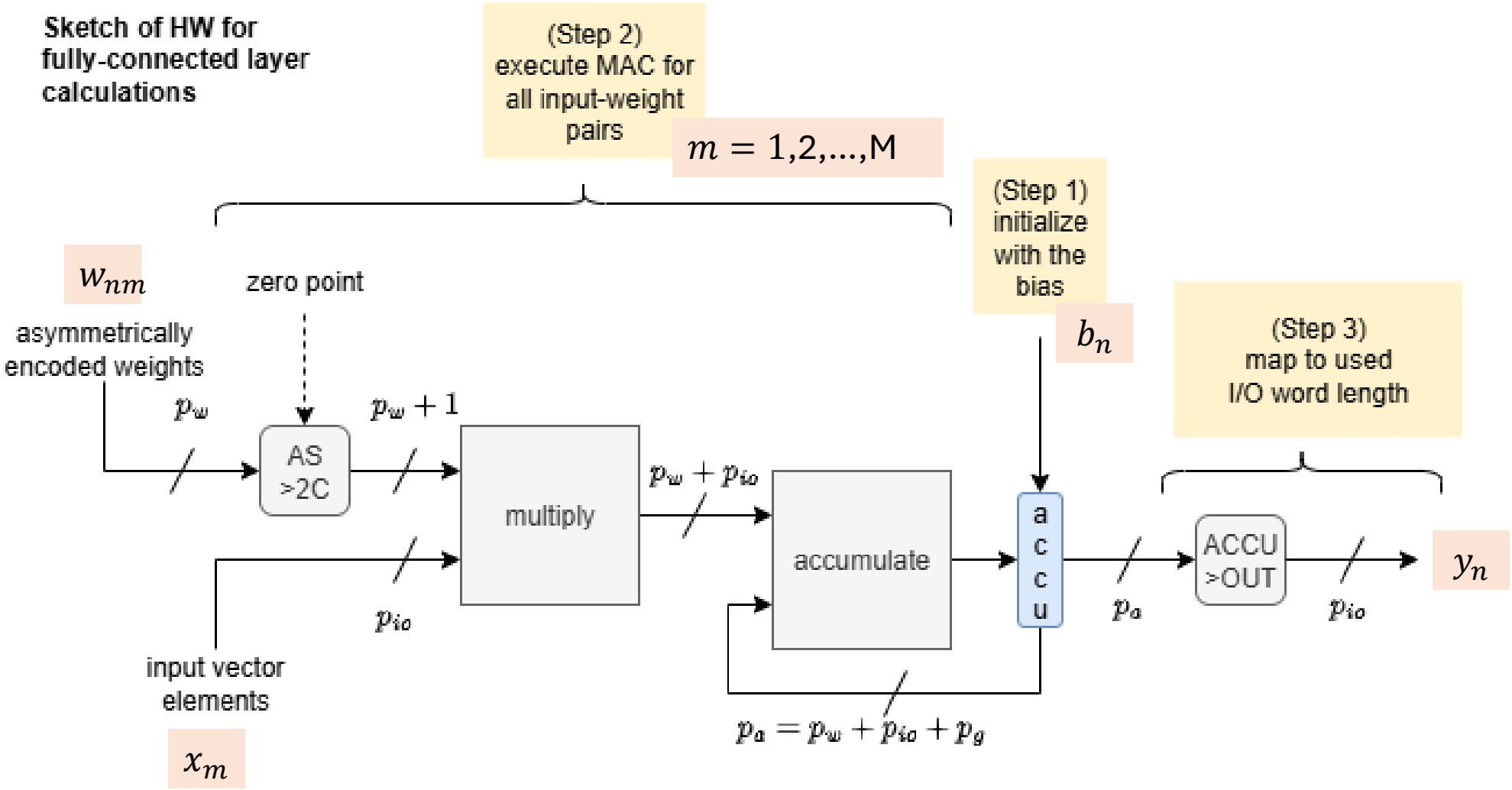
$$y_n = \sum_{m=1}^{M} w_{nm} x_m + b_n$$

Implementation using MAC computation …

MAC for fully connected layer

Executed for each $n = 1,2,\ldots,N$

$$y_n = \sum_{m=1}^{M} w_{nm} x_m + b_n$$



MAC for fully connected layer

$$y_n = \sum_{m=1}^{M} w_{nm} x_m + b_n$$

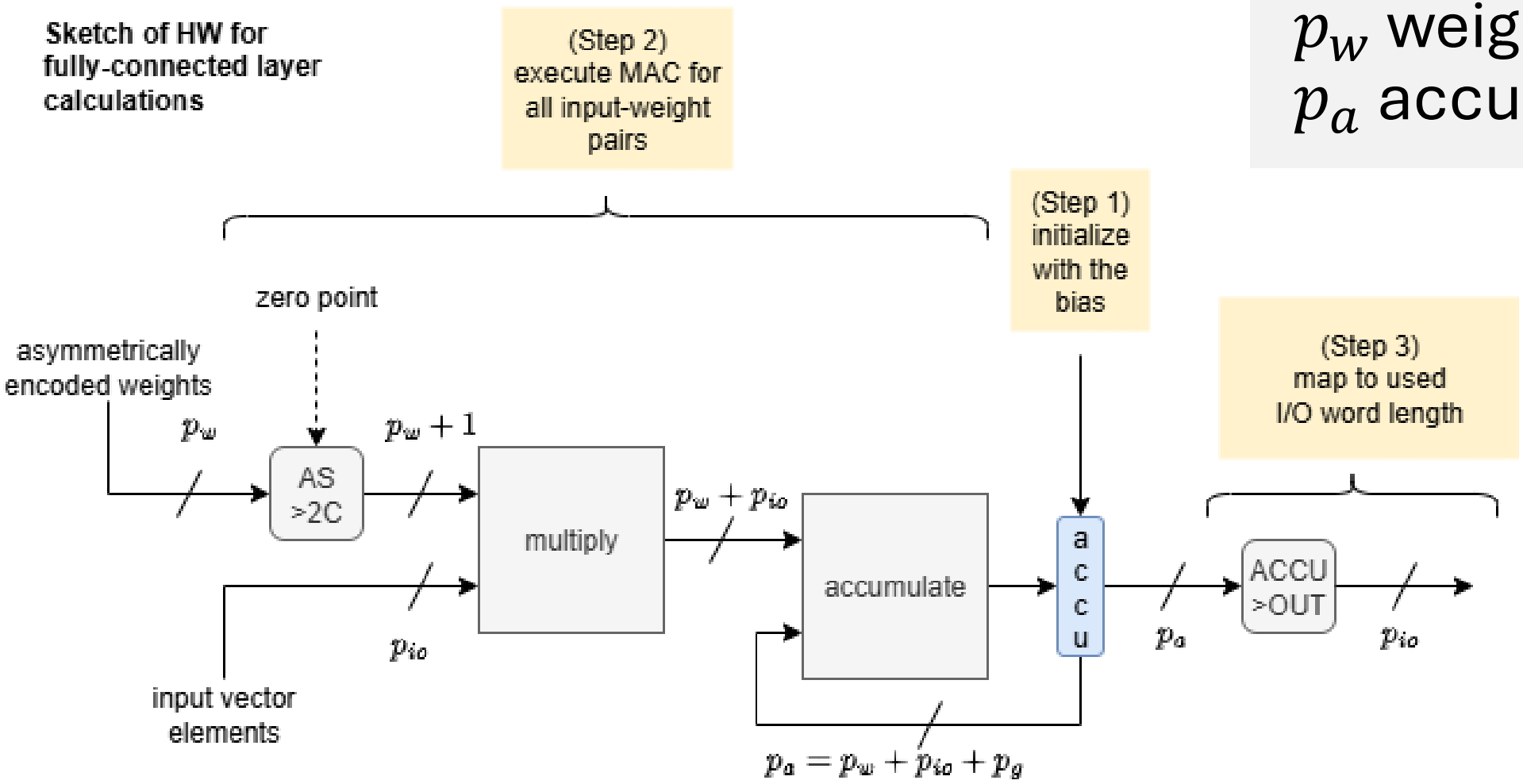
Word lengths

p_{io} signal i/o

p_w weights

p_a accumulator

Sketch of HW for fully-connected layer calculations

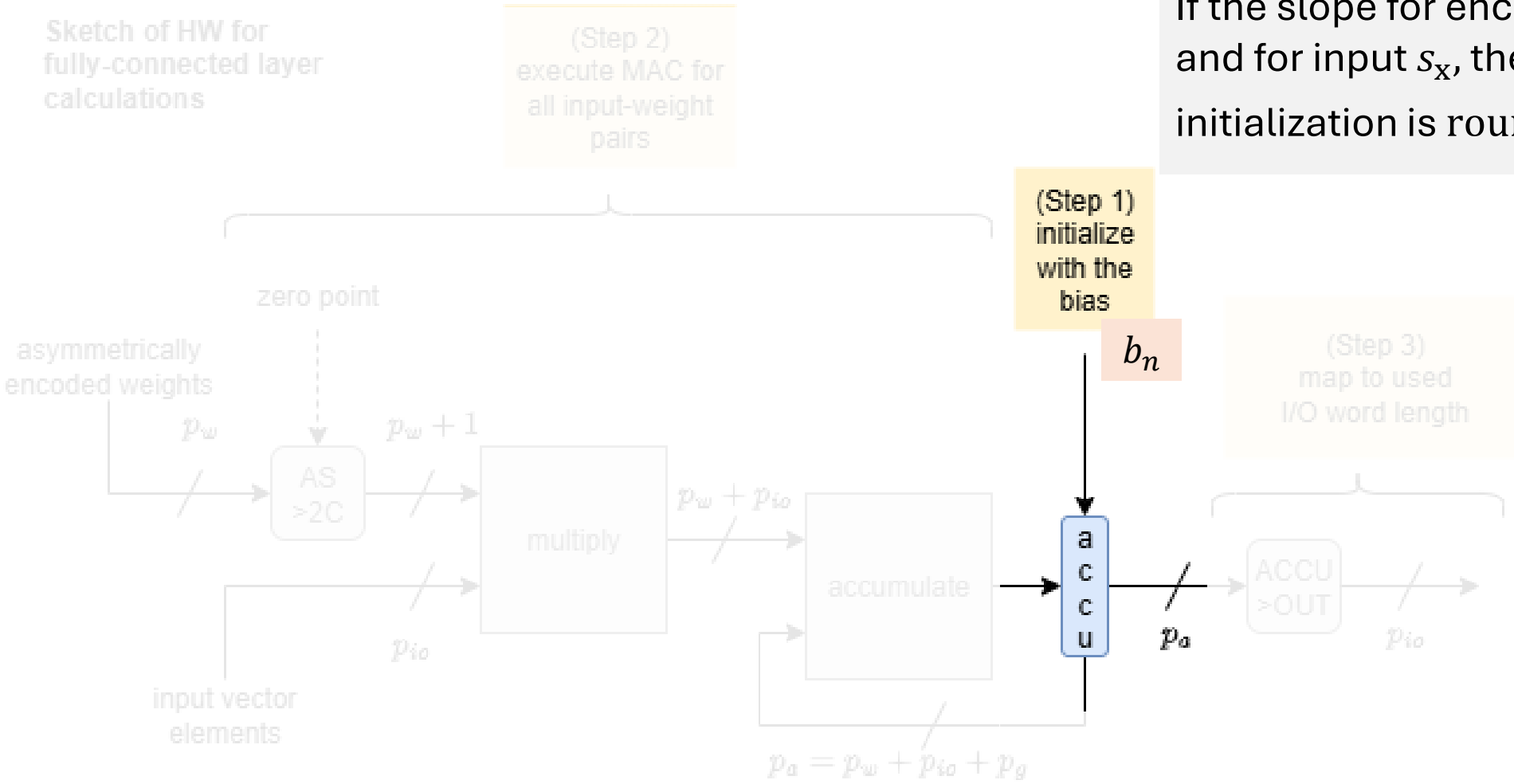


MAC for FC layer, step 1

$$y_n = \sum_{m=1}^{M} w_{nm} x_m + b_n$$

Accumulator is initialized with a signed integer, that corresponds to b_n

If the slope for encoded weights is s_w and for input s_x, then the integer for initialization is round $\left(\frac{b_n}{s_w s_x}\right)$

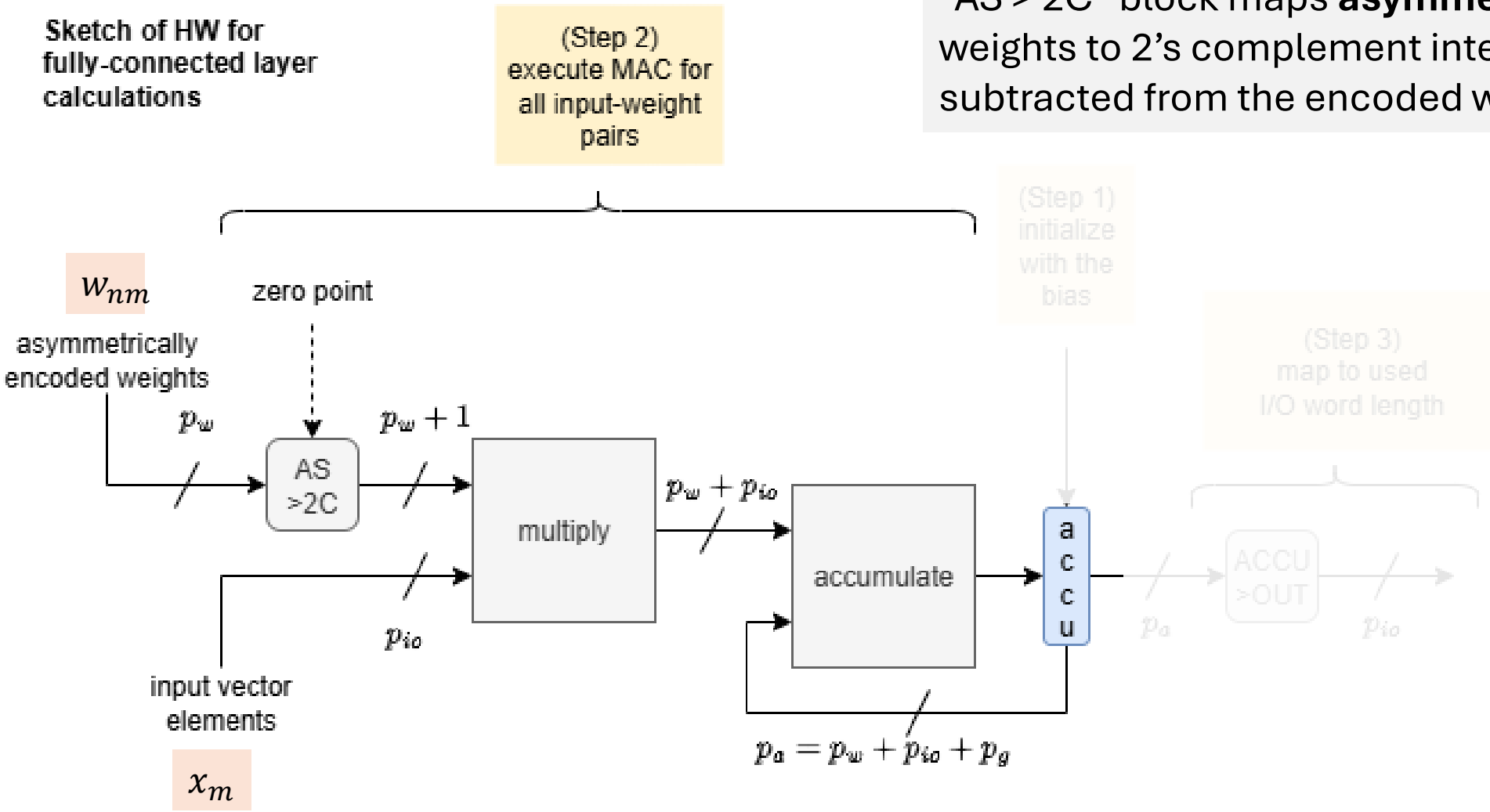


MAC for FC layer, step 2

$$y_n = \sum_{m=1}^{M} w_{nm} x_m + b_n$$

MAC operation is done for each pair (w_{nm}, x_m)

"AS > 2C" block maps **asymmetrically encoded** weights to 2's complement integers: zero-point subtracted from the encoded weight



Ex. Asymmetric quantization (1)

Uniform affine quantization

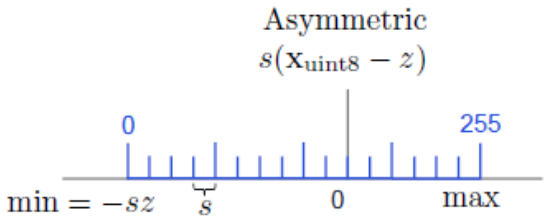
Assume that NN layer weights are in the range -0.8 … +1.2, and we want to save weights using 8 bit integers.

Moreover, we decide to use asymmetric quantization to take advantage of the full range of 8-bit values.

We use the encoding explained on the right and must define three parameters: b = 8, but how we select s and z?

Note that zero-point z must be an integer!

We are specifying a quantization grid



Uniform affine quantization, also known as *asymmetric quantization*, is defined by three quantization parameters: the *scale factor* s, the *zero-point* z and the *bit-width* b. The scale factor and the zero-point are used to to map a floating point value to the integer grid, whose size depends on the bit-width. The scale factor is commonly represented as a floating-point number and specifies the *step-size* of the quantizer. The zero-point is an integer that ensures that real zero is quantized without error. This is important to ensure that common operations like zero padding or ReLU do not induce quantization error.

Once the three quantization parameters are defined we can proceed with the quantization operation. Starting from a real-valued vector $\mathbf{x}$ we first map it to the *unsigned* integer grid $\{0, \dots, 2^b - 1\}$:

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right), \tag{4}$$

where $\lfloor \cdot \rceil$ is the round-to-nearest operator and clamping is defined as:

$$\text{clamp}\,(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c. \end{cases} \tag{5}$$

To approximate the real-valued input $\mathbf{x}$ we perfrom a *de-quantization* step:

$$\mathbf{x} \approx \widehat{\mathbf{x}} = s\,(\mathbf{x}_{\text{int}} - z) \tag{6}$$

Combining the two steps above we can provide a general definition for the *quantization function*, $q(\cdot)$, as:

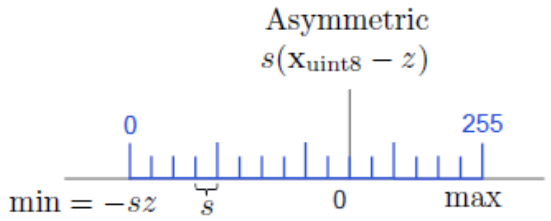
$$\widehat{\mathbf{x}} = q(\mathbf{x}; s, z, b) = s\left[\text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right) - z\right], \tag{7}$$

Through the de-quantization step, we can also define the quantization grid limits $(q_{\text{min}}, q_{\text{max}})$ where $q_{\text{min}} = -sz$ and $q_{\text{max}} = s(2^b - 1 - z)$. Any values of $\mathbf{x}$ that lie outside this range will be clipped to its limits, incurring a *clipping error*. If we want to reduce the clipping error we can expand the quantization range by increasing the scale factor. However, increasing the scale factor leads to increased *rounding error* as the rounding error lies in the range $\left[-\frac{1}{2}s, \frac{1}{2}s\right]$. In section 3.1, we explore in more detail how to choose the quantization parameters to achieve the right trade-off between clipping and rounding errors.

[Nagel et al, 2021]

Markus Nagel et al. A White Paper on Neural Network Quantization. 2021. arXiv: 2106.08295 [cs.LG].

Ex. Asymmetric quantization (2)



Let us make a simple selection, min is -0.8 and max is +1.2

$$-0.7 = -sz$$

$$+1.2 = s(255 - z)$$

=>

$$+1.2 = 255s - 0.7$$

=>

$$s = \frac{2.0}{255}$$

=>

$$z = 0.8 \times \frac{255}{2.0} = 102$$

With these parameters, weights are encoded using Equation 4.

Uniform affine quantization, also known as *asymmetric quantization*, is defined by three quantization parameters: the *scale factor* s, the *zero-point* z and the *bit-width* b. The scale factor and the zero-point are used to to map a floating point value to the integer grid, whose size depends on the bit-width. The scale factor is commonly represented as a floating-point number and specifies the *step-size* of the quantizer. The zero-point is an integer that ensures that real zero is quantized without error. This is important to ensure that common operations like zero padding or ReLU do not induce quantization error.

Once the three quantization parameters are defined we can proceed with the quantization operation. Starting from a real-valued vector $\mathbf{x}$ we first map it to the *unsigned* integer grid $\{0, \dots, 2^b - 1\}$:

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right), \tag{4}$$

where $\lfloor \cdot \rceil$ is the round-to-nearest operator and clamping is defined as:

$$\text{clamp}\,(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c. \end{cases} \tag{5}$$

To approximate the real-valued input $\mathbf{x}$ we perfrom a *de-quantization* step:

$$\mathbf{x} \approx \widehat{\mathbf{x}} = s\,(\mathbf{x}_{\text{int}} - z) \tag{6}$$

Combining the two steps above we can provide a general definition for the *quantization function*, $q(\cdot)$, as:

$$\widehat{\mathbf{x}} = q(\mathbf{x}; s, z, b) = s\left[\text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right) - z\right], \tag{7}$$

Through the de-quantization step, we can also define the quantization grid limits $(q_{\text{min}}, q_{\text{max}})$ where $q_{\text{min}} = -sz$ and $q_{\text{max}} = s(2^b - 1 - z)$. Any values of $\mathbf{x}$ that lie outside this range will be clipped to its limits, incurring a *clipping error*. If we want to reduce the clipping error we can expand the quantization range by increasing the scale factor. However, increasing the scale factor leads to increased *rounding error* as the rounding error lies in the range $\left[-\frac{1}{2}s, \frac{1}{2}s\right]$. In section 3.1, we explore in more detail how to choose the quantization parameters to achieve the right trade-off between clipping and rounding errors.

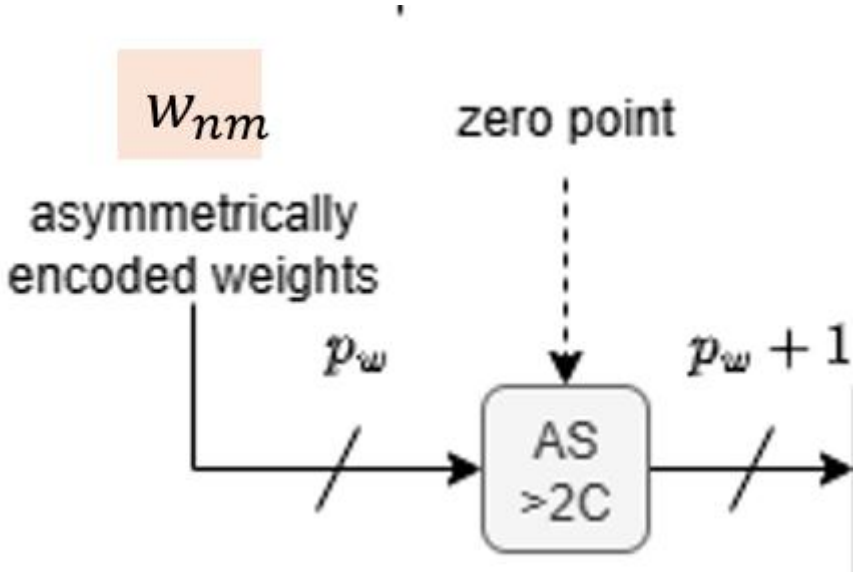
[Nagel et al, 2021]

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right)$$

Asymmetric encoding can be mapped to 2's complement simply by subtracting zero point.

Asymmetric encoding can be mapped to 2’s complement simply by subtracting zero point.

This is what the block ”AS>2C” is doing.



Alternative: symmetric quantization

Formulae on the right.

For the case considered previously, preparation action is not needed in the MAC implementation.

We may select that +127 correspons to +1.2.
The slope with this choice is $s = \frac{1.2}{127}$.

-0.8 maps to integer

$$[-0.8 \times 127/1.2] = [-84.6667] = -85$$

We are not using full range of 8-bit integers!

$$s \approx \mathbf{9.449 \times 10^{-3}}$$

With previous asymmetric quantization, it was

$$s = \frac{2.0}{255} \approx \mathbf{7.843 \times 10^{-3}}$$

Symmetric quantization is a simplified version of the general asymmetric case. The symmetric quantizer restricts the zero-point to 0. This reduces the computational overhead of dealing with zero-point offset during the accumulation operation in equation (3). But the lack of offset restricts the mapping between integer and floating-point domain. As a result, the choice of signed or unsigned integer grid matters:

$$\widehat{\mathbf{x}} = s\,\mathbf{x}_{\text{int}} \tag{8a}$$

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil; 0, 2^b - 1\right) \quad \text{for unsigned integers} \tag{8b}$$

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil; -2^{b-1}, 2^{b-1} - 1\right) \quad \text{for signed integers} \tag{8c}$$

Unsigned symmetric quantization is well suited for one-tailed distributions, such as ReLU activations (see figure 3). On the other hand, signed symmetric quantization can be chosen for distributions that are roughly symmetric about zero.

[Nagel et al, 2021]

So, better precision with asymmetric quantization.

But, testing would be needed to see, how much better it is in practice. Note that with symmetric quantization that ”AC>2C” is not needed.

Notes

Formula

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right)$$

maps to unsigned integers

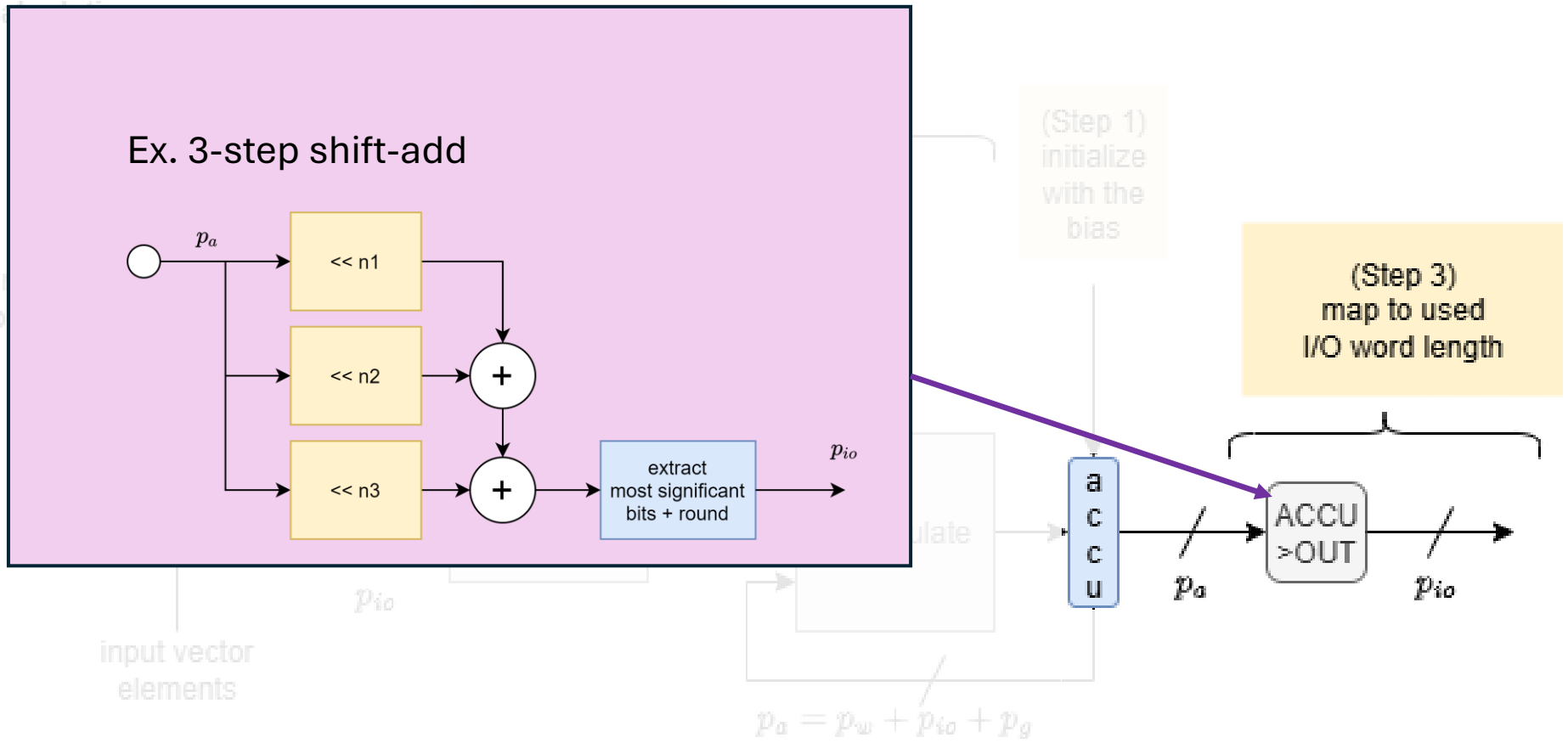
- In the case of asymmetric quantization, mapping to **unsigned integers** is sufficient. There is no need to consider signed integers, but of course one can do it!
  - For example:
    - 8-bit **unsigned int** encoding with zero point +102
    - Corresponds to 8-bit **signed int** encoding with zero point -26
    - As both have range [-102, +153]
- Asymmetric quantization can provide benefit in cases, when values to be stored into memory are have asymmetric distribution around zero. Otherwise, symmetric quantization is enough.

MAC for FC layer, step 3

$$y_n = \sum_{m=1}^{M} w_{nm} x_m + b_n$$

”ACCU>OUT” block maps the accumulated result to 2’s complement integer.

Before taking MSB part, the accumulator value is multiplied using shift-add ops.



Why multiplication before taking MSB?

Assume that with calibration data we observe that 14-bit accumulator has range

+494 = 00000111101110

-488 = 11111000011000

Assume that output word length is 6 bits and we take MSB of above. We get

000001 = +1 (with truncation) 000010 = +2 (with round-to-nearest)

111110 = -2

Conclusion: Six-bit precision available is almost completely useless!!

However, if we multiply accumulator by 2^4=16, which is ”<< 4”, four-bit shift to right, we get values in the range

+7904 = 01111011100000

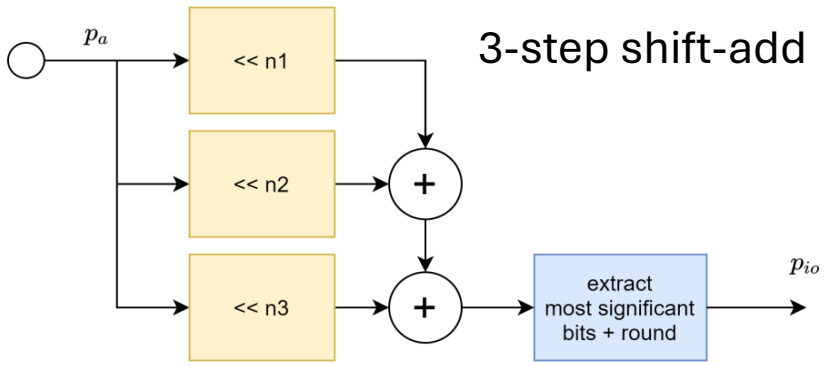
-7808 = 10000110000000

Now, if we take MSB of these values, we get

011110 = +30 (with truncation)

100001 = -31

Conclusion: Six-bit precision available is utilized quite well!

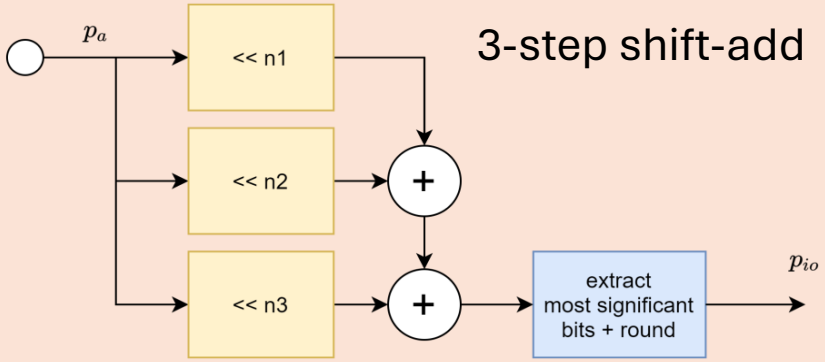


3-step shift-add

Full-multiplier not introduced due to complexity. Few shift-adds can introduce necessary gain.

Quick test

- What are n1, n2 and n3 on the right, if one wants to achieve multiplication by 2.75?
  - Note: negative value corresponds to shift-to-right



Goto **presemo.oulu.fi/spslec3** and provide your solution

DT3 Problem 3

- Copy files from Moodle to a folder and set it as the working folder in Matlab
- **QNN_comparison.mlx** is a Matlab live script
  - Open it in Matlab
- In the first section of it, call to **hyperparameters** provides setup for a group
  - Set GROUP variable for your group number

This computation is demonstrated in a Matlab live script **QNN_comparison.mlx**. The script runs quantized NN simulator **NN_FCRELUFC_SIM.m** in a loop, where varying word lengths are used for NN input/output.

Use your group number as the parameter for the **hyperparameters** function call that can be found in the beginning of the live script.

a) Your task is to implement the NN weight quantization. The function is named **compute_qweights** and it can be found at the end of the **search_asymmetric_parameters**. Remove the call to function **quantizew**, and replace it with your own code. You must also saturate the values if they exceed the limits of the data type used.

Hint: Before you proceed to the next task, make sure that your quantization works. You can get the correct result by calling the quantizew function on the Matlab command line. Your implementation should produce identical results. See the documentation of **compute_qweights** *if you need help calling the quantizew function.*

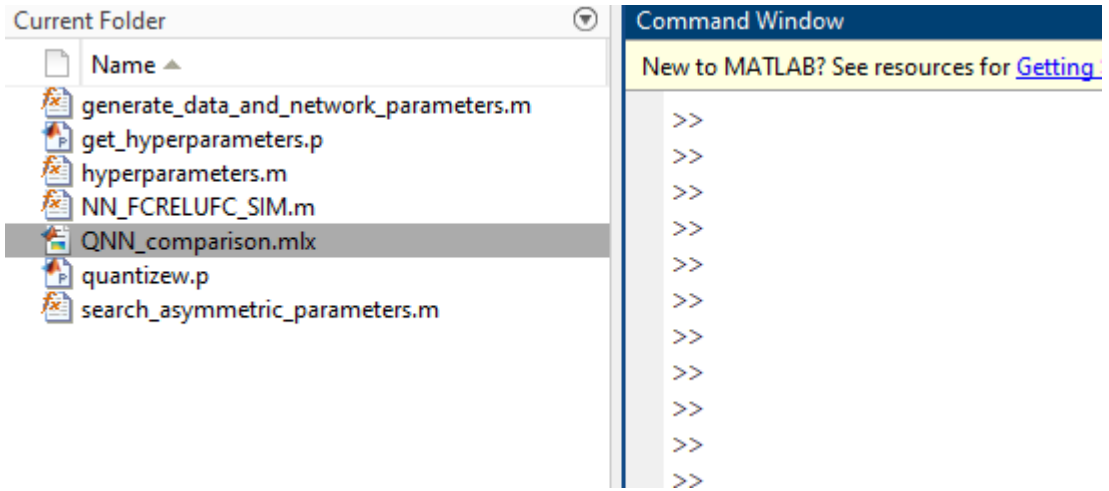
b) When you have implemented the function in a), run the live script. You should see a warning. Investigate the warning. In your report, explain what the warning is and why it is raised. It is possible to change the parameters of the simulation from the live script file **QNN_comparison.mlx** so that the warning disappears. Find a way to do this. Report what parameters you changed and what the parameters are.

c) The simulation is run with varying bit width for the fully connected layer inputs. This includes the dataset as it is the input for the first fully connected layer. Find the figure at the end of the Matlab live script where the average difference and standard deviation between the floating point outputs and quantized outputs are illustrated. What is the least amount of bits that will result in a standard deviation of less than T (you can find the T for your group from the Table at the end of this document)?

Return your modified **search_asymmetric_parameters.m** and **QNN_comparison.mlx** files to Moodle together with your pdf report.

DT3 Problem 3

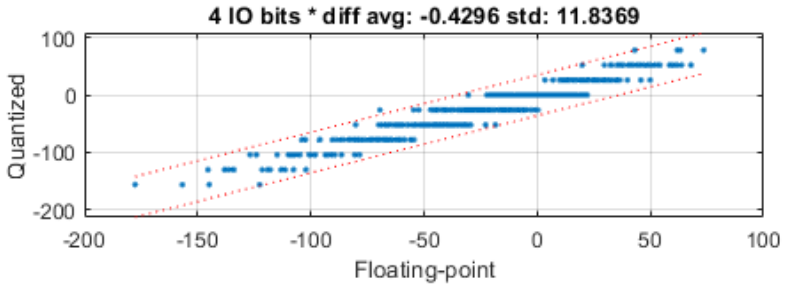
- Copy files from Moodle to a folder and set it as the working folder in Matlab
- **QNN_comparison.mlx** is a Matlab live script
  - Open it in Matlab
  - In the first section of it, call to **hyperparameters** provides setup for a group
  - Set GROUP variable for your group number

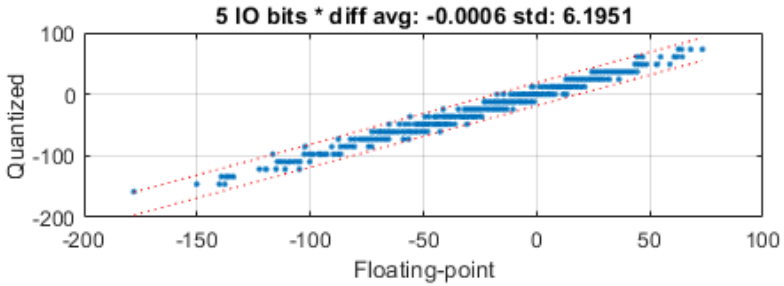


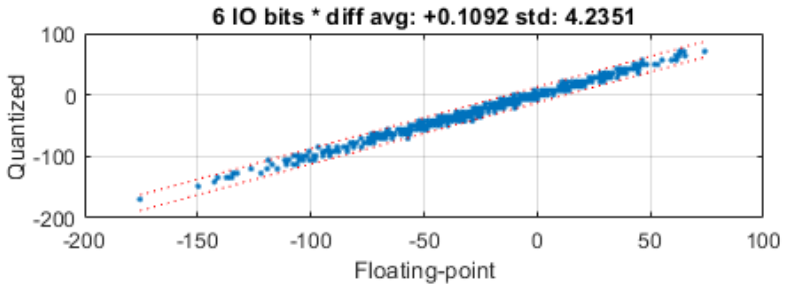
Live script **QNN_comparison.mlx**

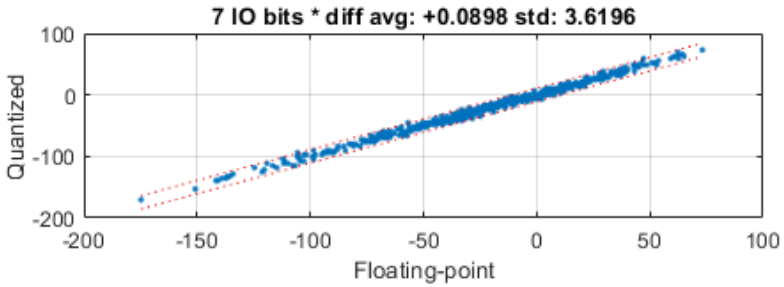
Contains three sections:

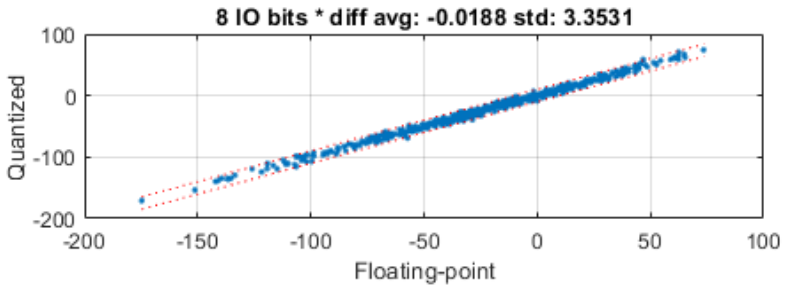
1. fetch of group's data for experiments
2. loop performing simulation for different I/O word lengths
3. display of results obtained in step 2











X-axis : outputs of original floating-point NN
Y-axis: outputs of integer NN scaled

When I/O word length increased, results become more and more similar.

QNN_comparison.mlx

```
GROUP = 0; % Replace with your group number. Then Run Section from the button above

[rng_seed, N_data_points, data_dimension, layer_0_dimension, output_dimension] = hyperparameters(GROUP);

rng(rng_seed)

[input_data, calibration_data, layer_0_weights, layer_0_bias, output_weights, output_bias] = ...
generate_data_and_network_parameters(data_dimension, N_data_points, layer_0_dimension, output_dimension);

flparams.weights_0 = layer_0_weights;
flparams.weights_1 = output_weights;
flparams.bias_0 = layer_0_bias;
flparams.bias_1 = output_bias;

fprintf('Neural network parameters, calibration data, and test data ready for Group %d.\n',GROUP);
fprintf('NN input layer has size %d, intermediate layers have size %d, and output layer size ia %d.\n',size(flparams.weight
fprintf('There are %d calibration data samples and %d test data samples.\n',size(calibration_data,2),size(input_data,2));
fprintf('Ready for experiments.\n');
```

Original floating-point NN to be quantized

```
fprintf('Running experiment for different I/O word lengths.\n');

IO_word_lengths = 4:8;

N = length(IO_word_lengths);

hsims = cell(1,N);
evaluations = cell(1,N);

hwprops.NbitsW = 4;     % Number of bits used for weights
hwprops.Nguard = 1;     % Number of guard bits for accumulator (word length = NbitsIO + NbitsW + Nguard)
hwprops.postQsas = 2;   % Number of shift-adds available for accumulator multiplication to improve output precision

for ind = 1:N
    fprintf('Quantized NN for IO word length %d\n',IO_word_lengths(ind));
    hwprops.NbitsIO = IO_word_lengths(ind);  % Number of bits used for I/O
    hsims{ind} = NN_FCRELUFC_SIM(flparams,hwprops);
    hsims{ind}.calibrate(calibration_data,max(abs(calibration_data(:))));
    evaluations{ind} = hsims{ind}.evaluate(input_data);
end
```

Setting HW properties (I/O word length in loop)

PTQ is done in calibration

Testing quantized NN with other data

Before you can obtain these results in (c), you should make a working NN in subproblems (a) and (b).

Problem 3(a) – at the end of **search_asymmetric_parameters.m** there is

```
%%%%%%%%%%%%%%%%%%
%
% Parameters
% weights  The original floating point weights
% Nbits    Number of bits used in quantization
% s        Scaling factor
% zeropt   Zero point
%
% Returns
% w_uint   Integer quantized weights. Note that you are supposed only to
%          simulate unsigned integer arithmetics, the data type of this
%          variable must be double.
% w_approx Dequantized coefficients.
%
%%%%%%%%%%%%%%%%%%
function [w_uint,w_approx] = compute_qweights(weights,Nbits,s,zeropt)
    [w_uint, w_approx] = quantizew(weights, Nbits, s, zeropt);
end
```

Obfuscated code quantizew.p produces right **w_uint** and **w_approx**.

Write a Matlab code which does the same: substitute quantizew call with your own code.

Hint: study formulas 4-7 in Nagel's paper

Hint: use vectorized processing ...

Matlab programming hints

1. Generating a random 5x3 matrix containing integers in the range -5 ... +10
2. Set elements greater than +7 equal to -20

Two approaches:

- Element-wise processing using loops (inefficient)
- Vectorized processing (efficient, simple code)

```
A = zeros(5,3);
for r = 1:size(A,1)
        for c = 1:size(A,2)
                A(r,c) = round (15 * rand(1)) - 5;
                if A(r,c) > 7, A(r,c) = -20;
        end
end
```

```
A = round(15 * rand(5,3)) - 5;
B = A > 7;
A(B) = -20;
```

```
A = round(15 * rand(5,3)) - 5;
A(A > 7) = -20;
```

Problem 3(b) – get rid of saturation warnings

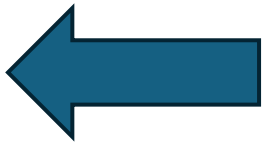
```
Quantized NN for IO word length 4
* HW properties *
I/O word length: 4 (used for all NN layers)
Weight encoding: 4 bits, asymmetric
ACCU word length: 9 (1 guard bits)
ACCU output mapping, gain shift-add count: 2

Same input quantization for floating-point NN.
* Calibration *
DIFF provide information about the quality of results at various stages of NN co

# DIFF at 'input': max(abs) 0.000000 avg -0.000000 stdev 0.000000 flo/range -1.9
# NOTE: no difference because floating-point input is tuned to match with intege
Asymmetric coding for layer 0 weights: scale 0.129927 zeropt 7 Nbits 4 - uint/ra
Symmetric coding for layer 0 bias: scale 0.051970 - int/range -7..7

Warning: layer 0: accumulator saturates for 994 (49.70%) inputs

# DIFF at 'layer0-before-Q (i.e. accu)': max(abs) 57.694649 avg -0.199374 stdev
Shifts: 0 implement 1.000000
# DIFF at 'layer0-after-Q (i.e. output)': max(abs) 57.694649 avg -0.184199 stdev
# NOTE: Compared to previous difference, increase in difference (less precision)
# DIFF at 'relu-out': max(abs) 53.783829 avg -2.919176 stdev 6.916803 flo/range
Asymmetric coding for layer 1 weights: scale 0.183895 zeropt 10 Nbits 4 - uint/r
Symmetric coding for layer 1 bias: scale 0.305827 - int/range -2..3
# DIFF at 'layer1-before-Q (i.e. accu)': max(abs) 121.096046 avg 6.926952 stdev
Shifts: 0 implement 1.000000
# DIFF at 'layer1-after-Q (i.e. NN output)': max(abs) 116.814468 avg 6.772508 st
# NOTE: Compared to previous difference, increase in difference (less precision)
```



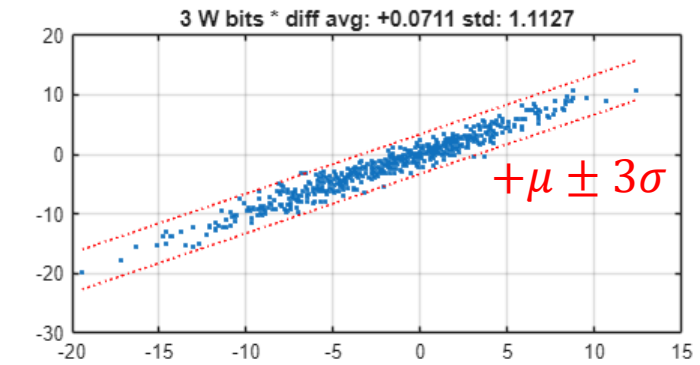
Another experiment (varying number of weight bits)

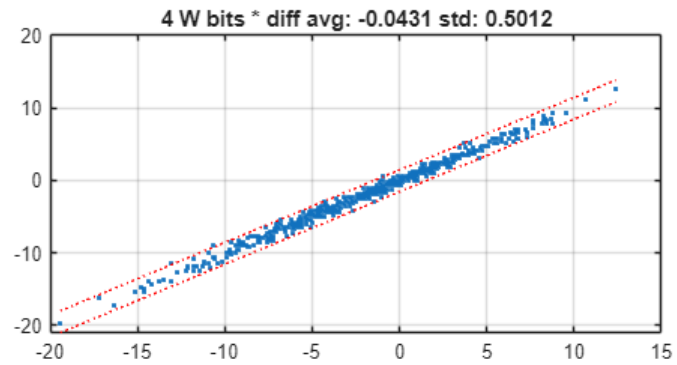
Difference of original floating-point NN's output and quantized NN's scaled output.

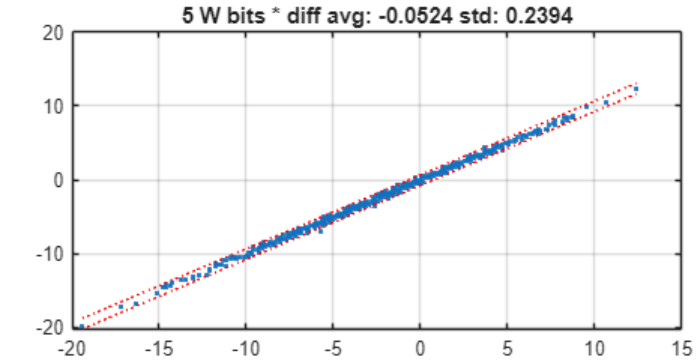
Number of weight bits 3-8.

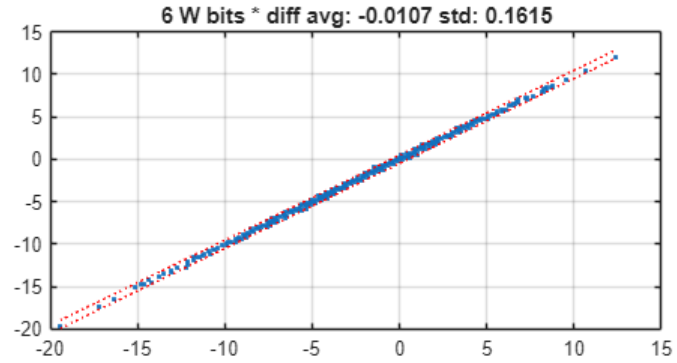
The result indicates that good performance is possible with just few weight bits!

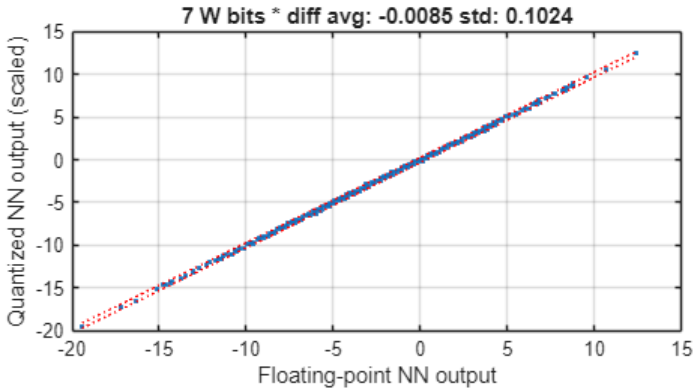
Depends on the use of NN outputs, how accurate and precise values are needed.

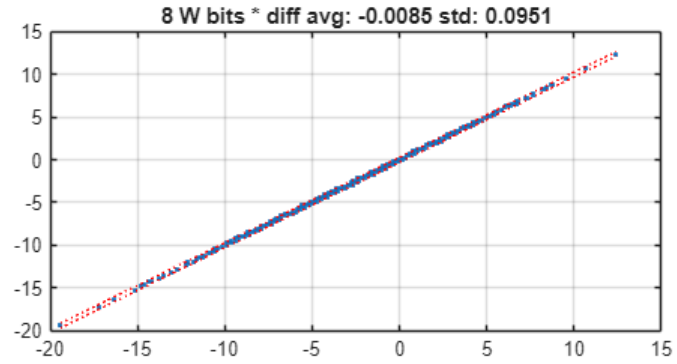












What about energy efficiency?

Mark Horowitz. Computing's energy problem (and what we can do about it). In 2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC), pp. 10–14. IEEE, 2014.

Operation	Integer		Floating Point	
	8-bit	32-bit	16-bit	32-bit
Addition	0.03 pJ	**0.1 pJ**	0.4 pJ	0.9 pJ
Multiplication	0.2 pJ	3.1 pJ	1.1 pJ	**3.7 pJ**

Table 1: Energy cost of various arithmetic operations cited from Horowitz (2014).

Explanation:

Addition 32+32 vs 8+8 – about 4 times more work $\frac{0.1}{0.03} \approx 3.3$

Multiplication 32*32 vs 8*8 – about 16 times more work $\frac{3.1}{0.2} = 15.5$

Based on these figures, let us approximate and plot, how much energy savings there will be for the sample system. Considering only fully-connected layers, RelU layers ignored.

Excel sheet for comparison

power-budget-fcrelufc.xls

Will be put in Moodle under Lectures

NN (FC+RELU+FC) power budget

Indicative estimates

	Number of bits	Vector length	Energy (8bit) / pJ
I/O	6	Input 15	ADD 0.03
Weights	6	FC0 out 10	MPLY 0.20
Guard	3	Output 3	
Accu	15		
		Q shift-adds 3	Frequency/kHz 30

			Complexity		Times	Energy					
Layer	**Step**	**Operations & associated word lengths**	**Add**	**Mply**		**Add**	**Mply**	**Total / pJ**	**Proportion**	**Power / mW**	
Fully-conn. layer 0	as>2c	**Bw** bit addition, **Min*Mfc** times	0.75		150	3.38	0	3.38	8.44 %	0.101	
	MAC	**(Bw+1)*Bio** bit multiplication, **Ba** bit addition, **Min*Mfc** times	1.875	0.65625	150	8.44	19.6875	28.13	70.32 %	0.844	
	accu>out	K **Ba** bit additions, **Mfc** times	5.625		10	1.69	0	1.69	4.22 %	0.051	
RelU ignored											
Fully-conn. layer 1	as>2c	**Bw** bit addition, **Mfc*Mout** times	0.75		30	0.68	0	0.68	1.69 %	0.020	
	MAC	**(Bw+1)*Bio** bit multiplication, **Ba** bit addition, **Mfc*Mout** times	1.875	0.65625	30	1.69	3.9375	5.63	14.06 %	0.169	
	accu>out	K **Ba** bit additions, **Mout** times	5.625		3	0.51	0	0.51	1.27 %	0.015	
							Sum	**39.99**		**1.200**	

Versus	
FLP16	**FLP32**
14.81 %	4.83 %

Floating-point (16-bit)

Energy (16-bit FLP) / pJ

ADD 0.40

MPLY 1.10

Layer	**Step**	**Operation count**	**Add**	**Mply**		**Add**	**Mply**	**Total / pJ**	**Proportion**	**Power / mW**
Fully-conn. layer 0	MAC	Min*Mfc times			150	60	165	225.00	83.33 %	6.75
RelU ignored										
Fully-conn. layer 1	MAC	Mfc*Mout times			30	12	33	45.00	16.67 %	1.35
								270.00		**8.10**

Floating-point (32-bit)

Energy (32-bit FLP) / pJ

ADD 0.90

MPLY 3.70

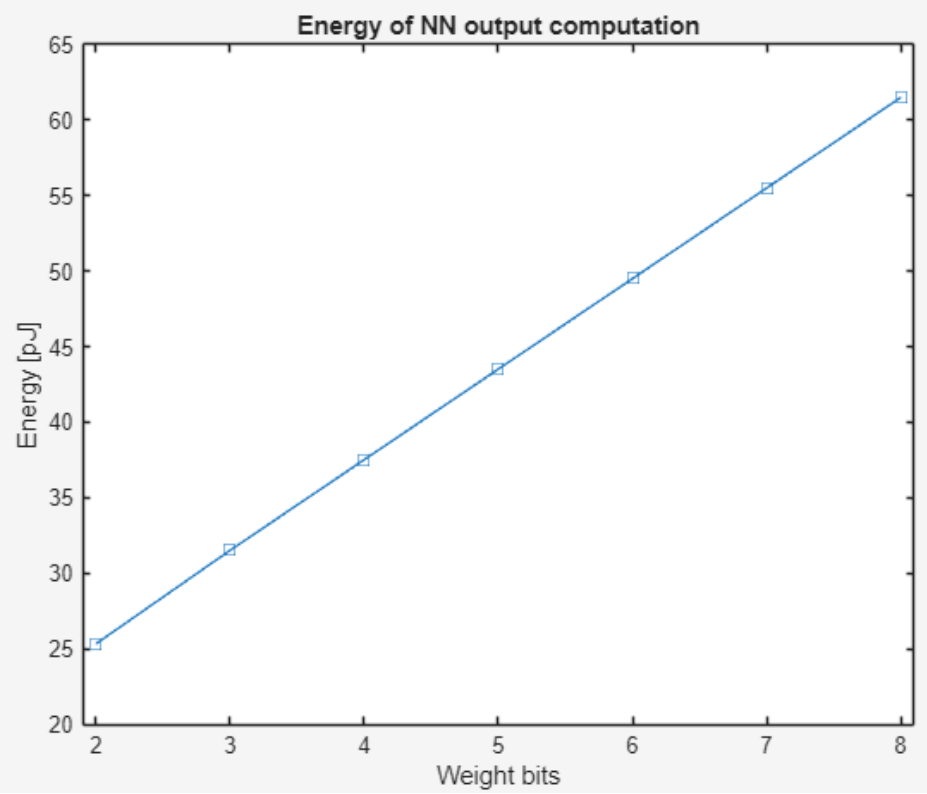
Layer	**Step**	**Operation count**	**Add**	**Mply**		**Add**	**Mply**	**Total / pJ**	**Proportion**	**Power / mW**
Fully-conn. layer 0	MAC	Min*Mfc times			150	135	555	690.00	83.33 %	20.7
RelU ignored										
Fully-conn. layer 1	MAC	Mfc*Mout times			30	27	111	138.00	16.67 %	4.14
								828.00		**24.84**

Energy-efficiency can be much better with integer arithmetic.
But note: 8-bit minifloats are not considered in the sheet.

Plotting energy against varying parameter

Fixed: 8 bit I/O, 3 guard bits

Dependence of energy on the number of weight bits:



Summary

- The choice between fixed-point and floating-point alternatives depends on the application
  - If large variations in numeric values are expected, then floating-point seems to be better alternative as scaling can be hard to manage
  - Neural networks may contain huge number of parameters, but the arithmetic is relatively simple and scaling is manageable. Accuracy of numerical results might also not be so critical. Then, fixed-point arithmetic becomes attractive.
- FIR filtering and NN computations are applications for MAC based processing (fixed-point)

This week …

- Deadline for group registrations is Wednesday
- Quiz 1 on Thursday (begins after short lecture, about 11:00)
  - Sample questions will be put available under lectures
- Deadline of Design Task 1: Thursday evening

- Design Task 2 handout: Thursday morning
  - Lecture on that day is the 1st one related to it

Before leaving today, put your name to the list to get 0.5 points for participation!