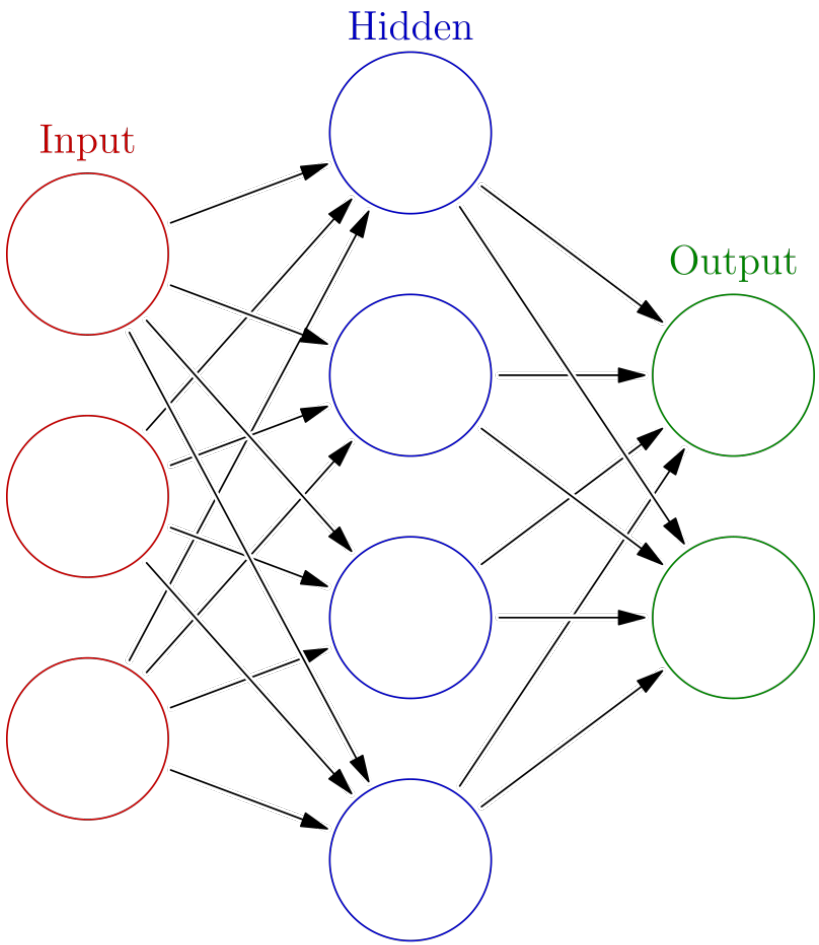
Parametrized HDL Code Generation For Activation Functions

Aurélien Delmotte, Thibault Hilaire, Andrea Pinna
Email : {*Forename.Surname*}*@lip6.fr*

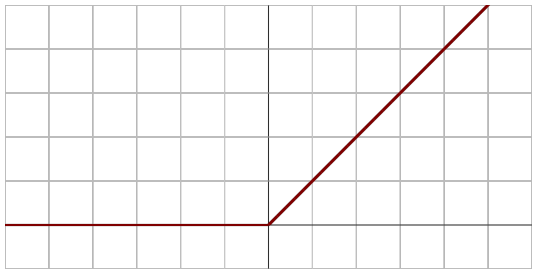
July 6, 2026

Outiline

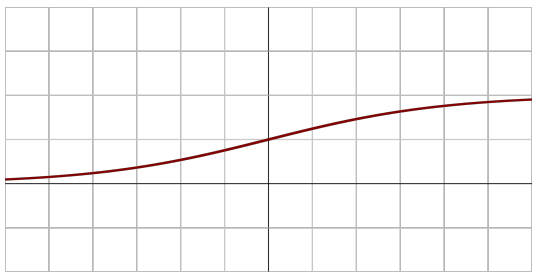
Activation functions



1-hidden layer neural network



RELU



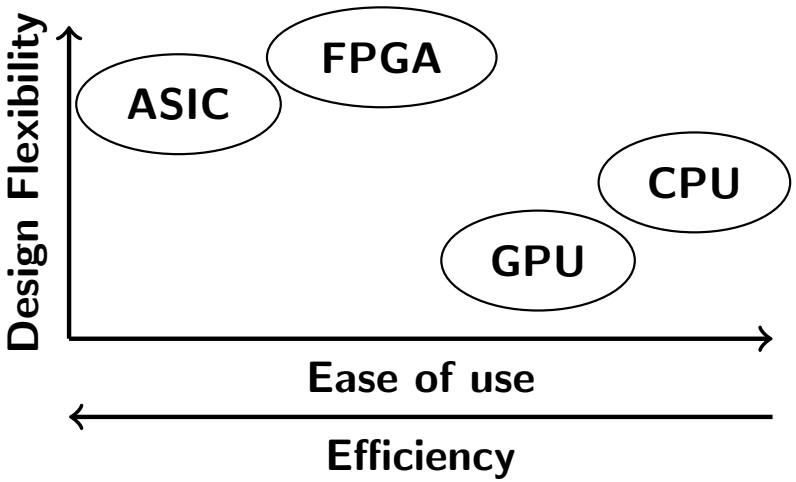
Sigmoid $(\frac{1}{1+e^{-x}})$

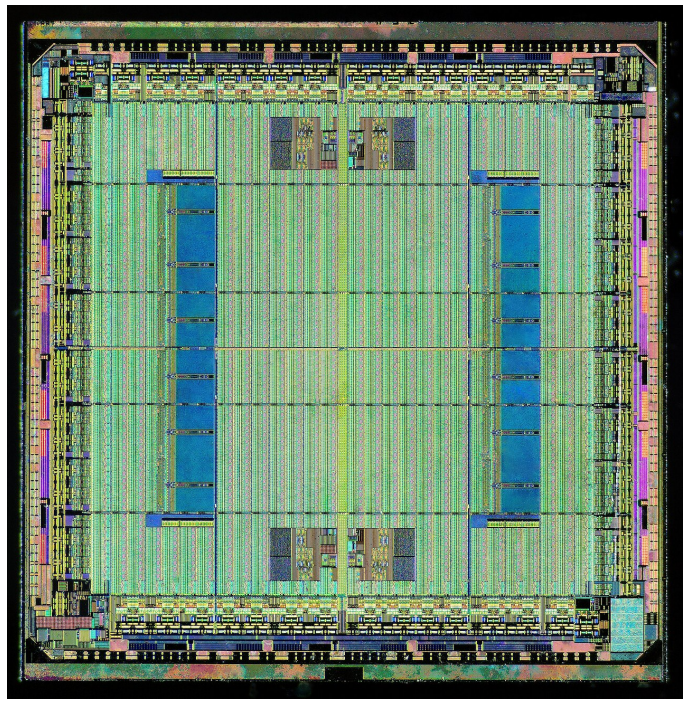
$$\frac{e^{x_i}}{\sum_j e^{x_j}}$$

Softmax

The Challenge: Some are compute-intensive. At the "edge", every computation costs energy and time and memory.

FPGAs



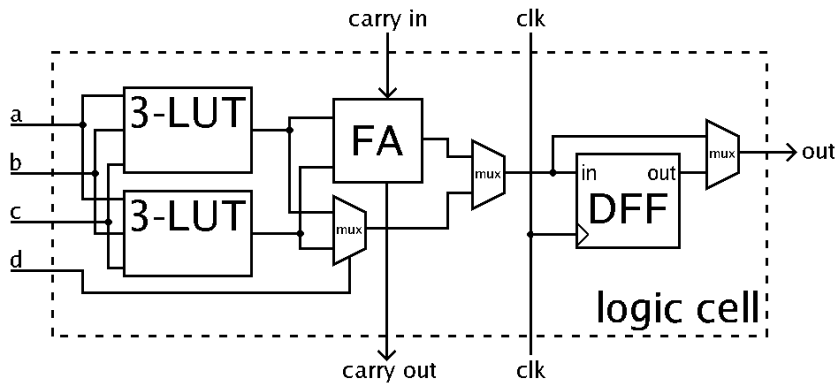


Key advantages:

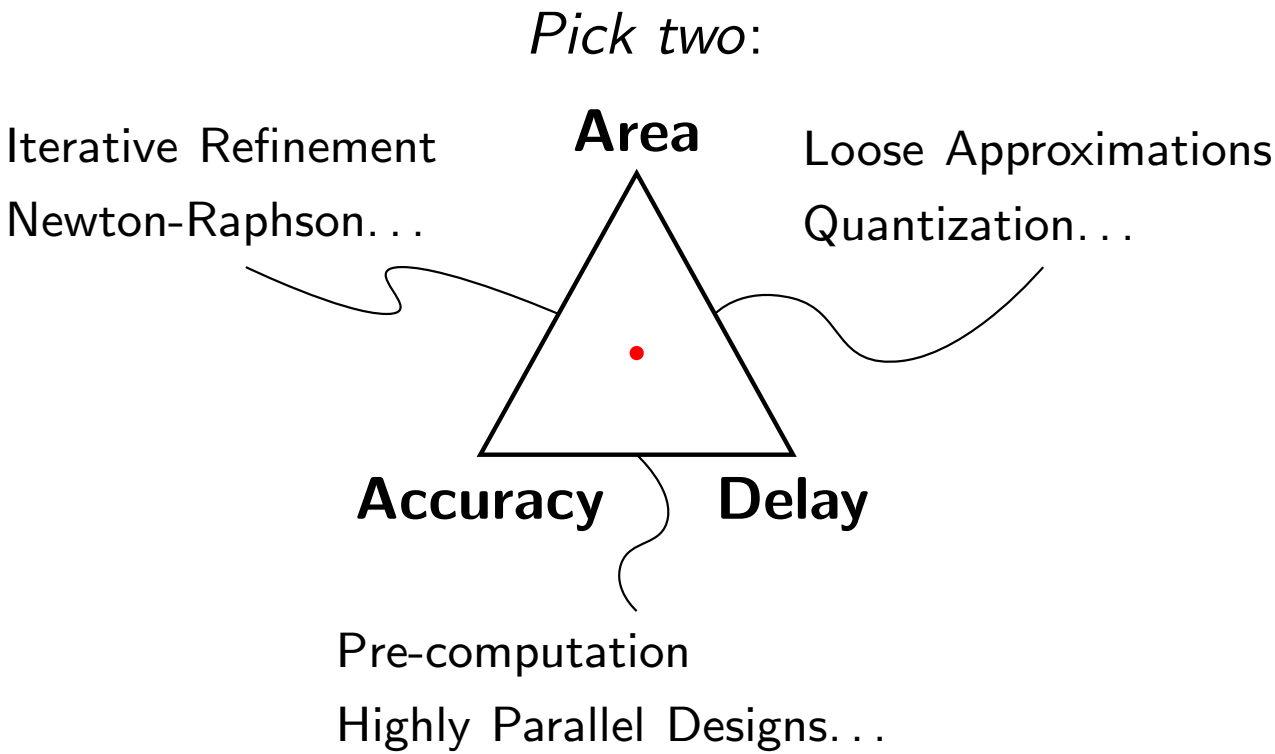
- Flexibility
- Lower energy consumption
- High degrees of Parallelism
- Reprogramability

Disadvantages :

- Cost
- Design Complexity



Constraints



But there's also:

- Throughput
- Power
- Different ressource types

Problem statement

How to efficiently implement complex functions like **Softmax** on **FPGA** for Edge AI?

- HLS allows automatic generation but is inflexible
- Vendor tools are easy to use but not fully parametrizable
- Manual hardware design is slow and error-prone
- FloPoCo is efficient but lacks key functions such a max function or a **softmax**

Code Generator

We Thus propose a Python tool to help design complex functions, structured as follows:

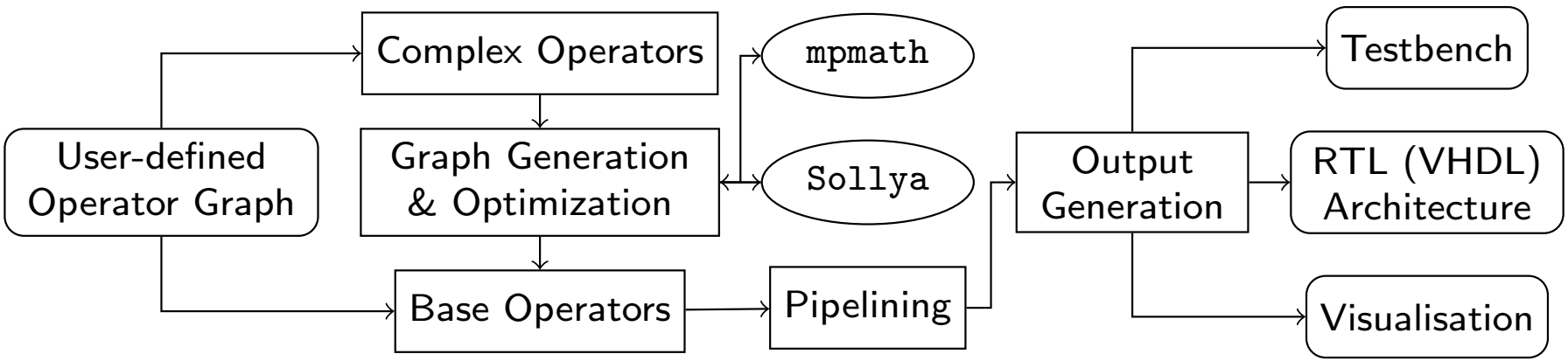
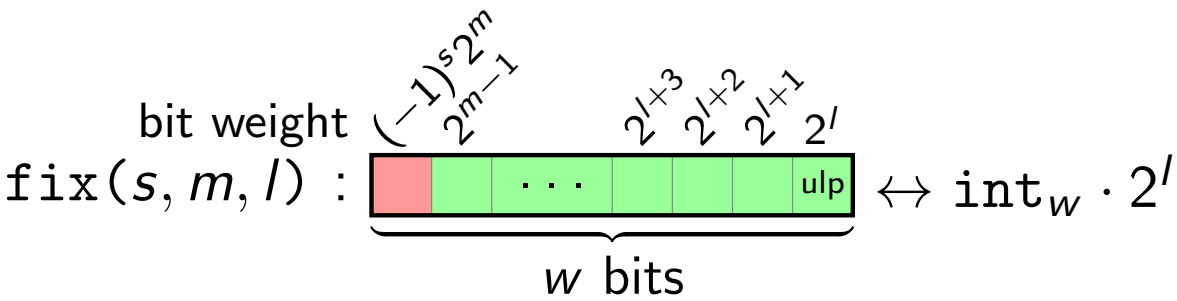


Figure: Generator workflow

We aim to implement:

- Tight Designs
- Parametrizable fixed-point operators
- Custom Accuracy (as opposed to FloPoCo's 1ulp target)
- Target Specific Optimizations

Fixed-point numbers



Compared to Floating-point:

- Cheap integer operators
- Lack of overhead
- Smaller dynamic range
- Need to track scaling factors at design-time

Error Analysis

Some operators may incur error:

- Rounding/Truncation, Function Approximation...

To inform decisions on operator accuracy we need ways to:

- **Compute error budgets**
  Backward error analysis on each operator to find input error budgets based on output error
- **Distribute this budget to minimize cost**
  NP-Hard problem, no clear cost function
  $\rightarrow$ Currenlty ad-hoc Greedy Algorithm
- **Check design accuracy**
  Interval arithmetic with interval splitting for Look-up Tables:

$$\tilde{x} = x + \delta \in [\underline{x}, \overline{x}] + [\underline{\delta}, \overline{\delta}] \rightarrow \tilde{x} \in [\underline{x_0}, \overline{x_0}] \cup [\underline{x_1}, \overline{x_1}] \cdots + [\underline{\delta}, \overline{\delta}]$$

Rounding

The Problem: Standard Truncation is Biased

- Simple truncation introduces a negative error (bias).
- This bias accumulates through computations.
- For an exact value x and its truncated version $\tilde{x}$:

$$[\tilde{x} - x] < 0$$

Our Solution: Bias-Compensated Rounding

- Actively compensates for the known truncation bias.
- Relies on error interval ($\delta = [\underline{\delta}, \overline{\delta}]$) tracking.
- Calculates a compensation factor Δ_{bias} and adds it before truncation:

$$\Delta_{\text{bias}} = \left\lfloor \frac{1}{2} \left(\underline{\delta} + \overline{\delta} + 2^{l'} - 2^{l} \right) \right\rfloor_{l}$$

- Results in a **symmetric error distribution** around zero, minimizing maximum absolute error.

Softmax

Mathematical Definition:

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

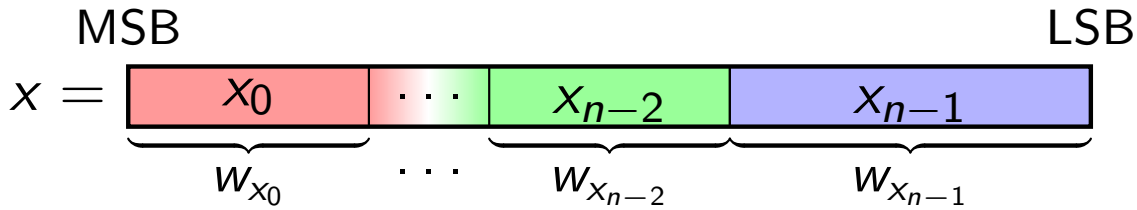
(Exponents shifted by $\max(x)$ *to ensure better bounds)*

Inferred steps:

1. Compute $x_m = \max(x)$ Comparator tree
2. Then compute each $x_i - x_m$ Simple substractions
3. Compute $e^{x_i - x_m}$ Need Approximation
4. Compute $\sum_{j=0}^{k-1} e^{x_j - x_m}$ Adder tree
5. And then $\frac{1}{\sum_{j=0}^{k-1} e^{x_j - x_m}}$ Complex Algorithm
6. Finally assemble each $\text{softmax}(x)_i = \frac{e^{x_i - x_m}}{\sum_{j=0}^{k-1} e^{x_j - x_m}}$ Multipliers

Exponential

Exponential Strategy:



$$x = \sum_{i=0}^{n-1} x_i \rightarrow e^x = e^{\sum_i x_i} = \prod_{i=0}^{n-1} e^{x_i}$$

1. **Range Reduction**: Break x into smaller parts.
2. **Approximation**: Compute e^{x_i} on each part using:
   - Lookup Tables
   - Degree 1 Taylor aproximation: $e^{x_i} \sim 1 + x_i$
3. **Reconstruction**: Multiply the partial results together and round to specified format.

Exponential

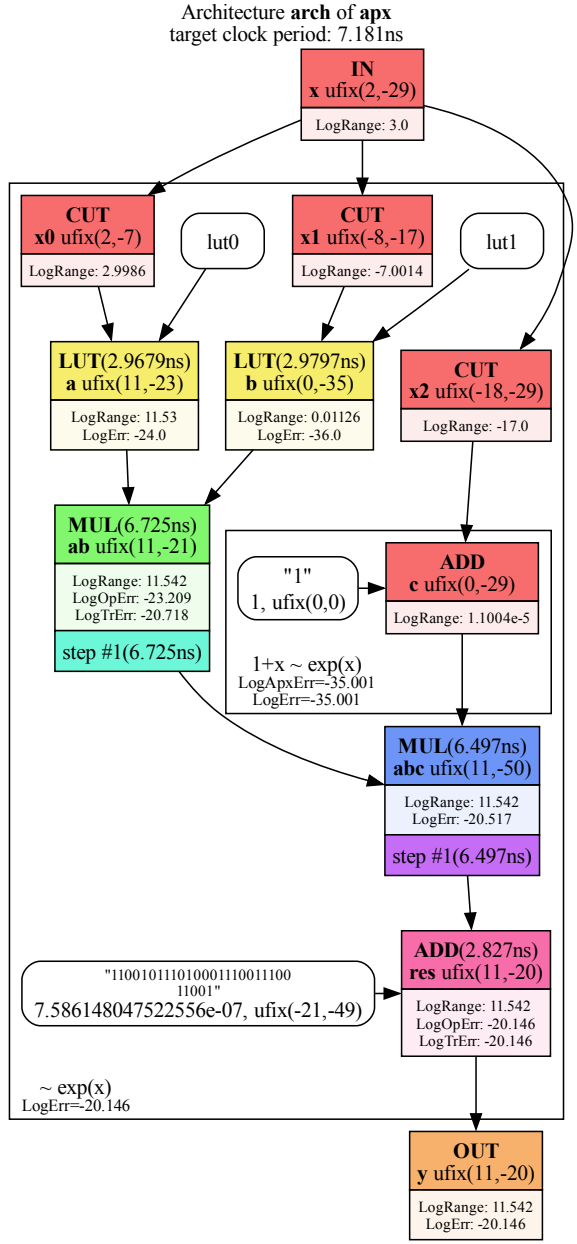


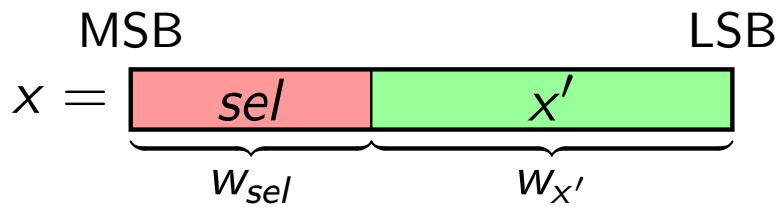
Figure: e^x with x in $\texttt{ufix}(2, -29)$ and output in 32bits with 1ulp accuracy and 10bit wide tables

Reciprocal

Multiple Methods exist such as:

- Newton-raphson
- Goldschmidt Algorithm
- Generic function approximation

We chose **Piecewise polynomial approximation**:



$$\frac{1}{x} \sim f_{sel}(x') = c_0[sel] + c_1[sel] \cdot x' \ldots$$

- **Range Reduction**: split domain into sel and x'
- **Approximation**: Use `sollya` to compute the coefficients of f_{sel} for each sel *(Degree is found iteratively based on desired accuracy.)*
- **Evaluation**: Use either Horner's scheme or Estrin's method to evaluate the polynomials.

Reciprocal

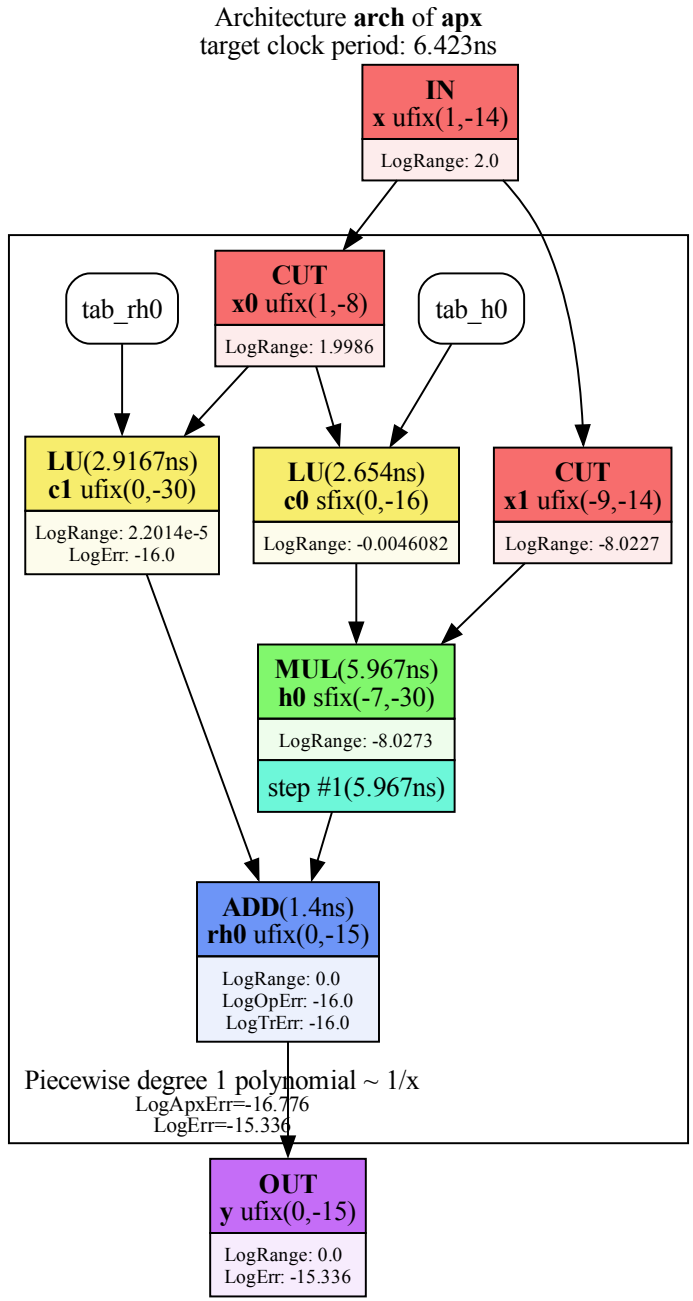


Figure: $1/x$ with x in [1,2[with 16bit I/O, 1ulp accuracy and 10bit wide tables

Softmax

Mathematical Definition:

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

(Exponents shifted by $\max(x)$ *to ensure better bounds)*

Inferred steps:

1. Compute $x_m = \max(x)$ Comparator tree
2. Then compute each $x_i - x_m$ Simple substractions
3. Compute $e^{x_i - x_m}$ Need Approximation
4. Compute $\sum_{j=0}^{k-1} e^{x_j - x_m}$ Adder tree
5. And then $\frac{1}{\sum_{j=0}^{k-1} e^{x_j - x_m}}$ Complex Algorithm
6. Finally assemble each $\text{softmax}(x_i) = \frac{e^{x_i - x_m}}{\sum_{j=0}^{k-1} e^{x_j - x_m}}$ Multipliers

Softmax

Mathematical Definition:

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

(Exponents shifted by $\max(x)$ *to ensure better bounds)*

Actual steps:

1. Compute $x_m = \max(x)$
2. Then compute each $x_i - x_m$

2.5. Compute error budget for exponentials

3. Compute $e^{x_i - x_m}$
4. Compute $\sum_{j=0}^{k-1} e^{x_j - x_m}$
5. And then $\frac{1}{\sum_{j=0}^{k-1} e^{x_j - x_m}}$

5.5. In case degree is too large, tighten budget and goto 3.

6. Finally assemble each $\text{softmax}(x_i) = \frac{e^{x_i - x_m}}{\sum_{j=0}^{k-1} e^{x_j - x_m}}$ (and Round)

Softmax

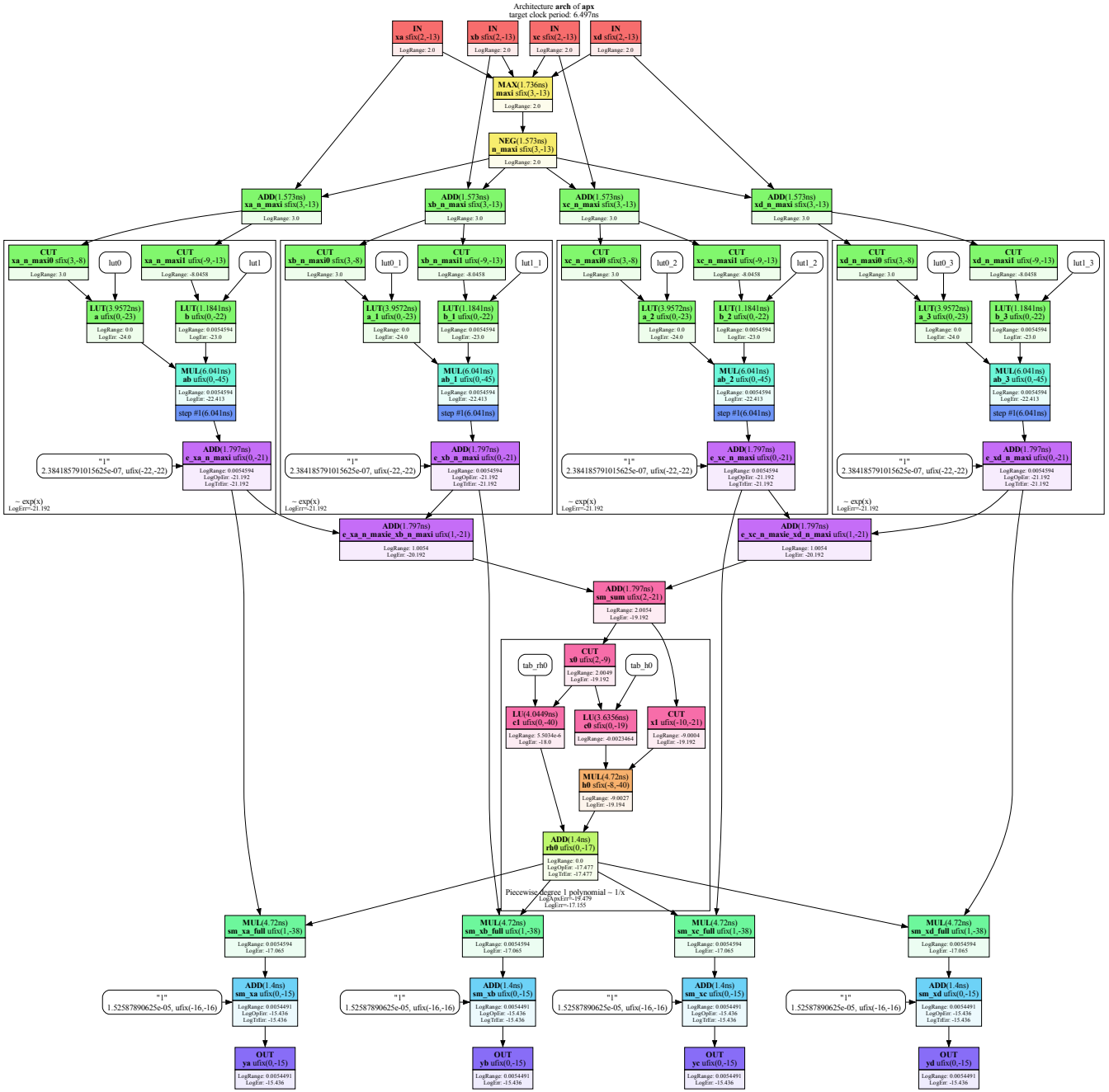


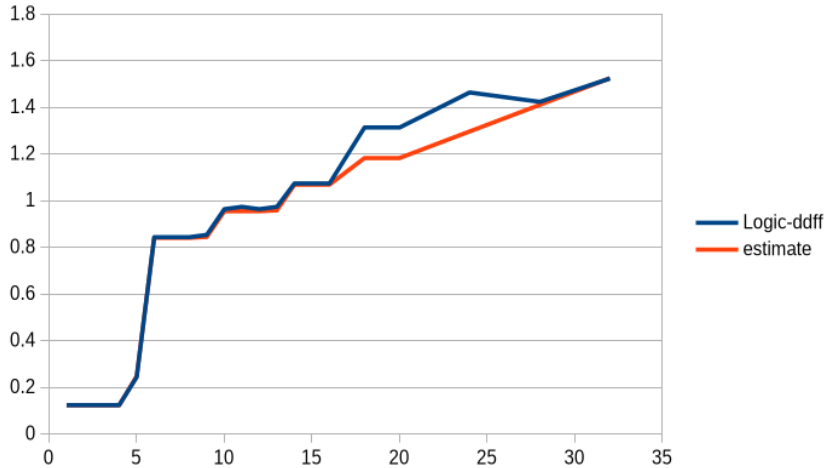
Figure: 4 input 16 bit softmax with 12bit wide tables

Delay Heuristics

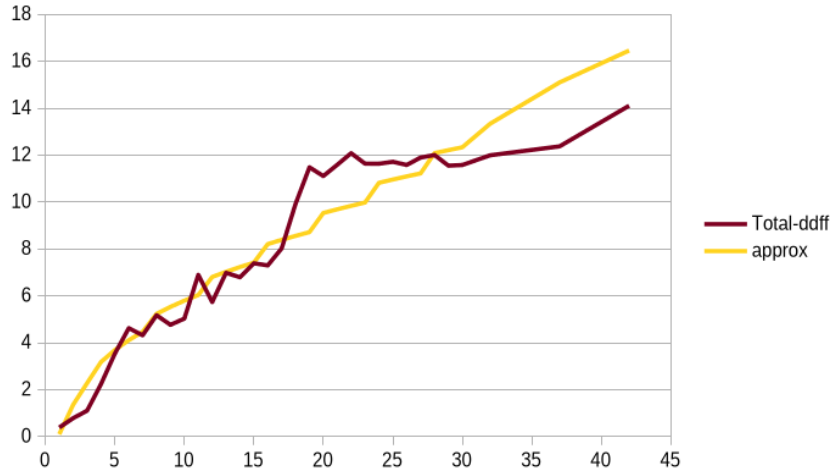
don't look too hard at the formulas

$$t_{\text{add}}(w) = (\lceil w/4 \rceil - 2) \cdot t_{carry4} + t_{base}$$

$$t_{\text{mul}}(w) = \frac{\lceil \log_2 w \rceil \cdot t_{\text{add}}(w)}{\sqrt{2}} + t_{\text{base}}$$

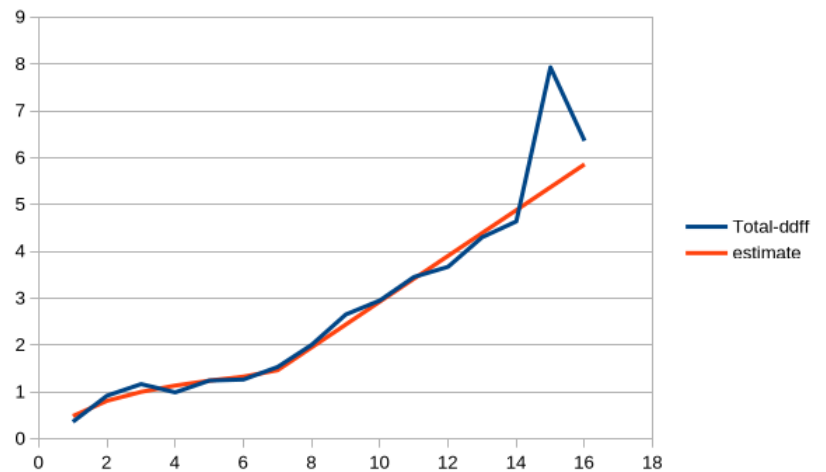


(a) Addition

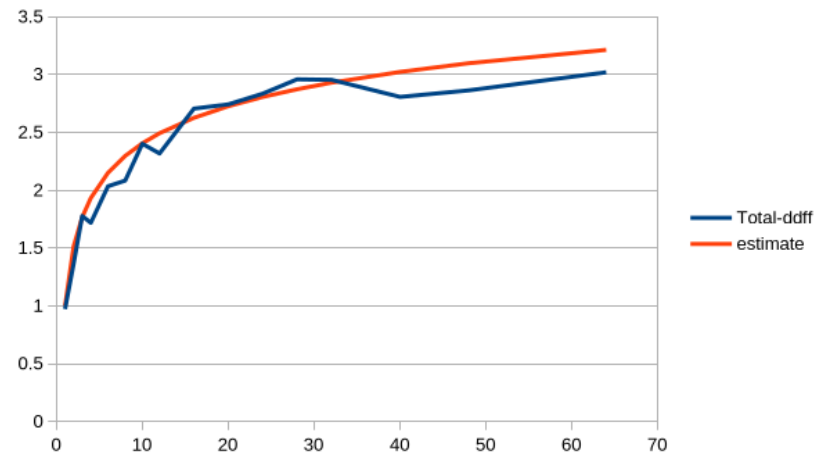


(b) Multiplication

$$t_{tab}(w_i, w_o) = \frac{\log_2(2 \cdot w_o) \cdot 6}{3} \cdot \max\left(\frac{1}{2} + \frac{\log_2(w_i)}{3}, \frac{w_i - 4}{2}\right)$$



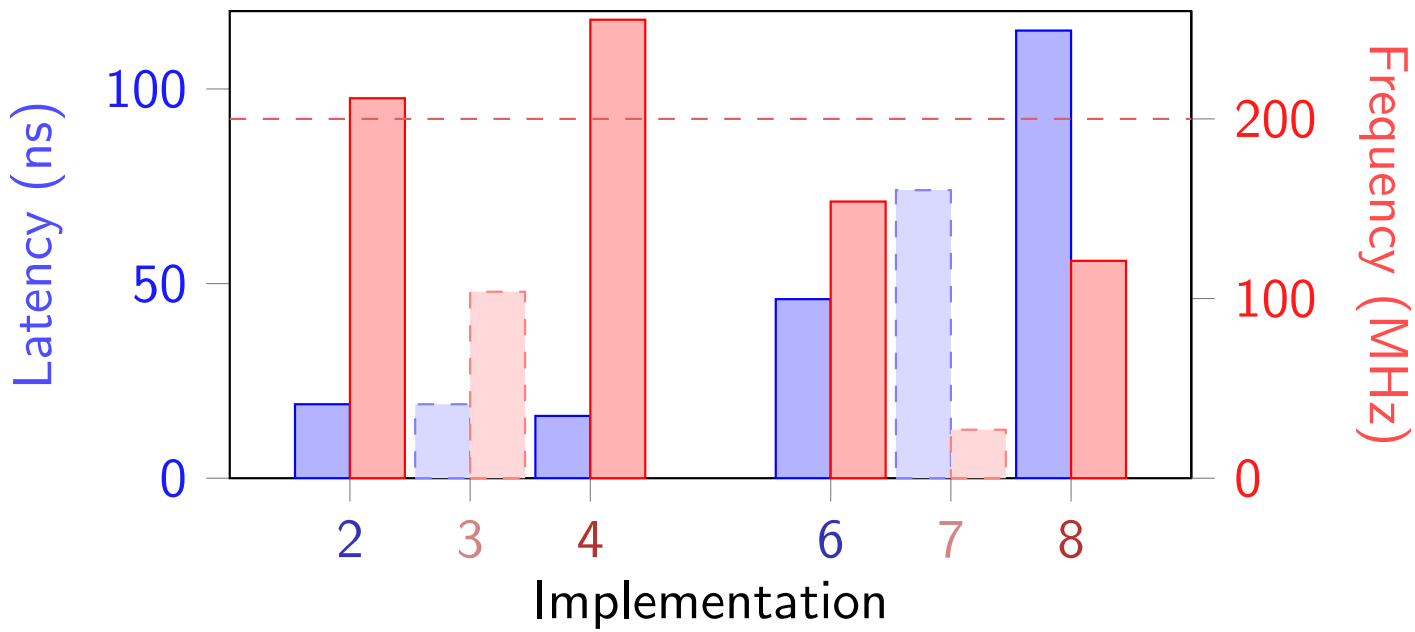
(c) Tables (fixed w_o at 32bits)



(d) Tables (fixed w_i at 10bits)

Figure: Delay models fitted to measurement data (ns against width)

Results

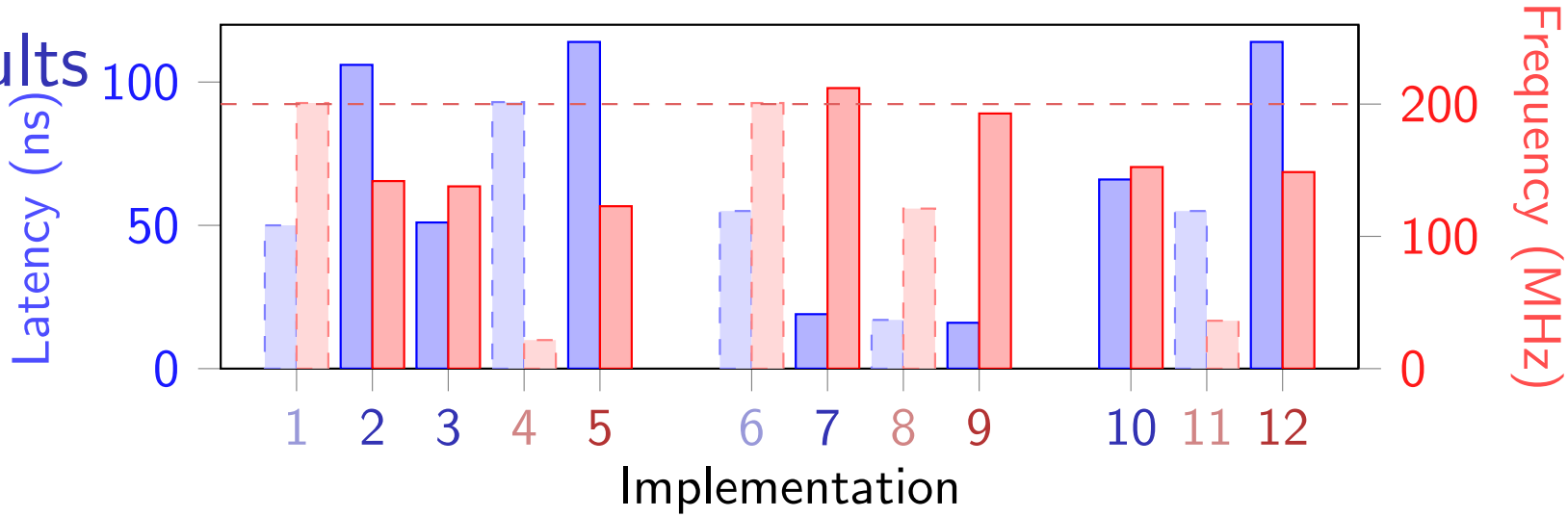


Implementation (For Zynq7000@200MHz)	**Bitwidth**	**LUT**	**FF**	**DSP**
Exponential Approximation in [0,4[and [0,16[				
Ours (w_{max}=8) with round half-up	16 bits	112	63	1
2 Ours (w_{max}=8) with BCR (**Same as above**)	16 bits	112	63	1
3 FloPoCo[a] Piecewise polynomial with d=1	16 bits	303	51	0
4 FloPoCo[b] Piecewise polynomial with d=1	16 bits	370	151	0
Ours (w_{max}=10) with round half-up	32 bits	1065	368	12
6 Ours (w_{max}=10) with BCR	32 bits	1001	329	8
7 FloPoCo[a] Piecewise polynomial with d=4	32 bits	465	79	6
8 FloPoCo[b] Piecewise polynomial with d=4	32 bits	2240	1156	4

[a] Using `plainVHDL=1`.

[b] Target frequency manually tweaked to be as close as possible to 200MHz.

Results

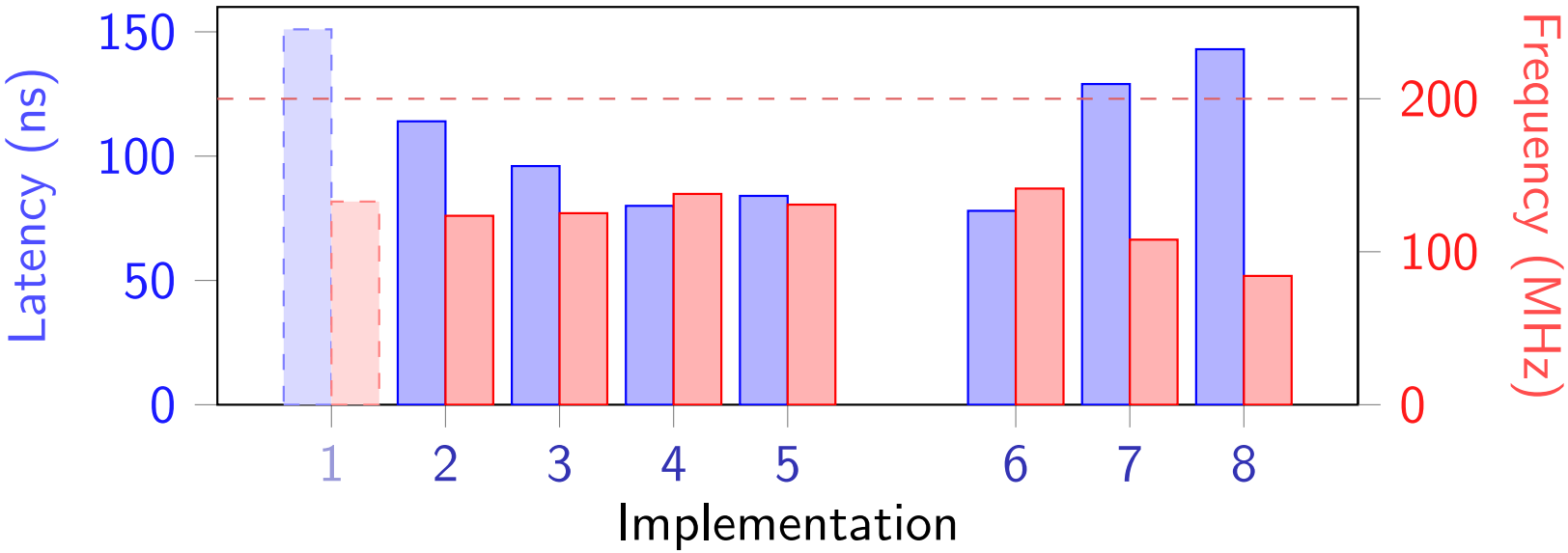


Implementation (For Zynq7000@200MHz)	**Bitwidth**	**LUT**	**FF**	**DSP**
Polynomial Approximation in [1,2[				
1 Ours with Horner's scheme in [1,1.5]	16 bits	158	229	4
2 Ours with Horner's scheme	16 bits	336	555	6
3 Ours with Estrin's scheme	16 bits	300	508	14
4 FloPoCo[a]	16 bits	46	17	7
5 FloPoCo[b]	16 bits	1268	818	3
Piece-wise Polynomial Approximation (with Horner's scheme) in [1,2[				
6 Ours using logarithmic splitting $w = 15$	16 bits	195	283	4
7 Ours using linear splitting $w = 10$	16 bits	370	102	1
8 FloPoCo[a] with degree 1	16 bits	125	33	0
9 FloPoCo[b] with degree 1	16 bits	186	66	0
10 Ours using linear splitting $w = 10$	32 bits	1322	610	5
11 FloPoCo[a] with degree 3	32 bits	299	66	6
12 FloPoCo[b] with degree 3	32 bits	1463	1109	2

[a] Using `plainVHDL=1`.

[b] Target frequency manually tweaked to be as close as possible to 200MHz.

Results



Implementation (For Zynq7000@200MHz)	**Bitwidth**	**LUT**	**FF**	**DSP**
Softmax in [-4,4[				
1 Ours with 4 inputs $w = 8$	23 bits	2540	2492	53
2 Ours with 4 inputs $w = 8$	16 bits	1627	894	19
3 Ours with 4 inputs $w = 10$	16 bits	2430	1003	15
4 Ours with 4 inputs $w = 9, 12$	16 bits	2357	789	13
5 Ours with 4 inputs $w = 12$	16 bits	6995	757	13
6 Ours with 4 inputs $w = 8$	8 bits	660	419	9
7 Ours with 8 inputs $w = 8$	8 bits	1234	961	17
8 Ours with 12 inputs $w = 8$	8 bits	1785	1303	26

Conclusion & Future Work

- Optimize base operators (adders,multipliers...).
- Add metrics and optimization targets for Ressource and Power use.
- Switch to better retiming and optimization algorithms.
- Add more efficient approximation methods (such as HOTBM).
- Support more number formats (floating-point, LNS...).
- Implement more activation functions.
- Test with AI models.

  ⋮

The code will also be shortly ready for a public release!

Any questions?

Thank you for your attention!